Algorithms and Data Structures I.
Lecture Notes

Tibor Ásványi
Department of Computer Science
Eötvös Loránd University, Budapest
asvanyi@inf.elte.hu

February 15, 2024

Contents

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms (Third Edition), *The MIT Press*, 2009.

[2] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[3] Narashima Karumanchi, Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[4] Neapolitan, Richard E., Foundations of algorithms (Fifth edition), *Jones & Bartlett Learning*, 2015. ISBN 978-1-284-04919-0 (pbk.)

[5] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++ (Fourth Edition), *Pearson*, 2014. `http://aszt.inf.elte.hu/~asvanyi/ds/DataStructuresAndAlgorithmAnalysisInCpp_2`

[6] Wirth, N., Algorithms and Data Structures, *Prentice-Hall Inc.*, 1976, 1985, 2004. `http://aszt.inf.elte.hu/~asvanyi/ds/AD.pdf`

[7] Burch, Carl, B+ trees http://aszt.inf.elte.hu/~asvanyi/ds/B+trees.zip

1 Notations and Basic Notions

$\mathbb{N} = \{0; 1; 2; 3; \dots\}$ = natural numbers.
$\mathbb{Z} = \{\cdots -3; -2, -1; 0; 1; 2; 3; \dots\}$ = integer numbers
$\mathbb{R}$ = real numbers
$\mathbb{P}$ = positive real numbers
$\mathbb{P}_0$ = nonnegative real numbers

$$\log n = \begin{cases} \log_2 n & \text{if} \quad n > 0 \\ 0 & \text{if} \quad n = 0 \end{cases}$$

$half(n) = \lfloor \frac{n}{2} \rfloor$, where $n \in \mathbb{N}$
$Half(n) = \lceil \frac{n}{2} \rceil$, where $n \in \mathbb{N}$

We use the structure diagram notation in our pseudo codes. We use a UML-like notation with some C++/Java/Pascal flavour in the structure diagrams. Main points:

1. A program is made up of declarations. A structure diagram represents each of them, but we do not care about their order.

2. We can declare subroutines, classes, variables and constants.

3. The subroutines (i.e., subprograms) are the functions, the procedures (i.e., void functions), and the methods of the classes.

4. **while** loop is the default loop ; **for** loops are also used.

5. The operators $\neq, \geq, \leq$ are written by the usual mathematical notation. The assignment statements and the "is equal to" comparisons are written in Pascal style. For example, $x := y$ assigns the value of y to x, and $x = y$ checks their equality.

6. Only scalar parameters can be passed by value (and it is their default). Non-scalar parameters are always passed by reference. For example, an object cannot be passed by value, just by reference.

7. An assignment statement can copy a scalar value or a subarray of scalar values like $A[u..v] := B[p..q]$ where $v - u = q - p$. $A[u..v] := x$ means that each element of $A[u..v]$ is initialized with x.

8. In the definitions of classes, we use a simple UML notation. The name of the class comes in the first part. The data members are in the second part, and the methods, if any, are in the third part. A "$-$" sign prefixes a private declaration/specification. A "$+$" sign prefixes a public declaration/specification. We use neither templates nor inheritance (nor protected members/methods).

9. We do not use libraries.

10. We do not use exception handling.

1.1 Time complexities of algorithms

Time complexity (operational complexity) of an algorithm reflects some abstract time. We count **the subroutine calls + the number of loop iterations** during the program's run. (This measure is approximately proportional to the actual runtime of the program, and we can omit constant factors here because they are helpful only if we know the programming environment [for example, (the speed of) the hardware, the operation system, the compiler etc.]).

We typically calculate the time complexity of an algorithm as a function of the size of the input data structure(s) (for example, the length of the input array). We distinguish $MT(n)$ (Maximum Time), $AT(n)$ (Average or expected Time), and $mT(n)$ (minimum Time). Clearly $MT(n) \geq AT(n) \geq mT(n)$. If $MT(n) = mT(n)$, then we can speak of a general time complexity $T(n)$ where $T(n) = MT(n) = AT(n) = mT(n)$.

Typically, the time complexities of the algorithms are not calculated precisely. Only we calculate their *asymptotic order* or make *asymptotic estimation(s)* using the *big-O* notation.

1.2 The *big-O* notation

Definition 1.1 *Given* $g : \mathbb{N} \to \mathbb{R}$ *;*
g is asymptotically positive (AP), **iff**
there exists an $N \in \mathbb{N}$ *so that* $g(n) > 0$ *for each* $n \geq N$ *.*

In this chapter, we suppose that f, g, h denote *asymptotically positive* functions of type $\mathbb{N} \to \mathbb{R}$ in each case because they represent time (or space) complexity functions and these satisfy this property. Usually, it is not easy to give such a function exactly. We make estimations.

- When we make an *asymptotic upper estimate* g of f, then we say that "f is big-O g", and write $f \in O(g)$. Informally, $f \in O(g)$ means that function f is at most proportional to g. ($f(n) \leq d * g(n)$ for some $d > 0$, if n is large enough.)

- When we make an *asymptotic lower estimate* g of f, then we say that "f is Omega g", and write $f \in \Omega(g)$. Informally $f \in \Omega(g)$ means that function f is at least proportional to g. ($f(n) \geq c * g(n)$ for some $c > 0$, if n is large enough.)

- When we make an *asymptotic upper and lower estimate* g of f, then we say that "f is Theta g", and write $f \in \Theta(g)$ which means that $f \in O(g) \wedge f \in \Omega(g)$. In this case, we also say that the *asymptotic order* of f is $\Theta(g)$.

Definition 1.2

$$f \prec g \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

"$f \prec g$" is read as "f is asymptotically less than *g". It can also be written as "$f \in o(g)$" which means that*

$$o(g) = \{h \,|\, h \prec g\}$$

Definition 1.3

$$g \succ f \iff f \prec g$$

"$g \succ f$" is read as "g is asymptotically greater than *f". It can also be written as "$f \in \omega(g)$" which means that*

$$\omega(g) = \{h \mid h \succ g\}$$

Definition 1.4 $O(g) = \{f \mid$ *there exist positive constants* d *and* n_0
. *such that* $f(n) \leq d * g(n)$ *for all* $n \geq n_0\}$

("$f \in O(g)$" can be read as "f is at most proportional to g".)

Definition 1.5 $\Omega(g) = \{f \mid$ *there exist positive constants* c *and* n_0
. *such that* $f(n) \geq c * g(n)$ *for all* $n \geq n_0\}$

("$f \in \Omega(g)$" can be read as "f is at least proportional to g".)

Definition 1.6

$$\Theta(g) = O(g) \cap \Omega(g)$$

("$f \in \Theta(g)$" can be read as "f is roughly proportional to g".)

Consequence 1.7 .
$f \in O(g) \iff \exists d, n_0 > 0,$ *and* $\psi : \mathbb{N} \to \mathbb{R}$ *so that* $\lim_{n\to\infty} \frac{\psi(n)}{g(n)} = 0,$ *and*

$$f(n) \leq d * g(n) + \psi(n)$$

for each $n \geq n_0$.

Consequence 1.8 .
$f \in \Omega(g) \iff \exists c, n_0 > 0,$ *and* $\varphi : \mathbb{N} \to \mathbb{R}$ *so that* $\lim_{n\to\infty} \frac{\varphi(n)}{g(n)} = 0$ *and*

$$c * g(n) + \varphi(n) \leq f(n)$$

for each $n \geq n_0$.

Consequence 1.9 $f \in \Theta(g) \iff f \in O(g) \wedge f \in \Omega(g)$.

Note 1.10 .

- *If $f \in O(g)$, we can say that g is an asymptotic upper bound of f.*
- *If $f \in \Omega(g)$, we can say that g is an asymptotic lower bound of f.*

- *If $f \in \Theta(g)$, we can say that f and g are asymptotically equivalent.*
- *We never use the popular notations "$f = O(g)$", "$f = \Omega(g)$" or "$f = \Theta(g)$". We think one should not use it for at least two reasons.*
  1. *$O(g)$, $\Omega(g)$ and $\Theta(g)$ are sets of asymptotically positive (AP) functions while f is just a single AP function. Thus, the equalities above are simply false claims.*
  2. *Suppose one is aware that, for example, "$f = O(g)$" means $f \in O(g)$. Still "$f = O(g)$" and "$h = O(g)$" does not imply $f = h$. Similarly, according to this popular notation "$1 = \Theta(1) \wedge 2 = \Theta(1) \wedge 1 \neq 2$", etc.*

Theorem 1.11

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f \prec g \Longrightarrow f \in O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{P} \Longrightarrow f \in \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f \succ g \Longrightarrow f \in \Omega(g)$$

Proof. The first and last statements follow directly from the definition of the $\prec$ and $\succ$ relations. In order to prove the middle one, consider that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$. Consequently, if n is sufficiently large, $\left|\frac{f(n)}{g(n)} - c\right| < \frac{c}{2}$. Thus

$$\frac{c}{2} < \frac{f(n)}{g(n)} < 2c$$

Because g is AP, $g(n) > 0$ for sufficiently large n values can multiply with it both sides of this inequality. Therefore

$$\frac{c}{2} * g(n) < f(n) < 2c * g(n)$$

As a result, $f \in \Theta(g)$ $\square$

Consequence 1.12

$$k \in \mathbb{N} \wedge a_0, a_1, \ldots, a_k \in \mathbb{R} \wedge a_k > 0 \Longrightarrow a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \in \Theta(n^k)$$

Proof.

$$\lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0}{n^k} =$$

$$\lim_{n\to\infty} \left(\frac{a_k n^k}{n^k} + \frac{a_{k-1} n^{k-1}}{n^k} + \cdots + \frac{a_1 n}{n^k} + \frac{a_0}{n^k} \right) =$$

$$\lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) =$$

$$\lim_{n\to\infty} a_k + \lim_{n\to\infty} \frac{a_{k-1}}{n} + \cdots + \lim_{n\to\infty} \frac{a_1}{n^{k-1}} + \lim_{n\to\infty} \frac{a_0}{n^k} =$$

$$a_k + 0 + \cdots + 0 + 0 = a_k \in \mathbb{P} \Longrightarrow$$

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \in \Theta(n^k)$$

□

Property 1.13 *(Relations of the different classes of functions)*

$$\Theta(g) = O(g) \cap \Omega(g)$$

$$o(g) \subsetneqq O(g) \setminus \Omega(g)$$

$$\omega(g) \subsetneqq \Omega(g) \setminus O(g)$$

Example of the asymptotic order of AP functions:
$\log n \prec n \prec n \log n \prec n^2 \prec n^2 \log n \prec n^3$

Property 1.14 *(Transitivity)*

$$f \in O(g) \wedge g \in O(h) \Longrightarrow f \in O(h)$$

$$f \in \Omega(g) \wedge g \in \Omega(h) \Longrightarrow f \in \Omega(h)$$

$$f \in \Theta(g) \wedge g \in \Theta(h) \Longrightarrow f \in \Theta(h)$$

$$f \prec g \wedge g \prec h \Longrightarrow f \prec h$$

$$f \succ g \wedge g \succ h \Longrightarrow f \succ h$$

Property 1.15 *(Symmetry)*

$$f \in \Theta(g) \iff g \in \Theta(f)$$

Property 1.16 *(Exchanged symmetry)*

$$f \in O(g) \iff g \in \Omega(f)$$

$$f \prec g \iff g \succ f$$

Property 1.17 *(Asymmetry)*

$$f \prec g \Longrightarrow \neg(g \prec f)$$

$$f \succ g \Longrightarrow \neg(g \succ f)$$

Property 1.18 *(Reflexivity)*

$$f \in O(f) \wedge f \in \Omega(f) \wedge f \in \Theta(f)$$

Consequence 1.19 *($\Longrightarrow$: 1.15, 1.14.3 ; $\Longleftarrow$: 1.18.3)*

$$f \in \Theta(g) \iff \Theta(f) = \Theta(g)$$

Property 1.20 *(Relations $\prec$ and $\succ$ are irreflexive.)*

$$\neg(f \prec f)$$

$$\neg(f \succ f)$$

Consequence 1.21 *Since the binary relation $\cdot \in \Theta(\cdot)$ is reflexive, symmetric and transitive, it gives a classification of the set of asymptotically positive functions, where f and g belong to the same equivalence class,* **iff** $f \in \Theta(g)$. *In this case, function f is* asymptotically equivalent *to function g.*

We will see that such equivalence classes can be identified, and they will be fundamental from the point of view of calculating the efficiency of the algorithms. For example, we have already seen that any k-degree polynomial with a major positive coefficient is *asymptotically equivalent* to the n^k function. (See Consequence 1.12.) The asymptotic order of such equivalence classes can be established, and it is based on the following property.

Property 1.22

$$f_1, g_1 \in \Theta(h_1) \wedge f_2, g_2 \in \Theta(h_2), \wedge f_1 \prec f_2 \Longrightarrow g_1 \prec g_2$$

Definition 1.23

$$\Theta(f) \prec \Theta(g) \iff f \prec g$$

Lemma 1.24 Sometimes the following so called **L'Hospital rule** can be applied for computing limes $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ in Theorem 1.11.

If the real extensions of the functions f and g are differentiable for sufficiently large substitution values and

$$\lim_{n\to\infty} f(n) = \infty \wedge \lim_{n\to\infty} g(n) = \infty \wedge \exists \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \Longrightarrow$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Property 1.25 *(Based on Theorem 1.11 and on Lemma 1.24.)*

$$c, d \in \mathbb{R} \wedge c < d \Longrightarrow n^c \prec n^d$$

$$c, d \in \mathbb{P}_0 \wedge c < d \Longrightarrow c^n \prec d^n$$

$$c, d \in \mathbb{R} \wedge d > 1 \Longrightarrow n^c \prec d^n$$

$$d \in \mathbb{P}_0 \Longrightarrow d^n \prec n! \prec n^n$$

$$c, d \in \mathbb{P} \wedge c, d > 1 \Longrightarrow \log_c n \in \Theta(\log_d n)$$

$$\varepsilon \in \mathbb{P} \Longrightarrow \log n \prec n^\varepsilon$$

$$c \in \mathbb{R} \wedge \varepsilon \in \mathbb{P} \Longrightarrow n^c \log n \prec n^{c+\varepsilon}$$

Proof. In order to prove that $\varepsilon \in \mathbb{P} \Longrightarrow \log n \prec n^\varepsilon$, we need the L'Hospital rule, i.e. Lemma 1.24.

$$\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} = \log e \lim_{n\to\infty} \frac{\ln n}{n^\varepsilon} = \log e \lim_{n\to\infty} \frac{\ln' n}{(n^\varepsilon)'} = \frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{\frac{1}{n}}{n^{\varepsilon-1}} =$$

$$= \frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{1}{n^\varepsilon} = \frac{\log e}{\varepsilon} 0 = 0$$

□

Consequence 1.26

$$\Theta(\log n) \prec \Theta(n) \prec \Theta(n * \log n) \prec \Theta(n^2) \prec \Theta(n^2 * \log n) \prec \Theta(n^3)$$

Property 1.27
(Function classes $O(\cdot), \Omega(\cdot), \Theta(\cdot), o(\cdot), \omega(\cdot)$ are closed for the following ops.)

$$f \in O(g) \wedge c \in \mathbb{P} \Longrightarrow c * f \in O(g)$$

$$f \in O(h_1) \wedge g \in O(h_2) \Longrightarrow f + g \in O(h_1 + h_2)$$

$$f \in O(h_1) \wedge g \in O(h_2) \Longrightarrow f * g \in O(h_1 * h_2)$$

$$f \in O(g) \wedge \varphi : \mathbb{N} \to \mathbb{R} \wedge \lim_{n\to\infty} \frac{\varphi(n)}{f(n)} = 0 \Longrightarrow f + \varphi \in O(g)$$

(Similarly for function classes $\Omega(\cdot), \Theta(\cdot), o(\cdot), \omega(\cdot)$.)

Property 1.28

$$max(f, g) \in \Theta(f + g) \text{ where } max(f, g)(n) = max(f(n), g(n)) \ (n \in \mathbb{N})$$

Proof. Because f and g are AP, $f(n) > 0$ and $g(n) > 0$ for sufficiently large $n \in \mathbb{N}$ values, and

$$max(f(n), g(n)) = \frac{f(n) + g(n)}{2} + \frac{|f(n) - g(n)|}{2} \quad \wedge$$

$$\frac{f(n) + g(n)}{2} + \frac{|f(n) - g(n)|}{2} \leq \frac{f(n) + g(n)}{2} + \frac{f(n) + g(n)}{2} = f(n) + g(n)$$

$$\wedge \quad \frac{f(n) + g(n)}{2} + \frac{|f(n) - g(n)|}{2} \geq \frac{1}{2} * (f(n) + g(n))$$

$$\implies \frac{1}{2} * (f(n) + g(n)) \leq max(f(n), g(n)) \leq 1 * (f(n) + g(n))$$

$$\implies max(f, g) \in \Theta(f + g)$$

□

1.3 Note on time complexity measures

We measure each algorithm's efficiency by counting the *subroutine calls + iterations*. (The nonrecursive calls may be omitted.) Counting the *key comparisons* also works in case of comparison sorts (like insertion sort, merge sort, heap sort, quicksort, etc.) because the two measures have the same asymptotic order. But counting the *subroutine calls + iterations* is more general because it can be applied to each algorithm. Thus, we use this one. Otherwise, we should find a good running time measure for each algorithm that is not a comparison sort. Therefore, we use this universal measure of time complexity: we count the *subroutine calls + iterations* for all the algorithms.

1.4 Space complexities of algorithms

Space complexity of an algorithm is an abstract measure of the algorithm's space (i.e. memory) requirements.

It depends on the depth of the subroutine calls and the sizes of the data structures (like arrays, linked lists, trees, etc.) generated or needed by that algorithm. The size of the input is typically excluded from the space requirements.

We typically calculate the space complexity of an algorithm as a function of the size of the input data structure(s) (for example, the length of the input array).

Even if the input size is the same, an algorithm's different runs may need different amounts of space. Thus, we distinguish $MS(n)$ (Maximum Space complexity), $AS(n)$ (Average or expected Space complexity), and $mS(n)$ (minimum Space complexity). Clearly $MS(n) \geq AS(n) \geq mS(n)$. If $MS(n) = mS(n)$, then we can speak of a general space complexity $S(n)$ where $S(n) = MS(n) = AS(N) = mS(n)$.

Typically, the space complexities of the algorithms are not calculated precisely. Only we calculate their *asymptotic order* or make *asymptotic estimation(s)* using the *big-O* notation.

Clearly, for any algorithm, $mS(n) \in \Omega(1)$. Thus, provided that $MS(n) \in O(1)$, it follows that $MS(n), AS(n), mS(n) \in \Theta(1)$.

Definition 1.29 *An algorithm works* in-place,
iff $MS(n) \in O(1)$ *is satisfied for it.*

Definition 1.30 *An algorithm works* non-strictly in-place,
iff $MS(n) \in O(\log n)$ *is satisfied for it.*

Notice that this later definition is based on the fact that although $\lim_{n\to\infty} \log(n) = \infty$, still function $\log(n)$ grows very-very slowly. For example $\log_2 10^3 < 10$, $\log_2 10^6 < 20$, $\log_2 10^9 < 30$, $\log_2 10^{12} < 40$.

2 Elementary Data Structures and Data Types

A **data structure** (DS) is a way to store and organise data to facilitate access and modifications. No single data structure works well for all purposes, so it is essential to know the strengths and limitations of several of them ([1] 1.1).

A **data type** (DT) is a **data structure** + its **operations**.

An **abstract data type** (ADT) is a mathematical or informal structure + its operations described mathematically or informally.

A *representation* of an ADT is an appropriate DS.

An *implementation* of an ADT is some program code of its operations.

[1] Chapter 10; [6] 4.1 - 4.4.2; [3] 3-5

2.1 Arrays, memory allocation and deallocation

The most common data type is the **array**. It is a finite sequence of data. Its data elements can be accessed and updated directly and efficiently through indexing. Most programming languages support arrays. We can access any element of an array in $\Theta(1)$ time.

In this book, arrays must be declared in the following way.

$$A, Z : \mathcal{T}[n]$$

In this example, A and Z are two arrays of element type $\mathcal{T}$ and of size n.

In our model, the array data structure is an *array object* containing its size and elements, and it is accessed through a so-called *array pointer* containing its memory address. The operations of the array can
– read its size like $A.length$ and $Z.length$: $A.length = Z.length = n$ here,
– access (read and write) its elements through indexing as usual, for example, $A[i]$ and $Z[j]$ here.
Arrays are indexed from 0.

If an object or variable is created by declaring it, it is deleted automatically when the subroutine containing it finishes. Then, the memory area reserved by it can be reused for other purposes.

If we want to declare array pointers, we can do it the following way.

$$P : \mathcal{T}[]$$

Now, given the declarations above, after the assignment statement $P := Z$, P and Z refer to the same array object, $P[0]$ is identical with $Z[0]$, and so on, $P[n-1]$ is identical with $Z[n-1]$.

Array objects can be created (i.e. allocated) dynamically, like in C++, but our array objects always contain their size, unlike in C++. For example, the statement

$$P := \textbf{new}\, \mathcal{T}[m]$$

creates a new array object, pointer P refers to it, $P.length = m$.

Note that any object (especially an array object) generated dynamically must be deleted (i.e. deallocated) explicitly when the object is not needed any more. Deletion is done, like in C++, to avoid memory leaking.

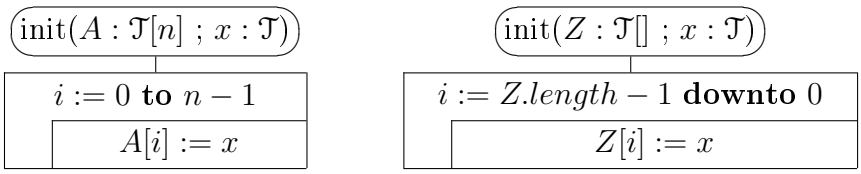
$$\textbf{delete}\, P$$

The above statement suffices here. It does not delete the pointer but the pointed object. Having deleted the object, the memory area reserved by it can be reused for other purposes.

Unfortunately, we cannot say anything about the efficiency of memory allocation and deallocation in general. Sometimes, these can be performed with $\Theta(1)$ time complexity, but usually, they need much more time, and their efficiency is often unpredictable. Therefore, we avoid their overuse, and we apply them only when they are essential.

Suppose we want to pass an array parameter to a subroutine. In that case, the actual parameter must be the array identifier, the formal parameter has to be specified as an array pointer, and the parameter passing copies the address of the actual array object into the formal parameter. Writing an identifier between the square brackets is a short notation for the array's length.

The following two procedures are equivalent because in the left structure diagram, $n = A.length$.



Given an array A, subarray $A[u..v]$ denotes the following sequence of elements of A: $\langle A[u], \ldots, A[v] \rangle$ where u and v are valid indexes in the array. If $u > v$, the subarray is empty (i.e. $\langle\rangle$). In this case, u or v may be an invalid index.

Given an array A, subarray $A[u..v)$ denotes the following sequence of elements of A: $\langle A[u], \ldots, A[v-1] \rangle$ where u and $v-1$ are valid indexes in the array. If $u \geq v$, the subarray is empty (i.e. $\langle\rangle$).

2.2 Stacks

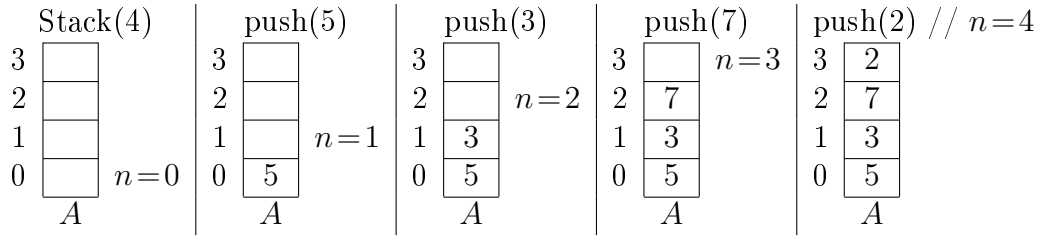
A stack is a LIFO (Last-In, First-Out) data storage. It can be imagined as a vertical sequence of items similar to a tower of plates on a table. We can push a new item at the top and check or remove (i.e., pop) the topmost item.

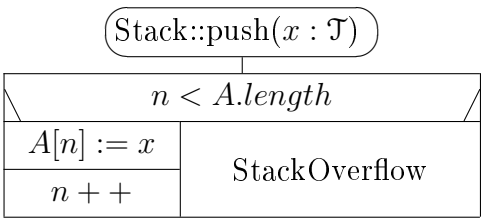
The stack elements are stored in subarray $A[0..n)$ in the following representation. Provided that $n > 0$, $A[n-1]$ is the top of the stack. Provided that $n = 0$, the stack is empty.

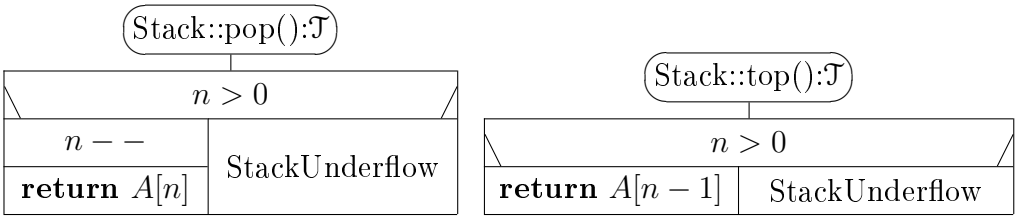
$T(n) \in \Theta(1)$ for each method because neither iteration nor subroutine invocation exists in their code. The time complexities of the constructor and the destructor depend on the **new** and **delete** expressions.

