SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I-Data Structures – SCSA1205

Unit-I

Introduction to Data vs Information - Data Structures - Classification – Abstraction - Abstract data types (ADT) - Array - characteristics - Storage Representations. Array Order Reversal- Recursion- Array operations, Algorithm-complexity – Time and Space trade off.

I. ALGORITHM

An algorithm is a well-defined computational procedure having well defined steps for solving a particular problem. Algorithm is finite set oflogic or instructions, written in order for accomplishing the certain predefined task.

Each algorithm must have:

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions:** The condition(s) on output.

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** An algorithm must have 0 or well-defined inputs.
- **Output:** An algorithm must have 1 or well-defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

The performance of algorithm is measured on the basis of following properties:

- **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity:** It is the amount of memory space required byan algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

2.INTRODUCTION TO DATA STRUCTURES

2.1 Data

In the modern context, data stands for both singular and plural. Data means a value or set of values. Data can be defined as an elementaryvalue or the collection of values, for example, student's name and its idare the data about the student.

2.2 Structure

A building is a structure. A bridge is structure. In general, a structureis made up of components. It has a form or shape. It is made up of parts.A structure is an arrangement of and relations between parts or elements.

2.3 Data Structures

A data structure is an arrangement of data elements. 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 aspectof Computer Science i.e. Operating System, Compiler Design, Artificial Intelligence, Graphics and many more.

2.3.1 Needs

As applications are getting complex and amount of data is increasing day by day, the following issues might be araised:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day tothe 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 in a web server, it fails to process the requests.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items arenot required to be searched and required data can be searched instantly.

Data structures are important for the following reasons

1. Data structures are used in almost every program or software system.
2. Specific data structures are essential ingredients of many efficient algorithms, and make possible

the management of huge amounts of data, such as large integrated collection of databases.

3. Some programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. element. Hence, using array may not be very efficient here.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place.

Abstraction: Data structure is specified by the ADT which provides alevel of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

3.Classification of Data Structure

The data structure is classified into two different types, primitive and non-primitive data structures is shown in Fig.1.

Primitive Data Structures

Simple data structure can be constructed with the help of primitivedata structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, pointers, structures, unions, etc. are examples of primitive data structures.

Non Primitive Data Structures

Non Primitive Data Structures are classified as linear or non-linear. Arrays, linked lists, queues and stacks are linear data structures. Trees and Graphs are non-linear data structures. Except arrays, all other data structures have many variations. Non Primitive data structure can be constructed with the help of any one of the primitive data structure andit is having a specific functionality. It can be designed by user.

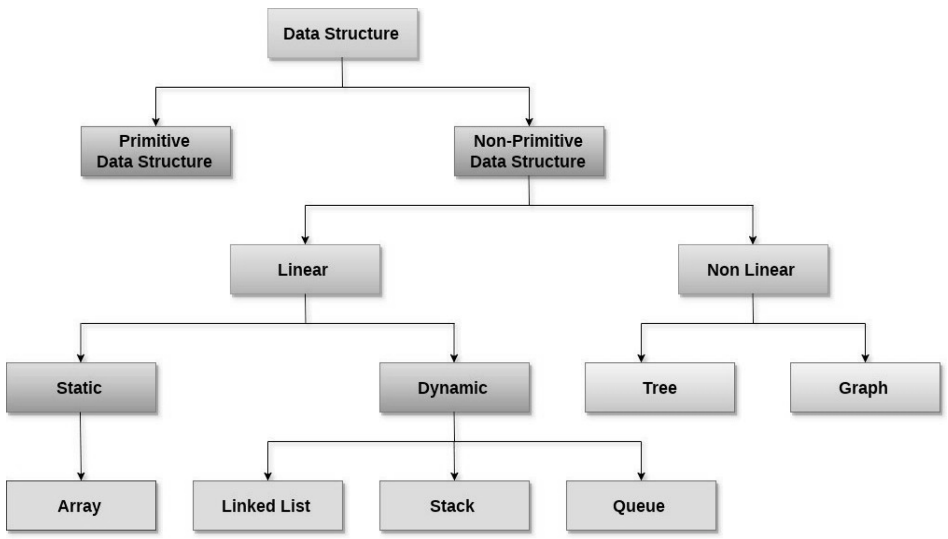


Fig. 1 Classification of Data Structure

Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element. In linear data structure the elements are stored in sequential order.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items stored in consecutive memory location and is referred by common name; each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The elements of array share the same variable name but each onecarries a different index number known as subscript.

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node. It is a collection of data of same data type but the data items need not be stored in consecutive.

Stack: Stack is a linear list in which insertion and deletions are allowedonly at one end, called top. It is a Last-In-First-Out linear data structure.

A stack is an abstract data type (ADT), can be implemented in mostof the programming languages. It is named as stack because it behaveslike a real-world stack, for example, piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only atone end called rear and deleted only at the other end called front. It isa First-In-First-Out Linear data structure.

It is an abstract data structure, similar to stack. Queue is opened atboth end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Operations applied on Linear Data Structure

The following list of operations applied on linear data structures

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

Non-linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement.The data elements are not arranged in sequential structure. Elements are stored based on the hierarchical relationship among the data. The following are some of the Non-Linear data structures.

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottom most nodes are called leaf node while the top most node is called root node. Each node contains pointers to point adjacent nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Operations applied on non-linear data structures

The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

4. ABSTRACT DATA TYPES

Abstract Data Type (ADT) is a type or class for objects whose behaviour is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. The process of providing only the essentials and hiding the details is known as abstraction is shown in fig,2.

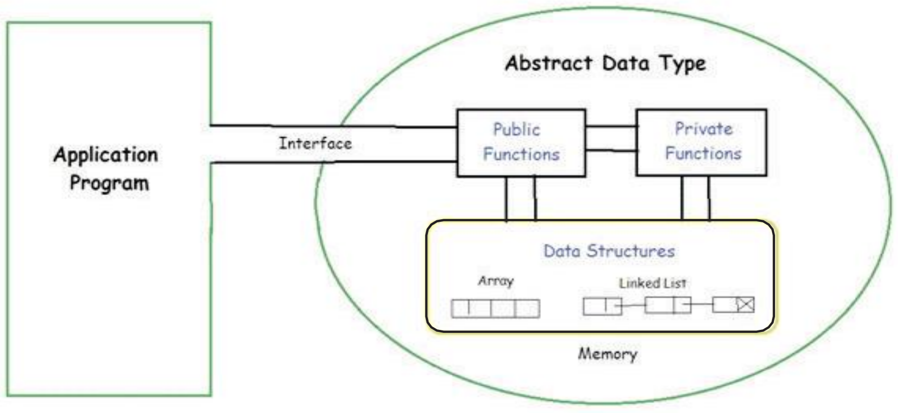


Fig. 2 Abstract Data Type

4.1 List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list is shown in fig.3.

- The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

```
//List ADT Type Definitions
    typedef struct node{

  void *DataPtr;
struct node *link;

    } Node;
typedef struct{
   int count;
  Node *pos;
 Node *head;
  Node *rear;

     int (*compare) (void *argument1, void *argument2);
```

Fig.3 List ADT

A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get() – Return an element from the list at any given position.

- insert() – Insert an element at any position of the list.

- remove() – Remove the first occurrence of any element from anon-empty list.
- removeAt() – Remove the element at a specified location from anon-empty list.
- replace() – Replace an element at any position by anotherelement.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise returnfalse.
- isFull() – Return true if the list is full, otherwise return false.

The List ADT Functions is shown in Fig.4.

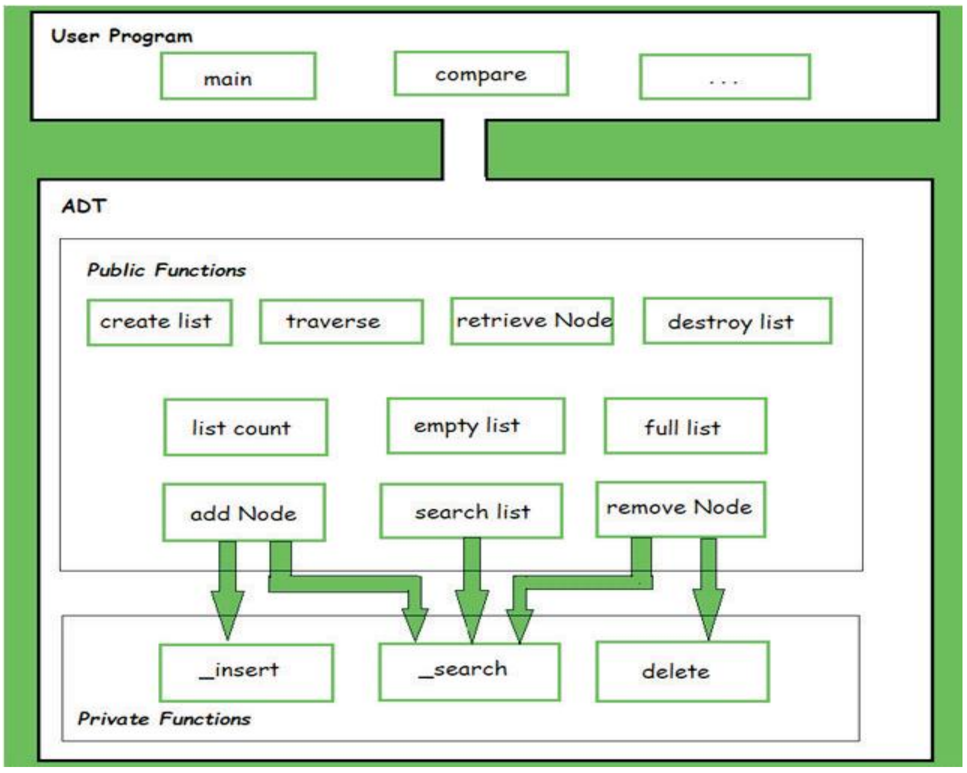


Fig. 4 List ADT Functions

4.2 Stack ADT

* In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
* The program allocates memory for the data and address is passed to the stack ADT is shown in Fig.5.
* The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
* The stack head structure also contains a pointer to top and countof number of entries currently in stack.
* A Stack contains elements of the same type arranged in sequential order.

```
//Stack ADT Type Definitions
typedefstruct node{

void *DataPtr;
struct node *link;

} StackNode;
typedefstruct{
int count;
StackNode *top;
```

Fig. 5 Stack ADT

All operations take place at a single end that is top of the stackand following operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack,if it is not empty.
- peek() – Return the element at the top of the stack withoutremoving it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise returnfalse.
- isFull() – Return true if the stack is full, otherwise return false.

4.3 Queue ADT

∗ The queue Abstract Data Type (ADT) follows the basic design of the stack abstract data type is shown in fig.6.

∗ A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at theend and deletion is done at the front. Following operations can be performed.

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue,if the queue is not empty.
- peek() – Return the element of the queue without removing it,if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise returnfalse.
- isFull() – Return true if the queue is full, otherwise return false.

```
//Queue ADT Type Definitions
typedefstructnode {
void*DataPtr;

 structnode *next;

} QueueNode;
typedefstruct {
QueueNode *front;
QueueNode *rear;
intcount;
```

Fig 6. Queue ADT

6.ARRAYS

The number of data items chunked together into a unit is known as data structure. When the data structure consists simply a sequence of data items, the data structure of this simple variety is referred as an array.

Definition: Array is a collection of homogenous (same data type) dataelements that are stored in contiguous memory locations.

Array Syntax

Syntax to declare an array:

∗ dataType [ ] arrayName;

arrayName= new dataType[n]; //keyword new performs dynamicmemory location

(or)

∗ dataType [ ] arrayName = new dataType[n];

Example:

int [ ] x; x=new int [10];(or)

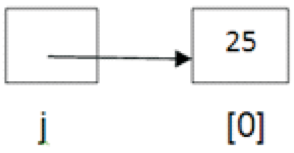
int [] x=new int [10];

Array Initialization

The values of an array can be initialized as follows,

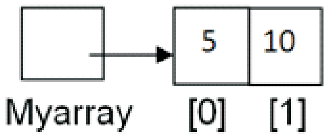
Example 1:

int [] j=new int [1]; j[0] =10;**(or)**int [] j= {25};



Example 2:

int [] myarray= {5, 10};



5.1 Characteristics of an Array

The following are the characteristics of an array data structure.

(i) Array stores elements of same data type.

(ii) The elements of an array are stored in subsequent memorylocations.

(iii) The name of array represents the starting address of the elements.

(iv) When data objects are stored in array, individual objects areselected by an index.

(v) By default an array index starts from 0 and ends with (n-1). Indexis also referred as subscripts.

(vi) The size of the array is mentioned at the time of declaring array.

(vii) While declaring 2D array, number of columns should be specifiedwhereas for number of rows there is no such rule.

(viii) Size of array can’t be changed at run time.

5.2Array Types

1.One-Dimensional Array or Linear arrays

2. Multi-Dimensional Array

3.Two dimensional (2D) Arrays or Matrix Arrays

5.2.1 One-Dimensional Array

In one dimensional array each element is represented by a single subscript. The elements are stored in consecutive memory locations. E.g.A [1], A [2], ..., A [N].

5.2.2 Two dimensional (2-D) arrays or Matrix Arrays

In two dimensional arrays each element is represented by two subscripts. Thus a two dimensional m x n array has m rows and n columns and contains m * n elements. It is also called matrix array because in this case, the elements form a matrix. For example A [4] [3]has 4 rows and 3 columns and 4*3 = 12 elements.

```
int [] [] A = new int [4] [3];
```

5.2.3 Multi dimensional arrays:

In it each element is represented by three subscripts. Thus a three dimensional m x n x l array contains m * n * l elements. For exampleA [2] [4] [3] has 2 * 4 * 3 = 24 elements.

6.STORAGE REPRESENTATION

An array is a set of homogeneous elements. Every element is referredby an index. Memory storage locations of the elements are not arrangedas a rectangular array with rows and columns. Instead, they are arranged in a linear sequence beginning with location 1, 2, 3 and so on. The elements are stored either column-by-column or row-by-row. The first one is called column-major order and later is referred as row-major order.

6.1 Row Major Order

The table 1 shows the linear arrangement of data in row major order.

Example

- Rows : 3
- Columns : 4

Data (A):

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

Table 1 Linear Arrangement of Array A in Row Major Order

Linear Arrangement of Array A												
Row	**1**				**2**				**3**			
Index	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(2,2)**	**(2,3)**	**(2,4)**	**(3,1)**	**(3,2)**	**(3,3)**	**(3,4)**
Memory	**100**	**102**	**104**	**106**	**108**	**110**	**112**	**114**	**116**	**118**	**120**	**122**
Data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

The formula is:

Location (A [j,k]) = Base Address of (A) + w [(N * (j-1)) + (k-1)]

Location (A [j, k]): Location of j^{th} row and k^{th} column

Base (A)	: Base Address of the Array A
w	: Bytes required to represent single element of the Array A

N	: Number of columns in the Array
j	: Row position of the element
k	: Column position of the element

Example

Suppose to find the address of (3,2) then

Base (A) = 100

w = 2 Bytes (integer type)

N = 4

j = 3

k = 2

Location (A [3, 2]) = 100 + 2 [(4 * (3-1) + (2-1)]

= 118

6.2 Column Major Order

The table 2 shows the linear arrangement of data in column major order.

- Rows : 3
- Columns : 4

Data (A):

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

Linear Arrangement of Array A												
Column	**1**			**2**			**3**			**4**		
Index	**(1,1)**	**(2,1)**	**(3,1)**	**(1,2)**	**(2,2)**	**(3,2)**	**(1,3)**	**(2,3)**	**(3,3)**	**(1,4)**	**(2,4)**	**(3,4)**
Memory	**100**	**102**	**104**	**106**	**108**	**110**	**112**	**114**	**116**	**118**	**120**	**122**
Data	**1**	**5**	**9**	**2**	**6**	**10**	**3**	**7**	**11**	**4**	**8**	**12**

Table 2 Linear Arrangement of Array A in Column Major Order

The formula for column major order is:

Location (A [j, k]) = Base Address of (A) + w [(M * (k-1)) + (j-1)]

Location (A [j, k]): Location of jth row and kth column

Base (A)	: Base Address of the Array A
w	: Bytes required to represent single element of the Array A
M	: Number of rows in the Array
j	: Row position of the element
k	: Column position of the element

Example

Base (A) = 100

w = 2 Bytes (integer type)

M = 3

j = 3

k = 2

Location (A [3, 2]) = 100 + 2 [(3 * (2-1) + (3-1)]

= 110

7.Array Order Reversal

Given an array (or string), the task is to reverse the array/string.

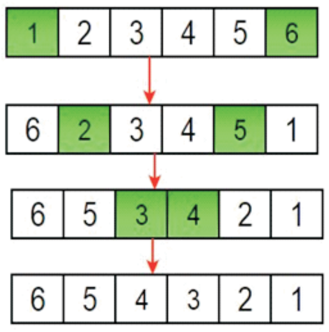
Examples:

Input : arr[] = {1, 2, 3}   Output : arr[] = {3, 2, 1}

Input : arr[] = {4, 5, 1, 2} Output : arr[] = {2, 1, 5, 4}

Algorithm

1) Initialize start and end indexes as start = 0, end = n-1.
2) In a loop, swap arr[start] with arr[end] and change start and end as follows : start = start +1, end = end – 1



8.Array Counting

1. Create a function arraycounting(array, size)

2. Find largest element in array and store it in max

3. Initialize count array with all zeros

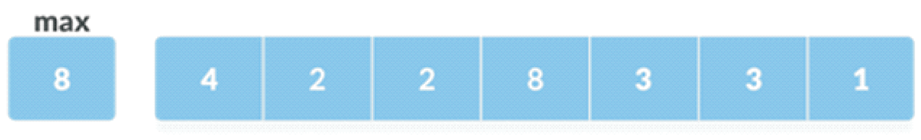
4. for j = 0 to size

5. Find the total count of each unique element and

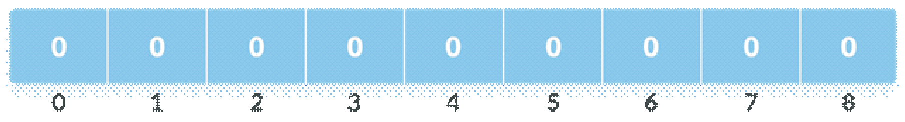
6. Store the count at jth index in count array

Example A={4,2,2,8,3,3,1}

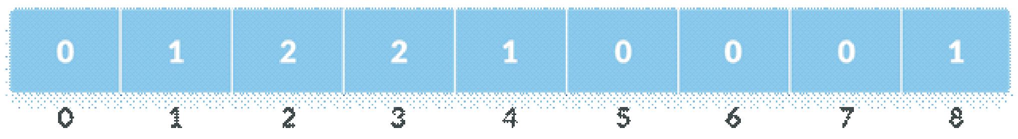
1. Find out the maximum element (let it be max) from the given array.



2. Initialize an array of length max+1 with all elements 0. This arrayis used for storing the count of the elements in the array.



3. Store the count of each element at their respective index in count array. For example: If the count of element "4" occurs 2 times then2 is stored in the 4th position in the count array. If element "5" isnot present in the array, then 0 is stored in 5th



8. Finding the maximum Number in a Set

Algorithm

* Read the array elements
* Initialize first element of the array as max.
* Traverse array elements from second and compare every elementwith current max
* Find the largest element in the array and assign it as max
* Print the largest element.

Example: A={56,78,34,23,70}

Step 1: Initialize max=0 , n=len(A)

```
Step 2: Repeat  step 3  until n
Step 3:  56>0 yes, Assign max=56
         78>56, yes  Assign max=78
         34>78, No
         23>78, No
         70>78, No
Step 4:  print Max       Output:78
```

9. RECURSION

Recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one calledto the first.

A simple recursive algorithm:

∗ Solves the base cases directly.

∗ Recurs with a simpler sub problem or sub problems.

∗ May do some extra work to convert the solution to the simpler subproblem into a solution to the given problem.

Some Example Algorithms

- Factorial
- All permutations
- Tree traversal
- Binary Search
- Quick Sort
- Towers of Hanoi

9.1 Design Methodology and Implementation of Recursion

∗ The recursive solution for a problem involves a two-way journey:

∗ First we decompose the problem from the top to the bottom

∗ Then we solve the problem from the bottom to the top.

Rules for Designing a Recursive Algorithm

(a) First, determine the base case.

(b) Then determine the general case.

(c) Combine the base case and the general cases into an algorithm.

(d) Each recursive call must reduce the size of the problem and moveit toward the base case.

(e) The base case, when reached, must terminate without a call to therecursive algorithm; that is, it must execute a return.

9.2 Broad Categories of Recursion

Recursion is a technique that is useful for defining relationships, andfor designing algorithms that implement those relationships. It is a natural way to express many algorithms in an optimized way. Recursive function is defined in terms of itself.

∗ Linear Recursion

∗ Binary Recursion

Linear Recursion:

Linear recursion is by far the most common form of recursion. In this style of recursion, the function calls itself repeatedly until it hits the termination condition (Base condition).

Binary Recursion

Binary recursion is another popular and powerful method. This formof recursion has the potential for calling itself twice instead of once asbefore. This is pretty useful in scenarios such as binary tree traversal,generating a Fibonacci sequence, etc.

Tail Recursion

A function call is said to be tail recursive if there is nothing to doafter the function returns except return its value. Since the current recursive instance is done executing at that point, saving its stack frameis a waste

For example the following C function print () is tail recursive.

```
// An example of tail recursive functionPrint (n) {
If (n < 0) return;
Display n;
Print (n-1);

}
```

10. FIBONACCI SERIES

Fibonacci Series generates subsequent number by adding two previous numbers. Fibonacci series starts from two numbers F0 & F1. The initial values of F0 & F1 can be

∗ 0 and 1

∗ 1 and 1 respectively.

The Fibonacci series looks like

F8 = 0 1 1 2 3 5 8 13

The algorithm for generating Fibonacci series can be drafted in 2 ways

1. Fibonacci Iterative Algorithm.

2. Fibonacci Recursive Algorithm.

Fibonacci RecursiveAlgorithm

```
Algorithm Fibo (n)
If n = 0
        Return 0Else If n = 1
Return 1
Else
Fibo (n) = Fibo (n-1) + Fibo (n-2)Return Fibo (n)
```

11.FACTORIAL

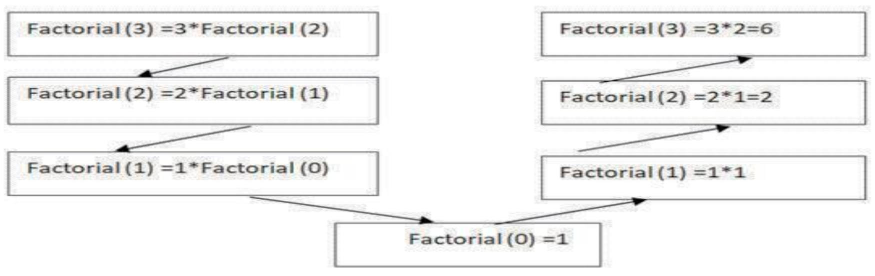
The factorial of a positive number is the product of the integral values from 1 to the number: Factorial of the given number can be calculated as

```
Algorithm
RecursiveFactorial (n)if (N equals 0)
Return 1
else
Return (n*recursiveFactorial (n-1))
end if
```

end recursiveFactorial

Calling a Recursive Factorial Algorithm with n=3

Fig.6 shows the steps for calculating the factorial using Recursion for n=3.



Factorial (3) =3*Factorial (2)
Factorial (2) =2*Factorial (1)
Factorial (1) =1*Factorial (0)
Factorial (0) =1
Factorial (1) =1*1
Factorial (2) =2*1=2
Factorial (3) =3*2=6

Fig 6. Factorial using Recursion Steps

Output: 6

12. TOWERS OF HANOI

Tower of Hanoi is a mathematical puzzle which consists of three tower (pegs) and more than one ring; as depicted in Fig.7.

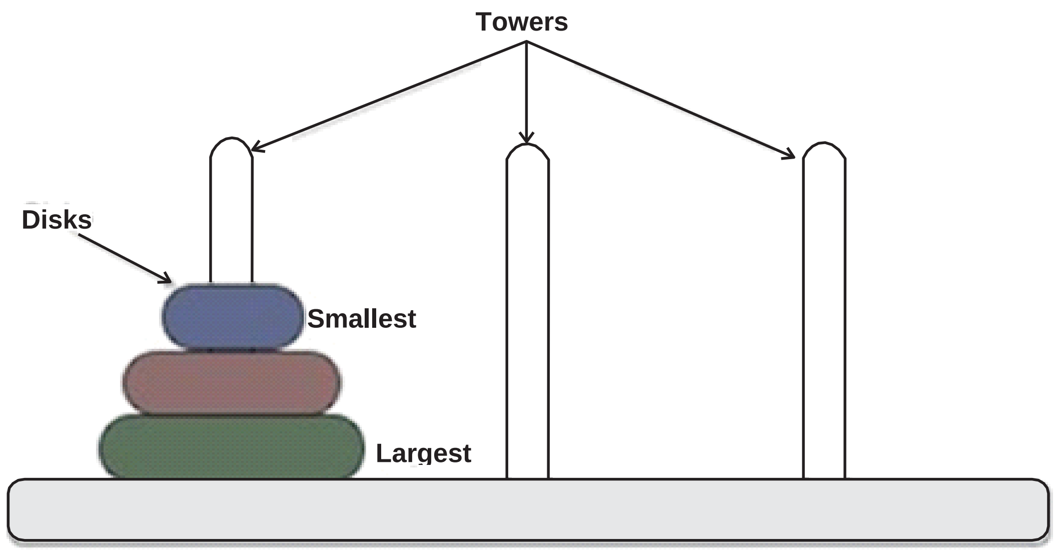


Fig.7 Tower of Hanoi

These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variationsof puzzle where the number of disks increases, but the tower count remains the same.

7.1 Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for Tower of Hanoi

* Only one disk can be moved among the towers at any given time.

* Only the “top” disk can be removed.

* No large disk can sit over a small disk.

Steps for solving the Towers of Hanoi problem

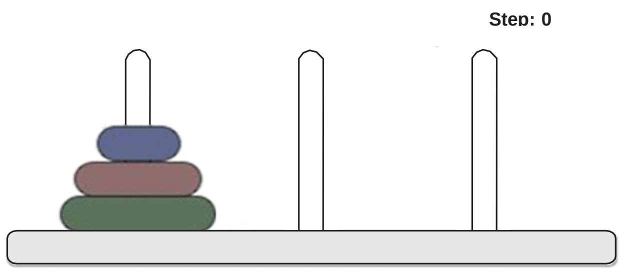
The following steps are to be followed.

