BSc (hons) Computing and Information Systems

CIS226

Software engineering, algorithm design and analysis (vol.2)

Subject guide

Written by Ida M Pu, Department of Computing, Goldsmiths College, University of London.

Published November 7, 2006

Printed by Central Printing Service, The University of London

Publisher:
University of London Press
Senate House
Malet Street
London
WC1E 7HU

Contents

Preface **v**

1 Algorithm analysis **1**
1.1 Essential reading 1
1.2 Learning outcomes 1
1.3 Problems and algorithms 1
1.3.1 Implementation 2
1.4 Pseudo-code for algorithm description 2
1.5 Efficiency 3
1.6 Measures of performance 3
1.7 Algorithm analysis 4
1.8 Model of computation 5
1.8.1 Counting steps 6
1.8.2 Implementation 6
1.8.3 Characteristic operations 7
1.9 Asymptotic behaviour 8
1.9.1 Big O notations 8
1.9.2 Comparing orders of two functions 9
1.10 The worst and average cases 9
1.10.1 Implementation 10
1.10.2 Typical growth rates 12
1.11 Verification of an analysis 12

2 Abstract data types I: lists and hashing tables **15**
2.1 Essential reading 15
2.2 Learning outcomes 15
2.3 From Abstraction to Implementation 16
2.3.1 The string abstract data type 17
2.3.2 The matrix abstract data type 18
2.3.3 The keyed list abstract data type 18
2.3.4 Common operations 19
2.4 Data structures and software performance 19
2.5 Motivation of abstract data types 19
2.6 Arrays 20
2.6.1 Applications 21
2.6.2 Array of objects 23
2.6.3 Two and multi-dimensional arrays 24
2.7 Lists 24
2.7.1 References, links or pointers 25
2.7.2 Implementation of the links 26
2.7.3 Linked lists 27
2.7.4 Operations on lists 28
2.7.5 Add one node to a linked list 29
2.7.6 Delete one node from a linked list 30
2.7.7 Implementation 31
2.7.8 Construct a list 31
2.7.9 Implementation 32
2.7.10 Other operations 34
2.7.11 Comparison with arrays 34
2.8 Stacks 36
2.8.1 Operations on stacks 37

2.8.2 Implementation . . . 37
2.8.3 Applications . . . 39
2.9 Queues . . . 40
2.9.1 Operations on queues . . . 40
2.9.2 Implementation of queues . . . 41
2.9.3 Variation of queues . . . 43
2.10 Hashing . . . 44
2.10.1 Collision . . . 46
2.10.2 Collision resolving . . . 46
2.10.3 Extra work for retrieval process . . . 50
2.10.4 Observation . . . 50

3 Algorithm design techniques 53
3.1 Essential reading . . . 53
3.2 Learning outcomes . . . 53
3.3 Recursion . . . 53
3.3.1 Implementation . . . 56
3.3.2 What happens . . . 57
3.3.3 Why recursion? . . . 58
3.3.4 Tail recursion . . . 63
3.3.5 Principles of recursive problem solving . . . 63
3.3.6 Common errors . . . 63
3.4 Divide and conquer . . . 65
3.4.1 Steps in the divide and conquer approach . . . 66
3.4.2 When Divide and Conquer inefficient . . . 69
3.5 Dynamic programming . . . 70
3.5.1 Overlapped subproblems . . . 70
3.5.2 Dynamic programming approach . . . 71
3.5.3 Efficiency of dynamic programming . . . 72
3.5.4 Similarity to the Divide and Conquer approach 72
3.5.5 Observation . . . 73

4 Abstract data types II: trees, graphs and heaps 75
4.1 Essential reading . . . 75
4.2 Learning outcomes . . . 75
4.3 Trees . . . 75
4.3.1 Terms and concepts . . . 76
4.3.2 Implementation of a binary tree . . . 77
4.3.3 Recursive definition of Trees . . . 79
4.3.4 Basic operations on binary trees . . . 80
4.3.5 Traversal of a binary tree . . . 80
4.3.6 Construction of an expression tree . . . 81
4.4 Priority queues and heaps . . . 84
4.4.1 Binary heaps . . . 84
4.4.2 Basic heap operations . . . 86
4.5 Graphs . . . 92

5 Traversal and searching 101
5.1 Essential reading . . . 101
5.2 Learning outcomes . . . 101
5.3 Traversal . . . 101
5.3.1 Traversal on a linear data structure . . . 102
5.3.2 Binary tree traversal . . . 102
5.3.3 Graph traversal . . . 102
5.3.4 Depth-first traversal . . . 102
5.3.5 Breadth-first traversal . . . 102
5.4 Searching . . . 104
5.5 Sequential search . . . 105
5.6 Binary search . . . 106

5.7 Binary search trees . . . 109

6 Sorting 113
6.1 Essential reading . . . 113
6.2 Learning outcomes . . . 113
6.3 Introduction . . . 113
6.4 Motivation . . . 113
6.5 Insertion Sort . . . 114
6.5.1 Algorithm analysis . . . 115
6.6 Selection sort . . . 115
6.7 Shellsort . . . 117
6.8 Mergesort . . . 118
6.9 Quicksort . . . 121
6.10 General lower bounds for sorting . . . 124
6.11 Bucket sort . . . 126
6.12 Sorting large records . . . 127
6.13 Heapsort . . . 128

7 Optimisation problems 137
7.1 Essential reading . . . 137
7.2 Learning outcomes . . . 137
7.3 Optimisation problems . . . 137
7.3.1 Multiplication of matrices . . . 138
7.3.2 Knapsack problem . . . 139
7.3.3 Coin changes . . . 141
7.4 Greedy approach . . . 145
7.4.1 Huffman coding . . . 145
7.4.2 Huffman decompression algorithm . . . 148

8 Limits of computing 151
8.1 Essential reading . . . 151
8.2 Learning outcomes . . . 151
8.3 Computability and computational complexity theory . 151
8.4 Computational model . . . 152
8.5 Decision problems . . . 153
8.6 Measure of problem complexity . . . 154
8.7 Problem classes . . . 155
8.8 Class $\mathcal{P}$ . . . 156
8.9 Class $\mathcal{NP}$ . . . 156
8.10 $\mathcal{P}$ and $\mathcal{NP}$ . . . 157
8.11 NP-complete problems . . . 158
8.11.1 What do we mean by *hard*? . . . 158
8.11.2 How to prove a new problem is NP-complete . . 159
8.11.3 The first NP-complete problem . . . 159
8.11.4 More NP-complete problems . . . 160

9 Text string matching 161
9.1 Essential reading . . . 161
9.2 Learning outcomes . . . 161
9.3 String matching . . . 161
9.4 The string ADT . . . 162
9.5 String matching . . . 164
9.5.1 Naive string matching . . . 164
9.5.2 Observation . . . 166
9.5.3 Boyer-Moore Algorithm . . . 166
9.5.4 Observation . . . 168
9.6 KMP Algorithm . . . 169
9.7 Tries . . . 174
9.7.1 Standard tries . . . 174

9.7.2 Compressed tries 175

10 Graphics and geometry **177**
10.1 Essential reading 177
10.2 Learning outcomes 177
10.3 Quadtrees 177
10.4 Octtrees 178
10.5 Grid files 178
10.6 Operations 178
10.7 Simple geometric objects 179
10.8 Parameter spaces 179

11 Revision for CIS226b examination **181**
11.1 Examination 181
11.2 Revision materials 181
11.3 Questions in the examination 181
11.4 Read questions carefully 182
11.5 Recommendation 182
11.6 Revision topics I 183
11.7 Revision topics II 183
11.8 Good luck! 183

A Sample examination paper **185**

B Sample solutions **189**

C Pseudocode notation **195**
C.1 Values 195
C.2 Types 195
C.3 Operations 195
C.4 Priority 195
C.5 Data structures 196
C.6 Other reserved words 196
C.7 Control keywords 196
C.8 Examples of sequential structures 196

Preface

Introduction

Abstract data types or data structures provide powerful methods of organising large amounts of data. Algorithm analysis enables you to make a decision about the most suitable algorithm before programming. Techniques using abstract data types and design patterns are essential in conventional software development.

Once a good solution method is determined, a program must still be written and implementation has to be completed. In this module we therefore conduct a one hour laboratory session for every three hours of lectures at Goldsmiths College, University of London. This approach is to give you the opportunity to enhance your programming skills in Java.[1] In addition, you will do two assignments to gain problem-solving experience.

[1] Or in other similar programming languages.

Textbooks

Although there are many books on algorithms and data structures available on the market, it has been difficult to find a single text that suits all the needs of this module. On the other hand, you may already have some books on algorithms and data structures from previous programming modules such as CIS212 or CIS109 that you may not have finished studying.

Instead of any single text for essential and desirable reading, we therefore recommend chapters from a number of text books. *All topics covered in these chapters and in this subjectguide are examinable* [2]. There is also a list of books for further reading, and for supporting and historical background reading in the subject guide. Details on availability of the books can be found at `www.amazon.co.uk` or in your institution libraries.

[2] See Chapter 11 for a list of important examinable topics.

The materials covered in the books may overlap and not every chapter of a book is required for the examination. Hence you are not expected to read all the books, nor all the chapters of a single book on the list. You do, however, need to get hold of at least ONE book on algorithms and data structures for frequent reference and for studying individual topics in depth. Your book does *not* have to be in Java but it should cover at least 80 percent of the examinable topics below[3]:

1. Algorithms and efficiency analysis
2. Abstract data types and data structures
3. Lists, stacks, queues and sets
4. Recursions, divide and conquer
5. Trees, graphs and maps

[3] Of course, you still need to have an access to the other 20 percent materials in various sources such as in the library.

6. Sorting, searching and hashing
7. Dynamic programming
8. Graph algorithms, greedy heuristics and approximation
9. Complexity theory
10. Text processing.

Note it would be inconvenient, if not impossible, for you to have to share a library's textbook with other students to study more than 20 percent of the examinable materials, because the book may be unavailable just when you need it urgently for a coursework or examination.

To best study module CIS226b, you need to follow closely the reading instruction for each chapter in the subject guide, paying a particular attention to whether the word 'and' is used between reading items. For example, if 'A and B' is found on the reading list, you are strongly recommended to consult the materials in both A and B.

Below is a collection of textbooks that are recommended at various points in the subject guide which are undated periodically. Note the books are updated frequently these days so search on the internet for the newest edition before any purchase.

Essential reading

The chapters of various books are specified as essential reading. These can be found at the beginning of each chapter of the subject guide. You should read carefully the specified chapter(s) in *at least* one of the books on the list for each chapter of this subject guide. If there is any doubt, you should consult other books on the list. A full list of these books can be found in the next section.

Desirable reading

The following texts are recommended because they provide the detailed background for understanding and appreciation of some topics in this module.

However, you are *not* required to study all the topics in these texts, for no single text can entirely meet the requirements of this course unit. The essential reading chapters are listed at the beginning of each chapter of the subject guide.

List A

Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2005, fifth edition) [ISBN10 0-471-73884-0], [ISBN13 978-471-73884-8].

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].

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6].

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957].

Supporting and historical reading

These books are of interest and are recommended, but you will be fully prepared for the examination should you have studied only those essential texts above. These are provided for completeness and to allow the interested reader to pursue some topics in more depth. You will not be examined on the content of those books listed in the references other than where the material appears in the texts listed above.

List B

Additional reading

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7].

Jeffrey Kingston *Algorithms & Data Structures: Design, Correctness, Analysis*. (Addison-Wesley, 1997, or 1998, second edition) [ISBN 0-201-40374-9].

Richard Johnsonbaugh and Marcus Schaefer *Algorithms*. (Pearson Education International, 2004) [ISBN 0-13-122853-6].

Michael Main, *Data Structures and Other Projects Using Java*. (Addison Wesley Longman Inc., 1999) [ISBN 0-201-35744-5].

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].

Thomas H. Cormen, Stein Clifford Charles E. Leiserson and Ronald L. Rivest *Introduction to Algorithms*. (The MIT Press and McGraw-Hill Book Company, 2001) second edition [ISBN 0-262-03141-8] (MIT Press); [ISBN 0-07-013143-0] (McGraw-Hill).

Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3].

Derick Wood *Data Structures, Algorithms, and Performance*. (Addison-Wesley Publishing Company, Inc., 1993) [ISBN 0-201-52148-2].

Gregory J.E. Rawlins, *Compared to What? - An Introduction to the Analysis of Algorithms*. (Computer Science Press, 1992) [ISBN 0-7167-8243-X].

Christos H. Papadimitriou *Computational Complexity*. (Addison-Wesley Publishing Company, 1994) [ISBN 0-201-53082-1].

Sara Baase *Computer Algorithms: Introduction to Design and*

Analysis. (Addison-Wesley Publishing Company, 1988) second edition [ISBN 0-201-06035-3].

Niklaus Wirth *Algorithms + Data Structures = Programs*. (London: Englewood Cliffs; Prentice-Hall, 1976) [ISBN 0130224189].

About this subject guide

This subject guide outlines the main topics in the syllabus. It can be used as a reference which summarises, highlights and draws attention to some important points of the subject. It cannot, however, replace a textbook although it is fairly self-contained. The guide sets out a sequence which helps you to study the topics in the module within limited hours. The guide provides some additional background material including examples, lab exercises and sample examination questions. It also provides guidance for further reading. recommended textbooks.

One thing you should always bear in mind is the fact that the algorithm subject, like subjects in any other area of computer science, has kept evolving and has been updated frequently. You should therefore not be surprised if you find different approaches, explanations or results in the books you read including this guide.

About CIS226b, volume 2 of CIS226

This module provides an introduction to algorithm design and analysis techniques. Our *aim* is to give students an insight into various standard abstract data types and the common techniques for designing efficient algorithms.

The *objectives* include to:

- introduce fundamental issues in algorithm design such as efficiency, recursion, abstract data types (data structures) and complexity
- demonstrate the standard techniques in algorithm design
- study some well-known problems and algorithms
- further develop your skills in Java programming.

The *learning outcomes* are:

On completion of the second half of the module, students should be able to:

- demonstrate the knowledge of techniques for identifying and solving a computing problem
- choose appropriate data structures for different computation problems
- conduct a basic analysis on time-efficiency of an algorithm in the worst case
- design efficient algorithms and implement them in Java programs

- explain the limit of computations and the complexity classes for decision problems.

Prerequisites

You should be familiar with some basic discrete mathematics, such as functions, big-O notations, sets and logarithms. These topics can be found in CIS102 or equivalent.

You should have already had some prior experience with programming in Java, which is covered in CIS109 or equivalent. This module, CIS226b, focuses on algorithm design and analysis and relies upon most of your programming skills to implement algorithms. It would be an advantage if you are familiar with built-in data structures in Java such as arrays and vectors, concepts in Java such as types versus classes, inheritance, constructor methods, method overloading, method overriding.

Most importantly, you must also have easy access to a Java platform or have a Java platform installed on a computer at home.

Installing Java

There are lots of public domain versions of Java among which the most popular one is called JDK (free). It is at `http://www.javasoft.com/` or `http://java.sun.com/`. A great amount of information is provided on these sites and you can download the software.

If you are using Linux, then the free software package normally already includes a free Java platform.

Testing the installation

An easy way to test your installation is to type the following at a command prompt:

```
java -version
```

A message in response should be seen on the screen with other system information on your platform.

For example:

```
java version "1.5.0_06"
Java(TM) 2 Runtime Environment, Standard Edition (build
1.5.0_06-b05) Java HotSpot(TM) Client VM (build 1.5.0_06-b05,
mixed mode, sharing)
```

Or, for the earlier version:

```
Kaffe Virtual Machine

Copyright (c) 1996-2004 Kaffe.org project contributors
```

```
(please see the source code for a full list of contributors).
All rights reserved. Portions Copyright (c) 1996-2002
Transvirtual Technologies, Inc.

The Kaffe virtual machine is free software, licensed under
the terms of the GNU General Public License.  Kaffe.org is
an independent, free software community project, not directly
affiliated with Transvirtual Technologies, Inc.  Kaffe is
a Trademark of Transvirtual Technologies, Inc.  Kaffe comes
with ABSOLUTELY NO WARRANTY.

Engine: Just-in-time v3   Version: 1.1.4   Java Version: 1.1
```

If you see a flawed response such as a `bad command or file name` or a `command not found` in response to your command '`java -version`', you know that your installation may have not been completely successful. This sometimes may be simply because your system cannot find the correct version of the file which runs Java programs.

As Java has grown to so many versions and variations, we recommend that you focus on the basic functions which can be run in all environments.

CmapTool

Algorithm design, like other types of design, requires a process of development from vague ideas to production details. CmapTools is free software that may help designers to ease the journey of converting their concepts to the design objects. You can find more information about the CmapTool and download the software from `http://cmap.ihmc.us`.

Study time

The materials covered in this module are taught internally at Goldsmiths College, University of London in one academic term, that is a three hour lecture and a one hour supervised lab session per week for ten weeks. For each one hour lecture, students are expected to spend at least two further hours on homework including revision, attempting exercises and lab implementation.

You would, however, normally have to double the study hours if you could not attend lectures. For example, if you self study at home, you would expect to spend six hours on intensive study of the materials and two hours for the lab exercises, plus a similar amount of additional homework time, every week for ten weeks or the equivalent.

It is, of course, impossible to tell you precisely the number of hours required for you to study the materials in this module. It may depend on many elements such as your academic background, the condition of the environment, your health status, the complexity of the subjects and your study methods. I recommend that you add an extra couple of free hours to your plan at least in the first two weeks

and record the time it took you to meet the requirements, and adjust your plan accordingly.

Study methods

One effective way to study algorithms is to commit yourself to various DIY (do it yourself) activities. You should not believe an algorithm until you have implemented and tested it. In theory, the performance of an algorithm can be explored by analysis or implementation. Implementation is not the main concern in this module. However, implementation is an important way forward. It is the only way, sometimes, to prove correctness (or more likely, incorrectness) in practice. Investigation of certain behaviours of an algorithm can also be an important motivation of algorithm design and analysis.

You should try to implement as many algorithms as possible in a conventional language such as Java. Attempting exercises and doing courseworks often offer good opportunities to help your understanding.

It can be useful to remember the Confucian[4] saying about learning as you start your studies:

[4] A famous ancient Chinese philosopher.

> Tell me and I forget;
>
> Show me and I remember;
>
> Let me do it and I understand.

As experts have predicted that more and more people in future will write programs without being programmers, you are recommended to learn the important principles and apply them in your programming practice as much as possible. The experience could be very useful for your future career whatever you do.

More specifically:

1. For every hour of study on new material in a lecture, two hours of lab work and two hours of revision or exercises are highly recommended.
2. Use examples to help gain understanding of each problem.
3. Always ask the question: 'Is there a better solution for the problem?'
4. Practise as much as you can.

Laboratory exercises

There is a one-hour supervised lab session every week for each student at Goldsmiths, University of London.

Lab exercise sheets are set for students to practise their programming skills using the theoretical knowledge gained from the course and are available soon after lectures each week. If you are studying at an institution, your lecturer may provide a similar resource.

Examination

> **Important** *The information and advice given in the following section are based on the examination structure used at the time this guide was written. However, the university can alter the format, style or requirements of an examination paper without notice. Because of this we strongly advise you to check the rubric/instructions on the paper you actually sit*.

The content covered in CIS226b will be examined in the *second* half of a three-hour examination for CIS226.

Students will normally be required to answer a number of questions;[5] each includes a few subquestions (or 'parts') in each half of the paper. These subquestions may be *Bookwork, Similar* and *Unseen*.

[5] You should check the details before the examination.

Details about the examination and revision can be found in Chapter 11.

Every year we advise the candidates to read the questions on the examination paper **carefully**. You should make sure that you fully understand what is required and what subquestions are involved in an examination question. You are encouraged to make notes (crossed through later as not to be marked), if necessary, while attempting the questions. Above all, you should be completely familiar with the course material. To achieve a good grade, you need to have prepared well for the examination and to be able to solve problems by applying the knowledge gained from your studies of the module.

Content and plan

The main topics that we normally cover internally at Goldsmiths College, University of London for CIS226b are as below, but you may adjust them according to your level and your own time available.

```
Week 1
Lecture 1-3
        The aim, objectives and plan of the course
        Problems and algorithms
        Big-O notation
        Pseudocode
        Cmaptools (http://cmap.ihmc.us)
Ex  1   Time efficiency

Week 2
Lecture 4-6
        Abstract Data Types
          array, lists, stacks, queues, sets, (trees,
          graphs, hash tables, heaps)
        Specialised data structures
        Algorithms Design and Implementation
Ex  2   Abstract Data Types
Lab 1   Estimating time efficiency
```

```
Week 3
Lecture 7-9
        Algorithm Design Techniques (1)
          Sorting, selection, searching and traversal.
Ex  3   Sorting, searching, traversal and selection
Lab 2   Implementation of ADT list, or binary tree

Week 4
Lecture 10-12
        Algorithm Design Techniques (2)
           Divide and conquer, Recursion
Ex  4   Divide and conquer, Recursion
Lab 3   Implementation of searching a sorted list, or
           traversal of a connected graph.

Week 5
Lecture 13-15
        Algorithm Design Techniques (3)
           Dynamic programming
        trees, graphs
Ex  5   Dynamic programming
Lab 4   Implementation of a Recursion programme

Study week
        no lectures/labs

Week 7
Lecture 16-18
        Algorithm Design Techniques (4)
           Greedy approach and heuristics
        hash tables, heaps
Ex  7   Greedy approach and heuristics
Lab 5   Implementation of dynamic programming

Week 8
Lecture 19-21
        Limits of Computing
           Intractable problems and approximation
           Introduction to NP-completeness
Ex  8   Intractability and approximation
Lab 7   Greedy approach and heuristics

Week 9
Lecture 22-24
        Some well known problems and algorithms (1)
           String matching problems
Ex  9   String matching problems
Lab 8   Intractability and approximation

Week 10
Lecture 25-27
        Some well known problems and algorithms (2)
           Computational geometry problems
Ex  10  Computational geometry problems
Lab 9   String matching problems

Week 11
Lecture 28-30
```

```
        Revision
Ex  11  Sample examination questions
Lab 11  Computational geometry problems
```

Activity 0.0

WEB SITES[6] AND SOFTWARE

[6] These addresses were accessible when the Guide was written. In case they have changed, you may use a search engine such as Google to search the new web address.

1. Free Java Books
   (a) Thinking in Java
   `http://www.mindview.net/Books/TIJ/`
   (b) Java Gently
   `http://javagently.cs.up.ac.za/jg3e/`
2. What are covered in the first year Java courses in other places?
   (a) David J.Eck's Java course:
   `http://math.hws.edu/eck/cs124/`
   (b) Java Tutorial
   `http://java.sun.com/docs/books/tutorial/`
3. Installing Java system

   `http://burks.bton.ac.uk`
   `http://java.sun.com/`
   `http://textpad.com`
4. Download and install CmapTool

   `http://cmap.ihmc.us`

Chapter 1

Algorithm analysis

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 Algorithms in 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

1.2 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.

1.3 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.

Example 1.1 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?

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: $min \leftarrow x$
6: **end if**
7: **end while**
8: print min;

1.3.1 Implementation

Algorithm 1.1 can be *implemented* to a Java method and run on a computer:

```
import java.util.Scanner;

int min() {

   Scanner input = new Scanner( System.in );
   System.out.println("x=?  (999 to end) ");
   int x = input.nextInt();
   int min = x;

   while (x!=999) {
      System.out.println("x=?  (999 to end) ");
      x = input.nextInt();
      if (x < min) {
         min = x;
      }
   }
   return min;
}
```

1.4 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 (see Appendix C for examples). 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. We encourage you 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.

1.5 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.

Example 1.2 *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.*

1.6 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 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 accomplishment 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.

Observation

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

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.

1.7 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.

1.8 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]

[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.

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.

The assumptions help keep analysis feasible and focused, for 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 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, if anything, make an analysis too complicated to be carried out.

1.8.1 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.

Example 1.3
Problem*: Compute $\sum_{k=1}^{n} k$, where n is an integer.*

Algorithm 1.2 Sum1(n)

INPUT: n
OUTPUT: sum
1: $sum \leftarrow 0$
2: **for** $k \leftarrow 1, k \leq n, k \leftarrow k+1$ **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 $n(n+1)/2$

1.8.2 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.

We have done an analysis. Had we, however, to undertake all the counting every time to analyse an algorithm, the task would be tedious and quickly become *infeasible*. We need some easier approach.

1.8.3 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.

Example 1.4 *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.

To ease the analysis of algorithms, it would be useful to summarise the time complexity for common algorithmic structures below as a reference. You are encouraged to spend some time to derive the formulae by yourself and then compare yours with the ones in a textbook.

- For loops:
- Consecutive statements:
- If-then-else:

Think of an algorithm which takes a number of steps of the following order:

- Logarithmic
- Exponential.

1.9 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.

Example 1.5 *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, $T(n) \sim n$ when $n \to \infty$ [2]

because

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

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 behaviour* of $T(n)$ when $n \to \infty$ and $n \to 0$ respectively.

[2] Here symbol '$\sim$' means 'will be approaching'; and '$\to$' 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.'.

1.9.1 Big O notations

In general, the asymptotic behaviour of functions can be described by 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 $T(n) \leq cg(n)$ when $n \geq n_0$.

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

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

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

Example 1.6 *Let $T_A(n)$ be the time complexity of an algorithm A, and n is the input size of the algorithm. Suppose $T_{A_1} = \frac{n^2}{2}$ and $T_{A_2} = 7n$.*

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

1.9.2 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:

1. If $L = 0$ then $T(n) = o(g(n))$
2. If $L = a \neq 0$ then $T(n) = \Theta(g(n))$
3. If $L = \infty$ then $g(n) = o(T(n))$
4. If L oscillates then there is no relation (but this will not happen in our context).

Example 1.7 *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?*

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 $T_{A_2}(n)$.*

1.10 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 Algorithm 1.4 which determines whether integer x is an element of array $Y[0..n-1]$, where n is a non-negative integer.

Example 1.8

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

```
1: i ← 0
2: boolean found ← false
3: while (notfound) && (i < length(Y)) do
4:     found ← (x = Y[i])
5:     i ← i + 1
6: end while
7: returnfound
```

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 to x, then the while loop (step 3–6 in the algorithm) will only be executed once, and the number of execution steps is only $1 + 1 + 1 \times (1 + 1 + 1 + 1) + 1 = 7$,

including step 1, step 2, $1 \times$ (step $3, 4, 5, 6$), and step 7. If x equals the kth element of Y, the number of execution steps becomes $1 + 1 + k \times (1 + 1 + 1 + 1) + 1 = 3 + 4k$, where $k \leq length(Y)$, including step 1, step 2, $k \times$ (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]

[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.

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 ith instance, for $1 \leq i \leq k$. Let p_i be the probability that this instance occurs.

Then the time complexity for

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

[4] This can be extremely complex sometimes.

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, $A(n) \leq W(n)$.

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.

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

1.10.1 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;
  }
```

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 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 Example 1.9:

1. Place the method `foundFirstX` in Section 1.10.1 in a class `experimentCount`;
2. Write the main class `test` which inputs a `x` and an array, say, `A`, and print out the `foundFirstX(x, A)`.

Example 1.9 `class experimentCount {`

```
  boolean foundFirstX( int x, int Y[]) {
```

```
      ...     // copy the method here
  }
}

class test {

   public static void main(String args[]) {
      experimentCount acount = new experimentCount();

      int A[] = {9, 4, 5, 7, 1, 2};
      int B[] = {2, 4, 5, 7, 1, 2};

      acount.printArray(A);
      System.out.println(acount.foundFirstX(2, A));
      System.out.println();
      acount.printArray(B);
      System.out.println(acount.foundFirstX(2, B));
   }
}
```

1.10.2 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
c	constant
$\log n$	logarithmic
$\log^2 n$	log-squared
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential

1.11 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.

Activity 1.11

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
```

3. A certain algorithm always requires 1000 operations, regardless of the amount of data input. Provide a big-O classification of the algorithm that reflects the efficiency of the algorithm as accurately as possible.
4. Modify the main class `test` in Example 1.9 (Section 1.10.1) so that the program can take a `x` and an array `A` of integers from the keyboard, and print out the number of execution steps of program `foundFirstX(x, A)`.

   Hint You only need to re-write some lines of the main method below (see the comments):

```
class test {

   public static void main(String args[]) {
      experimentCount acount = new experimentCount();

      // ... to be modified from here to the end.
      int A[] = {9, 4, 5, 7, 1, 2};
      int B[] = {2, 4, 5, 7, 1, 2};
      acount.printArray(A);
      System.out.println(acount.foundFirstX(2, A));
      System.out.println();
      acount.printArray(B);
      System.out.println(acount.foundFirstX(2, B));
   }
}
```

Chapter 2

Abstract data types I: lists and hashing tables

2.1 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 to Design and Analysis*. (Addison-Wesley Publishing Company, 2000) third edition [ISBN 0-201-61244-5]. Chapter 2

2.2 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
- implement the lists in a conventional computer language such as Java
- choose appropriate list data structures for a given 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.

2.3 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.

Software development is a complex process. There are creative elements such as the *design* process as well as fundamental tasks of implementation. One of the benefits of introducing ADTs in software development is to separate the design issues from the implementation issues.

The design and implementation processes are highly correlated entities but different in terms of the requirements for changes. The two processes have contrasting properties.

The design process is subject to changes and often involves a series of refinements from ambiguous ideas to clarification. The implementation process makes an idea actually work. It requires a substantial amount of labour to build software that is actually usable in practice. The design process tends to be dynamic and requires frequent changes of plans. In contrast, the program implementation process is static. Ideally, the computer program languages used for implementation should not be changed, and each program should serve its purpose and run forever once it is implemented.

The design process involves naturally a top-down approach and the implementation process a bottom-up, for the top-down approach helps explore the details and the bottom-up approach helps to make sure that everything works properly.

A natural way to develop software is to start from the conceptual objects that we are familiar with. The ideas of ADTs come from real world objects and their environment. Every object in the real world has certain properties. For example, some objects, such as cars and tables, are movable and others, such as trees and buildings, are static. The objects interact with each other according to their pre-defined behaviours.

We now discuss briefly three ADTs as an introduction. They are *string*, *matrix* and *keyed list*. We shall focus on two things for each ADT: first, the logical relationships between the data, and the second, a list of predefined operations.

2.3.1 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.

For example, the word 'algorithm' contains 9 characters and the first character is 'a' and the second is 'l' and so on in that order. An appropriate data structure for a string can be linear such as an array, a linked list or be a hierarchical such as a trie.[1]

[1] We shall study the characteristics of these individually later.

We give each operation a name which is in a similar format to a class name in Java. It consists a string of characters ended by a pair of brackets to quote arguments, which can be empty. The arguments represent the input data required by the operation.

Examples of commonly used operations can be defined as follows:

Typical operations (Strings):

create() initiate a data structure

readString() input a string into an internal data structure such as an array

writeString(s) output string s to an outside device, such as a printer or a file

length(s) return the number of characters in the given string s

concatenate(x,y) to append one string y after another string x

search(c,s) check if a character c, or a substring is contained in a given string s

insert(c,n) add an character c in a given position n in the string and shift all the characters after the position

delete(n, s) remove from a given string s the character that was at position n

equal(x,y) return a *true* if the two given strings x and y are identical and a *false* otherwise

lessThan(x,y) return a *true* if character x is before character y according to the English alphabet, and a *false* otherwise

greatThan(x,y) return a *true* if the corresponding character x is after character y according to the English alphabet, and a *false* otherwise.

Each of these operations will be further developed into a program method (in Java) or procedure or function (in C or others).

2.3.2 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. The adjacency matrix for a simple graph, etc.

Typical operations (Matrices):

create() initiate a data structure

search(c,m) check if a character c, or a substring is contained in a given matrix m

display(m) show the content of the matrix m

add(x,i,j) add an element x into the matrix location (i,j)

update(i,j) update the element value at the matrix location (i,j).

2.3.3 The keyed list abstract data type

Keyed lists are specially useful for information retrieval. In a database, for example, each entry contains at least a field that is used as an 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.

Examples of operations on keyed lists can be defined as follows:

Typical operations (Keyed lists):

create() initiate a data structure

add(d,l) add an element d into the list l

delete(i,l) remove an element at index i from the list l

traverse(l) access each of the elements in list l.
This can be more specific, for example, display, update and print each element in the list l (Section 5.3).

2.3.4 Common operations

From the above examples, it is clear that there are overlaps of the set of operations for various abstract data types. An object oriented computer language provides convenience for defining the common abstract data types and, more importantly, how to reuse certain parts of the programs.

2.4 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.

A data structure represents the logical relationship among the data. 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 $O(1)$ 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 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 precessor. It would naturally take longer to reach the last element in a long list than the one in front.

2.5 Motivation of abstract data types

Historically, programming was regarded as an art, which means that all approaches are acceptable and depends on the personalities of the programmers. Calling it ‘art’ is a relative term and saying the approaches are ‘ad hoc’ methods or immature might be a blunt truth.

As computer science developed, people found that many problems need same fundamental computations which are often repeated since they are required by most programmes.

To save energy, people started to look at engineering approaches, such as define the problem, create frameworks etc.

As computers are required to deal with larger and more complicated problems, people realised how large differences in the performace of programs can result from different data structures.

One important advantage of taking an engineering approach is to enable handling large data sets and large projects by using preassembled and ready-to-use parts or blocks.

Another big advantage of using abstract data types is to allow programmers to concentrate on problem solving and free them from implementation details and to separate design issues from the implementation issues.

We shall show you more examples later on how the abstract data types allow us to separate design and implementation issues.

Activity 2.5

OOP AND ADT

1. Explain, with an example,[2] three principles of Object-oriented programming:
   (a) Encapsulation
   (b) Inheritance
   (c) Polymorphism.
2. Using 'a list of integers' as an example, explain what is meant by ADT and what is meant by Data Structure. What is the main difference between the meaning of the terms ADT list and list?
3. Define an ADT appointment book for the programming problem below:

   Design a computerised appointment book (program) that records your appointments with your tutor during one academic year. To simplify the problem, suppose that you have only one tutor and, for each appointment, the appointment book will allow a brief annotation about the purpose of each appointment along with the date and the time.

   Hint

   (a) The data items in this ADT are the appointments, of which each consists of a date, time and a brief note about the purpose.
   (b) The operations can be:[3]
   - Make an appointment for some purpose for a certain *available* date, time.
   - Cancel the appointment for a certain date and time.
   - Ask whether you have an appointment at a given time.
   - Determine the purpose of the appointment at a given time.
   - Some initialisation operations.

   (c) You should present your design using diagrams and pseudocode as well as brief explanation.

[2] Ideally you should be able to provide a few pieces of simple code in Java. If you cannot, some diagrams are acceptable with reduced marks.

[3] Other correct operations are possible.

2.6 Arrays

Array is a powerful data structure with a constant access time. The structure can be used to mimic a real linear memory storage because 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. For example, an array of integers can be called by one name, say A, and each element in the collection is called by the array name with a unique fixed index of the data item. For example, *read* $A[i]$ or *print* $A[i]$, where i is the index of the data item of interest.

Arrays are often described as *static* data structures if the number of data items has to be specified at the beginning. The maximum number of the index remains the same no matter what operations are performed on the data. 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.

2.6.1 Applications

Example 2.1 *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.

```
(i)     0   1   2   3   4   < -- (j)

 0      23  30  40  90  20
 1      45  55  11  40  30
 2      ...
 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 of 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.

Now the total marks can be computed by adding `mark`$[i, 0 \cdots 4]$ for each i, $i = 0, \cdots, 9$, and the average marks by first adding `mark`$[0 \cdots 9, j]$ for each $j, j = 0, \cdots, 4$ and then divided by 5.

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`$[i, 0 \cdots 4]$ for each i to `mark`$[i, 10]$, where $i = 0, \cdots, 9$.
3. Compute the average marks `average(i)`, i.e. add `mark`$[0 \cdots 9, j]$ for each $j, j = 0, \cdots, 4$, and divided by 5, the

number of subjects.

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) random generate some marks.
   The following method generates an array of marks ranging [a–b] inclusive.

```
void getMarkTable(int a, int b) {
   Random randomNumbers = new Random();

   for (int i=0; i<marks.length; i++){
      for (int j=0; j<marks[i].length; j++){
         marks[i][j]=a+randomNumbers.nextInt(b);;
      }
   }
}
```

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 student number,
// 2nd index j the subject number.
void printMarkTable (int marks[][]){
   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();
   }
}
```

2.6.2 Array of objects

We often need to store a collection of *different* types of data. Taking a simple version of the above problem of student objects as an example, we may like to store one mark, say the average mark of the five subjects, for each student. In addition, we would also like to store more information, for example, the student name, and whether (s)he is an overseas or home student. Here the name should be a *string* type, the mark a *real*, and overseas a *boolean*.

index	name	mark	overseas
1	Tom	54.5	O
2	Alex	78.0	H
3	Anna	90.5	O

Therefore each record consists of three fields of different type:

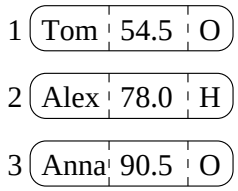


Figure 2.1: An array of student objects

Example 2.2 *If* `studentX` *is such an object in Java, the following statements can be used to assign each data field: (we use '*`true`*' to mean 'overseas student' and '*`false`*' to mean 'home student'.)*

```
studentX.name = 'Peter';
studentX.mark = 98;
studentX.overseas = false;
```

Now studentX has the value (Peter, 98, false).

The next example shows a one dimensional array of objects:

Example 2.3 *The assignment of a value for each field could be:*

```
student[1].name = 'Peter';
student[1].mark = 98;
student[1].overseas = false;

student[3].name = 'John';
student[3].mark = 61.5;
student[3].overseas = true.
```

Now the one dimensional array `student` contains 2 records:

1. student[1]: (Peter, 98, false)
2. student[3]: (John, 61.5, true)

2.6.3 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 a n-dimensional array if n indices are required for access.

