LEDA-SM:
External Memory Algorithms and Data Structures in Theory and Practice

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der naturwissenschaftlich-technischen Fakultät I
der Universität des Saarlandes

von

Andreas Crauser

Saarbrücken
2. März 2001

Datum des Kolloqiums: 2. März 2001

Dekan der naturwissenschaftlich-technischen Fakultät I:
Professor Dr. Rainer Schulze-Pillot-Ziemen

Gutachter:
Professor Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken
Professor Dr. Paolo Ferragina, Universita di Pisa, Pisa, Italien

Deutsche Kurzzusammenfasung

Die zu verarbeitenden Datenmengen sind in den letzten Jahren dramatisch gestiegen, so daß Externspeicher (in Form von Festplatten) eingesetzt wird, um die Datenmengen zu speichern. Algorithmen und Datenstrukturen, die den Externspeicher benutzen, haben andere algorithmische Anforderungen als eine Vielzahl der bekannten Algorithmen und Datenstrukturen, die für das RAM-Modell entwickelt wurden. Wir geben in dieser Arbeit erst einen Einblick in die Funktionsweise von Externspeicher anhand von Festplatten und erklären die wichtigsten theoretischen Modelle, die zur Analyse von Algorithmen benutzt werden. Weiterhin stellen wir ein neu entwickelte C++ Klassenbibliothek namens LEDA-SM vor. LEDA-SM ist bietet eine Sammlung von speziellen Externspeicher Algorithmen und Datenstrukturen. Im zweiten Teil entwickeln wir neue Externspeicher-Prioritätswarteschlangen und neue Externspeicher-Konstruktionsalgorithmen für Suffix Arrays. Unsere neuen Verfahren werden theoretisch analysiert, mit Hilfe von LEDA-SM implementiert und anschließend experimentell getestet.

Short Abstract

Data to be processed has dramatically increased during the last years. Nowadays, external memory (mostly hard disks) has to be used to store this massive data. Algorithms and data structures that work on external memory have different properties and specialties that distinguish them from algorithms and data structures, developed for the RAM model. In this thesis, we first explain the functionality of external memory, which is realized by disk drives. We then introduce the most important theoretical I/O models. In the main part, we present the C++ class library LEDA-SM. Library LEDA-SM is an extension of the LEDA library towards external memory computation and consists of a collection of algorithms and data structures that are designed to work efficiently in external memory. In the last two chapters, we present new external memory data structures for external memory priority queues and new external memory construction algorithms for suffix arrays. These new proposals are theoretically analyzed and experimentally tested. All proposals are implemented using the LEDA-SM library. Their efficiency is evaluated by performing a large number of experiments.

Deutsche Zusammenfassung

In den letzten Jahren werden immer größere Datenmengen maschinell verarbeitet. Die Größe der Daten bedingt, das sie im Externspeicher (realisiert durch Festplatten) gespeichert werden müssen und nicht in ihrer Gesamtheit in den Hauptspeicher heutiger Computer passen. Diese großen Datenmengen stellen besondere Anforderungen an Algorithmen und Datenstrukturen. Eine Vielzahl der existierenden Algorithmen und Datenstrukturen wurde für das theoretische RAM Modell [AHU74] entworfen. Zu den Kern-Eigenschaften des Modells gehört unbegrenzter Speicher; weiterhin kostet der Zugriff auf unterschiedliche Speicherzellen gleich viel (Einheitskostenmaß). Das RAM Modell wurde und wird heute benutzt, um Hauptspeicher-Algorithmen zu analysieren. Der Externspeicher hat jedoch andere Eigenschaften als der Hauptspeicher: ein Zugriff ist bis zu 100.000 mal langsamer als ein Hauptspeicher- oder Cache-Zugriff. Weiterhin liefert ein Zugriff immer einen Block von Daten zurück. Algorithmen, die im Externspeicher laufen, greifen somit auf zwei verschiedene Speicherhierarchien (Hauptspeicher und Externspeicher) zu, die unterschiedliche Zugriffszeiten und Eigenschaften haben. Die Annahme des Einheitskostenmaßes bei Speicherzugriffen ist somit fragwürdig. RAM Algorithmen und Datenstrukturen, die im Externspeicher laufen, haben meist eine sehr schlechte Laufzeit, da ihre Speicherzugriffe keine Lokalität aufweisen und man daher nicht den blockweisen Zugriff auf den Externspeicher optimal ausnutzen kann. Meist führen diese Algorithmen so viele Externspeicherzugriffe aus, das man kaum noch einen Fortschritt in der Berechnung sehen kann. Ausgehend von dieser Problematik wurden daher eigene Algorithmen und Datenstrukturen für Externspeicheranwendungen entwickelt.

Diese Arbeit beschäftigt sich mit Externspeicheranwendungen, sowohl im theoretischen als auch im praktischen Sinn. Im ersten Kapitel gebe ich einen Überblick über die Arbeit. Im zweiten Kapitel erläutere ich die Funktionalität des Externspeichers, der heute durch Festplatten realisiert wird. Weiterhin erkläre ich die Funktionalität von Dateisystemen am Beispiel des Solaris UFS Dateisystems. Anschließend stelle ich die wichtigsten Externspeichermodelle vor, genauer das I/O-Modell von Vitter und Shriver und die Erweiterung von Farach *et al.* In Vitter und Shriver's Modell besitzt ein Computer einen beschränkten Hauptspeicher der Größe M. Der Externspeicher ist durch D Festplatten realisiert; ein Externspeicherzugriff (kurz I/O) überträgt $D \cdot B$ Daten vom Externspeicher in den Hauptspeicher oder zurück, $1 \leq D \cdot B \leq M/2$. Farach *et al*'s Modell erlaubt es zusätzlich, I/Os in I/Os zu "zufälligen" Stellen (*engl.: random I/Os*) und in konsekutive I/Os (*engl. bulk I/Os*) zu klassifizieren. Bulk I/Os sind vorzuziehen, da sie in der Praxis schneller sind als random I/Os. Algorithmische Leistung wird in beiden Modellen gemessen, indem man (i) die Zahl der I/Os, (ii) die Zahl der CPU Instruktionen (mittels des RAM Modells) und (iii) den belegten Platz im Externspeicher ermittelt.

Im dritten Kapitel stelle ich eine von mir entwickelte C++ Klassenbibliothek namens `LEDA-SM` vor. LEDA-SM bietet spezielle Externspeicheralgorithmen und Datenstrukturen an und ermöglicht weiterhin durch Prototyping, schnell neue Algorithmen und Datenstrukturen zu entwickeln. LEDA-SM ist modular aufgebaut. Der sogenannte Kern der Bibliothek ist für den Zugriff und die Verwaltung des Externspeichers zuständig. Der Externspeicher kann durch Dateien des Dateisystems "simuliert" werden, er kann aber auch direkt durch Festplatten dargestellt werden. Wir realisieren explizit eine Abbildung von Vitter und Shriver's I/O Modell. Der Externspeicher wird in Blöcke eingeteilt, jede I/O überträgt ein Vielfaches der Blockgröße. Der Kern der Bibliothek stellt weiterhin Schnittstellen zur Verfügung, um einfach auf Festplattenblöcke zugreifen zu können und um einfach Daten blockweise zu lesen oder zu schreiben. Der Applikationsteil der Bibliothek besteht aus einer Sammlung von Algorithmen und Datenstrukturen, die speziell für Externspeicheranwendungen entwickelt wurden. Diese Algorithmen benutzen nur die C++ Klassen des Bibliothekskerns sowie Algorithmen und Datenstrukturen der C++ Bibliothek LEDA. Wir erläutern die wichtigsten Design– und Implementierungskonzepte von LEDA-SM, geben ein Implementierungsbeispiel für ein Externspeicherdatenstruktur und führen erste Leistungstests durch.

In den letzten beiden Kapiteln zeige ich anhand von zwei Fallstudien die Entwicklung und experimentelle Analyse von Datenstrukturen und Algorithmen für Externspeicheranwendungen. Die erste Fallstudie beschäftigt sich mit Prioritätswarteschlangen. Wir analysieren theoretisch – mittels der I/O-Modelle von Kapitel 2 – und experimentell – durch Implementierung mittels LEDA-SM – die Leistung von Prioritätswarteschlangen. Wir stellen weiterhin zwei neue Prioritätswarteschlangen für Externspeicheranwendungen vor. Unsere erste Variante, *R-Heaps*, ist eine Weiterentwicklung von Ahuja *et al.*'s redistributive heaps und unsere zweite Variante, *array heaps*, ist eine Weiterentwicklung einer Hauptspeichervariante von Thorup. Die erste Variante benötigt nicht negative, ganze Zahlen als Prioritätstyp und setzt weiterhin voraus, das die gelöschten Minima eine nicht fallende Sequenz bilden. Weiterhin müssen alle im Heap gespeicherten Elemente im Intervall $[a, \ldots, a + C]$ liegen, wobei a der Wert des zuletzt gelöschten Minimums ist (Null sonst). Diese Anforderungen werden z.B. von Dijkstra's kürzestem Wege Algorithmus erfüllt. Dieser Heap ist sehr schnell in der Praxis, platzoptimal im Externspeicher, jedoch nur suboptimal in Vitter und Shriver's D-Festplattenmodell. Er erlaubt Einfügen von neuen Elementen in $O(1/B)$ I/Os und Löschen des Minimums in $O((1/B) \log_{M/(B \log C)}(C))$ I/Os. Unsere zweite Variante basiert auf einer Menge von sortierten Feldern. Dieser Heap erreicht Optimalität im D-Festplattenmodell und ist weiterhin platzoptimal. Er unterliegt keinen Restriktionen hinsichtlich des Prioritätstyps, ist jedoch beschränkt in der maximalen Anzahl zu speichernder Elemente. Er unterstützt Einfügen in $18/B(\log_{cM/B}(N/B)$ I/Os und Minimum löschen in $7/B$ I/Os. Wir analysieren zwei Varianten des Heaps mit unterschiedlichen Eingaberestriktionen und zeigen, das die Größenrestriktion in der Praxis irrelevant ist. Diese neuen Datenstrukturen werden dann in einer Vielzahl von Experimenten gegen andere bekannte Externspeicher-Prioritätswarteschlangen wie buffer trees, B-trees, sowie gegen Hauptspeicher-Prioritätswarteschlangen, wie Fibonacci heaps, k–näre Heaps und andere, getestet.

Die zweite Fallstudie beschäftigt sich mit der Konstruktion von *Suffix Array*, einer Volltext-Indexdatenstruktur zur Textsuche. Suffix arrays sind eine von Manber und Myers eingeführte Indexdatenstruktur, die es ermöglicht, Volltextsuche durchzuführen. Suffix arrays spielen als Grunddatenstruktur eine wichtige Rolle, da andere Volltextindexstrukturen direkt aus Suffix Array aufgebaut werden können, wie z.B. SB-Trees von Ferragina und Grossi. Weiterhin sind Suffix Array platzeffezienter als andere Volltextindexstrukturen und besitzen eine einfache Struktur, ein Feld. Der Aufbaualgorithmus von Manber und Myers erweist sich im Externspeicher jedoch als nicht effektiv, da er keine Lokalität ausnutzt und unstrukturiert auf den Externspei-

cher zugreift. Wir analysieren zuerst mittels der I/O-Modelle von Vitter und Shriver sowie mittels des Modells von Farach *et al.* zwei bekannte Konstruktionsalgorithmen für Externspeicher bzgl. ihres Platzbedarfs und ihrer Externspeicherzugriffe, genauer Karp-Miller-Rosenberg's repeated doubling Verfahren und Baeza-Gonnet-Snider's Algorithmus. Ausgehend von diesen zwei Verfahren entwickeln wir drei neue Konstruktionsalgorithmen. Diese Algorithmen haben die gleiche I/O-Schranke wie Karp-Miller-Rosenberg's Algorithmus' $(O((N/B)\log_{M/B}(N/B)\log(N)))$, benötigen jedoch weniger Platz. Alle Konstruktionsalgorithmen werden von uns mittels LEDA-SM implementiert und auf verschiedenen Eingaben experimentell getestet. Wir schließen unsere Studie mit zwei weiteren theoretischen Betrachtungen. Zuerst zeigen wir, das alle Konstruktionsalgorithmen auch dazu benutzt werden können, um wortbasierte suffix arrays zu konstruieren. Abschließend verbessern wir die Anzahl der I/O Operationen von Baeza-Yates-Gonnet-Snider's Algorithmus. Obwohl der Platzbedarf von Baeza-Gonnet-Snider's Algortihmus gut ist, führt der Algorithmus im schlechtesten Fall eine Anzahl von Externspeicherzugriffen aus, die kubisch in der Größe der Eingabe ist. Wir zeigen, wie man dies auf quadratische Größe reduzieren kann.

Abstract

Data to be processed is getting larger and larger. Nowadays, it is necessary to store these huge amounts of data in external memory (mostly hard disks), as their size exceeds the internal, main memory of today's computers. These large amounts of data pose different requirements to algorithms and data structures. Many existing algorithms and data structures are developed for the RAM model [AHU74]. The central features of this model are that memory is infinitely large and that access to different memory cells is of unit cost. The RAM model has been and is still used to analyze algorithms that run in main memory. External memory, however, has different features than main memory: an access to external memory is up to 100,000 times slower than an access to main memory or cache memory. Furthermore, an access to external memory always delivers a block of data. Thus, external memory algorithms access two memory layers (main and external memory) that have different access times and features so that assuming unit cost memory access is questionable. As a result, most RAM algorithms behave very inefficient when transfered to the external memory setting. This comes from the fact that they normally do not rely on locality of reference when accessing their data and therefore cannot profit from blockwise access to external memory. As a consequence, special external memory algorithms and data structures were developed.

In this thesis, we develop external memory algorithms and data structures. Development consists of theoretical analysis as well as practical implementation.The first chapter is used to give an overview. In the second chapter, we explain the functionality of external memory that is realized by hard disks. We then explain the functionality of file systems at the example of Solaris' UFS file system. At the end of chapter two, we introduce the most popular external memory I/O models, which are Vitter and Shriver's I/O model and the extension of Farach *et al.* In Vitter and Shriver's model, a computer consists of a bounded internal memory of size M, external memory is realized by D independent disk drives. An access to external memory (shortly called I/O) transfers up to $D \cdot B$ items ($1 \leq D \cdot B \leq M/2$) to or from internal memory, B items from or to each disk at a time. Farach *et al.*'s model additionally allows to classify I/Os into (i) I/Os to random locations (*random I/Os*) and (ii) I/Os to consecutive locations (*bulk I/Os*). Bulk I/Os are faster than random I/Os due to caching and prefetching of modern disk drives. In both models, algorithmic performance is measured by counting (i) the number of executed I/Os, (ii) the number of CPU instructions (using the RAM model), and (iii) by counting the used disk space in external memory.

In the third chapter, we introduce our new C++ class library LEDA-SM. Library LEDA-SM offers a collection of external memory algorithms and data structures. By fast prototyping it is possible to quickly develop new external memory applications. Library LEDA-SM is designed in a modular way. The so called kernel is responsible for the realization and the access to external memory. In LEDA-SM, external memory is either realized by files of the file system or by hard disks themselves. We realize an explicit mapping of Vitter and Shriver's I/O model, *i.e.* external memory is divided into blocks of size B and each I/O transfers a multiple of the block size. The kernel furthermore offers interfaces that allow to access disk blocks and that allow to read or write blocks of data. The application part of the library consists of a collection of algorithms and data structures, which are developed to work in external memory. The implementation of these algorithms only uses C++ classes of LEDA-SM and of the

C++ internal memory class library `LEDA`. We first describe the main design and implementation concepts of LEDA-SM, we then show the implementation of an external memory data structure and give first performance results.

In the last two chapters we derive new external memory algorithms and data structures. The first case study covers external memory priority queues. We theoretically analyze (using the I/O models of Chapter 2) and experimentally compare (using LEDA-SM) state-of-the-art priority queues for internal and external memory. We furthermore propose two new external memory priority queues. Our first variant, *R-heaps*, is an extension of Ahuja *et al.*'s redistributive heaps towards secondary memory. It needs nonnegative integer priorities and furthermore assumes that the sequence of deleted minima is nondecreasing. Additionally, all elements, currently stored in the heap, must have priorities in an interval $[a, \ldots, C]$, where a is the priority value of the last deleted minimum (zero otherwise). This requirements are for examples fulfilled in Dijkstra's shortest path algorithm. This heap is very fast in practice, space optimal in the external memory model, but unfortunately only suboptimal in the multi disk setting of Vitter's and Shriver's D-disk model. Radix heaps support insert of a new element in $O((1/B)$ amortized I/Os and delete minimum in $O((1/B)\log_{M/(B\log C)}(C))$ amortized I/Os. Our second proposal, called *array heap*, is based on a sequence of sorted arrays. This variant reaches optimality in the D-disk I/O model and is also disk space optimal. There are no restrictions according to the priority type, but the size of the heap (number of stored elements) is restricted. Array heaps supports insert in $18/B(\log_{cM/B}(N/B)$ amortized I/Os and delete minimum in $7/B$ amortized I/Os. We analyze two variants with different size restrictions and show that the size restriction does not play an important role in practical applications. In the experimental setting, we compare our LEDA-SM implementation of both new approaches against other well known internal and external memory priority queues, such as Fibonacci heaps, k–ary heaps, buffer trees and B-trees.

Our second case study covers external memory construction algorithms for *suffix arrays*, a full text indexing data structure. Suffix arrays were introduced by Manber and Myers and allow to perform full text search on texts. Suffix arrays are an important base data structure as other important full text indexing data structures, for example SB-trees of Ferragina and Grossi, can directly built by using the suffix array. Additionally, suffix arrays are among the most space efficient full text indexing data structures in external memory. Unfortunately, the construction algorithm of Manber and Myers is not efficient if transfered to the external memory setting because it does not exploit locality of reference in its data structures. We analyze two well-known construction algorithms, namely Karp-Miller-Rosenberg's repeated doubling algorithm and Baeza-Yates-Snider's construction algorithm. We use the I/O model of Vitter and Shriver as well as the extension of Farach *et al.* to analyze the number of I/Os (including bulk and random I/Os) and the used space of different construction algorithms. We furthermore develop three new construction algorithms that all run in the same I/O bound as the repeated doubling algorithm ($O((N/B)\log_{M/B}(N/B))$ I/Os) but use less space. All construction algorithms are implemented using LEDA-SM and tested on real world data and artificial input. We conclude the case study by addressing two issues: we first show that all construction algorithms can be used to construct word indexes. Secondly, we improve the performance of Baeza-Yates-Snider's construction algorithm. In the worst case, this algorithm performs a cubic number of I/Os that is cubic in the size of the text. We show that this can be reduced to a quadratic number of I/Os.

Acknowledgments

When I look back at the last four years I remember a lot of people who influenced my work and my private life. First of all, I want to thank my advisor Prof. Dr. Kurt Mehlhorn. It was a real pleasure to work in his group and his excellent teaching and research style influenced me a lot. He introduced me to the topic of external memory algorithms and also encouraged me to start a programming project. Second, I want to thank my co-advisor Prof. Dr. Paolo Ferragina. The one and a half years, you stayed at MPI, were a very productive and pleasant time and I missed our conversations a lot when you went back to Pisa. It was not easy to keep in contact via email and still to write papers together but at least we managed it.

A lot of other people at MPI deserve special thanks. First of all I want to thank the whole algorithmics group. I enjoyed being your system administrator and you never bored me with your questions. My special thanks are to Ulli Meyer and Mark Ziegelmann for listening to ideas and problems of real and scientific life. Thanks to Klaus Brengel who programmed for the LEDA-SM project. I also want to thank the Rechnerbetriebsgruppe (Jörg, Wolfram, Bernd, Uwe, Thomas and Roland) for all the support they gave me and for the extraordinary working conditions they provide. It was a pleasure to work together with the RBG and I learned a lot about system administration and support.

Life is full of ups and downs! Thanks to Ulli Meyer, Mark Ziegelmann, and Michael Seel who celebrated my research successes with me and who also listened to my private problems. I hope that it is possible to give some help back to you and I hope that we will have enough time next year to go to skiing and to do some nice motorbike trips together.

A very special thanks to Claudia Winkler for being such a true friend. You always believed in me and helped me not to lose perspectives. However, I am by far most grateful to my family who gave me more love and encouragement that I can ever pay back!

Andreas Crauser
Saarbrücken, März 2001

Contents

Chapter 1

Introduction

When designing algorithms and data structures people normally have in mind minimizing the computation time as well as the working space for their solution. Theoretically, they design algorithms for the Random Access Machine model (RAM) [AHU74]. One of the main features of this model is the fact that the memory is infinitely large and access to different memory cells is of unit cost. Also in practice, most software is written for a RAM-like machine where it is assumed that there is a huge (nearly infinite) memory where individual memory cells can be accessed at unit cost.

Todays computer architectures consist of several memory layers where each layer has different features, sizes, and access times (see Figure 1.1). Closest to the processor (CPU) is the small cache memory that can often be accessed in one to two CPU clock cycles. The size of the caches can vary between a few kilobytes and several Megabytes [1]. The cache memory stores copies of the *main memory* and, thus, allows the CPU to access data or program code much faster. Main (*internal*) memory is up to a factor of one hundred slower than cache memory, but can reach sizes of several Gigabytes. Main memory is the central storage for program code and program data. The largest and slowest memory is the *secondary / external memory* that is today provided by hard disks. Thus, in contrast to all other memory layers, hard disks are mechanical devices while the other memory layers are built of electronic devices (DRAMs or SRAMs). Hard disks are up to a factor of one thousand slower in access time than main memory and each disk can store up to 50 Gigabytes, *i.e.* it is possible to provide Terabytes of storage in a large disk array.

In the early years of computer science, processors were quite slow, cache was often not existent and it was reasonable to focus the optimization on internal computation and to ignore the effects of the memory subsystem. In the last decades, processor speed has increased by 30% to 50% per year while the speed of the memory subsystem has only increased by 7% to 10% per year. Roughly speaking, today there is a factor of one hundred thousand to one million in the different access times to different memory layers so that assuming unit cost access time in algorithmic design is questionable.

Especially large-scale applications, as they can be found in geometric applications (geographic information systems), computational biology (DNA and amino acid databases), text indexing (web search engines), and many more must manipulate and

---

[1] One kilobyte equals 1024 bytes, one Megabyte equals 1024 kilobytes.

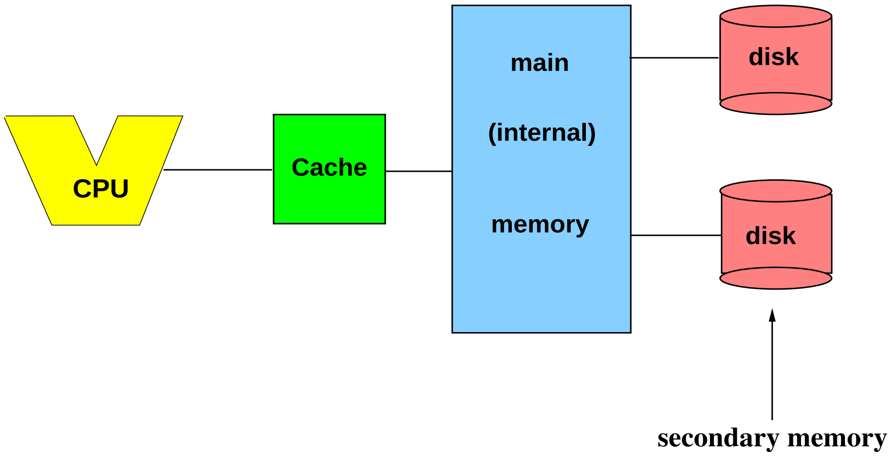


Figure 1.1: *Hierarchical memory in modern computer systems.*

work on data sets that are too large to fit into internal memory and that therefore reside in secondary memory. In these applications, communication between internal and secondary memory is important, as data has to be exchanged between the two memory layers. This data exchange is called *I/O operation* (short I/O); a single I/O operation always moves a block of data.

The time spent for the I/O operations is often the bottleneck in the computation, if the applications perform huge amounts of I/Os, or the application is not designed to work "well" in secondary memory. Today, applications often rely on the operating system to minimize the I/O bottleneck. Large working space is provided by means of *virtual memory*. Virtual memory is a combination of internal main memory and secondary memory and is provided by the operating system. To the programmer, virtual memory looks like a large contiguous piece of memory that can be accessed at unit cost. Technically, parts of that storage space reside on secondary memory (hard disks). The virtual (logical) address space is divided into *pages* of fixed size. Pages can reside in main memory or in secondary memory, if pages are accessed that are not in main memory, the operating system transports them to main memory (*page in*) eventually exchanging it with another page that must then be stored in secondary memory (*page out*). The main memory is handled as a pool of virtual pages (*page pool*) that are either occupied or free. Translating logical virtual page addresses to physical addresses in main memory, paging in and out, as well as running the page-replacement algorithm are the tasks of the virtual memory management system, which is part of the operating system and is mostly implemented in hardware.

By using caching and prefetching strategies, the operating system tries to assure that data, which is actually needed, is present in internal memory when it is accessed; thus trying to minimize the I/O bottleneck. However, these strategies are of general nature and cannot take full advantage of the specific algorithmic problem structure. Most of the algorithms, developed for the RAM-model, access the secondary memory in an unstructured way without exploiting any locality in the data. They spend most of their time in moving the data between external and internal memory, and thus, suffer from the I/O bottleneck.

To circumvent this drawback, the algorithmic community recently started to develop data structures and algorithms that explicitly take the I/O communication into account. These so-called *external or secondary memory algorithms* (shortly often called *external* algorithms) consider the memory to be divided into a limited size internal memory and a number of secondary memory devices. Internal and secondary memory have different access time and features. Main memory cells can be accessed at unit cost while an access to secondary memory is more expensive and always delivers a contiguous block of data. In the external memory model, algorithmic performance is measured by counting (i) the number of CPU operations (using the RAM-model), (ii) the number of secondary memory accesses, and (iii) by measuring the occupied working space in secondary memory.

In this thesis we study the complexity of external memory algorithms in theory and in an experimental setting where we write software for several external memory problems. These algorithms are then compared against algorithms that are solely designed for main memory usage (also called *internal memory or in-core algorithms*), and against other external memory algorithms for that problem.

1.1 Outline

Accurate modeling of secondary memory and disk drives is a complex task [RW94]. In the next chapter, we review the functionality and features of modern disk drives and file systems. We introduce simple engineering disk models and the theoretical secondary memory models and compare them to each other. One of the theoretical models is the standard *external memory model* introduced in [VS94b]. It uses the following parameters:

N = the number of elements in the input instance
M = the number of elements that fit into internal memory
B = the number of elements that fit into one disk block;

where $M < N$ and $1 \leq B \leq M/2$. We will compare this model to realistic engineering models and will later on introduce some modifications that are used in Chapter 4 and 5 to allow more realistic performance predictions. This chapter serves as a basic understanding in how to implement I/O operations and how to analyze secondary memory algorithms.

In Chapter 3 we turn to implementing secondary memory algorithms. Secondary memory algorithms move data in the memory hierarchy (from disk to main memory and vice versa) and process data in main memory. A platform for secondary memory computation therefore has to address two issues: performing I/O operations and co-operation with internal memory algorithms. We propose `LEDA-SM` as a platform for secondary memory computation. LEDA-SM is a C++ class library and extends LEDA [MN95, MN99] to secondary memory computation; therefore, LEDA-SM is directly connected to an efficient internal memory library of data structures and algorithms. LEDA-SM is portable, efficient, and easy to use. The library is divided into a kernel layer that manages secondary memory and the access to it, and into an application layer which is a collection of secondary memory algorithms and data structures. These algorithms and data structures are based on the kernel; however, their usage requires little knowledge of the kernel. LEDA-SM together with LEDA supports fast prototyping of secondary memory algorithms and can therefore be used to experimentally analyze new data structures and algorithms in secondary memory. In Chapter 3, we describe the implementation of LEDA-SM, both for the kernel and for the applications. We also give an overview of implemented data structures and algorithms. At the end of Chapter 3, we experimentally analyze the core library I/O performance as well as the performance of basic algorithms like sorting.

Experimental analysis is often a problem, as it is not easy to find good and interesting input problems. Especially for secondary memory computation it turns out to be hard, as we have to provide very large reasonable inputs. In Chapters 4 and 5, we perform two case studies for secondary memory algorithms and data structures. In these chapters, algorithmic performance is studied theoretically (by using some of the models of Chapter 2) and experimentally by implementing the algorithms and data structures using LEDA-SM.

In Chapter 4 we study the behavior of priority queues in secondary memory. A priority queue is a data structure that stores a set of items, each consisting of a tuple which contains some *(satellite) information* plus a *priority* value (also called *key*) drawn from a totally ordered universe. A priority queue supports (at least) the following operations: `access_minimum` (returns the item in the set having minimum key), `delete_minimum` (returns and deletes the item having minimum key) and `insert` (inserts a new item to the set). Priority queues have numerous applications: combinatorial optimization (*e.g.* Dijkstra's shortest path [Dij59]), time forward processing [CGG$^+$95], job scheduling, event simulation, and online sorting, just to cite a few. Many priority queue implementations exist for small data sets fitting into internal memory, *e.g* k-nary heaps, Fibonacci heaps, and radix heaps and most of them are publicly available in the LEDA library. However, in large-scale event simulation or on instances of very large graphs, the performance of these internal-memory priority queues significantly deteriorates, thus being the bottleneck of the overall application. In Chapter 4, we study the theoretical and experimental performance of priority queues. Some of them are internal memory priority queues running in virtual memory (k-nary heaps, Fibonacci heaps, and radix heaps) while four others are secondary memory data structures (B-trees [BM72], buffer-trees [Arg95], external radix heaps [BCMF99] and array-heaps [BCMF99]). The first two secondary memory data structures are actually search tree variants while the last two approaches are our new proposals. All data structures are theoretically analyzed using the theoretical I/O models from Chapter 2. B-trees support `insert` and `delete_min` in $O(\log_B(N))$ I/Os, Buffer-trees support them in $O((1/B)\log_{M/B}(N/B))$ I/Os, radix heaps support `insert` in $O(1/B)$ I/Os and `delete_min` in $O((1/B)\log_{\frac{M}{B\log C}}(C))$ I/Os and array heaps support `insert` in $O((1/B)\log_{M/B}(N/B))$ I/Os and `delete_min` in $O(1/B)$ I/Os. In the analysis of Chapter 4, we will pose particular attention to differentiate between disk accesses to random locations and to consecutive locations, in order to reasonably explain some practical I/O-phenomena, which are related to the experimental behavior of these algorithms and to current disk drive characteristics.

All priority queues will then be experimentally analyzed using artificial input and real word data. We perform tests to determine the core speed of operations `insert` and `delete_min`, thus testing the I/O performance, as well as tests with intermixed sequences of `inserts` and `delete_mins`, thus testing the speed of the internally used structures of the priority queues. In the end, we come up with a hierarchy of secondary memory priority queue data structures according to running time, I/O behavior, and working space.

In Chapter 5, we investigate the construction of suffix arrays in external memory. Suffix arrays were introduced by Manber and Myers [MM93] and are a static data structure for full-text indexing. A full text index data structure is a search data structure, built on every text position, allowing to search for arbitrary character sequences. We review some well-known construction algorithms for suffix arrays, which are Manber-Myers' construction [MM93], BaezaYates-Gonnet-Snider's construction [GHGS92]
(BGS), and construction by repeated doubling [RMKR72]. Manber-Myers' algorithm is an internal-memory algorithm running in $O(N \log N)$ I/Os (where N is the size of the text in characters), BGS runs in $O((N^3 \log M)/(MB))$ I/Os and doubling runs in $O((N/B) \log_{M/B}(N/B)\ \log N)$ I/Os. We furthermore introduce three new construction algorithms that all run in the same I/O-bound as the doubling approach, but that perform substantially better by reducing internal computation, working space and I/Os. All algorithms are theoretically analyzed using the theoretical I/O models from Chapter 2. All construction algorithms will then be experimentally tested using real word data, consisting of natural English text, amino acid sequences, and random text of varying alphabet sizes. All these text inputs have different features and characteristics that affect the efficiency of the different construction methods. In the end we give a precise hierarchy of the different construction approaches according to working space, construction time, and I/O costs. We conclude that chapter by addressing two issues. The former concerns the problem of building word-indexes where, in contrast to full-text indexes, only words are indexed. We show that our results can be successfully applied to this case too, without any loss in efficiency and without compromising the simplicity of programming so to achieve a uniform, simple and efficient approach to both the indexing models. The latter issue is related to the intriguing and apparently counterintuitive "contradiction" between the effective practical performance of the construction by BaezaYates-Gonnet-Snider's algorithm, verified in our experiments, and its unappealing (*i.e.* cubic) worst-case behavior. We devise a new external memory algorithm that follows the basic philosophies underlying BaezaYates-Gonnet-Snider's construction but in a significantly different manner, thus resulting in a novel approach (running in $O(N^2/(MB))$ I/Os), which combines good worst-case bounds with efficient practical performance.

Chapter 2

Disks, File Systems and Secondary Memory

Magnetic disk drives are todays media of choice to store massive data in a permanent manner. Disk drives contain a *mechanism* and a *controller*. The mechanism is made up of the recording components and the positioning components. The disk controller contains a microprocessor, some buffer memory, and an interface to some bus system. The controller manages the storage and retrieval of data to and from the mechanism and performs the mapping between logical addresses and physical disk sectors. that store the information. The mechanism consists of the components that are necessary to physically store the data on the disk. The process of storing or retrieving data from a disk is called *I/O operation* (short: I/O).

2.1 Disk Drives

A disk drive consists of several *platters* that rotate on a *spindle*. The surface of each platter is organized like coordinates; data is stored in concentric *tracks* on the surfaces of each platter. Each track is divided into units of a fixed size (typically 512 bytes), these are the so called *sectors*.

Sectors are the smallest individually addressable units of the disk mechanism. I/O operations always transfer data in multiples of the sector size. A *cylinder* describes the group of all tracks located at a specific position across all platters. A typical disk drive contains up to 20 platters and approx. 2000–3000 tracks per inch. The number of sectors per track is not fixed; today more sectors are packed onto the outer tracks than onto the inner tracks. This technique is called *multiple zone recording*. For a 3.5 inch disk, the number of sectors per track can range from 60 to 120 under multiple zone recording. This technique increases the total storage capacity by 25 percent and data can be read faster (up to a factor of two) from the outer zones of the disk.

2.1.1 Read/Write Heads and Head Positioning

Each platter surface has an associated *disk head* responsible for recording (writing) or sensing (reading) the magnetic flux variations on the platter's surface. The heads are

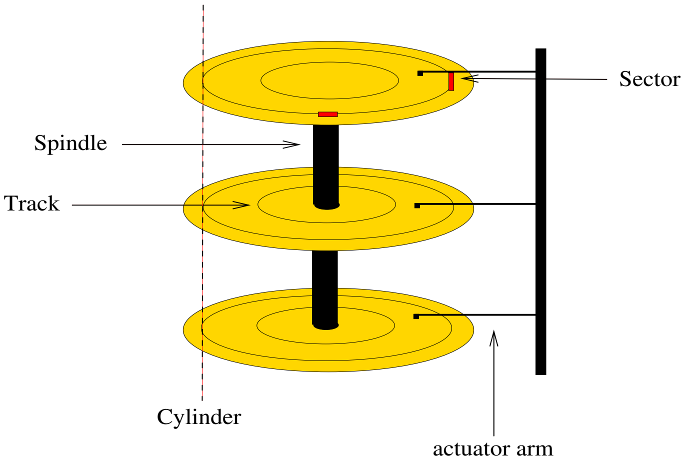


Figure 2.1: *A small disk drive example*

mounted on the *actuator arm* that is moved by the disk drive's motor. All heads move synchronously. The disk drive has a single *read/write channel* that can be switched between the heads. This channel is responsible for encoding and decoding the digital data stream into or from an analog series of magnetic phase changes stored on the disk. Multichannel disks can support more than one read/write channel at a time thus making higher data transfer rates possible. However these disks are relatively costly because of technical difficulties such as controlling the cross talk between concurrently active channels and keeping multiple heads aligned on their platters simultaneously. A lot of effort was done in the last decades to improve disk heads and read/write channel efficiency. Different disk head materials where used to improve the disk head performance leading to todays *magnetoresistive* heads that use different materials for read and write elements. Together with new channel technologies (*e.g.* PRML channels) this allows to pack data more closer on the disk surface, thus increasing areal density and data throughput.

The most tricky business in disk mechanics is *head positioning* also known as *track following*. Todays disk drives rotate at speeds between 5400 to 10900 rotations per minute (rpm). The head is flying 7 microinches above the platter surface and tracks are 300 microinches apart from each other. Besides the technical difficulties introduced by small flying height and close track-to-track distance, a number of variables is working against accurate head positioning. These include temperature variations as well as shock and vibration. To counter these effects, disk drives use an electro-mechanical technique called servo positioning, which provides feedback to the drive electronics to control the position of the head.

2.1.2 Optimizations in Disk Geometry

Disk drives are optimized for sequential access. When larger amounts of data are read or written, it is likely that head switches to the next platter or even track switches occur.

During a head switch or track switch the disk keeps on rotating so that if all starting sectors (sector zero) lay on the same position, we have to wait a full revolution of the disk before the read or write can continue. To avoid this, the starting sector of the next platter or track is skewed by the amount of time required to cope with the worst case head- or track switch times. As a result, data can be read or written at nearly full media speed. Each zone of the disk has its own *cylinder* or *track skew* factors.

2.1.3 Disk Controller and Bus Interface

The disk controller consists of a microprocessor and an embedded cache. The disk controller is responsible for controlling the actuator motor (thus the head movement), it runs the track following system (thus head positioning), it decodes and encodes the data in the read/write channel, it manages the embedded cache and transfers data between the disk and its client. Interpreting disk requests takes some time, this time is called *controller overhead* and is typically in the range of 0.3 to 1 millisecond (shortly ms).

The disk drive is connected to its client via a *bus interface*. The most common bus interface is *SCSI* (Small Computer System Interface) which is currently defined as a bus. Todays modern hard disks can drive the SCSI bus at a speed of 20 MBps (Megabytes per second) and the standard is defined up to 40 MBps. Because SCSI is a bus, more than one device can be attached to it. This can lead to bus contention. To avoid this, most disks use the *disconnect-reconnect* scheme. The operating system requests the bus and sends the SCSI command. After an acknowledge by the disk, the disk disconnects from the bus and reconnects later if it has data to transfer. This cycle may take 200 microseconds but allows other devices to access the bus. The limitations of the SCSI bus are the number of drives that can be connected to it and the effective cable length. These are currently 16 drives and an effective bus cable length of 25/12 meters (fast/ultra wide, differential SCSI interface). With todays high-speed drives that can reach transfer rates of up to 20 MBps, the bus speed of SCSI is a limitation when using multiple drives connected to one bus.

State-of-the art technology uses optical fiber interconnect (known as *Fiber Channel SCSI*) to carry the SCSI protocol. It runs at 25 MBps in each direction and allows cable length for 1000 meters and more. The latest type of fiber channel is full speed at 100 MBps in both directions.

2.1.4 Caching in Disk Drives

As most disk drives take data off the media more slowly than they can send it over the bus, the disk drives use speed-matching buffers (or read caches) to hide the speed difference. During read access, the drive partially fills its read cache before attempting to start the bus data transfer. The amount of data that is read into the cache before the transfer is initiated is called *fence*. By using data prefetching during read access (*read ahead*), read caching tries to take advantage of the fact that most accesses are sequential. Should the data be requested by a subsequent command, the transfer takes microseconds instead of milliseconds as the requested data already resides in the cache. During write accesses, write caching (*write behind*) allows host–to–disk-buffer and disk-buffer–to–disk transfers to occur in parallel. This eliminates rotational latency during sequential accesses and allows to overlap seek time and rotational latency with system processing during random write accesses.

When write caching is enabled, for the user's view the write request has finished as soon as the data has arrived in the disk cache. This is only failsafe, if the disk is battery-buffered so that the cache content can be written to the surface in case of a power failure. Caching has a lot of effect for small request sizes, for large read or write operations, the cache is bypassed [Bos99].

2.1.5 Measuring Hard Disk Performance

A lot of performance specifications are given by the manufacturer. The most common are seek time and data transfer rate. This section will explain the most common disk specifications.

- Seek Time
  The amount of time it takes the actuator arm to move the head to a specific track is called *seek time*. Seek time is limited by the power available for the pivot motor and by the actuator arm's stiffness. Accelerations between 30 to 40g are needed to achieve good seek times. A seek is composed of a *speedup* where the arm is accelerated, a *coast* (for long seeks) where the arm moves with maximum velocity, a *slowdown* where the head is brought close to the desired track and a *settle* where the disk controller adjusts the head position. The seek time is not linear in the distance. Very short seeks are dominated by the settle time, short seeks spend most time in the acceleration phase plus the settle phase (their time is proportional to the square root of the distance plus the settle time) and long distance seeks spend most time in moving the head at constant speed. Only long distance seeks are proportional to the seek distance. The most common specification is the average seek time for a request.

- Rotational Latency
  Once the head is positioned over the desired track, it has to wait until the desired sector arrives. This time is called *rotational latency* (rotational delay) and is measured in milliseconds. The rotational latency is inverse proportional to the drive's rotations per minute. On average the drive has to wait half a rotation (the desired locations are drawn uniformly and independent from each other). Typical rotational delays are 5.7 ms (5400 rpms) and 4.2 ms (7200 rpms).

- Head Switch Time
  The actuator arm moves all heads synchronously. However, only one head is active at a time. The time needed to switch between two different heads is called *head switch time* and is measured in ms.

- Cylinder or Track Switch Time
  A Cylinder switch or track switch requires to move the actuator arm to the next track. *Cylinder switch time* measures the average time it takes to switch to the next cylinder when reading or writing data. This time is measured in ms.

- Data Access Time
  One of the most common measurements is average data access time. It mea-

sures the average time it takes to read or write a sector of interest. Therefore data access time is a combination of seek time, head switch time and rotational latency.

- Data Transfer Rate
  The data transfer rate measures how much data the disk can transfer to the host in a given time interval. It depends on two measures: the disk transfer rate, or how fast data is transfered from the surface of the platter to the disk controller (i.e. to the disk cache), and the host transfer rate, or how fast data can be transfered via the bus by the disk controller. Data transfer rate is measured in Megabytes per second (MBps).

The two most important measurements are data access time and data transfer rate as they are used to calculate the amount of data that can be written or read from the disk. In real situations, a lot of additional effects play an important role for the overall performance. Internal memory performance, I/O bus speed, and the number of disk drives connected to the bus are also important as they affect system performance and as each of them can be the bottleneck for the overall performance. We refer to [Bos99] who looks in more detail at the overall I/O system and effects like the number of disk drives connected to a single I/O bus.

2.2 File Systems

File systems are part of the operating system and manage disk space in a user-friendly way. A file itself is a collection of disk blocks (sectors). The file system provides file access and file creation/deletion methods to the programmer and manages file storage as well as a file integrity mechanism. Files can be referred by their symbolic names. Additional functionality is often provided such as backup and recovery utilities for damaged files or encryption and decryption of files. The file system also provides user-friendly interface that allows the programmer to access the file system from a logical view (access to files) rather than to care about the physical layout (attached disk devices and placement of data). We now describe in more detail UNIX-like file systems.

2.2.1 UNIX-like File Systems

Unix file systems are *hierarchical* file systems. The file system consists of *directories* and *files* organized in a hierarchical tree structure with the *root directory* as the root of the tree and the files as the leaves. Directories are a special kind of files that allow to logically group files together. Files and directories are referenced by their names (character strings). A Unix file system is data structure resident on the disk; it contains a *superblock* that defines the file system, an array of *inodes* that define the files/directories, and the actual file data blocks plus a collection of free blocks. As the superblock is critical data, it is replicated on the file system to prevent catastrophic loss. The file system is a collection of fixed size blocks. This *logical disk blocks* are further divided into *fragments*. Each logical disk block can consist of 2, 4 or 8 fragments, the number of fragments per logical disk block is fixed at the time of file system creation. Fragments are the smallest addressable and allocatable units in a file system, a fragment must be at least as large as a disk sector (normally 512 bytes). The file system records space availability at the fragment level. The logical disk block size is fixed for a file system and normally varies between 4 kbytes to 8 kbytes although larger values are possible. Inodes contain the information to locate all of a file's physical disk blocks. The information of an inode points directly to the first blocks of the file and then uses various levels of indirection to point to the remaining blocks (see [Dei90] for a detailed description). Thus the inode structure is fast on small files (which was typical in Unix environments about 10 years ago) and is slower on larger files. Today, large files can be supported more efficiently by decreasing the inode density during file system creation (thus increasing the fixed block size) [Won97]. Standard disk block and fragment values for the Solaris UFS file system are a logical disk block size of 8 kbytes and 8 fragments per logical block. Thus the smallest addressable size of a file is 1 kbytes; this is also the minimal size of a file. Another important method is *clustering*. At the time of file system creation, one can specify the maximum number of blocks belonging to one file, that will be allocated contiguously before inserting a rotational delay on the disk. This allows to achieve higher I/O throughput during sequential access as one can read or write larger consecutive portions. The standard cluster size value is 56 kbytes, this value can be increased by file system tuning commands.

File system I/O is handled via a special operating system cache (buffer cache or file system cache). All data is first transfered to this cache and then to the disk (file write) or to the user space (file read). File system requests are always multiples of the fragment size. When file I/O is performed by the user process, the process blocks (stops running) to some extent. Writes are normally asynchronous, the user process blocks until the data is placed into the caches. Cache data is synchronized with the disk by a special daemon process *flush* that writes back meta data (inodes) and user data to the disk after a specified time interval. Reads are synchronous as the program has to wait until the data is actually read into main memory. Handling writes in an asynchronous way allows to optimize the access pattern to the disk as writes are more time consuming, however, if a disk failure occurs, it can happen that some of the file data or meta data were not correctly written to the disk leaving the file system damaged.

2.2.2 Caching in File Systems

A few years ago, Unix operating systems used the "buffer cache" to cache all disk data, assigning about 10% of the available main memory. Todays more advanced Unix operating Systems (see e.g. Solaris-2 [CP98, Won97]) integrate the buffer cache into the virtual memory system of the machine thus being more flexible with the cache size. Hence, the cache can be as big as the total available memory, the operating system only ensures that at least a minimum number of memory pages (specified by system constant *min_free*) are always available.

Today, separate caches are used for file system caching. Reading and writing disk data is directly handled via the virtual memory. Therefore the page pool can consist of program data pages, program text pages (the code of the program) and file data pages. Data that should be written to the file system resides cached in main memory thus possibly saving read requests to the same page afterwards. As file operations are solely handled by the "buffer cache", the cache data structure is responsible to logically map memory pages to disk block pages or fragments (as their size can be different).

As most of todays machines have large main memories, the cache for disk I/O is thus much larger than the originally used buffer cache. I/O intensive applications can benefit a lot from this feature. File system meta data (as for example inodes) is placed in different caches. If a file was read in its whole, it is likely that the inode data, necessary to locate the physical disk blocks of the file must not be reread again from the disk itself and still resides in the cache. As all I/Os transfer at least a fragment of a logical disk block, data alignment problems can cause multiple I/Os:

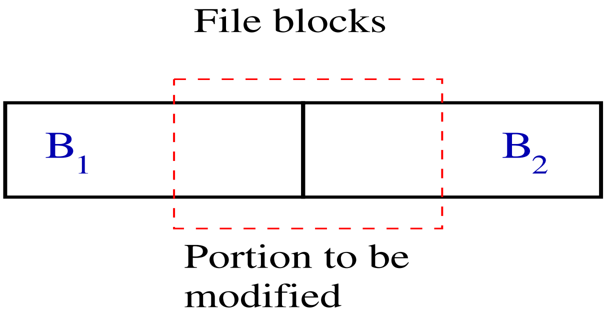


Figure 2.2: *Data alignment problem using file I/O*

The portion to be modified is not fully aligned to the underlying pages/fragments. Therefore, if we just issue one write command to change this portion, the file system first reads B_1, modifies and writes B_1, then reads B_2, modifies it and writes it back. Would the data be aligned to pages/fragments, a single write would be enough!

2.2.3 File Access Methods

File access methods are functions (provided by the operating system) that work on the file system. There exist several classes of file access methods. As a standard, each Unix operating system provides a number of standardized *system calls for file I/O*. These system calls are *open(), read(), write(), lseek(),* and *close()*. System call `open()` opens an existing file on the file system, it can also create new files. It returns a *file descriptor* which is a logical identification of the opened file. All later operations on the opened file use the file descriptor to refer to that file. Files can be opened with different privileges: read-only, write-only, read-write, and append-at-end. This is specified by parameter `oflag`. There exist a lot of other additional parameters like specifying synchronized write and so on. For a list of full parameters we refer to the UNIX man pages (man open(2)). System call `lseek()` allows to seek to a specific position in the file by repositioning the so called *seek pointer*. Parameter `offset` specifies the offset in bytes to seek from either the current position, beginning of file or end of file (this is specified by parameter `whence`). System calls `read` and `write` allow to read or write a specified amount of bytes starting from the position of the seek pointer. Parameter `buf` points to the data to be written or to the memory region where the data should be read into, and `nbyte` specifies the size of the data in bytes to be read or written.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>

int open(const char *pathname, int oflag, ..., /*, mode_t mode */);
```

```
#include<unistd.h>

ssize_t read(int filedes, void *buf, size_t nbyte);

ssize_t write(int filedes, const void *buf, size_t nbyte);
```

```
#include<sys/types.h>
#include<unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

The UNIX standard application interface (API) defines that during file system I/O, data is exchanged from an user-specified I/O buffer to a system I/O buffer and then to the disk or vice versa. Significant amount of time can be spent in copying between the buffers.

Memory-mapped I/O establishes a mapping between the file's disk space and the virtual memory. Instead of performing read and write operations on a file descriptor, the file system places the file in the virtual memory system. The virtual memory can then be addressed and manipulated via pointers and the effects also occur in the file. The operating system is not involved in managing I/O buffers, instead it only manages virtual memory. This saves costly buffer-to-buffer copy routines.

```
#include<sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int filedes, off_t off);
```

Before establishing the mapping, the file must be opened using `open()`. The call `mmap()` returns a pointer to a memory region *pa*. This is a mapping of region [`off`, `off+len`] of the file, specified by file descriptor `filedes` to memory region [*pa*, *pa*+`len`]. Parameter `addr` is an implementation dependent parameter that influences the computation of *pa*, parameter `prot` specifies how the mapped memory region can be accessed (read/write/execute/no access) and parameter `flags` provides additional information about the handling of the mapped pages (see also man mmap(2)).

Standard I/O (short *stdio*) is a set of I/O routines provided by the standard I/O library (see also man stdio(3s)). Stdio was originally introduced to handle character-based I/O that occurs when reading from a character-based device like a keyboard. Instead of reading single characters and thus issuing one system call per character, stdio uses a buffer in the user-space to reduce the number of read and write system calls. Instead of handling file descriptors, when opening or creating files we associate a stream with the file. All actual operations are performed on that stream. The stream is buffered, the kind of buffering is dependent on the type of the stream. Data is written when the buffer is full. When data is read, we always read until the buffer is full. Stdio provides similar functions to system-call I/O, now called *fopen(), fread(), fwrite(), fseek()* and *fclose()*. Stdio uses system call I/O to actually read or write the data, but it always reads or writes the whole buffer content. The size of the buffer is normally 1024 bytes but can be increased or decreased.

Some UNIX systems also provide the feature to access the raw disk device. Using the standard file system calls open(), read(), write(), lseek() and close(), they allow to open the disk device specific file and perform read, write and seek operations directly on the disk device. This functionality mainly bypasses the file system layer and buffer caches.

