II YEAR

III SEMESTER

COURSE CODE: 7BCE3C1

CORE COURSE-V–DATA STRUCTURES AND COMPUTER ALGORITHMS

Unit I

Introduction to data structure: The need for data Structure-Definitions-Data Structures-Arrays: Introduction, range of an array-one dimensional array-two dimensional array- special types of matrices-linked lists: Introduction - benefits and limitations of linked list-Types- singly linked lists-circular linked lists-doubly linked lists.

Unit II

Stack: Introduction- ADT stack - implementation of stack- application of stack - **Queue:** Introduction - implementation of basic operations on array based and linked list based queue - circular Queues.

Unit III

Trees: Introduction–binary Trees-representation of binary trees-Binary tree Traversals - Recursive procedures of traversal methods-Expression Trees-Threaded Trees-Application of Trees.

Unit IV

Algorithms: Introduction: What is an Algorithm? – Algorithm Specification – Performance Analysis – Divide and Conquer: General method – Binary Search – Finding the maximum and minimum – Merge Sort – Quick Sort – Selection –Strassen's Matrix Multiplication.

Unit V

The Greedy Method: General Method – Knapsack problem – Job Sequencing with deadlines – Optimal Storage on tapes – Optimal merge patterns

Minimum cost spanning trees: Prim's Algorithm – Kruskal Algorithm – Dynamic Problem: All pairs of shortest path – single source shortest path-Travelling salesman problem.

Graph: Graph terminology-connecteed graph-graph traversal techniques-

Text Books:

1. Data Structures, A. Chitra, P. T. Rajan, Vijay Nicol Imprints Pvt Ltd, 2006, McGrawHillEducation of India Pvt Ltd.
   UNIT I – Chapter 1, 3 (Except Multi-dimensional Arrays) and 4 (Except Simple Algorithms on linked lists, Circular doubly linked lists, multiple linked lists, applications, polynomial representation, polynomial addition, representation of polynomials)
   UNIT II – Chapters 5, 6 (Except Tower of Hanoi, Dequeue)
   UNIT III – Chapters (Except Priority Queues)
2. Fundamentals ofComputer Algorithms, Ellis Horrowitz, SaratajSahni, Galgottia Publications Pvt Ltd, New Delhi
   UNIT IV – Chapter 1 (Except 1.4), Chapter 3 (Except 3.2, 3.9)
   UNIT V – Chapter 4 (Except 4.2, 4.6.3)

Books for Reference:

1. Data Structure and Algorithm Analysis in C – Mark Allen Weiss – Second Edition, Addison Wesley publishing company, 1997.
2. C and C++ Programming concepts and Data Structures, P.S.Subramanyam, BS Publications, 2013.
3. Data Structures and Algorithms, Alfred V.Aho, John E.Hopcraft and Jeffrey D.Ullman, Pearson Education, Fourteenth Impression, 2013.

DEPARTMENT OF COMPUTER SCIENCE

SEMESTER: III

7BCE3C1- DATA STRUCTURES AND COMPUTER ALGORITHAMS

Unit - I

Introduction: Basic Terminology, Elementary Data Organization, Algorithm, Efficiency of an Algorithm, Time and Space Complexity, Asymptotic notations: Big-Oh, Time-Space trade-off.

Abstract Data Types (ADT)

Arrays: Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order, Application of arrays, Sparse Matrices and their representations.

Linked lists: Array Implementation and Dynamic Implementation of Singly Linked Lists, Doubly Linked List, Circularly Linked List, Operations on a Linked List. Insertion, Deletion, Traversal, Polynomial Representation and Addition, Generalized Linked List.

Unit – II

Stacks: Abstract Data Type, Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack in C, Application of stack: Prefix and Postfix Expressions, Evaluation of postfix expression, Recursion, Tower of Hanoi Problem, Simulating Recursion, Principles of recursion, Tail recursion, Removal of recursion Queues, Operations on Queue: Create, Add, Delete, Full and Empty, Circular queues, Array and linked implementation of queues in C, Dequeue and Priority Queue.

Unit – III

Trees: Basic terminology, Binary Trees, Binary Tree Representation: Array Representation and Dynamic Representation, Complete Binary Tree, Algebraic Expressions, Extended Binary Trees, Array and Linked Representation of Binary trees, Tree Traversal algorithms: Inorder, Preorder and Postorder, Threaded Binary trees, Traversing Threaded Binary trees, Huffman algorithm.

Unit – IV

Graphs: Terminology, Sequential and linked Representations of Graphs: Adjacency Matrices, Adjacency List, Adjacency Multi list, Graph Traversal : Depth First Search and Breadth First Search, Connected Component, Spanning Trees, Minimum Cost Spanning Trees: Prims and Kruskal algorithm. Transistive Closure and Shortest Path algorithm: Warshal Algorithm and Dijikstra Algorithm, Introduction to Activity Networks.

Unit – V

Searching : Sequential search, Binary Search, Comparison and Analysis Internal Sorting: Insertion Sort, Selection, Bubble Sort, Quick Sort, Two Way Merge Sort, Heap Sort, Radix Sort, Practical consideration for Internal Sorting.Search Trees: Binary Search Trees(BST), Insertion and Deletion in BST, Complexity of Search Algorithm, AVL trees, Introduction to m-way Search Trees, B Trees & B+ Trees .Hashing: Hash Function, Collision Resolution Strategies Storage Management: Garbage Collection and Compaction.

References:

1. Aaron M. Tenenbaum,YedidyahLangsam and Moshe J. Augenstein "Data Structures Using C and C++", PHI Learning Private Limited, Delhi India.
2. Horowitz and Sahani, "Fundamentals of Data Structures", *Galgotia Publications* Pvt Ltd Delhi India.
3. A.K. Sharma ,Data Structure Using C, Pearson Education India.
4. Rajesh K. Shukla, "Data Structure Using C and C++" Wiley Dreamtech Publication.
5. Lipschutz, "Data Structures" Schaum's Outline Series, Tata Mcgraw-hill Education (India) Pvt. Ltd .
6. Michael T. Goodrich, Roberto Tamassia, David M. Mount "Data Structures and Algorithms in C++", Wiley India.
7. P.S. Deshpandey, "C and Datastructure", Wiley Dreamtech Publication.
8. R. Kruse etal, "Data Structures and Program Design in C", Pearson Education
9. Berztiss, A.T.: Data structures, Theory and Practice :, Academic Press.
10. Jean Paul Trembley and Paul G. Sorenson, "An Introduction to Data Structures with applications", McGraw Hill.

Unit -I

Introduction to Data Structure and it's Characteristics

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.

Need of Data Structures

Three types of function

1. Storing
2. Accessing
3. Manipulation

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store; If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Abstract data type(ADT)

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc

ARRAY

An array refers to a collection of homogeneous element, stored in contiguous locations in

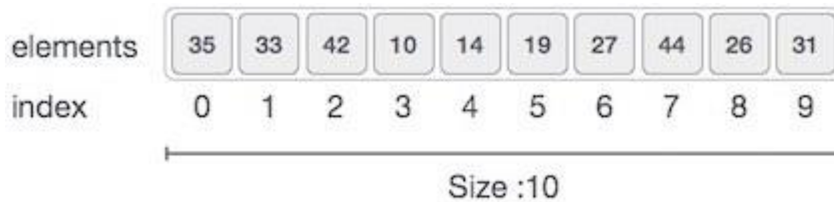
memory

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

```
int arr[10]; char arr[10]; float arr[5]
```

Range of an Array

Range of an array is the difference between the maximum and minimum element in an **array**, Input and Output Format: Input consists of n+1 integers where n corresponds to the number of elements in the **array**. The first integer corresponds to n and the next n integers correspond to the elements in the **array**.

Following are the few example of array declarations in C

1. Int data[10]
    a. Array of 10 integer values
    b. Lower bound 0,Upper bound 9 and range 10

2. Char code[20]
    a. Array of 10 Charcter values
    b. Lower bound 0,Upper bound 19 and range 20

Access Array Elements

Array indexing is the same as **accessing** an **array element**. You can **access** an **array element** by referring to its index number. The indexes in NumPy **arrays** start with 0, meaning that the first **element** has index 0, and the second has index 1 etc.

For example in the array data[10], the elements are accessed as data[0].data[1],…..data[9].

The array may be categorized into –

- One dimensional array
- Two dimensional array
- Multidimensional array

One Dimensional Array

In c programming language, one dimensional arrays are used to store list of values of same datatype. In other words, one dimensional arrays are used to store a row of values. In one dimensional array, data is stored in linear form. one dimensional arrays are also called as **one-dimensional arrays**, **Linear Arrays** or simply **1-D Arrays**.

Declaration of One Dimensional Array

We use the following general syntax for declaring a single dimensional array...

datatype arrayName [size] ;

```
int rollNumbers [60] ;
```

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each with the name **rollNumbers** and tells the compiler to allow only integer values into those memory locations.

Initialization of OneDimensional Array

We use the following general syntax for declaring and initializing a single dimensional array with size and initial values.

datatype arrayName [size] = {value1, value2, ...} ;

Example Code

```
int marks [6] = { 89, 90, 76, 78, 98, 86 } ;
```

The above declaration of single dimensional array reserves 6 contiguous memory locations of 2 bytes each with the name **marks** and initializes with value 89 in first memory location, 90 in second memory location, 76 in third memory location, 78 in fourth memory location, 98 in fifth memory location and 86 in sixth memory location.

We can also use the following general syntax to initialize a single dimensional array without specifying size and with initial values...

datatype arrayName [ ] = {value1, value2, ...} ;

The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

Example Code

```
int marks [] = { 89, 90, 76, 78, 98, 86 } ;


char studentName [] = "btechsmartclass" ;
```

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and arrayName will be a valid C identifier. A two- dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Representation of two dimensional arrays in memory

A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

Row Major Order: First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.

.

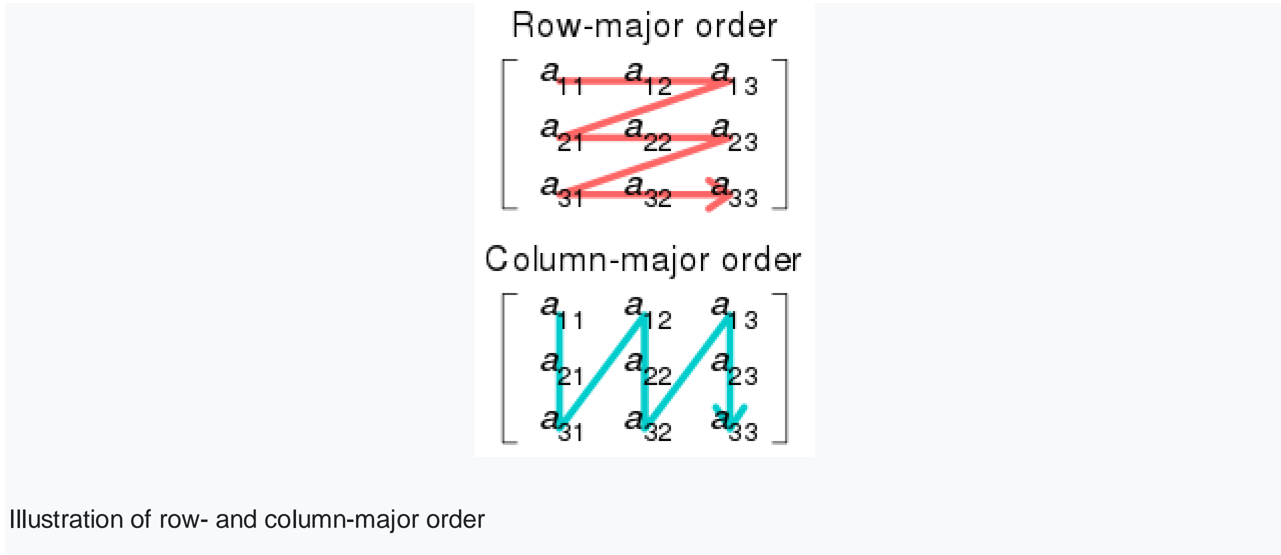


Illustration of row- and column-major order

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

In the row-major order layout (adopted by C for statically declared arrays), the elements in each row are stored in consecutive positions and all of the elements of a row have a lower address than any of the elements of a consecutive row:

1	2	3	4	5	6	7	8	9

In column-major order (traditionally used by Fortran), the elements in each column are consecutive in memory and all of the elements of a column have a lower address than any of the elements of a consecutive column:

1	4	7	2	5	8	3	6	9

To determine element address A[i,j]:

Location (A[i,j]) =Base Address + ((i*COL) + j)*e

For example:

Given an array [1…5] of integers. Calculate address of element a[2][3], where

BA=100. Sol

Location int A [2,3]) = BA + (2 x 5) +3)*e

= 100+ 13*2
= 100+ 26

= 126

Column Major Order: Order elements of first column stored linearly and then comes elements of next column.

To determine element address A[i,j]:
Location (A[i,j]) =Base Address + ((j*ROW) + i)*e

For example:

Given an array [1…5] of integers. Calculate address of element a[2][3], where

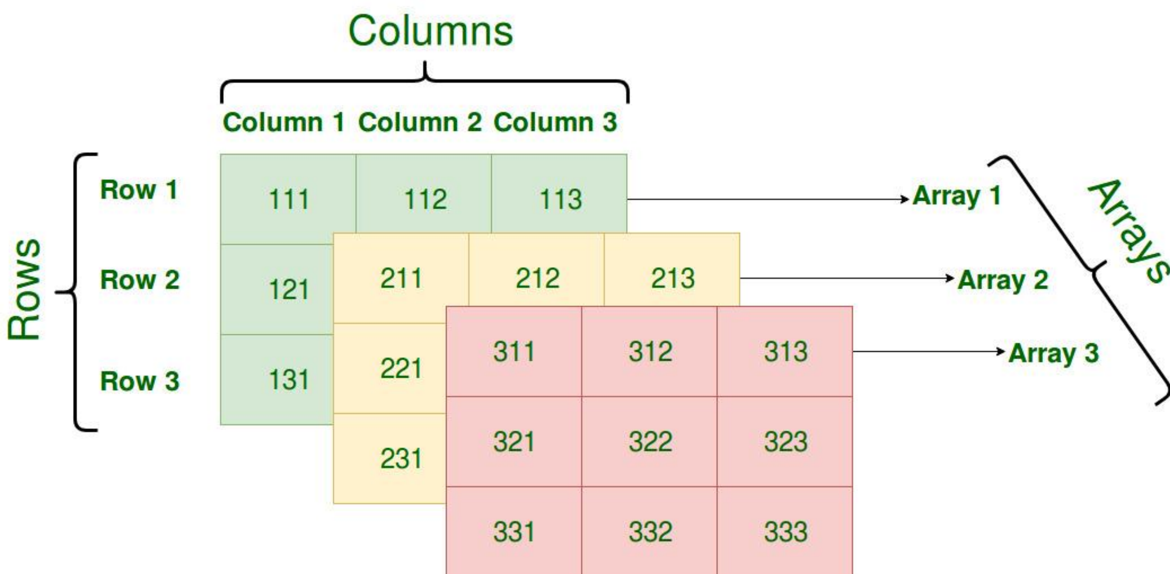
BA=100. Sol

Location int A [2,3]) = BA + (3 x 5) +2*e

= 100+ 17*2
= 100+ 34

= 134

Multi-Dimensional Array



Initializing Multi-Dimensional Array: Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18, 19,
                  20, 21, 22, 23};
```

A multidimensional arrays is defined as A[lb1……..ub1], [lb2……..ub2]…………….. [lbn…..ubn]

Special Types of Matrices

- Diagonal: M(i,j) = 0 for i ≠ j.
- Tridiagonal: M(i,j) = 0 for |i-j| < 1.
- Lower triangular: M(i,j) = 0 for i < j.
- Upper triangular: M(i,j) = 0 for i > j.
- Symmetric M(i,j) = M(j,i) for all i and j.

$$\begin{matrix}2&0&0&0\\0&1&0&0\\0&0&4&0\\0&0&0&6\end{matrix}$$

(a) Diagonal

$$\begin{matrix}2&1&0&0\\3&1&3&0\\0&5&2&7\\0&0&9&0\end{matrix}$$

(b) Tridiagonal

$$\begin{matrix}2&0&0&0\\5&1&0&0\\0&3&1&0\\4&2&7&0\end{matrix}$$

(c) Lower triangular

$$\begin{matrix}2&1&3&0\\0&1&3&8\\0&0&1&6\\0&0&0&0\end{matrix}$$

(d) Upper triangular

$$\begin{matrix}2&4&6&0\\4&1&9&5\\6&9&4&7\\0&5&7&0\end{matrix}$$

(e) Symmetric

The diagonal of a square matrix helps define two type of matrices: **upper-triangular** and **lower-triangular**. Indeed, the diagonal subdivides the matrix into two blocks: one above the diagonal and the other one below it. If the lower-block consists of zeros, we call such a matrix **upper-triangular**. If the upper-block consists of zeros, we call such a matrix **lower-triangular**. For example, the matrices are upper-riangular,

A square matrix $[a_{ij}]$ is called an **upper triangular matrix**, if $a_{ij} = 0$, when i > j.

is an upper triangular matrix of order 3 x 3.

$$\begin{pmatrix} a & 0 \\ c & d \end{pmatrix} \text{ and } \begin{pmatrix} a & 0 & 0 \\ d & e & 0 \\ g & h & k \end{pmatrix}$$

are lower-triangular. Now consider the two matrices

A square matrix is called a **lower triangular matrix**, if $a_{ij} = 0$ when i < j.

$$A = \begin{pmatrix} a & 0 & 0 \\ d & e & 0 \\ g & h & k \end{pmatrix} \text{ and } B = \begin{pmatrix} a & d & g \\ 0 & e & h \\ 0 & 0 & k \end{pmatrix}.$$

Tridiagonal matrix

In linear algebra, a *tridiagonal matrix* is a band matrix that has nonzero elements on the main diagonal, the first diagonal below this, and the first diagonal above the main diagonal only.

For example, the following matrix is tridiagonal:

$$\begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 5 & 2 & 7 \\ 0 & 0 & 9 & 0 \end{pmatrix}$$

The determinant of a tridiagonal matrix is given by the **continuant** of its elements.

Sparse matrix

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

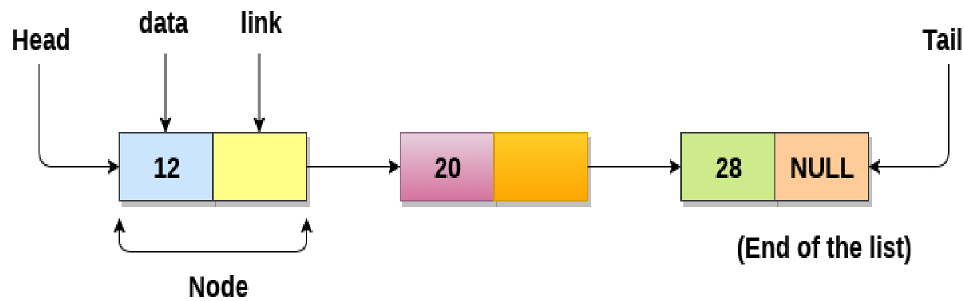
LINKED LIST

INTRODUCTION

- ❖ Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- ❖ A node contains two fields i.e. data stored at that particular addressand the pointer which contains the address of the

next node in thememory.

- The last node of the list contains pointer to the null.



BENEFITS AND LIMITATIONS OF LINKED LIST BENEFITS

Dynamic Data Structure

Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give initial size of linked list.

Insertion and Deletion

Insertion and deletion of nodes are easier. In linked list we just have to update the address present in next pointer of a node.

No Memory Wastage

As size of linked list can increase or decrease at run time so there is no memory wastage. In linked list the memory is allocated onlywhen required.

Implementation

Data structures such as stack and queues can be easily implemented using linked list.

LIMITATIONS

Memory Usage

More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

Traversal

Elements or nodes traversal is difficult in linked list. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large

Reverse Traversing

In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

Types of linked list

1. Singly Linked list
2. Double linked list
3. Circular linked list

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node.

A single linked list is shown in figure 3.2.1.

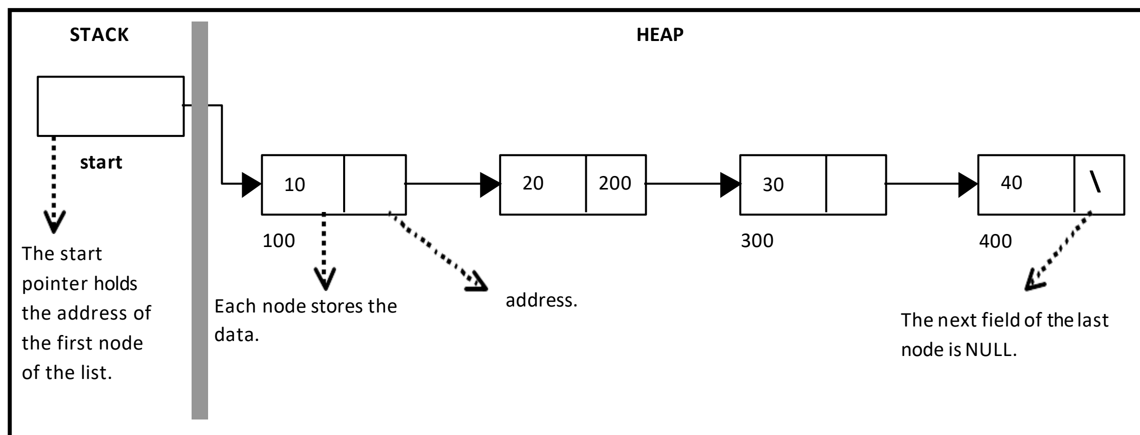


Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list.

Implementation of Single Linked List:

we need to create a **start** node**,** used to create and access other nodes in the linked list.

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialise the start pointer to be NULL.

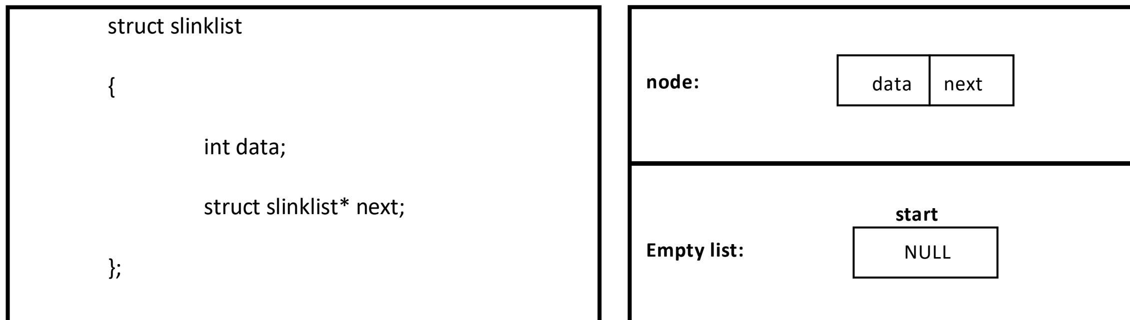


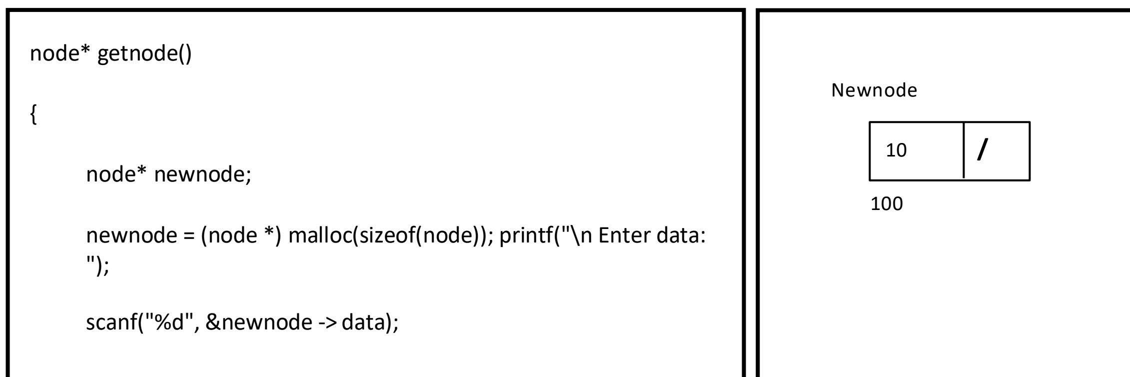
Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.



Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node , before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  newnode = getnode();

- If the list is empty then *start = newnode.*

- If the list is not empty, follow the steps given below:
  newnode -> next = start;
  start = newnode;

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

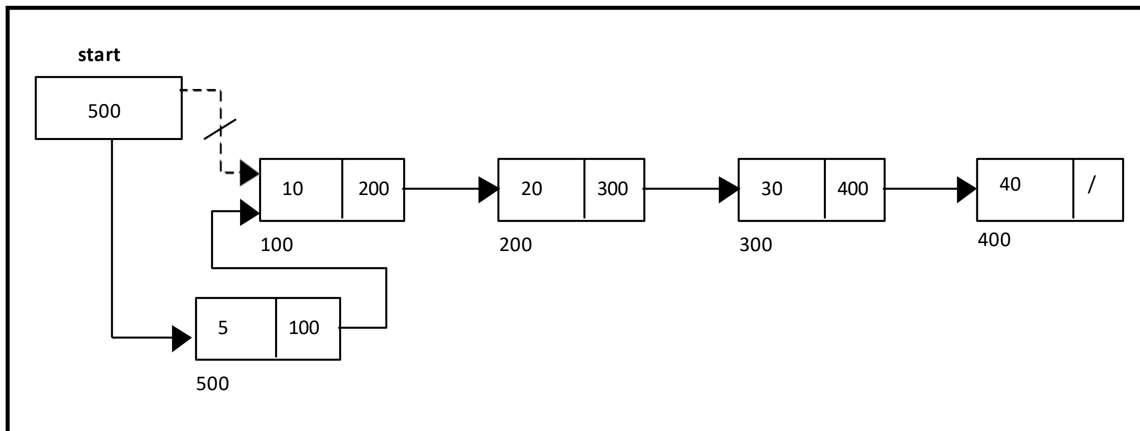


Figure 3.2.5. Inserting a node at the beginning

The function insert_at_beg(), is used for inserting a node at the beginning

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
node *newnode;
newnode = getnode();
if(start == NULL)
{
start = newnode;
}
else
{
newnode -> next = start;
start = newnode;
}
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

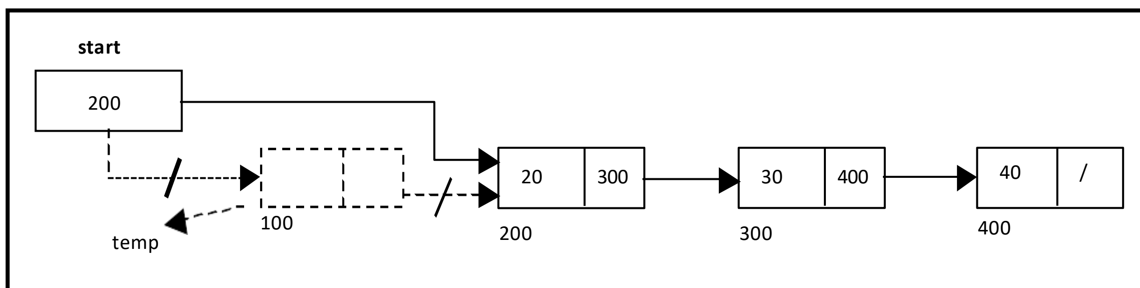


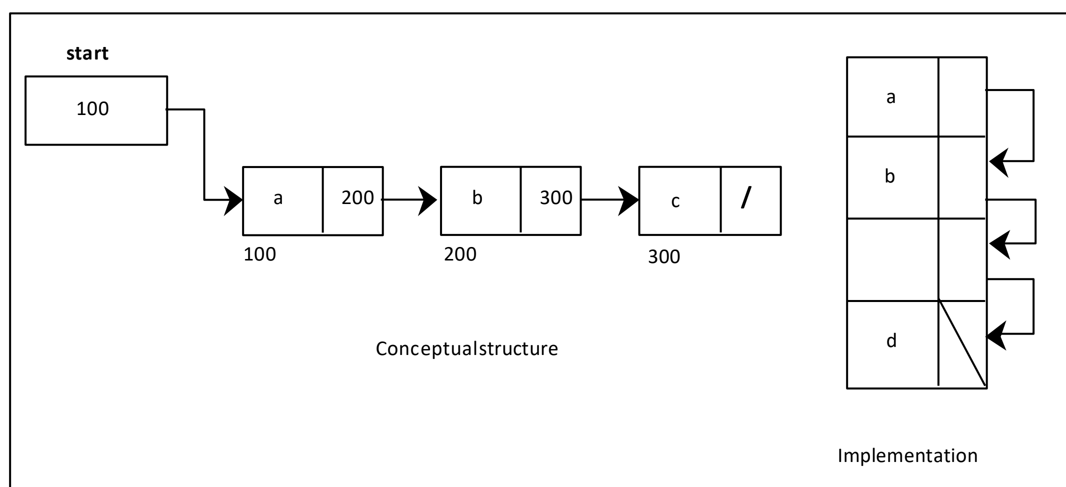
Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
node *temp;
if(start == NULL)
{
                printf("\n No nodes are exist..");
}               return ;
else
{
                temp = start;
                start = temp -> next; free(temp);
                printf("\n Node deleted ");
}

}
```

