# The Buffer Tree: A New Technique for Optimal I/O Algorithms*

Lars Arge†

BRICS‡
Department of Computer Science
University of Aarhus
Aarhus, Denmark

August 1996

**Abstract**

In this paper we develop a technique for transforming an internal-memory tree data structure into an external-memory structure. We show how the technique can be used to develop a search tree like structure, a priority queue, a (one-dimensional) range tree and a segment tree, and give examples of how these structures can be used to develop efficient I/O algorithms. All our algorithms are either extremely simple or straightforward generalizations of known internal-memory algorithms—given the developed external data structures. We believe that algorithms relying on the developed structure will be of practical interest due to relatively small constants in the asymptotic bounds.

## 1 Introduction

In the last few years, more and more attention has been given to Input/Output (I/O) complexity of existing algorithms and to the development of new I/O-efficient algorithms. This is due to the fact that communication between fast internal memory and slower external memory is the bottleneck in many large-scale computations. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity [21]. Currently, technological advances are increasing CPU speed at an annual rate of 40-60% while disk transfer rates are only increasing by 7-10% annually [24].

A lot of work has already been done on designing external memory versions of known internal-memory data structures (e.g. [6, 14, 16, 17, 18, 23, 25, 26, 29]), but practically all of these data structures are designed to be used in on-line settings, where queries should be answered immediately and within a good worst case number of I/Os. This effectively means

*This paper is a revised and extended version of BRICS report 94-16. An extended abstract version was presented at the Fourth Workshop on Algorithms and Data Structures (WADS'95)

†This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II) and by Aarhus University Research Foundation. Part of the work was done while a Visiting Scholar at Duke University. Email: large@brics.dk

‡Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

that using these structures to solve off-line problems yields non-optimal algorithms because they are not able to take full advantage of the large internal memory. Therefore a number of researchers have developed techniques and algorithms for solving large-scale off-line problems without using external memory data structures [1, 11, 14].

In this paper we develop external data structures that take advantage of the large main memory. This is done by only requiring good amortized performance of the operations on the structures, and by allowing search operations to be batched. The data structures developed can then be used in simple and I/O-efficient algorithms for computational geometry and graph problems. As pointed out in [11] and [14] problems from these two areas arise in many large-scale computations in e.g. object-oriented, deductive and spatial databases, VLSI design and simulation programs, geographic informations systems, constraint logic programming, statistics, virtual reality systems, and computer graphics.

### 1.1 I/O Model and Previous Results

We will be working in an I/O model introduced by Aggarwal and Vitter [1]. The model has the following parameters:

$$\begin{aligned} N &= \#\text{ of elements in the problem instance;} \\ M &= \#\text{ of elements that can fit into main memory;} \\ B &= \#\text{ of elements per block,} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. The model captures the essential parameters of many of the I/O-systems in use today, and depending on the size of the data elements, typical values for workstations and file servers are on the order of $M = 10^6$ or $10^7$ and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

An I/O operation in the model is a swap of $B$ elements from internal memory with $B$ consecutive elements from external memory. The measure of performance we consider is the number of such I/Os needed to solve a given problem. Internal computation is for free. As we shall see shortly the quotients $N/B$ (the number of blocks in the problem) and $M/B$ (the number of blocks that fit into internal memory) play an important role in the study of I/O-complexity. Therefore, we will use $n$ as shorthand for $N/B$ and $m$ for $M/B$. Furthermore, we say that an algorithm uses a linear number of I/O operations if it uses at most $O(n)$ I/Os to solve a problem of size $N$. In [31] the I/O model is extended with a parameter $D$. Here the external memory is partitioned into $D$ distinct disk drives, and if no two blocks come from the same disk, $D$ blocks can be transferred per I/O. The number $D$ of disks range up to $10^2$ in current disk arrays.

Early work on I/O algorithms concentrated on algorithms for sorting and permutation-related problems in the single disk model [1], as well as in the extended version of the I/O-model [19, 20, 30, 31]. External sorting requires $\Theta(n \log_m n)$ I/Os,[1] which is the external memory equivalent of the well-known $\Theta(N \log N)$ time bound for sorting in internal memory. Note that this means that $O(\frac{\log_m n}{B})$ is the I/O bound corresponding to the $O(\log_2 N)$ bound on the operations on many internal-memory data structures. More recently external-memory researchers have designed algorithms for a number of problems in different areas. Most notably I/O-efficient algorithms have been developed for a large number of computational

[1] We define for convenience $\log_m n = \max\{1, (\log n)/(\log m)\}$.

geometry [5, 14] and graph problems [11]. In [4] a general connection between the comparison-complexity and the I/O-complexity of a given problem is shown in the "comparison I/O model" where comparison of elements is the only allowed operation in internal memory.

### 1.2 Our Results

In this paper we develop a technique for transforming an internal-memory tree data structure into an external memory data structure. We use our technique to develop a number of external memory data structures, which in turn can be used to develop optimal algorithms for problems from the different areas previously considered with respect to I/O-complexity. All these algorithms are either extremely simple or straightforward generalizations of known internal-memory algorithms—given the developed external data structures. This is in contrast to the I/O-algorithms developed so far, as they are all very I/O-specific. Using our technique we on the other hand manage to isolate all the I/O-specific parts of the algorithms in the data structures, which is nice from a software engineering point of view. Ultimately, one would like to give the task of transforming an ordinary internal-memory algorithm into a good external memory one to the compiler. We believe that our technique and the developed structures will be useful in the development of algorithms for other problems in the mentioned areas as well as in other areas. Examples of this can be found in [2, 3, 5, 9]. More specifically, the results in this paper are the following:

**Sorting:** We develop a simple dynamic tree structure (*The Buffer Tree*) with operations *insert*, *delete* and *write*. We prove amortized I/O bounds of $O(\frac{\log_m n}{B})$ on the first two operations and $O(n)$ on the last. Using the structure we can sort $N$ elements with the standard tree-sort algorithm in the optimal number of I/Os. This algorithm is then an alternative to the sorting algorithms developed so far. The algorithm is the first I/O-algorithm that does not need all the elements to be present by the start of the algorithm.

**Graph Problems:** We extend the buffer tree with a *deletemin* operation in order to obtain an external-memory *priority queue*. We prove an $O(\frac{\log_m n}{B})$ amortized bound on the number of I/Os used by this operation. Using the structure it is straightforward to develop an extremely simple algorithm for "circuit-like" computations as defined in [11]. This algorithm is then an alternative to the "time-forward processing technique" developed in the same paper. The time-forward processing technique only works for large values of $m$, while our algorithm works for all $m$. In [11] the time-forward processing technique is used to develop an efficient I/O algorithm for external-memory list-ranking, which in turn is used to develop efficient algorithms for a large number of graph-problems.[2] All these algorithms thus inherit the constraint on $m$ and our new algorithm removes it from all of them. Finally, the structure can of course also be used to sort optimally.

**Computational Geometry Problems:** We also extend the buffer tree with a *batched rangesearch* operation in order to obtain an external (one-dimensional) *range tree* structure. We prove an $O(\frac{\log_m n}{B}+r)$ amortized bound on the number of I/Os used by the operation. Here $r$ is the number of *blocks* reported. Furthermore, we use our technique to develop an external version of the *segment tree* with operations *insert/delete* and *batched search* with the same I/O bounds as the corresponding operations on the range tree structure. The two structures enable us to solve the orthogonal line segment intersection, the batched range searching, and

[2] Expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning tree verification, connected components, minimum spanning forest, biconnected components, ear decomposition, and a number of problems on planar *st*-graphs.

the pairwise rectangle intersection problems in the optimal number of I/O operations. We can solve these problems with exactly the same plane-sweep algorithms as are used in internal memory. As mentioned, large-scale computational geometry problems arise in many areas. The three intersection reporting problems mentioned especially arise in VLSI design and are certainly large-scale in such applications. The pairwise rectangle intersection problem is of special interest, as it is used in VLSI design rule checking [8]. Optimal algorithms for the three problems are also developed in [14], but as noted earlier these algorithms are very I/O-specific, while we manage to "hide" all the I/O-specific parts in the data structures and use the known internal-memory algorithms. A note should also be made on the fact that the search operations are *batched*. Batched here means that we will not immediately get the result of a search operation. Furthermore, parts of the result will be reported at different times when other operations are performed. This suffices in the plan-sweep algorithms we are considering, as the sequence of operations done on the data structure in these algorithms does not depend on the results of the queries in the sequence. In general, problems where the whole sequence of operations on a data structure is known in advance, and where there is no requirement on the order in which the queries should be answered, are known as *batched dynamic problems* [13].

