INTRODUCTION TO DATA STRUCTURE

A program is said to be efficient when it executes in minimum time and with minimum memory space.

Good program is a program that

- It runs correctly
- It is easy to read and understand
- It is easy to debug and
- It is easy to modify.

DATA STRUCTURE

Definition

A data structure is a particular way of storing and **organizing data either in computer's memory or on the disk storage so that it can be used efficiently**. Data structures are used in almost every program or software system. Common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables etc.

Applications of Data Structures

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial Intelligence

The major data structures used in the Network data model is graphs, Hierarchical data model is trees, and RDBMS is arrays. Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently.

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

When selecting a data structure to solve a problem, the following steps must be performed.

1.Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.

2. Quantify the resource constraints for each operation.

3. Select the data structure that best meets these requirements.

In this approach, there are three concern

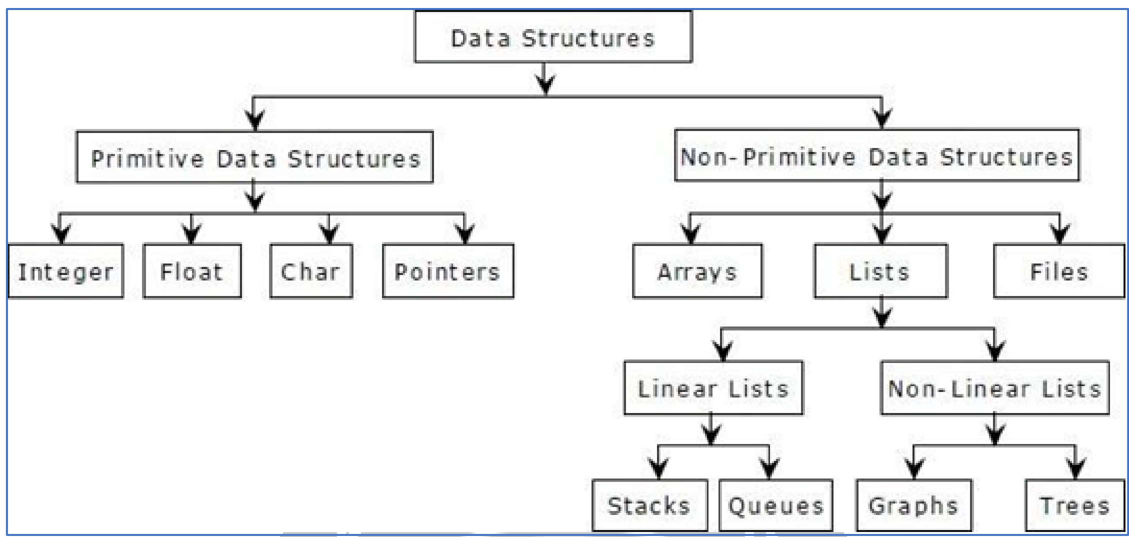
- The first concern is data and the operations that are to be performed on them.
- The second concern about representation of the data.
- The third concern about the implementation of that representation.

ELEMENTARY DATA STRUCTURE ORGANIZATION

Data structures are building blocks of a program. The term data means a value or set of values. It specifies either the value of a variable or a constant. A record is a collection of data items. A file is a collection of related records. Each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a primary key, and the values K1, K2 ... in such field are called keys or key values.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally categorized into two classes: primitive and non-primitive data structures.



- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean.
- Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non- linear data structures.

LINEAR AND NON-LINEAR STRUCTURES

LINEAR DATA STRUCTURES

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

NON-LINEAR DATA STRUCTURES

If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

OPERATIONS ON DATA STRUCTURES

Traversal: Visit every part of the data structure.

Search: Traversal through the data structure for a given element.

Insertion: Adding new elements to the data structure.

Deletion: Removing an element from the data structure.

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

Merging: Combining two similar data structures into one.

ABSTRACT DATA TYPE

An abstract data type (ADT) is the way we look at a data structure, **focusing on what it does and ignoring how it does its job.** The abstract data type is a triple of D-set of Domains, F-Set of functions, A-Axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

ADT = Type + Function Names + Behavior of each function

Examples:

- Stacks
- Queues
- Linked List

ADT Operations

- While modeling the problems the necessary details are separated out from the unnecessary details. This process of modeling the problem is called abstraction.
- The model defines an abstract view to the problem. It focuses only on problem related stuff and that you try to define properties of the problem.

These properties include

- The data which are affected

- The operations which are identified.

Abstract data type operations are

Create: create the database.

Display: displaying all the elements of the data structure.

Insertion: elements can be inserted at any desired position.

Deletion: desired element can be deleted from the data structure.

Modification: any desired element can be deleted from the data structure.

Advantage of using ADTs

- It is reusable, robust
- It can be re-used at several places and it reduces coding efforts
- Encapsulation ensures that data cannot be corrupted
- The Working of various integrated operation cannot be tampered with by the application program
- ADT ensures a robust data structure

List ADT

List is the collection of elements in sequential order. In memory we can store the list in two ways.

- Sequential Memory Location - Array
- Pointer or links to associate the elements sequential – Linked List.

THE LIST ADT

List is an ordered set of elements. The general form of the list is

A1, A2, A3, , AN

A1 - First element of the list AN - Last element of the list N - Size of the list

If the element at position i is Ai then its successor is Ai+1 and its predecessor is Ai-1.

Various operations performed on List

- create an empty list
- printList () – prints all elements in the list construct a (deep) copy of a list
- find(x) – returns the position of the first occurrence of x
- remove(x) – removes x from the list if present
- insert (x, position) – inserts x into the list at the specified position isEmpty () – returns true if the list has no elements
- makeEmpty () – removes all elements from the list findKth (int k) – returns the element in the specified position

LINKED LIST

Definition

A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.

Reason for Linked List

- Array is a linear collection of data elements in which the elements are stored in consecutive memory locations.
- While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks [10], then the array can store a maximum of 10 data elements but not more than that.
- But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations.
- So, there must be a data structure that removes the restrictions on the maximum number
- of elements and the storage condition to write efficient programs.

Advantages of using linked list

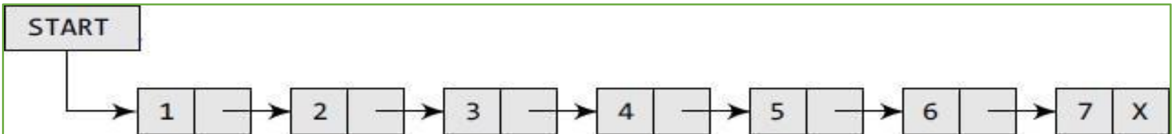
- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.

Basic Terminologies

- A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes.
- Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

OBSERVE OPTIMIZE OUTSPREAD
ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI

- Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.



- In above figure, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.
- The left part of the node which contains data may include a simple data type, an array, or a structure.
- The right part of the node contains a pointer to the next node (or address of the next node in sequence).
- The last node will have no next node connected to it, so it will store a special value called NULL. In above figure, the NULL pointer is represented by X.
- While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.
- Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

Use of START pointer variable

- Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node.
- Using this technique, the individual nodes of the list will form a chain of nodes. If
- START = NULL, then the linked list is empty and contains no nodes.

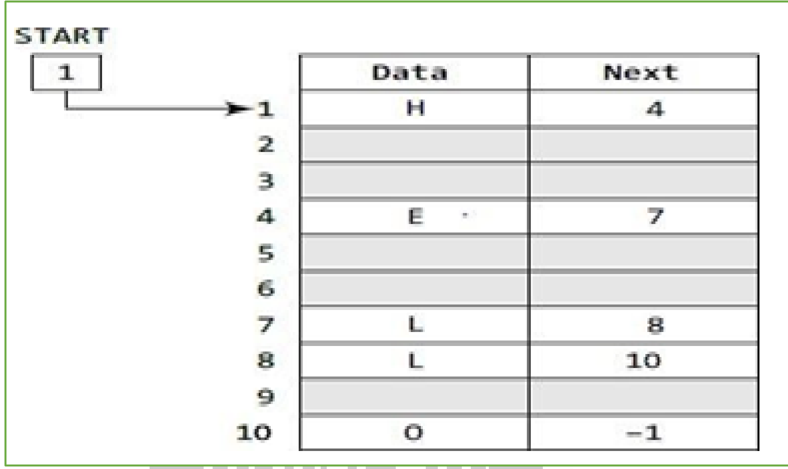
Sample Linked List code

```
struct node
{
int data;
struct node *next;
```

```
};
```

Memory representation of Linked List

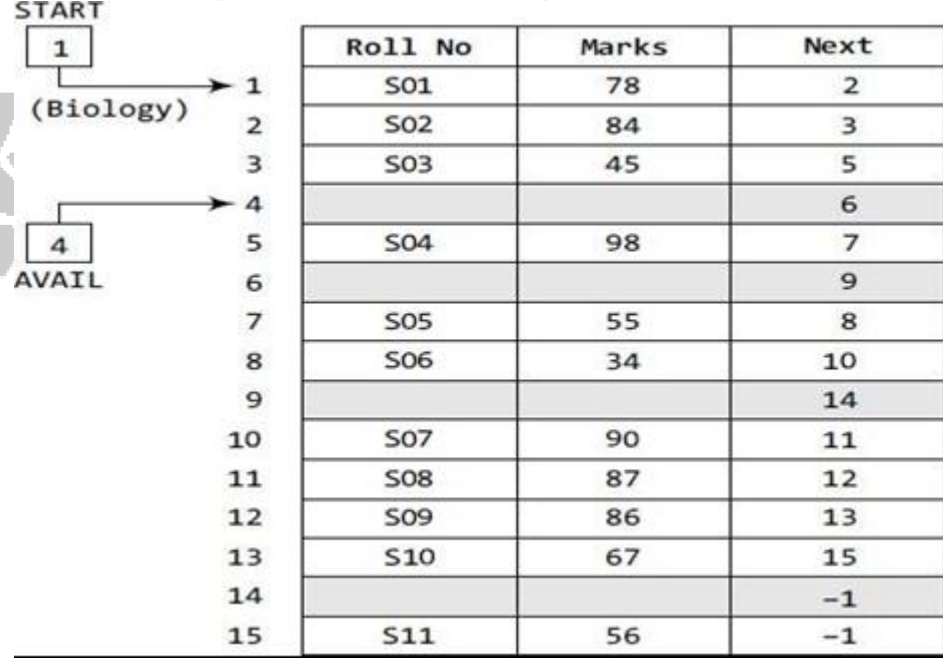
- Let us see how a linked list is maintained in the memory. In order to form a linked list, we need a structure called node which has two fields, DATA and NEXT.
- DATA will store the information part and NEXT will store the address of the next node in sequence.
- Consider the below figure, we can see that the variable START is used to store the address of the first node.
- Here, in this example, START = 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node.
- From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.
- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.
- Note that in the below figure shows a chunk of memory locations which range from 1 to 10. The shaded portion contains data for other applications.
- Remember that the nodes of a linked list need not be in consecutive memory locations. In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.



	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	-1

FREE POOL AND AVAIL POINTER VARIABLE

- The computer maintains a list of all free memory cells. This list of available space is called the free pool.
- We have seen that every linked list has a pointer variable START which stores the address of the first node of the list. Likewise, for the free pool (which is a linked list of all free memory cells), we have a pointer variable AVAIL which stores the address of the first free space.
- Let us revisit the memory representation of the linked list storing all the students' marks in Biology.
- Now, when a new student's record has to be added, the memory address pointed by AVAIL will be taken and used to store the desired information.
- After the insertion, the next available free space's address will be stored in AVAIL.



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			-1
15	S11	56	-1

- For example, in the above Figure when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.
- This was all about inserting a new node in an already existing linked list. Now, we will discuss deleting a node or the entire linked list.
- When we delete a particular node from an existing linked list or delete the entire linked list, the space occupied by it must be given back to the free pool so that the memory can be reused by some other program that needs memory space.
- The operating system does this task of adding the freed memory to the free pool. The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space.
- The operating system scans through all the memory cells and marks those cells that are being used by some program.
- Then it collects all the cells which are not being used and adds their address to the free pool, so that these cells can be reused by other programs. This process is called garbage collection.

Array Implementation of list:

Array: A set of data elements of same data type is called array. Array is a static data structure i.e., the memory should be allocated in advance and the size is fixed. This will waste the memory space when used space is less than the allocated space. An array implementation allows the following operations.

The basic operations are:

a. Creation of a List.

b. Insertion of a data in the List

c. Deletion of a data from the List

d. Searching of a data in the list

Global Declaration:

```
int list[25], index=-1;
```

Note: The initial value of index is -1.

Create Operation:

Procedure

- The list is initially created with a set of elements.
- Get the no. of elements (n) to be added in the list.
- Check n is less than or equal to maximum size. If yes, add the elements to the list.
- Otherwise, give an error message

Program

```
void create()
{
int n,i;
```

```
printf("\nEnter the no.of elements to be added in the list");

s.canf("%d",&n); if(n<=maxsize) for(i=0;i<n;i++)

{

scanf("%d",&list[i]); index++;

}

else

printf("\nThe size is limited. You cannot add data into the list");

}
```

Insert Operation:

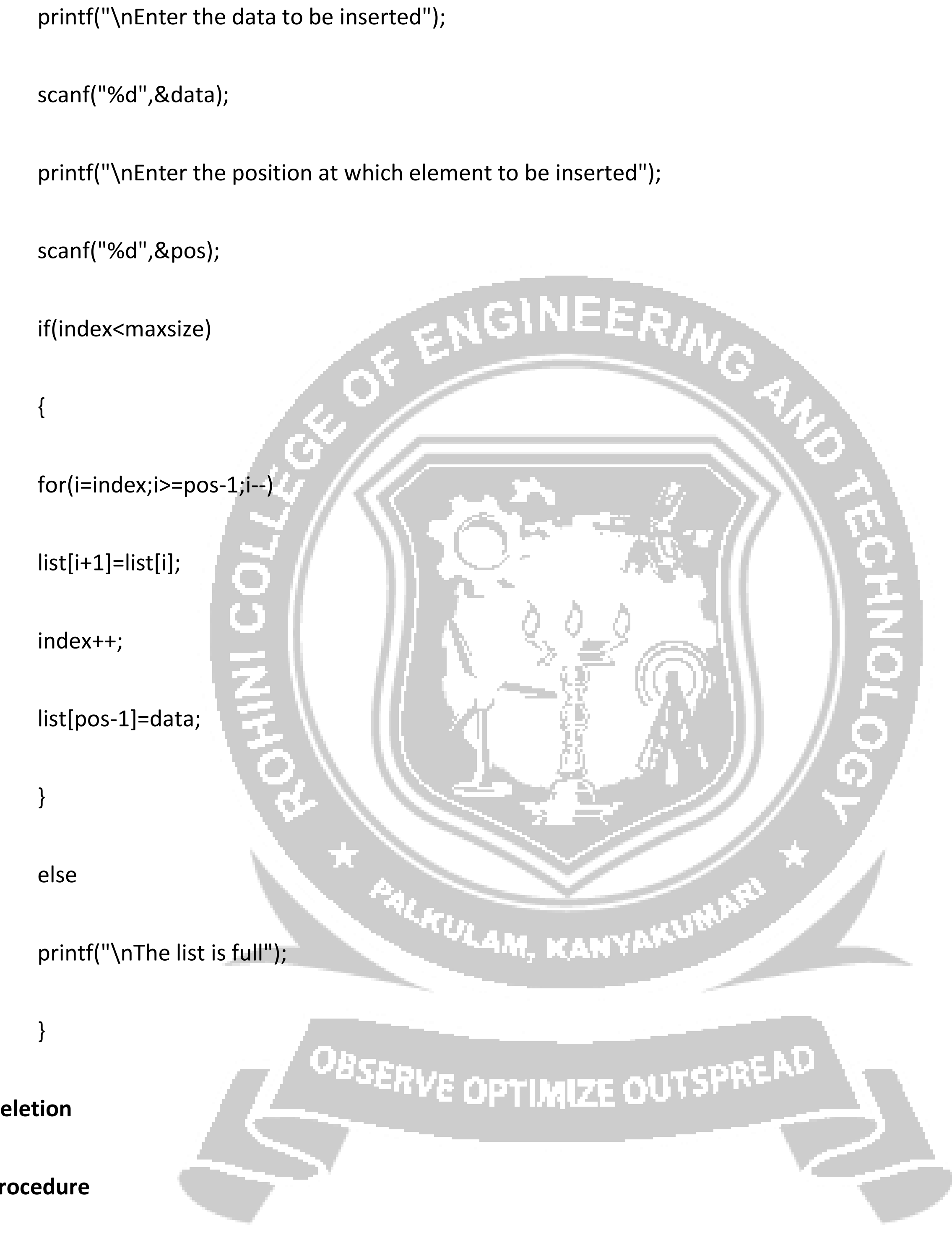
Procedure:

- Get the data element to be inserted.
- Get the position at which element is to be inserted.
- If index is less than or equal to maxsize, then Make that position empty by altering the position of the elements

  in the list.
- Insert the element in the poistion.
- Otherwise, it implies that the list is empty.

Program

```
void insert()

{

int i,data,pos;
```



```
printf("\nEnter the data to be inserted");
scanf("%d",&data);
printf("\nEnter the position at which element to be inserted");
scanf("%d",&pos);
if(index<maxsize)
{
for(i=index;i>=pos-1;i--)
list[i+1]=list[i];
index++;
list[pos-1]=data;
}
else
printf("\nThe list is full");
}
```

Deletion

Procedure

- Get the position of the element to be deleted.
- Alter the position of the elements by performing an assignment operation, list[i-1]=list[i], where i value ranges from position to the last index of the array.

Program:

```
void del()
{
int i,pos;
printf("\nEnter the position of the data to be deleted");
scanf("%d",&pos);
printf("\nThe data deleted is %d",list[pos-1]);
for(i=pos;i<=index;i++)
list[i-1]=list[i];
index--;
}
```

Display

Procedure

- Formulate a loop, where i value ranges from 0 to index (index denotes the index of the last element in the array.
- Display each element in the array.

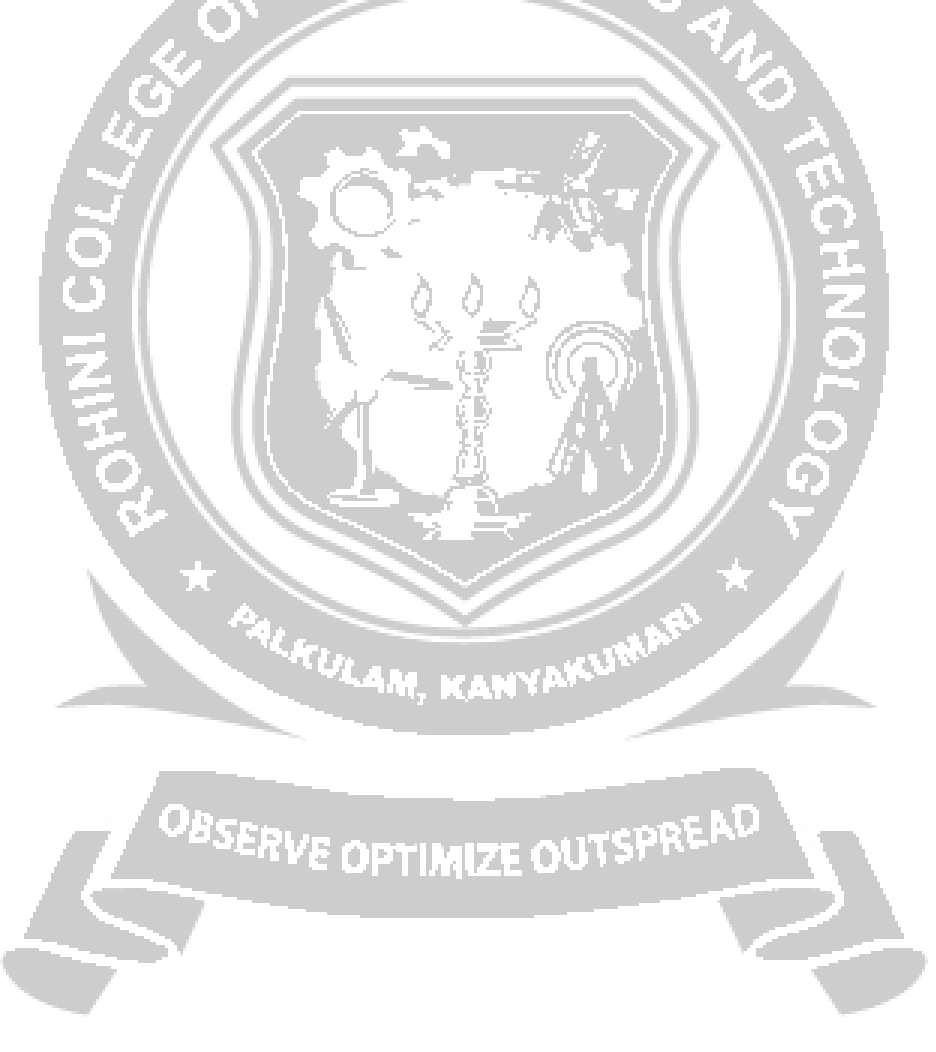
Program

```
void display()
{
```

```
int i; for(i=0;i<=index;i++) printf("\t%d",list[i]);

}
```

Limitation of array implementation

- An array size is fixed at the start of execution and can store only the limited number of elements.
- Insertion and deletion of array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.



Linked list implementation – Singly linked lists

Linked Lists Versus Arrays

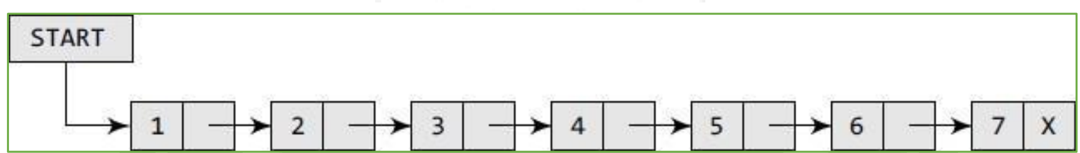
Arrays and linked lists are a linear collection of data elements

Array	Linked Lists
It stores its nodes in consecutive memory locations	It does not store its nodes in consecutive memory locations
It allows random access of data	It does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner
It cannot add any number of elements in the array	It can add any number of elements in the list

