CHAPTER 6

# Data Structures


## K. MEHLHORN

*Fachbereich 14, Informatik, Universität des Saarlandes, D-6600 Saarbrücken, FRG*

## A. TSAKALIDIS

*Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece, and Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece*


*Contents*


1. Introduction . . . . . 303
2. The dictionary problem . . . . . 305
3. The weighted dictionary problem and self-organizing data structures . . . . . 319
4. Persistence . . . . . 323
5. The *Union-Split-Find* problem . . . . . 324
6. Priority queues . . . . . 326
7. Nearest common ancestors . . . . . 328
8. Selection . . . . . 329
9. Merging . . . . . 331
10. Dynamization techniques . . . . . 332

References . . . . . 334



HANDBOOK OF THEORETICAL COMPUTER SCIENCE
Edited by J. van Leeuwen

## 1. Introduction

Data structuring is the study of *concrete implementations* of frequently occurring *abstract data types*. An abstract data type is a set together with a collection of operations on the elements of the set. We give two examples for the sake of concreteness.

In the data type *dictionary* the set is the powerset of a universe $U$, and the operations are *insertion* and *deletion* of elements and the *test of membership*. A typical application of dictionaries are symbol tables in compilers. In the data type *priority queues* the set is the powerset of an ordered universe $U$ and the operations are insertion of elements and finding and deleting the minimal element of a set. Priority queues arise frequently in network optimization problems, see, e.g. Chapter 10 on graph algorithms (this Handbook).

This chapter is organized as follows. In Sections 2 and 3 we treat unweighted and weighted dictionaries respectively. The data structures discussed in these sections are ephemeral in the sense that a change to the structure destroys the old version, leaving only the new version available for use. In contrast, persistent structures, which are covered in Section 4, allow access to any version, old or new, at any time. In Section 5 we deal with the *Union-Split-Find* problem and in Section 6 with priority queues. Section 7 is devoted to operations on trees and Sections 8 and 9 cover selection and merging. Finally, Section 10 is devoted to dynamization techniques. Of course, we can only give an overview; we refer the reader to the textbooks of Knuth [112], Aho, Hopcroft, Ullman [2, 4], Wirth [219], Horowitz and Sahni [98], Standish [187], Tarjan [195], Gonnet [83], Mehlhorn [145], and Sedgewick [181] for a more detailed treatment. In this section we build the basis for the latter sections. In particular, we review the types of complexity analysis and the machine models used in the data structure area.

The machine models used in this chapter are the pointer machine (pm-machine) [112, 179, 193] and the random-access machine (RAM-machine) [2]. A thorough discussion of both models can be found in Chapter 1 of this Handbook. In a pointer machine memory consists of a collection of *records*. Each record consists of a fixed number of *cells*. The cells have associated *types*, such as *pointer*, *integer*, *real*, and access to memory is only possible by "pointers". In other words, the memory is structured as a directed graph with bounded out-degree. The edges of this graph can be changed during execution. Pointer machines correspond roughly to high-level programming languages without arrays. In contrast, the memory of a RAM consists of an array of cells. A cell is accessed through its address and hence address arithmetic is available. We assume in both models that storage cells can hold arbitrary numbers and that the basic arithmetic and pointer operations take constant time. This is called the uniform cost assumption. All our machines are assumed to be deterministic except when explicitly stated otherwise.

Three types of complexity analysis are customary in the data structure area: worst-case analysis, average-case analysis and amortized analysis. In a *worst-case analysis* one derives worst-case time bounds for each single operation. This is the most frequent type of analysis. A typical result is that dictionaries can be realized with

$O(\log n)$ worst-case cost for the operations *Insert*, *Delete*, and *Access*; here $n$ is the current size of the set manipulated.

In an *average-case analysis* one postulates a probability distribution on the operations of the abstract data type and computes the expected cost of the operations under this probability assumption. A typical result is that static dictionaries (only operation *Access*) can be realized with $O(H(\beta_1, \ldots, \beta_n))$ expected cost per access operation; here $n$ is the size of the dictionary, $\beta_i$, $1 \leqslant i \leqslant n$, the probability of access to the $i$th element and $H(\beta_1, \ldots, \beta_n)$ is the entropy of the probability distribution. (This type of analysis is extensively discussed in Chapter 9 on analysis of algorithms and data structures.)

In an *amortized analysis* we study the worst-case cost of a sequence of operations. Since amortized analysis is not very frequently used outside the data structure area, we discuss it in more detail. An amortized analysis is relevant whenever the cost of an operation can fluctuate widely and only averaging the costs of a sequence of operations over the sequence leads to good results. The technical instrument used to do the averaging is an account (banker's view of amortization) or potential (physicist's view of amortization) associated with the data structure.

In the banker's view of amortization, which was implicitly used by Brown and Tarjan [33] and then more fully developed by Huddleston and Mehlhorn [101], we view a computer as coin- or token-operated. Each token pays for a constant amount of computer time. Whenever a user of a data structure calls a certain operation of the data type, we charge him a certain amount of tokens; we call this amount the amortized cost of the operation. The amortized cost does in general not agree with the actual cost of the operation (that is the whole point of amortization). The difference between the amortized and the actual cost is either deposited into or withdrawn from the account associated with the data structure. Clearly, if we start with no tokens initially, then the total actual cost of a sequence of operations is just the total amortized cost minus the final balance of the account.

More formally, let *bal* be a function which maps the possible configurations of the data structure into the real numbers. For an operation *op* which transforms a configuration $D$ into a configuration $D'$, define the amortized cost [184] by

$$amortized\_cost(op) = actual\_cost(op) + bal(D') - bal(D).$$

Then for a sequence $op_1, \ldots, op_m$ of operations we have

$$\sum_i amortized\_cost(op_i) = \sum_i actual\_cost + bal(D_m) - bal(D_0)$$

where $D_0$ is the initial data structure and $D_m$ is the final data structure. In particular, if $bal(D_0) = 0$ and $bal(D) \geqslant 0$ for all data structures $D$, i.e., we never borrow tokens from the bank, the actual cost of the sequence of operations is bounded by its amortized cost. In the physicist's view of amortization which was developed by Sleator and Tarjan [184], we write $pot(D)$ instead of $bal(D)$, call it the potential of the data structure and view deposits and withdrawals as increases and decreases of potential. Of course, the two viewpoints are equivalent. A more thorough discussion of amortized analysis can be found in [198] and [145, Vol. I, Section I.6.1.1].

Let us illustrate these concepts by an example, the data type *counter*. A counter takes nonnegative integer values, the two operations are *Set-to-zero* and *Increment*. We realize counters as sequences of binary digits. Then *Set-to-zero* returns a string of zeros and *Increment* has to add 1 to a number in binary representation. We implement *Increment* by first increasing the least significant digit by 1 and then calling a (hidden) procedure *Propagate-carry* if the increased digit is 2. This procedure changes the digit 2 into a zero, increases the digit to the left by 1 and then calls itself recursively. The worst-case cost of an increment operation is, of course, unbounded. For the amortized analysis we define the potential of a string $\ldots \alpha_1 \alpha_1 \alpha_0$ as the sum $\sum_i \alpha_i$ of its digits. Then *Set-to-zero* creates a string of potential zero. Also, a call of procedure *Propagate-carry* has actual cost 1 and amortized cost 0 since it decreases the potential of a string by 1. Finally, changing the last digit has actual cost 1 and amortized cost 2 since it increases the potential by 1. Thus the total cost of a sequence of one *Set-to-zero* operation followed by $n$ *Increments* is bounded by $1 + 2 \cdot n$, although some increments may cost as much as $\log n$.

There is one other view of amortization which we now briefly discuss: the buyer's and seller's view. A buyer (user) of a data structure wants to be charged in a predictable way for his uses of the data structure, e.g. the same amount for each call of *Increment*. A seller (implementer) wants to account for his actual uses of computing time. Amortization is the trick which makes both of them happy. The seller charges the buyer two tokens for each call of increment; the analysis given above ensures the seller that his actual costs are covered and the buyer pays a uniform price for each call of *Increment*.

## 2. The dictionary problem

Let $S \subseteq U$ be any subset of the universe $U$. We assume that an item of information $inf(x)$ is associated with every element $x \in S$. The *dictionary problem* asks for a data structure which supports the following three operations:

(1) *Access*$(x)$: return the pair (**true**, $inf(x)$) if $x \in S$, and return **false** otherwise;
(2) *Insert*$(x, inf)$: replace $S$ by $S \cup \{x\}$ and associate the information $inf$ with $x$;
(3) *Delete*$(x)$: replace $S$ by $S - \{x\}$.

Note that operations *Insert* and *Delete* are destructive, i.e., the old version of set $S$ is destroyed by the operations. The nondestructive version of the problem is discussed in Section 4. In the *static dictionary problem* only operation *Access* has to be supported.

Since the mid-fifties many sophisticated solutions were developed for the dictionary problem. They can be classified into two large groups, the comparison-based and the representation-based data structures. Comparison-based data structures only use comparisons between elements of $U$ in order to gain information, whilst the less restrictive representation-based structures mainly use the representation, say as a string over some alphabet, of the elements of $U$ for that purpose. Typical representatives of the two groups are respectively search trees and hashing.

### 2.1. Comparison-based data structures

We distinguish two types of data structures: explicit and implicit. An implicit data structure for a set $S$, $|S| = n$, uses a single array of size $n$ to store the elements of $S$ and

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

Fig. 1. Program 1.

only O(1) additional storage. The explicit data structures use more storage; mostly for explicit pointers between elements.

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 $S[1 \ldots n]$, i.e. $S[i]=x_i$, and let $x \in U$. In order to decide $x \in S$, we compare $x$ with some table element and then proceed with either the lower or the upper part of the table (see Program 1 in Fig. 1). Various algorithms can be obtained from this scheme by replacing lines (2) and (7) by specific strategies for choosing next. Linear search is obtained by $next \leftarrow low$, binary search by $next \leftarrow [(low+high)/2]$. The worst-case access time is $O(n)$ for linear search and $O(\log n)$ for binary search. It is clear that $\log n$ is also a lower bound for the access time for any comparison-based data structure.

*2.1.1. Implicit data structures*

There are a number of ways to implement dictionaries implicitly. If we store the elements in an unordered list, then update can be done in constant time, but searching requires linear time. On the other hand, if the dictionary is maintained as an array sorted in increasing order, then searching can be done in logarithmic time, but updates may require that all the elements in the array be moved. Rotated lists are arrays that can be sorted into increasing order by performing a cyclic shift (rotation) of the elements. They are not much more difficult to search, but only half the elements have to be moved in the worst case.

Munro and Suwanda [156] were the first to explicitly consider the dynamic implicit dictionary problem. They showed that if the elements are stored partially sorted in a triangular grid, then search and update can be performed in $O(\sqrt{n})$ steps. Using blocks of rotated lists (sorted relative to one another), they were able to improve the search time to $O(\log n)$, keep the number of moves per update at $O(\sqrt{n})$, and only increase the number of comparisons per update to $O(\sqrt{n}\log n)$. Combining these ideas, they also produced an implicit dictionary that can be searched or updated in $O(n^{1/3}\log n)$ time. By using rotated lists in a recursive manner, Frederickson [67] was able to achieve $O(\log n)$ search time and $O(n^{\sqrt{2/\log n}}(\log n)^{3/2})$ update time. Munro [152, 153] created implicit dictionaries that use $O((\log n)^2)$ time for both search and update. His basic approach is to have the order of elements within blocks of the array implicitly represent pointers and counters.

Since the $O((\log n)^2)$ upper bound falls short of the $O(\log n)$ upper bound for explicit structures, the interesting question of lower bounds arises. Munro and Suwanda [156] proved that their triangular grid scheme is optimal within a restricted class of implicit structures. Borodin et al. [28] prove a trade-off between search and update time which is valid for *all* implicit data structures. In particular, their result implies that if the update time is constant then the search time is $\Omega(n^{\varepsilon})$ for some constant $\varepsilon > 0$.

A related problem is the searching problem for semisorted tables. Assume that the elements of the set $S$ can be arranged to any one of $p$ different permutations. Then Alt and Mehlhorn [9] showed that $\Omega(p^{1/n})$ comparisons are necessary for an access operation in the worst case, under the restrictive assumption that all comparisons must involve the element sought for. Their lower bound is even true in the average case and for nondeterministic algorithms. If the $p$ permutations are the set of extensions of a single partial order, then a more precise bound was given by Linial and Saks [127]; they show that $O(\log N)$ stops are necessary and sufficient where $N$ is the number of ideals in the partial order; we remind the reader that a subset $A$ of a partially ordered set $(P, \geqslant)$ is an *ideal* if $x \in A$ and $y < x$ implies $y \in A$.

Partial orders naturally arise as the result of preprocessing. Borodin et al. [29] considered the problem of determining the trade-off between preprocessing and search time. If $P(n)$ and $S(n)$ denote the number of comparisons performed to respectively preprocess and search an initially unsorted array of length $n$, then, in the worst case, $P(n) + n \log S(n)$ is $\Omega(n \log n)$. Mairson [132] proved that this result also holds in the average case. Many of the lower-bound results mentioned were recently unified and extended by McDiarmid [137].

*2.1.2. Explicit data structures or search trees*

A search tree for a set $S = \{x_1 < \cdots < x_n\}$ is a full binary tree, i.e., each node has either two or no son, with $n$ leaves (=nodes with no son). The $n$ leaves are labelled by the elements from $S$ from left to right. The internal nodes (=nodes with two sons) are labelled by elements of $U$ such that elements in the left subtree of a node $v$ are labelled no larger than $v$ and elements in the right subtree are labelled larger than $v$. Figure 2 shows a search tree for $\{1, 7, 19, 23\} \subseteq \mathbb{Z}$. The root could be labelled by any integer between 7 and 18 inclusive. Our convention for storing the set $S$ is frequently called *leaf-oriented*. The alternative *node-oriented* organization, where the elements are stored in the internal nodes will not be discussed; most results carry over with only small modifications.

An access operation takes time proportional to the height of the tree (=length of the

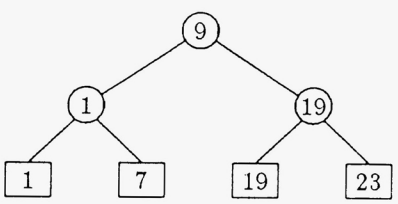


Fig. 2. A search tree for $\{1, 7, 19, 23\}$.

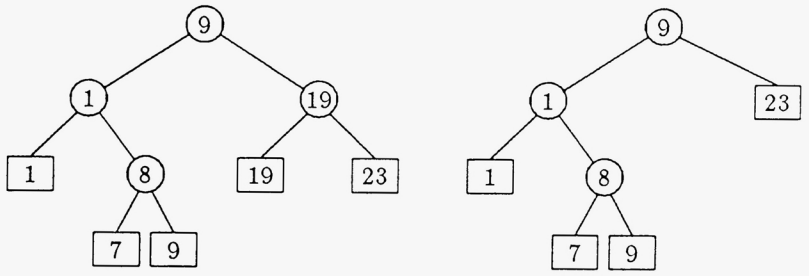


Fig. 3. Insertion of 9 and deletion of 19.

longest path from the root to a leaf) in the worst case: compare the element $x$ to be sought for with the label of the root, proceed to the right subtree if it is larger and to the left subtree otherwise. Insertions and deletions are also easily accomplished. In order to insert $x$, we search for $x$ and then replace the leaf where the search ended by a tree with two leaves. In order to delete $x$, we only have to delete the leaf labelled $x$ and its parent from the tree. Figure 3 shows the tree of Fig. 2 after the insertion of 9 and the deletion of 19.

The worst-case time for an *Access*, *Insert* or *Delete* is $O(n)$ since trees may degenerate to linear lists. On the average, we can expect to do much better.