We have seen a two dimensional array in Example 2.1.

An example of an element of a three dimensional array is $A[3, 1, 2]$ which is defined uniquely by a triple address $(3, 1, 2)$.

An element of a n dimensional array, e.g. $A(1..n)$ (or $A(0..n-1)$) can be accessed by an address with n indices.

Activity 2.6

ARRAYS

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.

2.7 Lists

Arrays are powerful for storing data but they are after all a data *holder*. Data is stored in contiguous locations (called *storage cells* and the size of the array has to be fixed before any usage of the array. This makes some updating operations such as *insertion* or *deletion* or *updating* extremely inefficient. For example, in order to insert a datum at some place in an array of integers, all the data from that location on have to be moved one cell towards the end to generate a cell for the new datum.

Example 2.4 *We would like to store four integers: (12, 10, 3, 4). We first store the four integers into the array A in a normal way, i.e. in a contiguous way (Figure 2.2 (a)).*

Suppose that we need to insert one integer before 10. We shall then have to first move data 10, 3, 4 one location to the right. If the array is a long one of size n, the worst case would be that we have to shift n data items, one at a time.

Could we leave some 'empty' cells when we store the data (Figure 2.2 (b))? Yes, but it is impossible to decide how many empty cells we should leave. It would be a waste if we leave too many empty cells unused and it does not help if we do not leave enough.

Figure 2.2 (c) suggests a method to form a logical link among the data in an array. Instead of an array of integers, let A be an array of

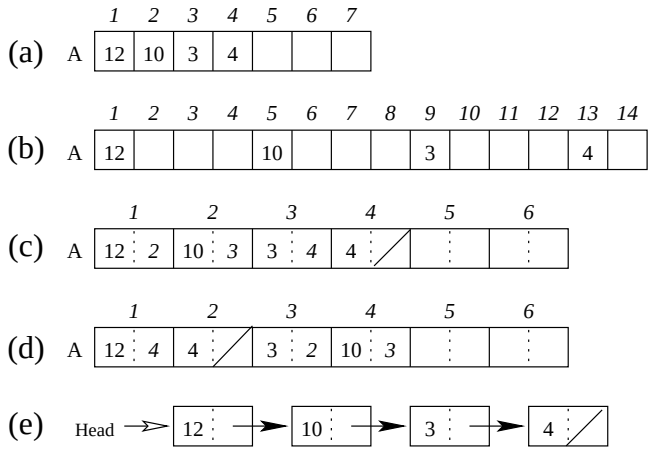


Figure 2.2: From a static array to a flexible linked list

objects with two fields, one contains the data and the other is used to store the address of the next datum (an index of array A). Let the first field be `data` and the other, `next`.

For each datum, the field `next` points to the following datum, where the sign "/" represents the *end* of the linked set (i.e. the special value `null` in Java). For example, the next object after $\{12,2\}$ should be $\{10,3\}$.

Figure 2.2 (d) shows that the physical location of each datum does not really matter as long as the data link gives the correct order. Figure 2.2 (c) and (d) both have the same logical link (i.e. data order) as 2.2 (e). Now it is an easy matter to insert an element before 10. We could store the new datum at the first available physical location and modify the links of two nodes.

2.7.1 References, links or pointers

The link that we discussed in the previous section 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.

Example 2.5 *Here simple variables p and q are pointers and their values are 105 and 205.*

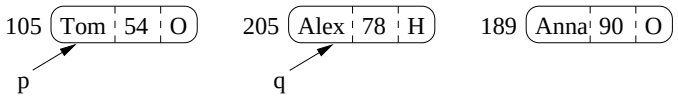


Figure 2.3: References or pointers

It is easy to move our focus to different record objects using pointers. For example, by a simple assignment $p \leftarrow q$, and $q \leftarrow 189$. we access two records as in Figure 2.4 (a) and (b) respectively.

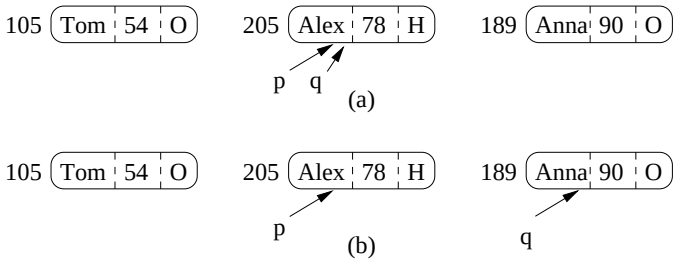


Figure 2.4: Pointers being updated

2.7.2 Implementation of the links

Pointers can be implemented in arrays.

Example 2.6 *In Figure 2.5, each object consists of two fields. The first field is the datum itself, and the second is a pointer which indicates the address of the following object.*

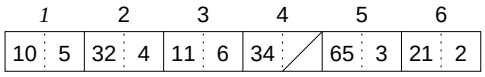


Figure 2.5: An implementation of pointers

Pointers are implemented more natually using the built-in pointer type in some languages. **Note** Unlike C or C++, Java does not have an built-in type as *pointer* explicitly. However, the *object* type is itself reference based. We can therefore easily define a node type that contains both the data and link fields in Java. In the following example, we use "item" and "next" to name the data and the link fields respectively.

Example 2.7 *The following Java source code can be used to define such a node type and some operations:*

```
import java.io.*;

// A class of Node which allows us to construct
// a node

// This defines a 'normal node' (type).
public class Node {
  private Object item;
  private Node next;

// This defines a node at the end of the list.
  public Node(Object newItem) {
    item = newItem;
    next = null;
  } // Constructor

// This is to create a 'normal' node.
  public Node(Object newItem, Node nextNode) {
    item = newItem;
```

```
    next = nextNode;
  } // Constructor

// This is to update the item field of a node.
  public void setItem(Object newItem) {
    item = newItem;
  } // end setItem

// This is to read the item field of a node.
 public Object getItem() {
    return item;
  } // end getItem

// This is to update the next field of a node.
 public void setNext(Node nextNode) {
    next = nextNode;
  } // end nextNode

// This is to read the next field of a node
 public Node getNext() {
    return next;
  } // end getNext()

} // end class Node
```

2.7.3 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.

Example 2.8 *(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. Figure 2.6 and 2.7 shows their data structures.*

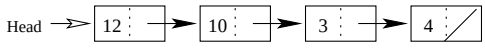


Figure 2.6: A list of integers

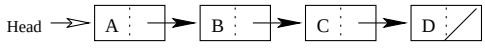


Figure 2.7: A list of characters

Following the *link* of a node, we can access its next node. Similarly, following the link of the next node we can access the next node of the next node. In this way, all the nodes can be accessed one after another.

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*:

Example 2.9 *((((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 Figure 2.8.*

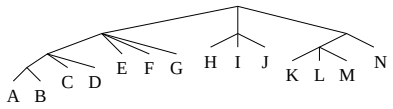


Figure 2.8: A list of lists

Example 2.10 *The arithmetic expression* $Q = ((a + b)(a + b))/((a - b)/(a - b))$ *can be represented by a list of lists as follows: .*

Let A be (a+b)(a+b) and B be (a-b)(a-b). We have $A = ((a, b, +), (a, b, +), *)$ *and* $B = ((a, b, -), (a, b, -), *)$*. Therefore,* $Q = (A, B, /) = (((a, b, +), (a, b, +), *), ((a, b, -), (a, b, -), *), /)$

2.7.4 Operations on lists

We write the functional description of these operations before developing the algorithms.

Typical operations (Linked lists):

initialise() create an empty list.

empty() check if the list is empty.

addOneNode() add one node to an existing list.

deleteOneNode() remove one node from the list.

printList() display the data field of each node in the list.

These are sufficient for us to demonstrate how to design algorithms. Of course, you can define as many operations or functions as you wish.

The next step is to look at each operation closely and design the algorithm for each of the above operations.

We first look at the two easiest algorithms, e.g. initialise() and empty(). They are both typed algorithms, `initialise()` makes the list *head* point to *null*, and `empty()` returns a `true` if the list head points to *null* and `false` if the list contains one or more nodes.

Algorithm 2.1 initialise()

1: $head \leftarrow null$;

Algorithm 2.2 boolean isEmpty(Node l)

1: return ($l = null$)

2.7.5 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.

First, we look at how to insert one 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 Figure 2.9 and Algorithm 2.3.

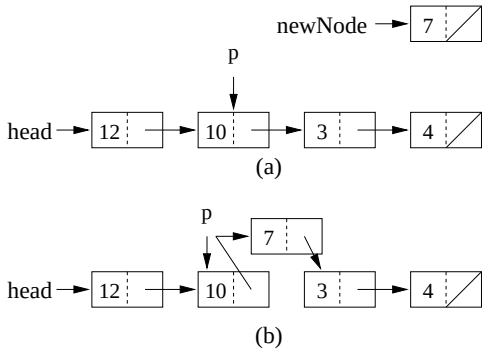


Figure 2.9: Add one node after the current node

Algorithm 2.3 addOneNodeAfter(Node p, newNode)

1: $newNode.next \leftarrow p.next$
2: $p.next \leftarrow newNode$

Next, we consider adding one node before the current one.

If we want to add one node before the current node, the address of the node immediately before the current one needs to be known.

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 Figure 2.10 and Algorithm 2.4. It is interesting to notice that the Algorithm 2.4 is actually unnecessary. Adding one node before a non-`head` node p is equivalent to adding the node after the node before p.

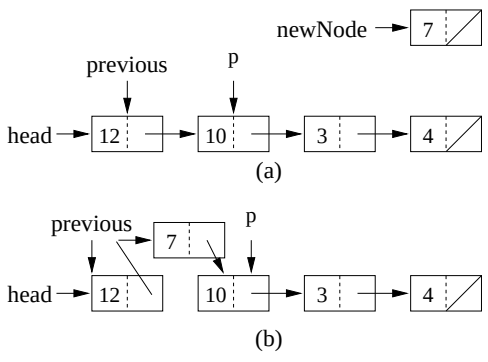


Figure 2.10: Add one node before the non-head current node

Algorithm 2.4 addOneNodeBefore(Node previous, p, newNode)

1: $newNode.next \leftarrow p$
2: $previous.next \leftarrow newNode$

However, since $p = previous.next$, the two statements in Algorithm 2.4 can be rewritten to:

1: $newNode.next \leftarrow previous.next$
2: $previous.next \leftarrow newNode$

This is equivalent to Algorithm 2.3. Finding the previous node is what we actually need. If we want to add one node before a node p, we only need to call Algorithm 2.3 with the current node pointing to the previous node of p.

2. The current node is the `head` of the list:
This can be seen from Figure 2.11 and Algorithm 2.5.

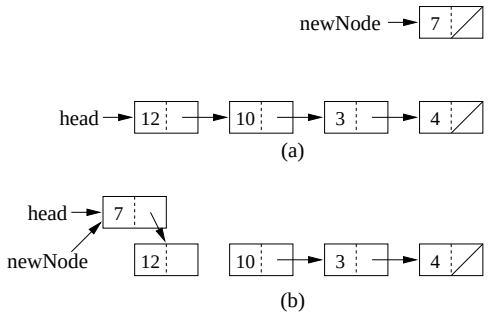


Figure 2.11: Add one node before `head`

Algorithm 2.5 addOneNodeBeforeHead(Node newNode)

1: $newNode.next \leftarrow head$
2: $head \leftarrow newNode$

2.7.6 Delete one node from a linked list

Suppose that p points to the current node. The deletion process can be seen from Figure 2.12, and Algorithm 2.6.

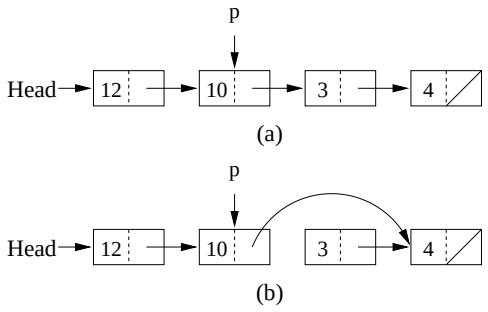


Figure 2.12: Delete one node after the current node

Algorithm 2.6 deleteOneNodeAfter(Node p)

1: $p.next \leftarrow p.next.next$

2.7.7 Implementation

Algorithms 2.1 – 2.6 can be implemented in Java as follows:

```
class List {
   Node head;

   public void initilise() {
      head = null;
   }

   public boolean empty() {
      return (head==null);
   }

   public void addOneNodeAfter(Node p, Node newNode) {
      newNode.next = p.next;
      p.next = newNode;
   }

   // unnecessary
   public void addOneNodeBefore(
                Node previous, Node p, Node newNode) {
      previous.next = newNode;
      newNode.next = p;
   }

   public void addOneNodeBeforeHead(Node newNode) {
      newNode.next = head;
      head = newNode;
   }

   public void deleteOneNodeAfter(Node p) {
      p.next = p.next.next;
   }
}
```

2.7.8 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.

Figure 2.13 shows how to construct the list in Figure 2.2(e) starting from the last node, adding a new node each time into the list by insertion at the front of the list.

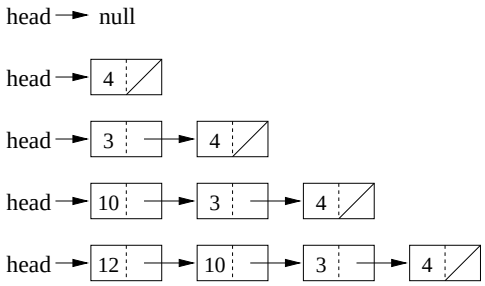


Figure 2.13: Construct a list by insertion at the front

The following example, in Figure 2.14, shows how to construct the list of Figure 2.2(e) starting from the first node, appending one node each time at the end of the list:

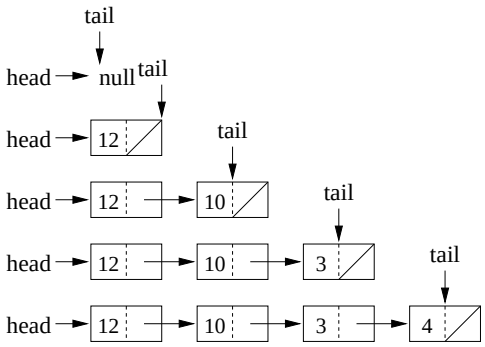


Figure 2.14: Construct a list by appendant at the tail

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.

2.7.9 Implementation

Example 2.11 gives two Java methods for construction of a linked list of integers. One is in a backwards fashion: `constructListFront()`, and the other is in a forward fashion: `constructListTail()` including a method to find the tail of a list `Node tail()`. The user of the program needs to input the integers from the keyboard. The construction process ends when the user inputs a '999'. The method `printList()` displays the content of a list which is useful for testing our methods.

Example 2.11

```
void constructListFront() {
   Scanner input = new Scanner( System.in );

   int x = input.nextInt();
   Node p = new Node( x );
   while ( x != 999 ) {
      addOneNodeBeforeHead( p );
```

```
      x = input.nextInt();
      p = new Node( x );
   }
}

Node tail() {
   Node p = head;
   while ( p.next != null ) {
      p = p.next;
   }
   return p;
}

void constructListTail() {
   Scanner input = new Scanner( System.in );

   int x = input.nextInt();
   Node p = new Node( x );
   Node t = tail();

   while ( x != 999 ) {
      addOneNodeAfter( t,p );
      t = t.next;
      x = input.nextInt();
      p = new Node( x );
   }
}

void printList() {
   Node p = head;
   while (p != null) {
      System.out.printf( " " + p.item );
      p = p.next;
   }
   System.out.println();
}
```

You need to write a main class to test the methods that we have seen so far. Example 2.12 shows such a main class that can be used to test the methods.

Example 2.12

```
public class linkList {
   public static void main( String argv[] ) {

   Node n = new Node(2);
   System.out.println(n.item);

   List l = new List();
   //  Because List and Node is defined in separate
   // class, the access a node has to be via a List.

   l.initilise();
   System.out.println("A empty list? "+l.empty());

   l.head = n;
   System.out.println("A empty list? "+l.empty());
```

```
   Node m = new Node(3);
   l.addOneNodeAfter(l.head, m);
   System.out.println(n.item + "  " + n.next.item);
   l.printList();
   l.constructListFront();
   l.printList();
   l.constructListTail();
   l.printList();
   }
}
```

A running example is as follows (`linked.lists]$` is the command prompt):

```
linked.lists]$ java linkList
2
A empty list? true
A empty list? false
2  3
 2 3
1
2
3
4
999
 4 3 2 1 2 3
1
2
3
4
5
6
7
999
 4 3 2 1 2 3 1 2 3 4 5 6 7
linked.lists]$
```

2.7.10 Other operations

These operations are mainly for routine maintenance of a list. For example, `traversal()` represents a group of operations that require an access every node of a list. Examples of traversals include updating each member of the list, and counting the number of elements in a list.

It is not difficult to write programs (methods) in Java for these operations. They are similar to the method `printList()` in Example 2.11 that displays the content of a list. We leave the implementation of some of these algorithms in Java as exercises.

2.7.11 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 kth element in an array but it would take k steps from the head of the list before accessing the kth node.

Activity 2.7

IMPLEMENTATION OF LINKED LISTS

1. Review the concept of a *constructor* in Java.

   For example, what is a constructor?
   How can we use a constructor to define a new type?
   What do we have to take care of when writing a constructor method, in terms of its name and of polymorphism?

2. Analyse the following class. What do the constructors do?

```
public class Node {
  private String item;
  private Node next;

  public Node(String newItem) {
    item = newItem;
    next = null;
  }

  public Node(String newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
  }
}
```

3. Using the constructor approach, construct a new class of data structure `Node` (for a linked list) which contains two fields of different type:

   - `item`: of Object type (this means it could be of anything)
   - `next`: of Node type.

4. Analyse the statements below. What do they do in each case?

   (a) `Node n = new Node("pen");`

   (b) `Node n = new Node("pencil", null);`

   (c)
```
Node n = new Node("pen", null);
n = new Node("pencil", null);
```

   (d)
```
Node n = new Node("an apple?");
n = new Node("ate", n);
n = new Node("cat", n);
```

   (e)
```
public class Node {
  private String item;
  private Node next;

  public Node(String newItem) {
```

```
    item = newItem;
    next = null;
  } // Constructor

  public Node(String newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
  } // Constructor
} // end Node

public class list {
  public static void main(String [] args) {
    Node n = new Node("an apple?");
    n = new Node("ate", n);
    n = new Node("cat", n);
  } // end main
} // end list
```

5. Following the above, write and add four methods which allow access to the value of each field in the class `Node`. For example, you may write the following four methods and include them into the class:
   - `setItem(Object newItem)`: to set the value of `item` field, i.e. to write the `newItem` in the `item` field of a node.
   - `getItem()`: to read the value of `item` field.
   - `setNext(Node nextNode)`: to set the value of `next` field, i.e. to write the `nextNode` in the `next` field of a node.
   - `getNext()`: to read the value of `next` field.
6. Write a method that displays all the items in a list.
   Hint: the method takes the head of a list as the input and displays the `item` field of each node in the list.
7. Write a method that constructs an empty list (of Objects).
   Hint: the method assigns a `null` to the head of the list.
8. Write a method that checks if a list is empty.
   Hint: the method takes the head of a list and returns *true* if the head has a `null` value.
9. Write a *recursive* method to count the number of nodes in a list.
   Hint: the method takes the head of a list and returns the number of nodes in the list.
10. Using the approaches in this lab exercise, implement an ADT *appointmentBook*.
    Hint: you should first define the interfaces for the ADT *appointmentBook*, and then implement the necessary data structures and basic operations.

2.8 Stacks

A *stack* is a restricted list in which entries are added to and removed from one designated end called the *top*. The other end is called *bottom*. A stack follows the *last in first out* (LIFO)[4] principle. It is like a pile of plates. The only plate that can be removed is the top one and the only place to add a plate is to add one on top of the pile.

[4] pronounced '`lie-foe`'.

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. We shall see this more clearly later in Section 2.8.3.

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. Figure 2.15 shows a stack of integers with a '1' on the top.

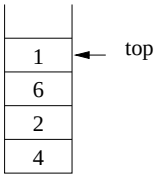


Figure 2.15: A stack

2.8.1 Operations on stacks

Like linked lists, stacks are dynamic data structures. That is, the size of a stack may change and the data come and go. We can define certain standard operations to, for example, initialise, add an element to, delete a datum from and check the content in a stack:

Typical operations (Stack):

initialise() initialise a stack.

push(x) add a datum `x` to the top of a stack. We say 'a datum is *pushed* onto the top of the stack'.

pop() remove a datum from the top of a stack. We say 'a datum is *popped* off the stack'.

empty() return *true* if the stack is empty, *false* otherwise.

top() check the value of the top of a stack but do not remove it from the stack.

full() check if the stack is full (for array implementations).

Stacks are sometimes called *LIFO queues* (see Section 2.9 for queues).

2.8.2 Implementation

1. By an array
   A stack can be implemented by an array. We define, for example, a class `stack` that consists of an array `stackArray[0..max-1]` (where `max` is some predefined upper index limit), a special pointer variable `top`, and the following operations (Algorithms 2.7–2.11):

Algorithm 2.7 initialise()

1: $top \leftarrow 0$

Algorithm 2.8 boolean empty()

1: return ($size$ = 0)

Algorithm 2.9 boolean full()

1: return ($size$ = max)

Algorithm 2.10 push(Object x)

1: **if** not full() **then**
2: $top \leftarrow top + 1$
3: $stackArray[top] \leftarrow x$
4: **end if**

Algorithm 2.11 pop(Object x)

1: **if** not empty() **then**
2: $x \leftarrow stackArray[top]$
3: $top \leftarrow top - 1$
4: **end if**

2. By a linked list
   Here we define a class `stack` that consists of a linked list with a special pointer variable `top` that points to the head of the list (Figure 2.16), and the following operations:

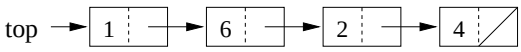


Figure 2.16:

The standard operations can be defined as follows (Algorithms 2.12–2.15):

Algorithm 2.12 initialise()

1: $top \leftarrow null$

Algorithm 2.13 boolean empty():

1: return (top = $null$)

Algorithm 2.14 push(Object newNode)

1: $newNode.next \leftarrow top$
2: $top \leftarrow newNode$

Algorithm 2.15 Object pop()

1: **if** not empty() **then**
2: $x \leftarrow top.data$, $top \leftarrow top.next$, return x
3: **end if**

You can write other commonly used operations using a similar approach.

2.8.3 Applications

Stacks have many important applications. For example, stacks can be very handy for checking for a correct number of nested closing brackets in a program or in normal text.

Example 2.13 *Consider the arithmetic expression* $\{[-(2+3)*2]+4\}/4$.

We can define a stack `brackets` *that is empty initially. The expression characters will be scanned starting from the first one. Let the* `current` *be the current character read. If the current character is a left bracket* `(`, `[` *or* `{`*, we push it into the stack; and pop the stack if it is a right bracket* `)`, `]` *or* `}`*. After scanning the whole expression, the stack* `brackets` *would be empty if there are same number of the left brackets as that of the right bracket. Report an error otherwise.*

Stacks are important especially in *operating systems* and *compilers*.

Example 2.14 *Keep the correct return locations of subprocedures.*

Figure 2.17 represents the subprocedure calls that commonly used in certain system programs. The vertical line represents the flow of execution of programs a–f. There are 2 subprocedure calls b and d at points 102 and 278, 1 subprocedure call in b at 21, two subprocedure calls in d at 100 and 189.

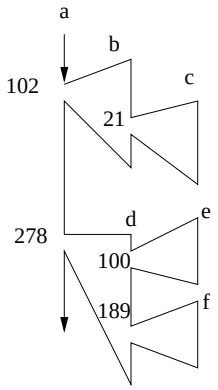


Figure 2.17: Procedure calls

Figure 2.18 shows that the leaving and return points of each program/procedure can be stored and retrieved using a stack. For example, the leaving point 102 is pushed into the stack when subprocedure b is called (1). Next 21 is pushed into the stack when subprocedure c is called (2), and it popped and used as the return address when procedure c is completed (3). Figures (4) to (10) shows the content of the stack at each of the remaining stages.

In a real operation system not only the leaving and return points of the procedure calls, but also the entire environment including the values of local variables are stored in and retrieved from stacks.

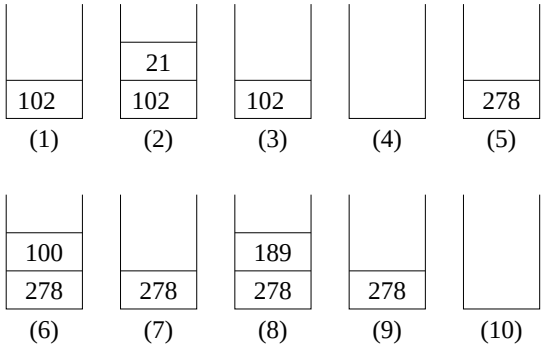


Figure 2.18: Stack content at each stage

2.9 Queues

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.

A queue is a data structure that follows the *first in first out* (FIFO[5]) principle. Objects are added to the rear of the queue, and are removed from the front of the queue. The concept of *queues* is used in computing as they are in real life.

[5] pronounced 'fie-foe'.

Example 2.15

- *A queue of people waiting for a bus.*
- *A queue of printing jobs sharing one printer.*

2.9.1 Operations on queues

Two addresses are 'remembered' for a queue, the *rear* (R) and the *front* (F).

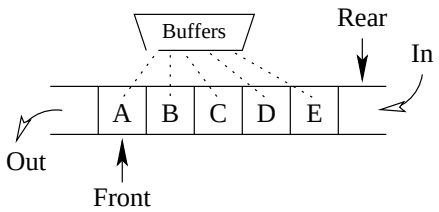


Figure 2.19: A queue

Typical operations (Queue):

empty() *true* if the queue is empty, else *false*.
enqueue(x) add an element `x` at the rear end of the queue.
dequeue() remove an element `x` from the front end of the queue.

2.9.2 Implementation of queues

Similarly to stacks, queues can be implemented by both arrays and linked lists.

Suppose F and R are variables holding the addresses of the front and rear of a queue respectively. Figure 2.20 shows the states (a) before and (b) after *adding* one item '4' to the queue implemented by an array[1..10] of integers.

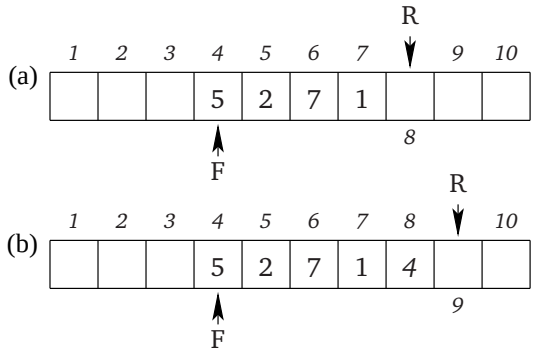


Figure 2.20: enqueue('4')

As we can see, in (a) the rear of the queue R pointed to address 8, an empty cell, before in (b) '4' being added at the rear and R being updated to index address 9.

1. Implementation by an array (Algorithms 2.16–2.20):

Algorithm 2.16 initialise()

```
1: size ← 0
2: front ← 1
3: rear ← 0
```

Algorithm 2.17 boolean empty()

```
1: return size = 0
```

Algorithm 2.18 boolean full()

```
1: return size = max
```

Algorithm 2.19 enqueue(Object x)

```
1: if size < max then
2:     size ← size + 1
3:     rear ← (rear mod max) + 1
4:     arrayQueue[rear] ← x
5: end if
```

You may have noticed the modulus computation in Step 3 in Algorithm 2.19 and 4 in in Algorithm 2.20.
Why do not we just simply increase `rear` by 1 or decrease `front` by 1 instead?

Algorithm 2.20 dequeue(Object x)

```
1: if size > 0 then
2:     size ← size − 1
3:     x ← arrayQueue[front]
4:     front ← (front mod max) + 1
5: end if
```

One potential problem with that simple approach is that after a number of `enqueues`, the rear may reach *max*. The queue appears to be full with empty cells in front of the array as in Figure 2.21(a). An easy solution for this problem is that whenever `front` or `rear` gets to the end of the array, it is cycled around to the beginning of the array as in Figure 2.21(b). The modular computation realises this process.

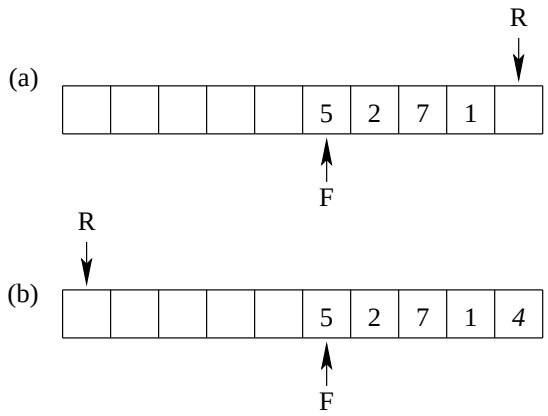


Figure 2.21: R has reached max and is cycled to the beginning of the queue array

The queue implemented in this way is sometimes called a *circular* queue, since the front and the rear can continue to be moved without any boundary as in Figure 2.22.

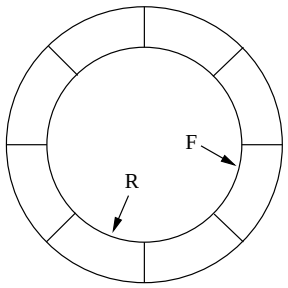


Figure 2.22: A circular queue

2. Implementation by a linked list (Algorithms 2.21–2.23):

Algorithm 2.21 initialise()

```
1: front ← null
2: rear ← null
```

You may define more operations for your convenience.

Algorithm 2.22 enqueue(Node p)

```
1: if p ≠ null then
2:    if front = null then
3:       front ← p
4:       rear ← p
5:    else
6:       rear.next ← p
7:       rear ← p
8:    end if
9:    p.next ← null {The end of the list}
10: end if
```

Algorithm 2.23 dequeue(Node p)

```
1: if front ≠ null then
2:    x ← front.data
3:    p ← front
4:    front ← front.next
5:    if front=null then
6:       rear ← null
7:    end if
8: end if
```

2.9.3 Variation of queues

There are useful data structures derived from the basic structures.

For example, a *priority queue* is a queue in which the entries are inserted into a position according to some priority criterion, rather than the arrival order only. We shall see priority queues (also called *Heaps*) later (Section 4.4).

A *deque* is a double-ended queue in which objects can be added or removed at either end of the queue.

The operations of a deque are similar to a normal queue, for example:

Typical operations (Deque):

push(x) insert object `x` on the front end of the deque;
pop() remove the front object of the deque and return the value;
inject(x) insert object `x` on the rear end of the deque;
eject(x) remove the rear object and return the value
empty() *true* if the deque is empty, else *false*.

Note the difference between the term `dequeue` and `deque`: `Dequeue` is an operation on a queue, and `deque` is a special queue that allows the addition or deletion of an element from both ends.

Activity 2.9

STACKS AND QUEUES

1. Write a program that outputs a sequence of strings in the *reverse* order of entry using a pointer implementation of *stacks*. The program keeps taking lines of strings from the keyboard until the string "end-of-input" is input.

   Example 2.16

```
Please input a few lines of strings
          (ended by 'end-of-input') >
> I like
> programming. How
> about you?
> end-of-input

The reversed sequence of strings is:
: about you?
: programming. How
: I like
```

   Hint

   (a) Write the main program for the problem using standard stack commands `push, pop, initialise, empty`, etc..

   (b) Write the methods or classes one by one.

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.

   Example 2.17

```
Please input a few lines of strings
          (ended by 'end-of-input') >
> I like
> programming. How
> about you?
> end-of-input

The sequences of strings are:
: I like
: programming. How
: about you?
```

2.10 Hashing

Hashing is another technique of storing and retrieving data. The data structure involved is essentially a one-dimensional array called *hash table*. Similar to the data structure for Bucket sort in Section 6.11, the index location of 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. As we can see from Example 2.18 below, a hash code can be calculated easily given the hash function and the key.

Example 2.18 *Assume that a hash function* $h(k) = k$ *mod* 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:

$$h(7) = 7 \text{ mod } 11 = 7$$

The given hash function is a modular function. To compute 7 *mod* 11*, we need to first compute* 7 *div* 11 = 0 *and then calculate the remainder* $7 - 0 \times 11 = 7$.

Similarly, we have

$$h(31) = 31 \text{ mod } 11 = 9$$
$$h(159) = 159 \text{ mod } 11 = 5$$
$$h(189) = 189 \text{ mod } 11 = 2$$
$$h(23) = 23 \text{ mod } 11 = 1$$
$$h(6) = 6 \text{ mod } 11 = 6.$$

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[$h(k)$].

Example 2.19 *Suppose that we search for key* $k = 23$*. We check the location and find* $h(23) = 23$ *mod* 11 = 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.*

Example 2.20 *Suppose that we search for key* $k = 50$*. We check the location and find* $h(50) = 50$ *mod* 11 = 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*[6].

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

Observation

As we see from the Example 2.18:

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.

2.10.1 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. Let us look at another example:

Example 2.21 *Suppose that the hash function* $h(k) = k \bmod 11$ *is used again and the hash table is empty initially. Show the content of the hash table after inserting the data (29, 93, 31, 159, 51, 189, 27, 23, 17, 9).*

Again we compute the hash codes first and get: (7, 5, 9, 5, 7, 2, 5, 1, 6, 9). We found duplicate hash codes, such as, two 7s, three 5s and two 9s. This means that at least two data, for example, 93 and 159, are allocated to the same address 5 (So do (29 and 51); and (93, 159, 27)).

We say that 'data 93 and 159 are collided' meaning that they are both mapped to the same address (5 in this example). This is the so-called *collision problem* in hashing.

2.10.2 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.

The first approach is called *closed address hashing*, for it does not consume any extra addresses of the hashtable. The number of addresses of the hashtable will remain the same.

The second approach is called *open address hashing*, for the number of addresses of the hashtable may be increased.

2.10.2.1 Closed address hashing

Closed address hashing is one easy solution for collision. The method chains the collided data together, using an array of linked lists. The size of the hash table remains the same during closed address hashing.

Example 2.22 *Suppose the hash function is* $h(k) = k$ *mod* 11*, and the rehash function is then* $h(k) = (k + 1)$ *mod* 11*. The hash table is empty initially. Show the content of the hash table after inserting the data (29, 93, 31, 159, 51, 189, 27, 23, 17, 9).*

Solution *Again we compute the hash code(s) first and get: (7, 5, 9, 5, 7, 2, 5, 1, 6, 9).*

Collisions occur since 159 and 27 have the same hash code as for 93; 51 has the same hash code as that for 29, and 9 has the same hash code as for 31. We link, therefore, 93, 159 and 27 together, link 29 and 51 together, and link 31 and 9 together as follows.

The content of the hashtable is (where symbol '↓' represents a link):

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
H[i]		*23*	*189*			*93* ↓ *159* ↓ *27*	*17*	*29* ↓ *51*		*31* ↓ *9*	

During the retrieval process, not only each hash cell but each linked list will also be searched.

As we can see, the addresses of the hashtable remain the same despite extra storage space required for the linked lists.

2.10.2.2 Open address hashing

This approach is to store 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.

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 h_1 is occupied, then $h_1 + 1$ is offered if it is empty, otherwise, the next location $h_1 + 2$ is offered, and so on.[7]

[7] 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.

This is equivalent to using a similar hash function for rehashing when there is a collision:

$$rh(k) = (k + 1) \bmod h$$

Example 2.23 *Suppose the hash function is* $h(k) = k$ *mod* 11*, and the rehash function is then* $h(k) = (k + 1)$ *mod* 11*. The hash table is empty*

initially. Show the content of the hash table after inserting the data (29, 93, 31, 159, 51, 189, 27, 23, 17, 9).

Solution *Again we compute the hash code(s) first, using the hash function* $h(k) = k$ *mod* 11 *and get: (7, 5, 9, 5, 7, 2, 5, 1, 6, 9).*

Found the collisions for 159, 51, 27, 17 and 9 at hash cell $H[i]$*, we rehash each of them by probing the next hash cell* $H[i+1]$*. This process continues until each of them can be placed in a free hash cell. For example, since cell* $H[5]$ *is occupied, we probe cell[6] and found it available, so place 159 to cell* $H[6]$.

Similarly, 51 is placed in cell $H[8]$ *after a collision is found in cell* $H[7]$.

There are 5 linear probes, at cell location $6, 7, 8, 9, 10$ *before placing 27 at cell* $H[10]$.

There are 5 probes at cell location 6–10 before placing 17 at cell $H[0]$.

Finally, there are 5 probes at cell location 10–3 before placing 9 at cell $H[3]$.

