Chapter 3. Sets

Many algorithms manipulate sets. We consider some typical examples. A compiler uses a symboltable to keep track of the identifiers used in the program. in einem Programm vorkommenden Bezeichnungen (auch Schlüssel, Namen oder Identifier) eingetragen werden. An identifier together with relevant information (e.g., type, scope, address, ...) is entered into the symboltable when its declaration is processed by the compiler. Applied occurrences of the same identifier refer to the defining occurrence; the compiler has to access the symboltable and to look up the relevant information. More abstractly, a compiler deals with a set ST of objects consisting of a name (key) and associated information. Two kinds of operations are performed on the set ST. New objects (x, I) are added to set ST, i.e., ST is replaced by $ST \cup \{(x, I)\}$, and the objects are accessed via their key, i.e., given an identifier x we want to find I such that $(x, I) \in ST$. In most programming languages identifiers are strings of letters and digits of some bounded length, say at most length 6. Then the number of possible identifiers is $(26 + 10)^6 \approx 2 \cdot 10^9$; in every program only a small subset of the set of all possible identifiers is used. Set ST is small compared to the very large universe of all possible identifiers.

A second example is the index of authors of a library. The name of the author is the key, the associated information are titles of books, their locations on the shelf, ... This example exhibits all characteristics of the problem above; however, at least one additional operating is performed on the set: print a library catalogue in ascending lexicographic order.

Let us consider a third example: the set of accounts in a bank. The account number is the key, the associated information is the holder of the account, the balance, etc. Typical operations are Access, Insert, Delete, List in ascending order; i.e., the same operations as in our previous example. However, there is one major difference to the previous example: the size of the universe. A bank might have $5 \cdot 10^5$ different accounts and use account numbers with 6 decimal digits, i.e., there are 10^6 possible account numbers and half of them are actually used. The universe of possible keys and the set of keys used are of about the same size. Graph algorithms (Chapter 4) also provide us with plenty of examples of this phenomenon.

We treat the case of drastic size difference between the universe and the set stored first (Sections 3.1 to 3.7) and discuss the case of equal size later (Section 3.8). The major difference between the solutions is based on the fact that that accessing an information via a key is a difficult problem in the first case, but can be made trivial in the second case by use of an array.

Sets consist of objects. An object is typically a pair consisting of key (name) and information associated with the key. As in chapter 2we will concentrate on the key part and identify object and key.

$S \subseteq U$ be the subset of a universe U. We consider the following operations:

Name of operation	*Effect*
Access(x, S)	**if** $x \in S$

	then the information associated with x **else** a message, that x is no element of S **fi**;
Member(x, S)	**if** $x \in S$ **then** "Yes" **else** "No" **fi**;
Insert(x, S)	$S \leftarrow S \cup \{x\}$;
Delete(x, S)	$S \leftarrow S - \{x\}$.

Operations Insert and Delete change set S, i.e., the former version of set S is destroyed by the operations. Operation names Access and Member are used interchangeably; we will always use Member when the particular application only requires yes/no answers.

For the following operations we assume in addition, that $(U, \leq)$ is linearly ordered.

Ord(k, S)	the k-th element of the linear arrangement of set S
List(S)	a list of the elements of set S in ascending order.

Additional operations will be considered in later sections.

We already know one data structure which supports all operations mentioned above, the linear list of Section 1.3.2. A set $S = \{x_1, \ldots, x_n\}$ is represented as a list, whose i-th element is x_i. All operations above can be realized by a single traversal of the list and hence take time $O(n)$ in the worst case.

In this chapter we study data structures which are considerable more efficient than linear lists. These data structures can be divided in two large groups: comparison based methods (Sections 3.3 to Section 3.7) and methods based on representation (Section 3.1 and 3.2). The former methods only use comparisons $(\leq, <, =, >, \geq)$ between elements to gain information, the latter methods use the representation of keys as strings over some alphabet to gain information. Search trees and hashing are typical representatives of the two classes.

III.1.1. TRIES

3.1. Digital Search Trees

One of the most simplest ways of structuring a file is to use the digital representation of its elements; e.g., we may represent $S = \{121, 102, 211, 120, 210, 212\}$ by the following TRIE

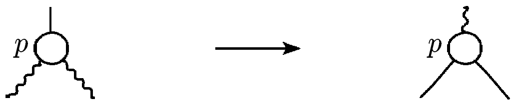

Figure 1. TRIE für $S = \{121, 102, 211, 120, 210, 212\}$

The general situation is as follows: the universe U consists of all strings of length l over some alphabet of say k elements, i.e., $U = \{0, \ldots, k-1\}^l$. A set $S \subseteq U$ is represented as the k-ary tree consisting of all prefixes of elements of S. An implementation which immediately comes to mind is the use of an array of length k for every internal node of the tree. (We will see a different implementation in Section 6.3). in particular, if the reverse of all elements of S is stored (in our example this would be set $\{121, 201, 112, 021, 012, 212\}$) then the following Program 1 will realize operation Access(x)

```
v ←root;
y ← x;
do l times (i, y) ← (y mod k, y div k);
           v ← i-th son of v
od;
if x = CONTENT[v] then "Yes" else "No" fi.
```

Program 1

This program takes time $O(l) = O(\log_k N)$ where $N = |U|$. Unfortunately, the space requirement of a TRIE as described above can be horrendous: $O(n \cdot l \cdot k)$. For each element of set S, $|S| = n$, we might have to store an entire path of l nodes, all of which have degree one and use up space $O(k)$. There is a very simple method of reducing storage requirement to $O(n \cdot k)$. We only store internal nodes which are at least binary. Since a TRIE for a set S of size N has n leaves there will be at most $n-1$ internal nodes of degree 2 or more. Chains of internal nodes of degree 1 are replaced by a single number, the number of nodes in the chain. In our example we obtain:

Here internal nodes are drawn as arrays of length 3. On the pointers from fathers to sons the numbers indicate the increase in depth, i.e., 1+the length of the eliminated chain of nodes of degree one. In our example the 2 on the pointer from the root to the son with name b indicates that after branching on the first digit in the root we have to branch on the third (since $3 = 2 + 1$) digit in the son.

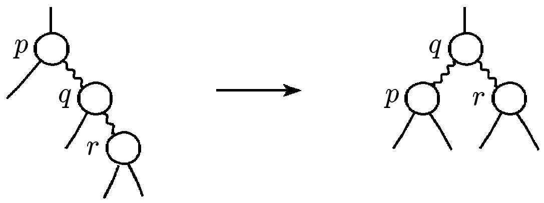

Figure 2. TRIE ohne unäre Knoten

The algorithms for Access, Insert and Delete become slightly more complicated for compressed TRIES but still run in time $O(l)$.

Theorem 1. *A compressed TRIE supports operations Access, Insert and Delete with time bound* $O(\log_k N)$ *where* N *is the size of the universe and* k *is the branching factor of the TRIE. A set* S *of* n *elements requires space* $O(k \cdot n)$.

Proof: by the discussion above. ∎

Note that TRIES exhibit an interesting time space trade-off. Choosing k large will make TRIES faster but more space-consuming, choosing k small will make TRIES slower but less space-consuming. For static sets, i.e., only operation Access is supported, we describe in 3.1.2 a further compression technique below.

A disturbing fact about TRIES is that the worst case running time depends on the size of the universe rather than on the size of the set stored. We will next show that the average case behavior of TRIES is $O(\log_k n)$; i.e., average access time is logarithmic in the size of the set stored. We use the following *probability assumption*:

Every subset $S \subseteq U$, $|S| = n$, is equally likely.

Theorem 2. *Let* $E(d_n)$ *be the expected depth of a* k*-ary compressed TRIE for a set of* n *elements. Then*

$$E(d_n) \leq 2\log_k n + O(1).$$

Proof: Let q_d be the probability that the TRIE has depth D or more. Then $E(d_n) = \sum_{d \geq 1} q_d$. Let $S = \{x_1, \ldots, x_n\} \subseteq U = \{0, \ldots, k-1\}^l$ be a subset of U of size n. The compressed TRIE for S has depth less than d if the function $trunc_{d-1}$ which maps an element of U into its first $d-1$ digits is injective on S. Furthermore, $trunc_{d-1}$ is an injective mapping on set S iff $\{trunc_{d-1}(x);\ x \in S\}$ is a subset of size n of $\{0, \ldots, k-1\}^{d-1}$. Thus there are exactly $\binom{k^{d-1}}{n} k^{(l-(d-1))n}$ subsets S of size n of U such that $trunc_{d-1}$ is injective on S. Hence

$$\begin{aligned} q_d &\le 1 - \frac{\binom{k^{d-1}}{n} k^{(l-(d-1))n}}{\binom{k^l}{n}} \\ &\le 1 - \frac{k^{d-1} \cdot (k^{d-1}-1) \cdots (k^{d-1}-(n-1)) \cdot k^{(l-(d-1))n}}{(k^l)^n} \\ &= 1 - \prod_{i=0}^{n-1} (1 - \frac{i}{k^{d-1}}). \end{aligned}$$

Next note that , daß für $n < k^{d-1}$ gilt

$$\begin{aligned} \prod_{i=0}^{n-1} (1 - \frac{i}{k^{d-1}}) &= e^{\sum_{i=0}^{n-1} \ln(1-i/k^{d-1})} \\ &\ge e^{\int_0^n \ln(1-x/k^{d-1})dx} \\ &\ge e^{-n^2/k^{d-1}}. \end{aligned}$$

The first inequality follows from the fact that $f(x) = \ln(1-x/k^{d-1})$ is a decreasing function in x and hence $f(i) \ge \int_i^{i+1} f(x)\,dx$. The last inequality can be seen by evaluating the integral using the substitution $x = k^{d-1} \cdot (1-t)$ unter Benutzung
i.D.mehr von $\ln(1+x) \le x$ (s. Anhang). Hence

$$\begin{aligned} q_d &\le 1 - e^{-n^2/k^{d-1}} \\ &\le n^2/k^{d-1}, \end{aligned}$$

i.D.mehr since $e^x - 1 \ge x$ and hence $1 - e^x \le -x$, zumindest wenn $n < k^{d-1}$. Wegen $q_d \le 1$ gilt die Abschätzung aber sogar für alle n. Let $c = 2\lceil \log_k n \rceil$. Then

$$\begin{aligned} E(d_n) &= \sum_{q=1}^{c} q_d + \sum_{d \ge c+1} q_d \\ &\le c + \sum_{d \ge c} n^2/k^d \\ &\le 2\lceil \log_k n \rceil + (n^2/k^c) \sum_{d \ge 0} k^{-d} \\ &\le 2\lceil \log_k n \rceil + \frac{1}{1-1/k}\,. \end{aligned}$$

∎

Theorem 2 shows that the expected depth of a random TRIE is at most $2\log_k n + O(1)$ and so the expected Access-, Insert-, and Delete-time is $O((\log n)/(\log k))$.

Space requirement is $O(n \cdot k)$. Parameter k, the basis of the digital representation, allows us to trade between space and time.

3.1.2. STATIC TRIES or Compressing Sparse Tables

In this section we show that the space requirement can be drastically reduced for static TRIES without an increase in access time. Static TRIES only support operation ACCESS. The reduction is done in two steps. As a first step we describe a general overlay technique for compressing large sparse tables. This technique is not only applicable to TRIES but also to other areas where large sparse tables arise, e.g., parsing tables. The first step reduces space requirement to $O(n \log\log n)$. As a second step we reduce the space requirement further to $O(n)$ by packing several numbers into one storage location.

A TRIE for a set of n elements requires space $O(k \cdot n)$. There are at most $n-1$ nodes each of which requires storage for an array of k elements. At most $2n-2$ of the $(n-1)k$ possible outgoing pointers will be non-nil, because the TRIE has n leaves and at most $n-2$ internal nodes unequal the root. This suggests to use an overlay technique to reduce storage requirement (An alternative approach is discussed in Section 2.3).

Hinw.Abb. In our example, we may use an array of length 10

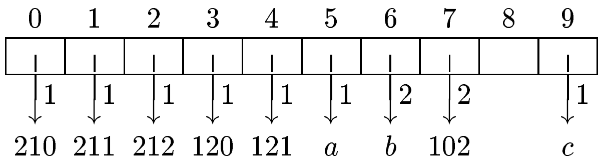

Figure 3. Komprimierung des TRIES aus III.1.1. in ein großes Feld

to store all four arrays of length 3. The root node starts at location 4, node a starts at 7, node b starts at 0 and c starts at location 3. This information is stored an an additional table (cf. Fig. 4).

a	b	c	Wurzel
7	0	3	4

Figure 4. Eintrittspunkte in das große Feld

Suppose now that we search for 121. We follow the 1-st pointer out of the root node, i.e., the pointer in location $4+1$ of the large array. This leads us to node a. We follow the 2-nd pointer out of a, i.e., the pointer in location $7+2$ of the big array which leads us to node c, ... It is also instructive to perform an unsuccessful search, say search for 012. We follow the 0-th pointer out of the root, i.e., the pointer in location $4+0$. This leads us to a leaf with content $121 \neq 012$. So 012 is not member of the set.

We will now describe one particular compression technique in more detail. Let A be an r by s 0–1 matrix with exactly m entries equal to one. We want to find small row displacements $rd(i)$, $0 \le i \le r-1$, such that $A[i,j] = 1 = A[i',j']$ and $(i,j) \ne (i',j')$ implies $rd(i) + j \ne rd(i') + j'$ for all pairs (i,j) and (i',j'), i.e., if we store the i-th row of matrix A beginning at position $rd(i)$ of a one-dimensional array $C[0\,..\,]$ then no two ones will collide. In our example the 0's correspond to the nil pointers and the 1's correspond to the non-nil pointers. One method for computing the row displacements is the **First-Fitdecreasing method**:

1) Sort the rows in non-decreasing order according to the number of ones in the row, i.e., the row with the maximal number of ones comes first.
2) Set $rd(0) = 0$. For $i \ge 1$ choose $rd(i) \ge 0$ minimal such that no collision with the previously placed rows 0 to $i-1$ occurs.

In our example step 1 could produce the following matrix. In step 2 we choose
Hinw.Abb. $rd(b) = 0$, $rd(c) = 3$, $rd(\text{root}) = 4$ and $rd(a) = 7$. These choices are illustrated by the second matrix.

b:	1	1	1
c:	1	1	0
root:	0	1	1
a:	1	0	1

0	1	2	3	4	5	6	7	8	9
1	1	1							
			1	1	0				
				0	1	1			
							1	0	1

Figure 5. Overlaytechnik mit First-Fit-Methode

The First-Fitdecreasing method produces small row displacements if ones of matrix A are evenly distributed across the matrix.

Theorem 3. *Let $m(l)$ be the total number of ones in rows of A which contain $l+1$ or more ones. If $m(l) \le m/(l+1)$ for all l then the First-Fitdecreasing method*
$O(r \cdot m \cdot s)$? *produces row displacements $rd(i) \le m$ for $0 \le i \le r-1$ in time $O(r \cdot s + m^2)$.*

Proof: Consider a row with exactly l ones. When that row is placed in step 2 only rows with $\ge l$ ones have been placed and hence the array C can contain at most $m(l-1)$ ones. Each such one can block at most l possible choices for the row displacement. Since $l \cdot m(l-1) \le m$ by assumption we can always find $rd(i) \le m$ for all i, $0 \le i \le r-1$.

We still have to prove the time bound. We compute the number of ones in each row in time $O(r \cdot s)$. Bucketsort will then sort the rows according to the
$O(r+m)$? number of ones in time $O(r+s)$ Finally, in order to compute $rd(i)$ one has to
Sinn?! try up to m candidates. Dazu sucht man für jede zu plazierende Eins von Zeile i eine Null in C. Wenn die i-te Zeile r_i Einsen hat, kann ein Kandidat in Zeit $O(r_i)$ getestet werden. Dazu speichern wir die Einsen einer jeden Zeile zusätzlich in verketteten Listen ab. Also finden wir $rd(i)$ in Zeit $O(m \cdot r_i)$. Die Gesamtlaufzeit ist $O(r \cdot s + r + s + \sum_{i=0}^{r-1} m \cdot r_i) = O(r \cdot s + m^2)$. We obtain a total of $O(r \cdot s + r + m + r \cdot m \cdot s)$ time units. ∎

The hypothesis of Theorem 3, the harmonic decay property, imposes a severe restriction on the distribution of ones in matrix A. We have $m(1) \le m/2$ and thus at least half of the ones have to lie in rows with exactly one entry equal to one. Also $m(\sqrt{m}) < m/\sqrt{m} = \sqrt{m}$ and hence no row of A can contain $\sqrt{m}$ or more ones. What can we do if A does not have the harmonic decay property? We try to find column displacements $cd(j)$, $0 \le j \le s-1$, such that matrix B obtained from matrix A by displacing columns has the harmonic decay property. Then we apply the previously described compression technique to matrix B.

Für unser Beispiel könnten wir etwa $cd(0) = 0$, $cd(1) = 2$ und $cd(2) = 3$ wählen und erhalten die (7×3)-Matrix B aus Abbildung 6.

B :

	0	1	2
0	1		
1	1		
2	0	1	
3	1	1	1
4		1	0
5		0	1
6			1

cd :

0	1	2
0	2	3

Figure 6. Situation nach Spaltenverschiebungen

Jetzt sind wir in der Lage, die Zeilenverschiebungen gemäß der absteigenden First-Fit-Methode zu wählen. Beachten Sie, daß Schritt 1) unseres Algorithmus die Zeilen von B in die Reihenfolge 3, 0, 1, 2, 4, 5, 6 umordnet. Schritt 2) wählt $rd(3) = 0$, $rd(0) = 3$, $rd(1) = 4$, $rd(2) = 4$, $rd(4) = 5$, $rd(5) = 5$ und $rd(6) = 6$ (cf. Fig. 7).

	0	1	2	3	4	5	6	7	8
0				1	0	0			
1					1	0	0		
2					0	1	0		
3	1	1	1						
4						0	1	0	
5						0	0	1	
6							0	0	1

Figure 7. Situation nach Spalten- und Zeilenverschiebungen

Somit entsteht das Feld C der Länge 9 aus Abbildung 8.

C :

0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1

Figure 8. Die komprimierte Matrix C

The important fact to remember is that all ones of matrix A are mapped on distinct ones of matrix C, namely $A[i,j] = 1$ implies $A[i,j] = B[i+cd(j),j] = C[rd(i+cd(j))+j]$. The question remains how to choose the column displacements. We will use the First-Fitmethod. Let m_j be the number of ones in columns $0,\ldots,j$ of matrix A. We choose $cd(0)$, $cd(1)$, $\ldots$ in that order. Suppose we have chosen $cd(0),\ldots,cd(j-1)$ and applied these displacements to the first j columns of matrix A. Let B_{j-1} be the matrix obtained in that way and let $m_{j-1}(l)$ be the number of ones in rows of B_{j-1} with $l+1$ or more ones. We want $B = B_{s-1}$ to have the harmonic decay property, i.e.,

$$m_{s-1}(l) \le m/(l+1) \qquad \text{for all } l \ge 0.$$

In order to ensure the harmonic decay property after all columns displacements have been chosen we impose a more stringent restriction during the selection process, i.e., we impose

$$(*) \qquad m_j(l) \le m/f(l,m_j),$$

where f will be chosen later. The boundary conditions for f are $f(l,m_{s-1}) \ge l+1$ and $f(0,m_j) \le m/m_j$. The former condition ensures the harmonic decay property at the end of the construction, the latter condition makes sure that we can choose
i.D. mehr $cd(0) = 0$ and that the requirement can be satisfied for $l = 0$. Note that $m_j(0) = m_j$ for all j and $m_0(l) = 0$ for $l > 0$.

We choose the First-Fitmethod to choose the column displacements: Choose
$cd(0) = 0$ $cd(j) = 0$. For $j > 0$ choose $cd(j) \ge 0$ minimal such that $m_j(l) \le m/f(l,m_j)$ for all $l \ge 0$.

We need an upper bound on the values $cd(j)$ obtained in that way. We ask the following question: Consider a *fixed* $l \ge 1$ with the requirement

$$(l) \qquad m_j(l) \le m/f(l,m_j).$$

How many choices of $cd(j)$ may be blocked because fo violation of requirement (l)? If a particular choice, say k, of $cd(j)$ is blocked then

$$m_j(l) > m/f(l,m_j),$$

where $m_j(l)$ is computed using displacement k for the j-th column. Since requirement (l) is satisfied after choosing $cd(j-1)$ we also have

$$m_{j-1}(l) \le m/f(l,m_{j-1}),$$

and hence

$$m_j(l) - m_{j-1}(l) > m/f(l,m_j) - m/f(l,m_{j-1}).$$

Let $q = m/f(l,m_j) - m/f(l,m_{j-1})$. We can interpret q as follows: In matrix B_j the number of ones in rows with $\ge l+1$ ones is at least q more than in matrix B_{j-1}. We can count these q ones in a different way. There are at least $q/(l+1)$ pairs (a row of B_{j-1} with $\geq l$ ones, a one in column j) which are aligned when column displacement $cd(j) = k$ is chosen. Note that any pair contributes either 1 (if the row of B_{j-1} has $\geq l+1$ ones) or $l+1$ (if the row of B_{j-1} has exactly l ones) to q. Since the number of rows of B_{j-1} with $\geq l$ ones is bounded by $m_{j-1}(l-1)/l \leq m/(l \cdot f(l-1, m_{j-1}))$ and since the j-th column of A contains exactly $m_j - m_{j-1}$ ones the number of such pairs is bounded by

$$p \;=\; \frac{(m_j - m_{j-1}) \cdot m}{l \cdot f(l-1, m_{j-1})}$$

i.D.mehr Da die zu den Blockaden gehörenden Mengen von Paaren disjunkt sind, Hence there are at most $p/(q/(l+1))$ possible choices k for $cd(j)$ which are blocked because of violation of requirement l, $l \geq 1$. Hence the total number of blocked values is bounded by

$$BV \;=\; \sum_{l=1}^{l_0} \frac{(l+1) \cdot (m_j - m_{j-1}) \cdot m}{l \cdot f(l-1, m_{j-1}) \cdot [m/f(l, m_j) - m/f(l, m_{j-1})]}\,,$$

where $l_0 = \min\{l;\; m/f(l, m_{j-1}) \leq l\}$. Note that there are no rows of B_{j-1} with $> l_0$ ones and hence p (as defined above) will be 0 for $l > l_0$. Hence we can always choose $cd(j)$ such that $0 \leq cd(j) \leq BV$.

A bound on BV remains to be derived . We rewrite the expression for BV as follows:

$$BV = \sum_{l=1}^{l_0} \frac{l+1}{l} \cdot \frac{m_j - m_{j-1}}{f(l, m_{j-1})/f(l, m_j) - 1} \cdot \frac{f(l, m_{j-1})}{f(l-1, m_{j-1})}.$$

This expression involves only quotients of f and therefore we set $f(l, m_j) = 2^{g(l, m_j)}$ for some function g to be chosen later. We obtain

$$\begin{aligned} BV &= \sum_{l=1}^{l_0} \frac{l+1}{l} \cdot \frac{m_j - m_{j-1}}{2^{g(l, m_{j-1}) - g(l, m_j)} - 1} \cdot 2^{g(l, m_{j-1}) - g(l-1, m_{j-1})} \\ &\leq \sum_{l=1}^{l_0} \frac{l+1}{l} \cdot \frac{m_j - m_{j-1}}{[g(l, m_{j-1}) - g(l, m_j)] \cdot \ln 2} \cdot 2^{g(l, m_{j-1}) - g(l-1, m_{j-1})}, \end{aligned}$$

since $2^x - 1 \geq x \cdot \ln 2$. Next we note that the two differences involve only one argument of g. This suggests to set $g(l, m_j) = h(l) \cdot k(m_j)$ for some functions h and k to be chosen later. The upper bound for BV simplifies to

$$BV \leq \sum_{l=1}^{l_0} \frac{l+1}{l} \cdot \frac{m_j - m_{j-1}}{h(l) \cdot [k(m_{j-1}) - k(m_j)] \cdot \ln 2} \cdot 2^{(h(l) - h(l-1)) \cdot k(m_{j-1})}.$$

This sum will further simplify if we choose h and k to be linear functions, say $h(l) = l$ and $k(m_j) = 2 - m_j/m \le 2$. Then

$$\begin{aligned}
BV &\le \sum_{l=1}^{l_0} \frac{l+1}{l} \cdot \frac{m_j - m_{j-1}}{l \cdot (m_j/m - m_{j-1}/m) \cdot \ln 2} \cdot 2^2 \\
&= \sum_{l=1}^{l_0} \frac{4m}{\ln 2} \cdot \frac{l+1}{l^2} \\
&= \frac{4m}{\ln 2} \left(\sum_{l=1}^{l_0} 1/l + \sum_{l=1}^{l_0} 1/l^2 \right) \\
&\le \frac{4m}{\ln 2} (\ln l_0 + 1 + \pi^2/6) \\
&\qquad \left(\text{since } \sum_{l=1}^{l_0} 1/l \le 1 + \ln l_0 \qquad \text{(cf. appendix)}\right. \\
&\qquad \left.\text{and } \sum_{l \ge 1} 1/l^2 = \pi^2/6\right) \\
&\le 4m \log l_0 + 15.3m \ .
\end{aligned}$$

Finally observe that $f(l, m_j) = 2^{l \cdot (2 - m_j/m)} \ge 2^l$ and therefore $l_0 \le \log m$. Also $f(l, m_{s-1}) = 2^l \ge l + 1$ for all l and $f(0, m_j) = 2^0 = 1 \le m/m_j$ for all j and so f satisfies the boundary conditions. Thus we can always find $cd(j)$ such that $0 \le cd(j) \le 4m \log \log m + 15.3m$ gilt.

Theorem 4. *Given a matrix $A[0\mathinner{..}r-1, 0\mathinner{..}s-1]$ with m nonzero entries one can find column displacements $cd(j)$, $0 \le j \le s-1$, such that $0 \le cd(j) \le 4m \log \log m + 15.3m$ and such that matrix B obtained from A using those displacements has the harmonic decay property. The column displacements can be found in time*
Zeit!? $O(s(r + m \log \log m)^2)$.

Proof: The bound on the column displacements follows from the discussion above. The time bound can be seen as follows. For every row of B_j we keep a count on the number of ones in the row and keep the numbers $m_j(i)$ and m_j. In order to find $cd(j+1)$ we have to test up to $4m \log \log m + O(m)$ possible values.
versch.Erg. Einen Kandidaten plazieren wir in Zeit $O(s_{j+1})$, wobei s_{j+1} die Anzahl der Einsen in Spalte $j+1$ ist. Dazu verwalten wir zusätzlich die Einsen einer jeden Spalte als verkettete Liste. Weiter müssen wir für den untersuchten Kandidaten in Zeit $O(s_{j+1})$ die betroffenen $m_j(i)$ aktualisieren, gegebenenfalls wieder zurücksetzen und in Zeit $O(l_0) = O(\log m)$ auf Verletzung der Bedingung (l) für $1 \le l \le l_0$ überprüfen. Also ist die Gesamtzeit für die Berechnung der Spaltenverschiebungen $O(m \log \log m \cdot \sum_{j=0}^{s-1}(s_i + \log m)) = O(m^2 \log \log m)$. ∎

Let us combine the Theorems 3 and 4. Given a matrix $A[0\,..\,r-1, 0\,..\,s-1]$ with m nonzero entries, Theorem 4 gives us column displacements $cd(j)$, $0 \le j < s$, and
i.D.mehr a matrix B such that und eine Matrix B, die die Eigenschaft des harmonischen Abstiegs erfüllt, so daß

$$A[i,j] \neq 0 \quad \text{implies} \quad B[i + cd(j), j] = A[i,j]$$

B has s columns and $r' \le r + 4m \log\log m + 15.3m$ rows. Of course, B has at most m rows with at least one nonzero entry. Next we apply Theorem 3 and obtain row displacements $rd(i)$, $0 \le i < r'$ and a one-dimensional matrix C such that

$$B[h,j] \neq 0 \quad \text{implies} \quad C[rd(h) + j] = B[h,j],$$

or in other words

$$A[i,j] \neq 0 \quad \text{implies} \quad C[rd(i + cd(j)) + j] = A[i,j].$$

Since B has only m nonzero entries there are at most m different h's such that $rd(h) \neq 0$. Furthermore, array C has length at most $m + s$ since $rd(h0 \le m$ for all h.

So far, we have obtained the following reduction in space: C uses $m+s$ storage locations, cd uses s storage locations and rd uses $r' \le r + 4m \log\log m + 15.3m$ storage locations.

Let us apply our compression technique to a TRIE for a set of n elements with branching factor $k \le n$; $k = n$ will give the best result with respect to access time. We can view the array representation of the internal nodes as an r, $r \le n-1$, by s, $s = k$, array with $\le 2n-2$ non-nil entries. Compression gives us a matrix C with $O(k+n) = O(n)$ locations, a set of $k = O(n)$ column displacements and a set of $O(n \log\log n)$ row displacements. So total space requirement is $O(n \log\log n)$.

The entries of arrays rd, cd and C are numbers in the range of 0 to $O(n \log\log n)$, i.e., bitstrings of length $\le O(\log n)$. As we observed above, array rd has $c \cdot n \log\log n$ entries for some constant c all but $2n-2$ of which are zero. We will next describe a compression of vector rd based on the following assumption: A storage location can hold several numbers if their total length is less than $\log n$.

Let $i_0, i_1, \ldots, i_{t-1}$, $t \le 2n-2$, be the indices of the nonzero elements of vector rd. We compress vector rd into a vector crd of length t by storing only the nonzero entries, i.e., $crd[l] = rd[i_l]$ for $0 \le l \le t-1$. We still have to describe a way of finding l given i_l.

Let $d = \lfloor \log\log n \rfloor$. We divide vector rd into $c \cdot n \log\log n/d \le 2c \cdot n$ blocks of length d each. For each block we write down the minimum l such that i_l lies in that block, if any such l exists. This defines a vector *base* of length $2c \cdot n$. For any other element of a block we store the offset with respect to l in a two-dimensional array *offset*, i.e.,

$$base[v] = \begin{cases} \min\{l;\ i_l \text{ div } d = v\} & \text{if } \exists l : i_l \text{ div } d = v; \\ -1 & \text{otherwise} \end{cases}$$

for $0 \le v < 2c \cdot n$ and

$$\mathit{offset}[v,j] = \begin{cases} l - \mathit{base}[v] & \text{if } v \cdot d + j = i_l \text{ for some } l; \\ -1 & \text{otherwise} \end{cases}$$

for $0 \le j < d$.

Then is $rd[h] = 0 \iff \mathit{offset}[h \text{ div } d, h \bmod d] = -1$ and $rd[h] \ne 0$ implies $rd[h] = crd[\mathit{base}[h \text{ div } d] + \mathit{offset}[h \text{ div } d, h \bmod d]]$.

For any fixed v ist $\mathit{offset}[v,\ ]$ is a list of d numbers in the range $-1,\dots,d-1$. We combine these numbers into a single number $\mathit{off}[v]$

$$\mathit{off}[v] = \sum_{j=0}^{d-1} (\mathit{offset}[v,j] + 1) \cdot (d+1)^j .$$

Then $0 \le \mathit{off}[v] \le (d+1)^d$ and thus $\mathit{off}[v]$ fits into a single storage location. Also

$$\mathit{offset}[v,j] = ((\mathit{off}[v] \text{ div } (d+1)^j) \bmod (d+1)) - 1,$$

and so $\mathit{offset}[v,j]$ can be computed in time $O(1)$ from $\mathit{off}[v]$ and a table of the powers of $d+1$. Altogether we decreased the space requirement to $O(n)$, namely to $O(n)$ for array crd, base and off, and respectively increased access time only by a constant factor.

We illustrate these definitions by an example in Figure 9, $d = 3$, $i_0 = 1$, $i_1 = 3$, $i_2 = 5$ and $i_3 = 11$. Note that $\mathit{off}[1] = (1 + \mathit{offset}[1,0]) \cdot 4^0 + (1 + \mathit{offset}[1,1]) \cdot 4^1 + (1 + \mathit{offset}[1,2]) \cdot 4^2 = 1 \cdot 4^0 + 0 \cdot 4^1 + 2 \cdot 4^2 = 33$.

rd :

0 1 0	1 0 1	0 0 0	0 0 1

crd :

$rd(1)$	$rd(3)$	$rd(5)$	$rd(11)$

base :

0	1	−1	3

offset :

−1	0	−1
0	−1	1
−1	−1	−1
−1	−1	0

und *off* :

4
33
0
16

Figure 9. Komprimierung von rd

Theorem 5. *Let $S \subseteq U$, $|S| = n$ and $|U| = N$. Then an n-ary TRIE supports operation Access in time $O(\log_n N)$ worst case and $O(1)$ expected case. The TRIE can be stored in $O(n)$ storage locations (of $O(\log n)$ bits each).*

Proof: The time bound follows from Theorems 1 and 2. The space bound follows from the preceding discussion. ∎

We will improve on Theorem 5 in Section 3.2.3 on perfect hashing and show how to obtain $O(1)$ access time in the worst case. We included Theorem 5 because the compression technique used to prove Theorem 5 is of general interest. In particular, it compresses large sparse tables without seriously degrading access time. Note that an access to array A is replaced by an access to the three arrays rd, cd and C and a few additions, at least as long as we are willing to tolerate the use of $O(n \log \log n)$ storage locations.

3.2. Hashing

The ingredients of hashing are very simple: an array $T[0\,..\,m-1]$, the hash table, and a function $h : U \mapsto [0\,..\,m-1]$, the hash function. U is the universe; we will assume $U = [0\,..\,N-1]$ throughout this section. The basic idea is to store a set S as follows: $x \in S$ is stored in $T[h(x)]$. Then an access is an extremely simple operation: compute $h(x)$ and look up $T[h(x)]$.

Suppose $m = 5$, $S = \{3, 15, 22, 24\} \subseteq [0\,..\,99]$ and $h(x) = x \bmod 5$. Then the
Abb.Hinw. hash table appears as follows:

0	15
1	
2	22
3	3
4	24

Figure 10. Eine Hashtafel

There is one immediate problem with this basic idea: what to do if $h(x) = h(y)$ for some $x, y \in S$, $x \neq y$? Such an event is called a **collision**. There are two main methods for dealing with collisions: chaining and open addressing.

3.2.1. Hashing with Chaining

The hash table T is an array of linear lists. A set $S \subseteq U$ is represented as m linear lists. The i-th list contains all elements $x \in S$ with $h(x) = i$.

Figure 11 shows the representation of set $S = \{1, 3, 4, 7, 10, 17, 21\}$ in a table of length $m = 3$. We use hash function $h(x) = x \bmod 3$.

figure does not exist yet

Figure 11. Beispiel für Hashing mit Verkettung

Operation Access(x, S) is realized by the following program:

1) compute $h(x)$;

2) search for element x in 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 $T[h(x)]$. The time complexity of hashing with chaining is easy to determine: the time for evaluating hash function h plus the time for searching through 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 key x as $O(1 + \delta_h(x, S))$ where S is the set of stored elements and

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

and

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

The worst case complexity of hashing is easily determined. The worst case occurs when the hash function h restricted to set S is a constant, i.e., $h(x) = i_0$ for all $x \in S$. Then hashing deteriorates to searching through 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. We analyze the complexity of a sequence of n insertions, deletions and accesses starting with an empty table, i.e., of a sequence $Op_1(x_1), \ldots, Op_n(x_n)$, where $Op_k \in$ {Insert, Delete, Access} and $x_k \in U$, on the following probability assumptions (we will discuss these assumptions later).

1) Hash function $h : U \mapsto [0\,..\,m-1]$ distributes the universe uniformly over the interval $[0\,..\,m-1]$, i.e., for all $i, i' \in [0\,..\,m-1]$: $|h^{-1}(i)| = |h^{-1}(i')|$.

2) All elements of U are equally likely as an argument of any one of the operations in the sequence, i.e., the argument of the k-th operation of the sequence is equal to a fixed $x \in U$ with probability $1/|U|$.

Our two assumptions imply that value $h(x_k)$ of the hash function on the argument of the k-th operation is uniformly distributed in $[0\,..\,m-1]$, i.e., $\text{prob}\big(h(x_k) = i\big) = 1/m$ for all $k \in [1\,..\,n]$ and $i \in [0\,..\,m-1]$.

Theorem 2. *On the assumptions above, a sequence of n insertions, deletions and access-operations takes time $O((1 + \beta/2) \cdot n)$ where $\beta = n/m$ is the (maximal) load factor of the table.*

Proof: We will first compute the expected cost of the $(k+1)$-st operation. Offensichtlich sind diese Kosten am größten, wenn die ersten k Operationen sämtlich
i.D.mehr Einfügungen sind. Daher analysieren wir diesen Fall. Assume that $h(x_{k+1}) = i$, i.e., the $(k+1)$-st operation accesses the i-th list. Let $\mathrm{prob}(l_k(i) = j)$ be the probability that the i-th list has length j after the k-th operation. Then

$$EC_{k+1} = \sum_{j \geq 0} \mathrm{prob}(l_k(i) = j) \cdot (1 + j)$$

is the expected cost of the $(k+1)$-st operation. Um die Abschätzungen übersichtlich zu halten, unterlassen wir es, diesen und die folgenden Ausdrücke (Beweise der Sätze
i.D.mehr 3, 4 und 5) in O-Notation zu schreiben. Next note that

$$\mathrm{prob}(l_k(i) = j) \ \leq \ \binom{k}{j} \cdot (1/m)^j \cdot (1 - 1/m)^{k-j}$$

(with equality if the first k operations are insertions) and hence

$$\begin{aligned} EC_{k+1} &\leq 1 + \sum_{j \geq 0} \binom{k}{j} \cdot (1/m)^j \cdot (1 - 1/m)^{k-j} \cdot j \\ &= 1 + \frac{k}{m} \sum_{j \geq 1} \binom{k-1}{j-1} \cdot (1/m)^{j-1} \cdot (1 - 1/m)^{k-j} \\ &= 1 + \frac{k}{m}. \end{aligned}$$

Thus the total expected cost of n operations is

$$\begin{aligned} \sum_{k=1}^{n} EC_k &= O\Big(\sum_{k=1}^{n} (1 + \frac{k-1}{m})\Big) \\ &= O\Big(n + \frac{(n-1)n}{2m}\Big) \\ &= O\Big(n + \frac{\beta n}{2}\Big). \end{aligned}$$

∎

We will next discuss the probability assumptions used in the analysis of hashing. The first assumption is easily satisfied. Hash function h distributes the universe U uniformly over the hash table. Suppose $U = [0 \,..\, N-1]$ and $m | N$, i.e., m divides N. Then $h(x) = x \bmod m$ will satisfy the first requirement (division method). If m does not divide N but N is much larger than m then the division method almost satisfies assumption 1) and Theorem 2 remains true.

