

**COURSE
GUIDE****CIT 216
FUNDAMENTALS OF DATA STRUCTURES**

Dr. Vivian Nwaocha (Course Developer/Writer) - NOUN

Dr. S. Reju (Programme Leader) - NOUN

Dr. Vivian Nwaocha (Course Developer/Writer) - NOUN

**NATIONAL OPEN UNIVERSITY OF NIGERIA**

© 2022 by NOUN Press
National Open University of Nigeria
Headquarters
University Village
Plot 91, Cadastral Zone
Nnamdi Azikiwe Expressway
Jabi, Abuja

Lagos Office
14/16 Ahmadu Bello Way
Victoria Island, Lagos

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

First Printed 2009
Revised and Reprinted 2022

ISBN: 978-058-031-X

All Rights Reserved

CONTENTS	PAGE
Introduction.....	1
What this Course Will Help You Do.....	1
Course Aims.....	1
Course Objectives.....	1
Working through this Course.....	2
Course Materials.....	2
Online Materials.....	2
Equipment.....	2
Study Units	3
Assessment.....	5
Course Overview.....	5
How to Get the Most from this Course.....	5
Facilitators/Tutors and Tutorials.....	6
Summary.....	7

Introduction

The course, Data Structures, is a foundational course for students studying towards acquiring the Bachelor of Science in Communication Technology degree. In this course, we will study programming techniques including data structures and basic algorithms for manipulating them.

The overall aims of this course are to introduce you to programming concepts as and algorithm techniques. Topics related to data structures and storage management are equally discussed.

The bottom-up approach is adopted in structuring this course. We start with the basic building blocks of object-oriented programming concepts and move on to the fundamental principles of data structures and algorithms.

What this Course Will Help You Do

The overall aims and objectives of this course provide guidance on what you should be achieving in the course of your studies. Each unit also has its own unit objectives which state specifically what you should be achieving in the corresponding unit. To evaluate your progress continuously, you are expected to refer to the overall course aims and objectives as well as the corresponding unit objectives upon the completion of each.

Course Aims

The overall aims and objectives of this course include:

Develop your knowledge and understanding of the underlying principles of foundational data structures.

Build up your capacity to evaluate different algorithm techniques.

Develop your competence in analysing data structures.

Build up your capacity to write programmes for developing simple applications.

Course Objectives

Upon completion of the course, you should be able to:

Describe the basic operations on stacks, lists and queue data structures.

Explain the notions of trees, hashing and binary search trees.

Identify the basic concepts of object-oriented programming.

Develop java programmes for simple applications.

Discuss the underlying principles of basic data types: lists, stacks and queues.

Describe structures and algorithms for external storage: external sorting, external search trees.

Identify directed and undirected graphs.

Discuss sorting: internal and external sort.

Describe the efficiency of algorithms, recursion and recursive programmes.

Discuss the algorithm design techniques: greedy algorithms, divide-and-conquer algorithms, dynamic programming.

Working through this Course

We designed this course in a systematic way, so you need to work through it from Module one, Unit 1 through to Module 6, Unit 6. This will enable you appreciate the course better.

Course Materials

Basically, we made use of textbooks and online materials. You are expected to search for more literature and web references for further understanding. Each unit has references and web references that were used to develop them.

Online Materials

Feel free to refer to the websites provided for all the online reference materials required in this course. The website is designed to integrate with the print-based course materials. The structure follows the structure of the units and all the reading and activity numbers are the same in both media.

Equipment

In order to get the most from this course, it is essential that you make use of a computer system which has internet access.

Recommended System Specifications:**Processor**

2.0 GHZ Intel compatible processor
1GB RAM
80 GB hard drive with 5 GB free disk
CD-RW drive
3.5" Floppy Disk Drive
TCP/IP (installed)

Operating System

Windows XP Professional (Service Pack)
Microsoft Office 2007
Java Programming Language
Norton Antivirus

Monitor*

19-inch
1024 X 768 resolution
16-bit high color
*Non Standard resolutions (for example, some laptops) are not supported.

Hardware

Open Serial Port (for scanner)
120W Speakers
Mouse + pad
Windows keyboard

Laser Printer

Hardware is constantly changing and improving, causing older technology to become obsolete. An investment in newer, more efficient technology will more than pay for itself in improved performance results.

If your system does not meet the recommended specifications, you may experience considerably slower processing when working in the application. Systems that exceed the recommended specifications will provide better handling of database files and faster processing time, thereby significantly increasing your productivity.

Study Units

There are 6 modules in this course. Each module comprises 5 units which you are expected to complete in 3 hours. The 6 modules and their units are listed below.

Module 1 Foundational Data Structures

Unit 1	Fundamentals
Unit 2	Arrays
Unit 3	The List Data Structure
Unit 4	The Stack Data Structure
Unit 5	The Queue Data Structure

Module 2 Hashing and Trees

Unit 1	Hashing
Unit 2	Trees
Unit 3	Search Trees
Unit 4	Garbage Collection
Unit 5	Memory Allocation

Module 3 Introduction to Java Programming

Unit 1	Object-Oriented Programming Concepts
Unit 2	Variables
Unit 3	Operators
Unit 4	Expressions, Statements and Blocks
Unit 5	Control Flow Statements

Module 4 Java Programming

Unit 1	Classes
Unit 2	Objects

Unit 3	Interfaces and Inheritances
Unit 4	Numbers and Strings
Unit 5	Generics

Module 5 Algorithms

Unit 1	Introduction to Algorithms
Unit 2	Vectors and Matrices
Unit 3	Greedy Algorithm
Unit 4	Divide-and-Conquer Algorithm
Unit 5	Dynamic Programming Algorithm

Module 6 Graphs and Sorting

Unit 1	Graph Algorithm
Unit 2	Sorting
Unit 3	Bubble Sort
Unit 4	Insertion Sort
Unit 5	Selection Sort
Unit 6	Merge Sorting

From the preceding, the content of the course can be divided into two major blocks:

Foundational Data Structures
Introduction to Java Programming

Modules one and two describe the foundational data structures and their underlying principles. Modules three and four define the basic concepts of an object-oriented programming language (Java). It uses java as a programming language to implement a variety of data structures, while modules five and six discuss the analysis of algorithms and algorithm techniques.

Assessment

The course, Data Structures entails attending a three-hour final examination which contributes 50% to your final grading. The final examination covers materials from all parts of the course with a style similar to the Tutor-marked assignments.

The examination aims at testing your ability to apply the knowledge you have gain throughout the course, rather than your ability to memorise the materials. In preparing for the examination, it is essential that you receive the activities and Tutor-marked assignments you have completed in each unit. The other 50% will account for all the TMAs at the end of each unit.

Course Overview

This section proposes the number of weeks that you are expected to spend on the three modules comprising 30 units and the assignments that follow each of the units.

We recommend that each unit with its associated TMA is completed in one week, bringing your study period to a maximum of 30 weeks.

How to Get the Most from this Course

In order for you to learn various concepts in this course, it is essential to practice. Independent activities and case activities which are based on a particular scenario are presented in the units. The activities include open questions to promote discussion on the relevant topics, questions with standard answers and programme demonstrations on the concepts. You may try to delve into each unit adopting the following steps:

- Read the study unit
- Read the textbook, printed or online references
- Perform the activities
- Participate in group discussions
- Complete the tutor-marked assignments
- Participate in online discussions.

This course makes intensive use of materials on the world-wide web. Specific web address will be given for your reference. There are also optional readings in the units. You may wish to read these to extend your knowledge beyond the required materials. They will not be assessed.

Facilitators/Tutors and Tutorials

About 20 hours of tutorials will be provided in support of this course. You will be notified of the dates, time and location for these tutorials, together with the name and phone number of your tutor as soon as you are allotted a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your TMAs to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by phone or e-mail, if you need help. The following might be circumstances in which you would find help necessary. You can also contact your tutor if:

- i. You do not understand any part of the study units or the assigned readings.
- ii. You have difficulty with the TMAs.
- iii. You have a question or problem with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend tutorials, since it is the only opportunity to have an interaction with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain maximum benefit from the course tutorials, you are advised to prepare a list of questions before attending the tutorial. You will learn a lot from participating in discussions actively.

Summary

The course, Data Structures, is intended to develop your understanding of the basic concepts of object-oriented programming, thus enabling you acquire skills in programming using java. This course also provides you with practical knowledge and hands-on experience in designing and implementing foundational data structures.

We hope that you will find the course enlightening and that you will find it both interesting and useful. In the longer term, we hope you will get acquainted with the National Open University of Nigeria and we wish you every success in your future.

CONTENTS	PAGE
Module 1 Foundational Data Structures.....	1
Unit 1 Fundamentals.....	1
Unit 2 Arrays.....	7
Unit 3 The List Data Structure.....	12
Unit 4 The Stack Data Structure.....	18
Unit 5 The Queue Data Structure.....	24
Module 2 Foundational Data Structures.....	32
Unit 1 Hashing.....	32
Unit 2 Trees.....	43
Unit 3 Search Trees.....	54
Unit 4 Garbage Collection.....	70
Unit 5 Memory Allocation.....	80
Module 3 Introduction to Java Programming.....	85
Unit 1 Object-Oriented Programming Concepts.....	85
Unit 2 Variables.....	95
Unit 3 Operators.....	99
Unit 4 Expressions, Statements and Blocks.....	107
Unit 5 Control Flow Statements.....	112
Module 4 Introduction to Java Programming.....	122
Unit 1 Classes.....	122
Unit 2 Objects	129
Unit 3 Interfaces and Inheritances.....	140
Unit 4 Numbers and Strings.....	150
Unit 5 Generics.....	159
Module 5 Algorithms.....	168
Unit 1 Introduction to Algorithms.....	168
Unit 2 Vectors and Matrices.....	174
Unit 3 Greedy Algorithms.....	179
Unit 4 Divide-and-Conquer Algorithm.....	182
Unit 5 Dynamic Programming Algorithm.....	185
Module 6 Algorithms.....	189
Unit 1 Graph Algorithm.....	189
Unit 2 Sorting.....	193
Unit 3 Bubble Sort.....	197
Unit 4 Insertion Sort.....	201
Unit 5 Selection Sort.....	205
Unit 6 Merge Sorting.....	210

MODULE 1 FOUNDATIONAL DATA STRUCTURES

Unit 1	Fundamentals
Unit 2	Arrays
Unit 3	The List Data Structure
Unit 4	The Stack Data Structure
Unit 5	The Queue Data Structure

UNIT 1 FUNDAMENTALS**CONTENTS**

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	Data Type
3.2	Data Type Classification
3.2.1	Examples of Data Type
3.3	Abstract Data Type
3.3.1	Examples of Abstract Data Type
3.4	What is a Data Structure?
3.5	Classification of Data Structures
3.5.1	Linear Data Structure
3.5.2	Non-Linear Data Structure
3.6	Data Structures and Programmes
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings

1.0 INTRODUCTION

This unit introduces some basic concepts that the student needs to be familiar with before attempting to develop any software. It describes data type and data structures, explaining the operations that may be performed on them. The unit introduces you to the fundamental notions of data structures, thus guiding you through and facilitating your understanding of the subsequent units.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe and use the following notions; data type, abstract data type and data structure
- outline the classification of data type
give typical examples of data type
- explain the relevance of data structures in programming.

3.0 MAIN CONTENT

3.1 Data Type

In computer programming, a **data type** simply refers to a defined kind of data, that is, a set of possible values and basic operations on those values.

When applied in programming languages, a data type defines a set of values and the allowable operations on those values.

Data types are important in computer programmes because they classify data so that a translator (compiler or interpreter) can reserve appropriate memory storage to hold all possible values, e.g. integers, real numbers, characters, strings, and Boolean values, all have very different representations in memory.

A *data type* consists of:

- a domain (= a set of values)
- a set of operations that may be applied to the values.

3.2 Data Type Classification

Some data items may be used singly whilst others may be combined together and arranged to form other data items. The former are classified as 'simple data types' whereas the latter are classified as 'data structures'. However, the following classification is appropriate for study at this level. The simple data types are classified as follows:

- a. Character
- b. Numeric integer
- c. Numeric real
- d. Boolean (logical).

3.2.1 Examples of Data Types

Almost all programming languages explicitly include the notion of data type, though different languages may use different terminology. Common data types in programming languages include those that represent integers, floating point numbers, and characters, and a language may support many more.

Example 1: *Boolean* or *logical* data type provided by most programming languages.

Two values: true, false.

Many operations, including: AND, OR, NOT, etc.

Example 2: In Java programming language, the “**int**” type represents the set of 32-bit integers ranging in value from -2,147, 483, 648 to 2,147, 483, 647 and the operation such as addition, subtraction and multiplication that can be performed on integers.

3.3 Abstract Data Type

An Abstract Data Type commonly referred to as **ADT**, is a **collection of data objects characterized by how the objects are accessed**; it is an abstract human concept meaningful outside of computer science. (Note that "object", here, is a general abstract concept as well, i.e. it can be an "element" (like an integer), a data structure (e.g. a list of lists), or an instance of a class. (e.g. a list of circles). A data type is abstract in the sense that it is independent of various concrete implementations.

Object-oriented languages such as C++ and Java provide explicit support for expressing abstract data types by means of *classes*. A first class abstract data type supports the creation of multiple instances of ADT and the interface normally provides a constructor, which returns an abstract handle to new data, and several operations, which are functions accepting the abstract handle as an argument.

3.3.1 Examples of Abstract Data Type

Common abstract data types (ADT) typically implemented in programming languages (or their libraries) include: Arrays, Lists, Queues, Stacks and Trees.

3.4 What is a Data Structure?

A **data structure** is the **implementation of an abstract data type** in a particular programming language. Data structures can also be referred

to as “data aggregate”. A carefully chosen data structure will allow the most efficient algorithm to be used. Thus, a well-designed data structure allows a variety of critical operations to be performed using a few resources, both execution time and memory spaces as possible.

3.5 Classification of Data Structures

Data structures are broadly divided into two:

Linear Data Structures

Non-Linear Data Structures.

3.5.1 Linear Data Structures

Linear data structures are data structures in which individual data elements are stored and accessed linearly in the computer memory. For the purpose of this course, the following linear data structures would be studied: lists, stacks, queues and arrays in order to determine how information is processed during implementation.

3.5.2 Non-Linear Data Structures

A non-linear data structure, as the name implies, is a data structure in which the data items are not stored linearly in the computer memory, but data items can be processed using some techniques or rules. Typical non-linear data structures to be studied in this course are Trees.

3.6 Data Structures and Programmes

The structure of data in the computer is very important in software programmes, especially where the set of data is very large. When data is properly structured and stored in the computer, the accessibility of data is easier and the software programme routines that make do with the data are made simpler; time and storage spaces are also reduced.

In the design of many types of programmes, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure.

SELF ASSESSMENT EXERCISE 1

Define exhaustively the term ‘Abstract Data Type’.

SELF ASSESSMENT EXERCISE 2

What are the constituents of a Data Type? Give 2 typical examples of data types.

4.0 CONCLUSION

In this unit, you have learned about the classification of abstract data type, commonly referred to as ADT. You have also been able to understand the meaning of some notions such as; data type, abstract data type and data structures. Finally, you have been able to appreciate the significance of data structures in developing high-quality programmes.

5.0 SUMMARY

What you have learned borders on the basic notions of data structures. The subsequent units shall build upon these fundamentals.

6.0 TUTOR-MARKED ASSIGNMENT

You have just been nominated as a Programmer of a Software firm responsible for developing software for tertiary institutions. How would your knowledge of this course facilitate your task?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme*, (2nd Edition). New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL*, (2nd Edition), New Jersey: Prentice Hall,

Shaffer, Clifford A. (1998). *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 2 ARRAYS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Arrays
 - 3.2 Arrays and Programming
 - 3.3 Declaration of Arrays
 - 3.4 Multi-Dimensional Arrays
 - 3.5 Classification of Arrays
 - 3.6 Application of Arrays
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will learn about arrays, their declaration, dimensionality and applications. You will also learn how to distinguish between static and dynamic arrays.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe an array, its dimensionality and declaration
- explain the terms; element and array name
- express a two-dimensional array linearly
- distinguish between static and dynamic arrays
- explain the importance of arrays in computer applications.

3.0 MAIN CONTENT

3.1 Arrays

In Computer Science, an *array* is a data structure consisting of a group of elements that are accessed by indexing. Each data item of an array is known as an element, and the elements are referenced by a common name known as the array name.

3.2 Arrays and Programming

In Java, as in most programming languages, an array is a structure that holds multiple values of the same type. A Java array is also called an object. An array can contain data of the primitive data types. As it is an object, an array must be declared and instantiated. For example:

```
int[] anArray;  
anArray = new int[10];
```

An array can also be created using a shortcut. For example:

```
int[] anArray = {1,2,3,4,5,6,7,8,9,10}
```

An array element can be accessed using an index value. For example:

```
int i = anArray[5]
```

The size of an array can be found using the length attribute. For example:

```
int len = anArray.length
```

Before any array is used in the computer, some memory locations have to be created for storage of the elements. This is often done by using the **DIM** instruction of BASIC programming language or **DIMENSION** instruction of FORTRAN programming language. For example, the instruction:

```
DIM LAGOS (45)
```

will create 45 memory locations for storage of the elements of the array called LAGOS.

In most programming languages, each element has the same data type and the array occupies a contiguous area of storage. Most programming languages have a built-in array data type. Some programming languages support array programming which generalises operations and functions to work transparently over arrays as they do with scalars, instead of requiring looping over array members.

3.3 Declaration of Arrays

Variables normally only store a single value but, in some situations, it is useful to have a variable that can store a series of related values - using an array. For example, suppose a programme is required that will calculate the average age among a group of six students. The ages of the students could be stored in six integer variables in C:

```
int age1;
```

```
int age2;
int age3;
```

However, a better solution would be to declare a six-element array:

`int age[6];` This creates a six element array; the elements can be accessed as `age[0]` through `age[5]` in C.

A two-dimensional array (in which the elements are arranged into rows and columns) declared by say `DIM X(3,4)` can be stored as linear arrays in the computer memory by determining the product of the subscripts. The above can thus be expressed as `DIM X (3 * 4)` or `DIM X (12)`.

Multi-dimensional arrays can be stored as linear arrays in order to reduce the computation time and memory.

3.4 Multi-dimensional Arrays

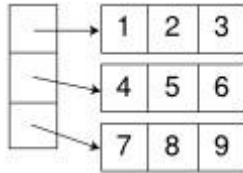
Ordinary arrays are indexed by a single integer. Also useful, particularly in numerical and graphics applications, is the concept of a *multi-dimensional array*, in which we index into the array using an ordered list of integers, such as in `a[3,1,5]`. The number of integers in the list used to index into the multi-dimensional array is always the same and is referred to as the array's *dimensionality*, and the bounds on each of these are called the array's *dimensions*. An array with dimensionality k , is often called k -dimensional. One-dimensional arrays correspond to the simple arrays discussed thus far; two-dimensional arrays are a particularly common representation for matrices. In practice, the dimensionality of an array rarely exceeds three. Mapping a one-dimensional array into memory is obvious, since memory is logically itself a (very large) one-dimensional array. When we reach higher-dimensional arrays, however, the problem is no longer obvious. Suppose we want to represent this simple two-dimensional array:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

It is most common to index this array using the *RC*-convention, where elements are referred in *row, column* fashion or $A_{row,col}$, such as:

$$A_{1,1} = 1, A_{1,2} = 2, \dots, A_{3,2} = 8, A_{3,3} = 9$$

Multi-dimensional arrays are typically represented by one-dimensional arrays of references (Ilf vectors) to other one-dimensional arrays. The subarrays can be either the rows or columns.



3.5 Classification of Arrays

Arrays can be classified as **static arrays** (*i.e.* whose size cannot change once their storage has been allocated), or **dynamic arrays**, which can be resized.

3.6 Applications of Arrays

Arrays are employed in many computer applications in which data items need to be saved in the computer memory for subsequent reprocessing. Due to their performance characteristics, arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks and strings.

4.0 CONCLUSION

In this unit, you have learned about the arrays and their dimensionality. You have also been able to understand the meaning of some notions such as; array name, element and array declaration. Finally, you have been able to distinguish between the static and dynamic arrays as well as understand the applications of arrays.

5.0 SUMMARY

What you have learned in this unit is focused on arrays, their declaration, classification and application. In the next unit, we will discuss another data structure known as Lists.

SELF ASSESSMENT EXERCISE 1

Interpret the instruction DIM Y (80).

SELF ASSESSMENT EXERCISE 2

Given DIMENSION A (5, 20), express the array linearly.

6.0 TUTOR-MARKED ASSIGNMENT

Describe a suitable data structure for details of stock items numbered in the range 1 to 100. Each stock item may be held at each of 20 locations. The number of items held at each location needs to be recorded.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A, (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 3 THE LIST DATA STRUCTURE

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 What is a List?
 - 3.2 Elements of a List
 - 3.3 Operation
 - 3.4 List Implementation
 - 3.4.1 Array List
 - 3.4.2 Linked List
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

What you will learn in this unit borders on Lists, their operations and implementations. Typical examples are given to facilitate the student's understanding of these features.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe a List
- identify the elements of a List
- explain the operations and implementations of Lists.

3.0 MAIN CONTENT

3.1 What is a List Data Structure?

A list data structure is a sequential data structure, i.e. a collection of items accessible one after the other, beginning at the **head** and ending at the **tail**. It is a widely used data structure for applications which do not need random access.

Lists differ from the stacks and queues data structures in that additions and removals can be made at any position in the list.

3.2 Elements of a List

The sentence “Dupe is not a boy” can be written as a list as follows:

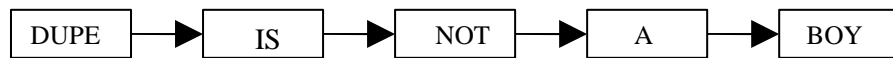


Fig. 1.0: Elements of a List

We regard each word in the sentence above as a data-item or datum, which is linked to the next datum, by a *pointer*. Datum plus pointer make one *node* of a list. The last pointer in the list is called a *terminator*. It is often convenient to speak of the first item as the *head* of the list, and the remainder of the list as the *tail*.

3.2 Operations

The main primitive operations of a list are known as:

Add	adds a new node
Set	updates the contents of a node
Remove	removes a node
Get	returns the value at a specified index
IndexOf	returns the index in the list of a specified element

Additional primitives can be defined:

IsEmpty	reports whether the list is empty
IsFull	reports whether the list is full
Initialise	creates/initialises the list
Destroy	deletes the contents of the list (may be implemented by re-initialising the list)
Initialise	Creates the structure – i.e. ensures that the structure exists but contains no elements e.g. <i>Initialise(L)</i> creates a new empty queue named Q

Add

e.g. *Add(1,X,L)* adds the value X to list L at position 1 (the start of the list is position 0), shifting subsequent elements up L



Fig. 1.1: List before adding value

L

A	X	B	C
---	---	---	---

Fig. 1.2: List after adding value**Set**

e.g. *Set(2,Z,L)* updates the values at position 2 to be Z

L

A	X	Z	C
---	---	---	---

Fig. 1.3: List after update**Remove**

e.g. *Remove(Z,L)* removes the node with value Z

L

A	X	Z	C
---	---	---	---

Fig. 1.4: List before removal

L

A	X	C
---	---	---

Fig. 1.5: List after removal**Get**

e.g. *Get(2,L)* returns the value of the third node, i.e. C

IndexOf

e.g. *IndexOf(X,L)* returns the index of the node with value X, i.e. 1

3.4 List Implementation

Lists can be implemented in many ways, depending on how the programmer will use lists in their programme. Common implementations include:

1. Array List
2. Linked List

3.4.1 Array Lists

This implementation stores the list in an array. The Array List has the following properties:

1. The position of each element is given by an index from 0 to $n-1$, where n is the number of elements.
2. Given any index, the element with that index can be accessed in constant time – i.e. the time to access does not depend on the size of the list.
3. To add an element at the end of the list, the time taken does not depend on the size of the list. However, the time taken to add an element at any other point in the list does depend on the size of the list, as all subsequent elements must be shifted up. Additions near the start of the list take longer than additions near the middle or end.
4. When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than removals near the middle or end.

3.4.2 Linked List

The Linked List is stored as a sequence of linked nodes. As in the case of the stack, each node in a linked list contains data AND a reference to the next node. The Linked List has the following properties:

The list can grow and shrink as needed.

The position of each element is given by an index from 0 to $n-1$, where n is the number of elements.

Given any index, the time taken to access an element with that index depends on the index. This is because each element of the list must be traversed until the required index is found.

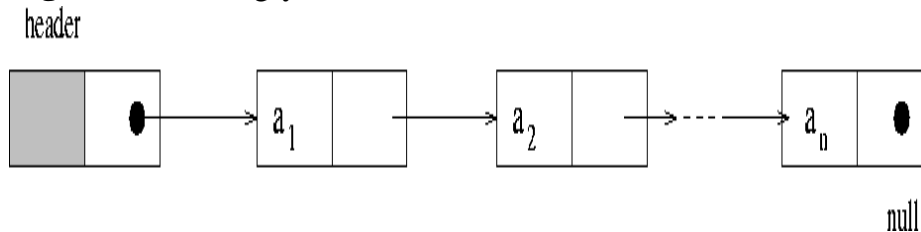
The time taken to add an element at any point in the list does not depend on the size of the list, as no shifts are required. It does, however, depend on the index. Additions near the end of the list take longer than additions near the middle or start. The same applies to the time taken to remove an element. A list needs a reference to the front node.

There are many variations on the Linked List data structure, including:

i. Singly Linked Lists

A singly linked list is a data structure in which the data items are chained (linked) in one direction. Figure 1 shows an example of a singly linked list.

Figure 1.6: A singly linked list



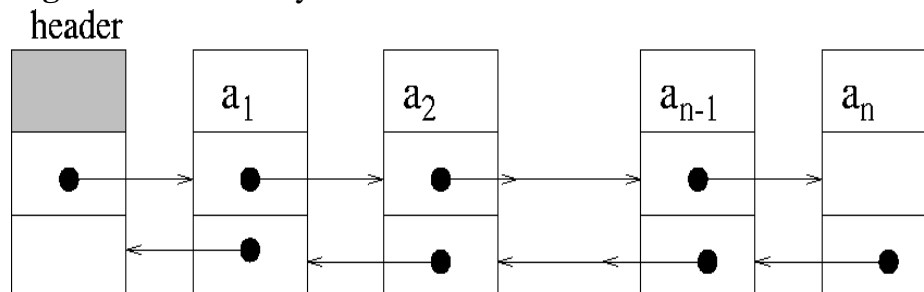
ii. Circularly Linked Lists

In a circularly linked list, the tail of the list always points to the head of the list.

iii. Doubly Linked Lists

These permits scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) In this case, the node structure is altered to have two links:

Figure 1.7: A doubly linked list



iii. Sorted Lists

Lists can be designed to be maintained in a given order. In this case, the Add method will search for the correct place in the list to insert a new data item.

SELF ASSESSMENT EXERCISE 1

‘Ola studies his courses’. Represent this statement as a list, identifying the different elements.

SELF ASSESSMENT EXERCISE 2

Mention at least two types of lists.

4.0 CONCLUSION

In this unit you have learned about Lists. You have also been able to identify the elements of a List. You should also have learned about operations and implementations of lists.

5.0 SUMMARY

What you have learned in this unit concerns the Lists, their operations and implementations. In the next unit, you shall learn about another linear data structure, known as Queues.

6.0 TUTOR-MARKED ASSIGNMENT

What would the contents of a list be after the following operation?
Add (1, X, L)

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 THE STACK DATA STRUCTURE

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 The Stack Data Structure
 - 3.2 Application of Stacks
 - 3.3 Operations on a Stack
 - 3.4 Stack Storage Modes
 - 3.4.1 Static Data Structures
 - 3.4.2 Dynamic Data Structures
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, we will look at an abstract data structure – the Stack Data Structure.

This structure stores and accesses data in different ways, which are useful in different applications. In all cases, the stack data structure follows the principle of data abstraction (the data representation can be inspected and updated only by the abstract data type's operations). Also, the algorithms used to implement the operations do not depend on the type of data to be stored.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe the stack data structure
- identify two basic modes of implementing a stack outline the applications of stacks in computing
- explain the two methods of storing a stack.

3.0 MAIN CONTENT

3.1 The Stack Data Structure

A **stack** is a linear data structure in which all insertions and deletions of data are made only at one end of the stack, often called the top of the

stack. For this reason, a stack is referred to as a LIFO (last-in-first-out) structure.

Figure 1.0 shows a stack.

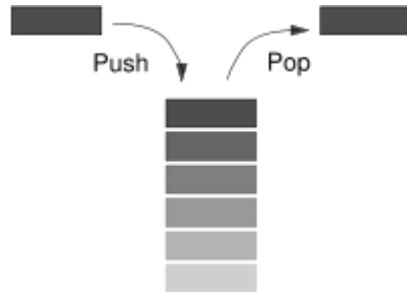


Fig. 1.0: Simple representation of a stack

A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, *pushing* the stack of plates down. As the top plate is removed from the stack, the plates *pop* back up, and the second plate becomes the top of the stack.