2.2.4 File systems versus raw device

File systems and raw disk access offer different kinds of advantages and disadvantages. File systems provide an easy way to handle access to disks without actually caring about the physical data placement on the disk. Caching allows to speed-up disk requests but can also slow down the system. Write performance is increased as we immediately return to computation as soon as the data resides in the cache. Caching however can also slow down the efficiency as the data resides in the page pool as long as no other process needs main memory. The system can run into a memory shortage and the help of the page daemon is needed to free main memory. This increases the percentage that the CPU spends in running operating system processes and reduces the percentage that the CPU spends for the user request. Another disadvantage is that the file is only logically consecutive and the disk data is interspersed with the inode information. However, on nearly empty file systems one can rely on getting almost consecutive physical disk blocks for newly created files. The main advantage of using file systems is that programs, written to use file access methods, are portable across several platforms. File systems are still efficient and are not necessarily slow. Careful tuning of the operating system allows to increase the read and write efficiency [CP98]. Unfortunately, the tuning process is not easy and needs a lot of operating system knowledge.

Raw disk device access is not portable across several platforms. For example, Solaris platforms add the functionality of accessing the raw device via system calls, Linux platforms however do not add this functionality. Raw disk device access bypasses all the caches, therefore it can be faster but it may not be. The main advantage of raw disk device access is that consecutively allocated disk blocks are now really consecutive and there is no interspersed meta data such as inodes. However, allocation of disk blocks and calculation of addresses is now up to the user. It is not necessarily true that raw device access is faster than file system access. Obviously, file systems “waste” internal memory by using the buffer cache. This is not true for raw devices as the data is directly transfered to the disk.

2.2.5 Solaris File Systems

Most of our implementation work, which I will describe in the next chapter, was performed on SUN workstations running Solaris. Solaris file systems are different from standard Unix file systems (as they are, for example, described in [Dei90]). Workstations running Solaris use the whole virtual memory system to represent the buffer cache. The standard file system is the so-called *UFS file system*. It uses either 4 or 8 kbytes as logical file block size and 8 fragments per block. Thus, the smallest addressable unit is either 512 bytes or 1 kbytes, which is also the smallest possible file size. As I/O data is buffered in main memory and as it is possible to use the whole main memory for buffering, the system can run out of free pages in main memory due to massive I/O. To circumvent that problem, a specific operating system process (*page daemon*) wakes up as soon as the number of free pages in the page pool drops below `min_free` (a system constant). This process tries to free pages by stealing pages from other processes. In normal mode, it is able to free 64 Mbytes/sec [1]. This slows down the system as another process is competing for CPU resources. The work of the daemon can be monitored by using the Solaris monitoring tool `vmstat`. It reports a value `sr` which is the number of stolen pages per second. If this value stays above 200 for a long time, this system is out of memory [CP98]. To circumvent this problem, Solaris file systems are able to use the raw device on which the file system is located. This feature called *direct io*. For larger I/O requests (larger than the standard file block size which is either 4 or 8 kbytes), the file system does not use the buffer cache. Instead it uses a direct path to the disk device which is quite similar to a raw device access. This increases the performance for large sequential accesses and does not waste internal memory.

Additionally, Solaris uses a lot of implementation tricks in the different file access to increase their performance. First, it avoids the one copy process for system call I/O. The Unix API specifies that the data is first copied from a user buffer to an operating system buffer and then to disk. Solaris uses techniques from memory mapping to avoid the first copy step. Second, it uses 8 kbytes of buffer for stdio instead of 1 kbytes. As a consequence, the buffer perfectly aligns to the size of a virtual memory page (and also to the size of a logical file block).

Solaris operating systems and their UFS file systems can be tuned by setting a lot of kernel variables. Unfortunately, system tuning looks like magic but if done properly, UFS file systems nearly achieve the same I/O throughput as raw devices.

2.3 Modeling Secondary Memory

In secondary memory computation, disks can be seen from a theoretical or engineering point of view. From a theoretical point of view, one wants to rank algorithms in complexity classes while still modeling the main features of disk drives. The theoretical model should be as simple as possible, and should hopefully allow us to compare algorithms among the various complexity classes. Engineering models are used to predict running times for I/Os exactly, thus modeling almost every feature of modern disk drives. Lets us take a closer look at engineering and theoretical disk models.

2.3.1 A simple engineering model

We introduce a very simple engineering model. An I/O transfers r bytes of data to the disk or vice versa. Our model calculates the time for that transfer as follows:

$$t_{service}(r) = t_{seek} + req\_size(r)/t_{transfer} \tag{2.1}$$

---

[1] This can be changed by system tuning.

where t_{seek} is the average seek time plus rotational delay in milliseconds (ms), $t_{transfer}$ is the maximum data transfer rate in Mbytes/sec and $req\_size(r)$ is the size of r in bytes. $t_{transfer}$ can by simply calculated by multiplying the number of sectors per track with the sector size and the disk's rotations per second. This transfer rate ranges between 5.9 MB/s (9 GB, 5 " drive with 5400 rpm) to 10 MB/s (9 GB, 3.5" drive with 7200 rpm). The calculation above ignores command overhead, cylinder switch time and time to transfer the data from the cache to the machine. Figure 2.3 shows the data transfer rate dependent from the request size for a 5400 rpm disk with an average seek time of 11 ms and an average rotational delay of 5.6 ms. It is obvious that on disks with a single read-write channel, the throughput is always limited by the amount of data, the disk can take from the surface in a unit of time (assuming that the bus is not saturated). In this model, t_{seek} and $t_{transfer}$ are just two constants. It is obvious that for small requests, $t_{service}$ is dominated by the seek time t_{seek} while for large requests, the dominant part is $req\_size(r)/t_{transfer}$. Service time is linear in the request size (see Figure 2.4). There is some startup time which consists of the average access time (average seek plus average rotational delay). Using figures 2.4 and 2.3, we can predict a request size for a random access that achieves reasonable data throughput together with a small service time. In our example, a good request size is somewhere between 64–128 kbytes. Service time is important because during read access, we have to wait until the data really arrives.

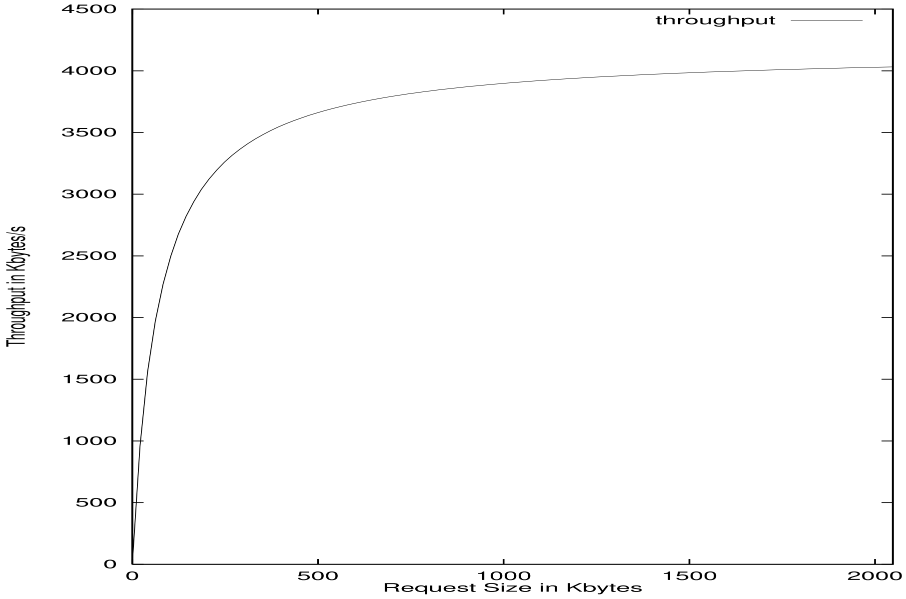


Figure 2.3: *Disk data transfer rate for a random access using the above model. The disk used is a 5400 rpm drive with an average seek time of 11 ms, a rotational delay of 5.6 ms and a sequential disk-to-cache transfer rate of 4168 kbytes/s.*

This formula tells us, that one should use big request sizes in order to hide the startup time t_{seek}. For example, an access to a single data item of one byte size would take

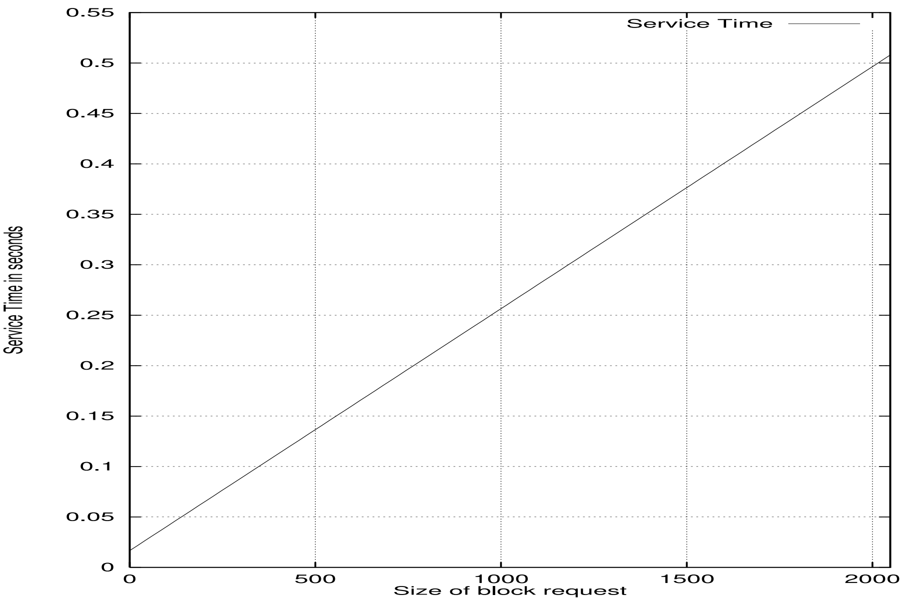


Figure 2.4: *Service time for a random access using the above model. The disk is the same as in Figure 2.3.*

$$t_{service} = 0.0166 \text{ sec} + 1/(4168 * 1024) \text{ sec} = 0.016600234 \text{ sec}$$

while an access to a block of 32 kbytes takes

$$t_{service} = 0.0166 \text{ sec} + 32/4168 \text{ sec} = 0.024277543 \text{ sec}.$$

The service time for the large request is only 1.46 times higher but the large request transfers 32868 times more data. This simple model resembles communication models in parallel computers where there is a large startup time and a high communication bandwidth.

Although the model describes the main disk features, it is not very accurate. Indeed, a seek on the disk is not independent from the request. Seek time can vary from a few milliseconds (track–to–track seek) to up to 20 milliseconds (for a full seek). Average seek time is often calculated by using the assumption that the requests are uniformly distributed over the surface. This also holds for rotational latency where it is assumed that half of a rotation is necessary to access the right sector. In reality, requests are not independent from each other and not uniformly distributed. $t_{transfer}$ is also not a fixed constant because the transfer rate varies with the zones of the disk: the transfer rate on the outer zones can be double as high as on the inner zones.

Cache is totally ignored as well as distribution of disk requests. Caches were introduced to speed up sequential access for read and write. During sequential access it is not as important to use large requests, as the disk caches either read ahead data or perform write behind. [Bos99] did a detailed measurement on a specific SCSI hard disk to investigate more in the effects of caching and zoning. [2]

---

[2] Thanks to Peter Bosch for providing figures 2.5 and 2.6.

Figure 2.5 shows the write throughput to the disk with caching enabled. The different plots refer to the different zones of the disk (0 is the outmost zone). Figure 2.6 shows the read throughput with disk cache disabled. Two observations can be made: Throughput on the inner zones is lesser than on the outer zones. With caches, it is possible to achieve the maximum data throughput already for smaller block sizes (compare the gradient of both curves). In Figure 2.5 we see that there is a peak at a block size of 200000 where the performance on the inner zone starts to degrade. The peak is somehow amazing. Data transfer to disk occurs at full bus speed to the disk cache. The theoretical throughput on the inner zones however is smaller than the bus throughput. With smaller block sizes, the cache is still able to hide the performance decrease on the inner zones. Although still simple, this model tells us important properties: very small block sizes are inefficient according to throughput and always increasing the block size does not necessarily lead to increased data throughput.

It is still not possible to accurately predict the run time for an I/O trace. Even without modeling disk caches, there is a big gap between this simple model and reality. At least the following modifications should be done according to Ruemmler and Wilkes [RW94]:

- accurate seek modeling
  Both [RW94, Bos99] did exact measurements to determine the seek function. The seek function is dependent on the seek distance. It is the square root of the distance for short and medium seeks and is linear in the seek distance for long seeks. Using a fixed constant t_{seek} often overestimates seek overhead and therefore as a conclusion we tend to choose larger block sizes (which is not always the best choice).

- modeling accurate rotational position
  It is necessary to model the exact location on the track in order to drop the assumption that the starting position of the request is uniformly distributed and drawn independently for each request.

- modeling cache behavior
  Cache has two important properties: it allows to reduce service time and it increases bandwidth for smaller requests. Figures 2.6 and 2.5 show that for a block size of 200000 bytes the bandwidth can be a factor of two greater for disk with caches compared to disks without caches. Thus without modeling disk caches we would prefer larger block sizes that would also lead to larger service times.

Ruemmler and Wilkes [RW94] developed a disk simulator that was able to execute I/O traces. They compared the accurate model to the real disk drive running the I/O trace. They were able to predict the real running time of their I/O trace within an error of 5.7 %.

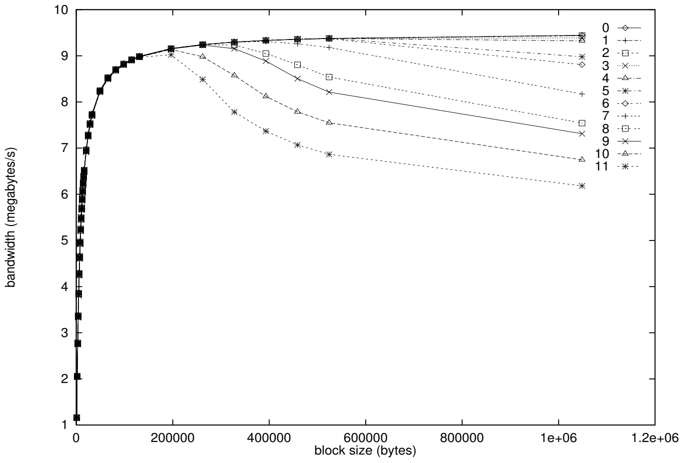


Figure 2.5: *Measured write performance on a Quantum Atlas-II disk by using factory settings, i.e. disk cache enabled.*

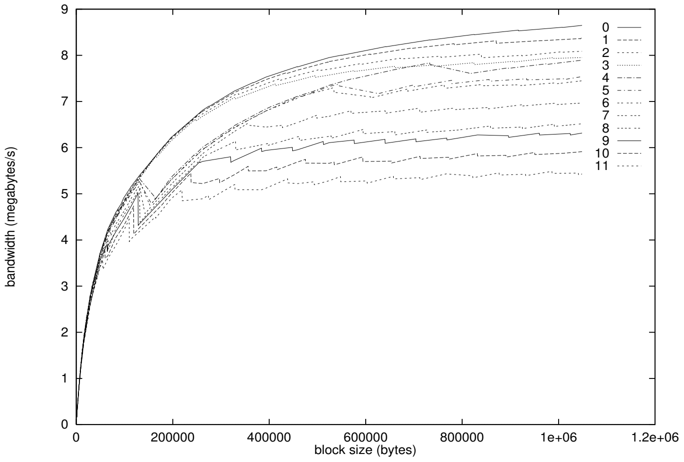


Figure 2.6: *Measured synchronous read performance of a Quantum Atlas-II disk with the disk cache disabled.*

2.3.2 Theoretical Secondary Memory Models

The earliest theoretical secondary memory model was introduced by Aggarwal and Vitter [AV88]. In their model, a machine consists of a CPU and a fast, internal memory of size M. Secondary memory is modeled by a disk having $P \geq 1$ independent heads. An I/O moves B items from the disk to the internal memory or vice versa. If $P > 1$, $P \cdot B$ items can be moved from independent positions on the disk to internal memory in one I/O. From a practical point of view, $P > 1$ does not exist in modern disk drives. Although each disk drive has several disk heads, they can't be moved independently from each other.

This model was refined in 1994 by Vitter and Shriver [VS94b]. They use a parameter D that describes the number of independent disks drives (see Figure 2.7). Additionally, they also specified multiprocessor disk models where P stands for the number of processors in the system. The D disks are connected to the machine, so that it is still possible to transfer $D \cdot B$ items in one I/O. The main difference to the original model of Aggarwal and Vitter is the fact that one single disk contains no parallelism in forms of independent heads. Although it is still possible to transfer $D \cdot B$ items in one I/O, data layout on the D disks now plays an important role to achieve a speedup of D over the one single disk case. Both models normally assume that disk blocks are indivisible and that it is only possible to perform a computation on data that resides in internal memory.

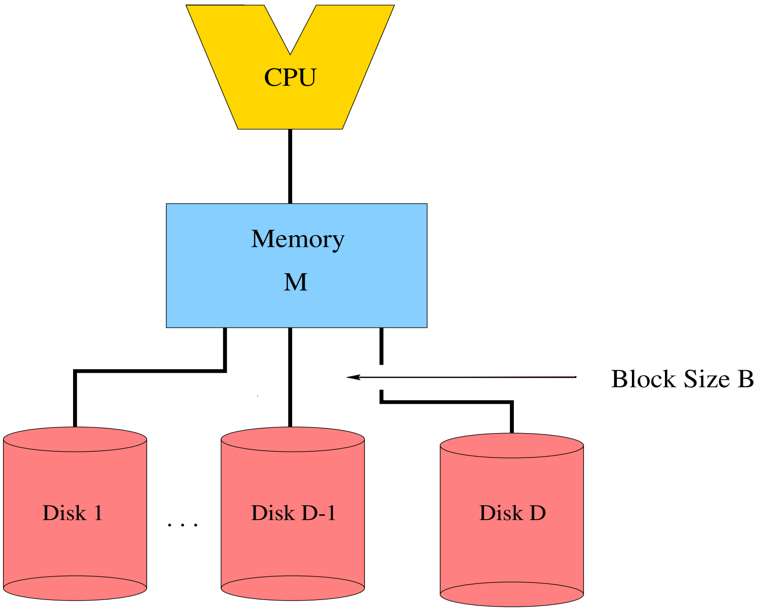


Figure 2.7: *The secondary memory model of Vitter and Shriver*

This is up to now the standard complexity model for secondary memory computation. Algorithmic performance is measured in this model by counting:

(i) the number of executed I/O operations

(ii) the number of performed CPU operations (RAM model), and
(iii) the number of occupied disk blocks.

We summarize important lower bounds in that model:

1. *Scanning* (or *streaming* or *touching*) a set of N items takes $\Theta(N/(DB))$ I/Os.
2. *Sorting* a set of N items takes $\Theta(\frac{N}{DB} log_{M/B}(N/B))$ I/Os [AV88].
3. *Sorting* a set of N items consisting of k distinct keys, $k < N$, takes $\Theta(\frac{N}{DB} log_{M/B} k)$ I/Os [MSV00].
4. *Online search* among N items takes $\Theta(log_{DB} N)$ I/Os.

The model is quite simple and allows to rank algorithms in form of Big Oh-notation. It tries to capture the most important disk properties in the following way: (i) it amortizes the seek overhead of disk drives by always transfering B items at a time. Our simple engineering model already tells us that one should transfer large blocks of data in order to achieve reasonable disk data throughput together with small service times. The indivisibility assumption of blocks comes from the fact that the smallest transfer unit of a hard disk is a sector. Some limitations still exist. This model does not separate between random and sequential I/Os, *i.e.* it does not model the seek accurately. In practice, there is a big difference in terms of service time and transfer time, if we compare x requests to random disk locations with x requests to consecutive disk locations. The reasons are caching and non-linearity of seeks (see Section 2.3.1). If two algorithms perform exactly the same number of I/Os, the one with the more sequential I/Os is faster. This fact should be integrated into the model without significantly enlarging the number of its parameters.

2.3.2.1 The I/O Model of Farach *et al.*

Farach *et al.* [FFM98] refined the secondary memory model of [VS94b] and introduced a difference between random and sequential I/Os. In the classical secondary memory model, all I/Os are to random disk locations. Their model simply accounts I/Os in a different way: they introduce the term *bulk* I/O which are $c \cdot \frac{M}{B}$ I/Os to consecutive disk locations. All other I/Os are random I/Os. The model can estimate costly seeks in a more reasonable way than the model of Aggarwal and Vitter. Costly seeks are upper-bounded by the sum of bulk and random I/Os. This is still an upper bound but it is much better than that derived by Aggarwal and Vitter where every I/O involves a costly seek. Furthermore it helps to understand more easily the influence of caches as they can speed up the disk throughput and thus often the running time of the algorithm when lots of disk accesses to consecutive disk locations occur. For sorting, Farach *et al.* state the following lemma:

Lemma 2.3.1. *[FFM98] Sorting N items is possible in $O((N/B)\log_{M/B}(N/B))$ random I/Os or $O((N/M)\log_2(N/M))$ bulk I/Os, which is optimal.*

A simple observation shows that choosing $c = \frac{1}{2}$ maximizes the number of bulk I/Os in external multiway mergesort but leads to two way merging . This is not desirable as the total number of I/Os is $O((N/B)\log_2(N/B))$ I/Os while the optimal number of I/Os is $O((N/B)\log_{M/B}(N/B))$ [AV88]. This is in some sense “strange”, but for the experimental analysis choosing the bulk size as a constant allows to get more deeper knowledge about the algorithmic performance. As a drawback, algorithmic design and analysis gets now more complicated as a detailed data layout for disk data is necessary in order to analyze bulk and random I/Os. In later chapters, we will explain in more detail how to choose a reasonable bulk size for a given disk system.

2.4 Summary

There is a lot of difference between theoretical secondary memory models and engineering disk models. While engineering models are used to predict the running time for I/O traces thus capturing all important disk features and hence using a lot of parameters; theoretical models are used to analyze I/O intensive algorithms and data structures. The exact disk access pattern cannot be predicted accurately enough to use engineering I/O models. Therefore the theoretical models are used to count in a simple way the number of performed I/O operations. Theoretical models should be simple and use a small number of parameters. During algorithmic description, we will always use the secondary memory model of Vitter and Shriver [VS94b] to analyze secondary memory algorithms. The engineering model of Section 2.3.1 will be used to choose the block size B in a reasonable manner, and the model of Farach *et al.* [FFM98] will be used in the experiments to get some knowledge about the number of bulk I/Os and to derive upper bounds on the number of seeks. In the next chapter, we will describe the layout and implementation of LEDA-SM, a C++ software library for secondary memory computation.

Chapter 3

LEDA-SM

During the last years, many software libraries for *in-core* computation have been developed. As data to be processed has increased dramatically, these libraries are often used in a *virtual memory* setting where the operating system provides enough working space for the computation by using the computer's secondary memory (mostly disks). Secondary memory has two main features that distinguishes it from internal main memory:

1. Access to secondary memory is slow. Hard disks are mechanical devices while main memory is an electronic device. An access to a data item on a hard disk requires moving disk mechanics (see Chapter 2) and therefore takes much longer than an access to the same item in main memory. The relative speed of a fast cache and a slow secondary memory is close to one million and is still in the thousands for main memory and secondary memory.

2. Secondary memory rewards locality of reference. The access time to a single data item and a consecutive block of data items is approximately the same (see the example in Section 2.3.1 and Figure 2.4). Therefore, secondary memory (disks) and main memory exchange data in blocks to amortize the seek time overhead of the disk drive. A block transfer between secondary and main memory (and vice versa) is called *I/O operation* (shortly I/O).

Most of the data structures and algorithms, implemented in todays software libraries, are designed for the RAM model [AHU74]. The main features of this model are unlimited memory and unit cost access to memory. It has been observed that most of the algorithms, designed for the RAM-model, access memory in an unstructured way. If these algorithms are transfered to a secondary memory setting they cannot profit from the locality of reference of disk drives and hence frequently suffer an intolerable slowdown as they perform huge amounts of I/Os.

In the recent years, the algorithmic community has addressed this issue and has developed I/O-efficient data structures and algorithms for many data structure, graph-theoretic, and geometric problems [VS94b, BGV97, CGG$^+$95, GTVV93, FG95]. The data structure community has a long tradition dealing with secondary memory. Efficient index structures, *e.g.* B-trees [BM72] and extendible hashing [FNPS79], have been designed and highly optimized implementations are available. Besides the classical sorting and searching problems, ”general purpose” secondary memory computation has never been a major concern for the database community. In the algorithmic community, implementation work is still in its infancy. There are few implementations of particular data structures [Chi95, HMSV97] or I/O simulators [BKS99]. They aim at investigating the relative merits of different data structures, but not at external memory computation at large. There is no general concept of providing secondary memory computation to the user, file I/O is directly done in the algorithm implementation and the programmer has to directly care about handling files and read or write access. I/O simulators are even more restrictive. While specialized implementations provide code that really performs the computation and delivers a computational result in the end, simulators only count I/Os. They do not actually perform the I/Os and are therefore only able to compare secondary memory algorithms by counting I/Os. However, this does not always lead to correct results as, even for secondary memory algorithms, CPU time cannot be ignored and I/O may not always be the dominating part in the total execution time. Thus comparing secondary algorithms only by counting I/Os may lead to wrong results. At the moment there are only two systems that provide secondary memory computation in a more general and flexible context. TPIE [VV95] (Transparent Parallel I/O Environment) is a C++ library that provides some *external programming paradigms* like scanning, sorting and merging sets of items [1]. TPIE realizes secondary memory by using several files on the file system. In fact, each data structure or algorithm uses its own file. Several different file access methods are implemented. TPIE only offers some more advanced secondary memory data structures and most of them are based on the external programming paradigms. Direct access to single disk (file) blocks is possible but somehow complicated as the design goal of TPIE is to always use the external programming paradigms. TPIE offers no connection to an efficient internal-memory library that is necessary when implementing secondary memory algorithms and data structures. Both features were missed by users of TPIE [HMSV97], it is planned to add both features to TPIE [Arg99].

Another different approach for secondary memory computation is ViC*, the Virtual Memory C^* compiler [CC98]. The ViC* system consists of a compiler and a runtime system. The compiler translates C* programs with shapes declared `outofcore`, which describe parallel data stored on disk. The compiler output is a program in standard C with I/O and library calls added to efficiently access out-of-core parallel data. At the moment, most of the work is focussed on out-of-core fast Fourier transform [CWN97], BMMC permutations [CSW98] and sorting. No other secondary data structures are provided. As for TPIE, there is no connection to a highly efficient internal memory library that is commonly needed for algorithmic design of new data structures or algorithms.

In 1997, we decided to develop a new library for secondary memory computation. The main idea was to make secondary memory computation publicly available in an easy way. The library, later called *LEDA-SM* (LEDA for secondary memory), should be easy to use (even by non-experts) and easy to extend. LEDA-SM is a C++ library that is connected to the internal-memory library LEDA (Library of Efficient Data types and Algorithms) [MN99, MN95]. It therefore offers the possibility to use advanced

---

[1] TPIE is developed at the University of Duke (see also www.cs.duke/edu/˜tpie).

and efficient internal memory data structures and algorithms in secondary memory algorithmic design and therefore saves important time in the development of new data structures and algorithms as recoding of efficient internal memory data structures or algorithms is not necessary.

3.1 Design of the Library LEDA-SM

We describe in detail the design of the LEDA-SM library. When we started to develop LEDA-SM, our main idea was to follow the main design goals of the LEDA library which are *ease-of-use, portability* and *efficiency*.

1. *Ease-of-use*
   Algorithms and data structures provided by the library should be easy to use. Setup of secondary memory should be easy, all functions should be well documented. We provide a precise and readable specification for all data structures and algorithms. The specifications are short and abstract so as to hide all details in the implementation. This concept of specification and documentation of algorithmic work was adopted from the LEDA project. We use CWEB [KL93] and LEDA's CWEB extensions to document our code. Secondary memory setup is done semi-automatically by a setup-file and checked at program start. No specific knowledge about system implementation is necessary. We will describe this in more detail in the following sections.

2. *Portability*
   The library should be portable to several computing platforms, *i.e.* using several compilers, hardware platforms and operating systems. This means that access and management of secondary memory must be portable across several platforms. As a consequence we must provide at least one access method to secondary memory that works on all supported platforms so that the implementation of secondary memory and the access to it does not rely on machine-dependent routines or information.

3. *Efficiency*
   All supported data structures and algorithms should be efficient in terms of CPU usage, internally used memory, disk space and number of I/Os. Portability and efficiency can be conflicting goals: in order to be portable we cannot rely on machine-specific implementations but these are often the most efficient ones. We circumvent this problem by providing portable and non-portable code for specific platforms.

The main design goal of the library was to follow the theoretical model of [VS94b] as described in Section 2.3.2. There each disk is a collection of a fixed number of blocks of fixed size; the blocks are indivisible and must be loaded into internal memory if computation should be performed on them. We therefore directly pay attention to the specific features of hard disks (sectors are the smallest accessible indivisible units, see Section 2.1) and file systems (fractions of logical disk blocks are the smallest accessible, indivisible units, see Section 2.2).

Library LEDA-SM is implemented in C++. The library is divided into two layers, *kernel layer* and *application layer*. The kernel layer is responsible for managing secondary memory and the access to it. The application layer consists of secondary memory data structures and algorithms. LEDA is used to implement the in-core part of the secondary memory algorithms and data structures and the kernel data structures of LEDA-SM. One of the central design questions is the realization of secondary memory and the access to it. Secondary memory consists of several disks and each disk is a collection of blocks. The following questions arose:

1. How do we provide secondary memory?
   There are several choices to model secondary memory. One way is to use files of the file system and the other is to use directly the disk devices. Furthermore, we can either model a whole disk drive, or we can assign portions of the secondary memory (*i.e.* files) to data structures.
2. How do we implement access to secondary memory?
   Depending on the choice of the secondary memory realization, I/O operations can either be done by file access methods or one can use direct disk access to the disk drive. The latter method possibly needs specific disk device drivers.
3. Is it necessary to model disk block locations and disk block users?
   If secondary memory is modeled as disk drives, we need a way to manage the system resources of disk drives which are the individual disk blocks. As several data structures might own disk blocks on the same disk, it is necessary that disk block owner checks can be performed.

On the basis of these questions, the following three design methods are possible:

Method I. The first method is using several disk drives and low-level access to the disk, thus directly providing secondary memory by means of the hard disk itself. The advantage is that this method provides fast disk access as it does not use any additional software layers of the operating system, such as file systems, to handle I/O operations. Access to disk data is handled by using machine-dependent low-level disk access routines (known as *disk device drivers*) to transport data to or from the disk. In order to avoid that one data structure overwrites data of another data structure on the disk, it is required to map used disk blocks to data structures/algorithms (*users*) and to check if it is allowed to access a specific disk block. This task in general resembles the problem of managing main memory in multitasking operating systems (see for example [Dei90]).

Methods II and III provide secondary memory by using the file system of the operating system. Secondary memory is provided by files on the file system and access is realized by file access methods (see Section 2.2). As portability was one of the major design goals and low-level disk access is different on every platform, we decided to use a file system based view for secondary memory, thus concentrating on methods II or III. Method II uses a file for each data structure/algorithm or parts of it, while method III uses a file for each disk that is modeled. Methods II and III both have several advantages and disadvantages.

Method II. As every data structure uses its own file(s), there is no need to manage users and block locations directly and there is no need for a general protection mechanism in order to avoid overwriting data of other data structures. All these tasks are indirectly handled by the file system itself. Logical disk block locations inside the file can simply be translated into seek-commands, there is no need to manage these logical disk block locations, and allocation of new space is done via an extension of the files. There are however some limitations. Operating systems normally limit the allowed number of open files per process. The reason is that each file descriptor needs some amount of space that is often statically allocated to the process. As a consequence, one could simply reach that limit and thus force the library to close and reopen files. Unfortunately opening and closing files is a costly process in terms of CPU usage. Much more important is the fact that allocation and deallocation of disk space is only done by the operating system. Since many files can be used on the file system and since files dynamically grow throughout computation, it is very likely that most of theses files are not physically contiguous on disk. This fact reduces I/O performance on the file system and makes prefetching of data (read-ahead) quite useless. Furthermore it is not possible to convert this view of secondary memory to low-level disk access without completely changing the library design. An example where this implementation idea was used is TPIE.

Method III. An abstract disk is modeled by a specific file of fixed size and is logically divided into blocks of a fixed size. The size of the disk (*i.e.* the corresponding file) is fixed, thus modeling the fact that the size of a disk is also limited in reality. As several data structures (users) normally use disk space on the same abstract disk, there is a need to model abstract disk blocks and owners of these disk blocks. Disk blocks can now be in use or free, the kernel must be able to keep track of this and must provide methods for allocating and freeing disk blocks. During access to disk blocks, it is necessary to perform owner checks in order to avoid overwriting data of other users. This approach has again limitations. Depending on the platform, file size is limited to 2 Gigabytes (32 bit file systems) thus restricting the abstract disk size. More modern operating systems support 64 bit file systems and thus allow files to grow up to 2 Terabytes. Method III is a software model of method I where we simulate a real hard disk via the file system. The simulation is simplified as we do not model cylinders and tracks but we see a disk as a collection of disk blocks [2]. The main advantage of this form of design is the fact that it is possible to switch to a machine-dependent low-level (*raw*) disk access without rewriting the whole kernel. For Solaris-based operating systems, this can even be done without code changes. The raw disk device can be accessed via standardized file system calls. As the kernel manages disk blocks, users and allocation/deallocation by its own, one can simply switch to the low-level disk access by using device names for the abstract disk instead of file names in the file system. For all other operating systems that do not provide raw disk device access as an operating system feature, it is necessary to write device drivers. At the moment, we only support raw device access for Solaris platforms.

We have chosen design method III for the kernel of LEDA-SM. It provides the flex-

---

[2] This view is also often provided by low-level SCSI disk device drivers.

ibility of using raw disk access along with file access and therefore allows us to use either file system independent, low-level disk access or portable file access. The file size limit is actually not a problem for modern operating systems as most of them provide a 64-bit file system. In the LEDA-SM library, secondary memory consists of NUM_OF_DISKS logical disks (*i.e.* files), $max\_blocks[d]$ blocks can reside on the d-th disk, $0 \leq d < NUM\_OF\_DISKS$. A block on disk is able to store EXT_BLK_SZ items of type $GenPtr$, where type $GenPtr$ is the generic pointer of the machine (C++ type $void*$). Disk blocks are indivisible, *i.e.* we always have to transfer whole disk blocks during read or write.

The LEDA-SM library consists of two layers. The *kernel layer* is responsible for disk space management and disk access. It furthermore implements an interface that allows to perform block-oriented disk access in a user friendly way. The kernel of LEDA-SM consists of seven C++ classes and is divided into the abstract and the concrete kernel (see Table 3.1). The concrete kernel consists of four C++ classes. These classes are responsible for performing I/Os, managing used and non-used disk blocks and managing users of disk blocks. There are several different implementations for each of these tasks. These classes are built into the library and cannot be changed. The abstract kernel implements a user-friendly access interface to the concrete kernel. It consists of three C++ classes; two of them model abstract disk block locations and owners/users of disk blocks. The third class is a container class that simplifies the block oriented access to secondary memory. It is able to transport blocks of items of data type E to and from secondary memory (see in detail Section 3.2.3). The *application layer* consists of a collection of external memory algorithms and data structures. The current implementations of all applications use the classes of the abstract kernel to simplify the access to secondary memory. This is not absolutely necessary but it allows to write readable applications and simplifies the development process. In other words, the classes of the abstract kernel define an *access interface* to secondary memory that can be used by application programmers to simplify data structure and algorithm development.

Layer	Major Classes
algorithms	sorting, graph algorithms, ...
data structures	ext_stack, ext_array, ...
abstract kernel	$block{<}E{>}$, B_ID, U_ID
concrete kernel	$ext\_memory\_manager$, ext_disk, ext_free_list, $name\_server$

Table 3.1: *The different layers of library LEDA-SM.*

Figure 3.1 summarizes the kernel layout. We use UML as modeling language [FS00] (see Appendix B for a short introduction to UML). The dark horizontal line separates the abstract and the concrete kernel. From now on it is necessary to distinguish between blocks as physical objects (= a region of storage on disk) and logical

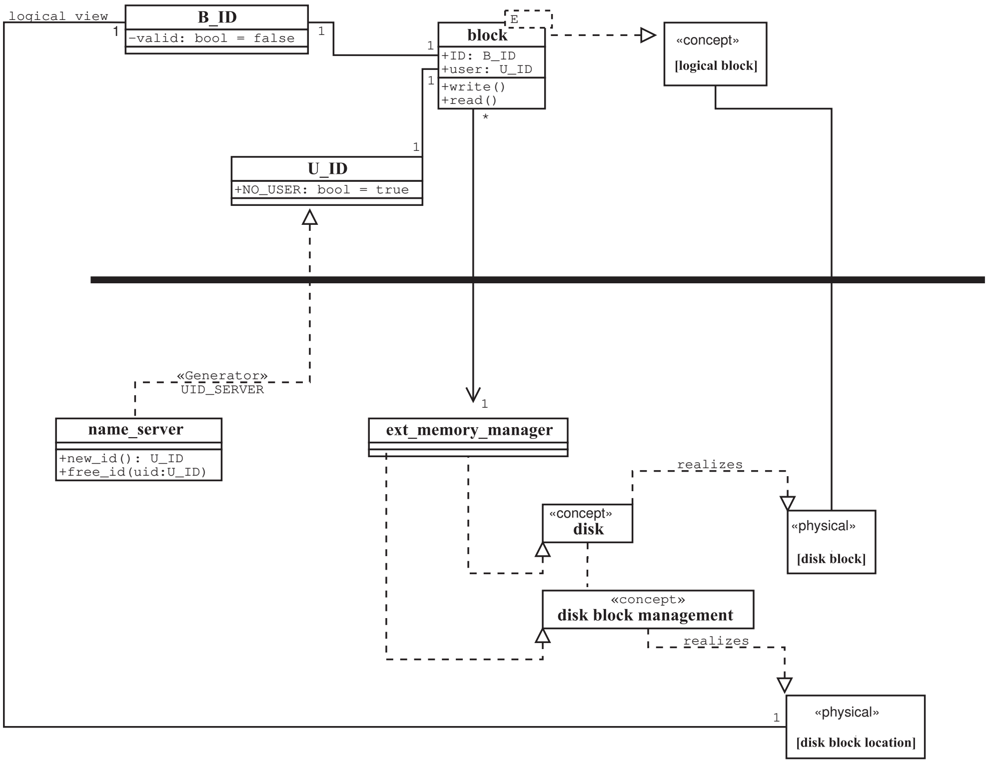


Figure 3.1: *Layout of LEDA-SM's kernel. We use UML as a modeling language.*

objects (= a bit pattern of particular size in internal memory). We will use the word *disk block* for the physical object and reserve the word *block* for the logical object.

The central classes of the concrete kernel are class $name\_server$ and class $ext\_memory\_manager$. Class $ext\_memory\_manager$ realizes disks and disk block management, while $name\_server$ generates user identifiers. There is only one instance of each of these two classes. Each disk consists of a collection of linearly ordered disk blocks, a disk block is an array of type $void*$ of size EXT_BLK_SZ, where EXT_BLK_SZ is a system constant that specifies the size of a disk block (parameter B of the secondary memory model). Class $ext\_memory\_manager$ uses class ext_disk to realize the disks and the access to the disk blocks; class $ext\_freelist$ is used to manage the used and free disk blocks of a disk.

The abstract kernel provides logical concepts of disk block identifiers (B_ID) and blocks ($block{<}E{>}$), these concepts are associated with their physical counterparts in the concrete kernel (see Figure 3.1). Logical block identifiers are used to specify a disk block on a specific disk. Class $block$ is used to provide a typed view (type E) of a disk block. We associate with each instance of class block one object of type B_ID and one object of type U_ID. Type U_ID is used to model users of disk blocks as well as logical blocks. The concepts "$block$", "B_ID" and "U_ID" are associated with the concrete kernel in the following way:
each valid object of type B_ID refers to a disk block location in secondary memory,

and each block refers to a disk block in secondary storage via its unique block identifier (B_ID). Disk blocks are either owned by a specific user or they are free (U_ID `NO_USER`).

The seven classes of the kernel together with the in-core algorithms of the LEDA library are used to implement all secondary memory data structures and algorithms currently available in the LEDA-SM library.

We will now describe the different kernel classes and their implementation in more detail.

3.2 The abstract kernel

The abstract kernel consists of the three C++ classes B_ID, U_ID and $block$<E>. Class B_ID is used to simplify the access to disk block locations in secondary memory. All file access and raw device access methods specify the location as an offset in bytes (see Section 2.2.3). We simplify this by dividing the disk file into fixed size blocks of size EXT_BLK_SZ. These blocks are numbered linearly. Thus, block x specifies the disk file locations $[x \cdot EXT\_BLK\_SZ, \dots, (x+1) \cdot EXT\_BLK\_SZ - 1]$. Class U_ID models users of disk blocks. This is necessary as we manage allocation and deallocation of disk blocks by our own, and as we need to avoid that a data structure inadvertently overwrites disk blocks of another data structure. Parameterized class $block$<E> is used to simplify access to and from disk blocks. Since data in files and on the disk is untyped, we need a container class that is able to transport a block of items of type E to or from the disk. Typing and untyping of the data, it contains, should be done automatically. We will now describe all classes of the abstract kernel in more detail.

3.2.1 Disk Block Identifier (B_ID)

An instance BID of type B_ID is a tuple (d, num) of integers where d specifies the disk and num specifies the logical block of disk d. Block identifiers are abstract objects that are used to access physical locations in external memory. It is allowed to perform basic arithmetic operations like addition, substraction, etc. on objects of type B_ID. Arithmetic operations only affect entry num of type B_ID. A block identifier (shortly `block id`) is called $valid$ if $0 \leq d < NUM\_OF\_DISKS$ and $0 \leq num < max\_blocks[d]$. If num is equal to -1, we call the block identifier $inactive$. An inactive block identifier is not connected to any physical disk location.

Class B_ID is implemented in a simple way:

32 ⟨*block identifier* 32⟩≡

```
class B_ID
{
  int D;
  int num;
```

```
public:
[ Member functions ]
};
```

where D specifies the number of the disk and num specifies the logical block number on that disk. There exist member functions to create or modify block identifiers as well as to perform arithmetic operations on type B_ID (see manual page of class B_ID in the Appendix). Block identifiers are used in the application code to specify logical disk block locations in secondary memory. They are mostly used together with class $block{<}E{>}$ to access physical disk blocks in an easy way. They are necessary for the management of unused and used disk blocks.

3.2.2 User Identifier (U_ID)

User identifiers (shortly `user id`) are used to specify users of disk blocks. A user identifier (type U_ID) is implemented by type int. There exists a special user id NO_USER which is used by class $block$. A specific concrete kernel class ($name\_server$) is used to manage allocated and free user identifiers (see Section 3.3).

The most important class in the abstract kernel is responsible for providing a simple method to transport blocks of data type E to secondary memory or vice versa and for representing blocks in internal memory. This is done by the parameterized class $block{<}E{>}$. The next subsection gives a detailed specification of class $block{<}E{>}$. This kind of documentation is used in the LEDA-SM manual.

3.2.3 Logical Blocks ($block{<}E{>}$)

Class $block$ was designed to provide the abstract view to secondary memory as introduced in the theoretical I/O model of Vitter and Shriver [VS94b] (see also Section 2.3.2). Physically, a disk block is a consecutive region of storage of fixed size B on the disk. Logically, a disk block contains some fixed number of elements of type E, this elements can be indexed like in an array. Thus the main difference is that disk blocks are untyped (type $void\ *$) while logical blocks are typed (type E). Class `block` is a container class, parameterized by type E, that allows indexed access to the elements of type E and that is able to transport these items to secondary memory and vice versa.

1. Definition

An instance B of the parameterized type $block{<}E{>}$ is a typed view of logical internal memory blocks. It consists of an array of links and an array of variables of data type E. The array of links stores links to other blocks, a link is an object of data type B_ID. The second array stores variables of data type E. The variables of type E are indexed from 0 to $blk\_sz - 1$, the links are indexed from 0 to $num\_of\_bids - 1$. The number of links num_of_bids is set during the creation. The number of variables of type E, blk_sz, is dynamically calculated at time of construction: $blk\_sz = (BLK\_SZ - num\_of\_bids * sizeof(B\_ID))/sizeof(E)$, *i.e.* the size of a block is determined by the system constant BLK_SZ and the possible number of links num_of_bids to other blocks. We furthermore associate a block identifier and a user identifierwith each instance B of type $block$. The block identifier (B_ID) and the user identifier (U_ID) of instance B are used during write or read access to external memory.

2. Creation

$block{<}E{>}\ B$;	creates an instance B of type $block$ and initializes the number of links to zero. At the time of creation, the block identifier is $invalid$ to mark that the block is not connected to a physical location in external memory. The internal user identification is set to NO_USER.
$block{<}E{>}\ B(U\_ID\ uid, int\ bids = 0)$;	creates an instance B of type $block$ and initializes the number of links num_of_bids to $bids$ and the user id to uid. At the time of creation, the block identifier is $invalid$ to mark that the block is not connected to a physical location in external memory. There is the possibility to create a block with user identification NO_USER and to later set the user indentification to the actual user. The user-id can only be set once.

3. Operations

int	B.size( )	returns the the number of elements of data type E of B.
int	B.bytes( )	returns the size of B in bytes.
int	B.num_of_links( )	returns the number of links to other blocks.
B_ID	B.write( )	writes the block into the disk block specified by the block's internal block id. If the block id is inactive, a new unused block id is requested from the external memory manager and the block is written. *Precondition*: B's user identification is not equal to NO_USER. Otherwise an error message is produced and the application is stopped.

void	B.write(B_ID bid)	writes the block into disk block specified by bid and sets the block identifier of B to the inactive block indentifier. *Precondition*: Block identifier bid must be $valid$, the user of B must own the disk block specified by bid and B's user id must be different from NO_USER. If this is not the case, an error message is produced and the application is stopped.
void	B.read( )	reads the disk block specified by B's internal block identifier into B. *Precondition*: the internal block id must be $valid$ and B's internal user identification must be different from NO_USER.
void	B.read(B_ID bid)	reads the disk block specified by block identifier bid into B. *Precondition*: bid must be valid and B's internal user identification must be different from NO_USER. If this is not the case, an error message is produced and the application is stopped.
void	B.read_ahead(B_ID bid, B_ID $ahead\_bid$)	reads the disk block bid into B and starts read-ahead of disk block $ahead\_bid$. *Precondition*: The disk-I/O implementation must allow asynchronous I/O, all preconditions of $read()$ must be fullfilled for block ids bid and $ahead\_bid$.
$B\_ID\&$	B.bid(*int* i)	is used to access the i–th link element of B. A synonym is $B(i)$. *Precondition*: $0 \leq i < num\_of\_links$
B_ID	B.id( )	returns the internal block identifier of B. This block identifier is either inactive or bound to a specific disk block of the external memory manager.
void	B.set_id(B_ID $newid$)	sets the internal block identifier of B to $newid$.
U_ID	B.user( )	returns the internal user identifier of B.
void	B.set_user(U_ID uid)	sets the internal user identifier of B to uid. *Precondition*: the internal user identifier must be `NO_USER`.
bool	B.is_active( )	returns true if B has an active block identifier, false otherwise.

Array Operators

$E\&$ $B[\mathit{int}\ i]$

returns a reference to the contents of variable element i.
Precondition: $0 \leq i < \mathit{blk\_sz}$

$B\_ID\&$ $B(\mathit{int}\ i)$

returns a reference to the contents of link element i.
Precondition: $0 \leq i < \mathit{num\_of\_links}$

Assignment Operators and Copy Operators

void $B\ =\ \mathit{const\ block{<}E{>}\&\ t}$

copies block t into B.

Static Members

void $\mathit{block}::$gen_array$(\mathit{array}{<}\ \mathit{block}{<}E{>}\ {>}\&\ A,\ \mathit{int\ number},\ U_ID\ \mathit{uid},\ \mathit{int\ bids} = 0)$

initializes an array of blocks. A is a reference to the array to be initialized, number is the size of the array. The user-identifier of each block is set to uid and the number of links of each block is set to bids.

B_ID $\mathit{block}::$write_array$(\mathit{array}{<}\ \mathit{block}{<}E{>}\ {>}\&\ A)$

writes the array of blocks A to the external memory. The function writes to consecutive locations in external memory, the internal block identifier of the first array entry of A is used to determine the starting position ($A[0].\mathit{bid}()$). In the case that this block identifier is inactive, disk blocks in external memory are requested. The return value is the block identifier to which the first array element of A was written.
Precondition: The preconditions are the same as for method write.

void $\mathit{block}::$read_array$(\mathit{array}{<}\ \mathit{block}{<}E{>}\ {>}\&\ A)$

reads $A.\mathit{size}(\ )$ consecutive blocks from external memory into the array of blocks B. The function starts reading from the block position that is specified by $A[0]$ internal block identifier ($A[0].\mathit{bid}()$).
Precondition: The internal block identifier of $A[0]$ is not inactive. All other preconditions of read apply.

void	*block*::read_array(*array*< *block*<*E*> >& A, *B_ID bid*, *int num* $= -1$)	
		reads $\min\{A.size(\,), num\}$ consecutive blocks from external memory starting at block location *bid* into A. If *num* is negative, $A.size(\,)$ blocks are read. *Precondition*: All preconditions of $read(B\_ID)$ apply.
int	*block*::elements(*int bids* $= 0$)	
		returns number of elements of type E which fit into one block using *bids* links.

4. Implementation of Class *block*<*E*>

Parameterized data type *block*<*E*> is implemented as follows:

37a ⟨ *block*<*E*> 37a⟩≡

```
template <class E>
class block
{
  int blk_sz;
  int num_of_bids;
  GenPtr *A;

public:

[ method declarations; ]


};
```

An item B of data type *block*<*E*> consists of two variables of type *int* and a pointer of type *GenPtr*. The first variable *blk_sz* holds the number of entries of type E that can be stored in item B of type *block*<*E*>. The second variable *num_of_bids* stores the number of links of type *B_ID* to other blocks. Pointer A of type *GenPtr* (*GenPtr* is the generic system pointer type, normally type *void*∗) is later used to allocate space of size $EXT\_BLK\_SZ * sizeof(GenPtr)$ bytes. This is done at the time of B's construction. The array of *GenPtrs* is used to store the objects of type *B_ID*, the objects of type E and the internal block-id and user-id of B. These objects are created in-place inside the array A.

37b ⟨*block*<*E*>*::block()* 37b⟩≡

```
template<class E>
block<E>::block(U_ID myid,int bids)
{
  if(bids < 0)
    {
      std::cerr << "block::block\n";
```

```
      std::cerr << "number of bids is negative:" << bids
                << std::endl;
      abort();
    }
  if(num_of_bids*sizeof(B_ID)> BLK_SZ)
    {
      std::cerr << "block::block is to small, number of bids "
           << num_of_bids
           << " total size in bytes is "
           << num_of_bids*sizeof(B_ID)
           << " ,block size is "
           << BLK_SZ
           << " , application is stopped !\n";
      abort();
    }
  num_of_bids = bids;
  A = LEDA_NEW_VECTOR(GenPtr, EXT_BLK_SZ);
  for(int i=0;i<EXT_BLK_SZ;i++) A[i] = nil;
  if(sizeof(E) >= sizeof(GenPtr))
    blk_sz = (sizeof(GenPtr)*BLK_SZ -
              num_of_bids*sizeof(B_ID))/sizeof(E);
  else
    blk_sz = (sizeof(GenPtr)*BLK_SZ -
              num_of_bids*sizeof(B_ID))/sizeof(GenPtr);

  A[BID_POS] = (GenPtr) -1;
  A[BID_POS+1] = (GenPtr)  -1;
  A[UID_POS] = (GenPtr) myid;
}
```

The non-default constructor $block(myid, bids)$ is normally used to create a new item B of type $block{<}E{>}$. Parameter $myid$ of type U_ID identifies the user of item B; $bids$ specifies the number of desired links of type B_ID that are stored inside of B. We dynamically allocate space for the array A of EXT_BLK_SZ entries of type $GenPtr$ by using the LEDA macro `LEDA_NEW_VECTOR()`. BLK_SZ entries are available to store items of type E and type EB_ID, the rest of the available space ($EXT\_BLK\_SZ - BLK\_SZ$ GenPtrs) is used to store maintainance information, consisting of B's user identifier and block identifier. Depending on the size of type E (`sizeof(E)`) and the number bid of links of type B_ID, we calculate the number of entries of type E that can be stored in A. This value is assigned to variable blk_sz. Elements of type E and type B_ID are later created in-place of array A. Entries of type B_ID are stored before the entries of type E. The in-place creation and assignment is forced by the following code segment (for type E):

38 ⟨*block<E>::operator[](int it)* 38⟩≡

```
template<class E>
E& block<E>::operator[](int i)
{
  if ((i< 0) || (i > blk_sz) )
    {
```

```
        std::cerr << "block[]::index out of bound " << i
                  << std::endl;
        std::cerr << "block_size is " << blk_sz
                  << std::endl;
        abort();
      }

    if (sizeof(E) >= sizeof(GenPtr))
      return (E&)A[(num_of_bids*sizeof(B_ID)+
                i*sizeof(E))/sizeof(GenPtr)];
    else
      return *(E*)(&A[(num_of_bids*sizeof(B_ID))/
                      sizeof(GenPtr)+i]);
  }
```

If data type E is smaller than data type $GenPtr$, we use an item of type $GenPtr$ and create data type E in-place of the $GenPtr$. This may possibly waste some space, *i.e.* type $char$ is one byte and type $GenPtr$ is four bytes (on a 32-bit CPU). The code line $*(E*)(\&A[...])$ forces the in-place-assignment. If data type E is larger than type $GenPtr$, we use $\lceil sizeof(E)/sizeof(GenPtr)\rceil$ many consecutive array entries to create the object of type E in-place. This is done by the code line $(E\&)A[...]$. A similar code segment is used to access the items of type B_ID.

