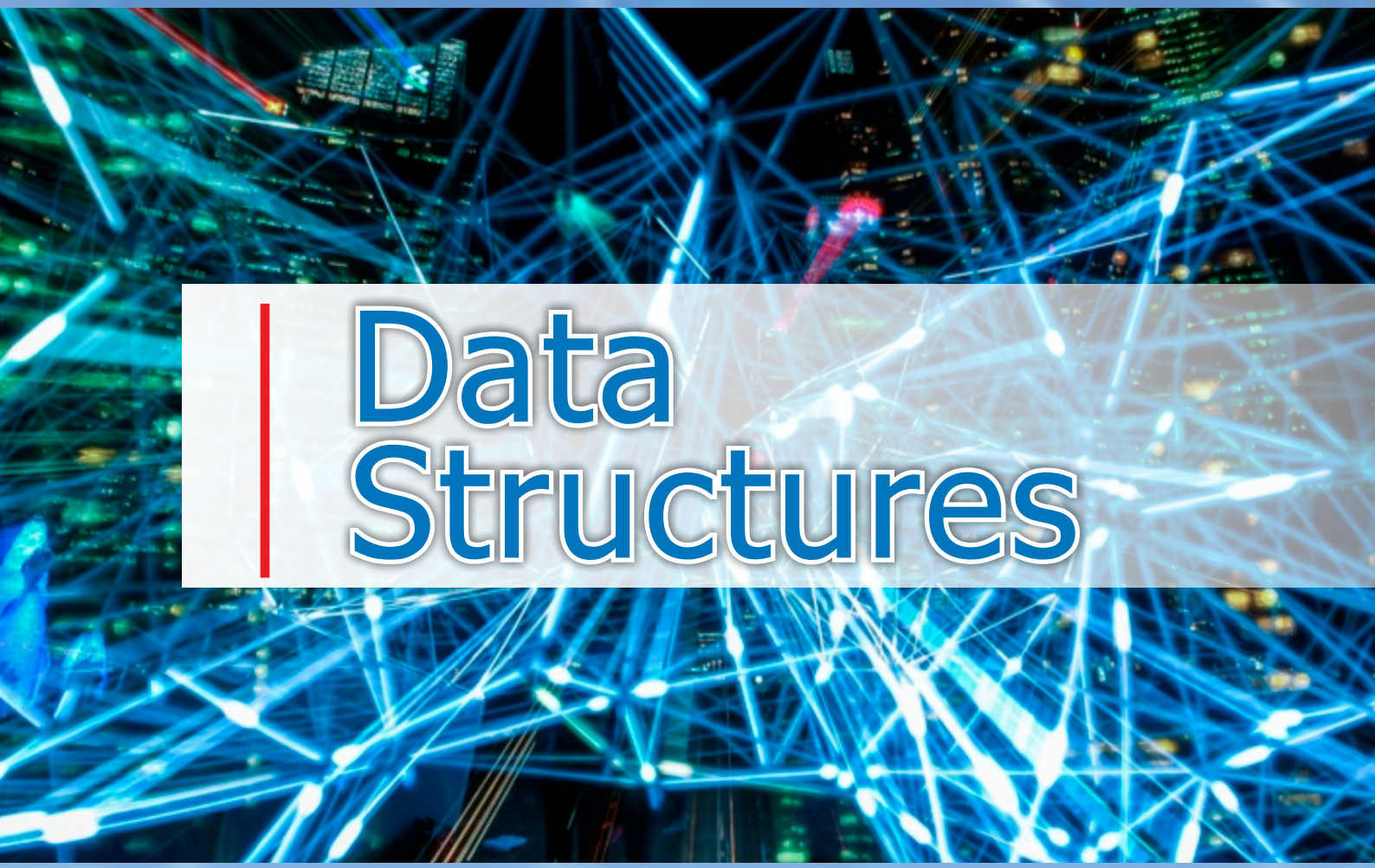
CPT 222



Data Structures

CODeL
FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA
CENTRE FOR OPEN DISTANCE AND e-LEARNING

FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA
NIGER STATE, NIGERIA

CENTRE FOR OPEN DISTANCE AND e-LEARNING (CODeL)

B. TECH. COMPUTER SCIENCE PROGRAMME

COURSE TITLE
DATA STRUCTURES

COURSE CODE
CPT 222

COURSE CODE

CPT 222

COURSE UNIT

3

Course Coordinator

Bashir MOHAMMED (Ph.D.)
Department of Computer Science
Federal University of Technology (FUT) Minna
Minna, Niger State, Nigeria.

Course Development Team

Subject Matter Experts	F.A. OGUNTOLU A. YUSUF Federal University of Technology, Minna, Nigeria.
Course Coordinator	Bashir MOHAMMED (Ph.D.) Department of Computer Science FUT Minna, Nigeria.
Instructional Designers	Oluwole Caleb FALODE (Ph.D.) Bushrah Temitope OJOYE (Mrs.) Centre for Open Distance & e-Learning, Federal University of Technology, Minna, Nigeria
ODL Experts	Amosa Isiaka GAMBARI (Ph.D.) Nicholas Ehikioya ESEZOBOR
Language Editors	Chinenye Priscilla UZOCHUKWU (Mrs.) Mubarak Jamiu ALABEDE
Centre Director	Abiodun Musa AIBINU (Ph.D.) Centre for Open Distance & e-Learning FUT Minna, Nigeria.

CPT 222: Study Guide

Introduction

CPT 222: Data Structures is a 3 Credit unit course for students studying towards acquiring a Bachelor of Technology in Computer Science and other related disciplines. The course is divided into 4 modules and 15 study units. It first takes a brief review of object-oriented design and fundamental data structures. This course will then go ahead to deal with the stacks and queues data structures, hash tables, trees and search trees. The course went further to deal with graphs and sorting. This course also deals with fundamental issues in language design.

The course guide therefore gives you an overview of what the course; CPT 222 is all about, the textbooks and other materials to be referenced, what you expect to know in each unit, and how to work through the course material.

What you will learn in this Course

The overall aim of this course, CPT 222 is to introduce you to basic concepts of Data Structures in order to enable you to understand the basic elements of data structures.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating-point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term “data structure” to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. We study data structures so that we can learn to write more efficient programs.

This course highlights different types of data structures and approaches in the conduct of data organization. These will enable you to write more efficient programs.

Course Aim

This course aims to introduce students to the basic concepts of data structures. To develop your knowledge and understanding of the underlying principles of foundational data structures, develop your competence in analyzing data structures and build up your capacity to write programmes for developing simple applications.

Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objectives of this course. On completing this course, you should be able to:

i. Describe the basic operations on stacks, lists and queue data structures.
ii. Explain the notions of trees, hashing and binary search trees.
iii. Identify the basic concepts of object-oriented programming.
iv. Develop java programmes for simple applications.
v. Discuss the underlying principles of basic data types: lists, stacks and queues.
vi. Identify directed and undirected graphs.
vii. Discuss sorting
viii. Discuss the fundamental issues in language design.

Working through this Course

This course is designed in a systematic way and to complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some Self-Assessment Exercises and tutor marked assignments, and at some point, in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course.

Course Materials

The major components of the course are:

i. Course Guide
ii. Study Units
iii. Text Books
iv. Assignment File
v. Presentation Schedule

Study Units

There are 15 study units and 4 modules in this course. They are:

Module 1: Fundamental Data Structures

Unit 1: Review of Object Oriented Design
Unit 2: Fundamental Data Structures
Unit 3: The Array Structure, File and Records
Unit 4: Linked Structures

Module 2: Stacks and Queues Data Structures

Unit 1: Stacks Data Structures
Unit 2: Queues Data Structures
Unit 3: Hash Tables and Trees
Unit 4: Search Trees and Graphs

Module 3: Sorting

Unit 1 Sorting and Bubble Sort
Unit 2 Insertion Sort
Unit 3 Selection Sort
Unit 4 Merge Sorting

Module 4: Fundamental Issues in Language Design

Unit 1 General principles of language design
Unit 2: Data structures models
Unit 3: Control structure models and abstraction mechanisms

Recommended Texts

These texts and especially the internet resource links will be of enormous benefit to you in learning this course:

1. Addison, W. (1995): Design patterns: elements of reusable object-oriented software. Addison Wesley. 1995. ISBN 0-201-63361-2
2. Clifford, A. S. (2012). Data Structures and Algorithm Analysis. Edition 3.2 (Java Version). http://people.cs.vt.edu/˜shaffer/Book/errata.html
3. Deitel, H. M. and Deitel, P. J. (1998). C++ How to programme (2nd Edition), New Jersey: Prentice Hall.
4. French C. S. (1992). Computer science, DP Publications, (4th Edition), 199-217.
5. Ford, W. and Topp, W. (2002). Data structures with C++ using the STL (2nd Edition), New Jersey: Prentice Hall.
6. Shaffer, Clifford A. A, (1998). Practical introduction to data structures and algorithm analysis, New Jessey: Prentice Hall, pp. 77–102.
7. Martins R. (1999): Designing object oriented applications using UML, 2d. ed., Robert C. Martin, Prentice Hall, 1999.
8. Robert C. M. (2000). Design principles and design patterns. Available at www.objectmentor.com Downloaded on 9/6/2012
9. Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

Online Resources

http://trireme.com/whitepapers/design/objects/object_oriented_design_what.html
http://en.wikipedia.org/wiki/Object-oriented_design
http://en.wikipedia.org/wiki/Data_structure
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
http://www.techterms.com/definition/datatype
http://www.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

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.cofficer.com/programming-language/issues-in-language-design-2/
http://java.sun.com/docs/white/langenv/Intro.doc2.html
http://www.hit.ac.il/staff/leonidm/information-systems/ch62.html
http://www.answers.com/topic/abstract-data-type

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 15 tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavour to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor Marked Assignments (TMAs)

There are TMAs in this course. You need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination for CPT 222 will last for a period of 2 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the Self-Assessment Exercise and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better

to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

The following are practical strategies for working through this course

1. Read the course guide thoroughly
2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.
8. Review the objectives of each study unit to confirm that you have achieved them.
9. If you are not certain about any of the objectives, review the study materiaI and consult your tutor.
10. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.
11. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult you tutor as soon as possible if you have any questions or problems.

12. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated 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 tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

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

1. You do not understand any part of the study units or the assigned readings.
2. You have difficulty with the self-test or exercise.
3. You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavour to attend the tutorials. This is the only opportunity to have face to face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

GOODLUCK!

Table of Content

Module 1

Fundamental Data Structures

Unit 1: Review of Object Oriented Design

Unit 2: Fundamental Data Structures

Unit 3: The Array Structure, File and Records

Unit 4: Linked Structures

Unit 1

Review of Object-Oriented Design

Contents

1.0 Introduction

Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one approach to software design. An object contains encapsulated data and procedures grouped together to represent an entity. The 'object interface', how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects.

Object-oriented design is the discipline of defining the object and their interactions to solve a problem that was identified and documented during object-oriented analysis. What follows is a description of the class-based subset of object-oriented design, which does not include object prototype-based approaches where objects are not typically obtained by instancing classes but by cloning other (prototype) objects.

Therefore, the unit introduces you to the fundamental notions of object-oriented design, thus guiding you through and facilitating your understanding of data structures in the subsequent units.

2.0 Learning Outcomes

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

i. Describe the input (sources) of object-oriented design;

ii. Explain object-oriented concepts;

iii. Describe designing concepts;

iv. Understand Output (deliverables) of object-oriented design;

v. Explain Some design principles and strategies

3.0 Learning Content

3.1 Input (Sources) for Object-Oriented Design

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process.

Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot. Some typical input artifacts for object-oriented design are:

Conceptual model

Conceptual model is the result of object-oriented analysis; it captures concepts in the problem. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

Use case

Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

System Sequence diagram (SSD)

System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

User interface documentations

User interface documentations (if applicable) is a document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

Relational data model

Relational data model (if applicable): A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel and the growth of an artifact can stimulate the refinement of other artifacts.

Self-Assessment Exercise

1. Differentiate between conceptual model and user interface documentations?

Self-Assessment Answer

3.2 Object Oriented Concepts

The five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names: object/class; information hiding; inheritance; interface and polymorphism.

Object/Class

Object/Class: A tight coupling or association of data structures with the methods or functions that act on the data. This is called a *class*, or *object* (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what it is and what it can do. An object can be part of a class, which is a set of objects that are similar.

For example:

Object variables and methods are accessed using dot notation. Use instance_name.variable or instance_name.method_name(args) to reference instance objects declared with new. Use class_name.variable or class_name.method_name(args) to reference static variables or methods.

```
Employee e=new Employee();
e.name="Al Bundy"; e.setSalary(1000.00); // refs instance
Employee.getCount( ); // references static class variable
```

Information hiding

Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*.

Inheritance

Inheritance: The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data.

Example:

A classic recursion is factorials where n factorial is the product of positive integer n and all the products before it down to one. In Java this could be programmed as:

```
Class Factorial
{
      Int factorial ( int n)
      {
      If ( n= =1 ) { return 1};
            Return ( n * factorial (n-1 ) ) ;
      }
}
```

Note: This short method is not very well written as negative and floating calling parameters are illegal in factorials and will cause problems in terminating the loop. Bad input should always be trapped.

Interface

Interface: The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them.

Polymorphism

Polymorphism: The ability to replace an *object* with its *subobjects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *subobjects*.

Self-Assessment Exercise

1. Explain the terms information hiding and interface in object-oriented concepts?

Self-Assessment Answer

3.3 Designing Concepts

Design concepts involve the following steps;

i. Defining objects, creating **class diagram** from **conceptual diagram**: Usually map entity to class.

ii. Identifying **attributes.**

iii. Use **design patterns** (if applicable): A design pattern is not a finished design; it is a description of a solution to a common problem, in a context. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between

classes or objects, without specifying the final application classes or objects that are involved.

iv. (Define **application framework** (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.

v. Identify persistent objects/data (if applicable): Identify objects that have to last longer than a single runtime of the application. If a relational database is used, design the object relation mapping.

vi. Identify and define remote objects (if applicable).

Self-Assessment Exercise

1. Why do you think a design pattern should be used if applicable in design concepts?

Self-Assessment Answer

3.4 Output (Deliverables) of Object-Oriented Design

The two outputs (deliverables) of object-oriented design are as follows;

i. **Sequence Diagram**: Extend the **System Sequence Diagram** to add specific objects that handle the system events.

ii. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

iii. **Class diagram:** A class diagram is a type of static structure **UML** diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

3.5 Some Design Principles and Strategies

i. **Dependency injection**: The basic idea is that if an object depends upon having an instance of some other object then the needed object is "injected" into the dependent object; for example, being passed a database connection as an argument to the constructor instead of creating one internally.

ii. **Acyclic dependencies principle:** The dependency graph of packages or components should have no cycles. This is also referred to as having a directed acyclic graph. For example, package C depends on package B, which depends on package A. If package A also depended on package C, then you would have a cycle.

iii. **Composite reuse principle**: Favor polymorphic composition of objects over inheritance.

Self-Assessment Exercise

1. Explain class diagram as an output of object-oriented design?

Self-Assessment Answer

4.0 Conclusion

In this unit, you have learned about the object-oriented designs topics such as the input (sources) for object-oriented design; object-oriented concepts; design concepts; output (deliverables) of object oriented-design; and some design principles and strategies. You have also been able to understand the meaning of all these objected-oriented paradigms. This object-oriented review will help you in understanding data structures better. This unit serves as a basis for the next unit.

5.0 Summary

You have learnt that:

i. The input for object-oriented design is provided by the output of object-oriented analysis.

ii. The five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names: object/class; information hiding; inheritance; interface and polymorphism.

iii. Design concepts involves defining objects, creating class diagram from conceptual diagram, identifying attributes, use design patterns (if applicable) Define application framework (if applicable), identify persistent objects/data (if applicable) and identify and define remote objects (if applicable).

iv. The two output (deliverables) of object-oriented design are sequence diagram and class diagram

v. Some design principles and strategies are dependency injection, acyclic dependencies principle and composite reuse principle

6.0 Tutor-Marked Assignment

A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine. What are some of the objects that can be created for this design?

7.0 References/Further Reading

Addison, W. (1995): Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. 1995. ISBN 0-201-63361-2

Martins R. (1999): Designing Object Oriented Applications Using *UML*, 2d. Ed., Robert C. Martin, Prentice Hall, 1999.

Robert C. M. (2000). Design Principles and Design Patterns. Available at www.objectmentor.com Downloaded on 9/6/2012

Online Resources

http://trireme.com/whitepapers/design/objects/object_oriented_design_what.html

http://en.wikipedia.org/wiki/Object-oriented_design

Unit 2

Fundamental Data Structures

Contents

1.0 Introduction

The modern digital computer was invented and designed as a device that ought to facilitate and speed up complicated and time-consuming computations. Therefore, in this unit some basic concepts that the student needs to be familiar with before attempting to develop any software are introduced. It gives descriptions data type and data structures, with explanation on the operations that may be performed on them. It also explains the fundamental notions of data structures which will serve as guide and basis for your understanding of the subsequent units.

2.0 Learning Outcomes

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

(i) Explain and use the primitive data types and standard data types

(ii) Delineate the classification of data type

(iii) Present standard examples of data type

(iv) Explain the importance of data structures in programming

3.0 Learning Content

3.1 Definition of Data Type

A **data type** in computer programming simply refers to a classification of various kinds of data that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.

A *data type* consists of:

(i) a domain (= a set of values)

(ii) a set of operations that may be applied to the values.

These fundamental operators are defined for most data types, but it should be noted that their execution may involve a substantial amount of computational effort, if the data are large and highly structured.

For the standard primitive data types, we postulate not only the availability of assignment and comparison, but also a set of operators to create (compute) new values. Thus we introduce the standard operations of arithmetic for numeric types and the elementary operators of propositional logic for logical values.

Self-Assessment Exercise

1. What do you understand by a data type?

Self-Assessment Answer

3.2 Primitive Data Types

A new, primitive type is definable by enumerating the distinct values belonging to it. Such a type is called an *enumeration type*. Its definition has the form

```
TYPE  T = (c1, c2, ... , cn)
```

T is the new type identifier, and the c_i are the new constant identifiers.

Examples

```
TYPE shape = (rectangle, square, ellipse, circle) TYPE color
= (red, yellow, green)

TYPE sex = (male, female)

TYPE weekday = (Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday)

TYPE currency = (franc, mark, pound, dollar, shilling, lira,
guilder, krone, ruble, cruzeiro,

               yen)

TYPE destination = (hell, purgatory, heaven)

TYPE vehicle = (train, bus, automobile, boat, airplane)

TYPE rank = (private, corporal, sergeant, lieutenant, captain,
major, colonel, general)

TYPE object = (constant, type, variable, procedure, module)
TYPE structure = (array, record, set, sequence)

TYPE condition = (manual, unloaded, parity, skew)
```

Self-Assessment Exercise

1. With an example, explain primitive data type?

Self-Assessment Answer

3.3 Standard Primitive Types

Standard primitive includes the whole numbers, the logical truth values, and a set of printable characters. We denote these types by the identifiers

INTEGER, REAL, BOOLEAN, CHAR, SET

The type INTEGER

The type INTEGER comprises a subset of the whole numbers whose size may vary among individual computer systems.

Example: 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.

If a computer uses *n* bits to represent an integer in two's complement notation, then the admissible values x must satisfy $-2^{n-1} \leq x < 2^{n-1}$. It is assumed that all operations on data of this type are exact and correspond to the ordinary laws of arithmetic, and that the computation will be interrupted in the case of a result lying outside the representable subset. This event is called *overflow*. The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/, DIV).

The type REAL

The type REAL denotes a subset of the real numbers. Whereas arithmetic with operands of the types INTEGER is assumed to yield exact results, arithmetic on values of type REAL is permitted to be inaccurate within the limits of round-off errors caused by computation on a finite number of digits. This is the principal reason for the explicit distinction between the types INTEGER and REAL, as it is made in most programming languages.

Example: 1.2; -2.3; 3.5

The type BOOLEAN

The two values of the standard type BOOLEAN are denoted by the identifiers TRUE and FALSE. The Boolean operators are the logical conjunction, disjunction, and negation.

Examples 1: Examples of Boolean operators are OR, NOT, AND, etc

Example 2: In Boolean expression. For instance, given Boolean variables p and q and integer variables x = 5, y = 8, z =10, the two assignments

```
p := x = y
```

```
q := (x ≤ y) & (y < z)
yield p = FALSE and q = TRUE.
```

The type CHAR

The standard type CHAR comprises a set of printable characters. Unfortunately, there is no generally accepted standard character set used on all computer systems. Therefore, the use of the predicate "standard" may in this case be almost misleading; it is to be understood in the sense of "standard on the computer system on which a certain program is to be executed."

The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set. The ASCII set is therefore tabulated in Appendix A. It consists of 95 printable (graphic) characters and 33 control characters, the latter mainly being used in data transmission and for the control of printing equipment.

Self-Assessment Exercise

1. Explain the type Boolean with examples?

Self-Assessment Answer (s) 3

3.4 Abstract Data Type (ADT)

An Abstract Data Type commonly known 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.

Examples of Abstract Data Type

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

Self-Assessment Exercise

1. What is an abstract data type?

Self-Assessment Answer

3.5 Definition of 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 collection”. A cautiously chosen data structure will permit the most efficient algorithm to be used. Thus, a well-designed data structure allows a range of critical operations to be performed using a few resources, both execution time and memory spaces as possible.

Self-Assessment Exercise

1. Explain a data structure?

Self-Assessment Answer

3.6 Classification of Data Structures

Data structures are largely divided into two:

(i) Linear Data Structures

(ii) Non-Linear Data Structures.

Linear Data Structures

The data structures in which individual data elements are stored and accessed linearly in the computer memory are called linear data structures. In 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.

Non-Linear Data Structures

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 is called a non-linear data structure. Typical non-linear data structures to be considered in this course are Trees.

Self-Assessment Exercise

1. Describe linear data structures?

Self-Assessment Answer

3.7 Data Structures and Programmes

In software programmes, the structure of data in the computer is very important, especially where the set of data is very large. A well-structured data that are stored in the computer, makes the accessibility of data easier and the software programme routines that make do with the data are made simpler; time and storage spaces are also reduced. The choice of data structures is a primary design consideration in the design of many types of programmes, as experience in building huge systems has revealed that the complexity of implementation and the quality and performance of the final result depends greatly on choosing the best data structure.

Self-Assessment Exercise

1. Explain why the structure of data in the computer is very important?

Self-Assessment Answer

4.0 Conclusion

In this unit, you have learned about the data types, which are primitives and standard types. You have also been able to understand the meaning of some notions such as; data type, integer, real, Boolean data types and standard type CHAR. You have also learned about abstract data type (ADT) and data structures. Finally, you have been able to know and appreciate the relevance of data structures in developing high quality computer programme.

5.0 Summary

You have learnt that:

(i) A data type in computer programming simply refers to a classification of various kinds of data that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.

(ii) A new, primitive type is definable by enumerating the distinct values belonging to it. Such a type is called an *enumeration type*. Its definition has the form

```
TYPE T = (c1, c2, ..., cn)
```

(iii) Standard primitive includes the whole numbers, the logical truth values, and a set of printable characters. We denote these types by the identifiers INTEGER, REAL, BOOLEAN, CHAR, SET

(iv) An Abstract Data Type commonly known as ADT, is a collection of data objects characterized by how the objects are accessed.

(v) A data structure is the implementation of an abstract data type in a particular programming language.

(vi) Data structures are largely divided into two: Linear data structures and non-linear data structures.

(vii) In software programmes, the structure of data in the computer is very important, especially where the set of data is very large.

6.0 Tutor-Marked Assignment

(1) Given a set of odd numbers between 1 and 10, explain how you will divide each of this number by 2 and get INTEGER as results.

(2) Given Boolean variables p and q and integer variables x = 6, y = 7, z =12, the two assignments

```
p := x = y
q := (x ≤ y) & (y < z)
will yields what?
```

(3) Discuss in detail the term “Abstract Data Type”?

7.0 References/Further Reading

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

Dan A., Cogito, E (2000). Cognitive processes of students dealing with data structures. In proceedings of SIGCSE'00, pages 26–30, ACM Press, March 2000.

Ken, A. and James, G. (2006). The java programming language. Addison-Wesley, Reading, MA, USA, fourth edition.

NOUN (2009). Course Guide on CIT 341: Data structures. National Open University of Nigeria, Headquarters, Victoria Island, Lagos.

Shaffer, Clifford A. (1998). *A practical introduction to data structures and algorithm analysis.* Prentice Hall, pp. 77–102.

Online Resources

http://en.wikipedia.org/wiki/Data_structure

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

http://www.techterms.com/definition/datatype

http://www.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 3

The Array Structure, File and Records

Contents

1.0 Introduction

In this unit, you will learn about arrays, their declaration, dimensions and applications. You will also learn how to distinguish between the different types of arrays and learn about the applications of array to computer programming.