SINGLY LINKED LISTS

Definition

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way



Traversing a Linked List

Accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

Algorithm for traversing a linked list

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: EXIT
```

- In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list.
- Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL.
- In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR.
- In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

Algorithm to print the number of nodes in a linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:            SET COUNT = COUNT + 1
Step 5:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

We will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed.

Searching for a Value in a Linked List

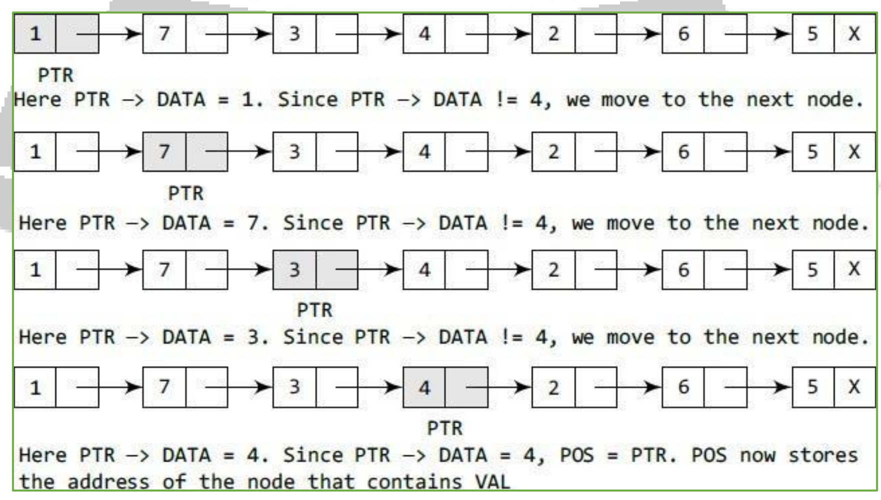
Searching a linked list means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                  SET POS = PTR
                  Go To Step 5
            ELSE
                  SET PTR = PTR -> NEXT
            [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made.
- If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Example: Illustration of Searching algorithm

Consider the linked list shown in Figure. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



Inserting a New Node in a Linked List

How a new node is added into an already existing linked list? We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

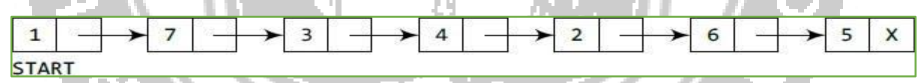
Case 3: The new node is inserted after a given node.

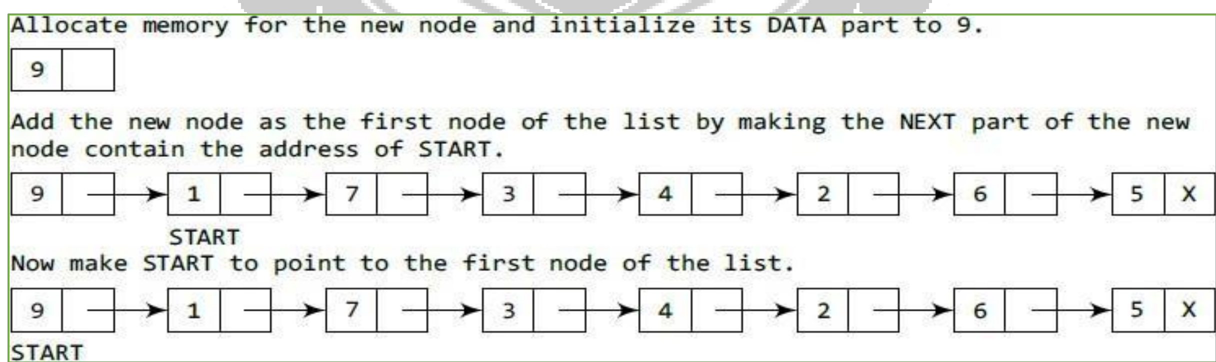
Case 4: The new node is inserted before a given node. OVERFLOW

Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Case 1: Inserting a Node at the Beginning of a Linked List

Add a new node with data 9 and add it as the first node of the list.





Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

- In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.

Note the following two steps:

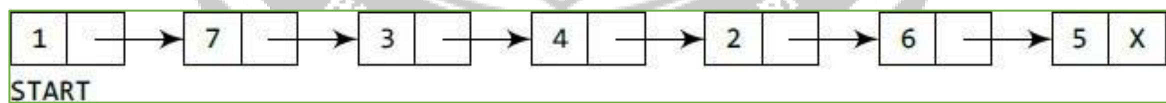
Step 2: SET NEW_NODE = AVAIL

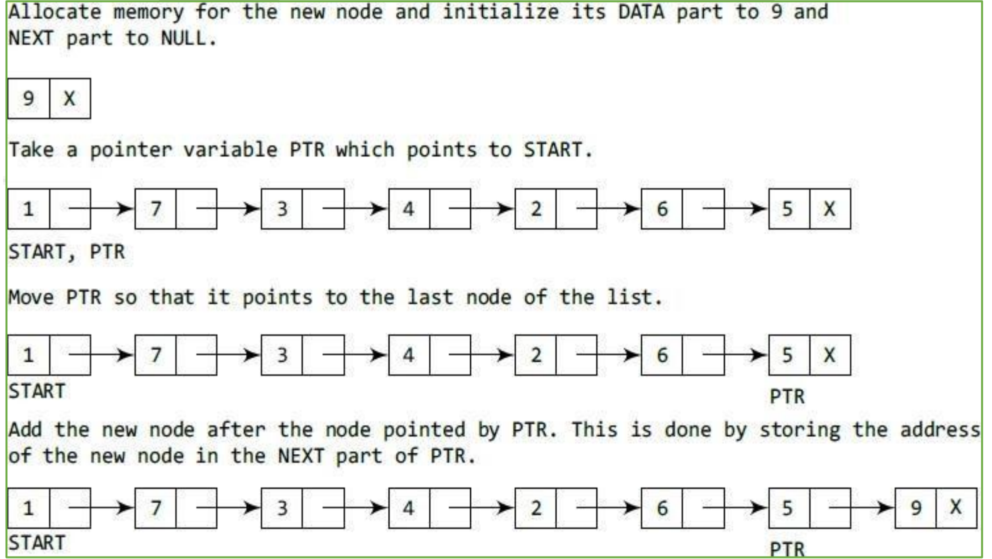
Step 3: SET AVAIL = AVAIL -> NEXT

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.

Case 2: Inserting a Node at the End of a Linked List

Add a new node with data 9 as the last node of the list





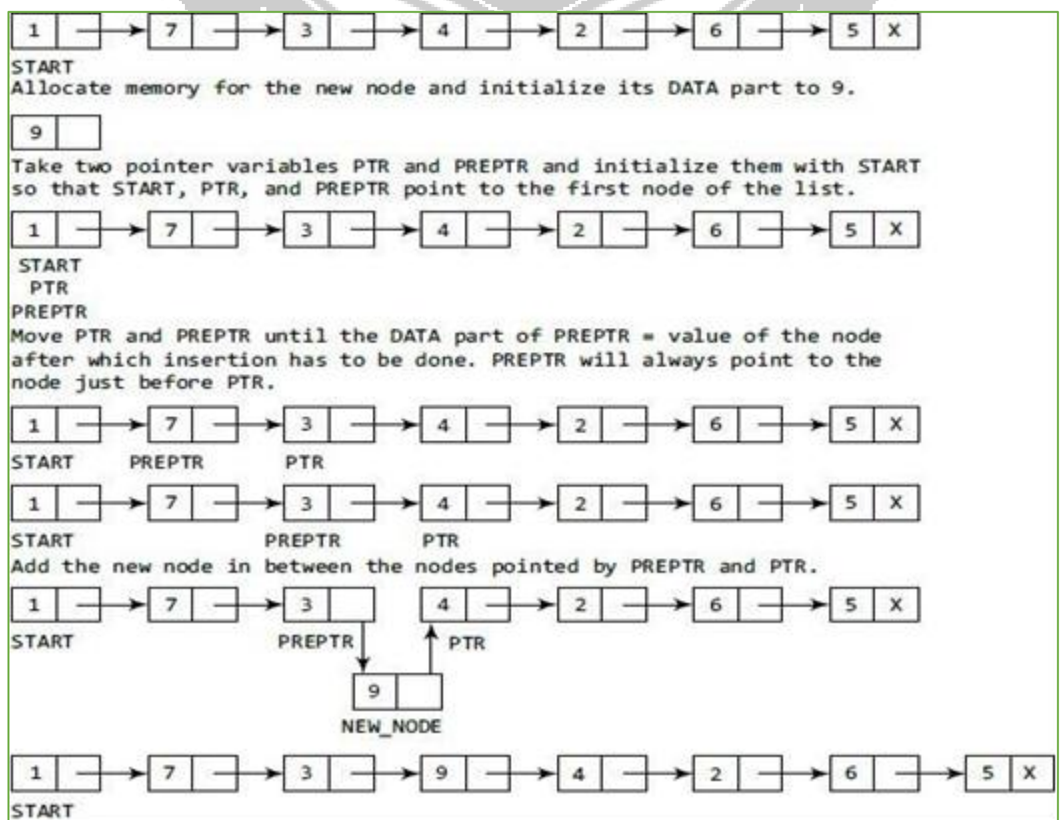
Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

This algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

Case 3: Inserting a Node After a Given Node in a Linked List

Add a new node with value 9 after the node containing data 3.



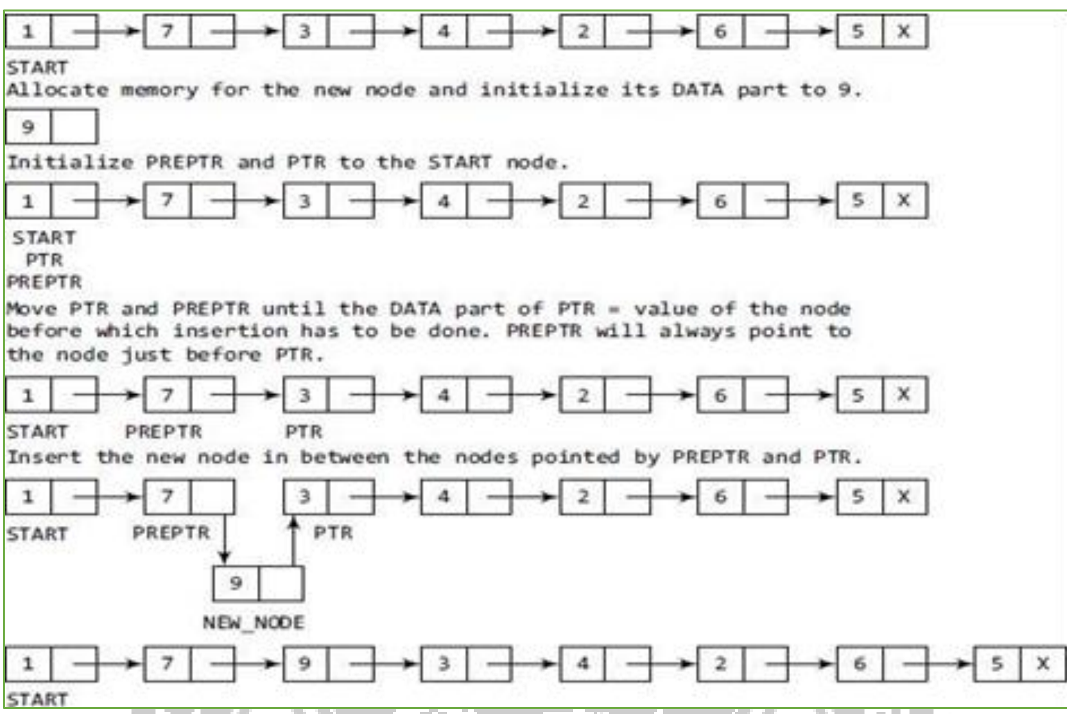
Algorithm to insert a new node after a node that has value NUM

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

- In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR.
- So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

Case 4: Inserting a Node Before a Given Node in a Linked List

Add a new node with value 9 before the node containing 3.



Algorithm to insert a new node before a node that has value NUM

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

- In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted before this node.

- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

Deleting a Node from a Linked List

We will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

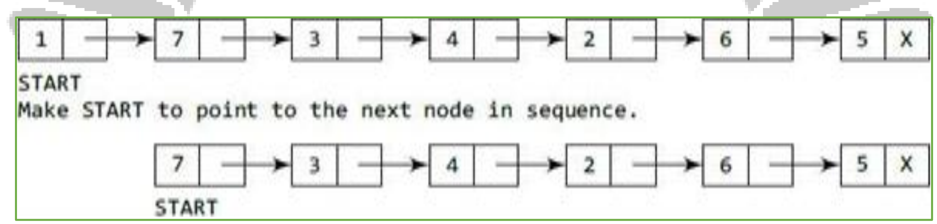
Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

- Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete.
- Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node.
- The memory is returned to the free pool so that it can be used to store other programs and data.
- Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Case 1: The first node is deleted.

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



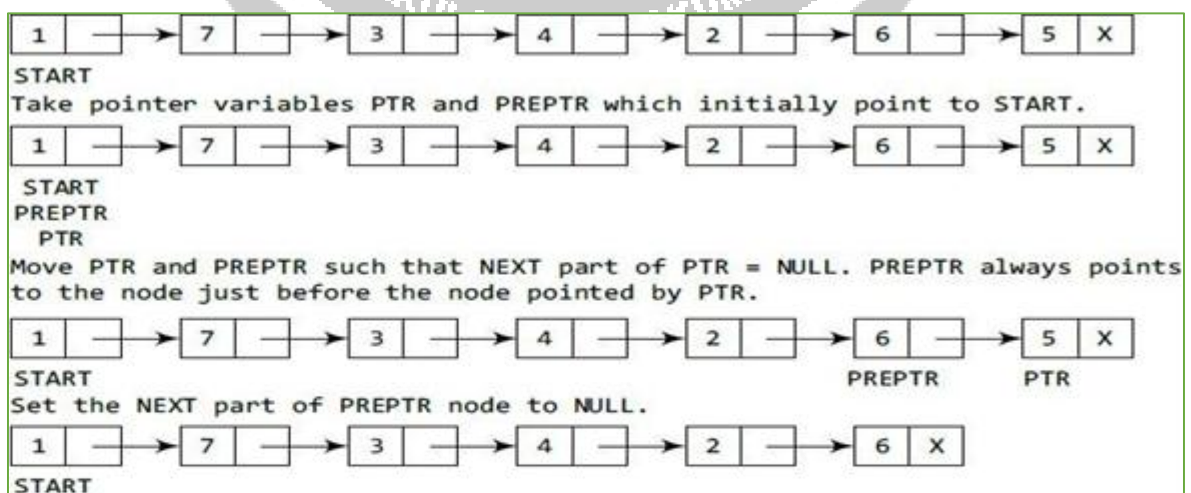
Algorithm to delete the first node

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

- In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Case 2: The last node is deleted.

We want to delete the last node from the linked list, then the following changes will be done in the linked list.



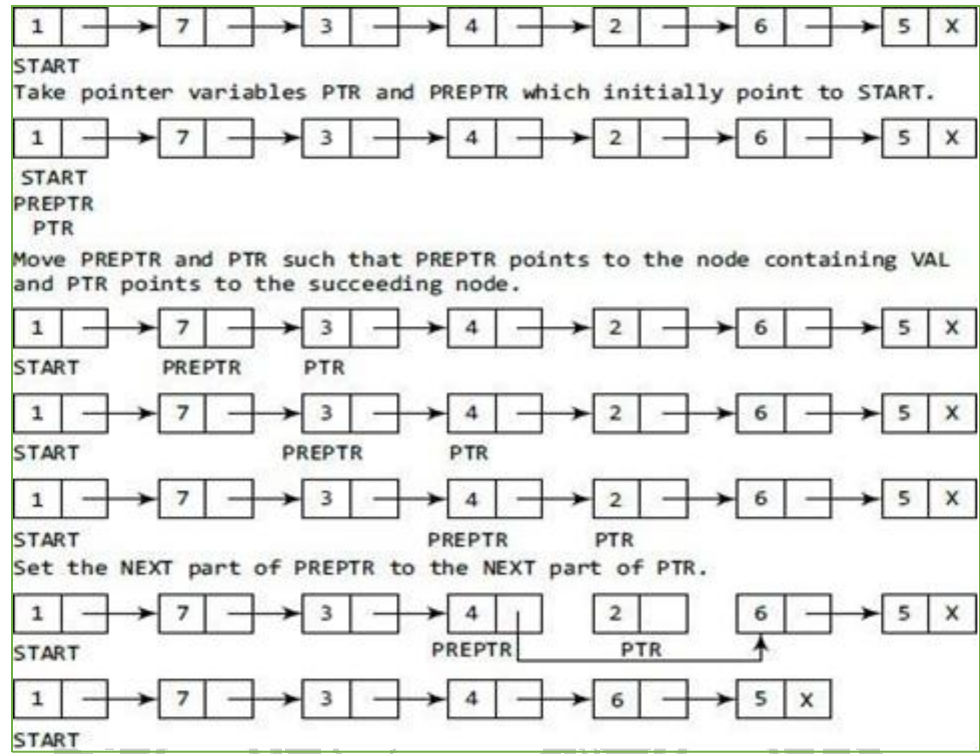
Algorithm to delete the last node

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:       SET PREPTR = PTR
Step 5:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned back to the free pool.

Case 3: The node after a given node is deleted.

We want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



Algorithm to delete the node after a given node

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it.
- The memory of the node succeeding the given node is freed and returned back to the free pool.

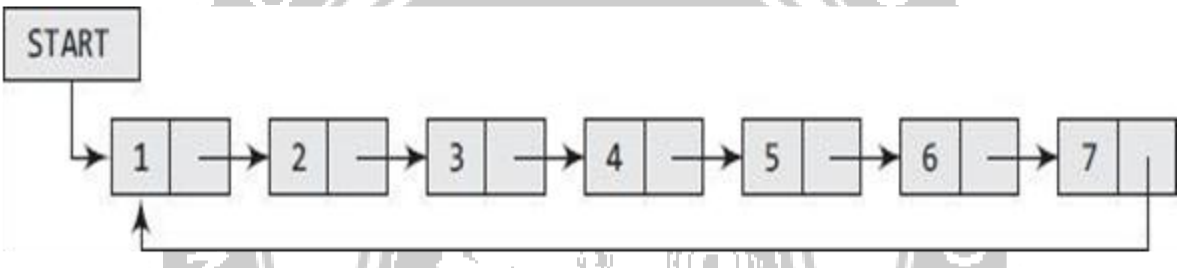
CIRCULAR LINKED LISTs

Definition

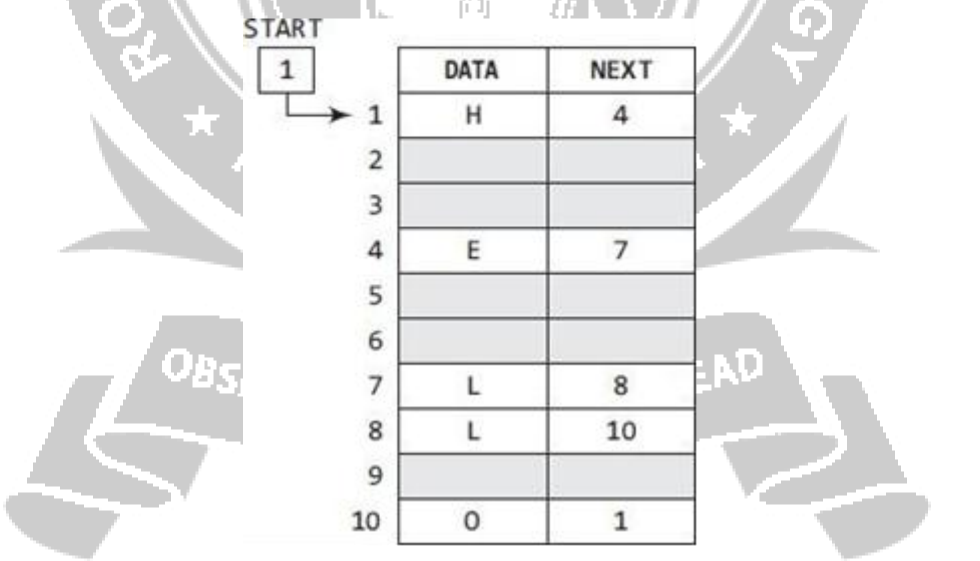
In a circular linked list is similar to singly linked list except that the last node contains a pointer to the first node of the list.

Types

- Circular singly linked list
- Circular doubly linked list.



Memory representation of a circular linked list



START: 1

	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

- We can traverse the list until we find the NEXT entry that contains the address of the first node of the list.
- This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above figure stores characters that when put together form the word HELLO.

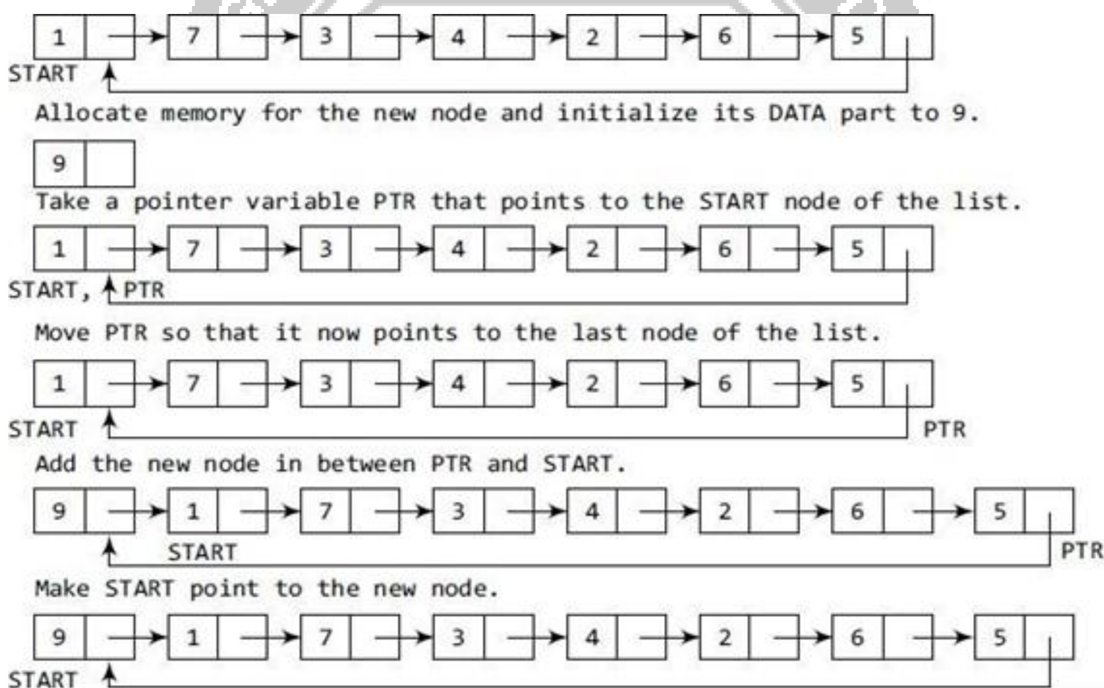
Inserting a New Node in a Circular Linked List

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

Case 1: The new node is inserted at the beginning of the circular linked list.

We want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.



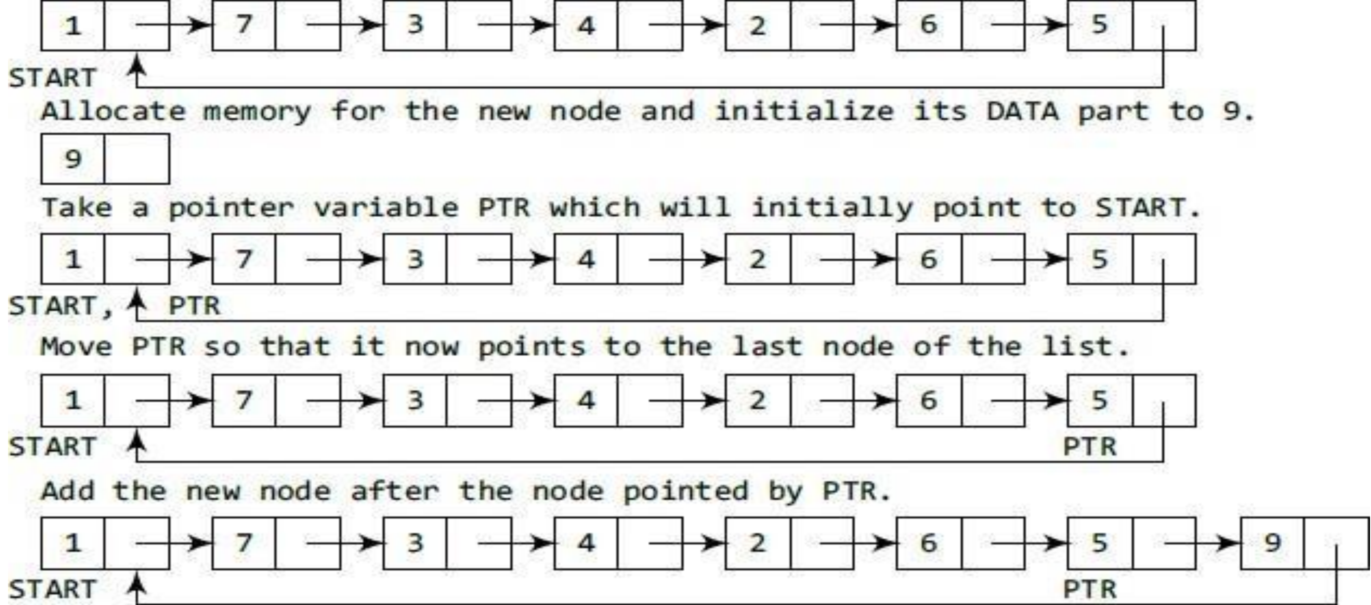
Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:      PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.
- While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list.
- Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

Case 2: The new node is inserted at the end of the circular linked list.

We want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

Deleting a Node from a Circular Linked List

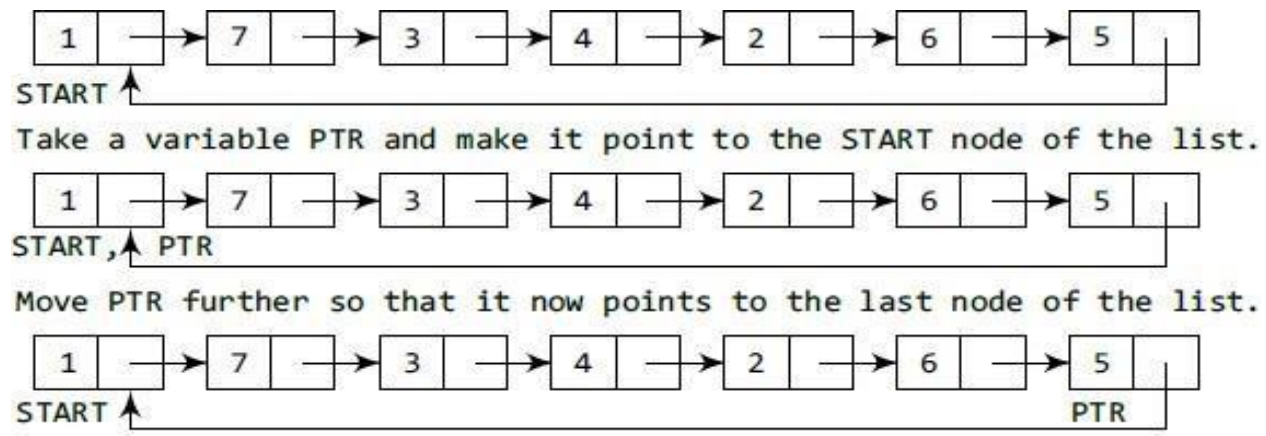
Case 1: The first node is deleted.

Case 2: The last node is deleted.

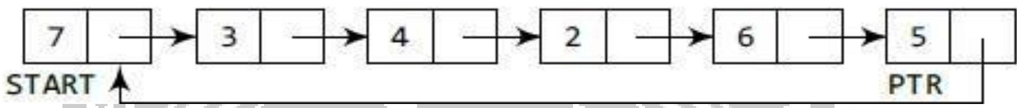
Case 1: The first node is deleted.

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD



The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.



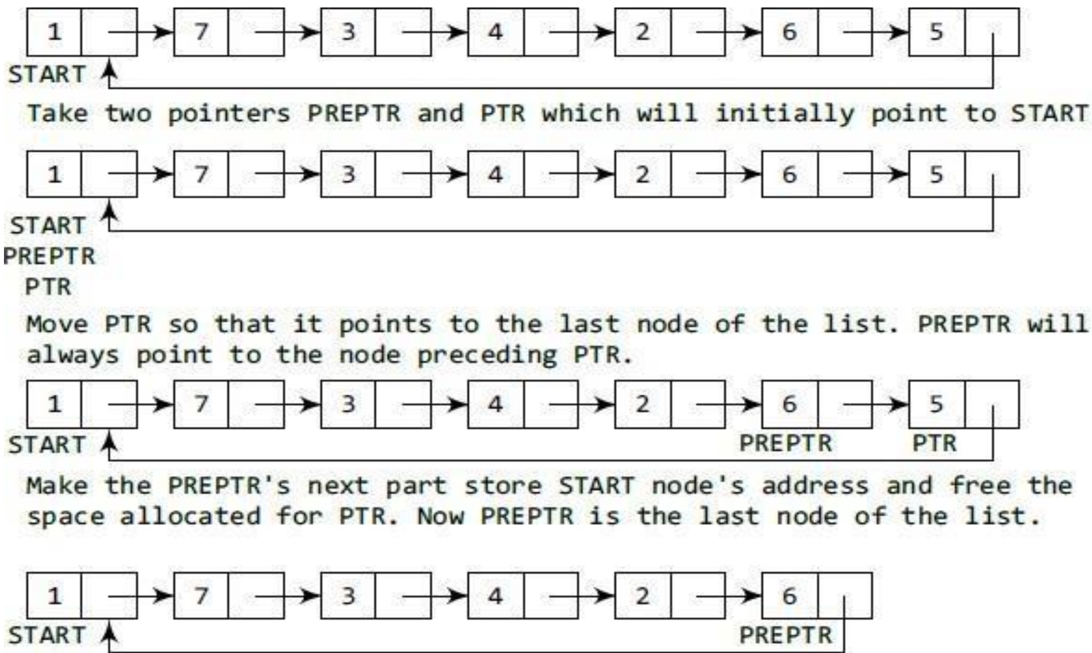
Algorithm to delete the first node

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

- In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list.
- In Step 6, the memory occupied by the first node is freed.

- Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

Case 2: The last node is deleted.



we want to delete the last node from the linked list, then the following changes will be done in the linked list.

Algorithm to delete the last node

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

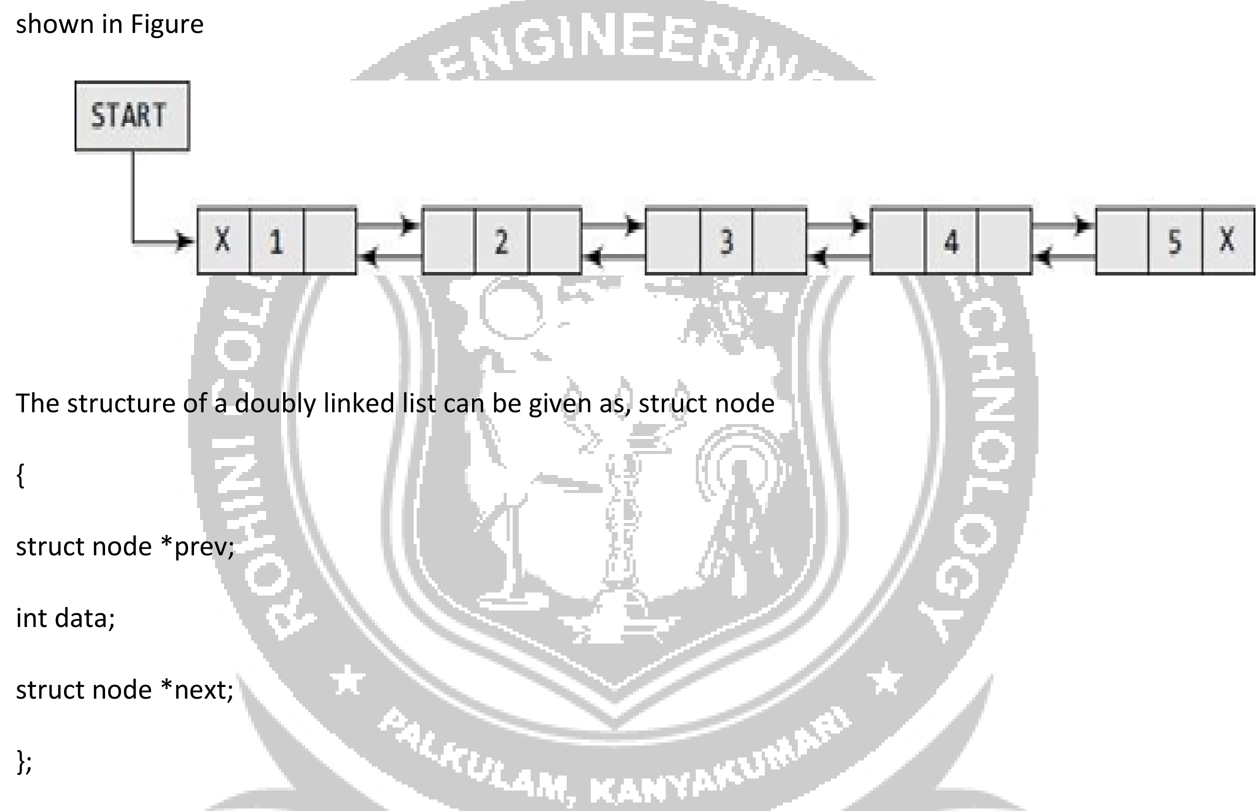
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.

- Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

DOUBLY LINKED LISTS

Definition

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Figure

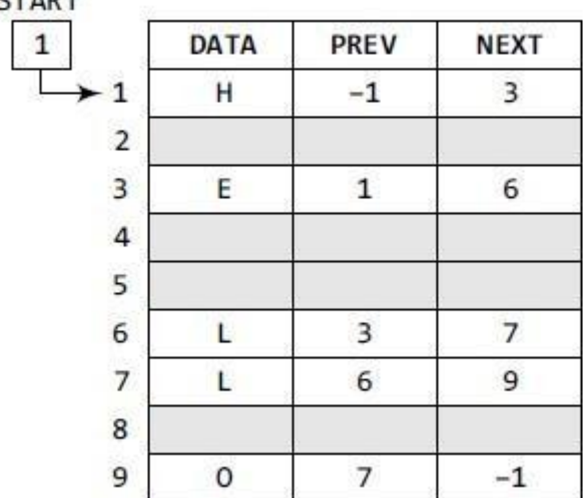


The structure of a doubly linked list can be given as, struct node

