PKFOKAM INSTITUTE OF EXCELLENCE

Course support

ALGORITHM ANALYSIS

Narcisse TALLA
PhD, Fotso Victor University Institute of Technology
University of Dschang, Cameroon
ntalla@gmail.com

August 2013

SUMMARY

CHAPTER 1. ALGORITHM ANALYSIS FUNDAMENTALS

1.1 PRELIMINARY

1.1.1 Essential reading

- Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 2;
- Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithmsin Java*. (John Wiley & Sons, Inc., 2005, fifth edition) [ISBN10 0-471-73884-0], [ISBN13 978-471-73884-8]. Chapter 4;
- Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3]. Chapter 5;
- Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 9;
- Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6]. Chapter 16.

1.1.2 Learning outcomes

This chapter is concerned with *algorithms and algorithm analysis*. Having read this chapter and consulted the relevant material, you should be able to:

- explain the term ***algorithm*** and the concept of the ***efficiency of an algorithm*** in terms of big-O notation;
- develop a perspective on the study of computer science beyond the learning of a particular programming language such as Java;
- describe commonly-used big-O categories.

1.2 PROBLEMS AND ALGORITHMS

In this section, we introduce the concept of algorithms and discuss the issues of fundamental analysis of algorithms.

A *problem* in this course is a general question to be answered, usually possessing one or more *parameters*. It can be specified by describing the form of parameters taken and the questions about the parameters. An *instance* of a problem is an assignment of values to the parameters. An *algorithm* is a clearly specified set of simple instructions to be followed to solve a problem. In other words, it is a step-by-step procedure for taking *any* instance of a problem and producing a correct answer for that instance.

1.2.1 Example: Finding the minimum*:*

Problem*: Given a non-empty set of numbers, what is the minimum element of the set?*

Instance: What is the minimum element of (2, 5, 8, 3) entered on a single line from the keyboard?

1.2.2 Solution

The following *Algorithm 1.1* is a solution for solving the problem of finding the minimum of a set of data that are input from the keyboard. The algorithm should give a correct answer for any set of data.

Algorithm 1.1 Minimum key

INPUT: nothing
OUTPUT: the minimum

```
1: read min;
2: while not eoln do
3:     read x;
4:     if x < min then
5:         min ← x
6: end while
7: print min;
```

1.3 PSEUDO-CODE FOR ALGORITHM DESCRIPTION

Using a natural language such as plain English to describe an algorithm is not impossible. However, one quickly realises that any human language is too rich to be concise or precise enough for the task. A common practice is to use so-called *pseudo-code* to describe algorithms. The syntax of any pseudo-code is similar to that of a high-level computer language such as Java. It is therefore much more convenient for an algorithm in pseudo-code to be translated into a computer program in some higher language than from a human language.

Thus once an algorithm for a problem is properly developed, it is a relatively easy matter to implement or translate it into a program in some computer language.

In this course, instead of giving a formal definition of the pseudo-code, we just borrow conventional syntax in Java, or the like. It is better to separate the algorithm design and implementation in the first and second stages respectively. During the first stage, i.e. the algorithm design stage, we ignore the implementation details and focus on problem-solving techniques and algorithmic issues.

1.4 EFFICIENCY

In this module, our goal is not only to develop a working algorithm, but also an efficient algorithm for a given problem.

The speed of hardware computation of basic operations has been improved dramatically, but efficiency matters more than ever today. This is because our ambition for computer applications has grown with computer power. Many areas demand a great increase in speed of computation. Examples include the simulation of continuous systems, high resolution graphics, and the interpretation of physical data, medical applications, and information systems.

On the other hand (and more importantly), an algorithm may be so inefficient that, even with computation speed vastly increased, it would not be possible to obtain a result within a useful period of time. The time that many algorithms take to execute is a non-linear function of the input size. This can reduce their ability to benefit from the increase in speed when the input size is large.

1.4.1 Example

A particular sorting algorithm takes n^2 comparisons to sort n numbers. Suppose that computing speed increases by a factor of 100. In the time that it was required to take to

execute the n^2 comparisons, it is now possible to do $100n^2 = (10n)^2$ comparisons. Unfortunately, with 100 times speed-up, only 10 times as many numbers can be sorted as before.

1.5 MEASURES OF PERFORMANCE

Naturally, the efficiency of an algorithm is estimated by its performance. The performance of an algorithm can be measured by the ***time*** and *the* ***space*** required in order to fulfil a task. The time and space requirement of an algorithm is called the ***computational complexity*** of the algorithm. The greater the amount of the time and space required, the more complex is the algorithm.

The ***time complexity*** of an algorithm is, loosely speaking, an imaginary *execution time* of the algorithm. The execution time can be measured by the number of some *characteristic operations* performed by the algorithm in order to transform the input data to the results.

Note that we do *not* measure the time complexity by the running time of a program. This means that we do not use the common time units such as *second, minute* and *hour*. The unit of the time complexity, strictly speaking, should be the ***number of execution steps***, although we often do not use any unit.

An algorithm consists of a set of ordered instructions and the **time complexity**, that is, the number of execution steps in an algorithm, is irrelevant to the real time.

Note that our main interest here is in an algorithm instead of a program. *A program is an implementation of an algorithm*. The execution time of a program depends on the implementation including not only the operating system, but also the speed of the computer itself. The same program may run faster on a computer with a faster CPU, but the same algorithm should perform the same number of algorithmic steps to accomplish a task.

Normally we are concerned with the ***time complexity*** rather than ***space complexity*** of an algorithm. The reasons are that firstly, it becomes easier and cheaper to obtain space. Secondly, techniques to achieve space efficiency by spending more time are available.

In what follows, we use **'complexity'** to mean the *time complexity* if not otherwise indicated.

1.5.1 Observation

- The complexity of an algorithm normally depends on the size of input.
- The number of operations may depend on a particular input.

1.5.2 Solution

- For different ***sets of input data***, we analyse the performance of an algorithm in the ***worst*** case or in the ***average case***.
- For different algorithms, we focus on the ***growth rate*** of the time taken by the algorithms as the input size increases.

The time complexity of an algorithm can be expressed by a function of input size: $T(n)$. We are normally interested in the behaviour of $T(n)$ as n grows large.

1.6 ALGORITHM ANALYSIS

With the measures of performance introduced earlier, it is possible to conduct an analysis and estimate the cost of an algorithm.

Many reasons can be given to explain why we need algorithm analysis. One main reason is that there are usually several algorithmic ideas for a problem, and we would like to eliminate the inefficient algorithms *early*. By early, we mean that we would like to compute or estimate the computational complexity of any two algorithms *before* actually implementing (coding) them into programs.

Secondly, algorithms may behave differently for different input sizes, and we would like to estimate their computational complexity for *large* inputs. For some problems we simply have not found an efficient algorithm yet, but we may find those algorithms feasible and useful for input within some limited range.

Furthermore, the ability to do an analysis usually provides insight into designing efficient algorithms. The analysis also could pinpoint the bottlenecks which should be taken care of during coding.

In what follows, we shall introduce some fundamental methods for algorithm analysis. Before moving on, we have to first agree a model of 'normal' computer.

1.7 MODEL OF COMPUTATION

We adopt a so-called *'random access machine'* (RAM) model. In this model, certain hardware constraints for a real computer are ignored in order to focus on algorithmic issues. Routine operations at machine level, such as fetching instructions or data from the memory are also ignored for the same reason.

We assume that our computer has the following convenient properties:

1. It has a single processor (CPU) and runs our pseudo-code algorithms in a sequential manner. A pseudo-code algorithm consists of an ordered sequence of instructions in pseudo-code (Appendix C). The instructions in each algorithm are executed one after another in the given order.
2. It takes exactly ***one time unit*** to execute a standard instruction (in pseudo code) for operations such *as addition, subtraction, multiplication, division, comparison, assignment and conditional control*. No complex operations, such as sorting and matrix inversion, can be done in one time unit.
3. The storage for integers is of a fixed size, for example, 32 bits.
4. It has an infinitely large memory.[1]

The assumptions are necessary because our analysis result depends on the model. For example, assigning 100 data into an array would require 100 unit times in our model. The same task can be done in one time unit, however, in a parallel computation model such as a 'parallel random-access machine' (PRAM), in which the 100 memory cells can be accessed simultaneously. Certain hardware details are ignorable from an algorithmic point of view. For example, standard operations such as *addition, subtraction, multiplication, division, comparison, assignment* and *conditional control* would require

[1] So there is no need to consider any overflow issues. This assumption is based on the fact that memory techniques has developed to allow the logical memory to be of a larger size than its physical size.

different amount of time to run on a real computer. The storage of real computers is limited and the memory for integers and reals may be of a different size. However, taking the difference made by these hardware details into consideration gives little impact on the result in comparison of different algorithms, because these standard operations are required for almost every algorithm. Taking too many details into consideration can make an analysis too complicated to be carried out.

1.7.1 Counting steps

Given two algorithms A_1 and A_2, which one is more efficient? In other words, which one has lower computational complexity? To answer this question, one way is to simply count the *execution* steps of each algorithm and compare the numbers of the steps of the two.

1.7.2 Example

Problem: *Compute* $\sum_{k=1}^{n} k$ *, where* n *is an integer.*

Algorithm 1.2 Sum1(n)

INPUT: n
OUTPUT: sum
1: sum ← 0;
2: **for** $k \leftarrow 1, k \le n, k \leftarrow k + 1$ **do**
3: $sum \leftarrow sum + k$;
4: **end for**
5: print *sum*.

From the fact $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, *we have*

Algorithm 1.3 Sum2(n)

INPUT: n
OUTPUT: sum
1: print $n(n + 1)/2$

1.7.3 Implementation

These algorithms can be implemented to the following Java methods:

```
int sum1( int n) {
  int sum = 0;
  for (int k=1; k<=n; k++) {
    sum = sum + k;  }
  return sum;
}

int sum2( int n) {
  int sum;
  return (n*(n+1)/2);
}
```

We look at the *time* complexity by counting the steps taken in execution. Let any assignment, arithmetic computation +, −, ∗ , /, read, print be all counted as *one* step. So for *Algorithm 1.2*, it takes $1 + n \times 1 + 1 = n + 2$ steps; and for *Algorithm 1. 3*, it takes 1 + 1 + 1 = 3 steps to execute. Obviously, *Algorithm 1.3* is more efficient in terms of *execution time*.

How about the space efficiency? Let a simple variable require one unit of storage. *Algorithm 1.2* needs three units since it involves three variables *sum, k* and *n*, and *Algorithm 1.3* only needs one unit since it involves only one variable n. We can therefore conclude that *Algorithm 1.3* is more efficient in terms of *storage*, too.

In fact, *Algorithm 1.3* has an important advantage; that is, it takes *constant* time to execute no matter how large n is. This means that it takes the same amount of time to run no matter how many such numbers need to be added up.

1.7.4 Characteristic operations

A complexity analysis gives an *estimate* of the resources consumed by an algorithm. The relationship between the amount of time and space allows us to focus on the time efficiency only. The time complexity is, after all, $T(n)$, a *function* of the input size n. The more data, the longer it takes to run the algorithm. The task of counting steps can be made easier if a reasonable measure of the *input* size of some major operations can be established and then only those operations relevant to the input size need to be considered.

1.7.5 Example

For each algorithm below, we choose a relevant characteristic operation:

1. An algorithm searching an element x in a list of names:

   Choose the comparison of x with an entry in the list;

2. An algorithm multiplying two matrices with real entries:

   Choose the multiplication of two real numbers;

3. An algorithm sorting a list of numbers:

   Choose the comparison of two list entries;

4. An algorithm to traverse a binary tree:

   Choose the visit of a tree node.

The characteristic operations could be some very expensive operations compared to others, or they might be of some theoretical interest. It allows a good sense of flexibility to choose these fundamental operations as a measure of time complexity.

1.8 ASYMPTOTIC BEHAVIOUR

We are often interested in the rate of growth of the time required for an algorithm when the input size gets larger. So the lower order terms of the time complexity $T(n)$ could be ignored, where n is a positive integer. In other words, we only need to master the asymptotic behaviour of $T(n)$. Here the term *asymptotic* means approximate in a specific way.