2.0 Learning Outcomes

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

(i) illustrate an array, its dimensionality and declaration
(ii) express a two-dimensional array linearly
(iii) distinguish between static and dynamic arrays
(iv) explain the importance of arrays in computer applications.
(v) Explain lists, files and records

3.0 Learning Content

3.1 Arrays

An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. Arrays are most frequently used in programming. Mathematical problems like matrix, algebra and etc. can be easily handled by arrays. If an element of an array is referenced by single subscript, then the array is known as one Dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two-dimensional array and so on. Analogously the arrays whose elements are referenced by two or more subscripts are called multi-dimensional arrays.

Declaration of Arrays

In computer programming, 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 instance, 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;
int age4;
int age5;
int age6;
```

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.

One Dimensional Array

One-dimensional array (or linear array) is a set of 'n' finite numbers of homogenous data elements such as:

(i) The elements of the array are referenced respectively by an index set consisting of 'n' consecutive numbers.

(ii) The elements of the array are stored respectively in successive memory locations. 'n' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as

```
A[0], A[1], A[2], ..... A[n –1].
```

The number 'n' in A[n] is called a subscript or an index and A[n] is called a subscripted variable. If 'n' is 10, then the array elements A[0], A[1].....A[9] are stored in sequential memory locations as follows :

A[0]	A[1]	A[2]	. .. . .	A[9]

In C, array can always be read or written through loop. To read a one-dimensional array, it requires one loop for reading and writing the array, for example: For reading an array of 'n' elements

```
for (i = 0; i < n; i ++)
     scanf ("%d",&a[i]);
```

For writing an array

```
for (i = 0; i < n; i ++)
     printf ("%d", a[i]);
```

As in most programmeming languages, like in Java, 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]
```

Multi-Dimensional Array

If we are reading or writing two-dimensional array, two loops are required. Similarly, the array of ‘n’ dimensions would require ‘n’ loops. The structure of the two-dimensional array is illustrated in the following figure:

```
int A [10] [10];
```

A_{00}	A_{01}	A_{02}	A_{03}					A_{08}	A_{09}
A_{10}									A_{19}
A_{20}									A_{29}
A_{30}									
A_{80}									A_{89}
A_{90}	A_{91}							A_{98}	A_{99}

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.

Applications of Arrays

Arrays are engaged in many computer applications in which data items need to be saved in the computer memory for later reprocessing.

Due to their performance uniqueness, arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks and strings.

Self-Assessment Exercise(s)

(i) An array is …….. data elements described by a single name.

(ii) In computer programming, ……. only store a single value

Self-Assessment Answer (s)

3.2 Lists

As we have discussed, an array is an ordered set, which consist of a fixed number of elements. No deletion or insertion operations are performed on arrays. Another main disadvantage is its fixed length; we cannot add elements to the array. Lists overcome all the above limitations. A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list. Any other list is said to be nonlinear. List can be implemented by using pointers. Each element is referred to as nodes; therefore, a list can be defined as a collection of nodes as shown below:

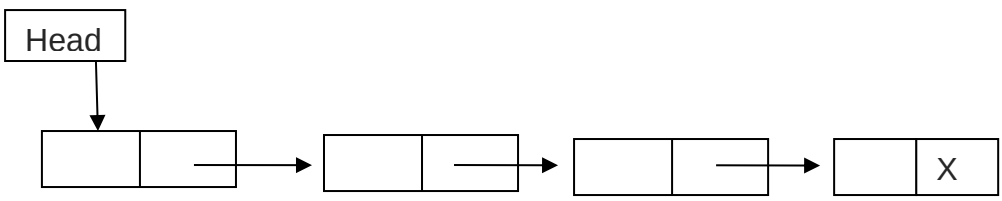


Figure3.1: List representation

Self-Assessment Exercise

1. What is a list?

Self-Assessment Answer

3.3. Files and Records

A file is typically a large list that is stored in the external memory (e.g., a magnetic disk) of a computer.

A record is a collection of information (or data items) about a particular entity. More

Specifically, a record is a collection of related data items, each of which is called a filed or attribute and a file is a collection of similar records.

Although a record is a collection of data items, it differs from a linear array in the

Following ways:

(a) A record may be a collection of non-homogeneous data; i.e., the data items in a record may have different data types.

(b) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

Self-Assessment Exercise

1. What is a file?

Self-Assessment Answer

3.4 Characteristics of Strings

In computer terminology the term ‘string’ refers to a sequence of characters. A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string.

String Representation

Strings are stored or represented in memory by using following three types of structures:

• Fixed length structures

• Variable length structures with fixed maximum

• Linear structures

Fixed Length Representation: In fixed length storage each line is viewed as a record, where all records have the same length. That is each record accommodates maximum of same number of characters.

The main advantage of representing the string in the above way is:

1. To access data from any given record easily.

2. It is easy to update the data in any given record.

The main disadvantages are:

1. Entire record will be read even if most of the storage consists of inessential blank space. Time is wasted in reading these blank spaces.
2. The length of certain records will be more than the fixed length. That is certain records may require more memory space than available.

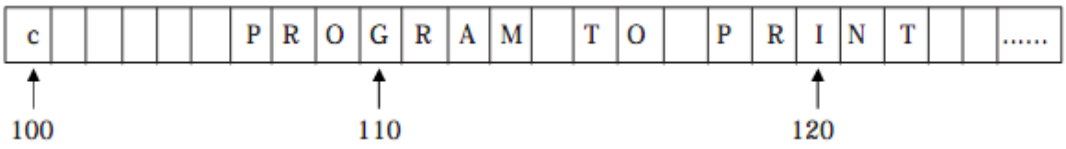


Figure 3.2: Input

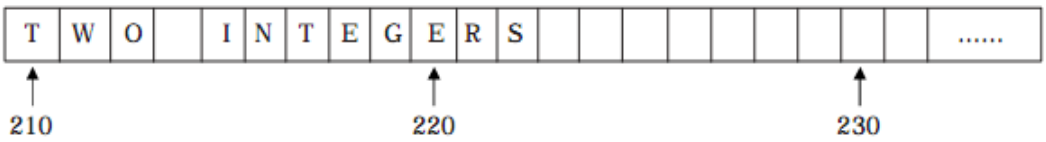


Figure 3.3: Fixed length representation

Fig. 3.3 is a representation of input data (which is in Fig. 3.2) in a fixed length (records) storage media in a computer.

Variable Length Representation: In variable length representation, strings are stored in a fixed length storage medium. This is done in two ways.

1. One can use a marker, (any special characters) such as two-dollar sign ($$), to signal the end of the string.

2. Listing the length of the string at the first place is another way of representing strings in this method.

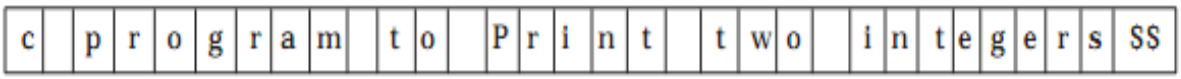


Figure 3.4: String representation using marker

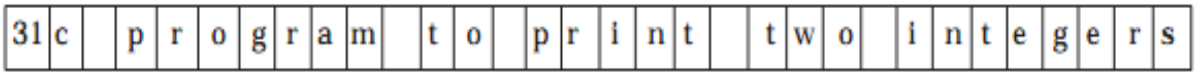


Figure 3.5: String representation by listing the length

Linked List Representations: In linked list representations each character in a string are sequentially arranged in memory cells, called nodes, where each node contain an item and link, which points to the next node in the list (i.e., link contain the address of the next node).

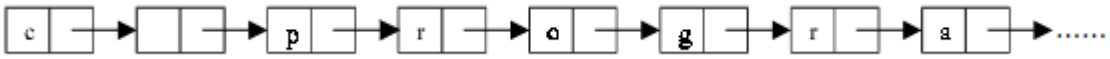


Figure 3.6: One character per node

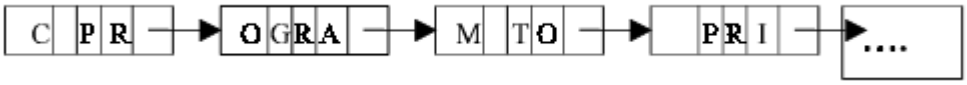


Figure 3.7: Four character per node

We will discuss the implementation issues of linked list in the subsequent units.

Sub String

Group of consecutive elements or characters in a string (or sentence) is called substring. This group of consecutive elements may not have any special meaning. To access a sub string from the given string we need following information:

(a) Name of the string

(b) Position of the first character of the sub string in the given string

(c) The length of the sub string

Finding whether the sub string is available in a string by matching its characters is called pattern matching.

Self-Assessment Exercise

1. How is string represented in the memory?

Self-Assessment Answer

4.0 Conclusion

In this unit, you have learned about the arrays and their dimensionality. You have also been able to recognize the meaning of some concept such as; array name, element and array declaration. You have been able to differentiate between the static and dynamic arrays as well as recognize the applications of arrays. Also, you have learned about Lists, files and records. You have also been able to identify the elements of a List. You should also have learned about strings and different ways of string representation.

5.0 Summary

You have learnt that:

(i) An array is a collection of homogeneous data elements described by a single name.

(ii) A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list.

(iii) A file is typically a large list that is stored in the external memory (e.g., a magnetic disk) of a computer. A record is a collection of information (or data items) about a particular entity.

(iv) A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string.

6.0 Tutor-Marked Assignment

Describe a suitable data structure for details of storing the name of students who are 30 in a level.

7.0 References/Further Readings

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.

Vinus V. D. (2008). Principles of Data Structures using C and C++. New Age International (P) Limited, New Delhi, India.

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 4

Linked Structures

Contents

1.0 Introduction

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

2.0 Learning Outcomes

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

(i) Describe a Linked List

(ii) Identify the elements of a linked list

(iii) Explain the operations and implementations of linked lists.

(iv) Explain the different types of linked list

3.0 Learning Content

3.1 What is a Linked List?

A linked list is a linear collection of specially designed data elements, called nodes, linked tone another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field. Figure 1.1 shows a typical example of node.

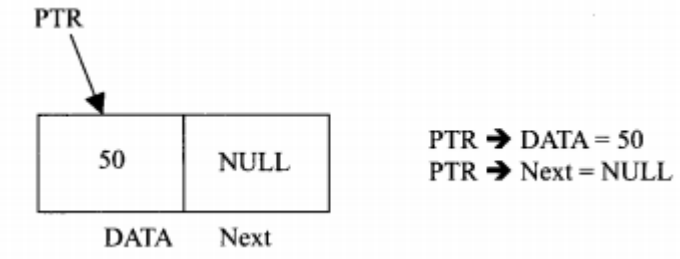


Figure 1.1 Node

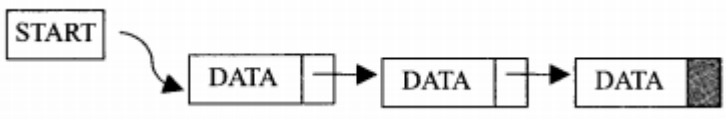


Figure 1.2 Linked

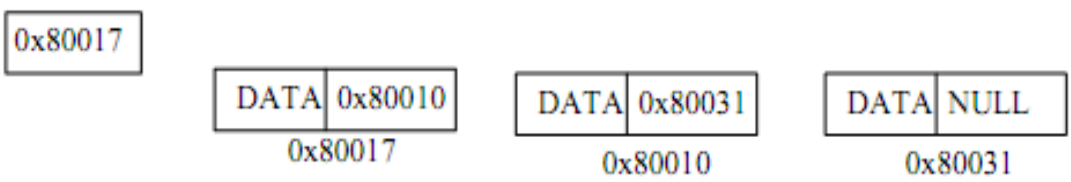


Figure 1.3 Linked list representations in the memory

Fig. 1.2 shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START =NULL if there is no list (i.e.; NULL list or empty list).

Self-Assessment Exercise

1. What is a linked list?

Self-Assessment Answer

3.2 Representation of Linked List

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

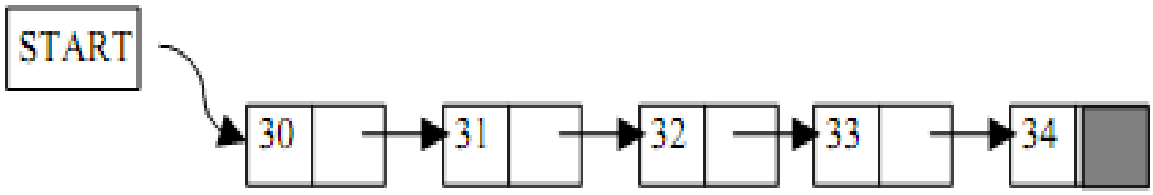


Figure 1.4 Linked list representations of integers

The linear linked list can be represented in memory with the following declaration.

```
struct Node
{
int DATA; //Instead of 'DATA' we also use 'Info'
struct Node *Next; //Instead of 'Next' we also use
'Link'
};
typedef struct Node *NODE;
```

3.3 Advantages and Disadvantages

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

Self-Assessment Exercise

1. Why are linked list referred to as dynamic data structure?

Self-Assessment Answer

3.4 Operation on Linked List

The primitive operations performed on the linked list are as follows

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Concatenation

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(a) At the beginning of the linked list

(b) At the end of the linked list

(c) At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

(a) Beginning of a linked list

(b) End of a linked list

(c) Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Concatenation is the process of appending the second list to the end of the first list.

Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes.

Self-Assessment Exercise

1. What is creation operation on linked list?

Self-Assessment Answer

3.5 Types of Linked List

Basically, we can divide the linked list into the following three types in the order in which they (or node) are arranged.

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly Linked List

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure.

The following figures explain the different operations on a singly linked list.

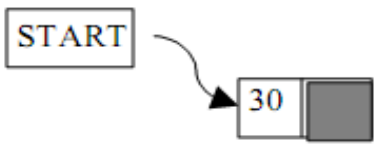


Figure 1.5 Create a node with DATA(30)

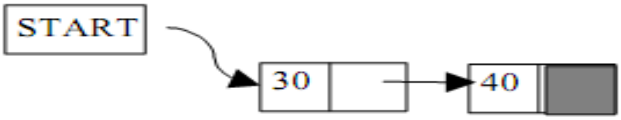


Figure 1.6 Insert a node with DATA(40) at the end

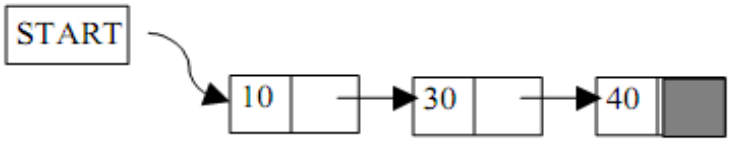


Figure 1.7 Insert a node with DATA(10) at the beginning

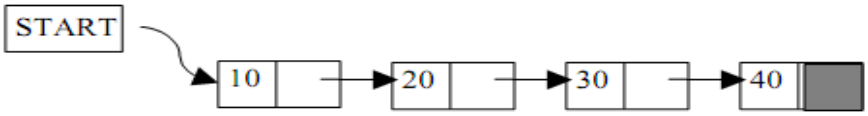


Figure 1.8 Insert a node with DATA(20) at the second position

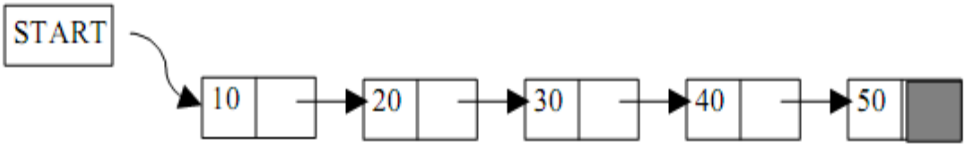


Figure 1.9 Insert a node with DATA(50) at the

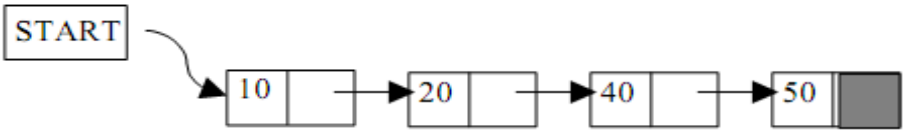


Figure 1.10 Delete the node from the list

Doubly Linked Lists

These allow scanning or searching of the list in both directions. In this case, the node structure is altered to have two links:

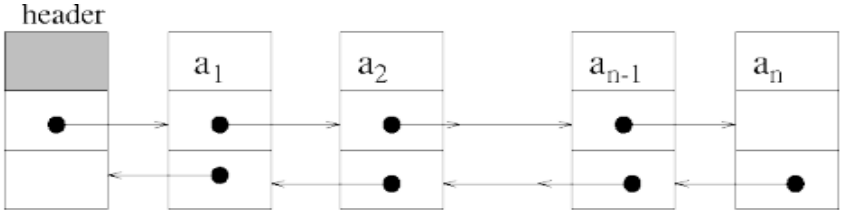


Figure 1.11 Doubly linked list

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.

Circularly Linked Lists

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

```
/** Singly linked list node */
class Link<E> {
  private E element;          // Value for this node
  private Link<E> next;       // Pointer to next node in list

  // Constructors
  Link(E it, Link<E> nextval)
    { element = it;  next = nextval; }
  Link(Link<E> nextval) { next = nextval; }

  Link<E> next() { return next; }  // Return next field
  Link<E> setNext(Link<E> nextval) // Set next field
    { return next = nextval; }     // Return element field
  E element() { return element; }  // Set element field
  E setElement(E it) { return element = it; }
}
```

Program 1.2: A simple singly linked node implementation in Java

Program 1.2 shows the implementation for list nodes, called the **Link** class. Objects in the **Link** class contain an **element** field to store the element value, and a **next** field to store a pointer to the next node on the list. The list built from such nodes is called a singly linked list, or a one-way list, because each list node has a single pointer to the next node on the list. The **Link** class is quite simple. There are two forms for its constructor, one with an initial element value and one without. Member functions allow the link user to get or set the **element** and **link** fields.

Self-Assessment Exercise

1. List the three types of linked list?

Self-Assessment Answer

4.0 Conclusion

In this unit you have learned about linked lists. You have also been able to identify the elements of a linked list as well as the advantages and disadvantages of linked lists. You should also have learned about different operations that can be performed on linked list and implementations of linked lists.

5.0 Summary

You have learnt that:

(i) A linked list is a linear collection of specially designed data elements, called nodes, linked tone another by means of pointers.
(ii) Linked list has many advantages such as linked list is dynamic data structure.
(iii) The primitive operations performed on the linked list are as follows creation, insertion, deletion, traversing, searching, and concatenation.
(iv) Basically, we can divide the linked list into the following three types in the order in which they (or node) are arranged. Singly linked list; Doubly linked list and Circular linked list.

6.0 Tutor-Marked Assignment

With the aid of diagrams explain how you can create a node with a data and then insert a node with another at the end of it.

7.0 References/Further Readings

Clifford, A. S. (2012). Data structures and algorithm analysis. Edition 3.2 (Java Version). http://people.cs.vt.edu/˜shaffer/Book/errata.html

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

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

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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India.

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Module 2

Stacks and Queues Data Structures

Unit 1: Stacks Data Structures

Unit 2: Queues Data Structures

Unit 3: Hash Tables and Trees

Unit 4: Search Trees and Graphs

Unit 1

Stacks Data Structures

Contents

1.0 Introduction

In this unit, we will consider an abstract data structure – the Stack Data Structure. This structure stores and accesses data in diverse ways, which are useful in diverse applications. In all cases, the stack data arrangement 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 Learning Outcomes

By the end of this unit, the student should be able to:

(i) Describe the stack data structure
(ii) Identify two basic modes of implementing a stack
(iii) Outline the applications of stacks in computing
(iv) Explain the two methods of storing a stack.

3.0 Learning Content

3.1 The Stack Data Structure

A **stack** is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called **Last-in-First-out (LIFO)**. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.

The operation of the stack can be illustrated as in Fig. 3.1.

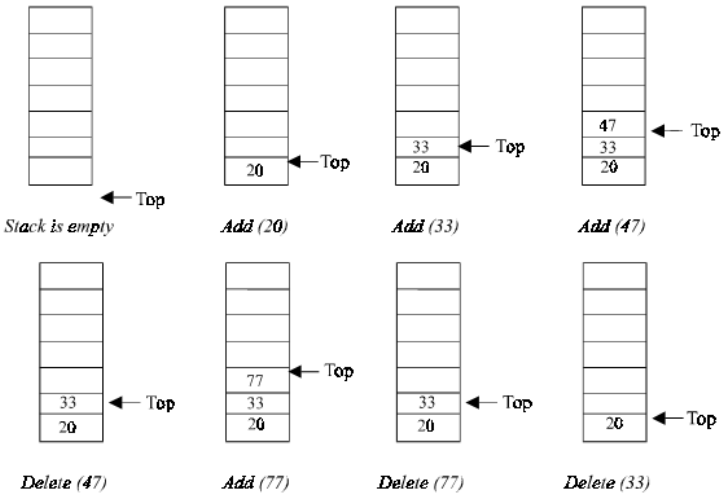


Fig. 3.1. Stack operation.

The insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop. A stack is said to be empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

Self-Assessment Exercise

1. What is insertion operation in stack operations?

Self-Assessment Answer

3.2 Operations Performed on Stack

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

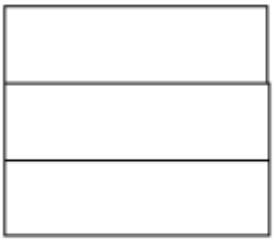
Additional primitives can be defined:

```
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

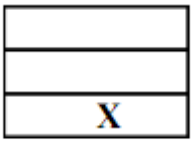
Creates the structure – i.e. ensures that the structure exists but contains no elements

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

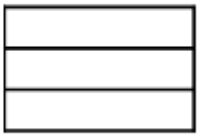


S

Pop

Figure 3.2: Stack after adding the value

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



S

Figure 3.3: Stack after removing the top node

Self-Assessment Exercise(s)

(i) …….. reports whether the stack is empty

(ii) ……. reports whether the stack is full

Self-Assessment Answer (s)

3.3 Stack Implementation

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased). Moreover, static implementation is not an efficient method when resource optimization is concerned (i.e., memory utilization). For example, a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity.

The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

Self-Assessment Exercise

i. List the two ways that stack can be implemented?

Self-Assessment Answer (s)

3.4 Stack using Arrays

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an Element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one-dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

```
1. If TOP = SIZE – 1, then:
(a) Display "The stack is in overflow condition"
(b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit
```

Algorithm for pop

Suppose STACK[SIZE] is a one-dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

```
1. If TOP < 0, then
(a) Display "The Stack is empty"
(b) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit
```

Self-Assessment Exercise(s)

1. Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:
   (a) Display "The stack is in overflow condition"
   (b) …….
2. TOP = TOP + 1
3. ……
4. Exit

Self-Assessment Answer

3.5 Applications of Stacks

There are a number of applications of stacks; three of them are discussed briefly in the preceding sections. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

Self-Assessment Exercise

1. If we want to implement a recursive function non-recursively, ……. explicitly.

Self-Assessment Answer

4.0 Conclusion

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

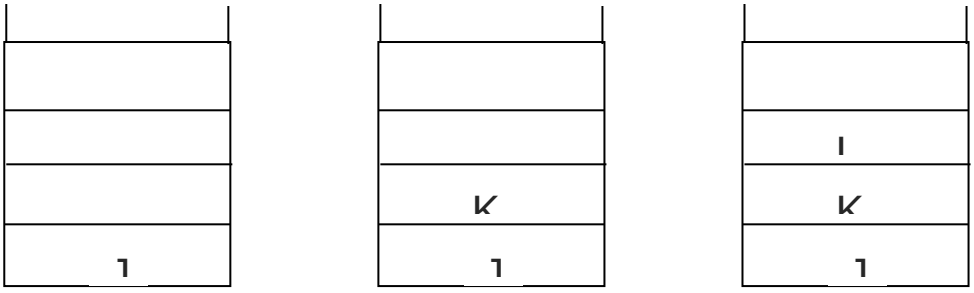
5.0 Summary

You have learnt that:

(i) A stack is one of the most important and useful non-primitive linear data structure in computer science.
(ii) The primitive operations performed on the stack are push and pop
(iii) Stack can be implemented in two ways: Static implementation (using arrays) and Dynamic implementation (using pointers)
(iv) Implementation of stack using arrays is a very simple technique.
(v) Stack is internally used by compiler when we implement (or execute) any recursive function.

6.0 Tutor-Marked Assignment

(1)



S.PUSH('J') S.PUSH('K') S.PUSH('L')

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.

(2) Write on two applications of stacks.

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.

Vinus V. D. (2008). Principles of data structures using C and C++. New age international (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 2

Queues Data Structures

Contents

1.0 Introduction

This unit is on the queue data structure, at the end of this unit, the student will be able to learn about queue data structures as well as its applications and operations. Typical examples are given to facilitate your understanding of these concepts.

2.0 Learning Outcomes

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

ii. Describe a queue data structure
iii. State applications of queues
iv. Explain the operations on a queue
v. Explain two basic modes of queue storage.

3.0 Learning Content

3.1 The Queue Data Structure

A queue is logically a first in first out (FIFO or first come first serve) linear data structure. The concept of queue can be understood by our real-life problems. For example, a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre. It is a homogeneous collection of elements in which new elements are added at one end called rear, and the existing elements are deleted from another end called front.

3.2 Operations on a Queue

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)

2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-1 rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

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

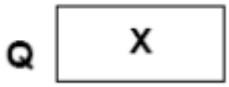


Fig. 1.1: Queue after adding the value X to the tail of Q

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

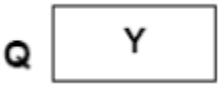


Fig. 1.3: Queue after removing Q from the head node

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

Queue can be implemented in two ways:

1. Using arrays (static)

2. Using pointers (dynamic)

Using arrays (static) is explained in section 3.3 and using pointers (dynamic) is explained in section 3.4

Self-Assessment Exercise

1. Define a queue?

Self-Assessment Answer

3.3 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 is an example of storing a queue in an array of length 5:

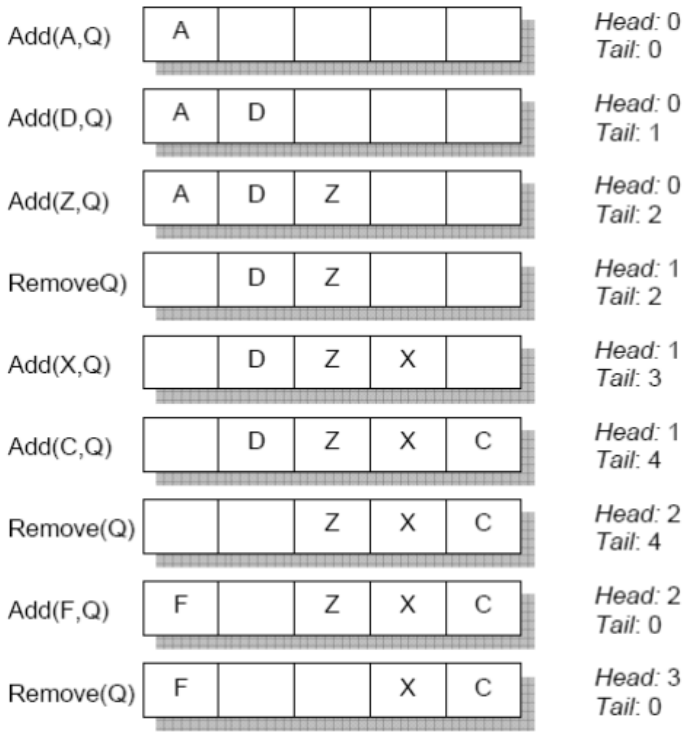


Fig. 1.4: Storing Queue in an array of length 5

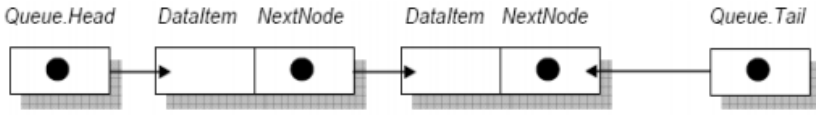
Self-Assessment Exercise

1. Storing a …….. in a static data structure is ……. that stores the queue in ……?

Self-Assessment Answer

3.4 Storing a Queue in a Dynamic Data Structure

A queue requires a reference to the head node AND a reference to the tail node. Figure 1.5 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***

Fig. 1.5: Storage of a Queue.

Adding a Node (Add)

The process of adding a node is as follows: 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. See figure 1.6.

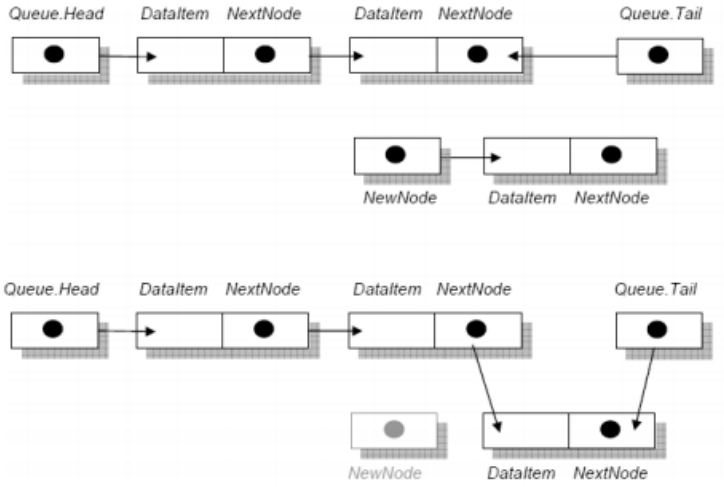


Fig. 1.6: Add a new node to a Queue.

Removing a Node (Remove)

To remove a node: 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. See figure 1.7.

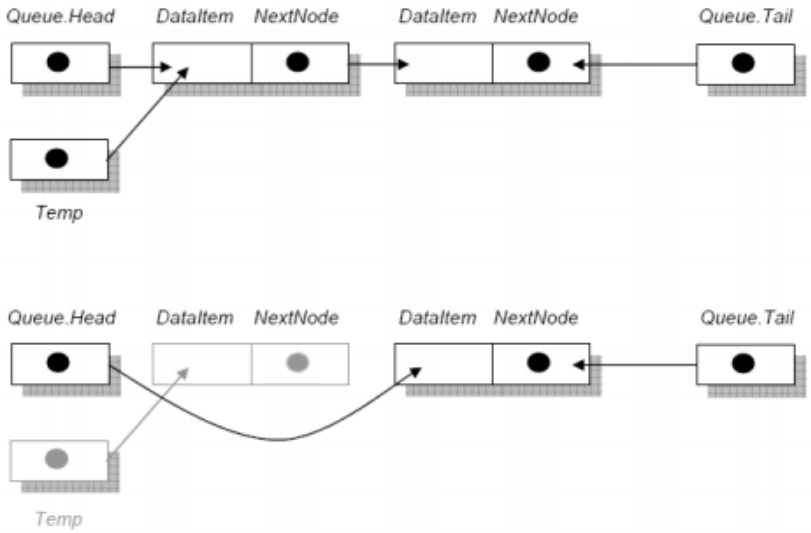


Fig. 1.7: Remove a node from a Queue.

Self-Assessment Exercise

1. What are the requirements for storing a queue in a dynamic data structure?

Self-Assessment Answer

3.5 Other Queues

There are three major variations in a simple queue. They are

1. Circular queue

2. Double ended queue (de-queue)

3. Priority queue

3.6 Applications of Queue

1. Round robin techniques for processor scheduling is implemented using queue.

2. Printer server routines (in drivers) are designed using queues.

3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.

Self-Assessment Exercise

1. List any one applications of queue?

Self-Assessment Answer

4.0 Conclusion

You have learned about the queue data structure in this unit. We also considered Queue applications and operations. You equally have learned about the queue storage in static and dynamic data structures. What you have learned in this unit borders on queues, their operations and applications. The subsequent units shall build upon issues discussed in this unit.

5.0 Summary

You have learnt that:

(i) A queue is logically a first in first out (FIFO or first come first serve) linear data structure.
(ii) The basic operations that can be performed on queue are insert (or add) an element to the queue (push) and delete (or remove) an element from a queue (pop).
(iii) A queue requires a reference to the head node AND a reference to the tail node.
(iv) There are three major variations in a simple queue; Circular queue; Double ended queue (de-queue) and Priority queue.
(v) Applications of queue: Printer server routines (in drivers) are designed using queues.

6.0 Tutor-Marked Assignment

(1) Create a Queue with array of length 4 and show the state of the queue after the following operations:

Add (E,Q)

Remove (Q)

(2) Explain the term Queue and its Pop operation?

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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 3

Hash Tables and Trees

Contents

1.0 Introduction

In this unit, we will look at 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. Also, you will learn about different kinds of trees as well as different tree traversal algorithms. In addition, you will learn how trees can be used to represent arithmetic expressions and how we can estimate an arithmetic expression by doing a tree traversal.

2.0 Learning Outcomes

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

(i) State the basic idea of hashing
(ii) Explain hash keys and functions
(iii) Explain the basic implementation of hash functions
(iv) Describe a hash table
(v) Outline the applications of hash tables.
(vi) Define a tree
(vii) Explain binary trees

3.0 Learning Content

3.1 Hashing

Hashing is a technique where we can compute the location of the desired record in order to retrieve it in a single access (or comparison). Let there is a table of n employee records and each employee record is defined by a unique employee code, which is a key to the record and employee name. If the key (or employee code) is used as the array index, then the record can be accessed by the key directly. If L is the memory location where each record is related with the key. If we can locate the memory address of a record from the key then the desired record can be retrieved in a single access. For notational and coding convenience, we assume that the keys in k and the address in L are (decimal) integers. So the location is selected by applying a function which is called hash function or hashing function from the key k. Unfortunately, such a function H may not yield different values (or index or many address); it is possible that two different keys k_1 and k_2 will yield the same hash address. This situation is called Hash Collision, which is discussed in the next topic.

Self-Assessment Exercise

1. What is hashing?

Self-Assessment Answer

3.2 Hash Function

The basic idea of hash function is the transformation of the key into the corresponding location in the hash table. A Hash function H can be defined as a function that takes key as input and transforms it into a hash table index. Hash functions are of two types:

1. Distribution- Independent function

2. Distribution- Dependent function

We are dealing with Distribution - Independent function. Following are the most popular Distribution - Independent hash functions:

1. Division method

2. Mid Square method

3. Folding method.

Division Method

TABLE is an array of database file where the employee details are stored. Choose a number m, which is larger than the number of keys k. i.e., m is greater than the total number of records in the TABLE. The number m is usually chosen to be prime number to minimize the collision. The hash function H is defined by

```
H(k) = k (mod m)
```

Where H(k) is the hash address (or index of the array) and here k (mod m) means the remainder when k is divided by m.

For example:

Let a company has 90 employees and 00, 01, 02, ... 99 be the two digit 100 memory address (or index or hash address) to store the records. We have employee code as the key.

Choose m in such a way that it is greater than 90. Suppose m = 93. Then for the following employee code (or key k) :

```
H(k) = H(2103) = 2103 (mod93) = 57
H(k) = H(6147) = 6147 (mod 93) = 9
H(k) = H(3750) = 3750 (mod93) = 30
```

Then a typical employee hash table will look like as in Figure 1.

Hash Address	*Employee Code (keys)*	*Employee Name and other Details*
0		
1		
..		
..		
..		
9	6147	Anish
..		
..		
30	3750	Saju
..		
..		
57	2103	Rarish
..		
..		
99		

Figure 1: Hash table

So, if you enter the employee code to the hash function, we can directly retrieve

TABLE[H(k)] details directly. Note that if the memory address begins with 01-m instead of 00-m, then we have to choose the hash function

```
H(k) = k (mod m)+1.
```

Mid Square Method

The key k is squared. Then the hash function H is defined by

$$H(k) = k^2 = l$$

Where l is obtained by digits from both the end of k^2 starting from left. Same number of digits must be used for all of the keys. For example, consider following keys in the table and its hash index:

K	4147	3750	2103
K^2	17197609	14062500	4422609
H(*k*)	97	62	22

17197609
14062500
4422609

Hash Address	*Employee Code (keys)*	*Employee Name and other Details*
0		
1		
..		
..		
..		
22	2103	Giri
..		
..		
62	3750	Suni
..		
..		
..		
97	4147	Renjith
..		
99		

Figure 2: Hash table with mid square division

Folding Method

The key K, k_1, k_2,..... k_r is partitioned into number of parts. The parts have same number of digits as the required hash address, except possibly for the last part. Then the parts are added together, ignoring the last carry. That is

$$H(k) = k_1 + k_2 + \ldots\ldots + k_r$$

Here we are dealing with a hash table with index from 00 to 99, i.e., two-digit hash table. So, we divide the K numbers of two digits.

K	2103	7148	12345
$k_1\ k_2\ k_3$	21, 03	71, 46	12, 34, 5
H(k) $= k_1 + k_2 + k_3$	H(2103) = 21+03 = 24	H(7148) = 71+46 = 19	H(12345) = 12+34+5 = 51

Figure 3: Using folding method

Extra milling can also be applied to even numbered parts, k_2 , k_4, ... are each reversed before the addition.

K	2103	7148	12345
k_1, k_2, k_3	21, 03	71, 46	12, 34, 5
Reversing k_2, k_4	21, 30	71, 64	12, 43, 5
H(k) $= k_1 + k_2 + k_3$	H(2103) = 21+30 = 51	H(7148) = 71+64 = 55	H(12345) = 12+43+5 = 60

Figure 4: Using folding method

```
H(7148) = 71 + 64 = 155, here we will eliminate the leading
carry (i.e., 1).
So H(7148) = 71 + 64 = 55.
```

Self-Assessment Exercise

What is the basic idea of hash function?

Self-Assessment Answer

Please insert Answer to SAE

3.3 Hash Collision

It is possible that two non-identical keys K_1, K_2 are hashed into the same hash address. This situation is called Hash Collision.

Location	(Keys)	Records
0	210	
1	111	
2		
3	883	

4	244	
5		
6		
7		
8	488	
9		

Figure 5: Hash collision table

Let us consider a hash table having 10 locations as shown in Figure 5. Division method is used to hash the key.

```
H(k) = k (mod m)
```

Here m is chosen as 10. The Hash function produces any integer between 0 and 9 inclusions, depending on the value of the key. If we want to insert a new record with key 500 then

```
H(500) = 500(mod 10) = 0.
```

The location 0 in the table is already filled (i.e., not empty). Thus collision occurred.

Collisions are almost impossible to avoid but it can be minimized considerably by introducing any one of the following three techniques:

1. Open addressing

2. Chaining

3. Bucket addressing

Open Addressing

In open addressing method, when a key is colliding with another key, the collision is resolved by finding a nearest empty space by probing the cells.

Suppose a record R with key K has a hash address H(k) = h. then we will linearly

search h + i (where i = 0, 1, 2, ... m) locations for free space (i.e., h, h + 1, h + 2, h + 3 .....hash address).

To understand the concept, let us consider a hash collision which is in the hash table shown in Figure 5. If we try to insert a new record with a key 500 then

```
H(500) = 500(mod 10) = 0.
```

The array index 0 is already occupied by H(210). With open addressing we resolve the hash collision by inserting the record in the next available free or empty location in the table. Here next location, i.e., array hash index 1, is also occupied by the key 111.

Next available free location in the table is array index 2 and we place the record in this free location.

Location	(Keys)	Records
0	210	
1	111	
2	500	
3	883	
4	244	
5		
6		
7		
8	488	
9		

Figure 6: Open addressing method

The position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found. This type of probing is called Linear Probing.

The main disadvantage of Linear Probing is that substantial amount of time will take to find the free cell by sequential or linear searching the table. Other two techniques, which are discussed in the following sections, will minimize this searching time considerably.

QUADRATIC PROBING

Suppose a record with R with key k has the hash address