Stack
$-A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the max. size of the stack
$-n : \mathbb{N}$ // $n \in 0..A.length$ is the actual size of the stack
+ Stack($m : \mathbb{N}$) $\{A :=$ **new** $\mathcal{T}[m]$; $n := 0\}$ // create an empty stack
+ push($x : \mathcal{T}$) // push x onto the top of the stack
+ pop() : $\mathcal{T}$ // remove and return the top element of the stack
+ top() : $\mathcal{T}$ // return the top element of the stack
+ isFull() : $\mathbb{B}$ $\{$**return** $n = A.length\}$
+ isEmpty() : $\mathbb{B}$ $\{$**return** $n = 0\}$
+ setEmpty() $\{n := 0\}$ // reinitialize the stack
+ $\sim$ Stack() { **delete** A }

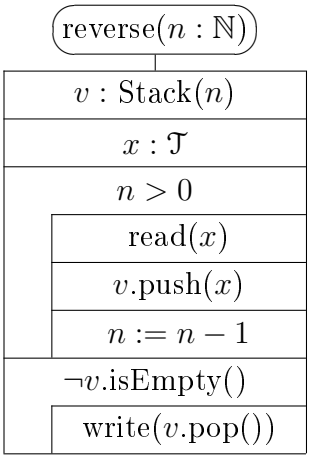
Some operations of a Stack







Example for a simple use of a stack: printing n input items in reversed order. We suppose that read(x) reads from the current input the next piece of data and write(x) prints the value of x to the current output. $T_{\text{reverse}}(n) \in \Theta(n)$.



2.3 Queues

A queue is a FIFO (First-In, First-Out) data storage. It can be imagined as a horizontal sequence of items similar to a queue at the cashier's desk. We can add a new item to the end of the queue, and we can check or remove the first item.

In the following representation, the elements of the queue are stored in

$$\langle Z[k], Z[(k+1) \bmod Z.length], \ldots, Z[(k+n-1) \bmod Z.length] \rangle$$

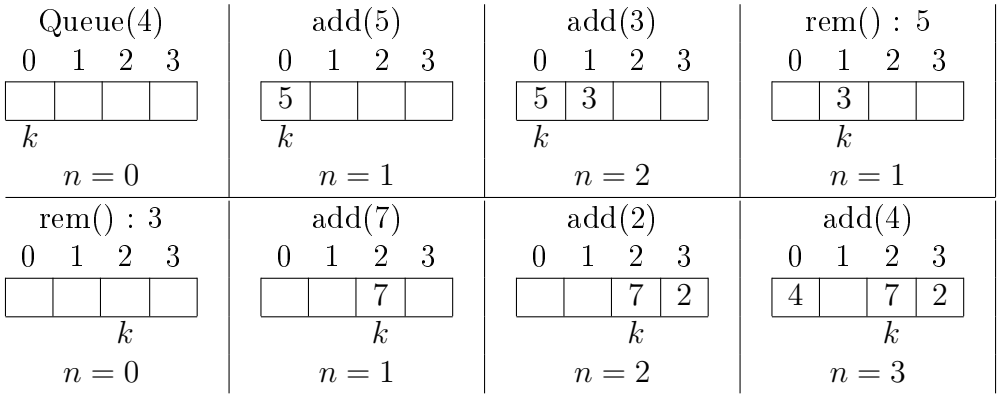
Provided that $n > 0$, $Z[k]$ is the first element of the queue, where n is the length of the queue. Provided that $n = 0$, the queue is empty.

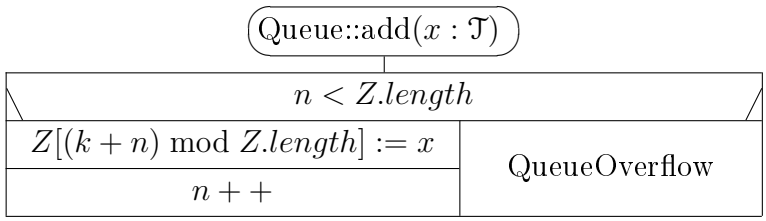
$T(n) \in \Theta(1)$ for each method because neither iteration nor subroutine invocation exists in their code. The time complexities of the constructor and the destructor depend on the **new** and **delete** expressions.

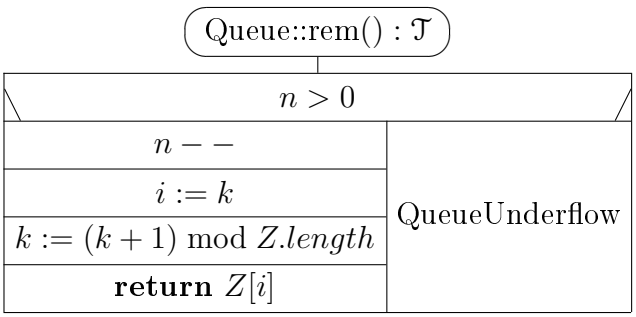
Queue
$-Z : \mathcal{T}[]$ // $\mathcal{T}$ is some known type $-n : \mathbb{N}$ // $n \in 0..Z.length$ is the actual length of the queue $-k : \mathbb{N}$ // $k \in 0..(Z.length-1)$ is the starting position of the queue in array Z

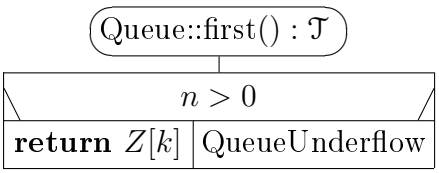
+ Queue($m : \mathbb{N}$){ $Z :=$ **new** $\mathcal{T}[m]$; $n := 0$; $k := 0$ } // create an empty queue
+ add($x : \mathcal{T}$) // join x to the end of the queue
+ rem() : $\mathcal{T}$ // remove and return the first element of the queue
+ first() : $\mathcal{T}$ // return the first element of the queue
+ length() : $\mathbb{N}$ {**return** n}
+ isFull() : $\mathbb{B}$ {**return** $n = Z.length$}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ $\sim$ Queue() { **delete** Z }
+ setEmpty() {$n := 0$} // reinitialize the queue

Some operations of a Queue









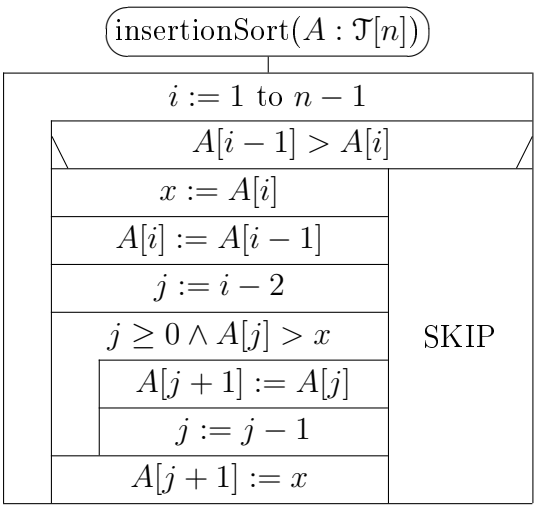
We described the methods of stacks and queues with simple codes: we applied neither iterations nor recursions. Therefore, the time complexity of each method is $\Theta(1)$. This is a fundamental requirement for each implementation of stacks and queues.

Note that with linked list representations, this constraint can be guaranteed only if $MT_{\text{new}}, MT_{\text{delete}} \in \Theta(1)$.

Considering the above implementations of stacks and queues, the destructor's space complexity and each method's space complexity are also $\Theta(1)$ because they create no data structure but use only a few temporal variables. The space complexity of the constructor is $\Theta(m)$.

3 Algorithms in computing: Insertion Sort

[1] Chapter 1-3; [4] Chapter 1, 2, 7



$$mT_{IS}(n) = 1 + (n-1) = n$$

$$MT_{IS}(n) = 1 + (n-1) + \sum_{i=1}^{n-1}(i-1) = n + \sum_{j=0}^{n-2} j = n + \frac{(n-1)*(n-2)}{2}$$

$$MT_{IS}(n) = \frac{1}{2}n^2 - \frac{1}{2}n + 1$$

$MT_{IS}(n) \approx (1/2) * n^2$, if n is large.

Suppose our computer can perform
$2 * 10^9$ elementary operations /second.
Considering the code of insertion sort above and counting these operations as a function of n, we receive that $mT(n) \geq 8 * n$ and $MT(n) > 6 * n^2$ elementary operations. Counting with $mT(n) \approx 8 * n$ and $MT(n) \approx 6 * n^2$, we receive the following table on running times as lower estimates:

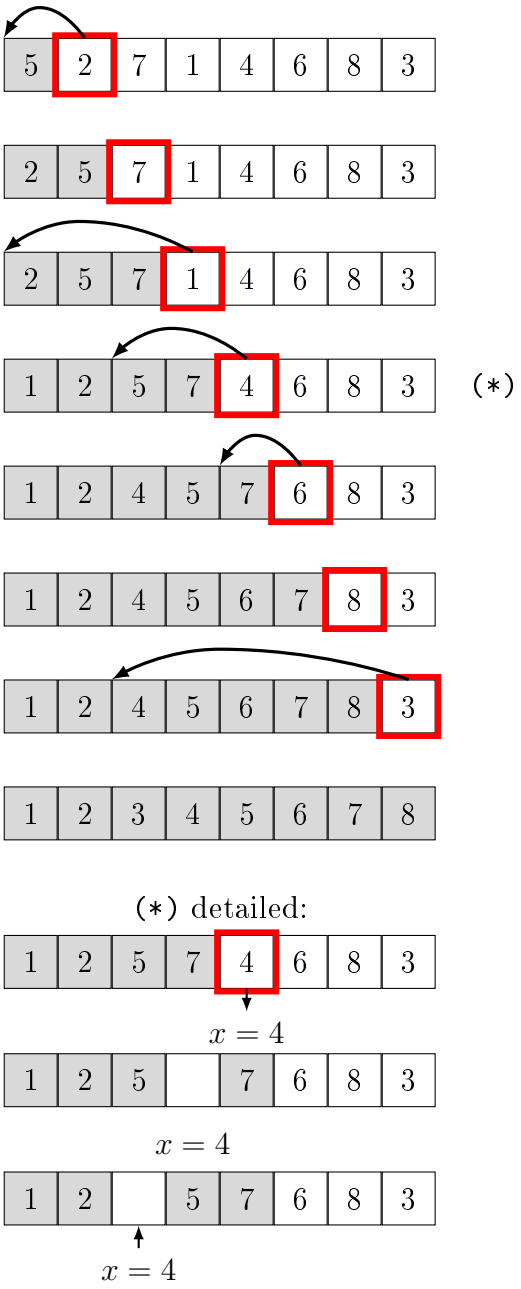


Figure 1: An illustration of Insertion Sort. In such examples, $key = key'$ for any key. This way, we demonstrate handling different occurrences of the same key. Exercise: Give a detailed illustration of the last insertion.

n	$mT(n)$	in secs	$MT(n)$	in time
1000	8000	$4 * 10^{-6}$	$6 * 10^{6}$	0.003 sec
10^6	$8 * 10^{6}$	0.004	$6 * 10^{12}$	50 min
10^7	$8 * 10^{7}$	0.04	$6 * 10^{14}$	$\approx$ 3.5 days
10^8	$8 * 10^{8}$	0.4	$6 * 10^{16}$	$\approx$ 347 days
10^9	$8 * 10^{9}$	4	$6 * 10^{18}$	$\approx$ 95 years

In the worst case, insertion sort is too slow to sort one million elements, and it becomes impractical if we try to sort a huge amount of data. Let us consider the average case:

$$AT_{IS}(n) \approx 1 + (n-1) + \sum_{i=1}^{n-1} \left(\frac{i-1}{2} \right) = n + \frac{1}{2} * \sum_{j=0}^{n-2} j =$$

$$= n + \frac{1}{2} * \frac{(n-1) * (n-2)}{2} = \frac{1}{4}n^2 + \frac{1}{4}n + \frac{1}{2}$$

This calculation shows that the expected or average running time of insertion sort is roughly half of the time needed in the worst case, so even the expected running time of insertion sort is too long to sort one million elements, and it becomes completely impractical if we try to sort huge amount of data. The asymptotic time complexities:

$$mT_{IS}(n) \in \Theta(n)$$
$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

Let us note that the minimum running time is perfect. There is no chance to sort elements faster than in linear time because each piece of data must be checked. One may say that this best case does not have much gain because, in this case, the items are already sorted. If the input is *nearly sorted*, we can remain close to the best case, and insertion sort turns out to be the best to sort such data.

Insertion sort is also *stable*: Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). A sorting algorithm is stable if there are two records, R and S, with the same key and R appearing before S in the original list; R will appear before S in the sorted list. (For example, see keys 2 and 2' in Figureinsertion-sort.) Stability is an essential property of sorting methods in some applications.

The space complexity of insertion sort is $\Theta(1)$ because it creates no data structure but uses only a few temporal variables.

4 Fast sorting algorithms based on the *divide and conquer* approach

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for many problems, for example, sorting (e.g., quicksort, mergesort).

4.1 Merge sort

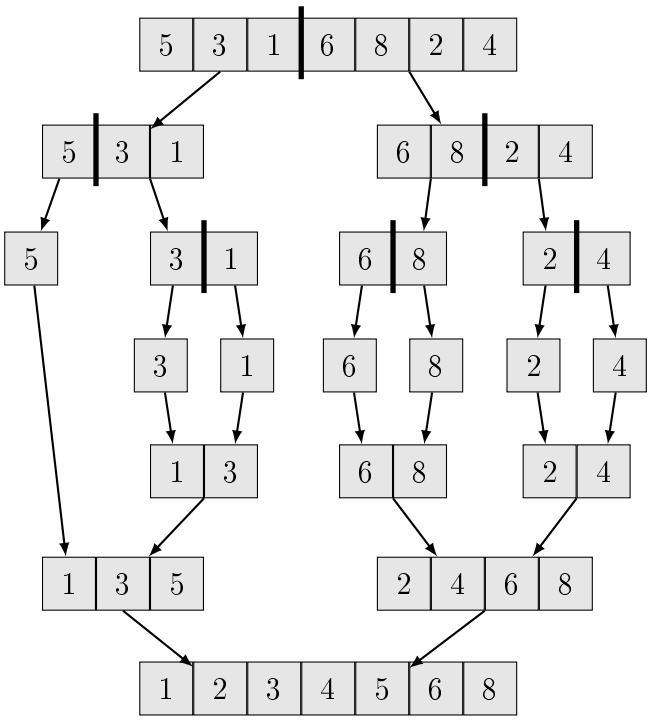
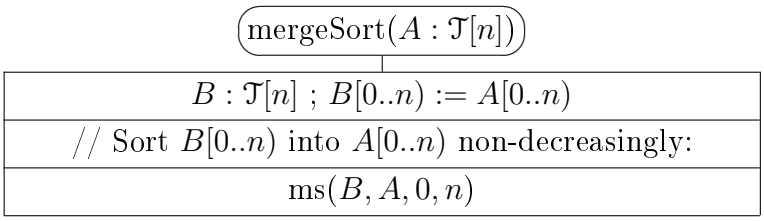


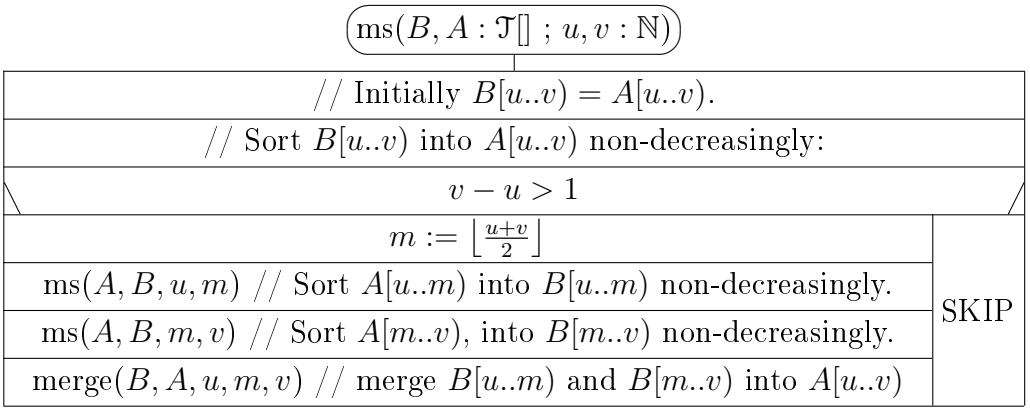
Figure 2: An illustration of merge sort.

The divide and conquer paradigm is often used to find the optimal solution to a problem. Its basic idea is to decompose a given problem into two or more similar but simpler subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity

are solved directly. For example, to sort a given list of n keys, split it into two lists of about $n/2$ keys each, sort each in turn, and interleave (i.e. merge) both results appropriately to obtain the sorted version of the given list. (See Figure 2.) This approach is known as the merge sort algorithm.

Merge sort is stable (preserving the input order of items with equal keys), and its worst-case time complexity is asymptotically optimal among comparison sorts. (See section 7 for details.)

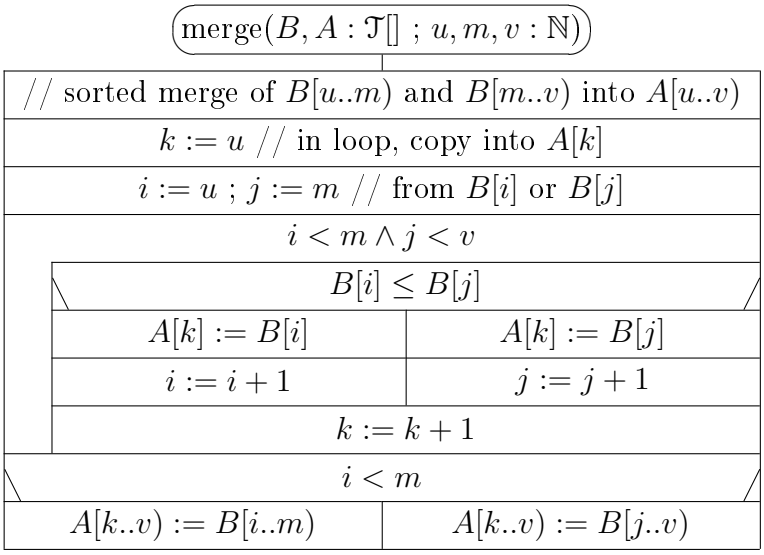




Using $m = \lfloor \frac{u+v}{2} \rfloor$, $A[u..m)$ and $A[m..v)$ have the same length, if the length of $A[u..v)$ is even number; and $A[u..m)$ is shorter by one than $A[m..v)$, if the length of $A[u..v)$ is odd number; because

$$length(A[u..m)) = m - u = \left\lfloor \frac{u+v}{2} \right\rfloor - u =$$

$$\left\lfloor \frac{u+v}{2} - u \right\rfloor = \left\lfloor \frac{v-u}{2} \right\rfloor = \left\lfloor \frac{length(A[u..v))}{2} \right\rfloor$$



The stability of merge sort is ensured in the explicit loop of merge because in the case of $B[i] = B[j]$, $B[i]$ is copied into $A[k]$, and $B[i]$ came earlier in the input.

Merge makes l loop iterations where $l = v - u$ is the length of the actual subarrays $A[u..v)$ and $B[u..v)$. To prove this statement, consider the value of k after the explicit merge loop. This loop fills subarray $A[u..k)$ and copies one element/iteration. Thus, it iterates $k - u$ times. In addition, the implicit loop is hidden into $A[k..v) := \ldots$ and iterates $v - k$ times. Therefore, the sum of the loop iterations is $(k - u) + (v - k) = v - u = l$. Consequently, for the body of procedure merge (mb) we have:

$$T_{\mathrm{mb}}(l) = l \quad (l = v - u)$$

4.1.1 The time complexity of merge sort

Merge sort is one of the fastest sorting algorithms, and there is not a big difference between its worst-case and best-case (i.e. maximal and minimal) running time. For our array sorting version, we state:

$$MT_{\mathrm{mergeSort}}(n) = mT_{\mathrm{mergeSort}}(n) = T_{\mathrm{mergeSort}}(n) \in \Theta(n \log n)$$

Proof: Clearly $T_{\mathrm{mergeSort}}(0) = 2$. We suppose that $n > 0$. First, we count all the loop iterations of the procedure merge. Next, we count the procedure calls of merge sort.

Loop iterations: We proved above that a single call of merge(B, A, u, m, v) makes $T_{\mathrm{mb}}(l) = l$ iterations where $l = v - u$. Let us consider the levels of recursion of procedure ms(B, A, u, v). At level 0 of the recursion, ms is

called for the whole array $A[0..n)$. Considering all the recursive calls and the corresponding subarrays at a given recursion depth, at level 1, this array is divided into two halves, at level 2 into 4 parts and so on. Let n_{ij} be the length of the jth subarray of the form $A[u..v)$ at level i, and let $m = \lfloor \log n \rfloor$. We have:

At level 0: $2^m \leq n_{01} = n < 2^{m+1}$
At level 1: $2^{m-1} \leq n_{1j} \leq 2^{m+1-1}$ $(j \in [1..2^1])$
At level 2: $2^{m-2} \leq n_{2j} \leq 2^{m+1-2}$ $(j \in [1..2^2])$
...
At level i: $2^{m-i} \leq n_{ij} \leq 2^{m+1-i}$ $(i \in [1..m], j \in [1..2^i])$
...
At level $m-1$: $2 \leq n_{(m-1)j} \leq 4 = 2^2$ $(j \in [1..2^{m-1}])$
At level m: $1 \leq n_{mj} \leq 2 = 2^1$ $(j \in [1..2^m])$
At level $m+1$: $n_{(m+1)j} = 1$ $(j \in [1..(n-2^m)])$

Thus, these subarrays cover the whole array at levels $[0..m]$. At levels $[0..m)$, merge is called for each subarray, but at level m, it is called only for those subarrays with length 2, and the number of these subarrays is $n-2^m$. Merge makes as many iterations as the length of the actual $A[u..v)$. Consequently, at each level in $[0..m)$, merge makes n iterations in all the merge calls of the level altogether. At level m, the sum of the iterations is $2 * (n - 2^m)$, and there is no iteration at level $m+1$. Therefore, the total of all the iterations during the merge calls is

$$T_{mb[0..m]}(n) = n * m + 2 * (n - 2^m).$$

The number of procedure calls: The ms calls form a strictly binary tree. (See section 6.2.) The leaves of this tree correspond to the subarrays with length 1. Thus, this strictly binary tree has n leaves and $n-1$ internal nodes. Consequently, we have $2n-1$ calls of ms and $n-1$ calls of the merge. Adding to this the single call of mergeSort(), we receive $3n-1$ procedure calls altogether.

And there are n iterations hidden into the initial assignment $B[0..n) := A[0..n)$ in procedure mergeSort. Thus, the number of steps of mergeSort is

$$T(n) = (n+n*m+2*(n-2^m))+(3*n-1) = n*\lfloor \log n \rfloor + 2*(n-2^{\lfloor \log n \rfloor})+4*n-1$$

$$n * \log n + 2 * n \leq n * (\log n - 1) + 4 * n - 1 \leq T(n) < n * \log n + 6 * n.$$

Thus $T(n) \in \Omega(n \log n)$ (see Consequence 1.8) and $T(n) \in O(n \log n)$ (see Consequence 1.7). After all we have for mergeSort$(A : \mathcal{T}[n])$

$$T(n) \in \Theta(n \log n).$$

4.1.2 The space complexity of merge sort

Considering the above merge sort code, we create the temporal array $B : \mathcal{T}[n]$ in the main procedure, and we need a temporal variable for the implicit for loop. We need $\Theta(n)$ working memory here. In addition, we have seen in the previous subsection (4.1.1) that the number of levels of recursion is $\lfloor \log n \rfloor + 1$ which is approximately $\log n$, and we have a constant amount of temporal variables at each level. Thus, we need $\Theta(\log n)$ working memory inside the call stack to control the recursion. And $\Theta(\log n) \prec \Theta(n)$. These measures do not depend on the content of the array to be sorted. For this reason, the space complexity of this array version of merge sort is $S(n) \in \Theta(n)$.

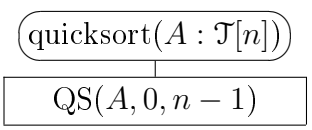
4.2 Quicksort

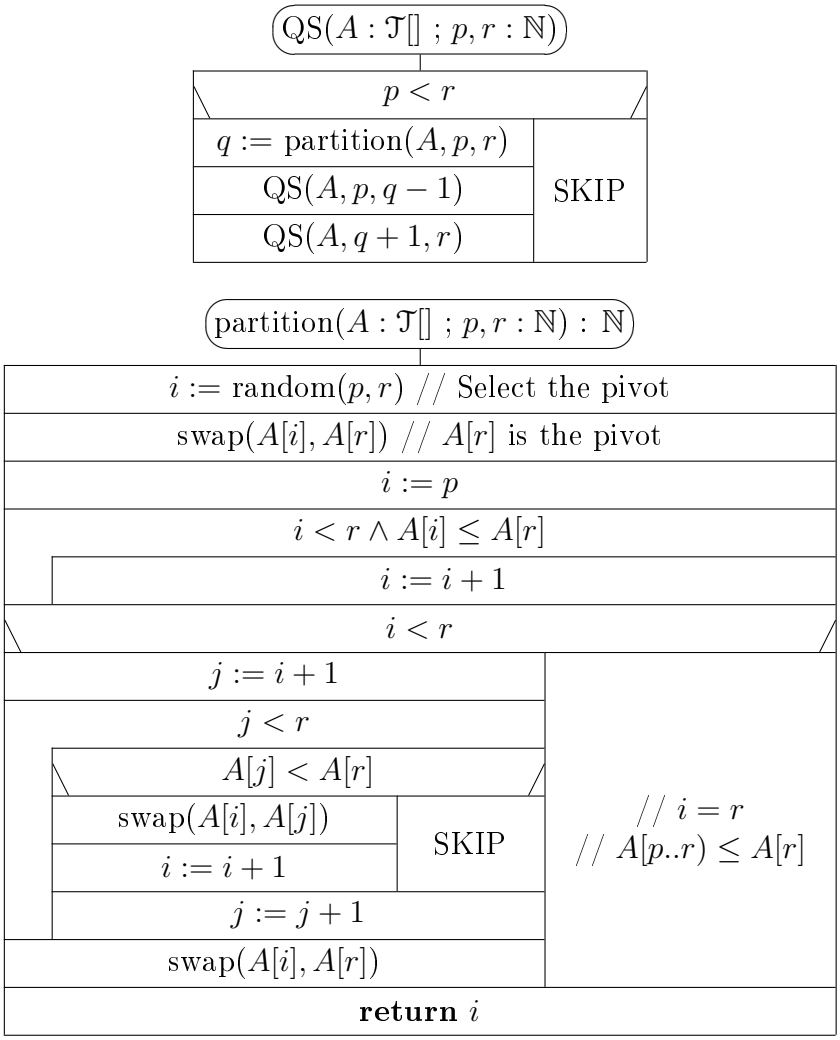
Quicksort is a divide-and-conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

- Pick an element, called *pivot*, from the array.
- Partitioning: reorder the array so that all elements with values less than the *pivot* come before the *pivot*, while all elements with values exceeding the *pivot* come after it (equal values can go either way). After this partitioning, the *pivot* is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

The *pivot* selection and partitioning steps can be done in several ways; the choice of specific implementation schemes significantly affects the algorithm's performance.