As mentioned some work has already been done on designing external versions of known internal dynamic data structures, but practically all of it has been done in the I/O model where the size of the internal memory equals the block size. The motivation for working in this model has partly been that the goal was to develop structures for an on-line setting, where answers to queries should be reported immediately and within a good worst-case number of I/Os. This means that if we used these structures to solve the problems we consider in this paper, we would not be able to take full advantage of the large main memory. Consider for example the well-known B-tree [7, 12, 18]. On such a tree one can do an *insert* or *delete* operation in $O(\log_B n)$ I/Os and a *rangesearch* operation in $O(\log_B n + r)$ I/Os. This means that using a B-tree as sweep-structure in the standard plane-sweep algorithm for the orthogonal line segment intersection problem results in an algorithm using $O(N \log_B n + r)$ I/Os. But an optimal solution for this problem only requires $O(n \log_m n + r)$ I/Os [4, 14]. For typical systems $B$ is less than $m$ so $\log_B n$ is larger than $\log_m n$, but more important, the B-tree solution will be slower than the optimal solution by a factor of $B$. As $B$ typically is on the order of thousands this factor is crucial in practice. The main problem with the B-tree in this context is precisely that it is designed to have a good worst-case on-line search performance. In order to take advantage of the large internal memory, we on the other hand use the fact that we only are interested in the *overall* I/O use of the algorithm for an *off-line* problem—that is, in a good amortized performance of the involved operations—and sometime even satisfied with *batched* search operations.

As mentioned we believe that one of the main contributions of this paper is the development of external-memory data structures that allow us to use the normal internal-memory algorithms and "hide" the I/O-specific parts in the data structures. Furthermore, we believe that our structures will be of practical interest due to relatively small constants in the asymptotic bounds. We hope in the future to be able to implement some of the structures in the transparent parallel I/O environment (TPIE) developed by Vengroff [27]. Results of experiments on the practical performance of several algorithms developed for the I/O model are reported in [9, 10, 28].

The main organization of the rest of this paper is the following: In the next section we

sketch our general technique. In section 3 we then develop the basic buffer tree structure which can be use to sort optimally, and in section 4 and 5 we extend this structure with a deletemin and batched rangesearch operation, respectively. The external version of the segment tree is developed in section 6. Using techniques from [20] all the developed structures can be modified to work in the $D$-disk model—that is, the I/O bounds can be divided by $D$. We discuss such an extension in Section 7. Finally, conclusions are given in Section 8.

## 2 A Sketch of the Technique

In this section we sketch the main ideas in our transformation technique. When we want to transform an internal-memory tree data structure into an external version of the structure, we start by grouping the (binary) nodes in the structure into super-nodes with fan-out $\Theta(m)$—that is, fan-out equal to the number of blocks that fits into internal memory. We furthermore group the leaves together into blocks obtaining an $O(\log_m n)$ "super-node height". To each of the super-nodes we then assign a "buffer" of size $\Theta(m)$ blocks. No buffers are assigned to the leaves. As the number of super-nodes on the level just above the leaves is $O(n/m)$, this means that the total number of buffers in the structure is $O(n/m)$.

Operations on the structure—updates as well as queries—are then done in a "lazy" manner. If we for example are working on a search tree structure and want to insert an element among the leaves, we do not right away search all the way down the tree to find the place among the leaves to insert the element. Instead, we wait until we have collected a block of insertions (or other operations), and then we insert this block in the buffer of the root. When a buffer "runs full" the elements in the buffer are "pushed" one level down to buffers on the next level. We call this a *buffer-emptying process*. Deletions or other and perhaps more complicated updates, as well as queries, are basically done in the same way as insertions. This means that we can have several insertions and deletions of the same element in the tree, and we therefore time stamp the elements when we insert them in the top buffer. It also means that the queries get batched in the sense that the result of a query may be generated (and reported) lazily by several buffer-emptying processes.

The main requirement needed to show the I/O bounds mentioned in the introduction is that we should be able to empty a buffer in $O(m + r')$ I/O operations. Here $r'$ is the number of blocks reported by query operations in the emptied buffer. If this is the case, we can do an amortization argument by associating a number of credits to each block of elements in the tree. More precisely, each block in the buffer of node $x$ must hold $O(\text{the height of the tree rooted at } x)$ credits. As we only do a buffer-emptying process when the buffer runs full, that is, when it contains $\Theta(m)$ blocks, and as we can charge the $r'$-term to the queries that cause the reports, the blocks in the buffer can pay for the emptying-process as they all get pushed one level down. On insertion in the root buffer we then have to give each update element $O(\frac{\log_m n}{B})$ credits and each query element $O(\frac{\log_m n}{B} + r)$ credits, and this gives us the desired bounds. Of course we also need to consider e.g. rebalancing of the transformed structure. We will return to this, as well as the details in other operations, in later sections. Another way of looking at the above amortization argument is that we touch each block a constant number of times on each level of the structure. Thus the argument still holds if we can empty a buffer in a linear number of I/Os in the number of elements in the buffer. In later sections we will use this fact several times when we show how to empty a buffer containing $x$ blocks, where $x$ is bigger than $m$, in $O(m + x) = O(x)$ I/Os. Note also that the amortization

argument works as long as the fan-out of the super-nodes is $\Theta(m^c)$ for $0 < c \leq 1$, as the super-node height remains $O(\log_m n)$ even with this smaller fan-out. We will use this fact in the development of the external segment tree.

# 3 The Buffer Tree

In this section we will develop the basic structure—which we call the *buffer tree*—and only consider the operations needed in order to use the structure in a simple sorting algorithm. In later sections we then extend this basic structure in order to obtain an external priority queue and an external (one-dimensional) range tree.

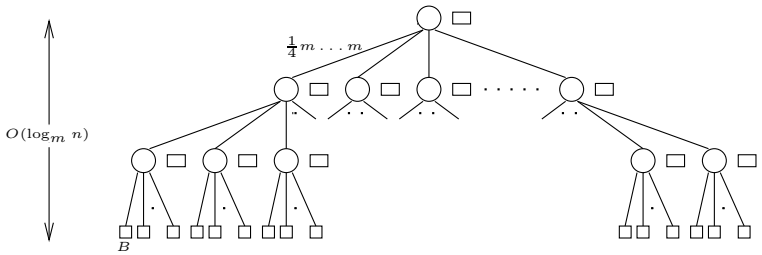



Figure 1: The buffer tree.