The key trick of this implementation is that the underlying data structure is an array of type $GenPtr$. This easily allows to directly write item B of type $block{<}E{>}$ to disk using any of the discussed file access methods of Section 2.2.3 as all file access methods assume that the data to be written or read is stored in a buffer of type $void*$. Thus, reading or writing an item B of type $block{<}E{>}$ to disk is simple:

39 ⟨*block<E>::write(B_ID bid)* 39⟩≡

```
  template<class E>
  void block<E>::write(B_ID bid)
  {
    U_ID uid;
    B_ID inactive;

    uid = (U_ID) A[UID_POS];
    if (uid != NO_USER)
    {
      A[BID_POS] = (GenPtr) bid.number();
      A[BID_POS+1] = (GenPtr) bid.get_disk();

      ext_mem_mgr.write_block(bid,uid,A);

      A[BID_POS] = (GenPtr) inactive.number();
      A[BID_POS+1] = (GenPtr) inactive.get_disk();
    }
    else
    {
      std::cerr << "write::current user id is NO_USER.\n";
```

```
    abort();
  }
}
```

This member function allows to write item B to any disk block specified by block identifier bid. We first extract the user identifier uid of item B. If B has no valid user (NO_USER), we are not able to write the block to disk. Otherwise, we set B's internal block identifier to bid and then ask the *external memory manager* to issue a write command. Member function $ext\_mem\_mgr.write\_block(bid, uid, A)$ of class $ext\_memory\_manager$ then uses a write routine of the concrete kernel to write the contents of item B (*i.e.* array A) to the disk block specified by bid. The external memory manager of the concrete kernel is responsible for performing the correctness checks, *i.e.* it checks if block-id bid is valid and if user-id uid is allowed to access the disk block specified by block-id bid. By this mechanism, class $block{<}E{>}$ is connected to the concrete kernel (see the directed association in Figure 3.1 between block and ext_mem_manager). This is a one way connection in such a way that data type $block$ accesses member functions of class $ext\_memory\_manager$ but not the other way around. Member function $read(B\_ID)$ basically works in the same manner.

We conclude the description of the abstract kernel by giving a simple code example to highlight the simplicity of the abstract kernel classes. Our example shows that the external programming paradigms of TPIE can also be easily implemented in LEDA-SM. Our function is a simple scan function. This function scans k blocks and applies a function $f(\ )$ to each item of the block.

40 ⟨ *scan.c* 40⟩≡

```
template<class E> void scan (B_ID start, unsigned int k,
                             void (*f)(E&), U_ID uid)
{
   block<E> B(uid);
   for(B_ID i=start;i<start+k;i++)
      {
         B.read(i);
         for(int j=0;j<B.size();j++) f(B[i]);
         B.write(i);
      }
}
```

Our function takes as input the starting block-id $start$ from where we want to scan k blocks, the user uid of these blocks and a pointer to the modification function f. We see in this simple example, that it is not necessary to understand how the concrete kernel performs the read or write accesses to the disk as the abstract kernel completely relieves the programmer from that task. A general scan-routine that also takes care of the number of links in a block and that is also able to scan linked disk blocks can be easily derived from the simple routine above.

3.3 The concrete kernel

The concrete kernel is responsible for performing I/Os and managing disk space and users in secondary memory and consists of classes $ext\_memory\_manager$, ext_disk, $ext\_freelist$ and $name\_server$. Figure 3.2 shows the kernel layout in UML.

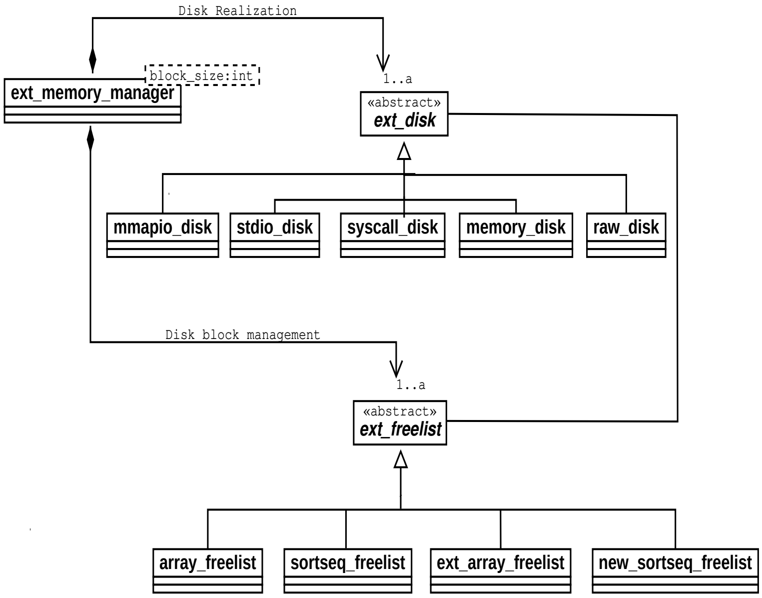


Figure 3.2: *UML specification of the concrete kernel without class name_server.*

We will now discuss in more detail the functionality and implementation of the classes of the concrete kernel.

3.3.1 Class $name\_server$

Class $name\_server$ is responsible for managing user identifiers. This class allows to allocate a new user identifier or to free a formerly used user identifier. The class is implemented by a variable max_name of type int and a LEDA priority queue having priority type int and information type $char$. The information type is actually useless, therefore we use type $char$ to waste the least possible extra space. At time of creation of class $name\_server$, max_name is zero and pq is empty. pq is used to store freed user-ids. As long as pq is empty, a new user-id is allocated by returning max_name and incrementing it. If pq is not empty, it returns its minimal key as newly allocated user-id. The constructor of class $name\_server$ works in such a manner that it is not possible to create more than one instance of class $name\_server$ (see Appendix 3.3.2 for this implementation trick).

3.3.2 Class *ext_memory_manager*

Class *ext_memory_manager* is the central class of the concrete kernel. It manages the library's access to secondary storage. Therefore, only one instance of this class exists at a time. We will later see in this section how the implementation can guarantee that only one instance exists. In detail, the class is responsible for:

- creation of secondary storage
  At the time of creation the class parses a configuration file that contains the number of disks, the names (file or device names) of the disks, and the size of each disk in blocks. The class checks this information for correctness and then uses class *ext_disk* to create the secondary memory, *i.e.* it opens the files or devices and sets up the disk block management (for details see Section 3.3.3).

- management of occupied and free disk blocks
  At startup-time, all disk blocks on each disk are free, *i.e.* they don't belong to a specific user. It is possible to allocate block(s) for a specific user and to free block(s). These requests are outreached to class *ext_freelist*. This class is responsible for the actual management of disk blocks of each disk (see in detail Section 3.3.4).

- transaction of physical I/Os
  Physical I/O requests to specific disk block locations are first checked for correctness and then outreached to class *ext_disk* which does the actual physical I/O. The correctness check contains out-of-bound checks for the block identifiers as well as user checks. The initial correctness check setting allows to read every disk block but only to write to disk blocks with matching user identifier.

Besides the functionality of initiating I/Os and managing disk space, the concrete kernel provides other functionalities:

- Counting of I/Os
  It is possible to count read and write accesses for each disk. Furthermore it allows to count bulk I/Os, the size of a bulk I/O can be set by the external memory manager (see Section 2.3.2 for a description of bulk I/Os). The routines count "logical" I/Os, *i.e.* I/O requests that are scheduled by the external memory manager. It can happen that the file system does not have to perform the I/O as the data is already buffered.

- logging
  The system is able to log status messages of the various tasks of the concrete kernel into a log-file. The log-file can be found under `‘‘/tmp/LEDA-SM-LOG’’`.

- Kernel configuration
  The kernel is configured at program startup time. The constructor of class *ext_memory_manager* reads a configuration file (named `.config_leda_sm`). This file specifies the number of disks, name of the disk files, size of the disks files, used I/O-implementation (implementation class for class *ext_disk*) and freelist implementation. The last section of the configuration file specifies if kernel recovery is enabled and if the kernel should read the recovery file during startup (see Section 3.4 for details).

- Kernel and data structure recovery
  Secondary memory is provided by files of the operating system. In the earlier versions of LEDA-SM, these files were deleted when computation ended. At start-time, the files were created from scratch again. Although data structures and algorithms had the functionality of writing their data to output files, we missed the important feature of recovering data structures and algorithms from previously used disk files. This functionality avoids to write the contents of data structures to intermediate output file and reread these files later, and additionally, it allows to stop the computation and restart it at any later point in time (so-called *checkpointing*). The recovery mechanism consists of two parts: we first save all kernel data structures that manage disk blocks and users (*kernel recovery*) and second, we save the necessary information to recover a data structure from the disk file. The recovery mechanism is in detail described in Section 3.4.

Class $ext\_memory\_manager$ is implemented in such a way that it does not use any classes of the abstract kernel. By this, it is possible to extend or change the abstract kernel without touching the concrete kernel. We now describe the implementation of class $ext\_memory\_manager$.

43 ⟨ *ext_memory_manager.h* 43⟩≡ 44 ▷

```
template<int bz>
class ext_memory_manager
{
  ext_freelist **freelists;
  ext_disk     **disks;
  int          NUM_OF_DISKS;
  [...]

public:

  [ public member functions ]
};
```

Class $ext\_memory\_manager$ is parameterized in the disk block size bz. The following private member variables implement the logical disks and the disk block management [3]:
Variable NUM_OF_DISKS stores the number of realized logical disks. Pointer variable $*disks$ points to an array of type $ext\_disk\ *$ of size NUM_OF_DISKS; pointer variable $*freelists$ points to an array of size NUM_OF_DISKS of type $ext\_freelist*$. The first array realizes the supported disks and the second array manages the disk blocks for each of the disks. The space for both arrays is allocated in the constructor of class $ext\_memory\_manager$. We now describe the most important member functions of class $ext\_memory\_manager$.

---

[3] We omit some variables to simplify the description. Most of them are variables that count random or bulk I/Os. A full code description is given in the appendix.

44 ⟨ *ext_memory_manager.h* 43⟩+≡ ◁43 45a▷

```
template<int bz> B_ID
ext_memory_manager<bz>::new_block(U_ID uid, int D)
{
  int i,block_num;

  if (D < -1 || D >= NUM_OF_DISKS)
    {
      std::cerr << "new_block::disk out of bound: " << D
                << std::endl;
      abort();
    }
  if(D == -1) // memory manager chooses disk
  {
    for(i=0;i<NUM_OF_DISKS;i++)
    {
      if ((block_num=freelists[i]->new_blocks(uid,1))>=0)
        break;
    }
    if (i==NUM_OF_DISKS)
    {
      std::cerr << "new_block::no free block on any disk!\n";
      abort();
    }
    D=i;
  }
  else
  {
    if((block_num=freelists[D]->new_blocks(uid,1))<0)
    {
      std::cerr << "new_block::no free block on disk " << D
                << std::endl;
      abort();
    }
  }

  return B_ID(block_num,D);
}
```

$new\_block(uid, D)$ tries to allocate a single disk block on disk D for user uid. If $D = -1$, the system can choose an arbitrary disk. The member function first checks, if D is out of bounds. Then, it starts asking each freelist data structure, if there is a free block on its disk (case $D = -1$) or it asks the freelist data structure of the specific disk D. This is done by the call `freelists[i]->new_block(uid,1)` that uses class $ext\_freelist$. The return value of this call is either the allocated disk block number or -1 if no block is free. If the system does not find a free disk block, it issues a log message. The return value of the function is the block-id, consisting of the allocated disk block number and D. Member function $new\_blocks(uid, k, D)$ allocates k consecutive blocks and basically works in the same manner.

45a ⟨ *ext_memory_manager.h* 43⟩+≡ ◃44 47b▹

```
template<int bz>
void ext_memory_manager<bz>::free_block(B_ID bid, U_ID uid)
{
  if (check_bounds(bid,1))
  { std::cerr << "free_block " << bid << "is invalid\n";
    abort();
  }
  if (freelists[bid.get_disk()]->free_blocks(bid.number(),uid))
  {
    std::cerr << "\nfree_block:access to wrong block, "
              << "you are not owner of this block "
              << "request is ignored\n";
    abort();
  }

}
```

Member function $free\_block(bid, uid)$ frees the block-id bid which was previously allocated by user uid. We first check by function $check\_bounds(bid, k)$, if the block-id is valid. If this is the case, we ask class *ext_freelist* to free the block. The call `freelists[..]->free_blocks()` returns the freed disk block number or -1 if it failed. The request can only fail for two reasons: the disk block was already free or allocated by a user different from uid. In both cases we issue an error message and abort.

We now show how to read and write a block to disk.

45b ⟨*ext_memory_manager.h* 45b⟩≡ 47a▹

```
template<int bz>
void ext_memory_manager<bz>::write_block(B_ID bid, U_ID uid,
                                         ext_block B)
{
  if (check_bounds(bid,1))
    {
      std::cerr << " invalid B_ID "<<bid<<" in write_block\n";
      abort();
    }

  if(freelists[bid.get_disk()]->check_owner(bid.number(),uid))
    {

      disks[bid.get_disk()]->write_blocks(bid.number(),B);
      write_count[bid.get_disk()]++;


      if(count_bid[bid.get_disk()]+1 == bid)
        {
```

```
          cons_count1[bid.get_disk()]++;
          if(cons_count1[bid.get_disk()] == bulk_value)
            {
              cons_count1[bid.get_disk()] = 0;
              cons_count[bid.get_disk()]++;
            }
          count_bid[bid.get_disk()] = bid;
        }
      else
        {
          rand_count[bid.get_disk()] +=
               cons_count1[bid.get_disk()]+1;
          cons_count1[bid.get_disk()]=0;
          count_bid[bid.get_disk()] = bid;
        }
    }
  else
    {
      std::cerr << "ext_memory_manager::write_block\n";
      std::cerr << "disk block " << bid
                << " is not owned by user "
                << uid << std::endl;
      abort();
    }
}
```

We first check again if bid is valid. This is done by the private member function $check\_bounds$. In the next step, we ask class $ext\_freelist$ if block-id bid is owned by user uid. If this is not the case, we abort the application, because otherwise we would overwrite data of other users. If all checks passed, we issue the write command by information class ext_disk to write the block (`disks[bid.get_disk()]->write_blocks(bid.number(),B);`). In the rest of the member function $write(bid, uid, B)$ we increment our write counters and try to determine if the I/O belonged to a bulk I/O or not. Member function $read$ works similarly. The functionality of the rest of the member functions of class $ext\_memory\_manager$ is described in its manual in Appendix C.

The implementation has to ensure that only one instance of class $ext\_memory\_manager$ exists at a time. The reason for this is easy to see: if more than one instance of this class exists for a single application and both instances want to manage the same disk resources (same disk files), we can easily get into trouble as the information contained in their freelists is not necessarily identical. Therefore, we need to guarantee that only one instance can exist at a time. This is a little bit tricky if the following situation occurs:
Several C++ files each contain a code line that includes $ext\_memory\_manager.h$. These C++ files are later compiled to different object-files and then linked together.
Normally, the situation is simply solved by putting the unique instance into a library. Unfortunately, this concept does not work for the following reason:
A design goal of LEDA-SM was that it should be possible to change the disk block size (system constant EXT_BLK_SZ) without the need to recompile the library. In-

stead a recompilation step of the application program should suffice. Therefore it was necessary to keep constant *EXT_BLK_SZ* outside of the library code and it was not possible to put the unique instance of class *ext_memory_manager* into the library as it directly depends on *EXT_BLK_SZ*. We have therefore chosen the following method:

47a ⟨*ext_memory_manager.h* 45b⟩+≡ ◁45b

```
template<int bz>
class ext_memory_manager_init
{
  static ext_memory_manager<bz> MM;

  [ private member functions of ext_memory_manager ]
public:
  [ public member functions of ext_memory_manager ]
};
```

We provide a special initialization class *ext_memory_manager_init* that is also parameterized in the constant *bz*. Class *ext_memory_manager_init* has a private static member of type *ext_memory_manager<bz>*. This static member is initialized in *ext_memory_manager.h* using the code line

47b ⟨ *ext_memory_manager.h* 43⟩+≡ ◁45a

```
template<int bz>
ext_memory_manager<bz>
ext_memory_manager_init<bz>::MM=ext_memory_manager<bz>();

static ext_memory_manager_init<DISK_BLOCK_SIZE> ext_mem_mgr;
```

Then, a static object of type *ext_memory_manager_init<bz>* is created. If we now use several instances of file *ext_memory_manager.h*, the template instantiation mechanism of the compiler takes care of having only one unique instance of type *ext_memory_manager*. Although there might be several instances of the initialization class *ext_memory_manager_init<bz>*, each of these instances refers to the unique instance of type *ext_memory_manager<bz>*, thus solving the described problem. This unique instance is called *ext_mem_mgr*.

3.3.3 Class *ext_disk*

Class *ext_disk* implements the logical disk drive and the access (read or write) to it. Class *ext_disk* is a virtual base class from which we derive the actual implementation, *i.e.* class *ext_disk* only describes the functionality while the actual different implementations are encapsulated in different classes (see Figure 3.2). The actual implementation is chosen at the time of creation of class *ext_memory_manager*.

47c ⟨*Class ext_disk* 47c⟩≡

```
class ext_disk
{
public:
```

```
    virtual void open_disk(int num)=0;

    virtual void close_disk()=0;

    virtual int write_blocks(int block_num,ext_block B,
                             int k=1)=0;

    virtual int read_blocks(int block_num,ext_block B,
                            int k=1)=0;

    virtual int read_ahead_block(int block_num,int ahead_num,
                                 ext_block B)=0;

    virtual char* get_disktype()=0;
  };
```

Member function $open\_disk(num)$ creates the disk space for logical disk num, $0 \leq num < NUM\_OF\_DISKS$. Each disk consists of a fixed number of blocks of size EXT_BLK_SZ items of type $GenPtr$. It uses information from the external memory manager, *i.e.* the file name of the disk and the number of blocks of the disk. Member function $close\_disk()$ closes the disk files, *i.e.* it disconnects the disks from the system. Member functions $read\_blocks()$ and $write\_blocks()$ perform the actual physical I/Os. They read/write k consecutive blocks starting from disk block number $block\_num$. Member function $read\_ahead\_block()$ reads a single disk block and starts an asynchronous read-ahead of a second disk block. This is not possible in every implementation class, if read-ahead is not available, it performs a normal $read\_block$. At the moment, five different implementation classes, derived from class ext_disk, are available:

- Class $memory\_disk$:
  Class $memory\_disk$ is a realization of disks as a fixed portion of internal memory. Each disk is modeled by an array of GenPtrs, thus no external memory in form of disk space is used. In this implementation, an I/O-operation is nothing else then copying memory regions. This implementation can easily consume a lot of internal memory. Therefore it should only be used for test purposes.

- Class $stdio\_disk$:
  Class $stdio\_disk$ uses standard file I/O (stdio.h) and the file system to implement each disk. An I/O-operation is realized by the use of $fwrite$ and $fread$ (see also Section 2.2.3). The files are written in binary format to reduce their size. All read or write operations of the standard I/O are buffered. Due to the buffering it is possible to save I/Os as the standard I/O library only performs an I/O-operation if the buffer is full or more data has to be written or read. However, buffering is only effective for consecutive operations and can be the bottleneck for random read operations because the buffer is always filled up to its total size. The size of the buffer can be changed, it is normally set to 8 kbytes. We do not change this value and bypass the buffering as our block size is at least 8 kbytes.

- Class $syscall\_disk$:
  Class $syscall\_disk$ uses the file system and the standard file I/O system calls $open$, $read$, $write$, $lseek$ and $close$ to access each file (see Section 2.2.3). We do not use synchronized I/O operations to speed up $read$ and $write$. In the synchronized I/O mode, each operation will wait for both the file data and file status to be physically updated on the disk. This leads to a dramatic slowdown in the I/O performance. Class $syscall\_disk$ is portable to almost every platform because it just uses the standard file system calls that should be implemented in every operating system. It is also possible to use class $syscall\_disk$ for low-level raw disk device access if the operating systems provides the corresponding device driver interfaces (see *raw_disk*).

- Class $mmapio\_disk$:
  Class $mmapio\_disk$ uses the file system and memory mapped I/O (see Section 2.2.3). Memory mapped I/O maps a file on disk into a buffer in memory, so that when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. The advantage of memory mapped I/O is that the operating system is always doing the I/O directly from or to the buffer in memory, all other file access methods like system calls and standard file I/O first copy the data into a kernel buffer and then perform the I/Os on the kernel buffer. We always map exactly one disk block (or k if we perform consecutive operations). Mapping the whole file is useless as for bigger files this can easily exceed the physical main memory so that swap space must be used to establish the mapping (which is nonsense as swap space is disk space). Data is transfered using a low-level system memory copy routine called $memcpy$. Memory mapped I/O should be available on almost every UNIX system.

- Class aio_disk:
  Class aio_disk uses the file system and the $asynchronous\ I/O\ library\ (aio)$. This library allows to perform asynchronous read and write requests. Besides the standard file I/O system calls, this library supports asynchronous read calls (`aioread`). The function call returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately). Notification of the completion of an asynchronous I/O operation may be obtained synchronously through the `aiowait` [4]. function, or asynchronously by installing a signal handler for the `SIGIO` signal. The asynchronous I/O library is not available on every system, therefore it is not included in the standard LEDA-SM package.

- Class raw_disk:
  Class raw_disk uses standardized file system system calls and the operating system's special files [5] to access the raw disk device. This allows to bypass the buffering mechanisms of the file system, and therefore allows to measure real disk performance and to save internal memory. Furthermore, logically consecutive disk blocks are also physically consecutive on raw devices. Raw device

---

[4] see Unix manual page `aiowat(3)`

[5] /dev/rdsk for Solaris operating systems.

disk access via special files is not available on every operating system and is therefore not portable across several platforms.

3.3.4 Class $ext\_freelist$

Freelist data structures (we call them *allocators* for the remainder of this section) have a long tradition in the area of *dynamic storage allocation* or *"heap" storage allocation*. Their task is to keep track of which parts of the memory (in our specific case disk memory) are in use, and which parts are free. The design goal is to minimize wasted space without undue time cost, or vice versa. All allocators have to deal with the problem of *fragmentation*. Fragmentation is the inability to reuse memory that is free. Fragmentation is classed as *external* or *internal* [Ran69]. External fragmentation arises when free blocks of memory are available for allocation, but cannot be used to hold objects of the sizes actually requested by a program. Internal fragmentation arises when a large enough block is allocated to hold an object, but there is a poor fit because the block is larger than needed. Two main techniques are used to combat fragmentation: *splitting* and *coalescing* of free blocks. Internal fragmentation often occurs because the allocator does not split larger blocks. This may be intended to speed up the allocation or it is done as a strategy to avoid external fragmentation; splitting might produce small blocks, if these blocks are not likely to be requested they block the freelist and might increase the search time in the freelist. External fragmentation is combated by coalescing adjacent free blocks (neighboring free blocks in address order), thus combining them into larger blocks that can be used to satisfy requests for larger blocks. When a request for a memory piece of size p is processed, different strategies can be used to choose among all free blocks. The most widely used strategies are *best-fit* or *first-fit*. In the former case, the algorithm chooses the smallest consecutive area of m blocks such that $m \geq n$. In the second case the algorithm chooses the first consecutive area that has more than n blocks. There are a lot of other strategies such a next-fit, worst-fit, half-fit, just to cite a few, but we do not discuss them here in detail. We refer to [WJNB95] for a nice overview on the work of dynamic storage allocation. All in all, there is no clear winner among the strategies as their performance also depends on the specific pattern of allocation and deallocation requests.

Several different data structures for allocators exist. Most of the data structures are implemented using space in the free blocks. We will avoid this as we do not want to perform I/Os for allocation tasks.

A classical data structure are *doubly linked lists of free blocks*. For this implementation, it is important how to return free blocks to the list. Several variants exist, among them are LIFO (last-in-first out) which inserts the block at the front of the list, FIFO (first-in-first out) which pushes the freed block to the end of the list, and address-ordered fit which keeps the block sorted by starting address and inserts the freed block in the proper location. Search strategy is normally first-fit (or Knuth's variant next-fit) as best-fit must search the list exhaustively.

Segregated freelists use an array of free lists, where each list holds free blocks of a particular size [Com64]. When a block of memory is freed, it is simply pushed onto the free list for that size. These allocators allow splitting and coalescing of blocks. A common variation is to use size classes for the lists, for example powers of two. Requests for a given size can be rounded up to the nearest size class and then be satisfied by any block in that list. The search strategy is normally best-fit.

Buddy systems [Kno65, PN77] are a variant of segregated lists. At the beginning, the heap is conceptually divided in two large areas, those areas are further split into two areas, and so on. This forms a binary tree. This tree is used to constrain where objects are located, what their allowable sizes are and how they may be coalesced into new larger areas. A classical example are binary buddies where the binary representation of the address space is used for the partitioning. In this scheme all sizes are powers of two and each size is divided into two equal parts. Each size has its unique buddy and coalescing can only be done between these two buddies. For example, if we have block $< a_n, a_{n-1}, \dots, a_k, 0, 0, \dots 0 >$ [6], its unique buddy of size 2^k is $< a_n, a_{n-1}, \dots, a_k, 1, 0 \dots 0 >$. Thus, splitting and coalescing can be computed in a simple way by bitwise logical operations. The search actually implements the best-fit strategy as we round the request size to the nearest power of two. The main problem with binary buddies is the internal fragmentation as arbitrary splitting of blocks is not allowed. Variants with closer size class spacing exist, for example *Fibonacci buddies* [Hir73].

Indexed fits are a quite general class that uses indexing data structures to manage the free blocks. It is possible to index blocks by exactly the characteristics of interest to the desired policy and to support efficient searching according to those characteristics. The indexing data structure can be embedded in the free blocks themselves or can be kept separately. Supported strategies can be best-fit, all variants of first-fit (FIFO, LIFO, address-ordered) and others. The index data structure can be totally or partially ordered. For dynamic storage allocation of internal memory, a drawback of indexed fits is the fact that search time is normally logarithmic in the number of the free memory blocks.

Bitmapped fits use a bitmap to record which parts of the memory are free and which not. We reserve a bit for each block of the memory (traditionally for each word of the memory). The bit is one if the block is free, and zero otherwise. This bitmap is stored as a separate data structure. First-fit and best-fit strategies can be easily implemented. To our knowledge, bitmaps have never been used in conventional allocators. The main reason might be the fact that searching is believed to be slow. [WJNB95] proposes methods to speed-up searching by the use of clever implementation techniques. An idea is to use lookup tables to localize the search. Heuristics can be used to decide where to start the search, thus avoiding to scan the whole bitmap and reducing fragmentation.

In the LEDA-SM scenario, several tasks are different from classical in-core memory allocators. First of all, we do not want to interweave the allocator data structure with the free disk space, *i.e.* we are not willing to pay I/Os when allocating disk space. Secondly, all our allocations occur in multiples of disk blocks. Therefore, internal fragmentation does not occur. Thirdly, we have to manage different users, thus it is not enough to mark a disk block used, we also have to remember the user of this block. We first describe the general functionality of our allocator and then turn to the different implementations.

Class *ext_freelist* is responsible for managing free and allocated disk blocks. Class *ext_disk* is implemented as a virtual base class from which we derive the actual imple-

---

[6] $< a_n, a_{n-1}, \dots, a_0 >$ is the binary representation of the number $\sum_{i=0}^{n} a_i \cdot 2^i$.

mentation classes (see Figure 3.2). The actual implementation is chosen at the time of creation of class $ext\_memory\_manager$.

52 ⟨*Class ext_freelist* 52⟩≡

```
class ext_freelist
{
public:

  virtual void init_freelist(int num)=0;

  virtual int new_blocks(U_ID uid,int k=1)=0;

  virtual int free_blocks(int block_num, U_ID uid,
                          int k=1)=0;

  virtual void free_all_blocks(U_ID uid)=0;

  virtual bool check_owner(int block_num, U_ID uid,
                           int k=1)=0;

  virtual int get_blocks_on_disk()=0;

  virtual int get_free_blocks()=0;

  virtual int get_cons_free_blocks()=0;

  virtual int get_used_blocks()=0;

  virtual char* get_freelist_type()=0;

  virtual int size()=0;
};
```

Member function $init\_freelist(num)$ initializes the freelist for disk num, $0 \leq num < NUM\_OF\_DISKS$. Member function $new\_blocks(\ )$ allocates disk blocks, if $k > 1$, the blocks must be consecutive. The return value is the block number of the first allocated block on disk num. Member function $free\_blocks()$ returns previously allocated disk blocks of disk num back to the freelist, and $free\_all\_blocks()$ frees all disk blocks of disk num that were allocated to user uid. Member function $check\_owner()$ is used to check if the disk blocks $block\_num, .., block\_num + k - 1$ are owned by user uid. $get\_blocks\_on\_disk()$ returns the number of disk blocks of this disk, $get\_free\_blocks()$ returns the number of free disk blocks, $get\_cons\_free\_blocks()$ returns the maximum number of consecutive free disk blocks, $get\_used\_blocks()$ returns the number of currently allocated disk blocks, $get\_freelist\_type()$ returns the name of the freelist implementation and $size()$ returns the internal memory space consumption of the freelist.

3.3.4.1 Implementation

We now describe the different implementation classes that are derived from base class $ext\_freelist$:

- Class $array\_freelist$:
  $array\_freelist$ implements a *bitmapped fit*. The search strategy is first-fit. The implementation uses an internal array to manage used and free blocks for a specific disk. Array entry $A[i]$ is either set to NO_USER or to the user uid, owning block i. Let k be the number of requested blocks. If $k = 1$, $new\_blocks()$ runs in $O(1)$ time, otherwise it runs in $O(N)$ time. Operations $free\_all\_blocks()$ and $get\_cons\_free\_blocks$ run in $O(N)$ time, operation $free\_blocks()$ runs in $O(1)$ time if $k = 1$ and in $O(k)$ time otherwise. All other operations run in $O(1)$ time. The space consumption in internal memory is $N * sizeof(U\_ID) + O(1)$ bytes where N is the maximum number of blocks on the disk and $sizeof(U\_ID)$ is the size of a user-id in bytes (4 bytes on a 32-bit machine). Thus, for a 9 GB disk with 32 kbytes disk block size, the data structure needs a little bit more than one Mbytes of internal memory.

- Class $ext\_array\_freelist$:
  This implementation is similar to the previous implementation but in order to save space in internal memory, the array is kept in a temporary file of the file system. We keep a buffer of fixed size (ORG_SZ) in internal memory. In case of a hit in the buffer, we can answer the requests in main memory, otherwise we load the next ORG_SZ array elements and possibly write back the old buffer contents. The implementation uses buffered standard I/O-routines for reading and writing buffers. The internal memory space consumption is minimal, only $O(ORG\_SZ)$ bytes are used in internal memory. The disadvantage is that we now have to perform I/Os for managing the freelist. In detail, new_blocks runs in $O(1)$ I/Os if $k = 1$ and in $O(N/ORG\_SZ)$ I/Os otherwise. Operation $free\_all\_blocks$ and $get\_cons\_free\_blocks$ runs in $O(N/ORG\_SZ)$ I/Os, operation $free\_blocks$ runs in $O(1)$ I/Os if $k = 1$ and in $O(k/ORG\_SZ)$ I/Os otherwise. All other operations run in $O(1)$ time with no I/Os.

- class $new\_sortseq\_freelist$:
  $new\_sortseq\_freelist$ is an *indexed fit*. It implements the best-fit strategy together with general splitting and coalescing of blocks. The allocator consists of a freelist data structure and a usedlist data structure. The freelist data structure is a variation of LEDA's sorted sequence (*multi_sortseq*). Items of this data structure consist of a key which is the size of the free block and an information which is the starting address of the free block. We start with one big block that contains the whole disk space. Multi_sortseqs are totally ordered by key value and allow to maintain multiple entries with the same key value (this is not the case for LEDA data type $sortseq$ where only distinct keys are allowed) [7]. Used disk blocks are stored in the usedlist which is of type $map< U\_ID, sortseq<int, int> >$. LEDA's parameterized data type $map<I, E>$ implements hashing with chaining and table doubling. Type I is the index set (the universe) and type E is the information that we associate with an item of type I. We could have chosen an array or a list instead of type map, the idea was that we do not assume to manage many different users and therefore type map is faster and more space-efficient. Type E of the map is a sorted sequence. The key data tpye I of the sorted se-

---

[7] Multi_sortseq and sortseq are implemented by skiplists [Pug90].

quence is the start address of the used block and the information is the size of the used block. Coalescing of used blocks is easy as the sorted sequence is address ordered. We now describe member functions new_block and $free\_blocks$. When there is a request of user uid for k new blocks, the following happens: We first check if the total number of free blocks in the freelist is at least k. This is done by looking at a counter variable that dynamically maintains the actual number of free disk blocks. If enough free blocks are available, we search the multi_sortseq for key k, getting either an exact match or a block of size $k_1, k_1 > k$, where k_1 is the smallest available block size which is bigger than k (best-fit). If the search results in a block being bigger than k, the block is split into two blocks, one of size k which is the request size and one of size $k_1 - k$. If we do not find such a block, we do not have contiguous space of the requested size in the multi_sortseq. We then start coalescing blocks in the multi_sortseq and then try to satisfy the request again (*coalescing on demand*). Coalescing is performed by sorting the items of the multi_sortseq according to the information (the start address of the free block) and combining contiguous free blocks. If the request cannot be satisfied after the coalescing step, we cannot proceed due to external fragmentation of our data structure. Return value of an answered request is the start address of the newly allocated block. After the block was allocated, we have to change the usedlist. We hand over the triple `(start address, k ,uid)` to the usedlist. There, the tuple `(start address, size)` is inserted into the sorted sequence for the user, specified by uid ($map[uid]$). If there are neighboring blocks, they are automatically coalesced.

Operation $free\_blocks(block\_num, uid, k)$ is simple: we first check if user uid owns the blocks $block\_num, \ldots\ block\_num + k - 1$. This is done by a simple search in its sorted sequence ($map[uid]$) of the usedlist. If the user does not own the blocks, we issue an error message and abort the application. If the user owns the blocks, we delete the blocks $block\_num, \ldots\ block\_num + k - 1$ in the sorted sequence $map[uid]$ and insert the item $(k, block\_num)$ into the freelist data structure. $free\_all\_blocks(uid)$ works similarly. We simply delete all items of $map[uid]$ and insert them into the freelist. $free\_blocks$ and $free\_all\_blocks$ increase the size of the freelist data structure ($multi\_sortseq$). As there is no automatic coalescing of neighboring blocks in the $multi\_sortseq$, we use the following approach: whenever x items have been inserted into the $multi\_sortseq$ due to calls to $free\_blocks$ or $free\_all\_blocks$, we start to coalesce neighboring blocks. This is done by sorting as described above. Items in this context are items of the usedlist and not single free blocks. An item can be a consecutive region of blocks as the usedlist automatically coalesces neighboring blocks. We note that this *preemptive coalescing* of the freelist data structure does not necessarily reduce the space (number of items) of the the $multi\_sortseq$.

We now analyze the time bounds: let N be the total number of disk blocks, F be the number of items in the freelist structure and U be the number of items in the usedlist structure. We note that $F + U \leq N$ and that in most cases the sum of F and U is much smaller than N due to coalescing of neighboring blocks. However, in the worst case, $F = U = N/2$. This situation occurs if each occupied disk block is followed by a single free disk block. All insert, search, and delete operations on the *multi_sortseq* or the *sortseqs* of the *usedlist* have time bounds logarithmic in the number of items of the structure. Note that most of the functions have to access the map first. Access to the map runs in $O(1)$ expected time or in average $O(\log N/\log\log N)$ worst case time. Thus, the average worst case time is bounded above by $O(\log N)$. Compacting the *multi_sortseq* runs in $O(F \log F)$ time which is again $O(N \log N)$ in the worst case. Operation *get_cons_free_blocks* runs in $O(\log F)$ time and all other operations run in $O(1)$ time. The expected size of the data structure is $2 * (76/3 + 8) * (F + U) + O(1) = 67 * (F + U) + O(1)$ bytes [MN99].

- Class *sortseq_freelist*:
  sortseq_freelist is also an indexed fit. It implements the address ordered first-fit strategy. The allocator consists again of a freelist data structure and an usedlist data structure. Both data structures have the same meaning as above and are again called *usedlist* and *freelist*. They are implemented by the LEDA data type *sorted sequence*. The key type of the *freelist* is the block number, the information type is the number of consecutively free blocks starting at this block number. In the *usedlist*, the information type additionally stores the user identifier. Both data structures perform automatic coalescing of neighboring blocks, *i.e.* if there is an item (x, u, k), where x is the block number, u is the user-id, and k is the number of allocated blocks, we look at both the predecessor and successor item. If this item is of the form $(x - k, u, z)$ (resp. $(x + k, u, z)$) we merge the items together thus producing an entry of the form $(x - k, u, z + k)$ (resp. $(x, u, z + k)$). This task is easy, as both data structures are address-ordered. The worst case situation occurs, if two different users alternately allocate single blocks. In this situation, joining neighboring blocks is not possible and the space consumption gets linear in the total number of requested blocks. Let us now discuss the time bounds. Let again N be the total number of disk blocks, F be the number of items in the freelist structure and U be the number of items in the usedlist structure. Let k be the number of requested blocks. Operation *new_blocks*() runs in $O(\log F + \log U)$ time if $k = 1$, and in $O(F + \log U)$ time otherwise. Operation *free_blocks*() runs in $O(\log F + \log U)$ time. Operation *free_all_blocks*(uid) runs in $O(U + w \log F)$ time, where w is the items in the usedlist that are associated with user uid (items (x, uid, k)). Operation *check_owner*() runs in $O(\log U)$ time, operation *get_cons_free_blocks*() runs in $O(F)$ time, all other operations run in $O(1)$ time. We again note that in the worst case, $F = U = N/2$. A note to the time bounds: *new_blocks* does not necessarily run in $O(F + \log U)$ time if $k > 1$. This time bound is due to the first-fit strategy where we start at the beginning of the freelist, check if we can satisfy the request, and if this is not the case, look at the successor item. In the worst-case, we always have to search to the end of the freelist. Experiments [WJNB95] have shown that this situation does not occur often in practice, and that in normal situations, one must only inspect a constant number of items. Sorted sequences are implemented by skiplists, therefore the expected space requirement is $(76/3+8)*F+(76/3+12)*U+O(1) = 33.3*F+37.3*U+O(1)$ bytes [MN99].

Clearly, the ideal approach minimizes both the internal space consumption and the access time to the allocator data structure. Our two bitmapped fits have a running time linear in the number of allocated disk blocks which is bad for large consecutive requests. Additionally, the space cost is only acceptable, if we either have large main memory or if we are willing to perform I/Os to access the allocator data structure (see *ext_array_freelist*). Buddy systems could improve the access time, but we think that it is not acceptable to afford internal fragmentation and waste disk space. Indexed fits seem to be the most compromising allocator data structures for secondary memory. The running time is logarithmic in the number of disk blocks (best-fit, see *new_sortseq_freelist*) and still reasonable for the first-fit approach as in most practical cases, one does not have to inspect all items of the freelist data structure during a search. Although the space consumption per item is high (≈ 70 bytes), the data structure does not consume a lot of space due to coalescing of neighboring blocks in both the *freelist* and the *usedlist* structure. We also do not expect to have a lot of different users and one can assume that most of the requests of a user go to consecutive disk block locations. For all the experiments that we performed, our indexed fits consumed very little space as most of the allocation requests went to consecutive disk block locations and the coalescing compacted the data structure.

We now proceed in the kernel description by explaining the system startup, system configuration and system recovery.

3.4 System startup

The system starts up by creating the unique instance of class *ext_memory_manager*. At the time of creation, the following happens:
the constructor tries to open a system configuration file named `.config_leda-sm` which must be located in the current working directory. This configuration files contains a number of setup parameters (see Table 3.2). It defines the number of disks, the locations of the disks (disk file names or device names), and the size of each disk measured in blocks of size *EXT_BLK_SZ* `GenPtrs`. Furthermore it defines the implementation for class *ext_disk* (the value can be chosen out of the implementation classes of Section 3.3.3) and for class *ext_freelist* (the value can be chosen out of the implementation classes of Section 3.3.4).
The last entries specify the recovery behavior of the kernel. This part of the config file is two-divided into recovery behavior at system startup (in the presence of a recovery file) and generation of recovery information at program end. If the configuration file is not of the form as described in Table 3.2, the system halts immediately. In the absence of a configuration file, the system tries to start with predefined values, *i.e.* one disk file named `/tmp/disk0` of size 5000 blocks, `stdio_disk` as I/O implementation, `sortseq_freelist` as freelist implementation, and no recovery. After parsing the configuration file, a configuration check is executed. It checks if it is possible to create the disk files on the file system or to open the device files. It furthermore checks if the predefined space for each disk is available by examining either the file system, where the disk files are located, or the device files if raw disk device access is used. This check can fail for several reasons:

1. File creation error or disk device access error

number of disks	**integer**
blocks per disk	number of disks blocks, one **integer** one per line
disk names	number of disks **filenames**, one per line
I/O implementation	out of **stdio_disk, syscall_disk, memory_disk,mmapio_disk, raw_disk**
freelist implementation	out of **array_freelist, sortseq_freelist, new_sortseq_freelist, ext_array_freelist**
recover to	either **no** or **filename**
recover from	either **no** or **filename**

Table 3.2: *LEDA-SM config file. The value of the first column is the name of the parameter, the description for this parameter is in the second column. The config file is line-oriented, each parameter is separated by a new line.*

The disk files cannot be created because the directory permissions are wrong or the disk device files cannot be opened [8]. In this case, the system produces an error message and aborts the application.

2. The requested disk space is not available
   The total sum of required disk space is larger than the available space on either the file system or on the raw disk device. The system then calculates the maximum available space, recalculates and downgrades the disk space for each disk, and then informs the user that he requested more disk space then available and that a downgrade was necessary.

3. Check not possible or faultiness
   The checks that are performed rely on correct file system information or raw-device information. On some platforms the information is either quite useless or simply wrong. File systems for example express the number of free blocks in their own block size. Sometimes the system reports that this block size is zero. Low-level disk device check is device driver dependent. For some computing platforms such checks are not available at the moment. If the system decides that the check did not pass or was not possible, it prints a status message and assumes that the original values as given in the configuration file are “hopefully” correct.

After the system has checked the configuration, it sets up the disks and the disk block management structures. This is done by the following code lines:

---

[8] One needs root-privileges to change the raw disk device permissions as Solaris assumes that there is a file system on each disk.

57 ⟨*ext_memory_manager<bz>::ext_memory_manager()* 57⟩≡

```
for(j=0;j<NUM_OF_DISKS;j++)
{
  disks[j] = new impl_disk;
  freelists[j] = new impl_freelist;
}
```

where `impl_disk` ist out of `memory_disk, stdio_disk, syscall_disk, mmapio_disk, raw_disk` and `impl_freelist` is out of `array_freelist, sortseq_freelist ext_array_freelist,new_sortseq_freelist`.

Note that this works as all implementation classes are derived from the virtual base classes *ext_disk* or *ext_freelist* and therefore it is possible to convert objects of the base class to objects of the implementation class. After executing the constructor for the unique instance *ext_mem_mgr* the system starts with the first line of `main()`.

Recovery Mode. If the user has asked to generate recovery information at the end of the computation, the constructor informs the destructor of class *ext_memory_manager*, to not delete the disk files at the end (this is only done if the disk files are files on the file system) and it informs the destructor to save the information stored in the freelists.

58 ⟨*ext_memory_manager<bz>:: ext_memory_manager()* 58⟩≡

```
template<int bz>
ext_memory_manager<bz>::~ext_memory_manager()
{
  int i;
  if(recover_info==false)
    {
      for(i=0;i<NUM_OF_DISKS;i++)
        disks[i]->close_disk();
    }
  else
    {
      write_recovery_information(recover_name);
    }
}
```

The member function *write_recovery_information* of class *ext_memory_manager* then writes the necessary kernel recovery information to a file, specified in the config file (entry `recover to`). This information consists of the information stored in the config file, *i.e.* the number of disks, the disk file names, the size of each disk file, the I/O and the freelist implementation), of the information stored in each of the freelists plus the used user-ids that are maintained by class *name_server*.

Recovery startup. If recovery files are present at system startup and the configuration files asks to use the recovery files, the system does a recovery startup. The process is quite the same as a normal system startup besides the fact that:

- no disk files are created, instead the old disk files are used;
- the information of the freelists are restored by using the recovery file;
- the $name\_server$ reinitializes its user id management structures to keep track of already used user ids.

To recover a specific data structure, one has to additionally save some recovery information. We will see an example in the next section.

3.5 A simple example

In this section we show the simplicity of designing data structures using LEDA-SM's kernel concept. Our application is simple, hence it allows to show the basic features of data structure and algorithm design using LEDA and LEDA-SM. In the following, we describe data structure ext_stack<E> which is a generalization of data type $stack$ towards secondary memory.
A external memory stack S for elements of type E (ext_stack<E>) is realized by an internal array A of $2a$ blocks of type $block$<E> and a linear list of disk blocks. Each block in A can hold blk_sz elements, *i.e* A can hold up to $2a \cdot blk\_sz$ elements. We may view A as a one-dimensional array of elements of type E. The slots 0 to top of this array are used to store elements of S with top designating the top of the stack. The older elements of S, *i.e.* the ones that do not fit into A, reside on disk. We use bid to store the block identifier of the elements moved to disk most recently. Each disk block stores one link of type B_ID, used to point to the predecessor disk block, and blk_sz items of type E. The number of elements stored on disk is always a multiple of $a \cdot blk\_sz$.

59 ⟨*ext_stack* 59⟩≡ 60a ▷

```
template <class E>
class ext_stack
{
  array< block<E> > A;
  int top_cnt, a_sz, s_sz, blk_sz;
  B_ID bid;

public:

ext_stack(int a = 1);
void push(E x);
E pop();
E top();
int size() { return s_sz; };
void clear();
~ext_stack();
void read_recovery_information(char *name);
void write_recovery_information(char *name);
};
```

We next discuss the implementation of the operations $push$ and pop. We denote by $a\_sz = 2a$ the size of array A. A push operation $S.push(x)$ writes x into the location $top + 1$ of A except if A is full. If A is full ($top\_cnt == a\_sz * blk\_sz - 1$), the first half of A is moved to disk, the second half of A is moved to the first half, and x is written to the first slot in the second half.

60a ⟨*ext_stack* 59⟩+≡ ◁59 60b▷

```
template<class E>
void ext_stack<E>::push(E x)
{
  int i;
  if (top_cnt ==  a_sz*blk_sz - 1)
  {
    A[0](0) = bid;
    bid = ext_mem_mgr.new_blocks(myid,a_sz/2);
    block<E>::write_array(A,bid,a_sz/2);
    for(i=0;i<a_sz/2;i++)
      A[i] = A[i+a_sz/2];
    top_cnt = (a_sz/2)*blk_sz-1;
  }
  top_cnt++;
  A[top_cnt/blk_sz][top_cnt%blk_sz] = x;
  s_sz++;
}
```

The interesting case of $push$ is the one where we have to write the first half of A to disk. In this step we have to do the following: we must reserve $a = a\_sz/2$ disk blocks on disk and we must add the first a blocks of array A to the linked list of blocks on disk. The blocks are linked by using the entry of type B_ID of class $block$ (see Section 3.2) and the block least recently written is identified by block identifier bid. The command $A[0](0) = bid$ creates a backwards linked list of disk blocks which we use during pop-operations later. We then use the kernel to allocate a consecutive free disk blocks by the command $ext\_mem\_mgr.new\_blocks$. The return value is the first allocated block identifier. The first half of array A is written to disk by calling $write\_array$ of class $block$ which tells the kernel to initiate the necessary physical I/Os. In the next step, we copy the last a blocks of A to the first a blocks and reset top_cnt. Now the normal push can continue by copying element x to its correct location inside A.

A pop operation $S.pop(\,)$ is also trivial to implement. We read the element in slot top except if A is empty. If A is empty and there are elements residing on disk we move $a \cdot blk\_sz$ elements from disk into the first half of A.

60b ⟨*ext_stack* 59⟩+≡ ◁60a

```
template<class E>
E ext_stack<E>::pop()
{
  if (top_cnt == -1 && s_sz > 0)
  {
    B_ID oldbid = bid;
    block<E>::read_array(A,oldbid,a_sz/2);
```

```
    bid = A[0](0);
    top_cnt = (a_sz/2)*blk_sz - 1;
    ext_mem_mgr.free_blocks(oldbid,myid,a_sz/2);
  }
  s_sz--;
  top_cnt--;
  return
    A[(top_cnt+1)/blk_sz][(top_cnt+1)%blk_sz];
}
```

If array A is empty ($\mathit{top\_cnt} = -1$) we load a blocks from disk into the first a array positions of A by calling $\mathit{read\_array}$. These disk blocks are identified by bid. We then restore the invariant that block identifier bid stores the block identifier of the blocks least recently written to disk. As the disk blocks are linked backwards, we can retrieve this block identifier from the first entry of the array of block identifiers of the first loaded disk block ($A[0](0)$). Array A now stores $a \cdot \mathit{blk\_sz}$ data items of type E. Variable $\mathit{top\_cnt}$ is reset to this value. The just loaded disk blocks are now stored internally, therefore there is no reason to keep them again on disk. These disk blocks are freed by calling the kernel routine $\mathit{ext\_mem\_mgr.free\_blocks}$. Return value of operation pop is the top most element of A.

Operations push and pop move a blocks at a time. As the read and write requests for the a blocks always target consecutive disk locations, we can choose a in such a way that it maximizes disk-to-host throughput rate. After the movement, A is half-full and hence there are no I/Os for the next $a \cdot \mathit{blk\_sz}$ stack operations. Thus, the amortized number of I/Os per operations is $1/\mathit{blk\_sz}$, which is optimal. Stacks with fewer than $2a \cdot \mathit{blk\_sz}$ elements are managed in-core.

We described in the previous section how the kernel saves recovery information. Additionally, each data structure has to store some information which is necessary to set up the data structure after a kernel recovery startup. For the stack, it is necessary to know the block-id of the last disk block that was written to disk as all other blocks are linked together. We additionally need the user-id, the contents of the array A, and the variables $\mathit{a\_size}$, $\mathit{top\_cnt}$, $\mathit{blk\_sz}$, and $\mathit{s\_size}$.