1.8.1 Example

Suppose that the time complexity of an algorithm is $T(n)=\frac{1+n^2}{n}$ where n is the input size to the algorithm. It is easy to see some asymptotic behaviours of T(n) such as, T(n) ~ n when n → ∞[2], because

$$T(n)=\frac{1+n^2}{n}=\frac{\frac{1}{n}+n}{1}\ \square\ \frac{0+n}{1}=n$$

similarly, $T(n)\sim\frac{1}{n}$ *when* $n\to 0$

Here n and $\frac{1}{n}$ are both simpler than $T(n)$ and it is easier to handle their behaviour in an analysis. We say that n and $\frac{1}{n}$ are *asymptotic behaviours* of $T(n)$ when $n \to \infty$ and respectively when $n \to 0$.

1.8.2 Big *O* notations

In general, the asymptotic behaviour of functions can be described by the so-called "big-O" notations, often consisting of four members $O()$, $\Omega()$, $\Theta()$ and $o()$. They are called "big-oh", "omega", "theta" and "small-oh" respectively.

Let g be a function of n. Each of $O(g)$, $\Omega(g)$, $\Theta(g)$ and $o(g)$ defines a set of functions related to g.

- $O(g(n))$ is a set of functions that grow *at most* as fast as g when $n \to \infty$.
- $\Omega(g(n))$ is a set of functions that grow *at least* as fast as g when $n \to \infty$.
- $\Theta(g(n))$ is a set of functions that have *the same* growth rate as g when $n \to \infty$.
- $o(g(n))$ is a set of functions that grow *slower* than g when $n \to \infty$.

Conventionally, we use $T(n) = O(g(n))$ to mean $T(n)\in O(g(n))$. We define $T(n)=O(g(n))$ if there are positive constants c and n_0 such that $T(n)\le cg(n)$ when $n \ge n_0$.

Similarly, $T(n)=\Omega(g(n))$ if there are positive constants c and n_0 such that $T(n)\ge cg(n)$ when $n \ge n_0$.

- $T(n)=\Theta(g(n))$ if and only if $T(n)=O(g(n))$ and $T(n)=\Omega(g(n))$.
- $T(n)=o(g(n))$ if and only if $T(n)=O(g(n))$ and $T(n)\neq\Theta(g(n))$.

1.8.3 Example

Let $T_A(n)$ be the time complexity of an algorithm A, where n is the input size of the algorithm. Suppose $T_{A_1}=\frac{n^2}{2}$ and $T_{A_2}=7n$.

[2] Here symbol '~'means'will be approaching'; and'→' means'goes to'. So'$T(n) \sim n$ when $n \to \infty$' reads'The values $T(n)$ will be approaching n when n grows to infinitely large'.

Illustrating definitions, we see that $7n$ is $O(n^2)$ but that $7n \neq \Theta\left(\frac{n^2}{2}\right)$ because $\frac{n_2}{2} \neq \Theta(7n)$.

1.8.4 Comparing orders of two functions

When comparing two functions in terms of *order*, it is often convenient to take the alternative definitions: Let $\lim_{n\to\infty} \frac{T(n)}{g(n)} = L$. The limit can have four possible values:

- If L = 0 then $T(n) = o(g(n))$;
- If L = a≠0 then $T(n) = \Theta(g(n))$;
- If L=∞ then $g(n) = o(T(n))$;
- If L oscillates, then there is no relation (but this will not happen in our context).

1.8.4.1 Example

Given $T_{A_1}(n) = 1000n$ and $T_{A_2}(n) = n^3$, which function grows faster when $n \to \infty$? What is the relationship between the two functions?

1.8.4.2 Solution

$\lim_{n\to\infty} \frac{1000n}{n^3} = 0$. Therefore, $T_{A_1}(n)$ *is* $o(n^3)$, which means $T_{A_1}(n)$ grows strictly slower than $T_{A_2}(n)$.

1.9 THE WORST AND AVERAGE CASES

The behaviour of an algorithm usually depends not only on the size of the input, but also on the input itself. Look at the following Algorithm 1.4 which determines whether integer x is an element of array Y[0..n —1], where n is a non-negative integer.

1.9.1 Example

Algorithm 1.4 boolean foundFirstX(int x, int Y[])

```
1: i ← 0
2: boolean found ← false
3: while (not found) && (i < length(Y)) do
4:     found ← (x = Y[i])
5:     i ← i + 1
6: end while
7: return found
```

The time complexity of foundFirstX(x,Y) depends on the value of *x* and on the contents of the array *Y*. For example, if *Y*[0] = x, that is, the first element in array *Y* equals *x*, then the while loop (step 3 – 6 in the algorithm) will only be executed once, and the number of execution steps is only 1 + 1 + 1 x (1 + 1 + 1 + 1) + 1 = 7, including step 1, step 2, 1 × (step 3, 4, 5, 6), and step 7. If *x* equals the k^{th} element of *Y*, the number of execution steps becomes 1+ 1 + *k* × (1 + 1 + 1 + 1) + 1 = 3 + 4*k*, where $k \leq length(Y)$, including step 1,

step 2, k × (step 3, 4, 5, 6), and step 7. Each of these different situations is called a *case*. In each case, the algorithm gives a different performance.

We therefore need to consider the behaviour of the algorithm for two special cases, namely the *worst* case and the *average* case.[3]

In terms of time complexity, the worst case is the situation where the algorithm would take the longest time. The average case is the case where the average behaviour is estimated after every instance of the problem has been taken into consideration. We define two functions of n, the input size, for the two cases respectively.

In general, an algorithm may accept k different instances of size n. Let $T_i(n)$ be the time complexity of the algorithm when given the i^{th} instance, for $1 \le i \le k$. Let p_i be the probability that this instance occurs.

Then the time complexity for:

- The worst case is $W(n) = \max_{1 \le i \le k} T_i(n)$
- The average case[4] is $A(n) = \sum_{i=1}^{k} p_i T_i(n)$

In words, $W(n)$ is the maximum number of characteristic operations performed by the algorithm on any input of size n. $A(n)$ gives the behaviour of the algorithm on average for different instances. Clearly, $A(n) \le W(n)$.

The worst case analysis could help to provide an estimate for a time limit for a particular implementation of an algorithm. It is particularly useful in real time applications. The average case analysis is more meaningful in providing an overall picture because it computes the number of steps performed for each possible input instance of size n and then takes the (probability-weighted) average.

In this course, we shall consider only the worst case analysis if not specified otherwise. The result of an algorithm analysis can sometimes turn out to be unsatisfactory or extremely difficult to achieve. In these cases, an empirical[5] approach should be considered.

1.9.2 Implementation

Algorithm 1.4 can be realised in the following Java method:

```
boolean foundFirstX( int x, int Y[]) {
  boolean found = false;
  int i=0;
  while ( ( !found) && ( i< Y.length) ) {
    found = ( x==Y [i] ) ;
      i++;
  }
  return found; }
```

[3] Although it is desirable, the best case is not very interesting, for it does not help much with a budget. In contrast, the ***worst*** case prepares us for the most costly situation, and the ***average*** case tells us what to normally expect.

[4] This can be extremely complex sometimes.

[5] An empirical approach is a means of investigating the efficiency of an algorithm by experiments.

It is easy to modify the program slightly in order to compute the *actual* number of Java statements executed. In the example below, we define a variable count before (or after) each statement, and display the value of count at a few places of the program.

```
boolean foundFirstX( int x, int Y[]) {
  int count = 1;
  boolean found = false;
  count ++;
  int i=0;
  System.out.println("Step 1--2: "+count);
  count ++;
  while ( ( !found) && ( i< Y.length) ) {
    count ++;
    found = ( x==Y [i] ) ;
    count ++;
    i++;
    count ++; // for while
   }
   System.out.println("Step 1--6: "+count);
   count ++;
   System.out.println("All steps: "+count);
   return found;
}
```

You can conduct an experiment in which different integer arrays Y[] are input and the different number of execution steps are displayed on your screen. For example, if you call the methods with **"int** A [] = 9, 4, 5, 7, 1, 2;**"**, you would see

```
...
Step 1 and 2: 2
Step 1--6: 21
All steps: 22
...
```

If with "int A [] = 2, 4, 5, 7, 1, 2;", you would get

```
...
Step 1--2: 2
Step 1--6: 6
All steps: 7
...
```

This can be implemented in two classes as in the following example:

1.9.3 Typical growth rates

Some functions are commonly seen with typical growth rates in the algorithm analysis. We list some common ones here.

Note All logarithms in this subject guide are of base 2 if not stated otherwise.

Functions	Name	Functions	Name
c	constant	$n \log n$	n-log-n
$\log n$	logarithmic	n^2	quadratic
$\log^2 n$	log-squared	n^3	cubic

n	linear	2^n	exponential

1.9.4 Verification of an analysis

It is important to conduct an algorithm analysis, before any implementation, to avoid any unnecessary expensive labour. It is even more important to make sure that the analysis result is correct. Hence verification of an analysis is necessary and highly recommended, although it sometimes turns out to be difficult.

For example, we can always:

- check if the empirical running time matches the running time predicted by the analysis;
- for a range of n, compute the value $T(n)/f(n)$ where $f(n)$ is the analysis result and $T(n)$ is the empirically observed running time. This should be ideally a constant as n varies.

1.9.5 Activity

1.9.5.1 Time Complexity

1. Discuss briefly the time complexity in the *worst* case for the algorithm below. Indicate the input, output of the algorithm and the main comparison you have counted.

Algorithm 1.5 insertionsort(int array[0..n-1])

```
1 : for i←1, i ≤ n − 1, i + + do
2 : current ← array[i]
3 : position ← i − 1
4: while position ≥ 1 and current < array[position] do
5 :   array[position + 1] ← array[position]
6 :   position ← position − 1
7 : end while
8 : array[position + 1] ← current
9 : end for
```

2. Consider the big O behaviour of the code below in terms of N. Discuss briefly its time complexity.

```
1 : k ← 1;
2 : repeat;
3 : k ← 2 × k;
4 : until k ≥ N;
```

CHAPTER 2. ABSTRACT DATA TYPES I: LISTS AND HASHING TABLES

2.1 PRELIMINARY

2.1.1 Essential reading

- Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2001, fourth edition) [ISBN 0-201-35744-5]. Chapter 3, 5, 9.2
- Duane A. Bailey *Java Structures: Data Structures in Java for the principled programmer*. (McGraw-Hill Companies, Inc. 1999, McGraw-Hill International editions) [ISBN 0-13-489428-6]. Chapter 13
- Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 5
- Sara Baase and Allen Van Gelder *Computer Algorithms: Introduction toDesign and Analysis*. (Addison-Wesley Publishing Company, 2000) third edition [ISBN 0-201-61244-5]. Chapter 2

2.1.2 Learning outcomes

We introduce basic list abstract data structures, including linked lists, stacks, queues and hash tables in this chapter. Having read this chapter and consulted the relevant material, you should be able to:

- distinguish the concepts of Abstract Data Types and Data Structures;
- choose appropriate list data structures for a given problem;
- Calculate the complexities of various algorithms;
- Choose appropriate algorithm for a related problem.

In this chapter, we shall introduce the concepts of *Abstract Data Types* (ADTs) and three fundamental *Data Structures*.

Any data to be processed by a computer program need to be organised in specific logical structures for primarily retrieval purposes. The logical relationship among the data is called *data structure*. A data structure can be linear, which means the data are related each one to another. It can also be hierarchical, in which one datum is conceptually superior to another.

Certain routine operations must be allowed to be performed on the data items in each data structure. For example, data may be added into or removed from a data structure to maintain a certain structure. A collection of these operations and the relationships between data items is called an *abstract data type* (ADT in short).

We distinguish the two concepts in this module for convenience of discussion even though that the terms *abstract data structure* and *abstract data type* are often used interchangeably in different environments. The term *data structure* is more appropriate when the logical relations are being studied among the data items in storage. The term *abstract data type* is more convenient when standard operations on a storage structure are being viewed as the behaviours of an object to be processed in a computer program.

The word *abstract* is used to reflect the fact that the data, the basic operations and the relations between data are being dealt with independently of any implementation.

An *implementation* of a data structure or an abstract data type consists of the arrangement of storage structures for data items and the programs for fundamental operations on the data. In this way, algorithmic ideas for the operations are represented in a specific computer language and can be realised on a conventional computer.

The term *data abstraction* is frequently used in *Software Engineering*. Its concept involves a so-called *top-down* approach to software development. Since the definition of a data structure is separated from its implementation, it is possible for people to study and use a data structure without being concerned about the details of its implementation.