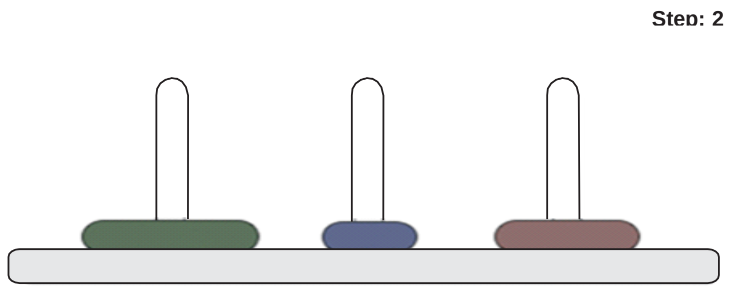
Step 1: Move n-1 disks from source to aux.

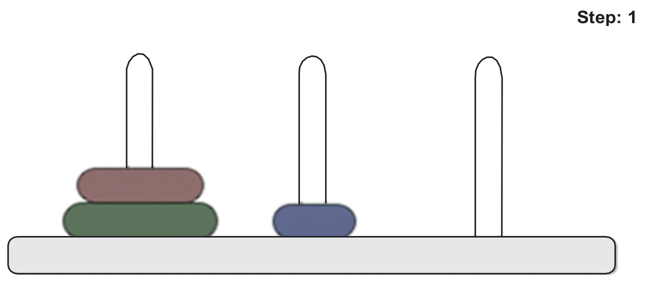
Step 2: Move nth disk from source to destination

Step 3: Move n-1 disks from aux to destination.

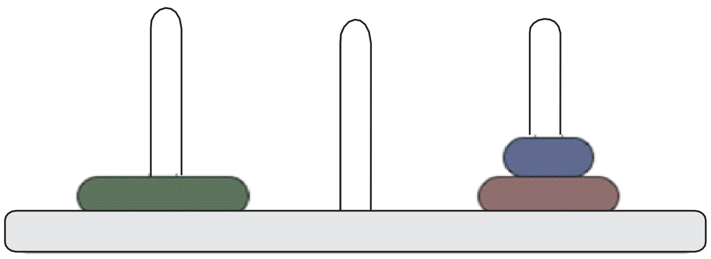
Fig.8 illustrates the step by step movement of the disks to implement Tower of Hanoi.



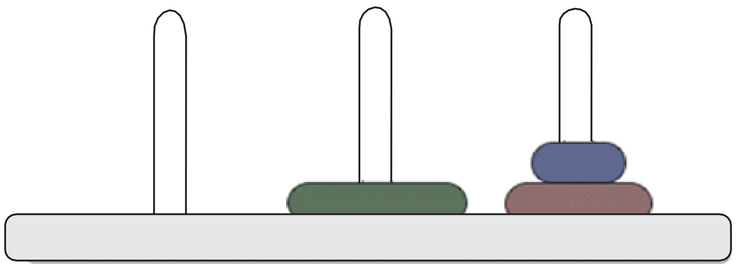




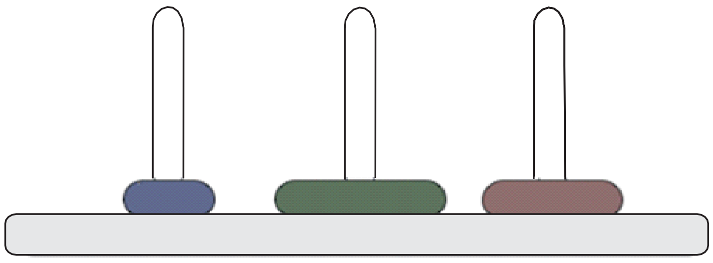
Step: 3



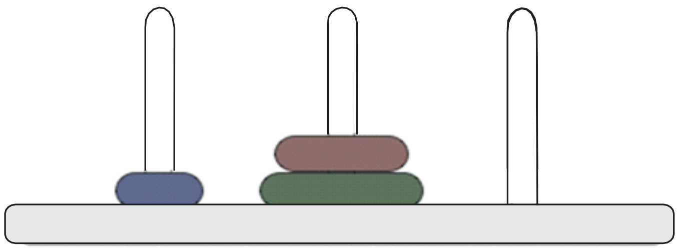
Step: 4



Step: 5



Step: 6



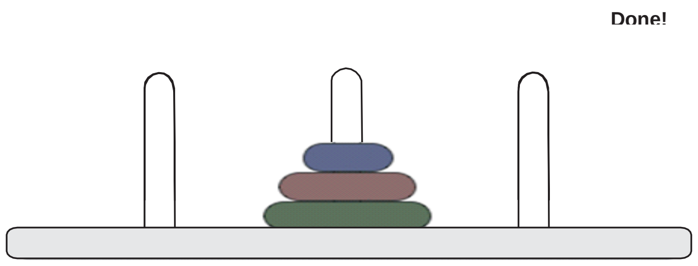


Fig.8 Tower of Hanoi

A recursive algorithm for Tower of Hanoi can be driven as followsSTART

Procedure **Hanoi** (disk, source, dest, aux)

IF disk = 0, THEN

Move disk from source to destELSE

Hanoi (disk-1, source, aux, dest) //Step1Move disk from source to dest //Step2 Hanoi (disk-1, aux, dest, source) //Step3 ENDIF

END

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited “A” Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II-Data Structures – SCSA1205

UNIT 2

LINKED LIST

Array Vs Linked List – Singly linked list - Representation of a linked list in memory - Operations on a singly linked list - Merging two singly linked lists into one list - Reversing a singly linked list – Polynomial Manipulation using List - Advantages and disadvantages of singly linked list - Circular linked list - Doubly linked list - Circular Doubly Linked List.

2.1 Array Vs Linked List

ARRAY	LINKED LIST
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, up to that element.
In an array, elements are stored in contiguous memory location or consecutive manner in the memory.	In a linked list, new elements can be stored anywhere in the memory. Address of the memory location allocated to the new element is stored in the previous node of linked list, hence forming a link between the two nodes/elements.
In array, Insertion and Deletion operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.Insertion and Deletion operations are fast in linked list.

Memory is allocated as soon as the array is declared, at compile time. It's also known as Static Memory Allocation.	Memory is allocated at runtime, as and when a new node is added. It's also known as Dynamic Memory Allocation.
ARRAY	**LINKED LIST**
In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can be single dimensional, two dimensional or multidimensional.	Linked list can be Linear(Singly) linked list, Doubly linked list or Circular linked list linked list.
Size of the array must be specified at time of array declaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the Stack section.	Whereas, linked list gets memory allocated in Heap section.

2.2 Linked List

A Linked list is a collection of elements called nodes, each of which stores two items called data or info and link or pointer field. Info is an element of the list and a link is a pointer to the next element. The linked list is also called a chain.

The different types of Linked lists are,

- *Singly linked list.*
- *Doubly linked list*
- *Circular linked list*

Singly Linked List

- The first node is the head node and it points to next node in the sequence.
- The last node’s reference is null indicating the end of the list is shown in Fig.2.1

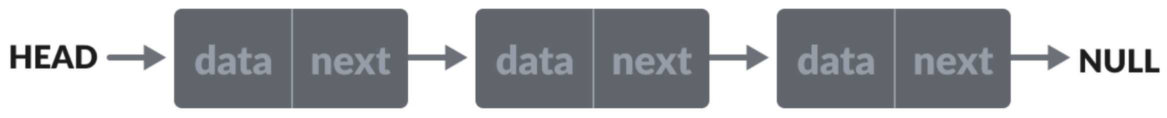


Fig.2.1 Singly Linked List

Doubly Linked List

- Every node has two pointers, one for pointing to next node and the other for pointing to the previous node.
- The **next** pointer of the last node and the previous pointer of the first node (head) are null is shown in Fig 2.2

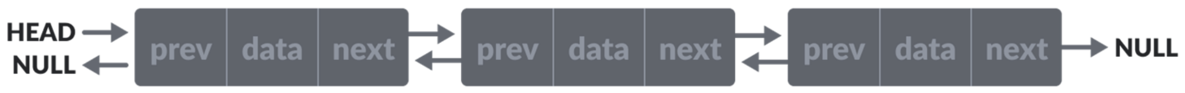


Fig.2.2 Doubly Linked List

Circular Linked List

- Circular Linked List is very similar to a singly linked list except that, here the last node points to the first node making it a circular list as shown in fig 2.3

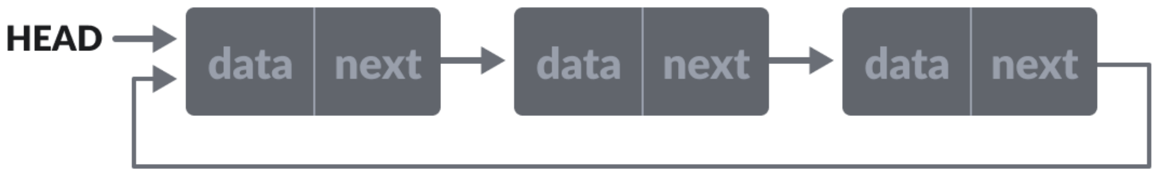


Fig.2.3 Circular Linked List

2.2.1 Singly Linked List

A singly linked list is a linked list in which each node contains only one link pointing to the next node in the list.

A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.

2.2.1.1 Representation of a linked list in memory

A linked list is a linear data structure consisting of a group of nodes where each node points to the next node by means of a pointer.

Each node is composed of **data** and a **reference to the next node** in the sequence. The last node has a reference to null which indicates the end of the linked list.

Head node is the starting node of the linked list(first node) and it contains the reference to the next node in the list. The head node will have a null reference when the list is empty. The fig 2.4 gives you an idea of how a Linked List looks.

A **linked list** is **represented** by a pointer to the first node of the **linked list**. The first node is called the head. If the **linked list** is empty, then the value of the head is NULL. In C, we can **represent** a node using structures.

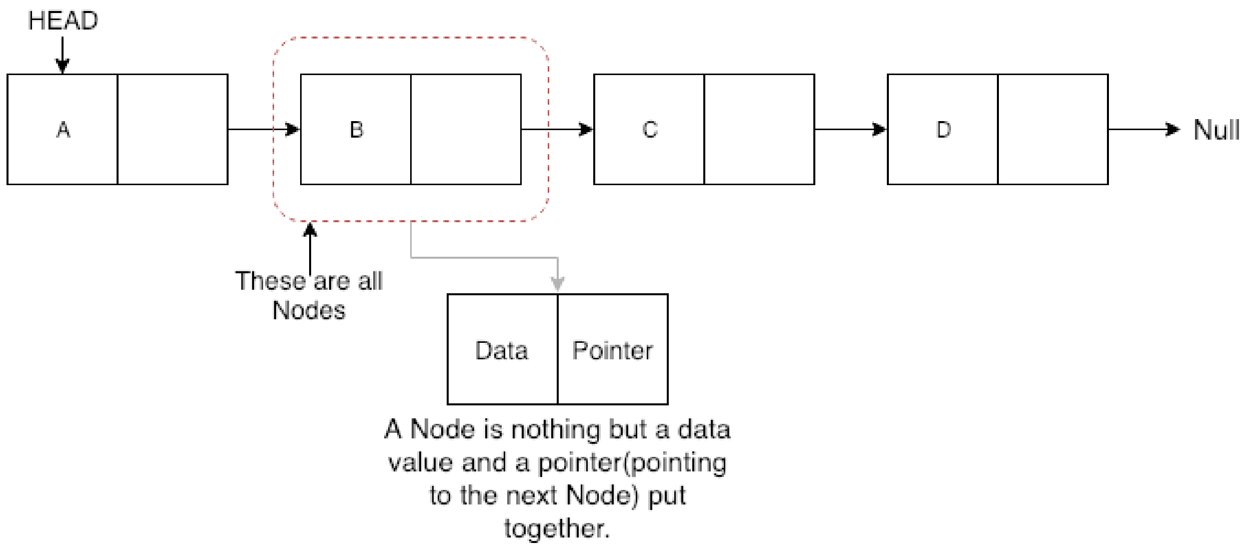

HEAD
A
B
C
D
Null
These are all Nodes
Data
Pointer
A Node is nothing but a data value and a pointer(pointing to the next Node) put together.

Fig 2.4 Singly Linked List representation

In the picture above we have a linked list, containing 4 nodes, each node has some data(A, B, C and D) and a pointer which stores the location of the next node.

In a singly linked list, the first node always pointed by a pointer called HEAD. If the link of the node points to NULL, then that indicates the end of the list.

Operations of Singly Linked List

The operations that are performed on a linear list are,

- Count the number of elements.
- Add an element at the beginning of the list.
- Add an element at the end of the list.
- Insert an element at the specified position in the list.
- Delete an element from the list.
- Search x in the list.
- Display all the elements of the list.

Add an element at the beginning of the list.

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.

Step 4 - If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Algorithm

```
 Addatbeg()
Begin
    Newnode->data=K
    Newnode->next=NULL
    If(Head==NULL)
        Head=Newnode
    Else
        Newnode->next=Head
        Head=Newnode
    Endif
End
```

The fig.2.5 shows how a node is added at the beginning of the linked list.

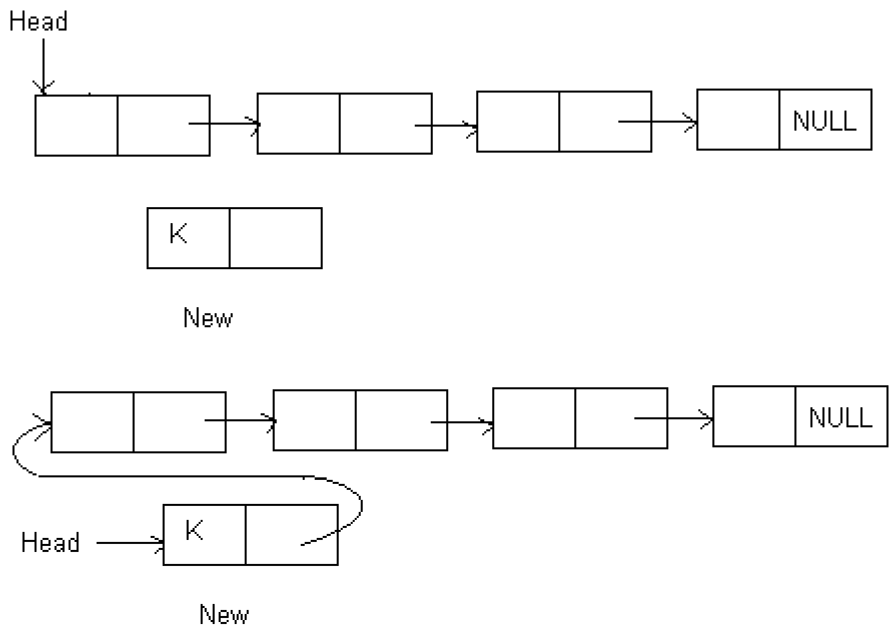


Fig 2.5 Adding node at the beginning of the linked list

Add an element at the end of the list.

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6 - Set **temp → next** = **newNode**.

Algorithm

```
Addatend()
Begin
    Newnode->data=K
    Newnode->next=NULL
    If(Head==NULL)
        Head=Newnode
    Else
        Temp=Head
        While(temp->next !=NULL)
            temp=temp->next
        endwhile
    temp->next=Newnode
    Endif
end
```

The Fig 2.6 shows how the node is added at the end of the list.

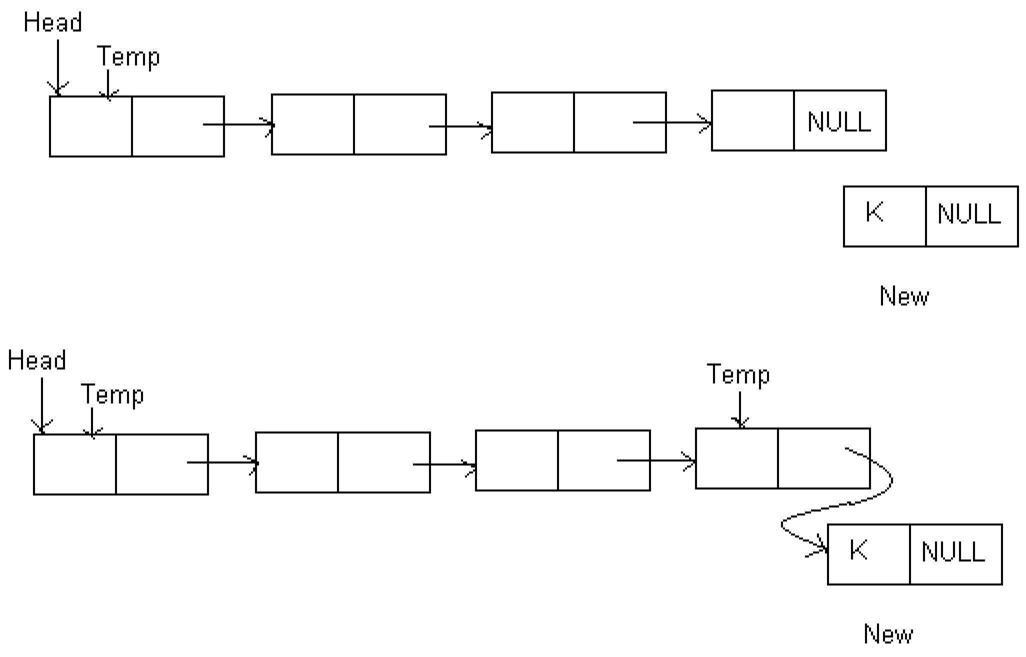


Fig 2.6 Node added at the end of the list

Insert an element at the specified position in the list:

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7 - Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

The operation 'Insert' inserts the given element x in the kth position. A temporary pointer Temp is created and made to point to Head. Now the Temp pointer is moved to the k – 1 th node. A new node with value x is created and the link of the new node is made to point to the position where the link of temp is pointing. Then the link of temp is made to point to the new node. Thus the given element is inserted in the position k is shown in fig.2.7.

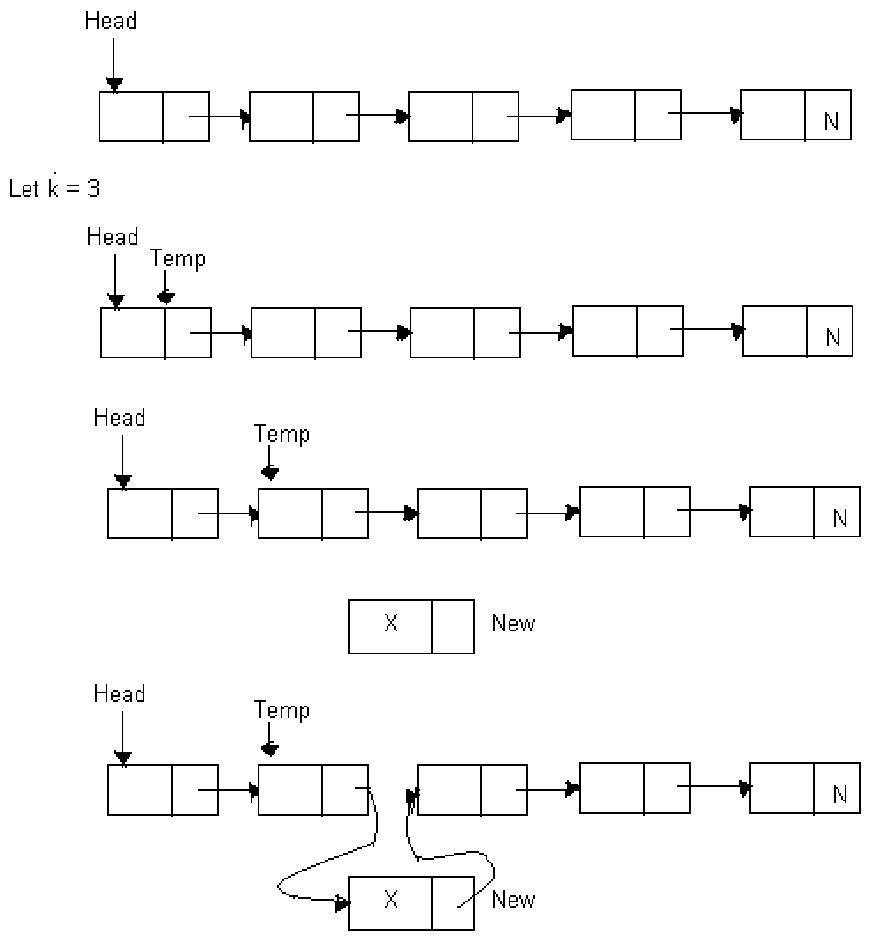


Fig 2.7 Inserting node at position

Algorithm

```
Function  insertin_mid()
Begin
Write    "Enter the position:"
Read pos;
If  head==NULL AND pos=1
then
Insert the node at the beginning
End if
If  head->next==NULL AND pos=2
then
insert the node
End if
Else if  pos>=2 AND pos<=ct
```

```
            then
            prev=head
            for I = 2 to pos -  1 step +1
            do
    prev=prev->link
            END FOR
            next=prev->link
            temp=new node
    Write"Enter the data:"
            Read temp->data
            temp->link=next
            prev->link=temp
            ct=ct+1
    else
    Write  "Enter a valid position & try again"
    End if
    End
```

Deletion
In a single linked list, the deletion operation can be performed in three ways. They are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list
We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1 → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).

Step 11 - If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Algorithm

```
Function del()
Begin
 if(head= NULL)
then
              Write “Empty list"
else
              Write “Enter the position:"
              Read  pos
              if(pos==1)
              then
                      next=head->link
                        head=next
                       ct=ct-1
             else
              if((pos>=2)&&(pos<=ct))
              prev=head
                          for(int i=2;i<=pos-1;i++)
                          Do
                                          prev=prev->link
                      End FOR
                      temp=prev->link
                      next=temp->link
                      prev->link=next
                       ct=ct-1
       End if
End If
End
```

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node

Step 5 - Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Counting

```
count()
begin
          int co=0;
          do
                    co++;
                    c=c->link;
          while(c!=NULL);
                    return co;
 end
```

Searching

The operation Search(x), searches the given value x in the list. If found returns the node position where it is found. A temporary pointer Temp is created and made to point to the Head. Now info part of Temp is compared with the value x. If the value matches the node position number is returned otherwise Temp pointer is moved to the next node. This is repeated till the end of the list is reached or till the element to be searched is found.

Algorithm

```
Function search()
Begin
flag=0
if(head=NULL)
then
          Write  "Empty list"
else
           Write "Enter the element to be searched:"
            Read e
           cur=head
           FOR I = 1 to ct Step +1
           Do
                     if(cur->data= e)
                    then

                              pos=I
                              flag++
                              break

                    else

                             cur=cur->link
                    End if
          ENDFOR
           If (flag =1)
```

```
            then
                        Write "Element found in position:" pos

             else

                          Write ”Element not found"
             End if

 End Search
```

2.3 Merging

```
Node* MergeLists(Node* list1, Node* list2)
begin
   Node* mergedList;
   if(list1 == null && list2 ==null)
      return null;
   if(list1 == null)
      return list2;

   if(list2 == null){
      return list1;
   }
   if(list1.data < list2.data){//initialize mergedList pointer to list1 if list1's data is lesser
      mergedList = list1;
   }else{//initialize mergedList pointer to list2 if list2's data is lesser or equal
      mergedList = list2;
   }
   while(list1!=null && list2!=null){
      if(list1.data < list2.data){
         mergedList->next = list1;
         list1 = list1->next;
      }else{
         mergedList->next = list2;
         list2 = list2->next;
      }
   }
   if(list1 == null){//remaining nodes of list2 appended to mergedList when list1 has
reached its end.
      mergedList->next = list2;
   }else{//remaining nodes of list1 appended to mergedList when list2 has reached its end
      mergedList->next = list1;
   }
   return mergedList;
}
```

Traversing

The operation Display( ), displays all the value in each node of the list. A temporary pointer is created and made to point to Head initially. Now info part of Temp is printed and Temp is moved to the next node. This is repeated till the end of the list is reached.

Algorithm

```
Function display()
Begin
  cur=head
  cout<<"\nNo.of nodes is:"<<ct
    cout<<"\nThe data is:"
  while (cur< >NULL)
  Do
     Write "["<<cur->data<<"]->"
     cur=cur->link
  End while
End
```

2.4 Reverse the Linked List

Algorithm

```
reverse()
{
        struct node *p1,*p2,*p3;
        if(start->link==NULL)
                return;
        p1=start;
        p2=p1->link;
        p3=p2->link;
        p1->link=NULL;
        p2->link=p1;
        while(p3!=NULL)
        {
                p1=p2;
                p2=p3;
                p3=p3->link;
                p2->link=p1;
        }
        start=p2;
  }
```

2.5 Polynomial Manipulation using List

$5x^2 + 3x^1 + 1$

$12x^3 - 4x^1$

$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials 1 + x and x + 1 are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials 5x4 – 8x3 + 2x2 + 4x1 + 9x0 illustrates in fig 2.8

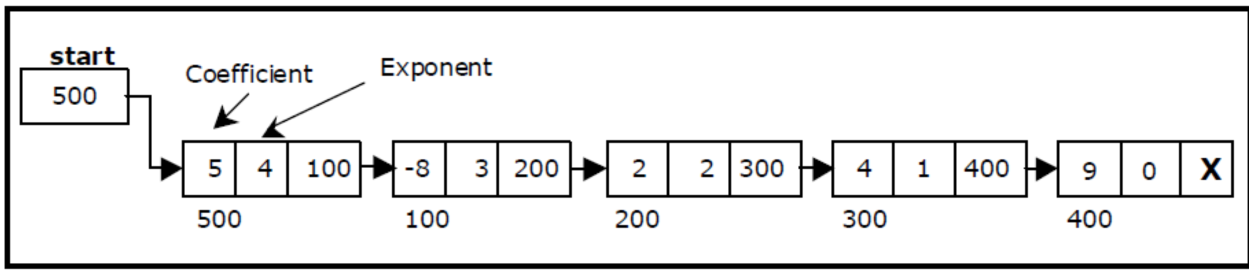


Fig2.8 Polynomial Manipulation

Algorithm

```
node * getnode()
{
node *tmp;
tmp =(node *) malloc( sizeof(node) );
printf("\n Enter Coefficient : ");
fflush(stdin);
scanf("%f",&tmp->coef);
printf("\n Enter Exponent : ");
fflush(stdin);
scanf("%d",&tmp->expo);
tmp->next = NULL;
return tmp;
}
node * create_poly (node *p )
{
char ch;
node *temp,*newnode;
while( 1 )
{
printf ("\n Do U Want polynomial node (y/n): ");
ch = getche();
if(ch == 'n')
break;
newnode = getnode();
if( p == NULL )
p= newnode;
else
{
temp = p;
while(temp->next != NULL )
temp = temp->next;
temp->next = newnode;
}
}
return p;
}
void display (node *p)
{
node *t = p;
while (t != NULL)
{
printf("+ %.2f", t -> coef);
printf("X^ %d", t -> expo);
t=t -> next;
}
}
```

2.6 Advantages and disadvantages of singly linked list

Advantages of Singly Linked List

- it is very easier for the accessibility of a node in the forward direction.
- the insertion and deletion of a node are very easy.
- the Requirement will less memory when compared to doubly, circular or doubly circular linked list.
- the Singly linked list is the very easy data structure to implement.
- During the execution, we can allocate or deallocate memory easily.
- Insertion and deletion of elements don’t need the movement of all the elements when compared to an array.