61 ⟨ *ext_stack* 61⟩≡

```
template<class E>
void ext_stack<E>::write_recovery_information(char *name)
{
  std::ofstream ofile(name);
  recovered=true;
  ofile << "Recovery information\n";
  ofile << "a_size:" << a_size << std::endl;
  ofile << "top_cnt:" << top_cnt << std::endl;
  ofile << "blk_sz:" << blk_size << std::endl;
  ofile << "s_size:" << s_size << std::endl;
  ofile << "myid:" << myid << std::endl;
  ofile << "bid:" << bid << std::endl;
  for(int i=0;i<=top_cnt;i++)
    ofile << A[i/blk_size][i%blk_size] <<std::endl;
```

```
}
```

Operation $write\_recovery\_information$ saves all the necessary information to recover the data structure at any later point. The information is saved into file $name$. Recovery information includes the size of the internal array A (a_size), the pointer to the topmost element of A (top_cnt), the block size (blk_sz), the size of the stack (s_size), the user-id (uid) of the stack, and the block id (bid) of the topmost disk block of the stack. Finally, the first top_cnt elements of array A are saved. Operation $read\_recovery\_information$ now simply consists of parsing that file and setting all class variables according to these values.

3.6 Data Structures and Algorithms

We survey the data structures and algorithms currently available in LEDA-SM. Theoretical I/O-bounds are given in the classical external memory model of [VS94b], where M is the size of the main memory, B is the size of a disk block, and N is the input (see also Section 2.3.2). For the sake of simplicity we assume that D, the number of disks, is equal to one.

Stacks and Queues: External stacks and queues are simply the secondary memory counterpart to the corresponding internal memory data structures. Operations $push$, pop, and $append$ are implemented in optimal $O(1/B)$ amortized I/Os.

Priority Queues: Secondary memory priority queues can be used for large-scale event simulation, in graph algorithms, or for online-sorting. Three different implementations are available. Buffer trees [Arg96] achieve optimal $O((1/B)\log_{M/B}(N/B))$ amortized I/Os for operations `insert` and `delete_min`. Radix heaps are an extension of [AMOT90] towards secondary memory. This integer-based priority queue achieves $O(1/B)$ I/Os for `insert` and $O((1/B)\log_{M/B}(C))$ I/Os for `delete_min` where C is the maximum allowed difference between the last deleted minimum key and the actual keys in the queue. Array heaps [BK98, BCMF99] achieve $O((1/B)\log_{M/B}(N/B))$ amortized I/Os for `insert` and $O(1/B)$ amortized I/Os for `delete_min`. We will do an extensive comparison of internal and external priority queues in Chapter 4.

Arrays: Arrays are a widely used data structure in internal memory. The main drawback of internal-memory arrays is the fact that when used in secondary memory, it is not possible to control the paging. Our external array data structure consists of a consecutive collection of disk blocks and an internal-memory data structure of fixed size that implements a multi-segmented cache. When we access index i of array A, we first look if element $A[i]$ resides in the cache. If this is not the case, we load a block of B elements, including $A[i]$ into the cache. As the cache has a fixed size, we possibly have to evict a block from the cache. This task is done by the page replacement algorithm. Several page-replacement strategies are supported like LRU, random, fixed, etc. The user can also implement his/her own page-replacement strategy. The cache has several advantages: its fixed size allows to control the internal-memory usage of the external array. The blockwise replacement allows to scan an external array in an optimal number of I/Os and furthermore, as the cache is multi-segmented, it is possible to index different regions of the external array by using different segment of the cache for each region.

Sorting: Sorting is implemented by multiway-mergesort [Knu97]. Internal sorting during the run-creation phase is realized by LEDA's quicksort algorithms, which is a fast and generic code-inlined template sorting routine. Sorting N items takes optimal $O((N/B)\log_{M/B}(N/B))$ I/Os.

B-Trees: B-Trees [BM72] are the classical secondary memory online search trees. We use a B^+-implementation and support operations `insert`, `delete`, `delete_min`, and `search` in optimal $O(\log_B(N))$ I/Os. Parent pointers are avoided inside the nodes, instead the implementation stores the parent pointers (which are necessary for the rebalancing process) in a small external stack. We support two algorithms to construct B-trees; one is by repeatedly inserting elements (online construction), the other is by sorting all items and then build the tree bottom-up (offline version). To speed up version one, we implemented some kind of path-caching. The B-tree caches the most frequently used disk pages in a small buffer. This allows to save I/Os: consider the case of doing an online-insertion of an increasingly sorted set. All insertions take place in the leftmost leaf and the B-tree will cache the path to the leftmost leaf.

Suffix arrays and strings: Suffix arrays [MM93] are a full-text indexing data structure for large text strings. We provide several different construction algorithms [CF99] for suffix arrays as well as procedures to perform exact searching, 1- and 2-mismatch searching, and 1- and 2-edit searching routines. Suffix arrays are discussed in detail in Chapter 5.

Matrix operations: We provide matrices with entry type `double`. The operations $+$, $-$, and $*$, as well as scalar operations, are realized for dense matrices within optimal I/O-bounds [UY91].

Graphs and graph algorithms: We offer a data type external graph and simple graph algorithms like depth-first search, topological sorting, and Dijkstra's shortest path computation. The external graph data type uses an adjacency list representation, the implementation is based on external arrays. Nodes and edges are template container classes, auxiliary node or edge information is stored inside the container class. The graph data type is *static*, *i.e.* updates are not possible [9]. Node and edge container need 16 bytes plus the space for the additional information. Graph algorithms are either *semi-external* or *fully external*. The semi-external variant assumes that either node or edge information can be stored in main memory, while the fully-external variant assumes that neither node nor edge information can be completely stored in-core. We will explain the difference between fully-external and semi-external algorithms in more detail in Section 3.8.2 where we discuss depth first search.

3.6.1 Specialties of LEDA-SM applications

Secondary-memory data structures and algorithms use a specific amount of internal memory. LEDA-SM allows the user to control the amount of memory that each data structure and/or algorithm uses. The amount of memory is either specified at the time

---

[9] Updates can be implemented but they are expensive in terms of I/Os.

of construction of the data structure or it is an additional parameter of a function call. If we look at our stack example in Section 3.5, we see that the constructor of data type ext_stack has a parameter a, the number of blocks of size blk_size that are held in internal memory. We therefore immediately know that the internal memory space occupancy is $a \cdot blz\_size + O(1)$ bytes. If we use several data structures together, the total amount of internally used space is the sum of the used internal space per data structure. This method has some drawbacks because we also have to take into account that LEDA's data structures use space. Other libraries use different methods to control the internal amount of space:
TPIE uses an overloaded new operator to automatically keep track of the used space. Unfortunately, this does not work for LEDA-SM as LEDA already overloads new. Additionally, this method is not able to count non-dynamically allocated space, *i.e.* C-style arrays. Therefore, both methods are not accurate. The best way is always to use system commands like `top` to check that the application does not use too much memory. However, using mechanisms like the one of LEDA-SM or that of TPIE allows to find useful start settings.

3.7 Low-level LEDA-SM benchmarks

In a first experiment we want to analyze the speed of LEDA-SM's kernel. Our simple test programs write and read consecutive blocks, or write and read random blocks to disk. We test all different I/O methods, which are `syscall`, `stdio`, `mmap_io`, and raw device access. The tests are performed on a SUN UltraSparc-1 with a single 143 MHz processor and a local 9 GB SCSI hard disk running Solaris-2.6 as operating system. The specifications of the disk vendor are given in Appendix A. In a first step, the blocks are written to disk and in a second step they are read back. To prevent the operating system from buffering the I/O operations after the write step, we clean the memory before we proceed with the read step. Our test measures the user time (also called CPU time), the system time, and the total time.

Table 3.3 compares sequential and random I/O performance of LEDA-SM's kernel using `stdio` as I/O method. Sequential write is 2.2 times faster than random write. Sequential read is 4 times faster than random read. The data transfer rate for sequential write is approximately 4722 kbytes and 5723 kbytes for sequential read. The transfer rate for random operations drops to 2153 kbytes for write (1437 kbytes for read).

Table 3.4 compares sequential and random I/O performance of LEDA-SM's kernel using `syscall` as I/O method. Sequential write is 2.31 times faster than random write. Sequential read is 4 times faster than random read. The data transfer rate for sequential write is approximately 5000 kbytes and 5890 kbytes for sequential read. The transfer rate for random operations drops to 2162 kbytes for write (1445 kbytes for read).

Table 3.5 compares sequential and random I/O performance of LEDA-SM's kernel using `mmap` as I/O method. Sequential write is 4.27 times faster than random write. Sequential read is 2.07 times faster than random read. The data transfer rate for sequential write is approximately 4794 kbytes and 5048 kbytes for sequential read. The transfer rate for random operations drops to 1122 kbytes for write (2435 kbytes for read).

Sequential, stdio, 32 kbytes

N	write			read		
	u-time	s-time	t-time	u-time	s-time	t-time
200	0.05	0.15	1.32	0.03	0.14	2.74
400	0.14	0.23	2.51	0.13	0.24	3.3
800	0.19	0.57	4.95	0.18	0.47	5.31
1000	0.29	0.67	6.25	0.24	0.61	7.11
2000	0.58	1.34	12.66	0.57	1.13	12.07
4000	0.96	2.99	25.85	1.06	2.25	23.37
6000	1.6	4.28	38.33	1.81	3.26	33.31
8000	2.24	5.54	51.08	2.28	4.68	45.74
10000	2.95	7.34	66.31	2.66	5.94	54.8
20000	5.9	15.36	133.2	5.81	11.65	111.61
30000	8.17	23.84	199.78	8.97	16.9	164.06
40000	11.4	31.38	270.01	11.17	23.48	218.22
50000	13.98	38.72	338.78	13.54	30.06	279.56

Random, stdio, 32 kbytes

N	write			read		
200	0.06	0.16	1.85	0.07	0.1	3.82
400	0.13	0.27	4.29	0.14	0.25	5.85
800	0.29	0.64	8.71	0.26	0.53	11.82
1000	0.36	0.56	10.9	0.27	0.54	15.72
2000	0.73	1.48	22.25	0.47	1.41	30.87
4000	1.2	2.9	46.95	0.92	3.16	61.46
6000	1.64	4.48	73.55	1.55	4.69	95.1
8000	2.48	6.14	100.88	2.44	6.4	130.76
10000	3	7.08	127.49	2.93	7.95	164.51
20000	6.53	15.79	269.19	5.68	16.19	342.9
30000	9.62	24.54	417.57	8.89	25.7	560.3
40000	12.12	34.12	573.96	11.8	34.92	823.52
50000	15.38	41.9	743.72	14.07	45.14	1113.81

Table 3.3: *Performance of LEDA-SM kernel using stdio as I/O method. We use a block size of 32 kbytes. N is the number of operations. u-time (user), s-time (system) and t-time (total) are given in seconds.*

`syscall` is the overall fastest I/O-method for sequential I/Os. It is slightly faster than `stdio`. If we look at the performance of `mmap`, we see that it is the second fastest method for sequential write and the slowest for sequential read. If we consider random operations, `stdio` and `syscall` perform nearly identical while `mmap` performs differently. `mmap` has the worst random write performance (1.92 times slower than `syscall`) while random reads are 1.67 times faster compared to `stdio` and `syscall`. The winner among these different methods is not easy to find as it depends on whether we expect to execute more random or more sequential I/O operations. If we expect to execute more sequential operations, one should either choose `stdio` or `syscall` as I/O-method for the LEDA-SM kernel. We note that the outcome of our experiments may change on other systems.

In a second test we add raw-device disk access. We use Solaris' character-special de-

Sequential, syscall, 32 kbytes						
N	write			read		
	u-time	s-time	t-time	u-time	s-time	t-time
200	0	0.14	1.22	0.01	0.08	1.26
400	0.03	0.22	2.35	0.02	0.18	2.55
800	0	0.52	4.68	0.01	0.38	4.99
1000	0.01	0.6	5.9	0	0.41	5.3
2000	0.01	1.28	12.25	0.05	0.95	10.82
4000	0.06	2.53	24.62	0.04	1.96	21.91
6000	0.17	3.7	36.39	0.06	2.91	31.66
8000	0.14	5.02	48.73	0.12	4.07	42.62
10000	0.2	7.02	62.89	0.2	5.13	52.99
20000	0.34	14.27	126.18	0.31	10.4	108.51
30000	0.53	21.21	190.72	0.52	15.47	160.96
40000	0.78	28.56	254.59	0.66	20.97	220.68
50000	0.92	35.59	320.27	1.05	25.51	271.62

Random, syscall, 32 kbytes						
N	write			read		
200	0	0.14	2.06	0	0.05	2.47
400	0.03	0.3	3.88	0.01	0.27	4.91
800	0.02	0.47	9.01	0.04	0.52	11.03
1000	0.02	0.63	10.68	0.06	0.71	14.13
2000	0.04	1.36	22.14	0.08	1.36	29.01
4000	0.08	2.61	47	0.07	2.84	58.15
6000	0.09	3.76	73.09	0.08	4.24	88.66
8000	0.13	5.42	101.89	0.18	5.77	124.83
10000	0.21	6.55	126.93	0.15	7.02	162.96
20000	0.62	14.38	269.28	0.32	14.62	341.63
30000	0.77	22.25	415.04	0.63	23.37	561.33
40000	1.06	30.45	571.09	0.73	31.59	809.85
50000	1.18	38.57	740.64	0.89	42.28	1107.99

Table 3.4: *Performance of LEDA-SM kernel using syscall as I/O method. We use a block size of 32 kbytes. N is the number of operations. u-time (user), s-time (system) and t-time (total) are given in seconds.*

vice file [10] to access the same disk. It may seem strange to use the character-special device to access a block-driven disk device, but under Solaris, block-special files use the normal file buffering mechanism while character-special files access the raw disk device without using buffering mechanisms. We note that we used a second partition on the same disk, this partition was located on the innermost cylinders. Therefore, we expect a performance decrease compared to the file-access (which was located on the outer cylinders) due to disk zoning.

Table 3.6 summarizes the test results. We experience a slowdown of a factor of two for sequential writes using disk blocks of size 32 kbytes. Sequential read is slightly slower compared to `syscall` and `stdio`. This test shows the effect of operating system buffers compared to raw device access: the file system also buffers the write requests and then writes big chunks at a time. The raw disk access performs a physical I/O for each write request and is therefore slowed down. An obvious consequence is

---

[10] These files are located in directory /dev/rdsk/.

Sequential, mmap, 32 kbytes						
N	write			read		
	u-time	s-time	t-time	u-time	s-time	t-time
200	0.04	0.17	0.24	0.04	0.09	1.42
400	0.12	0.31	0.44	0.17	0.17	2.37
800	0.08	1.07	1.18	0.24	0.58	6.3
1000	0.23	1.17	1.44	0.19	0.66	7.96
2000	0.38	2.52	9.79	0.36	1.64	16.88
4000	0.88	4.93	20.61	1.11	3.26	28.12
6000	1.33	7.34	31.44	1.47	4.82	52.06
8000	1.89	10.25	47.58	2.02	6.22	69.92
10000	2.25	9.23	60.08	2.32	3.95	87.14
20000	4.25	19.3	107.18	4.8	7.54	150.09
30000	6.92	42.2	196.82	7.04	23.25	205.59
40000	8.74	38.69	250.88	9.74	15.81	262.04
50000	11.14	70.57	333.73	11.78	38.86	316.91
Random, mmap, 32 kbytes						
N	write			read		
200	0.06	0.13	0.2	0.02	0.06	2.74
400	0.06	0.3	0.38	0.11	0.1	1.48
800	0.22	0.48	0.72	0.18	0.3	4.93
1000	0.26	0.63	0.93	0.21	0.3	8.98
2000	0.42	1.36	1.84	0.55	0.71	30.65
4000	0.79	2.79	3.89	0.89	1.41	34.24
6000	1.16	4.31	5.76	1.38	2.3	74.96
8000	1.81	8.6	34.81	2.2	6.21	130.95
10000	2.15	6.94	56.89	2	4.36	148.31
20000	4.51	15.42	459.48	4.63	9.6	247.79
30000	6.73	24.19	760.61	7.41	14.6	377.86
40000	9.04	32.5	1083.46	10.09	19.51	520.83
50000	11.94	40.66	1425.12	12.47	25.23	657.57

Table 3.5: *Performance of LEDA-SM kernel using mmap as I/O method. We use a block size of 32 kbytes.* N *is the number of operations. u-time (user), s-time (system) and t-time (total) are given in seconds.*

that $B = 32$ kbytes is not an optimal block size for raw device access. For sequential read, there is no big performance difference as the disk itself is doing read ahead. We experience the same behavior for random write operations. However, random read behaves differently; it is faster than `stdio` and `syscall`. The time overhead induced by the buffering mechanism of the file system does not speed up read operations very much. Indeed, for random reads the buffering mechanism leads to a slowdown. We also see that the system and user time for the raw device accesses are much lower than for file accesses. Indeed, there is no copying between user address space and kernel buffer space for raw devices. This notably decreases system time. It is obvious that bigger block sizes can speed up the raw disk device access method. We therefore tried to figure out the "optimal" block size (see Table 3.7).

A change in the block size has no big effect on the overall read performance of consecutive requests. This can be easily explained by the disk cache and the read ahead strategy of the disk drive. For write, however, we see that the total time to

Sequential, raw, 32 kbytes						
N	write			read		
	u-time	s-time	t-time	u-time	s-time	t-time
200	0.03	0.04	3.35	0.01	0.02	1.25
400	0.02	0.01	6.88	0	0.03	2.38
800	0.03	0.06	13.78	0.01	0.06	4.62
1000	0.06	0.2	17.36	0	0.08	5.63
2000	0.03	0.11	33.72	0.05	0.14	13.07
4000	0.06	0.26	68.43	0.05	0.28	22.5
6000	0.12	0.54	100.4	0.09	0.52	36.34
8000	0.22	0.66	134.71	0.16	0.53	46.49
10000	0.17	0.85	167.36	0.16	0.73	57.87
20000	0.31	1.54	335.62	0.35	1.39	113.07
30000	0.63	2.32	504.73	0.4	2.14	170.72
40000	0.91	2.89	675.66	0.65	2.77	229.53
50000	0.96	3.87	845.16	0.76	3.56	288.5
Random, raw, 32 kbytes						
N	write			read		
200	0	0.02	2.94	0.01	0	3.2
400	0	0.05	6.07	0.02	0.03	6.36
800	0.02	0.06	12.13	0.01	0.04	12.67
1000	0.03	0.08	15.3	0.01	0.1	16.19
2000	0.03	0.06	31.76	0.05	0.14	31.18
4000	0.06	0.29	67.17	0.06	0.27	65.23
6000	0.13	0.47	101.74	0.11	0.57	99.42
8000	0.2	0.65	138.79	0.15	0.58	133.1
10000	0.2	0.67	175.28	0.12	0.66	170.1
20000	0.34	1.34	366.48	0.5	1.55	354.71
30000	0.53	2.15	563.12	0.64	2.31	544.25
40000	0.82	3.01	766.1	0.69	3	739.11
50000	1.09	3.56	975.38	0.68	3.97	939.8

Table 3.6: *Performance of LEDA-SM kernel using raw disk device access. We use a block size of 32 kbytes.* N *is the number of operations. u-time (user), s-time (system) and t-time (total) are given in seconds.*

write the data decreases with increasing block size. Although the total system time is already negligibly small for $B = 32$ kbytes, it can further be reduced. The optimum is achieved for $B = 1024$ kbytes. The transfer rates are 5380 kbytes for write and 5520 kbytes for read. Table 3.8 performs a similar test as for the other file access methods (see Tables 3.3, 3.4, 3.5) but uses blocks of size 1 Mbytes. The same amount of data is transfered, but with less I/O operations as for the tests in Tables 3.3, 3.4, and 3.5.

We notice the following: using the "optimal" disk block size for raw devices we achieve approximately the same performance as for the file system tests. Actually, the write test is a little bit faster while the read test is a little bit slower. We note that the raw device was located on the innermost cylinders of the disk and the file system was located on the outermost cylinders. Thus, we expect the raw device to be faster than the file system as the throughput on the outermost cylinders is higher.

Sequential, raw, variable block size						
	write			read		
B (in kbytes)	u-time	s-time	t-time	u-time	s-time	t-time
32	0.17	0.85	167.36	0.16	0.73	57.87
64	0.09	0.5	112.38	0.05	0.45	56.3
128	0.03	0.25	83.77	0.05	0.21	56.17
256	0.04	0.14	69.7	0.03	0.14	57.21
512	0	0.04	62.75	0.01	0.05	56.89
1024	0.02	0.03	59.47	0	0.03	57.97
2048	0	0.03	59.72	0.01	0.02	60.19

Table 3.7: *Performance of LEDA-SM kernel using raw disk device access. We use a variable block size and write or read 320 Mbytes. u-time (user), s-time (system) and t-time (total) are given in seconds.*

Sequential, raw, 1 Mbytes						
N	write			read		
	u-time	s-time	t-time	u-time	s-time	t-time
7	0	0	1.33	0	0.02	1.46
13	0	0	2.49	0	0.01	2.47
25	0	0	4.72	0	0.02	5.12
32	0	0	6.03	0	0	6.55
63	0	0.01	11.9	0	0	12.01
125	0	0	23.79	0	0	23.33
250	0.01	0.04	48.14	0.02	0.08	46.29
313	0.01	0.07	59.7	0	0.09	57.8
625	0	0.08	119.87	0.01	0.09	115.58
938	0	0.11	181.77	0.04	0.06	173.49
1250	0.02	0.18	242.36	0.02	0.18	234.27
1563	0.01	0.24	304.72	0.03	0.22	293.13

Table 3.8: *Performance of LEDA-SM kernel using raw disk device access. We use a block size of 1 Mbytes. N is the number of operations. u-time (user), s-time (system) and t-time (total) are given in seconds.*

3.8 Application Benchmarks

Our second set of benchmarks covers application algorithms. The first algorithm we choose is *sorting a set of items*. Sorting is a frequent operation in many applications. It is not only used to produce sorted output, but also in many sort-based algorithms such as grouping with aggregation, duplicate removal, sort-merge join, as well as set operations including union, intersect, and except.

3.8.1 Sorting

Sorting is possible as either an *offline* variant where the set of items is available at the beginning and one has to produce the sorted order using a defined linear order '$\leq$' on the items, or as an *online* variant where the items arrive over time and at each time step, one must be able to access the sorted set of items (that are available at that time step). The second (online) variant is normally solved by search tree data structures (see buffer trees [Arg95, Arg96] and B-trees [BM72]). Offline sorting in secondary memory is either handled by *multiway mergesort* variants or *distribution sort* variants. Distribution sort recursively partitions the N data items to be sorted by $S-1$ partition elements into S buckets. The $S-1$ partition elements are chosen so that the buckets have roughly the same size. When this is the case, the bucket size decreases by a $\Theta(S)$ factor from one level of recursion to the next so that $O(\log_S(N/B))$ levels are enough. In each level, the N items are read (using $O(N/B)$ I/Os) and thrown into the corresponding buckets (using $O(N/B)$ I/Os). The partition elements must be chosen using $O(N/B)$ I/Os. Deterministic methods [NV93, VS94a] choose $\sqrt{M/B}$ partition elements resulting in an algorithm that overall performs $O((N/B)\log_{M/B}(N/B))$ I/Os, which is optimal. Multiway mergesort works somehow orthogonal. In the run-creation phase the input data is read and $O(N/M)$ sorted runs of size $O(M)$ each are created. This takes $O(N/B)$ I/Os. In the merge-phase, R runs are iteratively merged into one single run until after $O(\log_R(N/M))$ iterations we end up with one sorted run of size N (the sorted input). Choosing $R = O(M/B)$ leads to overall $O((N/B)\log_{M/B}(N/B))$ I/Os which is optimal. Both methods also work for the D-disk case, but things get more complicated: for distribution sort, it is not easy to find $\Theta(S)$ partitioning elements with $O(N/(DB))$ I/Os. Nodine and Vitter showed [NV93] how this can be solved for the general D-disk case (see also [VS94a]). For mergesort the question is how to merge R runs together in each merge pass using $O(N/(DB))$ I/Os. This was positively answered by Barve *et al.* using a randomized merge approach [BGV97].

Other variants were developed, most of them addressed the problem of sorting efficiently in the multi-disk case. The Greed Sort method of Nodine and Vitter [NV95] was the first optimal deterministic algorithm for sorting with multiple disks. It is merge based, but relaxes the merge condition so that only an approximately merged run is created but its saving grace is that no two inverted items are too far apart. Then it invokes Columnsort [Lei85] to produce the final order. Another approach is bundle-sort [MSV00] which assumes that the input consists of $k < N$ different keys. Bundle-sort (which is a variant of distribution sort) sorts these keys in $O((N/B)\log_{M/B} k)$ I/Os which is optimal. The advantage of bundle sort is that it sorts in-place.

Our test consists of comparing multiway mergesort in the single disk case with standard internal-memory quicksort approaches. Both are classical offline sorting approaches. Multiway mergesort is the classical external sorting algorithm for single and multiple disks. Compared to the distribution sort approach, it offers the advantage that we do not rely on how to choose partition elements. Deterministic methods choose $\sqrt{M/B}$ partition elements which has the effect that the number of levels in the recursion is $\lceil \log_{\sqrt{M/B}}(N/B) \rceil$ and is hence doubled compared to mergesort ($\lceil \log_{M/B}(N/B) \rceil$). Random sampling methods are normally superior to deterministic methods. If the sample size (number of randomly picked elements) is chosen carefully, it is possible to achieve the optimal number of recursive levels or the optimal number of levels plus one.

Mergesort approaches have the advantage that one can produce initial runs of size bigger than M. By a technique called *natural selection* [FW72], it is possible to create initial runs of average length eM where e is the Euler constant. This technique can effectively decrease the number of recursive merge steps.

Quicksort [Hoa62] is one of the standard internal memory sorting algorithms. The average run time of quicksort is $O(N \log(N))$ time. The advantage of quicksort is that it runs in-place.

We perform the following test: we sort random integer numbers using either quicksort or multiway mergesort. The test is performed on a SUN UltraSparc-1 with a single 143 MHz processor and a local 9 GB SCSI hard disk running Solaris-2.6 as operating system. The machine has 256 Mbytes internal memory. Table 3.9 summarizes the results. The external sorting algorithm uses approx. 16 Mbytes of internal memory

Sorting random integers						
	mergesort			quicksort		
N(in million)	u-time	s-time	total	u-time	s-time	total
1	2.55	0.15	3.14	1.95	0	1.96
2	5.31	0.25	6.6	4.07	0	4.08
4	11.06	0.45	13.31	8.6	0	8.64
8	30.62	1.83	36.83	17.96	0	18.03
10	39.14	1.86	46.31	22.91	0	22.98
20	82.49	4	96.45	47.98	0	48.22
40	183.94	10.02	270.39	99.83	0	100.13
60	305	17.4	442.06	155.27	2.22	289.09
80	444.01	22.77	597.14	210.37	10.82	967.95
100	592.1	28.37	770.65	265.99	21.2	1751.18
120	760.59	34.3	968.21	331.01	35.3	2622.38
140	953.87	40.46	1200.65	379.56	35.64	2763.28
160	1157.11	45.19	1445.08	440.89	37.87	3122.1
180	1365.2	51.61	1669.22	499.12	47.02	3786.7
200	1603.87	56.42	1950.44	559.54	65.3	4830.88

Table 3.9: *Sorting random integers with LEDA quicksort and LEDA-SM multiway mergesort. The disk block size is 32 kbytes and mergesort uses 512 internal blocks (of size 32 kbytes) for sorting. A "-" indicates that the test could not be started due to insufficient swap space.*

while quicksort is allowed to use the whole internal memory plus swap-space which resides on the same local disk. For multiway mergesort, it is possible to merge all initially created runs in one phase. If we look in more detail at Table 3.9, we notice the following: the user time (u-time) of multiway mergesort is more than twice as high as the user time of quicksort. System time (s-time) is quite the same with mergesort being a little bit faster. Concerning the total time, for our last test (200 million random integers), mergesort is 2.47 times faster than quicksort. The higher user time of mergesort can be easily explained: the run creation step of multiway mergesort has the same CPU time complexity as the quicksort algorithm and additionally, we have to merge the runs. The run creation phase accounts for at least half the total running time as all runs are merged in a single phase. Multiway mergesort is CPU bound, it takes longer to create a sorted run in internal memory via an internal sorting algorithm than to load the run from disk, *i.e.* a single run of 4 Mbytes size is created in 3 seconds, reading and writing takes about 1.1 seconds and sorting takes 1.9 seconds. We notice that the CPU time (u-time) accounts for 80% of the total time (t-time).

In the next example, we will sort random tuples of two integer values. A comparison of tuple (i, j) and (i', j') is done lexicographically, *i.e.* $(i, j) < (i', j')$, if $i < i'$ or $i = i'$ and $j < j'$. We sort the same number of items as in the previous test, the results are summarized in Table 3.10. Thus, the total amount of data that we transfer to/from the disk in each stage of the mergesort is double as high as for our first test (a tuple is 8 bytes while an integer is 4 bytes). However, a comparison of a tuple is more complicated than a comparison of integers so that we expect that the total executed time will be more than double as high.

Sorting random tuples						
	mergesort			quicksort		
N(in million)	u-time	s-time	total	u-time	s-time	total
1	7.65	0.5	9.16	2.53	0.08	2.69
2	15.64	1.04	18.81	5.31	0.2	5.61
4	27.48	2.32	34.12	11.01	0.44	11.61
8	47.45	3.98	59.97	23.14	0.95	27.2
10	58.21	4.77	78.01	29.25	5.81	396.07
20	118.58	11.52	194.94	63.15	25.48	1808.49
40	269.91	24.65	435.08	133.57	66.92	4573.72
60	455.88	35.2	677.54	207.39	110.57	9322.54
80	684.07	47.21	995.2	-	-	-
100	954.31	61.51	1412.86	-	-	-
120	1246.43	74.3	1875.19	-	-	-
140	1583.1	86.88	2393.79	-	-	-
160	2012.9	102.33	3029.53	-	-	-
180	2373.47	120.79	3638	-	-	-
200	2832.32	139.41	4517.16	-	-	-

Table 3.10: *Sorting random tuples with LEDA quicksort and LEDA-SM multiway mergesort. The disk block size is 32 kbytes and mergesort uses 512 internal blocks (of size 32 kbytes) for sorting. A "-" indicates that the test could not be started due to insufficient swap space.*

We see again that mergesort is CPU bound. The user time accounts for more than $2/3$ of the total execution time. Mergesort has approximately double the user time than quicksort. The reason for this is again the fact that mergesort consists of two phases and each phase has the same time complexity as the internal quicksort approach. For quicksort we see that it is I/O bound: user and system time account for only 3 % of the total time. As a result, external mergesort is now up to 10 times faster than the internal quicksort approach. We now look in more detail at the different phases of mergesort and compare our two tests for integers and tuples. For both tests, the run creation phase is faster than the merge phase. Both phases have the same CPU-complexity of $O(N \log_2 N)$. In the run creation phase we perform I/Os to consecutive disk block locations, while in the merge phase we expect a larger number of random I/Os. Thus, with increasing N, the merge phase dominates the run time (see Table 3.11).

Mergesort phases				
	integers		tuples	
N(in million)	run creation	merge	run creation	merge
1	3	0	9	0
2	7	0	19	0
4	14	0	28	6
8	27	10	46	14
10	32	14	55	23
20	64	33	119	76
40	128	141	244	191
60	241	196	337	340
80	293	295	447	549
100	369	401	555	858
120	432	536	671	1204
140	500	700	780	1614
160	571	873	895	2135
180	650	1019	1014	2624
200	719	1231	1127	3390

Table 3.11: *Time distribution for run creation and merging in LEDA-SM multiway mergesort. The disk block size is 32 kbytes and mergesort uses 512 internal blocks (of size 32 kbytes) for sorting. Time is given in seconds.*

3.8.2 Simple Graph Algorithms

In this subsection, we look at simple graph algorithms. Our tests consist of depth-first search and Dijkstra's shortest path algorithm [Dij59] on directed graphs. Graph algorithms are considered to be hard problems in external memory. The I/O complexity of several basic graph algorithms remains open, including depth-first search, breadth-first search and shortest paths. We first review some known results for external memory graph algorithms and then perform some experiments using a LEDA-SM implementation of these algorithms.

Let G be a directed graph, let V be its set of nodes, $E \subseteq V \times V$ be its set of edges. We denote by $|V|$ the size of the set of nodes ($|E|$ respectively for the size of the set of edges). Let $e = (u, v)$ be an edge of G; u is called the *source node* of e, v is called the *target node* of e. u and v are also called endpoints of e. An edge is said to be *incident* to its endpoints. All edges having source node u are said to be *adjacent* to u. Graphs are often represented by so called *adjacency lists*. Here, the graph representation consists of $|V|$ lists, where the list, labeled x, stores all edges that are adjacent to x. Both, LEDA-SM and LEDA use adjacency list representations for graphs.

Our first example consists of doing a depth-first search traversal (DFS) of G. In internal memory, this problem can be solved in $O(|V| + |E|)$ time, which is optimal. Below is the original LEDA code for depth-first search.

74 ⟨*internal_dfs* 74⟩≡

```
static void dfs(node s, node_array<bool>& reached,
                list<node>& L)
{
   L.append(s);
   reached[s] = true;
   node v;
   forall_adj_nodes(v,s)
      if ( !reached[v] ) dfs(v,reached,L);
 }

list<node> DFS(const graph&, node v,
               node_array<bool>& reached)
{
  list<node> L; dfs(v,reached,L); return L;
}
```

Boolean array $reached$ is used to store the nodes of G that were already reached during the traversal. At the end, list L stores the result of the depth-first search traversal.

In external memory, graph algorithms can be classified to be either *fully-external* or *semi-external*. Fully-external graph algorithms assume that it is not possible to store information of size $\Theta(|V|)$ or $\Theta(|E|)$ in internal memory. Semi-external graph algorithms are able to store information of that size in internal memory. The above example for DFS is semi-external, if we assume that $reached$ can be stored in internal memory, and then it runs in $O(|V| + |E|/B)$ I/Os. Chiang *et al.* [CGG^{+}95] showed that a natural extension of the internal memory algorithm is able to solve the DFS in $O((|V|/M) \cdot (|E|/B))$ I/Os for the fully-external case. The algorithm assumes that the boolean array *reached* cannot be stored in internal memory. Instead, the algorithm keeps a dictionary of size at most M. Visited nodes are stored in the dictionary. When the dictionary gets full, the graph G is compacted, *i.e.* edges that are pointing to nodes that were already visited are deleted from G, the dictionary is cleared and we proceed with the algorithm.

Kumar and Schwabe [KS96] proposed a different algorithm that solves DFS in $O(|V| \log_2(|V|) + (|E|/B) \log_2(|E|/B))$ I/Os using so called tournament trees plus an external memory priority queue. The original algorithm of Kumar and Schwabe only worked for undirected graphs. Buchsbaum *et al.* [BGVW00] extended the algorithm to the directed graph case without changing the I/O bounds. Meyer [Mey01] showed that DFS on undirected planar graphs needs $O(\frac{|V|}{\gamma \cdot \log_3 B} + Sort(|V| \cdot B^{\gamma}))$ I/Os and $O(|V| \cdot B^{\gamma})$ external space, for any $0 < \gamma \leq 1/2$. Unfortunately, no lower bound is known for this simple graph algorithm. However, the community believes that the $\Theta(|V|)$ term in the I/O-bound cannot be reduced for arbitrary graphs. In practice, fully-external algorithms are not useful. For many practical cases, $|E| = O(|V|)$ and the term $|E|/B$ is negligible compared to the $O(|V|)$ term. Therefore, the focus is on semi-external algorithms as even for very large graphs like telephone call graphs, one can assume that it is possible to store information of size $O(|V|)$ in internal memory [Abe99].

We implemented the algorithm of Chiang *et al.* for the fully-external case and the semi-external algorithm (the semi-external case is the standard DFS using an in-core boolean array, but an external memory graph data structure). Additionally, we implemented a DFS variant where we use an external array $ext\_array{<}bool{>}$ instead of an internal $node\_array{<}bool{>}$. This algorithm performs $O(|V| + |E|)$ I/Os in the worst case. We then performed several tests on a SUN UltraSparc-1 with a single 143 MHz processor, 256 Mbytes main memory and a local 9 GB SCSI hard disk running Solaris-2.6 as operating system. We used a block size of 8 kbytes for the external memory algorithms so that the external disk block size equals the page size of the virtual memory subsystem. By this, we were able to fairly compare in-core and secondary memory algorithms as both use the same disk block size so that only the algorithmic and data structure features matter.

In a first test, we compared the different external-memory DFS variants against each other on random graphs with n nodes and m edges. We tested Chiang *et-al.*'s algorithm with varying dictionary sizes against our external memory variant of DFS that uses an external array $ext\_array{<}bool{>}$ to keep track of already visited nodes, and against a semi-external variant that uses an internal memory array to keep track of visited nodes. We note that all tests execute at least $\Omega(n)$ I/Os. Table 3.12 summarizes the results.

External memory DFS variants							
	Chiang-et-al					ext. bit-array	int. bit-array
(n, m)	1k	4k	8k	16k	32k		
$10^3, 4 \cdot 10^4$	76	66	66	77	78	95	48
$10^4, 5 \cdot 10^4$	99	81	77	89	90	139	54
$10^4, 10^5$	150	106	96	100	101	227	57
$10^5, 10^5$	491	309	286	289	297	275	179
$10^5, 5 \cdot 10^5$	1798	695	511	421	380	1272	219
$10^5, 10^6$	3034	1015	675	507	427	2321	227
$10^6, 10^6$	3496	1148	756	561	468	2345	226
$10^6, 2 \cdot 10^6$	6311	1922	1199	836	652	4467	258
$10^6, 5 \cdot 10^6$	14374	4355	2651	1769	1344	10946	395
$10^6, 10^7$	27924	8483	5323	3574	2699	21861	750

Table 3.12: *Comparison of external memory DFS variants on random graphs with n nodes and m edges. The first five variants are Chiang-et-al.'s algorithm with dictionary size 1000, 4000, 8000, 16000 and 32000. The last two variants use an external memory bit-array and an internal memory bit-array. All run times are given in seconds.*

Chiang *et al.*'s algorithm gets faster with increasing dictionary size. The run time converges towards the run time of external DFS with an internal bit array, if the dictionary size of Chiang *et al.*'s algorithm is large enough. The variant that uses the external memory bit array is the overall slowest algorithm. Clearly, the semi-external DFS variant that uses an in-core boolean array is the fastest algorithm.

In the next test we compare LEDA's DFS algorithm against LEDA-SM's the semi-external DFS variant that uses an internal boolean array to keep track of nodes that were already visited. Our tests are performed on different graph types, namely random graphs with n nodes and m edges, complete graphs with n nodes, and $n \times m$ grid graphs. The graph types have different features. For the random graphs, we choose m to be in the interval $[n, 10 \cdot n]$. Thus, the number of edges is small compared to the number of nodes and hence the $O(n)$ I/O term in the semi-external DFS variant dominates the $O(m/B)$ term. The graphs are chosen in such a way that for the larger graphs, swap space must be used for the in-core DFS algorithm. We expect that the semi-external graph variant outperforms the in-core DFS algorithm, as the operating system is not able to perform the paging in a clever way.

For complete graphs, $m = \Omega(n^2)$. n is quite small so that the $O(n)$ I/O term of the semi-external algorithm does not play an important role if compared to the $O(n^2/B)$ I/O term used to scan the edges. It is therefore interesting to see if the semi-external variant is faster than the in-core algorithm.

$n \times m$ grid graphs consist of $m \cdot n$ nodes and $2 \cdot n \cdot m - n - m$ edges. The graph consists of n rows, each having m nodes. The edge set is formed of the edges $(v_{i,j}, v_{i,j+1})$, $i = 1, \ldots, n$, $j = 1, \ldots m - 1$ and $(v_{i,j}, v_{i+1,j})$, $i = 1, \ldots, n - 1$, $j = 1, \ldots, m$. For this test, the number of edges is again small compared to the number of nodes. As for random graphs it is interesting to experiment if the semi-external variant can beat the in-core variant. The results of our tests are summarized in Table 3.13.

External memory DFS against LEDA DFS								
Random graphs			complete graphs			grid graphs		
(n, m)	LEDA	LEDA-SM	n	LEDA	LEDA-SM	$n \times m$	LEDA	LEDA-SM
$10^3, 4 \cdot 10^4$	1	48	2000	3	220	110000,10	4	85
$10^4, 5 \cdot 10^4$	1	54	2100	4	248	120000,10	5	98
$10^4, 10^5$	1	57	2200	4	273	130000,10	9	103
$10^5, 10^5$	1	179	2300	29	290	140000,10	50	133
$10^5, 5 \cdot 10^5$	3	219	2400	137	319	150000,10	88	155
$10^5, 10^6$	5	227	2500	231	323	160000,10	208	161
$10^6, 10^6$	7	226	2600	327	364	170000,10	273	184
$10^6, 2 \cdot 10^6$	25	258	2700	428	404	180000,10	313	203
$10^6, 5 \cdot 10^6$	1767	395	2800	469	454	190000,10	366	227
$10^6, 10^7$	20122	750	2900	527	489	200000,10	498	232
Aver. Time	2193	241.3		215.9	338.4		181.4	158.1

Table 3.13: *Comparison of LEDA DFS and external memory DFS with an internal bit array on random graphs with n nodes and m edges, on complete graphs with n nodes and on $n \times m$ grid graphs. All execution times are given in seconds. The last row gives the arithmetic mean execution time.*

For random graphs, in-core DFS is much faster than semi-external DFS on the first eight test graphs. Only for large random graphs (the last two inputs) there is a dramatic slowdown of in-core DFS and semi-external DFS is faster. If we look at the mean execution time, the semi-external variant is superior to the in-core variant. If we additionally consult Table 3.12, we see that also Chiang *et al.*'s algorithm is able to beat the in-core DFS routine. The largest LEDA graph has a size of approximately 492 Mbytes so that surely swap space must be used. Obviously, the paging algorithm of the operating system is not able to exploit locality of reference, although the LEDA graph data structure (adjacency list) shows locality of reference. If one could tell the operating system to always keep boolean array $reached$ in internal memory, in-core DFS would be as fast as semi-external DFS.

For complete graphs, in-core DFS is superior to semi-external DFS. Although we are able to beat in-core DFS on the last 3 graphs, the mean execution time of in-core DFS is lower than that of semi-external DFS. However, we expect that for larger graphs than the ones we tested, the gap between in-core DFS and semi-external DFS will close resulting in semi-external DFS being much faster than in-core DFS. We note that Chiang *et al.*'s algorithm will not be much slower than semi-external DFS as $|V|$, the number of nodes, is small so that one or two rounds are enough to complete Chiang *et al.*'s algorithm. If we look at grid graphs, we see that for half of the tests, semi-external DFS is faster than in-core DFS. Again as for random graphs, the number of nodes is large compared to the number of edges. However, the gap between the execution time is not as large as for random graphs.

The tests show that the external DFS algorithms are fast if n, the number of nodes, is quite large compared to the number of edges. We do not think that very large complete graphs ($|V| \gg M$) can occur in practice as then, the total input size would be $\Theta(M^2)$, which is not reasonable for today's machines with Gigabytes of main memory as in that case, the total input size would be in the range of Hexabytes. Hence, in practical applications, the average outdegree of a node is a constant and for this setting, semi-external and fully-external DFS algorithms are superior to in-core variants.

In a second test we compare in-core Dijkstra's shortest path algorithm against a semi-external variant. In external memory, the problem of Dijkstra's algorithm is the fact that it relies on `decrease_priority` operations for the priority queue and this operation is not yet supported in an efficient way for external memory priority queues. The use of `decrease_priority` operations in Dijkstra's algorithm ensures that there are at most n nodes in the priority queue. This problem of a lacking `decrease_priority` operation is circumvented in the semi-external case as follows: We only perform `insert` and `delete_min` operations on the priority queue. In the case that Dijkstra's internal-memory algorithm performs a `decrease_priority` operation on node u, we simply insert node u with its new distance value into the priority queue. Therefore, for any node u there can be multiple entries of u with different distance values in the priority queue. Whenever a node u is deleted from the priority queue, we know that its distance is the shortest distance from the starting node s to u. All other occurrences of u that still remain in the priority queue are spurious hits. To circumvent the problem of revisiting node u again, we use an internal bit array to memorize that node u was deleted. The above algorithm performs $|E|$ priority queue operations. It runs in $O(|V| + |(E/B)| \log_{M/B}(|E|/B))$ I/Os under the hypothesis that we are able to maintain a bit array for $|V|$ entries in internal memory.

We tested this algorithm against the internal memory Dijkstra algorithm, implemented with LEDA. Table 3.14 summarizes the results for random graphs with n nodes and m edges.

We see again a tremendous slowdown of the in-core algorithm if compared to the semi-external variant. Although the in-core Dijkstra algorithm is much faster, if the graph and the necessary data structures fit into main memory, the semi-external algorithm outperforms the internal memory algorithm for the last four test graphs. In the end, it is more than 22 times faster.

External memory Dijkstra against LEDA Dijkstra			
n	m	LEDA	LEDA-SM
10000	30000	1	16
10000	50000	1	23
10000	100000	1	20
100000	100000	1	77
100000	500000	3	98
100000	1000000	5	105
1000000	1000000	5	107
1000000	2000000	25	132
1000000	3000000	451	168
1000000	3500000	1767	201
1000000	5000000	6931	316
1000000	10000000	15942	717
Aver. time		2094.41	165

Table 3.14: *Comparison of Dijkstra's algorithm implemented with LEDA and LEDA-SM on random graphs with n nodes and m edges. All execution times are given in seconds. The last row gives the arithmetic mean execution time.*

3.9 Summary

In this chapter we introduced the library LEDA-SM. Our library is an extension of the well known LEDA library towards external memory computation. LEDA-SM provides a simple but yet quite exact model of underlying hard disks by modeling each disk as a collection of fixed sized blocks. Although this might seem to be nonelastic from the programmer's point of view, it is conform with the functionality that hard disks provide. By the use of simple C++ classes like $block\ identifiers$ (B_ID) and $logical\ blocks$ ($block{<}E{>}$), it is relatively easy to develop external memory algorithms and data structures. The connection to LEDA's huge collection of in-core data structures and algorithms simplifies that process. The kernel of LEDA-SM seems to be too complex, there is the necessity of disk block and user management. Other libraries like TPIE and VIC* used much simpler approaches. However, these libraries are not as flexible as LEDA-SM. External memory is provided by hard disks and the disk drives themselves are quite complex (Chapter 2 tried to give some intuition). File systems are one layer above the real hard disk and they are developed for other kinds of application patterns than the ones that external memory algorithms create. Although file systems can be tuned, this process is quite difficult and not straightforward at all. Some things are still quite uncontrollable like the space used for the buffer cache. Raw devices allow to circumvent some of these problems at the cost of the need to manage disk blocks directly. As secondary memory algorithms access the hardware of the computers, we believe that it is natural to give low-level access to the hardware.

The tests in Section 3.8 are quite simple so that people might say that it is obvious that the external memory algorithms are faster. Although the external memory algorithms are faster for all the tests, the gap is sometimes not as big as one might expect. Especially sorting is not dramatically faster. Some speedups are however amazing, *e.g.* for the graph examples. In the next two chapters we will develop new external memory algorithms and data structures. These algorithms will not only be compared against their in-core counterparts but also against existing external memory algorithms. We will use LEDA-SM to implement all these external memory algorithms and data structures.

Chapter 4

Case Study 1: Priority Queues

A *priority queue* is a data structure that stores a set of items, each one consisting of a tuple which contains some *(satellite) information* plus a *priority* value (also called *key*), drawn from a totally ordered universe. A priority queue supports the following operations on the processed set: `access_minimum` (returns the item in the set having minimum key), `delete_min` (returns and deletes the item in the set having the minimum key), and `insert` (inserts a new item into the set). Priority queues (hereafter PQs) have numerous important applications: combinatorial optimization (*e.g.* Dijkstra's shortest path algorithm [Dij59]), time forward processing [CGG$^+$95], job scheduling, event simulation and online sorting, just to cite a few. Many PQ implementations currently exist for small data sets fitting into the *internal memory* of the computer, *e.g.* k–ary heaps [Wil64], Fibonacci heaps [FKS84], radix heaps [AMOT90], and some of them are also publicly available to the programmers (see *e.g.* the LEDA library [MN99]). However, in large-scale event simulations or on instances of very large graph problems (as they recently occur *e.g.* in geographical information systems), the performance of these *internal-memory* PQs may significantly deteriorate, thus being a bottleneck for the overall underlying application. In fact, as soon as parts of the PQ do not fit entirely into the internal memory of the computer, but reside in its external memory (*e.g.* on the hard disk), we may observe a heavy paging activity of the external storage devices, because the pattern of memory accesses is not tuned to exhibit any locality of reference.

In this chapter[1], we study the behavior of priority queues in a secondary memory setting. We will compare several known internal memory and secondary memory priority queue data structures both theoretically and experimentally. We use the classical secondary memory model of Vitter and Shriver to measure the I/O performance (see also Section 2.3.2). Later on, we will consider bulk I/Os [FFM98] in a practical setting where we implement all priority queues using LEDA-SM. Furthermore, we introduce two novel priority queues that are especially designed for an external memory setting. The first PQ proposal is an adaptation of the *two-level radix heap* [AMOT90] to the external memory. This external PQ supports monotone insertions and manages integer keys in a range-size C. It achieves an amortized I/O-bound of $O(1/(DB))$ for the `insert` and $O((1/(DB))\log_{\frac{M}{(DB\log C)}} C)$ for the `delete_min` operation.

---

[1] Parts of this work appeared in [BCMF99] and will also appear in [BCMF00].

The space requirement is optimal, and the I/O-constants are actually very small, thus, we expect good practical performances. Our second PQ proposal is a simplification of [BK98], carefully adapted to exploit the advantage of a collection of fixed-size lists over balanced tree structures. The resulting array-based PQ is easy to implement, is I/O-optimal in the amortized sense, does not impose any constraints on the priority values (cfr. radix heaps), and involves small constants in the I/O, time, and space bounds. Consequently, this structure turns out to be very promising in the practical setting and therefore deserves a careful experimental analysis to validate its conjectured superiority over the other PQs.

In the second part of the chapter, we will perform an extensive set of experiments comparing the implementation of four external-memory PQs: one based on buffer trees [Arg95], another based on B-trees [BM72], and our two new proposals: r-heaps and array-heaps. Additionally, we will compare these PQs against four internal-memory priority queues: Fibonacci heaps [FT87], k-ary heaps [Meh84a] , pairing heaps, [SV87] and internal radix heaps [AMOT90]. Our experimental framework includes some simple tests, which are used to determine the actual I/O-behavior of `insert` and `delete_min` operations, as well as more advanced tests aimed to evaluate the CPU-speed of the internally used data structures. As a final experimental result, we will also test the performance of our proposed PQs in a real setting by considering sequences of `insert`/`delete_min` operations that were traced from runs of Dijkstra's shortest path algorithm. These sequences will allow us to check the behavior of our PQs on a "non-random", but application driven, pattern of disk accesses.

Previous Work It has been observed by several researchers that k–ary heaps perform better than the classical binary heaps on multi-level memory systems [NMM91, LL96]. Consequently, a variety of external PQs, already known in the literature, follow this design paradigm by using a *multi-way tree* as a basic structure. Buffer trees [Arg95, HMSV97] and M/B–ary heaps [KS96, FJJT99] are multi-way trees and achieve optimal $O((1/B)\log_{M/B} N/B)$ amortized I/Os per operation. Unfortunately, most of these data structures are quite complex to implement (the simplest proposal is given in [FJJT99]), and the constants hidden in the space and I/O bounds are not negligible. For instance, all these data structures require the maintenance of some kind of child pointers and some rebalancing information, which induce space overhead and entail to write non-trivial rebalancing code. Recently, starting from an idea of Thorup [Tho96] for RAM priority queues, Brodal and Katajainen [BK98] designed an external-memory PQ consisting of a *hierarchy of sorted lists* that are merged upon level– or internal-memory overflows [2]. Their main result is to achieve optimal worst-case I/O-performance. However, their focus on worst-case efficiency complicates the algorithm such that it is less attractive for practical applications. We remove this deficit by redesigning the data structure. Additionally, we provide an implementation of the hierarchical approach, which combines attractive amortized bounds with simplicity. We show in Section 4.2 that this *hierarchical approach* offers some advantages over the tree-based data structures, which make it appealing in practice.