```
{
struct node *prev;
int data;
struct node *next;
};
```

- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- Doubly linked list calls for more space per node and more expensive basic operations.
- However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Memory representation of a doubly linked list



START = 1

	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

- Variable START is used to store the address of the first node.
- In this example, START = 1, so the first data is stored at address 1, which is H.
- Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field.
- We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Inserting a New Node in a Doubly Linked List

Case 1: The new node is inserted at the beginning.

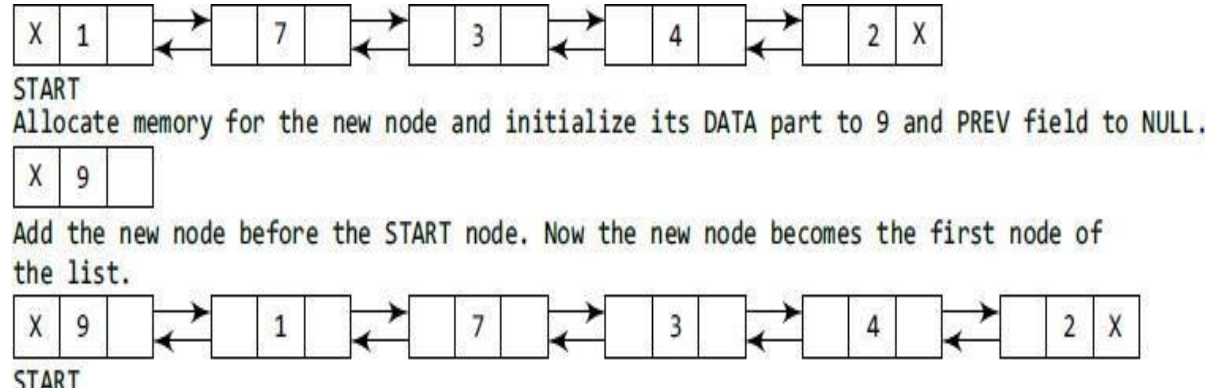
Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

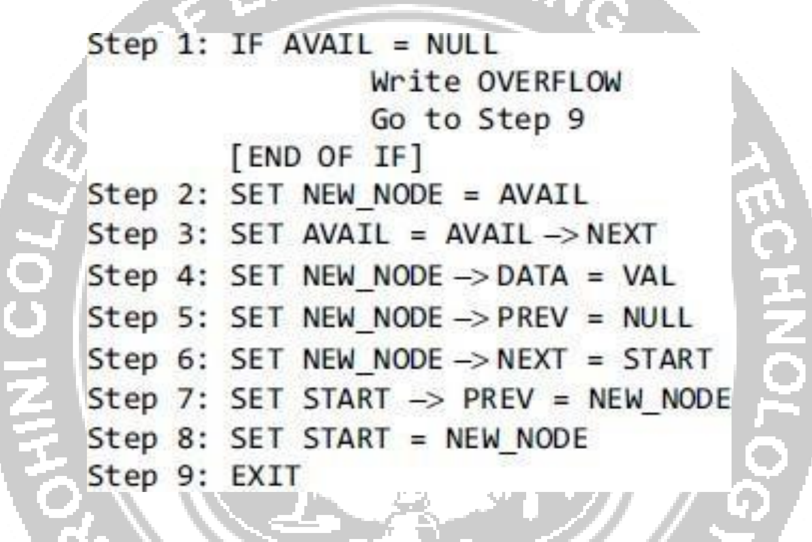
Case 4: The new node is inserted before a given node.

Case 1: The new node is inserted at the beginning.

We want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.



Algorithm to insert a new node at the beginning

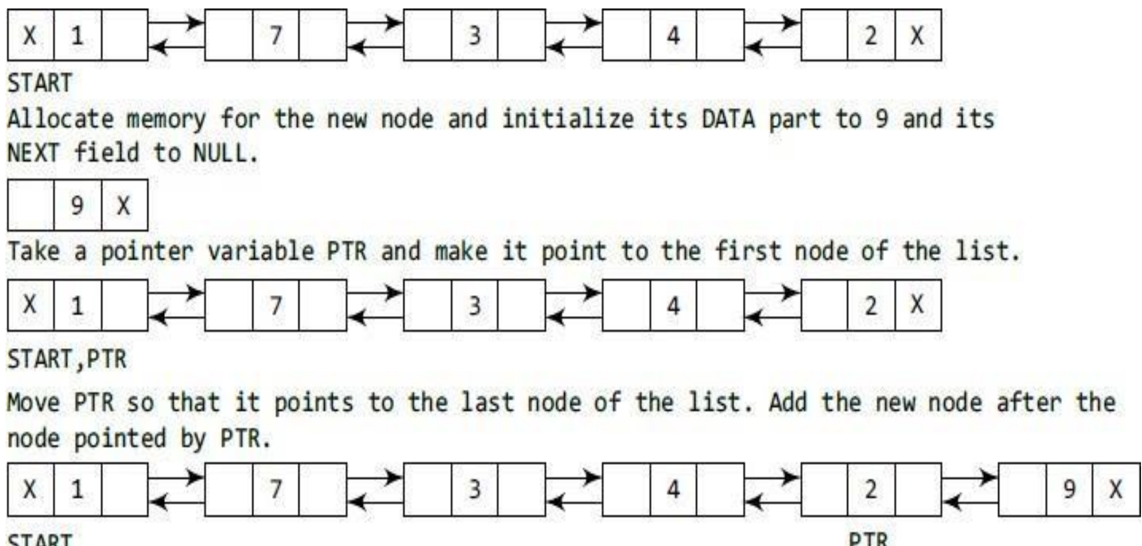


```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

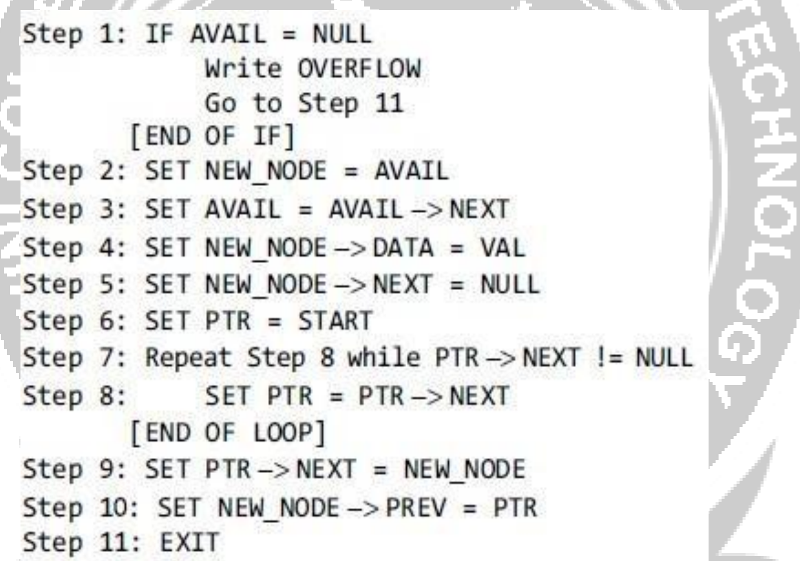
- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

Case 2: The new node is inserted at the end.

We want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



Algorithm to insert a new node at the end

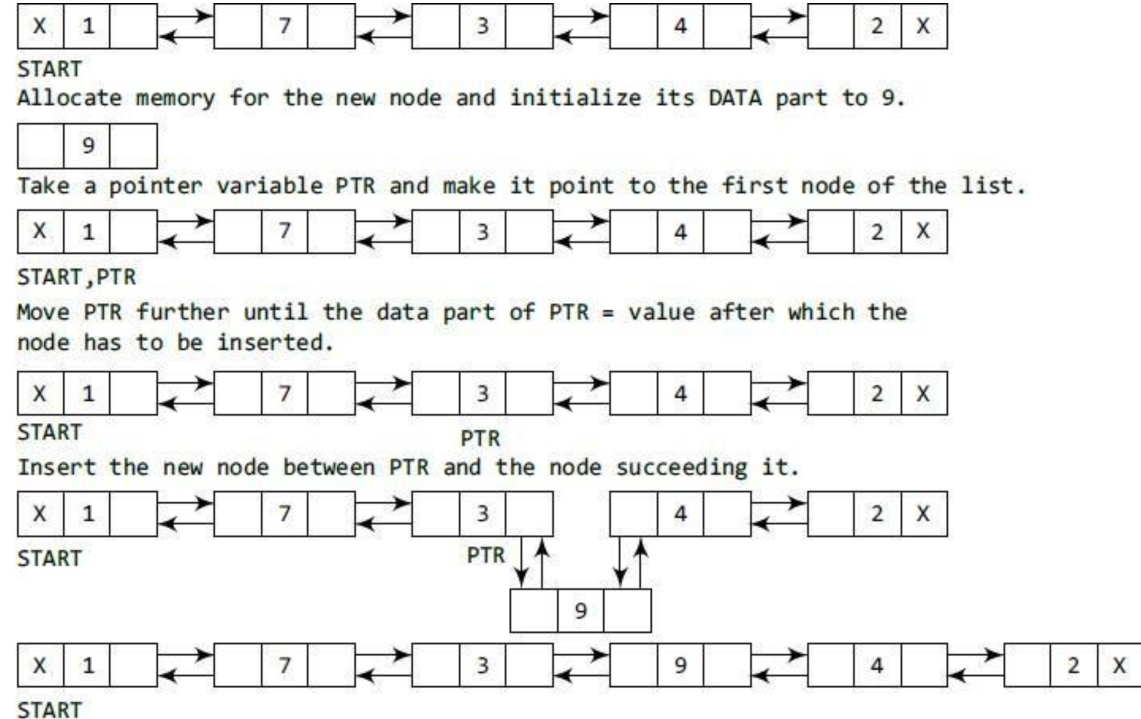


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

- In Step 6, we take a pointer variable PTR and initialize it with START.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Case 3: The new node is inserted after a given node.

We want to add a new node with value 9 after the node containing 3.



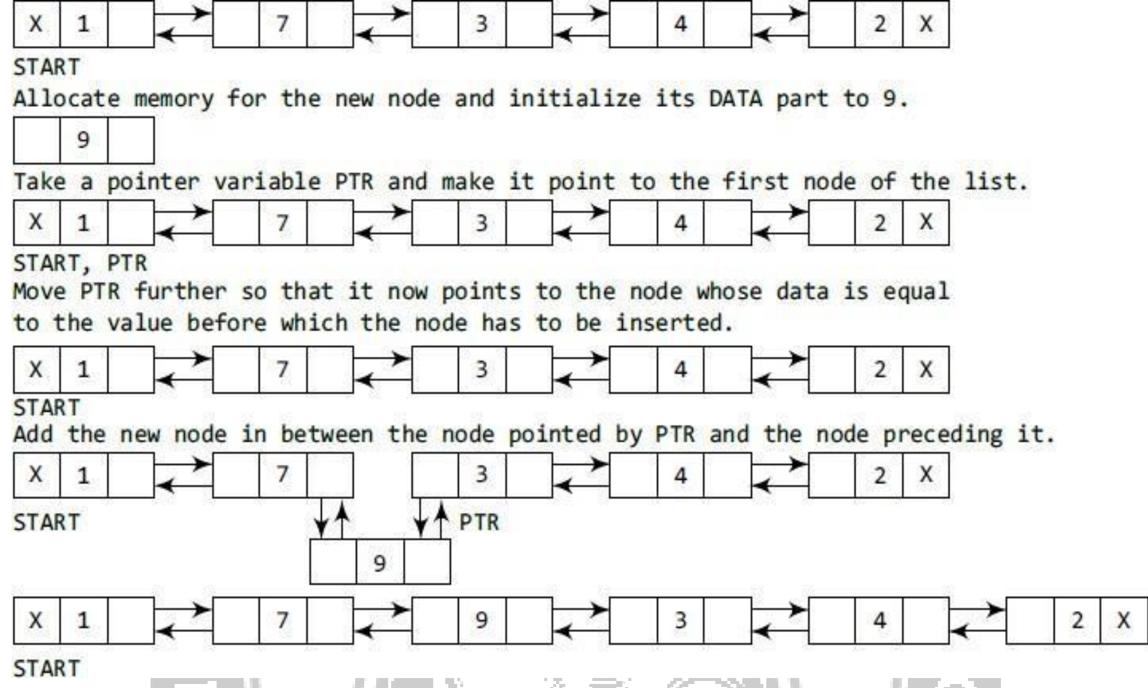
Algorithm to insert a new node after a given node

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

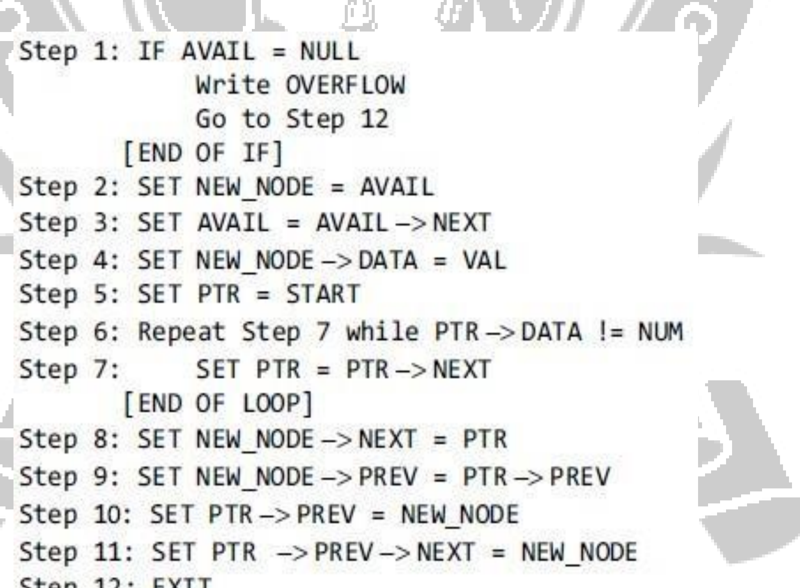
- In Step 5, we take a pointer PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

Case 4: The new node is inserted before a given node.

We want to add a new node with value 9 before the node containing 3.



Algorithm to insert a new node before a given node



```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.

- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

Deleting a Node from a Doubly Linked List

In this section, we will see how a node is deleted from an already existing doubly linked list

Case 1: The first node is deleted.

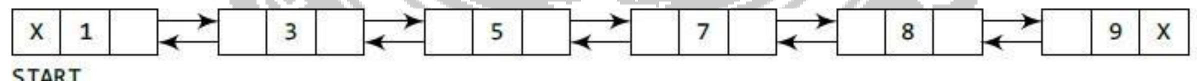
Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 4: The node before a given node is deleted.

Case 1: The first node is deleted.

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list



START

Free the memory occupied by the first node of the list and make the second node of the list as the START node.

START

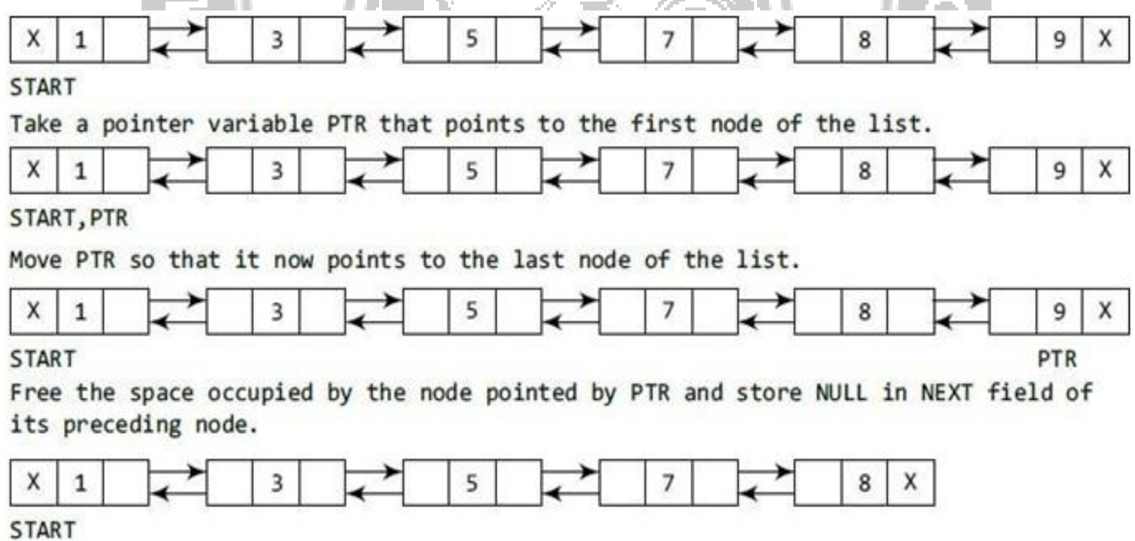
Algorithm to delete the first node

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 6
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

Case 2: The last node is deleted.

We want to delete the last node from the linked list, then the following changes will be done in the linked list.



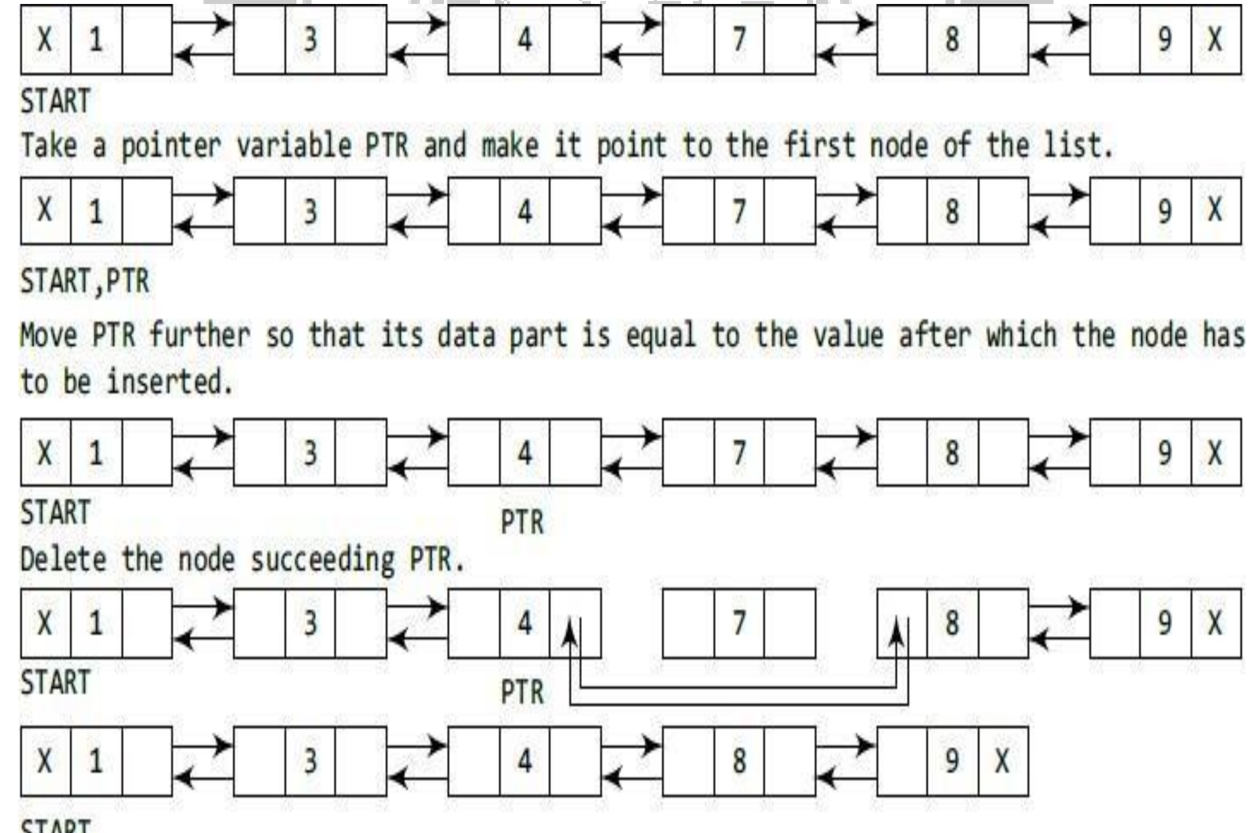
Algorithm to delete the last node

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 7
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Case 3: The node after a given node is deleted.

We want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



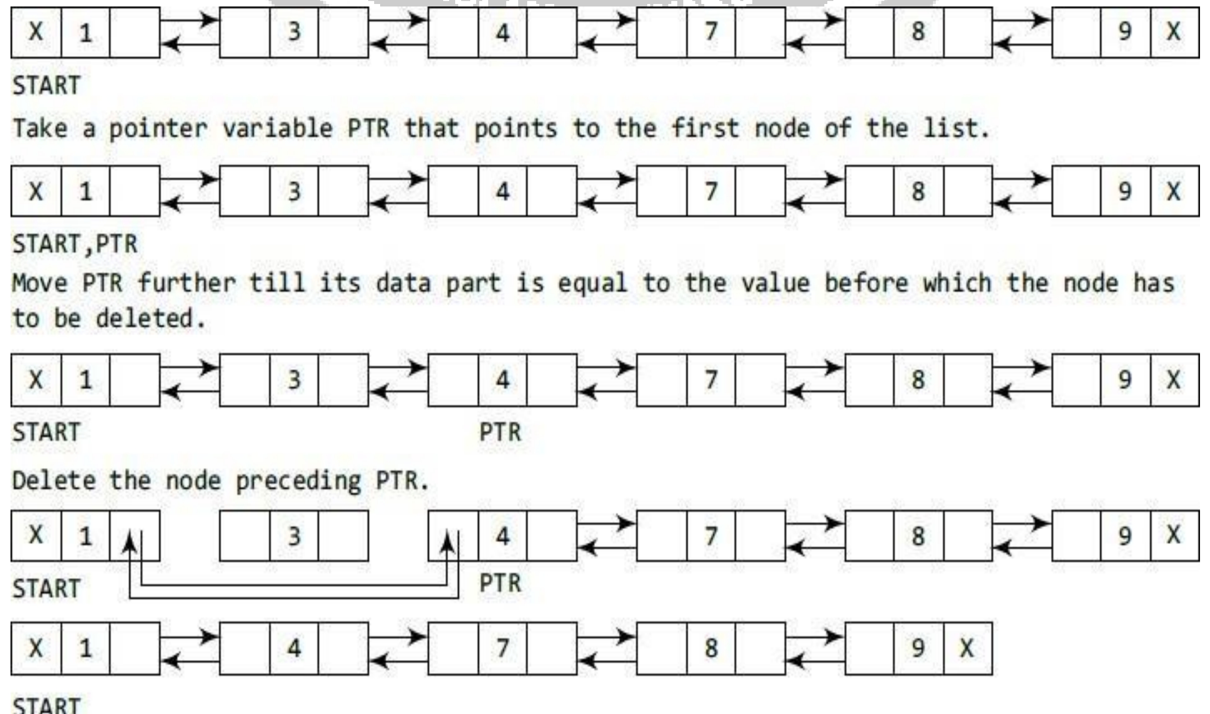
Algorithm to delete a node after a given node

```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Case 4: The node before a given node is deleted.

Suppose we want to delete the node preceding the node with value 4



Algorithm to delete a node before a given node

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- The while loop traverses through the linked list to reach the desired node.
- Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR.
- The memory of the node preceding PTR is freed and returned to the free pool.