The buffer tree is an $(a, b)$-tree [15] with $a = m/4$ and $b = m$, extended with a buffer in each node. In such a tree all nodes except for the root have fan-out between $m/4$ and $m$, and thus the height of the tree is $O(\log_m n)$. The buffer tree is pictured in Figure 1. As discussed in section 2 we do the following when we want to do an update on the buffer tree: We construct a new element consisting of the element to be inserted or deleted, a time stamp, and an indication of whether the element is to be inserted or deleted. When we have collected $B$ such elements in internal memory, we insert the block in the buffer of the root. If the buffer of the root still contains less than $m/2$ blocks we stop. Otherwise, we empty the buffer. The buffer-emptying process is described in Figure 2 and 5. We define *internal nodes* to be all nodes which do not have leaves as children, and the basic part of the process which is used on these nodes (corresponding to the discussion in the last section) is given in Figure 2. Note that the buffer-emptying process is only done recursively on internal nodes. Only after finishing *all* buffer-emptying processes on internal nodes, we empty the buffers of the *leaf nodes* as we call the nodes which are not internal. That the buffer-emptying process

- Load the partitioning (or routing) elements of the node into internal memory.
- Repeatedly load (at most) $m/2$ blocks of the buffer into internal memory and do the following:
  1. Sort the elements from the buffer in internal memory. If two equal elements—an insertion and a deletion—"meet" during this process, and if the time stamps "fit", then the two elements annihilates.
  2. Partition the elements according to the partitioning elements and output them to the appropriate buffers one level down (maintaining the invariant that at most one block in a buffer is non-full).
- If the buffer of any of the children now contains more than $\frac{1}{2}m$ blocks, and if the children are internal nodes, then recursively apply the emptying-process on these nodes.

Figure 2: The buffer-emptying process on internal nodes.

Rebalancing after inserting an element below $v$:

```
DO v has b+1 children ->
  IF v is the root ->
    let x be a new node and make v its only child
  ELSE
    let x be the parent of v
  FI
  Let v' be a new node
  Let v' be a new child of x immediately after v
  Split v:
    Take the rightmost ⌈(b+1)/2⌉  children
     away from v and make them children of v'.
  Let v=x
OD
```

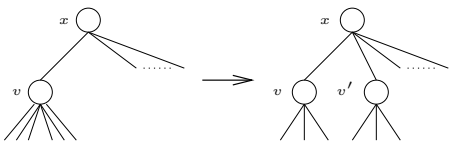
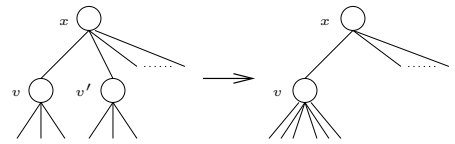



Figure 3: Insert in $(a, b)$ tree.

Rebalancing after deleting an element below $v$:

```
DO v has a-1 children AND
   v' has less than a+t+1 children ->
  Fuse v and v':
    Make all children of v' children of v
  Let v=x
  Let v' be a brother of x
  IF x does not have a brother (x is the root)
     AND x only has one child ->
    Delete x
    STOP
  FI
  Let x be the parent of v.
OD
(* either v has more than a children and we
 are finished, or we can finish by sharing *)
IF v has a-1 children ->
  Share:
    Take s children away from v' and
    make them children of v.
FI
```



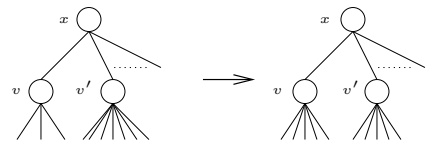




Figure 4: Delete from $(a, b)$-tree ($s = \lceil((b/2 - a) + 1)/2\rceil$ and $t = (b/2 - a) + s - 1$).

on an internal node can be done in a linear number of I/Os as required is easily realized: The elements are loaded and written ones, and at most $O(m)$ I/Os are used on writing non-filled blocks every time we load $m/2$ blocks. Note that the cost of emptying a buffer containing $o(m)$ blocks remains $O(m)$, as we distribute the elements to $\Theta(m)$ children.

The emptying of a leaf buffer is a bit more complicated as we also need to consider rebalancing of the structure when we empty such a buffer. The algorithm is given in Figure 5. Basically the rebalancing is done precisely as on normal $(a, b)$-trees [15]. After finding the position of a new element among the elements in the leaves of an $(a, b)$-tree, the rebalancing is done by a series of "splits" of node in the structure. We give the algorithm in Figure 3. Similarly, after deleting an element in a leaf the rebalancing is accomplished by a series of

- As long as there is a leaf node $v$ with a full buffer (size greater than $m/2$ blocks) do the following ($x$ is the number of leaves of $v$):
  1. Sort the elements in the buffer of $v$ with an optimal I/O sorting algorithm and remove "matching" insert/delete elements.
  2. Merge the sorted list with the sorted list of elements in the leaves of $v$ while removing "matching" insert/delete elements.
  3. If the number of blocks of elements in the resulting list is **smaller** than $x$ do the following:
     (a) Place the elements in sorted order in the leaves of $v$.
     (b) Add "dummy-blocks" until $v$ has $x$ leaves and update the partition elements in $v$.
  4. If the number of blocks of elements in the resulting list is **bigger** than $x$ do the following:
     (a) Place the $x$ smallest blocks in the leaves of $v$ and update the partition elements of $v$ accordingly.
     (b) Repeatedly insert one block of the rest of the elements and rebalance.
- Repeatedly delete one dummy block and rebalance—while performing a buffer-emptying process on the relevant nodes involved in a rebalance operation ($v'$ of Figure 4) before the operation is done (if $v'$ is a leaf node its buffer is emptied as described above).

  If the delete (or rather the buffer-emptying processes done as a result of it) results in any leaf buffer becoming full, these buffers are emptied as described above before the next dummy block is deleted.

Figure 5: Emptying the buffers of the leaf nodes.

node "fusions" possibly ending with a node "sharing". The algorithm is given in Figure 4. In the buffer tree case we need to modify the delete rebalancing algorithm slightly because of the buffers. The modification consists of doing a buffer-emptying process before every rebalance operation. More precisely, we do a buffer-emptying process on $v'$ in Figure 4 when it is involved in a fuse or share rebalancing operation. This way we can do the actual rebalancing operation as normally, without having to worry about elements in the buffers. This is due to the fact that our buffer-emptying process on internal nodes maintains the invariant that if the buffer of a leaf node runs full then all nodes on the path to the root have empty buffers. Thus when we start rebalancing the structure (insert and delete the relevant blocks) after emptying all the leaf buffers (Figure 5), all nodes playing the role of $v$ in split, fuse or share rebalance operations already have empty buffers. Also if the emptying of the buffer of $v'$ results in a leaf buffer running full, the invariant will be fulfilled because all nodes on the path from $v'$s parent $x$ to the root have empty buffers. Note that the reason for not doing buffer-emptying processes on leaf nodes recursively, is to prevent different rebalancing operations from interfering with each other. This is also the reason for the special way of handling deletes with dummy blocks; while deletion of a block may result in several buffer-emptying processes, this is not the case for insertions as no buffer-emptying process are necessary in this rebalancing algorithm.

We can now prove our main theorem.

**Theorem 1** *The total cost of an arbitrary sequence of $N$ intermixed insert and delete operation on an initially empty buffer tree is $O(n \log_m n)$ I/O operations, that is, the amortized cost of an operation is $O(\frac{\log_m n}{B})$ I/Os.*

*Proof*: As discussed in Section 2 the total cost of all the buffer-emptying processes on *internal* nodes with *full* buffers is bounded by $O(n \log_m n)$ I/Os. This follows from the fact that one such process uses a linear number of I/Os in the number of blocks pushed one level down.

During the rebalancing operations we empty a number of *non-full* buffers using $O(m)$ I/Os, namely one for each rebalancing operation following a deletion of a block. Furthermore, it is easy to realize that the administrative work in a rebalancing operation—updating partitioning elements and so on—can be performed in $O(m)$ I/Os. In [15] it is shown that the number of rebalancing operations in a sequence of $K$ updates on an initially empty $(a, b)$-tree is bounded by $O(K/(b/2 - a))$ if $b > 2a$. As we are inserting blocks in the $(m/4, m)$-tree underlying the buffer tree this means that the total number of rebalance operations in a sequence of $N$ updates on the buffer tree is bounded by $O(n/m)$. Thus the total cost of the rebalancing is $O(n)$.