Consider the following scenario. We start with a tree consisting of a single leaf. Before the $n$th step we have a tree with $n$ leaves. One of the leaves is chosen at random and replaced by a tree with two leaves. Let $D(n)$ be the expected total depth of the $n$ leaves before the $n$th step, i.e.,

$$D(n)=\sum_T\left[p(T)\sum_i d_i(T)\right]$$

where $p(T)$ is the probability that a certain tree $T$ with $n$ leaves arises and $d_i(T)$ is the depth of the $i$th leaf of $T$. Then $D(1)=1$ and

$$D(n+1)-D(n)=\sum_T p(T)\left(2+\sum_i d_i(T)/n\right)$$

since the increase in depth is $d_i(T)+2$ if the $i$th leaf of $T$ is replaced and the probability for this event is $p(T)/n$. Thus $D(n+1)-D(n)=2+1/n\cdot D(n)$ or $D(n+1)=2+(1+1/n)\cdot D(n)$. This recurrence relation has solution

$$D(n)=2n\cdot\sum_{i=1}^{n}1/i-n=O(n\log n).$$

This shows that under random insertions we can expect an expected access time of $O(\log n)$. An excellent survey of the behavior of random trees is [83]. Recent results can be found in [44, 45].

We have seen that depth is a crucial parameter for the behavior of search trees. This led to the development of balanced trees where the depth is guaranteed to be $O(\log n)$. We distinguish two major types of balanced trees, the height- and the weight-balanced

trees. In the first class the height of subtrees is balanced; it includes AVL-trees [1], (2–3)-trees [2], B-trees [15], HB-trees [162], red-black trees [89], half-balanced trees [160], $(a, b)$-trees [101] and balanced-binary-trees [196]. The behavior of AVL-trees was studied by Foster [65, 66], Knuth [112], Brown [31], Mehlhorn [142], Mehlhorn and Tsakalidis [151], and Tsakalidis [205]. In the second class the weight of the subtrees is balanced; it includes the BB[$\alpha$]-trees [159] and the internal-path trees [82].

We treat only (2, 4)-trees and refer the reader to the textbooks mentioned above for the other classes. In a (2, 4)-tree all leaves have the same depth and every internal node has between 2 and 4 children. We use $\rho(v)$ to denote the number of children (= arity) of a node $v$. In a node $v$ we store $\rho(v) - 1$ labels in order to guide searches. We will describe the insertion and deletion algorithms and their *amortized analysis* next. For a tree $T$, let the potential of $T$ be twice the number of 4-nodes (= nodes of arity 4) plus the number of 2-nodes.

Let us consider an insertion first. We start by adding a new leaf. This may convert the parent from a 4-node to a 5-node, which is not allowed in a (2, 4)-tree. We split such a 5-node into a 2-node and a 3-node. This may create a new 5-node, which we split in turn. We continue splitting newly created 5-nodes, moving up the tree, until either the root splits or no new 5-node is created, see Fig. 4. If the root splits, we create a new root, a 2-node, causing the tree to grow in height by 1. The time needed for the insertion is proportional to 1 plus the number of splits.

Let us define the actual time of an insertion to be 1 plus the number of splits. Then the amortized time of an insertion is at most 3: each split costs 1 but converts a node that was originally a 4-node into a 2-node and a 3-node, for a net potential drop of 1; in addition, the insertion can create one new 2-node or 4-node.

Deletion is an inverse process, only slightly more complicated. To delete a given item, we destroy the leaf containing it. This may make the parent a 1-node. If this 1-node has a neighboring sibling that is a 3-node or a 4-node, we move a child of this neighbor to the 1-node and the deletion stops. (This is called *borrowing*.) If the 1-node has a neighboring sibling that is a 2-node, we combine the 1-node and the 2-node. (This is called *fusing*.) Fusing may produce a new 1-node, which we eliminate in the same way. We move up the tree eliminating 1-nodes until either a borrowing occurs or the root

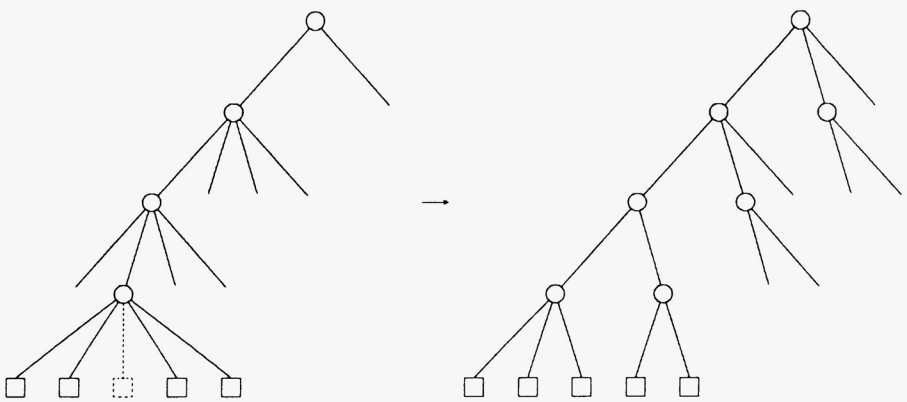

Fig. 4.

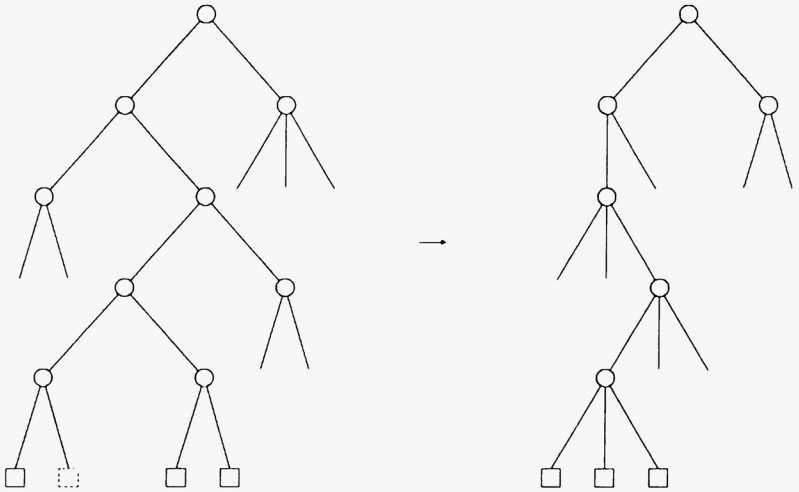

Fig. 5.

becomes a 1-node, which we destroy. (See Fig. 5). The time needed for the deletion is proportional to 1 plus number of fusings.

Let us define the actual time of a deletion to be 1 plus the number of fusings. Then the amortized cost is at most 2: each fusing costs 1 but converts two nodes that were originally 2-nodes into a 3-node, for a net potential drop of 2; in addition, the deletion can create one new 2-node.

We summarize. The amortized cost of an insertion or deletion is O(1) although the actual cost may be as large as O(log $n$). This result is due to Huddleston and Mehlhorn [101] and Maier and Salveter [131]. The particularly simple proof given above is due to Hoffmann et al. [95]. Levcopoulus and Overmars [126] have recently shown that constant insertion and deletion time can also be obtained in the worst case and not only in the amortized sense. It is open whether their scheme can also support the various extensions of (2, 4)-trees to be discussed next.

How frequently do expensive insertions and deletions arise? Huddleston and Mehlhorn [101] showed that the number of insertions and deletions which cause $t$ fusings or splittings decreases as an exponential function of $t$. (2, 4)-Trees are strongly related to red-black trees [89], symmetric binary B-trees [14] and half-balanced trees [160]. All of the results about amortized rebalancing cost mentioned above carry over to these trees. Moreover, red-black trees and half-balanced trees have the additional property that only O(1) structural changes of the tree are required after an insertion or deletion; all other changes concern only bookkeeping information [160, 196]. These properties and their simplicity make red-black trees a good choice for practical implementations of balanced trees; they make them also the ideal basis for *finger search trees*. Finger search trees represent ordered lists into which one can maintain pointers, called fingers, from which searches can start; the time for a search, insertion or deletion is O(log $d$), where $d$ is the number of items between the search starting point and the accessed item. The O(log $d$) bound can either be achieved in the amortized sense [33, 101] or in the worst-case sense [88, 117, 100, 194, 203, 204]. Finger trees lead to optimal algorithms for the basic set operations union, intersection, difference, ...; i.e.,

these operations take time

$$O\left(\log\binom{n+m}{\min(n,m)}\right)$$

when applied to sets of size $n$ and $m$ respectively [101]. Another application of finger trees is efficient list splitting. Suppose that we have a list of $n$ items which we want to split into sublists of length $d$ and $n-d$ respectively. The position of the split is determined by a finger search starting from both ends of the list. The finger search takes time $O(\log \min(d, n-d))$ and this is also a bound on the amortized cost of the split [95]. Consider now a sequence of splits, i.e., we start with a single list of length $n$ which we split repeatedly until we obtain $n$ lists of length 1 each. The worst-case cost $T(n)$ of this process is given by

$$T(1)=0 \quad \text{and} \quad T(n)=\max_{1\leqslant d\leqslant n-1}[T(d)+T(n-d)+O(\log \min(d,n-d))]$$
$$\text{for } n\geqslant 1.$$

This recurrence has solution $T(n)=O(n)$, i.e., the amortized cost of each split is $O(1)$.

Frequently, the same search has to be performed on many lists. Several researchers [209, 128, 216, 55, 41, 104] observed that the naive strategy of locating the key separately in each list by binary search is far from optimal and that more efficient techniques frequently exist. Chazelle and Guibas [39] distilled from these special-case solutions a general data structuring technique and called it *fractional cascading* which supports under very general assumptions the search for an item in $d$ lists out of $n$ lists in time $O(d+\log N)$, where $N$ is the total size of the $n$ lists. This has numerous applications to computational geometry. Mehlhorn and Näher [146] studied dynamic fractional cascading and showed that the amortized analysis of balanced trees carries over to the more general situation of fractional cascading; more precisely, they showed that insertions and deletions have amortized cost $O(\log \log N)$ and that a search for an item in $d$ lists out of $n$ lists takes time $O((d+\log N)\log \log N)$.

Weight-balanced trees [159] share many of the properties of height-balanced trees; their implementation is more cumbersome, however. On the other hand, they have the following *weight-property* [217, 25]: A node of weight $w$ (i.e., $w$ descendants) participates in only $O(n/w)$ structural changes of the tree when a sequence of $n$ insertions and deletions is processed. The weight-property makes weight-balanced trees superior to height-balanced trees whenever structural changes of the tree are very costly, e.g., when their cost depends linearly on the size of the subtree changed. This is frequently the case in applications to multidimensional search or computational geometry where trees are augmented with substantial additional information. In these applications weight-balanced trees guarantee good amortized behavior which cannot be obtained with height-balanced trees.

### *2.2. Representation-based data structures*

In these data structures the representation, say as a string of digits, of the elements stored is used to compute their position in memory. We consider interpolation search (Section 2.2.1), hybrid data structures (Section 2.2.2) and hashing (Section 2.2.3).

*2.2.1. Interpolation search*

We first consider static interpolation search, which is an implicit data structure. Interpolation search was suggested by Peterson [170] as a method for searching in sorted arrays. It is obtained from Program 1 (Fig. 1), by replacing lines (2) and (7) by

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

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 $S[0]=0$, $S[n+1]=1$, $x=1/(n+1)$ and $0<S[i]<x$ for $1 \leqslant i \leqslant n$. Then $next=low$ always and interpolation search deteriorates to linear search. The average-case behavior is much better. Average access time is $O(\log \log n)$ under the assumption that the keys $x_1, \ldots, x_n$ are drawn independently from a uniform distribution over the open interval $(x_0, x_{n+1})$. This was shown by Yao and Yao [222], Pearl et al. [168] and Gonnet et al. [86].

A very intuitive explanation of the behavior of interpolation search can be found in [169], where a variant called *binary interpolation search* is presented. We discuss this variant: 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 could obtain $\log \log n$ access time. So let us consider the question whether there is a fast (at least on the average) way to find the node on the search path 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 the linear search jumps over $\sqrt{n}$ 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 the 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 $Access(x, S)$ is as follows. Let $low=1$, $high=n$ and let $next=\lceil p \cdot n \rceil$ be defined as above; here $p=(x-x_0)/(x_{n+1}-x_0)$. If $x>S[next]$ then compare $x$ with $S[next+\sqrt{n}]$, $S[next+2\sqrt{n}], \ldots$ until an $i$ is found with $x \leqslant S[next+(i-1) \cdot \sqrt{n}]$. This will use up to $i$ comparisons. If $x<S[next]$ then we proceed analogously. In any case, the subtable of size $\sqrt{n}$ thus found is then searched by applying the same method recursively. Let $p_i$ be the probability that $i$ or more steps are required in the search for the subtable. Then the expected cost of the search for the subtable is $O(\sum_i p_i)$. Also, if $i$ or more steps are required then the actual position of element $x$ in the array differs by more than $i\sqrt{n}$ from the expected position (which is $pn$) and hence $p_i=O(1/i^2)$ by an application of the Chebyshev inequality. This shows that time $O(1)$ is spent at each level of the recursion and hence the expected access time is $O(\log \log n)$.

Santoro and Sidney [176] proposed another variant called *interpolation-binary* search which has $O(\log \log n)$ average-case access time for the uniform distribution and $O(\log n)$ worst-case time.

Willard [215] has shown that the log log $n$ asymptotic retrieval time of interpolation search does not hold for most nonuniform probability distributions. However, he was able to modify interpolation search such that its expected running time is O(log log $n$) on static $\mu$-random files where $\mu$ is any *regular* probability density. A density $\mu$ is regular if there are constants $b_1$, $b_2$, $b_3$, $b_4$ such that $\mu(x)=0$ for $x<b_1$ or $x>b_2$, $\mu(x)\geqslant b_3\geqslant 0$ and $|\mu'(x)|\leqslant b_4$ for $b_1\leqslant x\leqslant b_2$. A file is $\mu$-random if its elements are drawn independently according to density $\mu$. It is important to observe that Willard's algorithm does not have to know the density $\mu$. Rather, its running time is O(log log $n$) on $\mu$-random files provided that $\mu$ is *regular*. Thus his algorithm is fairly robust.

Dynamic interpolation search, i.e., data structures which support insertions and deletions as well as interpolation search, was discussed by Frederickson [67] and Itai, Konheim, Rodeh [105]. Frederickson presents an implicit data structure which supports insertions and deletions in time $O(n^{\varepsilon})$, $\varepsilon>0$, and accesses with expected time O(log log $n$). The structure of Itai, Konheim and Rodeh has expected insertion time O(log $n$) and worst-case insertion time $O((\log n)^2)$. It is claimed to support interpolation search, although no formal analysis of its expected behavior is given. Both papers assume that the files are generated according to the uniform distribution.

Mehlhorn and Tsakalidis [150] extend dynamic interpolation search to nonuniform distributions. They achieve an amortized insertion and deletion cost of O(log $n$) and an expected amortized insertion and deletion cost of O(log log $n$). The expected search time on files generated by $\mu$-random insertions and random deletions is O(log log $n$) provided that $\mu$ is a *smooth* density. An insertion is $\mu$-random if the key to be inserted is drawn from density $\mu$. A deletion is random if every key present in the current file is equally likely to be deleted. These notions of randomness are called $I_r$ and $D_r$ respectively in [113]. A density $\mu$ is *smooth* if there are constants $d$ and $\alpha<1$ such that, for all $a$, $b$, $c$ with $a<c<b$ and all integers $m$ and $n$, $m=[n^{\alpha}]$,

$$\int_{c-(b-a)/m}^{c} \hat{\mu}(x)\,\mathrm{d}x \leqslant dn^{-1/2}$$

where $\hat{\mu}(x)=0$ for $x<a$ or $x>b$ and $\hat{\mu}(x)=\mu(x)/p$ for $a\leqslant x\leqslant b$ where $p=\int_a^b \mu(x)\,\mathrm{d}x$. Every regular density of Willard is smooth in this sense.