Explanation of function partition: To illustrate the operation of function partition, let us introduce the following notation:

- $A[k..m) \leq A[r], \iff$ for each $l \in [k..m)$, $A[l] \leq A[r]$
- $A[k..m) \geq A[r], \iff$ for each $l \in [k..m)$, $A[l] \geq A[r]$

We suppose that subarray $A[p..r]$ is partitioned, and the *pivot* is the second 5, i.e. the 4th element of this subarray (with index p+3).

After the $i :=$ random(p, r) assignment:

	p			i				r
A :	5	3	8	**5**	6	4	7	**1**

Preparations for the first loop: After swap$(A[i], A[r])$, $A[r]$ is the pivot.

	p							r	
$A:$	5	3	8	1	6	4	7	⑤	$pivot = A[r] = 5$

If some element exceeds the *pivot*, the first loop searches for the first such item. (The occurrences of index i reflect its left-to-right movement.)

	i=p	i	i					r
$A:$	5	3	**8**	1	6	4	7	⑤

We have found the first item that is greater than the *pivot*.
Variable j starts at the next item.

	p		i	j				r
$A:$	5	3	**8**	1	6	4	7	⑤

We have cut $A[p..r]$ into four sections:

$$\underset{A[p..i) \le pivot,}{\underset{p}{\le}} \quad \underset{A[i..j) \ge pivot,}{\underset{i}{<}} \quad \underset{A[j..r) \text{ (unchecked)},}{\underset{j}{\le}} \quad \underset{A[r] \text{ (the } pivot).}{r}$$

This is an invariant of the second loop. The first section, i.e. $A[p..i)$ contains items less or equal to the *pivot*, the *nonempty* second section, i.e. $A[i..j)$ contains items greater or equal to the *pivot*, the third section, i.e. $A[j..r)$ contains the unchecked items, while the last section, i.e. $A[r]$ is the *pivot*. (Notice that the items equal to the *pivot* may be either in the first or the second section of $A[p..r]$.)

The elements of the third section are then connected in sequence to the first or second section of elements until the third section is exhausted. Finally, the *pivot* is inserted between the first two sections. Offsetting the second section would result in unacceptably poor efficiency. Thus, this is avoided when attaching to the first section and when the final step is performed. In both cases, we can prevent shifting the second section by swapping the current element with the first element of the second section. The consequence is that the *quicksort algorithm is unstable.*

Starting the run of the second loop, it turns out that $A[j] = 1 < pivot$ must be exchanged with $A[i] = 8 \ge pivot$ to move $A[j]$ into the first section of $A[p..r]$ (which contains items $\le$ than the *pivot*), and to move $A[i]$ ($A[i] \ge pivot$) to the end of the second section (which contains items $\ge$ than the *pivot*). (The items to be exchanged are printed in bold.)

	p		i	j				r
$A:$	5	3	**8**	**1**	6	4	7	⑤

Now we exchange $A[i]$ and $A[j]$. Thus, the length of the first section of $A[p..r]$ (containing items $\leq$ than the *pivot*) has been increased by 1, while its second section (containing items $\geq$ than the *pivot*) has been moved by one position. So, we increment variables i and j by 1 to keep the loop invariant.

	p			i	j			r
$A:$	5	3	1	8	**6**	4	7	⑤

Now $A[j] = 6 \geq pivot = 5$, so we add $A[j]$ to the second section of $A[p..r]$, i.e. we increment j by 1.

	p			i		j		r
$A:$	5	3	1	**8**	6	**4**	7	⑤

And now $A[j] = 4 < pivot = 5$, so $A[j]$ must be exchanged with $A[i]$. (The items to be exchanged are printed in bold.)

Now we exchange $A[i]$ and $A[j]$. Thus, the length of the first section of $A[p..r]$ (containing items $\leq$ than the *pivot*) has been increased by 1, while its second section (containing items $\geq$ than the *pivot*) has been moved by one position. So, we increment variables i and j by 1 to keep the loop invariant.

	p				i		j	r
$A:$	5	3	1	4	6	8	**7**	⑤

Now $A[j] = 7 \geq pivot = 5$, so we add $A[j]$ to the second section of $A[p..r]$, i.e. we increment j by 1.

And now $j = r$, therefore the third (the unknown) section of $A[p..r]$ disappeared.

	p				i			j=r
$A:$	5	3	1	4	**6**	8	7	⑤

Now, the first two sections cover $A[p..r)$, and the second section is not empty because of invariant $i < j$. Thus the *pivot* can be put in between the items $\leq$ than it, and the items $\geq$ than it: we swap the first element ($A[i]$) of the second section ($A[i..j)$) with the ‚*pivot*‚which is $A[r]$, i.e. we perform swap$(A[i], A[r])$. (We show with a "+" sign that the *pivot* is already at its final place.)

	p				i			j=r
$A:$	5	3	1	4	+**5**	8	7	6

The partitioning of the subarray $A[p..r]$ has been completed. We return the position of the *pivot* (i) to inform procedure Quicksort(A, p, r), which subarrays of $A[p..r]$ must be sorted recursively.

In the other case of procedure partition, the *pivot* is a maximum of $A[p..r]$. This case is trivial. Let the reader consider it.

The time complexity of the function partition is linear because the two loops perform $r-p-1$ or $r-p$ iterations together.

There is a big difference between the *best-case* and *worst-case* **time complexities** of **quicksort**. At each level of recursion, counting all the elements in the subarrays of that level, we have to divide not more than n elements. Therefore, the time complexity of each recursion level is $O(n)$.

In a lucky case, at each recursion level, in the partitioning process, the *pivot* divides the actual subarray into two parts of approximately equal lengths, and the recursion depth will be about $\log n$. Thus, the best-case time complexity of quicksort is $O(n \log n)$. It can be proved that it is $\Theta(n \log n)$.

In an unlucky case, the *pivot* will be the maximum or minimum of the actual subarray in the partitioning process at each recursion level. After partitioning, the shorter partition will be empty, but the more extended partition will have all the elements except the *pivot*. Considering always the longer subarray, at level zero, we have n elements; at level one, $n-1$ elements;...at level i, $n-i$ elements ($i \in 0..(n-1)$). As a result, in this case, we have n levels of recursion, and we need approximately $n-i$ steps only for partitioning at level i. Consequently, the worst-case time complexity of quicksort is $\Omega(n^2)$. It can be proved that it is $\Theta(n^2)$.

Fortunately, the probability of the worst case is extremely low, and the average case is much closer to the best case. As a result,

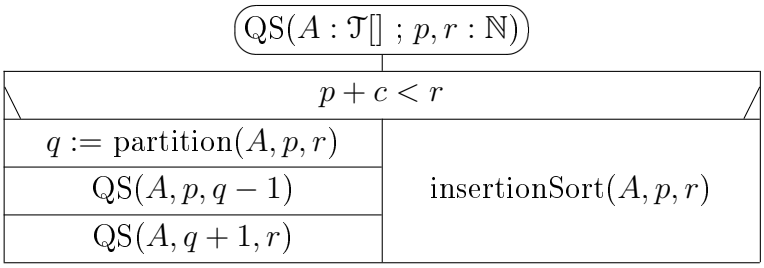
$$mT_{\text{quicksort}}(n), AT_{\text{quicksort}}(n) \in \Theta(n \log n) \quad \wedge \quad MT_{\text{quicksort}}(n) \in \Theta(n^2)$$

Considering its **space complexity, quicksort** does not use any temporal data structure. Thus, the memory needs are determined by the number of recursion levels. As we have seen, it is n in the worst case. And in the best case, it is about $\log n$. Fortunately, in the average case, it is $\Theta(\log n)$. Consequently

$$mS_{\text{quicksort}}(n), AS_{\text{quicksort}}(n) \in \Theta(\log n) \quad \wedge \quad MS_{\text{quicksort}}(n) \in \Theta(n)$$

It is known that insertion sort is more efficient on short arrays than the fast sorting methods (merge sort, heap sort, quicksort). Thus, the procedure quicksort(A, p, r) can be optimised by switching to insertion sort on short

subarrays. Because in the recursive calls, we determine a lot of short subarrays, the speed-up can be significant, although it does change neither the time nor the space complexities.



$c \in \mathbb{N}$ is a constant. Its optimal value depends on many factors, but usually, it is between 20 and 50.

Exercise 4.1 *How do you speed up the merge sort in a similar way?*

Considering its **expected running time, quicksort** is one of the most efficient sorting methods. However, if (for example), by chance, the function partition always selects the maximal or minimal element of the actual subarray, it slows down. [In this case, the time complexity of quicksort is $\Theta(n^2)$.] The probability of such cases is low. However, to avoid such cases, we can pay attention to the recursion depth and switch to heap sort (see later) when recursion becomes too deep (for example, deeper than $2 * \log n$). With **this optimisation**, even the worst-case time complexity is $\Theta(n \log n)$, and even the worst-case space complexity is $\Theta(\log n)$.

5 Linked Lists

One-way lists can represent finite sequences of data. They consist of zero or more elements (i.e. nodes) where each list element contains some data and linking information.

5.1 One-way or singly linked lists

In singly linked lists, each element contains a *key* and a *next* pointer referring to the next element of the list. A pointer referring to the front of the list identifies the list. The *next* pointer of the last item typically contains a so-called NULL (i.e. $\oslash$) pointer[1], which is the address of no object. (With some abstraction, the nonexisting object with address $\oslash$ can be called a NO object.) In this book, E1 is the element type of the different kinds of one-way lists.

E1
$+key : \mathcal{T}$ ... // satellite data may come here $+next$: **E1***
$+$**E1**() { $next := \oslash$ }

5.1.1 Simple one-way lists (S1L)

An S1L is identified by a pointer referring to its first element, or this pointer is $\oslash$ if the list is empty.

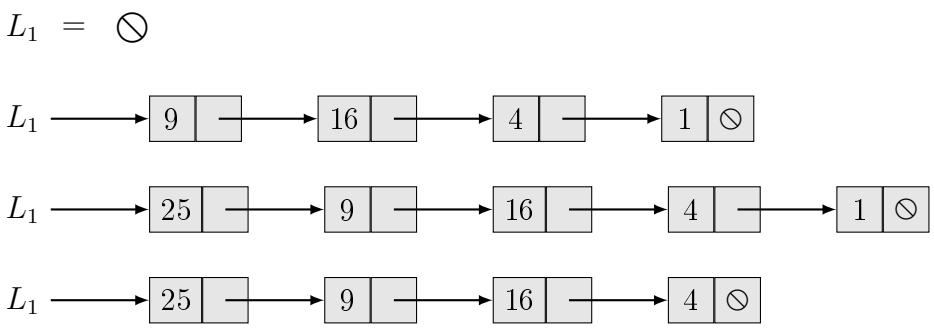
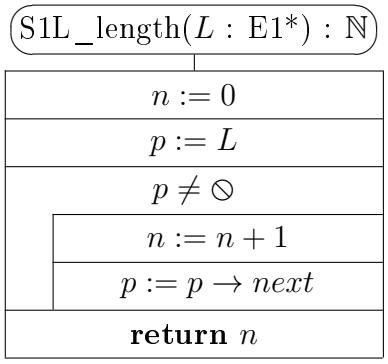


Figure 3: Pointer L_1 identifies simple one-way lists at different stages of an imaginary program. In the first line, the S1L is empty.

[1] except in cyclic lists



If n is the length of list L,
then
the loop iterates n times,
so
$T_{S1L\_length}(n) = 1 + n$,
therefore
$T_{S1L\_length}(n) \in \Theta(n)$.

5.1.2 One-way lists with header node (H1L)

The header node or simply header of a list is an extra *zeroth* element with undefined *key*. A list with a header cannot be $\oslash$; it always contains a header. The pointer identifying the list always refers to its header.

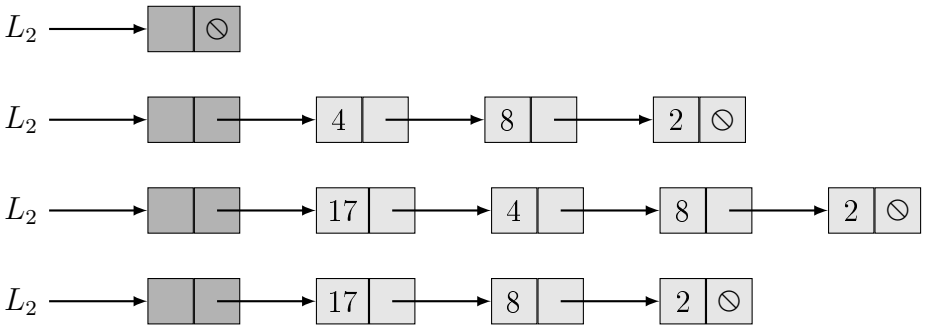
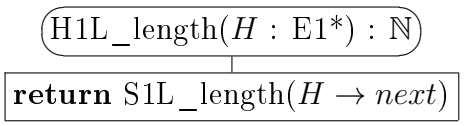


Figure 4: Pointer L_2 identifies one-way lists with heather at different stages of an imaginary program. In the first line, the H1L is empty.



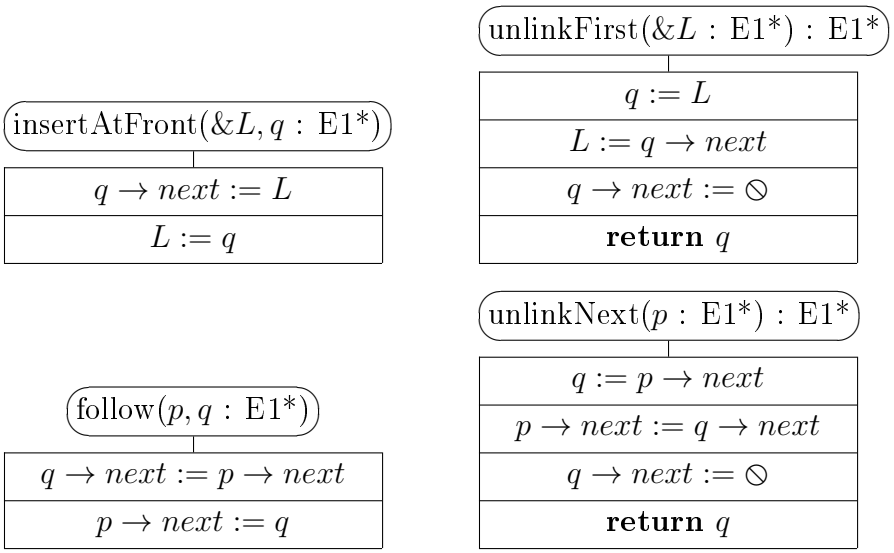
$T_{H1L\_length}(n) \in \Theta(n)$
where n is the length of H1L H.

5.1.3 One-way lists with trailer node

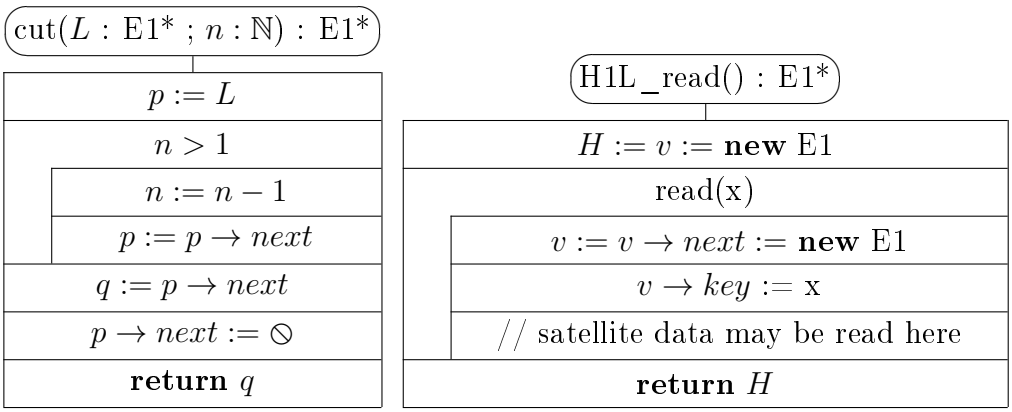
Some one-way lists contain a trailer node instead of a header. A trailer node is always at the end of such lists. The data members (*key* and *next*) of the trailer node are typically undefined, and we have two pointers identifying the list: One refers to its first element and the other to its trailer node. If the list is empty, both identifier pointers refer to its trailer node.

Such a list is ideal for representing a queue. Its add($x : \mathcal{T}$) method copies x into the *key* of the trailer node and joins a new trailer node to the end of the list.

5.1.4 Handling one-way lists



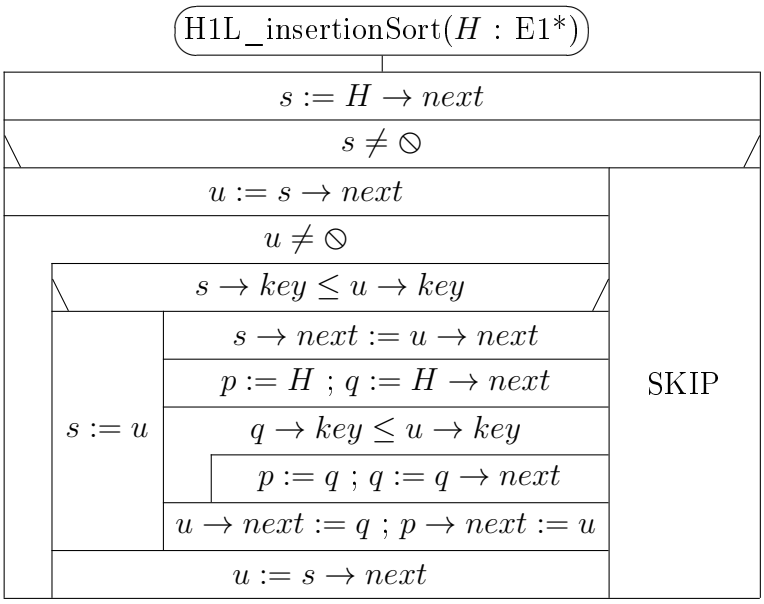
$$T_{insertAtFront}, T_{unlinkFirst}, T_{follow}, T_{unlinkNext} \in \Theta(1)$$
$$S_{insertAtFront}, S_{unlinkFirst}, S_{follow}, S_{unlinkNext} \in \Theta(1)$$



$$T_{cut}(n) \in \Theta(n) \quad \wedge \quad S_{cut}(n) \in \Theta(1)$$
$$T_{H1L\_read}(n) \in \Theta(n) \quad \wedge \quad S_{H1L\_read}(n) \in \Theta(n)$$

where n is the length of the list.

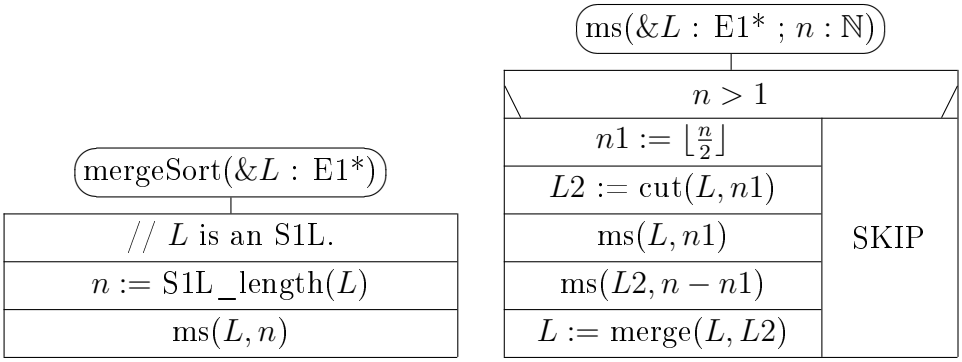
5.1.5 Insertion sort of H1Ls

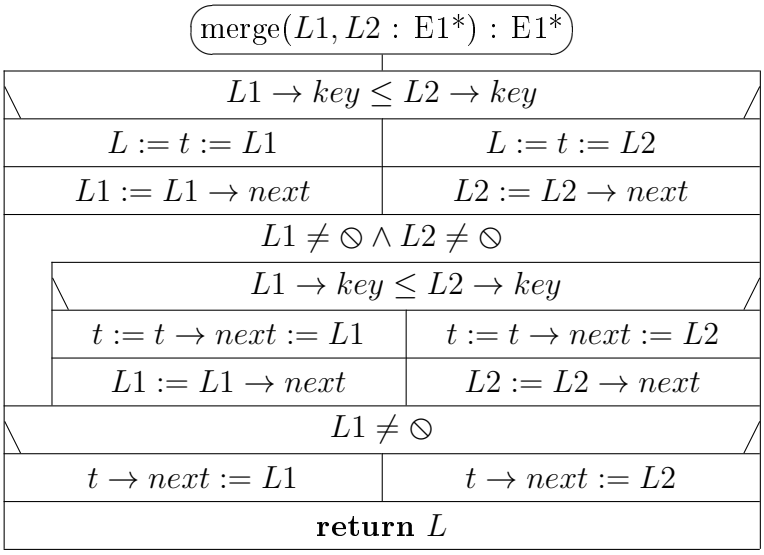


$$mT_{IS}(n) \in \Theta(n) \quad \wedge \quad AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2) \quad \wedge \quad S_{IS}(n) \in \Theta(1)$$

where n is the length of H1L H. Clearly procedure insertionSort(H : E1*) is stable. Like the array version of it, it also sorts in place.

5.1.6 Merge sort of S1Ls





$$mT_{MS}(n), MT_{MS}(n) \in \Theta(n \log n) \quad \wedge \quad S_{MS}(n) \in \Theta(\log n)$$

where n is the length of S1L L. Clearly procedure mergeSort(&L : E1*) is stable. Unlike the array version of merge sort, it sorts non-strictly in place.

5.1.7 Cyclic one-way lists

The last element of the list does not contain $\oslash$ in its *next* field, but this pointer points back to the beginning of the list.

If a cyclic one-way list does not contain a header node and this list is nonempty, the *next* field of the list's last element refers to the list's first element. If it is empty, it is represented by the $\oslash$ pointer. A pointer identifying a nonempty, cyclic one-way list typically refers to its last element.

If a cyclic one-way list has a header node, the *next* field of the list's last element refers to the list's header node. If the list is empty, the *next* field of its header node refers to this header node. Notice that the header of a cyclic list is also the trailer node of that list.

Such lists are also good choices for representing queues. Given a queue represented by a cyclic list with a header, the add(x : $\mathfrak{T}$) method copies x into the *key* of the trailer/header node and inserts a new trailer/header node into the list.

5.2 Two-way or doubly linked lists

In a two-way list, each element contains a *key* and two pointers: a *next* and a *prev* pointer referring to the next and previous elements of the list. A

pointer referring to the front of the list identifies the list.

5.2.1 Simple two-way lists (S2L)

An S2L is identified by a pointer referring to its first element, or this pointer is $\oslash$ if the list is empty.

The *prev* pointer of the first element and the *next* pointer of the last item contains the $\oslash$ pointer.

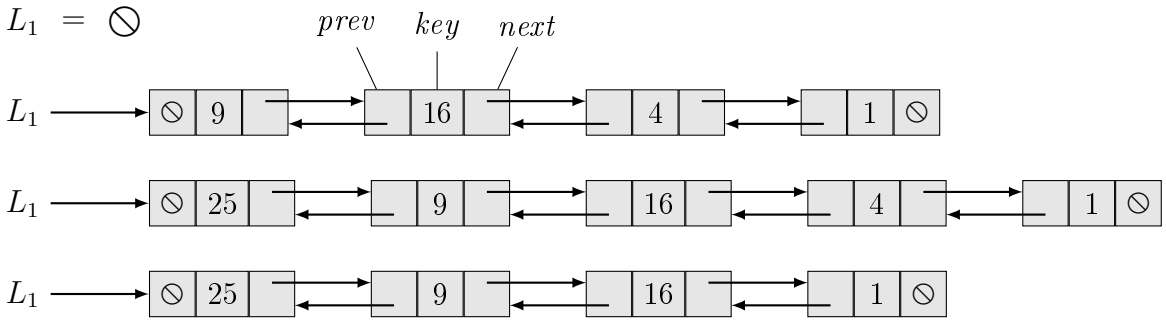


Figure 5: Pointer L_1 identifies simple two-way lists (S2L) at different stages of an imaginary program. In the first line list L_1 is initialized to empty.