Conventional high level computer languages provide abstract data types: for example, *integer, real, char*, *arrays*, *vectors* and *linked lists* can be used in Java without any further implementation requirement. We shall, in order to understand their uses, study closely these abstract data types and define our own versions of ADTs for these.

2.2 FROM ABSTRACTION TO IMPLEMENTATION

An Abstract Data Type (ADT) is a collection of *data objects* with a defined set of properties and operations for processing the objects. The abstract data type is *abstract* since there is no information on how each datum is represented and how the operations are implemented in the data type.

2.2.1 The string abstract data type

A string in our context is defined as a finite sequence of characters excluding a 'null' character. The word 'sequence' indicates a relationship between characters in a linear fashion. The order of the characters is important.

2.2.2 The matrix abstract data type

Matrices are two dimensional arrays and have following characteristics:

1. a collection of data of the same type arranged in a two dimensional table consisting of rows and columns;
2. the rows and columns themselves are each indexed by a separate contiguous range of some ordinal data type, e.g. integers.

Matrices are very useful in applications. For example, the logical coordinate system for a graphic display.

2.2.3 The keyed list abstract data type

Keyed lists are especially useful for information retrieval. In a database for example, each entry contains at least a field that is used as a unique identifier for searching purpose. Such a field is called a *key field*. A keyed list contains usually a collection of records with a comparable key field of a unique value for each record.

2.3 DATA STRUCTURES AND SOFTWARE PERFORMANCE

Data structure is one important factor that affects the time complexity of an algorithm. From experience, we know that the speed of many operations on data depends on how

the data are stored and how they are accessed. For example, an array, one simple type that is normally built-in for most high level computer language, represents a sequential relationship among the data in the structure. In an array, each datum can be accessed instantly by its index in 0(1) time.

i	1 2 3 4 5
E [i]	3 4 6 2 5

Another commonly used data structure is so called a *linked list*. In a linked list structure, each datum consists of at least two fields: *data* and *next*, where *data* is of a simple data type and the next is an address or index pointing to the datum after the current one. The relationship among data is *one-after-another*. In a linked list, a datum can only be accessed by following the link from its predecessor. It would naturally take longer to reach the last element in a long list than the one in front.

2.4 ARRAYS

Array is a powerful data structure with a constant access time. The data items in an array are placed one after another and each item is associated with an index (called an *address*). An array is a powerful way to store a large number of data of the same type. It is often described as *static* data structures because the number of data items has to be specified at the beginning. The size of the array cannot be changed once the array is defined. Static data structures are not convenient for applications where a data set needs to be updated frequently.

2.4.1 Applications

2.4.1.1 Example

Suppose we want to keep the marks of 5 different subjects for 10 students. Compute the total mark of 5 subjects for a student, and the average mark of the students for a subject.

Let i be the row number indexing student, and j the column number indexing subject in the matrix below

(j) / (i)	0	1	2	3	4
0	23	30	40	90	20
1	45	55	11	40	30
2	98	99	89	94	91
3					
4					
5	.	.	.	.	.
6	.	.	.	.	.
7	.	.	.	.	.
8					
9	44	55	12	48	39

Note this is an instance of a simple record-keeping problem. Our algorithm and the program to be developed should be able to compute the marks of any numbers subjects for any number of students within a pre-defined range.

We define a two dimensional array called marks[], where entry marks [i, j] represents the student *i's* marks for subject *j*.

The operations required are therefore:

1. Input the marks *getMarkTable*, i.e. store the marks of each subject for each student in the array `mark[i,j];`
2. Compute the total marks total (j), i.e. add `mark[i,0...4]` for each *i* to `mark [i,10]` , where *i* = 0,..., 9.
3. Compute the average marks average (i), i.e. add `mark [0... 9, j]` for each *j*, *j* = 0, … , 4, and divided by 5, the number of subjects.
4. Print all the marks in the array:

These operations can be implemented easily in a high level computer language such as Java. We give an example of simple implementation in Java below.

1. Input the marks getMarkTable:

   The marks can be

   (a) input into the array marks from a keyboard;

   (b) read in from a text file;

   (c) randomly generated some marks.

2. Compute the total marks total(i) of a student *i* for all the subjects:

```
void totalForStudent(int i) {
  double total = 0;
  for (int j=0; j<marks[i].length; j++){
    total=total+marks[i][j];
  }
  System.out.println("The total marks for student"+i+":"+total);
}
```

3. Compute the average marks average(j) of all the students for a subject *j*:

```
void averageForSubject(int j){
  double average = 0;
  int n = marks[j].length;
  for (int i=0; i<n; i++){
  average=average+marks[i][j];
  }
  if (n!=0){
    average=average/n;
      System.out.println("The average marks for subject"+j+": "+average);
  }
```

4. Print all the marks in the array:

```
 // 1st index i represents the student number,
// 2nd index j represents the subject number.
void printMarkTable (int marks.length; i++){
```

```
    for (int i=0; i<marks.length; i++){
      for (int j=0; j<marks[i].length; j++){
          system.out.printf(marks[i][j]+",");
        }
        system.out.println();
    }
}
```

2.4.2 Exercise

- Write the previous four algorithms using different approaches;
- Calculate the complexities of the previous algorithms;
- Compare using big-O notations the proposed algorithms with the previous ones.

2.4.3 Two and multi-dimensional arrays

An array in which each data item can be accessed by a pair of indices is called a *two* dimensional array. Similarly, an array is called an *n*-dimensional array if n indices are required for access.

2.4.4 Activity

1. Define an array to store student marks. Suppose each student has only one mark and there are at most one thousand students.
2. Following the above, write a method that displays the marks of the students.
3. Write a boolean type method which takes (1) an array of integers and (2) the size of the array as parameters and determines if all the integers in the array are between 10 and 50 inclusive.
4. Calculate the time complexity of each algorithm.

2.5 LISTS

2.5.1 References, links or pointers

The link is called a reference in Java or a *pointer* in many other high level programming languages. A pointer is defined to be a *variable* that gives the location of some other variable, typically of an object containing data that we wish to use. In other words, a pointer is a variable that stores an address of some other variable, typically of object type.

2.5.2 Linked lists

A *linked list* (or list) is an abstract data structure consisting of a collection of data objects in which each item contains not only a *data* field but also at least one *link* field to point to the following object.

2.5.2.1 Example

(12, 10, 3, 4) and ('A', 'B', 'C', 'D') are two simple lists.

The first one is a list of integers and the second one is a list of characters. The following figures show their data structures.

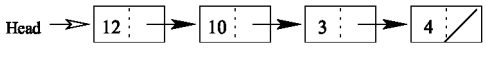


Figure: A list of integers

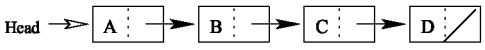


Figure: A list of characters

Two nodes are special in such a linked list, one is the first node, and the other is the last node. The first node of a linked list is called the *head* of the list, which cannot be accessed following the link of any list nodes. The head of a list has to be initialised. The last node of a linked list points to nobody and has a null value for its *link* field.

Each data item in a list can either be of a simple type as in the example above or of any defined type, e.g. a list. A list, therefore, can also represent a hierarchical structure, namely, a *list of lists*:

2.5.2.2 Example: list of lists

((((A,B),C,D),E,F,G),(H,I,J),((K,L,M),N)) is another list of lists that represents a hierarchical structure as in the following figure.

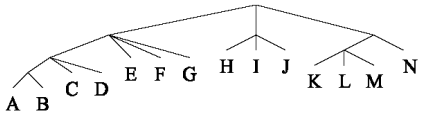


Figure A list of lists

2.5.3 Add one node to a linked list

There are, given the address of a current node, two possibilities to insert a node in an existing linked list. One is to add the new node before the current node and the other is to insert it after the current node.

2.5.3.1 Insertion after the current node.

Suppose that *p* points to the current node, and *newNode* points to a new node to be inserted. There are 2 fields for each node, namely data and next. This can be seen from the following figure and algorithm.

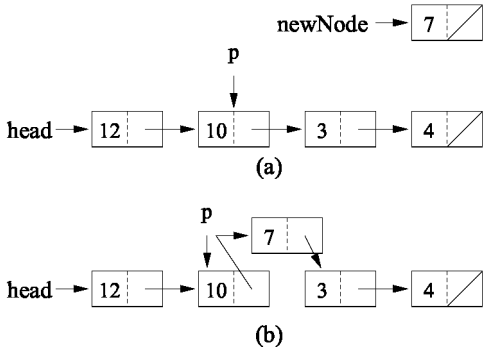


Figure. Add one node after the current node

Algorithm 2.3 addOneNodeAfter(Node p, newNode)

1: *newNode.next* ← *p.next*

2: *p.next* ← *newNode*

2.5.3.2 Insertion before the current node

There are two cases depending on whether the current node is, or not, the head of the list.

1. **The current node is not the head of the list**

This can be seen from the following figure and Algorithm. Adding one node before a non-head node *p* is equivalent to adding the node after the node before *p*.

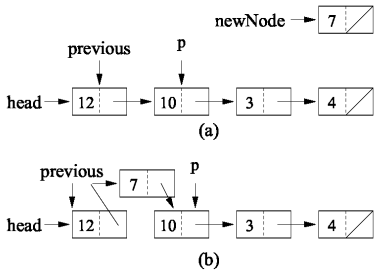


Figure: Add one node before the non-head current node

Algorithm 2.4 addOneNodeBefore(Node *previous, p, newNode*)

1: *newNode.next* ← *p*

2: *previous.next* ← *newNode*

2. The current node is the head of the list

This can be seen from the following figure and algorithm.

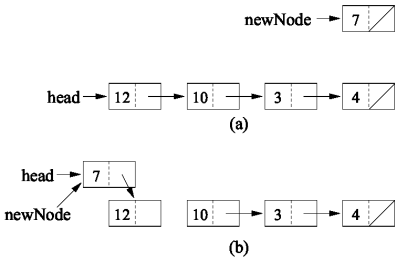


Figure Add one node before head

Algorithm 2.5 addOneNodeBeforeHead(Node *newNode*)

1: *newNode.next* ← *head*

2: *head* ← *newNode*

2.5.4 Delete one node from a linked list

Suppose that *p* points to the current node. The deletion process can be seen from the following figure, and algorithm.

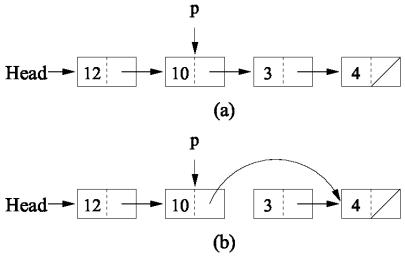


Figure: Delete one node after the current node

Algorithm 2.6 deleteOneNodeAfter(Node *p*)

1: *p.next* ← *p.next.next*

2.5.5 Construct a list

Constructing a list is a repeated process of adding one node into a linked list with initially an empty list. There are two ways to construct a list. One is to construct a linked list in a backwards fashion, i.e. to construct the last node of the list and then each time add one node at the head of the list. The other is to construct the first node of the list first and then append one node each time at the tail of the list.

There is no difference between the two approaches in time complexity if a pointer tail is maintained to address the last node for the appending approach. It takes $O(n)$ amount of time for both approaches to construct a list of n nodes, where n is the number of nodes in the list.

2.5.6 Comparison with arrays

Compared with arrays, linked lists as dynamic data structures have advantages in storage administration, and being flexible for handling frequently updating operations such as *insertion* or *deletion* of elements. The disadvantage of a linked list is the loss of the constant time in accessing a datum in the list. It would take 1 step to access the k^{th} element in an array but it would take k steps from the head of the list before accessing the k^{th} node.

2.5.7 Activity

1. Write an algorithm that displays all the items in a list.
2. Write an algorithm that checks if a list is empty.
3. Write an algorithm that counts the number of nodes in a list.
4. Calculate the complexity of each algorithm.

2.6 STACKS AND QUEUES

2.6.1 Definition and principle

A *stack* is a very useful data structure. Many problems in the real world can be easily modelled by a stack. Although the work done by a stack can also be done by a list, it would obscure its essential nature without distinguishing it from a list. Similarly to the special pointer variable *head* for a linked list, *top* is a pointer variable that always points to the top of a stack.

A queue is a list that restricts insertions to one end called *rear* and deletions from the other end called *front* of a list structure. It is a data structure that follows the *first in first*

out (FIFO[6]) principle. Objects are added to the rear of the queue, and are removed from the front of the queue.