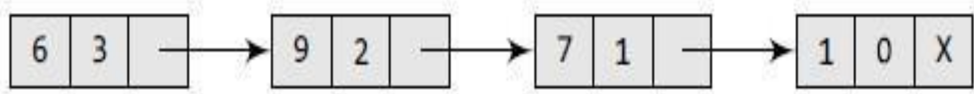
Polynomial Manipulation

Linked lists can be used to represent polynomials and the different operations that can be performed on them

Polynomial Representation

Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively. Every term of a polynomial can be represented as a node of the linked list

Linked representation of a polynomial



Polynomial manipulations such as addition, subtraction & differentiation etc.. can be performed using linked list

Declaration for Linked list implementation of Polynomial ADT

```
struct poly
{
int coeff;
int power;
struct poly * Next;
}
*list1,*list2,*list3;
```

Creation of the Polynomial

```
poly create(poly *head, poly *newnode)
{
```

```
poly*ptr;
if(head==NULL)
{
head=newnode;
return(head);
}
else
{
ptr=head;
while(ptr-> next!=NULL)
ptr=ptr->next;
ptr->next=newnode;
}
return(head);
}
```

Addition of Polynomials:

To add two polynomials, we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials, follow the following steps:

- Read two polynomials.
- Add them.

- **Display the resultant polynomial.**

Addition of Polynomials:

```
void add()
{
poly *ptr1, *ptr2, *newnode;
ptr1=list1;
ptr2=list2;
while(ptr1!=NULL && ptr2!= NULL)
{
newnode=malloc(sizeof(struct poly));
if(ptr1->power==ptr2->power)
{
newnode->coeff = ptr1->coeff + ptr2->coeff;
newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr->next;
ptr2=ptr2->next;
}
else
{
if(ptr1->power > ptr2->power)
{
newnode->coeff = ptr1->coeff
newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr1->next;
}
else
{
newnode->coeff = ptr2->coeff
```

```
newnode->power=ptr2->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr2=ptr2->next;
}
}
}
```

FOR SUBTRACTION OF POLYNOMIALS ,

add this statement in the above program newnode->coeff = ptr1->coeff - ptr2->coeff

STACK ADT

Stack is an abstract data type and it is also called linear data structure. It follows last in, first out (LIFO) strategy. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO.

The stack operations are given below.

- Stack() - creates a new stack that is empty. push(item) - adds a new item to the top of the stack. pop ()- removes the top item from the stack.
- peek() - returns the top item from the stack but does not remove it.
- isEmpty() - tests to see whether the stack is empty. It returns a boolean value.
- size() - returns the number of items on the stack.

Stack using Array:

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack

Step 1: Check whether stack is FULL. (top == SIZE-1)

Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

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

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 delete stack[top] and decrement top value by one (top--).

display() - Displays the elements of a Stack

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 4: Repeat above step until i value becomes '0'.

Program:

```
#include<stdio.h>

#include<conio.h>

#define SIZE 10

void push(int); void pop(); void display();

int stack[SIZE], top = -1;

void main()

{

int value, choice;

clrscr();

while(1)
```

```
{
printf("\n\n***** MENU *****\n");
printf("1. Push\n2. Pop\n3. Display\n4. Exit"); printf("\nEnter your choice: "); scanf("%d",&choice);
switch(choice){
case 1:
        printf("Enter the value to be insert: ");
        scanf("%d",&value);
        push(value);
        break;
case 2:
        pop();
        break;
case 3:
        display();
        break;
case 4:
        exit(0);
default: printf("\nWrong selection!!! Try again!!!");
}
}
}
```

```
void push(int value)

{

        if(top == SIZE-1)

        printf("\nStack is Full!!! Insertion is not possible!!!");

        else{

        top++;

        stack[top] = value;

        printf("\nInsertion success!!!");

        }

}

void pop()

{

        if(top == -1)

        printf("\nStack is Empty!!! Deletion is not possible!!!");

        else{

        printf("\nDeleted : %d", stack[top]);

        top--;

        }

}

void display()

{
```

```
            if(top == -1)
            printf("\nStack is Empty!!!");
            else{
            inti;
            printf("\nStack elements are:\n");
            for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
            }
        }
```

Stack using Linked List

- The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself.
- Stack implemented using array is not suitable, when we don't know the size of data which we are going to use.
- A stack data structure can be implemented by using linked list data structure.
- The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation.
- The Stack implemented using linked list can organize as many data values as we want.
- 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 next node in the list.
- The next field of the first element must be always NULL.

OPERATIONS

Step 1: Define a 'Node' structure with two member’s data and next.

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY PALKULAM, KANYAKUMARI OBSERVE OPTIMIZE OUTSPREAD

Step 2: Define a Node pointer 'top' and set it to NULL.

Step 3: Implement the main function by displaying Menu with list of operations and make suitable function calls in the main function.

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.

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 7: Finally, delete 'temp' (free(temp)).

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'.

Program:

```
#include<stdio.h>
#include<conio.h>
struct Node
{
int data;
struct Node *next;
}*top = NULL;
void push(int); void pop(); void display();
void main()
{
    int choice, value;
    clrscr();
    printf("\n:: Stack using Linked List ::\n");
    while(1){
    printf("\n****** MENU ******\n");
```

```
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
        case 1:
                printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(value);
                break;
        case 2:
                 pop();
                break;
        case 3:
                display();
                break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
        }
        }
}
```

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

```
void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}
void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else
    {
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
```

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

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

APPLICATION ON STACK

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure.

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

- Read all the symbols one by one from left to right in the given Postfix Expression
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
- Finally! perform a pop operation and display the popped value as final result

Example

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

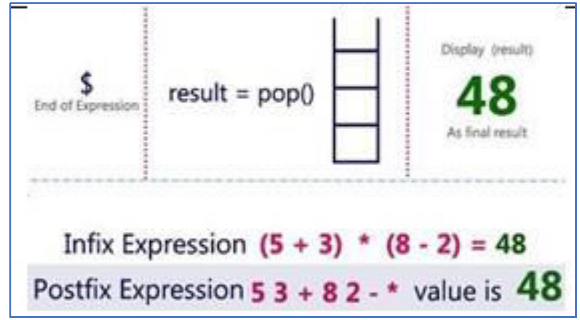
Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8 8	(5 + 3)
2	push(2)	2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		48	value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)



ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

Infix to Postfix Conversion

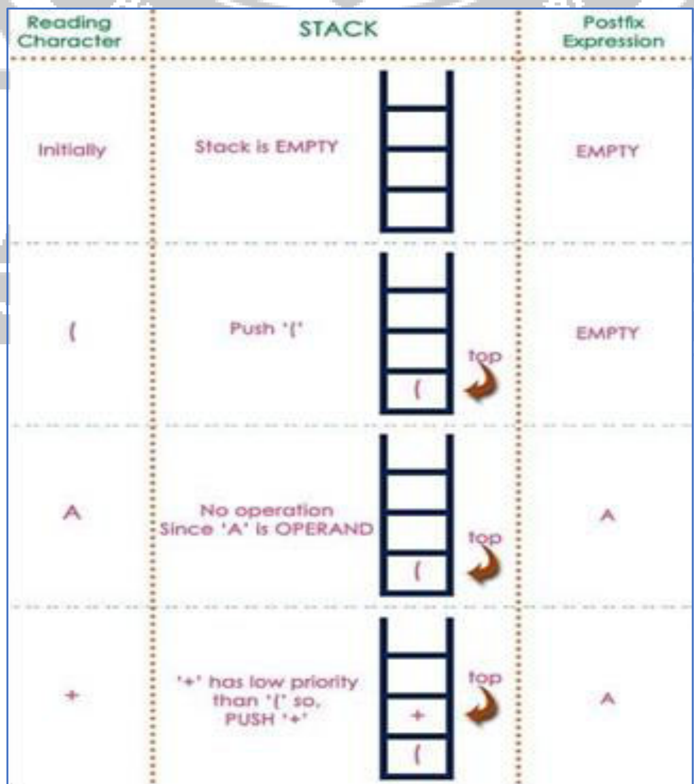
To convert Infix Expression into Postfix Expression using a stack data structure, Read all the symbols one by one from left to right in the given Infix Expression.

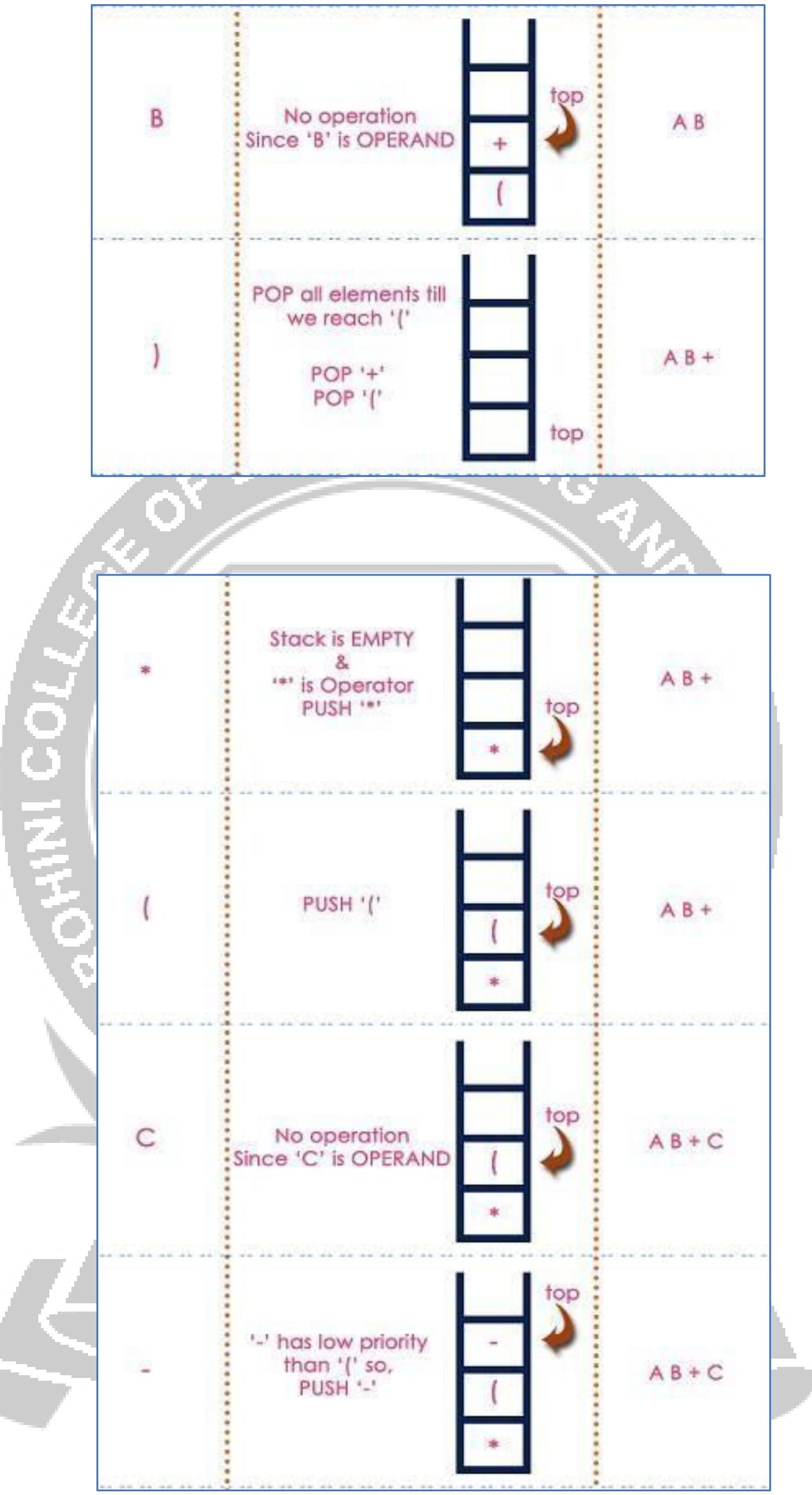
1. If the reading symbol is operand, then directly print it to the result (Output).
2. If the reading symbol is left parenthesis '(', then Push iton to the Stack.
3. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
4. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

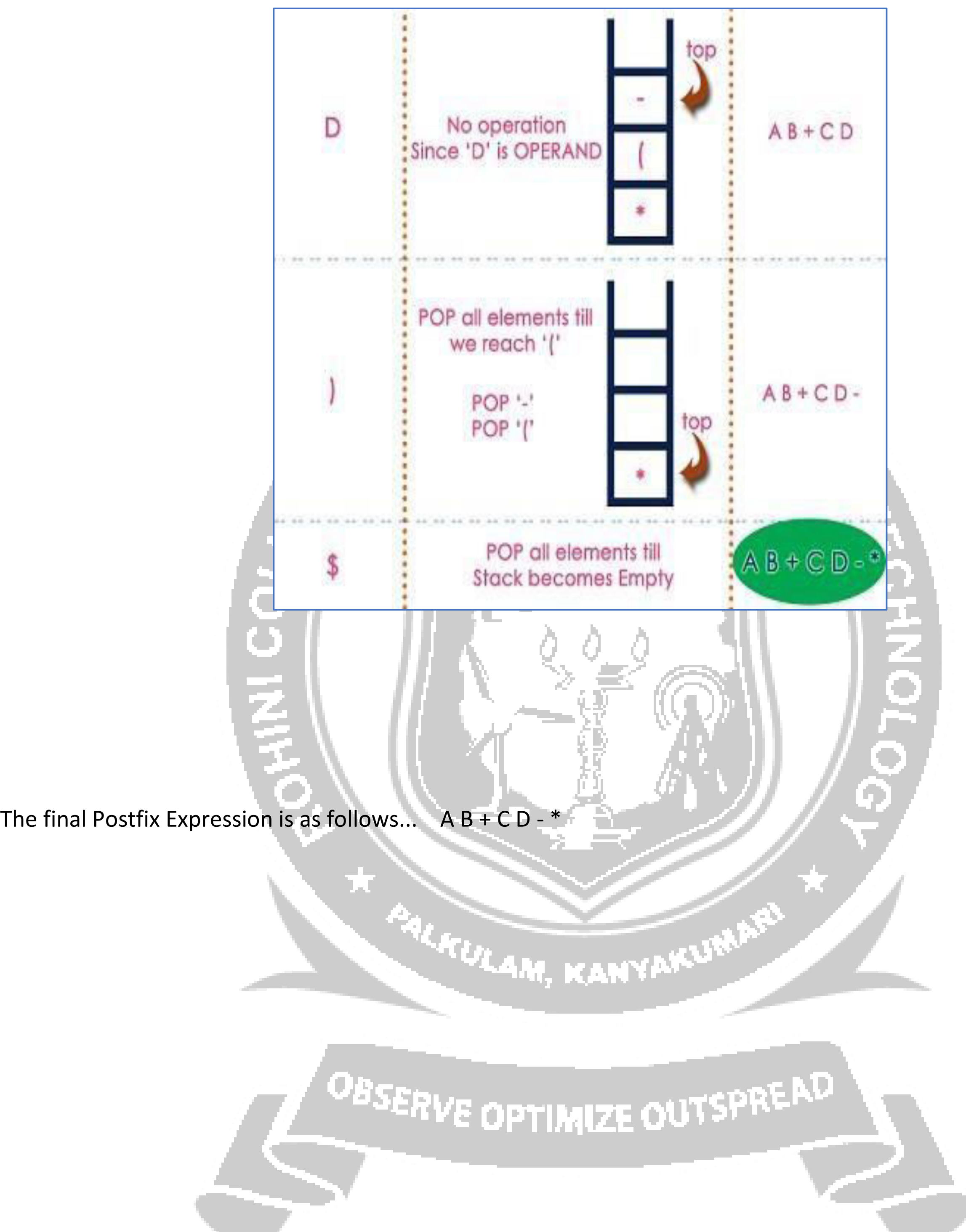
Example

Consider the following Infix Expression... (A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...



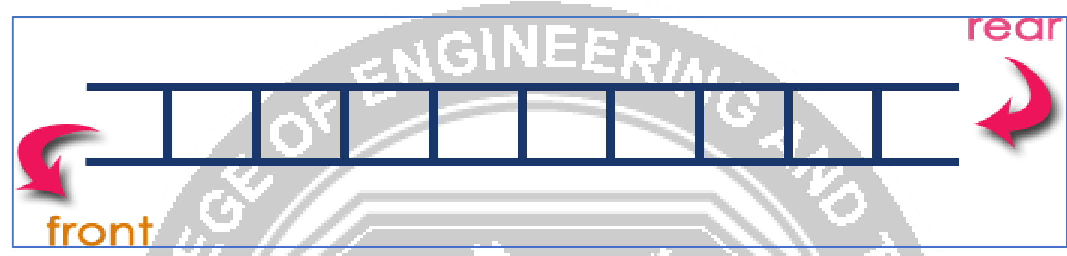

B
No operation
Since 'B' is OPERAND
+
(
top
A B
)
POP all elements till
we reach '('
POP '+'
POP '('
top
A B +
*
Stack is EMPTY
&
'*' is Operator
PUSH '*'
*
top
A B +
(
PUSH '('
(
*
top
A B +
C
No operation
Since 'C' is OPERAND
(
*
top
A B + C
-
'-' has low priority
than '(' so,
PUSH '-'
-
(
*
top
A B + C



The final Postfix Expression is as follows... A B + C D - *

QUEUE

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

Queue data structure using array can be implemented as follows

Before we implement actual operations, first follow the below steps to create an empty queue.

Step1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the user defined functions which are used in queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from frontposition. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Displays the elements of a Queue

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

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

Step 4: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

Program:

```
#include<stdio.h>

#include<conio.h>

#define SIZE 10 void enQueue(int); void deQueue(); void display();

int queue[SIZE], front = -1, rear = -1;

void main()

{

    int value, choice;

    clrscr();

    while(1)

    {

    printf("\n\n***** MENU *****\n");

    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");

    printf("\nEnter your choice: ");

    scanf("%d",&choice);

    switch(choice)

    {

    case 1:

        printf("Enter the value to be insert: ");

        scanf("%d",&value);

        enQueue(value);
```

```
                break;
        case 2:
                deQueue();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Try again!!!");
        }
        }
}
void enQueue(int value)
{
        if(rear == SIZE-1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
        else
        {
                if(front == -1)
                front = 0;
                rear++;
                queue[rear] = value;
```

```
printf("\nInsertion success!!!");
}
}
void deQueue()
{
if(front == rear)
printf("\nQueue is Empty!!! Deletion is not possible!!!");
else
{
printf("\nDeleted : %d", queue[front]);
front++;
if(front == rear) front= rear = -1;
}
}
void display()
{
if(rear == -1)
printf("\nQueue is Empty!!!");
else
{
inti;
```

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

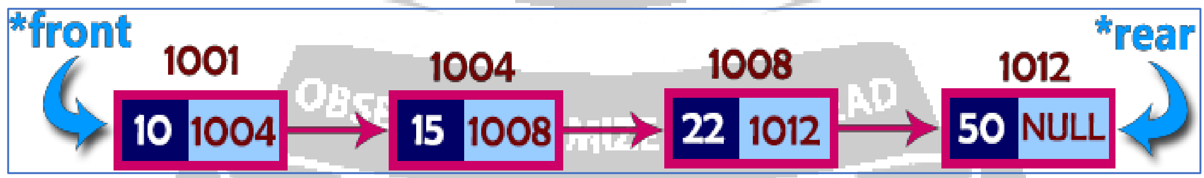
```
printf("\nQueue elements are:\n");
for(i=front; i<=rear; i++)
printf("%d\t",queue[i]);
}
}
```

QUEUE USING LINKED LIST

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

Example



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.

Step 2: Define a 'Node' structure with two members data and next.

Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.

Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

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

Step 1: Create a newNode with given value and set 'newNode → next' to NULL.

Step 2: Check whether queue is Empty (rear == NULL)

Step 3: If it is Empty then, set front = newNode and rear = newNode.

Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

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

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

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

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.

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

Step 4: Finally! Display 'temp → data ---> NULL'.

Program:

```
#include<stdio.h>

#include<conio.h>

struct Node

          {

           int data;

          struct Node *next;

          }*front = NULL,*rear = NULL;

void insert(int);

void delete();

void display();

void main()

{

          int choice, value;

          clrscr();

          printf("\n:: Queue Implementation using Linked List ::\n");

          while(1){

          printf("\n****** MENU ******\n");
```

```
printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
printf("Enter your choice: ");
scanf("%d",&choice);
switch(choice){
case 1:
        printf("Enter the value to be insert: ");
        scanf("%d", &value);
        insert(value);
        break;
case 2:
        delete();
        break;
case 3:
        display();
        break;
case 4: exit(0);
default: printf("\nWrong selection!!! Please try again!!!\n");
}
}
}
void insert(int value)
```

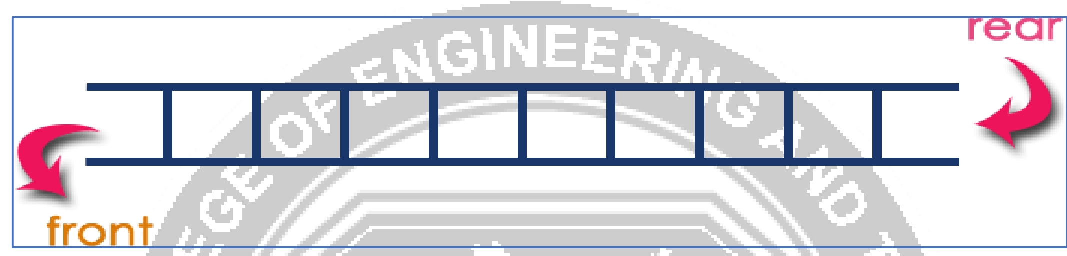
ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

```
{
        struct Node *newNode;
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = value;
        newNode -> next = NULL;
        if(front == NULL)
                front = rear = newNode;
        else
                {
                rear -> next = newNode;
                rear = newNode;
                }
        printf("\nInsertion is Success!!!\n");
}
void delete()
{
        if(front == NULL)
                printf("\nQueue is Empty!!!\n");
        else
        {
                struct Node *temp = front;
```

```
                front = front -> next;
                printf("\nDeleted element: %d\n", temp->data);
                free(temp);
        }
}
void display()
{
        if(front == NULL)
                printf("\nQueue is Empty!!!\n");
        else
        {
                struct Node *temp = front;
                while(temp->next != NULL)
                {
                printf("%d--->",temp->data); temp = temp -> next;
                }
        printf("%d--->NULL\n",temp->data);
        }
}
```

QUEUE

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

Queue data structure using array can be implemented as follows

Before we implement actual operations, first follow the below steps to create an empty queue.

Step1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the user defined functions which are used in queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from frontposition. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Displays the elements of a Queue

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

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

Step 4: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

Program:

```
#include<stdio.h>

#include<conio.h>

#define SIZE 10 void enQueue(int); void deQueue(); void display();

int queue[SIZE], front = -1, rear = -1;

void main()

{

    int value, choice;

    clrscr();

    while(1)

    {

    printf("\n\n***** MENU *****\n");

    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");

    printf("\nEnter your choice: ");

    scanf("%d",&choice);

    switch(choice)

    {

    case 1:

        printf("Enter the value to be insert: ");

        scanf("%d",&value);

        enQueue(value);
```

```
            break;
        case 2:
            deQueue();
            break;
        case 3: display();
            break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Try again!!!");
        }
        }
}
void enQueue(int value)
{
        if(rear == SIZE-1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
        else
        {
            if(front == -1)
            front = 0;
            rear++;
            queue[rear] = value;
```

```
                printf("\nInsertion success!!!");
        }
}
void deQueue()
{
        if(front == rear)
                printf("\nQueue is Empty!!! Deletion is not possible!!!");
        else
        {
        printf("\nDeleted : %d", queue[front]);
        front++;
        if(front == rear) front= rear = -1;
        }
}
void display()
{
        if(rear == -1)
        printf("\nQueue is Empty!!!");
        else
        {
        inti;
```

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

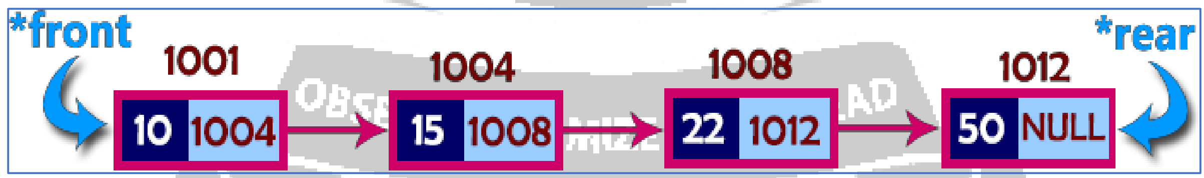
```
printf("\nQueue elements are:\n");
for(i=front; i<=rear; i++)
printf("%d\t",queue[i]);
}
}
```

QUEUE USING LINKED LIST

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

Example



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.

Step 2: Define a 'Node' structure with two members data and next.

Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.

Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

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

Step 1: Create a newNode with given value and set 'newNode → next' to NULL.

Step 2: Check whether queue is Empty (rear == NULL)

Step 3: If it is Empty then, set front = newNode and rear = newNode.

Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

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

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

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

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.

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

Step 4: Finally! Display 'temp → data ---> NULL'.

Program:

```
#include<stdio.h>

#include<conio.h>

struct Node

        {

         int data;

        struct Node *next;

        }*front = NULL,*rear = NULL;

void insert(int);

void delete();

void display();

void main()

{

        int choice, value;

        clrscr();

        printf("\n:: Queue Implementation using Linked List ::\n");

        while(1){

        printf("\n****** MENU ******\n");
```

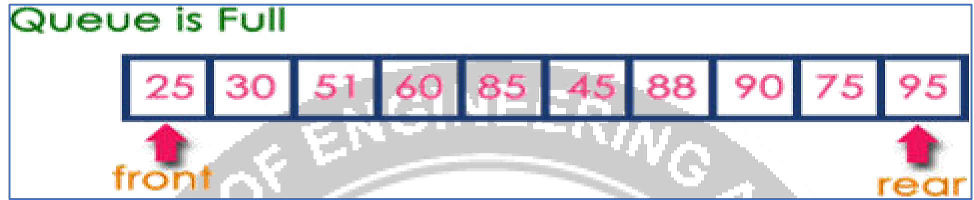
```
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
        case 1:
            printf("Enter the value to be insert: ");
            scanf("%d", &value);
            insert(value);
            break;
        case 2:
            delete();
            break;
        case 3:
            display();
            break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
        }
        }
    }
    void insert(int value)
```

```
{
        struct Node *newNode;
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = value;
        newNode -> next = NULL;
        if(front == NULL)
                front = rear = newNode;
        else
                {
                rear -> next = newNode;
                rear = newNode;
                }
        printf("\nInsertion is Success!!!\n");
}
void delete()
{
        if(front == NULL)
                printf("\nQueue is Empty!!!\n");
        else
        {
                struct Node *temp = front;
```

```
        front = front -> next;

        printf("\nDeleted element: %d\n", temp->data);

        free(temp);

    }

}

void display()

{

    if(front == NULL)

        printf("\nQueue is Empty!!!\n");

    else

    {

        struct Node *temp = front;

        while(temp->next != NULL)

        {

        printf("%d--->",temp->data); temp = temp -> next;

        }

    printf("%d--->NULL\n",temp->data);

    }

}
```

CIRCULAR QUEUE

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example, consider the queue below After inserting all the elements into the queue.