The only I/Os we have not counted so far are the ones used on emptying *leaf buffers*. The number of I/Os used on a buffer-emptying process on a leaf node is dominated by the sorting of the elements (Figure 5, step 1). As any given element will only once be involved in such a sorting the *total* number of I/Os used to empty leaf buffers is bounded by $O(n \log_m n)$. This proves the lemma. □

In order to use the transformed structure in a simple sorting algorithm, we need a empty/write operation that empties all the buffers and then reports the elements in the leaves in sorted order. The emptying of all buffers can easily be done just by performing a buffer-emptying process on all nodes in the tree—from the top. As emptying one buffer costs $O(m)$ I/Os amortized, and as the total number of buffers in the tree is $O(n/m)$, we have the following:

**Theorem 2** *The amortized I/O cost of emptying all buffers of a buffer tree after performing $N$ updates on it, and reporting all the remaining elements in sorted order, is $O(n)$.*

**Corollary 1** *$N$ elements can be sorted in $O(n \log_m n)$ I/O operations using the buffer tree.*

As mentioned the above result is optimal and our sorting algorithm is the first that does not require all the elements to be present by the start of the algorithm. In Section 7 we discuss how to avoid the sorting algorithm used in the buffer-emptying algorithm.

Before continuing to design more operations on the buffer tree a note should be made on the balancing strategy used. We could have used a simpler balancing strategy than the one presented in this section. Instead of balancing the tree bottom-up we can balance it in a top-down style. We can make such a strategy work, if we "tune" our constants (fan-out and buffer-size) in such a way that the maximal number of elements in the buffers of a subtree is guaranteed to be less that half the number of elements in the leaves of the subtree. If this is the case we can do the rebalancing of a node when we empty its buffer. More precisely, we can do a split, a fuse or a sharing in connection with the buffer-emptying process on a node, in order to guarantee that there is room in the node to allow all its children to fuse or split. In this way we can make sure that rebalancing will never propagate. Unfortunately, we have not been able to make this simpler strategy work when rangesearch operations (as discussed in Section 5) are allowed.

# 4 An External Priority Queue

Normally, we can use a search tree structure to implement a priority queue because we know that the smallest element in a search tree is in the leftmost leaf. The same strategy can be

used to implement an external priority queue based on the buffer tree. There are a couple of problems though, because using the buffer tree we cannot be sure that the smallest element is in the leftmost leaf, as there can be smaller elements in the buffers of the nodes on the leftmost path. However, there is a simple strategy for performing a deletemin operation in the desired amortized I/O bound. When we want to perform a deletemin operation we simply do a buffer-emptying process on all nodes on the path from the root to the leftmost leaf. To do so we use $O(m \cdot \log_m n)$ I/Os amortized. After doing so we can be sure not only that the leftmost leaf consists of the $B$ smallest elements, but also that (at least) the $\frac{1}{4}m \cdot B$ smallest elements in the tree are in the children (leaves) of the leftmost leaf. If we delete these elements and keep them in internal memory, we can answer the next $\frac{1}{4}m \cdot B$ deletemin operations without doing any I/Os. Of course we then also have to check insertions and deletions against the minimal elements in internal memory. This can be done in a straightforward way without doing extra I/Os, and a simple amortization argument gives us the following:

**Theorem 3** *The total cost of an arbitrary sequence of $N$ insert, delete and deletemin operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations, that is, the amortized cost of an operation is $O(\frac{\log_m n}{B})$ I/Os.*

Note that in the above result we use $m/4$ blocks of internal memory to hold the minimal elements. In some applications (e.g. in [5]) we would like to use less internal memory for the external priority queue structure. Actually, we can make our priority queue work with as little as $\frac{1}{4}m^{1/c}$ $(0 < c \leq 1)$ blocks of internal memory, by decreasing the fan-out and the size of the buffers to $\Theta(m^{1/c})$ as discussed in Section 2.

### 4.1 Application: Time-Forward Processing

As mentioned in the introduction a technique for evaluating circuits (or "circuit-like" computations) in external memory is developed in [11]. This technique is called time-forward processing. The problem is the following: We are given a bounded fan-in boolean circuit, whose description is stored in external memory, and want to evaluate the function computed by the network. It is assumed that the representation of the circuit is topologically sorted, that is, the labels of the nodes come from a total order $<$, and for every edge $(v, w)$ we have $v < w$. Nothing is assumed about the functions in the nodes, except that they take at most $M/2$ inputs. Thinking of vertex $v$ as being evaluated at "time" $v$ motivates calling an evaluation of a circuit a time-forward processing. The main issue in such an evaluation is to ensure that when one evaluates a particular vertex one has the values of its inputs in internal memory.

In [11] an external-memory algorithm using $O(n \log_m n)$ I/Os is developed (here $N$ is the number of nodes plus the number of edges). The algorithm uses a number of known as well as new I/O-algorithm design techniques and is not particularly simple. Furthermore, the algorithm only works for large values of $m$, more precisely it works if $\sqrt{m/2} \log(M/2) \geq 2 \log(2N/M)$. For typical machines this constraint will be fulfilled. Using our external priority queue however, it is obvious how to develop a simple alternative algorithm—without the constraint on the value of $m$. When we evaluate a node $v$ we simply send the result forward in time to the appropriate nodes, by inserting a copy of the result in the priority queue with priority $w$ for all edges $(v, w)$. We can then obtain the inputs to the next node in the topological order just by performing a number of deletemin operations on the queue. The $O(n \log_m n)$ I/O-bound follows immediately from Theorem 3.

In [11] a randomized and two deterministic algorithms for external-memory list ranking are developed. One of these algorithms uses time-forward processing and therefore inherits the constraint that $m$ should not be too small. The other has a constraint on $B$ not being to large (which in turn also results in a constraint on $m$ not being to small). As mentioned, the list ranking algorithm is in turn used to develop efficient external algorithms for a number of problems. This means that by developing an alternative time-forward processing algorithm without the constraint on $m$, we have also removed the constraint from the algorithm for list ranking, as well as from a large number of other external-memory graph algorithms.

# 5 An External (one-dimensional) Range Tree Structure

In this section we extend the basic buffer tree with a rangesearch operation in order to obtain an external (one-dimensional) range tree structure.

Normally, one performs a rangesearch with $x_1$ and $x_2$ on a search tree by searching down the tree for the positions of $x_1$ and $x_2$ among the elements in the leaves, and then one reports all the elements between $x_1$ and $x_2$. However, the result can also be generated while searching down the tree, by reporting the elements in the relevant subtrees on the way down. This is the strategy we use on the buffer tree. The general idea in our rangesearch operation is the following: We start almost as when we do an insertion or a deletion. We make a new element containing the interval $[x_1, x_2]$ and a time stamp, and insert it in the tree. We then have to modify our buffer-emptying process in order to deal with the new rangesearch elements. The basic idea is that when we meet a rangesearch element in a buffer-empty process, we first determine whether $x_1$ and $x_2$ are contained in the same subtree among the subtrees rooted at the children of the node in question. If this is the case we just insert the element in the corresponding buffer. Otherwise we "split" the element in two—one for $x_1$ and one for $x_2$—and report the elements in the subtrees for which *all* elements in them are contained in the interval $[x_1, x_2]$. The splitting only occurs once and after that the rangesearch elements are pushed downwards in the buffer-emptying processes like insert and delete elements, while elements in the subtrees for which all the elements are in the interval are reported. As discussed in the introduction and Section 2 this means that the rangesearch operation gets batched.

In order to make the above strategy work efficiently we need to overcome several complications. One major complication is the algorithm for reporting all elements in a subtree. For several reasons we cannot just use the simple algorithm presented in Section 3, and empty the buffers of the subtree by doing a buffer-emptying process on all nodes and then report the elements in the leaves. The major reason is that the buffers of the tree may contain other rangesearch elements, and that we should also report the elements contained in the intervals of these queries. Also in order to obtain the desired I/O bound on the rangesearch operation, we should be able to report the elements in a tree in $O(n_a)$ I/Os, where $n_a$ is the actual number of blocks in the tree, that is, the number of blocks used by elements which are not deleted by delete elements in the buffers of the tree. This number could be a low as zero. However, if $n_d$ is the number of blocks deleted by delete elements in the tree, we have that $n = n_a + n_d$. This means that if we can empty all the buffers in the tree—*and* remove all the delete elements—in $O(n)$ I/Os, we can charge the $n_d$ part to the delete elements, adding $O(\frac{1}{B})$ to the amortized number of I/Os used by a delete operation (or put another way; we in total use $O(n)$ I/Os extra to remove *all* the delete elements).