2.6.2 Activity

1. Write an algorithm that outputs a sequence of strings in the *reverse* order of entry using a *stack* data type. The program keeps taking lines of strings from the keyboard until the string "end-of-input" is input.
2. Modify the Reverse program in question 1 so it outputs lines of strings in the order of entry using a pointer implementation of *queues*. Again, the program keeps taking lines of strings from the keyboard until "end-of-input" is input.
3. Calculate the complexities of the two previous algorithms.
4. Compare the two complexities using the big-O notations.

2.7 HASHING

Hashing is another technique of storing and retrieving data. The data structure involved is essentially a one-dimensional array called *hash table*. Each datum depends on the value of its own key, and is calculated by a *hash function*. We denote the hash function by $h(k)$, where k is the key of a datum.

Given a hash function h and a datum key k, the value of $h(k)$ is called a *hash code*. It is used as the address of datum k in the hash table. A hash code can be calculated easily, given the hash function and the key.

2.7.1 Example: Constructing a hash table

Assume that a hash function $h(k) = k \bmod 11$ *is used and the hash table is empty initially. Show the content of the hash table after inserting the data (7, 31, 159, 189, 23, 6).*

Solution

We first compute the hash code for each datum using the hash function. We have:

- h(7)= $7 mod 11 = 7$;
- h(31)= $31 \bmod 11=9$;
- h(159) = $159 \bmod 11 = 5$;
- h(189) = $189 \bmod 11 = 2$;
- h(23) = $23 \bmod 11 = 1$;
- h(6) = $6 \bmod 11 = 6$.

Since the hash codes are the indices (i) for the corresponding data in the hashtable, the content of the hashtable (H) is, therefore,

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
H		*23*	*189*			*159*	*6*	*7*		*31*		

Once we have the hashtable built up, searching for a datum is straightforward. Given a key k, all it is required is to compare the k to Hashtable $[h(k)]$.

[6] pronounced 'fie-foe'.

2.7.2 Example: Searching an element in a hash table

Suppose that we search for key *k = 23*. We check the location and find *h(23)=23mod11=1*, and compare the key k and the Hashtable[1]. Since *k = Hashtable[1] = 23*, we know that *23* is in the hashtable and can return its location *1*.

Suppose now that we search for key *k = 50*. We check the location and find *h(50)=50mod11 = 6*, and compare the key *k* and the *Hashtable[6]*. Since *k = 50*, but *Hashtable[6] = 6*, there is no match, this means that 50 is not in the hashtable[7].

2.7.3 Observation

As we see from the previous example:

1. Each element in the table can be stored and accessed in potentially O(1) time.
2. The Hashing technique can map a large key space of the data into a relatively small range of integers which are used as the indices of the hash table. In the example, the key space [6 – 189] is mapped to [0 – 11].
3. Hashing can be very efficient in terms of both the time and the space complexity.

2.7.4 Collision

However, hashing has a *collision* problem. A collision happens when a datum is attempted to be stored in an already occupied cell. Collisions are unavoidable in general because it is possible for two keys to have an identical hash code.

2.7.5 Collision resolving

The collisions can be resolved in various ways. Of course, the hash function can be adjusted but this is not easy. The cause of the collisions is due to the attempt to map a large key space to a limited hashtable range. A natural solution is hence to re-allocate the collided data elsewhere. We look at some simple approaches here, namely,

1. closed address hashing
2. open address hashing
   - linear probing
   - double hashing.

2.7.5.1 Closed address hashing

The *closed address hashing* does not consume any extra addresses of the hashtable. It is one easy solution for collision. The method chains the collided data together, using an array of linked lists.

2.7.5.2 Open address hashing

The *open address hashing* increases the number of addresses of the hashtable. This approach stores all the elements right in the hashtable array without using any extra linked lists. The collided data need to be reallocated to other available cells. Thus additional cells may be required and the original hashtable may be extended.

[7] Note: This is true only if there is no collision (Section 2.10.1).

It is possible that a sequence of alternative hashing addresses will be allocated before a free hash cell is found. If a hash cell is occupied, a new hash code will be generated. If the new location is occupied again, a new hash code will be generated again. The process of computing the alternative addresses is called *rehashing* or *probing*.

- **Linear probing**

Linear probing simply allocates the collided datum to the next available location. For example, if the first hash location h1 is occupied, then $h_1 + 1$ is offered if it is empty, otherwise, the next location $h_1 + 2$ is offered, and so on.[8]

This is equivalent to using a similar hash function for rehashing when there is a collision: ***rh(k) = (k + 1) mod h.***

2.7.5.3 Double hashing

This is to apply an alternative hash function for probing. If the first hashing is unsuccessful, the second hash function can be used to resolve collisions.

2.7.6 2.10.5 Observation

Both *Linear probing* and *double hashing* are called open addressed hashing because the original hashtable may grow in size after hashing. The extra hash cells may be required as a consequence of rehashing.

2.7.7 Activity

1. Write an algorithm that creates a hash table of integers, taking values from the keyboard, ended with a negative number.
2. For each method of collision resolving, write an algorithm.
3. Calculate the complexity of each algorithm.
4. Compare the complexities of the previous algorithms using the big-O notations.

[8] In practice, the next probling location is h_1 + r for some integer r > 1 in order to better spread keys across the address space.

CHAPTER 3. ALGORITHM DESIGN TECHNIQUES

3.1 PRELIMINARY

3.1.1 Essential reading

- Duane A. Bailey *Java Structures: Data Structures in Java for the principled programmer*. (McGraw-Hill Companies, Inc. 1999, McGraw-Hill International editions) [ISBN 0-13-489428-6]. Chapter 4.2
- Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 4.1–4.4
- Richard Johnsonbaugh and Marcus Schaefer *Algorithms*. (Pearson Education International, 2004) [ISBN 0-13-122853-6]. Chapter 5.2
- Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 4.10, 5.12
- Sara Baase and Allen Van Gelder *Computer Algorithms: Introduction to Design and Analysis*. (Addison-Wesley Publishing Company, 2000) third edition [ISBN 0-201-61244-5]. Chapter 3, 4.3, 10.3

3.1.2 Learning outcomes

We discuss a few commonly used techniques in algorithm design including the concepts of recursive methods, divide and conquer, dynamic programming and the greedy approach in this chapter.

Having read this chapter and consulted the relevant material you should be able to:

- Explain the concept of recursion and the advantage of the recursive approach;
- Describe the concept of dynamic programming and the greedy approach;
- Develop simple recursive algorithms;
- Calculate the complexities of the related algorithms.

3.2 RECURSION

A subprogram that calls itself is said to be *recursive*. Recursion is a powerful algorithmic tool for problem solving. In Mathematics, recursion is a process (or phenomenon) of finding solutions to a sequence of sub-problems that are identical to the original problem but smaller in problem size. If the right hand side of the equation of a function contains an identical function to the left hand side, the function is a recursive function. For example, the function for computing the famous n^{th} Fibonacci item can be written as:

$$f(0) = f(1) = 1$$
$$f(n) = f(n-1) + f(n-2), \quad \text{for } n > 1$$

This is a recursive function: The first line of the specification gives the so-called *base case*, and in the second line, expression *f*(*n*) on the left side of the equation recurs on the

right side of the equation twice in the same form as $f(n - 1)$ and $f(n - 2)$, but the arguments become smaller as $n-1$ and $n-2$.

A function is recursive if its solution contains a call of the function itself as part of the solution for a given problem. Similarly, a program (or subprogram) is recursive if it contains a statement which calls itself as part of the solution.

3.2.1 Why recursion?

We first summarise the reasons and then look at an example.

- It may be the best way of thinking to solve some problems such as the Towers of Hanoi problem below.
- Algorithms can be very short as compared to an iterative version.
- It can help us understand certain problems.

It would be difficult to solve a problem such as *Towers of Hanoi* without the use of recursion.

3.2.2 Example: Towers of Hanoi problem

There are 3 pegs (A, B, C) and a set of n disks of n different sizes. We mark them 1 … n with disk 1 the smallest, and n the largest (there are n = 64 disks in the legend). The n disks are to be moved from one peg to another, for example, from peg A to C. However, only one disk is allowed to be moved on each step and the disks must remain in the sorted order with the smallest on the top at any peg, so no disk is above a smaller one. The following figure shows that three disks on peg A need to be moved to peg C.

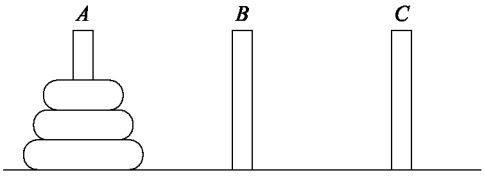


Figure: Towers of Hanoi problem

Let us summarise the problem again:

- *Consider three pegs (A, B, C) and 3 disks (1, 2, 3) as in the previous figure, with disk 1 the smallest, and 3 the largest.*
- *Objective: to move the disks from peg A to C;*
- *Two rules:*
  - *one disk may be moved at a time;*
  - *a larger disk can never be placed on a smaller disk.*

Suppose n is the number of disks. We take a recursive approach.

1. *Base case:*

The solution for 1-disk problem ($n = 1$)*:*

Move the disk from peg A to peg C

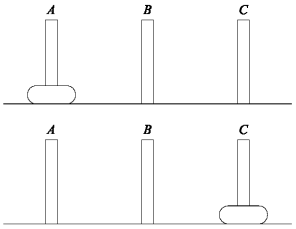


Figure: One disk

2. Inductive steps:

(a) *The solution for 2-disk problem* (n = 2):

i. Use the *one-disk solution* to move disk 1 to peg B,
ii. Move disk 2 to peg C and
iii. Use the *one-disk solution* to move disk 1 to peg C.

(b) *We can now write the solution for n-disk problem:*

i. Use the *(n – 1)-disk solution* to move n – 1 disks (disk 1, …, n – 1) to peg B;
ii. Move disk n to peg C;
iii. Use the *(n –1)-disk solution* to move n – 1 disks from peg B to peg C.

Once we have the formula, it is easy to write the recursive algorithm for the problem.

Algorithm 3.6 listMoves(int NumDisks, char StartPeg, LastPeg, SparePeg)

INPUT: The number of disks to move, the initial peg StartPeg, the destination peg LastPeg and the working peg SparePeg
OUTPUT: Nothing (but display the moves)

```
1: if NumDisks = 1 then
2:    writeln ("Move a disk from", StartPeg, "to", LastPeg);
3: else
4:    ListMoves(NumDisks-1, StartPeg, SparePeg, LastPeg);
5:    writeln ("Move a disk from", StartPeg, "to", LastPeg);
6: ListMoves(NumDisks-1, StartPeg, LastPeg, SparePeg);
7: end if
```

In fact, this program can be simplified to the following algorithm.

Algorithm 3.7 listMoves(int NumDisks, char StartPeg, LastPeg, SparePeg)

INPUT: The number of disks to move, the initial peg StartPeg, the destination peg LastPeg and the working peg SparePeg
OUTPUT: Nothing (but display the moves)

```
1: if NumDisks > 0 then
2:    ListMoves(NumDisks-1, StartPeg, SparePeg, LastPeg);
3:    writeln ("Move a disk from", StartPeg, "to", LastPeg);
4:    ListMoves(NumDisks-1, StartPeg, LastPeg, SparePeg);
5: end if
```

Note: A non-recursive solution may be more time and space efficient, i.e. may use less time and space. So do not use recursion if the problem can be easily solved without it.

We could give a recursive definition of a linked list.

A linked list is a pointer that is either null or references a head node. The head node contains a data field and a pointer that satisfies the criteria for being a linked list.

3.2.3 Tail recursion

A recursive program is "tail-recursive" if only one recursive call appears in the program and that recursive call is the last operation performed at that recursive level. For example, Algorithm 3.4 is tail-recursive but Algorithm 3.6 is not. Tail recursive methods are desirable due to their relative space efficiency.

3.2.4 Principles of recursive problem solving

One way to help recursive thinking is always ask: "What could I do if I had a solution to a simpler, or smaller, version of the same problem?"

Example: *Given the declarations for the linked list structure as below, write a recursive procedure to search the list for a particular item* x *and return a pointer to the item in the list if it is found. If the item is not found in the list, a null pointer should be returned.*

Algorithm 3.9 find(node *l*, object *x*)

```
1: if l ≠ null then
2:    if l.data ≠ x then
3:       find(l.next, x)
4:    end if
5: end if
```

Alternatively, the previous algorithm can be simplified to the following.