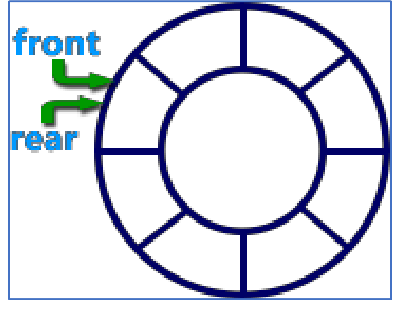
Now consider the following situation after deleting three elements from the queue...

This situation also says that Queue is full and we cannot insert the new element because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue we cannot make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem, we use circular queue data structure.

A Circular Queue can be defined as follows...

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all user defined functions used in circular queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (intcQueue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check rear == SIZE - 1 &&front != 0 if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Circular Queue

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

Step 1: Check whether queue is EMPTY. (front == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i<= rear' becomes FALSE.

Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i<= SIZE - 1' becomes FALSE.

Step 6: Set i to 0.

Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i<= rear' becomes FALSE.

Program:

```
#include<stdio.h>

#include<conio.h>

#define SIZE 5

void enQueue(int);
```

```
void deQueue();

void display();

intcQueue[SIZE], front = -1, rear = -1;

void main()

{

        int choice, value;

        clrscr();

        while(1)

        {

        printf("\n****** MENU ******\n");

        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d",&choice);

        switch(choice)

        {

        case 1:

                printf("\nEnter the value to be insert: ");

                scanf("%d",&value);

                enQueue(value);

                break;

        case 2:
```

```
                deQueue();
                break;
        case 3:
                display();
                break;
        case 4: exit(0);
        default: printf("\nPlease select the correct choice!!!\n");
        }
        }
}
void enQueue(int value)
{
        if((front == 0 && rear == SIZE - 1) || (front == rear+1))
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
        else
        {
        if(rear == SIZE-1 && front != 0)
        rear = -1;
        cQueue[++rear] = value;
        printf("\nInsertion Success!!!\n");
        if(front == -1)
```

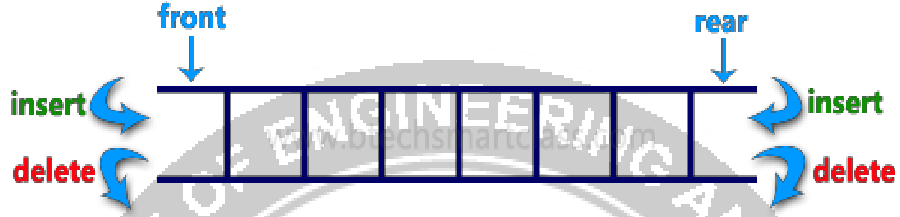
```
            front = 0;

            }

            }

void deQueue()

{

            if(front == -1 && rear == -1)

            printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");

            else

            {

            printf("\nDeleted element : %d\n",cQueue[front++]);

            if(front == SIZE)

            front = 0;

            if(front-1 == rear)

            front = rear = -1;

            }

}

void display()

{

            if(front == -1)

            printf("\nCircular Queue is Empty!!!\n");

            else
```

```
{
inti = front;
printf("\nCircular Queue Elements are : \n");
if(front <= rear){ while(i<= rear) printf("%d\t",cQueue[i++]);
}
Else
{
while(i<= SIZE - 1) printf("%d\t", cQueue[i++]); i = 0;
while(i<= rear)
printf("%d\t",cQueue[i++]);
}
}
}
```

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

Double Ended Queue (Dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

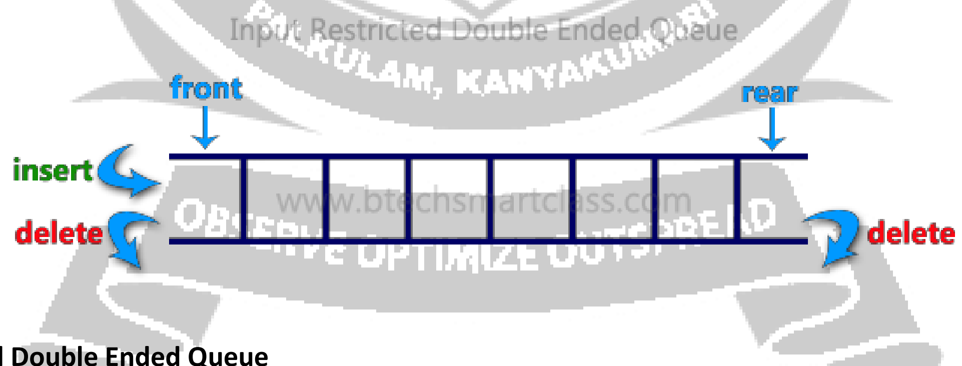


Double Ended Queue can be represented in TWO ways, those are as follows... Input Restricted Double Ended Queue

- Output Restricted Double Ended Queue
- Input Restricted Double Ended Queue

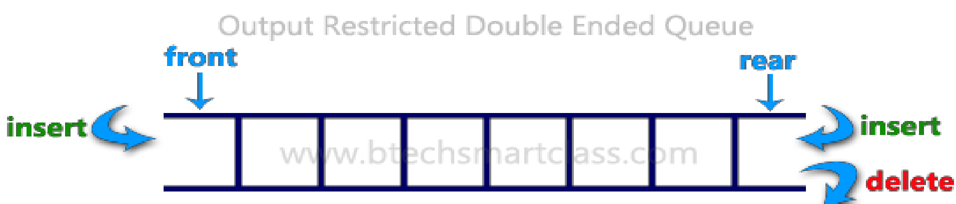
Input restricted double ended queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Program

```
#include<stdio.h>

#include<conio.h>

#define SIZE 100

void enQueue(int);

int deQueueFront();

int deQueueRear();

void enQueueRear(int);

void enQueueFront(int);

void display();

int queue[SIZE];

int rear = 0, front = 0;

int main()

{

    char ch;

    int choice1, choice2, value;

    printf("\n******* Type of Double Ended Queue *******\n");

    do

    {

    printf("\n1.Input-restricted deque \n");

    printf("2.output-restricted deque \n");
```

```
printf("\nEnter your choice of Queue Type : "); scanf("%d",&choice1);
switch(choice1)
{
case 1:
printf("\nSelect the Operation\n");
printf("1.Insert\n2.Delete from Rear\n3.Delete from Front\n4. Display");
do
{
printf("\nEnter your choice for the operation in c deque: ");
scanf("%d",&choice2);
switch(choice2)
{
        case 1:
                enQueueRear(value);
                display();
                break;
        case 2:
                value = deQueueRear();
                printf("\nThe value deleted is %d",value);
                display();
                break;
```

```
        case 3:

        value=deQueueFront();

        printf("\nThe value deleted is %d",value);

        display();

        break;

        case 4: display();

        break;

        default:printf("Wrong choice");

}

printf("\nDo you want to perform another operation (Y/N): ");

ch=getch();

getch();

 }while(ch=='y'||ch=='Y');

break;

case 2 :

printf("\n---- Select the Operation---- \n");

printf("1. Insert at Rear\n2. Insert at Front\n3. Delete\n4. Display");

do

{

        printf("\nEnter your choice for the operation: ");

        scanf("%d",&choice2);
```

```
        switch(choice2)
        {
        case 1:
                enQueueRear(value);
                display();
                break;
        case 2:
                enQueueFront(value);
                display();
                break;
        case 3:
                value = deQueueFront();
                printf("\nThe value deleted is %d",value);
                display();
                break;
        case 4: display();
                break;
        default:printf("Wrong choice");
        }
printf("\nDo you want to perform another operation (Y/N): ");
ch=getch();
```

```
        getch();
        }
         } while(ch=='y'||ch=='Y');
        break ;
        printf("\nDo you want to continue(y/n):");
        ch=getch();
        }while(ch=='y'||ch=='Y');
}
void enQueueRear(int value)
{
        char ch;
        if(front == SIZE/2)
        {
        printf("\nQueue is full!!! Insertion is not possible!!! ");
        return;
        }
        do
        {
        printf("\nEnter the value to be inserted:");
        scanf("%d",&value); queue[front] = value;
```

```
        front++;
        printf("Do you want to continue insertion Y/N");
        ch=getch();
        }while(ch=='y');
}
void enQueueFront(int value)
{
        char ch;
        if(front==SIZE/2)
        {
        printf("\nQueue is full!!! Insertion is not possible!!!");
        return;
        }
        do
        {
        printf("\nEnter the value to be inserted:");
        scanf("%d",&value);
        rear--;
        queue[rear] = value;
        printf("Do you want to continue insertion Y/N");
        ch = getch();
```

```
}
while(ch == 'y');
}
int deQueueRear()
{
        int deleted;
        if(front == rear)
        {
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
        return 0;
        }
        front--;
        deleted = queue[front+1];
        return deleted;
}
int deQueueFront()
{
        int deleted;
        if(front == rear)
        {
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
```

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

```
        return 0;
        }
        rear++;
        deleted = queue[rear-1];
        return deleted;
}


void display()
{
        inti;
        if(front == rear)
        printf("\nQueue is Empty!!! Deletion is not possible!!!")
        else{
        printf("\nThe Queue elements are:");
        for(i=rear; i< front; i++)
        {
        printf("%d\t ",queue[i]);
        }
        }
}
```

INTRODUCTION

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.

Basic Terminology

Root node- The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

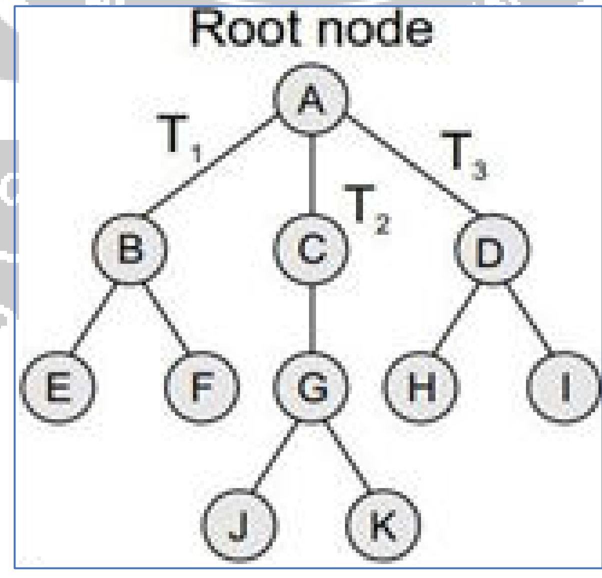
Sub-trees- If the root node R is not NULL, then the trees T1,T2,and T3 are called the sub- trees of R.

Leaf node - A node that has no children is called the leaf node or the terminal node.

Path - A sequence of consecutive edges is called a path. For example, in Fig., the path from the root node A to node I is given as: A, D, and I.

Ancestor node - An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig., nodes A, C, and G are the ancestors of node K.

Descendant node- A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig., nodes C, G, J, and K are the descendants of node A.



TYPES OF TREES

Trees are of following 6 types:

- General trees
- Forests
- Binary trees
- Binary search trees
- Expression trees
- Tournament trees

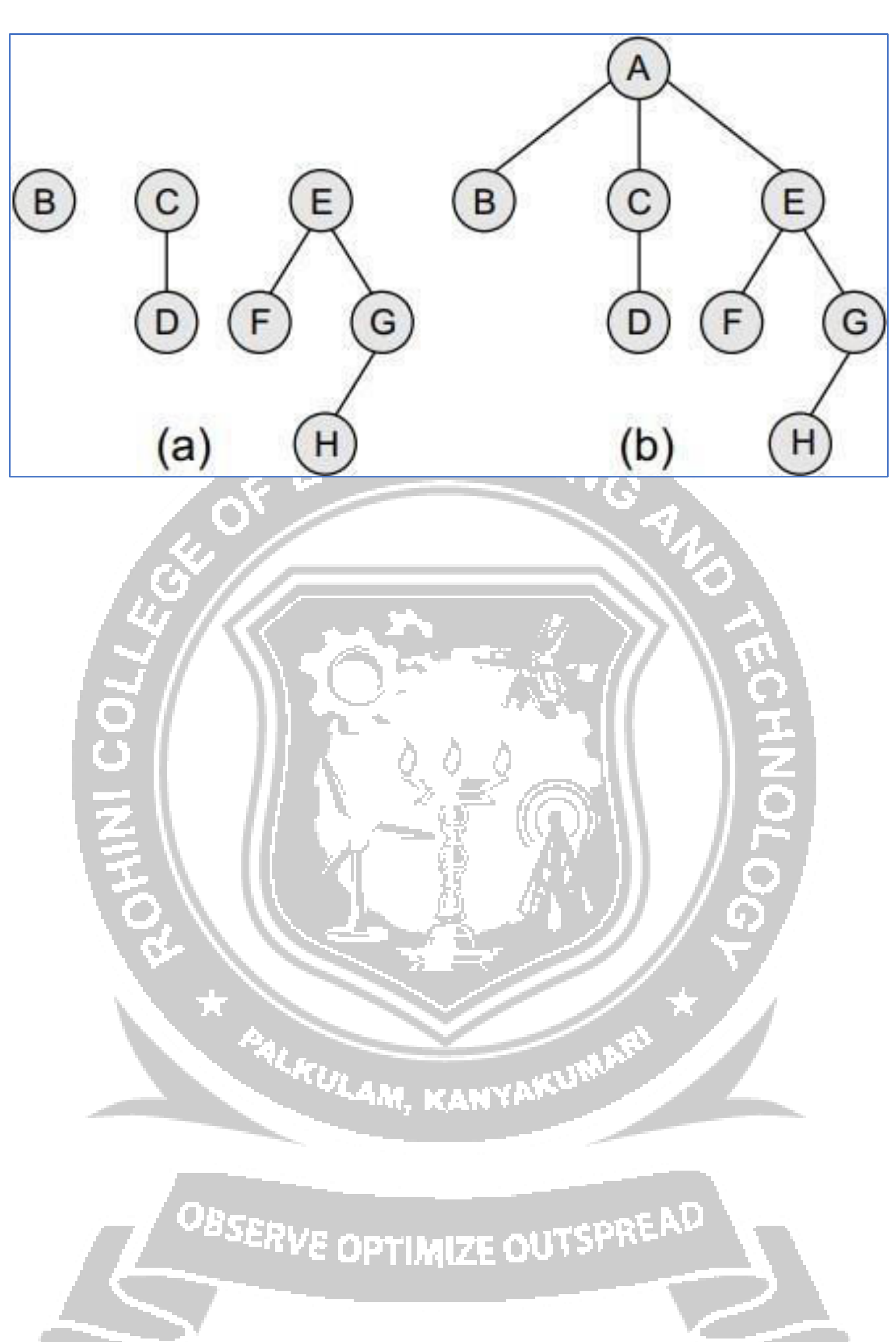
General Trees

General trees are data structures that store elements hierarchically. The top node of a tree is the root node and each node, except the root, has a parent.

- A node in a general tree (except the leaf nodes) may have zero or more sub-trees.
- General trees which have 3 sub-trees per node are called ternary trees. However, the number of sub-trees for any node may be variable. For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

Forests

- Forest is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.
- Every node of a tree is the root of some sub-tree. Therefore, all the sub-trees immediately below a node form a forest. We can convert a forest into a tree by adding a single node as the root node of the tree. For example, below Fig (a) shows a forest and Fig.(b) shows the corresponding tree. Similarly, we can convert a general tree into a forest by deleting the root node of the tree.


A
B
C
D
E
F
G
H
(a)
(b)

TRAVERSING A BINARY TREE

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a non-linear data structure in which the elements can be traversed in many different ways.

Pre-order Traversal

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,

2. Traversing the left sub-tree, and finally

3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            Write TREE -> DATA
Step 3:            PREORDER(TREE -> LEFT)
Step 4:            PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

NOTE: Pre-order traversal is also called as depth-first traversal or NLR traversal algorithm (Node-Left-Right).

In-order Traversal

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,

2. Visiting the root node, and finally

3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

NOTE: In-order traversal is also called as symmetric traversal (or) LNR traversal algorithm (Left-Node-Right).

Post-order Traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node.

The algorithm works by:

1. Traversing the left sub-tree,

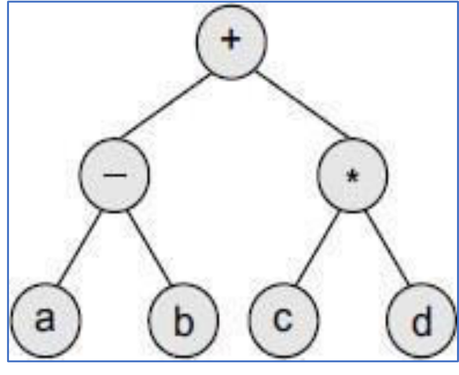
2. Traversing the right sub-tree, and finally

3. Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             POSTORDER(TREE -> LEFT)
Step 3:             POSTORDER(TREE -> RIGHT)
Step 4:             Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

NOTE: Post-order algorithm is also known as the LRN traversal algorithm (Left- Right- Node)

Example 1: Find the In-order, Pre-order and post-order traversal of given tree



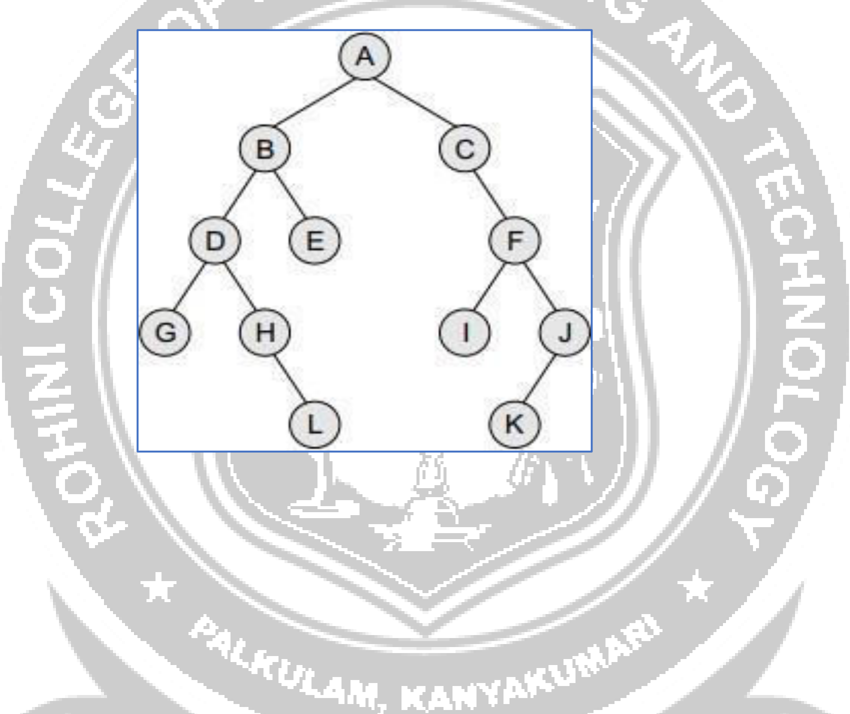
Solution

Pre-order Traversal : + – a b * c d

In-order Traversal : a-b+c*d

Post-order Traversal : ab-cd*+

Example 2: Find the In-order, Pre-order and post-order traversal of given tree



Solution

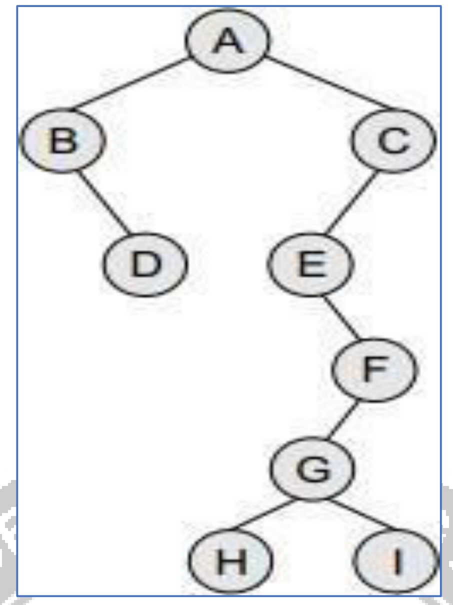
Pre-order Traversal : A, B, D, G, H, L, E, C, F, I, J, and K

In-order Traversal : G, D, H, L, B, E, A, C, I, F, K, and J

Post-order Traversal : G, L, H, D, E, B, I, K, J, F, C, and A

Example 3: Find the In-order, Pre-order and post-order traversal of given tree

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD



Solution

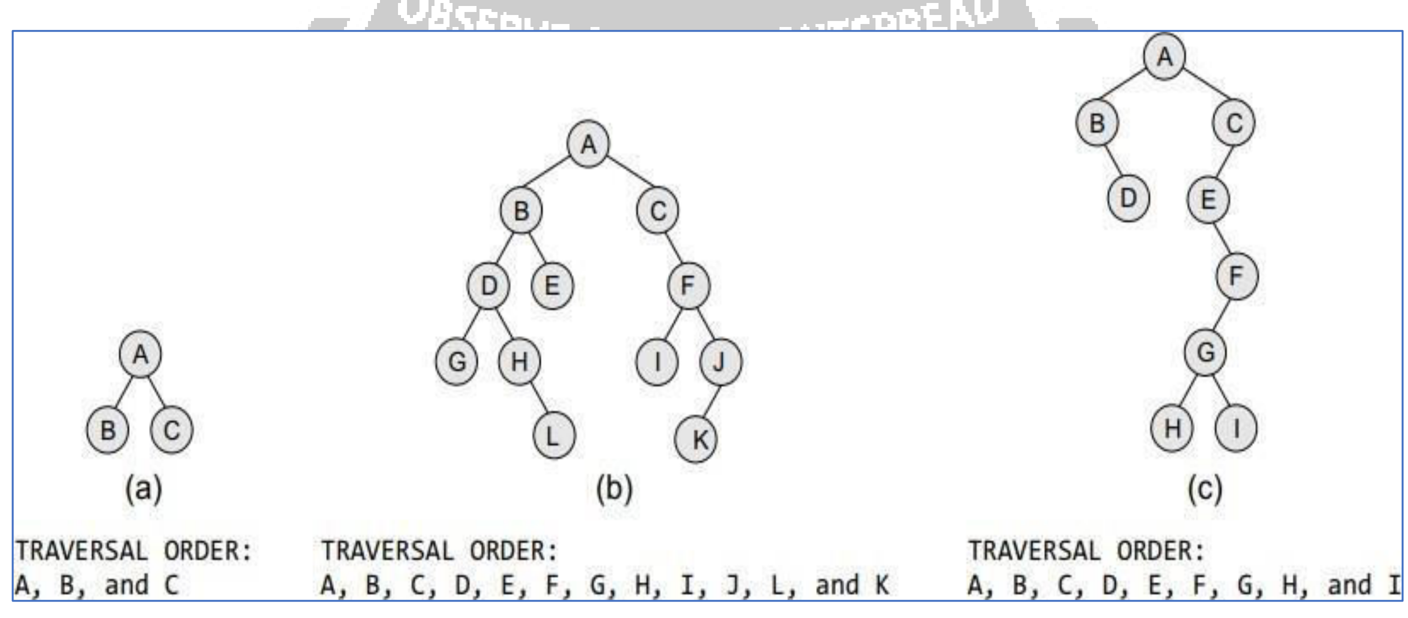
Pre-order Traversal : A, B, D, C, D, E, F, G, H, and I

In-order Traversal : B, D, A, E, H, G, I, F, and C

Post- order Traversal : D, B, H, I, G, F, E, C, and A

Level-order Traversal

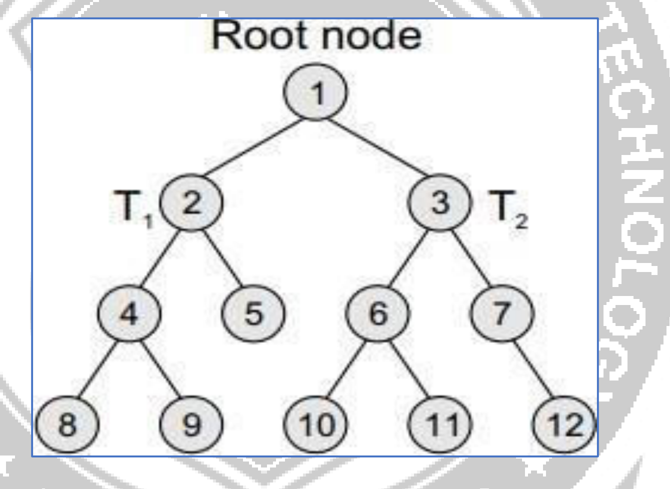
In level-order traversal, all the nodes at a level are accessed before going to the next level. This algorithm is also called as the breadth-first traversal algorithm. Consider the trees given in Fig. and note the level order of these trees.



BINARY TREES

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.

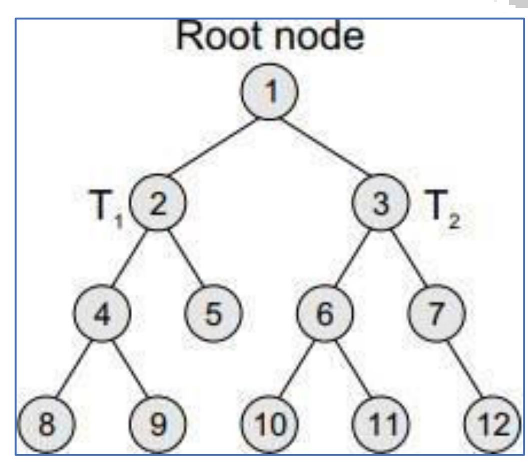
- Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.
- In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.



TERMINOLOGY

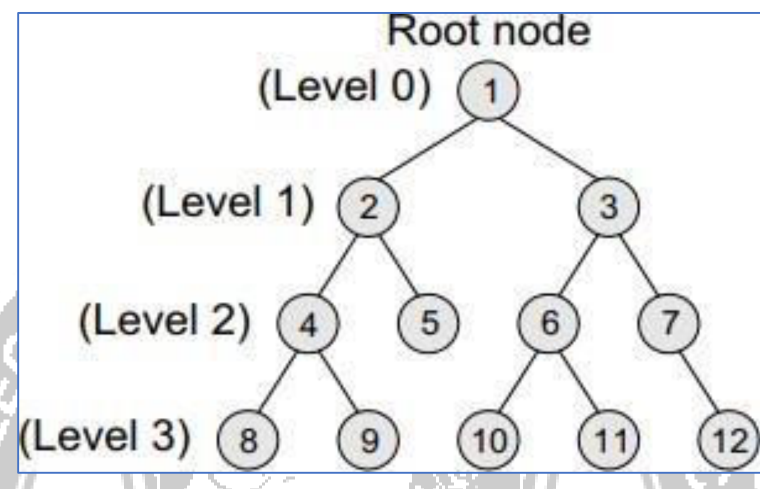
Parent

If N is any node in T that has left successor S1 and right successor S2, then N is called the parent of S 1 and S2. Correspondingly, S1 and S2 are called the left child and the right child of N. Every node other than the root node has a parent



Level number

Every node in the binary tree is assigned a level number. The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.