3.2 Application of Stacks

Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. Among other uses, stacks are used to run a Java Virtual Machine, and the Java language itself has a class called "Stack", which can be used by the programmer.

Stacks have many other applications. For example, as processor executes a programme, when a function call is made, the called function must know how to return back to the programme, so the current address of programme execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the programme resumes. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return back to calling programme. Stacks support recursive function calls, subroutine calls, especially when "reverse polish notation" is involved.

Solving a search problem, regardless of whether the approach is exhaustive or optimal, needs stack space. Examples of exhaustive search

methods are *bruteforce* and *backtracking*. Examples of optimal search exploring methods are *branch and bound* and *heuristic solutions*. All of

these algorithms use stacks to remember the search nodes that have been noticed but not explored yet.

Another common use of stacks at the architecture level is as a means of allocating and accessing memory.

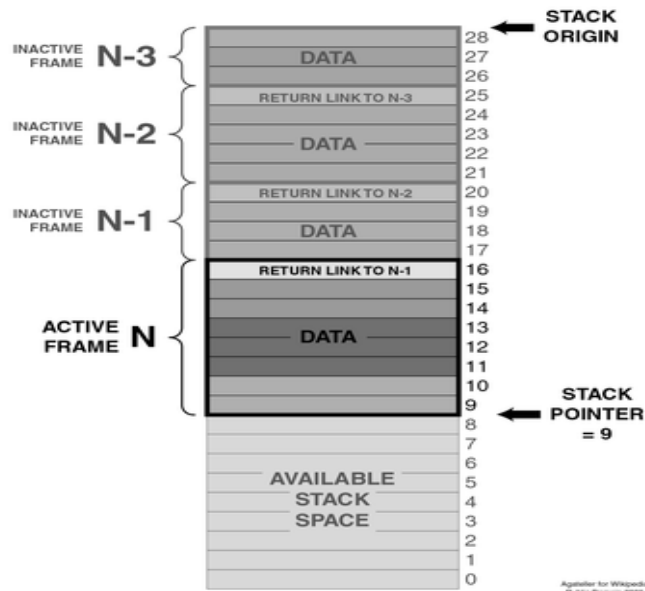


Fig. 1.1: Basic Architecture of a Stack

3.3 Operations on a Stack

The stack is usually implemented with two basic operations known as "push" and "pop". Thus, two operations applicable to all stacks are:

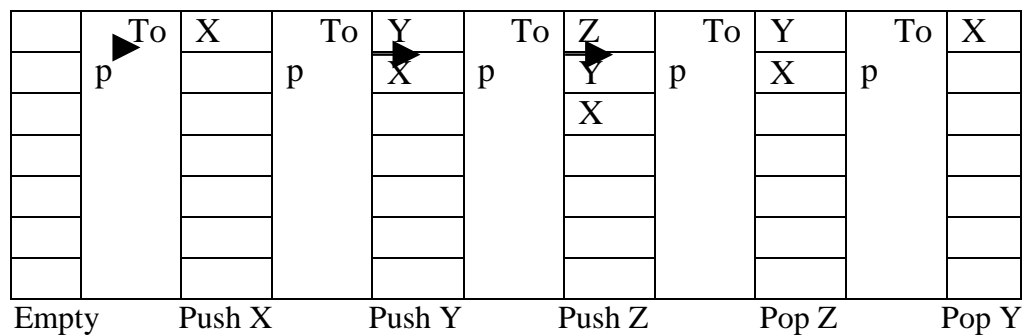
A *push* operation, in which a data item is placed at the location pointed to by the stack pointer and the address in the stack pointer is adjusted by the size of the data item; *Push* adds a given node to the top of the stack leaving previous nodes below.

A pop or *pull* operation, in which a data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item. *Pop* removes and returns the current top node of the stack.

The main primitives of a stack are known as:

Push adds a new node
Pop removes a node

Figure 1.2 shows the insertion of three data X, Y and Z to a stack and the removal of two data, Z and Y, from the stack.



Stack

Fig. 1. 2: Insertion and removal of data from stack

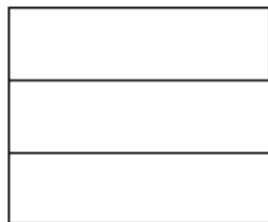
Additional primitives can be defined:

IsEmpty reports whether the stack is empty
IsFull reports whether the stack is full
Initialise creates/initialises the stack
Destroy deletes the contents of the stack
 (may be implemented by re-initialising the stack)

Initialise

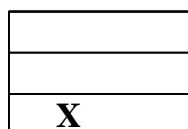
Creates the structure – i.e. ensures that the structure exists but contains no elements

e.g. **Initialise(S)** creates a new empty stack named S



S

e.g. **Push(X,S)** adds the value X to the Top of the stacks, S

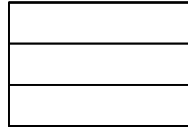


S

Fig 1.4: Stack after adding the value X

Pop

e.g. **Pop(S)** removes the TOP node and returns its value



S

Fig. 1.5: Stack after removing the top node

3.4 Stack Storage Modes

A stack can be stored in two ways:

a static data structure

OR

a dynamic data structure

3.4.1 Static Data Structures

These define collections of data which are fixed in size when the programme is compiled. An **array** is a static data structure.

3.4.2 Dynamic Data Structures

These define collections of data which are variable in size and structure. They are created as the programme executes, and grow and shrink to accommodate the data being stored.

SELF ASSESSMENT EXERCISE 1

A stack is referred to as a LIFO structure, true or false? Give reasons for your answer.

SELF ASSESSMENT EXERCISE 2

Write on two applications of stacks.

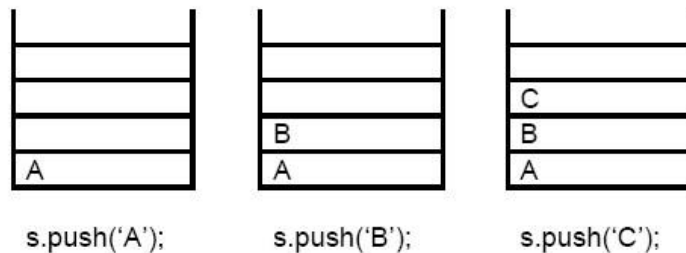
4.0 CONCLUSION

In this unit, you have learned about the stack data structure. You have also been able to understand the basic operations on a stack. You should also have learned about applications of stacks in computing.

5.0 SUMMARY

What you have learned in this unit concerns the stack data structure, their operations and applications. In the next unit, you shall learn about another linear data structure, known as Queues.

6.0 TUTOR-MARKED ASSIGNMENT



Applying the LIFO principle to the third stack S, what would be the state of the stack S, after the operation **S. POP ()** is executed? Illustrate this with a simple diagram.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A, (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 THE QUEUE DATA STRUCTURE

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 The Queue Data Structure
 - 3.2 Application of Queues
 - 3.3 Operations on a Queue
 - 3.3.1 Other Queue Operations
 - 3.4 Storing a Queue in a Static Data Structure
 - 3.5 Storing a Queue in a Dynamic Data Structure
 - 3.5.1 Adding a Node
 - 3.5.2 Removing a Node
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, the student will gain knowledge of the queue data structure as well as its applications and operations. Typical examples are given to facilitate your understanding of these concepts.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe a queue data structure
- give at least three applications of queues
- explain the operations on a queue
- describe two basic modes of queue storage.

3.0 MAIN CONTENT

3.1 The Queue Data Structure

The **queue** data structure is characterised by the fact that additions are made at the end, or tail, of the queue while removals are made from the front, or head, of the queue. For this reason, a queue is referred to as a FIFO structure (First-In First-Out). Figure 1.0 shows a queue of part of English alphabets.





Fig. 1.0: Example of a Queue

3.2 Application of Queues

Queues are very important structures in computer simulations, data processing, information management, and in operating systems. In simulations, queue structures are used to represent real-life events such as car queues at traffic light junctions and petrol filling stations, queues of people at the check-out point in super markets, queues of bank customers, etc.

In operating systems, queue structures are used to represent different programmes in the computer memory in the order in which they are executed. For example, if a programme, J is submitted before programme K, then programme J is queued before programme K in the computer memory and programme J is executed before programme K.

3.3 Operations on a Queue

The main primitive operations on a queue are known as:

Add adds a new node
Remove removes a node

Additional primitives can be defined thus:

IsEmpty reports whether the queue is empty
IsFull reports whether the queue is full
Initialise creates/initialises the queue
Destroy deletes the contents of the queue (may be implemented by re-initialising the queue)
Initialise

Creates the structure – i.e. ensures that the structure exists but contains no elements.

e.g. *Initialise(Q)* creates a new empty queue named Q

Add

e.g. **Add(X,Q)** adds the value X to the tail of Q

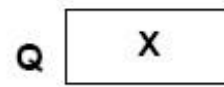


Fig. 1.1: Queue after adding the value X to the tail of Q

then, **Add (Y, Q)** adds the value Y to the tail of Q

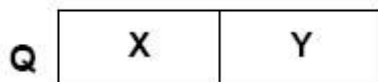


Fig. 1.2: Queue after adding the value Y to the tail of Q

Remove

e.g. **Remove(Q)** removes the head node and returns its value

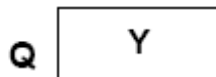


Fig. 1.3: Queue after removing Q from the head node

3.3.1 Other Queue Operations

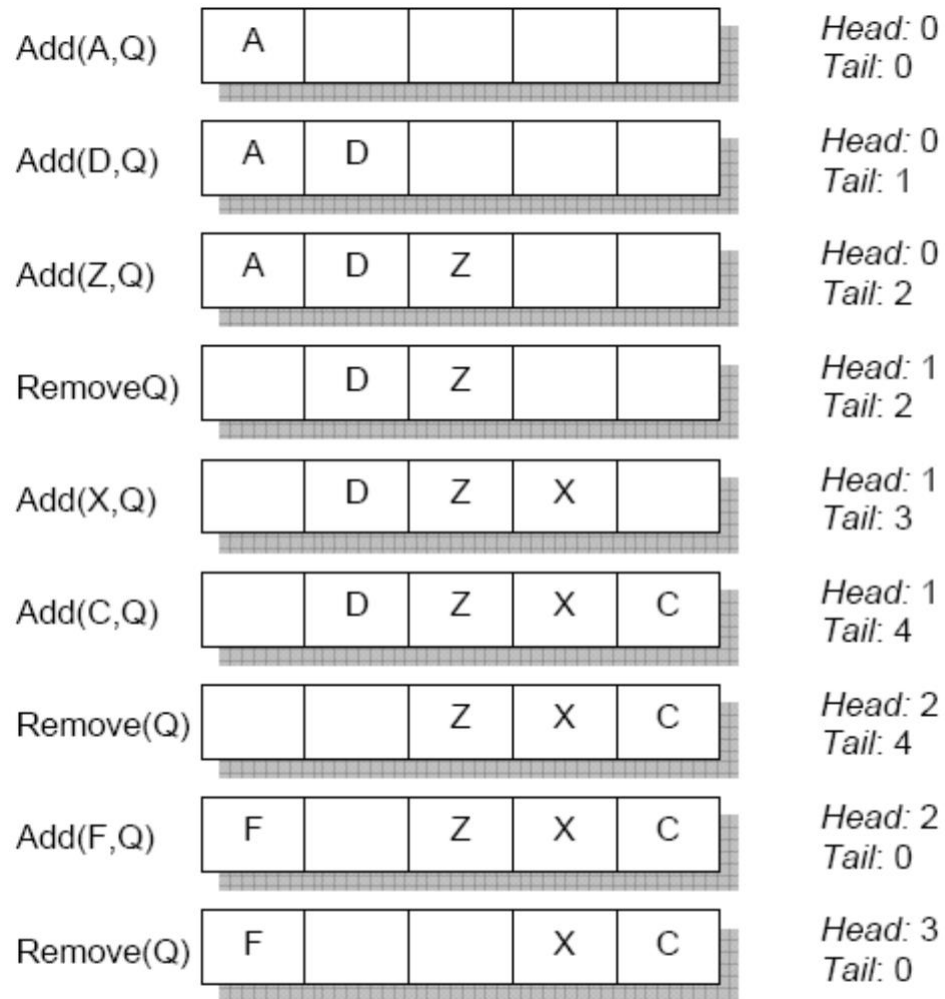
Action	Contents of queue Q after operation	Return value
Initialise (Q)	empty	
Add (A,Q)	A	-
Add (B,Q)	A B	-
Add(C,Q)	A B C	-
Remove (Q)	B C	A
Add (F,Q)	B C F	-
Remove (Q)	C F	B
Remove (Q)	F	C
Remove (Q)	empty	F

3.4 Storing a Queue in a Static Data Structure

This implementation stores the queue in an array. The array indices at which the head and tail of the queue are currently stored must be maintained. The head of the queue is not necessarily at index 0. The array can be a “circular array” in which the queue “wraps round” if the last index of the array is reached.

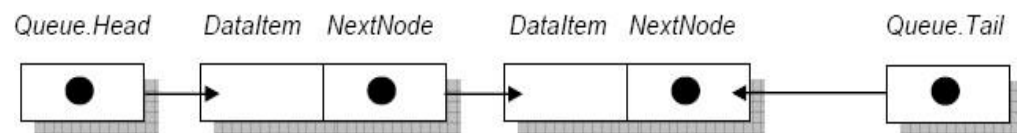
Figure 1.4 below is an example of storing a queue in an array of length

5:



3.5 Storing a Queue in a Dynamic Data Structure

A queue requires a reference to the head node AND a reference to the tail node. The following diagram describes the storage of a queue called Queue. Each node consists of data (DataItem) and a reference (NextNode).



The first node is accessed using the name **Queue.Head**.

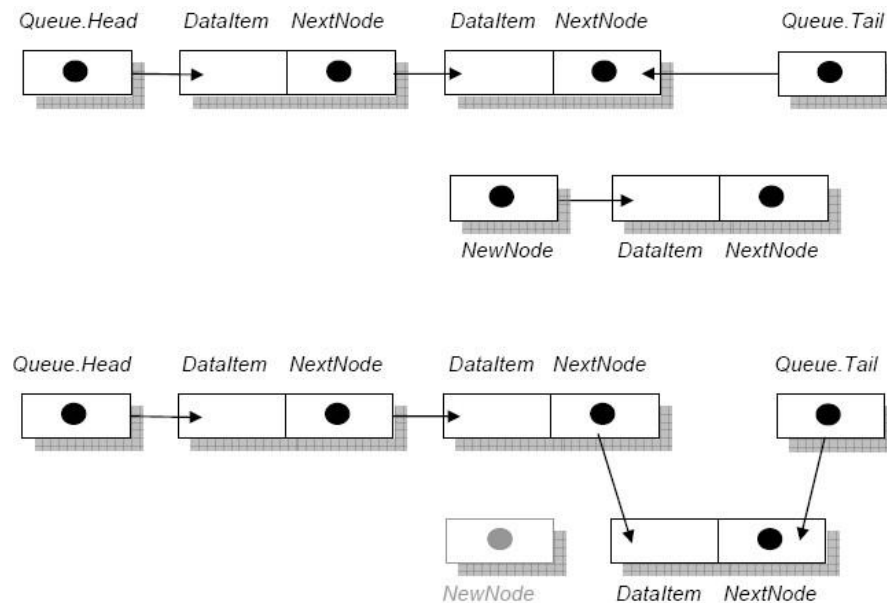
Its data is accessed using **Queue.Head.DataItem**

The second node is accessed using **Queue.Head.NextNode**

The last node is accessed using **Queue.Tail**

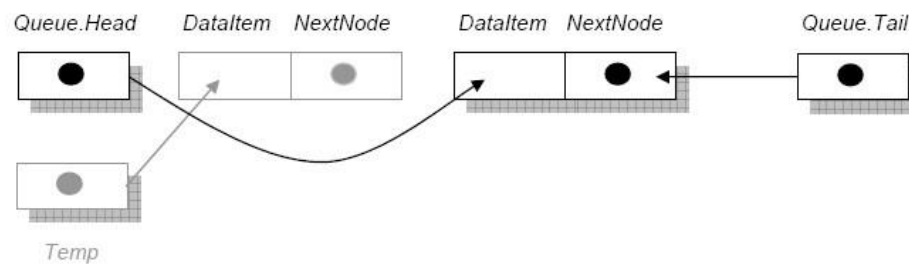
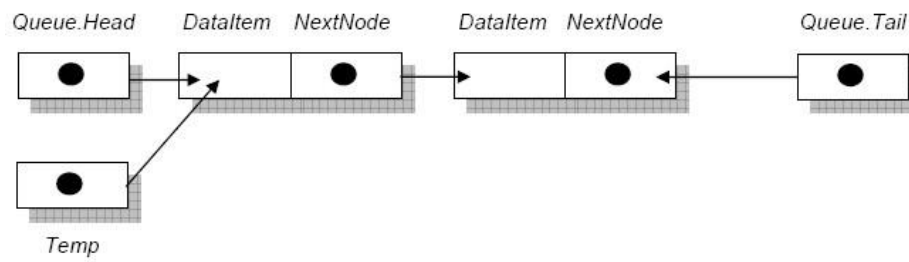
3.5.1 Adding a Node (Add)

The new node is to be added at the tail of the queue. The reference **Queue.Tail** should point to the new node, and the **NextNode** reference of the node previously at the tail of the queue should point to the **DataItem** of the new node.



3.5.2 Removing a Node (Remove)

The value of **Queue.Head.DataItem** is returned. A temporary reference **Temp**, is declared and set to point to head node in the queue (**Temp = Queue.Head**). **Queue.Head** is then set to point to the second node instead of the top node. The only reference to the original head node is now **Temp** and the memory used by this node can then be freed.



SELF ASSESSMENT EXERCISE 1

Why is a queue referred to as a FIFO structure?

SELF ASSESSMENT EXERCISE 2

Describe at least three applications of queues.

4.0 CONCLUSION

In this unit, you have learned about the queue data structure. Queue applications and operations were equally considered. You should also have learned about the queue storage in static and dynamic data structures.

5.0 SUMMARY

What you have learned in this unit concerns queues, their operations and applications. The units that follow shall build upon issues discussed in this unit.

6.0 TUTOR-MARKED ASSIGNMENT

Taking up again the example given in figure 1.4 above, show the state of the queue after the following operations:

Add (E,Q)

Remove (Q)

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

MODULE 2 HASHING AND TREES

Unit 1	Hashing
Unit 2	Trees
Unit 3	Search Trees
Unit 4	Garbage Collection and Other Heap
Unit 5	Memory Allocation

UNIT 1 HASHING**CONTENTS**

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	Hashing-The Basic Idea
3.2	Hash Keys and Functions
3.3	Hash Function Implementation
3.4	What is a Hash Table?
3.4.1	Abstract Hash Tables
3.5	Separate Chaining
3.6	Applications
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings
1.0	INTRODUCTION

In this unit, we will examine the basic idea of hashing. Hash keys and functions are equally described, giving the basic implementation of hash functions. We then define hash tables and give their applications.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the basic idea of hashing
- describe hash keys and functions
- give the basic implementation of hash functions
- define a hash table
- explain the applications of hash tables.

3.0 MAIN CONTENT

3.1 Hashing – The Basic Idea

Ideally we would build a data structure for which both the insertion and find operations are $O(1)$ in the worst case. However, this kind of performance can only be achieved with complete *a priori* knowledge. We need to know beforehand specifically which items are to be inserted into the container. Unfortunately, we do not have this information in the general case. So, if we cannot guarantee $O(1)$ performance in the *worst case*, then we make it our design objective to achieve $O(1)$ performance *in the average case*.

The constant time performance objective immediately leads us to the following conclusion: Our implementation must be based in some way on an array rather than a linked list. This is because we can access the k^{th} element of an array in constant time, whereas the same operation in a linked list takes $O(k)$ time.

In the previous chapter, we considered two searchable containers--the *ordered list* and the *sorted list*. In the case of an ordered list, the cost of an insertion is $O(1)$ and the cost of the find operation is $O(n)$. For a sorted list, the cost of insertion is $O(n)$ and the cost of the find operation is $O(\log n)$ for the array implementation.

Clearly, neither the ordered list nor the sorted list meets our performance objectives. The essential problem is that a search, either linear or binary, is always necessary. In the ordered list, the find operation uses a linear search to locate the item. In the sorted list, a binary search can be used to locate the item because the data is sorted. However, in order to keep the data sorted, insertion becomes $O(n)$.

In order to meet the performance objective of constant time insert and find operations, we need a way to do them *without performing a search*. That is, given an item x , we need to be able to determine directly from x the array position where it is to be stored.

Example

We wish to implement a searchable container which will be used to contain character strings from the set of strings K ,

$$K = \{\text{"ett"}, \text{"två"}^1, \text{"tre"}, \text{"fyra"}, \text{"fem"}, \text{"sex"}^2, \\ \text{"sju"}, \text{"åtta"}, \text{"nio"}, \text{"tio"}, \text{"elva"}, \text{"tolv"}\}.$$

Suppose we define a function $h : K \mapsto \mathbb{Z}$ as given by the following table:

x	$h(x)$
"ett"	1
"två"	2
"tre"	3
"fyra"	4
"fem"	5
"sex"	6
"sju"	7
"åtta"	8
"nio"	9
"tio"	10
"elva"	11
"tolv"	12

Table 1.0 Defining a Hash Function

Then, we can implement a searchable container using an array of length $n=12$. To insert item x , we simply store it at position $h(x)-1$ of the array. Similarly, to locate item x , we simply check to see if it is found at position $h(x)-1$. If the function $h(\cdot)$ can be evaluated in constant time, then both the insert and the find operations are $O(1)$.

We expect that any reasonable implementation of the function $h(\cdot)$ will run in constant time, since the size of the set of strings, K , is a constant! This example illustrates how we can achieve $O(1)$ performance in the worst case when we have complete, *a priori* knowledge.

3.2 Hash Keys and Functions

We are designing a container which will be used to hold some number of items of a given set, K . In this context, we call the elements of the set K *keys*. The general approach is to store the keys in an array. The position of a key in the array is given by a function $h(\cdot)$, called a *hash function*, which determines the position of a given key directly from that key.

In the general case, we expect the size of the set of keys, $|K|$, to be relatively large or even unbounded. For example, if the keys are 32-bit integers, then $|K| = 2^{32}$. Similarly, if the keys are arbitrary character strings of arbitrary length, then $|K|$ is unbounded.

On the other hand, we also expect the actual number of items stored in the container to be significantly less than $|K|$. That is, if n is the number of items actually stored in the container, then $n \ll |K|$. Therefore, it seems prudent to use an array of size M , where M is at least as great as the maximum number of items to be stored in the container.

Consequently, what we need is a function $h : K \mapsto \{0, 1, \dots, M - 1\}$. This function maps the set of values to be stored in the container to subscripts in an array of length M . This function is called a *hash function*.

In general, since $|K| \geq M$, the mapping defined by hash function will be a *many-to-one mapping*. That is, there will exist many pairs of distinct keys, x and y , such that $x \neq y$, for which $h(x) = h(y)$. This situation is called a *collision*. Several approaches for dealing with collisions are explored in the following sections.

What are the characteristics of a good hash function?

A good hash function avoids collisions.

A good hash function tends to spread keys evenly in the array.

A good hash function is easy to compute.

3.3 Hash Function Implementation

In reality, we cannot expect that the keys will always be integers. Depending on the application, the keys might be letters, character strings or even more complex data structures such as Associations or Containers.

In general, given a set of keys, K , and a positive constant, M , a hash function is a function of the form

$$h : K \mapsto \{0, 1, \dots, M - 1\}.$$

In practice, it is convenient to implement the hash function, h , as the composition of two functions, f and g . The function, f , maps keys into integers:

$$f : K \mapsto \mathbb{Z},$$

where \mathbb{Z} is the set of integers. The function, g , maps non-negative integers into $\{0, 1, \dots, M - 1\}$:

$$g : \mathbb{Z} \mapsto \{0, 1, \dots, M - 1\}.$$

Given appropriate functions, f and g , the hash function, h , is simply defined as the composition of those functions:

$$h = g \circ f$$

That is, the hash value of a key, x , is given by $g(f(x))$.

By decomposing the function, h , in this way, we can separate the problem into two parts: The first involves finding a suitable mapping from the set of keys, K , to the non-negative integers. The second involves mapping non-negative integers into the interval $[0, M-1]$. Ideally, the two problems would be unrelated. That is, the choice of the function, f , would not depend on the choice of g and *vice versa*. Unfortunately, this is not always the case. However, if we are careful, we can design the functions in such a way that $h = g \circ f$ is a good hash function.

This is precisely the domain of the function g . Consequently, we have already examined several different alternatives for the function, g . On the other hand, the choice of a suitable function for f depends on the characteristics of its domain.

In the following sections, we consider various different domains (sets of keys) and develop suitable hash functions for each of them. Each domain considered corresponds to a Java class. Recall that every Java class is ultimately derived from the `Object` class and that the `Object` class declares a method called `hashCode`:

```
public class Object
{
    public int hashCode ();
    // ...
}
```

The `hashCode` method corresponds to the function, f , which maps keys into integers.

3.4 What is a Hash Table?

A *hash table* is a searchable container. As such, its interface provides methods for putting an object into the container, finding an object in the container, and removing an object from the container. The `HashTable` interface extends the `SearchableContainerInterface` defined in programme 1.0 below. One additional method, called `getLoadFactor`, is declared.

```

1 public interface HashTable
2     extends SearchableContainer
3 {
4     double getLoadFactor ();
5 }

```

Programme 1.0: HashTable interface.

3.4.1 Abstract Hash Tables

As shown in Figure 1.0, we define an AbstractHashTable class from which several concrete realizations are derived.

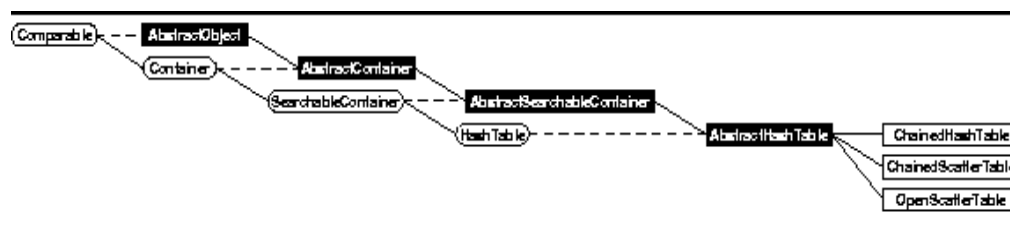


Fig. 1.0: Object class hierarchy

Programme 1.1 introduces the AbstractHashTable class. The AbstractHashTable class extends the AbstractSearchableContainer class introduced in Programme 1.1 and it implements the HashTable interface.

```

1 public abstract class AbstractHashTable
2     extends AbstractSearchableContainer
3     implements HashTable
4 {
5     public abstract int getLength ();
6
7     protected final int f (Object object)
8         { return object.hashCode (); }
9
10    protected final int g (int x)
11        { return Math.abs (x) % getLength (); }
12
13    protected final int h (Object object)
14        { return g(f(object)); }
15    // ...
16 }

```

Programme1.1: AbstractHashTable methods.

Programme 1.1 introduces four methods--`getLength`, `f`, `g`, and `h`. The `getLength` method is an abstract method. This function returns the *length* of a hash table.

The methods `f`, `g`, and `h` correspond to the composition $h = g \circ f$ discussed. The `f` method takes as an object and calls the `hashCode` method on that object to compute an integer. The `g` method uses the *division method* of hashing defined to map an integer into the interval $[0, M-1]$, where M is the length of the hash table. Finally, the `h` method computes the composition of `f` and `g`.

We will consider various ways of implementing hash tables. In all cases, the underlying implementation makes use of an array. The position of an object in the array is determined by hashing the object. The main problem to be resolved is how to deal with collisions--two different objects cannot occupy the same array position at the same time. In the following section, we consider an approach which solves the problem of collisions by keeping objects that collide in a linked list.

3.5 Separate Chaining

Figure 1.1, shows a hash table that uses *separate chaining* to resolve collisions. The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to one of the linked lists. The linked list to which it is appended is determined by hashing that item.