---

[2] A similar idea was used by Sanders to design cache efficient priority queues[San99].

4.1 External Radix Heaps

4.1.1 One-disk model

Our first external-heap proposal consists of a simple and fast data structure based on two-level radix heaps [AMOT90] (hereafter shortly *R–heaps*). Let C be a positive integer constant and assume that the element priorities are no-negative integers. R–heaps work under the following condition:

Condition 4.1.1 (R-Heaps). *Upon insertion, any priority value must be a non-negative integer in the range* $[\mathtt{m}, \mathtt{m}+\mathtt{C}]$*, where* $\mathtt{m}$ *is the priority value of the last element removed from the heap via a delete_min operation (*$\mathtt{m} = \mathtt{0}$ *if no delete_min was performed up to now).*

Hence the queue is *monotone* in the sense that the priority values of the deleted elements form a *nondecreasing* sequence. This requirement is fulfilled in many applications, *e.g.* in Dijkstra's shortest path algorithm. The need for integer priorities is not severe, since interpreting the binary representation of a non-negative floating point number as an integer does not change the ordering relation [IEE87]. R–heaps with $C = 2^{64}$ can therefore also be used for 64 bit floating point numbers.

The structure of the *external* memory version of the R-heap data structure is defined as follows. Let r be an arbitrary positive integer (also called *radix*) and choose the parameter h to be the minimum integer such that $r^h > C$ (*i.e.* $h = \lceil \log_r(C+1) \rceil$). Let k be the priority of an arbitrary element in the queue and let $k_v \dots k_h k_{h-1} \dots k_0$ be its representation in base r. Similarly, let $m_v \dots m_h m_{h-1} \dots m_0$ be the representation in base r of the current minimum priority $\mathtt{min}$. According to Condition 4.1.1, we know that if an element with priority k belongs to the queue, then $k - \mathtt{min} \le r^h$. Consequently, $k_v = m_v, k_{v-1} = m_{v-1}, \dots, k_{h+1} = m_{h+1}$ and either $k_h = m_h$ or $k_h = (m_h + 1) \bmod r$. The latter case only occurs, if $m_h = r - 1$ and $k_h = 0$. Namely, if the two priorities differ, they must differ in their least significant h digits.

The external R–heap consists of three parts (see also Figure 4.1):

1. A collection of h arrays, each of size r. Every array entry is a linear list of blocks called a *bucket*. Let $\mathcal{B}(i, j)$ denote the bucket associated with the j–th entry of the i–th array for $0 \le i < h$, $0 \le j < r$. Each bucket keeps its first block (disk page) in main memory. Bucket $\mathcal{B}(0, m_0)$ stores the first two blocks in main memory.

2. A special bucket $\mathcal{N}$ that also stores its first block in main memory. The internal space requirement of point 1 and 2 constrain r to satisfy the relation $(h \cdot r + 2) \cdot B \le M$.

3. An internal memory priority queue $\mathcal{Q}$ containing all indices of the non-empty buckets. These indices are ordered lexicographically, *i.e.* $(i, j) < (i', j')$ if either $i < i'$ or $i = i'$ and $j < j'$. $\mathcal{Q}$ never stores more than $h \cdot r$ indices.

All elements of the R-heap reside in any of the buckets, either in bucket $\mathcal{N}$ or in a bucket $\mathcal{B}(i, j)$. An element with priority k is stored according to the difference between its r-ary representation and the r-ary representation of $\mathtt{min}$. Let $k = k_v, \dots, k_0$ and `min`$= m_v, \ldots, m_0$ be the corresponding r-ary representations. The element with priority k is stored in bucket $\mathcal{N}$ if $k_h = (m_h + 1) \bmod r$, in all other cases it is stored in bucket $\mathcal{B}(i, j)$ where

$$i = \max(\{l \mid m_l \neq k_l, 0 \leq l < h\} \cup \{0\}) \textbf{ and } j = k_i. \tag{4.1}$$

The elements inside a bucket are not sorted. Note that an element with priority equal to `min` is stored in bucket $\mathcal{B}(0, m_0)$. Furthermore, all elements that are stored in buckets $\mathcal{B}(0, j)$ must have the same priority value j and the buckets $\mathcal{B}(0, j)$,with $j < m_0$ must be empty. We now describe operations `insert` and `delete_min` .

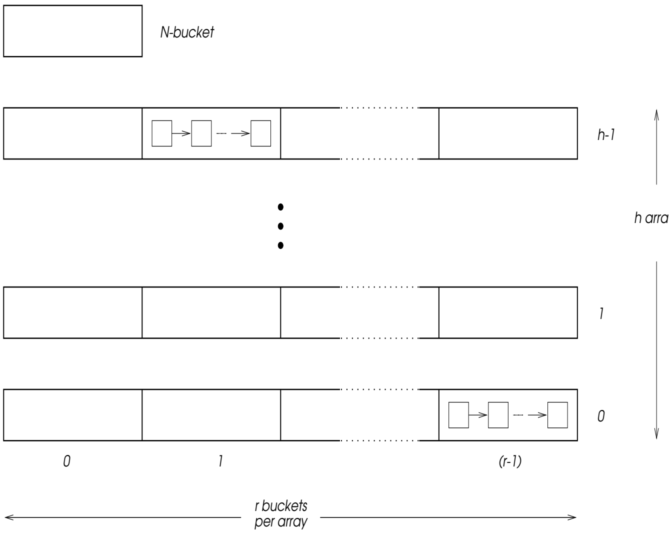


Figure 4.1: *The structure of the external radix heap.*

Insert: In order to insert a new element with priority k in the external R–heap, we first compute the least significant $h+1$ digits of k in base r , thus taking $O(h)$ time and no I/Os. Then, we insert that element into the bucket $\mathcal{N}$ whenever $k_h = (m_h+1) \bmod r$; otherwise, we insert that element into the bucket $\mathcal{B}(i, k_i)$, where $i = \max(\{l \mid m_l \neq k_l, 0 \leq l < h\} \cup \{0\})$. This takes $O(1)$ time. If bucket $\mathcal{B}(i, k_i)$ was empty before the `insert` operation, we also insert the index (i, k_i) into $\mathcal{Q}$ taking $O(\log{(rh)})$ time. Notice that the insertion of an element with priority k into the first block of bucket $\mathcal{B}(i, k_i)$ can completely fill it. In this case, we write that block to the disk and link it with the previous heading block of that bucket, *i.e.* the disk block is inserted at the front of the linked list. This takes $O(1)$ I/Os, thus amortized $O(1/B)$ I/Os per inserted element. Bucket $B(0, m_0)$ is treated in an special way. We keep two blocks in main memory, *i.e.* bucket $\mathcal{B}(0, m_0)$ works as an external stack. When the second block is filled by inserted elements, we move the second block two disk and copy the first block to the second block (see also Section 3.5 were we describe external stacks in detail). This also takes $O(1/B)$ I/Os per inserted element. The special treatment of bucket $\mathcal{B}(0, m_0)$ is necessary because of the `delete_min` operation. This will be explained below.

Delete_min: If the bucket $\mathcal{B}(0, m_0)$ is not empty, we delete an arbitrary element from this bucket. Notice that all elements in bucket $\mathcal{B}(0, m_0)$ have the same priority and they are all equal to `min`. It may be necessary to load a block from disk, as a previously occurring `delete_min` operation completely deleted both internal blocks of bucket $\mathcal{B}(0, m_0)$. As bucket $\mathcal{B}(0, m_0)$ is treated as an external stack, we are able to guarantee $O(1/B)$ I/Os and $O(1)$ time per operation in an intermixed sequence of inserts and delete operations on bucket $B(0, m_0)$. Note that this is not possible, if we only store a single block in-core.

If instead the bucket $\mathcal{B}(0, m_0)$ is empty, we access the internal priority queue $\mathcal{Q}$ and determine the lexicographically first non-empty bucket, which is either some bucket $\mathcal{B}(i, j)$ or the bucket $\mathcal{N}$. In both cases, we scan the selected bucket and determine the new value for the minimum element `min`. Since the minimum has changed, we need to reorganize the elements in the current bucket $B(i, j)$. These elements are redistributed according to the rule exploited for the `insert` operation, thus they are moved to buckets in lower levels. These buckets were empty before the redistribution, so that their filling induces a change into the internal priority queue $\mathcal{Q}$. It is crucial to observe that all buckets $\mathcal{B}(i', j')$, $i' > i$ or $i' = i$ and $j' > j$, including bucket $\mathcal{N}$ do not change, because the new `min` has changed *only* in the least significant i digits wrt. the old `min`. Hence, the elements that differed in the i-th digit from the old `min` also differ in the i-th digit from the new `min`. Therefore, these elements remain in their buckets and overall no other bucket, except than $\mathcal{B}(i, j)$, must be reorganized.

We point out that each element in the PQ can be redistributed at most h times, namely once for each of the h arrays. In fact, each time an element is redistributed, it moves from a bucket $\mathcal{B}(i, j)$ to a bucket $\mathcal{B}(i', j')$ with $i > i'$. The redistribution process touches each element of a bucket twice, when it is read out of the current bucket and when it is written to its new bucket according to the redistribution rule. This amounts for two scans of the current bucket; thus the total number of I/Os is linear in the size of the current bucket and $O(1/B)$ amortized I/Os per moved element suffice. The CPU time is $O(\log(hr))$, since that time amount is required to find the first non-empty bucket (by using $\mathcal{Q}$). Consequently, the cost we spend for each element, from its insertion up to its deletion (as minimum), is $O(h \log (hr))$ time and $O(h/B)$ I/Os.

It remains to determine the appropriate values for r and h that allow the R–heap data structure to work efficiently. The constraint we previously imposed on these parameters was that $(r \cdot h + 2) \cdot B + O(r \cdot h) \leq M$. This ensured that the first block of every bucket, and the internal queue $\mathcal{Q}$, fit in internal memory. The second additive term is nearly negligible because B is large, and thus will not be taken into account. Now, since $h = \Theta(\log_r C)$, it suffices to choose the maximum value of r such that (setting $m = M/B$):

$$r \log_r C \leq m \quad \Leftrightarrow \quad \frac{r}{\log r} \leq \frac{m}{\log C} \tag{4.2}$$

Setting $a := \frac{m}{\log C}$, the solution of equation 4.2 has the following form:

$$r = a \cdot (\log a + \log (a + \log (a + \log (\ldots))))$$

Thus, we set

$$r = \Theta(\frac{m}{\log C}\log\frac{m}{\log C}) \tag{4.3}$$

From the previous discussions we derive the following result:

Theorem 4.1.2. *Let* $m = M/B$, $r = \Theta(\frac{m}{\log C}\log\frac{m}{\log C})$ *. An* `insert` *into an external R–heap takes amortized* $O(1/B)$ *I/Os and* $O\log(r\log_r C))$ *amortized CPU time. A* `delete_min` *takes* $O((1/B)\log_r C)$ *amortized I/Os and* $O((\log_r C)\cdot\log(r\log_r C))$ *amortized CPU time.*

Proof. According to the discussion above $O(h\log(hr))$ time and $O(h/B)$ I/Os are required to manage a single element from its insertion up to its deletion from the queue (as minimum element). Upon insertion, each element get $O(h/B)$ credits. Operation `insert` needs $O(1/B)$ credits. The cost of the redistribution process accounts for $O(h/B)$ credits and is charged to the `delete_min` operation. The bound follows by setting $h = \Theta(\log_r C)$. ☐

We remark that the amortized cost of the `delete_min` is larger than the one of the `insert`, but this is just a matter of the accounting. Nonetheless, the `delete_min` operation is so simple that it will be very fast in practice, as we experimentally prove in Section 4.3. Notice that for typical values of the parameters M, B, and C, the value of $\log_r C$ is about two or three, and thus negligible in the final performance. As far as the radix r is concerned, we observe that its setting depends on M, B, and C. To better evaluate the relationship among all these values we plot them by assuming $m = M/B = 2^x$ and $C = 2^y$ (hence $r = (2^x/y)\cdot(x - \log(y))$). See Figure 4.2 for details. External radix heaps are efficient in applications where C is small. Examples

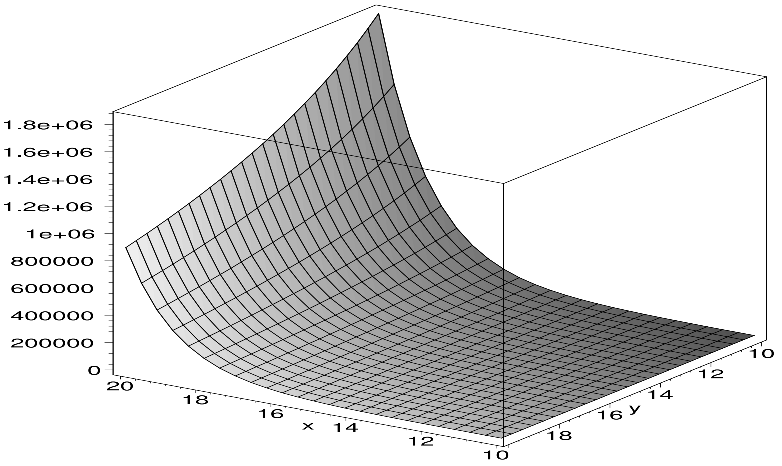


Figure 4.2: *Relationship between* r, M, B *and* C. $M/B = 2^x$ *and* $C = 2^y$*. The z-axis shows the dependancy of radix* r *on* x *and* y*.*

are time-scheduling, where C is the time at which an event takes place, or in Dijkstra's shortest path computations, where C is an upper bound on the edge weight. It is also possible to use radix heaps as an online-variant for bundle sorting [MSV00] (see Section 3.8).

As far as the disk space consumption is concerned, we observe that only *one* disk page can be non-full in each bucket (by looking at a bucket as a stack). But this page does not reside on the disk, so that there are no partially filled disk pages. We can therefore conclude that:

Lemma 4.1.3. *An external R–heap storing N elements occupies no more than N/B disk pages.*

4.1.2 D–Disk Model

Radix-heaps can be easily extended to work efficiently in the D–disk model by means of the *disk striping* technique. The D disks are not seen independently, instead we read exactly one block of each disk from the same location in a synchronized way. Thus, disk striping on D disks can be seen as using a single disk with block size $B' = D \cdot B$. If we apply this to our R–heaps, we need to change the constraint $r \cdot h \cdot B \leq M$ into $r \cdot h \cdot B \cdot D \leq M$ and store the first D blocks of each bucket internally. This implies that we can choose $r = \Theta((m/\log C) \log (m/\log C))$ with $m = M/DB$.

Theorem 4.1.4. *Let $m = M/DB$ and let $r = \Theta(\frac{m}{\log C} \log \frac{m}{\log C})$. An insert into an external R–heap takes amortized $O(1/(DB))$ I/Os and a* `delete_min` *takes $O((1/(DB)) \log_r C)$ amortized I/Os.*

We note that this bound is not optimal in the D-disk secondary memory model because it is not D-times smaller than the bound for the one-disk case. Actually, D occurs in the base of the logarithm. In Section 4.2 we will introduce a priority queue that achieves optimality in the D-disk model.

4.1.3 Implementing radix heaps

Radix heaps are implemented using LEDA-SM in a natural way. Each bucket is organized as a linked list of blocks. Constant C is specified at time of construction. The internal priority queue is implemented by a LEDA Fibonacci heap. To achieve higher throughput when reading or writing to disk, it is possible to keep a constant number c of disk blocks per bucket in main memory thus changing the condition $r \cdot h \cdot B \leq M$ to $r \cdot h \cdot cB \leq M$. Instead of linking single disk blocks, we link chains of c consecutive disk blocks. The constant c can also be specified at time of construction. Operations `insert` and `delete_min` need to compute the r-ary representations of either min or the element to be inserted. This computation can be done by a bitwise rightshift if r is a power of two. Otherwise one has to use modulo-computation if r is not a power of two. Bitshift computation is faster than modulo computation. This was tested in an early development phase of radix heaps (see [CMA$^+$98]). Therefore, r is chosen to be a power of two. We note two specialties for the actual bucket implementation. Normally, all buckets store items that are parameterized in $< P, I >$, where P is the priority data type and I is the information data type. Buckets $\mathcal{B}(0, k), k = 0, \ldots, r-1$ and buckets $\mathcal{B}(j, 0), j = 1, \ldots, h-1$ are implemented differently. In the first case, all items in a bucket $B(0,k)$ have exactly the same priority value k. Therefore it is sufficient to store only information type I. In the second case, buckets $\mathcal{B}(j,0), j = 1, \ldots, h-1$ will never store items according to equation 4.1. This holds because the j-digit in the r-ary representation is zero and must be bigger than the j-digit of min which is a contradiction. Thus it is not necessary to allocate any space for these buckets.

4.2 External Array-Heaps

Radix heaps can be used if the priority data type are non-negative integers and if the queue is *monotone*, *i.e.* if the priority values of the deleted elements form a non-decreasing sequence. For certain applications, this form of priority queue is not suitable and a more general form of priority queues without any restrictions is necessary. Furthermore, radix heaps do not achieve optimality in the D-disk model. Our second proposal (called *array heap*) is a general priority queue and it achieves optimal I/O bounds in the D-disk model.

We start with a rough description of the data structure. The heap structure is a simplification of [BK98][3] and consists of two parts: an internal heap $\mathcal{H}$ and an external data structure $\mathcal{L}$ consisting of a collection of sorted arrays of different lengths. The external part, $\mathcal{L}$, is in turn subdivided into L levels $\mathcal{L}_i$, $1 \leq i \leq L$, each consisting of $\mu = (cM/B) - 1$ arrays (called *slots*) having length $l_i = (cM)^i/B^{i-1}, c < 1$. Each slot of $\mathcal{L}_i$ is either empty or it contains a sorted sequence of at most l_i elements.

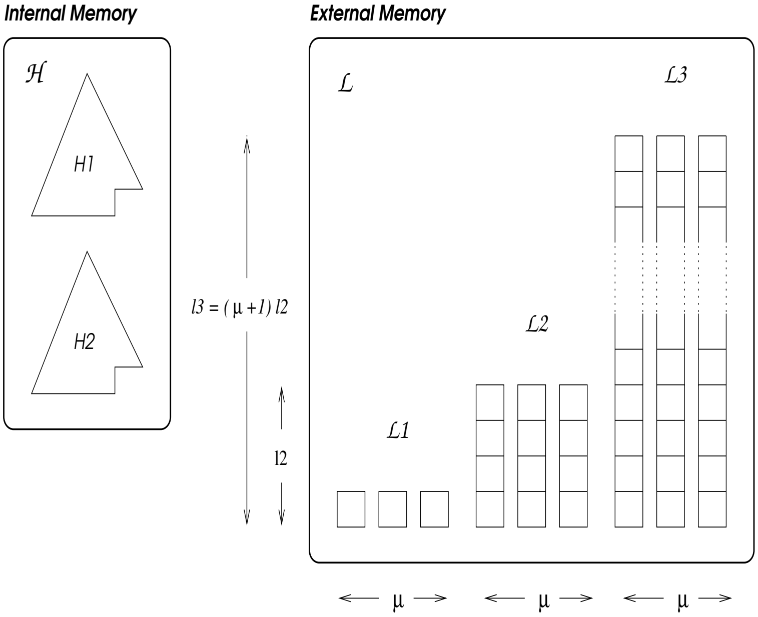


Figure 4.3: *The array heap*

---

[3] Brodal and Katajainen as well as Crauser, Ferragina and Meyer independently discovered the same heap structure.

The following property is easy to prove:

Property 4.2.1. *The total size of* $(\mu + 1)$ *slots, each containing* l_i *elements, is equal to* l_{i+1}.

Proof. $(\mu + 1)l_i = cM/B \cdot (cM)^i/B^{i-1} = (cM)^{i+1}/B^i := l_{i+1}$ □

Elements are inserted into $\mathcal{H}$; if $\mathcal{H}$ gets full, then $l_1 = cM$ of these elements are moved to the disk and stored in sorted order (according to the order defined on the keys) into a free slot of $\mathcal{L}_1$. If there is no such free slot (we call this *overflow*), we merge all the slots of $\mathcal{L}_1$ with the elements coming from $\mathcal{H}$, thus forming a sorted list which is moved to the next level $\mathcal{L}_2$. If no free slot of $\mathcal{L}_2$ exists, the overflow process is repeated on $\mathcal{L}_2$, searching for a free slot in $\mathcal{L}_3$. We continue until a free slot is eventually found.

The `delete_min` operation maintains the invariant that the smallest element always resides in $\mathcal{H}$ (see Lemma 4.2.2). Therefore $\mathcal{H}$ needs to be refilled by deleting some appropriate blocks from the sorted slots of $\mathcal{L}$ whenever the minimum element is removed from the internal heap $\mathcal{H}$ or $\mathcal{H}$ runs out of elements.

We describe now in more detail two versions of this heap. The first is a simplified non-optimal structure intended to be fast in practice (see Section 4.2.1). The second proposal (see Section 4.2.3) is slightly more complicated but reaches I/O-optimality.

4.2.1 A practical version

The internal heap $\mathcal{H}$ is divided into two parts, H_1 and H_2. H_1 stores the newly inserted elements (at most $2cM$ in total), whereas H_2 stores at most the smallest B elements of each non-empty slot in (any level of) $\mathcal{L}$. As a result, less than $cM(2 + L)$ elements reside in internal memory. Furthermore, $(\mu + 1)B = cM$ internal memory space is needed upon level overflow, in order to merge the μ slots of any level plus one sequence arriving from the previous level. This imposes that L must behave as a constant in order to ensure that we have enough internal memory to host the required data structures. We will further comment on this at the end of this section, and now just observe that for practical values of M, B and N, we have $L \leq 4$ so that the above relation can be fulfilled in practice by setting $c = 1/7$. In the following, we describe four basic tools that are used in operations `insert` and `delete_min` to manage overflows and insufficient fillings of slots.

- *Merge-Level(i,S,S')* produces a sorted sequence S' by merging the sorted sequences of the μ slots in $\mathcal{L}_i$ (including their first blocks in $\mathcal{H}$) and the sorted sequence S. This operation takes $O((|S| + l_{i+1})/B + 1) = O(l_{i+1}/B)$ I/Os.

- *Store(i,S)* assumes that $\mathcal{L}_i$ contains an empty slot and that sequence S has a length in the range $[l_i/2, l_i]$. S is stored into an empty slot of $\mathcal{L}_i$ and its smallest B elements are moved to $\mathcal{H}$, thus requiring $O(|S|/B + 1) = O(l_i/B)$ I/Os.

- *Load(i, j)* fetches the next B smallest elements from the jth slot of $\mathcal{L}_i$ into the internal heap $\mathcal{H}$. It therefore takes one I/O.

- *Compact(i)* is executed if there are at least two slots in level $\mathcal{L}_i$ for which the sum of their elements, including those in $\mathcal{H}$, is at most l_i. Then these two slots (plus their first blocks in $\mathcal{H}$) are merged into a new slot, thus freeing a slot of $\mathcal{L}_i$. S is moved to the freed slot and the smallest B elements of the new slot are moved to $\mathcal{H}$. Globally, this needs at most $O(l_i/B)$ I/Os.

Insertion. A new element is initially inserted into $\mathcal{H}$. If $\mathcal{H}$ gets full, the largest cM elements of $\mathcal{H}$ form the set S which is moved to the level $\mathcal{L}_1$ (the smallest B elements among them stay in internal memory, *i.e.* they are copied to $\mathcal{H}_2$). This insertion may influence the L levels in $\mathcal{L}$ in a cascading way. In fact at a generic step, the sorted sequence S has been originated from level $\mathcal{L}_{i-1}$ (initially $\mathcal{L}_0 = \mathcal{H}$) and must be moved to the next level $\mathcal{L}_i$. If $\mathcal{L}_i$ contains an empty slot, then S is stored into it (via *Store(i,S)*). Otherwise a further overflow occurs at $\mathcal{L}_i$ and, since all slots of $\mathcal{L}_i$ contain at least $l_i/2$ elements, they are merged with S thus forming a new sorted sequence S' (*Merge-Level(i,S,S')*). Such a sequence S' is moved to the next level and the loop above is repeated with S' playing now the role of S. Of course, at most L iterations of the loop suffice to find a free slot, because we eventually reach the last level.

Deletion. The smallest element is removed from the internal heap $\mathcal{H}$. If it was the last element coming from a slot j of some level $\mathcal{L}_i$, we load the next B elements from this slot into H_2 (*Load(i,j)*). This way, the corresponding internal buffer is refilled with the smallest B elements of that slot. In order not to use unreasonably many partially filled slots, we check the sizes of the slots after each load operation and possibly execute the *Compact* operation. *Compact* merges two partially filled slots into one. Consequently, every level $\mathcal{L}_i$ consists of at most one slot containing fewer than $l_i/2$ elements. The *Compact* operation moves the smallest B elements of the newly formed sequence to internal memory, so that the block of the slot containing the largest elements might be partially filled in external memory. This partially filled block resides in external memory.

4.2.2 Correctness and I/O-bounds

To prove correctness of our data structure we must show that during overflows, there is enough space in one of the next level to store the newly arriving elements, and we must guarantee that the minimum element is always in the internal heaps H_1 or H_2. As elements only move to levels with increasing level-number, we do not lose any elements during the overflow process. Therefore elements either reside in the internal $\mathcal{H}$ or in the external $\mathcal{L}$.

Lemma 4.2.2. *The smallest element is always stored in $\mathcal{H}$. After the execution of Store(i,S), Compact(i) and Merge-Level(i,S,S') the smallest element still resides in $\mathcal{H}$.*

Proof. The minimum element was either newly inserted and is therefore stored in H_1 or it belongs to a specific slot in some level of the external structure. As the elements in the slots are linearly ordered, the smallest element is stored in the first block of this slot and is therefore copied to H_2. Therefore, we inductively assume that the smallest elements resides in $\mathcal{H}$ before the execution of any of these operations. *Compact(i)* merges two slots of level i including the elements in H. As the slots are linearly ordered and the first B elements of the newly formed slot are moved to H, the smallest element still resides in h after the execution of *Compact(i)*. Similar arguments hold for *Merge-Level* and *Store*: linearly ordered slots (including their first elements) in H are merged into a new sequence and the smallest B elements of this sequence are moved to H. ☐

We now show that newly arriving elements can always be stored in level i. Elements come from the previous level either by executing *Merge-Level(i-1,S,S')* and then *Store(i,S')* or by only executing *Store(i,S')*.

Lemma 4.2.3. *When Store(i,S) is executed (after an overflow in level $i - 1$) we are guaranteed that S contains at least $l_i/2$ elements and at most l_i elements.*

Proof. The first inequality comes from the algorithm; *i.e.* *Store(i,S)* is only executed if the size of S is at least $l_i/2$. In the worst case, all previous levels were full, thus S was formed by a recursive merge of all previous levels including the elements from H_1. From property 4.2.1 it follows that the size of S is exactly l_i thus proving the last part of the lemma. ☐

For the remainder odf this section we assume that $cM > 3B$. We now turn on proving the I/O-bounds for operations `insert` and `delete_min`. We first observe:

Lemma 4.2.4. *After N operations the heap consists of at most $L \leq \log_{cM/B}(N/B)$ levels.*

Proof. In the worst case, no deletions occur and therefore each overflow moves the maximum possible number of elements from one level to another. Thus, the number of elements in j levels is $\sum_{i=1}^{j} \frac{(cM)^{i+1}}{B^i} < \frac{(cM)^{j+2}}{B^{j+1}}$. We must choose the smallest j so that $(cM)(cM/B)^{j+1} \geq N$. It follows that $cM(cM)^{j+1} \geq B(cM)^{j+1} \geq N$ because $cM > 3B$. Hence, $j \geq log_{cM/B}(N/B) - 1$.

☐

In a next step we determine the cost of our tools *Store*, *Compact* and *Merge-Level*.

Lemma 4.2.5. *Store(i,S) takes no more than $3 \cdot l_i/B$ I/Os. Compact(i+1) and Merge-Level(i,S,S') take no more than $3 \cdot l_{i+1}/B$ I/Os.*

Proof. When *Store(i,S)* is executed, S is read and then stored into an empty slot of $\mathcal{L}_i$ and its smallest B elements are moved to $\mathcal{H}$. This takes at most $2\lceil l_i/B \rceil \leq 3l_i/B$ I/Os since $l_i \geq l_1 = cM > 3B$. *Merge-Level(i,S,S')* produces a sorted sequence S' by merging the sorted sequences in the μ slots of $\mathcal{L}_i$ (including their first blocks in $\mathcal{H}$) and the sorted sequence S. Every block of sequence S and every block of each of the slots in $\mathcal{L}_i$ is read and written once. Therefore, the operation takes at most $2\lceil (|S| + \mu l_i)/B \rceil \leq 2\lceil (l_{i+1})/B \rceil \leq 3l_{i+1}/B$ I/Os because $l_{i+1} \geq l_1 = cM > 3B$. *Compact(i)* is executed if there are at least two slots in level $\mathcal{L}_i$ for which the sum of their elements, including those in $\mathcal{H}$, is at most l_i. Then these two slots (plus their first blocks in $\mathcal{H}$) are merged into a new slot, thus freeing a slot of $\mathcal{L}_i$. Let j and k be the slots of level $\mathcal{L}_i$ that are merged during the execution of *Compact(i)* and let e_j (resp. e_k) be the number of elements in slot j (resp. k). *Compact(i)* needs at most

$\lceil e_j/B \rceil + \lceil e_k/B \rceil + \lceil l_i/B \rceil \leq (e_j + e_k)/B + l_i/B + 3 \leq 2l_i/B + 3 \leq 3l_i/B$ I/Os since $l_i \geq l_1 = cM > 3B$.

□

We are now ready to show:

Theorem 4.2.6. *Let* $N \leq B \cdot (\frac{cM}{B})^{\frac{1}{c}-3}, 0 < c < 1/3$ *and* $cM > 3B$*. In a sequence of* N *intermixed* `insert` *and* `delete_min` *operations,* `insert` *takes amortized* $\frac{18}{B}(\log_{cM/B}(N/B))$ *I/Os and* `delete_min` *operations take* $7/B$ *amortized I/Os.*

Proof. We prove our amortized bounds by using an accounting argument. We associate with each non-empty slot j of level $\mathcal{L}_i$ a deposit $D_{i,j}$ that accounts for $6x/B$ credits, where x is the number of its empty entries. Initially, all slots are empty and thus the accounts have zero credits. Furthermore, we associate with the internal heap $\mathcal{H}$ a deposit for that we ensure to contain at least 1 credit in the case that a Load operation must be performed. This credit will be used to pay for the Load.

Each element is to be inserted with $18L/B$ credits. Each element uses $18/B$ credits from that amount everytime it moves from one level to the next one according to the *Merge-Level* operation. In case of a deletion, $7/B$ credits are left in the internal heap $\mathcal{H}$ (notice that the deleted element resided in $\mathcal{H}$ before deletion).

It remains to show that whenever an operation `insert` or `delete_min` is executed we have enough credits to pay for them. Recall that `insert` relies on operations *Merge-Level* and *Store*, whereas `delete_min` uses *Compact* and *Load*.

In the case of an overflow at level $\mathcal{L}_i$ triggered by an `insert` operation, we use *Merge-Level(i,S,S')* and then execute *Store(i+1,S')* to store the sequence S' in a free slot of level $\mathcal{L}_{i+1}$ (if any). From above, *Merge-Level(i,S,S')* needs $3l_{i+1}/B$ I/Os and *Store(i+1,S')* needs $3l_{i+1}/B$ I/Os, so that merging and storing these elements is possible in $6l_{i+1}/B$ I/Os. Since the size of S' is between $[l_{i+1}/2, l_{i+1}]$, this I/O-cost can be payed by taking $12/B$ credits from the deposit of each element affected by this moving operation. In fact, we are moving at least $l_{i+1}/2$ elements from level $\mathcal{L}_i$ to level $\mathcal{L}_{i+1}$, and thus we can employ the credits associated with their i–th level. We note that the slot of level $\mathcal{L}_{i+1}$, say j, might be partially filled after the storage of the elements and hence it is necessary to put credits in its deposit $D_{i+1,j}$ to maintain the property that $D_{i+1,j}$ accounts for the number of its empty entries. We observe that the number of empty entries $l_{i+1} - S'$ is at most $l_{i+1}/2$ so that we have to fill in $3l_{i+1}/B$ credits. Since $|S'| \geq l_{i+1}/2$ we can take $6/B$ credits from each of its elements and put them in the deposit $D_{i+1,j}$. Globally, $18/B$ credits have been used to move an element from $\mathcal{L}_i$ to $\mathcal{L}_{i+1}$, and this is the amortized cost associated with the `insert` operation for one level. In all, $18L/B$ credits suffice to pay for the cost of `insert` .

In the case of a `delete_min` operation, we leave $7/B$ credits in the deposit of $\mathcal{H}$. If a *Load* is required, its cost is paid by using $B(1/B) = 1$ credit from the deposit of $\mathcal{H}$. These credits were left by the B elements removed before that *Load*. The other $B(6/B)$ credits left by these elements, are added to the deposit of the slot j on which the *Load* operated, thus preserving the invariant on the slot account $D_{i,j}$. If the element later participates again in an overflow operation, it will re-enter the external level structure exactly in level $\mathcal{L}_i$. Therefore it still has enough credits to pay for the following *Merge-Level* operations.

Finally, we recall that *Compact(i)* needs $3l_i/B$ credits, and this can be paid by using the credits associated with the deposits of the slots to be compacted. In fact, when the compaction is executed on two slots, say j' and j'', their number of empty entries is at least l_i and hence, $6l_i/B$ credits are available in the accounts $D_{i,j'}$ and $D_{i,j''}$. When we merge the two slots, slot j'' becomes empty and slot j' contains at most $l_i/2$ empty entries. Hence we use $3l_i/B$ credits from the available $6l_i/B$ credits above to pay for the compaction, and the other $3l_i/B$ credits are used to refill the deposit of $D_{i,j'}$ (thus accounting for at most $l_i/2$ empty entries). Summing up the credits, the bound for the `delete_min` operation follows.

Since `insert` and `delete_min` operations are based upon the above four tools and their cost can be paid using the credits associated with the inserted and deleted items, the theorem follows. The bound on N, the number of elements stored in the heap, comes from the internal space bound of Lemma 4.2.8. The array heap uses $c \cdot M(3 + \log_{cM/B}(N/B))$ internal space so that if we solve the equation $c \cdot M(3 + \log_{cM/B}(N/B)) \leq M$ for N, the bound follows.

☐

Brengel [Bre00] reanalyzed the constants in his master thesis and showed that the version of array heaps that he implemented performs `insert` operations in $4/B$ I/Os and `delete_min` operations in $7/B$ I/Os.

Let us consider the space requirements of the array heap.

Lemma 4.2.7. *Every level $\mathcal{L}_i$ contains at most one slot which is non-empty and consists of less than $l_i/2$ elements.*

Proof. The jth slot in $\mathcal{L}_i$ only looses elements after the execution of *Load*(i, j). By induction, let us assume that at most one slot in $\mathcal{L}_i$ stores less than $l_i/2$ elements, say the j'th slot. After *Load*(i, j), the deletion operation checks if the number of elements, stored in slots j and j' is less than l_i. If this is the case, it calls *Compact(i)* thus emptying j and refilling j'. Otherwise, slot j' is still the only slot in level $\mathcal{L}_i$ that has less than $l_i/2$ elements and slot j must have more than $l_i/2$ elements, or we have a contraction that j' is the only slot storing less than $l_i/2$ elements.

After *Compact(i)*, no other non-empty slot in $\mathcal{L}_i$ has less than $l_i/2$ elements, so that the invariant is preserved. ☐

Lemma 4.2.8. *The total number of used disk pages is bounded above by $2(X/B) + L$, where X is the number of elements currently in the heap. The total required internal space is $cM(3 + L)$.*

Proof. Partially filled pages are created by the `delete_min` operation. Recall that a *Load* operation may trigger a *Merge-Level* or a *Compact* operation; in this case the internal-memory blocks of the slots involved by these operations are merged thus possibly causing some elements in $\mathcal{H}$ of these slots to be moved to external memory. As a consequence the smallest B of these elements remain into $\mathcal{H}$, but the moved elements might create a partially filled block in external memory. In any slot (of any level), only one page can be partially filled (the one on the top). Additionally, only one slot per level is non-empty and stores less than $l_i/2$ elements (see Lemma 4.2.7), so that in the worst case, it can consist of only one partially filled page (thus giving the term L). For all the other slots, since they are at least half full, they consist of at least one full page. Therefore, the number of their top (partially filled) pages is no more than the total number of occupied pages, which is indeed X/B. For what concerns the internal space, $\mathcal{H}_1$ stores $2cM$ elements and $\mathcal{H}_2$ stores one disk page per slot which is $(\frac{cM}{B} - 1) \cdot B \cdot L \leq cM \cdot L$. An additional amount of $(\mu + 1)B$ is used to merge the slots of any level. If we sum up, we achieve $cM(3 + L)$ thus proving the bound. ☐

We take a closer look at the internal memory constraint of Lemma 4.2.8. Assume that $M = 10^9, B = 10^6$. Thus

$$\begin{aligned} c \cdot M(3 + \log_{cM/B}(N/B)) \leq M \Leftrightarrow \log_{cM/B}(N/B) &\leq \frac{1}{c} - 3 \\ \Leftrightarrow N &\leq B \cdot (\frac{cM}{B})^{\frac{1}{c}-3} \\ \Leftrightarrow N &\leq 10^6 \cdot (c \cdot 10^3)^{\frac{1}{c}-3} \end{aligned} \tag{4.4}$$

If we fix $c = \frac{1}{7}$, it follows that:

$$N \leq 10^6 \cdot (\frac{10^{12}}{7^4}) = 0.416 \cdot 10^{15} \tag{4.5}$$

In fact, $L \leq 4$.

4.2.2.1 Reducing the working space

Our goal is to reduce the working space. We first identify, how partially filled disk pages are produced and then show, how to circumvent the problem of partially filled disk pages.

At the beginning, when a sequence S of elements is stored in the first level L_1, we assume that the size of S is a multiple of B. The first B elements of S are moved to $\mathcal{H}_2$, thus the number of remaining elements of S, that are still stored in L_1, is still a multiple of B. In the absence of the `delete_min` operation, no partially filled disk pages are produced as *Merge-Level(i,S,S')* always merges slots that contain a multiple of B elements and thus *Store(i,S)* always stores a sequence without producing any partially filled disk pages.

Disk pages become partially filled by the use of the `delete_min` operation. The first B elements (the first disk page) of slot j in level i is brought to $\mathcal{H}_2$ via *Load(i,j)*. If this page loses elements via `delete_min` and then later takes place in a *Compact* or *Merge-Level* operation before it is exhausted, we might produce a sequence of elements whose size is not divisible by B. This results in a disk page which resides in the external part $\mathcal{L}$ and which is not completely filled. Partially filled disk pages can thus be avoided if the implementation of the array heaps takes care of the following property:

Property 4.2.9. *If the elements of $\mathcal{H}_2$ do not take part in Merge-Slot and Compact operations and $\forall i : l_i \text{ mod } B = 0$ and $l_i \text{ mod } (2B) = 0$, then no partially filled disk pages are produced.*

Two implement this property we pose the following assumptions:

1. We replace l_i by l'_i and μ by μ'. We set $\mu' = \frac{cM-B}{B} - 1 = \frac{cM}{B} - 2$ and $l'_i = (cM - B)^i / B^{i-1}$
2. We assume that $(cM)\ modB = 0$ and $(cM)/B$ is odd. Both conditions are needed to ensure that for all i, l_i can be properly divided by B and $2B$.
3. Operations *Merge-Level* and *Compact* do not take into account the elements of the slots stored in $\mathcal{H}_2$.
4. *Store(i,S)* stores the sequence S is a free slot (say j). If there are still $x, 0 < x \leq B$ elements of slot j of level i in $\mathcal{H}_2$ we merge S with the x elements and keep the smallest x elements in $\mathcal{H}_2$. Otherwise, we simply store S in in slot j and move the smallest B elements to $\mathcal{H}_2$.

Assumptions 1 to 4 ensure that all pages that are stored in the external levels $\mathcal{L}_i$ are always completely filled. Assumptions 3 and 4 ensure that the `delete_min` operation does not produce partially filled pages that take place in a *Compact* or *Merge-Level* operation. Assumptions 1 and 2 guarantee that for all i: $l'_i\ mod\ B = 0$ and $l'_i\ mod\ (2B) = 0$. This can be easily proved by induction over i.

We now show that these changes do not effect correctness and that the I/O bounds of Theorem 4.2.6 hold, possibly with a change of the leading constants.
Property 4.2.1 still holds if we replace μ by μ' and l_i by l'_i. Clearly, point 4 ensures that the elements of any slot (including its elements in $\mathcal{H}_2$) are always linearly ordered so that Lemma 4.2.2 still hold. We now show why these four assumptions only change the leading constants of the I/O bounds. Clearly Lemma 4.2.4 still holds. We therefore only have to show that the assumptions 1 to 4 do not change our credit argument in the proof of Theorem 4.2.6. The important changes are that *Store(i,S)* possibly merges the sequence S with some elements in $\mathcal{H}_2$, this needs $O(|S|/B)$ I/Os, thus $\Theta(1/B)$ credits per element. The elements in $\mathcal{H}_2$ might go back to a level but only if we try to store a sequence S with smaller elements in their corresponding slot. If an element goes back, it goes back to the level $\mathcal{L}_i$ where it came from. Thus if an element reenters a level it still has enough credits left to pay for further *Merge-Level* calls. We therefore know that $O(L/B)$ credits can pay for insert. *Load(i,j)* still needs one credit and it is still true that this credit can be found in $\mathcal{H}$ and that a *Load* only follows after B `delete_min` operations. As *Store* and *Compact* got a little bit more complicated, the leading constants of `insert` and `delete_min` can increase. However, we have shown

Theorem 4.2.10. *Under assumptions 1 to 4 the simplified array heap can be implemented using X/B pages where X is the number of elements currently stored in the heap. The amortized I/O bounds for* `insert` *and* `delete_min` *are the same as in Theorem 4.2.6 besides the leading constants.*

4.2.3 An improved version

We introduce a slight modification to the external structure $\mathcal{L}$. The idea is to avoid the need of copying the first block of each non-empty slot to main memory. In fact, this is the reason why we get the additive term $L\mu B = cML$ in the internal space requirement.

We associate with each level $\mathcal{L}_i$ a slot of length l_i, called *min-buffer*$_i$. The *min-buffer*$_i$ stores the smallest l_i elements of level $\mathcal{L}_i$. The idea is that instead of moving these elements *directly* into $\mathcal{H}_2$ (as done in the simplified version with the smallest B elements of each slot), we store them into the intermediate slot *min-buffer*$_i$, and we move *only one block* of this slot into the internal heap $\mathcal{H}_2$. Indeed $\mathcal{L}_i$ moves to $\mathcal{H}_2$ just its smallest B elements, instead of the smallest B elements of each slot, namely $B\mu$ elements. Hence, $\mathcal{H}$ consists now of $2cM$ elements (the elements in $\mathcal{H}_1$) plus LB elements coming from the external structure $\mathcal{L}$.

In order to maintain the correctness of the `delete_min` operation we must ensure two facts: first, *min-buffer*$_i$ must hold a set of elements which are *always smaller* than the ones stored in the slots of $\mathcal{L}_i$; second, we must ensure that the*min-buffers* never run out of elements until the whole level gets empty.

The first requirement can be fulfilled for a given $\mathcal{L}_i$ by checking each arriving overflow from $\mathcal{L}_{i-1}$ against the elements in *min-buffer*$_i$: smaller elements are included into the sorted sequence of *min-buffer*$_i$ without changing their total number, the other elements are managed as in the previous section and they constitute to the 'moved' set S. The total number of used slots does not change in comparison to the basic algorithm, the worst-case complexity of a level-overflow increases only by a factor of *two* (two merges instead of one). The cost of these two merges can be (clearly) charged on the account of S's elements.

The second property is fulfilled by maintaining the invariant that *minbuffer*$_i$ *stores between* $l_i/2$ *and* l_i *elements if the number of elements in level* $\mathcal{L}_i$ *is bigger than* l_i*, all elements of* L_i *otherwise.* This invariant is maintained as follows:
After some `delete_min` operations all the B elements that moved from *min-buffer*$_i$ to $\mathcal{H}_2$ could have been deleted. We may therefore need to load another page of B elements from *min-buffer*$_i$. This is done by the operation *Load(i)* which constitutes the natural simplification of *Load(i,j)* described in the previous section (now it is executed only on the slot *min-buffer*$_i$). After this loading process, *min-buffer*$_i$ might contain less than $l_i/2$ elements so that a *refilling* step is needed. *Refill(i)* operation takes care of this by merging the μ slots (including *min-buffer*$_i$) in $\mathcal{L}_i$ until we computed their smallest l_i elements. The merge cost can be charged on the $l_i/2$ deletions that have been performed before *min-buffer*$_i$ became empty. *Compact(i)* must be changed because *min-buffer*$_i$ can't take place in a *Compact(i)* operation as otherwise, we might destroy the invariant that *min-buffer*$_i$ stores the smallest elements of level $\mathcal{L}_i$. We note that after a *Refill* operation, it might be necessary to execute *Compact(i)* because the *Refill* operation can produce slots that store less than $l_i/2$ elements.

The cost of the *Refill* operation can be easily charged to the cost of the `delete_min` operation. As a result, the above changes increase the amortized costs for both `insert` and `delete_min` by only a *constant factor*. As far as the space occupancy is concerned we observe that: Internally, we hold $2cM + BL$ elements. Setting $c \leq (1/2)(1 - BL/M)$ we guarantee that our internal structure $\mathcal{H}$ can be kept into internal memory. As a consequence, our bound on N also changes. Notice that $M \geq BL$ is very reasonable. This still allows to deal with problems that have exponential size in the number of blocks fitting in main memory. Externally, each element is stored in exactly one slot (possibly a *min-buffer*). Only the last block of each slot may be partially full. By an argument similar to the one adopted in Lemma 4.2.8, it easily follows that the total number of occupied pages is $2N/B + L$ over the sequence of N updates, which turns out to be asymptotically optimal. We are therefore ready to state the following result:

Theorem 4.2.11. *Under the assumption* $N \leq B \cdot (\frac{cM}{B})^{(1-3c)\frac{M}{B}}, 0 < c < 1/3$, *the new variant of the array heap occupies* $O(N/B)$ *disk pages,* `insert` *requires amortized* $O((1/B)\log_{M/B}(N/B))$ *I/Os and* `delete_min` *requires* $O(1/B)$ *amortized I/Os in a sequence of* N *intermixed* `insert` *and* `delete_min` *operations.*

4.2.4 Array-Heaps in the D-Disk Model

Since our aim is to get a simple and effective heap data structure for multiple disks, we combine array-heaps with the elegant approach of Barve, Grove and Vitter [BGV97]. Indeed, we stripe the blocks of each sorted-sequence among the disks using a *random starting point*. This idea was introduced in [BGV97] where it was used for multiway-merging in a randomized multiway mergesort approach. We discuss only the implementation of *Merge-Level(i,S,S')* and *Refill(i)* because all the other basic tools (*i.e. Store*, *Compact*, *Load*) easily generalize to the D-disk setting in an optimal way by just assuming that each slot of any level is stored in a *striped* way among the D disks.

The efficiency of the randomized mergesort algorithm of [BGV97] relies on the random starting disks for the striped runs and also on a *forecast* data structure that makes sure to have in internal memory the appropriate disk blocks when they are needed.

The following result is from [BGV97]:

Theorem 4.2.12 (Barve, Grove, Vitter). *The expected number of I/O read operations during one merge step is bounded by* $\frac{N'}{RB}\mathcal{C}(R, D)$.

This means that *to proceed by* R *blocks in the merging, the number of reads, the merging step needs, is less than the maximum when* R *blocks are randomly thrown into* D *bins*. Here $\mathcal{C}(X, D)$ is the expectation of the maximum number of balls in any bin where X is the number of thrown balls and D is the number of bins. See for example [KSC78] for balls-into-bin results.

Our main idea is therefore to use the merge-pass of that algorithm to produce just the first ℓ elements of the final sorted sequence, thus requiring only $O(\ell/(DB))$ I/Os. For *Merge-Level(i,S,S')* we set $\ell \leq l_{i+1}$, for *Refill(i)* we set $\ell = l_i$. Having this tool in our hands, we can use it to implement *Merge-Level(i,S,S')* and *Refill(i)* taking $O(l_{i+1}/(DB))$ and $O(l_i/(DB))$ *optimal I/Os*, respectively. Slight modifications to the forecast data structure will be needed in the case that the merging stops before all runs are consumed and shall be continued at some later point; this situation will occur when *Refill(i)* is invoked. We therefore save the forecast information to disk between two consecutive calls of *Refill(i)*. This does not change the asymptotic I/O bounds.

By reasonably assuming that $D = O(B)$ we get: [4]

---

[4] According to [BGV97], the number of runs merged at each round of randomized mergesort is $R = (M/B - 4D)/(2 + D/B)$ which is $\Theta(M/B)$ for $D = O(B)$; and this is optimal.

Theorem 4.2.13. *Given a sequence of N operations, the Array Heap data structure requires $O((1/(DB))\ \log_{M/B}(N/(DB)))$ amortized expected I/Os for an* `insert` *and $O(1/(DB))$ amortized expected I/Os for a* `delete_min`. *The total occupied space is $O(N/B)$ disk pages, which is optimal.*

It is also possible to use Nodine and Vitter's deterministic merge approach as proposed in GreedSort [NV95]. This leads to a similar result for the case $D = O(M/B^{\gamma}), 1/2 \leq \gamma < 1$ (see also [CFM98]).

4.2.5 Implementing array heaps

The simplified version of the array heap was implemented in the master thesis of K. Brengel [Bre00]. We review the highlights of the implementation.
$\mathcal{H}_1$ is implemented as a capacity restricted binary heap. The heap structure is implemented by arrays. Brengel implemented 5 different methods to handle an overflow of $\mathcal{H}_1$ where the $|\mathcal{H}_1|/2$ biggest elements are moved to $\mathcal{L}_1$. The first method sorts the array (this does not violate the heap property) then moves the upper half to $\mathcal{L}_1$. The second method uses a *select* algorithm to determine the median and then partitions the array into two halvess. Then it creates a heap on both halves of the array (via *heapify*) and uses the second heap (created on the upper half) to write these elements to $\mathcal{L}_1$. Methods 3 and 4 actually do not move the biggest $|\mathcal{H}_1|/2$ elements to $\mathcal{L}_1$, instead they simply move any $|\mathcal{H}_1|/2$ elements to $\mathcal{L}_1$. Note that this does not effect the correctness of the array heap. Method 3 simply sorts the last $|\mathcal{H}_1|/2$ elements, method 4 simply divides the array of the original heap into two halves, builds a heap on both halves and uses the second heap to move its elements to $\mathcal{L}_1$. The fifth method simply writes the whole heap $\mathcal{H}_1$ to level $\mathcal{L}_1$. All these methods have the same asymptotic run time of $O(|\mathcal{H}_1| \log |\mathcal{H}_1|)$ time for moving the overflow to $\mathcal{L}_1$.

