

BCA(N)-202 Data Structure & Program Methodology



**School of Computer Science & IT
Uttarakhand Open University,
Haridwari**

Contents

Block-1.....	8
Unit-I.....	8
Introduction to Data Structure	8
1.1 Learning Objectives.....	8
1.2 Introduction	8
1.3 What is Data Structure?	9
1.4 Methods of Interpreting bit setting	9
1.5 Types of Data Structure	12
Check Your Progress	16
1.6 Dynamic Memory Allocation	17
1.7 Abstract Data Types.....	19
Check Your Progress	19
1.8 Answer to Check Your Progress.....	20
1.9 Model Questions.....	21
UNIT II	21
Introduction to Algorithms	21
1.1 Learning Objectives	21
1.2 Introduction.....	22
1.3 Algorithm.....	22
1.4 Algorithmic Complexity	23
1.4.1 Space Complexity	24
1.4.2 Time Complexity	25
Check Your Progress	26
1.5 Asymptotic Notation.....	26
1.5.1 Big Omega Notation $\Omega(f)$	26
1.5.2 Big Oh Notation $o(f)$	27
1.5.3 Big Theta Notation.....	27
Check Your Progress	27
1.6 ANSWERS TO CHECK YOUR PROGRESS	28
1.7 Model Questions	29
UNIT III: LINEAR DATA STRUCTURES.....	30
1.1 Learning Objectives	30


1.2 Introduction.....	30
1.3 Linear Data Structure	31
Check Your Progress	31
1.4 Introduction to Stack.....	31
1.5 Introduction to Queue	35
Introduction:.....	35
Check Your Progress	44
1.6 Answer to Check Your Progress.....	45
1.7 Model Questions	45
UNIT IV: LINKED LIST	45
1.1 Learning Objectives.....	46
1.2 Introduction.....	46
1.3 Linked Lists	47
1.4 Inserting and Removing Nodes from a list	47
1.5 Linked Implemented of Stacks	49
1.6 Getnode and Freenode Operation	50
1.7 Linked Implemented of Queue	52
1.8 List Implementation of Priority Queue	53
1.9 Header Nodes.....	54
1.10 Circular Lists.....	54
1.11 Doubly linked list.....	56
Check Your Progress	57
1.12 Answer to Check Your Progress.....	57
1.13 Model Questions	57
Block-2.....	58
UNIT V: SORTING.....	58
1.1 Learning Objectives.....	58
1.2 Introduction.....	58
1.3 Sink Sort	59
1.4 Insertion Sort.....	60
1.5 Selection Sort.....	63
1.6 Bubble Sort	66
1.7 Merge Sort	71
1.8 Quick Sort.....	73

1.9 Radix Sort	79
Check Your Progress	80
1.10 Answer to Check Your Progress	81
1.11 Model Questions	81
UNIT VI: SEARCHING.....	82
Learning Objectives	82
1.2 Introduction.....	83
1.3 Searching	83
1.3.1 Linear Search	84
CHECK YOUR PROGRESS	87
1.3.2 Binary Search.....	87
1.4 Performance and Complexity	92
CHECK YOUR PROGRESS	93
1.5 Answer to Check Your Progress	94
1.6 Model Questions	95
UNIT VII: GRAPHS I: REPRESENTATION AND TRAVERSAL	95
1.1 Learning Objectives	96
1.2 Introduction.....	96
1.3 Graph	97
1.4 Terminologies of Graph.....	97
1.5 Different Types of Graph.....	101
1.6 Representation of Graph	104
1.6.1 Sequential Representation of Graph	104
1.6.2 Linked Representation of Graph	105
CHECK YOUR PROGRESS	106
1.7 Traversal in Graphs.....	106
1.7.1 Depth First Search	107
1.7.2 Breadth First Search.....	108
1.8 Königsberg Bridge Problem.....	110
CHECK YOUR PROGRESS	111
1.9 Answer to Check Your Progress.....	112
1.10 Model Questions.....	112
UNIT VIII: GRAPHS II: BASIC ALGORITHMS	113
1.1 Learning Objectives	113

1.2 Introduction.....	114
1.3 Minimum Spanning Tree	115
1.5 Single Source Shortest Path.....	117
Check Your Progress	120
1.6 Answer to Check Your Progress.....	120
1.7 Model Questions	120
Block-III	121
UNIT IX: BINARY TREES	121
1.1 Learning Objectives	121
1.2 Introduction.....	122
1.3 Definition of tree.....	122
1.4 Binary Tree	123
1.5 Binary Tree Representation	126
1.6 Tree Traversal Algorithms	129
1.6.1 Preorder Traversal.....	130
1.6.2 Inorder Traversal.....	130
1.6.3 Postorder Traversal	131
1.7 Prefix, Postfix and Infix Notations	135
CHECK YOUR PROGRESS	136
1.8 Answer to Check Your Progress.....	138
1.9 Model Questions	138
UNIT X: HEAP SORT.....	139
1.1 Learning Objectives	140
1.2 Introduction.....	140
1.3 Heap sort	140
1.4 Heap Representation	141
Check Your Progress	141
1.5 Heapification.....	141
1.6 Priority Queue.....	145
Check Your Progress	147
1.7 Answer to Check Your Progress.....	147
1.8 Model Questions	148
UNIT XI: SEARCH TREES.....	148
1.1 Learning Objectives	148

1.2 Introduction.....	149
1.3 Avl- Tree.....	149
1.4 Representation of AVL Tree.....	150
1.5 B-Tree.....	157
1.6 Multiway Search Trees	157
1.7 Operations on B - Tree.....	158
Check Your Progress	165
1.8 Answer to Check Your Progress.....	166
1.9 Model Questions	166
UNIT XII: TABLES	167
1.1 Learning Objectives.....	167
Introduction.....	168
1.3 Hashing Techniques.....	168
1.4 Why Hashing?.....	169
1.5 Methods of Dealing with Hash Clash	169
1.6 DOUBLE HASHING	173
Check Your Progress	174
1.7 Clustering.....	174
1.8 DYNAMIC AND EXTENDIBLE HASHING	176
Check Your progress	180
1.9 Answer to Check Your Progress.....	180
1.10 Model Questions	181
Block-IV	181
UNIT XIII: SETS.....	181
1.1 Learning Objectives.....	182
1.2 Introduction:.....	182
1.3 Bit Vector Representation	182
1.4 Linked list representation.....	183
Check Your Progress	186
1.5 Answer to Check Your Progress.....	186
1.6 Model Questions	186
UNIT XIV: STRING ALGORITHM	187
1.1 Learning Objective	188
1.2 Introduction.....	188

1.3 String Function	188
1.3.1 STRING LENGTH.....	189
1.3.2 STRING CONCATENATION	190
1.3.3 String Copy	190
1.4 Pattern Matching	191
Check Your Progress	192
1.5 Brute Force String Matching algorithm.	192
1.6 Knuth-Morris-Pratt(KMP) string matching algorithm.....	194
Check Your Progress	196
1.7 Answer to Check Your Progress	197
1.8 Model Questions.....	198
UNIT XV	199
PROGRAM DEVELOPMENT& PROGRAM TESTING AND VERIFICATION	199
1.1 Learning Objectives	199
1.2 Introduction.....	200
1.3 Life Cycle	200
1.4 Code Designing.....	201
1.5 Coding.....	202
1.6Programming Style	202
Check Your Progress	204
1.7 Testing Method.....	204
1.8 Verification Procedure.....	205
Check Your Progress	208
1.9 Answer to Check Your Progress.....	209
1.10 Model Questions	210
Referencing.....	210

Title	Data Structure
Authors	Dr. Pradip K Das and Dr. S.V. Rao
Adaption and Typesetting	Balam Singh Dafouti Academic Consultant- School of CS & IT, Uttarakhand Open University, Haldwani
ISBN:	
Acknowledgement	
<p>The University acknowledges with thanks to NPTEL and the author for providing the study material on NPTEL portal under Creative Commons Attribution-ShareAlike - CC BY-SA.</p> <p>The University also acknowledges with thanks to KKHSOU for providing the study material on http://eslm.kkhsou.ac.in/#E-SLM-for-Learner%2F1st%20Sem%2FMaster%20Degree%2FMBA%2FAccounting%20Managers portal a Creative Commons Attribution-Non Commercial-Share Alike 4.0</p> <p>License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/</p>	
 <p>Uttarakhand Open University, 2020</p> <p>This work by Uttarakhand Open University is licensed under a Creative Commons Attribution--Share Alike 3.0 United States License. It is attributed to the sources marked in the References, Article Sources and Contributors section.</p>	
Published By: Uttarakhand Open University	

Block-1

Unit-I Introduction to Data Structure

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 What is Data Structure
- 1.4 Methods of Interpreting bit setting
- 1.5 Types of Data Structure
- 1.6 Dynamic Memory Allocation
- 1.7 Abstract Data Types
- 1.8 Answer to check your progress
- 1.9 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will able to learn about:

- What is data structure?
- Methods of Interpreting bit setting
- Types of Data Structure

1.2 Introduction

It is important for every Computer Science student to understand the concept of *Information* and how it is organized or how it can be utilized.

If we arrange some data in an appropriate sequence, then it forms a Structure and gives us a meaning. This meaning is called *Information*. The basic unit of Information in Computer Science is a bit, Binary Digit.

So, we found two things in Information: One is *Data* and the other is *Structure*.

1.3 What is Data Structure?

1. A *data structure* is a systematic way of organizing and accessing data.
2. A *data structure* tries to structure data!
 - Usually more than one piece of data
 - Should define legal operations on the data
 - The data might be grouped together (e.g. in an linked list)
3. When we define a data structure we are in fact creating a new data type of our own.
 - I.e. using predefined types or previously user defined types.
Such new types are then used to reference variables type within a program

1.4 Methods of Interpreting bit setting

Why Data Structures?

1. Data structures study how data are stored in a computer so that operations can be implemented efficiently
2. Data structures are especially important when you have a large amount of information
3. Conceptual and concrete ways to organize data for efficient storage and manipulation.

Methods of Interpreting bit Setting

1. Binary Number System
 - Non Negative
 - Negative
 - Ones Complement Notation
 - Twos Complement Notation
2. Binary Coded Decimal
3. Real Number

4. Character String

Non-Negative Binary System

In this System each bit position represents a power of 2. The right most bit position represents 2^0 which equal 1. The next position to the left represents $2^1 = 2$ and so on. An Integer is represented as a sum of powers of 2. A string of all 0s represents the number 0. If a 1 appears in a particular bit position, the power of 2 represented by that bit position is included in the Sum. But if a 0 appears, the power of 2 is not included in the Sum. For example 10011, the sum is

Ones Complement Notation

Negative binary number is represented by ones Complement Notation. In this notation we represent a negative number by changing each bit in its absolute value to the opposite bit setting. For example, since 001001100 represent 38, 11011001 is used to represent -38. The left most number is reserved for the sign of the number. A bit String Starting with a 0 represents a positive number, where a bit string starting with a 1 represents a negative number.

Twos Complement Notation

In Twos Complement Notation is also used to represent a negative number. In this notation 1 is added to the Ones Complement Notation of a negative number. For example, since 11011001 represents -38 in Ones Complement Notation 11011010 used represent -38 in Twos Complement Notation

Binary Coded Decimal

In this System a string of bits may be used to represent integers in the Decimal Number System. Four bits can be used to represent a Decimal digit between 0 and 9 in the binary notation. A string of bits of arbitrary length may be divided into consecutive sets of four bits. With each set representing a decimal digit. The string then represents the number that is formed by those decimal digits in conventional decimal notation. For example, in this system the bit string 00110101 is separated into two strings of four bits each: 0011 and 0101. The first of these represents the decimal digit 3 and the second represents the decimal 5, so that the entire string represents the integer 35.

In the binary coded decimal system we use 4 bits, so these four bits represent sixteen possible states. But only 10 of those sixteen possibilities are used. That means, whose binary values are 10 or larger, are invalid in Binary Coded Decimal System.

Real Number

The Floating Point Notation use to represent Real Numbers. The key concept of floating-point notation is Mantissa, Base and Exponent. The base is usually fixed and the Mantissa and the Exponent vary to represent different Real Number. For Example, if the base is fixed at 10, the number -235.47 could be represented as $-23547 * 10^{-2}$. The Mantissa is 23547 and the exponent is -2. Other possible representations are $-.23547 * 10^3$ and $-235.47 * 10^0$

In the floating-point notation a real number is represented by a 32-bit string. Including in 32-bit, 24-bit use for representation of Mantissa and remaining 8-bit use for representation of Exponent .Both the mantissa and the exponent are twos complement binary Integers. For example, the 24-bit twos complement binary representation of -23547 is 11111111010010000000101, and the 8-bit twos complement binary representation of -2 is 11111110. So the representation of 235.47 is 111111110100100000001011111110. It can be used to represent extremely large or extremely small absolute values.

Character Strings

In computer science, information is not always interpreted numerically. Item such as names, address and job title must also be represented in some fashion with in computer. To enable the representation of such nonnumeric objects, still another method of interpreting bit strings is necessary. Such information is usually represented in character string form. For example, in some computers, the eight bits 11000000 is used to represent the character "A" and 11000001 for character "B" and another for each character that has a representation in a particular machine. So, the character string "AB" would be represented by the bit string 1100000011000001.

1.5 Types of Data Structure

The assignment of bit string to character may be entirely arbitrary, but it must be adhered to consistently. It may be that some convenient rule is used in assigning bit string to character. The number of bits varies computer wise used to represent a character.

Some computers are use 7-bit (therefore allow up to 128 possible characters), some computers are use 8-bits (up to 256 character), and some use 10-bits (up to 1024 possible characters). The number of bits necessary to represent a character in a particular computer is called the **byte size** and a group of bits that number is called a **byte**.

Array

In computer programming, a group of homogeneous elements of a specific data type is known as an array, one of the simplest data structures. Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a subscript. The index usually uses a consecutive range of integers, (as opposed to an associative array) but the index can have any ordinal set of values.

Some arrays are multi-dimensional, meaning they are indexed by a fixed number of integers, for example by a tuple of four integers. Generally, one- and two-dimensional arrays are the most common. Most programming languages have a built-in array data type.

Link List

In computer science, a linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential data type because it contains a link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.



Types of Link List

1. Linearly-linked List
 - Singly-linked list
 - Doubly-linked list
2. Circularly-linked list
 - Singly-circularly-linked list
 - Doubly-circularly-linked list
3. Sentinel nodes

Stack

A stack is a linear Structure in which item may be added or removed only at one end. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the end of the list, not in the middle. Two of the Data Structures that are useful in such situations are Stacks and queues. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the Top. This means, in particular, the elements are removed from a stack in the reverse order of that which they are inserted in to the stack. The stack also called "last-in first -out (LIFO)" list.

Special terminology is used for two basic operation associated with stack:

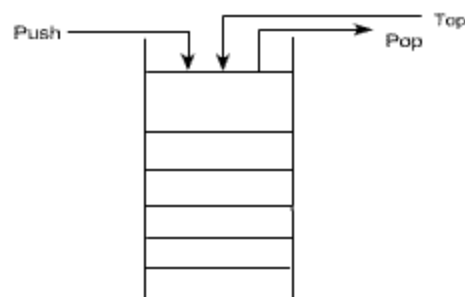


Fig: Stack

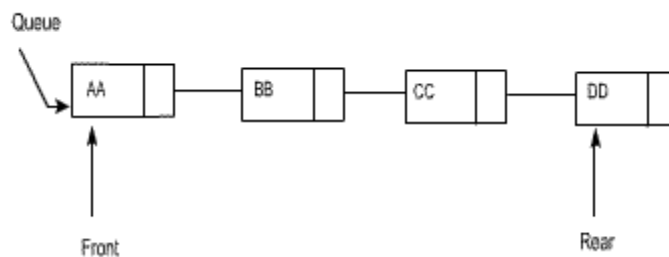
"Push" is the term used to insert an element into a stack.

"Pop" is the term used to delete an element from a stack.

Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the "front" and insertion can take place only at the other end, called "rear". The term "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called "first-in first-out" (FIFO) list. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is the order in which they leave. The real life example: the people waiting in a line at Railway ticket Counter form a queue, where the first person in a line is the first person to be waited on. An important example of a queue in computer science occurs in timesharing system, in which programs with the same priority form a queue while waiting to be executed.



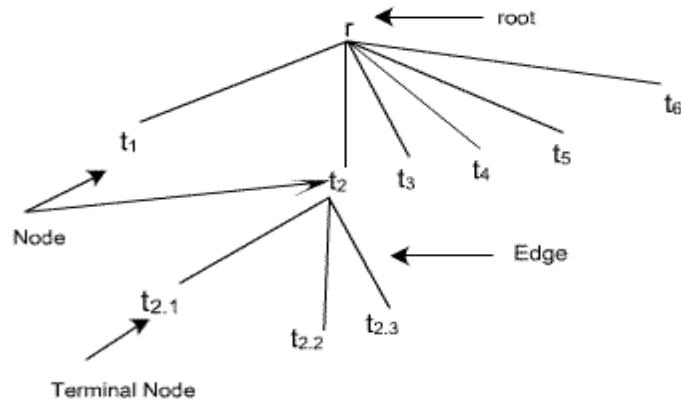
Tree

Data frequently contain a hierarchical relationship between various elements. This non-linear Data structure which reflects this relationship is called a rooted tree graph or, tree.

This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g. record, family tree and table of contents.

A tree consist of a distinguished node r , called the root and zero or more (sub) tree t_1, t_2, \dots, t_n , each of whose roots are connected by a directed edge to r .

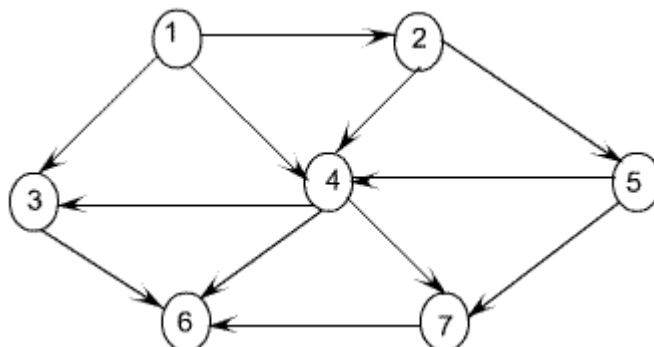
In the tree of figure, the root is A, Node t 2 has r as a parent and t 2.1, t 2.2 and t 2.3 as children. Each node may have arbitrary number of children, possibly zero. Nodes with no children are known as leaves.



Graph

A graph consists of a set of nodes (or Vertices) and a set of arc (or edge). Each arc in a graph is specified by a pair of nodes. A node n is incident to an arc x if n is one of the two nodes in the ordered pair of nodes that constitute x. The degree of a node is the number of arcs incident to it. The indegree of a node n is the number of arcs that have n as the head, and the outdegree of n is the number of arcs that have n as the tail.

The graph is the nonlinear data structure. The graph shown in the figure represents 7 vertices and 12 edges. The Vertices are { 1, 2, 3, 4, 5, 6, 7 } and the arcs are {(1,2), (1,3), (1,4), (2,4), (2,5), (3,4), (3,6), (4,5), (4,6), (4,7), (5,7), (6,7) }. Node (4) in figure has indegree 3, outdegree 3 and degree 6.



Abstract Data Type

Abstract Data Types (ADT's) are a model used to understand the design of a data structure

'Abstract ' implies that we give an implementation-independent view of the data structure

ADTs specify the type of data stored and the operations that support the data viewing a data structure as an ADT allows a programmer to focus on an idealized model of the data and its operations

Check Your Progress

Choose the correct one

1. Which of the following is non-linear data structure?
 - A) Stacks
 - B) List
 - C) Strings
 - D) Trees
2. Which of the following data structure is linear type?
 - A) Graph
 - B) Trees
 - C) Binary tree
 - D) Stack
3. A queue is a linear list of elements in which deletions can take place only at one end, called the
 - A. Front
 - B. Rear
 - C. Both A and B
 - D. None of the above
4. A stack is a linear Structure in which item may be added or removed only
 - A. One end
 - B. front end
 - C. Middle
 - D. None of the above

1.6 Dynamic Memory Allocation

The memory allocation process may be classified as static allocation and dynamic allocation. In static allocation, a fixed size of memory are reserved before loading and execution of a program. If that reserved memory is not sufficient or too large in amount then it may cause failure of the program or wastage of memory space. Therefore, C language provides a technique, in which a program can specify an array size at run time. The process of allocating memory at run time is known as dynamic memory allocation.

There are three dynamic memory allocation functions and one memory deallocation (releasing the memory) function. These are **malloc()**, **calloc()**, **realloc()** and **free()**.

malloc() : The function *malloc()* allocates a block of memory. The *malloc()* function reserves a block of memory of specified size and returns a pointer of type void. The reserved block is not initialize to zero. The syntax for using *malloc()* function is :

ptr = (cast-type *) malloc(byte-size);

where *ptr* is a pointer of type cast-type. The *malloc()* returns a pointer (of cast-type) to an area of memory with size byte-size.

Suppose *x* is a one dimensional integer array having 15 elements.

It is possible to define *x* as a pointer variable rather than an array. Thus, we write,

int *x;

instead of `int x[15];` or `#define size 15`

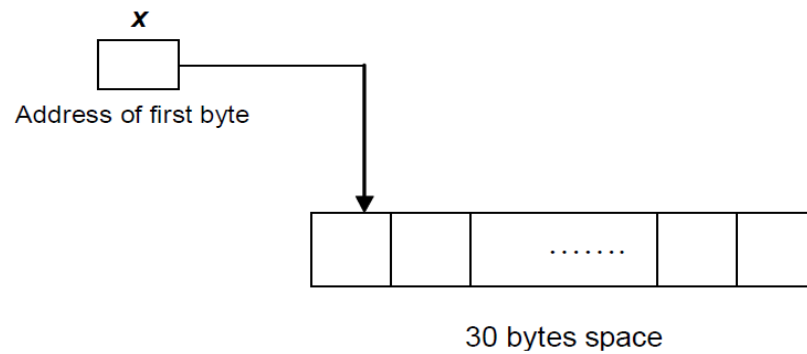
`int x[size];`

When *x* is declared as an array, a memory block having the capacity to store 15 elements will be reserved in advance. But in this case, when *x* is declared as a pointer variable, it will not be assigned a memory block automatically.

To assign sufficient memory for *x*, we can make use of the library function *malloc*, as follows :

x = (int *) malloc(15 * sizeof (int));

This function reserves a block of memory whose size (in bytes) is equivalent to 15 times the size of an integer. The address of the first byte of the reserved memory block is assigned to the pointer *x* of type *int*. Pictorial representation is shown below:



Representation of dynamic memory allocation

- **calloc()** : *calloc* is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. The main difference between the *calloc* and *malloc* function is that *malloc* function allocates a single block of storage space while the *calloc* function allocates amultiple blocks of storage space having the same size and intialize the allocated bytes to zero. The syntax for usinig *calloc()* function is :

$$ptr = (cast-type *) calloc (n, element-size)$$
 where *n* is the number of contiguous blocks to be allocate each of having the size *element-size*.
- **realloc()** : *realloc()* function is used to change the size of the previously allocated memory blocks. If the previously allocated memory is not sufficient or is much larger then by using the *realloc* function block size can be maximized or minimized. The syntax for usinig *realloc()* function is :

$$ptr = realloc (ptr, newsize);$$
 where *newsiz*e is the size of the memory space to be allocated.
- **free()** : It is necessary to free the memory allocated previously so that the memory can be reused. The *free()* function

deallocates memory that was previously allocated with `malloc()`, `calloc()` or `realloc()`. The syntax for using `free()` function is :

`free(ptr);`

where *ptr* is a pointer to a memory block, which has already been created by `malloc()` or `calloc()`.

1.7 Abstract Data Types

By now, you are well acquainted with data types, like integers, arrays, and so on. To access the data, you have used operations defined in the programming language for the data type. For example, array elements are accessed by using the square bracket notation; or scalar values are accessed simply by using the name of the corresponding variables.

This approach doesn't always work on large and complex programs in the real world. A modification to a program commonly requires a change in one or more of its data structures. It is the programmer's responsibility to create special kind of data types. The programmer needs to define everything related to the new data type such as:

- how the data values are stored,
- the possible operations that can be carried out with the custom data type
- new data type should be free from any confusion and must behave

like a built-in type. Such custom data types are called abstract data types.

Thus, an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc. It hides the detailed implementation of the data type and provides an interface to manipulate them.

Check Your Progress

Choose the correct one

5. In.....a fixed size of memory are reserved before loading and execution of a program.
- A. Static allocation.
 - B. Dynamic Allocation
 - C. Both A and B
 - D. None of the above
6. The functionallocates a block of memory
- A. malloc()
 - B. Calloc()
 - C. Relock()
 - D. One of the above
7. function is used to change the size of the previously allocated memory blocks
- A.Malloc()
 - B. realloc()
 - C. Both A and B
 - D. None of the above
8. is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures.
- A. calloc
 - B. ralloc
 - C. malloc
 - D. None of the above

1.8 Answer to Check Your Progress

- 1. D) Trees
- 2. D) Stack
- 3. A. Front
- 4. A. One end
- 5. A. Static allocation.
- 6. A. malloc()
- 7. B. realloc()
- 8. A. calloc

1.9 Model Questions

1. What is information? Explain with examples.
2. What is data? Explain with examples.
3. Name and describe the four basic data types in C.
4. What is a data structure? Why an array is called a data structure?
5. How does a structure differ from an array? How is a structure member accessed?
6. What is a pointer? How is a pointer initialized?
7. What do you mean by dynamic memory allocation? How is it useful?
8. What is the difference between the functions malloc() and calloc()?
9. What do you mean by abstract data type? Explain.

UNIT II

Introduction to Algorithms

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Algorithm
- 1.4 Complexity
 - 1.4.1 Space Complexity
 - 1.4.2 Time Complexity
- 1.5 Algorithmic Notation
 - 1.5.1 Big Theta Notation
 - 1.5.2 Big Oh Notation
 - 1.5.3 Big Omega Notation
- 1.6 Answers to Check Your Progress
- 1.7 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to:

- define and understand the term algorithm
- describe time and space complexity for algorithm
- define big theta asymptotic notation
- define big oh asymptotic notation
- define big omega asymptotic notation

1.2 Introduction

In this unit we will learn to define an algorithm and learn the basics of time and space complexity. In addition to these, different types of algorithmic notations will also be discussed. Moreover, we will be able to compare between different algorithms by using the concepts covered in this unit.

1.3 Algorithm

An algorithm is a step by step procedure for solving a problem. Algorithms can also be described as a sequence of computational steps that transform the input into output. There are five basic properties to be fulfilled by an algorithm. They are:

Input: An algorithm should have one or more inputs.

Output: An algorithm should have one or more outputs.

Finite: An algorithm should be a finite one in the sense that it must have a clear stopping point. It must be of fixed length.

Unambiguous: The algorithm must be unambiguous, which means that it should not contain any ambiguous or confusing part.

Easy to understand: The algorithm should be easy to understand so that it should be solvable even without using a computer or any computational tools.

Algorithms can be expressed with the help of both natural languages and programming languages. English is one of the common natural languages that is used to describe an algorithm. Let us now look at some examples to understand how to write an algorithm.

Example: Algorithm to calculate average and sum of three numbers

Step 1: Input A, B, C

Step 2: Compute Sum (S) as $A + B + C$

Step 3: Compute Average (AVG) as $S/3$

Step 4: Display Sum, Average

Example: Algorithm to find the smaller of two numbers

Step 1: Input A, B

Step 2: If A is less than B

Small = A

Else

Small = B

Step 3: Display Small

Example: Algorithm to find factorial of a number

Step 1: Input A

Step 2: Set Fact = 1, I = 1

Step 3: While $I \leq A$

$F = F \times I$

$I = I + 1$

End

Step 4: Display Fact

An algorithm is written with the objective to solve a problem. For example, let us consider the sorting problem. Suppose we have 15 numbers and we want to sort the numbers in ascending order. Sorting is a common problem and we may be presented with many different solutions to the same sorting problem. Each solution presented can be described using an algorithm. But now, if we have to pick an algorithm from the presented solutions then which solution do we choose and how do we compare the different algorithms to find the best algorithm for solving the problem? We compute the complexity of the different algorithms for the purpose of comparison.

1.4 Algorithmic Complexity

The measure of an algorithm is made based on its algorithmic complexity. Algorithmic complexity can be of two types – in terms of space and in terms of time. We use data structures to implement an algorithm.

Some of the data structure will require more space but less time while others may take less space and more time. We will need to find a trade-off between these two to find the most efficient algorithm. Let us look at the basics of space and time complexity now.

1.4.1 Space Complexity

Space complexity can be defined as the amount of space an algorithm needs from its start to its completion. The amount of space used by the algorithm from its start to its end is the space complexity of the algorithm. The memory space is measured in terms of bits and bytes. An algorithm with lesser memory space is preferred over the one with more memory space.

To understand what we mean by amount of space taken let us take a simple example. Suppose we need to swap the values of two numbers.

The following two programs show the swapping of two nos. in two different ways:

Example: Swap the values of two numbers Algorithm A: Swap two numbers using third variable

Step 1: Input A, B

Step 2: Set $C = A$

Step 3: Set $A = B$

Step 4: Set $B = C$

Step 5: Display A, B

Algorithm B: Swap two numbers without using third variable

Step 1: Input A, B

Step 2: Set $A = A + B$

Step 3: Set $B = A - B$

Step 4: Set $A = A - B$

Step 5: Display A, B

Here, we see that both the algorithms have the same number of steps (5). Still, in the first algorithm, we have used a third additional variable to swap the values between the first two numbers. The space required for

algorithm “A” will be more to store the third variable C. In the second algorithm “B”, we have swapped the values of the two numbers without using any third variable. Even though we had to perform additional computation we did not require any additional space in the second algorithm. The space required for the second algorithm “B” will be the space required to store only two variables. Here, the second algorithm “B” requires less space than algorithm “A”.

1.4.2 Time Complexity

Time complexity of an algorithm can be defined as the running time of the algorithm. Running time is measured not by computing the actual execution time needed by the algorithm. If this is the case, then the running time would be different for different machines. So, we want a measure that is machine independent, that is, it does not depend on the machine. The running time is measured based on the number of primitive or key operations performed by the algorithm. The number of operations does not change from one machine to another. The number of computational steps performed in the algorithm gives us the running time of that algorithm.

To understand what we mean by primitive or key operations let us take an example. Suppose we need to find the maximum of “N” numbers.

Following is an algorithm for this problem

Example: Algorithm to find maximum of N numbers

Step 1: Input N

Step 2: Read NUM

Step 3: Set MAX = NUM

Step 4: For I = 2 to N Do

Read NUM

If NUM > MAX then

MAX = NUM

Step 5: Display MAX

For the above algorithm, we need to find out the key operations first.

The comparison and exchange of elements are the key operations in the above algorithm. Once we know the key operation we need to count the number of times that operation has been executed. For the above case, the

comparison instruction is inside for loop which goes from 2 to N so the number of comparisons will be “N-1”.

Check Your Progress

- Q1. What are the properties of an algorithm?
- Q2. Define space and time complexity.
- Q3. Write an algorithm to find the product of three numbers.
- Q4. Write an algorithm to find the greatest of three numbers.

1.5 Asymptotic Notation

The time complexity of an algorithm is measured with the help of asymptotic notations. There are three basic asymptotic notations. They are as follows:

- 1) Big Omega Ω (f)
- 2) Big Oh $O(f)$
- 3) Big Theta θ (f)

The notations used to describe the asymptotic running time of an algorithm are defined in terms of function whose domains are the set of natural numbers. Let us look at the definitions of these three basic asymptotic notations.

1.5.1 Big Omega Notation Ω (f)

Big Omega notation provides an asymptotic lower bound for a function $f(n)$. A function $f(n)$ is said to be Ω ($g(n)$) if there exists positive values k and c , such that

$$f(n) \geq c * g(n), \text{ for all } n \geq k.$$

For a function $g(n)$, we denote by Ω ($g(n)$) the set of functions $(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } k \text{ such that}$

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq k$$

For all values n that are at or to the right of k , the value of function $f(n)$ is on or above $cg(n)$.

1.5.2 Big Oh Notation $O(f)$

Big Oh notation provides an asymptotic upper bound for a function $f(n)$. A function $f(n)$ is said to be $O(g(n))$ if there exist positive values k and c ,

such that

$$f(n) \leq c * g(n), \text{ for all } n \geq k.$$

For a function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } k \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq k$$

For all values n that are at or to the right of k , the value of function $f(n)$ is on or below $cg(n)$.

1.5.3 Big Theta Notation

Big Theta function provides both an asymptotic upper and lower bound for a function $f(n)$. A function $f(n)$ is said to be $\theta(g(n))$ if there exist positive values k , c_1 and c_2 , such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n), \text{ for all } n \geq k.$$

For a function $g(n)$, we denote by $\theta(g(n))$ the set of functions

$$\theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } k \text{ such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n), \text{ for all } n \geq k.$$

For all values n that are at or to the right of k , the value of function $f(n)$ lies at or above $c_1g(n)$ and at or below $c_2g(n)$.

Check Your Progress

Q5. Define omega notation.

Q6. Define big oh notation.

Q7. Define theta notation.

1.6 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: The properties of an algorithm are:

Input: An algorithm should have one or more inputs.

Output: An algorithm should have one or more outputs.

Finite: An algorithm must have a finite in the sense that it must have a clear

stopping point. It must be of fixed length.

Unambiguous: The algorithm must be unambiguous, which means that it should not contain any ambiguous part.

Easy to understand: The algorithm should be easy to understand such that it should be solvable even without using a computer or any computational tools.

Answer to Q2 : Space complexity can be defined as the amount of space an algorithm needs from its start to its completion. The amount of space used by the algorithm from its start to its end is the space complexity of the algorithm.

Answer to Q3: Algorithm for finding the product of three numbers:

Step1: Input A, B, C

Step 2: $\text{Prod} = A * B * C$

Step 3: Display Prod

Answer to Q4: Algorithm for finding the greatest of three numbers:

Step1: Input A, B, C

Step 2: IF $A > B$ do

IF $A > C$

Set Great = A

ELSE

Set Great = C

End IF

Else IF $B > C$ do

Set Great = B

ELSE

Set Great = C

End IF

Step 3: Display Great

Answer to Q5: For a function $g(n)$, we denote by $\Omega(g(n))$ the set of functions $\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } k \text{ such that}$

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq k$$

Answer to Q6: For a function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } k \text{ such that}$$

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq k$$

Answer to Q7: For a function $g(n)$, we denote by

$\theta(g(n))$ the set of functions

$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } k \text{ such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n),$
for all $n \geq k$.

1.7 Model Questions

Q1. Define algorithm?

Q2. What are the properties of an algorithm?

Q3. Write an algorithm to find the factors of a given number.

Q4. Write an algorithm to find whether a given number is prime or not.