The second assumption is more critical because it postulates a certain behavior of the user of the hash function. In general, the exact conditions of latter use are not known when the hash function is designed and therefore one has to be very careful about applying Theorem 2. We discuss one particular application now and came back to the general problem in the sections on perfect and universal hashing. fragefehlt i.E. Insbesondere werden wir im Abschnitt über universelles Hashing einfache Klassen von guten Hashfunktionen kennenlernen.

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 nonuniformity to obtain even better behavior than predicted by Theorem 2. Inside a computer identifiers are represented as bitstrings; usually 8 bits (a byte) are used to represent one character. In other words, we assign a number $num(C) \in [0\,..\,255]$ 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 $256|(a - b)$. As identifiers of the form I0, I1, I2, and X0, X1, X2 are used so frequently, we want

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

i.e., if $h(x) = x \bmod m$ for one m then we want for this m

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

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 even better practical performance
i.D.mehr than predicted by Theorem 2. In Abbildung 12 fassen wir die Ergebnisse einer Untersuchung von Lum (71) über das mittlere Verhalten von Hashing mit Verkettung zusammen.

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

Figure 12. Mittlere Zeitkomplexität von Hashing mit Verkettung

Hashing as we described it entails further problems apart from assumption 2). 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. This implies that either one 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 increases storage requirement, which also implies an increased running time in many environments. If the hash table is too large to be stored in main storage, then a small load factor will increase the number of page faults.

In the second case we can use a sequence T_0, T_1, T_2, ... of hash tables of size m, $2m$, $4m$, ... respectively. Also, for each i we have a hash function h_i. We start with hash table T_0. Suppose now, that we use hash table T_i and the load factor reaches 1 or 1/4. In the first case, we move to hash table T_{i+1} by storing all $2^i m$ elements present in T_i in table T_{i+1}. On the average, this will cost $O(2^i m)$ time units. Also, the load factor of table T_{i+1} is 1/2 and therefore we can execute at least $(1/4) \cdot 2^{i+1} m = (1/2) \cdot 2^i m$ operations until the next restructuring is required. In the second case, we move to hash table T_{i-1} by storing all $(1/4) \cdot 2^i m$ elements present in T_i in table T_{i-1}. On the average, this will cost $O((1/4) \cdot 2^i m)$ time units. Also, the load factor of table T_{i-1} is 1/2 and therefore we can execute at least $(1/4) \cdot 2^{i-1} m$ operations without any further restructuring. In either case, the cost of restructuring is twice the number of subsequent operations which can be performed without further restructuring. Hence, we can distribute (conceptually) the cost of restructuring to the operations following the restructuring but preceding the next restructuring process. In this way we will assign cost $O(1)$ to each operation and so the average cost of an operation is still $O(1)$. Also, the load factor is always at least 1/4 (except when table T_0 is in use).

We continue this section with a remark on operations $\mathrm{Ord}(k, S)$ and $\mathrm{List}(S)$. Since the elements of set S are completely scattered over the hash table T both operations cannot be realized at reasonable cost. One of the main applications of operation $\mathrm{List}(S)$ is batching requests. Suppose that we want to perform n_1 operations Access, Insert, Delete on set S. Instead of performing n_1 single operations it is often better to sort the requests bt the key referred to and to perform all requests by a single sweep of set S. If set S is stored in a hash table we may still use this method. We apply the hash function to the requests and sort the requests by the hashed key. Then we process all requests during a single scan through the hash table.

We end this section with a second look ast the worst case behavior of hashing and discuss the expected worst case behavior of hashing with chaining. Suppose that a set of n elements is stored in the hash table. Then $\max_{x \in U} \delta_h(x, S)$ is the worst case cost of an operation when set S is stored. We want to compute the expected value of the worst case cost on the assumption that S is a random subset of the universe. More precisely, we assume that $S = \{x_1, \ldots, x_n\}$ and that $\mathrm{prob}(h(x_k) = i) = 1/m$ for all $k \in [1\,..\,n]$ and $i \in [0\,..\,m-1]$. The very same assumption underlies Theorem 2.

Theorem 3. *On the assumption above, the expected worst case cost of hashing with chaining is* $O((\log n)/(\log\log n))$ *provided that* $\beta = n/m \leq 1$.

Proof: Let $l(i)$ be the number of elements of S which are stored in the i-th list.

Then $\text{prob}(l(i) \geq j) \leq \binom{n}{j}(1/m)^j$. Also

$$\begin{aligned}\text{prob}((\max_i l(i)) \geq j) &\leq \sum_{i=0}^{m-1} \text{prob}(l(i) \geq j)\\ &\leq m \cdot \binom{n}{j} \cdot (1/m)^j\\ &\leq n \cdot (n/m)^{j-1} \cdot (1/j!).\end{aligned}$$

Thus the expected worst case cost EWC, i.e., the expected length of the longest chain, is

$$\begin{aligned}EWC &= \sum_{j\geq 1} \text{prob}((\max_i l(i)) \geq j)\\ &\leq \sum_{j\geq 1} \min(1, n \cdot (n/m)^{j-1} \cdot (1/j!)).\end{aligned}$$

Let $j_0 = \min\{j;\ n \cdot (n/m)^{j-1} \cdot (1/j!) \leq 1\} \leq \min\{j;\ n \leq j!\}$, since $n/m \leq 1$. From $j! \geq (j/2)^{j/2}$ we conclude $j_0 = O((\log n)/(\log\log n))$. Thus

$$\begin{aligned}EWC &\leq \sum_{j=1}^{j_0} 1 + \sum_{j>j_0} 1/j_0^{j-j_0}\\ &= O((\log n)/(\log\log n)).\end{aligned}$$

∎

3.2.2. Hashing with Open Addressing

Each element $x \in U$ defines a sequence $h(x,i)$, $i = 0,1,2,\ldots$ of table positions. This sequence of positions is searched through whenever an operation referring to key x is performed.

A very popular method for defining function $h(x,i)$ is to use the linear combination of two hash functions h_1 and h_2:

$$h(x,i) = [h_1(x) + i \cdot h_2(x)] \bmod m$$

i.D.mehr wobei $h_2(x)$ nie den Wert 0 annehmen darf. We illustrate this method by an example; let $m = 7$, $h_1(x) = x \bmod 7$ and $h_2(x) = 1 + (x \bmod 4)$. If we insert 3, 17, 6, 9, 15, 13, 10 in that order, we obtain Figure 13.

Hashing with open addressing does not require any additional space. However, its performance becomes poor when the load function is nearly one and it does not support deletion.

We analyze the expected cost of an operation Insert(x) into a table with load factor $\beta = n/m$ on the following assumption: Sequence $h(x,i)$, $i = 0,1,\ldots,m-1$ is a random permutation of the set $\{0,\ldots,m-1\}$. The cost of operation Insert(x) is
i.D.mehr $1 + \min\{i;\ T[h(x,i)]$ is not occupied$\}$. (Wir müßten hier eigentlich $O(1 + \min\ldots)$ schreiben, vernachlässigen aber in diesem Abschnitt bewußt die O-Notation.)

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

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

Figure 13. Example for hashing with open addressing

Theorem 4. *The expected cost of an Insert into a table with load factor $\beta = n/m$ is $(m+1)/(m-n+1) \approx 1/(1-\beta)$ provided that $n/m < 1$.*

Proof: Le $C(n,m)$ be the expected cost of an Insert into table with m positions, n of which are occupied. Let $q_j(n,m)$ be the probability that table positions $h(x,0)$,..., $h(x,j-1)$ are occupied. Then $q_0(n,m) = 1$, $q_j(n,m) = \bigl[n/m\bigr] \cdot \bigl[(n-1)/(m-1)\bigr] \cdots \bigl[(n-(j-1))/(m-(j-1))\bigr]$ and

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

The expression for $q_j(n,m)$ can be justified as follows. Position $h(x,0)$ is occupied in n out of m cases. If $h(x,0)$ is occupied then position $h(x,1)$ is occupied in $n-1$ out of $m-1$ cases (note that $h(x,1) \neq h(x,0)$), ...

We prove $C(n,m) = (m+1)/(m-n+1)$, by induction on m and for fixed m by induction on n. Note first that $C(0,m) = q_0(0,m) = 1$. Note first that $q_j(n,m) = (n/m) \cdot q_{j-1}(n-1,m-1)$ for $j \geq 1$. Hence

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

∎

The expected cost of a successful Access operation is now easy to determine. Suppose that set S is stored. When Zugriff(x), $x \in S$, is executed we step through sequence $h(x, 0)$, $h(x, 1)$, ... until we find an i with $T[h(x, i)] = x$. Of course, the very same i determined the cost of inserting element $x \in S$ in the first place. Hence the expected cost of a successful search in a table with n elements is

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

Theorem 5. *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. ∎

Theorems 4 and 5 state that Insert and Access time will go up steeply as β approaches 1, and that open addressing works fine as long as β is bounded below one, say $\beta \le 0.9$ ((cf. Fig. 14)).

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

Figure 14. Anzahl von Operationen bei Hashing mit offener Adressierung

3.2.3. Perfect Hashing

We based our analysis of the expected performance of hashing with chaining on two assumptions, the second of which is very critical. It postulates uniformity on the key space, an assumption which is almost always violated. However, we saw in Section 2.1. that it is possible to tune the hash function according to the bias which is present in the key space. In this section we will consider the ultimate form of tuning. Set S is known when the hash function is chosen, h is chosen such that it operates injectively on S.

Definition:

a) A function $h : [0\,..\,N-1] \mapsto [0\,..\,m-1]$ is a **perfect hash function** for $S \subseteq [0\,..\,N-1]$ if $h(x) \neq h(y)$ for all $x, y \in S$, $x \neq y$.

b) A set H of functions $h : [0\,..\,N-1] \mapsto [0\,..\,m-1]$ is called **(N; m; n)-perfekt** (or "perfect" for short) if for every $S \subseteq [0\,..\,N-1]$, $|S| = n$, there is $h \in H$ such that h is perfect for S. ∎

Of course, every set S has a perfect hash function. In our example of the beginning of Section 2 we can take $h(x) =$ (last digit of x) mod 4. The most interesting questions concerning the hash functions are:

a) How large are the programs for perfect hash functions, i.e., is there always a short program computing a perfect hash function?

b) How difficult is it to find a perfect hash function given S, m and N?

c) How difficult is it to evaluate perfect hash functions?

In this section we will essentially answer all three questions completely. We first have to give a more precise definition of program size. The **size of a program** is just the length of the program written as a string over some finite alphabet. We prove upper bounds on program size by explicitly exhibiting programs of a certain length. We prove lower bounds on program length by deriving a lower bound on the number of different programs required and then exploiting the fact that the number of words (and hence programs) of length $\leq L$ over an alphabet of size c is equal to $(c^{L+1}-1)/(c-1)$.

Theorem 6. *Let N, m, $n \in \mathbb{N}$ and let H be a (N, m, n)-perfect class of hash functions. Then*

a) $|H| \geq \dfrac{\binom{N}{n}}{(N/m)^n \cdot \binom{m}{n}}$.

b) $|H| \geq \dfrac{\log N}{\log m}$.

c) *There is at least one $S \subseteq [0\,..\,N-1]$, $|S| = n$, such that the length of the shortest program computing a perfect hash function for S is*

$$\max\left(\frac{n(n-1)}{2m\ln 2} - \frac{n(n-1)}{2N\ln 2}\Big/\left(1-\frac{n-1}{N}\right),\ \log\log N - \log\log m\right)\Big/\log c - 1\ ,$$

here c is the size of the alphabet used for coding programs.

Proof: a) There are $\binom{N}{n}$ different subsets of $[0\,..\,N-1]$ of size n. It therefore suffices to show that any fixed function $h : [0\,..\,N-1] \mapsto [0\,..\,m-1]$ is perfect for at most $(N/m)^n \cdot \binom{m}{n}$ different subsets $S \subseteq [0\,..\,N-1]$, $|S| = n$. If h is perfect for S then

$|h^{-1}(i) \cap S| \le 1$ for all $i \in [0\,..\,m-1]$. Hence the number of sets S such that h is perfect for S is bounded by

$$\sum_{0 \le i_1 < i_2 < \cdots < i_n < m} |h^{-1}(i_1)| \cdot |h^{-1}(i_2)| \cdots |h^{-1}(i_n)|.$$

This expression is maximal if $|h^{-1}(i)| = N/m$ for all $i \in [0\,..\,m-1]$ and its value is equal to $(N/m)^n \cdot \binom{m}{n}$.

b) Let $H = \{h_1, \ldots, h_t\}$. We construct $U_i \subseteq U$, $0 \le i \le t$, such that for every
fehlt i.D. $S \subseteq U$, $|S \cap U_i| \ge 2$, functions $h_1, \ldots, h_i$ are not perfect for S. Then we must have $|U_t| \le 1$. Let $U_0 = U$ and

$$U_{i+1} = U_i \cap h_{i+1}^{-1}(j) \qquad \text{for } i < t$$

where j is such that $|U_i \cap h_{i+1}^{-1}(j)| \ge |U_i \cap h_{i+1}^{-1}(l)|$ for every $l \in [0\,..\,m-1]$. Then $|U_{i+1}| \ge |U_i|/m$ and hence $|U_{i+1}| \ge N/m^{i+1}$. Also functions $h_1, \ldots, h_{i+1}$ are
Sinn? constant on U_{i+1} and hence $|U_t| \le 1$. Weil H perfekt sein soll, ist $|U_t| \le 1$. Thus $1 \ge N/m^t$ or $t \ge \log N/\log m$.

c) Let LBa and LBb be the lower bounds for $|H|$ proven in parts a) and b). Since there are at most c^{L+1} different programs of length $\le L$ and different functions require different programs we conclude that there are least one $S \subseteq [0\,..\,N-1]$, $|S| = n$, such that the shortest program computing a perfect hash function for S has length $\max(\log LBa,\ \log LBb)/\log c - 1$. But

$$\log LBb = \log\log N - \log\log m$$

and

$$\begin{aligned}
\log LBa &= \log \frac{N \cdots (N-n+1) \cdot m^n}{N^n \cdot m \cdots (m-n+1)} \\
&= \log \left[\prod_{i=0}^{n-1} \frac{N-i}{N} \Big/ \prod_{i=0}^{n-1} \frac{m-i}{m} \right] \\
&= \left[\sum_{i=0}^{n-1} \ln(1-i/N) - \sum_{i=0}^{n-1} \ln(1-i/m) \right] \Big/ \ln 2 \\
&\ge \left[-\left(1 - \frac{n-1}{N}\right)^{-1} \cdot \sum_{i=0}^{n-1} \frac{i}{N} + \sum_{i=0}^{n-1} \frac{i}{m} \right] \Big/ \ln 2 \\
&\ge \frac{n(n-1)}{2m \ln 2} - \frac{n(n-1)}{2N \ln 2 \cdot (1-(n-1)/N)}.
\end{aligned}$$

Dabei gilt die erste Ungleichung, since $\ln(1-i/m) \le -i/m$ and hence $-\ln(1-i/m) \ge i/m$ and $\ln(1-i/N) \ge -(i/N)/(1-(n-1)/N)$ for $0 \le i \le n-1$ (cf. appendix). ∎

Informally, we can state part c) of Theorem 6 as follows. The length of the shortest program for a perfect hash function is at least $\frac{1}{2}[(\beta/(2\ln 2))n + \log\log N]/\log c$, i.e., there is an initial cost $\log\log N$ due to the size of the universe and there is an incremental cost of $\beta/(2\ln 2)$ bits per element stored . The incremental cost depends on the load factor $\beta = n/m$.

How good is that lower bound? We give the answer in two steps. Wer will first show by a non-constructive argument that the lower bound is almost achievable (Theorem 7 and 8). Unfortunately, the programs constructed in Theorem 8 are extremely inefficient, i.e., Theorem 8 settles question a) concerning the program size of perfect hash functions but does not give convincing answers to questions b) and c). As a second step we will then derive answers for questions b) and c).

Theorem 7. *Let* N, m, $n \in \mathbb{N}$. *If*

$$t \geq n \ln N \cdot e^{n^2/m},$$

then there is a (N, m, n)*-perfect class* H *with* $|H| = t$.

Proof: We may represent any class $H = \{h_1, \ldots, h_t\}$ by an N by t matrix

$$M(H) = (h_i(x))_{0 \leq x \leq N-1,\ 1 \leq i \leq t}$$

with entries in $[0\,..\,m-1]$, i.e., the i-th column of matrix $M(H)$ is the table of function values for h_i. Conversely, every N by t matrix is the representation of a class H of hash functions. There are $m^{N \cdot t}$ matrices of dimension N by t with entries in $[0\,..\,m-1]$.

We want to derive an upper bound on the number of non-perfect matrices, i.e., matrices which correspond to non-perfect classes of hash functions. If H does not contain a perfect hash function for $S = \{x_1 < x_2 < \cdots < x_n\}$, then the submatrix of $M(H)$ given by rows $x_1, \ldots, x_n$ cannot have a column of n different values. Hence the columns of that submatrix can be chosen out of $m^n - m \cdot (m-1) \cdots (m-n+1)$ possibilities (namely, the number of functions from n points into a set of m elements minus the number of injective functions), and hence the number of such submatrices is bounded by $\left[m^n - m \cdot (m-1) \cdots (m-n+1)\right]^t$. Recall that the submatrix has t columns. verschiedene Arten gewählt werden kann, Since S can be chosen in $\binom{N}{n}$ different ways, the number of non-perfect matrices is bounded by

$$\binom{N}{n} \left[m^n - m \cdot (m-1) \cdots (m-n+1)\right]^t m^{(N-n)t} .$$

Note that the rows corresponding to elements not in S may be filled arbitrarily. Thus there is a perfect class H, $|H| = t$, if

$$\binom{N}{n} \left[m^n - m \cdot (m-1) \cdots (m-n+1)\right]^t m^{(N-n)t} < m^{N \cdot t}$$

or

$$\binom{N}{n}\left[1-\frac{m\cdot(m-1)\cdots(m-n+1)}{m^n}\right]^t \;<\; 1$$

or

$$t \;>\; \frac{\ln\binom{N}{n}}{-\ln(1-m\cdot(m-1)\cdots(m-n+1)/m^n)}\;.$$

Since

$$\ln\binom{N}{n} \le n\ln N$$

and

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

and

$$\begin{aligned}\sum_{i=0}^{n-1}\ln(1-i/m) &\ge \int_0^n \ln(1-x/m)\,dx\\ &= m\cdot\big[(1-n/m)\cdot(1-\ln(1-n/m))-1\big]\\ &\ge m\cdot\big[(1-n/m)\cdot(1+n/m)-1\big]\\ &= -\,n^2/m\end{aligned}$$

there will be a perfect class H with $|H| = t$ provided that $t \ge n\ln N\cdot e^{n^2/m}$. ∎

Theorem 7 gives us an upper bound on the cardinality of perfect classes of hash functions; it does not yet give us an upper bound on program size. The upper bound on program size is given by the next theorem.

Theorem 8. *Let N, m, $n \in \mathbb{N}$. For every $S \subseteq [0\,..\,N-1]$, $|S| = n$, there is a program of length*

$$O\big(n^2/m + \log\log N + 1\big)$$

which computes a perfect hash function $h : [0\,..\,N-1] \mapsto [0\,..\,m-1]$ for S.

Proof: We will explicitly describe a program. The program implements the proof of Theorem 7; it has essentially 4 lines:

(1) $k \leftarrow \lceil \ln N\rceil$ written in binary;

(2) $t \leftarrow \lceil n\cdot k\cdot e^{n^2/m}\rceil$ written in binary;

(3) $i \leftarrow$ some number between 1 and t depending on S written in binary;

$e^k?2^k?$ (4) search through all $\lceil e^k \rceil$ by t matrices with entries in $[0\,..\,m-1]$ until a (N,m,n)-perfect matrix is found; use the i-th column of that matrix as the table of the hash function for S;

The correctness of this program follows immediately from Theorem 7; by Theorem 7 there is a $\lceil e^k \rceil$ by t perfect matrix and hence we will find it in step (4). One column of that matrix describes a perfect hash function for S, say the i-th. We set i to the appropriate value in line (3).

The length of this program is $O(\log\log N + 1)$ for line (1), $O(\log n + \log\log N + n^2/m + 1)$ for lines (2) and (3) and $O(1)$ for line (4). Note that the length of the text for line (4) is independent of N, t and m. This proves the claim. ∎

Theorems 6 and 8 characterize the program size of perfect hash functions; upper and lower bounds are of the same order of magnitude. Unfortunately, the program constructed in Theorem 8 is completely useless. It runs extremely slowly and it uses an immense amount of work space. Therefore, we have to look for constructive upper bounds if we also want to get some insight into questions b) and c).

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

Lemma 1. *Let N be a prime, let $S \subseteq [0\,..\,N-1]$, $|S| = n$. For every k, $1 \le k < N$, and $m \in \mathbb{N}$ let*

$$b_i^k = |\{x \in S;\ ((kx) \bmod N) \bmod m = i\}|$$

for $0 \le i < m$.

a) *For every m 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 $O(n \cdot N)$.*

Teil neu **b)** *For jedes m und ϵ, $0 < \epsilon \le 1$, kann ein k mit $\sum_{i=0}^{m-1} (b_i^k)^2 \le n + (2+\epsilon)n(n-1)/m$ in randomisierter Zeit $O(n/\epsilon)$ gefunden werden.*

Proof: a) For k, $1 \le k < N$, let $h_k(x) = ((kx) \bmod N) \bmod m$ be the hash function defined by multiplier k. Then

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

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

$$(k_1x) \bmod N - (k_1y) \bmod N = a\cdot((k_2x) \bmod N - (k_2y) \bmod N)$$

und daher

$$(k_1 - ak_2)(x-y) \bmod N = 0.$$

Weil $x\neq y$ und N prim (d.h. Primzahl) ist, folgt daraus, daß $(k_1 - ak_2) \bmod N = 0$ 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 1. *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 9. *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, $1 \le k < N$, 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, $0 \le i < n$, there is a k_i, $1 \le k_i < N$, 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 $m = \sum_{i=0}^{n-1} c_i = n + (\sum_i b_i^2 - \sum_i b_i) < n + 3n - n = 3n$. Dabei ist x die
i.D.mehr Eingabe:

(1) $i \leftarrow ((kx) \bmod N) \bmod n$;
(2) $j \leftarrow ((k_i x) \bmod N) \bmod c_i$;
(3) output $\sum_{l=0}^{i-1} c_l + j$

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 $m = \sum_{i=0}^{n-1} \lfloor((1+\delta)b_i(b_i-1)+1)\rfloor \le (1+\delta)\sum_{i=0}^{n-1} b_i^2 \le (1+\delta)(3+\delta)n$. ∎

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 10. *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.

Claim 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 1. *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*
i.E.mehr *prime.*

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
i.E.mehr primes $p \leq a \cdot n^2 \ln N$ will not divide D and hence satisfy a). Furthermore, the prime number 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 11. *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, $1 \leq k < p$, 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 $\geq 2n(n-1)+1$ 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 $p \leq a \cdot n^2 \log N$ 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 $\beta = n/mexceeding 1/3$, 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 $O(\log n)$.

3.2.4. Universal Hashing

In diesem Abschnitt lernen wir einfache Klassen von guten Hashfunktionen kennen. Universal hashing is a method for dealing with the basic problem of hashing: its linear worst case behavior. We saw in Section 3.2.1 that hashing provides us with $O(1)$ expected access time and $O(n)$ worst case access time. The average was taken over all sets $S \subseteq [0\,..\,N-1]$, $|S| = n$. In other words, a fixed hash function $h : [0\,..\,N-1] \mapsto [0\,..\,m-1]$ works well for a random subset $S \subseteq U$, but there are also some very "bad" inputs for h. Thus it is always very risky to use hashing when the actual distribution of the inputs is not known to the designer of the hash function. It is always conceivable, that the actual distribution favours worst case inputs and hence will lead to large average access times.

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 hash function 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 $x, y \in U$, $x \neq y$, a random element $h \in H$ should lead to collision, i.e., $h(x) = h(y)$, with fairly small probability.

Definition: Let $c \in \mathbb{R}$, $N, m \in \mathbb{N}$. A collection $H \subseteq \{h;\ h\colon [0\,..\,N-1] \mapsto [0\,..\,m-1]\}$ is **c-universal**, if for all $x, y \in [0\,..\,N-1]$, $x \neq y$

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

A collection H is c-universal if only a fraction c/m of the functions in H leads to collision on any pair $x, y \in [0\,..\,N-1]$. It is not obvious that universal classes of hash functions exist. We exhibit an almost 1-universal class in Theorem 12; in Exercise 6 it is shown that there are no c-universal classes for $c < 1 - n/m$¿ Thus Theorem 12 is almost optimal in that respect.

Theorem 12. *Let $m, N \in \mathbb{N}$ and let N be a prime. Then*

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

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

Proof: Note first that $|H_1| = N^2$. Let $x, y \in [0 \mathinner{..} N - 1]$, $x \neq y$. We have to show that

$$\big|\{(a,b);\ h_{a,b}(x) = h_{a,b}(y)\}\big| \leq c \cdot N^2/m.$$

If $h_{a,b}(x) = h_{a,b}(y)$ then there are $q \in [0 \mathinner{..} m - 1]$ and $r, s \in [0 \mathinner{..} \lceil N/m\rceil - 1]$ such that

$$ax + b = q + r \cdot m \bmod N$$
$$ay + b = q + s \cdot m \bmod N.$$

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. Hence

$$\big|\{(a,b);\ h_{a,b}(x) = h_{a,b}(y)\}\big| = m \cdot \lceil N/m\rceil^2$$

and therefore class H_1 is c-universal. ∎

Universal class h_1 has size N^2, i.e., $O(\log N)$ bits are required to specify a function of the class. A random element of H_1 may be selected by choosing two random numbers a, b in the range $[0 \mathinner{..} N-1]$. In Theorem 15 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 on 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 13. *Let $c \in \mathbb{R}$ and let H be a c-universal class of hash functions.*

a) *Let $S \subseteq [0 \mathinner{..} N - 1]$, $|S| = n$ and let $x \in [0 \mathinner{..} N - 1]$. Then*

$$\sum_{h\in H} (1 + \delta_h(x, S))/|H| \ \leq\ \begin{cases} 1 + c \cdot n/m & \text{if } x \notin S; \\ 1 + c \cdot (n-1)/m & \text{if } x \in S. \end{cases}$$

b) *The expected cost of an Access, Insert or Delete operation is $O(1+c\cdot\beta)$, where $\beta = n/m$ is the load factor.*

c) *The expected cost of a sequence of n Access, Insert and Delete operations starting with an empty table is $O((1+c\cdot\beta/2)\cdot n)$ where $\beta = n/m$ is the (maximal) load factor.*

Proof: a)

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

b) obvious from part a).

c) obvious from part b) and the observation that the expected cost of the i-th operation in the sequence is $O(1+c\cdot i/m)$. ∎

Theorem 13c) 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 Theorem 2 each subset $S\subseteq U$, $|S| = n$, was assumed to be equally likely, in Theorem 13c) 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.

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.

$N\in\mathbb{N}, t>0$ **Theorem 14.** *Let $c\in\mathbb{R}$ 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., $\mu = \big(\sum_{h\in H}\delta_h(x,S)\big)/|H|$. Then the probability that $\delta_h(x,S)\ge t\cdot\mu$ is less than $1/t$.*

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

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

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 15. *Let $N, m \in \mathbb{N}$ and $N \geq m$.*

a) *Let $H \subseteq \{h;\ h : [0 \mathinner{..} N-1] \mapsto [0 \mathinner{..} m-1]\}$ 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 $H_2 \subseteq \{h;\ h\colon [0 \mathinner{..} N-1] \mapsto [0 \mathinner{..} m-1]\}$ of hash functions with*

$$\log |H_2| = O(\log m + \log\log N).$$

Proof: a) Let $H = \{h_1, \ldots, h_t\}$. As in the proof of Theorem 6b) we construct a sequence $U_0 = [0 \mathinner{..} N-1]$, U_1, U_2, …, 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

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

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

b) Let $N, m \in \mathbb{N}$ 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

$$H_2 = \{g_{c,d}(h_l(x));\ t < l \leq 2t,\ 0 \leq c, d < p_{2t}\},$$

where

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

and

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

Then $|H_2| = t \cdot p_{2t}^2$ and hence $\log |H_2| = O(\log t) = O(\log m + \log\log N)$, since $\log p_{2t} = O(\log t)$ by the prime number theorem. It remains to be shown that H_2 is 8-universal. Let $x, y \in [0 \mathinner{..} N-1]$, $x \neq y$, arbitrary. We have to show that

$$|\{(c, d, l);\ g_{c,d}(h_l(x)) = g_{c,d}\bigl(h_l(y)\bigr)\}| \leq 8 \cdot |H_2|/m.$$

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

$$\begin{aligned} t \cdot m \cdot (\lceil p_{2t}/m \rceil)^2 \leq t \cdot m \cdot (1 + p_{2t}/m)^2 &= (t \cdot p_{2t}^2/m) \cdot (1 + m/p_{2t})^2 \\ &= (|H_2|/m) \cdot (1 + m/p_{2t})^2. \end{aligned}$$

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

$$\begin{aligned} |L| \cdot m \cdot \lceil (p_{2t}/m) \rceil \cdot p_{2t} &\leq (t \cdot p_{2t}^2/m) \cdot (1 + m/p_{2t}) \\ &= (|H_2|/m) \cdot (1 + m/p_{2t}). \end{aligned}$$

Altogether, we have shown that the number of solutions (c, d, l) is bounded by $2(1+\epsilon)^2|H_2|/m$ where $\epsilon = m/p_{2t} \leq 1$. (Note that $p_{2t} \leq m$ would imply $t \ln p_t <$
i.E.mehr $p_{2t} \ln p_{2t} \leq m \ln m \leq m \ln N$, a contradiction to the definition of t). Hence H_2 is 8-universal. ∎

3.2.5. 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 secondary storage 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 respect to bucket size b and hash function h is defined as

i.E.mehr

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

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 $T[0\,..\,2^{d(S)} - 1]$, 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 $0\,..\,2^{d(S)} - 1$.

(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 15 veranschaulichte Situation. ∎

figure does not exist yet

Figure 15. Example for extendible hashing

As one can see from Figure 15, 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 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 15 steht die lokale Tiefe der Körbe in der linken oberen Ecke.

i.E.anders

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.

Example: (continued). Suppose that we want to insert x with $h(x) = 0010$. It will cause the first bucket to overflow. We obtain Figure 16.

figure does not exist yet

Figure 16. Nach Einfügen von 0010 in die Beispielmenge

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 $T[a\overline{z}a_1]$ and $\overline{z} = 1 - z$, 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 fehlt i.D relevant quantities? 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.

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 16.

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 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 17 lists expected directory size for various choices of b and n.

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

Figure 17. Mittlere Kataloggrößen bei erweiterbarem Hashing

We can see from Figure 17 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 $n \le 10^{10}$. 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 $p_{n,m} \le 1/e$ if n exceeds $\sqrt{m}$. In other words, a directory of size m suffices for
$1 - 1/e$? a set of $n = \sqrt{m}$ elements with probability less than $1/e$. So directory size must grow non-linearly in file size and should in fact be quadratic in file size.

3.3. Searching Ordered Sets

We will now turn to comparison based methods for searching. We assume that U is linearly ordered and denote the linear ordering by "$\le$". The assumption of U being linearly ordered is no real restriction; there is always an ordering on the internal representations of the elements of U. The basis for all comparison-based methods is the following algorithm for searching ordered arrays. Let $S = \{x_1 < x_2 < \cdots < x_n\}$ be stored in array $S[1\,..\,n]$, i.e., $S[i] = x_i$, and let $a \in U$. In order to decide $a \in S$, we compare a with some table element and then proceed with either the lower or the upper part of the table. Programm 2 zeigt diesen Algorithmus in unserer höheren Programmiersprache.

```
(1)  low ← 1; high ← n;
(2)  next ← an integer in [low .. high];
(3)  while a ≠ S[next] and high > low
(4)  do if a < S[next]
(5)       then high ← next − 1
(6)       else low ← next + 1 fi;
(7)       next ← an integer in [low .. high]
(8)  od;
(9)  if a = S[next] then "successful" else "unsuccessful" fi.
```

Program 2

Various algorithms can be obtained from the scheme of Program 2 by replacing lines (2) and (7) by specific strategies for choosing *next*. **Linear search** is obtained by

$$next \leftarrow low,$$

binary search by

$$next \leftarrow \left\lceil \frac{high + low}{2} \right\rceil$$

and **interpolation search** by

$$next \leftarrow (low - 1) + \left\lceil \frac{a - S[low-1]}{S[high+1] - S[low-1]} \cdot (high - low + 1) \right\rceil.$$

fehlt Hierbei nehmen wir an, daß die Positionen $S[0]$ und $S[n+1]$ mit künstlichen Elementen gefüllt sind. We discuss these strategies in greater detail below.

The correctness of Program 2 is independent of the particular choice made in line (2) and (7). This can be seen from the fact that the following predicate P is an invariant of the body of the while-loop:

$$\big(a \in S \Rightarrow a \in \{S[low], \ldots, S[high]\}\big) \qquad \text{and}$$
$$\big(low \leq high \Rightarrow low \leq next \leq high\big).$$

P is certainly true before execution of the while-loop. If the loop body is entered then $a \neq S[next]$ and hence either $a < S[next]$ or $a > S[next]$. If $a < S[next]$ then certainly $a \notin \{S[next], \ldots, S[high]\}$ and hence $a \in S$ implies $a \in \{S[low], \ldots, S[next-1]\}$. The case $a > S[next]$ is treated similarly. Also a number in $[low\,..\,high]$ is assigned to $next$ in line (7) provided that $low \leq high$.

Thus when the while-loop terminates we have P and $((a = S[next])$ or $(high \leq low))$. If $a = S[next]$ then the search is successful. Suppose now that $a \neq S[next]$. Since P holds we know that $a \in S$ implies $a \in \{S[low], \ldots, S[high]\}$. If $high < low$ then certainly $a \notin S$. If $high = low$ then $next = high$ by P and hence $a \notin S$. In either case the search is unsuccessful.

Program 2 terminates because $high - low$ is decreased by at least one at each execution of the loop body.

3.3.1. Binary Search and Search Trees

Binary search is obtained by replacing lines (2) and (7) by

$$next \leftarrow \left\lceil \frac{high + low}{2} \right\rceil.$$

It is very helpful to illustrate the behavior of the algorithm by a binary tree. We do so for $n = 6$. In the tree shown in Figure 18 node $S[4]$ represents a comparison with $S[4]$.

figure does not exist yet

Figure 18. Binärer Suchbaum für 6 Elemente

If $a < S[4]$ then we go left, if $a > S[4]$ then we go right, if $a = S[4]$ then we are done. We start the search by comparing a and $S[4]$. If $a > S[4]$ then a can only be equal to $S[5]$ or $S[6]$. Next we compare a and $S[6], \ldots$. We can also see from Figure 18 that an unsuccessful search gives us information about the **rank** of a in
fehlt i.E. S d.h. die Anzahl der Elemente von S kleiner als a.

Since the value of $high - low + 1$ is halved at each iteration of the loop, the loop body is executed at most $(\log n)$ times. Thus an operation Access(a, S) takes time $O(\log |S|)$. Operations Ord(k, S) and Sequ(S) are trivial in this structure, however, Insert and Delete cannot be executed efficiently. An Insert would require that we move part of the array, a costly process. An elegant way out is to represent the tree above explicitly by pointers and not implicitly by an array. Then operations Insert and Delete can also be executed fast as we will see shortly. This leads to the following definition.

Definition: A **binary search tree** for set $S = \{x_1 < x_2 < \cdots < x_n\}$ is a binary tree with n nodes $\{v_1, \ldots, v_n\}$. The nodes are labelled with the elements of S, i.e., there is an injective mapping $CONTENT : \{v_1, \ldots, v_n\} \mapsto S$. The labelling preserves the order, i.e., if v_i (v_j) is a node in the left (right) subtree of the tree with root v_k then $CONTENT[v_i] < CONTENT[v_k] < CONTENT[v_j]$. ∎

An equivalent definition is as follows: a traversal of a search tree for S in symmetric order reproduce the order on S. We will mostly identify nodes with their labellings, i.e., instead of speaking of node v with label x we speak of node x and write ⓧ or simply x. Node x corresponds to the test aus Programm 3.

```
if a < x
   then go to the left son
   else if a = x
            then terminate search
            else go to the right son
        fi
fi.
```

Program 3

The $n + 1$ leaves represent unsuccessful access operations. It is not necessary to store leaves explicitly. Each leaf represents one of the $n + 1$ open intervals of U generated by the elements of S. We draw the leaf corresponding to interval $x_i < a < x_{i+1}$ as (x_i, x_{i+1}) durch ein rechteckiges Kästchen einrahmen (cf. Fig. 19). In the text we simply speak about (x_i, x_{i+1}). Leaf $(\,, x_1)$ represents all $a \in U$ with $a < x_1$. Sometimes we will write (x_0, x_1) instead of $(\,, x_1)$. A similar remark applies to leaf $(x_n, \,)$.

figure does not exist yet

Figure 19. Binärer Suchbaum mit Blättern

So let T be a binary search tree for set S. Then the Program 4 realizes operation Access(a, S):

If program 4 terminates in node v then $a = CONTENT[v]$. Otherwise it terminates in a leaf, say (x_i, x_{i+1}). Then $x_i < a < x_{i+1}$. Operation Insert(a, S) is now very easy to implement. We only have to replace leaf (x_i, x_{i+1}) by the tree aus Abbildung 20.

Deletion is slightly more difficult. A search for a yields node v with content a. We have to distinguish two cases.

```
v ← root of T;
while v is a node and a ≠ CONTENT[v]
do if a < CONTENT[v]
    then v ← LSOHN[v]
    else  v ← RSOHN[v]
    fi
od.
```

Program 4

figure does not exist yet

Figure 20. Neuer Unterbaum bei Einfüge(a, S)

Case 1: At least one son of v is a leaf, say the left.
Then we replace v by its right son and delete v and its left son from the tree.

Case 2: No son of v is a leaf.
Let w be the rightmost node in the left subtree of v. Node w can be found by following the left pointer out of v and then always the right pointer until a leaf is hit. We replace $CONTENT[v]$ by $CONTENT[w]$ and delete w as described in Case 1. Note that w's right son is a leaf.

Figure 21 illustrates both cases. The node with content 4 is deleted, leaves are not
fehlt i.E. drawn. da sie auch nicht explizit abgespeichert werden.

figure does not exist yet

Figure 21. Löschen von 4 in verschiedenen binären Suchbäumen

Operations Access, Insert and Delete essentially consist of a single pass down the tree followed by some local updating. Updating takes time $O(1)$ and hence the total cost is $O(h(T))$, where $h(T)$ is the height of tree T. Operation Sequ(S) is equivalent to the traversal of tree T in symmetric order. We have already seen (see 1.4.3 and 1.5) that tree traversal takes time $O(|S|)$. Finally, we show how to realize operation Ord(k, S) in time $O(h(T))$ by slightly extending the data structure. In each node we store the number of elements stored in the left subtree. These numbers are easily updated during insertions an deletions. With the use of these numbers Ord(k, S) takes time $O(h(T))$.