Algorithm 3.10 find(node l, object x)

```
1: if (L ≠ null) and (l.data ≠ x) then
2:    find(l.next, x)
3: end if
```

3.2.5 3.3.11Activity

1. For each of these problems, write an iterative and a recursive algorithm:
   (a) To compute the factorial of a non-negative integer;
   (b) To write a character string backwards, i.e. in the reverse order;
   (c) To compute the n^{th} term in a Fibonacci sequence.
2. Calculate the complexity of each of the previous algorithms;
3. For each problem, compare using big-O notations, the iterative and the recursive approaches;
4. Deduce the suitable algorithm for each problem.

3.3 DIVIDE AND CONQUER

This approach divides an instance of a problem P into at least two smaller instances, P_1 and P_2 for example. P_1 and P_2 are of the same problem in nature to P, the original, but much smaller in size[9]. The smaller problems are called *subproblems*. If the solutions for the smaller instances are available individually, the solution to the original P can be derived by simply combining the solutions to the subproblems P_1 and P_2. If the solution to P_1 or P_2 is unavailable, P_1 or P_2 should be divided further into even smaller instances. The dividing processes continue until the solutions to the subproblems are found and the combination processes start to return the partial solutions to each subproblem at higher levels.

3.3.1 Steps in the divide and conquer approach

An algorithm taking the divide and conquer approach usually includes the following main steps:

1. Divide an instance of a problem into smaller instances;
2. Solve the smaller instances recursively;
3. Combine, if necessary, the solutions of the subproblems to form the solution to the original problem.

3.3.1.1 Binary search

Given a sorted list L[0..n – 1] and a *key X*, we want to know if X is in the list. If yes, the position of X in the list is returned. Otherwise, a null is returned.

The idea is to check if X is the middle element of the list $L[mid]$, where mid is the index of the middle element. If not, $L[mid]$ divides the list into two halves, and only one half needs to be checked.

If $X = L[mid]$ then X is found in the list. Its location index mid is then returned as the searching result. Otherwise, if $X < L[mid]$ then the first half is further checked, else the second half is checked. This process is continued recursively until either X is found or the whole list is checked.

Example: *Suppose that we want to search for an item* $X = 6$ *from a given ordered list (1, 3, 4, 6,7, 8, 9, 20, 23, 25, 27)*

The searching process is as below, with $L[mid]$ *in bold:*

0	1	2	3	4	5	6	7	8	9	10		mid
1	3	4	6	7	**8**	9	20	23	25	27		5
1	3	**4**	6	7							Since 8 > 6,	
											$l = 0, r = 4$	2
			6	7							since 4 < 6,	
											$l = 3, r = 4$	3
			6								Found 6	

[9] e.g half of the original size or smaller.

3.3.1.2 Merge sort

Merge sort is another example of applying the divide and conquer technique. The idea is to first divide the original list into two halves, *lList* and *rList*, then merge-sort the two sublists, recursively. The two sorted halves *lList* and *rList* are merged to form a sorted list.

Generally speaking, to merge two objects means to combine them, so as to become part of a larger whole object. However, the word *merge* used in our context implies one type restriction: that is, the two original list objects and the combined list object after merging must be of the same type. Hence, merging two sorted lists cannot be achieved by simply appending one sorted list to another as in the case of unsorted lists. Merging two sorted lists is a combination process that must produce a *sorted* list.

One easy way to merge two sorted lists is to first store the two lists in two queues, *lList* and *rList*. An empty queue combination is used to store the result. In each iteration, we compare the front elements of *lList* and *rList*, and dequeue the smaller element and append it to the combination. This process repeats until the end of one list is reached. Finally the remains of other list will be appended to the result list.

Example: *Demonstrate the process of merging two sorted lists* l = (8, 34, 51, 64) *and* r = (21,32) *into one list* c.

l	*r*	*Result in* c
8 *34 51 64*	*2132*	
34 51 64	**21***32*	*8*
34 51 64	**32**	*821*
34 51 64		*82132*
		8 21 32 34 5164

In each iteration, the front *elements of the two queues are compared, the smaller one (marked in bold) is then* dequeue*d and* enqueue*d to the result list* c*. When one queue (*r *in this example) is empty, the other* (*l* = (34, 51, 64) *in the example) is appended to the result list* c*.*

Example: *Sort the list 26 33 35 29 19 12 22 using* mergeSort *algorithm.*

The list is divided to two sublists lList = (*26 33 35 29*) *and rList* = (*19 12 22*)*, each of almost a half size of the original list. We now have two smaller subproblems of the original problem.*

The algorithm solves each of the smaller problems recursively, mergeSort(lList) first. So (26 33 35 29) is divided to two sublists (26 33) and (3529). The 'new lList*' (26 33) is further divided to (26) and (33). A one-element list is sorted, so the solutions 26 and 33 are merged. The solution (26 33) returns to* lList *at the previous recursive level. Similarly, (35 29) is divided into (35) and (29), and then merged to (29 35). This returns to rList. Next, the two smaller sorted lists (26 33) and (29 35) are merged to (26 29 33 35), which returns to the lList. A similar process happens on (19 12 22).*

This process is traced line by line in the next table.

lList	rList	Result on each recursive level
26 33 35 29	19 12 22	
26 33	35 29	
26	33	26 33
26 33		
35	29	29 35
	29 35	
26 33	29 35 26	29 33 35
26 29 33 35		
19 12	22	
19	12	12 19
12 19	22	12 19 22
	12 19 22	
26 29 33 3	12 19 22	12 19 22 26 29 33 35

3.3.1.3 Quick sort

Similar to Merge Sort, Quicksort also splits a list into two parts according to, however, the value of a so-called *pivot* element. A pivot can be a randomly selected element in the array, for example, the first element. The pivot value is used as a guide item to which all the values compare. On each recursive round, we idendify a correct pivot location such that, all the elements on its left are of smaller value than the pivot, and all the elements on its right are of larger value than the pivot. In this way, the pivot divides the given list into two parts.

Example: *Given a 33 26 35 29 19 12 22, with the first element ,* 33 *let select 33 as a pivot, the pivot loation should be where 33 is: 26 29 19 12 22 35, because all the values before the pivot position are smaller than 33 and all the values after the pivot are larger than 33.*

Let the original list be list, the left part be lList and the right part be rList. The two parts, lList and rList are then recursively applied the quicksort. The idea is that if both lList and rList are sorted, the whole list, i.e. [lList] pivot [rList] is sorted.

Example: *Sort the list 33 26 35 29 19 12 22.*

Each line in the following table shows the content of Pivot, lList, rList, Result on each recursive level *in each iteration. We select the first element as a pivot (underlined) each time.*

This is what happens:

lList	Pivot	rList	combined sublist
33 26 35 29 19 12 22			
26 29 19 12 22	33	35	
19 12 22	26	29	
12	19	22	
			12 19 22
12 19 22	26	29	12 19 22 26 29
12 19 22 26 29	33	35	12 19 22 26 29 33 35

3.3.2 When Divide and Conquer inefficient

The divide and conquer method can be inefficient for certain partitions of a given problem in terms of the time and space complexity.

3.3.2.1 Example

Consider the following algorithms:

Algorithm 3.15 int factorial(int *n*)

```
1: if (n ≤ 1) then
2:    factorial ← 1
3: else
4:    factorial ← n × factorial(n – 1)
5: end if
```

Algorithm 3.16 int fibonacciItem(int *n*); (where *n* > 0)

```
1: if (n = 1) or (n = 2) then
2:    return 1
3: else
4:    return fibonacciItem(n – 1) + fibonacciItem(n – 2)
5: end if
```

In the first algorithm, *factorial*(*n* – 1) is of similar size to the original problem size *n*. Intuitively, this is inefficient because the amount of work required to solve the slightly smaller problem is not reduced much in time, and there is extra space requirement for recursion implementation.

In the second algorithm, *fibonacciItem*(n – 1) and *fibonacciItem*(n – 2) are of similar size to the original problem size *n*. There are also overlaps between *fibonacciItem*(n – 1) and *fibonacciItem*(n – 2). It is costy in terms of both the time and space.

We should therefore avoid the divide and conquer approach for the following cases in general:[10]

1. An instance of size *n* is divided into two or more instances of similar size to *n*. Such a partition may lead to an exponential time algorithm.
2. An instance of size *n* is divided into about *n* instances of size *n*/*c* where c is a constant.

In a typical divide and conquer approach, a problem instance of size *n*, is divided into two instances of a half size (n/2), or, more generally, may be divided into *b* instances of approximately size *n*/*b*. Suppose *a*, a constant number and that these are solved

[10] Note: Some problems have exponential time algorithms only (see Chapter 8).

recursively. To simplify the analysis, we assume that $b > 1$ and $a \geq 1$, the problem size n is a power of b.

The time complexity $T(n)$ for such a recursive algorithm can be represented by the so-called *general divide and conquer recurrence* equation:

$$T(n) = a \times T(n/b) + f(n)$$

where $f(n)$ is the time spent on dividing the problem into smaller ones and combining their solutions.

3.4 DYNAMIC PROGRAMMING

The dynamic programming approach is used as a general algorithm design technique. It is particularly useful for solving problems with overlapped subproblems.

3.4.1 Overlapped subproblems

Consider the Fibonacci problem assuming n > 0

$$f(n) = \begin{cases} f(1) = f(2) = 1 \\ f(n) = f(n-1) + f(n-2), \text{ for } n > 2 \end{cases}$$ and the following recursive algorithm:

Algorithm 3.17 int fibonacciItem(**int** *n*); (where *n* > 0)

```
1: if (n = 1) or (n = 2)  then
2:    return 1
3: else
4:    return fibonacciItem(n – 1) + fibonacciItem(n – 2)
5: end if
```

In each $i>2$, two previous terms *fibonacciItem*(i–1) and *fibonacciItem*(i–2) are required. For iteration i–1, two previous terms *fibanacciItem*(i–2) and *fibanacciItem*(i–3) are required. That is,

$$f(i) = f(i-1) + \boxed{f(i-2)}, \text{ for } n > 2$$

$$f(i-1) = \boxed{f(i-2)} + f(i-3), \text{ for } n > 2$$

$$f(i-2) = f(i-3) + f(i-4), \text{ for } n > 2$$

As we can see, the overlapping *fibonacciItem*(i–2) recursive call appears in both iteration i and i–1. In fact, it overlaps for all $f(i)$, where $i>2$ which is very inefficient.

3.4.1.1 Dynamic Programming approach

The goal of Dynamic Programming is to solve the overlapped parts of the subproblems once only. Instead of repeatedly solving the overlapped subproblems again and again, the Dynamic programming approach records intermediate partial results in a table. The solutions to any of the subproblems, if available, can then be retrieved directly from the table to form the solution to another bigger (sub)problem on request.

To avoid the repeating computation for overlapped subproblems, we store all the partial results as the computation goes along in a table fibonacciItem[i] like a 'performance program' when you go to theatre:

i	1	2	3	4	5	6	7	8	9	. . .
fibonacciItem[i]	1	1	2	3	5	8	13	21	34	. . .

Now for each $i > 2$, only addition is required because the partial results for both previous terms *fibonacciItem*(i−1) and *fibanacciItem*(i−2) are already available from the 'program' table.

The main steps involved in dynamic programming are:

1. Divide the given problem into smaller problems which can be characterised by parameter size. This is the same as the 'dividing' step in Recursion or in the Divide and conquer approach;
2. Solve the subproblems from the *initial* state in a bottom-up fashion and store the partial results in a 'program' table for later calls.

Applying dynamic programming ideas, we can derive the following alternative algorithm for computing the Fibonacci item:

Algorithm 3.18 int fibonacciDP(**int** n); (where $n > 0$)

```
1: fibonacciItem[1] ← 1, fibonacciItem[2] ← 1
2: for i←3, i≤n, i←i+1 do
3:     fibonacciItem[i] ← fibonacciItem(i–1)+fibonacciItem(i–2);
4: end for
5: return fibonacciItem[n]
```

First, we use the same recurrence as before, and divide the problem of size n to two smaller subproblems of size n – 1 and n – 2. Secondly, we solve the two smallest subproblems *fibonacciItem*(1) and *fibonacciItem*(2) and store the respective results 1 and 1 in *fibonacciItem*[1] and *fibonacciItem*[2]. We then use the recurrent formula to compute *fibonacciItem*[3] = *fibonacciItem*[1] + *fibonacciItem*[2], that is, *fibonacciItem*[1] + *fibonacciItem*[2] = 1 + 1 = 2. The result is then stored in *fibonacciItem*[3]. This process continues for i = 4 … n. In this way, each *fibonacciItem*(i), for i = 3 … n, only needs to compute once.