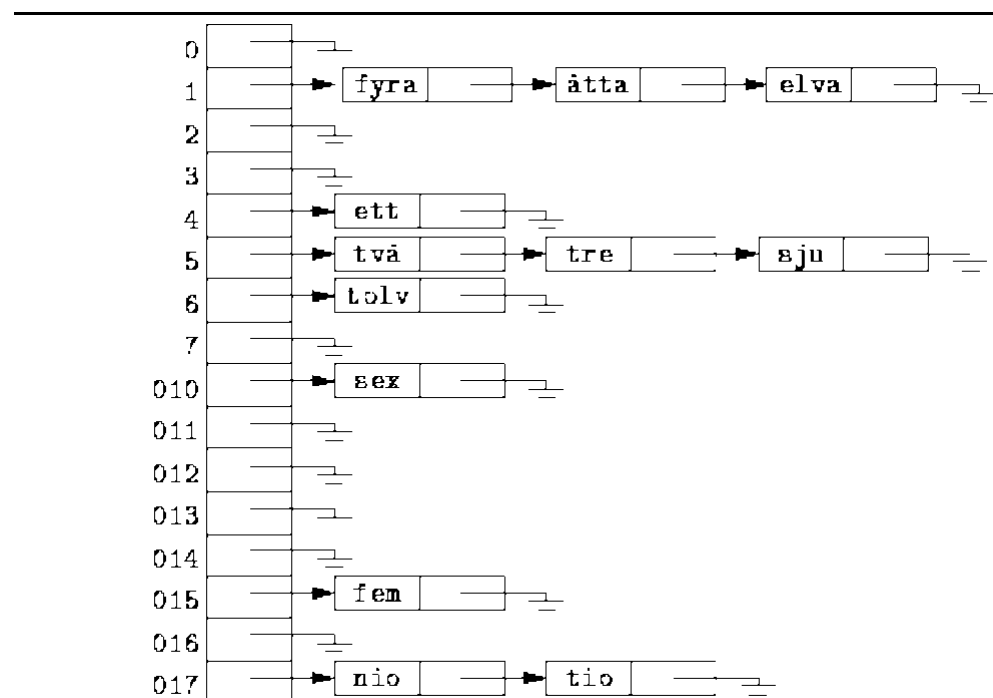


Figure 1.1: Hash table using separate chaining.

Figure 1.1 illustrates an example in which there are $M=16$ linked lists. The twelve character strings "ett"- "tolv" have been inserted into the table using the hashed values and in the order given in Table 1.1. Notice that in this example, since $M=16$, the linked list is selected by the least significant four bits of the hashed value given in Table 1.0. In effect, it is only the last letter of a string which determines the linked list in which that string appears.

3.6 Applications

Hash and Scatter tables have many applications. The principal characteristic of such applications is that keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, hash tables are often used to implement the *symbol table* of a programming language compiler. A symbol table is used to keep track of information associated with the symbols (variable and method names) used by a programmer. In this case, the keys are character strings and each key has, associated with it, some information about the symbol (e.g., type, address, value, lifetime, scope).

This section presents a simple application of hash and scatter tables. Suppose we are required to count the number of occurrences of each distinct word contained in a text file. We can do this easily using a hash or scatter table. Programme gives an implementation.

```

1  public class Algorithms
2  {
3      private static final class Counter
4          extends Int
5      {
6          Counter (int value)
7              { super (value); }
8          void increment ()
9              { ++value; }
10     }
11
12     public static void wordCounter (Reader in, PrintWriter out)
13         throws IOException
14     {
15         Hashtable table = new ChainedHashTable (1031);
16         StreamTokenizer tin = new StreamTokenizer (in);
17         while (tin.nextToken () != StreamTokenizer.TT_EOF)
18         {
19             String word = tin.sval;
20
21             Object obj = table.find (
22                 new Association (new Str (word)));
23
24             if (obj == null)
25                 table.insert (new Association (
26                     new Str (word), new Counter (1)));
27             else
28             {
29                 Association assoc = (Association) obj;
30                 Counter counter = (Counter) assoc.getValue ();
31                 counter.increment ();
32             }
33         }
34         out.println (table);
35     }
36 }

```

Programme 1.2: Hash/scatter table application--counting words.

The static inner class `Counter` extends the class `Int` defined in Section 1. In addition to the functionality inherited from the base class, the `Counter` class adds the method `increment` which increases the value by one.

The `wordCounter` method does the actual work of counting the words in the input file. The local variable `table` refers to a `ChainedHashTable` that is used to keep track of the words and counts. The objects which are put into the hash table are all instances of the class `Association`. Each association has as its key a `String` class instance, and as its value a `Counter` class instance.

The `wordCounter` method reads words from the input stream one at a time. As each word is read, a `find` operation is done on the hash table to determine if there is already an association for the given key. If none is found, a new association is created and inserted into the hash table. The given word is used as the key of the new association and the value is a counter which is initialised to one. On the other hand, if there is already an association for the given word in the hash table, the corresponding counter is incremented. When the `wordCounter` method reaches the end of the input stream, it simply prints the hash table on the given output stream.

The running time of the `wordCounter` method depends on a number of factors, including the number of different keys, the frequency of occurrence of each key, and the distribution of the keys in the overall space of keys. Of course, the hash/scatter table implementation chosen has an effect as does the size of the table used. For a reasonable set of keys we expect the hash function to do a good job of spreading the keys, uniformly in the table. Provided a sufficiently large table is used, the average search and insertion time is bounded by a constant. Under these ideal conditions, the running time should be $O(n)$, where n is the number of words in the input file.

SELF ASSESSMENT EXERCISE 1

What is a Hash table?

SELF ASSESSMENT EXERCISE 2

Describe at least one application of Hash Tables.

4.0 CONCLUSION

In this unit, you have learned about hashing, hash keys and functions. You have also been able to understand what hash tables are and how to implement hash functions. Finally, you have been able to appreciate the applications of hash tables.

5.0 SUMMARY

What you have learned borders on the basic notions of hashing, hash functions and hash tables and their applications.

6.0 TUTOR-MARKED ASSIGNMENT

What are the characteristics of a good hash function?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 2 TREES

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Trees
 - 3.2 Tree- Basics
 - 3.3 Binary Trees
 - 3.4 Tree Traversals
 - 3.4.1 Preorder Traversal
 - 3.4.2 Postorder Traversal
 - 3.4.3 Inorder Traversal
 - 3.5 Implementing Trees
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, we will consider different kinds of trees as well as different tree traversal algorithms. In addition, we show how trees can be used to represent arithmetic expressions and how we can evaluate an arithmetic expression by doing a tree traversal.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- give a basic definition of a tree
- describe binary trees
- explain tree traversals
- evaluate arithmetic expressions by means of tree traversals.

3.0 MAIN CONTENT

3.1 Trees

A tree is often used to represent a *hierarchy*. This is because the relationships between the items in the hierarchy suggest the branches of a botanical tree. For example, a tree-like *organisation chart* is often used to represent the lines of responsibility in a business as shown in

Figure 1.0. The president of the company is shown at the top of the tree and the vice-presidents are indicated below her. Under the vice-presidents, we find the managers and below the managers the rest of the clerks. Each clerk reports to a manager, each manager reports to a vice-president, and each vice-president reports to the president.

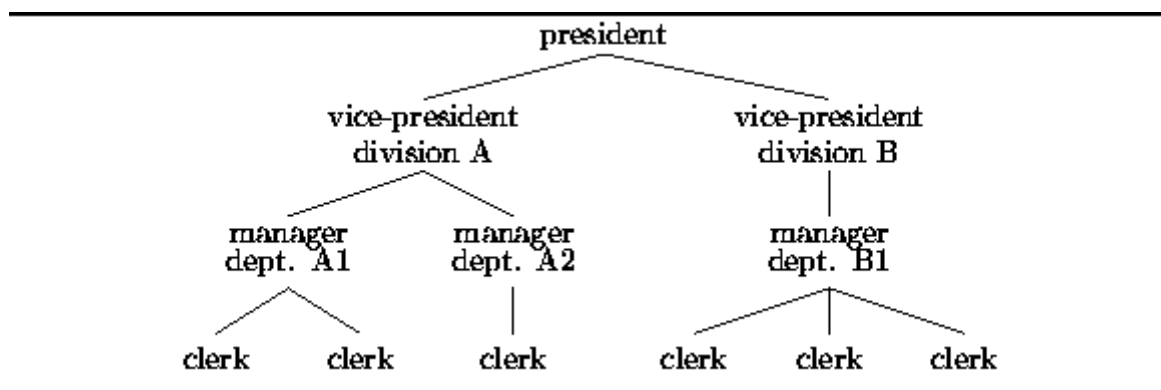


Fig. 1.0: Representing a hierarchy using a tree

It just takes a little imagination to see the tree in Figure 1.0. Of course, the tree is upside-down. However, this is the usual way the data structure is drawn. The president is called the *root* of the tree and the clerks are the *leaves*.

A tree is extremely useful for certain kinds of computations. For example, suppose we wish to determine the total salaries paid to employees by division or by department. The total of the salaries in division A can be found by computing the sum of the salaries paid in departments A1 and A2 plus the salary of the vice-president of division A. Similarly, the total of the salaries paid in department A1 is the sum of the salaries of the manager of department A1 and of the two clerks below her.

Clearly, in order to compute all the totals, it is necessary to consider the salary of every employee. Therefore, an implementation of this computation must *visit* all the employees in the tree. An algorithm that systematically *visits* all the items in a tree is called a *tree traversal*.

3.2 Tree- Basics

The following is a mathematical definition of a tree:

Definition (Tree) A *tree*, T , is a finite, non-empty set

$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n,$$
of *nodes*, with the following properties:

1. A designated node of the set, r , is called the *root* of the tree; and
2. The remaining nodes are partitioned into $n \geq 0$ subsets, T_1, T_2, \dots, T_n , each of which is a tree.

For convenience, we shall use the notation $T = \{r, T_1, T_2, \dots, T_n\}$ to denote the tree T .

Notice that this definition is *recursive*—that is, a tree is defined in terms of itself! Fortunately, we do not have a problem with infinite recursion because every tree has a *finite* number of nodes and because in the base case, a tree has $n=0$ subtrees.

It follows from the definition that the minimal tree is a tree comprising a single root node. For example $T_a = \{A\}$ is such a tree. When there is more than one node, the remaining nodes are partitioned into subtrees. For example, the $T_b = \{B, \{C\}\}$ is a tree which comprises of the root node, B , and the subtree $\{C\}$. Finally, the following is also a tree

$$T_c = \{D, \{E, \{F\}\}, \{G, \{H, \{I\}\}, \{J, \{K\}, \{L\}\}, \{M\}\}. \quad (9.1)$$

How do T_a , T_b , and T_c resemble their arboreal namesake? The similarity becomes apparent when we consider the graphical representation of these trees shown in Figure 1.1. To draw such a pictorial representation of a tree, $T = \{r, T_1, T_2, \dots, T_n\}$, the following recursive procedure is used: First, we first draw the root node, r . Then, we draw each of the subtrees, T_1, T_2, \dots, T_n , beside each other below the root. Finally, lines are drawn from r to the roots of each of the subtrees.

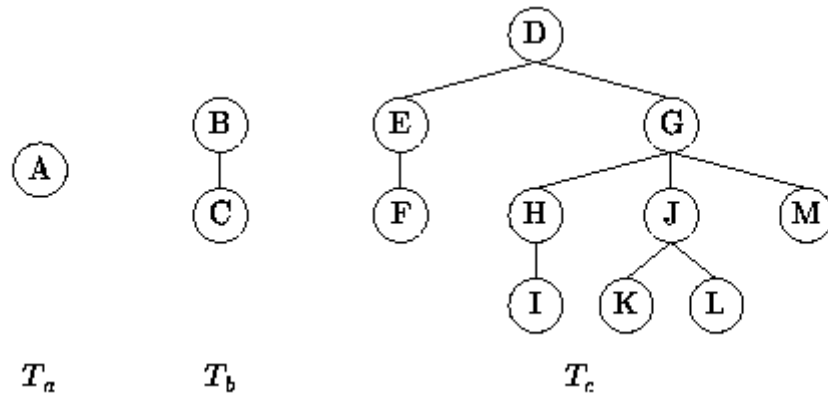


Fig. 1.1: Examples of trees.

Of course, trees drawn in this fashion are upside down. Nevertheless, this is the conventional way in which tree data structures are drawn. In fact, it is understood that when we speak of "up" and "down," we do so

with respect to this pictorial representation. For example, when we move from a root to a subtree, we will say that we are moving *down* the tree. The inverted pictorial representation of trees is probably due to the way that genealogical *lineal charts* are drawn. A *lineal chart* is a family tree that shows the descendants of some person. And it is from genealogy that much of the terminology associated with tree data structures is taken.

Terminology

Consider a tree $T = \{r, T_1, T_2, \dots, T_n\}$, $n \geq 0$, as given by the definition: The

degree of a node is the number of subtrees associated with that node. For example, the degree of tree T is n .

A node of degree zero has no subtrees. Such a node is called a *leaf*.

Each root r of subtree T_i of tree T is called a *child* of r . The term *grandchild* is defined in a similar manner.

The root node, r , of tree T , is the *parent* of all the roots r_i of the subtrees T_i , $1 \leq i \leq n$. The term, *grandparent*, is defined in a similar manner.

Two roots r_i and r_j of distinct subtrees T_i and T_j of tree T are called *siblings*.

There is still more terminology to be introduced, but in order to do that, we need the following definition:

Definition (Path and Path Length) Given a tree, T , containing the set of nodes R , a *path* in T is defined as a non-empty sequence of nodes

$P = \{r_1, r_2, \dots, r_k\}$,
where $r_i \in R$, $1 \leq i \leq k$, such that the i^{th} node in the sequence, r_i , is the $(i+1)^{\text{th}}$ node in the sequence r_{i+1} . The *length* of path P is $k-1$.

For example, consider again the tree T_c shown in Figure 1.1. This tree contains many different paths. In fact, if you count carefully, you should find that there are exactly 29 distinct paths in tree T_c . This includes the path of length zero, $\{D\}$; the path of length one, $\{E, F\}$; and the path of length three, $\{D, G, J, K\}$.

3.3 Binary Trees

In this section, we will consider an extremely important and useful category of tree structure--*binary trees*. A **binary tree** is an N -ary tree

for which N is two. Since a binary tree is an N -ary tree, all of the results derived in the preceding section apply to binary trees. However, binary trees have some interesting characteristics that arise from the restriction that N is two. For example, there is an interesting relationship between binary trees and the binary number system. Binary trees are also very useful for the representation of mathematical expressions involving the binary operations such as addition and multiplication.

Binary trees are defined as follows:

Definition (Binary Tree) A *binary tree*, T , is a finite set of *nodes* with the following properties:

1. Either the set is empty, $T = \emptyset$; or
2. The set consists of a root, r , and exactly two distinct binary trees T_L and T_R , $T = \{r, T_L, T_R\}$.

The tree, T_L , is called the *left subtree* of T , and the tree, T_R , is called the *right subtree* of T .

Binary trees are almost always considered to be *ordered trees*. Therefore, the two subtrees T_L and T_R are called the *left* and *right* subtrees, respectively. Consider the two binary trees shown in Figure 1.2. Both trees have a root with a single non-empty subtree. However, in one case, it is the left subtree which is non-empty; in the other case, it is the right subtree that is non-empty. Since the order of the subtrees matters, the two binary trees shown in Figure 1.2 are different.

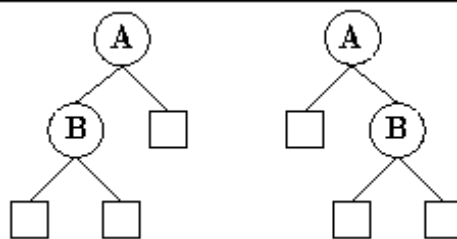


Fig. 1.2: Two distinct binary trees

We can determine some of the characteristics of binary trees from the theorems given in the preceding section by letting $N=2$. For example, we know that a binary tree with $n \geq 0$ internal nodes contains $n+1$ external node. This result is true regardless of the shape of the tree. Consequently, we expect that the storage overhead associated with the empty trees will be $O(n)$.

We thus learn that a binary tree of height $h \geq 0$ has at most $2^{h+1} - 1$ internal nodes. Conversely, the height of a binary tree with n internal nodes is at least $\lceil \log_2 n + 1 \rceil - 1$. That is, the height of a binary tree with n nodes is $\Omega(\log n)$.

Finally, a binary tree of height $h \geq 0$ has at most 2^h leaves. Conversely, the height of a binary tree with l leaves is at least $\lceil \log_2 l \rceil$. Thus, the height of a binary tree with l leaves is $\Omega(\log l)$.

3.4 Tree Traversals

There are many different applications of trees. As a result, there are many different algorithms for manipulating them. However, many of the different tree algorithms have in common the characteristic that they systematically visit all the nodes in the tree. That is, the algorithm walks through the tree data structure and performs some computation at each node in the tree. This process of walking through the tree is called a *tree traversal*.

There are essentially two different methods in which to visit systematically all the nodes of a tree--*depth-first traversal* and *breadth-first traversal*. Certain depth-first traversal methods occur frequently enough that they are given names of their own: *preorder traversal*, *inorder traversal* and *postorder traversal*.

The discussion that follows uses the tree in Figure 1.3 as an example. The tree shown in the figure is a general tree:

$$T = \{A, \{B, \{C\}\}, \{D, \{E, \{F\}, \{G\}\}, \{H, \{I\}\}\} \} \quad (9.3)$$

However, we can also consider the tree in Figure 1.3 to be an N -ary tree (specifically, a binary tree if we assume the existence of empty trees at the appropriate positions:

$$T = \{A, \{B, \emptyset, \{C, \emptyset, \emptyset\}\}, \{D, \{E, \{F, \emptyset, \emptyset\}, \{G, \emptyset, \emptyset\}\}, \{H, \{I, \emptyset, \emptyset\}, \emptyset\}\}$$

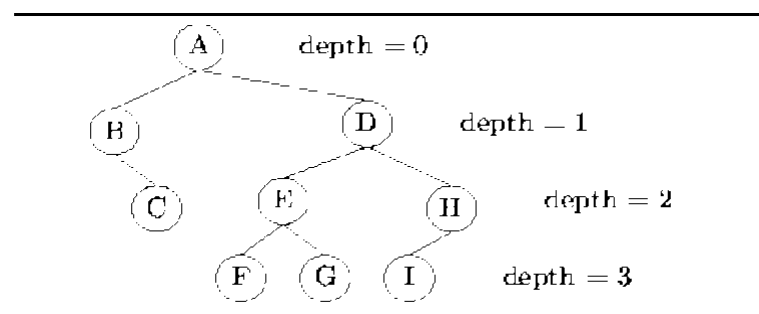


Fig. 1.3: Sample tree

3.4.1 Preorder Traversal

The first depth-first traversal method we consider is called *preorder traversal*. Preorder traversal is defined recursively as follows: To do a preorder traversal of a general tree:

1. Visit the root first; and then
2. Do a preorder traversal each of the subtrees of the root one-by-one in the order given.

Preorder traversal gets its name from the fact that it visits the root first. In the case of a binary tree, the algorithm becomes:

1. Visit the root first; and then
2. Traverse the left subtree; and then
3. Traverse the right subtree.

For example, a preorder traversal of the tree visits the nodes in the following order:

A, B, C, D, E, F, G, H, I.

Notice that the preorder traversal visits the nodes of the tree in precisely the same order in which they are written. A preorder traversal is often done when it is necessary to print a textual representation of a tree.


3.4.2 Postorder Traversal

The second depth-first traversal method we consider is *postorder traversal*. In contrast with preorder traversal, which visits the root first, postorder traversal visits the root last. To do a postorder traversal of a general tree:

1. Do a postorder traversal each of the subtrees of the root one-by-one in the order given; and then
2. Visit the root.

To do a postorder traversal of a binary tree

1. Traverse the left subtree; and then
2. Traverse the right subtree; and then
3. Visit the root.

A postorder traversal of the tree shown in Figure  visits the nodes in the following order:

C, B, F, G, E, I, H, D, A.

3.4.3 Inorder Traversal

The third depth-first traversal method is *inorder traversal*. Inorder traversal only makes sense for binary trees. Whereas preorder traversal visits the root first and postorder traversal visits the root last, inorder traversal visits the root *in between* visiting the left and right subtrees:

1. Traverse the left subtree; and then
2. Visit the root; and then
3. Traverse the right subtree.

An inorder traversal of the tree visits the nodes in the following order:

B, C, A, F, E, G, D, I, H.

3.5 Implementing Trees

In this section, we will consider the implementation of trees including general trees, *N*-ary trees, and binary trees. The implementations presented have been developed in the context of the abstract data type framework. That is, the various types of trees are viewed as classes of *containers* as shown in Figure 1.4.

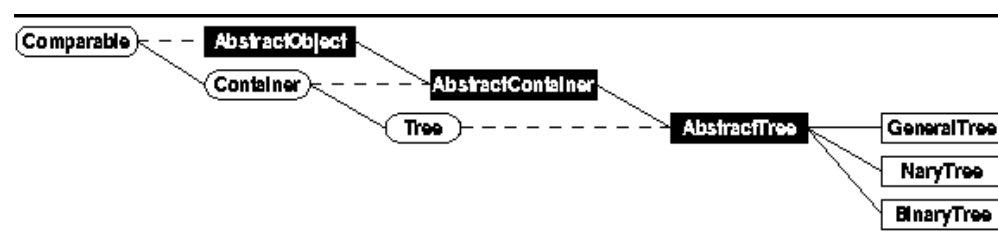


Fig. 1.4: Object class hierarchy

Programme 1.0 defines the `Tree` interface. The `Tree` interface extends the `Container` interface defined in Programme 1.0.

```

1  public interface Tree
2      extends Container
3  {
4      Object getKey ();
5      Tree getSubtree (int i);
6      boolean isEmpty ();
7      boolean isLeaf ();
8      int getDegree ();
9      int getHeight ();
10     void depthFirstTraversal (PrePostVisitor visitor);
11     void breadthFirstTraversal (Visitor visitor);
12 }
  
```

Programme 1.0: `Tree` interface.

The `Tree` interface adds the following methods to those inherited from the `Container` interface:

`getKey`

This method returns the object contained in the root node of a tree.

`getSubtree`

This method returns the i^{th} subtree of the given tree.

`isEmpty`

This boolean-valued method returns `true` if the root of the tree is an empty tree, i.e., an external node.

`isLeaf`

This boolean-valued method returns `true` if the root of the tree is a leaf node.

`getDegree`

This method returns the degree of the root node of the tree. By definition, the degree of an external node is zero.

`getHeight`

This method returns the height of the tree. By definition, the height of an empty tree is -1.

`depthFirstTraversal` and `breadthFirstTraversal`

These methods are like the `accept` method of the container class (see Section 4.1). Both of these methods perform a traversal. That is, all the nodes of the tree are visited systematically. The former takes a `PrePostVisitor` and the latter takes a `Visitor`. When a node is visited, the appropriate methods of the visitor are applied to that node.

SELF ASSESSMENT EXERCISE 1

What do you understand by the term tree traversals?

SELF ASSESSMENT EXERCISE 2

Give a brief description of the implementation of trees.

4.0 CONCLUSION

In this unit, you have learned about trees. You have also learned about binary trees and tree traversals. Finally, you have been able to learn how to implement trees.

5.0 SUMMARY

What you have learned in this unit is focused on trees, the common types and implementation of trees.

6.0 TUTOR-MARKED ASSIGNMENT

Describe trees and illustrate further by means of a diagram.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 3 SEARCH TREES**CONTENTS**

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Search Tree-Basics
 - 3.2 Searching a Search Tree
 - 3.2.1 Searching an M-way Tree
 - 3.2.2 Searching a Binary Tree
 - 3.3 Successful Search
 - 3.4 Unsuccessful Search
 - 3.5 AVL Search Trees
 - 3.6 Implementing AVL Trees
 - 3.6.1 Inserting Items into AVL Trees
 - 3.6.2 Removing Items from an AVL Tree
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit introduces Search Trees, describing successful and unsuccessful searching. In addition, we show the implementation of AVL search trees.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain what a search tree is
- describe a successful search
- describe an unsuccessful search
- explain the implementation of AVL search trees.

3.0 MAIN CONTENT**3.1 Search Tree-Basics**

A tree which supports efficient search, insertion, and withdrawal operations is called a *search tree*. In this context, the tree is used to store a finite set of keys drawn from a totally ordered set of keys, K . Each node of the tree contains one or more keys and all the keys in the tree are unique, i.e., no duplicate keys are permitted. What makes a tree into

a search tree is that the keys do not appear in arbitrary nodes of the tree. Instead, there is a *data ordering criterion* which determines where a given key may appear in the tree in relation to the other keys in that tree. The subsequent sections present two related types of search trees, *M*-way search trees and binary search trees.

3.2 Searching a Search Tree

The main advantage of a search tree is that the data ordering criterion ensures that it is not necessary to do a complete tree traversal in order to locate a given item. Since search trees are defined recursively, it is easy to define a recursive search method.

3.2.1 Searching an *M*-way Tree

Consider the search for a particular item, say x , in an *M*-way search tree. The search always begins at the root. If the tree is empty, the search fails. Otherwise, the keys contained in the root node are examined to determine if the object of the search is present. If it is, the search terminates successfully. If it is not, there are three possibilities: Either the object of the search, x , is less than k_1 , in which case subtree T_0 is searched; or x is greater than k_{n-1} , in which case subtree T_{n-1} is searched; or there exists an i such that $1 \leq i < n-1$ for which $k_i < x < k_{i+1}$, in which case subtree T_i is searched.

Notice that when x is not found in a given node, only one of the n subtrees of that node is searched. Therefore, a complete tree traversal is not required. A successful search begins at the root and traces a downward path in the tree, which terminates at the node containing the object of the search. Clearly, the running time of a successful search is determined by the *depth* in the tree of object of the search.

When the object of the search is not in the search tree, the search method described above traces a downward path from the root which terminates when an empty subtree is encountered. In the worst case, the search path passes through the deepest leaf node. Therefore, the worst-case running time for an unsuccessful search is determined by the *height* of the search tree.

3.2.2 Searching a Binary Tree

The search method described above applies directly to binary search trees. As above, the search begins at the root node of the tree. If the object of the search, x , matches the root r , the search terminates successfully. If it does not, then if x is less than r , the left subtree is

searched; otherwise x must be greater than r , in which case the right subtree is searched.

Figure 1.0 shows two binary search trees. The tree T_a is an example of a particularly bad search tree because it is not really very tree-like at all. In fact, it is topologically isomorphic with a linear, linked list. In the worst case, a tree which contains n items has height $O(n)$. Therefore, in the worst case an unsuccessful search must visit $O(n)$ internal nodes.

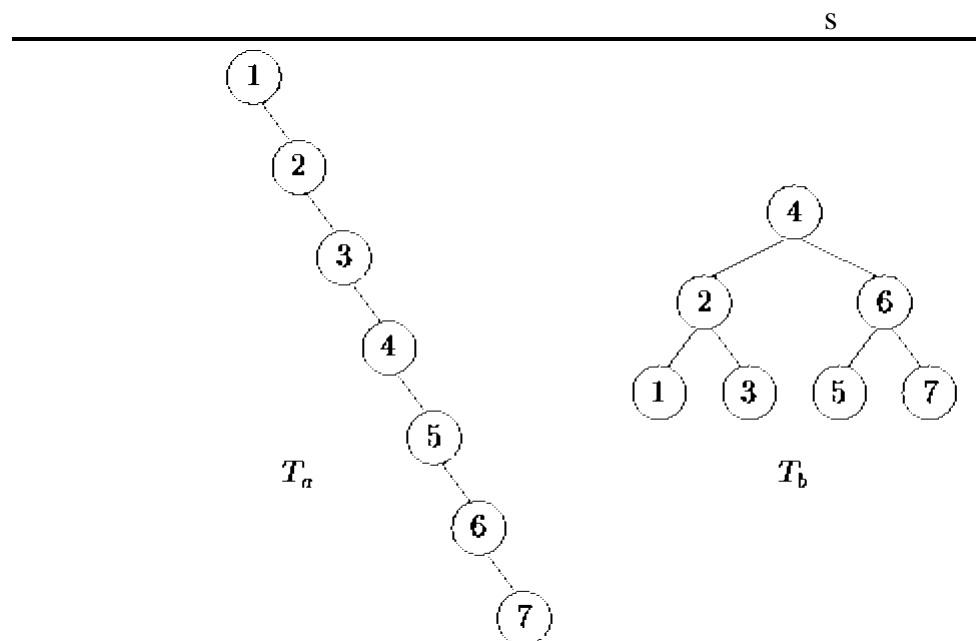


Figure 1.0: Examples of search trees

On the other hand, tree T_b in Figure 1.0 is an example of a particularly good binary search tree. This tree is an instance of a *perfect binary tree*. **Definition (Perfect Binary Tree)** A *perfect binary tree* of height $h \geq 0$ is a binary tree $T = \{r, T_L, T_R\}$ with the following properties:

1. If $h=0$, then $T_L = \emptyset$ and $T_R = \emptyset$.
2. Otherwise, $h > 0$, in which case both T_L and T_R are both perfect binary trees of height $h-1$.

It is fairly easy to show that a perfect binary tree of height h , has exactly $2^{h+1} - 1$ internal nodes. Conversely, the height of a perfect binary tree with n internal nodes is $\log_2(n+1)$. If we have a search tree that has the shape of a perfect binary tree, then every unsuccessful search visits $h+1 = \log_2(n+1)$

$$O(\log n)$$

exactly $h+1$ internal nodes, where h is the height of the tree. Thus, the worst case for unsuccessful search in a perfect tree is $h+1$.

3.3 Successful Search

When a search is successful, exactly $d+1$ internal nodes are visited, where d is the depth in the tree of object of the search. For example, if the object of the search is at the root which has depth zero, the search visits just one node--the root itself. Similarly, if the object of the search is at depth one, two nodes are visited, and so on. We shall assume that it is equally likely for the object of the search to appear in any node of the search tree. In that case, the *average* number of nodes visited during a successful search is $\bar{d} + 1$, where \bar{d} is the average of the depths of the

nodes in a given tree. That is, given a binary search tree with $n > 0$ nodes,

$$\bar{d} = \frac{1}{n} \sum_{i=1}^n d_i,$$

where d_i is the depth of the i^{th} node of the tree.