### 2.2.2. *Hybrid data structures*

This group of data structures combines arrays and pointers. A main representative is the digital search tree or trie. The trie was proposed by Fredkin [71] as a simple way to structure a file by using the digital representation of its elements; e.g. we may represent $S=\{121, 102, 211, 120, 210, 212\}$ by the trie in Fig. 6.

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 to use an array of length $k$ for every internal node of the tree. Then operations *Access*, *Insert* and *Delete* are very fast and are very simple to program; in particular, if the reverse of all elements of $S$ are stored (in our example this would be the set $\{121, 201, 112, 021, 012, 212\}$), then the program in Fig. 7 will realize operation *Access*($x$). This program takes time $O(l)=O(\log_k N)$ where $N=|U|$. Unfor-

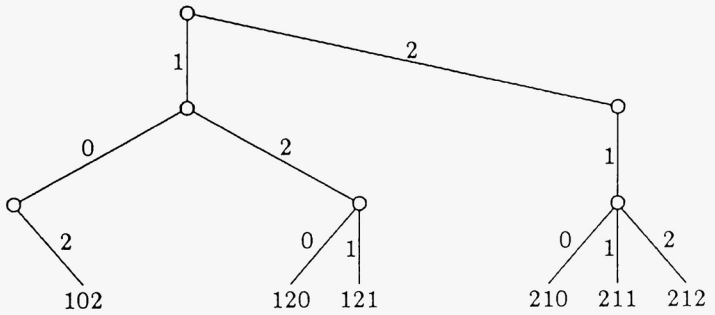


Fig. 6. A trie.

```
(1) v←root;
(2) y←x;
(3) do l TIMES (i, y)←(y MOD k, y DIV k);
(4)    v←ith son of v
(5) od;
(6) if x=CONTENT[v] then "yes" else "no".
```

Fig. 7. Program 2.

tunately, 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 1 and use up to space $O(k)$.

There is a very simple method to reduce the 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 Fig. 8. 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 plus the

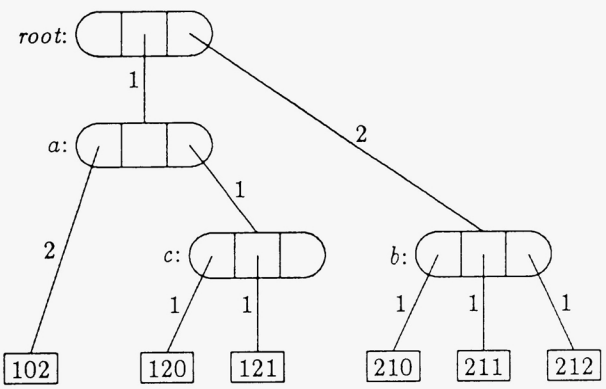


Fig. 8. The compressed trie.

length of the eliminated chain of nodes of degree 1. 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 $(1+2)$ digit in the son. The algorithms for *Access*, *Insert* and *Delete* become slightly more complicated for compressed tries but still run in time $O(l)$.

Thus 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 $O(k \cdot n)$.

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* being supported, there is a technique developed by Tarjan and Yao [200] which can store a set $S$ ($S \subseteq U$, $|S| = n$ and $|U| = N$) in an $n$-ary trie using $O(n)$ storage locations (of $O(\log n)$ bits each) such that it can support operation *Access* in time $O(\log_n N)$ worst case and $O(1)$ expected case. Their technique combines a trie structure with a method for compressing tables by using double displacements. This double displacement method is an elaboration of a single displacement method suggested by Aho and Ullman [6] and Ziegler [225] for compressing parsing tables.

Willard [214] introduces two dynamic trie structures, the *P*-fast trie and the *Q*-fast trie. A *P*-fast trie uses space $O(n\sqrt{\log N}\, 2^{\sqrt{\log N}})$ for representing a set $S$ ($S \subseteq U$, $|S| = n$, $|U| = N$) and can support access, insertion and deletion in worst-case time $O(\sqrt{\log N})$. The *Q*-fast trie is derived from the *P*-fast trie by using a pruning technique; it has the same time bounds but needs only $O(n)$ space.

Tries also support the following three additional operations:

(1) *Successor*$(x)$: find the least element in the set $S$ with key value greater than $x$;

(2) *Predecessor*$(x)$: find the greatest element in the set $S$ with key value less than $x$;

(3) *Subset*$(x_1, x_2)$: find (and produce) the list of those elements of $S$ whose key value lies between $x_1$ and $x_2$.

The *P*-fast and *Q*-fast tries can support the operations *Successor*$(x)$ and *Predecessor*$(x)$ in time $O(\sqrt{\log N})$ and *Subset*$(x_1, x_2)$ in time $O(\sqrt{\log N} + k)$, where $k$ is the size of the output.

Willard [213] also proposed another static trie structure, called *Y*-fast trie, which uses $O(n)$ space and supports *Access*$(x)$, *Successor*$(x)$ and *Predecessor*$(x)$ in worst-case time $O(\log \log N)$ and *Subset*$(x_1, x_2)$ in $O(\log \log N + k)$. This method uses perfect thashing (see Subsection 2.2.3.3.).

### *2.2.3. Hashing*

The ingredients are very simple: an array $T[0 \ldots m-1]$, the hash table, and a function $h: U \to [0 \ldots m-1]$, the hash function. $U$ is the universe; we will assume $U = [0 \ldots N-1]$ 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 $T[h(x)]$. 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. We briefly present both methods and then discuss two techniques for choosing the hash function $h$: perfect

and universal hashing. We do not treat the average behavior of hashing in detail but rather refer the reader to Chapter 14 on the complexity of finite functions.

*2.2.3.1. 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 $h(x) = i$. Operation *Access*$(x, S)$ is realized by the following program:

(1) compute $h(x)$;

(2) search for $x$ in list $T[h(x)]$.

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

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

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

The worst-case complexity of hashing is now 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 and any one of the three operations costs $O(|S|)$ time units.

The average-case behavior of hashing is much better. The expected cost of a sequence of $n$ insertions, deletions and accesses is $O((1 + \beta/2)n)$, where $\beta = n/m$ is the maximal load factor of the table. This is true under the following probability assumption:

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

(2) All elements of $U$ are equally likely as 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 $1/|U|$.

These assumptions imply that the value $h(x_k)$ of the hash function on the argument of the $k$th operation is uniformly distributed in $[0 \ldots m-1]$, i.e. $\Pr(h(x_k) = i) = 1/m$ for all $k \in [1 \ldots n]$ and $i \in [0 \ldots m-1]$. Thus the expected length of any one of the lists is at most $i/m$ after the $i$th operation and therefore the expected cost of the $(i+1)$st operation is $O(1 + i/m)$. Hence the total expected cost of all $n$ operations is $O((1 + \beta/2)n)$. More on the expected analysis of hashing with chaining can be found in Chapter 14.

*2.2.3.2. 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 the function $h(x, i)$ is to use the linear combination of two hash functions $h_1$ and $h_2$:

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

Hashing with open addressing does not require any additional space. However, its performance becomes poor when the load factor is nearly 1.

The average-case behavior of hashing with open addressing is easy to determine under the unrealistic assumption that the sequence $h(x, i)$ is a random permutation of the table positions and that the cost of an *Insert* is $1 + \min\{i: T[h(x, i)] \text{ is not occupied}\}$. Let $C(n, m)$ be the expected cost of an *Insert* when $n$ items are already stored in a table of size $m$. Then $C(n, m) = 1 + n/m \cdot C(n-1, m-1)$ if $n > 0$, since one probe is always required, this probe inspects an occupied position with probability $n/m$ and since in this case an insertion into a table of size $m-1$ holding already $n-1$ items is still required, and $C(0, m) = 1$. Thus

$$C(n, m) = (m+1)/(m-n+1) \approx 1/(1-\beta),$$

where $\beta = n/m$ is the load factor. The expected cost of an *Access* operation is $O((1/\beta) \cdot \ln 1/(1-\beta))$ by a similar analysis. More on the expected-case behavior of hashing with open addressing can be found in Chapter 14.

*2.2.3.3. Universal hashing.* Universal hashing was first described by Carter and Wegman [37]. It is a method to deal with the basic problem of hashing: its linear worst-case behavior. We saw in Subsection 2.2.3.1 that hashing provides us with O(1) expected access time and $O(n)$ worst-case access time. 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 favors 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 the 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. More precisely, let $c \in \mathbb{R}$ and $N, m \in \mathbb{N}$. A collection $H \subseteq \{h: h: [0 \ldots N-1] \to [0 \ldots m-1]\}$ is $c$-universal if for all $x, y \in [0 \ldots N-1]$, $x \neq y$,

$$|\{h: h \in H \text{ and } h(x) = h(y)\}| \leqslant c \cdot |H|/m.$$

Carter and Wegman [37] analyzed the expected behavior of universal hashing under the following assumptions:

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

(2) Hashing with chaining is used.

They showed that the expected cost of an *Access*, *Insert* or *Delete* operation is $O(1 + c\beta)$, where $\beta = n/m$ is the load factor, if a $c$-universal class of hash functions is used.

They also gave examples of universal classes. Let $m, N \in \mathbb{N}$ and let $N$ be a prime. For $a, b \in [0 \ldots N-1]$ let $h_{a,b}(x) = ((ax+b) \bmod N) \bmod m$ for all $x$. Then the class $H = \{h_{a,b}: a, b \in [0 \ldots N-1]\}$ is 4-universal. A member of this class requires $O(\log N)$ random bits for its selection. Mehlhorn [143] shows that there is a 8-universal class whose members can be selected by $O(\log m + \log\log N)$ bits and also shows that this is optimal. Other universal classes are described in [134].

*2.2.3.4. Perfect hashing.* Perfect hashing is the simplest solution to the collision avoidance problem. We just postulate that there are no collisions, i.e., the hash function operates injectively on the set to be stored. A function $h: [0 \ldots N-1] \to [0 \ldots m-1]$ is a *perfect hash function* for $S \subseteq [0 \ldots N-1]$ if $h(x) \neq (y)$ for all $x, y \in S$, $x \neq y$.

Perfect hashing was first discussed by Sprugnoli [186]. He describes heuristic methods for constructing perfect hash functions. Fredman, Komlós and Szemerédi [73] show that very simple perfect hash functions exist whenever $m \geqslant 3n$, $n = |S|$. The hash functions can be found in deterministic time $O(nN)$ and probabilistic time $O(n)$, can be evaluated in time $O(1)$ and are given as programs of length $O(n \log N)$ bits. The program size was improved to $O(n \log n + \log\log N)$ by Mehlhorn [145, p. 138], and to $O(n \log\log n + \log\log N)$ by Jacobs and van Emde Boas [106]. A lower bound for program size is $\Omega(n^2/m + \log\log N)$ [143].

Perfect hashing was first developed for static sets $S$. Aho and Lee [5] showed how to deal with insertions and deletions. They describe a hashing scheme which supports accesses and deletions in worst-case time $O(1)$ and insertions in expected time $O(1)$. The expectation is computed with respect to random inputs; cf. assumption (2) in Subsection 2.2.3.1.

A further improvement was made by Dietzfelbinger et al. [46]. This scheme combines the advantages of universal and perfect hashing. Accesses and deletions take worst-case time $O(1)$ and insertions take time $O(1)$ on the average. However, the average is now computed with respect to the random choices made by the algorithm and is worst-case with respect to inputs.

*2.2.3.5. Extendible hashing.* Our treatment of hashing in the previous subsections was based on the assumption that main memory is large enough to completely contain the hash table. Let us assume now that this assumption is no longer warranted. Let us also assume that the transfer of information between main and secondary memory is in pages of size $b$. In this situation extendible hashing as described by Fagin et al. [59] can be used. Similar schemes are known under the names dynamic hashing [122], virtual hashing [129] and expandable hashing [110].

Let $h: U \to \{0,1\}^*$ be an injective function. For an integer $d$ and $x \in U$ let $h_d(x)$ be the prefix of $h(x)$ of length $d$ and for a set $S \subseteq U$ let $d(S)$ be the minimal $d$ such that $|\{x \in S: h_d(x) = a\}| \leqslant b$ for all $a \in \{0,1\}^d$, i.e. $h_d$ partitions the set $S$ into subsets of size at most $b$. Extendible hashing uses a table $T[0 \ldots 2^{d(S)} - 1]$, called the directory, and some number of buckets of capacity $b$ to store set $S$. The directory entries are pointers to buckets. An element $x \in S$ is accessed by computing $i = h_d(x)$, looking up $T[i]$ and then accessing the bucket pointed to by $T[i]$. In this way at most two accesses to secondary memory are ever needed. The first access is to the page containing $T[i]$ and the second

access is to the bucket containing $x$. We allow different directory entries to point to the same bucket as long as the following buddy principle holds: If $r < d(S)$, $a \in \{0,1\}^r$, $a_1, a_2 \in \{0,1\}^{d(S)-r}$, $a_1 \neq a_2$ and $T[aa_1] = T[aa_2]$ then necessarily $T[aa_1] = T[aa_3]$ for all $a_3 \in \{0,1\}^{d(S)-r}$.

Insertions are easily processed except when a bucket overflows. In this case $d(S)$ increases by 1, the size of the directory doubles, and the overflowing bucket is split into two.

The expected behavior of extendible hashing is treated in the papers mentioned above and more completely in Flajolet [62]; cf. also Chapter 14. He shows that the expected number of buckets is $n/(b \ln 2)$ and that the expected size of the directory is about $e/(b \ln 2)n^{1+1/b}$. A variant of extendible hashing [191] achieves linear expected directory size and O(1) expected access time.

## 3. The weighted dictionary problem and self-organizing data structures

In the weighted dictionary problem a weight (= access frequency) is associated with every element of the set $S$. The basic goal is to make accesses to high-frequency elements faster than to low-frequency elements without sacrificing the efficiency of the other set operations.

The first problem studied was the construction of an optimal search tree for given access frequencies. More precisely, let $S = \{x_1, \ldots, x_n\}$ and let $w_i$ be the weight of item $x_i$. For a tree $T$ let $d_i$ be the depth of item $i$. Then $P = \sum_i w_i d_i$ is called the *weighted path length* of the tree $T$. It is not too hard to construct an optimum tree, i.e. a tree which minimizes the weighted path length, in time $O(n^3)$ by dynamic programming. This was improved to $O(n^2)$ by Knuth [111] (see also [224]). In the case of leaf-oriented storage an $O(n \log n)$ algorithm was found by Hu and Tucker [99] and Garsia and Wachs [81]. The latter algorithms resemble Huffman's algorithm [102] for the construction of optimal prefix codes. The original correctness proofs for both algorithms were fairly involved. A simple proof was recently found by Kingston [108].

A search tree for a set $S$ yields directly a prefix code for the symbols $x_1, \ldots, x_n$ over the binary alphabet {"*go left*", "*go right*"}. In view of the noiseless coding theorem it is therefore not surprising that the normalized weighted path length $\bar{P} = \sum_i (w_i/W) d_i$, where $W = \sum_i w_i$, is strongly related to the entropy $H = \sum_i (w_i/W) \log(W/w_i)$ of the access frequency distribution. In particular, $\bar{P} \geqslant H - \log H$ for node-oriented storage and $\bar{P} \geqslant H$ for leaf-oriented storage [13]. Also, there are trees achieving $\bar{P} \leqslant H + 2$ for both storage organizations [139]. Nearly optimal trees, i.e., trees which come within a constant factor of optimality and can be constructed in linear time $O(n)$ were described by Fredman [72], Mehlhorn [138], Bayer [13] and Korsch [116]. An implicit data structure achieving the same goal was found by Frederickson, [69].

We will now turn to the dynamic case. It has two facets: the underlying set $S$ may change and the access frequencies of the elements of $S$ may change.

Faller [60] and Gallager [78] proposed a scheme for dynamic Huffman coding. Knuth [114] gives a real-time implementation of this scheme. More precisely, suppose that the frequency $w$ of a leaf at depth $l$ changes by 1. Then an optimal Huffman tree for

the new distribution can be derived from the old Huffman tree in time $O(l)$. Vitter [210] refined this result. His scheme does not only work on-line in real time, it also uses at most one more bit per letter than the standard off-line Huffman algorithm.