Array based linked lists:

- Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list.
- .



Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.
.
The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

double linked list is shown in figure 3.3.1.

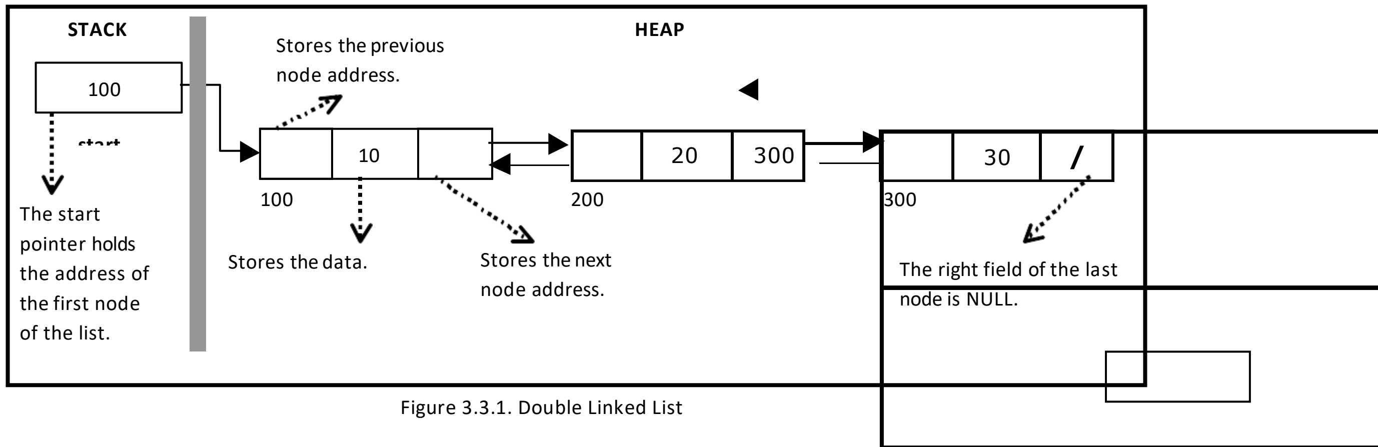


Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

```
struct dlinklist
{
struct dlinklist *left; int data;
struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

node:

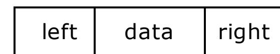


Figure 3.4.1.

Empty list: **start** NULL

Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL

```
node* getnode()

{

	node* newnode;

	newnode = (node *) malloc(sizeof(node));

	 printf("\n Enter data: ");
```

scanf("%d", &newnode -> data); newnode -> left = NULL;

newnode -> right = NULL;

return newnode;

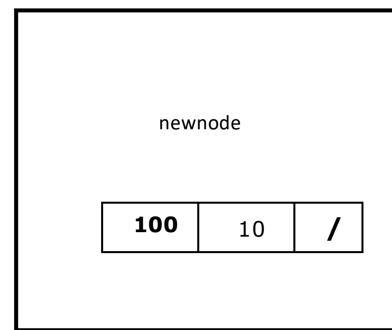


Figure 3.4.2. new node with a value of 10

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().

  newnode=getnode();

- If the list is empty then *start = newnode*.
- If the list is not empty, follow the steps given below:

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

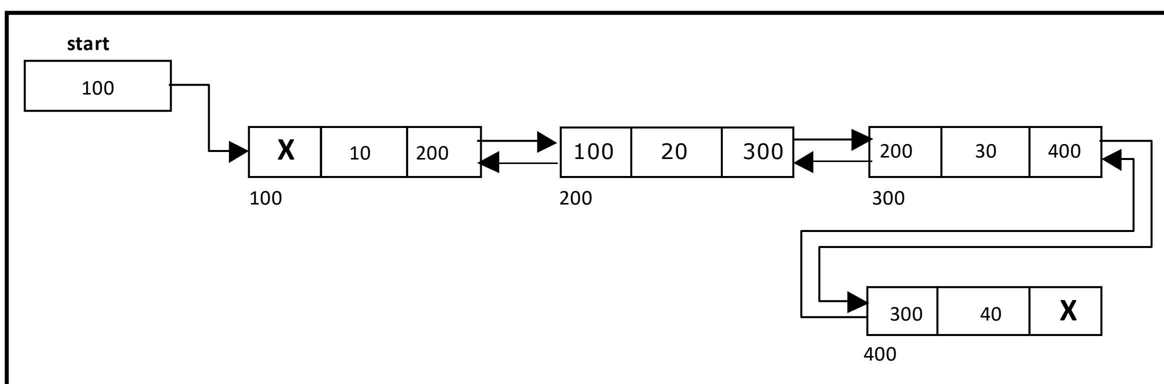


Figure 3.4.5. Inserting a node at the end

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure shows deleting a node at the beginning of a double linked list.

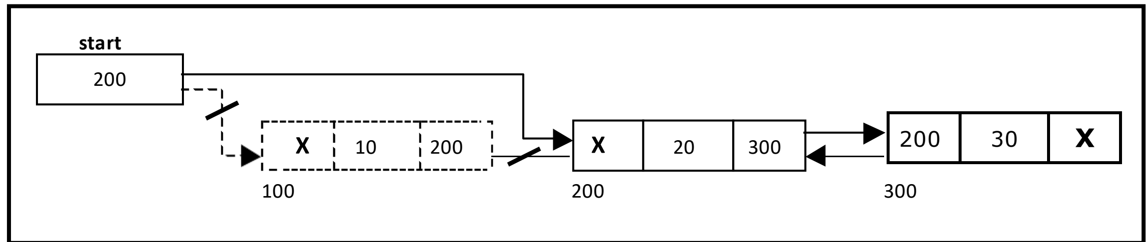
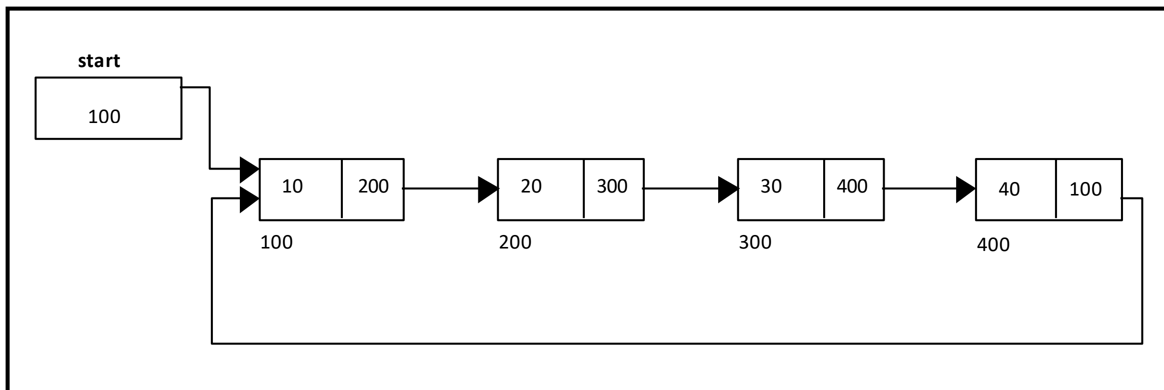


Figure 3.4.6. Deleting a node at beginning

Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.



The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
```

- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

- Repeat the above steps 'n' times.

- newnode -> next = start;

The function createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using getnode().

```
newnode = getnode();
```

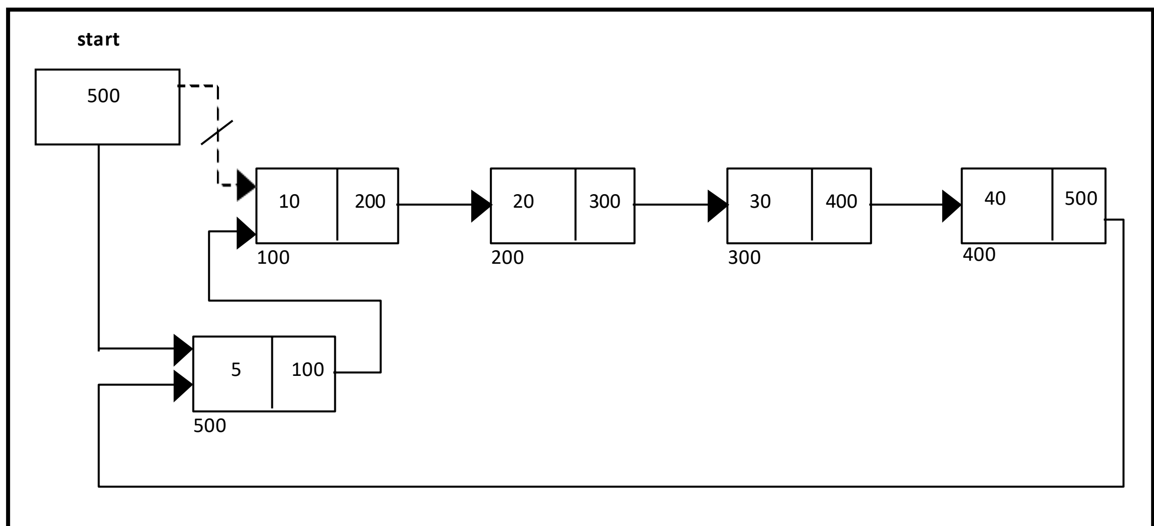
- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)
        last = last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure shows inserting a node into the circular single linked list at the beginning.



Deleting a node at the end:

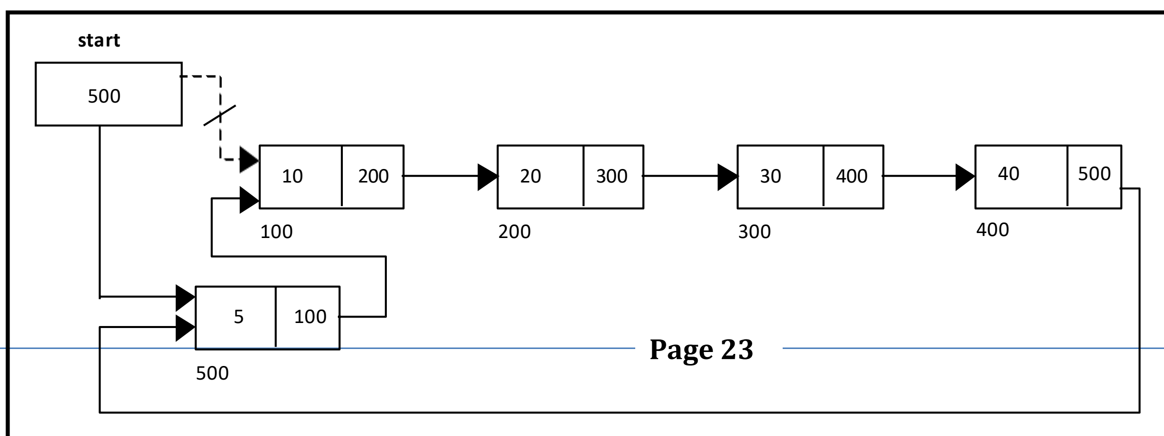
The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_last(), is used for deleting the last node in the list. shows inserting a node into the circular single linked list at the beginning.



Deleting a node at the end:

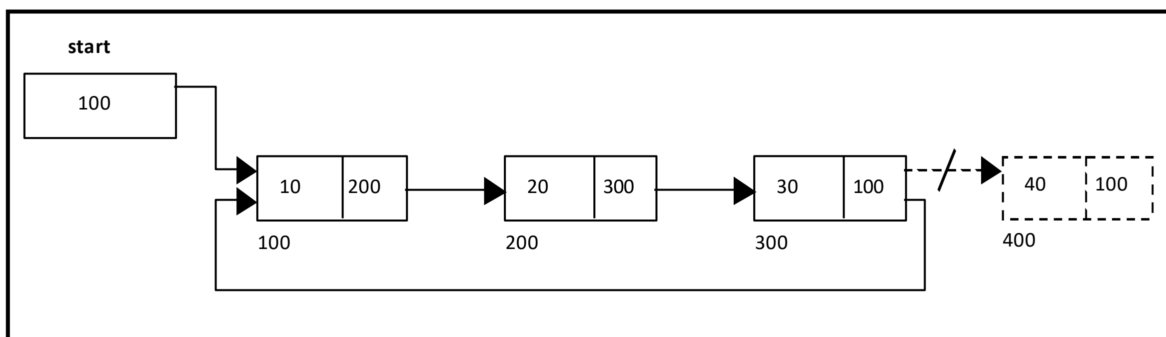
The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

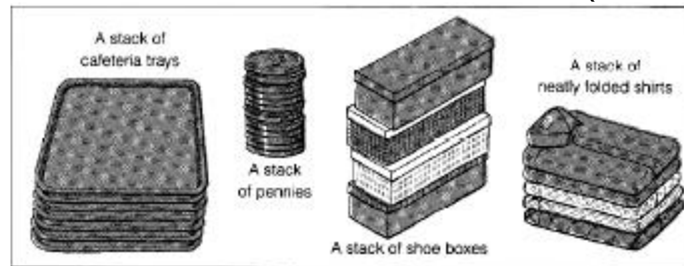
The function cll_delete_last(), is used for deleting the last node in the list. Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

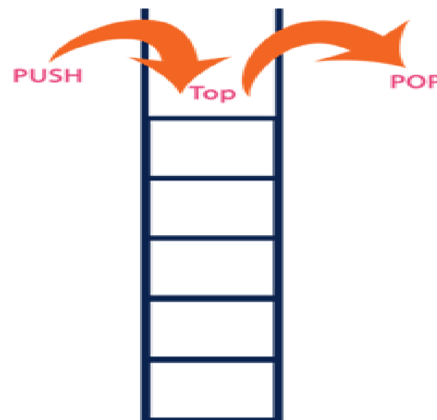


UNIT II

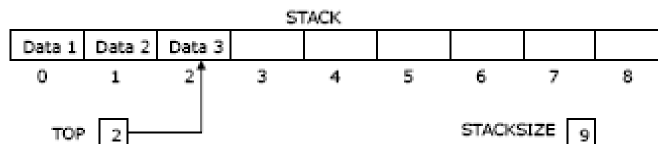
STACK

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).





one end, called ***TOP*** of the stack. The elements are removed in reverse order of that in which they were inserted into the stack.



STACK OPERATIONS

These are two basic operations associated with stack:

- ***Push()*** is the term used to insert/add an element into a stack.
- ***Pop()*** is the term used to delete/remove an element from a stack.

Other names for stacks are *piles* and *push-down lists*.

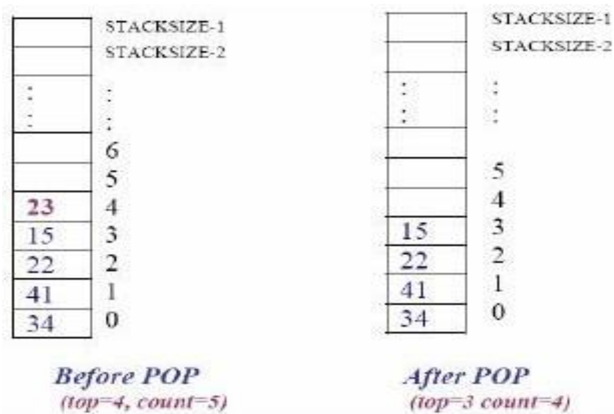
There are two ways to represent *Stack* in memory. One is using array and other is using linked list.

Array representation of stacks:

Usually the stacks are represented in the computer by a linear array. In the following algorithms/procedures of pushing and popping an item from the stacks, we have considered, a linear array STACK, a variable TOP which contain the location of the top element of the stack; and a variable STACKSIZE which gives the maximum number of elements that can be hold by the stack

Push Operation

Push an item onto the top of the stack (***insert an item***)



Algorithm for PUSH:

Algorithm: PUSH(STACK, TOP, STACKSIZE, ITEM)

1. [STACK already filled?]
   If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and Return.
2. Set TOP:=TOP+1. [Increase TOP by 1.]
3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
4. RETURN.

Algorithm for POP:

Algorithm: POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [STACK has an item to be removed? Check for empty stack]
   If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
3. Set TOP=TOP-1. [Decrease TOP by 1.]
4. Return.

Here are the minimal operations we'd need for an abstract stack (and their typical names):

- `Push`: Places an element/value on *top* of the stack.
- `Pop`: Removes value/element from *top* of the stack.
- `IsEmpty`: Reports whether the stack is Empty or not.
- `IsFull`: Reports whether the stack is Full or not.