The quantity $\sum_{i=1}^n d_i$ is called the *internal path length*. The internal path length of a tree is simply the sum of the depths (levels) of all the internal nodes in the tree. Clearly, the average depth of an internal node is equal to the internal path length divided by n , the number of nodes in the tree. Unfortunately, for any given number of nodes n , there are many different possible search trees. Furthermore, the internal path lengths of the various possibilities are not equal. Therefore, to compute the average depth of a node in a tree with n nodes, we must consider all possible trees with n nodes. In the absence of any contrary information, we shall assume that all trees having n nodes are equiprobable and then compute the average depth of a node in the average tree containing n nodes.

Let $I(n)$ be the average internal path length of a tree containing n nodes. Consider first the case of $n=1$. Clearly, there is only one binary tree that contains one node--the tree of height zero. Therefore, $I(1)=0$.

Now consider an arbitrary tree, $T_n(l)$, having $n \geq 1$ internal nodes altogether, l of which are found in its left subtree, where $0 \leq l < n$. Such a tree consists of a root, the left subtree with l internal nodes and a right subtree with $n-l-1$ internal nodes. The average internal path length for such a tree is the sum of the average internal path length of the left subtree, $I(l)$, plus that of the right subtree, $I(n-l-1)$, plus $n-1$ because the nodes in the two subtrees are one level lower in $T_n(l)$.

In order to determine the average internal path length for a tree with n nodes, we must compute the average of the internal path lengths of the trees $T_n(l)$ average over all possible sizes, l , of the (left) subtree, $0 \leq l < n$.

To do this we consider an ordered set of n distinct keys, $k_0 < k_1 < \dots < k_{n-1}$. If we select the key, k_l , to be the root of a binary search tree, then there are l keys, k_0, \dots, k_{l-1} , in its left subtree and $n-l-1$ keys, $k_{l+1}, k_{l+2}, \dots, k_{n-1}$, in its right subtree.

If we assume that it is equally likely for any of the n keys to be selected as the root, then all the subtree sizes in the range $0 \leq l < n$ are equally

$$\begin{aligned} I(n) &= \frac{1}{n} \sum_{i=0}^{n-1} (I(i) + I(n-i-1) + n-1), \quad n > 1 \\ &= \frac{2}{n} \sum_{i=0}^{n-1} I(i) + n-1. \end{aligned}$$

likely. Therefore, the average internal path length for a tree with $n \geq 1$ nodes is

Thus, in order to determine $I(n)$, we need to solve the recurrence

$$I(n) = \begin{cases} 0 & n = 1, \\ \frac{2}{n} \sum_{i=0}^{n-1} I(i) + n - 1 & n > 1. \end{cases} \quad (10.1)$$

To solve this recurrence, we consider the case $n > 1$ and then multiply Equation 10.1 by n to get

$$nI(n) = 2 \sum_{i=0}^{n-1} I(i) + n^2 - n. \quad (10.2)$$

Since this equation is valid for any $n > 1$, by substituting $n-1$ for n , we can also write

$$(n-1)I(n-1) = 2 \sum_{i=0}^{n-2} I(i) + n^2 - 3n + 2, \quad (10.3)$$

which is valid for $n > 2$. Subtracting Equation 10.3 from Equation 10.2 gives

$$nI(n) - (n-1)I(n-1) = 2I(n-1) + 2n - 2,$$

which can be rewritten as:

$$I(n) = \frac{(n+1)I(n-1) + 2n - 2}{n}. \quad (10.4)$$

Thus, we have shown the solution to the recurrence in the equation is the same as the solution of the recurrence

$$I(n) = \begin{cases} 0 & n = 1, \\ 1 & n = 2, \\ ((n+1)I(n-1) + 2n - 2)/n & n > 2. \end{cases} \quad (10.5)$$

3.4 Unsuccessful Search

All successful searches terminate when the object of the search is found. Therefore, all successful searches terminate at an internal node. In contrast, all unsuccessful searches terminate at an external node. In terms of the binary tree shown in Figure 1.0, a successful search terminates in one of the nodes which are drawn as circles and an unsuccessful search terminates in one of the boxes.

The preceding analysis shows that the average number of nodes visited during a successful search depends on the *internal path length*, which is simply the sum of the depths of all the internal nodes. Similarly, the average number of nodes visited during an unsuccessful search depends on the *external path length*, which is the sum of the depths of all the external nodes. Fortunately, there is a simple relationship between the internal path length and the external path length of a binary tree.

Theorem: Consider a binary tree T with n internal nodes and an internal path length of I . The external path length of T is given by

$$E = I + 2n.$$

In other words, Theorem \square says that the *difference* between the internal path length and the external path length of a binary tree with n internal nodes is $E - I = 2n$.

extbfProof (By induction).

Base Case: Consider a binary tree with one internal node and internal path length of zero. Such a tree has exactly two empty subtrees immediately below the root and its external path length is two. Therefore, the theorem holds for $n=1$.

Inductive Hypothesis: Assume that the theorem holds for $n = 1, 2, 3, \dots, k$, for some $k \geq 1$. Consider an arbitrary tree, T_k , that has k internal nodes. According to Theorem \square , T_k has $k+1$ external nodes. Let I_k and E_k be the internal and external path length of T_k , respectively. According to the inductive hypothesis, $E_k - I_k = 2k$.

Consider what happens when we create a new tree T_{k+1} by removing an external node from T_k and replacing it with an internal node that has two empty subtrees. Clearly, the resulting tree has $k+1$ internal nodes. Furthermore, suppose the external node we remove is at depth d . Then

the internal path length of T_{k+1} is $I_{k+1} = I_k + d$ and the external path length of T_{k+1} is $E_{k+1} = E_k - d + 2(d+1) = E_k + d + 2$.

The difference between the internal path length and the external path length of T_{k+1} is

$$\begin{aligned} E_{k+1} - I_{k+1} &= (E_k + d + 2) - (I_k + d) \\ &= E_k - I_k + 2 \\ &= 2(k+1). \end{aligned}$$

Therefore, by induction on k , the difference between the internal path length and the external path length of a binary tree with n internal nodes is $2n$ for all $n \geq 1$.

Since the difference between the internal and external path lengths of any tree with n internal nodes is $2n$, then we can say the same thing about the *average* internal and external path lengths average over all search trees. Therefore, $E(n)$, the average external path length of a binary search tree is given by

$$\begin{aligned} E(n) &= I(n) + 2n \\ &= 2(n+1)H_n - 2n \\ &\approx 2(n+1)(\ln n + \gamma) - 2n. \end{aligned}$$

A binary search tree with internal n nodes has $n+1$ external nodes. Thus, the average depth of an external node of a binary search tree with n internal nodes, \bar{e} , is given by

$$\begin{aligned} \bar{e} &= E(n)/(n+1) \\ &= 2H_n - 2n/(n+1) \\ &\approx 2(\ln n + \gamma) - 2n/(n+1) \\ &= O(\log n). \end{aligned}$$

These very nice results are the *raison d'être* for binary search trees. What they say is that the average number of nodes visited during either a successful or an unsuccessful search in the average binary search tree having n nodes is $O(\log n)$. We must remember, however, that these results are premised on the assumption that all possible search trees of n nodes are equiprobable. It is important to be aware that in practice, this may not always be the case.

3.5 AVL Search Trees

The problem with binary search trees is that while the average running times for search, insertion, and withdrawal operations are all $O(\log n)$,

any one operation is still $O(n)$ in the worst case. This is so because we cannot say anything in general about the shape of the tree.

For example, consider the two binary search trees shown in Figure 1. Both trees contain the same set of keys. The tree, T_a , is obtained by starting with an empty tree and inserting the keys in the following order

1, 2, 3, 4, 5, 6, 7.

The tree T_b is obtained by starting with an empty tree and inserting the keys in this order

4, 2, 6, 1, 3, 5, 7.

Clearly, T_b is a better search tree than T_a . In fact, since T_b is a *perfect binary tree*, its height is $\log_2(n+1) - 1$. Therefore, all three operations, search, insertion, and withdrawal, have the same worst case asymptotic running time $O(\log n)$.

The reason that T_b is better than T_a is that it is the more *balanced* tree. If we could ensure that the search trees we construct are balanced, then the worst-case running time of search, insertion, and withdrawal, could be made logarithmic rather than linear. But under what conditions is a tree *balanced*?

If we say that a binary tree is balanced if the left and right subtrees of every node have the same height, then the only trees which are balanced are the perfect binary trees. A perfect binary tree of height h , has exactly $2^{h+1} - 1$ internal nodes. Therefore, it is only possible to create perfect trees with n nodes for $n = 1, 3, 7, 15, 31, 63, \dots$. Clearly, this is an unsuitable balance condition because it is not possible to create a balanced tree for every n .

What are the characteristics of a good *balance condition*?

1. A good balance condition ensures that the height of a tree with n nodes is $O(\log n)$.
2. A good balance condition can be maintained efficiently. That is, the additional work necessary to balance the tree when an item is inserted or deleted is $O(1)$.

Adelson-Velskii and Landis were the first to propose the following balance condition and show that it has the desired characteristics.

Definition (AVL Balance Condition): An empty binary tree is *AVL balanced*. A non-empty binary tree, $T = \{r, T_L, T_R\}$, is AVL balanced if both T_L and T_R are AVL balanced and

$$|h_L - h_R| \leq 1,$$

where h_L is the height of T_L and h_R is the height of T_R .

Clearly, all perfect binary trees are AVL balanced. What is not so clear is that heights of all trees that satisfy the AVL balance condition are logarithmic in the number of internal nodes.

Theorem: The height, h , of an AVL balanced tree with n internal nodes satisfies

$$\log_2(n + 1) + 1 \leq h \leq 1.440 \log(n + 2) - 0.328.$$

Proof: The lower bound follows directly from Theorem 10.1. It is in fact true for all binary trees regardless of whether they are AVL balanced or not.

To determine the upper bound, we turn the problem around and ask the question, what is the minimum number of internal nodes in an AVL balanced tree of height h ?

Let T_h represent an AVL balanced tree of height h which has the smallest possible number of internal nodes, say N_h . Clearly, T_h must have at least one subtree of height $h-1$ and that subtree must be T_{h-1} . To remain AVL balanced, the other subtree can have height $h-1$ or $h-2$. Since we want the smallest number of internal nodes, it must be T_{h-2} . Therefore, the number of internal nodes in T_h is $N_h = N_{h-1} + N_{h-2} + 1$, where $h \geq 2$.

Clearly, T_0 contains a single internal node, so $N_0 = 1$. Similarly, T_1 contains exactly two nodes, so $N_1 = 2$. Thus, N_h is given by the recurrence

$$N_h = \begin{cases} 1 & h = 0, \\ 2 & h = 1, \\ N_{h-1} + N_{h-2} + 1 & h \geq 2. \end{cases} \quad (10.8)$$

The remarkable thing about Equation \square is its similarity with the definition of *Fibonacci numbers* (Equation \square). In fact, it can easily be shown by induction that

$$N_h \geq F_{h+2} - 1$$

for all $h \geq 0$, where F_k is the k^{th} Fibonacci number.

Base Cases

$$N_0 = 1, \quad F_2 = 1 \implies N_0 \geq F_2 - 1,$$

$$N_1 = 2, \quad F_3 = 2 \implies N_1 \geq F_3 - 1.$$

Inductive Hypothesis: Assume that $N_h \geq F_{h+2} - 1$ for $h = 0, 1, 2, \dots, k$. Then

$$\begin{aligned} N_{h+1} &= N_h + N_{h-1} + 1 \\ &\geq F_{h+2} - 1 + F_{h+1} - 1 + 1 \\ &\geq F_{h+3} - 1 \\ &\geq F_{(h+1)+2} - 1. \end{aligned}$$

Therefore, by induction on k , $N_h \geq F_{h+2} - 1$, for all $h \geq 0$.

According to Theorem \square , the Fibonacci numbers are given by

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n),$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Furthermore, since $\hat{\phi} \approx -0.618$ $|\hat{\phi}^n/\sqrt{5}| < 1$.

Therefore,

$$\begin{aligned} N_h \geq F_{h+2} - 1 &\Rightarrow N_h \geq \phi^{h+2}/\sqrt{5} - 2 \\ &\Rightarrow \sqrt{5}(N_h + 2) \geq \phi^{h+2} \\ &\Rightarrow \log_{\phi}(\sqrt{5}(N_h + 2)) \geq h + 2 \\ &\Rightarrow h \leq \log_{\phi}(N_h + 2) + \log_{\phi} \sqrt{5} - 2 \\ &\Rightarrow h \lesssim 1.440 \log_2(N_h + 2) - 0.328 \end{aligned}$$

This completes the proof of the upper bound.

So, we have shown that the AVL balance condition satisfies the first criterion of a good balance condition--the height of an AVL balanced tree with n internal nodes is $\Theta(\log n)$. What remains to be shown is that the balance condition can be efficiently maintained. To see that it can, we need to look at an implementation.

3.6 Implementing AVL Trees

Having already implemented a binary search tree class, `BinarySearchTree`, we can make use of much of the existing code to implement an AVL tree class. Programme [□](#) introduces the `AVLTree` class which extends the `BinarySearchTree` class introduced in Programme [□](#). The `AVLTree` class inherits most of its functionality from the binary tree class. In particular, it uses the inherited `insert` and `withdraw` methods! However, the inherited `balance`, `attachKey` and `detachKey` methods are overridden and a number of new methods are declared.

```

1 public class AVLTree
2     extends BinarySearchTree
3 {
4     protected int height;
5
6     // ...
7 }
```

Programme1.0: `AVLTree` fields.

Programme [□](#) indicates that an additional field is added in the `AVLTree` class. This turns out to be necessary because we need to be able to determine quickly, i.e., in $O(1)$ time, that the AVL balance condition is satisfied at a given node in the tree. In general, the running time required to compute the height of a tree containing n nodes is $O(n)$. Therefore, to determine whether the AVL balance condition is satisfied at a given node, it is necessary to traverse completely the subtrees of the given node. But this cannot be done in constant time.

To make it possible to verify the AVL balance condition in constant time, the field, `height`, has been added. Thus, every node in an `AVLTree` keeps track of its own height. In this way, it is possible for the `getHeight` method to run in constant time--all it needs to do is to return the value of the `height` field. And this makes it

possible to test whether the AVL balanced condition is satisfied at a given node in constant time.

3.6.1 Inserting Items into an AVL Tree

Inserting an item into an AVL tree is a two-part process. First, the item is inserted into the tree using the usual method for insertion in binary search trees. After the item has been inserted, it is necessary to check that the resulting tree is still AVL balanced and to balance the tree when it is not.

Just as in a regular binary search tree, items are inserted into AVL trees by attaching them to the leaves. To find the correct leaf, we pretend that the item is already in the tree and follow the path taken by the `find` method to determine where the item should go. Assuming that the item is not already in the tree, the search is unsuccessful and terminates at an external, empty node. The item to be inserted is placed in that external node.

Inserting an item in a given external node affects potentially the heights of all of the nodes along the *access path*, i.e., the path from the root to that node. Of course, when an item is inserted in a tree, the height of the tree may increase by one. Therefore, to ensure that the resulting tree is still AVL balanced, the heights of all the nodes along the access path must be recomputed and the AVL balance condition must be checked.

Sometimes increasing the height of a subtree does not violate the AVL balance condition. For example, consider an AVL tree $T = \{r, T_L, T_R\}$. Let h_L and h_R be the heights of T_L and T_R , respectively. Since T is an AVL tree, then $|h_L - h_R| \leq 1$. Now, suppose that $h_L = h_R + 1$. Then, if we insert an item into T_R , its height may increase by one to $h'_R = h_R + 1$. The resulting tree is still AVL balanced since $h_L - h'_R = 0$. In fact, this particular insertion actually makes the tree more balanced! Similarly if $h_L = h_R$, initially, an insertion in either subtree will not result in a violation of the balance condition at the root of T .

On the other hand, if $h_L = h_R + 1$ and the insertion of an item into the left subtree T_L increases the height of that tree to $h'_L = h_L + 1$, the AVL balance condition is no longer satisfied because $h'_L - h_R = 2$. Therefore, it is necessary to change the structure of the tree to bring it back into

balance.

3.6.2 Removing Items from an AVL Tree

The method for removing items from an AVL tree is inherited from the `BinarySearchTree` class in the same way as AVL insertion. All the differences are encapsulated in the `detachKey` and `balance` methods. The `balance` method is discussed above. The `detachKey` method is defined in the programme below:

```

1  public class AVLTree
2      extends BinarySearchTree
3  {
4      protected int height;
5
6      public Object detachKey ()
7      {
8          height = -1;
9          return super.detachKey ();
10     }
11     // ...
12 }

```

Programme 1.1: AVLTree class detachKey method

SELF ASSESSMENT EXERCISE

When is a search said to be successful?

4.0 CONCLUSION

In this unit, you have learned about trees. You have also learned about binary trees and tree traversals. Finally, you have been able to learn how to implement trees.

5.0 SUMMARY

What you have learned in this unit is focused on trees, the common types and implementation of trees.

6.0 TUTOR-MARKED ASSIGNMENT

What are the characteristics of a good *balance condition*?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 GARBAGE COLLECTION AND OTHER HEAP

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 What is Garbage?
 - 3.2 Reduce, Reuse, Recycle
 - 3.3 Helping the Garbage Collector
 - 3.4 Reference Counting Garbage Collection
 - 3.5 Mark-and-Sweep Garbage Collection
 - 3.6 The Fragmentation Problem
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit describes garbage and garbage collection. It describes three strategies for reducing garbage cost. Finally, it discusses the fragmentation problem.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe garbage
- explain garbage collection
- describe the mark-and-sweep garbage collection
- explain the fragmentation problem.

3.0 MAIN CONTENT

3.1 What is Garbage?

While Java provides the means to create an object, the language does not provide the means to destroy an object *explicitly*. As long as a programme contains a reference to some object instance, the Java virtual machine is required to ensure that the object exists. If the Java language provided the means to destroy objects, it would be possible for a programme to destroy an object even when a reference to that object still existed. This situation is unsafe because the programme could attempt later to invoke a method on the destroyed object, leading to unpredictable results.

The situation which arises when a programme contains a reference (or pointer) to a destroyed object is called a *dangling reference* (or dangling

pointer). By disallowing the explicit destruction of objects, Java eliminates the problem of dangling references.

Languages that support the explicit destruction of objects typically require the programme to keep track of all the objects it creates and to destroy them explicitly when they are not longer needed. If a programme somehow loses track of an object it has created then that object cannot be destroyed. And if the object is never destroyed, the memory occupied by that object cannot be used again by the programme.

A programme that loses track of objects before it destroys them suffers from a *memory leak*. If we run a programme that has a memory leak for a very long time, it is quite possible that it will exhaust all the available memory and eventually fail because no new objects can be created. It would seem that by disallowing the explicit destruction of objects, a Java programme is doomed to eventual failure due to memory exhaustion. Indeed, this would be the case, were it not for the fact that the Java language specification requires the Java virtual machine to be able to find unreferenced objects and to reclaim the memory locations allocated to those objects.

An unreferenced object is called *garbage* and the process of finding all the unreferenced objects and reclaiming the storage is called *garbage collection*. Just as the Java language does not specify precisely how objects are to be represented in the memory of a virtual machine, the language specification also does not stipulate how the garbage collection is to be implemented or when it should be done.

Garbage collection is usually invoked when the total amount of memory allocated to a Java programme exceeds some threshold. Typically, the programme is suspended while the garbage collection is done.

In the analyses presented in the preceding chapters, we assume that the running time of the `new` operator is a fixed constant, T_{new} and we completely ignore the garbage collection overhead. In reality, neither assumption is valid. Even if sufficient memory is available, the time required by the Java virtual machine to locate an unused region of memory depends very much on the data structures used to keep track of the memory regions allocated to a programme as well as on the way in which a programme uses the objects it creates. Furthermore, invoking the `new` operator may trigger the garbage collection process. The running time for garbage collection can be a significant fraction of the total running time of a programme.

3.2 Reduce, Reuse, Recycle

Modern societies produce an excessive amount of waste. The costs of doing so include the direct costs of waste disposal as well as the damage to the environment caused by the manufacturing, distribution, and ultimate disposal of products. The slogan ``*reduce, reuse, recycle*," prescribes three strategies for reducing the environmental costs associated with waste materials.

These strategies apply equally well to Java programmes! A Java programme that creates excessive garbage may require more frequent garbage collection than a programme that creates less garbage. Since garbage collection can take a significant amount of time to do, it makes sense to use strategies that decrease the cost of garbage collection.

3.3 Helping the Garbage Collector

The preceding section presents strategies for avoiding garbage collection. However, there are times when garbage collection is actually desirable. Imagine a programme that requires a significant amount of memory. Suppose the amount of memory required is very close to the amount of memory available for use by the Java virtual machine. The performance of such a programme is going to depend on the ability of the garbage collector to find and reclaim as much unused storage as possible. Otherwise, the garbage collector will run too often. In this case, it pays to help out the garbage collector.

How can we help out the garbage collector? Since the garbage collector collects only unreferenced objects it is necessary to eliminate all references to objects which are no longer needed. This is done by assigning the value `null` to every variable that refers to an object that is no longer needed. Consequently, helping the garbage collector requires a programme to do a bit more work.

3.4 Reference Counting Garbage Collection

The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist?

A simple expedient is to keep track in each object, the total number of references to that object. That is, we add a special field to each object called a *reference count*. The idea is that the reference count field is not accessible to the Java programme. Instead, the reference count field is updated by the Java virtual machine itself.

Consider the statement:

Object `p = new Integer (57);`
which creates a new instance of the `Integer` class. Only a single variable, `p`, refers to the object. Thus, its reference count should be one.

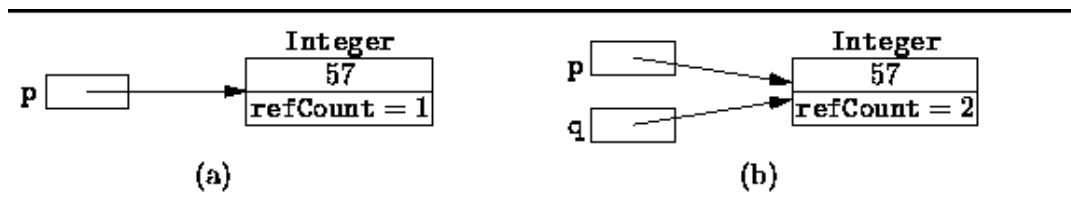


Figure: Objects with reference counters.

Now consider the following sequence of statements:

Object `p = new Integer (57);`

Object `q = p;`

This sequence creates a single `Integer` instance. Both `p` and `q` refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose `p` and `q` are both reference variables. The assignment

`p = q;`

would be implemented by the Java virtual machine as follows:

```
if (p != q)
{
  if (p != null)
    --p.refCount;
  p = q;
  if (p != null)
    ++p.refCount;
}
```

For example, suppose `p` and `q` are initialised as follows:

Object `p = new Integer (57);`

Object `q = new Integer (99);`

As shown in Figure □ (a), two `Integer` objects are created, each with a reference count of one. Now, suppose we assign `q` to `p` using the code sequence given above. Figure □ (b) shows that after the assignment, both `p` and `q` refer to the same object--its reference count is two. And

the reference count on Integer (57) has gone to zero which indicates that it is garbage.

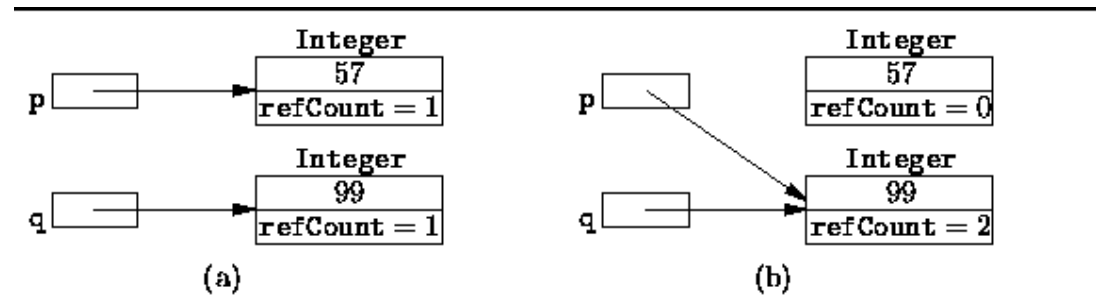


Figure: Reference counts before and after the assignment `p = q`.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the programme. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment `p = q` in the Java virtual machine as follows:

```
if (p != q)
{
  if (p != null)
    if (--p.refCount == 0)
      heap.release (p);
  p = q;
  if (p != null)
    ++p.refCount;
}
```

Notice that the `release` method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

3.5 Mark-and-Sweep Garbage Collection

This section presents the *mark-and-sweep* garbage collection algorithm. The mark-and-sweep algorithm was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structures. Variations of the mark-and-sweep algorithm continue to be among the most commonly used garbage collection techniques.

When using mark-and-sweep, unreferenced objects are not reclaimed immediately. Instead, garbage is allowed to accumulate until all available memory has been exhausted. When that happens, the execution of the programme is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the programme can resume.

The mark-and-sweep algorithm is called a *tracing* garbage collector because it *traces out* the entire collection of objects that are directly or indirectly accessible by the programme. The objects that a programme can access directly are those objects which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the *roots*. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be *live*. Conversely, an object which is not *live* is garbage.

The mark-and-sweep algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the *mark* phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the *sweep* phase. The algorithm can be expressed as follows:

```
for each root variable r
  mark (r);
sweep ();
```

In order to distinguish the live objects from garbage, we record the state of an object in each object. That is, we add a special **boolean** field to each object called, say, **marked**. By default, all objects are unmarked when they are created. Thus, the **marked** field is initially **false**.

An object **p** and all the objects indirectly accessible from **p** can be marked by using the following recursive **mark** method:

```
void mark (Object p)

if (!p.marked)

p.marked = true;
for each Object q referenced by p
mark (q);
```





Notice that this recursive `mark` algorithm does nothing when it encounters an object that has already been marked. Consequently, the algorithm is guaranteed to terminate. And it terminates only when all accessible objects have been marked.

In its second phase, the mark-and-sweep algorithm scans through all the objects in the heap, in order to locate all the unmarked objects. The storage allocated to the unmarked objects is reclaimed during the scan. At the same time, the `marked` field on every live object is set back to `false` in preparation for the next invocation of the mark-and-sweep garbage collection algorithm:

```
void sweep ()

for each Object p in the heap

if (p.marked)
p.marked = false
else
heap.release (p);
```

Figure  illustrates the operation of the mark-and-sweep garbage collection algorithm. Figure  (a) shows the conditions before garbage collection begins. In this example, there is a single root variable. Figure  (b) shows the effect of the *mark* phase of the algorithm. At this point, all live objects have been marked. Finally, Figure  (c) shows the objects left after the *sweep* phase has been completed. Only live objects remain in memory and the `marked` fields have all been set to `false` again.

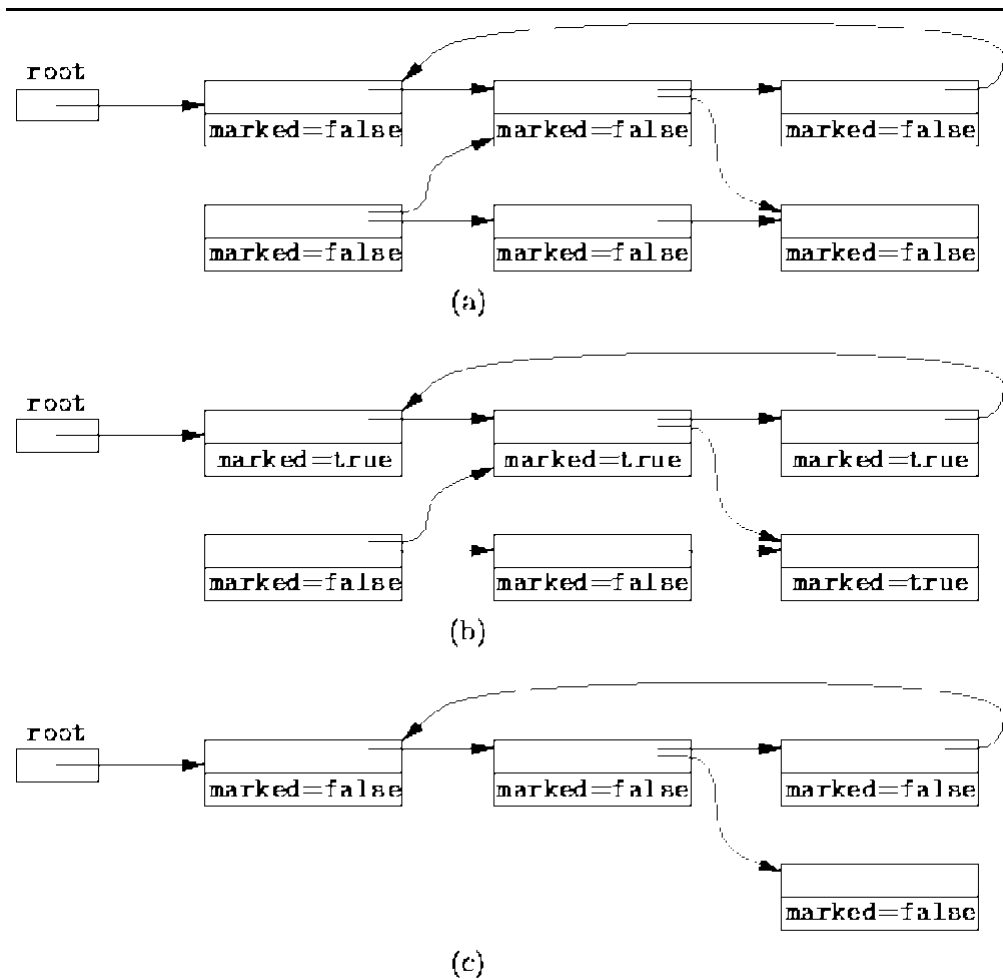


Figure: Mark-and-sweep garbage collection

Because the mark-and-sweep garbage collection algorithm traces out the set of objects accessible from the roots, it is able to correctly identify and collect garbage even in the presence of reference cycles. This is the main advantage of mark-and-sweep over the reference counting technique presented in the preceding section. A secondary benefit of the mark-and-sweep approach is that the normal manipulations of reference variables incur no overhead.