Disadvantages of Singly Linked List

- therefore, Accessing the preceding node of a current node is not possible as there is no backward traversal.
- the Accessing of a node is very time-consuming.

2.7 Circular linked list

Circular Linked List: Circular linked list is a linked list which consists of collection of nodes each of which has two parts, namely the data part and the link part. The data part holds the value of the element and the link part has the address of the next node. The last node of list has the link pointing to the first node thus making the circular traversal possible in the list is shown in Fig 2.9

Logical representation of the circular linked list:

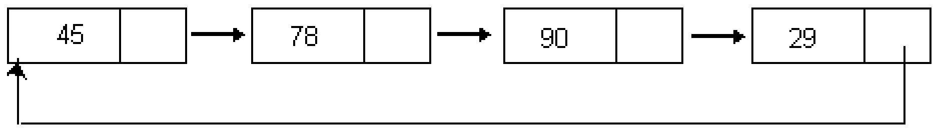


Fig 2.9 Circular Linked List

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;

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.

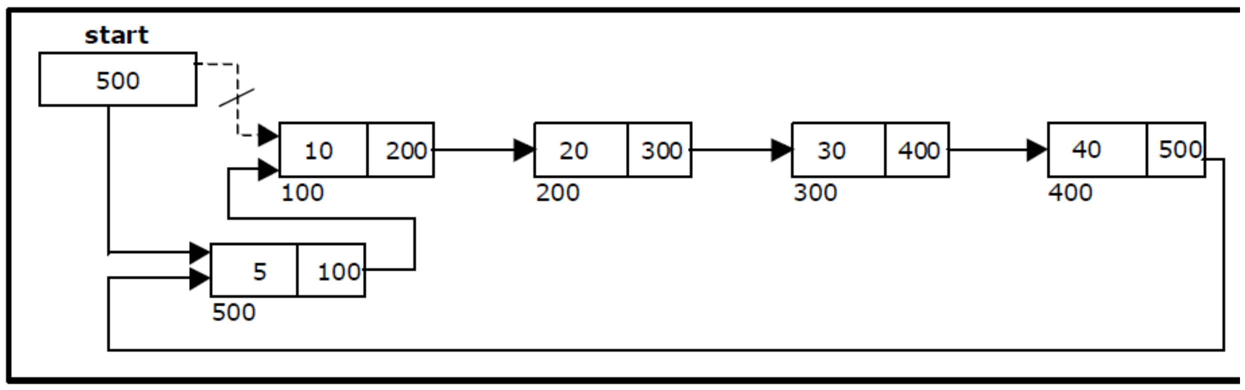


Fig 2.10 Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the 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:
temp = start;
while(temp -> next != start)
temp = temp -> next;
temp -> next = newnode;
newnode -> next = start;
The function cll_insert_end(), is used for inserting a node at the end.

Fig 2.11 shows inserting a node into the circular single linked list at the end.

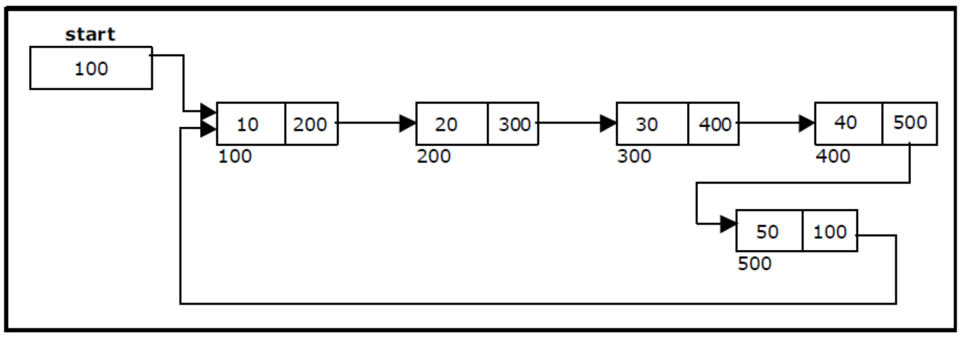


Fig.2.11 Inserting node at the end

Deleting a node at the beginning:

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

• If the list is empty, display a message ‘Empty List’.

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

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

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

The function cll_delete_beg(), is used for deleting the first node in the list. Fig 2.12 shows deleting a node at the beginning of a circular single linked list.

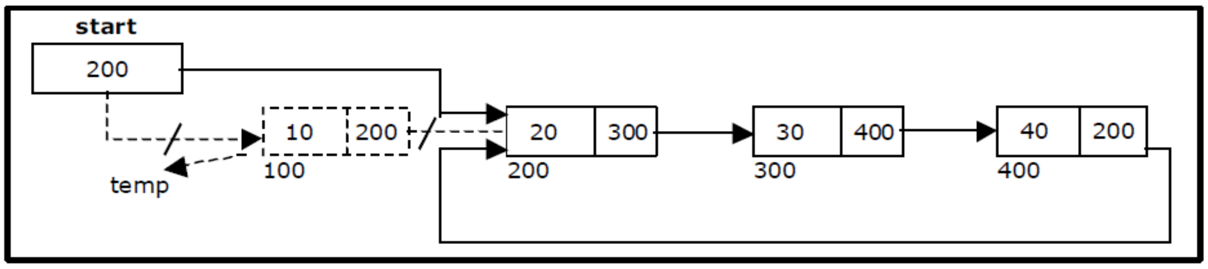


Fig 2.12 Deleting a node at the beginning

Deleting a node at the end:

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

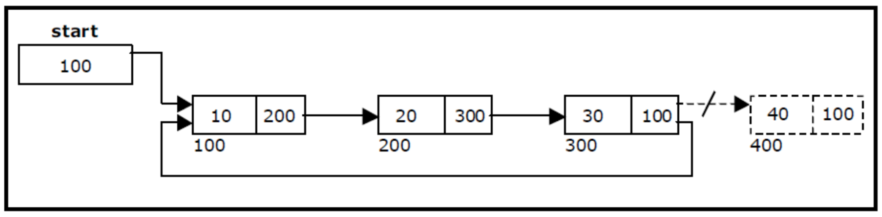
• If the list is empty, display a message ‘Empty List’.

• 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.Fig 2.13 shows deleting a node at the end of a circular single linked list.



2.13 Deleting the node at the end

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

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

```
temp = start;
do
{
printf("%d", temp -> data);
temp = temp -> next;
} while(temp != start);
```

2.7 Doubly linked list

Doubly linked list: The Doubly linked list is a collection of nodes each of which consists of three parts namely the data part, prev pointer and the next pointer. The data part stores the value of the element, the prev pointer has the address of the previous node and the next pointer has the value of the next node.

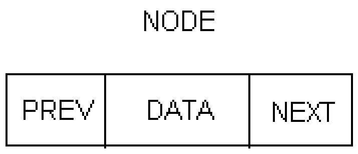


Fig 2.14 Doubly Linked List

In a doubly linked list, the head always points to the first node. The prev pointer of the first node points to NULL and the next pointer of the last node points to NULL shown in Fig.2.14

Operations on a Doubly linked list are,

- Count the number of elements.
- Add an element at the beginning of the list.
- Add an element at the end of the list.
- Insert an element at the specified position in the list.
- Delete an element from the list.
- Display all the elements of the list.

Addatbeg(x)

The operation Addatbeg(x) adds a given element x at the beginning of the list. A new node R is created and the value x is store in the data part of R. The prev pointer of R is made to point to NULL and the next pointer is made to point to head. Then the prev pointer of head is made to point to R and head is now moved to point to R making it the first node. Thus the new node is added at the beginning of the doubly linked list.

Algorithm

```
Function create()
Begin
   temp=new node
   Write"Enter the data:"
    Read temp->data
End

Function insert_begin()
Begin
 Call  create()
tmp->flink=head
head=tmp
head->blink=NULL
ct=ct+1
End
```

Addatend(x)

The Addatend(x) operation adds the given element x at the end of the doubly linked list. If the given list is empty then create a new node R and store the value of x in the data part of R. Now make the prev pointer and the next pointer of R point to NULL. Then head is pointed to R. If the list already contains some elements then, a temporary pointer is created and made to point to head. The temp pointer is now moved to the last node and then a new node R is created. The value x is stored in the data part of R and next pointer of R is made to point to NULL. The prev pointer of R is made to point to temp and next pointer of Temp is made to point to R. Thus the new node is added at the end of the doubly linked list is shown in fig.2.15.

Algorithm

```
Function   append()
Begin
 if(head=NULL)
 then

            insert_begin()
else
            create()
            temp->flink=NULL
            prev=head
           while(prev->flink< >NULL)
           Do
                  prev=prev->flink
           End while
           prev->flink=temp
           temp->blink=prev
            ct=ct+1
End if
End
```

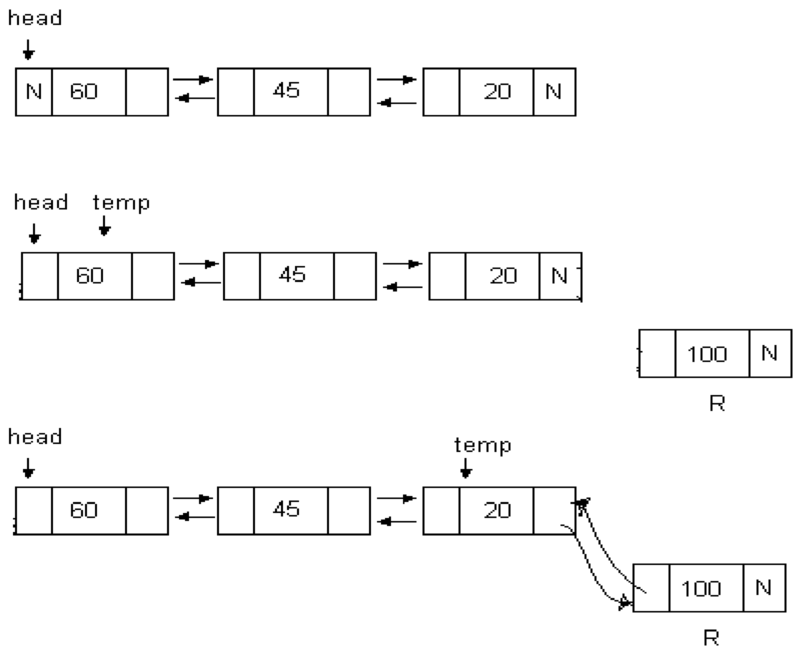


Fig. 2.15 Add at end

Insert(x, k)

The Insert(x) operation inserts a given element x at the specified position k. A temporary pointer is created and made to point to head. Then it is moved to the k-1th node. Now a new node R is created and the data part is stored with value of x. The next pointer of R is made to point to next(temp) and the prev pointer of next(temp) is made to point to R. Thus the links on the right side of the new node is established. Now the next of Temp is made to point to R and the prev pointer of R is made to point to temp thus establishing the links on the left side of the new node. Now the new node is inserted at the position k is shown in Fig.2.16.

Algorithm

```
Function  insertin_mid()
Begin
```

```
  Write    "Enter the position:"
  Read pos;
  If  head->flink=NULL AND pos=1
  then
         Call   insert_begin()
  End if
 If  head->flink==NULL AND pos=2
 then
  Call  append()
End if
Else if  pos>=2 AND pos<=ct
           then
            prev=head
            for I = 2 to pos -  1 step +1
           do
                prev=prev->flink
            END FOR
            next=prev->link
            temp=new node
            Write"Enter the data:"
            Read temp->data
            temp->flink=next
            prev->flink=temp
            temp->blink=prev;
            next->blink=temp;
            ct=ct+1
 else
           Write  "Enter a valid position & try again"
End if
End
```

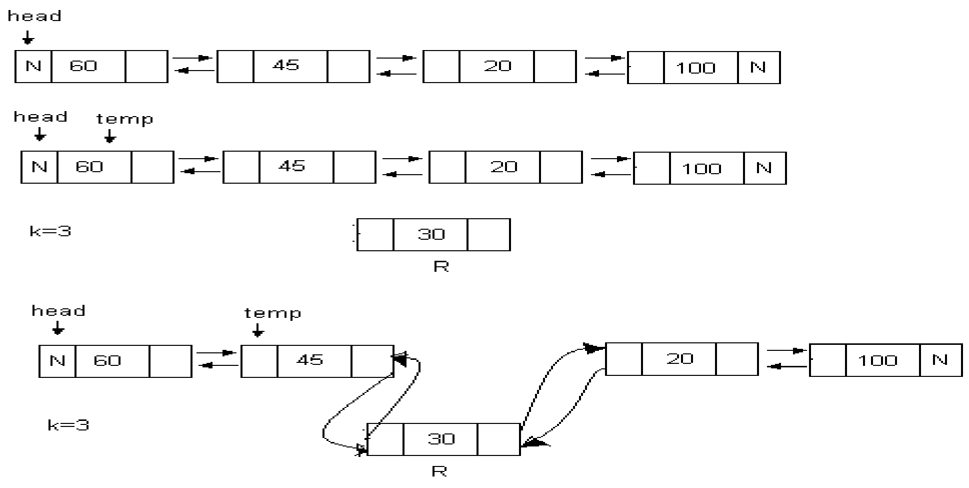


Fig.2.16 Insert at mid

Delete(x)

The Delete(x) operation deletes the element x from the doubly linked list. A temporary pointer is created and made to point to head. Now the data of temp is compared with x and if it matches that is the node to be deleted otherwise move to the next node and again compare. If the node to be deleted is first node, then prev(next(temp)) is made to point to NULL and head is pointed to next(temp). The node pointed by temp is deleted. When the node to be deleted is not the first node, then next(prev(temp)) is made to point to next(temp) and prev(next(temp)) is made to point to prev(temp). The node pointed by temp is deleted is shown in Fig.2.17.

Algorithm

```
Function del()
Begin
 if(head= NULL)
then
        cout<<"Empty list"
else
  Write  Enter the position\n"
        Read   pos
           pre=head
           if(pos<1 OR pos>ct)
                       Write  Enter a valid position"
           else
                 if(pos==1)
                 then
                         pre=pre->flink
                         head=pre
                          Write  "node gets deleted\n"
                         Ct=ct -1
                 else
                         for(i=2;i<pos;i++)
                                 pre=pre->flink
                         End For
                          tmp=pre->flink
                          nxt=tmp->flink
                          pre->flink=nxt
                          nxt->blink=pre
                          Write"node gets deleted\n"
                         Ct=ct -1
          End if
End if
```

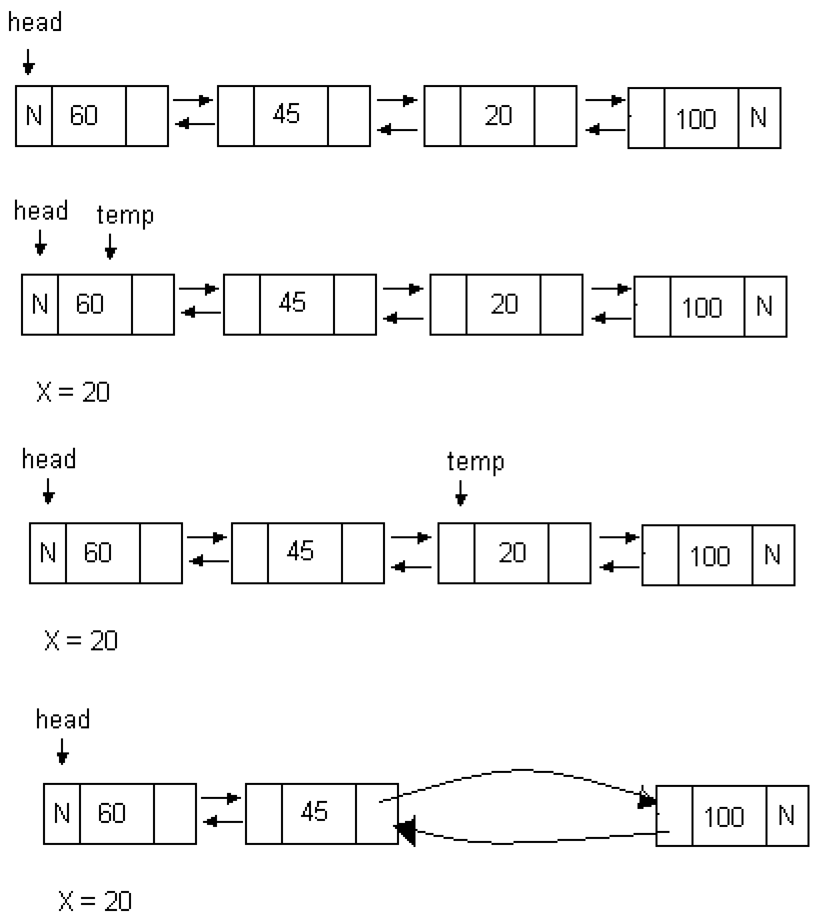


Fig.2.17 Deletion

2.8 Circular Doubly Linked List.

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in fig 2.18.

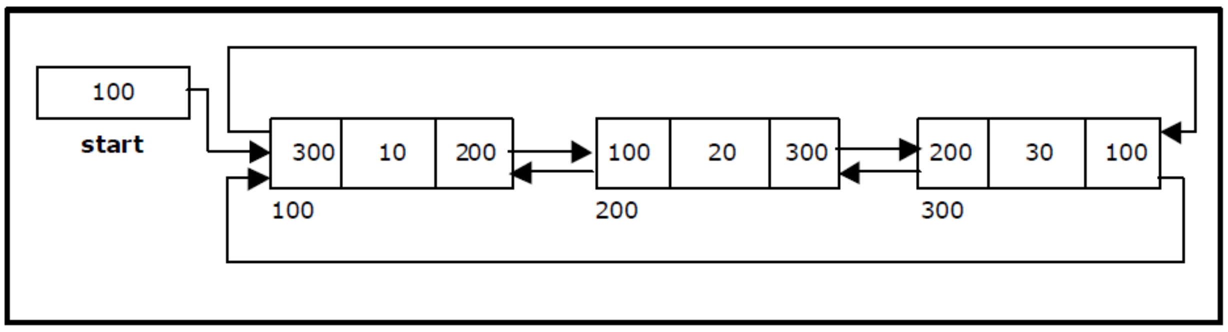


Fig 2.18 Circular Doubly Linked List

The basic operations in a circular double linked list are:

• Creation.

• Insertion.

• Deletion.
• Traversing.

Create node

```
create_node(int info)
begin
    new->val = info;
    new->next = NULL;
    new->prev = NULL;
    return new;
end
```

Add Node at the end

```
Procedure  add_node()
begin
    Read info
     if (first == last && first == NULL) // If list is empty
     begin
         first = last = new;
        first->next = last->next = NULL;
        first->prev = last->prev = NULL;
    endif
    Else // add the new node at the end
    begin
        last->next = new;
        new->prev = last;
        last = new;
        last->next = first;
        first->prev = last;
    end
end
```

INSERTS ELEMENT AT FIRST

```
insert_at_first()
begin
   Read info
   new = create_node(info); // create the new node
   if (first == last && first == NULL) // if the list is empty
    begin
        first = last = new;
        first->next = last->next = NULL;
        first->prev = last->prev = NULL;
    end
    else
    begin
```

```
        new->next = first;
        first->prev = new;
        first = new;
        first->prev = last;
        last->next = first;
    end
end
```

INSERTS ELEMNET AT END

```
insert_at_end()
begin
        Read info;
        new = create_node(info);
        if (first == last && first == NULL)
        begin
            first = last = new;
            first->next = last->next = NULL;
            first->prev = last->prev = NULL;
        endif
        else
        begin
            last->next = new;
            new->prev = last;
            last = new;
            first->prev = last;
            last->next = first;
        endif
end
```

INSERTS THE ELEMENT AT GIVEN POSITION

```
    insert_at_position()
    begin
        Declare info, pos, len = 0, i;
        Node *prevnode;
        Read info and pos
        new = create_node(info);
        if (first == last && first == NULL)
        begin
            if (pos == 1)
            begin
                first = last = new;
                first->next = last->next = NULL;
                first->prev = last->prev = NULL;
            endif
            else
            printf " empty linked list you cant insert at that particular position"
        endif
        else
        begin
            if (number < pos) // total number of node is stored in the variable number
                print " node cant be inserted as position is exceeding the linkedlist length"
             else

                for (ptr = first, i = 1;i <= number;i++)
                begin
                    prevnode = ptr;
                    ptr = ptr->next;
                    if (i == pos-1)
                    begin
                        prevnode->next = new;
                        new->prev = prevnode;
                        new->next = ptr;
                        ptr->prev = new;
                        print "inserted at position is succesfully"
                        break;
                    end
    end
    end
    end
    end
```

Deletion

```
    delete_node_position()
    begin
        int pos, count = 0, i;
        n *temp, *prevnode;
    read  the position which u wanted to delete

        if (first == last && first == NULL)
            print " empty linked list you cant delete"
         else
        begin
            if (number < pos)
                print " node cant be deleted at position as it is exceeding the linkedlist length"
             else
            begin
                for (ptr = first,i = 1;i <= number;i++)
                begin
                    prevnode = ptr;
                    ptr = ptr->next;
                    if (pos == 1)
                    begin
                    number--;
                    last->next = prevnode->next;
                    ptr->prev = prevnode->prev;
                    first = ptr;
                    printf("%d is deleted", prevnode->val);
                    free(prevnode);
                    break;
                end
                else if (i == pos - 1)
                begin
                    number--;
                    prevnode->next = ptr->next;
                    ptr->next->prev = prevnode;
                    printf("%d is deleted", ptr->val);
                    free(ptr);
                    break;
                end
            end
        end
    end
end
```

Searching

```
search()
begin
    int count = 0, key, i, f = 0;
     read the value to be searched in the variable key
     if (first == last && first == NULL)
        print "list is empty no elemnets in list to search"
    else
        for (ptr = first,i = 0;i < number;i++,ptr = ptr->next)
        begin
            count++;
            if (ptr->val == key)
            begin
                Print " the value is found at position  count ”
                f = 1;
            end
           end
        if (f == 0)
            print "the value is not found in linkedlist"
    end
end
```

DISPLAYING IN BEGINNING

Algorithm

```
display_from_beg()
begin
    int i;
    if (first == last && first == NULL)
        print "list is empty no elemnts to print"
    else
    begin
        Store total number of node in the variable , number
        for (ptr = first, i = 0;i < number;i++,ptr = ptr->next)
            print  ptr->val
    end
end
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERIN

UNIT – III-Data Structures – SCSA1205

Unit:III

Introduction – Array Representation of a Stack – Linked List Representation of a Stack - Stack Operations - Algorithm for Stack Operations - Stack Applications: Tower of Hanoi - Infix to postfix Transformation - Evaluating Arithmetic Expressions. Queue – Introduction – Array Representation of Queue – Linked List Representation of Queue - Queue Operations - Algorithm for Queue Operations - . Queue Applications: Priority Queue.

3.1 Introduction

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following fig 3.1 depicts a stack and its operations −

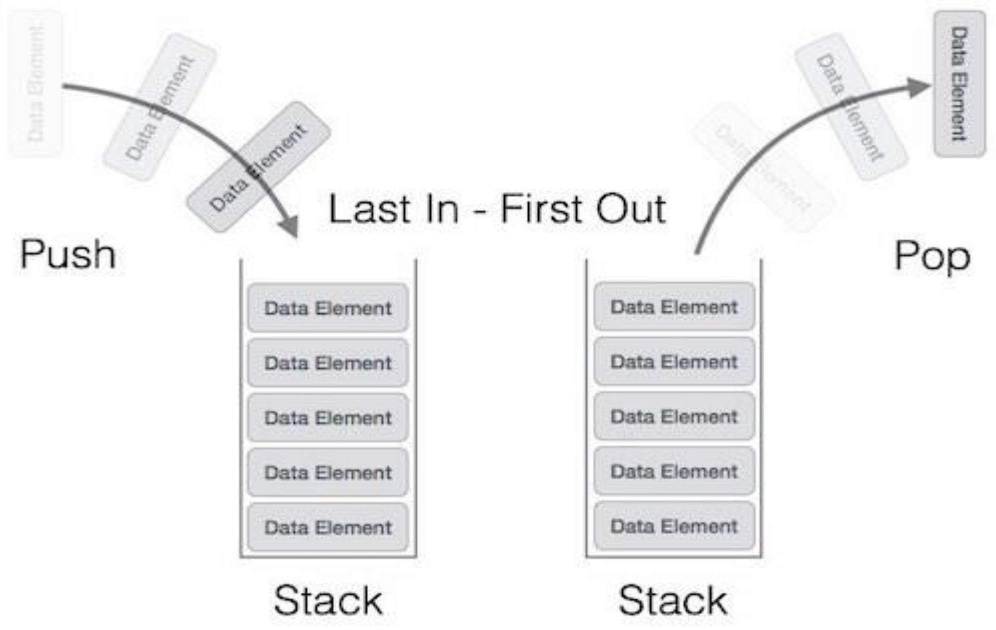


Fig 3.1 Stack Representation

3.2 Array representation of a Stack

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation. First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.

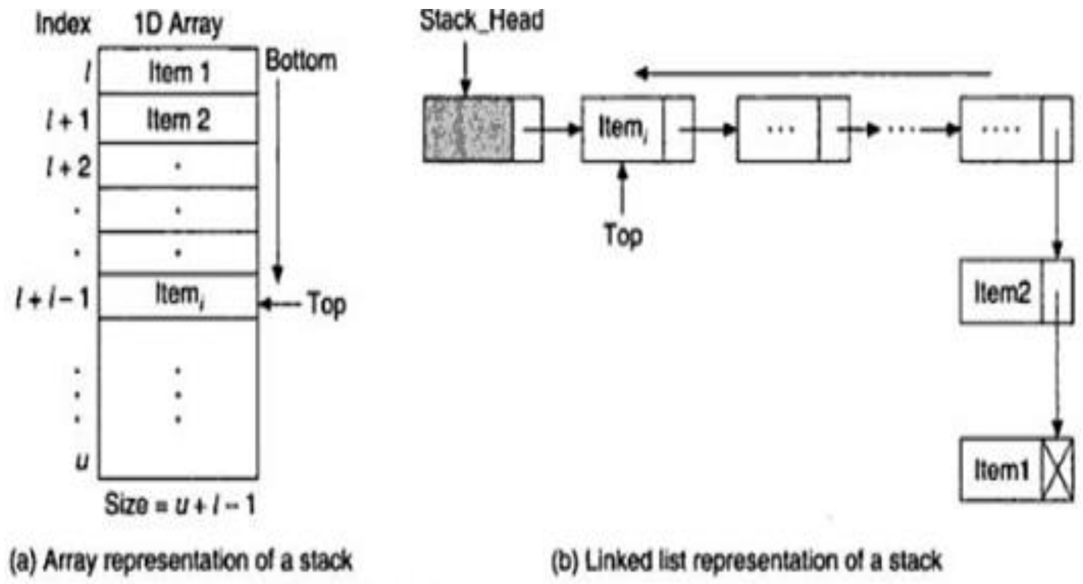


Fig 3.2 Representation of a stack

In Fig 3.2 a, Itemi denotes the *ith* item in the stack; l and *u* denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack.

3.3 Linked List Representation of Stacks

Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications, the size of the stack may vary during program execution. An obvious solution to this problem is to represent a stack using a linked list. A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usual, to point to the next' item. Above Figure b depicts such a stack using a single linked list. In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item. Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list is shown in Fig 3.2 b.

3.4 Stack Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

3.5 Algorithm for Stack Operations

Push Operation

The process of putting a new data element onto stack is known as a Push Operation is shown in fig.3.3. Push operation involves a series of steps −

Step 1 − Checks if the stack is full.

Step 2 − If the stack is full, produces an error and exit.

Step 3 − If the stack is not full, increments **top** to point next empty space.

Step 4 − Adds data element to the stack location, where top is pointing.

Step 5 − Returns success.

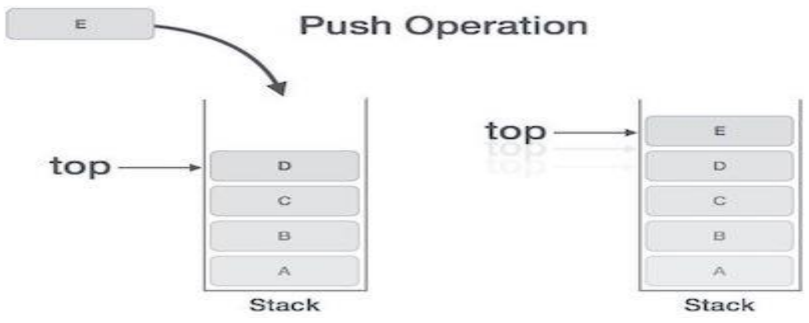


Fig 3.3 Push Operation

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
procedure push

   if stack is full
      return null
   endif
   top ← top + 1
   stack[top] ← data
end procedure
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space is shown in Fig.3.4.

A Pop operation may involve the following steps −

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which **top** is pointing.

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

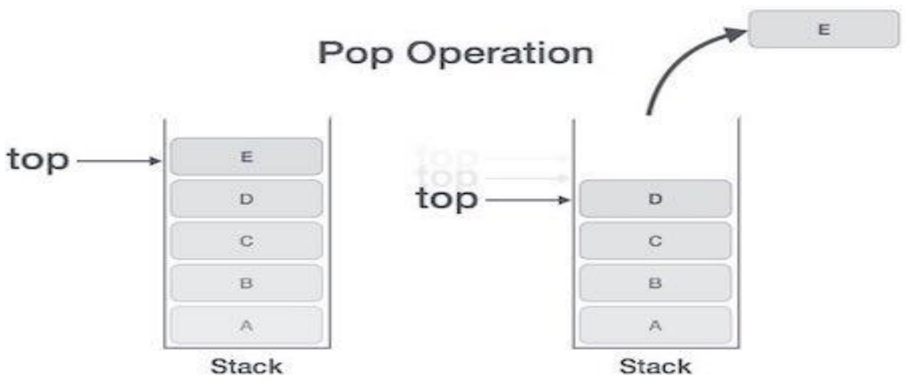


Fig 3.4 Pop Operation

Algorithm for Pop Operation

