Contents

0.1 Hashing

Hashing is one of the two main approaches to the dictionary problems, the other being search trees (see Chapter **??**). It allows one to associate information with keys. The keys are drawn from some universe U and we use S for the set of keys stored. Hashing supports the following three operations:

insert(*key* x, *inf* i);	add x to S and associate information i with it.
access(*key* x);	read the information associated with x.
change_inf(*key* x, *inf* i);	change the information associated with x to i.
delete(*key* x);	delete x from S.

The information associated with the elements of S is irrelevant for our theoretical discussion of hashing and hence we will suppress it in our theoretical discussion. We will also suppress the distinction between read- and write-access to the information associated with a key and hence will only distinguish three operations: *insert*, *access*, and *delete*.

Hashing is very efficient. All three operations have constant expected running time. The space complexity of hashing is linear.

For most part of the chapter we assume that $U = [0 .. N-1]$ for some large integer N. The important case of strings is discussed in Section 0.1.6. We use n to denote the cardinality of S. The ingredients of hashing are simple: an array $T[0 .. m-1]$, the *hash table*, and a function $h : U \mapsto [0 .. m-1]$, the *hash function*. The hash function partitions S into blocks which are (hopefully) so small that they can be handled by an elementary method, namely linear search. There are two ways for organizing this linear search; hashing with chaining uses a separate linear list for each block and open addressing uses a second hash function to define a sequence of hash values. We discuss the two methods in Sections 0.1.2 and 0.1.3, respectively.

It is clear that the efficiency of hashing depends heavily on the choice of the hash function. A good hash function is easy to evaluate and distributes S evenly over the table. We will see that good hash functions are abundant. We will obtain this result in three settings: (1) For random subsets S of the universe, (2) for fixed subsets and random choice of hash function (universal hashing, see Section 0.1.4), and (3) for fixed subsets and judicious choice of hash function (perfect hashing, see Section 0.1.8).

The size of the hash table influences the behavior of hashing in two ways. The larger the hash table, the larger the space consumption and the smaller the time complexity. The choice of the size of the hash table and its adjustment in a dynamic environment is discussed in Section 0.1.5).

Our theoretical discussion of hashing describes a space of possible solutions. Implementation issues are discussed in Section 0.1.6.

We complement our algorithmic discussion of hashing with the definition of an abstract data type *hashing arrays* in Section 3.

The use of hashing in secondary memory is discussed in Section 0.1.9.

0.1.1 Applications

The applications of hashing are numerous. We give some examples.

Dictionaries: We want to store German words together with their English translation. In this application the keys are strings. The spell-checker in your editor probably uses hashing to store the set words known to it. Your favorite compiler probably uses hashing for the symbol table. A symbol table stores all identifiers used in a program.

Spare Arrays: Consider the following situation. We are given a directed graph $G = (V, E)$ and want to check whether the graph is symmetric, i.e., whether for any edge (v, w) we also have the reverse edge (w, v). In a graph G with n nodes there are n^2 potential edges.

Assuming that the nodes are number from 0 to $n-1$, there is a trivial solution using a two-dimensional array A. We initialize the array to false and then iterate over all edges of G. For an edge (v, w), we set $A[v, w]$ to true. We then iterate a second time over the edges. For an edge (v, w), we check whether the entry $A[w, v]$ corresponding to the reversed edge is true. If this is not the case for some edge, the graph is not symmetric. The running time of this approach is $O(n^2 + m)$, where m is the number of edges of G. The $O(n^2)$-term results from the fact that we need to initialize the array.

Hashing reduces the running time to $O(m)$. We use a hash table for the set of edges. We interpret an edge (v, w) as an integer $v \cdot n + w$ and have the universe $U = [0 \mathinner{..} n^2-1]$. Again we iterate over the edges of G and insert them into the hash table. Then we iterate a second time and for each edge check whether the reversed edge is present. The running time reduces to $O(m)$, since the hash table does not require any time for initialization.

PRAM-Simulation: A PRAM (parallel random access machine) consists of a number n of (sequential processors) and a shared memory of size N. The distinctive feature

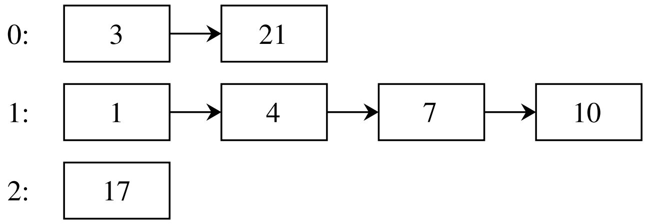

Figure 1: Hashing with Chaining: The table has size $m = 3$ and the hash function $h(x) = x \bmod 3$ is used. The set stored is $S = \{1, 3, 4, 7, 10, 17, 21\}$.

of PRAMs is the fact that any memory cell can be accessed from any processor in constant time. The PRAM offers a very convenient programming model as it frees its programmers from thinking about storage allocation. Unfortunately, PRAMs are hard to build as there is no known technology that supports parallel access by a large number of processors (say more than 100) to a large shared memory (say several Gigabytes).

Realistic parallel computers consist of processors, memory modules and an interconnection network that routes requests from processors to memory modules. Hashing can be used to simulate PRAMs on realistic parallel computers. The idea is to use a hash function for load balancing: A memory cell with address x is stored in memory module $h(x)$. The hope is that the hash function will distribute the requests evenly over the memory modules. Theorem 4 indicates that this is indeed the case. We refer the reader to [MV84, Ran91] for more details. The simulation of PRAMs on networked parallel computers by hashing is efficient enough to be the basis for the SB-PRAM project [ADK+93, AKP99]. A prototype with 64 processors is running, see http://www-wjp.cs.uni-sb.de/projects/sbpram/.

0.1.2 Hashing with Chaining

Hashing with chaining uses a linear list for each entry of the hash table. The list i-th list stores all elements $x \in S$ with $h(x) = i$. Figure 1 shows an example.

The following simple algorithm realizes an access operation *Access*(x, S):

1) compute $h(x)$;
2) search for x in the list $T[h(x)]$.

Operations *Insert*(x, S) and *Delete*(x, S) are implemented similarly. We only have to add x to or to delete x from list $T[h(x)]$.

We come to the analysis of hashing. The time complexity of all three operations is the time for evaluating the hash function h plus the time for searching through the list $T[h(x)]$. In this section we assume that h can be evaluated in constant time and therefore define the cost of an operation referring to a key x as $O(1 + \delta_h(x, S))$ where $\delta_h(x, S)$ measures the number of *collisions* between x and elements of S. We have

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$$

and

$$\delta_h(x, y) = \begin{cases} 1 & \text{if } h(x) = h(y) \text{ and } x \neq y; \\ 0 & \text{otherwise.} \end{cases}$$

In order to simplify notation, we will drop the big-O and use $1 + \delta_h(x, S)$ for the cost of an operation.

The worst case complexity of hashing is easily determined. The worst case occurs when h restricted to S is constant, i.e., $h(x) = i_0$ for some i_0 and all $x \in S$. Then hashing deteriorates to searching in a linear list; any one of the three operations costs $\Theta(|S|)$ time units.

Theorem 1 *The time complexity (in the worst case) of operations Access(x, S), Insert(x, S) and Delete(x, S) is $\Theta(|S|)$.*

Average case behavior is much better; in fact, it is excellent. We analyze it under the following assumptions (we will discuss their appropriateness later):

(A1) The hash function $h : U \mapsto [0 \mathinner{..} m-1]$ is *fair* and distributes the universe nearly uniformly over the interval $[0 \mathinner{..} m-1]$, i.e., $|h^{-1}(i)| \le \lceil N/m \rceil$ for all $i \in [0 \mathinner{..} m-1]$. We use (A1*) to denote the assumption that $|h^{-1}(i)| = N/m$ for all i.

(A2) The set stored is a random subset of U of size n, i.e., S is any of the $\binom{N}{n}$ subsets of U of size n with equal probability.

Theorem 2 *Under the assumptions (A1) and (A2), we have*

$$\mathrm{E}[\delta_h(x, S)] \le \frac{n}{m}$$

and hence the expected time complexity of insert, delete, and access is $O(1 + \beta)$, where $\beta = n/m$.

Proof: By the definition of expectations, we have

$$\mathrm{E}[\delta_h(x, S)] = \frac{1}{\binom{N}{n}} \sum_S \delta_h(x, S) \ ,$$

where the summation is over all subsets of U of size n. Let $U' = \{y \in U \setminus x \ ; h(y) = h(x)\}$ be the part of the universe that is mapped to the same location as x. Then $N' = |U'| \le \lceil N/m \rceil - 1 \le N/m$ by our assumption of fairness (the -1 stems from the fact that x does not belong to U') and $\delta_h(x, S)$ is simply the number of elements of S that lie in U'. We determine this number by defining for each $y \in U$ an indicator variable Δ_y which indicates whether y belongs to S.

$$\Delta_y(S) = \begin{cases} 1 & \text{if } y \in S \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$\delta_h(x, S) = \sum_{y \in U'} \Delta_y(S)$$

and hence

$$\mathrm{E}[\delta_h(x, S)] = \sum_{y \in U'} \mathrm{E}[\Delta_y(S)] = \sum_{y \in U'} \mathrm{prob}(y \in S) \ .$$

Consider a fixed $y \in U'$. There are $\binom{N}{n}$ subsets of U of size n and there are $\binom{N-1}{n-1}$ subsets that contain y (the remaining $n-1$ elements form an arbitrary subset of $U \setminus y$). Thus

$$\text{prob}(y \in S) = \frac{\binom{N-1}{n-1}}{\binom{N}{n}} = \frac{n}{N}$$

and hence

$$\sum_{y \in U'} \text{prob}(y \in S) = \sum_{y \in U'} \frac{n}{N} \leq \frac{N}{m} \cdot \frac{n}{N} = \frac{n}{m} \, .$$

∎

The quantity $\beta = n/m$ will reoccur in the analysis of hashing schemes. It is called *load factor*.

We will next discuss the probability assumptions used in the analysis of hashing. The first assumption is easily satisfied. An example is the *division method*. We use $h(x) = x \bmod m$, where $U = [0 \mathinner{..} N-1]$ and m divides $N = |U|$. If m does not divide N but N is much larger than m, the division method almost satisfies the first assumption and Theorem 2 remains essentially true.

The second assumption is more critical because it makes an assumption on the use of the hashing scheme: all subsets of U of size n are equally likely. The authors can think of no application of hashing where this assumption holds. There is always some bias in the data. Let us look at some of the examples of Section 0.1.1. In a spell-checker for English, most words looked up will be genuine English words. In a symbol table, the occurrence of an identifier *i2* makes it likely that *i1* is also present.

Does the fact that the second assumption underlying our analysis is unrealistic, make our analysis meaningless? Yes and no. Yes, because the theorem is never applicable in a mathematical sense. No, because the theorem gives us a feeling for the behavior of hashing. For example, the proof tells us that some bias does not really harm. For example, if the expected number of elements in S mapped to any particular location is cn/m for some small constant c, the expected access time is $O(1 + c \cdot \beta)$. We might even be able to exploit the bias in the data by a specific design of the hash function. We give an example from the symbol table domain.

Symbol tables in compilers are often realized by hash tables. Identifiers are strings over the alphabet $\{A, B, C, \ldots\}$, i.e., $U = \{A, B, C, \ldots\}^*$. The use of identifiers is definitely *not* uniformly distributed over U. Identifiers i1, i2, j1, ... are very popular and XYZ is not. We can use this non-uniformity to obtain even better behavior than predicted by Theorem 2.. Inside a computer identifiers are represented as bit-strings; usually 8 bits (a byte) are used to represent one character. In other words, we assign a number $num(C) \in [0 \mathinner{..} 255]$ to each character of the alphabet and interpret a string $C_r C_{r-1} \ldots C_0$ as a number in base 256, namely $\sum_{i=0}^{r} num(C_i) \cdot 256^i$; moreover, consecutive numbers are usually assigned to characters 1,2,3, ... Then strings i0, i1, i2, and x0, x1, x2 lead to arithmetic progressions of the form $a + i$ and $b + i$ respectively where $i = 0, 1, 2, \ldots$ and 256 divides $(a - b)$. As identifiers of the form i0, i1, i2, and

load factor	0.5	0.6	0.7	0.8	0.9
experiment	1.19	1.25	1.28	1.34	1.38
theory	1.25	1.30	1.35	1.40	1.45

Table 1: The average case complexity of hashing with chaining. The table shows the average cost of a search as predicted by Theorem 2 and as measured in a experiment ([LYD71]). The keys stored were identifiers and the hash function was designed to avoid collisions between frequently occurring keys.

x0, x1, x2 are used so frequently[1], we want

$$h(a+i) \neq h(b+j) \qquad \text{for } 0 \leq i, j \leq 9 \text{ and } 256|(a-b)\ .$$

We next specialize to a hash function of the form $h(x) = x \bmod m$ for some m. We want

$$(b-a) + (j-i) \neq 0 \bmod m.$$

Since $256|(b-a)$ we should choose m such that m does not divide numbers of the form $256 \cdot c + d$ where $|d| \leq 9$. In this way one can obtain even better practical performance than predicted by theory, see Table 1.

We come back to the theoretical analysis of hashing and analyze the expected number of empty buckets and the expected length of the longest chain.

Theorem 3 *Under assumptions (A1*) and (A2), the expected number of empty lists is bounded by $me^{-\beta}$, where $\beta = n/m$ is the load factor.*

Proof: We define for each bucket j an indicator variable Z_j telling us whether any key was mapped to bucket j, i.e., $Z_j = 1$ iff there is an $x \in S$ with $h(x) = j$, and $Z_j = 0$, otherwise. Then $\sum_{0 \leq j < m} Z_j$ is the number of empty buckets.

Let $U_j = \{x \in U\ ; h(x) = j\}$ be the set of elements that are mapped to j and let $N_j = |U_j|$ be their number. Then $Z_j = 1$ iff S is a subset of $U \setminus U_j$. Thus

$$\begin{aligned}
\text{prob}(Z_j = 1) = \frac{\binom{N-N_j}{n}}{\binom{N}{n}} &= \frac{(N-N_j)\cdots(N-N_j-n+1)}{N\cdots(N-n+1)} \\
&\leq \frac{(N-N_j)^n}{N^n} = \big(1 - \frac{N_j}{N}\big)^n \\
&= e^{n\ln(1-N_j/N)} \leq e^{-nN_j/N}
\end{aligned}$$

where the first inequality uses the fact that $(N - N_j - i)/(N - i) \leq (N - N_j)/N$ and the second inequality uses the fact that $\ln(1 + x) \leq x$ for all $x > -1$ (see XXXX).

The expected number of empty lists is therefore bounded by $\sum_{0 \leq j < m} e^{-nN_j/N} = me^{-\beta}$, since $N_j = N/m$ for all j by the fairness assumption (A1*)[2]. ∎

[1] Early programming languages restricted the length of identifiers to six characters. The use of identifiers i0, i1, . . . was more frequent then than it is now. Still, the example serves to make the point that bias in the data is not necessarily bad.

[2] A similar result can be obtained with the fairness assumption (A1). We need to determine the maximum of $\sum_{0 \leq j < m} e^{-nN_j/N}$ subject to the constraints that $\sum_j N_j = N$ and $N_j \leq \lceil N/m \rceil$ for all j. The maximum is attained if all but one N_j has either value $\lceil N/m \rceil$ or value 0. We wanted to spare our readers the tedious details of this calculation.

In Section 0.1.6 we will discuss two implementations of hashing with chaining. In one of the implementations, empty buckets constitute "wasted space". The theorem above will allow us to estimate the space requirement of this implementation fairly precisely.