The main disadvantage of the mark-and-sweep approach is the fact that normal programme execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a programme that interacts with a human user or that must satisfy real-time execution constraints. For example, an interactive application that uses mark-and-sweep garbage collection becomes unresponsive periodically.

3.6 The Fragmentation Problem

Fragmentation is a phenomenon that occurs in a long-running programme that has undergone garbage collection several times. The problem is that objects tend to become spread out in the heap. Live objects end up being separated by many, small unused memory regions. The problem in this situation is that it may become impossible to allocate memory for an object. While there may indeed be sufficient unused memory, the unused memory is not contiguous. Since objects typically occupy consecutive memory locations, it is impossible to allocate storage.

The mark-and-sweep algorithm does not address fragmentation. Even after reclaiming the storage from all garbage objects, the heap may still be too fragmented to allocate the required amount of space. The next section presents an alternative to the mark-and-sweep algorithm that also *defragments* (or *compacts*) the heap.

SELF ASSESSMENT EXERCISE 1

What do you understand by garbage collection?

SELF ASSESSMENT EXERCISE 2

Describe three strategies for reducing garbage cost.

4.0 CONCLUSION

In this unit, you have learned about garbage and garbage collection. You have also been able to learn about reference counting garbage collection and mark-and-sweep garbage collection. Finally, you have been able to understand the fragmentation problem.

5.0 SUMMARY

What you have learned borders on the garbage, garbage collection and the fragmentation problem.

6.0 TUTOR-MARKED ASSIGNMENT

The mark-and-sweep algorithm is referred to as a tracing garbage collection. True or False? Discuss.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 MEMORY ALLOCATION

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Memory Allocation
 - 3.2 First Fit
 - 3.3 Best Fit
 - 3.4 Fragmentation
 - 3.5 Buddy System
 - 3.6 Suballocators
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit describes the process of memory allocation. Different techniques of memory allocation are equally considered.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the concept of memory allocation
- discuss the first fit allocation technique
- explain the best fit allocation technique
- describe the buddy system.

3.0 MAIN CONTENT

3.1 Memory Allocation

Memory allocation is the process of assigning blocks of memory on request. Typically, the [allocator](#) receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses. A few are described briefly here:

Firstfit

Best fit

Buddysystem

Suballocators

These techniques can often be used in combination.

3.2 First Fit

In the firstfit algorithm, the allocator keeps a list of free blocks (known as the freelist) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.

The first fit algorithm performs reasonably well, as it ensures that allocations are quick. When recycling free blocks, there is a choice as to where to add the blocks to the list effectively in order the free list is kept:

3.3 Best Fit

The best fit is the allocation policy that always allocates from the smallest suitable free block. Suitable allocation mechanisms include sequential fit searching for a perfect fit, first fit on a size-ordered free block chain, segreated fits, and indexed fits. Many good fit allocators are also described as bestfit.

In theory, best fit may exhibit bad fragmentation, but in practice, this is not commonly observed.

3.4 Fragmentation

Fragmentation is the inability to use memory because of the arrangement of memory already in use. It is usually divided into external fragmentation and internalfragmentation.

3.5 Buddy System

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest, can be divided into two smaller blocks of permitted sizes.

When the allocator receives a request for memory, it rounds the requested size off to a permitted size, and returns the first block from

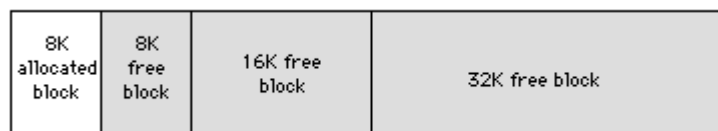
that size's free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size ([coalescence](#)). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address.

A binary buddy heap before allocation



A binary buddy heap after allocating a 8 kB block



A binary buddy heap after allocating a 10 kB block; note the 6 kB wasted because of rounding off



For example, an allocator in a binary buddy system might have sizes of 16, 32,... 64 kB. It might start off with a single block of 64 kB. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. It would then split the 64 kB block into two blocks of 32 kB, split one of them into two blocks of 16 kB, and split one of them into two blocks of 8 kB. The allocator would then return one of the 8 kB blocks to the application and keep the remaining three blocks of 8 kB, 16 kB, and 32 kB on the appropriate free lists. If the application then requested a block of 10 kB, the allocator would round this request off to 16 kB, and return the 16 kB block from its free list, wasting 6 kB in the process.

A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208,... bytes, such that each size is the sum of the two preceding sizes. When splitting a block from one free list, the two parts get added to the two preceding free lists.

A buddy system can work very well or very badly, depending on how the chosen sizes interact with typical requests for memory and what the

pattern of returned blocks is. The rounding typically leads to a significant amount of wasted memory, which is called [internal fragmentation](#). This can be reduced by making the permitted block sizes closer together.

3.6 Suballocators

There are many examples of application programmes that include additional memory management code called a suballocator. A suballocator obtains large blocks of memory from the system memory manager and allocates the memory to the application in smaller pieces. Suballocators are usually written for one of the following reasons:

- To avoid general inefficiency in the system memory manager;
- To take advantage of special knowledge of the application's memory requirements that cannot be expressed to the system memory manager;
- To provide memory management services that the system memory manager does not supply.

In general, suballocators are less efficient than having a single memory manager that is well-written and has a flexible interface. It is also harder to avoid memory management bugs if the memory manager is composed of several layers, and if each application has its own variation of suballocator.

Many applications have one or two sizes of blocks that form the vast majority of their allocations. One of the most common uses of a suballocator is to supply the application with objects of one size. This greatly reduces the problem of [external fragmentation](#). Such a suballocator can have a very simple allocation policy.

There are dangers involved in making use of special knowledge of the application's memory requirements. If those requirements change, then the performance of the suballocator is likely to be much worse than that of a general allocator. It is often better to have a memory manager that can respond dynamically to changing requirements.

SELF ASSESSMENT EXERCISE 1

State the main advantage of the buddy system

SELF ASSESSMENT EXERCISE 2

What do you understand by the phrase 'first fit'?

4.0 CONCLUSION

In this unit, you have learned about memory allocation. You have also been able to learn about the first fit, best fit and buddy systems of memory allocation.

5.0 SUMMARY

What you have learned borders on memory allocation and the techniques of memory allocation.

6.0 TUTOR-MARKED ASSIGNMENT

Describe the buddy system.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A, (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

MODULE 3 INTRODUCTION TO JAVA PROGRAMMING

Unit 1	Object-Oriented Programming Concepts
Unit 2	Variables
Unit 3	Operators
Unit 4	Expressions, Statements and Blocks
Unit 5	Control Flow Statements

UNIT 1 OBJECT-ORIENTED PROGRAMMING CONCEPTS

CONTENTS

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	Objects
3.2	What is a Class?
3.3	Inheritance
3.4	What is an Interface?
3.5	What is a Package?
3.6	Object-Oriented Programming
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings

1.0 INTRODUCTION

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This unit will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe an object
- explain what a class is
- define an Inheritance
- explain the term 'object-oriented programming'.

3.0 MAIN CONTENT

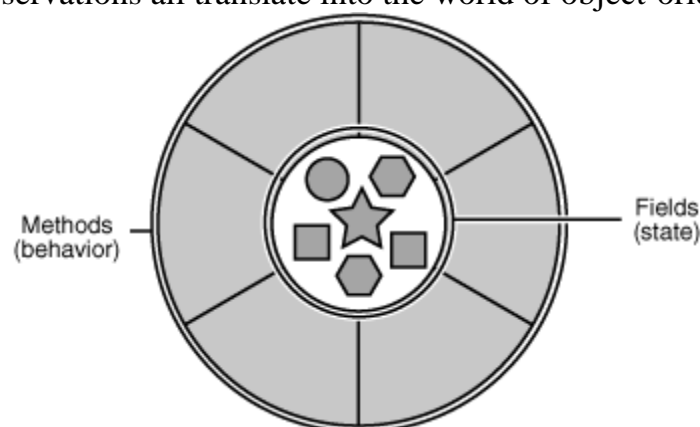
3.1 Objects

An object is a software bundle of related state and behaviour. Software objects are often used to model the real-world objects that you find in everyday life. This unit explains how state and behaviour are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behaviour*. Dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behaviour (changing gear, changing pedal cadence, applying brakes). Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behaviour can this object put up?" Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviours (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behaviour (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

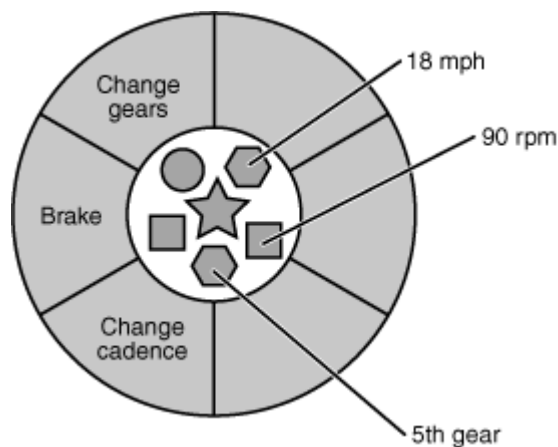


A software object

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in *fields*

(variables in some programming languages) and exposes its behaviour through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* – a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



A bicycle modeled as a software object

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has six gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your programme. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

4. **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to

fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

3.2 What is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behaviour of a real-world object. It intentionally focuses on the basics, showing how even a simple class can clearly model state and behaviour.

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

The following `Bicycle` class is one possible implementation of a bicycle:

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

```

        void printStates() {
            System.out.println("cadence:"+cadence+"speed:"+speed+" gear:"+gear);
        }
    }

```

The syntax of the Java programming language will look new to you. The fields `cadence`, `speed`, and `gear` represent the object's state, and the methods (`changeCadence`, `changeGear`, `speedUp` etc.) define its interaction with the outside world.

You may have noticed that the `Bicycle` class does not contain a `main` method. That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new `Bicycle` objects belongs to some other class in your application.

Here's a `BicycleDemo` class that creates two separate `Bicycle` objects and invokes their methods:

```

class BicycleDemo {
    public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();
        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}

```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```

cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3

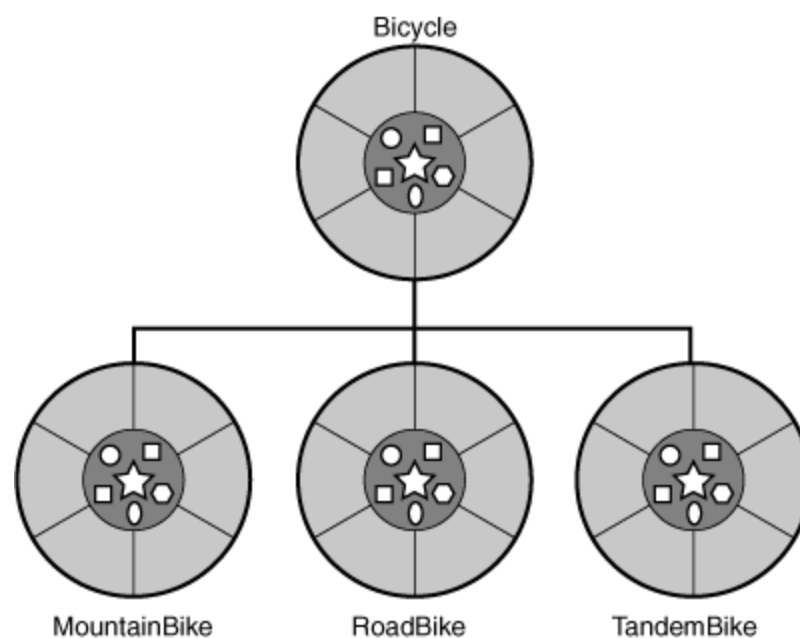
```

3.3 Inheritance

Inheritance provides a powerful and natural mechanism for organising and structuring your software. This section explains how classes inherit state and behaviour from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet, each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behaviour from other classes. In this example, `Bicycle` now becomes the *superclass* of `MountainBike`, `RoadBike`, and `TandemBike`. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:



A hierarchy of bicycle classes

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {
```

```
// new fields and methods defining a mountain bike would go here
```

```
}
```

This gives `MountainBike` all the same fields and methods as `Bicycle`, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behaviour that each superclass defines, since that code will not appear in the source file of each subclass.

3.4 What is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behaviour published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behaviour, if specified as an interface, might appear as follows:

```
interface Bicycle {
```

```
void changeCadence(int newValue);
```

```
void changeGear(int newValue);
```

```
void speedUp(int increment);
```

```
void applyBrakes(int decrement);
```

```
}
```

To implement this interface, the name of your class would change (to `ACMEBicycle`, for example), and you'd use the `implements` keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {
```

```
// remainder of this class implemented as before
```

```
}
```

Implementing an interface allows a class to become more formal about the behaviour it promises to provide. Interfaces form a contract between

the class and the outside world, and this contract is enforced at build

time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

3.5 What is a Package?

A package is a namespace for organising classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

Conceptually, you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organised by placing related classes and interfaces into packages. The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface" or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behaviour for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the file system; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

3.6 Object-Oriented Programming

Object-oriented programming (OOP) is a programming language model organised around "objects" rather than "actions" and data rather than logic.

Historically, a programme has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from

human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to

the little widgets on your computer desktop (such as buttons and scroll bars).

The first step in OOP is to identify all the objects you want to manipulate and how they relate to each other, an exercise often known as *data modeling*. Once you've identified an object, you generalise it as a class of objects (think of Plato's concept of the "ideal" chair that stands for all chairs) and define the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. A real instance of a class is called (no surprise here) an "object" or, in some environments, an "instance of a class." The object or class instance is what you run in the computer. Its methods provide computer instructions and the class object characteristics provide relevant data. You communicate with objects - and they communicate with each other - with well-defined interfaces called *messages*.

The concepts and rules used in object-oriented programming provide these important benefits:

The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding. Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other programme data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

The definition of a class is reusable not only by the programme for which it is initially created but also by other object-oriented programmes (and, for this reason, can be more easily distributed for use in networks). The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself. One of the first object-oriented computer languages was called *Smalltalk*. C++ and Java are the most popular object-oriented languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet.

4.0 CONCLUSION

In this unit you have learned about the object-oriented programming concepts-objects, class, inheritance, interface and package. You have also been able to understand object-oriented programming in general.

5.0 SUMMARY

What you have learned borders on the basic concepts of object-oriented programming. The subsequent units shall build upon these fundamentals.

SELF ASSESSMENT EXERCISE 1

Define a class.

SELF ASSESSMENT EXERCISE 2

What is an inheritance?

6.0 TUTOR-MARKED ASSIGNMENT

What do you understand by object-oriented programming?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 2 VARIABLES

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Basics
 - 3.2 Java Programming Variables
 - 3.3 Naming Conventions
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will learn about variables and their naming conventions.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- define a variable
- describe types of java programming variables
- explain the naming conventions of variables.

3.0 MAIN CONTENT

3.1 Basics

In computer science, a **variable** (sometimes called an *object* or *identifier*) is a symbolic representation used to denote a quantity or expression.

However, in computer programming, a variable is a special value (also often called a reference) that has the property of being able to be associated with another value (or not). What is variable across time is the association. Obtaining the value associated with a variable is often called *dereferencing*, and creating or changing the association is called *assignment*.

Variables are usually named by an identifier, but they can be anonymous, and variables can be associated with other variables. In the computing context, variable identifiers often consist of alphanumeric strings. These identifiers are then used to refer to values in computer memory. This convention of matching identifiers to values is but one of

several alternative programmatic conventions for accessing values in computer memory.

3.2 Java Programming Variables

In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

The Java programming language defines the following kinds of variables:

Instance Variables (Non-Static Fields): Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent of the `currentSpeed` of another.

Class Variables (Static Fields): A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually, the same number of gears will apply to all instances. The code, `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.

Local Variables: Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

Parameters: You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main (String [] args)`. Here, the `args` variable is the parameter to this method. The important thing to

remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

3.3 Naming Conventions

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarised as follows:

Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "\$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "\$" or "_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged. White space is not permitted.

Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases, it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`, `c`, and `g`. Also keep in mind that the name you choose must not be a keyword or reserved word.

SELF ASSESSMENT EXERCISE 1

What do you understand by variables? Give at least two examples.

4.0 CONCLUSION

In this unit, you have learned about variables, types of Java programming language variable as well as their naming conventions.

5.0 SUMMARY

What you have learned in this unit is based on variables and the conventions for naming them.

6.0 TUTOR-MARKED ASSIGNMENT

Variable names are case-sensitive. True or False? Discuss.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 3 OPERATORS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Operators
 - 3.2 The Simple Assignment Operators
 - 3.3 The Arithmetic Operators
 - 3.4 The Unary Operators
 - 3.5 The Equality and Relational Operators
 - 3.6 The Conditional Operators
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

What you will learn in this unit borders on operators. The common types of operators will equally be discussed.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the term ‘operators’
- describe simple assignment operators
- explain arithmetic operators
- discuss unary operators
- explain equality and relational operators
- discuss the conditional operators.

3.0 MAIN CONTENT

3.1 Operators

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower

precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated from right to left.

Operator Precedence

Operators	Precedence
Postfix	<i>expr++ expr--</i>
Unary	<i>++expr --expr +expr -expr ~ !</i>
Multiplicative	<i>* / %</i>
Additive	<i>+ -</i>
Shift	<i><< >> >>></i>
Relational	<i>< > <= >= instanceof</i>
Equality	<i>== !=</i>
bitwise AND	<i>&</i>
bitwise exclusive OR	<i>^</i>
bitwise inclusive OR	<i> </i>
logical AND	<i>&&</i>
logical OR	<i> </i>
Ternary	<i>? :</i>
assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

3.2 The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=". You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

```
int cadence = 0;
```

```
int speed = 0;
```

```
int gear = 1;
```

This operator can also be used on objects to assign *object references*, as discussed in [Creating Objects](#).

3.3 The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

+	additive operator (also used for String concatenation)
-	subtraction operator
*	multiplication operator
/	division operator
%	remainder operator

The following programme, [ArithmeticDemo](#), tests the arithmetic operators.

```
class ArithmeticDemo {  
public static void main (String[] args){
```

```
int result = 1 + 2; // result is now 3  
System.out.println(result);
```

```
result = result - 1; // result is now 2  
System.out.println(result);
```

```
result = result * 2; // result is now 4  
System.out.println(result);
```

```
result = result / 2; // result is now 2  
System.out.println(result);
```

```
result = result + 8; // result is now 10  
result = result % 7; // result is now 3
```



```
System.out.println(result);
}
}
```

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, `x += 1;` and `x = x + 1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following [ConcatDemo](#) programme:

```
class ConcatDemo
{
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = "a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

By the end of this programme, the variable, `thirdString`, contains "This is a concatenated string.", which gets printed to standard output.

3.4 The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

- `+` Unary plus operator; indicates positive value (numbers are positive without this, however)
- `-` Unary minus operator; negates an expression
- `++` Increment operator; increments a value by 1
- `--` Decrement operator; decrements a value by 1
- `!` Logical complement operator; inverts the value of a boolean

The following programme, [UnaryDemo](#), tests the unary operators:

```
class UnaryDemo {
```

```
    public static void main(String[] args){
        int result = +1; // result is now 1
        System.out.println(result);
        result--; // result is now 0
        System.out.println(result);
        result++; // result is now 1
        System.out.println(result);
    }
```

```

    result = -result; // result is now -1
    System.out.println(result); boolean
    success = false;
    System.out.println(success); // false
    System.out.println(!success); // true

}
}

```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code, `result++;` and `++result;` will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following programme, [PrePostDemo](#), illustrates the prefix/postfix unary increment operator:

```

class PrePostDemo {
public static void main(String[] args){
int i = 3;
i++;
System.out.println(i);    // "4"
++i;
System.out.println(i);    // "5"
System.out.println(++i);  // "6"
System.out.println(i++);  // "6"
System.out.println(i);    // "7"
}
}

```

3.5 The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use `"=="`, not `"="`, when testing if two primitive values are equal.

```

==    equal to
!=    not equal to
>     greater than
>=    greater than or equal to

```

< less than
 <= less than or equal to

The following programme, [ComparisonDemo](#), tests the comparison operators:

```
class ComparisonDemo {

    public static void main(String[] args){

        int value1 = 1;
        int value2 = 2;
        if(value1 == value2) System.out.println("value1 == value2");
        if(value1 != value2) System.out.println("value1 != value2");
        if(value1 > value2) System.out.println("value1 > value2");
        if(value1 < value2) System.out.println("value1 < value2");
        if(value1 <= value2) System.out.println("value1 <= value2");

    }
}
Output:
value1 != value2
value1 < value2
value1 <= value2
```

3.6 The Conditional Operators

The && and || operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behaviour, which means that the second operand is evaluated only if needed.

&& Conditional-AND
 || Conditional-OR

The following programme, [ConditionalDemo1](#), tests these operators:

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2)) System.out.println("value1 is 1
        AND value2 is 2");
        if((value1 == 1) || (value2 == 1)) System.out.println("value1 is 1 OR
        value2 is 1");
    }
}
```

```

    }
}

```

Another conditional operator is `?:`, which can be thought of as shorthand for an `if-then-else` statement (discussed in the [Control Flow Statements](#) section of this lesson). This operator is also known as the *ternary operator* because it uses three operands. In the following example, this operator should be read as: "If `someCondition` is

`true`, assign the value of `value1` to `result`. Otherwise, assign the value of `value2` to `result`."

The following programme, [ConditionalDemo2](#), tests the `?:` operator:

```

class ConditionalDemo2 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

        System.out.println(result);

    }
}

```

Because `someCondition` is `true`, this programme prints "1" to the screen. Use the `?:` operator instead of an `if-then-else` statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

SELF ASSESSMENT EXERCISE

Discuss the term, operators.

4.0 CONCLUSION

In this unit you have learned about operators. You have also been able to identify the common types of operators.

5.0 SUMMARY

What you have learned in this unit concerns operators and the common types.

6.0 TUTOR-MARKED ASSIGNMENT

Mention 4 common types of operators, stating their functions.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 EXPRESSIONS, STATEMENTS AND BLOCKS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Expressions
 - 3.2 Statements
 - 3.3 Blocks
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

Now that you understand variables and operators, it's time to learn about *expressions*, *statements*, and *blocks*. Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- define an expression
- describe statements, giving typical examples of expression statements
- discuss the concept of blocks.

3.0 MAIN CONTENT

3.1 Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

You've already seen examples of expressions, illustrated in bold below:

```
int cadence = 0;  
anArray[0] = 100;  
System.out.println("Element 1 at index 0: " + anArray[0]);
```

```
int result = 1 + 2; // result is now 3  
if(value1 == value2) System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression, `cadence = 0`, returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100    // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parenthesis: `(and)`. For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100  // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

```
x + y / 100
```

```
x + (y / 100) // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

3.2 Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution.

The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions.

Such statements are called *expression statements*. Here are some examples of expression statements.

```
aValue = 8933.234;      // assignment statement
aValue++;              // increment statement
System.out.println("Hello World!"); // method invocation statement
Bicycle myBike = new Bicycle (); // object creation statement
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
double aValue = 8933.234; //declaration statement
```

Finally, *control flow statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, [ControlFlowStatements](#)

3.3 Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, [BlockDemo](#), illustrates the use of blocks:

```
class BlockDemo {
public static void main(String[] args) {
boolean condition = true;
if (condition) { // begin block 1
System.out.println("Condition is true.");
} // end block one
else { // begin block 2
System.out.println("Condition is false.");
} // end block 2
}
```


}

SELF ASSESSMENT EXERCISE 1

What are the core components of statements?

SELF ASSESSMENT EXERCISE 2

Distinguish between statements and sentences.

4.0 CONCLUSION

In this unit you have learned about the expressions. You have also been able to distinguish between statements and sentences. You should also have learned about blocks.

5.0 SUMMARY

What you have learned in this unit concerns expressions, statements and blocks. In the next unit, you shall learn about control flow statements.

6.0 TUTOR-MARKED ASSIGNMENT

Identify the following kinds of expression statements:

```
aValue = 8933.234;
aValue++;
System.out.println("Hello World!");
Bicycle myBike = new Bicycle();
```

7.0 REFERENCES/FURTHER READINGS

- Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.
- French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.
- Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.
- Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>
<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 CONTROL FLOW STATEMENTS

CONTENTS

1.0 Introduction

- 2.0 Objectives
- 3.0 Main Content
 - 3.1 The Control Flow Statements
 - 3.2 The If-Then Statements
 - 3.3 The If-Then-Else Statements
 - 3.4 The Switch Statements
 - 3.5 The While and Do-While Statements
 - 3.6 The For Statements
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, the student will gain knowledge of control flow statements. The unit describes the decision-making statements (`if-then`, `if-then-else`, `switch`) and the looping statements (`for`, `while`, `do-while`), supported by the Java programming language.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe control flow statements
- gain knowledge of the decision-making statements
- explain the looping statements.

3.0 MAIN CONTENT

3.1 The Control Flow Statement

The statements inside your source files are generally executed from top to bottom, in the order that they appear. ***Control flow statements***, however, break up the flow of execution by employing decision making, looping, and branching, enabling your programme to *conditionally* execute particular blocks of code.

3.2 The If-Then Statements

The `if-then` statement is the most basic of all the control flow statements. It tells your programme to execute a certain section of code *only if* a particular test evaluates to `true`. For example, the `Bicycle`

class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

```
void applyBrakes(){
if (isMoving){ // the "if" clause: bicycle must be moving
currentSpeed--; // the "then" clause: decrease current speed
}
}
```

If this test evaluates to `false` (meaning that the bicycle is not in motion), control jumps to the end of the `if-then` statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes(){
if (isMoving) currentSpeed--; // same as above, but without braces
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

3.3 The If-Then-Else Statement

The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. You could use an `if-then-else` statement in the `applyBrakes` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes(){ if
(isMoving) {
currentSpeed--;
} else {
System.err.println("The bicycle has already stopped!");
}
```

```
}
```

The following programme, `IfElseDemo`, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {

    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the programme is:

```
Grade = C
```

You may have noticed that the value of `testscore` can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C';`) and the remaining conditions are not evaluated.

3.4 The Switch Statement

Unlike `if-then` and `if-then-else`, the `switch` statement allows for any number of possible execution paths. A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in [Classes and Inheritance](#)) and a few special classes that "wrap" certain primitive types: [Character](#), [Byte](#), [Short](#), and [Integer](#) (discussed in [SimpleDataObjects](#).)

The following programme, [SwitchDemo](#), declares an `int` named `month` whose value represents a month out of the year. The programme displays the name of the month, based on the value of `month`, using the `switch` statement.

```
class SwitchDemo {
public static void main(String[] args) {

int month = 8;
switch (month) {
case 1: System.out.println("January"); break;
case 2: System.out.println("February"); break;
case 3: System.out.println("March"); break;
case 4: System.out.println("April"); break; case
5: System.out.println("May"); break;
case 6: System.out.println("June"); break;
case 7: System.out.println("July"); break; case
8: System.out.println("August"); break;
case 9: System.out.println("September"); break;
case 10: System.out.println("October"); break; case
11: System.out.println("November"); break; case 12:
System.out.println("December"); break; default:
System.out.println("Invalid month.");break;
}
}
}
```

In this case, "August" is printed to standard output.

The body of a `switch` statement is known as a *switch block*. Any statement immediately contained by the `switch` block may be labeled with one or more `cases` or `default` labels. The `switch` statement evaluates its expression and executes the appropriate `case`.

Of course, you could also implement the same thing with `if-then-else` statements:

```
int month = 8;
if (month == 1) {
System.out.println("January");
} else if (month == 2) {
System.out.println("February");
}
... // and so on
```

Deciding whether to use `if-then-else` statements or a `switch` statement is sometimes a judgment call. You can decide which one to use based on readability and other factors. An `if-then-else` statement can be used to make decisions based on ranges of values or

conditions, whereas a `switch` statement can make decisions based only on a single integer or enumerated value.