```
procedure pop: stack

   if stack is empty
      return null
   endif
   data ← stack[top]
   top ← top - 1
   return data
```

Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).

Step 3 - Repeat above step until **i** value becomes '0'.

3.6 Stack Using Linked List

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example

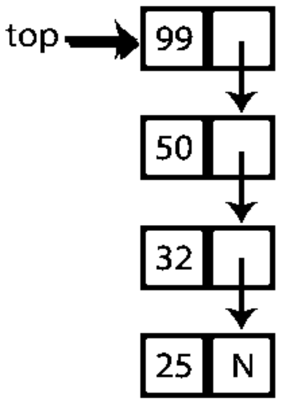


Fig 3.5 Linked List Representation

In the above Fig 3.5, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether stack is **Empty** (**top** == **NULL**)

Step 3 - If it is **Empty**, then set **newNode → next** = **NULL**.

Step 4 - If it is **Not Empty**, then set **newNode → next** = **top**.

Step 5 - Finally, set **top** = **newNode**.

Algorithm

```
push(int value)
begin
  newNode->data = value;
  if(top == NULL)
    newNode->next = NULL;
  else
    newNode->next = top;
  top = newNode;
 end
```

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1 - Check whether **stack** is **Empty** (**top == NULL**).

Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4 - Then set '**top = top → next**'.

Step 5 - Finally, delete '**temp**'. (**free(temp)**).

Algorithm

```
void pop()
 begin
   if(top == NULL)
      print("\nStack is Empty!!!\n");
   else
      print("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   endif
end
```

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1 - Check whether stack is **Empty** (**top** == **NULL**).

Step 2 - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

Step 4 - Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).

Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

Algorithm

```
void display()
begin
  if(top == NULL)
    print("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    while(temp->next != NULL)
     begin
          print("%d--->",temp->data);
          temp = temp -> next;
    end
    print("%d--->NULL",temp->data);
  end
```

3.7 Stack Applications

3.7.1Tower of Hanoi

Tower of Hanoi is a mathematical puzzle which consists of three tower (pegs) and more than one ring; as depicted in Fig.3.6.

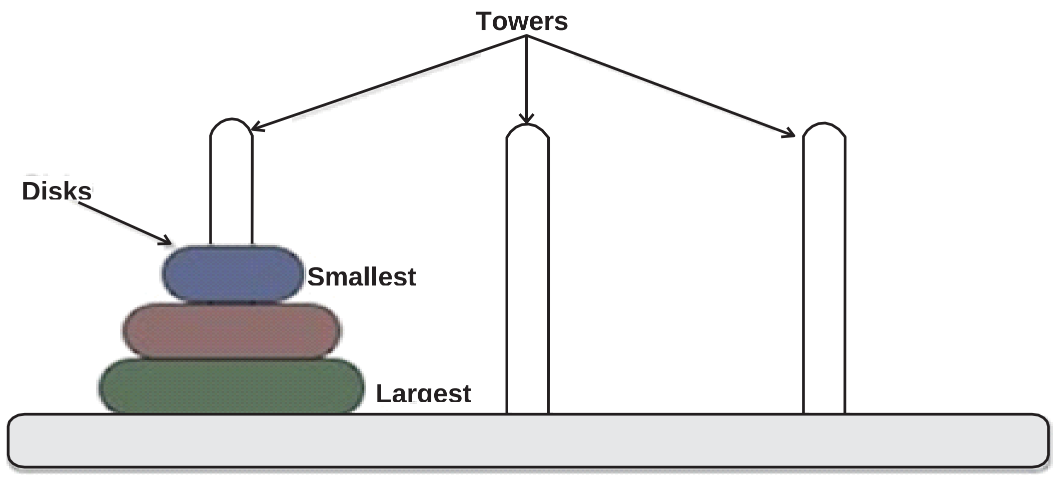


Fig.3.6 Tower of Hanoi

These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variationsof puzzle where the number of disks increases, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for Tower of Hanoi

- Only one disk can be moved among the towers at any given time.
- Only the “top” disk can be removed.
- No large disk can sit over a small disk.

Steps for solving the Towers of Hanoi problem

The following steps are to be followed.

Step 1: Move n-1 disks from source to aux.

Step 2: Move nth disk from source to destination

Step 3: Move n-1 disks from aux to destination.

Fig.3.7, illustrates the step by step movement of the disks to implement Tower of Hanoi.

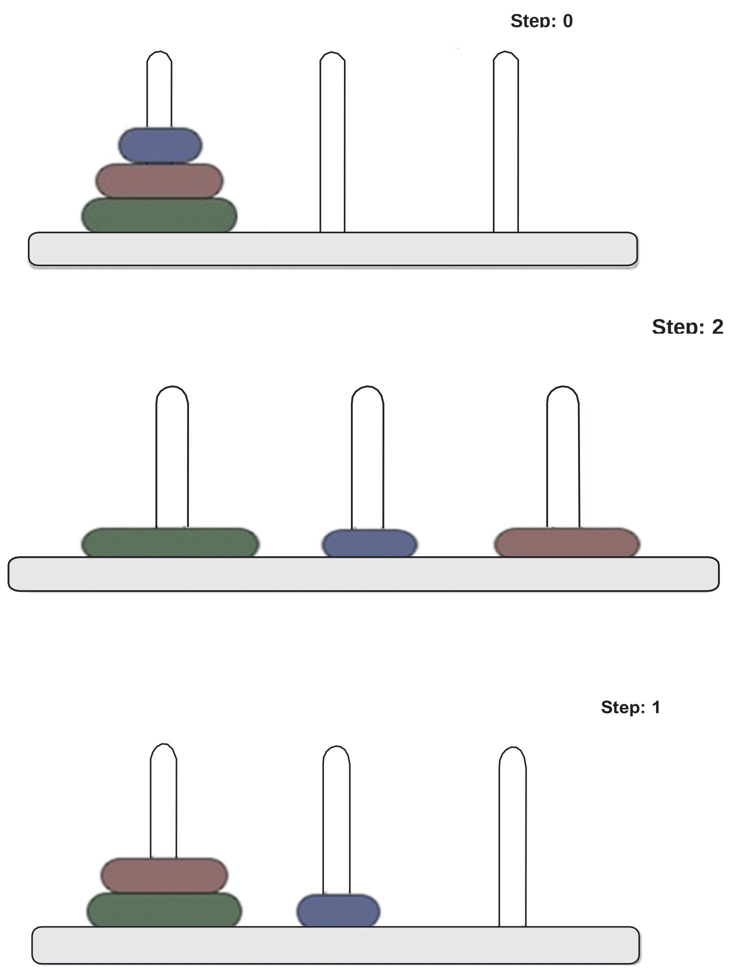


Fig 3.7 Tower of Hanoi

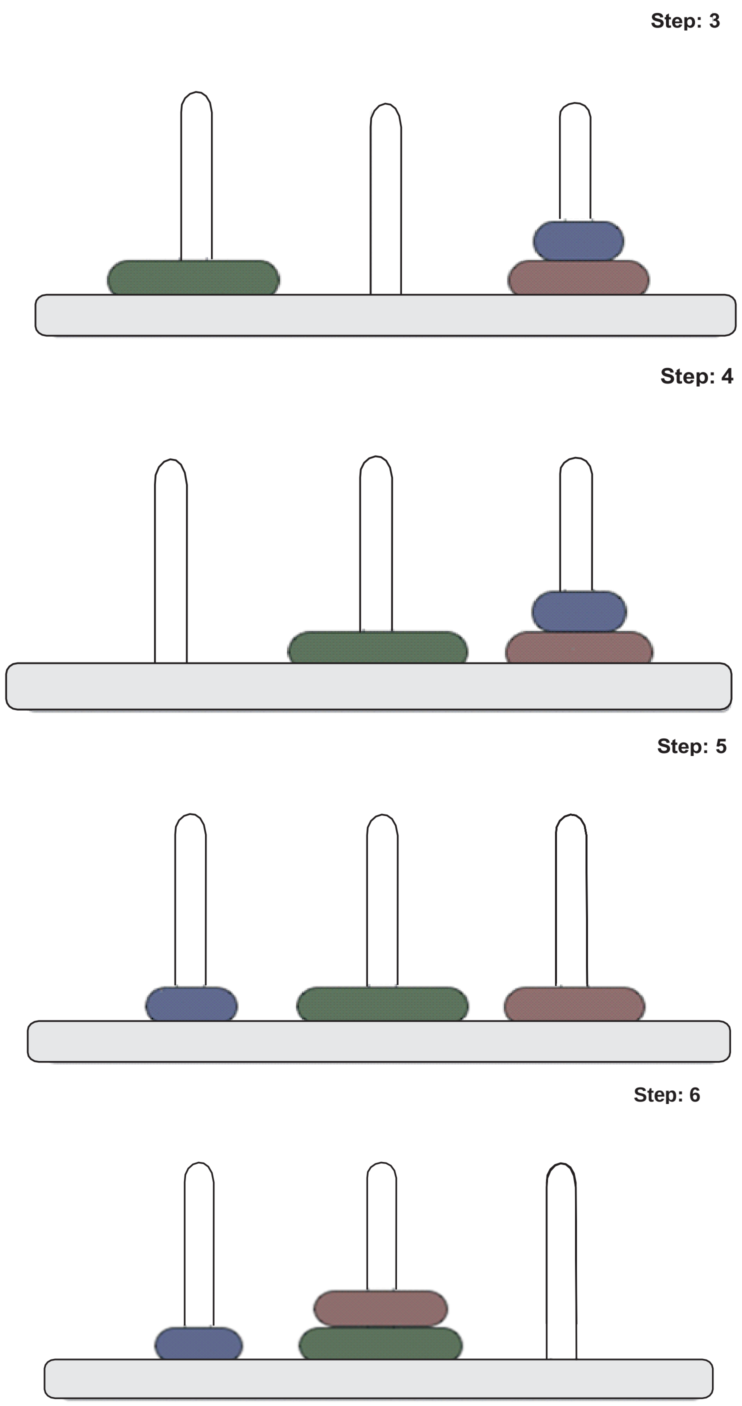


Fig 3.7 Tower of Hanoi

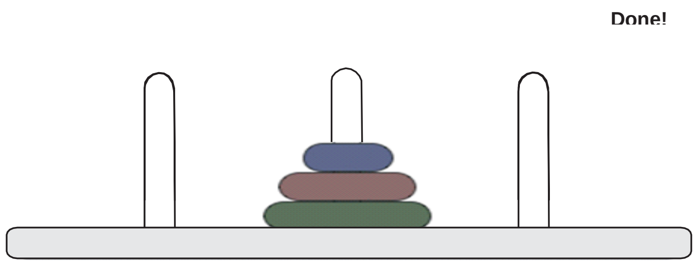


Fig.3.7 Tower of Hanoi

A recursive algorithm for Tower of Hanoi can be driven as followsSTART

```
Procedure Hanoi (disk, source, dest, aux)
IF disk = 0, THEN
Move disk from source to destELSE
Hanoi (disk-1, source, aux, dest) //Step1Move disk from source to dest //Step2 Hanoi
(disk-1, aux, dest, source) //Step3 ENDIF

END
```

3.7.2 Infix to postfix Transformation

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its **operands** have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle **operators** and parentheses.

Example:

1. A * B + C becomes A B * C +

The order in which the operators appear is not reversed. When the '+' is read, it has lower **precedence** than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. A + B * C becomes A B C * +

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when the are both popped off in lines 6 and 7, their order will be reversed.

3. A * (B + C) becomes A B C + *

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A B
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. A - B + C becomes A B - C +

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B

4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. A * B ^ C + D becomes A B C ^ * D +

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. A * (B + C * D) + E becomes A B C D * + * E +

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Algorithm

```
int top = -1;
Infixtopostfix()
begin
    print("\n\nRead the Infix Expression ? ");
     Read infx
     push('#');
     while ((ch = infx[i++]) != '\0')
      begin
           if (ch == '(')
                push(ch);
          else if (isalnum(ch))
                 pofx[k++] = ch;
               else if (ch == ')')
                   begin
                   while (s[top] != '(')
                   begin
                           pofx[k++] = pop();
                           elem = pop(); /* Remove ( */
                   end
                   else
                    while (pr(s[top]) >= pr(ch))
                          pofx[k++] = pop();
                   push(ch);
          end
    end
```

3.7.3 Evaluating Arithmetic Expressions

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. **Polish notation (prefix notation) –**
   It refers to the notation in which the operator is placed before its two operands . Here no parentheses are required, i.e., +AB

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they are appear.
3. When any operator encounter then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression the final result remains on the top of the stack.

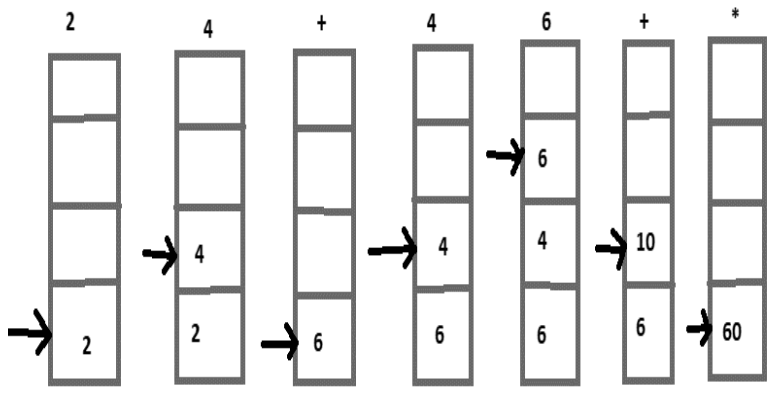
For example –

Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60

6. The stack operations for this expression evaluation is shown below:



Stack operations to evaluate (2+4)*(4+6)

Fig 3.8 Arithmetic Expressions

3.8 Queue

3.8.1 Introduction

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT** (also called **head**). This makes queue as **FIFO** (First in First Out) data structure, which means that element inserted first will be removed first. The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

3.8.2 Array Representation of Queue

If queue is implemented using arrays, the size of the array should be fixed maximum allowing the queue to expand or shrink.

Operations on a Queue

There are two common operations one in a queue. They are addition of an element to the queue and deletion of an element from the queue. Two variables front and rear are used to point to the ends of the queue. The front points to the front end of the queue where deletion takes place and rear points to the rear end of the queue, where the addition of elements takes place. Initially, when the queue is full, the front and rear is equal to -1.

Add(x)

An element can be added to the queue only at the rear end of the queue. Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the rear side.

ADDQ(x)

```
If rear = MAX – 1
Then
    Print “Queue is full”
    Return
Else
    Rear = rear + 1
    A[rear] = x
    If front = -1
    Then
        Front = 0
    End if
End if
End ADDQ( )
```

Del( )

The del( ) operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.

DELQ( )

```
If front = -1
Then
        Print “Queue is Empty”
        Return
Else
        Item = A[front]
        A[front] = 0
        If front = rear
        Then
                Front = rear = -1
        Else
                Front = front + 1
        End if
        Return item
End if
End DELQ( )
```

3.8.3 Linked List Representation of Queue

Queue can be represented using a linked list. Linked lists do not have any restrictions on the number of elements it can hold. Space for the elements in a linked list is allocated dynamically; hence it can grow as long as there is enough memory available for dynamic allocation. The queue represented using linked list would be represented as shown. The front pointer points to the front of the queue and rear pointer points to the rear of the queue is shown in Fig 3.9.

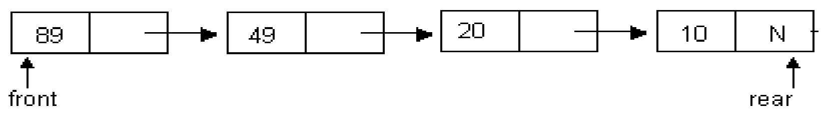


Fig 3.9 Linked List

Addq(x)

In linked list representation of queue, the addition of new element to the queue takes place at the rear end. It is the normal operation of adding a node at the end of a list.

ADDQ(x)

```
If front = NULL
Then
    Rear = front = temp
    Return
End if
Link(rear) = temp
Rear = link(rear)
End ADDQ( )
```

Delq( )

The delq( ) operation deletes the first element from the front end of the queue. Initially it is checked, if the queue is empty. If it is not empty, then return the value in the node pointed by front, and moves the front pointer to the next node.

DELQ( )

```
If front = NULL
    Print “Queue is empty”
    Return
Else
    While front ≠ NULL
        Temp = front
        Front = link(front)
        Delete temp
    End while
End if
End DELQ( )
```

3.8.5 Queue Applications

3.8.5.1 Priority Queue

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa is shown in Fig.3.10.

Basic Operations

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

Priority Queue Representation

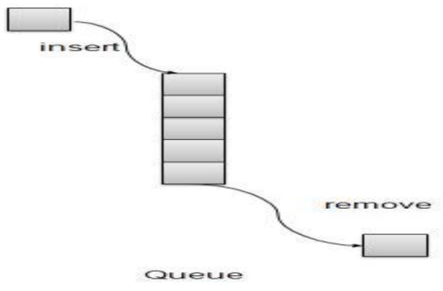


Fig 3.10 Priority Queue

There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority is shown in Fig 3.11.

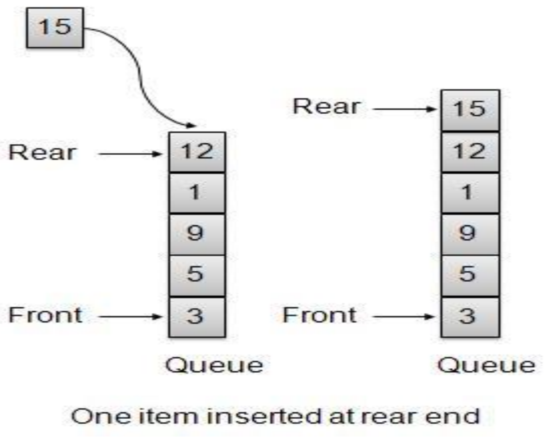


Fig 3.11 Insert/Enqueue operation