Theorem 4 *Under assumptions (A1*) and (A2) , the expected maximal number of elements in any bucket is $j_0 + O(1)$, where j_0 is the smallest integer j with $n \cdot (n/m)^{j-1} \cdot (1/j!) \leq 1$.*

For $\beta = n/m \leq 1$, we have $j_0 = O((\log n)/\operatorname{loglog} n)$, and for $n \geq m \log n$, we have $j_0 = O(n/m)$.

Proof: Let l_i be the number of elements of S which are stored in the i-th list. We want to bound the expectation of $\max_i l_i$. We use the formula (see XXX)

$$\mathrm{E}[\max_i l_i] = \sum_{j \geq 1} \mathrm{prob}(\max_i l_i \geq j) ,$$

In order to apply this formula we need a bound for $\mathrm{prob}(\max_i l_i \geq j)$. We start with the observation that for any fixed i we have

$$\mathrm{prob}(l_i \geq j) \leq \binom{n}{j}(1/m)^j ,$$

since the event "$l_i \geq j$" implies the existence of a set of subset S' of S of size j all of whose elements are mapped to j. For any particular element this probability is $1/m$. Next observe that the event "$\max_i l_i \geq j$" implies that there is at least one i with $l_i \geq j$ and hence

$$\mathrm{prob}(\max_i l_i \geq j) \leq m \cdot \binom{n}{j} \cdot (1/m)^j \leq n \cdot (n/m)^{j-1} \cdot (1/j!) .$$

For small j this bound is weak. We also have the trivial bound $\mathrm{prob}(\max_i l_i \geq j) \leq 1$. Combining our bounds we obtain

$$\mathrm{E}[\max_i l_i] \leq \sum_{j \geq 1} \min(1, n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!}) = j_0 + \sum_{j \geq j_0} n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!} .$$

Let $p_j = n \cdot (n/m)^{j-1} \cdot (1/j!)$. Then $p_{j+1}/p_j = \beta/(j+1)$. We now distinguish cases. For $\beta \leq 1$, we have $p_{j+1}/p_j \leq 1/(j+1) \leq 1/2$ and hence $\sum_{j \geq j_0} p_j \leq \sum_{j \geq 0} 2^{-j} = 2$. Also $j_0 \leq L = 2(\log n)/\operatorname{loglog} n$ for large n, since $\log(L/e) \geq (1/2) \operatorname{loglog} n$ for large n and hence (using $L! \geq (L/e)^L$)

$$L! \geq (L/e)^L = 2^{L \log(L/e)} \geq 2^{\log n} = n .$$

For $\beta \geq \log n$, we have $j_0 \leq L = 2e\beta$ since

$$p_L = n \cdot \beta^L \cdot (1/L!) \leq n \cdot \beta^L \cdot (e/L)^L = n(1/2)^L \leq 1 ,$$

and $p_{j+1}/p_j \leq \beta/(j+1) \leq 1/2$ for $j \geq L$. Thus $\sum_{j \geq L} p_j \leq \sum_{j \geq 0} 2^{-j} = 2$. ∎

The expected length of the longest chain is $O(\log n/\operatorname{loglog} n)$ if $\beta \leq 1$ and is $O(\beta)$ if $\beta \geq \log n$. In both cases the probability that the longest chain is much longer is small

0	1	2	3	4	5	6
13	15	9	3	10	17	6

$$\begin{aligned}
h(3, i) &= (3 + 4i) \bmod 7, && i = 0 \text{ works} \\
h(17, i) &= (3 + 2i) \bmod 7, && i = 1 \text{ works} \\
h(6, i) &= (6 + 3i) \bmod 7, && i = 0 \text{ works} \\
h(9, i) &= (2 + 2i) \bmod 7, && i = 0 \text{ works} \\
h(15, i) &= (1 + 4i) \bmod 7, && i = 0 \text{ works} \\
h(13, i) &= (6 + 2i) \bmod 7, && i = 4 \text{ works} \\
h(10, i) &= (3 + 3i) \bmod 7, && i = 5 \text{ works}
\end{aligned}$$

Figure 2: Example for hashing with open addressing: We use $m = 7$, $h_1(x) = x \bmod 7$ and $h_2(x) = 1 + (x \bmod 4)$. We insert 3, 17, 6, 9, 15, 13, and 10 in order and obtain the configuration shown above.

(see exercise 1). The length of the longest chain is clearly $\Omega(\beta)$ and hence our bound has the optimal order for large β. For $\beta \approx 1$, the expected length of the longest chain is $\Omega(\log n / \log\log n)$; a proof can be found in TODO.

We can now give more details on the use of hashing in PRAM simulations. A physical machine with n processors simulates $n \log n$ virtual processors. The simulation proceeds in rounds. In each round, a physical processor executes one instruction of each of the $\log n$ virtual processors assigned to it. This will generate $n \log n$ memory requests. The requests are distributed over the n memory modules by means of hashing. The theorem above tells us that hashing achieves almost perfect load balancing. The $n \log n$ requests are routed through the network (with a suitable network this will take time $O(\log n)$) and the requests are served. In this way, a phase takes logarithmic time.

0.1.3 Hashing with Open Addressing

Hashing with open addressing resolves conflicts within the hash table itself. It tries for each element $x \in U$ a sequence $h(x, i)$, $i = 0, 1, 2, \ldots$ of table positions. An element is placed at the first free position in its probe sequence. There are two popular methods for defining $h(x, i)$:

Linear probing uses a single hash function h and probes the table positions starting at $h(x)$ linearly, i.e., $h(x, i) = (h(x) + i) \bmod m$. It is guaranteed that all positions are probed.

Double hashing uses a two hash functions h_1 and h_2. The first hash function defines the first location to be probed and the second hash function defines the stride of the linear search, i.e., $h(x, i) = (h_1(x) + i \cdot h_2(x)) \bmod m$. Clearly, $h_2(x)$ has to be non-zero. If $h_2(x)$ and m are relatively prime, all table positions are probed (TODO: assemble the necessary facts about finite fields in the foundations chapters). Figure 2 illustrates the method.

Hashing with open addressing does not require any additional space besides the hash table itself. It does not support deletions (at least not in an obvious way) and care must be taken to bound the load factor away from 1.

We analyze the expected cost of an operation *insert*(x) into a table with load factor $\beta = n/m$ under the assumption that the probe sequence $h(x,i)$, $i = 0, 1, \ldots, m-1$ is a random permutation of the set $\{0, \ldots, m-1\}$. This is called *uniform hashing*. As before, we define the cost of the insertion as $1 + \min\{i \; ; T[h(x,i)] \text{ is not occupied}\}$.

Theorem 5 *The expected cost of an insertion into a table with load factor $\beta = n/m$ is $(m+1)/(m-n+1) \approx 1/(1-\beta)$ provided that $\beta < 1$.*

Proof: Let $C(n,m)$ be the expected cost of an insertion into a table with m positions, n of which are occupied. Let $q_j(n,m)$ be the probability that table positions $h(x,0)$, $\ldots$,$h(x,j-1)$ are occupied. Then $q_0(n,m) = 1$ and

$$q_j(n,m) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-(j-1)}{m-(j-1)} = \frac{n}{m} \cdot q_{j-1}(n-1, m-1)$$

for $j \geq 1$ since position $h(x,0)$ is occupied in n out of m cases and, if $h(x,0)$ is occupied then position $h(x,1)$ is occupied in $n-1$ out of $m-1$ cases, $\ldots$. Thus

$$\begin{aligned} C(n,m) &= \sum_{j=0}^{n} (q_j(n,m) - q_{j+1}(n,m)) \cdot (j+1) \\ &= \sum_{j=0}^{n} q_j(n,m) - (n+1) \cdot q_{n+1}(n,m) \\ &= \sum_{j=0}^{n} q_j(n,m) \; . \end{aligned}$$

We prove $C(n,m) = (m+1)/(m-n+1)$ by induction on m and for fixed m by induction on n. For $n = 0$ and any m we have $C(0,m) = q_0(0,m) = 1$. For $n \geq 1$, we have

$$\begin{aligned} C(n,m) &= \sum_{j=0}^{n} q_j(n,m) = 1 + \frac{n}{m} \cdot \sum_{j=0}^{n-1} q_j(n-1, m-1) = 1 + \frac{n}{m} \cdot C(n-1, m-1) \\ &= 1 + \frac{n}{m} \cdot \frac{m}{m-n+1} = \frac{m+1}{m-n+1} \; . \end{aligned}$$

∎

The expected cost of a successful access operation is now easy to determine. Let S be the set stored and assume we access a random $x \in S$. In a search for x we step through $h(x,0), h(x,1), \ldots$ until we find an i with $T[h(x,i)] = x$. Of course, the very same i determined the cost of inserting x in the first place. Hence the expected cost of a successful search in a table with n elements is

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} C(i,m) &= \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m-i+1} = \frac{m+1}{n} \left[\sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m-n+1} \frac{1}{j} \right] \\ &\approx \frac{1}{\beta} \ln \frac{m+1}{m-n+1} \qquad \text{(see. TODO)} \\ &\approx \frac{1}{\beta} \ln \frac{1}{1-\beta} \; . \end{aligned}$$

β	0.5	0.7	0.9	0.95	0.99	0.999
$1/(1-\beta)$	2	3.3	10	20	100	1000
$(1/\beta)\cdot\ln(1/(1-\beta))$	1.38	1.70	2.55	3.15	4.65	6.9

Table 2: The expected behavior of hashing with open addressing as a function of the load factor. The middle row shows the expected insertion cost and the last row shows the expected access cost.

Theorem 6 *The expected cost of a successful search in a table with load factor $\beta = n/m$ is $O((1/\beta)\cdot\ln(1/(1-\beta)))$.*

Proof: By the discussion above. ∎

This completes our analysis of open addressing. Theorems 5 and 6 show that open addressing works fine as long as β is bounded below one, say $\beta \le 0.9$, see Table 2. The analysis of linear probing and double hashing is beyond the scope of this book. As long as the load factor is kept small, the differences to the situation analysed here is small. The analysis of linear probing can be found in [SF96, paragraph 8.6] or in [Knu81, TODO].

0.1.4 Universal Hashing

In this section we get to know simple classes of good hash functions. We also learn how to free ourselves from the dependency on user behavior. We saw in Section 0.1.2 that hashing provides us with $O(1)$ expected access time. The average was taken over all subsets of the universe of size n. The worst case behavior is $\Theta(n)$. We are at the mercy of the user that he or she does not select a worst case set.

Universal hashing is a way out of this dilemma. We work with an entire class H of hash functions instead of a single hash function; the specific hash function in use is selected randomly from the collection H. If H is chosen properly, i.e., for every subset $S \subseteq U$ almost all $h \in H$ distribute S fairly evenly over the hash table, then this will lead to small expected access time for every set S. Note that the average is now taken over the functions in class H, i.e., the randomization is done by the algorithm itself not by the user: the algorithm controls the dices.

Let us reconsider the symbol table example. At the beginning of each compiler run the compiler chooses a random element $h \in H$. It will use h for the next compilation. In this way the time needed to compile any fixed program will vary over different runs of the compiler, but the time spent on manipulating the symbol table will have small mean.

What properties should the collection H of hash functions have? For any pair $x, y \in U$, $x \ne y$, a random element $h \in H$ should lead to a collision with small probability. The following definition captures this intuition.

Definition 1 *Let $c \in \mathbb{R}$, $N, m \in \mathbb{N}$. A collection H of functions from $[0\,..\,N-1]$ to $[0\,..\,m-1]$ is c-universal, if for all $x, y \in [0\,..\,N-1]$, $x \ne y$*

$$|\{h\,;\, h \in H \text{ und } h(x) = h(y)\}| \le c \cdot |H|/m \ ,$$

i.e., if only a fraction c/m of the functions in H lead to collision for any pair of distinct keys.

Universal classes guarantee $O(1 + c\beta)$ expected access, insertion and deletion time as will be shown in Theorem 9. The class of all functions from $[0 .. N-1]$ to $[0 .. m-1]$ is 1-universal, see Exercise 2. However, this class is much too large. Since there are m^N functions from $[0 .. N-1]$ to $[0 .. m-1]$, it requires $N \log m - O(1)$ bits to specify a random function. This is prohibitive; think of $N = 2^{30}$ and $m = 2^{15}$. We exhibit much smaller classes in Theorems 7 and **??**; the classes have c close to 1 (in fact equal to 1 for the second class). In Exercise 3 it is shown that there are no c-universal classes for $c < 1 - m/N$.

Theorem 7 *Let $m, N \in \mathbb{N}$ and let N be a prime. Then*

$$H_1 = \left\{h_{a,b}\ ;\ h_{a,b}(x) = [(ax+b) \bmod N] \bmod m,\ a, b \in [0 .. N-1]\right\}$$

is a c-universal class for $c = \left(\lceil N/m\rceil/(N/m)\right)^2$.

Proof: Note first that $|H_1| = N^2$. Let $x, y \in [0 .. N-1]$, $x \neq y$. We have to count the number of pairs (a, b) with $h_{a,b}(x) = h_{a,b}(y)$. If $h_{a,b}(x) = h_{a,b}(y)$, there are $q \in [0 .. m-1]$ and $r, s \in [0 .. \lceil N/m\rceil - 1]$ such that

$$\begin{aligned} ax + b &= q + r \cdot m \bmod N \\ ay + b &= q + s \cdot m \bmod N\ . \end{aligned}$$

Since $\mathbb{Z}_N$ is a field (N is a prime) there is exactly one solution in a, b of these equations for each choice of q, r and s. We conclude that the number of functions leading to a collision is bounded by

$$m \cdot \lceil N/m\rceil^2 = \left(\frac{\lceil N/m\rceil}{N/m}\right)^2 \cdot \frac{N^2}{m}$$

and therefore H_1 is c-universal class for the $c = \left(\lceil N/m\rceil/(N/m)\right)^2$.

∎

We give a second example of a universal class. We assume that m is prime and that $U = [0 .. m-1]^r$ for some r. In this situation we may view an element $x \in U$ as an r-tuple $x = (x_{r-1}, \ldots, x_0)$ of elements in $[0 .. m-1]$. For example, for $m = 257$, we may simply view an element in U as a sequence of bytes (the digit 256 is unused). For any $a = (a_{r-1}, \ldots, a_0)$ we define the hash function

$$h_a(x) = \sum_{0 \le i < r} a_i x_i \bmod m\ .$$

and set $H_2 = \{h_a\ ; a \in U\}$. Then $|H_2| = m^r = N$.

Theorem 8 *H_2 as defined above is 1-universal.*

Proof: Let x and y be distinct elements of U. We need to count the number of a with $h_a(x) = h_a(y)$. Since $x \neq y$, there is an index i_0 such that $x_{i_0} \neq y_{i_0}$ (if there is more than one, choose an arbitrary one). Then $h_a(x) = h_a(y)$ is equivalent to

$$a_{i_0}(x_{i_0} - y_{i_0}) + \sum_{i \neq i_0} a_i(x_i - y_i) \equiv 0 \pmod m$$

For each choice of the a_i's for $i \neq i_0$, this equation has exactly one solution[3] in a_{i_0}. Thus there are exactly $m^{r-1} = |H_2|/m$ functions $h_a \in H_2$ with $h_a(x) = h_a(y)$. ∎

Classes H_1 has size N^2 and N, respectively, i.e., $O(\log N)$ bits are required to specify a function of the class. In Theorem **??** we exhibit a smaller universal class; $O(\log m + \log\log N)$ bits suffice to specify a member of that class. We will also show that this is best possible.