Another point of interest is the `break` statement after each case. Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, case statements fall through; that is, without an explicit `break`, control will flow sequentially through subsequent case statements. The following programme, [SwitchDemo2](#), illustrates why it might be useful to have case statements fall through:

```
class SwitchDemo2 {
public static void main(String[] args) {

    int month = 2; int
    year = 2000; int
    numDays = 0;

    switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numDays = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numDays = 30;
        break;
    case 2:
        if ( ((year % 4 == 0) && !(year % 100 == 0))
            || (year % 400 == 0) )
            numDays = 29;
        else
```

```

        numDays = 28;
        break;
    default:
        System.out.println("Invalid month.");

        break;

    }
    System.out.println("Number of Days = " + numDays);
}
}

```

This is the output from the programme.

Number of Days = 29

Technically, the final `break` is not required because flow would fall out of the `switch` statement anyway. However, we recommend using a `break` so that modifying the code is easier and less error-prone. The `default` section handles all values that aren't explicitly handled by one of the `case` sections.

3.5 The While and Do-While Statements

The `while` statement continually executes a block of statements while a particular condition is `true`. Its syntax can be expressed as:

```

while (expression) {
    statement(s)
}

```

The `while` statement evaluates *expression*, which must return a boolean value. If the expression evaluates to `true`, the `while` statement executes the *statement(s)* in the `while` block. The `while` statement continues testing the expression and executing its block until the expression evaluates to `false`. Using the `while` statement to print the values from 1 through 10 can be accomplished as in the following [WhileDemo](#) programme:

```

class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}

```

implement an infinite loop using the `while` statement as follows:


```
while (true){
// your code goes here
}
```

The Java programming language also provides a `do-while` statement, which can be expressed as follows:

```
do {
statement(s)
} while (expression);
```

The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once, as shown in the following [DoWhileDemo](#) programme:

```
class DoWhileDemo {
public static void main(String[] args){
int count = 1;
do {
System.out.println("Count is: " + count);
count++;
} while (count <= 11);
}
}
```

3.6 The For Statement

The *For Statement* provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the `for`-statement can be expressed as follows:

```
for (initialization; termination; increment) {
statement(s)
}
```

When using this version of the `for`-statement, keep in mind that:

The *initialisation* expression initialises the loop; it's executed once, as the loop begins.

When the *termination* expression evaluates to `false`, the loop terminates.

The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following programme, [ForDemo](#), uses the general form of the `for` statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

The output of this programme is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialisation expression. The scope of this variable extends from its declaration to the end of the block governed by the `for`-statement, so it can be used in the termination and increment expressions as well. If the variable that controls a `for`-statement is not needed outside of the loop, it's best to declare the variable in the initialisation expression. The names `i`, `j`, and `k` are often used to control `for` loops; declaring them within the initialisation expression limits their life span and reduces errors.

The three expressions of the `for`-loop are optional; an infinite loop can be created as follows:

```
for ( ; ; ) { // infinite loop

    // your code goes here
}
```

The `for`-statement also has another form designed for iteration through Collections and arrays. This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following programme, [EnhancedForDemo](#), uses the enhanced `for` to loop through the array:

```
class EnhancedForDemo {
public static void main(String[] args){
int[] numbers = {1,2,3,4,5,6,7,8,9,10};

    for (int item : numbers) {
        System.out.println("Count is: " + item);
    }
}
```

In this example, the variable `item` holds the current value from the `numbers` array. The output from this programme is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the `for`-statement instead of the general form whenever possible.

SELF ASSESSMENT EXERCISE 1

Explain the If-Then-Statement.

SELF ASSESSMENT EXERCISE 2

Distinguish between the Do-While statement and the While statement

4.0 CONCLUSION

The `if-then` statement is the most basic of all the control flow statements. It tells your programme to execute a certain section of code *only if* a particular test evaluates to `true`. The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. Unlike `if-then` and `if-then-else`, the `switch` statement allows for any number of possible execution paths.

The `while` and `do-while` statements continually execute a block of statements while a particular condition is `true`. The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once. The `for`-statement provides a compact way to iterate over a range of values.

5.0 SUMMARY

What you have learned in this unit concerns control flow statements.

6.0 TUTOR-MARKED ASSIGNMENT

How do you write an infinite loop using the `for`-statement?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

MODULE 4 JAVA PROGRAMMING

Unit 1	Classes
Unit 2	Objects
Unit 3	Interfaces and Inheritances
Unit 4	Numbers and Strings
Unit 5	Generics

UNIT 1 CLASSES**CONTENTS**

1.0	Introduction
2.0	Objectives
3.0	Main Content
3.1	Classes
3.2	Declaring Classes
3.3	Declaring Member Variables
3.4	Access Modifiers
3.5	Types
3.6	Variable Names
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References/Further Readings

1.0 INTRODUCTION

With the knowledge you now have of the basics of the Java programming language, you can learn to write your own classes. In this unit, you will find information about defining your own classes, including declaring member variables, methods, and constructors. This unit also covers nesting classes within other classes, enumerations, and annotations

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- define your own classes
- describe how to declare member variables
- explain how to nest classes within other classes.

3.0 MAIN CONTENT

3.1 Classes

The introduction to object-oriented concepts in the unit titled [Object-Oriented Programming Concepts](#), used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as

subclasses. Here is a sample code for a possible implementation of a `Bicycle` class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear; cadence
        = startCadence; speed =
        startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
public class MountainBike extends Bicycle {
```

// **the MountainBike subclass has one *field***

```
public int seatHeight;
```

// **the MountainBike subclass has one *constructor***

```
public MountainBike(int startHeight, int startCadence, int startSpeed, int
startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

// **the MountainBike subclass has one *method***

```
public void setHeight(int newValue) {
    seatHeight = newValue;
}
```

```
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight`, and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

3.2 Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
```

```
//field, constructor, and method declarations
```

```
}
```

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initialising new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behaviour of the class and its objects.

The preceding class declaration is a minimal one—it contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
//field, constructor, and method declarations
}
```

means that `MyClass` is a subclass of `MySuperClass` and that it implements the `YourInterface` interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become

quite complicated. The modifiers *public* and *private*, which determine what other classes can access `MyClass`, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment, you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalised by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword, *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, `{ }`.

3.3 Declaring Member Variables

There are several kinds of variables:

Member variables in a class—these are called *fields*.

Variables in a method or block of code—these are called *local variables*.

Variables in method declarations—these are called *parameters*.

The `Bicycle` class uses the following lines of code to define its fields:

```
public int cadence;  
public int gear;  
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as `public` or `private`.
2. The field's type.
3. The field's name.

The fields of `Bicycle` are named `cadence`, `gear`, and `speed` and are all of data type integer (`int`). The `public` keyword identifies these fields as public members, accessible by any object that can access the class.

3.4 Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to, a member field. For the moment, consider only `public` and `private`. Other access modifiers will be discussed later.

`public` modifier—the field is accessible from all classes.

`private` modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields `private`. This means that they can only be *directly* accessed from the `Bicycle` class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {  
  
    private int cadence;  
    private int gear;  
    private int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    public int getCadence() {  
        return cadence;  
    }  
  
    public void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    public int getGear() {  
        return gear;  
    }  
  
    public void setGear(int newValue) {  
        gear = newValue;  
    }  
}
```

```
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

3.5 Types

All variables must have a type. You can use primitive types such as `int`, `float`, `boolean`, etc. Or you can use reference types, such as strings, arrays, or objects.

3.6 Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions that were covered in the Language Basics lesson, [Variables—Naming](#).

Note that the same naming rules and conventions are used for method and class names, except that

- the first letter of a class name should be capitalised
- the first (or only) word in a method name should be a verb.

SELF ASSESSMENT EXERCISE 1

Enumerate three kinds of variables.

SELF ASSESSMENT EXERCISE 2

What are parameters?

4.0 CONCLUSION

In this unit, you have learned about classes. You have also been able to understand how to declare classes and member variables.

5.0 SUMMARY

What you have learned borders on classes and their declarations.

6.0 TUTOR-MARKED ASSIGNMENT

What are the components of field declarations?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Pattern in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 2 OBJECTS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Objects
 - 3.2 Creating Objects
 - 3.3 Declaring a Variable to Refer to an Object
 - 3.4 Instantiating a Class
 - 3.5 Initialising an Object
 - 3.6 Referencing an Object's Fields
 - 3.7 Calling an Object's Methods
 - 3.8 The Garbage Collector
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit covers creating and using objects. You will learn how to instantiate an object, and, once instantiated, how to use the `dot` operator to access the object's instance variables and methods.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe how to create objects
- write a programme to create objects
- explain how to initialise objects
- describe the process of garbage collection.

3.0 MAIN CONTENT

3.1 Objects

A typical Java programme creates many objects, which as you know, interact by invoking methods. Through these object interactions, a programme can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small programme, called [CreateObjectDemo](#), that creates three objects: one [Point](#) object and two [Rectangle](#) objects. You will need all three source files to compile this programme.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        //Declare and create a point object
        //and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        //display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        //set rectTwo's position
        rectTwo.origin = originOne;

        //display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        //move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

This programme creates, manipulates, and displays information about various objects. Here's the output:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

The following three sections use the above example to describe the life cycle of an object within a programme. From them, you will learn how to write code that creates and uses objects in your own programmes.

You will also learn how the system cleans up after an object when its life has ended.

3.2 Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the [CreateObjectDemo](#) programme creates an object and assigns it to a variable:

```
Point originOne = new Point(23, 94);  
Rectangle rectOne = new Rectangle(originOne, 100, 200);  
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the [Point](#) class, and the second and third lines each create an object of the [Rectangle](#) class. Each of these statements has three parts (discussed in detail below):

1. **Declaration:** The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation:** The `new` keyword is a Java operator that creates the object.
3. **Initialisation:** The `new` operator is followed by a call to a constructor, which initializes the new object.

3.3 Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:
type name;

This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.


You can also declare a reference variable on its own line. For example:

```
Point originOne;
```

If you declare `originOne` like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the `new` operator, as described in the next section. You must assign an object to `originOne` before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, `originOne`, plus a reference

pointing to nothing):

originOne 

3.4 Instantiating a Class

The `new` operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The `new` operator also invokes the object constructor.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The `new` operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point (23, 94);
```

The reference returned by the `new` operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

```
int height = new Rectangle().height;
```

This statement will be discussed in the next section.

3.5 Initialising an Object

Here's the code for the `Point` class:

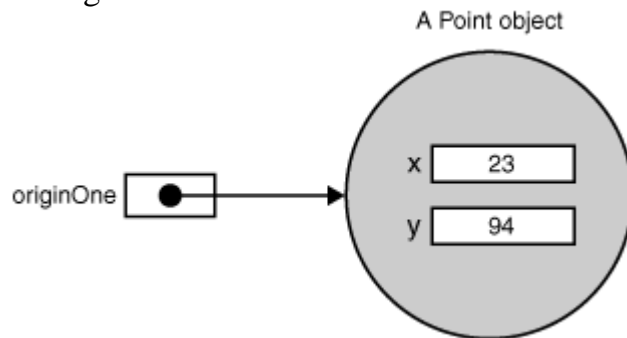
```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

This class contains a single constructor. You can recognise a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the `Point` class takes two integer

arguments, as declared by the code `(int a, int b)`. The following statement provides 23 and 94 as values for those arguments:

```
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:



Here's the code for the `Rectangle` class, which contains four constructors:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
}
```



```
// a method for moving the rectangle
public void move(int x, int y) {
    origin.x = x;
    origin.y = y;
}
```

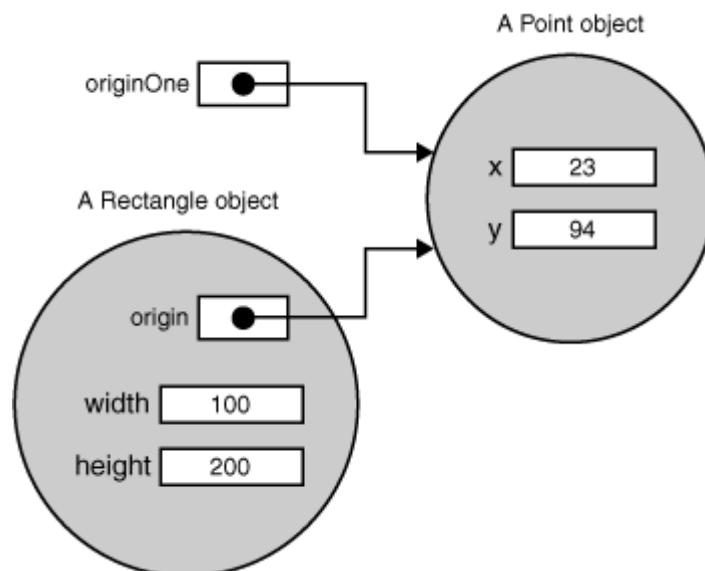
```
// a method for computing the area of the rectangle
```

```
public int getArea() {
    return width * height;
}
}
```

Each constructor lets you provide initial values for the rectangle's size and width, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the `Rectangle` class that requires a `Point` argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of `Rectangle`'s constructors that initialises `origin` to `originOne`. Also, the constructor sets `width` to 100 and `height` to 200. Now, there are two references to the same `Point` object—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the `Rectangle` constructor that requires two integer arguments, which provide the initial values for `width` and `height`. If you inspect the code within the constructor, you

will see that it creates a new `Point` object whose `x` and `y` values are initialised to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The `Rectangle` constructor used in the following statement doesn't take any arguments, so it's called a *no-argument constructor*:

```
Rectangle rect = new Rectangle();
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the *default constructor*. This default constructor calls the class parent's no-argument constructor, or the `Object` constructor if the class has no other parent. If the parent has no constructor (`Object` does have one), the compiler will reject the programme.

Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

3.6 Referencing Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the `Rectangle` class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, `width` and `height` are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (`.`) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the `CreateObjectDemo` class is outside the code for the `Rectangle` class. So to refer to the `origin`, `width`, and `height` fields within the `Rectangle` object named `rectOne`, the `CreateObjectDemo` class must use the names `rectOne.origin`, `rectOne.width`, and `rectOne.height`, respectively. The programme uses two of these names to display the width and the height of `rectOne`:

```
System.out.println("Width of rectOne: " + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

Attempting to use the simple names - `width` and `height`, from the code in the `CreateObjectDemo` class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the programme uses similar code to display information about `rectTwo`. Objects of the same type have their own copy of the same instance fields. Thus, each `Rectangle` object has fields named `origin`, `width`, and `height`. When you access an instance field through an object reference, you reference that particular object's field. The two objects, `rectOne` and `rectTwo`, in the `CreateObjectDemo` programme have different `origin`, `width`, and `height` fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the `new` operator returns a reference to an object. So you could use the value returned from `new` to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new `Rectangle` object and immediately gets its height. In essence, the statement calculates the default height of a `Rectangle`. Note that after this statement has been executed, the programme no longer has a reference to the created `Rectangle`, because the programme never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

3.7 Calling an Object's Methods

An object reference is used to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (`.`). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
or
objectReference.methodName();
```

The `Rectangle` class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the

CreateObjectDemo code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());
```

```
...
```

```
rectTwo.move(40, 72);
```

The first statement invokes `rectOne`'s `getArea()` method and displays the results. The second line moves `rectTwo` because the

`move()` method assigns new values to the object's `origin.x` and `origin.y`.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The `new` operator returns an object reference, so you can use the value returned from `new` to invoke a new object's methods:

```
new Rectangle(100, 50).getArea()
```

The expression `new Rectangle(100, 50)` returns an object reference that refers to a `Rectangle` object. As shown, you can use the dot notation to invoke the new `Rectangle`'s `getArea()` method to compute the area of the new rectangle.

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

3.8 The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value, `null`. Remember that a programme can have multiple references

to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

SELF ASSESSMENT EXERCISE

What do you understand by the phrase ‘instantiating a class’?

4.0 CONCLUSION

In this unit, you have learned about objects. You have also learned how to create and initialise objects. Finally, you have been able to learn the process of garbage collections.

5.0 SUMMARY

What you have learned in this unit is focused on objects, creating, initialising and using these objects.

6.0 TUTOR-MARKED ASSIGNMENT

Explain the process of garbage collection.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Pattern in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 3 INTERFACES AND INHERITANCE

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Interfaces-Basics
 - 3.2 Interfaces in Java
 - 3.3 Interfaces as APIs
 - 3.4 Interfaces and Multiple Inheritance
 - 3.5 Inheritance
 - 3.6 Definitions
 - 3.7 The Java Platform Class Hierarchy
 - 3.8 What You can Do in a Subclass
 - 3.9 Private Members in a Superclass
 - 3.10 Casting Objects
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit describes the way in which you can derive one class from another. That is, how a *subclass* can inherit fields and methods from a *superclass*. You will learn that all classes are derived from the `Object` class, and how to modify the methods that a subclass inherits from superclasses. This unit also covers interface-like *abstract classes*.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- derive one class from another
- describe how to modify methods that a subclass inherits from superclasses
- define subclass and superclass
- explain what an interface is.

3.0 MAIN CONTENT

3.1 Interface-Basics

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group

should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts. For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile – stop, start, accelerate, turn left, and so forth. Another industrial group, electronic

guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning Satellite) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

3.2 Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {

    // constant declarations, if any

    // method signatures
    int turn(Direction direction, // An enum with values RIGHT, LEFT
             double radius, double startSpeed, double endSpeed);
    int changeLanes(Direction direction, double startSpeed, double
                    endSpeed);
    int signalTurn(Direction direction, boolean signalOn);
    int getRadarFront(double distanceToCar, double speedOfCar);
    int getRadarRear(double distanceToCar, double speedOfCar);
    .....
    // more method signatures
}
```


Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:

    int signalTurn(Direction direction, boolean signalOn) {
        //code to turn BMW's LEFT turn indicator lights on
        //code to turn BMW's LEFT turn indicator lights off
        //code to turn BMW's RIGHT turn indicator lights on
        //code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes
    // not visible to clients of the interface

}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

3.3 Interfaces as APIs

The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programmes. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its

implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

3.4 Interfaces and Multiple Inheritance

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance (inheritance is discussed later in this lesson), but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be a type of an interface, its value can reference any object that is instantiated from any class that implements the interface. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

3.5 Inheritance

In the preceding units, you have seen *inheritance* mentioned several times. In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

3.6 Definitions

A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to `Object`.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In

doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

3.7 The Java Platform Class Hierarchy

The [Object](#) class, defined in the `java.lang` package, defines and implements behaviour common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

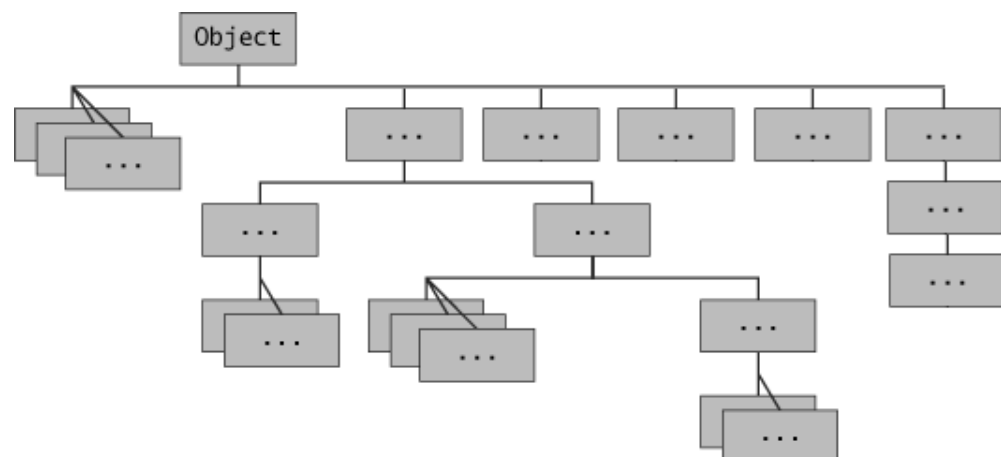


Fig. 1.0: All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialised behaviour.

An Example of Inheritance

Here is the sample code for a possible implementation of a `Bicycle` class that was presented in the `Classes and Objects` lesson:

```
public class Bicycle
{
```

// the Bicycle class has three
fields

```
public      int
cadence;
public      int
gear;
public      int
speed;
```

// the Bicycle class has one
constructor

```
public Bicycle(int startCadence, int startSpeed, int startGear)
{
    gear = startGear;
    cadence =
startCadence; speed =
startSpeed;
}
```

// the Bicycle class has four *methods*

```
public void setCadence(int newValue) {
    cadence = newValue;
}
```

```
public void setGear(int newValue) {
    gear = newValue;
}
```

```
public void applyBrake(int decrement) {
    speed -= decrement;
}
```

```
public void speedUp(int increment) {
    speed += increment;
}
```

```
}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {
```

// the MountainBike subclass adds one *field*

```
public int seatHeight;
```

// **the MountainBike subclass has one *constructor***

```
public MountainBike(int startHeight, int startCadence, int startSpeed, int
startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

// **the MountainBike subclass adds one *method***

```
public void setHeight(int newValue) {
    seatHeight = newValue;
}
```

```
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the

methods in the Bicycle class were complex and had taken substantial time to debug.

3.8 What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package – private* members of the parent. You can use the inherited members as follows: replace them, hide them, or supplement them with new members:

The inherited fields can be used directly, just like any other field.

You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).

You can declare new fields in the subclass that are not in the superclass.

The inherited methods can be used directly as they are.

You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.

You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.

You can declare new methods in the subclass that are not in the superclass.

You can write a subclass constructor that invokes the constructor of the

superclass, either implicitly or by using the keyword, `super`.

The following sections in this unit will expand on these topics.

3.9 Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

3.10 Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write
`public MountainBike myBike = new MountainBike();`
 then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
then obj is both an Object and a Mountainbike (until such time
as obj is assigned another object that is not a Mountainbike). This
is called implicit casting.
```

If, on the other hand, we write

```
MountainBike myBike = obj;
we would get a compile-time error because obj is not known to the
compiler to be a MountainBike. However, we can tell the compiler that
we promise to assign a MountainBike to obj by explicit
casting:
```

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `Mountainbike` at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here, the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

SELF ASSESSMENT EXERCISE 1

Is the following interface valid?

```
public interface Marker {
}
```

SELF ASSESSMENT EXERCISE 2

What do you understand by the term, interface?

4.0 CONCLUSION

In this unit you have learned about inheritance. You have also learned about interfaces. Finally, you have been able to learn how to modify methods.

5.0 SUMMARY

What you have learned in this unit borders on inheritance and interfaces.

6.0 TUTOR-MARKED ASSIGNMENT

What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
        System.out.println("Hi Mom");
    }
}
```

}

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Pattern in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 NUMBERS AND STRINGS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 The Number Classes
 - 3.2 Creating Strings
 - 3.3 String Length
 - 3.4 Concatenating Strings
 - 3.5 Creating Format Strings
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit begins with a discussion of the `Number` class in the `java.lang` package, its subclasses, and the situations where you would use instantiations of these classes rather than the primitive number types. It also presents the `PrintStream` and [`DecimalFormat`](#) classes, which provide methods for writing formatted numerical output. Finally, the `Math` class in `java.lang` is discussed.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe number classes
- explain how to create strings
- explain concatenating of strings.

3.0 MAIN CONTENT

3.1 The Number Classes

When working with numbers, most of the time you use the primitive types in your code. For example:

```
int i = 500;  
float gpa = 3.65;  
byte mask = 0xff;
```

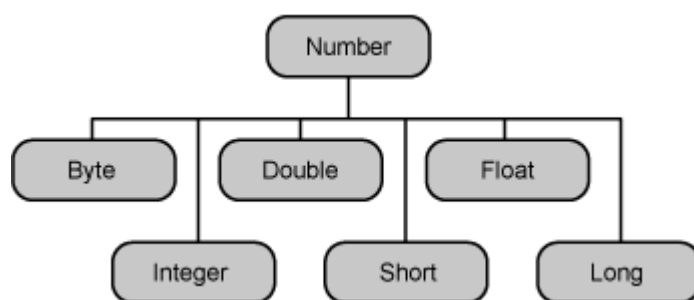
There are, however, reasons to use objects in place of primitives, and the

Java platform provides *wrapper* classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler *boxes* the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler *unboxes* the object for you.

Here is an example of boxing and unboxing:

```
Integer x, y;  
x = 12; y = 15;  
System.out.println(x+y);
```

When `x` and `y` are assigned integer values, the compiler boxes the integers because `x` and `y` are integer objects. In the `println()` statement, `x` and `y` are unboxed so that they can be added as integers. All of the numeric wrapper classes are subclasses of the abstract class `Number`:



Note: There are four other subclasses of `Number` that are not discussed here. `BigDecimal` and `BigInteger` are used for high-precision calculations. `AtomicInteger` and `AtomicLong` are used for multi-threaded applications.

There are three reasons why you might use a `Number` object rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

The following table lists the instance methods that all the subclasses of the `Number` class implement.

Methods Implemented by all Subclasses of `Number`

Method	Description
<code>byte byteValue()</code> <code>short shortValue()</code> <code>int intValue()</code> <code>long longValue()</code> <code>float floatValue()</code> <code>double doubleValue()</code>	Converts the value of this <code>Number</code> object to the primitive data type returned.
<code>int compareTo(Byte anotherByte)</code> <code>int compareTo(Double anotherDouble)</code> <code>int compareTo(Float anotherFloat)</code> <code>int compareTo(Integer anotherInteger)</code> <code>int compareTo(Long anotherLong)</code> <code>int compareTo(Short anotherShort)</code>	Compares this <code>Number</code> object to the argument.
<code>boolean equals(Object obj)</code>	<p>Determines whether this number object is equal to the argument. The methods return <code>true</code> if the argument is not <code>null</code> and is an object of the same type and with the same numeric value.</p> <p>There are some extra requirements for <code>Double</code> and <code>Float</code> objects that are described in the Java API documentation.</p>

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Conversion Methods, Integer Class

Method	Description
<code>static Integer decode(String s)</code>	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.

<code>static int parseInt(String s)</code>	Returns an integer (decimal only).
<code>static int parseInt(String s, int radix)</code>	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.
<code>String toString()</code>	Returns a <code>String</code> object representing the value of this <code>Integer</code> .
<code>static String toString(int i)</code>	Returns a <code>String</code> object representing the specified integer.
<code>static Integer valueOf(int i)</code>	Returns an <code>Integer</code> object holding the value of the specified primitive.
<code>static Integer valueOf(String s)</code>	Returns an <code>Integer</code> object holding the value of the specified string representation.
<code>static Integer valueOf(String s, int radix)</code>	Returns an <code>Integer</code> object holding the integer value of the specified string representation, parsed with the value of radix. For example, if <code>s = "333"</code> and <code>radix = 8</code> , the method returns the base-ten integer equivalent of the octal number 333.

3.2 Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o',  
                      '.'};
```

```
String helloString = new String(helloArray);
```

```
System.out.println(helloString);
```

The last line of this code snippet displays hello.

Note: The `String` class is immutable, so that once it is created, a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

3.3 String Length

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, `len` equals 17:

```
String palindrome = "Dot saw I was Tod";  
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient programme to reverse a palindrome string. It invokes the `String` method `charAt(i)`, which returns the i^{th} character in the string, counting from 0.

```
public class StringDemo {  
    public static void main(String[] args) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        char[] tempCharArray = new char[len];  
        char[] charArray = new char[len];  
  
        // put original string in an array of chars  
        for (int i = 0; i < len; i++) {  
            tempCharArray[i] = palindrome.charAt(i);  
        }  
        // reverse array of chars  
        for (int j = 0; j < len; j++) {  
            charArray[j] = tempCharArray[len - 1 - j];  
        }  
    }  
}
```

```

String reversePalindrome = new
String(charArray);
System.out.println(reversePalindrome);

}
}

```

Running the programme produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the programme had to convert the string to an array of characters (first `for` loop), reverse the array into a second array (second `for` loop), and then convert back to a string. The [String](#) class includes a method, `getChars()`, to convert a string, or a portion of a string, into an array of characters so we could replace the first `for` loop in the programme above with

```
palindrome.getChars(0, len - 1, tempCharArray,
0);
```

3.4 Concatenating Strings

The `String` class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is `string1` with `string2` added to it at the end.

