Chapter 2

Hash Tables

If you want to get a book from the central library of the University of Karlsruhe, you have to order the book an hour in advance. The library personnel takes the book from the magazine. You can pick it up in a room with many shelves. The shelves are numbered. You find your book in the shelf numbered with the last *digits of your library card. Why the last digits and not the leading digits? Probably because this distributes the books more evenly about the shelves. For example, many first year students are likely to have the same leading digits in their card number. If they all try the system at the same time, many books would have to be crammed into a single shelf.*

The subject of this chapter is robust and efficient implementation of the above "delivery shelf data structure" in a computer where they are called *hash tables*. The meaning of "to hash" that gives the best hint to what is meant is "to bring into complete disorder" — elements of a set are intentionally stored in disorder to make them easier to find. Although this sounds paradoxical we will see that it can make perfect sense. Hash table accesses are among the most time critical parts in many computer programs. For example, scripting languages like `awk` [1] or `perl` [10] use hash tables as their *only* data structures. They use them in the form of *associative arrays* that allow arrays to be used with any kind of index type, e.g., strings. Compilers use hash tables for their *symbol table* that associates identifiers with information about them. Combinatorial search programs often use hash tables to avoid looking at the same situation multiple times. For example, chess programs use them to avoid evaluating a position twice that can be reached by different sequences of moves. One of the most widely used implementations of the *join* operation in relational data bases temporarily stores one of the participating relations in a hash table. Exercise 2.4 gives a special case example. Hash tables can often be used to replace applications of sorting (Chapter **??**) or of search trees (Chapter **??**). When this is possible one often gets a significant speedup.

More formally, we want to store a set M of elements. Each element e has a unique key $key(e)$. We want to support the following *dictionary* operations:

$M.insert(e : Element)$**:** $M := M \cup \{e\}$

$M.remove(k : Key)$**:** $M := M \setminus \{e\}$ where e is the unique element with $key(e) = k$, i.e., we assume that *key* is a one-to-one function.

$M.find(k : Key)$ If there is an $e \in M$ with $key(e) = k$ return e otherwise return a special element $\perp$.

In addition, we assume a mechanism that allows us to retrieve all elements in M. Since this *forall* operation is usually easy to implement and is not very much affected by the details of the implementation, we only discuss it in the exercises.

For example, in an English-German dictionary, the keys are English words and an element is an English word together with its German translations. In the library example, Keys are the library card numbers and elements are the book orders. Although there are many ways to implement dictionaries (for example using the *search trees* discussed in Chapter **??**), we concentrate on particularly efficient implementations using hash tables.

The basic idea behind a hash table is to map keys to array positions using a *hash function* $h : Key \to 0..m-1$. In the library example, h is the function extracting the least significant digits from a card number. Ideally, we would like to store element e in a table entry $t[h(key(e))]$. If this works, we get constant execution time for all three operations *insert*, *remove*, and *find*.[1]

Unfortunately, storing e in $t[h(key(e))]$ will not always work because several elements might *collide*, i.e., they might map to the same table entry. In Section 2.1 we investigate a conceptually very easy fix of this problem. When several book requests get directed to the same delivery shelf in the library example, there will be several books on this shelf and we have to search through all of them to find our order. In a computer we store a sequence of elements in each table entry. Because the sequences in each table entry are usually implemented using singly linked lists, this hashing scheme is known as *hashing with chaining*. We will first analyze hashing with chaining using rather optimistic assumptions on the properties of the hash function. In this model, we achieve constant expected time for dictionary operations.

