Data Structures*

Roberto Tamassia *Bryan Cantrill*

Department of Computer Science
Brown University
115 Waterman Street
Providence, RI 02912–1910
{rt,bmc}@cs.brown.edu

January 27, 1996

*Revised version for Chapter 6 of the *CRC Handbook of Computer Science and Engineering*.

Contents

List of Figures

List of Tables

1 Introduction

The study of data structures, i.e., methods for organizing data that are suitable for computer processing, is one of the classic topics of computer science. At the hardware level, a computer views storage devices such as internal memory and disk as holders of elementary data units (bytes), each accessible through its address (an integer). When writing programs, instead of manipulating the data at the byte level, it is convenient to organize them into higher level entities, called *data structures*.

1.1 Containers, Elements, and Locators

Most data structures can be viewed as *containers* that store a collection of objects of a given type, called the *elements* of the container. Often a total order is defined among the elements (e.g., alphabetically ordered names, points in the plane ordered by x-coordinate). We assume that the elements of a container can be accessed by means of variables called *locators*. When an object is inserted into the container, a locator is returned, which can be later used to access or delete the object. A locator is typically implemented with a pointer or an index into an array.

A data structure has an associated repertory of operations, classified into *queries*, which retrieve information on the data structure (e.g., return the number of elements, or test the presence of a given element), and *updates*, which modify the data structure (e.g., insertion and deletion of elements). The performance of a data structure is characterized by the space requirement and the time complexity of the operations in its repertory. The *amortized* time complexity of an operation is the average time over a suitably defined sequence of operations.

However, efficiency is not the only quality measure of a data structure. Simplicity and ease of implementation should be taken into account when choosing a data structure for solving a practical problem.

1.2 Abstract Data Types

Data structures are concrete implementations of *abstract data types* (ADT's). A *data type* is a collection of objects. A data type can be mathematically specified (e.g., real number, directed graph) or concretely specified within a programming language (e.g., `int` in C, `set` in Pascal). An ADT is a mathematically specified data type equipped with operations that can be performed on the objects. Object-oriented programming languages, such as C++, provide support for expressing ADTs by means of *classes*. ADTs specify the data stored and the operations to be performed on them.

1.3 Main Issues in the Study of Data Structures

The following issues are of foremost importance in the study of data structures.

Static vs. Dynamic A *static* data structure supports only queries, while a dynamic data structure supports also updates. A *dynamic* data structure is often more complicated than its static counterpart supporting the same repertory of queries. A *persistent* data structure (see, e.g., [9]) is a dynamic data structure that supports operations on past versions. There are many problems for which no efficient dynamic data structures are known. It has been observed that there are strong similarities among the classes of problems that are difficult to parallelize and those that are difficult to dynamize (see, e.g., [31]). Further investigations are needed to study the relationship between parallel and incremental complexity [25].

Implicit vs. Explicit Two fundamental data organization mechanisms are used in data structures. In an *explicit* data structure, pointers (i.e., memory addresses) are used to link the elements and access them (e.g., a singly linked list, where each element has a pointer to the next one). In an *implicit* data structure, mathematical relationships support the retrieval of elements (e.g., array representation of a heap, see Section 3.4.4). Explicit data structures must use additional space to store pointers. However, they are more flexible for complex problems. Most programming languages support pointers and basic implicit data structures, such as arrays.

Internal vs. External Memory In a typical computer, there are two levels of memory: internal memory (RAM) and external memory (disk). The internal memory is much faster than external memory but has much smaller capacity. Data structures designed to work for data that fit into internal memory may not perform well for large amounts of data that need to be stored in external memory. For large-scale problems, data structures need to be designed that take into account the two levels of memory [1]. For example, two-level indices such as B-trees [6] have been designed to efficiently search in large databases.

Space vs. Time Data structures often exhibit a tradeoff between space and time complexity. For example, suppose we want to represent a set of integers in the range $[0, N]$ (e.g., for a set of social security numbers $N = 10^{10}-1$) such that we can efficiently query whether a given element is in the set, insert an element, or delete an element. Two possible data structures for this problem are an N-element bit-array (where the bit in position i indicates the presence of integer i in the set), and a balanced search tree (such as a 2-3 tree or a red-black tree). The bit-array has optimal time complexity, since it supports queries, insertions and deletions in constant time. However, it uses space proportional to the size N of the range, irrespectively of the number of elements actually stored. The balanced search tree supports queries, insertions and deletions in logarithmic time but uses optimal space proportional to the current number of elements stored.

Theory vs. Practice A large and ever growing body of theoretical research on data structures is available, where the performance is measured in asymptotic terms ("big-Oh" notation). While asymptotic complexity analysis is an important mathematical subject, it does not completely capture the notion of efficiency of data structures in practical scenarios, where constant factors cannot be disregarded and the difficulty of implementation substantially affects design and maintenance costs. Experimental studies comparing the practical efficiency of data structures for specific classes of problems should be encouraged to bridge the gap between the theory and practice of data structures.

1.4 Fundamental Data Structures

The following four data structures are ubiquitously used in the description of discrete algorithms, and serve as basic building blocks for realizing more complex data structures. They are covered in detail in the textbooks listed in Section 6 and in the additional references provided.

Sequence A sequence is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of sequences include stacks and queues, where insertions and deletions can be done only at the head or tail of the sequence. The basic realization of sequences are by means of arrays and linked lists. Concatenable queues (see, e.g., [17]) support additional operations such as splitting and splicing, and determining the sequence containing a given element. In external memory, a sequence is typically associated with a file.

Priority Queue A priority queue is a container of elements from a totally ordered universe that supports the basic operations of inserting an element and retrieving/removing the largest element. A key application of priority queues is to sorting algorithms. A heap is an efficient realization of a priority queue that embeds the elements into the ancestor/descendant partial order of a binary tree. A heap also admit an implicit realization where the nodes of the tree are mapped into the elements of an array (see Section 3.4.4). Sophisticated variations of priority queues include min-max heaps, pagodas, deaps, binomial heaps, and Fibonacci heaps. The buffer tree is efficient external-memory realization of a priority queue.

Dictionary A dictionary is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. Hash tables provide an efficient implicit realization of a dictionary. Efficient explicit implementations include skip lists [30], tries, and balanced search trees (e.g., AVL-trees, red-black trees, 2-3 trees, 2-3-4 trees, weight-balanced trees, biased search trees, splay trees). The technique of fractional cascading [3] speeds up searching for the same element in a collection of dictionaries. In external memory, dictionaries are typically implemented as B-trees and their variations.

Union-Find A union-find data structure represents a collection disjoint sets and supports the two fundamental operations of merging two sets and finding the set containing a given element. There is a simple and optimal union-find data structure (rooted tree with path compression) whose time complexity analysis is very difficult to analyze. See, e.g., [15].

Examples of fundamental data structures used in three major application domains are mentioned below.

Graphs and Networks adjacency matrix, adjacency lists, link-cut tree [33], dynamic expression tree [5], topology tree [14], SPQR-tree [8], sparsification tree [11]. See also, e.g., [12, 22, 34].

Text Processing string, suffix tree, Patricia tree. See, e.g., [16].

Geometry and Graphics binary space partition tree, chain tree, trapezoid tree, range tree, segment-tree, interval-tree, priority-search tree, hull-tree, quad-tree, R-tree, grid file, metablock tree. See, e.g., [4, 10, 13, 22, 26, 27, 29].

1.5 Organization of the Chapter

The rest of this chapter focuses on three fundamental abstract data types: sequences, priority queues, and dictionaries. Examples of efficient data structures and algorithms for implementing them are presented in detail in Sections 2, 3 and 4, respectively. Namely, we cover arrays, singly- and doubly-linked lists, heaps, search trees, (a, b)-trees, AVL-trees, bucket arrays, and hash tables.

2 Sequence

2.1 Introduction

A *sequence* is a container that stores elements in a certain order, which is imposed by the operations performed. The basic operations supported are:

- InsertRank: insert an element in a given position;
- Remove: remove an element.

Sequences are a basic form of data organization, and are typically used to realize and implement other data types and data structures.

2.2 Operations

Using locators (see Section 1.1), we can define a more complete repertory of operations for a sequence S:

SIZE(N) return the number of elements N of S;

HEAD(c) assign to c a locator to the first element of S; if S is empty, c is a null locator;

TAIL(c) assign to c a locator to the last element of S; if S is empty, a null locator is returned;

LOCATERANK(r, c) assign to c a locator to the r-th element of S; if $r < 1$ or $r > N$, where N is the size of S, c is a null locator;

PREV(c', c'') assign to c'' a locator to the element of S preceding the element with locator c'; if c' is the locator of the first element of S, c'' is a null locator;

NEXT(c', c'') assign to c'' a locator to the element of S following the element with locator c'; if c' is the locator of the last element of S, c'' is a null locator;

INSERTAFTER(e, c', c'') insert element e into S after the element with locator c', and return a locator c'' to e;

INSERTBEFORE(e, c', c'') insert element e into S before the element with locator c', and return a locator c'' to e;

INSERTHEAD(e, c) insert element e at the beginning of S, and return a locator c to e;

INSERTTAIL(e, c) insert element e at the end of S, and return a locator c to e;

INSERTRANK(e, r, c) insert element e in the r-th position of S; if $r < 1$ or $r > N + 1$, where N is the current size of S, c is a null locator;

REMOVE(c, e) remove from S and return element e with locator c;

MODIFY(c, e) replace with e the element with locator c.

Some of the above operations can be easily expressed by means of other operations of the repertory. For example, operations HEAD and TAIL can be easily expressed by means of LOCATERANK and SIZE.

2.3 Implementation with an Array

The simplest way to implement a sequence is to use a (one-dimensional) array, where the i-th element of the array stores the i-th element of the list, and to keep a variable that stores the size N of the sequence. With this implementation, accessing elements takes $O(1)$ time, while insertions and deletions take $O(N)$ time.

Table 1 shows the time complexity of the implementation of a sequence by means of an array.