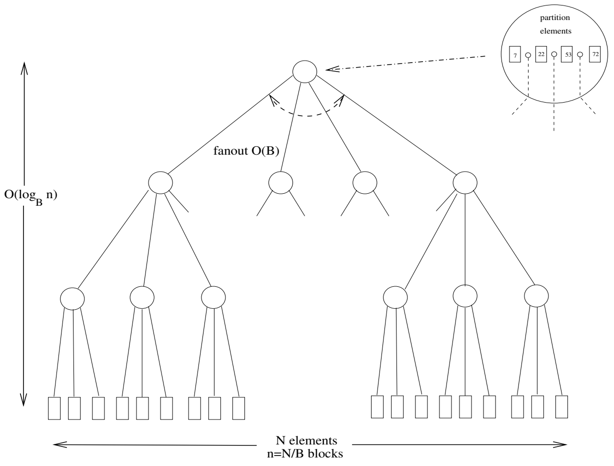
$\mathcal{H}_2$ is implemented in two different ways. The first implementation uses a list of arrays, the second method uses a list of small binary heaps. The first method supports insert in $O(1)$ and delete_min in $O(\mu)$ time, the second method supports both operations in $O(\log \mu)$ time. Thus overall 10 different combinations exist. The most efficient combination is method 1 for both, $\mathcal{H}_1$ and $\mathcal{H}_2$. This was experimentally found in the master thesis of Brengel [Bre00].

The external part consists of $\mathcal{L}_i$ levels where a level is formed of $\mu + 1$ slots of size l_i. Each slot is a linked list of disk blocks. The pointers for the link information are kept inside the disk blocks. The linked disk blocks are allocated by LEDA-SM in such a way that they are consecutive on the disk (like an array of disk blocks). Keeping the blocks additionally linked allows us to support an easy and efficient `delete_min` operation. During the `Load` operation (see Section 4.2.1), we can simply use the pointer information to easily identify the interesting disk block.

4.3 Experiments

We compare eight different PQ implementations, namely array heaps, external radix heaps, buffer trees, B-trees, Fibonacci heaps, k-ary heaps, pairing heaps and internal radix heaps. The first four priority queues are explicitly designed to work in external memory, whereas the last four ones are LEDA-implementations of well-known internal-memory priority queues.

B-trees [BM72] are a natural extension of (a, b)-trees to external memory where the fanout of each internal node is increased to $O(B)$, so that the entire node fits in a single disk page.

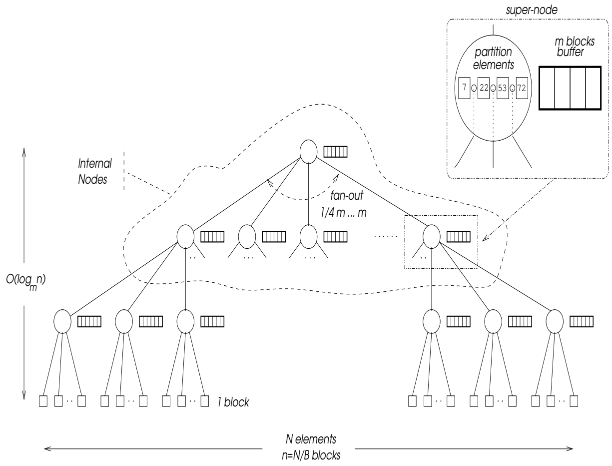


`Insert` and `delete_min` require $O(\log_B(N))$ I/Os in the worst case. B-trees are implemented as B^+-trees, *i.e.* the original items consisting of keys and information are only stored in the leaves of the tree. The internal nodes of the tree just contain routing information and pointers to their child nodes. The tree structure is implemented by linking disk blocks via pointers; these pointers are stored inside the blocks themselves.

Buffer-Trees [Arg95] are also an extension of (a, b)-trees but they are intended to reach better amortized I/O-bounds for the `insert` and `delete_min` operations, namely $O((1/B)\log_{M/B}(N/B))$ amortized I/Os. The fanout of the internal nodes is $\Theta(M/B)$, so that in practice the height is reduced to $O(\log_{M/B}(N/B))$, unlike the height $\log_B(N/B)$ of B-trees (notice that the case $M = O(B)$ does not occur in practice). Each internal node has associated a buffer of size $\Theta(M)$. Instead of storing a newly inserted element directly in the correct leaf position, the element is put into the root's buffer. When this buffer is full, we empty it and move its elements to the appropriate buffers of the internal nodes in the next level. The smallest element now resides somewhere in the buffers of the nodes on the leftmost path. The whole PQ data structure now consists of two parts: one kept in internal memory and one stored on the disk. The internal part keeps at most $\Theta(M)$ elements stored into a standard PQ (implemented by LEDA's sorted sequence); the external part contains the rest of the elements and is implemented by a buffer tree as indicated in [Arg95].[5] Newly inserted elements are checked against the ones stored in the internal part; if the internal data structure becomes full then $\Theta(M)$ of its elements are moved to the buffer tree. If the internal structure runs out of elements due to a sequence of `delete_mins`, then we empty all buffers on the leftmost path, take the smallest $\Theta(M)$ elements stored in the leftmost leaf (the overall smallest $\Theta(M)$ elements stored in the buffer tree) and store them in the internal data structure. Internal buffer tree nodes are implemented by

---

[5] Starting from the original search tree implementation, we add functionality so that the buffer tree can work as a priority queue.



an array of disk blocks; buffers are implemented by linked lists of disk blocks. The original buffer tree of Arge assumes that a buffer that is attached to an internal node may only contain M elements. The condition for the buffer size can be relaxed; it is also possible to not restrict the buffer size for internal nodes. This change was also observed by Hutchinson-et-al [HMSV97]. It speeds up `insert` at the cost of slowing down `delete_min` . Actually, after all `insert` operations, all elements are stored in the root buffer. The first `delete_min` operation constructs the whole tree and is therefore very expensive. However, early tests showed (see [CMA$^+$98]) that this is faster than the original buffer tree of Arge[Arg95]. For completeness we also give the results for the standard buffer tree implementation with a restricted buffer.

4.3.1 Experimental Setup

We use the following three tests to analyze the performance of these PQs:

Insert-All-Delete-All We perform N `insert` followed by N `delete_min` operations. This test allows us to compare the raw I/O-performance of the various PQs. The keys are `integers`, randomly drawn from the interval $[0, 10^7]$.

Intermixed insertions and deletions This test is used to check the I/O- and the CPU-speed of the internal-memory part of the external PQs. We first insert 20 million keys in the queue and then randomly perform `insert` and `delete_min` operations. An `insert` occurs with probability $1/3$ and a `delete_min` occurs with probability $2/3$.

Dijkstra's shortest-path algorithm We simulate Dijkstra's algorithm in internal memory on a large graph (using a large compute server). We use a modified Dijkstra algorithm where only `insert` and `delete_min` operations are performed on the PQ (see Section 3.8.2). We store the sequence of `insert` and `delete_min` operations executed by the algorithm on its internal PQ. This sequence is then used to test our external PQs, thus allowing us to study the behavior of the external heaps in the case of “non-random”, but application driven, update operations.

The tests were performed on a SPARC ULTRA 1/143 with 256 Mbytes of main memory and a local 9 Gbytes fast-wide SCSI disk. The internal memory priority queues use swap-space on the local disk to provide the necessary working space and a page size of 8 kbytes. The external memory priority queues used a disk block size of 32 kbytes [6]. Each external memory data structure uses about 16 Mbytes of main memory for the in-core data structures. Using only 16 out of 256 Mbytes of internal memory leaves enough space for the buffer cache of the file system. In order to have a better picture of the I/O-behavior of the experimented data structures, we decided to estimate also the actual “distribution” of the I/Os so to understand their degree of “randomness”. This is a very important parameter to be considered since it is very well known [RW94] that accessing one page from the disk in most cases decreases the cost of accessing the page succeeding it, so that *bulk* I/Os are less expensive per page than *random* I/Os. This difference becomes much more prominent if we also consider the reading-ahead/buffering/caching optimizations which are common in current disks and operating systems. In order to take into account this technical feature, without introducing new parameters that would make the analysis much more complex, we follow [FFM98] by choosing a reasonable value for the *bulk load size* and by counting the number of random and total I/Os. Since the transfer rates are more or less stable (and currently large) while seek times are highly variable and costly (because of their mechanical nature), we choose a value for the bulk size which allows to hide the *extra cost* induced by the seek step when data are fetched.

According to the accounting scheme we adopted in this paper, we need to choose a reasonable value for the *bulk load size* (notice that the parameters B and M are fixed by default according to the computer features). Since the transfer rates are more or less stable (and currently large) while seek times are highly variable and costly (because of their mechanical nature), our idea is to choose a value for the bulk size which allows to hide the *extra cost* induced by the seek step when data are fetched. This would allow us to “uniformly” access every datum stored on the disk, thus working at the highest speed allowed by its bandwidth. In some sense, we would like to *hide* the mechanical nature of the disk system [Coo]. For the disk used in our experiments, the average `t_seek` is 11 msecs, the `disk_bandwidth` is 7 Mbytes/sec. We have chosen `bulk_size` $=$ 8 disk pages for a total of 512 kbytes. It follows that `t_seek` is 24$\%$ of the total transfer time needed for a bulk I/O. Additionally, the bulk size of 256 kbytes allows us to achieve 75$\%$ of the maximum data transfer rate of our disk while keeping the service time of the requests still low. Using a page size of 32 kbytes, we still keep the service time for random blocks reasonably small and the throughput rate reasonably high.

4.3.2 Experimental Results

We comment below on the experimental results obtained for the tested priority queues. Table 4.1 summarizes the running time in seconds and the number of I/Os on the

---

[6] This value is optimal for request time versus data throughput.

insert-all-delete-all test.

`Insert/Delete_min` time performance of the external queues (in secs)

N [$*10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)	B-tree
1	6/24	18/11	56/34	62/43	11287/259
5	17/97	74/63	148/309	235/345	66210/1389
10	35/178	353/89	201/882	415/741	-
25	85/372	724/295	311/2833	1096/2302	-
50	164/853	1437/645	445/6085	3462/7592	-
75	246/1416	2157/1005	569/9880	7042/11907	-
100	325/1957	2888/1408	734/19666	12508/16909	-
150	478/3084	4277/2297	*	25051/27181	-
200	628/4036	5653/3234	*	*	-

Random/Total I/Os for external queues

N [$*10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)
1	44/420	24/720	228/668	228/668
5	422/3550	120/4560	16722/21970	13381/15501
10	1124/8620	168/9440	35993/47297	30950/35374
25	2780/21820	570/29520	93789/123285	86495/97879
50	7798/56830	1288/66160	190147/249955	179809/201921
75	12466/89370	2016/102480	286513/376625	275713/310977
100	17736/124740	2776/139760	383518/504134	381023/426615
150	27604/192500	4216/210080	*	595157/659933
200	38284/211570	5712/284320	*	

`Insert/Delete_min` time performance of the internal queues (in secs)

N [$*10^6$]	Fibonacci heap	k-ary heap	pairing heap	radix heap
1	3/32	4/33	3/19	3/11
2	6/73	8/75	6/45	5/27
5	17/208	21/210	14/126	11/71
7.5	172800*/-	32/344	22/207	18/124
10	-/-	43/482	30/291	23/162
20	-/-	172800*/-	172800*/-	172800*/-

Table 4.1: *Experimental results on the insert-all-delete-all test. In the time performance tables, the notation a/b indicates that a (resp. b) seconds were taken to perform N* `insert` *(resp. N* `delete_min`*) operations. A '-' indicates that the test was not started, x^* indicates that the test was stopped after x seconds and a $*$ indicates that the test was not started due to insufficient disk space.*

4.3.2.1 Results for Buffer Trees

Our implementation follows an idea proposed in [HMSV97] and uses big unbounded buffers instead of buffers of size M [Arg95]. This allows to save much time in the costly rebalancing process which does not occur as often as in the standard implementation. Another speed-up in our implementation is induced by the use of a simple in-core data structure (`sorted sequence` of LEDA) which reduces the CPU time. As a result, the `insert` operation is very fast as shown in Table 4.1. It is faster than array heaps, but not faster than radix heaps. However, `delete_min` is slower than that of radix heaps and array heaps, and this is due to the costly rebalancing step. If we sum up the time required for the `insert` and `delete_min` operations, we find that buffer trees are three times slower than array heaps and six times slower than radix heaps (see Table 4.1). They perform approximately four (resp. three) times the number of I/Os executed by radix heaps (array heaps), see Table 4.1. The 75% of these I/Os are random I/Os, which are mainly executed during the `delete_min` operation and are triggered by the costly rebalancing code. Overall, buffer trees are the second best "general-purpose" PQ, when no restriction is imposed on the priority value.

In a second test, we used the standard buffer tree (buffer tree orig.) with restricted internal buffer size. We immediately see that the time for operations `insert` and `delete_min` are now more balanced. However, operation `insert` gets slower as we also need to rebalance the data structure during `insert` operations. As a result the buffer tree with restricted buffer is slower than the buffer tree with unrestricted buffers but it executes slightly fewer I/Os. Hutchinson *et al.* [HMSV97] experienced similar results as ours. Earlier experiments [CM99, HMSV97] have already shown that the internal buffer size heavily influences the performance of buffer trees. Unfortunately, the optimal value of the buffer is machine dependent and must be found experimentally.

4.3.2.2 Results for B-Trees

B-Trees are not developed to work well in a priority queue setting. The worse `insert` performance of $O(\log_B N)$ I/Os leads to an experimentally large number of I/Os, large running time, and hence to an overall poor performance. The B-tree executes only random I/Os during the various operations. `delete_min` can be speeded up by caching the path leading to the leftmost child. However, the final performance does not yet reach any of the other external priority queues.

4.3.2.3 Results for R-heaps

External radix heaps are the fastest integer-based PQ. Their simple algorithmic design allows to support very fast `insert` and `delete_min` operations. There is no need to maintain an internal data structure for the minimum elements as it is required for array heaps and buffer trees. This obviously reduces the CPU time. If we sum up the time required for the `insert` and `delete_min` operations, radix heaps are about 2.5 times faster than array heaps and six times faster than buffer trees (see Table 4.1). They execute the smallest number of I/Os (see Table 4.1), and additionally only 15% of these I/Os are random. These random I/Os mainly occur during the operation `delete_min`. Unfortunately, there are two disadvantages incurred by radix heaps: they cannot be used for arbitrary key data types, and the queue must be monotone. Consequently their overall use, although very effective in practical applications, is restricted.

4.3.2.4 Results for Array Heaps

The array heap obviously needs a tricky in-core data structure to differentiate between newly inserted elements (structure H_1 is implemented by a modified binary heap) and between minimum elements coming from the external slots (structure H_2 is implemented by a set of arrays). This leads to a slowdown in the CPU-time for the operation `insert`, which is actually substantial. Indeed `insert` is up to ten times slower than in radix heaps or in buffer trees[7] even if, in the latter case, array-heaps execute a smaller number of both random and total I/Os. Hence, on our machine the `insert`

---

[7] However, array heaps are faster than the original buffer tree.

Time performance on mixed operations			
N[$*10^6$]	radix heap	array heap	buffer-tree
50	544	770	4996
75	609	945	5862
100	619	1027	6029
Random/Total I/Os on mixed operations			
50	2935/19615	22325/26997	153321/177201
75	5128/26752	24256/28384	171615/196647
100	5782/30094	24220/28380	171658/196578

Table 4.2: *Execution Time and I/Os (random/total) for the mixed operation test*

behavior of array-heaps is not I/O-bounded, but the CPU-cost is predominant (see also Section 4.4). On the other side, array heaps achieve a slightly better performance on the `delete_min` operation than radix heaps, and they result up to nine times faster than buffer trees. In this case, there is no costly rebalancing operation as it instead occurs in buffer trees. In conclusion, if we sum up the time for both update operations, we find that array heaps are more than three times faster than buffer trees and 2.5 times slower than radix heaps. Consequently, array heaps are the fastest general-purpose priority queue among the ones we have tested. If we look at the I/O-behavior, we see that array heaps perform only slightly more I/Os than radix heaps, and only about 2% of them are random. These random I/Os are quite evenly distributed between `insert` and `delete_min` operations.

4.3.2.5 Internal-Memory Priority Queues

Our tests considered standard PQs whose implementations are available in LEDA. All these data structures are very fast when running in internal memory, and among them the best choice are again the radix heaps. However, there is a dramatic degradation of their performance (see Table 4.1) when the item set is so large that it cannot be fit into the internal memory of the computer. In this case, the PQs make heavy use of the swap space on disk. This jump occurs between 5 and 7.5 million items (Fibonacci heaps) or 10 and 20 million items (radix heaps, pairing heap and k-ary heap). The early breakdown for Fibonacci heaps is due to the fact that a single item occupies a large amount of space: about 40 bytes for a 4 byte priority value and a 4 byte information value. As we expected, none of the tested internal memory PQs is a good choice for large data sets: the heavy use of pointers causes these data structures to access the external memory in an unstructured and random way, so that hardly any locality can be exploited by the underlying caching/prefetching policies of the operating system.

4.3.2.6 Mixed Operations

The test on an intermixed sequence of `insert` and `delete_min` operations is used to check the speed of the internal data structures used in the best external PQs. The results of this test are summarized in Table 4.2. We see again that radix heaps are superior to array heaps because they do not need to manage a

Time performance on the Dijkstra's test (graph fanout 3)			
N[$*10^6$]	radix heap	array heap	buffer tree
1	55	139	263
3	208	375	1350
5	423	597	2480
7	507	988	3579
10	708	1435	5331
15	1103	2202	8415
Total/Random I/Os			
1	136/1232	32/1040	2973/3557
3	598/5110	136/5360	32174/37710
5	1096/9016	256/10000	6406/72733
7	1558/12950	328/14240	95051/107483
10	2246/18774	488/24160	141016/159496
15	3500/28868	832/41440	221567/250279

Time performance on the Dijkstra's test (graph fanout 5)			
N[$*10^6$]	radix heap	array heap	buffer tree
1	95	211	657
3	422	755	2598
5	621	1313	4709
7	878	1811	6958
10	1376	2642	10694
15	2030	4115	*
Total/Random I/Os			
1	328/3080	80/3120	16156/19380
3	1028/9716	240/11040	66599/76887
5	1774/16758	456/3760	124935/143575
7	2546/23730	728/36560	185329/214537
10	3574/32998	1128/56800	275748/318268
15	6916/58772	1816/90800	*/*

Table 4.3: *Execution Time and I/Os (random/total) for the Dijkstra test applied on a random d-regular graph consisting of N nodes and dN edges, $d \in \{3, 5\}$. A $*$ indicates that the test was not started due to insufficient disk space.*

Time performance on the Dijkstra's test (graph fanout 7)			
N[$*10^6$]	radix heap	array heap	buffer tree
1	232	425	1147
3	747	1384	4069
5	1262	2368	7332
7	1897	3309	11029
Total/Random I/Os			
1	464/4648	128/5040	25486/30510
3	1310/13118	392/19280	97596/112620
5	2210/22106	752/38480	182328/211248
7	3646/33334	1136/56880	267891/314395

Table 4.4: *Execution Time and I/Os (random/total) for the Dijkstra test applied on a random d-regular graph consisting of N nodes and $7N$ edges ($d = 7$).*

complicated internal data structure, which reduces CPU-time, and also execute much fewer random (and fewer total) I/Os. This leads to a speed up of a factor 1.5. Buffer trees are more complicated and rebalancing is costly. The buffer tree is about six times slower than array heaps and seven times slower than radix heaps. They execute about nine times more I/Os and nearly all of them are random.

4.3.2.7 Tests for Dijkstra's Algorithm

We considered d-regular random graphs with N nodes and dN edges, $d \in \{3, 5, 7\}$. Edge weights are integers drawn randomly and uniformly from the interval $[1, 1000]$. Figure 4.4 shows the priority queue behavior of Dijkstra's algorithm on a random regular graph with 7 million nodes and 49 million edges ($d = 7$). The algorithm performs 96 million priority queue operations. At the beginning, we perform more insert operations than delete_min operations until the queue reaches it maximum number of elements. After that, more elements are deleted than new elements come into the queue. This behavior is typical for Dijkstra's algorithm on random d-regular graphs (see also [CMS98]).

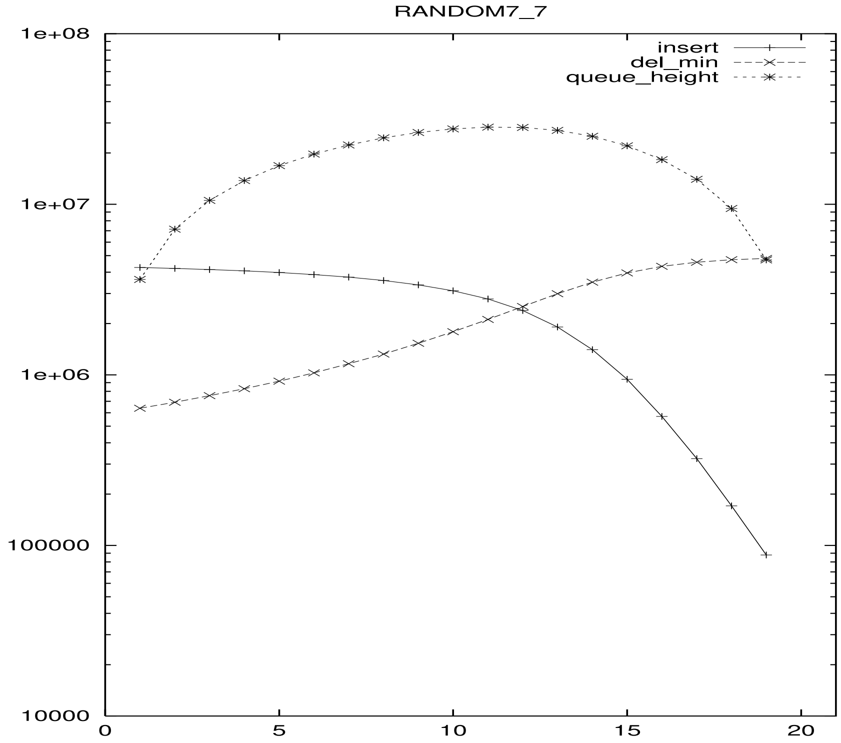


Figure 4.4: *Queue behavior on a random graph example with 7 million nodes and 49 million edges ($d = 7$).The y-axis counts the number of elements that are inserted, deleted or currently in the queue. The x-axis counts the number of rounds. Each round consists of $1/20$-th of the totally performed operations on the queue.*

Again radix heaps execute fewer total I/Os than both the other priority queues, and result in being the fastest PQ (see Tables 4.3 and 4.4). Array heaps perform the least random I/Os but they are two times slower than radix heaps because they perform approximately double the number of total I/Os and the `insert` operation is CPU-intensive. Buffer trees are four times slower than array heaps and eight times slower than radix heaps; they perform a huge amount of I/Os (compared to the other heaps) and most of these I/Os are random. Although buffer trees have many different nice applications, they are not designed to work well in a priority queue setting. We also see from this example that external radix heaps perfectly fit in Dijkstra's shortest path algorithm because they can profit from bounded edge weights (see also [AMOT90]).

4.4 Summary

We have compared eight different priority queue implementations: four of them were explicitly designed for external memory whereas the others were standard-internal queues available in LEDA. As we expected, all in-core priority queues failed to offer acceptable performance when managing large data sets. The fastest PQ turns out to be the radix heap: its simple structure allows to achieve effective I/O and time performances. Unfortunately, radix heaps are restricted to work on integer keys and monotone heaps. In the general case, array heaps become the natural choice. In the light of their good performance, it would be interesting to re-engineer this data structure concentrating the attention on its internal part in order to hopefully speed up the operation `insert` (recall that it was CPU-bounded). Another interesting topic would be to experimentally investigate the best choice for the parameters involved in the buffer-tree design (see also [HMSV97]). We believe that buffer trees can be tuned to perform better in a priority queue setting.

Chapter 5

Case Study 2: Suffix Arrays

In the information age, one of the fastest growing category of databases are the textual databases—like AP-news, Digital Libraries, Genome databases, book collections [FAFL95, RBYN00, WMB94]. Their ultimate impact heavily depends on the ability to efficiently *store* and *search* the information contained in them. Because of the continued decline in the cost of external storage devices (like disks and CD-ROMs), the storage issue is nowadays no big problem, compared to the challenges posed by the fast retrieval of the user-requested informations. In order to achieve fast retrieval of data, specialized indexing data structures and searching tools have been introduced. Their main idea is to build an *index* that allows to focus the search for a given pattern string only on a very small portion of the text collection. The improvement in the query-performance is paid by the additional space necessary to store the index. Most of the research in this field has been directed to design indexing data structures which offer a good trade-off between the query time and the space usage. The two main approaches are: *word* indexes and *full-text* indexes.

Word-indexes exploit the fact that for natural linguistic texts the universe of distinct words is small. They store all the occurrences of each word in a table that is indexed via a hashing function or a tree structure (they are usually called *inverted lists* [WMB94]). To reduce the size of the table, common words are either not indexed (*e.g.* the, at, a) or the index is later compressed. The advantage is to support very fast word (or prefix-word) queries, while a weakness is related to the inefficiency of indexing *non* natural linguistic texts, like DNA-sequences or Chinese texts [Fen97]. (For alternative approaches to word-indexes see [WMB94].)

Full-text indexes have been designed to overcome the limitations discussed above by dealing with arbitrary texts and general queries, at the cost of an increase in the additional space occupied by the underlying indexing data structure. Examples of such indexes are: suffix trees [McC76, CM96], suffix arrays [MM93] (cfr. PAT-arrays [GHGS92]), PAT-trees [GHGS92] and String B-trees [FG96]. They have been successfully applied to fundamental string-matching problems as well as text compression [BW94, Lar96], analysis of genetic sequences [Gus97] and recently to the indexing of special linguistic texts [Fen97]. General full-text indexes are therefore the natural choice to perform fast complex searches without any restrictions. The most important complexity measures for evaluating their efficiency are [BYBZ96, ZMR96]: (i) the time and the extra space required to build the index, (ii) the time required to search for a string, and (iii) the space used to store the index. Points (ii) and (iii) have been largely studied in the scientific literature [CM96, FG96, GHGS92, MM93, McC76]. In this chapter, we will investigate Point (i) by addressing the efficient *construction* of full-text indexes on very large text collections. This is a hot topic nowadays[1] because the construction phase may be a bottleneck that can even prevent these indexing tools to be used in large-scale applications. In fact, known construction algorithms are very fast when employed on textual data that fit in the internal memory of computers [AN95, MM93, Sad98, Kur99] but their performance immediately degrades when the text size becomes so large that the texts must be arranged on (slow) external storage devices [CM96, FG96].

For simplicity of exposition, we will use N to denote the size of the whole text collection and we will assume that the index is built on *only one* text, obtained by concatenating all the available texts separated by proper special characters (*i.e.* end-markers).

The most famous full-text indexing data structure is the *suffix tree* [McC76]. In internal memory, a suffix tree can be constructed in $O(N)$ time [McC76, FFMar]; in external memory, Farach *et al.* [FFMar] showed that a suffix tree can be optimally constructed within the same I/O-bound as sorting N atomic items. Nonetheless, known practical construction algorithms [CM96] for external memory still operate in a brute-force manner requiring $\Theta(N^2)$ total I/Os in the worst-case. The main limit of these algorithms is inherent in the working space, which depends on the text structure, is not predictable in advance and turns out to require between $16N$ and $26N$ bytes (assuming that $N \leq 2^{32}$ [Kur99, MM93]). [2] This makes them impractical even for moderately large text collections (consider what happens for $N \approx 100$ Mbytes, the suffix tree would occupy 1.7Gbytes !). Searching for an arbitrary string of length p takes $O(p)$ time in internal memory (which is optimal for bounded alphabets), but it does not gain any speed up from the block-transfer when the suffix tree is stored on the disk [FG96].

To circumvent these drawbacks, the *String B-tree* data structure has been introduced in [FG96]. Searching for an arbitrary pattern string of length p takes $O(p/B + \log_B N)$ random I/Os (which is optimal for unbounded alphabets). The total occupied space is asymptotically optimal and needs exactly $12.3N$ bytes. The String B-tree is a *dynamic* data structure which supports efficient update operations, but its construction from scratch on a text collection of size N takes $O(N \log_B N)$ random I/Os. Hence, space and construction time may still be a bottleneck in large-scale applications.

Since the space occupancy is a crucial issue when building and using full-text indexes on large text collections, Manber and Myers [MM93] proposed the *suffix array* data structure (cfr. PAT-array [GHGS92]), which consists of an array of pointers to text positions and thus occupies overall $4N$ bytes (thus being 4 times smaller than a suffix tree, and 3 times smaller than a String B-tree). Suffix arrays can be efficiently constructed in $O(N \log_2 N)$ time [MM93] and $O((N/B)(\log_2 N)\ \log_{M/B}(N/B))$ random I/Os [AFGV97]. Recently, Ferragina and Manzini[FM00] showed that an idea called *opportunistic data structure* allows to construct a full-text index in-core using

---

[1] Zobel *et al.* [ZMR96] say that: "*We have seen many papers in which the index simply 'is', without discussion of how it was created. But for an indexing scheme to be useful it must be possible for the index to be constructed in a reasonable amount of time, ...*".

[2] In [Kur99] the working space has been reduced to $10.1N$ to $20N$, assuming also that $N < 2^{27}$.

$O(H_k(N) + o(1))$ bits per input symbol, where $H_k(T)$ is the k-th order empirical entropy of T. Searching is possible in $O(p + occ \log^{\epsilon}(N))$ for any $\epsilon > 0$. However, it is not clear if this approach can be efficiently transfered to external memory.

In external memory, searching is not as fast as in String B-trees but it still achieves good I/O-performances. Suffix arrays have been recently the subject of experimental investigations in internal memory [MM93, Sad98], external memory [GHGS92] and distributed memory systems [KNRNZ97, NKRNZ97]. The motivation has to be probably found in their simplicity, reduced space occupancy, and in the small constants hidden in the big-Oh notation, which make them suitable for achieving reasonable performance on large text collections. Suffix arrays do not require the text to be composed of natural linguistic words, instead they allow to handle any sequence of symbols. They are also able to perform more complex queries at pattern-matching level including simple words and phrases, regular expressions, and in the presence of errors. Suffix arrays do allow to compute statistics over the indexed texts, like longest repeated substrings and even compare a text against itself to find auto-repetitions with and without errors. Suffix arrays also present some natural advantages over the other indexing data structures for what concerns the construction phase. Indeed, their simple topology (*i.e.* an array of pointers) avoids all the problems related to the efficient management of tree-based data structures (like suffix trees and String B-trees) on external storage devices [Knu81]. Additionally and more importantly, efficient practical procedures for building suffix arrays are definitively useful for efficiently constructing suffix trees, String B-trees and other indexing data structures, so that they can allow to overcome their main bottleneck (*i.e.* expensive construction phase).

With the exception of some preliminary and partial experimental work [MM93, GHGS92, NKRNZ97], to the best of our knowledge, no full-range comparison exists among the known algorithms for building large suffix arrays. This will be the main goal of this chapter[3], where we will theoretically analyze and experimentally study *seven* suffix-array construction algorithms. Some of them are the state-of-the-art in the practical setting [GHGS92], others are the most efficient theoretical ones [MM93, AFGV97], whereas *three* other algorithms are our new proposals obtained either as slight variations of the previous ones or as a careful combination of known techniques which were previously employed only in the theoretical setting. We will study these algorithms by evaluating their working space and their construction complexity both in terms of number of (random and bulk) I/Os and CPU-time. In the design of the new algorithms we will address mainly two issues: (i) simple algorithmic structure, and (ii) reduced working space. The first issue has clearly an impact on the predictability and practical efficiency of the proposed algorithms, which are also *flexible* enough to be used in distributed memory systems. In fact, since our new algorithms will be based on two basic routines—*sorting and scanning of a set of items*—they will immediately provide us with very fast suffix-array construction algorithms for D-disk array systems (thus achieving a speedup factor of approximately D [NV95]) and clusters of P workstations (thus achieving a speedup factor of approximately P [Goo99]). Additionally, our algorithms will be not faced with the problems of carefully setting some system parameters, as it happens in the results of [KNRNZ97, NKRNZ97]. The second issue (*i.e.* space usage) will be also carefully taken into account because the real disk size

---

[3] Part of this work appeared in [CF99] and [CF01].

is limited and thus a large working space could prevent the use of a construction algorithm even for moderately large text collections (see Knuth [Knu81][Sect. 6.5] "*space optimization is closely related to time optimization in a disk memory*"). Our aim will therefore be to keep the working space of our algorithms as small as possible without worsening their total running time.

We will discuss all the algorithms according to these two resources and we will pose particular attention to differentiate between random and bulk I/Os (see Farach *et al*"s disk model in Section 2.3.2). For the asymptotic analysis, this allows to take the most significant disk characteristics into account, thus making reasonable predictions on the practical behavior of these algorithms. To validate our conjectures, we will perform an extensive set of experiments based on three text collections—English texts, Amino-acid sequences and random data. As a result of the theoretical and experimental analysis, we will give a precise hierarchy of suffix-array construction algorithms according to their working-space vs. construction-time tradeoff; thus providing a wide spectrum of possible approaches for anyone who is interested in building large full-text indexes.

We further regard two issues: construction of word-indexes and worst-case performance of the BaezaYates-Gonnet-Snider's algorithm [GHGS92], one of the most effective algorithms in our proposed hierarchy. As far as the former issue is concerned, we show that our new algorithms can be successfully applied to construct word-indexes without any loss in efficiency and without compromising the ease of programming so to achieve a uniform, simple and efficient approach to both the two indexing models. The latter issue deserves much attention and is related to the intriguing, and apparently counterintuitive, "contradiction" between the effective practical performance of the BaezaYates-Gonnet-Snider's algorithm [GHGS92], verified in our experiments, and its unappealing (*i.e.* cubic) worst-case behavior. This fact motivated us to deeply study its algorithmic structure and exploit finer external-memory models [FFMar] for explaining its experimental performances. This has finally lead us to devise a *new* external-memory construction algorithm that follows the BGS's basic philosophy but in a significantly different manner, thus resulting in a novel approach which combines good practical qualities with efficient worst-case performances.

In the next section, we give some basic definitions and fix our notation. In Section 5.2, we review three well-known algorithms for constructing suffix arrays and analyze their space requirements. In Section 5.3, we describe three new external-memory algorithms for constructing suffix arrays on large text collections. In Section 5.4, we introduce our benchmark suite and discuss our experimental settings. In Section 5.5, we present and discuss the experimental results on the three data sets: Reuters corpus, Amino-acid collection and random texts. In Section 5.6, we address the problem of constructing word-indexes and show how our results can be easily extended to this indexing model. In Section 5.7, we describe a new algorithm for suffix-array construction which follows the basic philosophy of the BaezaYates-Gonnet-Snider's algorithm but in a significantly different manner, thus resulting effective both in theory and in practice.

5.1 The Suffix Array data structure

Given a string $T[1, N]$, let $T[1, i]$ denote the i-th prefix of T and $T[i, N]$ denote the i-th suffix of T. The symbol $\leq_L$ denotes the *lexicographic order* between any two strings and the symbol $\leq_i$ denotes the lexicographic order between their length-i prefixes: $S \leq_i T$ if and only if $S[1, i] \leq_L T[1, i]$.

The suffix array SA built on the text $T[1, N]$ is an array containing the lexicographically ordered sequence of text suffixes, represented via pointers to their starting positions (*i.e.* integers). For instance, if $T = ababc$ then $SA = [1, 3, 2, 4, 5]$. This way, SA occupies $4N$ bytes if $N \leq 2^{32}$. Manber and Myers [MM93] introduced this data structure in the early 90s and proposed an interesting algorithm to efficiently search for an arbitrary string $P[1, p]$ in T by taking advantage of the information coded into SA. They proved that all the *occ* occurrences of P in T can be retrieved in $O(p \log_2 N + occ)$ time in the worst-case using the plain SA, and that this bound can be further improved to $O(p + \log_2 N + occ)$ time if another array of size $4N$ is provided. If SA is stored on the disk, divided into blocks of size B, the search for P takes $O((p/B) \log_2 N + occ/B)$ random I/Os. In practical cases, $p << B$, so that $p/B = 1$. The simplicity of the search procedure, the small constants hidden in the big-Oh notation, and the reduced space occupancy are the most important characteristics that make this data structure very appealing for practical applications.

5.2 Constructing a suffix array

We now review three known algorithms for constructing the suffix array data structure on a string $T[1, N]$. We analyze their construction time in terms of CPU-time and number of I/Os (both random and bulk) in the external-memory model. We also address the issues related to their space requirements, by assuming $N \leq 2^{32}$ so that 4 bytes are sufficient to encode a (suffix) pointer. We remark that the working space of all algorithms is *linear* in the length N of the indexed text, and thus *asymptotically optimal*. However, since the constants hidden in the big-Oh notation differ a lot and since the available disk space is not unlimited, we will carefully evaluate the space usage of these algorithms in order to study their practical applicability (see Table 5.1 for a summary).

5.2.1 The algorithm of Manber and Myers

It is the fastest theoretically known algorithm for constructing a suffix array in internal memory [MM93]. It requires $O(N \log_2 N)$ worst-case time and consists of $\lceil \log_2 (N + 1) \rceil$ stages, each taking $O(N)$ time. In the first stage, the suffixes are put into buckets according to their first symbol (via radix sort). Before the generic h-th stage starts, the algorithm has inductively identified a sequence of buckets, each containing a set of suffixes. Any two suffixes in the same bucket share the first 2^{h-1} characters, whereas any two suffixes in two different buckets are $\leq_L$-sorted according to the bucket-ordering (initially, we have just one bucket containing all of T's suffixes). In stage h, the algorithm lexicographically sorts the suffixes of each bucket according to their first 2^h characters, thus forming new smaller buckets which preserve the inductive hypothesis. After the last stage, all the buckets will contain exactly one suffix,

Algorithm	Working space	CPU–time	total number of I/Os
Manber–Myers (Sect. 5.2.1)	$8N$	$N \log_2 N$	$N \log_2 N$
BGS (Sect. 5.2.2)	$8N$	$(N^3 \log_2 M)/M$	$(N^3 \log_2 M)/(MB)$
Doubling (Sect. 5.2.3)	$24N$	$N(\log_2 N)^2$	$\frac{N}{B}\,(\log_{\frac{M}{B}}(\frac{N}{B}))\,\log_2 N$
Doubl.+Discard (Sect. 5.3.1)	$24N$	$N(\log_2 N)^2$	$\frac{N}{B}\,(\log_{\frac{M}{B}}(\frac{N}{B}))\,\log_2 N$
Doubl.+Disc.+Radix (Sect. 5.3.1)	$12N$	$N(\log_2 N)^2$	$\frac{N}{B}\,(\log_{\frac{M}{B \log N}}(N))\,\log_2 N$
Doubl.+Disc.+aheap (Sect. 5.3.1)	$12N$	$N(\log_2 N)^2$	$\frac{N}{B}\,(\log_{\frac{M}{B}}(\frac{N}{B}))\,\log_2 N$
Constr. in L pieces (Sect. 5.3.3)	$\max\{\frac{24N}{L},$ $2N + \frac{8N}{L}\}$	$N(\log_2 N)^2$	$\frac{N}{B}\,(\log_{\frac{M}{B}}(\frac{N}{B}))\,\log_2 N$
New BGS (Sect. 5.7)	$8N$	$N^2(\log_2 M)/M$	(N^2/MB)

Table 5.1: *The CPU-time and the number of I/Os are expressed in big-Oh notation; the working space is evaluated exactly; L is an integer constant greater than 1. BaezaYates-Gonnet-Snider algorithm (BGS), and its new variant (called* new BGS*), operate via only disk scans, whereas all the other algorithms mainly execute random I/Os. Notice that with a tricky implementation, the working space of BGS can be reduced to $4N$.*

thus giving the final suffix array. The efficiency of this algorithm depends on the speed of the sorting step in a generic stage. Manber and Myers [MM93] showed how to perform it in linear time by using only two integer arrays, for a total of $8N$ bytes. If this algorithm is used in a virtual memory setting, it tends to perform $\Theta(N \log_2 N)$ random I/Os.

5.2.2 The algorithm of BaezaYates-Gonnet-Snider

The algorithm [GHGS92] incrementally computes the suffix array SA of the text string $T[1, N]$ in $\Theta(N/M)$ stages. Let $\ell < 1$ be a positive constant fixed below, and assume to set $\mu = \ell M$. The latter parameter will denote the size of the text pieces loaded into internal memory at each stage. We also assume for the sake of presentation that μ divides N.

At each stage, the algorithm maintains the following invariant:

At the beginning of stage h, where $h = 1, 2, \ldots, N/\mu$, the algorithm has stored an array SA_{ext} on the disk, which contains the sequence of the first $(h-1)\mu$ text suffixes ordered lexicographically and represented via their starting positions in T.

During the h–th stage, the algorithm *incrementally updates SA_{ext}* by properly inserting into it the text suffixes which start in the substring $T[(h-1)\mu + 1, h\mu]$, and by maintaining their lexicographic order. This preserves the invariant above. Hence, after all N/μ stages are executed, $SA_{ext} = SA$. We are therefore left to show how the generic h-th stage works.

The text substring $T[(h-1)\mu + 1, h\mu]$ is loaded into internal memory, and the suffix array SA_{int} containing only the suffixes starting in that text substring is built by possibly accessing the disk, if needed. [4] Then, SA_{int} is merged with the current SA_{ext} to produce the new array and to preserve the invariant. This merging process is

---

[4] The comparison between any two suffixes can require to access the substring $T[h\mu + 1, N]$, which is still on disk, thus inducing some random I/Os.

executed in two steps with the help of a counter array $C[1, \mu + 1]$. In the former step, the text T is scanned rightwards and the lexicographic position j of each text suffix $T[i, N], 1 \leq i \leq (h-1)\mu$, is determined in SA_{int} via a binary search. The entry $C[j]$ is then incremented by one unit in order to record the fact that $T[i, N]$ lexicographically lies between the $SA_{int}[j - 1]$-th and the $SA_{int}[j]$-th suffix of T. In the latter step, the information, kept in the array C, is employed to quickly merge SA_{int} with SA_{ext}: entry $C[j]$ indicates how many consecutive suffixes in SA_{ext} follow the $SA_{int}[j-1]$-th text suffix and precede the $SA_{int}[j]$-th text suffix. This implies that a simple disk scan of SA_{ext} is sufficient to perform such a merging process. At the end of these two steps, the invariant on SA_{ext} has been properly preserved so that h can be incremented and the next stage can start correctly.

Some comments are in order at this point. It is clear that the algorithm proceeds by mainly executing two disk scans: one is performed to load $T[(h - 1)\mu + 1, h\mu]$ in internal memory, and another is performed to merge SA_{int} and SA_{ext} via the counter-array C. However, some random disk accesses may be necessary in two distinct situations: either when SA_{int} is built or when the lexicographic positions of each text suffix $T[i, N]$ is determined in SA_{int}. In both cases, we may need to compare a pair of text suffixes which share a long prefix not entirely available in internal memory (*i.e.* out of $T[(h - 1)\mu + 1, h\mu]$). In the worst case, this comparison will require two sequential disk scans (initiated at the starting positions of these two suffixes) taking $O(N/M)$ bulk I/Os.

As far as the worst-case I/O-complexity is concerned, let us consider the pathological case in which we have $T = a^N$. Here, we need $O((\mu \log_2 \mu)N/\mu)$ bulk I/Os to build SA_{int}; $O((h - 1)\mu(\log_2 \mu)(N/\mu))$ bulk I/Os to compute the array C; and $O(h\mu/M) = O(h)$ bulk I/Os to merge SA_{int} with SA_{ext}. No random I/Os are executed, so that the global number of bulk I/Os is $O((N^3 \log_2 M)/M^2)$. Since the algorithm processes each loaded block, we may assume that it takes $\Theta(M)$ CPU-time to operate on each of them thus requiring $O((N^3 \log_2 M)/M)$ overall CPU-time. The total space required is $4N$ bytes for SA_{ext} and 8μ bytes for both C and SA_{int}; plus μ bytes to keep $T[(h - 1)\mu + 1, h\mu]$ in internal memory (ℓ's value is derived consequently). The merging step can be easily implemented using some extra space (indeed additional $4N$ bytes), or by employing just the space allocated for SA_{int} and SA_{ext} via a more tricky implementation. For simplicity, we adopt the former strategy.

Since the worst-case number of total I/Os is cubic, a purely theoretical analysis would classify this algorithm much less interesting than the others discussed in the following sections (see Table 5.1). But there are some considerations that are crucial to shed new light on it, and look at this algorithm from a different perspective. First of all, we must observe that in practical situations, it is very reasonable to assume that each suffix comparison usually finds the constant number of characters, needed to compare the two involved suffixes, in internal memory. Consequently, the practical behavior is more reasonably described by the formula: $O(N^2/M^2)$ bulk I/Os and $O((N^2 \log_2 M)/M)$ CPU time. Additionally, the analysis above has pointed out that all I/Os are *sequential* and that the effective number of random seeks is actually $O(N/M)$ (*i.e.* at most a constant number per stage). Consequently, the algorithm takes fully advantage of the large bandwidth of current disks and of the high computation-power of current processors [Coo, RW94]. Moreover, the reduced working space facilitates the prefetching and caching policies of the underlying operating system (remember Knuth's quote cited in this chapter's introduction) and finally, a careful look to the algebraic calculations shows that the constants hidden in the big-Oh notation are very small. As a result, the adopted accounting scheme does not label the BGS-algorithm as "worse" but drives us to conjecture good practical performances, leaving the final judgment to depend on disk and system specialties. These aspects will be addressed in Section 5.5.

Implementation Issues. The implementation of this algorithm is tricky if one wants to avoid various pathological cases occurring in the merging step of SA_{int} and SA_{ext}, and in the construction of SA_{int}. These cases could force the algorithm to execute many I/Os that would *apparently* be classified as random (according to the accounting scheme of Farach *et al.*) but which are likely to be buffered by the system and thus can be executed much faster. Our main idea is twofold:

- The array SA_{int} is built only on the first $\mu - B$ suffixes of $T[(h-1)\mu + 1, h\mu]$, thus we discard the last B suffixes (one page) of that text piece. These discarded suffixes will be (re)considered in the next stage. In such a way, during the construction of SA_{int}, each suffix is guaranteed to have a *prefix of at least* B characters available in internal memory, hence *significantly* reducing the probability of a page fault for a suffix comparison.
- To merge SA_{int} and SA_{ext} we need to load the suffixes starting in $T[1, (h-1)\mu]$ and to search them in SA_{int}. These suffixes are loaded via a rightward scan of T, which brings text pieces into internal memory, whose size is *twice* the size of a bulk I/O. Then, only the suffixes starting in the *first half* of this piece are searched in SA_{int}, thus guaranteeing to have in internal memory at least cM characters for their comparison.

These two simple tricks avoid some *"border cases"* which are very likely to induce random I/Os and that instead can be easily canceled via a careful programming. We experimented a dramatic reduction in the total number of random I/Os and consequently a significant speedup in the final performance of the implemented BGS-algorithm (see Section 5.5).

5.2.3 The doubling algorithm

This algorithm was introduced in [AFGV97] as a variant of the *labeling technique* of [RMKR72], properly adapted to work in external memory. The main idea is first to *logically* pad T with N *special* blank characters which are smaller than any other text character; and then to assign names (*i.e.* small integers) to the power-of-two length substrings of $T[1, 2N]$ in order to satisfy the so called *lexicographic naming property*:

Property 5.2.1. *Given two substrings* α *and* β *of length* 2^h *occurring in* T*, it is* $\alpha \leq_L \beta$ *if and only if the name of* α *is smaller than the name of* β*.*

These names are computed inductively by exploiting the following observation:

Observation 5.2.2. *The lexicographic order between any two substrings* α, β *of length* 2^h *can be obtained by splitting them into two equal-length parts* $\alpha = \alpha_1\alpha_2$ *and*

$\beta = \beta_1\beta_2$, and by using the order between the two pairs of names inductively assigned to (α_1, α_2) and (β_1, β_2).

After $q = \lceil \log_2 (N+1) \rceil$ stages, the algorithm has computed the lexicographic names for the 2^q-length substrings of $T[1, 2N]$ starting at positions $1, \ldots, N$ (where $2^q \geq N$). Consequently, the order between any two text suffixes, say $T[i, N]$ and $T[j, N]$, can be obtained in constant time by comparing the lexicographic names of $T[i, i + 2^q - 1]$ and $T[j, j + 2^q - 1]$. This is the rationale behind the doubling algorithm in [AFGV97], whose implementation details are sketched below.

At the beginning, the algorithm scans the string $T[1, N]$ and creates a list of N tuples each consisting of four components, say $\langle 0, 0, i, T[i] \rangle$ (the third component is fixed throughout the algorithm). [5] During all $q = \lceil \log_2 (N+1) \rceil$ stages, the algorithm manipulates these tuples by preserving the following invariant:

> *At the beginning of stage h (initially $h = 1$), tuple $\langle *, *, j, n_j \rangle$ keeps some information about the substring $T[j, j + 2^{h-1} - 1]$ and indeed n_j is its lexicographic name.* [6]

After the last q-th stage, the suffix array of T is obtained by executing two steps: (i) sort the tuples in output from stage q according to their fourth component; (ii) construct SA by reading from left-to-right the third component of the tuples in the ordered sequence.

We are therefore left with showing how stage h can preserve the invariant above by computing the lexicographic names of the substrings of length 2^h with the help of the names inductively assigned to the 2^{h-1}-length substrings. This is done in four steps as follows.

1. The N tuples (in input to stage h) are sorted according to their third component, thus producing a list such that its i-th tuple has the form $\langle *, *, i, n_i \rangle$ and thus keeping the information regarding the substring $T[i, i + 2^{h-1} - 1]$.
2. This list is scanned rightwards and the i-th tuple is substituted with $\langle n_i, n_{i+2^{h-1}}, i, 0 \rangle$, where $n_{i+2^{h-1}}$ is the value contained in the fourth component of the $(i + 2^{h-1})$-th tuple in the list. [7]
3. The list of tuples is sorted according to their first two components (lexicographic naming property).
4. The sorted sequence is scanned rightwards and different tuples are assigned (in their fourth component) with *increasing* integer numbers. This way, the lexicographic naming property is preserved for the substrings of T having length 2^h.

---

[5] The blank characters are only *logically* appended to the end of T.

[6] In what follows, the symbol $*$ is used to denote an arbitrary value for a component of a tuple, which is actually not important in the discussion.

[7] The rationale behind this step is to represent each substring $T[i, i + 2^h - 1]$ by means of the lexicographic names assigned to its prefix and its suffix of length 2^{h-1}. These names are inductively available at the beginning of stage h.

The correctness of this approach immediately derives from the lexicographic naming property and from the invariant preserved at every stage h which actually guarantees that:

Property 5.2.3. *Each tuple* $\langle *, *, i, * \rangle$ *contains some compact* lexicographic information *about the* 2^h*-length prefix of the suffix* $T[i, N]$.

As far as the I/O complexity is concerned, each stage applies twice a sorting routine (steps 1 and 3) and twice a scanning routine (steps 2 and 4) on a sequence of N tuples. The total number of random I/Os is therefore $O(sort(N)\ \log_2 N)$, and the total number of bulk I/Os is $O((N/M)\ \log_2 N)$, where $sort(N) = (N/B) \log_{M/B}(N/B)$ is the (random) I/O-complexity of an optimal external-memory sorting algorithm [Vit98]. The CPU-time is $O(N \log_2^2 N)$ since we perform $O(\log_2 N)$ sorting steps on N items. As far as the space complexity is concerned, this algorithm sorts tuples of four components, each consisting of an integer (*i.e.* 4 bytes). Hence, it seems that 16 bytes per tuple are necessary. Instead, by carefully redesigning the code it is possible to save one entry per tuple, thus using only 12 bytes[8]. In summary, the total space complexity is $24N$ bytes because the implementation of the multiway mergesort routine [Knu81], used in our experiments to sort the tuples, needs $2Xb$ bytes for sorting X items of b bytes each (see Section 5.4).

In [WMB94], a variant of the multiway mergesort is discussed that uses $NX + (N/B)\epsilon$ bytes to sort N items of size X bytes each ($\epsilon > 0$). Although this approach is I/O-optimal, we do not believe that it will outperform the standard mergesort approach. The reason for this is the fact that the pattern of disk accesses is distributed randomly and the algorithm produces a linked list of sorted disk blocks as output. Hence, the approach introduces a lot of random I/Os in order to save space. Furthermore it is questionable if this approach is transferable to the D-disk model. We will use different approaches in Section 5.3 to reduce the working space.