Q5. Write an algorithm to display the Fibonacci series till its 10th element.

Q6. Define asymptotic notations.

Q7. Define Big Theta and Big Oh asymptotic notation.

Q8. Define omega notation.

Q9. How the asymptotic notations are used in evaluating algorithms?

UNIT III: LINEAR DATA STRUCTURES

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Linear Data Structure
- 1.4 Introduction to Stack
- 1.5 Introduction to Queue
- 1.6 Answer to Check Your Progress
- 1.7 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to learn about:

- Array
- Linked List
- Stacks
- Queues

1.2 Introduction

Data can be organized in many different ways. Therefore, you can create as many data structures as you want. However, data structures have been classified in several ways. Basically, data structures are of two types : linear data structure and nonlinear data structure.

Linear data structure: a data structure is said to be linear if the elements form a sequence i.e., while traversing sequentially, we can reach only one element directly from another. For example: Array, Linked list, Queue etc.

Nonlinear data structure: elements in a nonlinear data structure do not form a sequence i.e each item or element may be connected with two or more other items or elements in a non-linear arrangement. Moreover, removing one of the links could divide the data structure into two disjointed pieces. For example, Trees and Graphs etc. The following figure shows the linear and nonlinear data structures.

1.3 Linear Data Structure

A data structure is said to be linear if its elements form a sequence or a linear list.

Examples:

- Array
- Linked List
- Stacks
- Queues

Operations on linear Data Structures

Traversal: Visit every part of the data structure

Search: Traversal through the data structure for a given element

Insertion: Adding new elements to the data structure

Deletion: Removing an element from the data structure.

Sorting: Rearranging the elements in some type of order (e.g Increasing or Decreasing)

Merging: Combining two similar data structures into one

Check Your Progress

Q.1. State whether the following statements are true(T) or false(F):

- i) Stack follows a first-in-first-out technique.
- ii) Insertion of data into the stack is called the push operation.
- iii) Removal of element is termed as pop operation.

1.4 Introduction to Stack

1. Stack is basically a data object

2. The operational semantic (meaning) of stack is LIFO i.e. last in first out

Definition: It is an ordered list of elements n , such that $n > 0$ in which all insertions and deletions are made at one end called the top.

Primary operations defined on a stack:

1. PUSH: add an element at the top of the list.
2. POP: remove at the top of the list.

3. Also "IsEmpty()" and IsFull" function, which tests whether a stack is empty or full respectively.

Example:

1. Practical daily life: a pile of heavy books kept in a vertical box, dishes kept one on top of another

In computer world: In processing of subroutine calls and returns; there is an explicit use of stack of return addresses.

Also in evaluation of arithmetic expressions, stack is used.

Large number of stacks can be expressed using a single one dimensional stack only. Such an array is called a multiple stack array.

Push and Pop:

Algorithms

<i>Push (item,array , n, top)</i>	<i>Pop (item,array,top)</i>
<pre> Push (item,array , n, top) { If (n >= top) Then print "Stack is full" ; Else { top = top + 1; array[top] = item ; } } </pre>	<pre> Pop (item,array,top) { if (top <= 0) Then print "stack is empty". Else { item = array[top]; top = top - 1; } } </pre>

Arithmetic Expressions:

Arithmetic expressions are expressed as combinations of:

1. Operands
2. Operators (arithmetic, Boolean, relational operators)

Various rules have been formulated to specify the order of evaluation of combination of operators in any expression.

The arithmetic expressions are expressed in 3 different notations:

1. Infix:

- In this if the operator is binary; the operator is between the 2 operands.

And if the operator is unary, it precedes the operand.

2. Prefix:

- In this notation for the case of binary operators, the operator precedes both the operands.

3. Simple algorithm using stack can be used to evaluate the final answer.

Postfix:

- In this notation for the case of binary operators, the operator is after both the corresponding operands.
- Simple algorithm using stack can be used to evaluate the final answer.

Always remember that the order of appearance of operands does not change in any Notation. What changes is the position of operators working on those operands.

Rules Expressions:

RULES FOR EVALUATION OF ANY EXPRESSION:

An expression can be interpreted in many different ways if parentheses are not mentioned in the expression.

- For example the below given expression can be interpreted in many different ways:
- Hence we specify some basic rules for evaluation of any expression :

A priority table is specified for the various type of operators being used:

PRIORITY LEVEL	OPERATORS
6	** ; unary - ; unary +
5	* ; /
4	+ ; -
3	< ; > ; <= ; >= ; !> ; !< ; !=
2	Logical and operation
1	Logical or operation

Arithmetic Expressions:

Algorithm for evaluation of an expression E which is in prefix notation :

- We assume that the given prefix notation starts with IsEmpty ().

- If number of symbols = n in any infix expression then number of operations performed = some constant times n.
- Here next token function gives us the next occurring element in the expression in a left to right scan.
- The PUSH function adds element x to stack Q which is of maximum length n

Evaluate (E)

```

{
    Top = 0;
    While (1)
    {
        x = next token (E)
        If (x == infinity)
        {
            Print value of stack [top] as the
            output of the expression
        }
    }
}
Else
{
    If (x == operand)
        PUSH (Q, top, n, x);
    If (x == operator)
    {
        Pop correct number of operands according to
        the operator (unary/binary) and then perform the
        operation and store result onto the stack
    }
}
}

```

Multiple stacks:

- Here only one single one-dimensional array (Q) is used to store multiple stacks.
- B (i) denotes one position less than the position in Q for bottommost element of stacks i.
- T (i) denotes the top most element of stack i.
- m denotes the maximum size of the array being used.
- n denotes the number of stacks.

We also assume that equal segments of array will be used for each stack.

Initially let $B(i) = T(i) = [m/n] * (i-1)$ where $1 \leq i \leq n$.

Again we can have **push** or **pop** operations, which can be performed on each of these stacks.

Algorithms:

```

Push (i, x)
{
    if (((i < n) && (T(i) == B(i+1))) ||
        ((i >= n) && (T(i) == m)));

        Then call STACK_FULL;
    Else
    {
        T(i) = T(i) + 1;
        Q[T(i)] = x;
    }
}

```

```

Pop (i,x)
{
    if ( T(i) = B(i) )
        Then print that the stack is empty.
    Else
    {
        x = Q[T(i)];
        T[i] = T[i] - 1;
    }
}

```

ALGORITHM TO BE APPLIED WHEN $T [i] = B [i]$ CONDITION IS ENCOUNTERED WHILE DOING PUSH OPERATION.

```

{
    1. Find j such that  $i < j \leq n$  and there is a free space between stack j and stack (j +1).
    if such a j exist , then move stack i+1 , i+2 , .....till j one position to the right and hence create space
    for element between stack i and i +1.

    2. Else
        Find j such that  $1 \leq j < i$  and there is a free space between stack j and stack(j +1).
        if such a j exist , then move stack j+1 , j+2 , .....till i one position to the left and hence create space
        for element between stack i and i +1 .

    3. Else
        If none of above is possible then there is no space left out in the one-dimensional array used hence
        print no space for push operation.
}

```

1.5 Introduction to Queue

Introduction:

1. It is basically a data object
2. The operational semantic of queue is FIFO i.e. first in first out

Definition:

It is an ordered list of elements n , such that $n > 0$ in which all deletions are made at one end called the front end and all insertions at the other end called the rear end.

Primary operations defined on a Queue:

1. **EnQueue** : This is used to add elements into the queue at the back end.
2. **DeQueue** : This is used to delete elements from a queue from the front end.
3. Also "IsEmpty()" and "IsFull()" can be defined to test whether the queue is Empty or full.

Example:

1. PRACTICAL EXAMPLE: A line at a ticket counter for buying tickets operates on above rules
2. IN COMPUTER WORLD: In a batch processing system, jobs are queued up for processing.

Circular queue:

In a queue if the array elements can be accessed in a circular fashion the queue is a circular queue.

Priority queue:

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

CIRCULAR QUEUE

Primary operations defined for a circular queue are:

1. add_circular - It is used for addition of elements to the circular queue.
2. delete_circular - It is used for deletion of elements from the queue.

We will see that in a circular queue, unlike static linear array implementation of the queue; the memory is utilized more efficient in case of circular queue's.

The shortcoming of static linear that once rear points to n which is the max size of our array we cannot insert any more elements even if there is space in the queue is removed efficiently using a circular queue.

As in case of linear queue, we'll see that condition for zero elements still remains the same i.e.. rear=front

ALGORITHM FOR ADDITION AND DELETION OF ELEMENTS

Data structures required for circular queue:

1. front counter which points to one position anticlockwise to the 1st element
2. rear counter which points to the last element in the queue
3. an array to represent the queue

```

add _ circular ( item,queue,rear,front)
{
    rear=(rear+1)mod n;
    if (front == rear )
        then print " queue is full "
    else
        {
            queue [rear]=item;
        }
}

```

delete operation :

```

delete_circular (item,queue,rear,front)
{
    if (front == rear)
        print ("queue is empty");
    else
        {
            front= front+1;
            item= queue[fromt];
        }
}

```

ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : addition is done only at the rear end of a queue like in a ticket counter line

add (item ,queue , n ,rear)

```
{
    if (rear==n)
        then print " queue is full "
    else
        {
            rear=rear+1;
            queue [rear]=item;
        }
}
```

ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : deletion is allowed only at the front end of the queue

delete (item , queue , rear , front)

```
{
    if (rear==front)
        then print "queue is empty";
    else
        {
            item = queue [front] ;
            front=front+1 ;
        }
}
```

Priority queue:

Queues are dynamic collections which have some concept of order. This can be either based on order of entry into the queue - giving us First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) queues. Both of these can be built with linked lists: the simplest "add-to-head" implementation of a linked list gives LIFO behavior. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

Performance

A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and independent of the number of items in the queue. Thus we class both addition and deletion as an $O(1)$ operation. For any given real machine + operating system + language combination, addition may take c_1 seconds and deletion c_2 seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, etc. The key point is that the time is not dependent on n - producing $O(1)$ algorithms.

Once we have written an $O(1)$ method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, etc will produce some significant improvement. However $O(1)$ methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

PRIORITY QUEUE:

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

This situation arises often in process control systems. Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem

(even if it is mass evacuation because nothing can stop the imminent explosion!).

Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

As we have seen, we could use a tree structure - which generally provides $O(\log n)$ performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to $O(n)$ in pathological cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other life-critical systems.

C ++ IMPLEMENTATION OF QUEUE USING CLASSES

```

#include <iostream.h>
#include <conio.h>

#define MAX 5 // MAXIMUM CONTENTS IN QUEUE

class queue
{
private:
    int t[MAX];
    int al; // Addition End
    int dl; // Deletion End
public:
    queue()
    {
        dl=-1;
        al=-1;
    }

    void del()
    {
        int tmp;
        if(dl!=-1)
        {

            cout<<"Queue is Empty";
        }
        else
        {
            for(int j=0;j<=al;j++)
            {
                if((j+1)<=al)
                {
                    tmp=t[j+1];
                    t[j]=tmp;
                }
                else
                {
                    al--;
                }

                if(al!=-1)
                    dl=-1;
                else
                    dl=0;
            }
        }
    }

    void add(int item)
    {
        if(dl!=-1 && al!=-1)
        {
            dl++;
            al++;
        }
        else
        {
            al++;
            if(al==MAX)
            {
                cout<<"Queue is Full\n";
                al--;
                return;
            }
        }
        t[al]=item;
    }

    void display()
    {
        if(dl!=-1)
        {
            for(int i=0;i<=al;i++)
                cout<<t[i]<<" ";
        }
        else
            cout<<"EMPTY";
    }
};

```

```

void main()
{
queue a;
int data[5]={32,23,45,99,24};

cout<<"Queue before adding Elements: ";
a.display();
cout<<endl<<endl;

for(int i=0;i<5;i++)
{
a.add(data[i]);
cout<<"Addition Number : "<<(i+1)<<" : ";
a.display();
cout<<endl;
}
cout<<endl;
cout<<"Queue after adding Elements: ";
a.display();

```

```

cout<<endl<<endl;

for(int i=0;i<5;i++)
{
a.del();
cout<<"Deletion Number : "<<(i+1)<<" : ";
a.display();
cout<<endl;
}
getch();
}

```

OUTPUT:

Queue before adding Elements: EMPTY

Addition Number : 1 : 32

Addition Number : 2 : 32 23

Addition Number : 3 : 32 23 45

Addition Number : 4 : 32 23 45 99

Addition Number : 5 : 32 23 45 99 24

Queue after adding Elements: 32 23 45 99 24

Deletion Number : 1 : 23 45 99 24

Deletion Number : 2 : 45 99 24

Deletion Number : 3 : 99 24

Deletion Number : 4 : 24

Deletion Number : 5 : EMPTY

As you can clearly see through the output of this program that addition is always done at the end of the queue while deletion is done from the front end of the queue.

Problems-Linear Data Structure

Tower of Hanoi Problem

Tower of Hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in

the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must confirm to the following rules,

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other.
3. The larger disk should not rest upon a smaller one.

Question: write a c program to implement tower of Hanoi using stack?

Solution:

```
#include <stdio.h>
#include <conio.h>
void move ( int, char, char, char );
void main()
{
    int n = 3 ;
    clrscr() ;
    move ( n, 'A', 'B', 'C' ) ;
    getch() ;
}
void move ( int n, char sp, char ap, char ep )
{
    if ( n == 1 )
        printf ( "\nMove from %c to %c ", sp, ep ) ;
    else
    {
        move ( n - 1, sp, ep, ap ) ;
        move ( 1, sp, ' ', ep ) ;
        move ( n - 1, ap, sp, ep ) ;
    }
}
```

Function Calls and Stack

A stack is used by programming languages for implementing function calls.

Write a program to check how function calls are made using stack.

SOLUTION

```
/* To show the use of stack in function calls */  
  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
#include <dos.h>  
  
unsigned int far *ptr ;  
void ( *p )( void ) ;  
  
void f1() ;  
void f2() ;  
  
void main()   
{  
    f1() ;  
    f2() ;  
    printf ( "\nback to main..." ) ;  
    exit ( 1 ) ;  
}  
  
void f1()   
{  
    ptr = ( unsigned int far * ) MK_FP ( _SS, _SP + 2 ) ;  
    printf ( "\n%d", *ptr ) ;  
    p = ( void ( * )( ) ) MK_FP ( _CS, *ptr ) ;  
    ( *p ) ( ) ;  
    printf ( "\nI am f1() function " ) ;  
}  
  
void f2()   
{  
    printf ( "\nI am f2() function " ) ;  
}
```

Check Your Progress

Q.2. Select the appropriate option for each of the following questions :

- i) The push() operation is used
 - a) to move an element
 - b) to remove an element
 - c) to insert an element
 - d) none of these
- ii) The stack is based on the rule
 - a) first-in-first-out
 - b) last-in-first-out
 - c) both (a) and (b)
 - d) none of these
- iii) A stack holding elements equal to its capacity and if push is performed then the situation is called
 - a) Stack overflow
 - b) Stack underflow
 - c) Pop
 - d) illegal operation
- iv) The top pointer is increased

- a) when push() operation is done
 - b) when pop() operation is done
 - c) both (a) and (b)
 - d) none of the above
- v) The pop operation removes
- a) the element lastly inserted
 - b) first element of the stack
 - c) any element randomly
 - d) none of the above

1.6 Answer to Check Your Progress

Ans. to Q. No. 1 : i) False; ii) True; iii) True

Ans. to Q. No. 2 : i) c) insert an element; ii) b) last-in-first-out; iii) a) Stack overflow; iv) a) when push() operation is done; v) a) the element lastly inserted

1.7 Model Questions

1. What is a stack? What different operations can be performed on stacks?
2. What is stack overflow and stack underflow?
3. What is linked implementation of stack?
4. Write a C program to implement stack with array. Perform push and pop operation.
5. What is queue? What are its properties?
6. What are the applications of queue?
7. What is overflow and underflow in a queue?

UNIT IV: LINKED LIST

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Linked Lists
- 1.4 Inserting and Removing Nodes from a list
- 1.5 Linked Implemented of Stacks
- 1.6 Getnode and Freenode Operation
- 1.7 Linked Implemented of Queue
- 1.8 List Implementation of Priority Queue
- 1.9 Header Nodes
- 1.10 Circular Lists
- 1.11 Doubly linked list
- 1.12 Answer to Check Your Progress
- 1.13 Model Questions

1.1 Learning Objectives

After going through this unit, you will able to:

- describe a linked list
- learn the types of linked list
- describe the advantages of linked list over arrays

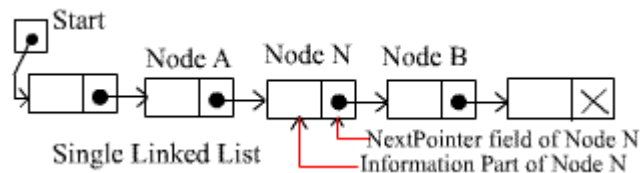
1.2 Introduction

We are already familiar with the array data structure. Array is a linear and homogenous data structure. It is very simple to represent a list of elements by using the array but it has some limitations. Once the size of the array is declared, it cannot be changed during program execution. The unused portion of an array also occupies the memory space. Insertion and deletion operation in an array is time consuming and it needs shifting of elements to rearrange the numbers in order. To overcome such types of difficulties, we will introduce another data structure called linked list in this unit. The different types of linked lists are also discussed in this unit.

1.3 Linked Lists

What is Linked Lists?

A linked list is simply a chain of structures which contain a pointer to the next element. It is dynamic in nature. Items may be added to it or deleted from it at will.



This definition applies only to Singly Linked Lists - Doubly Linked Lists and Circular Lists are different.

A list item has a pointer to the next element or to 0 if the current element is the tail (end of the list). This pointer points to a structure of the same type as itself. This structure that contains elements and pointers to the next structure is called a Node.

Each node of the list has two elements:

- the item being stored in the list *and*
- a pointer to the next item in the list

Some common examples of a linked list:

- Hash tables use linked lists for collision resolution
- Any "File Requester" dialog uses a linked list
- Binary Trees
- Stacks and Queues can be implemented with a doubly linked list
- Relational Databases (e.g. Microsoft Access)

1.4 Inserting and Removing Nodes from a list

Algorithm for inserting a node to the List

- allocate space for a new node,
- copy the item into it,
- make the new node's next pointer point to the current head of the list and

- Make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list.

Below is given C code for inserting a node after a given node.

C Implementation

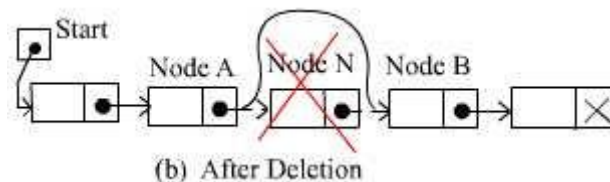
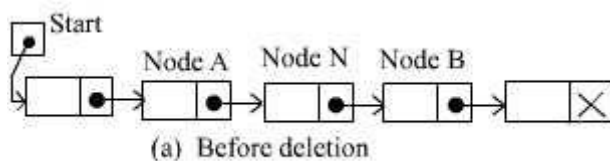
```

/* inserts an item x into a list after a node pointed to by p */

void insafter(int p, int x)
{
int q;
if(p == -1)
{
printf("void insertion\n");
return;
}
q = getnode(); /* getnode() returns a pointer to newly allocated node */
node[q].info = x;
node[q].next = node[p].next;
node[p].next = q;
return;
} /* end insafter

```

Algorithm for deleting a node from the List



Step-1: Take the value in the 'nodevalue' field of the TARGET node in any intermediate variable. Here node N.

Step-2: Make the previous node of TARGET to point to where TARGET is currently pointing

Step-3: The nextpointer field of Node N now points to Node B, Where Node N previously pointed.

Step-4: Return the value in that intermediate variable

There are also two special cases. If the deleted node N is the first node in the list, then the Start will point to node B; and if the deleted node N is the last node in the list, then Node A will contain the NULL pointer.

Below is a C code for deleting a node after a given node.

C Implementation

```
/* routine delafter(p,px), called by the statement delafter(p,&x), deletes the
node following node(p) and stores its contents in x */
```

```
void delafter(int p, int *px)
```

```
{
    int q;
    if ((p == -1) || (node[p].next == -1))
    {
printf("void deletion\n");
return;
    }
    q=node[p].next;
    *px = node[q].info;
    node[p].next = node[q].next;
    freenode(q);
return;
}
```

```
/* end delafter */
```

1.5 Linked Implemented of Stacks

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element on to a stack. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its

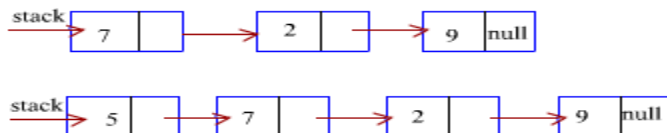
first element. Similarly, removing the first element from a linked list is analogous to popping from a stack.

A linked-list is somewhat of a dynamic array that grows and shrinks as values are added to it and removed from it respectively. Rather than being stored in a continuous block of memory, the values in the dynamic array are linked together with pointers. Each element of a linked list is a structure that contains a value and a link to its neighbor. The link is basically a pointer to another structure that contains a value and another pointer to another structure, and so on. If an external pointer p points to such a linked list, the operation $\text{push}(p, t)$ may be implemented by

```
f = getnode();
    info(f) = t;
    next(f) = p;
p = f;
```

The operation $t = \text{pop}(p)$ removes the first node from a nonempty list and signals underflow if the list is empty

```
if(empty(p))
    {
    printf('stack underflow');
    exit(1);
    }
else {
    f = p;
    p = next(f);
    t = info(f);
    freenode(f);
    }
```



1.6 Getnode and Freenode Operation

GETNODE AND FREENODE OPERATIONS

The `getnode` operation may be regarded as a machine that manufactures nodes. Initially there exist a finite pool of empty nodes and it is impossible to use more than that number at a given instant. If it is desired to use more than that number over a given period of time, some nodes must be reused. The function of `freenode` is to make a node that is no longer being used in its current context available for reuse in a different context.

The list of available nodes is called the available list. When the available list is empty that is all nodes are currently in use and it is impossible to allocate any more, overflow occurs.

Assume that an external pointer `avail` points to a list of available nodes. Then the operation `getnode` and `freenode` are implemented as follows:

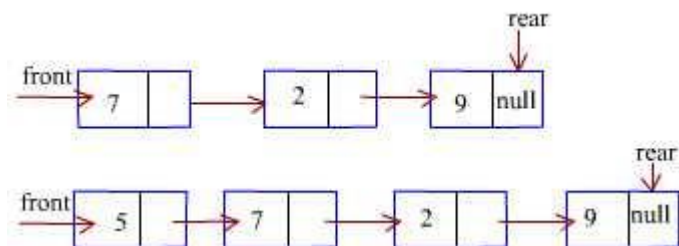
```
int getnode(void)
{
int p;
    if (avail == -1)
    {
        printf("overflow\n");
        exit(1);
    }
    p = avail;
    avail = node[avail].next;
return(p);
} /* end getnode */

void freenode (int p)
{
node[p].next = avail;
avail = p;
return;
} /* end freenode */
```

1.7 Linked Implemented of Queue

Queues can be implemented as linked lists. Linked list implementations of queues often require two pointers or references to links at the beginning and end of the list. Using a pair of pointers or references opens the code up to a variety of bugs especially when the last item on the queue is dequeued or when the first item is enqueued.

In a circular linked list representation of queues, ordinary 'for loops' and 'do while loops' do not suffice to traverse a loop because the link that starts the traversal is also the link that terminates the traversal. The empty queue has no links and this is not a circularly linked list. This is also a problem for the two pointers or references approach. If one link in the circularly linked queue is kept empty then traversal is simplified. The one empty link simplifies traversal since the traversal starts on the first link and ends on the empty one. Because there will always be at least one link on the queue (the empty one) the queue will always be a circularly linked list and no bugs will arise from the queue being intermittently circular. Let a pointer to the first element of a list represent the front of the queue. Another pointer to the last element of the list represents the rear of the queue as shown in fig. illustrates the same queue after a new item has been inserted.



Under the list representation, a queue q consists of a list and two pointers, $q.front$ and $q.rear$. The operations are insertion and deletion. Special attention is required when the last element is removed from a queue. In that case, $q.rear$ must also be set to null, Since in an empty queue both $r.front$ and $q.rear$ must be null.

The pseudo code for deletion is below:

```
if (empty(q))
{
```

```

printf("Queue is Underflow");
exit(1);
}
f = q.front;
t = info(f);
q.front = next(f);
if (q.front == null)
q.rear = null;
freenode(f);
return(t);

```

The operation insert algorithm is implemented

```

f = getnode();
info(f) = x;
next(f) = null;
if (q.rear == null)
q.front = f;
else
next(q.rear) = f;
q.rear = f;

```

1.8 List Implementation of Priority Queue

We can use a list to represent a priority queue in ordered list or unordered list. For an ascending Priority queue, insertion is implemented by the place operation, which keeps the list ordered, and deletion of the minimum element is implemented by the delete operation, which removes the first element from the list. A Descending priority queue can be implemented by keeping the list in descending order rather than ascending, or by using remove to delete the minimum element. In an ordered list, if you want to insert an element to the priority queue, it will require examining an average of approximately $n/2$ nodes but only one search for deletion.

An unordered list may also be used as a priority queue. If you want to insert an element to the list always requires examining only one node but always requires examining n elements for removal of an element.

The advantage of a list over an array for implementing a priority queue is that an element can be inserted into a list without moving any other elements, where as this is impossible for an array unless extra space is left empty.

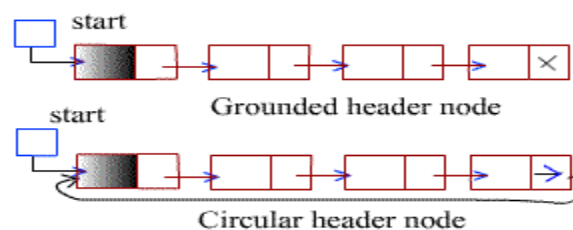
1.9 Header Nodes

A header linked list is a linked list which always contains a special node called the header node at the beginning of the list. It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused. There are two types of header list

- Grounded header list : is a header list where the last node contain the null pointer.
- Circular header list : is a header list where the last node points back to the header node.

More often, the information portion of such a node could be used to keep global information about the entire list such as:

- number of nodes (not including the header) in the list
 - count in the header node must be adjusted after adding or deleting the item from the list
- pointer to the last node in the list
 - it simplifies the representation of a queue
- pointer to the current node in the list
 - eliminates the need of a external pointer during traversal



1.10 Circular Lists

Circular lists are like singly linked lists, except that the last node contains a pointer back to the first node rather than the null pointer. From any point in such a list, it is possible to reach any other point in the list. If we begin at a given node and travel the entire list, we ultimately end up at the starting point. Note that a circular list does not have a natural "first or "last" node. We must therefore, establish a first and last node by convention - let external pointer point to the last node, and the following node be the first node.

Stack as a Circular List

A circular list can be used to represent a stack.

The following is a C function to push an integer x onto the stack. It is called by push (&stack, x), where stack is a pointer to a circular list acting as a stack.

```
void push(NODEPTR *pstack, int x)
{
NODEPTR p;

p = getnode();

p->info = x;

if (*pstack == NULL)

*pstack = p;

else

p->next = (*pstack) -> next;

(*pstack) -> next = p;

} /*end push*/
```

Queue as a circular list

It is easier to represent a queue as a circular list than as a linear list. If considered as a linear list, a queue is specified by two pointers, one to the front of the list and other to its rear. However, by using a circular list, a queue may

be specified by a single pointer q to that list. Node (q) is the rear of the queue and the following node is its front.

Queue as a circular list

The following is a C function to insert an integer x into the queue and is called by insert (&q, x)

```
Void insert (NODEPTR *pq, int x)
{
NODEPTR p;
p = getnode();
p->info = x;
if (*pq == NULL)
    *pq = p;
else
p->next = (*pq) -> next;
(*pq) -> next = p;
*pq = p;
return;
} /*end insert*/
```

Queue as a circular list

To insert an element into the rear of a circular queue, the element is inserted into the front of the queue and the circular list pointer is then advanced one element, so that the new element becomes the rear.

Exercise: Write an algorithm and a C routine to concatenate two circular lists.

1.11 Doubly linked list

Doubly linked lists are like singly linked lists, in which for each node there are two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.
- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list and
- You can delete nodes very easily.

Doubly linked lists may be either linear or circular and may or may not contain a header node.

Check Your Progress

Q.1. Select the appropriate answer from the following:

- a) The elements of linked list are stored in
 - i) successive memory locations
 - ii) alternate memory locations
 - iii) random memory locations
 - iv) all of the above
- b) The linked list where first and last node cannot be identified is
 - i) doubly linked list
 - ii) singly linked list
 - iii) circular linked list
 - iv) all the above
- c) The linked list in which the first node have the NULL values is
 - i) doubly linked list
 - ii) singly linked list
 - iii) circular linked list
 - iv) all the above
- d) The function used for memory allocation in linked list is
 - i) malloc ()
 - ii) calloc ()
 - iii) realloc ()
 - iv) free ()

1.12 Answer to Check Your Progress

Ans. to Q. No. 1: a) i) successive memory locations b) iii) Circular linked list
c) i)Doubly linked list d) i) malloc ()

1.13 Model Questions

1. Why is a linked list called a dynamic data structure?
2. What are the advantages of using a linked list over arrays?
3. Explain types of linked lists. On which basis is the linked list classified?

4. List the advantages of doubly linked list over singly linked list.
5. Compare circular linked list with doubly linked list and list their relative merits and demerits.

Block-2

UNIT V: SORTING

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Sink Sort
- 1.4 Insertion Sort
- 1.5 Selection Sort
- 1.6 Bubble Sort
- 1.7 Merge Sort
- 1.8 Quick Sort
- 1.9 Radix Sort
- 1.10 Answers to Check Your Progress
- 1.11 Model Questions

1.1 Learning Objectives

After going through this unit, you will able to

- define sorting
- describe insertion sort algorithm
- describe selection sort algorithm
- explain bubble sort algorithm
- describe quick sort algorithm
- Compute the complexity of the above sorting algorithms

1.2 Introduction

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Efficient sorting is important for optimizing the use of other algorithms that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. In this unit, we will introduce you to the fundamental concepts of sorting. Also, we shall discuss the various sorting algorithms i.e. insertion sort, selection sort, bubble sort and quick sort including their complexity.

1.3 Sink Sort

- Main idea in this method is to compare two adjacent elements and put them in proper order if they are not.
- Do this by scanning the element from first to last.
- After first scan, largest element is placed at last position. This is like a largest object sinking to bottom first.
- After second, scan second largest is placed at second last position.
- After n passes all elements are placed in their correct positions, hence sorted.

Input	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8
12	12	12	12	12	12	12	12	12
32	18	18	18	18	18	14	14	14
18	24	24	19	19	14	18	18	18
24	30	19	24	14	19	19	19	19
30	19	28	14	24	24	24	24	24
19	28	14	28	28	28	28	28	28
28	14	30	30	30	30	30	30	30
14	32	32	32	32	32	32	32	32

The algorithmic description of the Sink sort is given below:

Algorithm Sink-Sort (a[n])

Step 1: for i = 0 to n-2 do

Step 2: for $j = 0$ to $n-i$ do

Step 3: if $(a[j] > a[j+1])$ then

Step 4 : $\text{swap}(a[j],a[j+1]);$

The algorithm can be modified, such that, after each pass next smallest element reach to the top position. This is like a bubble coming to top. Hence called Bubble-Sort.

1.4 Insertion Sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting of all following elements over by one.

Let A is an array with n elements $A[0], A[1], \dots, \dots, \dots, A[N-1]$ in memory.

The insertion sort algorithm scans the array from $A[0]$ to $A[N-1]$, and the process of inserting each element in proper place is as follow

Pass 1: $A[0]$ by itself is sorted because of one element.

Pass 2: $A[1]$ is inserted either before or after $A[0]$ so that $A[0], A[1]$ are sorted.

Pass 3: $A[2]$ is inserted into its proper place in $A[0], A[1]$, i.e. before $A[0]$, Between $A[0]$ and $A[1]$, or after $A[1]$, so that $A[0], A[1], A[2]$ are sorted.

Pass 4 : $A[3]$ is inserted into its proper place in $A[0], A[1], A[2]$ so that $A[0], A[1], A[2], A[3]$ are sorted.

.....
.....

Pass N : $A[N-1]$ is inserted into its proper place in $A[0], A[1], \dots, \dots, \dots, A[N-2]$ so that $A[0], A[1], \dots, \dots, \dots, A[N-1]$ is sorted.

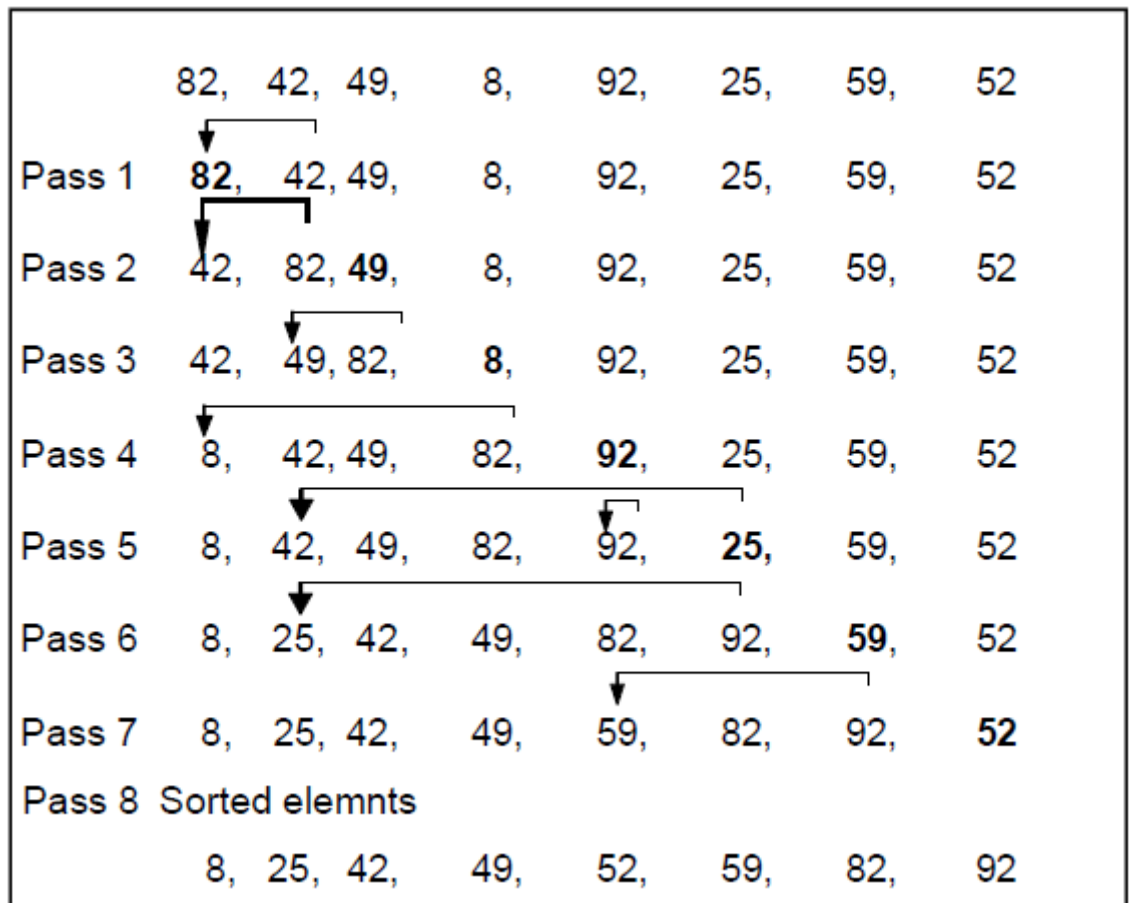
Insertion sort algorithm is frequently used when n is small.