You can also use the `concat()` method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the `+` operator, as in

```
"Hello, " + " world" + "!"
```

which results in

```
"Hello, world!"
```

The `+` operator is widely used in print statements. For example:

```
String string1 = "saw I was ";
```

```
System.out.println("Dot " + string1 + "Tod");
```

which prints

```
Dot saw I was Tod
```

Such a concatenation can be a mixture of any objects. For each object that is not a `String`, its `toString()` method is called to convert it to a `String`.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the `+` concatenation operator at the end of each line in a multi-line string. For example,

```
String quote = "Now is the time for all good" +
"men to come to the aid of their country.";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in `print` statements.

3.5 Creating Format Strings

You have seen the use of the `printf()` and `format()` methods to print output with formatted numbers. The `String` class has an equivalent class method, `format()`, that returns a `String` object rather than a `PrintStream` object.

Using `String`'s static `format()` method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```
System.out.printf("The value of the float  
variable is %f, while the value of the " +  
"integer variable is %d, and the string is %s",  
floatVar, intVar, stringVar);
```

you can write

```
String fs;  
fs = String.format("The value of the float  
variable is %f, while the value of the " +  
"integer variable is %d, and the string is %s",  
floatVar, intVar, stringVar);
```

```
System.out.println(fs);
```

SELF ASSESSMENT EXERCISE 1

What would be the result of concatenating these strings with the + operator?

```
"Hello," + " world" + "!"
```

SELF ASSESSMENT EXERCISE 2

State three reasons why might use a `Number` object rather than a primitive

4.0 CONCLUSION

In this unit you have learned about number classes. You have also learned how to create and concatenate strings.

5.0 SUMMARY

What you have learned in this unit is focused on numbers and strings.

6.0 TUTOR-MARKED ASSIGNMENT

What `Integer` method would you use to convert a string expressed in base 5 into the equivalent `int`? For example, how would you convert the string "230" into the integer value 65?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Pattern in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 **GENERIC**S

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Generics—Basics
 - 3.2 A Simple Box Class
 - 3.3 Generic Types
 - 3.4 Type Parameter Naming Conventions
 - 3.5 Generic Methods and Constructors
 - 4.0 Conclusion
 - 5.0 Summary
 - 6.0 Tutor-Marked Assignment
 - 7.0 References/Further Readings

1.0 INTRODUCTION

This unit will focus primarily on simple "collections-like" examples that we'll design from scratch. This hands-on approach will teach you the necessary syntax and terminology while demonstrating the various kinds of problems that generics were designed to solve.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the notion of generics
- identify generic types
- list the type parameter naming conventions
- describe generic methods.

3.0 MAIN CONTENT

3.1 Generics – Basics

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, tell you immediately that something is wrong; you can use the compiler's error messages to figure out what the problem is

and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in time that's far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time. Some programmers choose to learn generics by studying the Java Collections Framework; after all, generics *are* heavily used by those classes.

3.2 A Simple Box Class

Let's begin by designing a nongeneric `Box` class that operates on objects of any type. It need only provide two methods: `add`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {  
  
    private Object object;  
  
    public void add(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}
```

Since its methods accept or return `Object`, you're free to pass in whatever you want, provided that it's not one of the primitive types. However, should you need to restrict the contained type to something specific (like `Integer`), your only option would be to specify the requirement in documentation (or in this case, a comment), which of course the compiler knows nothing about:

```
public class BoxDemo1 {  
  
    public static void main(String[] args) {  
  
        // ONLY place Integer objects into this box!  
        Box integerBox = new Box();  
  
        integerBox.add(new Integer(10));  
        Integer someInteger = (Integer)integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```

```

}
}

```

The [BoxDemo1](#) programme creates an `Integer` object, passes it to `add`, then assigns that same object to `someInteger` by the return value of `get`. It then prints the object's value (10) to standard output. We know that the cast from `Object` to `Integer` is correct because we have honoured the "contract" specified in the comment. But remember, the compiler knows nothing about this — it just trusts that

our cast is correct. Furthermore, it will do nothing to prevent a careless programmer from passing in an object of the wrong type, such as `String`:

```

public class BoxDemo2 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large
        // application
        // modified by one programmer.

        integerBox.add("10"); // note how the type is
        // now String

        // ... and this is another, perhaps written
        // by a different programmer
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}

```

In [BoxDemo2](#) we've stored the number 10 as a `String`, which could be the case when, say, a GUI collects input from the user. However, the existing cast from `Object` to `Integer` has mistakenly been overlooked. This is clearly a bug, but because the code still compiles, you wouldn't know anything is wrong until runtime, when the application crashes with a `ClassCastException`:

```

Exception in thread "main"
java.lang.ClassCastException:
java.lang.String cannot be cast to
java.lang.Integer

```

```
at BoxDemo2.main(BoxDemo2.java:6)
```

If the `Box` class had been designed with generics in mind, this mistake would have been caught by the compiler, instead of crashing the application at runtime.

3.3 Generic Types

Let's update our `Box` class to use generics. We'll first create a *generic type declaration* by changing the code `"public class Box"` to `"public class Box<T>"`; this introduces one *type variable*, named

`T`, that can be used anywhere inside the class. This same technique can be applied to interfaces as well. There's nothing particularly complex about this concept. In fact, it's quite similar to what you already know about variables in general. Just think of `T` as a special kind of variable, whose "value" will be whatever type you pass in; this can be any class type, any interface type, or even another type variable. It just can't be any of the primitive data types. In this context, we also say that `T` is a *formal type parameter* of the `Box` class.

```
/**
 * Generic version of the Box class.
 */
public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

As you can see, we've replaced all occurrences of `Object` with `T`. To reference this generic class from within your own code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you're passing a *type argument* — `Integer` in this case — to

the `Box` class itself. Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "Box of Integer", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
integerBox = new Box<Integer>();
```

Or, you can put the entire statement on one line, such as:

```
Box<Integer> integerBox = new Box<Integer>();
```

Once `integerBox` is initialised, you're free to invoke its `get` method without providing a cast, as in [BoxDemo3](#):

```
public class BoxDemo3 {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no
        cast!
        System.out.println(someInteger);
    }
}
```

Furthermore, if you try adding an incompatible type to the box, such as `String`, compilation will fail, alerting you to what previously would have been a runtime bug:

```
BoxDemo3.java:5:      add(java.lang.Integer)      in
Box<java.lang.Integer>
cannot be applied to (java.lang.String)
integerBox.add("10");
                ^
1 error
```

It's important to understand that type variables are not actually types themselves. In the above examples, you won't find `T.java` or `T.class` anywhere on the filesystem. Furthermore, `T` is not a part of the `Box` class name. In fact, during compilation, all generic information

will be removed entirely, leaving only `Box.class` on the filesystem. We'll discuss this later in the section on **Type Erasure**.

Also note that a generic type may have multiple type parameters, but each parameter must be unique within its declaring class or interface. A declaration of `Box<T, T>`, for example, would generate an error on the second occurrence of `T`, but `Box<T, U>`, however, would be allowed.

3.4 Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable [naming](#) conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V etc. - 2nd, 3rd, 4th types.

You will see these names used throughout the Java SE API and the rest of this tutorial.

3.5 Generic Methods and Constructors

Type parameters can also be declared within method and constructor signatures to create *generic methods* and *generic constructors*. This is similar to declaring a generic type, but the type parameter's scope is limited to the method or constructor in which it's declared.

```
/**
 * This version introduces a generic method.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

```

}
public <U> void inspect(U u) {
    System.out.println("T: " +
        t.getClass().getName());
    System.out.println("U: " +
        u.getClass().getName());
}

public static void main(String[] args) {

    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    integerBox.inspect("some text");
}
}

```

Here, we have added one generic method, named `inspect`, that defines one type parameter, named `U`. This method accepts an object and prints its type to standard output. For comparison, it also prints out the type of `T`. For convenience, this class now also has a `main` method so that it can be run as an application.

The output from this programme is:

```

T: java.lang.Integer
U: java.lang.String

```

By passing in different types, the output will change accordingly.

A more realistic use of generic methods might be something like the following, which defines a static method that stuffs references to a single item into multiple boxes:

```

public static <U> void fillBoxes(U u,
    List<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    }
}

```

To use this method, your code would look something like the following:

```

Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;

```

The complete syntax for invoking this method is:


```
Box.<Crayon>fillBoxes(red, crayonBoxes);
```

Here, we have explicitly provided the type to be used as U, but more often than not, this can be left out and the compiler will infer the type that's needed:

```
Box.fillBoxes(red, crayonBoxes); // compiler
infers that U is Crayon
```

This feature, known as *type inference*, allows you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets.

SELF ASSESSMENT EXERCISE

List at least three commonly used type parameter names.

4.0 CONCLUSION

Specifically, you learned that generic type declarations can include one or more type parameters; you supply one type argument for each type parameter when you use the generic type. You also learned that type parameters can be used to define generic methods and constructors. Bounded type parameters limit the kinds of types that can be passed into a type parameter; they can specify an upper bound only. Wildcards represent unknown types, and they can specify an upper or lower bound.

5.0 SUMMARY

What you have learned in this unit is focused on generics, their methods and constructors.

6.0 TUTOR-MARKED ASSIGNMENT

Distinguish between the type parameter and the variable naming conventions.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Pattern in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

MODULE 5 ALGORITHMS

Unit 1	Introduction to Algorithms
Unit 2	Vectors and Matrices
Unit 3	Greedy Algorithm
Unit 4	Divide-and-Conquer Algorithm
Unit 5	Dynamic Programming Algorithm

UNIT 1 INTRODUCTION TO ALGORITHMS**CONTENTS**

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 What is an Algorithm?
 - 3.2 Algorithm's Performance
 - 3.3 Algorithm Analysis
 - 3.3.1 Worst-Case Complexity
 - 3.3.2 Average-Case Complexity
 - 3.4 Optimality
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit will introduce you to algorithms, their performance and analysis. You will also be introduced to the concept of an optimal algorithm.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- define an algorithm
- explain an algorithm's performance
- describe algorithm analysis
- explain the notion of an optimal algorithm.

3.0 MAIN CONTENT

3.1 What is an Algorithm?

An algorithm can be defined as a finite step-by-step procedure to achieve a required result.

In terms of data structures, an algorithm can be described as a sequence of operations performed on data that have to be organised in data

structures. An algorithm is also an abstraction of a programme to be executed on a physical machine (model of Computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is Euclid's algorithm for calculating the greatest common divisor of two integers.

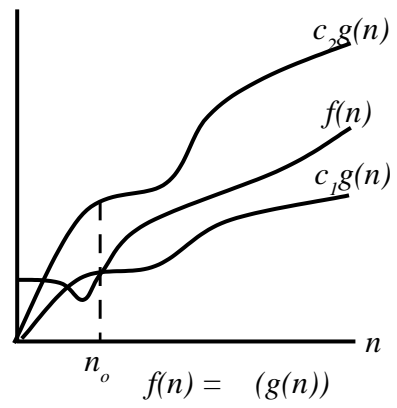
3.2 Algorithm's Performance

Two important ways to characterise the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the \mathbf{O} (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are $\mathbf{\Theta}$ (Big Theta) notation and $\mathbf{\Omega}$ (Big Omega) notation. The performance evaluation of an algorithm is obtained by totalling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size, n , and is to be considered modulo, a multiplicative constant.

The following notations are commonly used notations in performance analysis and used to characterize the complexity of an algorithm.

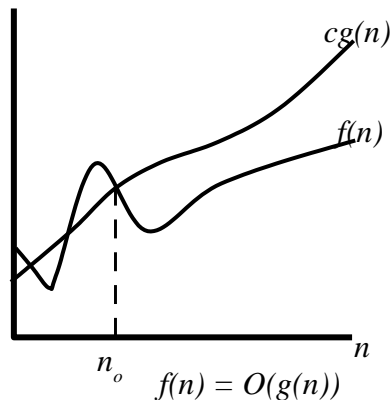
$\mathbf{\Theta}$ -Notation (Same order)

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



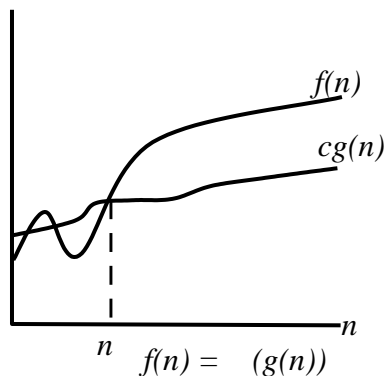
O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$.



Ω -Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.



3.3 Algorithm Analysis

Analysis of algorithms is a field in computer science whose overall goal is an understanding of the complexity of algorithms. The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n .

There are two interpretations of upper bound.

3.3.1 Worst-case Complexity

The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.

3.3.2 Average-case Complexity

The running time for any given size input will be the average number of operations over all problem instances for a given size.

Because, it is quite difficult to estimate the statistical behaviour of the input, we mostly content ourselves to a worst case behaviour. Most of the time, the complexity of $g(n)$ is approximated by its family $o(f(n))$ where $f(n)$ is one of the following functions. n (linear complexity), $\log n$ (logarithmic complexity), n^a where $a \geq 2$ (polynomial complexity), a^n (exponential complexity).

3.4 Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem. For example, any algorithm solving “the intersection of n segments” problem will execute at least n^2 operations in the worst case even if it does nothing but print the output. This is abbreviated by saying that the problem has $\Omega(n^2)$ complexity. If one finds an $O(n^2)$ algorithm that solves this problem, it will be optimal and of complexity $\Theta(n^2)$.

SELF ASSESSMENT EXERCISE 1

What do you understand by algorithm analysis?

SELF ASSESSMENT EXERCISE 2

List three notations used to characterise the complexity of an algorithm.

4.0 CONCLUSION

In this unit you have learned about algorithms, their performance and analysis. You have also been able to understand the optimality of an algorithm.

5.0 SUMMARY

What you have learned borders on algorithms, their performance and analysis.

6.0 TUTOR-MARKED ASSIGNMENT

When is an algorithm said to be optimal?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

<http://java.sun.com/docs/books/tutorial/java/concepts/index.html>

<http://cs.wvc.edu/~aabyan/OOP/>

UNIT 2 VECTORS AND MATRICES**CONTENTS**

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Vectors
 - 3.2 Addition of Two Vectors
 - 3.3 Multiplication of a Vector by a Scalar
 - 3.4 Dot Product and Norm
 - 3.5 Matrices
 - 3.6 Matrix Addition
 - 3.7 Matrix Multiplication
 - 3.8 Transpose
 - 4.0 Conclusion
 - 5.0 Summary
 - 6.0 Tutor-Marked Assignment
 - 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will learn about vectors and matrices. Simple arithmetic operations will also be carried out on the vectors and matrices.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- identify a vector
- identify a matrix
- add and multiply vectors
- add and multiply matrices
- determine the transpose of a matrix.

3.0 MAIN CONTENT**3.1 Vectors**

A vector, u , means a list (or n-tuple) of numbers:

$$u = (u_1, u_2, \dots, u_n)$$

where u_i are called the components of u . If all the u_i are zero i.e., $u_i = 0$, then u is called the zero vector.

Given vectors u and v are equal i.e., $u = v$, if they have the same number of components and if corresponding components are equal.

3.2 Addition of Two Vectors

If two vectors, u and v , have the number of components, their sum, $u + v$, is the vector obtained by adding corresponding components from u and v .

$$\begin{aligned} u + v &= (u_1, u_2, \dots, u_n) + (v_1, v_2, \dots, v_n) \\ &= (u_1 + v_1, u_2 + v_2, \dots, u_n + v_n) \end{aligned}$$

3.3 Multiplication of a Vector by a Scalar

The product of a scalar, k , and a vector, u , i.e., ku , is the vector obtained by multiplying each component of u by k :

$$\begin{aligned} ku &= k(u_1, u_2, \dots, u_n) \\ &= ku_1, ku_2, \dots, ku_n \end{aligned}$$

Here, we define $-u = (-1)u$ and $u - v = u + (-v)$

It is not difficult to see $k(u + v) = ku + kv$ where k is a scalar and u and v are vectors.

3.4 Dot Product and Norm

The dot product or inner product of vectors $u = (u_1, u_2, \dots, u_n)$ and $v = (v_1, v_2, \dots, v_n)$ is denoted by $u \cdot v$ and defined by

$$u \cdot v = u_1v_1 + u_2v_2 + \dots + u_nv_n$$

The norm or length of a vector, u , is denoted by $\|u\|$ and defined by

$$\|u\| = \frac{\sqrt{u \cdot u}}{\sqrt{u_1^2 + u_2^2 + \dots + u_n^2}}$$

3.5 Matrices

Matrix, A , means a rectangular array of numbers.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

The m horizontal n -tuples are called the rows of A , and the n vertical m -tuples, its columns. Note that the element, a_{ij} , called the ij -entry, appear in the i^{th} row and the j^{th} column.

In algorithmic (study of algorithms), we like to write a matrix A , as $A(a_{ij})$.

3.6 Matrix Addition

Let A and B be two matrices of the same size. The sum of A and B is written as $A + B$ and obtained by adding corresponding elements from A and B .

$$\begin{aligned}
 A+B &= \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix} \\
 &= \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{pmatrix}
 \end{aligned}$$

3.7 Matrix Multiplication

Suppose A and B are two matrices such that the number of columns of A is equal to number of rows of B . Say matrix A is an $m \times p$ matrix and matrix B is a $p \times n$ matrix. Then the product of A and B is the $m \times n$ matrix whose ij -entry is obtained by multiplying the elements of the i th row of A by the corresponding elements of the j th column of B and then adding them.

It is important to note that if the number of columns of A is not equal to the number of rows of B , then the product, AB , is not defined.

3.8 Transpose

The transpose of a matrix A is obtained by writing the row of A , in order, as columns and denoted by A^T . In other words, if $A = (a_{ij})$, then $B = (b_{ij})$ is the transpose of A if $b_{ij} = a_{ji}$ for all i and j .

It is not hard to see that if A is an $m \times n$ matrix, then A^T is an $n \times m$ matrix.

For example if $A =$, then $A^T =$

SELF ASSESSMENT EXERCISE

Find the product of any two matrices of your choice.

4.0 CONCLUSION

In this unit, you have learned about vectors and matrices. You have also learned how to carry out addition and multiplication on vectors and matrices.

5.0 SUMMARY

You have considered vectors and matrices in this unit.

6.0 TUTOR-MARKED ASSIGNMENT

Find the transpose of the matrix

784
653

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

<http://www.purplemath.com/modules/variable.htm>

UNIT 3 GREEDY ALGORITHM

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Greedy Algorithm Overview
 - 3.2 Greedy Algorithm Approach
 - 3.3 Features of Problems Solved by Greedy Algorithm
- 3.4 Structure Greedy Algorithm
- 3.5 Definitions of Feasibility
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

What you will learn in this unit borders on greedy algorithms. The greedy algorithm approach and functions will equally be discussed.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the merits of greedy algorithm
- describe greedy algorithm approach
- list four functions of greedy algorithm.

3.0 MAIN CONTENT

3.1 Greedy Algorithm – Overview

Greedy algorithms are simple and straightforward algorithms. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time, quite efficient. Greedy algorithms are used to solve optimization problems.

3.2 Greedy Algorithm Approach

Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later.

3.3 Features of Problems solved by Greedy Algorithms

To construct the solution in an optimal way, an algorithm maintains two sets. One contains chosen items and the other contains rejected items.

The greedy algorithm consists of four (4) functions.

1. A function that checks whether chosen set of items provide a solution.
2. A function that checks the feasibility of a set.
3. The selection function tells which of the candidates is the most promising.
4. An objective function, which does not appear explicitly, gives the value of a solution.

3.4 Structure Greedy Algorithm

- i. Initially the set of chosen items is empty i.e., solution set.
- ii. At each step

item will be added in a solution set by using selection function.

IF the set would no longer be feasible

- reject items under consideration (and is never considered again).

ELSE IF set is still feasible THEN

- add the current item.

3.5 Definitions of Feasibility

A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem. In particular, the empty set is always promising why? (because an optimal solution always exists).

Unlike Dynamic Programming, which solves the subproblems bottom-up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each problem to a smaller one.

SELF ASSESSMENT EXERCISE

List four functions of greedy algorithms.

4.0 CONCLUSION

In this unit, you have learned about greedy algorithms. You have also been able to identify a promising feasible set.

5.0 SUMMARY

What you have learned in this unit concerns greedy algorithms.

6.0 TUTOR-MARKED ASSIGNMENT

When is a feasible set said to be promising?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 DIVIDES AND CONQUER ALGORITHM

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Divide-and-Conquer Algorithm
 - 3.2 Binary Search
 - 3.3 Sequential Search
 - 3.4 Analysis
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit introduces the divide-and-conquer algorithm as a design technique. It explains the phases involved in this technique of design.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe the divide-and-conquer design technique
- explain the phases involved in the divide-and-conquer paradigm
- describe the application of divide-and-conquer.

3.0 MAIN CONTENT

3.1 Divide-and-Conquer Algorithm

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of the following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,

- Solve the sub-problem recursively (successively and independently), and then

- Combine these solutions to subproblems to create a solution to the original problem.

3.2 Binary Search (simplest application of divide-and-conquer)

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element, we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reach the required (given) element.

Problem Let $A[1 \dots n]$ be an array of non-decreasing sorted order;

that is $A[i] \leq A[j]$ whenever $1 \leq i \leq j \leq n$. Let ' q ' be the query point. The problem consists of finding ' q ' in the array A . If q is not in A , then find the position where ' q ' might be inserted.

Formally, find the index i such that $1 \leq i \leq n+1$ and $A[i-1] < x \leq A[i]$.

3.3 Sequential Search

Look sequentially at each element of A until either we reach the end of an array A or find an item no smaller than ' q '.

Sequential search for ' q ' in array A

```
for i = 1 to n do
  if  $A[i] \geq q$  then
    return index i
return n + 1
```

3.4 Analysis

This algorithm clearly takes a $\theta(r)$, where r is the index returned. This is $\Omega(n)$ in the worst case and $O(1)$ in the best case.

If the elements of an array A , are distinct and query point q is indeed in the array, then loop executed $(n + 1) / 2$ average number of times. On average (as well as the worst case), sequential search takes $\theta(n)$ time.

SELF ASSESSMENT EXERCISE

Describe at least one application of divide-and-conquer.

4.0 CONCLUSION

In this unit, you have learned about divide-and-conquer algorithm. You have also gained knowledge of binary and sequential search.

5.0 SUMMARY

What you have learned in this unit concerns divide-and-conquer algorithm.

6.0 TUTOR-MARKED ASSIGNMENT

Divide-and-conquer is a top-down design technique. True or False? Discuss.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 ALGORITHMS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Dynamic Programming
 - 3.2 The Principle of Optimality
- 3.3 Dynamic Programming Algorithm
- 3.4 Analysis
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit introduces dynamic programming as opposed to the divide-and-conquer. It explains the bottom-up technique and states the principle of optimality.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the dynamic programming method
- identify four design steps in dynamic programming
- distinguish between divide-and-conquer and dynamic programming
- explain the concept of bottom-up
- state the principle of optimality.

3.0 MAIN CONTENT

3.1 Dynamic Programming

Dynamic programming is a stage-wise search method suitable for optimisation problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution, it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Dynamic programming design involves four major steps:

1. Develop a mathematical notation that can express any solution and subsolution for the problem at hand.
2. Prove that the Principle of Optimality holds.
3. Develop a recurrence relation that relates a solution to its subsolutions, using the math notation of step 1. Indicate what the initial values are for that recurrence relation, and which term signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

Dynamic programming takes advantage of the duplication and arrange to solve each subproblem only once, saving the solution (in table or something) for later use. The underlying idea of dynamic programming is: avoid calculating the same stuff twice, usually by keeping a table of known results of subproblems. Unlike divide-and-conquer, which solves the subproblems top-down, a dynamic programming is a bottom-up technique.

Bottom-up means

- i. Start with the smallest subproblems.
- ii. Combining these solutions, obtain the solutions to subproblems of increasing size.
- iii. Until the solution of the original problem is arrived at.

3.2 The Principle of Optimality

The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also optimal.

The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its subproblems.

The difficulty in turning the principle of optimality into an algorithm is that it is not usually obvious which subproblems are relevant to the problem under consideration.

3.3 Dynamic-Programming Algorithm

The finishing times are in a sorted array $f[i]$ and the starting times are in array $s[i]$. The array $m[i]$ will store the value m_i , where m_i is the size of the largest of mutually compatible activities among activities $\{1, 2, \dots, i\}$. Let $\text{BINARY-SEARCH}(f, s)$ returns the index of a number i in the sorted array f such that $f(i) \leq s \leq f[i + 1]$.

```

for  $i = 1$  to  $n$ 
  do  $m[i] = \max(m[i-1], 1 + m[\text{BINARY-SEARCH}(f, s[i])])$ 
  We have  $P[i] = 1$  if activity  $i$  is in optimal selection, and  $P[i] = 0$ 
  otherwise

```

```

 $i = n$ 
while  $i > 0$ 
do if  $m[i] = m[i-1]$ 
then  $P[i] = 0$ 
   $i = i - 1$ 
else
   $i = \text{BINARY-SEARCH}(f, s[i])$ 
   $P[i] = 1$ 

```

3.4 Analysis

The running time of this algorithm is $O(n \lg n)$ because of the binary search which takes $\lg(n)$ time as opposed to the $O(n)$ running time of the greedy algorithm. This greedy algorithm assumes that the activities are already sorted by increasing time.

SELF ASSESSMENT EXERCISE 1

List four design steps in dynamic programming.

SELF ASSESSMENT EXERCISE 2

State the principle of optimality.

4.0 CONCLUSION

In this unit, you have learned about dynamic programming. You have also gained insight of bottom-up technique and the principle of optimality.

5.0 SUMMARY

What you have learned in this unit concerns dynamic programming and its analysis.

6.0 TUTOR-MARKED ASSIGNMENT

Explain the dynamic programming method.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

MODULE 6 GRAPHS AND SORTING

Unit 1	Graph Algorithm
Unit 2	Sorting
Unit 3	Bubble Sort
Unit 4	Insertion Sort
Unit 5	Selection Sort
Unit 6	Merge Sorting

UNIT 1 GRAPH ALGORITHM**CONTENTS**

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 The Graph Theory
 - 3.2 Digraph
 - 3.3 Algorithm Transpose
 - 3.4 Algorithm Matrix Transpose
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, the student will gain knowledge of the graph theory and its applications. The unit describes the digraph and determines the transpose of an algorithm.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe the graph theory, stating some of its applications
- gain knowledge of a digraph
- explain the algorithmic transpose.

3.0 MAIN CONTENT

3.1 The Graph Theory

Graph Theory is an area of mathematics that deals with the following types of problems:

- Connection problems
- Scheduling problems
- Transportation problems
- Network analysis
- Games and Puzzles.

However, the graph theory has important applications in Critical path analysis, Social psychology, Matrix theory, Set theory, Topology, Group theory, Molecular Chemistry, and Searching.

3.2 Digraph

A directed graph, or digraph, G , consists of a finite nonempty set of vertices V , and a finite set of edges E , where an edge is an ordered pair of vertices in V . Vertices are also commonly referred to as nodes. Edges are sometimes referred to as arcs.

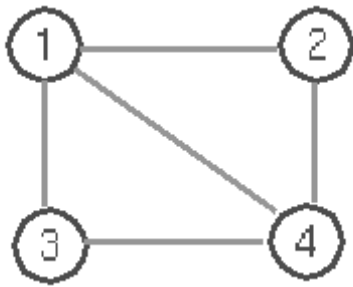
As an example, we could define a graph $G=(V, E)$ as follows:

$$V = \{1, 2, 3, 4\}$$

$$E = \{ (1, 2), (2, 4), (4, 2) (4, 1) \}$$

Here is a pictorial representation of this graph.

The definition of graph implies that a graph can be drawn just knowing its vertex-set and its edge-set. For example, our first example



has vertex set V and edge set E where: $V = \{1,2,3,4\}$ and $E = \{(1,2), (2,4), (4,3), (3,1), (1,4), (2,1), (4,2), (3,4), (1,3), (4,1)\}$. Notice that each edge seems to be listed twice.

Another example, the following Petersen Graph $G=(V,E)$ has vertex set, V , and edge set E where: $V = \{1,2,3,4\}$ and $E = \{(1,2), (2,4), (4,3), (3,1), (1,4), (2,1), (4,2), (3,4), (1,3), (4,1)\}$.

3.3 Algorithm Transpose

If graph $G = (V, E)$ is a directed graph, its transpose, $G^T = (V, E^T)$ is the same as graph G with all arrows reversed. We define the transpose of adjacency matrix $A = (a_{ij})$ to be the adjacency matrix $A^T = ({}^T a_{ij})$ given by ${}^T a_{ij} = a_{ji}$. In other words, rows of matrix A become columns of matrix A^T and columns of matrix A become rows of matrix A^T . Since in an undirected graph, (u, v) and (v, u) represented the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^T$.

Formally, the transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$. Thus, G^T is G with all its edges reversed.

We can compute G^T from G in the adjacency matrix representations and adjacency list representations of graph G .

Algorithm for computing G^T from G in representation of graph G is:

3.4 Algorithm Matrix Transpose (G, G^T)