The process can be described precisely as to rehash 159, 51, 27, 17 and 9 as follows:

$$rh(159) = (159 + 1) \textit{ mod } 11 = 160 \textit{ mod } 11 = 6\ (1\ probe)$$

$$rh(51) = (51 + 1) \textit{ mod } 11 = 52 \textit{ mod } 11 = 8\ (1\ probe)$$

$$rh_1(27) = (27 + 1) \textit{ mod } 11 = 6, rh_2(27) = (27 + 2) \textit{ mod } 11 = 7,$$

$$rh_3(27) = (27 + 3) \textit{ mod } 11 = 8, rh_4(27) = (27 + 4) \textit{ mod } 11 = 9,$$

$$rh_5(27) = (27 + 5) \textit{ mod } 11 = 10\ (5\ probes)$$

$$rh_1(17) \cdots rh_5(17) = (17 + 5) \textit{ mod } 11 = 0\ (5\ probes)$$

$$rh_1(9) \cdots rh_5(9 + 5) = 14 \textit{ mod } 11 = 3\ (5\ probes)$$

The content of the hashtable is (the data in bold which are rehashed to the location.):

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
H	***17***	*23*	*189*	***9***		*93*	***159***	*29*	***51***	*31*	***27***

As we can see from the table below (where symbol '⇓' represents a rehash), while 29, 93, 31, 189 and 23 can be stored in the hashtable immediately, 159 and 51 have to be rehashed to the next locations $H[6]$ and $H[8]$ after one probe. In contrast, 27, 17 and 9 have to be rehashed to the location $H[10]$, $H[0]$ and $H[3]$ after 5 probes:

k	29	93	31	159	51	189	27	23	17	9
h(k)	7	5	9	5	7	2	5	1	6	9
				⇓	⇓		⇓		⇓	⇓
$rh_1(k)$				6	8		6		7	10
							⇓		⇓	⇓
$rh_2(k)$							7		8	0
							⇓		⇓	⇓
$rh_3(k)$							8		9	1
							⇓		⇓	⇓
$rh_4(k)$							9		10	2
							⇓		⇓	⇓
$rh_5(k)$							10		0	3

Note the overlaps between the rehashing sequences for 159, 51, 27 and 17, especially 27 and 17. They hash and rehash to the same locations $H[5 \cdots 10]$. This is called *clustering*. *Primary clustering* occurs when a number of different valued keys hash to the same location and rehash to locations with collisions with the same set of keys, such as for 159 and 27. *Secondary clustering* occurs when keys that initially hash to different locations eventually rehash to the same sequence of locations, such as for 27 and 17.

To avoid secondary rehashing, we introduce a rehashing technique called double hashing next.

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.

Example 2.24 *Suppose the hash function is* $h(k) = k$ *mod* 11*, and the rehash function is* $h(k) = k$ *mod* 13*. The hash table is empty initially. Show the content of the hash table after inserting the data (29, 93, 31, 159, 51, 189, 27, 23, 17, 9).*

Solution *Again we compute the hash code(s) first and get: (7, 5, 9, 5, 7, 2, 5, 1, 6, 9).*

Rehash 159, 51, 27, 23 and 9:

$$rh(159) = 159 \text{ mod } 13 = 3$$

$$rh(51) = 51 \text{ mod } 13 = 12$$

$$rh(27) = 27 \text{ mod } 13 = 1$$

$$rh(23) = 23 \text{ mod } 13 = 10$$

$$rh(9) = 9 \text{ mod } 13 = 9$$

The content of the hashtable becomes (the data in bold which are rehashed to the location.):

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
H		***27***	*189*	***159***		*93*	*17*	*29*		*31*	***23***		***51***
										9			

As we can see, the double hashing is more efficient than linear probing in this example. After one rehashing, only 9 is still collided with 31. This means that one probe is not sufficient using the double hashing. In this case, we need to take another approach to resolve the collision for 9. For example, we can apply linear probing on either of the hash functions, or apply an alternative rehash function. We shall leave you to solve the collision for 9.

2.10.3 Extra work for retrieval process

A hash function that produces a hashtable without any collision is called a *perfect hash function*. A perfect hash function requires a good prior knowledge of the set of potential keys. Two states are sufficient for each cell of the hashtable, namely *empty* and *occupied*.

Most hash functions are imperfect unfortunately so rehashing are required, as we see in the previous sections. When a hashtable is searched, the third state is needed for each cell in order to indicate that the data have been rehashed. Let the third state be `rehash`. All the rehash sequence of cells have to be flagged for the rehashed keys.

Example 2.25 *Consider the process of searching key 159 in Example 2.23.*

We first find the location for 159: $h(159) = 159 \bmod 11 = 5$. *Since the state is* `rehash`*, we proceed rehashing to probe location 6. This time, the state of the cell is* `occupied`*, we know there is no further rehashing is required and the key is found in* $H[6]$ *since* $H[6] = 159$.

The situation, however, requires further indication since cell $H[6]$ is the actual key for 159, but is one of the rehash cells for 27 and 17. In this case, the state of each cell needs to be key oriented as well. That is, the state of the same cell should depend on the associated key. For example, for key 27 and 17, the state of cell $H[6]$ should be assigned to `rehash`, but the key 159, it should be assigned to `occupied`.

2.10.4 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.

Note the word *may* used. It does not say that the size of the hashtable will definitely grow by these approaches. It depends on the data, the occupation state of the hashtable, and the hash functions. It is possible that the original hashtable is not extended after certain rehashing. For example, if the the data in a hashtable is sparse, a perfect hashing may be possible. Similarly, all the collisions may be resolved within the address range of the original hashtable, and no extra cell is required for certain data and hash functions. This, however, is different from closed address hashing where the size of the hashtable is fixed and there is definitely no change with the number of hash cells.

Activity 2.10

HASHING

1. What is it meant by *Hashing*?
2. What are the hash code and a hash function? How are they used in the hashing technique? Give an example of a hash code and a hash function.
3. In the context of hash addressing, what is a collision? Why are collisions undesirable and why are they usually unavoidable?
4. Describe, with an example, the methods of closed address hashing, of linear probing and of double hashing.

Chapter 3

Algorithm design techniques

3.1 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

3.2 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 and implement simple recursive programs
- Explain the dynamic programming and the greedy approach with an example

3.3 Recursion

We know that a program can call another subprogram. Can a subprogram call itself?

A subprogram can be referred as a *method* in Java. In most conventional computer languages, a method in Java, procedure or function in C can call itself. A subprogram that calls itself is said to be *recursive* or referred to as a *recursive* method, procedure or *recursive* function respectively.

Recursion is a powerful algorithmic tool for problem solving. As we shall see later, recursion is another way to realise repetition.

In Mathematics, recursion is a process (or phenomenon) of finding solutions to a sequence of subproblems that are identical to the original problem but smaller in problem size. This characteristic can be identified easily from a function. 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 nth Fibonacci item (see Example 3.3) can be written as

$$f(0) = f(1) = 1$$

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

This is a recursive function: The first line of the specification gives the so-called *base case*, and in the second line of the specification, 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 $n-2$. (Note the process from n to $n-1$, to $n-2$, $\cdots$, and finally to 1 is inefficient for a large n. This makes the recursive algorithm inefficient.)

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.

Many problems can be solved by recursion.

Example 3.1 *Given a non-negative integer n, compute n factorial.*

Solution

```
0!=1;
1!=1;
for n>1, n!= n * (n-1)!
```

For instance, when $n = 5$,

$$\begin{aligned} 5! &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \\ &= 5 * 4 * 3 * 2 * 1 \end{aligned}$$

The solution here shows that once some factorial has been found, it can be used to compute the next (bigger) factorial. In other words, The n factorial is known if the previous $n-1$ factorial is known.

Example 3.2 *Given n, an integer ≥ 0, and x, a real, compute the n consecutive powers of some number x.*

Solution

$n = 0$, $x^n = 1$

$n > 0$, $x^n = x * x^{n-1}$

For instance, when $x = 3$, $n = 5$,

$3^0 = 1$;
$3^5 = 3 \times 3^4$, $3^4 = 3 \times 3^3$, $3^3 = 3 \times 3^2$, $3^2 = 3 \times 3^1$, $3^1 = 3 \times 3^0 = 3 \times 1$.

In general, a function is said to be defined *recursively* if its definition consists of the following two parts:

1. Base case
   This must be a well defined termination
2. Inductive or recursive steps
   This consists of well defined inductive (or recursive) steps that must lead to a termination state.

Question: In previous examples, which are the *base cases* and which are the *inductive steps*?

Example 3.3 *Let us look at the well-known Fibonacci sequence* `1, 1, 2, 3, 5, 8, 13, 21, 24, 34, ...`. *Fibonacci sequence starts with two 1s. Each Fibonacci number thereafter is the sum of the two preceding numbers.*

This infinite sequence can be defined recursively by
$a_1 = 1$, $a_2 = 1$ *(Base case)*
$a_n = a_{n-1} + a_{n-2}$, *for* $n > 2$ *(Recursive step)*.

Recursive programs

A procedure or function can be easily interpreted from the recursive formulae discussed previously (Algorithm 3.1 – 3.3):

Algorithm 3.1 int fibonacciItem(int n); (where $n > 0$)

```
1: if (n = 1) or (n = 2)) then
2:     return 1 {where n > 0}
3: else
4:     return fibonacciItem(n − 1) + fibonacciItem(n − 2)
5: end if
```

Algorithm 3.2 int factorial(int n)

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

Note The examples of recursion discussed here are actually so called *direct recursion* which means that the procedures or functions have referenced themselves directly.

Another type of recursion is *indirect recursion* which occurs when a subprogram does not directly call itself but it is called eventually after a chain of other subprogram calls. For example, procedure A calls procedure B, which calls procedure C, which calls procedure A again.

Algorithm 3.3 real power(real x, int n)

```
1: if (x = 1) or (n = 0) then
2:    return 1
3: else
4:    if (n > 0) then
5:       return x × power(x, n − 1)
6:    end if
7: end if
```

Although we do not cover any details of the topic in the course, it is important for you to know that indirect recursions are possible.

3.3.1 Implementation

Algorithms 3.1–3.3 can be implemented in Java as follows:

```
static int fibonacciItem(int n) {
  int f=0;
  if (n=1) || (n=2) {
     f=1;
  }
  else
     {
     f=fibonacciItem(n-1)+fibonacciItem(n-2);
     }
  return f;
}

static int factorial(int n) {
  int f=0;
  if (n<2) {
     f=1;
  }
  else
     {
     f=n*factorial(n-1);
     }
  return f;
}

static double power(double x, int n) {
  double p=0.0;
  if ((x==1) || (n==0)) {
     p=1;
  }
  else
     if (n>0) {
         p=x*power(x, n-1);
     }
  return p;
}
```

3.3.2 What happens

What happens when a subprogram calls itself? We discuss this in Example 3.4.

Example 3.4 *Given an integer n, compute $\sum_{j=1}^{n} j$.*

Solution

Algorithm 3.4 int sigma(int n)

1: **if** $n \leq 0$ **then**
2: return 0
3: **else**
4: return $n + sigma(n-1)$
5: **end if**

Algorithm 3.4 can be implemented as follows in Java:

```
static int sigma(int n) {
  int f=0;
  if (n<1) {
     f=0;
  }
  else {
     f=n+sigma(n-1);
     }
  return f;
}
```

Two phases are involved during the execution of a recursive procedure or function. It is no exception for our example.

1. *Keep calling the recursive procedures or functions until a base case is reached.*

$$\begin{aligned}
\sum_{j=1}^{5} j &= 5 + \sum_{j=1}^{4} j \\
&= 5 + 4 + \sum_{j=1}^{3} j \\
&= 5 + 4 + 3 + \sum_{j=1}^{2} j \\
&= 5 + 4 + 3 + 2 + \sum_{j=1}^{1} j \\
&= 5 + 4 + 3 + 2 + 1 + \sum_{j=1}^{0} j \\
&= 5 + 4 + 3 + 2 + 1 + 0
\end{aligned}$$

2. *Keep replacing procedures or functions by the values returned.*

$$\sum_{j=1}^{5} j = 5 + 4 + 3 + 2 + \underbrace{1 + 0}_{1}$$

$$
\begin{aligned}
&= 5 + 4 + 3 + \underbrace{2 + 1}_{3} \\
&= 5 + 4 + \underbrace{3 + 3}_{6} \\
&= 5 + \underbrace{4 + 6}_{10} \\
&= \underbrace{5 + 10}_{15} \\
&= 15
\end{aligned}
$$

As we have seen, calculating function values may require considerable bookkeeping to record information at the various levels of the recursive evaluation. The information can be used to backtrack from one level to the preceding one. Of course, the compiler generates automatically all the necessary bookkeeping and backtracking support in most modern high-level languages.

Note Frequently, it is extremely difficult to track down the actual sequence of recursive calls. Therefore, the normal approach to understanding a program based on tracing its behaviour should *not* be used for recursive programs. We should try to find the base case and the inductive formula instead.

In principle, a recursive subprogram can be rewritten in a non-recursive manner. The following procedure is to compute $\Sigma_{j=1}^{n} j$ using a simple `for` loop.

Example 3.5

Algorithm 3.5 int sigmaIteration(int n)

1: $sum \leftarrow 0$
2: **for** $j \leftarrow 1, j \leq n, j \leftarrow j + 1$ **do**
3: $\quad sum \leftarrow sum + j$
4: **end for**
5: return sum

It can be implemented in Java:

```
static int sigmaIteration(int n) {
   int sum= 0;
   for (int j=1; j<=n; j++){
      sum = sum + j;
   }
   return sum;
}
```

In practice, however, it is sometimes difficult to solve a problem without recursion. This is, in fact, one of the reasons of writing recursive procedures and functions.

3.3.3 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. We may derive an iterative version later, based on the recursive one. In this way, a more efficient algorithm may be developed. the Quicksort algorithm.

It would be difficult to solve a problem such as *Towers of Hanoi* without the use of recursion. We only discuss briefly the recursive solution here.

Example 3.6 *(Towers of Hanoi problem) This is a game with a long history. There are 3 pegs (A, B, C) and a set of n disks of n different sizes. We mark them $1 \cdots n$ 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. Figure 3.1 shows that three disks on peg A need to be moved to peg C.*

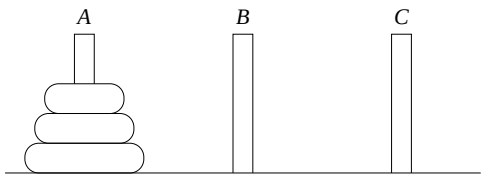


Figure 3.1: Towers of Hanoi problem

Let us summarise the problem again:

- *Consider three pegs (A, B, C) and 3 disks (1, 2, 3) as in Figure 3.1, with disk 1 the smallest, and 3 the largest.*
- *Objective: to move the disks from peg A to C;*
- *Two rules:*
  1. *one disk may be moved at a time*
  2. *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 3.2).

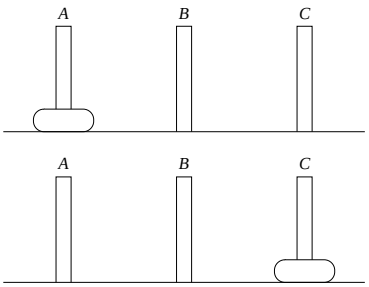


Figure 3.2: One disk

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. *then move disk 2 to peg C and*
      iii. *use the solution to the one-disk problem to move disk 1 to peg C (Figure 3.3).*

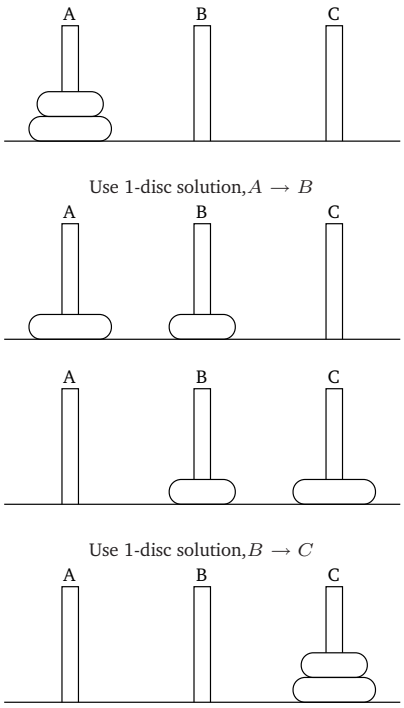


Figure 3.3: Two disks

   (b) *The solution for 3-disk problem ($n = 3$):*
      i. *Use the two-disk solution to have disk 1 and 2 in order on peg B;*
      ii. *then move disk 3 to peg C.*
      iii. *Finally, use the solution to move the two disks from peg B to C (Figure 3.4).*

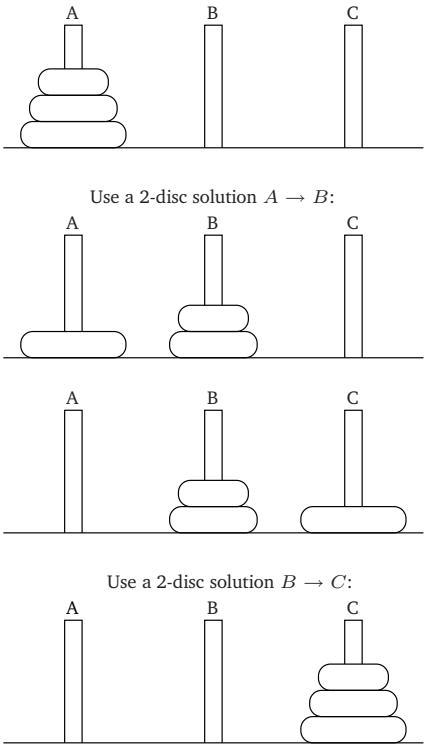


Figure 3.4: Three disks

(c) *The solution for 4-disk problem (n = 4) (Figure 3.5):*
   i. *Use 3-disk solution to move 3 disks (disk $1, 2, 3$) to peg B.*
   ii. *Move disk 4 to peg C.*
   iii. *Use 3-disk solution to move 3 disk from peg B to C.*

(d) *We can now write the solution for n-disk problem:*
   i. *Use $(n-1)$-disk solution to move $n-1$ disks (disk $1, \cdots, n-1$) to peg B.*
   ii. *Move disk n to peg C.*
   iii. *Use $(n-1)$-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 procedure (Algorithm 3.6) 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, SparePeg, LastPeg, StartPeg)
7: end if
```

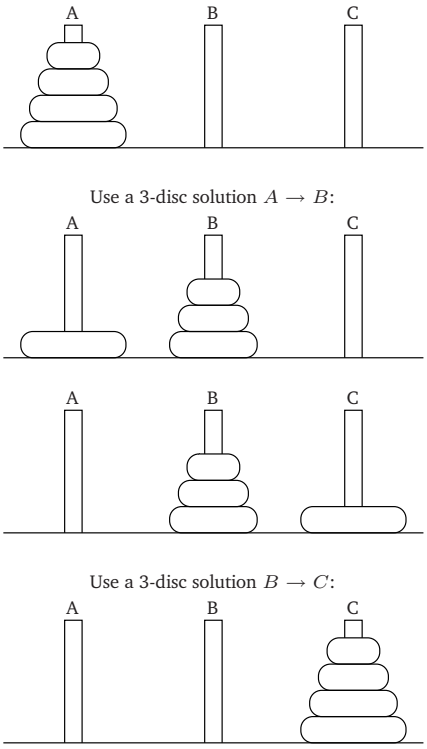


Figure 3.5: Four disks

In fact, this program can be simplified to Algorithm 3.7.

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, SparePeg, LastPeg, StartPeg)
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.

3.3.3.1 Linked lists as recursive data structures

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.

Example 3.7 *Recursive version of a linked list traversal (Algorithm 3.8).*

Algorithm 3.8 linkedTraverse(object head, processNode(object item, linkedListData))

```
1: if head ≠ null then
2:     processNode(head.data)
3:     linkedTraverse(head.next)
4: end if
```

3.3.4 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.

3.3.5 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 3.8 *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, Algorithm 3.9 can be simplified to Algorithm 3.10.

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
```

3.3.6 Common errors

It is very important for you to make sure that both base cases and inductive steps are correctly presented in recursive programs. We show some common errors in recursive programs written by students.

Error: Forgetting about the termination condition

Example 3.9 Error

The function below is meant to compute $\sum_{j=1}^{n} j$.

Algorithm 3.11 int sigma(int n)

```
1: sigma ← n + sigma(n − 1)
```

Correction

Algorithm 3.12 int sigma(int n)

```
1: if n ≤ 0 then
2:     sigma ← 0
3: else
4:     sigma ← n + sigma(n − 1)
5: end if
```

Error: Using redundant loops in recursive programs

Recursion is one way to achieve iteration. Iterations are achieved by recursive calls during the execution of the program. Usually, it causes errors if you use `for, while` and `repeat` in a recursive program.

Example 3.10 Error

The procedure below is meant to display the data in a linked list one by one.

Algorithm 3.13 display(node p)

```
1: if p = null then
2:     writeln(p.data)
3: else
4:     while p ≠ null do
5:         display(p.next)
6:     end while
7: end if
```

Correction

Algorithm 3.14 display(node p)

```
1: if p = null then
2:     writeln(p.data)
3: else
4:     display(p.next)
5: end if
```

Activity 3.3

RECURSION

1. Implement *one* of the recursive algorithms discussed in the Section 3.3 that:

   (a) computes the factorial of a non-negative integer

   (b) writes a character string backwards, i.e. in the reverse order

   (c) computes the *n*th term in a Fibonacci sequence.

   Note: you need to:

   (a) define the recurrence relationship

   (b) construct a class including a main method to check whether your methods work.

2. Using comment /*Recursive Call Here*/, mark each recursive call in your program (class).

3. Modify your program (class) above so it will report, e.g. display, Recursive Call(s) each time when a recursive method is called.

4. Using an iterative approach, write a non-recursive method for the same problem chosen above.

5. Compare the number of execution steps in the recursive and in the iterative method.

6. Design and implement a recursive algorithm to print a given string backwards.

3.4 Divide and conquer

This is a useful approach to solve an algorithmic computational problem. The name *divide and conquer* was used as the brilliant fighting strategy by the French emperor Napoleon in the Battle of Austerlitz on 2 December, 1805.

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[1]. 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.

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

If the solution to P_1 or P_2 is unavailable, P_1 or P_2 should be divided further into even smaller instances, e.g. the subproblems of the subproblem. 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.

Classical algorithms such as the *binary search*, *merge sort* and *quick sort* are good examples of the divide and conquer approach.

Binary search

Given a sorted list and *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.

Let the sorted list be $L[0..n-1]$ and the key be X. 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.

Let l, r be the index of the first (left most) and the last element (right most) of a list respectively. The middle index can then be defined as $mid = \lfloor (l + r)/2 \rfloor$ (Here $\lfloor x \rfloor$ reads 'floor of x', which rounds x to the nearest integer$\leq x$. For example, $\lfloor 2.96 \rfloor = 2$).

If $X < L[mid]$, the left half $L[l..mid-1]$ is selected for further check. Otherwise, the right half $L[mid+1..r]$. Initially, $l = 0$ and $r = n - 1$.

Example 3.11 *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	

The binary search algorithm can be found in Algorithm 5.8 in Section 5.6.

3.4.1 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.

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 it is easy to merge two random (unsorted) lists. For example, two random lists $(5, 1, 2)$ and $(4, 9, 6, 7)$ can be merged by placing one after another in either order to get a combined longer random list: $(5, 1, 2, 4, 9, 6, 7)$ or $(4, 9, 6, 7, 5, 1, 2)$.

However, merging two sorted lists cannot be achieved by simply appending one sorted list to another. For example, appending (21, 32) to (8, 34, 51, 64) will give (8, 34, 51, 64, 21, 32), a *unsorted* list. This process has changed the 'sorted' nature of the two original lists so it is not *merging* by our definition.

Merging two sorted lists is a combination process that must produce a *sorted* list. For example, two sorted lists $(8, 34, 51, 64)$ and $(21, 32)$ can be merged to become a longer sorted list $(8, \mathit{21}, \mathit{32}, 34, 51, 64)$. [2]

[2] The data from the second list are in italic.

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 3.12 *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*	*21 32*	
34 51 64	**21** *32*	*8*
34 51 64	**32**	*8 21*
34 51 64		*8 21 32*
		8 21 32 34 51 64

In each iteration, the `front` *elements of the two queues are compared, the smaller one (marked in bold) is then* `dequeued` *and* `enqueued` *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`.

We now consider the Merge-sort algorithm. The Merge-sort algorithm takes, as the input, a list of data that must be comparable in some way, and returns a sorted list. The list is often stored in an array and is divided according to the middle index. If a list is stored in an array $L[1..n]$, it is then divided to two sublists $lList[1..p]$ and $rList[(p+1)..n]$, where $p = \lceil n/2 \rceil$ (Here $\lceil x \rceil$ reads 'ceiling of x', which rounds x to the nearest integer$\geq x$. For example, $\lceil 2.26 \rceil = 3$). Alternatively, choose the floor $p = \lfloor n/2 \rfloor$.

Example 3.13 *Sort the list 26 33 35 29 19 12 22 using* `mergeSort` *in Algorithm 6.10.*

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 (35 29). 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

The mergeSort algorithm can be found in Algorithm 6.10.

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 3.14 *Given a 33 26 35 29 19 12 22, with the first element 33 is selected as a pivot, the pivot loation should be where 33 is: 26 29 19 12 22* $\boxed{33}$ *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 3.15 *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

We first select 33, the first element as the pivot. The original list is then divided into two sublists: $lList$ = `(26 29 19 12 22)` and $rList$ = `(35)`. Next the first element 26 of the `lList` is divided into $lList$ = `(19 12 22)` and $rList$ = `(29)`. Next 19 is the pivot and divides the $lList$ into $lList$ = `(12)` and $rList$ = `(22)`, each contains a single element, which is sorted. Now the 'bottom-up' process begins: the $lList$ and $rList$ is put together to get a sorted sublist: `(12 19 22)` in the format: `[lList] pivot [rList]`. The process continues until the whole list is sorted.

The `quickSort` algorithm can be found in Algorithm 6.11.

3.4.2 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.

Example 3.16 *Consider Algorithm 3.15, and 3.16 (a copy of Algorithm 3.1).*

Algorithm 3.15 int factorial(int n)

1: **if** ($n \leq 1$) **then**
2: $factorial \leftarrow 1$
3: **else**
4: $factorial \leftarrow n \times factorial(n-1)$
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 Algorithm 3.15, $factorial(n-1)$ 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 Algorithm 3.16, $fibonacciItem(n-1)$ 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:[3]

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

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 of these are solved 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* equition:

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

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

A recurrence equition cannot, in general, be solved easily, but we know that the $O(T(n))$ depends on the values of a, b and $f(n)$ of the divide and conquer method used, and a theorem called *Master Theorem* (or *Main Recurrence Theorem*) can be used to estimate $T(n)$, the solution to the general divide and conquer recurrence. Due to the time limit, we exclude the details here.

3.5 Dynamic programming

Dynamic programming is another important technique for algorithm design (like recursion, divide and conquer). Note the word *dynamic* is used to mean the dependent recurrent steps on each intermediate state, and the word *programming* here means arranging the partial results (to derive the final result). The name *dynamic programming* was used first by Richard Bellman, a mathematician in the US, in 1957 to describe control optimisation problems (Section 7.3).

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

3.5.1 Overlapped subproblems

Consider the Fibonacci problem assuming $n > 0$

$$f(1) = f(2) = 1$$

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

and the recursive Algorithm 3.17 (a copy of Algorithm 3.16):

In each $i > 2$, two previous terms $fibanacciItem(i-1)$ and $fibanacciItem(i-2)$ are required. For iteration $i-1$, two previous

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
```

terms $fibanacciItem(i-2)$ and $fibanacciItem(i-3)$ are required. That is,

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

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

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

$$\vdots$$

As we can see, the overlapping $fibanacciItem(i-2)$ 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.

3.5.2 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 in Section 3.5.1, 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 $fibanacciItem(i-1)$ and $fibanacciItem(i-2)$ are already available from the 'program' table.

Since the 'program' is constructed in a 'build-as-you-go' fashion, the approach is called 'dynamic programming'.

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 to Algorithm 3.17, 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 $i = 4 \cdots n$. In this way, each $fibonacciItem(i)$, for $i = 3 \cdots n$, only needs to compute once.

3.5.3 Efficiency of dynamic programming

Dynamic programming can be more time-efficient because repeating computations are avoided for overlapped subproblems. Note Algorithm 3.18 is implemented 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 Algorithm 3.18 uses an array, but this can be avoided sometimes, for example, as in Algorithm 3.19.

Algorithm 3.19 int fibonacciDP1(int n)

```
1: if n = 1 then
2:     return 1
3: else if n = 2 then
4:     return 1
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
```

3.5.4 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.

Let us look at Factorial items as an example.

Example 3.17 *For the divide and conquer approach (top down):*

$$\begin{aligned} 5! &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \end{aligned}$$

Example 3.18 *Dynamic programming approach (bottom up):*

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 * 1! \\ 3! &= 3 * 2! \\ 4! &= 4 * 3! \\ 5! &= 5 * 4! \end{aligned}$$

We shall see more examples of dynamic programming approach for optimisation problems in Chapter 7.

3.5.5 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.

Activity 3.5

DIVIDE AND CONQUER, AND DYNAMIC PROGRAMMING

1. What is divide and conquer? What is dynamic programming?
2. What are the main similarity of, and difference between, the divide and conquer approach and the dynamic programming approach?
3. Design a algorithm to find the largest element in an array of n numbers, using the divide and conquer approach.
4. Develop a algorithm to find the smallest elements in an array of n numbers, using the divide and conquer approach.
5. Implement and test your algorithms in questions 3 and 4.

Chapter 4

Abstract data types II: trees, graphs and heaps

4.1 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

4.2 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
- implement the binary tree data structure in Java or other languages
- describe some applications of binary trees.
- demonstrate how to represent a graph in computers
- describe the two main algorithms of graph traversal
- outline the algorithms for solving some classical graph problems.

4.3 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. Recall *arrays, linked lists, stacks and queues*. Each of these data structures represents a *relationship* between data. 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.

Example 4.1 *The file system in a user account (Figure 4.1).*

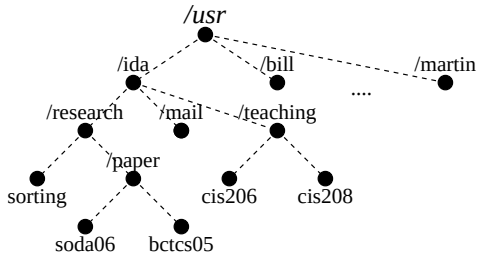


Figure 4.1: A tree structure in a file system

Trees are very rich in properties. We shall find out later that there are actually many ways to define a tree.

4.3.1 Terms and concepts

To ease the discussion, we first define some terms and concepts about trees and look at an example in Figure 4.1.

root The top most node is called the '*root*'. For example, '`usr`' is the root of the (free) tree.

leaf A node that has no children is called a *leaf*. For example, '`soda06`' is a leaf.

parent The predecessor node that every node has (except the root). For example, '`ida`' is the parent of '`research`' and '`research`' is the parent of '`sorting`'.

child A successor node that each node has (except leaves). For example, '`sorting`' is a child of '`research`'.

siblings Successor nodes that share a common parent. For example, '`sorting`' and '`paper`' are siblings.

subtrees A subtree is a substructure of a tree. Each node in a tree may be thought of as the root of a *subtree*. For example, '`paper`' is the root of a three-nodes subtree consisting of '`paper`', '`soda06`' and '`bctcs05`'.

degree of a tree node The number of children (or subtrees) of a node. For example, the node '`ida`' has a degree of 3.

degree of the tree The maximum of the degrees of the nodes of the tree, which is 3 in the example.

ancestors of a node All the nodes along the path from the root to that node. For example, the ancestors of node '`research`' are '`usr`' and '`ida`' (or '`usr/ida`').

path from node n_1 to n_k A sequence of nodes from n_1 to n_k. The *length* of the path is the number of edges on the path. For example, the length from '`research`' to '`cis208`' is 3.

depth of a node The length of the unique path from the root to the node. For example, the depth of node '`cis208`' is 3.

In addition,

forest A collection of trees is called a *forest*.

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

Example 4.2 *An arithmetic expression: $A * B + C$ can be represented by a binary tree.*

Recall we represented an arithmetic expression by a list to emphasise the hierarchy of operators, i.e. $*$, $/$ prior to $+$, $-$. An arithmetic expression tree more naturally shows the hierarchy (Figure 4.2).

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.

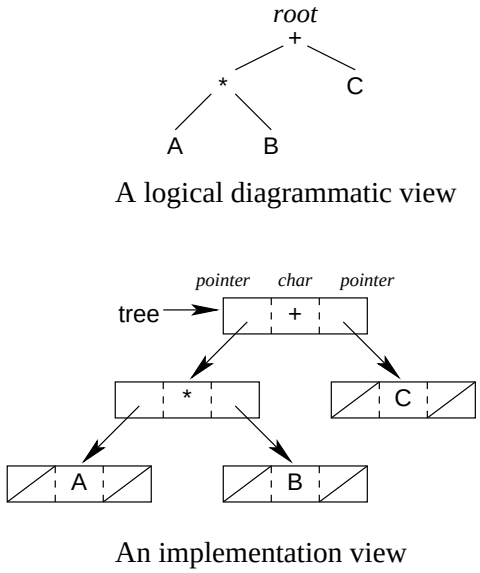


Figure 4.2: An expression tree

For example, consider the node '$*$' in Figure 4.2. Let $A = 2$, $B = 3$. The value of the subtree at '$*$' is $2 \times 3 = 6$.

4.3.2 Implementation of a binary tree

There are actually many kinds of trees. We are only concerned with *rooted, labelled* and *binary trees* here.

First, we need to define a tree node structure. A simple way to achieve this is to use a similar approach to the node definition for linked lists in section 2.7.2.

Example 4.3 *We need to define three fields, namely* `left, da, right`*, where* `left` *and* `right` *are the links to the left child and right child respectively, and* `da` *is the data field. In this way, similar to a linked list represented by its head, a tree can be referenced by its root.*

```
import java.io.*;

// A class of treeNode which allows us to construct
// a tree node

// This defines a 'simple tree node' (type).
public class treeNode {
  private Object da;
  private treeNode left;
  private treeNode right;

// This defines a new tree node.
  public treeNode(Object newItem) {
    da = newItem;
    left = null;
    right = null;
  } // Constructor

// This is to create a 'normal' node.
  public treeNode(Object newItem, treeNode leftNode,
    treeNode rightNode) {
      da = newItem;
      left = leftNode;
      right = rightNode;
  }

// This is to update the da field of a node.
  public void setItem(Object newItem) {
    da = newItem;
  }

// This is to read the item field of a node.
 public Object getItem() {
    return da;
  }

// This is to update the left field of a node.
 public void setLeft(treeNode leftNode) {
    left = leftNode;
  }

// This is to read the left field of a node
 public treeNode getLeft() {
    return left;
  }

// This is to update the right field of a node.
 public void setRight(treeNode rightNode) {
    right = rightNode;
```

```
  }

// This is to read the right field of a node
 public treeNode getRight() {
    return right;
  }