The element inserted in the proper place is compared with the previous elements and placed in between the i th element and $(i+1)$ th element if :

element \leq i th element

element \geq $(i+1)$ th element

Let us take an example of the following elements:



Finally, we get the sorted array. The following program uses the insertion sort technique to sort a list of numbers.

```

/* Program of sorting using insertion sort */
#include<stdio.h>
#include<conio.h>
void main()
{
int a[25], i, j, k, n;
clrscr();
printf("Enter the number of elements : ");
scanf("%d",&n);
for (i = 0; i < n; i++)
{
printf("Enter element%d : ",i+1);
scanf("%d", &a[i]);
}

```

```

printf("Unsorted list is :\n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
/*Insertion sort*/
for(j=1;j<n;j++)
{
k=a[j]; /*k is to be inserted at proper place*/
for(i=j-1;i>=0 && k<a[i];i--)
a[i+1]=a[i];
a[i+1]=k;
printf("Pass %d, Element %d inserted in proper
place \n",j,k);
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
}
printf("Sorted list is :\n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
getch();
} /*End of main()*/

```

Analysis: In insertion sort we insert the element i before or after and we start comparison from the first element. Since the first element has no other elements before it, so it does not require any comparison. Second element requires 1 comparison, third element requires 2 comparisons, fourth element requires 3 comparisons and so on. The last element requires $n-1$ comparisons.

So the total number of comparisons will be

$$1 + 2 + 3 + \dots + (n-2) + (n-1)$$

It is a form of arithmetic progression series, so we can apply

the formula $\text{sum} = \frac{n}{2} \{2a + (n-1)d\}$

where $d = \text{common difference i.e. first term} - \text{second term}$,
 $a = \text{first term in the series}$, $n = \text{total term}$

$$\begin{aligned}\text{Thus sum} &= \frac{(n-1)}{2} \{2 \times 1 + (n-1) \times 1\} \times 1 \\ &= \frac{(n-1)}{2} (2 + n - 2) \\ &= n^2 \frac{(n-1)}{2}\end{aligned}$$

which is of $O(n^2)$.

It is the worst case behavior of insertion sort where all the elements are in reverse order. If we compute the best case of the above algorithm then it will be of $O(n)$. The insertion sort technique is very efficient if the number of element to be sorted are very less.

1.5 Selection Sort

Let us have a list containing n elements in unsorted order and we want to sort the list by applying the selection sort algorithm. In selection sort technique, the first element is compared with all remaining $(n - 1)$ elements.

The smallest element is placed at the first location. Again, the second element is compared with the remaining $(n - 2)$ elements and the smallest element is picked out from the list and placed in the second location and so on until the largest element of the list.

Let A is an array with n elements $A[0], A[1], \dots, \dots, A[N-1]$. First you will search the position of smallest element from $A[0] \dots \dots A[N-1]$. Then you will interchange that smallest element with $A[0]$. Now, you will search the position of the second smallest element (because the $A[0]$ is the first smallest element) from $A[1] \dots \dots A[N-1]$, then interchange that smallest element with $A[1]$. Similarly, the process will be for $A[2] \dots \dots A[N-1]$. The whole process will be as follows –

Pass 1: Search the smallest element from $A[0], A[1], \dots, \dots, A[N-1]$.

Interchange $A[0]$ with the smallest element

Result : $A[0]$ is sorted.

Pass 2: Search the smallest element from A[1], A[2], ..., ...,A[N-1].

Interchange A[1] with the smallest element

Result: A[0], A[1] is sorted.

.....

Pass N-1: Search the smallest element from A[N-2],A[N-1]

Interchange A[N -2] with the smallest element

Result :A[0],A[1], ..., ..., ...,A[N-2],A[N-1] is sorted.

Thus A is sorted after N -1 passes.

Let us take an example of the following elements:

Pass 1	75	35	42	13	87	24	64	57
Pass 2	13	35	42	75	87	24	64	57
Pass 3	13	24	42	75	87	35	64	57
Pass 4	13	24	35	75	87	42	64	57
Pass 5	13	24	35	42	87	75	64	57
Pass 6	13	24	35	42	57	75	64	87
Pass 7	13	24	35	42	57	64	75	87
Sorted								
Elements	13	24	35	42	57	64	75	87

```
/*Program of sorting using selection sort*/
```

```
#include <stdio.h>
```

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int a[25], i, j, k, n, temp, smallest;
```

```
clrscr();
```

```
printf("Enter the number of elements : ");
```

```
scanf("%d",&n);
```

```
for (i = 0; i < n; i++)
```

```
{
```

```

printf("Enter element%d : ",i+1);
scanf("%d", &a[i]);
}
/* Display the unsorted list */
printf("Unsorted list is : \n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
/*Selection sort*/
for(i = 0; i < n - 1 ; i++)
{
/*Find the smallest element*/
smallest = i;
for(k = i + 1; k < n ; k++)
{
if(a[smallest] > a[k])
smallest = k ;
}
if( i != smallest )
{
temp = a [i];
a[i] = a[smallest];
arr[smallest] = temp ;
}
printf("After Pass %d elements are : ", i+1);
for (j = 0; j < n; j++)
printf("%d ", a[j]);
printf("\n");
} /*End of for*/
printf("Sorted list is : \n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
getch();

```

```
} /*End of main()*/
```

Analysis: As we have seen, selection sort algorithm will search the smallest element in the array and then that element will be at proper position. So, in Pass 1 it will compare $n-1$ elements, in Pass 2 comparison will be $n-2$ because the first element is already at proper position. Thus, we can write the function for comparison as

$$F(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

This is an arithmetic series, solving the series we will get

$$\begin{aligned} F(n) &= \frac{(n-1)}{2} [2(n-1) + \{(n-1)-1\} \{(n-2)-(n-1)\}] \\ &= \frac{(n-1)}{2} [2n-2 + (n-1-1)(n-2-n+1)] \\ &= \frac{(n-1)}{2} [2n-2 + (n-2)(-1)] \\ &= \frac{(n-1)}{2} [2n-2-n+2] \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Thus, the number of comparisons is proportional to (n^2) . It is the worst case behavior of selection sort. If we compute the average case of the above algorithm then it will be of $O(n^2)$. The best thing with the selection sort is that in every pass one element will be at correct position, very few temporary variables will be required for interchanging the elements and it is simple to implement.

1.6 Bubble Sort

Bubble sort is a commonly used sorting algorithm. In this algorithm, two successive elements are compared and interchanging is done if the first element is greater than the second one.

Let A is an array with n elements $A[0], A[1], \dots, \dots, A[N-1]$. The bubble sort algorithm works as follows :

Step 1 First compare $A[0]$ and $A[1]$ and arrange them so that

$A[0] < A[1]$. Then compare $A[1]$ and $A[2]$ and arrange them so that $A[1] < A[2]$. Then compare $A[2]$ and $A[3]$ and arrange them so that $A[2] < A[3]$. Continue until we compare $A[N-2]$ and $A[N-1]$ and arrange them into the desired order so that $A[N-2] < A[N-1]$. Observe that Step 1 involves $n-1$ comparisons. During the Step 1, the largest element is “bubbled up” to the n th position or “sinks” to the n th position. When Step 1 is completed, $A[N-1]$ will contain the largest element.

Step 2 Repeat Step 1 and finally the second largest element will occupy $A[N-2]$. In this step there will be $n-2$ comparisons.

Step 3 Repeat Step 1 and finally the third largest element will occupy $A[N-3]$. In this step there will be $n-3$ comparisons.

.....

Step $N-1$ Compare $A[1]$ and $A[2]$ and arrange them so that $A[1] < A[2]$. After $n-1$ steps, the list will be sorted in ascending order.

Let us take an example of the following elements

	0	1	2	3	4	5	6	7
A	13	32	20	62	68	52	38	46

we will apply the bubble sort algorithm to sort the elements.

Pass 1

We have the following comparisons

- a) Compare A_0 and A_1 , $13 < 32$, no change
- b) Compare A_1 and A_2 , $32 > 20$, interchange
 13 20 32 62 68 52 38 46
- c) Compare A_2 and A_3 , $32 < 62$, no change
- d) Compare A_3 and A_4 , $62 < 68$, no change
- e) Compare A_4 and A_5 , $68 > 52$, interchange
 13 20 32 62 52 68 38 46
- f) Compare A_5 and A_6 , $68 > 38$, interchange
 13 20 32 62 52 38 68 46
- g) Compare A_6 and A_7 , $68 > 46$, interchange

13 20 32 62 52 38 46 68

Pass 2

13 20 32 62 52 38 46 68

- a) Compare A0 and A1, $13 < 20$, no change
- b) Compare A1 and A2, $20 < 32$, no change
- c) Compare A2 and A3, $32 < 62$, no change
- d) Compare A3 and A4, $62 > 52$, interchange

13 20 32 52 62 38 46 68

- e) Compare A4 and A5, $62 > 38$, interchange

13 20 32 52 38 62 46 68

- f) Compare A5 and A6, $46 > 62$, interchange

13 20 32 52 38 46 62 68

At the end of the Pass 2 the second largest element 62 has moved to its proper place.

Pass 3

13 20 32 52 38 46 62 68

- a) Compare A0 and A1, $13 < 20$, no change
- b) Compare A1 and A2, $20 < 32$, no change
- c) Compare A2 and A3, $32 < 52$, no change
- d) Compare A3 and A4, $52 > 38$, interchange

13 20 32 38 52 46 62 68

- e) Compare A4 and A5, $52 > 46$, interchange

13 20 32 38 46 52 62 68

Pass 4

13 20 32 38 46 52 62 68

- a) Compare A0 and A1, $13 < 20$, no change
- b) Compare A1 and A2, $20 < 32$, no change
- c) Compare A2 and A3, $32 < 38$, no change
- d) Compare A3 and A4, $38 < 46$, no change

Pass 5

13 20 32 38 46 52 62 68

- a) Compare A0 and A1, $13 < 20$, no change

b) Compare A1 and A2, $20 < 32$, no change

c) Compare A2 and A3, $32 < 38$, no change

Pass 6

13 20 32 38 46 52 62 68

a) Compare A0 and A1, $13 < 20$, no change

b) Compare A1 and A2, $20 < 32$, no change

Pass 7

13 20 32 38 46 52 62 68

a) Compare A0 and A1, $13 < 20$, no change

Since the list has 8 elements, it is sorted after the 7th Pass. The list was actually sorted after the 4th Pass.

```
/* Program of sorting using bubble sort */
#include<stdio.h>
#include<conio.h>
void main()
{
int a[25], i, j, k, temp, n, xchanges;
clrscr();
printf("Enter the number of elements : ");
scanf("%d",&n);
for (i = 0; i < n; i++)
{
printf("Enter element %d : ", i+1);
scanf("%d",&a[i]);
}
printf("Unsorted list is :\n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\n");
/* Bubble sort*/
for (i = 0; i < n-1 ; i++)
{
xchanges=0;
```

```

for (j = 0; j <n-1-i; j++)
{
if (a[j] > a[j+1])
{
temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
xchanges++;
} /*End of if*/
} /*End of inner for loop*/
if(xchanges==0) /*If list is sorted*/
break;
printf("After Pass %d elements are : ",i+1);
for (k = 0; k < n; k++)
printf("%d ", a[k]);
printf("\n");
} /*End of outer for loop*/
printf("Sorted list is :\n");
for (i = 0; i < n; i++)
printf("%d ", a[i]);
getch();
printf("\n");
} /*End of main()*/

```

Analysis: As we have seen, bubble sort algorithm will search the largest element in the array and place it at proper position in each Pass. So in Pass 1 it will compare n -1 elements, in Pass 2 comparison will be n -2 because the first element is already at proper position. Thus, we can write the function for comparison as

$$F(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

This is an arithmetic series, solving the series we will get

$$\begin{aligned}
F(n) &= \frac{n}{2} \{2a + (n-1)d\} \\
&= \frac{(n-1)}{2} \{2 \times 1 + (n-1)-1 \times 1\} \\
&= \frac{(n-1)}{2} (2 + n-2) \\
&= \frac{n(n-1)}{2} \\
&= O(n^2)
\end{aligned}$$

Thus, the time required to execute the bubble sort algorithm is proportional to (n^2) , where n is the number of input items.

1.7 Merge Sort

- The time complexity of the sorting algorithms discussed till now are in $O()$. That is, the numbers of comparisons performed by these algorithms are bound above by c , for some constant $c > 1$.
- Can we have better sorting algorithms? Yes, merge sort method sorts given a set of number in $O(n \log n)$ time.
- Before discussing merge sort, we need to understand what is the meaning of merging?

Merge sort Method:

Definition: Given two sorted arrays, $a[p]$ and $b[q]$, create one sorted array of size $p + q$ of the elements of $a[p]$ and $a[q]$.

- Assume that we have to keep $p+q$ sorted numbers in an array $c[p+q]$.
- One way of doing this is by copying elements of $a[p]$ and $b[q]$ into $c[p+q]$ and sort $c[p+q]$ which has time complexity of at least $O(n \log n)$.
- Another efficient method is by merging which is of time complexity of $O(n)$
- Method is simple, take one number from each array a and b , place the smallest element in the array c . Take the next elements and repeat the procedure till all $p+q$ element are placed in the array c

Pseudocode of the merging is given below.

Algorithm Merge (a[p], b[q])

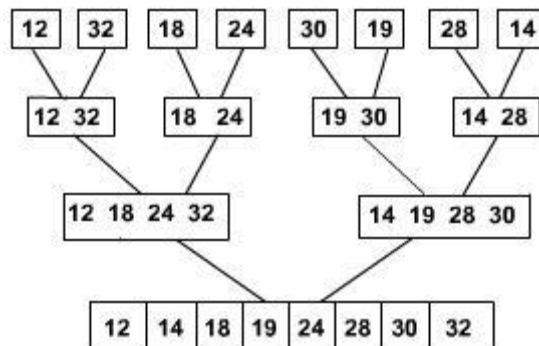
```

Step 1:  i = 0
Step 2:  j = 0
Step 3:  k = 0
Step 4:  while (i < p) && (j < q) do{
Step 5:      if (a[i] < b[j]) {
Step 6:          c[k] = a[i] ;
Step 7:          i = i + 1 ; }
Step 8:      else {
Step 9:          c[k] = b[j]
Step 10:         j = j + 1 }
Step 11:         k = k + 1 }
Step 12:  if (i == p) then
Step 13:      while ( j < q) do{
Step 14:          c[k] = b[j] ;
Step 15:          k = k + 1 ;
Step 16:          j = j + 1 ;
                }
Step 17:  else
Step 18:      while(i < p) do
Step 19:          { c[k] = a[i] ;
Step 20:            k = k + 1 ;
Step 21:            i = i + 1 ;
                }

```

We can assume each element in a given array as a sorted sub array. Take adjacent arrays and merge to obtain a sorted array of two elements. Next step, take adjacent sorted arrays, of size two, in pair and merge them to get a sorted array of four elements. Repeat the step until whole array is sorted.

Following figure illustrates the procedure.



- Assume that procedure Merge1 (a, i, j, k) takes two sorted arrays a[i..j] and a[j+1..k] as an input and puts their merged output in the array a[i..k]. That is, in the same locations. Pseudo code of the merge sort is given below.

Algorithm Merge-Sort (a, p, q)

```
Step 1:  if (p < q) {  
Step 2:      mid = (p+q)/2 ;  
Step 3:      Merge-Sort(a, p, mid) ;  
Step 4:      Merge-Sort(a, mid+1, q) ;  
Step 5:      Merge(a,p,mid, q) ;  
        }  
        }
```

- The above procedure sorts the elements in the sub array a[p..q].
- To sort given any a[n] invoke the algorithm with parameters (a, 0, n -1).

1.8 Quick Sort

This sorting algorithm was invented by C.A.R. Hoare in 1960. This algorithm is based on partition, hence, it falls under the divide and conquer technique. In this algorithm, the list of elements are divided into two sub lists.

For example, a list of n elements are to be sorted. The quick sort marks the first element in the list called as pivot or key. Consider the first element as P. Shift all the elements whose value is less than P towards the left and elements whose value is greater than P to the right of P. Now, the pivot element divides the main list into two parts. It is not necessary that the selected key element must be in the middle.

Now, the process for sorting the elements through quick sort is as follows:

1. Take the first element of list as pivot.
2. Place pivot at the proper place in list. So at least one element of the list i.e. pivot will be at its proper place.
3. Create two sub lists in left and right side of pivot.
4. Repeat the same process until all elements of list are at proper position in list.

For placing the pivot element at proper place we need to do the

Following process–

5. Compare the pivot element one by one from right to left for getting the element which has value less than pivot element.
6. Interchange the element with pivot element.
7. Now the comparison will start from the interchanged element position from left to right for getting the element which has higher value than pivot.
8. Repeat the same process until pivot is at its proper position.

Let us take a list of elements and assume that 48 is the pivot element.

48 29 8 59 72 88 42 65 95 19 82 68

We have to start comparison from right to left. Now the first element less than 48 is 19. So interchange it with pivot i.e. 48.

19 29 8 59 72 88 42 65 95 48 82 68

Now, the comparison will start from 19 and will be from left to right.

The first element greater than 48 is 59. So interchange it with pivot.

19 29 8 48 72 88 42 65 95 59 82 68

Now, the comparison will start from 59 and will be from right to left.

The first element less than 48 is 42. So interchange it with pivot.

19 29 8 42 72 88 48 65 95 59 82 68

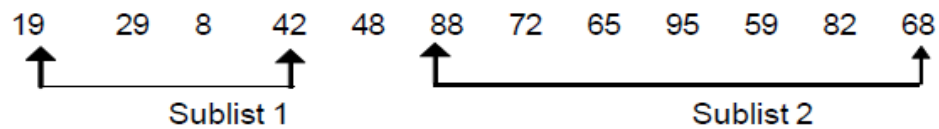
Now, the comparison will start from 42 and will be from left to right.

The first element greater than 48 is 72. So interchange it with pivot.

19 29 8 42 48 88 72 65 95 59 82 68

Now, the comparison will start from 72 and will be from right to left.

There is no element less than 48. So, now 48 is at its proper position in the list. So we can divide the list into two sub list, left and right side of pivot.



Now, the same procedure will be followed for the sublist1 and sublist2 and finally you will get the sorted list.

The following program will demonstrate the quick sort algorithm:

```
#include<stdio.h>

#include<conio.h>

enum bool { FALSE,TRUE };

void main()

{

int elem[20],n,i;

clrscr();

printf("Enter the number of elements : ");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Enter element%d : ",i+1);

scanf("%d",&elem[i]);

}

printf("Unsorted list is :\n");

display(elem,0,n-1);

printf("\n");

quick(elem,0,n-1);

printf("Sorted list is :\n");

show(elem,0,n-1);

getch();
```

```

printf("\n");

}/*End of main() */

quick(int a[ ],int low, int up)
{
int piv, temp, left, right;

enumbool pivot_placed=FALSE;

left=low;

right=up;

piv=low; /*Take the first element of sublist as piv */

if(low>=up)

return;

printf("Sublist : ");

show(a,low,up);

/*Loop till pivot is placed at proper place in the sublist*/

while(pivot_placed==FALSE)

{

/*Compare from right to left */

while( a[piv]<=a[right] && piv!=right )

right=right-1;

if( piv==right )

pivot_placed=TRUE;

if( a[piv] > a[right] )

{

temp=a[piv];

a[piv]=a[right];

```

```

a[right]=temp;

piv=right;

}

/*Compare from left to right */

while( a[piv]>=a[left] && left!=piv )

left=left+1;

if(piv==left)

pivot_placed=TRUE;

if( a[piv] < a[left] )

{

temp=a[piv];

a[piv]=a[left];

a[left]=temp;

piv=left;

}

} /*End of while */

printf("-> Pivot Placed is %d -> ",a[piv]);

show(a,low,up);

printf("\n");

quick(a, low, piv-1);

quick(a, piv+1, up);

}/*End of quick()*/

show(int a[ ], int low, int up)

{

int i;

```

```

for(i=low;i<=up;i++)
printf("%d ",a[i]);
}

```

Analysis: The time required by the quick sorting method (i.e. the efficiency of the algorithm) depends on the selection of the pivot element.

Suppose that, the pivot element is chosen in the middle position of the list so that it divides the list into two sub lists of equal size. Now, repeatedly applying the quick sort algorithm on both the sub lists we will finally have the sorted list of the elements.

Now, after 1st step, total elements in correct position is $= 1 = 2^1 - 1$

after 2nd step, total elements in correct position is $= 3 = 2^2 - 1$

after 3rd step, total elements in correct position is $= 7 = 2^3 - 1$

.....

after lth step, total elements in correct position is $= 2^l - 1$

Therefore, $2^l - 1 = n - 1$

or $2^l = n$

or $l = \log_2 n$

Here, the value of l is the number of steps. If n is comparison per step then for $\log_2 n$ steps we get $= n \log_2 n$ comparisons. Thus, **the overall time complexity of quick sort is $= n \log_2 n$**

The above calculated complexity is called the *average case complexity*. So, *the condition for getting the average case complexity is choosing the pivot element at the middle of the list.*

Now, suppose one condition is that, the given list of the elements is initially sorted. We consider the first element of the list as the pivot element.

In this case, the number of steps needed for obtaining the finally sorted list is $= (n-1)$.

Again, the number of comparisons in each step will be almost n i.e. it will be of $O(n)$.

So, the total number of comparisons for $(n-1)$ steps is

$$= (n-1) O(n)$$

$$= O(n^2)$$

Thus, the time complexity in this case will be $O(n^2)$. This is called the **worst case complexity** of the quick sort algorithm. *So, the condition for worst case is if the list is initially sorted.*

1.9 Radix Sort

- All the algorithms discussed till now are comparison based algorithms. That is, comparison is the key for sorting.
- Assume that the input numbers are three decimal digits. First we sort the numbers using least significant digit. Then by next significant digit and so on.
- See following example for illustration.

327	470	418	146
476	382	327	173
285	173	146	259
418	285	259	285
568	476	568	327
382	146	470	382
146	327	173	418
259	418	476	470
173	568	382	476
470	259	285	568
Input	Sorted by LS digit	Sorted by Middle digit	Sorted by Most Sig digit

- Next question is how to sort with respect to a least significant digit or any significant digit?
- We can use any sorting algorithm we have discussed.
 - Another way by maintaining a BIN for each digit from 0 to 9. If the digit is X put the number in BIN X. Concatenate the BINs from 0 to 9 to get a sorted list of numbers of that significant digit.
- The method is given in the following pseudo code

Algorithm radix-sort (a)

```

Step 1:  for i = 0 to 9 do
Step 2:      empty-bin(i);
Step 3:      for position = least significant digit to most significant digit
Step 4:          for i = 1 to n do
Step 5:              x = digit in the position of a[i];

Step 6:              put a[i] in BIN x;
Step 7:              i = 1;
Step 8:              for j = 0 to 9 do
Step 9:                  while (BIN(j) <> empty) do
Step 10:                     a[i] = get-element(BIN(j));
Step 11:                     i = i + 1;

```


- Procedure get-element (bin) gets the next element from the bin.
- We can use Queues for implementing Bins.

Check Your Progress

Q.1. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category?

- A. $O(n \log n)$ sorts
- B. Divide-and-conquer sorts
- C. Interchange sorts
- D. Average time is quadratic.

Q.2. What is the worst-case time for quick sort to sort an array of n elements?

- A. $O(\log n)$
- B. $O(n)$
- C. $O(n \log n)$
- D. $O(n^2)$

Q.3. When is insertion sort a good choice for sorting an array?

- A. Each component of the array requires a large amount of memory.
- B. Each component of the array requires a small amount of memory.
- C. The array has only a few items out of place.
- D. The processor speed is fast.

Q.4. How does a quick sort perform on :

- A. An array that is already sorted
.....
- B. An array that is sorted in reversed order

Q.5 The Bubble sort, the Selection sort, and the Insertion sort are all $O(n^2)$ algorithms. Which is the fastest and which is the slowest among them?

.....

1.10 Answer to Check Your Progress

Ans. to Q. No. 1: C

Ans. to Q. No. 2: D

Ans. to Q. No. 3: C

Ans. to Q. No. 4: The quick sort is quite sensitive to input. The Quick Sort degrades into an $O(n^2)$ algorithm in the special cases where the array is initially sorted in ascending or descending order. This is because if we consider that the pivot element will be the first element.

So, here it produces only 1 sub list which is on the right side of first element and starts from second element. Similarly, other sub lists will be created only at right side. The number of comparison for first element is n ; second element requires $n-1$ comparison and so on. Thus, we will get the complexity of order n^2 instead $\log n$.

Ans. to Q. No. 5: The Bubble Sort is slower than the Selection Sort, and the Insertion Sort (in most cases) is a little faster.

1.11 Model Questions

1. Write a procedure for the merge procedure Merge1 (a, i, j, k).
2. Write non-recursive procedure for merge sort.
3. Modify merge sort procedure to sort number in non-ascending order.
4. For what input, merge sort takes maximum number of comparisons.
5. Implement merge sort algorithm in c.
6. Trace the Quick sort algorithm on sorted array of elements 1 to 10.
7. Trace the Quick sort algorithm on elements 10, 9, 8, 1.
8. Write a program for quick sort method.
9. Implement the radix sort.
10. Design an algorithm to sort given names using radix sort.
11. Explain the different types of sorting methods.
12. What is sorting? Write a function for Bubble Sort.
13. Write a program for sorting a given list by using Insertion Sort :
14. 2, 32, 45, 67, 89, 4, 3, 8, 10
15. Write a program to sort the following members in ascending order using Selection Sort :

16. 12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

UNIT VI: SEARCHING

1.1 Learning Objectives

1.2 Introduction

1.3 Searching

1.3.1 Linear Search

1.3.2 Binary Search

1.4 Performance and Complexity using Big 'O' notation

1.5 Answers to Check Your Progress

1.6 Model Questions

Learning Objectives

After going through this unit, the learner will be able to:

- learn about searching techniques.
- describe linear search and binary search.
- search an element in sorted and unsorted list of elements.

- learn about performance and complexity of search technique using Big 'O' notation.

1.2 Introduction

In our day-to-day life there are various applications, in which the process of searching is to be carried. Searching the name of a person from the given list, searching a specific card from the set of cards, etc., are some examples of searching. Searching is an important function in computer science. Many advanced algorithms and data structures have been devised for the sole purpose of making searches more efficient. As the data sets become larger and larger, good search algorithms will become more important. Searching methods are designed to take advantage of the file (that consists of many records/data) organisation and optimise the search for a particular record or to establish its absence.

This unit will focus on searching for data stored in a linear data structure such as an array or linked list.

1.3 Searching

Searching is a technique of finding an element from the given data list or set of the elements like an array etc. For example, let us consider an array of 20 elements. These data elements are stored in successive memory locations. We need to search an element from the array. In the searching operation, assume a particular element e to be searched. The element e is compared with all the elements in a list starting from the first element of an array till the last element. In other words, the process of searching is continued till the element is found or list is completely exhausted. When the exact match is found then the search process is successfully terminated.

In case, no such element exists in the array, the process of searching should be abandoned. Suppose, our element to be searched is 8. In that case if the given element is exist in the set of elements or array, then the search process is said to be successful as per the below Fig (a). It indicates that the given element belongs to the array.

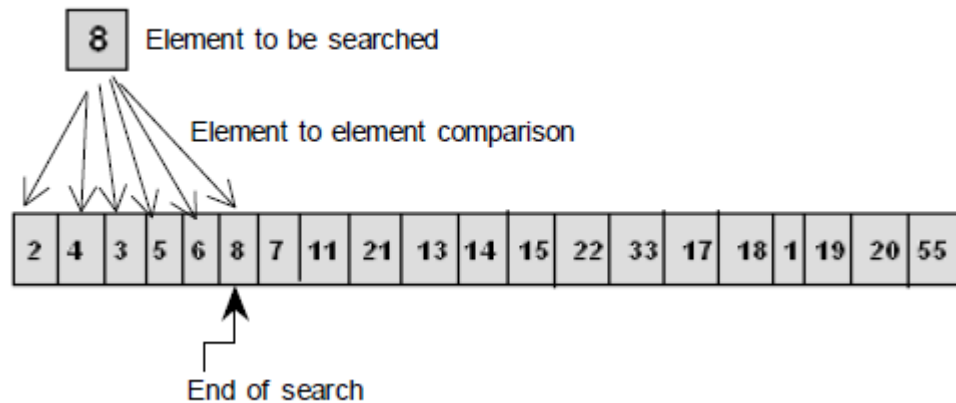


Fig: (a) Successful search

Again, let us consider our element to be searched is 66. The search is said to be unsuccessful if the given element does not exist in the array as per given Fig (b). It means that the given element does not belong to the array or collection of the items.

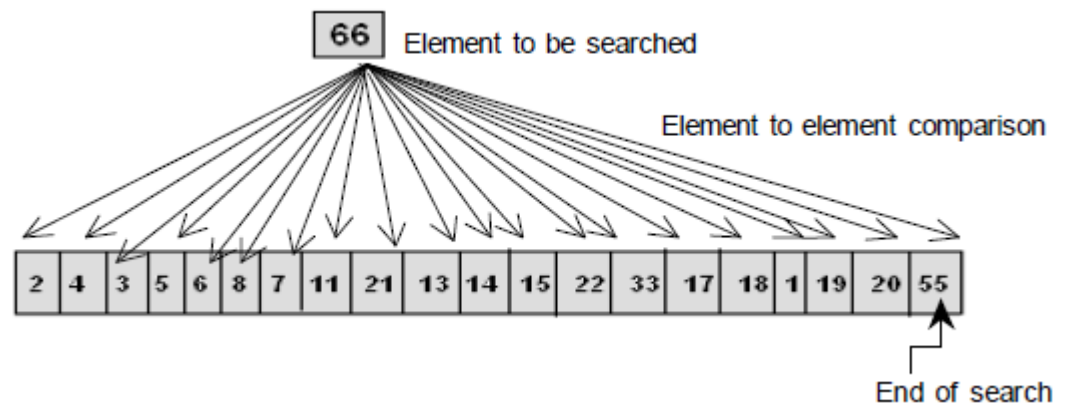


Fig. (b) : Unsuccessful search

We will now discuss two searching techniques:

- Linear or Sequential search
- Binary search

1.3.1 Linear Search

Linear search, also known as sequential search, means starting at the beginning of the data and checking each item in turn until either the desired item is found or the end of the data is reached.

This is the most natural searching method. Though, it is simple and straightforward, it has some limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in systematic manner. The linear search can be applied on sorted or unsorted linear data structure.

Algorithm of linear search: Let us start with an array or list, L which may have the item in question.

Step 1: If the list L is empty, then the list has nothing. The list does not have the item in question. Stop here.

Step 2: Otherwise, look at all the elements in the list L.

Step 3: For each element: If the element equals the item in question, the list contains the item in question. Stop here. Otherwise, go onto the next element.

Step 4: The list does not have the item in question.

```
/* Program: Program to search an element from the entered numbers (a
program for linear searching) */
```

```
# include<stdio.h>
```

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int arr[20],n,i,item;
```

```
printf("How many elements you want to enter in the array : ");
```

```
scanf("%d",&n);
```

```
for(i=0; i < n;i++)
```

```
{
```

```
printf("\nEnter element% d : ",i+1);
```

```
scanf("%d", &arr[i]);
```

```
}
```

```
printf("\nEnter the element to be searched : ");
```

```
scanf("%d",&item);
```

```
for(i=0;i < n;i++)
```

```
{
```

```
if(item == arr[i])
```

```
{
```

```
printf("\n% d found at position% d\n",item,i+1);
```

```

break;
}
}
if(i == n)
printf("\nItem%d not found in array\n",item);
getch();
}

```

In the program above, suppose the user enters 10 numbers i.e., n=10 using the first for loop. The element which is to be searched is stored in the variable item. By using a second for loop the element is compared with each element of the array. If the element in item variable is matched with any of the elements in the array then the location is displayed. Otherwise the item is not present in the array.

Analysis of Linear Search: We have carried out linear search on lists implemented as arrays. Whether the linear search is carried out on lists implemented as arrays or linked list or on files, the criteria part in performance is the comparison loops (i.e., Step 3 of the algorithm). Obviously, the fewer the number of comparisons, the sooner the algorithm will terminate.

The lowest possible comparison is equal to 1 when the required item is the first item in the list. The maximum comparisons will be equal to N (total numbers of elements in the list) when the required item is the last in the list. Thus, if the required item is in position i in the list, then only i comparisons are required. Hence the average number of comparisons done by linear (or sequential) search will be:

$$\begin{aligned}
& \frac{1+2+3+\dots+i+\dots+N}{N} \\
&= \frac{N(N+1)}{2 \times N} \\
&= \frac{N+1}{2}
\end{aligned}$$

Advantages and Disadvantages of Linear Search: Linear search algorithm is easy to write and efficient for short lists. It does not require sorted data.

However, it is lengthy and time consuming for long lists. There is no way of quickly establishing that the required item is not in the list or of finding all occurrences of a required item at one place. The linear search situation will be in the worst case if the element is at the end of the list.

CHECK YOUR PROGRESS

Q.1. Select the appropriate option for each of the following questions :

- i) Linear search is efficient in case of
 - A. short list of data
 - B. long list of data
 - C. both a) and b)
 - D. none of these

- ii) The process of finding a particular record is called
 - A. indexing
 - B. searching
 - C. sorting
 - D. none of these

Q.2. What are the advantages of sequential search?

.....
.....

1.3.2 Binary Search

The binary search approach is different from the linear search. Let us consider a list in ascending sorted order. It would work to search from the beginning until an item is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine in which half the item is in (because the list is sorted). This effectively reduces the working range into half of the original list with a single test. By repeating the procedure, the result is a highly efficient search algorithm called binary search.

The binary search is based on the divide-and-conquer approach. We compare the element with middle element of the array or list. If it is less than the middle element then we search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the list. Now, we will take that portion only for search and compare it with the middle element of that portion. This process will be in iteration until we find the element or the middle element has no left or right portion to search.