Let us consider search trees next. The relevant operations are as follows:

(1) *Access*$(x, S)$: if item $x$ is in set $S$ then return a pointer to its location, otherwise return **nil**;
(2) *Insert*$(x, S)$: insert $x$ into set $S$;
(3) *Delete*$(x, S)$: delete $x$ from set $S$ and return the resulting set;
(4) *Join*$(S_1, S_2)$: return a set representing the items in $S_1$ followed by the items in $S_2$, destroying $S_1$ and $S_2$ (this assumes that all elements of $S_1$ are smaller than all elements of $S_2$);
(5) *Split*$(x, S)$: returns two sets $S_1$ and $S_2$; $S_1$ contains all items of $S$ smaller than $x$ and $S_2$ contains all items of $S$ larger than $x$ (this assumes that $x$ is in $S$); set $S$ is destroyed;
(6) *ChangeWeight*$(x, S, \delta)$: changes the weight of element $x$ by $\delta$.

What time bounds can we hope for? We mentioned above that the normalized weighted path length is related to the entropy of the frequency distribution and therefore the best we can hope for operation *Access* is an $O(\log(W/w_i))$ time bound. We call this bound the *ideal access time* of item $i$. Similarly, inserting an element of weight $w$ can be no cheaper than accessing the new element or one of its neighbors in the new tree and we should therefore content ourself with an

$$O(\log(W+w)/\min(w^+, w, w^{-1}))$$

time bound, where $w^-$ and $w^+$ are the weights of the two neighbors of the new element.

Bayer [13] gave a heuristics for the weighted dynamic dictionary problem, but he gave no theoretical results, and indeed his trees do not have ideal access time in the worst case. Unterauer [207] achieves ideal access time, but does not analyze the worst-case time required for updates. Mehlhorn [140, 141] introduced *D-trees*; D-trees extend BB[$\alpha$]-trees and achieve ideal worst-case *Access*, *Change*, *Insert* and *Delete* time. He did not consider *Join* and *Split*. D-trees have the weight-property of BB[$\alpha$]-trees mentioned in Section 2. Operations *Join* and *Split* were then considered by Güting and Kriegel [87], Kriegel and Vaishnavi [119], Bent, Sleator and Tarjan [18], achieving good worst-case bounds for all five operations.

A very elegant alternative which achieves good amortized time bounds for all operations was found by Sleator and Tarjan [183]. Their splay trees show the following behavior:

- The amortized cost of *Access*$(x, S)$ is $O(\log(W/w))$.
- The amortized cost of *Delete*$(x, S)$ is $O(\log(W/\min(w, w^-)))$, where $w^-$ is the weight of the predecessor of $x$ in $S$.
- The amortized cost of *Join*$(S_1, S_2)$ is $O(\log((w_1+w_2)/w))$, where $w$ is the weight of the largest element in $S_1$ and $w_i$ is the weight of $S_i$.
- The amortized cost of *Insert*$(x, S)$ is $O(\log(W+w/\min(w^-, w^+, w)))$, where $w^-$ is the weight of the predecessor, $w^+$ is the weight of the successor of $x$.
- The amortized cost of *Split*$(x, S)$ is $O(\log(W/w))$.
- The amortized cost of *ChangeWeight*$(x, S, \delta)$ is $O(\log((W+\delta)/w))$.

Splay trees achieve this behavior *without* keeping any explicit information about the weights of the elements; they are a so-called *self-organizing data structure.* In self-organizing data structures an item $x$ is moved closer to the entry point of the data structure whenever it is accessed. This will make subsequent accesses to $x$ cheaper. In this way the elements of $S$ compete for the good places in the data structure and high-frequency elements are more likely to be there. Note however, that we do *not* maintain any explicit frequency counts or weights; rather, we hope that the data structure self-organizes to a good data structure.

Since no global information is kept in a self-organizing data structure, the worst-case behavior of a single operation can always be horrible. However, the average and the amortized behavior may be good. For an average case analysis we need to have probabilities for the various operations. The data structure then leads to a Markov chain whose states are the different incarnations of the data structure. We can then use probability theory to compute the stationary probabilities of the various states and use these probabilities to derive bounds on the expected behavior of the data structure.

For the amortized behavior, we take a combinatorial point of view and analyze the cost of sequences of operations. Experiments [20] suggest that the behavior of the heuristics on real data is more closely described by the amortized analysis than by the probabilistic analysis. We will now describe self-organizing linear search in some detail and then briefly return to search trees.

*Self-organizing linear search* is quite simple. Let $S=\{x_1,\ldots,x_n\}$ and let us assume that $S$ is organized as a linear list. For simplicity, we consider only the operation *Access*$(x)$ and postulate that the cost of *Access*$(x)$ is *pos*$(x)$ where *pos*$(x)$ denotes the position of $x$ in the current list. Two popular strategies for self-organizing linear search are the Move-to-Front and the Transposition Rule.

*Move-to-Front Rule* (*MFR*): Operations *Access*$(x)$ and *Insert*$(x)$ make $x$ the first element of the list and leave the order of the remaining elements unchanged.

*Transposition Rule* (*TR*): Operation *Access*$(x)$ interchanges $x$ with the element preceding $x$ in the list.

For the average-case analysis we postulate a probability distribution $p_1,\ldots,p_n$ and assume that the accesses are independent random variables and that an access to item $x_i$ has probability $p_i$. We may assume $p_1\geqslant p_2\geqslant\cdots\geqslant p_n$ w.l.o.g. Assuming complete knowledge of the distribution it is clearly optimal to arrange the items according to *decreasing frequency* and to never change this arrangement. The expected access cost is $E_{\mathrm{FD}}=\sum_i p_i\cdot i$ in this case. Let us denote the expected access cost of the two rules by $E_{\mathrm{MF}}$ and $E_{\mathrm{T}}$ respectively. The expected behavior of the Move-to-Front Rule was investigated by McCabe [136], Burville and Kingman [34], Knuth [112], Hendricks [93], Rivest [175] and Bitner [22]. In particular,

$$E_{\mathrm{MF}}=\sum_{i=1}^{n} p_i\left(1+\sum_{j\neq i}\frac{p_j}{p_i+p_j}\right)\leqslant 2\cdot E_{\mathrm{FD}}.$$

This can be seen as follows. The expected position of element $x_i$ under the Move-to-Front Rule is 1 plus the expected number of $x_j$'s which are in front of $x_i$. Next observe that $x_j$ is ahead of $x_i$ if there is an integer $k$ such that the last $k$ accesses were an access to $x_j$ followed by $k-1$ accesses to items different from $x_i$ and $x_j$. Hence the

probability that $x_j$ is ahead of $x_i$ is given by

$$p_j \sum_{k \geq 0} (1-(p_i+p_j))^k = p_j/(p_i+p_j)$$

and the expression for $E_{MF}$ follows. Next observe that $p_j/(p_i+p_j) \leq 1$ and hence

$$E_{MF} = 1 + 2 \cdot \sum_{i=1}^{n} p_i \sum_{j=1}^{i-1} \frac{p_j}{p_i+p_j} \leq 1 + 2 \cdot \sum_i p_i(i-1) \leq 2 \cdot E_{FD}.$$

This shows, that the expected access cost under the Move-to-Front Rule is at most twice the optimum. The Transposition Rule is never worse. More precisely, Rivest [175] showed that $E_T \leq E_{MF}$, with strict inequality unless $n=2$ or $p_i = 1/n$ for all $i$. He further conjectured that *Transpose* minimizes the expected access time for any $p$, but Anderson et al. [10] found a counterexample. Chung et al. [40] showed $E_{MF} \leq \frac{1}{2}\pi \cdot E_{FD}$ and Gonnet et al. [85] gave an example where this bound is obtained. Thus the worst-case ratio $E_{MF}/E_{FD}$ is $\frac{1}{2}\pi$. The convergence speed of the two rules was compared by Bitner [22] and it was found that the Move-to-Front Rule converges more quickly. A combination of the Move-to-Front Rule with the more traditional method of keeping frequency counts is discussed in [121].

We will next present an analysis of the amortized behavior of the Move-to-Front Rule due to Bentley and McGeoch [20]. Assume that we perform a sequence of $m$ accesses to this list. Let $h_i$, $1 \leq i \leq n$, be the number of accesses to $x_i$. We may assume w.l.o.g. that $h_1 \geq h_2 \geq \cdots \geq h_n$. If we start with the list $x_1, \ldots, x_n$ and never change it then the total cost of the $m$ accesses is $C_{FD} = \sum_i i \cdot h_i$. Suppose now that we start with the list $x_1, \ldots, x_n$ and use the Move-to-Front Rule. Let $C_{MF}$ be the total cost of $m$ accesses under the Move-to-Front Rule. Let $t_j^i$, $1 \leq j \leq h_i$, be the cost of the $j$th access to item $x_i$. Then

$$C_{MF} - C_{FD} \leq \sum_{i=1}^{n} \sum_{j=1}^{h_i} t_j^i - \sum_{i=1}^{n} i h_i = \sum_{i=1}^{n} \sum_{j=1}^{h_i} (t_j^i - i).$$

Consider an arbitrary term $t_j^i - i$ in this sum. If $t_j^i > i$ then there were at least $t_j^i - i$ accesses to items $x_h$, $h > i$, between the $(j-1)$st access and the $j$th access to $x_i$ (the 0th access is the initial configuration). Thus

$$\sum_{j=1}^{h_i} (t_j^i - i) \leq h_{i+1} + \cdots + h_n$$

and hence

$$C_{MF} - C_{FD} \leq \sum_{i=1}^{n} (h_{i+1} + \cdots + h_n) = \sum_{i=1}^{n} (i-1) h_i = C_{FD} - m.$$

In particular, $C_{MF} \leq 2 \cdot C_{FD}$. A much stronger result was shown by Sleator and Tarjan [184]. They proved that no rule whatsoever can beat the Move-to-Front Rule by more than a factor of 2. This is even true for off-line rules which know the complete sequence of accesses in advance.

Allen and Munro [8] were the first to consider self-organizing binary search. They

show that the expected search time under the Move-to-Root heuristics, where an accessed item is moved to the root by a sequence of rotations, is only a constant factor above the expected search time in an optimum search tree. They also showed that the transposition heuristics does not have this property. Bitner [22] investigates the expected behavior of several other heuristics.

Splay trees [183] refine the Move-to-Root heuristics by combining it with the concept of path compression, cf. Section 5 on the *Union-Split-Find* problem. The access time in splay trees is within a constant factor of the access time of static optimum search trees; this is the analogue of the relationship $C_{MF} \leqslant 2 \cdot C_{FD}$ derived above. It is open whether this result can be extended as it was in the case of the Move-to-Front Rule for linear search. A first step was taken by Tarjan [197] who showed that accessing the nodes of a splay tree in sequential order takes time $O(n)$.

## 4. Persistence

Ordinary data structures are *ephemeral* in the sense that a change to the structure destroys the old version, leaving only the new version available for use. In contrast, a *persistent* structure allows access to any version, old or new, at any time. We distinguish *partial* and *full* persistence. A data structure is partially persistent if all versions can be accessed but only the newest version can be modified, and fully persistent if every version can be both accessed and modified.

A number of researchers have developed partially or fully persistent forms of various data structures, including stacks [157], queues [96], search trees [158, 163, 174, 190, 177] and related structures [38, 41, 49, 53]. In the articles of Dobkin and Munro [49], Overmars [163, 165], Chazelle [38], the problem of persistence is called *searching in the past.*

Dobkin and Munro [49] consider the following problem. Let $S$ be a universe of $n$ objects subject over time to $m$ deletions and insertions in arbitrary order. Assuming that there exists a total order among the objects, they describe a method for computing in time $O(\log n \log m)$, the *rank* of any object at any given time, i.e., the number of objects that precede it at that time. The space used is $O(n+m \log n)$. Overmars [164] improves the query time to $O(\log(n+m))$. Chazelle [38] presents a data structure that requires $O(n+m)$ storage and allows the computation of any neighbor of a new object at time $\theta$, in $O(\log n \log m)$ time. The method also allows the set $S$ to be only partially ordered.

Overmars [165] studies three simple but general ways to obtain partial persistence. One method is to explicitly store every version, copying the entire ephemeral structure after each update operation. This costs $\Omega(n)$ time and space per update. An alternative method is to store no versions but instead to store the entire sequence of update operations, rebuilding the current version from scratch each time an access is performed. A hybrid method is to store the entire sequence of update operations and in addition every $k$th version for some suitably chosen value of $k$.

Cole [41] devises a partial persistent representation of sorted sets that occupies $O(m)$ space and has $O(\log m)$ access time, where $m$ is the total number of updates (insertions

and deletions) starting from an empty set. His solution is off-line, i.e., the entire sequence of updates must be known in advance. Sarnak and Tarjan [177] propose a simple data structure that overcomes this drawback. They present a persistent form of binary search tree with an $O(\log m)$ worst-case access/insert/delete time and an amortized space requirement of $O(1)$ per update.

Swart [190], using the idea of *path copying*, presents a fully persistent representation of sorted sets and lists with an $O(\log m)$ time bound per operation and an $O(\log m)$ space bound per update.

Driscoll et al. [53] finally propose a general solution for the persistence problem. They develop simple, systematic and efficient techniques for making different linked data structures persistent. They show first that if an ephemeral structure has nodes of bounded in-degree, then the structure can be made partially persistent at an amortized space cost of $O(1)$ per update step and a constant-factor increase in the amortized cost of access and update operations. Second, they present a method which can make a linked structure of bounded in-degree fully persistent at an amortized time and space cost of $O(1)$ per update and a worst-case time of $O(1)$ per access step. At last they present a partial persistent implementation of balanced search tree with a worst-case time per operation of $O(\log n)$ and an amortized space cost of $O(1)$ per insertion or deletion. Combining this result with a *delayed updating* technique of Tsakalidis [204] they obtain a fully persistent form of balanced search trees with the same time and space bounds as in the partially persistent case. Using another technique they can make the $O(1)$ space bound for insertion and deletion worst-case instead of amortized. The technique employed by Driscoll et al. is strongly related to fractional cascading, cf. Subsection 2.1.2. This relationship can be used to support a forget operation which permits to explicitly delete versions and thus improves the space requirement [148].

## 5. The *Union-Split-Find* problem

We consider the problem of maintaining a collection of disjoint sets under the operations of *Union* and *Split*. More precisely, the problem is to carry out three kinds of operations on a partition of the universe $U = \{1, \ldots, n\}$: *Find*, which determines the set containing a given element; *Union*, which combines two sets into one; and *Split*, which splits a set at a given element into two. In order to identify the sets, we assume that the algorithms maintain with each set a unique name. The precise formulation of the three operations is as follows:

(1) *Find*$(x)$: return the name of the set containing $x$;

(2) *Union*$(A, B)$: combine the two sets with names $A$ and $B$, destroying the two old sets, and return a name for the new set;

(3) *Split*$(A, x)$: split the set with name $A$ into the sets $A_1 = \{y \in A : y \leqslant x\}$ and $A_2 = A - A_1$, destroying the old set, and return names for the two sets formed.

### 5.1. *The Union-Find problem*

Here, we start with the partition of $U$ into singleton sets and allow only the operations *Find* and *Union*. A popular solution, proposed by Galler and Fischer [79],

represents each set by a rooted tree. The nodes of the tree are the elements of the set and the name of a set is the root of the tree. With each element $x$ we store a pointer $p(x)$ to its parent; the parent pointer of a root is **nil**. Initially, all parent pointers are **nil**. To carry out *Find*$(x)$, we follow parent pointers starting in $x$ until we reach a root and return the root. To carry out *Union*$(A, B)$, we define $p(A)$ to be $B$ and return $B$.