In Subsection 5.1 we design an algorithm for emptying all the buffers of a (sub-) buffer tree in a linear number of I/Os. Our algorithm reports all relevant "hits" between rangesearch and normal elements in the tree. In Subsection 5.2 we then show precisely how to modify the buffer-emptying process on the buffer tree in order to obtain the efficient rangesearch operation.

### 5.1 Emptying all Buffers of an External Range Tree

In order to design the algorithm for emptying all buffers we need to restrict the operations on the structure. In the following we assume that we only try to delete elements from a buffer tree which were previously inserted in the tree. The assumption is natural (at least) in a batched dynamic environment. Having made this assumption we obtain the following useful properties: If $d, i$ and $s$ are matching delete, insert and rangesearch elements (that is, "$i = d$" and "$i$ is contained in $s$"), and if we know that their time order is $d, s, i$ (and that no other elements—especially not rangesearch elements—are between $s$ and $i$ in the time order), then we can report that $i$ is in $s$ and remove $i$ and $d$. If we know that the time order is $s, i$ (knowing that no element—especially not $d$—is between $s$ and $i$ in the time order), we can report that $i$ is in $s$ and interchange their time order. Similarly, if we know that the time order is $d, s$, we can again report that $d$ is in $s$ (because we know that there is an $i$ matching $d$ "on the other side of $s$") and interchange their time order.

We define a set of (insert, delete and rangesearch) elements to be in *time order representation* if the time order is such that all the delete elements are "older" than (were inserted before) all the rangesearch elements, which in turn are older than all the insert elements, *and* if the three groups of elements are internally sorted (according to $x$ and not time order—according to $x_1$ as far as the rangesearch elements are concerned). Using the above properties about interchanging the time order, we can now prove two important lemmas.

**Lemma 1** *A set of less than $M$ elements in a buffer can be made into time order representation, while the relevant $r \cdot B$ matching rangesearch elements and elements are reported, in $O(m + r)$ I/Os.*

*Proof*: The algorithm simply loads the elements into internal memory and use the special assumption on the delete elements to interchange the time order and report the relevant elements as discussed above. Then it sorts the three groups of elements individually and writes them back to the buffer in time order representation. □

**Lemma 2** *Let two sets $S_1$ and $S_2$ in time order representation be a subset of a set $S$ such that all elements in $S_2$ are older than all elements in $S_1$, and such that each of the other elements in $S$ is either younger or older than all elements in $S_1$ and $S_2$. $S_1$ and $S_2$ can be "merged" into one set in time order representation, while the relevant $r \cdot B$ matching rangesearch elements and elements are reported, in $O((|S_1| + |S_2|)/B + r)$ I/Os.*

*Proof*: The algorithm for merging the two sets is given in Figure 6. In step one we push the delete elements $d_1$ in $S_1$ down in the time order by "merging" them with the insert elements $i_2$ in $S_2$, and in step two we push them further down by "merging" them with the rangesearch elements $s_2$ in $S_2$. That we in both cases can do so without missing any rangesearch-element "hits" follows from the time order on the elements and the assumption on the delete elements

When we in the following write that we report "hits", we actually accumulate elements to be reported in internal memory and report/write them as soon as we have accumulated a block.

1. Interchange the time order of $d_1$ and $i_2$ by "merging" them while removing delete/insert matches.
2. Interchange the time order of $d_1$ and $s_2$ by "merging" them while reporting "hits" in the following way:

   During the merge a third list of "active" rangesearch elements from $s_2$ is kept—except for the $B$ most recently added elements—on disk.

   When a rangesearch element from $s_2$ has the smallest $x$ (that is, $x_1$) value, it is insert in the list. When a delete element from $d_1$ has the smallest $x$-value, the list is scanned and it is reported that the element is in the interval of all rangesearch elements that have not yet "expired"—that is, elements whose $x_2$ value is less than the value of the element from $d_2$. At the same time all rangesearch elements that have expired are removed from the list.
3. Interchange the time order of $s_1$ and $i_2$ by "merging" them and reporting "hits" like in the previous step.
4. Merge $i_1$ with $i_2$, $s_1$ with $s_2$ and $d_1$ with $d_2$.

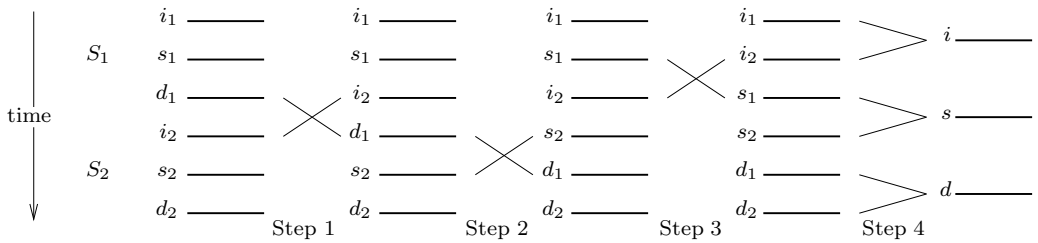



Figure 6: Merging sets in time order representation.

as discussed above. Then in step three the time order of $s_1$ and $i_2$ is interchanged, such that the relevant lists can be merged in step four. That step one and four are done in a linear number of I/Os is obvious, while a simple amortization argument shows that step two and three are also done in a linear number of I/Os, plus the number of I/Os used to report "hits". □

After proving the two lemmas we are now almost ready to present the algorithm for emptying the buffers of all nodes in a subtree. The algorithm will use that all elements in the buffers of nodes on a given level of the structure are always in correct time order compared to all *relevant* elements on higher levels. By relevant we mean that an element in the buffer of node $v$ was inserted in the tree before all elements in buffers of the nodes on the path from $v$ to the root of the tree. This means that we can assume that all elements on one level were inserted before all elements on higher levels.

**Lemma 3** *All buffers of a (sub-) range tree with $n$ leaves, where all buffers contain less than $M$ elements, can be emptied and all the elements collected into time order representation in $O(n + r)$ I/Os. Here $r \cdot B$ is the number of matching element and rangesearch elements reported.*

*Proof*: The empty algorithm is given in Figure 7. The correctness of the algorithm follows from Lemma 1 and Lemma 2 and the above discussion. It follows from Lemma 1 that step one creates the time order representation of the elements on each of the levels in a number

1. Make three lists for each level of the tree, consisting of the elements on the level in question in time order representation:

   For a given level all buffers are made into time order representation using Lemma 1, and then the resulting lists are appended after each other to obtain the total time order representation.
2. Repeatedly and from the top level, merge the time order representation of one level with the time order representation of the next level using Lemma 2.

After step one



After step two

Figure 7: Emptying all buffers and collecting the elements in time order representation.

of I/Os equal to $O(m)$ times the number of nodes in the tree, plus the number of I/Os used to report hits. That the total number of I/Os used is $O(n + r)$ then follows from the fact that the number of nodes in a tree with $n$ leaves is $O(n/m)$. That one merge in step two takes a linear number of I/Os in the number of elements in the lists, plus the I/Os used to report hits, follows form Lemma 2. That the total number of I/Os used is $O(n + r)$ then follows from the fact that every level of the tree contains more nodes than all levels above it put together. Thus the number of I/Os used to merge the time order representation of level $j$ with the time order representation of all the elements above level $j$ is bounded by $O(m)$ times the number of nodes on level $j$. The bound then again follows from the fact that the total number of nodes in the tree is $O(n/m)$. □

### 5.2 Buffer-emptying Process on External Range Tree

Having introduced the time order representation and discussed how to empty the buffers of a subtree, we are now ready to describe the buffer-emptying process used on the external range tree. The process is given in Figure 8 and it relies on an important property, namely that when we start emptying a buffer the elements in it can be divided into two categories—what we call "old" and "new" elements. The new elements are those which were inserted in the buffer by the buffer-emptying process that just took place on the parent node, and which triggered the need for a buffer-emptying process on the current node. The old elements are the rest of the elements. We know that the number of old elements is less than $M$ and that they were all inserted before the new elements. As we will maintain the invariant that we distribute elements from one buffer to the buffers one level down in time order representation, this means that we can construct the time order representation of *all* the elements in a buffer as described in the first two steps of the algorithm in Figure 8.