```
H(k) = h.
```

Then instead of searching the location with address h, h + 1, h + 2,..... h + i ....., we search for free hash address h, h + 1, h + 4, h + 9, h + 16, ... h + i^2

DOUBLE HASHING

Second hash function H_1 is used to resolve the collision. Suppose a record R with key k has the hash address H(k) = h and H1(k) = h^1, which is not equal to m. Then we linearly search for the location with addresses

$h, h + h^1, h + 2h^1, h + 3h^1, ... h + i (h^1)^2$

(where i = 0, 1, 2, ...).

Note: The main drawback of implementing any open addressing procedure is the implementation of deletion.

Chaining

In chaining technique, the entries in the hash table are dynamically allocated and entered into a linked list associated with each hash key. The hash table in Figure 7 can represented using linked list as in Figure 8.

Location	*(Keys)*	*Records*
0	210	30
1	111	12
2		
3	883	14
4	344	18
5		
6	546	32
7		
8	488	31
9		

Figure 7: Hash chaining table

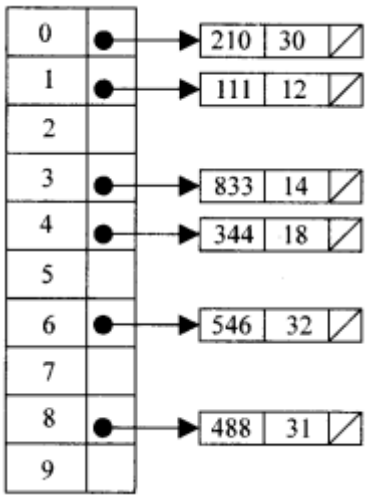


Figure 8: Hash chaining method

If we try to insert record with a key 500 then H(500) = 500(mode 10) = 0. Then the collision occurs in normal way because there exists a record in the 0^{th} position. But in chaining corresponding linked list can be extended to accommodate the new record with the key as shown in Figure 9.

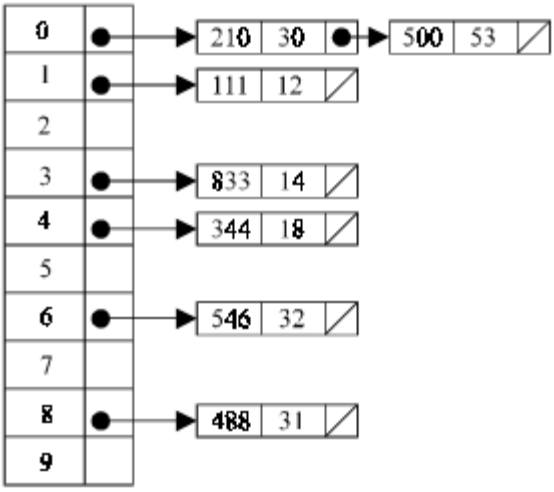


Figure 9: Chaining method

Bucket Addressing

Another solution to the hash collision problem is to store colliding elements in the same position in table by introducing a bucket with each hash address. A bucket is a block of memory space, which is large enough to store multiple items.

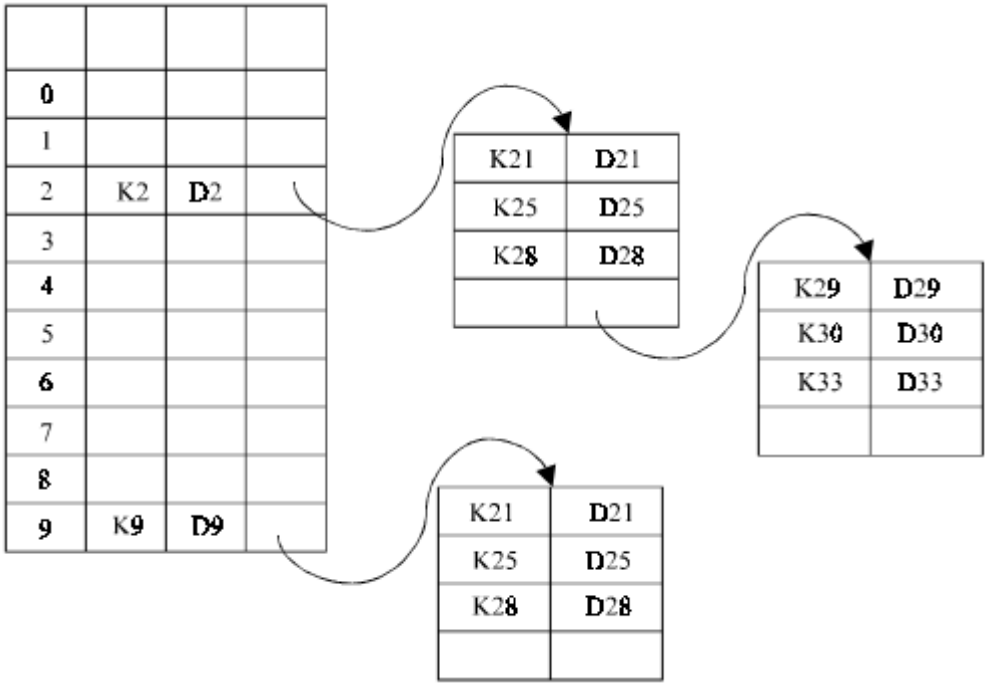


Figure 10: Avoiding collision using buckets

Figure 10 shows how hash collision can be avoided using buckets. If a bucket is full, then the colliding item can be stored in the new bucket by incorporating its link to previous bucket.

Self-Assessment Exercise

1. What is hash collision?

Self-Assessment Answer

3.4 Hash Deletion

A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node from a linked list. But in linear probing, when a data is deleted with its key the position of the array index is made free. The situation is same for other open addressing methods.

3.5 Applications of Hash Tables

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.

Self-Assessment Exercise

1. What is hash deletion?

Self-Assessment Answer

3.6 Trees

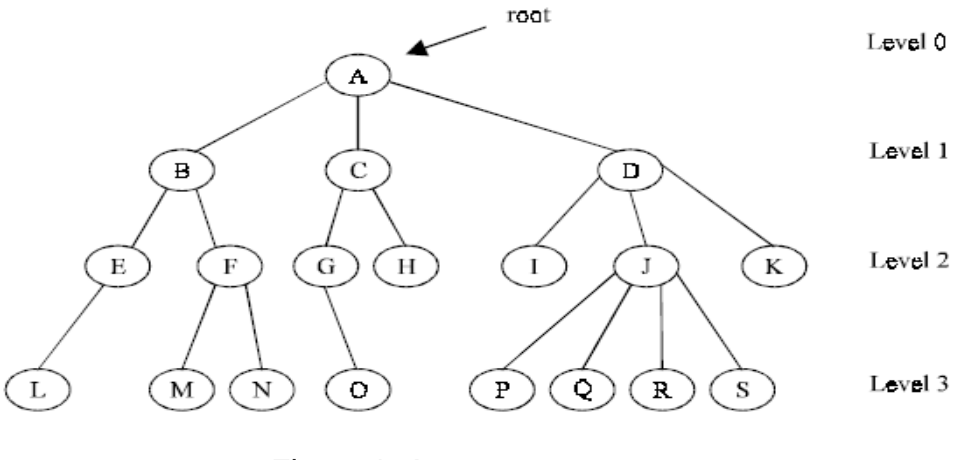
Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grandchildren as so on.

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.

2. Removing nodes (or data item) are partitioned into number of mutually exclusive

(*i.e.,* disjoined) subsets each of which is itself a tree, are called sub tree.



A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.

2. Removing nodes (or data item) are partitioned into number of mutually exclusive

(*i.e.,* disjoined) subsets each of which is itself a tree, are called sub tree.

Basic Terminologies

Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Figure 1. Each data item in a tree is called a *node*. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of subtrees of a node in a given tree. In figure 1

The degree of node A is 3
The degree of node B is 2
The degree of node C is 2
The degree of node D is 3

The degree of a tree is the maximum degree of node in a given tree. In the above tree, degree of a node J is 4. All the other nodes have less or equal degree. So the degree

of the above tree is 4. A node with zero is called a terminal node or leaf. For instance, in figure1 M, N, I O etc are leaf node. Any node whose degree is not zero is called non-terminal node. They are intermediate nodes in traversing the given tree from its root node to the terminal nodes.

The tree structured in different levels. The entire tree is leveled in such a way that the root node is always of level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level n+1.

Depth of a tree is the maximum level of any node in a given tree. That is a number of level one can descend the tree from its root node to the terminal nodes (leaves). The term height is also used to denote the depth.

Trees can be divided in different classes as follows:

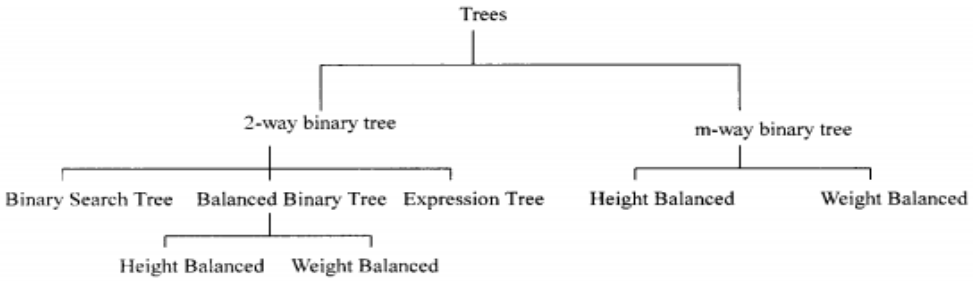


Figure 2: Tree structure

Self-Assessment Exercise

1. A ……. is an ideal data structure for representing ……..

Self-Assessment Answer

3.7 Binary Trees

A binary tree is a tree in which no node can have more than two children. Typically these children are described as “left child” and “right child” of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that;

1) T is empty (i.e if T has no nodes called null tree or empty tree)

2) T contains a special node R, called root node of T, and the remaining nodes of T from an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub-tree of R. if T1 is non-empty then its root is called the left successor of R, similarly if T2 is non-empty then its root is called the right successor of R.

Consider a binary tree T in Figure 3. Here ‘A’ is the root node of the binary tree T. Then ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’ i.e., ‘A’ is a father of ‘B’ and ‘C’. The node ‘B’ and ‘C’ are called brothers, since they are left and right child of the same father. If a node has no child then it is called a leaf node. Nodes P,H,I,F,J are leaf node in Figure 3.

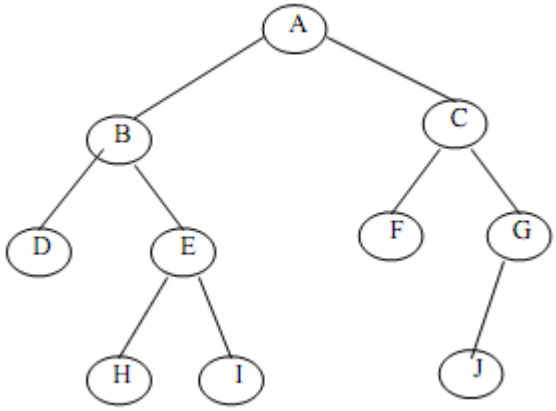


Figure 3: Binary tree

The tree is said to be strictly binary tree, if every non-leaf made in a binary tree has non-empty left and right sub trees. A strictly binary tree with n leaves always contains 2n – 1 nodes. The tree in Figure 4 is strictly binary tree, whereas the tree in Figure 3 is not. That is every node in the strictly binary tree can have either no children or two children. They are also called 2-tree or extended binary tree.

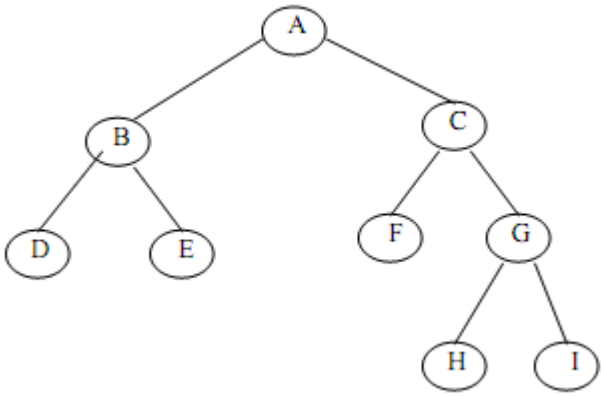


Figure 4: Strictly binary tree

The main application of a 2-tree is to represent and compute any algebraic expression using binary operation.

For example, consider an algebraic expression E.

E = (a + b)/((c – d)*e)

E can be represented by means of the extended binary tree T as shown in Figure 5. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

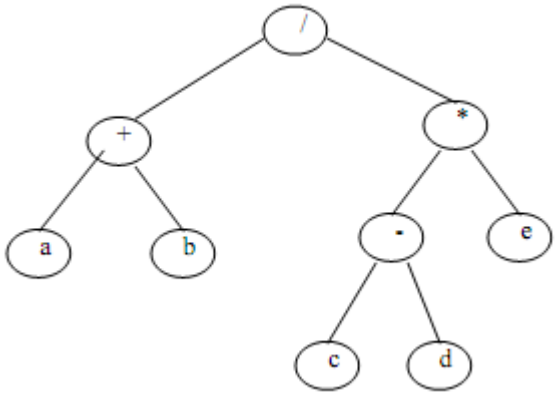


Figure 5: Expression tree

A complete binary tree at depth ‘d’ is the strictly binary tree, where all the leaves are at level d. Figure 6 illustration the complete binary tree of depth 2.

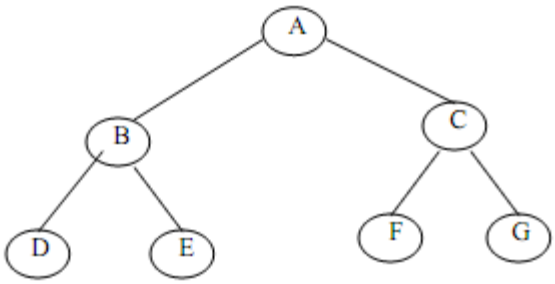


Figure 6: Complete binary tree

A binary tree with n nodes, n > 0, has exactly n – 1 edges. A binary tree of depth d, d > 0, has at least d and at mast 2^d – 1 nodes in it. If a binary tree contains n nodes at level l, then it contains at most 2n nodes at level l + 1. A complete binary tree of depth d is the binary tree of depth d contains exactly 2 l nodes at each level l between 0 and d. Finally, let us discuss in briefly the main difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.

2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.

3. The sub tree of each element in a binary tree are ordered, left and right sub trees. The sub trees in a tree are unordered.

If a binary tree has only left sub trees, then it is called left skewed binary tree. Figure

7(a) is a left skewed binary tree.

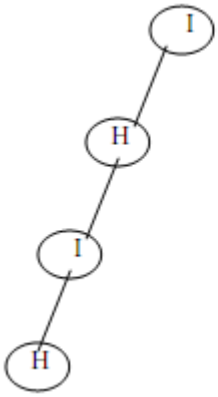


Figure 7(a). Left skewed

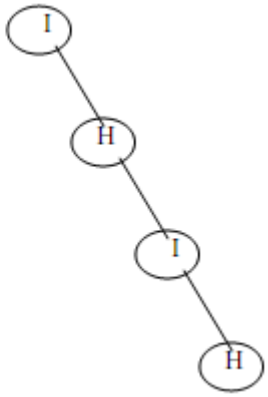


Figure 7(b). Right skewed

If a binary tree has only right sub trees, then it is called right skewed binary tree. Figure 7(b) is a right skewed binary tree.

Binary Tree Representation

This section discusses two ways of representing binary tree T in memory:

1. Sequential representation using arrays

2. Linked list representation

Array Representation

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most $2^d - 1$ nodes can be there in T.(i.e., SIZE = $2^d - 1$) So the array of size "SIZE" to represent the binary tree. Consider a binary tree in Figure 8 of depth 3.

Then SIZE = $2^3 - 1 = 7$. Then the array A[7] is declared to hold the nodes.

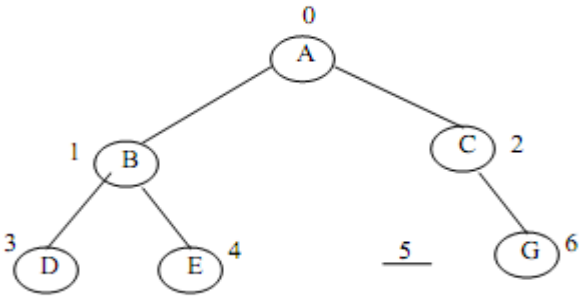


Figure 8: Binary tree of depth 3

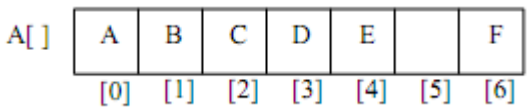


Figure 9: Array representation of the binary tree

The array representation of the binary tree is shown in Figure 9. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

1. The father of a node having index n can be obtained by (n – 1)/2. For example to find the father of D, where array index n = 3. Then the father nodes index can be obtained

= (n – 1)/2

= 3 – 1/2

= 2/2

= 1

i.e. A [1] is the father D, which is B.

2. The left child of a node having index n can be obtained by (2n+1). For example to find the left child of C, where array index n = 2. Then it can be obtained by

= (2n +1)

= 2*2 + 1

= 4 + 1

= 5

i.e. A[5] is the left child of C, which is NULL. So, no left child for C. 3. The right child of a node having array index n can be obtained by the formula (2n + 2). For example, to find the right child of B, where the array index n = 1. Then

= (2n + 2)

= 2*1 + 2

= 4

i.e. A [4] is the right child of B, which is E.4. If the left child is at array index n, then its right brother is at (n + 1). Similarly, if the right child is at index n, then its left brother is at (n – 1).

The array representation is more ideal for the complete binary tree. The Figure 8 is not a complete binary tree. Since there is no left child for node C, i.e. A [5] is vacant. Even though memory is allocated for A [5] it is not used, so wasted unnecessarily.

Linked List Representation

The most popular and practical way of representing a binary tree is using linked list

(or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as:

(a) Left Child (LChild)

(b) Information of the Node (Info)

(c) Right Child (RChild)

The L Child links to the left node of the parent node, info holds the information of every node and R Child holds the address of right child node of the parent node. Figure 10 shows the structure of a binary tree node.

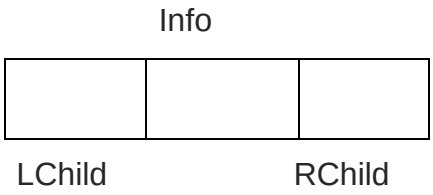


Figure 10: Child table

Following figure 11 shows the linked list representation of the binary tree in figure 8.

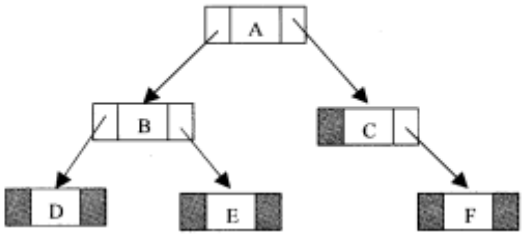


Figure 11: Linked list representation of binary tree

If a node has left or/and right node, corresponding L Child or R Child is assigned to NULL. The node structure can be logically represented in C/C++ as:

```
 Struct Node
{
        Int Info
        struct Node * Lchild;
        struct Node * Rchild;
}
Typedef struct Node*NODE;
```

Operations on Binary Tree

The basic operations that are commonly performed on a binary tree is listed below;

1. Create an empty Binary Tree
2. Traversing a Binary Tree
3. Inserting a New Node
4. Deleting a Node

5. Searching for a Node

6. Copying the mirror image of a tree

7. Determine the total no: of Nodes

8. Determine the total no: leaf Nodes

9. Determine the total no: non-leaf Nodes

10. Find the smallest element in a Node

11. Finding the largest element

12. Find the Height of the tree

13. Finding the Father/Left Child/Right Child/Brother of an arbitrary node

Some primitive operations are discussed in the following sections. Implementation other operations are left to the reader.

Self-Assessment Exercise

1. What is the most popular and practical way of representing a binary tree?

Self-Assessment Answer

3.8 Traversing Binary Trees Recursively

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree.

They are:

1. Pre Order Traversal (Node-left-right)

2. In order Traversal (Left-node-right)

3. Post Order Traversal (Left-right-node)

Pre Orders Traversal Recursively

To traverse a non-empty binary tree in pre order following steps one to be processed

1. Visit the root node

2. Traverse the left sub tree in preorder

3. Traverse the right sub tree in preorder

That is, in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively. It can be implement in C/C++ function as below:

```
Void preorder(Node*Root)
{
    If(Root!=NULL)
    {
        Printf("d\n", Root →Info);
        Preorder(Root → L child);
        Preorder(Root → R child);
    }
}
```

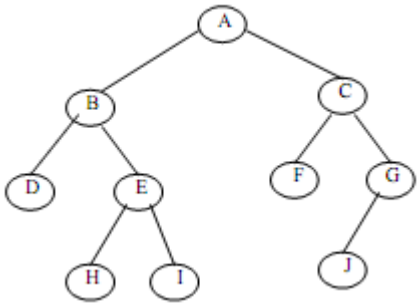


Figure 12: Pre-order tree

The preorder traversal of a binary tree in Fig. 8.12 is A, B, D, E, H, I, C, F, G, J.

In Order Traversal Recursively

The in order traversal of a non-empty binary tree is defined as follows:

1. Traverse the left sub tree in order

2. Visit the root node

3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root.

After visiting the root the right sub tree is traversed recursively, in order fashion. The procedure for an in order traversal is given below:

```
void inorder(Node*Root)
{
    If(Root!=NULL)
    {
        inorder(Root → L child);
        Printf("d\n", Root →Info);

        inorder(Root → R child);
    }
}
```

The in-order traversal of a binary tree in Fig. 8.12 is D, B, H, E, I, A, F, C, J, G.

Post Order Traversal Recursively

The post order traversal of a non-empty binary tree can be defined as:

1. Traverse the left sub tree in post order

2. Traverse the right sub tree in post order

3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

```
void postorder(Node*Root)
{
      If(Root!=NULL)
      {
         postorder(Root → L child);
         postorder(Root → R child);
         Printf("d\n", Root a Info);

      }
}
```

The post order traversal of a binary tree in Fig. 8.12 is D, H, I, E, B, F, J, G, C, A

Traversing Binary Tree Non-Recursively

In this section we will discuss the implementation of three standard traversals algorithms, which were defined recursively in the last section, non-recursively using stack.

Preorder Traversal Non-Recursively

The preorder traversal non-recursively algorithms uses a variable PN (Present Node), which will contain the location of the node currently being scanned. Left(R) denotes the left child of the node R and Right(R) denoted the right child of R. A stack is used to hold the addresses of the nodes to be processed. Info(R) denotes the information of the node R.

Preorder traversal starts with root node of the tree i.e., PN = ROOT. Then repeat the following steps until PN = NULL.

Step 1: Process the node PN. If any right child is there for PN, push the Right (PN) into the top of the stack and proceed down to left by PN = Left (PN), if any left child is there (i.e., Left (PN) not equal to NULL).

Repeat the step 2 until there is no left child (i.e., Left (PN) = NULL).

Step 2: Now we have to go back to the right node(s) by backtracking the tree. This can be achieved by popping the top most element of the stack. Pop the top element from the stack and assigns to PN.

Step 3: If (PN is not equal to NULL) go to the Step 1

Step 4: Exit

The implementation of the preorder non-recursively traversal algorithm can be illustrated with an example. Consider a binary tree in Figure 13. Following steps are generated when the algorithm is applied to the following binary tree:

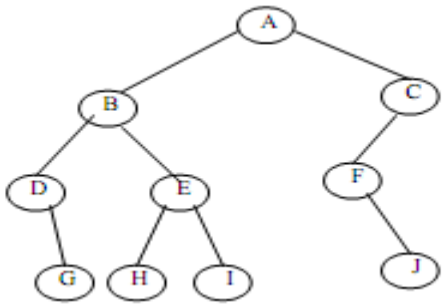


Figure 13: Tree

1. Initialize the Root node to PN

SATCK:

PN = ROOT (i.e., PN = A)

2. Process the node PN (i.e., A)

If PN has the right child push it into stack (i.e., C)

If PN has the left child proceed down to left by PN = Left(A) (i.e., PN = B)

STACK: C

3. Process the node PN (i.e., B)

If PN has the right child (i.e., Right (PN) not equal to NULL) then push the right child of PN into the stack (i.e., Right(B) is E) If PN has the left child proceed down to left by PN = Left(B) (i.e., Now PN = D) is G)

STACK: C, E

4. Process or display the node PN (i.e., D)

If PN has the right child, then push the right child of PN into the stack (i.e., Right(D)

If PN has the left child proceed down to left. Here Left(PN) is equal to NULL, so no left child.

STACK: C, E, G

5. Now the backtracking process will start (i.e., when Left(PN) = NULL)

Pop the top element G from the stack and assign it to PN (i.e., PN = G)

STACK: C, E

6. Process the node G

Check for right child of PN (i.e., G) No right child (i.e., Right(G) = NULL

Check for left child of PN (i.e., G) No left child also (i.e., Left(G) = NULL)

STACK: C, E

7. Again pop the top element E from the stack and assign it to PN (i.e., PN = E)

STACK: C

8. Process the node E (PN)

Since (Right(E) is not equal to NULL)

Push(Right(E)) (i.e., Right(E) is E)

Since (Left(E) is not equal to NULL)

PN = Left(PN) = Left(E) (i.e., PN = H)

STACK: C, I

9. Process the node H

Since (Right(H) = NULL)

Do nothing

Since (Left(H) = NULL)

Do nothing

STACK: C, I

10. Backtracking to right sub tree elements

Pop the top element I from the stack and assign it to PN (i.e., PN = I)

STACK: C

11. Process the node I

No left child for I

No right child for I

STACK: C

12. Again backtracking

Pop the top element C and assign it to PN (i.e., PN=C)

STACK:

13. Display (or process) the node C

Since (Right(C) = NULL)

Do nothing

Since (Left(C) is not equal to NULL)

PN = Left(PN) = Left(C) (i.e., PN = F)

STACK:

14. Display the node F

Since (Right(F) is not equal to NULL)

Push Right(F) to the stack (i.e., J)

Since (Left(F) = NULL)

Do Nothing

STACK: J

15. Backtracking to right node(s)

Pop the top element J and assign it to PN (i.e., PN = J)

STACK:

16. Display the node J

(Right(J) = NULL)

(Right(J) = NULL)

STACK:

17. Backtracking for right nodes. Now the top pointer is pointing to NULL. Assign the top value to PN. (i.e., PN=NULL)

18. When (PN = NULL) STOP

The nodes are processed or displayed in the order A, B, D, G, E, H, I, C, F, J.

ALGORITHM

An array STACK is used to hold the addresses of nodes. TOP pointer points to the top most element of the STACK. ROOT is the root node of tree to be traversed. PN is the address of the present node under scanning. Info(PN) if the information contained in the node PN.

1. Initialize TOP = NULL, PN = ROOT

2. Repeat step 3 to 5 until (PN = NULL)

3. Display Info(PN)

4. If (Right(PN) not equal to NULL)

(a) TOP = TOP+1

(b) STACK(TOP) = Right(PN);

5. If(Left(PN) not equal to NULL)

(a) PN = Left(PN)

6. Else

(a) PN = STACK[TOP]

(b) TOP = TOP–1

7. Exit

In Order Traversal Non-Recursively

The in-order traversal algorithm uses a variable PN, which will contain the location of the node currently being scanned. Info (R) denotes the information of the node R, Left

(R) denotes the left child of the node R and Right (R) denotes the right child of the node R. In-order traversal starts from the ROOT node of the tree (i.e., PN = ROOT). Then repeat the following steps until PN = NULL :

Step 1: Proceed down to left most node of the tree by pushing the root node onto the stack.

Step 2: Repeat the step 1 until there is no left child for a node.

Step 3: Pop the top element of the stack and process the node. PN = STACK[TOP]

Step 4: If the stack is empty then go to step 6.

Step 5: If the popped element has right child then PN = Right(PN). Then repeat the

step from 1.

Step 6: Exit.

The in-order traversal algorithm can be illustrated with an example. Consider a

binary tree in Figure 13. Following steps may generate if we try to traverse the tree in inorder fashion:

1. Initialize root Node to PN (i.e., PN = ROOT = A)

STACK:

2. Since(Left(PN) is not equal to NULL

Push (PN) to the stack

PN = Left(PN) (i.e., = B)

STACK: A

3. Since(Left(PN) is not equal to NULL

Push (PN) to the stack (i.e., = B)

PN = Left(PN) (i.e., = D

STACK: A, B

4. Since(Left(PN) = NULL

Display the node D

STACK: A, B

5. Since(Right(PN) is not equal to NULL

PN = Right(PN) = G

STACK: A, B

6. Since(Left(PN) = NULL

Display the node G

STACK: A, B

7. Since(Right(PN) = NULL)

Pop the topmost element of the stack

PN = STACK[TOP] (i.e., = B)

Display the node B

STACK: A

8. Since(Right(PN) is not equal to NULL)

PN = Right(PN) = E

STACK: A

9. Since(Left(PN) is not equal to NULL

Push(PN) to the stack (i.e., E)

PN = Left(PN) = H

STACK: A, E

10 Since(Left(PN) = NULL)

Display the node H

STACK: A, E

11. Since(Right(PN) = NULL

Pop the topmost element of the stack

PN = STACK[TOP] = E

Display the node E

STACK: A

12. Since(Right(PN) is not equal to NULL)

PN = Right(PN) = I

STACK: A

13. Since(Left(PN) = NULL)

Display the node I

STACK: A

14. Since(Right(PN) = NULL)

Pop the topmost element of the stack

PN = STACK[TOP] = A

Display the node A

STACK:

15. Since(Right(PN) not equal to NULL)

PN = Right(PN) = C

STACK:

16. Since(Left(PN) is not equal to NULL)

Push(PN) to the stack (i.e., = C)

PN = Left(PN) = F

STACK: C

17. Since(Left(PN) = NULL)

Display the node F

STACK: C

18. Since(Right(PN) is not equal to NULL)

PN = Right(PN) = J

STACK: C

19. Since(Left(PN) = NULL

Display the node J

STACK: C

20. Since(Right(PN) = NULL)

Pop the element from the stack

PN = STACK[TOP] = C

Display the node C

STACK:

21. Since(Right(PN) = NULL)

Try to pop an element from the stack. Since the stack is empty PN=NULL and Stop

The nodes are displayed in the order of D, G, B, H, E, I, A, F, J, C.

ALGORITHM

An array STACK is used to temporarily store the addresses of the nodes. TOP pointer always points to the topmost element of the STACK.

1. Initialize TOP = NULL and PN = ROOT
2. Repeat the Step 3, 4 and 5 until (PN = NULL)
3. TOP = TOP +1
4. STACK[TOP] = PN
5. PN = Left(PN)
6. PN = STACK[TOP]
7. TOP = TOP–1
8. Repeat steps 9, 10, 11 and 12 until (PN = NULL)
9. Display Info(PN)
10. If(Right(PN) is not equal to NULL

(a) PN = Right(PN)

(b) Go to Step 6

11. PN = STACK[TOP]
12. TOP = TOP –1
13. Exit

Self-Assessment Exercise

1. What is tree traversal?

Self-Assessment Answer

3.9 Implementing Trees

We will consider the implementation of trees as well as 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 14.

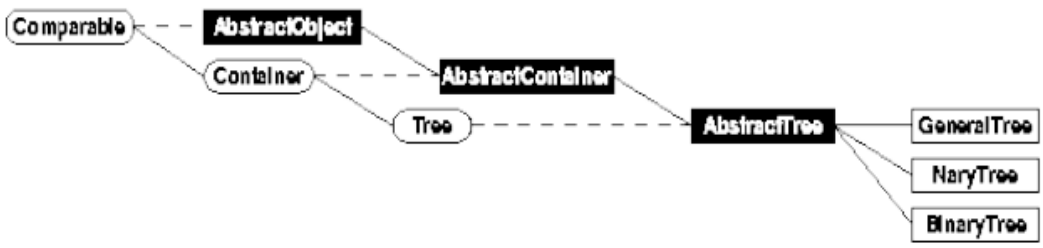


Figure 14: Object class hierarchy

The programme below defines the Tree interface. The Tree interface extends the Container interface defined in this programme.