Handling S2Ls is inconvenient because all the modifications of an S2L must be done differently at the front of a list, at the end of the list, and in between. Consequently, we prefer cyclic two-way lists (C2L) in general. (In hash tables, however, S2Ls are usually preferred to C2Ls.)

5.2.2 Cyclic two-way lists (C2L)

A cyclic two-way list (C2L) contains a header node (i.e. header) by default. The list is identified by a pointer referring to its header. The *next* field of the list's last element refers to the list's header, considered a zeroth element. The *prev* pointer of the list's first element refers to the list's header, and the *prev* pointer of the list's header refers to the list's last element. If a C2L is empty, the *prev* and *next* fields of the list's header refer to this header. Notice that the header of a cyclic list is also the trailer node of the list.

The element type of C2Ls follows.

E2
$+prev, next$: **E2*** // refer to the previous and next neighbour or be **this** $+key : \mathcal{T}$
+ E2() { $prev := next :=$ **this** }

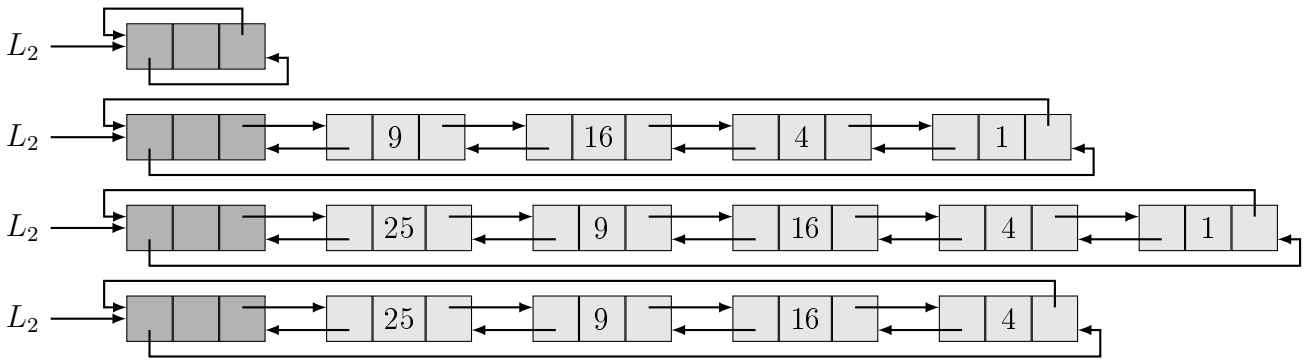


Figure 6: Pointer L_2 identifies cyclic two-way lists (C2L) at different stages of an imaginary program. In the first line list L_2 is empty.

Some basic operations on C2Ls follow. Notice that these are simpler than the appropriate operations of S2Ls. $T, S \in \Theta(1)$ for each of them.

precede(q, r : E2*)
// $(*q)$ will precede $(*r)$
$p := r \to prev$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$

follow(p, q : E2*)
// $(*q)$ will follow $(*p)$
$r := p \to next$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$

unlink(q : E2*)
// remove $(*q)$
$p := q \to prev$; $r := q \to next$
$p \to next := r$; $r \to prev := p$
$q \to prev := q \to next := q$

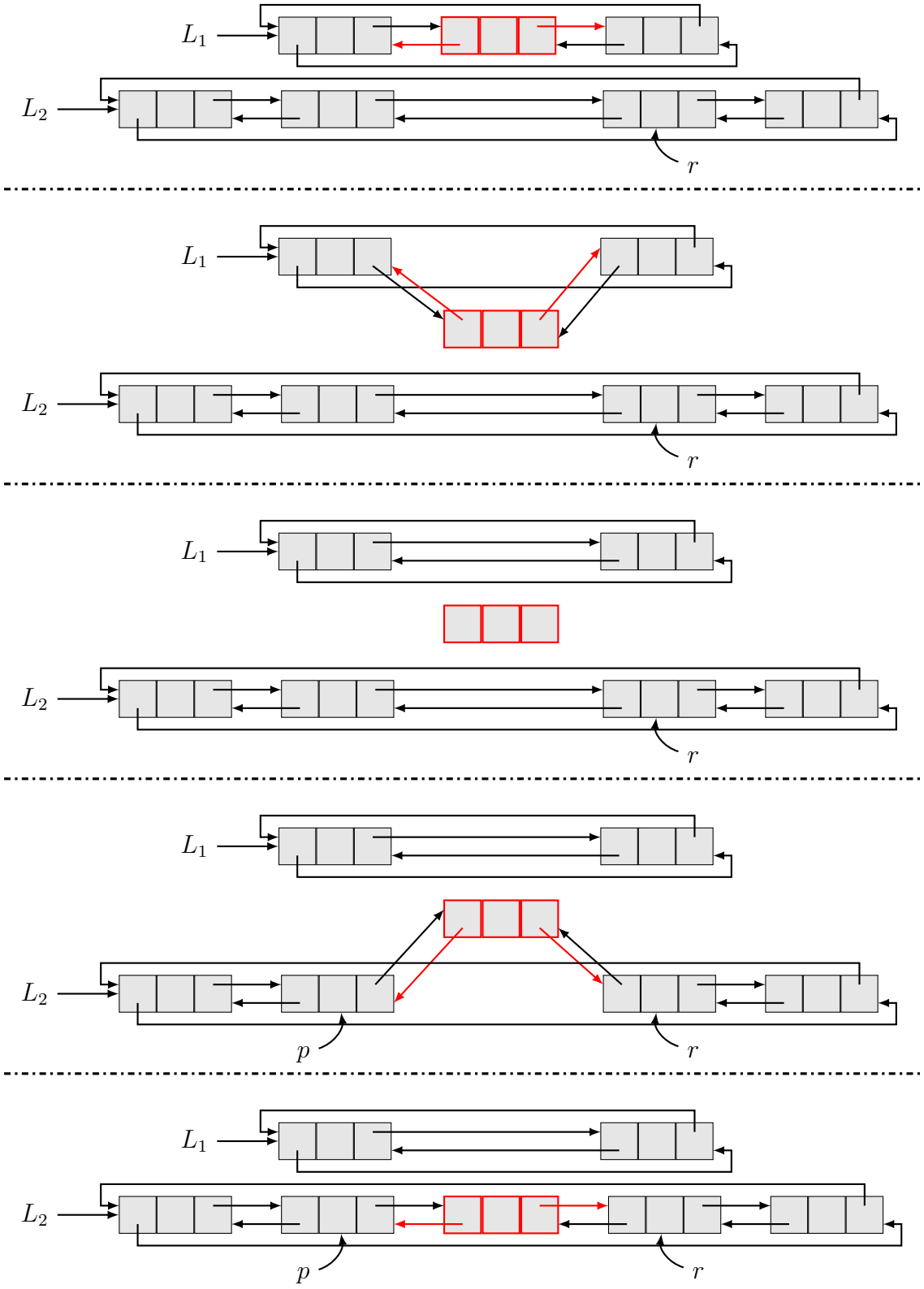
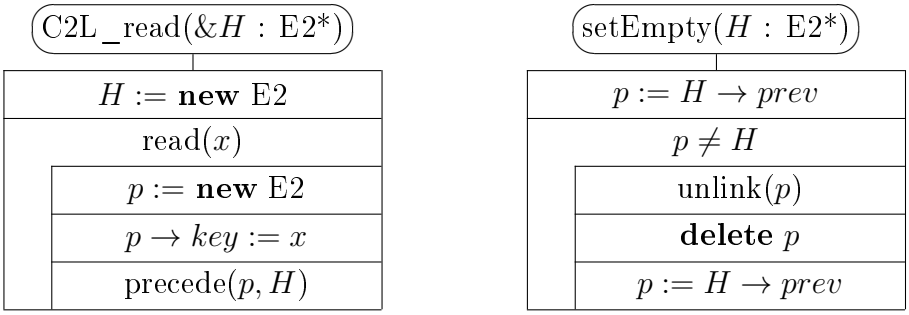


Figure 7: Illustrating unlink(q) and precede(q, r). Insertion is called on the red element ($*q$), and it is inserted left to ($*r$) into list L_2.

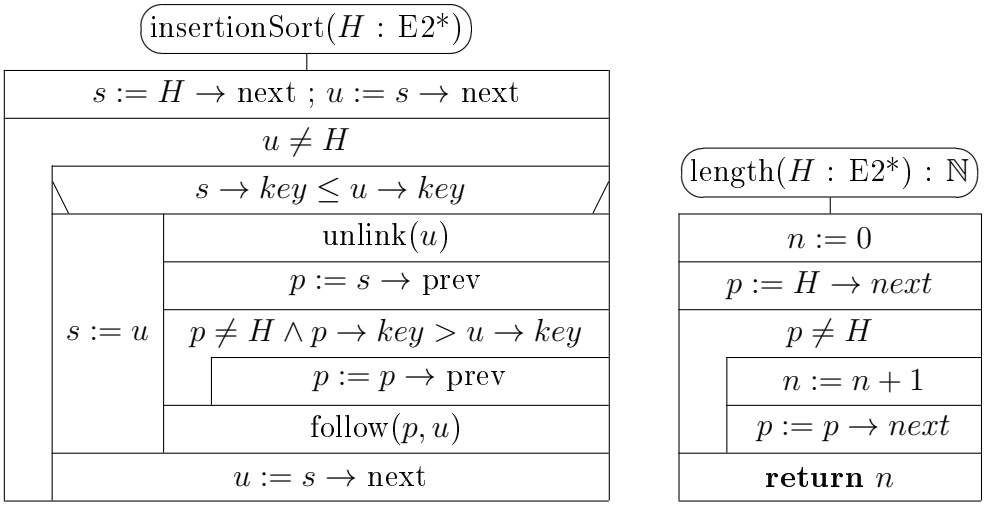
5.2.3 Example programs on C2Ls

We can imagine a C2L straightened, for example.
$H \to$ [/]—[5]—[2]—[7]—[/] $\leftarrow H$ representing $\langle 5; 2; 7\rangle$
or empty: $H \to$ [/]—[/] $\leftarrow H$ representing $\langle\rangle$.



$H \to$ [/]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[2]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[2]—[7]—[/] $\leftarrow H$

$$T_{C2L\_read}(n), T_{setEmpty}(n), T_{lenght}(n) \in \Theta(n)$$



$H \to$ [/]—[5]—[2]—[7]—[2′]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[5]—[7]—[2′]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[5]—[7]—[2′]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[2′]—[5]—[7]—[/] $\leftarrow H$

$$mT_{IS}(n) \in \Theta(n); AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2) \quad \wedge \quad S_{IS}(n) \in \Theta(1)$$

where n is the length of C2L H. Clearly procedure insertionSort(H : E2*) is stable.

Example 5.1 *Let us suppose that H_u and H_i are strictly increasing C2Ls. We write procedure unionIntersection(H_u, H_i : E2*) which calculates the strictly increasing union of the two lists in H_u and the strictly increasing intersection of them in H_i.*

We use neither memory allocation **(new)** *nor deallocation* **(delete)** *statements nor explicit assignment statements to data members. We rearrange the lists only with unlink(q), precede(q,r), and follow(p,q). $MT(n,m) \in O(n+m)$ and $S(n,m) \in \Theta(1)$ where $n = lenght(H_u)$ and $m = length(H_i)$.*

Let us have $q, r : E2*$ initialised to point to the first items of H_u and H_i, respectively. For example:

$H_u \rightarrow$ [/]—[$\overset{q}{2}$]—[4]—[6]—[/] $\leftarrow H_u$
$H_i \rightarrow$ [/]—[$\underset{r}{1}$]—[4]—[8]—[9]—[/] $\leftarrow H_i$

We work with the following invariant where

$$key(q,H) = \begin{cases} q \rightarrow key & \text{if} \quad q \neq H \\ \infty & \text{if} \quad q = H \end{cases}$$

(H, q) is the sublist of the items between H and q,
$[q, H)$ is the sublist of the items starting with q but before H.

- C2Ls H_u and H_i are strictly increasing consisting of the original bag of list items, q is a pointer on H_u and r on H_i,
- (H_u, q) is the prefix of the sorted union containing the keys less than $min(key(q, H_u), key(r, H_i))$,
- (H_i, r) is the prefix of the sorted intersection containing the keys less than $min(key(q, H_u), key(r, H_i))$,
- $[q, H_u)$ and $[r, H_i)$ are still unaltered.

Illustration of the run of the program:

$H_u \rightarrow$ [/]—[$\overset{q}{2}$]—[4]—[6]—[/] $\leftarrow H_u$
$H_i \rightarrow$ [/]—[$\underset{r}{1}$]—[4]—[5]—[8]—[9]—[/] $\leftarrow H_i$

$H_u \rightarrow$ [/]—[1]—[$\overset{q}{2}$]—[4]—[6]—[/] $\leftarrow H_u$
$H_i \rightarrow$ [/]—[$\underset{r}{4}$]—[5]—[8]—[9]—[/] $\leftarrow H_i$

$H_u \to$ [/]—[1]—[2]—[$\overset{q}{4}$]—[6]—[/] $\leftarrow H_u$
$H_i \to$ [/]—[$\underset{r}{4}$]—[5]—[8]—[9]—[/] $\leftarrow H_i$

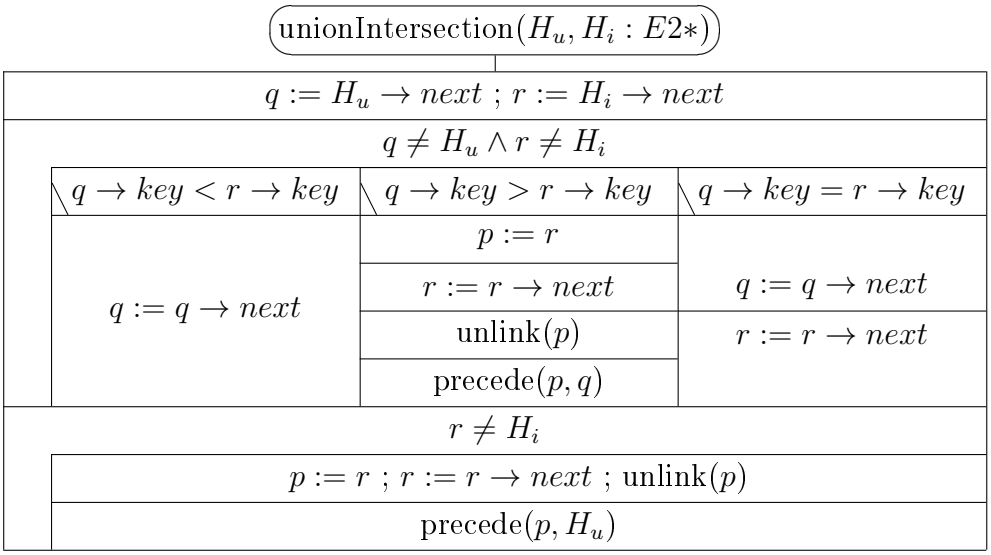
$H_u \to$ [/]—[1]—[2]—[4]—[$\overset{q}{6}$]—[/] $\leftarrow H_u$
$H_i \to$ [/]—[4]—[$\underset{r}{5}$]—[8]—[9]—[/] $\leftarrow H_i$

$H_u \to$ [/]—[1]—[2]—[4]—[5]—[$\overset{q}{6}$]—[/] $\leftarrow H_u$
$H_i \to$ [/]—[4]—[$\underset{r}{8}$]—[9]—[/] $\leftarrow H_i$

$H_u \to$ [/]—[1]—[2]—[4]—[5]—[6]—[$\overset{q}{/}$] $\leftarrow H_u$
$H_i \to$ [/]—[4]—[$\underset{r}{8}$]—[9]—[/] $\leftarrow H_i$

$H_u \to$ [/]—[1]—[2]—[4]—[5]—[6]—[8]—[$\overset{q}{/}$] $\leftarrow H_u$
$H_i \to$ [/]—[4]—[$\underset{r}{9}$]—[/] $\leftarrow H_i$

$H_u \to$ [/]—[1]—[2]—[4]—[5]—[6]—[8]—[9]—[$\overset{q}{/}$] $\leftarrow H_u$
$H_i \to$ [/]—[4]—[$\underset{r}{/}$] $\leftarrow H_i$



Exercise 5.2 *Write the structure diagrams of quickSort(H:E2∗){ QS(H, H) } which sort C2L H increasingly. QS(p, s:E2∗) can be a recursive procedure sorting sublist (p, s) with quicksort. It starts with partitioning sublist (p, s). Let the pivot (∗q) be the first element of the sublist. Let your version of quicksort be stable.*

Illustration of the partition part of QS(H, H) on C2L H below:
(Partition of the sublist strictly between p and s, i.e. partition of sublist (p, s). The pivot is $*q$, r goes on sublist (q, s), elements smaller than the pivot are moved before the pivot.)

$H \to \overset{p}{[/]}—[\underset{q}{5}]—[\underset{r}{2}]—[4]—[6]—[5]—[2']—\overset{s}{[/]} \leftarrow H$

$H \to \overset{p}{[/]}—[2]—[\underset{q}{5}]—[\underset{r}{4}]—[6]—[5]—[2']—\overset{s}{[/]} \leftarrow H$

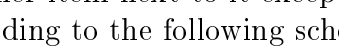
$H \to \overset{p}{[/]}—[2]—[4]—[\underset{q}{5}]—[\underset{r}{6}]—[5]—[2']—\overset{s}{[/]} \leftarrow H$

$H \to \overset{p}{[/]}—[2]—[4]—[\underset{q}{5}]—[6]—[\underset{r}{5}]—[2']—\overset{s}{[/]} \leftarrow H$

$H \to \overset{p}{[/]}—[2]—[4]—[\underset{q}{5}]—[6]—[5]—[\underset{r}{2'}]—\overset{s}{[/]} \leftarrow H$

$H \to \overset{p}{[/]}—[2]—[4]—[2']—[\underset{q}{5}]—[6]—[5]—\overset{s}{[/]} \leftarrow H$

6 Trees, binary trees

Up till now, we have worked with arrays and linked lists. They have a common property: There is a first item in the data structure, and any element has exactly one other item next to it except for the last element, which has no successor, according to the following scheme:  .

Therefore, the arrays and linked lists are called *linear data structures.* (Although in the case of cyclic lists, the linear data structure is circular.) Thus, the arrays and linked lists represent linear graphs.

6.1 General notions

The *trees* are also unique finite graphs at an abstract level. They consist of nodes (i.e. vertices) and edges (i.e. arcs). If the tree is empty, it contains neither node nor edge. $\oslash$ denotes the empty tree. If it is nonempty, it always contains a *root node.*[2] Each node of the graph has zero or more immediate successor nodes, which are called its *children*, and it is called their *parent.* A directed edge goes from the parent to each of its children. A node with no child is called a *leaf.* The root has no parent. Each other node of the tree has exactly one parent. The non-leaf nodes are also called *internal nodes.*

Notice that a linear data structure can be considered a tree where each internal node has a single child. Such trees will be called *linear trees.*

The *descendants* of a node are its children and the descendants of its children. The *ancestors* of a node are its parent and the ancestors of its parent. The root (node) has no ancestor, but each other node of the tree is a descendant of it. Thus, the root is the ancestor of each non-root node in the tree. Therefore, the root of a tree identifies the whole tree. The leaves have no descendants.

Traditionally, a tree is drawn in a reversed manner: The topmost node is the root, and the edges go downwards (so the arrows at the ends of the edges are often omitted). See figure 8.
Given a tree, it is *subtree* of itself. If the tree is not empty, its other subtrees are the subtrees of the trees rooted by its children (recursively).

A tree's *size* is the number of nodes. The size of tree t is denoted by $n(t)$ or $|t|$ ($n(t) = |t|$). The number of internal nodes of t is $i(t)$. The number of leaves of t is $l(t)$. Clearly $n(t) = i(t) + l(t)$ for a tree. For example, $n(\oslash) = 0$, and considering figure 8, $n(t_1) = i(t_1) + l(t_1) = 1 + 2 = 3$,

[2] In this chapter the trees are always *rooted, directed trees.* We do not consider *free trees*, undirected connected graphs without cycles.

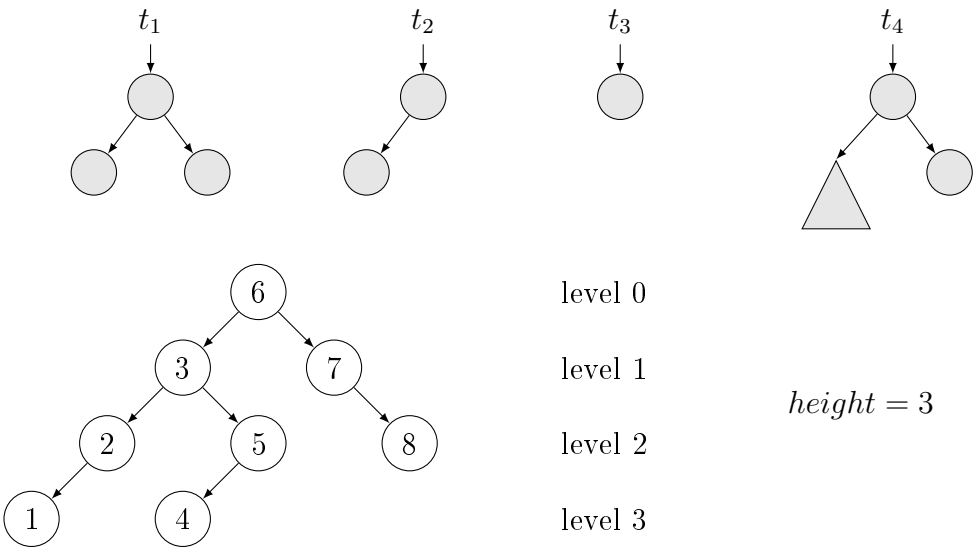


Figure 8: Simple binary trees. Circles represent the nodes of the trees. If the structure of a subtree is not important or unknown, we denote it with a triangle.

$n(t_2) = i(t_2) + l(t_2) = 1 + 1 = 2$, $n(t_3) = i(t_3) + l(t_3) = 0 + 1 = 1$, and $n(t_4) = 2 + n(t_4{\rightarrow}left)$ where "$t_4{\rightarrow}left$" is the unknown left subtree of t_4.

We can speak of a nonempty tree's *levels*. The root is at level zero (the topmost level), its children are at level one, its grandchildren are at level two, and so on. Given a node at level i, its children are at level $i+1$ (always in a downward direction).

A nonempty tree's *height* equals the leaves' lowest level. The height of the empty tree is -1. The height of tree t is denoted by $h(t)$. For example, $h(\oslash) = -1$, and considering figure 8, $h(t_1) = 1$, $h(t_2) = 1$, $h(t_3) = 0$, and $h(t_4) = 1 + max(h(t_4{\rightarrow}left), 0)$ where "$t_4{\rightarrow}left$" is the unknown left subtree of t_4.

All the hierarchical structures can be modelled by trees, for example, the directory hierarchy of a computer. Each tree node is typically labelled by some *key* and maybe with other data.

Sources: [1] Chapter 10, 6, 12; [6] 4.4; [3] 6-7

6.2 Binary trees

Binary trees are helpful, for example, for representing sets and multisets (i.e. bags) like dictionaries and priority queues.

A *binary tree* is a tree where each internal (i.e. non-leaf) node has at most two children. If a node has two children, they are the $left$ child and the $right$ child of their parent. If a node has exactly one child, then this child is the $left$ or $right$ child of its parent. This distinction is essential.

If t is a nonempty binary tree (i.e. $t \neq \oslash$), then $*t$ is the root node of the tree, $t \to key$ is the key labelling $*t$, $t \to left$ is the left subtree of t, and $t \to right$ is the right subtree of t. If $*t$ does not have the left child, $t \to left = \oslash$, i.e. the left subtree of t is empty. Similarly, if $*t$ does not have the right child, $t \to right = \oslash$, i.e. the right subtree of t is empty.

If $*p$ is a node of a binary tree, p is the (sub)tree rooted by $*p$, thus $p \to key$ is its key, $p \to left$ is its left subtree, and $p \to right$ is its right subtree. If $*p$ has the left child, it is $*p \to left$. If $*p$ has the right child, it is $*p \to right$. If $*p$ has a parent, it is $*p \to parent$. The tree rooted by the parent is $p \to parent$. (Notice that the infix operator $\to$ binds stronger than the prefix operator $*$. For example, $*p \to left = *(p \to left)$.) If $*p$ does not have a parent, $p \to parent = \oslash$.

If $p = \oslash$, all the expressions $*p$, $p \to key$, $p \to left$, $p \to right$, $p \to parent$ etc. are erroneous.

Properties 6.1

$$h(\oslash) = -1$$

$$t \neq \oslash \text{ binary tree} \Longrightarrow h(t) = 1 + max(h(t \to left), h(t \to right))$$

A *strictly binary tree* is a binary tree where each internal node has two children. (*Strictly binary trees* are also called *full binary trees*.) The following property can be proved easily by induction on $i(t)$.

Property 6.2 *Given a $t \neq \oslash$ strictly binary tree,*
$l(t) = i(t) + 1$.
Thus $n(t) = 2i(t) + 1 \wedge n(t) = 2l(t) - 1$

A *perfect binary tree* is a strictly binary tree with all the leaves at the same level. It follows that in a perfect nonempty binary tree of height h, we have 2^0 nodes at level 0, 2^1 nodes at level 1, and 2^i nodes at level i $(0 \leq i \leq h)$. Therefore, we get the following property.