Degree of a node

Degree of a node is equal to the number of children that a node has The degree of a leaf node is zero.

- In-degree

  In-degree of a node is the number of edges arriving at that node.

- Out-degree

  Out-degree of a node is the number of edges leaving that node.

It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in above tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

Path

A sequence of consecutive edges. For example, in above Fig., the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

Sibling

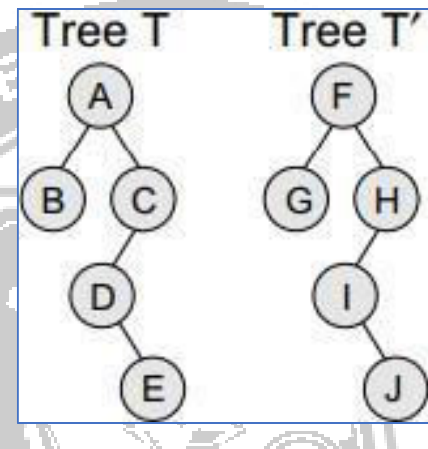
All nodes that are at the same level and share the same parent are called siblings (brothers). For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.

Leaf node

A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.

Similar binary trees

Two binary trees T and T' are said to be similar if both these trees have the same structure. Figure shows two similar binary trees.



Edge

It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

Depth

The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

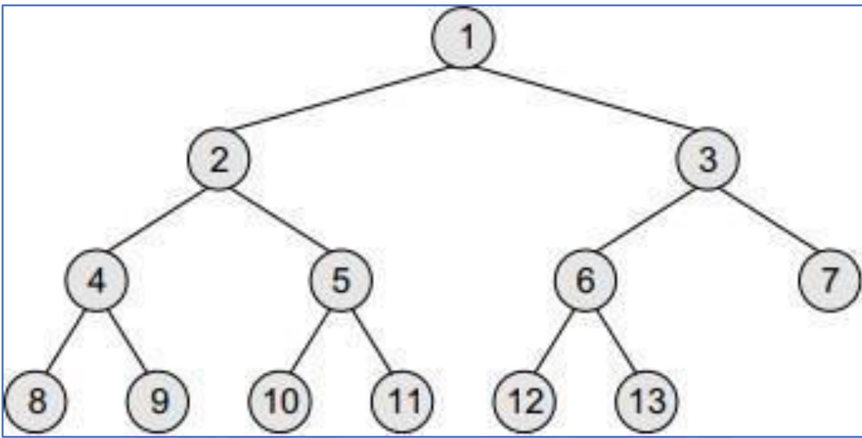
Height of a tree

It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

COMPLETE BINARY TREES

- A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.
- In a complete binary tree Tn , there are exactly n nodes and level r of T can have at most 2r nodes.

Figure shows a complete binary tree.



Above tree has exactly 13 nodes.

- The formula can be given as—if K is a parent node, then its left child can be calculated as $2 \times K$ and its right child can be calculated as $2 \times K + 1$.
- For example, the children of the node 4 are 8 (2×4) and 9 ($2 \times 4 + 1$). Similarly, the parent of the node K can be calculated as $| K/2 |$. Given the node 4, its parent can be calculated as $| 4/2 | = 2$.
- The height of a tree Tn having exactly n nodes is given as:

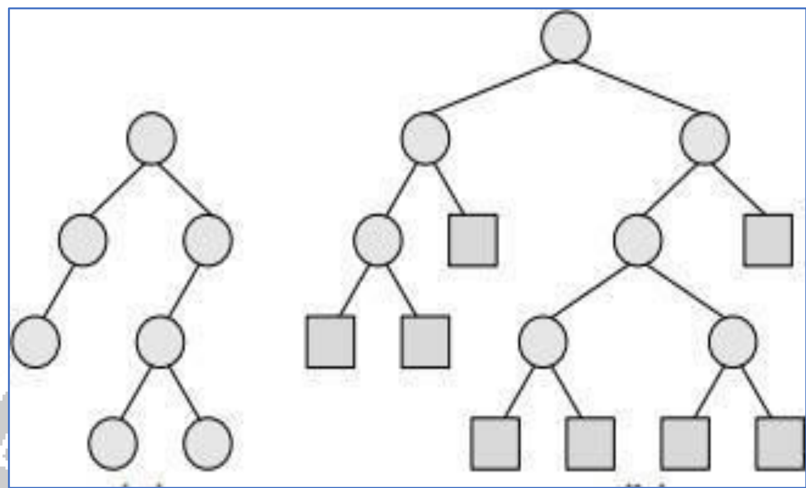
$$H_n = | \log_2 (n + 1) |$$

NOTE: In Fig. 9.7, level 0 has 20 = 1 node, level 1 has 21 = 2 nodes, level 2 has 22 = 4 nodes, level 3 has 6 nodes which is less than the maximum of 23 = 8 nodes.

Extended Binary Trees

- A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children.
- In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes.
- In Given Fig., the internal nodes are represented using circles and the external nodes are represented using squares.

- To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes. Below Figure shows how an ordinary binary tree is converted into an extended binary tree.



Representation of Binary Trees in the Memory

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.

1. Linked representation of binary trees

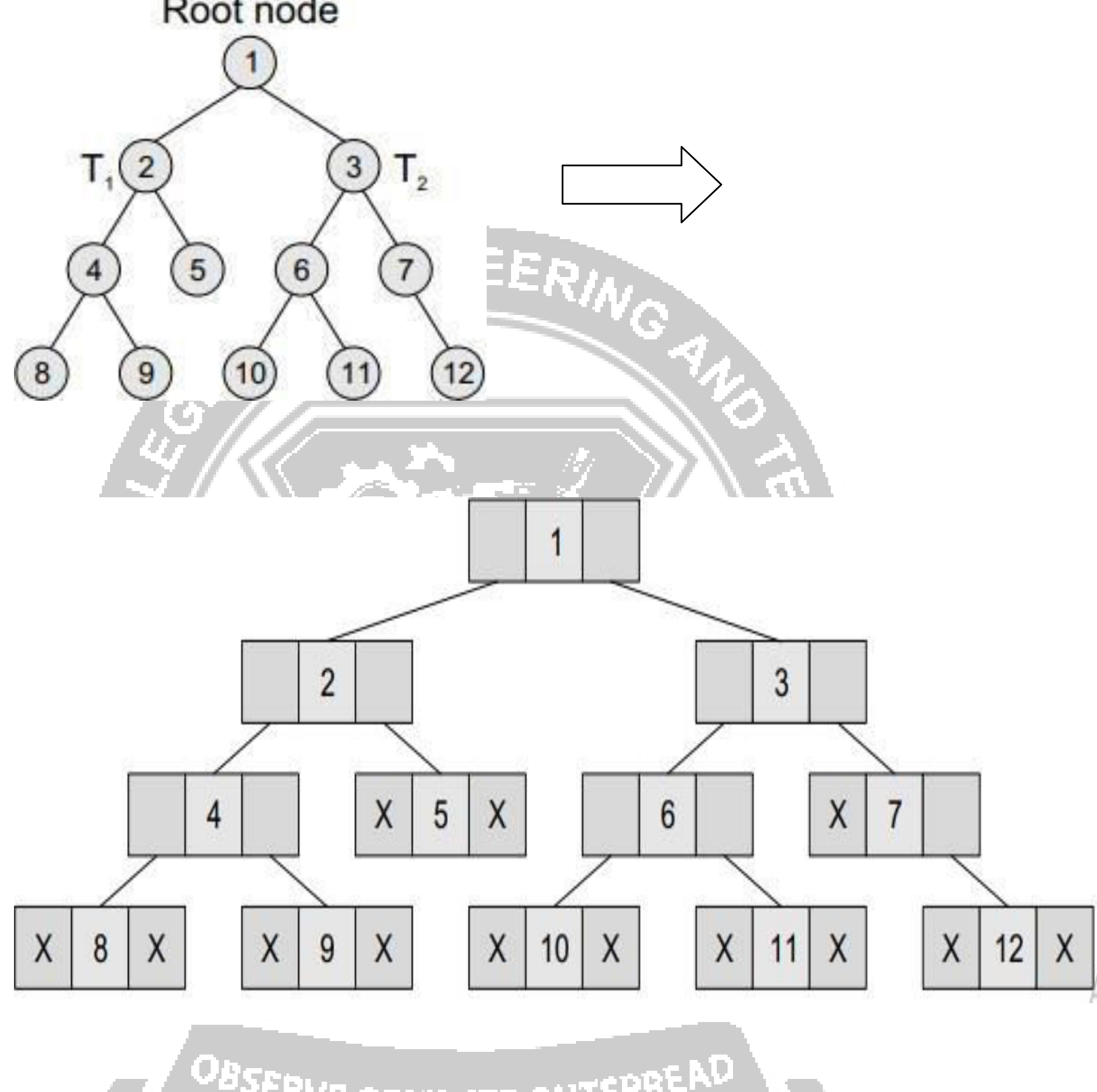
In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.

```
struct node
{
struct node *left;
int data;
struct node *right;
};
```

- Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree.
- If ROOT = NULL, then the tree is empty.

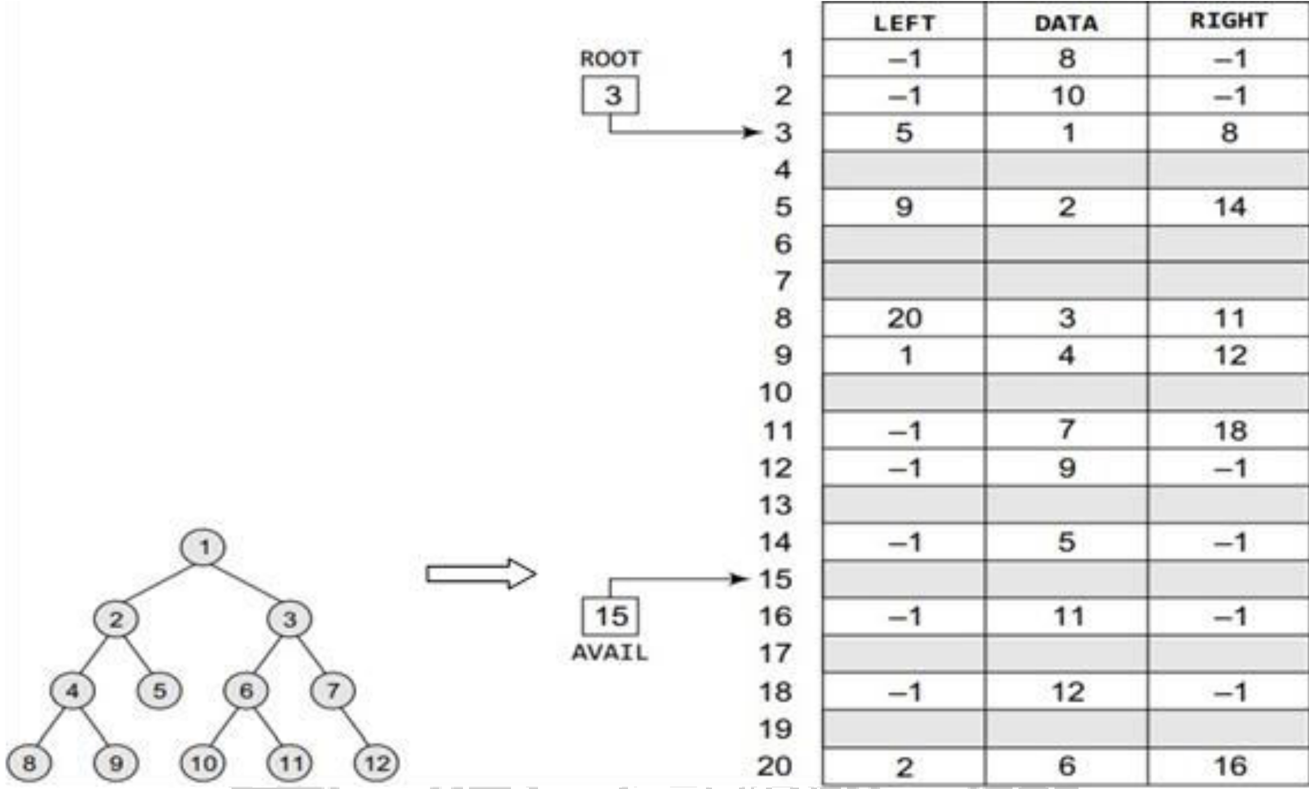
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

Consider the binary tree given. The schematic diagram of the linked representation of the binary tree is shown in Fig.



In Fig, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

Look at the tree given. Note how this tree is represented in the main memory using a linked list



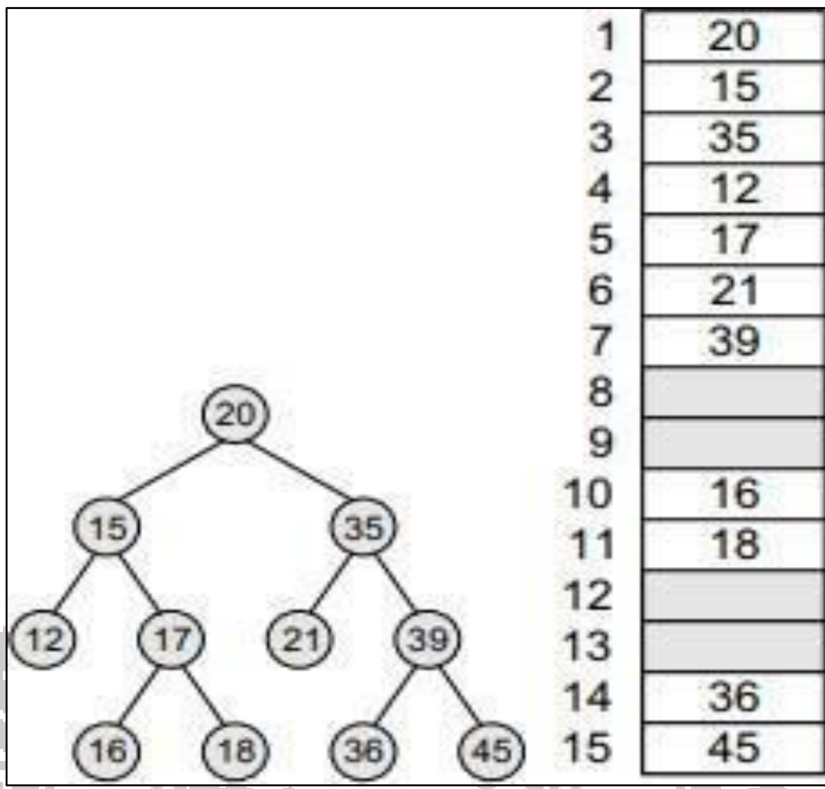
	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16

2. Sequential representation of binary trees

Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space.

A sequential binary tree follows the following rules:

- A one-dimensional array, called TREE, is used to store the elements of tree.
- The root of the tree will be stored in the first location. That is, TREE [1] will store the data of the root element.
- The children of a node stored in location K will be stored in locations (2 × K) and (2× K+1).
- The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE [1] = NULL, then the tree is empty.

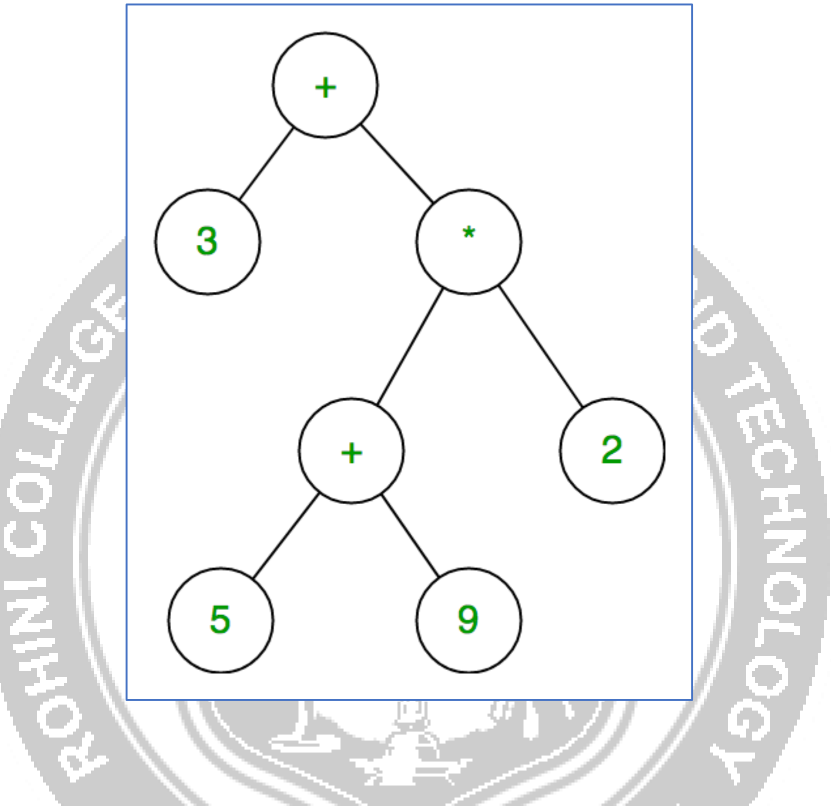


This shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4

PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

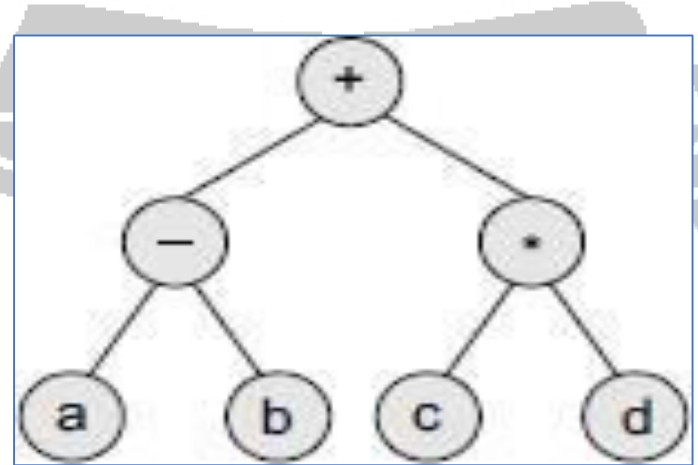
Expression Trees

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:



Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression given as:

Exp = (a – b) + (c * d)



Evaluating the expression represented by an expression tree:

Let t be the expression tree

If t is not null then

If t.value is operand then

Return t.value

A = solve(t.left)

B = solve(t.right)

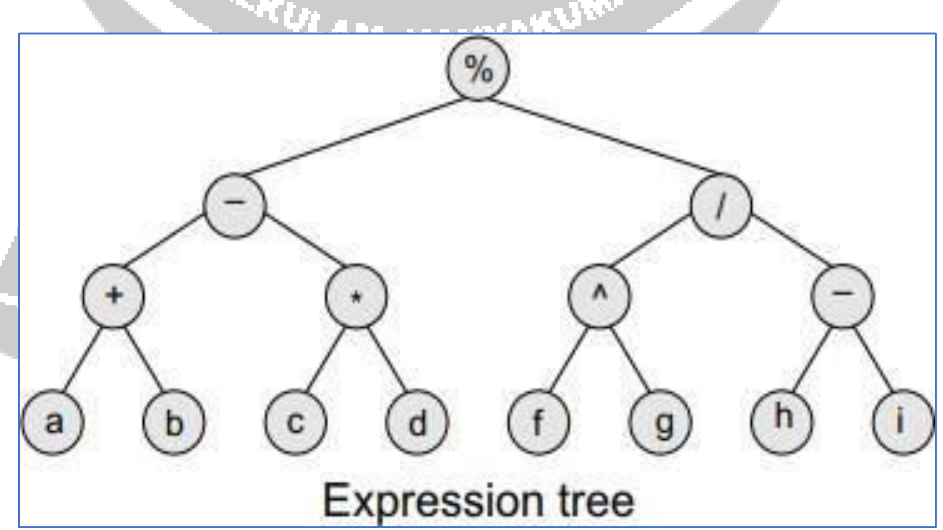
// calculate applies operator 't.value'

// on A and B, and returns value

Return calculate(A, B, t.value)

Example 1. Given an expression, Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)), construct the corresponding binary tree.

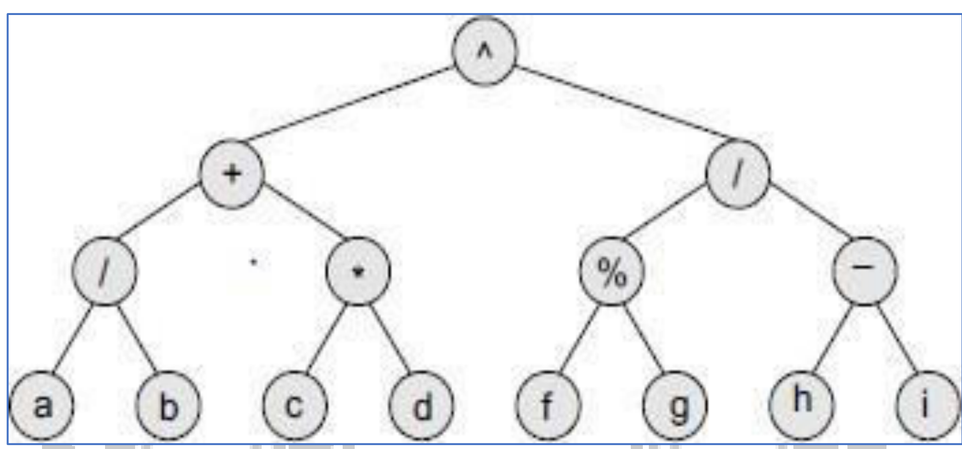
Solution



Expression tree

Example 2. Given the binary tree, write down the expression that it represents.

Solution



Construction of Expression Tree:

Now for constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

- If a character is an operand push that into the stack
- If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

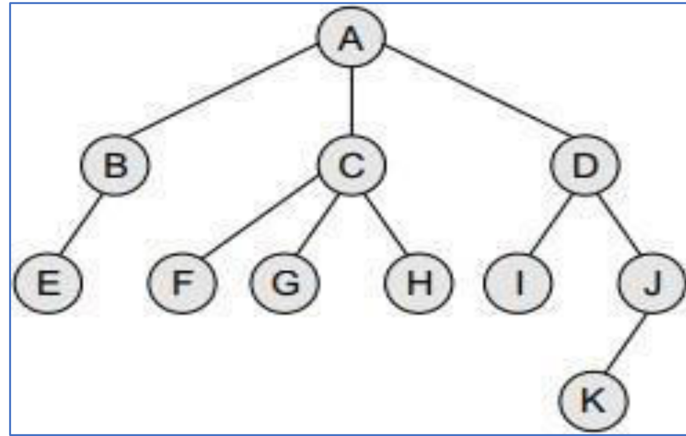
Creating a Binary Tree from a General Tree

The rules for converting a general tree to a binary tree are given below. Note that a general tree is converted into a binary tree and not a binary search tree.

Rule 1: Root of the binary tree = Root of the general tree

Rule 2: Left child of a node = Leftmost child of the node in the binary tree in the general tree

Rule 3: Right child of a node in the binary tree = Right sibling of the node in the general tree



Convert the given general tree into a binary tree

Now let us build the binary tree.

Step 1: Node A is the root of the general tree, so it will also be the root of the binary tree.

Step 2: Left child of node A is the leftmost child of node A in the general tree and right child of node A is the right sibling of the node A in the general tree. Since node A has no right sibling in the general tree, it has no right child in the binary tree.

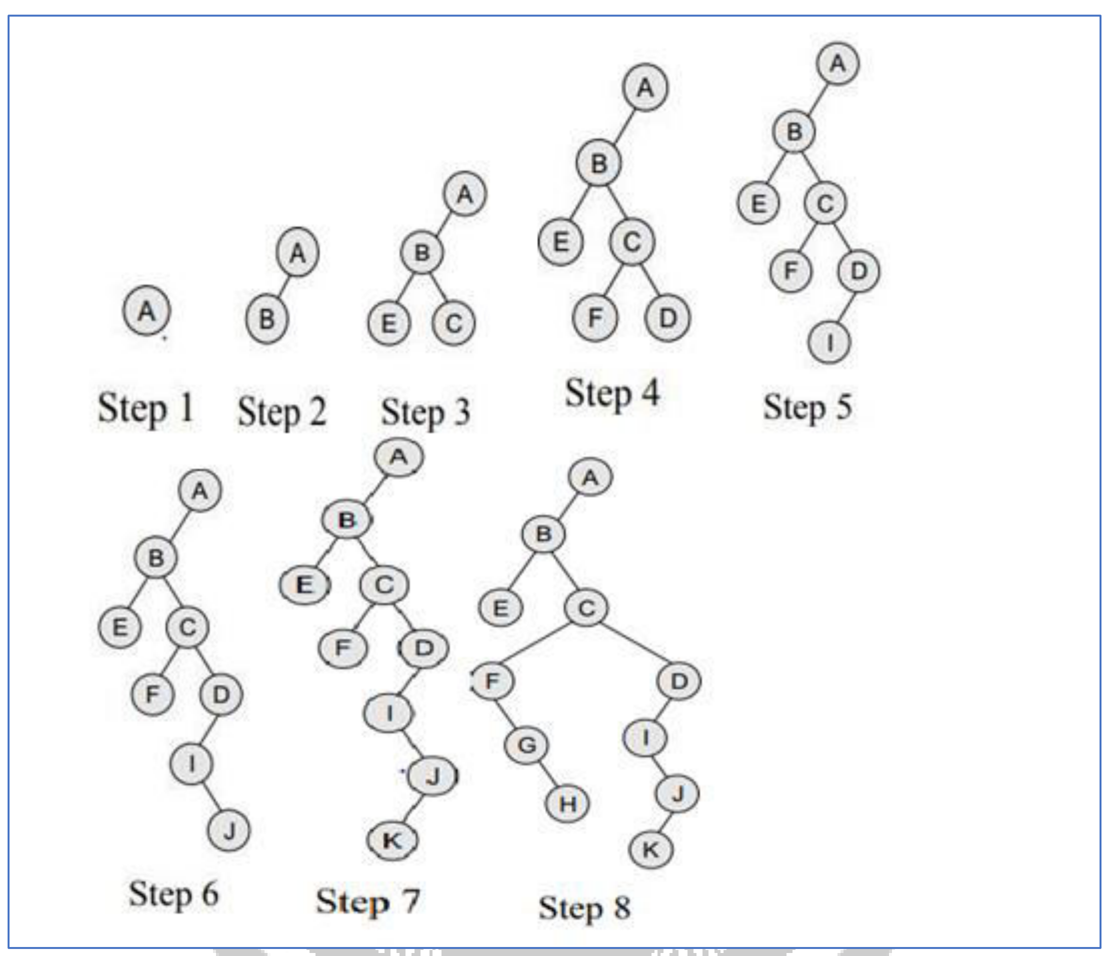
Step 3: Now process node B. Left child of B is E and its right child is C (right sibling in general tree). Step 4: Now process node C. Left child of C is F (leftmost child) and its right child is D (right sibling in generaltree).