In Section 2.2 we drop these assumptions and explain how to construct hash functions that come with (probabilistic) performance guarantees. Note that already our

[1] Strictly speaking we have to add additional terms to the execution time for moving elements and for evaluating the hash function. To simplify notation, we assume in this chapter that all this takes constant time. Timings in the more detailed model can be recovered by adding the time for one hash function evaluation and for a constant number of element moves in *insert* and *remove*.

simple examples show that finding good hash functions is nontrivial. For example, if we would apply the least significant digit idea from the library example to an English-German dictionary, we might come up with a hash function based on the last four letters of a word. But then we would have lots of collisions for words ending on ‘tion’, ‘able’, …

We can simplify hash tables (but not their analysis) by returning to the original idea to store all elementsn in the table itself. When a newly inserted element e finds entry $t[h(key(x))]$ occupied, it scans the table until a free entry is found. In the library example this would happen if the shelves are too small to hold more than one book. The librarians would then use the adjacent shelves to store books that map the same delivery shelf. Section 2.3 eloborates this idea which is known as *hashing with open adressing and linear probing*. Measurements indicate that linear probing is not only simpler but also faster than hashing with chaining.

Exercises

Exercise 2.1 (Scanning a hash table.) Implement the *forall* operation for your favorite hash table data type. The total execution time for accessing all elements should be linear in the size of the hash table. Your implementation should hide the representation of the hash table from the application. Here is one possible interface: Implement an *iterator* class that offers a view on the hash table as a sequence. It suffices to support initialization of the iterator, access to the current element and advancing the iterator to the next element.

Exercise 2.2 Assume you are given a set M of pairs of integers. M defines a binary relation R_M. Give an algorithm that checks in expected time $O(|M|)$ whether R_M is symmetric. A relation is symmetric if $\forall(a,b) \in M : (b,a) \in M$. Space consumption should be $O(|M|)$.

Exercise 2.3 Write a progam that reads a text file and produces a statistics about the 100 most frequent words in the text. Expected execution time should be linear in the size of the file.

Exercise 2.4 (A billing system:) Companies like Akamai[2] deliver millions and millions of files every day from their thousands of servers. These files are delivered for other E-commerce companies who pay for each delivery (fractions of a cent). The servers produce log files. One line for each access. Each line contains informations that can be used to deduce the price (fractions of a cent) and the customer ID. Explain

[2] `http://www.akamai.com/index_flash.html`

how to write a program that reads the log files once a day and outputs the total charges for each customer (a few hundred)⊗

2.1 Hashing with Chaining

Hashing with chaining maintains an array t with m entries each of which stores a sequence of elements. Assuming a hash function $h : Key \to 0..m-1$ we can implement the three dictionary functions as follows:

insert(e)**:** Insert e somewhere in sequence $t[h(key(e))]$.

remove(k)**:** Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, remove it and return.

find(k) : Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, return it. Otherwise, return $\perp$.

Figure **??** gives an example⊗

Let us analyze hashing with chaining. Insertion can be done in worst case constant time if the sequences are implemented using linked lists. Space consumption is at most $O(n+m)$ (see Exercise 2.7 for a more accurate analysis).

The other two operations have to scan the sequence $t[h(k)]$. In the worst case, for example if *find* looks for an element that is not there, the entire list has to be scanned. If we are unlucky all elements could be mapped to the same table entry. For n elements, we could would get execution time $O(n)$. Can we find hash functions that never fail? Unfortunately, the answer is no. Once we fix h, it is always possible to find a set of $n < |Key|/m$ keys that all map to the same table entry.

To get a less pessimistic — and hopefully realistic — estimate, we now look at the *average case* efficiency in the following sense: We fix a set of n elements and analyze the execution time averaged over all possible hash functions $h : Key \to 0..m-1$. Equivalently, we can ask what the expected time of a *remove* or *find* is if we pick a hash function uniformly at random from the set of all possible hash functions.

Theorem 2.1 *If n elements are stored in a hash table with m entries using hashing with chaining, the expected execution time of remove or find is $O(1+n/m)$ if we assume a random hash function.*

The proof is very simple once we know the probabilistic concepts of random variables, their expectation and the linearity of expectation.