Operation	**Time**
Size	$O(1)$
Head	$O(1)$
Tail	$O(1)$
LocateRank	$O(1)$
Prev	$O(1)$
Next	$O(1)$
InsertAfter	$O(N)$
InsertBefore	$O(N)$
InsertHead	$O(N)$
InsertTail	$O(1)$
InsertRank	$O(N)$
Remove	$O(N)$
Modify	$O(1)$

Table 1: Performance of a sequence implemented with an array. We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

2.4 Implementation with a Singly-Linked List

A sequence can also be implemented with a singly-linked list, where each element has a pointer to the next one. We also store the size of the sequence, and pointers to the first and last element of the sequence.

With this implementation, accessing elements takes $O(N)$ time, since we need to traverse the list, while while some insertions and deletions take $O(1)$ time.

Table 2 shows the time complexity of the implementation of sequence by means of singly-linked list.

Operation	Time
Size	$O(1)$
Head	$O(1)$
Tail	$O(1)$
LocateRank	$O(N)$
Prev	$O(N)$
Next	$O(1)$
InsertAfter	$O(1)$
InsertBefore	$O(N)$
InsertHead	$O(1)$
InsertTail	$O(1)$
InsertRank	$O(N)$
Remove	$O(N)$
Modify	$O(1)$

Table 2: Performance of a sequence implemented with a singly-linked list. We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

2.5 Implementation with a Doubly-Linked List

Better performance can be achieved, at the expense of using additional space, by implementing a sequence with a doubly-linked list, where each element has pointers to the next and previous elements. We also store the size of the sequence, and pointers to the first and last element of the sequence.

Table 3 shows the time complexity of the implementation of sequence by means of a doubly-linked list.

3 Priority Queue

3.1 Introduction

A *priority queue* is a container of elements from a totally ordered universe that supports the following two basic operations:

- Insert: insert an element into the priority queue;
- RemoveMax: remove the largest element from the priority queue.

Here are some simple applications of a priority queue:

Operation	Time
SIZE	$O(1)$
HEAD	$O(1)$
TAIL	$O(1)$
LOCATERANK	$O(N)$
PREV	$O(1)$
NEXT	$O(1)$
INSERTAFTER	$O(1)$
INSERTBEFORE	$O(1)$
INSERTHEAD	$O(1)$
INSERTTAIL	$O(1)$
INSERTRANK	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Table 3: Performance of a sequence implemented with a doubly-linked list. We denote with N the number of elements in the sequence at the time the operation is performed. The space complexity is $O(N)$.

Scheduling A scheduling system can store the tasks to be performed into a priority queue, and select the task with highest priority to be executed next.

Sorting To sort a set of N elements, we can insert them one at a time into a priority queue by means of N INSERT operations, and then retrieve them in decreasing order by means of N REMOVEMAX operations. This two-phase method is the paradigm of several popular sorting algorithms, including *Selection-Sort*, *Insertion-Sort*, and *Heap-Sort*.

3.2 Operations

Using locators, we can define a more complete repertory of operations for a priority queue Q:

SIZE(N) return the current number of elements N in Q;

MAX(c) return a locator c to the maximum element of Q;

INSERT(e, c) insert element e into Q and return a locator c to e;

REMOVE(c, e) remove from Q and return element e with locator c;

REMOVEMAX(e) remove from Q and return the maximum element e from Q;

MODIFY(c, e) replace with e the element with locator c.

Note that operation REMOVEMAX(e) is equivalent to MAX(c) followed by REMOVE(c, e).

3.3 Realization with a Sequence

We can realize a priority queue by reusing and extending the sequence abstract data type (see Section 2). Operations SIZE, MODIFY and REMOVE correspond to the homonymous sequence operations.

3.3.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation MAX can be performed by scanning the sequence with an iteration of NEXT operations, keeping track of the maximum element encountered. Finally, as observed above, operation REMOVEMAX is a combination of MAX and REMOVE. Table 4 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

Operation	Time
SIZE	$O(1)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
REMOVEMAX	$O(N)$
MODIFY	$O(1)$

Table 4: Performance of a priority queue realized by an unsorted sequence, implemented with a doubly-linked list. We denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

3.3.2 Sorted Sequence

An alternative implementation uses a sequence that is kept sorted. In this case, operation MAX corresponds to simply accessing the last element of the sequence. However, operation INSERT now requires scanning the sequence to find the appropriate position where to insert the new element. Table 5 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

Operation	Time
SIZE	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
REMOVEMAX	$O(1)$
MODIFY	$O(N)$

Table 5: Performance of a priority queue realized by a sorted sequence, implemented with a doubly-linked list. We denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

Realizing a priority queue with a sequence, sorted or unsorted, has the drawback that some operations require linear time in the worst case. Hence, this realization is not suitable in many applications where fast running times are sought for all the priority queue operations.

3.3.3 Sorting

For example, consider the sorting application (see Section 3.1). We have a collection of N elements from a totally ordered universe, and we want to sort them using a priority queue Q. We assume that each element uses $O(1)$ space, and any two elements can be compared in $O(1)$ time. If we realize Q with an unsorted sequence, then the first phase (inserting the N elements into Q) takes $O(N)$ time. However the second phase (removing N times the maximum element) takes time:

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2).$$

Hence, the overall time complexity is $O(N^2)$. This sorting method is known as *Selection-Sort*.

However, if we realize the priority queue with a sorted sequence, then the first phase takes time:

$$O\left(\sum_{i=1}^{N} i\right) = O(N^2),$$

while the second phase takes time $O(N)$. Again, the overall time complexity is $O(N^2)$. This sorting method is known as *Insertion-Sort*.

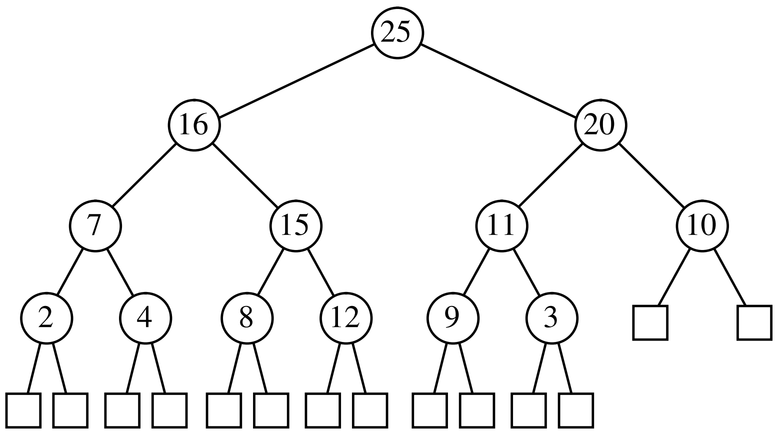


Figure 1: Example of a heap storing 13 elements.

3.4 Realization with a Heap

A more sophisticated realization of a priority queue uses a data structure called *heap*. A heap is a binary tree T whose internal nodes store each one element from a totally ordered universe, with the following properties (see Figure 1):

Level Property: all the levels of T are full, except possibly for the bottommost level, which is left-filled;

Partial Order Property: let μ be a node of T distinct from the root, and let ν be the parent of μ; then the element stored at μ is less than or equal to the element stored at ν.

The leaves of a heap do not store data and serve only as "placeholders". The level property implies that heap T is a minimum-height binary tree. More precisely, if T stores N elements and has height h, then each level i with $0 \leq i \leq h-2$ stores exactly 2^i elements, while level $h-1$ stores between 1 and 2^{h-1} elements. Note that level h contains only leaves. We have:

$$2^{h-1} = 1 + \sum_{i=0}^{h-2} 2^i \leq N \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1,$$

from which we obtain:

$$\log_2(N+1) \leq h \leq 1 + \log_2 N.$$

Now, we show how to perform the various priority queue operations by means of a heap T. We denote with $x(\mu)$ the element stored at an internal node μ of T.

We denote with ρ the root of T. We call *last node* of T the rightmost internal node of the bottommost internal level of T.

By storing a counter that keeps track of the current number of elements, SIZE consists of simply returning the value of the counter. By the partial order property, the maximum element is stored at the root, and hence operation MAX can be performed by accessing node ρ.

3.4.1 Operation INSERT

To insert an element e into T, we add a new internal node μ to T such that μ becomes the new last node of T, and set $x(\mu) = e$. This action ensures that the level property is satisfied, but may violate the partial-order property. Hence, if $\mu \neq \rho$, we compare $x(\mu)$ with $x(\nu)$, where ν is the parent of μ. If $x(\mu) > x(\nu)$, then we need to to restore the partial order property, which can be locally achieved by exchanging the elements stored at μ and ν. This causes the new element e to move up one level. Again, the partial order property may be violated, and we may have to continue moving up the new element e until no violation occurs. In the worst case, the new element e moves up to the root ρ of T by means of $O(\log N)$ exchanges. The upward movement of element e by means of exchanges is conventionally called *upheap*.

An example of a sequence of insertions into a heap is shown in Fig. 2.

3.4.2 Operation REMOVEMAX

To remove the maximum element, we cannot simply delete the root of T, because this would disrupt the binary tree structure. Instead, we access the last node λ of T, copy its element e to the root by setting $x(\rho) = x(\lambda)$, and delete λ. We have preserved the level property, but we may have violated the partial order property. Hence, if ρ has at least one nonleaf child, we compare $x(\rho)$ with the maximum element $x(\sigma)$ stored at a child of ρ. If $x(\rho) < x(\sigma)$, then we need to restore the partial order property, which can be locally achieved by exchanging the elements stored at ρ and σ. Again, the partial order property may be violated, and we continue moving down element e until no violation occurs. In the worst case, element e moves down to the bottom internal level of T by means of $O(\log N)$ exchanges. The downward movement of element e by means of exchanges is conventionally called *downheap*.

An example of operation REMOVEMAXin a heap is shown in Fig. 3.

3.4.3 Operation REMOVE

To remove an arbitrary element of heap T, we cannot simply delete its node μ, because this would disrupt the binary tree structure. Instead, we proceed as before and delete the last node of T after copying to μ its element e. We have preserved the level property, but we may have violated the partial order property, which can be restored by performing either upheap or downheap.