Property 6.3 *Given a $t \neq \oslash$ perfect binary tree ($n = n(t), h = h(t)$)*
$n = 2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$

A nonempty *nearly complete binary tree* is a binary tree which becomes perfect if we delete its lowest level. The $\oslash$ is also a nearly complete binary tree. (`An equivalent definition`: A *nearly complete binary tree* is a binary tree which can be received from a perfect binary tree by deleting zero or more nodes from its lowest level.) Notice that the perfect binary trees are also nearly complete, according to this definition.

Because a nonempty, nearly complete binary tree is perfect, possibly except at its lowest level h, it has $2^h - 1$ nodes at its first $h-1$ levels, at least 1 node and at most 2^h nodes at its lowest level. Thus $n \geq 2^h - 1 + 1 \wedge n \leq 2^h - 1 + 2^h$ where n is the size of the tree.

Properties 6.4 *Given a* $t \neq \oslash$ *nearly complete binary tree* $(n = n(t), h = h(t))$. *Then*
$n \in 2^h..(2^{h+1}-1)$. *Thus*
$h = \lfloor \log n \rfloor$.

Now, we consider the relations between the size n and height h of a nonempty binary tree. These will be important in binary search trees: good representations of data sets stored in the RAM.

A nonempty binary tree of height h has the most nodes if it is perfect. Therefore, from property 6.3, we receive that for the size n of any nonempty binary tree $n < 2^{h+1}$. Thus $\log n < \log(2^{h+1}) = h + 1$,[3] so $\lfloor \log n \rfloor < h + 1$. Finally, $\lfloor \log n \rfloor \leq h$.

On the other hand, a nonempty binary tree of height h has the least number of nodes if it contains just one node at each level (i.e. it is linear), and so for the size n of any nonempty binary tree $n \geq h+1$. Thus $h \leq n-1$.

Property 6.5 *Given a* $t \neq \oslash$ *binary tree* $(n = n(t), h = h(t))$,
$\lfloor \log n \rfloor \leq h \leq n - 1$
where $h = \lfloor \log n \rfloor$ *if the tree is nearly complete, and*
$h = n - 1$ *iff the tree is linear (see page 45).*

Notice that there are binary trees with $h = \lfloor \log n \rfloor$, although they are not nearly complete.

6.3 Linked representations of binary trees

The empty tree is represented by a null pointer, i.e. $\oslash$. A nonempty binary tree is identified by a pointer referring to a **Node**, which represents the root of the tree:

[3] Remember the definition of $\log n$ given in section 1: $\log n = log_2(n)$ if $n > 0$, and $\log 0 = 0$.

Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $left, right$: Node*
+ Node() { $left := right := \oslash$ } // generate a tree of a single node. + Node($x : \mathcal{T}$) { $left := right := \oslash$; $key := x$ }

Sometimes, having a parent pointer in the tree's nodes is helpful. We can see a binary tree with parent pointers in figure 9.

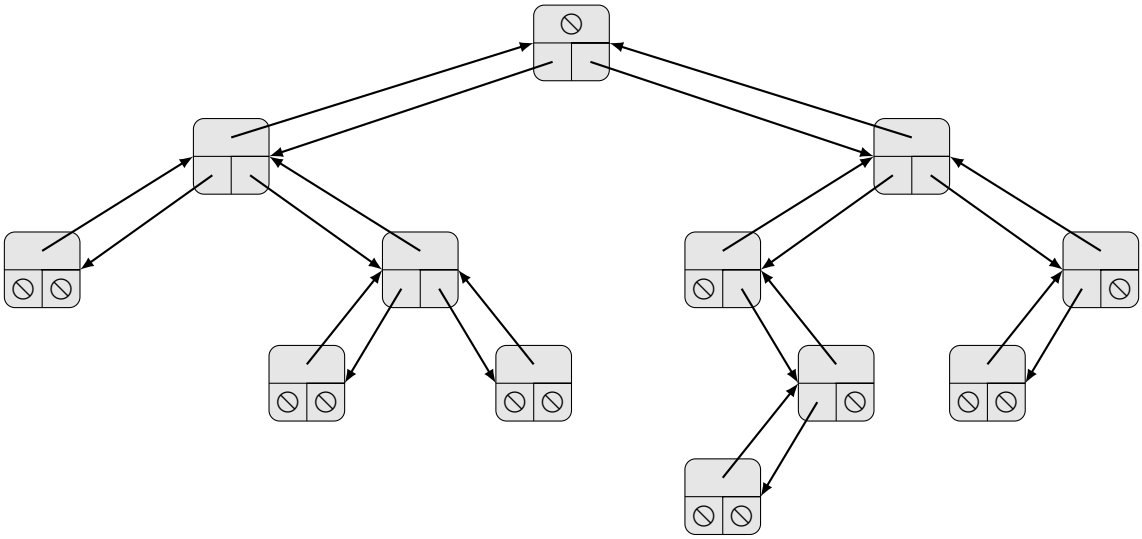
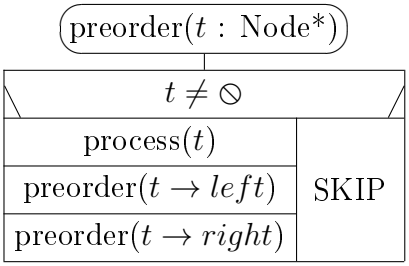
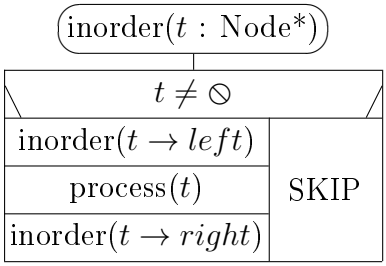


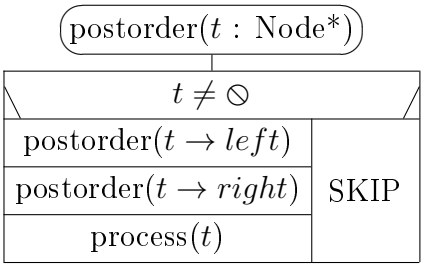
Figure 9: Binary tree with parent pointers. (We omitted the keys of the nodes here.)

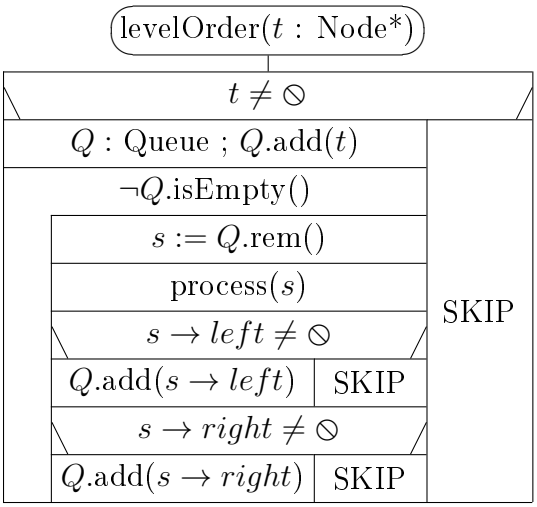
Node3
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $left, right, parent$: Node3*
+ Node3(p:Node3*) { $left := right := \oslash$; $parent := p$ } + Node3($x : \mathcal{T}$, p:Node3*) { $left := right := \oslash$; $parent := p$; $key := x$ }

6.4 Binary tree traversals

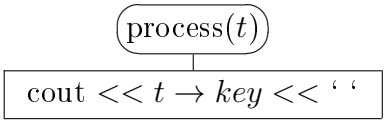




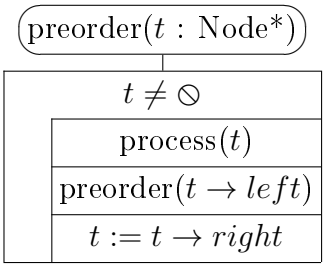


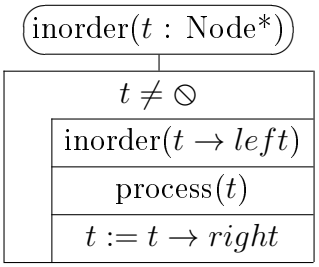


$T_{\text{preorder}}(n), T_{\text{inorder}}(n), T_{\text{postorder}}(n), T_{\text{levelOrder}}(n) \in \Theta(n)$ where $n = n(t)$ and the time complexity of process(t) is $\Theta(1)$. For example, process(t) can print $t \to key$:



We can slightly reduce preorder and inorder traversal running times if we eliminate the tail recursions. (It does not affect the time complexities.)





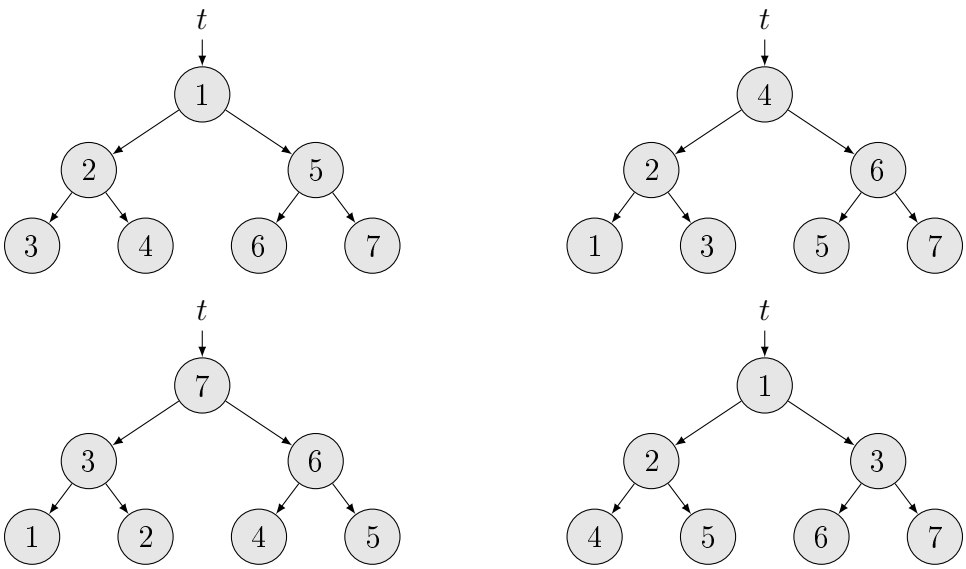
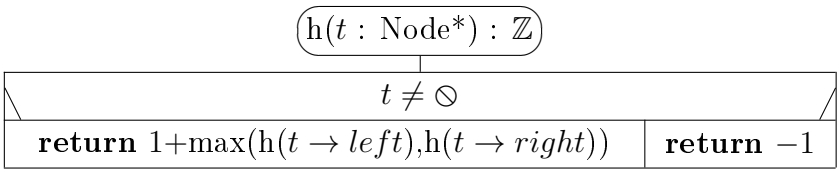


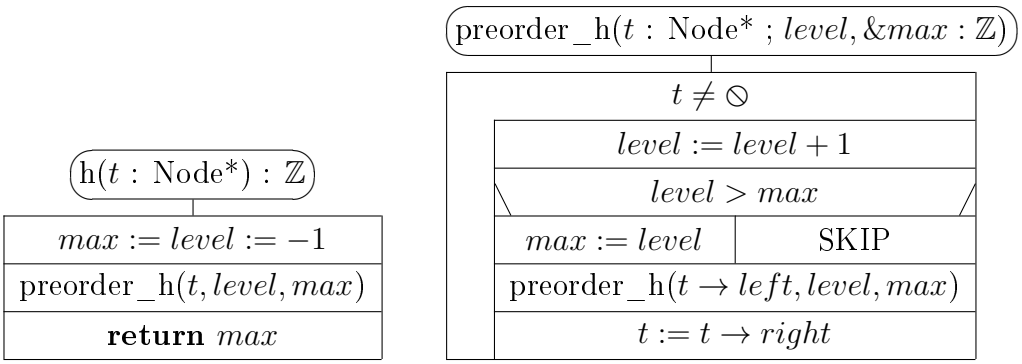
Figure 10: left upper part: preorder, right upper part: inorder, left lower part: postorder, right lower part: level order traversal of binary tree t.

6.4.1 An application of traversals: the height of a binary tree

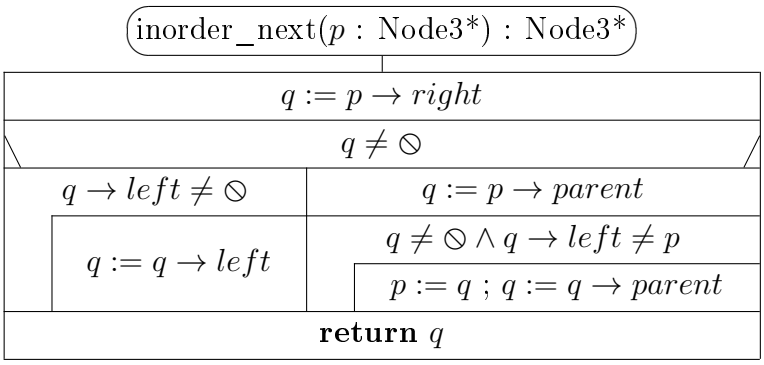
Postorder:



Preorder:



6.4.2 Using parent pointers



$MT(h) \in O(h)$ where $h = h(t)$

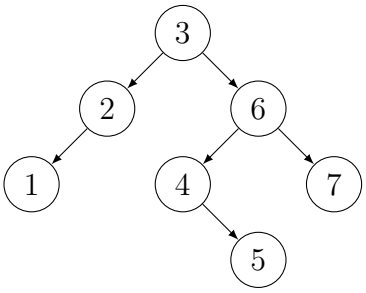
Exercise 6.6 *What can we say about the space complexities of the different binary tree traversals and other subroutines above?*

6.5 Parenthesised, i.e. textual form of binary trees

Parenthesised, i.e. the textual form of a nonempty binary tree:

(LeftSubtree Root RightSubtree)

A pair of brackets represents the empty tree. We omit the empty subtrees of the leaves. We can use different kinds of parentheses for easier reading. For example, see Figure 11.



The binary tree on the left in

- simple textual form:
  (((1) 2 ()) 3 ((() 4 (5)) 6 (7)))
- elegant parenthesised form:
  { [(1) 2 ()] 3 [(⟨⟩ 4 ⟨5⟩) 6 (7)] }

Figure 11: The same binary tree in graphical and textual representations. Notice that by omitting the parenthesis from the textual form of a tree, we receive the inorder traversal of that tree.

6.6 Binary search trees

»*Binary search trees* (BST) are a particular type of container: data structures that store "items" (such as numbers, names, etc.) in memory. They allow fast lookup, addition and removal of items. They can be used to implement either dynamic sets of items or lookup tables that allow finding an item by its key (e.g., locating the phone number of a person by name).

Binary search trees keep their keys in sorted order so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, each comparison allows the operations to skip about half of the tree so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array but slower than the corresponding operations on hash tables.«

(The source of the text above is Wikipedia.)

A *binary search tree* (BST) is a binary tree whose nodes each store a *key* (and, optionally, some associated value). The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than any key stored in the left subtree and less than any key stored in the right subtree. (See Figure 12. Notice that there is no duplicated key in a BST.) A *binary sort tree* is similar to a BST but may have equal keys. The key in each node must be greater than or equal to any key stored in the left subtree and less than or equal to any key stored in the right subtree. In this Lecture Notes, all the subsequent structure diagrams refer to BSTs.

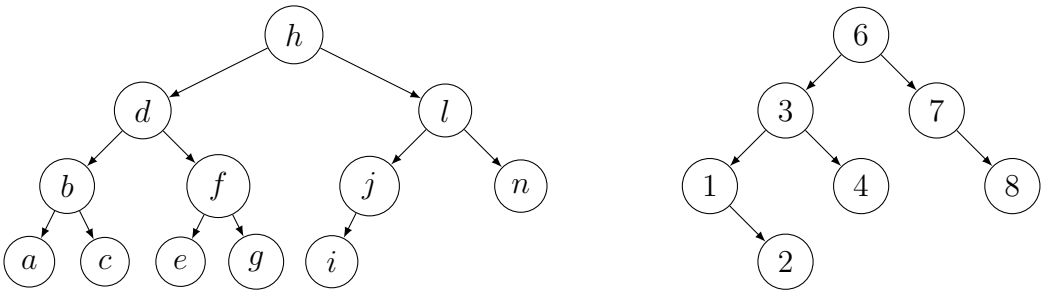
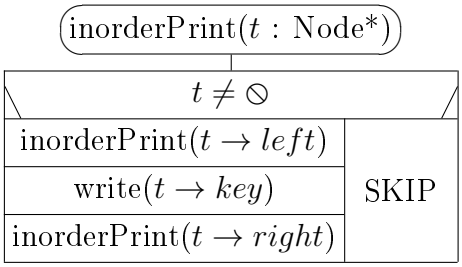
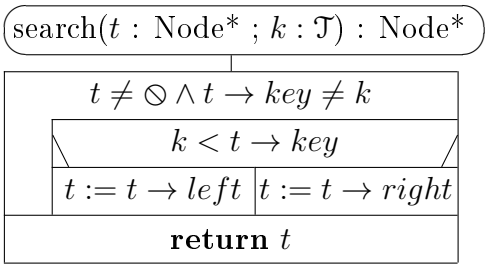
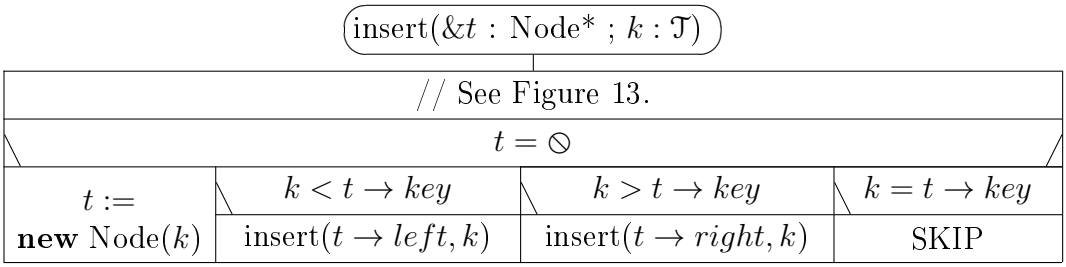


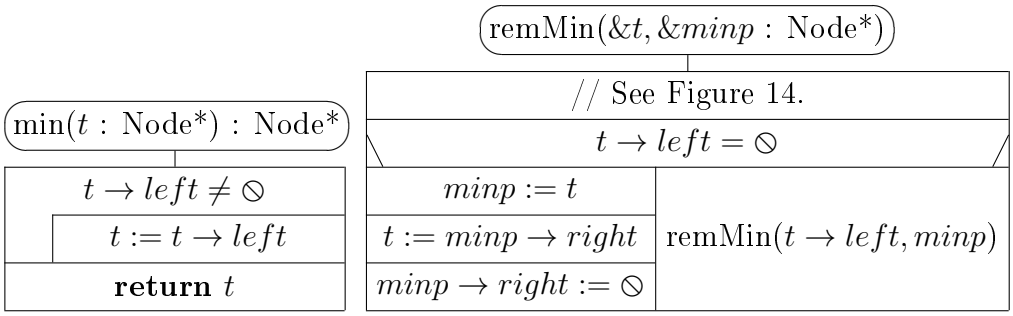
Figure 12: Two *binary search trees* (BSTs)



Property 6.7 *Procedure inorderPrint(t: Node*) prints the keys of the binary tree t in strictly increasing order,* $\iff$ *binary tree t is a search tree.*







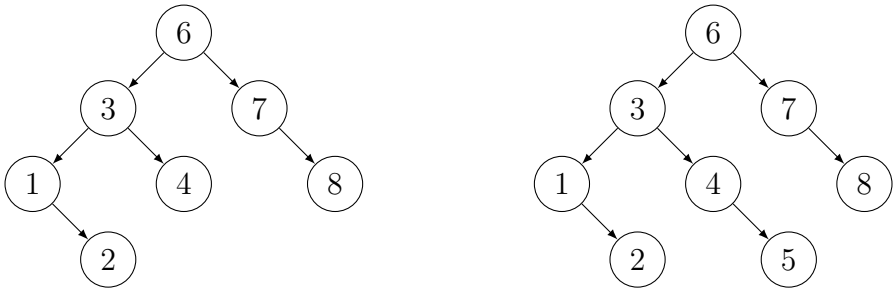


Figure 13: Inserting 5 into the BST on the left gives the BST on the right: First we compare 5 with the root key, which is 6, and $5 < 6$. Thus, we insert 5 into the left subtree. Next, we compare 5 with 3, and $5 > 3$, so we insert 5 into the right subtree of node 3, and again $5 > 4$ therefore we insert 5 into the right subtree of node 4. The right subtree of this node is empty. Consequently, we create a new node with key 5 and substitute that empty right subtree with this new subtree consisting of this single new node.

del(&t : Node* ; $k : \mathcal{T}$)

// See Figure 16.			
$t \neq \oslash$			
$k < t \to key$	$k > t \to key$	$k = t \to key$	SKIP
del($t \to left, k$)	del($t \to right, k$)	delRoot(t)	

delRoot(&t : Node*)

// See Figure 17.		
$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	remMin($t \to right, p$)
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$; $p \to right := t \to right$
delete p	**delete** p	delete t ; $t := p$

$MT_{\text{search}}(h), MT_{\text{insert}}(h), MT_{\min}(h) \in \Theta(h)$
$MT_{\text{remMin}}(h), MT_{\text{del}}(h), MT_{\text{delRoot}}(h) \in \Theta(h)$
where $h = h(t)$.

Exercise 6.8 *What can we say about the space complexities of the different BST operations above?*

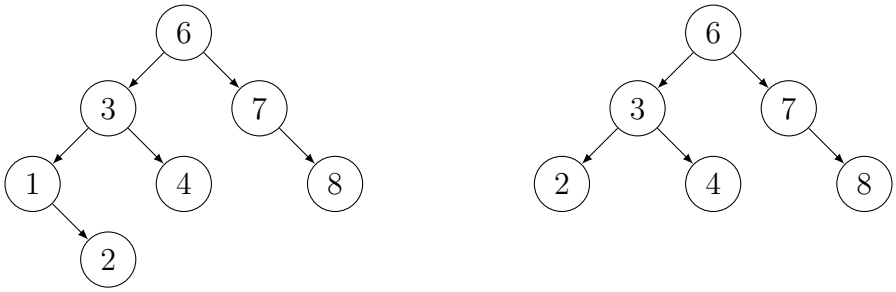


Figure 14: Removing the node with the minimal key from the BST on the left gives the BST on the right: The leftmost node of the original BST is substituted by its right subtree. Notice that deleting the node with key 1 from the BST on the left gives the same BST result: This node's left subtree is empty. Thus, deleting it means substituting it with its right subtree.

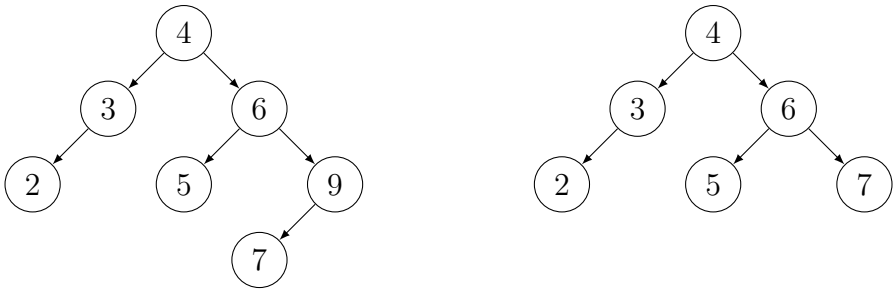


Figure 15: Removing the node with the maximal key from the BST on the left gives the BST on the right: The rightmost node of the original BST is substituted by its left subtree. Notice that deleting the node with key 9 from the BST on the left gives the same BST result: This node's right subtree is empty. Thus, deleting it means substituting it with its left subtree.

6.7 Complete binary trees, and heaps

A binary tree is *complete*, **iff** all levels of the tree, except possibly the last (deepest) one, are filled (i.e. it is nearly complete), and, provided that the deepest level of the tree is not complete, the nodes of that level are filled from left to right. A node is an internal node of a *complete* tree if and only if it has the left child in the tree.

A *binary maximum heap* is a complete binary tree where the key stored in each node is greater than or equal to ($\geq$) the keys in the node's children. (See Figure 18.)

A *binary minimum heap* is a complete binary tree where the key stored in each node is less than or equal to ($\leq$) the keys in the node's children.

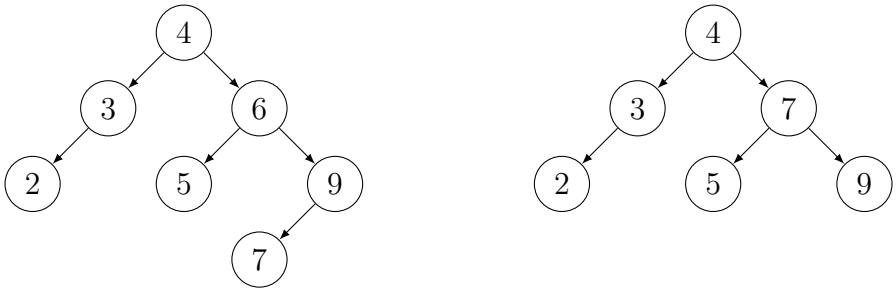


Figure 16: Deleting node 6 of the BST on the left gives the BST on the right: Node 6 has two children. Thus, we remove the minimal node of its right subtree to substitute node 6 with it.