}
```

With the tree structure, it is easy to access a tree node.

Example 4.4 *In this example, we first display the data field value of the root, then move to the left child of the root and display the data field value, finally move to the right child of the left child of the root, and display the value.*

1: *print*($tree.da$)
2: $tree \leftarrow tree.left$
3: *print*($tree.da$)
4: $tree \leftarrow tree.right$
5: *print*($tree.da$)

Let us look at an arithmetic expression tree, for $(a + b)^2/(a - b)$, in Figure 4.3.

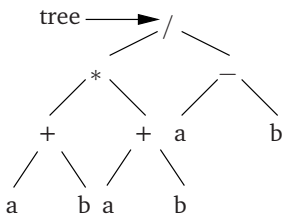


Figure 4.3: Another expression tree

The expression $(a + b)^2/(a - b)$ can be thought of as the operator '/' applies to two sub-expressions $(a + b)^2$ and $(a - b)$. This corresponds to the fact that the root of the expression tree has two subtrees, the left subtree and the right with their roots '$*$' and '$-$' respectively. If we move our view from the root '/' to '$*$', we see a similar picture where '$*$' applies to two subtrees $(a + b)$ and $(a + b)$ respectively. This actually applies to every non-leaf node of a tree. In fact, the most natural and easy way to define a tree is to define the tree *recursively*.

4.3.3 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.

Each (non-empty) subtree is called a *child* of r, and r is the *parent* of each subtree of r. Nodes with no children are called *leaves*. Nodes with the same parent are called *siblings*.

A *binary tree* (see Figure 4.4) 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.

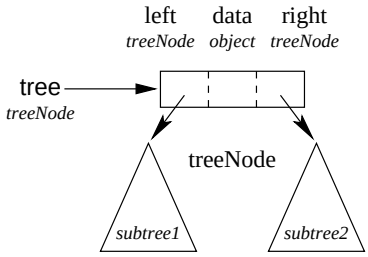


Figure 4.4: Recursive definition

Recall that many problems can be solved by including a subtask that calls itself as part of the solution. A program that calls itself at some stage is called a *recursive program*. An algorithm is called *recursive* if it contains a statement that calls itself.

Trees are recursive in nature and it is natural to write tree algorithms (programs) that use recursion. A recursive algorithm can be simpler and more elegant than a non-recursive equivalent, but one needs to be careful to make sure to implement recursion correctly. The reader is encouraged to undertake some revision of recursive algorithms (Chapter 3) before moving on to the next section.

4.3.4 Basic operations on binary trees

1. `initialise()` - create an empty binary tree.
2. `empty()` - *true* if the binary tree is empty, *false* otherwise.
3. `create(x)` - create a one-node binary tree `T`, where `T.da=x`.
4. `combine(l,r)` - create a tree `T` whose left subtree is `l` and right subtree is `r`.
5. `traverse()` - traverse every node of a tree `T` (see Section 4.3.5).

Algorithm 4.1 object initialise()

1: return null

Algorithm 4.2 boolean empty(treeNode t)

1: return ($t = null$)

Algorithm 4.3 treeNode combine(l,r)

1: $t \leftarrow null$
2: $setLeft(l)$, $setRight(r)$
3: return t

4.3.5 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 (Algorithms 4.4– 4.6):

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(tree 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 4.5 *A*B+C* *(See Figure 4.2)*

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

4.3.6 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()

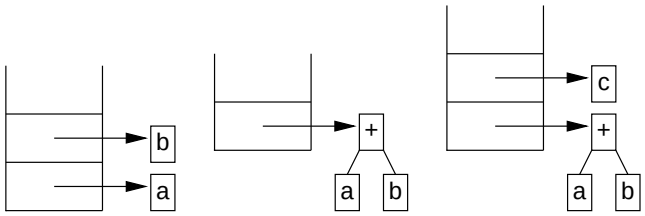
INPUT: An expression in array
RETURN: The root of the expression tree

1: Read the expression one symbol at a time.
2: **if** the symbol is an operand **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 respectively.
6: A pointer to this new tree is then pushed onto the stack.
7: **end if**
8: return the top of the stack

Example 4.6 *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 defined earlier for trees (Section 4.3.4), and stacks (Section 2.8.1), 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.*

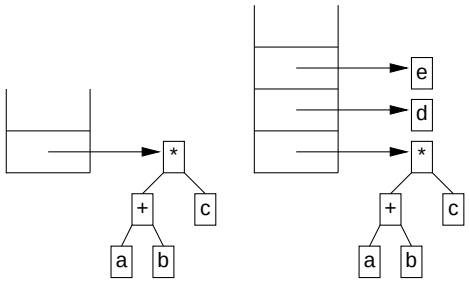
1. *Read a, create('a',T), push(S,T) read b, create('b',T), push(S,T) (Figure 4.5(a))*
2. *Read +, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.5(b))*
3. *Read c, create('c',T), push(S,T) (Figure 4.5(c))*



(a) step 1 (b) step 2 (c) step 3

Figure 4.5: steps 1–3

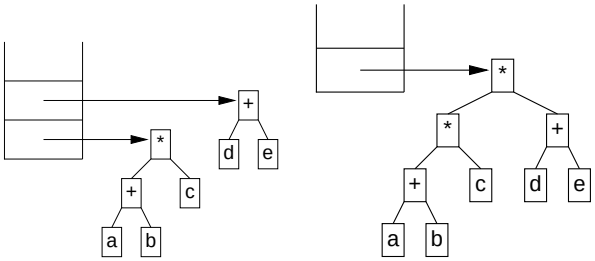
4. *Read *, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.6(a))*
5. *Read d, create('d',T), push(S,T)*
6. *Read e, create('e',T), push(S,T) (Figure 4.6(b))*
7. *Read +, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.7(a))*



(a) step 4 (b) step 5,6

Figure 4.6: steps 4–6

8. *Read *, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.7(b))*



(a) step 7 (b) step 8

Figure 4.7: steps 7–8

Applications

There are too many tree applications to list completely here. We will look at some of them in later chapters. The reader is encouraged to study more examples in the text books.

Activity 4.3

TREES

1. What is the difference between a tree and a binary tree?
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. For each expression tree drawn in the above question, list the sequence of elements encountered in inorder, preorder and postorder traversals.
5. Describe, with an example of 5 nodes, the topological characteristics that distinguish a tree from a linked list.
6. Define a treeNode class for tree nodes that each consists of following 4 fields:

parent	leftChild	rightChild	data

7. Following the above node definition, define and implement a binary tree class with necessary access methods for the 4-field treeNodes, for example, `setXXX`, `getXXX`, `isEmpty`, etc..
8. Write a method 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 subtree is `t1` and whose right subtree is `t2`. Both `t1` and `t2` should be empty on completion of the execution.

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[1] has an *order* property as well as a *structural* property. We first look at the structural property.

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

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 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 4.7 *The binary tree in Figure 4.8 is a complete binary tree, and a full binary tree in which all its leaves are at the same level.*

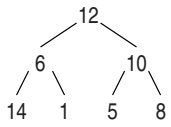


Figure 4.8: A full binary tree

A complete binary tree can be derived from a full binary tree by deleting some right most leaves (and edges leading to them). For example, if we delete the right most leaf '8' in the full binary tree in Figure 4.8. We get another complete binary tree.[2]

[2]Note the term 'complete binary tree' is sometimes also used as a synonym for full tree in some books. We distinguish the two concepts in this module to avoid confusion.

The structural property of the heap makes an array implementation easy. The nodes in the complete binary tree in Figure 4.8 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 div 2], its left child is A[$2i$] and the right child is A[$2i + 1$].

Example 4.8 *Figure 4.9 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.*

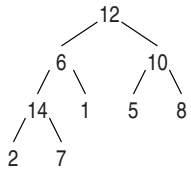


Figure 4.9: A complete binary tree

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

Example 4.9 *Figure 4.10 is a binary tree but not a complete binary tree because the left most node is missing from the last level. An incomplete binary tree leaves gaps in an array structure. The tree structure can, of course, still be stored in an array but dummy elements such as "%" are required.*

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

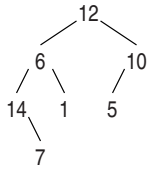


Figure 4.10: A binary tree but not complete

4.4.1.2 The order property

The order property requires, for every node in the structure, the value of both its child nodes must be smaller (or bigger) than the value at the current node.

Example 4.10 *Figure 4.11 shows a heap with the minimum value at the root. The value at each node is smaller than either child.*

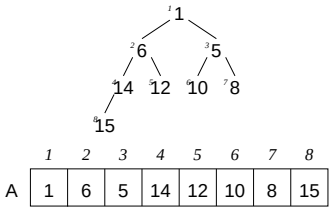


Figure 4.11: A heap with a MinKey root

or

A heap with the maximum value at the root, and the value at each node is smaller than either child (Figure 4.12).

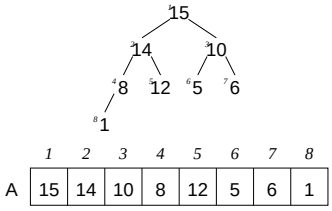


Figure 4.12: A heap with a MaxKey root

Note the complete binary tree in Figure 4.9 is not a heap because it does not have the order property required.

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 (Binary heap):

buildHeap() construct the initial heap from a list of items (keys) in arbitrary order (Figure 4.16).

deleteMin() remove the smallest element from the root and maintain the heap (Figure 4.13).

deleteMax() remove the largest element from the root and restore the heap (Figure 4.14).

insertOne(x) add one element x to the heap and maintain the heap (Figure 4.15).

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 4.11 *Figure 4.13 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.*

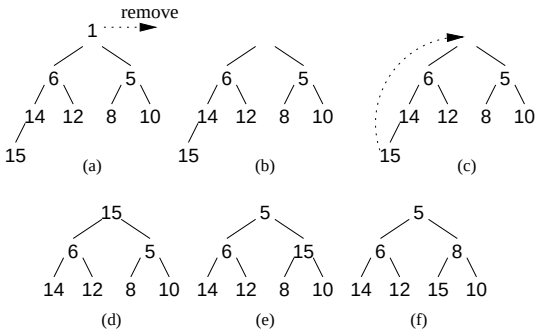


Figure 4.13: DeleteMin

A similar restoring process takes place for a max-heap.

Example 4.12 *Consider the max-heap in Figure 4.12. Figure 4.14 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. (Figure 4.14(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. (Figure 4.14(b)).

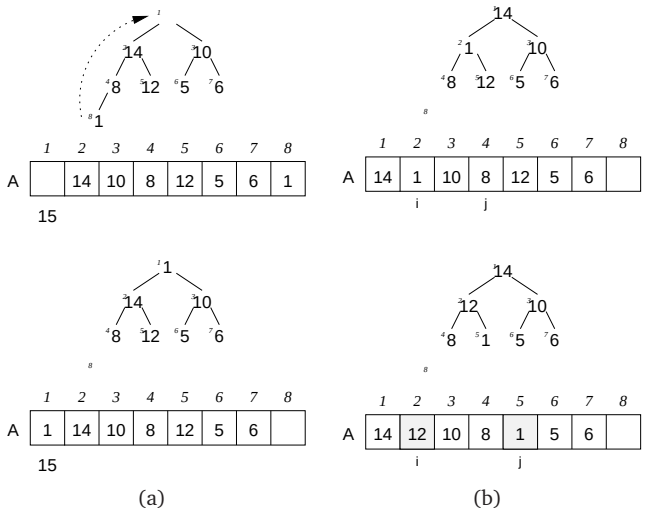


Figure 4.14: addRoot-1,2

This process can be described in Algorithm 4.8 and 4.9.

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
11:   end if
12: end if
```

Algorithm 4.9 buildHeap()

```
1: for k ← (length(A) div 2, k ≤ 1, 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 4.13 *Figure 4.15 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. (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.*

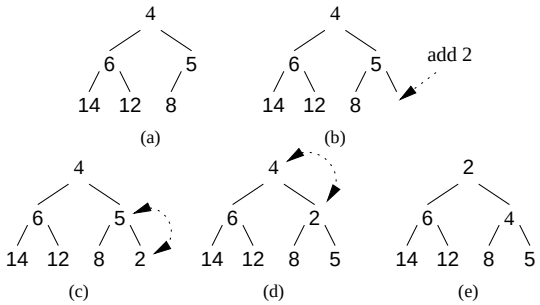


Figure 4.15: InsertOne

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. We look at this from Example 4.14.

Example 4.14 *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.*

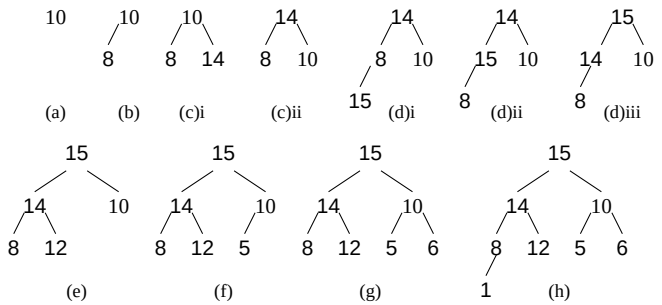


Figure 4.16: Construct a binary max-heap

Figure 4.16 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. (a) The first element 10 is the root, (b) 8 is added as its left child, (c)i 14 is

added and ii 10 and 14 are swapped to restore the order property, (d)i 15 is added, ii 15 is swapped with its father 8, iii 15 is swapped with its father 14 to restore the order property, (e)—(h) elements 12, 5, 6, and 1 are added respectively into the heap without a position swap.

This process is implemented on an array, where the new element x is marked by a box, e.g. $\boxed{\text{x}}$ and two swap elements x and y are underlined, e.g. $\underline{x} \cdots \underline{y}$.

$\boxed{10}$	8	14	15	12	5	6	1
10	$\boxed{8}$	14	15	12	5	6	1
10	8	$\boxed{14}$	15	12	5	6	1
$\underline{10}$	8	$\underline{14}$	15	12	5	6	1
14	8	10	$\boxed{15}$	12	5	6	1
14	$\underline{8}$	10	$\underline{15}$	12	5	6	1
$\underline{14}$	$\underline{15}$	10	8	12	5	6	1
15	14	10	8	$\boxed{12}$	5	6	1
15	14	10	8	12	$\boxed{5}$	6	1
15	14	10	8	12	5	$\boxed{6}$	1
15	14	10	8	12	5	6	$\boxed{1}$

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 (see Section 6.13).

We look at a simple example.

Example 4.15 *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 as in Figure 4.12. 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 (in shade in Figures 4.17–4.19). 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* $k \leftarrow k-1$.

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.*

The following ordered steps show the sorting process:

1. *Figure 4.17(a).*
2. *Figure 4.17(b)*
3. *Figure 4.18(a).*
4. *Figure 4.18(b).*
5. *Figure 4.19(a).*

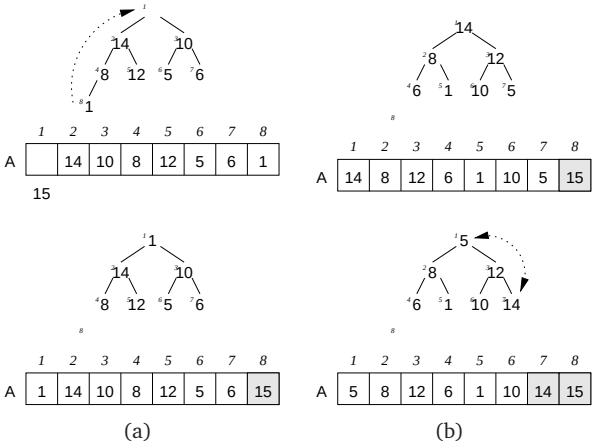


Figure 4.17: steps 1–2: addRoot-1,2

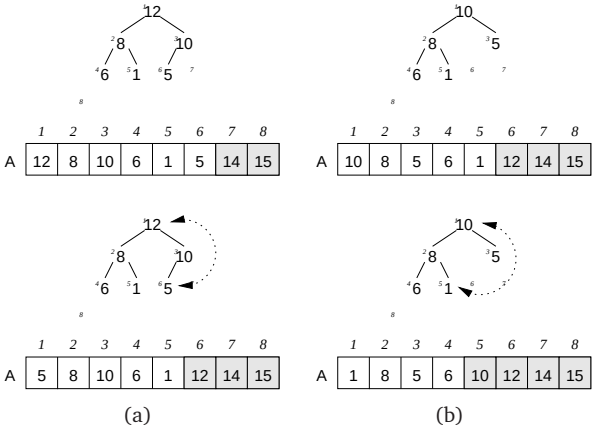


Figure 4.18: steps 3–4: addRoot-3,4

6. *Figure 4.19(b)*.
7. *Figure 4.19(c)*.

The above steps can be summarised in Algorithm 4.10.

Algorithm 4.10 heapSort(heapArray A)

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

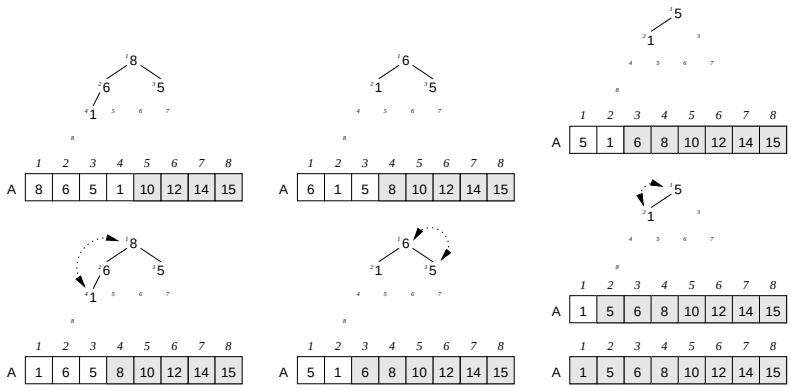


Figure 4.19: steps 5–7: addRoot-5,6,7

Activity 4.4

HEAPS

1. What is the difference between a binary tree and a tree in which each node has at most two children?
2. What is the approximate number of comparisons of keys required to find a target in a complete binary tree of size n?
3. What is a heap?
4. What is a priority queue?
5. Demonstrate, step by step, how to construct a max-heap for the list of integers $A[1..8] = (12, 8, 15, 5, 6, 14, 1, 10)$ (in the given order), following Example 4.14.
6. Demonstrate, step by step, how to sort a list of integers (2, 4, 1, 5, 6, 7) using a max-heap, following Example 4.15.

4.5 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:

Suppose that we want to connect computers in four buildings (Figure 4.20) by cable. The *question* is: Which pairs of buildings should be directly connected so that the total installation cost would be minimum?

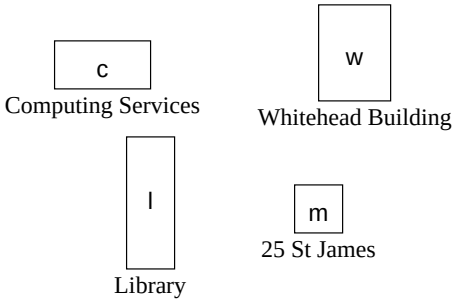


Figure 4.20: Four buildings

Naturally, the four buildings could be represented by four vertices with labels c, w, l, m (Figure 4.21). We then mark the cost between each pair of vertices by lines between vertices.

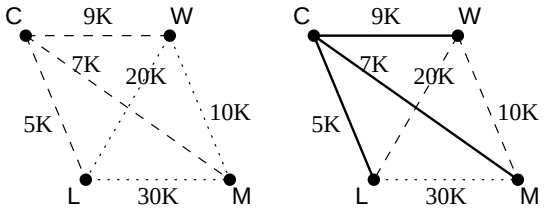


Figure 4.21: A graph

We look at all the possible connections (Figure 4.22):

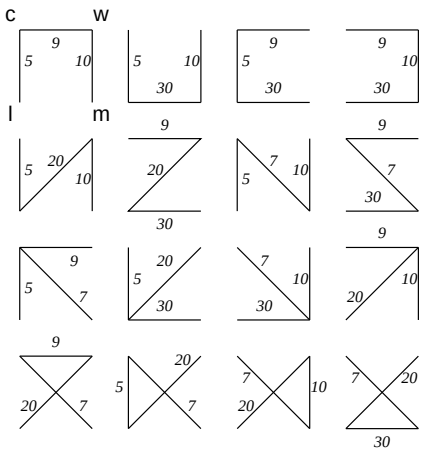


Figure 4.22: Possible connections

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.

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 (Figure 4.21), $V = \{c, l, w, m\}$ 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*. For graphs, the edge set consists of a *non-ordered* pair of vertices, e.g. (1,2)=(2,1), (2,3)=(3,2) etc. For directed graphs digraphs, the edge set consists of an *ordered* pair of vertices, e.g. (1,2)≠(2,1), (3,2)≠(2,3), etc. (Figure 4.23).

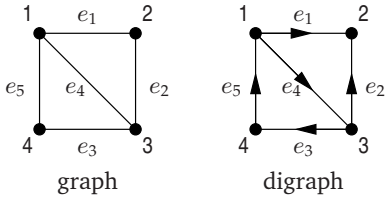


Figure 4.23: A graph and a digraph

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

Typical operations (Graphs):

order The number of vertices in a graph is called the *order* of the graph.

size The number of the edges in a graph is called the *size* of the graph.

path A path is a sequence of vertices $v_1, v_2, \ldots, v_k$, 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 length of a path is the number of edges on the path, which equals $k - 1$, where k is the number of vertices on the path.

simple path A simple path is 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 self-loop is a path from a vertex v to itself (v, v).

cycle A path $v_1, v_2, \ldots, v_k$ is a cycle if $v_1 = v_k$.

spanning tree A spanning tree of a connected graph is a subgraph and tree that contains all the vertices of the graph.

minimum spanning tree A minimum spanning tree of a weighted graph is a spanning tree of minimum total weight.

connected graph A graph is connected if there is a path between any two vertices of the graph.

complete graph A graph is complete if there is a path between *every* two vertices of the graph.

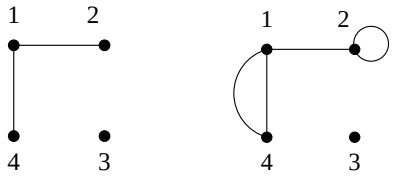
labelled graph A graph is labelled if every vertex has a fixed identity.

unlabelled graph A graph is unlabelled if there is no fixed identity for each vertex.

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

simple graph A simple graph is a graph that contains no self-loop nor parallel edges.

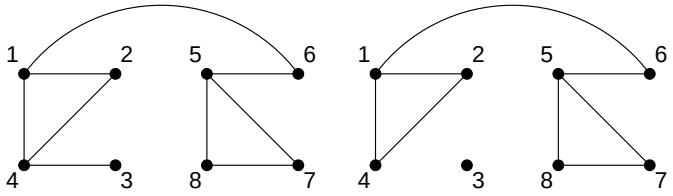
Example 4.16 *Figure 4.24(a) shows a simple graph while Figure 4.24(b) shows a non-simple graph with a parallel edge and self-loop. They are both disconnected.*



(a) simple (b) non-simple

Figure 4.24: A simple and non-simple graph

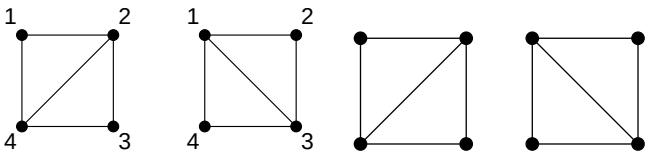
Example 4.17 *Figure 4.25(a) shows a connected graph. Figure 4.25(b) shows a disconnected graph with an isolated vertex. They are both unweighted.*



(a) A connected graph (b) A disconnected graph

Figure 4.25: A connected and disconnected graph

Example 4.18 *Figure 4.26(a) shows two labelled graphs and they are different, but the two unlabelled graphs in Figure 4.26(b) are the same. The graphs are connected and unweighted in both figures.*



(a) Two different labelled graphs (b) Two identical unlabelled graphs

Figure 4.26: Labelled and unlabelled graphs

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

Representation of graphs

We discuss three commonly used data structures to represent graphs. They are *adjacency matrices, incidence matrices* and *adjacency lists*. As we shall see later, the use of different data

structures can sometimes improve (or worsen) the efficiency of algorithms.

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.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

   where

$$a_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

a graph a digraph

Figure 4.27: A graph and digraph

Example 4.19 *The Adjacency matrix for the graph in Figure 4.27 is,*

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

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

and for the digraph in Figure 4.27 is,

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

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

Note The Adjacency matrix for an undirected graph is always symmetric if row and column nodes are listed in the same order.

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 $n \times m$ matrix:

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix}$$

where for a *graph*

$$b_{i,j} = \begin{cases} 1 & \text{if vertex } i \text{ and edge } j \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

For a *digraph*,

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

Example 4.20 *The incidence matrix for the graph in Figure 4.27 is*

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

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

and for the digraph in Figure 4.27 is,

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

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

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

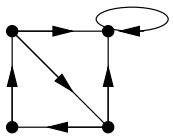


Figure 4.28: A digraph with a self-loop

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.

   Why an adjacency list?

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

   Example 4.21 *Suppose that we need to store the digraph with few edges in Figure 4.27 . Suggest a data structure for the digraph.*

   Solution *An adjacency list can save space for a sparse graph (Figure 4.29).*

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

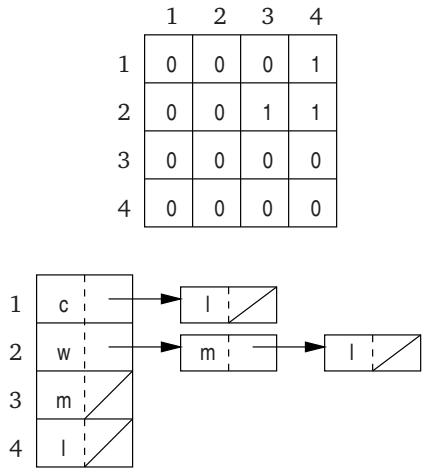


Figure 4.29: An adjacency list

numerals. For our example (see the digraph in Figure 4.27), 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:

Example 4.22 *See Figure 4.30.*

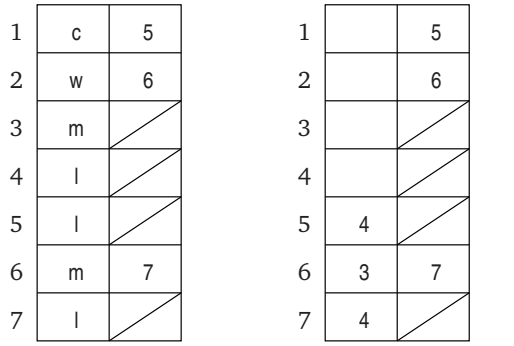


Figure 4.30: Implementation

Graph algorithms

For many important problems about graphs, no efficient algorithms have been found. A lot of effort has been expended on seeking efficient algorithms for graph problems. Interesting algorithms have been found for some graph problems but a comprehensive account of these problems would fill volumes. We will look at a few of these problems here.

One important class of graph problems is about graph traversal. As with binary trees, we would like to investigate all the vertices in a graph in some systematic order. We also want to avoid cycles during the traversal. With many possible orders for visiting the vertices of a graph, two traversals are particularly important, namely *Depth-first traversal* and *Breadth-first traversal*. They are also called *Depth-first*

search and *Breadth-first search*.

See Section 5.3.3 for the algorithms.

Activity 4.5

GRAPHS

1. Consider the adjacency matrix of a graph below:

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

   (a) Draw the graph

   (b) Write the adjacency list for the graph

   (c) Discuss the suitability of using an adjacency matrix and the adjacency list for the graph. Justify your answer.

2. Using the adjacency matrix approach, write a program to store a simple graph and display the graph.

   Example 4.23

```
          Store and Display
           a simple graph
      ------------------------
        1. Store a graph
        2. Display a graph
        0. Quit
   Please input your choice (0-2) >
```

   Hint An easy approach may be:

   (a) define a data structure for the graph

   (b) decide a means to input the graph, for example, you may

      i. type the entries of the adjacency matrix on the keyboard

      ii. read the entries of the adjacency matrix from a text file

      iii. generate a random adjacency matrix by a program.[3]

   (c) write the main program or method with interfaces of the sub-methods or procedures

   (d) develop each part of the program.

3. Implement the graphs in question 1 using the adjacency list approach.

[3] Use a random generator to generate a 0 or 1 uniformly at random for each entry of the matrix.

Chapter 5

Traversal and searching

5.1 Essential reading

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 6.1, 6.2

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 7.3, 7.4, 7.6

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 12

5.2 Learning outcomes

In this chapter, we shall study the problems of *Traversal*, and *Searching*. Having read this chapter and consulted the relevant material you should be able to:

- describe the traversal and searching algorithms as classes
- apply big-O notation in classifying the time efficiency of algorithms involving iterative or recursive constructs.
- explain the difference between an algorithm and its implementation
- select appropriate searching algorithms for problems
- explain the relative efficiency of a binary search compared with a sequential search

5.3 Traversal

A *traversal* problem can be described as the situation where every datum in a given data structure needs to be visited at least once. Here the term *visit* can be interpreted in many ways with various meanings. For example, printing out each value can be a traversal, and updating such as adding, subtracting, multiplication, division, assigning 0s can be another traversal.

Traversal is a fundamental step in many other more complicated operations, such as checking how many data exceed a certain threshold. In particular, traversal is necessary for any searching algorithm in the worst case. For this reason, some traversal algorithms are described as searching algorithms. Examples include the popular *depth-first search* and *breadth-first search*

5.3.1 Traversal on a linear data structure

The problem of traversal of a linear data structure such as an array, queue, stack or set is easy. A single loop can be used to have each datum visited in the structure.

Example 5.1 *Given an array* $S[1..10]$*, initialise the array to 0s.*

```
1: for i ← 1; i ≤ 10; i + + do
2:     S[i] ← 0
3: end for
```

5.3.2 Binary tree traversal

There are three standard ways to traverse a binary tree, namely *preorder*, *inorder* and *postorder* traversal.

Review Section 4.3.5 for details.

5.3.3 Graph traversal

The graph traversal problem can be broadly divided into two types: one is to visit every node of a graph and the other is to traverse every edge of a graph. We consider only *simple* graphs, i.e. graphs which contain no self-loops or parallel edges, in this subject guide.

We look at two traversal algorithms on a graph, namely *depth-first* and *breadth-first* traversal. They are also called *depth-first search* and *breadth-first search* historically. The word *traversal* is sometimes interchangeable with *search* in this context since traversal problems are so closely related to searching problems. Strictly speaking, the word search is different from traverse. A search will stop as soon as the searched item is found, but traversal will visit every item in the data structure.

5.3.4 Depth-first traversal

The main idea of *depth-first traversal* is to go as far as possible along a path without revisiting any node, then backtrack to the last turning point and go as far as possible down the next path, and so on, until all the nodes are visited.

Depth-first search is analogous to preorder traversal of an ordered tree (Algorithm 5.1).

Recursive version (Algorithm 5.2):

5.3.5 Breadth-first traversal

The *breadth-first traversal* is to visit all vertices within one same radius (the same path length from the start point) before visiting all

Algorithm 5.1 DFS(headerList adjacencyList, vertexType v)

```
1: initialise(S)
2: visit, mark, and push(S, v)
3: while not empty(S) do
4:    while there is an unmarked vertex w adjacent to top(S) do
5:       visit, mark, and push(S, w)
6:       pop(S, x)
7:    end while
8: end while
```

Algorithm 5.2 DFS(vertexType v)

```
1: visit and mark v
2: while there is an unmarked vertex vertexType w adjacent to v
   do
3:    DFS(w)
4: end while
```

vertices within the next radius further.

Breadth-first search is analogous to level-by-level traversal of an ordered tree.

Algorithm 5.3 gives details of the Breadth-first traversal, where Q represents a queue and w, x are variables of vertexType.

Algorithm 5.3 BFS(headerList adjacencyList, vertexType v)

```
1: initialise(Q)
2: visit and mark v, enqueue(Q, v)
3: while not empty(Q) do
4:    dequeue(Q, x)
5:    for each unmarked vertex w adjacent to x do
6:       visit and mark w
7:       enqueue(Q, w)
8:    end for
9: end while
```

Example 5.2 *Consider a connected graph in Figure 5.1. Starting from vertex* **A***, write the vertex sequence in the order that each vertex is visited applying the*

1. *depth first* traversal algorithm
2. *breadth first* traversal algorithm.

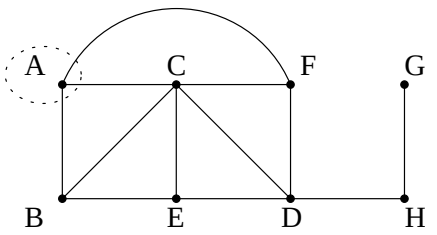


Figure 5.1: Traversals on a graph

Solution

1. ABEDHGFC (or ABEDHGCF)
2. ABCFEDHG (or AFCBEDHG)

Note there may be other correct traversals and the result depends on the implementation.

5.4 Searching

Searching is a class of problems for which the goal is to discover whether or not a particular element is contained in a collection. One of the most important functions of any computer application is *information retrieval*, from a small database allows simple queries to a web-based search engine such as Google. The main step in information retrieval is *searching*.

The efficiency of searching depends on the relationships within a large set of data records, each of which consists of a number of *fields*. The field is called a `key` of the record if it can be used to identify each record uniquely. It is often time-consuming to search for a given key value to locate a record from a large data collection, especially if the searching needs to be performed frequently. Thus the research on how to arrange the record storage and what is the best method for searching is intensive.

There are many ways of defining search problems. We define a simplified version of the problem as follows.

Problem Given an integer key value X and a list of integers $L = (a_1, a_2, \ldots, a_n)$, find if X is in L. If yes, supply the location.

Searching divides naturally into two categories depending on *where* the records to be searched are stored, namely:

- Internal searching: the records are stored entirely within the computer memory, directly accessible by the processor in fixed time.
- External searching: the large amount of records have to be stored in files on disks or tapes.

They focus on different issues although the main concerns are the same. In this course unit, however, we are only concerned with internal searching problems due to the time constraints.

We consider a type of searching problem that is based on comparison of two data of the same type. This helps us to concentrate on the fundamental characteristics of searching.

In order to focus on the main issues of the searching problem, we further assume that:

1. All the elements in the given list L are unique
2. X and any element in L are comparable.

Our *goal* is to minimise the number of comparisons such as $<$, $=$ or $>$ during searching.

5.5 Sequential search

The simplest way to search for a key in a list is to scan the whole list, from the start to the end, until the key is found or the finish end is reached. This is called *sequential search*.

Example 5.3 $L = (12, 34, 2, 9, 7, 5)$, $X = 7$.

We compare the key "7" with each element in the list from left to right.

```
12 34 2 9 7 5
7

12 34 2 9 7 5
   7

12 34 2 9 7 5
      7

12 34 2 9 7 5
        7

12 34 2 9 7 5
          7        found!
```

Algorithm outline

Given *max*, a predefined constant, and an array [1..max] of data of some *targetType*, a sequential search algorithm is as follows:

1. For data in an array of fixed length: Algorithm 5.4.

Algorithm 5.4 seqSearch(list L, targetType X, boolean found, int location

```
1: found ← false
2: location ← 1
3: while not found and location ≤ max do
4:    if L[location] = X then
5:       found ← true
6:    else
7:       location ← location + 1
8:    end if
9: end while
```

2. For data in a linked list: Algorithm 5.5.

Algorithm 5.5 seqSearch(node L, targetType X, boolean found, node location

```
1: found ← false
2: while (not found) and (L ≠ null) do
3:    if L.da = X then
4:       found ← true
5:       location ← L
6:    else
7:       L ← L.next
8:    end if
9: end while
```

Complexity analysis

The analysis is straightforward.

1. Worst case: n comparisons (where n is the number of elements in the list to be searched). This is when X appears only in the last position in the list or when X is not in the list at all.
2. Average case: $(n+1)/2$ comparisons. Since X could be in any position of $1..n$ with equal probability, the number of comparisons done would be $1..n$ respectively. So on average, the number of comparisons is $\frac{1+2+\cdots+n}{n} = \frac{n+1}{2}$.

So the sequential search algorithm takes $O(n)$ time in both worst case and average case.

5.6 Binary search

If the list is pre-sorted, then a more efficient way of searching strategy called *binary search* can be used. The idea of binary search is to first compare the key with one at the centre of the list[1] and then move our attention to only one of the first or the second half of the list. In this way, at each step we reduce the length of the list to be searched by approximately *half*.

[1] or as close to the centre as possible if there are an even number of entries.

Example 5.4 $L = (3, 7, 11, 12, 15, 19, 24, 33, 41, 55)$, $X = 20$.

```
i     1  2  3  4  5  6  7  8  9  10          centre element
L[i]  3  7  11 12 15 19 24 33 41 55               15
                  20

L[i]                  19 24 33 41 55              33
                           20

L[i]                  19 24                       19
                      20

L[i]                     24                       24
                         20
                                  20 is not found!
```

Algorithm outline

Suppose the list of elements is stored in array `L`. We define variables `low, high` as the index of the first and of the last element,

respectively, of the list at each stage, and `mid` as the index of the centre element of the list. The main idea of binary search is outlined in Algorithm 5.6, which does not work yet (why?):

Algorithm 5.6 bSearchDraft(index low, high, location) (to complete)

```
1: if low < high then
2:    find centre position mid
3:    compare X with L[mid]
4:    if X < L[mid] then
5:       bSearch(low, mid − 1)
6:    else if X > L[mid] then
7:       bSearch(mid + 1, high)
8:    else
9:       location ← mid
10:   end if
11: end if
```

A good way to work on the details of an algorithm is to play with examples. So we look at an example first.

Example 5.5 *Suppose the key value to be searched is 20, again. Let us demonstrate how the algorithm works by tracing the index values for the locations* `low, high, mid` *and the flag* `found` *in the binary search algorithm. Let* `low` *be 1 and* `high` *be some integer* `n` *initially. Suppose* $mid = (low + high)$ *DIV* 2.