```
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 breathFirstTraversal (Visitor visitor);
12     }
```

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 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. 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(s) 8

1. What is **isLeaf** in tree implementation?

Self-Assessment Answer (s) 8

4.0 Conclusion

In this unit, you have learned concerning 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 understand the applications of hash tables. Also, you have learned concerning trees. You have as well learned about binary trees and tree traversals. To end with, you have been able to learn how to implement trees.

5.0 Summary

You have learnt that:

(i) Hashing is a technique where we can compute the location of the desired record in order to retrieve it in a single access (or comparison).
(ii) The basic idea of hash function is the transformation of the key into the corresponding location in the hash table.
(iii) It is possible that two non-identical keys K_1, K_2 are hashed into the same hash address. This situation is called Hash Collision.
(iv) A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node from a linked list.
(v) 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.
(vi) Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grandchildren as so on.
(vii) A binary tree is a tree in which no node can have more than two children. Typically
2. these children are described as “left child” and “right child” of the parent node.
(viii) Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner.

6.0 Tutor-Marked Assignment

(1) Explain the term hashing? What is the basic idea of hash function?

(2) Using a suitable example, explain hash table?

(3) Explain trees and illustrate it further by means of a diagram.

(4) Briefly describe the implementation of trees.

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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 4

Search Trees and Graphs

Contents

1.0 Introduction

This unit gives introduction to Search Trees, describing successful and unsuccessful searching. Also, we show the implementation of AVL search trees. This unit also discusses another nonlinear data structure, graphs. Graphs representations have found application in almost all subjects like geography, engineering and solving games and puzzles, 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 Learning Outcomes

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

(i) Explain a search tree

(ii) State the process for implementing AVL search trees.

(iii) Explain the graph theory, stating some of its applications

(iv) State the algorithmic transpose

3.0 Learning Content

3.1 Search Tree-Basics

A *search tree* is a tree which supports efficient search, insertion, and withdrawal operations. Therefore, 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 keen on a search tree is that the keys do not appear in arbitrary nodes of the tree. In its place, there is a data *ordering criterion* which determines where a specified key may appear in the tree in relation to the other keys in that tree. In the subsequent sections we present two related types of search trees, *M*-way search trees and binary search trees.

Searching a Search Tree

The main benefit 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. Search trees are defined recursively so; it is easy to define a recursive search method.

Searching an *M*-way Tree

Think of search for a particular item, say *x*, in an *M*-way search tree. The search all the time 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_O 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.

Note 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.

At what time 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. Consequently, the worst-case running time for an unsuccessful search is determined by the *height* of the search tree.

Searching a Binary Tree

The above described search method applies directly to binary search trees. As stated 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 shows two binary search trees. The tree T_a is an instance of a particularly bad search tree for the reason that 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.

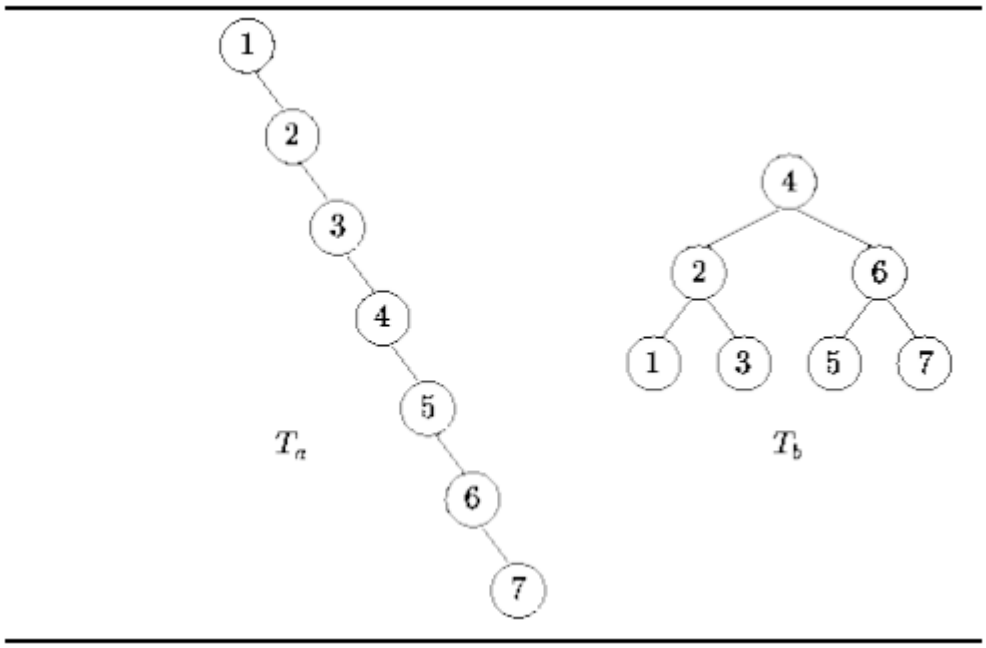


Figure 1: Examples of search trees

Then again, tree T_b in Figure 1 is an example of a particularly good binary search tree. This tree is a case of a *perfect binary tree*.

Definition (Perfect Binary Tree) A *perfect binary tree* of height h≥0 is a binary tree T= {r, T_L, T_R} with the following properties:

(1) If *h*=0, then T_L = 0 and T_R = 0.

(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. On the other hand, 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 visit exactly *h*+1 internal nodes, where h= $\log_2(n+1)$. So, the worst case for unsuccessful search in a perfect tree is O (log n).

Successful Search

While a search is successful, precisely d+ 1 internal node 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 $\bar{d}$ the average of the depths of the nodes is 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.

At this time consider an arbitrary tree, $T_n(l)$, having n≥1 internal nodes altogether, *l* of which are found in its left subtree, where 0≤l<n. Such a tree consists of a root, the left subtree with *l* internal nodes and 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 $T_n(l)$.

Consecutively 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 Tn(l) average over all possible sizes, l, of the (left) subtree, 0≤l<n..

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

If we suppose that it is equally likely for any of the *n* keys to be selected as the root, th en all the subtree sizes in the range 0≤l<n are equally likely. Therefore, the average internal path length for a tree with n≥1 nodes is

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

…… (10)

Therefore, in order to determine *I*(*n*), we need to solve the recurrence

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

In solving this recurrence, we consider the case *n*>1 and then multiply

Equation 1 by *n* to get

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

Given that this equation is valid for any *n*>1, by substituting *n*-1 for *n,* we can also write

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

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:

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

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

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

Unsuccessful Search

Every successful searches end when the object of the search is found. Consequently, 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 previous 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 1: 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 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*=2*n*.

Proof (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, …, k for some k ≥ 1. Consider an arbitrary tree, T_k, that has *k* internal nodes. According to Theorem 1, 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

$$\begin{aligned} E_{k+1} - I_{k+1} &= (E_k + d + 2) - (I_k + d) \\ &= E_k - I_k + 2 \\ &= 2(k+1). \end{aligned} \qquad (10.6)$$

So, 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 2*n* for all n≥1.

Since the difference between the internal and external path lengths of any tree with *n* internal nodes is 2*n*, 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

$$\begin{aligned} E(n) &= I(n) + 2n \\ &= 2(n+1)H_n - 2n \\ &\approx 2(n+1)(\ln n + \gamma) - 2n. \end{aligned} \qquad (10.7)$$

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 articulate 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(logn). 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.

Self-Assessment Exercises

1. What is a search tree?

Self-Assessment Answer

3.2 AVL Search Trees

The difficulty 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 instance, 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.

Obviously, T_b is a better search tree than T_a. In fact, since 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 cause 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. 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.

Evidently, 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 2: 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 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 $N_h=N_h + N_{h-2} + 1$, where $h \geq 2$.

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

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

The remarkable thing about Equation 10.8 is its similarity with the definition of *Fibonacci numbers*. 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 $h = 0, 1, 2, \ldots, k$.
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 2, 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,

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

This completes the proof of the upper bound.

As a result, 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 O(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.

Implementing AVL Trees

Haven 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 1.0. 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.

```
public class AVLTree
       extends BinarySearchTree
{
         protected int height;
          //…..
}
```

Programme 1.0: AVLTree fields.

Programme 1.0 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 likely 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.

Inserting Items into an AVL Tree

There is two-part process in inserting an item into an AVL tree. 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 like 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 h_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

$$h_L - h'_R = 0$$

Actually, 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.

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:

```
public class AVLTree
          extends BinarySearchTree
{
            protected int height;
            public Object detachKey()
            {
                    height = -1;
                     return super.detachKey();
            }
             //…..
}
```

Programme 1.1: AVLTree class detachKey method

Self-Assessment Exercise

1. What is the difficulty with binary search tree?

Self-Assessment Answer

Please insert Answer to SAE

3.3 The Graph Theory

A graph G consist of

1. Set of vertices V (called nodes), (V = {V_1, V_2, V_3, V_4, ,.....}) and

2. Set of edges E (*i.e.,* E {e_1, e_2, e_3, ...cm}

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge '*e*' in E is identified with a unique pair (*a*, *b*) of nodes in V, denoted by *e* = [*a*, *b*].

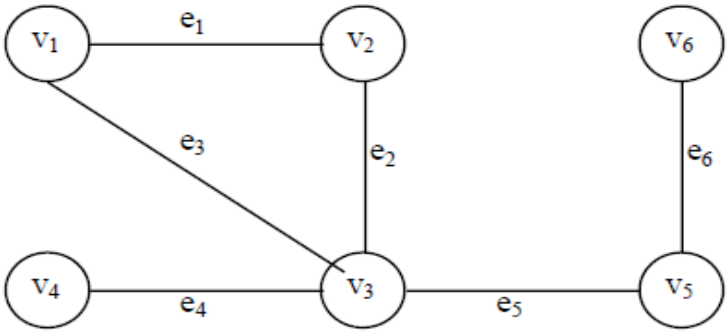


Figure 1: Graph

Consider a graph, G in Figure 1. Then the vertex V and edge E can be represented as:

V = {v_1, v_2, v_3, v_4, v_5, v_6} and E = {e_1, e_2, e_3, e_4, e_5, e_6} E = {(v_1, v_2) (v_2, v_3) (v_1, v_3,) (v_3, v_4), (v_3, v_5) (v_5, v_6)}. There are six edges and vertex in the graph

Basic Terminologies

A *directed graph* G is defined as an ordered pair (V, E) where, V is a set of vertices and the ordered pairs in E are called edges on V. A directed graph can be represented geometrically as a set of marked points (called vertices) V with a set of arrows (called edges) E between pairs of points (or vertex or nodes) so that there is at most one arrow from one vertex to another vertex. For example, Figure 2 shows a directed graph, where G = {*a*, *b*, *c*, *d* }, {(*a*, *b*), (*a*, *d*), (*d*, *b*), (*d*, *d*), (*c*, *c*)}

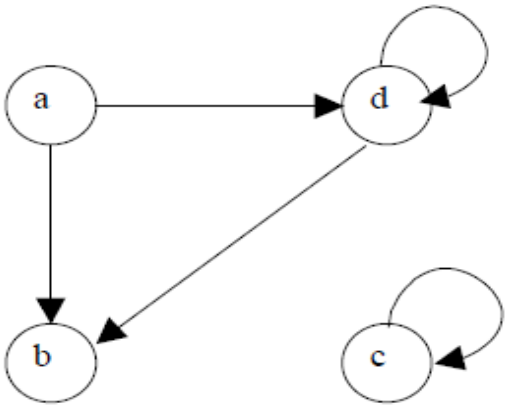


Figure 2: Directed graph

An edge (*a*, *b*), in said to the incident with the vertices it joints, *i.e., a*, *b*. We can also say that the edge (*a*, *b*) is incident from *a* to *b*. The vertex a is called the *initial vertex* and the vertex *b* is called the *terminal vertex* of the edge (*a*, *b*). If an edge that is incident from and into the same vertex, say (*d*, *d*) of (*c*, *c*) in Figure 2, is called a *loop*.

Two vertices are said to be adjacent if they are joined by an edge. Consider edge (*a*, *b*), the vertex *a* is said to be adjacent to the vertex *b*, and the vertex *b* is said to be adjacent from the vertex *a*. A vertex is said to be an *isolated vertex* if there is no edge incident with it. In Figure 2 vertex C is an isolated vertex.

An *undirected graph* G is defined abstractly as an ordered pair (V, E), where V is a set of vertices and the E is a set at edges. An undirected graph can be represented geometrically as a set of marked points (called vertices) V with a set at lines (called edges) E between the points. An undirected graph G is shown in Figure 3.

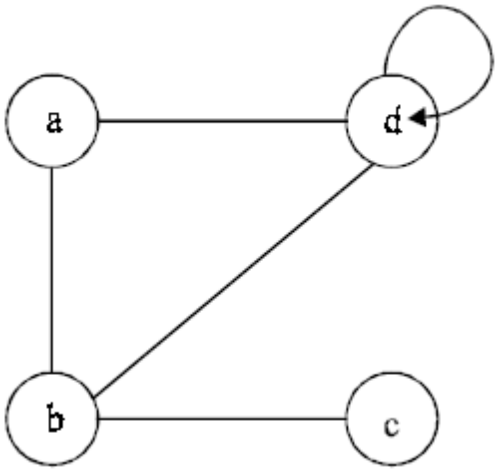


Figure 3: Undirected graph

Two graphs are said to be *isomorphic* if there is a one-to-one correspondence between their vertices and between their edges such that incidence are prevented. Figure 4 show an isomorphic undirected graph.

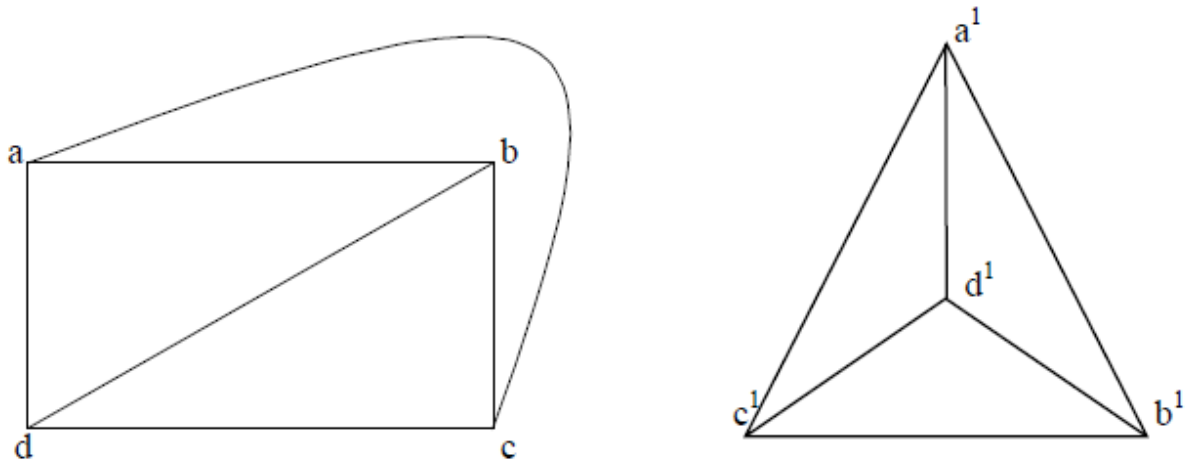


Figure 4: Isomorphic graph

Let G = (V, E) be a graph. A graph $G^1 = (V^1, E^1)$ is said to be a *sub-graph* of G if E^1 is a subset at E and V^1 is a subset at V such that the edges in E1 are incident only with the vertices in V^1. For example Figure 5 (*b*) is a sub-graph at Figure 5(*a*). A sub-graph of G is said to be a *spanning sub-graph* if it contains all the vertices of G. For example Figure 5(*c*) shows a spanning sub-graph at Figure 5(*a*).

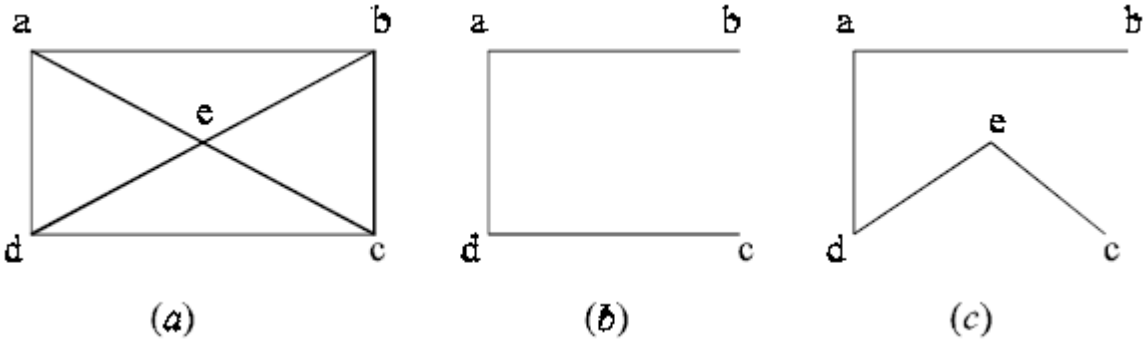


Figure 5: Spanning sub-graph

The number of edges incident on a vertex is its *degree*. The degree of vertex *a*, is written as degree (*a*). If the degree of vertex a is zero, then vertex *a* is called isolated vertex. For example, the degree of the vertex *a* in Figure 5 is 3.

A graph G is said to be *weighted graph* if every edge and/or vertex in the graph is assigned with some weight or value. A weighted graph can be defined as G = (V, E, W_e, W_v) where V is the set of vertices, E is the set at edges and W_e is a weight of the

edges whose domain is E and W_v is a weight to the vertices whose domain is V. Consider a graph.

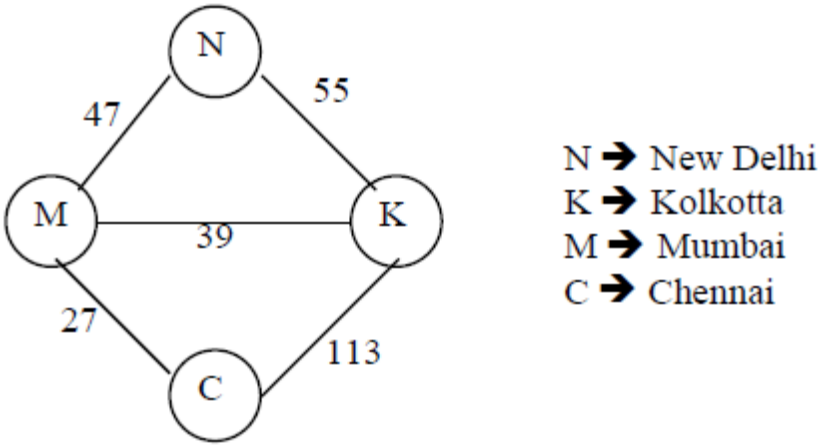


Figure 6: Weighted graph

In Figure 6 which shows the distance in km between four metropolitan cities in India.

Here V = {N, K, M, C,} E = {(N, K), (N, M,), (M, K), (M, C), (K, C)} W*e* = {55,47, 39, 27, 113} and

W_v = {N, K, M, C} The weight at the vertices is not necessary to maintain have become the set W_v and V are same.

An undirected graph is said to be *connected* if there exist a path from any vertex to any other vertex. Otherwise it is said to be *disconnected*.

Figure 7 shows the disconnected graph, where the vertex *c* is not connected to the graph. Figure 8 shows the connected graph, where all the vertexes are connected.

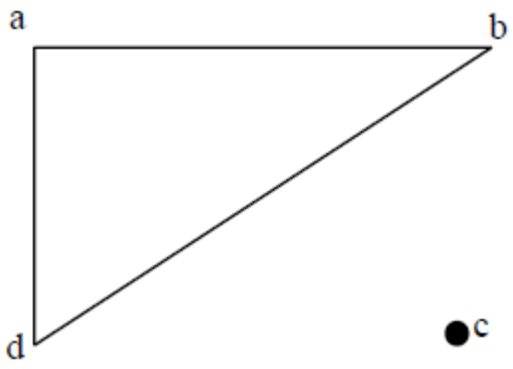


Figure 7: Disconnected graph

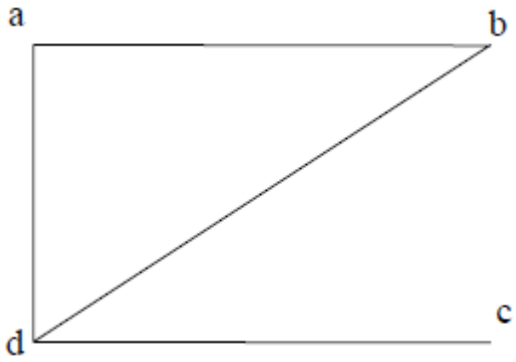


Figure 8: Connected graph

A graph G is said to complete (or fully connected or strongly connected) if there is a path from every vertex to every other vertex. Let *a* and *b* are two vertices in the directed graph, then it is a complete graph if there is a path from *a* to *b* as well as a path from *b* to *a*. A complete graph with *n* vertices will have *n* (*n* – 1)/2 edges. Figure 9 illustrates the complete undirected graph and Figure 10 shows the complete directed graph.

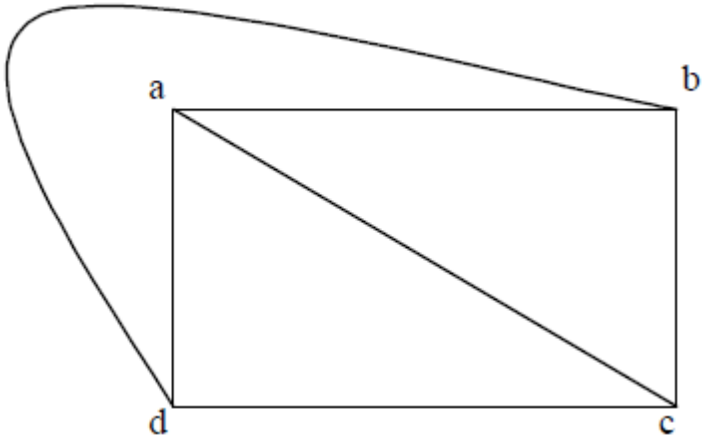


Figure 9: complete undirected graph

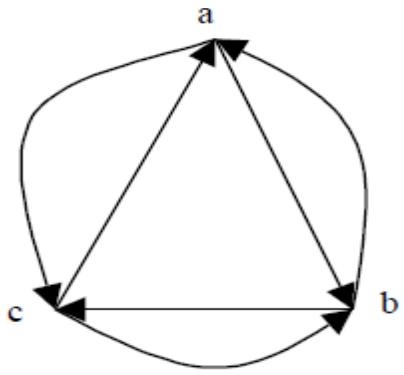


Figure 10: complete directed graph

In a directed graph, a *path* is a sequence of edges (e_1, e_2, e_3, ... e_n) such that the edges are connected with each other (*i.e.,* terminal vertex e_n coincides with the initial vertex e_1). A path is said to be *elementary* if it does not meet the same vertex twice. A path is said to be *simple* if it does not meet the same edges twice. Consider a graph in Figure 11.

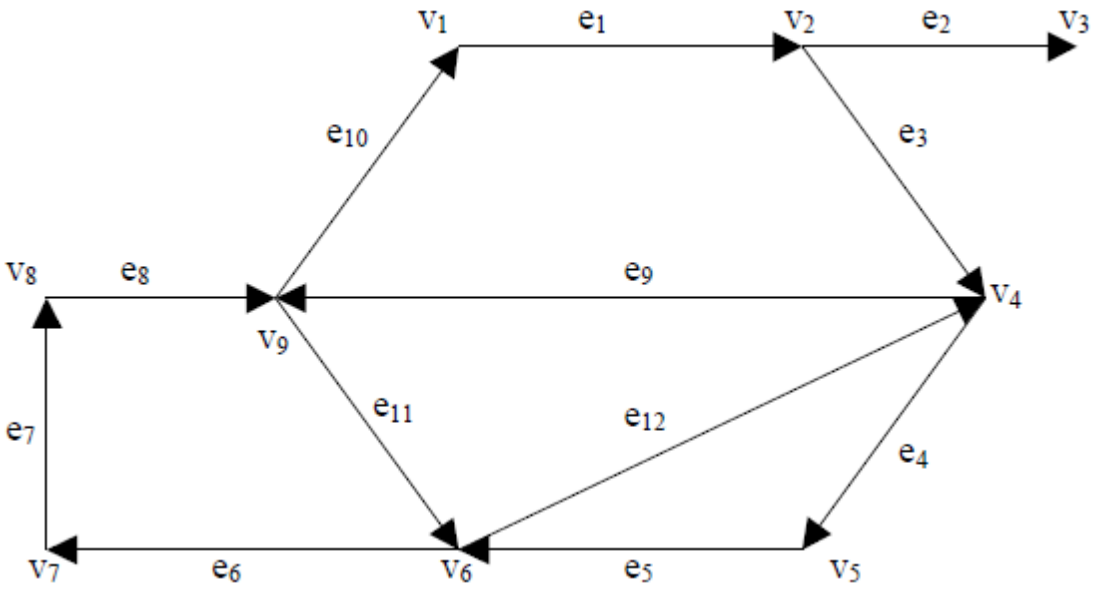


Figure 11: Graph with simple paths

Where (e_1, e_2, d_3, e_4, e_5) is a path; (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{11}, e_6, e_7, e_8, e_{11}) is a path but not a simple one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{11}, e_{12}) is a simple path but not elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8) is an elementary path.

A circuit is a path (e_1, e_2, ... e_n) in which terminal vertex of en coincides with initial vertex of e1. A circuit is said to be simple if it does not include (or visit) the same edge twice. A circuit is said to be elementary if it does not visit the same vertex twice. In Figure 11 (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{10}) is a simple circuit but not a elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{10}) is an elementary circuit.

Self-Assessment Exercise

1. How can you represent a graph?

Self-Assessment Answer

3.4 Representation of Graph

Graph is a mathematical structure and finds its application is many areas, where the problem is to be solved by computers. The problems related to graph G must be represented in computer memory using any suitable data structure to solve the same. There are two standard ways of maintaining a graph G in the memory of a computer.

1. Sequential representation of a graph using adjacent

2. Linked representation of a graph using linked list

3.5 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 $A^T = (Ta_{ij})$ given by $Ta_{ij} = a_{ji}$. In other words, rows of matrix A become columns of matrix AT 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*) □ V×V: (u, *v*)□E. Thus, GT 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:

Algorithm Matrix Transpose (G, GT)

For i = 0 to i < V[G]

For j = 0 to j V[G]

GT (j, i) = G(i, j)

j = j + 1;

i = i + 1

Self-Assessment Exercise

1. State the algorithm matrix transpose (G, GT)?

Self-Assessment Answer

4.0 Conclusion

In this unit, you have been educated about trees. You have also learned about binary trees and tree traversals. You have been able to learn how to implement trees. The graph theory and representation of graph were considered in this unit. Also, you have learned about algorithmic transpose.

5.0 Summary

You have learnt that:

(i) A *search tree* is a tree which supports efficient search, insertion, and withdrawal operations.

(ii) The difficulty 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.

(iii) (iii) A graph G consist of set of vertices V (called nodes), (V = {V_1, V_2, V_3, V_4, ,.....}) and set of edges E (*i.e.,* E {e_1, e_2, e_3, ...cm}

(iv) Graph is a mathematical structure and finds its application is many areas, where the problem is to be solved by computers.

(v) 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.

6.0 Tutor-Marked Assignment

(1) Define a search tree

(2) 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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Module 3

Sorting

Unit 1 Sorting and Bubble Sort

Unit 2 Insertion Sort

Unit 3 Selection Sort

Unit 4 Merge Sorting

Unit 1

Sorting and Bubble Sort

Contents

1.0 Introduction

This unit explains sorting algorithm. It also considers the two kinds of sorting as well as the classes of sorting. In this unit also, you will learn about bubble sort, implementation and memory requirement for bubble sort.

2.0 Learning Outcomes

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

(i) Explain the intend of sorting algorithm

(ii) Explain the types of sorting

(iii) Classify sorting algorithm

(iv) Describe the bubble sort

(v) State the implementation of a bubble sort.

3.0 Learning Content

3.1 Sorting

The purpose of the sorting algorithm is to reorganize the records so that their keys are ordered according to various well-defined ordering regulations.

Problem: Given an array of n real number A[1.. n].

Objective: Sort the elements of A in ascending order of their values.

Internal Sort

Whenever 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.

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.

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.

Self-Assessment Exercise

1. What is the purpose of sorting algorithm?

Self-Assessment Answer

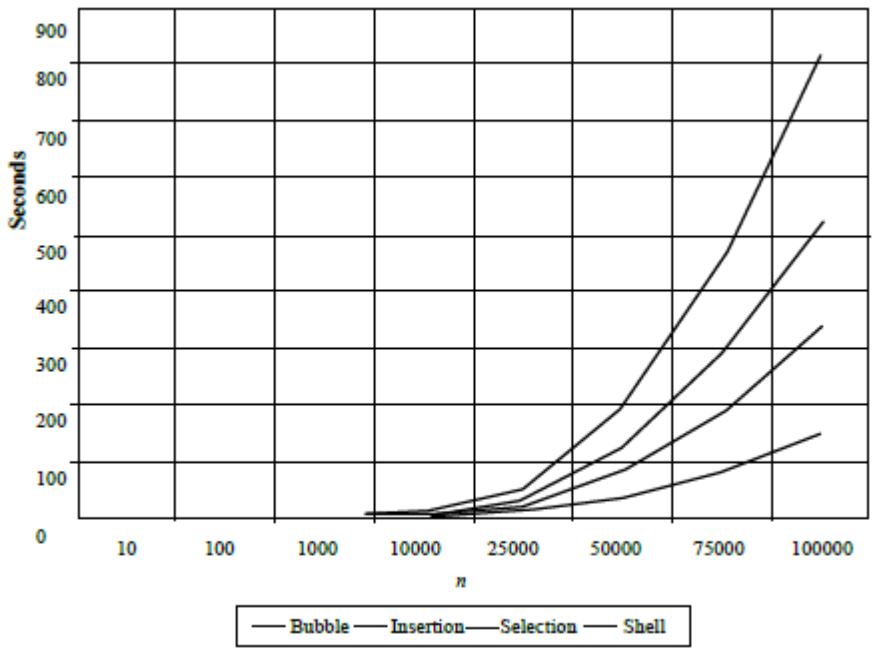
3.2 Stability

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.

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. When is a sorting algorithm stable?

Self-Assessment Answer

3.3 Bubble Sort

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Let A be a linear array of *n* numbers. Sorting of A means rearranging the elements of A so that they are in order. Here we are dealing with ascending order. *i.e.,* A[1] < A[2] < A[3] < ..... A[*n*].

Suppose the list of numbers A[1], A[2], …… A[*n*] is an element of array A. The bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in the (or desired) ascending order, so that A[1] < A[2].that is if A[1] is greater than A[2] then interchange the position of data by swap = A[1]; A[1] = A[2]; A[2] = swap. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Continue the process until we compare A[N – 1] with A[N].

Note: Step1 contains *n* – 1 comparisons *i.e.,* the largest element is “bubbled up” to the *n*th position or “sinks” to the *n*th position. When step 1 is completed A[N] will contain the largest element.

Step 2: Repeat step 1 with one less comparisons that is, now stop comparison at A [*n* – 1] and possibly rearrange A[N – 2] and A[N – 1] and so on.

Note: in the first pass, step 2 involves n–2 comparisons and the second largest element will occupy A[*n*-1]. And in the second pass, step 2 involves *n* – 3 comparisons and the 3rd largest element will occupy A[*n* – 2] and so on.

Step *n* – 1: compare A[1]with A[2] and arrange them so that A[1] < A[2]

After *n* – 1 steps, the array will be a sorted array in increasing (or ascending) order.

The following figures will depict the various steps (or PASS) involved in the sorting of an array of 5 elements. The elements of an array A to be sorted are: 42, 33, 23, 74, 44

FIRST PASS

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

23 swapped	23	23
33	33 no swapping	33
42	42	42 no swapping
44	44	44
74	74	74

23 no swapping	23
33	33 no swapping
42	42
44	44
74	74

23 no swapping
33
42
44
74

Thus the sorted array is 23, 33, 42, 44, 74.

ALGORITHM

Let A be a linear array of *n* numbers. Swap is a temporary variable for swapping (or interchange) the position of the numbers.

1. Input *n* numbers of an array A
2. Initialize $i = 0$ and repeat through step 4 if ($i < n$)
3. Initialize $j = 0$ and repeat through step 4 if ($j < n - i - 1$)
4. If (A[*j*] > A[*j* + 1])

(*a*) Swap = A[*j*]

(*b*) A[*j*] = A[*j* + 1]

(*c*) A[*j* + 1] = Swap

5. Display the sorted numbers of array A
6. Exit.

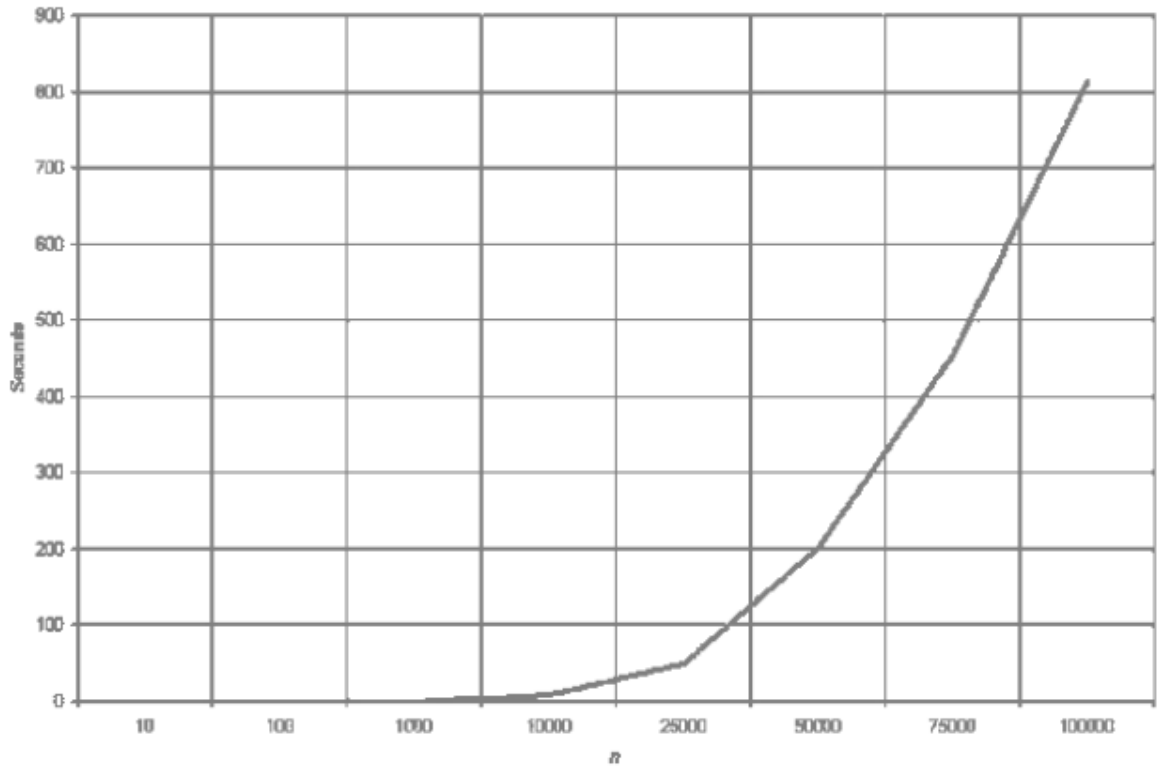


Figure 1.0 n^2 nature of the bubble sort

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.

Memory Requirement

Clearly, bubble sort does not require extra memory.

Self-Assessment Exercise

1. What is bubble sort?

Self-Assessment Answer

3.4 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;
}
}
}
}
```

4.0 Conclusion

In this unit, you have learned concerning sorting algorithm. You have as well been able to identify classes of sorting algorithm. Also, in this unit, you have learned about bubble sort. You have also learned about its memory requirement and implementation. What you have learned limits on sorting algorithms and their classes.

5.0 Summary

You have learnt that:

(i) The purpose of the sorting algorithm is to reorganize the records so that their keys are ordered according to various well-defined ordering regulations.

(ii) A sorting algorithm is called stable if it preserves the relative order of equal keys in the file.

(iii) In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed.

6.0 Tutor-Marked Assignment

(1) Name two classes of sorting algorithm.

(2) Describe the internal 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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 2

Insertion Sort

Contents

1.0 Introduction

In the previous unit you learnt about sorting and bubble sort. This unit is a continuation of sorting. Therefore, in this unit you will learn about insertion sort and its analysis. You will equally learn about the stability and implementation of insertion sort.

2.0 Learning Outcomes

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

(i) Describe the insertion sort

(ii) Analyze an insertion sort

(iii) Describe the stability of an insertion sort

(iv) State how insertion sort is implemented.

3.0 Learning Content

3.1 Insertion Sort

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first, if the first element is greater than second; place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place. And place the third value to first place and so on.

Let A be a linear array of n numbers A [1], A [2], A [3], ... A[*n*]. The algorithm scan the array A from A [1] to A [n], by inserting each element A[k], into the proper position of the previously sorted sub list. A [1], A [2], A [3], ... A [*k* – 1]

Step 1: As the single element A [1] by itself is sorted array.

Step 2: A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2]

is sorted array.

Step 3: A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A [1] and A [2] and placed before A [1], between A [1] and A [2], or after A [2] so that A [1], A [2], A [3] is a sorted array.

Step 4: A [4] is inserted in to a proper place in A [1], A [2], A [3] by comparing it; so that A [1], A [2], A [3], A [4] is a sorted array.

Step 5: Repeat the process by inserting the element in the proper place in array

Step n : A [*n*] is inserted into its proper place in an array A [1], A [2], A [3], ... A [*n* – 1] so that A [1], A [2], A [3], ... ,A [*n*] is a sorted array.

Insertion Sort (A)

1. For j = 2 to length [A] do
2. key = $A[j]$
3. {Put $A[j]$ into the sorted sequence $A[1 . . j\text{-}1]$
4. $i \leftarrow j$ -1
5. while $i > 0$ and $A[i] >$ key do
6. $A[i+1] = A[i]$
7. $i = i$-1
8. $A[i+1]$ = key

Self-Assessment Exercise

1. What is insertion sort?

Self-Assessment Answer

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.

$T(n) = an + b = O(n)$

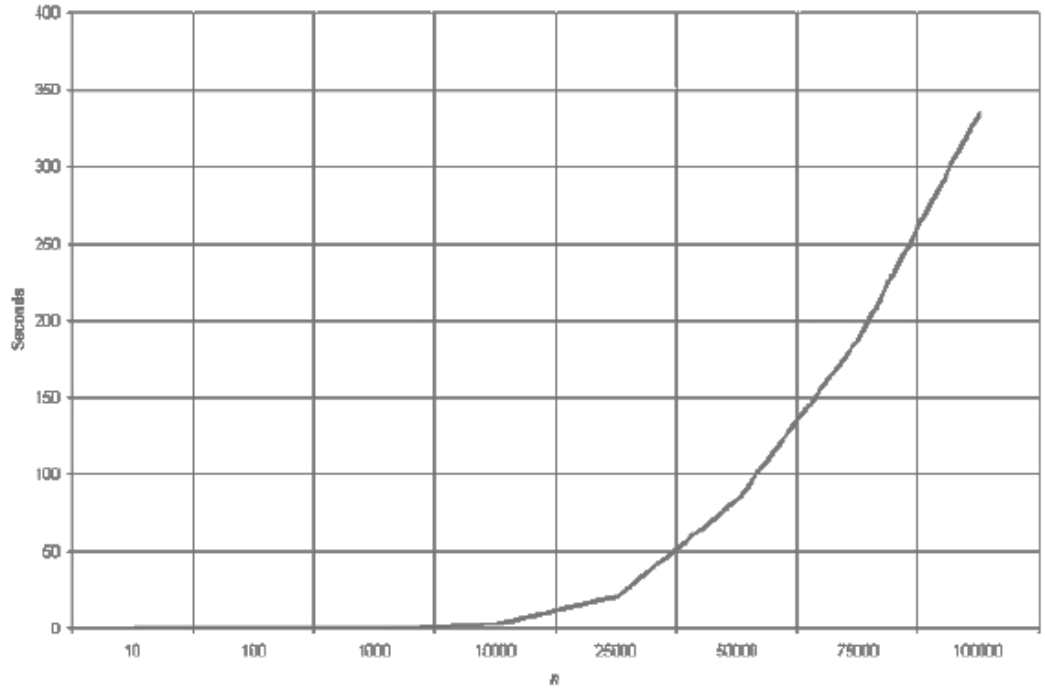
It is a linear function of n.

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.

Stability

In view of the fact that 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.

Self-Assessment Exercise

1. Explain the best-case of insertion sort?

Self-Assessment Answer

3.3 Extra Memory

This algorithm does not need 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.

3.4 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

1. State the running time for worst-case of insertion sort?

Self-Assessment Answer

4.0 Conclusion

In this unit you have learned about insertion sort. Furthermore, you have also learned about the analysis stability and implementation of insertion sort. What you have learned in this unit borders on insertion sort, its analysis and implementation.

5.0 Summary

You have learnt that:

(i) Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file.

(ii) 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.

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

(iii) This algorithm does not need extra memory.

6.0 Tutor-Marked Assignment

(1) List the pseudocode for an insertion sort

(2) Explain why 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.

Vinus V. D. (2008). Principles of data structures using C and C++. New AgeInternational (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 3

Selection Sort

Contents

1.0 Introduction

What you learnt in the previous unit was on insertion sort. This unit is a continuation of sorting. In this unit, we will look at selection sort, differentiating it from insertion sort. In addition, the implementation of selection sort is also discussed.

2.0 Learning Outcomes

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

(i) Explain selection sort

(ii) Differentiate selection sort from insertion sort

(iii) Explain the straight selection sort

(iv) Describe the implementation of selection sort

3.0 Learning Content

3.1 Selection Sorting

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Let A be a linear array of '*n*' numbers, A [1], A [2], A [3],..... A [*n*].

Step 1: Find the smallest element in the array of *n* numbers A[1], A[2], ... A[*n*]. Let LOC is the location of the smallest number in the array. Then interchange A[LOC] and A[1] by swap = A[LOC]; A[LOC] = A[1]; A[1] = Swap.

Step 2: Find the second smallest number in the sub list of *n* – 1 elements A [2] A [3] ..... A [*n* – 1] (first element is already sorted). Now we concentrate on the rest of the elements in the array. Again A [LOC] is the smallest element in the remaining array and LOC the corresponding location then interchange A [LOC] and A [2].Now A [1] and A [2] is sorted, since A [1] less than or equal to A [2].

Step 3: Repeat the process by reducing one element each from the array *Step n* – 1: Find the *n* – 1 smallest number in the sub array of 2 elements (*i.e*., A(*n*–1), A (*n*)). Consider A [LOC] is the smallest element and LOC is its corresponding position.

Then interchange A [LOC] and A(*n* – 1). Now the array A [1], A [2], A [3], A [4],…….A [*n*] will be a sorted array.

3.2 The Differences between Selection Sorting and Insertion Sorting

Since the elements are added to the sorted sequence in order for selection sorting, 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.

Self-Assessment Exercise

Define selection sort?

Self-Assessment Answer

Please insert Answer to SAE

3.3 Straight Selection Sorting

The simplest of the selection sorts is called *straight selection*. Figure

1.0 shows 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).

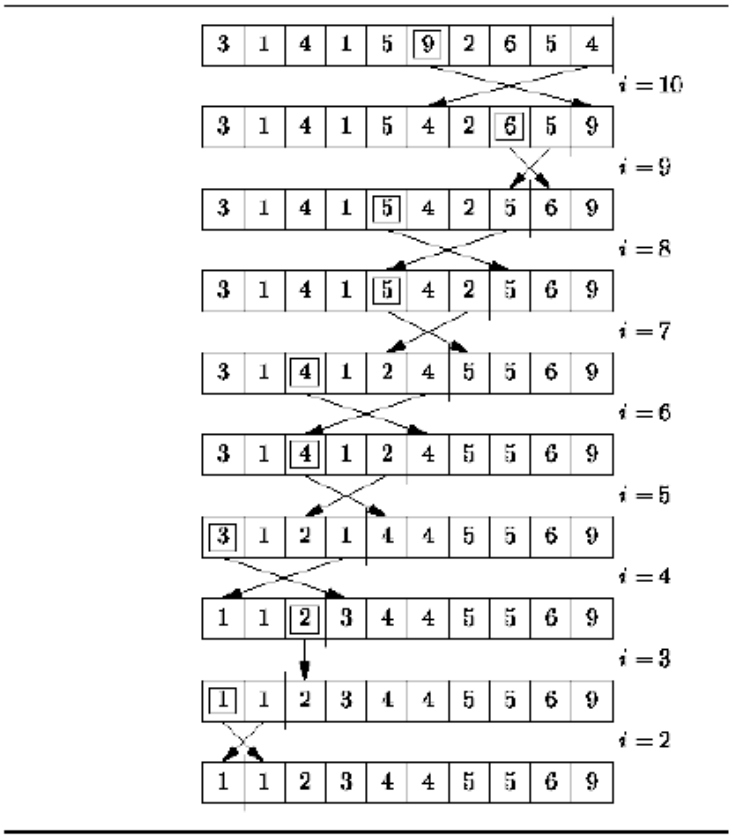


Figure 1.0: Straight selection sorting

At every step of the algorithm, a linear search of the unsorted elements is made in order to decide 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.

3.4 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).

```
public class StraightSelectionSorter
    extends AbstractSorter
{
    protected void sort ()
    {
        for (int i = n; i > 1; --i)
        {
            int max = 0;
            for (int j = 1; j < i; ++j)
                if (array [j].isGT (array [max]))
                    max = j;
            swap (i - 1, max);
        }
    }
}
```

Programme 1.0: StraightSelectionSorter class sort method

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.

Also, in the 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(s)

What is the name of the simplest of the selection sort?

Self-Assessment Answer (s) 2

Please insert Answer to SAE

4.0 Conclusion

In this unit you have been educated about selection sort and its implementation. Also, you have learned about straight selection sort. What you have learned in this unit borders on selection sort and its implementation.

5.0 Summary

You have learnt that:

(i) Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.
(ii) Since the elements are added to the sorted sequence in order for selection sorting, they are always added at one end.
(iii) The simplest of the selection sorts is called *straight selection*.

6.0 Tutor-Marked Assignment

(1) Explain the term straight selection sort.

(2) 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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Unit 4

Merge Sorting

Contents

1.0 Introduction

This unit will center principally on merge sorting. It gives an outline of steps to be adopted in sorting a sequence of elements. We will as well consider how to implement a Two Way Merge Sorter.

2.0 Learning Outcomes

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

(i) Explain merge sorting
(ii) Delineate the steps to be taken in sorting a sequence of n > 1 elements
(iii) Show how to implement a two-way merge sorting
(iv) Describe the merge method of a Two Way Merge Sorter class.

3.0 Learning Content

3.1 Merge Sorting

Merging is the process of combining two or more sorted array into a third sorted array. It was one of the first sorting algorithms used on a computer and was developed by John Von Neumann. Divide the array into approximately *n*/2 sub-arrays of size two and set the element in each sub array. Merging each sub-array with the adjacent sub-array will get another sorted sub-array of size four. Repeat this process until there is only one array remaining of size *n*.

Since at any time the two arrays being merged are both sub-arrays of A, lower and upper bounds are required to indicate the sub-arrays of a being merged. l1 and *u*1 represents the lower and upper bands of the first sub-array and l2 and *u*2 represents the lower and upper bands of the second sub-array respectively.

Let A be an array of *n* number of elements to be sorted A[1], A[2] ..... A[*n*].

Step 1: Divide the array A into approximately *n*/2 sorted sub-array of size 2. *i.e*., the elements in the (A [1], A [2]), (A [3], A [4]), (A [*k*], A [*k* + 1]), (A [*n* – 1], A [*n*]) sub-arrays are in sorted order.

Step 2: Merge each pair of pairs to obtain the following list of sorted sub-array of size 4; the elements in the sub-array are also in the sorted order. (A [1], A [2], A [3], A [4]),..... (A [*k* – 1], A [*k*], A [*k* + 1], A [*k* + 2]),..... (A [*n* – 3], A [*n* – 2], A [*n* – 1], A [*n*].

Step 3: Repeat the step 2 recursively until there is only one sorted array of size *n*.

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 2*n* in *O*(2*n*)=*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.

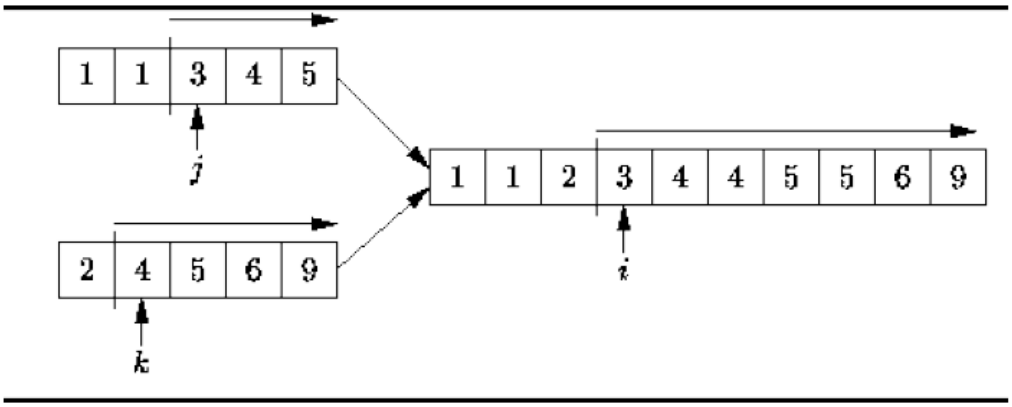


Figure 1.0: Two-way merging

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 element:

3. Divide the sequence into two sequences of length [n/2] and [n/2];
4. Recursively sort each of the two subsequences; and then,
5. Merge the sorted subsequences to obtain the final result.

Figure 1.1 illustrates the operation of the two-way merge sort algorithm.

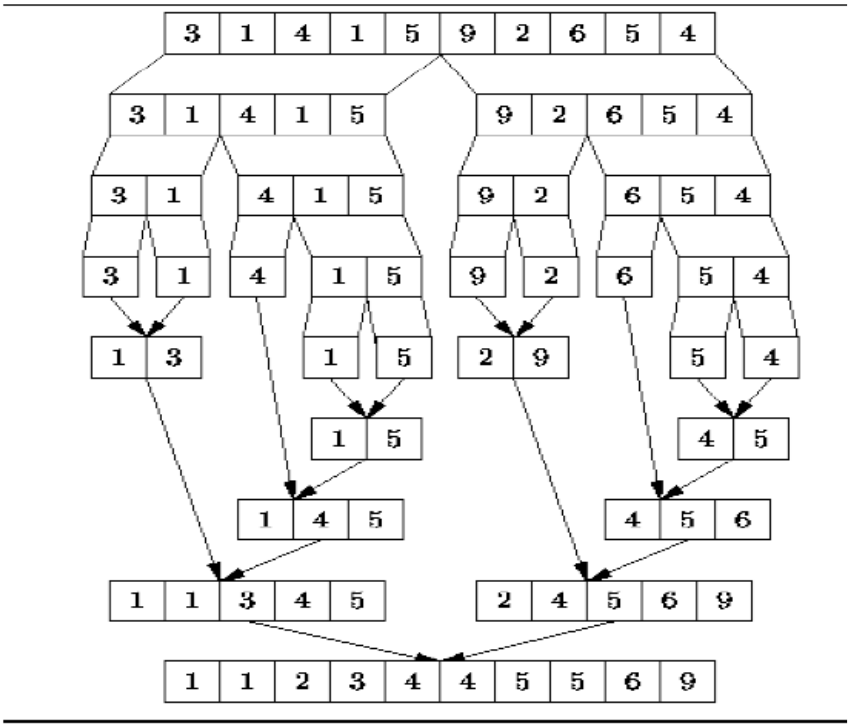


Figure 1.1: Two-way merge sorting

Self-Assessment Exercise(s) 1

What is merging?