3.4.2 Efficiency of dynamic programming

Dynamic programming can be more time-efficient because repeating computations are avoided for overlapped subproblems. Note the implementation by iteration in the dynamic programming approach. This can also save substantial amount of time in comparison with recursive approaches when n is large.

On the other hand, dynamic programming is often space-inefficient. You may have noticed that the previous algorithm uses an array, but this can be avoided sometimes, for example, as in the following lgorithm.

Algorithm 3.19 int fibonacciDP1(**int** n)

```
1: if n = 1 then
2:     return 1;
3: else if (n=2) then
4:    return;
5: end if
6:  fibonacciItem_1 ← 1, fibonacciItem_2 ← 1;
7: for i←3, i≤n, i←i+1 do
8:     fibonacciItem ← fibonacciItem_1 + fibonacciItem_2;
9:     fibonacciItem_2 ← fibonacciItem_1;
10:    fibonacciItem_1 ← fibonacciItem;
11: end for
12: return fibonacciItem ;
```

3.4.3 Similarity to the Divide and Conquer approach

Dynamic programming is similar to Divide and Conquer since they both divide a problem into several smaller subproblems. The difference is that the dynamic programming approach computes the solutions to the subproblems in a bottom-up fashion while Divide and Conquer takes a top-down approach. If the subproblems are overlaps to each other, the bottom up approach such as dynamic programming may save time and space resources.

3.4.4 Observation

1. The dynamic programming approach is effective when the problem can be reduced to several slightly smaller subproblems.
2. All the subproblems are computed and the results are stored in a table to avoid repeating computation.
3. Dynamic programming often requires a large space.

3.5 ACTIVITY

1. Design an algorithm to find the largest element in an array of *n* numbers, using each of the three approaches.
2. Write an algorithm to calculate the factorial of a positive number *n*, using each of the three approaches;
3. Write an algorithm to calculate the n^{th} term of the Fibonacci sequence using each of the three approaches;
4. For each question, calculate the complexity of each algorithm and establish a relationship using big-O notations among the three algorithms.

CHAPTER 4. ABSTRACT DATA TYPES II: TREES, GRAPHS AND HEAPS

4.1 PRELIMINARY

4.1.1 Essential reading

- Duane A. Bailey *Java Structures: Data Structures in Java for the principled programmer*. (McGraw-Hill Companies, Inc. 1999, McGraw-Hill International editions) [ISBN 0-13-489428-6]. Chapter 10, 11
- Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2001, fourth edition) [ISBN 0-201-35744-5]. Chapter 7, 8, 13
- Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 6.4
- Michael Main, *Data Structures and Other Projects Using Java*. (Addison Wesley Longman Inc., 1999) [ISBN 0-201-35744-5]. Chapter 9
- Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 5
- Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3].Chapter 6, 17, 18, 20

4.1.2 Learning outcomes

This chapter introduces one of the most important abstract data structures which is called a *binary tree*. Having read this chapter and consulted the relevant material you should be able to:

- explain the importance of binary trees, graphs and heaps;
- describe and design main algorithms on trees;
- describe and design main algorithms on graphs;
- describe and design main algorithms on heaps;
- calculate and compare the complexities of various algorithms.

4.2 TREES

A tree (also called a free tree) is defined as a set of vertices (also called nodes) connected by their edges so that there is exactly one way to traverse from any vertex to any other vertex.

Trees are a very important and widely used abstract data structure in computer science. A *tree* represents a hierarchical relationship. Each node in a tree spawns one or more branches that each leads to the top node of a *subtree*. Almost all operating systems store sets of files in trees or tree-like structures.

4.2.1 Terms and concepts

To ease the discussion, we first define some terms and concepts about trees.

Root: The top most node in the tree.

Leaf: A node that has no children.

Parent: The predecessor node that every node has (except the root).

Child: A successor node that each node has (except leaves).

Siblings: Successor nodes that share a common parent.

Subtrees: A subtree is a substructure of a tree. Each node in a tree may be thought of as the root of a *subtree*.

Degree of a tree node: The number of children (or subtrees) of a node.

Degree of the tree: The maximum of the degrees of the nodes of the tree.

Ancestors of a node: All the nodes along the path from the root to that node.

Path from node n_1 to n_k: A sequence of nodes from n_1 to n_k.

L*ength* of the path: The number of edges on the path.

Depth of a node: The length of the unique path from the root to the node.

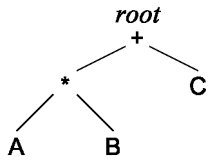
Forest: A collection of trees.

Binary trees: The trees in which every node has at most two subtrees, although either or both subtrees could be empty.

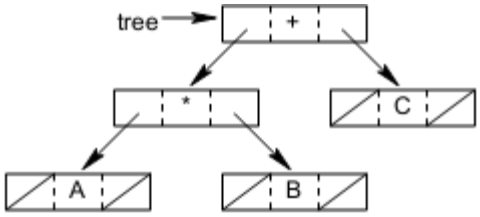
4.2.1.1 Example

An arithmetic expression: $A * B + C$ can be represented by a binary tree.

An expression tree is a binary tree. The operators, such as $*$, /, +, are stored on the internal nodes and the operands, such as A, B, C, are on the leaves. The value of the subtree rooted at an internal node can be derived by applying the operator at the node to the operands at its children recursively.



A logical diagrammatic view.



An implementation view.

Figure: An expression tree

For example, consider the node '$*$' in the figure above. Let $A = 2$, $B = 3$. The value of the subtree at the node '$*$' is $2 \times 3 = 6$.

4.2.2 Recursive definition of Trees

A tree is a collection of nodes. The collection can be *empty*. Otherwise, a tree consists of a distinguished node r (for *root*) and zero or more subtrees each of whose roots are connected by a directed edge from r.

A *binary tree* (see the following figure) is either empty, or it consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree* of the root.

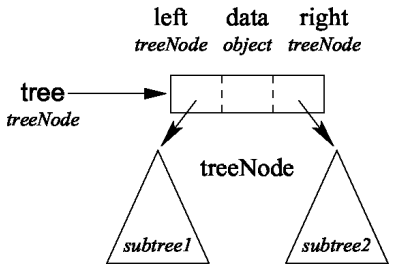


Figure: Recursive definition of a binary tree.

4.2.3 Basic operations on binary trees

1. **initialise()** - creates an empty binary tree.
2. **empty ()** - returns *true* if the binary tree is empty and *false* otherwise.
3. **create (x)** - creates a one-node binary tree T, where T.data=x.
4. **combine (l, r)** - creates a tree T whose left subtree is *l* and right subtree is *r.*
5. **traverse()** - traverse every node of a tree T.

4.2.4 Traversal of a binary tree

Traversal of a data structure means visiting each node exactly once. This is more interesting in trees than in lists because trees offer no natural linear sequence to follow.

There are three particularly important traversals of a binary tree, namely *preorder, inorder* and *postorder traversal*. A binary tree may be empty, in which case there is no node to visit. Otherwise, the tree consists of a root node, left subtree and right subtree which must be visited. The only difference between the three traversals is the order of the steps: preorder visits the root first, postorder visits it last and inorder visits it in between the two subtree traversals. The left subtree is always visited before the right.

The recursive algorithms for the three traversals are as follows:

Algorithm 4.4 preorder(treeNode T)

```
1: if not empty(T) then
2:     print(T.da);
3:     preorder(T.left);
4:     preorder(T.right)
5: end if
```

Algorithm 4.5 inorder(tree T)

```
1: if not empty(T) then
2:     inorder(T.left);
3:     print(T.da);
4:     inorder(T.right)
5: end if
```

Algorithm 4.6 postorder(treeNode T)

```
1: if not empty(T) then
2:     postorder(T.left);
3:     postorder(T.right);
4:     print(T.da)
5: end if
```

The three traversals, i.e. the *preorder*, *inorder* and *postorder* traversal on an expression tree can result in three forms of arithmetic expression, namely, *prefix*, *infix* and *postfix* form respectively.

Example: Let consider the following expression tree *A*B+C*. Then, we have the following results:

Traversal	*Nodes visited*	*Arithmetic expression*
postorder:	*AB*C+*	*postfix form*
preorder:	*+*ABC*	*prefix form*
inorder:	*A*B+C*	*infix form*

4.2.5 Construction of an expression tree

As an example, we show how to construct an arithmetic expression tree given the expression in its *postfix* form. The postfix form is an expression where the operator is placed after both operands. For example, the postfix form of an arithmetic expression $a+b$ is $ab+$, and the postfix form of $(a+b)*c$ is $ab+c*$.

The normal form of an arithmetic expression such as $a+b$ and $(a+b)*c$ is called the *infix* form. Similarly, an expression can also be written in *prefix* where the operator is placed before operands. For example, the prefix form of the above expressions are $+ab$ and $*+abc$ respectively. Brackets are not needed in the postfix or prefix form and therefore are economical for computer storage. However, they are not easily recognisable by humans.

Suppose the expression is stored in an array and a stack, initially empty, is used to store the partial results.

Algorithm 4.7 treeNode construct()

```
INPUT:   An expression in array;
RETURN:  The root of the expression tree;
1: Read the expression one symbol at a time
2: if the symbol is open then
3:     create a one-node tree and push the pointer to it onto a stack
4: else if the symbol is an operator then
5:     pop pointers to two trees T1 and T2 from the stack and form a new tree whose
       root is the operator and whose left and right children point to T2 and T1 resp;
6:     A pointer to this new tree is then pushed onto the stack;
7: end if
8: return the top of the stack;
```

Example: *We first write the arithmetic expression* (a+b)∗c∗(d+e) *in the postfix form (by hand for the moment):* ab+c∗de+∗*, and store it in an array* S.

We use the standard procedures and functions for trees and stacks where simple variables l, r point to the left and the right subtree respectively, T is the root of the expression tree to be built.

1. *Read a, create('a',T), push(S,T) read b, create('b',T), push(S,T) (Figure (a))*
2. *Read +, pop (S,r), pop (S,l), combine (l,r) and push(S,T) (Figure (b))*
3. *Read c, create('c',T), push(S,T) (Figure (c))*

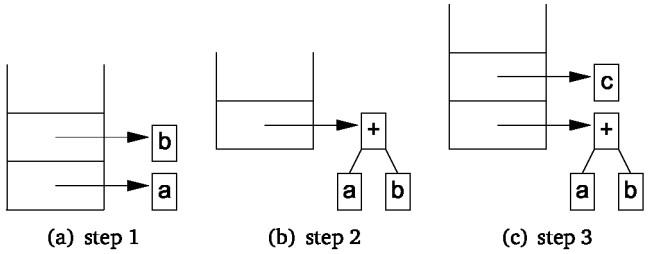


Figure: steps 1 – 3

1. *Read ∗, pop(S, r), pop (S, l), combine (l, r) and push(S,T) (Figure (a))*
2. *Read d, create('d',T), push(S,T)*
3. *Read e, create('e',T), push(S,T) (Figure (b))*
4. *Read +, pop (S,r), pop (S,l), combine (l,r) and push(S,T)*

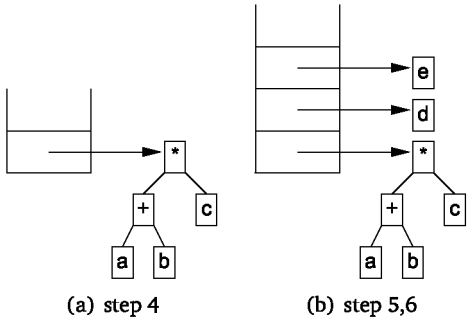


Figure: steps 4–6

*8. Read *, pop(S,r), pop (S,l), combine (l,r) and push(S,T) (Figure (b))*

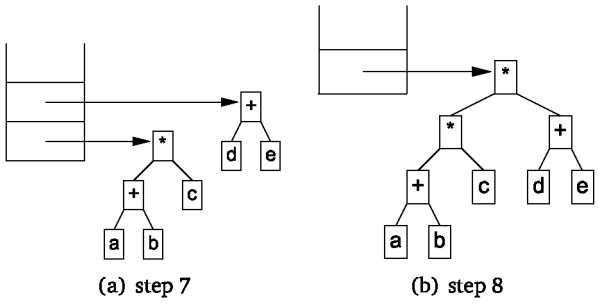


Figure: steps 7 – 8

4.3 ACTIVITY

2. Draw an expression tree for each of the following expressions:

(a) 5
(b) (5 + 6∗4)/2

(c) (5 + 6*4)/2-3/7
(d) 1 + 9*((5+6*4)/2-3/7)
(e) $A \times B$ — $(C + D) \times (P/Q)$

3. Hand draw binary expression trees that correspond to the expressions for which
   (a) The infix representation is $P/(Q + R) * X - Y$
   (b) The postfix representation is $XYZPQR * +/ - *$
   (c) The prefix representation is $+ * - MNP/RS$
4. Write an algorithm that takes two binary trees T1, T2 and a binary tree node v as the arguments. It constructs and returns a new binary tree that has v as its root and whose left sub-tree is T1 and whose right sub-tree is T2. Both T1 and T2 should be empty on completion of the execution.
5. Write an algorithm that deletes the root of a tee and returns both the left and the right sub-trees.
6. For each of the following cases, write an algorithm that traverses a tree and inserts in a queue the values of each the tree nodes:
   a. Preorder principle;
   b. Inorder principle;
   c. Postorder principle.
7. Write an algorithm that deletes a tree of integers;
8. For each of the previous algorithms, calculate the algorithmic complexity.