We analyze the expected behavior of universal hashing under the following assumptions:

1. The hash function h is chosen at random from some c-universal class H, i.e., each $h \in H$ is chosen with probability $1/|H|$.

2. Hashing with chaining is used.

Theorem 9 *Let $c \in I\!R$ and let H be a c-universal class of hash functions. Let $S \subseteq [0 .. N-1]$, $|S| = n$ and let $x \in [0 .. N-1]$. Then*

$$\sum_{h \in H} (1 + \delta_h(x, S))/|H| \leq 1 + c \cdot n/m \ .$$

The expected cost of an access, insert or delete operation is $O(1 + c \cdot \beta)$.

Proof: We have

$$\begin{aligned}
\sum_{h\in H}(1+\delta_h(x,S)) &= |H| + \sum_{h\in H}\sum_{y\in S}\delta_h(x,y) && \text{(definition of } \delta_h\text{)}\\
&= |H| + \sum_{y\in S}\sum_{h\in H}\delta_h(x,y) && \text{(reordering)}\\
&\leq |H| + \sum_{y\in S-\{x\}} c\cdot|H|/m && \text{(since } \delta_h(x,x)=0 \text{ and}\\
& && H \text{ is } c\text{-universal)}\\
&\leq |H|\cdot(1+c\cdot n/m)
\end{aligned}$$

∎

Theorem 9 reads the same as Theorem 2 (except from the factor c). However, there is a major difference between the two theorems. They are derived on completely different assumptions. In Theorem2 each subset $S \subseteq U$, $|S| = n$, is assumed to be equally likely, in Theorem 9 each element $h \in H$ is assumed to be equally likely. So universal hashing gives exactly the same performance as standard hashing, but the dices are now controlled by the algorithm not by the user.

The remainder of this section is in preliminary form and is only of theoretical interest. It may be skipped. TODO

[3] TODO: move this footnote to the chapter on foundations. Observe that the sum $\sum_{i \neq i_0} a_i(x_i - y_i)$ evaluates to some element $C \in [0 .. m-1]$. Also observe that for $w = x_{i_0} - y_{i_0} \neq 0$, the equation $a_{i_0} w + C \equiv 0 \pmod m$ has exactly one solution for a_{i_0}. The latter claim follows from the fact that the numbers $0w \bmod m$, $1w \bmod m$, $2w \bmod m$, ..., $(m-1)w \bmod m$ are pairwise different ($iw \bmod m = jw \bmod m$ implies that m divides $(i-j)w$. Since m is prime and does not divide w, it must divide $i-j$. Thus $i = j$.) and hence assume each of the values in $[0 .. m-1]$ exactly once.

For every fixed set $S \subseteq U$, a random element $h \in H$ will distribute S fairly evenly over the hash table, i.e., almost all functions $h \in H$ work well for S and only very few will give us bad performance on any fixed set S. Theorem 14 gives us an upper bound on the probability of bad performance.

delete this theorem; it is simply Chebychev's inequality.

Theorem 10 *Let $c \in I\!R$ and let H be a c-universal class. Let $S \subseteq [0\,..\,N-1]$, $|S| = n$ and $x \in [0\,..\,N-1]$. Let μ be the expected value of $\delta_h(x,S)$, i.e., $\mu = \left(\sum_{h\in H} \delta_h(x,S)\right)/|H|$. Then the probability that $\delta_h(x,S) \geq t \cdot \mu$ is less than $1/t$.* $N \in I\!N, t > 0$

Proof: Let $H' = \{h \in H;\ \delta_h(x,S) \geq t \cdot \mu\}$. Then

$$\begin{aligned}
\mu &= \Big(\sum_{h\in H} \delta_h(x,S)\Big)/|H| \\
&\geq \Big(\sum_{h\in H'} \delta_h(x,S)\Big)/|H| && (\text{since } \delta_h(x,S) \geq 0) \\
&\geq t \cdot \mu \cdot |H'|/|H| && (\text{since } \delta_h(x,S) \geq t\cdot\mu \text{ for } h \in H')
\end{aligned}$$

and hence $|H'| \leq |H|/t$.

∎

We infer from Theorem 14 that the probability that the performance is more than t times the expected performance is at most $1/t$. Much better bounds can often be obtained for specific universal classes; cf Exercise 7 where a 1-universal class with an $O(1/t^2)$ bound is described. We end this section with an estimate of the size of universal classes of hash functions.

Theorem 11 *Let $N, m \in I\!N$ and $N \geq m$.*

a) *Let $H \subseteq \{h\,;\, h : [0\,..\,N-1] \mapsto [0\,..\,m-1]\}$ be a c-universal class. Then $|H| \geq m\left(\lceil \log_m N\rceil - 1\right)/c$ and hence*

$$\log|H| = \Omega(\log m + \log\log N - \log c).$$

b) *There is a 8-universal class H of hash functions with*

$$\log|H| = O(\log m + \log\log N).$$

Proof: a) Let $H = \{h_1, \ldots, h_t\}$. As in the proof of Theorem 6b) we construct a sequence $U_0 = [0\,..\,N-1]$, $U_1, U_2, \ldots$, such that $h_1, \ldots, h_i$ are constant functions on U_i and $|U_i| \geq |U_{i-1}|/m \geq N/m^i$. Let $t_0 = \lceil \log_m N\rceil - 1$. Then $|U_{t_0}| > 1$. Let $x, y \in U_{t_0}, x \neq y$. Then

$$t_0 \leq \Big|\{h \in H\,;\, h(x) = h(y)\}\Big| \leq c \cdot |H|/m$$

since H is c-universal. Thus $|H| \geq m \cdot (\lceil \log_m N\rceil - 1)/c$.

b) Let $N, m \in I\!N$ and let t be minimal such that $t \ln p_t \geq m \ln N$. Here p_t denotes the t-th prime. Angenommen, $p_{2t} \leq m$. Dann würde folgen, daß $t \ln p_t < p_{2t} \ln p_{2t} \leq m \ln m \leq m \ln N$, ein Widerspruch zur Definition von t. Daraus folgt, daß $m/p_{2t} \leq 1$. Weiter ist $t = O(m \ln N)$. Let

$$H = \left\{g_{c,d}(h_l(x));\ t < l \leq 2t,\ 0 \leq c, d < p_{2t}\right\},$$

where

$$h_l(x) = x \bmod p_l$$

and

$$g_{c,d}(z) = \big[(c \cdot z + d) \bmod p_{2t}\big] \bmod m.$$

Then $|H| = t \cdot p_{2t}^2$ and hence $\log|H| = O(\log t) = O(\log m + \log\log N)$, since $\log p_{2t} = O(\log t)$ by the prime number theorem. It remains to be shown that H is 8-universal. Let $x, y \in [0 \mathinner{..} N-1]$, $x \neq y$, arbitrary. We have to show that

$$|\{(c,d,l);\ g_{c,d}(h_l(x)) = g_{c,d}\big(h_l(y)\big)\}| \leq 8 \cdot |H|/m.$$

If $g_{c,d}(h_l(x)) = g_{c,d}(h_l(y))$ then by definition of h_l and $g_{c,d}$

$$\big[c \cdot (x \bmod p_l) + d\big] \bmod p_{2t} = \big[c \cdot (y \bmod p_l) + d\big] \bmod p_{2t} \pmod{m}.$$

Thus there must exist $q \in [0 \mathinner{..} m-1]$ and $r, s \in [0 \mathinner{..} \lceil p_{2t}/m \rceil - 1]$ such that

$$\begin{aligned}\big[c \cdot (x \bmod p_l) + d\big] \bmod p_{2t} &= q + r \cdot m\\ \big[c \cdot (y \bmod p_l) + d\big] \bmod p_{2t} &= q + s \cdot m.\end{aligned}$$

We have to count the number of triples (c,d,l) which solve this pair of equations. we count the solutions in two groups: The first group contains all solutions (c,d,l) with $x \bmod p_l \neq y \bmod p_l$ and the second group contains all solutions (c,d,l) with $x \bmod p_l = y \bmod p_l$.

Group 1: Of course, there are at most t different l's such that $x \bmod p_l \neq y \bmod p_l$. For each such l and any choice of q, r and s there is exactly one pair (c,d) which solves our equations. This follows from the fact $\mathbb{Z}_{p_{2t}}$ is a field. Hence the number of solutions in group one is bounded by

$$\begin{aligned}t \cdot m \cdot (\lceil p_{2t}/m \rceil)^2 \leq t \cdot m \cdot (1 + p_{2t}/m)^2 &= (t \cdot p_{2t}^2/m) \cdot (1 + m/p_{2t})^2\\ &= (|H|/m) \cdot (1 + m/p_{2t})^2.\end{aligned}$$

Group 2: Let $L = \{l;\ t < l \leq 2t \text{ and } x \bmod p_l = y \bmod p_l\}$ and let $P = \prod_{l \in L} p_l$.
$P \leq N$ Then $P \geq p_t^{|L|}$. Also P divides $x - y$ and hence $P < N$. Thus $|L| \leq (\ln N)/\ln p_t \leq t/m$ by definition of t.

Consider any fixed $l \in L$ and any choice of q, r and s. If $r \neq s$ then there s no pair (c,d) solving our pair of equations. If $r = s$ then there are exactly p_{2t} pairs (c,d) solving our pair of equations. Hence the number of solutions in group two is at most

$$\begin{aligned}|L| \cdot m \cdot \lceil (p_{2t}/m) \rceil \cdot p_{2t} &\leq (t \cdot p_{2t}^2/m) \cdot (1 + m/p_{2t})\\ &= (|H|/m) \cdot (1 + m/p_{2t}).\end{aligned}$$

Altogether, we have shown that the number of solutions (c,d,l) is bounded by $2(1+\epsilon)^2|H|/m$ where $\epsilon = m/p_{2t} \leq 1$. (Note that $p_{2t} \leq m$ would imply
i.E.mehr $t \ln p_t < p_{2t} \ln p_{2t} \leq m \ln m \leq m \ln N$, a contradiction to the definition of t). Hence H is 8-universal. ∎

0.1.5 Controlling the Load Factor

The expected behavior of hashing depends on the load factor $\beta = n/m$. An operation can be executed in time $O(1)$ if β is bounded by a constant[4]. This implies that one either has to choose m large enough to begin with or one has to increase (decrease) the size of the hash table from time to time. The first solution is not general as it requires an a-priori bound on the size of the set to be stored and increases storage requirement, which also implies an increased running time in many environments. For example, a larger hash table is less likely to reside in cache memory.

We show how to keep the load factor in the vicinity of one by a simple doubling/halving strategy. We use a sequence $T_0, T_1, T_2, \ldots$ of hash tables of size 2^0, $2^1, 2^2, \ldots$ respectively. Also, for each i we have a hash function h_i. We start with the table T_0.

Suppose now, that we are currently using T_i and the load factor reaches 1 or $1/4$. In the first case, we switch to T_{i+1} by moving all 2^i elements present in T_i to T_{i+1}. The load factor of T_{i+1} will be $1/2$. In the second case, we switch to T_{i-1} by moving all $(1/4) \cdot 2^i$ elements present in T_i to T_{i-1}. The load factor of T_{i-1} will be $1/2$.

In either case we have a table with load factor 1/2 after restructuring. Before the next restructuring, we must either double the number of elements or half the number of elements. Also, the cost of restructuring is constant per element moved; observe that moving an element amounts to an insertion with the additional precondition that the element is not yet present. Thus there is no need to search through the list to which the element is a mapped. We might as well insert it at the beginning of the list and hence incur a cost of $O(1)$ independently of the length of the list. We can pay for a restructuring by charging it to the elements inserted or deleted since the last restructuring. In this way we charge a constant amount of work to each *insert* or *delete*.

Theorem 12 *Doubling/Halving keeps the load factor in* $[1/4, 1]$ *(except when T_0 is in use). The expected amortized cost of insertions, deletions and accesses is $O(1)$.*

We next describe a simple choice for the hash functions h_i. Let m be larger that the maximal size of the hash table and let h be a hash function from U to $[0\,..\,m-1]$; h may come from a universal class or be chosen otherwise. We set

$$h_i(x) = h(x) \bmod 2^i \ .$$

Is this a good choice? We show that our assumptions underlying the average case analysis of hashing stay essentially valid with this choice.

Our first analysis was based on assumptions (A1) and (A2), namely that the hash function is fair and that the set stored is a random subset of the universe. We show that h_i is almost as fair as h.

Lemma 1 *Let $2^i \le m$ and let h be a function from U to $[0\,..\,m-1]$. If $h^{-1}(j) \le cN/m$ for all $j \in [0\,..\,m-1]$, then $h_i^{-1}(j) \le c(1+2^i/m) \cdot N/2^i \le 2c \cdot N/2^i$ for all $j \in [0\,..\,2^i-1]$, i.e., h_i is almost as fair as h.*

[4]This assumes hashing with chaining. For hashing with open addressing, the load factor has to be kept below some constant less than one. The results of this sections are easily transfered to hashing with chaining.

Proof: There are at most $\lceil m/2^i \rceil$ elements $j' \in [0 \mathinner{..} m-1]$ with $j' \bmod 2^i = j$. We conclude that the number of elements in U that are mapped to j by h_i is bounded by

$$\lceil m/2^i \rceil cN/m \le c(N/2^i + N/m) = c(1 + 2^i/m) \cdot N/2^i \ .$$

∎

Our second analysis is based on assumptions (H1) and (H2), namely that the hash function is chosen randomly from a c-universal class of hash functions. In general, universality of a class H does not imply universality of the class $\{h_i \ ; h \in H\}$. For our concrete classes H_1 and H_2, the reduction $\bmod 2^i$ preserves universality. We leave H_1 to the exercises.

Lemma 2 *Let $2^i \le m$, let $U = [0 \mathinner{..} m-1]^r$ and for each $h \in H_2$ let h_i be defined by $h_i(x) = h(x) \bmod m$. The set $\{h_i \ ; h \in H_2\}$ is a $(2+2^i/m)$-universal class of functions from U to $[0 \mathinner{..} 2^i - 1]$.*

Proof: Let x and y be distinct elements of U. We need to count the number of a such that $(h_a)_i(x) = (h_a)_i(y)$. If $(h_a)_i(x) = (h_a)_i(y)$ then $h_a(x) - h_a(y) \equiv 2^i \pmod m$ and hence $h_a(x) - h_a(y) = \ell 2^i$ for some ℓ with $-\lfloor m/2^i \rfloor \le \ell \le \lfloor m/2^i \rfloor$. As in the proof of Theorem 8 we can argue that for each choice of ℓ there are exactly m^{r-1} functions h_a with $h_a(x) - h_a(y) = \ell 2^i$. Thus there are at most $m^{r-1}(2m/2^i + 1)$ functions $(h_a)_i$ with $(h_a)_i(x) = (h_a)_i(y)$. We conclude the proof by observing that

$$m^{r-1}(2m/2^i + 1) = 2m^r/2^i + m^{r-1} = (2 + 2^i/m) \cdot m^r/2^i \ .$$

∎

0.1.6 Implementation Issues

We discuss implementations of hashing and a hash function for strings. Hashing with chaining employs a hash table and for each entry of the hash table a linear list. There are different ways of organizing the lists. We discuss two (see also Figure **??**):

- We have a table of pointers to list nodes. The entry $T[i]$ points to the first node of the i-th list.
- We have a table of list nodes. The entry $T[i]$ is the first node of the i-th list.

In both designs we may or may not use sentinels. In both designs, the lists are comprised of nodes with three fields: a key, an information, and a pointer to the next node. We assume that the space requirement for a node is $1 + c$ times the space requirement for a pointer, e.g., if key and information are *int*s or pointers we have $c = 2$.