Self-Assessment Answer(s) 1

Please insert answer(s) to SAE.

3.2 Implementation

Programme 1.0 declares the Two Way Merge Sorter class. The Two Way Merge Sorter 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.

```
public class TwoWayMergeSorter
    extends AbstractSorter
{
    Comparable[] tempArray;

    // ...
}
```

Program 1.0: Two Way Merge Sorter fields.

3.3 Merging

The merge method of the Two Way Merge Sorter class is defined in Programme 1.1. Altogether, this method takes three integer parameters, left, middle, and right. It is assumed that

$$\texttt{left} \leq \texttt{middle} < \texttt{right}.$$

Furthermore, it is assumed that the two subsequences of the array,

$$\texttt{array[left]}, \texttt{array[left} + 1], \ldots, \texttt{array[middle]},$$

and

$$\texttt{array[middle} + 1], \texttt{array[middle} + 2], \ldots, \texttt{array[right]},$$

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

$$\texttt{array[left]}, \texttt{array[left} + 1], \ldots, \texttt{array[right]}.$$

```
public class TwoWayMergeSorter
    extends AbstractSorter
{
    Comparable[] tempArray;

    protected void merge (int left, int middle, int right)
    {
        int i = left;
        int j = left;
        int k = middle + 1;
        while (j <= middle && k <= right)
        {
            if (array [j].isLT (array [k]))
                tempArray [i++] = array [j++];
            else
                tempArray [i++] = array [k++];
        }
        while (j <= middle)
            tempArray [i++] = array [j++];
        for (i = left; i < k; ++i)
            array [i] = tempArray [i];
    }
    // ...
}
```

Programme 1.1: Two Way Merge Sorter class merge method

In order to determine the running time of the merge method, it is necessary to recognize that the total number of iterations of the two loops (lines 11-17, lines 18-19) is **right – 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 = right – left + 1** is the total number of elements in the two subsequences that are merged.

Self-Assessment Exercise(s)

In order to determine the running time of the merge method, it is necessary to recognize that the ……. number of iterations of the two loops is……. in the worst case.

Self-Assessment Answer(s)

Please insert answer(s) to SAE.

4.0 Conclusion

Specially, you learned about merge sorting. Also, you would have learned about steps to be adopted in sorting a sequence of elements. Finally, the implementation of Two Way Merge Sorter was also considered. What you have learned in this unit is tailored on merge sorting and its implementation.

5.0 Summary

You have learnt that:

(i) Merging is the process of combining two or more sorted array into a third sorted array.

(ii) In order to determine the running time of the merge method, it is necessary to recognize that the total number of iterations of the two loops (lines 11-17, lines 18-19) is **right – left + 1**, in the worst case. The total number of iterations of the third loop (lines 20-21) is the same.

6.0 Tutor-Marked Assignment

(1) What is merge sorting?

(2) Explain the steps involved in merge sorting?

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.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

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.nou.edu.ng/noun/NOUN_OCL/pdf/pdf2/CIT%20341%20MAIN.pdf

Module 4

Fundamental Issues in Language Design

Unit 1 General Principles of Language Design

Unit 2: Data Structures Models

Unit 3: Control Structure Models and Abstraction Mechanisms

Unit 1

General Principles of Language Design

Contents

1.0 Introduction

This unit discusses issues that borders on general principles of language design. The unit describes the issues like simple, clear and unified set of primitives; clear syntax; support for abstraction; ease of verification/ provability; portability; ease and efficiency of implementation and clear semantics. The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development and distribution of software. This unit discusses design goals for any desirable programming language as a buildup of the previous unit.

2.0 Learning Outcomes

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

(i) Explain simple, clear and unified set of primitive issues
(ii) Explain clear syntax issues in language design
(iii) Describe support for abstraction as issues in language design
(iv) Explain simple, object-oriented and familiar goal
(v) Describe architecture neutral and portable goal

3.0 Learning Content

3.1 Simple, Clear and Unified Set of Primitives

Programming language are how people talk to the computer and thus it is desirable that it is simple, clear and unified set of primitives for expressing algorithms and data structures that would avoid anything like ambiguity. The language should be easy to learn and to write programs.

Clear Syntax

The issue of clear syntax raises two requirements – free from ambiguity and greater readability for expressing algorithms and data structures. You cannot imagine an ambiguous language to be implemented in developing programs. Equally important, readability cannot be ignored. It is important because the bugs in software are explored after years and by then the creator may not be present to fix them. Thus the source code must be readable like a book.

BASIC, Algol and Pascal were intentionally designed to facilitate clarity of expression. BASIC had a very small instruction set and Pascal as well was explicitly designed as a teaching language with features that facilitated the use of structured programming principles.

Self-Assessment Exercise(s)

1. What are the two requirements of clear syntax?

Self-Assessment Answer(s)

Please insert answer(s) to SAE.

3.2 Support for Abstraction

Abstraction is a fundamental aspect of the program design process. Programmers spend a lot of time building abstractions, both data abstractions (such as array, record, stack) and procedural abstractions (such as procedures, functions, loops etc.), to exploit the reuse of code and avoid reinventing it. A good programming language supports data and procedural abstraction so well that it is a preferred design to in most applications.

Java for example includes class libraries that contain implementations of basic data structures like vectors and stacks.

Ease of Verification/ Provability

Ease of verification and the quality that you can easily prove the logic is another issue that is becoming more reasonable in designing programming language. You should be able to verify your programs and so the language should also provide support for verification – provability of programs. Not necessarily machine based provability but may be the hand-based provability.

Self-Assessment Exercise(s)

1. ……. is a fundamental aspect of the program design process.

Self-Assessment Answer (s)

Please insert Answer to SAE

3.3 Portability

The language should be oriented towards the end user and not towards architecture or Machine. Simply because a machine or the assembly has a certain feature, it does not necessarily make you include that feature on the language. The other machine might not have that. So the Machine independence means you should provide as far as possible an abstract form that is not based on machine architecture. The amount of change needed to move the program to another architecture should be minimum.

Ease and Efficiency of Implementation

You should be able to easily implement without compromising the portability, availability of ready algorithms and everything. This and the fact that it used very low level primitives is perhaps, the most important reason for the success of programming language like C. The programs written with the language should be efficient – they

should run fast. This is related to compile time efficiency – how fast your programs can compile.

Clear semantics

To be a language generally acceptable, it has a clear definition of what its constructs do. The common clear semantics of each of the constructs of the language, you can expect the wider acceptability.

Apart from above discussed issues some more as Run-time efficient, Ease of maintenance, Fast compilation/translation, Support for extensibility, Support for subset are some other issues worth considering.

Support for subset is controversial. Many talk about a language where it is possible to divide a language into small kernel and large set of extensions. However ADA has specified that there should be no support for subset for the reason it affects the portability.

Self-Assessment Exercise(s)

1. The language should be oriented towards the …….. and not towards architecture or machine.

Self-Assessment Answer (s)

Please insert Answer to SAE

3.4 Simple, Object Oriented, and Familiar

Primary characteristics of the any programming language should include a simple language that can be programmed without extensive programmer training while being attuned to current software practices. The fundamental concepts of such language should be grasped quickly and should make programmers to be productive from the very beginning of learning such language.

Robust and Secure

One of the design goals of a programming language is that it should be designed for creating highly reliable software. It should provide extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits.

Self-Assessment Exercise(s)

1. Primary ……. of the any programming language should include a …… language that can be programmed without extensive programmer training.

Self-Assessment Answer (s)

Please insert Answer to SAE

3.5 Architecture Neutral and Portable

Programming language should be designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces.

High Performance

Performance is always a consideration. The language platform should achieve superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the language platform. In general, users should perceive that interactive applications respond quickly even though they are interpreted.

Interpreted, Threaded, and Dynamic

In an interpreted platform such a language should be technology-based system; the link phase of a program should be simple, incremental, and lightweight. Users should benefit from much faster development cycles--prototyping, experimentation, and rapid development which are the normal case, versus the traditional heavyweight compile, link, and test cycles.

Self-Assessment Exercise(s)

How should be a programming language be designed?

Self-Assessment Answer (s)

Please insert Answer to SAE

4.0 Conclusion

The programming language design issues were considered in this unit. Also, you have learned how to explain each of the issues. What you have learned in this unit borders general principles of language design. Also, what you have learned in this unit

concerns design goals of programming language. The goals are to be considered in designing any software.

5.0 Summary

You have learnt that:

(i) Programming language are how people talk to the computer and thus it is desirable that it is simple, clear and unified set of primitives for expressing algorithms and data structures that would avoid anything like ambiguity.
(ii) Abstraction is a fundamental aspect of the program design process.
(iii) Ease of verification and the quality that you can easily prove the logic is another issue that is becoming more reasonable in designing programming language.
(iv) The language should be oriented towards the end user and not towards architecture or machine.
(v) Primary characteristics of the any programming language should include a simple
(i) language. Programming language should be designed to support applications that will be deployed into heterogeneous network environments.

6.0 Tutor-Marked Assignment

(1) Explain the two requirements of clear syntax?

(2) What is portability in language design?

7.0 References/Further Readings

Liang, Y. D (2004). Introduction to java programming: comprehensive version. 6th ed. Pearson education, Inc. Pearson Prentice Hall. Upper Saddle River, NJ 07458
Vinus V. D. (2008). Principles of data structures using C and C++. New age international (P) Limited, New Delhi, India
http://www.cofficer.com/programming-language/issues-in-language-design-2/
http://java.sun.com/docs/white/langenv/Intro.doc2.html
http://www.hit.ac.il/staff/leonidm/information-systems/ch62.html

Unit 2

Data Structures Models

Contents

1.0 Introduction

As a continuation of the last unit which was on general principles of language design. This unit discusses the data model. It also describes the roles of data model, gives three perspectives of data model. Finally, describes the types of data models.

2.0 Learning Outcomes

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

(i) Explain a data model
(ii) Describe the roles of data model
(iii) Explain the three perspectives of data model
(iv) Describe the types of data model

3.0 Learning Content

3.1 Data Model

A **data model** in software engineering is an abstract model that documents and organizes the business data for communication between team members and is used as a plan for developing applications, specifically how data are stored and accessed.

A data model explicitly determines the structure of data or *structured data*. Typical applications of data models include database models, design of information systems, and enabling exchange of data. Usually data models are specified in a data modeling language.

Communication and precision are the two key benefits that make a data model important to applications that use and exchange data. A data model is the medium which project team members from different backgrounds and with different levels of experience can communicate with one another. Precision means that the terms and rules on a data model can be interpreted only one way and are not ambiguous.

A data model can be sometimes referred to as a data structure, especially in the context of programming languages. Data models are often complemented by function models, especially in the context of enterprise models.

3.2 The Role of Data Models

The main aim of data models is to support the development of information systems by providing the definition and format of data. According to West and Fowler (1999) "if this is done consistently across systems then compatibility of data can be achieved. If the same data structures are used to store and access data then different applications can share data. The results of this are indicated above. However, systems and interfaces often cost more than they should, to build, operate, and maintain.
Data models are often complemented by function models, especially in the context of enterprise models.

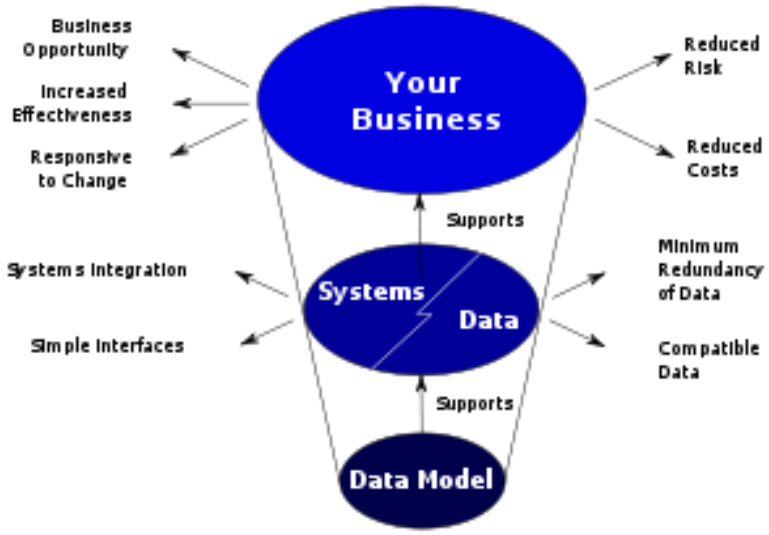


Figure 3.1: How data models deliver benefit.

Self-Assessment Exercise(s) 1

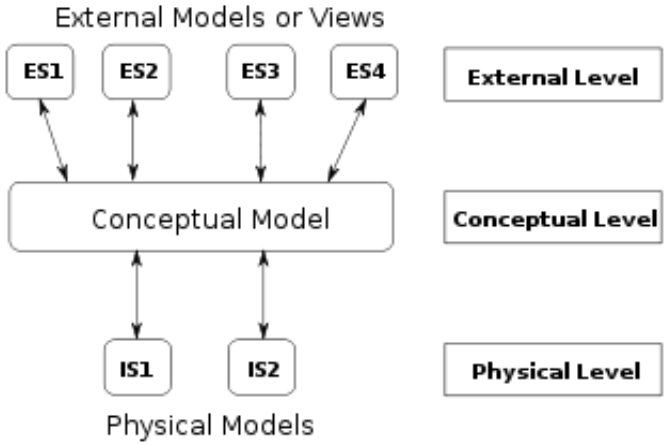
(i) What is a data model?

Self-Assessment Answer (s) 1

Please insert Answer to SAE

3.3 Three Perspectives

The ANSI/SPARC three level architecture. This shows that a data model can be an external model (or view), a conceptual model, or a physical model. This is not the only way to look at data models, but it is a useful way, particularly when comparing models.



A data model *instance* may be one of three kinds according to ANSI in 1975

(i) **Conceptual schema** : describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model. The use of conceptual schema has evolved to become a powerful communication tool with business users. Often called a subject area model (SAM) or high-level data model (HDM), this model is used to communicate core data concepts, rules, and definitions to a business user as part of an overall application development or enterprise initiative. The number of objects should be very small and focused on key concepts. Try to limit this model to one page, although for extremely large organizations or complex projects, the model might span two or more pages.

(ii) **Logical schema** : describes the semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.

(iii) **Physical schema** : describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

The significance of this approach, according to ANSI, is that it allows the three perspectives to be relatively independent of each other. Storage technology can change without affecting either the logical or the conceptual model. The table/column structure can change without (necessarily) affecting the conceptual model. In each case, of course, the structures must remain consistent with the other model. The table/column structure may be different from a direct translation of the entity classes and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure. Early phases of many software development projects emphasize the design of a conceptual data model. Such a design can be detailed into a logical data model. In later stages, this model may be translated into physical data model. However, it is also possible to implement a conceptual model directly.

Self-Assessment Exercise(s)

1. What is logical schema?
2. Describes the semantics, as represented by a particular data manipulation technology.

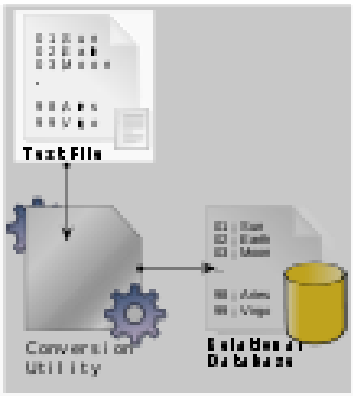
Self-Assessment Answer (s) 2

1. Please insert Answer to SAE

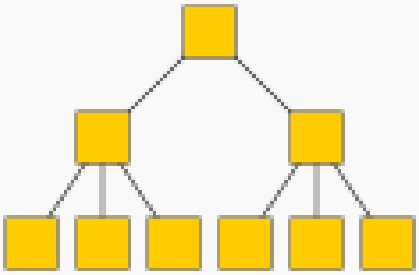
3.4 Types of Data Models

Database Model

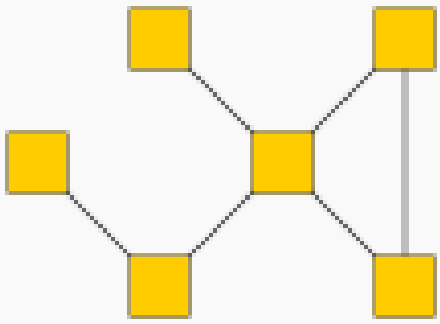
A database model is a theory or specification describing how a database is structured and used. Several such models have been suggested. Common models include:

- 

  Flat model

- 

  Hierarchical model

- 

  Network model

(i) **Flat model**: This may not strictly qualify as a data model. The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another.

(ii) **Hierarchical model**: In this model data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a sort field to keep the records in a particular order in each same-level list.

(iii) **Network model**: This model organizes data using two fundamental constructs, called records and sets. Records contain fields, and sets define one-to-many relationships between records: one owner, many members.

(iv) **Relational model**: is a database model based on first-order predicate logic. Its core idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values.

(v) **Object-relational model**: Similar to a relational database model, but objects, classes and inheritance are directly supported in database schemas and in the query language.

(vi) **Star schema**: is the simplest style of data warehouse schema. The star schema consists of a few "fact tables" (possibly only one, justifying the name) referencing any number of "dimension tables". The star schema is considered an important special case of the snowflake schema.

4.0 Conclusion

What you have learned in this unit concerns data structures models. You would have learned the roles of data models, the three perspectives of data model and the types of data model. These are useful for designing a good computer program.

5.0 Summary

You have learnt that:

(i) A data model in software engineering is an abstract model that documents and organizes the business data for communication between team members and is used as a plan for developing applications, specifically how data are stored and accessed.
(ii) The main aim of data models is to support the development of information systems by providing the definition and format of data.
(iii) The ANSI/SPARC three level architecture. This shows that a data model can be an external model (or view), a conceptual model, or a physical model.
(iv) A database model is a theory or specification describing how a database is structured and used. Several such models have been suggested.

6.0 Tutor-Marked Assignment

(1) What is a data model?

(2) Differentiate between flat model and network model?

7.0 References/Further Readings

Liang, Y. D (2004). Introduction to java programming: comprehensive version. 6th ed. Pearson Education, Inc. Pearson Prentice Hall. Upper Saddle River, NJ 07458.

Steve, H. (2009). Data modeling made simple. 2nd Edn. Technical publication. LLC2009.

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

http://www.cofficer.com/programming-language/issues-in-language-design-2/

http://java.sun.com/docs/white/langenv/Intro.doc2.html

http://www.hit.ac.il/staff/leonidm/information-systems/ch62.html

http://en.wikipedia.org/wiki/Data_model

Unit 3

Control Structures Models and Abstraction Mechanisms

Contents

1.0 Introduction

This unit discusses the basic principles that underlie structured program design and functional decomposition. The purpose of functional decomposition is to design structured programs that are easy to test, debug, and maintain. The basic idea is to break down (or decompose) a program into logically independent modules based on the processes or tasks they perform.

This unit is also on the data abstraction. It discusses the usefulness of data abstraction, relationship to object-oriented-paradigm as well as abstract variables and instance creation. Finally, it gives the advantages of abstract data typing.

2.0 Learning Outcomes

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

(i) Explain the control structures
(ii) Describe a good module design
(iii) Explain the different kinds of module design
(iv) Define abstraction and gives its usefulness
(v) Explain abstract variables

3.0 Learning Content

3.1 The Control Structure

A well-designed structured program consists of a set of independent, single function modules linked by a control structure that resembles a military chain of command or an organization chart (figure 1). Each module is represented by a rectangle. At the top of the control structure is a single module called the root (or the main control module). All control flows from the root which calls (or invokes) its level-2 child (or son) modules. The level-2 modules, in turn, call their level-3 children, and so on. The calling module (sometimes called the parent) passes data and/or control information to the child and receives data and/or control information back from the child; otherwise, the modules are viewed as independent black boxes. Note that control always returns to the calling module.

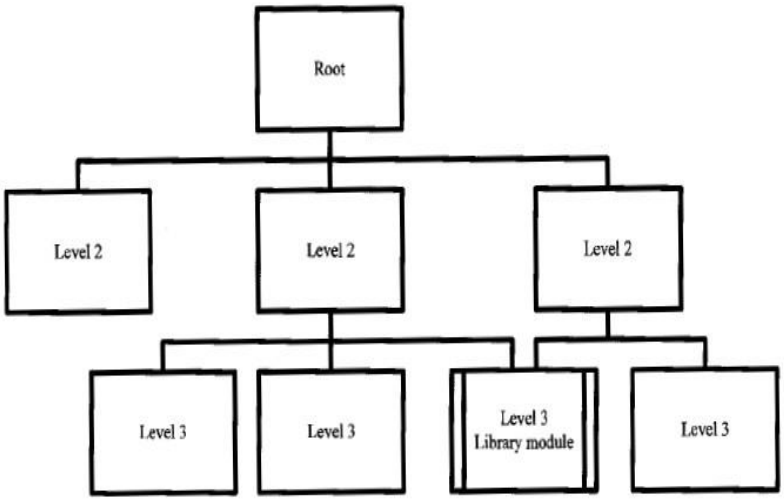


Figure 1: A well-designed structured program consists of a set of independent single function modules linked by a control structure.

A module with no children (a lowest-level module) is called a leaf and often implements a single algorithm. Library modules (e.g., a standard subroutine) are indicated by a rectangle marked with two vertical lines; see the leaf labeled *Library module* in figure 1. Note that a library module can be called by more than one parent.

The modules are often assigned identifying numbers or codes that indicate their relative positions in the hierarchy. For example, the root might be designated module 1.1, the level-2 modules might be designated 2.1, 2.2, 2.3, and so on. Other designers use letters (or even Roman numerals) to designate levels; for example, module A.1 is the root, module B.3 is the third module at level 2, module C.6 is the sixth module at level 3, and so on. Sometimes, more complex numbering schemes are used to indicate a path through the hierarchy. The key is consistency.

Self-Assessment Exercise(s)

1. What is a lowest-level module?

Self-Assessment Answer (s)

1. Please insert Answer to SAE

3.2 Module Design

A good module is cohesive and loosely coupled.

Cohesion

Cohesion is a measure of a module's completeness. Every statement in the module should relate to the same function, and all of that function's logic should be in the same

module. When a module becomes large enough to decompose, each submodule should perform a cohesive subfunction.

The best form of cohesion is called functional cohesion. A functionally cohesive module performs a single logical function, receives and returns no surplus data, and performs only essential logical operations. Functional cohesion is the designer's objective. A module is not considered function-ally cohesive if it exhibits other forms of cohesion.

Coincidental cohesion is the weakest type. In a coincidentally cohesive module, there is little or no logical justification for grouping the operations; the instructions are related almost by chance. In a logically cohesive module, all the elements are related to the same logical function; for example, all input operations or all data verification operations might be grouped to form a module. The elements that form a temporally cohesive module are related by time; for example, a setup module might hold all operations that must be performed at setup time.

Procedural cohesion is an intermediate form of cohesion, halfway between coincidental cohesion and functional cohesion. All the elements in a procedurally cohesive module are associated with the same procedural unit, such as a loop or a decision structure. Communicational cohesion groups elements that operate on the same set of input or output data (more generally, on the same data structure). With sequential cohesion, the modules form a chain of transformations, with the output from one module serving as input to the next. The three types of cohesion described in this paragraph often result from viewing the program as a flowchart.

Coupling

Coupling is a measure of a module's independence. Perfect independence is impossible because each module must accept data from and return data to its calling routine. Because global data errors can have difficult-to-trace ripple effects, a module should never change the value of any global data element that is not explicitly passed to it. If that rule is enforced, the list of parameters becomes a measure of how tightly the module is linked to the rest of the program. Fewer parameters imply looser coupling.

With data coupling (or input-output coupling), only data move between the modules. Data coupling is necessary if the modules are to communicate. Control coupling involves passing control information (e.g., a switch setting) between the modules. Hybrid coupling is a combination of data coupling and control coupling. For example, if module A modifies an instruction in module B, the operation looks like data coupling to module A and control coupling to module B. Whenever possible, control and hybrid coupling should be eliminated.