```
void insert(int data){
  int i = 0;
  if(!isFull()){
    // if queue is empty, insert the data
    if(itemCount == 0){
      intArray[itemCount++] = data;
    }else{
      // start from the right end of the queue
      for(i = itemCount - 1; i >= 0; i-- ){
        // if data is larger, shift existing item to right end
        if(data > intArray[i]){
          intArray[i+1] = intArray[i];
        }else{
          break;
        }
      }
      // insert the data
      intArray[i+1] = data;
      itemCount++;
    }
  }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.

```
int removeData(){
  return intArray[--itemCount];
}
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV-Data Structures – SCSA1205

UNIT IV

Preliminaries of Tree ADT - Binary Trees - The Search Tree ADT–Binary Search Trees - AVL Trees - Tree Traversals - B-Trees - Heap Tree – Preliminaries of Graph ADT - Representation of Graph – Graph Traversal - BFS – DFS – Applications of Graph – Shortest - Path Algorithms – Dijkstra's Algorithm Minimum Spanning Tree – Prims Algorithm

4.1 PRELIMINARIES OF TREE ADT

A tree is a collection of nodes (Fig 4.1). The collection can be empty or it consists of

a) a distinguished node r called the root and
b) zero or more nonempty subtree, each of whose roots are connected by a directed edge from r.
c) subtrees must not connect together. Every node in the tree is the root of some subtree.
d) There are N-1 edges in a tree with N nodes.
e) Normally the root is drawn at the top.

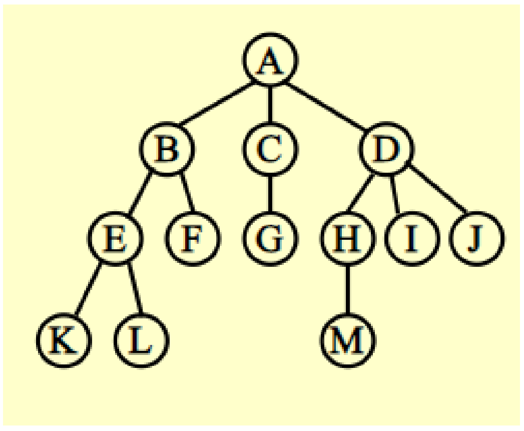


Fig 4.1 Tree Structure

Degree of a node = number of subtrees of the node

Degree (A) = 3, degree (F) = 0

Degree of tree = $\max\limits_{node \in tree} \{degree(node)\}$

Degree of tree =3

Parent = node that has subtrees

Children = the roots of the subtrees of parent

Siblings = children of the same parent

Leaf (terminal node) = a node with degree 0 (no children)

Path from n_1 to n_k = a unique sequence of nodes from $n_1, n_2, \dots, n_k$

Length of path = number of edges on the path

Depth of n = length of the unique path from root to n_i, Depth (root) = 0

Height of n = length of longest path from n_i to leaf. Height (D) =2

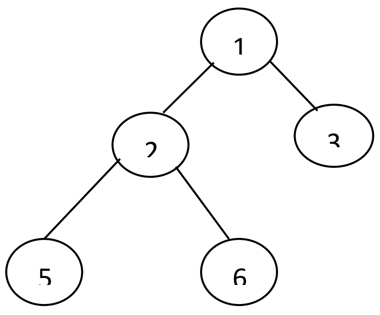
Height of tree =height(root)=depth(deepest leaf)

Ancestors of node= all nodes along the path from node upto the root

Descendants of node = all nodes in its subtrees

4.2 Binary Trees

A binary tree is a tree in which no node can have more than two children. Each node can have 0, 1 or 2 children is shown Fig 4.2.



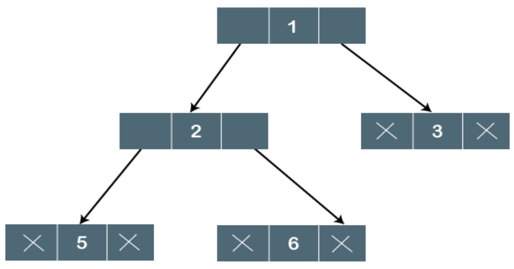


Fig 1.2 Binary tree and list representation

In this tree, node 1,2 and 3 contains two points i.e, left and right pointer pointing to the left and right node respectively. Similarly nodes 3, 5 and 6 are the leaf nodes so these nodes have NULL pointer on both left and right parts.

Properties of Binary Tree

a) At each level of i, the maximum number of nodes is 2^i.

b) The height of tree is defines as the longest path from the root node to the leaf node. Her example has height 3 and the maximum number of nodes at height 3 is (1+2+4+8)=15.

c) The minimum number of nodes possible at height h is equal to h+1.

d) If the number of nodes is minimum, then the height of the tree would be maximum. Similarly, if the number of nodes is maximum, then the height of the tree would be minimum.

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left shown in Fig.4.3.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is 2^{h+1}-1.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is $\log_2(n+1)$ -1.

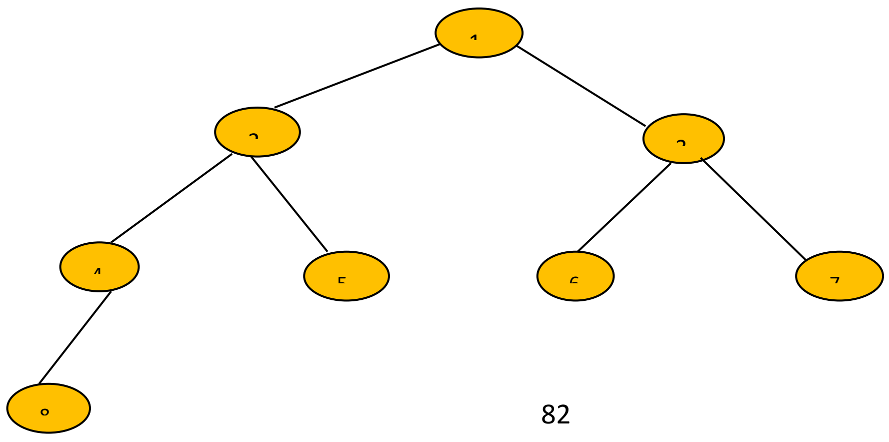


Fig 4.3 complete binary tree

Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level. All perfect binary trees are the complete binary binary trees but it's not true for vice versa, i.e., all complete binary trees cannot be the perfect binary trees is shown in Fig 4.4.

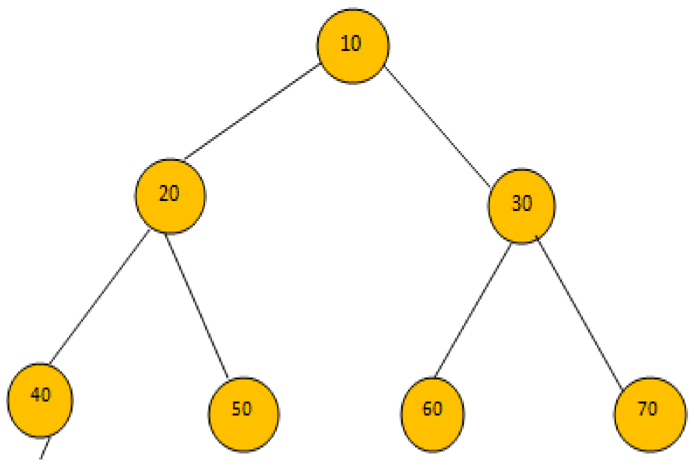


Fig 4.4 Perfect Binary tree

4.3 BINARY SEARCH TREE

Binary Search Tree is a node-based binary tree data structure which has the following properties

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- Left_subtree (keys) < node(key) ≤ right_subtree(keys) is shown in Fig 4.5.

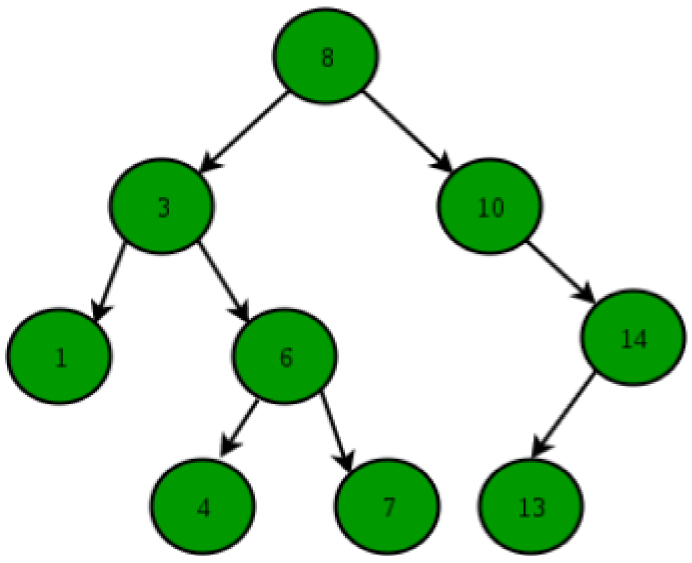


Fig 4.5Binary search tree

Basic Operations

1. Search
2. Insert
3. Pre-order Traversal
4. In-order Traversal
5. Post-order Traversal

1. **Search Operation**

Whenever an element is to be searched, start searching from the root node. If the data is less than the key value, then search for the element in the left subtree. Otherwise, search for the element in the right subtree. Fig 4.6 shows the example of search operation.

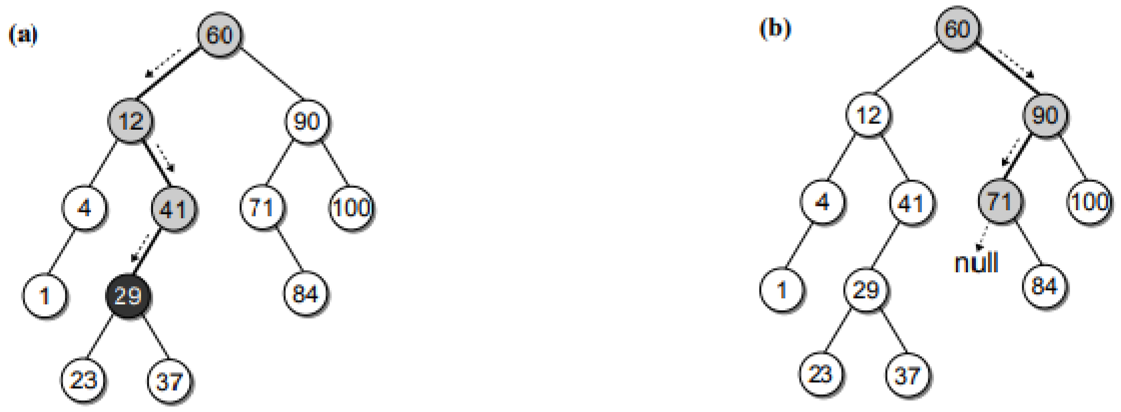


Fig 4.6 Search operation in Binary search tree

Min and Max Values

Another operation similar to a search that can be performed on a binary search tree is finding the minimum or maximum key values. From the definition of the binary search tree, we know the minimum value is either in the root or in a node to its left is shown in fig 4.7.

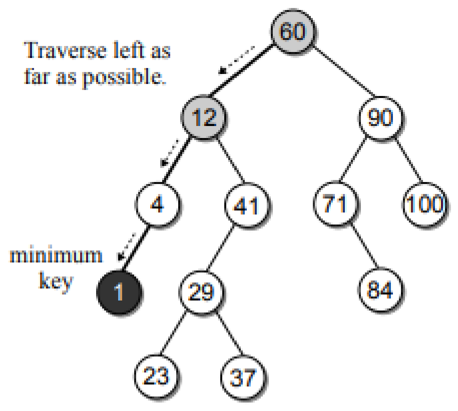


Fig 4.7 Find Min and max values

2. **Insertion Operation**

When a binary search tree is constructed, the keys are added one at a time. As the keys are inserted, a new node is created for each key and linked into its proper position within the tree. Suppose we want to build a binary search tree from the key list [60, 25, 100, 35, 17, 80] by inserting the keys in the order they are listed is shown in Fig 4.8.

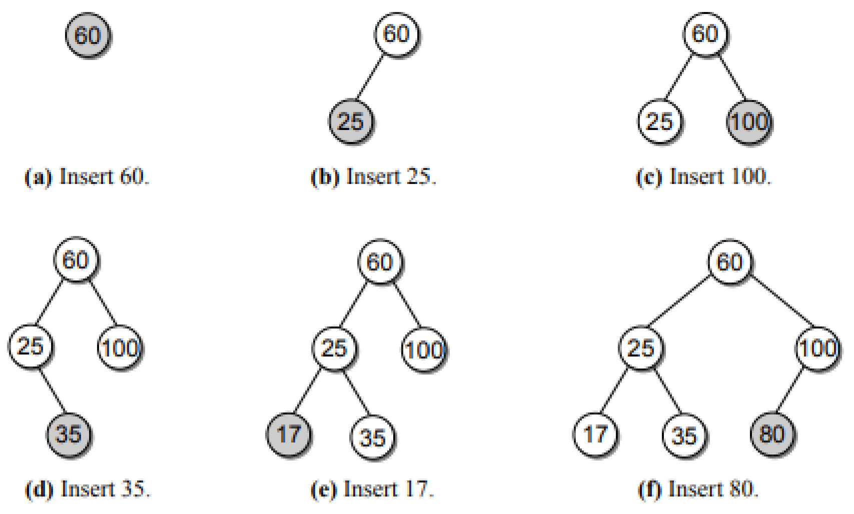


Fig 4.8 Insert operation in binary search tree

3. **Deletions**

To remove an element from a binary search tree is a bit more complicated than searching for an element or inserting a new element into the tree. A deletion involves searching for the node that contains the target key and then unlinking the node to remove it from the tree. When a node is removed, the remaining nodes must preserve the search tree property. There are three cases to consider once the node has been located:

1. The node is a leaf.
2. The node has a single child.
3. The node has two children.

Example:

✓ Removing leaf node from binary search tree

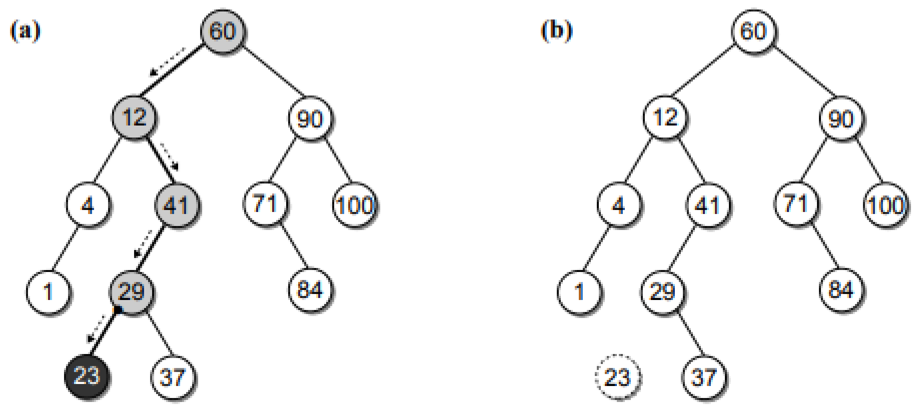
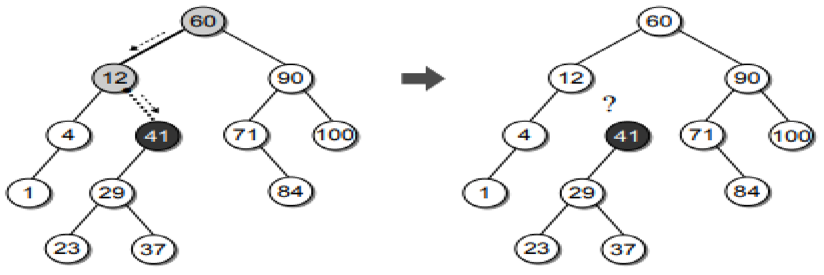
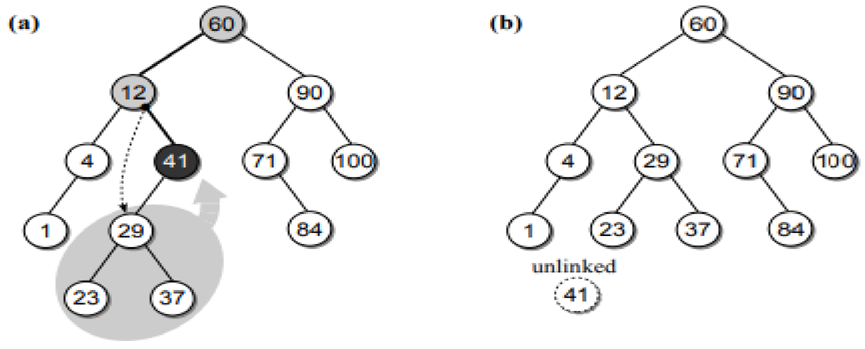


Fig 4.9 Removing leaf node

✓ Removing an interior node with one child shown in Fig 4.10.





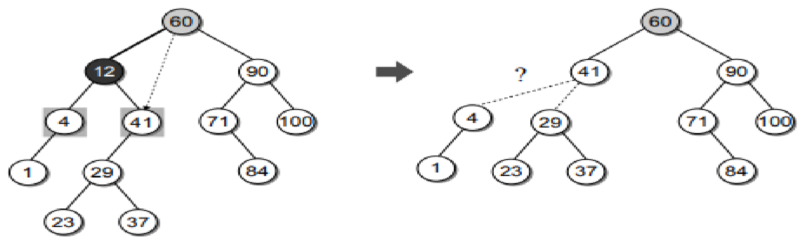


Fig 4.10 removing interior node with one child

✓ Removing an interior node with two children is shown in Fig 4.11.

For node 12, its predecessor is node 4 and its successor is node 23. For removing an interior node with two children requires three steps:

1. Find the logical successor, S, of the node to be deleted, N.
2. Copy the key from node S to node N.
3. Remove node S from the tree.

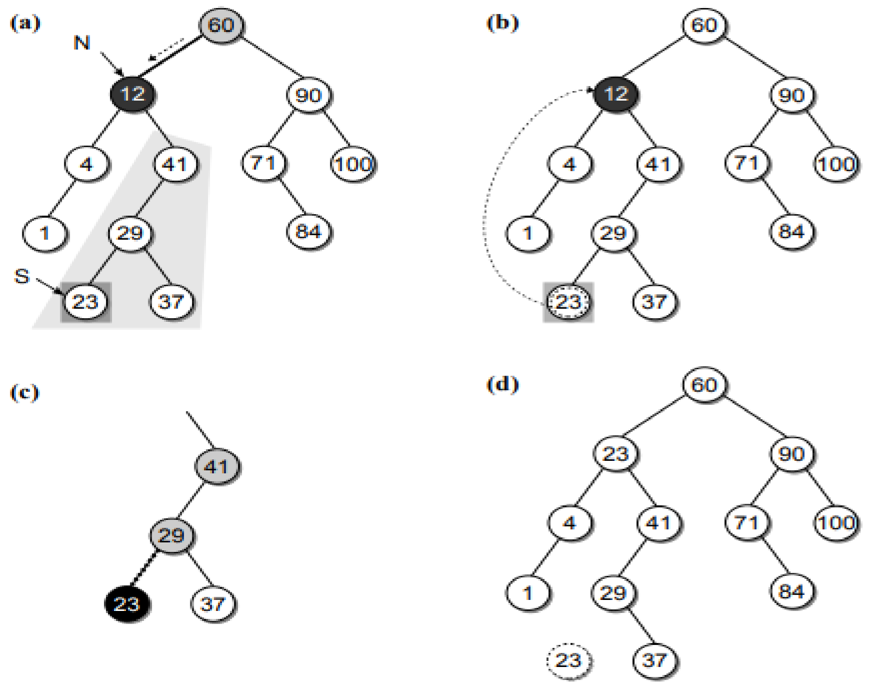


Fig 4.11 removing interior node with two children

4.4 AVL TREES

The AVL tree, which was invented by G. M. Adel'son-Velskii and Y. M. Landis in 1962, improves on the binary search tree by always guaranteeing the tree is height balanced, which allows for more efficient operations. A binary tree is balanced if the heights of the left and right subtrees of every node differ by at most is shown in Fig 4.12.

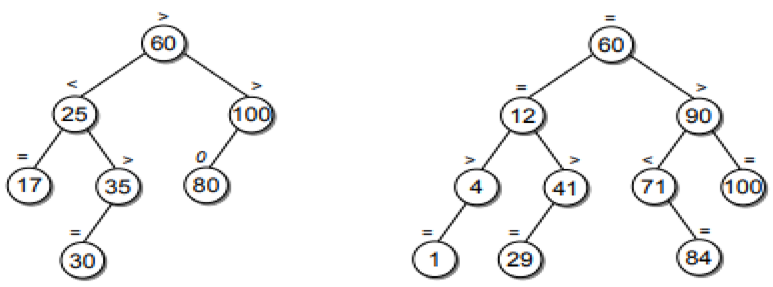


Fig 4.12 AVL tree model

With each node in an AVL tree, we associate a balance factor, which indicates the height difference between the left and right branch. The balance factor can be one of three states:

left high (>): When the left subtree is higher than the right subtree.

equal high (=): When the two subtrees have equal height.

right high (<): When the right subtree is higher than the left subtree

Insertions

Inserting a key into an AVL tree begins with the same process used with a binary search tree. We search for the new key in the tree and add a new node at the child link where we fall off the tree. When a new key is inserted into an AVL tree, the balance property of the tree must be maintained. If the insertion of the new key causes any of the subtrees to become unbalanced, they will have to be rebalanced is shown in Fig.4.13.

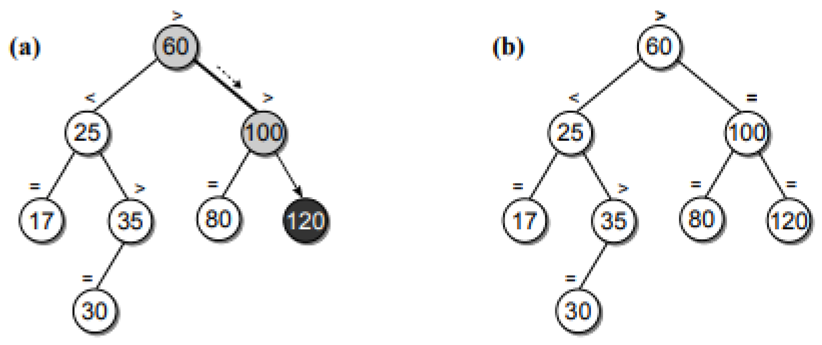


Fig.4.13 Insertion in AVL tree

Rotations

Multiple subtrees can become unbalanced after inserting a new key, all of which have roots along the insertion path. But only one will have to be rebalanced: the one deepest in the tree and closest to the new node. After inserting the key, the balance factors are adjusted during the unwinding of the recursion. The first subtree encountered that is out of balance has to be rebalanced. The root node of this subtree is known as the pivot node. An AVL subtree is rebalanced by performing a rotation around the pivot node. This involves rearranging the links of the pivot node, its children, and possibly one of its grandchildren. There are four possible cases is shown in Fig 4.14.

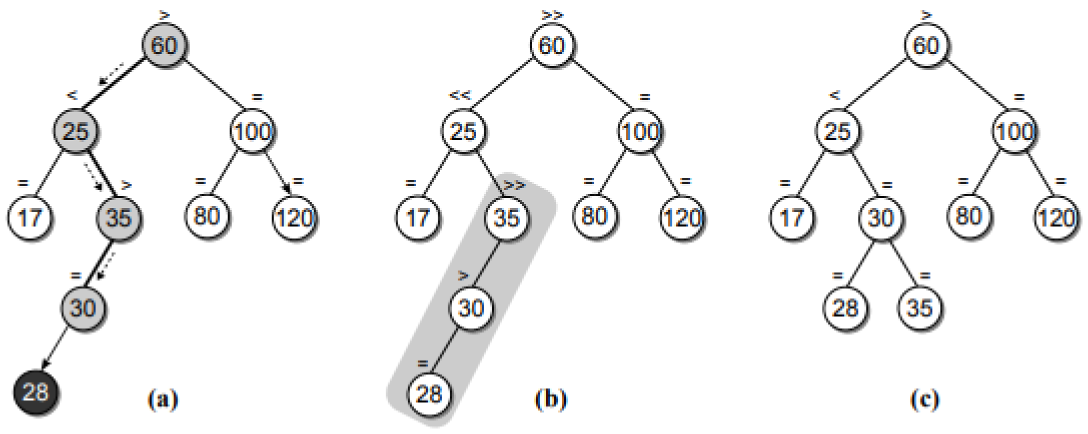


Fig.4.14 Rotation in AVL tree

i) **Case 1:** When the balance factor of the pivot node (P) is left high before the insertion and the new key is inserted into the left child (C) of the pivot node. To rebalance the subtree, the pivot node has to be rotated right over its left child. The rotation is accomplished by changing the links such that P becomes the right child of C and the right child of C becomes the left child of P is shown in Fig 4.15.

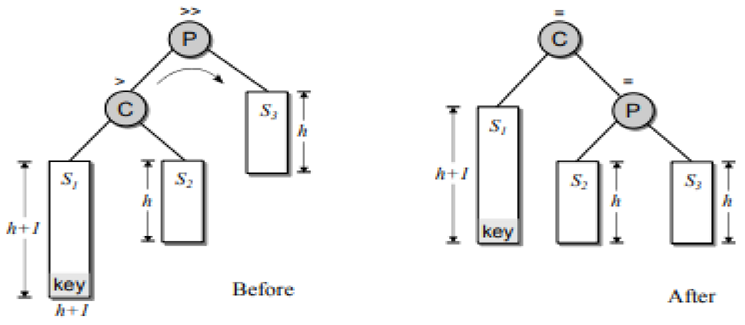


Fig 4.15 Case 1

ii) **Case 2:** the pivot (P), the left child of the pivot (C), and the right child (G) of C. For this case to occur, the balance factor of the pivot is left high before the insertion and the new key is inserted into either the right subtree of C. Node C has to be rotated left over node V and the pivot node has to be rotated right over its left child. The right child of G as the new left child of the pivot node, changing the left child of G to become the right child of C, and setting C to be the new left child of G is shown in Fig 4.16.

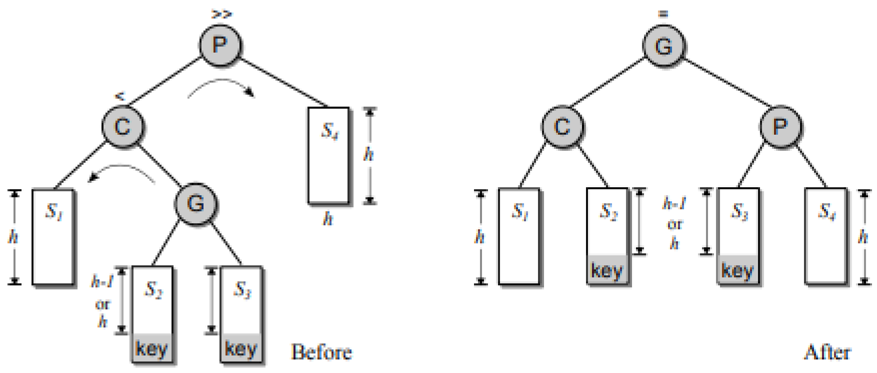


Fig 4.16 Case 2

iii) **Case 3 and 4:** The third case is a mirror image of the first case and the fourth case is a mirror image of the second case. The difference is the new key is inserted in the right subtree of the pivot node or a descendant of its right subtree is shown in Fig.4.17.

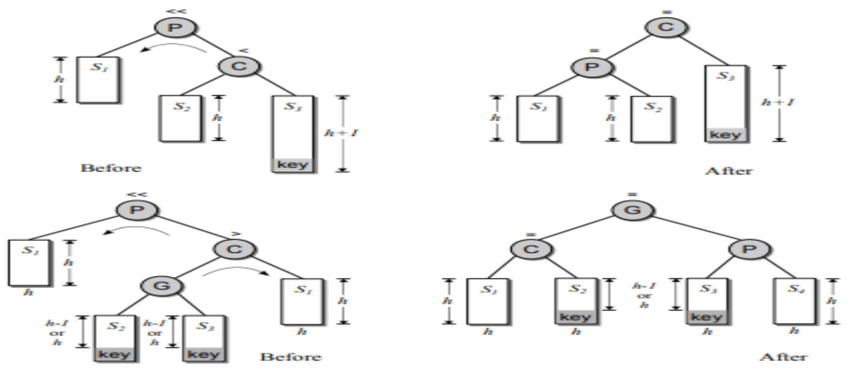


Fig 4.17 Case 3

Deletions

When an entry is removed from an AVL tree, we must ensure the balance property is maintained. For example, suppose we want to remove key 17 from the AVL tree in Figure 14.21(a). After removing the leaf node, the subtree rooted at node 25 is out of balance, as shown below. A left rotation has to be performed pivoting on node 25 to correct the imbalance is shown in Fig 4.18.

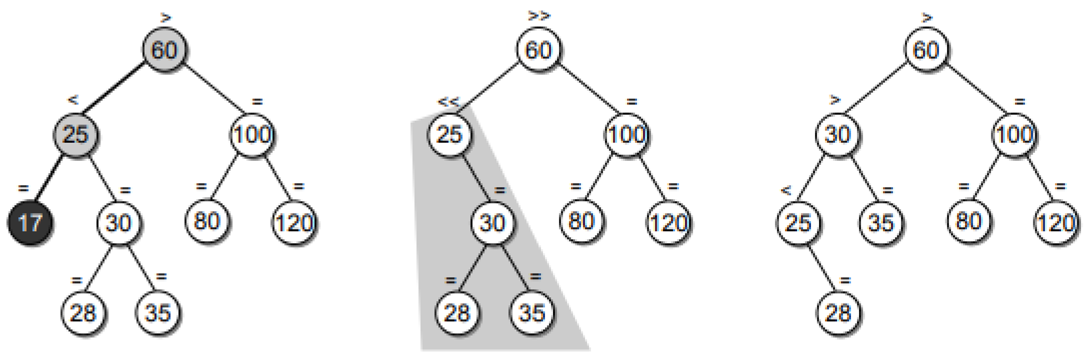


Fig 4.18 Deletion in AVL Tree

In the insertion operation, at most one subtree can become unbalanced. After the appropriate rotation is performed on the subtree, the balance factors of the node's ancestors do not change. Thus, it restores the height-balance property both locally at the subtree and globally for the entire tree. This is not the case with a deletion. When a subtree is rebalanced due to a deletion, it can cause the ancestors of the subtree to then become unbalanced. This effect can ripple up all the way to the root node. So, all of the nodes along the path have to be evaluated and rebalanced if necessary.

4.5 Tree Traversal

There are three standard ways of traversing a binary tree T with root R. They are:

(i) Preorder Traversal

(ii) Inorder Traversal

(iii) Postorder Traversal

Preorder Traversal:

(1) Process the root R.

(2) Traverse the left subtree of R in preorder.

(3) Traverse the right subtree of R in preorder.

Inorder Traversal:

(1) Traverse the left subtree of R in inorder.

(2) Process the root R.

(3) Traverse the right subtree of R in inorder.

Postorder Traversal:

(1) Traverse the left subtree of R in postorder.

(2) Traverse the right subtree of R in postorder.

(3) Process the root R.

Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree. The difference between the algorithms is the time at which the root R is processed. The three algorithms are sometimes called, respectively, the node-left-right (NLR) traversal, the left-node-right (LNR) traversal and the left-right-node (LRN) traversal.

Traversal algorithms

Preorder Traversal

Consider the following case where we have 6 nodes in the tree A, B, C, D, E, F. The traversal always starts from the root of the tree. The node A is the root and hence it is visited first. The value at this node is processed. Now we check if there exists any left child for this node if so apply the preorder procedure on the left subtree. Now check if there is any right subtree for the node A, the preorder procedure is applied on the right subtree.

Since there exists a left subtree for node A, B is now considered as the root of the left subtree of A and preorder procedure is applied. Hence we find that B is processed next and then it is checked if B has a left subtree is shown in Fig. 4.19

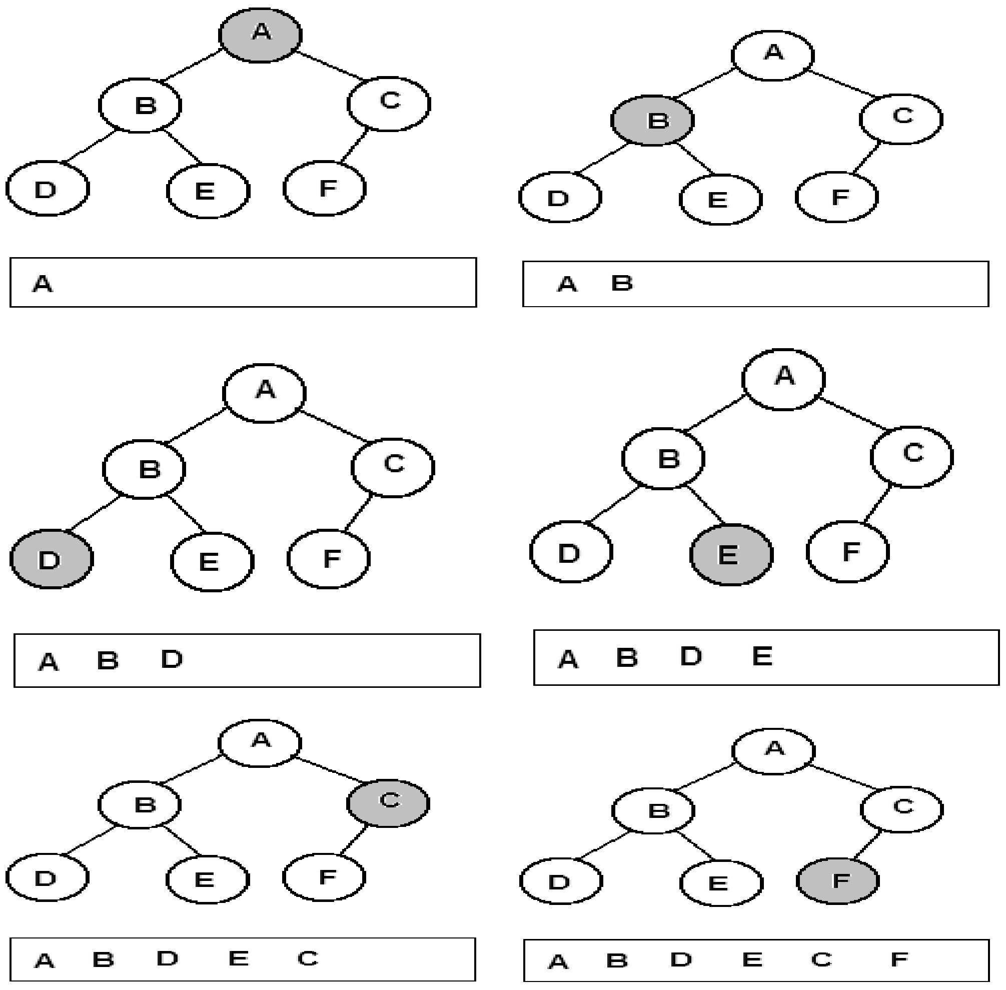


Fig. 4.19 Preorder Traversal

The algorithm for the above method is presented in the pseudo-code form below:

Algorithm

PREORDER(ROOT)

```
Temp = ROOT
If temp = NULL
    Return
End if
Print info(temp)
If left(temp) ≠ NULL
    PREORDER( left(temp))
End if
If right(temp) ≠ NULL
    PREORDER(right(temp))
End if
End PREORDER
```

Inorder Traversal

In the Inorder traversal method, the left subtree of the current node is visited first and then the current node is processed and at last the right subtree of the current node is visited. In the following example, the traversal starts with the root of the binary tree. The node A is the root and it is checked if it has the left subtree. Then the inorder traversal procedure is applied on the left subtree of the node A. Now we find that node D does not have left subtree. Hence the node D is processed and then it is checked if there is a right subtree for node D. Since there is no right subtree, the control returns back to the previous function which was applied on B. Since left of B is already visited, now B is processed. It is checked if B has the right subtree. If so apply the inorder traversal method on the right subtree of the node B. This recursive procedure is followed till all the nodes are visited is shown in Fig.4.20.

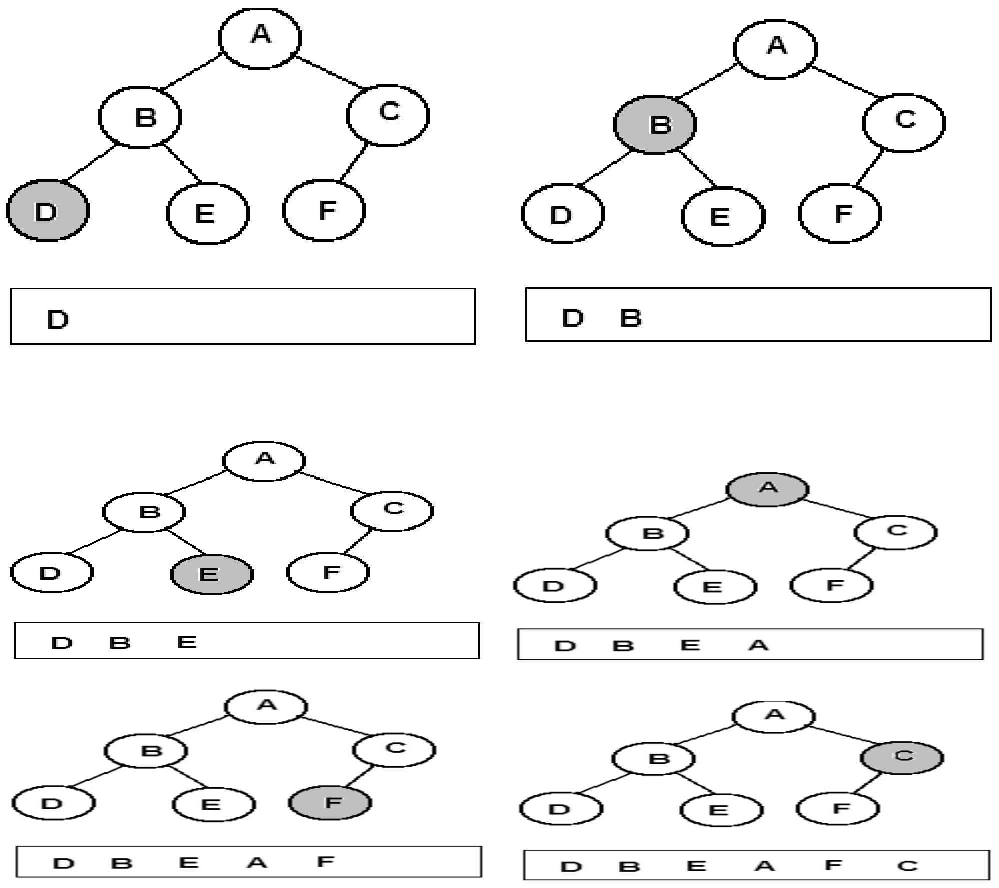


Fig 4.20 Inorder Traversal

Algorithm

INORDER(ROOT)

```
Temp = ROOT
If temp = NULL
        Return
End if
If left(temp) ≠ NULL
        INORDER(left(temp))
End if
Print info(temp)
If right(temp) ≠ NULL
        INORDER(right(temp))
End if
End INORDER
```

Postorder Traversal

In the postorder traversal method the left subtree is visited first, then the right subtree and at last the current node is processed. In the following example, A is the root node. Since A has the left subtree the postorder traversal method is applied recursively on the left subtree of A. Then when left subtree of A is completely is processed, the postorder traversal method is recursively applied on the right subtree of the node A. If right subtree is completely processed, then the current node A is processed is shown in Fig 4.21.

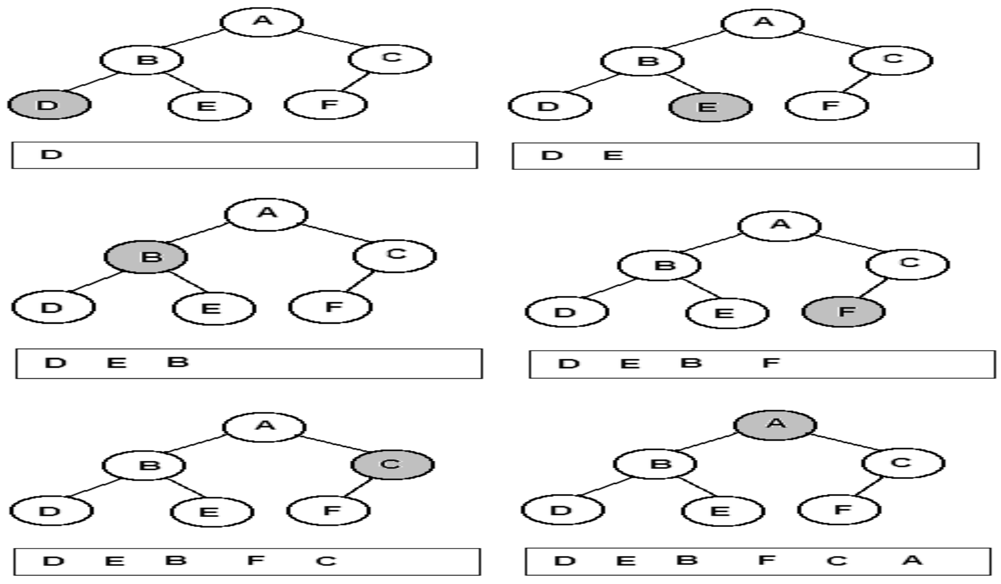


Fig 4.21 Postorder Traversal

Algorithm

```
POSTORDER( ROOT )
Temp = ROOT
If temp = NULL
    Return
End if
If left(temp) ≠ NULL
    POSTORDER(left(temp))
End if
If right(temp) ≠ NULL
    POSTORDER(right(temp))
End if
Print info(temp)
End POSTORDER
```

4.6 B TREES

An extension of a multiway search tree of order m is a B-tree of order m. This type of tree will be used when the data to be accessed / stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node. A B-tree of order m is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.

2. Each nonroot and each nonleaf node have at most m nonempty children and at least m/2 nonempty children.

3. The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.

4. All leaves are on the same level.

Searching

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

Insertion

The condition that all leaves must be on the same level forces a characteristic behavior of Btrees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root. When inserting into a B-tree, a value is inserted directly into a leaf is shown in Fig 4.22 and Fig.4.23. This leads to three common situations that can occur:

1. A key is placed into a leaf that still has room.

2. The leaf in which a key is to be placed is full.

3. The root of the B-tree is full.

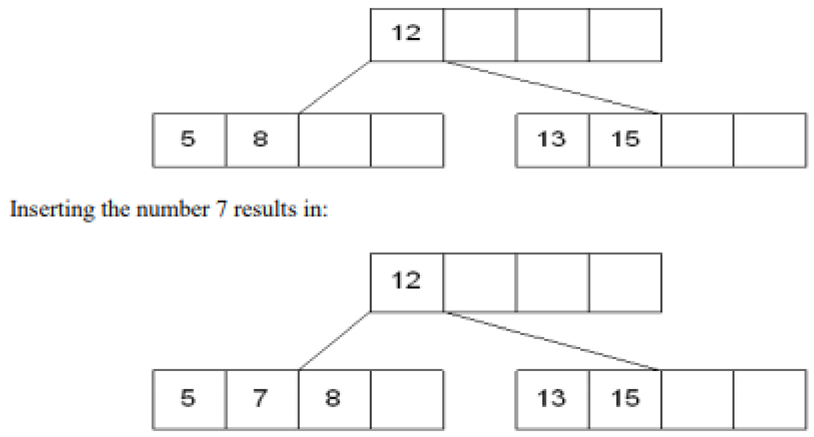
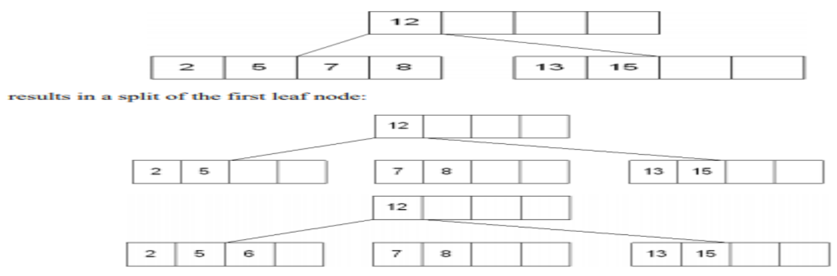
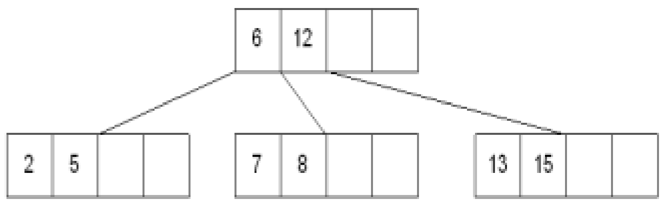


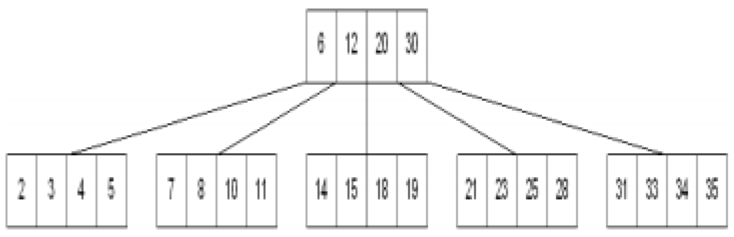
Fig 4.22 B-tree Insertion

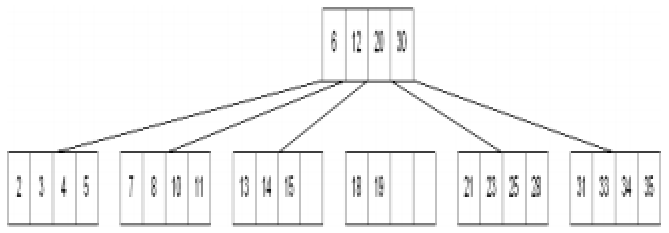
Insert 6 into the following B-tree:


results in a split of the first leaf node:

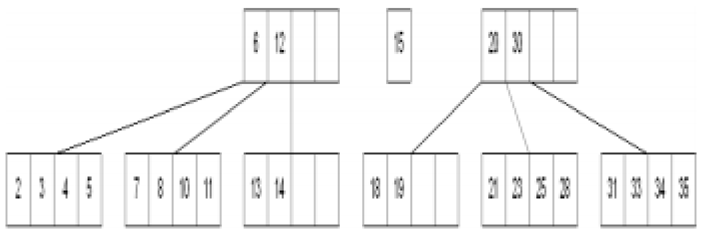


Inserting 13 into the following tree:





The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:

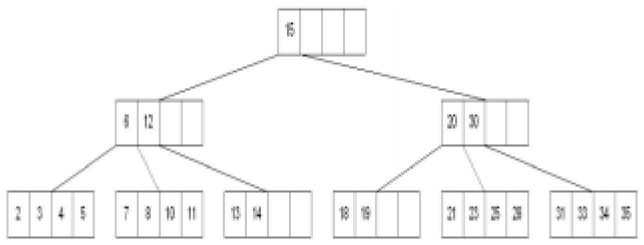


Fig 4.23 steps in B-tree insertion operation

Deletion

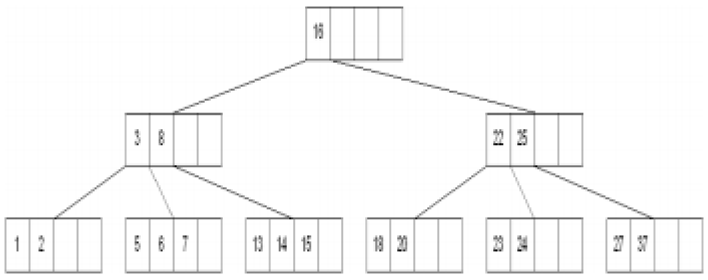
The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained is shown in Fig 4.24 ,Fig 4.25 and Fig 4.26.

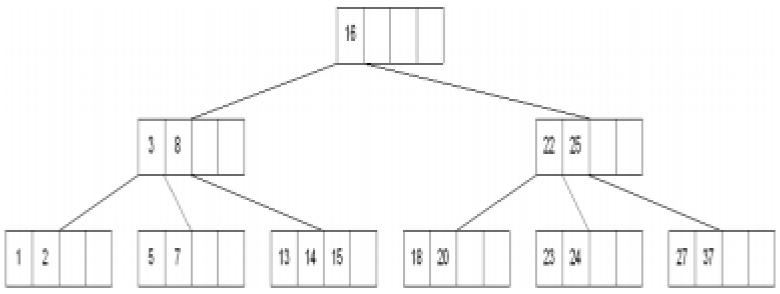
There are two main cases to be considered:

1. Deletion from a leaf

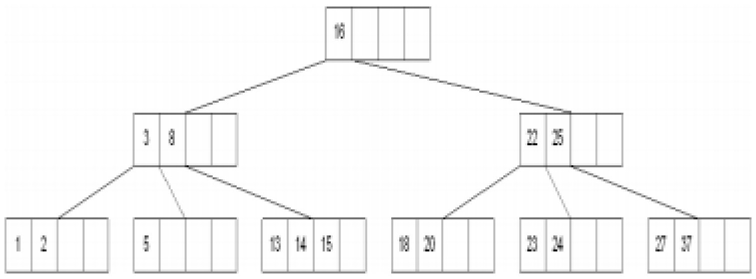
2. Deletion from a non-leaf

Deleting 6 from the following tree:





Deleting 7 from the tree above results in:



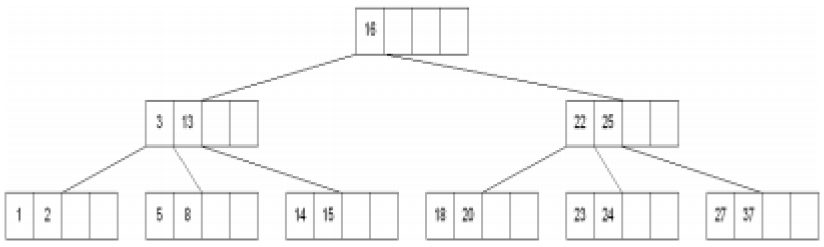
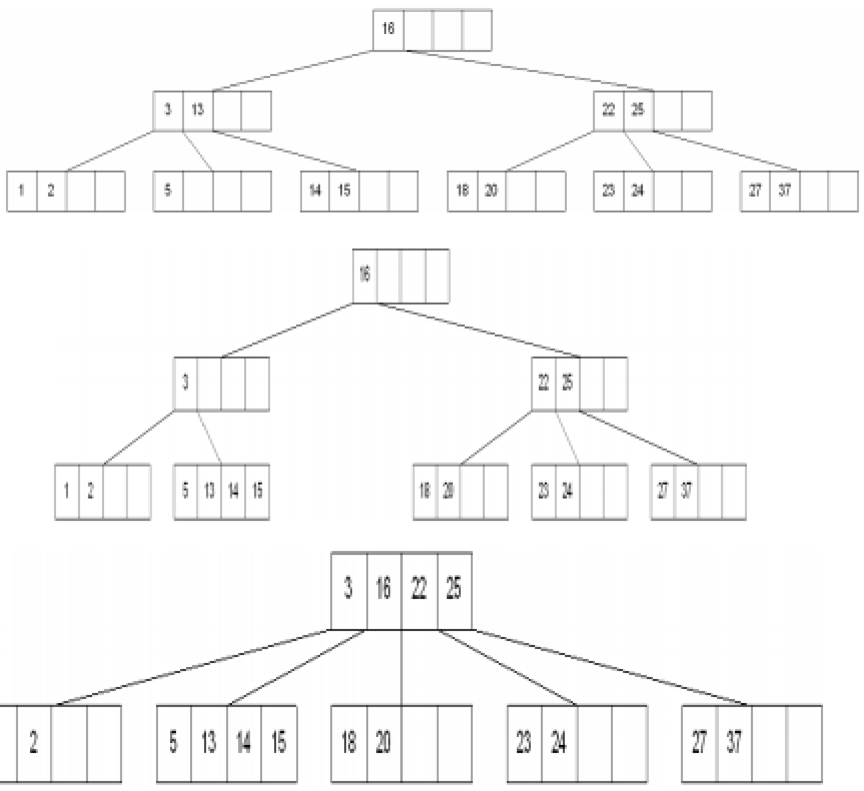


Fig 4.24 Deletion

Now delete 8 from the tree:



Deleting 16 from the tree above results in:

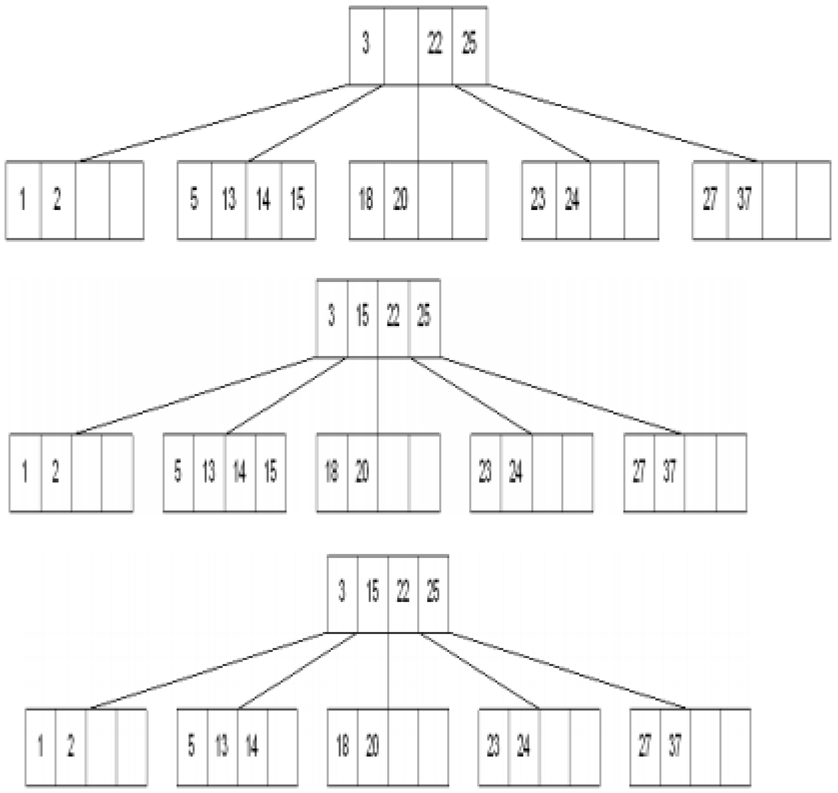


Fig 4.25 Deletion

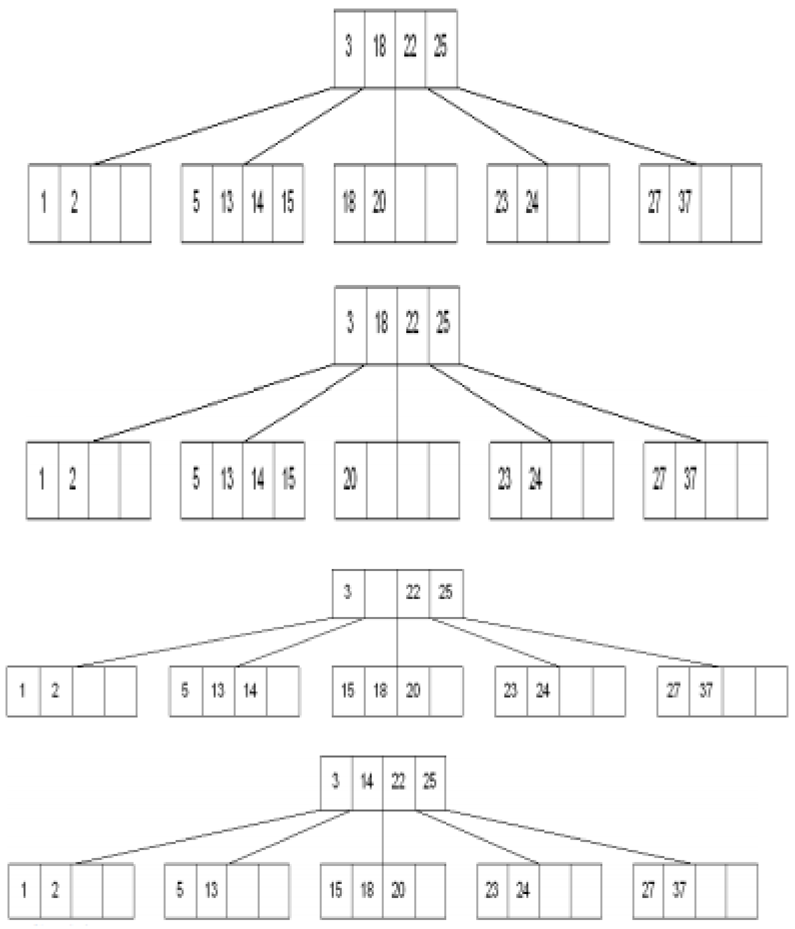


Fig 4.26 steps in B Tree deletion operation

4.7 HEAP TREE

Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. A heap is a complete binary tree in which the nodes are organized based on their data entry values. There are two variants of the heap structure. A max-heap has the property, known as the heap order property, that for each non-leaf node V, the value in V is greater than the value of its two children. The largest value in a max-heap will always be stored in the root while the smallest values will be stored in the leaf nodes. The min-heap has the opposite property. For each non-leaf node V, the value in V is smaller than the value of its two children is shown in Fig.4.27 and Fig.4.28.

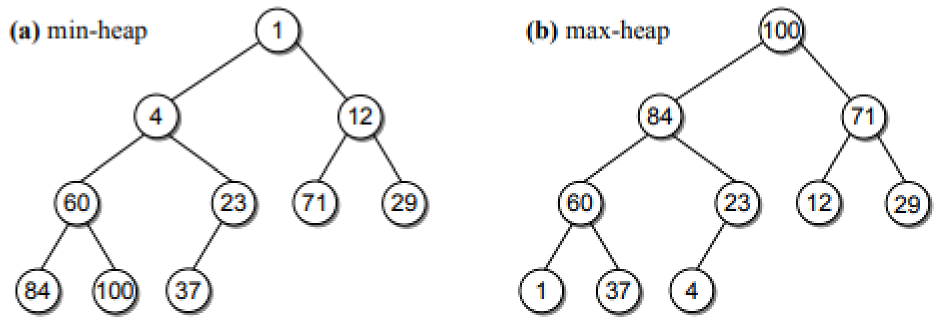


Fig 4.27 Min and Max heap

<u>Example</u>

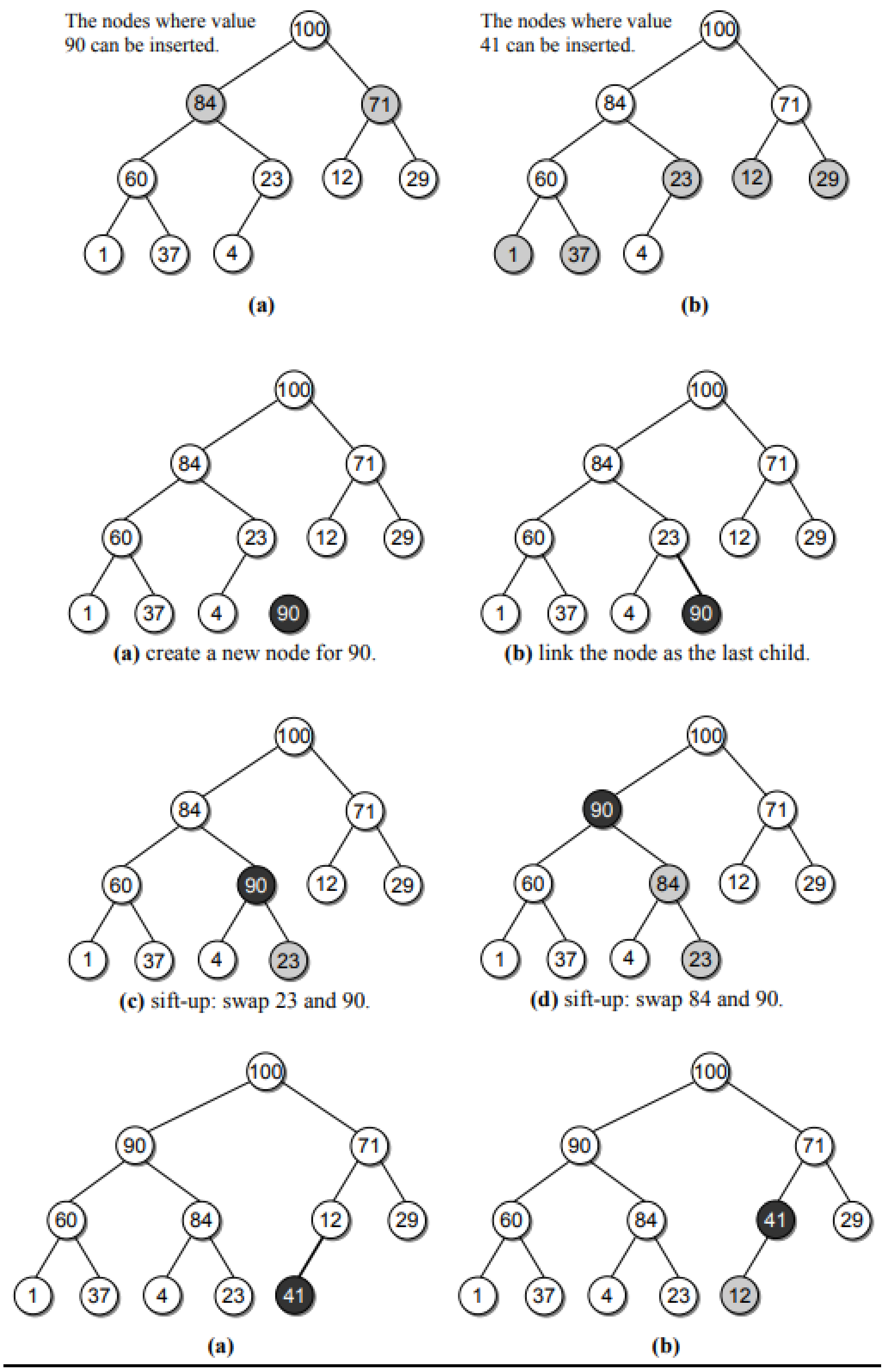

The nodes where value 90 can be inserted.
(a)
The nodes where value 41 can be inserted.
(b)
(a) create a new node for 90.
(b) link the node as the last child.
(c) sift-up: swap 23 and 90.
(d) sift-up: swap 84 and 90.
(a)
(b)

Fig 4.28 Heap tree insertion

4.8 PRELIMINARIES OF GRAPH ADT

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads. A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G=(V,E) Where V={A,B,C,D,E} and E={(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges is shown in Fig.4.29.

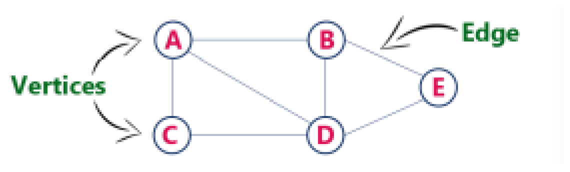


Figure 4.29 A Graph

Graph Terminology

1. Vertex : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.
2. Edge : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1. Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

2. Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3. Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1. Undirected Graph
   A graph with only undirected edges is said to be undirected graph.

2. Directed Graph
   A graph with only directed edges is said to be directed graph.

3. Complete Graph

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.

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

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

6. Acyclic Graph
   A graph without cycle is called acyclic graphs.

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

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

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

Incidence

In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

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

Closed walk

A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk. If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

A open walk in which no vertex appears more than once is called a path. If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

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

Circuit

A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit. A circuit having three vertices and three edges.

Sub Graph

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of $\subseteq$ V(G) and E(G’) $\subseteq$S has the same end vertices in S as in G. A subgraph of G is a graph G’ such that V(G’) E(G)

Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise, G is disconnected. This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it. In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

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

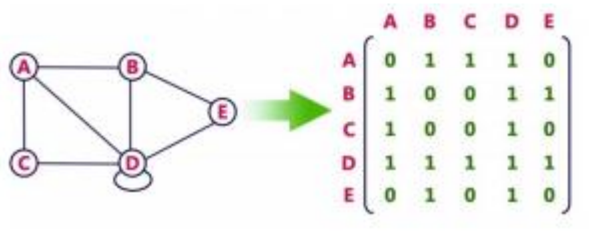
4.9 Graph Representations
Graph data structure is represented using following representations
1. Adjacency Matrix
2. Adjacency List
3. Adjacency Multilists

1.Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size. In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

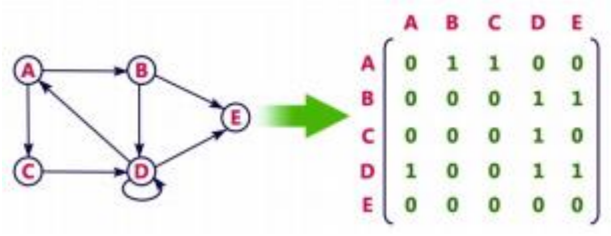
Adjacency Matrix: let G = (V, E) with n vertices, n≥1. The adjacency matrix of G is a 2-dimensional n x n matrix, A, A(i, j) = 1 iff $(v_i\ v_j) \epsilon E(G)$ ($<v_i, v_j>$ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

For a Directed graph



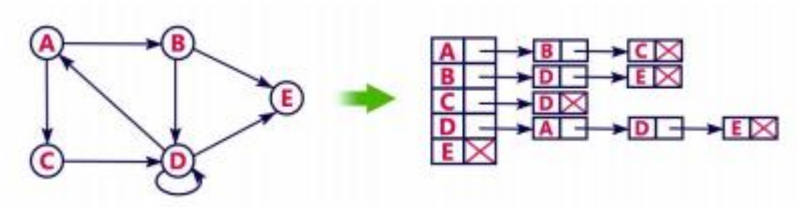
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric. The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

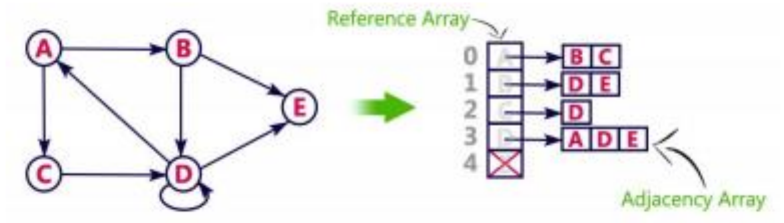
- Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.
It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies.

So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. example: consider the following directed graph representation implemented using linked list



This representation can also be implemented using array

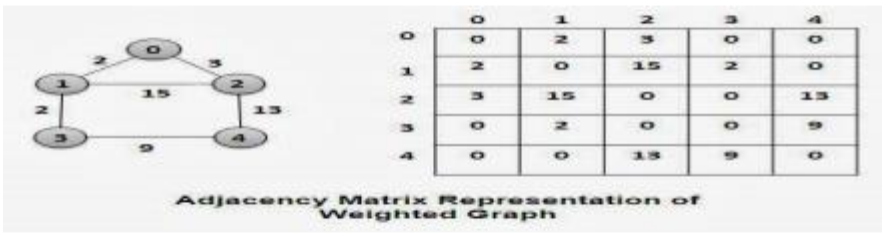


- Adjacency Multilists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on that list for v. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

- Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list nodes by including an additional field weight. A graph with weighted edges is called a network.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Fig 27 Weighted Graph - Adjacency Matrix representation

4.10 GRAPH TRAVERSAL

BREADTH FIRST SEARCH (BFS) TRAVERSAL FOR A GRAPH

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.

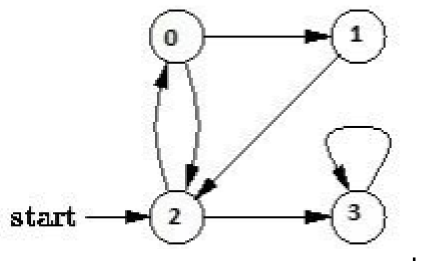


Fig 28 BFS Traversal graph

Algorithm: Breadth-First Search Traversal

BFS(V, E, s)