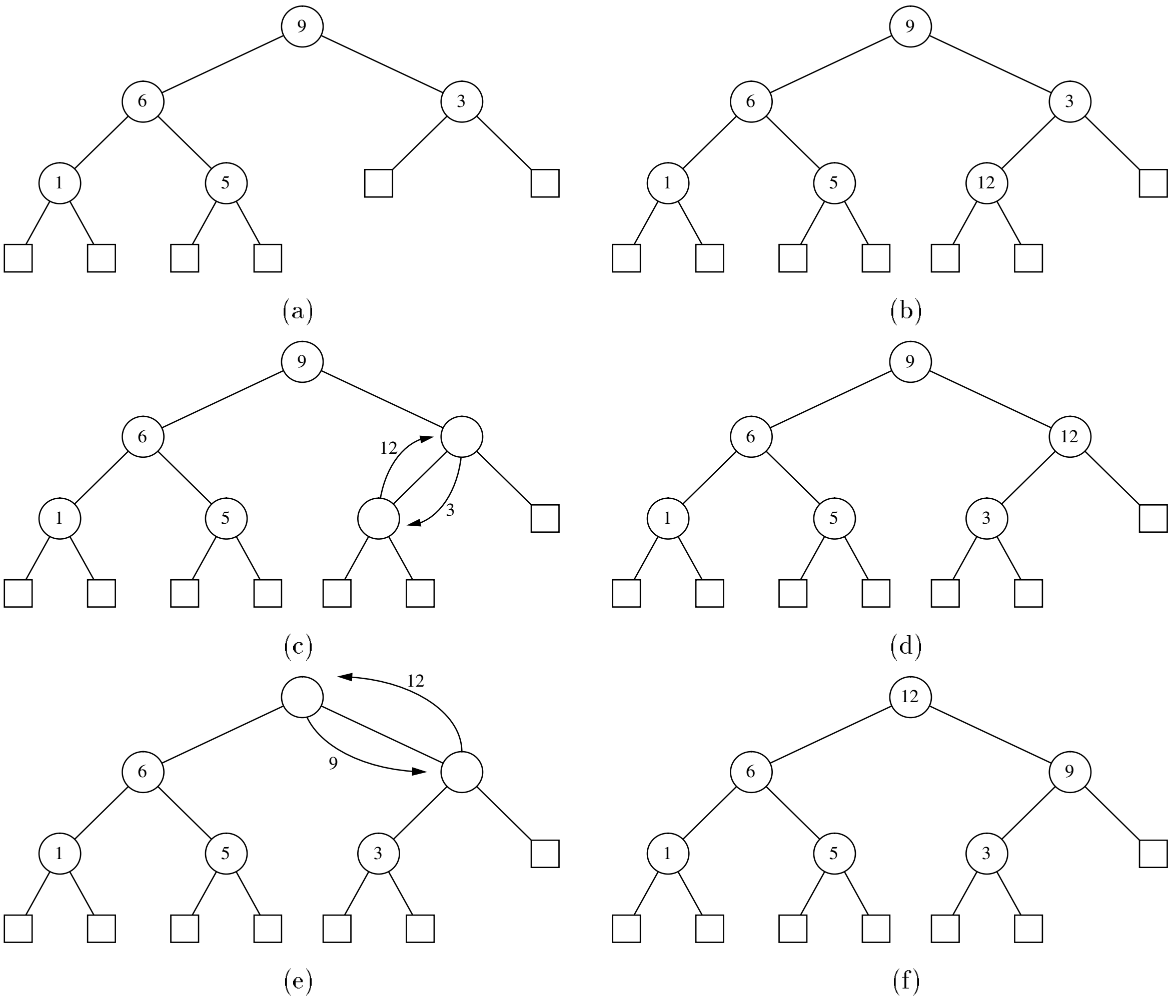


Figure 2: Example of insertion into a heap.

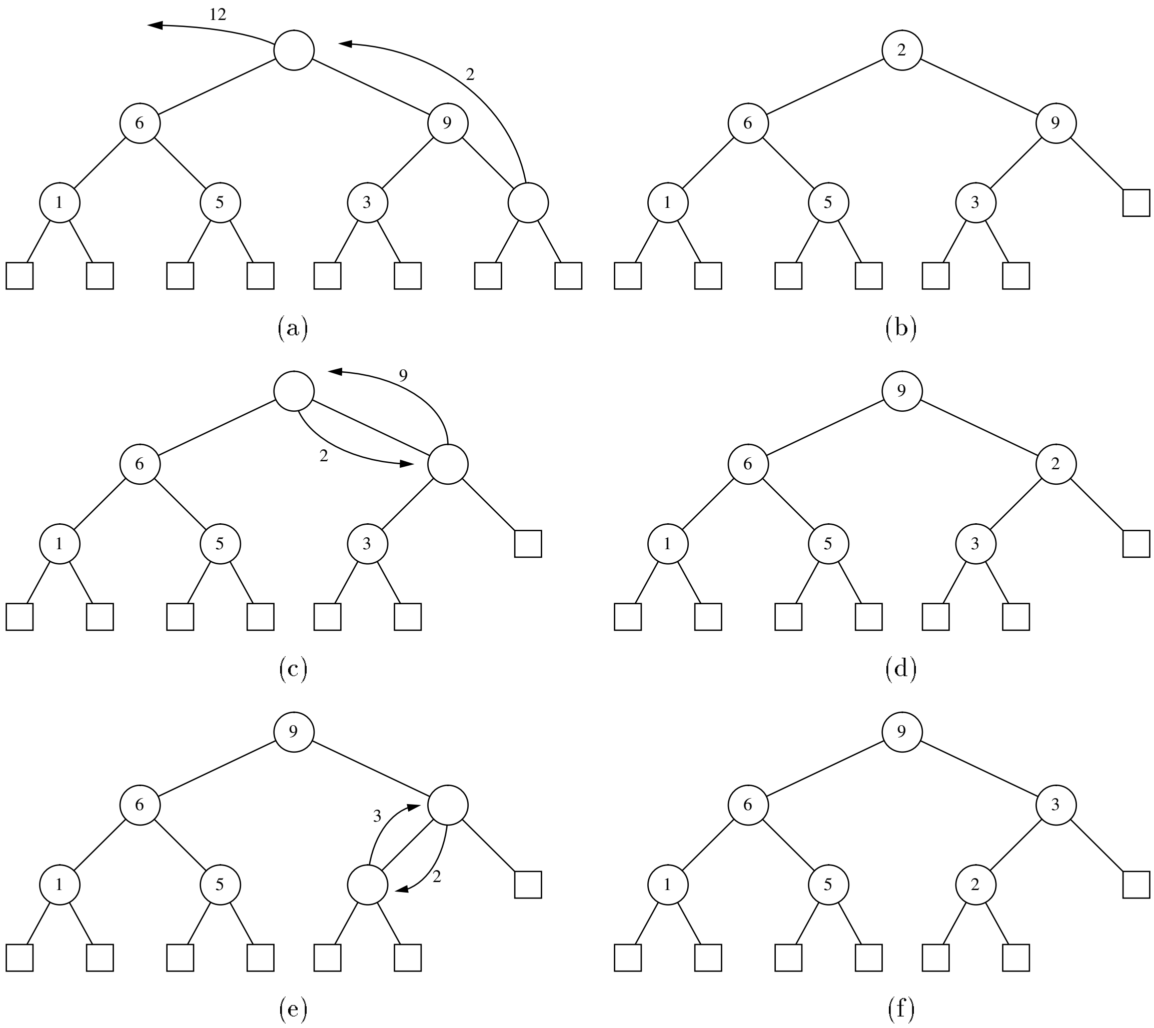


Figure 3: RemoveMax operation in a heap.

Finally, after modifying an element of heap T, if the partial order property is violated, we just need to perform either upheap or downheap.

3.4.4 Time Complexity

Table 6 shows the time complexity of the realization of a priority queue by means of a heap. We assume that the heap is itself realized by a data structure for binary trees that supports $O(1)$-time access to the children and parent of a node. For instance, we can implement the heap explicitly with a linked structure (with pointers from a node to its parents and children), or implicitly with an array (where node i has children $2i$ and $2i+1$).

Operation	**Time**
Size	$O(1)$
Max	$O(1)$
Insert	$O(\log N)$
Remove	$O(\log N)$
RemoveMax	$O(\log N)$
Modify	$O(\log N)$

Table 6: Performance of a priority queue realized by a heap, implemented with a suitable binary tree data structure. We denote with N the number of elements in the priority queue at the time the operation is performed. The space complexity is $O(N)$.

Let N the number of elements in a priority queue Q realized with a heap T at the time an operation is performed. The time bounds of Table 6 are based on the following facts:

- in the worst case, the time complexity of upheap and downheap is proportional to the height of T;
- if we keep a pointer to the last node of T, we can update this pointer in time proportional to the height of T in operations Insert, Remove, and RemoveMax, as illustrated in Fig. 4.
- the height of heap T is $O(\log N)$;

The $O(N)$ space complexity bound for the heap is based on the following facts:

- the heap has $2N+1$ nodes (N internal nodes and $N+1$ leaves);
- every node uses $O(1)$ space;

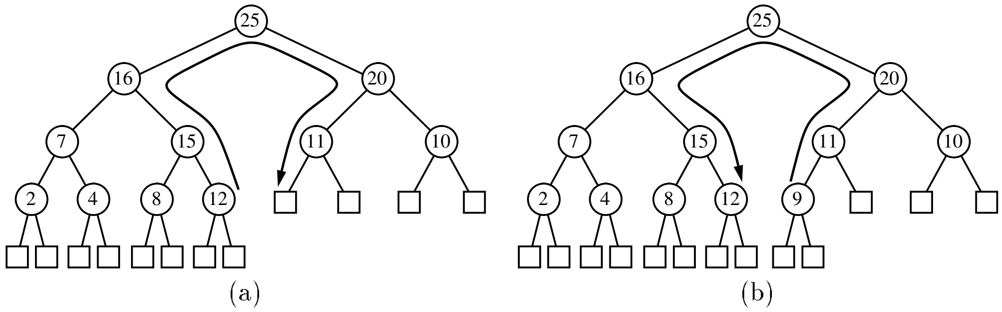


Figure 4: Update of the pointer to the last node: (a) INSERT; (b) REMOVE or REMOVEMAX.