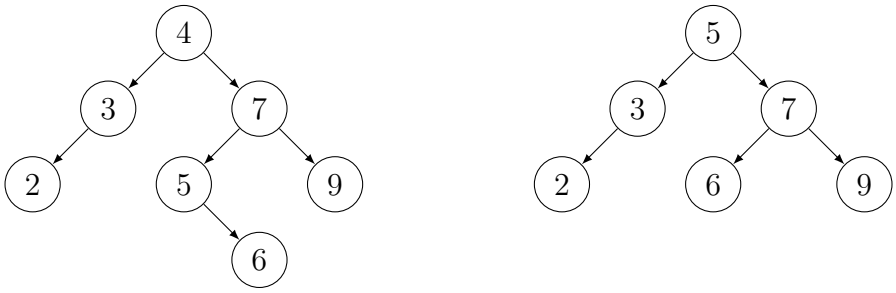


Figure 17: Deleting the root node of the BST on the left gives the BST on the right: The root node has two children. Thus, we remove the minimal node of its right subtree and substitute the root node with it. (Notice that we could remove the maximal node of the left subtree and substitute the root node with it.)

In this lecture notes, a *heap* is a *binary maximum heap* by default.

We usually represent priority queues with heaps in arithmetic representation.

6.8 Arithmetic representation of complete binary trees

A complete binary tree of size n is typically represented by the first n elements of an array A: We put the tree nodes into the array in level order. (See Figure 19.)

If an internal node has index i in array A, let $left(i)$ be the index of its left child. If it has the right child, too, let $right(i)$ be the index of it. In the latter case, $right(i) = left(i) + 1$. If $A[j]$ represents a node different from the root, Let $parent(j)$ be the index of its parent.

Let us suppose that we use the first n elements of an array (indexed from

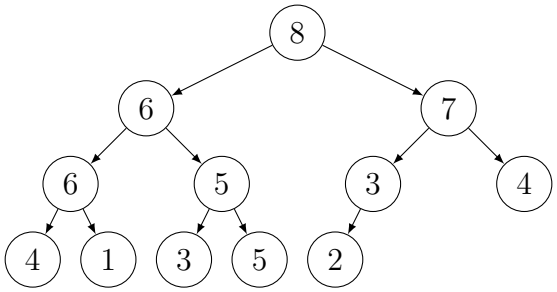


Figure 18: A *(binary maximum) heap*. It is a complete binary tree. The key of each parent is $\geq$ to the child's key.

zero), for example, $A : \mathcal{T}[m]$, i.e. we use the subarray $A[0..n)$. Then, the root node of a nonempty tree is $A[0]$. Node $A[i]$ is internal node $\iff left(i) < n$. Then $left(i) = 2i + 1$. The right child of node $A[i]$ exists $\iff right(i) < n$. Then $right(i) = 2i + 2$. Node $A[j]$ is not the tree's root node $\iff j > 0$. Then $parent(j) = \lfloor \frac{j-1}{2} \rfloor$.

6.9 Heaps and priority queues

A priority queue is a bag (i.e. multiset). We can add a new item to it and check or remove a maximal item of it.[4] Often, there is a priority function: $\mathcal{T} \to$ some number type, where $\mathcal{T}$ is the element type of the priority queue, and the items are compared according to their priorities. For example, computer processes are often scheduled according to their priorities.

Our representation is similar to our representation of the Stack type, and some operations are also the same, except for the names. The actual elements of the priority queue are in the subarray $A[0..n)$ containing a (binary maximum) heap.

[4]There are also min priority queues where we can check or remove a minimal item.

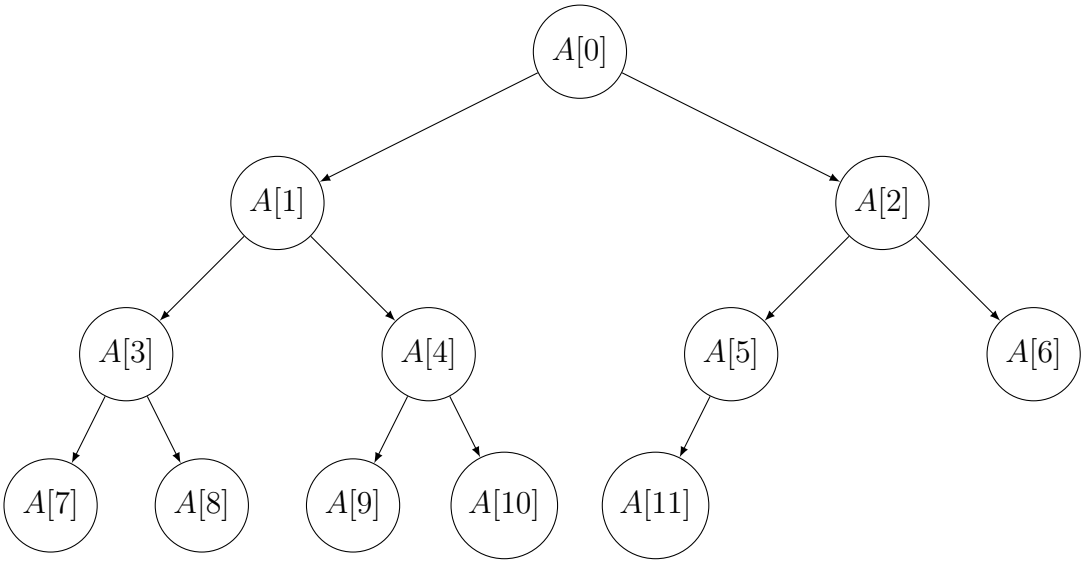


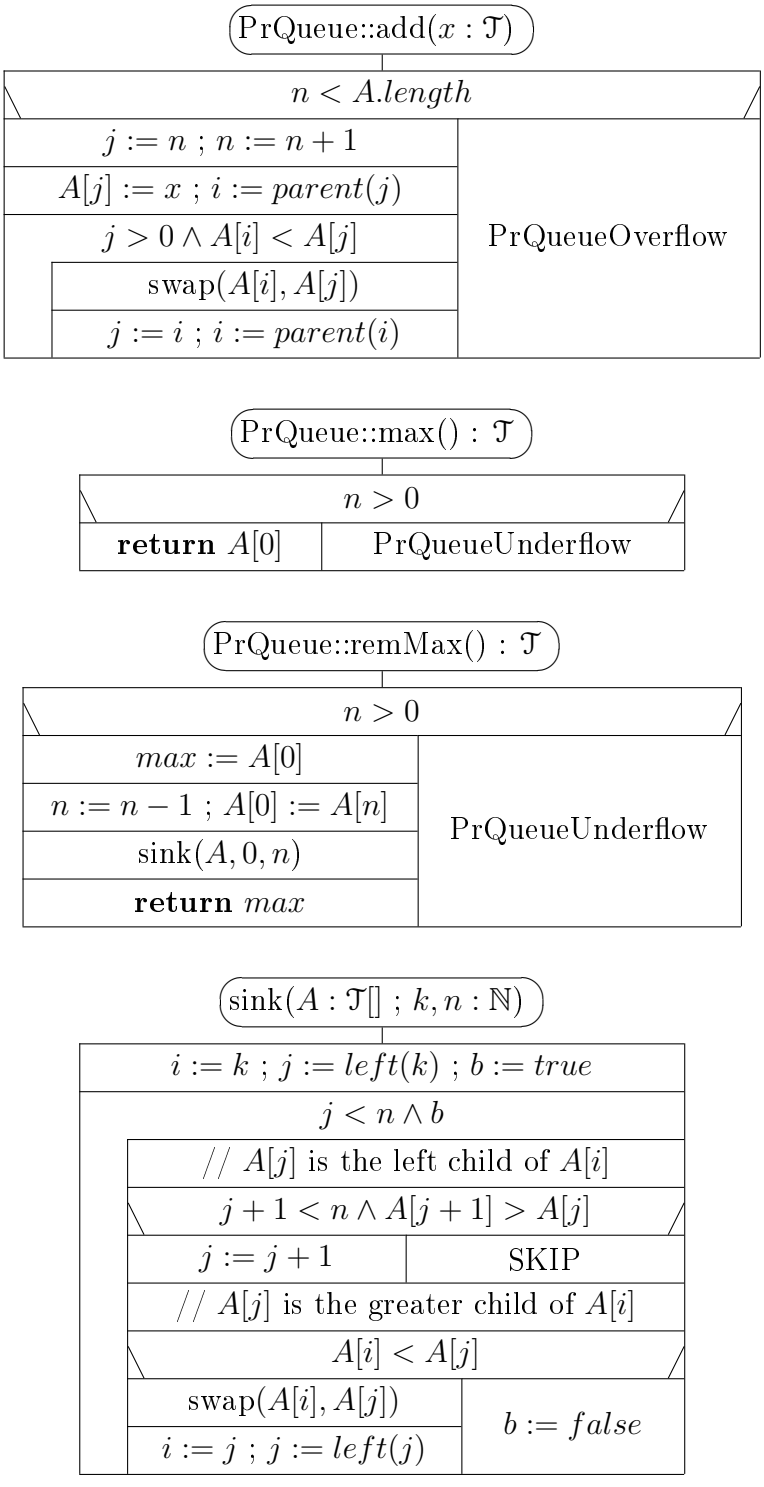
Figure 19: A complete binary tree of size 12 represented by the first 12 elements of an array A: We put the tree nodes into the array in level order.

PrQueue
$-$ $A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type $-$ $n : \mathbb{N}$ // $n \in 0..A.length$ is the actual length of the priority queue

+ PrQueue($m : \mathbb{N}$){ $A :=$ **new** $\mathcal{T}[m]$; $n := 0$ } // create an empty priority queue
+ add($x : \mathcal{T}$) // insert x into the priority queue
+ remMax():$\mathcal{T}$ // remove and return the maximal element of the priority queue
+ max():$\mathcal{T}$ // return the maximal element of the priority queue
+ isFull() : $\mathbb{B}$ {**return** $n = A.length$}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ $\sim$ PrQueue() { **delete** A }
+ setEmpty() {$n := 0$} // reinitialize the priority queue

Provided that subarray $A[0..n)$ is the arithmetic representation of a heap, $MT_{\text{add}}(n) \in \Theta(\log n)$, $mT_{\text{add}}(n) \in \Theta(1)$, $MT_{\text{remMax}}(n) \in \Theta(\log n)$, $mT_{\text{remMax}}(n) \in \Theta(1)$, $T_{\max}(n) \in \Theta(1)$.

$MT_{\text{add}}(n) \in \Theta(\log n)$, and $MT_{\text{remMax}}(n) \in \Theta(\log n)$, because the main loops of subroutines “add" and “sink" below iterates maximum h times if h is the height of the heap, and $h = \lfloor \log n \rfloor$.



Exercise 6.9 *What can we say about the space complexities of the different heap and priority queue operations above?*

op	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-	8	6	7	6	5	3	4	4	1	3	5	2			
add(8)	8	6	7	6	5	*3	4	4	1	3	5	2	#8		
...	8	6	*7	6	5	#8	4	4	1	3	5	2	3		
...	*8	6	#8	6	5	7	4	4	1	3	5	2	3		
.	8	6	8	6	5	7	4	4	1	3	5	2	3		
add(2)	8	6	8	6	5	7	*4	4	1	3	5	2	3	#2	
.	8	6	8	6	5	7	4	4	1	3	5	2	3	2	
add(9)	8	6	8	6	5	7	*4	4	1	3	5	2	3	2	#9
...	8	6	*8	6	5	7	#9	4	1	3	5	2	3	2	4
...	*8	6	#9	6	5	7	8	4	1	3	5	2	3	2	4
.	9	6	8	6	5	7	8	4	1	3	5	2	3	2	4
remMax()	∼9	6	8	6	5	7	8	4	1	3	5	2	3	2	∼4
$max := 9$	*4	6	#8	6	5	7	8	4	1	3	5	2	3	2	
...	8	6	*4	6	5	7	#8	4	1	3	5	2	3	2	
...	8	6	8	6	5	7	*4	4	1	3	5	2	3	#2	
return 9	8	6	8	6	5	7	4	4	1	3	5	2	3	2	
remMax()	∼8	6	8	6	5	7	4	4	1	3	5	2	3	∼2	
$max := 8$	*2	6	#8	6	5	7	4	4	1	3	5	2	3		
...	8	6	*2	6	5	#7	4	4	1	3	5	2	3		
...	8	6	7	6	5	*2	4	4	1	3	5	2	#3		
return 8	8	6	7	6	5	3	4	4	1	3	5	2	2		

Figure 20: Changes of $A : \mathbb{Z}[15]$ while applying add() and remMax() operations to it. At the add() operations, the "#" prefix identifies the actual element, and "$*$" is the prefix of its parent. Similarly, at sinking, "$*$" denotes the parent, and "#" is the prefix of its greatest child. At the remMax() operations, "$\sim$" is the prefix of both the maximum to be removed and the key to be moved into its place.

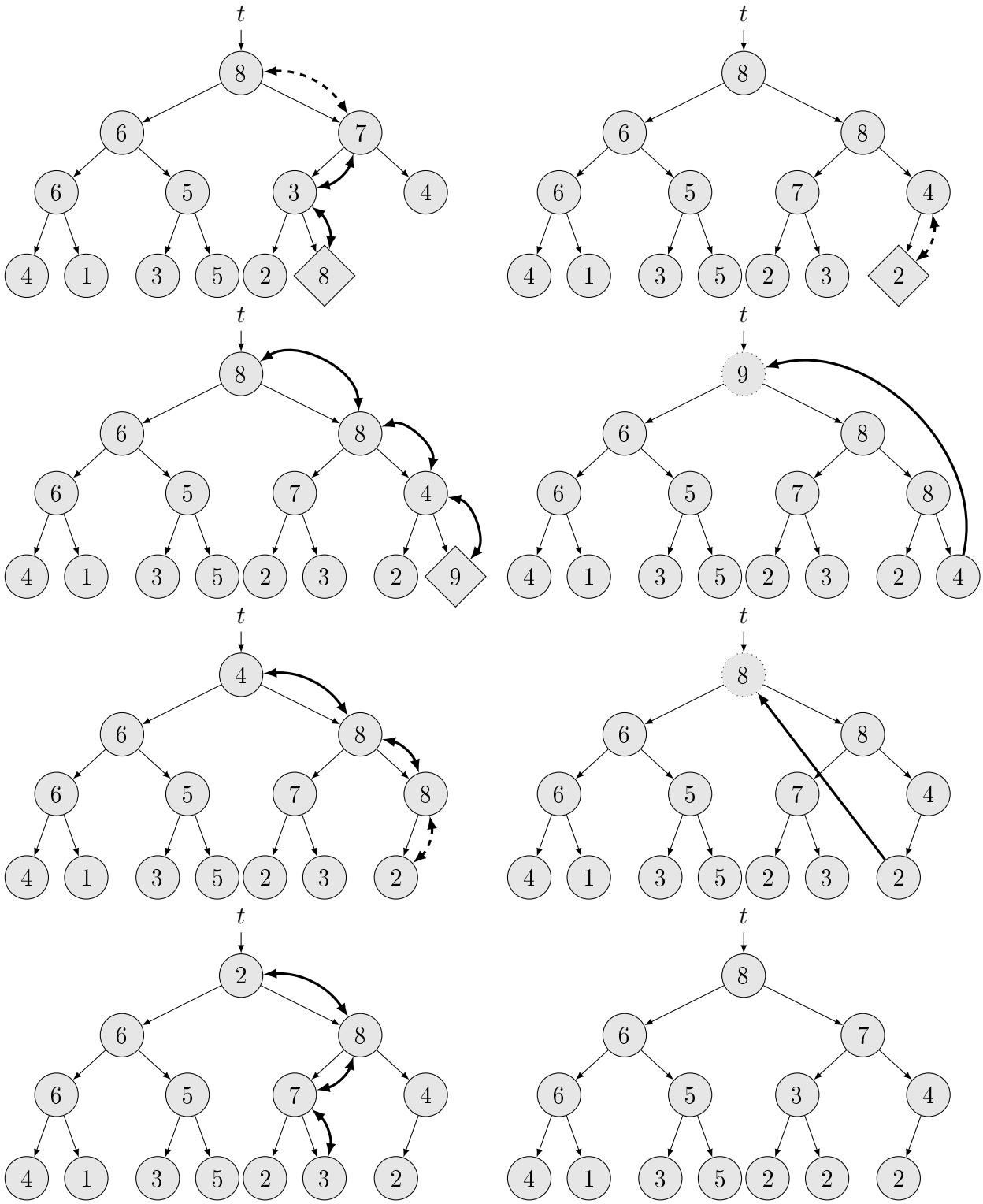
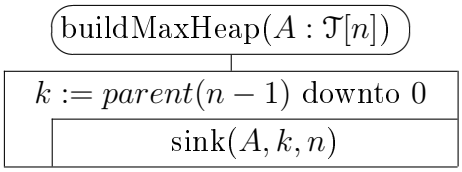


Figure 21: Heap operation visualization based on Figure 20:
add(8), add(2), add(9), remMax()=9, remMax()=8.

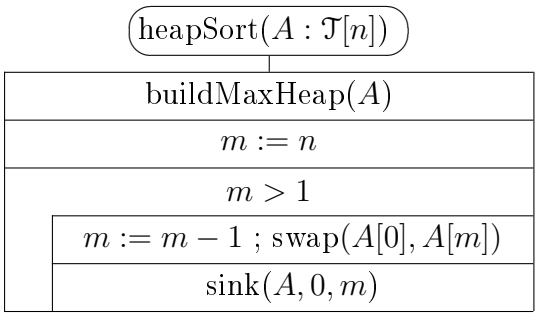
6.10 Heap sort

In *Heap sort*, first, we take the array to be sorted and build a heap from it. We consider the array as a complete binary tree.

While building the heap, we consider the level order of the tree, and we start from the last internal node of it. We go back in level order and sink the root node of each subtree, making a heap out of it. (See Figure 22.)



When the heap is ready, we swap its first and last items. Then, the last element is the maximal element of the array. We cut it from the tree. Next, we sink the tree's root and receive a heap again. Again, we swap, cut and sink, and we have the two largest elements at the end of the array and a heap without these elements before them. We repeat this process of *swap, cut and sink* until we have only a single element in the heap, which is the minimum of the original array. And the whole array is sorted at this moment. (See Figure 24.)



The time complexity of each sinking is $O(\log n)$ because $h \leq \lfloor \log n \rfloor$ for each subtree. Thus, $MT_{\text{buildMaxHeap}}(n) \in O(n \log n)$ and the time complexity of the main loop of heapSort() is also $O(n \log n)$.

$MT_{\text{heapSort}}(n) \in O(n \log n)$ because $MT_{\text{buildMaxHeap}}(n) \in O(n \log n)$ and the time complexity of the main loop of heapSort() is also $O(n \log n)$.

In chapter 7, we prove that $MT_{\text{heapSort}}(n) \in \Theta(n \log n)$.

Heap sort has a space complexity of $\Theta(1)$ since it is not recursive and uses no temporal data structures, only a few simple temporal variables. Thus, *Heap sort* sorts in-place like *Insertion sort*.

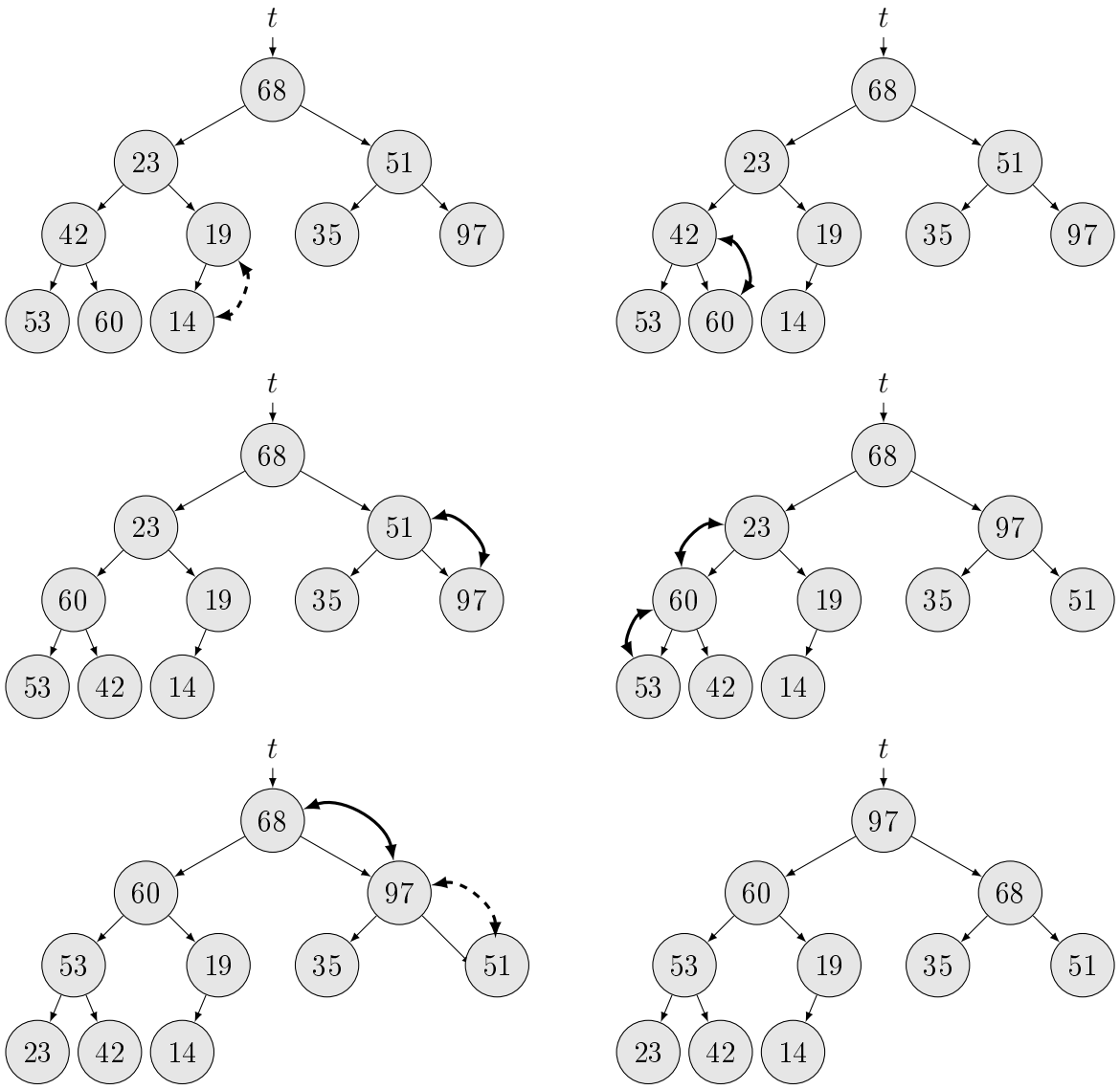


Figure 22: From array $\langle 68, 23, 51, 42, 19, 35, 97, 53, 60, 14\rangle$ we build a maximum heap. Notice that we work on the array from the beginning to the end. The binary trees show the logical structure of the array. Finally, we receive array $\langle 97, 60, 68, 53, 19, 35, 51, 23, 42, 14\rangle$. Dashed arrows compare where the sinking ends because no swap is needed.