The considerations above show that the height of the search tree plays a crucial role in the efficiency of the basic set operations. We will see in 3.5 that $h(T) = O(\log |S|)$ can be achieved by the use of various balanced tree schemes. In Exercise 9b) it is shown that the average height of a random grown tree is $O(\log n)$.

3.3.2. Interpolation Search

Interpolation search is obtained by replacing lines (2) and (7) by

$$\mathit{next} \leftarrow (\mathit{low} - 1) + \left\lceil \frac{a - S[\mathit{low} - 1]}{S[\mathit{high} + 1] - S[\mathit{low} - 1]} \cdot (\mathit{high} - \mathit{low} + 1) \right\rceil .$$

It is assumed that positions $S[0]$ and $S[n+1]$ are added and filled with artificial elements. The worst case complexity of interpolation search is clearly $O(n)$; consider the case that $S[0] = 0$, $S[n+1] = 1$, $a = 1/(n+1)$ and $S \subseteq (0, a)$. Then $\mathit{next} = \mathit{low}$ and interpolation search deteriorates to linear search. Average case behavior is much better. Average access time is $O(\log\log n)$ under the assumption that keys $x_1, \ldots, x_n$ are drawn independently from a uniform distribution over the open interval (x_0, x_{n+1}).

An exact analysis of interpolation search is beyond the scope of this book. However, we discuss a variant of interpolation search, which also has $O(\log\log n)$ expected behavior: quadratic binary search.

Binary search has access time $O(\log n)$ because it consists of a single scanning of a path in a complete binary tree of depth $\log n$. If we could do binary search on the paths of the tree then we would obtain $O(\log\log n)$ access time. So let us consider the question whether there is a fast way (at least at the average) to find the node on the path of search which is halfway down the tree, i.e., the node on the path of search which has depth $\frac{1}{2}\log n$.

There are $2^{1/2\ \log n} = \sqrt{n}$ of these nodes and they are $\sqrt{n}$ apart in the array representation of the tree. Let us make an initial guess by interpolating and then search through these nodes by linear search. Note that each step of linear search skips $\sqrt{n}$ elements of S and hence as we will see shortly only $O(1)$ steps are required on the average. Thus an expected cost of $O(1)$ has reduced size of the set from n to $\sqrt{n}$ (or in other words determined the first half of the path of search) and hence total expected search time is $O(\log\log n)$.

The precise algorithm for Access(a, S) is as follows: Let $\mathit{low} = 1$, $\mathit{high} = n$ and $\mathit{next} = \lceil p \cdot n \rceil$ be defined as above; here $p = (a - x_0)/(x_{n+1} - x_0)$. If $a > S[\mathit{next}]$ the compare a with $S[\mathit{next} + \sqrt{n}]$, $S[\mathit{next} + 2\sqrt{n}]$, $\ldots$, until an i is found with $a \le S[\mathit{next} + (i-1)\sqrt{n}]$. This will use up i comparisons. If $a < S[\mathit{next}]$ then we proceed analogously. In any case, the subtable of size $\sqrt{n}$ thus found is then searched by applying the same method recursively.

We must determine the expected number C of comparisons required to determine the subtable of size $\sqrt{n}$. Let p_i be the probability that i or more comparisons are required for finding the subtable. Then

$$C = \sum_{i \ge 1} i \cdot (p_i - p_{i+1}) = \sum_{i \ge 1} p_i .$$

We still have to estimate p_i. Note first that $p_1 = p_2 = 1$ since two comparisons are always required. So let $i > 2$. If i or more comparisons are needed then

$$|\text{actual rank of } a - \mathit{next}| \ge (i-2)\sqrt{n}$$

where the "actual rank of a" denotes the number of x_j's smaller than a. Hence

$$p_i \;\le\; \text{prob}\big(|\text{actual rank of } a - \mathit{next}| \ge (i-2)\sqrt{n}\big).$$

We use Chebyshev's inequality (cf. W. Feller, "An Introduction to Probability Theory, and its Applications", *John Wiley, New York* 1968) to derive a bound on p_i: Sie lautet:

$$\text{prob}\big(|X-\mu| \ge t\big) \;\le\; \sigma^2/t^2$$

for a random variable X with mean μ and variance σ^2. Let random variable X be the number of x_j's smaller than a. Recall, that we assume that $x_1,\ldots,x_n$ are drawn independently from a uniform distribution over (x_0, x_{n+1}). Then, since the x_i's are independent and since $p = (a-x_0)/(x_{n+1}-x_0)$ is the probability that any one of them is less than a, the probability that exactly j out of n are less than a is $\binom{n}{j}p^j(1-p)^{n-j}$. Thus the expected number of keys less than a, i.e., the expected rank of a is

$$\mu = \sum_{j=0}^{n} j\binom{n}{j}p^j(1-p)^{n-j} = p\cdot n\sum_{j=0}^{n}\binom{n-1}{j-1}p^{j-1}(1-p)^{n-j} = p\cdot n$$

with variance

$$\sigma^2 = \sum_{j=0}^{n}(j-\mu)^2\cdot\binom{n}{j}p^j(1-p)^{n-j} = p(1-p)n.$$

fehlt i.E. Um die letzte Gleichung zu verifizieren, schreibt man $(j-\mu)^2 = j^2 - 2j\mu + \mu^2$ und behandelt dann die 3 Summen nach dem bei der Berechnung von μ benutzten Schema. Thus

$$\begin{aligned}
p_i &\le \text{prob}\big(|\text{actual rank of } a - p\cdot n| \ge (i-2)\sqrt{n}\big)\\
&\le \frac{p(1-p)n}{((i-2)\sqrt{n})^2}\\
&= \frac{p(1-p)}{(i-2)^2}\\
&\le \frac{1}{4(i-2)^2}
\end{aligned}$$

since $p(1-p) \le 1/4$ for $0 \le p \le 1$. Substituting into our expression for C yields

$$C \le 2 + \sum_{i\ge 3}\frac{1}{4(i-2)^2} = 2 + \pi^2/24 \approx 2.4.$$

Finally, let $\overline{T}(n)$ be the average number of comparisons used in searching a random table of n keys for a. Since the subtables of size $\sqrt{n}$ are again random, we have

$$\overline{T}(n) \le C + \overline{T}(\sqrt{n}) \qquad \text{for } n \ge 3$$

and

$$\overline{T}(1) \le 1,\ \overline{T}(2) \le 2$$

and thus

$$\overline{T}(n) \le 2 + C \log\log n \qquad \text{for } n \ge 2,$$

as is easily shown by induction on n.

Theorem 1. *The expected cost of quadratic binary search is $O(\log\log n)$.* ▮

The worst case access time of quadratic binary search is $O(\sqrt{n})$. Note that the maximal number of comparisons used is $n^{1/2} + n^{1/4} + n^{1/8} + \cdots = O(\sqrt{n})$. This worst case behavior can be reduced to $O(\log n)$ as follows without sacrificing the $O(\log\log n)$ expected behavior. Instead of using linear search to determine the i with $S[next + (i-2)\sqrt{n}] < a \le S[next + (i-1)\sqrt{n}]$ with i comparisons we use exponential + binary search (cf. III.4.2, Theorem 11) for a cost of only $O(\log i)$ comparisons. Since $\log i \le i$ the expected behavior remains the same. However, the maximal number of comparisons is now $\log n^{1/2} + \log n^{1/4} + \log n^{1/8} + \ldots = (1/2 + 1/4 + 1/8 + \ldots)\log n = \log n$. Thus worst case behavior is $O(\log n)$ if exponential + binary search are used.

3.4. Weighted Trees

i.D. Einfüge(x, S)? In this section we consider operation Access applied to weighted sets S. We associate a weight (access probability with each element of S. Large weight indicates that the element is important and accessed frequently; it is desirable that these elements are high in the tree and can therefore be accessed fast.

Definition: Let $S = \{x_1 < x_2 < \cdots < x_n\}$ and let β_i bzw. α_j be the probability of operation Access(a, S) where $a = x_i$ $(x_j < a < x_{j+1})$ for $1 \le i \le n$ $(0 \le j \le n)$. Dabei sind x_0 und x_{n+1} zusätzliche künstliche Elemente. Then $\beta_i \ge 0$, $\alpha_j \ge 0$ and $\sum \beta_i + \sum \alpha_j = 1$. The $(2n+1)$-tuple $(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n)$ is called **access (probability) distribution**. ▮

Let T be a search tree for set S, let b_i^T be the depth of node x_i and let a_j^T be the depth of leaf (x_j, x_{j+1}). Consider a search for element a of the universe. If $a = x_i$ then we compare a with $b_i^T + 1$ elements in the tree, if $x_j < a < x_{j+1}$ then we compare a with a_j^T elements in the tree. Hence

$$P^T = \sum_{i=1}^{n} \beta_i \cdot (1 + b_i^T) + \sum_{j=0}^{n} \alpha_j \cdot a_j^T$$

is the average number of comparisons in a search. P^T is called (normalized) **weighted path length** of tree T. We take P^T as the basic measure for the mittlere?? efficiency of operation Access; the actual

We will suppress index T if tree T can be inferred from the context. Figure 22 shows a search tree for set $S = \{x_1, x_2, x_3, x_4\}$ with $(\alpha_0, \beta_1, \dots, \beta_4, \alpha_4) = (1/6, 1/24, 0, 1/8, 0, 1/8, 1/8, 0, 5/12)$. In this tree $b_1 = 1$, $b_2 = 2$, $b_3 = 0$, $b_4 = 1$, $a_0 = a_3 = a_4 = 2$ and $a_1 = a_2 = 3$. Weighted path length is equal to 2. There is no search tree for set S with smaller weighted path length; the tree shown in Figure 22 is optimal with respect to the given access distribution.

figure does not exist yet

Figure 22. Optimaler Suchbaum für Beispiel

3.4.1. Optimum Weighted Trees, Dynamic Programming and Pattern Matching

With every search tree for set S we associated a single real number, its weighted path length. We can therefore consider the tree with the minimal weighted path fehlt i.D. length. This tree will then also optimize average access time.

Definition: Let $S = \{x_1 < x_2 < \cdots < x_n\}$ be a set and let $(\alpha_0, \beta_1, \alpha_1, \dots, \beta_n, \alpha_n)$ be an access distribution. Tree T is an **optimum binary** search tree for set S if its weighted path length is minimal among all search trees for set S. We use T_{opt} to denote an optimum binary search tree and P_{opt} to denote its weighted path length. ∎

Theorem 1. *An optimum binary search tree for set S and distribution* $(\alpha_0, \beta_1, \alpha_1, \dots, \beta_n, \alpha_n)$ *can be constructed in time* $O(n^2)$ *and space* $O(n^2)$.

Proof: We use dynamic programming, i.e., we will construct optimal solutions for increasingly larger subproblems in a systematic way. In our case this means the construction of optimum search trees for all pairs, triples, ... of adjacent nodes. A search tree tree for set S has nodes $x_1, \dots, x_n$ and leaves $(x_0, x_1), \dots, (x_n, x_{n+1})$. A subtree might have nodes $x_i, \dots, x_j$ and leaves $(x_{i-1}, x_i), \dots, (x_j, x_{j+1})$ with $1 \le i, j \le n$, $i \le j + 1$. Such a subtree is a search tree for set $\{x_i, \dots, x_j\}$ with respect to the universe (x_{i-1}, x_{j+1}). The access probabilities innerhalb eines Unterbaumes are given by the conditional probabilities

$$\overline{\beta}_k = \beta_k / w_{ij} \qquad \text{und} \qquad \overline{\alpha}_h = \alpha_h / w_{ij}$$

where

$$w_{ij} = \alpha_{i-1} + \beta_i + \cdots + \beta_j + \alpha_j, \ i \le k \le j \text{ and } i - 1 \le h \le j.$$

for i **from** 0 **to** n
do $w_{i+1,i} \leftarrow \alpha_i$; $\overline{P}_{i+1,i} \leftarrow 0$ **od**;
for k **from** 0 **to** $n-1$
do for i **from** 1 **to** $n-k$
    do $j \leftarrow i+k$;
        $w_{ij} \leftarrow w_{i,j-1} + \beta_j + \alpha_j$;
        let m, $i \le m \le j$, such that $\overline{P}_{i,m-1} + \overline{P}_{m+1,j}$ is minimal;
        in case of ties choose the largest such m;
        $r_{ij} \leftarrow m$;
        $\overline{P}_{ij} \leftarrow w_{ij} + \overline{P}_{i,m-1} + \overline{P}_{m+1,j}$
    od
od.

Program 5

We use T_{ij} to denote an optimum binary search tree for set $\{x_i, \ldots, x_j\}$ with access distribution $(\overline{\alpha}_{i-1}, \overline{\beta}_i, \ldots, \overline{\beta}_j, \overline{\alpha}_j)$. Let P_{ij} be the weighted path length of T_{ij} and let r_{ij} be the index of the root of T_{ij}, i.e., x_m with $m = r_{ij}$ is the root of T_{ij}.

Lemma 1.

a) $P_{i,i-1} = 0$

b) $w_{ij}P_{ij} = w_{ij} + \min\limits_{i \le m \le j}(w_{i,m-1}P_{i,m-1} + w_{m+1,j}P_{m+1,j})$ *for* $i \le j$.

Proof: a) $T_{i,i-1}$ consists of a single leaf (x_{i-1}, x_i) which has depth zero. Thus $P_{i,i-1} = 0$.

b) Let T_{ij} be an optimum tree for set $\{x_i, \ldots, x_j\}$. Then T_{ij} has weighted path length P_{ij} and root x_m with $m = r_{ij}$. Let T_l (T_r) be the left (right) subtree of T_{ij} with weighted path length P_l (P_r). Since

$$\begin{aligned} b_k^{T_{ij}} &= 1 + b_k^{T_l} && \text{for} \quad i \le k \le m-1, \\ b_m^{T_{ij}} &= 0 && \text{and} \\ b_k^{T_{ij}} &= 1 + b_k^{T_r} && \text{for} \quad m+1 \le k \le j \end{aligned}$$

and analogously for the leaf weights we have

$$w_{ij}P_{ij} = w_{ij} + w_{i,m-1}P_l + w_{m+1,j}P_r.$$

T_l is a search tree for set $x_i, \ldots, x_{m-1}$ and therefore $P_l \ge P_{i,m-1}$. We must have $P_l = P_{i,m-1}$ because otherwise we could replace T_l by $T_{i,m-1}$ and obtain a tree with smaller weighted path length. This proves b). ∎

Lemma 1 suggests a way for computing all values $w_{ij}P_{ij}$. Initialization is given in part a) and the iterative step is given in part b). In the complete program below we use $\overline{P}_{ij}$ to denote $w_{ij}P_{ij}$.

Program 5 is a direct implementation of the recursion formula of Lemma 1. On termination the optimum search tree is given implicitly by the array r_{ij}. Node x_m with $m = r_{1,n}$ is the root of the search tree. The root of the left subtree is x_k with $k = r_{1,m-1}, \ldots$. It is easy to see that the search tree can be explicitly constructed in time $O(n)$ from array r_{ij}.

So let us consider the complexity of the algorithm above. The program uses three arrays $\overline{P}_{ij}$, w_{ij} and r_{ij} and hence has space complexity $O(n^2)$. We next turn to time complexity. Note first that one execution of lines (5), (6), (8), (9) takes time $O(1)$ and one execution of line (7) takes time $O(j-i+1) = O(k+1)$ for fixed i and k. Thus the total running time is

$$\sum_{k=0}^{n-1} \sum_{i=1}^{n-k} O(k+1) = O(n^3).$$

This falls short of the $O(n^2)$ bound promised in the theorem. Before we sketch an improvement we illustrate the algorithm on the example of the beginning of 3.4. Arrays $\overline{P}_{ij}$, w_{ij}, $1 \le i \le 5$, $0 \le j \le 4$ and r_{ij}, $1 \le i \le 4$, $1 \le j \le 4$ are given in Figure 23.

$$\overline{P} = \frac{1}{24} \cdot \begin{pmatrix} 0 & 5 & 11 & 25 & 48 \\ & 0 & 3 & 12 & 31 \\ & & 0 & 6 & 22 \\ & & & 0 & 13 \\ & & & & 0 \end{pmatrix} \qquad 24w = \frac{1}{24} \cdot \begin{pmatrix} 4 & 5 & 8 & 14 & 24 \\ & 0 & 3 & 9 & 19 \\ & & 0 & 6 & 16 \\ & & & 3 & 13 \\ & & & & 10 \end{pmatrix}$$

$$r = \begin{pmatrix} 1 & 1 & 3 & 3 \\ & 2 & 3 & 4 \\ & & 3 & 4 \\ & & & 4 \end{pmatrix}$$

Figure 23. $\overline{P}_{ij}, w_{ij}$, und r_{ij} für das Beispiel aus III.4

We learn one important fact from this example: Matrix r is monotone in each row and column, i.e., $r_{i,j-1} \le r_{i,j} \le r_{i+1,j}$ for all i, j. We postpone the proof of this fact for a while (cf. Lemma 3 below). The monotonicity of r has an important consequence; we may change line (7), the search for r_{ij}, into

let m, $r_{i,j-1} \le m \le r_{i+1,j}$ be such ...

without affecting the correctness of the algorithm. (Für $k = 0$ ist die Suche nach m sowieso trivial.) However, the change does have a dramatic effect on the running

i.D.mehr

time. It is now

$$\begin{aligned} & O\Big(n + \sum_{k=1}^{n-1}\sum_{i=1}^{n-k}(1 + r_{i+1,i+k} - r_{i,i+k-1})\Big) \\ = \; & O\Big(n + \sum_{k=1}^{n-1}(n - k + r_{n-k+1,n} - r_{1,k})\Big) \\ = \; & O\Big(n + \sum_{k=1}^{n-1} n\Big) \\ = \; & O(n^2). \end{aligned}$$

This proves Theorem 1. ∎

We still have to justify the monotonicity of r. We will do so in a more general context: Dynamic programming with quadrangle inequalities.

Let $w(i,j) \in \mathbb{R}$ for $0 \le i < j \le n$ and let $c(i,j)$ be defined by

$$c(i,i) = w(i,i) = 0 \qquad \text{and}$$
$$c(i,j) = w(i,j) + \min_{i<k\le j}\big(c(i,k-1) + c(k,j)\big) \quad \text{for } i < j.$$

Optimum binary search trees are a special case of these recursion equations: take
i.E.mehr $w(i,j) = w_{i+1,j} = \alpha_i + \beta_{i+1} + \cdots + \beta_j + \alpha_j$; then $c(i,j) = \overline{P}_{i+1,j}$.

The algorithm given in Program 5 also allows us to compute $c(i,j)$, $1 \le i < j \le n$, in time $O(n^3)$. However, there is a faster way of computing $c(i,j)$ if function $w(i,j)$ satisfies the quadrangle inequality

(QI) $$w(i,j) + w(i',j') \le w(i',j) + w(i,j') \quad \text{for } i \le i' \le j \le j'.$$

Theorem 2. *If w satisfies* (QI) *and is monotone with respect to set inclusion of*
$\le w(i',j)$? *intervals, i.e., $w(i,j') \ge w(i',j)$ if $i \le i' < j \le j'$, then function c as defined above can be computed in time $O(n^2)$.*

Remark: Before we give a proof of Theorem 2 we apply it to optimum binary search trees. Function $w(i,j) = w_{i+1,j}$ is obviously monotone and satisfies (QI), in fact with equality.

Proof: Theorem 2 is proven by establishing the following two lemmas.

Lemma 2. *If w satisfies* (QI) *and is monotone then function c defined above also satisfies* (QI), *i.e., $c(i,j) + c(i',j') \le c(i,j') + c(i',j)$ for $i \le i' \le j \le j'$.*

We use $c_k(i,j)$ to denote $w(i,j) + c(i,k-1) + c(k,j)$ a we define $K(i,j) = \max\{k;\ c_k(i,j) = c(i,j)\}$ for $i < j$. Also $K(i,i) = i$. Then $K(i,j)$ is the largest index where the minimum is achieved in the definition of $c(i,j)$. Then

Lemma 3. *If c satisfies* (QI) *then K is monotone, i.e.,*

$$K(i,j) \le K(i,j+1) \le K(i+1,j+1) \quad \text{for } i \le j.$$

Lemma 3 is the key for improving the running time of dynamic programming to $O(n^2)$ as we have seen above. Lemmas 2 and 3 remains to be proven.

Proof: (Lemma 2). We use induction on the "length" $l = j' - i$ of the quadrangle inequality for c

$$\text{(QIc)} \qquad c(i,j) + c(i',j') \le c(i,j') + c(i',j) \quad \text{for } i \le i' \le j \le j'.$$

This inequality is trivially true if $i = i'$ or $j = j'$. This proves (QIc) for $l \le 1$. For the induction step we have to distinguish two cases: $i' = j$ or $i' < j$.

Case 1: $i < i' = j < j'$.
In this case (QIc) reduces to

$$c(i,j) + c(j,j') \le c(i,j'),$$

an (inverse) triangle inequality. Let $k = K(i,j')$. We must distinguish two symmetric subcases: $k \le j$ or $k > j$.

Case 1.1: $k \le j$.
We have $c(i,j') = w(i,j') + c(i,k-1) + c(k,j')$ and therefore

$$\begin{aligned} c(i,j) + c(j,j') &\le w(i,j) + c(i,k-1) + c(k,j) + c(j,j') \\ &\qquad\qquad \text{(definition of } c(i,j)\text{)} \\ &\le w(i,j') + c(i,k-1) + c(k,j) + c(j,j') \\ &\qquad\qquad \text{(monotonicity of } w\text{)} \\ &\le w(i,j') + c(i,k-1) + c(k,j') \\ &\qquad\qquad \text{(inverse triangle inequality for } k \le j \le j' \text{ nach I.A.)} \\ &= c(i,j'). \end{aligned}$$

Case 1.2: $k > j$.
This is symmetric to Case 1.1 and left to the reader.

Case 2: $i < i' < j < j'$.
Let $y = K(i',j)$ and $z = K(i,j')$. Again we have to distinguish two symmetric cases: $z \le y$ and $z > y$. We only treat the case $z \le y$. Note first that $z \le y \le j$ by

definition of y and $i < z$ by definition of z. We have

$$\begin{aligned}
c(i',j') + c(i,j) &\le c_y(i',j') + c_z(i,j) \\
&= w(i',j') + c(i',y-1) + c(y,j') + w(i,j) + c(i,z-1) + c(z,j) \\
&\le w(i,j') + w(i',j) + c(i',y-1) + c(i,z-1) + c(z,j) + c(y,j') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by (QI) for } w) \\
&\le w(i,j') + w(i',j) + c(i',y-1) + c(i,z-1) + c(y,j) + c(z,j') \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the induction hypothesis, i.e.,} \\
&\qquad\qquad\qquad\qquad\qquad \text{(QIc) angewandt auf } z \le y \le j \le j') \\
&= c(i,j') + c(i',j) \qquad\qquad\qquad \text{(by definition of } y \text{ and } z).
\end{aligned}$$

This complete the induction step and thus proves Lemma 2. ∎

Proof: (Lemma 3). Lemma 3 is trivially true when $i = j$, and so we only have to consider $i < j$. We will only prove $K(i,j) \le K(i,j+1)$. Recall that $K(i,j)$ is the largest index where the minimum is assumed in the definition of $c(i,j)$. It therefore suffices to show

$$\left[c_{k'}(i,j) \le c_k(i,j)\right] \Rightarrow \left[c_{k'}(i,j+1) \le c_k(i,j+1)\right]$$

for all $i < k \le k' \le j$, i.e., if $K(i,j)$ prefers k' over k then so does $K(i,j+1)$. In fact, we will show the stronger inequality

$$c_k(i,j) - c_{k'}(i,j) \le c_k(i,j+1) - c_{k'}(i,j+1)$$

or equivalently

$$c_k(i,j) + c_{k'}(i,j+1) \le c_{k'}(i,j) + c_k(i,j+1)$$

or equivalently by expanding all four terms using their definition

$$c(k,j) + c(k',j+1) \le c(k',j) + c(k,j+1).$$

However, this is simply the (QIc) at $k \le k' \le j \le j+1$. Damit sind Lemma 3 und schließlich Satz 2 bewiesen. ∎ ∎

Dynamic programming is a very versatile problem solving method. The reader finds many applications in Exercises 10–12 and in Sections 5.2 and 6.6.1. Dynamic programming is closely related to problem solving by backtracking. We illustrate the connection by two examples: optimum binary search trees and simulation 2-way deterministic pushdown automata on random access machines.

A call $OST(1, n)$ of the recursive procedure aus Programm 6 computes the cost fehlt i.E. of an optimum search tree

```
real function OST(i, j : integer);
   co OST computes the weighted path length
      of an optimum search tree for x_i, ..., x_j oc
if i = j + 1
then OST ← 0
else  OST ← ∞;
      for k from i to j
      do OST ← min(OST, w_ij + OST(i, k − 1) + OST(k + 1, j))
      od
fi
end.
```

Program 6

The running time of procedure OST is exponential. If $T(n)$ is the time required by OST for finding the optimum cost of a tree for a set of n elements then $T(n) = O(n) + \sum_{i=0}^{n-1}(T(i) + T(n-i-1)) = O(n) + 2\sum_{i=0}^{n-1} T(i)$. Subtracting the equations for $T(n+1)$ and $T(n)$ yields $T(n+1) - T(n) = O(1) + 2 \cdot T(n)$ and hence $T(n+1) = O(1) + 3 \cdot T(n)$. This shows that $T(n)$ grows exponentially.

Of course, the exponential running time of OST stems to the fact that subproblems $OST(i, j)$ are solved repeatedly. It is therefore a good idea to introduce a global table $P[i, j]$ and to record the values computed by OST in this table. This leads to Program 7.

```
real function OST(i, j : integer);
if P[i, j] is defined
then OST ← P[i, j]
else if i = j + 1
     then OST ← 0
     else  OST ← ∞
           for k from i to j
           do OST ← min(OST, w_ij + OST(i, k − 1) + OST(k + 1, j))
           od
     fi;
     P[i, j] ← OST
fi
end.
```

Program 7

Note that lines (4')–(11') are executed at most once for every pair (i,j), $i+1 \leq j$. Hence the total time spent in lines (4')–(11') of OST is $O(n^3)$. Die Gesamtzahl der Aufrufe von OST ist $O(n^3)$, weil OST nur in Zeile (8') rekursiv aufgerufen

fehlt i.E. wird. Also note that the total number of calls of OST is $O(n^3)$ since lines (4')–(11') are executed at most once for every pair (i,j). Thus the total time spent in lines (1')–(3') and (12')–(13') of OST is also $O(n^3)$.

We can see from this example that tabulating function values in exhaustive search algorithms can have a dramatic effect on the running time. In fact, the revised Program 7 is essentially our dynamic programming algorithm. The only additional idea required for the dynamic programming algorithm is the observation that recursion can be replaced by iteration.

We describe one more application of this approach. A **two-way deterministic pushdown automaton** consists of a finite set of states, an input tape with a two-way reading head and a pushdown store. Abbildung 24 zeigt das Gerippe eines

!! 2DPDA.

figure does not exist yet

Figure 24. Ein 2DPDA

Formally, a **2DPDA** is a 7-tuple $(S, A, B, F, q_0, C, \delta)$ consisting of a finite set S of states, an input alphabet A, a pushdown alphabet B, a set $F \subseteq S$ of accepting states, a start state $q_0 \in S$, a initial pushdown symbol $C \in B$ and a transition function

$$\delta : S \times B \times \big(A \cup \{\$, \not{C}\}\big) \mapsto S \times \big(\{\epsilon\} \cup B^2\big) \times \{-1, 1\}.$$

fehlt i.E. On input $a_1\ a_2\ \ldots\ a_n \in A^*$, wobei A^* die Menge aller Worte über A ist, the machine is started in state q_0 with $\$\ a_1\ \ldots\ a_n\ \not{C}$ on its input tape, its reading head on a_1 and with C in the pushdown store. The machine then operates as given by δ. More precisely, if the machine is in state $q \in S$, reads $a \in A \cup \{\$, \not{C}\}$, has βD, $\beta \in B^*$, $D \in B$ in its pushdown store and $\delta(q, D, a) = (q', \alpha, \Delta)$ then the new state is q', the new content of the pushdown store is $\beta\alpha$ and the input head is moved by Δ. Note that $\alpha \in \{\epsilon\} \cup B^2$. If $\alpha = \epsilon$ then the move is called a pop-move, if $\alpha \in B^2$ then the move is called a push-move. A 2DPDA is not allowed to leave the portion of the input tape delimited by \$ and $\not{C}$, i.e., $\delta(\ ,\ ,\$) = (\ ,\ ,+1)$ and $\delta(\ ,\ ,\not{C}) = (\ ,\ ,-1)$. A 2DPDA halts if it empties its pushdown store. It accepts if it halts in an accepting state.

Example: We describe a 2DPDA M which does pattern matching, i.e., it accepts all strings $a_1\ \ldots\ a_m\ \#\ b_1\ \ldots\ b_n$, $a_r, b_s \in \{0,1\}$ such that there is a j with $a_i = b_{j+i-1}$ for $1 \leq i \leq m$. In other words, M decides whether pattern $a_1\ \ldots\ a_m$ occurs in text $b_1\ \ldots\ b_n$. Program 8 shows how machine M operates. The correctness of this machine is implied by the following observation. Before step (2) the input head is on a_1 and the pushdown store contains $b_j b_{j+1}\ \ldots\ b_n$ with b_j at the top. If in step (2) und zu Schritt (3) kommen we move the input head to a_{i+1} then $a_h = b_{j+h-1}$ for $1 \leq h \leq i$ and $a_{i+1} \neq b_{j+i}$. Next observe, that $i = m$ implies that we found an occurrence of the pattern and that $j+i=n+1$ implies that the text is too short to contain an occurrence of the pattern. In step (5) we push $b_{j+i-1}, b_{j+i-2}, \ldots, b_j$ in that order onto the pushdown store, thus restoring the initial configuration. In (6) we remove b_j from the stack and then search for an occurrence of the pattern starting at b_{j+1}. ∎

---

(1) move the head to the right endmarker $\not{C}$;
 then move left and store $b_n, b_{n-1}, \ldots, b_1$ in the pushdown store;
 b_1 is at the top of the pushdown store; position the input head on a_1;
(2) **while** the symbol under the input head and
  the top pushdown symbol agree
 do move the input head to the right and
  delete one symbol from the pushdown store
 od;
(3) **if** the symbol under the input head is #
 then empty the pushdown store and accept **fi**;
(4) **if** the top symbol on the pushdown store is C (the special symbol)
 then empty the pushdown store and reject **fi**;
(5) Move the input head to a_1. While moving left push
 all symbols scanned onto the pushdown store
(6) Remove one symbol from the pushdown store and go to (2).

Program 8

Of course, a 2DPDA can be directly implemented on a RAM. Die Laufzeit des fehlt i.E. RAM-Programms ist dann gleich der Laufzeit des 2DPDA. The disadvantage of this approach is that the running time might be quite large. In our example, the worst case running time of the 2DPDA and hence the RAM is $O(m \cdot n)$. We can do much better.

Theorem 3. *If a language L is accepted by a 2DPDA then there is a RAM which accepts L in linear time.*

Proof: We will first define a recursive procedure which accepts L and then improve its running time by tabulating function values.

Let $M = (S, A, B, F, q_0, C, \delta)$ be a 2DPDA and let $a_1\ a_2\ \ldots\ a_n \in A^*$ be an input. A triple (q, D, i) with $q \in S$, $D \in B$ and $0 \le i \le n+1$ is called surface configuration of M; here i denotes the position of the input head on the input tape. Note that the number of surface configurations on input $a_1\ \ldots\ a_n$ is $|S| \cdot |B| \cdot (n+2) = O(n)$. Define partial function $\mathit{term} : S \times B \times [0\,..\,n+1] \mapsto S \times [0\,..\,n+1]$ by $\mathit{term}(q, D, i) = (p, j)$ if M started in configuration (q, D, i) will eventually empty its pushdown store and is in state p and input position j in this case.

Next note that $term(q_0, C, 1) = (p, j)$ for some $p \in F$ und ein $j \in \{0, \ldots, n+$ fehlt i.E. $1\}$ $\iff$ M accepts $a_1\ a_2\ \ldots\ a_n$. Zur Simulation von M auf einer RAM it thus suffices to compute *term*. The following (inefficient) recursive procedure *TERM* aus Programm 9 computes *term*.

```
(1)  function TERM(q, D, i);
(2)  if δ(q, D, a_i) = (p, ε, Δ)
(3)  then TERM ← (p, i + Δ)
(4)  else  let δ(q, D, a_i) = (p, GH, Δ), where G, H ∈ B;
(5)        (r, j) ← TERM(p, H, i + Δ);
(6)        TERM ← TERM(r, G, j)
(7)  fi
(8)  end.
```

Program 9

The correctness of this program is easy to establish. Procedure *TERM* certainly computes *term* if line (3) applies. If lines (4)–(6) apply then the computation starts out of configuration (q, D, i) with a push move. When the pushdown store has length 1 for the next time, the machine is in state r and scans position j (line (5). The content of the pushdown store is G at this point. Therefore *term* is correctly computed in line (6). The running time of this program is equal to the running time of the underlying 2DPDA.

As in our previous example we observe that *TERM* may be called repeatedly for the same argument. This suggests to tabulate function values in a table which we call *TE*. In table *TE* we store terminators which were already computed and we store $*$ in order to indicate that a call of *TERM* is initiated but not yet completed. Initially, all entries of *TE* are undefined. We obtain Program 10.

Claim:

a) *The total running time of call $TERM(q_0, C, 1)$ is $O(n)$.*

b) *A call $TERM(q_0, C, 1)$ correctly computes $term(q_0, C, 1)$.*

Proof: a) Recall that the number of surface configuration is $O(n)$. Observe next, that lines (7')–(13') are executed at most once for each surface configuration. Hence the total time spent in lines (7')–(13') is $O(n)$ and the total number of calls of *TERM* i.E.mehr is $O(n)$. Thus the total time spent in lines (1')–(6'), (14')–(15') of *TERM* is also $O(n)$. This shows that the total running time is $O(n)$.

b) Observe first that if $TERM(q, D, i)$ returns pair (r, j) then $term(q, D, i) = (r, j)$. We still have to consider the case that the simulation stops in line (3'). This can only be the case if there is a call $TERM(q', D', i')$ which (indirectly) initiates call $TERM(q', D', i')$ before its own completion. In this case we detected an infinite loop in the computation of M.

Damit sind die Behauptung und Satz 3 bewiesen. ∎∎

```
function TERM(q, D, i);
if TE[q, D, i] = *
then halt and reject
        because the 2DPDA is in an infinite loop;
fi;
if TE[q, D, i] is defined
then TERM ← TE[q, D, i]
else  TE[q, D, i] ← *;
        if δ(q, D, a_i) = (p, ε, Δ)
        then TERM ← TE[q, D, i] ← (p, i + Δ)
        else  let δ(q, D, a_i) = (p, GH, Δ), where G, H ∈ B;
                (r, j) ← TERM(p, H, i + Δ);
                TERM ← TE[q, D, i] ← TERM(r, G, j)
        fi
fi
end.
```

Program 10

Theorem 3 has a very pleasant consequence. Pattern matching can be done in linear time on a RAM. Of course, the algorithm obtained by applying Theorem 3 to the 2DPDA described in the exercise above is quite involved and will be hard to be understand intuitively. We therefore give an alternative simple linear time algorithm for pattern matching next.

Let $a_1 \ \ldots \ a_n$ be a pattern and let $b_1 \ \ldots \ b_m$ be a text, $a_i, b_j \in \{0,1\}$. We want to find all occurrences in the text. Define $f : [1 \mathinner{..} n] \mapsto [0 \mathinner{..} n]$, the **failure function** for the pattern, by

$$f(i) = \max\{h < i;\ a_{i-k} = a_{h-k} \text{ for } 0 \leq k < h\}.$$

The significance of function f is as follows. Suppose that we started to match the pattern at position j of the text and succeeded up to position i of the pattern, i.e., $a_l = b_{j+l-1}$ for $1 \leq l \leq i$ and $a_{i+1} \neq b_{j+i}$ (cf. Fig. 25).

$$\begin{array}{ccccccccc}
\cdots\cdots & b_j & b_{j+1} & & \cdots\cdots & b_{j+i-1} & b_{j+i} & \cdots\cdots \\
 & \| & \| & & & \| & \nparallel & \\
 & a_1 & a_2 & & & a_i & a_{i+1} & \\
 & & & a_1 \cdots\cdots & & a_h & a_{h+1} &
\end{array}$$

Figure 25. Vergleich des Musters mit dem Text

At this point we can slide the pattern to the right and start matching at some later position in text b. If we move the pattern to the right such that a_h is below

b_{j+i-1} then a match can only succeed if we have $a_{i-k} = a_{h-k}$ for $0 \le k < h$. Thus the only sensible values that should be tried are $f(i)$, $f(f(i))$, ... We obtain Program 11. In this algorithm we assume that we added a special symbol a_{n+1} to the pattern which does not match any symbol in the text.

---

(1) $i \leftarrow 0$; $j \leftarrow 0$;
(2) **while** $j \le m$
(3) **do co** $a_1 \ldots a_i = b_{j-i+1} \ldots b_j$ and
        ($l > i$ implies $a_1 \ldots a_l \ne b_{j-l+1} \ldots b_j$) **oc**
(4)   **if** $a_{i+1} = b_{j+1}$
(5)   **then** $i \leftarrow i + 1$; $j \leftarrow j + 1$
(6)   **else** **if** $i = n$ **then** report match starting at b_{j-n+1} **fi**;
(7)   **if** $i = 0$
(8)   **then** $j \leftarrow j + 1$
(9)   **else** $i \leftarrow f(i)$
(10)   **fi**
(11)   **fi**
(12) **od**.

Program 11

Lemma 4. *Program 11 determines all occurrences of pattern $p = a_1 \ldots a_n$ in text $b_1 \ldots b_m$ in time $O(m)$.*

Proof: In order to improve the correctness it suffices to verify the loop invariant. It is certainly true initially, i.e., if $i = j = 0$. So assume the invariant is true before executing the loop body. $a_{i+1} = b_{j+1}$ then the invariant trivially holds after execution of the body, if $a_{i+1} \ne b_{j+1}$ and $i = 0$ then it also holds trivially and if $i > 0$ then it holds by definition of f.