Proof: Consider the execution time of *remove* or *find* for a key k. Both need constant time plus the time for scanning the sequence $t[h(k)]$. Hence, even if this sequence

has to be scanned completely, the expected execution time is $O(1+\mathrm{E}[X])$ where the random variable X stands for the length of sequence $t[h(k)]$. Let e_i denote element i in the table. Let $X_1, \ldots, X_n$ denote *indicator* random variables where $X_i = 1$ if $h(key(e_i)) = h(k)$ and otherwise $X_i = 0$. We have $X = \sum_{i=1}^{n} X_i$. Using the linearity of expectation, we get

$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \mathrm{prob}(X_i = 1) \; .$$

A random hash function will map e_i to all m table entries with the same probability, independent of $h(k)$. Hence, $\mathrm{prob}(X_i = 1) = 1/m$. We get $\mathrm{E}[X] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$ and overall an expected execution time of $O(n/m+1)$. ∎

We can achieve linear space requirements and constant expected execution time of all three operations if we keep $m = \Theta(n)$ at all times. This can be achieved using adaptive reallocation analogous to the unbounded arrays described in Section 1.1 so that we ask you to work out the details yourself in Exercise 2.6

Exercises

Exercise 2.5 (Implementation details) Implement hashing with chaining in your favorite programming language. The following representation of table entries seems to be good: For the table entries use singly linked list where the first element of each list is stored in the table itself. The last element of each sequence points to a dummy element that is used as a sentinel for scanning the list. All lists share the same dummy item.

Exercise 2.6 (Unbounded Hash Tables) Explain how to implement hashing with chaining in such a way that $m = \Theta(n)$ at all times. Assume that there is a hash function $h' : Key \to \mathbb{N}$ and that you set $h(k) = h'(k) \bmod m$.

Exercise 2.7 (Waste of space) Waste of space in hashing with chaining is due to empty table entries. Assuming a random hash function, compute the expected number of empty table entries as a function of m and n. Hint: Define indicator random variables $Y_0, \ldots, Y_{m-1}$ where $Y_j = 1$ if $t[i]$ is empty.

2.2 Universal Hash Functions

The performance guarantees for hashing from Theorem 2.1 have a serious flaw. It is very expensive to get hold of a truly random hash function $h : Key \to 0..m-1$. We would have to compute a table of random values with $|Key|$ entries. This is prohibitive; think of $Key = 0..2^{32}-1$ or even strings. On the other hand, we have seen that we cannot use a single fixed hash function. Some randomness is necessary if we want to cope with all possible inputs. The golden middle way is to choose a hash function randomly from some smaller set of functions. For the purposes of this chapter we only need the following simple condition:

Definition 2.2 A family $\mathcal{H} \subseteq \{0..m-1\}^{Key}$ of functions from keys to table entries is *c-universal* if for all x, y in *Key* with $x \neq y$ and random $h \in \mathcal{H}$,

$$\mathrm{prob}(h(x) = h(y)) \leq \frac{c}{m} \quad .$$

For c-universal families of hash functions, we can easily generalize the proof of Theorem 2.1 for fully random hash functions.

Theorem 2.3 *If n elements are stored in a hash table with m entries using hashing with chaining, the expected execution time of remove or find is $O(1+cn/m)$ if we assume a hash function taken randomly from a c-universal family.*

Now it remains to find c-universal families of hash functions that are easy to compute. We explain a simple and quite practical 1-universal family in detail. and give further examples in the exercises. In particular, Exercise 2.9 gives an family that is perhaps even easier to understand since it uses only simple bit operations. Exercise 2.11 introduces a simple family that is fast for small keys and gives an example where we have c-universality only for $c > 1$.

Assume that the table size m is a prime numbers. Set $w = \lfloor \log m \rfloor$ and write keys as bit strings. Subdivide this bit string into pieces of at most w bits each, i.e., view keys as k-tuples of integers between 0 and $2^w - 1$. For $\mathbf{a} = (a_1, \ldots, a_k)$ define

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m$$

where $\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$ denotes scalar product.

Theorem 2.4

$$H^{\cdot} = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k \right\}$$

is a 1-universal family of hash functions if m is prime.

In other words, we get a good hash function if we compute the scalar product between a tuple representation of a key and a random vector.

Proof: Consider two distinct keys $\mathbf{x}$ and $\mathbf{y}$. To determine $\mathrm{prob}(h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y}))$, we count the number of choices for $\mathbf{a}$ such that $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.