Two practical improvements are still possible. The first improvement can be obtained by coding four consecutive characters of T into one integer before the first stage is executed. This allows to save the first two stages and hence overall four sorting and four scanning steps. This improvement is not negligible in practice due to the time required by the sorting routine (see the sorting benchmarks in Section 3.8). The second improvement comes from the observation that it is not necessary to perform $\Theta(\log_2 N)$ iterations, instead the doubling process can be stopped as soon as all the 2^h-length substrings of T are different (*i.e.* all tuples get different names in step 4). This modification does not change the worst-case complexity of the algorithm, but it ensures that only six stages are usually sufficient for natural linguistic texts [CM96].

5.3 Three new proposals

In this section we introduce three new algorithms which asymptotically improve the previously known ones by offering better trade-offs between total number of I/Os and working space. Their algorithmic structure is simple because it is based only upon *sorting* and *scanning* routines. This feature has two immediate advantages: The algorithms

---

[8] One can drop the fourth entry of the tuple as it can be temporarily coded in the first and second entries.

are expected to be fast in practice because they can benefit from the prefetching of the disk [Coo, RW94]; they can be easily adapted to work efficiently on D-disk arrays and clusters of P workstations. It suffices to plug-in proper sorting/scanning routines to obtain a speed-up of a factor D [NV95] or P [Goo99] (cfr. [KNRNZ97, NKRNZ97]).

5.3.1 Doubling combined with a discarding stage

Our first new algorithm is based on the following observation:

Observation 5.3.1. *In each stage of the doubling approach, all tuples are considered although the final position of some of them in* SA *might already be known.*

Therefore all those tuples could be discarded from the succeeding sorting steps, thus reducing the overall number of operations, and hence I/Os, executed in the next stages (cfr. [Sad98]). Although this discarding strategy does not determine an asymptotic speed up on the overall performance of the algorithm, it is nonetheless expected to induce a significant improvement on experimental data sets because it tends to reduce the number of items on which the sorting/scanning routines are required to work on. [9]

The main idea is therefore to identify in step 4 of the doubling algorithm (see Section 5.2.3), all those tuples whose final lexicographic position can be inferred using the available information. These tuples are then discarded from the next stages. However, these tuples cannot be completely excluded because they might be necessary in step 1 of the succeeding stages in order to compute the names of longer prefixes of suffixes whose position has not yet been established. In what follows, we exploit Property 5.2.3 to cope with this problem.

As in the original doubling algorithm, we assume that a tuple has three entries (the fourth one has been dropped, see Section 5.2.3). We call a tuple **finished** if its second component is set to -1. The new algorithm inductively keeps two lists of tuples: `FT` (finished tuples) and `UT` (unfinished tuples). The former is a sorted list of tuples corresponding to suffixes whose final position in SA is known; they have the form $\langle pos, -1, i\rangle$, where pos is the final position of suffix $T[i, N]$ in SA (*i.e.* $SA[pos] = i$). `UT` is a list of tuples $\langle x, y, i\rangle$, corresponding to suffixes whose final position is not yet known. Here, $x, y \geq 0$ denote *lexicographic names* and $T[i, N]$ is the suffix to which this (unfinished) tuple refers.

At the beginning, the algorithm creates the list `UT` with tuples having the form $\langle 0, T[i], i\rangle$, for $1 \leq i \leq N$, sets `FT` to the empty list and initializes the counter $j = 0$. Then, the algorithm proceeds into stages each consisting of the following steps:

1. **Sort** the tuples in `UT` according to their first two components. If `UT` is empty goto step 6.
2. **Scan** `UT`, mark the "finished" tuples and assign new names to all tuples in `UT`. Formally, a tuple is "finished" if it is preceded and followed in `UT` by two tuples which are different in at least one of their first two components. In this case, the algorithm marks the current tuple "finished" by setting its second component to

---

[9] Knuth [Knu81][Sect. 6.5] says: "*space optimization is closely related to time optimization in a disk memory*".

-1. The new names for all tuples of `UT` are computed differently from what it was done in step 4 of the doubling algorithm (see Section 5.2.3). Indeed, the first component of a tuple $t = \langle x, y, * \rangle$ is now set equal to $(x + c)$, where c is the number of tuples that precede t in `UT` and have the form $\langle x, p, * \rangle$ with $p \neq y$.

3. **Sort** `UT` according to the third component of its tuples (*i.e.* according to the starting position of the corresponding suffix).
4. **Merge** the lists `UT` and `FT` according to the third component of their tuples. `UT` will keep the final merged sequence, whereas `FT` will be emptied.
5. **Scan** `UT` and for each *unfinished* tuple $t = \langle x, y, i \rangle$ (with $y \neq -1$), take the next tuple at distance 2^j (say $\langle z, *, i + 2^j \rangle$) and change t to $\langle x, z, i \rangle$. If a tuple is marked "finished" (*i.e.* $y = -1$), then discard it from `UT` and put it into `FT`. Finally set $j = j + 1$ and go to step 1.
6. (`UT` is empty) `FT` is sorted according to the first component of its tuples. The third components of the sorted tuples, read rightwards, form the final suffix array.

The correctness can be proved by the following invariant:

Property 5.3.2. *At a generic stage j $(j \geq 0)$, the execution of step 2 ensures that a tuple $t = \langle x, y, i \rangle$ satisfies the property that x is the number of T's suffixes whose prefix of length 2^j is strictly smaller than $T[i, i + 2^j - 1]$.*

Proof. Before step 2 is executed, x inductively accounts for the number of suffixes in T whose 2^{j-1}-length prefix is lexicographically smaller than the corresponding one of $T[i, N]$. At the first stage ($j = 0$), the algorithm has indeed safely set the first component of each tuple to 0. When step 2 is executed and tuple t is processed, the variable c accounts for the number of suffixes whose 2^{j-1}-length prefix is equal to $T[i, i + 2^{j-1} - 1]$ but their 2^j-length prefix is smaller than $T[i, i + 2^j - 1]$. From the inductive hypothesis on the value of x, it then follows that the new value $(x + c)$ correctly accounts for the number of suffixes of T whose 2^j-length prefix is lexicographically smaller than the corresponding one of $T[i, N]$. ☐

The logic underlying the algorithm above is similar to the one behind the original doubling algorithm (see Section 5.2.3). However, the new names are assigned by following a completely different approach, which does not only guarantee the lexicographic naming property but also a proper coding of some useful information (Property 5.3.2). This way when a tuple is marked "finished", its first component correctly determines the final suffix array position of the corresponding suffix (denoted by its third component). Therefore, the tuple can be safely discarded from `UT` and put into `FT` (step 5).

Doubling combined with the discarding strategy performs $O(\frac{N}{B}(\log_{\frac{M}{B}}(\frac{N}{B})\log(N))$ random I/Os, $O((N/M)\log_2 N)$ bulk I/Os, and occupies $24N$ bytes (see Section 5.2.3), exactly the same I/O-complexity as the Doubling algorithm. In our implementation, we will also use the compression scheme discussed at the end of Section 5.2.3, to save the first two stages and thus four sorting and four scanning steps. As conjectured at the beginning of this section, we expect that the discarding strategy induces a significant speed-up in the practical performance of the Doubling algorithm.

5.3.2 Doubling+Discard and Priority Queues

Although the doubling technique gives the two most I/O-efficient algorithms for constructing large suffix arrays, it has the major drawback that its working space is large (*i.e.* $24N$ bytes) compared to the other known approaches (see also Table 5.1). This is due to the fact that it uses an external mergesort [Knu81] to sort the list of tuples and this requires an auxiliary array to store the intermediate results. Our new idea is to reduce the overall space requirements by making use of some of the external priority queues, introduced in Chapter 4.

External radix heaps (see Section 4.1) use an exact number N/B of disk pages to store N items (see Lemma 4.1.3). Unfortunately, the I/O complexity of radix heaps depends on value C, *i.e.* all elements in the queue must have priority values in the range $[min, \ldots, min + C]$ where min is the minimal priority value, currently stored in the queue , and C is a constant that must be specified in advance. Hence radix heaps are space efficient but their I/O-performance degenerates when C is large. Our new construction algorithm replaces the mergesort in the Doubling+Discard algorithm (see steps 1 and 3 in Section 5.3.1) with a sorting routine based on an external radix heap. This reduces the overall required space to $12N$ bytes, but at the cost of increasing the I/O–complexity to $O((N/B)\ (\log_{M/(B \log_2 N)} N))\ \log_2 N)$ random I/Os (and $O((N/M) \log_2 N)$ bulk I/Os), because $C = N$ in the step 3 of Section 5.3.1 (the third component of the sorted tuples ranges in $[1, N]$). We observe that this algorithm should outperform the doubling approach during the first stages because the range of assigned names, and thus the value of C, is sufficiently small to take advantage from the radix-heap structure. On the other hand, the algorithm performance degenerates as more stages are executed because C becomes larger and larger. It is therefore interesting to experimentally investigate this solution since it significantly saves space and is expected to behave well in practice even in the light of the reduction in the number of tuples to be sorted in each stage. Probably, this reduction compensates the time increase of the radix heap approach and thus it is worth to be experimented on real data (see Section 5.5).

Array heaps (see Section 4.2) are able to store N elements using $N/B + \log_{M/B}(N/B)$ disk pages (see Theorem 4.2.10). This is worse compared to radix heaps but still nearly optimal. In contrast to radix heaps, there is no restriction on the priority values so that insert runs in $O((1/B) \log_{M/B}(N/B))$ I/Os and delete_minimum in $O(1/B)$ I/Os. It is not necessary to use the space-improved version of array heaps as the secondary memory usage (see Lemma 4.2.8) reduces to N/B for the special case of how array heaps are used in the construction of suffix arrays. Actually, as we first perform all insert operations and then all delete_min operations, we do not produce any partially filled pages. Thus, this approach reduces the overall required space to 12 N bytes and maintains the original I/O cost of $O((N/B)(\log_{M/B}(N/B)) \log_2(N))$. It is therefore worth to experimentally explore whether this solution is able to reduce both the space and the construction time if compared to the doubling+discard variant that uses a mergesort approach.

5.3.3 Construction in L pieces

The approaches described before are I/O-efficient but they use at least $8N$ bytes of working space. If the space issue is a primary concern, and we still wish to keep the total number of I/Os small, different approaches must be devised that require much fewer space but still guarantee good I/O-performance [10]. In this section, we describe one such approach which improves over all previous algorithms in terms of both I/O complexity, CPU time and space occupancy. It constructs the suffix array into *pieces of equal size* and thus it turns out to be useful either when the underlying application *does not* need the suffix array as a unique data structure, but allows to keep it in a distributed fashion [BCF$^+$99], or when we operate in a distributed-memory environment [KNRNZ97, NKRNZ97].

The most obvious way to achieve this goal might be to partition the original text string $T[1, N]$ into equal-length substrings and then apply any known suffix-array construction algorithm on these pieces. However, this approach should cope with the problem of correctly handling the suffixes which *start close to the border* of any two text pieces. To circumvent this problem, some authors [KNRNZ97, NKRNZ97] reduce the suffix array construction process to a string sorting process associating to each suffix of T, its prefix of length X, and then sorting N strings of length X each. Clearly, the correctness of this approach heavily depends on the value of X which also influences the space occupancy (it is actually NX bytes). Statistical considerations and structural informations about the underlying text might help, but anyway the choice of the parameter X strongly influences the final performance (see *e.g.* [KNRNZ97, NKRNZ97]).

The approach, we introduce below, is very simple and applies in a different way, useful for practical purposes, a basic idea known so far only in the theoretical setting (see *e.g.* [FFMar]). Let us denote by $\mathcal{A}_{sa}$ any external-memory algorithm for building a suffix array, $\mathcal{A}_{string}$ any external-memory algorithm for sorting a set of strings, and let L be a constant integer parameter to be fixed later. For simplicity of exposition, we assume that N is a multiple of L, and that T is *logically* padded with L blank characters.

The new approach constructs L suffix arrays, say $SA_1, SA_2, \ldots, SA_L$ each of size $\Theta(N/L)$. Array SA_i stores the lexicographically ordered sequence of suffixes $\{T[i, N], T[i + L, N], T[i + 2L, N], \ldots, \}$. The logic underlying our algorithm is to first construct SA_L, by using $\mathcal{A}_{sa}$ and $\mathcal{A}_{string}$, and then to derive all the others arrays $SA_{L-1}, SA_{L-2}, \ldots, SA_1$ by means of a simple external-memory algorithm for sorting *triples of integers*.

The suffix array SA_L is built in two main stages: In the first stage, the string set $S = \{T[L, 2L-1], T[2L, 3L-1] \ldots, T[N-L, N-1], T[N, N+L-1]\}$ is formed and lexicographically sorted by means of algorithm $\mathcal{A}_{string}$. In the second stage, a compressed text T' of length N/L is derived from $T[L, N + L - 1]$ by replacing each string having the form $T[iL, (i + 1)L - 1]$, for $i \geq 1$, with its *rank* in the *sorted* set S. Then, algorithm $\mathcal{A}_{sa}$ builds the suffix array SA' of T', and finally derives SA_L by setting $SA_L[j] = SA'[j] \cdot L$, for $j = 1, 2, \ldots, N/L$.

Subsequently, the other $L - 1$ suffix arrays are constructed by exploiting the following observation:

---

[10] See the footnote 9 and refer to [ZMR96] where Zobel *et al.* say that: "*A space-economical index is not cheap if large amounts of working storage are required to create it.*"

Observation 5.3.3. *Any suffix $T[i + kL, N]$ in SA_i can be seen as the concatenation of the character $T[i + kL]$ and the suffix $T[i + 1 + kL, N]$ occurring into SA_{i+1}.*

Therefore, if SA_{i+1} is known, the order between $T[i + kL, N]$ and $T[i + hL, N]$ can be obtained by comparing the two pairs of integers $\langle T[i + kL], pos_{i+1+kL}\rangle$ and $\langle T[i + hL], pos_{i+1+hL}\rangle$, where pos_s is the position of suffix $T[s, N]$ in SA_{i+1}. This immediately means that the construction of SA_i can be reduced to sorting $\Theta(N/L)$ tuples, once SA_{i+1} is known.

Sorting the short strings of length L via algorithm $\mathcal{A}_{string}$ needs $O(Sort(N))$ random I/Os and $2N + 8N/L$ bytes of storage, where $Sort(N) = (N/B)\ \log_{M/B}(N/B)$ [AFGV97]. Building the L suffix arrays SA_i takes $O(Sort(N/L)\ \log_2(N/L) + L\ Sort(N/L)) = O(Sort(N)\ \log_2 N)$ random I/Os, $O(N/M \log_2(N))$ bulk I/Os and $24N/L$ bytes. Of course, the larger the constant L, the larger is the number of suffix arrays that will be constructed, but the smaller is the working space required. By setting $L = 4$, we get an interesting algorithm for constructing large suffix arrays: it needs $6N$ working space, $O(Sort(N)\ \log_2 N)$ random I/Os and $O(N/M \log_2(N))$ bulk I/Os. Its practical performance will be evaluated in Section 5.5. Notice that this approach builds four suffix arrays, thus its query performance is slowed down by a constant factor four, but this is practically negligible in the light of suffix-array search performance.

5.4 Our experimental settings

5.4.1 An external-memory library

We implemented all algorithms discussed so far by using library `LEDA-SM`. For what concerns the implementation of our suffix-array construction algorithms, we used the external array data structure and the external sorting/scanning algorithms provided by `LEDA-SM` library. In particular, we used an implementation of multiway mergesort that needs $2Xb$ bytes for sorting X items of b bytes each. Radix heaps and array heaps are implemented as described in Sections 4.1.3 and 4.2.5. The other in-core algorithms and data structures used in our experiments are taken from the `LEDA` library. To avoid that the internal-memory size prevents the use of Manber-Myers' algorithm on large text collections, we run it in a virtual memory setting by using swap space. All other algorithms are not faced with this problem because they are directly designed to work in external memory.

5.4.2 System parameters

The computer used in our experiments is a SUN ULTRA-SPARC 1/143 with 64 Mbytes of internal memory running the SUN Solaris 2.6 operating system. It is connected to one single Seagate Elite-9 SCSI disk via a fast-wide differential SCSI controller. We used a standard Solaris file system on the local disk to provide LEDA-SM's logical disk, the disk is divided into blocks of $B = 8$ kbytes. The block size is actually small and higher transfer rates could be achieved by larger block sizes (see Section 3.7). We have chosen 8 kbytes as this is exactly the page size of the operating system. Therefore, the comparison of secondary memory algorithms and in-core algorithms is fair because the secondary memory algorithms can only take advantage of a better algorithmic structure and not of bigger block sizes. We have chosen the standard I/O (stdio) for the file system accesses because it is available on almost every system. We have designed our external-memory algorithms so that they use approximately 48 Mbytes of internal memory. This is obtained by properly choosing the internal memory size dedicated to the external memory data structures (*e.g.* external arrays and mergesort buckets). Thus, we guarantee that the studied algorithms do not incur in the *paging phenomenon* even for accessing their internal data structures and furthermore, that there is enough internal memory to keep the I/O buffers of the operating system.

5.4.3 Choosing the bulk-load size

As discussed in Section 4.3.1 we need a realistic value for the bulk size. Let the transfer time be described by the formula `t_seek + bulk_size/disk_bandwidth` [RW94] (see also Section 2.3.1). We wish that the first term is much smaller than the second one. Consequently, from one side we should increase `bulk_size` as much as possible (at maximum M/B), but from the other side a large `bulk_size` might reduce the significance of our accounting scheme because many *long* sequential disk scans could result in being shorter than `bulk_size` and thus would be counted as 'random', whereas they nicely exploit the disk caching and prefetching strategies [FFMar]. Hence a proper "tuning" of this parameter is needed according to the mechanical features of the underlying disk system.

In the disk used for our experiments, the average `t_seek` is 11 msecs, the `disk_bandwidth` is 7 Mbytes/sec. We have therefore chosen `bulk_size` $= 64$ disk pages, for a total of 512 Kbytes. It follows that `t_seek` is 15% of the total transfer time needed for a bulk I/O. Additionally, the bulk size of 512 Kbytes allows us to achieve 81% of the maximum data transfer rate of our disk while keeping the service time of the requests still low.

According to our considerations above (see also [Coo]), we think that this is a reasonable choice even in the light of the disk cache size of 1 Mbytes. Our choice is different from the one of Section 4.3.1 as we now especially focus on algorithms that perform a large number of bulk I/Os. Surprisingly, we also noticed in our experiments that this value allows us to catch the execution of numerous bulk I/Os by subroutines (*e.g.* multiway mergesort) which were defined "mainly random" in a theoretical investigation. Clearly, other values for `bulk_size` might be chosen and experimented, thus achieving different trade-offs between random/bulk disk accesses. However, the *qualitative* considerations on the algorithmic performance drawn at the end of the next section will remain mostly unchanged, thus fitting our experimental desires.

5.4.4 Textual data collections

For our experiments we collected over various WEB sites three textual data sets. They consist of:

- The Reuters corpus[11] together with other natural English texts whose size sums up to 26 Mbytes. This collection has the nice feature of presenting long repeated substrings.
- A set of amino-acid sequences taken from a SWISSPROT database[12] summing up to around 26 Mbytes. This collection has the nice feature of being an unstructured text so that full-text indexing is the obvious choice to process these data.
- A set of *randomly generated* texts consisting of three collections: one formed by texts randomly drawn from an alphabet of size 4, another formed by texts randomly drawn from an alphabet of size 16, and the last one formed by texts randomly drawn from an alphabet of size 64. These collections have two nice features: they are formed by unstructured texts, and they constitute a good test-bed to investigate the influence of the *length of the repeated substrings* on the performance of some studied algorithms. For each alphabet size we consider texts of 25 Mbytes and 50 Mbytes, thus further enlarging the spectrum of text sizes on which the algorithms are tested.

At a first look one might think that, although this chapter is on constructing suffix arrays on *large* text collections, our experimental data sets seem to be small! If we just look at their sizes, the involved numbers are actually not big (at most 50 Mbytes); but, as it will soon appear clear, our data sets are sufficiently large to evaluate/compare in a fair way the I/O-performance of the analyzed algorithms, and investigate their *scalability* in an external-memory setting. In fact, the suffix array SA needs $4N$ bytes to index a text of length N. Hence, the text plus SA globally occupy 200 Mbytes, when $N = 50$ Mbytes. Additionally, each of the algorithms discussed in our paper requires at least $8N$ bytes (the space of the BaezaYates-Gonnet-Snider algorithm) of working space; this means 400 Mbytes for the largest text size. The doubling variants either need between 600 Mbytes and 1200 Mbytes of working space. In summary, *more than* 600 Mbytes and up to 1400 Mbytes will be used during the overall construction of SA in each of the experimented algorithms, when $N = 50$ Mbytes ! Now, since 64 Mbytes is the size of the available internal memory of our computer, all the experiments will run on disk, and therefore the performance of the studied algorithms will properly reflect their I/O-behavior.

5.5 Experimental Results

We now comment the results obtained for our seven different construction algorithms: Manber and Myers' algorithm (MM), BaezaYates-Gonnet-Snider's algorithm (BGS), original doubling (Doubl), doubling with discarding (Doubl+Disc), doubling with discarding and external radix heaps (Doubl+Disc+Radix), doubling with discarding and array heaps (Doubl+Disc+aheap) and the 'construction into pieces' approach (L-pieces). The overall results are reported in Figure 5.1, and they are detailed in Tables 5.2 and 5.3 below.

---

[11] We used the text collection "Reuters-21578, Distribution 1.0" available from David D. Lewis' professional home page, currently: `http://www.research.att.com/~lewis`

[12] See the site: `http://www.bic.nus.edu.sg/swprot.html`

<table>
<tr><th colspan="8">The Reuters corpus</th></tr>
<tr><th>N</th><th>MM</th><th>BGS</th><th>Doubl</th><th>Doubl+Disc</th><th>Doubl+Disc+ radix</th><th>Doubl+Disc+ aheap</th><th>L-pieces</th></tr>
<tr><td>1324350</td><td>67</td><td>125</td><td>1201</td><td>982</td><td>1965</td><td>1288</td><td>331</td></tr>
<tr><td>2578790</td><td>141</td><td>346</td><td>2368</td><td>1894</td><td>3739</td><td>2645</td><td>582</td></tr>
<tr><td>5199134</td><td>293</td><td>1058</td><td>5192</td><td>4070</td><td>7833</td><td>6223</td><td>1119</td></tr>
<tr><td>10509432</td><td>223200</td><td>4808</td><td>11530</td><td>8812</td><td>16257</td><td>13369</td><td>2701</td></tr>
<tr><td>20680547</td><td>–</td><td>16670</td><td>28944</td><td>20434</td><td>37412</td><td>27466</td><td>5937</td></tr>
<tr><td>26419271</td><td>–</td><td>27178</td><td>42192</td><td>28937</td><td>50324</td><td>36334</td><td>7729</td></tr>
<tr><td colspan="8">The Amino-Acid Test</td></tr>
<tr><td>26358530</td><td>–</td><td>20703</td><td>37963</td><td>24817</td><td>41595</td><td>28498</td><td>6918</td></tr>
</table>

Table 5.2: *Construction time (in seconds) of all experimented algorithms on two text collections: the Reuters corpus and the Amino-acid data set.* N *is the size of the text collection in bytes. The symbol '–' indicates that the test was stopped after* 63 *hours.*

<table>
<tr><th colspan="6">The Reuters corpus</th></tr>
<tr><th>BGS</th><th>Doubl</th><th>Doubl+Disc</th><th>Doubl+Disc+ radix</th><th>Doubl+Disc+ aheap</th><th>L-pieces</th></tr>
<tr><td>120/7865</td><td>2349/256389</td><td>2242/199535</td><td>4872/377549</td><td>3689/284491</td><td>837/57282</td></tr>
<tr><td>317/20708</td><td>4517/500151</td><td>4383/395018</td><td>10075/787916</td><td>7650/590482</td><td>1693/177003</td></tr>
<tr><td>929/60419</td><td>9095/1009359</td><td>8916/809603</td><td>22466/1761273</td><td>16196/1249236</td><td>3386/360210</td></tr>
<tr><td>4347/282320</td><td>18284/2041285</td><td>18126/1655751</td><td>47571/3728159</td><td>35354/2718016</td><td>6849/730651</td></tr>
<tr><td>14377/933064</td><td>35935/4017664</td><td>35904/3293234</td><td>96292/7550794</td><td>73544/5643378</td><td>14243/1530995</td></tr>
<tr><td>24185/1568947</td><td>45911/5132822</td><td>45842/4202902</td><td>129071/10001152</td><td>95074/7292393</td><td>18178/1956557</td></tr>
<tr><td colspan="6">The Amino-Acid Test</td></tr>
<tr><td>24181/1568773</td><td>41709/4656578</td><td>39499/3539148</td><td>105956/8222236</td><td>74463/5743046</td><td>16118/1719883</td></tr>
</table>

Table 5.3: *Number of I/Os (bulk/total) of all experimented algorithms on two text collections: the Reuters corpus and the Amino-acid data set.* N *takes the same values as in Table 5.2 and is the size of text collection in bytes;* 64 *disk pages form a bulk-I/O.*

Results for the Manber-Myers' algorithm. It is not astonishing to observe that the construction time of MM-algorithm is outperformed by every other studied algorithm as soon as the working space exceeds the memory size. This worse behavior is due to the fact that the algorithm accesses the suffix array in an unstructured and unpredictable way. In fact its paging activity almost crashes the system, as we monitored by using the Solaris-tool *vmstat*. The vmstat value *sr*, which monitors the number of page scans per second, was constantly higher than 200. According to the Solaris system guide, this indicates that the system is constantly out of memory. Looking at Table 5.1, we infer that MM-algorithm should be chosen only when the text is shorter than $M/8$. In this case, the data structures fit into internal memory and thus the disk is never used. In our experimental setting, this actually happens for $N \leq 8$ Mbytes. When $N > 8$ Mbytes, the time complexity of MM-algorithm is still *quasi-linear* but the constant hidden in the big-Oh notation is very large due to the paging activity, thus making the algorithmic behavior unacceptable.

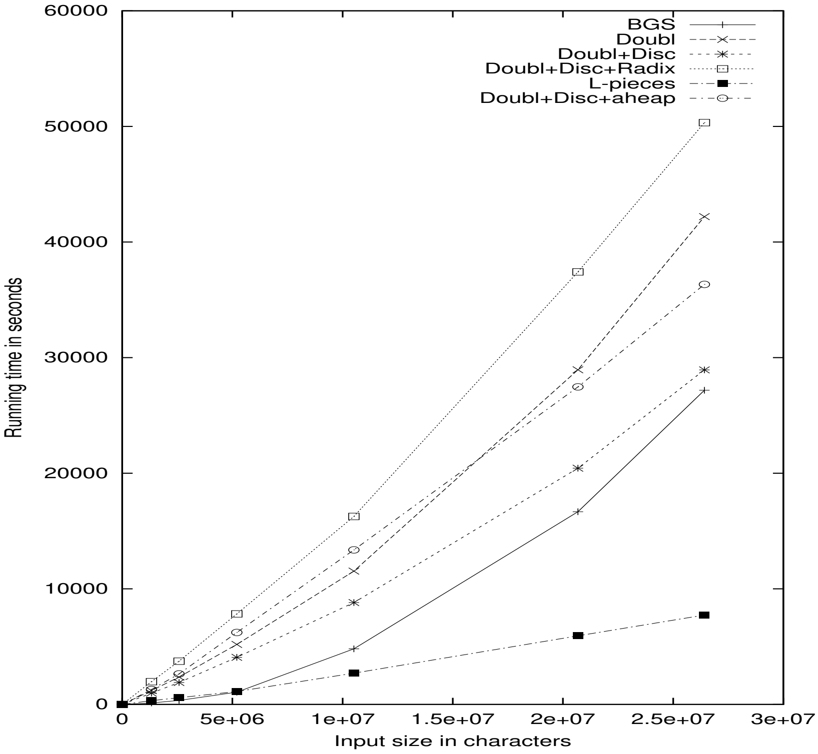


Figure 5.1: *Run-time of all construction approaches on the Reuters corpus.*

Results for the BaezaYates–Gonnet–Snider's algorithm. As observed in Section 5.2.2, the main *theoretical* drawback of this algorithm is the *cubic worst-case* complexity; but its small working space, its regular pattern of disk accesses and the small constants hidden in the big-Oh notation has led us to think favorably of BGS for practical uses as discussed in the previous sections. Moreover, we conjectured a quadratic I/O-behavior in practice because of the *short* repeated substrings which usually occur in real texts. Our experiments show that we were right with all these assumptions. Indeed, if we double the text size, the running time increases by nearly a factor of four (see Table 5.2), and the number of bulk and random I/Os also increases quadratically (see Table 5.3). The number of total and bulk I/Os is nearly identical for all data sets (Reuters, Amino-Acid and Random, see Table 5.3 and 5.5), so that the *practical behavior* is actually quadratic. Furthermore, it is not astonishing to experimentally verify that BGS is *faster* than any Doubling variant on the Reuters corpus and the Amino-Acid data set (see Figure 5.1). Consequently, it turns out to be the fastest algorithm for building a (unique) suffix array when $N \leq 25$ Mbytes. This scenario probably remains unchanged for text collections which are slightly larger than the ones we experimented; after that, the quadratic behavior of BGS will be probably no longer *"hidden"* by its nice algorithmic properties.

Using Table 5.3, we can compute that (i) only the 1% of all disk accesses are random I/Os [13] (hence, most of them are bulk I/Os !); (ii) the algorithm performs the least number of random I/Os on both the data sets when building a unique suffix array; (iii) BGS is the fastest algorithm to construct one unique suffix array, and it is the second fastest algorithm in general. Additionally, we observe that the quadratic CPU-time complexity of the BGS-algorithm heavily influences (*i.e.* slows down) its efficiency and thus I/O is not the only bottleneck.

In summary, the BGS-algorithm is amazingly fast on *medium-size* text collections, and remains reasonably fast on larger data sets. It is not the absolutely fastest on larger and larger text collections because of its quadratic CPU- and I/O-complexities. Nonetheless, the small working space (possibly $4N$ bytes via a tricky implementation) and the ease of programming make the BGS-algorithm very appealing for software developers and practitioners, especially in applications where the space issue is the primary concern.

Results for the Doubling algorithm. The doubling algorithm performs 11 stages on the Reuters corpus, hence it performs 21 scans and 21 sorting steps on N tuples, where N is the text size. Consequently, we can conclude that there is a repeated substring in this text collection of length about 2^{12} (namely we detected a duplicated article). Figure 5.1 shows that the Doubling algorithm scales well in the tested input range: [14] If the size of the text doubles, the total running time doubles too. Among all disk accesses, 41% of them are random, and the number of bulk and random I/Os is larger than the one executed by every other tested algorithm, except the Doubl+Disc+Radix variant which has higher worst-case complexity but smaller working space (see Table 5.3 and Section 5.3). Due to the high number of random I/Os and to the large working space (see Table 5.1), the Doubling algorithm is expected to surpass the performance of BGS only for *very large* values of N.

In summary, although theoretically interesting and almost asymptotically optimal, the Doubling algorithm is not appealing in practice. In fact, this motivated us to develop the two variants discussed in Section 5.3 (namely, Doubl+Disc and Doubl+Disc+Radix algorithms).

Results for the Doubl+Disc algorithm. If we add the discarding strategy to the Doubling algorithm (as described in Section 5.3.1), we achieve better performance as conjectured. The gain induced by the discarding step is approximately 32% of the original running time for both the Reuters corpus and the Amino-acid data set. If we look in detail at the number of discarded tuples (see Figure 5.3), we see that for the Reuters corpus, this is small in the first two stages, while it becomes significant in stages three and four, where nearly 55% of the tuples are thrown away. Since we

---

[13] Random I/Os are computed by means of the following formula: $IO_{random} = IO_{total} - (IO_{bulk} * 64)$.

[14] Notice that the curve for Doubling is not always linear. There is a “jump” in the running time as soon as the input has size $21 \cdot 10^6$. This is due to the system’s pager-daemon that is responsible for managing the memory-pages. During the merge-step of multiway-mergesort, the number of free pages falls below a given threshold and the pager-daemon tries to free some memory pages. This goal can’t be achieved because the mergesort constantly loads new blocks into memory. Therefore, this pager process runs almost all the time and increases the CPU-time of the merge-step of a factor of two.

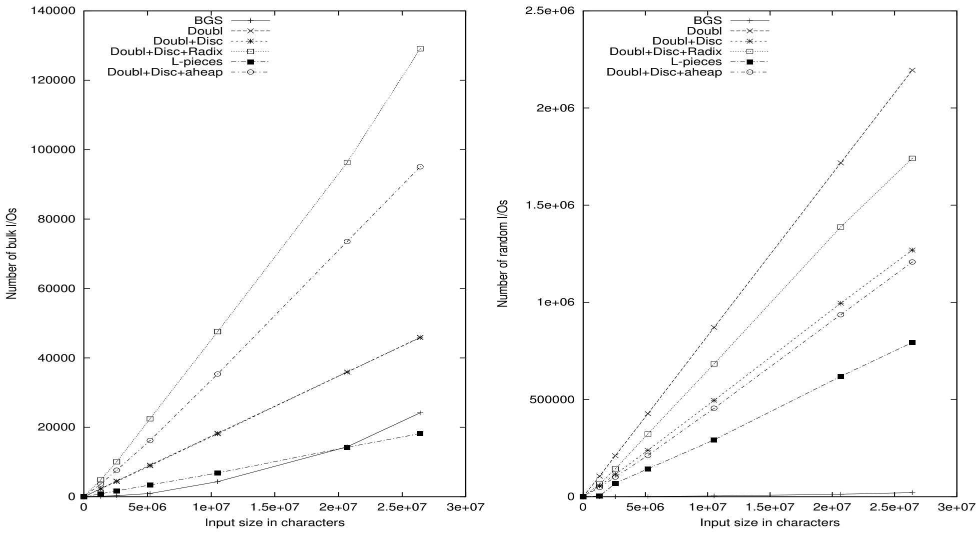


Figure 5.2: *Bulk and Random I/Os for all construction approaches. The bulk size is* 64 *disk pages.*

double the length of the substrings at each stage and we use the compression scheme of Section 5.2.3, we can infer that the Reuters corpus has a lot of substrings of length 16 to 32 characters that occur once in the collection but their prefixes of length 8 to 16 occur at least twice. We also point out that the curve indicating the number of discarded tuples is nearly the same as the size of the test set increases. This means that the number of discarded tuples is a "function" of the structure of the indexed text. For our experimental data sets, we save approximately 19% of the I/Os compared to Doubling. The percentage of random I/Os is 28%, this is much fewer than Doubling (42%), and hence discarding helps in reducing mainly the random I/Os (see also Table 5.3). The savings induced by the discarding strategy are expected to pay off much more on larger text collections, because of the significant reduction in the number of manipulated tuples at the early stages, which should facilitate caching and prefetching operations. Consequently, if the time complexity is a much more important concern than the space occupancy, the Doubl+Disc algorithm is definitively the choice for building a (unique) *very large* suffix array.

Results for the Doubl+Disc+Radix algorithm. This algorithm is not as fast as we conjectured in Section 5.3.2, even for the tuple ordering of Step 2. The reason we have drawn from the experiments is the following. Notice that radix heaps are integer priority queues, and thus we cannot keep the parameter C small (to exploit Radix Heaps properties) by defining the first two components of a tuple as its priority. Hence, the sorting in Step 2 must be implemented via two sorting phases and this naturally *doubles* the work executed in this step. Additionally, the compression scheme of Section 5.2.3 cannot be used here because it would increase C too much in the early stages. Hence, the algorithm performs two more stages than Doubl-algorithm, and

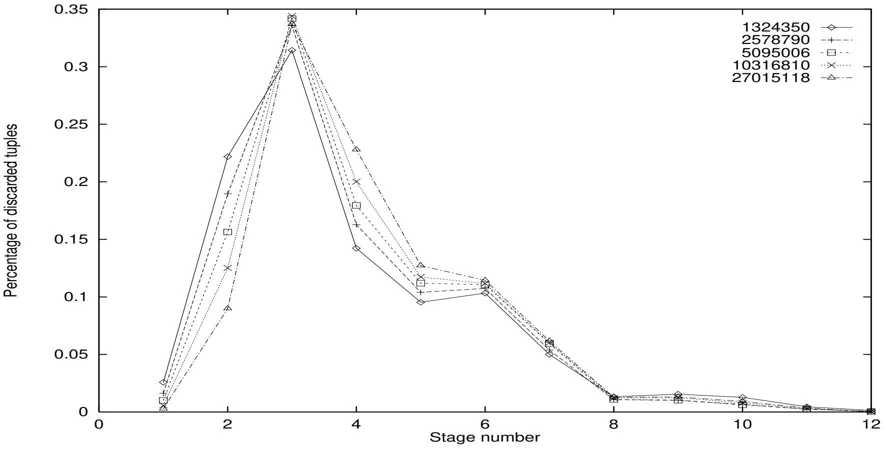


Figure 5.3: *The percentage of discarded tuples at each stage of the Doubl+Disc algorithm on the Reuters corpus.*

furthermore it does not take advantage of the discarding strategy because the number of discarded tuples in this two initial stages is very small. This way, it is not surprising to observe in Table 5.3 that the Doubl+Disc+Radix algorithm performs *twice* the I/Os of the other Doubling variants, and it is the slowest among all the tested algorithms. Hence the *compensation* between the number of discarded tuples and the increase in the I/O-complexity, conjectured in Section 5.3.2, does not actually arise in practice.

In summary, the Doubl+Disc+Radix algorithm can be interesting only in the light of its space requirements. However, if we compare space vs. time trade-off we can reasonably consider the Doubl+Disc+Radix algorithm *worse* than the BGS-algorithm because the former requires larger working space and it is expected to surpass the BGS-performance only for *very large* text collections (see Section 5.8 for further comments).

Results for Doubling+Disc+aheap algorithm. The algorithm is the second fastest Doubling+Discard variant and also the second fastest of all Doubling approaches. It executes 12 rounds as the compression scheme of Section 5.2.3 can be used. The total number of I/Os is nearly double the number of I/Os of the Doubling+disc variant (see Table 5.3) and 17% of the I/Os are random. Unfortunately, the array-heap leads to a higher I/O rate and also to a higher CPU time usage than the Doubling+Discard approach which uses a mergesort algorithm for sorting. The constants in the I/O-bounds of the array-heap operations are higher than that of multiway-mergesort (see Lemma 4.2.6 in Section 4.2.2) . Therefore, Doubl+Disc+aheap executes twice the number of total I/Os if compared to Doubl+Disc. Additionally, the internal heap structure $\mathcal{H}$ of the array heap is more complicated than the internal structure of the mergesort so that we pay this by an increase in the CPU time. Overall the Doubl+Disc+aheap approach is about 1.3 times slower than the Doubl+Disc approach. We conclude that the small number of random I/Os compensates the higher number of total I/Os and the slightly higher CPU time. The Doubl+Disc+aheap approach is a good choice if the space complexity is important and one does not want to sacrifice a lot of time.

Results for the L-pieces algorithm. We fixed $L = 4$, used multi-way mergesort for sorting short strings and the Doubl-algorithm for constructing SA_L (see Section 5.3.3). Looking at Table 5.3 we notice that 40% of the total I/Os are random, and that the algorithm executes slightly more I/Os than the BGS-algorithm. Nonetheless, as shown in Figure 5.1 this algorithm is the fastest one. It is three to four times faster than BGS (due to its quadratic CPU-time) and four times faster than the Doubl+Disc algorithm (due to the larger number of I/Os). The running time distributes as follows: 63% of the overall time is used to build the compressed suffix array SA_L; the sorting of the short strings required only 4% of the overall time; the rest is used to build the other three suffix arrays. It must be said that for our test sizes, the short strings fit in internal memory at once thus making their sorting stage very fast. However, it is also clear that sorting short strings never takes more time than *one* stage of the Doubl-algorithm. Furthermore, the construction of the other three suffix arrays, when executed entirely on disk, would account for no more than 1.5 stages of the Doubl-algorithm. Consequently, even in the case where the short strings and the other three suffix arrays reside on the disk, it is not hazardous to *conjecture* that this algorithm is still significantly faster than all the other approaches. Its only "drawback" is that it constructs the suffix array in four distinct pieces. Clearly, if the underlying text retrieval applications does not impose to have *one unique* suffix array, then this approach turns out to be de-facto *'the'* choice for constructing such a data structure (see Section 5.8 for further discussions).

Comparison of all construction approaches. We first compare all algorithms with respect to their time performance for increasing text lengths (see Figure 5.1). It is obvious from the discussions above that the MM-algorithm is the fastest one when the suffix array can be built entirely in internal memory. As soon as the working space crosses the 'memory border', there are various possible choices. The L-pieces algorithm is the natural choice whenever the splitting of the suffix array does not prevent its use in the underlying application. If instead a unique suffix array is needed, then the choice lies between the BGS-algorithm and the Doubl+Disc algorithm. For text collections which are not very big, the BGS-algorithm is preferable: it offers fast performance and very small working space. For larger text collections, the choice depends on the primary resource to be minimized: time or space ? In the former case, the Doubl+Disc algorithm is the choice; in the latter case, the BGS-algorithm is the best.

Doubling+Disc+aheap is a very interesting alternative as it allows to use only 12 N bytes and is able to construct the suffix array in a reasonable amount of time. It is only slightly slower than Doubl+Disc. but uses only half the space. We note that if we restrict N to be at most 2^{31}, this variant can be tuned to use only 8 N bytes.

With respect to the ease of programming, the BGS-algorithm still seems to be the best choice, unless the software developer has a library of general external sorting routines, in which case all Doubling variants turns out to be simple too.

5.5.1 Experiments on random data

Three important questions were left open in the previous section:

1. How do the structural properties of the indexed text influence the performance of the Doubl+Disc algorithm ? Do these properties decrease the number of stages, and thus significantly improve its overall performance ? How big is the improvement induced by the discarding strategy on more structured texts ?
2. What happens to the BGS-algorithm if we increase the text size ? Is its practical behavior "independent" of the text structure ?
3. What happens to the L-pieces algorithm when the four suffix arrays reside on disk ? Is this algorithm still significantly faster than all the other ones ?

In this section we will *positively* answer all these questions by performing an extensive set of experiments on *three* sets of textual data which are *randomly and uniformly* drawn from alphabets of size 4 (random-small), size 16 (random-middle), and size 64 (random-large). For each alphabet size, we will indeed generate *two* text collections of 25 Mbytes and 50 Mbytes.

The choice of "randomly generated data sets" is motivated by the following two observations. By varying the alphabet size we can study the impact that the *average length* of the repeated substrings has on the performance of the discarding strategy. Indeed, the smaller is this length, the larger should be the number of tuples which are discarded at earlier stages, and thus the bigger should be the speed-up obtained by the Doubl+Disc algorithm. The experiments carried out on the Reuters corpus (see Section 5.5) did not allow us to complete this analysis because of the structural properties of this text collection. In fact, the Reuters corpus represents a pathological case because it has many *long* repeated text-substrings and this is usually no typical for natural linguistic texts. This was the reason why we conjectured at the end of Section 5.5 a much larger gain from the discarding strategy on more structured texts. An early validation of this conjecture was provided by the experiments carried out on the Amino-acid data set (see Table 5.2). Now we expect that the random data collections will be a good test-bed for providing further evidence. The same thing can be said about the BGS-algorithm whose "independent behavior from the structure of the indexed text in practice", conjectured in Section 5.5, can now be tested on *various structured* texts.

Second, by varying the size of the indexed collection we can investigate the behavior of the L-pieces algorithm when the ordering of the short strings and the construction of the suffix arrays SA_1, SA_2, SA_3 operates directly on the disk (and not in internal memory). We can also test if the average length of the repeated substrings can influence the performance of $\mathcal{A}_{sa}$ when building SA_4. We notice that the construction of the other three suffix arrays is clearly not influenced by the structure of the underlying text, because it consists of just three sorting steps executed on a sequence of integer triples. In Section 5.5 we conjectured that a larger text collection should not significantly influence the overall performance of the L-pieces algorithm because, apart from the construction of SA_4, all the other algorithmic steps account for 2.5 stages of the Doubl-algorithm. Consequently, the overall work should be always much smaller than

Running times on the random texts				
N	L-pieces	Doubl+Disc	Doubl+Disc+aheap	BGS
Alphabet Size 4				
25000000	4133	14130	19570	21485
50000000	8334	34599	46296	72552
Alphabet Size 16				
25000000	3838	11011	14143	16162
50000000	7753	26450	32367	62001
Alphabet Size 64				
25000000	3400	10080	12528	15195
50000000	6802	25606	31567	58417

Table 5.4: Construction time (in seconds) required for random texts built on alphabet-sizes 4, 16 and 64.

the one executed by all the Doubling variants. In what follows, we will validate this conjecture by running the L-pieces algorithm on larger data sets.

Results for the BGS-algorithm. It may appear surprising that BGS is the *slowest* algorithm on the random texts, after its successes on real text collections claimed in Section 5.5. It is more than twice as slow as Doubl+Disc and up to nine times slower than L-pieces. However, nothing strange is going on in these experiments on random data because if we compare Table 5.5 to Table 5.3, we notice two things: (i) the number of bulk and total I/Os executed by BGS does not depend on the alphabet size and they are almost identical to those obtained on the Reuters corpus; (ii) the execution time of BGS decreases as the alphabet size grows, and it is smaller than the time required on the Reuters corpus (when $N \approx 25$ Mbytes, see also Table 5.4). The former observation implies that our conjecture on the " quadratic I/O-behavior in practice" is true, and in fact if we double the text size, the total number of I/Os increases by a factor of approximately four. The latter observation allows us to conclude that the CPU-time of BGS is affected by the length of the repeated substrings occurring in the data set. As the alphabet size increases, this length decreases on the average, and in turn the running time of BGS decreases. Such a dependence also shows that *both* I/O and CPU-time are significant bottlenecks for BGS, as already pointed out in Section 5.5.

Results for the Doubl+Disc algorithm. The algorithm performs four stages on alphabet size 4, and three stages on alphabet sizes 16 and 64. Therefore, the random test with alphabet size 4 is the *worst case* for Doubl+Disc. In fact, the gain of the first two stages is negligible ($0.8 \cdot 10^{-8}\%$ discarded tuples); in the third stage, 98% of the tuples are discarded; the rest of the tuples is thrown away in the last fourth stage. This gives us the following insight: There are many *different* text substrings of 32 characters (*i.e.* 98%) whose 16-length prefix occurs at least twice in the collection. This validates our conjecture that the Doubl+Disc performance heavily depends on the average length of the repeated text-substrings.

If we look at the results on alphabet size 64, we see that the gain of the first stage is much bigger (about 5%), whereas the second stage throws away about 94% of

I/Os (bulk/total) on the random texts				
N	L-pieces	Doubl+Disc	Doubl+Disc+aheap	BGS
Alphabet Size 4				
25000000	9466/970256	20661/2068791	721425/4197393	22347/1449884
50000000	18912/1942979	41418/4143990	1468183/8555351	85481/5543929
Alphabet Size 16				
25000000	8499/860120	15320/1518126	537060/2937508	22347/1449884
50000000	16984/1722707	30738/3042700	1091591/5986951	85481/5543929
Alphabet Size 64				
25000000	7531/749984	14643/1434118	499036/2674076	22347/1449884
50000000	15056/1502435	30375/2996868	1070563/5844131	85481/5543929

Table 5.5: Bulk and total I/Os required for random texts built on alphabets of size 4, 16 and 64.

the tuples. Consequently, for increasing alphabet sizes (approaching natural texts), Doubl+Disc gets faster and faster. Apart from the Reuters corpus, which seems indeed to be a pathological case, we therefore expect a much better performance on natural (and more structured) texts, so that we suggest its use in practice in place of the (plain) Doubling algorithm.

At this point it is worth to notice that the former two stages of Doubl+Disc do not discard any significant number of tuples (like on Reuters). Looking carefully to their algorithmic structure, we observe that these stages execute more work than the one required by the corresponding stages of Doubling because of Step 4 (Section 5.3.1). We experimentally checked this fact by running the Doubling algorithm on the random data sets and verifying an improvement of a factor of two ! Clearly, in the early two stages, the algorithm compares short substrings of length 1 to 16 characters, and therefore it is unlikely that those substrings occur only once in a long text (and can then be discarded). Consequently, an insight coming from these experiments is that a *tuned* Doubl+Disc should follow an *hybrid* approach to gain the highest advantage from both the doubling and the discarding strategies: (plain) Doubling executed in the early (*e.g.* two) stages, Doubl+Disc for the next stages.

Results for Doubl+Disc+aheap. The algorithm performs four stages on alphabet size 4 and three stages on alphabet size 16 and 64. As for Doubl+Disc, the gain of the first two stages is negligible and nearly all tuples are thrown away in stage 3. We can naturally conclude that all the textual results that we obtained in the analysis of Doubl+Disc on random texts also hold for the Doubl+Disc+aheap algorithm as both algorithms only use different sorting variants. As a consequence, if a unique suffix array is required and time as well as space is important, this variant is the definite choice.

Results for the L-pieces algorithm. We again fix $L = 4$ and use the Doubling algorithm to construct SA_4. By using bigger texts (up to 50 Mbytes), we are ensured that all the steps of this algorithm operate on disk. The running time distributes as follows: 6% is used to sort short strings, 37% is used to build SA_4, and 47% is used to build the other three suffix arrays (via multiway mergesort). [15] Comparing Table 5.2 to Table 5.4, we conclude that the ordering of the short strings remains fast even when it is executed on disk, and it will never be a bottleneck. Moreover, the time required to build SA_4 clearly depends on the length of the longest repeated substring (see comments on Doubling), but $\mathcal{A}_{sa}$ is executed on a *compressed* text (of size $N/4$) where that length is reduced by a factor 4. Consequently, the L-pieces algorithm is usually between 2.9 and 8.3 times faster than Doubl+Disc (see Table 5.4). This speed-up is larger than the one we observed on the Reuters corpus (see Section 5.5), and thus validates our conjecture that a bigger text collection does not slow down the L-pieces algorithm.

5.5.2 Concluding remarks on our experiments

We first compare all experimented algorithms with respect to their time performance for increasing text lengths (see Table 5.2 for a summary). It is obvious from the previous discussions that the MM-algorithm is the fastest one when the suffix array can be built entirely in internal memory. As soon as the working space exceeds the memory size, we can choose among different algorithms depending on the resource to be minimized, either *time* or *space*. The L-pieces algorithm is the obvious choice whenever splitting of the suffix array does not prevent its use in the underlying application. It is more than 3 times faster on the Reuters corpus than any other experimented algorithm; it is more than twice as fast as the best Doubling variant on random texts. If, instead, a unique suffix array is needed, the choice depends on the structural properties of the text to be indexed. In presence of long repeated substrings, BGS is a good choice till *reasonably large* collections. For *very large* text collections or short repeated substrings, the hybrid variant of the Doubl+Disc or Doubl+Disc+aheap algorithm is definitely worth to be used.

If the space resource is of primary concern, then BGS is a very good choice till reasonably large text collections. For huge sizes, Doubl+Disc+aheap is expected to be better in the light of its asymptotic I/O- and CPU-time complexity. However if one is allowed to keep the suffix array distributed into pieces, then the best construction algorithm results definitely the L-pieces algorithm: It is both the fastest and the cheapest in term of space occupancy (it only needs $6N$ bytes).

We wish to conclude this long discussion on our experimental data and tested algorithms by making a further, and we think necessary, consideration. The *running time* evaluations indicated in the previous tables and pictures are not clearly intended to be definitive. Algorithmic engineering and software tuning of the C++-code might definitively lead to improvements without anyway changing the algorithmic features of the experimented algorithms, and therefore without significantly affecting the scenario that we have depicted in these pages. Consequently, we do not feel confident to give an *absolute quantitative* measure of the time performance of these algorithms in order to claim which is the “winner”. There are too many system parameters (M, buffer size, cache size, memory bandwidth), disk parameters (B, seek, latency, bandwidth, cache), and structural properties of the indexed text that heavily influence those times. Nevertheless, the *qualitative* analysis developed in these sections should, in our

---

[15] An attentive reader might observe the the distribution of the time only sums up to 90% of the total construction time. The missing 10% is required to copy back the computed suffix array.

opinion, safely route and clarify to the software developers which algorithm fits best to their wishes and needs.