- in the array implementation, because of the level property the array elements used to store heap nodes are in the contiguous locations 1 through $2N-1$.

Note that we can reduce the space requirement by a constant factor implementing the leaves of the heap with null objects, such that only the internal nodes have space associated with them.

3.4.5 Sorting

Realizing a priority queue with a heap has the advantage that all the operations take $O(\log N)$ time, where N is the number of elements in the priority queue at the time the operation is performed. For example, in the sorting application (see Section 3.1), both the first phase (inserting the N elements) and the second phase (removing N times the maximum element) take time:

$$O\left(\sum_{i=1}^{N} \log i\right) = O(N \log N).$$

Hence, sorting with a priority queue realized with a heap takes $O(N \log N)$ time. This sorting method is known as *Heap-Sort*, and its performance is considerably better than that of Selection-Sort and Insertion-Sort (see Section 3.3.3), where the priority queue is realized as a sequence.

3.5 Realization with a Dictionary

A priority queue can be easily realized with a dictionary (see Section 4). Indeed, all the operations in the priority queue repertory are supported by a dictionary. To achieve $O(1)$ time for operation MAX, we can store the locator of the maximum element in a variable, and recompute it after an an update operations. This realization of a priority queue with a dictionary has the same asymptotic complexity bounds as the realization with a heap, provided the dictionary is suitably implemented, e.g., with an (a, b)-tree (see Section 4.4) or an AVL-tree (see Section 4.5). However, a heap is simpler to program than an (a, b)-tree or an AVL-tree.

4 Dictionary

A *dictionary* is a container of elements from a totally ordered universe that supports the following basic operations:

- FIND: search for an element;
- INSERT: insert an element;
- REMOVE: delete an element;

A major application of dictionaries are database systems.

4.1 Operations

In the most general setting, the elements stored in a dictionary are pairs (x, y), where x is the *key* giving the ordering of the elements, and y is the auxiliary information. For example, in a database storing student records, the key could be the student's last name, and the auxiliary information the student's transcript. It is convenient to augment the ordered universe of keys with two *special keys*: $+\infty$ and $-\infty$, and assume that each dictionary has, in addition to its *regular elements*, two *special elements*, with keys $+\infty$ and $-\infty$, respectively. For simplicity, we shall also assume that no two elements of a dictionary have the same key. An insertion of an element with the same key as that of an existing element will be rejected by returning a null locator.

Using locators (see Section 1.1), we can define a more complete repertory of operations for a dictionary D:

SIZE(N) return the number of regular elements N of D;

FIND(x, c) if D contains an element with key x, assign to c a locator to such an element, otherwise set c equal to a null locator;

LOCATEPREV(x, c) assign to c a locator to the element of D with the largest key less than or equal to x; if x is smaller than all the keys of the regular elements, c is a locator the special element with key $-\infty$; if $x = -\infty$, c is a null locator;

LOCATENEXT(x, c) assign to c a locator to the element of D with the smallest key greater than or equal to x; if x is larger than all the keys of the regular elements, c is a locator to the special element with key $+\infty$; if $x = +\infty$, c is a null locator;

LOCATERANK(r, c) assign to c a locator to the r-th element of D; if $r < 1$, c is a locator to the special element with key $-\infty$; if $r > N$, where N is the size of D, c is a locator to the special element with key $+\infty$;

PREV(c', c'') assign to c'' a locator to the element of D with the largest key less than that of the element with locator c'; if the key of the element with locator c' is smaller than all the keys of the regular elements, this operation returns a locator to the special element with key $-\infty$;

NEXT(c', c'') assign to c'' a locator to the element of D with the smallest key larger than that of the element with locator c'; if the key of the element with locator c' is larger than all the keys of the regular elements, this operation returns a locator to the special element with key $+\infty$;

MIN(c) assign to c a locator to the regular element of D with minimum key; if D has no regular elements, c is a null locator;

MAX(c) assign to c a locator to the regular element of D with maximum key; if D has no regular elements, c is null a locator;

INSERT(e, c) insert element e into D, and return a locator c to e; if there is already an element with the same key as e, this operation returns a null locator;

REMOVE(c, e) remove from D and return element e with locator c;

MODIFY(c, e) replace with e the element with locator c.

Some of the above operations can be easily expressed by means of other operations of the repertory. For example, operation FIND is a simple variation of LOCATEPREV or LOCATENEXT; MIN and MAX are special cases of LOCATERANK, or can be expressed by means of PREV and NEXT.

4.2 Realization with a Sequence

We can realize a dictionary by reusing and extending the sequence abstract data type (see Section 2). Operations SIZE, INSERT and REMOVE correspond to the homonymous sequence operations.

4.2.1 Unsorted Sequence

We can realize INSERT by an INSERTHEAD or an INSERTTAIL, which means that the sequence is not kept sorted. Operation FIND(x, c) can be performed by scanning the sequence with an iteration of NEXT operations, until we either find an element with key x, or we reach the end of the sequence. Table 7 shows the time complexity of this realization, assuming that the sequence is implemented with a doubly-linked list.

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
LOCATERANK	$O(N)$
NEXT	$O(N)$
PREV	$O(N)$
MIN	$O(N)$
MAX	$O(N)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Table 7: Performance of a dictionary realized by an unsorted sequence, implemented with a doubly-linked list. We denote with N the number of elements in the dictionary at the time the operation is performed.

4.2.2 Sorted Sequence

We can also use a sorted sequence to realize a dictionary. Operation INSERT now requires scanning the sequence to find the appropriate position where to insert the new element. However, in a FIND operation, we can stop scanning the sequence as soon as we find an element with a key larger than the search key. Table 8 shows the time complexity of this realization by a sorted sequence, assuming that the sequence is implemented with a doubly-linked list.

4.2.3 Sorted Array

We can obtain a different performance tradeoff by implementing the sorted sequence by means of an array, which allows constant-time access to any element of the sequence given its position. Indeed, with this realization we can speed up operation FIND(x, c) using the *binary search* strategy, as follows. If the dictionary is empty, we are done. Otherwise, let N be the current number of elements in the dictionary. We compare the search key k with the key x_m of the middle element of the sequence, i.e., the element at position $\lfloor N/2 \rfloor$. If $x = x_m$, we have found the element. Else, we recursively search in the subsequence of the elements preceding the middle element if $x < x_m$, or following the middle element if $x > x_m$. At each recursive call, the number of elements of the subsequence being searched halves. Hence, the number of sequence elements accessed and the number of comparisons performed by binary search is $O(\log N)$. While searching takes $O(\log N)$ time, inserting or deleting elements now takes

Operation	Time
SIZE	$O(1)$
FIND	$O(N)$
LOCATEPREV	$O(N)$
LOCATENEXT	$O(N)$
LOCATERANK	$O(N)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(1)$
MODIFY	$O(N)$

Table 8: Performance of a dictionary realized by a sorted sequence, implemented with a doubly-linked list. We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

$O(N)$ time.

Table 9 shows the performance of a dictionary realized with a sorted sequence, implemented with an array.

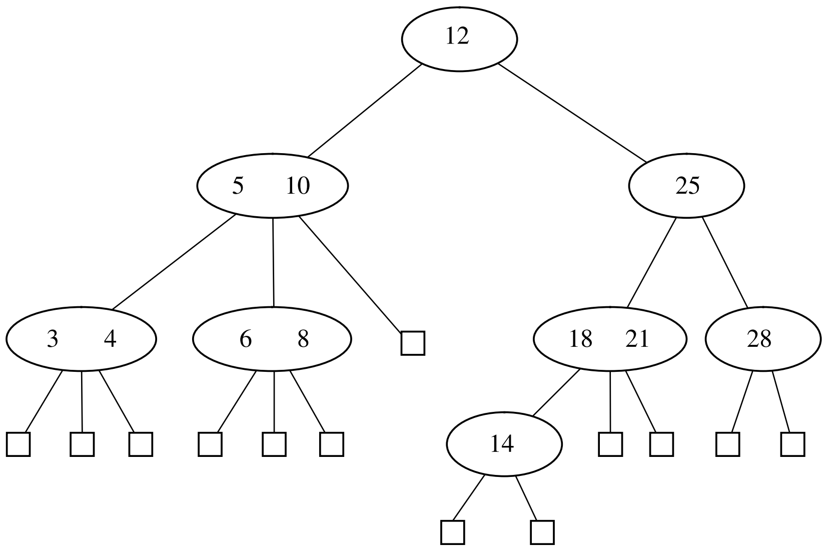
4.3 Realization with a Search Tree

A *search tree* for elements of the type (x, y), where x is a key from a totally ordered universe, is a rooted ordered tree T such that:

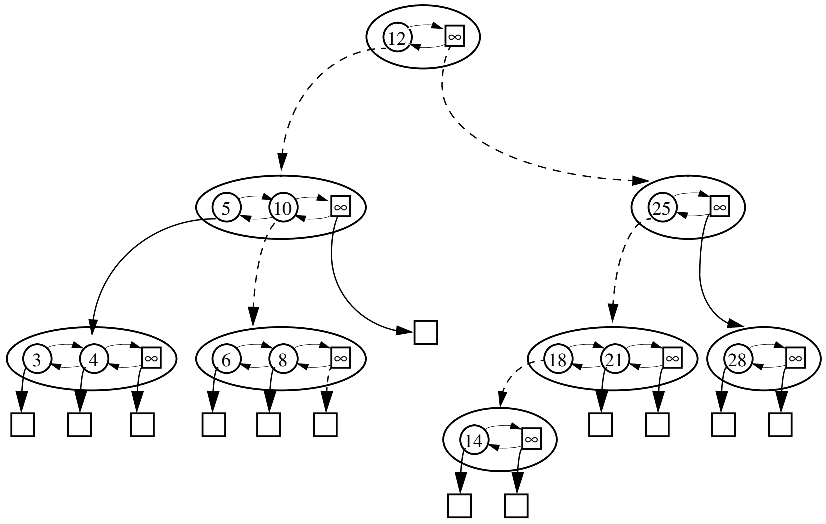
- each internal node of T has at least two children and stores a nonempty set of elements;
- a node μ of T with d children $\mu_1, \cdots, \mu_d$ stores $d-1$ elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$, where $x_1 \leq \cdots \leq x_{d-1}$;
- for each element (x, y) stored at a node in the subtree of T rooted at μ_i, we have $x_{i-1} \leq x \leq x_i$, where $x_0 = -\infty$ and $x_d = +\infty$.