4.4 PRIORITY QUEUES AND HEAPS

A priority queue is a queue with a conditional *dequeue* operation in addition to the FIFO principle. The elements in the queue have certain orderable characteristics which can be used to decide a *priority*.

For example, given a queue of integers, we want to

1. remove the smallest number from the queue
2. add a new integer to the queue according to the pre-defined priority.

The priority means the *smallest* (or biggest) in some comparable value. This task is such a common feature of many algorithms that the generic name of a *Priority Queue* has been given to such a queue.

A priority queue can be realised by a partially ordered data structure called *heap*.

4.4.1 Binary heaps

A binary heap is a partially ordered *complete* binary tree. Similar to a binary search tree, a heap[11] has an *order* property as well as a *structural* property. We first look at the structural property.

4.4.1.1 Structural property

The structure of a heap is as a complete binary tree. A complete binary tree is a binary tree in which every level is full except possibly the last (bottom) level where only the

[11] In this module, we use 'heap' to mean a binary heap unless stated otherwise.

rightmost leaves may be missing. A complete binary tree is referred as a *full* binary tree if all the leaves are at the same level.

Example: *The binary tree in the following figure is a complete binary tree, and a full binary tree in which all its leaves are at the same level.*

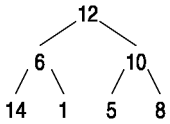


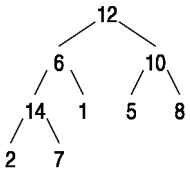
Figure: A full binary tree

The structural property of the heap makes an array implementation easy. The nodes in the complete binary tree in the previous figure can be stored in an array level by level contiguously, for example:

i	1	2	3	4	5	6	7
A[i]	12	6	10	14	1	5	8

The advantage of the array implementation is that each node can be accessed in $O(1)$ time. In addition, the parent or each child of a node can be accessed in $O(1)$ time. For example, node A[i]'s parent is: A[i div2], its left child is A[2i] and the right child is A[2i+1].

Example: *the following figure is another complete binary tree in which all the levels are full except the last level where the leaves are stored from the left to the right without gap.*



i	1	2	3	4	5	6	7	8	9
A[i]	12	6	10	14	1	5	8	2	7

Figure: A complete binary tree

4.4.1.2 The order property

The order property requires, for every node in the structure, that the value of both its child nodes must be smaller (or bigger) than the value at the current node. The complete binary tree in the previous figure is not a heap because it does not have the order property required.

Example: *The following figure shows a heap with the minimum value (a) and a heap with maximum value (b) at the root. The value at each node is smaller than either child.*

Figure: A heap with a MinKey and Maxkey root

4.4.2 Basic heap operations

In this implementation, the numbers in our set are placed at the nodes of a binary tree in such a way that the numbers stored at the children of any node are smaller (or larger) than the number stored at that node (order property). Moreover, the tree is a left complete binary tree, i.e. a binary tree that is completely filled except possibly the bottom level which is filled from left to right (structural property).

Typical operations on Binary heaps:

buildHeap(): constructs the initial heap from a list of items (keys) in arbitrary order.

deleteMin(): removes the smallest element from the root and maintains the heap.

deleteMax() removes the largest element from the root and restores the heap.

insertOne(x) add one element x to the heap and maintain the heap.

4.4.2.1 Deletion

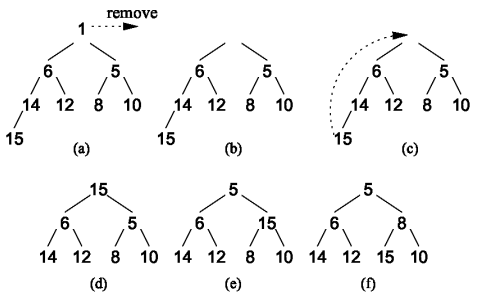
Consider the operation of removing the smallest element from this structure (and reconstituting the tree).

The smallest (or largest) element will always be at the root node.

Example: *The section (1) of the following figure shows how the order property is maintained after (a) the smallest element '1' is removed: (b) The root position becomes available. (c) (d) Datum '15' is moved from the leaf to the root. (e) Datum '15' is swapped with its smaller (the right) child '5', and then (e) swapped with its smaller (the left) child '8'. (f) Datum '15' is finally settled as a leaf and becomes the left child of node '8'.*

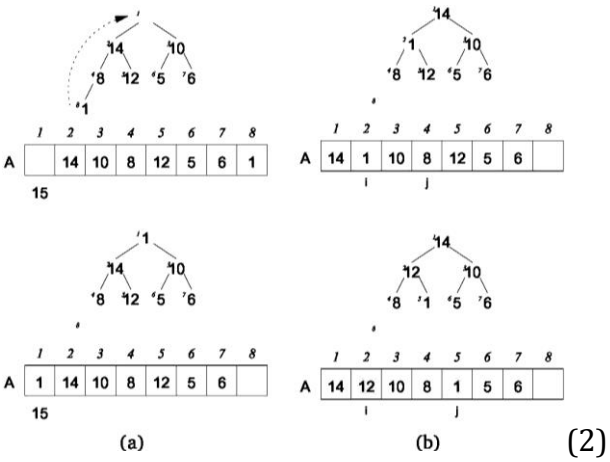
The section (2) of the following figure shows how, after the maximum root element 15 is deleted, the order property is restored by an insertion *and two* swaps *on the corresponding array (the heap).*

1. *Remove the root element 15, and move element 1, the rightmost leaf at the bottom level to the root. (a)*
2. *Restore the order property by repeatedly swapping element '1' with its larger child element '14', e.g. first swap '1' at the root with the left child '14', and then swap '1' with its right child '12', until the order property is restored. (b).*



(1)

Figure: DeleteMin



(2)

Figure: addRoot-1,2

This process can be described in the following Algorithms.

Algorithm 4.8 addRoot(index i, n)

```
1: j ← 2*i;
2: if j ≤ n then
3:    if j < n and A[j] < A[j+1] then
4:       j ← j+1;
5:       if A[i] < A[j] then
6:          tmp ← A[i];
7:          A[i] ← A[j];
8:          A[j] ← tmp;
9:           addRoot(j, n);
10:      end if
10:   end if
12: end if
```

Algorithm 4.9 buildHeap()

```
1: for (k ←1, k ≤ length(A) div 2, k ← k+1) do
2:    addRoot(k, length(A))
3: end for
```

Thus removing the minimum element takes a time proportional to the height of the tree in the worst case, i.e. $O(\log n)$ time, where n is the number of elements in the heap.

4.4.2.2 Insertion

Now consider the operation of adding an element to a heap (and reconstituting a binary heap). Since heaps are complete trees, a new node can easily be added to the first available location from the left at the bottom level. We then check the order property and adjust internal nodes. This is done in a 'bottom-up' fashion. We first compare the value of the new node with its father. If it satisfied the order property, the addition process is completed. If not, we swap it with its father. The checking process is then repeated on the new node on the level above. This process continues until the order property is satisfied (or the new element reaches the root position).

Example: *the following figure shows (a) a binary heap and how the order property is maintained after (b) a new element '2' is inserted at the bottom level of the heap, where the left most available location is identified. (c) '2' is to be swapped with its father '5' since '2' is smaller than '5', (d) '2' is to be swapped with its father '4' because '2' is smaller than '4'. (e) The process ends because '2' is at the root position and the order property is now restored.*

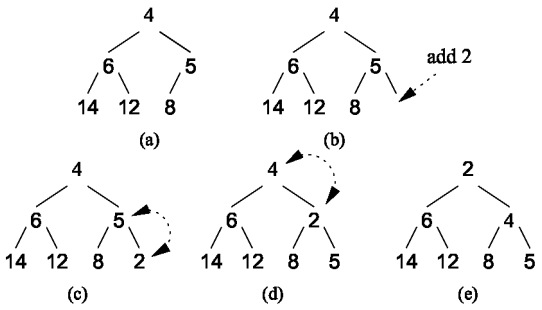


Figure: InsertOne

Thus adding an element also takes a time proportional to the height of the tree in the worst case, i.e. $O(\log n)$ time, where n is the number of elements in the heap.

Having studied how to add one element to a binary heap, we can construct a binary maximum-heap for a given list using the insertion method repeatedly.

Example: *Demonstrate, step by step, how to construct a max-heap for the list of integers* $A[1..8] = (10, 8,14,15,12, 5, 6, 1)$*. Assume the heap is empty initially.*

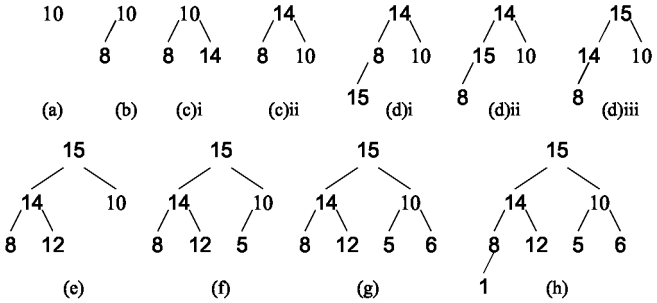


Figure: Construct a binary max-heap

The above figure shows the construction process from the initial state. Starting from the root, a new element is inserted, one by one, to the left most available position at the bottom level of the tree.

4.4.2.3 Applications

A heap is a very useful data structure and can be used in many useful applications. For example, heap sort is one of the efficient sorting algorithms using heaps.

Example: *Sort a list of integers* $A[1..8] = (10, 8,14,15,12,5,6, 1)$ *using a max-heap.*

The list of n = 8 *unsorted integers is first converted to a max-heap, i.e. a partially sorted, left complete binary tree with the largest integer at the root. The first position* $A[1]$ *is the root element. During the sorting process, the list is divided into two sections:* $A[1..k]$*, the heap and* $A[k+1..n]$*, the sorted part. We shall each time remove the root element at* $A[1]$*, the largest integer from the current heap and insert it to location* k+1*,and* k ← k–1*. This is followed by a restoring of the order property of the heap. When the root* r *is removed, we move the larger one of its children to the root position, and similarly, the larger one of the child's children to its position. The process repeats until the order property is restored.*

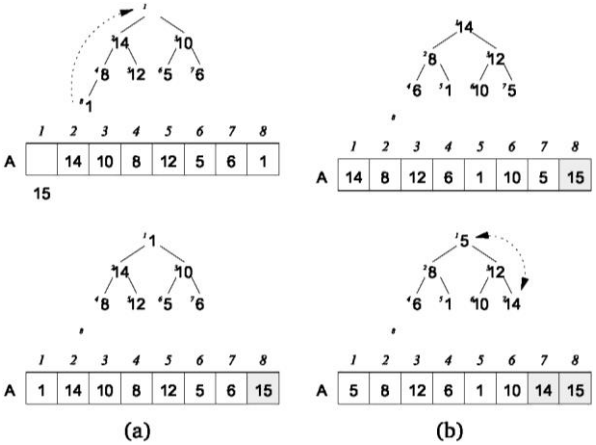


Figure: steps 1 – 2: addRoot-1,2

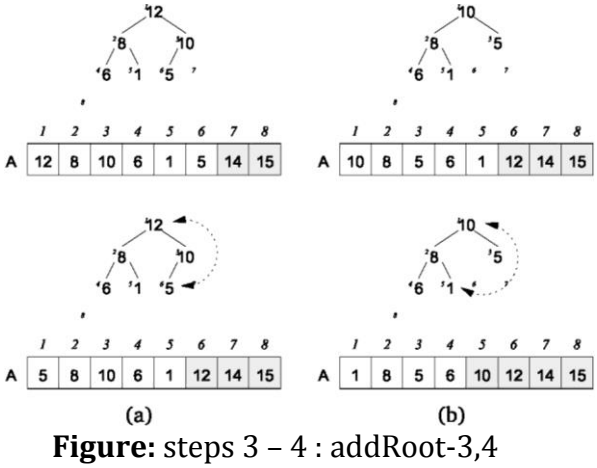


Figure: steps 3 – 4 : addRoot-3,4

The above steps can be summarised in the following Algorithm.