The naive algorithm is not very efficient, requiring $O(n)$ time per find in the worst case. However, there are two simple heuristics which improve the efficiency dramatically: the *weighted union* [79] and the *path compression* heuristics, the latter of which was suggested by McIlroy and Morris. We store with each set $A$ its cardinality *size*$(A)$ and carry out *Union*$(A, B)$ by defining $p(A)$ to be $B$ if *size*$(A) \leqslant$ *size*$(B)$, and $p(B)$ to be $A$ otherwise. In the algorithm for *Find*$(x)$ we not only traverse the path from $x$ to the root but also redirect all parent pointers of the nodes traversed to the root.

Suppose now that we carry out $m \geqslant n$ *Finds* and $n-1$ *Unions*. Let $t(m, n)$ be the maximum time required by any such sequence of instructions. Fischer [61] showed that $t(m, n) = O(n + m \cdot \log \log m)$, Hopcroft and Ullman [97] improved the bound to $t(m, n) = O(n + m \cdot \log^* n)$ and finally Tarjan [192] and Banachowski [12] improved the bound to $t(m, n) = O(n + m \cdot \alpha(m, n))$, where $\alpha$ is the inverse of Ackermann's function, i.e.,

$$\alpha(m, n) = \min\{z \geqslant 1 : A(z, 4\lceil m/n \rceil) > \log n\}$$

and

$$\begin{aligned} &A(i, 0) = 1 && \text{for all } i \geqslant 1, \\ &A(0, x) = 2x && \text{for all } x \geqslant 0, \\ &A(i+1,\ x+1) = A(i, A(i+1, x)) && \text{for all } i, x \geqslant 0. \end{aligned}$$

Tarjan and van Leeuwen [199] show that the same time bound applies to several variants of the *Union-Find* algorithm described above.

How good is this algorithm? Tarjan [193] has shown that any pointer-machine algorithm which obeys the *separation assumption* requires time $\Omega(m \cdot \alpha(m, n))$ to solve the *Union-Find* problem. The separation assumption states that sets correspond to connected components in the pointer structure.

All time bounds quoted above are amortized. The worst-case behavior was considered by N. Blum [24] and he proves a $\Theta(\log n / \log \log n)$ bound; again, the lower bound used the separation assumption.

The average-case behavior was considered by Doyle and Rivest [51], Yao [220], Knuth and Schönhage [115], Yao [221] and Bollobás and Simon [26]. They show that under various probability assumptions the expected running time of the *Union-Find* algorithm is linear, even if only one of the two heuristics is used.

Manilla and Ukkonen [133] consider a variant of the general *Union-Find* problem where backtracking over the *last* union operations is possible by means of a *Deunion* operation. They present two methods which allow to process a sequence of $m$ *Finds*, $k$ *Unions* and $k$ *Deunions* in time $O((m+k) \cdot \log n / \log \log n)$ and $O(k + m \log n)$ respectively [212]. A further generalization where weights are associated with the unions is considered by Gambosi et al. [80].

All of the algorithms mentioned above run on pointer machines. A linear-time RAM algorithm for the special case where the unions are known in advance was recently described by Gabow and Tarjan [77].

### 5.2. *The interval Split-Find problem*

Here, we start with the partiton of $U$ into a single block and allow only the operations *Find* and *Split*. Clearly, all blocks arising during the execution are intervals. Hopcroft and Ullmann [97] gave a solution which processes $n$ *Splits* in total time $O(n \log^* n)$ and each *Find* in worst-case time $O(\log^* n)$. Gabow [76] gave an improved algorithm which processes the $n$ *Splits* in total time $O(n \cdot \alpha(n, n))$ and each *Find* in amortized time $O(\alpha(n, m))$. All of this runs on a pointer machine.

Again, one can do better on a RAM. Gabow and Tarjan [77] gave a linear-time solution, which was extended by Imai and Asano [104], to the incremental set-splitting problem. In this version the underlying universe can be expanded by adding (operation *Add*) a new element next to a given element. Of course, the *Access* operation, which is implicit in a *Find*, becomes nontrivial now and is not accounted for in the linear time bound. However, in the applications discussed by Imai and Asano this causes no problems.

### 5.3. *The interval Union-Split-Find problem*

We finally turn to the full problem. Here, we start with the partition of $U$ into singletons and allow the operations *Find*, *Split* and *Union*. We postulate however, that the partition is always a partition into intervals, i.e., a union operation can only join adjacent intervals. It is customary, to use the right endpoint of an interval as the name of the interval in this problem.

Van Emde Boas, Kaas and Zijlstra [58] describe a pointer-machine solution which executes all three operations in worst-case time $O(\log \log n)$. The corresponding lower bound was shown by Mehlhorn, Näher and Alt [147]. The lower bound does not use the separation assumption and is even valid for the amortized complexity of the problem. Mehlhorn, Näher and Alt [147] also show that the lower bound increases to $\Omega(\log n)$ with the separation assumption.

Finally, Mehlhorn and Näher [146] show that the additional operations *Add* and *Erase*, which modify the underlying universe by adding an element next to a given element or removing a given element, can also be supported in amortized time $O(\log \log n)$.

## 6. Priority queues

A *priority queue* is an abstract data structure consisting of a set of *items*, each with a real-valued *key*, subject to the following operations:

(1) *Makequeue*: return a new, empty priority queue;
(2) *Insert*$(i, h)$: insert a new item $i$ with predefined key into priority queue $h$;
(3) *Findmin*$(h)$: return an item of minimum key in priority queue $h$; this operation does not change $h$;
(4) *Deletemin*$(h)$: delete an item of minimum key from $h$ and return it.

In addition, the following operations on priority queues are often useful:

(5) *Meld*$(h_1, h_2)$: return the priority queue formed by taking the union of the item-disjoint priority queues $h_1$ and $h_2$; this operation destroys $h_1$ ad $h_2$;
(6) *Decreasekey*$(\Delta, i, h)$: decrease the key of item $i$ in priority queue $h$ by subtracting the nonnegative real number $\Delta$; this operation assumes that the position of $i$ in $h$ is known;
(7) *Delete*$(i, h)$: delete arbitrary item $i$ from priority queue $h$. This operation assumes that the position of $i$ in $h$ is known.

In our discussion of priority queues we shall assume that a given item is in only one priority queue at a time and that a pointer to its priority queue position is maintained. It is important to remember that priority queues do *not* support efficient searching for an item.

Priority queues have many applications, most noteworthy sorting and network optimization problems. A set of $n$ items may be sorted by one *Makequeue*, $n$ *Insert*, $n$ *Findmin* and $n$ *Deletemin* operations. The single-source shortest path problem (cf. Chapter 10) on a graph with $n$ nodes and $m$ edges of nonnegative weight can be solved by one *Makequeue*, $n$ *Insert*, *Findmin* and *Deletemin* and $m$ *Decreasekey* operations. We infer from these applications that at least one of the operations *Insert*, *Findmin* and *Deletemin* must have cost $\Omega(\log n)$ because of the $\Omega(n \log n)$ lower bound for sorting, and that the operations can occur with drastically different frequencies in some applications.

Some solutions, to be described in the sequel, do not support the *Meld* operation. We will refer to a priority queue without this operation as a simple priority queue. Also note, that a *Decreasekey* operation can always be simulated by a *Delete* followed by an *Insert*.

For small universes, say the keys are drawn from the range $[1 \ldots N]$, any solution for the *Union-Split-Find* problem (cf. Section 5) gives a simple priority queue: the set of items in the queue corresponds to the marked items of that section, *Find*(1) is *Findmin*, *Union*$(i)$ is *Delete*$(i)$, *Union*(*Find*(1)) is *Deletemin*, and *Split*$(i)$ is *Insert*$(i)$. In particular, the simple queue operations can all be realized in time $O(\log \log N)$ per operation. Orlin and Ahuja [161] give a solution with an $O(1)$ bound for *Makequeue*, *Delete*, *Findmin*, *Decreasekey* and an $O(\log N)$ bound for *Insert* and *Deletemin*. Ahuja et al. [7] improve this to $O(1)$ for *Delete*, *Findmin* and *Decreasekey*, $O(n)$ for *Makequeue* and $O(\sqrt{\log N})$ for *Insert* and *Deletemin*; here $n$ is the number of *Insert* operations. This gives rise to an $O(m + n\sqrt{\log N})$ shortest path algorithm.

Most solutions for arbitrary universes are based on the concept of a *heap-ordered* tree. A heap-ordered tree is a rooted tree containing a set of items, one item in each node, with the items arranged in *heap order*: if $v$ is any node, then the key of the item in $v$ is no less than the key of the item in its parent $p(v)$, provided $v$ has a parent. Thus the tree root contains an item of minimum key.

A first realization of simple priority queues was given by Williams [218]. He uses complete binary heap-ordered trees and shows that the simple queue operations all take time $O(\log n)$ and that complete binary trees can be stored implicitly in an array without the use of pointers. Floyd [63] shows that a *Makequeue* followed by $n$ inserts can be made to run in time $O(n)$ instead of $O(n \log n)$.

Doberkat [47] gives an average-case analysis of Floyd's algorithm. Porter and Simon [171] analyzed the average cost of inserting a random element into a random heap in terms of exchanges. They proved that this average is bounded by the constant 1.61. Their proof does not generalize to sequences of insertions since random insertions into random heaps do not create random heaps. The repeated insertion problem was solved by Bollobás and Simon [27]; they show that the expected number of exchanges is bounded by 1.7645. The worst-case cost of *Inserts* and *Deletemins* was studied by Gonnet and Munro [84]; they give $\log\log n + O(1)$ and $\log n + \log^* n + O(1)$ bounds for the number of comparisons respectively.

In the 70s and 80s many different realizations of the full repertoire of priority queue operations were found: heap-ordered (2–3)-trees [2], leftist trees [42], binomial queues [211, 30], self-adjusting heaps [185] and pairing heaps [74]. They achieve $O(\log n)$ bounds for all operations.

Johnson [107] introduces $a$-ary heap-ordered trees, where $a$ is a parameter, and shows how to implement *Makequeue* and *Findmin* in time O(1), *Decreasekey* in time $O(\log n/\log a)$ and the other operations in time $O(a \log n/\log a)$. With $a = \max(2, \log(m/n))$ this gives rise to an

$$O(m \log n/\max(2, \log(m/n)))$$

shortest path algorithm.

Fredman and Tarjan [75] improve Johnson's result and obtain O(1) amortized time bounds for *Makequeue*, *Findmin*, *Insert*, *Meld* and *Decreasekey* and $O(\log n)$ amortized time bounds for *Delete* and *Deletemin*. They call their data structure Fibonacci-heaps. F-heaps give rise to an $O(n \log n + m)$ shortest path algorithm. An alternative to F-heaps are the relaxed heaps of Driscoll et al. [52]. In contrast to F-heaps the time bound for *Delete* and *Deletemin* is worst-case instead of amortized.

A generalization of priority queues was considered by Atkinson et al. [11]. They add the operations *Findmax* and *Deletemax* and show that the $O(\log n)$ time bound can be maintained.

## 7. Nearest common ancestors

In this section we consider the following problem: given a dynamically changing collection of rooted trees, answer queries of the form: "What is the nearest common ancestor of two given nodes $x$ and $y$, denoted by $nca(x, y)$?" Of course, this problem comes in many different flavors according to which update operations are permitted. We use $n$ to denote the total number of nodes, $m$ to denote the number of dynamic operations and $k$ to denote the number of queries.

In the *off-line* version of the nearest common ancestor problem, the collection of trees is static and the sequence of queries is known in advance. Aho, Hopcroft and Ullman [3] describe an $O(n + k \cdot \alpha(k + n, n))$ pm-algorithm using $O(n)$ space. Here $\alpha$ is the functional inverse of Ackerman's function (cf. Section 5). Gabow and Tarjan [77] give an $O(n + k)$ time RAM algorithm using $O(n)$ space.

In the *static, on-line* version the collection of trees is static but the queries are given

*on-line*, i.e., each query must be answered before the next one is given. Aho, Hopcroft and Ullman [3] propose a RAM algorithm requiring $O(n \log\log n)$ preprocessing time, $O(n \log\log n)$ space and $O(\log\log n)$ time per query. Harel and Tarjan [92] present a RAM algorithm running in $O(n)$ preprocessing time, $O(n)$ space and answering a query in $O(1)$ time; a simpler solution with the same performance is given by Schieber and Vishkin [178]. Van Leeuwen [123] gives a pm-algorithm requiring $O(n \log\log n)$ preprocessing time, $O(n \log\log n)$ space and $O(\log\log n)$ *optimal* query time. Harel and Tarjan [92] prove the optimality of this query time and claim that van Leeuwen's algorithm can be modified to run on a pointer machine in linear time and space. Another optimal pm-algorithm with $O(n)$ preprocessing time, $O(n)$ space and $O(\log\log n)$ query time is described in [125].

We now come to the dynamic versions of the problem. In the *linking roots version*, the queries are given on-line and two trees can be joined by linking their roots. Van Leeuwen [123] gives an $O(n+m \log\log n)$ time pm-algorithm answering each query in $O(\log\log n)$ time. Harel and Tarjan [92] present an $O(n+m \cdot \alpha(m+n, n))$ time RAM algorithm answering each query in $O(\alpha(m+n, n))$ amortized time.

In the *linking and cutting version* the queries are on-line and the collection of trees can be modified by two operations. *Link*$(x, y)$, where $y$ is a root, makes $y$ a child of the vertex $x$ (not necessarily a root) and *Cut*$(x)$ deletes the edge connecting $x$ to its parent and thus makes $x$ an additional root. Aho, Hopcroft and Ullman [3] consider only linkings. Their algorithm runs on a RAM in time $O((m+n)\log n)$ and space $O(n \log n)$ and the time required for a query is $O(\log n)$. Maier [130] considers linkings and cuttings, achieves the same time bound and improves upon the space bound. His solution requires less than $O(n \log n)$ but still more than linear space; it also runs on a RAM. Sleator and Tarjan [182] give the fastest solution which in addition runs on a pointer machine. At first they propose a solution with total running time $O(n+m \log n)$ for $m$ *Link* and *Cut* operations between $n$ nodes using space $O(n)$; the nearest common ancestor can also be determined in time $O(\log n)$. By using a more complicated data structure they can improve the above result so that each individual *Link* and *Cut* operation takes time $O(\log n)$.

In the *dynamic tree version* the queries are on-line, there is only one tree, and this tree can be updated by the insertion of leaves or deletion of nodes. Tsakalidis [206] presents a pm-algorithm which needs $O(n)$ space, performs $m$ arbitrary insertions and deletions on an initially empty tree in time $O(m)$ and allows to determine the nearest common ancestor of nodes $x$ and $y$ in time

$$O(\log(\min\{depth(x), depth(y)\}) + \alpha(k, k)),$$

where the second term is amortized over the $k$ queries and $depth(x)$ is the distance from node $x$ to the root.

## 8. Selection

The selection problem can be stated as follows: we are given a sequence $x_1, \ldots, x_n$ of pairwise distinct elements and an integer $k$, $1 \leqslant k \leqslant n$, and want to find the $k$th smallest

element of the sequence, i.e. an $x_j$ such that there are $k-1$ keys $x_l$ with $x_l < x_j$ and $n-k$ keys $x_l$ with $x_l > x_j$. For $k = n/2$ such a key is called *median*. Of course, selection can be reduced to sorting. We might first sort sequence $x_1, \ldots, x_n$ and then find the $k$th smallest element by a single scan of the sorted sequence. This results in an $O(n \log n)$ algorithm. A linear expected time algorithm was found by Hoare [94]. One simply splits the set $S$ into subsets $S_1 = \{x \in S: x \leqslant x_1\}$ and $S_2 = \{x \in s: x > x_1\}$ and then applies the algorithm recursively to the appropriate subset. If $T(i, n)$ denotes the expected time to determine the $i$th largest element in a collection of $n$ elements then

$$T(i,n) = n + \frac{1}{n}\left(\sum_{k=1}^{i-1} T(i-k, n-k) + \sum_{k=i}^{n} T(i,k)\right)$$

since $x_1$ is the $k$th largest element with probability $1/n$. Then $T(i, n) = O(n)$ as a simple induction shows.