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

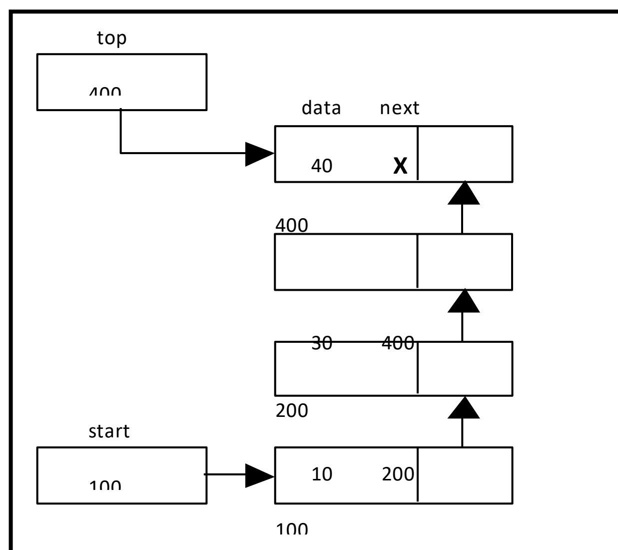


Figure 4.3. Linked stackrepresentation

Source code for stack operations, using linked list:

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


struct stack
{
int data;
struct stack *next;
};
void push(); void pop(); void display();
typedef struct stack node; node *start=NULL;
node *top = NULL;

node* getnode()
{
node *temp;
temp=(node *) malloc( sizeof(node)) ; printf("\n Enter data ");
scanf("%d", &temp -> data); temp -> next = NULL;
return temp;
}
void push(node *newnode)
{
node *temp;
if( newnode == NULL )
{
printf("\n Stack Overflow.."); return;
}
if(start == NULL)
{
                start = newnode; top = newnode;
}
else
{
                temp = start;
                while( temp -> next != NULL) temp = temp -> next;
                temp -> next = newnode; top = newnode;


}
printf("\n\n\t Data pushed into stack");
}
void pop()
{
node *temp; if(top == NULL)
{
printf("\n\n\t Stack underflow"); return;
}
temp = start;
if( start -> next == NULL)
{

}
else
{
```

```
        Printf("\n\n\t Popped element is %d ", top -> data); start = NULL;
        free(top); top = NULL;
    while(temp -> next != top)
    {

    temp = temp -> next;
    }
    temp -> next = NULL;
    printf("\n\n\t Popped element is %d ", top -> data); free(top);
    top = temp;
    }
    }
    void display()
    {
    node *temp; if(top == NULL)
    {
    }
    else
    {
    printf("\n\n\t\t Stack is empty ");

temp = start;
printf("\n\n\n\t\t Elements in the stack: \n"); printf("%5d ", temp -> data);
while(temp != top)
{
temp = temp -> next; printf("%5d ", temp -> data);
}
}
}
char menu()
{    char ch; clrscr();
printf("\n \tStack operations using pointers.. "); printf("\n -----------**********     \n");
printf("\n 1. Push ");
printf("\n 2. Pop "); printf("\n 3. Display"); printf("\n 4. Quit ");
printf("\n Enter your choice: "); ch = getche();
return ch;
}
void main()
{
char ch;
node *newnode; do
{
ch = menu(); switch(ch)
{
case '1' :
newnode = getnode(); push(newnode); break;
case '2' :
pop(); break;
case '3' :
display(); break;
case '4':
return;
}
getch();
} while( ch != '4' );
```

```
}
```

ARRAY AND LINKED IMPLEMENTATION OF STACK

A Program that exercise the operations on Stack Implementing Array

```
// i.e. (Push, Pop,
Traverse) #include
<conio.h> #include
<iostream.h>
#include
<process.h>
#define STACKSIZE 10 // int const STACKSIZE = 10;
// global variable and array
declaration int Top=-1;
int Stack[STACKSIZE];
void Push(int); // functions
prototyping int Pop(void);
bool
IsEmpty(void);
bool
IsFull(void);
void
Traverse(void);
int main( )
{ int item,
choice;
while( 1 )
{
cout<< "\n\n\n\n\n";
cout<< " ******* STACK OPERATIONS *********
\n\n"; cout<< " 1- Push item \n 2- Pop Item \n";
cout<< " 3- Traverse / Display Stack Items \n
4- Exit."; cout<< " \n\n\t Your choice ---> ";
cin>>
choice;
switch(ch
oice)
{ case 1: if(IsFull())cout<< "\n Stack
Full/Overflow\n"; else
{ cout<< "\n Enter a number: ";
cin>>item; Push(item); }
break;
case 2: if(IsEmpty())cout<< "\n Stack is
empty) \n"; else
{item=Pop();
cout<< "\n deleted from Stack =
"<<item<<endl;} break;
case 3: if(IsEmpty())cout<< "\n Stack is
empty) \n"; else
{ cout<< "\n List of Item pushed on Stack:\n";
```

```
Traverse();
}
break;

case 4:
exit(0);
default:
cout<< "\n\n\t Invalid Choice: \n";
} // end of switch block
} // end of while loop
} // end of of main()
function void Push(int item)
{
Stack[++Top] = item;
}
int Pop( )
{
return Stack[Top--];
}
bool IsEmpty( )
{ if(Top == -1 ) return true else return
false; } bool IsFull( )
{ if(Top == STACKSIZE-1 ) return true else return
false; } void Traverse( )
{ int TopTemp = Top;
do{ cout<< Stack[TopTemp--]<<endl;} while(TopTemp>= 0);
}
```

1- Run this program and examine its behavior.

// A Program that exercise the operations on Stack

// Implementing POINTER (Linked Structures) (Dynamic Binding)

// This program provides you the concepts that how STACK is

// implemented using Pointer/Linked Structures

```
#include
<iostream.h.h>
#include <process.h>
struct node {
int info;
struct node *next;
};
struct node *TOP = NULL;
void push (int x)
{ struct node *NewNode;
NewNode = new (node); // (struct node *)
malloc(sizeof(node)); if(NewNode==NULL) { cout<<"\n\n
Memeory Crash\n\n"; return; }
NewNode->info = x;
NewNode->next = NULL;
if(TOP == NULL) TOP = NewNode;
```

```
else
{ NewNode->next =
TOP; TOP=NewNode;
}
}
struct node* pop ()
{ struct node
*T; T=TOP;
TOP = TOP->next;
return T;
}
void Traverse()
{ struct node *T;
for( T=TOP ; T!=NULL ;T=T->next) cout<<T->info<<endl;
}
bool IsEmpty()
{ if(TOP == NULL) return true; else return
false; } int main ()
{ struct node
*T; int item,
ch; while(1)
{ cout<<"\n\n\n\n\n\n ***** Stack Operations
*****\n"; cout<<"\n\n 1- Push Item \n 2- Pop Item
\n";
cout<<" 3- Traverse/Print stack-values\n 4-
Exit\n\n"; cout<<"\n Your Choice --> ";
cin>>ch;
switch(ch
)
{ case 1:
cout<<"\nPut a value:
"; cin>>item;
Push(item)
; break;
case 2:
if(IsEmpty()) {cout<<"\n\n Stack is
Empty\n"; break;
}
T= Pop();
cout<< T->info <<"\n\n has been deleted
\n"; break;
case 3:
if(IsEmpty()) {cout<<"\n\n Stack is
Empty\n"; break;
}
Traverse()
; break;
case 4:
```

```
exit(0);
} // end of switch block
}

 return 0;
} // end of main function
```

Applications of stack

1. Checking for well formedness of parenthesis in an expression
2. Syntax check
3. Evaluation of postfix expression
4. Conversion of infix expression to postfix form
5. Recursive function
6. Tower of honai

Well formedness of parenthesis

An expression uses parenthesis to impose precedence of operation in ways different from the normal one

operation	precedence
parenthesis	highest
Unary operators	highest
^(exponentiation)	
*,/	
+,-	lowest

Syntax checking

A language can be expressed using regular expressions

Consider the expression wcw'{ w is in {a,b}}where w is its string in {a,b} and w' is its reverset.this denotes sentences of the form :a string followed by c and its reverse. Abcba

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *

Postfix expressions are easily evaluated

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	? × (4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

with the aid of a stack.

Postfix Evaluation Algorithm

The time complexity is O(n) because each operand is scanned once, and each operation is performed once.

A more formal algorithm:

```
create a new stack
while(input stream is not empty){
   token = getNextToken();
   if(token instanceof operand){
       push(token);
   } else if (token instance of operator) {
       op2 = pop();
       op1 = pop();
       result = calc(token, op1, op2);
       push(result);
   }
}
return pop();
```

Example 1:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Infix to Postfix

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Scan the infix input string/stream left to right
2. Initilise an empty stack.
3. If the current input token is an operand, simply append it to the output string
4. If the current input token is '(', push it onto the stack
5. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
6. If the end of the input string is found, pop all operators and append them to the output string.

. Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);

        return (result);
}


        else
        {


        }
```

Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

Factorial of a given number:

The operation of recursive factorial function is as follows:

Start out with some natural number N (in our example, 5). The recursive definition is:

n = 0, 0 ! = 1   Base Case
n > 0, n ! = n * (n - 1) !   Recursive Case

<u>Recursion Factorials:</u>

5! =5 * 4! = 5 *___= _____

4! = 4 *3! = 4 *____= _____

factr(5) = 5 * factr(4) = factr(4) = 4 * factr(3) =

3! = 3 * 2! = 3 *____= ____

2! = 2 * 1! = 2 *____= _____

factr(3) = 3 * factr(2) = factr(2) = 2 * factr(1) =

! = 1 * 0! = 1 * = factr(1) = 1 * factr(0) =

0! = 1 factr(0) =

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.

Figure 3.11.1, illustrates the initial setup of towers of Hanoi. The figure 3.11.2, illustrates the final setup of towers of Hanoi.

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

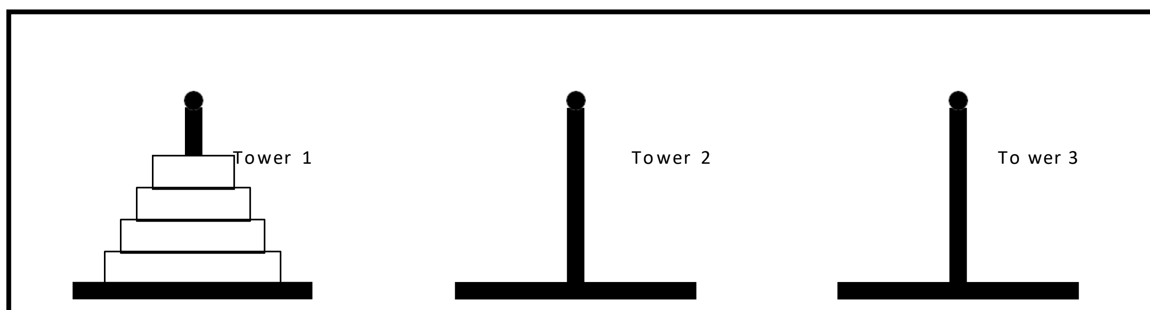


Fig. 3. 1 1. 1. In it ia l s et u p of T o w ers of Ha n o i

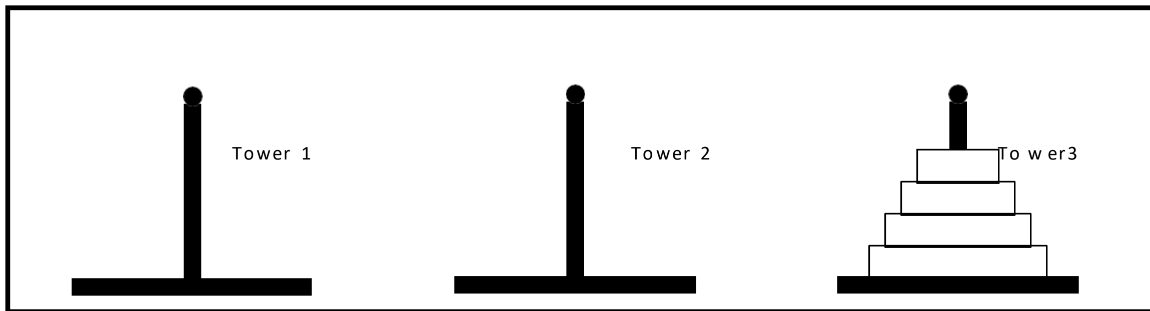


Fig 3. 1 1. 2. F in a l s et u p of T o w ers of Ha n o i

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.

FUNCTION MoveTower(*disk*, *source*, *dest*, *spare*):

IF *disk* == 0, THEN:

move *disk* from *source* to *dest*

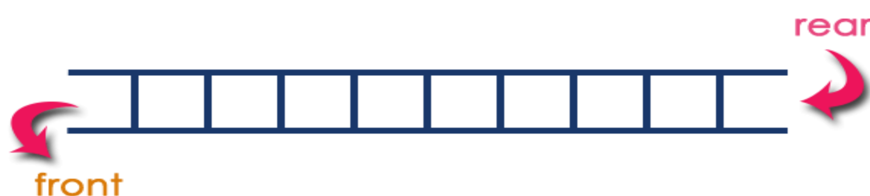
ELSE:

MoveTower(*disk* - 1, *source*, *spare*, *dest*)
// *Step 1 above* move *disk* from *source* to
dest //
Step 2 above MoveTower(*disk* - 1, *spare*,
dest, *source*) // *Step 3 above*

END IF

Queue:

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

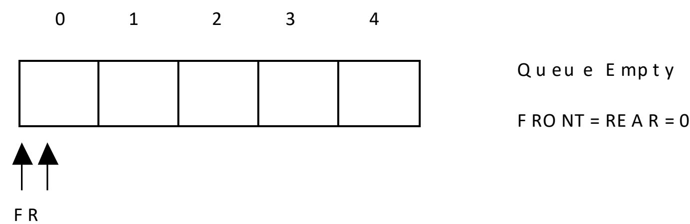


The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

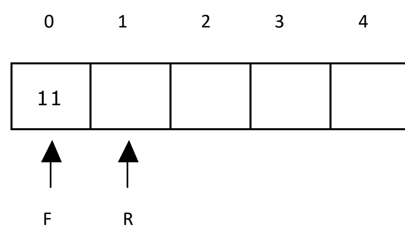
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Representation of Queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

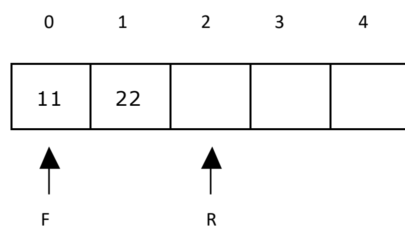


Now, insert 11 to the queue. Then queue status will be:



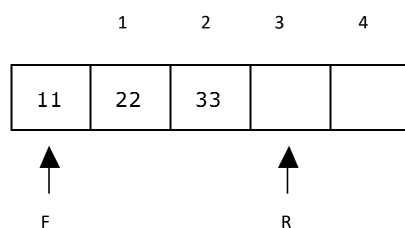
REAR = REAR + 1 = 1 FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



REAR = REAR + 1 = 2 FRONT = 0

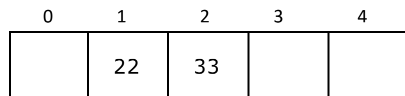
Again insert another element 33 to the queue. The status of the queue is:



REAR = REAR + 1 = 3

FRO NT = 0

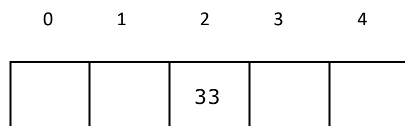
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:

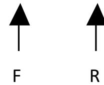


RE A R = 3

F RO NT = F R O NT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:

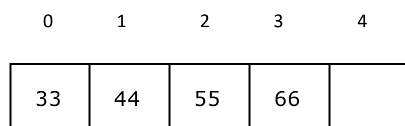




RE A R = 3

F RO NT = F R O NT + 1 = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



RE A R = 4 FRO NT = 0

Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

```
# include <conio.h>
#define MAX 6

int Q[MAX]; int
front, rear;


void insertQ()
{
int data;
if(rear == MAX)
{
            printf("\n Linear Queue is full"); return;
}
else
{
            printf("\n Enter data: "); scanf("%d", &data); Q[rear] = data;
            rear++;
            printf("\n Data Inserted in the Queue ");


}
}
void deleteQ()
{
if(rear == front)
{

}
else
{

}
}           printf("\n\n Queue is Empty.."); return;

            printf("\n Deleted element from Queue is %d", Q[front]); front++;
```

```
void displayQ()
{
int i;
if(front == rear)
{
}
else
{
printf("\n\n\t Queue is Empty"); return;


printf("\n Elements in Queue are: "); for(i = front; i < rear; i++)

{
printf("%d\t", Q[i]);
}
}
}
int menu()
{
int ch; clrscr();
printf("\n \tQueue operations using ARRAY.."); printf("\n -----------**********        \n");
printf("\n 1. Insert "); printf("\n 2. Delete "); printf("\n 3. Display"); printf("\n 4. Quit ");
printf("\n Enter your choice: "); scanf("%d", &ch);
return ch;
}
void main()
{
int ch; do
{
        ch = menu();
        switch(ch)
        {
        case 1:    insertQ(); break;

        case 2:    deleteQ(); break;

        case 3:    displayQ(); break;

        case 4:    return;

        }

        getch();
        } while(1);
        }
```

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

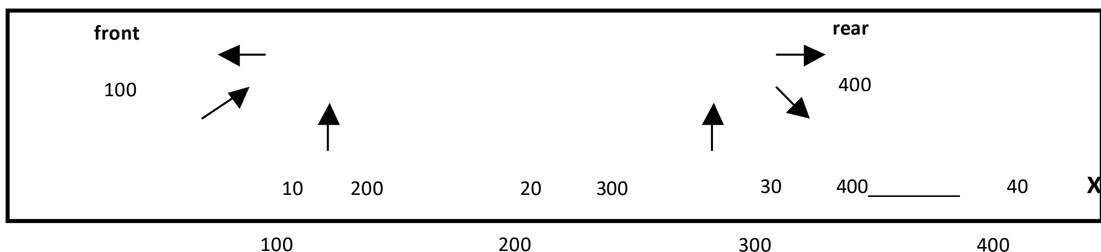


Figure 4.4. Linked Queue representation

Dequeue:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

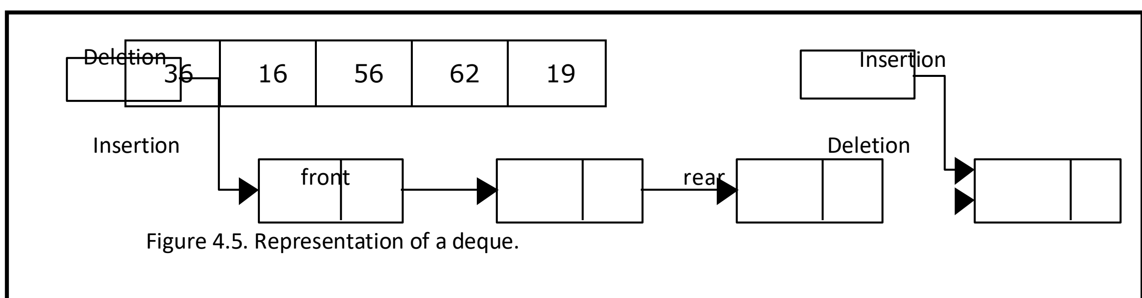


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

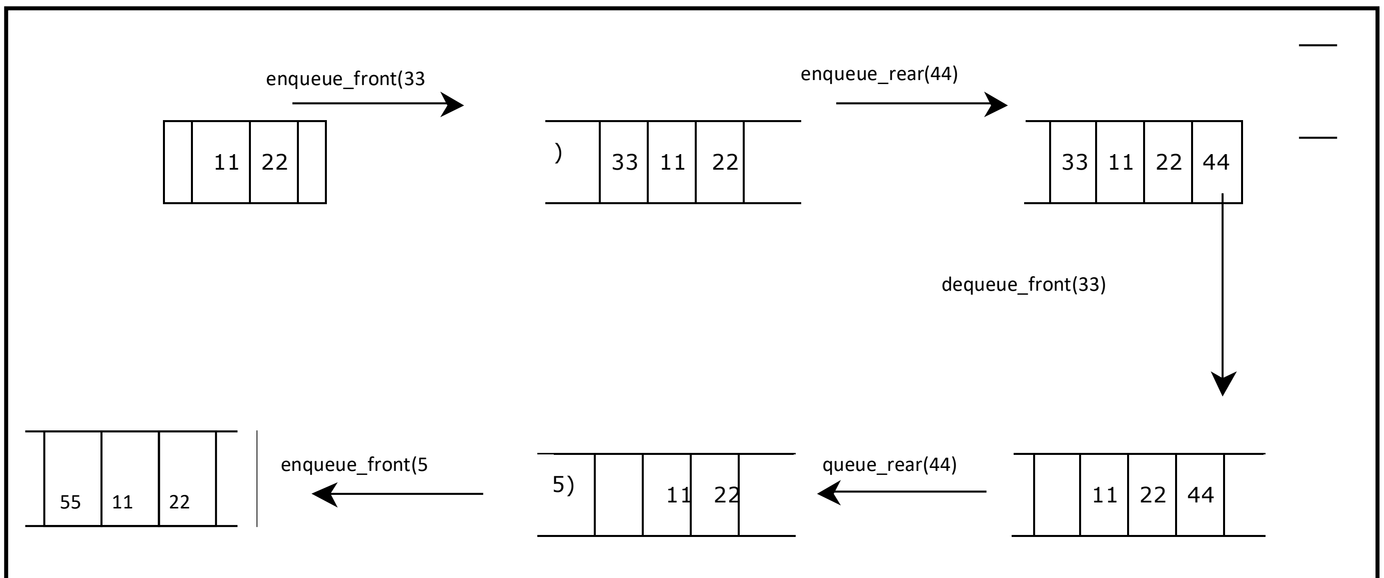


Figure 4.6. Basic operations on deque

Algorithm to add an element into DeQueue :

Assumptions: pointer f,r and initial values are -

1,-1 Q[] is an array

max represent the size of a queue

enq_front

step1. Start
step2. Check the queue is full or not as if
(f <> step3. If false update the pointer f
as f= f-1
step4. Insert the element at pointer f as Q[f] =
element step5. Stop

enq_back

step1. Start
step2. Check the queue is full or not as if (r == max-1) if yes
queue is full step3. If false update the pointer r as r= r+1
step4. Insert the element at pointer r as Q[r] = element

step5. Stop

Algorithm to delete an element from the DeQueue

deq_front

step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false update pointer f as f = f+1 and delete element at position f as element = Q[f]
step4. If (f== r) reset pointer f and r as f=r=-1
step5. Stop

deq_back

step1.start

step2. Check the queue is empty or not as if (f == r) if yes queue
is empty
step3. If false delete element at position r as element = Q[r]

step4. Update pointer r as r = r-1
step5. If (f == r) reset pointer f and r as f = r= -1
step6. Stop

Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.