In a search tree, each internal node stores a nonempty collection of keys, while the leaves do not store any key and serve only as "placeholders". An example of search tree is shown in Fig. 5.a. A special type of search tree is a *binary search tree*, where each internal node stores one key and has two children.

We will recursively describe the realization of a dictionary D by means of a search tree T, since we will use dictionaries to implement the nodes of



(a)



(b)

Figure 5: Realization of a dictionary by means of a search tree: (a) A search tree T. (b) Realization of the dictionaries at the nodes of T by means of sorted sequences. The search paths for elements 9 (unsuccessful search) and 14 (successful search) are shown with dashed lines.

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
LOCATERANK	$O(1)$
NEXT	$O(1)$
PREV	$O(1)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(N)$
REMOVE	$O(N)$
MODIFY	$O(N)$

Table 9: Performance of a dictionary realized by a sorted sequence, implemented with an array. We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

T. Namely, an internal node μ of T with children $\mu_1, \cdots, \mu_d$ and elements $(x_1, y_1) \cdots (x_{d-1}, y_{d-1})$ is equipped with a dictionary $D(\mu)$ whose regular elements are the pairs $(x_i, (y_i, \mu_i))$, $i = 1, \cdots, d-1$ and whose special element with key $+\infty$ is $(+\infty, (\cdot, \mu_d))$. A regular element (x, y) stored in D is associated with a regular element $(x, (y, \nu))$ stored in a dictionary $D(\mu)$, for some node μ of T. See the example in Fig. 5.b.

4.3.1 Operation FIND

Operation FIND(x, c) on dictionary D is performed by means of the following recursive method for a node μ of T, where μ is initially the root of T (see Fig. 5.b). We execute LOCATENEXT(x, c') on dictionary $D(\mu)$ and let $(x', (y', \nu))$ be the element pointed by the returned locator c'. We have three cases:

- $x = x'$: we have found x and return locator c to (x', y');
- $x \neq x'$ and ν is a leaf: we have determined that x is not in D and return a null locator c;
- $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

4.3.2 **Operation** Insert

Operations LocatePrev, LocateNext and Insert can be performed with small variations of the above method. For example, to perform operation Insert(e, c), where $e = (x, y)$, we modify the above cases as follows (see Fig. 6):

- $x = x'$: an element with key x already exists, and we return a null locator;
- $x \neq x'$ and ν is a leaf: we create a new leaf node λ, insert a new element $(x, (y, \lambda))$ into $D(\mu)$, and return a locator c to (x, y).
- $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

Note that new elements are inserted at the "bottom" of the search tree.

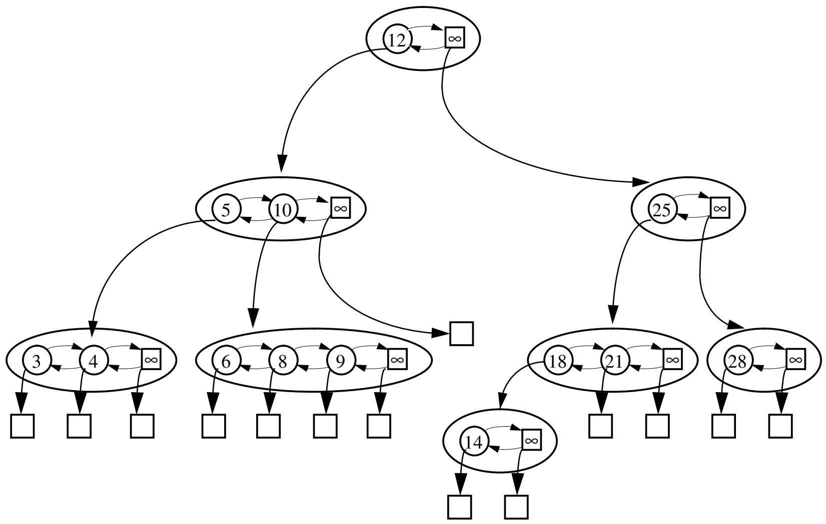


Figure 6: Insertion of element 9 into the search tree of Fig. 5.

4.3.3 **Operation** Remove

Operation Remove(e, c) is more complex (see Fig. 7). Let the associated element of $e = (x, y)$ in T be $(x, (y, \nu))$, stored in dictionary $D(\mu)$ of node μ.

- If node ν is a leaf, we simply delete element $(x, (y, \nu))$ from $D(\mu)$.
- Else (ν is an internal node), we find the successor element $(x', (y', \nu'))$ of $(x, (y, \nu))$ in $D(\mu)$ with a Next operation in $D(\mu)$.

- If ν' is a leaf, we replace ν' with ν, i.e., change element $(x', (y', \nu'))$ to $(x', (y', \nu))$, and delete element $(x, (y, \nu))$ from $D(\mu)$.
- Else (ν' is an internal node), while the leftmost child ν'' of ν' is not a leaf, we set $\nu' = \nu''$. Let $(x'', (y'', \nu''))$ be the first element of $D(\nu')$ (node ν'' is a leaf). We replace $(x, (y, \nu))$ with $(x'', (y'', \nu))$ in $D(\mu)$ and delete $(x'', (y'', \nu''))$ from $D(\nu')$.

The above actions may cause dictionary $D(\mu)$ or $D(\nu')$ to become empty. If this happens, say for $D(\mu)$ and μ is not the root of T, we need to remove node μ. Let $(+\infty, (\cdot, \kappa))$ be the special element of $D(\mu)$ with key $+\infty$, and let $(z, (w, \mu)$ be the element pointing to μ in the parent node π of μ. We delete node μ and replace $(z, (w, \mu)$ with $(z, (w, \kappa))$ in $D(\pi)$.

Note that, if we start with an initially empty dictionary, a sequence of insertions and deletions performed with the above methods yields a search tree with a single node. In the next sections, we show how to avoid this behavior by imposing additional conditions on the structure of a search tree.

4.4 Realization with an (a, b)-Tree

An (a, b)-*tree*, where a and b are integer constants such that $2 \leq a \leq (b + 1)/2$, is a a search tree T with the following additional restrictions:

Level Property: all the levels of T are full, i.e., all the leaves are at the same depth;

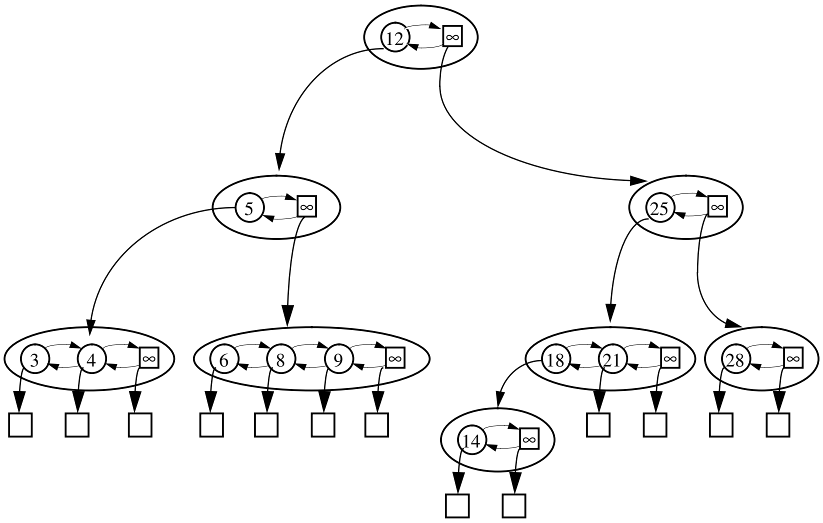
Size Property: let μ be an internal node of T, and d be the number of children of μ; if μ is the root of T, then $d \geq 2$, else $a \leq d \leq b$;

The height of an (a, b) tree storing N elements is $O(\log_a N) = O(\log N)$. Indeed, in the worst case, the root has two children, and all the other internal nodes have a children.

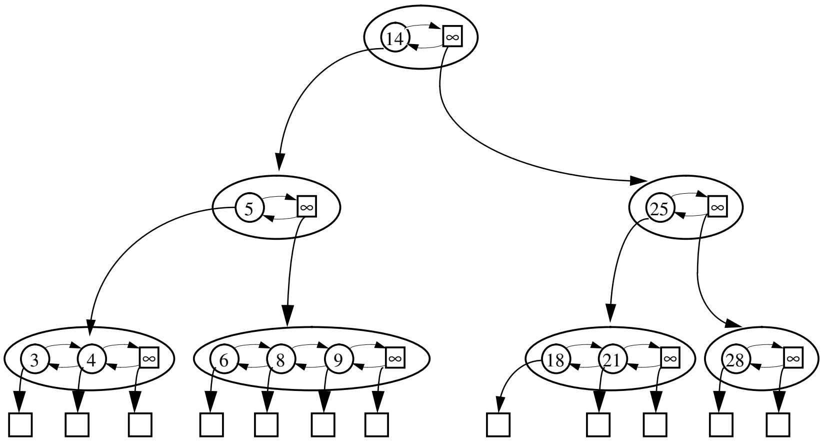
The realization of a dictionary with an (a, b)-tree extends that with a search tree. Namely, the implementation of operations Insert and Remove need to be modified in order to preserve the level and size properties. Also, we maintain the current size of the dictionary, and pointers to the minimum and maximum regular elements of the dictionary.

4.4.1 Insertion

The implementation of operation Insert for search trees given in Section 4.3.2 adds a new element to the dictionary $D(\mu)$ of an existing node μ of T. Since the structure of the tree is not changed, the level property is satisfied. However, if $D(\mu)$ had the maximum allowed size $b - 1$ before insertion (recall that the size of $D(\mu)$ is one less than the number of children of μ), the size property is violated at μ because $D(\mu)$ has now size b. To remedy this *overflow* situation, we perform the following *node-split* (see Fig. 8):



(a)



(b)

Figure 7: (a) Deletion of element 10 from the search tree of Fig. 6. (b) Deletion of element 12 from the search tree of part a.