We conclude this chapter by addressing two other issues. The former concerns with the problem of building word-indexes on large text collections; we show in the next section that our results can be successfully applied to this case too without any loss in efficiency and without compromising the ease of programming so to achieve a uniform, simple, and efficient approach for both indexing models. The latter issue is related to the intriguing, and apparently counterintuitive, “contradiction” between the effective practical performance of the BGS-algorithm and its unappealing worst-case behavior. In Section 5.7, we deeply study its algorithmic structure and propose a new approach that follows its *basic philosophy* but in a significantly different manner, thus resulting in a novel algorithm which combines good practical qualities with efficient worst-case performances.

5.6 Constructing word-indexes

By using a simple and efficient preprocessing phase, we are also able to build a suffix array on a text where only the beginning of each word is indexed (hereafter called *word-based* suffix array). Our idea is based on a proposal of Andersson *et al.* [ALS96] which was formulated in the context of suffix trees. We greatly simplify their presentation by exploiting the properties of suffix arrays. The preprocessing phase consists of the following steps:

1. **Scan** the text T and define as *index points* the text positions where a non-alphabetic character is followed by an alphabetic one.
2. Form a sequence S of strings which correspond to substrings of T occurring between consecutive *index points*.
3. **Sort** the strings in S via multiway mergesort.
4. Associate with each string its *rank* in the lexicographically ordered sequence, so that given $s_1, s_2 \in S$: if $s_1 = s_2$ then $rank(s_1) = rank(s_2)$, and if $s_1 <_L s_2$ then $rank(s_1) < rank(s_2)$.
5. **Sort** (backwards) S according to the starting positions in the original text T of its strings.
6. Create a compressed text T' via a simultaneous **scan** of T and (the sorted) S. Here, every substring of T occurring in S is replaced with its *rank*.

The symbols of T' are now *integers* in the range $[1, N]$. It is easy to show that the *word-based* suffix array for T is exactly the same as the suffix array of T'. Indeed, let us consider two suffixes $T[i, N]$ and $T[j, N]$ starting at the beginning of a word. They also occur in T' in a *compressed form* which preserves their lexicographic order because of the naming process. Consequently, the lexicographic comparison between $T[i, N]$ and $T[j, N]$ is the same as the one among the corresponding compressed suffixes of T'.

The cost of the preprocessing phase is dominated by the cost of sorting the string set S (step 3). As the average length of an English word is *six* characters [CM96], we immediately obtain from [AFGV97] (model A, strings shorter than B) that the I/O complexity of step 2 is $\Theta((N/B)\ \log_{M/B}(N/B))$, where N is the total number of string characters. Multiway mergesort is therefore an optimal algorithm to solve the string sorting problem in step 3. In general, we can guarantee that each string of S is always shorter than B characters by introducing some *dummy index points* that split the long substrings of T into *equal-length* shorter pieces. This approach does not suffer from the existence of very long repeated substrings that was reported by the authors of [NKRNZ97] in their quicksort/mergesort–based approaches; and therefore it is expected to work better on very large texts. With respect to [ALS96], our approach does not use *tries* [Meh84b] to manage the strings of set S and thus it reduces the overall working space and does not incur in the very well-known problems related to the management of unbalanced trees in external memory. Finally, since the preprocessing phase is based on sorting and scanning routines, we can again infer that this approach scales well on multi-disk and multi-processor machines, as we have largely discussed in the previous sections.

From the experiments executed on the L-pieces algorithm, we know that step 3 and step 5 above will be fast in practice. Furthermore, the compression in step 6 reduces the length of the repeated substrings in T, so that Doubl+Disc is expected to require very few stages when applied on T'. Consequently, we can expect that constructing word-indexes via suffix arrays is effective in real situations, and can benefit a lot from the study carried out in this chapter.

5.7 The new BGS-algorithm

The experimental results of the previous sections have led us to conclude that the BGS-algorithm is attractive for software developers because it requires only $4N$ bytes of working space, it accesses the disk in a sequential manner, and is very simple to be programmed. However, its worst-case performance is poor and thus its real behavior is not well predictable but heavily depends on the structure of the indexed text. This limits the broad applicability of BGS, making it questionable at theoretical eyes.

In this section, we propose a new algorithm which deploys the *basic philosophy* underlying BGS (*i.e.* very long disk scans) but in a completely different manner: *the text T is examined from the right to the left*. The algorithm will choreograph this new approach with some additional data structures that allow to perform the suffix comparisons using only the information *available* in internal memory, thus avoiding the *random I/Os* in the worst case. The resulting algorithm still uses small working space (*i.e.* $8N$ bytes on disk), it is very simple to be programmed, it hides small constants in the big-Oh notation, and additionally it achieves effective *worst-case performance* (namely $O(N^2/M^2)$ worst-case bulk I/Os). This *guarantees* a good practical behavior of the final algorithm for *any* indexed text *independent of its structure*, thus overcoming the (theoretical) limitations of the BGS-algorithm, and still keeping its attractive practical properties.

Let us set $\mu = \ell M$, where $\ell < 1$ is a positive constant to be fixed later in order to guarantee that some auxiliary data structures can be fit into the internal memory.

In order to simplify our discussion, let us also assume that N is a multiple of μ, say $N = k\mu$ for a positive integer k.

We divide the text T into k non-overlapping substrings of length μ each, namely $T = T_k T_{k-1} \cdots T_2 T_1$ (they are numbered going from the right to the left, thus reflecting the "stage" when the algorithm will process each of them). For the sake of presentation, we also introduce an operator $\diamond$ that allows to go from *absolute positions* in T to *relative positions* in a text piece T_h. Namely, if $T[x]$ lies in the text piece T_h then $T[x] = T_h[x \diamond \mu]$, where $x \diamond \mu = ((x-1) \texttt{ mod } \mu) + 1, 1 \leq x \leq N$.

The algorithm executes $\Theta(k) = \Theta(N/M)$ stages (like BGS) and processes the text *from the right to the left* (unlike BGS). The following invariant is inductively kept before stage h starts:

$S = T_{h-1} T_{h-2} \cdots T_1$ is the text part processed in the previous $(h-1)$ stages. The algorithm has computed and stored the following two data structures on disk: The suffix array SA_{ext} of the string S and its "inverse" array Pos_{ext}, which keeps at each entry $Pos_{ext}[j]$ the position of the suffix $S[j, |S|]$ in SA_{ext}.

After all k stages are executed, we have $S = T$ and thus $SA = SA_{ext}$. See Figure 5.4 for an illustrative example.

The main idea underlying the *leftward*-scanning of the text is that when the h-th stage processes the text suffixes starting in T_h, it has already accumulated into SA_{ext} and Pos_{ext} some information about the text suffixes starting to the right of T_h. This way, the comparison among the text suffixes starting in T_h will eventually exploit these two arrays, and thus use only *localized* information to eliminate *random I/Os*. The next Lemma formalizes this intuition:

Lemma 5.7.1. *A text suffix $T[i, N]$ starting in substring T_h can be represented succinctly via the pair $(T[i, i+\mu-1], Pos_{ext}[(i+\mu) \diamond \mu])$. Consequently, all the text suffixes starting in T_h can be represented using overall $O(\mu)$ space.*

Proof. Text suffix $T[i, N]$ can be seen as the concatenation of two strings $T[i, i+\mu-1]$ and $T[i+\mu, N]$, where the second string is an arbitrarily long text suffix. The position $i+\mu$ occurs in $T_{h-1} \cdots T_1$ $(= S)$ and in particular $T[i+\mu, N] = S[(i+\mu) \diamond \mu, |S|]$. This string can be succinctly represented with the number $Pos_{ext}[(i+\mu) \diamond \mu]$ which denotes its (lexicographic) position among S's suffixes. The space-bound easily follows by storing in internal memory the text substring $T_h\ T_{h-1}$ and the array $Pos_{ext}[2, \mu+1]$. $\square$

Stage h preserves the invariant above and thus updates SA_{ext} and Pos_{ext} by *properly* inserting the information regarding the text suffixes which start in T_h into SA_{ext}. This way, the new SA_{ext} and Pos_{ext} will correctly refer to the *extended* string $T_h\ S$, thus preserving the invariant for the $(h+1)$th stage (where $S := T_h S = T_h T_{h-1} \cdots T_1$). Details on the stage h follow (see Figure 5.4 below).

1. Load T_h and T_{h-1} in the internal memory and set $T^* = T_h\ T_{h-1}$. ($O(1)$ bulk I/Os.)
2. Load the first μ entries of Pos_{ext} into the array Pos_{int}. ($O(1)$ bulk I/Os.)

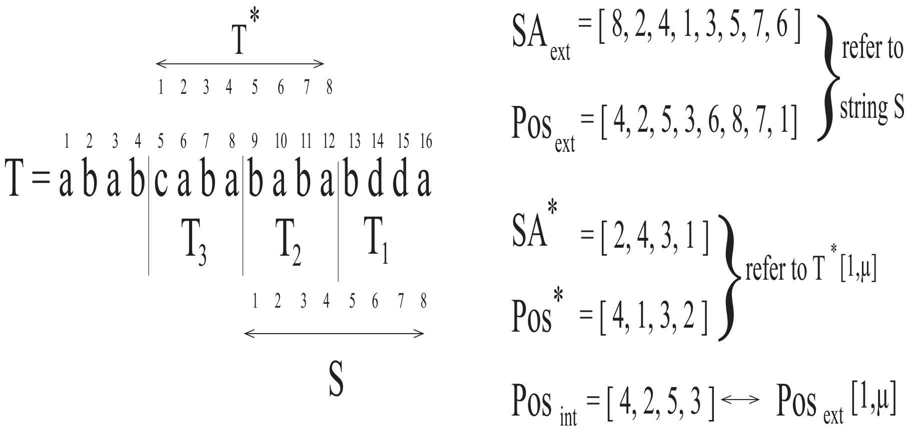


Figure 5.4: *The figure depicts the data structures used during stage* $h = 3$*, where* $T^* = T_3\, T_2$ *and* $S = T_2\, T_1$*.* SA^* *contains only the first* μ *suffixes of* T^**. Notice the order in* SA^* *of the two text suffixes starting at positions* 2 *and* 4 *of* T^* *(*i.e. 6 *and* 8 *of* T*). Restricted to their prefixes lying into* T^**, these two suffixes satisfy the relation* $4 \leq_L 2$*, but considering them in their entirety (till the end of* T*), it is* $2 \leq_L 4$*. We can represent compactly* $T[6, 16]$ *via the pair* $\langle$ *'abab'*$, 2\rangle$ *and* $T[8, 16]$ *via the pair* $\langle$ *'abab'*$, 3\rangle$*; hence the comparison of those pairs gives the correct order and can be executed in internal memory (Lemma 5.7.1).*

3. Construct the suffix array SA^* which contains the lexicographically ordered sequence of the text suffixes starting in T_h (recall that they extend till the end of T). This is done by means of three substeps which exploit the information kept in internal memory:

   (a) Build the suffix array of the string T^* using any internal memory algorithm (*e.g.* [MM93]). Then, throw away all suffixes of T^* which start in the second half of T^*. (No I/Os are required.)

   (b) Store the remaining μ entries into SA^*. [16] (No I/Os are required.)

   (c) Refine the order in SA^* taking into account the suffixes in their entirety (*i.e.* considering also their characters outside T^*). Let $T^*[x, 2\mu]$ and $T^*[y, 2\mu]$ be two suffixes which are adjacent in the current array SA^*, namely $SA^*[j] = x$ and $SA^*[j + 1] = y$ for some value j, and such that one of them is the *prefix* of the other. Their order in SA^* may possibly be not correct (see Figure 5.4). Hence, the correct order is computed by comparing the two pairs $\langle T^*[x, x + \mu - 1], Pos_{int}[(x + \mu) \diamond \mu]\rangle$ and $T^*[y, y + \mu - 1], Pos_{int}[(y + \mu) \diamond \mu]$ (see Lemma 5.7.1). [17] This comparison is done for all the suffixes of T^* for which the ambiguity above arises. (No I/Os are required.)

---

[16] Notice that this is not yet the correct SA^* because there might exist two text suffixes which start in T^* but have a common prefix which extends outside T^*. The next Step 3c takes care of this case without accessing the disk !

[17] This step is executed by using the unused space of array C.

4. Scan simultaneously the string S and the array Pos_{ext} ($O(h)$ bulk I/Os in total). For each suffix $S[j, |S|]$ do the following substeps:
    (a) Via a binary search, find the lexicographic position $pos(j)$ of that suffix in SA^* so that $S[j, |S|]$ follows the suffix of T starting at position $SA^*[pos(j) - 1]$ of T_h and precedes the suffix of T starting at position $SA^*[pos(j)]$ of T_h. Lemma 5.7.1 allows to perform the suffix comparisons of the binary search using only informations kept in internal memory.
    (b) Increment $C[pos(j)]$ by one unit.
    (c) Update the entry $Pos_{ext}[j] = Pos_{ext}[j] + pos(j) - 1$.
5. Build the new array $SA_{ext}[1, h\mu]$ by merging the old (external and shorter) array SA_{ext} with the (internal) array SA^* by means of the information available into $C[1, \mu + 1]$. This requires a single disk scan (like BGS) during which the algorithm also executes the computation $SA_{ext}[j] = SA_{ext}[j] + \mu$, in order to take the fact into account that in the next $(h + 1)$th stage the new string S has been appended to the front of the text piece T_h. (Globally $O(h)$ bulk I/Os.)
6. Process array C in internal memory as follows (no I/Os are executed):
    (a) Compute the Prefix-Sum of C.
    (b) Set $C[j] = C[j] + j$, for $j = 1, \dots, \mu + 1$.
    (c) Compute $Pos^*[1, \mu]$ as the inverse of SA^*, and then permute $C[1, \mu]$ according to $Pos^*[1, \mu]$. Namely, $C[i] = C[Pos^*[i]]$, simultaneously for all $i = 1, 2, \dots, \mu$.
7. Build the new array $Pos_{ext}[1, h\mu]$ by appending $C[1, \mu]$ to the front of the current Pos_{ext}. ($O(h)$ bulk I/Os.)

The correctness of the algorithm immediately derives from Lemma 5.7.1 and from the following observations. As far as Step 6 is concerned, we observe that:

- Step 6a determines for each entry $C[j]$ the number of text suffixes which start in S and are *lexicographically smaller* than the text suffix starting at position $SA^*[j]$ of T_h.
- Step 6b also regards the number of suffixes which start in T_h and are *lexicographically smaller* than the text suffix starting at position $SA^*[j]$ of T_h. These are $(j - 1)$ suffixes, so that the algorithm sums j to compute the final rank of that suffix (*i.e.* the one starting at $T_h[SA^*[j]]$) among all the suffixes of the string $T_h\ S$.
- Step 6c permutes the array C so that $C[j]$ gives the rank of the text suffix starting at $T_h[j]$ among all suffixes of the string $T_h\ S = T_h \cdots T_1$.

Since the string $T_h \cdots T_1$ corresponds to the string S of the next $(h + 1)$th stage, we can conclude that Step 6 correctly stores in C the first μ entries of the new array Pos_{ext} (Step 7). Finally, we point out that Step 4c correctly updates the entries of

$Pos_{ext}[1,(h-1)\mu]$ by considering the insertion of the μ text suffixes starting in T_h into SA_{ext}. Step 5 correctly updates the entries of $SA_{ext}[1,(h-1)\mu]$ as it regards to insert the μ suffixes, starting in T_h, to the front of S. In summary we can state the following result:

Theorem 5.7.2. *The suffix array of a text $T[1,N]$ can be constructed in $O(N^2/M^2)$ bulk-I/Os, no random-I/Os, and $8N$ disk space in the worst case. The overall CPU time is $O(\frac{N^2}{M}\log_2 M)$.*

The above parameter ℓ is set to fit SA^*, Pos^*, Pos_{int}, T^* and C into internal memory (notice that some space can be reused). We remark that the algorithm requires $4N$ bytes more space than the space-optimized variant of the BGS-algorithm (see Section 5.2.2). Nonetheless, we can still get the $4N$ space-bound if we accept to compute only the array $Pos_{ext} = SA^{-1}$ (implicit SA). In any case, the overall working space is much less than the one required by all Doubling variants, and it is exactly equal to the one required by our implementation of the BGS-algorithm. The new BGS-algorithm is also very simple to be programmed and has small constants hidden in the big-Oh notation. Therefore, it preserves the nice algorithmic properties of BGS, but it now guarantees good worst-case performances.

5.8 Summary

It is often observed that practitioners use algorithms which tend to be different from what is claimed as optimal by theoreticians. This is doubtless because theoretical models tend to be simplifications of reality, and theoretical analysis needs to use conservative assumptions. This chapter provided to some extent an example of this phenomenon—apparently *bad* theoretical algorithms are good in practice (see BGS and new-BGS). We actually tried to *bridge* this difference by analyzing more deeply some suffix array construction algorithms via the I/O accounting scheme of Farach *et al.*, *i.e.* by taking the specialties of current disk systems more into account. This has lead us to a reasonable and significant taxonomy of all these algorithms. As it appears clear from the experiments, the final choice of the best algorithm depends on the available disk space, on the disk characteristics (which induce different trade-offs between random and bulk I/Os), on the structural features of the indexed text, and also on the patience of the user to wait for the completion of the suffix array construction. The moral we draw from our experiments is that the design of an external-memory algorithm must take the current technological trends more and more into account, which boost interest toward the development of algorithms which prefer bulk rather than random I/Os because this paradigm can take advantage of the large disk-bandwidth and the high computational power of current computer systems.

Algorithms from this chapter were used to construct larger suffix arrays than the one constructed in the experimental part of this chapter. A novel database searching tool for DNA and protein databases, called QUASAR [BCF$^+$99] uses suffix arrays for the proximity search. We constructed the suffix array for the HUMAN Unigene database whose size is approx. 610 Mbytes[18]. This took around 22 hours.

---

[18] The construction space is more than 5.4 Gbytes so that it is not possible to compute that index on a 32-bit machine in main memory.

A number of questions remains open:
The algorithm presented in Section 5.3.3 is very efficient both in terms of I/Os and working space, but it produces $L = 4$ distinct suffix arrays. It would be interesting to establish how much it costs in practice to *merge* these four arrays into one unique suffix array. In this respect, it would be worth to use the *lazy-trie* data structure proposed in [AFGV97].

Recently, Farach *et al.* [FFMar] devised the first I/O-optimal algorithm for building *suffix trees*. Although asymptotically optimal (both in time and space), this algorithm uses more space than the algorithms discussed in this paper because it operates on a tree topology. Additionally, it uses a lot of subroutines from the area of PRAM-simulation [CGG$^+$95] from which it is known that they are not efficient in a practical setting. It is not yet clear how this approach could be used to *directly* construct suffix arrays. This is an important problem both theoretically and experimentally.

Ferragina and Manzini [FM00] showed that in internal memory, suffix arrays can be compressed and that it is possible to search in the compressed index without uncompressing all of it. Although we believe that the tested algorithms can be used to construct the their “opportunistic data structure”, it is not yet clear if it would be better to construct a unique data structure on a large file, or to build a pool of data structures on parts of the file.

Chapter 6

Conclusions

In this thesis, we studied external memory algorithms and data structures in theory and practice. While the community of researchers in external memory has grown in the last years and more and more theoretical results are published, implementation and experimental work is still in its infancy. Although algorithmic classification is possible using the I/O-model of Vitter and Shriver [VS94b], performance prediction is often not easy as this model ignores the structure of I/Os (random/bulk) and sometimes the internal work. Our main goal was therefore to provide a library of algorithms and data structures especially designed for external memory usage, so that every algorithm or data structure can be implemented and experimentally tested.

We introduced our library LEDA-SM in Chapter 3 where we described the main software layout and the most important implementation details. LEDA-SM was designed to model the one-disk case of Vitter and Shriver's I/O model directly. We especially set great store on modeling disks, disk blocks and disk block locations. As we manage disk block allocation by our own and do not rely on file system functionality for disk block management, we are able to switch to raw disk device access without the need to change or extend our library design. We showed for Solaris-based machines that this is possible for the current LEDA-SM library. Additionally, our abstract kernel classes free the application programmer of issuing complicated I/O calls and thus simplify the development process of new applications. We also showed by some low level experiments that our additional library layers do not significantly decrease the I/O throughput.

The modular design of LEDA-SM's kernel will in the future allow to easily extend the library to the parallel disk case of Vitter and Shriver's I/O model. Peter Sanders together with some students did some experiments where they implemented Barve *et.al.*'s randomized mergesort [BGV97] for the parallel independent disk case. They used threads to provide the parallelism; one thread per supported disk was used to manage the I/Os. This basic approach can be easily integrated into LEDA-SM's concrete kernel so that in the future it will also be possible to support parallel I/Os to independent disks.

In Chapters 4 and 5, we introduced several new external memory algorithms and data structures. The fourth chapter covered priority queue data structures. We introduced external memory radix heaps which are an extension of Ahuja *et.al.*'s redistributive heaps [AMOT90]. This heap is very fast in practice, uses only N/B disk pages but needs integer keys and monotonicity. Our second proposal, called array heap, is a general priority queue that also uses N/B disk pages and that supports `insert` and `delete_min` in optimal I/O bounds, even in the D-disk model. Both proposals were experimentally tested against a variety of existing in-core and external memory priority queues. Radix heaps are the fastest external memory priority queue and array heaps are the second fastest.

In the fifth chapter, we analyzed seven different construction algorithms for suffix arrays, were four of them are new approaches. Most of these algorithms perform the same amount of I/Os according to big Oh-notation. Even constant analysis for the I/Os does not allow to rank the algorithms as we experimented during our practical tests. We realized that the structure of I/Os (random or bulk) is really important so that it is necessary to either analyze this part or get an insight by performing experiments. Our large number of experiments in the end allowed to rank the algorithms. We finished this chapter by showing that our construction algorithms can be used to construct a word-based suffix array; and finally we showed that the run time and the I/O bound of BaezaYates-Gonnet-Snider's construction algorithm can be improved from cubic to quadratic.

I want to conclude this thesis by exposing some wishes: TPIE and LEDA-SM are clearly the two larger external memory libraries that currently exist. A lot of work was wasted in the last years as each group independently worked on its project and we did not manage to merge or join both libraries. Although both libraries use different concepts, I believe that it is possible to combine both approaches. A lot of implementation work could be saved and the resulting library could profit from the strengths of LEDA-SM and TPIE. My hope is that this will happen in the future.

Appendix A

The Elite-9 Fast SCSI-2 Wide Hard Disk

ST-410800W Elite 9	
Unformatted capacity	10800
Formatted capacity(512 byte blk)	9090
Average sectors per track	133
Actuator type	rotary voice coil
Tracks	132975
Cylinders	4925
Heads	27
Disks(5.25 in)	14
Media type	thin film
Head type	thin film
Recording method	ZBR RLL (1,7)
Internal transfer rate(mbits/sec)	44-65
Internal transfer rate avg(mbyte/sec)	7.2
External transfer rate(mbyte/sec)	20 (burst)
Spindle speed	5400
Average latency(msec)	5.55
Command overhead(msec)	<0.5
Buffer	1024 kbytes
Bytes per track	63000 - 91000
Sectors per drive	17845731
Bytes per cylinder	1058400 to 1587600
TPI(tracks per inch)	3921
Average access(msec) read/write	11/12
Single rack seek(msec) read/write	0.9/1.7
Max full seek(msec) read/write	23/24

Table A.1: *Technical Data of the Seagate Elite-9 Fast SCSI-2 Wide Disk. mbytes are defined as* 10^6 *bytes.*

Appendix B

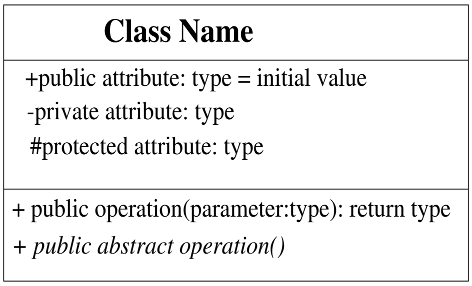
A short introduction to UML

The *Unified Modeling Language* (UML) is the successor to the wave of object-oriented analysis and design methods that appeared in the late ’80s and early ’90s. The modeling language uses a graphical notation to express designs. We will shortly describe the main concepts of this notation. A short introduction to UML can be found in the book of Martin Fowler [FS00]. We use UML in three ways: we either use the *conceptual* view, where we want to present the concepts of a design. These concepts will often relate to classes that implement them but there is often no direct mapping. The second view that we use is the *specification* view where we look at the interfaces of the software. The third view is the *implementation* view where we really have classes and we are laying the implementation bare.

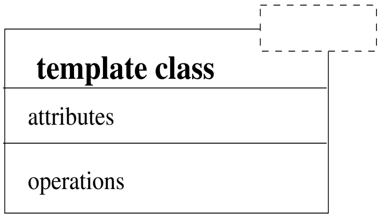
Classes and Objects The central concept of UML is the *class*. Classes have *attributes* and *operations*. Classes are pictured by a rectangle with the class name on top.The lower part is divided into two sections, one containing the attributes, the other containing the operations. Attributes come in three categories, public (prefix by “+”), private (prefixed by “-”) and protected (prefixed by “#”). Operations are processes that a class knows to carry out. An operation has the following syntax:
visibility name (parameter list):return-type-expression
where

- *visibility* is +(public), #(private) or -(protected)
- *name* is a string
- *parameter-list* contains comma-separated parameters whose syntax is similar to that for attributes
- *return-type-expression* is a comma-separated list of return types.

Abstract operations are typeset in italics. Objects (i.e. instances of classes) are pictured by a rectangle containing the underlined object name, followed by a colon and the class name.

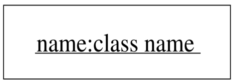


Class



template class

attributes and operations supressed



object

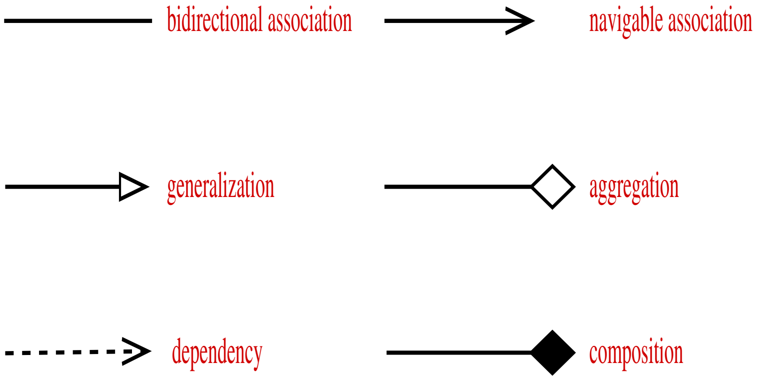
Relationships Relationships between objects and classes are represented by different kinds of lines and arrows.

1. *Association* Associations represent relationships between instances of classes. Each association has two association ends, each end is attached to one of the classes. An association can be labeled with its role name. Each association end has a multiplicity indicating how many objects may participate in the given relationship. Multiplicity may be given in ranges ($[a..b]$, $*$ represents infinity). Associations can represent conceptual relationships (conceptual view), responsibilities (specification perspective), or data structure relationships/pointers (implementation view). In the implementation and specification perspective, we often add arrows to the association end. These arrows indicate navigability. Associations are represented as solid lines.

2. *Generalization* Generalization at implementation level is associated with inheritance in programming languages. Within the specification model, generalization means that the interface of the subtype must include all elements of the interface of the supertype. Generalizations are represented as solid lines from the specialized class to the base class with a hollow triangle at the end of the base class.

3. *Aggregation/Composition* Aggregation models part-of-relationship. It's like saying that a computer has a CPU. A stronger variant of aggregations are compositions. In compositions, the part object may belong to only one whole, furthermore one often assumes that the parts live and die with the whole. Solid lines with hollow(filled) diamonds represent aggregations(compositions).

4. *Dependency* A dependency exists between two elements if changes to the definition of an element may cause changes to the other. Dependencies are represented by a dashed line with an arrow.


bidirectional association
navigable association
generalization
aggregation
dependency
composition

Appendix C

Manual Pages of the Kernel

C.1 Block Identifier (B_ID)

1. Definition

An instance BID of type B_ID is a tuple (d, num) of integers where d specifies the disk and num specifies the physical block of disk d. Block identifiers are abstract objects that are used to access physical locations in external memory. It is allowed to perform basic arithmetic operations like addition, substraction, etc. on objects of type B_ID. Arithmetic operations only effect entry num of type B_ID. A block identifier is called $valid$ if $0 \leq d < NUM\_OF\_DISKS$ and $0 \leq num < max\_blocks[d]$. If num is equal to -1, we call the block identifier $inactive$. An inactive block identifier is not connected to any physical disk location.

2. Creation

$B\_ID \;\; BID$;	creates an instance of type B_ID and initializes it to the inactive instance.
$B\_ID \;\; BID(const\ int\&\ i,\ const\ int\&\ j)$;	creates an instance of type B_ID and initializes the block number to i and the disk to j.

3. Operations

int	BID.get_disk( )	returns the disk number of instance BID.
int	BID.number( )	returns the block number of instance BID.
$void$	BID.set_number($const\ int\&\ j$)	sets the block number ofBID to `j`.

void	*BID*.set_disk(*const int& d*)	sets the disk number of *BID* to `d`
void	*BID*.set_inactive( )	sets *BID* to inactive.

Comparison Operators

bool	*const B_ID& t1* != *const B_ID& t2*	Not Equal.
bool	*const B_ID& t1* $>$ *const B_ID& t2*	Greater.
bool	*const B_ID& t1* $<$ *const B_ID& t2*	Lower.
bool	*const B_ID& t1* $\geq$ *const B_ID& t2*	Greater Equal.
bool	*const B_ID& t1* $\leq$ *const B_ID& t2*	Lower Equal.
bool	*const B_ID& t1* == *const B_ID& t2*	Equal.

Assignment Operators

B_ID&	*BID* = *const B_ID& t*	Assignment.
B_ID&	*BID* −= *int k*	Subtract integer and Assign.
B_ID&	*BID* ∗= *int k*	Multiply integer and Assign.
B_ID&	*BID* /= *int k*	Divide by integer and Assign.

Increment and Decrement

B_ID&	++*BID*	Increment Prefix.
B_ID	*BID*++	Increment Suffix.
B_ID&	− − *BID*	Decrement Prefix.

Arithmetic Operators

B_ID	$t + int\ s$	Addition of integer as friend.
B_ID	$t - int\ s$	Subtraction of integer as friend.
B_ID	$t * int\ s$	Multiplication of integer as friend.
B_ID	$t\ /\ int\ s$	Division by integer as friend.

I/O Operators

ostream&	*ostream& s* $\ll$ *id*	writes *B_ID id* to the output stream *s*.
istream&	*istream& s* $\gg$ *B_ID& id*	reads *B_ID* from input stream *s* into *id*. Block Identifiers are read in the format (*nr*,*d*) where *nr* is the block number of the block identfier to read and *d* is the disk.

4. Implementation

We use the obvious implementation with two *int* values.

C.2 External Memory Manager (ext_memory_manager)

1. Definition

An instance *ext_mem_mgr* of data type *ext_memory_manager* is an abstract realization of external memory. It consists of `D` abstract disks that are modeled by data type `ext_disk`. We connect to each disk a data strucutre of type *ext_freelist* that is responsible for managing disk blocks of this specific disk.

2. Creation

There is only one instance of the external memory manager. It is defined in
`ext_memory_manager.h`

3. Operations

B_ID	*ext_mem_mgr*.new_block(*U_ID uid*, *int* $D = -1$)	allocates a block on disk D with user identifier *uid*. Block identifier *bid* is returned to identify the block.

B_ID ext_mem_mgr.new_blocks($U\_ID\ uid$, $int\ k$, $int\ D = -1$)

allocates k consecutive blocks on disk D with user identifier uid. The block identifier bid of the first block is returned. The other $k-1$ following blocks are represented by block identifiers $bid+1$, $bid+2$,…,$bid+k-1$.

$void$ ext_mem_mgr.free_block($B\_ID\ bid$, $U\_ID\ uid$)

frees the block with block identifier bid. User identifier uid is used to check if block bid was allocated by user uid before. If this is not the case, the request is ignored and an error message is produced.

$void$ ext_mem_mgr.free_blocks($B\_ID\ bid$, $U\_ID\ uid$, $int\ k$)

frees k consecutive blocks. bid is the block identifier of the first block to free. The other $k-1$ following blocks are represented by block identifiers $bid+1$, $bid+2$,…,$bid+k-1.uid$ is used to check if the disk blocks were allocated by the user before. If this is not the case, the request is ignored and an error message is produced.

$void$ ext_mem_mgr.free_all_blocks($U\_ID\ uid$)

frees all blocks of user uid on all disks.

$void$ ext_mem_mgr.write_block($B\_ID\ bid$, $U\_ID\ uid$, $ext\_block\ B$)

writes block B to the block bid of external memory.
Precondition: User uid is the owner of block bid in external memory. Otherwise, the access is invalidated and an error message is produced.

$void$ ext_mem_mgr.write_blocks($B\_ID\ bid$, $U\_ID\ uid$, $ext\_block\ B$, $int\ k$)

writes k consecutive blocks B to the block bid of external memory. The target locations on the disks are represented by the block identifiers bid,…, $bid+k-1$.
Precondition: User uid is the owner of blocks bid,…,$bid+k-1$ in external memory. Otherwise, the access is invalidated and an error message is produced.

$void$ ext_mem_mgr.read_block($B\_ID\ bid$, $U\_ID\ uid$, $ext\_block\ B$)

reads block bid of external memory into B.
Precondition: User uid is the owner of block bid in external memory. Otherwise, the access is invalidated and an error message is produced.

$void$	ext_mem_mgr.read_blocks($B\_ID\ bid$, $U\_ID\ uid$, $ext\_block\ B$, $int\ k$)	reads k consecutive disk blocks starting from disk location bid into B. *Precondition*: User uid is the owner of blocks bid,...,$bid + k - 1$ in external memory. Otherwise, the access is invalidated and an error message is produced.
$void$	ext_mem_mgr.read_ahead_block($B\_ID\ bid$, $U\_ID\ uid$, $B\_ID\ ahead\_bid$, $ext\_block\ B$)	reads disk block bid of external memory into B and starts read-ahead of disk block $ahead\_bid$. *Precondition*: User uid is the owner of disk blocks bid and $ahead\_bid$. Otherwise, the access is invalidated and an error message is produced.
$bool$	ext_mem_mgr.check_owner($B\_ID\ bid$, $U\_ID\ uid$)	checks if user uid is the owner of disk block bid.
int	ext_mem_mgr.unused_blocks($int\ D = -1$)	returns the number of unused disk blocks of disk D (of all disks if D=-1).
int	ext_mem_mgr.unused_cons_blocks($int\ D = -1$)	returns the number of unused consecutive disk blocks of disk D (of all disks if D=-1).
int	ext_mem_mgr.used_blocks($int\ D = -1$)	returns the number of used disk blocks of disk D (of all disks if D=-1).
$long$	ext_mem_mgr.unused_mem_blocks( )	returns the number of unused blocks of internal memory. This command is operating system specific and does not work on every platform.
int	ext_mem_mgr.write_counter($int\ D = -1$)	returns the number of blocks written to disk D till now (to all disks, if D=-1).
int	ext_mem_mgr.read_counter($int\ D = -1$)	returns the number of blocks read from disk D (from all disks,if D=-1).
$void$	ext_mem_mgr.print_statistics($int\ D = -1$)	prints statistics of disk D (of all disks, if D=-1).

void	ext_mem_mgr.reset_statistics($int\ D = -1$)	
		resets statistics of external memory manager (if D==-1) or disk D. This includes the resetting of the write and read counters.

C.3 Name Server (name_server)

1. Definition

The name server is used to allocate user identifications. An user identification is a value of type U_ID. There is a special user identification called NO_USER. User identification NO_USER can only be used to create logical blocks of type $block$<E> any write or read requests for this user are invalidated.

2. Creation

There is only one instance of data type name server called UID_SERVER. It is defined in `name_server.h`.

3. Operations

U_ID	$name\_svr$.new_id( )	returns a new user-identifier
void	$name\_svr$.free_id($U\_ID\ uid$)	
		frees user-identifier uid.

4. Implementation

An user identifier is implemented by data type int. Data structure $name\_server$ uses a LEDA-priority queue to keep track of freed user-identifiers. The running time of operations new_id and $free\_id$ are dependent of the priority queue implementation ($O(\log n)$ and $O(1)$ if Fibonacci heaps are used).

Bibliography

[Abe99] J. Abello. personal communication, 1999.

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 540–548, El Paso, May 1997. ACM Press.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[ALS96] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. In *Proceedings of the 7th annual symposium on Combinatorial pattern matching (CPM)*, volume 1075 of *Lecture notes in computer science*. Springer, 1996.

[AMOT90] R. Ahuja, K. Mehlhorn, J.B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 3(2):213–223, 1990.

[AN95] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software Practice and Experience*, 2(25):129–141, 1995.

[Arg95] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95)*, volume 955 of *LNCS*, pages 334–345. Springer, 1995.

[Arg96] L. Arge. *Efficient external-memory data structures and applications*. PhD thesis, University of Aarhus, 1996.

[Arg99] L. Arge. personal communication, February 1999.

[AV88] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, 1988.

[BCF$^+$99] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array. In *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 77–83, Lyon, France, 1999. ACM Press.

[BCMF99] K. Brengel, A. Crauser, U. Meyer, and P. Ferragina. An experimental study of priority queues in external memory. In *Proceedings of the 3rd International Workshop on Algorithmic Engineering (WAE)*, volume 1668 of *Lecture notes in computer science*, pages 345–359. Springer, 1999.

[BCMF00] K. Brengel, A. Crauser, U. Meyer, and P. Ferragina. An experimental study of priority queues in external memory. *Journal of Experimental Algorithms*, page to appear, 2000.

[BGV97] R. D. Barve, E.F. Grove, and J.S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, June 1997.

[BGVW00] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[BK98] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the 6th Skandinavian workshop on Algorithm theory (SWAT-98)*, volume 1432 of *Lecture notes in computer science*. Springer, 1998.

[BKS99] J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study:heap construction. In *Proceedings of the 3rd International Workshop on Algorithmic Engineering (WAE)*, volume 1668 of *Lecture notes in computer science*, pages 301–315. Springer, 1999.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indizes. *Acta Informatica*, 1:173–189, 1972.

[Bos99] H. G. P. Bosch. *Mixed-media File Systems*. PhD thesis, Universiteit Twente, Netherlands, 1999.

[Bre00] K. Brengel. Externe Prioritätswarteschlangen. Master's thesis, Max-Planck-Institut für Informatik, Universität des Saarlandes, Germany, 2000.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report TR 124, Digital SRC Research, 1994.

[BYBZ96] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[CC98] A. Colvin and T.H. Cormen. ViC*: A compiler for virtual-memory C*. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 23–33, 1998.

[CF99] A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. In *Proceedings of the 7th Annual European Symposium on Algorithms (ESA)*, volume 1643 of *Lecture notes in computer science*, pages 224–235. Springer, 1999.

[CF01] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, page to appear, 2001.

[CFM98] A. Crauser, P. Ferragina, and U. Meyer. Efficient priority queues in external memory. MPI für Informatik, working paper, 1998.

[CGG$^+$95] Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-Siam Symposium on Discrete Algorithms (SODA)*, volume SODA 95, pages 139–149, San Francisco, 1995. ACM-SIAM.

[Chi95] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Algorithms*. PhD thesis, Brown University, 1995.

[CM96] D.R. Clark and J.I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. ACM/SIAM, 1996.

[CM99] A. Crauser and K. Mehlhorn. LEDA-SM, extending LEDA to secondary memory. In *Proceedings of the 3rd International Workshop on Algorithmic Engineering (WAE)*, volume 1668 of *Lecture notes in computer science*, pages 228–242. Springer, 1999.

[CMA$^+$98] A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, and R. Wirth. On the performance of LEDA-SM. Technical report, Max-Planck-Institut für Informatik, November 1998.

[CMS98] A. Crauser, K. Mehlhorn U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In *Proceedings of the 23rd international symposium on Mathematical foundations of computer science (MFCS)*, volume 1450 of *Lecture notes in computer science*, pages 722–73. Springer, 1998.

[Com64] W.T. Comfort. Multiword list items. *Communication of the ACM*, 7(6), October 1964.

[Coo] Quantum Cooperation. Storage technology and trends. http://www.quantum.com/src/tt/storage_tech_trends.htm.

[CP98] A. Cockcroft and R. Pettit. *Sun Performance and Tuning*. Prentice Hall, 1998.

[CSW98] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1998.

[CWN97] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 68–78, San Jose, CA, November 1997. ACM Press.

[Dei90] H. M. Deitel. *Operating Systems*. Addison Wesley, 1990.

[Dij59] E.W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.

[FAFL95] E.A. Fox, R.M. Akcyn, R.K. Furuta, and J.J. Leggett. Digital libraries. *Communications of the ACM*, 38(4):22–28, April 1995.

[Fen97] C.L. Feng. Pat-tree-based keyword extraction for chinese information retrieval. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, volume 31,special issue of *SIGIR Forum*, pages 50–58. ACM Press, 1997.

[FFM98] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 174–185. IEEE Computer Society, 1998.

[FFMar] M. Farach, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 2000 (to appear).

[FG95] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 693–702. ACM, May 1995.

[FG96] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382. ACM/SIAM, 1996.

[FJJT99] R. Fadel, K.V. Jakobsen, J. Katajainen J., and Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220, 1999.

[FKS84] M.L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *IEEE Foundations on Computer Science*, 2000.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979. Also published in/as: IBM, Research Report RJ2305, Jul. 1978.

[FS00] Martin Fowler and Kendall Scoot. *UML Distilled, A brief guide to the standard object modeling language*. Addison-Wesley, 2000.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[FW72] W. D. Frazer and C. K. Wong. Sorting by natural selection. *Communications of the ACM*, 15(10):910–913, October 1972.

[GHGS92] R. A. Baeza-Yates G. H. Gonnet and T. Snider. *Information Retrieval – Data Structures and Algorithms*. W.B. Frakes and R. BaezaYates Editors, Prentice-Hall, 1992.

[Goo99] T. Goodrich. Communication-efficient parallel sorting. *SICOMP: SIAM Journal on Computing*, 29, 1999.

[GTVV93] M.T. Goodrich, J-J Tsay, D.E. Vengroff, and J.S. Vitter. External-memory computational geometry. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[Gus97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[Hir73] D.S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, October 1973.

[HMSV97] D. Hutchinson, A. Maheshwari, J. Sack, and R. Velicescu. Early experiences in implementing buffer trees. *In Proceedings of the 2nd International Workshop on Algorithmic Engineering*, pages 92–103, 1997.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[IEE87] IEEE. Ieee standard 754-1985 for binary floating-point arithmetic. reprinted in SIGPLAN 22, 1987.

[KL93] D.E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, 1993.

[Kno65] K.C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965.

[KNRNZ97] J. P. Kitajima, G. Navarro, B. Ribeiro-Neto, and N. Ziviani. Distributed generation of suffix arrays: A quicksort based approach. In *Proceedings of the 4th South American Workshop on String Processing*, pages 53–69, Valparaiso, Chile, 1997. Carleton University Press.

[Knu81] D.E. Knuth. *The Art of Computer Programming (Volume 3): Sorting and Searching*. Addison-Wesley, third edition edition, 1981.

[Knu97] D.E. Knuth. *The Art of Computer Programming (Volume 1): Fundamental Algorithms*. Addison-Wesley, third edition edition, 1997.

[KS96] V. Kumar and E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP*, pages 169–177, 1996.

[KSC78] V.F. Kolchin, B.A. Sevastyanov, and V.P. Chestyakov. *Random allocations*. Winston & Sons, Washington, 1978.

[Kur99] S. Kurtz. Reducing the space requirement of suffix trees. *Software – Practice and Experience*, 29(3):1149–1171, 1999.

[Lar96] N.J. Larsson. Extended application of suffix trees to data compression. In *DCC: Data Compression Conference*. IEEE Computer Society TCC, 1996.

[Lei85] T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, April 1985.

[LL96] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. Technical Report TR-96-02-03, University of Washington, Department of Computer Science and Engineering, February 1996.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Meh84a] K. Mehlhorn. *Data structures and algorithms 1,2, and 3*. Springer, 1984.

[Meh84b] K. Mehlhorn. *Data Structures and Algorithms 3: Multidimensional Seaching and Computational Geometry*. Springer Verlag, 1984.

[Mey01] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the 12th Annual SIAM/ACM Symposium on Discrete Algorithms*, to appear, 2001.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[MN95] K. Mehlhorn and S. Näher. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MN99] K. Mehlhorn and S. Näher. *LEDA a platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[MSV00] Y. Matias, E. Segal, and J.S. Vitter. Efficient bundle sorting. In *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms*, 2000.

[NKRNZ97] G. Navarro, J.P. Kitajima, B.A. Ribeiro-Neto, and N. Ziviani. Distributed generation of suffix arrays. In *Combinatorial Pattern Matching*, pages 103–115, 1997.

[NMM91] D. Naor, C.U. Martel, and N.S. Matloff. Performance of priority queues in a virtual memory environment. *The Computer Journal*, 34(5):428–437, 1991.

[NV93] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June-July 1993.

[NV95] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[PN77] J.L. Peterson and T.A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[Ran69] Brian Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–372, July 1969.

[RBYN00] A. Moffat R. Baeza-Yates and G. Navarro. *Handbook of Massive Data Sets*. Kluwer, 2000.

[RMKR72] R. E. Miller R. M. Karp and A. L. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *4th ACM Symposium on Theory of Computing*, pages 125–136, 1972.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, pages 17–28, 1994.

[Sad98] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *IEEE Data Compression Conference*, 1998.

[San99] P. Sanders. Fast priority queues for cached memory. In *Workshop on Algorithmic Engineering and Experimentation (ALENEX)*, volume 1619 of *Lecture notes in computer science*, pages 312–327. Springer, 1999.

[SV87] J.T. Stasko and J.S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30:234–249, 1987.

[Tho96] M. Thorup. On RAM priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, New York/Philadelphia, January 28–30 1996. ACM/SIAM.

[UY91] Ullman and Yannakakis. The Input/Output Complexity of Transitive Closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360, 1991.

[Vit98] J.S. Vitter. External memory algorithms. *ACM Symposium on Principles of Database Systems*, 1998. Also invited paper in European Symposium on Algorithms,1998.

[VS94a] J.S. Vitter and E.A.M. Shriver. Optimal algorithms for parallel memory II:two-level memories. *Algorithmica*, 12(2-3):148–169, 1994.

[VS94b] J.S. Vitter and E.A.M. Shriver. Optimal algorithms for parallel memory I:two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[VV95] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Symposium on Parallel and Distributed Processing (SPDP '95)*, pages 74–77. IEEE Computer Society Press, October 1995.

[Wil64] J.W.J. William. Algoritm 232 (heapsort). *Communication of the ACM*, pages 347–348, 1964.

[WJNB95] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocaton:a survey and critical review. In *Proceedings of the 2nd international workshop on Rules in database systems (RIDS)*, volume 985 of *Lecture notes in computer science*, pages 1–116. Springer, 1995.

[WMB94] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, 115th Avenue, New York, 1994.

[Won97] B. L. Wong. *Configuration and Capacity Planning for Solaris Servers*. Prentice Hall, 1997.

[ZMR96] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(3):10–15, 1996.

Appendix D

Curriculum Vitae

Andreas Crauser
Richard-Wagner-Str. 64
66125 Dudweiler
Germany

Date of Birth 13.06.1971.

Citizenship German.

Education

MAX-PLANCK-INSTITUT FÜR INFORMATIK SAARBRÜCKEN, GERMANY
1996–present

PhD student.

UNIVERSITÄT DES SAARLANDES SAARBRÜCKEN, GERMANY
1991–1996

Diplom ($\approx$ Master's degree) in Computer Science in 1996. Thesis: *Implementierung des PRAM Betriebssystems PRAMOS: PRAM Seite*, advisor Prof. Dr. Wolfgang J. Paul.
Vordiplom ($\approx$ Bachelor's degree) in Computer Science in 1993.

WILLI-GRAF GYMNASIUM SAARBRÜCKEN, GERMANY
1990

German Abitur ($\approx$ Highschool Diploma).

Work and Teaching Experience

MAX-PLANCK-INSTITUT FÜR INFORMATIK SAARBRÜCKEN, GERMANY
1996–2000

Teaching experience in several lectures (Algorithms for Large Data Sets, various seminars). Responsible for hard- and software acquisition, maintenance of software, and system support for the research group of Prof. Dr. Kurt Mehlhorn.

Advisor of two master students in the area of external memory algorithms

UNIVERSITÄT DES SAARLANDES SAARBRÜCKEN, GERMANY
1991–1996

Teaching assistant for the following lectures:
Computer Science I
Computer Science III
Computer Science IV
Fortgeschrittenen-Praktikum (one-term programming project) simulation of parallel algorithms (implementation of mesh sorting algorithms in C).

Publications

Journal Publications

1. BRENGEL, K., CRAUSER, A., MEYER, U., AND FERRAGINA, P. An experimental study of priority queues in external memory. *Journal of Experimental Algorithms* (2000), to appear.
2. CRAUSER, A., AND FERRAGINA, P. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica* (2001), to appear.

Conference Publications

3. CRAUSER, A., MEHLHORN, K., MEYER, U., AND SANDERS, P. A parallelization of dijkstra's shortest path algorithm. In *Proceedings of the 23rd International Symposium on the Mathematical Foundations of Computer Science (MFCS-98)* (Brno, Czech Republic, August 1998), L. Brim, J. Gruska, and J. Zlatuska, Eds., vol. 1450 of *Lecture Notes in Computer Science*, Springer, pp. 722–731.
4. BRENGEL, K., CRAUSER, A., MEYER, U., AND FERRAGINA, P. An experimental study of priority queues in external memory. In *Proceedings of the 3rd International Workshop on Algorithmic Engineering (WAE)* (1999), vol. 1668 of *Lecture notes in computer science*, Springer, pp. 345–359.
5. BURKHARDT, S., CRAUSER, A., FERRAGINA, P., LENHOF, H.-P., RIVALS, E., AND VINGRON, M. q-gram based database searching using a suffix array. In *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB)* (Lyon, France, 1999), ACM Press, pp. 77–83.
6. CRAUSER, A., AND FERRAGINA, P. On constructing suffix arrays in external memory. In *Proceedings of the 7th Annual European Symposium on Algorithms (ESA)* (1999), vol. 1643 of *Lecture notes in computer science*, Springer, pp. 224–235.
7. CRAUSER, A., FERRAGINA, P., MEHLHORN, K., MEYER, U., AND RAMOS, E. A. I/O-optimal computation of segment intersections. In *Proceedings of the DIMACS Workshop on External Algorithms and Visualization* (New Brunswick, New Jersey, October 1999), J. M. Abello and J. S. Vitter, Eds., vol. 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS, AMS, pp. 131–138.
8. CRAUSER, A., AND MEHLHORN, K. LEDA-SM, extending LEDA to secondary memory. In *Proceedings of the 3rd International Workshop*

on Algorithmic Engineering (WAE) (1999), vol. 1668 of *Lecture notes in computer science*, Springer, pp. 228–242.

Technical Reports

9. Crauser, A., Mehlhorn, K., Althaus, E., Brengel, K., Buchheit, T., Keller, J., Krone, H., Lambert, O., Schulte, R., Thiel, S., Westphal, M., and Wirth, R. On the performance of LEDA-SM. Tech. Rep. MPI-I-98-1-028, Max-Planck-Institut für Informatik, November 1998.

Colloquia

10. Graph data types in internal and external memory. Given at the *SIAM Annual Meeting*, 1999.
11. I/O-optimal computation of segment intersections. Given at the *DIMACS Workshop External Memory Algorithms and Visualization*, 1998.
12. On contructing suffix arrays in external memory. Given at the *7th Annual European Symposium on Algorithms (ESA-99)*, 1999.

Index