2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

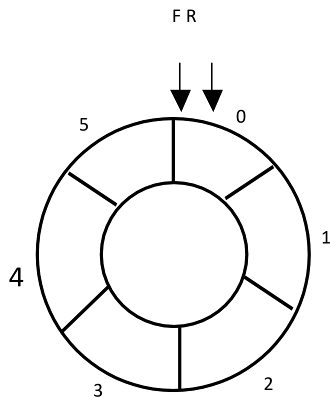
3. Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue:

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

Representation of Circular Queue:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



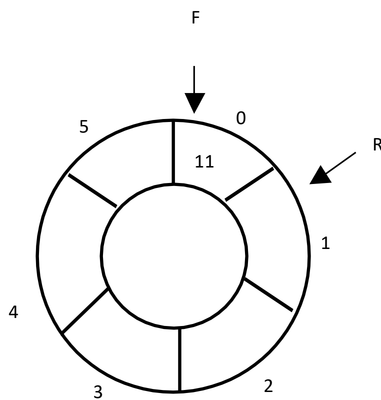
Queue Empty

MAX = 6

FRONT = REAR = 0
COUNT = 0

Circular Queue

Now, insert 11 to the circular queue. Then circular queue status will be:



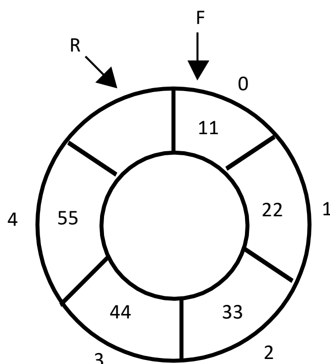
FRONT = 0

REAR = (REAR + 1) % 6 = 1

COUNT = 1

Circular Queue

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0

REAR = (REAR + 1) % 6 = 5

COUNT = 5

Circular Queue

Algorithm for Insert in a circular queue

```
        Void cqinsert (int queue[],int front,int rear,int item)
{
    If(front==0)
   front=front+1;

  If(((rear==SIZE) &&(front==1)) ||((rear!=0) && (front==*rear+1)))
{
Printf(Queue Overflow);
return;
}
If(rear==SIZE)
rear=1;
else
   rear=  rear+1;
  queue[rear]=item;
```

Algorithm for Deletion in a circular queue

```
Delete CircularQueue ( )
```

If (FRONT == 0) Then [Check for Underflow]
 Print: Underflow

Else

ITEM = QUEUE[FRONT]

If (FRONT == REAR) Then

Set FRONT = 0

Set REAR = 0

Else If (FRONT == N) Then
Set FRONT = 1

Else

Set FRONT = FRONT + 1 [Increment FRONT by 1]

. Print: ITEM deleted

. Exit

Unit III

TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The

links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

The figure 5.1.1 shows a tree and a non-tree.

Figure 5.1.1 A Tree and a not a tree

BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

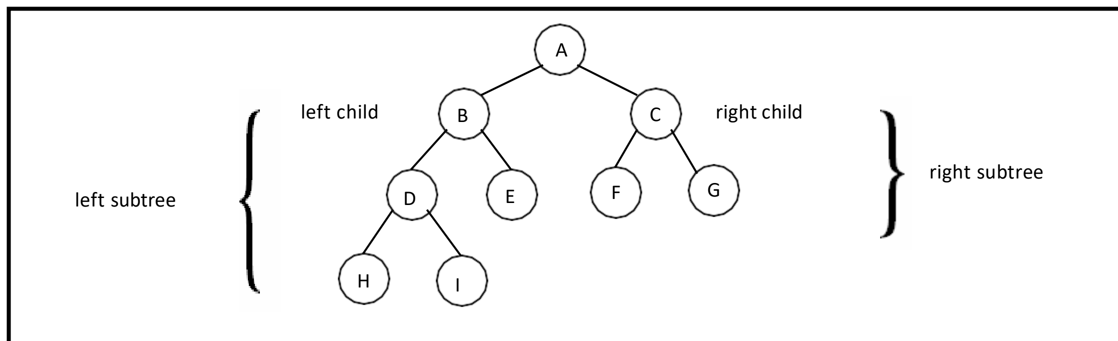


Figure 5.2.1. Binary Tree

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes n_1, n_2, . . ., n_k, such that n_i is the parent of n_{i+1} for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C

and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A*.*

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is* 2^n.

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

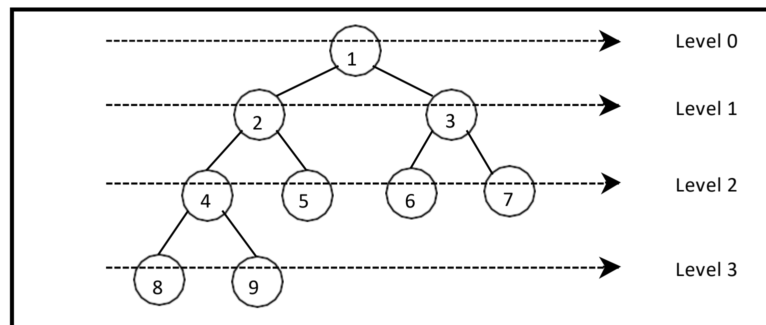


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

    a. Maximum number of leaves = 2^h

    b. Maximum number of nodes = $2^{h+1} - 1$

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.

4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.

A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes $(tn) = \sum_{l=0}^{h} 2^l = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

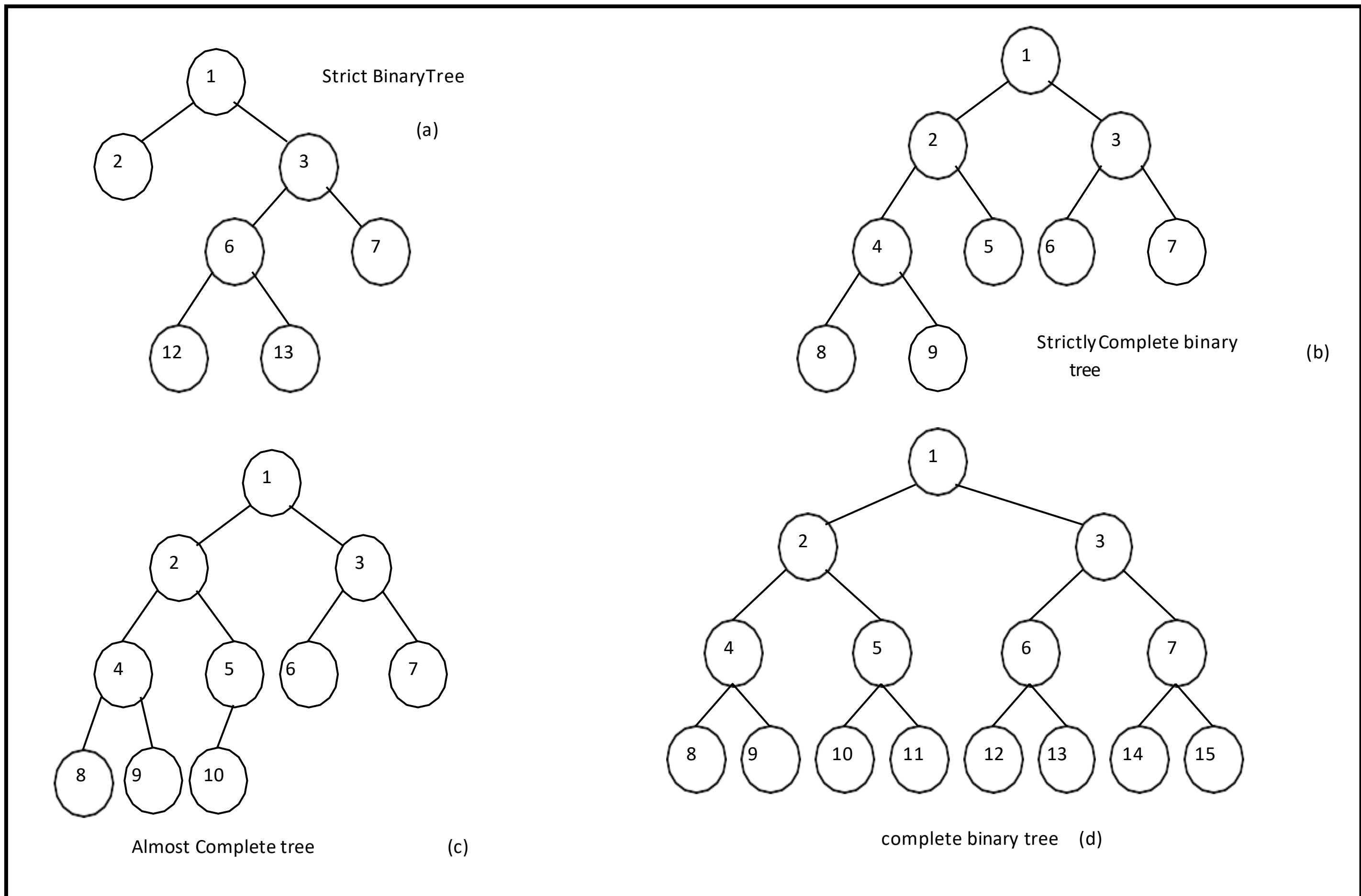


Figure 5.2.3. Examples of binary trees

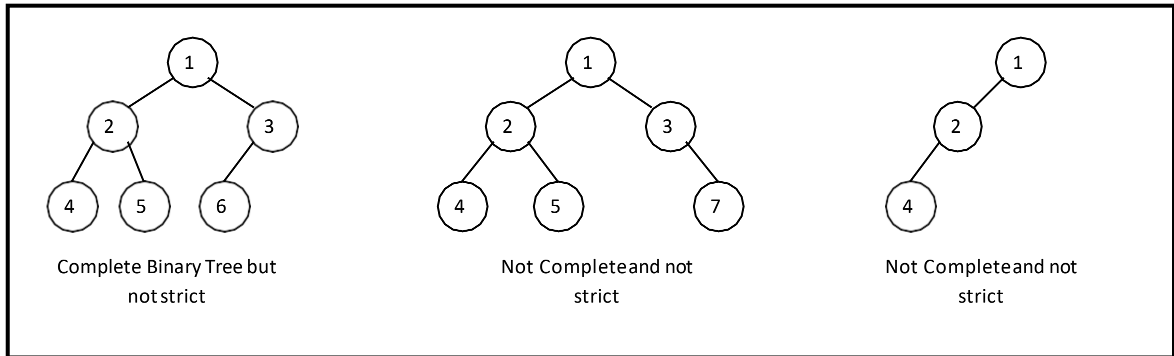
Complete Binary tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

Figure 5.2.4. Examples of complete and incomplete binary trees



Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is some times convenient to have another types of nodes, called an external node,. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

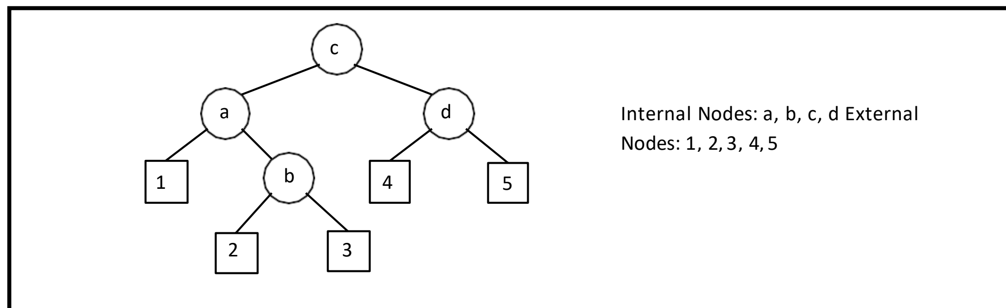


Figure 5.2.6. Internal and external nodes

Almost complete binary tree

The total number of nodes in a complete binary tree of depth d is given expression $2^{d+1}-1$

Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

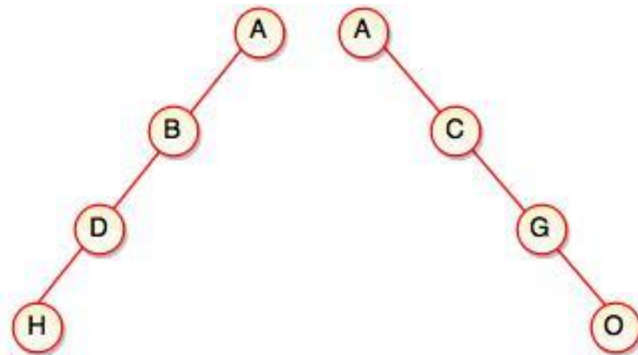


Fig. Left Skewed Binary Tree

Fig. Right Skewed Binary Tree

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order or breath first traversal*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
```

```
        print root -> data;
        inorder(root->rchild);
    }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}
```

Level order Traversal: or breath first traversal

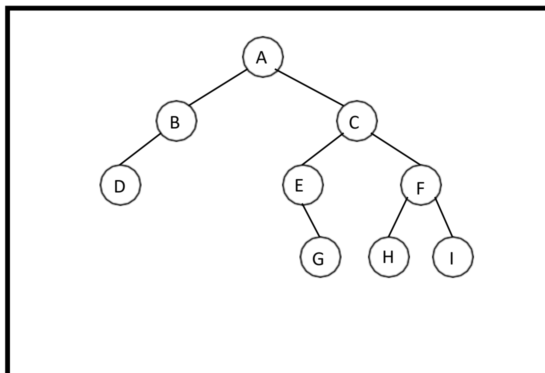
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

Algorithm for level order traversal is as follows:

```
void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        print tree[j] -> data;
        }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



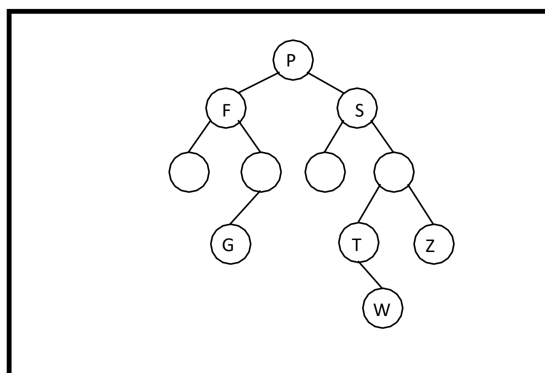
Binary Tree

- Preorder traversal yields: A, B, D, C, E, G, F, H, I
- Postorder traversal yields: D, B, G, E, H, I, F, C, A
- Inorder traversal yields: D, B, A, E, G, C, H, F, I
- Level order traversal yields: A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



Bin ary Tree

- Preorder traversal yields: P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields: B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields: B, F, G, H, P, R, S, T, W, Y, Z

Pre, Post, Inorder and level order Traversing

Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

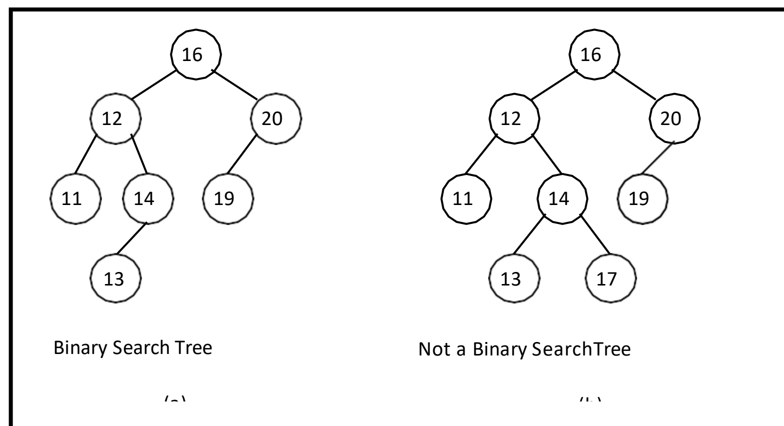


Figure 5.2.5. Examples of binary search trees

```
struct TreeNode {

    int key;

    int value;

    struct TreeNode *left;

    struct TreeNode *right;

};


bool isBST(struct TreeNode *node, int minKey, int maxKey) {

    if (node == NULL) return true;

    if (node->key < minKey || node->key > maxKey) return false;


    return isBST(node->left, minKey, node->key-1) && isBST(node->right, node->key+1,
maxKey);

}
```

General Trees (m-ary tree):

If in a tree, the outdegree of every node is less than or equal to ***m***, the tree is called general tree. The general tree is also called as an *m-ary* tree. If the outdegree of every node is exactly equal to m or zero then the tree is called *a full or complete m-ary* tree. For m = 2, the trees are called *binary* and *full binary trees.*

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

Converting a *m-ary* tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

Conversion from general tree to binary can be done in two stages.

Stage 1:

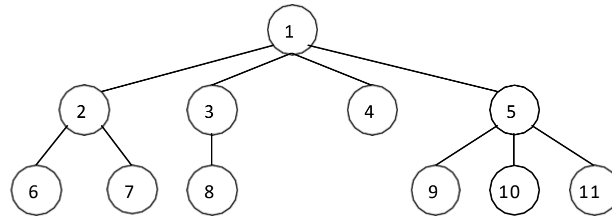
- As a first step, we delete all the branches originating in every node except the left most branch.
- We draw edges from a node to the node on the right, if any, which is situated at the same level.

Stage 2:

- Once this is done then for any particular node, we choose its left and right sons in the following manner:
  - The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

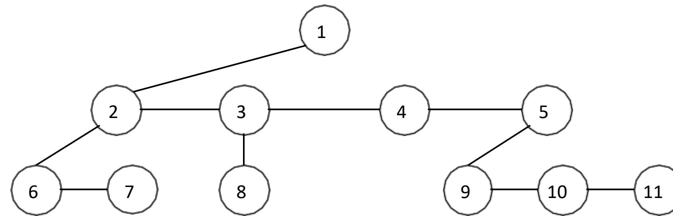
Example 1:

Convert the following ordered tree into a binary tree:

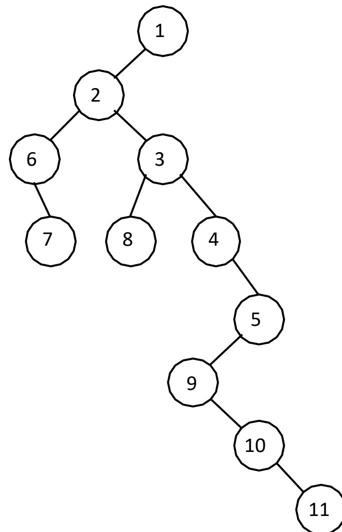


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



Stage 2 tree by using the above mentioned procedure is as follows:



Huffman coding

Every information in computer science is **encoded** as strings of **1s and 0s**. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

Fixed-Length encoding - Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively. Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’. Suppose we randomly assign binary codes to each character as follows-

a 0
b 011
c 111
d 11

Thus, the string “aabacdad” gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string 00011011111011, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11)
aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11)
aabbcb (0 | 0 | 011 | 011 | 111 | 011)

… and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the **“prefix rule”** which states that no binary code should be a prefix of another code. This will produce uniquely **decodable codes**. The above codes for ‘a’, ‘b’, ‘c’ and ‘d’ do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous **decodable codes.**

Lets reconsider assigning the binary codes to characters ‘a’, ‘b’, ‘c’ and ‘d’.

a 0
b 11
c 101
d 100

Using the above codes, string **“aabacdad”** gets encoded to 001101011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100). Now, we can decode it back to string **“aabacdad”**.

Problem Statement-

Input: Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output: Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-

Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by David Huffman in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child. Also `frequency(z)= frequency(x)+frequency(y)`

Step 5- Add z to min heap

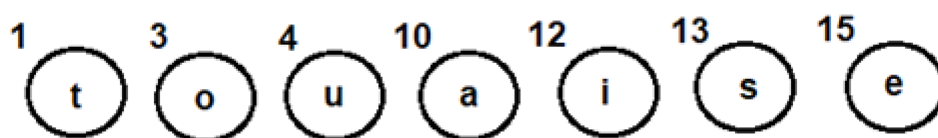
Step 6- Last node in the heap is the root of Huffman tree

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

- **Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)



Leaf nodes for each character

Step B- Repeat the following steps till heap has more than one nodes

- **Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap
- **Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)
- **Step 5**- Add z to min heap

i. Extract and Combinenode u with an internal node having 4 as the frequency
ii. Add the new internal node to priority queue-

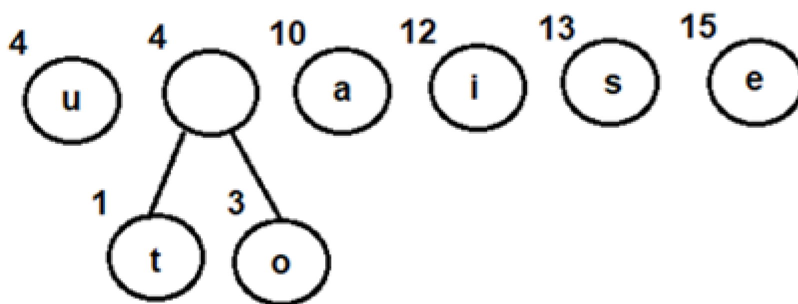


Fig 2: Combining nodes o and t

i. Extract and Combine node awith an internal node having 8 as the frequency
ii. Add the new internal node to priority queue-

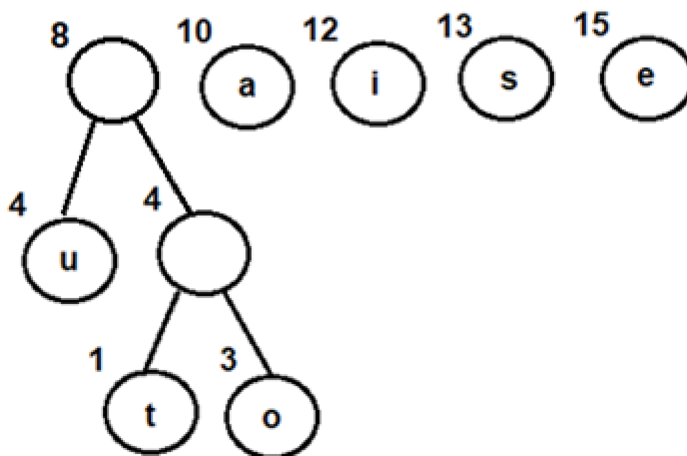


Fig 3: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s
ii. Add the new internal node to priority queue-

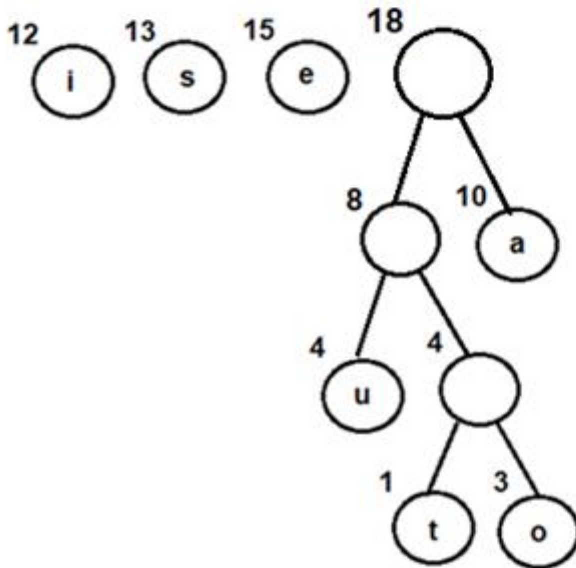


Fig 4: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s
ii. Add the new internal node to priority queue-

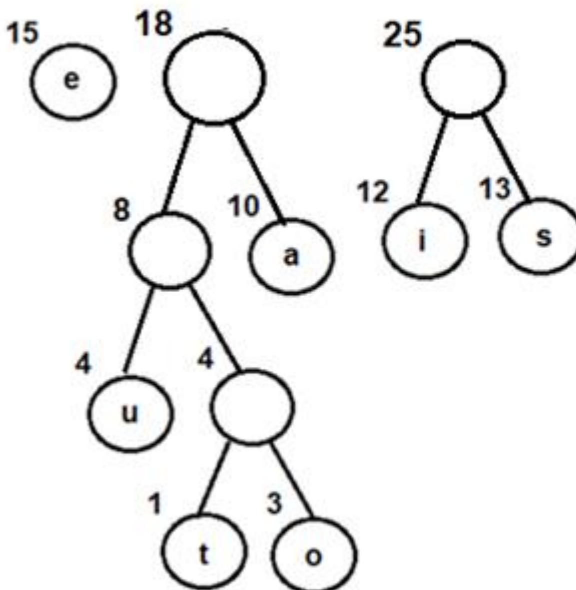


Fig 5: Combining nodes i and s

i. Extract and Combine node e with an internal node having 18 as the frequency
ii. Add the new internal node to priority queue-