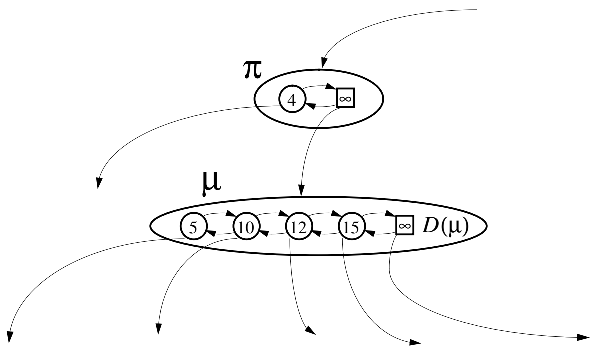
- Let the special element of $D(\mu)$ be $(+\infty, (\cdot, \mu_{b+1}))$. Find the median element of $D(\mu)$, i.e., the element $e_i = (x_i, (y_i, \mu_i))$ such that $i = \lceil (b+1)/2 \rceil$).
- Split $D(\mu)$ into:
  - dictionary D', containing the $\lceil (b-1)/2 \rceil$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = 1 \cdots i-1$ and the special element $(+\infty, (\cdot, \mu_i))$;
  - element e; and
  - dictionary D'', containing the $\lfloor (b-1)/2 \rfloor$ regular elements $e_j = (x_j, (y_j, \mu_j))$, $j = i+1 \cdots b$ and the special element $(+\infty, (\cdot, \mu_{b+1}))$.
- Create a new tree node κ, and set $D(\kappa) = D'$. Hence, node κ has children $\mu_1 \cdots \mu_i$.
- Set $D(\mu) = D''$. Hence, node μ has children $\mu_{i+1} \cdots \mu_{b+1}$.
- If μ is the root of T, create a new node π with an empty dictionary $D(\pi)$. Else, let π be the parent of μ.
- Insert element $(x_i, (y_i, \kappa))$ into dictionary $D(\pi)$.

After a node-split, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π has $b+1$ children, we repeat the node-split for $\mu = \pi$. Each time we perform a node-split, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the INSERT operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

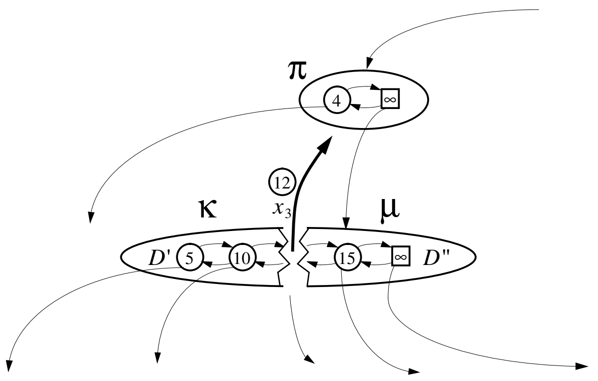
4.4.2 Deletion

The implementation of operation REMOVE for search trees given in Section 4.3.3 removes an element from the dictionary $D(\mu)$ of an existing node μ of T. Since the structure of the tree is not changed, the level property is satisfied. However, if μ is not the root, and $D(\mu)$ had the minimum allowed size $a-1$ before deletion (recall that the size of the dictionary is one less than the number of children of the node), the size property is violated at μ because $D(\mu)$ has now size $a-2$. To remedy this *underflow* situation, we perform the following *node-merge* (see Figs. 9–10):

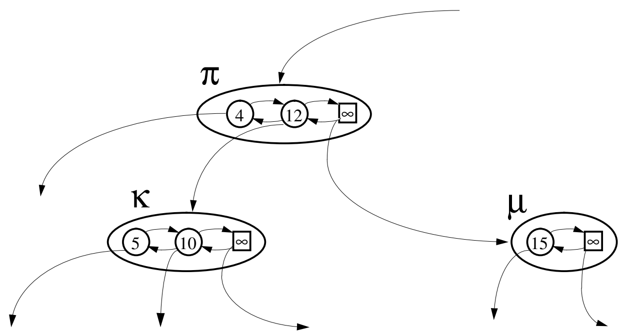
- If μ has a right sibling, let μ'' be the right sibling of μ and $\mu' = \mu$; else, let μ' be the left sibling of μ and $\mu'' = \mu$. Let $(+\infty, (\cdot, \nu))$ be the special element of $D(\mu')$.
- Let π be the parent of μ' and μ''. Remove from $D(\pi)$ the regular element $(x, (y, \mu'))$ associated with μ'.



(a)



(b)



(c)

Figure 8: Example of node-split in a 2-4 tree: (a) initial configuration with an overflow at node μ; (b) split of the node μ into μ' and μ'' and insertion of the median element into the parent node π; (c) final configuration.

- Create a new dictionary D containing the regular elements of $D(\mu')$ and $D(\mu'')$, regular element $(x, (y, \nu))$, and the special element of $D(\mu'')$.
- Set $D(\mu'') = D$, and destroy node μ'.
- If μ'' has more than b children, perform a node-split at μ''.

After a node-merge, the level property is still verified. Also, the size property is verified for all the nodes of T, except possibly for node π. If π is the root and has one child (and thus an empty dictionary), we remove node π. If π is not the root and has fewer than $a - 1$ children, we repeat the node-merge for $\mu = \pi$. Each time we perform a node-merge, the possible violation of the size property appears at a higher level in the tree. This guarantees the termination of the algorithm for the REMOVE operation. We omit the description of the simple method for updating the pointers to the minimum and maximum regular elements.

4.4.3 Complexity

Let T be an (a, b)-tree storing N elements. The height of T is $O(\log_a N) = O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. We assume that the dictionaries at the nodes of T are realized with sequences. Hence, processing a node takes $O(b) = O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 10 shows the performance of a dictionary realized with an (a, b)-tree.

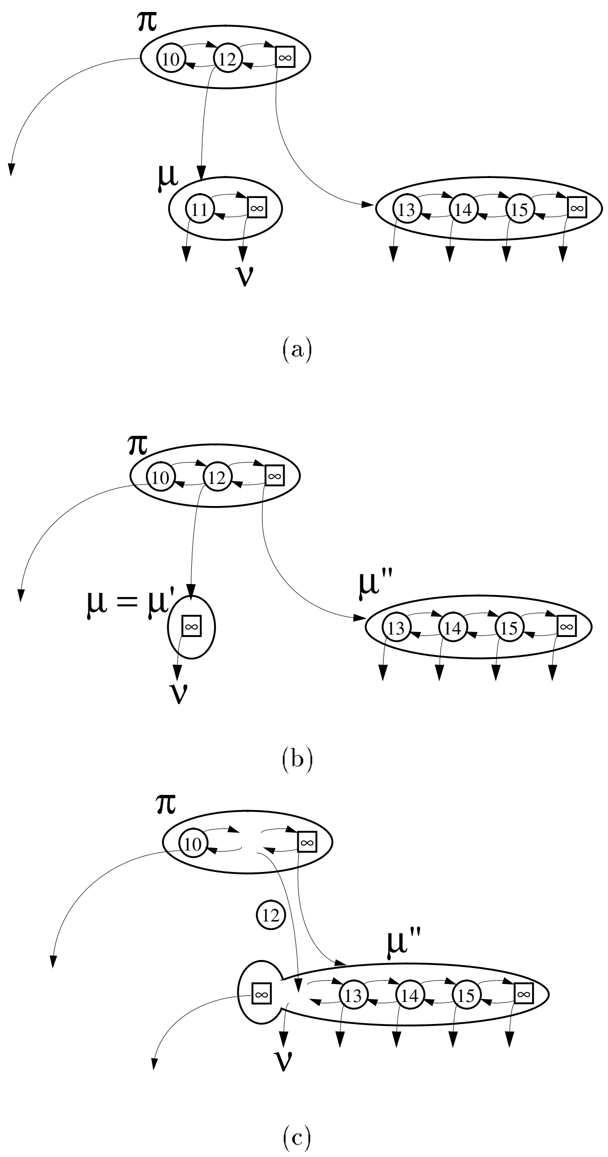


Figure 9: Example of node merge in a 2-4 tree: (a) initial configuration; (b) the removal of an element from dictionary $D(\mu)$ cauases an underflow at node μ; (c) merging node $\mu = \mu'$ into its sibling μ''.

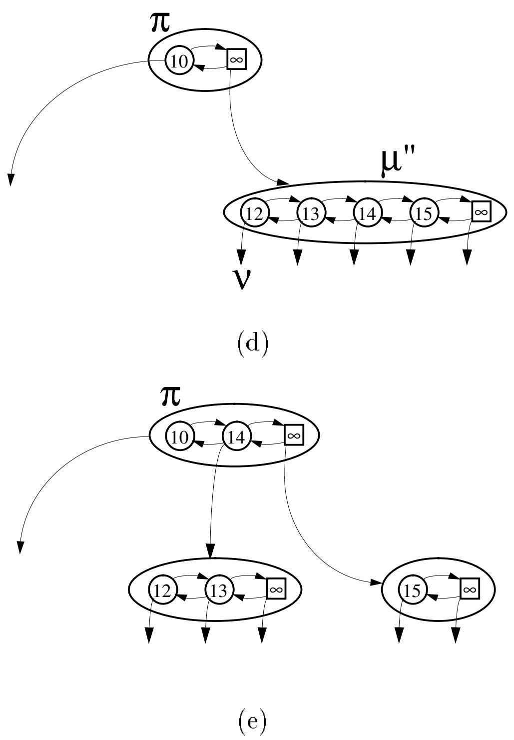


Figure 10: Example of node merge in a 2-4 tree: (d) overflow at node μ''; (e) final configuration after splitting node μ''.