The bound on the running time is shown as follows. At each iteration either line (5), line (8) or line (9) is executed. Lines (5) and (8) increase j and are therefore executed together exactly $m + 1$ times. In line (9) the value of i is decreased since $f(i) < i$. Since i is only increased in line (5) and $i \ge 0$ always we conclude that line (9) is executed at most $m + 1$ times. ∎

Lemma 4 shows that pattern matching requires linear time if failure function f ia available. Fortunately, f can be computed in time $O(n)$. In fact, the same algorithm can be used because f is the result of matching the pattern against itself. Programm 12 tut dies.

Lemma 5. *Program 12 computes failure function f in time $O(n)$.*

Proof: Similar to the proof of Lemma 4. ∎

We summarize in

$i \leftarrow 0$; $j \leftarrow 1$; $f(1) \leftarrow 0$;
while $j \leq n$
do co $a_1 \dots a_i = a_{j-i+1} \dots a_j$ and
  $(i < l < j$ implies $a_1 \dots a_l \neq a_{j-l+1} \dots a_j)$ **oc**
  if $a_{i+1} = a_{j+1}$
  then $f(j+1) \leftarrow i+1$; $i \leftarrow i+1$; $j \leftarrow j+1$
  else **if** $i = 0$
    then $f(j+1) \leftarrow 0$; $j \leftarrow j+1$
    else $i \leftarrow f(i)$
    fi
  fi
od.

Program 12

Theorem 4. *All occurrences of pattern $a_1 \ \dots \ a_n$ in string $b_1 \ \dots \ b_m$ can be found in linear time $O(n+m)$.*

Proof: follows immediately from Lemmas 4 and 5. ∎

The dynamic programming algorithm for optimum binary search trees has quadratic space and time requirement. It can therefore only be used for small and medium sized n. In the next section we will discuss algorithms which construct nearly optimal binary search trees in linear time. Note hat giving up optimality is really not that bad because usually access probabilities are only known approximately anyhow. There is one further advantage of nearly optimal trees. We will be able to bound the cost of every single access operation and not only the expected cost. Recall that average and worst case behavior can differ by large amounts, i.e., in Quicksort; a similar situation could arise here.

We will be able to directly relate weighted path length of optimum trees and nearly optimal trees. Rather we compare both of them to an independently yardstick, the entropy of the access distribution.

Definition: Let $(\gamma_1, \dots, \gamma_n)$ be a discrete probability distribution, i.e., $\gamma_i \geq 0$ and $\sum \gamma_i = 1$. Then

$$H(\gamma_1, \dots, \gamma_n) = -\sum_{i=1}^{n} \gamma_i \log \gamma_i$$

is called the **entropy** of the distribution. We use the convention $0 \cdot \log 0 = 0$. ∎

Some basic properties of the entropy function can be found in the appendix. In the sequel we consider a fixed search tree T for set $S = \{x_1, \dots, x_n\}$ and access distribution $(\alpha_0, \beta_1, \dots, \beta_n, \alpha_n)$. We use H to denote the entropy of the distribution, i.e., $H = H(\alpha_0, \beta_1, \dots, \beta_n, \alpha_n)$. and we use b_i (a_j) for the depth of node x_i (leaf

(x_j, x_{j+1})) for $1 \le i \le n$ ($0 \le j \le n$). P is the weighted path length of T. We will prove lower bounds on the search times in tree T. These lower bounds are independent of the structure of T; in particular, they are valid for the optimum tree. The proofs follow classical proofs of the noiseless coding theorem.

Lemma 6. *Let $c \in \mathbb{R}$ with $0 \le c < 1$. Let*

$$\overline{\beta}_i = ((1-c)/2)^{b_i} \cdot c,\ 1 \le i \le n$$
$$\overline{\alpha}_j = ((1-c)/2)^{a_j}\ ,\ 0 \le j \le n.$$

Then $\overline{\beta}_i$, $\overline{\alpha}_j \ge 0$ and $\sum \overline{\beta}_i + \sum \overline{\alpha}_j = 1$, i.e., $(\overline{\alpha}_0, \overline{\beta}_1, \dots, \overline{\beta}_n, \overline{\alpha}_n)$ is a probability distribution.

Proof: (By induction on n). If $n = 0$ then $a_0 = 0$ and hence $\overline{\alpha}_0 = 1$. So assume $n > 0$. Let x_k be the root of T, T_l (T_r) the left (right) subtree of T. Let b'_i (b''_i), $1 \le i \le k-1$ ($k+1 \le i \le n$) be the depth of x_i in tree T_l (T_r). Define a'_j and a''_j analogously. Then

$$b_i = \begin{cases} b'_i + 1 & \text{for } 1 \le i \le k-1; \\ 0 & \text{for } i = k; \\ b''_i + 1 & \text{for } k+1 \le i \le n \end{cases}$$

und

$$a_j = \begin{cases} a'_j + 1 & \text{for } 0 \le j \le k-1; \\ a''_j + 1 & \text{for } k \le j \le n. \end{cases}$$

Also by induction hypothesis

$$S_l = \sum_{i=1}^{k-1} ((1-c)/2)^{b'_i} \cdot c + \sum_{j=0}^{k-1} ((1-c)/2)^{a'_j} = 1.$$

An analogous statement holds for S_r. Furthermore, $\overline{\beta}_k = c$. Thus

$$\sum_i \overline{\beta}_i + \sum_j \overline{\alpha}_j = ((1-c)/2)S_l + c + ((1-c)/2)S_r = 1. \qquad \blacksquare$$

Theorem 5. (Lower bound on weighted path length). *Let $B = \sum \beta_i$. Then*

a) $\max\{(H - d \cdot B)/\log(2 + 2^{-d});\ d \in \mathbb{R}\} \le P$;

b) $H \le P + B \cdot (\log e - 1 + \log(P/B))$.
where we use the notation as defined above.

Proof: Define $\overline{\beta}_i$ and $\overline{\alpha}_j$ is in Lemma 6. Then

$b_i + 1 = 1 + (\log \overline{\beta}_i - \log c)/\log \overline{c}$ and

$a_j = \log \overline{\alpha}_j / \log \overline{c}$

where $\overline{c} = (1-c)/2$. An application of property E1) of the entropy function (cf. appendix) yields (note that $\log \overline{c} < 0$)

$$\begin{aligned}
P &= \sum \beta_i \cdot (b_i + 1) + \sum \alpha_j \cdot a_j \\
&= B \cdot (1 - \log c / \log \overline{c}) + (1/\log \overline{c}) \cdot \left[\sum \beta_i \log \overline{\beta}_i + \sum \alpha_j \log \overline{\alpha}_j\right] \\
&\geq B \cdot (1 - \log c / \log \overline{c}) - (1/\log \overline{c}) \cdot H \\
&= (H - B \log(\overline{c}/c)) / \log(1/\overline{c}).
\end{aligned}$$

Setting $d = \log(\overline{c}/c)$ and observing that $c/\overline{c} = 2c/(1-c)$ ia surjective mapping from
$\mathbb{R}_0^+$?? $0 \leq c < 1$ onto the reals completes the proof of part a).

b) Unfortunately, there is no closed form expression for the value of d which maximizes the left side of a). Numerical methods have to be used to compute d_{max} in every single application. A good approximation for d_{max} is $d = \log(P/2B)$. It yields

$$\begin{aligned}
H &\leq P \log(2 + 2^{-d}) + d \cdot B \\
&= P \log(2 + 2B/P) + B \log(P/2B) \\
&\leq P \cdot (1 + (B/P) \log e) + B \cdot (\log(P/B) - 1) \qquad (\log x \leq (x-1) \log e) \\
&= P + B \cdot (\log e - 1 + \log P/B).
\end{aligned}$$

∎

Special case $d = 0$ is also useful in some occasions. It yields $P \geq H/\log 3$. We will next turn to the behavior of single access operations. Theorem 5a) reads in expanded form

$$\begin{aligned}
&\sum \beta_i \cdot \left[(-\log \beta_i - d)/\log(2 + 2^{-d})\right] + \sum \alpha_j \cdot \left[-\log \alpha_j / \log(2 + 2^{-d})\right] \\
&\leq \sum \beta_i \cdot \left[(b_i + 1)\right] + \sum \alpha_j \cdot \left[a_j\right].
\end{aligned}$$

We show that the inequality above is almost true componentwise for the expressions in square brackets; more precisely, for $h \in \mathbb{R}$, $h > 0$ define

$$\begin{aligned}
N_h &= \{i;\ (-\log \beta_i - d - h)/\log(2 + 2^{-d}) \geq b_i + 1\} \qquad \text{and} \\
L_h &= \{j;\ (-\log \alpha_j - h)/\log(2 + 2^{-d}) \geq a_j\}.
\end{aligned}$$

Then

$$\sum_{i \in N_h} \beta_i + \sum_{j \in L_h} \alpha_j \leq 2^{-h},$$

i.e., for a set of leaves and nodes, whose total weight exceeds $1 - 2^{-h}$ Theorem 5a) "almost" holds componentwise. "Almost" has to be interpreted as: up to the figure does not exist yet

Figure 26. Eine Zugriffsverteilung

additive factor $-h/\log(2+2^{-d})$. The proof of this claim is as follows: Let $d = \log(\overline{c}/c)$ with $\overline{c} = (1-c)/2$ and $0 \le c < 1$. Then $d = \log\overline{c} - \log c$ and $\log(2 + 2^{-d}) = \log(1/\overline{c})$. A simple computation shows that the definitions of N_h and L_h are equivalent to

$$N_h = \{i;\ \beta_i \le 2^{-h}\overline{\beta}_i\} \qquad \text{and}$$
$$L_h = \{j;\ \alpha_j \le 2^{-h}\overline{\alpha}_j\}$$

where $\overline{\beta}_i$ and $\overline{\alpha}_j$ are defined as in Lemma 6. Thus

$$\begin{aligned} 1 &= \sum \overline{\beta}_i + \sum \overline{\alpha}_j \\ &\ge \sum_{i\in N_h} \overline{\beta}_i + \sum_{j\in L_h} \overline{\alpha}_j \\ &\ge 2^h\Big[\sum_{i\in N_h} \beta_i + \sum_{j\in L_h} \alpha_j\Big]. \end{aligned}$$

We summarize in

Theorem 6. (Lower Bounds for single access operations). *Let $c, h \in \mathbb{R}$ with $0 \le c < 1$ and $h > 0$. Define $\overline{\beta}_i, \overline{\alpha}_j$ as in Lemma 6 and let*

$$N_h = \{i;\ \beta_i \le 2^{-h}\overline{\beta}_i\} \qquad \textit{and}$$
$$L_h = \{j;\ \alpha_j \le 2^{-h}\overline{\alpha}_j\}.$$

Then

$$\sum_{i\in N_h} \beta_i + \sum_{j\in L_h} \alpha_j \ \le\ 2^{-h}.$$

∎

We give an explicit example of Theorems 5 and 6 at the end of Section 3.4.2.

3.4.2. Nearly Optimal Binary Search Trees

In binary search (cf. Section 3.3.1) we always compare the argument of the access operation with the middle element of the remaining array and hence exclude at least the half of the set in every step of the search.

We deal now with the more general situation that elements have different weights (probabilities). At each step we should therefore try to exclude one half of the elements *in probability*. Figure 26 shows for the example from the beginning of 3.4 the distribution on the unit line.

figure does not exist yet

Figure 28. Case A

Point 1/2 lies within a β_i or a α_j. In the first case we should choose x_i as the root of the search tree, in the second case we should either choose x_j if 1/2 lies in the left half of α_j or x_{j+1} if 1/2 lies in the right half of α_j. In our example we choose x_3 as the root. This also fixes the right subtree. For the left subtree we still have to decide whether we choose x_1 or x_2 as the root.

Method 1: The restriction of our access distribution to set $\{x_1, x_2\}$ is given by $(\frac{1}{6}W, \frac{1}{24}W, 0, \frac{1}{8}W, 0)$ where $W = (\frac{1}{6} + \frac{1}{24} + \frac{1}{8})^{-1}$. We proceed as described above and look for the root of the left subtree. In this way we will choose x_1 as the root of the left subtree. Method 1 is analyzed in Exercise 20.

figure does not exist yet

Figure 27. Durch Verfahren 2 konstruierter Suchbaum

Method 2: We proceed by strict bisection, i.e., we choose the root of the left subtree by considering reference point 1/4. Point 1/4 is contained in β_2 and therefore x_2 is chosen as the root of the left subtree. In this way the following tree of Figure 27 with weighted path length $P_{BB} = 50/24$ is constructed. We now describe method 2 in more detail. Let

$$\begin{aligned} s_0 &= \frac{\alpha_0}{2} \\ s_i &= s_{i-1} + \frac{\alpha_{i-1}}{2} + \beta_i + \frac{\alpha_i}{2} \qquad \text{for } 1 \le i \le n. \end{aligned}$$

Then a call *construct_tree*$(0, n, 0, 1)$ of Program 13 constructs a search tree according to method 2.

---

procedure *construct_tree*(i, j, cut, l);

co we assume that the actual parameters of any call of *construct_tree* satisfy the following conditions:

(1) i and j are integers with $0 \le i < j \le n$;
(2) l is an integer with $l \ge 1$;
(3) $cut = \sum_{p=1}^{l-1} x_p 2^{-p}$ with $x_p \in \{0, 1\}$ for all p;
(4) $cut \le s_i \le s_j \le cut + 2^{-l+1}$.

A call *construct_tree*$(i, j, \ ,\ )$ will construct a binary tree for nodes $i+1, \ldots, j$ and leaves $i, \ldots, j$.

oc

begin
if $i + 1 = j$ (Case A)
then return tree of Figure 28

else determine k such that
(5) $i < k \le j$;
(6) $k = i+1$ or $s_{k-1} \le cut + 2^{-l}$;
(7) $k = j$ or $s_k \ge cut + 2^{-l}$.
oc k exists because the actual parameters are supposed to satisfy condition (4).
co
if $k = i+1$ (Case B)
then return tree of Figure 29 **fi**;
einbildCase B
if $k = j$ (Case C)
then return tree of Figure 29 **fi**;

figure does not exist yet

Figure 29. Case C

if $i+1 < k < j$ (Case D)
then return tree of Figure 30 **fi**;

figure does not exist yet

Figure 30. Case D

fi
end.

Program 13

Theorem 7. *Let b_i be the depth of node x_i and let a_j be the depth of leaf (x_j, x_{j+1}) in tree T_{BB} constructed by construct_tree$(0, n, 0, 1)$. Then*

$$b_i \le \lfloor \log 1/\beta_i \rfloor \quad \text{and} \quad a_j \le \lfloor \log 1/\alpha_j \rfloor + 2.$$

Proof: We state several simple facts.

Fact 1. *If the actual parameters of a call construct_tree(i, j, cut, l) satisfy conditions (1) to (4) and that $i+1 \ne j$, then a k satisfying conditions (5) to (7) exists and the actual parameters of the recursive calls of construct_tree initiated by this call again satisfy conditions (1) to (4).*

Proof: Assume that the parameters satisfy conditions (1) to (4) and that $i+1 \ne j$. In particular, $cut \le s_j \le cut + 2^{-l+1}$. Suppose, that there is no k, $i < k \le j$, with $s_{k-1} \le cut + 2^{-l}$ and $s_k \ge cut + 2^{-l}$. Then either for all k, $i < k \le j$, $s_k < cut + 2^{-l}$ or for all k, $i < k \le j$, $s_{k-1} > cut + 2^{-l}$. In the first case $k = j$ satisfies (6) and (7), in the second case $k = i + 1$ satisfies (6) and (7). This shows that k always exists. It remains to be shown that the parameters of the recursive calls satisfy again (1) to (4). This follows immediately from the fact that k satisfies (5) to (7) and that $i + 1 \neq j$ and hence $s_k \geq cut + 2^{-l}$ in Case B and $s_{k-1} \leq cut + 2^{-l}$ in Case C. ∎

Fact 2. *The actual parameters of every call of construct_tree satisfy conditions* (1) *to* (4) *(if the arguments of the top-level call do).*

Proof: The proof is by induction, Fact 1 and the observation that the actual pa-
i.D.anders rameters of the top-level call *construct_tree*$(0, n, 0, 1)$ satisfy (1) to (4). ∎

Beachten Sie, daß die aktuellen Parameter des Aufrufes *construct_tree*$(0, n, 0, 1)$ die Bedingungen (1)–(4) erfüllen und damit die Voraussetzung von Faktum 2 erfüllt ist.

We say that node h (leaf h resp.) is **constructed** by the call *construct_tree*(i, j, cut, l) if $h = j$ ($h = i$ or $h = j$) ist and Case A is taken or if $h = i + 1$ ($h = i$) and Case B is taken or if $h = j$ ($h = j$) and Case C is taken or if $h = k$ and Case D is taken. Let b_i be the depth of node i and let a_j be the depth of leaf j in the tree returned by the call *construct_tree*$(0, n, 0, 1)$.

Fact 3. *If node h (leaf h) is constructed by the call construct_tree(i, j, cut, l), then $b_h + 1 = l$ $(a_h = l)$.*

Proof: The proof is by induction on l. ∎

Fact 4. *If node h (leaf h) is constructed by the call construct_tree(i, j, cut, l), then $\beta_h \leq 2^{-l+1}$ $(\alpha_h \leq 2^{-l+2})$.*

Proof: The actual parameters of the call satisfy condition (4) by Fact 2. Thus

$$\begin{aligned} 2^{-l+1} &\geq s_j - s_i = (\alpha_i + \alpha_j)/2 + \beta_{i+1} + \alpha_{i+1} + \cdots + \beta_j \\ &\geq \beta_h \quad \text{resp.} \quad \alpha_h/2. \end{aligned}$$ ∎

We infer from Facts 3 and 4, $\beta_h \leq 2^{-b_h}$ and $\alpha_h \leq 2^{-a_h+2}$. Taking logarithms and observing that b_h and a_h are integers proves the theorem. ∎

Theorems 6 and 7 together give fairly detailed information about the tree constructed by procedure *construct_tree*. In particular, we have

$$b_i \approx \log 1/\beta_i \quad \text{and} \quad a_j \approx \log 1/\alpha_j$$

for most nodes and leaves of tree T_{BB}. Substituting the bounds on b_i and a_j given in Theorem 7 into the definition of weighted path length we obtain

Theorem 8. *Let P_{BB} be the weighted path length of the tree constructed by construct_tree. Then*

$$\begin{aligned} P_{BB} &\le \sum \beta_i \lfloor \log 1/\beta_i \rfloor + \sum \alpha_j \lfloor \log 1/\alpha_j \rfloor + \sum \beta_i + \sum \alpha_j + \sum \alpha_j \\ &\le H(\alpha_0, \beta_1, \ldots, \beta_n, \alpha_n) + 1 + \sum \alpha_j \end{aligned}$$ ∎

and further

Theorem 9. *Let P_{BB} be the weighted path length of the tree constructed by construct_tree for distribution $(\alpha_0, \beta_1, \ldots, \beta_n, \alpha_n)$ and let P_{opt} be the weighted path length of an optimum tree. Then ($B = \sum \beta_i$)*

a) $\max \left\{ \dfrac{H - d \cdot B}{\log(2 + 2^{-d})};\ d \in \mathbb{R} \right\} \le P_{opt} \le P_{BB} \le H + 1 + \sum \alpha_j;$

b) $P_{BB} \le P_{opt} + B \cdot (\log e + \log(P_{opt}/B)) + 2 \sum \alpha_j$.

Proof: a) follows immediately from Theorem 5a) and 8 and a) follows immediately from Theorem 5b) and 8. ∎

Theorem 9 can be interpreted in two ways. On the one hand it shows that the weighted path length P_{BB} of tree T_{BB} is always very close to the optimum and hence T_{BB} is a good search tree. Essentially, part b) shows that

$$P_{BB} - P_{opt} \le \log P_{opt} \approx \log H.$$

On the other hand, it provides us with a small interval containing P_{opt} as well as P_{BB}. This interval is easily computable from the distribution and provides us with a simple a-priori estimate of the behavior of search trees. This estimate can be exploited for the decision whether or not to use weighted trees. The bounds given in Theorems 5–9 are sharp (cf. Exercises 18–19).

Let us illustrate our bounds by an example. There is an extensive literature dealing with word frequencies in natural languages. In English, the probability of occurrence of the i-th most frequent word (cf. E.S. Schwartz, *JACM* 10 (1963), S. 413–439) is approximately

$$\beta_i = c/i^{1.12} \quad \text{where} \quad c = \frac{1}{\sum_{i \ge 1} (1/i)^{1.12}}.$$

A simple calculation yields

$$H(\beta_1, \beta_2, \beta_3, \ldots) = -\sum \beta_i \log \beta_i \approx 10.2.$$

In the light of Theorem 9 we would therefore expect that the weighted path length of an optimum binary search tree for *all* English words is about 10.2 and certainly no larger than 11.2. This was also observed in experiments. Gotlieb/Walker took a text of 10^6 words and counted word frequencies. Then they constructed (nearly optimal binary search trees for the N most common words, $N = 10$, 100, 1000, 10000, 100000. Let P_N be the weighted path length of the tree constructed for the N most common words. Then $P_N \to 11$ for $N \to \infty$ as Figure 31 (due to Gotlieb/Walker (72)) below suggests. This is in agreement with Theorem 9.

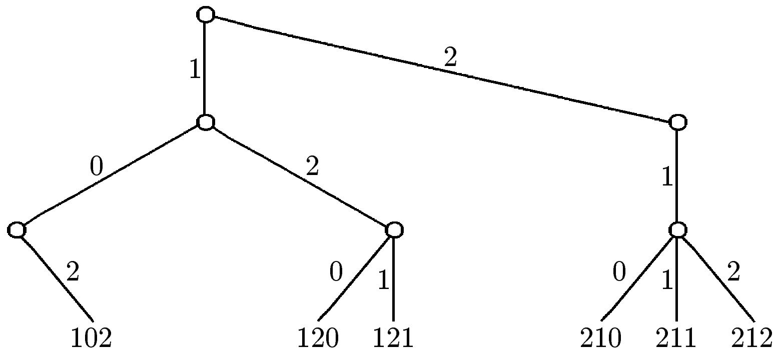

Figure 31. Wachstum von P_N

We will now turn to the time complexity of recursive procedure *construct_tree*. Let $T(n)$ be the maximal running time required by *construct_tree* for a tree with n nodes, i.e., $n = j - i$. If $n = 1$, then the body of *construct_tree* requires constant time. Hence

$$T(1) = c_1 \qquad \text{for some constant } c_1.$$

If $n > 1$ then k has to be determined and some recursive calls have to be initiated. Let $T_S(n,m)$ be the time required to find k, where $m = k-i$. We determine $T_S(n,m)$ below when we specify a concrete algorithm for finding k. *construct_tree* is called once or twice recursively within the body. In Case D the first call constructs a tree with $k-1-i = m-1$ nodes and the second call constructs a tree with $j-k = n-m$ nodes. Hence

$$T(n) \le \max_m \bigl[T(m-1) + T(n-m) + T_S(n,m) + c_2\bigr].$$

Constant c_2 measures the cost of parameter passing. If we define $T(0) = 0$ then the inequality above also holds true in Cases B and C of *construct_tree* where only one recursive call is started. With the convention $T_S(1,m) = 0$ and $c = \max(c_1, c_2)$ we can further simplify our inequality and obtain

$$\begin{aligned} &T(0) = 0 \qquad \text{and} \\ &T(n) \le \max_{1 \le m \le n} \bigl[T(m-1) + T(n-m) + T_S(n,m) + c\bigr]. \end{aligned}$$

We discuss two methods for finding k.

1) *Binary Search*: We first try $r = \lfloor (i+1+j)/2 \rfloor$. If $s_r \geq cut + 2^{-l}$ then $k \leq r$, otherwise $k \geq r$. We iterate as described in 3.3.1 and find k in $\log n$ steps. Thus $T_S(n,m) \leq d \log n$ for some constant d, and we obtain

$$T(0) = 0 \qquad \text{and}$$
$$T(n) \leq \max_{1 \leq m \leq n} \left[T(m-1) + T(n-m) + c + d \log n \right].$$

We infer $T(n) = O(n \log n)$, cf. Section II.1.3, conversely,

$$T(n) \geq T(n-1) + d \log n + c \geq d \sum_{i=1}^{n} \log i = \Omega(n \log n).$$

Also gilt

Theorem 10. *If the search for k in procedure construct_tree is implemented by binary search, then $T(n) = \Theta(n \log n)$.* ∎

2) *Exponential and Binary Search*: The running time of procedure *construct_tree* with binary search is $\Omega(n \log n)$ since we us up $\log(j-i)$ time units even if $k \approx i+1$ or $k \approx j$, i.e., even if the size of the problem to be solved is reduced only by a small amount. If we want to improve upon the $O(n \log n)$ time bound we have to use a search algorithm which finds k fast if k is close to the extremes. A first attempt is to use linear search and to start the search simultaneously at both ends. However, this also does not guarantee linear running time. If $k = (i+1+j)/2$ the this method will use $\Theta(j-i)$ steps to find k and again we obtain an $O(n \log n)$ algorithm (Exercise 21). So, how we can do better? We should start searching from the end but not in unit steps:

(1) Compare s_r with $cut + 2^{-l}$ for $r = \lfloor (i+1+j)/2 \rfloor$. If $s_r \geq cut + 2^{-l}$ then $k \in \{i+1, \ldots, r\}$, if $s_r \leq cut + 2^{-l}$ then $k \in \{r, \ldots, j\}$. We assume for the sequel that $k \in \{i+1, \ldots, r\}$. Step (1) has constant cost, say d_1.

(2) Find the smallest t, $t = 0, 1, 2, \ldots$, such that $s_{i+2^t} \geq cut + 2^{-l}$. Let t_0 be that value of t. We can find t_0 in time $d_2 \cdot (t_0 + 1)$ for some constant d_2. Then $i + 2^{t_0 - 1} < k \leq i + 2^{t_0}$, i.e., $2^{t_0} \geq k - i = m > 2^{t_0 - 1}$ and hence $\log m > t_0 - 1$. Thus the cost of step (2) is bounded by $d_2 \cdot (2 + \log m)$.

(3) Determine the exact value of k by binary search on the interval $i + 2^{t_0 - 1} + 1, \ldots, i + 2^{t_0}$. This takes $d_3 \cdot (\log(2^{t_0} - 2^{t_0 - 1}) + 1) = d_3 \cdot t_0 < d_3 \cdot (1 + \log m)$ time units for some constant d_3.

Exponential (step (2)) and binary search (step (3)) allow us to find k in $\leq d \cdot (1 + \log m)$ time units provided that $i < k \leq \lfloor (i+1+j)/2 \rfloor$. Here $m = k - i$ and d is a constant. Similarly, k can be found in $\leq d \cdot (1 + \log(n-m+1))$ time units if $\lfloor (i+1+j)/2 \rfloor < k$. Thus $T_S(n,m) \leq d \cdot (1 + \log \min(m, n-m+1))$ and we obtain the following recurrence relations for the worst case running time of *construct_tree*:

$$T(0) = 0 \qquad \text{and}$$
$$T(n) \leq \max_{1 \leq m \leq n} \left[T(m-1) + T(n-m) + d \cdot (1 + \log \min(m, n-m+1)) + c \right].$$

Theorem 11. *If the search for k in procedure construct_tree is implemented by exponential and binary search; then $T(n) = O(n)$.*

Proof: We show by induction on n:

$$T(n) \leq (2d+c)n - d \log(n+1).$$

This is certainly true for $n = 0$. For $n > 0$ we have

$$\begin{aligned} T(n) &\leq \max_{1 \leq m \leq n} \left[T(m-1) + T(n-m) + d \cdot (\log \min(m, n-m+1)) + d + c \right] \\ &= \max_{1 \leq m \leq (n+1)/2} \left[T(m-1) + T(n-m) + d \log m + d + c \right], \end{aligned}$$

by the symmetry of the expression in square brackets in $m-1$ and $n-m$. Next we apply the induction hypothesis and obtain

$$\begin{aligned} T(n) \leq & \max_{1 \leq m \leq (n+1)/2} \big[(2d+c) \cdot (m-1+n-m) - d \cdot (\log m + \log(n-m+1)) \\ & \qquad + d \log m + d + c \big] \\ &= (2d+c)n + \max_{1 \leq m \leq (n+1)/2} \left[-d \cdot (1 + \log(n-m+1) \right]. \end{aligned}$$

The expression in square brackets is always negative and is maximal for $m = (n+1)/2$. Thus

$$\begin{aligned} T(n) &\leq (2d+c)n - d \cdot (1 + \log((n+1)/2)) \\ &= (2d+c)n - d \log(n+1). \end{aligned}$$ ∎

Let us summarize. *construct_tree* constructs trees which are nearly optimal with respect to average search time (Theorem 9) as well as with respect to single search time (Theorems 6 and 7). We can make *construct_tree* run in linear time (Theorem 11).

We conclude this section by exemplifying Theorems 8 and 9 on the example of the beginning of Section 3.4. We start with Theorem 9 which concerns average search time. We have $H \approx 2.27$, $\sum \beta_i \lfloor \log 1/\beta_i \rfloor + \sum \alpha_j \lfloor \log 1/\alpha_j \rfloor \approx 2.04$, $\sum \beta_i \approx 0.29$ and $\sum \alpha_j \approx 0.71$. $d = 1.05$ maximizes the left hand side of 9a) and yields

$1.50 \le P_{opt} = 2.0 \le 2.04 \le 2.08 = P_{BB} \le 3.75 \le 3.98$. Of course, the additive constants play an almost dominating role in our bounds for that small value of H. We have seen in the application to an English dictionary that the estimates are much better for large values of H.

We will now turn to Theorems 6 and 7 on the behavior of single searches: We can see from the table shown in Figure 32 that the upper bounds given by Theorem 7 exceed the actual values by 1 or 2. If we apply Theorem 6 with $c = 1/2$ and $h = 2$ then $N_2 = \{3, 4\}$ and $L_2 = \{1, 2\}$ and

$$\sum_{i \in N_2} \beta_i + \sum_{j \in L_2} \alpha_j = 1/8 \le 1/4 = 2^{-h}.$$

Hence nodes and leaves with total probability $\ge 7/8$ satisfy Theorem 5a) componentwise (cf. the discussion preceding Theorem 6).

name of node or leaf	depth in T_{opt}	depth in T_{BB}	probability p	$\lfloor -\log p \rfloor$	$\overline{\beta}_i, \overline{\alpha}_j$ for $c = 1/2$
x_1	1	2	1/24	4	1/32
x_2	2	1	1/8	3	1/8
x_3	0	0	1/8	3	1/2
x_4	1	1	0	∞	1/8
$(\ , x_1)$	2	3	1/6	2	1/64
(x_1, x_2)	3	3	0	∞	1/64
(x_2, x_3)	3	2	0	∞	1/16
(x_3, x_4)	2	2	1/8	3	1/16
$(x_4, \ )$	2	2	5/12	1	1/16

Figure 32. Tabelle für Beispielwerte

3.5. Balanced Trees

We return to the discussion started in3.3.1: Realizing operations Access, Insert and Delete by binary search trees. In Section 3.3.1 we saw that the height of the tree plays a crucial role.

We consider only unweighted sets in this section, i.e., We are only interested in the size of the sets involved. In other words, if T is a search tree for set $S = \{x_1 < x_2 < \cdots < x_n\}$ then we assume uniform access probabilities, i.e., $\beta_i = 1/n$ and $\alpha_j = 0$ for $1 \le i \le n$, $0 \le j \le n$. As above, we use b_i to denote the depth of node x_i in T.

$$P = \frac{1}{n}\sum_i (b_i + 1)$$

i.E.mehr is called the **average path length** (internal path length) of T. Theorems 5b) and 6 of the previous section give $\log n = H \le P + \log P + 0.44$ and $\text{height}(T) \ge \log n - 1$. The second inequality is obtained by taking the limit $h \to 0$ in Theorem 6. Somewhat better bounds can be obtained by direct computation.

Theorem 1. *Let T be a binary search tree for set $S = \{x_1 < x_2 < \cdots < x_n\}$. Then*

a) $P \ge \lfloor \log(n+1) \rfloor - 1$.

b) $\text{height}(T) \ge \lceil \log(n+1) \rceil$.

Proof: b) Since T is a binary tree, there are at most 2^i nodes of depth i $(i \ge 0)$, and hence at most $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ nodes of depth $\le k$. In a tree with n nodes there must thus be at least one node of depth k where $2^{k+1} - 1 \ge n$. This proves b) since $\text{height}(T) = \max\{\text{depth}(v) + 1;\ v \text{ node of } T\}$.

a) Apparently, a tree T with n nodes has minimal average path length, if there are 2^0 node of depth 0, 2^1 nodes of depth 1, ... , 2^k nodes of depth k and $n - 2^{k+1} + 1$ nodes of depth $k+1$. Here $k = \lfloor \log(n+1) \rfloor - 1$. Thus

$$\begin{aligned} P &\ge \frac{1}{n}\left(\sum_{i=0}^{k} (i+1)2^i + (k+2)(n - 2^{k+1} + 1)\right) \\ &= \frac{1}{n}\left[k \cdot 2^{k+1} + 1 + (k+2)(n - 2^{k+1} + 1)\right] \qquad \text{(s. Anhang, S1)} \\ &\ge \lfloor \log(n+1) \rfloor - 1. \end{aligned}$$

∎

Theorem 1 shows that logarithmic behavior is the best we can expect from binary search trees in the worst case as well as in the average case. Also, logarithmic behavior is easy to obtain as long as we restrict ourselves to access operations. This is even true in the case of weighted sets as we saw in 3.4. Insertions and deletions create new problems; the naive insertion and deletions algorithms of 3.3.1

can create extremely unbalanced trees and thus lead to intolerable search times. Inserting $x_1, \ldots, x_n$ with $x_1 < x_2 < \cdots < x_n$ into an initially empty tree creates a
fehlt tree with average path length $(n+1)/2$, wie Abbildung 33 veranschaulicht.

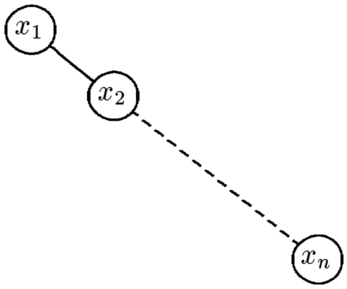

Figure 64. Entarteter binärer Suchbaum

Thus tree search deteriorates to linear search. Extreme deterioration is not very probable in the case of random insertions (Exercise 9). Deterioration can be completely avoided if the tree is rebalanced after each insertion or deletion. We will see later that rebalancing can be restricted to local changes of the tree structure along the path from the root to the inserted or deleted node. In this way, rebalancing time is at most proportional to search time and hence total cost is still logarithmic.

All known classes of balanced trees can ne divided into two groups: weight-balanced and height-balanced trees. In weight-balanced trees one balances the number of nodes in the subtrees, in height-balanced trees one balances the height of the subtrees. We will discuss one representation of each group in the text and mention some more in the exercises.

3.5.1. Weight-Balanced Trees

For this section α is a fixed real, $1/4 < \alpha \le 1 - \sqrt{2}/2$. The bounds on α will become
fehlt i.E. later on. (vgl. den Beweis von Lemma 1).

Definition:

a) Let T be a binary tree with left subtree T_l and right subtree T_r. Then

$$\rho(T) = |T_l|/|T| = 1 - |T_r|/|T|$$

is called the **root balance** of T. Here $|T|$ denotes the number of leaves of tree T.

b) Tree T is of **bounded balance** *ff*, if for every subtree T' of T:

$$\alpha \le \rho(T') \le 1 - \alpha.$$

c) **BB**[*ff*] is the set of all trees of bounded balance α. ∎

In the tree of Figure 34 (leaves are not shown) the subtrees with root u (v, w, x) have root balance 1/2 (2/3, 2/5, 5/14). The tree is in BB[α] for $\alpha \leq 1/3$.

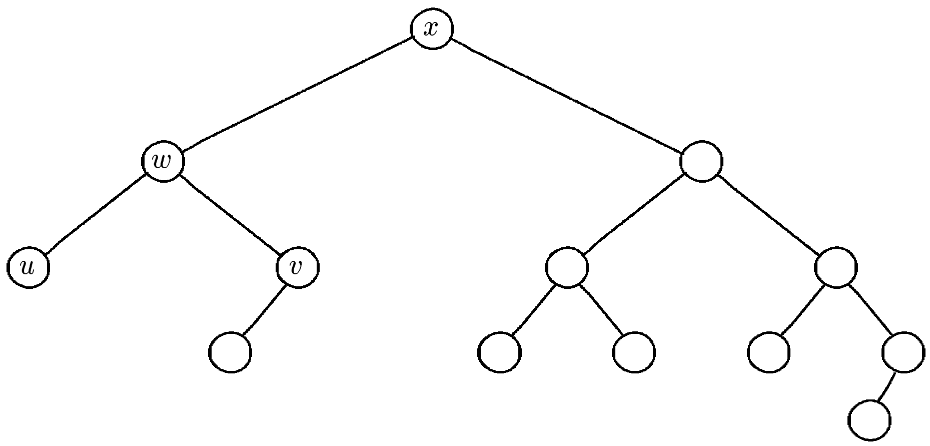

Figure 65. Ein BB[α]-Baum

Trees of bounded balance have logarithmic depth and logarithmic average path length.

Proof: Let $T \in \mathrm{BB}[\alpha]$ be a tree with n nodes. Then

a) $$P \leq \frac{(1+1/n)\cdot\log(n+1)}{H(\alpha,1-\alpha)} - 1$$

where $H(\alpha,1-\alpha) = -\alpha\log\alpha - (1-\alpha)\cdot\log(1-\alpha)$.

b) $$\mathrm{height}(T) \leq 1 + \frac{\log(n+1)-1}{\log(1/(1-\alpha))}.$$

Proof: a) For this proof it is easier to work with $\overline{P} = n\cdot P = \sum(b_i+1)$. We show $\overline{P} \leq (n+1)\cdot\log(n+1)/H(\alpha,1-\alpha) - n$ by induction on n. $\overline{P}$ is often called total (internal) path length of T. For $n=1$ we have $\overline{P}=1$. Since $0 \leq H(\alpha,1-\alpha) \leq 1$ (s.
fehlt i.E. Anhang, Formel E2), this proves the claim for $n=1$. So let us assume $n>1$. T has a left (right) subtree with l (r) nodes and path length $\overline{P}_l$ ($\overline{P}_r$). Then $n = l+r+1$ and $\overline{P} = \overline{P}_l + \overline{P}_r + n$ (cf. the proof of Lemma 1 in 3.4.1) and $\alpha \leq (l+1)/(n+1) \leq 1-\alpha$.