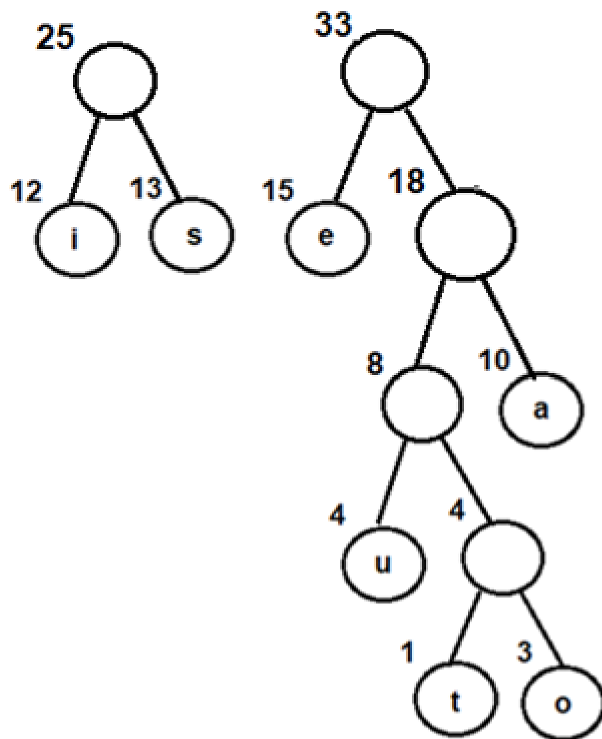


Fig 6: Combining node e with an internal node having 18 as frequency

i. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency
ii. Add the new internal node to priority queue-

Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to array while traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

u

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

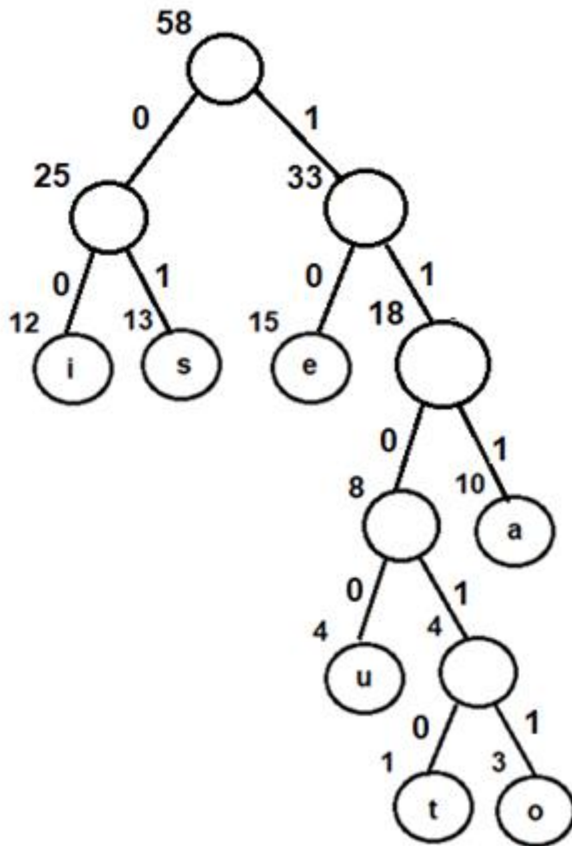


Fig 8: Assigning binary codes to Huffman tree

Characters	Binary Codes
i	00
s	01
e	10
u	1100

t	11010
o	11011
a	111

Using the above binary codes-

Suppose the string "staeiout" needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to **"0111010111100011011110011010" (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010)** at the sender side.

Once received at the receiver's side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A '1' or '0' in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

On similar lines-

- 111 gets decoded to 'a'
- 10 gets decoded to 'e'
- 00 gets decoded to 'i'
- 11011 gets decoded to 'o'
- 1100 gets decoded to 'u'
- And finally, 11010 gets decoded to 't', thus returning the string "stakeout" back

.Decoding

The **decoding** procedure is deceptively simple. Starting with the first bit in the stream, one then uses successive bits from the stream to determine whether to go left or right in the **decoding** tree. ... The next bit in the input stream is the first bit of the next character.

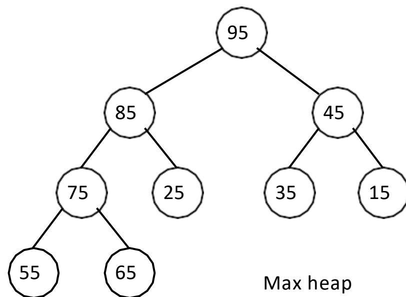
HEAP

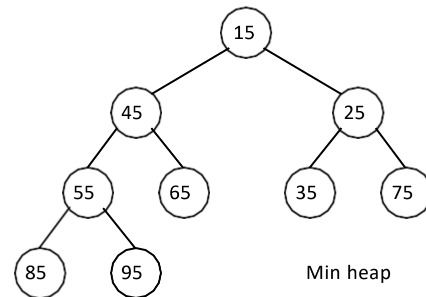
Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.





A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

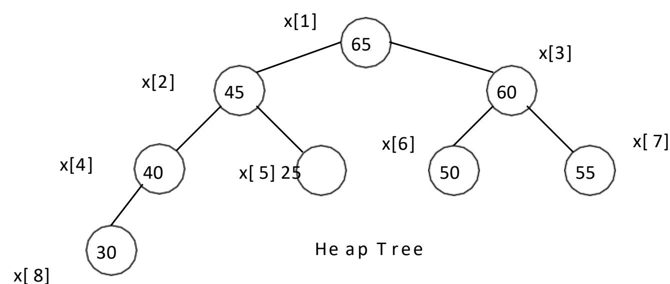
Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2**i*+1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

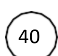
Max_heap_insert (a, n)

```
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        int i, n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[ ≤ i/2 ƒ ] < item ) do
        {
                a[i] = a[ ≤ i/2 ƒ ] ;                          // move the parent down
                i = ≤ i/2 ƒ  ;
        }
        a[i] = item ;
        return true ;
}
```

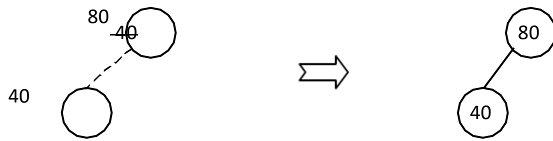
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

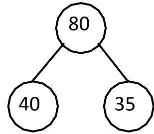
1. Insert 40:



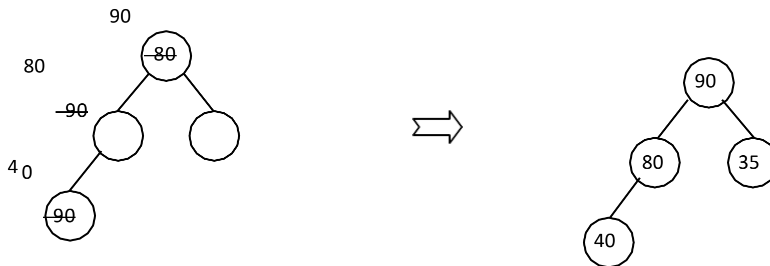
2. Insert 80:



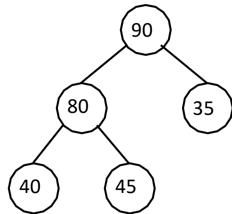
3. Insert 35:



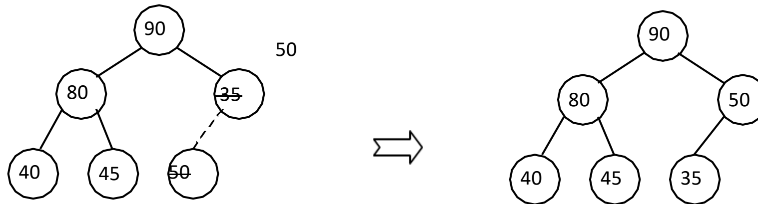
4. Insert 90:



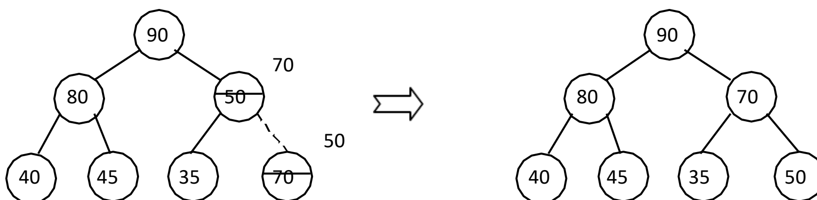
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

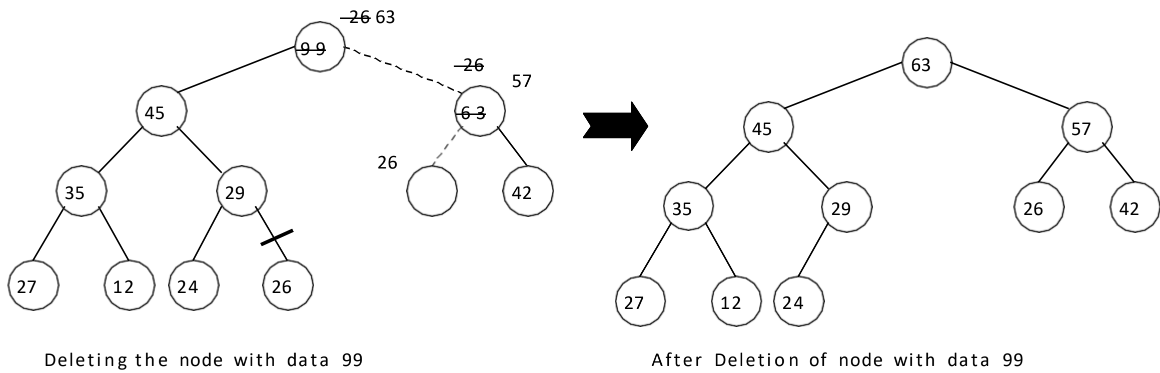
```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j © j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ≤ j / 2 ƒ ] = a[j]      // move the larger child up a level
                j = 2 * j;
        }
        a [ ≤ j / 2 ƒ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

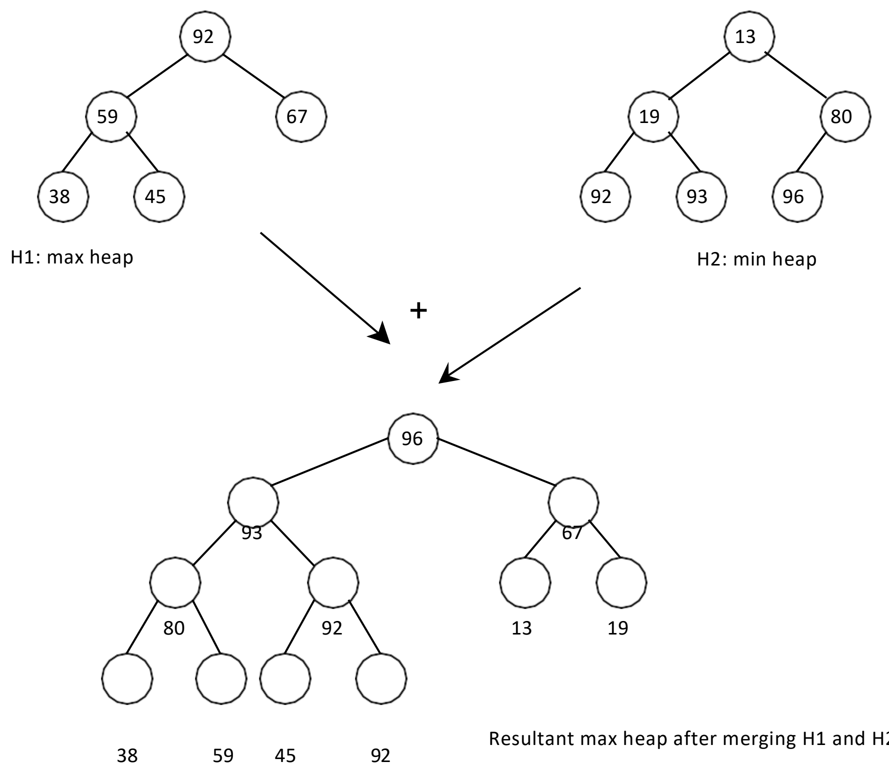
26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.



Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp  =  a[i];
                a[i]   =   a[1];
                a[1] = temp;
                adjust (a, 1, i – 1);
        }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
        for i © ≤ n/2 ƒ to 1 by – 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j © j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ≤ j / 2 ƒ ] = a[j]      // move the larger child up a level
                j = 2 * j;
        }
        a [ ≤ j / 2 ƒ ] = item;
}
```

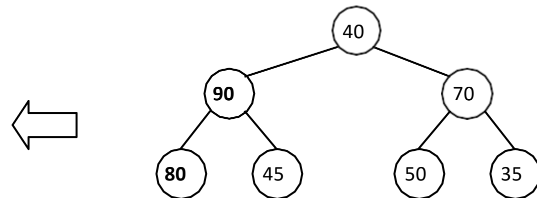
TimeComplexity:

Each 'n' insertion operations takes O(log k), where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time O(log k), where 'k' is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

Thus, for 'n' elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

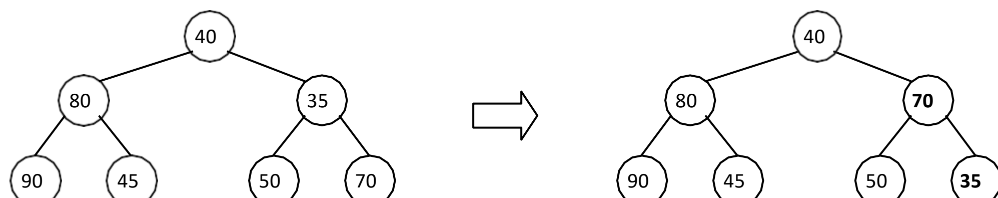
Example 1:

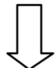


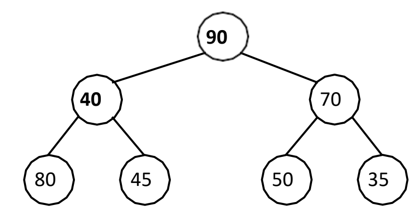
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

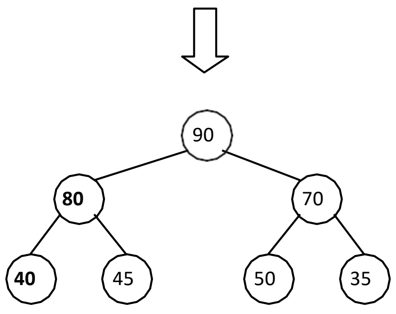
Solution:

First form a heap tree from the given set of data and then sort by repeated deletion operation:

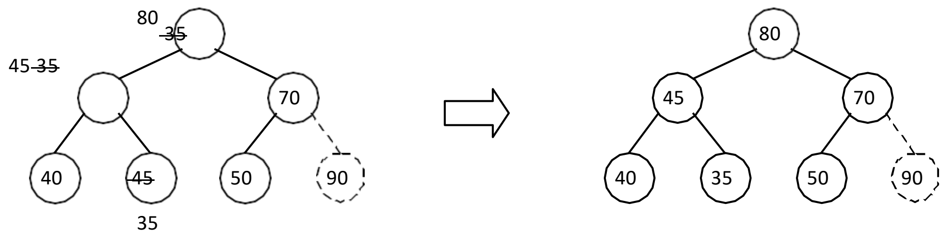




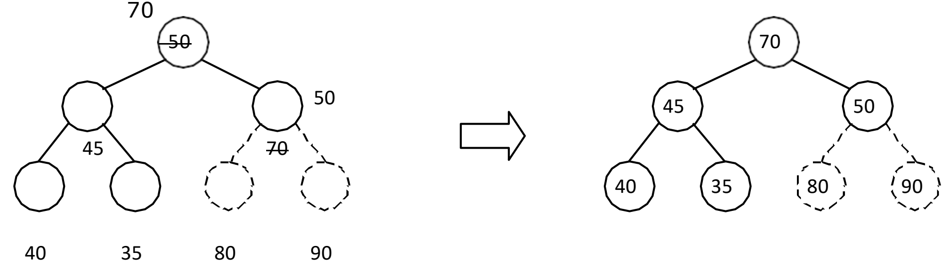




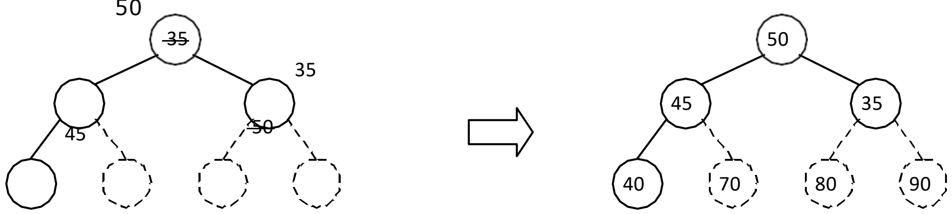
1. Exchange root 90 with the last element 35 of the array and re-heapify



2. Exchange root 80 with the last element 50 of the array and re-heapify

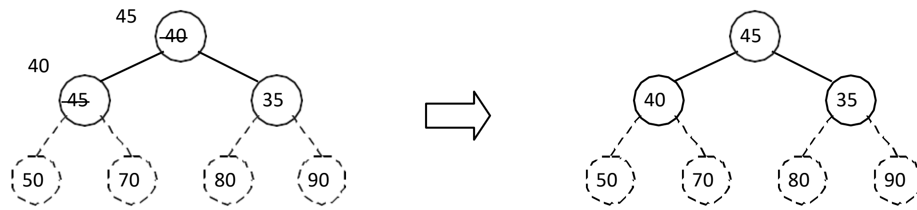


3. Exchange root 70 with the last element 35 of the array and re-heapify

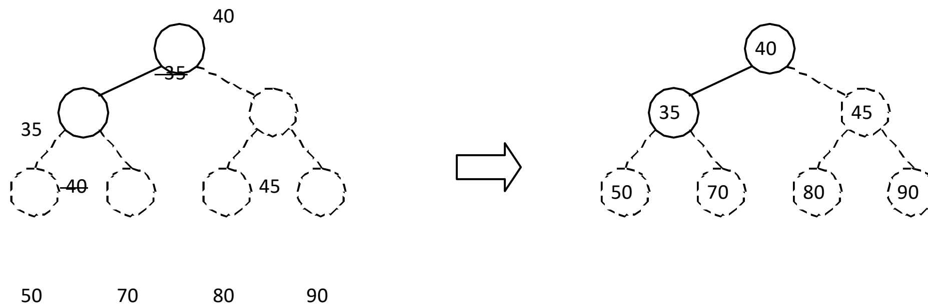


40 70 80 90

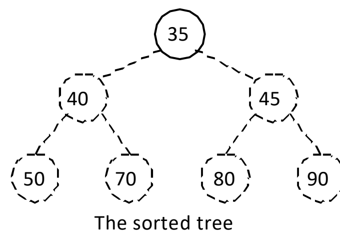
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
        int j, item;
        j = 2 * i;
        item = a[i];
        while(j <= n)
        {
                if((j < n) && (a[j] < a[j+1]))
                        j++;
                if(item >= a[j])
                        break;
                else
                {
```

```
                }
        }
                        a[j/2] = a[j];
                        j = 2*j;
        a[j/2] = item;
}

void heapify(int n, int a[])
{
        int i;
        for(i = n/2; i > 0; i--)
                adjust(i, n, a);
}

void heapsort(int n,int a[])
{
        int temp, i;
        heapify(n, a);
        for(i = n; i > 0; i--)
        {
                temp = a[i];
                a[i] = a[1];
                a[1] = temp;
                adjust(1, i - 1, a);
        }
}

void main()
{
        int i, n, a[20];
        clrscr();
        printf("\n How many element you want: ");
        scanf("%d", &n);
        printf("Enter %d elements: ", n);
        for (i=1; i<=n; i++)
                scanf("%d", &a[i]);
        heapsort(n, a);
        printf("\n The sorted elements are: \n");
        for (i=1; i<=n; i++)
                printf("%5d", a[i]);
        getch();
}
```

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Threaded Binary Tree

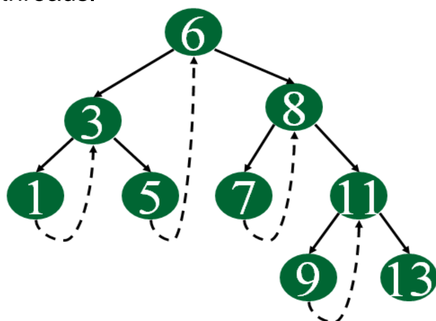
Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node

There are two types of threaded binary trees.

***Single Threaded**:* Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal. The threads are also useful for fast accessing ancestors of a node.

6 Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.

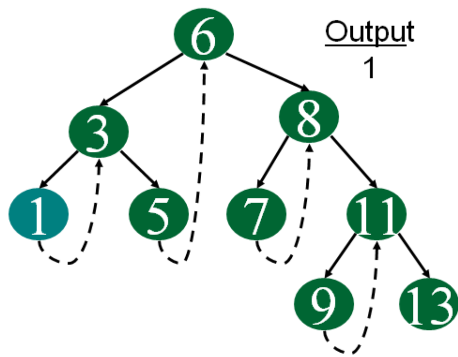



```
struct Node
```

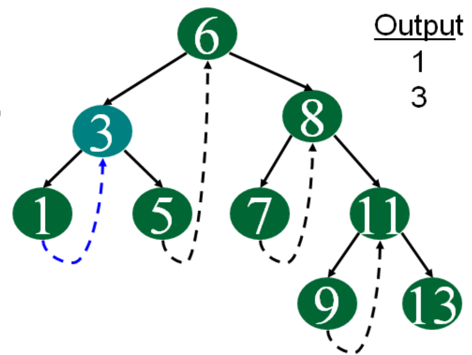
```
{
    int data;
    Node *left, *right;
    bool rightThread;
}

struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
       return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}
// C code to do inorder traversal in a threaded binary tree
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);
          // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
```