Now if we are working on a leaf node we can report the relevant "hits" between rangesearch element and normal element, just by merging the time order representation of the elements in the buffer with the time order representation consisting of the elements in the leaves below the node. Then we can remove the rangesearch elements and we are ready to empty the leaf buffer (and rebalance) with the algorithm used on the basic buffer tree.

- Load the less than $M$ old elements in the buffer and make them into time order representation using Lemma 1.
- Merge the the old elements in time order representation with the new elements in time order representation using Lemma 2.
- If we are working on a **leaf** node:
  1. Merge (a copy of) the time order representation with the time order representation consisting of the elements in the children (leaves) using Lemma 2.
  2. Remove the rangesearch elements from the buffer.
- If we are working on an **internal** node:
  1. Scan the delete elements and distribute them to the buffers of the relevant children.
  2. Scan the rangesearch elements and compute which of the subtrees below the current node should have their elements reported.
  3. For every of the relevant subtrees do the following:
     (a) Remove and store the delete elements distributed to the buffer of the root of the subtree in step one above.
     (b) Empty the buffers of the subtree using Lemma 3.
     (c) Merge the resulting time order representation with the time order representation consisting of the delete elements stored in (a) using Lemma 2.
     (d) Scan the insert and delete elements of the resulting time order representation and distribute a copy of the elements to the relevant leaf buffers.
     (e) Merge the time order representation with the time order representation consisting of the elements in the leaves of the subtree using Lemma 2.
     (f) Remove the rangesearch elements.
     (g) Report the resulting elements as being "hit" by the relevant search elements in the buffer.
  4. Scan the rangesearch elements again and distribute them to the buffers of the relevant children.
  5. Scan the insert elements and distribute them to the buffers of the relevant children. Elements for the subtrees which were emptied are distributed to the leaf buffer of these trees.
  6. If the buffer of any of the children now contains more than $m/2$ elements then recursively apply the buffer-emptying process on these nodes.
- When all buffers of the relevant internal nodes are emptied (and the buffers of all relevant leaf nodes have had their rangesearch elements removed) then empty all leaf buffers involved in the above process (and rebalance the tree) using the algorithm given in Figure 5 (Section 3).

Figure 8: Range tree buffer-emptying process.

If we are working on an internal node $v$ things are a bit more complicated. After computing which subtrees we need to empty, we basically do the following for each such tree: We empty the buffers of the subtree using Lemma 3 (step 3, b). We can use Lemma 3 as we know that all buffers of the subtree are non-full, because the buffer-emptying process is done recursively top-down. As discussed the emptying also reports the relevant "hits" between elements and rangesearch elements in the buffers of the subtree. Then we remove the elements which are deleted by delete elements in the buffer of $v$ (step 3, c). Together with the relevant insert elements from the buffer of $v$, the resulting set of elements should be inserted in or deleted from the tree. This is done by inserting them in the relevant leaf buffers (step 3, d and step 5), which are then emptied at the very end of the algorithm. Finally, we merge the time order representation of the elements from the buffers of the subtree with the elements in the leaves of the structure (step 3, e). Thus we at the same time report the relevant "hits" between

elements in the leaves and rangesearch elements from the buffers, and obtain a total list of (undeleted) elements in the subtree. These elements can then be reported as being "hit" by the relevant rangesearch elements from the buffer of $v$ (step 3, g).

After having written/reported the relevant subtrees we can distribute the remaining elements in the buffer of $v$ to buffers one level down—remembering to maintain the invariant that the elements are distributed in time order representation—and then recursively empty the buffers of the relevant children. When the process terminates we empty the buffers of all leaf nodes involved in the process. As these leaf nodes now do not contain any rangesearch elements, this can be done with the algorithm used on the basic buffer tree in Section 3.

**Theorem 4** *The total cost of an arbitrary sequence of $N$ intermixed insert, delete and rangesearch operations performed on an initially empty range tree is $O(n \log_m n + r)$ I/O operations. Here $r \cdot B$ is the number of reported elements.*

*Proof*: The correctness of the algorithm follows from the above discussion. It is relatively easy to realize (using Lemma 1, 2 and 3) that one buffer-emptying process uses a linear number of I/Os in the number of elements in the emptied buffer and the number of elements in the leaves of the emptied subtrees, plus the number of I/Os used to report "hits" between elements and rangesearch elements. The only I/Os we can not account for using the standard argument presented in Section 2 are the ones used on emptying the subtrees. However, as discussed in the beginning of the section, this cost can be divided between the elements reported and the elements deleted, such that the deleted elements pay for their own deletion. The key point is that once the elements in the buffers of the internal nodes of a subtree is removed and inserted in the leaf buffers by the described process, they will only be touched again when they are inserted in or deleted from the tree by the rebalancing algorithm. This is due to the fact mentioned in Section 3 that when a buffer is emptied all buffers on the path to the root are empty, and the fact that we empty all relevant leaf buffers at the end of our buffer-emptying algorithm. □

### 5.3 Application: Orthogonal Line Segment Intersection Reporting

The problem of *orthogonal line segment intersection reporting* is defined as follows: We are given $N$ line segments parallel to the axes and should report all intersections of orthogonal segments. The optimal plane-sweep algorithm (see e.g. [22]) makes a vertical sweep with a horizontal line, inserting the $x$ coordinate of a vertical segments in a search tree when its top endpoint is reached, and deleting it again when its bottom endpoint is reached. When the sweep-line reaches a horizontal segment, a rangesearch operation with the two endpoints of the segment is performed on the tree in order to report intersections. In internal memory this algorithm will run in the optimal $O(N \log_2 N + R)$ time.

Using precisely the same algorithm and our range tree data structure, and remembering to empty the tree when we are done with the sweep, we immediately obtain the following (using Theorem 4 and Lemma 3):

**Corollary 2** *Using our external range tree the orthogonal line segment intersection reporting problem can be solved in $O(n \log_m n + r)$ I/Os.*

As mentioned an algorithm for the problem is also developed in [14], but this algorithm is very I/O specific whereas our algorithm ''hides" the I/O in the range tree. That the algorithm

is optimal in the comparison I/O model follows from the $\Omega(N \log_2 N + R)$ comparison model lower bound, and the general connection between comparison and I/O lower bounds proved in [4].

# 6 An External Segment Tree

In this section we use our technique to develop an external memory version of the segment tree. As mentioned this will enable us to solve the batched range searching and the pairwise rectangle intersection problems in the optimal number of I/Os.

The segment tree [8, 22] is a well-known data structure used to maintain a dynamically changing set of segments whose endpoints belongs to a fixed set, such that given a query point all segments that contain the point can be found efficiently. Such queries are normally called *stabbing queries*. The internal-memory segment tree consists of a static binary tree (the base tree) over the sorted set of endpoints, and a given segment is stored in up to two nodes on each level of the tree. More precisely an interval is associated with each node, consisting of all endpoints below the node, and a segment is stored in all nodes where it contains this interval but not the interval associated with the parent node. The segments stored in a node is just stored in an unordered list. To answer a stabbing query with a point $x$, one just has to search down the structure for the position of $x$ among the leaves and report all segments stored in nodes encountered in this search.