Algorithm 4.10 heapSort(heapArray A)

1: buildHeap; {construct a heap}
2: **for** ($k \leftarrow$ length(A) down to 2) **do**
3: $tmp \leftarrow A[k]$;
4: $A[k] \leftarrow A[1]$;
5: $A[1] \leftarrow tmp$;{swap A[1] with A[k]}
6: *addRoot*(1, k – 1) {Restore the heap properties for the shortened array}
7: **end for**

4.5 ACTIVITY

1. What is the approximate number of comparisons of keys required to find a target in a complete binary tree of size n?
2. Write an algorithm that constructs a heap in any order of your choice.
3. Write an algorithm that deletes a node from a given position in the heap and restores the heap.
4. Write an algorithm that merges two heaps.
5. Calculate the complexities of the previous algorithms.

4.6 GRAPHS

Many relationships in the real world are not *hierarchical* but *bi-directional* or *multi-directional*. In this section, we study another abstract data structure called a *graph*, which represents such *bi-directional* relationships.

Problems as diverse as minimising the costs of communications networks, generating efficient assembly code for evaluating expressions, measuring the reliability of telephone networks, and many others, can be modeled naturally with graphs.

We first look at a real problem:

Problem: Suppose that we want to connect computers in four buildings designed in the following figure by cable. The *question* is: Which pairs of buildings should be directly connected so that the total installation cost would be minimum?

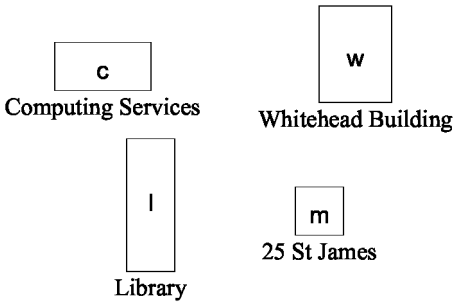


Figure: Four buildings.

Naturally, the four buildings could be represented by four vertices with labels *c*, *w*, *l*, *m* as in the following figure. We then mark the cost between each pair of vertices by lines between vertices.

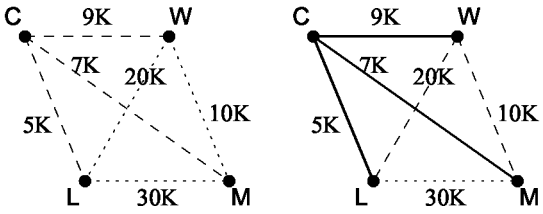


Figure: A graph

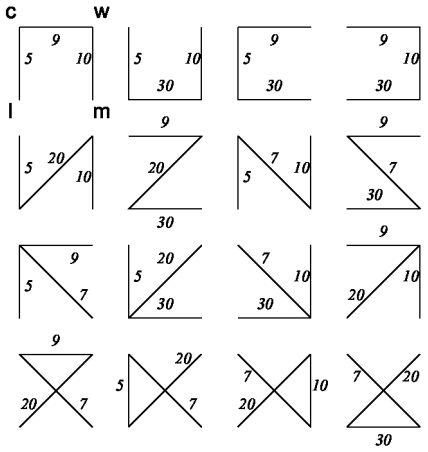


Figure: Possible connections

We look at all the possible connections. We could then make a decision to install the cable for the route that consists of direct connections (c,l), (c,m) and (c,w). The total cost would be: 5 + 7 + 9 = 21K.

4.6.1 Definitions

A Graph $G = (V, E)$ is a finite set of points (vertices) which are interconnected by a finite set of lines (edges) in a space, where V is a set of vertices and E is a set of edges.

In our example (previous figure), $V = \{c, l, w, m\}$ and $E = \{(c,l), (c, m), (c, w)\}$. Normally, the set of vertices are represented by labels of numbers and the edges by letters.

There are two main classes of graphs: *graphs* and *directed graphs* named *digraphs*. For graphs, the edge set consists of a *non-ordered* pair of vertices. For digraphs, the edge set consists of an *ordered* pair of vertices. The following figure illustrates the two classes of graphs.

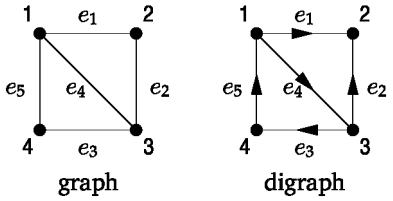


Figure: A graph and a digraph

As for trees, some terms and concepts are introduced for discussions on graphs:

4.6.2 Typical operations on Graphs

Order: The number of vertices in a graph.

Size: The number of the edges in a graph

Path: A sequence of vertices $v_1, v_2, ..., v_k$, where $k \geq 1$ is a path if $(v_i, v_{i+1}) \in V$ for $1 \leq i \leq k-1$.

Length of a path: The number of edges on the path, which equals $k-1$, where k is the number of vertices on the path.

Simple path: A path such that all vertices are distinct, except possibly the first and last vertices. No vertex on the path appears more than once.

Self-loop: A path from a vertex v to itself (v,v).

Cycle: A path $v_1, v_2, ..., v_k$ where $v_1=v_k$.

Spanning tree: A sub-graph and tree that contains all the vertices of the graph.

Minimum spanning tree: A spanning tree of minimum total weight.

Connected graph: A graph in which there is a path between *any two* vertices.

Complete graph: A graph in which there is a path between *every two* vertices of the graph.

Labelled graph: A graph in which every vertex has a fixed identity.

Unlabelled graph: A graph in which there is no fixed identity for each vertex.

Weighted graph: A graph in which every edge is associated with a real value as its weight (or cost).

Simple graph: A graph that contains no self-loop nor parallel edges.

Example: *The following figure shows an example of some specific graphs.*

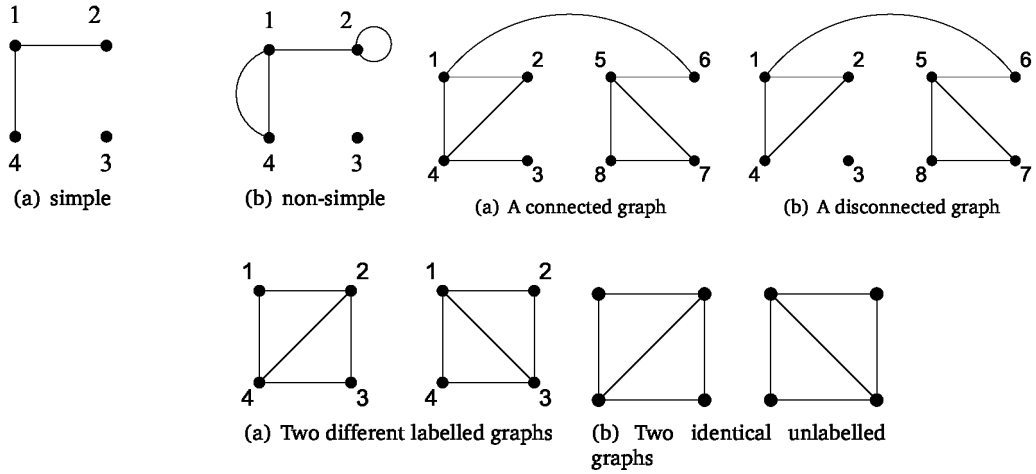


Figure: Some particular graphs.

We consider the simple graph in this course unit unless stated otherwise.

4.6.3 Representation of graphs

We discuss three commonly used data structures to represent graphs. They are *adjacency matrices, incidence matrices* and *adjacency lists*. Indeed, the use of different data structures can sometimes improve (or worsen) the efficiency of algorithms.

4.6.3.1 Adjacency matrices

A graph $G = (V, E)$ can be represented by a 0 – 1 matrix showing the relationship between each pair of vertices of the graph. We assign 1 or 0 depending on whether the two vertices are connected by an edge or not.

Given a graph $G = (V, E)$, let n be the number of vertices. The adjacency matrix of the graph is a $n \times n$ matrix.

$$A=\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \qquad \text{where } a_{i,j}=\begin{cases} 1 & \text{if } (i,j)\in E \\ 0 & \text{otherwise} \end{cases}$$

Example: Let consider the following graph and digraph :

Figure: A graph and digraph

Then we have the following Adjacency matrices.

	c	w	m	l
c	0	0	0	1
w	0	0	1	1
m	0	1	0	0
l	1	1	0	0

$$A=\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency matrix for the graph

	c	w	m	l
c	0	0	0	1
w	0	0	1	1
m	0	0	0	0
l	0	0	0	0

$$A=\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrix for the digraph

Figure: Adjacency matrices for the graph and digraph

4.6.3.2 Incidence matrices

An incidence matrix represents a graph *G* by showing the relationship between every vertex and every edge. We assign 1 or 0 depending on whether a vertex is incident to the edge.

Let *n* be the number of vertices of the graph, and *m* be the number of edges. The incidence matrix is an *n*×*m* matrix:

$$B=\begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix} \text{ Where} \qquad \text{For a graph, } b_{i,j}=\begin{cases} 1 & \text{if vertex } i \text{ and } j \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{For a diagraph} \qquad b_{i,j}=\begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \\ 1 & \text{if edge } j \text{ enters vertex } i \\ 0 & \text{otherwise} \end{cases}$$

Example: *The incidence matrices for the previous graph and digraph are:*

	e_1	e_2	e_3
c	1	0	0
w	0	1	1
m	0	0	1
l	1	1	0

$$B=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

	e_1	e_2	e_3
c	−1	0	0
w	0	−1	−1
m	0	0	1
l	1	1	0

$$B=\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Figure: Incidence matrices

Note Incidence matrices are *not* suitable for any digraph with a self-loop ('loop' for short).

4.6.3.3 Adjacency lists

In an Adjacency list representation, a graph $G = (V, E)$ is represented by an array of lists, one for each vertex in V. For each vertex u in V, the list contains all the vertices adjacent to u in an arbitrary order, usually in increasing or decreasing order for convenience.

This data structure is space efficient for sparse graphs where the number of edges is much less than the squared power of the number of vertices.

4.6.4 Implementations

Like other abstract data structures, the best implementation of a graph, here by an array of lists, or by an array, depends on the given problem. It is conventional to label the vertices of a graph by numerals. For our example, the labels for c, w, m, l can be replaced by 1, 2, 3, 4 respectively. Hence the data structure may be represented as follows:

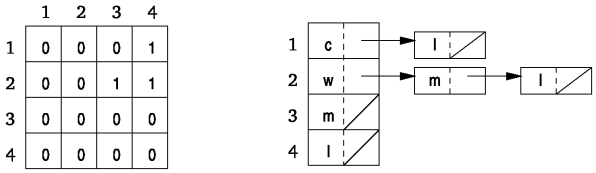


Figure: An adjacency list

4.7 ACTIVITY

1. Consider the following adjacency matrix of a graph: $A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$

   (a) Draw the graph
   (b) Write the adjacency list for the graph
3. Using the adjacency matrix approach, write an algorithm to store a simple graph and display the graph.
4. Write an algorithm to traverse a graph.
5. Calculate the algorithmic complexity of the above algorithms.