Fix an index j such that $x_j \neq y_j$. We claim that for each choice of the a_i's with $i \neq j$ there is exacly one choice of a_j such that $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$. To show this, we solve the equation for a_j:

$$\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \Leftrightarrow \sum_{1\leq i\leq k} a_i x_i &\equiv \sum_{1\leq i\leq k} a_i y_i (\bmod m) \\
\Leftrightarrow a_j(y_j - x_j) &\equiv \sum_{i\neq j, 1\leq i\leq k} a_i(x_i - y_i)(\bmod m) \\
\Leftrightarrow a_j &\equiv (y_j - x_j)^{-1} \sum_{i\neq j, 1\leq i\leq k} a_i(x_i - y_i)(\bmod m)
\end{aligned}$$

where $(x_j - y_j)^{-1}$ denotes the multiplicative inverse of $(x_j - y_j)$. This value is unique mod m and is guaranteed to exist because we have choosen j such that $x_j - y_j \not\equiv 0(\bmod m)$ and because the integers modulo a prime number form a field.

There are m^{k-1} ways to choose the a_i with $i \neq j$. Since the total number of possible choices for $\mathbf{a}$ is m^k, we get

$$\mathrm{prob}(h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})) = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$

∎

Is it a serious restriction that we need prime table sizes? On the first glance, yes. We cannot burden users with the requirement to specify prime numbers for m. When we adaptively grow or shrink an array, it is also not obvious how to get prime numbers for the new value of m. But there is a simple way out. Number theory tells us that there is a prime number just around the corner [4]. On the average it suffices to add an extra $O(\log m)$ entries to the table to get a prime m. Furthermore, we can afford to look for m using a simple brute force approach: Check the desired m for primality by trying to divide by all integers in $2..\lfloor\sqrt{m}\rfloor$. If a divisor is found, increment m. Repeat until a prime number is found. This takes time $O(\sqrt{m}\log m)$ — much less than the time needed to initialize the table with $\perp$ or to move elements at a reallocation.

Exercises

Exercise 2.8 (Strings as keys.) Implement the universal family $H^{\cdot}$ for strings of 8 bit characters. You can assume that the table size is at least $m = 257$. The time for evaluating a hash function should be proportional to the length of the string currently processed. Input strings may have arbitrary lengths not known in advance. Hint: compute the random vector $\mathbf{a}$ lazily, extending it only when needed.

Exercise 2.9 (Hashing using bit matrix multiplication.)$\otimes$For table size $m = 2^w$ and $Key = \{0,1\}^k$ consider the family of hash functions

$$H^{\oplus} = \left\{ h_M : M \in \{0,1\}^{w \times k} \right\}$$

where $h_M(\mathbf{x}) = Mx$ computes a matrix product using arithmetics mod 2. The resulting bit-vector is interpreted as an integer from $0..2^w - 1$. Note that multiplication mod 2 is the logical and operation and that addition mod 2 is the logical exclusive-or operation $\oplus$.

a) Explain how $h_M(\mathbf{x})$ can be evaluated using k bit-parallel exclusive-or operations.

b) Explain how $h_M(\mathbf{x})$ can be evaluated using w bit-parallel *and* operations and w *parity* operations. Many machines support a machine instruction $parity(\mathbf{y})$ that is one if the number of one bits in y is odd and zero otherwise.

c) We now want to show that $H^{\oplus}$ is 1-universal. As a first step show that for any two keys $x \neq y$, any bit position j where x and y differ, and any choice of the columns M_i of the matrix with $i \neq j$, there is exactly one choice of column M_j such that $h_M(\mathbf{x}) = h_M(\mathbf{y})$.

d) Count the number of ways to choose $k-1$ columns of M.

e) Count the total number of ways to choose M.

f) Compute the probability $\text{prob}(h_M(\mathbf{x}) = h_M(\mathbf{y}))$ for $x \neq y$ if M is choosen randomly.