Let us take a sorted array of 11 elements and they are shown in Fig: (a). Let us assume that we want to search the element 56 from the list of elements.

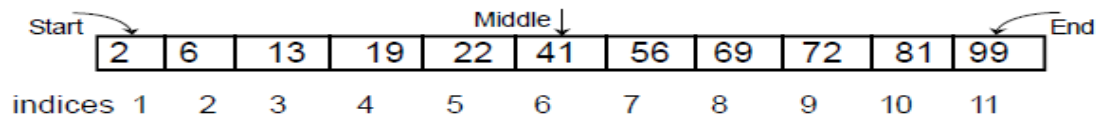


Fig: (a) Sorted array

For this, we will take 3 variables start, end and middle, which will keep track of the status of start, end and middle value of the portion of the array, in which we will search the element. The value of the middle will be:

$$\text{middle} = \frac{\text{start} + \text{end}}{2}$$

Initially, start = 1, end = 11 and the middle = (1+11) / 2 = 6.

The value at index 6 is 41 and it is smaller than the target value i.e (56). The steps are as follows:

Step 1: The element 2 and 99 are at start and end positions respectively.

Step 2: Calculate the middle index which is as

$$\text{middle} = (\text{start} + \text{end})/2$$

$$\text{middle} = (1+11)/2$$

$$\text{middle} = 6$$

Step 3: The key element 56 is to be compared with the middle value. If the key is less than the value at middle then the

key element is present in the first half else in other half; in our case, the key is on the right half. Now we have to search only on right half.

Hence, now start = middle + 1 = 6+1 = 7.

end will be same as earlier.

Step 4: Calculate the middle index of the second half

$$\text{middle} = (\text{start} + \text{end}) / 2$$

$$\text{middle} = (7+11) / 2$$

$$\text{middle} = 9$$

Step 5: Again, the middle divides the second half into the two parts, which is shown in fig: (b).

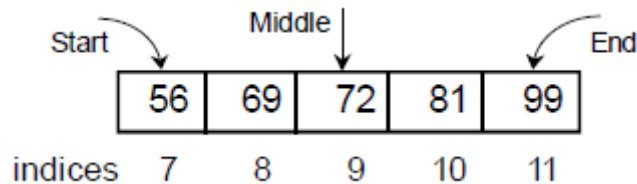


Fig: (b)

Step 6: The key element 56 is less than the value at middle which is 72, hence it is present in the left half i.e., towards the left of 72. Hence now, end = middle - 1 = 8. start will remain 7.

Step 7: At last, the middle is calculated as middle = (Start + End) / 2

$$\text{middle} = (7+8) / 2$$

$$\text{middle} = 7$$

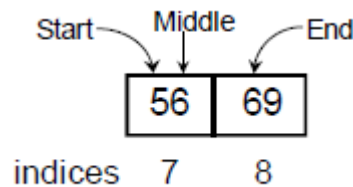


Fig: (c)

Step 8: Now, if we compare our key element 56 with the value at middle, then we see that they are equal. The key element is searched successfully and it is in 7th location.

//Program: Program to search an element using binary search

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
void main()
```

```
{
```

```
int arr[20],start,end,middle,n,i,item;
```

```
clrscr();
```

```
printf("How many elements you want to enter in the array : ");
```

```

scanf("%d",&n);
for(i=1;i<=n;i++) {
printf("Enter element%d : ",i);
scanf("%d",&arr[i]);
}
printf("\nEnter the element to be searched : ");
scanf("%d",&item);
start =1;
end = n;
middle = (start +end) / 2;
while(item!=arr[middle] && start<=end)
{
if(item>arr[middle])
start = middle+1;
else
end = middle -1;
middle = (start + end) / 2;
}
if(item==arr[middle])
printf("\n %d is found at position %d\n", item, middle);
if(start>end)
printf("\n%d is not found in array\n", item);
getch();
}

```

In Binary search, by dividing the working data set in half with each comparison, logarithmic performance, $O(\log n)$, is achieved.

/* Program: Write a program to search a name from a list of 10 names using binary search*/

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int start=1,end=10,mid,i,flag=0,value;

```

```

char name[15][10],nm[15];
clrscr();
printf("\nEnter 10 names:\n");
for(i=1;i<11;i++)
scanf("%s",&name[i]);
printf("\nEnter the name to search: ");
scanf("%s", &nm);
mid=(start+end)/2;
while(strcmp(nm,name[mid])!=0 && start<=end)
{
value=strcmp(nm,name[mid]);
if(value>0)
{
start=mid+1;
mid=(start+end)/2;
}
else
{
end=mid-1;
mid=(start+end)/2;
}
}
if(strcmp(nm,name[mid])==0)
{
flag=1;
}
if(flag==1)
printf("\nThe name %s is found successfully",nm);
else
printf("\nThe name %s is not found",nm);
getch();
}

```

1.4 Performance and Complexity

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. The ability to analyze a piece of code or an algorithm and understand its efficiency is vital for understanding computer science.

Suppose our problem is to input a name (key) and a list of names, and to determine whether the name occurs in the list and if so where does it occur.

For example, if a list contains 6 names i.e., $N=6$ and we are using linear search technique, then

	<i>Compares to find</i>
<i>Rahul</i>	1
<i>Joy</i>	2
<i>Rita</i>	3
<i>John</i>	4
<i>Kiran</i>	5
<i>Krishna</i>	6

Worst case: 6 compares.

Best case: 1 compare.

Average case: Expect to compare key with half of list

In linear (sequential) search Comparison is the key operation.

Number of compares depends on data (position of the element to be searched and the number of elements). If the list or array size is N :

Best case: 1 compare (if the sequential search finds the name at the first position of the list then it will behave like best case).

Worst case: Compare key with all names in list. The sequential search situation will be in the worst case if the name is at the end of the list,

i.e., the number of comparison needed is N .

Average case: Key may match with name in any position. Expect to compare with half of list.

Again, for example, if we are to determine a target value (key) that occurs in a list using binary search, then the algorithm will be as follows

Binary Search Algorithm

Input : (List, Key)

WHILE (List is not empty) DO

(Select “middle” entry in list as test entry

IF (Key = test entry) THEN (Output(“Found it”) Stop)

IF (Key < test entry) THEN (apply Binary Search to part of list preceding test entry)

IF (Key > test entry) THEN (apply Binary Search to part of list following test entry)

Output(“Not Found”)

Binary Search required a sorted array. Given that the array is of size n , split the array into half—test the middle element, originally $n/2$; if it is the value you are searching for, return the current index. Otherwise, if you have only one element in your sub array, return a sentinel indicating that you cannot find the value. If that is not the case, if your search key is less than the middle element, perform the same operation on the elements from the start to $n/2 - 1$; if the middle element is less than your search key, perform the search on the sub array of $n/2 + 1$ to n . Every time you do this, you cut the size of the array to search in half — so it takes $O(\log n)$ time with base 2.

Binary search is worst case $O(\log n)$ where n is the size of the list and \log is with base 2. Thus if $n = 8$, then $\text{Log } 8 = \text{Log } 2^3 = 3 \text{ Log } 2 = 3$.

CHECK YOUR PROGRESS

Q.3. State whether the following statements are true(T) or false(F)

- i) Element should be in sorted order in case of binary search.
- ii) Binary search cannot be applied in character array.
- iii) Sequential search is worst case and average case $O(n)$.
- iv) Binary search is worst case $O(\log n)$.

Q. 4. In the following function code which type of search algorithm is applied?

```
int find ( int a[ ], int n, int x )
{
    int i;
    for ( i = 0; i < n; i++ )
    {
        if ( a[i] == x )
            return i;
    }
    return 0;
}
```

Q.5. Write a function in C to find an element x in an array of n elements where the array size of the array and the element is passed as arguments.

1.5 Answer to Check Your Progress

Ans. to Q. No. 1: i) A, ii) B

Ans. to Q. No. 2: Sequential search is easy to implement and relatively efficient to use for small lists. It does not require a sorted list of elements or data.

Ans. to Q. No. 3: i) True; ii) False; iii) True; iv) True

Ans. to Q. No. 4: Sequential (Linear) search is applied.

Ans. to Q. No. 5: int find (int a[], int n, int x)

```
{
int i = 0;
while ( i < n )
{
int mid = ( n + i ) / 2;
if ( a[mid] < x )
n = mid;
else if ( a[mid] > x )
```

```
i = mid + 1;
else
return mid;
}
return 0;
}
```

1.6 Model Questions

1. Write a program to find the given number in an array of 20 elements. Also display how many times a given number exists in the list.
2. What are the advantages and disadvantages of sequential search technique?
3. What are the precondition of binary search technique?
4. Differentiate between linear and binary search.
5. Explain linear and binary search.
6. Write a program to demonstrate binary search. Use integer array and store 10 elements. Find the given element.
7. Write down the best, worst and average case time complexity of sequential search.

UNIT VII: GRAPHS I: REPRESENTATION AND TRAVERSAL

1.1 Learning Objectives

1.2 Introduction

1.3 Graph

1.4 Terminologies of Graph

1.5 Different Types of Graph

1.6 Representation of Graph

- 1.6.1 Sequential Representation of Graph
- 1.6.2 Linked Representation of Graph
- 1.7 Traversal in Graphs
 - 1.7.1 Depth First Search
 - 1.7.2 Breadth First Search
- 1.8 Königsberg Bridge Problem
- 1.9 Answer to Check Your Progress
- 1.10 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to:

- learn about the basic definition of graph
- describe the different types of graph
- describe the various representations of graph
- distinguish between depth first search and breadth first search

1.2 Introduction

Like tree, graphs are also nonlinear data structure. Tree has some specific structure whereas graph does not have a specific structure. It varies From application to application in our daily life. Graphs are frequently used in every walk of life. A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. For example, a graph that contains the cities of India is connected by means of roads. We can assume that the graph is the interconnection of cities by roads. If we provide our office or home address to someone by drawing the roads, shops etc. in a piece of paper for easy representation, then that will also be a good example of graph.

In this unit, we are going to discuss the representation of graph in memory and present various graph traversal techniques like depth first search and breadth first search.

1.3 Graph

A graph G is a collection of two sets V and E where V is the collection of vertices v_0, v_1, \dots, v_{n-1} and E is the collection of edges e_1, e_2, \dots, e_n where an edge is an arc which connects two vertices. The vertices are also termed as *nodes* and edges are termed as *arcs*. This can be represented as

$$G = (V, E)$$

where, $V(G) = \{v_0, v_1, v_2, \dots, v_{n-1}\}$ and $E(G) = \{e_1, e_2, e_3, \dots, e_n\}$.

For example let us consider the graph $G = (V, E)$ shown in Fig where $V(G) = \{A, B, C, D, E\}$ and $E(G) = \{(A, C), (C, B), (B, D), (D, A), (C, E)\}$

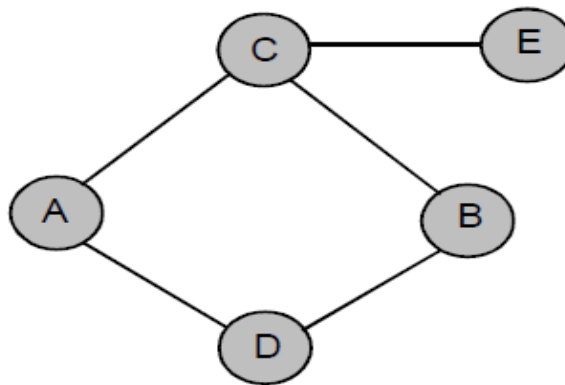


Fig.: A Graph

1.4 Terminologies of Graph

Weighted Graph: A graph is said to be weighted if there are some non negative values assigned to each edge of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.

A weighted graph is shown in figure.

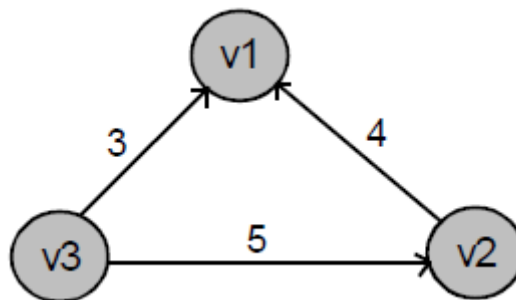


Fig.: A weighted graph

Adjacent nodes: When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence: In an undirected graph the edge between v_1 and v_2 is incident on node v_1 and v_2 .

Walk: A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk: A walk which is to begin and end at the same vertex is called closed walk. Otherwise it is an open walk.

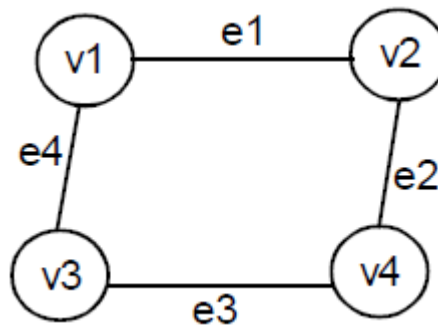


Fig.: Closed walk present in Graph

If e_1 , e_2 , e_3 and e_4 be the edges of pair of vertices (v_1, v_2) , (v_2, v_4) , (v_4, v_3) and (v_3, v_1) respectively, then $v_1 e_1 v_2 e_2 v_4 e_3 v_3 e_4 v_1$ can be its closed walk or circuit.

Path: An open walk in which no vertex appears more than once is called a path.

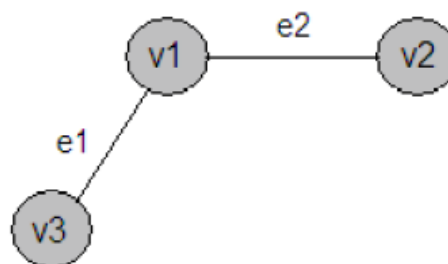


Fig.: A path

If e_1 and e_2 be the two edges between the pair of vertices (v_1, v_3) and (v_1, v_2) respectively, then $v_3 e_1 v_1 e_2 v_2$ is its path.

Length of a path: The number of edges in a path is called the length of that path. In the following diagram the length of the path is 3.

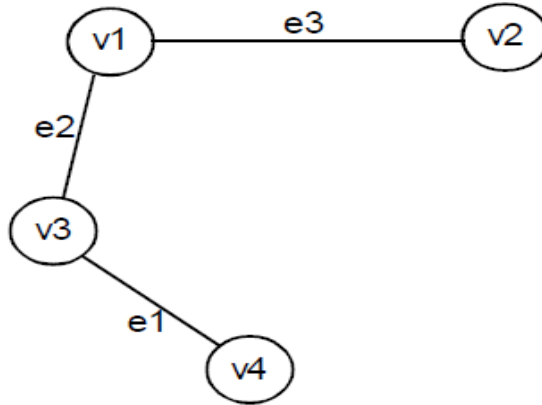


Fig. : An open walk graph

Circuit: A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

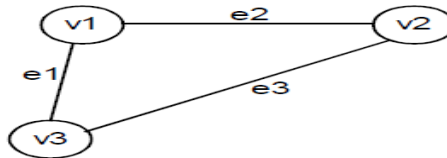


Fig. : A circuit having three vertices and three edges

Sub Graph: A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G , and each edge of S has the same end vertices in S as in G .

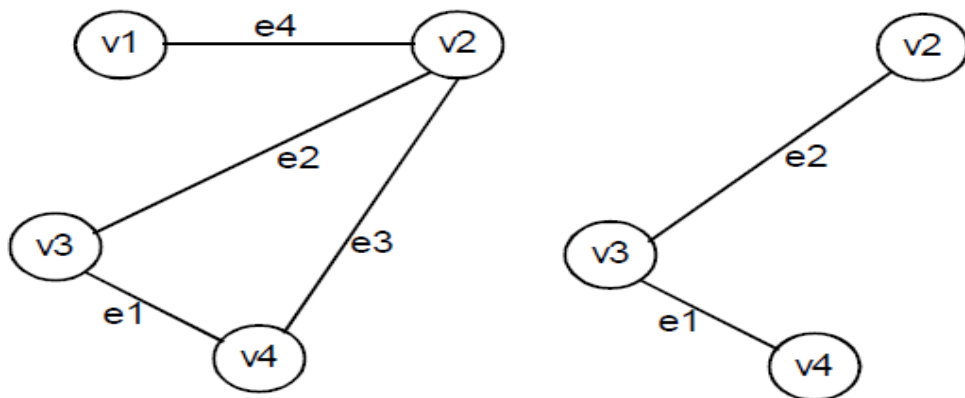


Fig.: Graph G

Graph S

In the above figure, S is a sub graph of the graph G , since all the vertices and edges in the graph S are also the vertices and edges of G .

Connected Graph: A graph G is said to be connected if there is at least one path between every pair of vertices in G (Fig). Otherwise, the graph is disconnected.

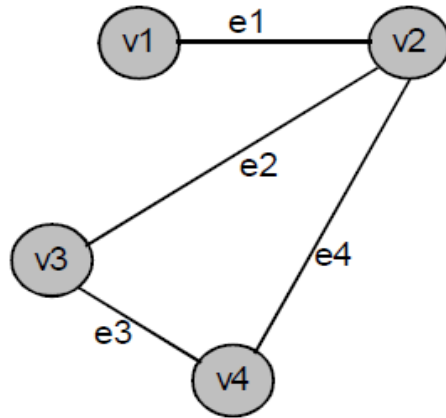


Fig.(a): A connected graph G

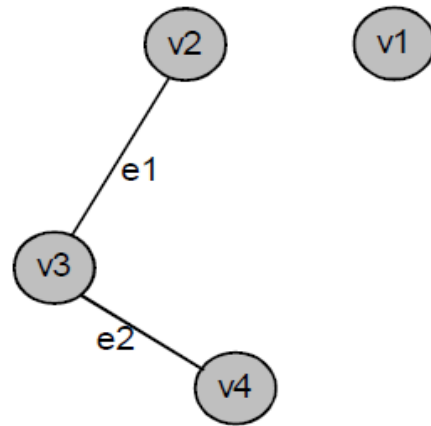


Fig.(b): A disconnected graph G

The graph in fig (b) is disconnected because the vertex v_1 is not connected with the other vertices of the graph.

Degree: In an undirected graph, the number of edges connected to a node is called the degree of that node or we can say that the degree of a node is the number of edges incident on it. In the graph shown in fig. (a), degree of vertex v_1 is 1, degree of vertex v_2 is 3, degree of v_3 and v_4 is 2.

Degree of the vertex v_1 of the graph shown in fig. (b) is zero.

Indegree: The indegree of a node is the number of edges connected to that node or, in other words, edges incident to it.

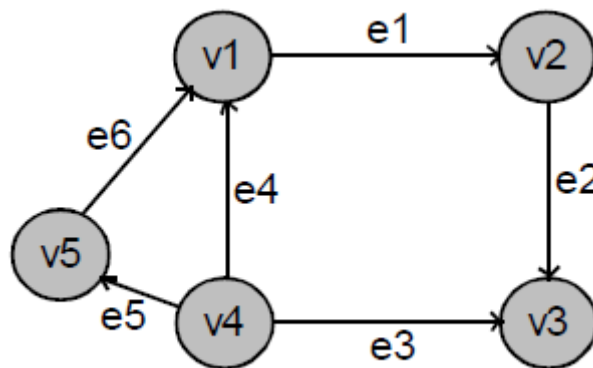


Fig. (c) A graph

In the figure (c), indegree of vertices v_1, v_3 is 2, indegree of vertices v_2, v_5 is 1 and indegree of v_4 is zero.

Outdegree: The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the edges incident from it. In the

graph shown in figure (c), the outdegree of vertex v_4 is 3, outdegree of vertex v_1 is 1.

1.5 Different Types of Graph

There are different types of graphs. Some of them are:

1. Digraph
2. Undirected graph
3. Complete graph
4. Regular graph
5. Cycle graph
6. Acyclic graph
7. Multigraph
8. Euler graph

Let us look at some of the different types of graph in detail.

Directed Graph (Digraph): A directed graph or digraph G consists of a vertex set $V(G)$ and an edge set $E(G)$, where each edge is an ordered pair of vertices. A simple digraph is a digraph in which each ordered pair of vertices occurs at the most once as an edge. In this type of graph each edge has direction, meaning, (v_1, v_2) and (v_2, v_1) will represent different edges.

In the figure (d), $V(G) = \{A, B, C, D\}$, $E(G) = \{(A, B), (B, D), (D, A), (A, C), (C, D)\}$. The graph has 4 vertices and 6 edges.

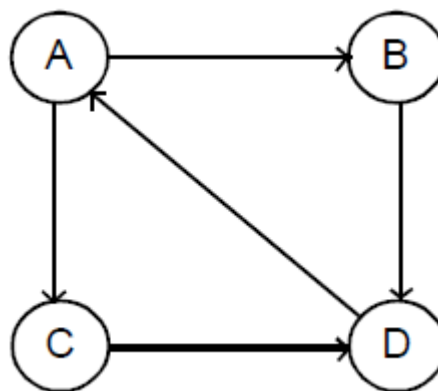


Fig. (d) : Directed Graph

Undirected Graph: A graph containing unordered pair of nodes is termed as undirected graph. If there is an edge between vertex v_0 and v_1 then it can be represented as (v_0, v_1) or (v_1, v_0) .

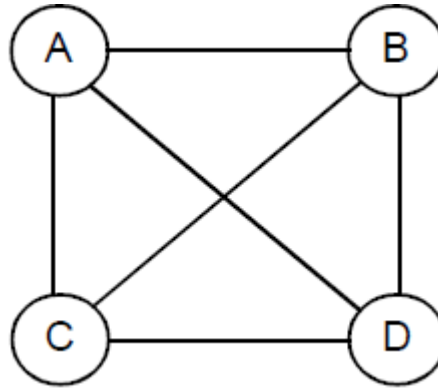


Fig. (e) Undirected Graph

In the graph shown in fig.(e), $V(G) = \{A, B, C, D\}$ and $E(G) = \{(A, B), (A, C), (A, D), (B, C), (B, D), (D, C)\}$. The graph has 4 vertices and 6 edges.

Complete Graph: A graph in which any node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains total edges equal to $= n(n-1)/2$ where n is the number of vertices present in the graph. The figure (f) shows a complete graph.

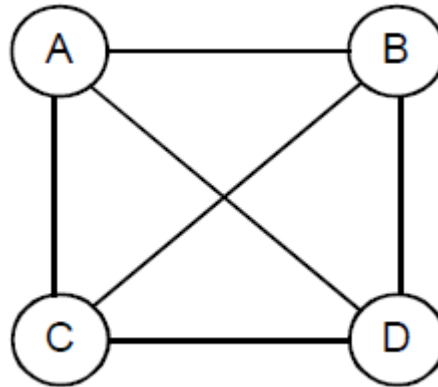


Fig.(f): A complete graph

Regular Graph: Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

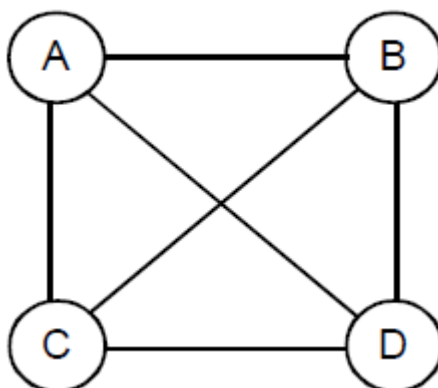


Fig.(g): A regular graph

Cycle Graph: A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.

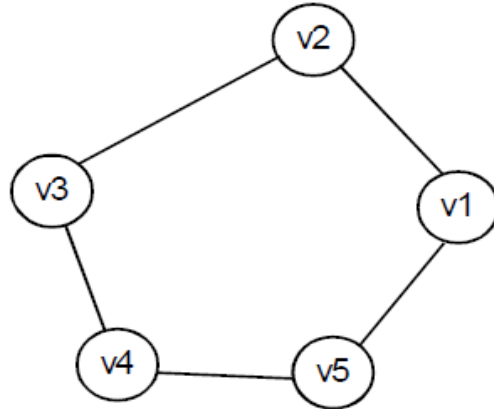


Fig. (h): A cycle graph

Acyclic Graph: A graph without cycle is called acyclic graph.

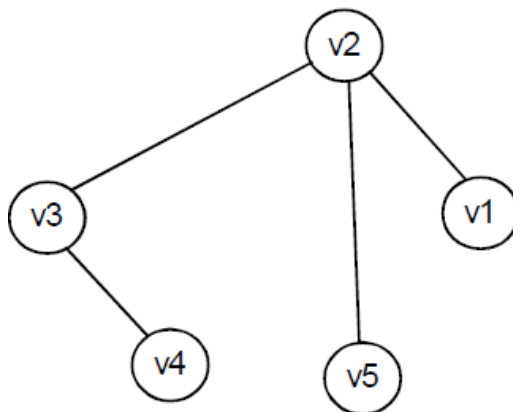


Fig. (i): A acycle graph

Multigraph: A graph in which more than one edge is used to join the vertices is called multigraph.

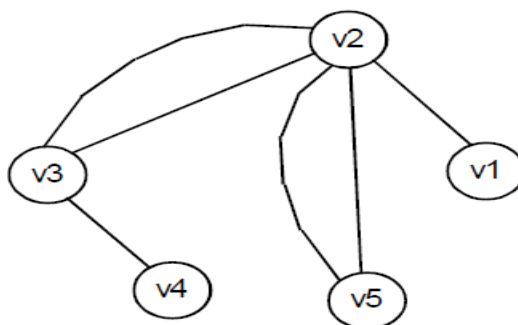


Fig.(j): A multigraph

Euler Graph: If some closed walk in a graph contains all the edges of the graph, then the walk is called an Euler line and the graph is called an Euler graph.

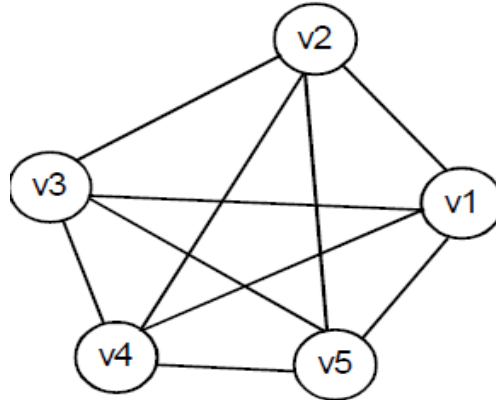


Fig. (k): An Euler graph

1.6 Representation of Graph

There are two standard ways of storing graph G in the memory of a computer. One way, called the sequential representation of G, by means of its adjacency matrix A. The other way called the linked representation of G, is by means of linked list.

1.6.1 Sequential Representation of Graph

Adjacency Matrix : Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, \dots, \dots, v_m$. Then the adjacency matrix $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:

$$a_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is adjacent to } v_j, \text{ that is, if there is an edge } (v_i, v_j) \\ 0, & \text{otherwise.} \end{cases}$$

Such a matrix A, which contains entries of only 0 and 1, is called a bit matrix or a Boolean matrix. For example: for the graph G in the following figure (l), we have the adjacency matrix A.

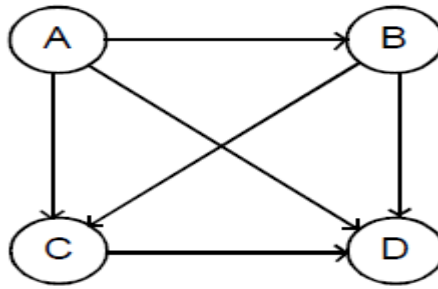


Fig. (l): Directed Graph G

The equivalent adjacency matrix A is:

	A	B	C	D
A	0	1	1	1
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

1.6.2 Linked Representation of Graph

In this type of representation a graph G is usually represented in memory by a linked representation. It is also called an adjacency list. Let us consider a graph G as shown below:

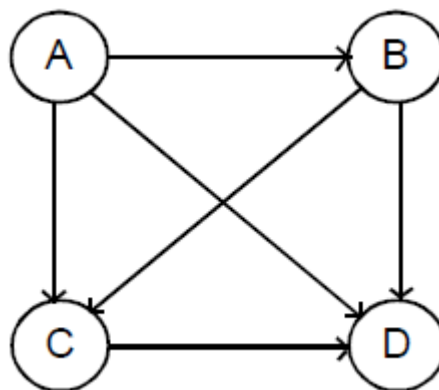


Fig.(m): Directed Graph G

The graph G can be represented by its adjacency list, which is its list of adjacent nodes, also called its successors or neighbors

Nodes	Adjacency List
A	B, C, D
B	C, D
C	D
D	

Fig. (n): Adjacency list of graph G.

CHECK YOUR PROGRESS

Q.A. Select the appropriate option:

1. Graph can be implemented using:

i) arrays; ii) linked list; iii) stack; iv) queue

a) i), ii) and iv)

b) i), ii) and iii)

c) ii) and iii)

d) i) and ii)

2. Adjacency matrix for a digraph is

a) unit matrix

b) anti symmetric

c) symmetrix matrix

d) none of these

3. A walk which is to begin and end at the same vertex is called

a) close walk

b) open walk

c) path

d) none of these

1.7 Traversal in Graphs

When dealing with graphs, one fundamental issue is to traverse the graph. In other words, to visit each vertex. To solve this problem, many algorithms exist. Two of them will be presented here: Depth First Search (DFS) and Breadth First Search (BFS).

1.7.1 Depth First Search

In this method, a node from graph is taken as a starting node. Traverse through the possible paths of the starting node. When the last node of the graph is obtained and path is not available from the node, then control returns to previous node. This process is implemented using stack data structure. Let us consider the following graph G:

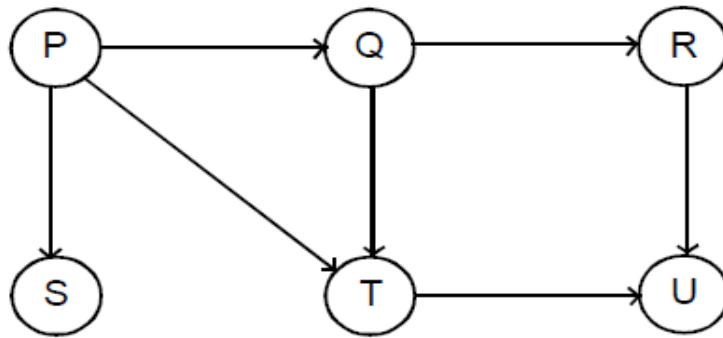


Fig.(o): A model graph G

Let us apply Depth First Search traversal method, and consider P as starting node. Then, we traverse the node adjacent to P and we will get Q and then R (adjacent to Q) and U (adjacent to R). The traversal will be P-> Q-> R-> U.

The search is always carried in forward direction. After reaching to U, we reach the end of the path and further movement in forward direction is not possible. Hence, the control goes to the previous node and again traverses through the available paths for

non-traversed nodes. We get the node R and it has no non-traversed node. Hence we go back to Q and it gives T. The node T gives U, but it is already visited. Therefore, control in reverse direction checks all the nodes. Then it goes back to P which gives node S. Hence the sequence of traversal will be P->Q->R->U->T->S.

The general idea behind a depth-first search beginning at a starting node A is as follows.

First we take the starting node A. Then we examine each node N along a path P which begins at the node A. That means we process a neighbor node of the node A. Then again a neighbor of a neighbor and proceed and so on. After coming to a “dead end” which is to the end of the path P, we backtrack on P until we can continue along another path P and so on. The algorithm is similar to the Breadth-First Search except that here we take a stack instead of queue.

In the algorithm given below, a field STAT is used to tell the current status of a node. Algorithm of DFS : Algorithm for Depth-First Search on a graph G beginning at a starting node A

Step 1. Initialize all nodes to the ready state [STAT=1]

Step 2. Push the starting node A onto STACK and change its status to the waiting state [STAT=2]

Step 3. Repeat Step 4 and Step 5 until STACK is empty.

Step 4. Pop the top node N of STACK. Process N and change its status to the processed state [STAT=3]

Step 5. Push onto STACK all the neighbors of N that are still in the ready state [STAT=1] and change their status to the waiting state [STAT=2]
[End of Step 3 loop]

Step 6. Exit.

The above algorithm will process only those nodes which are reachable from the starting node A.

1.7.2 Breadth First Search

This is one of the popular methods of traversing graph. This method uses the queue data structure for traversing nodes of the graph. Any node of the graph can act as a beginning node. Using any node as starting node, all other nodes of the graph are traversed.

This process is repeated till unvisited nodes are available.

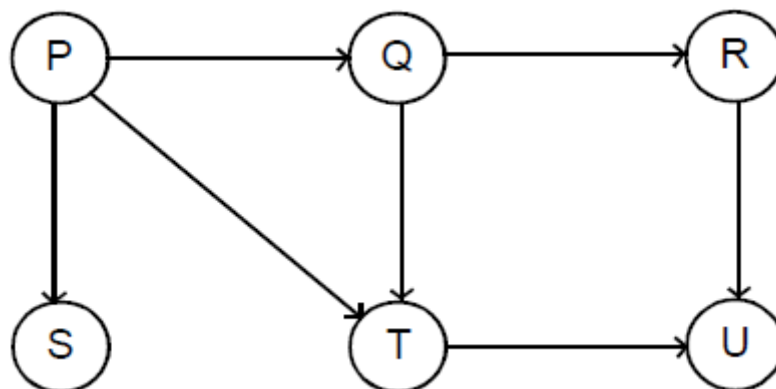


Fig.(p): A model graph G

Let us apply the Breadth First Search traversal method in the above graph G. Take the Pas a starting node and all the adjacent nodes of P are traversed, that

is Q, T and S. The traversal of nodes can be carried in any sequence. For example, the sequence of traverse of nodes is Q,S and T. The traversal will be P->Q->S->T.

First, all the nodes neighbouring Q are traversed, then neighbouring nodes of S and finally T are taken into account. The adjacent node of Q is R and T is U. Similarly, the adjacent node of T is U and S does not have any adjacent node. Hence, in this step the traversal now should be in the following way: P->Q->S->T->R->U.

The general idea behind a breadth-first search beginning at a starting node A is as follows :

First, we examine the starting node A. Then we verify all the neighbors of A. Then we examine all the neighbors of the neighbors of A and proceed so on. Here actually we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is done by using queue to hold the nodes that are waiting to be processed, and by using a field STAT which keep track of the current status of any node. The algorithm is as given below:

Algorithm of Breadth First Search: Algorithm for a breadth-first

search on a graph G beginning at a starting node A Step 1. Initialize all the nodes to the ready state [STAT=1]

Step 2. Put the starting node A in QUEUE and change its status to the waiting state [STAT=2]

Step 3. Repeat Step 4 and Step 5 until QUEUE is empty.

Step 4. Remove the front node N of QUEUE. Process N and change the status of N to the processed state [STAT=3]

Step 5. Add to the rear of QUEUE all the neighbors of N that are in the steady state [STAT=1] and change their status to the waiting state [STAT=2] [End of Step3 loop]

Step 6. Exit.