Applying the induction hypothesis yields

$$\begin{aligned}
\overline{P} &= n + \overline{P}_l + \overline{P}_r \\
&\le \frac{1}{H(\alpha, 1-\alpha)} \cdot \big[(l+1)\cdot\log(l+1) + (r+1)\cdot\log(r+1)\big] + 1 \\
&= \frac{n+1}{H(\alpha,1-\alpha)} \cdot \left[\log(n+1) + \frac{l+1}{n+1}\cdot\log\frac{l+1}{n+1} + \frac{r+1}{n+1}\cdot\log\frac{r+1}{n+1}\right] + 1 \\
&= \frac{(n+1)\cdot\log(n+1)}{H(\alpha,1-\alpha)} + 1 - (n+1)\cdot\frac{H(\frac{l+1}{n+1},\frac{r+1}{n+1})}{H(\alpha,1-\alpha)} \\
&\le \frac{1}{H(\alpha,1-\alpha)}\cdot(n+1)\cdot\log(n+1) - n,
\end{aligned}$$

since $H(x, 1-x)$ is monotonically increasing in x for $0 < x \le 1/2$.

b) Let $T \in \mathrm{BB}[\alpha]$ be a tree with n nodes, $k = \mathrm{height}(T)$, and let $v_0, v_1, \ldots, v_{k-1}$ be a path from the root to a node v_{k-1} of depth $k-1$. Let w_i be the number of leaves in the subtree with root v_i, $0 \le i \le k-1$. Then

$$\begin{aligned}
2 &\le w_{k-1} && \text{und} \\
w_{i+1} &\le (1-\alpha)\cdot w_i && \text{für } 0 \le i < k-1,
\end{aligned}$$

since T is of bounded balance α, and therefore

$$2 \le w_{k-1} \le (1-\alpha)^{k-1}\cdot w_0 = (1-\alpha)^{k-1}\cdot(n+1).$$

Taking logarithms completes the proof. ∎

For $\alpha = 1 - \sqrt{2}/2 \approx 0.2929$ Theorem 2 reads

$$\begin{aligned}
&P \le 1.15(1+1/n)\log(n+1) - 1 && \text{and} \\
&\mathrm{height}(T) \le 2\log(n+1) - 1.
\end{aligned}$$

A comparison with Theorem 1 shows that the average search time in trees in $\mathrm{BB}[1-\sqrt{2}/2]$ is at most 15% and the maximal search time is at most by a factor of 2 above the optimum.

Operations Access, Insert and Delete are performed as described in 3.3.1. However, insertions and deletions can move the root balance of some nodes on the path of search outside the permissible range $[\alpha, 1-\alpha]$. There are two transformations for remedying such a situation: **rotation** and **double rotation**. In the following Figures 35 and 36 nodes are drawn as circles and subtrees are drawn as triangles.

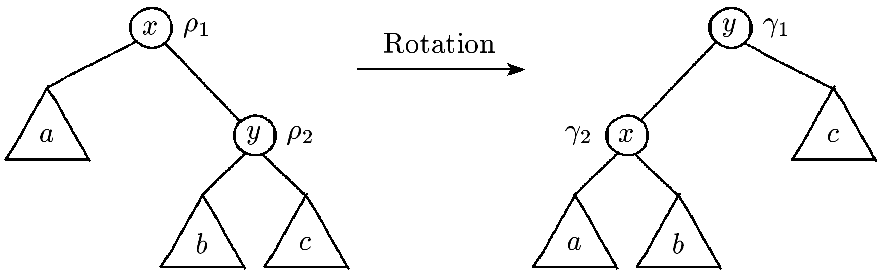

Figure 66. Rotation nach links

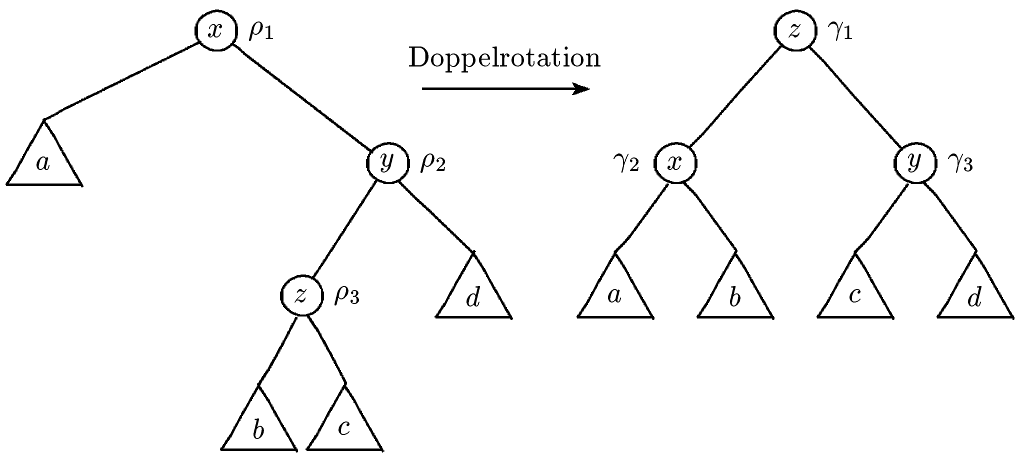

Figure 67. Doppelrotation nach links

The root-balances are given beside each node. The figures show transformations "to the left". The symmetrical variants also exist.

The root balances of the transformed trees can be computed from the old balances ρ_1, ρ_2 (and ρ_3) as given in the figures. We verify this claim for the rotation and leave the double rotation to the reader. Let a, b, c be the number of leaves in the subtrees shown. Then

$$\rho_1 = \frac{a}{a+b+c} \qquad \text{and} \qquad \rho_2 = \frac{b}{b+c}.$$

Since

$$\begin{aligned} a+b &= \rho_1 \cdot (a+b+c) + \rho_2 \cdot (b+c) \\ &= \rho_1 \cdot (a+b+c) + \rho_2 \cdot ((a+b+c)-a) \\ &= (\rho_1 + \rho_2 \cdot (1-\rho_1)) \cdot (a+b+c) \end{aligned}$$

the root-balance of node x after the rotation is given by

$$\frac{a}{a+b} = \frac{\rho_1}{\rho_1 + \rho_2 \cdot (1-\rho_1)},$$

and the root-balance of node y is given by

$$\frac{a+b}{a+b+c} = (\rho_1 + \rho_2 \cdot (1-\rho_1)).$$

Let us consider Insert first. Suppose that node a is added to the tree. Let $v_0, v_1, \ldots, v_k = a$ be the path from the root to node a after the insertion. Operation Insert(a) described in 3.3.1 creates the following subtree of Figure 37 with root balance 1/2.

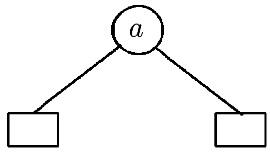

Figure 68. Durch Einfüge(a) entstehender Unterbaum

We will now walk back the path towards the root and rebalance all nodes on this path. So let us assume that we reach node v_i and that the root balances of all proper descendants of v_i are in the range $[\alpha, 1-\alpha]$. Then $0 \le i \le k-1$. If the root-balance of node v_i is still in the range $[\alpha, 1-\alpha]$ then we can move on to node v_{i-1}. If it is outside the range $[\alpha, 1-\alpha]$ we have to rebalance as described in the following lemma.

Lemma 1. *For all* $\alpha \in (1/4, 1-\sqrt{2}/2]$ *there are constants* $d \in [\alpha, 1-\alpha]$ *and* $0 \le \delta < 1$ *(if* $\alpha < 1-\sqrt{2}/2$ *then* $\delta > 0$*) such that for* T *a binary tree with subtrees* T_l *and* T_r *and*

(1) T_l *and* T_r *are in* BB$[\alpha]$

(2) $|T_l|/|T| < \alpha$ *and either*

  (2.1) $|T_l|/(|T|-1) \ge \alpha$ *(i.e., an insertion into the right subtree of* T *occurred)*

  (2.2) $(|T_l|+1)/(|T|+1) \ge \alpha$ *(i.e., a deletion from the left subtree occurred).*

(3) ρ_2 *is the root balance of* T_r

we have

(i) *if* $\rho_2 \le d$ *then rotation rebalances the tree, more precisely* $\gamma_1, \gamma_2 \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ *where* γ_1, γ_2 *are as shown in the figure describing rotation.*

(ii) *if* $\rho_2 > d$ *then a double rotation rebalances the tree, more precisely* γ_1, γ_2, $\gamma_3 \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ *where* $\gamma_1, \gamma_2, \gamma_3$ *are as shown in the figure describing double rotation.*

fehlt i.E.

Remark: Dieses Lemma besagt, daß die Wurzelbalancen nach der Rotation bzw. Doppelrotation zumindest im Bereich $[\alpha, 1-\alpha]$ liegen. Für $\alpha < 1 - \sqrt{2}/2$ liegen sie sogar in einem echt kleineren Bereich. Dies werden wir weiter unten im Beweis von Satz 4 ausnutzen.

Proof: A complete proof is very tedious and unelegant. It can be found in Blum/Mehlhorn (80). In that paper one can also find expressions for δ and d as a function of α. In order to give the reader an impression of the proof, we verify some parts of the claim for $\alpha = 3/11$, $d = 6/10$ and $\delta = 0.05$. Let us consider case (i), i.e., $\rho_2 \leq 6/10$ and a rotation is applied. We will only verify $\gamma_1 \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ and leave the remaining cases to the reader. Note first that $\gamma_1 = \rho_1 + (1-\rho_1)\rho_2$ is an increasing function of ρ_1 and ρ_2. Hence

$$\begin{aligned}
\gamma_1 &\leq \alpha + (1-\alpha)\cdot d \qquad\qquad (\text{since } \rho_1 \leq \alpha,\ \rho_2 \leq d)\\
&= \frac{78}{110}\\
&= 1 - \left(1 + \frac{1}{15}\right)\cdot\frac{3}{11}\\
&\leq 1 - 1.05\alpha.
\end{aligned}$$

We still have to prove a lower bound on γ_1. From $|T_l|/(|T|-1) \geq \alpha$ or $(|T_l|+1)/(|T|+1) \geq \alpha$ and $|T_l| \geq 1$ one concludes $\rho_1 = |T_l|/|T| \geq \alpha/(2-\alpha)$ and hence

$$\begin{aligned}
\gamma_1 &\geq \frac{\alpha}{2-\alpha} + \left(1 - \frac{\alpha}{2-\alpha}\right)\cdot\alpha\\
&= \frac{81}{209}\\
&\geq 1.05\alpha.
\end{aligned}$$

∎

Lemma 1 implies that a BB[α]-Baum can be rebalanced after an insertion by means of rotations and double rotations. The transformations are restricted to the nodes on the path from the root to the inserted element. von der Wurzel zum eingefügten Element. Thus height$(T) = O(\log|S|)$ transformations suffice; each transformation has cost of $O(1)$.

We still have to clarify two small points: how to find the path from the inserted element back to the root and how to determine whether a node is out of balance. The path back to the root can be easily found. Note that we traversed that very path when we searched for the leaf where the new element had to be inserted. We only have to store the nodes of this path in a stack; unstacking will lead us to the root. This solves the first problem. In order to solve the second problem in each node v of the tree we do not only store the left and the right son but also its size, i.e., the number of leaves in the subtree with root v. So the format of a node is

CONTENT	*LSON*	*RSON*	*SIZE*

The root balance of a node is then easily computed. Also the SIZE field is easily updated when we walk back the path of search to the root. We summarize: An operation Insert(a, S) takes time $O(\log |S|)$. This is also true for operation Delete(a, S). Delete(a, S) removes one node and one leaf from the tree as described in 3.3.1. (The node removed is necessarily the node with content a). Let $v_0, \ldots, v_k$ be the path from the root v_0 to the father v_k of the removed node. We walk back to the root along this path and rebalance the tree as described above.

Theorem 2. *Let $\alpha \in (1/4, 1 - \sqrt{2}/2]$. Then operations Access$(a, S)$, Insert$(a, S)$, Delete$(a, S)$, Min$(S)$, Deletemin$(S)$ and Ord(k, S) take time $O(\log |S|)$ in* BB$[\alpha]$-*trees. Also operation Sequ(S) takes time $O(|S|)$.*

Proof: The discussion preceding the theorem treats operations Access, Insert and Delete. The minimum of S can be found by always following left pointers starting at the root; once found the minimum can also be deleted in time $O(\log |S|)$. Operation Ord(k, S) is realized as described in 3.3.1 and Sequ(S) as in 1.5. ∎

We argued that at most height(T) transformations are required to rebalance a tree in BB$[\alpha]$ after an insertion or deletion. It is easy to find examples where one actually has to use that many transformations. Take a tree where all nodes have balance α and perform one insertion (at the proper place). Then all nodes on the path of search will leave the range $[\alpha, 1 - \alpha]$ and have to be rebalanced. Note however, that this will move the root balances of all nodes involved in the rebalancing operations into the intervals $[(1+\delta)\alpha, 1-(1+\delta)\alpha]$ where $\delta > 0$ if $\alpha \in (1/4, 1-\sqrt{2}/2)$. Therefore these nodes will not become unbalanced in the near future. This observation leads to

Theorem 3. *Let $\alpha \in (1/4, 1 - \sqrt{2}/2)$. Then there is a constant c such that the total number of rotations and double rotations required to process an arbitrary sequence of m insertions and deletions into the initially empty* BB$[\alpha]$-*tree is $\leq c \cdot m$*

Proof: Let T_0 be the BB$[\alpha]$-tree which consists of a single leaf and no node. The sequence of m insertions and deletions gives rise to a sequence of trees $T_1, \ldots, T_m$, where T_{j+1} comes from T_j by an insertion or deletion and subsequent rebalancing. We need some more notation.

A **transaction** is either an insertion or deletion. A transaction **goes through a node** v if v is on the path from the root to the node which is to be inserted or deleted. **A node** v **takes part** in a single rotation (double rotation) if it is one of the two (three) nodes explicitly shown in the figure defining the transformation. Furthermore, nodes retain their identity as shown in that figure, i.e., if a rotation to the left is applied to a subtree with root x, then node x has subtrees with weights a
fehlt i.E. and b respectively after the rotation. Die Gewichte sind dabei die Anzahlen der Blätter in den Unterbäumen. Note also that nodes are created by insertions and then have balance $1/2$ and that nodes are destroyed by deletions. Finally, a node v

causes a single rotation (double rotation) if v is node x in the figure defining the transformations, i.e., v is a node which went out of balance.

With every node v we associate accounts: The transaction accounts $TA_i(v)$ and the balancing operation accounts $BO_i(v)$, $0 \leq i < \infty$. All accounts have initial value zero. The j-te transaction, $1 \leq j \leq m$, has the following effect on the accounts of node v:

a) If the transactions does not go through v then all accounts of v remain unchanged.

b) If the transaction does go through v then let w be the weight of v in the tree T_{j-1}. Let i be such that $(1/(1-\alpha))^i \leq w < (1/(1-\alpha))^{i+1}$. Note that $w \geq 2$, $1/(1-\alpha) > 1$ and $i \geq 1$. We add one to transaction accounts $TA_{i-1}(v)$, $TA_i(v)$, $TA_{i+1}(v)$. If v causes rebalancing operation then we also add one to $BO_i(v)$.

Lemma 2. *For every node v and every i:*

$$BO_i(v) \leq \frac{(1-\alpha)^i}{\delta\alpha} \cdot TA_i(v)$$

where δ is as in Lemma 1.

Proof: We show how to count $\delta\alpha/(1-\alpha)^i$ increments of $TA_i(v)$ for every increment of $BO_i(v)$. Suppose $BO_i(v)$ is increased at the j-th transaction, i.e., the j-th transaction goes through v and moves v out of balance. Let w be the number of leaves in the subtree of T_{j-1} with root v. Then $(1/(1-\alpha))^i \leq w < (1/(1-\alpha))^{i+1}$.

Let $k < j$ be such that v takes part in a rebalancing operation at the k-th transaction or the k-th transaction created v and v does not take part in a rebalancing operations after the k-th and before the j-th transaction. In either case we have $\rho(v) = t'/w' \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ in T_k. Here t' (w') is the number of the leaves in the left subtree of v (in the tree with root v) in T_k. Since v causes a rebalancing operation at the j-th transaction we have $\rho(v) = t/w \notin [\alpha, 1-\alpha]$, say $\rho(v) < \alpha$ after the j-th transaction but before rebalancing v. We use t to denote the number of leaves in the left subtree of v in that tree.

Node v did not take part in rebalancing operations between the k-th and the j-th transaction. But its balance changed from t'/w' to t/w and hence many transactions went through v. Suppose that a insertions and b deletions went through v. Then $w = w' + a - b$ and $t \geq t' - b$.

Claim: $a + b \geq \delta\alpha w$.

Proof: Assume otherwise, i.e., $a + b < \delta\alpha w$. Then

$$\begin{aligned}(1+\delta)\alpha &\le t'/w' \\ &\le (t+b)/(w-a+b) \\ &< (t+b)/(w-\delta\alpha w+2b) \\ &\le (t+\delta\alpha w)/(w+\delta\alpha w) \\ &< (\alpha w+\delta\alpha w)/(w+\delta\alpha w) \\ &< (1+\delta)\alpha,\end{aligned}$$

contradiction. Note that $(t+b)/(w-\delta\alpha w+2b)$ is increasing in b. ∎

fehlt i.E. Eine analoge Behauptung gilt für $\rho(v) > 1-\alpha$. Ihr Beweis wird dem Leser überlassen. We have thus shown that at least $\delta\alpha w \ge \delta\alpha/(1-\alpha)^i$ transactions went through v between the k-th and the j-th transaction. During the last $\delta\alpha w$ of these transactions the weight (number of leaves in the subtree with root v) of v was at least $w - \delta\alpha w \ge 1/(1-\alpha)^{i-1}$ and at most $w + \delta\alpha w < 1/(1-\alpha)^{i+2}$. (hier wird $\delta < 1$ benutzt). Hence all these transactions were counted on $TA_i(v)$. Damit ist Lemma 2 bewiesen. ∎

Lemma 3. *For all* i: $\sum_v TA_i(v) \le 3m$.

Proof: Let $v_0, \ldots, v_k$ be the rebalancing path for the j-th transaction and let w_l be the weight of node v_l, i.e., the number of leaves in the tree with root v_l. Then $w_{l+1} \le (1-\alpha)w_l$ for $l \ge 0$. Thus there are at most three nodes on the path with $(1/(1-\alpha))^{i-1} \le w_l < (1/(1-\alpha))^{i+2}$. Hence at most three is added to $\sum_v TA_i(v)$ for each transaction. ∎ i.E.mehr

It is now easy to complete the proof. $\sum_i \sum_v BO_i(v)$ is the total number of single and double rotations required for processing the sequence of m transactions. We estimate this sum in two parts: $i < k$ and $i \ge k$ where k is some integer.

$$\begin{aligned}\sum_{i\ge k}\sum_v BO_i(v) &\le \sum_{i\ge k}\sum_v \frac{(1-\alpha)^i}{\delta\alpha}\cdot TA_i(v) && \text{(by Lemma 2)} \\ &\le \sum_{i\ge k}\frac{(1-\alpha)^i}{\delta\alpha}\cdot 3m && \text{(by Lemma 3)} \\ &\le (1-\alpha)^k\cdot 3m/\delta\alpha^2\end{aligned}$$

and

$$\sum_{i<k}\sum_v BO_i(v) < (k-1)\cdot m$$

since there is at most one node v for each transaction such that $BO_i(v)$ is increased by that transaction for any fixed i. Hence the total number of single and double rotations is bounded by $\left[(k-1) + 3(1-\alpha)^k/\delta\alpha^2\right]\cdot m$ for any integer k. ∎

It is worthwhile to compute constant c of Theorem 4 in order to give a concrete example. For $\alpha = 3/11$ we have $\delta = 0.05$ (cf. proof of Lemma 1). Choosing $k = 17$ yields $c = 19.59$. Experiments with random insertions suggest that this estimate is far too crude, the true value of c is probably close to one. Nevertheless, Theorem 4 establishing the fact that the total number of rebalancing operations is linear in the number of insertions and deletions. Furthermore, the proof of Theorem 4 also shows that $\sum_v BO_i(v) = O(m \cdot (1-\alpha)^i)$; thus rebalancing operations are very rare high up in the tree.

We can exploit this fact as follows. In Chapters 7 and 8 we will frequently augment BB[α]-trees by additional information. In this augmented trees the cost of a rotation or double rotation will not be $O(1)$; rather, the cost of a rebalancing operation caused by node v will depend on the current "thickness" of node v, i.e., if node v causes a rebalancing operation and the subtree with root v has w leaves then the cost of the rebalancing operation is $f(w)$ time units for some non-decreasing function f. In many applications we will have $f(x) = x$ or $f(x) = x \log x$.

Theorem 4. *Let $\alpha \in (1/4, 1 - \sqrt{2}/2)$ and let $f : \mathbb{R} \mapsto \mathbb{R}$ be a non-decreasing function. Suppose that the cost of performing a rotation or double rotation at node v of* BB[α]*-tree is $f(th(v))$ where $th(v)$ is the number of leaves in the subtree with root v. Then the total cost of the rebalancing operations required for a sequence of m insertions and deletions in an initially empty* BB[α]*-Baum is*

$$O\Big(m \cdot \sum_{i=1}^{c \log m} f((1-\alpha)^{-i-1}) \cdot (1-\alpha)^i\Big),$$

where $c = 1/\log(1/(1-\alpha))$.

Proof: If a node v with $(1-\alpha)^{-i} \le th(v) < (1-\alpha)^{-i-1}$ causes a rebalancing operation then the cost of this operation is at most $f((1-\alpha)^{-i-1})$ time units since f is non-decreasing. Every such rebalancing operation is recorded in account $BO_i(v)$. Hence the total cost of all rebalancing operations is

$$\begin{aligned}
&\le \sum_v \sum_i BO_i(v) \cdot f((1-\alpha)^{-i-1}) \\
&\le \sum_v \sum_i \frac{(1-\alpha)^i}{\delta\alpha} \cdot TA_i(v) \cdot f((1-\alpha)^{-i-1}) && \text{(by Lemma 2)} \\
&\le \frac{3m}{\delta\alpha} \sum_{i=1}^{c \log m} (1-\alpha)^i \cdot f((1-\alpha)^{-i-1}) && \text{(by Lemma 3)}
\end{aligned}$$

i.E.mehr and the observation that $TA_i(v) = 0$ for $i > c \log m$ by Lemma 1b. ∎

Theorem 5 has some interesting consequences. If $f(x) = x^a$ with $a < 1$ then the total rebalancing cost is $O(m)$ and if $f(x) = x(\log x)^a$ for some $a \geq 0$ then the total rebalancing cost is $O(m \cdot (\log m)^{a+1})$. Thus even $f(x)$ is fairly large the amortized rebalancing cost (i.e., rebalancing cost per insertion/deletion) is small. We will make extensive use of this fact in Chapters 7 and 8.

3.5.2. Height-Balanced Trees

Height-balanced trees are the second basic type of balanced tree. They appear in many different kinds: AVL-trees, B-trees, HB-trees, ... and (a, b)-trees which we describe here.

Definition: Let a and b be integers with $a \geq 2$ and $b \geq 2a - 1$. A tree T is an **$(a; b)$-tree if**

a) all leaves of T have the same depth
b) all nodes v of T satisfy $\rho(v) \leq b$
c) all nodes v except the root satisfy $\rho(v) \geq a$
d) the root r of T satisfies $\rho(r) \geq 2$.

Here $\rho(v)$ denotes the number of sons of node v. ∎

(a, b)-trees are known as B-trees if $b = 2a - 1$. In our examples we always use $a = 2$ and $b = 4$.

We have to deal with the following questions: how to store a set in an (a, b)-tree, how to store an (a, b)-tree in a computer, how to search in, insert into and delete from an (a, b)-tree?

Sets are stored in (a, b)-trees in a leaf-oriented way. This is not compulsory, but more convenient than node-oriented storage which we used so far. Let $S = \{x_1 < \cdots < x_n\}$ be a subset of ordered universe U and let T be an (a, b)-tree with n leaves. We store S in T as follows.

1) The elements of S are assigned to the leaves of T in increasing order from left to right.
2) To each node v of T we assign $\rho(v) - 1$ elements $k_1(v), \ldots, k_{\rho(v)-1}(v)$ of U such $k_1(v) < k_2(v) < \cdots < k_{\rho(v)-1}(v)$ und für alle Blätter w im i-ten Unterbaum von v, $1 < i < \rho(v)$, die Relation $k_{i-1}(v) < CONTENT[w] \leq k_i(v)$ gilt. Für alle Blätter w im 1-ten Unterbaum gilt $CONTENT[w] \leq k_1(v)$, für alle Blätter w im $\rho(v)$-ten Unterbaum gilt $k_{\rho(v)-1} < CONTENT[w]$.

i.E.anders

Figure 38 shows a (2,4)-tree for set $S = \{1, 3, 7, 8, 9, 10\} \subseteq \mathbb{N}$.

The simplest method of storing an (a, b)-tree in a computer is to reserve $2b - 1$ storage locations for each node of the tree, b to store the pointers to the sons and $b - 1$ to contain the keys stored in the node. In general, some of these storage

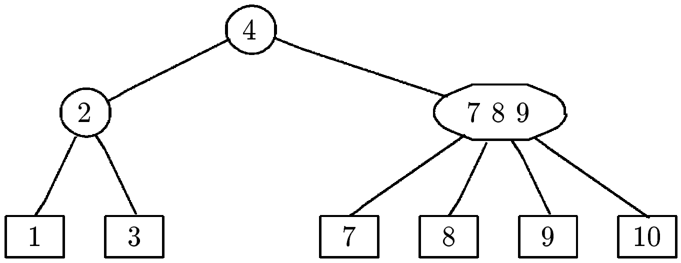

Figure 69. (a, b)-Baum für $S = \{1, 3, 7, 8, 9, 10\}$

locations are unused, namely, whenever a node has less than b sons. If a node has arity $\rho(v)$ then a fraction $(2\rho(v) - 1)/(2b - 1)$ of the storage locations will be used. Since $\rho(v) \geq a$ for all nodes (except the root) at least the fraction $(2a - 1)/(2b - 1)$ of the storage locations is used. In (2,4)-trees this might be as low as 3/7. We will see in Section 3.5.3.4 that storage location efficiency is much larger on the average. An alternative implementation is based on red-black trees and is given at the end of the section.

Searching for an element x in an (a, b)-tree with root r is quite simple. We search down the tree starting at the root until we reach a leaf. In each node v we use the sequence $k_1(v), \ldots, k_{\rho(v)-1}(v)$ in order to guide the search to the proper subtree. In Program 14 we assume that $k_0(v) < x < k_{\rho(v)}(v)$ for every element $x \in U$ and every node v of T.

$v \leftarrow$ root of T;
while v is not a leaf
do find i, $1 \leq i \leq \rho(v)$, such that $k_{i-1}(v) < x \leq k_i(x)$;
  $v \leftarrow i$-th son of v
od;
if $x = \mathit{CONTENT}[v]$
then "success" **else** "failure" **fi**.

Program 14

The cost of an search in tree T is apparently proportional to $b \cdot \text{height}(T)$; there are $\text{height}(T)$ iterations of the while-loop and at each iteration $O(b)$ steps are required to find the proper subtree. Since b is a constant we have $O(b \cdot \text{height}(T)) = O(\text{height}(T))$ and again the height of tree T plays a crucial role.

$O(\ldots)$?? i.D

Lemma 4. *Let T be an (a, b)-tree with n leaves and height h. Then*

a) $2a^{h-1} \leq n \leq b^h$.

b) $\log n / \log b \leq h \leq 1 + \log(n/2)/\log a$.

Proof: Since each node has at most b sons there are at most b^h leaves. Since the root has at least two sons and every other node has at least a sons there are at least $2a^{h-1}$ leaves. This proves a). Part b) follows from Part a) by taking logarithms. ∎

We infer from Lemma 4 and the discussion preceding it that operation Access(x, S) takes time $O(\log |S|)$. We will now turn to operation Insert(x, S). A search for element x in tree T ends in some leaf w. Let v be the father of w. If $x = CONTENT[w]$, then we are done. If $x \neq CONTENT[w]$, then we proceed as follows:

1) We expand v by giving it an additional son to the right of w (we also say: we **split** w), store x and $CONTENT[w]$ in w and the new leaf in appropriate order and store $\min(x, CONTENT[w])$ in v at the appropriate position, i.e., between the pointers to w and the new leaf.

Example: Insertion of 6 into the tree of Figure 69 yields the tree of Figure 39. ∎

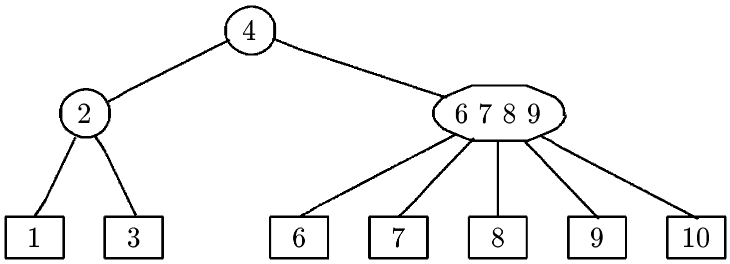

Figure 70. Nach Einfügen von 6

2) Adding a new leaf increases the arity of v by 1. If $\rho(v) \leq b$ after adding the new leaf then we are done. Otherwise we have to split v. Since splitting can propagate we formulate it as a loop in Program 15.

---

while $\rho(v) = b + 1$
do if v's father exists
  then let y be the father of v
  else let y be a new node and make v the only son of y
 fi;
 let v' be a new node;
 expand y, i.e., make v' an additional son of y
 immediately to the right of v;
 split v, i.e., take the rightmost $\lceil (b+1)/2 \rceil$ sons and keys
 $k_{\lfloor (b+1)/2 \rfloor + 1}(v), \ldots, k_b(v)$ away from v and incorporate them into v' and move key
 $k_{\lfloor (b+1)/2 \rfloor}(v)$ from v to y (between the pointers to v and v');
 $v \leftarrow y$
od.

Program 15

Example (continued): Splitting v yields Figure 40. ▮

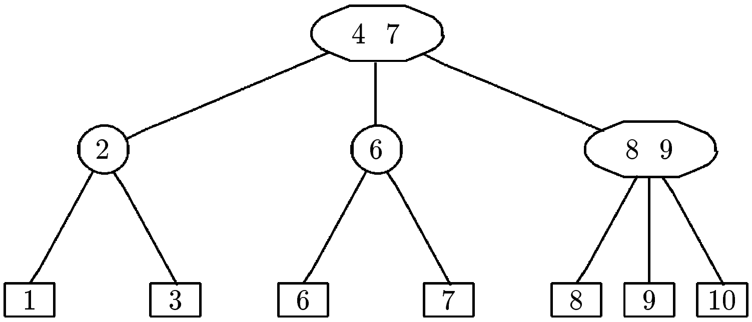

Figure 71. Nach Spalten von v

Lemma 5. *Let $a \geq 2$ and $b \geq 2a-1$. Inserting an element x into an (a,b)-tree for set S takes time $O(\log |S|)$.*

Proof: The search for x takes time $O(\log |S|)$. Also, we store the path of search in a pushdown store during the search. After adding a new leaf to hold x we return to the root using the pushdown store and apply some number of splitting operations. If a node v is split it has $b+1$ sons. It is split into nodes v and v' with $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ sons respectively. Since $b \geq 2a-1$ we have $\lfloor (b+1)/2 \rfloor \geq a$ and since $b \geq 3$ we have $\lceil (b+1)/2 \rceil \leq b$ and thus v and v' satisfy the arity constraint after the split. A split takes time $O(b) = O(1)$ and splits are restricted to a final segment of the path of search. This proves Lemma 5. ▮

Deletions are processed very similarly. Again we search for x, the element to be deleted. The search ends in leaf w with father v.

1) If $x \neq CONTENT[w]$ then we are done. Otherwise, we shrink v by deleting leaf w and one of the keys in v adjacent to the pointer to w (to be specific, if w is the i-th son of v then we delete $k_i(v)$ if $i < \rho(v)$ and $k_{i-1}(v)$ if $i = \rho(v)$.)

Example (continued): Deleting 6 yields Figure41. ▮

i.D.anders 2) Shrinking v decreases $\rho(v)$ by 1. Wenn $v = r$ und $\rho(v) = 1$, wird die Wurzel gestrichen. Wenn $v \neq r$ und $\rho(v)$ noch $\geq a$ ist, sind wir fertig. Otherwise v needs to be rebalanced by either fusing or sharing. Sei y linke oder rechte Bruder von v (ein beliebiger von beiden, falls beide existieren). Programm 16 zeigt, wie man die Rebalancierung in diesem Fall implementiert.

Example (continued): The tree shown in Figure 72 can be either rebalanced by sharing or fusing depending on the choice of y. If y is the left brother of v then fusing yields Figure 42. If y is the right brother of v then sharing yields Figure 43. ▮

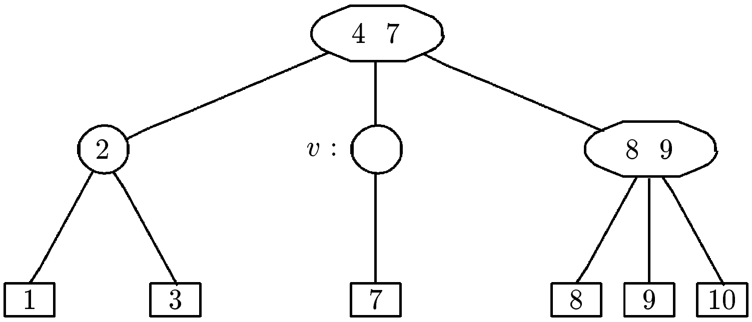

Figure 72. Nach Streichen von 6

---

while $\rho(v) = a - 1$ **and** $\rho(y) = a$
do let z be the father of v;
  fuse v and y, i.e., make all sons of y sons of v and
  move all keys from y to v and delete node y;
  also move one key (the key between the pointers to y and v)
  from z to v (note that this will shrink z, i.e., decrease the arity of z by one);
  if z is root of T
  then if $\rho(z) = 1$ **then** delete z und mache v zur Wurzel **fi**;
    goto completed
  fi;
  $v \leftarrow z$;
  let y be the a brother of v;
od;
co we have either $\rho(v) \geq a$ and rebalancing is completed
  or $\rho(v) = a - 1$ and $\rho(y) > a$ and rebalancing is completed by sharing;
oc
if $\rho(v) = a - 1$
then co we assume that y is the right brother of v **oc**
  take the leftmost son away from y and make it an additional (rightmost) son of v;
  also move one key (the key between the pointers to v and y
  from z down to v and replace it by the leftmost key of y;
fi;
completed:

Program 16

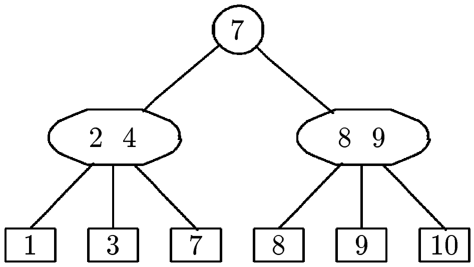

Figure 73. Rebalancierung durch Verschmelzen

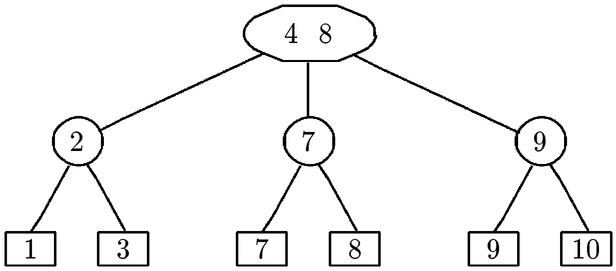

Figure 74. Rebalancierung durch Stehlen

Lemma 6. *Let $a \geq 2$ and $b \geq 2a-1$. Deleting an element from an (a,b)-tree for set S takes time $O(\log|S|)$.*

Proof: The search for x takes time $O(\log|S|)$. An (a,b)-tree is rebalanced after the removal of a leaf by a sequence of fusings followed by at most one sharing. Each fusing or sharing takes time $O(b) = O(1)$ and fusings and sharings are restricted to the path of search. Finally note that a fusing combines a node having $a-1$ sons with a node having a sons and yields a node with $2a-1 \leq b$ sons. ∎

We summarize Lemmas 4, 5 and 6 in

Theorem 5. *Let $a \geq 2$, $b \geq 2a-1$. If set S is represented by an (a,b)-tree then operations Access(x,S), Insert(x,S), Delete(x,S), Min(S), Deletemin(S) take time $O(\log|S|)$.*

Proof: For operations Access, Insert and Delete this is obvious from Lemma 4,5 and 6. For Min and Deletemin one argues as in Theorem 3. ∎

(a,b)-trees provide us with many different balanced tree schemes. For any choice of $a \geq 2$ and $b \geq 2a-1$ we get a different class. We will argue in 3.5.3.1 that $b \geq 2a$ is better than $b = 2a-1$ (= the smallest permissible value for b) on the basis that amortized rebalancing cost is much smaller for $b \geq 2a$. So let us assume for the moment that $b = 2a$. What is a good value for a?

We will see that the choice of a is depends heavily on the intended use of the tree. Is the tree kept in main storage, or is the tree stored on secondary memory? In the latter case we assume that it costs $C_1 + C_2 m$ time units to transport a segment of m contiguous storage locations from secondary to main storage. Here C_1 and C_2 are device dependent constants. We saw in Lemma 4 that the height of an $(a, 2a)$-tree with n leaves is about $\log n / \log a$. Let us take a closer look at the search algorithm in (a, b)-tree. The loop body which is executed $(\log n / \log a)$ times consists of two statements: in the first statement the proper subtree is determined for a cost of $c_1 + c_2 a$, in the second statement out attention is shifted to the son of the current node for a cost of $C_1 + C_2 a$. Here $C_2 = 0$ if the tree is in main memory and $C_1 + C_2 a$ is the cost of moving a node from secondary to main memory otherwise. Thus total search time is

$$(c_1 + c_2 a + C_1 + C_2 a) \log n / \log a$$

which is minimal for a such that

$$a \cdot (\ln a - 1) = (c_1 + C_1)/(c_2 + C_2).$$

If the tree is kept in main memory then typical values of the constants are $c_1 \approx c_2 \approx C_1$ and $C_2 = 0$ and we get $a = 2$ or $a = 3$. If the tree is kept in secondary storage, say on a disk, then typical values of the constants are $c_1 \approx c_2 \approx C_2$ and $C_1 \approx 1000 c_1$. Note that C_1 is the latency time and C_2 is the time to move one storage location. In this case we obtain $a \approx 100$. From this coarse discussion one sees that in practice one will either use trees with small arity or trees with fairly large arity.

We close this section with a detailed description of an implementation of $(2, 4)$-trees by red-black trees. A tree is **colored** (with colors red and black) if its edges are colored red and black. If v is a node we use $bd(v)$ to denote the number of black edges on the path from the root to v; $bd(v)$ is the **black depth** of node v. A **red-black tree** is a binary, colored tree satisfying the following three structural constraints:

1) all leaves have the same black depth,
2) all leaves are attached by black edges,
3) no path from the root to a leaf contains two consecutive red edges.

In the following diagrams we draw red edges as wiggled lines and black edges as straight lines.