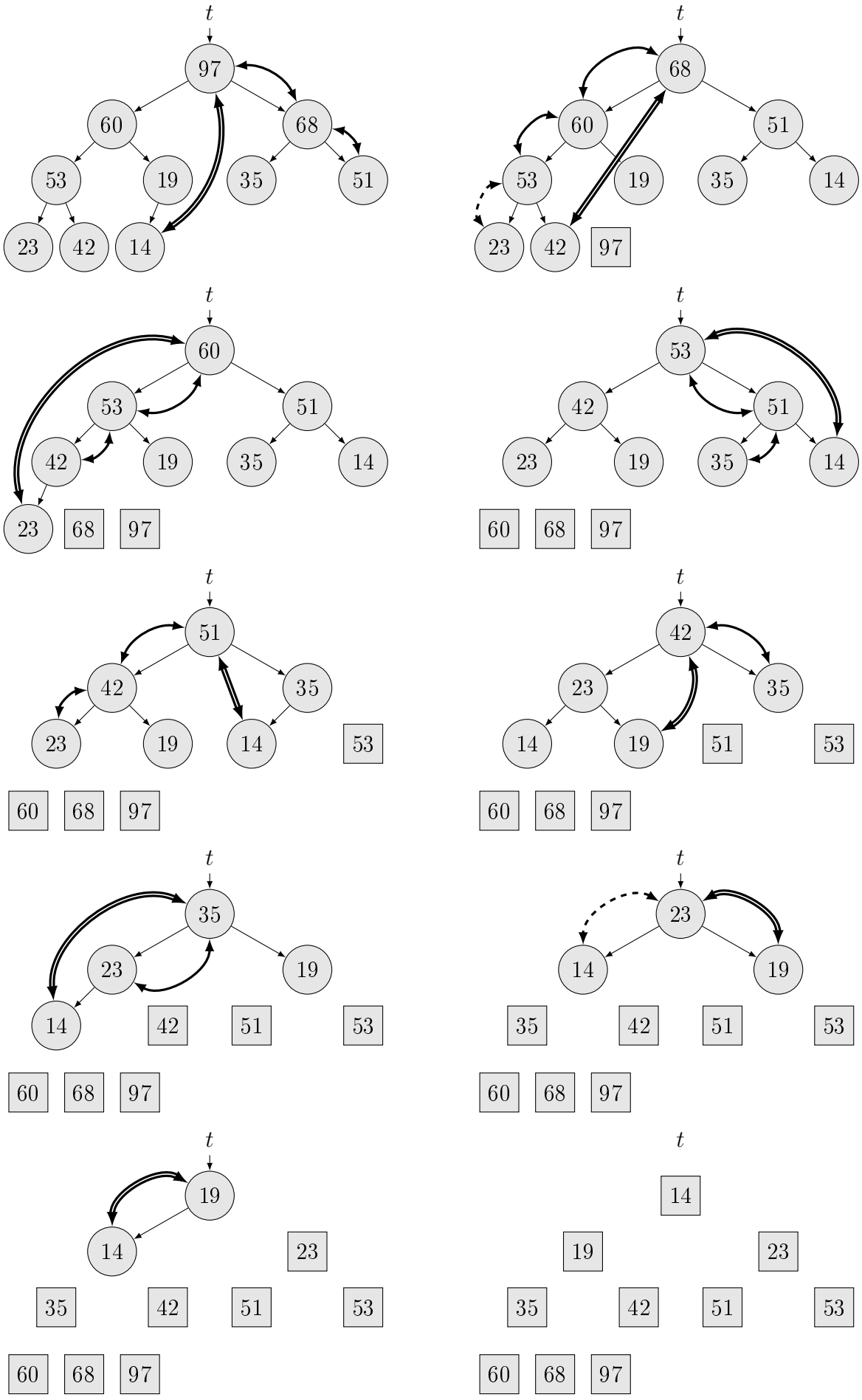


Figure 23: Heap sort visualization based on Figure 24. Double arrows represent the swap of the largest element with the last element of the heap, fitting into its desired place. The subsequent arrows demonstrate the sinking process after the initial swap. Dashed arrows compare where the sinking ends because no swap is needed.

op	0	1	2	3	4	5	6	7	8	9
sink	68	23	51	42	*19	35	97	53	60	#14
sink	68	23	51	*42	19	35	97	53	#60	14
sink	68	23	*51	60	19	35	#97	53	42	14
sink	68	*23	97	#60	19	35	51	53	42	14
...	68	60	97	*23	19	35	51	#53	42	14
sink	*68	60	#97	53	19	35	51	23	42	14
...	97	60	*68	53	19	35	#51	23	42	14
.	97	60	68	53	19	35	51	23	42	14
swap	∼97	60	68	53	19	35	51	23	42	∼14
sink	*14	60	#68	53	19	35	51	23	42	+97
...	68	60	*14	53	19	35	#51	23	42	+97
swap	∼68	60	51	53	19	35	14	23	∼42	+97
sink	*42	#60	51	53	19	35	14	23	+68	+97
...	60	*42	51	#53	19	35	14	23	+68	+97
.	60	53	51	*42	19	35	14	#23	+68	+97
swap	∼60	53	51	42	19	35	14	∼23	+68	+97
sink	*23	#53	51	42	19	35	14	+60	+68	+97
...	53	*23	51	#42	19	35	14	+60	+68	+97
swap	∼53	42	51	23	19	35	∼14	+60	+68	+97
sink	*14	42	#51	23	19	35	+53	+60	+68	+97
...	51	42	*14	23	19	#35	+53	+60	+68	+97
swap	∼51	42	35	23	19	∼14	+53	+60	+68	+97
sink	*14	#42	35	23	19	+51	+53	+60	+68	+97
...	42	*14	35	#23	19	+51	+53	+60	+68	+97
swap	*42	23	35	14	#19	+51	+53	+60	+68	+97
sink	*19	23	#35	14	+42	+51	+53	+60	+68	+97
swap	∼35	23	19	∼14	+42	+51	+53	+60	+68	+97
sink	*14	#23	19	+35	+42	+51	+53	+60	+68	+97
swap	∼23	14	∼19	+35	+42	+51	+53	+60	+68	+97
sink	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
swap	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
sink	*14	+19	+23	+35	+42	+51	+53	+60	+68	+97
.	+14	+19	+23	+35	+42	+51	+53	+60	+68	+97

Figure 24: Full example of Heapsort on array $A : \mathbb{Z}[10]$. At sinking, "$*$" denotes the parent, and "#" is the prefix of its greater child. At the swap operations, "$\sim$" is the prefix of both items to be exchanged. In their final places, the keys have "+" prefixes.

7 Lower bounds for sorting

Theorem 7.1 *For any sorting algorithm,* $mT(n) \in \Omega(n)$.

Proof. We have to check all the n items, and only a limited number of items is checked in a subroutine call or loop iteration (without the embedded subroutine calls and loop iterations). Let this limit be k. Thus
$mT(n) * k \geq n \Longrightarrow mT(n) \geq \frac{1}{k}n \Longrightarrow mT(n) \in \Omega(n)$. □

7.1 Comparison sorts and the decision tree model

Definition 7.2 *A sorting algorithm is a comparison sort if it only gains information about the sorted order of the input items by comparing them. That is, given two items* a_i *and* a_j*, to compare them, it performs one of the tests* $a_i < a_j$*,* $a_i \leq a_j$*,* $a_i = a_j$*,* $a_i \neq a_j$*,* $a_i \geq a_j$*, or* $a_i > a_j$*. We may not inspect the values of the elements or gain order information about them in any other way. [1]*

The sorting algorithms we have studied before, *insertion sort, heap sort, merge sort*, and *quick sort*, are comparison sorts.

In this section, we assume without loss of generality that all the input elements are distinct[5]. Given this assumption, comparisons of the form $a_i = a_j$ and $a_i \neq a_j$ are useless, so we can assume that no comparisons of this form are made[6]. We also note that the comparisons $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, and $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \leq a_j$. [1]

We can view comparison sorts abstractly in terms of decision trees. A decision tree is a strictly binary tree representing the comparisons between elements performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. [1]

Figure 25 shows the decision tree corresponding to the insertion sort algorithm from Section 3 operating on an input sequence of three elements.

[5] In this way, we restrict the set of possible inputs, and we are going to give a lower bound for the worst case of comparison sorts. Thus, if we provide a lower bound for the maximum number of key comparisons ($MC(n)$) and for maximum time complexity ($MT(n)$) on this restricted set of input sequences, it is also a lower bound for them on the whole set of input sequences, because $MC(n)$ and $MT(n)$ are indeed $\geq$ on a more extensive set than on a smaller set.

[6] Anyway if such comparisons are made, neither $MC(n)$ nor $MT(n)$ are decreased.

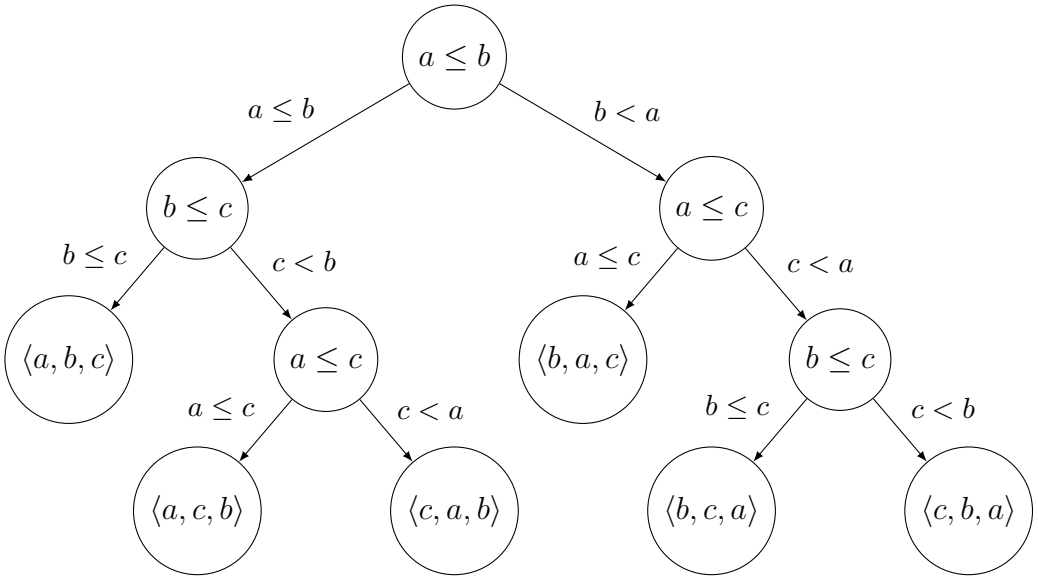


Figure 25: The decision tree for insertion sort operating on the input sequence $\langle a, b, c \rangle$. There are $3! = 6$ permutations of the 3 input items. Thus, the decision tree must have at least $3! = 6$ leaves.

Let us suppose that $\langle a_1, a_2, \ldots a_n \rangle$ is the input sequence to be sorted. Each internal node is labelled by $a_i \leq a_j$ for some input elements in a decision tree. We also annotate each leaf by a permutation of $\langle a_1, a_2, \ldots a_n \rangle$. The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \leq a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the appropriate ordering of $\langle a_1, a_2, \ldots a_n \rangle$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. Thus, we shall consider only decision trees where each permutation appears as a tree leaf. [1]

7.2 A lower bound for the worst case

Theorem 7.3 *Any comparison sort algorithm requires* $MC(n) \in \Omega(n \log n)$ *comparisons in the worst case.*

Proof. From the preceding discussion, it suffices to determine the height $h =$

$MC(n)$ of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l leaves corresponding to a comparison sort on n elements. The input has $n!$ permutations. Because each input permutation appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have $n! \leq l \leq 2^h$. [1]

Consequently

$$MC(n) = h \geq \log n! = \sum_{i=1}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log \left\lceil \frac{n}{2} \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil * \log \left\lceil \frac{n}{2} \right\rceil \geq$$

$$\geq \frac{n}{2} * \log \frac{n}{2} = \frac{n}{2} * (\log n - \log 2) = \frac{n}{2} * (\log n - 1) = \frac{n}{2} \log n - \frac{n}{2} \in \Omega(n \log n)$$

□

Theorem 7.4 *For any comparison sort algorithm* $MT(n) \in \Omega(n \log n)$.

Proof. Only a limited number of key comparisons are performed in a subroutine call or loop iteration (without the embedded subroutine calls and loop iterations). Let this upper limit be k. Then $MT(n) * k \geq MC(n) \Longrightarrow MT(n) \geq \frac{1}{k} MC(n) \Longrightarrow MT(n) \in \Omega(MC(n))$. Together with theorem 7.3 ($MC(n) \in \Omega(n \log n)$) and transitivity of relation "$\cdot \in \Omega(\cdot)$" we receive this theorem ($MT(n) \in \Omega(n \log n)$). □

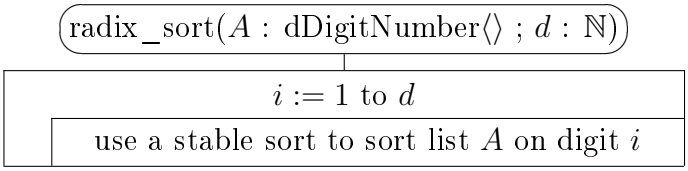
Let us notice that *heap sort* and *merge sort* are *asymptotically optimal* in the sense that their $MT(n) \in O(n \log n)$ asymptotic upper bound meets the $MT(n) \in \Omega(n \log n)$ asymptotic lower bound from Theorem 7.4. This proves that $MT(n) \in \Theta(n \log n)$ for both of them.

8 Sorting in Linear Time

It is essential to use a stable sorting method so that those numbers with equal second digits remain sorted according to their first digit.

8.1 Radix sort

The abstract code for the radix sort is straightforward. We assume that each element of abstract list A is a natural number of d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.



Radix sort solves the problem of sorting – counterintuitively – by sorting on the least significant (i.e. first) digit in its first pass.

In the second pass, it takes the result of the first pass and applies a stable sort to sort it on the second digit. A stable sorting method is essential so that numbers with equal second digits remain sorted by their first digits. Consequently, the numbers are sorted on the first two digits (the two lowest-order digits) after the second pass.

In the third pass, the Radix sort takes the result of the second pass and applies a stable sort to sort on the third digit. A stable sorting method is essential so that numbers with equal third digits remain sorted by their first two digits. Consequently, after the third pass, the numbers are sorted on the first three digits (the three lowest-order digits).

This process continues until the items have been sorted on all d digits. Remarkably, the items are sorted on the d-digit number at that point. Thus, only d passes are required to sort. [1]

The sorting method used in each pass can be any stable sort, but it should run in linear time to maintain efficiency.

Distributing sort works well on linked lists, and *counting sort* on arrays. Both of them are stable and work in linear time.

8.2 Distributing sort

Distributing sort is efficient on linked lists, and a version of radix sort can be built on it.

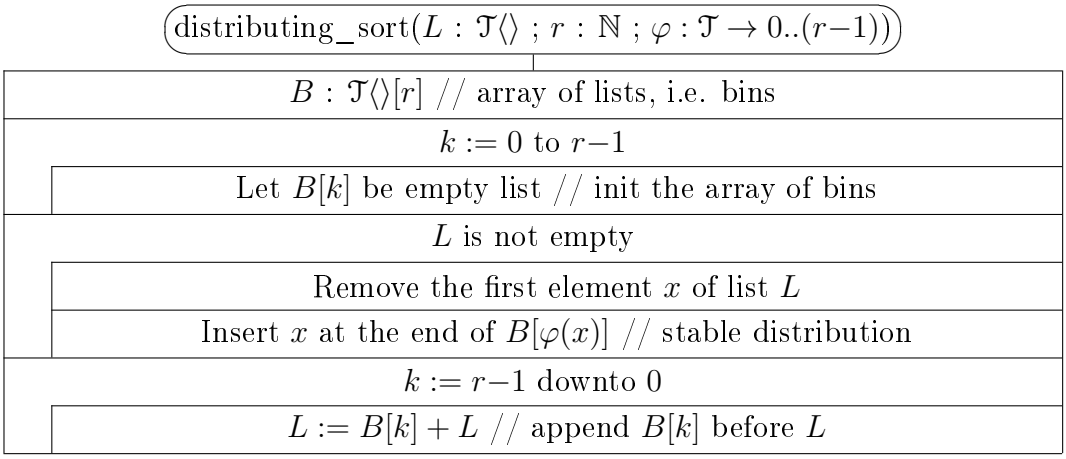
Remember that stable sorting algorithms maintain the relative order of records with equal keys (i.e. values).

Distributing sort is an ideal auxiliary method of radix sort because of its stability and linear time complexity.

Here comes a bit more general study than needed for radix sort. When distributing sort works for radix sort, its key function φ must select the appropriate digit.

The sorting problem: Given abstract list L of length n with element type $\mathcal{T}$, $r \in O(n)$ positive integer,
$\varphi : \mathcal{T} \to 0..(r-1)$ key selection function.

Let us sort list L with stable sort, with linear time complexity.



Efficiency of distributing sort: The first and the last loop iterates r times, and the middle loop n times. Provided that insertion and concatenation can be performed in $\Theta(1)$ time, the time complexity of distributing sort is consequently $\Theta(n + r) = \Theta(n)$ because of the natural condition $r \in O(n)$.

The size of array B determines its space complexity. Thus, it is $\Theta(r)$.

8.3 Radix sort on lists

The following example shows how the radix sort operates on an abstract list of seven 3-digit numbers with base (i.e. radix) 4. In each pass, we apply the distributing sort.

The input list (with a symbolic notation):
$L = \langle 103, 232, 111, 013, 211, 002, 012 \rangle$

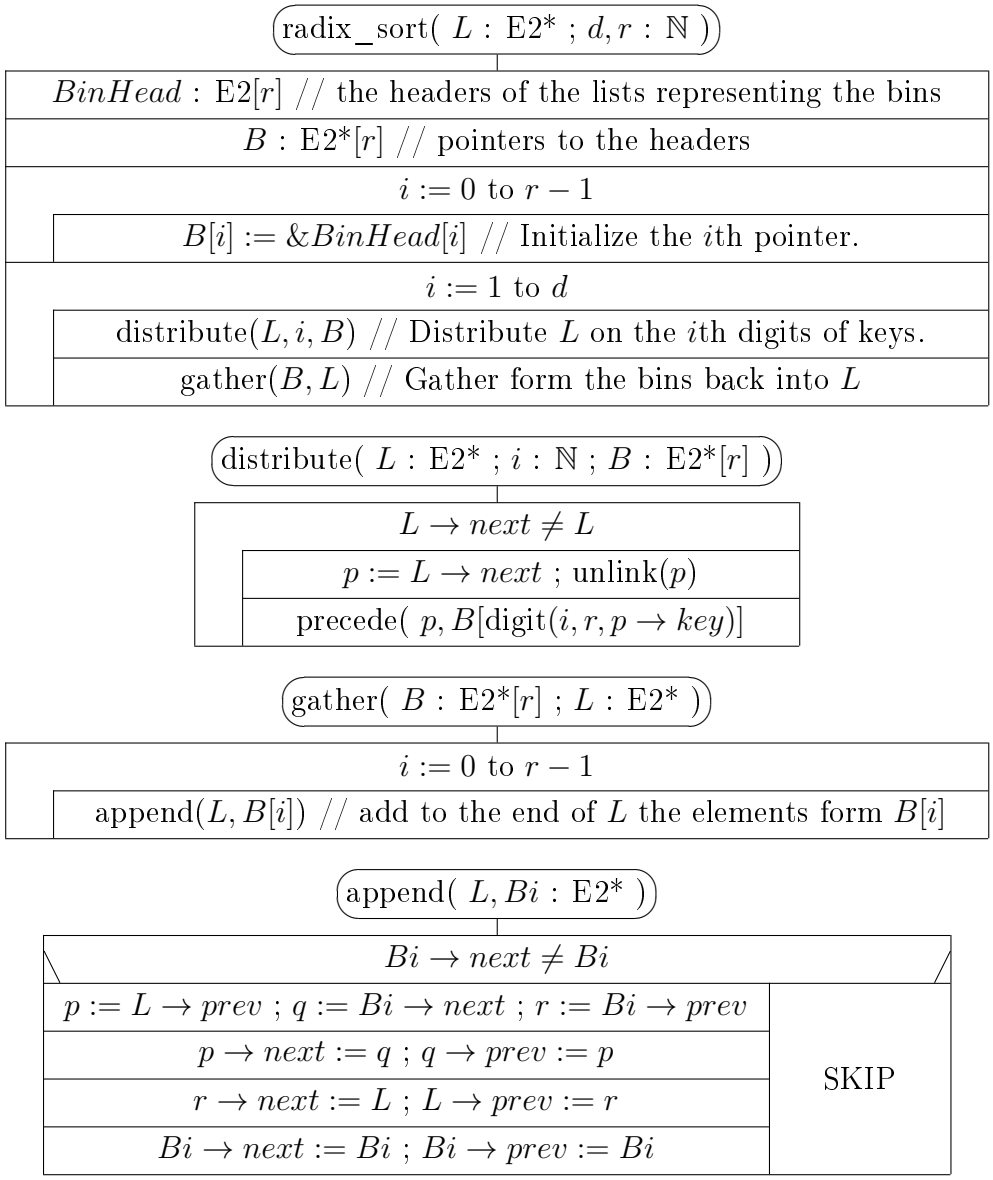
First pass (according to the rightmost digits of the numbers):
$B_0 = \langle\rangle$
$B_1 = \langle 111, 211\rangle$
$B_2 = \langle 232, 002, 012\rangle$
$B_3 = \langle 103, 013\rangle$
$L = \langle 111, 211, 232, 002, 012, 103, 013\rangle$

Second pass (according to the middle digits of the numbers):
$B_0 = \langle 002, 103\rangle$
$B_1 = \langle 111, 211, 012, 013\rangle$
$B_2 = \langle\rangle$
$B_3 = \langle 232\rangle$
$L = \langle 002, 103, 111, 211, 012, 013, 232\rangle$

Third pass (according to the leftmost digits of the numbers):
$B_0 = \langle 002, 012, 013\rangle$
$B_1 = \langle 103, 111\rangle$
$B_2 = \langle 211, 232\rangle$
$B_3 = \langle\rangle$
$L = \langle 002, 012, 013, 103, 111, 211, 232\rangle$

The digit sorts must be stable for the radix sort to work correctly. Distributing sort satisfies this requirement. If distributing sort runs in linear time ($\Theta(n)$ where n is the length of the input list), and the number of digits is constant d, radix sort also runs in linear time $\Theta(d * n) = \Theta(n)$.

Provided that we have to sort linked lists where the keys of the list elements are d-digit natural numbers with number base r, implementing the algorithm above is straightforward. For example, let us suppose that we have an L C2L with header, and function digit(i, r, x) can extract the ith digit of number x, where digit 1 is the lowest-order digit and digit d is the highest-order digit, in $\Theta(1)$ time.



Clearly, $T_{\text{append}} \in \Theta(1)$, so $T_{\text{gather}} \in \Theta(r)$ where $r = B.length$.
And $T_{\text{distribute}}(n) \in \Theta(n)$ where $n = |L|$.
Thus $T_{\text{radix\_sort}}(n, d, r) \in \Theta(r + d(n + r))$.
Consequently, if d is constant and $r \in O(n)$, then $T_{\text{radix\_sort}}(n) \in \Theta(n)$.

The space complexity of Radix sort is determined by the length of its temporal arrays, which is r. Thus, the space complexity of the Radix sort is $\Theta(r)$.

In a typical computer, a sequential random-access machine, we sometimes use Radix sort to sort records of information keyed by multiple fields. For

example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year. [1]

8.4 Counting sort

While the previous version of radix sort is efficient on linked lists, the counting sort can be applied efficiently to arrays, and another version can be built on it.

Remember that stable sorting algorithms maintain the relative order of records with equal keys (i.e. values).

The counting sort is stable and an ideal auxiliary method of radix sort because of its linear time complexity.

Here comes a bit more general study than needed for radix sort. When counting sort is used for radix sort, its key function φ must select the appropriate digit.

The sorting problem: Given array A:$\mathcal{T}[n]$, $r \in O(n)$ positive integer, $\varphi : \mathcal{T} \to 0..(r-1)$ key selection function.

Let us sort array A with stable sort and linear time complexity so that the result is produced in array B.

counting_sort($A, B : \mathcal{T}[n]$; $r : \mathbb{N}$; $\varphi : \mathcal{T} \to 0..(r-1)$)

$C : \mathbb{N}[r]$ // counter array
$k := 0$ to $r-1$
$\quad C[k] := 0$ // init the counter array
$i := 0$ to $n-1$
$\quad C[\varphi(A[i])]{+}{+}$ // count the items with the given key
$k := 1$ to $r-1$
$\quad C[k] \mathrel{+}= C[k-1]$ // $C[k]$:= the number of items with key $\leq k$
$i := n-1$ downto 0
$\quad k := \varphi(A[i])$ // k := the key of $A[i]$
$\quad C[k]--$ // The next one with key k must be put before $A[i]$ where
$\quad B[C[k]] := A[i]$ //Let $A[i]$ be the last of the {unprocessed items with key k}

The first loop of the procedure above assigns zero to each element of the counting array C.

The second loop counts the number of occurrences of key k in $C[k]$ for each possible key k.

The third loop sums the number of keys $\leq k$, considering each possible key k.

The number of keys ≤ 0 is the same as the number of keys $= 0$, so the value of $C[0]$ is unchanged in the third loop. Considering greater keys, we have that the number of keys $\leq k$ equals the number of keys $= k$ + the number of keys $\leq k-1$. Thus, the new value of $C[k]$ can be counted by adding the new value of $C[k-1]$ to the old value $C[k]$.

The fourth loop goes on the input array in the reverse direction. We put the elements of input array A into output array B: Considering any key k in the input array, first, we process its last occurrence. The element containing it is put into the last place reserved for keys $= k$, i.e. into $B[C[k]-1]$: First, we decrease $C[k]$ by one and put this element into $B[C[k]]$. According to this reverse direction, the next occurrence of key k will be the immediate predecessor of the actual item, etc. Thus, the elements with the same key remain in their original order, and we receive a stable sort.

The time complexity is $\Theta(n+r)$. Provided that $r \in O(n)$, $\Theta(n+r) = \Theta(n)$, and so $T(n) \in \Theta(n)$.

Regarding the space complexity, besides the input array, we have output array B of n items and counter array C of r items. As a result, $S(n) \in \Theta(n+r) = \Theta(n)$ since $r \in O(n)$.

Illustration of counting sort: We suppose that we have to sort numbers of two digits with number base four according to their right-side digit, i.e. function φ selects the rightmost digit.

The input:

	0	1	2	3	4	5
$A:$	02	32	30	13	10	12

The changes of the counter array C [the first column reflects the first loop initializing counter array C to zero, the next six columns reflect the second loop counting the items with key k, for each possible key; the column labeled by $\sum$ reflects the third loop which sums up the number of items with keys $\leq k$; and the last six columns reflect the fourth loop placing each item of the input array into its place in the output array]:

	C	02	32	30	13	10	12	$\sum$	12	10	13	30	32	02
0	0			1		2		2		1		0		
1	0							2						
2	0	1	2				3	5	4				3	2
3	0				1			6			5			

The output:

	0	1	2	3	4	5
$B:$	30	10	02	32	12	13

Now, we suppose that the result of the previous counting sort is to be sorted according to the left-side digits of the numbers (of number base 4), i.e. function φ selects the leftmost digits of the numbers.

The input:

	0	1	2	3	4	5
$B:$	30	10	02	32	12	13

The changes of counter array C:$\mathbb{N}[4]$:

	C	30	10	02	32	12	13	$\sum$	13	12	32	02	10	30
0	0			1				1				0		
1	0		1			2	3	4	3	2			1	
2	0							4						
3	0	1			2			6			5			4

The output:

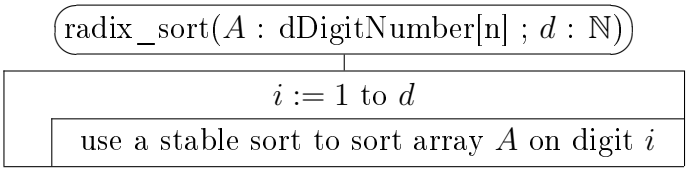
	0	1	2	3	4	5
$A:$	02	10	12	13	30	32

The first counting sort ordered the input according to the right-side digits of the numbers. The second counting sort ordered the result of the first sort according to the left-side digits of the numbers using a stable sort. Thus, in the final result, the numbers with the same left-side digits remained in order according to their right-side digits. Consequently, the numbers are sorted according to both digits in the final result.