Slim Tables: We have a table T of size m and a list node for each element of the set stored. The space requirement is $m + (1 + c)n = (1 + 1/c + 1/(c\beta)) \cdot cn$. An access operation to an element at the j-th position of its list (positions are counted starting at 1) requires $1 + j$ memory accesses. The first access is to the table to read the address of the first node and then there are j accesses to list nodes. In general, each access will lead to a cache fault (except if the table is so small that it fits into the cache) as we cannot expect any locality; after all, hashing works so well because it distributes the set stored fairly evenly over the table.

The expected number of memory accesses for an unsuccessful search or an insertion will be $1 + \beta$, the expected number of memory accesses for a successful search or a deletion will be $1 + \beta/2$ (on average, a successful search goes halfway down a list).

A rehash is triggered when β exceeds a certain threshold *max_beta*. For simplicity, we only deal with the case of a growing table. We allocate a table of twice the size and move all list nodes to the appropriate list in the new table. Observe that we can reuse the list nodes for the new table and there is no need to move the contents of list nodes around (this will be different for the other implementation). In this way, the space overhead varies between $(1 + 1/c + 1/(c \cdot \mathit{max\_beta}))$ and $(1 + 1/c + 2/(c \cdot \mathit{max\_beta}))$. The mean value is $(1 + 1/c + 1.5/(c \cdot \mathit{max\_beta}))$. For $c = 2$, the mean overhead is less than two if $\mathit{max\_beta} = 3/2$.

It is easy to provide an array like subscript operator, which given a key returns a reference to the associated information. We simply locate the node containing the key and return a reference to the information field of the node. Rehashes cause no problems, since a key together with its associated information always resides in the same node.

Assume that $c = 2$ and that we want to have a space overhead of at most 100%. Thus $1/\beta \leq 1$. We should therefore trigger a rehash when β reaches 2; then $\beta = 1$ after the rehash.

We next give a complete implementation. We postulate the existence of a function *int Hash*(*const K*&) that maps keys to integers. The key type K must also support the equality and the inequality operators `==`, `!=`.

⟨*implementation.c*⟩≡

```
#include <LEDA/misc.h>
⟨hash nodes⟩
⟨slim hash tables⟩
⟨fat hash tables⟩
```

A hash node consists of a key, an information, and a pointer to a hash node. We make it a friend of both implementations, which we call *slim_hash_array* and *fat_hash_array*, respectively.

⟨*hash nodes*⟩≡

```
template <class K, class I> class slim_hash_array;
template <class K, class I> class fat_hash_array_1;
template <class K, class I> class fat_hash_array_2;
template <class K, class I>
class hash_node{
  K key;
  I inf;
  hash_node* next;
  public:
  friend class slim_hash_array<K,I>;
  friend class fat_hash_array_1<K,I>;
  friend class fat_hash_array_2<K,I>;
```

```
  hash_node(const K& k, const I& i, hash_node* n = nil) :
    key(k), inf(i), next(n) {}
  hash_node(): next(nil) {}
  void set(const K& k, const I& i, hash_node* n = nil)
    { key = k; inf = i; next = n; }
  LEDA_MEMORY(hash_node);
};
```

We have two constructors: the first constructor allows us to set all three components of a node and the second constructor initializes only the next field. We also have a function *set* that allows us to change the data members of an already existing node. We use our efficient memory manager for hash nodes.

⟨*slim hash tables*⟩≡

```
template <class K, class I> class slim_hash_array{
  typedef hash_node<K,I> node;
  I idef;
  node** T;
  int m, n;
  double max_beta;
  mutable int index;
⟨private members of slim hash tables⟩
    slim_hash_array(const slim_hash_array<K,I>&); // no copy constructor
public:
slim_hash_array(const I& def, double space_factor = 2) :
  idef(def), m(1024), n(0)
{ T = new node*[m];
  for (int i = 0; i < m; i++ ) T[i] = 0;
  double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
  if ( space_factor <= 1 + 1/c )
  { error_handler(0,"space factor must be above 1 + 1/c.");
    space_factor = 1 + 2/c;
  }
  max_beta = 1.5 / (c * (space_factor - 1) - 1);
}
~slim_hash_array()
{ for (int index = 0; index < m; index++)
  { node* v = T[index];
    while ( v != nil )
      { node* w = v->next; delete v; v = w; }
  }
  delete[] T;
}
int size() const { return n; }
double beta() const { return max_beta; }
double min_space_factor() const
{ double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
 return 1 + 1/c;
```

```
}
⟨public members of slim hash tables⟩
};
```

We view a hash table as a function from keys to informations. Initially, all function values are equal to a default value *idef* given in the constructor. The other date members are the hash table T, the size m of the hash table, the number n of elements stored, and the maximal load factor *max_beta*, and an integer *index* that we use for communication between the members. We qualify *index* as mutable so that even const member functions can change it. We start with a table of size 1024. We do not use sentinels and hence all table entries are made to point to *nil* in the constructor. The constructor has an optional argument *space_factor* which is the mean overhead in space. We compute the maximal load factor from the equation $space\_factor = (1 + 1/c + 1.5/(c \cdot max\_beta))$. The space overhead cannot be chosen at or below $1 + 1/c$. The destructor empties all lists and then deletes the table. The *size* function returns n and the function *beta* returns *max_beta*.

The function *access* finds the node with key k and returns it. If there is no such node, it returns *nil*.

The function *insert* inserts the pair (k,i) into the hash table. It has the precondition that there is no pair with key k yet in the hash table and that k maps to location *index*; *index* will have been computed by a preceding call of *access* set If the load factor exceeds *max_beta*, a rehash is performed and *index* is recomputed. We discuss the implementation of *rehash* below. Otherwise, We add a new node at the beginning of the list $T[index]$ and return the new node. Note that it is safe to add the new node at the beginning of the list because k is known to be a new key.

⟨*private members of slim hash tables*⟩≡

```
node* access(const K& k) const
{ index = Hash(k) % m;
  node* v = T[index];
  while ( v != nil && v->key != k) v = v->next;
  return v;
}
node* insert(const K& k, const I& i)
{ if ( n >= max_beta * m)
  { rehash();
    index = Hash(k) % m;
  }
  T[index] = new node(k,i,T[index]);
  return T[index];
}
```

A read access first determines the node with key k. If there is none, *idef* is returned. Otherwise, the information field of the node is returned.

A write access also determines the node with key k. If there is none, a new node with content (k,i) is inserted. Otherwise, the information field of the node is changed.

⟨*public members of slim hash tables*⟩≡

```
I inf(const K& k) const
{ node* v = access(k);
  return ( v == nil ? idef : v->inf );
}
void set_inf(const K& k, const I& i)
{ node* v = access(k);
  if ( v == nil ) { n++; insert(k,i); }
  else v->inf = i;
}
```

It remains to give the details of a rehash. There is not much to explain. We save the old m and the old table, make a new table of twice the size and move all nodes to the appropriate list of the new table. Finally, we delete the old table.

⟨*private members of slim hash tables*⟩+≡

```
void rehash()
{
  node** old_T = T;
  int old_m = m;
  m = 2*m;
  T = new node*[m];
  for (int old_index = 0; old_index < old_m; old_index++)
  { node* v = old_T[old_index];
    T[old_index] = T[old_m + old_index] = 0; // initialization
    while ( v != nil )
    { node* w = v->next;
      int index = Hash(v->key) % m;
      v->next = T[index];
      T[index] = v;
      v = w;
    }
  }
  delete[] old_T;
}
```

A Subscript Operator for Slim Hash Tables: The functions *inf* and *set_inf* are awkward to use. It is much nicer to have an array-like interface throught the use of the subscript operator[5]. The subscript operator returns a reference to the information field associated with a key (the const subscript operator returns a constant reference).

We have two versions of the subscript operator. The first one returns a reference to the information associated with a key k; if there is no such information yet, the pair $(k, idef)$ is added to the hash table. The second applies only to const-objects of type slim hash array and returns a const-reference, i.e., a reference that can only be read but not modified. The second operator returns a reference to *idef*, if there is no pair with

[5]This paragraph requires advanced knowledge of C++ and maybe skipped.

key k yet.

⟨*public members of slim hash tables*⟩+≡

```
I& operator[](const K& k)
{ node* v = access(k);
  if ( v == nil ) { n++; v = insert(k,idef); }
  return v->inf;
}
const I& operator[](const K& k) const
{ node* v = access(k);
  if ( v == nil ) return idef;
  return v->inf;
}
```

We need to tell the reader a bit more about C++ syntax and semantics at this point. In C++, a reference to an object can either be const or not. The default are non-const objects. Only member functions qualified as const are applicable to const objects. If there are member functions whose signature differs only in the qualifier const (the return value is not part of the signature of a function), the member function qualified const is applied to const objects and the alternative is applied to non-const objects. In our case the second alternative of the subscript operator is applied to const hash tables and the first alternative is applied to non-const hash tables.

Consider the following code snippet. We create a hash array and read $n = 100000$ elements in it, once using the *inf* member function and once using the subscript operator.

⟨*operator_experiment.c*⟩≡

⟨*implementation.c*⟩

```
int main(){
slim_hash_array<int,int> H(0);
int S; int n = 100000;
for (int k = 0; k < n; k++) S += H.inf(k);
cout << H.size() << endl;
//for (int k = 0; k < n; k++) S += (((const slim_hash_array<int,int>) H))[k];
// syntax error
for (int k = 0; k < n; k++) S += (*((const slim_hash_array<int,int>*) &H))[k];
cout << H.size() << endl;
for (int k = 0; k < n; k++) S += H[k];
cout << H.size() << endl;
}
```

The snippet outputs

```
0
0
100000
```

Why? The *inf* member function does not make any additions to the hash table and hence the size of H is still zero after the n reads, the subscript operator (when applied to a non-const hash table) inserts non-existing keys and hence the size of H is n after the last n reads. Observe, that the H in the last n reads is only semantically a const object, syntactically it is a non-const object. In the middle n reads, we have casted H to a const-object and hence the const version of the subscript operator applies. A remark about the conversion is in order. We took the address of H, converted it a const-pointer and took the content of the pointer; this gives us a const-object of type *slim_hash_table*. The more direct conversion given in the line above is incorrect, as it will call the copy constructor to make a const-copy of H; it will not cast H itself to a const-object. The incorrect conversion will lead to a syntax error, since we made the copy constructor private. Had we not done so, we program would lead to a runtime error, because the default version of the copy constructor would be used.

The conversion to a const-object has a frightful syntax. Hash tables are frequently used in a two-phase scheme. In the first phase, only insertions are made, and in the second phase, only lookups are made. In such a situation it is worthwhile to cast the hash table to a const-object between the phases.

Fat Tables, Version One: We have a table T of hash nodes. The table entry $T[i]$ stores the first element of the i-th list. How do we encode empty lists? We let the next-pointer of $T[i]$ point to *nil* iff the i-th list is empty. If the i-th list is non-empty, we let the last element of the list point to a special element *STOP* which we can also use as a sentinel in searches.

Coding of fat table version of hashing with chaining is considerably simplified if we allocate a hash table of size $t\_size = fm$ for some factor $f > 1$ and use the nodes with index m to $t\_size - 1$ for the list nodes which are not first in their list. We keep an index *free* into the table; *free* is initialized to m. Whenever, a new list node is needed, the node $T[\mathit{free}]$ is used and *free* is incremented. When *free* reaches t_size a rehash is triggered and m is doubled.

The space requirement of the implementation is $fm(1+c) = f/\beta \cdot (1+1/c) \cdot cn$. The load factor β and the factor f are not independent parameters. One determines the other as we see next. Consider the situation just before a rehash. We have $n = fm - m'$ elements stored in the table, where m' is the number of empty lists; observe that $m - m'$ among the first m nodes and all other nodes are used. The number of empty lists is $m' = me^{-\beta}$ according to Theorem **??**. Thus $n = fm - m^e{-}\beta$ and hence $\beta + e^{-\beta} = f$. The space overhead just before a rehash is therefore equal to $(1 + e^{-\beta}/\beta) \cdot (1 + 1/c)$. Immediately, after a rehash it is twice this number. The mean value is $1.5(1 + e^{-\beta}/\beta) \cdot (1 + 1/c)$.

⟨*fat hash tables*⟩≡

```
#include <math.h>
template <class K, class I> class fat_hash_array_1{
  typedef hash_node<K,I> node;
  I idef;
  node* T;
  mutable node STOP;
  double f, max_beta;
  int m, t_size, free, n;
  mutable int index;
```

⟨*private members of fat hash tables, version 1*⟩

```
public:
```

```
fat_hash_array_1(const I& def, double space_factor = 3.0) :
  idef(def), m(1024), free(m), n(0)
  { double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
    if ( space_factor <= 1.5 * ( 1 + 1/c) )
      { error_handler(0,"space factor must be above 1.5(1 + 1/c)");
        space_factor = 1.5 + 3/c;
      }
    double max_beta_low = 1.5 * ( 1 + 1/c);
    max_beta = max_beta_low;
    do { max_beta *= 2;
    } while ( (1 + exp(- max_beta)/max_beta) * (1 + 1/c) > space_factor );
    double max_beta_high = max_beta;
    do { max_beta = (max_beta_low + max_beta_high)/2;
         if ( (1 + exp(- max_beta)/max_beta)*(1 + 1/c) > space_factor )
           max_beta_low = max_beta;
         else max_beta_high = max_beta;
    } while (max_beta_high - max_beta_low >= 0.01);
    f = max_beta + exp(- max_beta);
    t_size = (int)(f * m),
    T = new node[t_size];
  }
~fat_hash_array_1() { delete[] T; }
int size() const { return n; }
double beta() const { return max_beta; }
double min_space_factor() const
{ double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
 return 1.5 * (1 + 1/c);
}
⟨public members of fat hash tables, version 1⟩
};
```

In the constructor we compute *max_beta* and f from the allows space overhead. The space overhead is more than $1.5(1 + 1/c)$. We also set *idef*, *free*, and *n* as explained above and define a hash table of size m. The default constructor of *hash_nodes* initializes the next pointers to *nil*. The destructor simply deallocates the table. *size* returns n.

The access function first determines whether the list to which k is mapped is empty. If yes, it returns *nil*. If no, it uses linear search to search for k. In the linear search we may use *STOP* as a sentinel. An *insert*(k, i) triggers a rehash if the table is full. If the table is non-full and the k to which k is mapped is empty, we insert the pair (k, i) into the appropriate table entry. Otherwise, we add a new second element to the list.

Reads and writes are implemented in terms of *access* and *insert* as described before. We postpone the definition of the subscript operator.

⟨*private members of fat hash tables, version 1*⟩≡

```
node* access(const K& k) const
{ index = Hash(k) % m;
  node* v = &T[index];
  if ( v->next == nil ) return nil;
  while ( v != &STOP && v->key != k) v = v->next;
  return ( v == &STOP ? nil : v );
```

```
}
node* sentinel_access(const K& k) const
{ index = Hash(k) % m;
  node* v = &T[index];
  if ( v->next == nil ) return nil;
  STOP.key = k;
  while ( v->key != k) v = v->next;
  return ( v == &STOP ? nil : v );
}
node* insert(const K& k, const I& i)
{ if ( free == t_size)
  { rehash();
    index = Hash(k) % m;
  }
  node* v = &T[index];
  if (v->next == nil)
    { v->set(k,i,&STOP); return v; }
  else
    { T[free].set(k,i,v->next); v->next = &T[free]; free++; return &T[free]; }
}
```

⟨*public members of fat hash tables, version 1*⟩≡

```
I inf(const K& k) const
{ node* v = access(k);
  return ( v == nil ? idef : v->inf );
}
I sentinel_inf(const K& k) const
{ node* v = sentinel_access(k);
  return ( v == nil ? idef : v->inf );
}
void set_inf(const K& k, const I& i)
{ node* v = access(k);
  if ( v == nil ) { n++; insert(k,i); }
  else v->inf = i;
}
```

It remains to give the details of a rehash. We save the old table and the old table size and allocate a new table of twice the size. We then step through all entries of the old table and move the contents of all used nodes to the new table. Finally, we delete the old table.