```
}
```

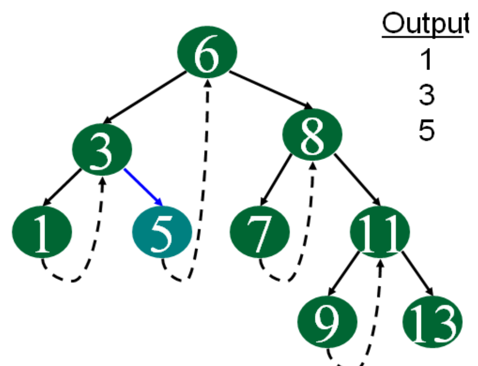
7 Following diagram demonstrates inorder order traversal using threads.



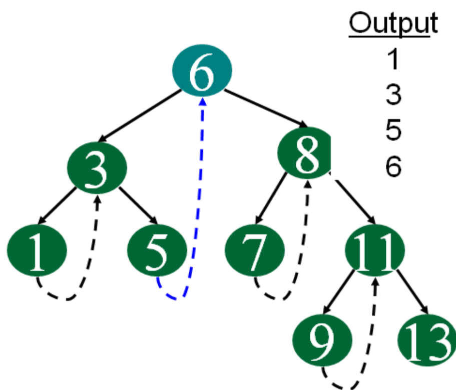
Start at leftmost node, print it



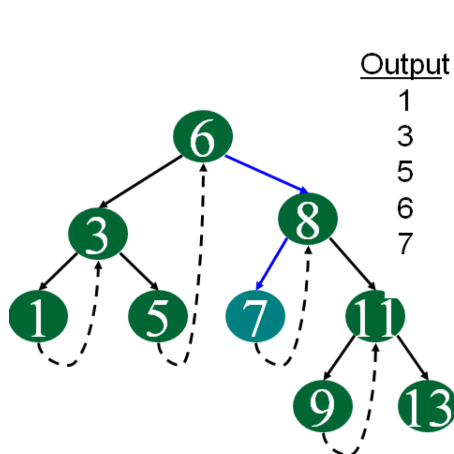
Follow thread to right, print node



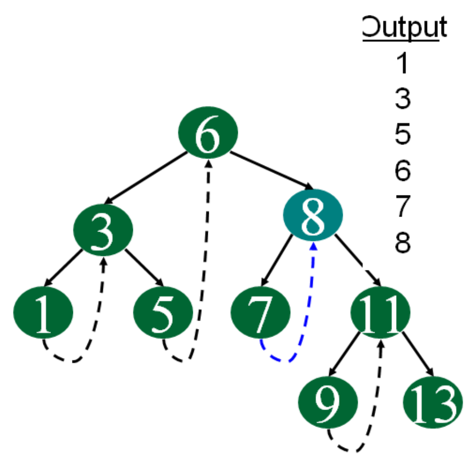
Follow link to right, go to leftmost node and print



Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow thread to right, print node

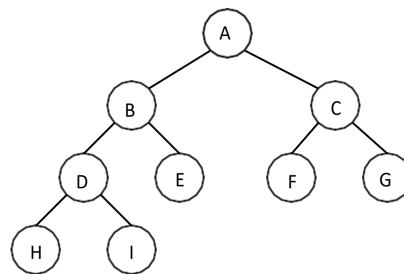
continue same way for remaining node.....

The linked representation of any binary tree has more null links than actual pointers. If there are 2n total links, there are n+1 null links.

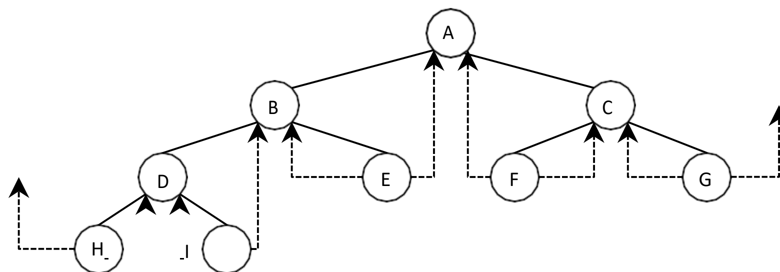
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the RCHILD(p) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:



The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

LBIT(P) = 1 if LCHILD(P) is a normal pointer
LBIT(P) = 0 if LCHILD(P) is a Thread

RBIT(P) = 1 if RCHILD(P) is a normal pointer
RBIT(P) = 0 if RCHILD(P) is a Thread

Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 5.4.1 shows some more expression trees that represent arithmetic expressions given in infix form.

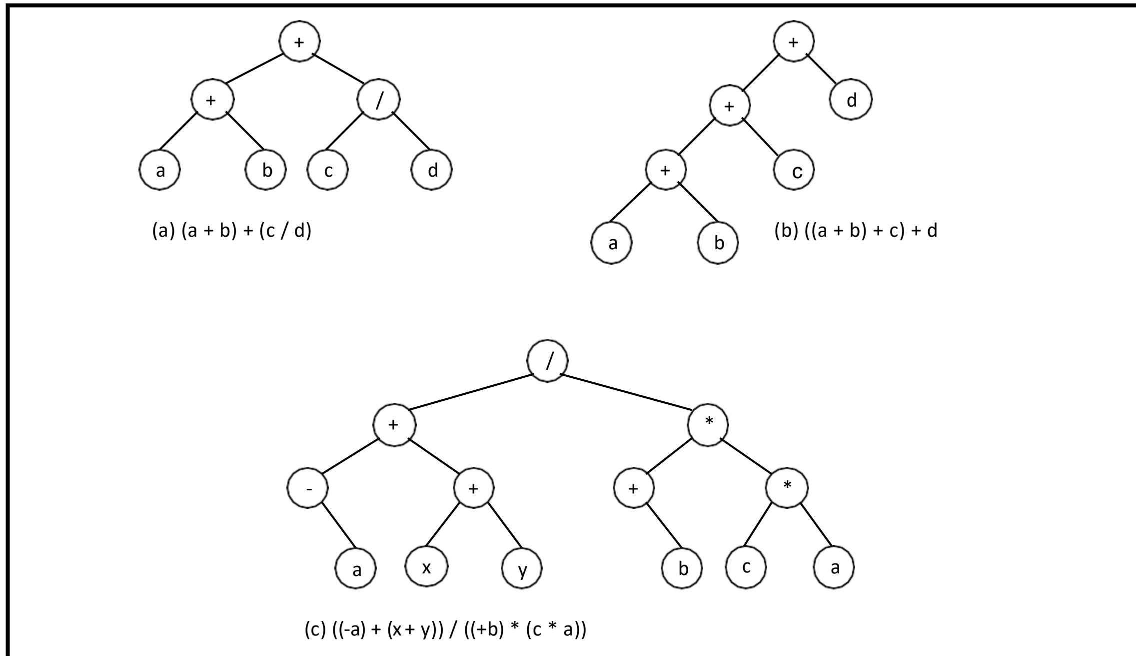


Figure 5.4.1 Expression Trees

An expression tree can be generated for the infix and postfix expressions.

An algorithm to convert a postfix expression into an expression tree is as follows:

1. Read the expression one symbol at a time.

2. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.

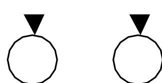
3. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

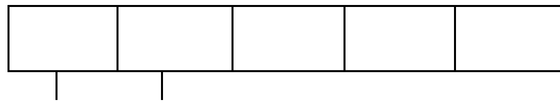
Example 1:

Construct an expression tree for the postfix expression: a b + c d e + * *

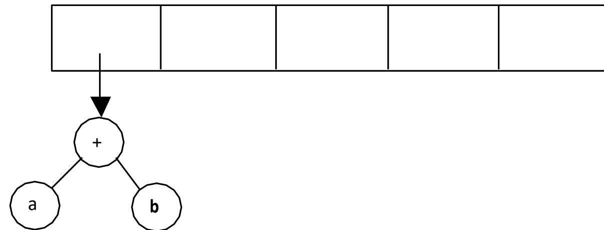
Solution:

The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.

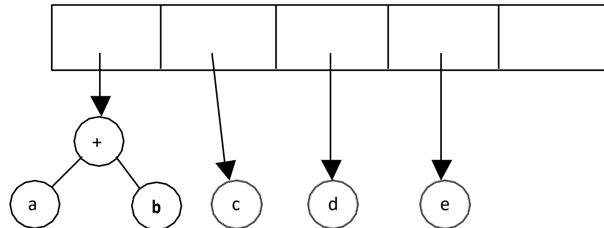




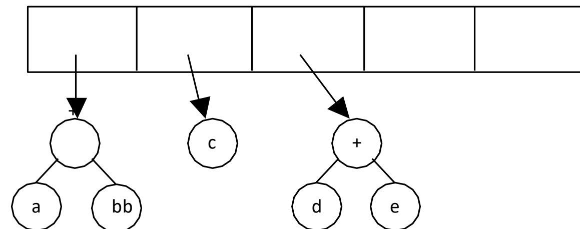
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



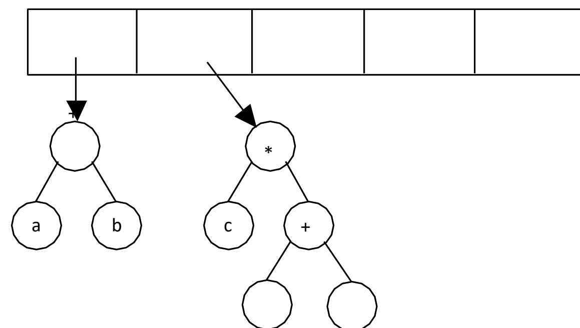
Next, c, d, and e are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



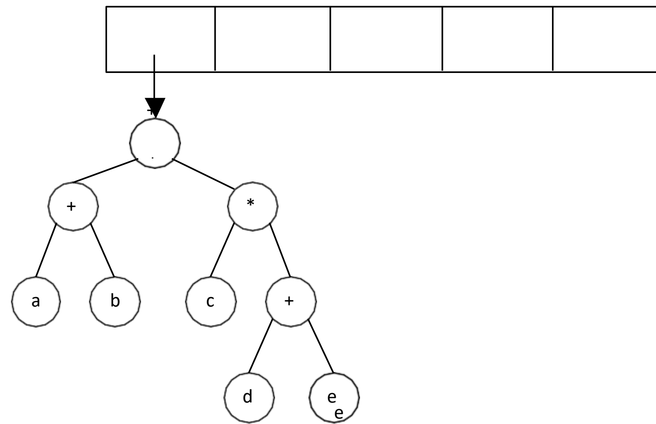
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: a + b * c * d + e

Preorder form of the expression: * + a b * c + d e

Postorder form of the expression: a b + c d e + * *

UNIT IV

Algorithm

An **a**lgorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness*:* each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).
3. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the ***TIME COMPLEXITY*** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Classification of Algorithms

If ‘n’ is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple– nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems. Whenever n doubles, the running time squares.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size ‘n’. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size ‘n’ of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as **analysis of algorithms.**

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O(n)
- Binary search is O(log n)
- Bubble sort is $O(n^2)$
- Quick sort is O(n log n)

For example, if the first program takes $100n^2$ milliseconds. While the second taken $5n^3$ milliseconds. Then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5\,n^3/100\,n^2 = n/20$$

for inputs n < 20, the program with running time $5n^3$ will be faster those the one with running time $100\,n^2$.

Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Divide and conquer

General Method

In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Finding the Maximum and Minimum

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Analysis

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y-x+1$, where **y** is greater than or equal to **x**.

$Max-Min(x,y)$ will return the maximum and minimum values of an array $numbers[x...y]$.

```
Algorithm: Max - Min(x, y)
if y – x ≤ 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x],
numbers[y]))
else
   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
   (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Analysis

Let ***T(n)*** be the number of comparisons made by $Max-Min(x,y)$, where the number of elements $n=y-x+1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n)=\begin{cases}T(\lfloor \frac{n}{2} \rfloor)+T(\lceil \frac{n}{2} \rceil)+2 & for\ n>2\\ 1 & for\ n=2\\ 0 & for\ n=1\end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, $n = 2^k$ where **k** is height of the recursion tree.

So,

$$T(n)=2.T(\frac{n}{2})+2=2.(2.T(\frac{n}{4})+2)+2.....=\frac{3n}{2}-2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Search Techniques

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For

example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)

{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
```

```
        index ++;
    }
    if(flag == 1)
        printf("Data found at %d position", index);

else

 }

 printf("data not found");
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:

45, we'll look at 1 element before success
39, we'll look at 2 elements before success
8, we'll look at 3 elements before success
54, we'll look at 4 elements before success
77, we'll look at 5 elements before success
38 we'll look at 6 elements before success
24, we'll look at 7 elements before success
16, we'll look at 8 elements before success
4, we'll look at 9 elements before success
7, we'll look at 10 elements before success
9, we'll look at 11 elements before success
20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

```
if(number[i] == data)
{
flag = 1; break;
}

if(flag == 1)

{
printf("\n Data found at location: %d", i+1);


else

printf("\n Data not found ");
}
```

BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)

{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ≤(low + high)/2 ƒ
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

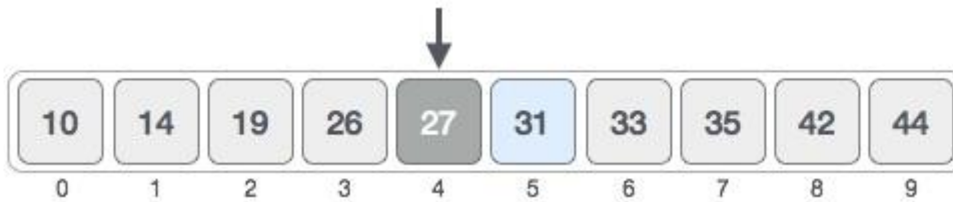
Example 1:

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

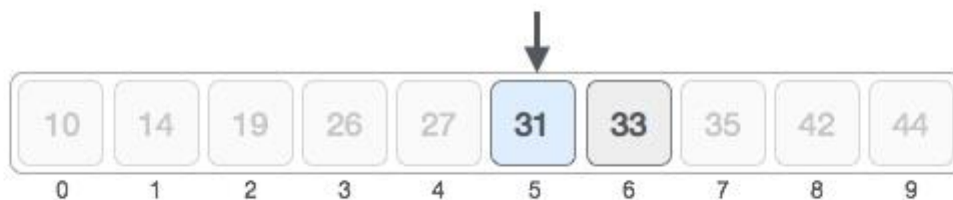
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1

mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

Merge sort

Merge Sort is a Divide and Conquer algorithm. It is simple sort that performs the following steps.

1. Split the input into two halves
2. Sort both halves using merge sort
3. Merge both sorted lists

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

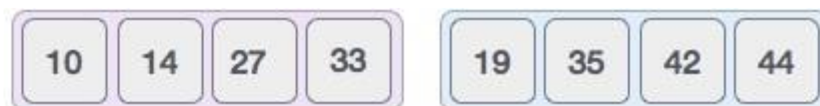
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

1. Algorithm Mergesort (data,n)

If n=1

Then return

Else

Split data into two halves data1 and data2

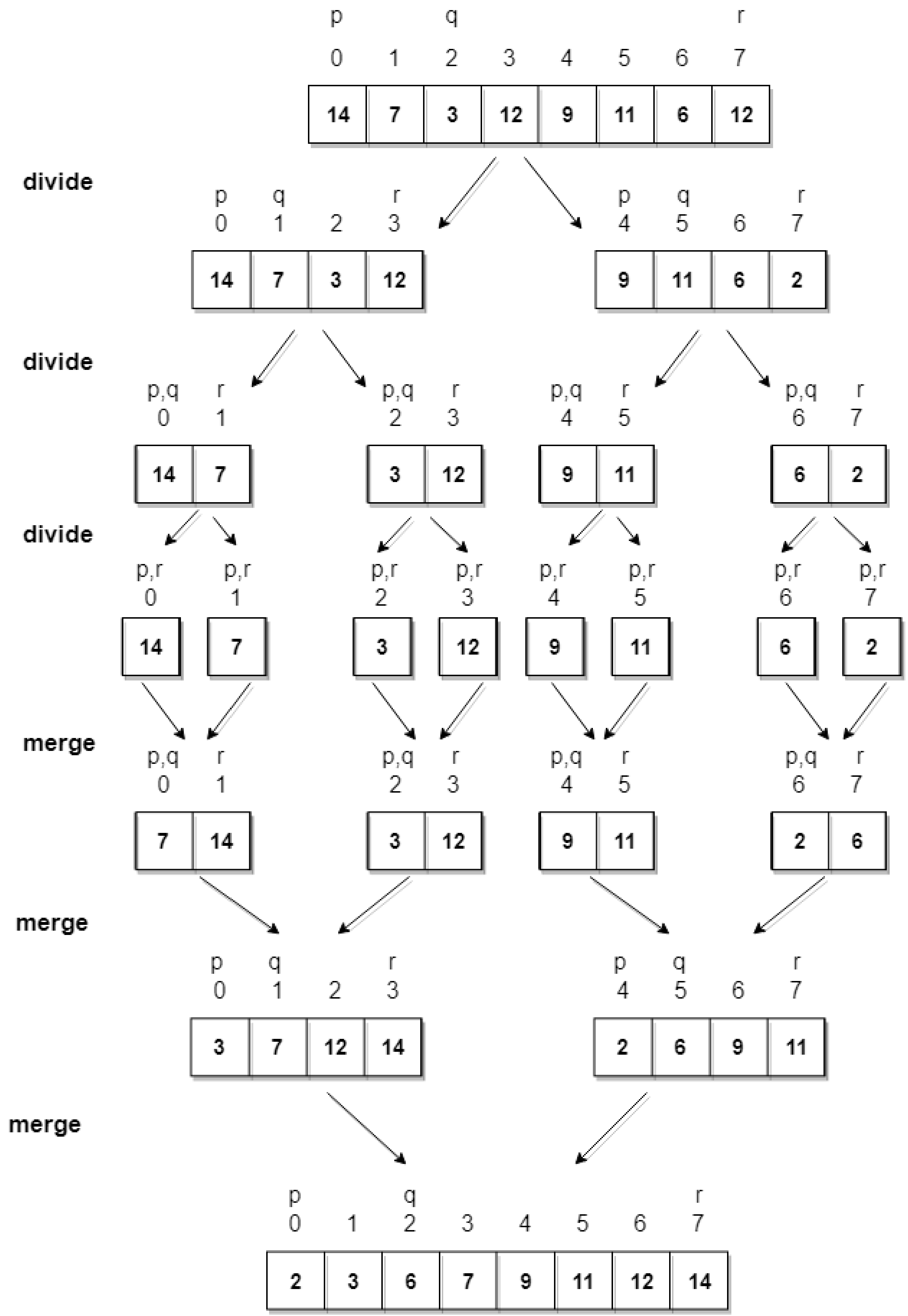
Merge(MergeSort(data1,n/2), MergeSort(data2,n/2))

End

```
2. MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
```

```
            Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
```

Example 2.



p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r
0
14
p,r
1
7
p,r
2
3
p,r
3
12
p,r
4
9
p,r
5
11
p,r
6
6
p,r
7
2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

Merge Sort is quite fast, and has a time complexity of `O(n*log n)`

Selection Sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

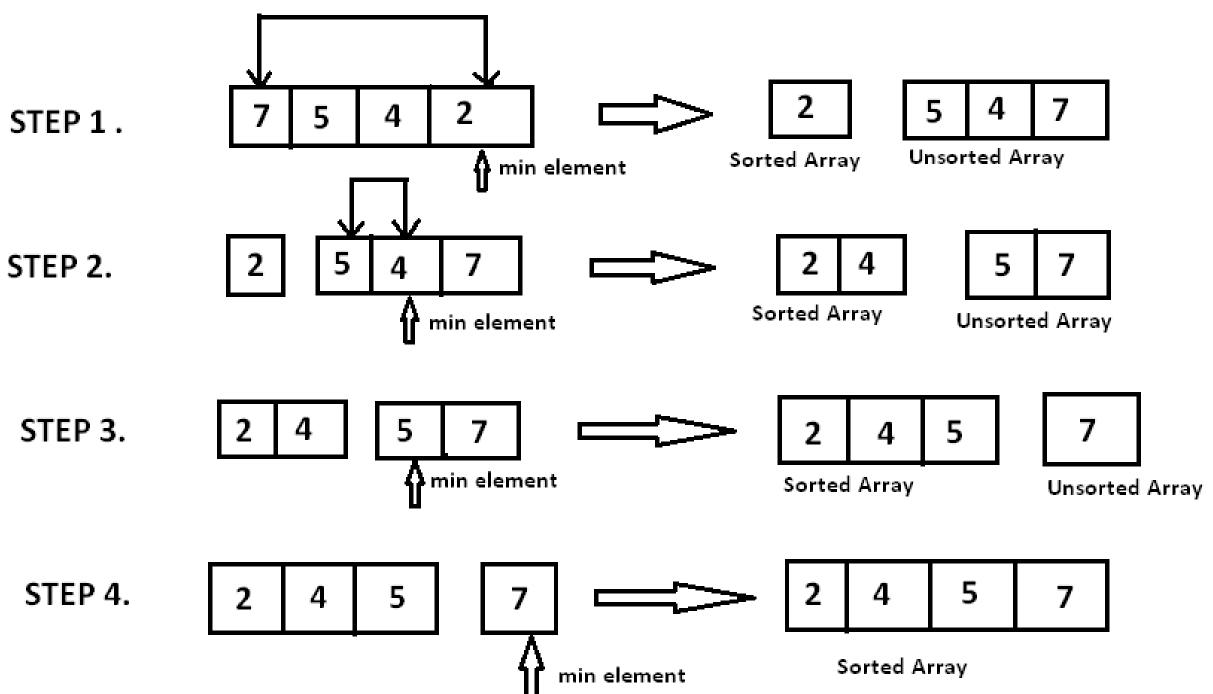
Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

Let's take a look at the implementation.

```
void selection_sort (int A[ ], int n) {
        int minimum;
                for(int i = 0; i < n-1 ; i++)   {

array .
             minimum = i ;


            for(int j = i+1; j < n ; j++ ) {
                if(A[ j ] < A[ minimum ])   {
minimum = j ;
                }
             }
          swap ( A[ minimum ], A[ i ]) ;
        }
   }
```

At ith iteration, elements from position 0 to i−1 will be sorted.



Time Complexity:

To find the minimum element from the array of N elements, N−1 comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to N−1 and then N−2 comparisons are required to find the minimum in the unsorted array.

Therefore (N−1) + (N−2) + ... + 1 = (N·(N−1))/2 comparisons and N swaps result in the overall complexity of O(N2).

Strassen's Matrix Multipilication

Introduction

Strassen's in 1969 which gives an overview that how we can find the multiplication of two **2*2 dimension matrix by the brute-force algorithm**. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

For multiplying the two 2*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition, subtraction, and in brute force algorithm, there is eight multiplication and four addition. The utility of Strassen's formula is shown by its asymptotic superiority when order **n** of matrix reaches infinity. Let us consider two matrices **A** and **B**, **n*n** dimension, where **n** is a power of two. It can be observed that we can contain four **n/2*n/2** submatrices from **A**, **B** and their product **C**. **C** is the resultant matrix of **A** and **B**.

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Procedure of Stassen's matrix multiplication

There are some procedures:

1. Divide a matrix of order of 2*2 recursively till we get the matrix of 2*2.
2. Use the previous set of formulas to carry out 2*2 matrix multiplication.
3. In this eight multiplication and four additions, subtraction are performed.
4. Combine the result of two matrixes to find the final product or final matrix.