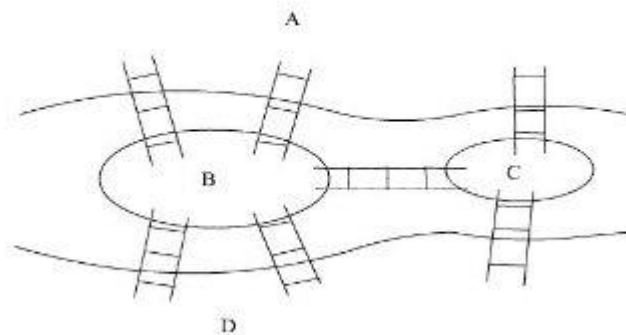
The above algorithm will process only those nodes which are reachable from the starting node A.

In an unweight graph, you can consider all edges to have an equal weight of 1. BFS visits all the vertices on the same level before proceeding to a “deeper”

level, so all vertices newly visited on a level will be of the same distance from the starting vertex. Further, since BFS proceeds level by level with distances of 0, 1, 2 etc, it will reach a vertex via the shortest path to that vertex.

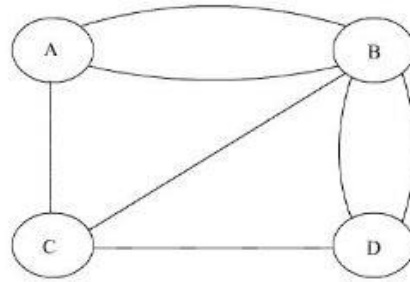
1.8 Königsberg Bridge Problem

- A Graph without loops and parallel edges is called a simple graph.
- A graphs with isolated vertices (no edges) is called null graph.
- Set of edges E can be empty for a graph but not set of vertices V.
- Graphs can be used in wide ranges of applications. For example consider the Königsberg Bridge Problem.
- Königsberg Bridge Problem: Two river islands B and C are formed by the Pregel river in Königsberg (then the capital of East Prussia, Now renamed Kaliningrad and in west Soviet Russia) were connected by seven bridges as shown in the figure below. Start from any land areas walk over each bridge exactly once and return to the starting point.



GRAPHS Representation:

- Euler (1707-1783) in 1736 formulated the Königsberg bridge problem as a graph problem and solved.
- Represent land area by vertices and bridges connecting them by edges. We get the following graph.



- The problem is nothing but a children's game of drawing a figure without lifting pen from the paper and without retracing a line.

CHECK YOUR PROGRESS

Q. B. Select the appropriate option :

1. Breadth First Search is a method to traverse

- all successors of a visited node before any successors of any of those successors
- a single path of the graph as far as it can go.
- the graph using shortest path
- none of these.

2. Identify the correct statements about DFS traversal

- It can be used to determine whether a graph is acyclic or not.
 - It identifies the connected component of an undirected graph.
 - Traverses a single path of the graph until it visits a node with no successor.
- i) and iii)
 - ii) and iii)
 - i) and ii)
 - i), ii) and iii)

3. A vertex of degree one is called ____

- pendant
- isolated vertex
- null vertex

- d) coloured vertex
4. A graph in which all nodes are of equal degree is known as
- a) multigraph
 - b) non regular graph
 - c) regular graph
 - d) complete graph
5. A simple graph with n vertices and k components can have at the most _____
- a) n edges
 - b) $n-k$ edges
 - c) $(n-k)(n-k-1)/2$ edges
 - d) $(n-k)(n-k+1)/2$ edges
6. If there exists at least one path between every pair of vertices in a graph, the graph is known as.
- a) complete graph
 - b) disconnected graph
 - c) connected graph
 - d) euler graph.

1.9 Answer to Check Your Progress

Ans. to Q. No. A: 1. d) i) and ii); 2. c) symmetrix matrix; 3. b) open walk.

Ans. to Q. No. B : 1. a); 2. d); 3. a); 4. c); 5. d); 6. c).

1.10 Model Questions

1. Distinguish between the directed and undirected graphs.
2. What are complete graphs?
3. Distinguish between Breadth First and Depth First search traversals in a graph.
4. How are graphs implemented in computer memory?
5. Distinguish between tree and graph.

6. Write short notes on: (a) Breadth First Search (BFS) and (b) Depth First Search (DFS).
7. Define degree, indegree and outdegree of graph. Explain with examples.

UNIT VIII: GRAPHS II: BASIC ALGORITHMS

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Basic Algorithms
- 1.4 Minimum Spanning Tree
- 1.5 Single Source Shortest Path
- 1.6 Answer to Check Your Progress
- 1.7 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will able to learn about:

- Basic of Algorithm
- Minimum Spanning Tree
- Single Source Shortest Path

1.2 Introduction

- Assume, in a country, there are many oil wells. It is necessary to connect them by a pipeline. Main objective of designing should be minimization of total cost of laying pipeline.
- This problem can be modeled as a graph problem. The set of oil wells are represented by vertices. The pipeline between any pair of oil wells is corresponding to the edge between the vertices denoting these oil wells.
- The cost of laying a pipeline between a pair of oil wells is the weight of the edge joining these oil wells.
- Consider the following example of five oil wells. The weight on the edges denotes the cost of the laying pipeline between their respective oil wells.

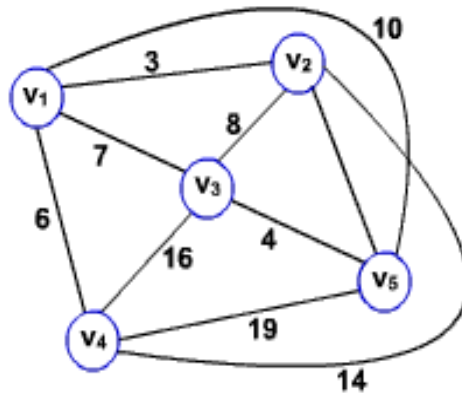


Fig. 1

- The best way of connecting them is shown in the following figure.

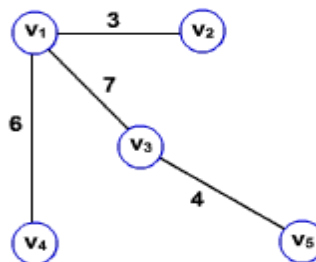


Fig. 2

- The cost of the laying the above pipe line connecting these oil wells is 20. Any other way of connecting increases the cost.
- Resultant graph is a tree which spans all vertices of the graph. Hence called spanning tree.

- The spanning tree with minimum cost is called Minimum Spanning tree (MST).

1.3 Minimum Spanning Tree

- Problem: Given a weighted undirected graph G , find the minimum spanning tree.
- One of the main property of a tree is cycle freeness.
- This property is exploited during construction of a MST for a given graph. That is, to obtain a spanning tree of a given graph, starts from a null graph add edges one after the other without forming cycles.
- If the edges considered for adding is in the increasing order we get a MST.
- Consider the following weighted graph.

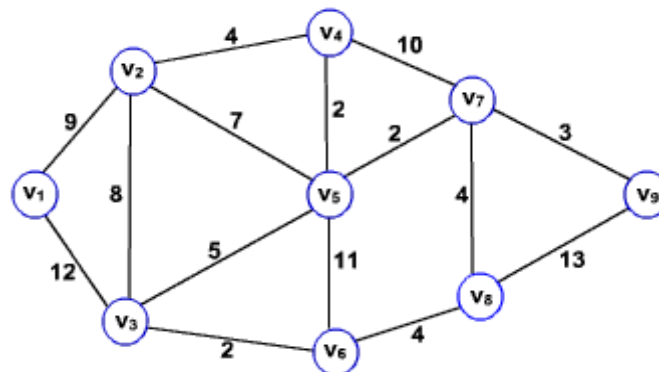


Fig.3

- In the above figure there are two least weighted edges. We can choose any one of them. Lets choose edge (v_4, v_5) .
- Next least weighted edge is between (v_3, v_6) , add this edge to MST edges, since it is not creating any cycle.
- Next edge is (v_7, v_9) . Repeat the processes of adding edges till we found $n-1$ edges without forming any cycle. The number of vertices's in the graph is n .
- This is method is Kruskal's algorithm, named after the inventor.
- The following sequence of figures shows a way of adding edges to obtain a MST.

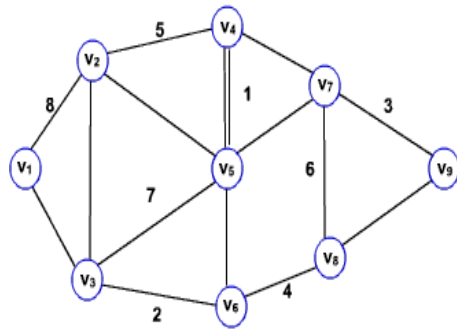


Fig. 4

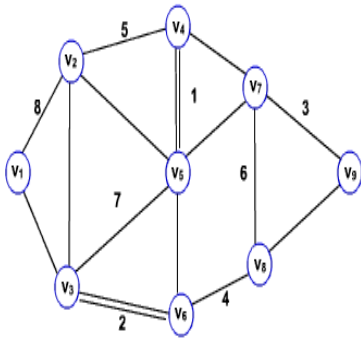


Fig. 5

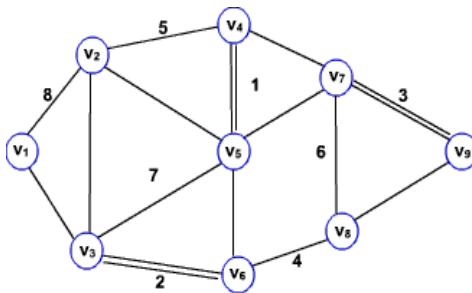


Fig:6

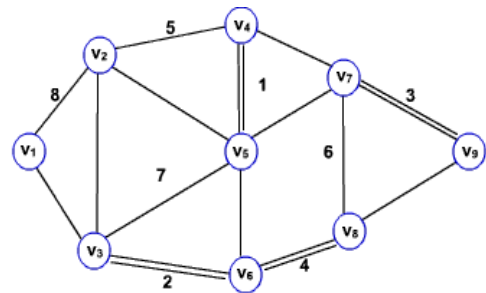


Fig:7

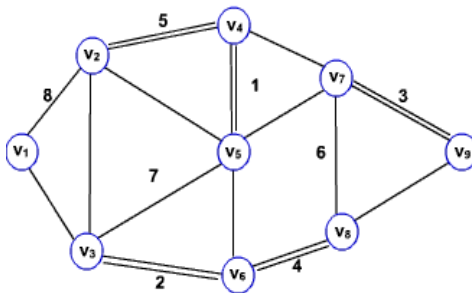


Fig : 8

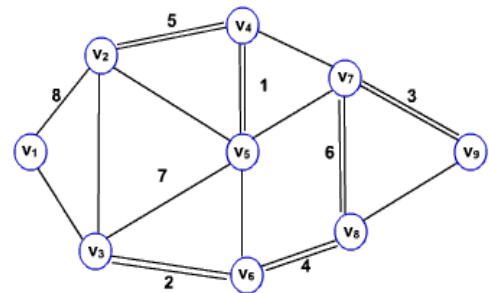


Fig : 9

- The weight of the MST is 33.

The above method is depicted in the algorithm below.

algorithm `Kruskal_MST(G)`

Step 1 : `Sort_E = Sort` the set E of edges by nondecreasing order

Step 2: `MST = NULL`

Step 3: for each edge (u, v) in E in the nondecreasing order do

Step 4: if $(MST + (u,v)$ does not contain cycle) then

Step 5: $MST = MST + (u,v)$

- This algorithm can also be viewed as follows: Initially, each vertex is component. Add an edge between a pair of components to make a bigger component without any cycle. The edge chosen is a least weighted. Repeatedly add edges one after the another till we have only one component.
- An implementation of the algorithm can be done using a set data structure. Each component is represented as a set. Add next least weighted edge (u,v) , if u and v are in two different components or sets. That is, make a bigger component by taking the union of the sets containing u and v .
- The Kurskal's algorithm using set data structure is given below.

Algorithm IMP_Kurskal_MST(G)

step 1: Sort_E= Sort the set E of edges by nondecreasing order.

step 2: For each vertex v in V do

Step 3: Make $set(v)$

Step 4: for each edge (u, v) in E in the nondencreasing order do

step 5: if $(u$ and v are in different components) then

step 6: Union (u, v)

1.5 Single Source Shortest Path

- A traveler wishes to find a shortest distance between New Delhi and Visakhapatnam, in a road map of India in which distance between pair of adjacent road intersections are marked in kilometers.
- One possible solution is to find all possible routes between New Delhi and Visakhapatnam, and find the distance of each route and take the shortest.
- There are many possible routes between given cities. It is difficult to enumerate all of them.
- Some of these routes are not worth considering. For example, route through Cinnai, since it is about 800KM out of the way.

- The problem can be solved efficiently, by modeling the given road map as a graph and find the shortest route between given pair of cities as suitable (single source shortest paths) graph problem.
- **Graph Representation of the road map:** The intersections between the roads are denoted as vertices and road joining them as edge in the graph representation of the given road map.
- **Single Source Shortest Path problem:** Given a graph $G = (V, E)$, we want to find the shortest path from given source to every vertex in the graph.
- The problem is solved efficiently by Dijkstra using greedy strategy. The algorithm is known as Dijkstra's algorithm.
- The method works by maintaining a set S of vertices for which shortest path from source v_0 is found. Initially, S is empty and shortest distance to each vertex from source is infinity.
- The algorithm repeatedly selects a vertex v_i in $V - S$ to which path from source is shorter than other vertices in $V - S$, adds to S , and estimates the shortest distance to other vertices from v_i .

Algorithm Dijkstra_shortest_path(G, v_0)

Step 1: for each vertex v_i in V do

Step 2: $dist[v_i] = \text{infinity}$

Step 3: $distance[v_0] = 0;$

Step 4: for each vertex v_i in V do

Step 5: $insert(Q, v_i)$

Step 6: $S = \text{empty}$

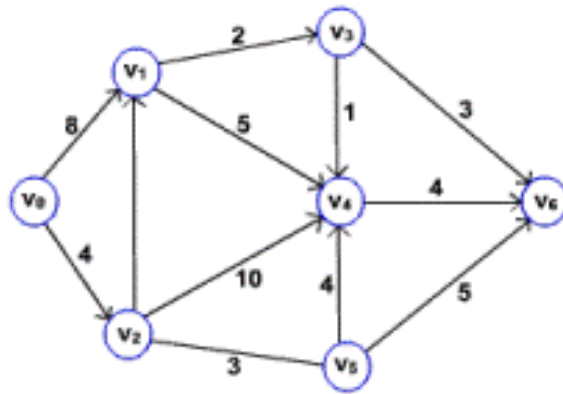
Step 7: while (Q not empty) do

Step 8: $v_i = \text{Min}(Q)$

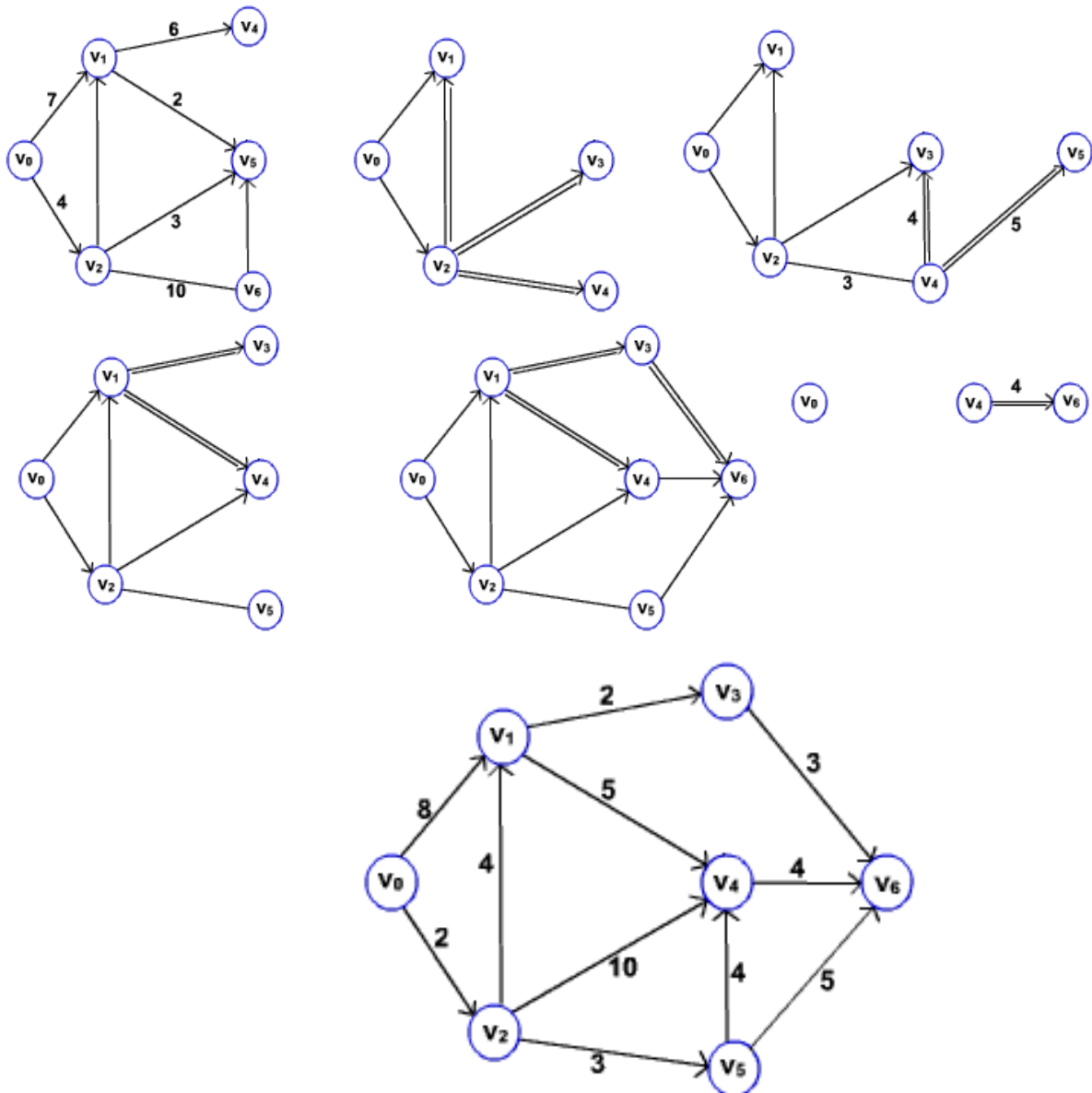
Step 9: for each vertex v_j adjacent to v_i do

Step 10: $distance[v_j] = \text{Minimum}(distance[v_j], distance[v_i] + w[v_i, v_j])$

Consider the following graph:



- The following sequence of figures shows the way of finding the shortest path from source vertex v_0 .



- The event queue in Dijkstra's algorithm can be implemented using array or heap.

All pairs shortest path

- Problem: In a given weighted directed graph find the shortest path between every pair of vertices.
- This problem can be solved by repeatedly invoking the single source shortest path algorithm for each vertex as a source.
- Another way of solving the problem is using dynamic programming technique. See any algorithms book.

Check Your Progress

Write true and false against the following

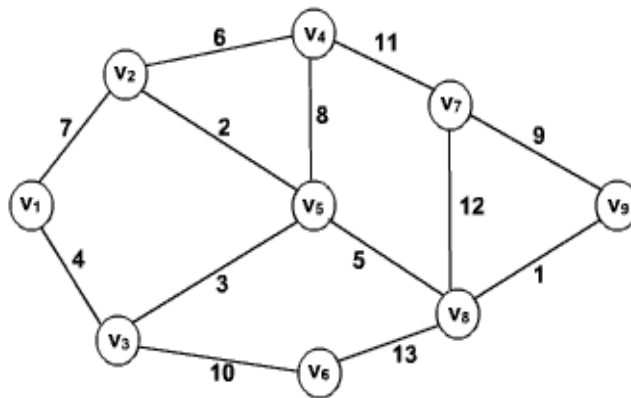
1. The spanning tree with minimum cost is called Minimum Spanning tree (MST).
2. Every graph has only one minimum spanning tree.
3. The travelling salesman problem can be solved using Minimum Spanning tree (MST)

1.6 Answer to Check Your Progress

1. True
2. False
3. True

1.7 Model Questions

1. Trace the Kruskal's algorithm on the following graph to find the MST. Show the partially constructed MST at each stage.



2. Implement the Kruskal's algorithm using set data structure.
3. Prove or disprove the statement "The MST of a graph is unique".

Block-III

UNIT IX: BINARY TREES

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Definition of Tree
- 1.4 Binary Tree
- 1.5 Binary Tree Representation
- 1.6 Tree Traversal Algorithms
 - 1.6.1 Preorder Traversal
 - 1.6.2 Inorder Traversal
 - 1.6.3 Postorder Traversal
- 1.7 Prefix, Postfix and Infix Notations
- 1.8 Answers to Check Your Progress
- 1.9 Model Questions

1.1 Learning Objectives

After going through this unit, you will be able to:

- learn about the basic definition of Tree, Binary Tree
- describe the different implementations of Tree
- describe the different traversal algorithms of Binary tree
- distinguish between Prefix, Infix, Postfix notations

1.2 Introduction

In the previous units, we have studied various data structures such as arrays, stacks, queues and linked list. All these are linear data structures as elements are stored in a linear or sequential order. A tree is a nonlinear data structure that is, the elements are not stored in sequential order but it represents data in a hierarchical manner. It associates every object to a node in the tree and maintains the parent/child relationships between those nodes. Here, in this unit, we will introduce you to the basic terminology and its different operations as well as implementations of the trees and binary trees which are the most important nonlinear data structures. We will also discuss about the different tree traversal techniques in this unit.

1.3 Definition of tree

The tree data structure is mainly used to represent the data containing a hierarchical relationship between elements, e.g. records, family trees and tables of elements.

Each tree must have at least one node, called the root, from which all nodes of the tree extend (and which has no parent of its own). The other end of the tree – the last level down—contains the leaf nodes of the tree.

A tree can be recursively defined as a set of one or more nodes where one nodes is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. The number of lines you pass through when you travel from the root until you reach a particular node is the depth of that node in the tree (node G in the figure 12.1 has a depth of 2 and node A has 0 depth). The height of the tree is the maximum depth of any node in the tree (the tree in Figure 12.1 has a

height of 3). The number of children emanating from a given node is referred to as its degree—for example, node A in the figure has a degree of 3 and node H has a degree of 1. Remember that the degree of a leaf node is zero. In-degree of a node is the number of edges arriving at that node and out-degree of a node is the number of edges leaving that node. Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level 1. Therefore, every node is at a level higher than its parent.

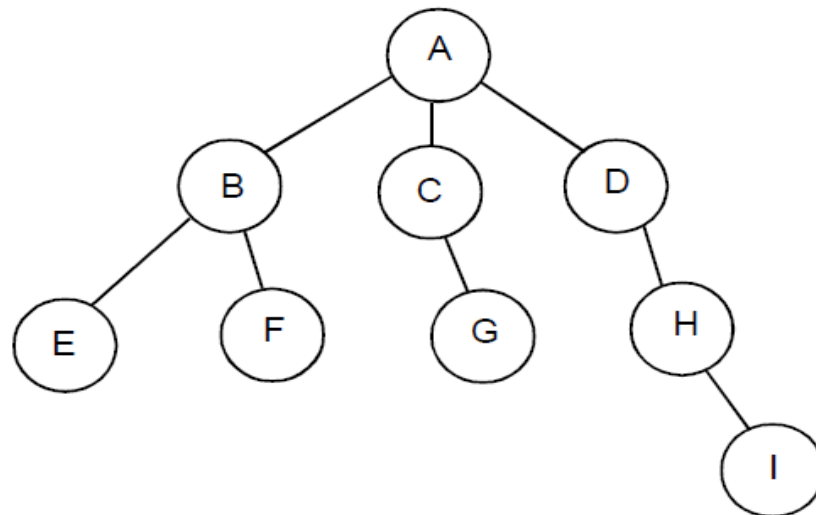


Fig. 12.1 : Tree

A tree is also defined as a finite nonempty set of elements in which one element is called the root and the remaining elements are partitioned into $m \geq 0$ disjoint subsets, each of which is itself a tree. Each element in a tree is called a node of the tree. A node with no sub-tree is a leaf.

An ordered tree is defined as a tree in which the sub-trees of each node form an ordered set. A forest is an ordered set of ordered tree.

A general tree is defined to be a nonempty finite set T of elements, called nodes, such that:

- 1) T contains a distinguished element R , called the root of T .
- 2) The remaining elements of T form an ordered collection of zero or more disjoint trees T_1, T_2, \dots, T_n . The trees T_1, T_2, \dots, T_n are called sub-trees of the root R , and the roots of T_1, T_2, \dots, T_n are called successors of R .

1.4 Binary Tree

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

- a) T is empty (called the null tree or empty tree), or
- b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.

If T does contain a root R, then the two trees T1 and T2 are called, respectively, the left and right sub-trees of R. If T1 is nonempty, then its root is called the left successor of R. Similarly, if T2 is nonempty, then its root is called the right successor of R. In a binary tree, the top most element is called the root node and each node has 0,1 or at most 2 children. Every node contains a data element, a left pointer which points to the left child and a right pointer which points to the right child.

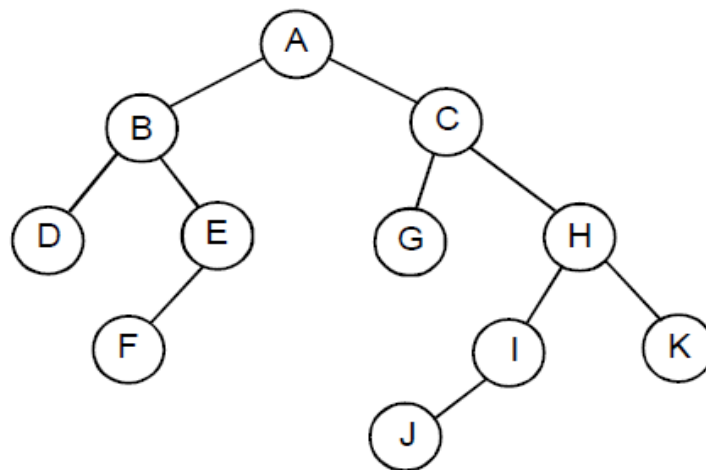


Fig. 12.2 : Binary Tree

In the above figure 12.2 a binary tree T is presented. Here in the binary tree T we have

- i) T consists of 11 nodes, represented by the letter A through K
- ii) The root of T is the node A at the top of the diagram.
- iii) B is a left successor and C is the right successor of the node A.
- iv) The left sub-tree of the root A consists of the nodes B, D, E, and F and the right sub-tree of A consists of the nodes C, G, H, I, J and K. Any node N in a binary tree T has 0, 1, or 2 successors. Here the nodes A, B, C and H have two successors. The nodes E and I have only one successor, and the nodes D, F, G, J and K have no successors. The nodes with no successors are called terminal nodes. A terminal node is also called a leaf, and a path ending in a leaf is called branch. Each node in a binary tree T is assigned a level number. The

root A of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent.

The depth (or height) of a tree T is the maximum number of nodes in a branch of T. This is 1 more than the largest level number of T. A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. A binary tree of n nodes has exactly n-1 edges. The tree T in the figure 12.2 has depth 5.

- **Complete Binary Tree:** For any binary tree T each node of T can have at most two children. Accordingly, we can show that level r of binary tree T has at the most 2^r nodes. The tree T is said to be a complete binary tree if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

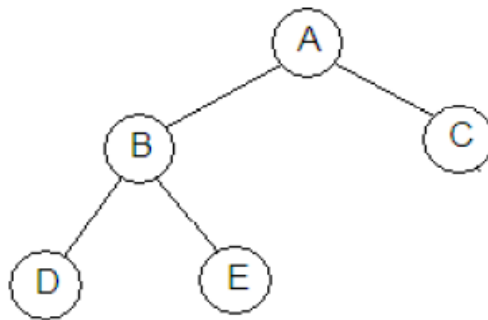


Fig. 12.3 : A complete Binary Tree

The height of a tree having exactly n nodes is given by $\log_2(n+1)$.

- **Strictly Binary Tree:** When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree. It is shown in the figure 12.4

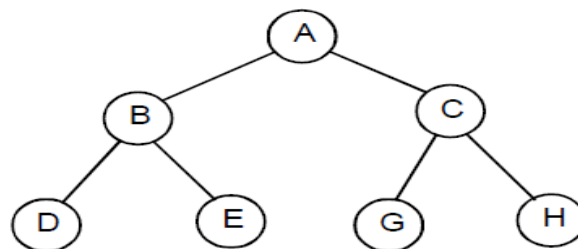


Fig.12.4 : A strictly Binary Tree

In the above strictly binary tree A, B and C are non-terminal nodes with non-empty left and right sub trees.

- **Extended Binary Tree:** When every node of a tree has either 0 or 2 children then such a tree is called extended binary tree or 2-tree. The nodes with two children are called internal nodes. The nodes without children are known as external nodes. Sometimes the nodes are distinguished in diagram by using circles for internal nodes and squares for external nodes.

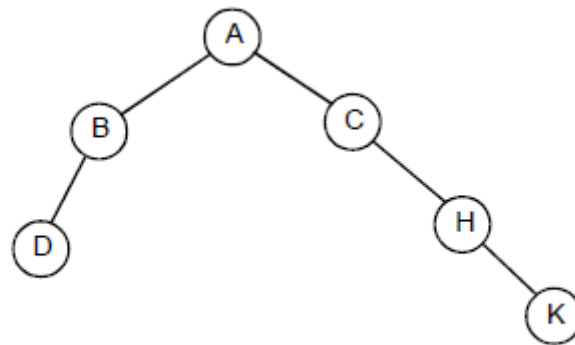


Fig. 12.5 (a) : Binary Tree

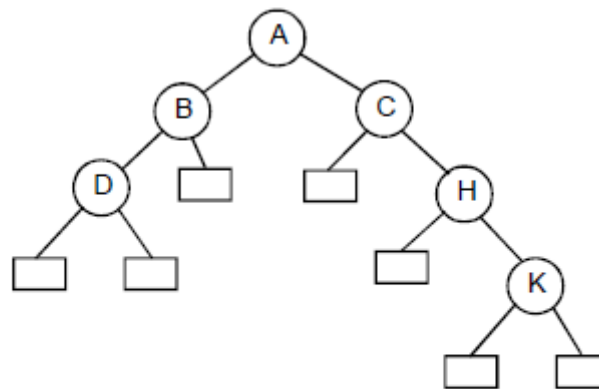


Fig. 12.5 (b) : Extended Binary Tree

1.5 Binary Tree Representation

Binary tree can be represented by two ways:

- i) Array representation
- ii) Linked representation

i) **Array representation:** In any type of data structure array representation plays an important role. The nodes of trees can be stored in an array. The nodes can be accessed in sequence one after another.

In array, element counting starts from zero (0) to $n-1$ where n is the maximum number of nodes.

For example, for an integer array declaration:

`int tree [n]` ; The root node of the tree always starts at index zero. Then successive memory locations are used for storing left and right child nodes.

Consider the following:

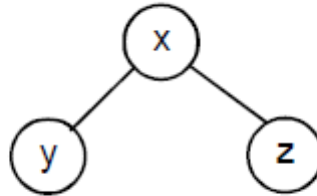


Fig. 12.6 : Array representation of tree

Note: Array representation of a binary tree with depth d will require an array with approximately 2^{d+1} elements.

ii) Linked representation: Let us consider a binary tree T . T can be maintained in memory by means of a linked representation which uses three parallel arrays INFO, LEFT and RIGHT, and a pointer variable ROOT as follows :

- 1) INFO[K] contains the data at the node N .
- 2) LEFT[K] contains the location of the left child of node N .
- 3) RIGHT[K] contains the location of the right child of node N . ROOT will contain the location of the root R of T . If any sub-tree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then ROOT will contain the null value.

Example: Write a C program to implement binary tree and display it.

```
#include <stdlib.h>
#include <stdio.h>
struct tree
{
char info;
struct tree *left;
struct tree *right;
};
struct tree *root; /* first node in tree */
struct tree *stree(struct tree *root,
struct tree *r, char info);
```



```

void print_tree(struct tree *root, int l);
void main()
{
char s[80];
root = NULL; /*initialize the root */
do
{
printf(Enter a letter: “);
gets(s);
root = stree(root, root, *s);
}while(*s);
print_tree(root,0);
getch();
} //end of main()

struct tree *stree( struct tree *root, struct tree *r, char info)
{
if(!r)
{
r = (struct tree *) malloc(sizeof(struct tree));
if(!r)
{
printf(“Out of Memory\n”);
exit(0);
}
r->left = NULL;
r->right = NULL;
r->info = info;
if(!root)
return r ;
if(info < root->info)
root->left = r;
else
root->right = r;
return r;
}
}

```

```

}
if(info < r->info)
stree(r, r->left, info);
else
stree(r, r->right, info);
return root;
}
void print_tree(struct tree *r, int l)
{
int i;
if(!r)
return ;
print_tree(r->right, l+1);
for (i=0; i<l; ++i) printf(" ");
printf ( "%c\n", r->info);
print_tree(r->left, l+1);
}

```

1.6 Tree Traversal Algorithms

To traverse a binary tree means to pass through the tree, enumerating each of its nodes once. There are three standard methods of traversing a binary tree T with root R. These methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right sub-tree. The only difference among the methods is the order in which these three operations are performed. These traversal methods are:

- a) Preorder traversal
- b) Inorder traversal
- c) Postorder traversal

1.6.1 Preorder Traversal

To traverse a binary tree in preorder (also called depth-first order), we perform the following three operations:

- 1) Visit the root
 - 2) Traverse the left sub-tree in preorder
 - 3) Traverse the right sub-tree in preorder
- Function for preorder traversal is as follows:

```
void preorder(struct node *r)
{
    if(r!=NULL)
    {
        printf("\t%d",r->data);
        preorder(r->left);
        preorder(r->right);
    }
}
```

1.6.2 Inorder Traversal

To traverse a binary tree in inorder (also called symmetric order), we perform the following three operations:

- 1) Traverse the left sub-tree in inorder
 - 2) Visit the root
 - 3) traverse the right subtree in inorder
- Function for inorder traversal is as follows:

```
inorder(struct node *r)
{
    if(r!=NULL)
    {
```

```

        inorder(r->left);
        printf("\t%d",r->data);
        inorder(r->right);
    }
}

```

1.6.3 Postorder Traversal

To traverse a binary tree in postorder , we perform the following three operations :

- 1) Traverse the left subtree in postorder
- 2) Traverse the right subtree in postorder
- 3) Visit the root

Function for postorder traversal is as follows:

```

void postorder(struct node *r)
{
    if(r!=NULL)
    {
        postorder(r->left);
        postorder(r->right);
        printf("\t%d",r->data);
    }
}

```

Example: Write a C program to implement traversals (preorder, inorder, postorder) of binary tree.

```

#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *right, *left;
}*root,*p,*q;
struct node *make(int y)
{

```

```

struct node *newnode;
newnode=(struct node *)malloc(sizeof(struct node));
newnode->data=y;
newnode->right=newnode->left=NULL;
return(newnode);
}
void left(struct node *r,int x)
{
if(r->left!=NULL)
printf("\n Invalid !");
else
r->left=make(x);
}
void right(struct node *r,int x)
{
if(r->right!=NULL)
printf("\n Invalid !");
else
r->right=make(x);
}
void inorder(struct node *r)
{
if(r!=NULL)
{
inorder(r->left);
printf("\t%d",r->data);
inorder(r->right);
}
}
void preorder(struct node *r)
{
if(r!=NULL)
{
printf("\t%d",r->data);

```

```

preorder(r->left);
preorder(r->right);
}
}
void postorder(struct node *r)
{
if(r!=NULL)
{
postorder(r->left);
postorder(r->right);
printf("\t%d",r->data);
}
}
void main()
{
int no;
int choice;
clrscr();
printf("\n Enter the root:");
scanf("%d",& no);
root=make(no);
p=root;
while(1)
{
printf("\n Enter another number:");
scanf("%d",& no);
if(no==-1)
break;
p=root;
q=root;
while(no!=p->data && q!=NULL)
{
p=q;
if(no<p->data)

```

```

q=p->left;
else
q=p->right;
}
else
{
right(p,no);
printf("\n Right Branch of %d is %d",p->data,no);
}
}
while(1)
{
printf("\n 1.Inorder Traversal \n 2.Preorder Traversal
\n 3.Postorder Traversal \n 4.Exit");
printf("\n Enter choice:");
scanf("%d",&choice);
switch(choice)
{
case 1 : inorder(root);
break;
case 2 : preorder(root);
break;
case 3 : postorder(root);
break;
case 4 : exit(0);
default:printf("Error ! Invalid Choice ");
break;
}
getch();
}
}

```

For example, let us look at the following traversals for the binary tree T given below:

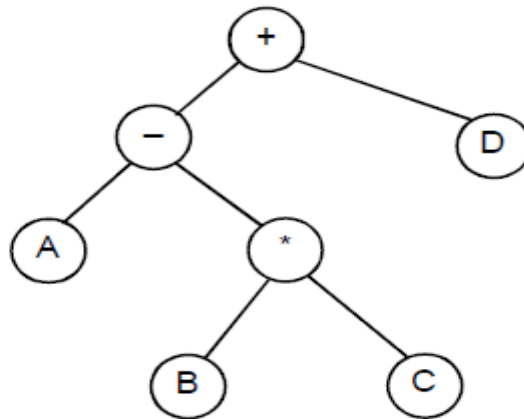


Fig. 12.7 Binary Tree

Inorder traversal : $A - B * C + D$

Preorder traversal : $+ - A * B C D$

Postorder traversal : $A B C * - D +$

Inorder traversal of a binary tree visits the nodes in ascending order.