```

For i = 0 to i < V[G]
  For j = 0 to j < V[G]
     $G^T(j, i) = G(i, j)$ 
  j = j + 1
i = i + 1

```

SELF ASSESSMENT EXERCISE

Edges are also referred to as arcs. True or False?

4.0 CONCLUSION

The graph theory and digraph were considered in this unit. You have also learned about algorithmic transpose.

5.0 SUMMARY

What you have learned in this unit concerns graph theory and algorithms.

6.0 TUTOR-MARKED ASSIGNMENT

List two applications of the graph theory.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 2 SORTING

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Sorting
 - 3.2 Internal Sort
 - 3.3 External Sort
 - 3.4 Memory Requirement
 - 3.5 Stability
 - 3.6 Classes of Sorting Algorithms
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignments
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit considers sorting algorithm. It delves into the two kinds of sorting as well as the classes of sorting.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- explain the aim of sorting algorithm
- describe the types of sorting
- explain the classes of sorting algorithm.

3.0 MAIN CONTENT

3.1 Sorting

The objective of the sorting algorithm is to rearrange the records so that their keys are ordered according to some well-defined ordering rule.

Problem: Given an array of n real number $A[1.. n]$.

Objective: Sort the elements of A in ascending order of their values.

3.2 Internal Sort

If the file to be sorted will fit into memory or equivalently, if it will fit into an array, then the sorting method is called internal. In this method, any record can be accessed easily.

3.3 External Sort

Sorting files from tape or disk.

In this method, an external sort algorithm must access records sequentially, or at least in the block.

3.4 Memory Requirement

1. Sort in place and use no extra memory except perhaps for a small stack or table.
2. Algorithms that use a linked-list representation and so use N extra words of memory for list pointers.
3. Algorithms that need enough extra memory space to hold another copy of the array to be sorted.

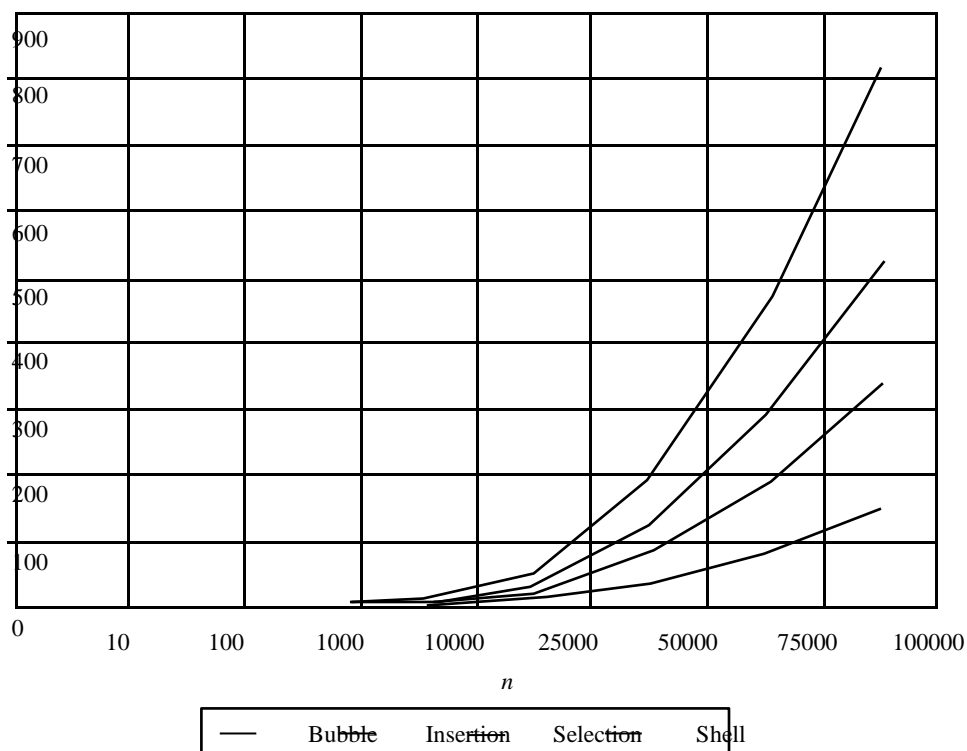
3.5 Stability

A sorting algorithm is called stable if it preserves the relative order of equal keys in the file. Most of the simple algorithms are stable, but most of the well-known sophisticated algorithms are not.

3.6 Classes of Sorting Algorithms

There are two classes of sorting algorithms namely, $O(n^2)$ -algorithms and $O(n \log n)$ -algorithms. $O(n^2)$ -class includes bubble sort, insertion sort, selection sort and shell sort. $O(n \log n)$ -class includes heap sort, merge sort and quick sort.

$O(n^2)$ Sorting Algorithms



$O(n \log n)$ Sorting Algorithms

SELF ASSESSMENT EXERCISE 1

Name two classes of sorting algorithm.

SELF ASSESSMENT EXERCISE 2

Describe the internal sort.

4.0 CONCLUSION

In this unit, you have learned about sorting algorithm. You have also been able to identify classes of sorting algorithm.

5.0 SUMMARY

What you have learned borders on sorting algorithms and their classes.

6.0 TUTOR-MARKED ASSIGNMENT

What do you understand by sorting algorithm?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 3 BUBBLE SORT

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Bubble Sort
 - 3.2 Memory Requirement
 - 3.3 Implementation
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit covers bubble sort, its implementation and memory requirement.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe the bubble sort
- explain the memory requirement
- state the implementation of a bubble sort.

3.0 MAIN CONTENT

3.1 Bubble Sort

Bubble Sort is an elementary sorting algorithm. It works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

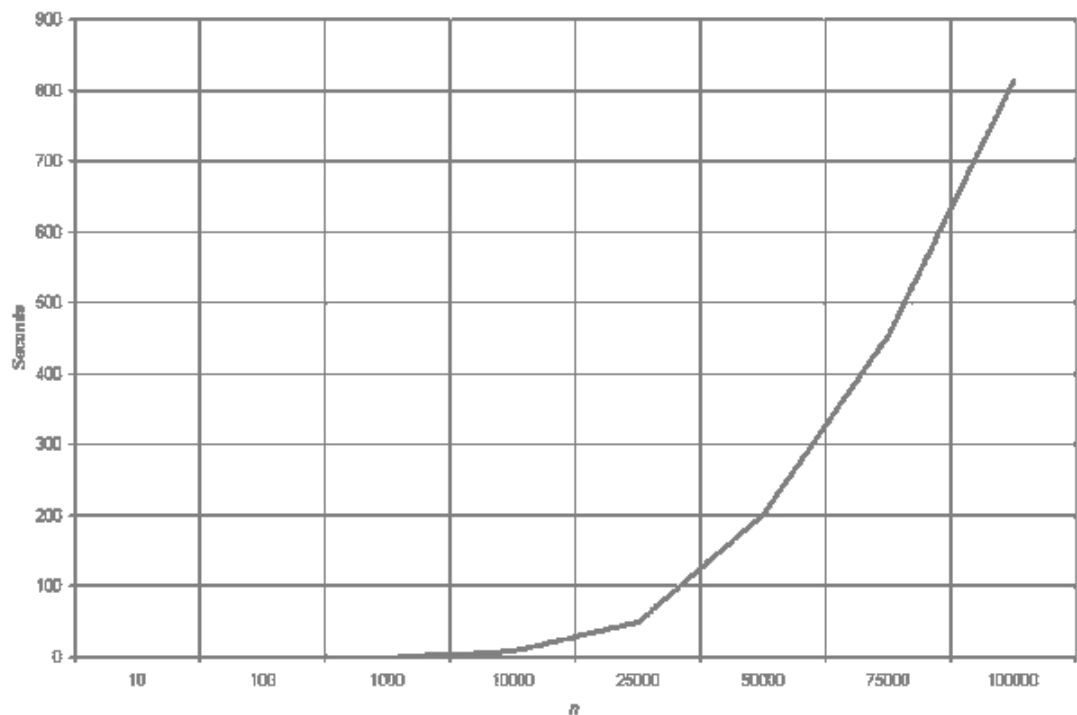


Figure 1.0 n^2 nature of the bubble sort

Clearly, the graph shows the n^2 nature of the bubble sort.

In this algorithm, the number of comparison is irrespective of data set i.e., input whether best or worst.

3.2 Memory Requirement

Clearly, bubble sort does not require extra memory.

3.3 Implementation

```
void bubbleSort(int numbers[], int array_size)
{
    int i, j, temp;
    for (i = (array_size - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (numbers[j-1] > numbers[j])
            {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
```

}

SELF ASSESSMENT EXERCISE

Bubble sort requires extra memory. True or False?

4.0 CONCLUSION

In this unit, you have learned about bubble sort. You have also learned about its memory requirement and implementation.

5.0 SUMMARY

What you have learned in this unit borders on bubble sort.

6.0 TUTOR-MARKED ASSIGNMENT

What do you understand by bubble sort?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 4 INSERTION SORT

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Insertion Sort
 - 3.2 Analysis
 - 3.3 Stability
 - 3.4 Extra Memory
 - 3.5 Implementation
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit considers insertion and its analysis. You will equally learn about the stability and implementation of insertion sort.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe the insertion sort
- analyse an insertion sort
- describe the stability of an insertion sort
- state how insertion sort is implemented.

3.0 MAIN CONTENT

3.1 Insertion Sort

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted).

Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time.

Insertion Sort (A)

1. For $j = 2$ to length $[A]$ do
2. $\text{key} = A[j]$
3. { Put $A[j]$ into the sorted sequence $A[1 \dots j-1]$
4. $i \leftarrow j - 1$
5. while $i > 0$ and $A[i] > \text{key}$ do
6. $A[i+1] = A[i]$
7. $i = i - 1$
8. $A[i+1] = \text{key}$

3.2 Analysis

On examining the statements above, we discover the following cases

Best-Case

The while-loop in line 5 executed only once for each j . This happens if given array A is already sorted.

$$T(n) = an + b = O(n)$$

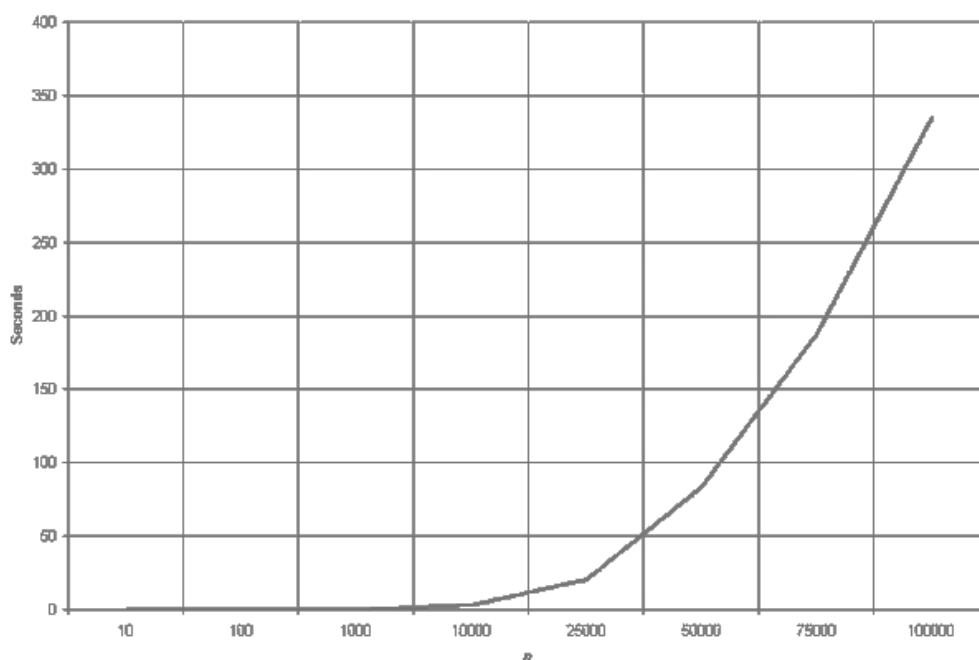
It is a linear function of n .

Worst-Case

The worst-case occurs, when line 5 executed j times for each j . This can happen if array A starts out in reverse order

$$T(n) = an^2 + bc + c = O(n^2)$$

It is a quadratic function of n .



The graph shows the n^2 complexity of the insertion sort.

3.3 Stability

Since multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array, Insertion sort is stable.

3.4 Extra Memory

This algorithm does not require extra memory.

For Insertion sort we say the worst-case running time is $\theta(n^2)$, and the best-case running time is $\theta(n)$.

Insertion sort uses no extra memory it sorts in place.

The time of Insertion sort depends on the original order of an input. It takes a time $\Omega(n^2)$ in the worst-case, despite the fact that a time in order of n is sufficient to solve large instances in which the items are already sorted.

3.5 Implementation

```
void insertionSort(int numbers[], int array_size)
{
int i, j, index;

for (i=1; i < array_size; i++)
{
index = numbers[i];
j = i;
while ((j > 0) && (numbers[j-1] > index))
{
numbers[j] = numbers[j-1];
j = j - 1;
}
numbers[j] = index;
}
}
```

SELF ASSESSMENT EXERCISE

Discuss memory requirement of insertion sort.

4.0 CONCLUSION

In this unit you have learned about insertion sort. You have also learned about the analysis stability and implementation of insertion sort.

5.0 SUMMARY

What you have learned in this unit borders on insertion sort, its analysis and implementation.

6.0 TUTOR-MARKED ASSIGNMENT

Why is an insertion sort said to be stable?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 5 ALGORITHMS

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Selection Sorting
 - 3.2 Straight Selection Sorting
 - 3.3 Implementation of the Selection Sort
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, we will consider selection sort, distinguishing it from insertion sort. The implementation of selection sort is also discussed.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- discuss selection sort
- distinguish selection sort from insertion sort
- describe the straight selection sort
- explain the implementation of selection sort.

3.0 MAIN CONTENT

3.1 Selection Sorting

Selection sorting is a class of sorting algorithm that comprises algorithms that sort *by selection*. Such algorithms construct the sorted sequence one element at a time by adding elements to the sorted sequence *in order*. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This is what makes selection sorting different from insertion sorting. In insertion sorting, elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the

sorted sequence at which each subsequent element is inserted is arbitrary.

The sorts are implemented by exchanging array elements.

Nevertheless, selection differs from exchange sorting because at each step, we *select*

the next element of the sorted sequence from the remaining elements and then we move it into its final position in the array by exchanging it

with whatever happens to be occupying that position.

3.2 Straight Selection Sorting

The simplest of the selection sorts is called *straight selection*. Figure 1.0 illustrates how straight selection works. In the version shown, the sorted list is constructed from the right (i.e., from the largest to the smallest element values).

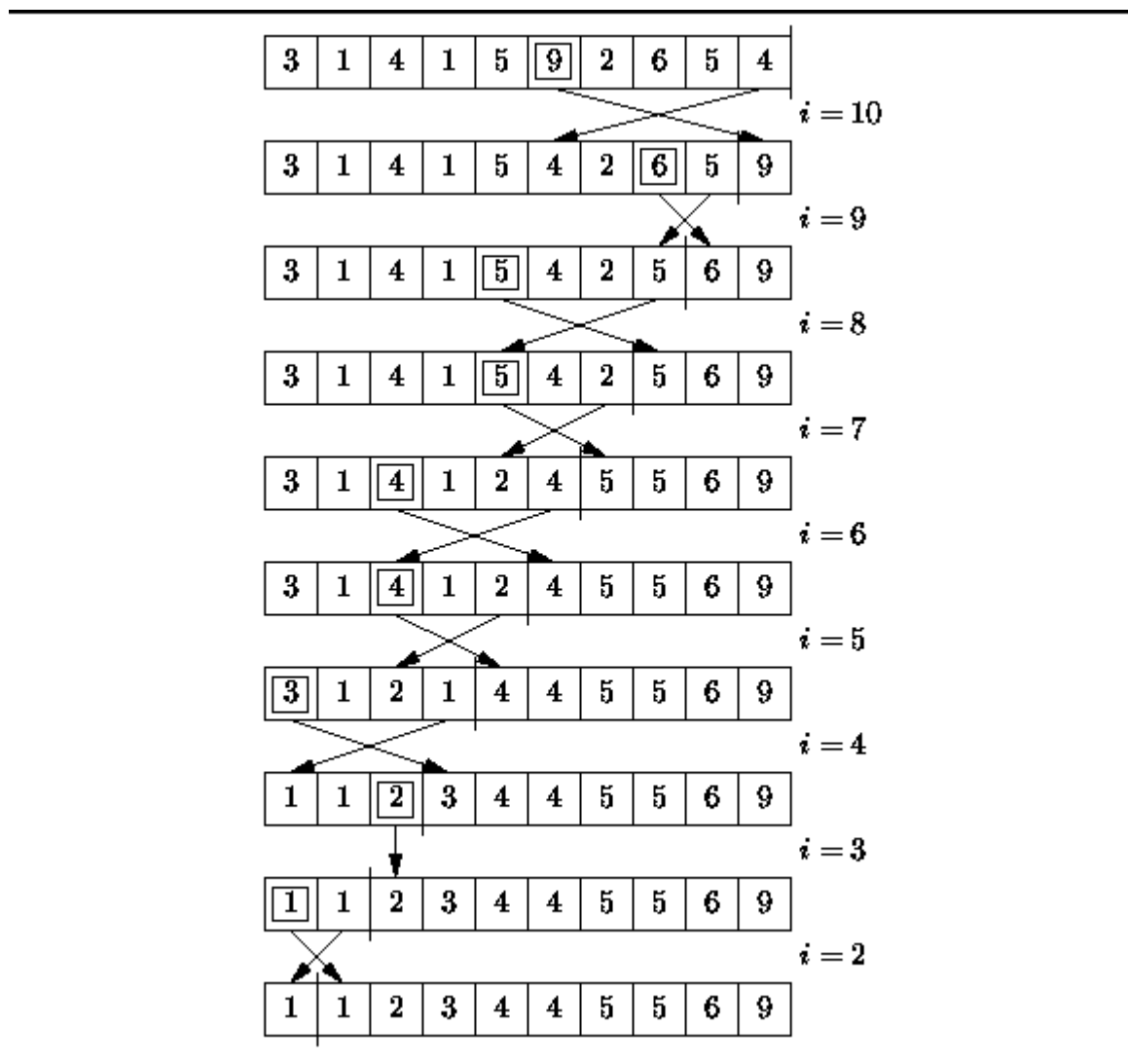


Figure 1.0: Straight selection sorting

At each step of the algorithm, a linear search of the unsorted elements is made in order to determine the position of the largest remaining

element. That element is then moved into the correct position of the array by swapping it with the element which currently occupies that position.

For example, in the first step shown in Figure 1.0, a linear search of the entire array reveals that 9 is the largest element. Since 9 is the largest element, it belongs in the last array position. To move it there, we swap it with the 4 that initially occupied that position. The second step of the algorithm identifies 6 as the largest remaining element and moves it next to the 9. Each subsequent step of the algorithm moves one element into its final position. Therefore, the algorithm is done after $n-1$ such steps.

3.3 Implementation of the Selection Sort

Programme 1.0 defines the `StraightSelectionSorter` class. This class is derived from the `AbstractSorter` base and it provides an implementation for the `no-arg sort` method. The `sort` method follows directly from the algorithm discussed above. In each iteration of the main loop (lines 6-13), exactly one element is selected from the unsorted elements and moved into the correct position. A linear search of the unsorted elements is done in order to determine the position of the largest remaining element (lines 9-11). That element is then moved into the correct position (line 12).

```

1  public class StraightSelectionSorter
2      extends AbstractSorter
3  {
4      protected void sort ()
5      {
6          for (int i = n; i > 1; --i)
7          {
8              int max = 0;
9              for (int j = 1; j < i; ++j)
10                 if (array [j].isGT (array [max]))
11                     max = j;
12                 swap (i - 1, max);
13             }
14         }
15     }

```

Programme 1.0: StraightSelectionSorter

```

class sort
m
e
t
h
o
d

```

In all $n-1$, iterations of the outer loop are needed to sort the array. Notice that exactly one swap is done in each iteration of the outer loop. Therefore, $n-1$ data exchanges are needed to sort the list.

Furthermore, in the i^{th} iteration of the outer loop, $i-1$ iterations of the inner loop are required and each iteration of the inner loop does one data comparison. Therefore, $O(n^2)$ data comparisons are needed to sort the list.

The total running time of the straight selection sort method is $O(n^2)$. Because the same number of comparisons and swaps are always done, this running time bound applies in all cases. That is, the best-case, average-case and worst-case running times are all $O(n^2)$.

SELF ASSESSMENT EXERCISE

What do you understand by straight selection sort.

4.0 CONCLUSION

In this unit you have learned about selection sort and its implementation. You have also learned about straight selection sort.

5.0 SUMMARY

What you have learned in this unit borders on selection sort and its implementation.

6.0 TUTOR-MARKED ASSIGNMENT

Distinguish between insertion sort and selection sort.

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>

UNIT 6 MERGE SORTING

CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
 - 3.1 Merge Sorting
 - 3.2 Implementation
 - 3.3 Merging
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Readings

1.0 INTRODUCTION

This unit will focus primarily on merge sorting. It gives an outline of steps to be adopted in sorting a sequence of elements. We will also consider how to implement a `TwoWayMergeSorter`.

2.0 OBJECTIVES

By the end of this unit, you will be able to:

- describe merge sorting
- outline the steps to be taken in sorting a sequence of $n > 1$ elements
- show how to implement a two-way merge sorting
- define the merge method of a `TwoWayMergeSorter` class.

3.0 MAIN CONTENT

3.1 Merge Sorting

Another class of sorting algorithm which we will consider comprises algorithms that sort *by merging*. Merging is the combination of two or more sorted sequences into a single sorted sequence.

Figure 1.0 illustrates the basic, two-way merge operation. In a two-way merge, two sorted sequences are merged into one. Clearly, two sorted sequences each of length n can be merged into a sorted sequence of length $2n$ in $O(2n)=O(n)$ steps. However, in order to do this, we need space in which to store the result. That is, it is not possible to merge the two sequences *in place* in $O(n)$ steps.

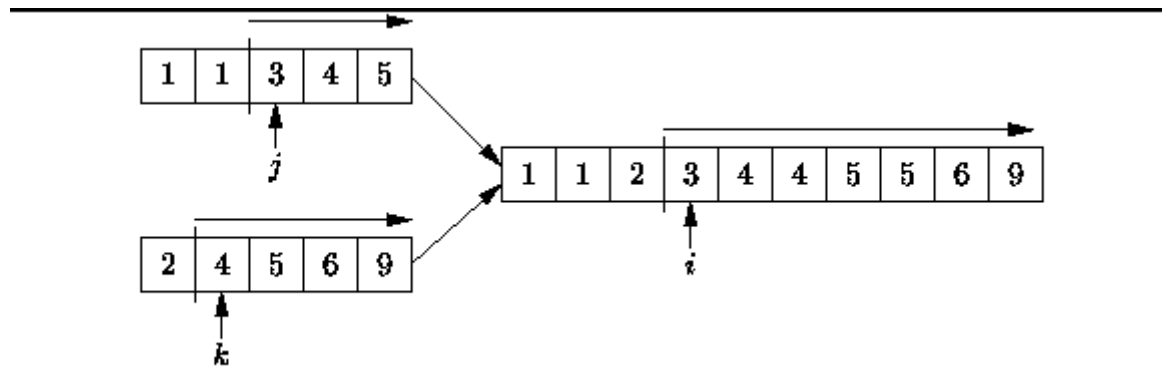


Figure 1.0: Two-way merging

Sorting by merging is a recursive, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of $n > 1$ elements:

1. Divide the sequence into two sequences of length $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$;
2. Recursively sort each of the two subsequences; and then,
3. Merge the sorted subsequences to obtain the final result.

Figure 1.1 illustrates the operation of the two-way merge sort algorithm.

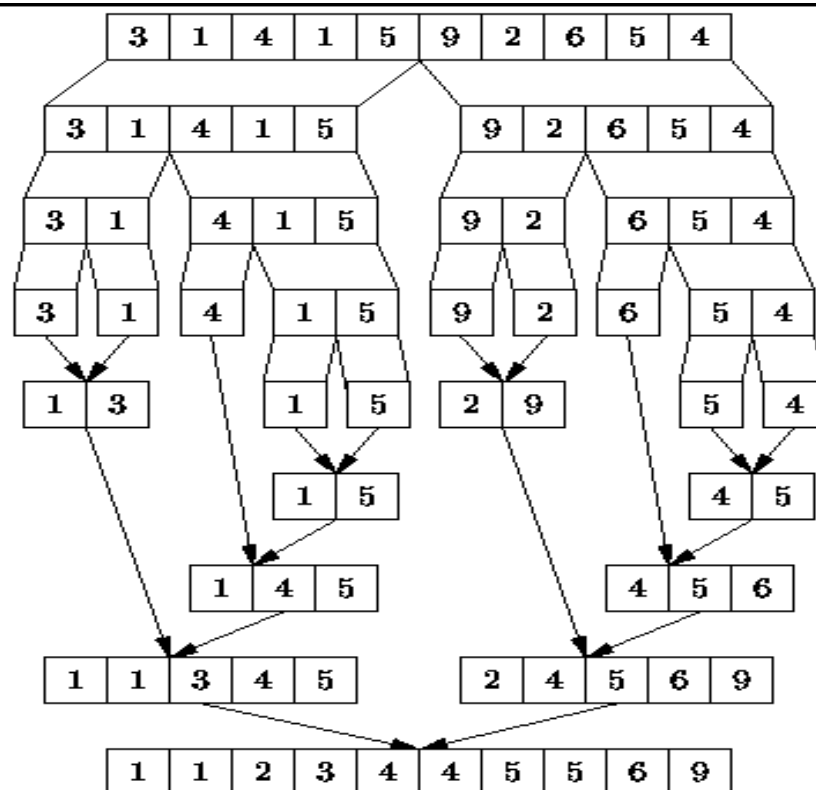


Figure 1.1: Two-way merge sorting

3.2 Implementation

Programme 1.0 declares the `TwoWayMergeSorter` class. The `TwoWayMergeSorter` class extends the `AbstractSorter` class defined in Programme 1.0. A single field, `tempArray`, is declared. This field is an array of `Comparable` objects. Since merge operations cannot be done in place, a second, temporary array is needed. The `tempArray` field keeps track of that array.

```

1 public class TwoWayMergeSorter
2     extends AbstractSorter
3 {
4     Comparable[] tempArray;
5
6     // ...
7 }

```

Programme 1.0: TwoWayMergeSorter fields.

3.3 Merging

The merge method of the `TwoWayMergeSorter` class is defined in Programme 1.1. Altogether, this method takes three integer parameters, `left`, `middle`, and `right`. It is assumed that

$$\text{left} \leq \text{middle} < \text{right}.$$

Furthermore, it is assumed that the two subsequences of the array,
`array[left], array[left + 1], ..., array[middle]`,
 and

`array[middle + 1], array[middle + 2], ..., array[right]`,
 are both sorted. The merge method merges the two sorted subsequences using the temporary array, `tempArray`. It then copies the merged (and sorted) sequence into the array at

$$\text{array[left], array[left + 1], ..., array[right]}.$$

```

1  public class TwoWayMergeSorter
2      extends AbstractSorter
3  {
4      Comparable[] tempArray;
5
6      protected void merge (int left, int middle, int right)
7      {
8          int i = left;
9          int j = left;
10         int k = middle + 1;
11         while (j <= middle && k <= right)
12         {
13             if (array [j].isLT (array [k]))
14                 tempArray [i++] = array [j++];
15             else
16                 tempArray [i++] = array [k++];
17         }
18         while (j <= middle)
19             tempArray [i++] = array [j++];
20         for (i = left; i < k; ++i)
21             array [i] = tempArray [i];
22     }
23     // ...
24 }

```

Programme 1.1: TwoWayMergeSorter class merge method

In order to determine the running time of the merge method, it is necessary to recognise that the total number of iterations of the two loops (lines 11-17, lines 18-19) is $\text{right} - \text{left} + 1$, in the worst case. The total number of iterations of the third loop (lines 20-21) is the same. Since all the loop bodies do a constant amount of work, the total running time for the merge method is $O(n)$, where $n = \text{right} - \text{left} + 1$ is the total number of elements in the two subsequences that are merged.

SELF ASSESSMENT EXERCISE 1

Describe the two-way merge operation.

SELF ASSESSMENT EXERCISE 2

What is the basic assumption in a merge method of the TwoWayMergeSorter class?

4.0 CONCLUSION

Specifically, you learned about merge sorting. You would have also learned about steps to be adopted in sorting a sequence of elements. The implementation of TwoWayMergeSorter was also considered.

5.0 SUMMARY

What you have learned in this unit is focused on merge sorting and its implementation.

6.0 TUTOR-MARKED ASSIGNMENT

What steps are to be adopted to sort a sequence of $n > 1$ elements?

7.0 REFERENCES/FURTHER READINGS

Cormen, T.H., Leiserson, C.E, and Rivest, R.L. (1989). *Introduction to Algorithms*, New York: McGraw-Hill.

French C. S. (1992). *Computer Science*, DP Publications, (4th Edition), 199-217.

Deitel, H.M. and Deitel, P.J. (1998). *C++ How to Programme* (2nd Edition), New Jersey: Prentice Hall.

Ford, W. and Topp, W. (2002). *Data Structures with C++ Using the STL* (2nd Edition), New Jersey: Prentice Hall.

Shaffer, Clifford A. A. (1998). *Practical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, pp. 77–102.

Bruno, R. P. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.

Online Resources

<http://www.gnu.org/manual/emacs-20.3/emacs.html>

<http://www.indiana.edu/~ucspubs/b131>

<http://yoda.cis.temple.edu:8080/UGAIWWW/help>

<http://www.cs.sunysb.edu/~skiena/214/lectures/>