Operation	Time
Size	$O(1)$
Find	$O(\log N)$
LocatePrev	$O(\log N)$
LocateNext	$O(\log N)$
LocateRank	$O(\log N)$
Next	$O(\log N)$
Prev	$O(\log N)$
Min	$O(1)$
Max	$O(1)$
Insert	$O(\log N)$
Remove	$O(\log N)$
Modify	$O(\log N)$

Table 10: Performance of a dictionary realized by an (a, b)-tree. We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.5 Realization with an AVL-tree

An *AVL-tree* is a search tree T with the following additional restrictions:

Binary Property: T is a binary tree, i.e., every internal node has two children, (left and right child), and stores one key.

Height-Balance Property: For every internal node μ, the heights of the subtrees rooted at the children of μ differ at most by one.

An example of AVL-tree is shown in Fig. 11. The height of an AVL-tree storing N elements is $O(\log N)$. This can be shown as follows. Let N_h be the minimum number of elements stored in an AVL-tree of height h. We have $N_0 = 0$, $N_1 = 1$, and

$$N_h = 1 + N_{h-1} + N_{h-2}, \text{ for } h \geq 2.$$

The above recurrence relation defines the well-known Fibonacci numbers. Hence $N_h = \Omega(\phi^N)$, where $1 < \phi < 2$.

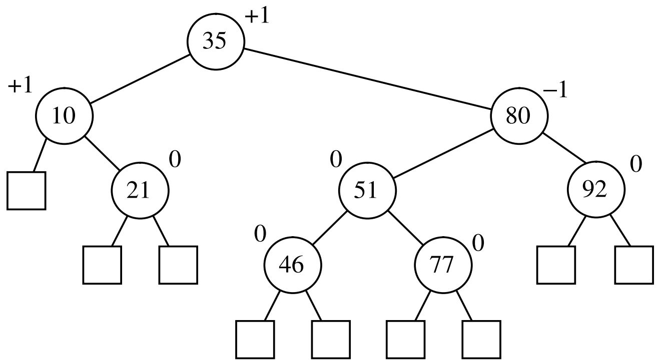


Figure 11: Example of AVL-tree storing 9 elements. The keys are shown inside the nodes, and the balance factors (see Section 4.5.2) are shown next to the nodes.

The realization of a dictionary with an AVL-tree extends that with a search tree. Namely, the implementation of operations INSERT and REMOVE need to be modified in order to preserve the binary and height-balance properties after an insertion or deletion.

4.5.1 Insertion

The implementation of INSERT for search trees given in Section 4.3.2 adds the new element to an existing node. This violates the binary property, and hence cannot be done in an AVL-tree. Hence, we modify the three cases of the INSERT algorithm for search trees as follows:

- $x = x'$: an element with key x already exists, and we return a null locator c;
- $x \neq x'$ and ν is a leaf: we replace ν with a new internal node κ with two leaf children, store element (x, y) in κ, and return a locator c to (x, y).
- $x \neq x'$ and ν is an internal node: we set $\mu = \nu$ and recursively execute the method.

We have preserved the binary property. However, we may have violated the height-balance property, since the heights of some subtrees of T have increased by one. We say that a node is balanced if the difference between the heights of its subtrees is -1, 0, or 1, and is unbalanced otherwise. The unbalanced nodes form a (possibly empty) subpath of the the path from the new internal node κ to the root of T. See the example of Fig. 12.

4.5.2 Rebalancing

To restore the height-balance property, we *rebalance* the lowest node μ that is unbalanced, as follows.

- Let μ' be the child of μ whose subtree has maximum height, and μ'' be the child of μ' whose subtree has maximum height.
- Let (μ_1, μ_2, μ_3) be the left-to right ordering of nodes $\{\mu, \mu', \mu''\}$, and (T_0, T_1, T_2, T_3) be the left-to-right ordering of the four subtrees of $\{\mu, \mu', \mu''\}$ not rooted at a node in $\{\mu, \mu', \mu''\}$.
- replace the subtree rooted at μ with a new subtree rooted at μ_2, where μ_1 is the left child of μ_2 and has subtrees T_0 and T_1, and μ_3 is the right child of μ_2 and has subtrees T_2 and T_3.

Two examples of rebalancing are schematically shown in Fig. 14. Other symmetric configurations are possible. In Fig. 13, we show the rebalancing for the tree of Fig. 12.

Note that the rebalancing causes all the nodes in the subtree of μ_2 to become balanced. Also, the subtree rooted at μ_2 now has the same height as the subtree rooted at node μ before insertion. This causes all the previously unbalanced nodes to become balanced. To keep track of the nodes that become unbalanced, we can store at each node a *balance factor*, which is the difference of the heights of the left and right subtrees. A node becomes unbalanced when its balance factor becomes $+2$ or -2. It is easy to modify the algorithm for operation INSERT such that it maintains the balance factors of the nodes.

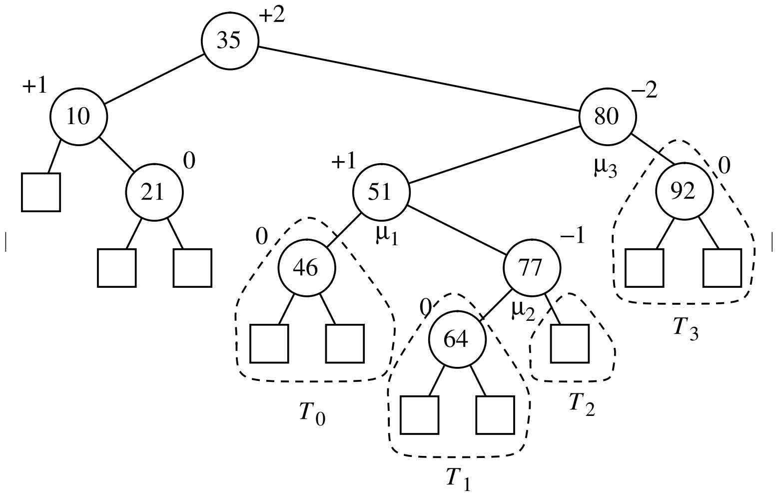


Figure 12: Insertion of an element with key 64 into the AVL-tree of Fig. 11. Note that two nodes (with balance factors +2 and −2) have become unbalanced. The dashed lines identify the subtrees that participate in the reabalancing, as illustrated in Fig. 14.

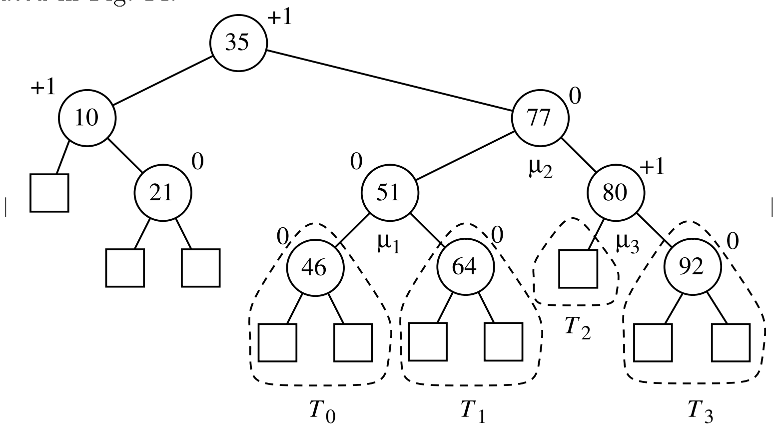


Figure 13: AVL-tree obtained by rebalancing the lowest unbalanced node in the tree of Fig. 11. Note that all the nodes are now balanced. The dashed lines identify the subtrees that participate in the reabalancing, as illustrated in Fig. 14.

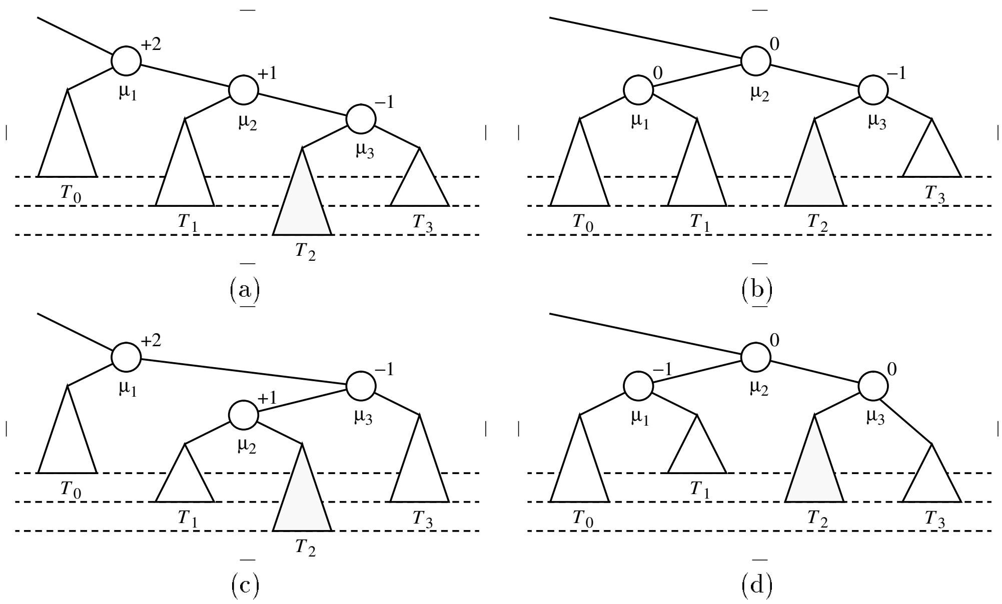


Figure 14: Schematic illustration of rebalancing a node in the INSERT algorithm for AVL-trees. The shaded subtree is the one where the new element was inserted. (a–b) Rebalancing by means of a "single rotation". (c–d) Rebalancing by means of a "double rotation".

4.5.3 Deletion

The implementation of REMOVE for search trees given in Section 4.3 preserves the binary property, but may cause the height-balance property to be violated. After deleting a node, there can be only one unbalanced node, on the path from the deleted node to the root of T.