Because a segment can be stored in $O(\log_2 N)$ nodes the technique sketched in section 2, where we just group the nodes in an internal version of the structure into super-nodes, does not apply directly. The main reason for this is that we would then be forced to use many I/Os to store a segment in these many lists. Instead, we need to change the definition of the segment tree. Our external segment tree is sketched in Figure 9. The base structure is a perfectly balanced tree with branching factor $\sqrt{m}$ over the set of endpoints. We assume without loss of generality that the endpoints of the segments are all distinct and that $\sqrt{m}$ divides $n$. A buffer and $m/2 - \sqrt{m}/2$ lists of segments are associated with each node. A list (block) of segments is also associated with each leaf. A set of segments is stored in this structure as follows: The first level of the tree (the root) partitions the data into $\sqrt{m}$ slabs $\sigma_i$, separated by dotted lines in Figure 9. The *multislabs* for the root are then defined as contiguous ranges of slabs, such as for example $[\sigma_1, \sigma_4]$. There are $m/2 - \sqrt{m}/2$ multislabs and the lists associated with a node are precisely a list for each of the multislabs. Segments such as $\overline{CD}$ that completely span one or more slabs are then called *long segments*, and a copy of each long segment is stored in a list associated with the largest multislab it spans. Thus,

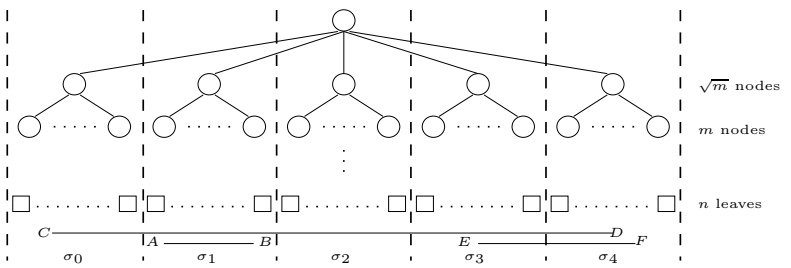



Figure 9: An external segment tree based on a set of $N$ segments, three of which, $\overline{AB}$, $\overline{EF}$ and $\overline{EF}$, are shown.

$\overline{CD}$ is stored in the list associated with the multislab $[\sigma_1, \sigma_3]$. All segments that are not long are called *short segments* and are not stored in any multislab list. Instead, they are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. Additionally, the portions of long segments that do not completely span slabs are treated as small segments. There are at most two such synthetically generated short segments for each long segment. Segments passed down to a leaf are just stored in one list. Note that we at most store one block of segments in each leaf. A segment is thus stored in at most two list on each level of the base tree.

Given an external segment tree (with empty buffers) a stabbing query can in analogy with the internal case be answered by searching down the tree for the query value, and at every node encountered report all the long segments associated with each of the multislabs that span the query value. However, answering queries on an individual basis is of course not I/O-efficient. Instead we use the buffer approach as discussed in the next subsection.

### 6.1 Operations on the External Segment Tree

Usually, when we use a segment tree to solve e.g. the batched range searching problem, we use the operations insert, delete and query. However, a delete operation is not really necessary, as we in the plane-sweep algorithm always know at which "time" a segment should be deleted when it is inserted in the tree. So in our implementation of the external segment tree we will not support the delete operation. Instead, we require that a delete time is given when a segment is inserted in the tree. Note that we already assume (like one normally does in internal memory) that we know the set of $x$ coordinates of the endpoints of segments to be inserted in the tree. In general these assumptions mean that our structure can only be used to solve batched dynamic problems as discussed in the introduction.

It is easy to realize how the base tree structure can be build in $O(n)$ I/O operations given the endpoints in sorted order. First we construct the leaves by scanning through the sorted list, and then we repeatedly construct one more level of the tree by scanning through the previous level of nodes (leaves). In constructing one level we use a number of I/Os proportional to the number of nodes on the previous level, which means that we in total use $O(n)$ I/Os as this is the total number of nodes and leaves in the tree.

When we want to perform an insert or a query operation on the buffered segment tree we do as sketched in Section 2. We make a new element with the segment or query point in question, a time-stamp, and—if the element is an insert element—a delete time. When we have collected a block of such elements, we insert them in the buffer of the root. If the buffer of the root now contains more than $m/2$ elements we perform a buffer-emptying process on it. The buffer-emptying process is presented in Figure 10, and we can now prove the following:

**Theorem 5** *Given a sequence of insertions of segments (with delete time) intermixed with stabbing queries, such that the total number of operations is $N$, we can build an external-memory segment tree on the segments and perform all the operation on it in $O(n \log_m n + r)$ I/O operations.*

*Proof*: In order to build the base tree we first use $O(n \log_m n)$ I/Os to sort the endpoints of the segments and then $O(n)$ I/Os to build the tree as discussed above. Next, we perform all the operations. In order to prove that this can be done in $O(n \log_m n + r)$ I/Os we should argue that we can do a buffer-emptying process in a linear number of I/Os. The bound then follows as previously.

**On internal nodes:**

- Repeatedly load $m/2$ blocks of elements into internal memory and do the following:
  1. Store segments:
     Collect all segments that should be stored in the node (long segments). Then for every multislab list in turn insert the relevant long segment in the list (maintaining the invariant that at most one block of a list is non full). At the same time replace every long segment with the small (synthetic) segments which should be stored recursively.
  2. Report stabbings:
     For every multislab list in turn decide if the segments in the list are stabbed by any query point. If so then scan through the list and report the relevant elements while removing segments which have expired (segments for which all the relevant queries are inserted after their delete time).
  3. Distribute the segments and the queries to the buffers of the nodes on the next level.
- If the buffer of any of the children now contains more than $\frac{1}{2}m$ blocks, the buffer-emptying process is recursively applied on these nodes.

**On leaf nodes:**

- Do exactly the same as with the internal nodes, except that when distributing *segments* to a child/leaf they are just inserted in a the segment block associated with the leaf.
  As far as the *queries* are concerned, report stabbings with segments from the multislab lists as on internal nodes (and the lists associated with the leaves) and remove the query elements.

Figure 10: The buffer-emptying process.

First consider the buffer-emptying process on an internal node. Loading and distributing the segments to buffers one level down can obviously be done in a linear number of I/Os. The key to realizing that step one also uses $O(m)$ I/Os on each memory load is that the number of multislab lists is $O(m)$. In analogy with the distribution of elements to buffers one level down, this means that the number of I/Os we use on inserting non-full blocks in the multislab lists is bounded by $O(m)$. The number used on full blocks is also $O(m)$, as this is the number of segments and as every segment is at most stored in one list. The number of I/Os charged to the buffer-emptying process in step two is also $O(m)$, as this is the number of I/Os used to load non-full multislab list blocks. The rest of the I/Os used to scan a multislab list can either be charged to a stabbing reporting or to the deletion of an element. We can do so by assuming that every segment holds $O(1/B)$ credits to pay for its own deletion. This credit can then be accounted for (deposited) when we insert a segment in a multislab list. Thus a buffer-emptying process on an internal node can be performed in a linear number of I/Os as required.

As far as leaf nodes are concerned almost precisely the same argument applies, and we have thus proved that we can build the base tree and perform all the operations on the structure in $O(n \log_m n + r)$ I/Os, where $r$ is the number of stabbings reported so far. However, in order to report the remaining stabbings we need to empty all the buffers of the structure. We do so as in Section 3 simply by performing buffer-emptying processes on all nodes level by level starting at the root. As there are $O(n/\sqrt{m})$ nodes in the tree one might think that this process would cost us $O(n\sqrt{m})$ I/Os, plus the number of I/Os used to report stabbings. The problem seems to be that there is $n/\sqrt{m}$ leaf nodes, each having $O(m)$ multislab lists, and that we when we empty the buffers of these nodes can be forced to use an I/O for each of the multislab lists which are not paid for by reportings. However, the number of multislab lists actually containing *any* segments must be bounded by $O(n \log_m n)$, as that is the number of I/Os performed so far. Thus it is easy to realize that $O(n \log_m n + r)$ must be a bound

on the number of I/Os we have to pay in order to empty the buffers of all the leaf nodes. Furthermore, as the number of internal nodes is $O(n/m)$, the buffers of these nodes can all be emptied in $O(n)$ I/Os. This completes the proof of the lemma.