A worst-case running time $O(n)$ can be obtained by choosing the splitting element more carefully [23]. The set $S$ is divided into subsets of five elements each and the median of each subset is computed. The algorithm is then used recursively to determine the median of the $n/5$ medians. The median of the medians is used as the splitting element. This choice guarantees that $\max(|S_1|, |S_2|) \leqslant 8n/11$ and therefore the worst-case running time is governed by the recurrence

$$T(n) \leqslant O(n) + T(n/5) + T(8n/11)$$

which has an $O(n)$ solution.

Knuth [112, Section 5.3.3], traces the history of the selection problem to the writings of Dodgson [50], and provides an excellent overview of the early developments of this problem. The original formulation of the problem by Dodgson [50] and Steinhaus [188] was in terms of lawn tennis tournaments. Let $V_k(n)$ ($\bar{V}_k(n)$) be the number (average number) of pairwise comparisons needed to find the $k$th smallest of $n$ numbers ($k \geqslant 2$) (assuming that all $n!$ orderings are equally likely). Of particular interest is the asymptotic behavior of $V_k(n)$ ($\bar{V}_k(n)$) when $n$ goes to infinity either with $k$ fixed or with $k = \alpha \cdot n$ for a fixed constant $\alpha$ in the range $0 \leqslant \alpha \leqslant \frac{1}{2}$.

Hadian and Sobel [91] described an algorithm, called *replacement selection* by Knuth, that proves the general bound

$$V_k(n) \leqslant n - k + (k-1)\lceil \log(n-k+2) \rceil.$$

For large $k$ (in particular for $k = \alpha \cdot n$), this was improved by M. Blum et al. [23] and further improved by Schönhage et al. [180] to $V_k(n) \leqslant 3n + o(n)$. The first general lower bound, $V_k(n) \geqslant n + k - 2$, was proved by M. Blum et al. [23]. For a detailed survey of all the lower bounds see [109]. Bent and John [17] improved Kirkpatrick's lower bound in [109]. Especially, they show

$$V_{\alpha n}(n) \geqslant (1 + H_\alpha)n + O(\sqrt{n}),$$

where $H_\alpha = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$ is the discrete entropy of $\alpha$. In particular, the bound for the *median* is

$$V_m(n) \geqslant 2n + O(\sqrt{n}), \quad \text{where } m = \lceil \tfrac{1}{2} n \rceil.$$

A considerable amount of work has also been done on estimating $\bar{V}_k(n)$. Matula [135] proved that, for some absolute constant $c$, $\bar{V}_k(n) \leqslant n + ck \ln \ln n$ as $n \to \infty$. A.C. Yao and F.F. Yao [223] showed that there exists an absolute constant $c' > 0$, such that $\bar{V}_k(n) \geqslant n + c'k \ln \ln n$ as $n \to \infty$, proving a conjecture of Matula. An upper bound of $\bar{V}_k(n) \leqslant n + k + o(n)$ for all $k$ and $n$ is due to Floyd and Rivest [64]. Cunto and Munro [43] prove that $\bar{V}_k(n) \geqslant n + k - O(1)$ in general and that for the *max-min-median* problem

$$2n - o(n) \leqslant \bar{V}_t(n) \leqslant \tfrac{9}{4}n + o(n), \quad \text{where } t \in \{1, n, \lceil \tfrac{1}{2}n \rceil\}.$$

Dobkin and Munro [48] show that a linear-time algorithm is possible for the median problem which requires an additional work space of only $\lceil \frac{1}{2}n \rceil + 1$ data cells. Ramanan and Hyafil [172] present some algorithms which solve efficiently the problem of selecting the $k$th smallest elements, and give their respective order of a totally ordered set of $n$ elements, when $k$ is small compared to $n$.

Munro and Paterson [154] consider time–space trade-offs for selection in a *tape input model*, i.e., inputs are stored on a read-only input tape and may be accessed only in a sequential scan of the entire tape. Output is to a write-only tape and workspace registers may hold input values. Let $T$ be the time, $S$ the workspace and $n$ the size of the set; then when time is measured in comparisons, their algorithm realizes a trade-off of $T \log S = O(n \log n)$ over the range $\Omega(\log^2 n) = S = O(2^{\sqrt{\log n}})$. Frederickson [70] modifies and extends their approach to realize the same trade-off over the broader range

$$\Omega(\log^2 n) = S = O(2^{(\log n / \log^* n)}).$$

## 9. Merging

The *two-way merging* problem consists of combining two sorted lists $A$ and $B$ to make one larger sorted list. The algorithms developed for this problem can be classified according to the following properties:

(i) Minimizing the number of comparisons and assignments.

(ii) Minimizing the workspace needed in addition to the space used to store the two files. An algorithm using O(1) additional workspace is said to be *in-place*.

(iii) Maintaining stability. A stable merging algorithm is one which preserves the relative orderings of equal elements from each of the sequences.

(iv) Maintaining searchability. An in-place merging algorithm is said to support searchability if at any stage in the process a search for an arbitrary element can be performed with a small number of comparisons.

If the lists $A$ and $B$ have $m$ and $n$ elements respectively, then there are $\binom{m+n}{n}$ possible placements of the elements of $B$ in the combined list; it follows that $\lceil \log \binom{m+n}{n} \rceil$ comparisons are necessary to distinguish these possible orderings. If we take $m \leqslant n$ then

$$\left\lceil \log \binom{m+n}{n} \right\rceil = \Theta\left( m \log \frac{n}{m} \right).$$

The "binary merging" algorithm of Hwang and Lin [103] (see also [112, Section 5.3.2])

requires fewer than $\lceil \log(\binom{m+n}{n}) \rceil + \min(m, n)$ comparisons to combine sets of size $m$ and $n$ and uses $O(\log n)$ additional workspace. Kronrad [118] showed that merging may be performed in place in linear time. His algorithm does not have the stability property.

Brown and Tarjan [32] note that there is a problem in implementing the *binary merging* algorithm so that it runs in time proportional to the number of comparisons it uses. They present a fast-merging algorithm, based on AVL-trees (Adel'son-Vel'skii and Landis [1]), which runs in optimal time $O(m \log(n/m))$ and uses additional workspace $O(m+n)$. This was extended by Huddleston and Mehlhorn [101] to other set operations than merging.

Trabb Pardo [202] gives a *stable merging* algorithm performing $O(m+n)$ assignments and comparisons in $O(1)$ workspace It yields a stable in-place sorting algorithm running in optimal time. The stable merging algorithm of Dudziński and Dydek [54] performs $O(m \log(n/m))$ comparisons and $O((m+n)\log m)$ assignments in $O(\log m)$ workspace. Carlsson [35] presented a fast stable merging algorithm, called "split-merge", which performs $O(m \log(n/m))$ comparisons and $O(m+n)$ assignments in $O(m \log n)$ space.

Let $M(m, n)$ be the minimum number of pairwise comparisons which will always suffice to merge two ordered lists of lengths $m$ and $n$. Stockmeyer and F.F. Yao [189] prove that $M(m, m+d) = 2m+d-1$ whenever $m \geqslant 2d-2$. (This shows that the standard linear merging algorithm is optimal whenever $m \leqslant n \leqslant \lfloor \frac{3}{2}m \rfloor + 1$.) Thanh et al. [201] present optimal expected-time algorithms for $(2, n)$ and $(3, n)$ merge problems.

Searchability is introduced by Munro and Poblete [155], where they show that a pair of sorted arrays can be merged in-place in linear time so that a logarithmic-time search may be performed at any point during the process.

## 10. Dynamization techniques

Data structures for static sets are always easier to discover than dynamic data structures which also support updates. We now discuss some general methods for turning static data structures into dynamic data structures.

Suppose that we know a *static* data structure, which can be constructed in time $P_S(n)$, requires space $S_S(n)$ and supports queries in time $Q_S(n)$ for a set of $n$ items. We assume throughout that $Q_S(n)$, $P_S(n)/n$ and $S_S(n)/n$ are nondecreasing. We also assume that the queries are *decomposable* in the following sense [19]: Consider a set $A$ and any partitition of $A$ into disjoint sets $B$ and $C$. It is then assumed that the answer to a query about $A$ can be obtained from the answers to the queries about $B$ and $C$ in time $O(1)$. A membership query is decomposable in this sense.

We now show how to support insertions. A brute-force solution reconstructs the static data structure from scratch after every insertion. A more efficient strategy is to partition a set $S$ of $n$ items into blocks $S_i$, to answer queries by combining the answers to the queries about the blocks and to insert an item by forming a new block consisting of a single item. In order to avoid the proliferation of blocks, several blocks are merged into a single block from time to time. The details are as follows. Let $n = \sum_{i \geqslant 0} a_i 2^i$, $a_i \in \{0, 1\}$, be the binary representation of $n$. Let $S_0, S_1, \ldots$ be any partition of $S$ with

$|S_i| = a_i 2^i$, $0 \leqslant i \leqslant \log n$. Then the dynamic structure $D$ is just a collection of static data structures, one for each nonempty $S_i$. The space requirement of $D$ is easily computed as

$$S_D(n) = \sum_i S_S(a_i 2^i) = \sum_i (S_S(a_i 2^i)/a_i 2^i) a_i 2^i$$

$$\leqslant \sum_i (S_S(n)/n) a_i 2^i = S_S(n).$$

The inequality follows from our basic assumption that $S_S(n)/n$ is nondecreasing.

Next note that a query about $S$ can be answered by combining the answers about the $S_i$'s and that there are never more than $\log n$ nonempty $S_i$'s. Hence a query can be answered in time

$$\log n + \sum_i Q_S(a_i 2^i) \leqslant \log n(1 + Q_S(n)) = O(\log n Q_S(n)).$$

Finally consider operation *Insert*$(x, S)$. Let $n+1 = \sum \beta_i 2^i$ and let $j$ be such that $\alpha_j = 0$, $\alpha_{j-1} = \alpha_{j-2} = \cdots = \alpha_0 = 1$. Then $\beta_j = 1$, $\beta_{j-1} = \cdots = \beta_0 = 0$. We process the $(n+1)$st insertion by taking the new point $x$ and the $2^j - 1 = \sum_{i=0}^{j-1} 2^i$ points stored in structures $S_0, S_1, \ldots, S_{j-1}$ and constructing a new static data structure for $\{x\} \cup S_0 \cup S_1 \cup \cdots \cup S_{j-1}$. Thus the cost of the $(n+1)$st insertion is $P_S(2^j)$. Next note that a cost of $P_S(2^j)$ has to be paid after insertions $2^j(2l+1)$, $l = 0, 1, 2, \ldots$, and hence at most $n/2^j$ times during the first $n$ insertions. Thus the total cost of the first $n$ insertions is bounded by

$$\sum_{j=0}^{\lfloor \log n \rfloor} P_S(2^j) n/2^j \leqslant n \sum_{j=0}^{\lfloor \log n \rfloor} P_S(n)/n \leqslant P_S(n)(\lfloor \log n \rfloor + 1).$$

We may summarize the argument above as $Q_D(n) = O(Q_S(n) \log n)$, $S_D(n) = O(S_S(n))$ and $\bar{I}_D(n) = O((P_S(n)/n) \log n)$ where $Q_D(n)$ is the worst-case query time, $S_D(n)$ is the worst-case space requirement and $\bar{I}_D(n)$ is the amortized insertion time for a set of $n$ elements [21].

A static data structure for the membership problem is a sorted array. Its performance is $S_S(n) = n$, $Q_S(n) = \log n$ and $P_S(n) = n \log n$. The construction above yields a dynamic data structure with $S_D(n) = O(n)$ and $\bar{I}_D(n) = Q_D(n) = O((\log n)^2)$.

Overmars and van Leeuwen [167] turned the amortized insertion time into a worst-case bound by spreading work over time. Of course, there is nothing unique about the use of the binary representation in the construction above. Using $b$-ary notation for some $b > 1$, i.e. expressing $n$ as $\sum a_i b^i$, $0 \leqslant a_i < b$, and using $a_i$ sets of size $b^i$ in the partition of $S$, a query time $Q_D(n) = O(b \cdot Q_S(n) \log n / \log b)$ and an amortized insertion time of $\bar{I}_D(n) = O((P_S(n)/n) \log n / \log b)$ results [21]. Mehlhorn and Overmars [149] describe a family of dynamization schemes which allow to trade query time for insertion time and vice versa. The corresponding lower bounds were first shown in [21] and later generalized in [144].

Deletions are somewhat harder to cope with. We need the additional assumption that the static data structure supports deletions (but not insertions) without increasing query time and storage requirement. With this additional assumption the results

mentioned above can be extended to also include deletions [124, 166, 167]. Extensions and variations of the basic dynamization scheme are discussed in [164, 56, 173].

## References


[1] ADEL'SON-VEL'SKII, G.M. and E.M. LANDIS, An algorithm for the organisation of information, *Dokl. Akad. Nauk SSSR* **146** (1962) 263–266 (in Russian); English translation in *Soviet. Math.* **3** (1962) 1259–1262.

[2] AHO, A.V., J.E. HOPCROFT and J.D. ULLMANN, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[3] AHO, A.V., J.E. HOPCROFT and J.D. ULLMAN, On finding lowest common ancestors in trees, *SIAM J. Comput.* **1** (1) (1976) 115–132.

[4] AHO, A.V., J.E. HOPCROFT and J.D. ULLMAN, *Data Structures and Algorithms* (Addison-Wesley, Reading, MA, 1983).

[5] AHO, A.V. and D. LEE, Storing a dynamic sparse table, in: *Proc 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 55–60.

[6] AHO, A.V. and J.D. ULLMAN, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).

[7] AHUJA, R.K., K. MEHLHORN, J.B. ORLIN and R.E. TARJAN, Faster algorithms for the shortest path problem, Tech. Report A04/88, Univ. of Saarland, Saarbrücken, FRG.

[8] ALLEN, B. and I. MUNRO, Self-organizing search trees, *J. ACM* **25**(4) (1978) 526–535.

[9] ALT H. and K. MEHLHORN, Searching semi-sorted tables, *SIAM.J. Comput.* **14**(4) (1985) 840–848.

[10] ANDERSON, E.J., P. NASH and R.R. WEBER, A counter example to a conjecture on optimal list ordering, *J. Appl. Probab.* **19**(3) (1982) 730–732.

[11] ATKINSON, M.D., J.R. SACK, N. SANTORO and T. STROTHOTTE, Min-max heaps and generalized priority queues, *Comm. ACM* **19**(10) (1986) 996–1000.

[12] BANACHOWSKI, L., A complement to Tarjan's result about the lower bound on the complexity of the set union problem, *Inform. Process. Lett.* **11**(2) (1980) 59–65.

[13] BAYER, P.J., Improved bounds on the cost of optimal and balanced binary search trees, Tech. Report, Dept. of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1975.

[14] BAYER, R., Symmetric binary *B*-trees: data structure and maintenance algorithms, *Acta Inform.* **1** (1972) 290–306.

[15] BAYER, R. and E.M. MCCREIGHT, Organisation and maintenance of large ordered indices, *Acta Inform.* **1** (1972) 173–189.

[16] BEN-AMRAM, A.M. and Z. GALIL, On pointers versus adresses, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 532–538.

[17] BENT, S.W. and J. JOHN, Finding the median requires $2n$ comparisons, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing* (1985) 213–216.

[18] BENT, S.W., D.D. SLEATOR and R.E. TARJAN, Biased search trees, *SIAM J. Comput.* **14**(3) (1985) 545–568.

[19] BENTLEY, J., Decomposable searching problems, *Inform. Process. Lett.* **8**(5) (1979) 244–251.

[20] BENTLEY, J. and C.C. MCGEOCH, Amortized analysis of self-organizing sequential search heuristics, *Comm. ACM* **28**(4) (1985) 405–411.

[21] BENTLEY, J. and J.B. SAXE, Decomposable searching problems I: static-to-dynamic transformations, *J. Algorithms* **1** (1980) 301–358.

[22] BITNER, J.R., Heuristics that dynamically organize data structures, *SIAM J. Comput.* **8**(1) (1979) 82–110.

[23] BLUM, M., R.W. FLOYD, V. PRATT, R.L. LEWIS and R.E. TARJAN, Time bounds for selection, *J. Comput. System Sci.* **7** (1973) 448–461.

[24] BLUM, N., On the single-operation worst-case time complexity of the disjoint set union problem, *SIAM J. Comput.* **15**(4) (1986) 1021–1924.

[25] BLUM, N. and K. MEHLHORN, On the average number of rebalancing operations in weight-balanced trees, *Theoret. Comput. Sci.* **11** (1980) 303–320.

[26] BOLLOBÁS, B. and I. SIMON, On the expected behavior of disjoint set union algorithms, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing* (1985) 224–231.

[27] BOLLOBÁS, B. and I. SIMON, Repeated random insertion into a priority queue, *J. Algorithms* **6** (1985) 466–477.

[28] BORODIN, A., F. FICH, F. MEYER AUF DER HEIDE, E. UPFAL and A. WIGDERSON, A tradeoff between search and update time for the implicit dictionary problem, in: *Proc. 13th Internat. Coll. on Automata, Languages an Programming*, Lecture Notes in Computer Science, Vol. 226 (Springer, Berlin, 1986) 50–59.

[29] BORODIN, A., L.J. GUIBAS, N.A. LYNCH and A.C. YAO, Efficient searching using partial ordering, *Inform. Process. Lett.* **12** (1981) 71 – 75.

[30] BROWN, M.R., Implementation and analysis of binomial queue algorithms, *SIAM J. Comput.* **7** (1978) 298–319.

[31] BROWN, M.R., A partial analysis of random height-balanced trees, *SIAM J. Comput.* **8**(1) (1979) 33–41.

[32] BROWN, M.R. and R.E. TARJAN, A fast merging algorithm, *J. ACM* **26**(2) (1979) 211–226.

[33] BROWN, M.R. and R.E. TARJAN, Design and analysis of a data structure for representing sorted lists, *SIAM J. Comput.* **9** (1980) 594–614.

[34] BURVILLE, P. and J. KINGMAN, On a model for storage and search, *J. Appl. Probab.* **10**(3) (1973) 697–701.

[35] CARLSSON, S., Splitmerge—a fast stable merging algorithm, *Inform. Process. Lett.* **22** (1986) 189–192.

[36] CARLSSON, S., J.F. MUNRO and P.V. POBLETE, An implicit binomial queue with constant insertion time, in: *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT 88)*, Halmstad, Sweden, Lecture Notes in Computer Science, Vol. 318 (Springer, Berlin, 1988) 1–13.

[37] CARTER, J.L. and M.N. WEGMAN, Universal classes of hash functions, in: *Proc. 9th Ann. ACM Symp. on Theory of Computing* (1977) 106–112.

[38] CHAZELLE, B., How to search in history, *Inform. and Control* **63** (1985) 77–99.

[39] CHAZELLE, B. and L.J. GUIBAS, Fractional cascading I: a data structuring technique, *Algorithmica* **1**(2) (1986) 133–162.

[40] CHUNG, F.R., and D.J. HAJELA and P.D. SEYMOUR, Self-organizing sequential search and Hilbert's inequalities, *J. Comput. System Sci.* **36** (1988) 148–157.

[41] COLE, R., Searching and storing similar lists, *J. Algorithms* **7** (1986) 202–220.

[42] CRANE, C.A., Linear lists and priority queues as balanced binary trees, Tech. Report, STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.

[43] CUNTO, W. and J.I. MUNRO, Average case selection, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 369–375.

[44] DEVROYE, L., A note on the height of binary search trees, *J. ACM* **33**(3) (1986) 489–498.

[45] DEVROYE, L., Branching processes in the analysis of the heights of trees, *Acta Inform.* **24** (1987) 277–298.

[46] DIETZFELBINGER, M., A. KARLIN, K.MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT and R.E. TARJAN, Dynamic perfect hashing: upper and lower bounds, in: *Proc. 29th Ann IEEE Symp. on Foundations of Computer Science* (1988) 524–531.

[47] DOBERKAT, E.E., An average case analysis of Floyd's algorithm to construct heaps, *Inform. and Control* **61** (1984) 114–131.

[48] DOBKIN, D. and J.I. MUNRO, Optimal minimal space selection algorithms, *J. ACM* **28**(3) (1981) 454–461.

[49] DOBKIN, D. and J.I. MUNRO, Efficient uses of the past, *J. Algorithms* **6** (1985) 455–465.

[50] DODGSON, C.L., *St. James Gazette* **5–6** (1 August 1883).

[51] DOYLE, I. and R. RIVEST, Linear expected time of a simple union-find algorithm, *Inform. Process. Lett.* **5** (1976) 146–148.

[52] DRISCOLL, J.R., H.N. GABOW, R. SHRAIRMAN and R.E. TARJAN, Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation, *Comm. ACM* **31**(11) (1988) 1343–1354.

[53] DRISCOLL, J.R., N. SARNAK, D.D. SLEATOR and R.E. TARJAN, Making data structures persistent, *J. Comput. System Sci.* **28** (1989) 86–124.

[54] DUDZIŃSKI, K. and A. DYDELL, On a stable minimum space merging algorithm, *Inform. Process. Lett.* **12**(1) (1981) 5–8.

[55] EDELSBRUNNER, H., L.J. GUIBAS and J. STOLFI, Optimal point location in a monotone subdivision, *SIAM J. Comput.* **15**(2) (1986) 317–340.
[56] EDELSBRUNNER, H. and M.H. OVERMARS, Batched dynamic solutions to decomposable searching problems, *J. Algorithms* **6** (1985) 515–542.
[57] EMDE BOAS, P. VAN, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977) 80–82.
[58] EMDE BOAS, P. VAN, R. KAAS and E. ZIJLSTRA, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977) 99–127.
[59] FAGIN, R., J. NIEVERGELT, N. PIPPENGER and H.R. STRONG, Extendible hashing—a fast access method for dynamic files, *ACM Trans. Database Systems* **4** (1979) 315–344.
[60] FALLER, N., An adaptive system for data compression, in: *Record 7th Asilomar Conf. on Circuits, Systems, and Computers* (1973) 593–597.
[61] FISCHER, M.J., Efficiency of equivalence algorithms, in: R.E. Miller and J.W. Thatcher, eds., *Complexity of Computer Computation* (Plenum Press, New York, 1972) 153–168.
[62] FLAJOLET, PH., On the performance evaluation of extendible hashing and trie searching, *Acta Inform.* **20** (1983) 345–369.
[63] FLOYD, R.W., Algorithm 245: Treesort 3, *Comm. ACM* **7** (1964) 701.
[64] FLOYD, R.W. and R.L. RIVEST, Expected time bounds for selection, *Comm. ACM* **18** (1975) 165–172.
[65] FOSTER, C.C., Information storage and retrieval using AVL-trees, in: *Proc. ACM 20th Nat. Conf.* (1965) 192–205.
[66] FOSTER, C.C., A generalization of AVL-trees, *Comm. ACM* **16**(8) (1973) 513–517.
[67] FREDERICKSON, G.N., Implicit data structures for the dictionary problem, *J. ACM* **30**(1) (1983) 80–94.
[68] FREDERICKSON, G.N., Self-organizing heuristic for implicit data structures, *SIAM J. Comput.* **13**(2) (1984) 277–291.
[69] FREDERICKSON, G.N., Implicit data structures for weighted elements, *Inform. and Control* **66** (1985) 61–82.
[70] FREDERICKSON, G.N., Upper bounds for time-space tradeoffs in sorting and selection, *J. Comput. System Sci.* **34** (1987) 19–26.
[71] FREDKIN, E., Trie memory, *Comm. ACM* **3** (1962) 490–499.
[72] FREDMAN, M.L., Two applications of a probabilistic search technique: sorting $X+Y$ and building balanced search trees, in: *Proc. 7th Ann. ACM Symp. on Theory of Computing* (1975) 240–244.
[73] FREDMAN, M.L., J. KOMLÓS and E. SZEMERÉDI, Storing a sparse table with O(1) worst case access time, *J. ACM* **31**(3) (1984) 538–544.
[74] FREDMAN, M.L., R. SEDGEWICK, D.D. SLEATOR and R.E. TARJAN, The pairing heap: a new form of self-adjusting heap, *Algorithmica* **1**(1) (1986) 111–129.
[75] FREDMAN, M.L. and R.E. TARJAN, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34**(3) (1987) 596–615.
[76] GABOW, H.N., A scaling algorithm for weighted matching in general graphs, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 90–100.
[77] GABOW, H.N. and R.E. TARJAN, A linear time algorithm for a special case of disjoint set union, *J. Comput. System Sci.* **30** (1985) 209–221.
[78] GALLAGER, R.G., Variations on a theme by Huffman, *IEEE Trans. Inform. Theory* **24** (1978) 668–674.
[79] GALLER, B.A. and M.J. FISCHER, An improved equivalence algorithm, *Comm. ACM* **7** (1964) 301–303.
[80] GAMBOSI, G., G.F. ITALIANO and M. TALAMO, Getting back to the past in the union-find problem, in: *Proc. STACS 88, 5th Ann. Symp.*, Lecture Notes in Computer Science, Vol. 294 (Springer, Berlin, 1988) 8–17.
[81] GARSIA, A.M. and M.L. WACHS, A new algorithm for minimal binary search trees, *SIAM J. Comput.* **6** (1977) 622–642.
[82] GONNET, G.H., Balancing binary trees by internal path reduction, *Comm. ACM* **26**(12) (1983) 1074–1081.
[83] GONNET, G.H., *Handbook of Algorithms and Data Structures*, International Computer Science Series (Addison-Wesley, Reading, MA, 1984).
[84] GONNET, G.H. and I. MUNRO, Heaps on heaps, *SIAM J. Comput.* **15**(4) (1986) 964–971.
[85] GONNET, G.H., I. MUNRO and H. SUWANDA, Exegesis of self-organizing linear search, *SIAM J. Comput.* **10**(3) (1981) 613–637.

[86] GONNET, G.H., L. ROGERS and I. GEORGE, An algorithmic and complexity analysis of interpolation search, *Acta Inform.* **3**(1) (1980) 39–52.
[87] GÜTING, H. and H.P. KRIEGEL, Multidimensional B-tree: an efficient dynamic file structure for exact match queries, *Informatik Fachberichte* **33** (1980) 375–388.
[88] GUIBAS, L.J., E.M. MCCREIGHT, M.F. PLASS and J.R. ROBERTS, A new representation for linear lists, in: *Proc. 9th Ann. ACM Symp. on Theory of Computing* (1977) 49–60.
[89] GUIBAS, L.J. and R. SEDGEWICK, A dichromatic framework for balanced trees, in: *Proc. 19th Ann. IEEE Symp. on Foundations of Computer Science* (1978) 8–21.
[90] GULBERSON, J., The effect of updates in binary search trees, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing* (1985) 205–210.
[91] HADIAN, A. and M. SOBEL, Selecting the $t$th largest using binary errorless comparisons, in: P. Erdös, A. Renyi and V.T. Sós, eds., *Combinatorial Theory and its Applications II*, Colloquia Mathematica Societatis Janos Bolyai, Vol. 4 (North-Holland, Amsterdam, 1969) 585–599.
[92] HAREL, D. and R.E. TARJAN, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13** (1984) 338–355.
[93] HENDRICKS, W.J., The stationary distribution of an interesting Markov chain, *J. Appl. Probab.* **9**(1) (1972) 231–233.
[94] HOARE, C.A.R., Quicksort, *Comput. J.* **5** (1962) 10–15.
[95] HOFFMANN, K., K. MEHLHORN, P. ROSENSTIEHL and R.E. TARJAN, Sorting Jordan sequences in linear time using level-linked search trees, *Inform. and Control* **68** (1986) 170–184.
[96] HOOD, R. and R. MELVILLE, Real-time queue operations in pure LISP, *Inform. Process. Lett.* **13** (1981) 50–54.
[97] HOPCROFT, J.E. and J.D. ULLMAN, Set merging algorithms, *SIAM J. Comput.* **2** (1973) 294–303.
[98] HOROWITZ, E. and S. SAHNI, *Fundamentals of Data Structures* (Computer Science Press, Potomac MD, 1976).
[99] HU, T.C. and A.C. TUCKER, Optimal computer-search trees and variable-length alphabetic codes, *SIAM J. Appl. Math.* **21** (1971) 514–532.
[100] HUDDLESTON, S., An efficient scheme for fast local updates in linear lists, Tech. Report, Dept. of Information and Computer Science, Univ. of California, Irvine, 1981.
[101] HUDDLESTON, S. and K. MEHLHORN, A new data structure for representing sorted lists, *Acta Inform.* **17** (1982) 157–184.
[102] HUFFMAN, D.A., A method for the construction of minimum redundancy codes, *Proc. IRE* **40** (1952) 1098–1101.
[103] HWANG, F.K. and S. LIN, A simple algorithm for merging two disjoint linearly ordered sets, *SIAM J. Comput.* **1**(1) (1972) 31–39.
[104] IMAI, H. and T. ASANO, Dynamic orthogonal segment intersection search, *J. Algorithms* **8** (1987) 1–18.
[105] ITAI, A., A.G. KONHEIM and M. RODEH, A sparse table implementation of priority queues, in: *Proc. 8th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 115 (Springer, Berlin, 1981) 417–431.
[106] JACOBS, T.M. and P. VAN EMDE BOAS, Two results on tables, *Inform. Process. Lett.* **22** (1986) 43–48.
[107] JOHNSON, D., Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977) 1–3.
[108] KINGSTON, J.H., A new proof of the Garsia–Wachs algorithm, *J. Algorithms* **9** (1988) 129–136.
[109] KIRKPATRICK, D.G., A unified lower bound for selection and set partitioning problem, *J. ACM* **28** (1981) 150–165.
[110] KNOTT, G.D., Expandable open addressing hash table storage and retrieval, in: *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control* (1971) 186–206.
[111] KNUTH, D.E., Optimum binary search tree, *Acta Inform.* **1** (1971) 14–25.
[112] KNUTH, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).
[113] KNUTH, D.E., Deletions that preserve randomness, *IEEE Trans. Software Engrg.* **3**(5) (1977) 351–359.
[114] KNUTH, D.E., Dynamic Huffman coding, *J. Algorithms* **6** (1985) 163–180.
[115] KNUTH, D.E. and A. SCHÖNHAGE, The expected linearity of a simple equivalence algorithm, *Theoret. Comput. Sci.* **6** (1978) 281–315.
[116] KORSCH, J.F., Greedy binary search trees are nearly optimal, *Inform. Process. Lett.* **13** (1981) 16–19.

[117] KOSARAJU, S.R., Localized search in sorted list, in: *Proc. 13th Ann. ACM Symp. on Theory of Computing* (1981) 62–69.

[118] KRONROD, M.A., Optimal ordering algorithm without operational field, *Soviet. Math. Dokl.* **10** (1969) 744–746.

[119] KRIEGEL, H.P. and V.K. VAISHNAVI, Weighted multidimensional B-trees used as nearly optimal dynamic dictionaries, in: *Proc. Mathematical Foundations of Computer Science*, Štrbské Pleso, Czechoslovakia, Lecture Notes in Computer Science, Vol. 118 (Springer, Berlin, 1981) 410–417.

[120] LAI, T.W. and D. WOOD, Implicit selection, in: *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT 88)*, Halmstad, Sweden, Lecture Notes in Computer Science, Vol. 318 (Springer, Berlin, 1988) 14–23.

[121] LAM, K., M.K. SIU and C.T. YU, A generalized counter scheme, *Theoret. Comput. Sci.* **16** (1981) 271–278.

[122] LARSON, P.A., Dynamic hashing, *BIT* **18** (1978) 184–201.

[123] LEEUWEN, J. VAN, Finding lowest common ancestors in less than logarithmic time, Unpublished report, Amherst, NY, 1976.