```
i     1  2  3  4  5  6  7  8  9  10   low  mid  high   found
L[i]  3  7  11 12 15 19 24 33 41 55
      l                       h     1         10    False
      l           m           h     1    5    10    False
                  m  l        h     6    5    10    False
                     l     m  h     6    8    10    False
                     l  h  m        6    8    7     False
                     m  h           6    6    7     False
                     m  l           7    6    7     False
                        m           7    7    7     False
                     h  m           7    7    6     False
```

The algorithm ends when $low > high$*. Since the flag* `found` *is still* `False`*, the algorithm will report the searching result "not found".*

We can now modify the draft Algorithm 5.6, and derive the working version in Algorithm 5.7.

The *BinarySearch1()* in Algorithm 5.7 enables a simple implementation but it can make unnecessary iterations, because it fails to recognise in time that X is found. We solve this problem by checking the variable 'found' in each iteration (see *BinarySearch 2* in Algorithm 5.8).

Complexity analysis

Using the decision tree technique in Chapter 6, we can easily see that the worst case and average case complexity for BinarySearch (i.e. BinarySearch 1 and 2) are both $O(\log n)$ comparisons.

Algorithm 5.7 BinarySearch1(list L, boolean $found$, index $location$)

```
1: high ← n, low ← 1
2: while high > low do
3:    mid ← (low + high) div 2
4:    if X > L[mid] then
5:       low ← mid + 1
6:    else
7:       high ← mid
8:    end if
9:    if high = 0 then
10:      found ← false
11:   else
12:      found ← (X = L[high])
13:   end if
14: end while
15: if found then
16:   location ← high
17: end if
```

Algorithm 5.8 BinarySearch2(list L, boolean $found$, index $location$)

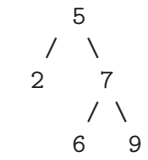
```
1: high ← n, low ← 1, found ← false
2: while (not found) and (high ≥ low) do
3:    mid ← (low + high) div 2
4:    if X < L[mid] then
5:       high ← mid − 1
6:    else if X > L[mid] then
7:       low ← mid + 1
8:    else
9:       found ← true
10:   end if
11: end while
12: if found then
13:   location ← high
14: end if
```

5.7 Binary search trees

Binary search trees (BST) are binary trees with an order property. They are particularly useful for searching. Each node in a binary search tree contains at least one key field of some rankable value. For every node Y in the tree, the values of all the keys in the left subtree are smaller than the key value of Y, and the values of all the keys in the right subtree are larger than the key value of Y.

To keep the structure simple, we assume that there are no identical values in the data collection.

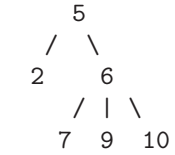
A BST looks like this:



Neither of these is a BST:

(Why? ... because they do not have the 'order property'.)

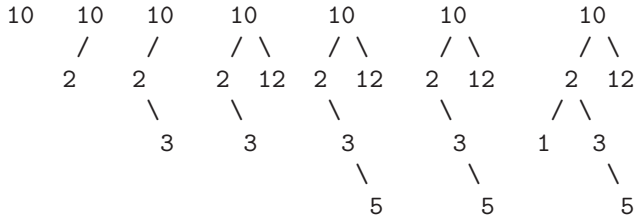
That is not a BST:



(Why? ... because it is not a binary tree.)

Next, let us look at how to construct a binary tree for a given set of data.

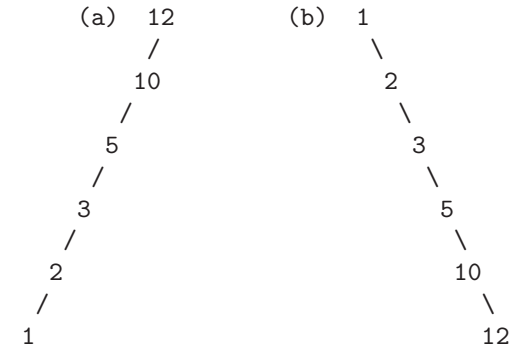
Example 5.6 *Given a list of integers: 10, 2, 3, 12, 5, 1, we can construct a BST by inserting the integers one by one in the given order:*



The shape of each binary tree constructed in this way depends on the insertion order of the data. Example 5.7 shows two BSTs constructed for the same set of integers (10, 2, 3, 12, 5, 1) in (a) descending and (b) ascending order.

Example 5.7 *Construct a BST for (a) 12, 10, 5, 3, 2, 1; (b) 1, 2, 3, 5, 10, 12.*

Solution



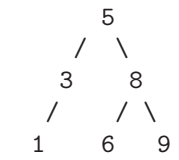
As you can see, the two BSTs constructed grow only to one direction, the first one (a) to the left and the second one (b) to the right. Such a tree is called a *Splay* tree.

In these cases, the advantages of a normal binary tree structure do not exist any more. In terms of the operations such as searching, the trees in these shapes are no different from a linked linear structure.

There are no advantages any more in terms of efficiency and no $O(\log n)$ can be achieved. So a binary search tree can be 'bad' as well as 'good'. When it is constructed according to a sorted list, in either ascending or descending order, the BST is completely imbalanced and is in its worst case.

There is a nice feature of a binary search tree. Let us look at another example, where we conduct a traversal on a relatively balanced binary search tree.

Example 5.8 *Consider a binary search tree below.*



We write down the nodes visited in each of the following tree traversals:

1. *pre-order traversal: 5 3 1 8 6 9*
2. *post-order traversal: 1 3 6 9 8 5*
3. *in-order traversal: 1 3 5 6 8 9.*

You have probably realised that the inorder traversal on a binary search tree gives a *sorted* list.

This means that given a list of integers, we can first store the integers in a binary search tree which takes $O(n \log n)$ time, and then conduct an in-order traversal to print out a list of integers, and this list of integers becomes sorted.

Implementation

We now consider how to insert a node into a binary search tree.

Suppose that there are three fields of each treeNode, namely `left`, `data` and `right`.

An easy way to construct a binary search tree for a list of data, e.g. integers, is to store them one by one in the order of entry and let each element be the left or right child of a leaf according to the value of the key. In this way the shape of the binary tree will depend on the order of entry of the integers as we discussed earlier.

Example 5.9 $L = (12, 34, 2, 9, 7, 5)$ *can be stored in a binary search tree as in Figure 5.2 (a).*

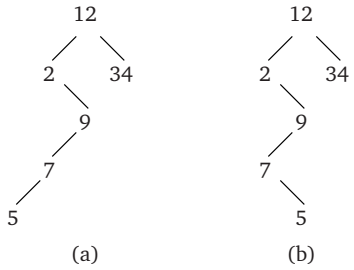


Figure 5.2: (a) A binary search tree (b) A binary tree

We can define a new type `treeNode` for a binary search tree node. A binary search tree can be identified simply by a variable, e.g. T of the `treeNode` type.

To construct a empty BST, we assign the root of the BST to a `null` (Algorithm 5.9).

Algorithm 5.9 makeEmpty(searchTree T)

1: $T \rightarrow null$

Algorithm 5.10 inserts a new node in a binary search tree T.

Algorithm 5.10 insert(data X, searchTree T)

1: **if** $T = null$ **then**
2: $T.data \leftarrow X$
3: $T.left \leftarrow null$
4: $T.right \leftarrow null$
5: **else if** $X < T.data$ **then**
6: insert$(X, T.left)$
7: **else if** $X > T.data$ **then**
8: insert$(X, T.right)$
9: **end if** {Do nothing if $X$is already in T $(X = T.data)$}

Algorithm 5.11 searches for X in T.

Algorithm 5.11 treeNode findNode(data X, searchTree T)

```
1: if T = null then
2:    return null
3: else if X = T.data then
4:    return T
5: else if X < T.data then
6:    return findNode(X, T.left)
7: else
8:    X > T.data
9:    return findNode(X, T.right)
10: end if
```

Algorithm analysis

If a binary search tree is balanced, the function `findNode` takes $O(\log n)$ time in the worst case. This is because, each time, we descend a level in the tree, thus operating on a tree that is now about half as large. Recall that a balanced binary tree is the tree in which the depths of the leaves are all the same.

However, a binary search tree can be so "bad" that it does not have any branching. For example, if the integers are inserted in ascending order using the procedure `insert` (Algorithm 5.10) started with an empty tree. In this case, the height of the binary search tree is $O(n)$. Hence the function `findNode` needs $O(n)$ time in the worst case.

Activity 5.7

SEARCHING AND TRAVERSAL

1. Design and implement a searching algorithm which searches for a *name* from a list of student records (linked structure). Let the data part of each record contains the following fields: `name`, `age` and `address`.
2. Implement the *depth-first* and the *breadth-first* searching algorithms in Algorithms 5.1 and Algorithms 5.3 respectively.

   Hint You need to construct a graph before any traversal. You may use the program that you have developed for Question 2(b)ii of Activity 4.5. If you have not done so, attempt one of questions below first:

   (a) Using the array approach, write a subprogram which constructs an adjacency matrix of a given graph. The entries of the matrix should be input from an external text file `vertices.txt`.

   (b) Define an array of linked lists. Implementing the adjacency list of a given graph, write a subprogram which constructs an adjacency list of a given graph. The vertex data should be input from an external text file `vertices.txt`.

Chapter 6

Sorting

6.1 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 11.1–11.4

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 6.3

Jeffrey Kingston *Algorithms & Data Structures: Design, Correctness, Analysis*. (Addison-Wesley, 1997, or 1998, second edition) [ISBN 0-201-40374-9]. Chapter 9, 13

Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3]. Chapter 8

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 4

6.2 Learning outcomes

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

- describe sorting algorithms as a class
- explain the difference between a sorting algorithm and an implementation of the algorithm
- analyse the time efficiencies of the Insertion sort, Selection sort, Mergesort and Quicksort algorithms.

6.3 Introduction

Sorting means rearranging a set of *elements* into *order*. Each element in the set has an identifier called the *key*. The elements are each *allocated* a logical position, according to the value of their keys, in *ascending*, (or descending) order.

Example 6.1 *Suppose a student record consists of several fields of name, age, address, telephone number and average mark of all the subjects. We want to sort an array of student records into ascending order according to the average marks of their courses.*

6.4 Motivation

There are several good reasons for studying sorting algorithms. First, sorting is done often in practice. A sorted set of records of data

has many advantages over an unsorted one, especially for searching (see Section 5.6). Secondly, sorting problems are good examples to show that one can take many different approaches to the same problem. Thirdly, sorting is one of the few classes of problems for which we can easily derive good lower bounds (Section 6.10) for the worst and average cases. Finally, sorting problems have been well studied and the literature on sorting is extremely rich.

To ease our discussion later, a few assumptions are made as follows:

- Sorting problems are of various forms. To simplify the discussion and without losing generality, we limit our discussions to integer sorting, i.e. to sorting a list of *non-repeating* integers (the keys), according to their values, in *ascending* (also called *increasing*) order.
- Practically, there are two classes of sorting, namely *internal* sorting and *external* sorting. Internal sorting is to perform the entire sort in the main memory, where the number of elements to be sorted is relatively small (depends on the size of each record and the internal memory available). External sorting is a type of sorting that is not performed entirely in the main memory and is done using disks or tapes, usually because the data size exceeds that of internal memory. We assume internal sorting if not stated otherwise.

6.5 Insertion Sort

The idea behind Insertion Sort is natural and general. The analyses for the worst case and average case are straightforward. We first look at an example and then derive the algorithm. We then analyse the algorithm for the worst case and average case. The implementations of the algorithms for different data structures are also discussed.

Example 6.2 *Sort the integers (34, 8, 64, 51, 32, 21) into ascending order.*

```
Sorted part        Unsorted part       Current   Integer(s)
                                        integer   moved
----------------   -----------------   -------   -----------
                   34 8  64 51 32 21
34                    8  64 51 32 21   34        -
8  34                    64 51 32 21   8         34
8  34 64                    51 32 21   64        -
8  34 51 64                    32 21   51        64
8  32 34 51 64                  21     32        34 51 64
8  21 32 34 51 64                      21        32 34 51 64
```

The idea is to first look at the elements one by one and build up the sorted list by inserting each element to the correct location.

Implementation

The most common implementation is to use an array. Algorithm 6.2 gives an algorithm to sort an array of *key* type (which can be an integer, char, real, or any ranked set of values.).

Algorithm 6.1 minimumKey()

```
1: for p ← 2, p < n, p ← p + 1 do
2:     x ← A[p]
3:     find location j
       {for j = p − 1, p − 2, · · · , 1, such that x < A[j]}
4:     move A[j]..A[p − 1] one location to the right (cell)
5:     place x in A[j]
6: end for
```

Algorithm 6.2 insertSort(inputArray A, integer n)

```
1: for xIndex ← 2, xIndex < n, xIndex ← xIndex + 1 do
2:     j ← xIndex
3:     x ← A[xIndex]
4:     while x < A[j − 1] do
5:         A[j] ← A[j − 1]
6:         j ← j − 1
7:     end while
8:     A[j] ← x
9: end for
```

6.5.1 Algorithm analysis

Worst case: For each $j = 2..n$, the maximum number of key comparisons in the *while* loop is $j - 1 = 1..n - 1$. The total number of comparisons is, therefore,

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2)$$

Hence $W(n) = O(n^2)$.

Best case: The input is already sorted in ascending order, the running time is $O(n)$.

Summary of insertion sort

1. Insertion sort is a comparison-based sorting method.
2. Insertion sort consists of $n - 1$ passes.
3. Elements in positions $xIndex - 1$ downto 1 at lowest are checked in each pass.

6.6 Selection sort

Selection sort is another simple sorting strategy. Repeatedly find the remaining minimum key from the input list, delete it and append it to the result list.

One major disadvantage of insertion sort is the possible *redundant movement* of elements. Even after most items have been already sorted into the first part of the list, the insertion of a later item may require to move many of them again. The objective of *Selection sort* is to minimise data movement.

Example 6.3 *Sort the integers (34, 8, 64, 51, 32, 21) into ascending order.*

```
Sorted part         Unsorted part       MinKey  Integers
                                        so far  to swap
-----------------   -----------------   ------  --------
                    34 8  64 51 32 21   8       8  34
8                      34 64 51 32 21   21      21 34
8  21                     64 51 32 34   32      32 64
8  21 32                     34 64 51   34      -
8  21 32 34                     64 51   51      51 64
8  21 32 34 51                     64   64      -
8  21 32 34 51 64
```

The idea is to, for each element $A[i]$, find the MinKey (the minimum key) and its location m, and swap MinKey and $A[i]$.

Algorithm 6.3 find the MinKey

1: $MinKey \leftarrow A[1]$
2: **for** each element $A[j]$ where $j \geq 2$ **do**
3: **if** $A[j] < MinKey$ **then**
4: $MinKey \leftarrow A[j]$
5: $m \leftarrow j$
6: **end if**
7: **end for**

Algorithm 6.4 swap MinKey and A[i]

1: $A[m] \leftarrow A[i]$
2: $A[i] \leftarrow MinKey$
3: $MinKey \leftarrow A[m]$

Implementation

Again, we use an array to implement the algorithm.

Algorithm 6.5 selectSort(list L, integer n)

1: index i, m
2: **for** $i \leftarrow n, i \leq 2, i \leftarrow i - 1$ **do**
3: $m \leftarrow minKeyIndex(i - 1, n, L)$
4: Swap(m, i, L)
5: **end for**

Algorithm 6.6 index function minKeyIndex(index $low, high$, list A)

1: index m, j
2: $m \leftarrow low$
3: **for** $j \leftarrow low + 1$, $j \leq high$, $j \leftarrow j + 1$ **do**
4: **if** $A[m] > A[j]$ **then**
5: $m \leftarrow j$
6: **end if**
7: **end for**
8: return m

Algorithm 6.7 procedure Swap(index i, j, list A)

```
1: itemType tmp
2: tmp ← A[i]
3: A[i] ← A[j]
4: A[j] ← tmp
```

Summary of selection sort

1. Selection sort is a comparison-based sorting method.
2. Selection sort consists of $n - 1$ passes.
3. At each pass, swap the smallest element from the unsorted part into its correct location.

6.7 Shellsort

Before discussing Shellsort, let us compare the two kinds of sorting methods so far: Insertion sort and Selection sort.

- Selection sort moves the items efficiently but may do many redundant comparisons.
- Insertion sort, in its best case, does the minimum number of comparisons, but is usually inefficient in moving items.

The reason that Insertion sort can move items only one position at a time is that it compares only adjacent elements. Shellsort compares and sorts elements far apart. Instead of comparing only adjacent keys, Shellsort uses the *increment sequence*, $h_1, h_2, \cdots, h_t$, where $h_t = 1$ and check the items h_k apart at pass k, where $k = 1 \ldots t$.

There are many choices for the increment sequence $h_1, h_2, \cdots, h_t$ as integers. We do not discuss the choices here because questions about best choices are still unsolved problems for general case[1]. We make a simple choice of sequence 5, 3, 1 for the example below.

[1] Sedgewick, R. Analysis of shellsort and related algorithms. In Algorithms -ESA '96, Fourth Annual European Symposium, Barcelona, Spain. Lecture Notes in Computer Science 1136, Springer-Verlag 1-11, 1996

Example 6.4 *Sort the integers (81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15) into increasing order.*

h_k	*Original list*	*Result in each pass*
	81 94 11 96 12 35 17 95 28 58 41 75 15	
$h_1 = 5$	*81 ··· 35 ··· 41*	*35 ··· 41 ··· 81*
	94 .... 17 75	*17 .... 75 .... 94*
	11 .... 95 15	*11 .... 15 .... 95*
	96 28	*28 .... 96*
	12 58	*12 .... 58*
	35 17 11 28 12 41 75 15 96 58 81 94 95	
$h_2 = 3$	*35 .. 28 .. 75 .. 58 .. 95*	*28 .. 35 .. 58 .. 75 .. 95*
	17 .. 12 .. 15 .. 81	*12 .. 15 .. 17 .. 81*
	11 .. 41 .. 96 .. 94	*11 .. 41 .. 94 .. 96*
	28 12 11 35 15 41 58 17 94 75 81 96 95	
$h_3 = 1$	*11 12 15 17 28 35 41 58 75 81 94 95 96*	

Note We have used the increment sequence 5, 3, 1. Many other choices might work as well or even better. However, a sequence of powers of 2, e.g. 8, 4, 2, 1 should be avoided because then the same keys would be compared repeatedly on every pass.

Shell sort is named after its inventor Donald Shell. It is one of the first algorithms to break the quadratic time barrier. However, it was not until several years later that this breakthrough was proved. Shell sort is sometimes called *diminishing increment sort*.

The main ideas are:

1. For every i, $A[i] \leq A[i + h_k]$ for each h_k after sorting stage k.
2. All elements h_k apart are sorted by insertion sort.
3. The increment sequence choice (by Shell) was $h_1 = \lfloor n/2 \rfloor$ and $h_k = \lfloor h_{k-1}/2 \rfloor$, where $k = 2, \ldots, t$ and $h_t = 1$.
4. An example of generating the increment sequence is:

$$h_k \leftarrow h_{k-1} \text{ div } 3 + 1$$

We outline the algorithm below:

```
1: choose the initial increment
2: repeat
3:    increment ← increment div 3 + 1
4:    for start ← 1, start < increment, start ← start + 1 do
5:       insertSort(start, increment, L)
         {where L is the list of items to sort}
6:    end for
7: until increment = 1
```

Algorithm analysis

Shellsort is simple to code but the analysis of its running time turns out to be exceedingly difficult. This is mainly because the running time of Shellsort depends on the choice of increment sequence. The average case analysis of Shellsort is unkonwn, except for very trivial increment sequences.

Summary of Shellsort

1. Shellsort sorts a list of n keys by successively sorting sublists whose elements are intermingled in the whole list.
2. The sublists are determined by an *increment sequence*, $h_1, \ldots, h_t$.
3. Some choices of increment sequence are better than others. Empirical studies show: for large n, the number of moves is in the range of $n^{1.25}$ to $1.6n^{1.25}$.
4. Shellsort is a substantial improvement over insertion sort in general.

6.8 Mergesort

The name Mergesort comes from the idea of taking two sorted lists of elements and *merging* them into a single sorted list. This is used for external sorting.

Merge two sorted lists

When merging two sorted lists l and r, the first entry of the result will be the smaller of the first elements of the two lists. After it is taken, the second element of the result list will be the smaller of what remains. This process repeats until the end of one list is reached. Finally the remains of the other list will be appended to the result list (See example 3.12).

Algorithm 6.8 merges two lists l and r into a result list c.

Algorithm 6.8 combine(queue l, r, c)

```
 1: while not (end of l or r) do
 2:    if front(l).key < front(r).key then
 3:       enqueue(dequeue(l), c)
 4:    else
 5:       enqueue(dequeue(r), c);
 6:    end if
 7:    if (end of l) then
 8:       append(r, c)
 9:    else
10:       append(l, c)
11:    end if
12: end while
```

Implementation

We use l, r to trace the 'current' node on the two sorted lists $lList$ and $rList$ respectively for comparison, and c points to the head of the merged list and s traces the last node of the combined list.

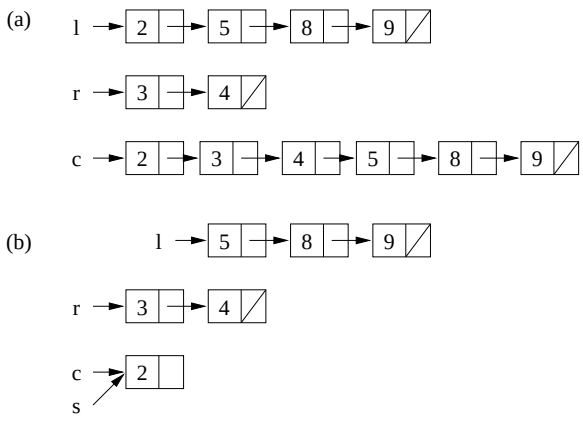


Figure 6.1: Merging two sorted lists

Algorithm 6.9 rearranges two sorted lists l and r to a new sorted list c. We assume that neither $lList$ nor $rList$ is empty. [2]

[2] Of course, if one of them is empty, the combined list is simply the other one.

Algorithm 6.9 combine2(node l, r, c)

```
1: if (l ≠ null) and (r ≠ null) then
2:    if l.data ≤ r.data then
3:       c ← l, l ← l.next
4:    else
5:       c ← r, r ← r.next
6:    end if
7:    s ← c
8:    while l ≠ null and r ≠ null do
9:       if l.data ≤ r.data then
10:         s.next ← l, s ← l, l ← l.next
11:      else
12:         s.next ← r, s ← r, r ← r.next
13:      end if
14:   end while
15:   if l = null then
16:      s.next ← r
17:   else
18:      s.next ← l
19:   end if
20: end if
```

Mergesort a list

The idea of Mergesort is to divide the list into halves (or two as equal ones as possible), then sort the two sublists recursively. Finally, the two sorted sublists are merged into one sorted list. Mergesort is a good example of a recursive algorithm and divide and conquer approach (see Example 3.13).

Algorithm 6.10 mergeSort(list)

```
1: if the List has length > 1 then
2:    partition the list into two sublists lList and rList
3:    mergeSort(lList) (recursively)
4:    mergeSort(rList) (recursively)
5:    combine(lList, rList, list)
6: end if
```

Algorithm analysis

There are several ways to analyse the MergeSort algorithm. We introduce two methods:

1. Compute recurrence equations:
   Let T(n) be the time complexity for a list of size n.
   We have $T(1) = 1$ and $T(n) = 2T(n/2) + n$. The last equation says that the total time for sorting a list of n elements is the sum of the time for sorting the two lists of a half of n elements and the time for merging the two sorted lists. The solution of the equations is $T(n) = n \log n + n$ which is $O(n \log n)$.
2. Counting comparisons:
   - Comparisons are done in Combine procedure.
   - After each comparison, one of the two keys is sent to the result list. So the number of comparisons $\leq$ the length of

the list n.

- A so-called *recursion tree* can be drawn to demonstrate that the depth of the tree is $\log n$.
- The total number of comparisons done on each level $\leq n$.

Therefore, the total number of comparisons is $\leq n \log n$, i.e. $O(n \log n)$.

Summary of Mergesort

- Although its running time is $O(n \log n)$, Mergesort is hardly ever used for main memory sorts. This is because the method firstly requires linear extra memory.
- Secondly, the sort actually has been slowed down considerably while copying elements to the temporary array and back throughout the algorithm.

6.9 Quicksort

Quicksort is the *fastest* known sorting algorithm *in practice*. For serious internal sorting applications, the algorithm of choice is Quicksort. It has been used in some high-leveled language such as *Perl* as a standard operation.

Quicksort uses a similar approach to Mergesort, namely *divide and conquer* (Section 3.4). This strategy decomposes the input into several smaller inputs, solves the smaller problems recursively, and combines the solutions to obtain a solution for the original input.

Instead of splitting the list by length, Quicksort splits the lists each time by the key value of each item compared with a "standard". Such a standard element is called the *pivot* of the list and can be randomly selected. All the items whose keys are smaller than pivot will be relocated to its left sublist and all the items with keys that are greater than the pivot will be relocated to its right (See Example 3.15).

This idea works, however, how would we decide the pivot location?

Suppose the list is stored in a one dimensional array $A[1..n]$ of n integers. One way to find the pivot location is to review each pair of the elements starting from both ends of the array and moving towards each other. Each time, we make sure the smaller integer goes to `lList`, the left sublist; and the larger one goes to `rList`, the right sublist. We define two pointer variables l and r as the indices of the two elements under review. If $A[l] > A[r]$, we swap the two elements. Initially, l and r point to $A[1]$ and $A[n]$ respectively, i.e. to both ends of the array, and then move toward each other until they are equal. When $l = r$, the location pointed by both indices is the pivot location.

Example 6.5 demonstrates how the pivot location can be found by moving two pointers `l` and `r` from both ends of the array A[1..7] toward each other until they meet. The pivot location is at $l = r = 6$.

Example 6.5 *Find the pivot location 33 26 35 29 19 12 22, where the first element is the pivot as underlined.*

	i	1	2	3	4	5	6	7	
(1)	A[i]	33	26	35	29	19	12	22	A[l]>A[r]
		l						r	
		22	26	35	29	19	12	33	
		l						r	
		22	26	35	29	19	12	33	
			l					r	
(2)		22	26	35	29	19	12	33	A[l]>A[r]
				l				r	
		22	26	33	29	19	12	35	
				l				r	
		22	26	33	29	19	12	35	
				l			r		
(3)		22	26	33	29	19	12	35	A[l]>A[r]
				l			r		
		22	26	12	29	19	33	35	
				l			r		
		22	26	12	29	19	33	35	
					l		r		
(4)		22	26	12	29	19	33	35	
							l r		

As we can see, in stage (1), $l = 1$, $r = 7$ and $p = l$, where pointer p refers to the pivot location. Since $A[l] > A[r]$, we swap them. Now $A[p] = A[r]$, so we move l one position to the right, i.e. $l \leftarrow l + 1$.

In stage (2), l is moved toward the right until $A[l] > A[r]$, and we swap them. Now $A[p] = A[l]$, so we move r one position to the left, i.e. $r \leftarrow r - 1$.

In stage (3), r is moved toward the left, but failed because $A[l] > A[r]$; so we swap them. Now $A[p] = A[r]$, and we move l one position to the right, i.e. $l \leftarrow l + 1$.

Algorithms

There are many versions of Quicksort and many more implementations. Here we concentrate on the ideas behind Quicksort.

Algorithm 6.11 quickSort(index first, last)

```
1: if first < last then
2:     quickSort(first, splitPoint(A,first,last)-1)
3:     quickSort(splitPoint(A,first,last)+1, last)
4: end if
```

Algorithm 6.12 splitPoint(int A, index l, r)

```
1: while (l < r) do
2:    while (l < r) && (A[p] < A[r]) do
3:       r − −
4:    end while
5:    if (l < r) then
6:       swap(A, p, r); p ← r; l + +
7:    else
8:       p ← l
9:    end if
10:   while (l < r) && (A[p] > A[l]) do
11:      l + +
12:   end while
13:   if (l < r) then
14:      swap(A, l, p); p ← l; r − −
15:   else
16:      p ← r;
17:   end if
18: end while
19: return p;
```

Algorithm 6.13 swap(int [] A, index l, r)

```
1: tmp ← A[l]
2: A[l] ← A[r]
3: A[r] ← tmp
```

Algorithm analysis

Similarly to Mergesort, the analysis for Quicksort requires solving a recurrence formula. Here we assume that it takes a constant time for Quicksort on an empty list or a list with one item, i.e. $T(0) = T(1) = 1$. The basic relation is

$$T(n) = T(i) + T(n - i - 1) + cn$$

where i is the number of elements smaller than *pivot*.

Implementation

The implementation becomes easy once the algorithms are well understood. The following Java code requires only a few basic Java statements.

```
int splitPoint (int A[], int l, int r) {
   int p=l;  // splitPoint for pivot
   while (l<r) {
//       move r to the left
      while (l<r && A[p]<A[r]) {
         r-- ;

         }
      if (l<r) {
         swap(A, p, r);
         p=r; l++;
```

```
            }
         else p=l;

//         move l to the right
         while (l<r && A[p]>A[l]) {
            l++ ;
            }
         if (l<r) {
            swap(A, l, p);
            p=l; r--;
            }
         else p=r;

      }
      return p;
   }

   void swap(int A[], int l, int r) {
      int tmp=A[l];
      A[l]=A[r];
      A[r]=tmp;
   }

   void quickSort(int A[], int first, int last) {
      if (first<last) {
         quickSort(A,first,splitPoint(A,first,last)-1);
         quickSort(A,splitPoint(A, first, last)+1,last);
      }
   }
```

Summary of Quicksort

1. Quicksort is another example of a recursive algorithm using the *divide and conquer* strategy.
2. Quicksort is the fastest known general sorting algorithm in practice on average.
3. The worst case is as bad as the worst case of selection sort ($O(n^2)$).

6.10 General lower bounds for sorting

We have seen several algorithms for sorting a list of elements. Among them, Shellsort, Quicksort and Mergesort are faster in general than Insertion Sort and Selection Sort. It is natural to ask if we can find another method that is even faster.

Unfortunately, every problem has an inherent complexity and there is some minimum amount of work (hence resources) required to solve a problem. Hence we may ask how many operations the algorithms actually *need* to solve the problem.

If we can prove a theorem of the form "All algorithms for problem P have the complexity $T(n) \geq f(n)$", then function $f(n)$ (whether an algorithm of complexity $f(n)$ for P has been found or not) is called a *lower bound* on the complexity of P (see Section 8.6 for a general

introduction).

An algorithm is said to be *optimal* (in the worst case) if there is no other algorithm in the class (including those that have not been found yet) that performs fewer basic operations (in the worst case). In other words, if we have fortunately developed an algorithm whose complexity reaches the *lower bound* for P, then the algorithm is optimal: it cannot be bettered.

Decision trees

In order to have an overview of all possible branches after each comparison, we define a tree structure recursively by associating each node with a comparison.

A *decision tree* is an effective abstraction used to prove lower bounds. It can be proved that every algorithm that sorts by using only comparisons can be represented by a decision tree. In our context, a decision tree is a binary tree. Each node represents a comparison and each edge the result (true or false) of the comparison. The nodes on the path from the root to current node represent a set of possible orderings, consistent with comparisons made at that stage, among the elements.

Each node may have:

- no children: the algorithm halts
- one child: the algorithm can only continue by taking either the left branch or the right one after the comparison
- two children: the algorithm would take the left branch if $A > B$ (i.e. if $A \leq B$ is false (F)), or take the right branch otherwise.

We give an example of the decision tree for sorting three elements.

Example 6.6 *A decision tree for three-element insertion sort (Figure 6.2).*

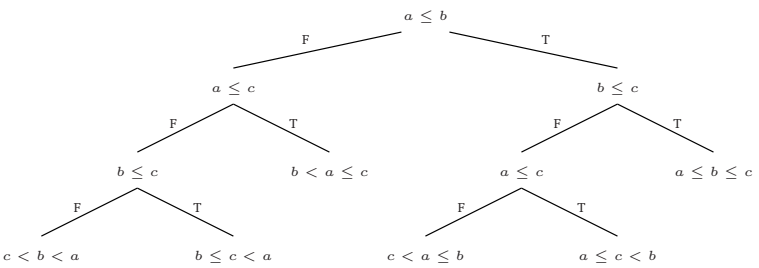


Figure 6.2: Comparison trees for Insertion Sort ($n = 3$)

The worst and average cases

The decision tree of any sorting algorithm displays several features of the algorithm. We summarise the correspondences between the decision tree and the comparisons in the algorithm as follows:

- Path from the root to a leaf: the actions of the sorting algorithm on a particular input.
- Number of nodes on a path from the root to a leaf: the number of comparisons on a particular input.
- Depth of the tree: the number of comparisons for the worst case.
- Average length of all the paths from the root to a leaf: the number of comparisons for the average case.

The following theorem states the general lower bound for sorting algorithms by comparisons.

Theorem: Any algorithm that sorts a list of n items by use of key comparisons must, in its worst case, perform at least $\lceil \log n! \rceil$ comparisons of keys; and in the average case, it must perform at least $\log n!$ comparisons of keys.

6.11 Bucket sort

With extra information about the keys to sort, it is possible to sort in linear time in special cases *without comparisons*. Bucket sort is such a simple example.

The following information are sufficient to be known:

- The keys to sort are positive integers only;
- The upper bound of the integers (let it be m) is known;
- It is feasible to use an array A[$1..m$] of counters.

Example 6.7 *If we know in advance that there are 100 items with distinguished keys of integers within the range [15, 120]. The best way to sort the 100 items is to define an array indexed 15..120. If a particular item has key i, then place '1' in location i. At the end of the process, the table contains 100 '1's. We can subsequently display all the indices where a '1' is stored in order of the indices.*

	15	16	17	18	···	118	119	120
A	1	0	1	1	···	0	1	0

Figure 6.3: Bucket sort

So the output of the sorted list is (15, 17, 18, ···, 119)

Bucket sort is sometimes regarded as too trivial an algorithm to be useful. However, there are many cases in practice where the input is indeed only small integers. We should not forget about this useful method in these cases.

Note This does not violate the lower bound.

Radix sort

Radix sort is an extension of Bucket sort. It assumes that the keys are d-digit numerals in base r (which is also called *radix*). Any digit

x_i of a key satisfies $1 \leq x_i \leq r - 1$, and so may be used to index an array of r linked lists.

The basic idea of radix sort is to make one pass through the entries, placing each entry at the back of the x_ith linked list, where x_i is the ith digit of the entry's key.

6.12 Sorting large records

We have assumed, during the discussion of sorting, that the elements to be sorted are simply integers. In practice, however, we often need to sort large records by a certain key. For example, we might have student records each of which consists of a name, age, sex, year, address, phone number, financial information, grades for each subject every year. We might want to sort the records by one particular field. In this case, swapping two records would involve moving many data.

A practical solution is to use *indirect sort*, i.e. to have the input array contain pointers to the records. We sort by comparing the keys the pointers point to, and only swap pointers if necessary.

Example 6.8 *Suppose we have a large array of records with a key field as in Figure 6.4.*

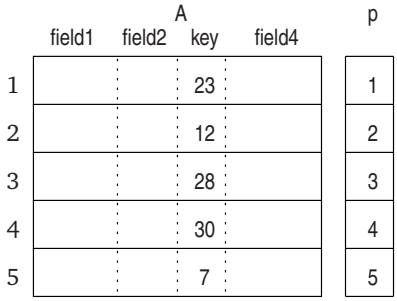


Figure 6.4: Sorting large records using pointers

We store the indices of the records in an one dimensional array P initially. It is updated after a swap between two records in the array A. Suppose that we apply `selection` *sort on the elements in record array A. Instead of comparing $A[i].key < A[j].key$, we compare $A[P[i]].key < A[P[j]].key$ each time for any two keys with indices i and j. Similarly, instead of swapping the large record $A[i]$ with $A[j]$,*

we swap merely pointers $P[i]$ with $P[j]$ each time.

i	*1*	*2*	*3*	*4*	*5*
A[i].key	*23*	*12*	*28*	*30*	*7*
P[i]	*1*	*2*	*3*	*4*	*5*
P[i]	*1*				
	2	*1*			
	2	*1*	*3*		
	2	*1*	*3*	*4*	
	5	*1*	*3*	*4*	*2*
	5	*1*			
	5	*1*	*3*		
	5	*1*	*3*	*4*	
	5	*2*	*3*	*4*	*1*
	5	*2*	*3*		
	5	*2*	*3*	*4*	
	5	*2*	*1*	*4*	*3*
	5	*2*	*1*	*4*	
	5	*2*	*1*	*3*	*4*

The sorted list of large records can be derived by Algorithm 6.14. In this example, $n = 5$.

Algorithm 6.14 displayRecords()

1: **for** $i \leftarrow 1, i \leq n, i \leftarrow i + 1$ **do**
2: print($A[P[i]].field1, A[P[i]].field2, A[P[i]].key, A[P[i]].field4$)
3: **end for**

6.13 Heapsort

In this section, we will look at a sorting algorithm on a special binary tree structure.

Recall that a *binary heap* (Section 4.4) is a binary tree with some special properties including the *structure property* and *order property*:

Structural property A heap is a *complete binary tree*, i.e. a binary tree that is completely filled except possibly the bottom level which is filled from left to right.

Order property The key value of each node is at most (or at least) as large as the keys of its children.[3]

[3] Note this is more relaxed than the order property of the BST (Section 5.7).

Informally, a heap is a complete binary tree possibly with some of the rightmost leaves removed.

Example 6.9 *Figure 6.5 gives an example of a binary heap.*

Ideas of Heapsort

If the keys can be stored in a heap, then we could get a sorted list by repeatedly removing the key from the root (the minimum remaining

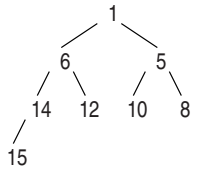


Figure 6.5: A binary heap

key), copying it to the output array, and rearranging the keys left in the heap to reestablish the heap properties.

Example 6.10 *We show how elements in the heap in Figure 6.5 can be sorted.*

1. *Remove the root 1 and copy it to* `Output`*. Move the leftmost leaf 15 at the bottom level (i.e. the last leaf in the table) to the root and maintain the order property by swapping with its child with the smaller data value (smaller child) if it exists, i.e. swap(15,5), swap(15,8) (Figure 6.6).*

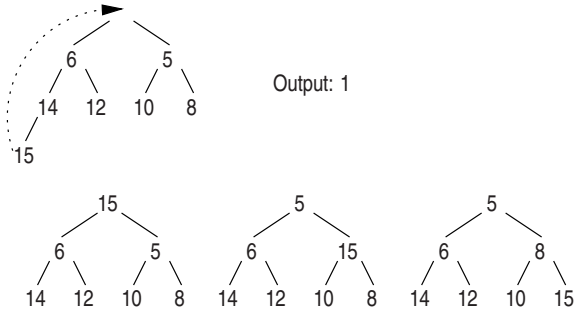


Figure 6.6: Remove 1 from the heap and reestablish the heap properties.

2. *Remove the root 5 and copy to* `Output`*. Move the last leaf 15 to the root and maintain the order property by swapping with its smaller child if it exists, i.e. swap(15,6), swap(15,12) (Figure 6.7).*

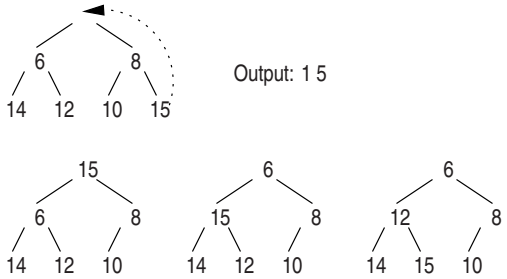


Figure 6.7: Remove 5 from the heap.

3. *Remove the root 6 and copy to* `Output`*. Move the last leaf 10 to the root and maintain the order property by swapping with its smaller child if it exists, i.e. swap(10,8) (Figure 6.8).*
4. *Remove the root 8 and copy to* `Output`*. Move the last leaf 15 to the root and maintain the order property by swapping with its smaller child if it exists, i.e. swap(15,10) (Figure 6.9).*

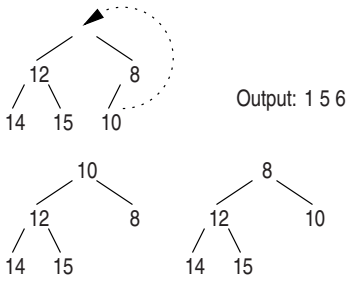


Figure 6.8: Remove 6 from the heap.

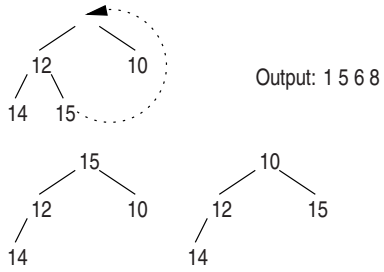


Figure 6.9: Remove 8 from the heap.

5. *Remove the root 10 and copy to* `Output`. *Move the last leaf 14 to the root and maintain the order property by swapping with its smaller child if it exists, i.e. swap(14,12) (Figure 6.10).*

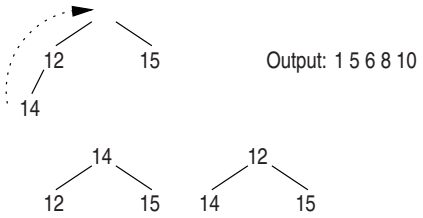


Figure 6.10: Remove 10 from the heap.

6. *Remove the root 12 and copy to* `Output`. *Move the last leaf 15 to the root and maintain the order property by swapping with its smaller child if it exists, i.e. swap(15,14) (Figure 6.11).*
7. *Remove the root 14 and copy to* `Output`. *Move the last leaf 15 to the root. Since 15 is the only element in the heap, we copy it to the* `output` *(Figure 6.12).*

Implementation

There are ways other than the usual linked structures to implement binary trees. For example, we could store a complete binary tree in a one-dimensional array A by first labelling the tree nodes, beginning with the root, from left to right on each level, then storing each node in the position shown by its label.

Note: the parent of an element A[i] is A[i div 2] if $i > 1$; the left child of A[i] is A[2i] and the right child is A[2i + 1] if they exist.

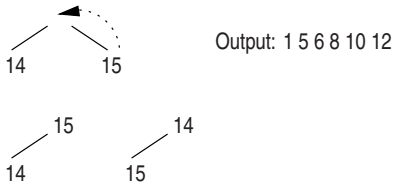


Figure 6.11: Remove 12 from the heap.

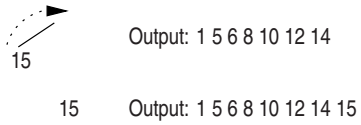


Figure 6.12: Remove 14 from the heap and output 15.

Example 6.11 *Figure 6.13 illustrates how an array can be used to implement a binary heap with a* Min*Key root, and Figure 6.14 with a* Max*Key root.*

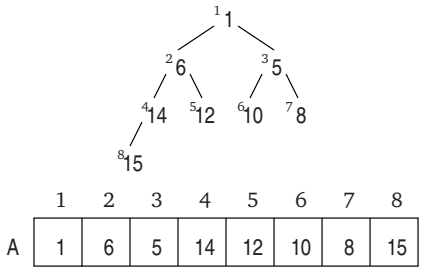


Figure 6.13: A heap with a MinKey root

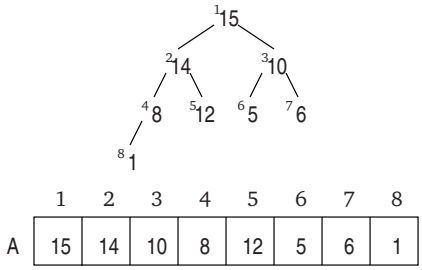


Figure 6.14: A heap with a MaxKey root

Now an array being sorted can be interpreted as a binary tree in a contiguous implementation. The next thing that we need to do is to maintain the order property of the heap by maintaining the array. Algorithm 6.15 shows the idea.

Algorithm 6.15 heapSort(heapArray A)

```
1: item tmp, index k
2: buildHeap()
   {Construct a heap}
3: for k ← Length(A), k < 2, k ← k − 1 do
4:    tmp ← A[k]
5:    A[k] ← A[1]
6:    A[1] ← tmp
      {swap A[1] with A[k]}
7:    addRoot(1, k − 1)
8: end for
```

Basic heap operations

In order to maintain the order property, the following operations are required:

addRoot: insert an element into the heap as a root and restore the heap properties.

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

We copy Algorithm 4.8 and 4.9 here (Algorithm 6.16 and 6.17) for convenience of discussion:

Algorithm 6.16 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
11:   end if
12: end if
```

Algorithm 6.17 buildHeap()

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

Note For implementation convenience, we build a heap with the MaxKey as the root (Figure 6.14). There are two improvements in this implementation.