□

### 6.2 Applications of the External Segment Tree

Having developed an external segment tree we can obtain efficient external-memory algorithms by using it in standard plane-sweep algorithms.

The batched range searching problem—given $N$ points and $N$ rectangles, report for each rectangle all the points that lie inside it—can be solved with a plane-sweep algorithm in almost the same way as the orthogonal line segment intersection problem. The optimal plane sweep algorithm makes a vertical sweep with a horizontal line, inserting a segment (rectangle) in the segment tree when the top segment of a rectangle is reached, and deleting it when the bottom segment is reached. When a point is reached a stabbing query is performed with it. Using our external segment tree in this algorithm yields the following:

**Corollary 3** *Using the external range tree the batched range searching problem can be solved in $O(n \log_m n + r)$ I/O operations.*

The problem of pairwise rectangle intersection is defined similar to the orthogonal line segment intersection problem. Given $N$ rectangles in the plane (with sides parallel to the axes) we should report all intersecting pairs. In [8] it is shown that if we—besides the orthogonal line segment intersection problem—can solve the batched range searching problem in time $O(N \log_2 N + R)$, we will in total obtain a solution to the rectangle intersection problem with the same (optimal) time bound. Thus we in external-memory obtain the following:

**Corollary 4** *Using our external data structures the pairwise rectangle intersection problem can be solved in $O(n \log_m n + r)$ I/O operations.*

That both algorithms are optimal in the comparison I/O model follows by the internal memory comparison lower bound and the result in [4]. Like in the orthogonal line segment intersection case, optimal algorithms for the two problems are also developed in [14].

## 7 Extending the Results to the $D$-disk Model

As mentioned in the introduction an approach to increase the throughput of I/O systems is to use a number of disks in parallel. One method of using $D$ disks in parallel is *disk striping*, in which the heads of the disks are moving synchronously, so that in a single I/O operation each disk read or writes a block in the same location as each of the others. In terms of performance, disk striping has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality when $D$ is very large, it is often the method of choice in practice for using parallel disks [28].

The non optimality of disk striping can be demonstrated via the sorting bound. While sorting $N$ elements using disk striping and one of the the one-disk sorting algorithms requires $O(n/D \log_{m/D} n)$ I/Os the optimal bound is $O(n/D \log_m n)$ I/Os [1]. Note that the optimal bound results in a linear speedup in the number of disk. Nodine and Vitter [19] managed to

develop a theoretical optimal $D$-disk sorting algorithm based on merge sort, and later they also developed an optimal algorithm based on distribution sort [20]. In the latter algorithm it is assumed that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$, an assumption which clearly is non-restrictive in practice. The algorithm works as normal distribution sort by repeatedly distributing a set of $N$ elements into $\sqrt{m}$ sets of roughly equal size, such that all elements in the first set is smaller than all elements in the second set, and so on. The distribution is done in $O(n/D)$ I/Os, and the main issue in the algorithm is to make sure that the elements in one of the smaller sets can be read efficiently in parallel in the next phase of the algorithm, that is, that they are distributed relatively evenly among the disks.

To obtain the results in this paper we basically only used three "paradigms"; *distribution*, *merging* and *sorting*. We used distribution to distribute the elements in the buffer of a node to the buffers of nodes on the next level, and to multislab lists in the segment tree case. We used merging of two lists when emptying all buffers in a (sub-) buffer tree, and we sorted a set of elements when emptying the leaf buffers of the buffer tree.[3] While we of course can use an optimal $D$-disk sorting algorithm instead of a one-disk algorithm, and while it is easy to merge *two* lists in the optimal number of I/Os on parallel disks, we need to modify our use of distribution to make it work with $D$ disks. As mentioned Nodine and Vitter [20] developed an optimal way of doing distribution, but only when the distribution is done $O(\sqrt{m})$-wise. As already mentioned we can make our buffer tree work with fan-out and buffer size $\Theta(\sqrt{m})$ instead of $\Theta(m)$, and thus we can use the algorithm from [20] to make our structure work in the $D$-disk model. The external segment tree already has fan-out $\sqrt{m}$, but we still distribute elements (segments) to $\Theta(m)$ multislab list. Thus to make our external segment tree work on $D$ disks we decrease the fan-out to $m^{1/4}$, which does not change the asymptotic I/O bounds of the operations, but decreases the number of multislab lists to $\sqrt{m}$. Thus we can use the algorithm from [20] to do the distribution. To summerize our structures work in the general $D$-disk model under the non-restrictive assumption that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$.

# 8 Conclusion

In this paper we have developed a technique for transforming an internal-memory tree data structure into an efficient external memory structure. Using this technique we have developed an efficient external priority queue and batched dynamic versions of the (one-dimensional) range tree and the segment tree. We have shown how these structures allow us to design efficient external-memory algorithms from known internal algorithms in a straightforward way, such that all the I/O specific parts of the algorithms are "hidden" in the data structures. This is in great contrast to previously developed algorithms for the considered problems. We have also used our priority queue to develop an extremely simple algorithm for "circuit evaluation", improving on the previously know algorithm.

Recently, several authors have used the structures developed in this paper or modified versions of them to solve important external-memory problems. In [2] the priority queue is used to develop new I/O efficient algorithms for ordered binary-decision diagram manipulation, and in [9] it is used in the development of several efficient external graph algorithm. In [5] an extension of the segment tree is used to develop efficient new external algorithms

[3] Note that we could actually do without the sorting by distributing elements in sorted order when emptying a buffer in the buffer tree, precisely as we in the range tree structure distribute them in time order representation.

for a number of important problems involving line segments in the plane, and in [6] the main idea behind the external segment tree (the notion of multislabs) is used to develop an optimal "on-line" versions of the interval tree.

We believe that several of our structures will be efficient in practice due to small constants in the asymptotic bounds. We hope in the future to be able to implement some of the structures in the transparent parallel I/O environment (TPIE) developed by Vengroff [27].

## Acknowledgments

I would like to thank all the people in the algorithms group at University of Aarhus for valuable help and inspiration. Special thanks also go to Mikael Knudsen for the discussions that lead to many of the results in this paper, to Sven Skyum for many computational geometry discussions, and to Peter Bro Miltersen, Erik Meineche Schmidt and Darren Erik Vengroff for help on the presentation of the results in this paper. Finally, I would like to thank Jeff Vitter for allowing me to be a part of the inspiring atmosphere at Duke University.

## References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995.

[3] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, February 1996.

[4] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 83–94, 1993.

[5] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. A full version is to appear in special issue of Algorithmica.

[6] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 1996.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indizes. *Acta Informatica*, 1:173–189, 1972.

[8] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[9] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results.* PhD thesis, Brown University, August 1995.

[10] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[11] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[12] D. Cormer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[14] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[15] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[16] C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93, 1987.

[17] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. Principles of Database Systems*, 1993. Invited to special issue of JCSS on Principles of Database Systems (to appear). A complete version appears as technical report 90-31, Brown University.

[18] D. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching*. Addison-Wesley, 1973.

[19] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1991.

[20] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[21] Y. N. Patt. The I/O subsystem — a candidate for improvement. *Guest Editor's Introduction in IEEE Computer*, 27(3):15–16, 1994.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[23] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, 1994.

[24] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[25] M. Smid. *Dynamic Data Structures on Multiple Storage Media*. PhD thesis, University of Amsterdam, 1989.

[26] S. Subramanian and S. Ramaswamy. The p-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[27] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[28] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.

[29] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.

[30] J. S. Vitter. Efficient memory access in large-scale computation (invited paper). In *Symposium on Theoretical Aspects of Computer Science, LNCS 480*, pages 26–41, 1991.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.