***Exercise 2.10 (More matrix multiplication.)** Define a class of hash functions

$$H^{\times} = \left\{ h_M : M \in \{0..p\}^{w \times k} \right\}$$

$H^{\times}$ that generalizes class $H^{\oplus}$ by using arithmetics mod p for some prime number p. Show that $H^{\times}$ is 1-universal. Explain how $H^{\cdot}$ is also a special case of $H^{\times}$.

Exercise 2.11 (Simple linear hash functions.) Assume $Key = 0..p-1$ for some prime number p. Show that the following family of hash functions is $(\lceil N/m \rceil / (N/m))^2$-universal.

$$H^{*} = \left\{ h_{(a,b)} : a,b \in 0..p-1 \right\}$$

where $h_{(a,b)}(x) = ax + b \bmod p \bmod m$.

Exercise 2.12 (A counterexample.) Consider the set of hash functions

$$H^{\text{fool}} = \left\{ h_{(a,b)} : a,b \in 0..p-1 \right\}$$

with $h_{(a,b)}(x) = ax + b \bmod m$. Show that there is a set of $\lfloor N/m \rfloor$ keys M such that

$$\forall x,y \in M : \forall h_{(a,b)} \in H^{\text{fool}} : h_{(a,b)}(x) = h_{(a,b)}(y)$$

even if m is prime.

Exercise 2.13 (Table lookup made practical.) Let $m = 2^w - 1$ and view keys as k-tuples of a-bit numbers for some small constant a. The idea is to replace the single lookup in a huge table from Section 2.1 by k lookups in smaller tables of size 2^a. Show that

$$H^{[]} = \left\{ h_{(t_1,\dots,t_k)} : t_i \in \{0..m-1\} \right\}^{\{0..w-1\}}$$

where

$$h_{(t_1,\dots,t_k)}((x_1,\dots,x_k)) = \bigoplus_{i=1}^{k} t_i[x_i] \enspace .$$

is 1-universal.

2.3 Hashing with Linear Probing

Hashing with chaining is categorized as a *closed* hashing approach because each entry of the table $t[i]$ has to cope with all the elements with $h(key(e)) = i$. In contrast, *open* hashing schemes open up other table entries to take the overflow from overloaded fellow entries. This added flexibility allows us to do away with secondary data structures like linked lists—all elements are stored directly in table entries. Many ways for organizing open hashing have been investigated. We only explore the simplest scheme that has been known since the early 50s [6]$\otimes$and has turned out to be the fastest on modern check machines. Figure **??** gives pseudo-code for this data structure.

Unused entries are filled with a special element $\perp$. An element e is stored in entry $t[h(key(e))]$ or further to the right. But we only go away from index $h(key(e))$ with good reason, i.e., only if the table entries between $t[h(key(e))]$ and e are occupied by other elements. The remaining implementation basically follows from this invariant. Figure **??** gives pseudo code.

To insert an element, we linearly scan the table starting at $t[h(key(e))]$ until a free entry is found where d is then stored. Similarly, to find an element with key k, we scan the table starting at $t[h(k)]$ until a matching element is found. The search can be aborted when an empty table entry is encountered because any matching element

```
// Hash table with m primary entries and an overflow area of size m'.
Class BoundedLinearProbing(m, m' : ℕ; h : Key → 0..m − 1)
    t = [⊥, . . . ,⊥] : Array [0..m + m' − 1] of Element
    invariant ∀i : t[i] ≠ ⊥ ⇒ ∀j ∈ {h(key(t[i]))..i − 1} : t[i] ≠ ⊥

    Procedure insert(e : Element)
        for i ∈ h(key(e)).. while t[i] ≠ ⊥ do ;
        assert i < n + m − 1                                  // no overflow
        t[i] := e

    Function find(k : Key): Element
        for i ∈ h(key(e)).. while t[i] ≠ ⊥ do
            if key(t[i]) = k then return t[i]
        return ⊥                                              // not found

    Procedure remove(k : Key)
        for i ∈ h(key(e)).. while key ≠ t[i] do
            if t[i] = ⊥ then return                           // nothing to do
        // Scan a cluster of elements.
        // i is where we currently plan for a ⊥ (a hole).
        for j ∈ i + 1.. while t[j] ≠ ⊥ do
            // Establish invariant for t[j].
            if h(key(t[j]) ≤ i then
                t[i] := t[j]                        // Overwrite removed element
                i := j                                    // move planned hole
        t[i] := ⊥                                     // erase removed element
```

Figure 2.1: Hashing with linear probing.

further to the right would violate the invariant. So far this sounds easy enough, but we have to deal with two complications.