1. It uses addRoot operations instead of repeated insertions to build the heap, i.e. to build the heap from a list consisting of n elements in arbitrary order.
2. It does not use an extra array for `Output`, but stores the extracted maximum elements at the top of the array as the heap shrinks.

Example 6.12 *Suppose an integer list* $L = (1, 6, 5, 14, 8, 10, 12, 15)$ *is stored in array* A*. Trace the content of the array* A *on execution of each iteration of the algorithm* `heapSort(L)`.

The content of array A *and the values of* `k`*,* `i`*,* `j` *are listed below. You should attempt the task of tracing the content first and the following answer is best used for checking your own solution.*

1. *Call buildHeap():*

```
index       1  2  3  4  5  6  7  8     k   i   j
                      k
A[index]    1  6  5  14 8  10 12 15
                      i           j     4   4   8
A[index]    1  6  5  15 8  10 12 14        8   16 >n=8

                   k
A[index]    1  6  5  15 8  10 12 14
                   i        j          3   3   6
                                  j           7
A[index]    1  6  12 15 8  10 5  14        7   14 >n

               k
A[index]    1  6  12 15 8  10 5  14
               i     j                   2   2   4
A[index]    1  15 12 6  8  10 5  14
                      i           j        4   8
A[index]    1  15 12 14 8  10 5  6         8   16 >n

            k
A[index]    1  15 12 14 8  10 5  6
            i  j                         1   1   2
A[index]    15 1  12 14 8  10 5  6
               i     j                      2   4
A[index]    15 14 12 1  8  10 5  6
                      i           j        4   8
A[index]    15 14 12 6  8  10 5  1         8   16 >n
```

2. *Execute the remaining algorithm:*

```
index       1  2  3  4  5  6  7  8     k   i   j
                                  k
A[index]    15 14 12 6  8  10 5  1     8
A[index]    1  14 12 6  8  10 5  15
            i  j                            1   2
A[index]    14 1  12 6  8  10 5  15
               i     j                      2   4
                        j                  2   5
A[index]    14 8  12 6  1  10 5  15        5   10 >k-1

                              k
A[index]    14 8  12 6  1  10 5  15    7
A[index]    5  8  12 6  1  10 14 15
            i  j                         1   2
                  j                          3
A[index]    12 8  5  6  1  10 14 15
                  i        j           3   6
A[index]    12 8  10 6  1  5  14 15        6   12 >k-1

                           k
```

```
A[index]    12 8  10 6  1  5  14 15    6
A[index]    5  8  10 6  1  12 14 15
            i  j                          1   2
                  j                       3
A[index]    10 8  5  6  1  12 14 15       3   6 >k-1
                     k
A[index]    10 8  5  6  1  12 14 15    5
A[index]    1  8  5  6  10 12 14 15
            i  j                          1   2
A[index]    8  1  5  6  10 12 14 15
               i     j                    2   4
A[index]    8  6  5  1  10 12 14 15       4   8 >k-1

                     k
A[index]    8  6  5  1  10 12 14 15    4
A[index]    1  6  5  8  10 12 14 15
            i  j                          1   2
A[index]    6  1  5  8  10 12 14 15       2   4 >k-1

                  k
A[index]    6  1  5  8  10 12 14 15    3
A[index]    5  1  6  8  10 12 14 15
            i  j                          1   2
A[index]    5  1  6  8  10 12 14 15       2   4 >k-1

               k
A[index]    1  5  6  8  10 12 14 15    2   1   2 >k-1
```

Worst case complexity

When n is as small as the example that we looked at earlier, Heapsort seems not particularly efficient: we sort by moving large keys slowly toward the beginning of the array before finally putting them away at the end. However, when n becomes large, such small quirks are not significant. Heapsort is actually one of very few sorting algorithms for contiguous lists that is guaranteed to finish in time $O(n \log n)$ with minimal space requirements.

It can be shown that the worst case complexity for Heapsort is $O(n \log n)$. The worst case for Heapsort is poorer than the average case for Quicksort, but Quicksort's worst case is far worse than the worst case of Heapsort for large n.

Activity 6.13

SORTING

1. Trace through the steps, by hand, or implementation of a program, that each of the sorting algorithms in Chapter 6 will use on each of the following lists. In each case, count the number of comparisons that will be made and the number of times an item will be moved.

   (a) The three words, (*triangle, square, pentagon*), to be sorted alphabetically.

(b) The three words, (*triangle, square, pentagon*), to be sorted according to the number of sides of the corresponding polygon, in *increasing* and *decreasing* order.

(c) The even numbers, (*26, 33, 35, 29, 19, 12, 22*), to be sorted into increasing order.

(d) The list of 14 names, (*Tim, Dot, Eva, Roy, Tom, Kim, Guy, Amy, Jon, Ann, Jim, Kay, Ron, Jan*), to be sorted into alphabetical order.

2. Implement and test Algorithm 6.10 for the merge sort.
3. Write classes to sort an array of integers by *three different algorithms*, e.g. selection sort, insertion sort, and quick-sort. Compute

   (a) the number of comparisons made;

   (b) the number of times that an item (say, the first item) is moved in each run of the program using the three sorting algorithms respectively.
4. Demonstrate how the heapsort algorithm works step by step with an example of 10 different characters.

Chapter 7

Optimisation problems

7.1 Essential reading

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 6.5

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 10.3

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 9.4

7.2 Learning outcomes

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

- describe the characteristics of optimisation problems
- outline the ideas of the greedy approach
- identify the problems for which a greedy approach may be suitable to use
- implement an algorithm that applies greedy approaches.

7.3 Optimisation problems

An optimisation problem is a problem that requires finding the maximum (or minimum) value according to some specified criteria. Optimisation problems form a special class where the prime goal is to find the 'best' of all possible answers. They are common in the real world.

Example 7.1 *A list of optimisation problems:*

1. *A manufacturing boss wants to produce several products to get the most value out of the limited number of hours.*
2. *A salesperson wants to find a best route on which he can visit all the cities on his sales list but spend the least amount of travel time.*
3. *A university wants to schedule the maximum number of classes with different requirements for facilities and with no class conflicts.*
4. *A student wants to achieve the highest average grade for a number of subjects but spend a limited number of study hours.*

Solving an optimisation problem often involves maximising (or minimising) the value of a predefined variable. The values of a set

of variables are determined by a set of *constraint functions*. Often the solutions to some optimisation problems can be adopted to solve others.

Examples of classic optimisation problems are:

minimum spanning tree problem Given a weighted simple graph, we want to find the minimum spanning tree.

knapsack problem Given a knapsack of volume n, and a number of objects of values $v_1, v_2, \cdots$, find the most valuable set of objects that fit in the knapsack.

travelling salesman problem Find a path through a weighted graph which starts and ends at the same vertex, includes every other vertex exactly once, and minimises the total cost of edges.

Optimisation problems are important with wide applications because of the potential gains in efficiency according to certain definitions. In this text, the solutions of an optimisation problem are algorithms that can find or achieve the maximum or minimum values and satisfy the predefined constraints of the problem. Unfortunately, most optimisation problems cannot be solved in polynomial time as we will see later.

Optimisation problems can be described naturally by asking the right questions. Let us see a few more examples first, namely, *Multiplication of matrices, Knapsack problem* and *Coin changes*.

7.3.1 Multiplication of matrices

Consider the multiplication of a number of matrices:

Suppose that we have four matrices A, B, C and D of dimensions 2×1, 1×5, 5×2 and 2×3 respectively, denoted by $A_{2\times1}, B_{1\times5}, C_{5\times2}, D_{2\times3}$. We want to compute the multiplication $ABCD$.

In which order of multiplying the matrices would the number of total operations ($\times$, $+$) be the smallest?

Since matrix multiplication is *associative*, we can conduct the multiplication between any two adjacent matrices prior to others. The final result will be the same. For example, we can multiply AB before multiplying C and finally D. That is, $((AB)C)D$. We can also follow another order: $A((BC)D)$, that is, multiply BC before multiply D, and finally multiply A and the product.

What is the difference? Let us look at an example:

Example 7.2 *Suppose that* $A_{2\times1} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$,

$B_{1\times5} = \begin{pmatrix} 2 & 1 & 3 & 1 & 3 \end{pmatrix}$, $C_{5\times2} = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 1 & 2 \\ 2 & 3 \\ 1 & 2 \end{pmatrix}$, $D_{2\times3} = \begin{pmatrix} 2 & 2 & 1 \\ 3 & 2 & 1 \end{pmatrix}$

If we follow the order of $((AB)C)D$, we can simply count the number of operations of multiplications ($\times$)and additions ($+$) and

have:

Matrices	Operations	Num of $\times$	Num of $+$
AB	$\begin{pmatrix}2\\3\end{pmatrix}\begin{pmatrix}2&1&3&1&3\end{pmatrix}$	$2\times1\times5$	$2\times0\times5$
(AB)C	$\begin{pmatrix}4&2&6&2&6\\6&3&9&3&9\end{pmatrix}\begin{pmatrix}1&2\\2&3\\1&2\\2&3\\1&2\end{pmatrix}$	$2\times5\times2$	$2\times4\times2$
((AB)C)D	$\begin{pmatrix}24&44\\36&66\end{pmatrix}\begin{pmatrix}2&2&1\\3&2&1\end{pmatrix}$	$2\times2\times3$	$2\times1\times3$

If we follow the order of $A((BC)D)$, we have:

Matrices	Operations	Num of $\times$	Num of $+$
BC	$\begin{pmatrix}2&1&3&1&3\end{pmatrix}\begin{pmatrix}1&2\\2&3\\1&2\\2&3\\1&2\end{pmatrix}$	$1\times5\times2$	$1\times4\times2$
(BC)D	$\begin{pmatrix}12&22\end{pmatrix}\begin{pmatrix}2&2&1\\3&2&1\end{pmatrix}$	$1\times2\times3$	$1\times1\times3$
A((BC)D)	$\begin{pmatrix}2\\3\end{pmatrix}\begin{pmatrix}90&68&34\end{pmatrix}$	$2\times1\times3$	$2\times0\times3$

In general, $A_{i\times j}B_{j\times k}$ results in a matrix of $i\times k$ and requires $i\times j\times k$ multiplications and $i\times(j-1)\times k$ additions.

In fact, we only need to consider about the number of multiplications, because the number of +'s is always fewer than that of $\times$'s, and it takes longer to complete a multiplication than addition. We have the following result:

Matrices	Num of $\times$	$=$
$((A_{2\times1}B_{1\times5})C_{5\times2})D_{2\times3}$	$2\times1\times5+2\times5\times2+2\times2\times3$	42
$(A_{2\times1}(B_{1\times5}C_{5\times2}))D_{2\times3}$	$1\times5\times2+2\times1\times2+2\times2\times3$	26
$A_{2\times1}((B_{1\times5}C_{5\times2})D_{2\times3})$	$1\times5\times2+1\times2\times3+2\times1\times3$	22
$A_{2\times1}(B_{1\times5}(C_{5\times2}D_{2\times3}))$	$5\times2\times3+1\times5\times3+2\times1\times3$	51
$(A_{2\times1}B_{1\times5})(C_{5\times2}D_{2\times3})$	$2\times1\times5+5\times2\times3+2\times5\times3$	70

We now can conclude that $A((BC)D)$ is the best order for the matrix multiplication because it requires the minimum number (22) of multiplications.

A good order of the multiplication can make a huge difference in term of computational efficiency. Even for a very small example like this, we notice a big difference between the number of the multiplications (and additions).

7.3.2 Knapsack problem

We want to have a nice holiday but are only allowed to take one knapsack. Suppose that we have n objects of different sizes and values. We now can only take some of them with us. So here is the question:

Which objects should we take in order to make a good use of the knapsack, i.e. to take with us the most 'valuable' set possible?

Here the 'valuables' can be defined in many ways to mean, for

example, the most favourite objects, the most useful tools, or simply the most expensive items. The goal is, however, to select a subset of the objects with the maximum total value to fit in the knapsack. The total size of the objects must not exceed the size limit of the knapsack.

We first look at a simplified version of the problem, where the size equals the value of the object, and the size is of one-dimension. That is, the longer in size, the more valuable the object is. We represent an object by a string, where the number of letters represents the size (length) and the value. Here is an instance:

Example 7.3 *Consider a knapsack of size* $K = 12$ *and five objects* `AAA`, `BBBB`, `CCCCC`, `DDDDDDD` *and* `EEEEEEEEE` *of length 3, 4, 5, 7 and 9 respectively as follows. What is the most valuable subset to fit in the knapsack?*

No	*Object*	*length and value*
1	*AAA*	*3*
2	*BBBB*	*4*
3	*CCCCC*	*5*
4	*DDDDDDD*	*7*
5	*EEEEEEEEE*	*9*

An easy way to find the most valuable set is to list all the possibilities. This approach is called *exhaustive-search*. We start from the most expensive item. For each object, we will see if any other object can fit in the remaining room. For example, we first take object 5, the most valuable single item and put it in the knapsack. We then take object 4, the next most valuable single item, but found it cannot fit in. So, we try object 3, the next most valuable item, and then object 2, and 1, etc.

We have:

Subset	Knapsack content	Total size	Total value
5,4	EEEEEEEEEDDDDDDD	16	not fit
5,3	EEEEEEEEECCCCC	14	not fit
5,2	EEEEEEEEEBBBB	13	not fit
5,1	EEEEEEEEEAAA	**12**	**12**
4,5	DDDDDDDEEEEEEEEE	16	not fit
4,3	DDDDDDDCCCCC	**12**	**12**
4,2	DDDDDDDBBBB	11	11
4,1	DDDDDDDAAA	11	11
3,2	CCCCCBBBB	9	9
3,1	CCCCCAAA	8	8
2,1	BBBBAAA	7	7
4,2,1	DDDDDDDBBBBAAA	14	not fit
3,2,1	CCCCCBBBBAAA	**12**	**12**
3,1,4	CCCCCAAADDDDDDD	15	not fit
2,1,5	BBBBAAAEEEEEEEEE	16	not fit

Now we can conclude that we can take two objects 1 and 5, or two objects 3 and 4, or three objects 1, 2 and 3.

We have just seen an instance of the simplified version of the knapsack problem. However, the same approach can be used for the situation where the value and the size of the objects are different.

Example 7.4 *Given a knapsack of size $K = 12$ and $n = 5$ objects (object $1, \cdots, n$) of lengths 3, 4, 5, 7 and 9, with values (£) of 80,20,10,40 and 5 respectively, what is the most valuable subset to fit in the knapsack?*

Object	length	Value £
1	3	80
2	4	20
3	5	10
4	7	40
5	9	5

Table 7.1:

We have

Subset	*Total size*	*Total value*
5,4	*16*	*not fit*
5,3	*14*	*not fit*
5,2	*13*	*not fit*
5,1	***12***	5 + 80 = 85
4,3	***12***	40 + 10 = 50
4,2	*11*	*60*
4,1	*10*	40 + 80 = 120
3,2	*9*	*30*
3,1	*8*	*90*
2,1	*7*	*100*
4,2,1	*14*	*not fit*
3,2,1	***12***	10 + 20 + 80 = 110
3,1,4	*15*	*not fit*
2,1,5	*16*	*not fit*

We can therefore conclude that we should take object 1 and 4.

It is interesting to see that the most valuable subset of the objects does not have to be the objects that fill exactly the knapsack any more.

The knapsack problem has many versions because the size can be interpreted to anything that can be a limit, for example, the weight, or size in from one to many dimensions. The valuables can be interpreted as anything that can be measured quantitatively. The question to be asked can be to fit in ($\leq$) or exactly fit/fill (=) or fill at least ($\geq$) the limit of the capacity of literally anything.

7.3.3 Coin changes

You go shopping and you need to pay, say £2.54. Handing over £10 cash, you would receive changes of £7.46. This consists of, for example, coins: $7 \times$ £1, $4 \times 10p$, and $1 \times 5p$ and $1 \times 1p$, a total of 7 + 4 + 1 = 12 coins. Alternatively, you may receive: $14 \times 50p, 2 \times 20p$ and $6 \times 1p$ coins, a total of 14 + 2 + 6 = 22 coins.

Suppose that you hate carrying coins around, hence you want to have change consisting of as few coins as possible.

The question for the optimisation problem is: what choice of the change consists of the minimum number of coins?

While searching for the optimal solution, we notice the following characteristics:

1. The problem has *constraints*. The coins can only be of values £2, £1, 50p, 20p, 10p, 5p, 2p and 1p, and the total value of the change coins has to meet the change value precisely;
2. The problem *cannot be solved in a single step* because of many possible choices;
3. A choice made in *one step affects directly the following choice* in the next step.

This suggests that we need to derive the solution in a number of steps. For each step, we need to identify certain subconstraints and subgains and always select the best subsolution in the hope that this will lead to the best solution. The constraints and gains in each step are usually referred to as *local* and the given constraints and gains for the original problem are *global*.

These are in fact the common characteristics for all optimisation problems, but let us first look at the the solution to the change making problem.

Example 7.5 *Suppose we need to give a change of £5.63.*

We will consider the changes for each digit from the highest value digit to the lowest one. The problem is therefore divided as follows:

1. *Make change of £5*
2. *Make change of* 60p
3. *Make change of* 3p.

We then make choices for each step:

1. *Choices for £5:*
   (a) $2 \times$ £2 + $1 \times$ £1
   (b) $5 \times$ £1
   (c) *etc. (worse choices)*
2. *Choices for* 60p*:*
   (a) $1 \times 50p + 1 \times 10p$
   (b) $3 \times 20p$
   (c) *etc. (worse choices)*
3. *Choices for* 3p*:*
   (a) $1 \times 2p + 1 \times 1p$
   (b) $3 \times 1p$

If we choose the fewest coins for each step, we will have $2 \times$ £2 + $1 \times$ £1, $1 \times 50p + 1 \times 10p$, *and* $1 \times 2p + 1 \times 1p$, *a total of* 2 + 1 + 1 + 1 + 1 + 1 = 7 *coins.*

Using recursion

For a currency with N coins $C_1, C_2, \ldots, C_N$, what is the minimum number of coins needed to make X smallest units of change?

Let the coins be of values 200p, 100p, 50p, 20p, 10p, 5p, 2p and 1p. What is the minimum number of coins needed to make Xp of change?

Suppose all the valid coin values, in pence of UK currency, are stored in descending order in array 'coins', where `coins[0]` $>$ `coins[1]` $> \cdots >$ `coins[7]`:

i	0	1	2	3	4	5	6	7
coins[i]	200	100	50	20	10	5	2	1

A simple approach for the above problem is to use recursion. This involves considering:

- Base case:
  1. If a change of 0p is required, the number of coins for the change is 0;
  2. Otherwise, if a change of a coin value, i.e. 200p, 100p, 50p, 20p, 10p, 5p, 2p or 1p, is required, the number of coins for the change is 1.
- Induction:
  Otherwise,
  1. If a change is smaller than a coin value (e.g. `coins[i]`), meaning the coin is too large to be included in the change, then the number of coins for the change is the number of coins with values ranging from the next coin value `coins[i+1], coins[i+2]` $\cdots$ `coins[7]`=1p.
  2. Otherwise, this is the case when a change is larger than a coin value `coins[i]`. So the minimum number of coins required for the change is K plus the number of coins required for the difference between the change and $K \times$ `coins[i]`, where $K =$ `change div coins[i]`, is the maximum multiple of the current coin value for the change.

The following Java code implements the ideas.

```
int coinChangeRecursive(int change, int[] coins, int i) {
   int K;
   if (change==0) return 0;
   else {
      //    match a single coin i
      if (change==coins[i]) {
         System.out.println("1 x "+coins[i]);
         return 1;
      }
      else {

         if (change<coins[i]) {  //  change < coins[i]
            return coinChangeRecursive(change, coins, i+1);
         }
         else { // (change > coins[i])
            K=change/coins[i];
            System.out.println(K+" x "+coins[i]);
            change=change-K*coins[i];
            return K+coinChangeRecursive(change, coins, i);
            }
       }
   }
}
```

The following program displays the minimum number of coins required for a given change input from the keyboard:

```
import java.util.Scanner;
class changeTest {

   int coinChangeRecursive(int change, int[] coins, int i) {
      int K;
      if (change==0) return 0;
      else {
         //    match a single coin i
         if (change==coins[i]) {
            System.out.println("1 x "+coins[i]);
            return 1;
         }
         else {
            if (change<coins[i]) {  //  change < coins[i]
               return coinChangeRecursive(change, coins, i+1);
            }
            else { // (change > coins[i])
               K=change/coins[i];
               System.out.println(K+" x "+coins[i]);
               change=change-K*coins[i];
               return K+coinChangeRecursive(change, coins, i);
               }
          }
      }
   }


   public static void main( String [] args ) {

      int [] coins = {200, 100, 50, 20, 10, 5, 2, 1};
      // int numberOfDifferentCoins = coins.length;

      Scanner keyboard = new Scanner(System.in);
      System.out.println("Please type a change in pence,
      e.g. 13 to mean '0.13', and press 'return':");
      int change = keyboard.nextInt();

      changeTest test=new changeTest();

      System.out.println("Total number of coins is: "+
               test.coinChangeRecursive(change, coins, 0));
   }
}
```

An alternative approach for solving the change making problem is to use iterations. The following Java code computes the minimum number of coins:

```
   void iterationCoinChange(int change, int [] coins,
                            int [] numberOfCoins) {
     int numberOfDifferentCoins = coins.length;

     for (int i=0; i<numberOfDifferentCoins; i++) {
         numberOfCoins[i]=0;
     }
```

```
    // Look for exact match with any single coin
    for (int i=0; i<numberOfDifferentCoins; i++) {
        if (coins[i]==change) {
           numberOfCoins[i]++;
           System.out.println("1 "+ coins[i]);
           break;
        }
    }

    System.out.println("i: \tn x\tcoin ");
    // No match
    for (int i=0; i<numberOfDifferentCoins; i++) {
        numberOfCoins[i]=change/coins[i];
        if (numberOfCoins[i]>0) {
            change=change-coins[i]*numberOfCoins[i];
        System.out.println(i+": \t"+numberOfCoins[i]
                   +"x\t"+coins[i]+"\t");
        }
    }
    System.out.println("Total number of coins:
                   "+totalNumberOfCoins(numberOfCoins));
}
```

7.4 Greedy approach

A solution is optimal if there is no better solution than it. The greedy approach aims to obtain an optimal solution by attempting at each step a number of local optimal choices. The idea of making a local optimal choice is to obtain the greatest immediate gains at each step in the hope of obtaining a globally optimal solution at the end. Since each choice is made regardless of any future consequence, a greedy approach does not necessarily lead to an optimal global solution. Nevertheless, it usually gives a good solution that, if it is not optimal, is close to the optimal solution.

The greedy approach can be seen from the coin change problem in Example 7.5. When we selected coins in each step, we followed a strategy of choosing the best possible partial solution for that step. For example, in step 1, we chose three coins (2 £2-coins and 1 £1-coin) instead of five coins (5 £1-coins), etc.. Although we did not know the final result, we made the effort towards the final goal of choosing as few coins as possible in each step.

This approach can also be seen from the following Huffman algorithm. An algorithm that applies such a 'greedy' strategy is called a *greedy algorithm*. Huffman coding algorithm is one of the successful examples of the greedy algorithms, for it does give an optimal solution.

7.4.1 Huffman coding

Huffman coding is a frequency based variable length coding scheme. Instead of using a fixed-length coding scheme, such as ASCII code, where 8 bits are used to encode every symbol in an alphabet, the more frequently occurring symbols are encoded by shorter

codewords using fewer bits than the less frequently occurring symbols.

The Huffman algorithm is useful for text file compression. In English text, some characters occur far more frequently than others. For example, characters such as e, a, o, t are used in texts much more frequently than others such as j, q, x. Using the Huffman algorithm, the total number of bits needed would be reduced.

Frequency table

How would we know which symbol occurs more frequently than others? Statistics results from the past (i.e. of symbols processed to date) can be used for an estimation. However, when time is not an issue, an easy approach is to simply count the number of each symbol occurrence in the given text. We show a small example here:

Example 7.6 *Suppose we want to compress a short text message 'BILL␣BEATS␣BEN.'*

Count each symbol in the text and we have the frequency table for the text:

Character	*Frequency*
B	*3*
I	*1*
L	*2*
E	*2*
A	*1*
T	*1*
S	*1*
N	*1*
␣	*2*
.	*1*
Total	*15*

Sort by frequency:

Character	*Frequency*
B	*3*
L	*2*
E	*2*
␣	*2*
I	*1*
A	*1*
T	*1*
S	*1*
N	*1*
.	*1*
Total	*15*

The Huffman compression algorithm will construct a binary tree based on such a frequency table.

Huffman's ideas

We outline the greedy approach of the Huffman algorithm below:

1: Construct a frequency table.
2: Build a *binary tree* from which the individual characters are derived.
Iterations: Until completion of the tree, take the bottom two entries from the frequency table, combine them and update the frequency table.
3: Start at the *root* and trace down to every *leaf*; mark '0' for a *left branch* and '1' for a *right branch*.

Algorithm 7.1 Huffman encoding

INPUT: a sorted list of one-node binary trees $(t_1, t_2, \cdots, t_n)$ for alphabet $(s_1, \cdots, s_n)$ with frequencies $(w_1, \cdots, w_n)$
OUTPUT: a Huffman code with n codewords

1: initialise a list of one-node binary trees $(t_1, t_2, \cdots, t_n)$ with weight $(w_1, w_2, \cdots, w_n)$ respectively
2: **for** $k \leftarrow 1$; $k < n$; $k \leftarrow k + 1$ **do**
3: take two trees t_i and t_j with minimum weights $(w_i \leq w_j)$
4: $t \leftarrow merge(t_i, t_j)$ with weight $w \leftarrow w_i + w_j$,
where $left\_child(t) \leftarrow t_i$ and $right\_child(t) \leftarrow t_j$
5: $edge(t, t_i) \leftarrow 0$; $edge(t, t_j) \leftarrow 1$
6: **end for**
7: output every path from the root of t to a leaf, where $path_i$ consists of consecutive edges from the root to $leaf_i$ for s_i

Figure 7.1 shows an example of how this approach works step by step. We ignore the '·' and the two '␣'s for simplicity.

In step (1), symbols `S` and `N`, the two symbols with the lowest frequency are combined to form a combination `SN` with a frequency of 2, the total frequencies of the two single symbols.

In step (2), the frequency table is then updated to remain being sorted. So `SN` is moved to the second place on the frequency table now.[1] The lowest two symbols `A` and `T` are combined to `AT` with frequency 2.

[1] Note that `SN` is inserted to the highest possible position in order to get a more balanced binary tree. It is, therefore, not placed between symbols `E` and `I`.

In step (3), `AT` is inserted to the second place of the frequency table and the lowest two symbols `E` and `I` are combined to `EI` with frequency 3, which is inserted to the beginning of the frequency table in step (4).

This process continues until all the symbols are combined to one symbol '(((SN)L)EI)(B(AT))' in step (8).

Now the binary tree (called Huffman tree) is constructed. We only need to assign a 0 to each left edge and 1 to the right edge. Collect the 0s and 1s from the root to a leaf, we will have a binary code (called Huffman code) for that leaf. For example, the Huffman code for symbol `S` is 0000, and 0001 for `N`, and etc.

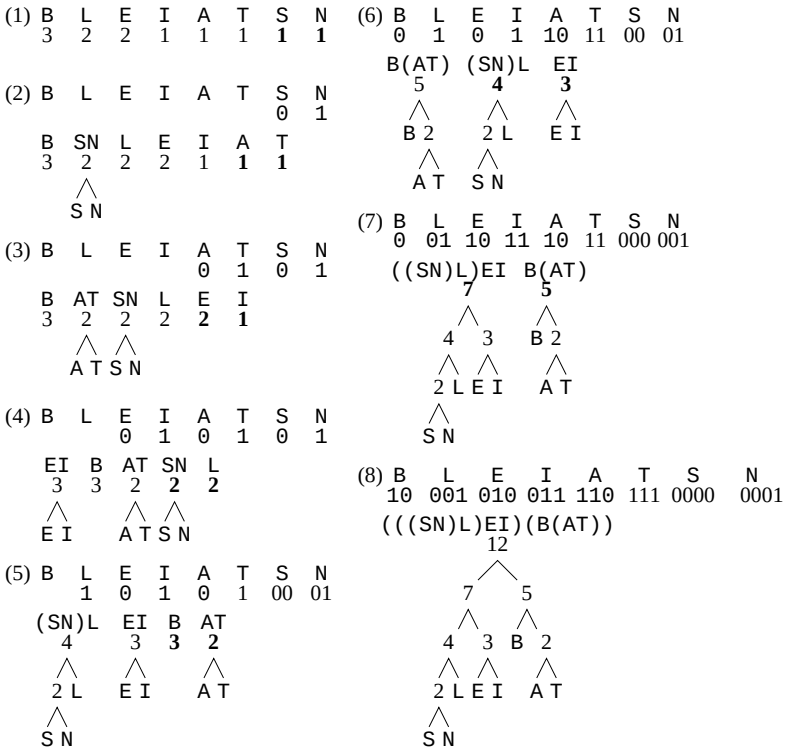


Figure 7.1: Deriving a Huffman code

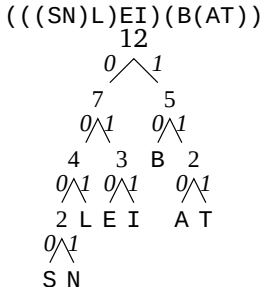


Figure 7.2: A Huffman tree

7.4.2 Huffman decompression algorithm

The decompression algorithm (Algorithm 7.2) involves the operations where the codeword for a symbol is obtained by 'walking' down from the root of the Huffman tree to the leaf for each symbol.

Example 7.7 *Decode the sequence 00000100001 using the Huffman tree in Figure 7.2.*

Figure 7.3 shows the first seven steps of decoding the symbols S and E. The decoder reads the 0s or 1s bit by bit. The 'current' bit is highlighted by shading in the sequence to be decompressed on each step. The edge chosen by the decompression algorithm is marked as a bold line. For example, in step (1), starting from the root of the Huffman tree, we move along the left branch one edge down to the left child since a bit 0 is read. In step (2), we move along the left branch again to the left child since a bit 0 is read, and so on. When

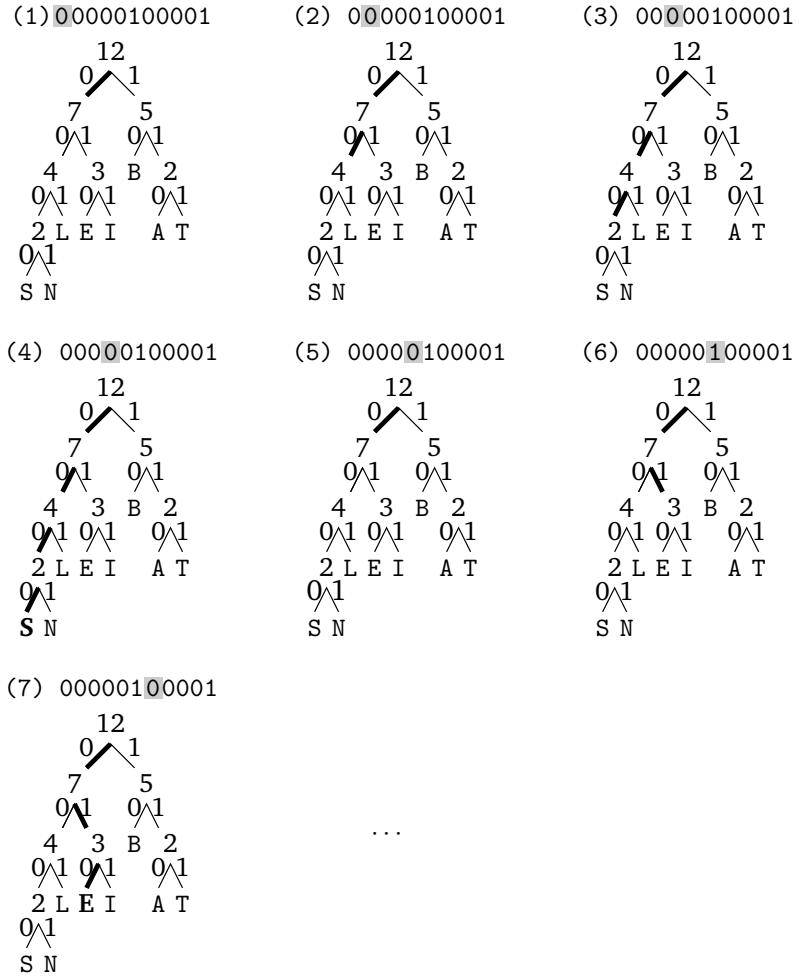


Figure 7.3: Huffman decompression process

we reach a leaf, for example, in step (4), the symbol (the bold 'S') at the leaf is output. This process starts from the root again (5) until step (7) when another leaf is reached and the symbol 'E' is output.

The decoding process ends when `EOF` is reached for the entire string.

Observation

Example 7.6 shows a very small instance of how the Huffman algorithm works. It does not, however, show how effective the compression algorithm is.

Consider the Huffman code generated:

Symbol	B	L	E	I	A	I	S	N
H. code	10	001	010	011	110	111	0000	0001
Length	2	3	3	3	3	3	4	4

There are total of 12 symbols. The probabilities of the symbols can be calculated according to the frequency table. The probability of symbol `B` is 3/12 given its frequency is 3. Similarly, the probabilities of symbols `L` and `E` are both 2/12, and the rest 5 symbols `I, A, T, S, N` has a probability of 1/12 each, given the frequency is 2 for the former and 1 for the later.

Algorithm 7.2 Huffman decoding

INPUT: a Huffman tree and a 0-1 bit string of encoded message
OUTPUT: decoded string

```
1: initialise p ← root
2: while not EOF do
3:    read next bit b
4:    if b = 0 then
5:       p ← left_child(p)
6:    else
7:       p ← right_child(p)
8:    end if
9:    if p is a leaf then
10:      output the symbol at the leaf
11:      p ← root
12:   end if
13: end while
```

The average length of the code is, therefore,

$$\begin{aligned} \bar{l} = \sum_{i=1}^{8} l_i p_i &= 2 \times 3/12 + 2(3 \times 2/12) + 3(3 \times 1/12) + 2(4 \times 1/12) \\ &= 0.5 + 1 + 0.75 + 0.67 \approx 2.92 \text{ bits} \end{aligned}$$

For such a small alphabet, it would be much easier to use 3-bit fixed length code instead of Huffman codes. However, in a real application such as the ASCII code, the standard alphabet would contain 256 symbols.