Formulas for Stassen's matrix multiplication

In **Strassen's matrix multiplication** there are seven multiplication and four addition, subtraction in total.

```
    1. D1 =  (a11 + a22) (b11 + b22)
    2. D2 =  (a21 + a22).b11
    3. D3 =  (b12 – b22).a11
    4. D4 =  (b21 – b11).a22
    5. D5 =  (a11 + a12).b22
    6. D6 =  (a21 – a11) . (b11 + b12)
    7. D7 =  (a12 – a22) . (b21 + b22)

    C11 = d1 + d4 – d5 + d7
    C12 = d3 + d5
    C21 = d2 + d4
    C22 = d1 + d3 – d2 – d6
```

Algorithm for Strassen's matrix multiplication

Algorithm Strassen(n, a, b, d)

```
begin
       If n = threshold then compute
              C = a * b is a conventional matrix.
       Else
              Partition a into four sub matrices  a11, a12, a21, a22.
              Partition b into four sub matrices b11, b12, b21, b22.
              Strassen ( n/2, a11 + a22, b11 + b22, d1)
              Strassen ( n/2, a21 + a22, b11, d2)
              Strassen ( n/2, a11, b12 – b22, d3)
              Strassen ( n/2, a22, b21 – b11, d4)
              Strassen ( n/2, a11 + a12, b22, d5)
              Strassen (n/2, a21 – a11, b11 + b22, d6)
              Strassen (n/2, a12 – a22, b21 + b22, d7)

              C = d1+d4-d5+d7     d3+d5
              d2+d4           d1+d3-d2-d6

       end if

       return (C)
end.
```

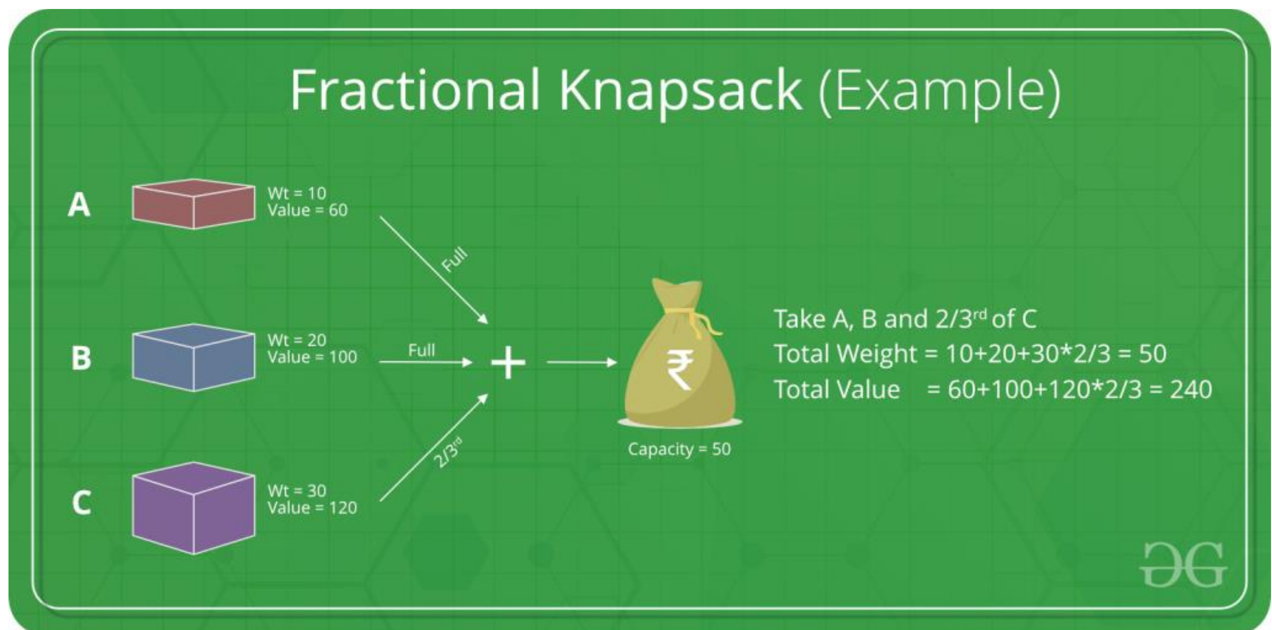
UNIT-V

Greedy Method

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

.

For example consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to global optimal solution because we allowed to take fractions of an item.



Most networking algorithms use the greedy approach. Here is a list of few of them −

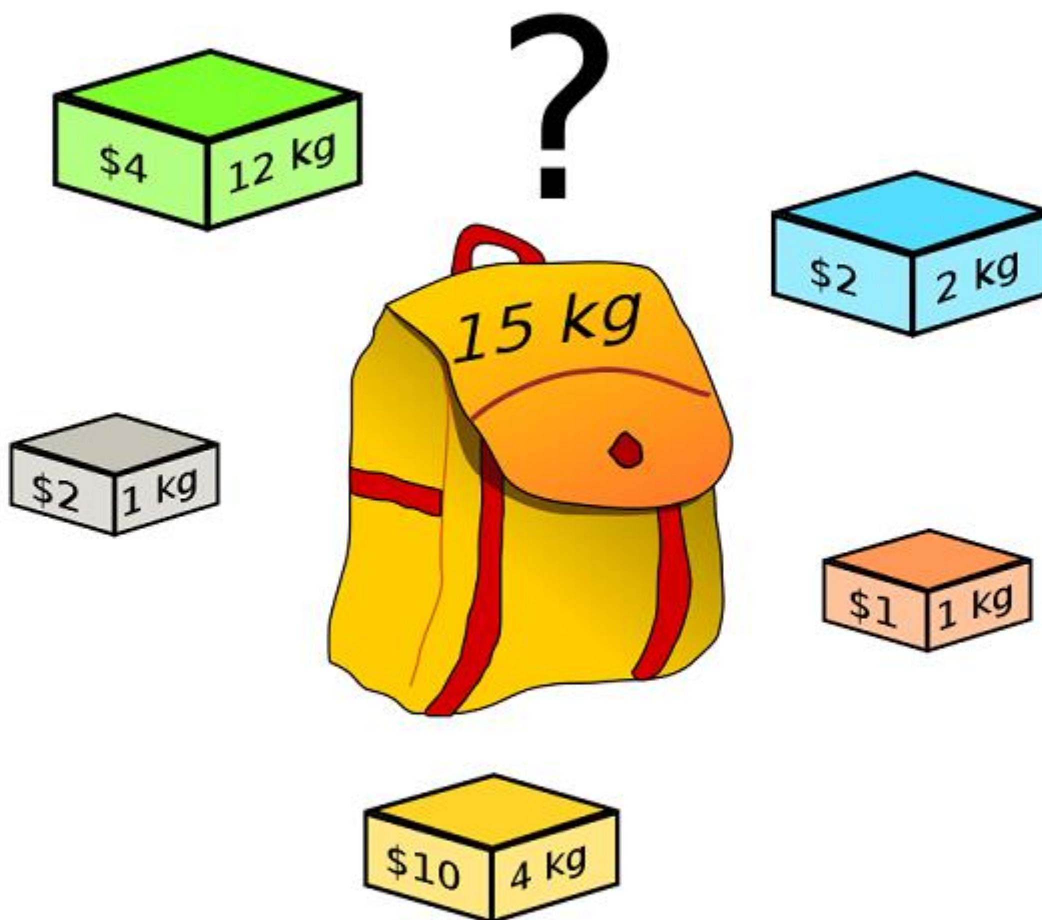
- Knapsack Problem
- Job Scheduling Problem with dead lines
- Optimal Storage on Tapes
- Optimal Merge Pattern

The Knapsack problem

I found the Knapsack problem tricky and interesting at the same time. I am sure if you are visiting this page, you already know the problem statement but just for the sake of completion :

Problem:

Given a Knapsack of a maximum capacity of W and N items each with its own value and weight, throw in items inside the Knapsack such that the final contents has the maximum value. Yikes !!



Here's the general way the problem is explained – Consider a thief gets into a home to rob and he carries a knapsack. There are fixed number of items in the home – each with its own weight and value – Jewellery, with less weight and highest value vs tables, with less value but a lot heavy. To add fuel to the fire, the thief has an old knapsack which has limited capacity. Obviously, he can't split the table into half or jewellery into 3/4ths. He either takes it or leaves it

Example :

```
Knapsack Max weight       :       W = 10 (units)
```

```
Total items               :       N = 4

Values of items           :       v[] = {10, 40, 30, 50}

Weight of items           :       w[] = {5, 4, 6, 3}
```

A cursory look at the example data tells us that the max value that we could accommodate with the limit of max weight of 10 is 50 + 40 = 90 with a weight of 7.

- Algorithm:
  - Assume knapsack holds weight W and items have value v_i and weight w_i
  - Rank items by value/weight ratio: v_i / w_i
    - Thus: $v_i / w_i \geq v_j / w_j$, for all $i \leq j$
  - Consider items in order of decreasing ratio
  - Take as much of each item as possible
- Code:

```
-- Assumes value and weight arrays are sorted by vᵢ/wᵢ
Fractional-Knapsack(v, w, W)
    load := 0
    i := 1
    while load < W and i ≤ n loop
        if wᵢ ≤ W - load then
            take all of item i
        else
            take (W-load) / wᵢ of item i
        end if
        add weight of what was taken to load
        i := i + 1
    end loop
    return load
```

- Example: Knapsack Capacity W = 30 and

Item	A	B	C	D
Value	50	140	60	60
Size	5	20	10	12
Ratio	10	7	6	5

-
- Solution:
  - All of A, all of B, and ((30-25)/10) of C (and none of D)
  - Size: 5 + 20 + 10*(5/10) = 30

- Value: $50 + 140 + 60*(5/10) = 190 + 30 = 220$

- Time: $\Theta(n)$, if already sorted

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute maximum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
            c[i, w] = vi + c[i-1, w-wi]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

JOB SEQUENCING WITH DEADLINES

- Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline.
- These jobs need to be ordered in such a way that there is maximum profit.
- It may happen that all of the given jobs may not be completed within their deadlines.
- Assume, deadline of **i**[th] job J_i is d_i and the profit received from this job is p_i.
- Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.
- Thus, D(i)>0D(i)>0 for 1⩽i⩽n1⩽i⩽n.
- Initially, these jobs are ordered according to profit, i.e. p1⩾p2⩾p3⩾...⩾pnp1⩾p2⩾p3⩾...⩾pn.

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

D(0) := J(0) :=

0 k := 1

J(1) := 1 // means first job is

selected for i = 2 … n do

r := k

while D(J(r)) > D(i) and D(J(r)) ≠ r

do r := r – 1

```
if D(J(r)) ≤ D(i) and D(i) > r
  then for l = k ... r + 1 by -1
  do

    J(l + 1) := J(l)

    J(r + 1) := i

    k := k + 1
```

- Let us consider a set of given jobs as shown in the following table.
- We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit.
- Each job is associated with a deadline and profit.

Job	**J_1**	**J_2**	**J_3**	**J_4**	**J_5**
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

- To solve this problem, the given jobs are sorted according to their profit in a descending order.
- Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J_2**	**J_1**	**J_4**	**J_3**	**J_5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.

- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (***J_2, J_1, J_3***), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

OPTIMAL STORAGE ON TAPES

- **Input:** We are given 'n' problem that are to be stored on computer tape of length
    - L and the length of program i is Li
- Such that 1 ≤i≤ n and Σ 1≤k≤j Li≤ 1
- **Output:** A permutation from all n! For the n programs so that when they are stored on tape in the order the MRT is minimized.
- **Example:**
- Let n = 3, (l1, l2, l3) = (8, 12, 2). As n = 3, there are 3! =6 possible ordering.
- All these orderings and their respective d value are given below:

Ordering	**d (i)**	**Value**
1, 2, 3	8 + (8+12) + (8+12+2)	50
1, 3, 2	8 + 8 + 2 + 8 + 2 + 12	40
2, 1, 3	12 + 12 + 8 +12 + 8 + 2	54
2, 3, 1	12 + 12 +2 +12 + 2 + 8	48
3, 1, 2	2 + 2 + 8 + 2 + 8+ 12	34
3, 2, 1	2 + 2 +12 + 2 + 12 + 8	38

The optimal ordering is 3, 1, 2.

- The greedy method is now applied to solve this problem.
- It requires that the programs are stored in non-decreasing order which can be done in O (nlogn) time.

Greedy solution:

i. Make tape empty

ii. Fori = 1 to n do;

iii. Grab the next shortest path

iv. Put it on next tape.

The algorithm takes the best shortest term choice without checking to see whether it is a big long term decision.

Algorithm:

```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
        {
        Write (i, k); // "Assign program", j, "to tape", k;
        k = (k +1) mod m;
        }
}
```

OPTIMAL MERGE PATTERNS

- **Optimal merge pattern** is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.
- If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
- There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.

- The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, `f (i)` represents the number of records in each file and `d (i)` represents the depth.

```
Algorithm Tree(n)

//list is a global list of n single node

{

       For  i=1 to i= n-1 do

       {

              // get a new tree node
              Pt: new treenode;

              // merge two trees with smallest length
              (Pt = lchild) = least(list);

              (Pt = rchild) = least(list);
```

Algorithm for optimal merge pattern

Example:

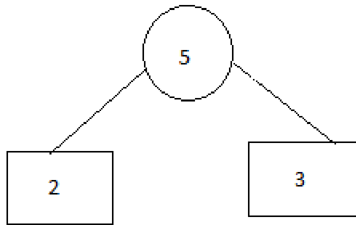
* Given a set of unsorted files: 5, 3, 2, 7, 9, 13
* Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13
* After this, pick two smallest numbers and repeat this until we left with only one number.

Now follow following

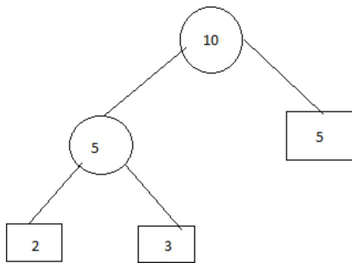
steps: Step 1: Insert 2, 3

2 3

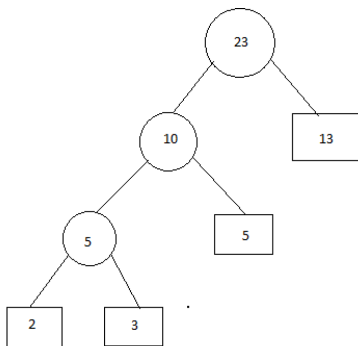
Step 2:



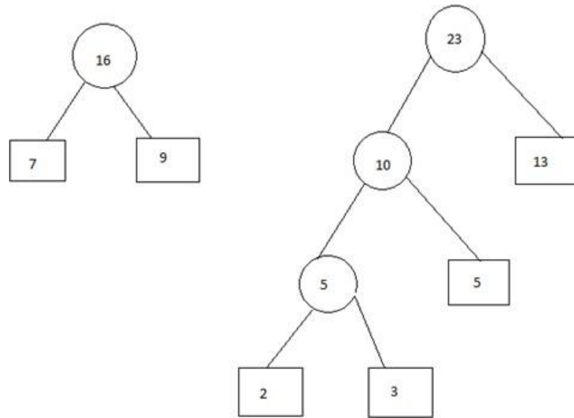
Step 3: Insert 5



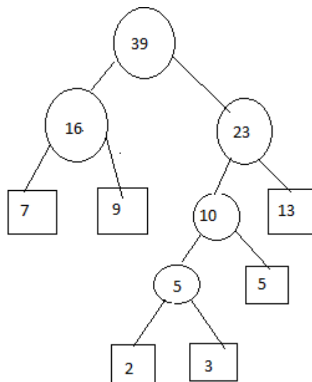
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



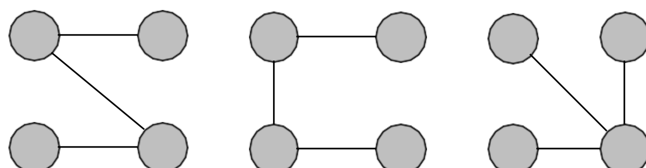
So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93

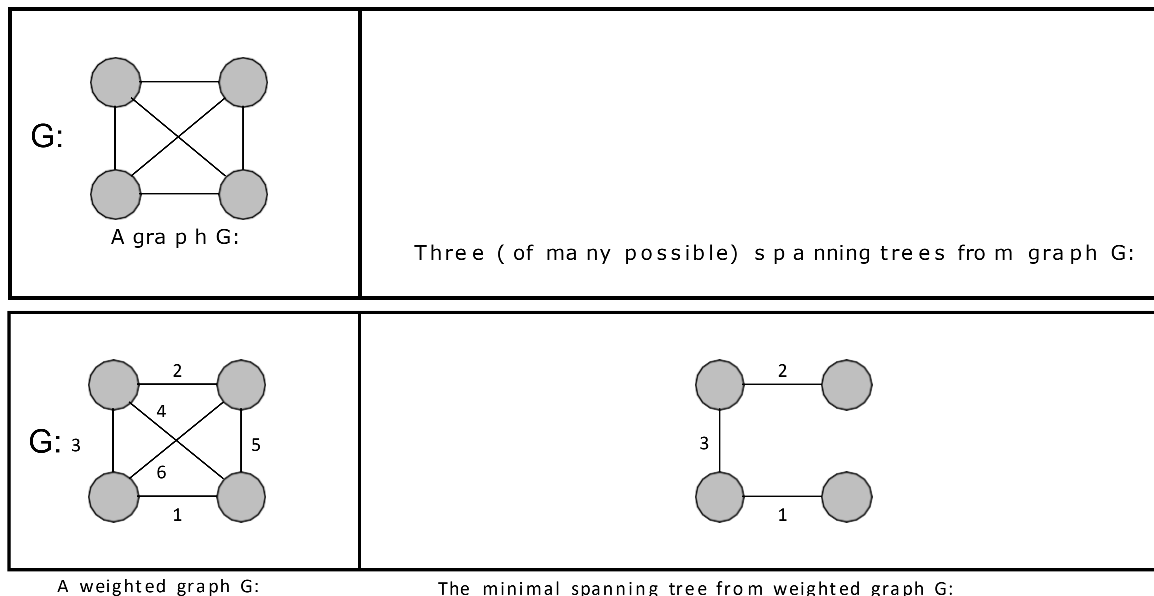
Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w(T) is the sum of weights of all edges in T. Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:





Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal's algorithm and
2. Prim's algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections* in determining the MST. In *Prim's algorithm at any instance of output it represents tree* whereas in *Kruskal's algorithm at any instance of output it may represent tree or not.*

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

The order in which the edges are chosen, in this case, does not matter. Different MST's

may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T

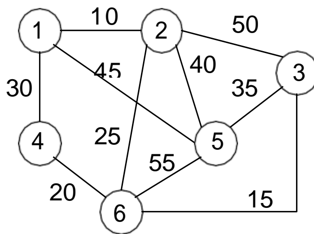
   then Add (v, w) to T

   else discard (v, w)

6. If T contains fewer than n - 1 edges then print no spanning tree.

Example 1:

Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

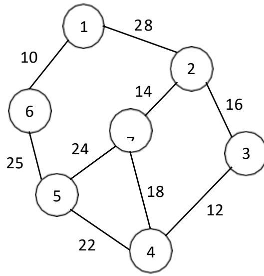
Cost	***10***	***15***	***20***	***25***	***30***	***35***	***40***	***45***	***50***	***55***
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:



Solution:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***12***	***14***	***16***	***18***	***22***	***24***	***25***	***28***
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	**COST**	**STAGES IN KRUSKAL'S ALGORITHM**	**REMARKS**
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