```
for each u in V − {s}
    do color[u] ← WHITE
        d[u] ← infinity
        π[u] ← NIL
        color[s] ← GRAY
       d[s] ← 0
       π[s] ← NIL
      Q ← {}
      ENQUEUE(Q, s)
    while Q is non-empty
    do u ← DEQUEUE(Q)
        for each v adjacent to u
            do if color[v] ← WHITE
                then  color[v] ← GRAY
                        d[v] ← d[u] + 1
                          π[v] ← u
                          ENQUEUE(Q, v)
          DEQUEUE(Q)
       color[u] ← BLACK
```

Applications of Breadth First Traversal

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also

be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney’s algorithm.

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).

10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

DEPTH FIRST TRAVERSAL FOR A GRAPH

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the 103 following graph is 2, 0, 1, 3

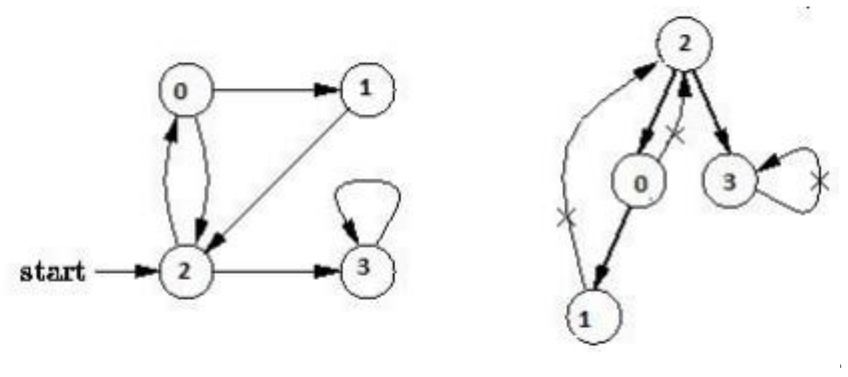


Fig 29 DFS Traversal Graph

Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

```
DFS (V, E)

for each vertex u in V[G]
     do color[u] ← WHITE
          π[u] ← NIL
   time ← 0
  for each vertex u in V[G]
     do if color[u] ← WHITE
          then DFS-Visit(u)

DFS-Visit(u)

   color[u] ← GRAY
   time ← time + 1
   d[u] ← time
   for each vertex v adjacent to u
     do if color[v] ← WHITE
          then π[v] ← u
               DFS-Visit(v)
   color[u] ← BLACK
   time ← time + 1
   f[u] ← time
```

Applications of Depth First Search

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) Path Finding We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

4) Topological Sorting

5) To test if a graph is bipartite We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

4.11 SHORTEST PATH ALGORITHM

Given a graph where edges are labeled with weights (or distances) and a source vertex, what is the shortest path between the source and some other vertex? Problems requiring us to answer such queries are broadly known as shortest-paths problems. Shortest-paths problem come in several flavors. For example, the single-source shortest path problem requires finding the shortest paths between a given source and all other vertices; the single-pair shortest path problem requires finding the shortest path between given a source and a given destination vertex; the all-pairs shortest path problem requires finding the shortest paths between all pairs of vertices.

4.11.1 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is an iterative algorithm that provides us with the shortest path from one particular starting mode to all other nodes in the graph. To keep track of the total cost from the start node to each destination we will make use of the distance instance variable in the vertex class. The distance instance variable will contain the current total weight of the smallest weight path from the start to the vertex. Dijkstra's algorithm finds the shortest path in a **weighted graph** containing only positive edge weights from a single source. It uses a priority based dictionary or a queue to select a node / vertex nearest to the source that has not been edge relaxed. Time complexity of Dijkstra's algorithm is O((E+V) Log(V)) for an adjacency list implementation of a graph. V is the number of vertices and E is the number of edges in a graph.

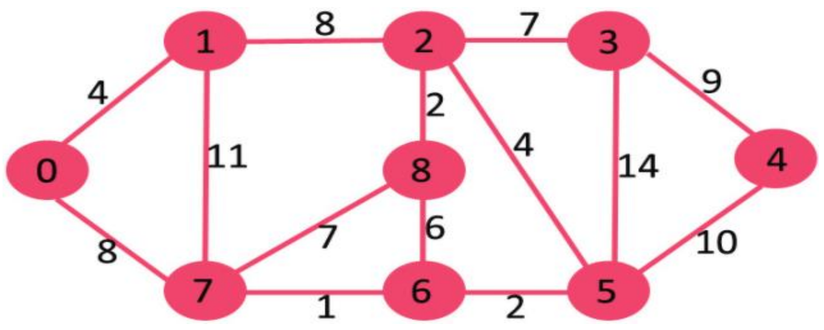
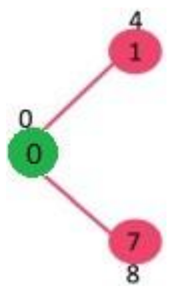
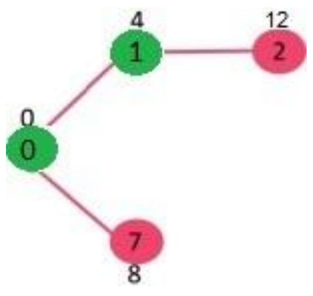
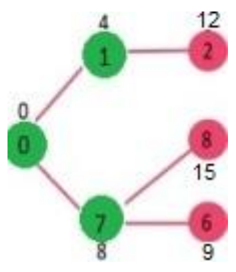


Fig 30 Example for Dijkstra's algorithm

Step 1: Step 2: Step 3:

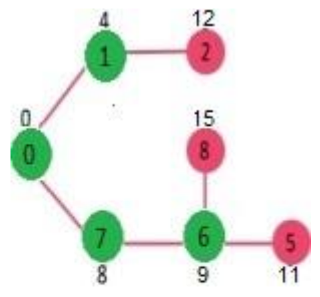




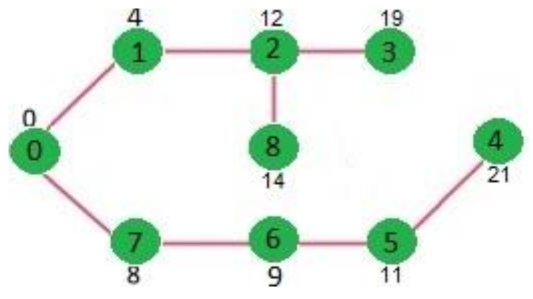


On repeating the above steps until the set contains all vertices of given graph. Then we get the following Shortest Path.

Step 4:



Step 5:



4.11.2 PRIM'S ALGORITHM

Prim's algorithm is also a greedy algorithm technique. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in minimum spanning tree
- Contain vertices not yet included

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.

a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.

b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

Example:

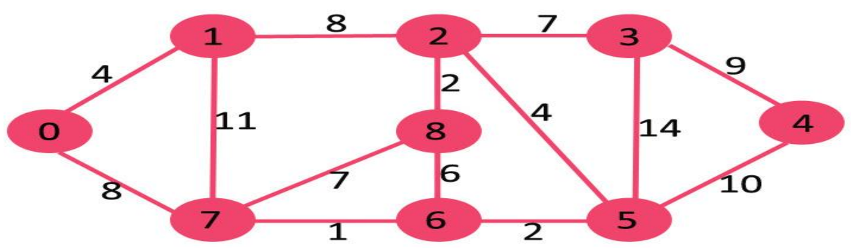
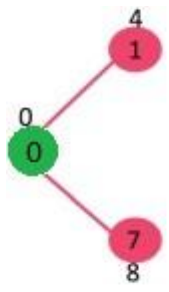
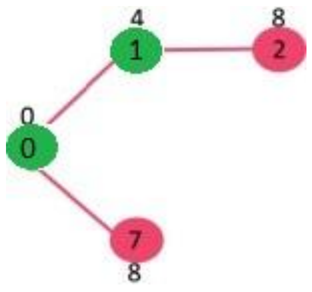


Fig 31 Example for Prim's algorithm

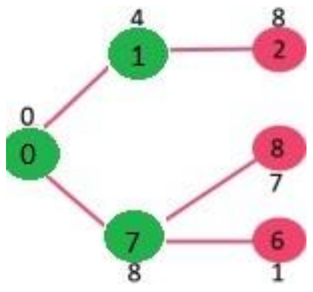
Step 1:



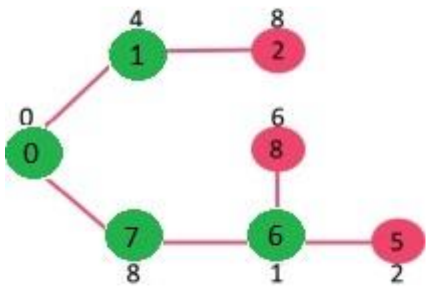
Step 2:



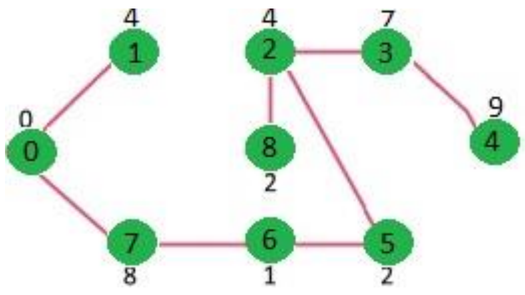
Step 3:



Step 4:



Step 5:



SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V-Data Structures – SCSA1205

UNIT V

Divide and Conquer Strategy – Greedy Algorithm – Dynamic Programming – Backtracking Strategy - List Searches using Linear Search - Binary Search - Fibonacci Search - Sorting Techniques - Insertion sort - Heap sort - Bubble sort - Quick sort - Merge sort - Analysis of sorting techniques.

5.1 DIVIDE AND CONQUER STRATEGY

Divide and conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps:

1. Divide: This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
2. Conquer: This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
3. Merge/Combine: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

The some examples of Divide and conquer problem based algorithms are

- Merge Sort
- Quick Sort
- Binary Search
- Master Theorem
- Fibonacci Search
- Strassen's Matrix multiplication
- Karatsuba Algorithm

Advantages of Divide and Conquer algorithm

- The complexity for the multiplication of two matrices using the naïve method is O(n^3), whereas using the divide and conquer approach. This approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

5.2 GREEDY ALGORITHM

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment, without worrying about the future result it would bring. In other words, the locally best choices aim at producing globally best results. This algorithm may not be the best option for all the problems. It may produce wrong results in some cases. This algorithm never goes back to reverse the decision made.

A greedy 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. Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions. This algorithm works in a top-down approach.

The main advantage of greedy algorithm is:

1. The algorithm is easier to describe
2. This algorithm can perform better than other algorithms.

Steps involved in greedy algorithm

- To begin with, the solution set is empty.
- At each step, an item is added into the solution set.
- If the solution set is feasible, the current item is kept.
- Else, the item is rejected and never considered again.

Some examples of networking algorithms using the greedy approach are

- Travelling salesman problem
- Prim’s Minimal Spanning Tree Algorithm
- Kruskal’s Minimal Spanning Tree Algorithm
- Dijkstra’s Minimal Spanning Tree Algorithm
- Graph – Map coloring
- Graph – vertex cover
- Knapsack Problem
- Job Scheduling Problem
- Huffman Coding

5.3 DYNAMIC PROGRAMMING

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But, unlike, divide and conquer, these sub-problems are not solved independently. Here the results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Some problems that use dynamic programming approach

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

5.4 BACKTRACKING ALGORITHM

A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach. This approach is used to solve problems that have multiple solutions. If you want an optimal solution, you must go for dynamic programming. **Backtracking** is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example:

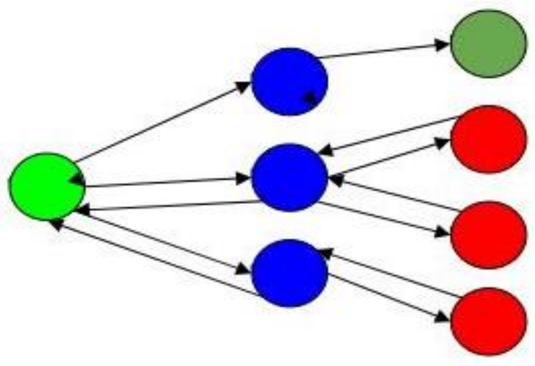


Fig 5.1 Backtracking algorithm method

In fig 5.1, Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution. Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find track to the next point to find solution.

Algorithm

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

Let’s use this backtracking problem to find the solution to **N-Queen Problem**.

In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do 4-Queen problem is shown in Fig 5.2.

Here, the solution is −

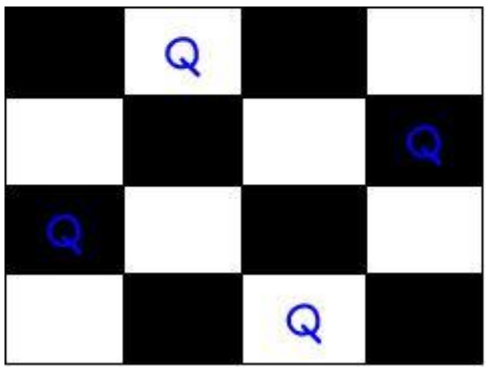


Fig 5.2 N Queens Problem

Here, the binary output for n queen problem with 1's as queens to the positions are placed.

{0 , 1 , 0 , 0}
{0 , 0 , 0 , 1}
{1 , 0 , 0 , 0}
{0 , 0 , 1 , 0}

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

State Space Tree

A space state tree is a tree representing all the possible states (solution or non-solution) of the problem from the root as an initial state to the leaf as a terminal state .

5.4 Linear Search

Linear search is the simplest method of searching. In this method, the element to be found is sequentially searched in the list (Hence also called sequential search). This method can be applied to a sorted or an unsorted list. Hence, it is used when the records are not stored in order.

Algorithm :

```
ALGORITHM LINEARSEARCH(K, N, X )
// K is the array containing the list of data items
// N is the number of data items in the list
// X is the data item to be searched
Repeat For I = 0 to N -1 Step 1
        If K( I ) = X
        Then
                WRITE("ELEMENT IS PRESENT AT LOCATION " I)
                RETURN
        End If
End Repeat
                WRITE("ELEMENT NOT PRESENT IN THE COLLECTION")
End LINEARSEARCH
```

In the above algorithm, K is the list of data items containing N data items. X is the data item, which is to be searched in K. If the data item to be searched is found then the position where it is found will be displayed. If the data item to be searched is not found then the appropriate message will be displayed to indicate the user, that the data item is not found.

The data item X is compared with each and every element in the list K During this comparison, if X matches with a data item in K, then the position where the data item was found will gets displayed and the control comes out of the loop and the procedure comes to an end. If X does not match with any of the data items in K, then finally the element not found will be displayed.

Example:

X → Number to be searched : **40**

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

$X \neq K[0]$

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[1]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[2]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[3]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[4]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[5]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X ≠ K[6]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

X = K[7] → I = 7 : Number found at location 7 i.e., as a 8th element

The **search( )** function gets the number to be searched in the variable 'x' as a argument and compares it with each and every element in the array K. If the number 'x' is found in the array, then the position 'i', where it is found will gets printed. If the number is not found in the entire list, then the function will display the "not found message" to the user.

In the **main( )** function receives the n values from the user and stored in the array K. The user is prompted to enter the number to be searched and is passed to the search( ) function as a argument. The search which receives the value x will give the appropriate message.

Advantages:

1. Simple and straight forward method.
2. Can be applied on both sorted and unsorted list.

Disadvantages:

1. Inefficient when the number of data items in the list increases.

Analysis

Worst Case :O(n)

Best Case:Ω(1)

Average Case:ϴ(n)

5.5 BINARY SEARCH

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order. Binary search cannot be applied on an unsorted list.

Principle: The data item to be searched is compared with the approximate middle entry of the list. If it matches with the middle entry, then the position will be displayed. If the data item to be searched is lesser than the middle entry, then it is compared with the middle entry of the first half of the list and procedure is repeated on the first half until the required item is found. If the data item is greater than the middle entry, then it is compared with the middle entry of the second half of the list and procedure is repeated on the second half until the required item is found. This process continues until the desired number is found or the search interval becomes empty.

Algorithm:

ALGORITHM BINARYSEARCH(K, N, X)