Activity 7.4

GREEDY APPROACH

1. Describe the main characteristics of optimisation problems.
2. What is the greedy approach? Outline the ideas of the approach.
3. Give an example of an optimisation problem and describe it.
4. Study the greedy approach for the change-making problem and describe it. Describe an instance of the optimisation problem for which the greedy algorithm does not yield an optimal solution.
5. Design and implement an algorithm for the knapsack problem in Section 7.3.2.
6. Implement compression and decompression parts of the Huffman algorithm.

Chapter 8

Limits of computing

8.1 Essential reading

Richard Johnsonbaugh and Marcus Schaefer *Algorithms*. (Pearson Education International, 2004) [ISBN 0-13-122853-6]. Chapter 10, 11

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 11.3

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6]. Chapter 15

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 13

8.2 Learning outcomes

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

- explain the intractability of some hard problems
- describe basic complexity classes such as P and NP
- describe a couple of classic NP-complete problems and the issues of finding efficient algorithms for them.

8.3 Computability and computational complexity theory

We have already seen how different algorithms can solve various problems. Some of them are so called *efficient algorithms* having polynomial running time, but others are inefficient due to exponential running time. Most of the algorithms that we have studied are efficient, for example, the algorithms for searching, sorting, traversal on a linear data structure. Some optimisation problems and graph problems are inefficient, such as the knapsack problem and the travel salesman problem. For some of those problems there is no efficient algorithm known. However, no evidence has yet been found that no efficient algorithm is possible for them either. There are other problems for which no algorithm at all exists!

In this situation, a natural question to ask would be:

- Is there an *efficient algorithm* for a given problem?
- How would we know if an algorithm is the *'best possible'* for the given problem?[1]
- What do we mean exactly by the terms *efficient* and *optimal*?
- What should we do if we *cannot* find an efficient algorithm for the problem despite of a lot of effort?

[1] A best possible algorithm is often said to be *optimal*.

Computability and Computational Complexity Theory are the subject fields in computer science concerned with the above questions and the answers to them. In these areas, the problems to be solved are the main concerns as well as their algorithms. We review the whole gamut of the problems, group them into classes according to their complexity, and look at the relationships between the classes. We want to know which problems are solvable and which are not; which problems are easy to solve and which are hard. We also want to understand why they are easy or hard.

Computability Theory tries, for example, to identify one particular class of problems that have no algorithms. These problems are called *undecidable problems* sometimes.

Example 8.1 *Halting problem: Write a program to detect whether a program has an infinite loop.*

It can be proved that no program can be written to solve the Halting problem. In other words, the Halting problem is *undecidable*. One intuitive reason is that such a program might have a *hard time* checking itself. These problems, therefore, are sometimes called *recursively undecidable*.

One goal of Computational Complexity Theory is to identify *solvable* problems that are intractable in the sense that *no efficient algorithm exists*.

Both areas are based on a theoretical *model of computation* and a formal definition of important concepts such as *problem, algorithm, problem complexity, algorithm efficiency, etc.* The computational model used in these areas is the so-called *Universal Turing Machine* which was defined by Alan Turing in 1936. The problems that we concentrate on are so-called *decision problems* (Section 8.5).

8.4 Computational model

If you have not studied the Turing machine, you only need to know the following in order to follow the rest discussion of the chapter: *Turning machine is a theoretical computation model. It consists of a limitless memory for both data and algorithms, and a moving head that can read from and write to each cell of the memory following the instructions in the algorithm. It can simulate arbitrary algorithms or programs without consequential loss of efficiency.*

Why an abstract computational model, but not a real computer?

There are two main reasons:

1. We need to be able to argue that our results on computability (tractability) apply to *all* types of computers, including the ones not only in the past or at present, but also in the *future*.
2. It can be proved that, for some decision problem X, there is no Turing machine program that computes X *if and only if* there is no Java (or C, C++, etc.) program that solves X.
   For *any* given program in Java, we can (in principle) write a Turing machine program which can simulate the behaviour of the Java program. Similarly, we can write a Java program that can simulate the behaviour of *any* given Turing machine program.

8.5 Decision problems

A decision problem is a computational problem that decides whether the input satisfies a specific property. In other words, a decision problem, for a given input, will always output a solution in terms of *yes* or *no*, i.e. '1' or '0'.

Example 8.2

Input: k, *a number*
Output: *1 if* k *is a positive number; 0 otherwise.*

Why decision problems?

There are mainly two reasons:

1. Since there are usually many ways to define a problem, we need to formalise the problems for any comparison. Decision problems allow us to concentrate on the nature of the problems.
2. It is easy to convert a problem to its version of a decision problem. Many problems can be easily formulated as a decision problem, especially the optimisation problems (Chapter 7).

Example 8.3 *A searching problem can be defined in the following two ways:*

1. *Given a key and a list of elements, search for the key in the list of the elements. Return the index of the key if it is found in the list and return a* `null` *otherwise.*
2. *Given a key and a list of elements, search for the key in the list of the elements. Return a 'yes' if the key is found in the list and return a* `no`*, otherwise.*

The second one is a decision problem, for it requires only a 'yes/no' answer and it is in a simpler form. The difference between the two versions of the searching problem, however, does not affect the nature nor complexity of the searching problem.

Example 8.4

Problem:	*Given a graph G, find the shortest tour that visits every vertex precisely once.*
Decision version:	*Given a graph G and a positive integer k, is there a 'tour' of all vertices of G with total distance at most k?*

The difference between the two versions does not affect the problem complexity. The decision version requires an input k as a constant, but the length of tour for the original problem is not required.

Example 8.5

Problem:	*Given a graph G, determine the smallest number of colours needed to colour the vertices of G.*
Decision version:	*Given a graph G and a positive integer k, is there a colouring of G using at most k colours?*

Again the difference between the two versions does not affect the problem complexity. Similarly, the decision version requires an input k as a constant, but the number of colours for the original problem is not required.

8.6 Measure of problem complexity

Intuitively, the complexity of a problem should be measured by the complexity of the algorithm(s) for the problem. For example, a problem is easy if there is an efficient algorithm taking 'little' time to solve it; and it is hard if there is no efficient algorithm, i.e. being 'too time-consuming' to produce a solution. Unfortunately, there are complications:

1. It is possible that the algorithm for a problem has not yet been found, or there is simply no algorithm for the problem
2. An algorithm behaves differently in various cases, typically, the best, worst and average case
3. An algorithm may be known but much less efficient than the optimal solution.

We introduce the concepts of an *upper* and *lower bound* of a problem to clarify the situation.

upper bound A problem's upper bound is the best algorithmic solution that we have found for the problem.

lower bound A problem's lower bound is the best algorithmic solution that is theoretically possible.

The upper bound of a problem refers to our best knowledge of how to solve the problem, while the lower bound refers to the minimum amount of work necessary in order to solve the problem. The upper bound of a problem depends on the current state of algorithm development, while the lower bound is calculated and mathematically proved by theoreticians (see Section 6.10 for an example, where a decision tree is used to derive the $O(n \log n)$ lower bound for comparison based sorting problems).

8.7 Problem classes

We can now divide problems into two broad classes, namely, *open problems* and *close problems*, according to the gap between the upper and lower bound of the problems.

closed problem A problem is a closed problem if its upper and lower bounds are the same.

open problem A problem is an open problem if its upper and lower bounds are different.

For a closed problem, new algorithms may improve some factors within the Big-O but will not change the Big-O for the problem. For an open problem, we cannot be certain what will happen about the gap between the upper and lower bound in the future. It could be narrowed if the previous upper bound is lowered, when more efficient new algorithms are discovered; or if the previous lower bound is raised, when perhaps a new lower bound is proved to be actually higher than we thought. The gap may, however, also remain the same with no progress in understanding the problem.

Next we can define two classes of problems in terms of their consistent upper and lower bounds in the worst cases:

tractable (polynomial-time) problems Tractable problems (also called the polynomial-time problems), are those problems of which both upper and lower bounds are polynomially bounded[2] in Big-O, for example, in $O(\log N), O(N), O(N \log N)$ and $O(N^k)$, where k is a constant.

[2] A problem is said to be *polynomially bounded* if there is a worst case polynomial algorithm for it.

intractable problems Intractable problems are those problems of which both upper and lower bounds are exponentially bounded in Big-O, for example, in $O(2^N), O(N!)$ and $O(N^N)$.

Polynomial-time problems are those problems that are known and proved to be solvable in polynomial time. Intractable problems are those that are known and proved to be unsolvable in polynomial time.

Note:

1. There are problems between the tractable and intractable, the two clearly defined classes, whose upper and lower bounds are inconsistent, for example, the problems with an exponential upper bound and a polynomial lower bound, or with a unknown upper or lower bound.
2. In order to look at algorithmic efficiency from a broader view, we have emphasised the difference between the problems that have efficient algorithms, i.e. whose execution times are bounded by a polynomial in the problem size and those which are not. We divide problems into classes in terms of their computational complexity.

We now discuss three important and related complexity classes, namely $\mathcal{P}$, $\mathcal{NP}$ and $NP - complete$.

8.8 Class $\mathcal{P}$

This is a class of tractable decision problems. That is, a class of problems that includes those with reasonably efficient algorithms.

In 1965, Jack Edmonds and Alan Cobham gave a definition of a feasible problem: A problem is feasible if it has a solution whose cost is at most polynomial. An algorithm is said to be *polynomial bounded* if its worst-case complexity is bounded by a polynomial function of the input size.

$\mathcal{P}$ is the class of *decision problems* that are polynomially bounded.

Why polynomials

1. Polynomials have nice so-called *closure* properties under composition and addition. If a polynomial time algorithm (feasible algorithm) calls another feasible algorithm as a component, the composed algorithm is again a polynomial algorithm. If two feasible algorithms run one after another, the whole algorithm is again a polynomial algorithm.
2. All sequential digital computers are polynomially related. If a problem is solvable in polynomial time on one conventional computer, it can then be solved in polynomial time on another computer.
3. We can say that if a problem is not in $\mathcal{P}$, it will be extremely expensive and probably impossible to solve in practice. In general, if an algorithm is of exponential (or worse) then it is feasible for *small* inputs only.

Note: the last observation is an empirical one and it does not hold in all cases. An algorithm may take a few minutes in practice in the average case but exponential in the worst case. Most polynomial algorithms that grow faster than an order of cubic are almost useless for a large input size in practice.

8.9 Class $\mathcal{NP}$

$\mathcal{NP}$ is, *informally* speaking, the class of decision problems that can only be solved efficiently by the *guess-and-verify* technique: guess a solution, and verify that the proposed solution is indeed a solution.

That is to say, *loosely*, for every problem in $\mathcal{NP}$, a given proposed solution for a given input can be checked quickly, i.e. it can be checked in polynomial time if the solution satisfies all the requirements of the problem.

Note: NP stands for *non-deterministic polynomial-time* not non-polynomial-time.

1. Composite

   Example 8.6

Input: *n-bit numeral x;*
Output: *1 if x is a composite number; 0 otherwise.*

2. Satisfiability (SAT)

Example 8.7
Input: *A boolean expression F over n variables $x_1, x_2, .., x_n$;*
Output: *1 if the variables of F have an assignment of n values of $x_1, x_2, .., x_n$ which makes F true, 0 otherwise.*

3. Hamiltonian Cycle (HC)

Example 8.8
Input: *Graph $G = (V, E)$ with n vertices.*
Output: *1 if there is a cycle of edges in G which includes each of the n nodes exactly once, 0 otherwise.*

4. CLIQUE

Example 8.9
Input: *Graph $G = (V, E)$ with n vertices, positive integer k.*
Output: *1 if there is a set of k vertices W in V such that every pair of vertices in W is joined by an edge in E, 0 otherwise.*

5. 3-colourability (3-COL)

Example 8.10
Input: *Graph $G = (V, E)$ with n vertices.*
Output: *1 if each vertex of G can be assigned one of three colours in such a way that no two nodes joined by an edge are same colour, 0 otherwise.*

All the above decision problems are 'checkable' by efficient (polynomial time) algorithms, if a solution is proposed.

For any decision problem f, we can define another decision problem CHECK(f), for example, as follows:

CHECK(f)
Input: Input instance I for f, possible witness W that $f(I) = 1$.
Output: 1 if W is a genuine witness that $f(I) = 1$, 0 otherwise.

Now

- CHECK($Composite$)
- CHECK(SAT)
- CHECK(HC)
- CHECK(CLIQUE)
- CHECK(3-COL)

are all *polynomial time decidable* due to the polynomially checkable nature of CHECK(f) and f.

Given a guessed solution f, we could define
$\mathcal{NP} = \{f$: CHECK(f) is in P$\}$.

Note: not all decidable problems are in $\mathcal{NP}$.

8.10 $\mathcal{P}$ and $\mathcal{NP}$

$\mathcal{P}$ is the class of problems that can be 'solved' efficiently. $\mathcal{NP}$ is the class of problems, once having been solved (possibly by guessing), whose solutions can be 'checked' efficiently. So $\mathcal{P}$ is a subset of $\mathcal{NP}$.

An interesting question (no answer yet) can be asked: Is $\mathcal{P}$ a proper subset of $\mathcal{NP}$? In other words, are there any decision problems f such that *checking* a solution is much easier (in polynomial time) than *finding* a solution?

The following problems are a few of the tens of thousands of known NP problems for which no efficient algorithms have been found (despite, in some cases, centuries of work on them):

- Hamiltonian cycle
- Satisfiability
- 3-Colouring
- Clique
- Travelling salesman problem
- Integer knapsack problem
- Longest path in a graph
- Edge colouring
- Optimal scheduling
- Minimising boolean expressions
- Minimum graph partition
- Composite number
- Primality

It is generally believed that no efficient algorithm exists for any of these decision problems except the last two. That is to say, they are believed to be intractable.

8.11 NP-complete problems

A subset of the problems in $\mathcal{NP}$ contains the *hardest*, which are known as *NP-complete problems*. The term 'complete' is used here to mean that a solution to any problem in the set can be applied to all others in the set.

Given a class of problems $\mathcal{C}$, a problem P is $\mathcal{C}$-hard if it is *at least* as hard as every problem in $\mathcal{C}$. If the problem P is also in $\mathcal{C}$, then P is called $\mathcal{C}$-complete.

8.11.1 What do we mean by $hard$?

From our experience, a problem with a *polynomial* (time) solution is 'easy'; a problem whose every solution grows faster than a polynomial is 'hard'. Let us look at the following situation.

A problem P is said to be *polynomially transformable* (reducible) to problem P_{hd} if we can transform *any* instance of P into some instance of P_{hd} in polynomial time. In principle, we could solve any instance of P by transforming the instance, in polynomial time, into an instance of P_{hd}, then solve P_{hd}.

Figure 8.1 shows how a problem P is solved by a polynomial transformation of P to P_{hd}. Here A is the algorithm for problem P and A_{hd} is an algorithm for problem P_{hd}. A_T is a polynomial time

algorithm which transforms every instance of P to that of P_{hd}. The solution for P_{hd} (i.e. the output of algorithm A_{hd}) must be the same as the solution for problem P (i.e. the output of algorithm A) regarding every input x.

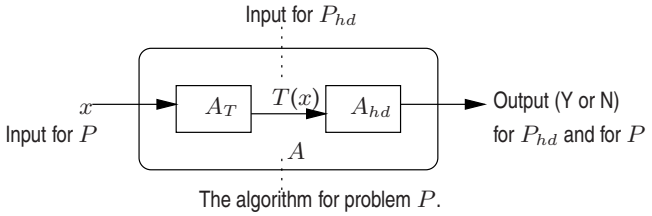


Figure 8.1: Transformation by a polynomial T

If P_{hd} is easy then P is also easy. If P_{hd} is hard, can we say that P is also hard? We cannot because there may be some other way to solve P. What we can say is that P_{hd} is *at least* as hard as P, or that P is no harder than P_{hd}.

Now we can define *NP-complete problems*. The NP-complete problems are the most difficult problems in $\mathcal{NP}$ in the sense that *if any single NP-complete problem has an efficient algorithm then all the problems in $\mathcal{NP}$ have efficient algorithms*. In other words, if we find an efficient algorithm for just one NP-complete problem, then we can construct efficient algorithms for all decision problems in $\mathcal{NP}$.

For *NP-complete* problems:

- no exact complexity solutions have yet been determined,
- either all problems in the class have polynomial-time solutions or none of them do.

8.11.2 How to prove a new problem is NP-complete

The common practice is to

1. Prove the new problem (P_{hd}) is in $\mathcal{NP}$.
2. Show a known NP-complete problem (P) can be transformed into the new problem in polynomial time.

This means that once we have identified a *single* NP-complete problem, f, in order to prove that another decision problem g is NP-complete, all we have to do is to find an efficient algorithm for transforming input instances for f into input instances for g.

8.11.3 The first NP-complete problem

The first NP-complete problem is known as *Satisfiability* or SAT problem. In 1971 Steven Cook proved that SAT is NP-complete by showing that all problems in $\mathcal{NP}$ could be transformed to satisfiability. He used the fact that every problem in $\mathcal{NP}$ can be solved in polynomial time by a non-deterministic Turing machine.

8.11.4 More NP-complete problems

Within a few months of Cook's result, dozens of new NP-complete problems had been identified. There is now a long list of problems known as being NP-complete.

Here are few examples:

- Bin packing
- Knapsack
- Graph colouring
- Clique
- Hamiltonian cycle
- Travelling salesman problem
- Set partitioning
- Longest path

A proof that a decision problem is NP-complete is accepted as evidence that the problem is *intractable*. However, it has still to be proved that any NP-complete problem does not have an efficient algorithm.

Most of the classical decision problems have now been determined as being in $\mathcal{P}$ or being NP-complete, but a number of important problems are still open. In particular, the problems called *Primality* and *Composite number* are both known to be in $\mathcal{NP}$, but neither has been shown to be NP-complete. Some people believe that both are solvable by efficient algorithms.

Activity 8.11

COMPLEXITY THEORY

1. Explain, with an example, what the upper bound and lower bound of a problem are.
2. What is class $\mathcal{P}$? What is class $\mathcal{NP}$?
3. What is a NP-complete problem?
4. What is important about polynomial complexity in classifying problems?
5. Describe a decision problem with an example.
6. Suppose algorithms A_1 and A_2 have worst-case time complexity bounded by t_1 and t_2 respectively. If algorithm A_3 consists of applying A_2 to the output of A_1, i.e. the input for A_3 is the input of A_1, what is the worst-case time bound for A_3?

Chapter 9

Text string matching

9.1 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 12

Richard Johnsonbaugh and Marcus Schaefer *Algorithms*. (Pearson Education International, 2004) [ISBN 0-13-122853-6]. Chapter 9

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6]. Chapter 10, 23.2

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 11

9.2 Learning outcomes

In this chapter, we study three popular string matching algorithms and an efficient data structure for strings. Having read this chapter and consulted the relevant material you should be able to:

- explain basic concepts in string matching
- describe a trie data structure
- demonstrate how the fundamental string maching algorithms, such as Naive, Boyer-Moore and KMP algorithms, work with examples.

9.3 String matching

A sequence of symbols is called a *text string* or *string* for short. Algorithms for processing strings are often called *string algorithms*. String algorithms have a long history due to ubiquitous applications from document processing, through searching on the internet, to DNA computation[1].

[1] see http://en.wikipedia.org/wik/Dna_computing

The number of symbols in the string is called the *length* of the string. For example, `abcaabbcab` is a string of length 10.

One fundamental type of string algorithms is for text searching. The text searching problem has a general form: Given two strings of different lengths usually, the goal is to search for a match of the shorter string within the longer string. For example, we want to search within a text document called *dictionary*, for a specific string of interest called *word*. This problem can be described more precisely as below:

Consider two strings, one is called a *text* and the other *pattern*. The

text is often longer than the pattern and the pattern is no longer than the text. The goal of the algorithm is to find a matched string called *pattern* P of length m within a text T of length n and the position of P in T.

Example 9.1 *A string T and pattern P:*

```
T        a b c a a b b c a b
P              a a b b
```

$n = 10$ *and* $m = 4$ $s = 3$*, where* s *is called a valid shift*

Algorithms that deal with strings and patterns, for example, searching for a pattern in a text string, are called *string matching algorithms*.

String matching algorithms have many applications. A popular form is to find all occurrences of some pattern in a text, for example, searching keywords in a text file, searching records from databases and operations in search engines such as Google.

String matching can be the bottlenecks in document processing in general, for it can be time consuming to identify the matched patterns and complete the required string search. In this chapter, we study three popular string matching algorithms, namely:

1. Naive approach
2. Boyer-Moore algorithm
3. KMP matching algorithm.

9.4 The string ADT

In order to concentrate on the application problem, strings can be defined as an ADT which not only consists of data structures but also the routine operations on the strings. For example, the following can be used to describe the string ADT.

1. String as an array of characters
2. *Trie* data structure (Section 9.7)
3. String pattern matching
4. Text compression.

Terminology

To ease the discussion, we need to define first the following commonly used terms.

string a sequence of characters: $S = s_1 s_2 \cdots s_n$, where n is the number of characters in the string, which is referred to as the *length* of the string.

substring a number of consecutive characters of a string: A substring of string $S = s_1 s_2 \cdots s_n$, is a string of any length m such that $P = p_1 p_2 \cdots p_m = s_{1+i} s_{2+i} \cdots s_{m+i}$, where $m \leq n$, and $0 \leq i \leq (n - m)$

proper substring a substring that is shorter than the original string: a number of consecutive characters of a string: A substring of string $S = s_1 s_2 \cdots s_n$, is a string $P = p_1 p_2 \cdots p_m$, where $m < n$.[2]

[2]The definition is same as *substring* except $m < n$ instead of $m \leq n$.

`null` **string** a string that contains no character, which is called sometimes an *empty string*.

prefix a substring of any length m such that $P = p_1 p_2 \cdots p_m = s_1 s_2 \cdots s_m$, where $m \leq n$ ($m < n$ for *proper prefix*).

suffix a substring of any length m such that $P = p_1 p_2 \cdots p_m = s_{n-m+1} s_{n-m+2} \cdots s_n$, where $m \leq n$ ($m < n$ for *proper suffix*)

immutable strings a string that once defined cannot be modified, for example, `String` class in Java.

mutable strings – a string that can be modified, for example, `StringBuffer` class in Java.

Example 9.2 *Given a string* `S="gihhdkkkkdhhhhhk"`*, demonstrate an example of a substring, proper substring, prefix and* `suffix`*:*

```
original string:               gihhdkkkkdhhhhhk
a substring of length 5:       hdkkk
the longest substring:         gihhdkkkkdhhhhhk
the longest proper substring: gihhdkkkkdhhhhh
                          or: ihhdkkkkdhhhhhk
a prefix of length 5:          gihhd
the longest proper prefix:     gihhdkkkkdhhhhh
a suffix of length 5:                     hhhhk
the longest proper suffix:     ihhdkkkkdhhhhhk
```

Java String class

These are routine operations that can be called upon:

int length() returns the length of `this` string[3]

[3]See **this** keyword in Java.

int indexOf(char ch) returns the index within `this` string of the first occurrence of the specified substring

char charAt(int i) returns the character at index i

boolean equals(Object o) compare `this` string to object o

boolean startsWith(String prefix) check if `this` string starts with string prefix

boolean endsWith(String suffix) check if `this` string ends with string suffix

String substring(int beginIndex) returns the substring of `this` string beginning with the character at beginIndex

String substring(int beginIndex, int endIndex) returns the substring of `this` string beginning with the character at beginIndex

String concat(String s) concatenates the specified string to the end of `this` string.

Example 9.3 *Write the output on completion of the following operations on the string* `S="gihhdkkkkdhhhhhk"`*, where* `S[0]='g'`*.*

```
length()                        16
indexOf("dkkkk")                5
charAt(5)                       k
equals("gihhdkkkkdhhhhhk")      true
startsWith("gihh")              true
endsWith("algorithm")           false
substring(5,10)                 kkkkdh
concat("yyy")                   gihhdkkkkdhhhhhkyyy
```

Java StringBuffer class

These are Java versions of some string operations:

StringBuffer append(S, Q) Replacing S with S+Q
INPUT: string S and Q
OUTPUT: stringBuffer S+Q

StringBuffer insert(i, Q) Inserting Q inside S starting at index i
INPUT: index i and string Q
OUTPUT: StringBuffer

StringBuffer reverse() Replace the character sequence contained in `this` string buffer by the reverse of the sequence

void setCharAt(i, ch) Set the character at index i of `this` string buffer to ch

char charAt(i) returns the character at index i of the currently represented string buffer.

9.5 String matching

For convenience of discussion, let us review our definitions:

1. text string `T`, $T.length = n$
2. pattern string `P`, $P.length = m$
3. match: a substring of T starting at some index i that matches P, character by character, so that $T[i] = P[1], T[i+1] = P[2], \ldots, T[i+m] = P[m]$ [4].
4. mismatch: $T[i] \neq P[1]$ or a partial match from $T[i] = P[1], \ldots, T[i+k] = P[k]$, where $k < m$ (m is the length of the pattern).

[4] or $T[i] = P[0], T[i+1] = P[1], \ldots, T[i+m-1] = P[m-1]$ if the index of pattern starts from 0 to $m-1$.

9.5.1 Naive string matching

This is a straightforward method. In each iteration, each symbol in pattern P is compared with each corresponding symbol in text T, one by one from the beginning of the text and from the left to the right. If there is a match, the algorithm returns the first match position in the text. Otherwise, the pattern is shifted 1 position to the right. The first corresponding symbol for comparison in T is shifted 1 position to the right, too. The next round of comparison process starts over again.

We use two index variables `i` and `j` to identify the 'current' two characters for comparison, where `i` for the character position in the text and `j` for that in the pattern. The corresponding symbols for

comparison in the text and the pattern can be specified using the index values `i` and `j`. For instance, in step (6) of Example 9.4, `i` = 6 and `j` = 1 point to the first pair of comparison symbols T[6] and P[1] respectively. While they are identical (marked by a '!'), let $i = i + 1; j = j + 1$. The next two symbols for comparison are T[i] and P[j] for the updated i and j, i.e. T[7] and P[2]. This process continuous until T[i] and P[j] are mismatched. When they are different (marked by a 'x'), let $i = i + 1$ and $j = 1$, i.e. P is shifted one position (marked by a '.') to the right. A new starts over again. As we can see, there is a shift by merely one position for each step for the naive algorithm.

Example 9.4 *Initially in step (1), $i = 1$ and $j = 1$ point to the first pair of comparison symbols P[1]='k' and T[1]='g', which are different. So let $j = 1$ and $i = i + 1$, i.e. P is shifted one position to the right in step (2). Next, P[1] and T[2] are compared and found different. So $j = 1$ and $i = i + 1$, i.e. P is shifted one position to the right in step (3). A similar process occurs in step (4) and (5). In step (6), since P[1] and T[6] are identical, j and i are both increased by 1. P[2] and T[7] are compared and found identical. So j and i are both increased by 1, again. Next, since P[3] and T[8] are different, $j = 1$ and $i = i + 1$, P is shifted one position to the right in step (7). The process continues in this fashion in step (7) and (8). In this example, the algorithm terminates when a complete match of P is found in step (8).*

```
(1)            (m)
      j   1 2 3 4
    P[j]  k k d h
    T[i]  g g h h d k k k k d h h h h h k
      i   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
          x                           (n)

(2)  P[j]  . k k d h
     T[i]  g g h h d k k k k d h h h h h k
             x

(3)  P[j]    . k k d h
     T[i]  g g h h d k k k k d h h h h h k
               x

(4)  P[j]      . k k d h
     T[i]  g g h h d k k k k d h h h h h k
                 x

(5)  P[j]        . k k d h
     T[i]  g g h h d k k k k d h h h h h k
                   x

(6)  P[j]          . k k d h
     T[i]  g g h h d k k k k d h h h h h k
                     ! ! x

(7)  P[j]            . k k d h
     T[i]  g g h h d k k k k d h h h h h k
                       ! ! x

(8)  P[j]              . k k d h
     T[i]  g g h h d k k k k d h h h h h k
                         ! ! ! !
```

This is a $O(mn)$ time algorithm in the worst case[5], where n is the length of the text, and m is the length of pattern, as we can see from the algorithm below:

[5] This is when the pattern is not in the text and all the symbols in the pattern have to be compared.

Algorithm 9.1 NaiveMatch(T, P)

INPUT: Strings T (text) and P (pattern)
RETURN: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

```
1: for i = 1, i < n − m, i = i + 1 do
2:    j ← 0
3:    while j < m and T[i + j] = P[j] do
4:       j ← j + 1
5:    end while
6:    if j = m then
7:       return i
8:    end if
9: end for
10: return 'no substring of T matches P'
```

The main steps in the worst case are in $(n - m + 1)(1 + m) + 1 = O(mn)$, where n is the length of the text, and m is the length of pattern.

9.5.2 Observation

If we consider steps 2,6 and 7 in the previous example, we realise that the NaiveMatch algorithm has:

1. 'Looking-Glass' heuristics: some comparisons for partial matching can be avoided by comparing characters in P backwards.
2. 'Character-Jump' Heuristic: this can be avoided by shifting more than one character whenever possible in the following sense: Consider a mismatch of text character $T[i] = c$ with the corresponding P[j]. If *c* is not in P, then jump P completely past T[i], else shift P until an occurrence of *c* gets aligned with T[i].

9.5.3 Boyer-Moore Algorithm

The Boyer-Moore algorithm[6] addresses the two heuristics discussed in Section 9.5.2.

[6] 'BM algorithm' for short.

First, in each internal iteration, the comparison between each pair of the symbols starts from the right to left.

Example 9.5 *Consider one step (below) in the BM algorithm. The numbers 1--5 on the left are line reference numbers for our discussion. The comparison takes place between the text segment* T[2..5] *and pattern* P[1..4]. *The BM algorithm starts the comparison from the* right to left*: first comparing* T[5] *and* P[4]*, and, since they are the same 'd', then* T[4] *and* P[3]*, and since they are the same 'h',* T[3] *and* P[2] *are compared. The process continues until* $T[3] \neq P[2]$*, where a mismatch is found. We mark the matched symbols by* '!' *and mismatched symbol by* 'x' *in line 5 in the example below:*

```
1        i   1 2 3 4 5 ...
2      T[i]  g g h h d k k k k d h h h h h k
3      P[j]  . g k h d
4        j     1 2 3 4
5              x ! !
```

Secondly, the BM algorithm shifts the pattern from the left to right in each outer iteration. However, the shift may skip certain positions. The shift distance depends on whether the mismatched symbol in the text appears in the not-yet-compared part of the pattern and, if yes, on its position in the pattern.

Example 9.6 *Consider the example below. The mismatched symbol 'd'* [7] *in T* [8] *is found in the second position (from the left) in P* [9]*. Therefore, the pattern is shifted to the right in such a way that the 'd' in P is line-up with the 'd' in T (see line 4 'after shift').*

[7] marked by 'x'
[8] marked by '!'
[9] marked by '!'

```
                 !
1      T[i]  g g h h d k k k k d h h h h h k
2      P[j]  . g d k k                      (before shift)
3              !   x
4      P[j]    . . g d k k                  (after shift)
5                  !
```

If no mismatched symbol in T appears in the not-yet-compared part of P, the pattern can be safely shifted to the past-mismatched postion, i.e. the immediate position next to that of the last pattern symbol.

Example 9.7 *Consider the example below. The mismatched symbol 'h' in T is not in the not-yet-compared part of P, i.e. not in (g). Therefore, the pattern P can be shifted safely to line-up with the first location after the mismatched position. As we can see, the pattern jumps from the location (see line 2) to the location (line 4) below, and shifted by 2 index positions.*

```
1      T[i]  g g h h d k k k k d h h h h h k
2      P[j]  . g k h d                    (before shift)
3              x ! !
4      P[j]    . g k h d                  (after shift)
```

Example 9.8 *Consider another example below. The mismatched the symbol 'd' in T is not in the not-yet-compared part of P, i.e. not in (gkkk). Therefore, the pattern P can be shifted safely to line-up the first location after the mismatched position (marked by 'x' in line 3). As we can see in line 4, the pattern jumps from the location (see line 2) to the location (line 4) below, shifted by 4 positions.*

```
1      T[i]  g g h h d k k k k d h h h h h k
2      P[j]  . g k k k                      (before shift)
3                x
4      P[j]    . . . . g k k k              (after shift)
```

BM algorithm is based on the rules discussed above. The pattern is shifted from the left to the right as before in the outer iteration, but the symbol pairs are compared from the right to the left in the internal iteration.

Example 9.9 *Consider the question in Example 9.4 again. The steps now become:*[10]

[10] we mark the previous step numbers in a double quote for comparison.

```
(1)                          1 1 1 1 1 1 1
        i   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
      T[i]  g g h h d k k k k d h h h h h k
                (m)                   (n)
        j   1 2 3 4
      P[j]  k k d h

(2)   T[i]  g g h h d k k k k d h h h h h k
      P[j]  . . . k k d h                      :"(4)"

(3)   T[i]  g g h h d k k k k d h h h h h k
      P[j]        . . k k d h                  :"(6)"

(4)   T[i]  g g h h d k k k k d h h h h h k    :"(8)"
      P[j]            . . k k d h
```

In step (1), the comparison process terminates when $T[3] \neq P[3]$. *Since '*h*' in T does not appear in the not-yet-compared part* kk *in P, the pattern is shifted to position* $i = 4$*, the next position to the mismatched postion* $i = 3$.

In step (2), a mismatch is found in position $i = 7$. *Since '*k*' in T appears in position* $j = 2$ *(line-up at* $i = 5$*), the pattern is shifted so that* $P[2] = 'k'$ *lines-up with* $T[7] = 'k'$.

In step (3), a mismatch is found in position $i = 9$. *Since '*k*' in T appears in position* $j = 2$ *(line-up at* $i = 7$*), the pattern is shifted so that* $P[2] = 'k'$ *lines-up with* $T[9] = 'k'$.

In step (4), a match is found after 4 comparisons: $P[4] = T[11] = 'h'$, $P[3] = T[10] = 'd', P[2] = T[9] = 'k'$, *and* $P[1] = T[8] = 'k'$. *The BM algorithm returns the matching position (e.g.* $i = 8$*) and terminates.*

Comparing the steps in Example 9.9 to those in Example 9.4, we find that the steps "(2)", "(3)", "(5)" and "(7)" in Example 9.4 are saved.

The BM algorithm was invented by researchers Boyer and Moore (Algorithms 9.2 and 9.3):

Algorithm 9.2 function last(c)

```
1: if c is in P then
2:     last(c)← the index of the last (right-most) occurrence of c in P
3: else
4:     last(c)← −1
5: end if
```

9.5.4 Observation

1. Main *inefficiency*: Despite many comparisons before a conclusion of nomatch, we did not use any information gained by the comparisons.
2. The KMP algorithm in the next section improves on this.

Algorithm 9.3 BMMatch(T,P)

INPUT: Strings T (text) and P (pattern)
RETURN: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

```
1: compute function last
2: i ← m − 1, j ← m − 1
3: repeat
4: if P[j] = T[i] then
5:    if j = 0 then
6:       return i
7:    else
8:       i ← i − 1, j ← j − 1
9:    end if
10: else
11:    i = i + m − min(j, 1 + last(T[i]))
12:    j = m − 1
13: end if
14: until i > n − 1
15: return 'no substring of T matches P'
```

9.6 KMP Algorithm

The algorithm was derived by researchers Knuth, Morris and Pratt. It is a linear time string matching algorithm, using an auxiliary function *failure function*[11] that can be used to indicate the 'legal' shift of pattern P and to reuse the comparisons made previously.

[11] also called 'prefix function'.

Let i be the current index of T and j the current index of P, where the first mismatch occurs, i.e. $T[i] \neq P[j]$. It is possible that a prefix of P appears again in P as the suffix of the first $j-1$ characters. In other words, it is possible that P[1..k]=P[j-k..j-1] for some $1 \leq k \leq j-1$.

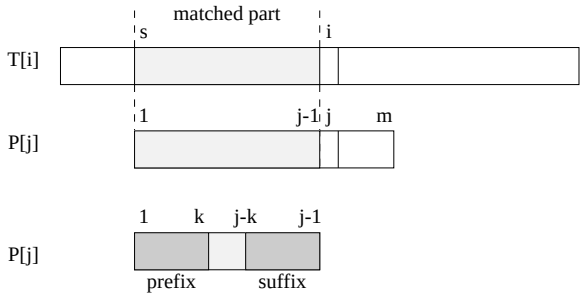


Figure 9.1:

For example, a mismatch occurs at $i = 8$ in text T and $j = 6$ in pattern P below, where $T[8] \neq P[6]$ (underlined), while the first $j - 1 = 5$ characters match those in $T[3..7]$ correspondingly.

```
   i  1 2 3 4 5 6 7 8
T[i]  b a a b a c a a b a c c a b a c a b a a b b
P[j]      a b a c a b a
   j      1 2 3 4 5 6
```

Consider the matched part of the pattern P[1..5]: abaca. Here 'a' in abacaba is a prefix of P that is also a suffix of abaca, the first $j-1$ characters. 'The length of the longest prefix of P that is a suffix of the first $j-1$ characters' in this example is therefore 1, i.e. $P[1..k] = P[j-k..j-1]$, where $k = 1$.

In this case, we can shift the pattern P to line up P[1]='a' with T[7]='a', that is to line up P[k] with T[i-1], where $k = 1$ and $i = 8$.

```
   i  1 2 3 4 5 6 7 8
T[i]  b a a b a c a a b a c c a b a c a b a a b b
P[j]      . . . . a b a c a b a
   j              1 2
```

Note, in the next step, there is no need to compare the first pattern symbol P[1] with T[7] because we know that, from the previous step, P[5]=T[7]=P[1]. The next comparison can start from $i = 8$ in T and $j = 2$ in P. The longer the matched prefix, the more comparisons may be saved.