With common-environment coupling, two or more modules interact with a common data environment, such as a shared communication region or a shared file. With content coupling, some or all of the contents of one module are included in the other. This problem often occurs when a module is given multiple entry points. Both common-

environment and content coupling can lead to severe ripple effects, and should be avoided.

Binding time, the time at which a module's values and identifiers are fixed, is another factor that influences coupling. A module can be fixed (rendered unchangeable) at coding time, at compilation time, at load time, or at execution time. Generally, the later the binding time the better the module.

Self-Assessment Exercise(s)

1. What is a cohesion?

Self-Assessment Answer(s)

1. Please insert Answer to SAE

3.3 Sequence, Selection, and Repetition

The modules that form a well-structured program are composed of three basic logical building blocks or constructs: sequence, selection (or decision), and repetition (or iteration). Go to or branch instructions are not permitted.

Sequence (figure 2) implies that the logic is executed in simple sequence, one block after another. Note that each block might represent one or more actual instructions.

Selection (or decision) logic provides alternate paths through the block depending on a run-time condition. With IF-THEN-ELSE logic (Figure 3), if the condition is true the logic associated with the THEN branch is executed and the ELSE block is skipped. If the condition is false the ELSE logic is executed and the THEN logic is skipped. A case structure (figure 4) provides more than two logical paths through the block of logic based (usually) on the value of a control variable.

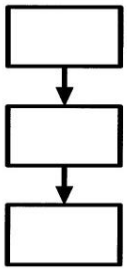


Figure 2. Sequence.

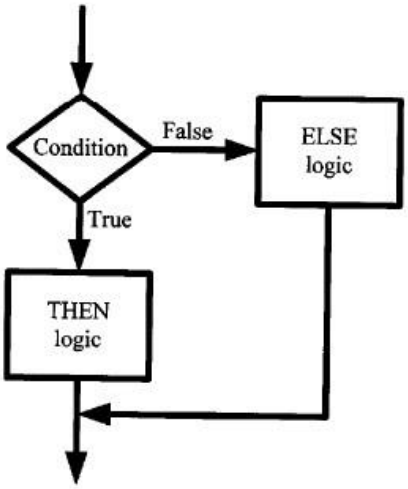


Figure 3: Selection.

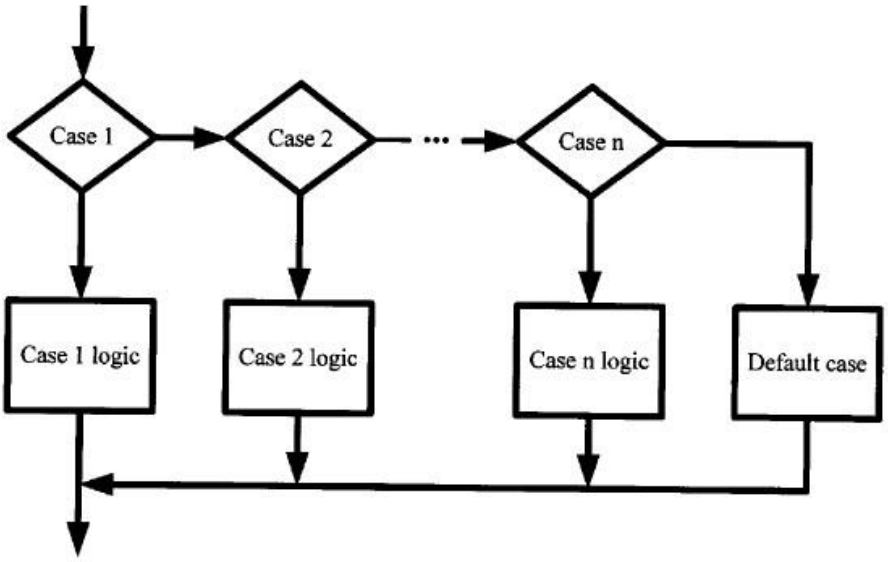


Figure 4: A case structure.

There are two basic patterns for showing repetitive logic: DO WHILE and DO UNTIL (Figure 5). In a DO WHILE block, the test is performed first and the associated instructions are performed only if (while) the test condition is true. In a DO UNTIL block, the associated instructions are executed first and then the exit condition is tested.

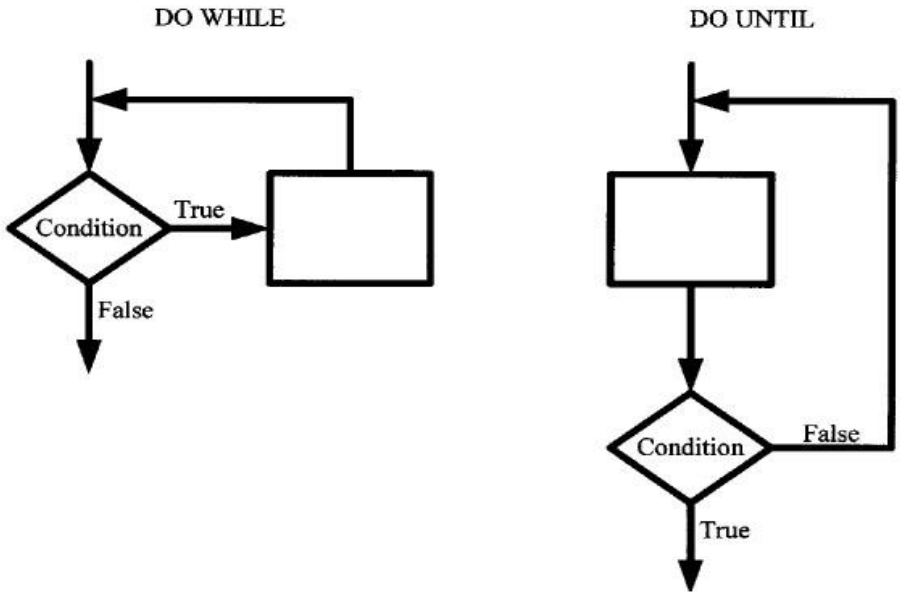


Figure 5: Repetition.

Self-Assessment Exercise(s)

1. What is a sequence?

Self-Assessment Answer (s)

1. Please insert Answer to SAE

3.4 Abstraction

To abstract is to ignore some details of a thing in favor of others. Abstraction is important in problem solving because it allows problem solvers to focus on essential details while ignoring the inessential, thus simplifying the problem and bringing to attention those aspects of the problem involved in its solution. Abstract data types are important in computer science because they provide a clear and precise way to specify what data a program must manipulate, and how the program must manipulate its data, without regard to details about how data are represented or how operations are implemented. Once an abstract data type is understood and documented, it serves as a specification that programmers can use to guide their choice of data representation and operation implementation, and as a standard for ensuring program correctness.

A realization of an abstract data type that provides representations of the values of its carrier set and algorithms for its operations is called a data type. Programming languages typically provide several built-in data types, and usually also facilities for programmers to create others. Most programming languages provide a data type realizing the Integer abstract data type, for example. The carrier set of the Integer abstract data type is a collection of whole numbers, so these numbers must be

represented in some way. Programs typically use a string of bits of fixed size (often 32 bits) to represent Integer values in base two, with one bit used to represent the sign of the number. Algorithms that manipulate these strings of bits implement the operations of the abstract data type.

Realizations of abstract data types are rarely perfect. Representations are always finite, while carrier sets of abstract data types are often infinite. Many individual values of some carrier sets (such as real numbers) cannot be precisely represented on digital computers. Nevertheless, abstract data types provide the standard against which the data types realized in programs are judged.

Usefulness

Such specifications of abstract data types provide the basis for their realization in programs. Programmers know which data values need to be represented, which operations need to be implemented, and which constraints must be satisfied. Careful study of program code and the appropriate selection of tests help to ensure that the programs are correct. Finally, specifications of abstract data types can be used to investigate and demonstrate the properties of abstract data types themselves, leading to better understanding of programs and ultimately higher-quality software.

Self-Assessment Exercise(s)

1. What does it mean to abstract?

Self-Assessment Answer (s)

1. Please insert Answer to SAE

3.5 Relation to Object-Oriented Paradigm

A major trend in computer science is the object-oriented paradigm, an approach to program design and implementation using collections of interacting entities called objects. Objects incorporate both data and operations. In this way they mimic things in the real world, which have properties (data) and behaviors (operations). Objects that hold the same kind of data and perform the same operations form a class.

Abstract data values are separated from abstract data type operations. If the values in the carrier set of an abstract data type can be reconceptualized to include not only data values but also abstract data type operations, then the elements of the carrier set become entities that incorporate both data and operations, like objects, and the carrier set itself is very much like a class. The object-oriented paradigm can thus be seen as an outgrowth of the use of abstract data types.

Defining an abstract data type (ADT)

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no

standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Self-Assessment Exercise(s)

1. A major trend in computer science is the …… an approach to program design and implementation using collections of interacting entities called …….

Self-Assessment Answer (s)

1. Please insert Answer to SAE

3.6 Abstract Variable

Imperative ADT definitions often depend on the concept of an *abstract variable*, which may be regarded as the simplest non-trivial ADT. An abstract variable *V* is a mutable entity that admits two operations:

- store(*V*,*x*) where *x* is a *value* of unspecified nature; and
- fetch(*V*), that yields a value;

with the constraint that

- fetch(*V*) always returns the value *x* used in the most recent store(*V*,*x*) operation on the same variable *V*.

As in so many programming languages, the operation store(*V*,*x*) is often written $V \leftarrow x$ (or some similar notation), and fetch(*V*) is implied whenever a variable *V* is used in a context where a value is required. Thus, for example, $V \leftarrow V + 1$ is commonly understood to be a shorthand for store(*V*,fetch(*V*) + 1).
In this definition, it is implicitly assumed that storing a value into a variable *U* has no effect on the state of a distinct variable *V*. To make this assumption explicit, one could add the constraint that

- if *U* and *V* are distinct variables, the sequence { store(*U*,*x*); store(*V*,*y*) } is equivalent to { store(*V*,*y*); store(*U*,*x*) }.

More generally, ADT definitions often assume that any operation that changes the state of one ADT instance has no effect on the state of any other instance (including other instances of the same ADT) — unless the ADT axioms imply that the two instances are connected (aliased) in that sense. For example, when extending the definition of abstract variable to include abstract records, the operation that selects a field from a record variable *R* must yield a variable *V* that is aliased to that part of *R*.
The definition of an abstract variable *V* may also restrict the stored values *x* to members of a specific set *X*, called the *range* or *type* of *V*. As in programming languages, such restrictions may simplify the description and analysis of algorithms, and improve their readability.

Note that this definition does not imply anything about the result of evaluating fetch(*V*) when *V* is *un-initialized*, that is, before performing any store operation on *V*. An algorithm that does so is usually considered invalid, because its effect is not defined. (However, there are some important algorithms whose efficiency strongly depends on the assumption that such a fetch is legal, and returns some arbitrary value in the variable's range.

Instance creation

Some algorithms need to create new instances of some ADT (such as new variables, or new stacks). To describe such algorithms, one usually includes in the ADT definition a create() operation that yields an instance of the ADT, usually with axioms equivalent to

- the result of create() is distinct from any instance *S* in use by the algorithm.

This axiom may be strengthened to exclude also partial aliasing with other instances. On the other hand, this axiom still allows implementations of create () to yield a previously created instance that has become inaccessible to the program.

Self-Assessment Exercise(s)

1. Imperative ADT definitions often depend on the concept of an……, which may be regarded as the simplest non-trivial ADT.

Self-Assessment Answer (s)

1. Please insert Answer to SAE

3.7 Preconditions, Postconditions, and Invariants

In imperative-style definitions, the axioms are often expressed by *preconditions*, that specify when an operation may be executed; *postconditions*, that relate the states of the ADT before and after the execution of each operation; and *invariants*, that specify properties of the ADT that are *not* changed by the operations.

Advantages of abstract data typing

- **Encapsulation:** Abstraction provides a promise that any implementation of the ADT has certain properties and abilities; knowing these is all that is required to make use of an ADT object. The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.
- **Localization of change:** Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed. Since any changes to the implementation must still comply with the interface, and since code using an ADT may only refer to properties and abilities specified in the interface, changes may

be made to the implementation without requiring any changes in code where the ADT is used.

- **Flexibility:** Different implementations of an ADT, having all the same properties and abilities, are equivalent and may be used somewhat interchangeably in code that uses the ADT. This gives a great deal of flexibility when using ADT objects in different situations. For example, different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

Self-Assessment Exercise(s)

1. In imperative-style definitions, the axioms are often expressed by……., that specify when an operation may be executed.

Self-Assessment Answer(s)

Please insert Answer to SAE

4.0 Conclusion

What you have learned in this unit borders control structures. These structures are used in developing software. These are sequence, selection and repetition, these are always used in software development, irrespective of any programming language. What you have learned in this unit also borders data abstraction mechanisms. These mechanisms are used in program development.

5.0 Summary

You have learnt that:

1. A well-designed structured program consists of a set of independent, single function modules linked by a control structure that resembles a military chain of command or an organization chart.
2. Cohesion is a measure of a module's completeness. Every statement in the module should relate to the same function, and all of that function's logic should be in the same module.
3. Coupling is a measure of a module's independence. Perfect independence is impossible because each module must accept data from and return data to its calling routine.
4. The modules that form a well-structured program are composed of three basic logical building blocks or constructs: sequence, selection (or decision), and repetition (or iteration).

6.0 Tutor-Marked Assignment

(1) Explain the two repetition control structures?

(2) What is coupling in data control structures?

7.0 References/Further Readings

Liang, Y. D (2004). Introduction to java programming: comprehensive version. 6th ed. Pearson Education, Inc. Pearson Prentice Hall. Upper Saddle River, NJ 07458

Vinus V. D. (2008). Principles of data structures using C and C++. New Age International (P) Limited, New Delhi, India

http://www.cofficer.com/programming-language/issues-in-language-design-2/

http://java.sun.com/docs/white/langenv/Intro.doc2.html

http://www.hit.ac.il/staff/leonidm/information-systems/ch62.html

ANSWERS TO SELF ASSESSMENT QUESTIONS

MODULE 1

UNIT 1

SELF ASSESSMENT QUESTION 1

Differentiate between conceptual model and user interface documentations?

Answer

Conceptual model

Conceptual model is the result of object-oriented analysis; it captures concepts in the problem. The conceptual model is explicitly chosen to be independent of

implementation details, such as concurrency or data storage.

User interface documentations

User interface documentations (if applicable) is a document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

SELF ASSESSMENT QUESTION 2

Explain the terms information hiding and interface in object oriented concepts?

Information hiding

Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*.

Interface

Interface: The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them.

SELF ASSESSMENT QUESTION 3

Why do you think a design pattern should be used if applicable in design concepts? Use **design patterns** (if applicable): A design pattern is not a finished design, it is

a description of a solution to a common problem, in a context. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

SELF ASSESSMENT QUESTION 4

Explain class diagram as an output of object-oriented design?

Class diagram: A class diagram is a type of static structure **UML** diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

UNIT 2

SELF ASSESSMENT QUESTION 1

What do you understand by a data type?

A **data type** in computer programming simply refers to a classification of various kinds of data that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.

A *data type* consists of:

(i) a domain (= a set of values)

(ii) a set of operations that may be applied to the values.

SELF ASSESSMENT QUESTION 2

With an example, explain primitive data type?

A new, primitive type is definable by enumerating the distinct values belonging to it. Such a type is called an *enumeration type*. Its definition has the form

TYPE T = (c1, c2, ... , cn)

T is the new type identifier, and the c_i are the new constant identifiers.

Examples

TYPE shape = (rectangle, square, ellipse, circle) TYPE color = (red, yellow, green)

TYPE sex = (male, female)

SELF ASSESSMENT QUESTION 3

Explain the type Boolean with examples?

The type BOOLEAN

The two values of the standard type BOOLEAN are denoted by the identifiers TRUE and FALSE. The Boolean operators are the logical conjunction, disjunction, and negation.

Examples 1: Examples of Boolean operators are OR, NOT, AND, etc

SELF ASSESSMENT QUESTION 4

What is an abstract data type?

An Abstract Data Type commonly known 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.

SELF ASSESSMENT QUESTION 5

Explain a data structure?

Definition of 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 collection”. A cautiously chosen data structure will permit the most efficient algorithm to be used. Thus, a well-designed data structure allows a range of critical operations to be performed using a few resources, both execution time and memory spaces as possible.

SELF ASSESSMENT QUESTION 6

Describe linear data structures?

Linear Data Structures

The data structures in which individual data elements are stored and accessed linearly in the computer memory are called linear data structures. In 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.

SELF ASSESSMENT QUESTION 7

Explain why is the structure of data in the computer is is very important?

A well-structured data that are stored in the computer, makes the accessibility of data easier and the software programme routines that make do with the data are made simpler; time and storage spaces are also reduced.

UNIT 3

SELF ASSESSMENT QUESTION 1

(i) An array is a collection of homogeneous data elements described by a single name.

(ii) In computer programming, variables normally only store a single value

SELF ASSESSMENT QUESTION 2

What is a list?

A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list.

SELF ASSESSMENT QUESTION 3

What is a file?

A file is typically a large list that is stored in the external memory (e.g., a magnetic disk) of a computer.

SELF ASSESSMENT QUESTION 4

Strings are stored or represented in memory by using following three types of structures :

• Fixed length structures

• Variable length structures with fixed maximum

• Linear structures

UNIT 4

SELF ASSESSMENT QUESTION 1

What is a linked list?

A linked list is a linear collection of specially designed data elements, called nodes, linked tone another by means of pointers.

SELF ASSESSMENT QUESTION 2

Why are linked list referred to as dynamic data structure?

Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.

SELF ASSESSMENT QUESTION 3

What is creation operation on linked list?

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

SELF ASSESSMENT QUESTION 4

List the three types of linked list?

Types of Linked List

Basically we can divide the linked list into the following three types in the order in which they (or node) are arranged.

1. Singly linked list

2. Doubly linked list

3. Circular linked list

MODULE 2

UNIT 1

SELF ASSESSMENT QUESTION 1

What is insertion operation in stack operations?

The insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop.

SELF ASSESSMENT QUESTION 2

(i) IsEmpty reports whether the stack is empty

(ii) IsFull reports whether the stack is full

SELF ASSESSMENT QUESTION 3

List the two ways that stack can be implemented?

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (using pointers)

SELF ASSESSMENT QUESTION 4

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:

(a) Display “The stack is in overflow condition”

(b) Exit

2. TOP = TOP + 1

3. STACK [TOP] = ITEM

4. Exit

SELF ASSESSMENT QUESTION 5

If we want to implement a recursive function non-recursively, stack is programmed explicitly.

UNIT 2

SELF ASSESSMENT QUESTION 1

Define a queue?

A queue is logically a first in first out (FIFO or first come first serve) linear data structure.

SELF ASSESSMENT QUESTION 2

Storing a queue in a static data structure is an implementation that stores the queue in an array.

SELF ASSESSMENT QUESTION 3

What are the requirements for storing a queue in a dynamic data structure?

Storing a Queue in a Dynamic Data Structure: A queue requires a reference to the head node AND a reference to the tail node.

SELF ASSESSMENT QUESTION 4

List any one applications of queue?

Applications of Queue

1. Round robin techniques for processor scheduling is implemented using queue.

2. Printer server routines (in drivers) are designed using queues.

3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.

UNIT 3

SELF ASSESSMENT QUESTION 1

What is hashing?

Hashing is a technique where we can compute the location of the desired record in order to retrieve it in a single access (or comparison).

SELF ASSESSMENT QUESTION 2

What is the basic idea of hash function?

The basic idea of hash function is the transformation of the key into the corresponding location in the hash table.

SELF ASSESSMENT QUESTION 3

What is hash collision?

Hash Collision: It is possible that two non-identical keys K_1, K_2 are hashed into the same hash address. This situation is called Hash Collision.

SELF ASSESSMENT QUESTION 4

What is hash deletion?

Hash Deletion

A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node from a linked list.

SELF ASSESSMENT QUESTION 5

A tree is an ideal data structure for representing hierarchical data.

SELF ASSESSMENT QUESTION 6

What is the most popular and practical way of representing a binary tree?

The most popular and practical way of representing a binary tree is using linked list

(or pointers).

SELF ASSESSMENT QUESTION 7

What is tree traversal?

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are

SELF ASSESSMENT QUESTION 8

What is **isLeaf** in tree implementation?

isLeaf

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

UNIT 4

SELF ASSESSMENT QUESTION 1

What is a search tree?

A *search tree* is a tree which supports efficient search, insertion, and withdrawal operations.

SELF ASSESSMENT QUESTION 2

What is the difficulty with binary search tree?

The difficulty 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.

SELF ASSESSMENT QUESTION 3

How can you represent a graph?

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge '*e*' in E is identified with a unique pair (*a*, *b*) of nodes in V, denoted by *e* = [*a*, *b*].

SELF ASSESSMENT QUESTION 4

State the algorithm matrix transpose (G, GT)?

Algorithm Matrix Transpose (G, GT)

For i = 0 to i < V[G]

For j = 0 to j V[G]

GT (j, i) = G(i, j)

j = j + 1;

i = i + 1

MODULE 3

UNIT 1

SELF ASSESSMENT QUESTION 1

What is the purpose of sorting algorithm?

The purpose of the sorting algorithm is to reorganize the records so that their keys are ordered according to various well-defined ordering regulations.

SELF ASSESSMENT QUESTION 2

When is a sorting algorithm stable?

A sorting algorithm is called stable if it preserves the relative order of equal keys in the file.

SELF ASSESSMENT QUESTION 3

What is bubble sort?

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed.

UNIT 2

SELF ASSESSMENT QUESTION 1

What is insertion sort?

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file.

SELF ASSESSMENT QUESTION 2

Explain the best-case of insertion sort?

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*.

SELF ASSESSMENT QUESTION 3

State the running time for worst-case of insertion sort?

For Insertion sort we say the worst-case running time is $\theta(n^2)$, and the best-case running

time is θ(*n*).

UNIT 3

SELF ASSESSMENT QUESTION 1

Define selection sort?

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

SELF ASSESSMENT QUESTION 2

What is the name of the simplest of the selection sort?

The simplest of the selection sorts is called *straight selection*.

UNIT 4

SELF ASSESSMENT QUESTION 1

What is merging?

Merging is the process of combining two or more sorted array into a third sorted array.

SELF ASSESSMENT QUESTION 2

In order to determine the running time of the merge method, it is necessary to recognize that the total number of iterations of the two loops is **right – left + 1**, in the worst case.

MODULE 4

UNIT 1

SELF ASSESSMENT QUESTION 1

What are the two requirements of clear syntax?

The issue of clear syntax raises two requirements – free from ambiguity and greater readability for expressing algorithms and data structures.

SELF ASSESSMENT QUESTION 2

Abstraction is a fundamental aspect of the program design process.

SELF ASSESSMENT QUESTION 3

The language should be oriented towards the end user and not towards architecture or machine.

SELF ASSESSMENT QUESTION 4

Primary characteristics of the any programming language should include a simple language that can be programmed without extensive programmer training.

SELF ASSESSMENT QUESTION 5

How should be a programming language be designed?

Programming language should be designed to support applications that will be deployed into heterogeneous network environments.

UNIT 2

SELF ASSESSMENT QUESTION 1

What is a data model?

A data model in software engineering is an abstract model that documents and organizes the business data for communication between team members and is used as a plan for developing applications, specifically how data are stored and accessed.

SELF ASSESSMENT QUESTION 2

What is logical schema?

Describes the semantics, as represented by a particular data manipulation technology.

UNIT 3

SELF ASSESSMENT QUESTION 1

What is a lowest-level module?

A module with no children (a lowest-level module) is called a leaf and often implements a single algorithm.

SELF ASSESSMENT QUESTION 2

What is a cohesion?

Cohesion is a measure of a module's completeness. Every statement in the module should relate to the same function, and all of that function's logic should be in the same module.

SELF ASSESSMENT QUESTION 3

What is a sequence?

Sequence implies that the logic is executed in simple sequence, one block after another. Note that each block might represent one or more actual instructions.

SELF ASSESSMENT QUESTION 4

What does it mean to abstract?

To abstract is to ignore some details of a thing in favor of others.

SELF ASSESSMENT QUESTION 5

A major trend in computer science is the object-oriented paradigm, an approach to program design and implementation using collections of interacting entities called objects.

SELF ASSESSMENT QUESTION 6

Imperative ADT definitions often depend on the concept of an *abstract variable*, which may be regarded as the simplest non-trivial ADT.

SELF ASSESSMENT QUESTION 7

In imperative-style definitions, the axioms are often expressed by *preconditions*, that specify when an operation may be executed.