```
// K is the array containing the list of data items
// N is the number of data items in the list
// X is the data item to be searched
Lower ← 0, Upper ← N – 1
While Lower ≤ Upper
        Mid ← ( Lower + Upper ) / 2
        If (X < K[Mid])Then
                        Upper ← Mid -1
                Else If (X>K[Mid]) Then
                        Lower ← Mid + 1
                Else
                        Write(“ELEMENT FOUND AT”, MID)
                        Quit
                    End If
                End If
End While
                Write("ELEMENT NOT PRESENT IN THE COLLECTION")
End BINARYSEARCH
```

In Binary Search algorithm given above, K is the list of data items containing N data items. X is the data item, which is to be searched in K. If the data item to be searched is found then the position where it is found will be printed. If the data item to be searched is not found then "Element Not Found" message will be printed, which will indicate the user, that the data item is not found.

Initially lower is assumed 0 to point the first element in the list and upper is assumed as N-1 to point the last element in the list because the range of any array is 0 to N-1. The mid position of the list is calculated by finding the average between lower and upper and X is compared with K[mid]. If X is found equal to K[mid] then the value mid will gets printed, the control comes out of the loop and the procedure comes to an end. If X is found lesser than K[mid], then upper is assigned mid – 1, to search only in the first half of the list. If X is found greater than K[mid], then lower is assigned mid + 1, to search only in the second half of the list. This process is continued until the element searched is found or the collection becomes becomes empty.

Example:

X → Number to be searched : **40**

U → Upper

L → Lower=N-1

M→ Mid

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = (0+9)/2 =4 U = 9

X< K[4] → U = 4 – 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = (0+3)/2=1 U = 3

X > K[1] → L = 1 + 1 = 2

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M = 2   U = 3

K > A [2] → L = 2 + 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M, U = 3

K = A[3] → P = 3 : Number found at position 3

The binarysearch( ) function gets the element to be searched in the variable X. Initially lower is assigned 0 and upper is assumed N – 1. The mid position is calculated and if K[mid] is found equal to X, then mid position will gets displayed. If X is less than K[mid] upper is assigned mid – 1 to search only in first half of the list else lower is assigned mid + 1 to search only in the second half of the list. This is process is continued until lower is less than or equal to upper. If the element is not found even after the loop is completed, then the Not Found Message will be displayed to the user indicating that the element is not found.

Advantages:

1. Searches several times faster than the linear search.
2. In each iteration, it reduces the number of elements to be searched from n to n/2.

Disadvantages:

1. Binary search can be applied only on a sorted list.

Analysis of Binary Search

Bestcase :O(1)

Worst Case: O($\log_2 n$)

Average Case: O($\log_2 n$)

5.6 FIBONACCI SEARCH

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array. Fibonacci search has some similarities and differences when compared to the binary search.

Similarities:

1. Works for sorted arrays

2. A Divide and Conquer Algorithm.
3. Has Log n time complexity.

Differences:

1. Fibonacci Search divides given array into unequal parts
2. Binary Search uses a division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
3. Fibonacci Search examines relatively closer elements in subsequent steps. So when the input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Fibonacci Numbers are recursively defined as F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1. First few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,… Observations:

Below observation is used for range elimination, and hence for the O(log(n)) complexity.

1. F(n - 2) &approx; (1/3)*F(n) and
2. F(n - 1) &approx; (2/3)*F(n).

Algorithm:

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
1. Compare x with the last element of the range covered by fibMm2
2. **If** x matches, return index
3. **Else if** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
4. **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

Analysis

- The complexity of Fibonacci search is O($\log_2 n$).
- The performance of Fibonacci search is poor than binary search.

- However, binary search involves division operation, where as in Fibonacci search only addition and subtraction operation is involved. Average performance of Fibonacci search may be better than binary search where division is more time consuming than addition or subtraction.

5.7 Sorting

Sorting is an operation of arranging data, in some given order, such as ascending or descending with numerical data, or alphabetically with character data.

Let A be a list of n elements $A_1, A_2, \ldots A_n$ in memory. Sorting A refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is, $A_1 \le A_2 \le A_3 \le \ldots \le A_n$

Sorting methods can be characterized into two broad categories:

- Internal Sorting
- External Sorting

Internal Sorting : Internal sorting methods are the methods that can be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory.

The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access in this memory space can be effectively used to sort the data items.

The various internal sorting methods are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort

External Sorting : External sorting methods are the methods to be used when the list to be sorted is large and cannot be accommodated entirely in the main memory. In this case some of the data is present in the main memory and some is kept in auxiliary memory such as hard disk, floppy disk, tape, etc.

Objectives involved in design of sorting algorithms.

The main objectives involved in the design of sorting algorithm are:

1. Minimum number of exchanges
2. Large volume of data block movement

This implies that the designed and desired sorting algorithm must employ minimum number of exchanges and the data should be moved in large blocks, which in turn increase the efficiency of the sorting algorithm.

5.7.1 INSERTION SORT

The main idea behind the insertion sort is to insert the ith element in its correct place in the ith pass. Suppose an array K with n elements K[1], K[2],…K[N] is in memory. The insertion sort algorithm scans K from K[0] to K[N-1], inserting each element K[I] into its proper position in the previously sorted subarray K[0], K[1],..K[I-1].

Principle: In Insertion Sort algorithm, each element K[I] in the list is compared with all the elements before it (K[1] to K[I-1]). If any element K[J] is found to be greater than K[I] then K[J] is inserted in the place of K[J}. This process is repeated till all the elements are sorted.

Algorithm:

```
ALGORITHM INSERTIONSORT(K, N)
// K is the array containing the list of data items
// N is the number of data items in the list
Repeat For I = 1 to N-1
    Repeat For J = 0 to I – 1
        If (K[I] < K[J])Then
            Temp ← K[I]
            Repeat For L = I-1 to J
                K[L +1] ← K[L]
            End Repeat
            K[J] ← Temp
        End If
    End Repeat
End Repeat
End INSERTIONSORT
```

In Insertion Sort algorithm, N represents the total number of elements in the array K. I is made to point to the second element in the list. In every pass the J is incremented to point to the next element and is continued till it reaches the last element. During each pass K[I] is compared all elements before it. If K[I] is lesser than K[J] in the list, then K[I] is inserted in position J. Finally, a sorted list is obtained.

For performing the insertion operation, a variable temp is used to safely store K[I] in it and then shift right elements starting from K[J] to K[I-1].

Example:

N = 10 → Number of elements in the list

L → Last

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

I=1 K[I] < K[0] → Insert K[I] at 0 L=9

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

I=2 L=9

K[I] is greater than all elements before it. Hence No Change

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

I=3 K[I] < K[0] → Insert K[I] at 0 L=9

11	**23**	**42**	**74**	**65**	**58**	**94**	**36**	**99**	**87**

I=4 L=9

K[I] < K[3] → Insert K[I] at 3

11	**23**	**42**	**65**	**74**	**58**	**94**	**36**	**99**	**87**

I=5 L=9

K[I] < K[3] → Insert K[I] at 3

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**

I=6 L=9

K[I] is greater than all elements before it. Hence No Change

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**
							I=7		L=9

K[I] < K[2] → Insert K[I] at 2

11	**23**	**36**	**42**	**58**	**65**	**74**	**94**	**99**	**87**
								I=8	L=9

K[I] is greater than all elements before it. Hence No Change

11	**23**	**36**	**42**	**58**	**65**	**74**	**94**	**99**	**87**
									I, L=9

K[I] < K[7] → Insert K[I] at 7

Sorted List:

11	**23**	**36**	**42**	**58**	**65**	**74**	**87**	**94**	**99**

Advantages:

- Sorts the list faster when the list has less number of elements.
- Efficient in cases where a new element has to be inserted into a sorted list.

Disadvantages:

- Very slow for large values of n.
- Poor performance if the list is in almost reverse order.

4.7.2 QUICK SORT

Quick sort is a very popular sorting method. The name comes from the fact that, in general, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small lists than one larger one. The basic strategy of quick sort is to divide and conquer. Quick sort is also known as ***partition exchange sort.***

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and moves an item a great distance in one move.

Principle: A pivotal item near the middle of the list is chosen, and then items on either side are moved so that the data items on one side of the pivot element are smaller than the pivot element, whereas those on the other side are larger. The middle or the pivot element is now in its correct position. This procedure is then applied recursively to the 2 parts of the list, on either side of the pivot element, until the whole list is sorted.

Algorithm:

ALGORITHM QUICKSORT(K, Lower, Upper)

```
// K is the array containing the list of data items
// Lower is the lower bound of the array
// Upper is the upper bound of the array
If (Lower < Upper) Then
BEGIN
        I← Lower
        J ← Upper
        pivot←K[Lower]
        If  (lower < Upper)
        then
        While (I < J)
        Begin
                While (K[I] <= pivot)
                        I ← I + 1
                End While
                While (K[J] > pivot)
                        J ← J – 1
                End While
                If (I < J)Then
                        K[I] ↔ K[J]

                End If
        End While
K[J] ↔ K[Lower]
```

```
QUICKSORT(K, Lower, J – 1)
QUICKSORT(K, J + 1, Upper)
End If
End QUICKSORT
```

In Quick sort algorithm, *Lower* points to the first element in the list and the *Upper* points to the last element in the list. Now I is made to point to the next location of *Lower* and J is made to point to the *Upper*. K[Lower] is considered as the *pivot* element and at the end of the pass, the correct position of the *pivot* element will be decided. Keep on incrementing I and stop when K[I] > Key. When I stops, start decrementing J and stop when K[J] < Key. Now check if I < J. If so, swap K[I] and K[J] and continue moving I and J in the same way. When I meets J the control comes out of the loop and K[J] and K[Lower] are swapped. Now the element at position J is at correct position and hence split the list into two partitions: (K{Lower] to K[J-1] and K[J+1] to K[Upper]). Apply the Quick sort algorithm recursively on these individual lists. Finally, a sorted list is obtained.

Example:

N = 10 → Number of elements in the list

U → Upper

L → Lower

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=0 U, J=9

Initially I=L+1 and J=U, Key=K[L]=42 is the pivot element.

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=2 J=7 U=9

K[2] > Key hence I stops at 2. K[7] < Key hence J stops at 7

Since I < J → Swap K[2] and A[7]

42	**23**	**36**	**11**	**65**	**58**	**94**	**74**	**99**	**87**

L=0 J=3 I=4 U=9

K[4] > Key hence I stops at 4. K[3] < Key hence J stops at 3

Since I > J → Swap K[3] and K[0]. Thus 42 go to correct position.

The list is partitioned into two lists as shown. The same process is applied to these lists individually as shown.

← List 1 → ← List 2 →

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=0, I=1 J,U=2

(applying quicksort to list 1)

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=0, I=1 U=2 J=0 Since I>0 K[L] &K[J] gets swapped i.e., K[0] gets swapped with same element because L,J=0

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=4 J=5 I=6 U=9

(applying quicksort to list 2)

(after swapping 58 & 65)

11	**23**	**36**	**42**	**58**	**65**	**94**	**74**	**99**	**87**

L=6 I=8 U, J=9

11	**23**	**36**	**42**	**58**	**65**	**94**	**74**	**87**	**99**

L=6 J=8 U, I=9

11	**23**	**36**	**42**	**58**	**65**	**87**	**74**	**94**	**99**

L=6 U, I, J=7

Sorted List:

11	**23**	**36**	**42**	**58**	**65**	**74**	**87**	**94**	**99**

Advantages:

1. Faster than any other commonly used sorting algorithms.
2. It has a best average case behavior.

Disadvantages:

1. As it uses recursion, stack space consumption is high.

5.7.3 MERGE SORT

Principle: The given list is divided into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file. Given a sequence of N elements K[0],K[1] ….K[N-1], the general idea is to imagine them split into various subtables of size is equal to 1. So each set will have a individually sorted items with it, then the resulting sorted sequences are merged to produce a single sorted sequence of N elements. Thus this sorting method follows Divide and Conquer strategy. The problem gets divided into various subproblems and by providing the solutions to the subproblems the solution for the original problem will be provided.

Algorithm:

```
ALGORITHM MERGE(K, low, mid, high)
// K is the array containing the list of data items
// Low is the lower bound of the collection
//high is the upper bound of the collection
//mid is the upper bound for the first collection
I ← low, J ← mid+1, L ← 0
While (I ≤ mid) and (J ≤ high)
    If (K[I] < K[J]) Then
            Temp[L] ← K[I]
            I ← I + 1
            L ← L+1
    Else
            Temp[L] ← K[J]
            J ← J + 1
            L ← L + 1
    End If
End While

If (I > mid) Then
    While (J ≤ high)
            Temp[L] ← K[J]
            J ← J + 1
            L ← L + 1
    End While
Else
    While (I ≤ mid)
            Temp[L] ← K[I]
            L ← L + 1
            I ← I + 1
    End While
End If
Repeat for m = 0 to L step 1
    K[Low+m] ← Temp[m]
End Repeat
End MERGE
```

ALGORITHM MERGESORT(A, low, high)

```
// K is the array containing the list of data items
If (low < high) Then
    mid ← (low + high)/2
    MERGESORT(low, mid)
    MERGESORT(mid + 1, high)
    MERGE(low, mid, high)
End If
End MERGESORT
```

The first algorithm MERGE can be applied on two sorted lists to merge them. Initially, the index variable I points to low and J points to mid + 1. K[I] is compared with K[J] and if K[I] found to be lesser than K[J] then K[I] is stored in a temporary array and I is incremented otherwise K[J] is stored in the temporary array and J is incremented. This comparison is continued till either I crosses mid or J crosses high. If I crosses the mid first then that implies that all the elements in first list is accommodated in the temporary array and hence the remaining elements in the second list can be put into the temporary array as it is. If J crosses the high first then the remaining elements of first list is put as it is in the temporary array. After this process we get a single sorted list. Since this method merges 2 lists at a time, this is called 2-way merge sort.

In the MERGESORT algorithm, the given unsorted list is first split into N number of lists, each list consisting of only 1 element. Then the MERGE algorithm is applied for first 2 lists to get a single sorted list. Then the same thing is done on the next two lists and so on. This process is continued till a single sorted list is obtained.

Example:

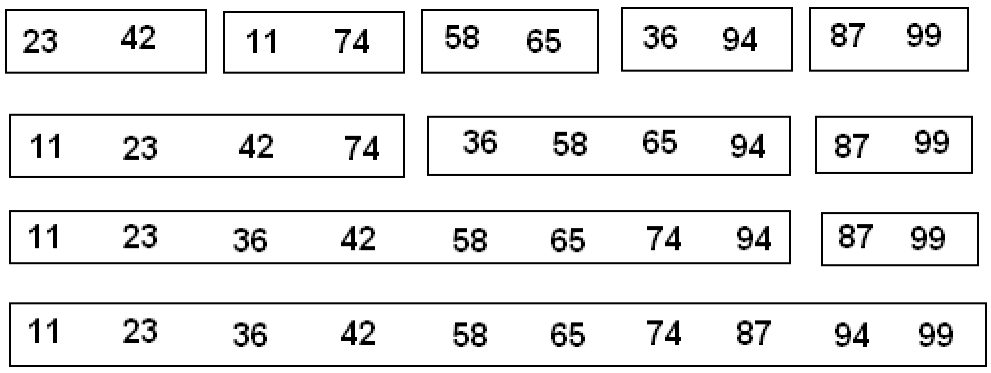
Let L → low, M→ mid, H → high

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**
U			M					H	

In each pass the mid value is calculated and based on that the list is split into two. This is done recursively and at last N number of lists each having only one element is produced as shown.

42	23	74	11	65	58	94	36	99	87

Now merging operation is called on first two lists to produce a single sorted list, then the same thing is done on the next two lists and so on. Finally a single sorted list is obtained.



5.7.4 HEAP SORT

Heap: A Heap is a compete binary tree with the property that the value at each node is at least as large as the values of its children (if they exist). If the value at the parent node is larger than the values on its children then it is called a **Max heap** and if the value at the parent node is smaller than the values on its children then it is called the **Min heap**.

Heap Sort is the sorting technique based on the interpretation of the given sequence of elements as a binary tree. For interpretation the principle given below has to be used.

- If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.**
- To check whether the right child exists or not, use the condition **R ≤ N.** If true, Right child exists otherwise not.
- The last node of the tree is **N/2.** After this position tree has only leaves.

Principle: The Max heap has the greatest element in the root. Hence the element in the root node is pushed to the last position in the array and the remaining elements are converted into a max heap. The root node of this new max heap will be the second largest element and hence pushed to the last but one position in the array. This process is repeated till all the elements get sorted.

HEAPSORT ALGORITHM:

```
FUNCTION  HEAPSORT()
BEGIN
        CALL  BUILDHEAP(A)
        FOR I=HEAPSIZE DOWN TO 2
         DO
        (*SWAP BETWEEN A[1] AND A[I]*)
        A[1]↔A[I]
```

```
    HEAPSIZE=HEAPSIZE-1
    CALL  HEAPIFY(A,1)
    END FOR
END FUNCTION HEAPSORT


FUNCTION BUILDHEAP(A)
BEGIN
    N=HEAPSIZE
    FOR    I= N/2 DOWN TO 1 STEP -1
    CALL HEAPIFY(A,I)
    END FOR
END BUILDHEAP

FUNCTION    HEAPIFY(A,I)
L=2 *I
R=L+1
IF L<=HEAPSIZE    AND     A[L]>A[I]
THEN
    LARGE=L
ELSE
    LARGE=I
END IF
IF R<=HEAPSIZE AND A[R]>A[LARGE]
THEN
    LARGE=R
END IF
IF  I<>LARGE
THEN
    (*SWAP A[I] AND A[LARGE]*)
    A[I] ↔A[LARGE]
```

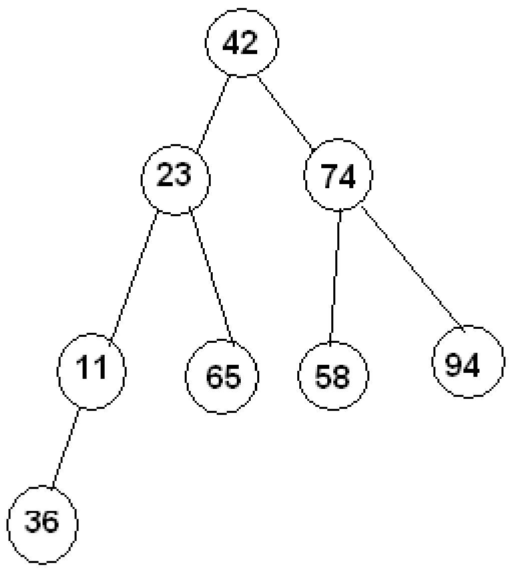
```
        CALL  HEAPIFY(A,LARGE)
END IF
END HEAPIFY
```

Example:

Given a list A with 8 elements:

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**

The given list is first converted into a binary tree as shown.

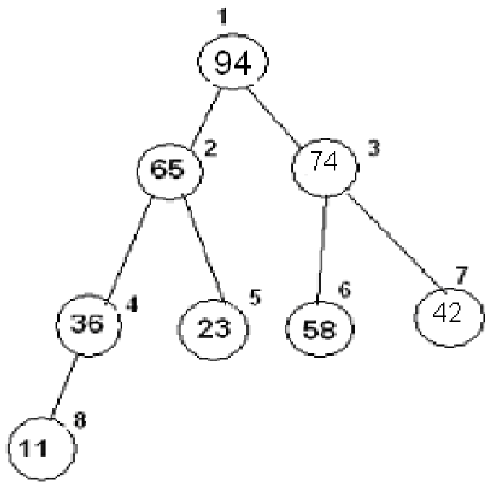


HEAP : (A Complete Binary tree)

Phase 1:

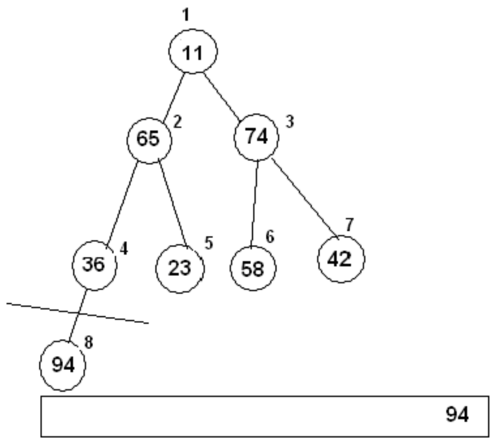
The rearranged tree elements after the first phase is

Max heap is constructed.

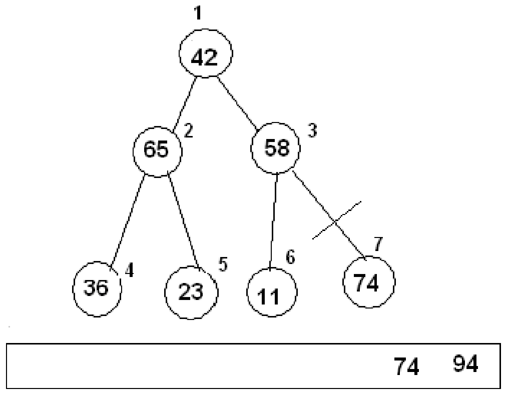


1
94
2
65
3
74
4
36
5
23
6
58
7
42
8
11

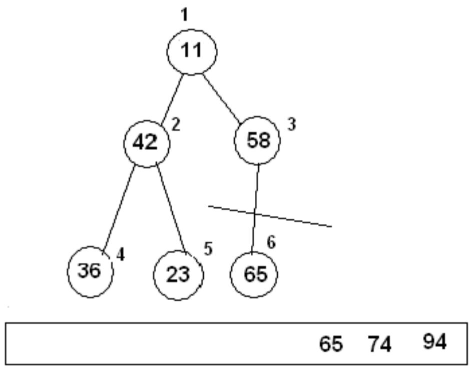
Phase 2:



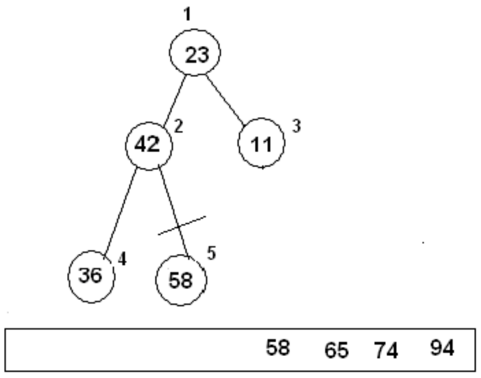
11
65
74
36
23
58
42
94
94



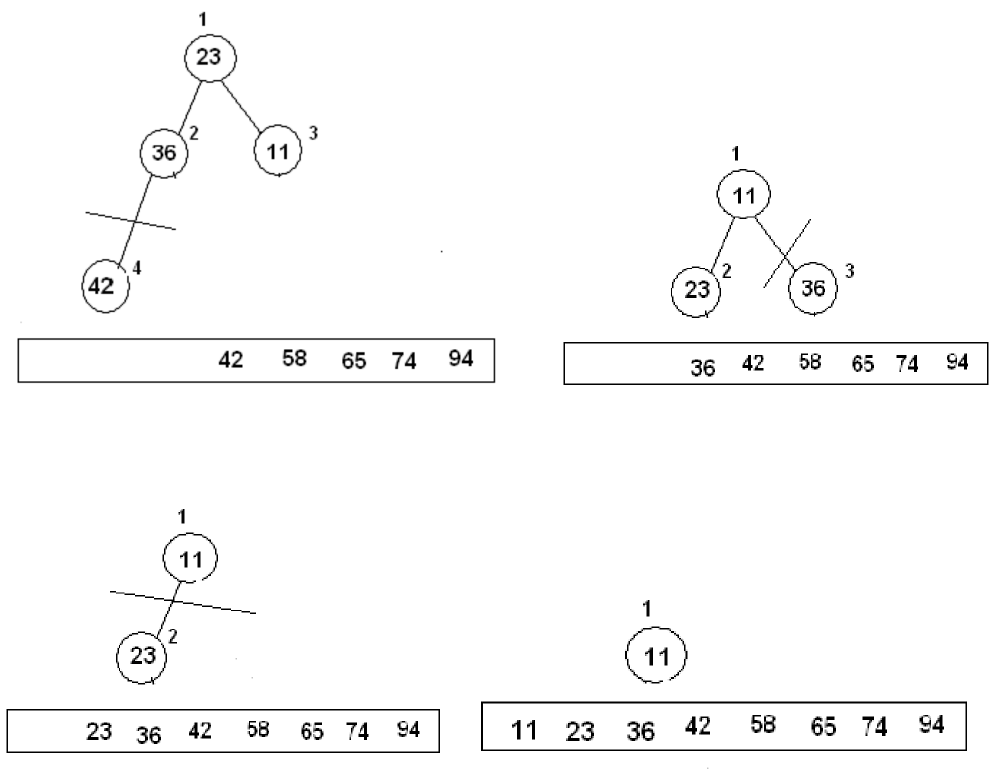
42
65
58
36
23
11
74
74 94



11
42
58
36
23
65
65 74 94



23
42
11
36
58
58 65 74 94



5.7.5 BUBBLE SORT

Bubble sort is a simple sorting algorithm where number of comparisons and number of swaps are more.

<u>Algorithm</u>

```
Function Bubble sort( )
Read n
For I= 0 to n-1
        Read a[I]
End for
//sort
For I=0 to n-2
        For J=I+1 to n-1
                If a[I]>a[J]
                Then
                        T=a[I]
                        a[I]=a[J]
                        a[J]=T
```

```
            End If
        End For J
End For I
 //print the sorted array
For I=0 to n-1
        Write a[I]
End For
End bubble sort
```

Example:

N = 10 → Number of elements in the list

L → Points to last element (Last)

<u>Pass 1</u>

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order → Swap L=9

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order → Swap L=9

23	**42**	**11**	**74**	**65**	**58**	**94**	**36**	**99**	**87**

Out of order → Swap L=9

23	**42**	**11**	**65**	**74**	**58**	**94**	**36**	**99**	**87**

Out of order → Swap L=9

23	**42**	**11**	**65**	**58**	**74**	**94**	**36**	**99**	**87**

Out of order → Swap L=9

23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**99**	**87**

Out of order → *Swap* L=9

Pass 2

23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**87**	**99**

Out of order → *Swap* L=8

23	**11**	**42**	**65**	**58**	**74**	**36**	**94**	**87**	**99**

Out of order → *Swap* L=8

23	**11**	**42**	**58**	**65**	**74**	**36**	**94**	**87**	**99**

Out of order → *Swap* L=8

23	**11**	**42**	**58**	**65**	**36**	**74**	**94**	**87**	**99**

Out of order → *Swap* L=8

Pass 3

23	**11**	**42**	**58**	**65**	**36**	**74**	**87**	**94**	**99**

Out of order → *Swap* L=7

23	**11**	**42**	**58**	**65**	**36**	**74**	**87**	**94**	**99**

Out of order → *Swap* L=7

Pass 4

23	**11**	**42**	**58**	**36**	**65**	**74**	**87**	**94**	**99**

Out of order → *Swap* L=6

11	**23**	**42**	**58**	**36**	**65**	**74**	**87**	**94**	**99**

Out of order → *Swap* L=6

<u>Pass 5</u>

11	**23**	**42**	**36**	**58**	**65**	**74**	**87**	**94**	**99**

Out of order → *Swap* L=5

<u>Pass 6</u>

Adjacent numbers are compared up to L=4. But no swapping takes place. As there was no swapping taken place in this pass, the procedure comes to an end and we get a sorted list:

11	**23**	**36**	**42**	**58**	**65**	**74**	**87**	**94**	**99**

Advantages:

1. Simple and works well for list with less number of elements.

Disadvantages:

1. Inefficient when the list has large number of elements.

Requires more number of exchanges for every pass.

5.8 Analysis of Sorting Techniques

ALGORITHM	**ALGORITHMIC TECHNIQUE**	**ORDER OF GROWTH**
BUBBLE SORT	BRUTE FORCE TECHNIQUE	$O(n^2)$
INSERTION SORT	INSERTION TECHNIQUE	$O(n^2)$

QUICK SORT	DIVIDE AND CONQUER TECHNIQUE	O(n log n)
MERGE SORT	DIVIDE AND CONQUER TECHNIQUE	O(n log n)
HEAP SORT	TREE SORTING (selection technique)	O(n log n)
SELECTION SORT	SELECTION	$O(n^2)$