⟨*private members of fat hash tables, version 1*⟩+≡

```
void rehash()
{
  node* old_T = T;
  int old_t_size = t_size;
  m = 2*m;
  t_size = 2*t_size;
  free = m;
  T = new node[t_size];
  for (int old_index = 0; old_index < old_t_size; old_index++)
```

```
    { node* v = &old_T[old_index];
      if ( v->next == nil ) continue;
      index = Hash(v->key) % m;
      insert(v->key,v->inf);
    }
    delete[] old_T;
  }
```

Fat Tables, Version Two: The solution described in the previous paragraph uses a large amount of space. We can obtain considerable savings in space, if we allocate the non-first nodes separately and not in the table.

The space requirement for n elements is is $(n+m')(c+1)$, where m' is the number of empty lists. The number of empty lists is about $me^{-\beta}$. Thus the space requirement is $(n+me^{-\beta})(c+1) = (1+e^{-\beta}/\beta)(1+1/c)\cdot cn$. The space overhead just before a rehash is therefore equal to $(1+e^{-betamax}/\beta_{max})(1+1/c)$. Immediately after a rehash it is $(1+2e^{-\beta_{max}/2}/\beta_{max})(1+1/c)$. The mean space overhead is approximately $(1+(2e^{-\beta_{max}/2}+e^{-\beta_{max}})/(2\beta_{max}))(1+1/c)$.

⟨*fat hash tables*⟩+≡

```
#include <math.h>
⟨class hash table ref⟩ // becomes relevant in the paragraph on the subscript
                          // operator for fat hash arrays
template <class K, class I> class fat_hash_array_2{
  typedef hash_node<K,I> node;
  I idef;
  node* T;
  mutable node STOP;
  double max_beta;
  int m, n;
  mutable int index;
⟨private members of fat hash tables, version 2⟩
public:
fat_hash_array_2(const I& def, double space_factor = 2.0) :
  idef(def), m(1024), n(0)
  { double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
    if ( space_factor <= ( 1 + 1/c) )
      { error_handler(0,"space factor must be above (1 + 1/c)");
        space_factor = 1 + 2/c;
      }
    double max_beta_low = 1 + 1/c; double f;
    max_beta = max_beta_low;
    do { max_beta = 2*max_beta;
         f  = (1 + (2*exp(- max_beta/2)+exp(-max_beta))/(2*max_beta))* (1
+ 1/c);
    } while ( f > space_factor );
    double max_beta_high = max_beta;
    do { max_beta = (max_beta_low + max_beta_high)/2;
         f = (1 + (2*exp(- max_beta/2)+exp(-max_beta))/(2*max_beta))* (1 + 1/c);
         if ( f > space_factor )
```

```
            max_beta_low = max_beta;
          else max_beta_high = max_beta;
      } while (max_beta_high - max_beta_low >= 0.01);
      T = new node[m];
    }
  ~fat_hash_array_2()
  { for (index = 0; index < m; index ++)
    { if ( T[index].next == nil ) continue;
      node* v = T[index].next;
      while ( v != &STOP ) { node* w = v->next; delete v; v = w; }
    }
    delete[] T;
  }
  int size() const { return n; }
  double beta() const { return max_beta; }
  double min_space_factor() const
  { double c = ((double)(sizeof(K) + sizeof(I)))/sizeof(void*);
   return 1 + 1/c;
  }
  ⟨public members of fat hash tables, version 2⟩
  };
```

In the constructor we compute *max_beta* and f from the allows space overhead. The space overhead is more than $1.5(1 + 1/c)$. We also set *idef*, *free*, and n as explained above and define a hash table of size m. The default constructor of *hash_nodes* initializes the next pointers to *nil*. The destructor simply deallocates the table. *size* returns n.

The access function first determines whether the list to which k is mapped is empty. If yes, it returns *nil*. If no, it uses linear search to search for k. In the linear search we may use *STOP* as a sentinel. An *insert*(k, i) triggers a rehash if the table is full. If the table is non-full and the k to which k is mapped is empty, we insert the pair (k, i) into the appropriate table entry. Otherwise, we add a new second element to the list.

Reads and writes are implemented in terms of *access* and *insert* as described before. We postpone the definition of the subscript operator.

⟨*private members of fat hash tables, version 2*⟩≡

```
  node* access(const K& k) const
  { index = Hash(k) % m;
    node* v = &T[index];
    if ( v->next == nil ) return nil;
    while ( v != &STOP && v->key != k) v = v->next;
    return ( v == &STOP ? nil : v );
  }
  node* sentinel_access(const K& k) const
  { index = Hash(k) % m;
    node* v = &T[index];
    if ( v->next == nil ) return nil;
    STOP.key = k;
    while ( v->key != k) v = v->next;
    return ( v == &STOP ? nil : v );
  }
```

```
node* insert(const K& k, const I& i)
{ if ( n >= max_beta * m )
  { rehash();
    index = Hash(k) % m;
  }
  node* v = &T[index];
  if (v->next == nil)
    { v->set(k,i,&STOP); return v; }
  else
    { v->next = new node(k,i,v->next);
      return v->next;
    }
}
```

⟨*public members of fat hash tables, version 2*⟩≡

```
I inf(const K& k) const
{ node* v = access(k);
  return ( v == nil ? idef : v->inf );
}
I sentinel_inf(const K& k) const
{ node* v = sentinel_access(k);
  return ( v == nil ? idef : v->inf );
}
void set_inf(const K& k, const I& i)
{ node* v = access(k);
  if ( v == nil ) { n++; insert(k,i); }
  else v->inf = i;
}
```

It remains to give the details of a rehash. We save the old table and the old table size and allocate a new table of twice the size. We then step through all entries of the old table and move the contents of all used nodes to the new table. Finally, we delete the old table.

⟨*private members of fat hash tables, version 2*⟩+≡

```
void rehash()
{
  node* old_T = T;
  int old_m = m;
  m = 2*m;
  T = new node[m];
  for (int old_index = 0; old_index < old_m; old_index++)
  { node* v = &old_T[old_index];
    if ( v->next == nil ) continue;
    while ( v != &STOP )
    { node* w = v->next;
      index = Hash(v->key) % m;
      insert(v->key,v->inf);
      v = w;
    }
```

	fat hash tables, 1			fat hash tables, 2			
s	β	I	L	β	I	L	L*
1.6	–	–	–	3.249	3.07	4.91	4.81
1.8	–	–	–	2.08	2.34	5.16	5.1
2	–	–	–	1.623	2.79	4.01	3.94
2.2	–	–	–	1.506	2.57	3.99	3.92
2.4	2.259	2.37	6.86	1.506	2.59	4.01	3.94
1.6	–	–	–	3.249	1.88	3.43	3.44
1.8	–	–	–	2.08	1.54	3.65	3.64
2	–	–	–	1.623	1.91	2.78	2.78
2.2	–	–	–	1.506	1.72	2.76	2.77
2.4	2.259	1.53	3.73	1.506	1.7	2.77	2.77

Table 3: The running times of our two hash implementations using fat tables for different space overheads s. We assume that a key and an information require the space of $c = 2$ pointers. The first column shows the desired mean overhead in space. For each of the four implementations, we show the value of β_{max}, the time for n insertions and the time for $4n$ lookups for $n = 10^6$. Dashes indicate that the desired space overhead cannot be achieved. In the case of the second version fat hash tables, the fourth column shows the effect of using a sentinel in a search. The upper part shows the running time on a SUN ULTRA SPARC and the lower part shows the running times on a INTEL PENTIUM. We see that the use of sentinels helps somewhat on the SPARC and makes hardly a difference for the PENTIUM.
TODO: make all times the average of then runs.

```
    }
    delete[] old_T;
  }
```

We come to the comparison of our two version of fat hash tables. Which one is better? The second version works for space overheads larger than $1 + 1/c$, the first version has an overhead of at least $1.5(1 + 1/c)$. For a fixed space overhead s, the second version can work with a smaller β. For example, for $s = 2.4$, the first version needs to use $\beta_{max} = 2.4$, whilst the second version can work with $\beta_{max} = 1.51$, i.e., the average length of the lists is $2.4/1.51 \approx 1.6$ times longer in the first version. Table **??** confirms this analysis. It compares the running times of our two implementations of hashing using fat tables for different space overheads. The second version is clearly superior.

Open Addressing: Hashing with open addressing uses a table of m slim hash nodes. A slim hash node stores a key and an information; there is no need for a next pointer. We implement open addressing with linear probing.

The space requirement of the implementation is $mc = 1/\beta \cdot cn$. Whenever β reaches *max_beta*, we move to a table twice the size. In this way the space overhead varies between $1/\beta_{max}$ and $2/\beta_{max}$. The mean overhead is $1.5/\beta_{max}$

How do we recognize empty table entries? We could try to encode it into the key (or information) field. For example, if we know that the keys are non-negative integers, we might use a key value of -1 to encode that an entry is unused. This solution is not general and cannot be used when for a parameterized data type hash table. When the key type can be anything, we cannot exclude a bit pattern as a possible key.

A general solution uses a bit vector. LEDA's *int_sets* implement bit-vectors. We reuse them here.

⟨*open hash tables*⟩≡

```
#include <math.h>
#include <LEDA/int_set.h>
template <class K, class I> class open_hash_array;
template <class K, class I>
class slim_node{
  K key;
  I inf;
  public:
  friend class open_hash_array<K,I>;
  slim_node(const K& k, const I& i) : key(k), inf(i) {}
  slim_node() {}
  void set(const K& k, const I& i) { key = k; inf = i; }
  LEDA_MEMORY(slim_node);
};

template <class K, class I>
class open_hash_array{
  typedef slim_node<K,I> node;
  I idef;
  node* T;
  double max_beta;
  int m, n;
  mutable int index, offset;
  int_set* used_indices;
⟨private members of open hash tables⟩
public:
open_hash_array(const I& def, double space_factor = 2.0) :
  idef(def), m(1024), n(0)
{ T = new node[m]; used_indices = new int_set(m);
  if ( space_factor <= 1.5 )
  { error_handler(0,"space factor must be above 1.5");
    space_factor = 2.0;
  }
  max_beta = 1.5/space_factor;
}
~open_hash_array() { delete[] T; }
int size() const { return n; }
double beta() const { return max_beta; }
double min_space_factor() const { return 1.5; }
⟨public members of open hash tables⟩
};
```

In the constructor we initialize the data members in the obvious way. We compute the maximal load factor from the space overhead according to the equation *space_factor* $= 1.5/\beta_{max}$. The space factor must be larger than 1.5. The destructor simply deletes the table and *size* returns n.

The access function evaluates the hash function and then uses linear probing to search for the key. Emptyness of a slot is checked by inspecting the bit vector *used_indices*. A successful search returns the index of the table entry containing the search key, and unsuccessful search returns the index of the empty slot into which the key could be inserted.

An *insert* performs a rehash when the maximal load factor is exceeded. After a rehash, we call *access* to compute the index of an empty slot.

Reads and writes are implemented in terms of *access* and *insert* as described before. We postpone the definition of the subscript operator.

⟨*private members of open hash tables*⟩≡

```
bool is_used(int index) const
{ return used_indices->member(index); }
int access(const K& k) const
{ index = Hash(k) % m;
  offset = 1;
  while ( is_used(index) && T[index].key != k) index = (index + offset) % m;
  return index;
}
void insert(const K& k, const I& i)
{ if ( n >= max_beta * m)
  { rehash();
    access(k);
  }
  T[index].key = k; T[index].inf = i; used_indices->insert(index);
}
```

⟨*public members of open hash tables*⟩≡

```
I inf(const K& k) const
{ index = access(k);
  return ( !is_used(index)? idef : T[index].inf );
}
void set_inf(const K& k, const I& i)
{ index = access(k);
  if ( !is_used(index) ) { n++; insert(k,i); }
  else T[index].inf = i;
}
```

It remains to give the details of a rehash. We save the old table, the old bit vector and the old table size and allocate a new table of twice the size. We then step through all entries of the old table and move the contents of all used nodes to the new table. Finally, we delete the old table and the old bit vector.

⟨*private members of open hash tables*⟩+≡

```
void rehash()
{
  node* old_T = T;
  int old_m = m;
  int_set* old_used_indices = used_indices;
  m = 2*m;
  T = new node[m];
  used_indices = new int_set(m);
  for (int old_index = 0; old_index < old_m; old_index++)
  {
    if ( !old_used_indices->member(old_index) ) continue;
    index = access(old_T[old_index].key);
    insert(old_T[old_index].key,old_T[old_index].inf);
  }
  delete[] old_T;
  delete old_used_indices;
}
```

Testing: We tested all implementations by running them against the LEDA implementation of hash arrays.

Comparison: We compare the slim table implementation, the second variant of fat tables, and open addressing, see Table 4. For $c = 2$ and $s = 1.8$, slim hash table need to use $\beta_{max} = 2.5$, the second variant of fat tables can use $\beta_{max} = 2.08$, and open addressing uses $\beta_{max} = 0.8333$. For an insertion or unsuccessful search[6], slim hash tables require an expected number of $1 + \beta$ memory accesses, fat tables require β memory accesses, and open addressing requires $1/(1-\beta)$ memory accesses; β varies between $\beta_{max}/2$ and β_{max}. The average number of memory accesses is therefore $1 + 3/4\beta_{max} \approx 2.75$ for slim tables, is 1.56 for fat tables, and is

$$(2/\beta_{max}) \int_{\beta_{max}/2}^{\beta_{max}} 1/(1-\beta)d\beta = -(2/\beta_{max}) \ln(1-\beta)|_{\beta_{max}/2}^{\beta_{max}}$$
$$= (2/\beta_{max}) \ln(1-\beta_{max}/2)/(1-\beta_{max}) \approx (\ln 3)/0.4 \approx 2.74$$

for open addressing. We should therefore expect that fat tables beat the other two methods and that slim tables and open addressing come out about even. The experiment confirms the preciction that fat tables beat slim tables. Open hash tables do much better than predicted. The reason is that we use linear probing and that linear probing benefits from cache memory. Assume, for example, that two nodes fit into a cache line. Then a search in a list of length l results in $1 + (l-1)/2$ memory accesses; the 1 counts the access to the first node and the remaining $l - 1$ nodes require an expected number of $(l-1)/2$ additional accesses[7]. TODO: what does this analysis of the caching behavior mean? Exercise 4 asks you to do a similar analysis for a large space overhead factor $s = 2.4$.

[6]We made our experiment with unsuccessful searches.