[124] LEEUWEN, J. VAN and H.A. MAURER, Dynamic systems of static data structures, Bericht 42, Institut für Informationsverarbeitung, TU Graz, Austria, 1980.

[125] LEEUWEN, J. VAN and A.K. TSAKALIDIS, An optimal pointer machine algorithm for nearest common ancestors, Tech. Report, UU-CS-88-17, Dept. of Computer Science, Univ. of Utrecht, Utrecht, 1988.

[126] LEVCOPOULUS, C. and M.H. OVERMARS, A balanced search tree with O(1) worst-case update time, *Acta Inform.* **26** (1988) 269–277.

[127] LINIAL, N. and M.E. SAKS, Information bounds are good for search problems on ordered data structures, in: *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Science* (1983) 473–475.

[128] LIPSKI, W., An $O(n \log n)$ Manhattan path algorithm, *Inform. Process. Lett.* **19** (1984) 99–102.

[129] LITWIN, W., Virtual hashing: a dynamically changing hashing, in: *Proc. Very Large Data Bases Conf.*, Berlin (1978) 517–523.

[130] MAIER, D., An efficient method for sorting ancestor information in trees, *SIAM J. Comput.* **8**(4) (1979) 599–618.

[131] MAIER, D. and S.C. SALVETER, Hysterical B-trees, *Inform. Process. Lett.* **12** (1981) 199–202.

[132] MAIRSON, H., Average case lower bounds on the construction and searching of partial orders, in: *Proc. 26th Ann. IEEE Symp. Foundation of Computer Science* (1985) 303–311.

[133] MANNILA., H. and E. UKKONEN, The set union problem with backtracking, in: *Proc. 13th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 226 (Springer, Berlin, 1986) 236–243.

[134] MARKOWSKY, G., J.L. CARTER and M.N. WEGMAN, Analysis of a universal class of hash functions, in: *Proc. Mathematical Foundations of Computer Science*, Zakapone, Poland, Lecture Notes in Computer Science, Vol. 64 (Springer, Berlin, 1978) 345–354.

[135] MATULA, D.W., Selecting the $t$-th best in average $n + O(\log \log n)$ comparisons, Tech. Report 73-9, Dept. of Mathematics, Washington Univ., St. Louis, 1973.

[136] MCCABE, J., On serial file with relocatable records, *Oper. Res.* **12** (1965) 609–618.

[137] MCDIARMID, C., Average-case lower bounds for searching, *SIAM J. Comput.* **17**(1) (1988) 1044–1060.

[138] MEHLHORN, K., Nearly optimal binary search trees, *Acta Inform.* **5** (1975) 287–295.

[139] MEHLHORN, K., A best possible bound for the weighted path length of binary search trees, *SIAM J. Comput.* **6**(2) (1977) 235–239.

[140] MEHLHORN, K., Dynamic binary search, *SIAM J. Comput.* **8** (1979) 175–198.

[141] MEHLHORN, K., Arbitrary weight changes in dynamic trees, *RAIRO Inform. Théor.* **15**(3) (1981) 183–211.

[142] MEHLHORN, K., A partial analysis of height-balanced trees under random insertions and deletions, *SIAM J. Comput.* **11** (1982) 748–760.

[143] MEHLHORN, K., On the program size of perfect and universal hash functions, in: *Proc. 23rd Ann. IEEE Symp. on Foundations of Computer Science* (1982) 170–175.

[144] MEHLHORN, K., Lower bounds on the efficiency of transforming static data structures into dynamic data structures, *Math. Systems Theory* **15** (1982) 1–16.

[145] MEHLHORN, K., *Data Structures and Algorithms 1: Sorting and Searching*, EATCS Monographs on Theoretical Computer Science (Springer, Berlin, 1984).

[146] MEHLHORN, K. and ST. NÄHER, Dynamic fractional cascading, Tech. Report A 06/86, Univ. of Saarland, Saarbrücken, FRG; also: *Algorithmica*, to appear.

[147] MEHLHORN, K., ST. NÄHER and H. ALT, A lower bound for the complexity of the union-split-find problem, in: *Proc. 14th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 267 (Springer, Berlin, 1987); also: *SIAM J. Comput.*, to appear.

[148] MEHLHORN, K., ST. NÄHER and CH. UHRIG, Deleting versions in persistent data structures, Tech. Report, Univ. of Saarland, Saarbrücken, 1989.

[149] MEHLHORN, K. and M.H. OVERMARS, Optimal dynamization of decomposable searching problems, *Inform. Process. Lett.* **12**(2) (1981) 93–98.

[150] MEHLHORN, K. and A.K. TSAKALIDIS, Dynamic interpolation search, in: *Proc. 12th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 194 (Springer, Berlin, 1985) 424–434.

[151] MEHLHORN, K. and A.K. TSAKALIDIS, An amortized analysis of insertions into AVL-trees, *SIAM J. Comput.* **15**(1) (1986) 22–33.

[152] MUNRO, I., An implicit data structure supporting insertion, deletion and search in $O(\log^2 n)$ time, *J. Comput. System Sci.* **33** (1986) 66–74.

[153] MUNRO, I., Developing implicit data structures, in: *Proc. Mathematical Foundations of Computer Science*, Bratislava, Czechoslovakia, Lecture Notes in Computer Science, Vol. 233 (Springer, Berlin, 1986) 168–176.

[154] MUNRO, I. and M.S. PATERSON, Selecting and sorting with limited space, *Theoret. Comput. Sci.* **12** (1980) 315–323.

[155] MUNRO, I. and P. POBLETE, Searchability in merging and implicit data structures, in: *Proc. 10th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 154 (Springer, Berlin, 1983) 527–535.

[156] MUNRO, I. and H. SUWANDA, Implicit data structures, *J. Comput. System Sci.* **21** (1980) 236–250.

[157] MYERS, E.W., An applicative random-access stack, *Inform. Process. Lett.* **17** (1983) 241–248.

[158] MYERS, E.W., Efficient applicative data types, in: *Conf. Record 11th Ann. ACM Symp. on Principles of Programming Languages* (1984) 66–75.

[159] NIEVERGELT, I. and E.M. REINGOLD, Binary search trees of bounded balance, *SIAM J. Comput.* **2** (1973) 33–43.

[160] OLIVIE, H.J., A new class of balanced search trees: half balanced binary search tree, *RAIRO Inform. Théor.* **16**(1) (1982) 51–71.

[161] ORLIN, J.B. and R.K. AHUJA, A fast and simple shortest path algorithm, in: *Cornell Workshop on Combinatorial Optimization*, Ithaca, NY, 1987.

[162] OTTMANN, TH. and H.W. SIX, Eine neue Klasse von ausgeglichenen Bäumen, *Angewandte Informatik* **18** (1976) 395–400.

[163] OVERMARS, M.H., Searching in the past I, Report RUU-CS-81-7, Dept. of Computer Science, Univ. of Utrecht, Utrecht, 1981.

[164] OVERMARS, M.H., Dynamization of order decomposable set problems, *J. Algorithms* **2** (1981) 245–260.

[165] OVERMARS, M.H., *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156 (Springer, Berlin, 1983).

[166] OVERMARS, M.H. and J. VAN LEEUWEN, Two general methods for dynamizing decomposable searching problems, *Comput.* **26** (1981) 155–166.

[167] OVERMARS, M.H. and J. VAN LEEUWEN, Worst-case optimal insertion and deletion methods for decomposable searching problems, *Inform. Process. Lett.* **12**(4) (1981) 168–173.

[168] PEARL, Y., A. ITAI and H. AVNI, Interpolation search—a log log $N$ search, *Comm. ACM* **21**(7) (1978) 550–554.

[169] PEARL, Y. and E.M. REINGOLD, Understanding the complexity of interpolation search, *Inform. Process. Lett.* **6** (1977) 219–222.

[170] PETERSON, W.W., Addressing for random storage, *IBM J. Res. Develop.* **1** (1957) 131–132.

[171] PORTER, T. and I. SIMON, Random insertion into a priority queue structure, *IEEE Trans. Software Engrg.* **1** (1975) 292–298.
[172] RAMANAN, P.V. and L. HYAFIL, New algorithms for selection, *J. Algorithms* **5** (1984) 557–578.
[173] RAO, N.S.V., V.K. VAISHNAVI and S.S. IYENGAR, On the dynamization of data structures, *BIT* **28** (1988) 37–53.
[174] REPS, T., T. TEITELBAUM and A. DEMERS, Incremental context-dependent analysis for language-based editors, *ACM Trans. Programming Systems and Languages* **5** (1983) 449–477.
[175] RIVEST, R., On self-organizing sequential search heuristics, *Comm. ACM* **2** (1976) 63–67.
[176] SANTORO, N. and I. SIDNEY, Interpolation-binary search, *Inform. Process. Lett.* **20** (1985) 179–181.
[177] SARNAK, N. and R.E. TARJAN, Planar point location using persistent search trees, *Comm. ACM* **29** (1986) 669–679.
[178] SCHIEBER, B. and U. VISHKIN, On finding lowest common ancestors: simplification and parallelization, in: *Proc. 3rd Aegean Workshop on VLSI Algorithms and Architecture*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 111–123.
[179] SCHÖNHAGE, A., Storage modification machines, *SIAM J. Comput.* **9**(3) (1980) 490–508.
[180] SCHÖNHAGE, A., M. PATERSON and N. PIPPENGER, Finding the median, *J. Comput. System Sci.* **13** (1976) 184–199.
[181] SEDGEWICK, R., *Algorithms* (Addison-Wesley, Reading, MA, 2nd ed., 1988).
[182] SLEATOR, D.D. and R.E. TARJAN, A data structure for dynamic trees, *J. Comput. System Sci.* **26** (1983) 362–391.
[183] SLEATOR, D.D. and R.E. TARJAN, Self-adjusting binary search trees, *J. ACM* **32**(3) (1985) 652–686.
[184] SLEATOR, D.D. and R.E. TARJAN, Amortized efficiency of list update and paging rules, *Comm. ACM* **28** (2) (1985) 202–208.
[185] SLEATOR, D.D. and R.E. TARJAN, Self adjusting heaps, *SIAM J. Comput.* **15**(1) (1986) 52–68.
[186] SPRUGNOLI, R., Perfect hash functions: a single probe retrieval method for static sets, *Comm. ACM* **20** (1977) 841–850.
[187] STANDISH, T.A., *Data Structure Techniques* (Addison-Wesley, Reading, MA, 1980).
[188] STEINHAUS, H., Some remarks about tournaments, in: *Calcutta Math. Soc. Golden Commemoration* **2** (1958).
[189] STOCKMEYER, P. and F.F. YAO, On the optimality of linear merge, *SIAM J. Comput.* **9**(1) (1980) 85–90.
[190] SWART, G.F., Efficient algorithms for computing geometric intersections, Tech. Report 85-01-02, Dept. of Computer Science, Univ. of Washington, Seattle, WA, 1985.
[191] TAMMINEN, M., Extendible hashing with overflow, *Inform. Process. Lett.* **15**(5) (1982) 227–233.
[192] TARJAN, R.E., Efficiency of a good but not linear set union algorithm, *J. ACM* **22** (1975) 215–225.
[193] TARJAN, R.E., A class of algorithms which require non-linear time to maintain disjoint sets, *J. Comput. System Sci.* **18** (1979) 110–127.
[194] TARJAN, R.E., Private communication, 1982.
[195] TARJAN, R.E., *Data Structures and Network Algorithms* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983).
[196] TARJAN, R.E., Updating a balanced search tree in O(1) rotations, *Inform. Process. Lett.* **16** (1983) 253–257.
[197] TARJAN, R.E., Sequential access in splay trees takes linear time, *Combinatorica* **5**(4) (1985) 367–378.
[198] TARJAN, R.E., Amortized computational complexity, *SIAM J. Alg. Discrete Meth.* **2**(6) (1985) 306–318.
[199] TARJAN, R.E. and J. VAN LEEUWEN, Worst-case analysis of set union algorithms, *J. ACM* **31** (1984) 245–281.
[200] TARJAN, R.E. and A.C. YAO, Storing a sparse table, *Comm. ACM* **22**(11) (1979) 606–611.
[201] THANH, M., V.S. ALAGAR and T.O. BUI, Optimal expected-time algorithms for merging, *J. Algorithms* **7** (1986) 341–357.
[202] TRABB PARDO, L., Stable sorting and merging with optimal space and time, *SIAM J. Comput.* **6**(2) (1977) 351–372.
[203] TSAKALIDIS, A.K., AVL-trees for localized search, *Inform. and Control* **67**(1–3) (1985) 173–194.
[204] TSAKALIDIS, A.K., A simple implementation for localized search, in: *Proc. WG'85, Internat. Workshop on Graph-theoretical Concepts in Computer Science*, Würzburg, FRG (Trauner-Verlag, Linz, 1985) 363–374; also: *SIAM J. Comput.*, to appear.

[205] TSAKALIDIS, A.K., Rebalancing operations for deletions in AVL-trees, *RAIRO Inform. Théor.* **19**(4) (1985) 323–329.
[206] TSAKALIDIS, A.K., The nearest common ancestor in a dynamic tree, *Acta Inform.* **25** (1988) 37–54.
[207] UNTERAUER, K., Dynamic weighted binary search trees, *Acta Inform.* **11** (1979) 341–362.
[208] VAISHNAVI, V.K., Weighted leaf AVL-trees, *SIAM J. Comput.* **16**(3) (1987) 503–537.
[209] VAISHNAVI, V.K. and D. WOOD, Rectilinear segment intersection layered segment trees and dynamization, *J. Algorithms* **3** (1982) 160–176.
[210] VITTER, J., Design and analysis of dynamic Huffman coding, *J. ACM* **34**(4) (1987) 825–845.
[211] VUILLEMIN, J., A data structure for manipulating priority queues, *Comm. ACM* **21** (1978) 309–314.
[212] WESTBROOK, J., and R.E. TARJAN, Amortized analysis of algorithms for set union with backtracking, *SIAM J. Comput.* **18** (1989) 1–11.
[213] WILLARD, D.E., Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, *Inform. Process. Lett.* **17** (1983) 81–84.
[214] WILLARD, D.E., New trie data structures which support very fast search operations, *J. Comput. System Sci.* **28** (1984) 379–394.
[215] WILLARD, D.E., Searching unindexed and nonuniformly generated files in $\log \log N$ time, *SIAM J. Comput.* **14**(4) (1985) 1013–1029.
[216] WILLARD, D.E., New data structures for orthogonal queries, *SIAM J. Comput.* **14**(1) (1985) 232–235.
[217] WILLARD, D.E. and G.S. LUECKER, Adding range restriction capability to dynamic data structures, *J. ACM* **32**(3) (1985) 597–617.
[218] WILLIAMS, J.W.J., Algorithm 232: Heapsort, *Comm. ACM* **7** (1964) 347–348.
[219] WIRTH, N., *Algorithms + Data Structures = Programs* (Prentice Hall, Englewood Cliffs, NJ, 1976).
[220] YAO, A.C., On the average behavior of set merging algorithms, in: *Proc. 8th Ann. ACM Symp. Theory of Computing* (1976) 192–195.
[221] YAO, A.C., On the expected performance of path compression algorithms, *SIAM J. Comput.* **14**(1) (1985) 129–133.
[222] YAO, A.C. and F.F. YAO, The complexity of searching on ordered random table, in: *Proc. 17th Ann. IEEE Symp. Foundations of Computer Science* (1976) 173–177.
[223] YAO, A.C. and F.F. YAO, On the average case complexity of selecting the $k$-th best, *SIAM J. Comput.* **11**(3) (1982) 428–447.
[224] YAO, F.F., Efficient dynamic programming using quadrangle inequalities, in: *Proc. 12th Ann. ACM Symp. Theory of Computing* (1980) 429–435.
[225] ZIEGLER, S.F., Smaller faster table driven parser, Unpublished manuscript, Madison Academic Computing Center, Univ. of Wisconsin, Madison, WI, 1977.