1.7 Prefix, Postfix and Infix Notations

Let us consider the sum of two variables A and B. We think of applying the operator “+” to the operands A and B and write the sum as A+B. This particular representation is called infix notation. There are two alternate notations for expressing the sum of A and B using the symbol A, B and +.

These are:

+ A B (prefix notation)

A B + (postfix notation)

These prefixes “pre-”, “post-” and “in-” refer to the relative position of the Operators with respect to the two operands. These techniques of representing arithmetic expressions where operators will be before or after operands were presented by Polish (of Poland) mathematician Jan Lukasiewicz. So, these notations are also called Polish notation. According to this:

Prefix Notation: In prefix notation, the operator precedes the operands.

For example: $+ A B C$

Postfix Notation: In postfix notation, the operator follows the operands.

For example: $A B + C -$

Infix Notation: In infix notation, the operator is between the operands. For

Example: $A + B - C$

For example, let us write these three notations for the expression

$A - B * C + D$

Infix notation is: $A - B * C + D$

Prefix notation is: $+ - A * B C D$

Postfix notation is: $A B C * - D +$

We can also express these three notations infix, prefix and postfix from the Binary tree traversal point of view.

For example, the expression $A - B * C + D$ can be represented by the binary tree as follows:

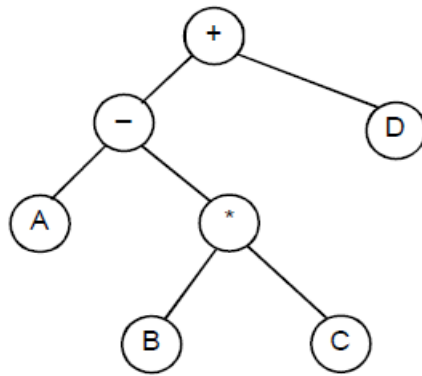


Fig. 12.8 : Binary Tree for $A - B * C + D$

If we traverse the above binary tree we see that:

Inorder traversal equivalent to infix expression: $A - B * C + D$

Preorder traversal equivalent to prefix expression: $+ - A * B C D$

Postorder traversal equivalent to postfix expression: $A B C * - D +$

CHECK YOUR PROGRESS

Q.1. If a binary tree is traversed in inorder then numbers of the nodes are printed

a) in ascending order

- b) in descending order
- c) randomly
- d) none of these.

Q.2. Binary expression tree is traversed in ____ traversal

- a) preorder
- b) postorder
- c)inorder
- d) both (a) & (b)

Q.3. The number of nodes in a full binary tree of depth 4 is

- a) 15
- b) 16
- c) 14
- d)13

Q.4. How many nodes does a complete binary tree of level 5 have ?

- a) 15
- b) 25
- c) 63
- d) 33

Q.5. At level r, binary tree T have at the most ____ nodes.

- a) $2^r + 1$
- b) 2^r
- c) $2^r - 1$
- d) none of these

Q.6. Postorder traversal is

- a) root,left subtree,right subtree
- b) left subtree, root ,right subtree
- c) left subtree, right subtree, root
- d) root, right subtree, left subtree

Q.7. Prefix notation representation of : $A + B - C$ is

- a) $A B + C -$
- b) $- + A B C$
- c) $ABC + -$
- d) none of these.

Q.8. Traversing a binary tree, first root and then left and right subtree is called

- a) postorder
- b) inorder
- c) perorder
- d) none of these

Q.9. The postfix equivalent of the prefix $* + a b - c d$ is

- a) $a b + c d - *$
- b) $a b c d + - *$
- c) $a b + c d * -$
- d) $a b + - c d *$

1.8 Answer to Check Your Progress

Ans. to Q. No. 1: a) ascending order

Ans. to Q. No. 2: c) inordered

Ans. to Q. No. 3: a) 15

Ans. to Q. No. 4: c) 63

Ans. to Q. No. 5: b) 2^r

Ans. to Q. No. 6: c) left subtree, right subtree, root

Ans. to Q. No. 7: b) $- + A B C$

Ans. to Q. No. 8: c) perorder

Ans. to Q. No. 9: a) $a b + c d - *$

1.9 Model Questions

1. Explain trees and binary trees.
2. Give the properties of binary tree.
3. Explain the various types of binary trees.
4. What do you mean by traversing? Explain different traversal methods of binary tree.
5. How are binary trees represented?

UNIT X: HEAP SORT

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Heap Sort
- 1.4 Heap Representation
- 1.5 Heapification
- 1.6 Priority Queue
- 1.7 Answer to Check Your Progress
- 1.8 Model Questions

1.1 Learning Objectives

After going through this, unit the learner will able to learn about:

- Heap Sort
- Heap Representation
- Heapification
- Priority Queue

1.2 Introduction

The first sort we looked at was insertion sort which had (average) run time $O(n^2)$ but required a constant (with respect to n) amount of additional storage. Merge sort on the other hand, had $O(n \lg n)$ run time, but the amount of memory required grew with n (since at each level of recursion new partial arrays are allocated). *Heap sort* is a sorting algorithm that maintains the $O(n \lg n)$ run time (like merge sort) but operates *in place* (like insertion sort). This sorting algorithm is based on a data structure known as a *heap* (which is also one way to implement a *priority queue*).

1.3 Heap sort

- Heap sort is an efficient sorting algorithm with average and worst case time complexities are in $O(n \log n)$.
- Heap sort is an in-place algorithm i.e. does not use any extra space, like merge sort.
- This method is based on a data structure called Heap.
- Heap data structure can also be used as a priority queue.

Heap:

- A binary heap is a complete binary tree in which each node other than root is smaller than its parent.
- Heap example:

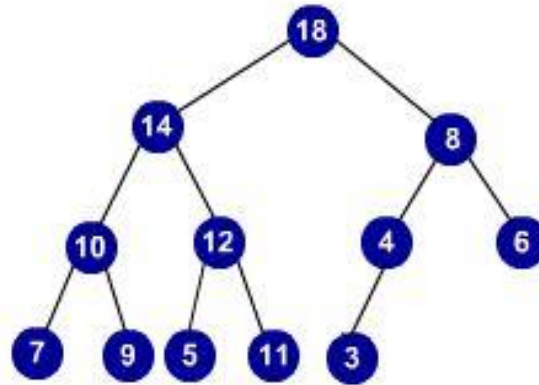


Fig: 1

1.4 Heap Representation

Heap Representation:

- A Heap can be efficiently represented as an array
- The root is stored at the first place, i.e. $a[1]$.
- The children of the node i are located at $2*i$ and $2*i + 1$.
- In other words, the parent of a node stored in i th location is at floor. $\left\lfloor \frac{i}{2} \right\rfloor$
- The array representation of a heap is given in the figure below.

1	2	3	4	5	6	7	8	9	10	11	12
18	14	8	10	12	4	6	7	9	5	11	3

Fig. 2

Check Your Progress

1. Write true and false against the following:
 - i. Heap data structure can also be used as a priority queue.
 - ii. A binary heap is a complete binary tree in which each node other than root is smaller than its parent.
 - iii. A Heap can be efficiently represented as a stack.

1.5 Heapification

- Before discussing the method for building heap of an arbitrary complete binary tree, we discuss a simpler problem.
- Let us consider a binary tree in which left and right sub-trees of the root satisfy the heap property, but not the root. See the following figure.
- Now the question is how to transform the above tree into a heap?
- Swap the root and left child of root, to make the root satisfy the heap property.
- Then check the sub-tree rooted at left child of the root is heap or not. If it is, we are done. If not, repeat the above action of swapping the root with the maximum of its children.
- That is, push down the element at root till it satisfies the heap property.
- The following sequence of figures depicts the heapification process.

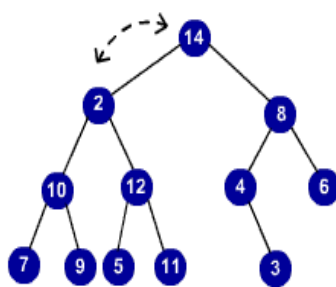


fig 4

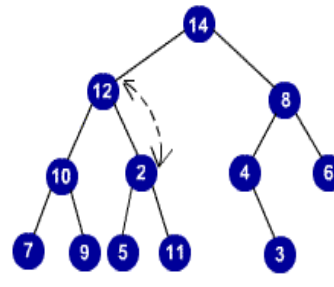


Fig : 4.1

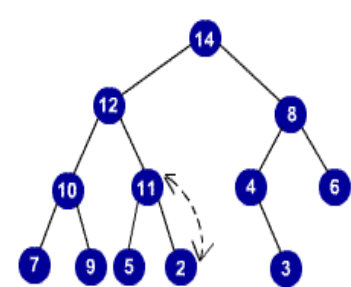


Fig : 4.2

Algorithm: Heapification (a,i,n)

Step 1 : left = 2i

Step 2 : right = 2i + 1

Step 3 : if (left < n) and (a[left] > a[i]) then

Step 4 : maximum = left

Step 5 : else

Step 6 : maximum = i

Step 7 : if (right < n) and (a[right] > a[maximum]) then

Step 8 : maximum = right

Step 9 : if (maximum != i) then

Step 10 : swap(a[i],a[maximum])

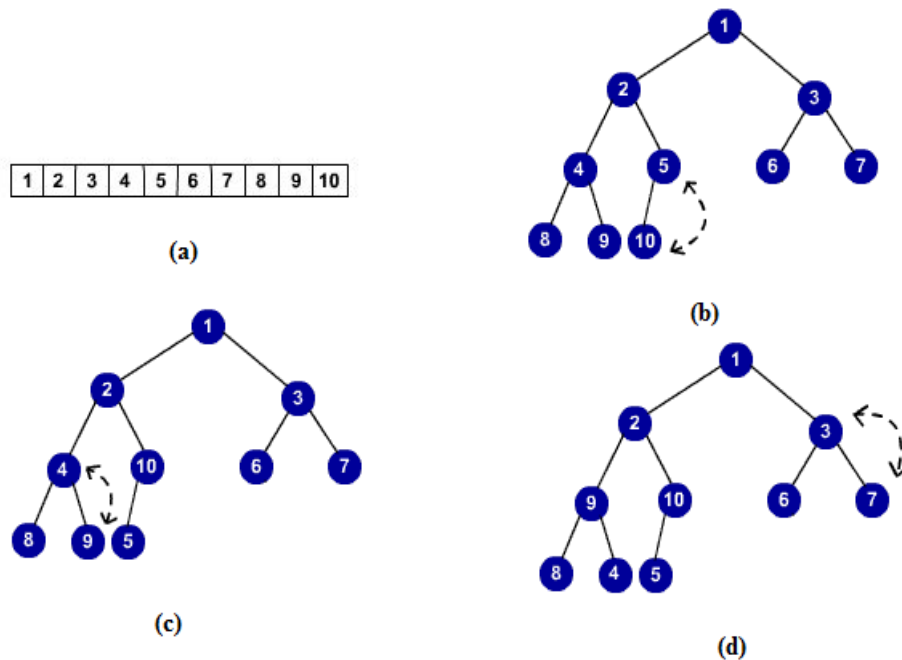
Step 11 : heapfication(a, maximum, n)

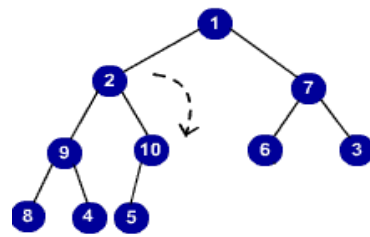
The time complexity of heapification is $O(\log n)$.

Build Heap

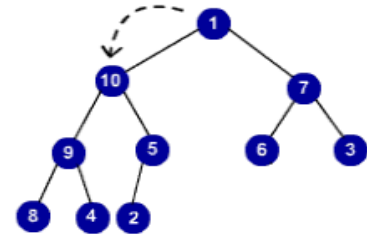
- Heap building can be done efficiently with bottom up fashion.
- Given an arbitrary complete binary tree, we can assume each leaf is a heap.
- Start building the heap from the parents of these leaves. i.e., heapify sub-trees rooted at the parents of leaves.
- Then heapify sub-trees rooted at their parents. Continue this process till we reach the root of the tree.

The following sequence of the figures illustrates the build heap procedure.

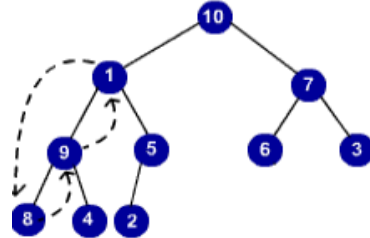




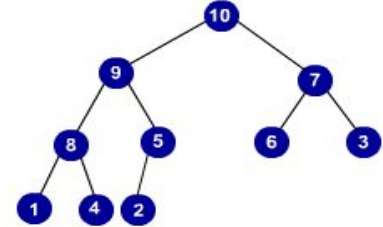
(e)



(f)



(g)



(h)

Algorithm : build_heap(a,i,n)

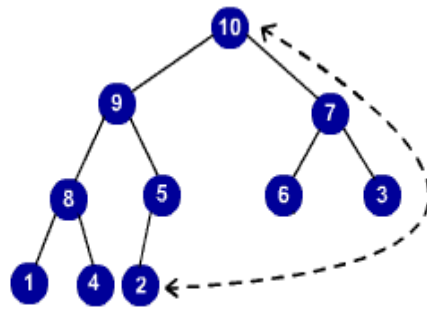
step 1 : for j = down to 1 do

Step 2 : heapification(a,j,n)

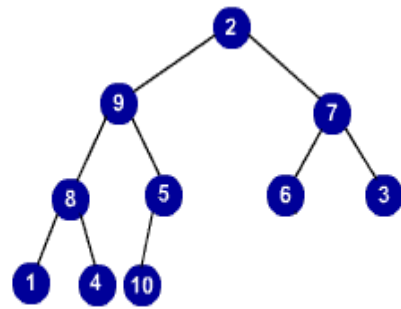
The time complexity of the build heap is in $O(n)$.

Heap sort

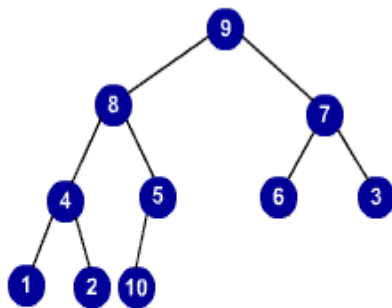
- Given an array of n element, first we build the heap.
- The largest element is at the root, but its position in sorted array should be at last. So, swap the root with the last element and heapify the tree with remaining n-1 elements.
- We have placed the highest element in its correct position. We left with an array of n-1 elements. Repeat the same of these remaining n-1 elements to place the next largest element in its correct position.
- Repeat the above step till all elements are placed in their correct positions.



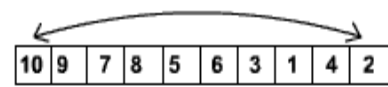
(a)



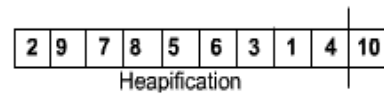
(b)



(c)



(d)



(e)

- Pseudocode of the algorithm is given below.

Algorithm Heap_Sort(a,i)

```

build_heap(a,i)
for j = i down to 1 do

    swap (a[1],a[j])
    heapification(a,1,j-1)

```

The time complexity of the heap sort algorithm is in $O(n \log n)$

1.6 Priority Queue

- Let consider a set S of elements, such that each element has priority.
- We want to design a data structure for these elements such that the highest priority element should be extracted/ deleted efficiently.

- We can use heap for this purpose since highest element, is always at the root, which can be extracted quickly.
- Pseudocode for extracting the maximum from the priority queue P is given below. Let the global variable size maintains the number of elements in P.

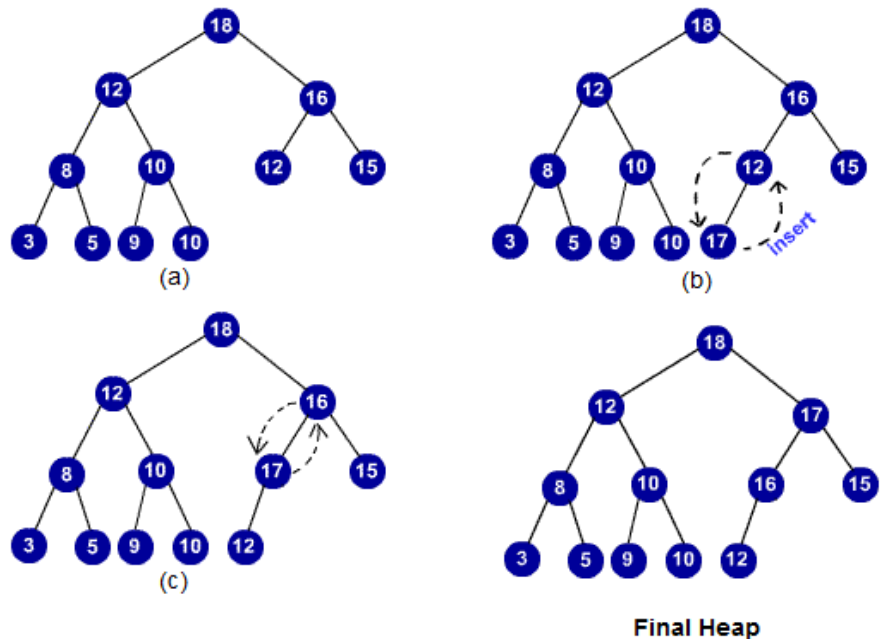
Algorithm : Max_Extract(P)

```

max = p[1]
P[1] = P[size]
size = size -1
heapification(P,1,size)
return (max)

```

- After extracting the maximum, we have to maintain remaining elements in the priority queue. So we heapify before returning the maximum.
- Other operation to be supported is to insert an element into the priority queue.
- Inserting an element into the priority queue can be done easily.
- Insert the new element as a new leaf, and push this up till it satisfies the heap property.
- The following sequence of figures illustrates the inserting procedure.



- The pseudo code is given below.

Algorithm: Insert (P, x)

size =size + 1

i = size

while (i > 1) and (x > P[i]) do

 P[i] = P[i-1]

 i = i - 1

P[i] = x

Check Your Progress

Choose the correct one

2. (i)What is the typical running time of a heap sort algorithm?

- a) $O(N)$
- b) $O(N \log N)$
- c) $O(\log N)$
- d) $O(N^2)$

(ii) What is its worst case time complexity of Heap sort?

- a) $O(n \log n)$
- b) $O(n^2)$
- c) $O(n^3)$
- d) None of the above

(iii) Which of the following sorting algorithm is stable?

- a) insertion sort
- b) bubble sort
- c) quick sort
- d) heap sort

1.7 Answer to Check Your Progress

1. i. True ii. True iii. False
2. i. b) $O(N \log N)$ ii. a) $O(n \log n)$ iii. d) heap sort

1.8 Model Questions

1. Write a program for heap sort.
2. Describe heapsort and show that its worst case performance is $O(n \log n)$.
3. Design a heap sort algorithm to sort in non-ascending order.
4. Describe the time complexity of inserting an element into a complete heap in terms of N , the number of elements in the heap, and in terms of H , the height of the tree.

UNIT XI: SEARCH TREES

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 AVL-Tree
- 1.4 Representation of AVL Tree
- 1.5 B-Tree
- 1.6 Multiway Search Trees
- 1.7 Operations On B – Tree
- 1.8 Answer to Check Your Progress
- 1.9 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to learn about:

- AVL-Tree
- Representation of AVL Tree
- B-Tree

- Multiway Search Trees
- Operations On B – Tree

1.2 Introduction

Avl- Tree

As we know that searching in a binary search tree is efficient if the height of the left sub-tree and right sub-tree is same for a node. But frequent insertion and deletion in the tree affects the efficiency and makes a binary search tree inefficient. The efficiency of searching will be ideal if the difference in height of left and right sub-tree with respect of a node is at most one. Such a binary search tree is called balanced binary tree (sometimes called AVL Tree).

1.3 Avl- Tree

- The AVL tree is a binary search tree that's always in balance.
 - Developed in 1962 by Georgii Adel'son-Vel'sky and Yevheniy Landis.
- Each unbalancing insertion or deletion restores balance as part of its operation.
- Not so popular these days.
 - More efficient and simpler alternatives are available.

An AVL Tree is self-balancing height-balanced Binary Search Tree. AVL stands for the inventors of this tree: **Adelson-Velskii** and **Landis**. Search, insertion and deletion are in $O(\log n)$ time in both average and worst cases. AVL Tree is better than Red-Black Tree for search operation because it is more strictly balanced.

AVL Tree approximately balances itself. Each node contains the attribute **balance factor** that indicates if the the sub-tree (whose node is considered as the root) is:

- Left-heavy: if the height of the left sub-tree is greater than the height of the right sub-tree.

- **Balanced:** if the height of the left and right sub-trees are the same.
- **right-heavy:** if the height of the right sub-tree is greater than the height of the left sub-tree.

If an insertion unbalance the tree, a **rotation** should be done to correct the balance

1.4 Representation of AVL Tree

In order to represent a node of an AVL Tree, we need four fields: - One for data, two for storing address of left and right child and one is required to hold the balance factor. The balance factor is calculated by subtracting the right sub-tree from the height of left sub - tree.

The structure of AVL Tree can be represented by: -

```

        Struct AVL
    {
        struct AVL *left;
        int data;
        struct AVL *right;
        int balfact;
    };

```

DETERMINATION OF BALANCE FACTOR

The value of balance factor may be -1, 0 or 1.

Any value other than these represent that the tree is not an AVL Tree

- If the value of balance factor is -1, it shows that the height of right sub-tree is one more than the height of the left sub-tree with respect to the given node.
- If the value of balance factor is 0, it shows that the height of right sub-tree is equal to the height of the left Sub-tree with respect to the given node.
- If the value of balance factor is 1, it shows that the height of right sub-tree is one less than the height of the left sub-tree with respect to the given node.

INVENTION AND DEFINITION

It was invented in the year 1962 by two Russian mathematicians named G.M. Adelson-Velskii and E.M. Landis and so named AVL Tree.

It is a binary tree in which difference of height of two sub-trees with respect to a node never differ by more than one (1).

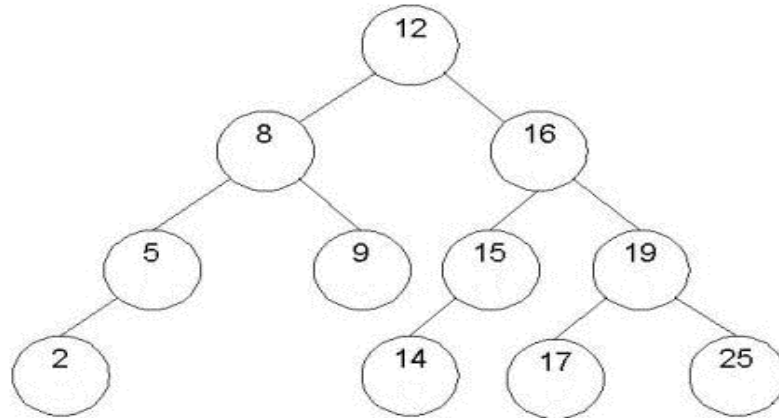


Diagram showing AVL Tree

INSERTION OF A NODE IN AVL TREE

Insertion can be done by finding an appropriate place for the node to be inserted. But this can disturb the balance of the tree if the difference of height of sub-trees with respect to a node exceeds the value one. If the insertion is done as a child of non-leaf node then it will not affect the balance, as the height doesn't increase. But if the insertion is done as a child of leaf node, then it can bring the real disturbance in the balance of the tree.

This depends on whether the node is inserted to the left sub-tree or the right sub-tree, which in turn changes the balance factor. If the node to be inserted is inserted as a node of a sub-tree of smaller height then there will be no effect. If the height of both the left and right sub-tree is same then insertion to any of them doesn't affect the balance of AVL Tree. But if it is inserted as a node of sub-tree of larger height, then the balance will be disturbed.

To rebalance the tree, the nodes need to be properly adjusted. So, after insertion of a new node the tree is traversed starting from the new node to the

node where the balance has been disturbed. The nodes are adjusted in such a way that the balance is regained.

ALGORITHM FOR INSERTION IN AVL TREE

```
int avl_insert(node *treep, value_t target)
{
/* insert the target into the tree, returning 1 on success or 0 if it
* already existed
*/
node tree = *treep;
node *path_top = treep;
while (tree && target != tree->value)
{
direction next_step = (target > tree->value);
if (!Balanced(tree)) path_top = treep;
treep = &tree->next[next_step];
tree = *treep;
}
if (tree) return 0;
tree = malloc(sizeof(*tree));
tree->next[0] = tree->next[1] = NULL;
tree->longer = NEITHER;
tree->value = target;
*treep = tree;
avl_rebalance(path_top, target);
return 1;
}
```

ALGORITHM FOR REBALANCING IN INSERTION

```
void avl_rebalance_path(node path, value_t target)
{
/* Each node in path is currently balanced. Until we find target, mark each
node as longer in the direction of rget because we know we have inserted
```

```

target there */
    while (path && target != path->value) {
        direction next_step = (target > path->value);
        path->longer = next_step;
        path = path->next[next_step];
    }
}
void avl_rebalance(node *path_top, value_t target)
{
    node path = *path_top;
    direction first, second, third;
    if (Balanced(path)) {
        avl_rebalance_path(path, target);
        return;
    }
    first = (target > path->value);
    if (path->longer != first) {
        /* took the shorter path */
        path->longer = NEITHER;
        avl_rebalance_path(path->next[first], target);
        return;
    }
    /* took the longer path, need to rotate */
    second = (target > path->next[first]->value);
    if (first == second) {
        /* just a two-point rotate */
        path = avl_rotate_2(path_top, first);
        avl_rebalance_path(path, target);
        return;
    }
    /* fine details of the 3 point rotate depend on the third step. However there
may not be a third step, if the third point of the rotation is the newly inserted
point. In that case we record the third step as NEITHER */
    path = path->next[first]->next[second];

```

```

if (target == path->value) third = NEITHER;
    else third = (target > path->value);
        path = avl_rotate_3(path_top, first, third);
        avl_rebalance_path(path, target);
    }

```

DELETION

A node in AVL Tree is deleted as it is deleted in the binary search tree. The only difference is that we have to do rebalancing which is done similar to that of insertion of a node in AVL Tree. The algorithm for deletion and rebalancing is given below:

ALGORITHM FOR DELETION IN AVL TREE

```

int avl_delete(node *treep, value_t target)
{
/* delete the target from the tree, returning 1 on success or 0 if it wasn't found
*/
    node tree = *treep;
    direction dir;
    node *targetp, targetn;
    while(tree) {
        dir = (target > value);
        if (target == value) targetp = treep;
            if (tree->next[dir] == NULL)
                break;
                if (tree->longer == NEITHER || (tree->longer == 1-
dir && tree->next[1-dir]->longer == NEITHER))
                    path_top = treep;
                    treep = &tree->next[dir];
                    tree = *treep;
                }
        if (targetp == NULL) return 0;
        targetp = avl_rebalance_del(path_top, target, targetp);
        avl_swap_del(targetp, treep, dir);
    }
}

```

```

        return 1;
    }

```

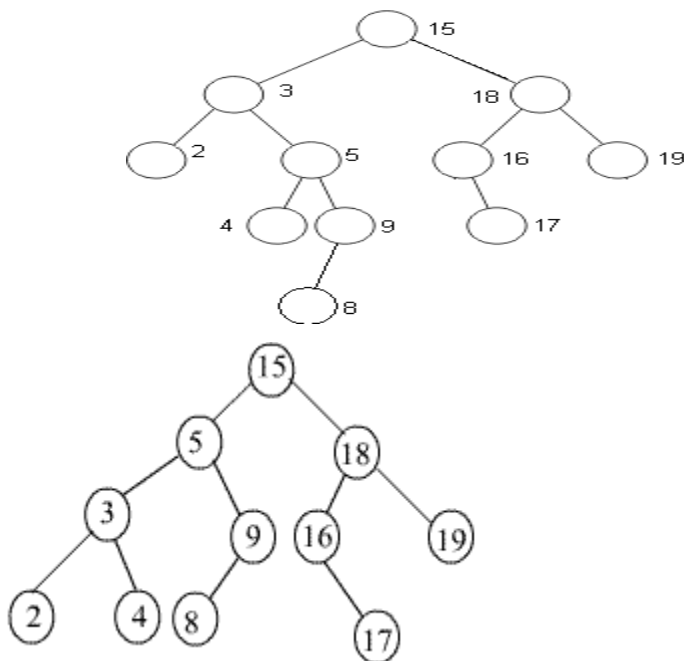
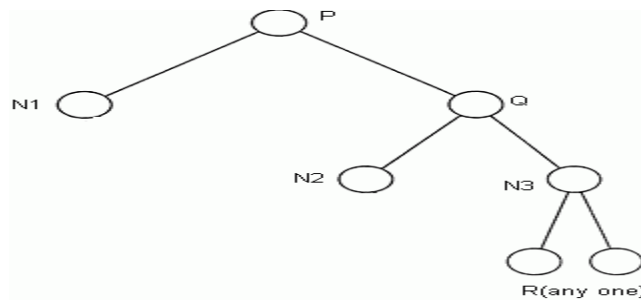
REBALANCING OF AVL TREE

When we insert a node to the taller sub-tree, four cases arise and we have different rebalancing methods to bring it back to a balanced tree form.

- Left Rotation
- Right Rotation
- Right and Left Rotation
- Left and Right Rotation

LEFT ROTATION

In general if we want to insert a node R (either as left child or right child) to N3 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called left rotation.



Before Rotation

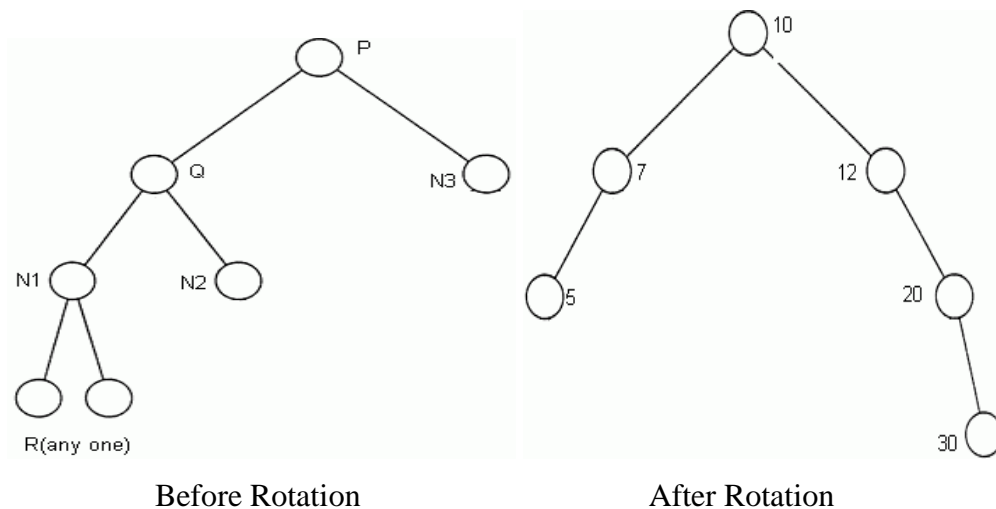
After Rotation

EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 8, it becomes the left child of node 9 and the balance doesn't exist, as the balance factor of node 3 becomes -2. So, we try to rebalance it. In order to do so, we do left rotation at node 3. Now node 5 becomes the left child of the root. Node 9 and node 3 becomes the right and left child of node 5 respectively. Node 2 and node 4 becomes the left and right child of node 3 respectively. Lastly, node 8 becomes the left child of node 9. Hence, the balance is once again attained and we get AVL Tree after the left rotation.

RIGHT ROTATION

In general if we want to insert a node R (either as left or right child) to N1 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called right rotation.



EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 7, it becomes the right child of node 5 and the balance doesn't exist, as the balance factor of node 20 becomes 2. So, we try to rebalance it. In order to do so, we do right rotation at node 20. Now node 10 becomes the root. Node 12 and node 7 becomes the right and left child of root respectively. Node 20 becomes the right child of node 12. Node 30 becomes the right child of node 20. Lastly, node 5 becomes the left child of node 7. Hence, the balance is once again attained and we get AVL Tree after the right rotation.