There is a close relationship between $(2, 4)$-trees and red-black trees. Let T be fehlt a $(2, 4)$-tree. If we replace nodes with three (four) sons by Gebilde aus Abbildung 44, then we obtain a red-black tree.

In the example of the beginning of the section (cf. Figure 69) we obtain the tree shown in Figure 45.

Conversely, if T is a red-black tree and we collapse by red edges then a $(2, 4)$-tree is obtained. Red-black-trees allow for a very efficient and elegant implementation. Each node and leaf is stored as a type node where

Figure 75. Grundbausteine eines Rot-Schwarz-Baumes

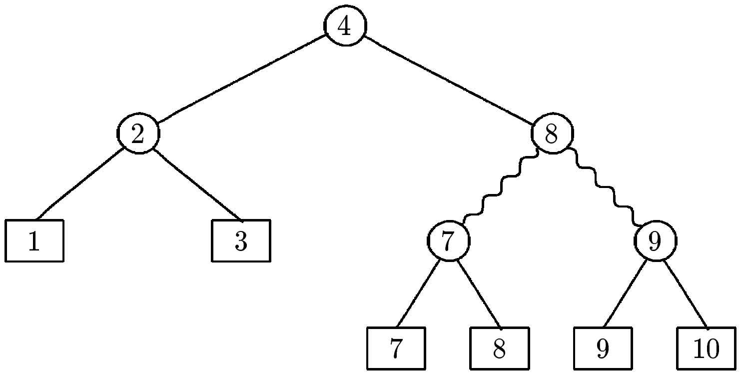

Figure 76. Rot-Schwarz-Baum für das Beispiel vom Anfang von III.5.2

```
type node = record content    : U;
                   color      : (red, black);
                   son[0..1]  : ↑node
            end;
```

The field *content* stores an element of the underlying universe, the field *color* contains the color of the ingoing edge (the color of the edge into the root is black by default), and fields $son[0]$ and $son[1]$ contain the pointers to the left and right son respectively. All son-pointers of leaves are nil.

In the following programs variables *root*, p, q and r are of type $\uparrow node$, but we simply talk about nodes instead of pointers to nodes. A node is called red (black) if its ingoing edge is red (black).

The program for operation Access is particularly simple. For later use we store the path of search in a pushdown store, more precisely we store pairs (z, d) where z is a pointer to a node and $d \in \{0, 1\}$ is the direction out of node z taken in the search.

Program 17 searches for $x \in U$.

```
p ← root;
while p↑.son[0] ≠ nil co a test, whether p is a leaf oc
do if x ≤ p↑.content
     then Push(p, 0);
            p ← p↑.son[0]
```

```
(6)          else Push(p, 1);
(7)               p ← p↑.son[1]
(8)          fi
(9)     od;
(10)    if p↑.content = x
(11)    then "successful"
(12)    else "unsuccessful"
(13)    fi.
```

Program 17

We will next turn to operation Insert. Suppose that we want to insert x and also suppose that we executed Program 17 above. It terminates in line (12) with p pointing to a leaf and with the path of search stacked in a pushdown store. The leaf p is not stacked. We initialize the insertion algorithm by replacing leaf p by a red node r with sons p and a new leaf containing x (cf. Fig. 46).

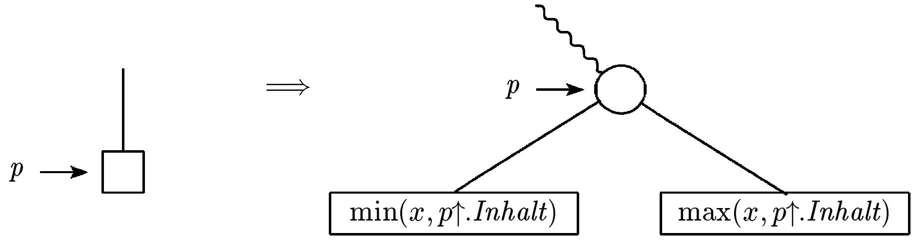

Figure 77. Beginn der Operation Einfüge(x)

This is done in Programs 18 und 19. We also redirect the pointer from p's father to the new node r. If p's father is black this completes the insertion.

```
new(q);
q↑.color ← black; q↑.son[0] ← q↑.son[1] ← nil;
q↑.content ← x;
new(r);
r↑.content ← min(x, p↑.content);
r↑.color ← red;
if x < p↑.content
   then r↑.son[0] ← q; r↑.son[1] ← p
   else r↑.son[1] ← q; r↑.son[0] ← p
fi;
(q, dir2) ← pop stack;
q↑.son[dir2] ← r;
if q↑.color = black
then – – "the insertion is completed"
fi;
```

Program 18

If node q (as tested in line (13)) is red then we created two consecutive red edges and therefore violated the third structural constraint. Program 19 restores the red-black properties. i.E.anders Nach Zeile (16) von Programm 19 sind wir in folgender Situation. Node p is black (because its son q is red), node q is red and has exactly one red son, namely r (otherwise, two consecutive red edges would have existed before the insertion). We will maintain this property as an invariant of the following loop. In this loop we distinguish two cases. If both sons of node p are red then we perform a color flip on the edges incident to p and propagate the "imbalance" closer to the root. A color flip corresponds to a spit in $(2,4)$-trees. If p has only one red son then we rebalance the tree either by a rotation $((dir1 = dir2))$ or a double rotation $((dir1 \neq dir2))$. Both subcases correspond to expanding node p in the related $(2,4)$-tree and hence terminate the insertion.

---

(16) $(p, dir1) \leftarrow$ pop stack;
(17) **while** true
(18) **do if** $p\uparrow.son[1 - dir1]\uparrow.color =$ red
(19)   **then co** p has two red sons and we perform a color flip **oc**
(20) $p\uparrow.son[0].color \leftarrow p\uparrow.son[1].color \leftarrow$ black;
(21) $p\uparrow.color \leftarrow$ red;
(22)   **if** $p = root$
(23)   **then** $p\uparrow.color \leftarrow$ black;
(24)   terminate the insertion algorithm
(25)   **fi**;
(26) $r \leftarrow p$;
(27) $(q, dir2) \leftarrow$ pop stack;
(28)   **if** $q\uparrow.color =$ black
(29)   **then** terminiere den Einfüge-Algorithmus
(30)   **fi**;
(31) $(p, dir1) \leftarrow$ pop stack
(32)   **else** **if** $dir1 = dir2$
(33)   **then co** we rebalance the tree by a rotation **oc**

```
                p↑.son[dir1] ← q↑.son[1 − dir1];
                q↑.son[1 − dir1] ← p;
                p↑.color ← red; q↑.color ← black;
                if p = root
                then root ← q
                else  (r, dir2) ← pop stack;
                      r↑.son[dir2] ← q
                fi;
                terminate the insertion algorithm
          else co we rebalance the tree by a double rotation oc
```

```
                p↑.son[dir1] ← r↑.son[dir2];
                q↑.son[dir2] ← r↑.son[dir1];
                r↑.son[dir1] ← q;
                r↑.son[dir2] ← p;
                p↑.color ← red; r↑.color ← black;
                if p = root
                then root ← r
                else  (p, dir2) ← pop stack;
                      p↑.son[dir2] ← r
                fi;
                terminate the insertion algorithm
          fi
    fi
od.
```

Program 19

At this point it is appropriate to make several remarks. We mentioned already that color flips in red-black trees correspond to splits in $(2,4)$-tree. What do rotations and double rotations correspond to? They have no equivalent in (2,4)-trees but correspond to the fact that the subtrees shown in Figure 47 are not "legal" realizations of nodes with four sons.

Figure 78. So dürfen 4-Knoten nicht aussehen

If we replaced the third structural constraint by "red components (= sets of nodes connected by red edges) consist of at most three nodes" then the subtree of Figure 78 would be legal and the rotations and double rotations would not be required. However, the code which realizes a split would become more cumbersome.

Deletions from red-black trees can be handled by a similar but slightly longer program. The program is longer because more cases have to be distinguished. We leave the details to the reader.

However, there is one important remark that should be made. The algorithms for red-black trees as described above simulate (2,4)-trees in he following sense. Let T be a (2,4)-tree and let T' be the red-black tree which corresponds to T as described above. Suppose now that we perform an insertion into (deletion from) T and the same operation on T'. Let T_1 and T_1' be the resulting (2,4)-tree and red-black tree respectively. Then T_1' corresponds to T_1 in the sense described above. Also, the number of color-flips required to process the operation on T' is the same as the number of splits required to process the operation on T. We will use this observation frequently in the sequel without explicitly mentioning it. More precisely, we will derive bounds on the amortized rebalancing cost in (2-4)-trees in the next section. These bounds holds also true for red-black trees (if the programs above are used to rebalance them).

Red-black trees can also be used to implement (a,b)-trees in general. We only have to replace the third structural constraint by: "red components have at least $a-1$ and at most $b-1$ nodes" (cf. Exercise 27).

Finally, we should mention that there are alternative methods for rebalancing (a,b)-trees, $b \geq 2a$, and red-black trees after insertions and deletions. A very useful alternative is top-down rebalancing. Suppose that we want to process an insertion.
i.E.mehr As usual, we follow a path down the tree. However, we also maintain the invariant now that the current node is not a b-node (a node with b sons). If the current node is a b-node then we immediately split it. Since the father is not a b-node (by the invariant) the splitting does not propagate towards the root. In particular, when the search reaches the leaf level the new leaf can be added without any problem. The

reader should observe that $b \geq 2a$ is required now because we split b-nodes instead of $(b+1)$-nodes now. The reader should also note that the results presented in Section 3.5.3.2 below are *not* true for the top-down rebalancing strategy. However, similar results can be shown provided that $b \geq 2a+2$ (cf. Exercises 29 and 32).

Top-down rebalancing of (a,b)-trees is particularly useful in a parallel environment. Suppose that we have several processors working on the same tree. Parallel searches cause no problems but parallel insertions and deletions do. The reason is that while some process modifies a node, e.g., in a split, no other process can use that node. In other words, locking protocols have to be used in order to achieve mutual exclusion. These locking protocols are fairly simple to design if searches and rebalancing operations proceed in the same direction (deadlock in a one-way street is easy to avoid), i.e., if top-down rebalancing is used. The protocols are harder to design and usually have to lock more nodes if searches and rebalancing operations proceed in opposite directions (deadlock in a two-way street is harder to avoid), i.e., if bottom-up rebalancing is used.

We return to the discussion of parallel operations on (a,b)-trees at the end of Section 5.3.2. In Section 5.3.2 we prove a result on the distribution of rebalancing operations on the levels of the tree. In particular, we will show that rebalancing operations close to the root where locking is particularly harmful are very rare.

3.5.3. Advanced Topics on (a,b)-Trees

(a,b)-trees are a very versatile data structure as we will see now. We first describe two additional operations on (a,b)-trees, namely Split and Concatenate, and then apply them to priority queues. In the second section we study the amortized rebalancing cost of (a,b)-trees with regard to sequences of insertions and deletions. We use the word **amortized** to denote the fact that the time bounds derived are valid for sequences of operations. The amortized cost per operation is much smaller than the worst case cost of a single operation. This analysis leads to finger trees in the third section. Finally, we introduce fringe analysis, a method for partially analyzing random (a,b)-trees.

3.5.3.1. Mergable Priority Queues

We introduce two more operations on sets and show how to implement them efficiently with the help of (a,b)-trees. We then describe the implementation of mergable priority queues based on (a,b)-trees.

Name der Operation	*Effekt der Operation*
Concatenate(S_1, S_2, S_3)	$S_3 \leftarrow S_1 \cup S_2$;
Split(S_1, y, S_4, S_5)	$S_4 \leftarrow \{x \in S_1;\ x \leq y\}$ and $S_5 \leftarrow \{x \in S_1;\ x > y\}$.

Operation Concatenate is only defined if $\max S_1 < \min S_2$. Both operations are destructive, i.e., sets S_1 and S_2 (set S_1 respectively) are destroyed by an application of Concatenate (Split).

Theorem 6. *Let $a \geq 2$ and $b \geq 2a - 1$. If sets S_1 and S_2 are represented by (a,b)-trees then operation Concatenate(S_1, S_2, S_3) takes time $O(\log \max(|S_1|, |S_2|))$ and operation Split(S_1, y, S_4, S_5) takes time $O(\log |S_1|)$.*

Proof: We treat Concatenate first. Let S_1 and S_2 be represented by (a,b)-trees T_1 and T_2 of height h_1 and h_2 respectively. We assume that the height of a tree and the maximal element of a tree are both stored in the root. It is easy to see that this assumption does not affect the time complexity of any of the operations considered so far.

Assume w.l.o.g that $h_1 \geq h_2$. Let r_2 be the root of T_2 and let v be the node of depth $h_1 - h_2$ on the right spine of T_1, i.e., v is reached from the root of T_1 by following the pointer to the rightmost son $(h_1 - h_2)$ times. Fuse v and r_2 and insert the maximal element stored in T_1 as the additional key in the combined node. The combined node has arity at most $2b$. If its arity is not larger than b then we are done. Otherwise we split it in the middle and proceed as in the case of an insertion. Splitting may propagate all the way to the root. In any case, the time complexity of the algorithm is $O(|h_1 - h_2| + 1) = O(\max(h_1, h_2)) = O(\log \max(|S_1|, |S_2|))$. Also note that the resulting tree has height $\max(h_1, h_2)$ or $\max(h_1, h_2) + 1$. This completes the description and analysis of Concatenate.

The algorithm for Split(S_1, y, S_4, S_5) is slightly more complicated. Let T_1 be an (a,b)-tree for S_1. The first thing to do is to search for y and to split all nodes on the path of search at the pointer which leads to that successor which is also on the path of search. In the example shown in Figure 48 along the path to leaf 12 is splitted.

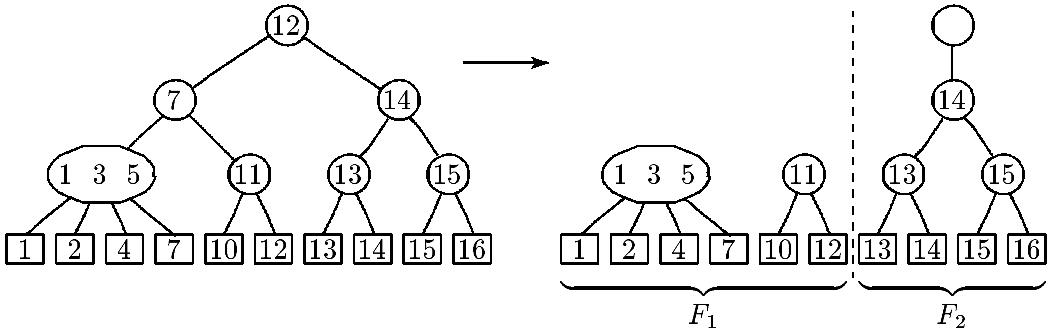

Figure 79. Zerlege$(S_1, 12, S_4, S_5)$

The splitting process yields two forests F_1 and F_2, i.e., two sets of trees. F_1 is the set of trees to the left of the path and F_2 is the set of trees to the right of the path. The union of the leaves of F_1 gives S_4 and the union of the leaves of

F_2 gives S_5. We show how to build the tree for S_4 in time $O(\log|S_1|)$. The root of each tree in F_1 is a piece of a node on the path of search. Hence F_1 contains at most $h_1 = \text{height}(T_1)$ trees. Let $D_1, \ldots, D_m$, $m \le h_1$, be the trees in F_1 from left to right. Then each D_i is an (a,b)-tree except for the fact that the arity of
fehlt i.D. the root might be only one. Let us call such a tree an almost (a,b)-tree. Also $\max D_i < \min D_{i+1}$ and $\text{height}(D_i) > \text{height}(D_{i+1})$ for $1 \le i < m$. Hence we can form a tree for set S_4 by executing the sequence of Concatenates of Program 20.

Concatenate(D_{m-1}, D_m, D'_{m-1});
Concatenate($D_{m-2}, D'_{m-1}, D'_{m-2}$);

⋮

Concatenate(D_1, D'_2, D'_1)

Program 20

It is easy to see that applying Concatenate to almost (a,b)-trees yields an almost (a,b)-tree. Hence D'_1 is an (a,b)-tree for set S_4. If the root of D'_1 has arity 1 then we only have to delete it and obtain an (a,b)-tree for S_4.

We still have to analyze the running time of this algorithm. The splitting phase is clearly $O(\log|S_1|)$. For the build-up phase let h'_i be the height of D'_i. Then the complexity of the build-up phase is $O(|h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m)$.

Claim: $h_{i+1} \le h'_{i+1} \le h_{i+1} + 1 \le h_i$.

Proof: Since D_{i+1} is used to form D'_{i+1} we clearly have $h'_{i+1} \ge h_{i+1}$. We show $h'_{i+1} \le h_{i+1} + 1 \le h_i$ by induction on (decreasing) i. For $i = m-2$ we have

$$h'_{m-1} \le \max(h_{m-1}, h_m) + 1 \le h_{m-1} + 1 \le h_{m-2},$$

since $h_m < h_{m-1} < h_{m-2}$ and for $i < m-2$ we have

$$\begin{aligned} h'_{i+1} &\le \max(h_{i+1}, h'_{i+2}) + 1 && \text{(, property of Concatenate)} \\ &\le h_{i+1} + 1 && \text{, } h'_{i+2} \le h_{i+1} \text{ by I.H.)} \\ &\le h_i. && \text{, since } h_{i+1} < h_i \quad \blacksquare \end{aligned}$$

Thus

$$\begin{aligned} &|h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m \\ &\le h_{m-1} - h_m + \sum_{i=1}^{m-2} (h_i - h'_{i+1}) + m \end{aligned}$$

$$\leq h'_{m-1} - h_m + \sum_{i=1}^{m-2} (h'_i - h'_{i+1}) + m$$

$$\leq m + h'_1 \;\leq\; m + h_1 + 1 \;=\; O(\log|S_1|). \qquad ∎$$

In our example F_1 consists of two trees. Concatenating them yields Figure 49.

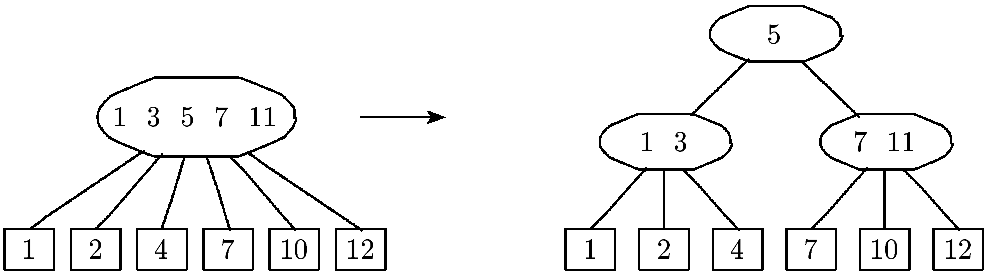

Figure 80. Nach Konkatenieren von F_1

Concatenate is a very restricted form of set union , da $\max S_1 < \min S_2$ fehlt i.E. gefordert wird. General set union of ordered sets is treated below in 3.5.3.3. However, Concatenate is good enough to handle the union question on *unordered* lists. A very frequent task is the manipulation of priority queues; we will see an example in Section 4.6 on shortest path algorithms. The name **mergable priority queue** refers to the problem of manipulating a set of sets under the operations Insert(x, S), Min(x, S), Deletemin(S) and Union(S_1, S_2, S_3). We will add two more operations later. Note that operation Access is not included in this list. Also note, that operation Access was the main reason for storing the elements of a set in sorted order in the leaves of an (a, b)-tree. We give up this principle now.

Definition: A set S, $|S| = n$, is represented as follows in an **unordered $(a; b)$-tree** T with n leaves:

1) the leaves of T contain the elements of S in some order
2) each node v of T contains the minimal element in the subtree with root v and a pointer to the leaf containing that element. ∎

Operation Min(S) is trivial in this data structure; it takes time $O(1)$. For Deletemin(S) we follow the pointer from the root to the minimal element and delete it. Then we walk back to the root, rebalance the tree by fusing and sharing and update the min values and pointers on the way. Note that the min value and pointer of a node can be found by looking at all its sons. Thus Deletemin(S) takes time fehlt i.E. $O(a \log|S|/\log a)$. (Dabei haben wir $b = O(a)$ angenommen.) We include factor a because we will see that other operations are proportional to $O(\log|S|/\log a)$ and because we will exploit that fact in 4.7.

Insert(x, S) is a special case of Union where one of the sets is a singleton. Finally Union(S_1, S_2, S_3) reduces to Concatenate and takes time $O(a \log(\max|S_1|, |S_2|))/\log a)$.

Delete in its pure form cannot be supported because there is no efficient method of finding an element. However, a modified form of Delete is still supported. Given a pointer to a leaf that leaf can be deleted in time $O(a \log |S|/\log a)$ as described for Deletemin above. We call this operation Delete*.

Finally we consider operation Demote* which takes a pointer to a leaf and an element x of the universe. Demote* is only applicable if x is smaller than the element currently stored in the leaf, and changes (demotes) the content of that leaf to x. Demote* takes time $O(\log |S|/\log a)$ because one only has to walk back to anders the root, alle Min-Werte größer als x auf diesem Pfad durch x ersetzen und die Min-Zeiger richtigstellen muß. We summarize in:

Theorem 7. *Let $a \geq 2$, $b \geq 2a - 1$ und $b = O(a)$. If sets are represented by* i.E.anders *unordered (a, b)-trees then dann haben die folgenden Operationen die angeführten Laufzeiten:*

operation	*running time*
Min(S)	$O(1)$
Deletemin(S)	$O(a \log \|S\|/\log a)$
Insert(x, S)	$O(a \log \|S\|/\log a)$
Union(S_1, S_2, S_3)	$O(a \log(\|S_1\| + \|S_2\|)/\log a)$
Delete$^*(\ , S)$	$O(a \log \|S\|/\log a)$
Demote$^*(\ , \ , S)$	$O(\log \|S\|/\log a)$

∎

3.5.3.2. Amortized Rebalancing Cost and Sorting Presorted Files

In this section we study the total cost of sequences of insertions and deletions in (a, b)-trees under the assumption that we start with an initially empty tree. We will show that the total cost is linear in the length of the sequence. The proof follows a general paradigm for analyzing the cost of sequences of operations, the bank account paradigm, which we already considered in the proof of Theorem 4 in 3.5.1. We associate a bank account with the tree; the balance of that account measures the balance of the tree. We will then show that adding or pruning a leaf corresponds to the withdrawal of a fixed amount from that account and that the rebalancing operations (fusing, sharing and splitting) correspond to deposits. Finally using the obvious bounds for the balance of the account, we obtain the result.

Theorem 8. *Let $a \geq 2$ and $b \geq 2a$. Consider an arbitrary sequence of i insertions and d deletions ($n = i + d$) into an initially empty (a,b)-tree. Let SP be the total number of node splittings, F the total number of node fusings and SH be the total number of node sharings. Then*

a) $SH \leq d \leq n$.

b) $(2c-1) \cdot SP + c \cdot F \leq n + c + \dfrac{c}{a+c-1} \cdot (i - d - 2)$,

wobei $c = \min\big(\min(2a-1, \lceil (b+1)/2 \rceil) - a, b - \max(2a-1, \lfloor (b+1)/2 \rfloor)\big)$.
Note that $c \geq 1$ *for* $a \geq 2$ *and* $b \geq 2a$ *and hence* $SP + F \leq n/c + 1 + (n-2)/a$.

Proof: a) Node sharing is executed at most once for each deletion. Hence $SH \leq d$.

b) We follow the paradigm outline above. For a node v (unequal the root) of an (a,b)-tree let $b(v)$ of v be

$$b(v) = \min(\rho(v) - a, b - \rho(v), c),$$

where c is defined as above. Note that $c \geq 1$ for $a \geq 2$ and $b \geq 2a$ and that $\rho(v') = \lfloor (b+1)/2 \rfloor$ and $\rho(v'') = \lceil (b+1)/2 \rceil$ implies $b(v') + b(v'') \geq 2c - 1$ and that $\rho(v) = 2a - 1$ implies $b(v) = c$. For rot r (r will always denote the root in the sequel) the balance is defined as

$$b^*(r) = \min(\rho(r) - 2, b - \rho(r), c).$$

Definition:

(i) (T, v) is a **partially rebalanced $(a;b)$-tree** where v is a node of T if

a) $a - 1 \leq \rho(v) \leq b + 1$ if $v \neq r$;
$1 \leq \rho(v) \leq b + 1$ if $v = r$;

b) $a \leq \rho(w) \leq b$ for all $w \neq v, r$;

c) $2 \leq \rho(r) \leq b$ if $v \neq r$.

(ii) Let (T, v) be a partially rebalanced (a,b)-tree. Then the **balance** of T is defined as the sum of the balance of its nodes

$$b(T) = b^*(r) + \sum_{\substack{v \text{ is node of } T \\ v \neq r}} b(v).$$ ∎

Fact 1. *Let T be an (a,b)-tree. Let T' be obtained from T by adding a leaf or pruning a leaf. Then $b(T') \geq b(T) - 1$.*

Proof: obvious. ∎

Fact 2. *Let (T, v) be a partially rebalanced (a, b)-tree with $\rho(v) = b+1$. Splitting v and expanding v's father x generates a tree T' with $b(T') \geq b(T) + (2c-1)$.*

i.D.anders *Proof*: Wir unterscheiden zwei Fälle.

Case 1: v is not the root of T.
Wegen $\rho(v) = b+1$ ist $b(v) = -1$. Let x be the father of v. v is split into nodes, v' and v'' say, of arity $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ respectively, and the arity of x is increased by one. Hence the balance of x decreases by at most one and we have

$$b(T') \geq b(T) + b(v') + b(v'') - b(v) - 1.$$

Furthermore, $b(v) = -1$ and $b(v') + b(v'') \geq 2c - 1$ by the remark above. This shows

$$\begin{aligned} b(T') &\geq b(T) + (2c-1) - (-1) - 1 \\ &\geq b(T) + (2c-1). \end{aligned}$$

Case 2: v is the root of T.
Wegen $\rho(v) = b+1$ ist $b^*(v) = -1$. Then the father x of v is newly created and hence has arity 2 after the splitting of v. Hence

$$\begin{aligned} b(T') &\geq b(T) + b(v') + b(v'') - b^*(v) + b^*(x) \\ &\geq b(T) + (2c-1) - (-1) + 0. \end{aligned}$$

∎

Fact 3. *Let (T, v) be a partially rebalanced (a, b)-tree with $\rho(v) = a-1$ and $v \neq r$ the root of T. Let y be a brother of v an let x be the father of v.*

- **a)** *If $\rho(y) = a$ then let T' be the tree obtained by fusing v and y and shrinking of x. Furthermore, if x is the root and has degree 1 after the shrinking, then x is deleted. Then $b(T') \geq b(T) + c$.*
- **b)** *If $\rho(y) > a$ then let T' be the tree obtained by sharing, i.e., taking 1 son away from y and making it son of v. Then $b(T') \geq b(T)$.*

i.E.mehr *Proof*: a) y and v are fused to a node, say w, of arity $\rho(w) = 2a-1$.

Case 1: x is not the root of T.
Then the arity of x is decreased by one and hence the balance of x is decreased by at most one. Weiterhin ist $b(w) = c$. Hence

$$\begin{aligned} b(T') &\geq b(T) + b(w) - b(v) - b(y) - 1 \\ &\geq b(T) + c - (-1) - 0 - 1 \\ &= b(T) + c. \end{aligned}$$

Case 2: x is the root of T.
If $\rho(x) \geq 3$ before the fusing then the analysis of case 1 applies. Otherwise we have $\rho(x) = 2$ and hence $b^*(x) = 0$ before the fusing, and x is deleted after the fusing
fehlt and w is the new root. , wobei $b^*(w) = c$. Hence

$$\begin{aligned} b(T') &= b(T) + b^*(w) - b(v) - b(y) - b^*(x) \\ &= b(T) + c - (-1) - 0 - 0 \\ &= b(T) + c + 1. \end{aligned}$$

In either case we have shown $b(T') \geq b(T) + c$.

b) Taking away one son from y decreases the balance of y by at most one. Giving v (of arity $a - 1$) an additional son increases $b(v)$ by one. Hence $b(T') \geq b(T)$. ∎

Fact 4. *Let T be an (a, b)-Baum with m leaves. Then*

$$0 \leq b(T) \leq c + (m - 2) \cdot \frac{c}{a + c - 1}.$$

Proof: For $0 \leq j < c$, let m_j be the number of interior nodes unequal the root of degree $a + j$ and let m_c be the number of interior nodes unequal the root of degree at least $a + c$. Then

$$b(T) \leq c + \sum_{j=0}^{c} j \cdot m_j,$$

since the balance of the root is at most c. Furthermore,

$$2 + \sum_{j=0}^{c} (a + j) \cdot m_j \leq m + \sum_{j=0}^{c} m_j$$

since the expression on the right side is equal to the number of edges (= number of leaves + non-root nodes) in T and the expression on the left hand side is a lower bound on that number. Hence

$$\sum_{j=0}^{c} (a + j - 1) \cdot m_j \ \leq \ m - 2.$$

Since

$$j/(a + j - 1) \ \leq \ c/(a + c - 1)$$

for $0 \le j \le c$ we conclude

$$\begin{aligned} b(T) &\le c + \sum_{j=0}^{c} j \cdot m_j \\ &= c + \sum_{j=0}^{c} \frac{j}{a+j-1} \cdot (a+j-1) \cdot m_j \\ &\le c + \frac{c}{a+c-1} \cdot (m-2). \end{aligned}$$ ▮

In order to finish the banking account paradigm we need to relate deposits, withdrawals and initial and final balance. Since we start with an empty tree T_0, the initial balance $b(T_0)$ is 0. Next we perform i insertions and d deletions and obtain T_n $(n = i+d)$. Since T_n has $i-d$ leaves we conclude $b(T_n) \le c+(i-d-2)c/(a+c-1)$. Also the total withdrawals are at most n by Fact 1 and the total deposits are at least $(2c-1) \cdot SP + c \cdot F$ by Facts 2 and 3. Hence

$$b(T_0) + (2c-1) \cdot SP + c \cdot F - n \le b(T_n)$$

and thus

$$(2c-1) \cdot SP + c \cdot F \le n + c + (i-d-2) \cdot c/(a+c-1).$$ ▮

Theorem 9 establishes an $O(1)$ bound on the amortized rebalancing cost in (a,b)-trees for $b \ge 2a$. In that respect it is a companion theorem to Theorem 4 on weight-balanced trees. Let us consider som concrete values of a and b. For $a = 2$, $b = 4$ we have $c = 1$ and hence $SP + F \le 3n/2$. For $(4,13)$-trees we have $c = 3$ and hence $SP + F \le 7n/12 + 1/2$. We should also mention that Theorem 9 does *not* hold if $b = 2a-1$ as can be seen from the example shown in Figure 50. rebalancing always runs all the way to the root. However, Theorem 9 is true for $b = 2a-1$ if one considers insertions only (Exercise 30).

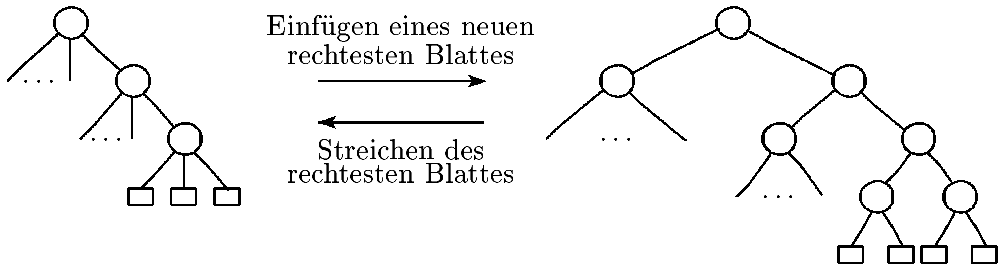

Figure 81. Schlechtes Verhalten von (2,3)-Bäumen

Theorem 9 has an interesting application to sorting, more precisely to sorting presorted files. Let $x_1, \ldots, x_n$ be a sequence which has to be sorted in increasing order. Suppose, that we know that this sequence is not random, but rather possesses some degree of sortedness. A good measure for sortedness is the number of inversions. Let

$$f_i = |\{x_j;\ j > i \text{ and } x_j < x_i\}|$$

and $F = \sum_{i=1}^{n} f_i$. Then $0 \le F \le n(n-1)/2$, F is called the number of inversions of the sequence $x_1, \ldots, x_n$. We take F as a measure of sortedness; $F = 0$ for a sorted sequence, $F \approx n^2$ for a completely unsorted sequence and $F \ll n^2$ for a
fehlt i.D. nearly sorted sequence. These remarks are not meant to be definitions. We can sort sequence $x_1, \ldots, x_n$ by insertion sort, i.e., we start with the sorted sequence
Lemma 5? x_n represented by an (a,b)-tree and then insert $x_{n-1}, x_{n-2}, \ldots, x_1$ in turn. By Theorem 5 this will take total time $O(n \log n)$. Note however, when we have sorted $x_{i+1}, \ldots, x_n$ already and next insert x_i, then x_i will be inserted at the (f_i+1)-th position from the left in that sequence. If $F \ll n^2$, then $f_i \ll n$ for most i and hence most x_i's will be inserted near the beginning of the sequence.

What does this mean for the insertion algorithm? It will move down the left spine (the path to the leftmost leaf) for a considerable amount of time and only branch off it near the leaf-level. It is then cheaper to search for the point of departure from the left spine by moving the spine upwards from the leftmost leaf. Of course, this requires the existence of son-to-father pointers on the left spine.

Let us consider the insertion of x_i in more detail. We move up the left spine through nodes $v_1, v_2, v_3, \ldots$ (v_1 is the father of the leftmost leaf and v_{j+1} is the father of v_j) until we hit node v_j with $x_i \le k_1(v_j)$, i.e., x_i has to be inserted into the leftmost subtree of node v_j but not into the leftmost subtree of node v_{j-1}, i.e., either $j = 1$ or $x_i > k_1(v_{j-1})$. In either case we turn round at v_j and proceed as in an ordinary (a,b)-tree search. Note that the search for x_i will cost $O(1+\text{height}(v_j))$
fehlt i.E. time units. Since für $j \ge 3$ all leaves stored in the subtree rooted at v_{j-2} must have content less than x_i we conclude that $f_i \ge$ number of leaves in subtree rooted at $v_{j-2} \ge 2a^{h(v_{j-2})-1}$ by Lemma 4a) and hence

$$\text{height}(v_j) = O(\log f_i).$$

Die Fälle $f_i = 0$ und $f_i = 1$ vernachlässigen wir der Einfachheit halber, denn sie ändern nichts am Ergebnis. The actual insertion of x_i costs $O(s_i)$ time units, where s_i is the number of splits required after adding a new leaf for x_i. This gives the following time bound for the insertion sort

$$\begin{aligned}\sum_{i=1}^{n} \big[O(\log f_i) + O(s_i)\big] &= O(n + \sum_{i=1}^{n} \log f_i + \sum_{i=1}^{n} s_i) \\ &= O(n + n\log(F/n))\end{aligned}$$

since $\sum \log f_i = \log \prod f_i$ and $(\prod f_i)^{1/n} \le \sum f_i/n$ (the geometric mean is never larger than the arithmetic mean) and since $\sum s_i = O(n)$ by Theorem 9. Thus we have

Theorem 9. *A sequence of n elements and F inversions can be sorted in time $O(n + n\log(F/n))$.*

Proof: By the preceding discussion. ∎

The sorting algorithm (**A-Sort**) of Theorem 10 may be called adaptive. The more sorted sequence $x_1, \ldots, x_n$ is, the faster is the algorithm. For $F = n\log n$, the running time is $O(n \log\log n)$. This is in marked contrast to all other sorting algorithms, e.g., Heapsort, Quicksort, Mergesort. In particular, it will be faster than
i.E.gestrichen all these other algorithms, provided that F is not too large. Eine sorgfältige, nicht asymptotische Analyse von A-Sort geht über den Rahmen dieses Buches hinaus; der Leser findet sie in Mehlhorn/Tsakalidis (82). Dort wird gezeigt, daß A-Sort besser als Quicksort ist, wenn $F \le 0.02n^{1.57}$.

Let us return to Theorem 9 and its companion Theorem 4 on weight-balanced trees. In the proof of Theorem 4 we also established that most rotations and double rotations occur near the leaves. We will now proceed to show a similar theorem for height-balanced trees. We need some more notation.

We say that a splitting (fusing,sharing) operation **occurs at height** h, if node v which is to be split (which is to be fused with its brother y or shares a son with its brother y) has height h; the height of a leaf being 0. A splitting (fusing) operation at height h expands (shrinks) a node at height $h+1$. An insertion (deletion) of a leaf expands (shrinks) a node at height 1.

Let (T, v) be a partially rebalanced (a, b)-tree. We define the balance of tree T at height h as:

$$b_h(T) = \begin{cases} \displaystyle\sum_{v \text{ node of } T \text{ of height } h} b(v) & \text{if } h \ne \text{height(Wurzel)}; \\ b^*(r) & \text{if } h = \text{height(Wurzel)}, \end{cases}$$

where b and b^* are defined as in the proof of Theorem 9.

Theorem 10. *Let $a \ge 2$ and $b \ge 2a$. Consider an arbitrary sequence of i insertions and d deletions ($n = i + d$) into an initially empty (a, b)-tree. Let SP_h (F_h, SH_h) be the total number of node spittings (fusings, sharings) at height h. Then*

$$SP_h + F_h + SH_h \le 2(c+2)n/(c+1)^h$$

where c is defined as in Theorem 9.

Proof: The first part of the proof parallels the proof of Theorem 9.

Fact 1. *Let T be an (a, b)-tree. Let T' be obtained from T by adding or pruning a leaf. Then $b_1(T') \ge b_1(T) - 1$ and $b_h(T') = b_h(T)$ for $h > 1$.* ∎

Fact 2. *Let (T, v) be a partially rebalanced (a, b)-tree with $\rho(v) = b+1$ and height of v equal h. Splitting v und Expandieren des Vaters von v generates a tree T' with $b_h(T') \geq b_h(T) + 2c$, $b_{h+1}(T') \geq b_{h+1}(T) - 1$ and $b_l(T') = b_l(T)$ for $l \neq h, h+1$.* ∎

fehlt i.E.

Fact 3. *Let (T, v) be a partially rebalanced (a, b)-tree with $\rho(v) = a - 1$, height of v equal h and $v \neq r$ the root of T. Let y be a brother of v and let x be the father of v.*

a) *if $\rho(y) = a$ then let T' be the tree obtained by fusing v and y and shrinking x. Furthermore, if x is the root of T and has degree 1 after the shrinking, then x is deleted. Then $b_h(T') \geq b_h(T) + c + 1$, $b_{h+1}(T') \geq b_{h+1}(T) - 1$ and $b_l(T') = b_l(T)$ for $l \neq h, h+1$.*

b) *if $\rho(y) > a$ then let T' be the tree obtained by sharing. Then $b_l(T') \geq b_l(T)$ for all l.* ∎

The proofs of Facts 1 to 3 are very similar to the proof of the corresponding facts in Theorem 9 and therefore left to the reader.