What happens if we reach the end of the table during insertion? We choose a very simple fix by allocating m' table entries to the right of the largest index produced by the hash function h. For 'benign' hash functions it should be sufficient to choose m' much smaller than m in order to avoid table overflows. Exercise 2.14 asks you to develop a more robust albeit slightly slower variant where the table is treated as a cyclic array.

A more serious question is how *remove* should be implemented. After finding the element we might be tempted to it with $\perp$. But this could violate the invariant for elements further to the right. Consider the example

$$t = [\ldots, \underset{h(z)}{x}, y, z, \ldots] \ .$$

When we naively *remove* element y and subsequently try to find z we would stumble over the hole left by removing y and think that z is not present. Therefore, most other variants of linear probing disallow *remove* or mark removed elements so that a subsequent *find* will not stop there. The problem with marking is that the number of nonempty cells (occupied or marked) keeps increasing so that searches eventually become very slow. This can only be mitigated by introduducing the additional complication of periodic reorganizations of the table.

Here we give a different solutions that is faster and needs no reorganizations [6, Algorithm R The idea is rather obvious—when an invariant may be violated, fix check it. Let i denote the index of the deleted element. We scan entries $t[j]$ to the right of i to check for violations of the invariant. If $h(key(t[j])) > i$ the invariant still holds even if $t[i]$ is emptied. If $h(key(t[k])) \leq i$ we can move $t[j]$ to $t[i]$ without violating the invariant for the moved element. Now we can pretend that we want to remove the duplicate copy at $t[j]$ instead of $t[i]$, i.e., we set $i := j$ and continue scanning. Figure **??** depicts this case. We can stop scanning, when $t[j] = \perp$ because elements to the right that violate the invariant would have violated it even before the deletion.

Exercises

Exercise 2.14 (Cyclic linear probing.) Implement a variant of linear probing where the table size is m rather than $m + m'$. To avoid overflow at the right end of the array make probing wrap around.

- Adapt *insert* and *delete* by replacing increments with $i := i + 1 \bmod m$.
- Specify a predicate $between(i, j, k)$ that is true if and only if j is cyclically between i and j.

- Reformulate the invariant using *between*.
- Adapt *remove*.

Exercise 2.15 (Unbounded linear probing.) Implement unbounded hash tables using linear probing and universal hash functions. Pick a new random hash function whenever the table is reallocated. Let $1 < \gamma < \beta < \alpha$ denote constants we are free to choose. Keep track of the number of stored elements n. Grow the table to $m = \beta n$ if $n > m/\gamma$. Shrink the table to $m = \beta n$ if $n < m/\alpha$. If you do not use cyclic probing as in Exercise 2.14, set $m' = \delta m$ for some $\delta < 1$ and reallocate the table if the right end should overflow.

2.4 Chaining Versus Linear Probing

We have seen two different approaches to hash tables, chaining and linear probing. Which one is better? We do not delve into details of a theoretical analysis since this is complicated for the case of linear probing and because the results would remain inconclusive. Therefore, we only discuss some qualitative issues without detailed proof and then proceed to a few experimental findings.

An advantage of linear probing is that in each table access, a contiguous piece of memory is accessed. The memory subsystems of modern processors are optimized for this kind of access patterns whereas they are quite bad at chasing pointers when the data does not fit in chache memory. Therefore we expect the outcome of the comparison to depend on the table size. A disadvantage of linear probing is that search times become very high when the number of elements approaches the table size. For chaining, the expected acces time remains very small. On the other hand, chaining wastes space for pointers that can be used to support a larger table in linear probing. Hence, the outcome is not so clear.

Messungen diskutieren. sizeof(Element)=8, 32 bit keys, n insertions, 3n successful finds, n unsuccessful finds, n deletions, plot T[chaining]/T[probing] over s=1.5..4 C++, compiler, machine, $n = 2^8, 2^16, 2^24$, warmup, access only erwaehnen? larger elements erwaehnen?

diskutiere weitere tradeoffs, quality of performance guarantees, referential integrity, space consumption,(teilweise in impl. notes?)

Exercises

Exercise 2.16 Repeat the measurements on your own machine using your favorite programming language. Also compare with hash table implementations from software libraries.

Exercise 2.17 (Large elements.) Repeat the measurements with element sizes 32 and 128. Also add an implementation of *slim chaining* where table entries only store pointers to the first list element (see also Section 2.5 below).

Exercise 2.18 (Large keys) Discuss the impact of large keys on the relative merits of chaining versus linear probing. Which variant will profit? Why?

Exercise 2.19 Implement a hash table data type for very large tables stored in a file. Should you use chaining or linear probing? Why?

2.5 Implementation Notes

The empty table entry $\perp$ can be implemented as an element with $key(\perp)$ different from all 'proper' key values. A general purpose implementation should allow this value to be specified by the application. In some implementations we could also implement $\perp$ as a null pointer. Which solutions is more elegant is a matter of taste. We prefer the first solution since it allows the more efficient implementations by linear probing.

practical hash functions evaluations using SIMD instructions? todo

Hashing with chaining uses only very specialized operations on sequences for which singly linked lists are ideally suited. Since these lists are extremely short, some deviations from the implementation scheme from Section 1.2 are in order. In particular, it would be wasteful to store a dummy item with each list. However, we can use a single shared dummy item to mark the end of the list. This item can then be used as a sentinel element for *find* and *remove* as in function *findNext* in Section 1.2.1. This trick not only saves space but also makes it likely that the dummy item is alway in the cache memory.

Comparing the space consumption of hashing with chaining and linear probing is even more subtle than outlined in Section 2.4. On the one hand, the linked lists burden the memory management with many small pieces of allocated memory, in particular if memory management for list items is not not implemented carefully as discussed in Section 1.2.1. On the other hand, some chaining implementations of unbounded hash tables can avoid occupying two tables during reallocation: First concatenate all lists to a single list ℓ. Deallocate the old table. Only then allocate the new table. Finally, scan ℓ moving the elements to the new table.

C++

There are two basic ways to implement hashing with chaining that both make sense. Entries of *slim* tables are pointers to singly linked lists of elements. *Fat* tables store the first list element in the table itself. Fat tables are usually faster and more space efficient. Slim tables may have advantages if the elements are very large. Slim tables have the additional advantage that references to table entries remain valid when tables are reallocated. We have already observed this complication for unbounded arrays in Section 1.5.

$\otimes$class names

Java

class names

Exercises

***Exercise 2.20 (Associative arrays.)** Implement a C++-class for associative arrays. Support `operator[]` for any index type that supports a hash function. Make sure that `H[x]=...` works as expected if x is the key of a new element.

2.6 Further Findings

more hash families, k-way independence, madh's two stage family? families with a lot of independence, random polynomials

hash function studies

detailed average case studies in Knuth and where else

Rasmus fragen

cuckoo hashing

perfect hashing, shifting trick

other useful universal classes

warn of other open hashing schemes

extendible hashing

fast primality checks

deterministic upper bound on distance to next prime? May be $\Omega\left(\log^2 n\right)$. Conjecture: but no more.

strongly universal hashing

Bibliography

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[2] M. Brown and R. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[4] J. F. Grantham and C. Pomerance. Prime numbers. In K. H. Rosen, editor, *Handbook of Discrete and Combinatorial Mathematics*, chapter 4.4, pages 236–254. CRC Press, 2000.

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

[6] D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[7] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[8] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[9] R. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[10] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.