1.5 B-Tree

Tree structures support various basic dynamic set operations including Search, Insert, and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be $\log n$ where n is the number of nodes in the tree.

To ensure that the height of the tree is as small as possible and therefore provide the Best running time, a balanced tree structure like AVL tree, 2-3 Tree, Red Black Tree or B-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

To reduce the time lost in retrieving data from secondary storage, we need to minimize the no. of references to the secondary memory. This is possible if a node in a tree contains more no. of values, then in a single reference to the secondary memory more nodes can be accessed. The AVL trees or red Black Trees can hold a max. of 1 value only in a node while 2-3 Trees can hold a max of 2 values per node. To improve the efficiency Multiway Search Trees are used.

1.6 Multiway Search Trees

A Multiway Search Tree of order n is a tree in which any node can have a maximum of $n-1$ values & a max. of n children. B - Trees are a special case of Multiway Search Trees.

A B Tree of order n is a Multiway Search Tree of order n with the following characteristics:

1. All the non leaf nodes have a max of n child nodes & a min of $n/2$ child nodes.

2. If a root is non leaf node, then it has a max of n non empty child nodes & a min of 2 child nodes.
3. If a root node is a leaf node, then it does not have any child node.
4. A node with n child nodes has n-1 values arranged in ascending order.
5. All values appearing on the left most child of any node are smaller than the left most value of that node while all values appearing on the right most child of any node are greater than the right most value of that node.
6. If x & y are two adjacent values in a node such that $x < y$, ie they are the i th & (i+1) th values in the node respectively, then all values in the (i+1) th child of that node are $> x$ but $< y$.

The topics to be discussed under B - Trees are as follows:

REPRESENTATION OF B – TREE

The B - Tree is represented as follows using structure:

```

Struct btnode
{
    int count;
    int value[max+1];
    Struct btnode * child[max + 1];
};

```

Count is the no. of children of any node. The values of node are in the array value.

The addresses of child nodes are in child array while the Max is a macro that defines the maximum no. of values any node can have.

1.7 Operations on B - Tree

THE following operations can be done on a B - Tree :

- Searching
- Insertion
- Deletion

SEARCHING OF A VALUE IN A B-TREE

Searching of a value k in a B-Tree is exactly similar to searching for values in a 2-3 tree. To begin with the value k is compared with the first value $key[0]$ of the root node. If they are similar then the search is complete. If k is less than $key[0]$ then the search is done in the first child node or the sub-tree of the root node.

If k is greater than $key[0]$ then it is compared with $key[1]$. If k is greater than $key[0]$ and smaller than $key[1]$ then k is searched in the second child node or sub-tree of the root node. If k is greater than the last value $key[i]$ of the root node then searching is done in the last child node or sub-tree of the root node. If k is searched in any of the child nodes or sub-tree of the root node then the same procedure of searching is repeated for that particular node or sub-tree.

If the value k is found in the tree then the search is successful. The address of the node in which k is present and the position of the value k in that node is returned. If the value k is not found in the tree, then the search is unsuccessful.

INSERTION OF A VALUE IN A B-TREE

When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.)

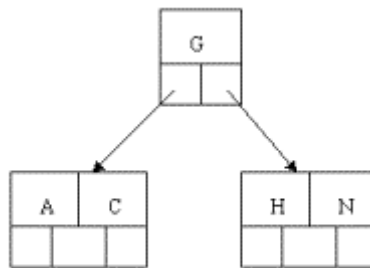
Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

Let's take an example. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes

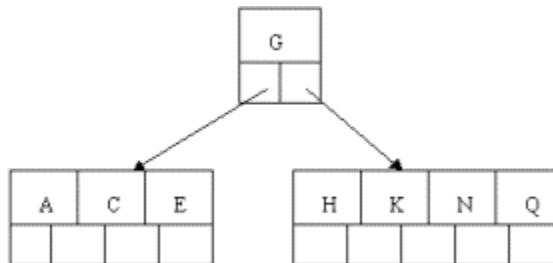
other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



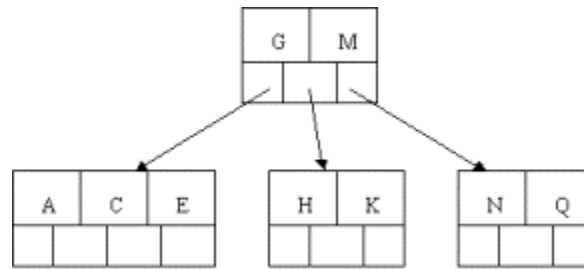
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



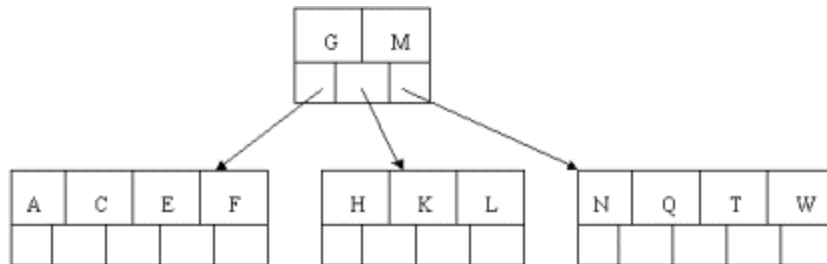
Inserting E, K, and Q proceeds without requiring any splits:



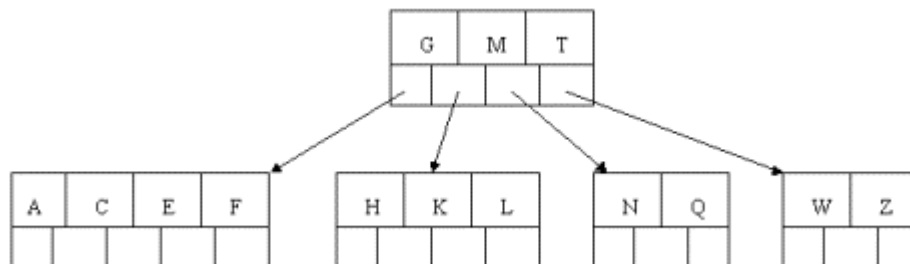
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



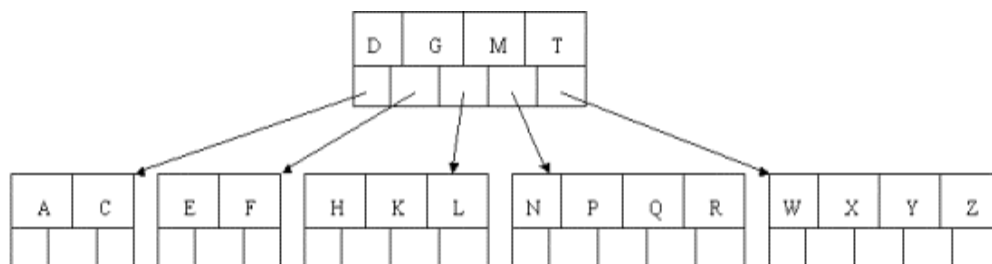
The letters F, W, L, and T are then added without needing any split.



When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

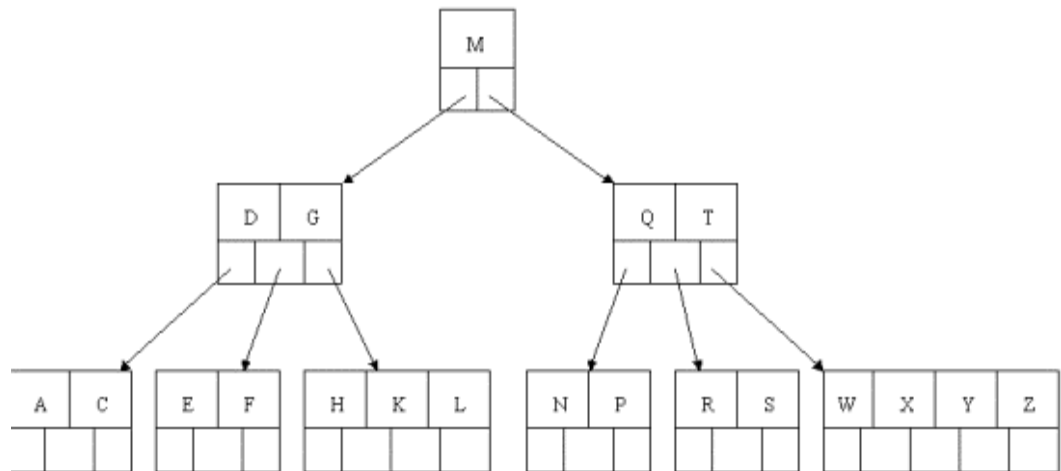


The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits,

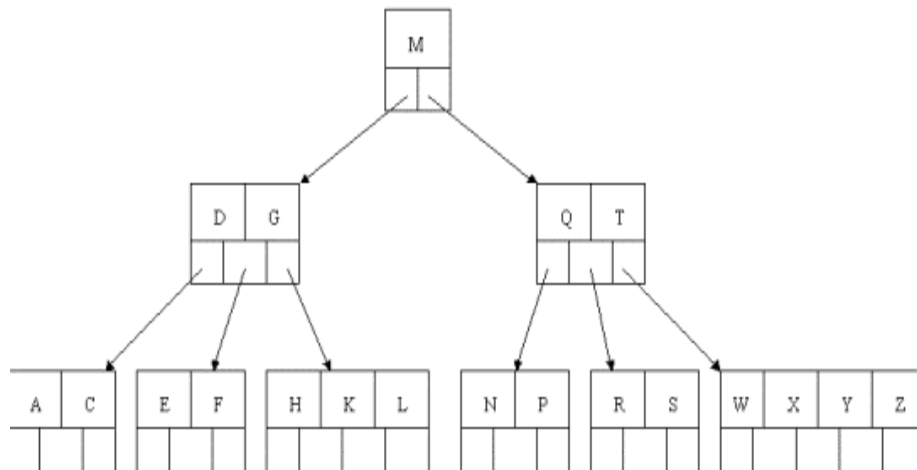
sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



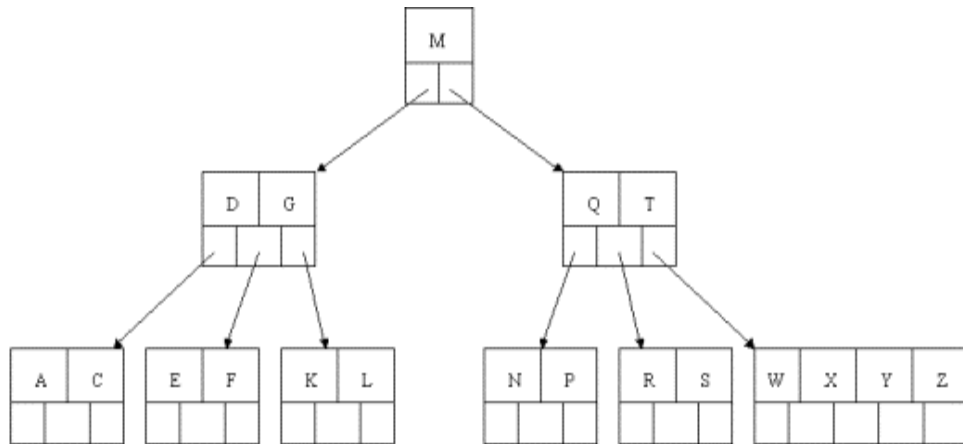
DELETION IN A B – TREE

Deletion in a B -Tree is similar to insertion. At first the node from which a value is to be deleted is searched. If found out, then the value is deleted. After deletion the tree is checked if it still follows B - Tree properties.

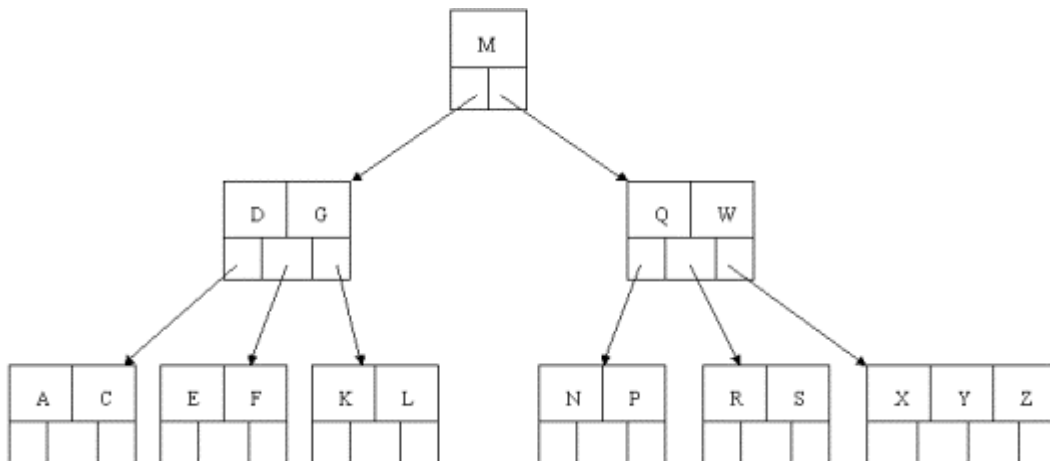
Let us take an example. The original B - Tree taken is as follows:



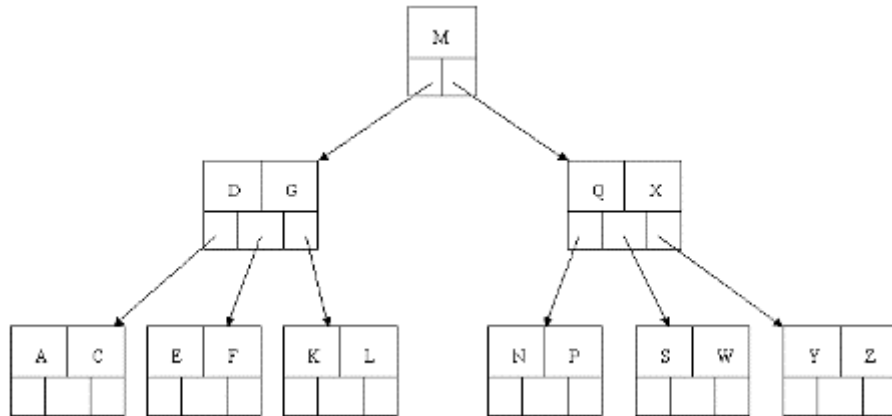
Delete H. first it is found out. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:



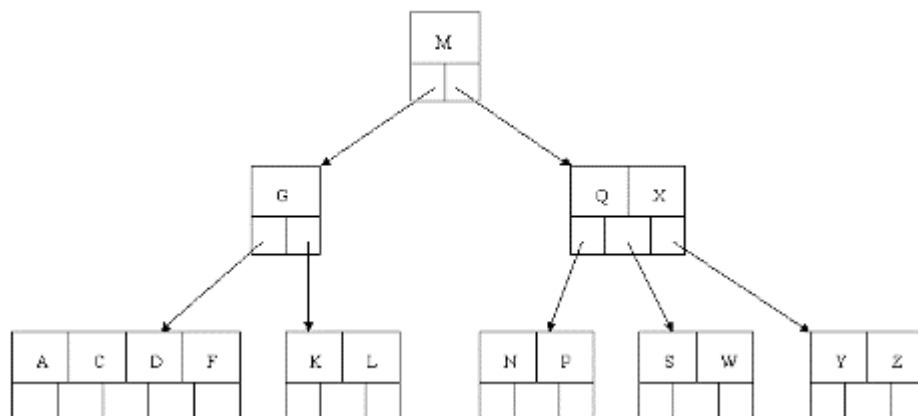
Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method



Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

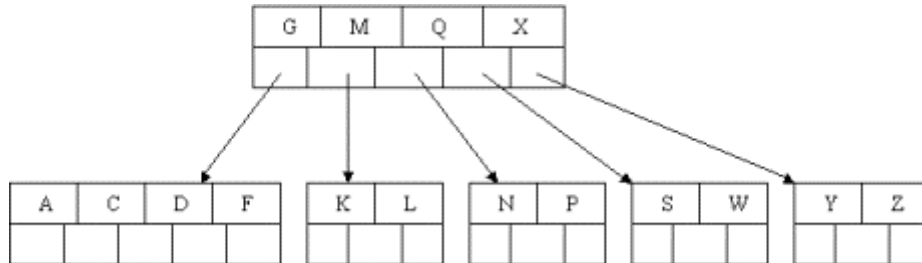


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.



Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M.

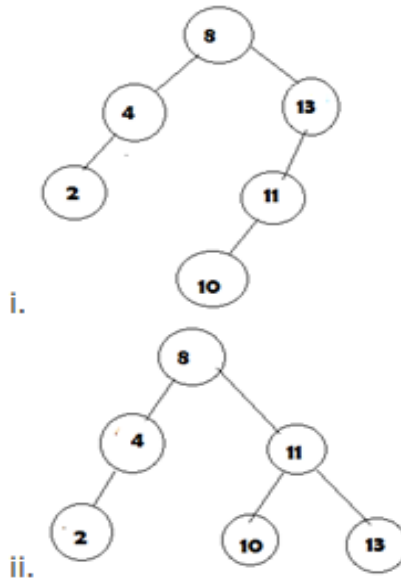
In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.



Check Your Progress

Choose the Correct one

1. What is an AVL tree?
 - a) a tree which is balanced and is a height balanced tree
 - b) a tree which is unbalanced and is a height balanced tree
 - c) a tree with three children
 - d) a tree with atmost 3 children
2. Which of the below diagram is following AVL tree property?



- a) only i
- b) only i and ii

- c) only ii
 - d) i is not a binary search tree
3. What maximum difference in heights between the leafs of a AVL tree is possible?
- a) $\log(n)$ where n is the number of nodes
 - b) n where n is the number of nodes
 - c) 0 or 1
 - d) atmost 1
4. The AVL tree is a binary search tree that's always in balance.
- a) True
 - b) False

1.8 Answer to Check Your Progress

- 1. a) a tree which is balanced and is a height balanced tree
- 2. b) only i and ii
- 3. a) $\log(n)$ where n is the number of nodes
- 4. a) True

1.9 Model Questions

- 1. What is AVL Tree?
- 2. Who invented AVL Tree?
- 3. How can we determine the balance factor?
- 4. Insert a node 3 in the AVL Tree of the tutorial and try to rebalance the tree by any of the methods?
- 5. How an AVL Tree or B - Tree can be better than a Binary Search Tree?
- 6. How an AVL Tree or B - Tree can be better than a Binary Search Tree?
- 7. What advantages does a multiway search Tree have over an AVL Tree?
- 8. What are 5the differences between a multiway search tree & a B - Tree?

UNIT XII: TABLES

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Hashing Techniques
- 1.4 Why Hashing?
- 1.5 Methods of Dealing with Hash Clash
- 1.6 Double Hashing
- 1.7 Clustering
- 1.8 Dynamic and Extendible Hashing
- 1.9 Answer to Check Your Progress
- 1.10 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to learn about:

- Hashing Techniques

- Why Hashing?
- Methods of Dealing with Hash Clash
- Double Hashing
- Clustering
- Dynamic and Extendible Hashing

Introduction

Hashing involves applying a hashing algorithm to a data item, known as the hashing key, to create a hash value. Hashing algorithms take a large range of values (such as all possible strings or all possible files) and map them onto a smaller set of values (such as a 128 bit number).

Hashing has two main applications. Hashed values can be used to speed data retrieval, and can be used to check the validity of data.

When we want to retrieve one record from many in a file, searching the file for the required record takes time that varies with the number of records. If we can generate a hash for the record's key, we can use that hash value as the "address" of the record and move directly to it; this takes the same time regardless of the number of records in the file.

The validity of data can be checked with a hash. This can be used to check both that a file transferred correctly, and that a file has not been deliberately manipulated by someone between me uploading it somewhere and you downloading it. If I post both the file and the hash value I generated from it, you can generate a hash value from the file you received and compare the hash values. If the hashing algorithm is a good *cryptographic hash*, it's extremely unlikely that accident or malice would have modified the file even a little yet it would still yield the same hash value.

1.3 Hashing Techniques

Hashing is a method to store data in an array so that sorting, searching, inserting and deleting data is fast. For this every record needs unique key.

The basic idea is not to search for the correct position of a record with comparisons but to compute the position within the array. The function that returns the position is called the 'hash function' and the array is called a 'hash table'.

1.4 Why Hashing?

In the other type of searching, we have seen that the record is stored in a table and it is necessary to pass through some number of keys before finding the desired one. While we know that the efficient search technique is one which minimizes these comparisons. Thus we need a search technique in which there are no unnecessary comparisons.

If we want to access a key in a single retrieval, then the location of the record within the table must depend only on the key, not on the location of other keys(as in other type of searching i.e. tree). The most efficient way to organize such a table is an array. It was possible only with hashing.

HASH CLASH

Suppose two keys k_1 and k_2 are such that $h(k_1)$ equals $h(k_2)$. When a record with key two keys can't get the same position. such a situation is called hash collision or hash clash.

1.5 Methods of Dealing with Hash Clash

There are three basic methods of dealing with hash clash. They are:

- Chaining
- Rehashing
- Separate chaining.

CHAINING

It builds a link list of all items whose key has the same value. During search, this sorted linked list is traversed sequentially for the desired key. It involves adding an extra link field to each table position. There are three types of chaining

1. Standard Coalsced Hashing
2. General Coalsced Hashing
3. Varied insertion coalsced Hashing

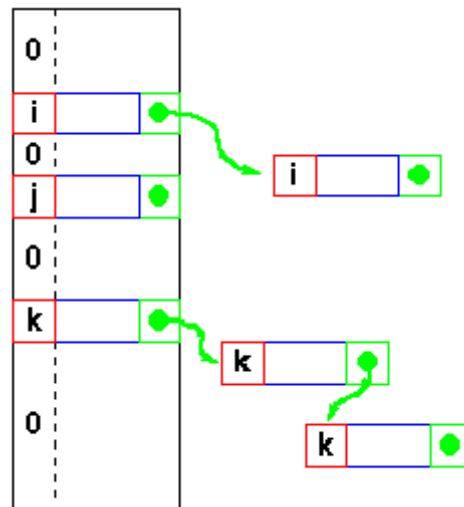
Standard Coalsced Hashing

It is the simplest of chaining methods. It reduces the average number of probes for an unsuccessful search. It efficiently does the deletion without affecting the efficiency

General Coalsced Hashing

It is the generalization of standard coalesced chaining method. In this method, we add extra positions to the hash table that can be used to list the nodes in the time of collision.

Varied insertion coalsced Hashing



It is the combination of standard and general coalesced hashing. Under this method, the colliding item is inserted to the list immediately following the hash position unless the list forming from that position containing a cellular element.

SOLVING HASH CLASHES BY LINEAR PROBING

The simplest method is that when a clash occurs; insert the record in the next available place in the table. For example in the table the next position 646 is empty. So we can insert the record with key 012345645 in this place which is

still empty. Similarly if the record with key %1000 = 646 appears, it will be inserted in next empty space. This technique is called linear probing and is an example for resolving hash clashes called rehashing or open addressing.

WORKING OF LINEAR PROBING ALGORITHM

It works like this: If array location $h(\text{key})$ is already occupied by a record with a different key, rh is applied to the value of $h(\text{key})$ to find the other location where the record may be placed. If position $rh(h(\text{key}))$ is also occupied, it too is rehashed to see if $rh(rh(h(\text{key})))$ is available. This process continues until an empty location is found. Thus we can write a search and insert algorithm using hashing as follows:

ALGORITHM

```
void insert( key, r )
typekey key; dataarray r;
{
    extern int n;
    int i, last;
    i = hashfunction( key );
    last = (i+m-1) % m;
    while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i].k!=key )
        i = (i+1) % m;
    if (empty(r[i]) || deleted(r[i]))
    {
        /*** insert here ***/
        r[i].k = key;
        n++;
    }
    else Error /*** table full, or key already in table ***/;}
```

DISADVANTAGES OF LINEAR PROBING

It may happen, however, that the loop executes forever. There are two possible reasons for this. First, the table may be full so that it is impossible to insert any

new record. This situation can be detected by keeping an account of the number of records in the table.

When the count equals the table size, no further insertion should be done. The other reason may be that the table is not full, too. In this type, suppose all the odd positions are empty and the even positions are full and we want to insert in the even position by $rh(i)=(i+2)\%1000$ used as a hash function. Of course, it is very unlikely that all the odd positions are empty and all the even positions are full.

However the rehash function $rh(i)=(i+200)\%1000$ is used, each key can be placed in one of the five positions only. Here the loop can run infinitely, too.

SEPARATE CHAINING

As we have seen earlier, we can't insert items more than the table size. In some cases, we allocate space much more than required resulting in wastage of space. In order to tackle all these problems, we have a separate method of resolving clashes called separate chaining. It keeps a distinct link list for all records whose keys hash into a particular value. In this method, the items that end with a particular number (unit position) is placed in a particular link list as shown in the figure. The 10's, 100's not taken into account. The pointer to the node points to the next node and when there is no more nodes, the pointer points to NULL value.

ADVANTAGES OF SEPERATE CHAINING

- No worries of filling up the table whatever be the number of items.
- The list items need not be contiguous storage
- It allows traversal of items in hash key order.

SITUATION OF HASH CLASH

What would happen if we want to insert a new part number 012345645 in the table. Using the hash function $key \%1000$ we get 645. Therefore for the part belongs in position 645. However record for the part is already being occupied by 011345645. Therefore the record with the key 012345645 must be inserted

somewhere in the table resulting in hash clash. This is illustrated in the given table:

POSITION	KEY	RECORD
0	258001201	
2	698321903	
3	986453204	
.		
.		
450	256894450	
451	158965451	
.		
.		
647	214563647	
648	782154648	
649	325649649	
.		
.		
997	011239997	
998	231452998	
999	011232999	

1.6 DOUBLE HASHING

A method of open addressing for a hash table in which a collision is resolved by searching the table for an empty place at intervals given by a different hash function, thus minimizing clustering. Double Hashing is another method of collision resolution, but unlike the linear collision resolution, double hashing uses a second hashing function that normally limits multiple collisions. The idea is that if two values hash to the same spot in the table, a constant can be calculated from the initial value using the second hashing function that can then be used to change the sequence of locations in the table, but still have access to the entire table.

Algorithm for double hashing

```
void insert( key, r )  
typekey key; dataarray r;  
{
```

```

extern int n;
int i, inc, last;
i = hashfunction( key );
inc = increment( key );
last = (i+(m-1)*inc) % m;
while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i].k!=key )
i = (i+inc) % m;
if ( empty(r[i]) || deleted(r[i]) )
{
/**/ insert here /**/
r[i].k = key;
n++;
}
else Error /**/ table full, or key already in table /***/;
}

```

Check Your Progress

Q. 1: Fill in the blanks

- i. Hashing involves applying a hashing algorithm to a data item, known as the.....
- ii.is a method to store data in an array so that sorting, searching, inserting and deleting data is fast.
- iii. double hashing uses a second hashing function that normally limits.....

1.7 Clustering

CLUSTERING

There are mainly two types of clustering:

Primary clustering

When the entire array is empty, it is equally likely that a record is inserted at any position in the array. However, once entries have been inserted and several hash clashes have been resolved, it doesn't remain true. For, example in the given above table, it is five times as likely for the record to be inserted at the position 994 as the position 401. This is because any record whose key

hashes into 990, 991, 992, 993 or 994 will be placed in 994, whereas only a record whose key hashes into 401 will be placed there. This phenomenon where two keys that hash into different values compete with each other in successive rehashes is called primary clustering.

CAUSE OF PRIMARY CLUSTERING

Any rehash function that depends solely on the index to be rehashed causes primary clustering

WAYS OF ELIMINATING PRIMARY CLUSTERING

One way of eliminating primary clustering is to allow the rehash function to depend on the number of times that the function is applied to a particular hash value. Another way is to use random permutation of the number between 1 and e , where e is (table size -1, the largest index of the table). One more method is to allow rehash to depend on the hash value. All these methods allow key that hash into different locations to follow separate rehash paths.

SECONDARY CLUSTERING

In this type, different keys that hash to the same value follow same rehash path.

WAYS TO ELIMINATE SECONDARY CLUSTERING

All types of clustering can be eliminated by double hashing, which involves the use of two hash function $h_1(\text{key})$ and $h_2(\text{key})$. h_1 is known as primary hash function and is allowed first to get the position where the key will be inserted. If that position is occupied already, the rehash function $rh(i, \text{key}) = (i + h_2(\text{key})) \% \text{table size}$ is used successively until an empty position is found. As long as $h_2(\text{key}_1)$ doesn't equal $h_2(\text{key}_2)$, records with keys h_1 and h_2 don't compete for the same position. Therefore one should choose functions h_1 and h_2 that distributes the hashes and rehashes uniformly in the table and also minimizes clustering.

DELETING AN ITEM FROM THE HASH TABLE:

It is very difficult to delete an item from the hash table that uses rehashes for search and insertion. Suppose that a record r is placed at some specific location. We want to insert some other record r_1 on the same location. We will have to insert the record in the next empty location to the specified original location. Suppose that the record r which was there at the specified location is deleted.

Now, we want to search the record r_1 , as the location with record r is now empty, it will erroneously conclude that the record r_1 is absent from the table. One possible solution to this problem is that the deleted record must be marked "deleted" rather than "empty" and the search must continue whenever a "deleted" position is encountered. But this is possible only when there are small numbers of deletions otherwise an unsuccessful search will have to search the entire table since most of the positions will be marked "deleted" rather than "empty".

1.8 DYNAMIC AND EXTENDIBLE HASHING

One of the serious drawbacks associated with hashing of external storage is its being insufficiently flexible. The contents of the external storage structure tend to grow and shrink unpredictably. The entire hash table structuring method that we have examined has a sharp space/time trade-off. Either the table uses a large amount of space for efficient access which results in wastage of large space or it uses a small amount of space and accommodates growth very poorly and sharply increasing the access time for overflow elements. So in order to tackle the above stated problems, we would like to develop a scheme that doesn't utilize too much extra space when a file is small but permits efficient access when it grows larger. Two such schemes are dynamic hashing and Extendible hashing.

DYNAMIC HASHING

Dynamic hashing is a hash table that grows to handle more items. The associated hash function must change as the table grows. Some schemes may shrink the table to save space when items are deleted.

EXTENDIBLE HASHING

A hash table in which the hash function is the last few bits of the key and the table refer to buckets. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

HASH TABLE REORDERING

When a hash table is nearly full, many items given by their hash keys are not at their specified location. Thus, we have to make a lot of key comparisons before finding such items. If an item is not in the table, entire hash table has to be searched. Then, we come to the conclusion that the key is not in the table. In order to tackle this situation, many techniques came forward.

AMBLE AND KNUTH METHOD

In this method, all the records that hash into same locations are placed in descending order (assuming that the NULLKEY is the smallest one). Suppose we want to search a key, we need not rehash repeatedly until an empty slot is found. As soon as an item whose key is less than the search key is found in the table, we come to the conclusion that the search key is not in the table. At the time of insertion, if we want to insert a key k , if the rehash accesses a key smaller than k , the associated record with k replaces it and the insertion process continues with the replaced key.

Note

The ordered hash table method can be used only in the technique in which a rehash depends only on the index and the key not in the technique in which a rehash function depends on the no of items the item is rehashed (Unless that number is kept in the table).

ADVANTAGES OF AMBLE AND KNUTH'S METHOD

It

reduces significantly the number of key comparisons necessary to determine that a key doesn't exist in the table.

DISADVANTAGES OF AMBLE AND KNUTH'S METHOD

- It doesn't change the average number of key comparisons required to find a key that is in the table
- The unsuccessful search needs same average number of probes as the successful search.
- Average number of probes in insertion is not reduced in ordered table.

BRENT'S METHOD

This technique involves rehashing the search argument until an empty slot is found. Then each of the keys in rehash path is itself rehashed to determine if placing one of those keys in an empty slot would require fewer rehashes. If this is the case the search argument replaces the existing key in the table and the existing key is inserted in its empty rehash slot.

BINARY TREE HASHING

Another method of reordering the hash table was developed by Gonnet and Munro and is called as binary tree hashing. It is seen as an improvement to Brent's algorithm. In this method, we assume to use double hashing. Whenever a key is inserted in the hash table, an almost complete binary tree is constructed. Figure below illustrates an example of such a tree in which the nodes are arranged according to the array representation of an almost complete binary tree. $node(0)$ is the root and $node(2*i+1)$ and $node(2*i+2)$ are the left and right children of $node(i)$ respectively. Each node of the tree contains an index into the hash table. In the explanation, $node(i)$ will be referred to as $index(i)$ and the key at that position is referred to as $k(-1)$.

HOW TO CONSTRUCT A TREE?

Firstly, we define the youngest right ancestor of $node(i)$ or $yra(i)$ as the node number of the father of the youngest son i.e. the right son. In the given figure,

yra(12) is 1, since it is the left son of its father node(6) and its father is also a left child. So the youngest ancestor of node(12) is node(3) and its father is node(1). Similarly, yra(10) is 2 and yra(18) is 4 and yra(14) is 3. If node(i) is the right son, yra(i) is defined as the node number of its father $(i-1)/2$. Thus, yra(15) is 7 and yra(13) is 6. If node(i) has no ancestor i.e. is a right son, yra(i) is defined as (-1). Thus, yra(16) is -1. The binary tree is constructed according to the node number. The table construction continues until a NULLKEY and an empty position is found in the table.

HOW TO CALCULATE yra(i)

As we have seen above, the entire algorithm depends on the routine yra(i). Fortunately, yra(i) can be calculated very easily. It can be derived directly using this method. Find the binary representation of (i+1). Delete all the trailing zero bits along with one bit preceding them. Subtract 1 from the result and you will get the resulting binary number to get the value of yra(i).

EXAMPLES

3. yra(11):

$$11+1=12$$

Binary representation: 1100.

Removing 100, we get 1, which is binary representation of 1.

Therefore, yra(11)=0.

4. yra(17):

$$17+1=18$$

Binary representation: 10010.

Removing 10, we get 100, which is binary representation of 4.

Therefore, yra(17)=3.

5. yra(15):

$$15+1=16$$

Binary representation: 010000.

Removing 10000, we get 0, which is binary representation of

0.

Therefore, yra(15)=-1.

HOW TO INSERT A KEY IN THE TABLE

Once the tree has been constructed, the keys along the path from the root to the last node are reordered in the hash table. Let, i be initialized to the last node of the tree. If $yra(i)$ is non-zero, $k(yra(i))$ and its associated record are shifted from the $table[index(yra(i))]$ to $table[index(i)]$ and i is reset to $yra(i)$. It is repeated until $yra(i)$ is -1 at which point insertion is complete.

EXAMPLE OF INSERTION

Suppose $yra(21)=10$ and $index(10)$ is j , the key and record from j is shifted to u which is the right child. Then suppose $yra(10)$ is 2, the record and key from position b is shifted to j which is the right child of index b . Finally since $yra(2)$ is -1, key is inserted in position b .

ADVANTAGE OF BINARY SEARCH TREE

Binary tree hashing yields results that are even closer to optimal than Brent's

Check Your progress

Q. 2: What is Dynamic Hashing?

Q. 3: Define extendible hashing

Q. 4: What is **Brent's Method**?

1.9 Answer to Check Your Progress

Ans to Q.1: i. hashing key ii. Hashing iii. multiple collisions.

Ans to Q.2: Dynamic hashing is a hash table that grows to handle more items. The associated hash function must change as the table grows. Some schemes may shrink the table to save space when items are deleted.

Ans to Q. 3: A hash table in which the hash function is the last few bits of the key and the table refer to buckets. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry

referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

Ans to Q. 4: This technique involves rehashing the search argument until an empty slot is found. Then each of the keys in rehash path is itself rehashed to determine if placing one of those keys in an empty slot would require fewer rehashes. If this is the case the search argument replaces the existing key in the table and the existing key is inserted in its empty rehash slot.

1.10 Model Questions

1. What is Direct Addressing? When is it used?
2. What is the advantage of using Hash Function over Direct Addressing? Explain.
3. When does a 'collision' occur? What are the methods to resolve it? Explain giving examples.
4. Explain the 'division method' for creating hash functions.
5. Differentiate between linear probing and quadratic probing.

Block-IV

UNIT XIII: SETS

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Bit Vector Representation
- 1.4 Linked list representation
- 1.5 Answer to Check Your Progress
- 1.6 Model Questions

1.1 Learning Objectives

After going this unit, the learner will able to learn about:

- Representation of a set is using an n-bit vector
- Representation of sets is using linked lists

1.2 Introduction:

- A set is a collection of distinct objects.
Example $S = \{ 2, 5, c, \text{ball}, \text{red}, 34.5 \}$
- In the above definition there is no restriction on objects. But, in many applications we deal with a set of objects of same type. For example, integers, characters, or strings.
- In this module, we discuss representation of sets whose element is integers in the range 1 between n.

1.3 Bit Vector Representation

A simple representation of a set is using an n-bit vector.

If i is a member of a set S then i th bit is 1 (or true), otherwise 0 (or false).

Example : $A = \{ 5, 3, 7, 9, 1 \}$

1	2	3	4	5	6	7	8	9	10
1	0	1	0	1	0	1	0	1	0

Fig: Representation of the set

- Inserting an integer x into a set can be done by changing the x th bit to 1.
- Similarly, deleting x is nothing but changing the x th bit to 0.
- Member checking, that is x is a member or not boils down to checking the x th bit. If x th bit is 1 then x is a member, otherwise not.

- That is, insert, delete, and member operation can be performed in constant time.
- Other operation like union, intersection, difference can be performed using logical bit operation.
- Assume, sets A, B, and C are represented using an arrays a, b, and c respectively of size n.
- Pseudocode for
Algorithm Union (a, b, c)
for i = 1 to n do
c[i] = a[i] OR b[i]
- Pseudocode for
Algorithm Intersection (a, b, c)
for i = 1 to n do
c[i] = a[i] AND b[i]
- Pseudocode for C = A - B
Algorithm Difference (a, b, c)
for i = 1 to n do
c[i] = a[i] AND (NOT b[i])
- These operations take time proportional to the n.
If n less than the size of a word in computer, the bit vector can be stored in a word. The operation Union, intersection, and difference can be performed using one word logical operation.

1.4 Linked list representation

- More general representation of sets is using linked lists. Each item in the list is a member of the set.
- The elements in the set need not be from any universal set.
- The size of the list proportional to the size of the set, but not proportional to the universal set.
- Assume, there exists a linear ordered relation $<$, amount the elements of the universal set.
- That is, for all x_i and x_j , exactly one of $x_i < x_j$, $x_i = x_j$, or $x_j < x_i$ is true.

- The relation $<$ is transitive, that is, for all x_i, x_j and x_k , if $x_i < x_j$ and $x_j < x_k$, then $x_i < x_k$.
- We store element in a set by the above linear ordered relation. That is, if $x_i < x_j$ then x_i is placed before x_j in the linked list.

$A = \{ 5, 3, 7, 9, 1 \}$

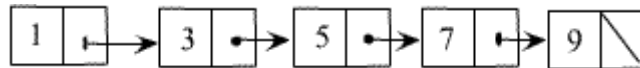


Fig 2: Set Representation using linked list

- So, operations inset, delete, and member are reduced to insert, delete, and search in the linked list of sorted elements. See module 3.
- Similarly, operations union, intersection, and difference also can reduce to linked list operations. This is left as an exercise, see problem.
- Another variation of linked lists representation of sets is using a pointer from each element to the set representing element.
- The structure of each object is shown below.

Struct Object

```
{
    Int data;
    Struct object      *nextobject;
    Struct object      *representative;
};
```

- The above structure describes that each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative.
- The first object in each linked list serves as its sets representative.
- Examples

Set1 = { a, b, c, d }

Set2 = { 1, 2, 3, 4 }

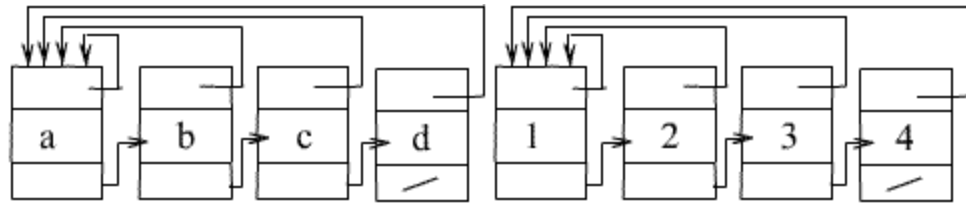


Fig 3: Set1

Fig 4: Set 2

- In Fig 3, ' a ' is the linked list representative for Set1 and in Fig4, '1' is the linked list representative for Set2.
- The function MAKE-SET(x) creates a new set whose only member is pointed by 'x'.

Object* MAKE-SET(x)

```

{
    Struct object *temp;
    temp= (struct object*)malloc(sizeof(struct object));
    temp->data = x;
    temp->representative=temp;
    return temp;
}

```

The function MAKE-SET(x) takes O(1) time

- The UNION operation using linked list representation takes significantly more time. We perform UNION (X,Y) by appending Y's list onto the end of X's list. We must update the pointer to the representative for each object originally on Y's List to X.

Void UNION (X,Y)

```

{
    For      each      object      a?Y
    For      each      object      b?X
    If(      b->data    a->data)
    {
        Add 'a' to X;

        a->representative = X;
    }
}

```

The following example shows the union of two sets for the previously mentioned Set1 and Set2. That is we are performing the operation UNION (a,1).

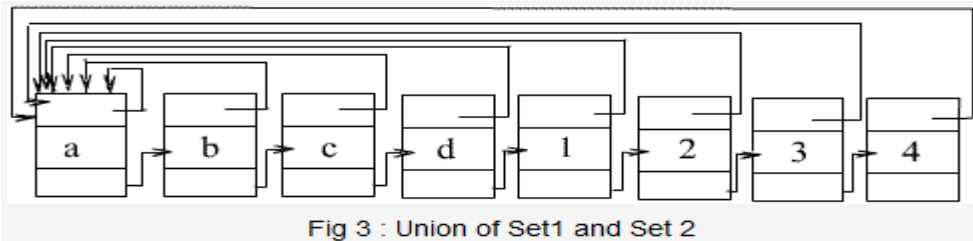


Fig 3 : Union of Set1 and Set 2

- The UNION operation takes $O(mn)$ time. Here 'm' and 'n' are the sizes of the given two lists.

Check Your Progress

Fill in the blanks

- Q.1 A is a collection of distinct objects.
- Q.2 The operation using linked list representation takes significantly more time.
- Q.3 Each item in the list is a member of the.....
- Q.4 A simple representation of a set is using.....

1.5 Answer to Check Your Progress

Ans to Q. 1: Set

Ans to Q. 2: UNION

Ans to Q. 3: Set

Ans to Q. 4: an n-bit vector.

1.6 Model Questions

1. What is Linked list representation? Explain.
2. Explain Bit Vector Representation.

UNIT XIV: STRING ALGORITHM

- 1.1 Learning Objective
- 1.2 Introduction
- 1.3 String Functions
 - 1.3.1 String length
 - 1.3.2 String concatenation
 - 1.3.3 String copy
 - 1.3.4 String matching
- 1.4 Pattern Matching
- 1.5 Brute Force String Matching algorithm.
- 1.6 Knuth-Morris-Pratt(KMP) string matching algorithm
- 1.7 Answer to Check Your Progress
- 1.8 Model Questions

1.1 Learning Objective

After going through this unit, the learner will able to learn about:

- String Algorithm
- String Copy
- Pattern Matching

1.2 Introduction

A string is a sequence of characters. In computer science, strings are more often used than numbers. We have all used text editors for editing programs and documents. Some of the Important Operations which are used on strings are: searching for a word, find -and -replace operations, etc.

1.3 String Function

There are many functions which can be defined on strings. Some important functions are

- a. String length: Determines length of a given string.

- b. String concatenation: Concatenation of two or more strings coping.
- c. String copy: Creating another string which is a copy of the original or a copy of a part of the original.
- d. String matching: Searching for a query string in given string.

1.3.1 STRING LENGTH

- Strings can have an arbitrary but finite length.
- There are two types of string data types:
 - a. Fixed length strings
 - b. Variable length strings
 - Fixed length strings have a maximum length and all the strings uses same amount of space despite of their actual size.
 - Variable length strings uses varying amount of memory depending on their actual size. Throughout of our discussion we assume that strings are of variable length type.
 - Variable length string is an array of characters terminated by a special character.
 - To find the length of a string we scan through the string from left to right until we find the special symbol and each time incrementing a counter to keep track of number of characters scanned so far.

ALGORITHM FOR String Length

We assume that the given string STR is terminated by special symbol '\0'.

```
length = 0, i=0; //Identity starts from '0'.
while STR[i] != '\0' //In C '\0' is used as end-of-string
markes.

    i++;
    length=i;
return length
```

1.3.2 STRING CONCATENATION

- Appending one string to the end of another string is called string concatenation

Example let STR1= "hello"

STR2= "world"

- If we concatenate STR2 with STR1, then we get the string "helloworld"

Algorithm

1. $i = 0, j = 0;$
2. while STR1[i] != '\0'

 $i++;$
3. while STR2[j] != '\0'

 STR1[i]= STR2[j];
 $i = i + 1$
 $j = j + 1$
4. STR1[i]= '\0';
5. Return STR1;

1.3.3 String Copy

- By string copy, we mean copying one string to another string character by character.
- The size of the destination string should be greater than equal to the size of the source string.

Algorithm

1. Set $i = 0$
2. while STR[i] != '\0'
3. {
 STR2[i]=STR1[i];
 $i = i + 1;$
 }
4. Set STR2[i]='\0'

5. Return STR2

1.4 Pattern Matching

STRING-MATCHING

- String matching is a most important problem.
- String matching consists of searching a query string (or pattern) P in a given text T .
- Generally the size of the pattern to be searched is smaller than the given text.
- There may be more than one occurrences of the pattern P in the text T . Sometimes we have to find all the occurrences of the pattern in the text.
- There are several applications of the string matching. Some of these are
 1. Text editors
 2. Search engines
 3. Biological applications
- Since string-matching algorithms are used extensively, these should be efficient in terms of time and space.
- Let $P [1..m]$ is the pattern to be searched and its size is m .
- $T [1..n]$ is the given text whose size is n
- Assume that the pattern occurs in T at position (or shift) i . Then the output of the matching algorithm will be the integer i where $1 \leq i \leq n - m$. If there are multiple occurrences of the pattern in the text, then sometimes it is required to output all the shifts where the pattern occurs.

Let Pattern $P = CAT$

Text = ABABNACATMAN

Then there is a match with the shift 7 in the text T

1	2	3	4	5	6	7	8	9	10	11	12
A	B	A	B	N	A	C	A	T	M	A	N
						C	A	T			

Check Your Progress

Fill in the Blanks

Q.1 A..... is a sequence of characters.

Q.2Determines length of a given string.

Q.3Concatenation of two or more strings coping.

Q.4Creating another string which is a copy of the original or a copy of a part of the original.

Q.5Searching for a query string in given string.

1.5 Brute Force String Matching algorithm.

- This algorithm is a simple and obvious one, in which we compare a given pattern P with each of the sub strings of the text T , moving from left to right, until a match is found
- Let S_i is the substring of T , beginning at the i th position and whose length is same as pattern P .
- We compare P , character by character, with the first substring S_1 . If all the corresponding characters are same, then the pattern P appears in T at shift 1. If some of the characters of S_1 are not matched with the corresponding characters of P , then we try for the next substring S_2 . This procedure continues till the input text exhausts.
- In this algorithm we have to compare P with $n-m+1$ substrings of T .

Example

- Let $P = abc$
 $T = aabab$
- Compare P with 1st substring of T

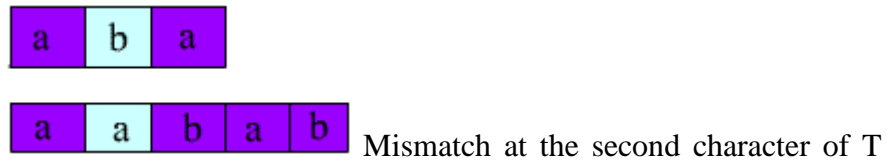


Fig: 2(a)

- Compare P with 2nd substring of T

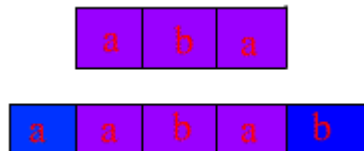


Fig: 2(b)

Since the corresponding characters are same, there is a match at shift 1.

- Compare P with 3rd substring of T

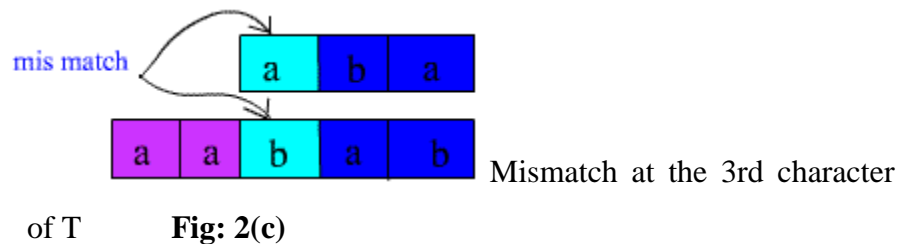


Fig: 2(c)

Algorithm For Brute-Force String matching

Let $P[0..m]$ is the given pattern and $T[0..n]$ is the text.

1. $i = 1$; [substring 1]
2. Repeat steps 3 to 5 while $i \leq n - m + 1$ do
3. for $j = 1$ to m [For each character of P]

If $P[j] \neq T[i+j-1]$ then
goto step 5

4. Print "Pattern found at shift i "
5. $i = i + 1$

6. exit

- The complexity of the brute force string matching algorithm is $O(nm)$
- On average the inner loop runs fewer than m times to know that there is a mismatch.
- The worst case situation arises when first m character are matched for all substrings S_i . If pattern is of the form $a^{m-1}b$ and text is of the form $a^{n-1}b$, where a^{n-1} denotes a repeated $n-1$ times. In this case the inner loop runs exactly for m times before knowing that there is a mismatch. In this situation there will be exactly $m*(n-m+1)$ number of comparisons.

1.6 Knuth-Morris-Pratt(KMP) string matching algorithm

- In brute force method, irrespective of the structure of the pattern P , if there is a mismatch, we restart the matching process with shift $s = s+1$.
- But if we know the structure of the pattern, we can intelligently avoid some number of shifts.

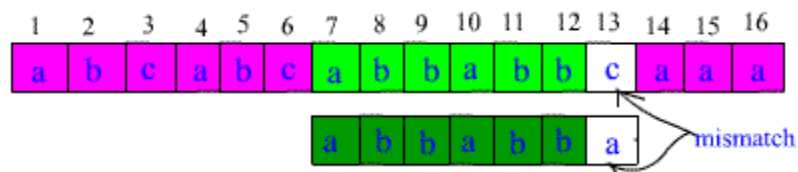


Fig: 3

- Suppose 6 characters are matched successfully and there is a mismatch at the 7th position. The shift $s = s+1$ is obviously invalid as the first pattern character 'a' would be aligned to the text character 'b', which is known to match the second pattern character.
- Also if we observe, the shift $s = s+2$ is also invalid.
- But the shift $s = s+3$ may be valid as first three characters of the pattern will be nicely aligned to the last three characters of the portion of the text which are already matched.

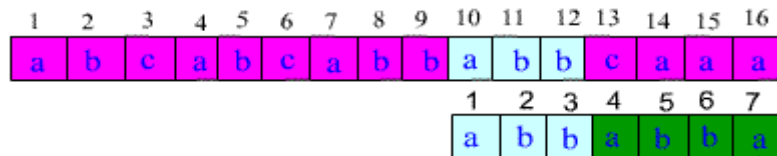


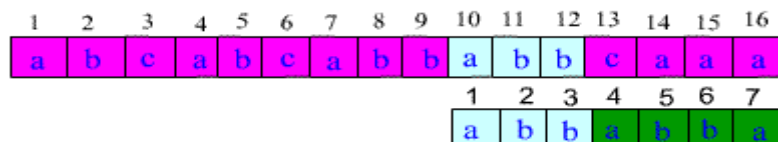
Fig: 4

- KMP (Knuth-Morris-pratt) algorithm avoids some redundant comparisons. When a mismatch occurs, the most we can shift the pattern so as to avoid redundant comparisons is the largest prefix of $P[1..j]$ that is also suffix of $P[2..j]$, where j is the number of characters that are known to be matched successfully before the mismatch.
- The amount of shift that is not necessarily invalid can be pre-computed by comparing the pattern against itself. For string matching process we need to find a "Failure Function" FF which will give the amount of shift that is not necessarily invalid. If j characters are already successfully matched and there is a mismatch position $j+1$, then the output of the FF is the largest prefix of $P[1..j]$ that is also a suffix of $P[2..j]$.

Algorithm for Failure function

FailureFunction(P)

1. $m = \text{stringlength}(P)$
2. $FF[1] = 0$
3. $k = 0$
4. for $j = 2$ to m
5. while $k > 0$ and $P[k+1] \neq P[j]$
6. do $k = FF[k]$
7. if $P[k+1] = P[j]$
8. then $k = k + 1$
9. $FF[j] = k$
10. return FF



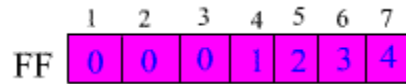


Fig: 5: failure function for the above pattern

The running time of FailureFunction is $O(m)$ where m is the length of the pattern P .

Algorithm for KMP string matcher

KMP string matching first preprocesses the given pattern P to compute Failure Function. Then the output of the Failure Function is used to skip some of the invalid comparisons. Note that the preprocessing is done once for each pattern. Since in general the size of the pattern to be searched is relatively small, it takes far less time than the string matching process.

KMP stringMatch (T, P)

1. $n = \text{stringlength}[T]$
2. $m = \text{length}[P]$
3. $FF = \text{FailureFunction}(P)$
4. $j = 0$
5. for $i = 1$ to n
6. while $j > 0$ and $P[j+1] \neq T[i]$
7. $j = FF[j]$
8. if $P[j+1] = T[i]$
9. then $j = j + 1$
10. if $j = m$
11. then print "Pattern found with shift " $i - m$
12. $j = FF[j]$

KMP algorithm runs in optimal time $O(m+n)$ time.

Check Your Progress

- Q. 6: How many types of string data types?
- Q. 7: What is Brute Force String Matching algorithm?
- Q. 8: Write Algorithm For Brute-Force String matching

1.7 Answer to Check Your Progress

Ans to Q.1: string

Ans to Q.2: String length:

Ans to Q.3: String concatenation:

Ans to Q.4: String copy

Ans to Q.5: String matching

Ans to Q. 6: There are two types of string data types:

- a. Fixed length strings
- b. Variable length strings

Fixed length strings have a maximum length and all the strings uses same amount of space despite of their actual size.

Variable length strings uses varying amount of memory depending on their actual size. Throughout of our discussion we assume that strings are of variable length type.

Variable length string is an array of characters terminated by a special character.

To find the length of a string we scan through the string from left to right until we find the special symbol and each time incrementing a counter to keep track of number of characters scanned so far.

Ans to Q.7: This algorithm is a simple and obvious one, in which we compare a given pattern P with each of the sub strings of the text T, moving from left to right, until a match is found

- Let S_i is the substring of T, beginning at the i th position and whose length is same as pattern P.
- We compare P, character by character, with the first substring S_1 . If all the corresponding characters are same, then the pattern P appears in T at shift 1. If some of the characters of S_1 are not matched with he corresponding characters of P, then we try for the next substring S_2 . This procedure continues till the input text exhausts.
- In this algorithm we have to compare P with $n-m+1$ substrings of T.

Ans to Q. 8: Let $P[0..m]$ is the given pattern and $T[0..n]$ is the text.

1. $i = 1$; [substring 1]
2. Repeat steps 3 to 5 while $i \leq n - m + 1$ do
3. for $j = 1$ to m [For each character of P]
If $P[j] \neq T[i+j-1]$ then
goto step 5
4. Print "Pattern found at shift i "
5. $i = i + 1$
6. exit

1.8 Model Questions

1. What is the complexity of the brute force string-matching algorithm in the best case?
2. Write a procedure to count the number of the time the word 'the' appears in a given text.
3. Find the output of The FailureFunction for the pattern $P =$
aabbaababbabaa
4. Give best case inputs (both pattern and text) for KMP string matching algorithm.

UNIT XV

PROGRAM DEVELOPMENT & PROGRAM TESTING AND VERIFICATION

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Life Cycle
- 1.4 Code Designing
- 1.5 Coding
- 1.6 Programming Style
- 1.7 Testing Method
- 1.8 Verification Procedure
- 1.9 Answer to Check Your Progress
- 1.10 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to learn about:

- Life Cycle
- Code Designing
- Coding
- Programming Style
- Testing Method
- Verification Procedure

1.2 Introduction

Program development life cycle (PDLC) The process containing the five phases of program development: analyzing, designing, coding, debugging and testing, and implementing and maintaining application software. In this Unit, we will discuss in detail.

1.3 Life Cycle

Software/program development is not just writing code, it is much more than that. It has a life cycle, which can be divided into following stages.

- **Requirements:** The description of what is required in general terms is called the requirements of the programmed.
- **Program Specification:** It is a complete description of what the program does.
- **Code Design:** Overall design of the programmed, including algorithms and data structures used, file formats, and modules definition.
- **Coding:** Writing code also involves standard coding practices like simplicity, clarity, commenting etc.
- **Testing:** Testing program for correctness.
- **Debugging:** Most programs fail first time. Finding the logical error and correcting the program is debugging process.
- **Documentation:** Writing a user manual for the program.
- **Maintenance:** After testing also programs are not perfect. Correcting bugs found in future is called the maintenance phase of the program.
- **Updating:** After using the program for some time, user may be need change program to add more futures or better algorithms and data structures.

The programmed life cycle is depicted in the below figure.

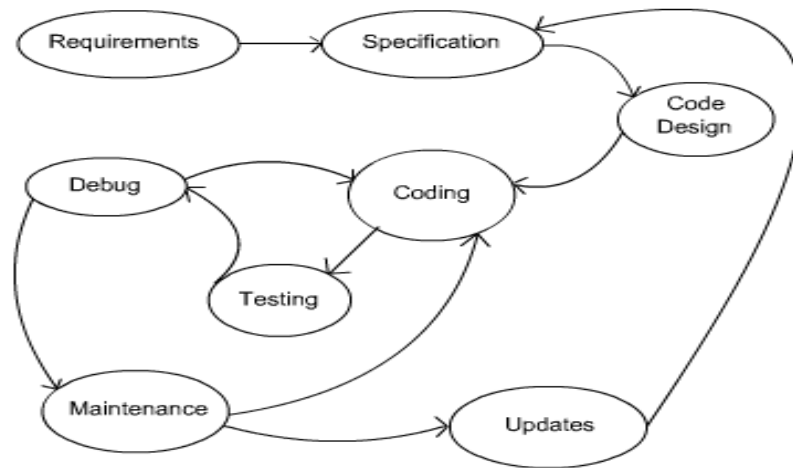


Fig.1

We discuss these phases for the following problem.

Develop a program to count number of character, words and lines of given text files.

- **Requirements:** Count number of character, words and lines in given text files.

- **Specification:** Program take text file names as arguments and output number of characters, words, and lines in each text file.

1. After making the first draft of the specifications, it can be given to user and/or circulated to people working on this project.
2. Update the specifications as per their feedback.
3. For example, what happens if a file does not exists?
4. Program has to give an error message if file does not exist and proceed to next file. This has to be added to specifications.

- **Revised Specification:** Program take text file names as arguments and output number of characters, words, and lines in each text file. If any file does not exist, it gives an error message and proceeds to next file.

1.4 Code Designing

- If the program is large, it is desirable to divide into logical sections called modules. Each module can be further subdivided into sub modules depending on their size. Since, developing and testing smaller modules or sub modules can be done separately and easily.

- If more than one programmer involved in development, the task of developing these modules can be divided among them.
- Some programming languages, like C, allow program can be split into multiple file. In such languages, each module or group of logically related modules can be developed in a set of files by programmers and tested independently.
- In this stage, we have to decide the file formats, data structures and algorithms required.
- We name our program "line-count". Idea in coding is to read one character at time till it reaches end of a word or a line then increase corresponding counter.
- The "line-count" program is divided into three modules, character counting, word counting, and line counting.
- These entire modules can be developed in a single file, since the program is simple.

1.5 Coding

- In this "line-count" program, we can develop a partial program to count number of words only. If it works correctly then we can extend to counting number line. This is because, code for both are very similar.
- Where there are similar tasks, it is better to write code for any one task and test it. Then extend to other tasks.
- Around more than 70% of time is spend on maintaining, upgrading, and debugging the program. So it is important to writing a simple and easy to read program. A good programming style is an important aspect in the programming development.

1.6 Programming Style

The following aspects are important for a good programming style.

- **Comments:** Write comments to explain everything programmer has to know. Depending on the importance of the comment, one can format

differently to get the program reader attention. For example most important comments can be kept in a comment box as shown below.

```

/ *** ----- ***
*** This program computes ***
*** number of lines, words ***
*** and character for the ***
*** input text file. ****
*** ----- ***/

```

- Less important comment can be shown as follows.

```

/****-----
           comments for each source file
-----**/

```

This type of comments use for beginning of a function.

```

/*****
*****

Comments beginning of each function
*****
*****

```

This is the begining of a section. It describe that how it works.

```

/***** Less Important Comments *****/

```

This type of comments use for less important message. For example, explaining how the next line work.

```

/----->> Simple comments explaining the
Next line <<-----/

```

- As per the requirements, one can have many different levels of comments.
- At the beginning of the program in main source file can contain boxed comment containing the heading, author, purpose, usage, references, file formats, limitations, and other related information.
 - Put comment on each declaration.

- Put comments to explain everything programmer has to know.
- Indentation: Indent the program for readability. Now-a-days, many editors are coming with automatic program indentation.
- Simplicity: Many cases, simplicity in coding gives better and easy understanding of the program. Some of simplicity rules are given below.
 - Function should not be too large,
 - Avoid complex logic.
 - Keep expressions simple and short.

Check Your Progress

Fill in the blanks

Q.1:is the description of what is required in general terms is called the requirements of the programmed.

Q.2:.....is a complete description of what the program does.

Q.3:..... is the overall design of the programmed, including algorithms and data structures used, file formats, and modules definition.

Q.4:is the program for correctness.

1.7 Testing Method

Important commands of gdb:

- `break [file :] function` : Set a breakpoint at the beginning of function (in file) .
- `break [file:] #n` : Set a breakpoint at line #n (in file) .
- `run [arglist]` : Start your program (with command line arglist , if any).
- `next`: Execute next line of the program (after stopping). That is, executes next line completely, including function calls (if any). It is called step over any function calls in the line.
- `c` : Continue executing the program (after stopping) till next breakpoint, if any.
- `print expr` : Display the value of an expression.
- `list [file :] function` : Type the text of the program in the vicinity of where it is presently stopped.

- step: Execute next program line (after stopping). If there is any function call present in the next program line, it goes into 1st line of that function. It is called step into any function calls in the line.
- quit : Exit from GDB.
- q : Exit from GDB.
- help : Show information about GDB commands.
- help [name] : Show information about GDB command name.

1.8 Verification Procedure

Palindrome problem:

A palindrome is a string that reads same in either direction. A palindrome can be string, phrase, numbers, or any sequence of symbols.

We debug the following program (palindrome.c) which takes a string as command line argument and tests whether it is a palindrome or not.

- Compile the program `gcc -g -o palindrome palindrome.c`
- It generates an executable program named `palindrome`.
- Start the debugger with the name of executable program to be debugged.
In our case, command is `gdb palindrome`
- The debugger outputs its prompt (gdb)
`$ gdb palindrome (gdb)`
- We keep first break point at the first instruction of the main function that is the line 7, to stop the execution immediately after started.
`(gdb) break main`
Breakpoint 1 at 0x80483da: file palindrome.c, line 7.
`(gdb)`
- Start the execution by the command `run madam`
`(gdb) run madam`
Starting program: `palindrome madam`
Breakpoint 1, main (argc=2, argv=0xafb79394) at
`palindrome.c:7`
`if (argc != 2) {`

(gdb)

- Program is stopped at the line 7.
- Now we can keep more break points or execute the program step by step.
- We will go for keeping more break points. We have to decide where to keep break points?
- To see the part of the code around the breakpoint use the command list. It displays 10 lines around the current breakpoint.

```
1. (gdb) list
2. #include<string.h>
3. main(int argc, char *argv[])
4. {
5.     int i,length, mid;
6.     if (argc != 2) {
7.         printf("Error in input.\n");
8.         exit(1);
9.     }
```

- It may not be necessary to keep the break point in the above part of the program. So we will see the code around line 15.

(gdb) list 15

```
    }

1.     length = strlen(argv[1]);
2.     mid = length/2;
3.     for (i=0; i<=mid; ++i){
4.         if (argv[1][i] != argv[1][length-i]){
5.             printf("\"%s\" is not a palindrome.\n", argv[1]);
6.             exit(0);
7.         }
8.     }
```

(gdb)

- There could be an error in the for loop. So, keep break point at the line 15.

(gdb) break 15

Breakpoint 2 at 0x8048435: file palindrome.c, line 15.

(gdb)

- Continue execution using the command `c` .

(gdb) c

Continuing.

Breakpoint 2, main (argc=2, argv=0xafb79394) at palindrome.c:15

```
15      if (argv[1][i] != argv[1][length-i]){
```

- (gdb)
- Program giving wrong output, that it is executing lines 16 and 17, which is in the if part.
- We have to check why if condition is satisfied?
- Use the print command to display the values stored in variables or expression.

```
(gdb) print length
```

```
$1 = 5
```

```
(gdb)
```

- Length of the input string “madam” is correct. Check other variables.

```
(gdb) print mid
```

```
$2 = 2
```

```
(gdb)
```

- Value stored in mid also correct. Check left and right expression of the if condition.

```
(gdb) print argv[1][i]
```

```
$3 = 109 'm'
```

```
(gdb) print argv[1][length-i]
```

```
$4 = 0 '\0'
```

```
(gdb)
```

- The left expression is 'm' and the right expression is '\0' which are not equal!
- Why? It is comparing 'm' and '\0'. It is supposed to compare 'm' and 'm', which are first and last character of the input string. Check indexes used to compare? That is, i and length - i .

```
(gdb) print i
```

```
$5 = 0
```



```
(gdb) print length-i
```

```
$6 = 5
```

```
(gdb)
```

- Index of an array of length 5 varies between 0 and 4. But we are comparing the character stored in the indexes 0 and 5, which is not correct.
- Index should be used in the right expression should be length-i-1.
- Quit the gdb.

```
(gdb) q
```

```
The program is running. Exit anyway? (y or n) y
```

```
$
```

- Edit the program, compile, and execute.
- Now, the program works correctly.

Run time Errors:

- Correcting run time errors are easier than correcting logical errors.
- We discuss most important run time errors.
- Segmentation fault: It occurs when a program try to access a memory location that it is not allowed to access.
 - 1 This location may part of the operating system, other user, or other processes.
- Divide by zero: Many operating systems report message Floating point exception, when there is expression which denominator of the division is 0.
- Stack Over flow: Most of the cases it happen, when there is infinite recursion. In recursive function calls, program has to store various environment variables, before start executing the calling function. It can also happen when there are many big temporary arrays.
- Whenever these errors occur, program execution stops with appropriate error message.
- These errors also can be debugged using gdb.

Check Your Progress

Fill in the blanks

Q.5: A is a string that reads same in either direction.

Q.6: A can be string, phrase, numbers, or any sequence of symbols.

Q.7: Write the commands of gdb.

1.9 Answer to Check Your Progress

Ans to Q.1: Requirements

Ans to Q.2: Program Specification

Ans to Q.3: Code Design

Ans to Q.4: Testing

Ans to Q.5: palindrome

Ans to Q.6: palindrome

Ans to Q.7:

- `break [file :] function` : Set a breakpoint at the beginning of function (in file) .
- `break [file:] #n` : Set a breakpoint at line #n (in file) .
- `run [arglist]` : Start your program (with command line arglist , if any).
- `next`: Execute next line of the program (after stopping). That is, executes next line completely, including function calls (if any). It is called step over any function calls in the line.
- `c` : Continue executing the program (after stopping) till next breakpoint, if any.
- `print expr` : Display the value of an expression.
- `list [file :] function` : Type the text of the program in the vicinity of where it is presently stopped.
- `step`: Execute next program line (after stopping). If there is any function call present in the next program line, it goes into 1 st line of that function. It is called step into any function calls in the line.
- `quit` : Exit from GDB.
- `q` : Exit from GDB.
- `help` : Show information about GDB commands.
- `help [name]` : Show information about GDB command name .

1.10 Model Questions

1. For each of the following programming assignment, follow programming life cycle to give from requirements to coding.
 - i. Write a program for four operator calculator.
 - ii. Write a program to find the day of the given date.
 - iii. Write a program to find all prime number in a given range of integers.
 - iv. Write a program for library user administration.
 - v. Write a program for air-line reservation.
2. Take any of your programs and run it using a interactive debugger and examine intermediate values.

Referencing

1. <http://eslm.kkhsou.ac.in/>
2. <http://ycpcs.github.io/cs360-spring2015/lectures/lecture07.html>
3. https://wiki.georges.savound.com/index.php/AVL_Tree
4. https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Hash_tables_and_hashing