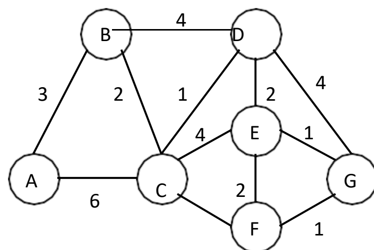
E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or ∝ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for  i :=1 to n do                              // Initialize near
                if (cost [i, l] < cost [i, k]) then near [i] := l;
                else near [i] := k;
        near [k] :=near [l] := 0;
        for  i:=2 to n - 1 do                           // Find n - 2 additional edges for t.
        {
                Let j be an index such that near [j] ≠ 0 and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j];
                mincost := mincost + cost [j, near [j]];
                near [j] := 0
                for  k:= 1 to n do                              // Update near[].
                        if ((near [k] ≠ 0) and (cost [k, near [k]] > cost [k, j]))
                                then near [k] := j;
        }
        return mincost;
}
```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



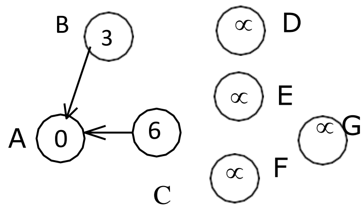
Solution:

The cost adjacency matrix is

$$\begin{pmatrix} 0 & 3 & 6 & \propto & \propto & \propto & \propto \\ 3 & 0 & 2 & 4 & \propto & \propto & \propto \\ 6 & 2 & 0 & 1 & 4 & 2 & \propto \\ \propto & 4 & 1 & 0 & 2 & \propto & 4 \\ \propto & \propto & 4 & 2 & 0 & 2 & 1 \\ \propto & \propto & 2 & \propto & 2 & 0 & 1 \\ \propto & \propto & \propto & 4 & 1 & 1 & 0 \end{pmatrix}$$

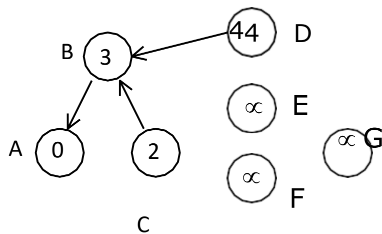
The stepwise progress of the prim's algorithm is as follows:

Step 1:



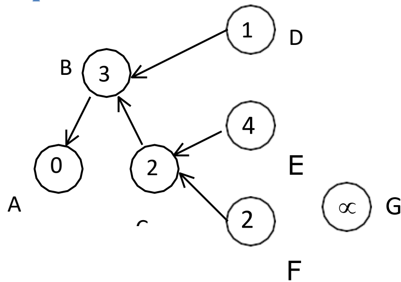
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



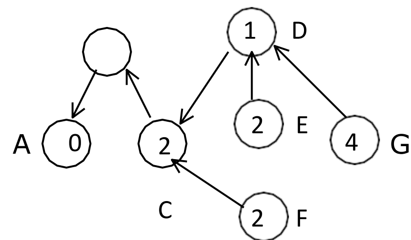
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



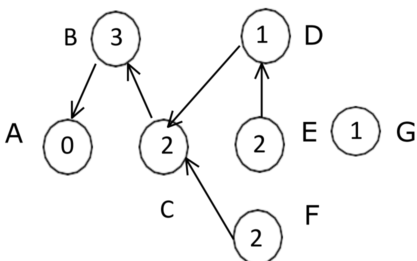
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step 4:



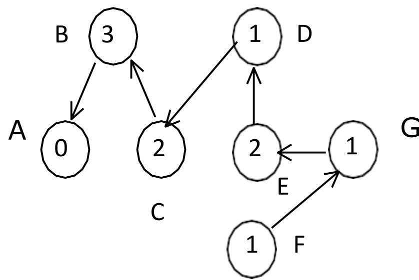
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



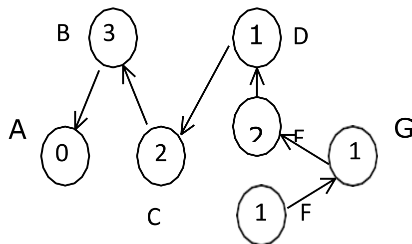
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

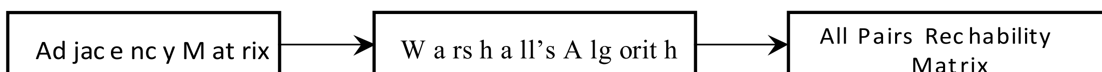
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Reachability Matrix (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.



It begins with the adjacency matrix for the given graph, which is called A_0, and then updates the matrix 'n' times, producing matrices called A_1, A_2,..........., A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i–paths. A one entry in the matrix A_i will correspond to the existence of i–paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n, contains the desired connectivity information.

A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix or path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

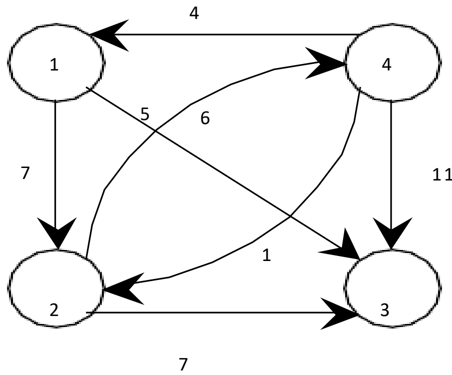
$$Ai [x, y] = Ai\text{-}1 [x, y] \vee (Ai\text{-}1 [x, i] \wedge Ai\text{-}1 [i, y]) \quad ---- \quad (1)$$

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph 'A_0'

$$A_0 = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

The first step is to compute 'A_1' matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A_0 must remain one in A_1, since in Boolean algebra 1 + (anything) = 1. Since these are only nine zero entries in A_0, there are only nine entries in A_0 that need to be updated.

A1[1, 1] = A0[1, 1] ∨ (A0[1, 1] ∧ A0[1, 1]) = 0 ∨ (0 ∧ 0) = 0

A1[1, 4] = A0[1, 4] ∨ (A0[1, 1] ∧ A0[1, 4]) = 0 ∨ (0 ∧ 0) = 0

A1[2, 1] = A0[2, 1] ∨ (A0[2, 1] ∧ A0[1, 1]) = 0 ∨ (0 ∧ 0) = 0

A1[2, 2] = A0[2, 2] ∨ (A0[2, 1] ∧ A0[1, 2]) = 0 ∨ (0 ∧ 1) = 0

A1[3, 1] = A0[3, 1] ∨ (A0[3, 1] ∧ A0[1, 1]) = 0 ∨ (0 ∧ 0) = 0

A1[3, 2] = A0[3, 2] ∨ (A0[3, 1] ∧ A0[1, 2]) = 0 ∨ (0 ∧ 1) = 0

A1[3, 3] = A0[3, 3] ∨ (A0[3, 1] ∧ A0[1, 3]) = 0 ∨ (0 ∧ 1) = 0

A1[3, 4] = A0[3, 4] ∨ (A0[3, 1] ∧ A0[1, 4]) = 0 ∨ (0 ∧ 0) = 0

A1[4, 4] = A0[4, 4] ∨ (A0[4, 1] ∧ A0[1, 4]) = 0 ∨ (1 ∧ 0) = 0

$$A_1 = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Next, A2 must be calculated from A1; but again we need to update

the 0 entries, A2[1, 1] = A1[1, 1] ∨ (A1[1, 2] ∧ A1[2, 1]) = 0 ∨ (1 ∧

0) = 0

A2[1, 4] = A1[1, 4] ∨ (A1[1, 2] ∧ A1[2, 4]) = 0 ∨ (1 ∧ 1) = 1

A2[2, 1] = A1[2, 1] ∨ (A1[2, 2] ∧ A1[2, 1]) = 0 ∨ (0 ∧ 0) = 0

A2[2, 2] = A1[2, 2] ∨ (A1[2, 2] ∧ A1[2, 2]) = 0 ∨ (0 ∧ 0) = 0

A2[3, 1] = A1[3, 1] ∨ (A1[3, 2] ∧ A1[2, 1]) = 0 ∨ (0 ∧ 0) = 0

A2[3, 2] = A1[3, 2] ∨ (A1[3, 2] ∧ A1[2, 2]) = 0 ∨ (0 ∧ 0) = 0

A2[3, 3] = A1[3, 3] ∨ (A1[3, 2] ∧ A1[2, 3]) = 0 ∨ (0 ∧ 1) = 0

A2[3, 4] = A1[3, 4] ∨ (A1[3, 2] ∧ A1[2, 4]) = 0 ∨ (0 ∧ 1) = 0

A2[4, 4] = A1[4, 4] ∨ (A1[4, 2] ∧ A1[2, 4]) = 0 ∨ (1 ∧ 1) = 1

$$A_2 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

This matrix has only seven 0 entries, and so to compute A_3, we need to do only seven computations.

A3[1, 1] = A2[1, 1] ∨ (A2[1, 3] ∧ A2[3, 1]) = 0 ∨ (1 ∧ 0) = 0

A3[2, 1] = A2[2, 1] ∨ (A2[2, 3] ∧ A2[3, 1]) = 0 ∨ (1 ∧ 0) = 0

A3[2, 2] = A2[2, 2] ∨ (A2[2, 3] ∧ A2[3, 2]) = 0 ∨ (1 ∧ 0) = 0

A3[3, 1] = A2[3, 1] ∨ (A2[3, 3] ∧ A2[3, 1]) = 0 ∨ (0 ∧ 0) = 0

A3[3, 2] = A2[3, 2] ∨ (A2[3, 3] ∧ A2[3, 2]) = 0 ∨ (0 ∧ 0) = 0

A3[3, 3] = A2[3, 3] ∨ (A2[3, 3] ∧ A2[3, 3]) = 0 ∨ (0 ∧ 0) = 0

A3[3, 4] = A2[3, 4] ∨ (A2[3, 3] ∧ A2[3, 4]) = 0 ∨ (0 ∧ 0) = 0

$$A_3 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Once A3 is calculated, we use the update rule to calculate A_4 and stop. This matrix is the reachability matrix for the graph.

A4[1, 1] = A3 [1, 1] ∨ (A3 [1, 4] ∧ A3 [4, 1]) = 0 ∨ (1 ∧ 1) = 0 ∨ 1 = 1

A4[2, 1] = A3 [2, 1] ∨ (A3 [2, 4] ∧ A3 [4, 1]) = 0 ∨ (1 ∧ 1) = 0 ∨ 1 = 1

A4[2, 2] = A3 [2, 2] ∨ (A3 [2, 4] ∧ A3 [4, 2]) = 0 ∨ (1 ∧ 1) = 0 ∨ 1 = 1

A4[3, 1] = A3 [3, 1] ∨ (A3 [3, 4] ∧ A3 [4, 1]) = 0 ∨ (0 ∧ 1) = 0 ∨ 0 = 0

A4[3, 2] = A3 [3, 2] ∨ (A3 [3, 4] ∧ A3 [4, 2]) = 0 ∨ (0 ∧ 1) = 0 ∨ 0 = 0

A4[3, 3] = A3 [3, 3] ∨ (A3 [3, 4] ∧ A3 [4, 3]) = 0 ∨ (0 ∧ 1) = 0 ∨ 0 = 0

A4[3, 4] = A3 [3, 4] ∨ (A3 [3, 4] ∧ A3 [4, 4]) = 0 ∨ (0 ∧ 1) = 0 ∨ 0 = 0

$$A_4 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself

GRAPH

Introduction to Graphs:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often c
e = |E|.

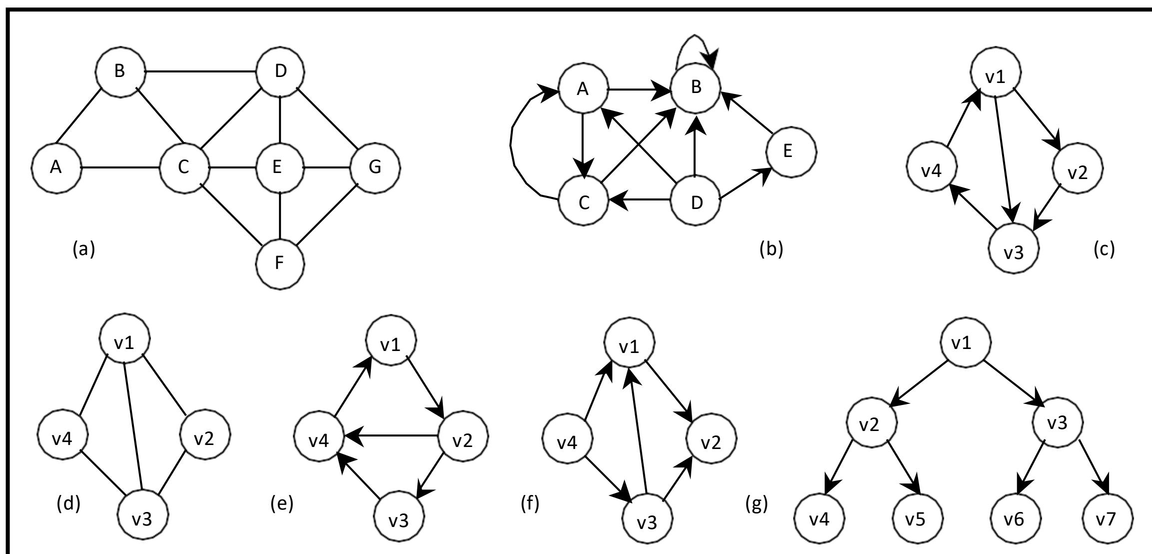
A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and
lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is
edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure
undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figu
directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there i
between any two nodes in G.

A graph G is said to be complete if every node a in G is adjacent to every other node v in G. A comple
nodes will have n(n-1)/2 edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graph

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes
path from u to v and also a path from v to u. On the other hand, G is said to be unilaterally connecte
(u, v) of nodes in G there is a path from u to v or a path from v to u. For example, the digraph shown
(e) is strongly connected.



We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph

In-degree and Out degree

The number of incoming edges to a vertex v is called in–degree of the vertex (denote indeg(v)). The number of outgoing edges from a vertex is called out-degree (denote outdeg(v)). For example, let us consider the digraph shown in figure 6.5.1(f),

indegree(v_1) = 2 outdegree(v_1) = 1

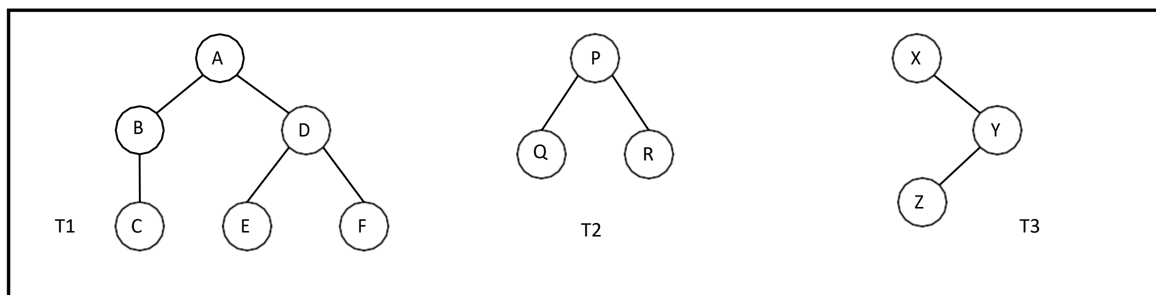
indegree(v_2) = 2 outdegree(v_2) = 0

A path is a sequence of vertices (v_1, v_2, , v_k), where for all i, (v_i, v_{i+1}) ε E. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph G′ = (V′, E′) is a sub-graph of graph G = (V, E) iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T1, T2 and T3.



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G. Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

Representation of Graphs:

There are two ways of representing digraphs. They are:

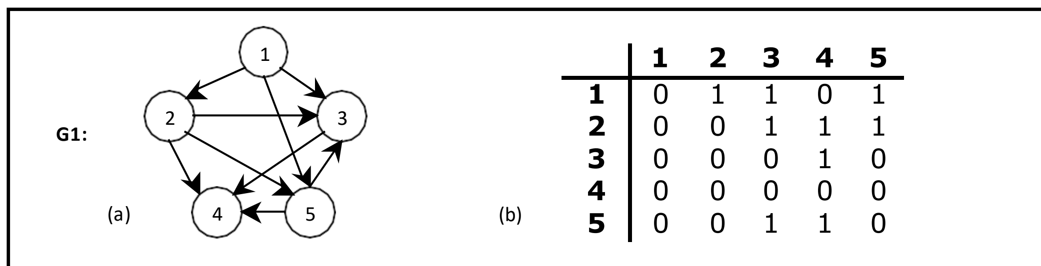
- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

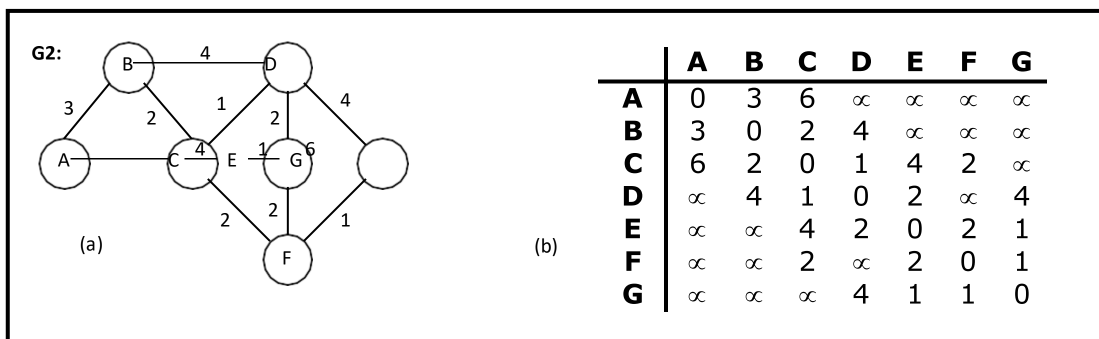


	1	2	3	4	5
1	0	1	1	0	1
2	0	0	1	1	1
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	1	1	0

Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G2 shown in figure 6.5.3(a).



	A	B	C	D	E	F	G
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝
D	∝	4	1	0	2	∝	4
E	∝	∝	4	2	0	2	1
F	∝	∝	2	∝	2	0	1
G	∝	∝	∝	4	1	1	0

Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, ... n] of pointers where for $1 \leq v \leq n$, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list in shown in figure 6.5.4 (b).

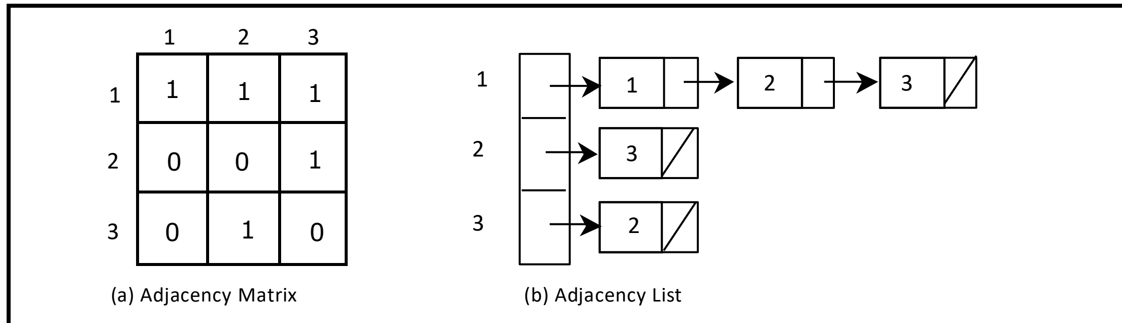


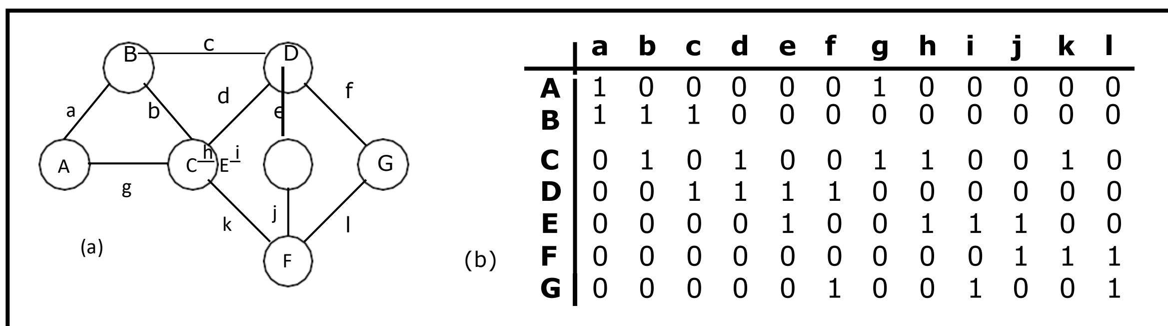
Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.



	a	b	c	d	e	f	g	h	i	j	k	l
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0
D	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	0	1	0	0	1	1	1	0	0
F	0	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	0	1	0	0	1	0	0	1

Figure 6.5.4 Graph and its incidence matrix

Figure 6.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 6.5.4(a).

Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

1. STATUS = 1 (Ready state): The initial state of the node N.

2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.

3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of neighbors of A. And so on. 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 accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A.

Initialize all nodes to the ready state (STATUS = 1).

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).

2. Repeat the following steps until QUEUE is empty:

   a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).

   b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

3. Exit.

Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFT except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).

2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).

3. Repeat the following steps until STACK is empty:

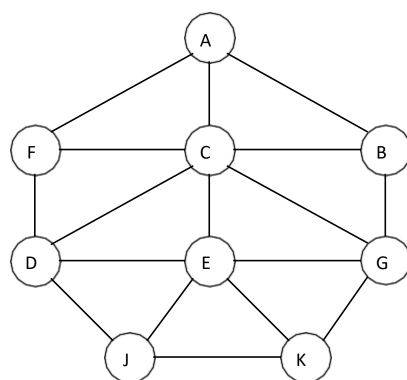
   a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).

   b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Gra p h G

Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Depth-first search and traversal:

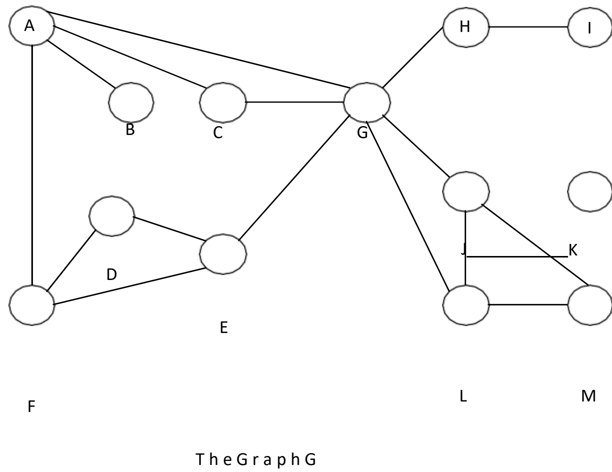
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

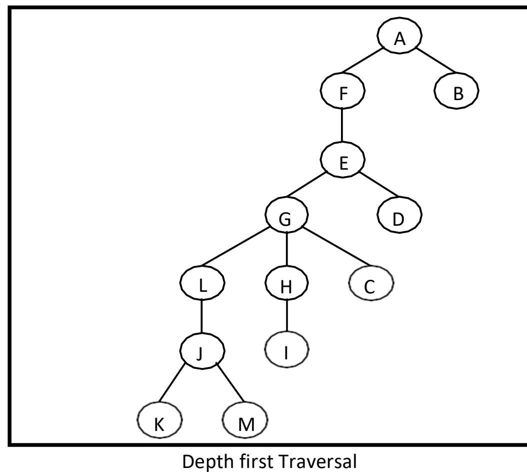


The Graph G

Node	Adjacency List
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

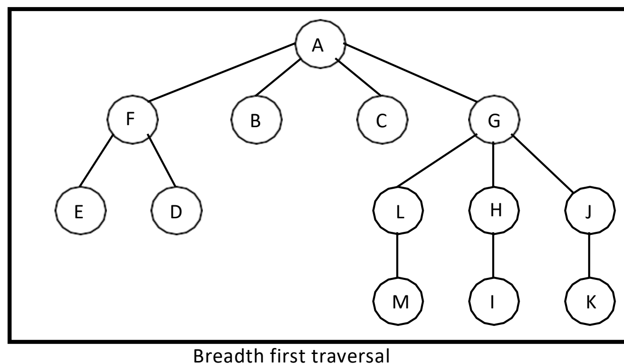
The adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F E G L J K M H I C D B***. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

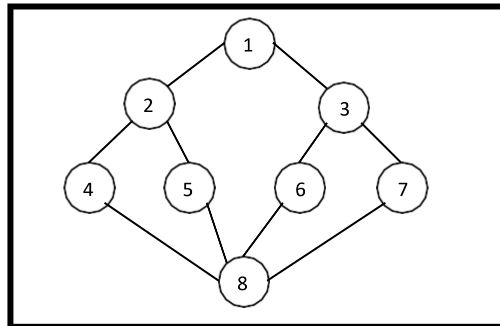
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F B C G E D L H J M I K***. The breadth first spanning tree is shown in the figure given below:



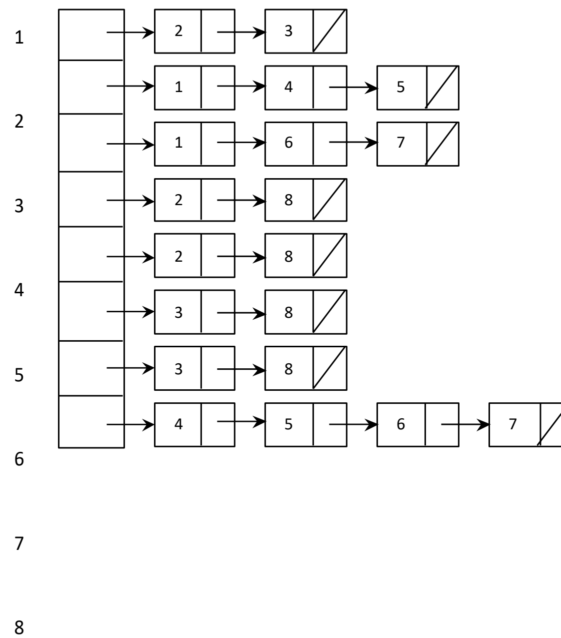
Breadth first traversal

Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



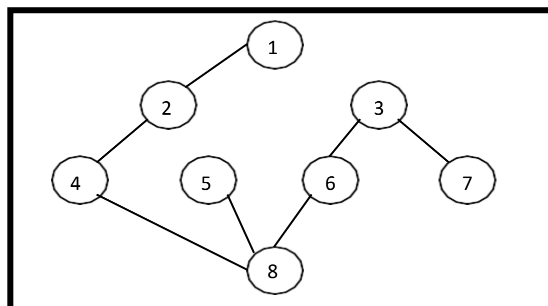
Graph G



Adjacency list f or graph G

Depth first search and traversal:

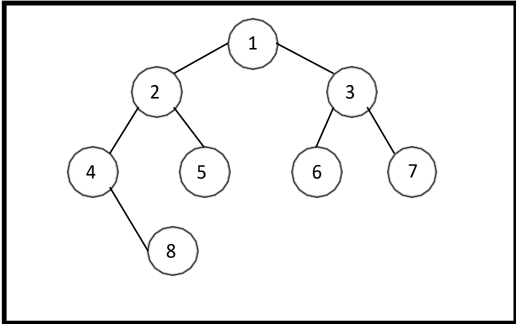
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:



Breadth First Spanning