Example 9.10 *The mismatched position in step (1) is $j = 8$ where $P[8] \neq T[10]$. The length of the longest prefix of P that is also the suffix of the first $j-1$ matched symbols is 3, where j is the first mismatch position, i.e. P[1..3]=P[5..7]='aca'.*

The pattern is shifted to the right to line-up $P[1..3]$ = 'aca' with $T[7..9]$ = 'aca' in step (2). Note we only need to start comparison from $j = 4$ (or $i = 10$) after the shift, because we know $P[1..3] = P[5..7] = T[7..9]$. Three comparisons are saved.

```
(1)     i  1 2 3 4 5 6 7 8 9 10 ···
     T[i]  b a a c a b a c a b c b a b ···
     P[j]      a c a b a c a c c a
        j      1 2 3 4 5 6 7 8 ···

(2)  T[i]  b a a c a b a c a b c b a b ···
     P[j]      . . . . a c a b a c a c c a
        j            1 2 3 4 5 ···
```

The failure function F(j) is defined as the length of the *longest* prefix of P that is a suffix of P[1..j], where $j = 1..m$, and $F(j) = 0$ if there is no such prefix. In particular, $F(1) = 0$. The importance of F(j) is that it encodes repeated substrings inside P itself, where $j = 1..m$..

For example, $F[1] = 0$, because there is no such a prefix:

```
P[j]  a c a b a c a c c a
   j  1 2 3 4 5 6 7 8 9 10
F[j]  0
```

Similarly, $F[2] = 0$, because there is no such a prefix:

```
P[j]  a c a b a c a c c a
   j  1 2 3 4 5 6 7 8 9 10
F[j]  0 0
```

However, $F[7] = 3$, because $P[1..3] = P[5..7]$, i.e. $k = 3$:

```
P[j]  a c a b a c a c c a
   j  1 2 3 4 5 6 7 8 9 10
F[j]  0 0        3
```

In this way, we have the following F[j] for the P[1..j]:

```
P[j]   a c a b a c a c c a
   j   1 2 3 4 5 6 7 8 9 10
F[j]   0 0 1 0 1 2 3 2 0 1
```

We now look at the string matching problem again and see how the F[j] can be used to find a pattern.

We use again i and j to point to the current character in T and P respectively. There can only be two cases: either found a pattern or not. This is when $i - s + 1 = m$ or $i - s + 1 < m$ respectively. Figure 9.2 shows when $j = m$, $i - s + 1 = m$, and so $P[1..m] = T[s..i]$, a match is found, where $j - 1 + 1 = i - s + 1$, i.e. $s = i - j$.

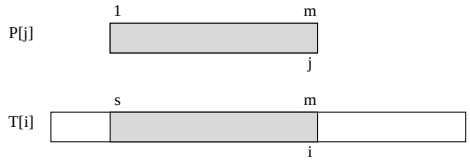


Figure 9.2: A pattern is found

When $i - s + 1 < m$, there are three cases (1–3) as shown in Figure 9.3(a), Figure 9.3(b) and Figure 9.3(c) respectively.

1. $P[1] \neq T[i]$, where $i = s$: $i = i + 1$
2. $P[1..j-1] = T[s..j-1]$, but $P[j] \neq T[i]$: shift the pattern P by $s = s - F[j]$
3. $P[1..j-1] = T[s..j-1]$, and $P[j] = T[i]$: $i = i + 1$, $j = j + 1$.

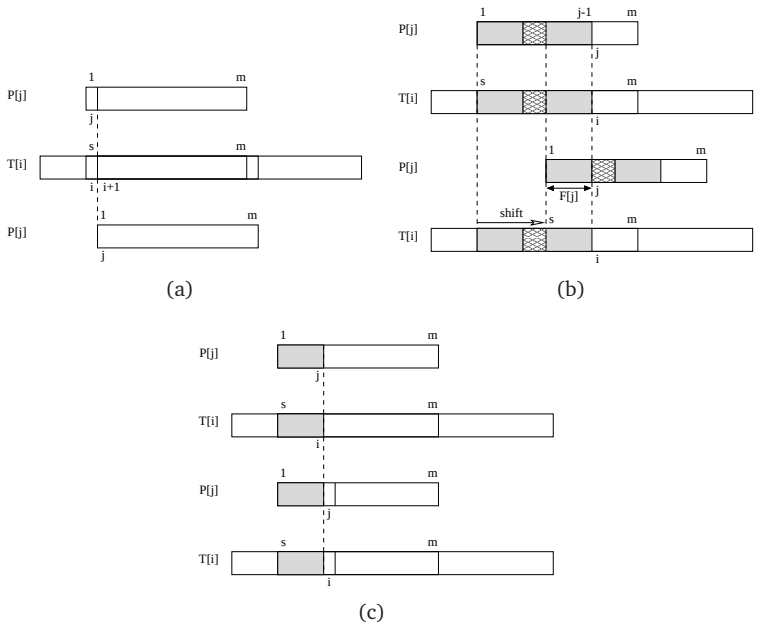


Figure 9.3: A pattern is not yet found

KMP string matching algorithm can be derived based on the ideas discussed so far.

KMP matching algorithm

Algorithm 9.4 KMPMatch(T,P)

INPUT: Strings T (text) and P (pattern)
RETURN: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

```
1: F ← KMPFailureFunction(P)
2: i ← 0, j ← 0
3: while i < n do
4:    if P[j] = T[i] then
5:       if j = m − 1 then
6:          return i − m + 1
7:       end if
8:       i ← i + 1, j ← j + 1
9:    else if j > 0 then
10:      j = F(j − 1)
11:   else
12:      i = i + 1
13:   end if
14: end while
15: return 'there is no substring of T matching P.'
```

Algorithm 9.5 KMPFailureFunction(P)

INPUT: String P with m characters
RETURN: The failure function F for P, which maps j to the length of the longest prefix of P that is a suffix of P[1..j].

```
1: i ← 1, j ← 0, F(0) = 0
2: while i < m do
3:    if P[j] = P[i] then
4:       F(i) ← j + 1, i ← i + 1, j ← j + 1
5:    else if j > 0 then
6:       j=F(j-1)
7:    else
8:       F(i) ← 0, i ← i + 1
9:    end if
10: end while
```

Example 9.11

```
1                       1 1 1 1 1 1 1 1 1 1 2
    i   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
  T[i]  a b a c a a b a c c a b a c a b a a b b
              (m)                       (n)
    j   1 2 3 4 5 7
  P[j]  a b a c a b
        ! ! ! ! ! x

2 T[i]  a b a c a a b a c c a b a c a b a a b b
  P[j]  . . . . a b a c a b
                x

3 T[i]  a b a c a a b a c c a b a c a b a a b b
  P[j]          . a b a c a b
                ! ! ! ! x

4 T[i]  a b a c a a b a c c a b a c a b a a b b
  P[j]            . . . . a b a c a b
                          x
```

```
5  T[i]  a b a c a a b a c c a b a c a b a a b b
   P[j]                  . a b a c a b
                           ! ! ! ! ! !
'!' or 'x' represent a comparison,
where '!' represents a match, and 'x' a mismatch.
```

Implementation

The KMP algorithms can be easily implemented in Java as follows:

```
int kmp(char T[], char P[], int F[]) {
   failureFunction(P,F);
   int n=T.length;
   int m=P.length;
   int i=0,j=0;
   System.out.println("i \t j \t T[i] \t P[j]");
   while (i<n) {
   System.out.println(i+"\t "+j+"\t "+T[i]+"\t "+P[j]+"\t ");
      if (P[j]==T[i]) {
         if (j==(m-1)) {return (i-m+1);}
         i++; j++;
      }
      else {if (j>0) {j=F[j-1];}
           else {i++;}
           }
   }
   System.out.println();
   return 0;
}

   void failureFunction(char P[], int F[]) {
   int m=P.length;
   int i=1; int j=0;
   F[0] = 0;
   while (i<m) {
      System.out.println(i+j+F[i]);
      if (P[j]==P[i]) {F[i]=j+1; i++; j++;}
      else if (j>0) {j=F[j-1];}
           else  {F[i]=0; i++;
           }
   }
}
```

Note: There are different approaches for implementation of the KMP algorithms. Here we refer mainly to the approach in Goodrich and Tamassia's text.

Activity 9.6

STRING MATCHINGS

1. Trace by hand the operations of the following algorithms step by step and calculate the number of comparisons made for the case where P='ABAA' for T='AAABBABABBAABABABAAABB'.

   (a) Naive algorithm

(b) the BM algorithm

(c) KMP

2. Implement the three exact string matching algorithms, i.e. the Naive, the BM and the KMP algorithms.
3. Convert your implementations in question 2 so your programs can demonstrate the shifts and comparisons made on each step.

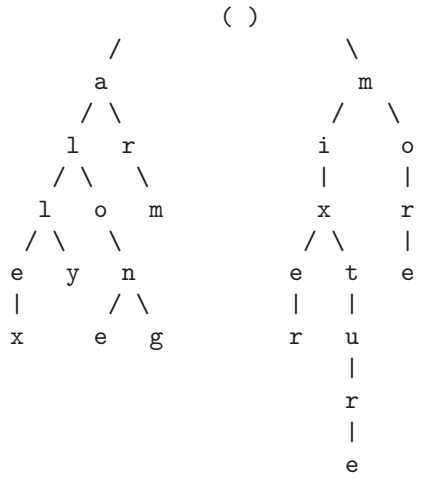
9.7 Tries

A trie (pronounced 'try')[12] is an ordered tree data structure for strings. Its name comes from a four-character substring of word 're*trie*val'. The main application is in information retrieval. The primary query operations that tries support are pattern matching and prefix matching.

[12] Also called *prefix tree*.

We first look at an example:

Example 9.12 *A trie for strings (allex, ally, alone, along, arm, mixer, mixture, more) from an alphabet (a,l,e,x,y,o,n,g,r,m,i,x,t,u).*



Example 9.13 `(all,ally,alone,along,arm,mix,mixer,mixture,more)`

First convert it to

`(all%,ally,alone,along,arm,mix%,mixer,mixture,more)`

by adding a special character (not in the alphabet) so that no word is a prefix of another in S.

9.7.1 Standard tries

A trie for a set S is an ordered tree with the following properties:

1. Each node except the root, is labelled with a character from an alphabet.
2. The ordering of the children follows a standard ordering of the alphabet.
3. Each concatenation of the labels along a path from the root to a leaf corresponds to a word in the set S.

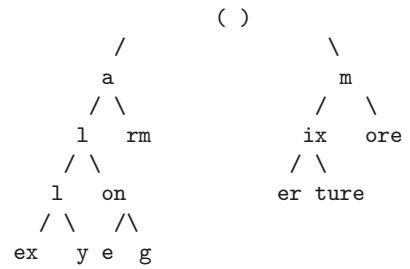
Standard tries have the following properties:

1. Every internal node of T has at most d children, where d is the size of the alphabet.
2. The height of T is equal to the length of the longest string in S
3. The number of nodes of T is $O(n)$.

9.7.2 Compressed tries

1. This is similar to standard tries.
2. Let T be a standard trie, an internal node v of T is *redundant* if v has one child.
3. A chain of $k \geq 2$ edges, $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$ is *redundant* if:
   - v_i is redundant for $i = 1, .., k-1$
   - v_0 and v_k are not redundant.
4. A compressed trie can be derived by replacing each redundant chain $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$ into a single edge (v_0, v_k).

Example 9.14 *A trie for strings (allex, ally, alone, along, arm, mixer, mixture, more) from an alphabet* `(a,l,e,x,y,o,n,g,r,m,i,x,t,u)`.



Compressed tries have the following properties:

1. Every internal node of T has at least two children and at most d children, where d is the alphabet size.
2. If T has s external nodes, the number of nodes of T is $O(s)$.

Activity 9.7

TRIES

1. Derive a standard trie, and a compressed trie, that store the following names: (aim, guy, jon, ann, jim, eva, amy, tim, ron, kim, tom, roy, kay, dot).
2. The *Hamming distance* between two strings of same length X and Y is the number of positions where the corresponding symbols are different. What is the Hamming distance between '101100' and '001101'?
3. The *edit distance*[13] between two strings X and Y is defined as the *minimum number of edit operations (insertion, deletion, or substitution of a single character) to transform from string X to Y*. What is the edit distance between 'algorithm' and 'rhythm'?

[13] Also called 'Levenshtein distance'.

4. Design an $O(nm)$ time algorithm for computing the edit distance between string X and Y of length n and m respectively.
5. Write a Java method `int editDistance(String X, Y)` to return the edit distance between two strings X and Y.

Chapter 10

Graphics and geometry

10.1 Essential reading

Jurg Nievergelt and Klaus H Hinrichs *Algorithms & Data Structures*. (Prentice Hall, Inc., 1993) [ISBN 0-13-489428-6]. Chapter 23.3

10.2 Learning outcomes

In this chapter, we introduce a new tree data structure and briefly discuss a few fundamentals in graphics.

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

- describe a quadtree and grid files with examples
- implement some simple geometric objects in Java.

10.3 Quadtrees

The term *graphics* is widely used in daily life. However, here we mean the digital data that can be stored in a normal computer and displayed on a monitor.

Quadtrees are a tree data structure often used for partitioning and storing images in a two dimensional space by recursively subdividing it into four quadrants or regions. In a quadtree, each internal node has at most 4 children. Figure 10.1 shows an example of a quadtree.

For quadtrees such as in Figure 10.1, the quadtree uses so-called *breadth-first addressing* where the root is labelled as 0, and the children will be 1, 2, 3 and 4; and grandchildren $5, 6, \cdots 16$, etc.

The tree nodes can be accessed by address computation similar to that for heaps. For example, the addresses of the children of node x are $4x + 1$, $4x + 2$, $4x + 3$ and $4x + 4$. Similarly, the address of the parent of any node x is $(x - 1)$ div 4.

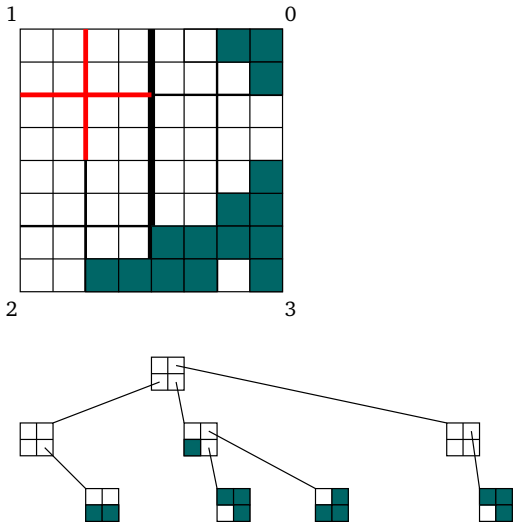


Figure 10.1: A quadtree

10.4 Octtrees

This is exactly the same idea as quadtrees but for three-dimensional space.

A cube is recursively partitioned into eight octants, using three orthogonal planes.

10.5 Grid files

Again this is a matrix data structure to store points and simple geometric objects in multidimensional space. An example in a two-dimensional domain: the Cartesian product X, Y, where X is a subrange of the integers, and Y is the ordered set of the 26 English letters. A bit map is a natural data structure for storing a set S of elements from $X \times Y$. Example (2004,w), (1902,r).

A boolean type of array $T[x, y]$ can be defined with the convention: 'if $T[x, y]$ = $true$ then $(x, y) \in S$' and vice versa.

10.6 Operations

Like other ADTs, routine operations and functions can be defined. For example,

Typical operations (Grid):

find(x,y) returns *true* if a point (x,y) is found in a given object
insert(x,y) add one point (x,y) in a given object
delete(x,y) remove one point (x,y) from a given object.

10.7 Simple geometric objects

In addition to the routine operations, fundamental objects can also be defined and this eases algorithm design. You may find these on the menu of many drawing software such as the *Xfig* in Linux (`www.xfig.org/userman`)and *Paint* in Windows.

Here are examples of some geometric objects:

- points
- lines
- polylines
- rectangles
- circles
- ellipses.

10.8 Parameter spaces

The graphic objects are much more mobile than text. Text is usually filled in pages consisting of characters with restricted number of fonts and sizes. In contrast, pictures can be placed in different places with thousands of colours, shapes and dimensions to choose from. The objects in a graphical environment can be viewed from almost any angles, too. Lights and shades are different for different view points. Therefore, graphics require a much larger parameter space. Almost every operation involves some geometry parameters. The most common parameters required are as follows:

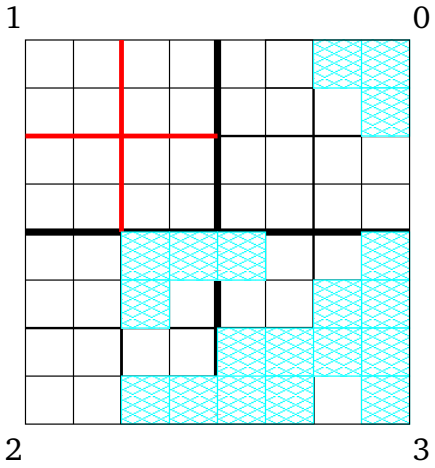
1. Location parameters: these are to indicate the location of an object, e.g. a point at (x, y).
2. Extension parameters: these are to represent the move between two locations (x_1, y_1) and (x_2, y_2), e.g. a shift (d_x, d_y), where $d_x = x_2 - x_1$, and $d_y = y_2 - y_1$.
3. Dependence of parameters: these are to define a geographical area, e.g. an interval (l_x, r_x), where $l_x \leq r_x$; or (l_y, r_y), where $l_y \leq r_y$.

A good choice of parameters and coordinate transformation may lead to more efficient data structures.

Activity 10.8

QUADTREES AND APPLICATION

1. Hand draw a *quadtree* for the following bitmap image:
2. Design and implement a GUI menu driven class to allow the user to create an empty binary search tree of `objects`, add one node to the binary search tree, and display all the nodes of the tree graphically.



Here is an example of a primitive menu:

```
    Binary Search Tree
  ----------------------
    1. Initialise
    2. Insert one node
    3. Display all nodes
    0. Quit
Please input your choice (0-3) >
```

3. Create a separate applet with a password that can be selected by a menu or button from an HTML page and will enable one of the users to change the font (or colour) of text "Java programming is great fun!" display interactively.

Chapter 11

Revision for CIS226b examination

This chapter covers the issues of revision and the examination.

11.1 Examination

The duration of the CIS226 examination will be *three* hours in all.

The examination paper contains *two* sections, for example, Section A and B. The examination for CIS226b, on the topics in this Subject Guide, will be in one section.

There are a number of (usually three to six) sub-questions in each question.

11.2 Revision materials

You should make good use of the following:

- this subject guide
- lecture notes or other handouts if you are studying at an institution
- lab exercises and your solutions
- coursework and your solutions
- textbook(s) on the reading list
- any other useful materials from the Internet.

11.3 Questions in the examination

The questions in the examination can be classified into the following three types:

- **Bookwork** The answers to this type of question can be found in the subject guide or directed reading.
- **Similar question** The question is similar to examples in the *revision materials* listed above. These are usually questions that require examples to be provided in addition to the definitions and explanation of some concepts.
- **Unseen question** You will not normally have seen this type of question before the examination, but you should be able to answer these questions using the knowledge and experience gained from the course. The questions require some deeper understanding from students.

The questions can also be classified according to the level of requirement of the student. For example:

- **Definition question** The answer to this type of question requires merely basic knowledge of a definition in the *Subject Guide*, or the *directed reading*. Questions are likely to begin with 'What is xxx ..?', 'Explain what xxx is ..'.
- **Conceptual question** This type of question requires a good understanding of some important concepts. The questions seek mainly for some concepts to be reviewed in relation to what you have learnt. The questions are likely to start with 'Discuss xxx ..', 'Explain why xxx ..'.
  The answers to these questions should include not only the definition, theorem, etc. learnt from the course, but also supporting examples to demonstrate good understanding of issues related to the concepts.
- **Do-it-yourself question** This type of question requires deeper understanding than that for Conceptual questions. The answers to the questions require not only knowledge gained from the course, but also a certain degree of problem-solving skills.
  The questions normally consist of instructions to complete certain tasks. For example, 'Write a piece of Java code to ...', 'Draw a diagram for ...', 'Traverse the graph ...', or 'Analyse xxx ...', etc.

11.4 Read questions carefully

Every year students are advised to read the questions on the examination paper *carefully*. You should make sure that you fully understand what is required and what subsections are involved in a question. You are encouraged to make notes, if necessary, while attempting the questions. Above all, you should be completely familiar with the course materials. To achieve a good grade, you need to have prepared well for the examination and to be able to solve problems by applying the knowledge gained from your studies on the course.

11.5 Recommendation

The following activities are recommended in your revision:

1. Review the Subject Guide, directed reading and lab exercises again.
2. Write out the definitions and new concepts learnt from the course in your own words.
3. Find a good example to support or help explain each definition or concept.
4. Review the examples given in the Subject Guide and directed reading. Try to solve some problems using the skills learnt from the course.

11.6 Revision topics I

We have intensively studied the following important topics:

1. Data structures and ADTs with Java objects and classes
2. Problem solving techniques
3. Algorithm design and implementation
4. Algorithm efficiency
5. Analysis in Big-O notation
6. Lists, stacks, queues
7. Binary trees, complete binary trees, balanced binary trees
8. Binary search trees, quad-trees, tries, heaps
9. Adjacency matrices, adjacency lists
10. Implementation in Java for basic structures of linked lists, binary tree structure and graphs
11. Divide and conquer
12. Recursive algorithms and methods
13. Implementing simple recursive algorithms in Java
14. Traversals, searching and sorting
15. Basic string matching algorithms.

11.7 Revision topics II

The following topics are important in Computer science although we have only had a limited amount of time to cover them briefly:

- Hashing (maps)
- Recursion as a problem-solving technique
- Dynamic programming
- Greedy approach
- Intractability

Both topics I and II are examinable.

11.8 Good luck!

Well done if you have made it this far to prepare for the examination!

I hope you find the materials in this course module stimulating and useful, and that you enjoy your studies as much as I have enjoyed writing this subject guide.

Finally, I wish you all the best in the examination!

Appendix A

Sample examination paper

Algorithm design and analysis

Note: Requirements may change from year to year.

Duration: 1 hour and 30 minutes are expected for each section[1].

[1] e.g. Section A: Software Engineering, Section B: Algorithm Design and Analysis.

You must answer **FOUR** *questions only,* TWO *questions from section A and* TWO *questions from section B.*

The full marks for each question are 25. The marks for each subquestion are displayed in brackets (e.g. [3]).

Electronic calculators are not allowed.

Section B

Question 4

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.* *[6]*

```
insertionsort(int array[0..n-1])
input:  _________________
output: _________________
for i=1 to n-1
  current=array[i]
  position=i-1
    while position>=1 and current<array[position]
      array[position+1]=array[position]
      position=position-1
    endwhile
  array[position+1]=current
endfor
```

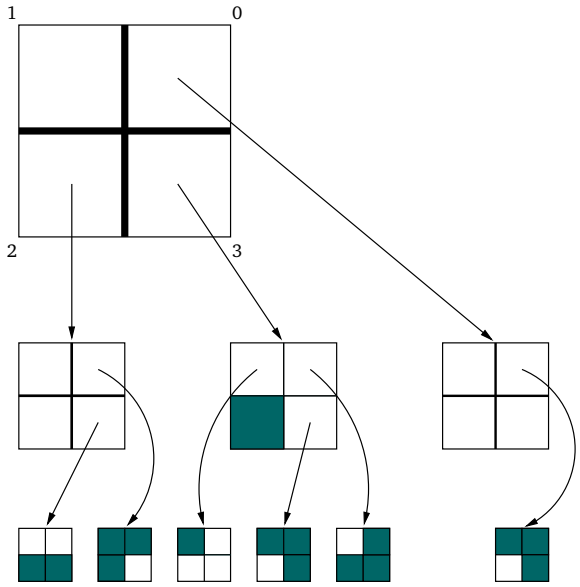
2. *Draw the binary search tree for the sequence valued 15 13 17 14 11 16 19, where the values are inserted in this order.* *[3]*
3. *What is a (binary) heap? What are the two main properties of a heap?* *[3]*
4. *Consider the list of integers in the array below, where i is the index of the array. Suppose that the search target is 9. Discuss briefly the difference between the behaviour of the sequential search algorithm and the binary search algorithm.* *[6]*

```
i      1 2 3 4 5 6 7
A[i]   7 4 8 3 9 1 2
```

5. *Derive and draw a diagram of the* `compressed trie` *for the set of strings below.* *[3]*

(“array”, “map”, “apply”, “middle”, “method”, “apple”, “key”, “kettle”).

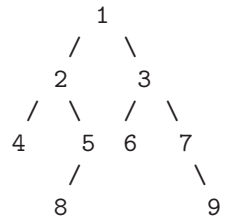
6. *Given the quad-tree below, draw the original black-white (b-w) pixel block (image) that the tree represents.* *[4]*



Question 5

1. *Derive and list the contents of an open-address hash table H[0···22] after the insertion of the keys (3,4,5,26,6,7,23,16,39,17,22,55), one by one in that order. You may assume $h(k) = k$ mod 23 with linear probing, and the hash table is empty initially.* *[5]*

2. *List the sequence of numbers visited in a postorder traversal of the binary tree below:* *[3]*



3. *Define a class* `TreeNode` *in Java which provides a reference-based implementation for the ADT binary tree. Each tree node should contain at least three data fields, namely* `leftChild`, `treeItem` *(*`Object` *type) and* `rightChild`*, and the necessary operations for initialising and accessing a tree node.* *[9]*

4. *Consider the adjacency list below. Draw the graph which is represented by the list and derive the adjacency matrix for the graph.* *[4]*

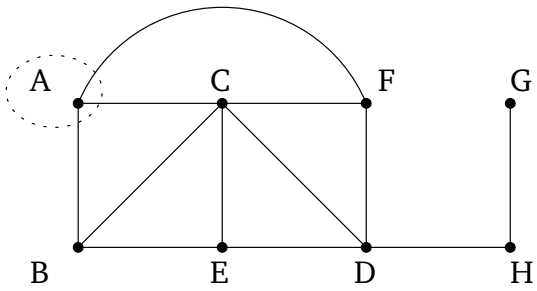
```
1 A --> E
2 B --> A
3 C
4 D
5 E --> B --> C --> D
```

5. *Draw a figure to illustrate step by step the comparisons done by the Boyer-Moore pattern matching algorithm for the case in which the text* `T` *and the Pattern* `P` *are:* *[4]*

```
T: a b a c a a b a d c a b a c a b a a b b
P: a b a c a b
```

Question 6

1. *Consider the connected graph below. Starting from vertex* **A**, *write the vertex sequence in the order that each vertex is visited applying the breadth first traversal algorithm.* *[3]*



2. *Consider the following algorithm* `StackAndQueue`. *Explain what it does essentially. Suppose that the stack* `S` *is, initially, empty and the queue* `Q` *contains* `3, 4, 5, 3, 2, 1` *with 3 in front. What are the elements in* `S` *and* `Q` *on completion of the execution of the algorithm?* *[4]*

```
StackAndQueue(queue: Q);

stack S
integer item
begin
  initialise(S)
  while not Empty(Q) do
    begin
      dequeue(Q,item)
      push(S,item)
    end
  while not Empty(S) do
    begin
      pop(S,item)
      enqueue(Q,item)
    end
end
```

3. *Write in pseudocode a recursive method* `PowersRecursive(x,n)` *to compute* x^n. *[5]*

```
method PowersRecursive(x,n)
input: a real x, and a positive integer n
output: nth power of x
```

4. *A (binary) heap can be implemented easily by an array. Given a heap stored in the array below, illustrate the heap structure in a diagram.* *[5]*

   2 7 6 10 12 11 8 15

5. *Using an array of integers (7,6,8,5,9,1,2) as an example, demonstrate the difference between the insertion sort algorithm and the selection sort algorithm by tracing the sequence of states of the array used with each iteration of the algorithms.* *[8]*

Appendix B

Sample solutions

Algorithms design and analysis

Duration: 1 hour and 30 minutes for each section.

Please be aware that these solutions are given in brief note form for the purpose of this section. Candidates for the examination are usually expected to provide coherent answers in full form to gain full marks. Also, there are other correct solutions which are completely acceptable but which are not included here.

Section B

Question 4

1. *The main comparison* `current<array[position]` *has been chosen as the basic operation.* *[1/6]*

   The input of the algorithm: An array of integers to be sorted *[0.5/6]*

   The output of the algorithm: An array of integers sorted in ascending order. *[0.5/6]*

   For n number of elements in the array, the for-loop executes $n-1$ times.

   Worst case: *[3/6]*

   ```
     i   position  number of comparisons
   -------------------------------------
     1     1       1
     2     ...
     3
     ...
     n-1
   ```

   So the time complexity at the worst case is $W(n) = O(n^2)$. *[1/6]*

2. *[3]*

   ```
          15
        /   \
       13     17
      / \    / \
     11 14 16  19
   ```

3. *A heap is a complete binary tree with a special data arrangement, in which no data in any node is bigger (or smaller) than that in its parent.* *[2/3]*

   They are structure property and order property. *[1/3]*

4. *The sequential search algorithm compares the target with every element in the worst case.*

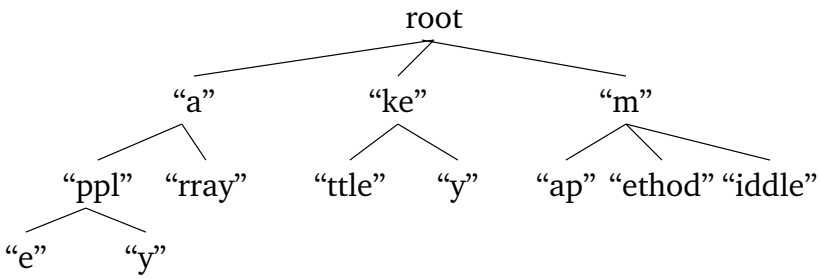
   A correct example... *[1+1 /6]*

   The binary search algorithm can only be applied to a sorted list. So the given list has to be sorted into B[]: *[2/6]*

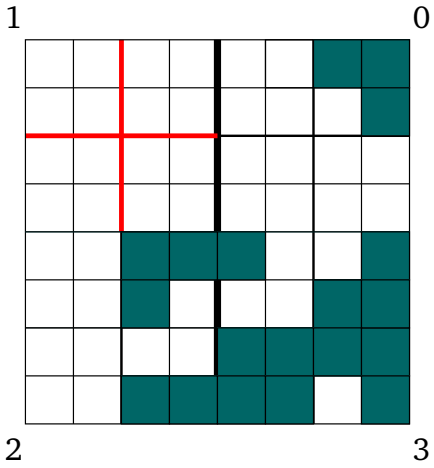
   ```
   i      1 2 3 4 5 6 7
   B[i]   1 2 3 4 7 8 9
   ```

   It then searches the target in each of the two half lists depending on whether the target 9 is smaller than the middle element. If yes, the search will be carried on the lower half of the list. Otherwise, the higher half. This process repeats until either the target is found or the list is exhausted. *[1+1 /6]*

5. *The compressed trie is:* *[3]*



6. *The original pixel block is as below:* *[4]*



Question 5

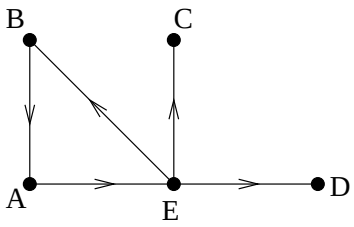
1. *[5]*

```
0  1 2 3 4 5 6  7 8 9  10 11 12 13 14 15 16 17 18 19 20 21 22
23     3 4 5 26 6 7 55                   16 39 17          22
```

2. *The numbers visited: postorder:* `4 8 5 2 6 9 7 3 1` *[3]*

3.
```
public class TreeNode {
  private Object treeItem;
  private TreeNode leftChild, rightChild;
```
[1/9]
```
  public TreeNode(Object newItem) {
    treeItem = newItem;
    leftChild = null;
    rightChild = null;
  } // Constructor

  public TreeNode(Object newItem, TreeNode left, TreeNode right) {
    treeItem = newItem;
    leftChild = left;
    rightChild = right;
  } // Constructor
```
[1+1/9]
```
  public void setItem(Object newItem) {
    treeItem = newItem;
  }

  public Object getItem() {
    return treeItem;
  }

  public void setLeft(TreeNode left) {
    leftChild = left;
  }

  public Node getLeft() {
    return leftChild;
  }

  public void setRight(TreeNode right) {
    rightChild = right;
  }

  public Node getRight() {
    return rightChild;
  }
} // end class TreeNode
```
[1/9 x6]

4. Figure: *[2/4]*



List: *[2/4]*

```
   A B C D E
A  0 0 0 0 1
B  1 0 0 0 0
C  0 0 0 0 0
D  0 0 0 0 0
E  0 1 1 1 0
```

5. *[0.5x6+1]*

```
a b a c a a b a d c a b a c a b a a b b
a b a c a b
         1 (comparison)

a b a c a a b a d c a b a c a b a a b b
  a b a c a b
       4 3 2

a b a c a a b a d c a b a c a b a a b b
    a b a c a b
             5

a b a c a a b a d c a b a c a b a a b b
      a b a c a b
               6

a b a c a a b a d c a b a c a b a a b b
                a b a c a b
                         7

a b a c a a b a d c a b a c a b a a b b
                  a b a c a b
                  1 1 1 1 9 8
                  3 2 1 0
```

Question 6

1. *One possible solution:*

   ABCFEDHG (or AFCBEDHG) *[3]*

2. *The algorithms will use the stack to reverse the order of all the items in the queue.* *[2/4]*

   So the elements of the queue will be `1,2,3,5,4,3`*,* *[1/4]*

   and the stack is empty. *[1/4]*

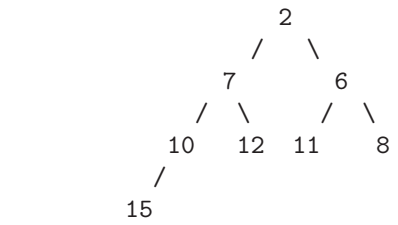
3.
```
double PowersRecursive(x,n)
input: a real x, a positive integer n
output: nth power of x
if n=1 then power=x
else
  if even(n) then power=PowersRecursive(x*x,n/2)
  else
      power=x*PowersRecursive(x*x,(n-1)/2)
  endif
endif
PowersRecursive=power
end.
```

   [1/5 x5]

   Other solutions may be acceptable. For example:

```
double PowersRecursive(x,n)
input: a real x, a positive integer n
output: nth power of x
if n=1 then power=x
else
  if n>1 then power=x*PowersRecursive(x,n-1)
  endif
endif
return power
end.
```

4. *[5]*



5. *Insertion sort:* *[4/8]*

i	1	2	3	4	5	6	7	current
A[i]	7	6	8	5	9	1	2	6
	6	7	8	5	9	1	2	8
	6	7	8	5	9	1	2	5
	5	6	7	8	9	1	2	9
	5	6	7	8	9	1	2	1
	1	5	6	7	8	9	2	2
	1	2	5	6	7	8	9	

   Selection sort: *[4/8]*

i	1	2	3	4	5	6	7	current
A[i]	7	6	8	5	9	1	2	7
	1	6	8	5	9	7	2	6
	1	2	8	5	9	7	6	8
	1	2	5	8	9	7	6	8

```
1 2 5 6 9 7 8     9
1 2 5 6 8 7 9     7
1 2 5 6 7 8 9     8
```

Appendix C

Pseudocode notation

Pseudocode is a convenient way to describe and convey algorithmic ideas. It would be acceptable to use any ad hoc English based language as long as the algorithmic idea is clear. However, one useful function of pseudocode is that it can serve as a bridge or stage for converting an algorithm to a source code of a computer program. So an ideal pseudocode would be *close* enough in syntax to a conventional computer language to remove ambiguity that can arise with natural language.

In this Subject Guide, we use a hybrid of adopted keywords and syntax from several commonly used high-level sequential computer languages. Of course, the user can extend the pseudocode vocabulary by adding more useful terms, function names etc.

C.1 Values

- integers or digits: 0, 1, $\cdots$, 9
- fraction, rational numbers
- *true*, *false*
- English characters or strings.

C.2 Types

boolean, int, real, char, string, object.

C.3 Operations

- $\leftarrow$ (assignment)
- (), [], {}
- +, $-$, $\times$, /
- and (&&), or (||), xor (`XOR`)
- $<$, $>$, $\leq$, $\geq$, $=$, $\neq$.

C.4 Priority

In some cases, there are nested structures, or a long expression consists of many items.

- nested brackets: from inside out
- an expression with equal priority: from left to right
- algorithm without line numbers: from the top line down.

C.5 Data structures

- array, list, queue, stack, set
- tree, graph, matrix
- hash-table.

C.6 Other reserved words

- function
- procedure
- method
- (typed) method.

Note: the terms `procedure` and `function` are used in languages such as Pascal, C, C++ and `method` is the concept in Java. In our pseudocode, keywords `function` and `typed method`, and `procedure` and `(void) method` are interchangeable.

C.7 Control keywords

These are the keywords to indicate a block of statements of an algorithm.

- **if – end if**
- **for – end for**
- **repeat – until**
- **while – end while**
- **return**.

C.8 Examples of sequential structures

1. A method, function or procedure

   method xxx (type a, b, c)
   input: type a, b, c
   output: $a + b + c$
   other statements

   type **method** xxx (type a, b, c)
   return $a + b + c$

   type **function** xxx (type a, b, c)
   return $a + b + c$

procedure xxx (type a, b, c)
return $a + b + c$

2. If-then-else

if condition **then**
  other statements
end if

if condition **then**
  other statements
else
  other statements
end if

if condition **then**
  other statements
else if condition **then**
  other statements
else
  other statements
end if

3. A Boolean function (or method)

boolean function xxx (int a, b)
return $a > b$

4. A loop (iteration) structure

for $i = 1$ to n **do**
  other statements
end for

while condition **do**
  other statements
end while

repeat
  other statements
until condition

Acknowledgements

The author wishes to thank Dr David Brownwigg and editor Christina Loziol-Webster for proof reading and valuable comments.