Therefore, the two counting sorts illustrated above form a radix sort's first and second passes. And our numbers have just two digits now, so we have performed a complete radix sort in this example.

8.5 Radix-Sort on arrays ([1] 8.3)

The keys of array A are d-digit natural numbers with number base r. The rightmost digit is the least significant (digit 1), and the leftmost digit is the most significant (digit d).



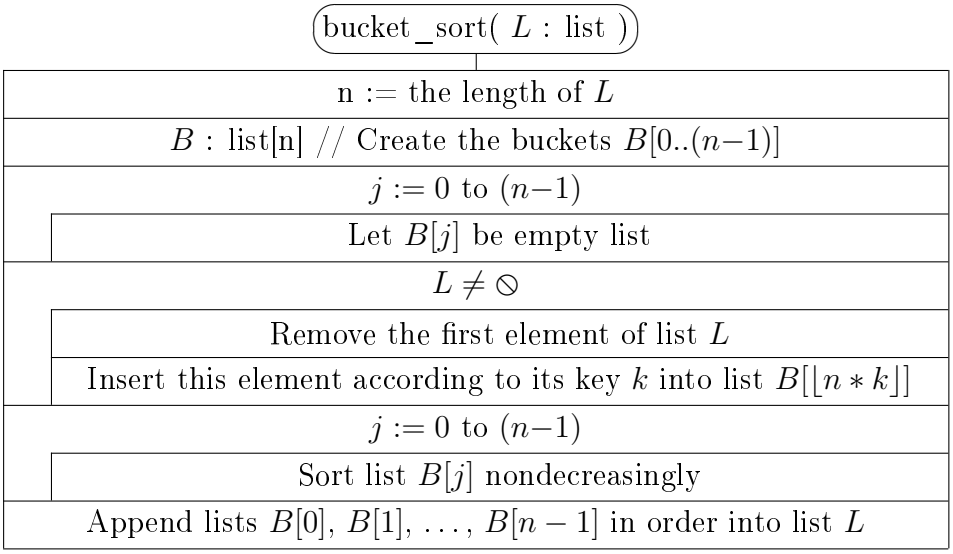
Provided that the stable sort is counting sort, the time complexity of Radix sort is $\Theta(d(n+r))$. If d is a constant and $r \in O(n)$, $\Theta(d(n+r)) = \Theta(n)$, i.e. $T(n) \in \Theta(n)$.

The space complexity of the Counting sort determines that of the Radix sort. Thus $S(n) \in \Theta(n+r) = \Theta(n)$ since $r \in O(n)$.

8.6 Bucket sort

We suppose the items to be sorted are elements of the real interval $[0; 1)$.

This algorithm is efficient if the input keys are equally distributed on $[0; 1)$. We sort the buckets with one of the known sorting methods, like insertion or merge sort.



Clearly $mT(n) \in \Theta(n)$. If the keys of the input are equally distributed on $[0; 1)$, $AT(n) \in \Theta(n)$. $MT(n)$ depends on the sorting method we use when sorting lists $B[j]$ nondecreasingly. For example, using Insertion sort $MT(n) \in \Theta(n^2)$; using Merge sort $MT(n) \in \Theta(n \log n)$.

Considering the space complexity of Bucket sort, we have a temporal array B of n elements. We do not need other data structures if our lists are linked. Provided that we use Merge sort as the subroutine of Bucket sort, the depth of recursion is $O(\log n)$ in any case. And Insertion sort sorts in

place. Consequently, $mS(n), MS(n) \in \Theta(n)$, if we sort a linked list, and we use Insertion Sort or Merge sort as the subroutine of Bucket sort.

9 Hash Tables

In everyday programming practice, we often need so-called dictionaries. These are collections of records with unique keys. Their operations are (1) inserting a new record into the dictionary, (2) searching for a record identified by a key, and (3) removing a record identified by a key (or localised by a previous search).

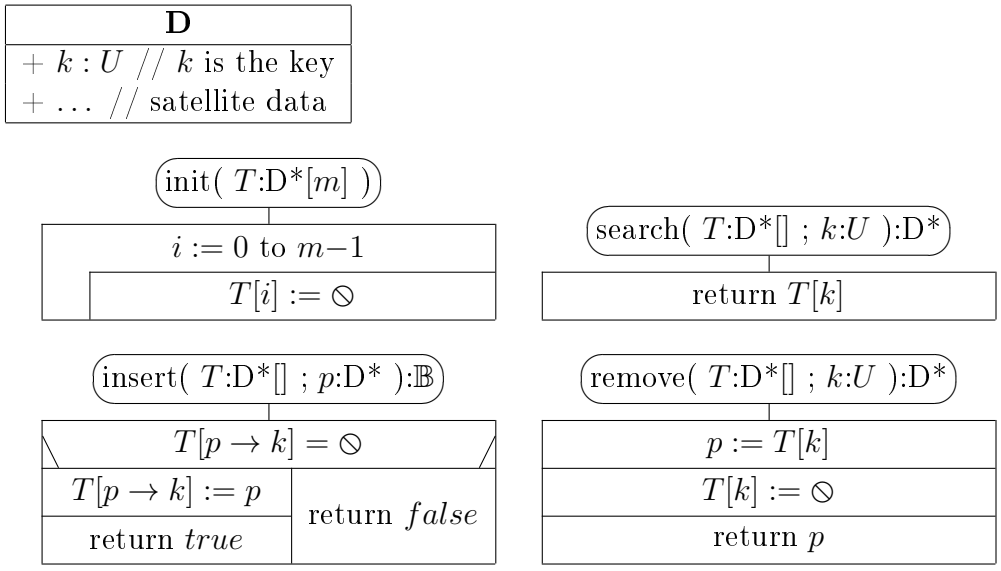
Besides AVL trees, B+ trees (see them in the next semester) and other kinds of balanced search tree dictionaries are often represented and implemented with hash tables, provided that we would like to optimise the average running time of the operations. Using hash tables, the average running time of the operations above is the ideal $\Theta(1)$, while the maximum of it is $\Theta(n)$. (With balanced search trees, each key-based operation's worst case and average case performance are $\Theta(\log n)$.)

Notations:
m: the size of the hash table
$T[0..(m-1)]$: the hash table
$T[0], T[1], \ldots, T[m-1]$: the slots of the hash table
$\oslash$: empty slot in the hash table (when we use direct-address tables or key collisions are resolved by chaining)
E: the key of empty slots in case of open addressing
D: the key of deleted slots in case of open addressing
n: the number of records stored in the hash table
$\alpha = n/m$: *load factor*
U : the universe of keys; $k, k', k_i \in U$
$h : U \to 0..(m-1)$: hash function

We suppose the hash table does not contain two or more records with the same key and that $h(k)$ can be calculated in $\Theta(1)$ time.

9.1 Direct-address tables

In the case of direct address tables, we do not have hash functions. We suppose that $U = 0..(m-1)$ where $m \geq n$, but m is not too big compared to n.
T : $D*[m]$ is the direct-address table. Its slots are pointers referring to records of type D. Each record has a key $k : U$ and contains satellite data. The direct-address table is initialised with $\oslash$ pointers.



Clearly $T_{\text{init}}(m) \in \Theta(m)$. And for the other three operations, we have $T \in \Theta(1)$.

9.2 Hash tables

Hash function: Provided that $|U| >> n$, direct address tables cannot be applied, or at least applying them is wasting space. Thus, we use a hash function $h : U \to 0..(m-1)$ where typically $|U| >> m$ (the size of the universe U of keys is much greater than the size m of the hash table). The record with key k is to be stored in slot $T[h(k)]$ of hash table $T[0..(m-1)]$.

Remember that the hash table should not contain two or more records with the same key and that $h(k)$ must be calculated in $\Theta(1)$ time.

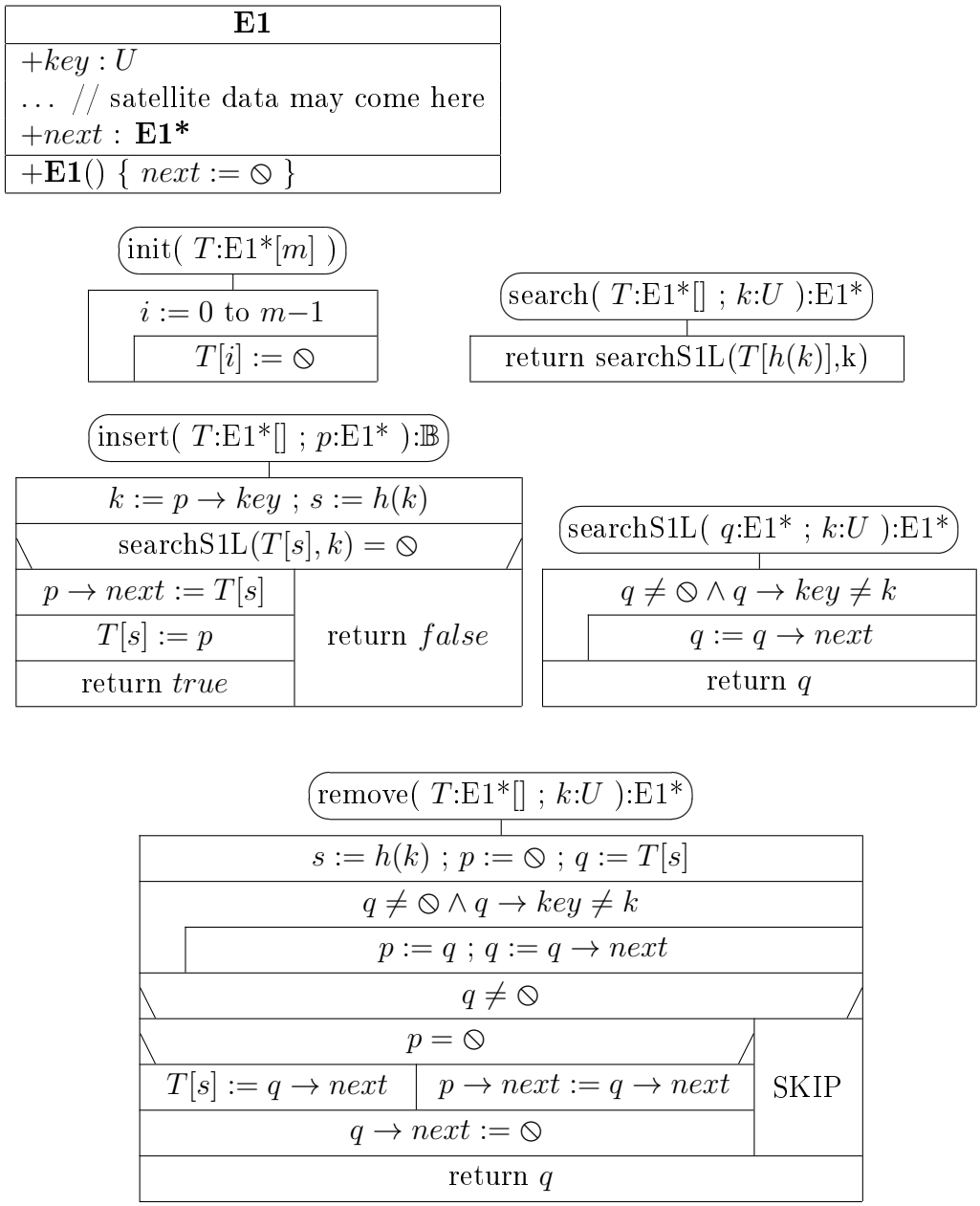
Function $h : U \to 0..(m-1)$ is *simple uniform hashing*, if it distributes the keys evenly into the slots, i.e. any given key is equally likely to hash into any of the m slots, independently of where other items have hashed to. Simple uniform hashing is a general requirement for hash functions.

Collision of keys: Provided that $h(k_1) = h(k_2)$ for keys $k_1 \neq k_2$ we speak of key collision. Usually $|U| >> m$, key collisions probably happen, and we have to handle this situation.

For example, suppose the keys are integer numbers and $h(k) = k \bmod m$. Then exactly those keys are hashed to slot s for which $s = k \bmod m$.

9.3 Collision resolution by chaining

We suppose that the slots of the hash table identify simple linked lists (S1L) that is T: E1*[m] where the elements of the lists contain the regular fields *key* and *next*, plus usually additional fields (satellite data). Provided the hash function maps two or more keys to the same slot, the corresponding records are stored in the list identified by this slot.



Clearly $T_{\text{init}}(m) \in \Theta(m)$. For the other three operations $mT \in \Theta(1)$, $MT(n) \in \Theta(n)$, $AT(n,m) \in \Theta(1+\frac{n}{m})$.

$AT(n,m) \in \Theta(1+\frac{n}{m})$ is satisfied, if function $h : U \to 0..(m-1)$ is *simple uniform hashing*, because the average length of the lists of the slots is equal to $\frac{n}{m} = \alpha$.

Usually $\frac{n}{m} \in O(1)$ is required. In this case, $AT(n,m) \in \Theta(1)$ is also satisfied for insertion, search, and removal.

9.4 Good hash functions

Division method: Provided that the keys are integer numbers,

$$h(k) = k \bmod m$$

is often a good choice because it can be calculated simply and efficiently. And if m is a prime number not too close to a power of two, it usually distributes the keys evenly among the slots, i.e. on the integer interval $0..(m-1)$.

For example, if we want to resolve key collision by chaining, and we would like to store approximately 2000 records with maximum load factor $\alpha \approx 3$, then $m = 701$ is a good choice: 701 is a prime number which is close to 2000/3, and it is far enough from the neighbouring powers of two, i.e. from 512, and 1024.

Keys in interval $[0\,;1)$: Provided that the keys are evenly distributed on $[0\,;1)$, function

$$h(k) = \lfloor k * m \rfloor$$

is also simple uniform hashing.

Multiplication method: Provided that the keys are real numbers, and $A \in (0;1)$ is a constant,

$$h(k) = \lfloor \{k * A\} * m \rfloor$$

is a hash function. ($\{x\}$ is the fraction part of x.) It does not distribute the keys equally well with all the possible values of A.

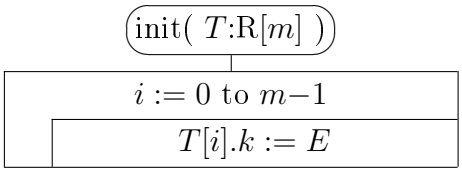
Knuth proposes $A = \frac{\sqrt{5}-1}{2} \approx 0.618$ because it is likely to work reasonably well. Compared to the division method, it has the advantage that the value of m is not critical.

Each method above supposes that the keys are numbers. If the keys are strings, the characters can be considered digits of unsigned integers with the appropriate number base. Thus, the strings can be interpreted as big natural numbers.

9.5 Open addressing

The hash table is $T : R[m]$. The records of type R are directly in the slots. Each record has a key field $k : U \cup \{E, D\}$ where $E \neq D$; $E, D \notin U$ are global constants in order to indicate empty (E) and deleted (D) slots.

R
+ $k : U \cup \{E, D\}$ // k is a key or it is Empty or Deleted
+ ... // satellite data



Notations for open addressing:
$h : U \times 0..(m-1) \to 0..(m-1)$: probing function
$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$: potential probing sequence

The hash table does not contain double keys.

The empty and deleted slots together are *free* slots. The other slots are *occupied*.) Instead of a single hash function, we have m hash functions now:

$$h(\cdot, i) : U \to 0..(m-1) \qquad (i \in 0..(m-1))$$

We try these in this order in open addressing, one after the other if needed.

9.5.1 Open addressing: insertion and search, without deletion

In many applications of dictionaries (and hash tables), we do not need deletion. Insertion and search are sufficient. In this case, insertion is more straightforward.

Suppose we want to insert record r with key k into the hash table. First, we probe slot $h(k,0)$. If it is occupied and its key is not k, we try $h(k,1)$, and so on, throughout $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ until
- we find an empty slot, or
- we find an occupied slot with key k, or
- all the slots of the *potential probing sequence* have been considered but found neither empty slot nor occupied slot with key k.

+ If we find an empty slot, we put r into it. Otherwise, insertion fails.

$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ is called *potential probing sequence* because during insertion, or search (or deletion) only a prefix of it is generated. This prefix is called *actual probing sequence*.

The potential probing sequence must be a permutation of $\langle 0, 1, \ldots, (m-1)\rangle$, which means that it covers the whole hash table, i.e. it does not refer twice to the same slot.

The length of the actual probing sequence of insertion/search/deletion is $i \iff$ this operation stops at probe $h(k, i-1)$.

When we search for the record with key k, again we follow the potential probing sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$.
- We stop successfully when we find an occupied slot with key k.
- The search fails if we find an empty slot or use up the potential probing sequence unsuccessfully.

In the ideal case, we have *uniform hashing:* the potential probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, (m-1)\rangle$.

Provided that
- the hash table does not contain deleted slots,
- its load factor $\alpha = n/m$ satisfies $0 < \alpha < 1$, and
- we have uniform hashing,

+ The expected length of an unsuccessful search / successful insertion is, at most

$$\frac{1}{1-\alpha}$$

+ and the expected length of a successful search / unsuccessful insertion is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

For example, the first result above implies that if the hash table is half full, the expected number of probes in an unsuccessful search (or in a successful insertion) is less than 2. If the hash table is 90% full, the expected number of probes is less than 10.

Similarly, the second result above implies that if the hash table is half full, the expected number of probes in a successful search (or unsuccessful insertion) is less than 1.387. If the hash table is 90% full, the expected number of probes is less than 2.559. [1].

9.5.2 Open addressing: insertion, search, and deletion

A successful deletion consists of a successful search for the slot $T[s]$ containing a given key + the assignment $T[s].k := D$ (let the slot be deleted). $T[s].k := E$ (let the slot be empty) is incorrect. For example, let us suppose that we inserted record r with key k into the hash table, but it could not be put into $T[h(k,0)]$ because of key collision, and we put it into $T[h(k,1)]$. And then, we delete the record at $T[h(k,0)]$. If this deletion performed $T[h(k,0)].k := E$, a subsequent search for key k would stop at the empty slot $T[h(k,0)]$, and it would not find record r with key k in $T[h(k,1)]$. Instead, deletion performs $T[h(k,0)].k := D$. Then the subsequent search for key k does not stop at slot $T[h(k,0)]$ (because neither it is empty nor it contains key k), and it finds record r with key k in $T[h(k,1)]$. (The search and deletion procedures are not changed despite the presence of deleted slots.)

Thus, during a search, we go through deleted slots, and we stop when
- we find the slot with the key we search for (successful search), or
- we find an empty slot or use up the initial probe sequence of the given key (unsuccessful search).

Insertion becomes more complex because of the presence of deleted slots. During the insertion of record r with key k, we perform a full search for k and remember the first deleted slot found during the search.
- If the search is successful, the insertion fails (because we do not allow duplicated keys).
- If the search is unsuccessful, but some deleted slot is remembered, we put r into it.
- If the search is unsuccessful, no deleted slot is remembered, but the search stops at an empty slot, then we put r into it.
- If the search is unsuccessful, and neither a deleted nor empty slot is found, the insertion fails because the hash table is full.

If we use a hash table for a long time, there may be many deleted slots and no empty slots, although the table is far from full. This means the unsuccessful searches will check all the slots, and the other operations will slow down, too. So we have to eliminate the deleted slots, for example, by rebuilding the whole table.

9.5.3 Linear probing

In this subsection, and the next two, we consider three strategies to generate actual probing sequences, that is, the adequate prefix of

$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$. In each case we have a primary hash function $h_1 : U \to 0..(m-1)$ where $h(k,0) = h_1(k)$. If needed, starting from this slot, we go step by step through the hash table slots, according to a well-defined rule, until we find the appropriate slot or find the actual operation impossible. h_1 must be a simple uniform hash function.

The most straightforward strategy is linear probing:

$$h(k,i) = (h_1(k) + i) \bmod m \qquad (i \in 0..(m-1))$$

It is easy to implement linear probing, but we have only m different probing sequences instead of the $m!$ probing sequences needed for uniform hashing: Given two keys, k_1 and k_2; if $h(k_1,0) = h(k_2,0)$ then their whole probing sequences are the same. In addition, different probing sequences tend to be linked into continuous, long runs of occupied slots, increasing the expected time of searching. This problem is called *primary clustering*. The longer such a cluster is, the more probable it becomes even longer after the subsequent insertion. For example, let us have two free slots with i occupied between them. Then, the probability that the subsequent insertion will increase its length is at least $(i+2)/m$. And it may even be linked with another cluster. Linear probing may be selected only if the probability of key collision is extremely low.

9.5.4 Quadratic probing*

$$h(k,i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m \qquad (i \in 0..m-1)$$

where $h_1 : U \to 0..(m-1)$ is the primary hash function; $c_1, c_2 \in \mathbb{R}; c_2 \neq 0$. The different probing sequences are not linked together. Still, we have only m different probing sequences instead of the $m!$ probing sequences needed for uniform hashing: Given two keys, k_1 and k_2; if $h(k_1,0) = h(k_2,0)$ then their whole probing sequences are the same. This problem is called *secondary clustering*.

Choosing the constants of quadratic probing: In this case, the potential probing sequence, i.e. $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$, may have equal members, which implies that it does not cover the hash table. Therefore, we must be careful about selecting the constants of quadratic probing.

For example, if size m of the hash table is a power of 2, then $c_1 = c_2 = 1/2$ is appropriate. In this case

$$h(k,i) = \left(h_1(k) + \frac{i+i^2}{2}\right) \bmod m \qquad (i \in 0..m-1)$$

Thus

$$(h(k,i+1)-h(k,i)) \bmod m = \left(\frac{(i+1)+(i+1)^2}{2} - \frac{i+i^2}{2}\right) \bmod m =$$

$$(i+1) \bmod m$$

So it is easy to compute the slots of the probing sequences recursively:

$$h(k,i+1) = (h(k,i)+i+1) \bmod m$$

Exercise 9.1 *Write the structure diagrams of the operations of hash tables with quadratic probing ($c_1 = c_2 = 1/2$) applying the previous recursive formula.*

9.5.5 Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m \qquad (i \in 0..(m-1))$$

where $h_1 : U \to 0..(m-1)$ and $h_2 : U \to 1..(m-1)$ are hash functions. The probing sequence covers the hash table $\iff$ $h_2(k)$ and m are relative primes. It is satisfied, for example, if $m > 1$ is a power of 2 and $h_2(k)$ is an odd number for each $k \in U$, or if m is a prime number. For example, if m is a prime number (which should not be close to powers of 2) and m' is a bit smaller (let $m' = m-1$ or $m' = m-2$), then the following (h_1, h_2) is an eligible choice.

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m')$$

In the case of double hashing for each different pair of $(h_1(k), h_2(k))$, there is a different probing sequence, and so we have $\Theta(m^2)$ different probing sequences.

Although the number of the probing sequences of double hashing is far from the ideal number $m!$ of probing sequences, its performance appears to be very close to that of the perfect scheme of uniform hashing.

Illustration of the operations of double hashing: Because $h(k,i) = (h_1(k) + ih_2(k)) \bmod m \quad (i \in 0..(m-1))$, therefore $h(k,0) = h_1(k)$ and $h(k,i+1) = (h(k,i)+d) \bmod m$ where $d = h_2(k)$. After calculating the place of the first probing $(h_1(k))$, we always make a step of distance d cyclically around the table.

Example: $m = 11$ $h_1(k) = k \bmod 11$ $h_2(k) = 1 + (k \bmod 10)$.

In the following table's "operations" (op) column, *ins*=insert, *src*=search, *ddel*=delete. Next, the *key* of the operation comes (being neither E nor D).

In this table, we do not handle satellite data. We show just the keys. In the next column of the table, there is $h_2(key)$, but only if needed. Next, we find the actual probing sequence. Insertion remembers the first deleted slot of the probing sequence, if any. In such cases, we underlined the index of this slot. In column "s", we have a "+" sign for a successful and an "−" sign for an unsuccessful operation. In the table, we do not handle satellite data; we process only the keys. (See the details in section 9.5.2.)

In the last 11 columns of the table, we represent the actual state of the hash table. The cells representing empty slots are left empty. We wrote the reserving key into each occupied slot cell, while the deleted slot cells contain the letter D.

op	key	h_2	probes	s	0	1	2	3	4	5	6	7	8	9	10
init				+											
ins	32		10	+											32
ins	40		7	+								40			32
ins	37		4	+					37			40			32
ins	15	6	4; 10; 5	+					37	15		40			32
ins	70	1	4; 5; 6	+					37	15	70	40			32
src	15	6	4; 10; 5	+					37	15	70	40			32
src	104	5	5; 10; 4; 9	−					37	15	70	40			32
del	15	6	4; 10; 5	+					37	D	70	40			32
src	70	1	4; 5; 6	+					37	D	70	40			32
ins	70	1	4; $\underline{\mathbf{5}}$; 6	−					37	D	70	40			32
del	37		4	+					D	D	70	40			32
ins	104	5	$\underline{\mathbf{5}}$; 10; 4; 9	+					D	104	70	40			32
src	15	6	4; 10; 5; 0	−					D	104	70	40			32

Exercise 9.2 (Programming of double hashing) *Write the structure diagrams of insertion, search, and deletion where x is the record to be inserted, k is the key we search for, and the key of the record we want to delete.*

The hash table is $T[0..(m-1)]$.

In a search, we try to find the record identified by the given key. After a successful search, we return the position (index of the slot) of the appropriate record. After an unsuccessful search, we return the number -1.

After a successful insertion, we return the position of the insertion. At an unsuccessful insertion, there are two cases. If there is no free place in the hash table, we return "$-(m+1)$". If the key of the record to be inserted is found at slot j, we return "$-(j+1)$".

In a deletion, we try to find and delete the record identified by the given key. After successful deletion, we return the appropriate record's position

(slot index). After an unsuccessful deletion, we return "−1".

Solution: