Chapter 1

Introduction

1.1 Notion of an Algorithm

The notion of an algorithm is a fundamental one in Computer Science and so we begin with an intuitive explanation of this idea. Among other things, a well-specified computational problem specifies what the output should be for a given valid input. An **algorithm** is a mechanism that takes a valid input and produces an output that satisfies the above relationship.

Computational problems often have associated with them a search space in which the solution lies. In this case, we can view an algorithm as a mechanism for exploring this search space to find a solution.

For a given problem, our goal is to come up with an algorithm that explores the associated search space or achieves the specified input/output relationship as efficiently as possible.

Consider the following computational problem.

> Given two positive integers m and n, their greatest common divisor (gcd, for short) is the largest positive integer that divides both. Find a method to determine this gcd efficiently.

Viewed as a search problem, the search space consists of all the numbers from 1 to the smaller of m and n, say n. One (naive) way of determining the solution is to go through this search space exhaustively, dividing m and n with the numbers starting from n down to 1 (why ?), stopping with the first number that divides them both.

A cleverer way of examining the search space, due to Euclid, is given in the scheme **Algorithm 1** that we subsequently refer to as the GCD algorithm.

Note that the algorithm described does not require m to be larger than n. If $m < n$, then after the first division step, the new values of m and n satisfy the relation $m > n$. To understand how Euclid's algorithm explores the search space, let us try to answer the following question:

> How many times does Step 1 execute for a given pair of inputs m and n ?

For this we first need to prove the little theorem below.

Theorem 1.1. If $m \geq n$, then the remainder in the division of m by n, namely, m mod $n < m/2$.

Algorithm 1 GreatestCommonDivisor

Input: Positive integers m and n
Output: Greatest common divisor of m and n

1: $r \leftarrow m \bmod n$ ▷ Compute the remainder r
2: **while** $r \neq 0$ **do**
3: $m \leftarrow n$ ▷ reset m amd n
4: $n \leftarrow r$
5: $r \leftarrow m \bmod n$ ▷ Recompute the remainder r
6: **end while**
7: **return** n

Proof. If $n \leq m/2$ then the claim follows from the fact that the remainder $r = m \bmod n < n$ and hence less than $m/2$. If $n > m/2$ then the remainder $r = m \bmod n = m - n$ (Why ?). Since we can rewrite $n > m/2$ as $n > m - m/2$, we have $m - n < m/2$ and hence $r < m/2$ in this case also. □

Note 1.1. In the statement of the theorem above, $m/2$ is not necessarily integral

Note 1.2. The correctness of the above claim can be immediately seen from Fig. 1.1.

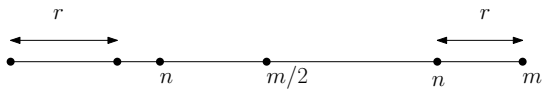


Figure 1.1: A geometric proof

Let r_0, r_1, r_2, ..., r_k be the sequence of remainders produced by the GCD algorithm above. We know that $r_0 > r_1 > r_2 > ... > r_{k-1} > r_k = 0$, that is, the sequence of remainders strictly decreases to 0. Using the little theorem above, we can discern some pattern in the way this happens.
Indeed we have:

$r_0 = m \bmod n < m/2$
$r_1 = n \bmod r_0 < n/2$
$r_2 = r_0 \bmod r_1 < r_0/2$
$r_3 = r_1 \bmod r_2 < r_1/2 < n/4$
$r_4 = r_2 \bmod r_3 < r_2/2$
$r_5 = r_3 \bmod r_4 < r_3/2 < n/8$
.
.
.

and so on.

Thus after two divisions the remainder becomes less than $n/2$, after four divisions the remainder becomes less than $n/4$, and so on. In general, after $2k$ divisions, $k >= 1$, the remainder becomes less than $n/2^k$. Let us assume that $r_0 > 0$ so that there are at least two divisions. With this assumption, we also exclude the trivial case where m is a multiple of n.

Claim 1.1. The remainder r becomes 0 after at most $2\lceil \log_2 n \rceil$ divisions.

Proof: This is not difficult to see if we rewrite $n/2^k$ as $2^{\log_2 n}/2^k$ and note that since $\log_2 n \leq \lceil \log_2 n \rceil$, we have $2^{\log_2 n}/2^k \leq 1$ for $k = \lceil \log_2 n \rceil$. This forces the remainder, after $2\lceil \log_2 n \rceil$ divisions, to lie in the interval $[0, 1)$. Since the remainder is always a non-negative integer, the only way it can belong

to this interval is to be 0. Hence the claim. ■

Note that the above bound covers all inputs for which $m > n > 1$.

Problem 1.1. Show that at most $2\lceil \log_2 n \rceil + 1$ divisions are made by the GCD algorithm in inputs for which $m > n = 1$.

Problem 1.2. Show that at most $2(\lceil \log_2 m \rceil + 1)$ divisions are made by the GCD algorithm in inputs for which $1 \leq m < n$.

Letting m = 1989 and n = 1590, the remainder sequence is:

399, 393, 6, 3, 0

A close look at the above remainder sequence shows that while the remainder 393 is close to the remainder 399, the next remainder 6 is less than half of 399.

Summary: When $m > n$, the GCD algorithm above looks at most $2\lceil \log_2 n \rceil + 1$ of the numbers from 1 to n instead of all the numbers from 1 to n to determine the greatest common divisor.

Thus we have completely answered the question as to how many times Step 1 is executed for a given pair of inputs m and n.

The GCD algorithm also serves to highlight several other issues germane to algorithm design. The first and foremost among these is the issue of proving that a proposed algorithm is **correct**. This means showing that the required input/output relationship holds for all valid input data. It is analogous to proving the correctness of a theorem in Mathematics. To drive home the issues involved, we prove the correctness of the above algorithm. We begin with the claim that:

Claim 1.2. gcd(m, n) = gcd(n, r)

Proof. What the claim says is that the gcd of the new pair that we generate after a division that generates a non-zero remainder r is equal to the gcd of the pair before this divison. Or, in other words, the gcd remains invariant over the division loop. It is easy to see why this claim is correct. We can write the process of dividing m by n in the following equational form:

$$m \quad = \quad q * n + r, \tag{1.1}$$

where $0 \leq r < n$.

Equation 1.1 implies that a common divisor of m and n is also a common divisor of n and r and vice versa. Hence the claim of the theorem.

The rest of the correctness proof is completed by an induction over the number of divisions performed. Indeed let (m_k, n_k) be the pair generated after the kth division step, where $(m_0, n_0) = (m, n)$.

The claim is that gcd(m, n) = gcd(m_k, n_k) after $k \geq 0$ divisions. Clearly, this is true for $k = 0$. Assume it to be true for some $k \geq 0$. Now, by the claim above, gcd(m_k, n_k) = gcd(m_{k+1}, n_{k+1}) after the $(k + 1)$th division; hence after $k + 1$ divisions, gcd(m, n) = gcd(m_{k+1}, n_{k+1}). This proves the claim for all $k \geq 0$. ■

A second important issue that an algorithm designer is confronted with is to show that an algorithm **terminates** after a finite number of steps for all valid inputs. Can we show that the **GCD** algorithm

terminates ? For this we must show that r goes to 0 in a finite number of steps. This is the case since the sequence of remainders strictly decreases, and being non-negative, must therefore become 0 in at most $2\lceil \log_2 n \rceil + 1$ steps, assuming $n \leq m$.

A third important issue that concerns us is the **quantification** of the performance of an algorithm. Any algorithm uses both time and space when implemented to run on some machine. These parameters, commonly used to quantify an algorithm, are called its space and time complexities. We will define these terms and the relevant issues formally in a subsequent chapter.

With reference to the GCD algorithm above, we note that its running time depends on the number of times Step 1 executes. This, as we showed above, is at most $2\lceil \log n \rceil + 1$, where n is the smaller of the inputs m and n.

Problem 1.3. The version of Euclid's algorithm described above exploits the fact that $gcd(m, n) = gcd(n, m \bmod n)$. Let $m > n$. Rewrite the algorithm using the observation that $gcd(m, n) = gd(n, m - n)$. Thus subtraction instead of division is to be used as the basic arithmetic operation to converge to the gcd of the input.

Problem 1.4. Design a recursive algorithm to compute the gcd of two positive integers.

Problem 1.5. How would you use the GCD algorithm described above to compute the lcm(*l*east *c*ommon *m*ultiple) of two positive integers m and n ?

Problem 1.6. Given positive integers m and n, there exists integers u and v such that $um + vn$ = gcd(m, n). How would you extend the GCD algorithm described above to determine u and v ?

Hint: Maintain a pair of values u and v such that the current remainder is $u * m + v * n$.

Problem 1.7. Assume that $m > n$. Design an alternate GCD algorithm based on the following observations. The right-hand side of each observation tells you how you can modify the inputs m and n, while keeping the gcd invariant (that is, the same as the gcd of m and n). You must specifically mention how your algorithm terminates.
1. gcd(m, n) = gcd$(m/2, n)$ if m is even
2. gcd(m, n) = gcd$(m, n/2)$ if n is even
3. gcd(m, n) = 2*gcd$(m/2, n/2)$ if m and n are both even
4. gcd(m, n) = gcd$((m + n)/2, (m - n)/2)$ if m and n are both odd and $m > n$
5. gcd(m, n) = gcd$((m + n)/2, (n - m)/2)$ if m and n are both odd and $n > m$

1.2 Data Structures

The subject of data structures is about different ways of organizing data. Why would we want to do this ? We want to do this because the efficiency of an algorithm often depends on how the data is organized. That's also why we study algorithms and data structures together. There is an organic connection between the two topics. The following example highlights this.

The median (or middle) M of a list of n numbers is a number (of the list) such that at least $\lfloor n/2 \rfloor$ numbers of the list are $\leq M$ and at least $\lfloor n/2 \rfloor$ numbers are $\geq M$. Consider a very simple problem.

```
Given a sorted list of n numbers, find the median element
```

The question is: how should we store the list? If we store it as an array, we can find the median element in constant time (means an amount of time that does not depend on the size of the list), whereas if we keep it as a linked list, we have to traverse half the list to find the median element,

assuming we know a priori the number of elements in the list.

Though simple, this problem illustrates very clearly that the way we organize our data depends very crucially on what we want to do with it.

We shall see other illustrations of this during the course of these lectures.

1.3 Chapter Notes

It is worth reading Chapter 0 from Edsger W. Dijkstra's classic work, "A discipline of programming". Using the same GCD algorithm we discussed in this chapter, he eloquently argues how executional abstraction is fundamental to the notion of an algorithm.

Chapter 2

Algorithm Analysis

2.1 The big-Oh, big-Omega and Theta notations

Let $f(n)$ and $g(n)$ be positive functions of the non-negative integer n. The big-Oh notation,

$$f(n) = O(g(n)) \tag{2.1}$$

is a short-hand way of saying that the inequality

$$f(n) \leq c * g(n) \tag{2.2}$$

holds for $n \geq n_0$, where n_0 and c are positive constants. In other words, $c * g(n)$ is an upper bound on $f(n)$ when n is sufficiently large, and hence that $f(n)$ grows at most as fast as $c * g(n)$ as n becomes large.

The graph in Figure 2.1 provides a geometric interpretation of this behavior.

By choosing $g(n)$ suitably, we shall use this big-Oh notation as a short-hand way of describing how $f(n)$ behaves for large values of n. Let's explain how.

The dominant term of $f(n)$ is the term whose contribution to $f(n)$ outgrows that of any other term as n becomes large.

For example, the dominant term of

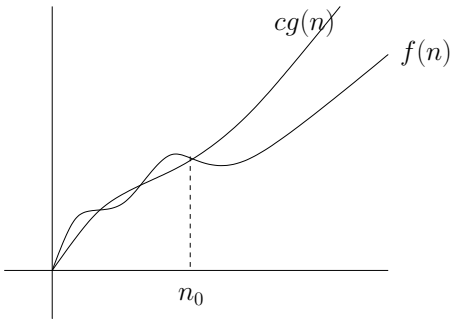


Figure 2.1: Graphical illustration of $f(n) = O(g(n))$

$$f(n) = 2n^3 + n^2 + n + 1 \tag{2.3}$$

is $2n^3$. We can check this out by building a small table:

n	1	n	n^2	$2n^3$	$f(n)$
10^0	1	1	1	2	5
10^1	1	10	100	2000	2111
10^2	1	100	10000	2000000	2010101
10^3	1	1000	1000000	2000000000	2001 001 001

To determine if $a(n)$ is the dominant term of $f(n)$, we just check if for any other term $b(n)$ of $f(n)$, the ratio $b(n)/a(n)$ goes to 0 as n goes to ∞. Thus for the example above, $2n^3$ is the dominant term as $n^2/2n^3, n/2n^3, 1/2n^3$ all go to 0 as n goes to ∞.

Example 2.1. $f(n) = \sqrt{n} + \log n + 1$
The term $\sqrt{n}$ is the dominant term as $\log n/\sqrt{n}, 1/\sqrt{n}$ all go to 0 as n goes to ∞.

Example 2.2. $f(n) = 2^n + n^2 + n \log n + 1$
The term 2^n is the dominant term as $n^2/2^n, n \log n/2^n, 1/2^n$ all go to 0 as n goes to ∞.

Note 2.1. Often, in trying to determine the dominant term of a given function, it is useful to know that $\log n/n^x$ for any fixed $x > 0$ goes to 0 as n goes to ∞.

Once we know the dominant term of a function $f(n)$, we can set $g(n)$ to be this dominant term without the constant factors.

Thus,

$$f(n) = 2n^3 + n^2 + n + 1 \leq 3n^3 \text{ for } n \geq 2$$

Hence $f(n) = O(n^3)$ according to the definition of the big-Oh notation above.

Note that any other function of n that dominates n^3 can also be used to put an upper bound on $f(n)$. Thus,

$$f(n) = n^3 + n^2 + n + 1 \leq n^4 \text{ for } n \geq 2,$$

a fact which we can also write using the big-Oh notation as $f(n) = O(n^4)$.

Example 2.3. For the C-style program fragment below, let $T(n)$ be the number of times the statement $sum = sum + 1$ is executed.

```
for (int i = 1; i ≤ n; i = i + 1)
    for (int j = 1; j ≤ i * i; j = j + 1)
        if (j%i == 0)
            for (int k = 0; k < j; k = k + 1)
                sum = sum + 1;
```

We estimate $T(n)$, using the big-Oh notation. For a fixed value of i and j, the innermost loop is executed j times. For a fixed i, the innermost loop is executed j times, where $j = i, 2*i, 3*i, \ldots, i*i$. Hence the toal number of times the two innermost loops is executed $i + 2 * i + 3 * i + \ldots + i * i$, that is $i * (i * (i + 1))/2$ times, Thus the total number of times $sum = sum + 1$ is executed is: $\Sigma_{i=1}^{i=n}(i * (i * (i + 1))/2)$. Thus $T(n) = O(n^4)$ (see Exercise below).

Subsequently, whenever we use the big-Oh notation to describe the behavior of some function $f(n)$ for large values of n we shall always do so using its dominant term.

Exercise 2.1. It is well-known that $\Sigma_{i=1}^{i=n} i = n(n+1)/2$. Use this result and the identity $i^3-(i-1)^3 = 3 * i^2 - 3 * i + 1$ to show that the sum $\Sigma_{i=1}^{i=n} i^2 = [n(n + 1)(2n + 1)]/6$. Going further, use both these results and the identity $i^4 - (i - 1)^4 = 4 * i^3 - 6 * i^2 + 4 * i - 1$ to show that $\Sigma_{i=1}^{i=n} i^3 = [n(n + 1)/2]^2$. This should explain why $T(n) = O(n^4)$ in the analysis of the program fragment above.

Exercise 2.2. Let $f(n) = n^4 + n^3 + n^2 + 1$. Give a formal proof of the assertion that $f(n) = O(n^4)$.

Exercise 2.3. For the C-style program fragment below, let $T(n)$ be the number of times the statement $sum = sum + 1$ is executed.

---

for (int $i = 0; i < n; i = i + 1$)
    for (int $j = 0; j < n; j = j + 1$)
        for (int $k = 0; k < j; k = k + 1$)
            $sum = sum + 1$;

Give a big-Oh estimate of $T(n)$ in terms of n.

The big-Ω and Θ notations can be defined using the big-Oh notation as below.

Definition 2.1. $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$ (iff is shorthand for if and only if).

Thus if $f(n) = n^2 + 1$ and $g(n) = 2n + 1$ then $f(n) = \Omega(g(n))$

Definition 2.2. $f(n) = \Theta(g(n))$ iff $g(n) = O(f(n))$ and $f(n) = O(g(n))$.

Thus if $f(n) = n^2 + 1$ and $g(n) = n^2 - 1$ then $f(n) = \Theta(g(n))$

Exercise 2.4. $f(n)$ and $g(n)$ are respectively the pairs of functions

1. $n + 2$ and $n + 3$
2. $2n + 3$ and $3n + 2$
3. $n^2 + 3$ and $n^3 + 2$
4. 2^n and 3^n
5. $\log n^2$ and $\log n^3$
6. $\log^2 n$ and $\log^3 n$
7. $2^{\log n}$ and $3^{\log n}$

In each of the above cases determine if $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ or $f(n) = \Theta(g(n))$

2.2 Computational Model and Complexities

With this notational machinery at our disposal, we are ready to address the issues related to the quantification of the behaviour of an algorithm on different inputs. The primary motive behind this is to be able to compare different algorithms that solve the same problem.

As a first step, some observations are in order.

O1 A program that implements an algorithm consumes resources of a m/c on which it is run - time (cpu cycles) and space (memory).

O2 The amount of resources consumed are directly related to the "size of the input"

Assuming that we are able to quantify the "size of an input", and we are trying to estimate the time consumed, we can make a table like this:

Input Size	Time
I_1	T_1
I_2	T_2
.	.
.	.
.	.

Can we use the above table to derive the following function ?

$$T = f(I) \tag{2.4}$$

The difficulty is that there are too many parameters to reckon with in trying to set up an exact interpolation formula. These are made up by the characteristics of the particular:

- m/c on which the program is run
- compiler used
- programming language used to code
- programmer who writes the code

The solution lies in abstraction. We imagine that we have an algorithm machine that runs our algorithms, writen in pseudo-code. On this machine:

- READ's and WRITE's take constant time
- All ARITHMETIC operations take constant time
- All LOGICAL operations take constant time.

Despite these substantially simplifying assumptions, the above question is still very difficult. So we ask a more limited question: Can we quantify the behavior of our algorithm on the worst-possible input ? This kind of analysis is called worst-case analysis. It is possible to do this, and is what is usually done.

2.3 Input Size

We deferred the question of input size. This is not always easy to determine and is often problem-dependent. Below, we give some examples to illustrate how the choices are made in each case.

Example 2.4. For graph-theoretic problems, the number of vertices and the number of edges in a given graph G form a natural measure of the input size.

Example 2.5. For a matrix-multiplication problem ($m \times n$ by $n \times p$) the number of rows, m, the number of columns (rows), n, and the number of columns, p form a natural measure of the input-size.

Example 2.6. For multiplying two positive integers u and v, the numbers of bits in u and v form a natural measure of the input size.

Example 2.7. For sorting a list of n integers, a natural measure is the number of integers n.

Where does the big-Oh notation fit into all this ? We can't determine equation (2.4) exactly, but we can say this:

$$T(I) = O(f(I)) \tag{2.5}$$

where I is a measure of the input size.

Thus when we say that the time-complexity $T(n)$ of the sorting algorithm $mergeSort$ on an input of size n is

$$T(n) = O(n \log n) \tag{2.6}$$

we mean that on no input of size n is the time-complexity of mergesort (this is an algorithm that sorts a list of numbers) worse than a constant times $n \log n$.

2.4 Analysis of some examples

Example 2.8. Given a list of n numbers find their minimum (or, maximum).

The following algorithm can be used to find the, minimum, for example.

Algorithm 2 findMinimum

Input: A list of n numbers
Output: A minimum number, min, from this list

```
1: min ← a_1
2: for i = 2 to n do
3:     if a_i < min then
4:         min ← a_i
5:     end if
6: end for
7: return min
```

In this case it is obvious that

$$T(n) = O(n)$$

since the **for**-loop examines $n-1$ elements after the first to determine the minimum. With some minor changes, the same algorithm can be used to determine the maximum.

Example 2.9. Given n points, $p_1, p_2, \ldots, p_n$, in the plane find a pair of points that are closest to each other.

The following algorithm will accomplish this task.

Algorithm 3 findClosestPair

Input: A list of n points $p_1, p_2, \ldots, p_n$ in the plane
Output: A closest pair of points p_i and p_j

```
1: minDistance ← ∞
2: for i = 1 to n do
3:     for j = i + 1 to n do
4:         if distance(p_i, p_j) < minDistance then
5:             minDistance ← distance(p_i, p_j)
6:         end if
7:     end for
8: end for
9: return minDistance
```

In this case,

$$T(n) = O(n^2)$$

since the algorithm that looks at all pairs of points to determine a closest pair.

Example 2.10. Given n points, $p_1, p_2, \ldots, p_n$, in a plane determine if any three are in straight line.

Three points p, q and r in the plane are collinear if the area of the triangle, $\Delta(p, q, r)$, formed by them is zero. The following algorithm implements this observation.

Algorithm 4 findThreeCollinearPoints

Input: A list of n points $p_1, p_2, \ldots, p_n$ in the plane
Output: A triplet of points p_i, p_j and p_k that are collinear

```
1: minDistance ← ∞
2: for i = 1 to n do
3:     for j = i + 1 to n do
4:         for k = j + 1 to n do
5:             if Area(Δ(p_i, p_j, p_k)) = 0 then
6:                 return (p_i, p_j, p_k) are collinear
7:             end if
8:         end for
9:     end for
10: end for
11: return "No three points are collinear"
```

In this case it is obvious that:

$$T(n) = O(n^3)$$

since the algorithm that looks at all triplets of points to decide if there are three that are collinear.

Note 2.2. There is a faster method for determining a pair of closest points from a given set of n points than examining each pair in turn ? You might want to look at the textbook on Computational Geometry by Preparata and Shamos for a solution.

Problem 2.1. Can you think of a better method for determining if there are 3 collinear points in a given set of n points than examining each triplet in turn ? Let me know if you can.

In the next section, we look into the analysis of a more elaborate example.

2.5 The maxiumum contiguous subsequence problem

Given a sequence of integers

$$a_1, a_2, a_3, \ldots, a_{n-1}, a_n,$$

a contiguous subsequence of the above sequence is:

$$a_i, a_{i+1}, \ldots, a_{j-1}, a_j,$$

where $1 \leq i \leq j \leq n$. In the discussion that follows, by a subsequence we shall mean a contiguous subsequence.

When $i > j$, the subsequence is said to be empty.

The maxiumum subsequence problem is to determine a subsequence such that the sum

$$a_i + a_{i+1} + \ldots + a_{j-1} + a_j$$

is maximum.

By definition, the value of this sum is 0 for an empty subsequence.

Example 2.11. For the sequence

-1, -2, -3, -4, -5, -6,

the maximum subsequence is the empty sequence with value 0.

Example 2.12. For the sequence

-1, 2, 3, -3, 2,

a maximum subsequence is

2, 3

with value 2 + 3 = 5

Note 2.3. Note that there can be several subseqences with the same maximum value. The problem is to find one such subsequence. For example, the sequence

-1, 1, -1, 1, -1, 1

has several maximum subsequences with the maximum value of 1.

We will use this example to show how to progress from a naive algorithm whose run-time is quadratic in the length, n, of the input sequence (referred to succinctly as an $O(n^2)$-time algorithm) to a clever algorithm whose run-time is proportional to the size of the input (referred to succinctly as an $O(n)$-time algorithm), highlighting in the process some issues germane to algorithm design.

2.5.1 An $O(n^2)$ algorithm

All search problems have an associated search space. The first thing that we need to figure out is how large is this search space. For this problem, the specific question is how many different subsequences do we need to examine ? For example, if the given sequence is:

$$-1, 2, 3, -3, 2, \tag{2.7}$$

then the sequences that begin with -1 are:

-1
-1, 2
-1, 2, 3
-1, 2, 3, -3
-1, 2, 3, -3, 2.

Those that begin with 2 are:

2
2, 3
2, 3, -3
2, 3, -3, 2.

Those that begin with 3 are:

3
3, -3
3, -3, 2.

Those that begin with -3 are:

-3
-3, 2

Those that begin with the last 2 (and end with it) are:

2

Thus, including the empty subsequence, we have a total of 16 subsequences to examine.

To generalize, given the sequence:

$$a_1, \ldots, a_{n-1}, a_n$$

we have n subsequences that begin with a_1:

a_1
a_1, a_2
a_1, a_2, a_3
$\vdots$
$a_1, a_2, a_3, \ldots, a_{n-1}, a_n$,

$n-1$ of them that begin with a_2:

$$\begin{array}{l} a_2 \\ a_2, a_3 \\ \vdots \\ a_2, a_3, \ldots, a_{n-1}, a_n, \end{array}$$

and so on, till we have 2 of them that begin with a_{n-1}:

$$\begin{array}{l} a_{n-1} \\ a_{n-1}, a_n \end{array}$$

and, finally, exactly one that begins (and ends) with a_n:

$$a_n$$

Thus we have a total of

$$1 + 2 + \ldots + n + (empty\ subsequence) \quad = \quad n(n+1)/2 + 1 \tag{2.8}$$

subsequences that make up the search space. In the algorithmics jargon, we say that the search space consists of $O(n^2)$ subsequences, as the dominant term in (2.8) is $n^2/2$.

Remark 1. Another, and simpler, way of arriving at the above count is this: there is 1 subsequence of length 0, n subsequences of length 1 and $C(n, 2)$ subsequences of length 2 or more, adding up to a total of $n(n+1)/2+1$ subsequences. To justify the last part of the counting argument, we note that $C(n, 2)$ is the number of distinct ways of choosing a $start$ index and an end index from n indices so that $start < end$.

An algorithm based on a brute-force search of the entire search space is given below:

Algorithm 5 MaxSeqBruteForce

Input: A sequence of integers $a_1, a_2, \ldots, a_n$
Output: Indices $start$ and end of a maximum contiguous subsequence and its sum, $maxSum$.

```
1:  start ← 0                          ▷ Initially, we have an empty sequence with maxSum 0
2:  end ← −1
3:  maxSum ← 0
4:  i ← 1
5:  while (i ≤ n) do                   ▷ i as a potential start index lies in the range [1..n]
6:      sum ← 0
7:      j ← i
8:      while (j ≤ n) do               ▷ j as a potential end index lies in the range [i..n]
9:          sum ← sum + a_j
10:         if (sum > maxSum) then
11:             maxSum ← sum
12:             if (start < i) then
13:                 start ← i
14:             end if
15:             end ← j
16:         end if
17:         j ← j + 1
18:     end while
19:     i ← i + 1
20: end while
21: return start, end, maxSum
```

Discussion:

How do we know that algorithm 5 has looked at all pairs of (i, j) for $1 \leq i \leq j \leq n$? The outer while-loop progresses through all values of i in the range $1 \leq i \leq n$. The inner while-loop progresses through all values of j in the range $i \leq j \leq n$ for a fixed i. Thus we have examined all admissible start and end index-pairs.

Analysis of *MaxSeqBruteForce*

The above algorithm has two nested for loops; the outer one loops over the start position i (lines 5 - 20), while the inner one loops over the end position $j(\geq i)$ (lines 8 - 18). For a fixed i, the cost of executing the inner loop is $c * (n - i + 1)$, where c is assumed to be the cost of executing a single j-loop (lines 8 - 18) and is at most 5.

The cost of a single iteration of the i-loop is $c' + c * (n - i + 1)$, where $c' \leq 5$. Thus the total cost is bounded by $\sum_{i=1}^{i=n} c' + c * (n - i + 1)$. Or in other words, the time-complexity, $T(n)$, where n is the size of the input list, is at most $c' * n + c * (n(n + 1)/2)$. Hence $T(n) = O(n^2)$, as the dominant term in the last expression is $c * n^2/2$.

2.5.2 A faster algorithm

To improve upon the brute-force algorithm we have to think of a way of avoiding looking at all the subsequences. For this we need the following notion. Given a subsequence,

$$a_i, a_{i+1}, \ldots, a_k, a_{k+1}, \ldots, a_j \tag{2.9}$$

the subsequence

$$a_i, a_{i+1}, \ldots, a_k$$

where $i \leq k \leq j$ is said to be a prefix of (2.9). The prefix sum is:

$$a_i + a_{i+1} + \ldots + a_k$$

Now the main observation is that if (2.9) is a maximum subsequence then no prefix sum can be negative. We verify this observation on the example sequence of (2.7).

Based on our observation, we can rule out the subsequences:

-1
-1,2
-1,2,3
-1,2,3,-3
-1,2,3,-3,2

and the subsequences

-3
-3,2

as being possible candidates.

The implication of our observation is that in our algorithm we can skip over portions of the array that stores the given sequence. More specifically,

- If ever $sum < 0$ we can skip over the index positions from $i + 1, \ldots, j$ as candidate starting positions (Why ?)
- Also, if $sum \geq 0$ always for a starting position i, then none of the positions $i + 1, \ldots, n$ is a candidate start position as all prefix sums are non-negative.

Thus the modified search below looks at each element a_i just once.

Consider the following sequence:

$$-1, 2, 3, -6, 7, 8, -9, 10, -11, 6, 4$$

The following table is a simulation of Algorithm 6. Each row describes all the parameters of the algorithm at the end of each iteration of the while loop.

Algorithm 6 MaxSeqImproved

Input: A sequence of integers $a_1, a_2, \ldots, a_n$
Output: Indices $start$ and end of a maximum contiguous subsequence and its sum, maxSum.

```
1:  start ← −1
2:  end ← −1
3:  maxSum ← 0
4:  i ← 1
5:  j ← 1
6:  sum ← 0
7:  while (j ≤ n) do                                   ▷ j ∈ [1..n] and i ≤ j
8:      sum ← sum + a_j
9:      if (sum > maxSum) then
10:         maxSum ← sum
11:         start ← i                                  ▷ reset i and j
12:         end ← j
13:     end if
14:     if (sum < 0) then
15:         i ← j + 1                                  ▷ jump increase of i
16:         sum ← 0
17:     end if
18:     j ← j + 1
19: end while
20: return start, end, maxSum
```

iteration	i	j	sum	$maxSum$	$start$	end
0	1	1	0	0	0	-1
1	2	2	-1	0	0	-1
2	2	3	2	2	2	2
3	2	4	5	5	2	3
4	5	5	0	5	2	3
5	5	6	7	7	5	5
6	5	7	15	15	5	6
7	5	8	6	15	5	6
8	5	9	16	16	5	8
9	5	10	5	16	5	8
10	5	11	11	16	5	8
11	5	12	15	16	5	8

Correctness of *MaxSeqImproved*

Let $i_1, i_2, \ldots, i_k$ be the candidate start positions set by the algorithm above. All other positions cannot be the start position of a maximum subsequence.

Since i_1 is the index of the first positive element from the left, all the elements in the preceding positions being negative cannot be the start of a maximum subsequence. Let l be an index lying strictly between i_j and i_{j+1}. This can't be the start of a maximum sequence for two reasons: first, if it were to include the element just preceding the element at i_{j+1}, then such a sequence would have a negative prefix and hence inadmissible; second, if it did not then since the sum of the elements preceding the one at l including and up to the element at i_j is non-negative, we could have a subsequence of value at least as large as the one that starts at l.

Analysis of *MaxSeqImproved*

Unlike algorithm MaxSeqBruteForce, the improved algorithm has a single for loop that examines each element of the array exactly once. Since the array is of size n, and the work done in the remaining part of the algorithm is in $O(1)$, the time-complexity of the for-loop dominates the complexity $T(n)$ which is therefore in $O(n)$.

Problem 2.2. Let $f(x, n) = \sum_{i=0}^{i=n} x^i$ be a polynomial of degree n; to evaluate $f(x, n)$ at $x = a$ means to find the value of $f(a, n)$. A straightforward algorithm for doing this consists of evaluating each term x^i and then adding up the values of all the terms. The total number of additions and multiplications thus required is in $O(n^2)$. Can you think of a smarter way of doing this so that the total number of additions and multiplications is in $O(n)$? (Hint: Think about writing the polynomial in a different way. For example, when $n = 3$, rewrite $f(x, 3) = x^3 + x^2 + x + 1$ as $f(x, 3) = xf(x, 2) + 1 = x(x(x + 1) + 1) + 1$.)

Problem 2.3. For a given sequence, show that a maximum subsequence can also be found in $O(n)$ time by scanning the sequence from right to left once (Hint: A maximum subsequence cannot have a negative suffix).

Problem 2.4. For a given sequence $a_1, a_2, \ldots, a_n$, design an algorithm that finds the maximum subsequence that starts at a_i, for each $i = 1, 2, \ldots, n$, in linear time.

Chapter 3

Linear lists: Stacks and Queues

In this chapter, we discuss two fundamental data structures that are useful in a wide variety of situations.

3.1 Stacks

A stack is a **LIFO** (**L**ast **I**n **F**irst **O**ut) list of elements. Addition to and deletion from this list are allowed only at one end of this list, known as the stacktop.

Pictorially, a stack looks like this

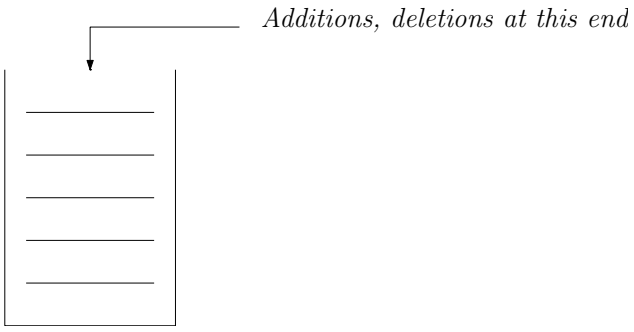


Figure 3.1: A stack, pictorially

Real-life examples of a stack are common enough, like a stack of dishes or a stack of paper. Let us look at some algorithmic problems where a stack is the natural data structure to use.

Balanced parentheses checking

Given a string made up of open "(" and closed ")" parentheses, an interesting problem is to design an algorithm to check if it is balanced. Some examples of strings of balanced parentheses are (), (()) and ()(()). If we look at these examples very carefully we come up with the following recursive definition of all balanced parentheses sequences. The sequence () is balanced. A balanced parentheses sequence of length $2n$, where $n \geq 2$, is either of the form (S), where S is a balanced parentheses sequences of length $2n-2$ or of the form SS, where each S stands for a balanced

parentheses sequence and whose total length is $2n$.

An elegant stack-based decision algorithm works in the following way. We scan the string from left to right. If the current parenthesis is an opening bracket, "(", we push it onto the stack. If it is a closing bracket, ")", and the stack is empty, the string is not balanced, otherwise we pop the opening bracket from the top of the stack. If, on the other hand, we exhaust the string and the stack is not empty then in this case too the string is not balanced. In effect, we match each closing bracket with the nearest opening bracket to its left, and this is to be found on the top of the stack.

For example, if the string is (((()(())()), an opening bracket will be left on the stack when we have exhausted the string.

If the string is)(), we will report that the string is not balanced because we would have encountered a closing bracket when the stack is empty.

For the example string (()), both the stack will be empty and the string will be exhausted since the string is balanced.

We can prove the correctness of the linear-scan algorithm by induction on the length of a balanced parentheses sequence. Clearly, the algorithm works correctly for the sequence (); assume, inductively, that the algorithm works correctly for a balanced parentheses sequence of length $2n$, $n \geq 1$.

Consider a balanced parentheses sequence of length $2n + 2$. Such a sequence is structurally of the form SS or (S). In the first case since the length of the first S is $\leq 2n$, the linear scan algorithm above would discover that it is balanced. By a similar argument it would discover that the second sequence S is also balanced since it is of length $\leq 2n$. In case the structure is of the form (S), the first "(" would be stacked and the algorithm would correctly discover that the nested S is balanced since it is of length $\leq 2n$. When it encounters the final closing parentheses, it would be matched with the opening parenthesis on the top of the stack and hence would correctly determine that (S) is also balanced. This proves the correctness of the algorithm.

Postfix arithmetic expression evaluation

The usual way of writing an arithmetic expression, for example $a + b$, is called its infix form. Another way of writing an arithmetic expression is called the postfix form, where two operands are immediately followed by their operator. The table below shows a list of arithmetic expressions in their infix forms along with their corresponding postfix forms.

Infix	Postfix
$a + b$	$ab+$
$a + b * c$	$abc * +$
$(a + b) * c$	$ab + c*$

As we note from the last line of the above table, the advantage of the postfix form is that we can dispense with brackets as a means of changing the normal rules of operator precedence.

Once again, the stack data structure underlies an interesting algorithm for evaluating an arithmetic expression in postfix form.

The algorithm is simple enough. We scan the given expression, putting operands on the stack until we come across an operator when we pop off the top two operands from the stack, apply the operator to these operands and stack the result.

We illustrate the scheme on the postfix expression: $abc*+$

```
Push a
Push b
Push c
Pop c
Pop b
d = b * c
Push d
Pop d
Pop a
e = a + d
Push e
```

The result of the evaluation, e, is now on the stack.

Implementation of a stack

A stack can be implemented by means of a variety of data structures available in a programming language. Your text, for example, discusses implementations by means of an array as well as a linked list.

However, whatever the implementation, in the analysis of any algorithm that uses a stack we shall assume that the primitive operations of $push$ and pop can be done in $O(1)$ time (this is a standard expression for an operation that takes constant time without committing ourselves to a specific value).

Reversing the digits of a number

Another interesting problem that involves the use of a stack is this: given a non-negative integer n, output a number n' whose digits are the reverse of the digits of the input number. For example, if the input is $n = 1234$, the output should be $n' = 4321$.

Problem 3.1. Using the big-Oh notation, analyze the number of $push$ and pop operations that are done by algorithm ReverseDigit.

3.2 Queues

A queue is a **FIFO** (**F**irst **I**n **F**irst **O**ut) list of elements so that:

- Addition to this list is done at one end, called the $back$ of the queue.
- Deletion from this list is done at the other end called the $front$ of the queue.

Pictorially, a queue looks like in Fig. 3.2.

Algorithm 7 ReverseDigit

Input: A non-negative integer n
Output: Non-negative integer n' whose digits are the reverse of the digits of n

1: **while** $n > 0$ **do**
2: Push $n \bmod 10$ on stack
3: $n \leftarrow n \text{ div } 10$
4: **end while**
5: $n' \leftarrow 0$
6: $pow \leftarrow 1$
7: **while** stack not empty **do**
8: Pop digit d from stack
9: $n' \leftarrow d * pow + n'$
10: $pow \leftarrow pow * 10$;
11: **end while**
12: **return** n'

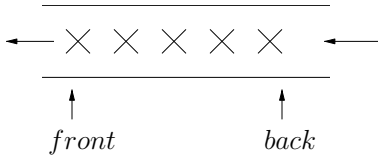


Figure 3.2: A queue, pictorially

Real-life examples of a queue are common enough - a queue at a bus-stop, a printer queue etc. The following example provides a fine illustration of the use of the queue data structure.

A rat, started off in the **START** square of the maze in Fig. 3.3, has to find the shortest path to the **FINISH** square. The shaded squares of the mesh are blocked off to the rat. The length of a path taken by the rat is measured by the number of squares it visits.

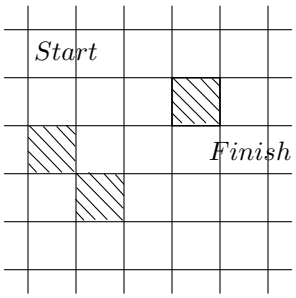


Figure 3.3: A maze

From a given square, a rat can move left, right, up or down, unless blocked by the surrounding wall. Assuming our rat knows about the queue data structure, it does the following. It labels all the squares it can visit from the **START** square with the number 1 and saves these positions at the back of the queue in an arbitrary order.

It then takes off the square from the front of the queue and if this square is labeled i, it marks all the unlabeled squares it can visit by $i + 1$ and stores these squares at the back of the queue in an

arbitrary order. It continues this until it has reached the square marked **FINISH**.

It now determines the shortest path by moving backwards. If the square **FINISH** would have received the label k, it moves back to a square labeled $k-1$; then, from this square, to a square labeled $k-2$ and so on, until it backtracks to the **START** square. Note that each move is possible since to reach a square labeled k the rat would have had to reach a square labeled $k-1$.

The following figure illustrates the algorithm. We have shown a worst case scenario in which among all the squares labeled 5, the one not adjacent to the $finish$ square was examined first.

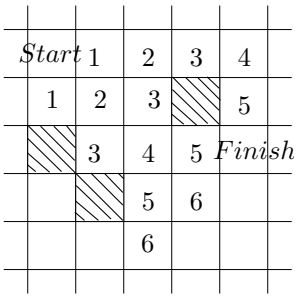


Figure 3.4: Traversal of the example mesh

Implementations of a queue

Some interesting problems arise when we attempt to implement the queue data structure by means of an array. In one solution we fix the $front$ of the queue at the index position 0, letting the $back$ be movable (see Fig 3.5).

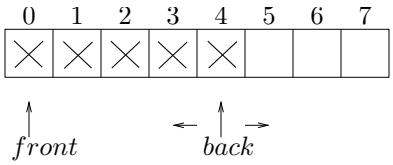


Figure 3.5: Array implementation of a queue: movable $back$

For this implementation, the enqueue operation takes $O(1)$ time, while the dequeue operation takes $O(n)$ time as we are required to move all the elements in the queue one position to the left every time we move an element off the front of the queue. This is rather inefficient.

So, in the second solution we let the $front$ of the queue be variable also (see Fig 3.6).

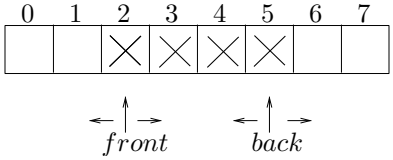


Figure 3.6: Array implementation of a queue: movable $back$ and $front$

This is turn introduces another problem: we can have a lot of unused positions in the queue if we insist that $front \leq back$ always.

The third solution is to allow wraparound by dropping the last requirement so that all empty positions become available for insertion into a queue. What we have now is in effect a circular queue.

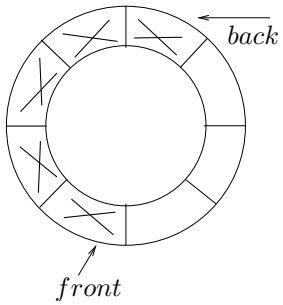


Figure 3.7: Array implementation of a circular queue

The descriptions of the queue operations as well as the boundary conditions that distinguish an empty queue from a full one require some care now.

If the queue is empty, we set the $back$ and $front$ pointers to -1, as we do initially. When the queue is full, the condition $back + 1$ mod $n = front$ holds. When the queue has a singleton element, $back = front > -1$.

Before enqueuing an element, we check if the queue is not full; if not, we add the new element to the queue at the index position

$$back + 1 \text{ mod } n,$$

which also becomes the new value of the pointer $back$. Note also that if the queue is empty, then this should also be set to be the value of the pointer $front$, as we have agreed that both these pointers should have the same value, when the queue has only one element.

Before attempting to dequeue an element, we must make sure that the queue is not empty. If not, we reset the $front$ pointer to:

$$front + 1 \text{ mod } n$$

unless the queue has a single element in it, when the queue becomes empty in which case we set both pointers to -1.

Problem 3.2. Discuss how we would implement a queue, using two stacks.

Chapter 4

Recursive Algorithms

Recursion involves self-reference, as in a recursive definition or in a recursive method. The chain of self-reference is terminated by a base case that we must define separately. Let us illustrate this by some examples.

Example 4.1. The factorial of a non-negative integer n is defined as:

$$\begin{aligned} fact(n) &= n * fact(n-1), \text{ when } n > 0 \\ &= 1, \text{ when } n = 0 \end{aligned}$$

Here $fact(n)$ is defined in terms of $fact(n-1)$, and $fact(0) = 1$ defines the base case.

Exercise 4.1. Compute $fact(5)$ using the above definition.

Example 4.2. The gcd of two non-negative integers, both not 0, is defined as:

$$\begin{aligned} gcd(m,n) &= gcd(n, m \; mod \; n), \text{ when } n > 0 \\ &= m, \text{when } n = 0 \end{aligned}$$

If we assume that $m > n$, the second argument $m \; mod \; n$ of gcd on the right-hand side is smaller than the second argument of n on the left-hand side. The base case is when the second argument is 0, and $gcd(m, 0) = m$

Exercise 4.2. Compute $gcd(15, 10)$ using the above definition.

Example 4.3. The product of two positive integers m and n is defined as:

$$\begin{aligned} P(m,n) &= m + P(m, n-1), \text{when } n > 1 \\ &= m, \text{when } n = 1 \end{aligned}$$

Here $P(m, n)$ is defined in terms of $P(m, n-1)$ and the base case is for $n = 1$

Exercise 4.3. Compute $P(5, 3)$ using the above definition.

Example 4.4. A classical example of a problem that is nicely solved by recursion is the Towers of Hanoi problem: n disks are stacked on a peg A in order of decreasing size. We are to move these disks to a peg C so that they are in the same decreasing order of size, using a third peg B as workspace, and observing the following rules:

Rule 1 We can move only one disk at a time (single move).

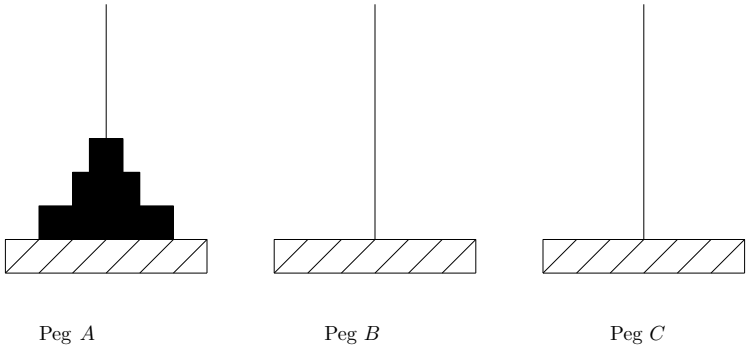


Figure 4.1: Towers of Hanoi: Initial configuration

Rule 2 We may not place a larger disk atop a smaller one in any move.

Pictorially, the initial situation is described in Fig. 4.1, while the final situation is shown in Fig. 4.2.

The recursive solution to the above problem can be expressed in the following way:

Move(n, A, C, B) := Move($n-1, A, B, C$) + Move(1, A, C, B) + Move($n-1, B, C, A$)

where the notation Move(i, X, Y, Z) means a move of i disks from peg X to peg Y, according to the above rules, using peg Z as workspace, and "+" means "followed by".

Exercise 4.4. Let $M_r(n)$ be the number of single moves made by the above recursive algorithm to move n disks from peg A to the peg C, following the stated rules. Show that $M_r(n) = 2^n - 1$.

Exercise 4.5. Show that the minimum number of moves required to move n disks from peg A to peg C is $2^n - 1$.

Example 4.5. Another nice example of recursion involves printing the digits of a number n from left to right.

The problem here is that we can extract the digits in the reverse order in which we wish to print them. This suggests that we postpone printing the least significant digits till we have printed the digits that are more significant. Recursion is an effective way of achieving this goal.

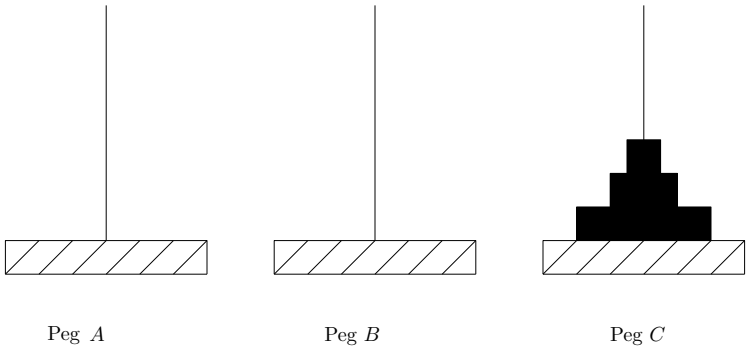


Figure 4.2: Towers of Hanoi - Final configuration

Algorithm 8 PrintDigits(n)

Input: A positive integer n
Output: The digits of n from left to right

```
1: if (n ≥ 10) then
2:     PrintDigits(n/10)
3: end if
4: Print n mod 10
5: return
```

Recursion is implemented by using a stack of activation records. Each activation record stores all parameters related to the present call so that it can be completed when we return to this call. We need a stack because calls are completed in the reverse order in which these are generated.

Recursion is a powerful tool; however, we need to be aware of its limitations. We discuss an example below to illustrate this point.

The Fibonacci sequence can be defined recursively as follows:

$$F_n = F_{n-1} + F_{n-2}, n > 1 \tag{4.1}$$

$$F_0 = 1; F_1 = 1 \tag{4.2}$$

Based on the above recursive definition, it can be shown that $F_n = 2^{0.694n}$.

The first few Fibonacci numbers are:

1, 1, 2, 3, 5, 8, 13, 21,....

A recursive algorithm for generating the above sequence is easy to design.

Algorithm 9 RecursiveFibonacci(n)

Input: A non-negative integer n
Output: The n-th Fibonacci number

```
1: if (n ≤ 1) then
2:     return 1
3: end if
4: return RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2)
```

This is not a good use of recursion as the following tree of recursive calls for computing F_5 shows that a call to the same function is made from different places in the tree. For example, the call to compute F_1 is made from 5 different nodes in the tree.

Problem 4.1. Design an iterative scheme for computing the nth Fibonacci number for a given non-negative integer n.

Problem 4.2. Consider the following matrix-based scheme for computing Fibonacci numbers:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

From this it follows that:

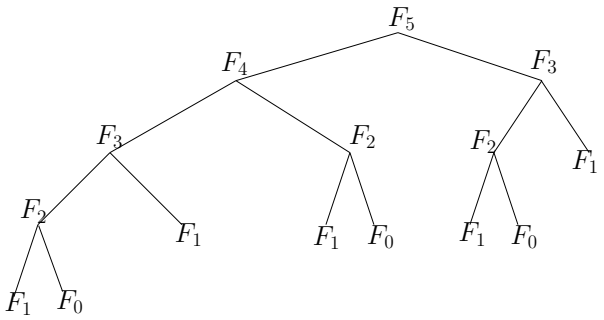


Figure 4.3: Tree of recursive calls to compute F_5

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

And in general, that

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Show that F_n can be computed in $O(\log n)$ time, if we assume all arithmetic operations can be done in $O(1)$ time.

4.1 Divide-and-Conquer

This is an important paradigm in algorithm design. The idea is quite simple. Let us assume that the"size" of the problem can quantified (refer to discussion in Chapter 1). Divide the given problem into a set of "smaller-sized" non-overlapping sub-problems. Conquer by stitching together solutions of sub-problems (obtained by a recursive application of the same principle) into a solution of the original problem.

Let us apply this paradigm to the old problem of computing a Maximum Contiguous Subsequence of a given sequence of integers.

We apply the first part of the principle to divide the given sequence

$$S = a_1, a_2, \ldots, a_{n-1}, a_n$$

into two halves, a left half:

$$S_l = a_1, a_2, \ldots, a_{\lceil n/2 \rceil}$$

and a right half:

$$S_r = a_{\lceil n/2 \rceil + 1}, a_{\lceil n/2 \rceil + 2}, \ldots a_n$$

We now apply the second part of the principle. Find a maximum subsequence of each half and use these solutions to obtain a solution of the original sequence.

Here we have to consider three different possibilities. The first is that the maximum subsequence of the original sequence is contained either entirely in the first half as it is for the sequence -1,-2, 3,

4, -5,-6,-7,-8; the second is that it is contained entirely in the second half as for the sequence -1,-2, -3, -4, 5, 6,-7,-8; the third possibility is that it straddles both halves as for the sequence -1,-2, 3, 4, 5, 6,-7,-8 and the conquer part of this paradigm is to figure out a way of determining such a straddling sequence.

The primary observation is this:

Observation 1. If $a_i, a_{i+1}, \ldots, a_{j-1}, a_j$ is a maximum subsequence that straddles both halves, then $a_i + a_{i+1} + \ldots + a_{\lceil n/2 \rceil}$ has the maximum value of all subsequences from the left half that end with $a_{\lceil n/2 \rceil}$, and $a_{\lceil n/2 \rceil + 1} + \ldots + a_j$ has the maximum value of all subsequences from the right half that begin with $a_{\lceil n/2 \rceil + 1}$.

This is quite obvious as otherwise such a straddling sequence cannot have maximum value. So we determine the maximum of the following sums:

$$
\begin{aligned}
& a_{\lceil n/2 \rceil} \\
& a_{\lceil n/2 \rceil} + a_{\lceil n/2 \rceil - 1} \\
& \vdots \\
& a_{\lceil n/2 \rceil} + a_{\lceil n/2 \rceil - 1} + \ldots + a_2 + a_1
\end{aligned}
$$

Let this be max_l.

We then determine the maximum of the following sums:

$$
\begin{aligned}
& a_{\lceil n/2 \rceil + 1} \\
& a_{\lceil n/2 \rceil + 1} + a_{\lceil n/2 \rceil + 2} \\
& \vdots \\
& a_{\lceil n/2 \rceil + 1} + a_{\lceil n/2 \rceil + 2} + \ldots + a_{n-1} + a_n
\end{aligned}
$$

Let this be max_r.

The maximum subsequence that straddles both halves is the one corresponding to the sum $max_l + max_r$. It does not matter if one of max_l or max_r is negative, since, quite obviously, in these cases the maximum straddling sequence cannot be a maximum subsequence of S.

A maximum subsequence of the original sequence is the one corresponding to the maximum of MCS of S_l, MCS of S_r, and $max_l + max_r$.

The algorithm can be formally described thus:

Algorithm 10 MaxSubSeqDivideNConquer($S[l..r]$)

Input: A sequence, S of $r - l + 1$ integers
Output: A maximum contiguous subsequence that begins at $start$ and terminates at end and has sum $maxSum_{l,r}$

1: **if** $(l = r)$ **then**
2: $start \leftarrow l$
3: $end \leftarrow r$
4: **return** $start, end$
5: **else**
6: $mid \leftarrow (l + r)/2$
7: $(maxSum_{l,mid}, start_l, end_l) \leftarrow MaxSubSeqDivideNConquer(S[l, mid])$
8: $(maxSum_{mid+1,r}, start_r, end_r) \leftarrow MaxSubSeqDivideNConquer(S[mid + 1, r])$
9: $(maxSuffix, l_{suffix}) \leftarrow maxSuffix(S[l, mid])$
10: $(maxPreix, r_{prefix}) \leftarrow maxPrefix(S[mid + 1, r])$
11: **if** $(maxSum_{l,mid} - maxSum_{mid+1,r}) > 0$ and $(maxSum_{l,mid} - (maxSuffix + maxPrefix)) > 0$ **then**
12: **return** $(maxSum, l, mid)$
13: **end if**
14: **if** $(maxSum_{mid+1,r} - maxSum_{1,mid}) > 0$ and $(maxSum_{mid+1,r} - (maxSuffix + maxPrefix)) > 0$ **then**
15: **return** $(maxSum, mid + 1, r)$
16: **end if**
17: **if** $((maxSuffix + maxPrefix) - maxSum_{1,mid}) > 0$ and $((maxSuffix + maxPrefix) - maxSum_{mid+1,r}) > 0$ **then**
18: **return** $(maxSuffix + maxPrefix, l_{suffix}, r_{prefix})$
19: **end if**
20: **end if**

We run the above algorithm on the example sequence below:

-1, 2, 3, 4, 5, -6

The two halves are:

-1, 2, 3

and

4, 5, -6

A maximum subsequence in the first half is:

2, 3

while one in the second half is:

4, 5

A maximum subsequence that straddles both halves is:

2, 3, 4, 5

The latter is certainly a maximum subsequence of the input sequence.

Complexity Analysis

Theorem 4.1. For an n-element sequence the time-complexity of finding a maximum straddling sequence is $O(n)$.

Proof. We look at each element exactly once. This is because max_l is found by a single scan from the middle of the sequence to the left-end, while max_r is found by a single scan from the middle to the right-end. □

Let $T(n)$ be the worst-case time complexity of finding a maximum subsequence of an n-element sequence. It satisfies the following recurrence relation:

$$\begin{aligned} T(n) &= 2T(n/2) + cn, \text{ when } n > 1 \text{ for some constant } c > 0 \\ &= 1, \text{ when } n = 1 \end{aligned}$$

where $T(n/2)$ on the right is the cost of solving a problem of size $n/2$, while cn is the cost of determining a straddling maximum subsequence.

There are several ways of solving a recurrence of the above type. We discuss two such methods.

Method 1

To keep the math simple, let us assume that n is a power of 2. If we draw the tree of recursive calls corresponding to the above recurrence, we observe that this tree has depth $O(\log n)$, while at each level of the tree the cost of the conquest steps for all the subproblems add up to $O(n)$. Thus $O(n \log n)$ in total.

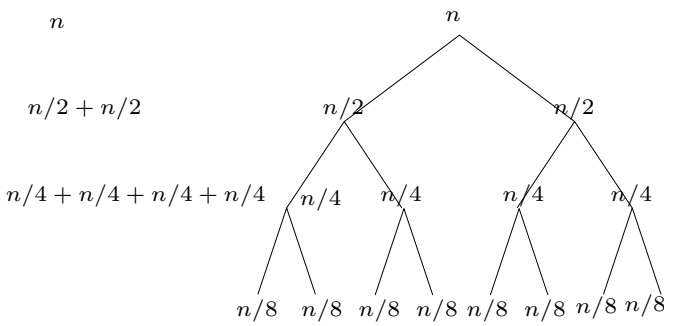


Figure 4.4: Solving a recurrence from the tree of recursive calls

Method 2

Substituting $n/2, n/4, \ldots,$ in the above recurrence, we get:

$$\begin{aligned} T(n/2) &= 2T(n/4) + cn/2 \\ T(n/4) &= 2T(n/8) + cn/4 \end{aligned}$$

.

$$\vdots$$
$$T(2) = 2T(1) + c * 2$$

Multiplying the equalities by $2^0, 2^1, 2^2, \ldots, 2^{\log n - 1}$ respectively, and adding them up we get, after some elementary manipulation, that:

$$T(n) = 2^{\log n} * T(1) + cn + 2 * cn/2 + 2^2 * cn/2^2 + \ldots + 2^{\log n - 1} * c * 2 <= cn \log n$$

Using the big-Oh notation, we can express the solution to the above recurrence as $T(n) = O(n \log n)$. We look at another problem that can be effectively solved, using the divide-and-conquer paradigm.

`Find the maximum and minimum of a set` S `of` n `elements.`

A naive solution would be to find the maximum in $n-1$ comparisons and the minimum in another $n-2$ comparisons (since the minimum can't be the maximum), for a total of $2n-3$ comparisons.

The interesting question is whether we can do better than this. In the sense of asymptotic complexity, of course, we can't since any method that finds the minimum would need to look at all the n elements. What we want to know is if we can reduce the factor 2 of n in the above total. The answer to this is yes.

The insight is that if we divide the n elements into two groups of $n/2$ each; and solve the problem for each group then the maximum of the entire set is the greater of the two maxima and the minimum is the smaller of the two minima.

If we let $T(n)$ denote the number of comparisons for a set of n elements, we have the following recurrence relation.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 2, \ n > 2 \\ &= 1, \ n = 2 \end{aligned}$$

Exercise 4.6. Show that $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$, for all n.

The solution to the above recurrence is: $T(n) = \lceil 3n/2 - 2 \rceil$. We are not to going to prove this, but we can't do better, as $\lceil 3n/2 - 2 \rceil$ comparisons are necessary as well as sufficient.

4.2 Dynamic Programming

The idea of solving a problem by putting together solutions of subproblems is something that we have already seen in the context of the divide-and-conquer paradigm. An important reason why this worked very well was that the number of subproblems generated was a linear function of the size of the problem; furthermore, each of these subproblems was a new one. Such an approach however may not always work. Consider the following problem:

Example 4.6. Given coins of denominations $C_1, C_2, \ldots, C_n$, make up change for k cents, using the fewest number of coins of the given denominations.

In order that there always be one way of making change for k cents, we assume that $C_1 = 1$. Taking a cue from the divide-and-conquer strategy, we attempt to find optimum ways of making change for r cents and $k-r$ cents, which are then combined to yield a way of making k cents in change. The problem is that we don't know what value we should choose for r. Really, there is no alternative

but to try all possible values of r in order to find the optimum solution.

Let $min(k)$ denote the minimum number of coins needed to make change for k cents. Then the left hand side $min(k)$ of the equation below is determined

$$min(k) = min_{0 \leq r \leq \lfloor k/2 \rfloor}\{min(r) + min(k - r)\} \tag{4.3}$$

if we know the values of $min(1), min(2), \ldots, min(k-1)$ that appear on the right. However, in trying to determine each one of the latter recursively by an equation of the same type, we must avoid solving the same problem again and again. This is the essence of the dynamic programming paradigm: it solves and records in a table the solutions to all smaller-sized problems that may be required by the current one.

In this particular case, an optimum way of making change for k cents includes at least one coin from among the n different denominations. This observation allows us to write the right-hand side of equation 4.3 in the simpler way shown below:

$$min(k) = min_{1 \leq i \leq n \& k \geq C_i}\{1 + min(k - C_i)\} \tag{4.4}$$

Indeed, the above simpler form allows us to determine the exact coins that make up the change for k cents.

Note that in equation 4.3 above we include the value 0 for r in the above range in order to accommodate the cases in which $k = C_1$ or C_2, ... or C_n.

Example 4.7. Let us make all this concrete by letting

$$C_1 = 1, C_2 = 5, C_3 = 10, C_4 = 25.$$

Since there are no ways of making change for 0 cents,

$$min(0) = 0$$

Using the simpler equation 4.4, we have

$$min(1) = min\{1 + min(1 - 1)\} = 1$$

As we already know the value of $min(1)$,

$$min(2) = min\{1 + min(2 - 1)\} = 1 + 1 = 2$$

Again, knowing the value of $min(2)$,

$$min(3) = min\{1 + min(3 - 1)\} = 3$$

Similarly, $min(4) = 4$.

Since there is a coin of denomination 5 cents, and we know the values of $min(0)$ and $min(4)$,

$$min(5) = min\{1 + min(5 - 1), 1 + min(5 - 5)\} = 1$$

For $k = 6$ we have:

$$min(6) = min\{1 + min(6 - 1), 1 + min(6 - 5)\} = 2$$

Algorithm 11 CoinChange

Input: The parameters $changeAmount$, $numOfDenominations$ and $valueOfDenomination[]$
Output: The arrays $minCoinsUsed[]$ and $lastCoinUsed[]$

```
1: Initialize minCoinsUsed[]                                   ▷ All entries initialized to 0's
2: Initialize lastCoinUsed[]                                   ▷ All entries initialized to 0's
3: cents ← 1                                  ▷ Variable used to count up to changeAmount
4: while (cents ≤ changeAmount) do
5:     minCoins ← cents   ▷ Initial value of the minimum numbers of coins   ▷ needed to make
   change for the amount cents
6:     newCoin ← 1                                             ▷ The first denomination to try
7:     j ← 1                          ▷ Counter for the number of denominations used so far
8:     while (j < numOfDenominations) do
9:         if (valueOfDenomination[j] ≤ cents) then
10:            if (minCoinsUsed[cents − valueOfDenomination[j]] + 1 < minCoins) then
11:                minCoins ← minCoinsUsed[cents − valueOfDenomination[j]] + 1
12:                newCoin ← valueOfDenomination[j]
13:            end if
14:        end if
15:        j ← j + 1                                           ▷ Next higher denomination
16:    end while
17:    minCoinsUsed[cents] ← minCoins
18:    lastCoin[cents] ← newCoin
19:    cents ← cents + 1
20: end while
21: return coinsUsed[], lastCoin[]
```

and similar equations for k = 7, 8, and 9

For $k = 10$, we have as expected:

$$min(10) = min\{1 + min(10-1), 1 + min(10-5), 1 + min(10-10)\} = 1$$

and so on.

Exercise 4.7. Use equation 4.4 to show that the minimum number of coins needed to make change for 24 cents is 6 and the coins that make up this change is 2 dimes and 4 pennies.

Below, we describe the above algorithm more formally, based on equation 4.4. For a given value of k it not only finds an optimum number, but also the coins that make up this number by recording in an array the C_i for which the minimum in equation 4.4 is obtained.

Complexity Analysis

If $T(n, k)$ is the time complexity of the algorithm CoinChange then $T(n, k) = O(nk)$, where n is the number of coins of different denominations and k is the amount for which change is to be made. This follows from the fact that in the algorithm above the index of the outer loop goes from 1 to k, while for each value of this index the inner loop tries out all the n denominational coins to determine the optimum for the current value of change.

Let us discuss the application of this paradigm to another example.

Example 4.8. Let $A_n = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, ..., a_{n-1}, a_n\}$, be n symbols over some finite alphabet set. A subsequence of the above sequence is:

$$a_{i_1}, a_{i_2}, a_{i_3}, a_{i_4}, ..., a_{i_k},$$

where $1 \leq i_1 < i_2 < i_3 < ... < i_k \leq n$.

For example, if the sequence is: a, b, c, a, d, b then a, c, a and a, b, b and a, b, d, b are some of its subsequences.

Definition 4.1. A $prefix$ of the given sequence is a subsequence consisting of k initial elements of the sequence, where $1 \leq k \leq n$. We denote such a subsequence by A_k.

Definition 4.2. The $length$ of a sequence or subsequence is the number of symbols in the sequence.

The $longest\ common\ subsequence$ (LCS) problem is this: Given two sequences over a common alphabet set:

$$A_n = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, ..., a_{n-1}, a_n\}$$

and

$$B_m = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, ..., b_{m-1}, b_m\}$$

find a subsequence common to both that is the longest.

For example, if a, b, b, a, d is another sequence over the alphabet set $\{a, b, c, d, ..\}$ then a, b and a, b, b and a, b, a, d are all common subsequences, the last being the longest.

The following theorem is fundamental to a dynamic programming approach to the solution of the problem. Let $C_k = \{c_1, c_2, ..., c_k\}$ be an LCS of the given sequences.

Theorem 4.2. If $a_n = b_m$ then C_{k-1} is an LCS of A_{n-1} and B_{m-1}. If $a_n \neq b_m$ then C_k is an LCS of A_n and B_{m-1} or an LCS of A_{n-1} and B_m

Letting $L[A_i, B_j]$ denote the length of an LCS of prefixes $A_i = \{a_1, a_2, \ldots, a_j\}$ and $B_j = \{b_1, b_2, \ldots, b_j\}$, the above theorem is a basis for the claim below:

Claim: $L[A_i, B_j] = 0$, when $i = 0$ or $j = 0$
$L[A_i, B_j] = L[A_{i-1}, B_{j-1}] + 1$, when $a_i = b_j$
$L[A_i, B_j] = max\{L[A_i, B_{j-1}], L[A_{i-1}, B_j]\}$, when $a_i \neq b_j$

It also suggests the following algorithm for computing a longest common subsequence, where we use the simpler notation $L[i, j]$ instead of $L[A_i, B_j]$. We store the lengths $L[i, j]$ in a two-dimensional array. As we need to know the values of $L[i-1, j-1]$, $L[i, j-1]$ and $L[i-1, j]$ before we can compute $L[i, j]$ we fill this matrix in row-major order. To be able to reconstruct an LCS we also have another array which stores one of the symbols l, d or u which is to indicate if we must move left, diagonally or up to find the next matching positions.

Complexity Analysis

If $T(m, n)$ is the time complexity of $LCS(X, Y)$ then $T(m, n) = O(mn)$.

Algorithm 12 LongestCommonSubsequence

Input: Sequences X and Y
Output: Longest Common Subsequence of X and Y

```
1: m ← length(X);
2: n ← length(Y);
3: i ← 0;
4: j ← 0;
5: for i ← 0 to n do
6:     L[i, 0] ← 0
7: end for
8: for j ← 0 to m do
9:     L[0, j] ← 0
10: end for
11: for i ← 1 to n do
12:     for j ← 1 to m do
13:         if (a_i = b_j) then
14:             L[i, j] = L[i − 1, j − 1] + 1
15:             P[i, j] = "d"
16:         end if
17:         if (L[i − 1, j] ≥ L[i, j − 1]) then
18:             L[i, j] = L[i − 1, j]
19:             P[i, j] = "u"
20:         end if
21:         if (L[i − 1, j] < L[i, j − 1]) then
22:             L[i, j] = L[i, j − 1]
23:             P[i, j] = "l"
24:         end if
25:     end for
26: end for
```

Example 4.9. To reinforce the idea of dynamic programming, let us discuss another example. Recall that a matrix is a two-dimensional array. For example,

$$M = \begin{bmatrix} 2 & 3 & 4 \\ 4 & 6 & 5 \end{bmatrix}$$

is a 2×3 array of integers, since it has 2 rows and 3 columns.

If M' is another (integer) matrix, then the matrix product $M \times M'$ is defined provided the number of rows of M' is the same as the number of columns of M. Thus if we let

$$M' = \begin{bmatrix} 2 & 3 \\ 4 & 6 \\ 5 & 7 \end{bmatrix}$$

then

$$M \times M' = \begin{bmatrix} 2*2+3*4+4*5 = 36 & 2*3+3*6+4*7 = 42 \\ 4*2+6*4+5*5 = 57 & 4*3+6*6+5*7 = 83 \end{bmatrix}$$

The number of scalar multiplications (a scalar multiplication is the multiplication of two entries of M and M') made in computing the matrix product is $2 \times 3 \times 2 = 12$. In general if M is an $p \times q$ matrix and M' is an $q \times r$ matrix then the number of scalar multiplications done in computing $M \times M'$ is $p \times q \times r$.

The problem that we want to solve is the following:
Find the minimum number of scalar multiplications required to compute the product $M_1 \times M_2 \times \ldots \times M_n$, where M_i for $i = 1, \ldots n$ is an $r_{i-1} \times r_i$ integer matrix.

You might be wondering if there is any problem here at all. There is! The number of scalar multiplications depends on how we group the matrices for computing the product. For example if we had to compute the product $M_1 \times M_2 \times M_3$, where M_1 is a 2×3 matrix, M_2 is a 3×4 matrix, and M_3 is a 4×5 matrix, then we make 64 scalar multiplications if we first multiply M_1 and M_2 and the result with M_3; on the other hand, we make 90 scalar mutiplications if we first multiply M_2 and M_3 and then M_1 with the result.

To see why dynamic programming is the appropriate tool to solve this problem, let C_{ij} be the optimal number of scalar multiplications needed to compute the product of the matrix subchain $M_i \times M_{i+1} \times \ldots \times M_j$. Then

$$C_{ij} \quad = \quad min_{i \leq k < j}(C_{ik} + C_{(k+1),j} + r_{i-1} * r_k * r_j) \tag{4.5}$$

The equation 4.5 tells the whole story. We first compute the number of scalar multiplications needed to compute all matrix subchains of size 2, namely $M_1 \times M_2$, $M_2 \times M_3$ and so on, storing these values in a table. Using equation 4.5, this allows us to compute the optimal number of scalar multiplications needed to compute the products of subchains of size 3, viz. $M_1 \times M_2 \times M_3$, $M_2 \times M_3 \times M_4$, etc.

It is not hard to see that the number of scalar multiplications needed to compute C_{1n} is in $O(n^3)$. For each fixed value of k in 4.5 above, we need 2 scalar multiplications and thus $2 * (j - i)$ scalar multiplications to determine C_{ij}. Setting $j - i = k$, we note that for each $k \geq 1$ we compute $n - k$ different matrix subchain products of length k. Thus the total number of scalar multiplications needed to compute C_{1n} is $\sum_{k=1}^{n-1}(n - k) * 2k = O(n^3)$.

Problem 4.3. Given a convex n-gon, we can triangulate this polygon in mnay different ways (How many ?). For example, a convex quadrilateral can be triangulated in two ways, a convex pentagon can be triangulated in 5 different ways. The cost of each triangulation is the total length of all the edges of this triangulation (that is, all the chords plus the sides of the convex n-gon). How would you use dynamic programming to find a triangulation of minimum cost ?

Problem 4.4. Write recursive program to generate all the permutations of $1, 2, 3, \ldots, n$. For example, if $n = 3$ the output should be

1,2,3
2,1,3
1,3,2
3,1,2
3,2,1
2,3,1

Problem 4.5. The algorithm for the coin-change problem is a pseudo-polynomial time algorithm. Use dynamic programming, to design a pseudo-polynomial time algorithm for the following partition

problem. Given a set $A = \{a_1, a_2, \ldots, a_n\}$ and weights $w(a_i), i = 1, 2, \ldots, n$ in Z^+ (set of positive integers), find a subset A' of A such that $\Sigma_{a_i \in A'} w(a_i) = \Sigma_{a_i \in A - A'} w(a_i)$.

Remark: The above problem shows that even though a problem is provably NP-complete, it does not rule out the possibility of designing a pseudo-polynomial time algorithm for such a problem.

4.2.1 Sequence Alignment

A DNA sequence is made up of the nucleotides A, C, G and T. An important problem in Computational Biology or Bioinformatics is to determine the similarity of two such sequences. This is done by aligning the two sequences and scoring this alignment. We would like to obtain an alignment with the highest score. Let us explain with the help of an example what is meant by alignment and how to score it . Consider the two nucleotide sequences $v = ATGTTAT$ and $w = ATCGTAC$. An alignment of the two sequences is shown in the table below:

A	T	G	-	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Each column shows a match (same nucleotide in both rows), a mismatch (different nucleotides in the two rows) or an indel (a blank in one of the rows). The score of such an alignment is $\mu * \#indels + \delta * \#mismatches + \gamma * \#matches$, where μ is the indel penalty, δ is the score of a mismatch and γ is the score of match.

An alignment of maximum score can be found by dynamic programming. This is done by building a grid graph Table 4.1.

	A	T	C	G	T	A	C
A							
T							
G							
T							
T							
A							
T							

Table 4.1: Grid Graph

Every path in the grid graph from (0,0) to (7,7) corresponds to an alignment. We find the path corresponding to an optimal alignment by computing the score $s_{i,j}$ at a grid vertex (i, j) based on the following formula:

$$s_{i,j} = max \begin{cases} s_{i-1,j} + \mu \\ s_{i,j-1} + \mu \\ s_{i-1,j-1} + \delta \\ s_{i-1,j-1} + \gamma \end{cases},$$

with $s_{0,0} = 0$ and $s_{0,j} = \mu * j$ and $s_{i,0} = \mu * j$.

Letting $\mu = -1$, $\delta = 0$ and $\gamma = 1$, the path corresponding to an optimal alignment of v and w is shown in the same grid graph.

4.3 Greedy paradigm

It is an interesting question whether we can reach a global optimum by a process of incremental local optimization. This is the basic idea underlying the greedy paradigm. It is, however, not always guaranteed to succeed.

Coin-changing

Consider again the coin-changing problem. If we have to make change for 31 cents with coins of denominations 1, 10 and 25 the solution based on the greedy paradigm is 25 + 1 + 1 + 1 + 1 + 1 + 1, requiring 7 coins. however, the optimal solution 10 + 10 + 10 + 1, needs only 4 coins.

Container loading

Consider another somewhat similar problem. Given a set of n containers of weights $w_1, w_2, \ldots, w_n$, we are required to design a loading scheme that puts the maximum number of these into the hold of a ship that can accommodate a total weight of W. Assume that $w_i \leq w_{i+1}$, $1 \leq i \leq (n-1)$.

A scheme based on the greedy paradigm always loads the $lightest$ of the remaining containers, till no more can be loaded, without exceeding capacity. This means that we load in the container of weight w_1 first, provided $w_1 \leq W$, then the container of weight w_2, provided $w_2 \leq W - w_1$ and so on.

The argument that this loads in the maximum number of containers, compared to any other scheme is not straightforward. Let the greedy solution be represented by the vector $(y_1, y_2, \ldots, y_n)$, where $y_i = 1$ if the i-th container is selected and 0 otherwise. From the nature of the greedy solution it is clear that for some k, $y_i = 1$, when $i \leq k$ and $y_k = 0$, for $i > k$. Let the vector $(z_1, z_2, \ldots, z_n)$ represent any other solution, where each z_i is either 0 or 1. We claim that

$$\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} z_i.$$

Let p be the number of index positions i such that $y_i \neq z_i$. We prove the above claim by induction on p. When $p = 0$, the solutions are identical and the claim is trivially true. Let the claim be true for $p = m$, for some integer $m(\geq 0)$. Assume that the solutions differ in $m + 1$ positions. Since $m + 1 > 0$, there exists an index $i > 0$ such that $y_i \neq z_i$. Let i be the smallest such index. Clearly, $i \leq k$, else it would imply that the greedy solution is not complete (Why ?). Since $y_i \neq z_i$, $z_i = 0$; we reset z_i to 1. The solution represented by the changed vector is either infeasible (total weight of selected containers exceeds W), or if feasible, differs from the greedy solution in m places. In the latter case, the claim is true by the inductive hypothesis. If it is infeasible, there must be some j, where $k < j \leq n$, such that $z_j \neq 0$. We set z_j to 0 and thereby create a feasible solution, in which we replace container w_j by w_i. Let the new solution be represented by the vector $(z'_1, z'_2, \ldots, z'_n)$. Since the new solution differs from the greedy solution in at most m places,

$$\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} z'_i = \sum_{i=1}^{n} z_i$$

This proves the claim and shows that the greedy paradigm works for this problem.

Fractional Knapsack

The greedy paradigm also works for the following problem, known in the literature as the fractional knapsack problem. We are given a knapsack of capacity (volume) C, and a set of n objects whose volumes are $c_1, c_2, \ldots, c_n$, $0 < c_i \leq 1$ for each i, having values $v_1, v_2, \ldots, v_n$ respectively. We have to fill the knapsack with a choice of objects that maximizes the total value. We are permitted to choose fractional amounts of each object.

The question is do we go greedy by volumes of the objects or their weights or something else ? It turns out that in this case the greedy paradigm succeeds if we fill in the knapsack with objects of decreasing value per unit volume. Let's look at an example. We have five objects of capacities 10 ccms, 20 ccms, 30 ccms, 40 ccms and 50 ccms with respective values \$20, \$30, \$66, \$40 and \$60. The capacity of the knapsack is 100 ccms. If we go greedy by volume, each time selecting the item of least volume as in the previous example, the value of the items that fill the knapsack is $\$156$. If we go greedy by value, selecting the item of largest value from the remaining items, the total value of the items selected is $\$146$. However, if we select items in the the order of largest value per unit volume the selected items add up to a value of $\$164$.

This can be proved. Mathematically, we can restate the problem in this way:

Maximize $\sum_{i=1}^{n} x_i v_i$ subject to the constraint that $\sum_{i=1}^{n} x_i c_i \leq C$, where $0 \leq x_i \leq 1$.

The proof goes as follows:

Let $(x_1, x_2, x_3, \ldots, x_n)$ represent the greedy solution. Clearly the value is maximum if all the x_i's are 1. So assume there is a j such that $0 < x_j < 1$, $x_i = 1$ for all $i < j$, while $x_i = 0$ for all $i > j$.

Let $(y_1, y_2, y_3, \ldots, y_n)$ be any other solution. Now the difference in the values of these two solutions is given by:

$$\sum_{i=1}^{n}(x_i - y_i)v_i = \sum_{i=1}^{n}(x_i - y_i)c_i v_i/c_i$$

Now for $i < j$, $x_i - y_i \geq 0$, while $v_i/c_i \geq v_j/c_j$ so that $(x_i - y_i)v_i/c_i \geq (x_i - y_i)v_j/c_j$. For $i > j$, since $(x_i - y_i) < 0$, and $v_i/c_i \leq v_j/c_j$, the last inequality still holds. Thus

$$\sum_{i=1}^{n}(x_i - y_i)c_i v_i/c_i \geq v_j/c_j \sum_{i=1}^{n}(x_i - y_i)c_i.$$

However $\sum_{i=1}^{n} y_i c_i \leq C = \sum_{i=1}^{n} x_i c_i$, implying that $\sum_{i=1}^{n}(x_i - y_i)v_i \geq 0$. Thus the greedy solution yields the maximum value.

In later chapters, we will look at other problems to which the greedy paradigm can be applied.

Matroid theory provides a fascinating exploration into a general framework for the applicability of the greedy paradigm to various problems. This, however, is material suitable for a more advanced course and, therefore, we will not discuss this in these lectures.

Chapter 5

Sorting

Sorting is one of the most well-studied problems in Computer Science. For an encyclopaedic reference on the subject see: "The Art of Computer Programming, Vol.3" by D.E. Knuth.

5.1 Comparison Sorts

A comparison sort orders a sequence of elements, drawn from a totally ordered set, by comparing elements pairwise. Formally, given a sequence of n elements

$$a_1, a_2, a_3, \ldots, a_{n-1}, a_n \tag{5.1}$$

find a permutation (or reordering) π of the numbers $1, 2, 3, \ldots, n$ such that:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq a_{\pi(3)} \leq \ldots \leq a_{\pi(n-1)} \leq a_{\pi(n)} \tag{5.2}$$

We will study the following comparison sorts:

- Insertion Sort
- Shellsort
- Mergesort
- Quicksort

5.1.1 Insertion Sort

We begin with a small example to illustrate the idea. To sort the sequence of integers

$$3, 1, 5, 4, 2 \tag{5.3}$$

we sort incrementally from left to right as follows:

Sort$\{3\} \rightarrow 3$
Sort$\{3,1\} \rightarrow 1,3$
Sort$\{1,3,5\} \rightarrow 1,3,5$
Sort$\{1,3,5,4\} \rightarrow 1,3,4,5$
Sort$\{1,3,4,5,2\} \rightarrow 1,2,3,4,5$

As one might guess, the idea is to keep the elements seen so far in sorted order. When a new element is input, we place it in the correct position to maintain the sorted order of the elements seen so far.

In general, at the ith step the first i elements

$$a_1, a_2, a_3, \ldots, a_{i-1}, a_i \tag{5.4}$$

are already in sorted order

$$a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \ldots, a_{\pi(i-1)}, a_{\pi(i)} \tag{5.5}$$

for some permutation π of $1, 2, 3, \ldots, i$. At the next step, a_{i+1} has to be inserted into the correct position with respect to the sorted order of the first i elements.

Algorithm 13 InsertionSort

Input: unsorted array $a[1..n]$, $n \geq 2$
Output: array $a[1..n]$, sorted in ascending order

```
1:  i ← 2
2:  while i ≤ n do
3:      temp ← a[i]                                   ▷ Save a[i]
4:      j ← i
5:      while (j > 1 && a[j − 1] > temp) do           ▷ Search for the correct location of a[i]
6:          a[j] ← a[j − 1]
7:          j ← j − 1
8:      end while
9:      a[j] ← temp
10:     i ← i + 1                                     ▷ next element
11: end while
12: return the array a[1 . . . n]
```

Complexity Analysis

The worst-case input for this sorting technique is when the input is in reverse sorted order. For example, if the input sequence is 5, 4, 3, 2, 1, the number of comparisons required to sort is 10, which is exactly equal to the total number of possible comparisons among a set of 5 elements.

For a sequence of n numbers $a_1, a_2, a_3, \ldots, a_n$ in reverse sorted order, this count is the total number of possible comparisons among a set of n elements, viz, $n(n-1)/2$. Thus the worst-case complexity of this sorting method is $O(n^2)$.

Exercise 5.1. Consider the following sorting scheme, known as selectionSort. In a linear scan of the list $a_1, a_2, a_3, \ldots, a_{n-1}, a_n$ from left- to-right determine the index of the smallest element; exchange it with the first element in the list; next scan left-to-right from the second element to find the smallest and exchange it the second element in the list. Carry on like this till the $(n-1)$-th element from the list. At this point the list will be sorted in ascending order. Write a formal algorithm similar to insertionSort above and analyze its worst-case running time.

5.1.2 ShellSort

ShellSort was invented by Donald Shell. The basic idea is to increase the "sortedness" of a given sequence in a well-defined set of steps.

We illustrate the idea by ShellSorting the following sequence

$$81, 94, 11, 96, 12, 35, 17, 95 \tag{5.6}$$

Step 1. InsertionSort all subsequences that are 4 apart

$$\begin{gathered} 81, 12 \rightarrow 12, 81 \\ 94, 35 \rightarrow 35, 94 \\ 11, 17 \rightarrow 11, 17 \\ 96, 95 \rightarrow 95, 96 \end{gathered}$$

This yields the new sequence:

$$12, 35, 11, 95, 81, 94, 17, 96 \tag{5.7}$$

Step 2. InsertionSort all subsequences of the last sequence that are 2 apart

$$\begin{gathered} 12, 11, 81, 17 \rightarrow 11, 12, 17, 81 \\ 35, 95, 94, 96 \rightarrow 35, 94, 95, 96 \end{gathered}$$

This yields the new sequence:

$$11, 35, 12, 94, 17, 95, 81, 96 \tag{5.8}$$

Step 3. InsertionSort all subsequences of the last sequence that are 1 apart

$$11, 35, 12, 94, 17, 95, 81, 96 \rightarrow 11, 12, 17, 35, 81, 94, 95, 96$$

This yields the new sequence:

$$11, 12, 17, 35, 81, 94, 95, 96 \tag{5.9}$$

The last sequence is now in sorted order.

Observation 4, 2, 1 is called a gap-sequence. Different gap sequences are possible. Every one of them must end with a 1.

Since in the last step we do insertionSort on the entire sequence what is the advantage of this method vis-a-vis plain insertionSort on the initial sequence. A rigorous proof that this scores over plain insertionSort is complex. We provide an intuitive justification by comparing the performances of insertionSort on the sequence output by Step 2 and on the initial sequence.

InsertionSort on :

81, 94, 11, 96, 12, 35, 17, 95

Sequence	Comparisons
81	0
81, 94	1
11, 81, 94	2
11, 81, 94, 96	1
11, 12, 81, 94, 96	4
11, 12, 35, 81, 94, 96	4
11, 12, 17, 35, 81, 94, 96	5
11, 12, 17, 35, 81, 94, 95, 96	2

The total number of comparisons required to sort the initial sequence is thus 19.

InsertionSort on :

11, 35, 12, 94, 17, 95, 81, 96

Sequence	Comparisons
11	0
11, 35	1
11, 12, 35	2
11, 12, 35, 94	1
11, 12, 17, 35, 94	3
11, 12, 17, 35, 94, 95	1
11, 12, 17, 35, 81, 94, 95	3
11, 12, 17, 35, 81, 94, 95, 96	1

The total number of comparisons required to insertionSort the partially sorted sequence is thus 12.

It can be shown that Hibbard's gap sequence, namely 1,3,7, ..., 2^k - 1, leads to a subquadratic algorithm, where k is roughly $\log\lceil n/2 \rceil$. Precisely, $T(n) = O(n^{3/2})$ in this case.

5.1.3 Mergesort

Mergesort is based on the divide and conquer strategy. We divide an input sequence into two halves, mergeSort each of the two halves recursively and then, in a conquest step, merge the two sorted subsequences into a final sorted sequence.

We illustrate the ideas on the example sequence of the last section.

To mergeSort the sequence 81, 94, 11, 96, 12, 35, 17, 95:

Step 1. MergeSort the subsequence 81, 94, 11, 96 to obtain 11, 81, 94, 96

Step 2. MergeSort the subsequence 12, 35, 17, 95 to obtain 12, 17, 35, 95

Step 3. Merge the sorted subsequences from Steps 1 and 2 into the sorted sequence 11, 12, 17, 35, 81, 94, 95, 96

The details of the last step are as follows. We have a moving pointer for each sorted subsequence, initially pointing to 11 and 12 respectively. As 11 is smaller we output 11, and move the pointer in this list to the next element. Next, we output 12 from the second list and move the pointer in this list to the next element. We continue this until one of the lists is exhausted, when we move the elements remaining in the second list to the output.

A formal description of this merge procedure is shown below, where the lists to be merged are subarrays of the array to be sorted:

For analyzing the complexity of the algorithm, we need the following theorem.

Theorem 5.1. To merge two sorted lists, each of length n, we need at most $2n - 1$ comparisons.

Algorithm 14 The mergeSort algorithm

```
1: procedure mergeSort(a[1..n])
2:     if n > 1 then
3:         return mergeLists(mergeSort(a[1..⌊(n/2)⌋]), mergeSort(a[⌊(n/2)⌋ + 1..n]))
4:     else
5:         return a
6:     end if
7: end procedure
```

Algorithm 15 Merging two sorted lists into one sorted list iteratively

Ensure: $p \leq q < r$

```
1: procedure mergeLists(a[p, q], a[q + 1..r])      ▷ merged sorted list written into global workspace
   array b[1..n]
2:     i ← p;
3:     j ← q + 1;
4:     k ← 1;
5:     while ((i <= q)&&(j <= r)) do
6:         if (a[i] > a[j]) then
7:             b[k] = a[j];
8:             j ← j + 1;
9:             k ← k + 1;
10:        else
11:            b[k] = a[i]
12:            i ← i + 1
13:            k ← k + 1
14:        end if
15:    end while
16:    while (i <= q) do
17:        b[k] = a[i];
18:        i ← i + 1;
19:        k ← k + 1;
20:    end while
21:    while (j <= r) do
22:        b[k] = a[j];
23:        j ← j + 1;
24:        k ← k + 1;
25:    end while
26:    i = p;
27:    for (k = 1; k <= r − p + 1; k + +) do
28:        a[i] = b[k];
29:        i ← i + 1;
30:    end for
31: end procedure
```

Proof: Each comparison increases the length of the merged list by 1. Now one of the two lists must get exhausted first. When this happens, the length of the merged list is $n + n - s$, equal to the number of comparisons made so far, where s is the number of elements that are left in the second list that are simply to be added to the merged list without any comparisons. This is maximum

when $s = 1$. Thus at most $2n - 1$ comparisons are needed.

Complexity of mergeSort

If $T(n)$ is the time complexity of mergeSorting a list of n elements, then

$$T(n) = 2T(n/2) + cn \text{ when } n > 1 \quad (5.10)$$
$$= c' \text{ when } n = 1 \quad (5.11)$$

From our discussion in Chapter 4, we know that the solution of this recurrence is $T(n) = O(n \log n)$

5.1.4 Quicksort

We first study a partitioning problem. Given the list

$$L = 5, 3, 2, 6, 4, 1, 3, 7, \quad (5.12)$$

partition L into two sublists L_1 and L_2 such that every element in L_1 is less than or equal to every element in L_2.

Certainly, we can achieve our goal if we make every element of $L_1 \leq$ the first element of L and every element of $L_2 \geq$ the first element of L. There are two points to be noted. One is that we don't care if an element equal to the first element (including itself) is put into list L_1 or in list L_2. Second is that we could have chosen any other element of L instead of the first one to do this partitioning. We read the list L into an array, and partition in place (this means without using any additional space), using two pointers - a left pointer (lp) and a right pointer (rp).

Starting with the initial configuration of Fig. 5.1,

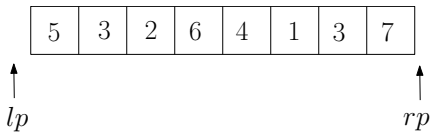


Figure 5.1: Initial configuration for the partition

In a loop, we move the pointers according to the following rules:

- Move lp to the right until we encounter an element ≥ 5 (the first element of L).
- Move rp to the left until we encounter an element ≤ 5 (the first element of L).
- If the pointers have not met or crossed ($lp < rp$), we exchange the elements pointed at by the pointers.
- If the pointers meet or cross ($lp \geq rp$), we exit the loop

The loop-invariant is that all elements to the left of and including the left pointer are less than or equal to all the elements to the right of and including the right pointer. On exit from this loop, we return rp to indicate that L_1 goes from 1 to rp, while L_2 goes from $rp + 1$ to n. For the example list, we have the following sublists:

$$L_1 = 3, 3, 2, 1, 4 \tag{5.13}$$
$$L_2 = 6, 5, 7 \tag{5.14}$$

The following figures show some of the intermediate steps in the partitioning scheme.

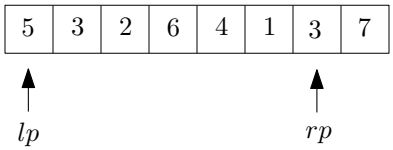


Figure 5.2: First stop

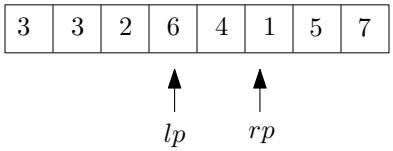


Figure 5.3: Second stop

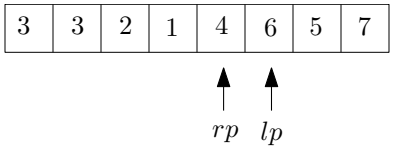


Figure 5.4: The pointers have crossed

Now we do the same with the lists L_1 and L_2 recursively. Partitioning stops if we have a list with only one element. All this done in place gives us the following sorted list:

$$L_{sorted} = 1, 2, 3, 3, 4, 5, 6, 7 \tag{5.15}$$

This is Quicksort!! We describe the partitioning procedure formally below.

Algorithm 16 The partition algorithm

```
1:  procedure Partition(L, p, q)                    ▷ Partitions the input list L[p..q]
2:      a ← L[p]                                    ▷ Initializations
3:      lp ← p − 1
4:      rp ← q + 1
5:      while (lp < rp) do
6:          repeat                                  ▷ Move left pointer right
7:              lp ← lp + 1
8:          until (L[lp] ≥ a)
9:          repeat                                  ▷ Move right pointer left
10:             rp ← rp − 1
11:         until (L[rp] ≤ a)
12:         if (lp < rp) then
13:             exchange(L[lp], L[rp])
14:         end if
15:     end while
16:     return rp
17: end procedure
```

We can now describe Quicksort in terms of the key ingredient PARTITION. To sort a list L, the initial call to Quicksort is Quicksort(L, 1, length(L)).

Algorithm 17 The Quicksort Algorithm

```
1: procedure quickSort(L, p, q)                     ▷ Outputs sorted list L[p, q]
2:     if (p < q) then
3:         r ← Partition(L, p, q)
4:         Quicksort(L, p, r)
5:         Quicksort(L, r + 1, q)
6:     else
7:         return
8:     end if
9: end procedure
```

Discussion

At each level of recursion we would like to split the lists as evenly as possible. This makes the choice of a partitioning element very important. Various heuristics have been proposed for this, a popular one being the median-of-three heuristic where we choose the median of three randomly chosen elements or the median of the first, last and middle elements of a list. A second heuristic is to choose the partitioning element at random.

Clearly, if we always manage to split a list into two equal halves the complexity of the worst case running time $T(n)$ would be a solution to the recurrence.

$$T(n) \quad = \quad 2T(n/2) + c * n \tag{5.16}$$

which is $O(n \log n)$.

On the other hand, if we always split a list into two that are of length 1 and length(L) -1, the worst case running time $T(n)$ would be a solution to the recurrence

$$T(n) \quad = \quad T(1) + T(n-1) + c * n \tag{5.17}$$

which is $O(n^2)$.

The actual behaviour of Quicksort is an average of these two extremes. In fact, we can prove that in a well-defined average sense the running time of Quicksort is $O(n \log n)$.

Problem 5.1. Let L be a list of n numbers. An element in this list has rank r, if it appears in the rth position when L is sorted in ascending order. Given r, devise an algorithm that finds an element whose rank is r in $O(n)$ time.

Problem 5.2. Design an $O(n \log n)$ algorithm that, given a set S of n real numbers and another number x, determines whether or not there exist two elements in S whose sum is x.

5.1.5 Sorting - in Linear time ?

It is an interesting question to ask if we can sort a list of elements in linear time ? The answer is YES, under some assumptions on the input. Among the various linear time sorting techniques, we have:

- Counting sort
- Bucket sort
- Radix sort (also called lexicographic sort)

Here, we will discuss Counting sort. The main assumption is that the input consists of integers in the range 1 to k. As an example, consider the following example.

Example 5.1. CountSort the list of integers:

$$L \quad = \quad 3, 6, 4, 1, 3, 4, 1, 4$$

The input list L is read into an array $A[1..8]$. Two auxiliary arrays $C[1..6]$ and $B[1..8]$ are used. On a first scan of the input array A from left to right, we record in the array element $C[i]$ the number of times the element i occurs. Next, we scan through the array $C[1..6]$ a second time, updating each entry $C[i]$ to $C[i] + C[i-1]$ so that each array element now records the number of elements of A that are less than or equal to i (see Fig. 5.6).

$A[1..8]$	3	6	4	1	3	4	1	4

$C[1..6]$	2	2	4	7	7	8

Figure 5.5: The A and C arrays, upon initialization

We now scan the array A from right to left and for each entry $A[j]$ access the element $C[A[j]]$ of $C[1..6]$ to determine the position of $A[j]$ in the array $B[1..8]$. We write the element $A[j]$ into this array position in $B[1..8]$ and decrement $C[A[j]]$ by 1 (see Fig. 5.6). A formal description of countingSort is shown in Algorithm 18.

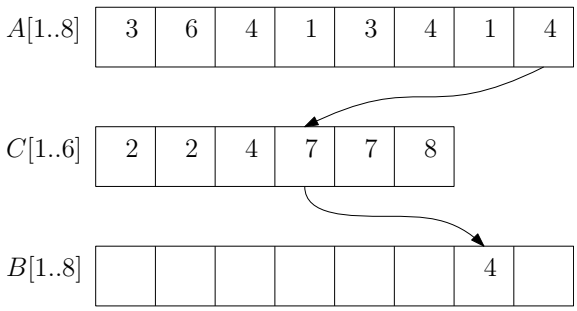


Figure 5.6: The first step of filling array B[1..8]

Algorithm 18 CountSort

Input: Arrays $A[1..n]$, $C[1..k]$ ▷ $1 \leq A[i] \leq k$ for each i
Output: Sorted array $B[1..n]$

1: **for** $i \leftarrow 1, k$ **do**
2: $C[i] \leftarrow 0$
3: **end for**
4: **for** $j \leftarrow 1, n$ **do**
5: $C[A[j]] \leftarrow C[A[j]] + 1$
6: **end for**
7: **for** $i \leftarrow 2, k$ **do**
8: $C[i] \leftarrow C[i] + C[i-1]$
9: **end for**
10: **for** $j \leftarrow n, 1$ **do**
11: $B[C[A[j]]] \leftarrow A[j]$
12: $C[A[j]] \leftarrow C[A[j]] - 1$
13: **end for**

Discussion

- Complexity: If $T(n,k)$ is the time complexity of countingSort from the separate loops on n (list size) and k (range of the input) in the above algorithm it is clear that $T(n,k) = O(n+k)$. Now if $k = O(n)$ then $T(n,k) = O(n)$, giving us a linear-time sort.
- Stability: A sorting method is said to be stable if equal elements are output in the same order as they are encountered in the input. In the sense of this definition countingSort is stable.

Lower bounds on sorting

Theorem 5.2. For any comparison sort on n elements $T(n) = \Omega(n \log n)$

Remark

$T(n) = \Omega((f(n))$ means that $T(n)$ grows at least as fast as $f(n)$, or, what is the same thing, that $f(n)$ grows at most as fast as $T(n)$, which is the same as saying that $f(n) = O(T(n))$.

Exercise 5.2. Simulate countingSort on the same input $L = 3, 6, 4, 1, 3, 4, 1, 4$ with one change to line 10 of Algorithm 18: $for\ j \leftarrow 1, n$. This means the second scan of the array A is done from right to left. Does it produce a sorted output as before. What is the difference ?

Chapter 6

Non-linear lists or Trees

6.1 Introduction

You must be wondering why we have invested so much energy into sorting. The reason is that it allows us to search efficiently. Imagine that you are given an unsorted list of n numbers (say) and you were querying this list for the presence of a given number. You would necessarily have to do a linear search and this can take $O(n)$ time in the worst-case. Consider on the other hand that this list was sorted. In this case we can do a binary search, and solve this decision problem in $O(\log n)$ time.

If our sorted list is static, we would be quite satisfied with a linear sorted list. However if our list changes over time due to insertions and deletions, the reorganization of our list after each insertion and deletion can take $O(n)$ time in the worst-case.

The leading question then is how we should organize our list if we want to search, insert into and delete from this list efficiently, let's say in $O(\log n)$ time, where n is the size of our list.

This leads us to consider non-linear lists, namely trees. Trees are an ubiquitous data structure in Computer Science. The UNIX directory structure, for example, is a tree. A tree, T, is best described recursively as follows. It is a non-empty set, one element of which is designated as the root of the tree, while the remaining elements are partitioned into non-empty sets $T_1, T_2, \ldots, T_k$ each of which is a subtree of the tree T. This relationship is shown in Fig 6.1 below. In Computer Science the elements are called nodes of the tree.

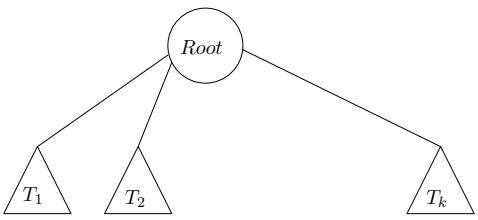


Figure 6.1: Recursive structure of a tree

A tree is usually drawn as shown in Fig. 6.2.

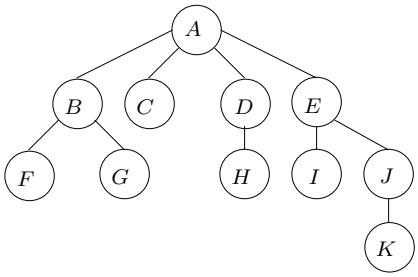


Figure 6.2: Example of a tree

6.1.1 Terminology of trees

Some of the common terminology associated with trees are introduced with reference to the example of Fig 6.2.

A is known as the $root$ of the tree; B, C, D, E are called the $children$ of (the root) A, while being $siblings$ of each other. E is an $ancestor$ of K, while K is a $descendent$ of E. F, G, C, H, I, K are called the $leaves$ of the tree.

The $depth$ of a node is the length of the path from the root to that node, the length of a path being the number of edges that make up the path. Thus the depth of the node J is 2.

The $height$ of a node is the length of the longest path from that node to a leaf of the subtree, rooted at that node. The height of a tree is the height of the root. Thus the height of the tree in Fig. 6.2 is 3.

6.1.2 Binary Trees

Definition 6.1. A binary tree T is a finite set of nodes that is either

- empty (denoted by $\emptyset$), or
- consists of a distinguished node, called the root of the binary tree, while the remaining nodes are divided into two subsets, a left binary subtree of the root and a right binary subtree of the root.

The root of the left binary subtree, when not empty, is called the left child of the root, while the root of the right binary subtree, when not empty, is called the right child of the root.

The degree of a node is the number of children it has. In the example binary tree of Fig 6.3 the nodes a, b, c all have degree 2; the leaves d, e, f, g are of degree 0.

Problem 6.1. Show by induction that the number of degree 2 nodes in any binary tree is 1 less than the number of leaves.

Proof. The proof is by induction on the number of nodes n in the binary tree. For $n = 0$, the claim is vacuously true in the sense that we can't produce a binary tree on $n = 0$ nodes for which the claim is false.

Suppose the claim is true for all binary trees with $\leq n$ nodes. Consider a binary tree on $n+1$ nodes. It has one of the three structures, shown in Fig. 6.4.

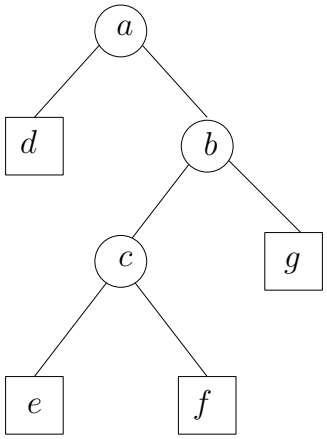


Figure 6.3: A binary tree

Let n_L and n_R be the number of nodes in the left (binary) subtree, T_L, and the right binary subtree, T_R, respectively. Since $n_L \leq n$ and $n_R \leq n$ the claim is true for the left subtree in case (i) and the right subtree in case (ii). Since in both the cases the degree 2 nodes and the leaves belong to these subtrees alone, the claim is true for the original binary tree.

In case (iii), the claim is true for both subtrees; this means

leaves in T_L = # degree 2 nodes in T_L + 1
leaves in T_R = # degree 2 nodes in T_R + 1

Adding up the two equations, we get:

leaves in T_L + # leaves in T_R = # degree 2 nodes in T_L + 1 + # degree 2 nodes in T_R + 1

This proves our claim since the left-hand side counts the total number of leaf nodes in the binary tree, T, while the sum of the first three terms on the right-hand side gives the total number of degree two nodes in T, the 1 counting for the root node of T.
Given a binary tree, we often need to

- Find its height
- Find its size or
- Visit its nodes in a systematic way

The height of a binary tree is recursively defined as follows (see Fig. 6.5).

$$Height(\emptyset) = -1 \quad (6.1)$$
$$Height(T) = 1 + max(Height(T_L), Height(T_R)) \quad (6.2)$$

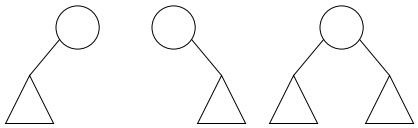


Figure 6.4: Three different structures

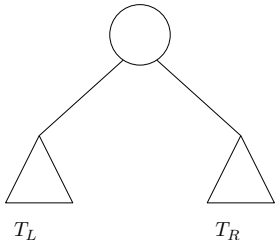


Figure 6.5: Height/Size of a binary tree

The size of a binary tree is recursively defined as follows (see Fig. 6.5).

$$Size(\emptyset) = 0 \quad (6.3)$$
$$Size(T) = (1 + Size(T_L) + Size(T_R)) \quad (6.4)$$

Using the above definitions, it is easy to write recursive programs to compute these quantities.

Exercise 6.1. Show by induction that a binary tree with n nodes has height at least $\lfloor \log n \rfloor$.

6.1.3 Tree traversals

Many problems require that we visit the nodes of a binary tree structure in a systematic way to perform a variety of tasks. We will discuss the following techniques.

- Inorder traversal
- Preorder traversal
- Postorder traversal

Preorder Traversal

1. Visit the root

2. Visit the left subtree recursively in preorder

3. Visit the right subtree recursively in preorder

For the binary tree of Fig. 6.3, a preorder visit to print node labels will produce the following output:

$$a, d, b, c, e, f, g$$

Postorder Traversal

1. Visit the left subtree recursively in postorder

2. Visit the right subtree recursively in postorder

3. Visit the root

For the binary tree of Fig. 6.3, a postorder visit to print node labels will produce the following output:

$$d, e, f, c, g, b, a$$

Inorder Traversal

1. Visit the left subtree recursively in inorder

2. Visit the root

3. Visit the right subtree recursively in inorder

For the binary tree of Fig. 6.3, an inorder visit to print node labels will produce the following output:

$$d, a, e, c, f, b, g$$

6.1.4 Heapsort

With the data structures developed so far, the binary tree in particular, we are in a position to discuss a new sorting algorithm called heapSort. To begin with, we need the following definition.

Definition 6.2. An (essentially) complete binary tree is a binary tree that is complete at all levels except possibly the last.

Some examples of such trees are shown in Fig. 6.6.

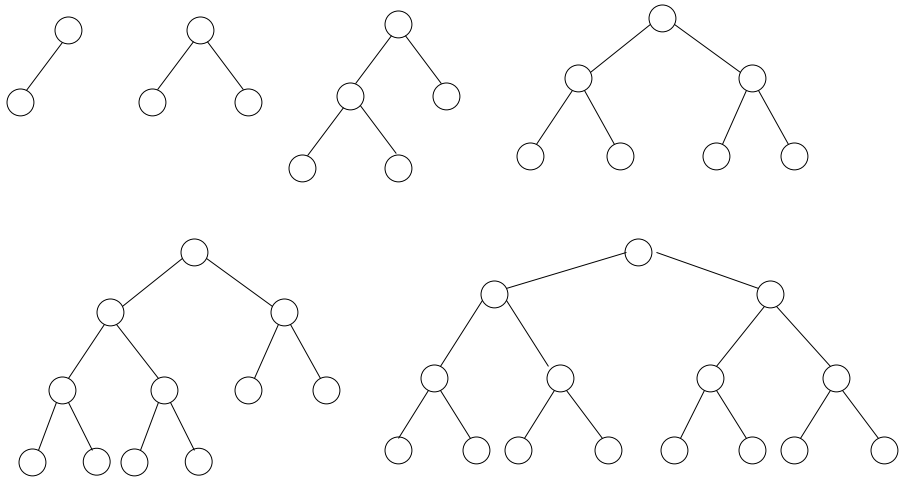


Figure 6.6: Some essentially complete binary trees

A heap is an essentially complete binary tree with the following additional property:

```
The key stored at any node is greater than or equal to (≥) the keys
stored at its children, if any.
```

Note 6.1. In the above definition we can replace ($\geq$) by ($\leq$).

Some examples of heaps are shown in Fig. 6.7.

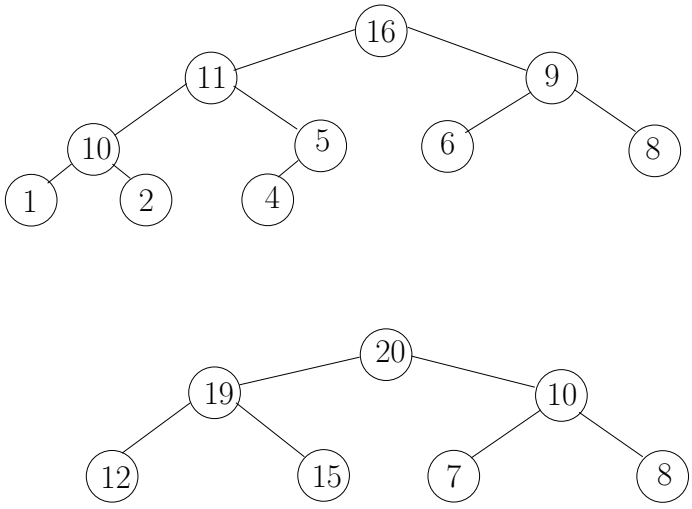


Figure 6.7: Examples of heaps

Each one of the heaps in the examples above can be represented by means of an array. The heap of Fig. 6.7(a) by the array of Fig. 6.8(a), while the heap of Fig. 6.7(b) can be represented by the array of Fig. 6.8(b). The rule for going from the binary tree representation to the array representation is simple. We write down the key values level by level from the top to the bottom and from left to right at each level. As a result, the children of the key at index position $i(\leq \lfloor n/2 \rfloor)$ is to be found at the index positions $2i$ and $2i+1$ (at $2i+1$ and $2i+2$ if we index the array from 0 to $n-1$).

Now an important question is this: Given an array $A[1..n]$ of key values how do we heapify this array, that is, turn this into a heap.

Left-to-Right Heapification

Let us show how this can be done by first scrambling the array entries, corresponding to the heap in Fig. 6.8(a). We turn it into a heap again, incrementally, looking at the elements one by one from left to right as in Fig. 6.9, letting each new element float to its correct level in the heap, constructed so far.

The process is visually more apparent if we construct an essentially complete binary tree, filling in the nodes with key values level by level, going left to right in the array. Then, with respect to this essentially complete binary tree, heapification as described above proceeds level by level from top

16	11	9	10	5	6	8	1	2	4

(a)

20	19	10	12	15	7	8

(b)

Figure 6.8: Array representations of heaps

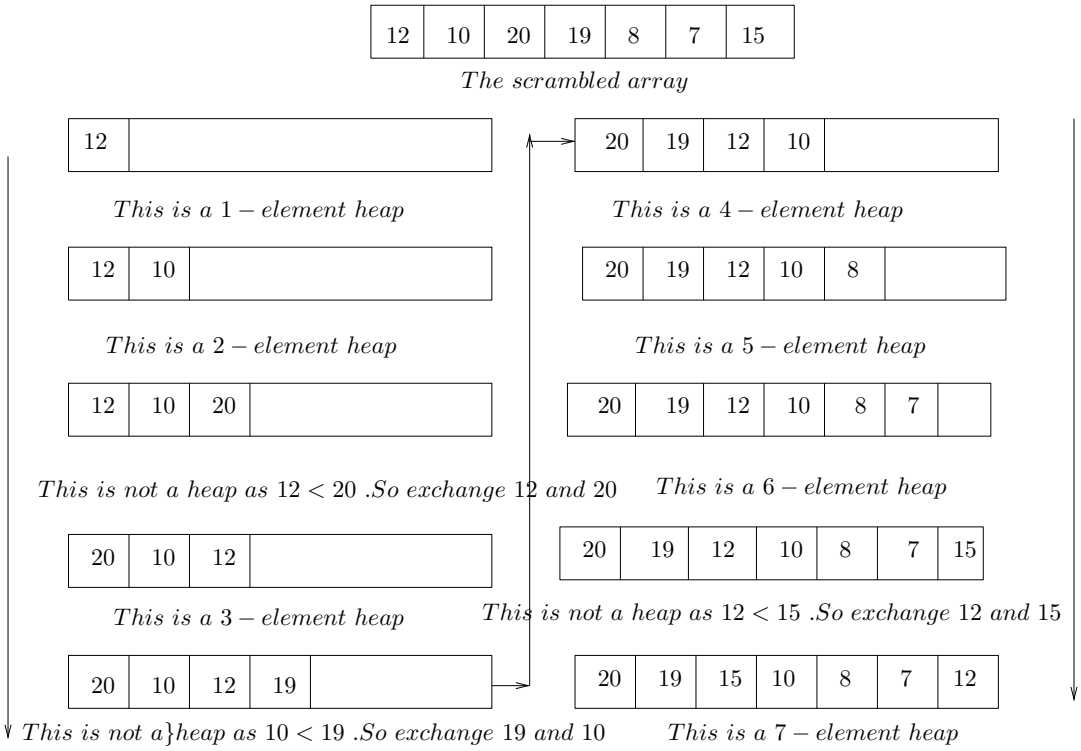


Figure 6.9: The heapification process

to bottom, and from left to right at each level. This is described in Fig. 6.10.

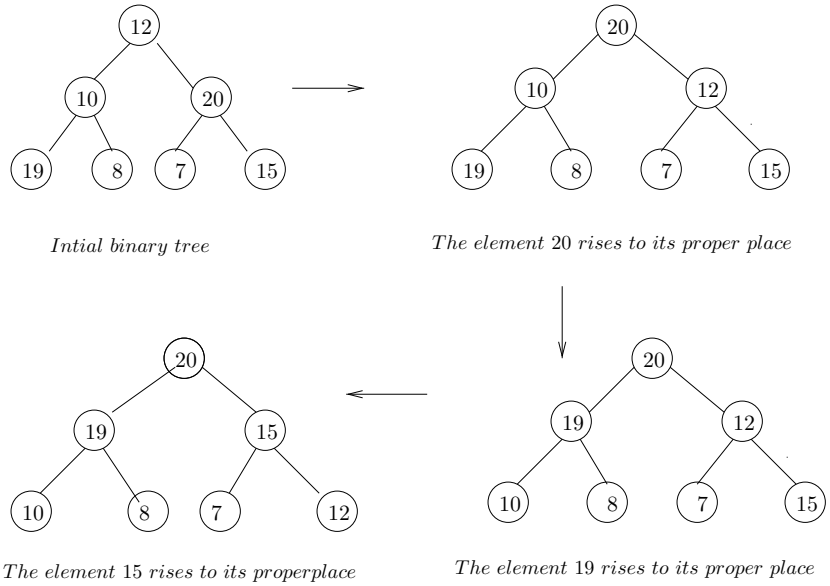


Figure 6.10: The heapification process

Right-to-Left Heapification

Heapification can also be done by going from right to left in the array. In terms of the essentially complete binary tree, this corresponds to going level by level from the bottom to the top, and from right to left at each level. This is shown in Fig. 6.11.

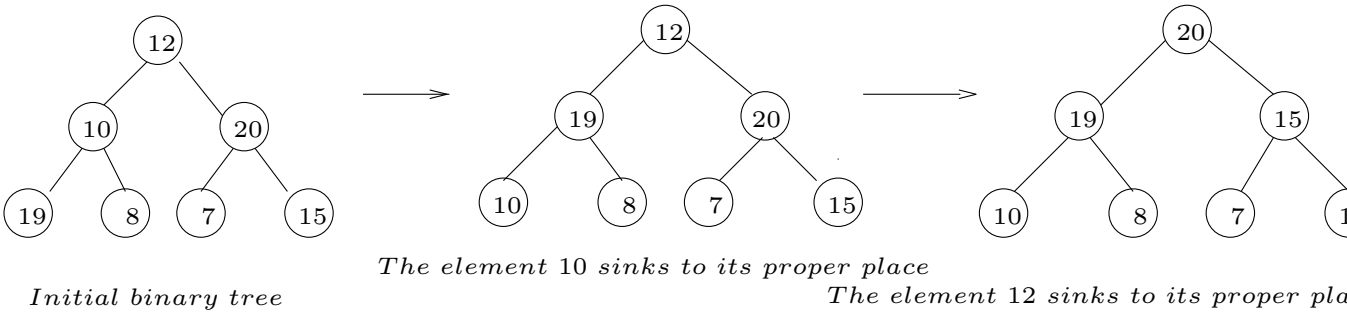


Figure 6.11: The heapification process

Algorithmically, we can describe the right-to-left heapification in terms of a subroutine called $Heapify(i, j)$, which heapifies the part of the array going from index i to index j.

Algorithm 19 The Heapify Algorithm

```
1:  procedure Heapify(i, j)
2:      if (j < 2i) then
3:          return
4:      end if
5:      if (j == 2i) then
6:          k = 2i
7:      else if (j > 2i) then
8:          if (A[2i] ≥ A[2i + 1]) then
9:              k = 2i
10:         else
11:             k = 2i + 1
12:         end if
13:     end if
14:     if (A[k] > A[i]) then
15:         exchange(A[i], A[k])
16:         Heapify(k, j)
17:     end if
18: end procedure
```

It is now easy to describe how to build a heap, using the procedure $Heapify(i, j)$.

Algorithm 20 Buildheap

Input: An array $A[1..n]$ of key-values
Output: An array $A[1..n]$ that satisfies the heap property

1: $i \leftarrow n$
2: **while** ($i \geq 1$) **do**
3: Heapify(i, n).
4: $i \leftarrow i - 1$
5: **end while**
6: **return**

Finally, we have all the ingredients to describe heapSort.

Algorithm 21 HeapSort

Input: An array $A[1..n]$ of key-values
Output: A sorted array $A[1..n]$ of key-values

1: Call BuildHeap
2: $i \leftarrow n$
3: **while** ($i > 1$) **do**
4: Exchange $A[1]$ with $A[i]$.
5: Heapify$(1, i - 1)$.
6: $i \leftarrow i - 1$
7: **end while**
8: **return**

Fig. 6.12 below shows how heapsort works on the array of Fig.6.9

Analysis of Heapsort

Theorem 6.1. The complexity of BUILDHEAP is in $O(n)$.

Proof: In a heap, represented as an essentially complete binary tree, at most $n/2^{i+1}$ (strictly speaking $\lfloor n/2^{i+1} \rfloor$) nodes have height i, where i lies between 0 and $\lfloor \log n \rfloor$.

Thus the number of exchanges required to restore the heap property for every node at height i is at most $i * \lceil n/2^{i+1} \rceil$. Thus the total number of exchanges require to heapify the entire tree is at most $S = \sum_{i=0}^{\lfloor \log n \rfloor} i * \lceil n/2^{i+1} \rceil$.

Now $S \leq \sum_{i=0}^{\lfloor \log n \rfloor} i * n/2^i = \sum_{i=1}^{\log n} i * n/2^i$
.
We show that $\sum_{i=1}^{\log n} i * n/2^i = O(n)$ or that $\sum_{i=1}^{\log n} i/2^i = O(1)$

Indeed, $\sum_{i=1}^{\log n} i/2^i < 2$. Let $x = 1/2$ and $\log n = h$. Then

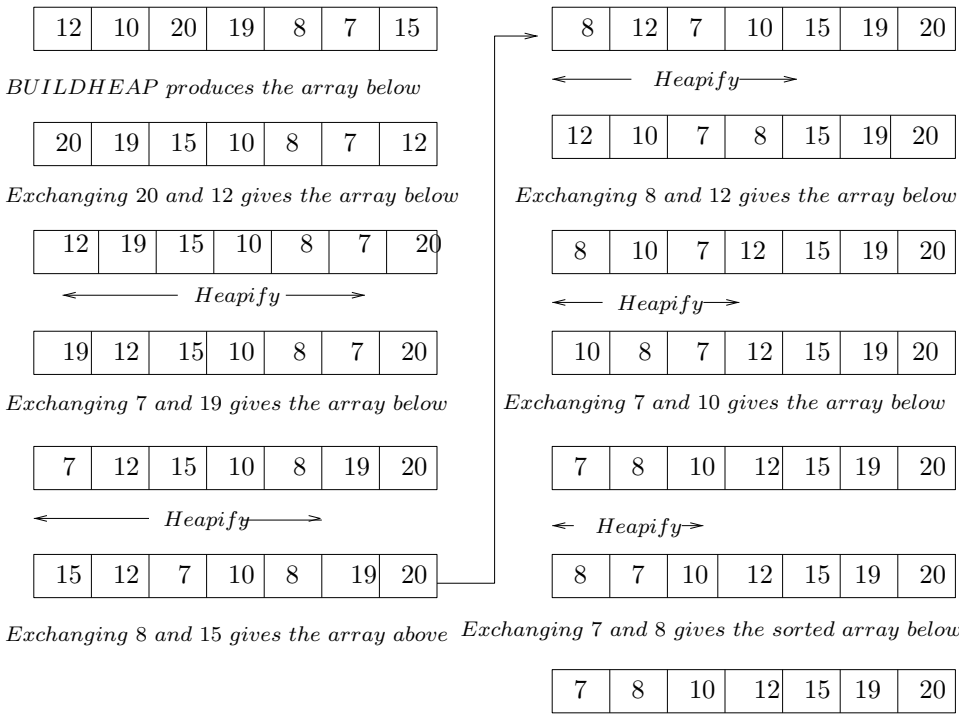


Figure 6.12: Heapsort on an example

$$\begin{aligned}
\sum_{i=1}^{h} i/2^i &= \sum_{i=1}^{h} i * x^i \\
&= \sum_{i=1}^{h} x * \frac{d}{dx} x^i \\
&< x * \frac{d}{dx} \sum_{i=0}^{\infty} x^i \\
&= x * \frac{d}{dx} \{(1/(1-x)\} \\
&= x/(1-x)^2 \\
&= 2
\end{aligned}$$

Hence the claim. □

The complexity of $heapSort$ is $O(n \log n)$. A rough argument goes as follows. It can be shown that the time complexity of $buildHeap$ is $O(n)$. We call $heapify$ n times, the time complexity of each heapification being in $O(\log n)$.

6.1.5 Binary Search Trees

A Binary Search Tree (BST, for short) is a binary tree; each node has at least the following four fields as shown in Fig. 6.13.

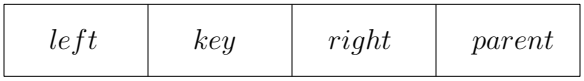


Figure 6.13: Node structure of a binary search tree

The key fields in a BST satisfy the following BST property:

Let x be a node in a BST. If y is a node in the left subtree of x then

$$x.key > y.key \tag{6.5}$$

If y is a node in the right subtree of x then

$$x.key < y.key \tag{6.6}$$

An example of a binary search tree is shown in Fig. 6.14.

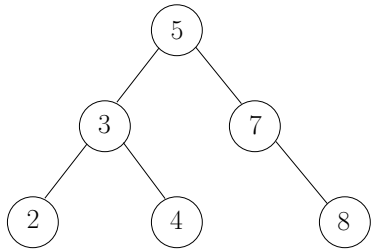


Figure 6.14: A binary search tree

Because of the BST property, an inorder traversal of the tree allows us to print the key values in ascending order. For the example binary tree this is: 2, 3, 4, 5, 7, 8.

Querying a BST

- Searching for a node with a given key value, k

  The following procedure returns a pointer to a node containing the key value k or NULL if there is no such node.

Algorithm 22 Searching a tree for a key

```
1:  procedure TreeSearch(x, k)
2:      if (x== NULL OR k == x.key) then
3:          return x
4:      end if
5:      if (k < x.key) then
6:          return TreeSearch(x.left, k)
7:      else
8:          return TreeSearch(x.right, k)
9:      end if
10: end procedure
```

- Minimum and Maximum keys

  The following procedures returns a pointer to a node containing the maximum and minimum key value respectively.

Algorithm 23 Searching a tree for a minimum key

```
1: procedure TreeMin(x)
2:     while (x.left != NULL) do
3:         x ← x.left
4:     end while
5:     return x
6: end procedure
```

Algorithm 24 Searching a tree for a maximum key

```
1: procedure TreeMax(x)
2:     while (x.right != NULL) do
3:         x ← x.right
4:     end while
5:     return x
6: end procedure
```

- Successor and Predecessor of a node

  Definition 6.3. The successor of a node x, if it exists, is the node with the smallest key value greater than $x.key$

  The following procedures returns a pointer to a node containing the successor of x, when it exists.

Algorithm 25 Finding the successor of a key

```
1: procedure TreeSuccessor(x)
2:     if (x.right != NULL) then
3:         return TreeMin(x.right)
4:     else
5:         y ← x.p
6:         while ((y!= NULL && x == y.right) do
7:             x ← y
8:             y ← y.p
9:         end while
10:    end if
11:    return x
12: end procedure
```

In Fig. 6.15 the arrows trace the search path for the successor of the node with key value 13.

Exercise 6.2. Discuss how the predecessor of a node can be determined.

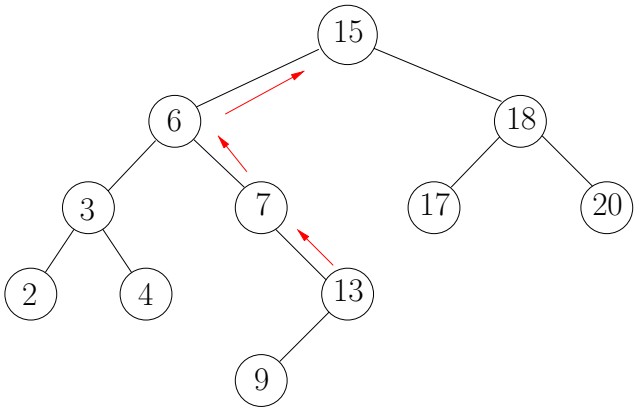


Figure 6.15: Successor of a node in a binary search tree

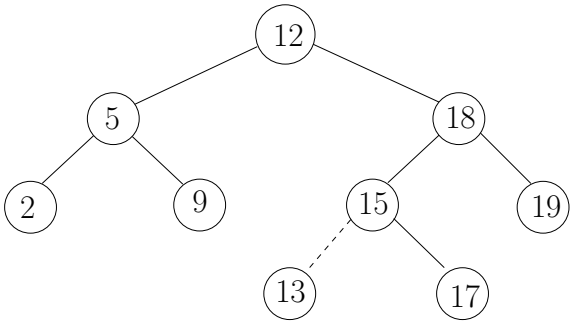


Figure 6.16: Inserting a key into a non-empty BST

Algorithm 26 Inserting a key into a tree

```
1:  procedure TreeInsert(x, y, k)
2:      if (x == NULL) then
3:          create new node with key k
4:          set parent pointer of the new node to y
5:      else if (k < x.key) then
6:          TreeInsert(x.left, x, k)
7:      else if (k > x.key) then
8:          TreeInsert(x.right, x, k)
9:      else
10:         return "key is in BST"
11:     end if
12: end procedure
```

- Insertion and Deletion

  In Fig. 6.16 we show how to insert a node with key-value 13.

  We discuss deletion with the help of several examples below. Fig. 6.17 illustrates how to delete a node with no children, Figs. 6.18a and 6.18b illustrate how to delete a node with one child, and Figs. 6.19a and 6.19b illustrate how to delete a node with two children.

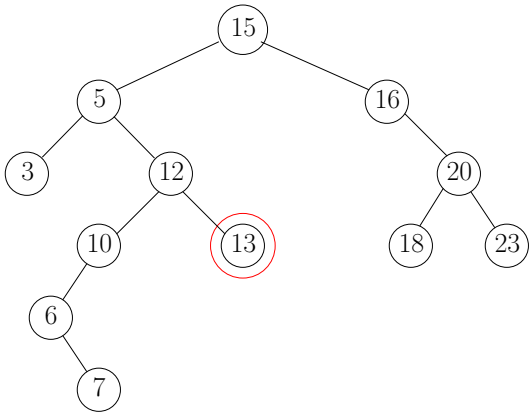
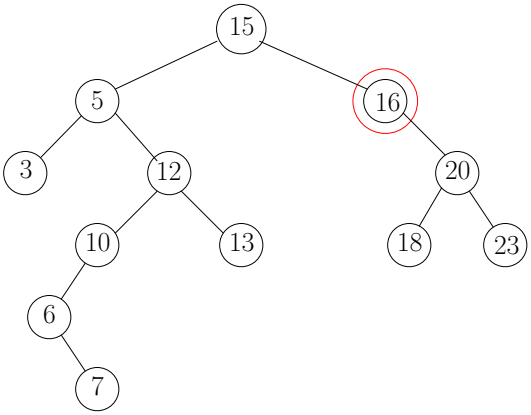
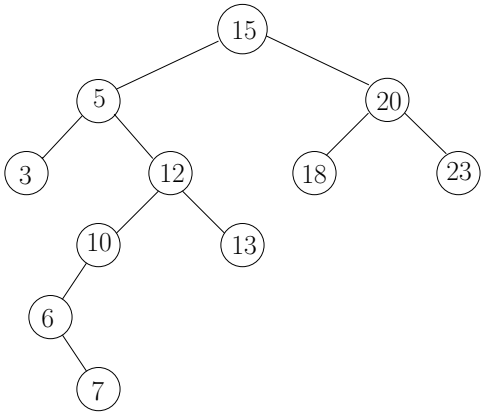


Figure 6.17: Deleting a node with no children

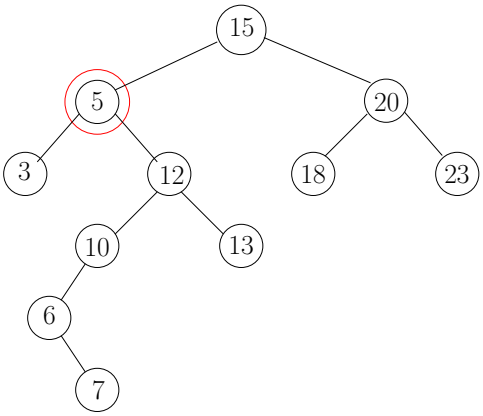


(a) Deleting a node with one child (before)

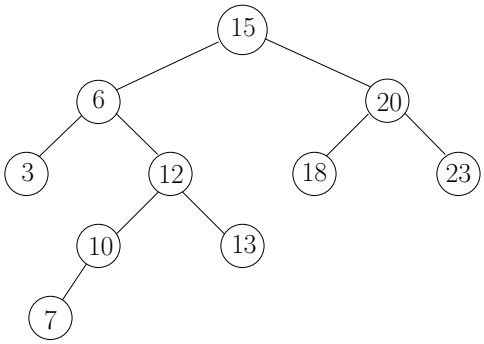


(b) Deleting a node with one child (after)

Figure 6.18: Deletions from a BST



(a) Deleting a node with two children (before)



(b) Deleting a node with two children (after)

Figure 6.19: Deletions from a BST

6.1.6 Balanced Search Trees

In this section, we discuss height-balanced search trees.

Definition 6.4. Binary Search Trees with a worst-case height of $O(\log n)$ are called balanced trees.

For a balanced tree, we can guarantee $O(\log n)$ performance for each search tree operation. We will discuss two types of balanced trees.

- AVL trees
- Red-black trees

AVL Trees

AVL trees are named after their Russian discoverers - Adelson-Velskii and Landis.

Definition 6.5. A empty binary tree is an AVL tree. If T is a non-empty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff (i) T_L and T_R are AVL trees and (ii) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

Definition 6.6. An AVL search tree is a binary search tree that is also an AVL tree

Henceforth, by an AVL tree we shall mean an AVL search tree. A node in an AVL tree has an additional entry to record the balance factor of the binary subtree rooted at that node as shown in Fig. 6.20 below.

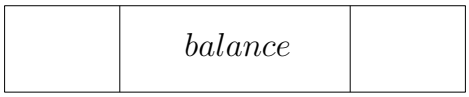
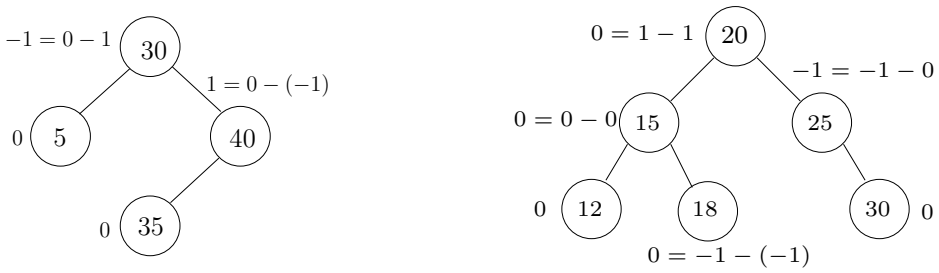


Figure 6.20: A node in a balanced tree

where, $balance$ = height of left subtree of the node - height of right subtree of the node

Examples of AVL search trees



(a) An AVL tree, (b) Another AVL tree

Figure 6.21: Labels outside a node indicate balance factor

Height of an AVL tree

Let N_h be the minimum number of nodes in an AVL tree of height h. Then

$$N_h = N_{h-1} + N_{h-2} + 1 \tag{6.7}$$

with $N_0 = 1$ and $N_1 = 2$, assuming the worst-case scenario in which one subtree has height $h-1$ and the other has height $h-2$.

The solution of the recurrence in (6.7) is:

$$N_h = F_{h+3} - 1 \tag{6.8}$$

where F_n is the nth Fibonacci number. Thus if n be the number of nodes in an AVL tree of height h then

$$\begin{aligned} n &>= N_h && (6.9)\\ &= F_{h+3} - 1 && (6.10)\\ &= \phi^{h+3}/\sqrt{5} - 1 && (6.11) \end{aligned}$$

where $\phi = (\sqrt{5}+1)/2$. Thus it follows that

$$h < 1.44 * \log(n+2) - 1.328 \tag{6.12}$$

and hence that

$$h = O(\log n) \tag{6.13}$$

Maintaining balance

Insertion and deletion into an AVL tree can cause it to become unbalanced. For example, the AVL tree of Fig. 6.21a becomes unbalanced due to insertion of the node with key value 32 (Fig. 6.22).

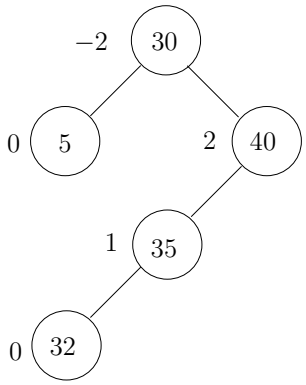
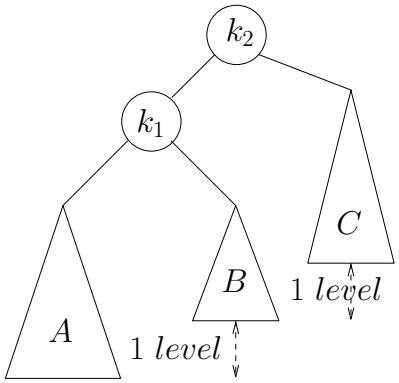
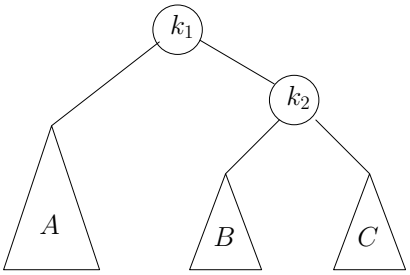


Figure 6.22: An unbalanced AVL tree

The imbalance is shown in a more generic fashion in Fig. 6.23a. The imbalance is corrected by an LL-rotation; the effect of this is shown by the generic figure below (Fig. 6.23b).



(a) Imbalance at k_2 due to insertion into the left subtree of the left child k_1



(b) LL rotation to correct imbalance

Figure 6.23: Generic figures showing imbalance and subsequent correction

On the particular example of Fig. 6.22, the effect of the LL-rotation is shown in Fig. 6.24.

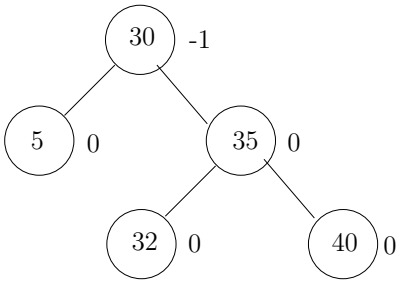
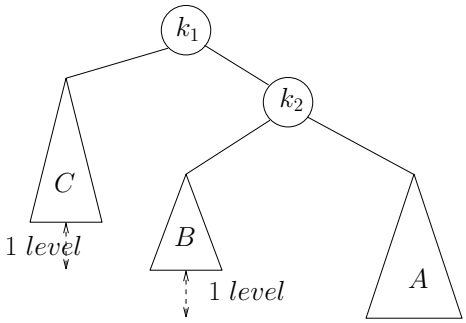
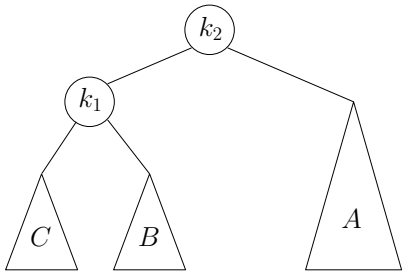


Figure 6.24: LL rotation to correct imbalance on example tree

Imbalances can also be caused by an insertion into the right subtree of the right child of a node as shown in Fig. 6.25a. Note that the subtree rooted at k_1 has become unbalanced. The imbalance is corrected by an RR-rotation; the effect of this is shown generically in Fig. 6.25b.



(a) Imbalance at k_1 on insertion into the right subtree of the right child k_2



(b) RR rotation to corrrect imbalance

Figure 6.25: Imbalance and correction

Imbalances can also be caused by an insertion into the right subtree of the left child of a node as shown in a generic way in Fig. 6.26.

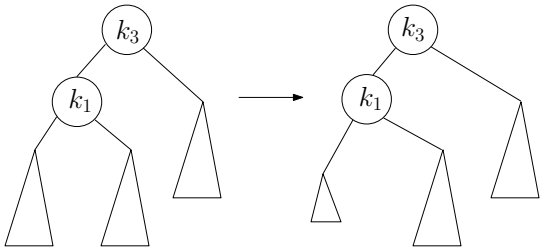


Figure 6.26: Imbalance at k_3 due to insertion into the right subtree of the left child k_1

The imbalance is corrected by an LR-rotation; the effect of this is shown in Fig. 6.27.

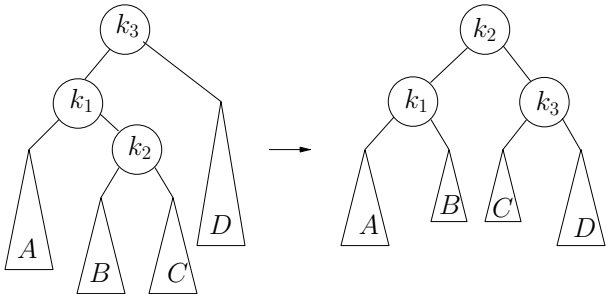
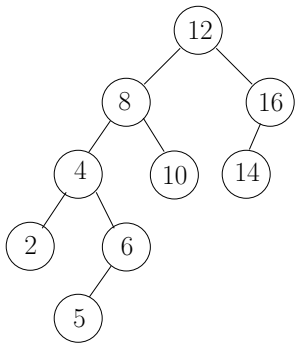


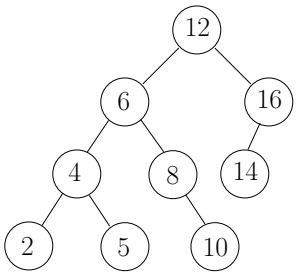
Figure 6.27: Imbalance corrected by LR rotation

A particular example of an imbalance corrected by an LR-rotation is shown in Fig. 6.28a. Note that the imbalance is at the node with key-value 8. The imbalance is corrected by an

LR-rotation as shown in Fig. 6.28b.



(a) Imbalance caused by insertion of key 5

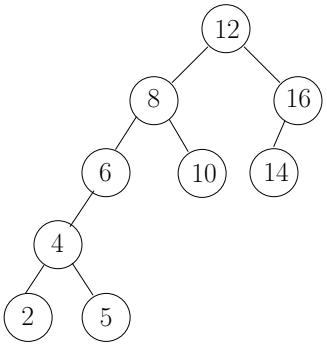


(b) Imbalance corrected by an LR rotation

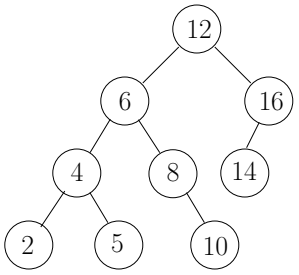
Figure 6.28: Imbalance and correction

Remark 2. An LR-rotation is often called a double rotation because it can be viewed as an RR rotation, followed by an LL-rotation.

Thus on the above example an RR-rotation gives the following tree (Fig. 6.29a). Followed by an LL-rotation gives the tree of Fig. 6.29b.



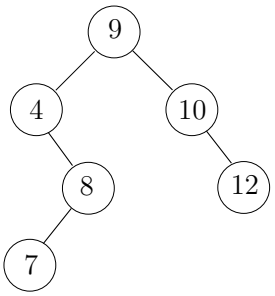
(a) Imbalance partially corrected by an RR rotation



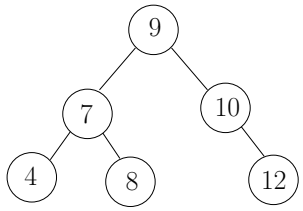
(b) Imbalance fully corrected by an LL rotation

Figure 6.29: Imbalance and correction

An example of an imbalance that calls for an RL-rotation is shown in Fig. 6.30a. The imbalance is caused by the insertion of key 7 into an initially balanced tree with the key values 5,4,10,8 and 12. Correcting the imbalance at the node with key-value 4, results in the tree of Fig. 6.30b.



(a) Imbalance caused on insertion of key 7



(b) Imbalance corrected by an RL rotation

Figure 6.30: Imbalance and correction

Exercise 6.3. Argue why just one rotation, single or double, suffices to restore the AVL property in case of an insertion.

Deletion from an AVL tree

Deletions are more complicated. We delete a node just as in a binary search tree. Let q be the parent of the physically deleted node. We might need to go all the way from q to the root, restoring balance at each node to make it AVL again.

Three cases arise:

c1 Balance at q has changed to 0.This implies that the height of the tree rooted at q has decreased by 1 and we need to check the balance factor of the ancestors of q.

c2 Balance at q has changed to +1 or -1. This implies that the height of the subtree rooted at q is unchanged, so the balance of all the ancestors of q remain unchanged.

c3 Balance at q has changed to +2 or -2. This implies that the tree needs to be balanced at q.

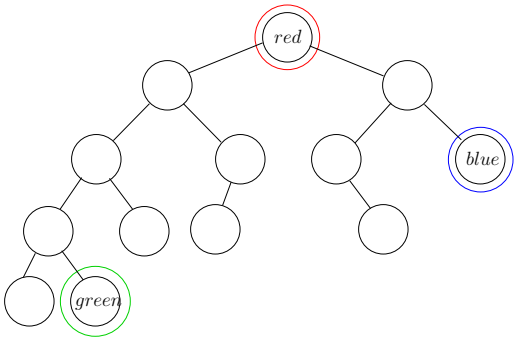


Figure 6.31: Three cases arising out of deletion, case **c1**: red circle, case **c2**: green circle, case **c3**: blue circle

Let A be the first node on the path from q to the root where the balance has changed to +2 or -2.

The imbalance is of type:

- L if deletion has taken place from the left subtree of A

– R if deletion has taken place from the right subtree of A

We discuss the R-type of imbalance. Such an imbalance with balance factor 2 implies that we had one of the following three configurations before deletion, shown on the left-hand sides of the figures below. The right-hand sides of the same figures show the configuration after deletion in each case.

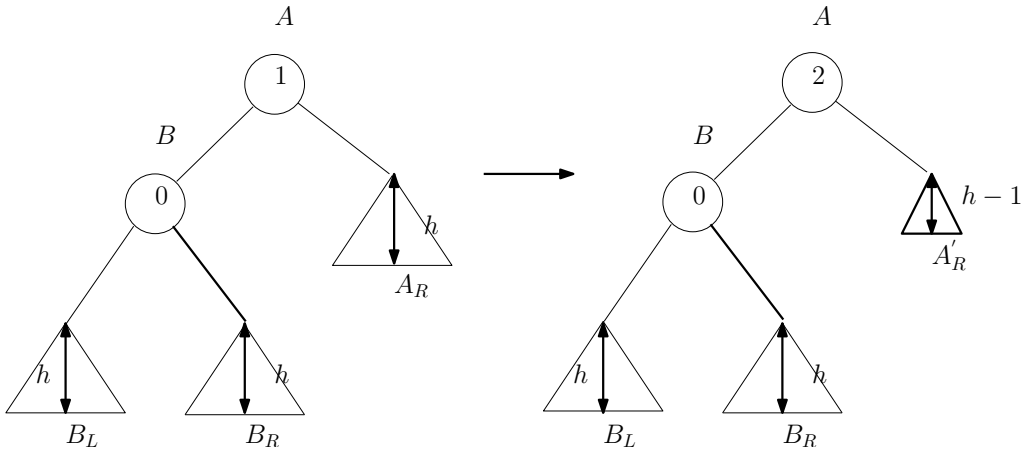


Figure 6.32: R0-type

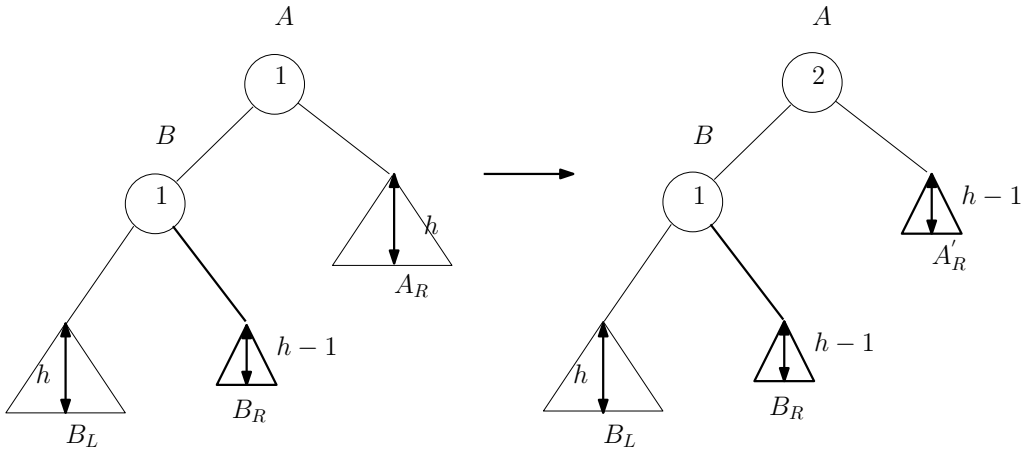


Figure 6.33: R1-type

The restorations in each of the cases above are shown in Fig. 6.35a, Fig. 6.35b, and Fig. 6.35c respectively.

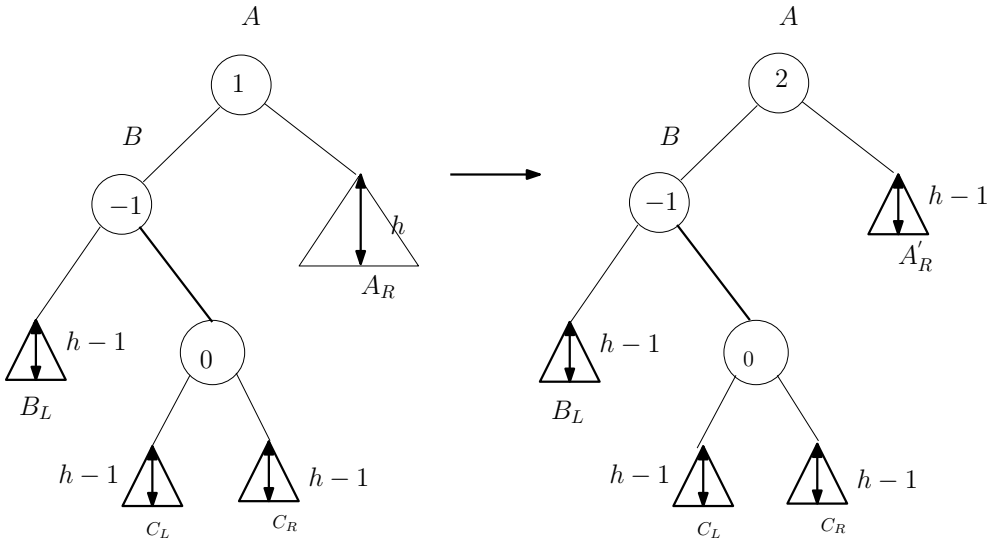
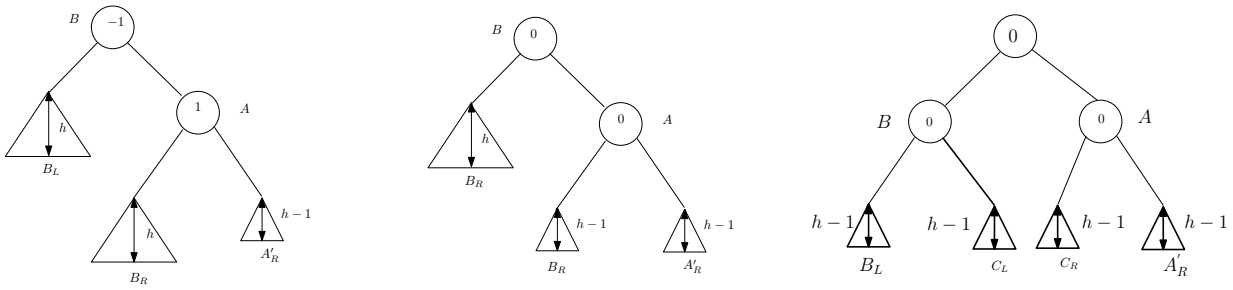


Figure 6.34: R-1-type



(a) LL rotation (b) LL rotation (c) LR rotation

Figure 6.35: Different types of rotation

Exercise 6.4. Draw all the tree configurations corresponding to an L-type of an imbalance and the corresponding restorations.

Let us illustrate all the cases that arise from deletions on the following AVL-tree (Fig. 6.36).

If we delete the node with key value 3, the new tree that we get is shown in Fig. 6.37:

This case corresponds to case **c2**, where the balance at the parent node has changed from 0 to 1; however, the effect does not propagate further up; no rebalancing is required.

If we delete the node with key value 14 from the original tree, the new tree that we get is shown in the Figure 6.38:

This case corresponds to case **c3**, where the balance at the parent node has changed from 1 to 2; An LR rotation at the node with key value 13 gives us the tree of Fig. 6.39a: However, the imbalance in this case has propagated further up, viz., at the root, where the balance has changed from 1 to 2, requiring us to rebalance at this node with an LL rotation. This gives us

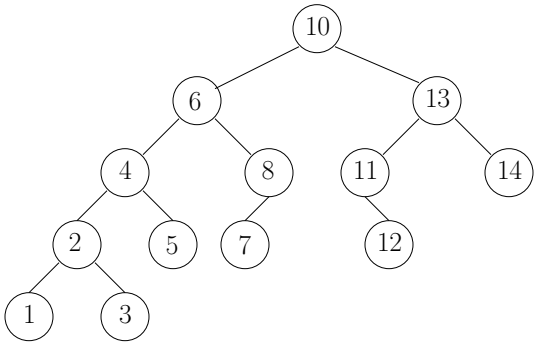


Figure 6.36: An example AVL tree

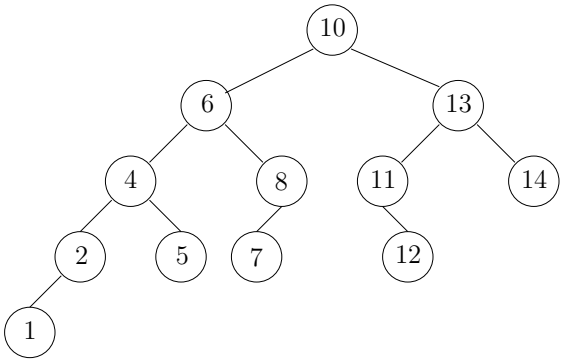


Figure 6.37: Tree resulting from deletion of key 3

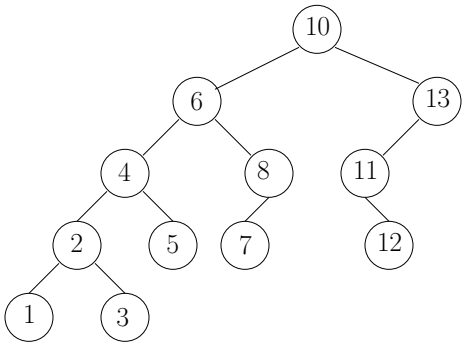
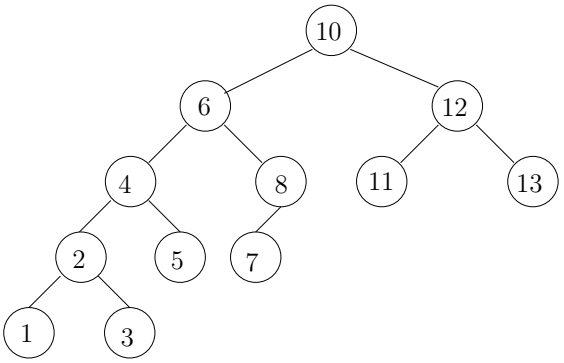
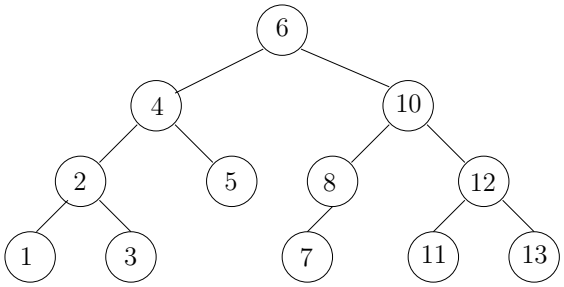


Figure 6.38: Tree resulting from deletion of key 14

the AVL tree of Fig. 6.39b.



(a) Tree resulting from LR rotation at node with key 13



(b) Tree resulting from LL rotation at node with key 13

Figure 6.39: Roatations to fix the imbalance

If we delete the node with key value 10 from the original tree, the new tree that we get is shown in the Figure 6.40, because of the way we delete a node with two children in a binary search tree.

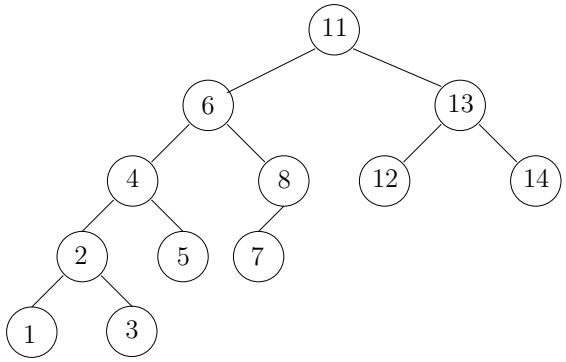


Figure 6.40: Tree resulting from deletion of node with key 10

This does not cause an imbalance at the node with key value 13, but the balance at the root has changed from 1 to 2 and can be restored by an LL rotation exactly as in the previous case.

Red-Black trees

Definition 6.7. A red-black tree is a binary search tree with the following properties:

P1 Each node is colored red or black

P2 The root is colored black

P3 A red node can only have black children

P4 Every path from a node to a null reference must contain the same number of black nodes

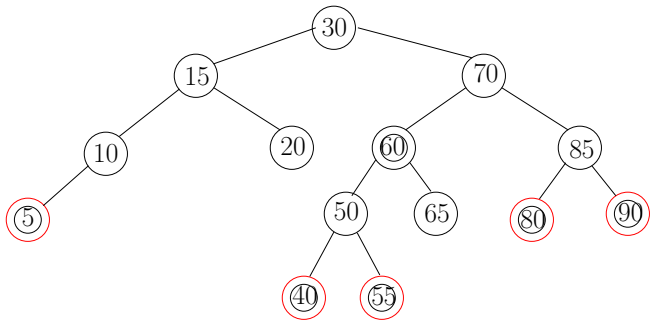


Figure 6.41: An example of a red-black tree

Note 6.2. In the figures of this section, we indicate a red node by a double circle, and a black node by a single circle

We first show that the height of a red-black tree on n nodes is $O(\log n)$.

Theorem 6.2. If every path from the root to a null node contains B black nodes then the number of black nodes is $\geq 2^B - 1$

Proof. By induction on B

Theorem 6.3. The height h of a red-black tree with n nodes is at most $2\log(n+1) - 1$

Proof. $n \geq 2^B - 1 \geq 2^{(h+1)/2} - 1$
$\rightarrow (h+1)/2 \leq \log(n+1)$
$\rightarrow (h+1) \leq 2\log(n+1)$
$\rightarrow h \leq 2\log(n+1) - 1$

The important conclusion is that:

$$h \quad = \quad O(\log n) \tag{6.14}$$

for a red-black tree with n nodes.

Insertion into a red-black tree

There are two ways of inserting a node into a red-black tree:

First Method

We first insert the node as a leaf, using its key value as we would in a binary search tree. The inserted node is colored red and we restore the red-black properties bottom-up. Two cases arise:

- Inserted node has a red parent, which has a black sibling

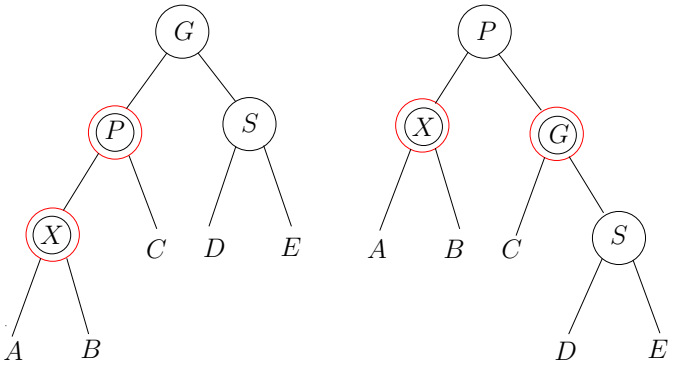


Figure 6.42: Single rotation restoration

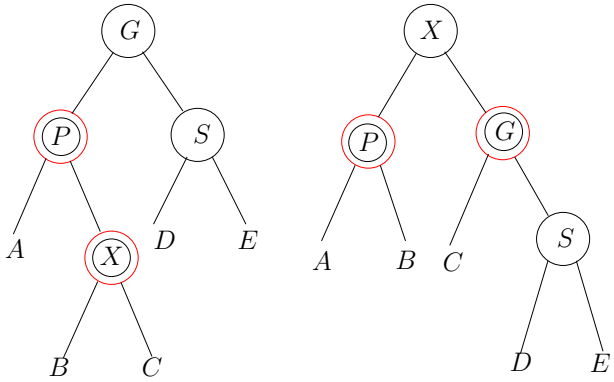


Figure 6.43: Double rotation restoration

- Inserted node has a red parent, which has a red sibling
  This case is problematic because P could have a red parent and we propagate the restoration to the root.

Second Method

In this method we insert a node top-down, observing the following rule: If we hit a node with red children, we flip the color of the node and the children.

We are helped by the following claim:

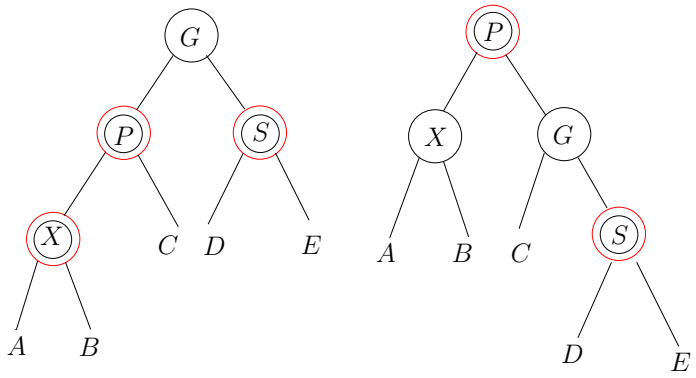


Figure 6.44: Double rotation restoration again

Claim 6.1. If the node's parent is red, then the node's sibling can't be red and we can apply a rotation to restore the red-black property.

We illustrate the ideas by inserting the key 45 into the example red-black tree of Fig. 6.41.

At the node with key value 50, we do a color flip. This gives us the tree of Fig. 6.45.

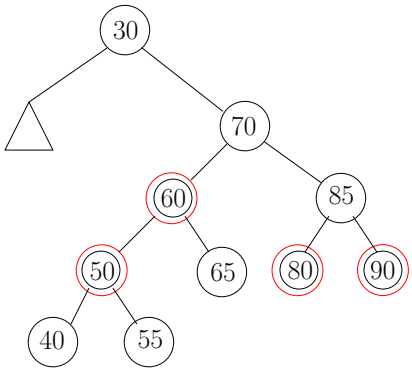
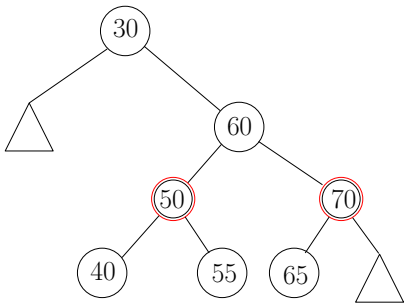
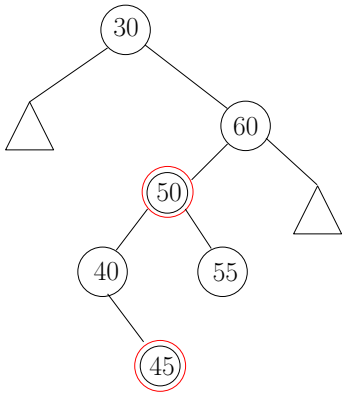


Figure 6.45: Color-flip at node with key 50 during top-down insertion of key 45

Then we do a single rotation at the node with key value 70. This gives us the tree of Fig. 6.46a. Continuing the descent according to the above rule, finally gives us the tree of Fig 6.46b.



(a) Single rotation at node with key value 70



(b) Key 45 finally in place

Figure 6.46: Rotations to fix the imbalance

Deletion from a red-black tree

Deletion is as in the case of a binary search tree, with the guarantee that the color of the physically deleted node is red.

As we search down the tree, we maintain the invariant that the color of the current node X is red.

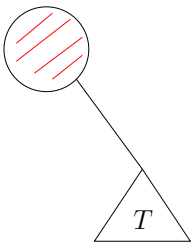


Figure 6.47: Introducing a sentinel node

Since the root node of a red-black tree is colored black, in order to maintain the above invariant we add a sentinel node, color it red and make the given red-black tree the right subtree of the sentinel node as in Fig. 6.47.

Let X be the current node. Let P be its parent and T its sibling. Inductively, P is red, implying that both X and T are black. Two cases arise:

Case 1 X has 2 black children

We have to deal with three different subcases.

* T has 2 black children
* T has an outer (inner) child that is red

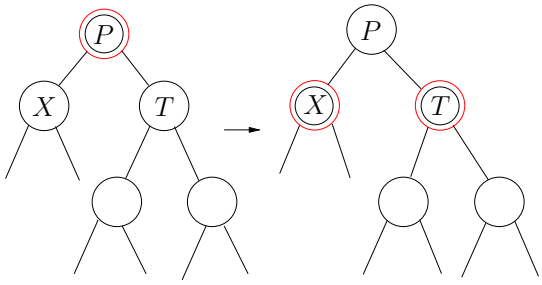
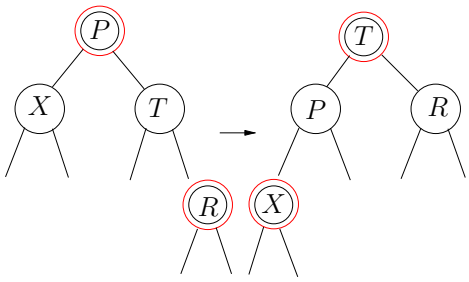
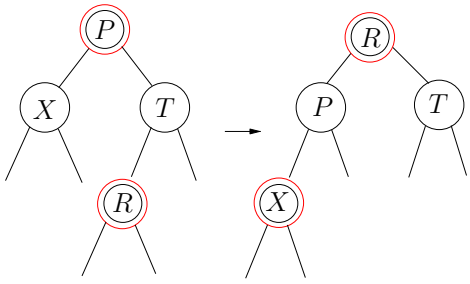


Figure 6.48: Flip colors when T has two black children



(a) Single rotation at P when T has an outer child that is red



(b) Double rotation at P when T has an inner child that is red

Figure 6.49: Roatations to fix the imbalance

Case 2 X has a red child

In this case drop one level, obtaining a new X, T and P and do a rotation as in the example of Fig 6.50.

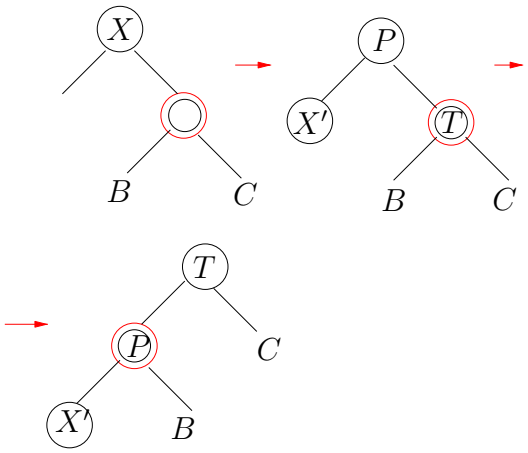


Figure 6.50: Single rotation continued from X'

Chapter 7

Hashing

We have seen that we can maintain a set of n keys in a balanced binary search tree (AVL or red-black) so that a FIND, INSERT, or DELETE operation takes $O(\log n)$ time in the worst case.

The question is can we do these operations in $O(1)$ time ? In a worst-case sense ? Yes, in this way:

If U denotes the universal set from which the keys are drawn, then we just maintain a bit vector the size of U that records the presence of an element by a 1 and its absence by a 0. This is called direct addressing. The disadvantage with this scheme is that if the size of the universal set is 2^{32}, on a 32-bit machine we would use up the entire memory. Thus we need to look for a scheme that strikes a more reasonable trade-off between time and space.

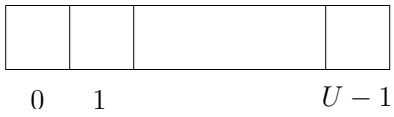


Figure 7.1: Bit vector corresponding to a universal set U

Yes, in an expected sense, by means of a hash table.

The following points about a hash function are noteworthy:

- The hash function, h, must be computationally simple.
- It must distribute keys "evenly" in the address space.
- Since the address space is usually much smaller than the size of the universal set U two or more keys can hash to the same spot in the hash table. This is called collision and we must have a collision-handling strategy.

7.0.1 Hash Functions

The division method

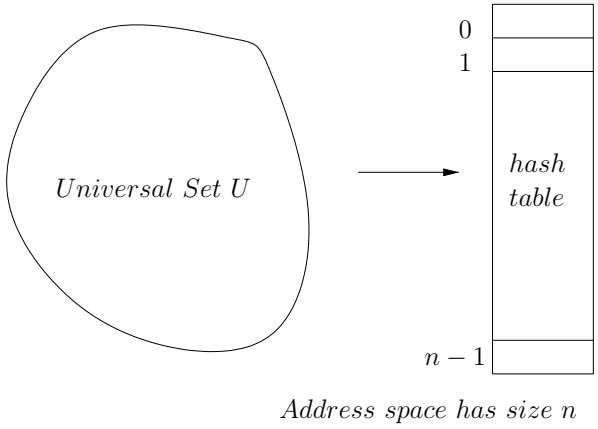


Figure 7.2: Mapping a universal set into a hash table

$$h : U \rightarrow \{0, 1, 2, \ldots, n-1\}$$

is defined as

$$h(k) \quad = \quad k \; mod \; n \tag{7.1}$$

Choice of n: Good values of n are primes not too close to exact powers of 2.

The multiplication method

In this method, we follow these steps:

- First, we multiply the key k by a constant A, $0 < A < 1$ and extract the fractional part of kA
- We multiply this value by n and take the floor of the result. That is

$$h(k) \quad = \quad \lfloor n(kA \; mod \; 1) \rfloor \tag{7.2}$$

  Here, the choice of n is not critical. We take it to be a power of 2. Knuth suggests the following value for A.

$$A \quad = \quad \frac{\sqrt{5}-1}{2} \tag{7.3}$$

  If $k = 123456$ and $n = 10000$, then with A as above, we get

$$h(k) \quad = \quad 41 \tag{7.4}$$

We will assume that the keys in the universal set U happens to be the set of natural numbers $N = \{0, 1, 2, \ldots, \}$. If they are not, we can suitably interpret them to be natural numbers. For example, a string over the set of ASCII characters, can be interpreted as an integer in the base

128. Thus the string "pt" can be interpreted as the integer 128.112 + 116 = 14452

Universal Hashing

No matter what hash function we may choose, there always exists adversarial choices of key values that map to the same location, giving rise to collisions. One approach to dealing with this problem is to have a collection of hash functions from which we choose one at random. It is possible to construct a class of hash functions, called 2-universal, such that if $h()$ is any function from this class then

$$\text{Probability}[h(x_1) = h(x_2)] < 1/n$$

What this implies is that $h()$ is as good as any random mapping from $U \to \{0, 1, 2, \ldots, n-1\}$. For more details see the text-book "Introduction to Algorithms", by Cormen et. al.

7.1 Open Addressing

In open addressing all elements are stored in the table itself. In these notes, we discuss only open addressing.

Insertion

To perform insertion, we successively examine or probe the hash table until we find an empty slot in which to put the key.

The sequence of positions probed depends on the key being inserted.

Searching

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. The search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence (this argument assumes that keys are not deleted from the hash table).

Deletion

When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by a special value instead of leaving it empty.

We next discuss various schemes for probing a hash table. An ideal scheme would be one that satisfies the assumption of uniform hashing. This means that all possible permutations of the hash table addresses $0, 1, \ldots, n-1$ are equally likely to be a probe sequence. We will examine each possible probe model vis-a-vis this assumption.

7.1.1 Linear Probing

Given a key k and a hash function $h(k)$, a linear probe sequence $h_i(k)$ for $i = 0, 1, \ldots$ is defined as follows:

$$h_0(k) = h(k)$$
$$h_i(k) = (h(k) + i) \bmod n,\ i = 1, 2, \ldots$$

where n is the size of the hash table.

As should be clear, there is exactly one probe sequence possible, determined by the first position probed. This behavior is quite far removed from the assumption of uniform hashing. Linear probing is easy to implement though, but it suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time.

Example

Let $h(k) = k\ mod\ 10$, and the hash table size = 10.

Insert 89

$$h(89) = 89\ mod\ 10 = 9$$

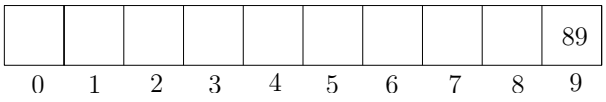


Figure 7.3: Insertion of 89 into hash table

Insert 18

$$h(18) = 18\ mod\ 10 = 8$$

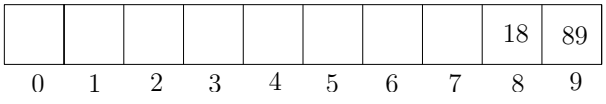


Figure 7.4: Insertion of 18 into hash table

Insert 49

$$h(49) = 49\ mod\ 10 = 9$$

We have a collision! Since

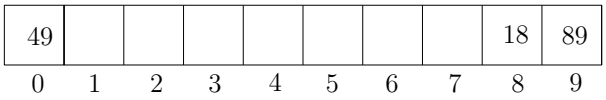


Figure 7.5: Insertion of 49 into hash table

$$\begin{aligned} h_1(49) &= (h(49) + 1) \bmod 10 \\ &= (9 + 1) \bmod 10 \\ &= 0 \end{aligned}$$

the search wraps around to the unoccupied location 0, where 49 is inserted.

Insert 58

$$h(58) = 58 \; mod \; 10 = 8$$

The first two locations $h_1(58) = 9$ and $h_2(58) = 0$ generated by the probe sequence lead to collisions and the search wraps around to the unoccupied location 1, where 58 is inserted.

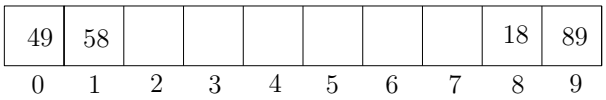


Figure 7.6: Insertion of 58 into hash table

Insert 9

$$h(9) = 9 \; mod \; 10 = 9$$

Since the locations generated by the probes $h_1(9) = (h(9) + 1) \bmod 10 = 0$ and $h_2(9) = (h(9)+2) \bmod 10 = 1$ are occupied (three successive collisions so far) the search wraps around to the unoccupied location 2, where the key 9 is finally inserted.

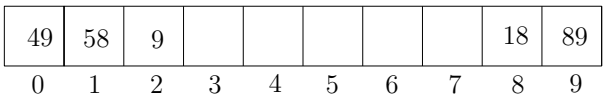


Figure 7.7: Insertion of 9 into hash table

7.1.2 Quadratic Probing

Given a key k and a hash function $h(k)$, a quadratic probe sequence $h_i(k)$ for $i = 0, 1, \ldots$ is defined by

$$h_0(k) = h(k)$$
$$h_i(k) = (h(k) + c_1 * i + c_2 * i^2) \bmod n, i = 1, 2, \ldots, n-1$$

where n is the size of the hash table and c_1 and c_2 are auxiliary constants. This works much better than linear probing.

Vis-a-vis the uniform hashing assumption, quadratic probing is no better than linear probing. For a given key k there is only one probe sequence generated, determined by the first position probed. The redeeming feature, however, is that we avoid the primary clustering problem of linear probing.

Example
Let the hash function $h(k) = k \ mod \ 10$ and the hash table size is 10. The probe sequence is generated by:

$$h_0(k) = h(k)$$
$$h_i(k) = (h(k) + i^2) \ mod \ 10, i = 1, \ldots$$

Insert 89

Since

$$h(89) = 89 \ mod \ 10 = 9$$

and the location 9 is unoccupied 89 is inserted into this location.

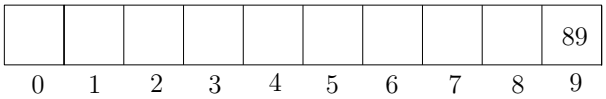


Figure 7.8: Insertion of 89 into hash table, quadratic probing

Insert 18

Since

$$h(18) = 18 \ mod \ 10 = 8$$

and the location 8 is unoccupied 18 is inserted into this location.

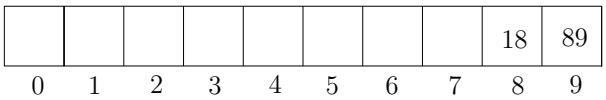


Figure 7.9: Insertion of 18 into hash table, quadratic probing

Insert 49

Since

$$h(49) = 49\ mod\ 10 = 9$$

and the location 9 is occupied we have a collision. The next location generated by the robe sequence is:

$$\begin{aligned} h_1(49) &= (h(49) + 1^2) \\ &= (9 + 1^2) \bmod 10 \\ &= 0 \end{aligned}$$

The location 0 is unoccupied and hence 49 is inserted here.

49								18	89
0	1	2	3	4	5	6	7	8	9

Figure 7.10: Insertion of 49 into hash table, quadratic probing

Insert 58

Since

$$h(58) = 58 \bmod 10 = 8$$

and the location 8 is occupied, we have a collision. The next location generated is:

$$\begin{aligned} h_1(58) &= (h(58) + 1^2) \bmod 10 \\ &= (8 + 1) \bmod 10 \\ &= 9 \end{aligned}$$

We have a collision again as location 9 is also occupied. The next location generated is:

$$\begin{aligned} h_2(58) &= (h(58) + 2^2) \bmod 10 \\ &= (8 + 4) \bmod 10 \\ &= 2 \end{aligned}$$

As location 2 is not occupied, we insert 58 into this location.

49		58						18	89
0	1	2	3	4	5	6	7	8	9

Figure 7.11: Insertion of 58 into hash table, quadratic probing

Insert 9

Since

$$h(9) = 9 \bmod 10 = 9$$

and the location 9 is occupied, we have a collision. The next location generated is:

$$\begin{aligned} h_1(9) &= (h(9) + 1^2) \bmod 10 \\ &= (9 + 1) \bmod 10 \\ &= 0 \end{aligned}$$

Since location 0 is occupied we still have a collision again! The next location generated is:

$$\begin{aligned} h_2(9) &= (h(9) + 2^2) \bmod 10 \\ &= (9 + 4) \bmod 10 \\ &= 3 \end{aligned}$$

Since the location 3 is not occupied, 9 is inserted into this location.

49		58	9					18	89
0	1	2	3	4	5	6	7	8	9

Figure 7.12: Insertion of 9 into hash table, quadratic probing

The analysis below also makes the assumption of uniform hashing.

Analysis of open-address hashing

Let $\alpha = m/n$, where m of n slots in the hash table are occupied. α is called the load factor and is clearly $\alpha \leq 1$.

Theorem 7.1. Given an open-address hash-table, with load factor $\alpha < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$

Theorem 7.2. Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average $\alpha < 1$

Theorem 7.3. The expected number of probes in a successful search is at most $1/\alpha ln(1/1 - \alpha) + 1/\alpha$, assuming each key in the table is equally likely to be searched for ($\alpha < 1$).

7.1.3 More on Quadratic Probing

- Will we always be able to insert element X if table is not full ?
- Ease of computation ?
- What happens when the load factor gets too high ? (applies to linear probing as well)

The following theorem in your text addresses the first issue.

Theorem 7.4. If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty. Also, no cell is probed twice in the course of insertion.

The next theorem shows how we can do the calculation more efficiently for the above example.

Theorem 7.5. If H_i denotes the ith probe, then

$$H_i = H_{i-1} + 2 * i - 1 (mod\, n)$$

where H_0 is the initial probe.

What do we do when the load factor gets too high ? We rehash!

- We double the size of the hash table
- Rehash the entries in the current table into the new one

Chapter 8

Applications

8.1 Huffman Coding

The first application is about encoding and decoding messages. A message is a piece of text over a fixed source alphabet. For example, the message M

$$M = aabccdaa...$$

is over the source alphabet set $\{a, b, c, d\}$.

The encoded message is:

0000011010110000..

obtained by replacing each letter by its codeword. The problem before us is to obtain an encoding scheme that minimizes the length of the encoding.

The above encoding was obtained from the following encoding scheme:

symbol	**codeword**
a	00
b	01
c	10
d	11

This is an example of a fixed-length encoding scheme where the codeword of each symbol in the message has the same number of bits.

Problem 8.1. How many bits do we need to encode uniquely each symbol in a message made up of symbols drawn from an n-letter alphabet ?

In a variable-length encoding scheme the codeword for each symbol in the message is of a different length. Thus for a message made up of symbols from the alphabet $\{a, b, c\}$ we can have the following encoding scheme:

symbol	codeword
a	10
b	1
c	0

The problem with this coding scheme is that during decoding it is difficult to distinguish a from bc as both have the same encoding, viz., 10. This problem can be corrected by the following modified scheme.

symbol	codeword
a	00
b	01
c	1

Prefix Codes

The last encoding scheme is an example of what we call a prefix code (really we should say prefix-free codes but this is the nomenclature followed in the literature). We define this formally.

Definition 8.1. The code of a symbol α is not the prefix of the code of a symbol β if the former is not a prefix of the latter. A prefix code is one in which this is true for every pair of symbols in the source alphabet.

Prefix codes are interesting because these allow compression. We take up an example to illustrate the point. Given the coding schemes of the tables below:

symbol	codeword
a	000
b	001
c	01
d	1

symbol	codeword
a	00
b	01
c	10
d	11

the message $M = addd$ encodes to 000111, using the scheme of the first table, while it encodes to 00111111, using the scheme of the second table. Clearly, the scond encoded form is longer.

8.1.1 Huffman's scheme

Huffman proposed a greedy scheme for generating prefix codes, using binary trees. We illustrate his algorithm, using a specific example.

Suppose the symbols a, b, c, d, e, f, g occur in a text T with frequencies, given by the following table.

symbol	**frequency**
a	1
b	3
c	4
d	10
e	12
f	13
g	15

Initially, we have a forest of binary trees, one corresponding to each character. The root of each tree in the forest is labeled with the frequency of the corresponding symbol.

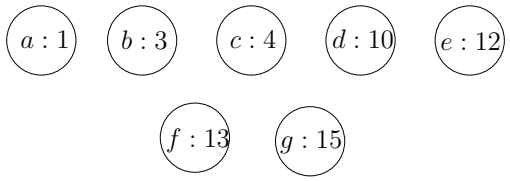


Figure 8.1: The initial forest

We reduce this forest to a single binary tree in a series of steps, according to the following rule:

Rule: At each step merge two trees with the smallest frequencies, where ties are broken arbitrarily. Label the left edge from the root of the merged tree 0, while the right edge is labeled 1. The frequency of the merged tree is the sum of the frequencies of the merged trees.

Thus following the above rule we have the following reduction of the forest to a single binary tree.

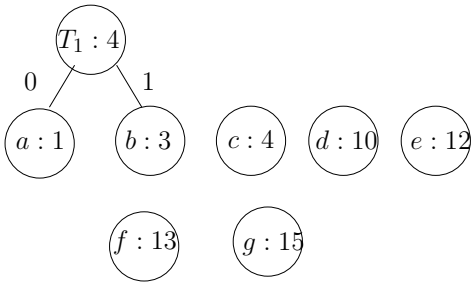


Figure 8.2: The forest after merging a and b

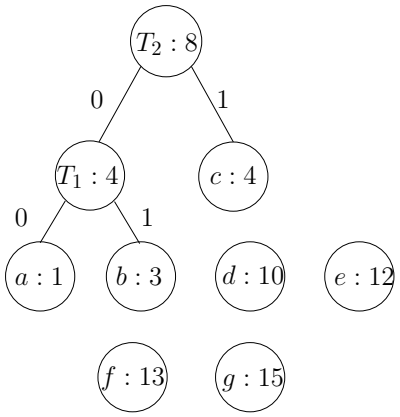


Figure 8.3: The forest after merging T_1 and c

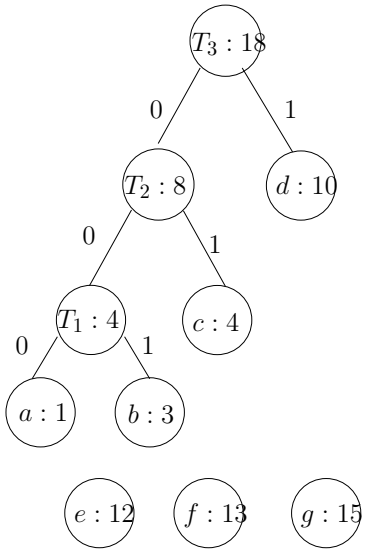


Figure 8.4: The forest after merging T_2 and d

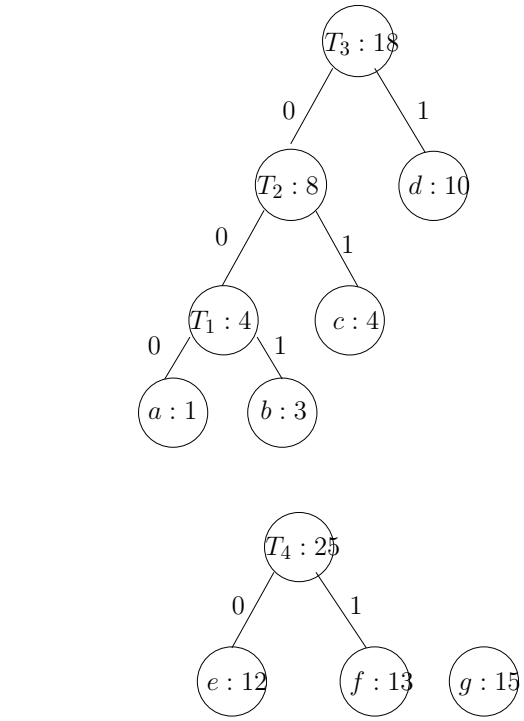


Figure 8.5: The forest after merging f and e

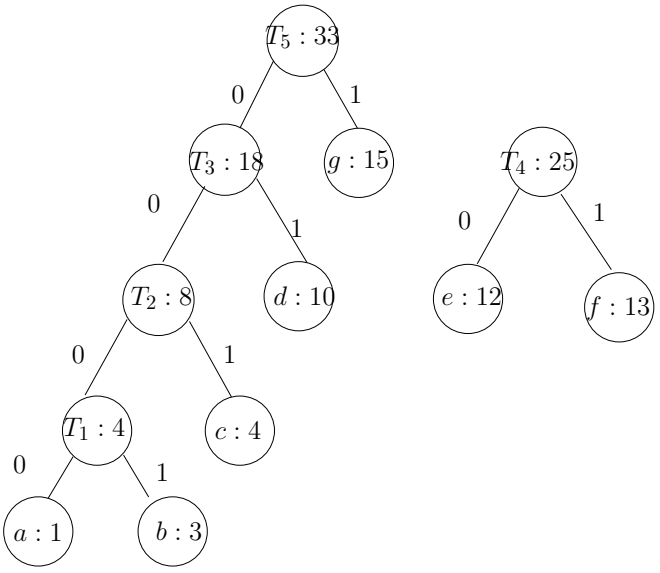


Figure 8.6: The forest after merging T_3 and g

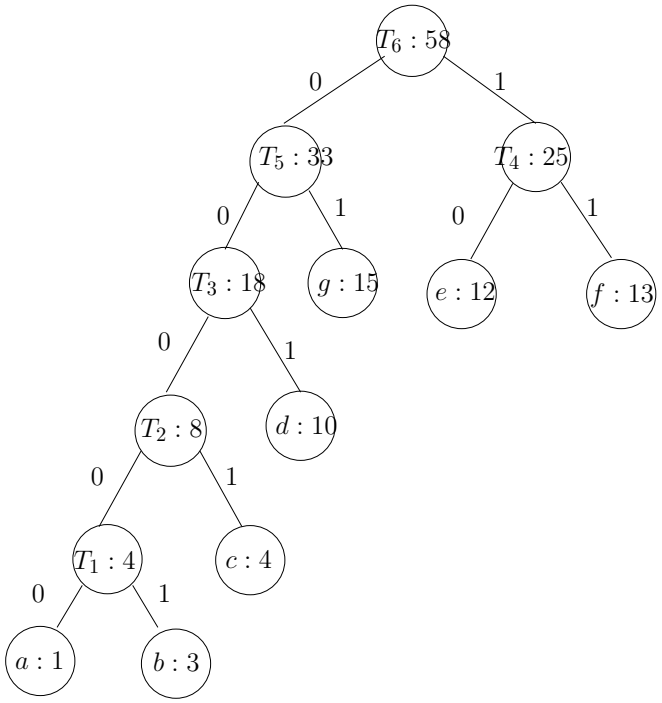


Figure 8.7: The forest after merging T_5 and T_4

From the binary tree obtained by the above reduction, we get the following table of prefix codes:

symbol	**codeword**
a	00000
b	00001
c	0001
d	001
e	10
f	11
g	01

Remark 3. Huffman's algorithm constructs a full binary tree. This tree has optimal cost, meaning that the value of $\sum_{i=1}^{m} l_i f_i$ is the smallest for the given frequencies, where l_i is the code length for the symbol that has frequency f_i and m is the number of leaves in the binary tree.

Remark 4. Other optimal codes can be obtained from Huffman's tree by exchanging leaf labels at the same depth.

8.2 Graph Algorithms

Definition 8.2. A graph consists of a finite set of vertices, and edges connecting pairs of vertices.

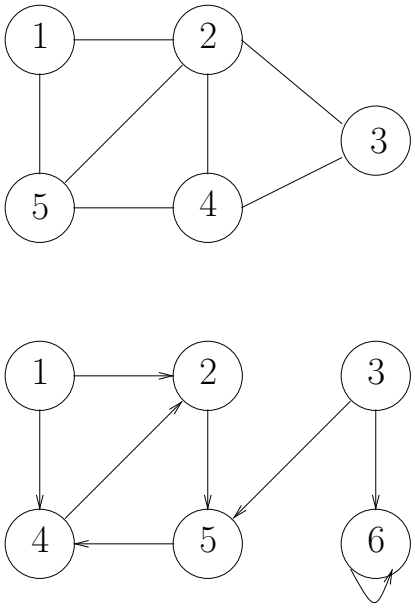


Figure 8.8: Examples of some graphs:(a) undirected (b) directed

The edges can be undirected as in Fig 8.8(a) or directed as in Fig 8.8(b). We denote a graph by $G = (V, E)$, where V is the set of vertices and E is the set of edges. G is directed if the edges are, else G is undirected.

8.2.1 Graph Representation

Graphs can be represented by adjacency lists. An adjacency list is an array of pointers, one for each vertex; each pointer points to a list of vertices it is adjacent to. The graph of Fig 8.8(a) is represented by the adjacency list below.

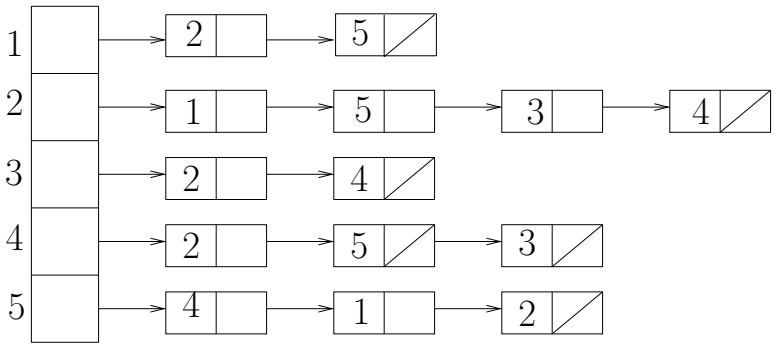


Figure 8.9: Adjacency list for graph in Fig.8.8(a)

The adjacency list representation of the graph of Fig. 8.8(b) is shown below.

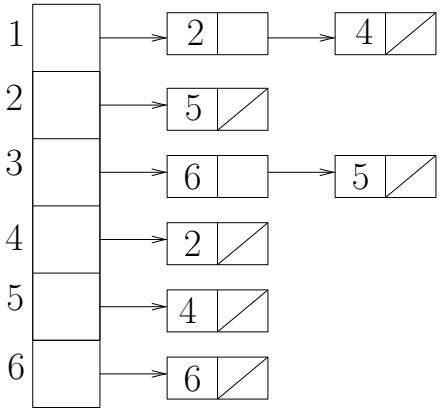


Figure 8.10: Adjacency list for graph in Fig.8.8(b)

Graphs can also be represented by adjacency matrices. An adjacency matrix is a $|V| \times |V|$ matrix. The entry in the ith row and jth column is 1 if there is an edge between the vertices i and j, else the entry is 0. Clearly, the matrix is symmetric if the graph is undirected.

The adjacency matrix representation of the undirected graph of Fig 8.8(a) is shown in the table below.

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

The adjacency matrix representation of the directed graph of Fig 8.8(b) is shown in the table below.

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

8.2.2 Breadth-first search

A fundamental tool for answering many questions related to graphs is breadth-first search. It is also called level-order search.

Given an undirected or directed graph $G = (V, E)$ and a "source" vertex $s \in V$, the search consists of systematically exploring vertices reachable from S to construct what is called a

breadth-first tree. The length of the path from S to each reachable vertex is the shortest path from S to this vertex.

We work out the details of this search on the example graph in Fig. 8.11 below.

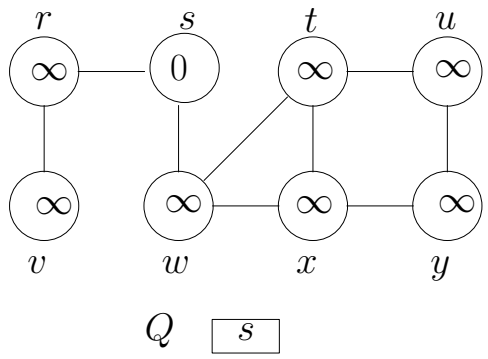


Figure 8.11: Example graph for breadth-first search

The value inside each circle, representing a vertex of the graph, denotes the distance of that vertex from the source vertex s. Initially, this distance is ∞ for all vertices except the source for which this is 0.

The data structure that facilitates the search is a queue Q. Each step of the search consists of the following steps. The vertex that we are currently visiting is dequeued and the vertices adjacent to this vertex are entered into this queue in any order. The queue is initialized with the source vertex s.

At any stage in the search, each vertex belongs to one of three groups: those that have already been visited, those that are currently in the queue and those that have not been visited yet.

For the example above, s is dequeued and the vertices r and w that are adjacent to it are enqueued. The labels of the vertices r and w are updated to 1. The status of the search is shown in Fig. 8.12 below. A line has been drawn through the vertices that are in the queue (r and w) to separate the vertices that have already been visited, viz., s from the vertices that have not yet been visited, viz., t, u, v, x and y.

Next, we dequeue the vetex w, the vertices t and x that are adjacent to it are enqueued, while the distance labels of these vertices are updated to 2 (see Fig. 8.13). The separating line has also been appropriately changed to go through the vertices that are currently in the queue.

The remaining steps in the search are shown in the figures below.

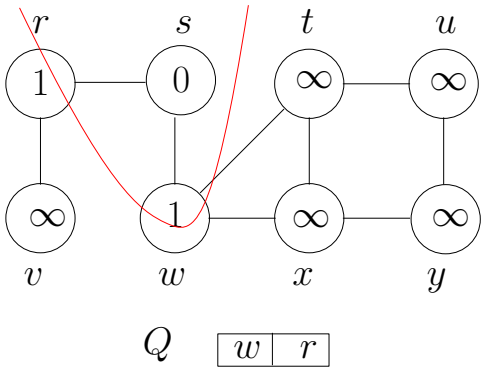


Figure 8.12: The graph after the first stage in the search

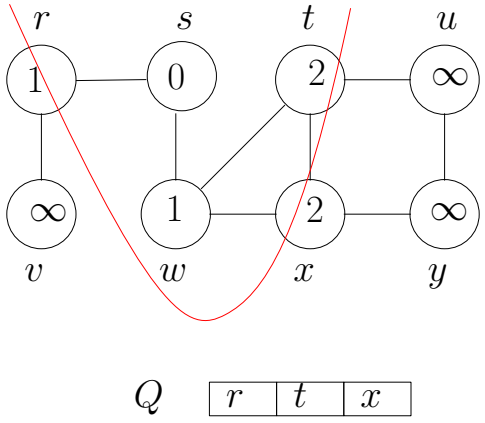


Figure 8.13: The graph after the second stage in the search

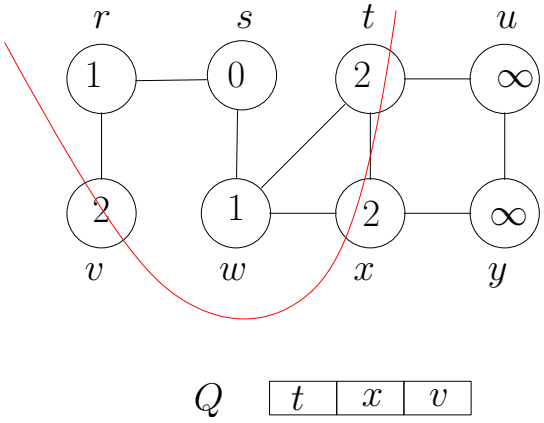


Figure 8.14: The graph after the third stage in the search

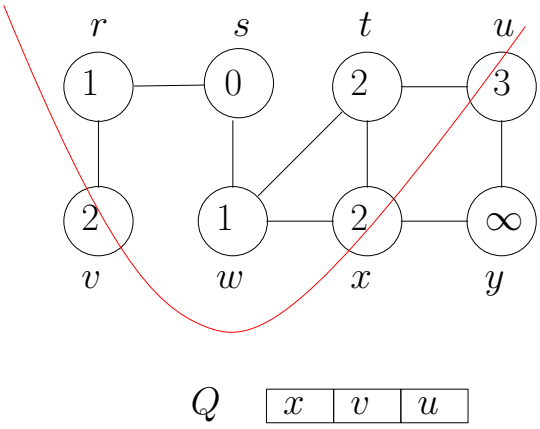


Figure 8.15: The graph after the fourth stage in the search

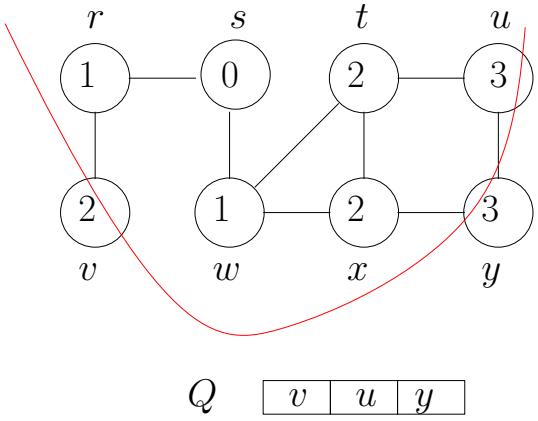


Figure 8.16: The graph after the fifth stage in the search

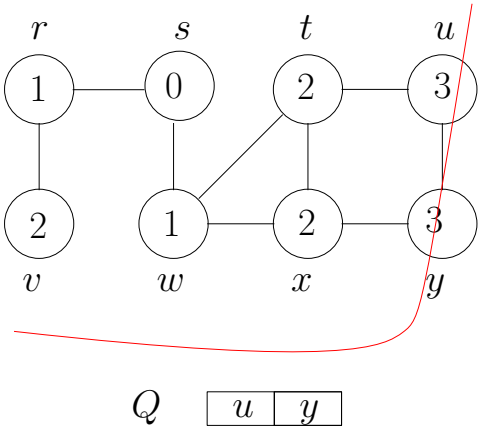


Figure 8.17: The graph after the sixth stage in the search

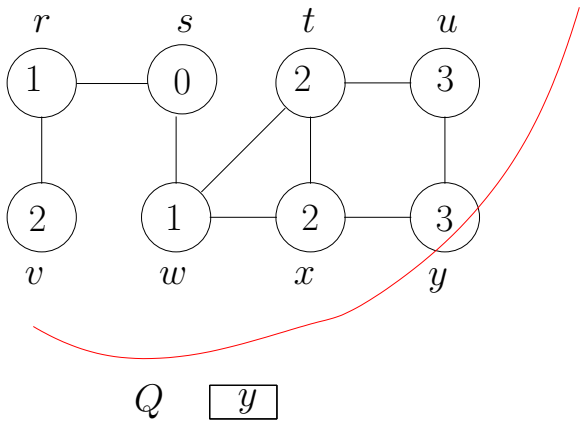


Figure 8.18: The graph after the seventh stage in the search

To describe the algorithm formally we use the following color terminology:

- Gray for vertices that are currently in the queue (separating line in the figures above)
- White for vertices that have not been visited
- Black for nodes that have already been visited

We also use the objected-oriented notation $o.f$, where f is a function defined on an object o.

Algorithm 27 BreadthFirstSearch

Input: A graph G and a source vertex s
Output: A BreadthFirstSearch tree

```
1:  for each vertex u ∈ V − {s} do
2:      u.color ← WHITE
3:      u.distance ← ∞
4:      u.parent ← NULL
5:  end for
6:  s.distance ← 0
7:  s.parent ← NULL
8:  s.color ← GRAY
9:  Add s to the queue Q
10: while Q ≠ ∅ do
11:     u ← Q.front
12:     for each v adjacent to u do
13:         if (v.color is WHITE) then
14:             v.color ← GRAY
15:             v.distance ← u.distance + 1
16:             v.parent ← u
17:             Add v to the queue Q
18:         end if
19:     end for
20:     Remove u from the queue Q
21:     u.color ← BLACK
22: end while
23: Output the BreadthFirstSearch tree
```

Complexity Analysis

Theorem 8.1. The running time of BFS is in $O(|V| + |E|)$

Proof. Queue operations take $O(|V|)$ time. Each edge in the graph is looked at most once. Hence $O(|E|)$ time and the theorem.

We can prove the following facts:

Fact 1 Breadth-first search discovers every vertex reachable from s

Fact 2 The length of the path from s to a reachable vertex v, is a shortest path from s to v.

8.2.3 Minimum Spanning Tree

In many practical applications, like laying a telephone network, we are often interested in laying a network that connects all the sites, while at the same time trying to keep the cost of the cabling as low as possible. This and many other problems can be abstractly reformulated as the problem of finding a minimum spanning tree (MST for short) in a graph whose edges have weights.

In the graph shown in Fig. 8.18, a minimum spanning tree is shown by thick edges. The number adjacent to each edge is its weight.

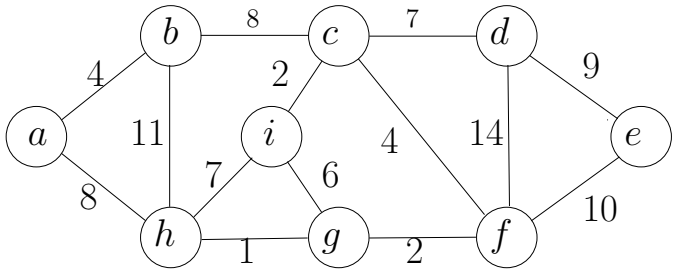


Figure 8.19: A weighted graph

Formally, we can state the problem as follows:

> Given a connected, weighted graph $G = (V, E)$ find an acyclic (means, free of cycles) subset $T \subset E$ that connects all the vertices such that $W(T) = \sum_{(u,v)\in T} w(u, v)$, where $w(u, v)$ is the weight of the edge connecting u and v, is minimum.

Growing an MST, greedily

Let A be a subset of edges of some MST.

Definition 8.3. An edge $e = (u, v)$ of G is safe for A if $A \cup \{(u, v)\}$ is also a subset of an MST.

Algorithm 28 GenericMST(G, w)

Input: A weighted graph G
Output: A Minimum Spanning Tree

1: $A \leftarrow \phi$
2: **while** (A does not form an MST) **do**
3: find a safe edge (u, v) for A ;
4: $A \leftarrow A \cup \{(u, v)\}$;
5: **end while**
6: **return** A ;

A generic MST algorithm

We first describe a generic MST algorithm, based on the greedy-paradigm.

Cuts and safe edges

Definition 8.4. A cut of G is a partition of V into two complementary subsets S and $V - S$, that we denote by $(S, V - S)$.

The figure below shows an example of a cut of the graph of Fig 8.18, where $V = \{a, b, d, e\}$ and $V - S$ is made up of the rest of the vertices.

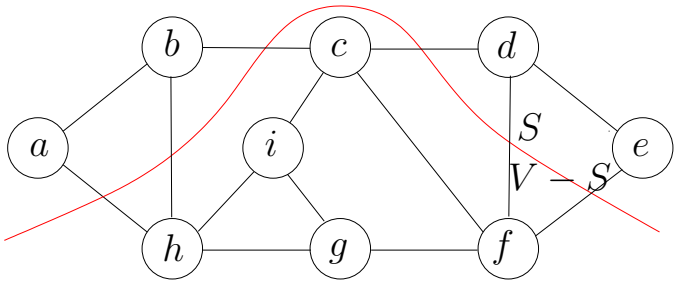


Figure 8.20: Example of a cut

We need some more definitions to describe a more specific algorithm.

Definition 8.5. An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its end-points is in S and the other in $V - S$.

Definition 8.6. A cut respects the set A if no edges in A crosses the cut.

Definition 8.7. An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

The following theorem characterizes safe edges for A.

Theorem 8.2. Let A be a subset of E that is included in some MST for G; let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

8.2.4 Prim's Algorithm

Prim's algorithm is based on the above definitions. We demonstrate the algorithm on the example graph of Fig 8.18 first and then state it formally.

We let $S = \{a\}$. The edges crossing the cut $(S, V - S)$ are (a, b) and (a, h), of which the former is a light edge crossing this cut.

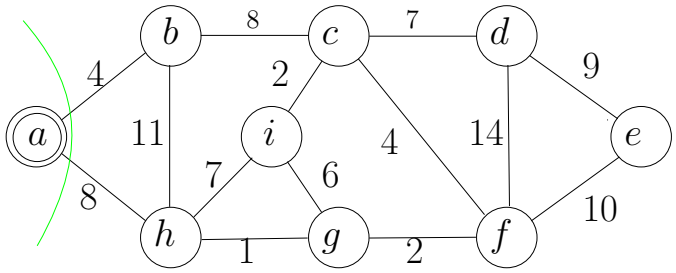


Figure 8.21: Selection of vertex a

This expands S to $\{a, b\}$. The edges crossing the new cut are $(a, h), (b, h)$ and (b, c). Of these both (a, h) and (b, c) are light edges. We choose (this is arbitrary) (b, c).

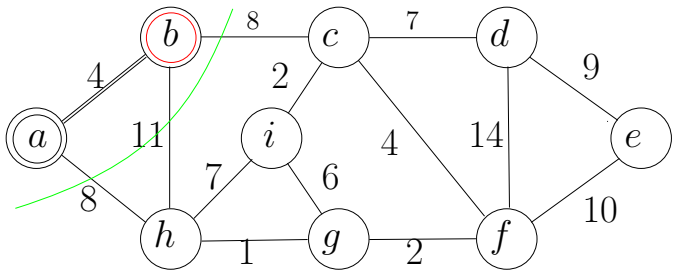


Figure 8.22: Selection of vertex b

This expands S to $\{a, b, c\}$. The edges crossing the new cut are $(a, h), (b, h), (c, i), (c, f)$ and (c, d). Of these (c, i) is a light edge crossing the cut.

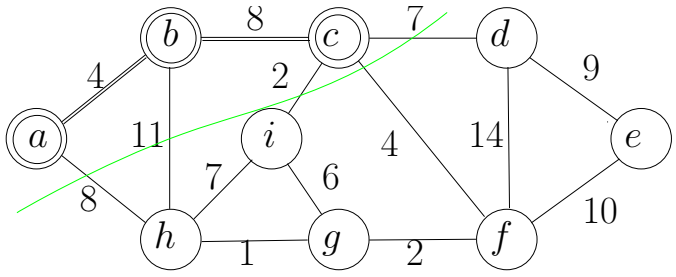


Figure 8.23: Selection of vertex c

This expands S to $\{a, b, c, i\}$. The edges crossing the new cut are $(a, h), (b, h), (i, h), (i, g), (c, d)$ and (c, f). Of these, (c, f) is a light edge crossing the cut.

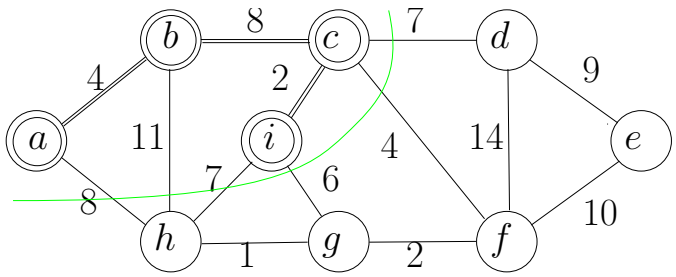


Figure 8.24: Selection of vertex i

This expands S to $\{a, b, c, i, f\}$. The edges crossing the new cut are $(a, h), (b, h)$, $(i, h), (i, g)$, $(f, g), (f, d), (f, e)$ and (c, d). Of these (f, g) is a light edge crossing the cut.

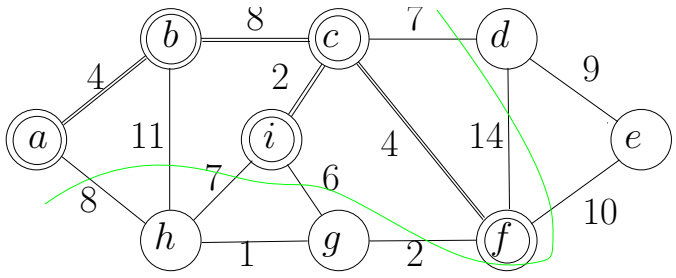


Figure 8.25: Selection of vertex f

This expands S to $\{a, b, c, i, f, g\}$. The edges crossing the new cut are $(a, h), (b, h), (i, h), (g, h), (f, d), (f, e)$ and (c, d). Of these (g, h) is a light edge crossing the cut.

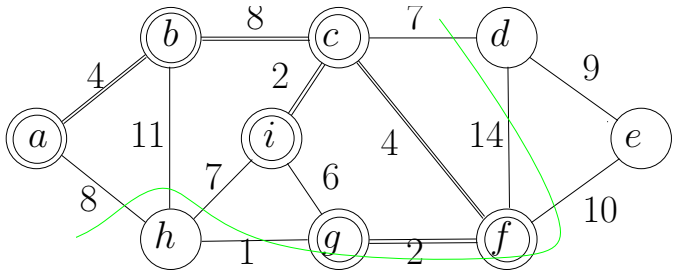


Figure 8.26: Selection of vertex g

This expands S to $\{a, b, c, i, f, g, h\}$. The edges crossing the new cut are $(c, d), (f, d)$ and (f, e). Of these (c, d) is a light edge crossing the cut.

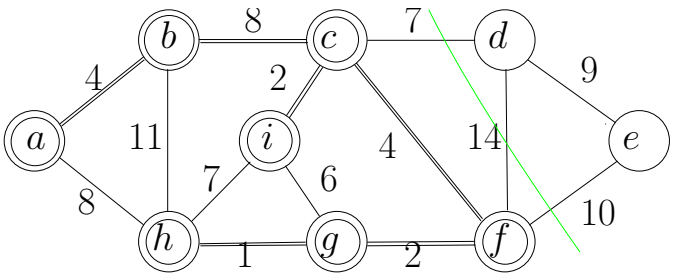


Figure 8.27: Selection of vertex h

This expands S to $\{a, b, c, i, f, g, h, d\}$. The edges crossing the new cut are (d, e) and (f, e). Of these (d, e) is a light edge crossing the cut.

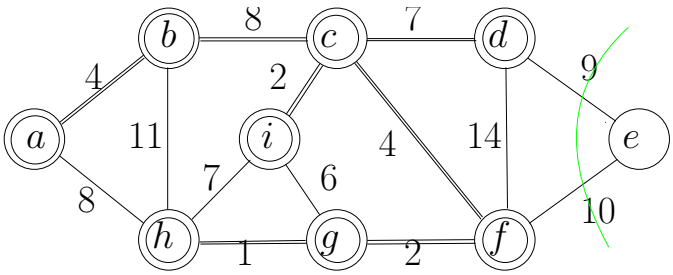


Figure 8.28: Selection of vertex d

The resulting edges shown by double lines in Fig. 8.28 below is a Minimum Spanning Tree for our example graph.

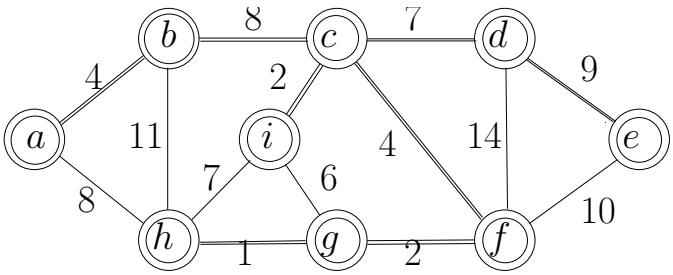


Figure 8.29: The final graph

The formal algorithm is set out below.

8.2.5 Priority Queues

Let S be a set of keys. A priority queue supports the following operations on S:

- Insert(S, x)
- Minimum(S) - returns minimum key
- ExtractMin(S) - returns and removes minimum key

Algorithm 29 MSTPrim

Input: A weighted graph G and a root vertex r
Output: A Minimum Spanning Tree

```
1: Q ← V[G]
2: for each u ∈ Q do
3:     u.key ← ∞
4: end for
5: r.key ← 0
6: r.parent ← NIL
7: while (Q ≠ ∅) do
8:     u ← extractMin(Q)
9:     for (each v ∈ Adj[u]) do
10:        if (v ∈ Q)&&(w(u, v) < v.key) then
11:            v.parent ← u
12:            v.key ← w(u, v)
13:        end if
14:    end for
15: end while
16: Output MST
17: return
```

A priority queue can be implemented using a heap. For the definition of a heap see the section on heapsort.

8.2.6 Single-source shortest path

Let G be a weighted, directed graph. A path in the graph from u to v is a sequence of edges that connects u to v. The weight of this path is the sum of the weights of its constituent edges.

Let $\delta(u, v)$ denote a minimum weight path from u to v and s be a source vertex of G. The single-source shortest path problem is this.

```
Compute a shortest path from s to each vertex v
```

Remark 5. A shortest path is a path of minimum weight.

Dijkstra's algorithm

We assume that the weights on the edges are non-negative. A notion that is important for this algorithm is that of relaxation.

Let $v.distance$ be an estimate of the length of a shortest-path from s to v. The relaxation procedure updates this estimate, based on the current shortest-path estimate of a vertex u to which v is adjacent, and the weight of the edge (u, v).

Figure 8.30 below shows the result of applying this relaxation procedure to an edge (u, v).

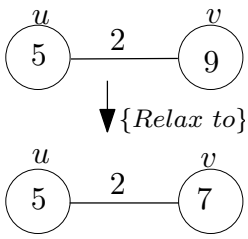


Figure 8.30: Relaxation changes v.distance from 9 to 7

Algorithm 30 The Relaxation routine

```
1: procedure relax(u, v, w)
2:     if (v.distance > u.distance + w(u, v)) then
3:         v.distance ← u.distance + w(u, v)
4:         v.parent ← u
5:     end if
6: end procedure
```

The main idea behind Dijkstra's algorithm is a to maintain a set of vertices, X, whose final shortest-path weights from s have already been determined. We first explain this algorithm on an example.

Figure 8.31: (a) Given Graph (b) Relaxing vertices adjacent to s

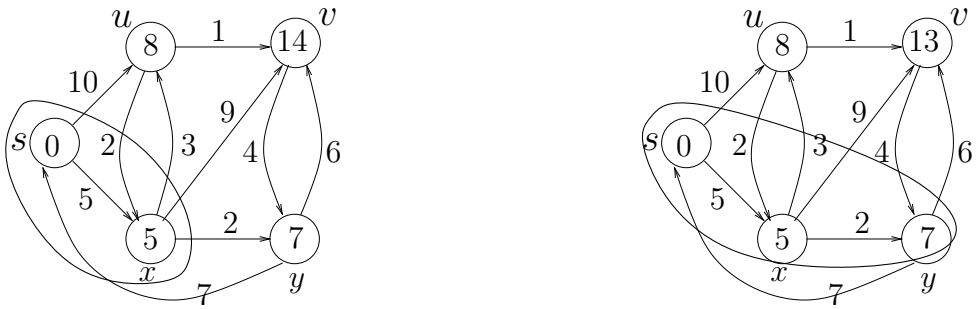


Figure 8.32: Relaxing vertices adjacent to (a) x and (b) y

Algorithm 31 The Initliaization routine

```
1: procedure InitializeSingleSource(G, s)
2:     for each vertex v ∈ V[G] − {s} do
3:         v.distance ← ∞
4:         v.parent ← nil
5:     end for
6: end procedure
```

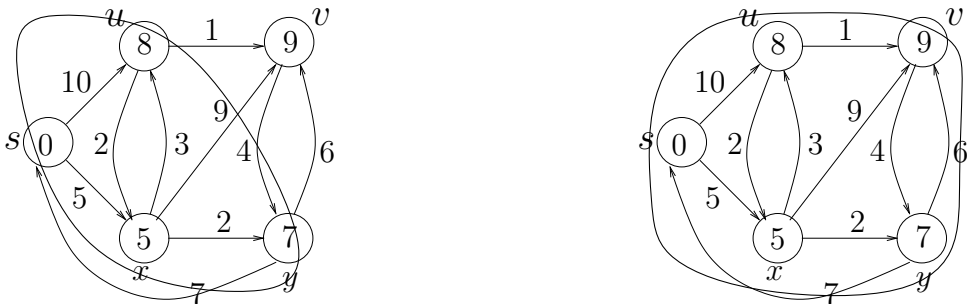


Figure 8.33: (a) Relaxing vertices adjacent to u; (b) final graph

The formal description of the algorithm is given below.

Algorithm 32 ShortesPathDijkstra

Input: A weighted graph G and a source vertex s
Output: Shortest paths to all vertices from s

```
1: InitializeSingleSource(G, s)
2: X ← 0
3: Q ← V[G]
4: while (Q ≠ ∅) do
5:     u ← extractMin(Q)
6:     X ← X ∪ {u}
7:     for (each vertex v ∈ Adj[u]) do
8:         Relax(u, v, w)
9:     end for
10: end while
11: return
```