[7]Observe either the first or the first two elements of a list are in the same cache line. Therefore the number of additional accesses is $(\lceil (l-1)/2 \rceil + \lceil (l-2)/2 \rceil)/2 = (l-1)/2$.

	slim hash tables				fat hash tables, 2				open hash tables			
s	β	I	L	T	β	I	L	T	β	I	L	T
1.6	7.5	4.48	11.09	15.57	3.249	3.52	4.92	8.44	0.9375	7.4	4.99	12.39
1.8	2.5	2.71	8.03	10.74	2.08	2.65	5.39	8.04	0.8333	3.82	5	8.82
2	1.5	2.89	6.54	9.43	1.623	3.09	4.17	7.26	0.75	3.1	5.06	8.16
2.2	1.071	2.36	6.44	8.8	1.506	2.98	4.18	7.16	0.6818	2.67	5.03	7.7
2.4	0.8333	2.94	5.74	8.68	1.506	3.11	4.27	7.38	0.625	2.55	4.96	7.51
1.6	7.5	3.17	7.08	10.25	3.249	2.41	3.43	5.84	0.9375	4.05	3.24	7.29
1.8	2.5	1.9	4.63	6.53	2.08	1.88	3.82	5.7	0.8333	2.27	3.22	5.49
2	1.5	1.94	3.44	5.38	1.623	2.12	2.86	4.98	0.75	1.86	3.23	5.09
2.2	1.071	1.55	3.47	5.02	1.506	2.05	2.87	4.92	0.6818	1.63	3.31	4.94
2.4	0.8333	1.92	2.85	4.77	1.506	2.04	2.86	4.9	0.625	1.54	3.19	4.73

Table 4: The running times of our hash implementations for different space overheads s. We assume that a key and an information require the space of $c = 2$ pointers. The first column shows the desired mean overhead in space. For each of the four implementations, we show the value of β_{max}, the time for n insertions, the time for $4n$ lookups for $n = 10^6$, and the total time. The upper part is for the SPARC and the lower part is for the PENTIUM.

In LEDA, we use the first variant of fat hash tables for *maps* and the second variant for *h_arrays*. In both cases we use sentinels. I do not recall why we made these choices. Certainly, the choices were made with incomplete information. Now we can make rational choices.

Sentinels: Should be use sentinels? If we use sentinels, we should use the same sentinel for all lists. If we do then we should use the same sentinel for all lists. Sentinel will be a high frequency item and hence will be in cache. Good for unsuccessful searches. However, lists are short. So we trade a constant amount of work (for copying the search key to the sentinel) for a constant amount of work (the comparisons in each step of the search). Table 4 shows that the use of sentinels does not make a big difference.

Hashing and Iteration: It is frequently necessary to iterate over the elements in a hash table. For example, we may want to print a list of the elements stored in the hash table. With any of our implementations, iteration is easily implemented. We leave the details to the reader (see exercise **??**.

Sometimes it is necessary to iterate over the elements in some predefined order. Consider the following example. We use a hash table to store account information. The hash key is an account number and the information associated with an account number are the details of the account. Assume further that we want to process a large set of updates, say a set of updates which affects a certain percentage p of the the accounts. Instead of making a lookup into the hash table for each update, we may alternatively use the following strategy. We compute the hash value of each update and sort the updates by hash value. We then iterate over the hash table and process all requests during a single scan over the hash table. For large values of p, the alternative strategy is more efficient.

A Subscript Operator for Fat Hash Tables and Open Hash Tables: We have the problem that rehashes move objects around and hence invalidate references to objects. Consider an assignment `H[x] = H[y];` most compilers will translate it into a sequence of three statements

```
I& ref_x = &H[x]; // establish a reference to the object associated with x
I  val_y =  H[y]; // obtain the object associated with y
   ref_x = val_y; // assign
```

A problem arises when the second line causes a rehash. The rehash will move the object associated with x to a new location and the assignment will be to the old location. We know of now general solution to this problem which does not suffer from a significant loss of efficiency. We next describe two partial solutions which support limited use of the subscript operator.

The first solution restricts the use of the subscript operator for writing access to the left hand side of assignments; the use for reading access is unrestricted.

```
// properties of first solution
H[x] = 5;              // supported
H[x] = H[y] + 7;       // supported
swap(H[x],H[y]);       // not supported; syntax error
H[x] = H[y] + H[z] ;   // supported
```

The second solution is used in LEDA's *maps* and *h_arrays* and allows to use the subscript operator in all statements that involve at most two accesses into a hash table.

```
// properties of second solution
H[x] = 5;              // supported
H[x] = H[y] + 7;       // supported
swap(H[x],H[y]);       // supported
H[x] = H[y] + H[z] ;   // may fail, three accesses in one statement
```

We come to the first solution. The first solution uses a technique known as *lazy evaluation* or *delayed evaluation*. The trick is to evaluate the subscript operator in two steps. In the first step, we construct an "artificial object" of a new class *hash_table_ref* in which we simply collect information about the call of the subscript operator. In the second step, we do the actual access to the hash table. The key point is that the intermediate object *hash_table_ref* gives us additional influence on the evaluation order. Hidden objects like *hash_table_ref*s are often called *proxys* or *surrogates* [**?**].

A *hash_table_ref* is a pair consisting of a pointer to a *fat_hash_table_2* and a key k in K. The subscript operator returns a *hash_table_ref* consisting of the *this*-pointer and the key k; there is no need for a special subscript operator for const-objects. The constructor for *hash_table_refs* simply stores the pair. There are two member functions. The conversion operator to type I realizes a read access and simply calls the *inf*-function of the hash table. The assignment operator takes the right hand side of an assigment (a value of type I) and assigns it using the *set_inf*-function of the hash table. Observe that the assignment operator expects a value of type I. This implies that the right hand side of the assignment must be evaluated completely before the left hand side is evaluated.

⟨*public members of fat hash tables, version 2*⟩+≡

```
hash_array_ref<I,K> operator[](const K& k)
{ return hash_array_ref<I,K>(this,k); }
```

	fat hash tables 2				fat hash tables 2 with ops			
s	β	I	L	T	β	I	L	T
1.6	3.249	2.87	6.46	9.33	3.249	3.09	4.95	8.04
1.8	2.08	2.28	6.71	8.99	2.08	2.39	5.27	7.66
2	1.623	2.78	5.51	8.29	1.623	2.69	4.06	6.75
2.2	1.506	2.61	5.52	8.13	1.506	2.59	4.14	6.73
2.4	1.506	2.56	5.55	8.11	1.506	2.57	4.08	6.65
1.6	3.249	1.92	3.41	5.33	3.249	2.09	3.5	5.59
1.8	2.08	1.59	3.62	5.21	2.08	1.63	3.69	5.32
2	1.623	1.9	2.73	4.63	1.623	1.82	2.84	4.66
2.2	1.506	1.72	2.78	4.5	1.506	1.77	2.85	4.62
2.4	1.506	1.74	2.77	4.51	1.506	1.73	2.86	4.59

Table 5: The running times of the second variant of fat hash tables without and with operators for different space overheads s. We assume that a key and an information require the space of $c = 2$ pointers. The first column shows the desired mean overhead in space. For each of the four implementations, we show the value of β_{max}, the time for n insertions, the time for $4n$ lookups for $n = 10^6$, and the total time.

⟨*class hash table ref*⟩≡

```
template <class K, class I> class hash_array_ref{
fat_hash_array_2<K,I>* ptr;
K                      k;
 public:
 hash_array_ref(fat_hash_array_2<K,I>* p, const K& key) : ptr(p), k(key) {}
 operator I(){ return ptr->inf(k); }
 I operator=(const I& i) { ptr->set_inf(k,i); return i; }
};
```

Table **??** compares the running times of the second variant of fat hash tables without and with operator syntax. We infer from the table that the solution has no overhead in running time. On the SPARC, the operator version is even significantly faster. I have no idea, why this is the case. After all, after optimizing the intermediate object of type *hash_table_ref* away, a statement of the form `H[x] = E` essentially amounts to the statement `H.set_inf(x,E)`. TODO: have a look at the assembler code.

We come to the second solution. Consider a statement involving two accesses to a hash table, say with keys x and y. We can view the execution of the statement as consisting of five parts.

(1) prelude;
(2) access to $H[x]$;
(3) interlude;
(4) access to $H[y]$;
(5) postlude;

We call an access *critical* if it causes a rehash. If the second access is non-critical, both accesses return references into the same table, and there is no problem. A problem may occur, if the second access is critical. In this situation, the rehash caused by the second access moves the object associated with x to a new table. If the postlude changes the object (reading is OK) associated with x, the change will affect an entry in the old table and not in the new. If the postlude does not change the object associated with x, the object associated with x in the new table and in the old table has the same value. In either case we could repair the situation, if we would carry out the following assignment after the statement is completed.

(6) move the obejct with key x from the old table to the new table; TODO: TeX

If the postlude wrote the object associated with x, the assignment propagates the change to the new table. If the postlude did not change the object, the assignment is a null operation[8]. The difficulty is that we cannot insert this assignment after the postlude. *We can however insert it at the beginning of the access following the critical access*. The details are as follows.

We maintain a variable *old_key* which always contains the key of the last non-critical access, i.e., a non-critical access sets *old_key* to its argument and a critical access does not change *old_key*. A rehash creates a new table and moves all entries from the old table to the new table. It does not (!!!) delete the old table. This task is left to the access following a critical access.

More precisely, any access checks first whether an old table exists or not. If an old table exists, the access is an access immediately following a critical access. At this point, *old_key* is still the key of the access preceding the critical access. We now perform accesses with *old_key* in both tables and move the object associated with *old_key* from the old table to the new table. Having moved the object, the old table is deleted. In this way, only an access following a critical access does additional work and it does exactly the work required to repair the error introduced by the rehash. We summarize the solution in the following pseudo-code:

// access with key k;
if (old table exists)
{ move the entry *old_key* from the old table to the new and delete the old table; }
if (present table is full)
{ create a new table and move all objects to the new table; }
else
{ *old_key* = k; }
return reference to the object associated with k;

Aren't we never repairing the error, if the critial access is the last access to the hash table? That's true, but a hash table that is not accessed anymore, may contain anything.

TODO: give the complete implementation. Can be copied from LEDA.

A Hash Function for Strings: We describe a hash function for strings. Our class H_2 comes handy. We view a string as a number in base 256^4 (four characters are combined in a single digit) and .... A small problem arises from the fact that strings have no

[8] Observe that the object exists. If the second access causes a rehash, H cannot be a const-object and hence even the use of the subscript operator in a read access will cause an insertion. Thus there is an actual object associated with x (and not just the default value) and hence it is safe to assign to this object.

maximal length. The solution is simple. We start with a tentative value of *lmax*, say 4, and increase it as needed.

TODO: the implementation cheats as it does not make a reduction modulo a prime. So the theory developed in the lecture does not apply.

⟨*Hash for strings*⟩≡
```
#include <LEDA/array.h>
class StringHashFunc{
int lmax;
array<int> coeff;
public:
StringHashFunc()
{ lmax = 4;
  coeff = array<int> (4);
  for (int i = 0; i < lmax; i++) coeff[i] = rand_int();
}
int eval(const string& x)
{ int l = x.length();
  if ( l > lmax)
  { int lold = lmax;
    while ( l > lmax) lmax *= 2;
    array<int> oldcoeff = coeff;
    coeff = array<int> (lmax);
    for (int i = 0; i < lold; i++)    coeff[i] = oldcoeff[i];
    for (int i = lold; i < lmax; i++) coeff[i] = rand_int();
  };
  int val = 0;
  for (int i = 0; i < l; i++) val += coeff[i]*x[i];
  return val;
}
};
StringHashFunc  string_hash_func;
int Hash(const string& x)
{ return string_hash_func.eval(x); }
```

0.1.7 The Abstract Data Type Sparse Array

Figure 3 shows the specification of an abstract data type *h_array*. The data type can be implemented by any one of the hashing schemes discussed in this chapter. (Question to ourselves: should we move the definition of the abstract data type to the beginning of the chapter?)

0.1.8 Perfect Hashing

This section is to be rewritten. In particular, the results of [DKM$^+$94] must be incorporated and the material in the section must be rearranged. The part on program size should either be deleted or moved to the end of the section. In this print-out, I have suppressed the parts on program size.

1. Definition

An instance A of the parameterized data type *h_array*<I, E> (hashing array) is an injective mapping from a type I, called the index type of A, to the set of variables of arbitrary type E, called the element type of A. A hash function `int Hash(const I&)` and the operators `==` and `!=` must be defined for the index type I. The equality operator must define an equivalence relation on I, the inequality operator must define the corresponding inequality, and the hash function must be compatible with the equality operator, i.e., if `x == y` then `Hash(x) == Hash(y)`. We use $A(i)$ to denote the variable indexed by i and we use $dom(A)$ to denote the set of "used indices". This set is empty at the time of creation and is modified by array accesses. Each hashing array has an associated default value *xdef*. The variable $A(i)$ has value *xdef* for all $i \notin dom(A)$.

2. Creation

h_array<I, E> A;	creates an injective function a from I to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* stays undefined) and $dom(A)$ to the empty set, and initializes A with a.
h_array<I, E> $A(E\ x)$;	creates an injective function a from I to the set of unused variables of type E, sets *xdef* to x and $dom(A)$ to the empty set, and initializes A with a.

3. Operations

E&	$A[I\ i]$	returns the variable $A(i)$. Depending on the implementation, the use of the subscript operator may be restricted.
bool	A.defined($I\ i$)	returns true if $i \in dom(A)$ and false otherwise.
void	A.undefine($I\ i$)	removes i from $dom(A)$ and sets $A(i)$ to *xdef*.
void	A.clear( )	makes $dom(A)$ empty.
int	A.size( )	returns $\|dom(A)\|$.

Iteration

forall_defined(i, A) { "the elements from $dom(A)$ are successively assigned to i" }

Figure 3: The manual page for a parametrized data type *h_array*. The data type can be implemented by any one of the hashing schemes discussed in this chapter. Exercise 8 discusses a hash function and a equality predicate that are not compatible.

Our constructive upper bounds are based on a detailed investigation into the *division method* for hashing. More precisely, we consider hash functions $h_k : x \mapsto ((kx) \bmod N) \bmod m$ where k, $1 \leq k < N$, is a multiplier, N is the size of the universe and m is the size of the hash table. The following lemma is crucial.