To restore the height-balance property, we *rebalance* the unbalanced node using the above algorithm. Notice, however, that the choice of μ'' may not be unique, since the subtrees of μ' may have the same height. In this case, the height of the subtree rooted at μ_2 is the same as the height of the subtree rooted at μ before rebalancing, and we are done. If instead the subtrees of μ' do not have the same height, then the height of the subtree rooted at μ_2 is one less than the height of the subtree rooted at μ before rebalancing. This may cause an ancestor of μ_2 to become unbalanced, and we repeat the rebalancing step. Balance factors are used to keep track of the nodes that become unbalanced, and can be easily maintained by the REMOVE algorithm.

4.5.4 Complexity

Let T be an AVL-tree storing N elements. The height of T is $O(\log N)$. Each dictionary operation affects only the nodes along a root-to-leaf path. Rebalancing a node takes $O(1)$ time. We conclude that each operation takes $O(\log N)$ time.

Table 11 shows the performance of a dictionary realized with an AVL-tree.

Operation	Time
SIZE	$O(1)$
FIND	$O(\log N)$
LOCATEPREV	$O(\log N)$
LOCATENEXT	$O(\log N)$
LOCATERANK	$O(\log N)$
NEXT	$O(\log N)$
PREV	$O(\log N)$
MIN	$O(1)$
MAX	$O(1)$
INSERT	$O(\log N)$
REMOVE	$O(\log N)$
MODIFY	$O(\log N)$

Table 11: Performance of a dictionary realized by an AVL-tree. We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N)$.

4.6 Realization with a Hash Table

The previous realizations of a dictionary make no assumptions on the structure of the keys, and use comparisons between keys to guide the execution of the various operations.

4.6.1 Bucket Array

If the keys of a dictionary D are integers in the range $[1, M]$, we can implement D with a *bucket array* B. An element (x, y) of D is represented by setting $B[x] = y$. If an integer x is not in D, the location $B[x]$ stores a null value. In this implementation, we allocate a "bucket" for every possible element of D.

Table 12 shows the performance of a dictionary realized a bucket array.

The bucket array method can be extended to keys that are easily mapped to integers. E.g., three-letter airport codes can be mapped to the integers in the range $[1, 26^3]$.

Operation	Time
SIZE	$O(1)$
FIND	$O(1)$
LOCATEPREV	$O(M)$
LOCATENEXT	$O(M)$
LOCATERANK	$O(M)$
NEXT	$O(M)$
PREV	$O(M)$
MIN	$O(M)$
MAX	$O(M)$
INSERT	$O(1)$
REMOVE	$O(1)$
MODIFY	$O(1)$

Table 12: Performance of a dictionary realized by bucket array. The keys in the dictionary are integers in the range $[1, M]$. The space complexity is $O(M)$.

4.6.2 Hashing

The bucket array method works well when the range of keys is small. However, it is inefficient when the range of keys is large. To overcome this problem, we can use a *hash function* h that maps the keys of the original dictionary D into integers in the range $[1, M]$, where M is a parameter of the hash function. Now, we can apply the bucket array method using the *hashed value* $h(x)$ of the keys. In general, a *collision* may happen, where two distinct keys x_1 and x_2 have the same hashed value, i.e.,$x_1 \neq x_2$ and $h(x_1) = h(x_2)$. Hence, each bucket must be able to accommodate a collection of elements.

A hash table of size M for a function $h(x)$ is a bucket array B of size M (primary structure) whose entries are dictionaries (secondary structures), such that element (x, y) is stored in the dictionary $B[h(x)]$. For simplicity of programming, the dictionaries used as secondary structures are typically realized with sequences. An example of hash table is show in Fig. 15.

If all the elements in the dictionary D collide, they are all stored in the same dictionary of the bucket array, and the performance of the hash table is the same as that of the kind of dictionary used as a secondary structures. At the other end of the spectrum, if no two elements of the dictionary D collide, they are stored in distinct one-element dictionaries of the bucket array, and the performance of the hash table is the same as that of a bucket array.

A typical hash function for integer keys is $h(x) = x \bmod M$. The size M of the hash table is usually chosen as a prime number. An example of hash table is shown in Fig. 15.

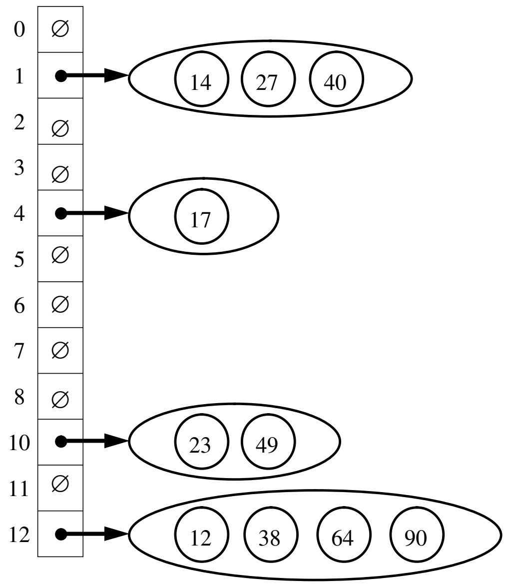


Figure 15: Example of hash table of size 13 storing 10 elements. The hash function is $h(x) = x \bmod 13$.

It is interesting to analyze the performance of a hash table from a probabilistic viewpoint. If we assume that the hashed values of the keys are uniformly distributed in the range $[1, M]$, then each bucket holds on average N/M keys, where N is the size of the dictionary. Hence, when $N = O(M)$, the average size of the secondary data structures is $O(1)$.

Table 13 shows the performance of a dictionary realized a hash table. Both the worst-case and average time complexity in the above probabilistic model are indicated.

5 Defining Terms

(a,b)-tree search tree with additional properties (each node has between a and b children, and all the levels are full); see Section 4.4.

abstract data type mathematically specified data type equipped with operations that can be performed on the objects; see Section 1.2.

AVL-tree binary search tree such that the subtrees of each node have heights that differ by at most one; see Section 4.5.

binary search tree search tree such that each internal node has two children; see Section 4.3.

	Time	
Operation	Worst-Case	Average
SIZE	$O(1)$	$O(1)$
FIND	$O(N)$	$O(N/M)$
LOCATEPREV	$O(N+M)$	$O(N+M)$
LOCATENEXT	$O(N+M)$	$O(N+M)$
LOCATERANK	$O(N+M)$	$O(N+M)$
NEXT	$O(N+M)$	$O(N+M)$
PREV	$O(N+M)$	$O(N+M)$
MIN	$O(N+M)$	$O(N+M)$
MAX	$O(N+M)$	$O(N+M)$
INSERT	$O(1)$	$O(1)$
REMOVE	$O(1)$	$O(1)$
MODIFY	$O(1)$	$O(1)$

Table 13: Performance of a dictionary realized by a hash table of size M. We denote with N the number of elements in the dictionary at the time the operation is performed. The space complexity is $O(N+M)$. The average time complexity refers to a probabilistic model where the hashed values of the keys are uniformly distributed in the range $[1, M]$.

bucket array implementation of a dictionary by means of an array indexed by the keys of the dictionary elements; see Section 4.6.1.

container abstract data type storing a collection of objects (elements); see Section 1.1.

dictionary container storing elements from a sorted universe supporting searches, insertions, and deletions; see Section 4.

hash table implementation of a dictionary by means of a bucket array storing secondary dictionaries; see Section 4.6.2.

heap binary tree with additional properties storing the elements of a priority queue; see Section 3.4.

locator variable that allows to access an object stored in a container; see Section 1.1.

priority queue container storing elements from a sorted universe supporting finding the maximum element, insertions, and deletions; see Section 3.

search tree rooted ordered tree with additional properties storing the elements of a dictionary; see Section 4.3.

sequence container storing object in a certain order, supporting insertions (in a given position) and deletions; see Section 2.

6 Further Information

Many textbooks and monographs have been written on data structures, e.g., [2, 7, 16, 18, 19, 20, 21, 22, 26, 28, 32, 34, 29, 36].

Recent papers surveying the state-of-the art in data structures include [4, 15, 24, 35].

The LEDA project [23] aims at developing a C++ library of efficient and reliable implementations of sophisticated data structures.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[3] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[4] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, Sept. 1992.

[5] R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13:245–265, 1995.

[6] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[8] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *Automata, Languages and Programming (Proc. 17th ICALP)*, volume 442 of *Lecture Notes in Computer Science*, pages 598–611, 1990.

[9] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[10] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Heidelberg, West Germany, 1987.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification: A technique for speeding up dynamic graph algorithms. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 60–69, 1992.

[12] S. Even. *Graph Algorithms.* Computer Science Press, Potomac, Maryland, 1979.

[13] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

[14] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. 4th ACM-SIAM Symp. Discrete Algorithms*, pages 175–184, 1993.

[15] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[16] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison Wesley, 1991.

[17] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Inform. Control*, 68:170–184, 1986.

[18] E. Horowitz, S. Sahni, and D. Metha. *Fundamentals of Data Structures in C++.* Computer Science Press, 1995.

[19] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1968.

[20] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[21] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms.* Harper Collins, 1991.

[22] K. Mehlhorn. *Data Structures and Algorithms.* Springer-Verlag, 1984. Volumes 1–3.

[23] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995. `http://www.mpi-sb.mpg.de/guide/staff/uhrig/leda.html`.

[24] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science.* Elsevier, Amsterdam, 1990.

[25] P. B. Miltersen, S. Sairam, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoret. Comput. Sci.*, 130:203–236, 1994.

[26] J. Nievergelt and K. H. Hinrichs. *Algorithms and Data Structures: With Applications to Graphics and Geometry.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[27] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[28] M. H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science.* Springer-Verlag, 1983.

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

[30] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 35:668–676, 1990.

[31] J. H. Reif. A topological approach to dynamic graph connectivity. *Inform. Process. Lett.*, 25:65–70, 1987.

[32] R. Sedgewick. *Algorithms in C++.* Addison Wesley, Reading, MA, 1992.

[33] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.

[34] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Mathematics, 1983.

[35] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 431–524. Elsevier, Amsterdam, 1990.

[36] D. Wood. *Data Structures, Algorithms, and Performance.* Addison Wesley, 1993.