Fact 4. *Let T_n be the tree obtained after i insertions and d deletions from T_0, the initial tree. Let $SP_0 + F_0 = i + d$. Then for all $h \geq 1$*

$$b_h(T_n) \geq b_h(T_0) - (SP_{h-1} + F_{h-1}) + (2c \cdot SP_h + (c+1) \cdot F_h).$$

Proof: Facts 1–3 imply that splits (fusings) at height h increase the balance at height h by $2c$ $(c+1)$ and decrease the balance at height $h+1$ by at most 1. ∎

Since $b_h(T_0) = 0$ for all h (recall we start with an empty tree) and $2c \geq c+1$ (recall that $c \geq 1$) we conclude from Fact 4

$$SP_h + F_h \leq \frac{b_h(T_n)}{c+1} + \frac{SP_{h-1} + F_{h-1}}{c+1}.$$

and hence

$$\begin{aligned} SP_h + F_h &\leq \sum_{l=0}^{h-1} \frac{b_{h-l}(T_n)}{(c+1)^{l+1}} + \frac{SP_0 + F_0}{(c+1)^h} \\ &= \frac{n}{(c+1)^h} + \sum_{l=1}^{h} \frac{b_l(T_n) \cdot (c+1)^l}{(c+1)^{h+1}}. \end{aligned}$$

Fact 5. *For all $h \geq 1$:*

$$\sum_{l=1}^{h} b_l(T_n) \cdot (c+1)^l \leq (c+1) \cdot (i-d) \leq (c+1) \cdot n.$$

Proof: Let $m_j(h)$ be the number of nodes of height h and degree $a+j$, $0 \leq j < c$, and let $m_c(h)$ be the number of nodes of height h and degree at least $a+c$. Then

$$(*) \qquad \sum_{j=0}^{c} (a+j) \cdot m_j(h) \leq \sum_{j=0}^{c} m_j(h-1),$$

for $h \geq 2$ since the number of edges ending at height $h-1$ is equal to the number of edges emanating at height h. Setting $\sum_{j=0}^{c} m_j(0) = i-d$ (the number of leaves of T_n) the inequality holds true for all $h \geq 1$. Using $b_l(T_n) \leq \sum_{j=0}^{c} j \cdot m_j(l)$ and relation $(*)$ we obtain

$$\begin{aligned}
&\sum_{l=1}^{h} b_l(T_n) \cdot (c+1)^l \\
&\leq \sum_{l=1}^{h} \left[(c+1)^l \cdot \sum_{j=0}^{c} j \cdot m_j(l) \right] \\
&\leq \sum_{l=1}^{h} \left[(c+1)^l \cdot \left[\sum_{j=0}^{c} m_j(l-1) - a \cdot \sum_{j=0}^{c} m_j(l) \right] \right] \\
&= (c+1) \cdot \sum_{j=0}^{c} m_j(0) \\
&\quad + \sum_{l=2}^{h} (c+1)^l \cdot \left[\sum_{j=0}^{c} m_j(l-1) - \frac{a}{c+1} \cdot \sum_{j=0}^{c} m_j(l-1) \right] \\
&\quad - (c+1)^h \cdot a \cdot \sum_{j=0}^{c} m_j(h) \\
&\leq (c+1) \cdot (i-d),
\end{aligned}$$

since $a/(c+1) \geq 1$ and $\sum_j m_j(0) = i-d$. ∎

Combining the bound of Fact 5 with the bound for $SP_h + F_h$ we obtain

$$SP_h + F_h \leq 2n/(c+1)^h.$$

With respect to sharing, note that fusing at height $h-1$ either completes rebalancing
fehlt i.E. or is followed by a sharing or fusing at height h. Ferner geht einem Stehlen auf Höhe $h > 1$ stets ein Verschmelzen auf Höhe $h-1$ und einem Stehlen auf Höhe 1 stets eine Streiche-Operation voraus. Hence

$$SH_h \leq F_{h-1} - F_h \leq SP_{h-1} + F_{h-1} \leq 2n/(c+1)^{h-1}$$

for $h \geq 2$ and

$$SH_1 \leq SH_1 + F_1 \leq d.$$

Combining everything we obtain

$$SP_h + F_h + SH_h \leq 2(c+2)n/(c+1)^h$$

for all $h \geq 1$ and Theorem 11 is proven. ∎

$SP_h + F_h + SH_h$ is the number of insertions and deletions which require rebalancing up to the height h or higher. Theorem 11 shows that this number is exponentially
of Theorem 11?? decreasing with h (recall $c \geq 1$). This is very similar to Lemma 2 of 3.5.1. Again note that the proof of Theorem 11 relies heavily on the fact that $b \geq 2a$. In fact, Theorem 11 is not true for $b = 2a-1$, as the example following the proof of Theorem 9 shows. There $SP_h + F_h + SH_h = n$ for all h.

We complete this section with a brief sketch of an application of Theorem 11: (a,b)-trees in a parallel environment. Some applications, in particular real-time data bank applications, require a very high transaction rate. Very high transaction rates can only be supported by concurrent tree manipulation, i.e., by many processors working on the same tree. Concurrent searches cause no problem; however, concurrent insertions/deletions do. Note that the rebalancing operations (splitting, fusing, sharing) require locking of nodes, i.e., whilst node v is rebalanced by one process, the other processes cannot use node v and have to wait if they want to use it. Also note, that locking node v will (on the average) block many other processes if v is close to the root and that it will hardly block any other process if v is close to the leaves. Theorem 11 guarantees that most rebalancing operations occur close to the leaves and that therefore blocking of processes is no insurmountable problem. We refer the reader to Bayer/Schkolnik (77) for detailed discussion.

3.5.3.3. Finger Trees

In this section we generalize from (a,b)-trees to finger trees. In the previous section we have seen that it is sometimes better to start the search in an (a,b)-tree at a leaf. This led to A-sort. Finger trees grow out out of that observation and generalize it.

Recall that we used (a,b)-trees to represent ordered lists. A **finger** into a list
fehlt i.E. is a pointer to an element of the list. (Im folgenden identifizieren wir oft den Finger auf ein Element mit dem Element selbst.) Dies wird jeweils aus dem Kontext klar.

Fingers may be used to indicate areas of high activity in the list and we aim at efficient searches in the vicinity of fingers. If the list is represented as the leaves of an (a, b)-tree then a finger is a pointer to a leaf. (a, b)-trees, as defined above, do not support efficient search in the vicinity of fingers. This is due to the fact that neighboring leaves may be connected only by a very long path. Therefore we introduce level-linked (a, b)-trees.

In **level-linked $(a; b)$-trees** all tree edges are made traversable in both directions (i.e., there are also pointers from sons to fathers); in addition each node has pointers to the two neighboring nodes on the same level. Figure 51 shows a level-linked $(2, 4)$-tree for list 2, 4, 7, 10, 11, 15, 17, 21, 22, 24.

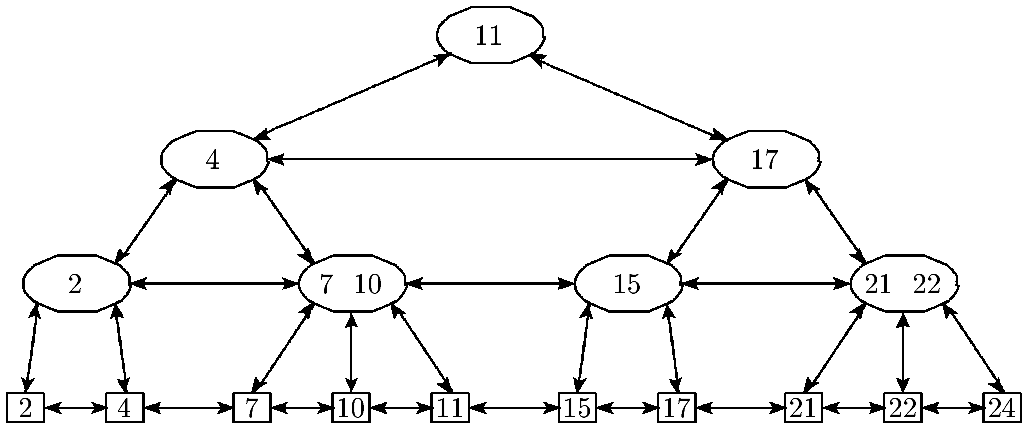

Figure 82. Ein niveau-verbundener (2,4)-Baum

A **finger tree** is a level-linked (a, b)-tree with pointers to some of its leaves, the fingers. Level-linked (a, b)-trees allow very fast searching in the vicinity of fingers.

Lemma 7. *Let p be a finger in a level-linked (a, b)-tree T. A search for a key k which is d keys away from p takes time $\Theta(1 + \log d)$.*

Proof: We first check whether k is to the left or right of p, say k is to the right of p. Then we move towards the root, say we reached node v. We check whether k is a descendant of v or v's right neighbor on the same level. If not, then we proceed to v's father. Otherwise we turn round and search for k in the ordinary way.

Suppose that we turn round at node w of height h. Let us be that son of w which is on the path to the finger p. Then all descendants of u's right neighbor lie between the finger p and key k. Hence the distance d is at least a^{h-1} and at most $2b^h$. The time bound follows.

Lemma 8. *A new leaf can be inserted in a given position of a level-linked (a, b)-tree in time $\Theta(1 + s)$, where s is the number of splittings caused by the insertion.*

Proof: This is obvious from the description of the insertion algorithm in 3.5.2. Note that it is part of the assumption of Lemma 8 that we start with a pointer to the leaf

??? which is to be split by the insertion. auf das Blatt beginnen, neben dem eingefügt werden soll. So no search is required, only rebalancing. ∎

Lemma 9. *A given leaf can be deleted from a level-linked (a, b)-tree in time $\Theta(1 + f)$, where f is the number of node fusings caused by the deletion.*

Proof: Obvious from the description of the deletion algorithm. Again note that it is part of the assumption of Lemma 9 that we start with a pointer to the leaf which has to be deleted. ∎

Lemma 10. *Creation or removal of a finger at a given leaf in a level-linked (a, b)-tree takes time $\Theta(1)$, wenn das Ziel des Fingers bekannt ist.*

Proof: Obvious. ∎

We can now refer to Theorem 9 and show that although the search time in level-linked (a, b)-trees can be greatly reduced by maintaining fingers, it still dominates the total execution time, provided that $b \geq 2a$.

Theorem 11. *Let $a \geq 2$ and $b \geq 2a$. Then any sequence of searches, finger creations, finger removals, insertions and deletions starting with an empty list takes time*

$$O(\textit{total cost of searches})$$

fehlt i.E. *if a level-linked (a, b)-tree is used to represent the list. Dabei wird vorausgesetzt, daß bei allen Operationen (außer bei Suche) das Blatt, an dem die Operation auszuführen ist, durch eine vorhergehende Suche bestimmt wird.*

Proof: Let n be the length of the sequence. Since every operation has to be preceded
i.D.weniger immediately by a search, the total cost for the searches is $\Omega(n)$ by Lemma 7. Die Gesamtkosten der Suchen sind $\Omega(n)$. On the other hand, the total cost for the finger creations and removals is $O(n)$ by Lemma 10 and the total cost of insertions and deletions is $O(n)$ by Lemmas 8 and 9 and Theorem 9. ∎

Theorem 10 on sorting presorted files is a special case of Theorem 12. Theorem 12 can be generalized in two directions. We can start with several empty lists and add Concatenate to the set of operations (Exercise 33) or we can start with a non-empty list. For the latter generalization we also need a more general version of Theorem 9. We need a bound on the total rebalancing cost even if we start with a non-empty tree.

Let T be any (a, b)-tree. Suppose now that we execute a sequence of insertions and deletions on T. It is intuitively obvious that only nodes are affected (this is made precise in Fact 1 below) which lie on a path from one of the inserted or deleted leaves to the root. We will derive a bound on the number of such nodes in Fact 5. Of course, only these nodes can change their balance and hence their number gives a bound on the difference of the balance of the initial tree and the final tree. An application of the bank account paradigm will then give us a bound on the number of splittings, fusings and sharings (Fact 4).

In order to facilitate the following discussion we introduce the following convention: if a leaf is deleted from an (a, b)-tree, it (conceptually) is not deleted but turns into a phantom. The rebalancing algorithms do not see the phantoms, i.e., the arity of nodes is solely determined by the non-phantom leaves. In the following Figure 52 phantom leaves are shown as dashed boxes.

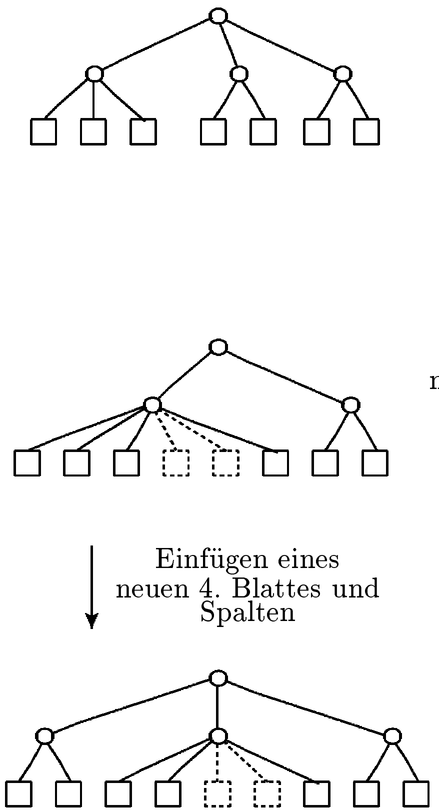

Figure 83. Phantome in einem $(2, 4)$-Baum

Theorem 12. *Let $a \geq 2$ and $b \geq 2a$ and let T be an (a, b)-tree with n leaves. Suppose that a sequence of s insertions and t deletions is performed and tree T' is obtained. Then tree T' has $n + s$ leaves, t of which are phantoms. Number the leaves from 1 to $n + s$ from left to right. Let $p_1, p_2, \ldots, p_{s+t}$ be the positions of the $s + t$ new leaves and phantoms in T', $p_1 \leq p_2 \leq \cdots \leq p_{s+t}$. Let SH (SP, F) be the total number of sharings (splittings, fusings) required to process this sequence.*

Then

$$SH + SP + F \le 9(s+t) + 4\left(\lfloor \log_a(n+s)\rfloor + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1)\rfloor\right).$$

Proof: We first introduce a marking process which (conceptually) marks nodes during the rebalancing process.

Adding a leaf: When a new leaf is added we mark the new leaf and its father.

Splitting a node: Suppose that node v (v will be marked at this point and has a marked son) is split into nodes v' and v''. Then v' is marked if it has a marked son, v'' is marked if it has a marked son and we mark the father of v' and v''.

Pruning a leaf: We mark the phantom and its father.

Fusing: Suppose that node v (v has a marked son at this point) is fused with its brother y. Then we mark v and its father.

Sharing: Suppose that node v (v has a marked son at this point) is given a son from its brother y. Then we mark v and we unmark y (if it was marked).

Fact 1. *If interior node v is marked then it has a marked son.*

Proof: Obvious. ∎

Since the only leaves which are marked are the new leaves and the phantoms we conclude from Fact 1 that all marked nodes lie on paths from the new leaves and the phantoms to the root. In Fact 5 we derive an upper bound on the number of such nodes.

Next we define some new concepts: critical and noncritical splitting operation and marked balance. A **splitting** of node v is **critical** if v's father exists and has arity b. All other splittings are **noncritical**. Let SPC be the number of critical splittings and let $SPNC$ be the number of noncritical splittings and let SH (F) be the number of sharings (fusings).

Fact 2. $SPNC + SH \le s + t$.

Proof: A sharing or noncritical splitting terminates rebalancing. ∎

The **marked balance** of a (a,b)-tree T is the sum of the balances of its marked nodes, i.e.,

$$mb(T) = \sum_{\substack{w \text{ marked node of } T\\ w \ne \text{root of } T}} b(w) + \textbf{if } \text{root is marked } \textbf{then } b^*(r) \textbf{ else } 0 \textbf{ fi},$$

where the balance of a node $w \neq r$ is defined as follows.

$$b(w) = \begin{cases} -1 & \text{if } \rho(w) \in \{a-1, b+1\}; \\ 0 & \text{if } \rho(w) = b; \\ 2 & \text{if } \rho(w) = 2a-1; \\ 1 & \text{if } a \leq \rho(w) < b \text{ and } \rho(w) \neq 2a-1 \end{cases}$$

and for the root

$$b^*(r) = \begin{cases} -1 & \text{if } \rho(r) \in \{+1, b+1\}; \\ 0 & \text{if } \rho(r) = b; \\ 2 & \text{if } \rho(r) = 2a{-}1; \\ 1 & \text{if } 2 \leq \rho(r) < b \text{ and } \rho(r) \neq 2a-1. \end{cases}$$

Fact 3. *Let T' be obtained from T by*

a) *adding a leaf, then* $mb(T') \geq mb(T) - 2$;

b) *pruning a leaf, then* $mb(T') \geq mb(T) - 2$;

c) *noncritical splitting, then* $mb(T') \geq mb(T)$;

d) *critical splitting, then* $mb(T') \geq mb(T) + 1$;

e) *fusing, then* $mb(T') \geq mb(T) + 2$;

f) *sharing, then* $mb(T') \geq mb(T)$.

Proof: a) Suppose we add a leaf and expand v. Then v is marked after adding the leaf and may be marked or not before. Hence

$$\begin{aligned} mb(T') = mb(T) &+ b(v \text{ after adding leaf}) \\ &- [b(v \text{ before adding leaf})], \end{aligned}$$

where the expression in square brackets only occurs if v is marked before adding the leaf. In either case it is easy to see that $mb(T') \geq mb(T) - 2$.

b) Similar to part a).

c) Suppose we split node v (which is marked at this point) into nodes v' and v'' and expand v's father x. Since the splitting is noncritical we have $\rho(x) < b$ before the splitting. Also, at least one of v' and v'' will be marked after the splitting, x will be marked after the splitting, x may be marked before or not. Hence

$$\begin{aligned} mb(T') \geq mb(T) &+ \min(b(v'), b(v'')) + b(x \text{ after splitting}) \\ &- b(v) - [b(x \text{ before splitting})], \end{aligned}$$

where the expression in square brackets only occurs if x was marked before splitting. Since $\min(b(v'), b(v'')) \geq 1$ (obviously $\lceil (b+1)/2 \rceil < b$) and since the balance of x decreases by at most two we conclude

$$mb(T') \geq mb(T) + 1 + 0 - (-1) - [2].$$

d) Argue as in Case c) but observe that $\rho(x) = b$ before splitting. Therefore

$$\begin{aligned} mb(T') &\geq mb(T) + 1 + (-1) - (-1) - [0] \\ &\geq mb(T) + 1. \end{aligned}$$

e) Suppose that we fuse v and its brother y and shrink their common father x. If x is the root and has arity 1 after the fusing then x is deleted. Also v and x are marked after the fusing and at least v is marked before. y and x may be marked before. Hence

$$\begin{aligned} mb(T') &\geq mb(T) + b(v \text{ after fusing}) \\ &\quad + b(x \text{ after fusing}) - b(v \text{ before fusing}) \\ &\quad - [b(x \text{ before fusing})] - [b(y \text{ before fusing})] \\ &\geq mb(T) + 2 + (-1) - (-1) - [0] - [0] \\ &\geq mb(T) + 2. \end{aligned}$$

fehlt i.D. since the balance of x can decrease by at most one.

f) Suppose that v takes away a son from y. Then v is marked before and after the sharing, y is not marked after the sharing and may be marked or not marked before the sharing. Also, the balance of y before the sharing is at most 2. Hence

$$\begin{aligned} mb(T') &\geq mb(T) + b(v \text{ after sharing}) \\ &\quad - b(v \text{ before sharing}) - [b(y \text{ before sharing})] \\ &\geq mb(T) + 1 - (-1) - [2] \\ &\geq mb(T). \end{aligned}$$

∎

Suppose now that we start with an initial empty tree T (no node is marked) and perform s insertions and t deletions and obtain T_n $(n = s + t)$.

Fact 4.

a) $mb(T_n) \geq SPC + 2F - 2 \cdot (s + t)$.

b) $mb(T_n) \leq 2m$, *where m is the number of marked nodes of T_n.*

c) *All marked nodes lie on a path from a new leaf or phantom to the root.*

d) $SPC + F \leq 2 \cdot (s + t) + 2 \cdot$*(number of nodes on the path from the new leaves and phantoms to the root).*

Proof: a) follows immediately from Fact 3, b) follows immediately from the definition of marked balance, c) follows from Fact 1 and d) is a consequence of a), b) and c). ∎

We still have to derive a bound on the number of marked nodes in tree T_n. In Fact 5 we use the name (a, ∞)-tree for any tree where each interior node unequal the root has at least a sons and where the root has at least 2 sons.

Fact 5. *Let T be an (a, ∞)-tree with N leaves. Let $1 \le p_1 \le p_2 \le \cdots \le p_r \le N$. Let m be the total number of nodes on paths from the root to the leaves with positions p_i, $1 \le i \le r$. Then*

$$m \le 3r + 2\left(\lfloor \log_a N \rfloor + \sum_{i=2}^{r} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor\right).$$

Proof: For every node v label the outgoing edges $0, \dots, \rho(v) - 1$ from left to right as shown in Figure 53.

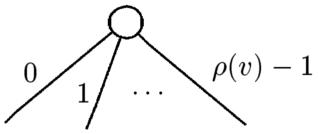

Figure 84. Markierung der in v ausgehenden Kanten

Then a path from the root to a node corresponds to a word over alphabet $\{0, 1, 2, \dots\}$ in a natural way.

Let A_i be the number of edges labelled 0 on the path from the root to leaf p_i, $1 \le i \le r$. Since an (a, ∞)-of height h has at least $2a^{h-1}$ leaves, we conclude $0 \le A_i \le 1 + \lfloor \log_a N/2 \rfloor$. Furthermore, let l_i be the number of interior nodes on the path from leaf p_i to the root which are not on the path from leaf p_{i-1} to the root. Then

$$m \le 1 + \lfloor \log_a N/2 \rfloor + \sum_{i=2}^{r} l_i.$$

Consider any $i \ge 2$. Let v be the lowest common node on the paths from leaves p_{i-1} and p_i, $p_{i-1} \ne p_i$, to the root. Then edge k_1 is taken out of v on the path to p_{i-1} and edge $k_2 > k_1$ is taken on the path to p_i. Note that the path from v to leaf p_{i-1} as well as to leaf p_i consists of $l_i + 1$ edges, da alle Blätter gleiche Tiefe haben.
fehlt i.E.

Claim: $A_i \ge A_{i-1} + l_i - 2\lfloor \log_a (p_i - p_{i-1} + 1) \rfloor - 3$.

fehlt i.E. *Proof*: Für $p_{i-1} = p_i$ ist die Behauptung klar. Sei $p_{i-1} \ne p_i$. The paths from p_{i-1} and p_i to the root differ only below node v. Let s be minimal such that

a) the path from v to p_{i-1} has the form $k_1 \alpha \beta$ with $|\beta| = s$ and α contains no 0.
b) the path from v to p_i has the form $k_2 0^{|\alpha|} \gamma$ for some γ with $|\gamma| = s$.

Note that either β starts with a 0 or γ starts with a non-zero and that $|\alpha| + |\beta| = l_i$. Figure 54 illustrates the situation. Hence

$$\begin{aligned} A_i = A_{i-1} &+ \textbf{if } k_1 = 0 \textbf{ then } -1 \textbf{ else } 0 \textbf{ fi} \\ &+ |\alpha| \\ &+ \text{number of zeroes in } \gamma \\ &- \text{number of zeroes in } \beta \\ \geq A_{i-1} &- 1 + (l_i - s) + 0 - s. \end{aligned}$$

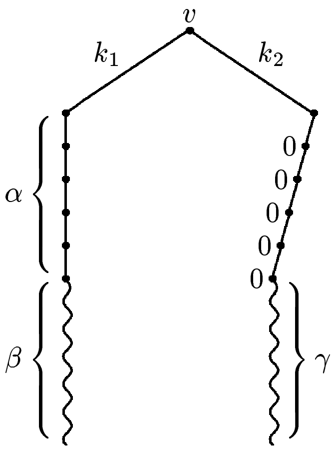

Figure 85. Zur Definition von s

It remains to be shown that $s \leq 1 + \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor$. This is certainly the case if $s = 0$. Suppose now that $s > 0$. We noted above that either β starts with a zero or that γ starts with a non-zero. In the first case consider node w which is reached from v via $k_1\alpha 1$, in the second case node w which is reached from v via $k_2 0^{|\alpha|} 0$. All leaf descendants of w lie properly between p_{i-1} and p_i. Furthermore, w has height $s - 1$ and hence at least a^{s-1} leaf descendants. This proves

$$a^{s-1} \leq p_i - p_{i-1} - 1 \leq p_i - p_{i-1} + 1$$

and hence

$$s \leq 1 + \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor.$$

Damit ist die obige Behauptung bewiesen. ∎

Using our claim repeatedly, we obtain

$$A_r \geq A_1 + \sum_{i=2}^{r} l_i - 2 \cdot \sum_{i=2}^{r} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor - 3 \cdot (r - 1).$$

Since $A_r \leq 1 + \lfloor \log_a N/2 \rfloor$ and $A_1 \geq 0$, this proves

$$\sum_{i=2}^{r} l_i \leq 3r - 3 + 1 + \lfloor \log_a N/2 \rfloor + 2 \cdot \sum_{i=2}^{r} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor$$

and hence

$$m \leq 3r - 1 + 2\lfloor \log_a N/2 \rfloor + 2 \cdot \sum_{i=2}^{r} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor .$$

This proves Fact 5. ∎

At this point, we combine everything and obtain a bound on the number of rebalancing operations.

$$\begin{aligned} SH + SP + F &= SH + SPNC + SPC + F \\ &\leq s + t + SPC + F && \text{(by Fact 2)} \\ &\leq 3(s+t) + 2m, && \text{(by Fact 4d)} \end{aligned}$$

where m is the total number of nodes on the paths from the new leaves and phantoms to the root in the final tree. Since the final tree is an (a, ∞)-tree and has $n + s$ leaves and phantoms leaves, we conclude from Fact 5

$$m \leq 3(s+t) + 2\Big(\lfloor \log_a (n+s) \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor \Big)$$

and hence

$$SH + SP + F \leq 9(s+t) + 4\Big(\lfloor \log_a (n+s) \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor \Big).$$

This proves Theorem 13. ∎

Theorem 13 provides us with a bound on total rebalancing cost if we start with a non-empty (a,b)-tree. We use Theorem 13 to generalize Theorem 12 to the case where we start with a non-empty list.

Theorem 13. *Let $a \geq 2$ and $b \geq 2a$. Let L be a sorted list of n elements represented as a level-linked (a,b)-tree with one finger established. Then any sequence* fehlt i.E. *of searches, finger creations, Fingerentfernungen, insertions and deletions, the total cost of the sequence is*

$$O(\log(n+s) + \textit{total cost of searches}),$$

wobei s die Anzahl der Einfüge-Operationen ist.

Proof: Let S be any sequence of search, finger creations, Fingerentfernungen, insertions and deletions containing exactly s insertions and t deletions. Let T_{final} be the (a,b)-tree, which represents list L after S is performed. Assume that we keep deleted elements as phantoms. Let the $n+s$ leaves (real leaves and phantoms) of T_{final} be named $1,\dots,n+s$ from left to right. Assign a label $l(p)$ to each leaf p, whose value is the number of leaves lying strictly to the left of p which were present initially and which did not become phantoms. These labels lie in the range
$[0\,..\,n]$ $[0\,..\,n-1]$.

Consider the searches in S which leads either to the creation of a new finger or the insertion or deletion of an element. Call an element of L **accessed** if it is either the source or the destination of such a search. (We regard an inserted item as the destination of the search which discovers where to insert it). Let $p_1 < p_2 < \cdots < p_l$ be the accessed items.

We shall consider graphs whose vertex set is a subset of $\{p_i;\ 1 \le i \le l\}$. We denote an edge joining $p_i \le p_j$ in such a graph by $p_i — p_j$ and we define the cost of this edge to be $\max(\lceil \log(l(p_j) - l(p_i) + 1)\rceil, 1)$. For each item p_i (except the initially fingered item) let q_i be the fingered item from which the search for p_i started. Each q_i is also in $\{p_i;\ 1 \le i \le l\}$, since each finger except the first must be established by a search. Consider the graph G with vertex set $\{p_i;\ 1 \le i \le l\}$ and edge set $\{(q_i,p_i);\ 1 \le i \le l$ and p_i is not the originally fingered item$\}$.

Some constant times the sum of edge cost in G is a lower bound on the total search cost, since $|l(p_i) - l(q_i)| + 1$ can only underestimate the actual distance between q_i and p_i when p_i is accessed. We shall describe a way to modify G, without increasing its cost, until it becomes

$$r_1 — r_2 — \cdots — r_k$$

where $r_1 \le r_2 \le \cdots \le r_k$ are the $k = s+t$ inserted or deleted leaves. Since the cost of this graph is $\sum_{1<i\le k} \max(\lceil \log(l(r_i) - l(r_{i-1}) + 1)\rceil, 1) \ge \frac{1}{8}(k + \sum_{1<i\le k} \log(r_i - r_{i-1} +$
i.D.anders $1))$, und somit dominieren die Suchkosten den ersten und den dritten Summanden der oberen Schranke aus Satz 13. (Für die Ungleichung beachte man, daß aus $r_i = r_{i-1}$ folgt $l(r_i) = l(r_{i-1})$ und daß aus $r_i > r_{i-1}$ folgt $l(r_i) - l(r_{i-1}) = r_i - r_{i-1} - 1$, da die Blätter zwischen r_i und r_{i-1} sämtlich von Anfang an vorhanden waren und nie gestrichen wurden. Daher gilt $\max(\log(l(r_i) - l(r_{i-1}) + 1), 1) \ge \frac{1}{8}(1 + \log(r_i - r_{i-1} + 1))$.) Daraus folgt die Behauptung.

The initial graph G is connected, since every accessed item must be reached from the initially fingered item. We first delete all but $l-1$ edges from G so as to leave a spanning tree; this only decreases the cost of G.

Next we repeat the following step until it is no longer applicable: Let $p_i — p_j$ be an edge of G such that there is an accessed item p_x satisfying $p_i < p_x < p_j$. Removing edge $p_i — p_j$ now divides G into exactly two connected components. If p_x is the same connected component as p_i, we replace $p_i — p_j$ by $p_x — p_j$; otherwise, we replace p_i—p_j by p_i—p_x. The new graph is still a tree spanning $\{p_i;\ 1 \le i \le l\}$ and the cost has not increased.

Finally, we eliminate each item p_j which is not an inserted or deleted leaf by transforming p_i—p_j—p_x to p_i—p_x and by removing edges p_j—p_x where there is no other edge incident to p_j. This does not increase the cost, and it results in the tree of inserted and deleted items

$$r_1 — r_2 — \cdots — r_k$$

as desired. ∎

Finger trees can be used efficiently for many basic operations such as union, intersection, difference, symmetric difference,....

Theorem 14. *Let A and B be sets represented as level-linked (a, b)-trees, $a \ge 2$ and $b \ge 2a$.*

Access zuviel

a) *Insert(x, A), Access(x, A), Delete(x, A), Concatenate(A, B), Split(x, A) take logarithmic time.*

b) *Let $n = \max(|A|, |B|)$ and $m = \min(|A|, |B|)$. Then $A \cup B$, $A \otimes B$, $A \cap B$ and $A \setminus B$ can be constructed in time $O\big(\log\binom{n+m}{m}\big)$.*

Proof: a) It is easy to see that Theorems 6 and 7 are true for level-linked (a, b)-trees.

b) We show how to construct $A \otimes B = (A - B) \cup (B - A)$. Assume w.l.o.g. $|A| \ge |B|$. The algorithm steht in Programm 21.

---

(1) establish a finger at the first element of A;
(2) **while** B not exhausted
(3) **do** take the next element, say x, of B
    and search for it in A starting at the finger;
(4)   insert x into or delete x from A, whatever is appropriate;
(5)   establish a finger at the position of x in A;
(6)   destroy the old finger
(7) **od**.

Program 21

Let $p_1, \ldots, p_m$, $m = |B|$, be the positions of the elements of B in the set $A \cup B$, let $p_0 = 1$. Then the above program takes time

$$O(\log(n+m) + \sum_{i=0}^{m-1} \log(p_{i+1} - p_i + 1))$$

by Theorem 14 and the observation that total search time is bounded by $\sum_{i=0}^{m-1} \log(p_{i+1} - p_i + 1)$.

This expression is maximized for $p_{i+1} - p_i = (n+m)/m$ for all i, and has value $O(\log(n+m)) + m\log((n+m)/m)) = O(m\log((n+m)/m)) = O\left(\log\binom{n+m}{m}\right)$.

In the case of $A \cup B$, we only make insertions in line (4). In the case of $A \cap B$, we collect the elements of $A \cap B$ in line (4) (there are at most m of them) and construct a level-linked (a,b)-tree for them afterwards in time $O(m)$.

Finally we have to consider $A \setminus B$. If $|A| \geq |B|$, then we use Program 21. If $|A| < |B|$ then we scan through A linearly, search for the of A in B as described above (roles of A and B reversed) and delete the appropriate elements from A. Apparently, the same time bound holds. ∎

Note that there are $\binom{n+m}{m}$ possibilities for B as subset of $A \cup B$. Hence $\log\binom{n+m}{m}$ fehlt i.E. im Entscheidungsbaum-Modell (s. II.1.6) is also a lower bound on the complexity of union and symmetric difference.

3.5.3.4. Fringe Analysis

Fringe analysis is a technique which allow us to treat some aspects of random (a,b)-trees. Let us first make the notion of random (a,b)-trees precise.

A **random $(a;b)$-tree** is grown by random insertions starting with the empty tree. Let T be an (a,b)-tree with j leaves. An insertion of a new element into T is random if each of the j leaves of T is equally likely to be split by the insertion. This can also be phrased in terms of search trees. Let S be a set with $j-1$ elements and let T be a search tree for $S \cup \{\infty\}$. Then the elements of S split the universe into j intervals. The insertion of a new element x is random if x has equal probability of lying in any one of the j intervals defined above.

We use fringe analysis to derive bounds on $\overline{n}(N)$, the expected number of nodes in a random (a,b)-tree with N leaves, i.e., a tree obtained by N random insertions from an empty tree. Then $\overline{n}(N)\cdot(2b-1)$ is the average number of storage locations fehlt required for a random (a,b)-tree with N leaves. Ein Knoten wird dabei durch $2b-1$ aufeinanderfolgende Speicherzellen dargestellt. Since any (a,b)-tree with N leaves has at least $(N-1)/(b-1)$ nodes and hence uses at least $(2b-1)(N-1)/(b-1)$ storage locations we can define

$$\overline{s}(N) = (N-1)/[(b-1)\cdot\overline{n}(N)]$$

as the storage utilization of a random (a,b)-tree. We show that $\overline{s}(N) \approx 0.69$ if $b = 2a$ and a is large. This in marked contrast to worst case storage utilization of $(a-1)/(b-1) \approx 0.5$.

For T an (a,b)-tree let $n(T)$ be the number of nodes of T and let $p_N(T)$ be the probability that T is obtained by N random insertions from the empty tree. Of course, $p_N(T)$ is zero if the number of leaves of T is unequal N. Then

$$\overline{n}(N) = \sum_T p_N(T)\cdot n(T).$$

Fringe analysis is based on the fact that most nodes of an (a,b)-tree are close to the leaves. Therefore a good estimate of $\overline{n}(N)$ can be obtained by simply estimating the number of nodes on the level one above the leaves.

Let $n_i(T)$ be the number of leaves of T which are sons of a node with exactly i sons, $a \le i \le b$. Then $\sum_i n_i(T) = |T|$, the number of leaves of T. Let

$$\overline{n}_i(N) = \sum_T n_i(T) \cdot p_N(T).$$

Then $\sum_i \overline{n}_i(N) = N$.

Lemma 11.

a) *Let T be an (a,b)-tree and let $r = \sum_i n_i(T)/i$. Then*

$$\frac{-1}{b-1} + \left(1 + \frac{1}{b-1}\right) \cdot r \;\le\; n(T) \;\le\; 1 + \left(1 + \frac{1}{a-1}\right) \cdot r.$$

b) *Let $\overline{r}(N) = \sum_i \overline{n}_i(N)/i$. Then*

$$\frac{-1}{b-1} + \left(1 + \frac{1}{b-1}\right) \cdot \overline{r}(N) \;\le\; \overline{n}(N) \;\le\; 1 + \left(1 + \frac{1}{a-1}\right) \cdot \overline{r}(N).$$

Proof: a) Let m be the number of nodes of T of height 2 or more. Then $n(T) = m + \sum_i n_i(T)/i$ since $n_i(T)/i$ is the number of nodes of arity i and height 1. Also

$$\frac{\sum_i n_i(T)/i - 1}{b-1} \;\le\; m \;\le\; \frac{\sum_i n_i(T)/i - 1}{a-1} + 1$$

since every node of height 2 or more has at most b and at least a sons (except for the root which has at least two sons) and since there are exactly $\sum_i n_i(T)/i$ nodes of height 1 in T. The 1 on the right hand side accounts for the fact that the degree of the root might be as low as 2.

b) immediate from part a) and the definition of $\overline{n}_i(N)$ and $\overline{n}(N)$. ∎

We infer from Lemma 11 that $\overline{n}(N)$ is essentially $\sum_i \overline{n}_i(N)/i$. We determine this quantity by setting up recursion equations for $\overline{n}_i(N)$ and by solving them.

Lemma 12. *Let $a \ge 2$ and $b = 2a$. Then*

$$\overline{n}_a(N+1) = \overline{n}_a(N) + a \cdot \frac{\overline{n}_b(N) - \overline{n}_a(N)}{N},$$

$$\overline{n}_{a+1}(N+1) = \overline{n}_{a+1}(N) + (a+1) \cdot \frac{\overline{n}_b(N) + \overline{n}_a(N) - \overline{n}_{a+1}(N)}{N},$$

$$\overline{n}_i(N+1) = \overline{n}_i(N) + i \cdot \frac{\overline{n}_{i-1}(N) - \overline{n}_i(N)}{N} \qquad \text{for } a+2 \le i \le b.$$

Proof: We proof the second equation and leave the two others to the reader. Let T be an (a, b)-tree with N leaves. Consider a random insertion into T resulting in T'. Note that $n_{a+1}(T') - n_{a+1}(T)$ grows by $a+1$ if the insertion is into a node with either exactly a (probability $n_a(T)/N$) or b (probability $n_b(T)/N$) leaf sons and that $n_{a+1}(T') - n_{a+1}(T)$ decreases by $a+1$ if the insertion is into a node of height 1 with exactly $a+1$ sons (probability $n_{a+1}(T)/N$). Thus

$$\begin{aligned}
&\overline{n}_{a+1}(N+1)\\
&= \sum_T p_N(T) \cdot \Big[n_{a+1}(T) + (a+1) \cdot \Big(\frac{n_b(T)}{N} + \frac{n_a(T)}{N} - \frac{n_{a+1}(T)}{N}\Big)\Big]\\
&= \overline{n}_{a+1}(N) + (a+1) \cdot \frac{\overline{n}_b(N) + \overline{n}_a(N) - \overline{n}_{a+1}(N)}{N}.
\end{aligned}$$

∎

Let $Q(N)$ be the column-vector $\big(\overline{n}_a(N), \ldots, \overline{n}_b(N)\big)^T$ und I die Einheitsmatrix. Then Lemma 12 can be written in matrix form as

$$Q(N+1) = \Big(I + \frac{1}{N} \cdot B\Big) \cdot Q(N)$$

where

$$B = \begin{pmatrix} -a & & & & & a \\ a+1 & -(a+1) & & & & a+1 \\ & a+2 & -(a+2) & & & \\ & & & \ddots & \ddots & \\ & & & & +b & -b \end{pmatrix}.$$

With $q(N) = (1/N) \cdot Q(N)$ this can be rewritten as

$$q(N+1) = \Big[I + \frac{1}{N+1} \cdot (B - I)\Big] \cdot q(N).$$

Note that matrix $B - I$ is singular since each column sums to zero. We show below that $q(N)$ converges to q, a right eigenvector of $B - I$ with respect to eigenvalue 0, as N goes to infinity. Also $|q(N) - q| = O(N^{-c})$ for some $c > 0$. This is shown in Theorem 17.

We will next determine $q = (q_a, \ldots, q_b)^T$. From $(B - I) = 0$ and $\sum_i \overline{n}_i(N)/N =$ 1 one concludes

anders

$$q_a = \frac{a}{(a+1)(b+1)(H_b - H_a)}$$

and

$$q_i = \frac{1}{(i+1)(H_b - H_a)} \qquad \text{for } a+1 \le i \le b$$

where $H_a = \sum_{i=1}^{a} 1/i$ is the a-th harmonic number. (Die q_i's, $a \le i \le b$, can be found as follows: take q_b as an indeterminate and solve $(B - I) \cdot q$ for $q_{b-1}, q_{b-2}, \ldots, q_{a+2}, q_a, q_{a+1}$ in that order. Then use $\sum_i q_i = 1$ to determine q_b). Thus

$$\begin{aligned}
\sum_{i=a}^{b} q_i/i &= \left(\frac{1}{(a+1)(b+1)} + \sum_{i=a+1}^{b} \frac{1}{i(i+1)}\right) \Big/ (H_b - H_a) \\
&= \left(\frac{1}{(a+1)(b+1)} + \sum_{i=a+1}^{b} \left(\frac{1}{i} - \frac{1}{i+1}\right)\right) \Big/ (H_b - H_a) \\
&= \left(\frac{1}{(a+1)(b+1)} + \frac{1}{a+1} - \frac{1}{b+1}\right) \Big/ (H_b - H_a) \\
&= \big[(b+1)(H_b - H_a)\big]^{-1}. \qquad \text{(since } b = 2a\text{)}
\end{aligned}$$

Theorem 15. *Let $a \ge 2$ and $b = 2a$, let $\epsilon > 0$ and let $\overline{s}(N) = (N-1)/\big[(b-1) \cdot \overline{n}(N)\big]$ be the storage utilization of a random (a,b)-tree with N leaves. Then*

$$|\overline{s}(N) - \ln 2| \ \le\ C/a + \epsilon$$

for some constant C independent of N and a and all sufficiently large N.

Proof: Note first that

$$\frac{N}{(b-1) \cdot (1 + 1/(a-1)) \cdot \overline{r}(N)} - \epsilon \ \le\ \overline{s}(N) \ \le\ \frac{N}{(b-1) \cdot (1 + 1/(b-1)) \cdot \overline{r}(N)} + \epsilon$$

for all sufficiently large N by nach Lemma 11b). Here $\overline{r}(N)/N = \sum_i q_i(N)/i = \sum_i q_i/i + O(N^{-c}) = 1/\big[(b+1) \cdot (H_b - H_a)\big] + O(N^{-c})$ for some constant $c > 0$. Furthermore $H_b - H_a = \sum_{i=a+1}^{b} 1/i = \ln 2 - O(1/a)$ since $\int_{a+1}^{b+1} (1/x)\,dx \le \sum_{i=a+1}^{b} 1/i \le \int_a^b (1/x)\,dx$. Thus $|\overline{s}(N) - \ln 2| \le C/a + \epsilon$ for all sufficiently large N and some constant C. ∎

fehlt Storage utilization of a random $(a, 2a)$-tree is thus about $\ln 2 \approx 69\%$ für große a. Fringe analysis can also be used to strengthen Theorem 9 on the total rebalancing cost. The bound in Theorem 9 was derived under the pessimistic assumption that every insertion/deletion decreases the value of the tree by 1. Fringe analysis provides us with a smaller bound at least if we restrict our attention to random sequences of insertions It is not hard to define what is meant by a random deletion; any leaf of the tree is removed with equal probability. However, it seems to be very hard to extend fringe analysis to random insertions *and* deletions.

We still have to prove convergence of sequence $q(N)$, $N = 1, 2, \ldots$. The answer is provided by the following general theorem which is applicable to fringe analysis problems of other types of trees also.

Theorem 16. *Let $H = (h_{ij})_{1 \le i,j \le m}$ be a matrix such that*

1) *all off-diagonal elements are non-negative and each column of H sums to zero and*
2) *there is an i such that for all j there are $j_0, \dots, j_k$ with $i = j_0$, $j = j_k$ and $h_{j_l, j_{l+1}} > 0$ for $0 \le l < k$.*

Then

a) *Let $\lambda_1, \dots, \lambda_m$ be the eigenvalues of H in decreasing order of real part. Then $\lambda_1 = 0 > \mathrm{Re}(\lambda_2) \ge \mathrm{Re}(\lambda_3) \ge \cdots \ge \mathrm{Re}(\lambda_m)$.*

b) *Let $q(0)$ be any non-zero m-vector and define $q(N+1) = (I + \frac{1}{N+1}H) \cdot q(N)$ for $N \ge 0$. Then $q(N)$ converges to q, a right eigenvector of H with respect to eigenvalues 0 and $|q - q(N)| = O(N^{\mathrm{Re}(\lambda_2)})$.*

Proof: a) H is singular since each column of H sums to zero. Thus 0 is a eigenvalue of H. Furthermore, all eigenvalues of H are contained in the union of the disks with center h_{jj} and radius $\sum_{i \ne j} |h_{ij}|$, $j = 1, 2, \dots, m$. This is the well-known Gerschgorin criterion (cf. Stoer/Bulirsch: "Numerische Mathematik II", page 77).
fehlt Thus all eigenvalues of H have non-positive real part. Eigenwerte ungleich 0 haben sogar einen echt negativen Realteil.

It remains to be shown that 0 is an eigenvalue of multiplicity one. This is the case iff the linear term in the characteristic polynomial of H (i.e., $\det(H - \lambda I)$) is non-null. The coefficient of the linear term is $\sum_i \det H_{ii}$ where H_{ii} is obtained from H by deleting the i-th row and column. Application of the Gerschgorin criterion to H_{ii} shows that all eigenvalues of H_{ii} have non-positive real part. Since $\det H_{ii} = \epsilon_1 \cdots \epsilon_{m-1}$ where $\epsilon_1, \dots, \epsilon_{m-1}$ are the eigenvalues of H_{ii}, we infer that either $\det H_{ii} = 0$ or $\mathrm{sign}(\det H_{ii}) = (-1)^{m-1}$. Beachten Sie, daß $\det H_{ii}$ reell ist. Thus $\sum_i \det H_{ii} = 0 \iff \det H_{ii} = 0$ four all i.

We will next show that $\det H_{ii} \ne 0$ where i satisfies assumption 2) of the theorem. Assume otherwise. Let $u = (u_1, \dots, u_{i-1}, u_{i+1}, \dots, u_m)$ be a left eigenvector of H_{ii} with respect to eigenvalue 0. Let u_t be a component of maximal absolute value in u and let $I = \{j;\ |u_j| = |u_t|\} \subseteq \{1, \dots, m\} - \{i\}$. Since $i \notin I$ and $I \ne \emptyset$ there must be $j \in I$, $k \notin I$, such that $h_{kj} > 0$ by assumption 2). We may assume w.l.o.g. that $u_j > 0$. Thus

$$\begin{aligned}
\sum_{l \ne i} u_l \cdot h_{lj} &\le u_j \cdot \left(\sum_{l \in I} h_{lj} + \sum_{l \notin I} \frac{|u_l|}{u_j} \cdot h_{lj} \right) \\
&\le u_j \cdot \left(\sum_l h_{lj} + \left(\frac{|u_k|}{u_j} - 1 \right) \cdot h_{kj} \right) \\
&< 0.
\end{aligned}$$

anders Widerspruch! Dabei folgt die zweite Ungleichung wegen $|u_l| \le u_j$ und $h_{jl} \ge 0$ für $l \ne j$ und die dritte wegen $\sum_l h_{lj} = 0$, $|u_k|/u_j < 1$ und $h_{kj} > 0$. Hence u is not left eigenvector of H_{ii} with respect to eigenvalue 0.

b) Let $f_n(x) = \prod_{j=1}^n (1 + x/j)$ and let $f(x) = \lim_{n\to\infty} f_n(x)$. Then $f(0) = f_n(0) = 1$. Also

Claim: $|f_n(x)| \le C \cdot n^{\mathrm{Re}(x)}$ *for some C depending on x.*

Proof: Note first that $|f_n(x)| = \prod_{j=1}^n |1 + x/j| = \exp\bigl(\sum_{j=1}^n \ln|1 + x/j|\bigr)$. Next note that

$$\begin{aligned}|1 + x/j|^2 &= |1 + \mathrm{Re}(x/j)|^2 + |\mathrm{Im}(x/j)|^2 \\ &\le |1 + \mathrm{Re}(x/j)|^2 (1 + c \cdot |\mathrm{Im}(x/j)|^2)\end{aligned}$$

for some constant c and all $j \ge j_0$ (c and j_0 depend on x). Taking square roots and substituting into the expression for $|f_n(x)|$ yields

$$\begin{aligned}|f_n(x)| \le \exp\Bigl(&\sum_{j<j_0} \ln|1 + x/j| \\ &+ \sum_{j=j_0}^{n} \bigl[\ln(1 + \mathrm{Re}(x/j)) + \ln((1 + c \cdot |\mathrm{Im}(x/j)|^2)^{1/2})\bigr]\Bigr) \\ \le C \cdot n^{\mathrm{Re}(x)}&\end{aligned}$$

for some constant C depending on x since $\ln(1 + \mathrm{Re}(x/j)) \le \mathrm{Re}(x)/j$ and $\ln((1 + c \cdot |\mathrm{Im}(x/j)|^2)^{1/2}) \le (c/2) \cdot |\mathrm{Im}(x)|^2/j^2$ (s. Anhang). ∎

Let $q(0)$ be some m-vector and let $q(N+1) = (I + \frac{1}{N+1} H) \cdot q(N)$ for $N \ge 0$. Then $q(N) = f_N(H) \cdot q(0)$ where $f_n(H) = \prod_{1 \le j \le n} (I + \frac{1}{j} H)$. We consider the matrix $f_n(H)$ in more detail. Let

$$J = T \cdot H \cdot T^{-1} = \begin{pmatrix} J_1 & & 0 \\ & \ddots & \\ 0 & & J_k \end{pmatrix}$$

be the Jordan matrix corresponding to H; $J_1, \dots, J_k$ are the blocks of the Jordan matrix. We have $J_1 = (0)$, i.e., J_1 is a one by one matrix whose only entry is zero. Also

$$J_l = \begin{pmatrix} \lambda_l & 1 & & 0 \\ & & \ddots & \\ & & \ddots & 1 \\ 0 & & & \lambda_l \end{pmatrix}$$

with $\mathrm{Re}(\lambda_l) < 0$. Then $f_n(H) = f_n(T^{-1} \cdot J \cdot T) = T^{-1} \cdot f_n(J) \cdot T$, as can be easily verified. Also

$$f_n(J) = \begin{pmatrix} f_n(J_1) & & & \\ & f_n(J_2) & & \\ & & \ddots & \\ & & & f_n(J_k) \end{pmatrix}.$$

We have (cf. Gantmacher, "Matrizen", Chapter 5, Example 2)

$$f_n(J_l) = \begin{pmatrix} f_n(\lambda_l) & \frac{f_n^{(1)}(\lambda_l)}{1!} & \frac{f_n^{(2)}(\lambda_l)}{2!} & \cdots & \frac{f_n^{(v_l-1)}(\lambda_l)}{v_l!} \\ & \ddots & \ddots & \ddots & \\ & & f_n(\lambda_l) & & \end{pmatrix},$$

where v_l is the multiplicity of λ_l and $f_n^{(h)}$ is the h-th derivative of f_n. The reader can easily verify this identity (for every polynomial f_n) by induction on the degree. Hence $f_n(J_1) = (1)$, the one by one matrix whose only entry is one and

$$f_n(J_l) = \begin{pmatrix} \epsilon_{1,1}(n) & \cdots & \epsilon_{1,v_l}(n) \\ & \ddots & \vdots \\ 0 & & \epsilon_{v_l,v_l}(n) \end{pmatrix},$$

where $\epsilon_{i,j} = O(n^{\mathrm{Re}(\lambda_2)})$. Thus $q(N)$ converges to $q = T^{-1} \cdot f(J) \cdot T \cdot q(0)$ as N goes to infinity. Here $f(J)$ is a matrix with a 1 in position (1,1) and all other entries equal to zero. Also $H \cdot q = (T^{-1} \cdot J \cdot T) \cdot (T^{-1} \cdot f(J) \cdot T \cdot q(0)) = T^{-1} \cdot J \cdot f(J) \cdot T \cdot q(0) = T^{-1} \cdot 0 \cdot T \cdot q(0)$ where 0 is the all zero matrix. Hence $H \cdot q = 0$, i.e., q is the right eigenvector of H with respect to eigenvalue 0. Finally, $|q(N) - q| = O(n^{\mathrm{Re}(\lambda_2)})$. ∎

3.8.2. Maintaining Dynamic Partitions of Linear Lists

In this section we treat the problem to maintain a dynamic partition of a linear list in intervals. Let B be a linear list of possibly marked objects. The marked objects are partitioning the linear list in intervals. We want to use the following operations on linear lists:

Find(x)	find then next marked object strictly to the right of x in the list;
Split(x)	mark the object x;
Union(x)	erase the mark of the object x;
Add(y, x)	insert a new unmarked object y directly in front of the object x in the list;
Erase(x)	remove the unmarked object x from the list.

For all five operations we consider the parameters of the operations to be pointers to the object x; in the static version of the problem (the operations Add and Erase are not allowed) we could also identify the linear list B with a beginning section of the integers. In this section we denote the length of the sequence B with N.

We will know several applications of the problem of interval partitioning: e.g. in the chapters about graph algorithms and multidimensional searching. The problem of interval partitioning is a variant of the priority queue problem. Let $S \subseteq B$ be the set of marked objects in the sequence B. Then Find(x) produces the successor of x in the set S (in the case of the priority queue problem we only can search for the smallest element in the set S), Split(x) inserts x in S, and Union(x) removes x from S; the operations Add and Erase "modify the universe" and don't have any analogy in the priority queue problem.

The problem of interval partitioning has a simple solution with running time $O(\log |S|)$ per operation: organize the set of marked objects in a balanced tree. In this section we describe a solution with running time $O(\log \log N)$; the time bound holds for the operations Find, Split and Union in the worst case and is amortized for the operations Add and Erase. So the new solution is better if the set of marked objects in the sequence B is dense enough.

The new solution is based on the Divide-and-Conquer Paradigma. Suppose that we divide the sequence B in about $\sqrt{N}$ subsequences of length about $\sqrt{N}$. For every subsequence we generate a representative who will get marked if one of the objects in the subsequence is. The representatives are collected in a linear list of representatives. Note that this list is of length about $\sqrt{N}$. Suppose now that we have a pointer to the object x to our disposal and want to compute Find(x). We consider two cases: Either the subsequence containing x also contains the result of the operation Find(x) or not. Both cases can be distinguished in time $O(1)$ if every representative has a pointer to the maximal marked object in the subsequence which it represents. Furtherly note that in the first case we reduced the length of the sequence to be considered from N to $\sqrt{N}$. This holds also for the second case, because we only find the successor of the representative of x in the list of the representatives (i.e. the subsequence containing x) and then we have to find the first marked object (we hold a pointer to it) in its subsequence. So in both cases time $O(1)$ is sufficient to reduce from N to $\sqrt{N}$ the length of the sequence to be considered. This implies time complexity $O(\log \log N)$; note the solution $T(N) = O(\log \log N)$ for the recursion equation $T(N) = T(\sqrt{N}) + O(1)$, as we will see later. Let's now take a look to the details.

Definition 1: A **layered tree** T for a sequence B of N objects consists of:

1) A **representative** $r(T)$ with components *mark*, *min* and *max* (the values of this components will be specified below);

and if $N > 4$:

2) A **Copy** B' of B, which is divided in subsequences $B'_1, \ldots, B'_m$, where $\frac{1}{2}\sqrt{N} \leq m \leq 2\sqrt{N}$ and every B'_i has between $\frac{1}{2}\sqrt{N}$ and $2\sqrt{N}$ elements;

3) layered trees $T'_1, \ldots, T'_m$ for the sequences $B'_1, \ldots, B'_m$;

4) a layered tree T_0 for the sequence R of representatives $r(T'_1), \ldots, r(T'_m)$. ∎

We call $T_1', \ldots, T_m'$ and T_r the **substructures** of T and T the **upper structure** of $T_1', \ldots, T_m'$ and T_r.

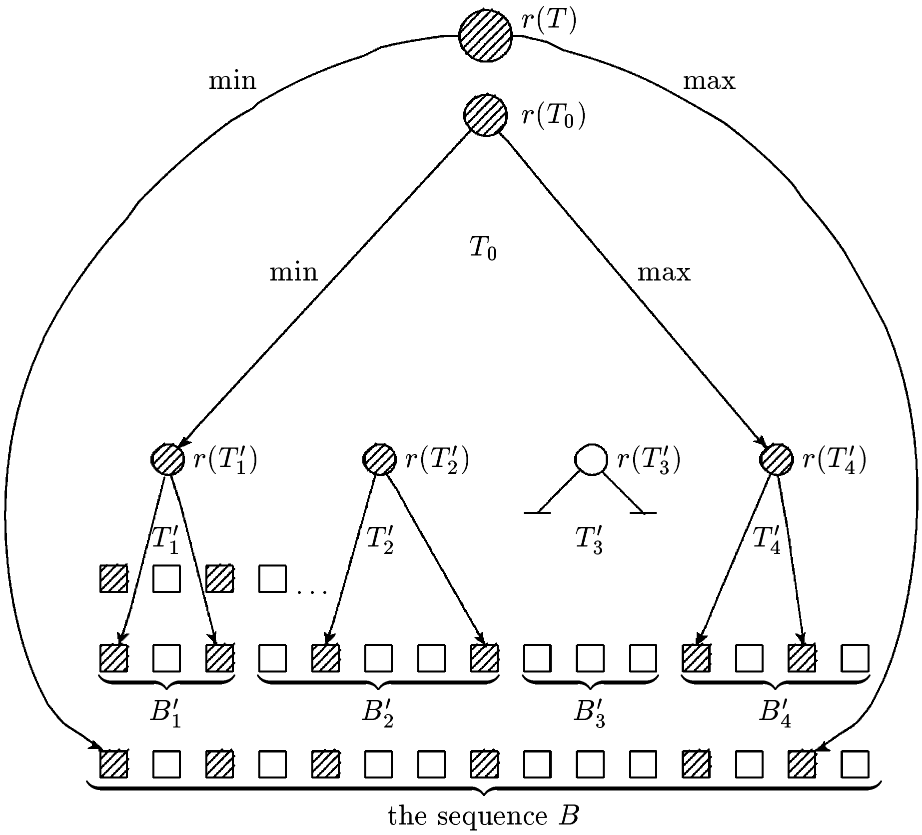

Figure 105. A layered tree

Figure 105 shows a layered tree for simplicity without the pointers connecting the objects. We realize layered trees by pointers as follows. Every object and every representative is described by a record of type node and consists of ten components:

mark	: boolean	*pred*	: ↑node
succ	: ↑node	*low*	: ↑node
high	: ↑node	*rep*	: ↑node
list	: ↑node	*size*	: integer
min	: ↑node	*max*	: ↑node

The components *pred* and *succ* organize the sequence B as a doubly linked linear list. The components *low* und *high* are pointers connecting nodes with its copies in substructures in both directions. *rep* connects every node with its representative.

The components *list* and *size* are only defined for representatives: *list* is a pointer to the first element of the doubly linked list for B and *size* contains the length of the list. We still have to define the components *mark*, *min* and *max*.

Definition 2: Let T with representative $r = r(T)$ be a layered tree for the sequence B.

1) $r.mark$ is true $\Longleftrightarrow$ the sequence B consists of at least one marked object. Then $r.min$ and $r.max$ point to the minimal respectively maximal marked object in the sequence B. If there is no marked object in B then the value of $r.min$ and $r.max$ is nil.

2) If B consists of at most one marked element then all substructures of T are "trivial", i.e., all nodes in these substructures are unmarked and all *min* and *max* pointers are nil.

3) If B consists of two or more objects then the copies (in B') of the marked objects of B are marked and the same rules are applied to all substructures T_i', $1 \leq i \leq k$ consisting of at least one marked object. Note that with the above the marks of the representatives $r(T_i')$, $1 \leq i \leq k$ are defined. Finally the same rules are applied to the substructure T_0. ∎

The recursive Procedure *buildstructure* from Program 21 constructs a layered tree in accordance with the Definitions 1 and 2.

---

procedure *buildstructure*(B : nodelist; **var** r : node;
const *triv* : boolean);
co Input: A doubly linked linear list B of objects
Effect: Construction of a layered tree for B and output of the representative r
Running time: $O(|B| \log\log |B|)$
The layered tree is trivial if *triv* is true.
oc
$N \leftarrow |B|$;
$r \leftarrow$ new node;
if *triv*
then $r.mark \leftarrow$ false; $r.min \leftarrow r.max \leftarrow$ nil
else $r.mark \leftarrow \vee_{k \in B} k.mark$;
$r.min \leftarrow$ leftmost marked element of B;
$r.max \leftarrow$ rightmost marked element of B;
$s \leftarrow$ number of the marked elements of B
fi;
$r.high \leftarrow nil$;
$r.size \leftarrow N$;
for all $k \in B$ **do** $k.rep \leftarrow r$; $k.low \leftarrow$ nil **od**;
if $N > 4$
then $B' \leftarrow \emptyset$;
for all $k \in B$
do $k' \leftarrow$ new node;

```
        if triv
        then k'.mark ← false
        else k'.mark ← k.mark
        fi;
        k'.min ← nil;
        k'.max ← nil;
        k'.high ← k;
        k.low ← k';
        B' ← B' ∪ {k'}
    od;
    divide B' into blocks B'_1, B'_2, ..., B'_n of size between √N and (3/2)√N;
    co then (2/3)√N ≤ n ≤ √N oc
    R ← ∅;
    for i = 1 to n
    do organize B'_i as a doubly linked linear list;
        buildstructure(B'_i, r', triv or s ≤ 1);
        R ← R ∪ {r'}
    od;
    buildstructure(R', r, triv or s ≤ 1)
fi
end.
```

Program 22

Lemma 13. *Let B be a sequence of N objects. A layered tree for T needs $O(N \log\log N)$ storage locations and may be constructed in time $O(N \log\log N)$.*

Proof: We show that a layered tree for a sequence of N objects consists of at most $O(N \log\log N)$ nodes. This implies the space bound and also the time bound, because the Procedure *buildstructure* needs $O(1)$ time units per node. To prove the bound for the number of nodes we show that there are only $O(\log\log N)$ copies per node, i.e., there are only $O(\log\log N)$ hierarchy levels. Let $g : \mathbb{R} \mapsto \mathbb{R}$ be defined by

$$g(x) = \begin{cases} 1 & \text{if } x \le 4; \\ 1 + g(\lfloor 2\sqrt{x} \rfloor) & \text{if } x > 4. \end{cases}$$

Then $g(N)$ is an upper bound for the hierarchy levels in a layered tree for a sequence of N objects. A simple inductive argument shows $g(x) \le 2 + (\log\log x)/\log(4/3)$. Indeed $g(x) \le 3$ for $x < 9$. If $x \ge 9$

$$\begin{aligned} g(x) &= 1 + g(\lfloor 2\sqrt{x} \rfloor) \\ &\le 1 + g(x^{3/4}) && \text{(because } 2\sqrt{x} \le x^{3/4}) \\ &\le 1 + 2 + (\log\log x^{3/4})/\log(4/3) && \text{(by induction hypothesis)} \\ &\le 2 + 1 + \log((3/4)\log x)/\log(4/3) \\ &\le 2 + 1 + (\log(3/4) + \log\log x)/\log(4/3) \\ &\le 2 + (\log\log x)/\log(4/3) \end{aligned}$$

∎

Now we take a look to the operations Find, Split and Union. To formulate these algorithms we suppose the existence of a boolean function leftof(x, y) which is defined for two marked objects x and y. It yields true $\iff$ x is in front of y in the sequence B. In all ours applications of layered trees the existence of the function leftof with running time $O(1)$ is obvious because the sequence B will always be a sorted sequence of keys and leftof can be computed by comparison of two keys. The general case will be treated in Exercise 23.

```
function find(k : node) : node;
co Input:          object k of the sequence B
   Output:         next marked successor of k in B or nil
   Running time:   O(1), if the output is nil; O(log log |B|) otherwise
oc
r ← k.rep;
if r.size ≤ 4
then find the next marked successor by linear search in time O(1)
else if (r.mark ≠ false) and leftof(k, r.max)
     then co find has a defined value oc
          if r.max = r.min
          then co exactly one marked node oc
               print r.max
          else co two or more marked nodes oc
               x ← find(k.low);
               if x = nil
               then co The call find(k.low) needed time O(1) oc
                    x ← find(k.low.rep);
                    print x.min.high
               else print x.high
               fi
          fi
     else co find has the value nil co
          print nil
     fi
fi
end.
```

Program 23

Lemma 14. *The Procedure find of Program 23 finds the next successor in time $O(\log\log N)$.*

Proof: The correctness of Program 23 is easy to show. If the representative r of k is not marked or k is not left of $r.max$ then *find* prints nil. Otherwise, we search for a marked successor of the copy $k.low$ in its subsequence. If there is such a successor then we print it. (More precisely, we print a copy one level higher in hierarchy.) If there is no such successor then we find the marked successor of the representative of the copy $k.low$ and print the node described by its *min* pointer. In both cases we correctly compute the next marked successor.

Also the time bound is easy to verify. Let be $r.size > 4$. Note first, that the then case always prints a defined value and that the else case has constant time complexity and prints nil. This implies running time $O(1)$ in the case of output nil. If the output is not equal to nil then we recursively call *find* either once or twice. If we call it twice then the first call prints nil and the running time is $O(1)$. Hence the total running time is $O(g(N))$, where g is defined as in Lemma 1. ∎

Now we turn to operation Split.

```
procedure split(k : Knoten);
co before:       k is an unmarked object in the sequence B, T is a valid
                 (i.e., follows Definitions 1 and 2) layered tree for B
   afterwards:   k.mark = true and T is valid
   Running time: O(1), if there is no marked object in B before split;
                 O(log log |B|) otherwise
oc
k.mark ← true;
r ← k.rep;
if r.mark = false
then co before the structure was empty oc
     r.mark ← true;
     r.min ← k;
     r.max ← k
else if r.min = r.max
     then co one marked object before Split co
          x ← r.min;
          if leftof(k, r.min)
          then r.min ← k
          else r.max ← k
          fi;
          co B contains the two marked objects x and k; the
             substructures must be initialized in accordance with the
             property 3 of Definition 2
          oc
          if r.size > 4
          then co there are substructures oc
               split(x.low);      co time O(1) oc
               split(x.low.rep);  co time O(1) oc
```

```
                if x.low.rep = k.low.rep
                then split(k.low)        co a nontrivial call oc
                else split(k.low);       co time O(1) oc
                     split(k.low.rep)    co a nontrivial call oc
                fi
            fi
    else    co two or more marked objects oc
            if leftof(r.max, k) then r.max ← k fi;
            if leftof(k, r.min) then r.min ← k fi;
            if k.low.rep.mark = false
            then split(k.low);       co time O(1) oc
                 split(k.low.rep)    co a nontrivial call oc
            else split(k.low);       co a nontrivial call oc
            fi
    fi
fi
end.
```

Program 24

Lemma 15. *Program 24 realizes the operation Split and has time complexity* $O(\log\log N)$.

Proof: To prove correctness we must show that the conditions of Definition 2 are fulfilled. This is obviously if B contains no or at least two marked objects before the Split. If B contains exactly one marked object before the Split then all substructures of B were trivial before the Split and the substructures T_0 and the substructures containing $x.low$ and $k.low$ are not trivial after the Split. This is precisely the effect of the recursive calls.

Let's turn to the time complexity. The essential observation (cf. the comments in the program) is that at most one recursive call splits the substructure containing a marked object and that a Split applied to a substructure containing no marked object costs $O(1)$. Hence the running time is $O(g(N)) = O(\log\log N)$. ∎

```
procedure union(k: node, var newmin, newmax: node);
co    before:          k is a marked object of the sequence B,
                       T is a valid layered tree for B
      afterwards:      k.mark = false, furthermore T is valid and newmin resp. newmax
                       is the new leftmost resp. rightmost marked object in B
      running time     O(1) if k is the only marked object in B
                       O(log log |B|) otherwise
oc
k.mark ← false;
r ← k.rep;
```

```
if r.min = r.max
then co just one marked node k = r.min = r.max before Union oc
        r.min ← nil;
        r.max ← nil;
        r.mark ← false
else  co r.min ≠ r.max oc
        if r.size ≤ 4
        then the obvious operations on the linear list B
        else  co there are nontrivial substructures; first
                    we erase recursively the mark of k.low and
                    recompute the new min and max pointers oc
                UNION(k.low, x, y);
                if x = nil
                then co k.low was the only marked node in the
                        substructure and hence the recursive call needed
                        time O(1)   oc
                        UNION(k.low.rep, x1, y1);
                        x ← x1.min;
                        y ← y1.max
                fi;
                if k = r.min then r.min ← x.high fi;
                if k = r.max then r.max ← y.high fi;
                if r.min = r.max
                then co now only one marked object;
                            we must trivialize the substructures oc
                        UNION(r.min.low, , );       co time O(1) oc
                        UNION(r.min.low.rep, , )  co time O(1) oc
                fi
        fi
fi;
newmin ← r.min;
newmax ← r.max
end.
```

Program 25

Lemma 16. *Program 25 realizes the operation Union in time $O(\log\log N)$.*

Proof: This is obvious with the comments in the program. ∎

The operations Add and Erase modify the linear list B by adding or erasing an object. We realize Erase(x) by erasing x and all of its copies in the pointer structure. This may lead to an violation of the balance criterion from Definion 1. Hence we store the outermost structure (in the hierarchy) which becomes to tight in a variable *outbal* and later we will call the Procedure *rebal* which will repair the balance. On the execution of Add(x, y) we similarly add y and its copies to the pointer structure giving them the same representatives as the corresponding copies of x; then we store the outermost structure which has become too heavy in a variable *outbal* and call *rebal* to repair the balance. The details are described in the two following programs.

---

procedure *add1*(y, x);
co before: $x \in B$ and T is a valid layered tree for B
effect: adds the unmarked object y directly in front of x to B and stores the representative of the first substructure which is too heavy in a variable *outbal* and the size of its upper structure in w.
running time: $O(\log\log|B|)$
oc
$r \leftarrow x.rep$;
$y.rep \leftarrow r$;
add y in front of x in the doubly linked list $r.list$;
$r.size \leftarrow r.size + 1$;
if $y.high \neq$ nil
then co the upper structure exists **oc**
  $r' \leftarrow y.high.rep$;
  if $(r.size > 2\sqrt{r'.size})$ **and** $(outbal = \text{nil})$
  then co overflow **oc**
    $outbal \leftarrow r$;
    $w \leftarrow r'.size$
  fi
fi;
if $r.size > 4$
then co substructure **oc**
  $z \leftarrow$ new node;
  $z.mark \leftarrow$ false;
  $z.high \leftarrow y$;
  $y.low \leftarrow z$;
  add1$(z, x.low)$
fi
end.

Program 26

---

procedure *erase*1(x);
co before: x is an unmarked object in B and T is a valid layered tree for B.
effect: erases x from B and stores the representative of the first

```
                    substructure which is too tight in a variable
                    outbal and the weight of its upper structure in a variable w;
   running time:    O(log log |B|)
oc
outbal ← nil;
r ← y.rep;
x.rep ← r;
erase x from the doubly linked list r.list;
r.size ← r.size − 1;
if x.high ≠ nil
then co the upper structure exists oc
     r' ← x.high.rep;
     if (r.size < ½√(r'.size)) and (outbal = nil)
     then co underflow oc
          outbal ← v;
          w ← r'.size
     fi
fi;
if r.size > 4
then co substructure oc
     ERASE1(x.low)
fi
end.
```

Program 27

Both procedures obviously have running time $O(\log\log|B|)$ and perform their specification. We still have to describe the procedure *rebal* which is called in the case if $outbal \neq$ nil after the call of *add1* and *erase1*, resp.

```
procedure rebal(r, w);
co before:          r is a representative of a substructure, say S;
                    the upper structure is of size w;
   effect:          joins S with a brother structure and repairs
                    the balance criterion of Definition 1;
   running time:    O(log log |C|), where C is the linear list described by s.
oc
if r.succ = nil
then r' ← r.pred
else  r' ← r.succ
fi;
C ← r'.list ∪ r.list
R ← r.rep.list;
divide C in sequences B_1, B_2, ..., B_c of size between √w and (3/2)√w;
for i = 1 to c
```

do *buildstructure*(B_i, x);
  insert x in front of r in R
od;
erase r and r' from R;
buildstructure$(R, r.rep)$
end.

Program 28

Lemma 17. *Let B be a sequence of N objects. buildstructure(B, r) followed by N Add and Erase operations has total costs $O(N \cdot (\log\log N)^2)$.*

Proof: Let S_0 be the layered tree constructed by *buildstructure*(B, r) and

$$S_0 \xrightarrow{Op_1} S_0' \xrightarrow{rebal} S_1 \xrightarrow{Op_2} S_1' \xrightarrow{rebal} S_2 \xrightarrow{Op_3} \quad \cdots \quad \xrightarrow{rebal} S_{N-1}$$

a sequence of N Update operations each of them followed by a call of *rebal*, where $Op_i \in \{add1,\ erase1\}$ for $1 \le i \le N-1$. We estimate the total costs of the N calls of the Procedure *rebal*. We do this by using the banking paradigma. We define for a layered tree S

$$\mathrm{bal}(S) = 2\log\log N \cdot \sum_{\substack{r \text{ representative} \\ w \text{ size of the upper structure}}} \max(0,\ r.size - \frac{3}{2}\sqrt{w},\ \sqrt{w} - r.size).$$

Then $\mathrm{bal}(S_0) = 0$, because S_0 was constructed by a call of *buildstructure* and so $\sqrt{w} \le r.size \le \frac{3}{2}\sqrt{w}$ for all representatives r in S. Note further that a call of *add1* or *erase1* changes the *size* components of $O(\log\log N)$ representatives and hence $\mathrm{bal}(S_i') \le \mathrm{bal}(S_i) + O((\log\log N)^2)$ for $0 \le i \le N-1$.

Let's take a look to the transition from S_i' to S_{i+1}. If $S_i' = S_{i+1}$, i.e., there was no need of a call of *rebal* then $\mathrm{bal}(S_i') = \mathrm{bal}(S_{i+1})$ and the transition has costs 0. Suppose now that a call of *rebal*$(outbal_i, w_i)$ was necessary. Such a call has costs $O(outbal_i.size \cdot \log\log N)$. Furthermore $\mathrm{bal}(S_{i+1}) \le \mathrm{bal}(S_i') - outbal_i.size \cdot \log\log N$. It follows that the declining of the balance can pay the costs of the transition. Therefore Lemma 5 is prooven. ∎

Now we want to formulate the main statement of this section.

Theorem 17. *Find, Union, Split, Add and Erase may be computed in time $O(\log\log N)$ per operation and need $O(N)$ storage locations. The time bound holds for Find, Union and Split in the worst case and for Add and Erase in the amortized case.*

Proof: First we describe an improvement of the time bound from Lemma 5 to $O(N \log\log N)$. Let B be a linear list of N objects. We divide B in groups of size between $\log\log N$ and $4\log\log N$ elements. Each group is realized by a doubly linked list. The $O(N/\log\log N)$ list headers are stored in a layered tree. This tree needs $O(N)$ storage locations. The costs of the operations Find, Union and Split are only increased by an additive factor $O(\log\log N)$ and remain therefore $O(\log\log N)$. Usually the operations Add and Erase are applied on linear lists; only every $O(\log\log N)$-th Update operation needs a "real" rectification in the layered tree (if a linear list gets too long, i.e., its length is $1 + 4\log\log N$ we divide it in two lists of length nearly $2\log\log N$. If a list gets too short, i.e., its length is $-1+\log\log N$ then we fuse it either with its brother list (if the length of the brother list is at most $2\log\log N$) or we steal $(\log\log N)/2$ objects from the brother list (if the length of the brother list is at least $2\log\log N$)), and therefore the amortisized costs of an Add or Erase operation are reduced to $O(\log\log N)$.

Now it is easy to complete the proof. We apply the above strategy on a sequence of N Update operations and reconstruct a totaly new structure. This costs $O(N\log\log N)$. ∎

We end this section with a short remark on the term "layered tree". Consider a sequence of N elements and its complete binary tree. Suppose now that all paths of marked leaves to the root are marked. We then could realize the operation Find by ascending the tree until we meet (coming from the left) a marked path and then following the marked path. A more economial method is to jump in the half of the tree (by using the *rep* pointers) to a node of height $\frac{1}{2}\log N$ and there to test (by using the *max* pointers) if we are already on a marked path. Then we continue this process in the half of the tree either above or below this node. This essentially corresponds to the binary search on the paths of the tree. The name "layered tree" emphasizes this clever using of the layer structure of trees.