Lemma 3 *Let N be a prime, let $S \subseteq [0 \,..\, N-1]$, $|S| = n$. For every k, $1 \le k < N$, and i, $0 \le i < m$, let $b_i^k = |\{x \in S\,;\, ((kx) \bmod N) \bmod m = i\}|$ be the number of elements mapped to bucket i by h_k.*

a) *There is a k such that $\sum_{i=0}^{m-1} (b_i^k)^2 \le n + 2n(n-1)/m$. Moreover, such a k can be found in time $O(n \cdot N)$.*

b) *For any ϵ, $0 < \epsilon \le 1$, at least a fraction ϵ of all k's satisfy $\sum_{i=0}^{m-1} (b_i^k)^2 \le n + (2+\epsilon)n(n-1)/m$. Such a k can be found in randomized time $O(n/\epsilon)$.*

Proof: We have

$$
\begin{aligned}
&\sum_{k=1}^{N-1} \left(\sum_{i=0}^{m-1} (b_i^k)^2 - n \right) \\
&= \sum_{k=1}^{N-1} \left(\sum_{i=0}^{m-1} |\{x \in S;\ h_k(x) = i\}|^2 - n \right) \\
&= \sum_{k=1}^{N-1} \sum_{i=0}^{m-1} |\{(x,y);\ x, y \in S, x \neq y, h_k(x) = h_k(y) = i\}| \\
&= \sum_{\substack{(x,y) \in S^2 \\ x \neq y}} |\{k;\ h_k(x) = h_k(y)\}|\ .
\end{aligned}
$$

Next note that $h_k(x) = h_k(y) \iff \big[(kx) \bmod N - (ky) \bmod N\big] \bmod m = 0$. We

Untersch.! are asking for the number of solutions k with $1 \le k < N$. Wir betrachten nun ein festes x und y und die Funktion $g(k) = (kx) \bmod N - (ky) \bmod N$. Wir müssen die Anzahl der k mit $1 \le k < N$ und $g(k) \bmod m = 0$ zählen. Beachten Sie zuerst, daß $-N + 2 \le g(k) \le N - 1$. Somit können wir die obige Summe umschreiben zu

$$
\sum_{i=1}^{\lfloor (N-1)/m \rfloor} \sum_{\substack{(x,y) \in S^2 \\ x \neq y}} |\{k;\ g(k) = \pm i \cdot m\}|.
$$

Wir zeigen, daß $|\{k;\ g(k) = \pm i \cdot m\}| \le 2$ ist. Seien $k_1, k_2 \in \{1, \ldots, N-1\}$ so, daß $g(k_1) = a \cdot g(k_2)$, wobei $a \in \{-1, +1\}$. Dann ist

$$
(k_1 x) \bmod N - (k_1 y) \bmod N = a \cdot ((k_2 x) \bmod N - (k_2 y) \bmod N)
$$

und daher

$$
(k_1 - a k_2)(x - y) \bmod N = 0.
$$

Weil $x \neq y$ und N prim (d.h. Primzahl) ist, folgt daraus, daß $(k_1 - a k_2) \bmod N = 0$ und daher $k_1 = k_2$ oder $k_1 = N - k_2$. Deshalb gilt

$$
\begin{aligned}
\sum_{k=1}^{N-1} \left(\sum_{i=0}^{m-1} (b_i^k)^2 - n \right) &\le \sum_{i=0}^{\lfloor (N-1/m) \rfloor} \sum_{\substack{(x,y) \in S^2 \\ x \neq y}} 2 \\
&\le 2n(n-1)(N-1)/m.
\end{aligned}
$$

Thus there is at least one k such that

$$\sum_{i=0}^{m-1}(b_i^k)^2 \le n + 2n(n-1)/m.$$

Finally note that k can be found b exhaustive search in time $O(n \cdot N)$.

b) In part a) we have shown that $\sum_{k=1}^{N-1}(\sum_{i=0}^{m-1}(b_i^k)^2 - n) \le 2n(n-1)(N-1)/m$. Aus $\sum_{i=0}^{m-1}(b_i^k)^2 - n \ge 0$ für alle k folgt, daß $\sum_{i=0}^{m-1}(b_i^k)^2 - n \le (2+\epsilon)n(n-1)/m$ ist für mindestens den Bruchteil $\epsilon/4$ der k zwischen 1 und $N-1$. Andernfalls wäre nämlich $\sum_{i=0}^{m-1}(b_i^k)^2 - n > (2+\epsilon)n(n-1)/m$ für mindestens $(1-\epsilon/4)(N-1)$ der k und daher $\sum_{k=1}^{N-1}(\sum_{i=0}^{m-1}(b_i^k)^2 - n) > (1-\epsilon/4)(N-1)(2+\epsilon)n(n-1)/m \ge 2n(n-1)(N-1)/m$. Also ist die mittlere Anzahl der benötigten Versuche, um ein k mit $\sum_{i=0}^{m-1}(b_i^k)^2 - n \le (2+\epsilon)n(n-1)/m$ zu finden, höchstens $4/\epsilon$. Testing a particular k for that property takes time $O(n)$.

∎

We will make use of Lemma 1 in two particular cases: $m = n$ and $m \approx n^2$.

Corollary 13 *Let N be a prime and let $S \subseteq [0 .. N-1]$, $|S| = n$. Let b_i^k be defined as in Lemma 1 und* $0 < \epsilon \le 1$.

a) *If $n = m$ then a k satisfying $\sum_{i=0}^{m-1}(b_i^k)^2 < 3n$ can be found deterministically in time $O(n \cdot N)$ und ein k mit $\sum_{i=0}^{m-1}(b_i^k)^2 < (3+\epsilon)n$ in randomisierter Zeit $O(n/\epsilon)$ gefunden werden.*

b) *Wenn $m = n(n-1)+1$ bzw. $m = \lfloor(1+\epsilon)n(n-1)+1\rfloor$, dann kann ein k, so daß $x \mapsto ((kx) \bmod N) \bmod m$ auf S injektiv ist, in deterministischer Zeit $O(n \cdot N)$ bzw. randomisierter Zeit $O(n/\epsilon)$ bestimmt werden.*

Proof: a) follows immediately from Lemma 1 by substituting $m = n$. For part b) we observe first that substituting $m = n(n-1)+1$ ($m = \lfloor(1+\epsilon)n(n-1)+1\rfloor$ respectively) into Lemma 1a) (1b) resp.) shows the existence of a k, $1 \le k < N$, such !! that $\sum_i (b_i^k)^2 < n+2$. Next note that $b_i^k \in \mathbb{N}_0$ for all i and $\sum_i b_i^k = n$. Thus $b_i^k \ge 2$ for some i implies $\sum_i (b_i^k)^2 \ge n+2$, contradiction. We conclude that $b_i^k \le 1$ for all i, i.e., the hash function induced by k operates injectively on S.

∎

Corollary 1 can be interpreted as follows. If $m > n(n-1)$ then the general division method directly provides us with a perfect hash function. Of course, nobody wants to work with a load factor as small as $1/n$. However, part a) of Corollary 1 suggests a method of improving upon the load factor by using a two-step hashing function. As a first step we partition set S into n subsets S_i, $0 \le i < n$, such that $\sum |S_i|^2 < 3n$ as described in part a) of corollary 1. As a second step we apply part b) to every subset. The details are spelled out in

Theorem 14 *Let N be a prime and let $S \subseteq [0 .. N-1]$, $|S| = n$. und* $0 < \epsilon \le 1$.

a) *A perfect hash function $h : S \mapsto [0 .. m-1]$, $m = 3n$, with $O(1)$ evaluation time and $O(n \log N)$ program size can be constructed in time $O(n \cdot N)$.*

b) *Eine perfekte Hashfunktion $h : S \mapsto [0..m-1]$, $m = \lfloor(3+\epsilon)n\rfloor$, mit Auswertungszeit $O(1)$ und Programmgröße $O(n \log N)$ kann in randomisierter Zeit $O(n/\epsilon)$ konstruiert werden.*

Proof: a) By Corollary 1a) there is a k, $1 \leq k < N$, such that $\sum_{i=0}^{n-1} |S_i|^2 < 3n$ where $S_i = \{x \in S;\ ((kx) \bmod N) \bmod n = i\}$. Moreover, k can be found in time $O(n \cdot N)$. Let $b_i = |S_i|$ and let $c_i = b_i(b_i-1)+1$. For every i, $0 \leq i < n$, there is a k_i, $1 \leq k_i < N$, such that $x \mapsto ((k_i x) \bmod N) \bmod c_i$ operates injectively on S_i by Corollary 1b). Moreover, k_i can be determined in time $O(b_i \cdot N)$.

The following program computes an injective function from S into $[0..m-1]$ where

i.D.mehr $m = \sum_{i=0}^{n-1} c_i = n + (\sum_i b_i^2 - \sum_i b_i) < n + 3n - n = 3n$. Dabei ist x die Eingabe

Text neu Dieses Programm benutzt die Konstanten k, N, n und zwei Felder von Konstanten, je eines für die k_i und die Partialsummen der c_i. Die c_i selbst können dann aus dem Feld der Partialsummen bestimmt werden. Jede der Konstanten kann durch $O(\log N)$ Bits spezifiziert werden, und daher ist die Größe des Programms $O(n \log N)$. Die Laufzeit ist $O(1)$. Finally, the time to find the program is bounded by $O(n \cdot N + \sum_i b_i \cdot N) = O(n \cdot N)$.

Bew.neu b) The proof of part b) is very similar to the proof of part a). Sei δ definiert durch $(3+\epsilon) = (1+\delta)(3+\delta)$. Ein k mit $\sum_{i=0}^{n-1} |S_i|^2 < (3+\delta)n$ kann in randomisierter Zeit $O(n/\delta)$ gefunden werden. Auch können die k_i, so daß $x \mapsto ((k_i x) \bmod N) \bmod \lfloor((1+\delta)b_i(b_i-1)+1)\rfloor$ auf S_i injektiv ist, in randomisierter Zeit $\sum_i O(b_i/\delta) = O(n/\delta)$ gefunden werden. Wenn man diese Funktionen wie oben beschrieben kombiniert, erhält man eine injektive Funktion von S nach $[0..m-1]$, wobei $m = \sum_{i=0}^{n-1} \lfloor((1+\delta)b_i(b_i-1)+1)\rfloor \leq (1+\delta)\sum_{i=0}^{n-1} b_i^2 \leq (1+\delta)(3+\delta)n$.

∎

Theorem 9 provides us with very efficient perfect hash functions. After all, evaluating tha hash functions constructed in the proof of theorem 9 requires only two multiplications and four divisions. However, the programs are hard to find, at least deterministically, and they are very large. Let us take a closer look. The method devised in part a) of Theorem 9 requires hash table of size $3n$ and storage space for $2n$ integers in the range $[1..N]$. Thus a total of $5n$ storage locations is needed. The situation is even worse for the hash functions constructed by probabilistic methods. They require a total of $6n$ storage locations, $4n$ for the hash table and $2n$ for the program. We will improve upon both hash functions to $O(n \log n + \log\log N)$ bits.

The key to the improvement is another variant of the division method which we will use to reduce the size of the universe. More precisely, we will show that for every $S \subseteq [0..N-1]$, $|S| = n$, there is $p = O(n^2 \ln N)$ such that $x \mapsto x \bmod p$ operates injectively on S. We can then apply Corollary 1 and Theorem 9 to set $S' = \{x \bmod p;\ x \in S\}$. Since the members of S' have size $O(n^2 \ln N)$ this will reduce the space requirement considerably.

Theorem 15 *Let $S = \{x_1 < x_2 < \cdots < x_n\} \subseteq [0..N-1]$.*

a) *There is a prime $p = O(n^2 \ln N)$ such that $x_i \bmod p \neq x_j \bmod p$ for $i \neq j$.*

b) *A number p satisfying $p = O(n^2 \ln N)$ and $x_i \bmod p \neq x_j \bmod p$ for $i \neq j$ can be found in time $O(n \log n \cdot (\log n + \log\log N))$ by a probabilistic algorithm, and in time $O(n^3 \log n \cdot \log N)$ deterministically.*

Proof: a) let $d_{ij} = x_j - x_i$ for $1 \leq i < j \leq n$. Then $x_i \bmod p \neq x_j \bmod p \iff d_{ij} \neq 0 \bmod p$. Let $D = \prod_{i<j} d_{ij} \leq N^{(n^2)}$. We need to find a bound on the size of the smallest prime which does not divide D.

Fact 1 *Let $m \in \mathbb{N}$. Then m has at most $O(\ln m / \ln\ln m)$ different prime divisors.*

Proof: Let m have q different prime divisors and let $p_1, p_2, p_3, \ldots, p_q$ be the list of primes in increasing order. Then

$$\begin{aligned} m &\geq p_1 \cdot p_2 \cdots p_q \geq q! = e^{\sum_{i=1}^{q} \ln i} \\ &\geq e^{\int_1^q \ln x \, dx} = e^{q \ln(q/e)+1} \geq (q/e)^q. \end{aligned}$$

Hence there is a constant c such that $q \leq c \cdot (\ln m)/(\ln\ln m)$. ∎

We infer from claim 1 that D has at most $c \ln D / \ln\ln D$ different prime divisors. Hence at least half of the $2c \ln D / \ln\ln D$ smallest primes will not divide D and hence not divide any d_{ij}. The prime number theorem gives us a bound on the size of this prime.

Fact 2 *There is $d \in \mathbb{R}$ such that $p_q \leq d \cdot q \ln q$ for all $q \geq 1$; p_q is the q-th smallest prime* i.E.mehr

Proof: Cf. I. Niven/H.S. Zuckermann: “Introduction to the Theory of Numbers”, Volume 2, Theorem 8.2.

∎

We infer from this fact that at least half of the primes

$$p \leq d \cdot \frac{2c \ln D}{\ln\ln D} \cdot \ln \frac{2c \ln D}{\ln\ln D} = O(\ln D) = O(n^2 \ln N)$$

satisfy part a).

b) We proved in part a) that there is a constant a such that at least half of the primes $p \leq a \cdot n^2 \ln N$ will not divide D and hence satisfy a) Furthermore, the prime number i.E.mehr theorem states that there are at least $b \cdot (a \cdot n^2 \ln N)/\ln(a \cdot n^2 \ln N)$ primes less than $a \cdot n^2 \ln N$ for some constant $b > 0$. These observations suggest a probabilistic algorithm for finding a number p (not necessarily prime) satisfying $p \leq a \cdot n^2 \ln N$ and $x_i \bmod p \neq x_j \bmod p$ for $i \neq j$. We select a number $p \leq a \cdot n^2 \ln N$ at random and check whether $x_i \bmod p \neq x_j \bmod p$ for $i \neq j$ in time $O(n \log n)$ by sorting numbers $x_i \bmod p$, $1 \leq i \leq n$. If p does not work then we try again, ...

It remains to derive a bound on the expected number of unsuccessful attempts. Note first that a random number $p \leq a \cdot n^2 \ln N$ is a prime with probability $\Omega(1/\ln(a \cdot n^2 \ln N))$ and that a random prime satisfies a) with probability $\geq 1/2$. Hence a random number $p \leq a \cdot n^2 \ln N$ leads to a mapping $x \mapsto x \bmod p$ which is injective on S with probability $\Omega(1/\ln(a \cdot n^2 \ln N))$; therefore the expected number of unsuccessful attempts is $O(\ln(a \cdot n^2 \ln N))$. Since each attempt costs $O(n \log n)$ time units the total expected cost is $O(n \log n \cdot (\log n + \log\log N))$.

Deterministically, we search through numbers $p = O(n^2 \ln N)$ exhaustively. This will cost at most $O(n^3 \log n \cdot \log N)$ time units.

∎

10,11,12 We end this section by combining Theorems 9 and 10⊗

Theorem 16 *Let $S \subseteq [0 .. N-1]$ with $|S| = n$. There is a program P of size $O(n \log n + \log\log N)$ bits and evaluation time $O(1)$ which computes a perfect hash function from S into $[0 .. m-1]$, $m = 3n$. P can be found in random time $O(n^3 \cdot (\log n + \log\log N))$.*

i.D.anders **Proof:** ⊗Let $S \subseteq [0 .. N-1]$, $|S| = n$. Nach dem Beweis von Satz 10a) haben mindestens die Hälfte der Primzahlen kleiner $a \cdot n^2 \log N$ für eine geeignete Konstante a die Eigenschaft, daß die Funktion $h_1 : x \mapsto x \bmod p$ auf S injektiv ist. Angenommen, wir haben nun eine Primzahl $p = O(n^2 \log N)$, so daß $h_1 : x \mapsto x \bmod p$ auf S injektiv ist. Sei weiter $S_1 = h_1(S) \subseteq [0 .. p-1]$. Nach Korollar 1b) mit $\epsilon = 1$ gibt es ein k, $1 \le k < p$, so daß die Funktion $h_2 : x \mapsto ((kx) \bmod p) \bmod (2n(n-1)+1)$ auf S_1 injektiv ist. k kann in randomisierter Zeit $O(n)$ gefunden werden. Sei $S_2 = h_2(S_1) \subseteq [0 .. 2n(n-1)] \subseteq [0 .. q-1]$, wobei q die kleinste Primzahl $\ge 2n(n-1)+1$ ist. Es gilt $q = O(n^2)$, und daher kann q nach Aufgabe 47 in Zeit $O(n^2 \log n)$ bestimmt werden. Nach Satz 9a) gibt es eine injektive Funktion $h_3 : S_2 \mapsto [0 .. m-1]$, $m = 3n$, mit Auswertungszeit $O(1)$ und Programmgröße $O(n \log q) = O(n \log n)$. Weiterhin kann h_3 in deterministischer Zeit $O(n \cdot q) = O(n^3)$ gefunden werden.

$h = h_3 \circ h_2 \circ h_1$ is the desired hash function. A program for h can be found in time $O(n^3)$ by a probabilistic algorithm. The program has size $O(\log p + \log p + n \log n) = O(n \log n + \log\log N)$ bits. Moreover, it can be evaluated in time $O(1)$.

fehlt i.E. Was haben wir jetzt erreicht?⊗Wenn $p = O(n^2 \log N)$ eine Primzahl ist, so daß $x \mapsto x \bmod p$ auf S injektiv ist, dann können wir die gewünschte Hashfunktion in randomisierter Zeit T mit $T = O(n^3)$ finden. Wenn wir also bis zu T Zeiteinheiten aufwenden, um h_2 und h_3 zu finden, dann haben wir mit Wahrscheinlichkeit mindestens 1/2 Erfolg. Also ist die Wahrscheinlichkeit mindestens $1/4$, daß für eine zufällige Primzahl $p \le a \cdot n^2 \log N$ die Funktion $h_1 : x \mapsto x \bmod p$ auf S injektiv ist und daß wir auch h_2 und h_3 in Zeit T finden. Also finden wir $h = h_3 \circ h_2 \circ h_1$ in randomisierter Zeit $O(n^3 \cdot (\log n + \log\log N))$ nach dem Argument von Satz 10b).

∎

Theorem 11 essentially settles all questions about perfect hashing. The program constructed in the proof of Theorem 11 has almost minimal size, namely $O(n \log n + \log\log N)$ bits instead of $O(n + \log\log N)$ bits, they achieve a load factor $\beta = n/m exceeding 1/3$, they are not too hard to find, and they are quite efficient. More precisely, evaluation requires one division of numbers of length $O(\log N)$, a multiplication and two divisions on numbers of length $O(\log n + \log\log N)$, and two multiplications and four divisions on numbers of length $O(\log n)$.

TODO

Figure 4: Example for extendible hashing

0.1.9 Extendible Hashing

Our treatment of hashing in Sections 3.2.1 to 3.2.4 based on the assumption that main memory is large enough to completely contain the hash table. In this section we discuss the application of hashing to secondary storage. We assume that is divided into buckets (pages) of size b; i.e., each bucket can hold up b elements of universe U.

Again, we start with a hash function $h : U \mapsto \{0, 1\}^k$ where k is some integer. h is assumed to be injective. For d, $0 \le d \le k$, we use h_d to denote the function which maps $x \in U$ onto the first d digits of $h(x)$, i.e., $h_d : U \mapsto \{0, 1\}^d$ and $h_d(x)$ is a prefix of $h(x)$ for all $x \in U$. Let $S \subseteq U$, $|S| = n$, be a subset of U. The *depth* $d(S)$ of S wit i.E.mehr
respect to bucket size b and hash function h is defined as

$$d(S) = \min \left\{ d;\ |\{x \in S;\ h_d(x) = a\}| \le b \text{ for all } a \in \{0, 1\}^d \right\}.$$

In other words, if we use a TRIE to partition $h(S)$ into subsets of size at most b then this TRIE has depth $d(S)$.

Extendible hashing uses a table $T[0 \mathinner{..} 2^{d(S)} - 1]$, called the *directory*, and some number of buckets to store set S. More precisely, the entries of table T are pointers to buckets. If x is an element of S then the bucket containing x can be found as follows.

1. Compute $h_{d(S)}(x)$ and interpret it as the binary representation of an integer, say i, in the range $[0 \mathinner{..} 2^{d(S)} - 1]$.

2. Use this integer index directory T; pointer $T[i]$ points to the bucket containing x.

Example: Let $h(S) = \{0000,\ 0001,\ 0100,\ 1100\}$ and let $b = 2$. Then $d(S) = 2$. Wir erhalten die in Abbildung **??** veranschaulichte Situation.

As one can see from Figure **??**, we allow entries of T to point to the same bucket. However, we require that sharing of bucket must conform to the buddy principle. Das i.E.anders
bedeutet folgendes: Jeder Korb hat eine lokale Tiefe r. Auf einen Korb der lokalen Tiefe r zeigen genau $2^{d(S)-r}$ Katalogeinträge. Ferner gibt es ein $a \in \{0, 1\}^r$, so daß genau die Katalogeinträge $T[aa_1]$ auf B zeigen für $a_1 \in \{0, 1\}^{d(S)-r}$. In Abbildung **??** steht die lokale Tiefe der Körbe in der linken oberen Ecke.

The insertion algorithm for extendible hashing is quite simple. Suppose that we want to insert $x \in U$. We first compute bucket B which should contain x. If B is not full, i.e., B contains less than b keys, then we insert x into B and are done. If B is full, say B contains $x_1, \ldots, x_b$, then additional work is required. Let r be the local depth of page B and let B' be a new page. If $r = d(S)$ then we double the size of directory T, i.e., we increase $d(S)$ by 1, create a directory T' of size $2^{d(S)+1}$ and initialize T'. Note that initializing T' essentially means to make two copies of T and to merge them. At this point we have $r < d(S)$ in either case, i.e., if $r < d(S)$ initially or if $r = d(S)$ and directory size was doubled. We complete the insertion by setting the local depth of B and B' to $r + 1$ and by inserting $x_1, \ldots, x_b, x$ into B or B' whatever is appropriate. Note that this might cause B or B' to overflow (in some rare case). In this case we go through the procedure once more.

TODO

Figure 5: Nach Einfügen von 0010 in die Beispielmenge

Example (continued): Suppose that we want to insert x with $h(x) = 0010$. It will cause the first bucket to overflow. We obtain Figure ??.

The deletion algorithm is also quite simple. Suppose that we want to delete x from S. We first determine bucket B containing x and delete x from B. Let B' be the buddy of B, i.e., if r is the local depth of B and $a \in \{0,1\}^{r-1}$ and $z \in \{0,1\}$ are such that all directories $T[aza_1]$, $a_1 \in \{0,1\}^{d(S)-r}$, point to B then B' is the buddy of B if all entries $T[a\bar{z}a_1]$ and $\bar{z} = 1 - z$, point to B'. Note that B' does not necessarily exist. Deshalb können auch Zeiger auf leere Körbe entstehen. fragefehltIf B' does exist and B and B' contain together at most b keys then we merge B and B' to a single bucket of local depth $r - 1$. In addition, if $r = d(S)$ and B and B' were the only buckets if depth $d(S)$ then we half the size of the directory. This completes the description of the deletion algorithm.

What can we say about the behavior of extendible hashing? What are the relevant quantities?$\otimes$We have already seen that an Access operation requires only two accesses to secondary memory. This is also true of an Insert, except for the case that a bucket overflows. In this case we have to obtain a new bucket, a third access and to distribute the elements of the bucket to split over the new buckets. In rare cases we will also have to double the directory. These remarks show that the time complexity of extendible hashing is very good.

fehlt i.D

The space complexity requires further investigation. The two main questions are: What is the size of the directory? How many buckets are used? First of all, as for all hashing schemes worst case behavior can be very, very bad. So let us discuss expected behavior. The analysis is based on the following assumption. We have $k = \infty$, i.e., hash function h maps U into bit strings of infinite length. Furthermore, $h(x)$ is uniformly distributed in interval $[0,1]$. Note that $h(x)$ an be interpreted as the binary representation of a real number. A discussion of expected behavior is particularly relevant in view of our treatment of universal hashing. Note that the behavior of extendible hashing heavily depends on the hash function in use and that universal hashing teaches us how to choose good hash functions. Unfortunately, a complete analysis of the expected behavior of extendible hashing is quite involved and far beyond the scope of the book. fragefehlt i.EEine vollständige Analyse von erweiterbarem Hashing kann nachgelesen werden in Flajolet/Steyaert (82). Eine approximative und einfachere Analyse von erweiterbarem Hashing steht in A. Yao (80). Es gilt

Theorem 17 **a)** *The expected number of buckets required to store a set of n elements is approximately $n/(b \ln 2)$.*

b) *The expected size of the directory for a set of n elements is approximately $(e/b \ln 2)n^{1+1/b}$.*

We finish this section with a discussion of Theorem 16. Part a) states that the expected number of buckets is $n/(b \ln 2)$ ist; in other words the expected number of elements per bucket is $b \ln 2 \approx 0.69b$. Expected storage utilization is 69%. This result is not surprising. After all, buckets can contain between 0 and b keys. Once a bucket overflows

b	$n = 10^5$	$n = 10^6$	$n = 10^8$	$n = 10^{10}$
2	6.2×10^7	1.96×10^8	1.96×10^{11}	1.96×10^{14}
10	1.2×10^5	1.5×10^6	2.4×10^8	3.9×10^{10}
50	9.8×10^3	1.0×10^6	1.1×10^8	1.2×10^{10}
100	4.4×10^3	4.5×10^4	4.7×10^6	4.9×10^8

Table 6: Mittlere Kataloggrößen bei erweiterbarem Hashing

it is split into two buckets. The size of the two parts is a random variable; however, it is very likely that each of the two buckets receives about 50% of the elements. We should thus expect that the expected number of elements per bucket is somewhere between 0 and b with a small inclination towards b. Eine sehr ähnliche Situation wird in Abschnitt III.5.3.4 untersucht. fragefehlt i.E.

Part b) is more surprising. Expected directory size is non-linear in the number of elements stored. The table aus Abbildung **??** lists expected directory size for various choices of b and n.

We can see from Figure **??** that the non-linear growth is clearly perceptible for small b, say $b \approx 10$, and that directory size exceeds the size of the file even for moderate n, say $n = 10^6$ bei $b = 50$. fragefehlt i.E.If b is larger, say $b \approx 100$, then the non-linearity is hardly noticeable for practical values of n, say $n \leq 10^{10}$. Moreover, the size of the directory will be only about 5% of the size fo the file.

Why does directory size grow non-linearly in the size of the file? In the case $b = 1$ this is not too hard to see and follows from the birthday paradox. If b is 1 then the directory size doubles whenever two elements of S hash to the same entry of the directory (have the same birthday). If directory size is m (the year has m days) and the size of S is n then the probability $p_{n,m}$ that two elements of S hash the same entry of the directory is

$$\begin{aligned} p_{n,m} &= m \cdot (m-1) \cdots (m-n+1)/m^n \\ &= \prod_{i=0}^{n-1} (1 - i/m) = e^{\sum_{i=0}^{n-1} \ln(1-i/m)} \\ &\approx e^{-\sum_{i=0}^{n-1} i/m} \approx e^{-n^2/m}. \end{aligned}$$

Thus $p_{n,m} \leq 1/e$ if n exceeds $\sqrt{m}$. In other words, a directory of size m suffices for a set of $n = \sqrt{m}$ elements with probability less than $1/e$. So directory size must grow $1 - 1/e$?
non-linearly in file size and should in fact be quadratic in file size.

Bibliographic Notes

Carter/Wegman [CW79, WC81] universal hashing

Exercises:

1. Let L be defined as in the proof of Theorem 4. Show that $\text{prob}(\max_j l_j \geq cL) \leq 2^{-(c-1)L}$ for any $c \geq 1$.

Solution: Let p_j be as defined in the proof of Theorem 4. Then

$$\mathrm{prob}(\max_j l_j \geq cL) = p_{cL} = \frac{p_{cL}}{p_{cL-1}} \cdot \frac{p_{cL-1}}{p_{cL-2}} \cdots \frac{p_{L+1}}{p_L} \cdot p_l \leq 2^{-(c-1)L}, \ ,$$

since $p_{j+1}/p_j \leq 2$ as shown in the proof of Theorem 4.

2. Show that the set of all functions from $[0..N-1]$ to $[0..m-1]$ is 1-universal.

   Solution: There are m^N functions from $[0..N-1]$ to $[0..m-1]$ since there are m choices for each of the N function values and there are m^{N-1} functions h with $h(x) = h(y)$, since such a function is specified by giving the values for all elements distinct from y. 1-universality follows.

3. Show that there is no c-universal class for $c < 1 - m/N$.

   Solution: Let H be a c-universal class. We count the number of triples (x, y, h) with $h \in H$, $x, y \in U$ and $x \neq y$. For a fixed $h \in H$ let N_j be the number of elements in U that are mapped to j by h. Then h contributes $\sum_{0 \leq j < m} N_j(N_j - 1)$ triples. This number is minimized if $N_j = N/m$ for all j. We conclude that the number of triples is at least $|H|m(N/m)(N/m-1)$. The number of pairs (x, y) with $x \neq y$ is equal to $N(N-1)$. Thus there is some pair (x, y) for which at least $|H|m(N/m)(N/m-1)/(N(N-1)) = (|H|/m)m^2(N/m)(N/m-1)/(N(N-1))$ functions lead to a collision. Thus $c \geq m^2(N/m)(N/m-1)/(N(N-1) = (N-m)/(N-1) \geq (1-m/N$.

4. Let H_1 be as defined in the text and let $m > 2^i$. Show that the class $\left\{x \mapsto h(x) \bmod 2^i \ ; x \in H_1\right\}$ is 2-universal.

   Solution: TODO

5. Add a function *delete*(*const K*& *key*) that deletes the entry with key k to the slim table version of hashing with chaining.

   Solution: Simply delete the node with key k.

6. Add a function *delete*(*const K*& *key*) that deletes the entry with key k to one of the fat table versions of hashing with chaining. The implentations should have expected running time $O(1 + \beta)$.

   Solution: Let v be the node with key k. If v is not the first node in its list, we can simply delete the node. In version two this will leave a hole in the table of nodes. If v is the first node and the list has more than one node, we move the contents of the second node into the first node and delete the second node. If v is the first and only nodes of its list, we simply change the *next*-pointer of v to *nil*.

7. Add a function *delete*(*const K*& *key*) that deletes the entry with key k to our implementation of hashing with open addressing. What difficulties arise?

   Solution: TODO

8. Consider a class *point*. Two points are equal if they agree in their coordinates. The hash value of a point is its address in memory. Is it OK to use points as indices into hash arrays? Define a hash function which is compatible with the equality predicate.

Solution: The hash function is not compatible with the equality predicate because we might have two objects of class point with the same coordinates. The hash function would map them to distinct integers. Any hash function that only depends on the coordinates is compatible with the equality operator.

9. Show how to implement the iterator over all defined indices in a hash table.

   Solution: TODO

10. Do a theoretical analysis of our different hashing schemes for large space overhead, say $s = 2.4$. Does your predicion agree with the measurements in Table 4?

    Solution: TODO

Bibliography

[ADK+93] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *The Computer Journal*, 36(8):756–762, 1993.

[AKP99] F. Abolhassan, J. Keller, and W.J. Paul. On the cost-effectiveness of PRAMs. *Acta Informatica*, 36, 1999.

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, april 1979.

[DKM+94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761, 1994.

[Knu81] D.E. Knuth. *The Art of Computer Programming (Volume III): Sorting and Searching*. Addison-Wesley, 1981.

[LYD71] V.Y. Lum, P.S.T. Yuen, and M. Dodd. Key to address transformation techniques. *CACM*, 14:228–239, 1971.

[MV84] K. Mehlhorn and U. Vishkin. Granularity of memory in parallel computation. *Acta Informatica*, 21:339–374, 1984.

[Ran91] A.G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, june 1991.

[SF96] R Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Publishing Company, 1996.

[WC81] M.N. Wegman and J.L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, june 1981.