Step 5: Now process node D. Left child of D is I (leftmost child). There will be no right child of D because it has no right sibling in the general tree.

Step 6: Now process node I. There will be no left child of I in the binary tree because I has no left child in the general tree. However, I has a right sibling J, so it will be added as the right child of I.

Step 7: Now process node J. Left child of J is K (leftmost child). There will be no right child of J because it has no right sibling in the general tree.

Step 8: Now process all the unprocessed nodes (E, F, G, H, K) in the same fashion, so the resultant binary tree can be given as follows.



ULAM, KANYAKU
OBSERVE OPTIMIZE OUTSPREAD

AVL TREES

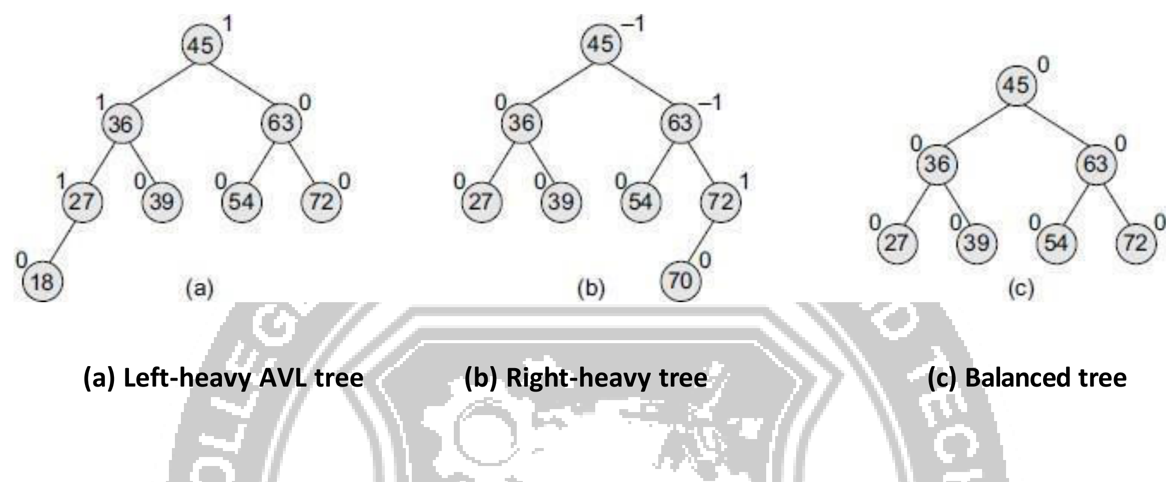
- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the ***AVL tree*** is also known as a ***height-balanced tree.***
- The key advantage of using an AVL tree is that it takes ***O(log n)*** time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to ***O(log n).***
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the ***Balance Factor.***
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of ***–1, 0, or 1*** is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a ***left-heavy tree.***
- If the balance factor of a node is 0, then it means that the height of the left sub- tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a ***right-heavy tree.***

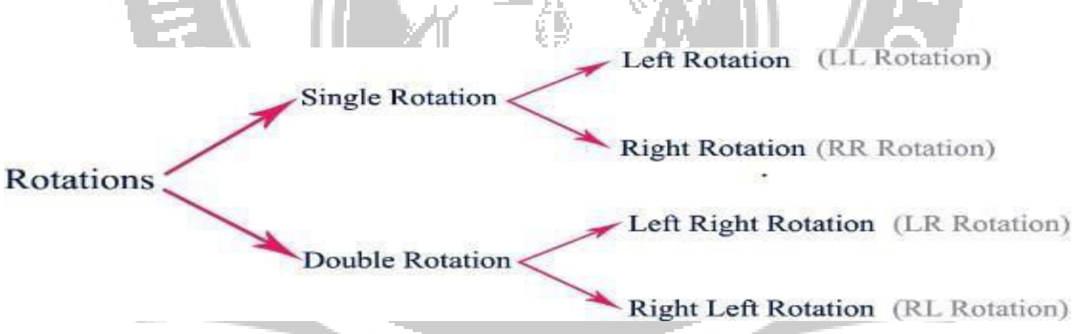
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

The trees given in given Fig. are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done.



(a) Left-heavy AVL tree **(b) Right-heavy tree** **(c) Balanced tree**

The tree is rebalanced by performing ***rotation*** at the critical node. There are four ***types of rotations***:



OPERATIONS on AVL TREE

1. Search

2. Insert

3. Delete

OBSERVE OPTIMIZE OUTSPREAD

1. Searching for a Node in an AVL Tree

Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. Due to the height-balancing of the tree, the search operation takes **O(log n**) time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step4: If botharenotmatching,then checkwhethersearchelementissmalleror larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree. **Step 6:** If search element is larger, then continue the search process in right subtree. **Step 7:** Repeat the same until we found exact element or we completed with a leaf node **Step 8:** Ifwereachto thenodewithsearchvalue,thendisplay"Element isfound" and terminate thefunction.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

2. Inserting a New Node in an AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

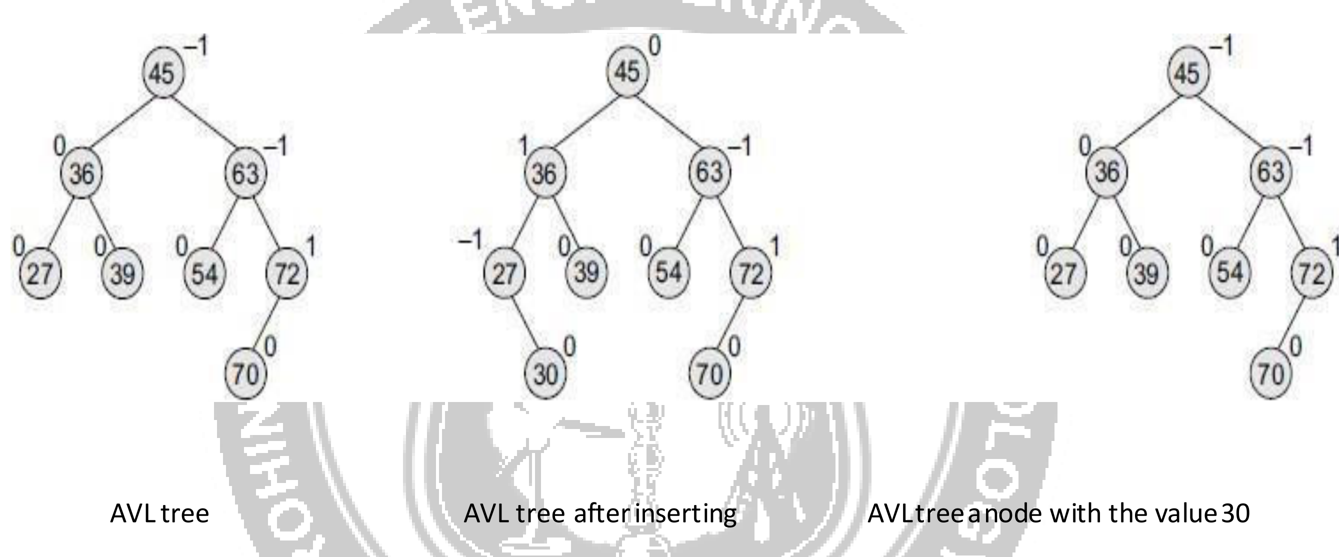
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the **Balance Factor** of every node.

Step 3: If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

Step 4: If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

Consider the AVL tree given in Fig. If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case and which shows the tree after inserting node 30.

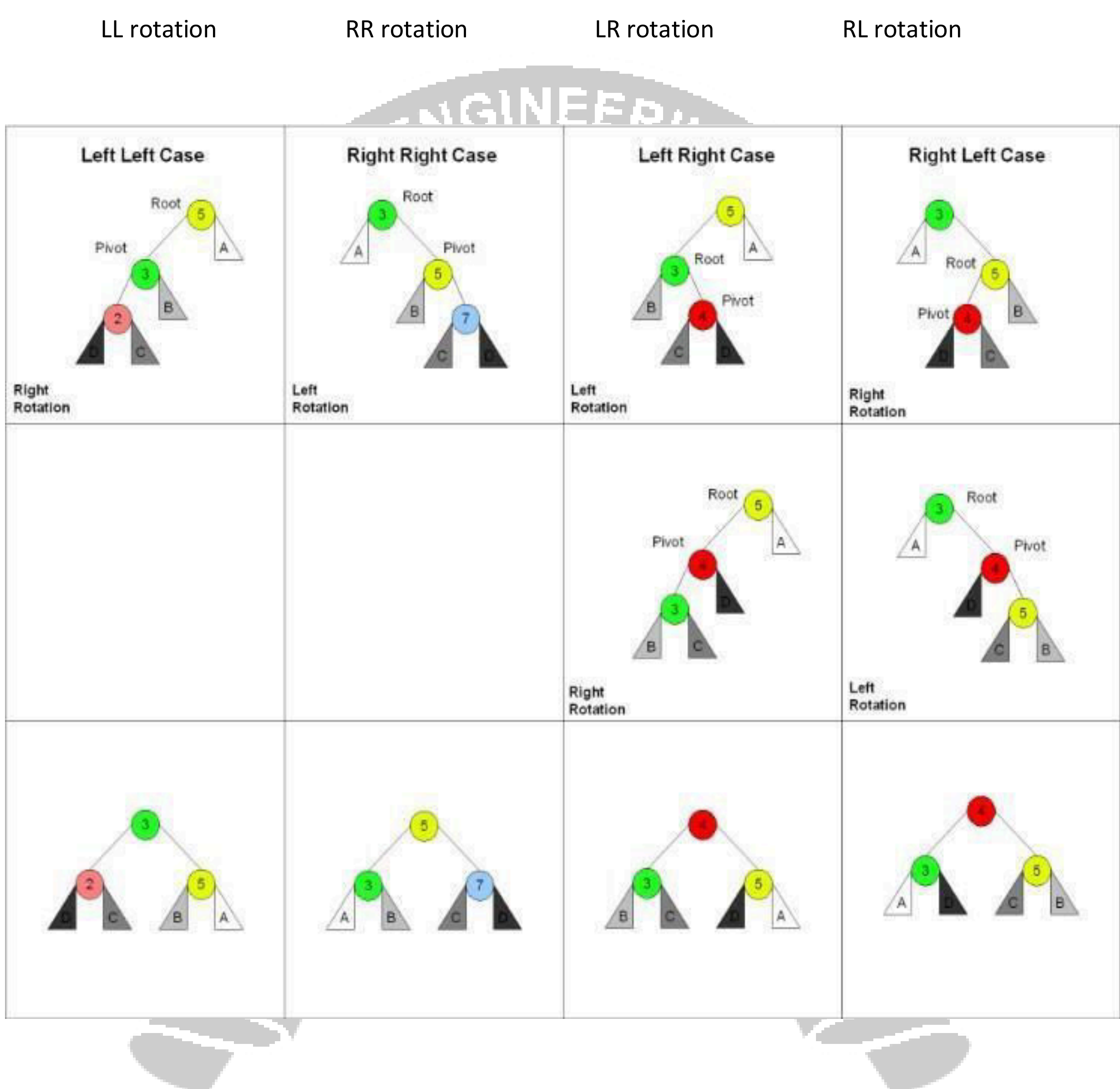


AVL tree AVL tree after inserting AVLtree a node with the value 30

The four categories of rotations are:

1. LL rotation The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
2. RR rotation The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
3. LR rotation The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
4. RL rotation The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

LL rotation RR rotation LR rotation RL rotation

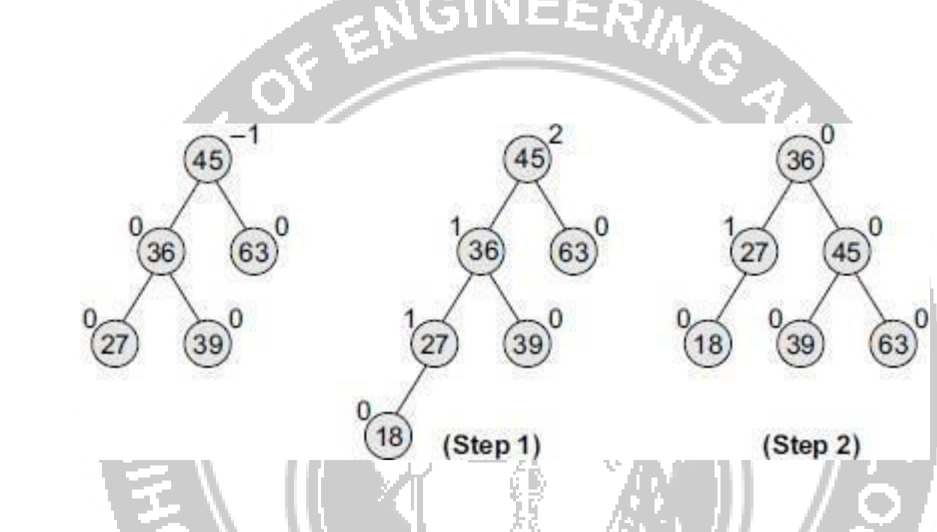


LL Rotation

Itisalsocalled asSingleleftrotation.ItRotates the edgeconnectingtherootand its right child in the binary tree

Example 1: ConsidertheAVLtreegiveninFig. andinsert18intoit.

Solution:

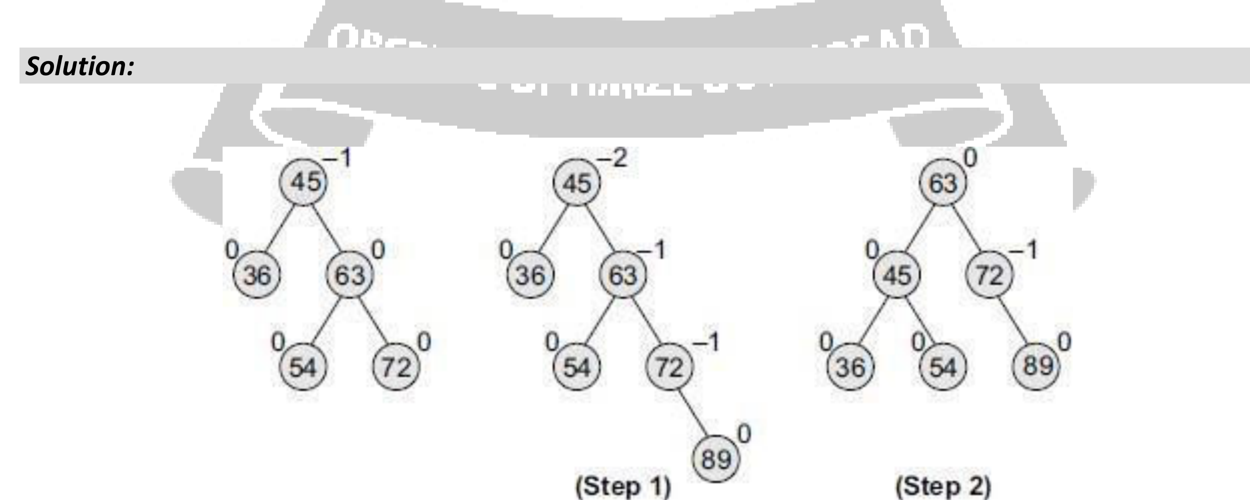


RR Rotation

It is also called as Single right rotation. It Rotates the edge connecting the root and its left child in the binary tree

Example 2: Consider the AVL tree given in Fig. and insert 89 into it.

Solution:



LR Rotation

- It is also called as Double left-right rotation.
- Combination of two rotations

  1. perform left rotation of the left sub-tree of root r
  2. perform right rotation of the new tree rooted at r

• It is performed after a new key is inserted into the right sub-tree of the left child of a tree whose root had the balance of +1 before the insertion

RL Rotation

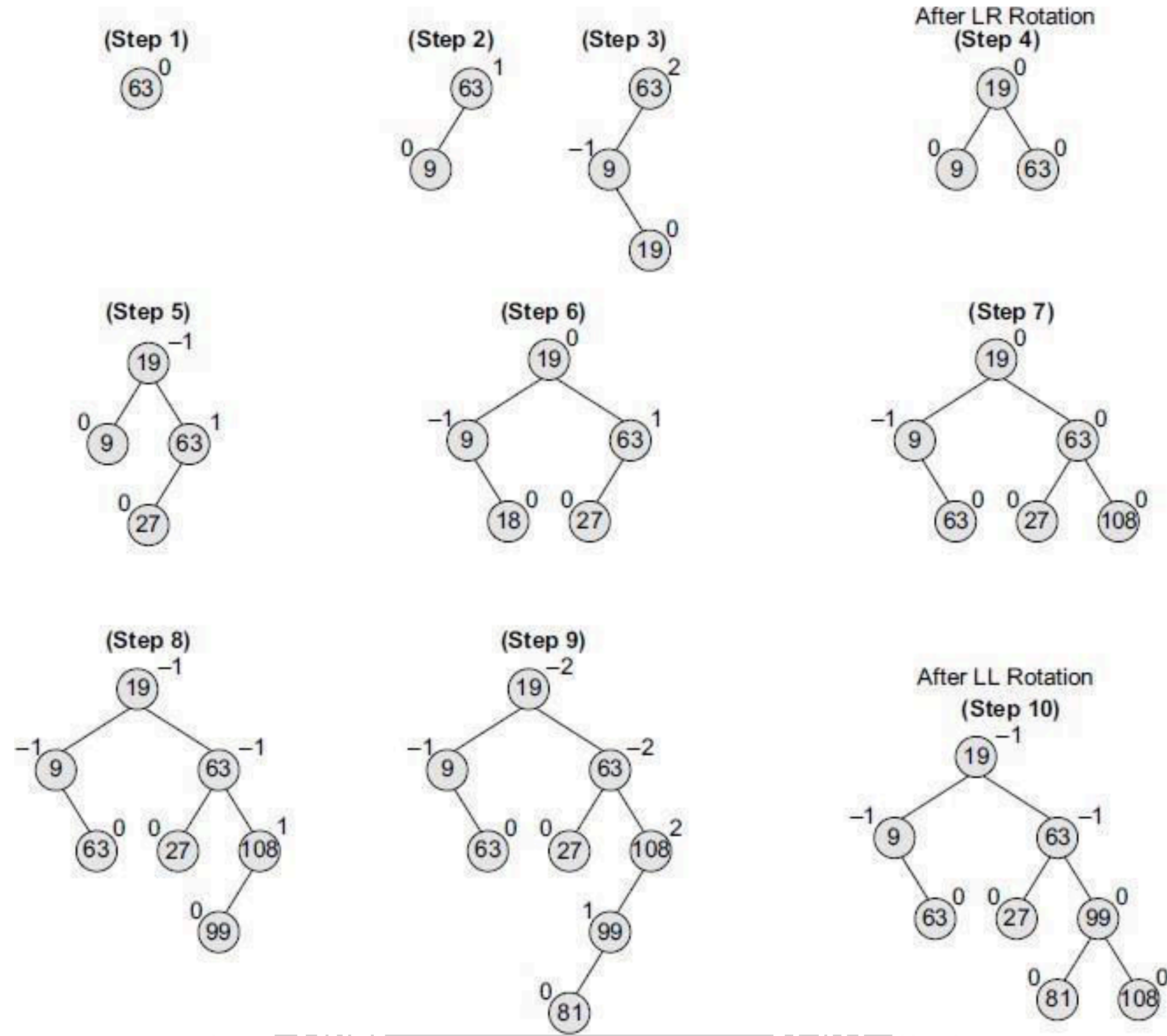
- It is also called as Double right-left rotation
- Combination of two rotations

  1. perform right rotation of the right sub-tree of root r
  2. perform left rotation of the new tree rooted at r

• It is performed after a new key is inserted into the left sub-tree of the right child of a tree whose root had the balance of -1 before the insertion

Example 1 Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

Solution:

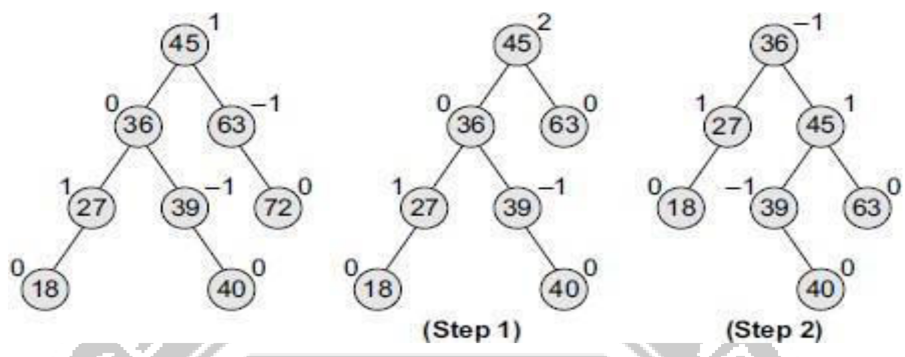


3. Deleting a Node from an AVL Tree

Deletion of a node in an AVL tree is similar to that of binary search trees. But, Deletion may disturb the AVL ness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation.

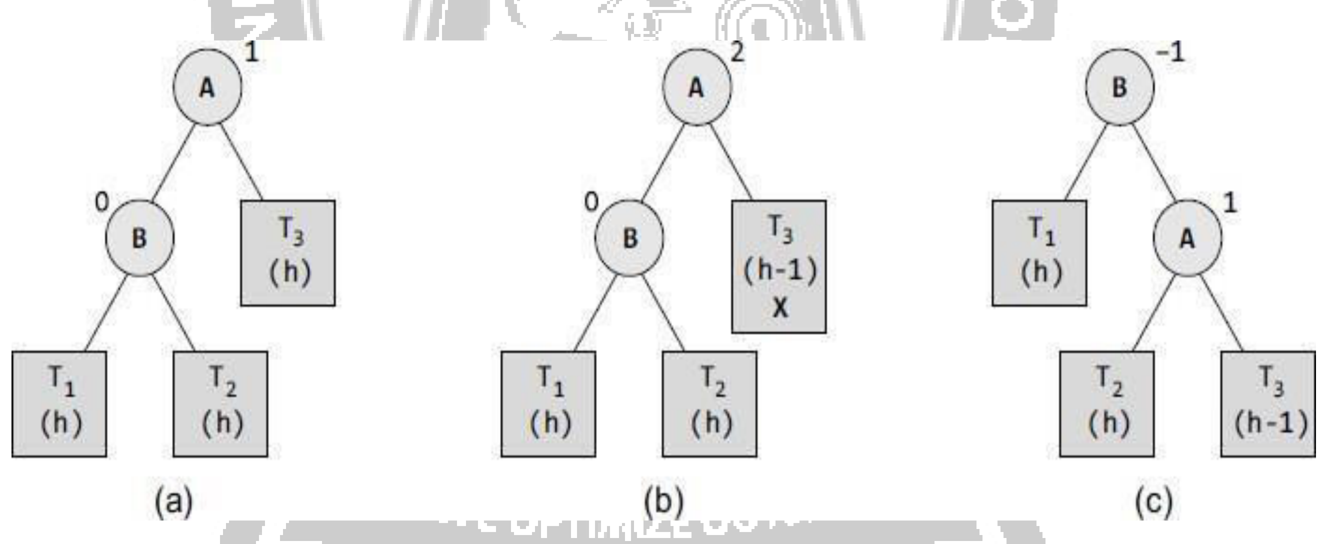
Example: Consider the AVL tree given in Fig. and delete 72 from it.

Solution:



R0 Rotation

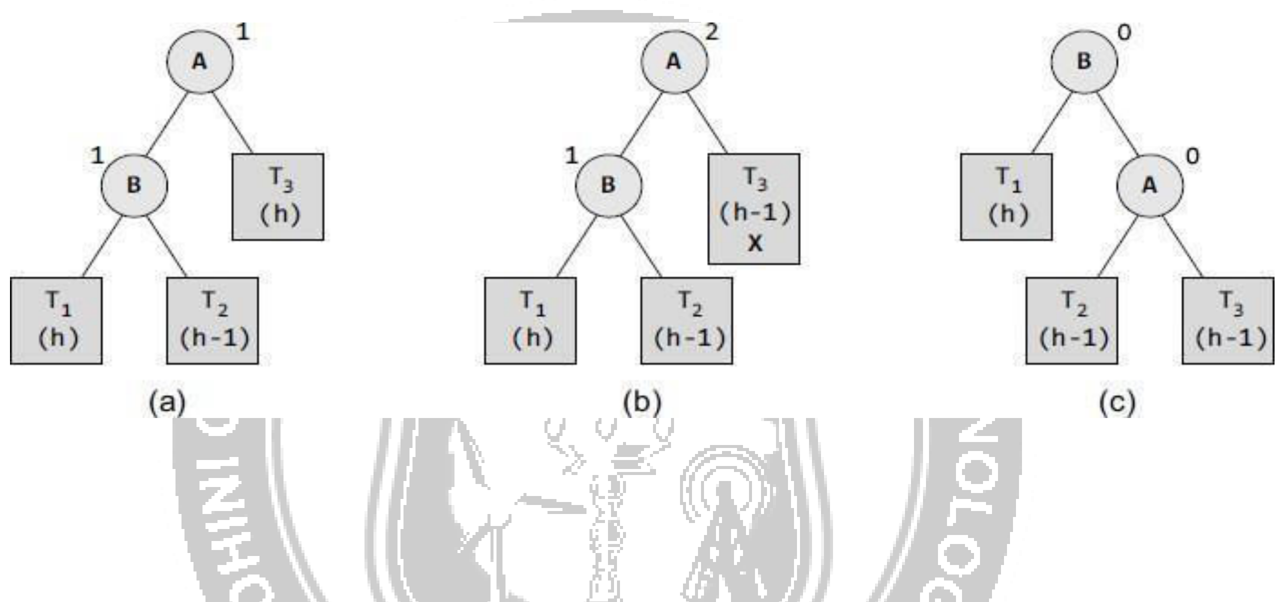
Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0.



Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 0, we apply R0 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

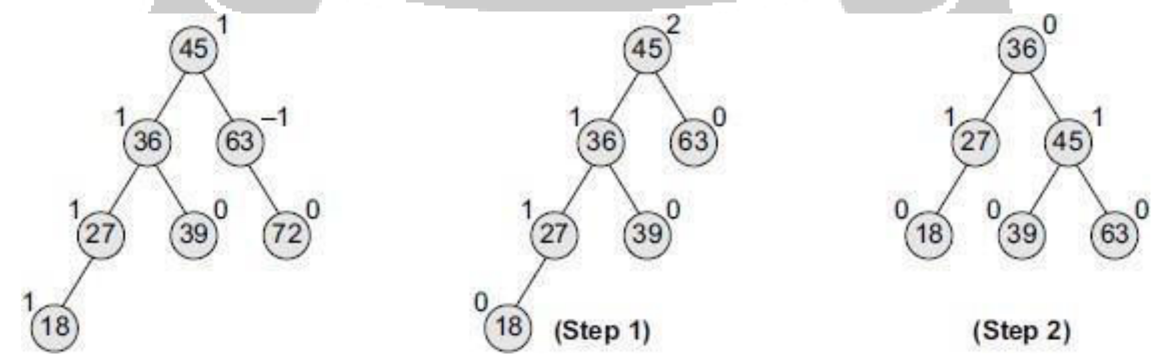
R1 Rotation

Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1. Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors. This is illustrated in Fig.

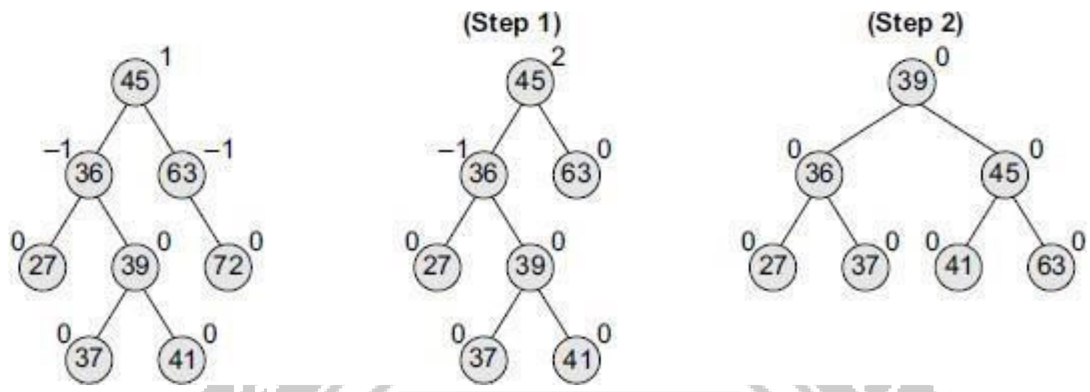


Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub- trees of A.

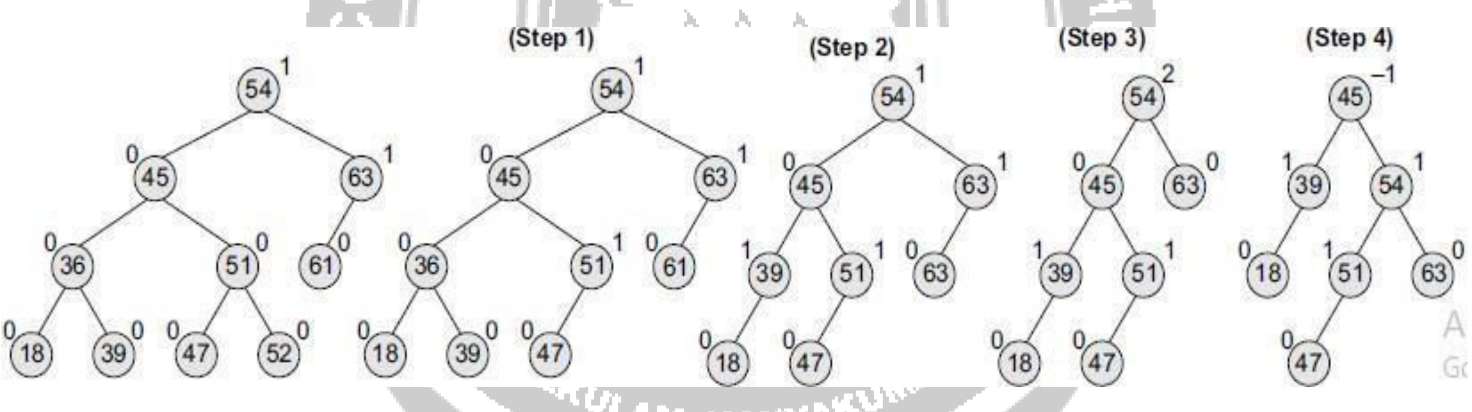
Example 1: Consider the AVL tree given in Fig. and delete 72 from it.



Example 2 Consider the AVL tree given in Fig. and delete 72 from it.



Example 3 Delete nodes 52, 36, and 61 from the AVL tree given in Fig.



Programming Example

```
#include <stdio.h>

#include <stdlib.h>

// Create Node

struct Node

{
```

OBSERVE OPTIMIZE OUTSPREAD

```
    int key;

    struct Node *left;

    struct Node *right;

     int height;

};

int max(int a, int b);

// Calculate height

int height(struct Node *N)

{

  if (N == NULL)

    return 0;

  return N->height;

}

int max(int a, int b)

{

  return (a > b) ? a : b;

}

// Create a node

struct Node *newNode(int key)
```

```
{
struct Node *node = (struct Node *)
 malloc(sizeof(struct Node));
 node->key = key;
 node->left = NULL;
 node->right = NULL;
 node->height = 1;
 return (node);
}
// Right rotate
struct Node *rightRotate(struct Node *y)
 {
  struct Node *x = y->left;
  struct Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;
  return x;
```

```
}

// Left rotate

struct Node *leftRotate(struct Node *x)

{

  struct Node *y = x->right;

  struct Node *T2 = y->left;

  y->left = x;

  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;

  y->height = max(height(y->left), height(y->right)) + 1;

  return y;

}

// Get the balance factor

int getBalance(struct Node *N)

{

  if (N == NULL)

    return 0;

  return height(N->left) - height(N->right);

}
```

```
// Insert node

struct Node *insertNode(struct Node *node, int key)

{

  // Find the correct position to insertNode the node and insertNode it

  if (node == NULL)

          return (newNode(key));

  if (key < node->key)

            node->left = insertNode(node->left, key);

  else if (key > node->key)

           node->right = insertNode(node->right, key);

  else

          return node;

  // Update the balance factor of each node and

  // Balance the tree

  node->height = 1 + max(height(node->left),

  height(node->right));

  int balance = getBalance(node);

  if (balance > 1 && key < node->left->key)
```

```
  return rightRotate(node);



  if (balance < -1 && key > node->right->key)

    return leftRotate(node);

  if (balance > 1 && key > node->left->key) {

    node->left = leftRotate(node->left);

    return rightRotate(node);

  }

  if (balance < -1 && key < node->right->key)

{

    node->right = rightRotate(node->right);

    return leftRotate(node);

  }

  return node;

}


struct Node *minValueNode(struct Node *node)

{

  struct Node *current = node;
```

```
  while (current->left != NULL)
    current = current->left;
  return current;
}
// Delete a nodes
struct Node *deleteNode(struct Node *root, int key)
{
  // Find the node and delete it
  if (root == NULL)
        return root;
  if (key < root->key)
        root->left = deleteNode(root->left, key);
  else if (key > root->key)
        root->right = deleteNode(root->right, key);
  else
{
        if ((root->left == NULL) || (root->right == NULL))
        {
        struct Node *temp = root->left ? root->left : root->right;
```

```
        if (temp == NULL) {
        temp = root;
          root = NULL;
        }
        else
          *root = *temp;
        free(temp);
 }
else {
   struct Node *temp = minValueNode(root->right);
   root->key = temp->key;
   root->right = deleteNode(root->right, temp->key);
  }
 }

 if (root == NULL)
   return root;
 // Update the balance factor of each node and
```

```
  // balance the tree

  root->height = 1 + max(height(root->left),  height(root->right));

  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->left) >= 0)

    return rightRotate(root);

  if (balance > 1 && getBalance(root->left) < 0) {

    root->left = leftRotate(root->left);

    return rightRotate(root);

  }

  if (balance < -1 && getBalance(root->right) <= 0)

    return leftRotate(root);

  if (balance < -1 && getBalance(root->right) > 0) {

    root->right = rightRotate(root->right);

    return leftRotate(root);

  }

  return root;

}

// Print the tree

void printPreOrder(struct Node *root) {
```

```
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  } }
int main() {
  struct Node *root = NULL;
  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);
  printPreOrder(root);
  root = deleteNode(root, 3);
  printf("\nAfter deletion: ");
  printPreOrder(root);
  return 0;}
```

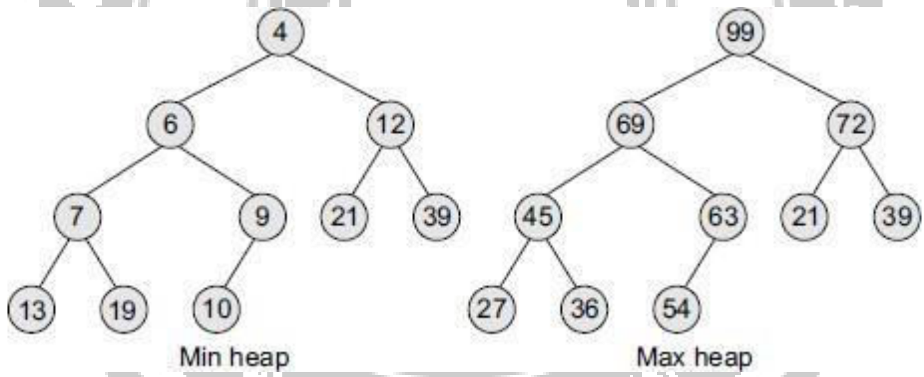
HEAPS

BINARY HEAPS

A binary heap is a complete binary tree in which every node satisfies the heap property which states that:

If B is a child of A, then key(A) ≥ key(B)

- This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a max-heap.
- Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a min-heap.



The properties of binary heaps are given as follows:

Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position 2i and its right child at position 2i+1. Conversely, an element at position ihas its parent stored at position i/2.

- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as log2n, where n is the number of elements.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

OPERATIONS:

1.Insertion

2.Deletion

Inserting a New Element in a Binary Heap

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.

2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well. To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are

swapped and Step 2 is repeated.

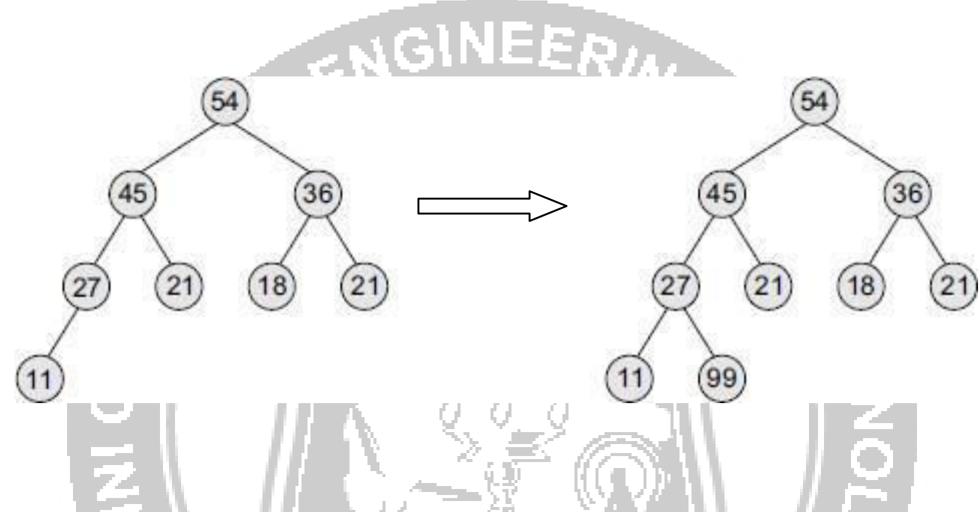
```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Algorithm to insert an element in a max heap

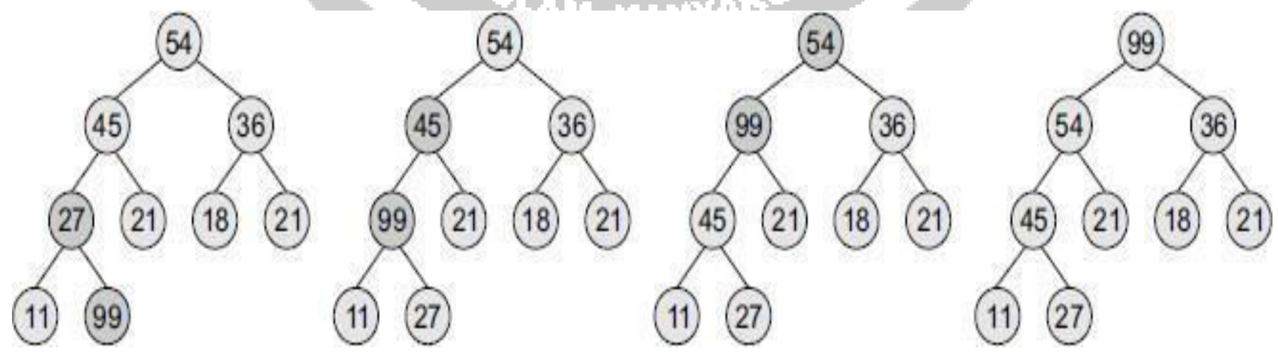
Example 1 Consider the max heap given in Fig. and insert 99 in it.

Solution

The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig. Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well

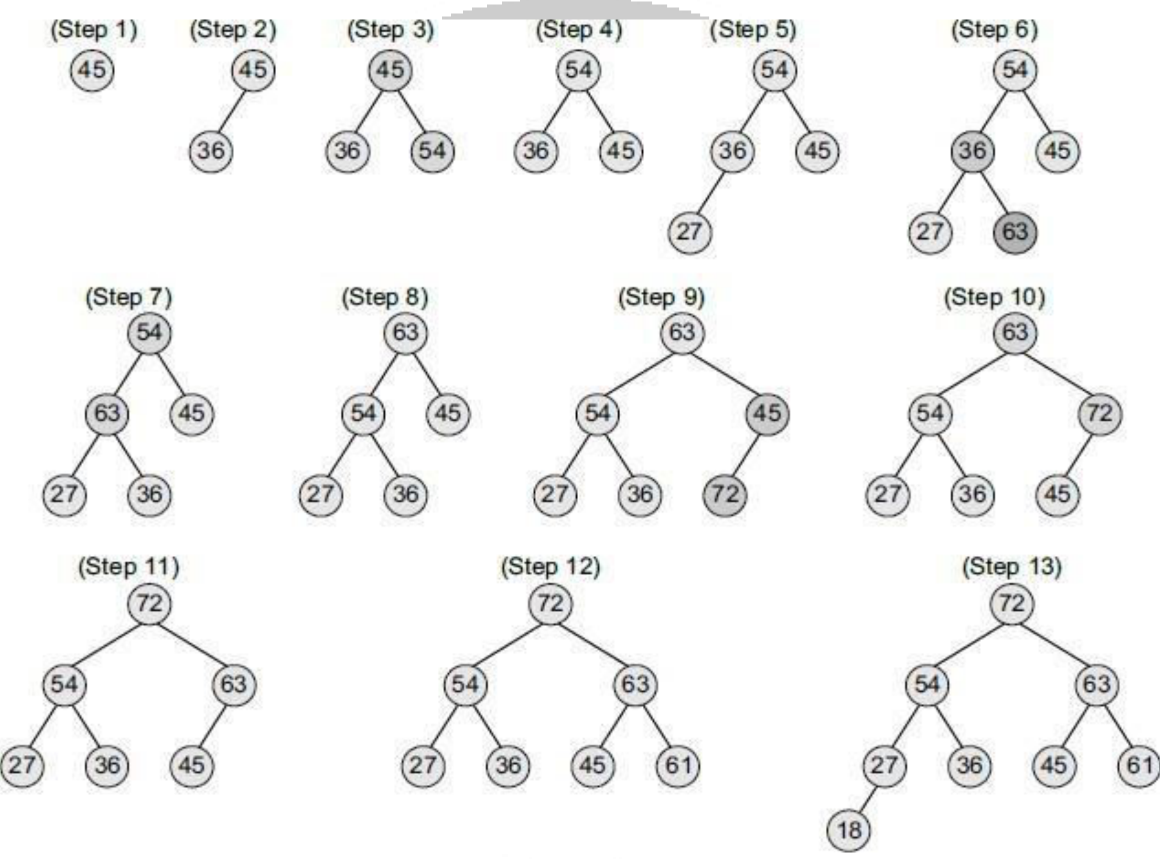


Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Fig.



Example 2 Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

Solution



OBSERVE OPTIMIZE OUTSPREAD

2. Deleting an Element from a Binary Heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node’s value with the last node’s value so that H is still a complete binary tree but not necessarily a heap.

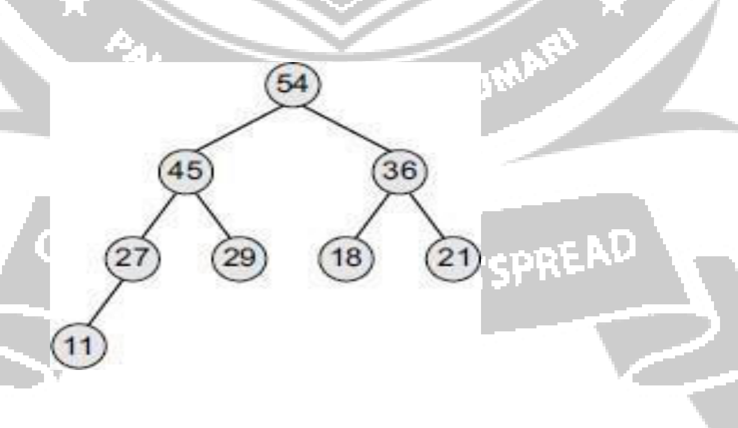
2. Delete the last node.

3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children). Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.

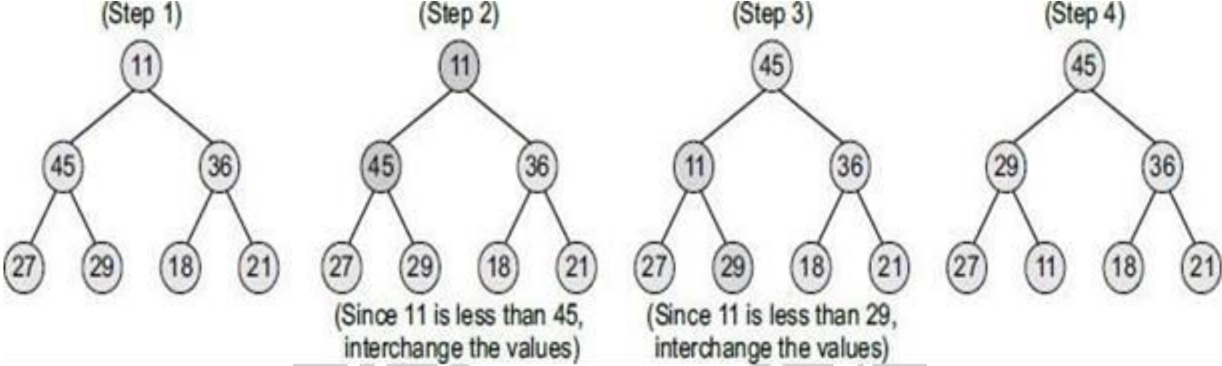
```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
         HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Algorithm to delete the root element from a max heap

Example 1 Consider the max heap H shown in Fig. 12.8 and delete the root node's value.



Solution



Applications of Binary Heaps

Binary heaps are mainly applied for

1. Sorting an array using heapsort algorithm.

2. Implementing priority queues.

Binary Heap Implementation of Priority Queues

- A priority queue is similar to a queue in which an item is dequeued (or removed) from the front. However, unlike a regular queue, in a priority queue the logical order of elements is determined by their priority. While the higher priority elements are added at the front of the queue, elements with lower priority are added at the rear.
- Though we can easily implement priority queues using a linear array, but we should first consider the time required to insert an element in the array and then sort it. We need O(n) time to insert an element and at least O(n log n) time to sort the array. Therefore, a better way to implement a priority queue is by using a binary heap which allows both enqueue and dequeue of elements in O(log n) time.

BINOMIAL HEAPS

A binomial heap H is a set of binomial trees that satisfy the binomial heap properties. First, let us discuss what a binomial tree is.

A binomial tree is an ordered tree that can be recursively defined as follows:

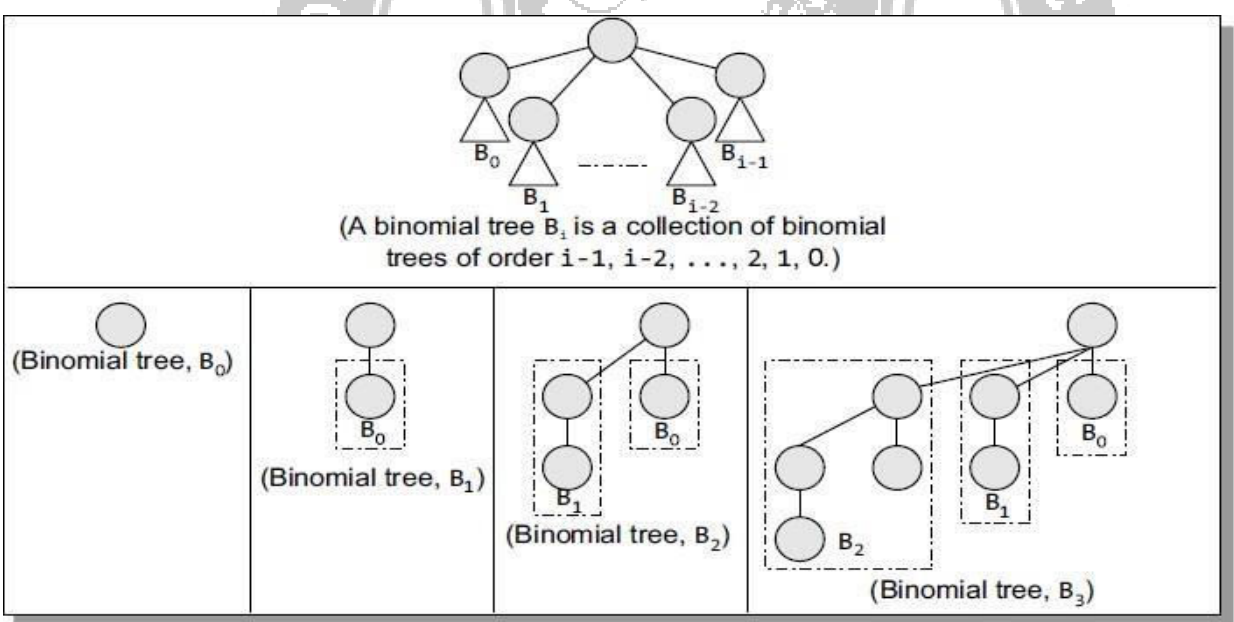
- A binomial tree of order 0 has a single node.

- A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order i–1, i–2, ..., 2, 1, and 0.

- A binomial tree Bi has 2i nodes.

- The height of a binomial tree Bi is i.

Look at Fig. which shows a few binomial trees of different orders. We can construct a binomial tree Bi from two binomial trees of order Bi–1 by linking them together in such a way that the root of one is the leftmost child of the root of another.



A binomial heap H is a collection of binomial trees that satisfy the following properties:

- Every binomial tree in H satisfies the minimum heap property (i.e., the key of a node is either greater than or equal to the key of its parent).

- There can be one or zero binomial trees for each order including zero order. According to the first property, the root of a heap-ordered tree contains the smallest key in the tree. The second property, on the other hand, implies that a binomial heap H having N nodes contains at most log (N + 1) binomial trees.

3. FIBONACCI HEAPS

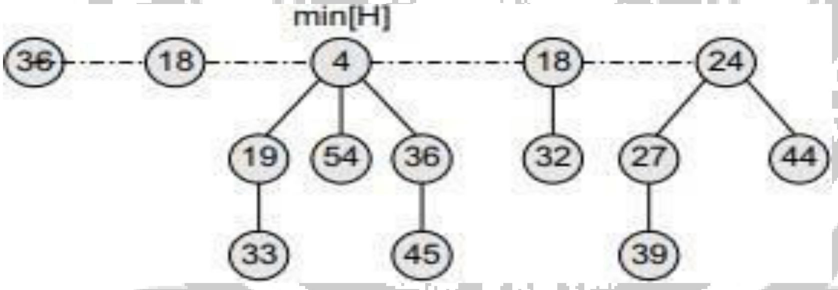
A Fibonacci heap is a collection of trees. It is loosely based on binomial heaps. If neither the decrease-value nor the delete operation is performed, each tree in the heap is like a binomial tree. Fibonacci heaps differ from binomial heaps as they have a more relaxed structure, allowing improved asymptotic time bounds.

1. Structure of Fibonacci Heaps

Although a Fibonacci heap is a collection of heap-ordered trees, the trees in a Fibonacci heap are not constrained to be binomial trees. That is, while the trees in a binomial heap are ordered, those within Fibonacci heaps are rooted but unordered.

Each node in the Fibonacci heap contains the following pointers:

- a pointer to its parent, and
- a pointer to any one of its children

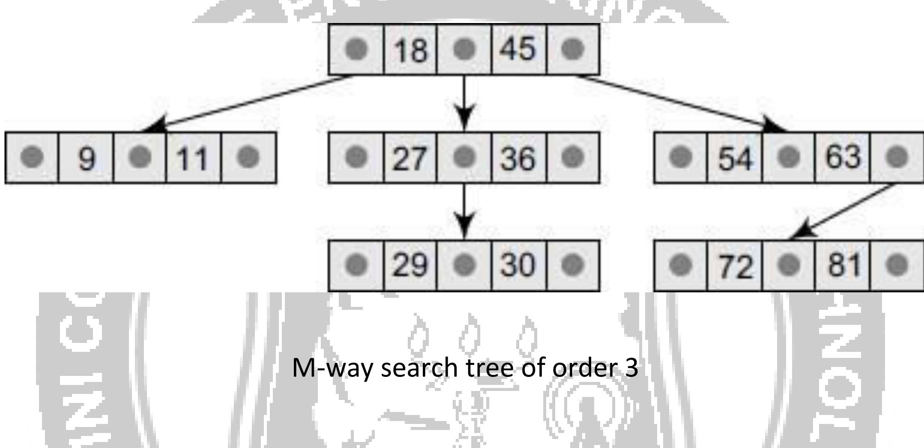


Applications of Heaps

- Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.
- Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

MULTI-WAY SEARCH TREES

We have discussed that every node in a binary search tree contains one value and two pointers, left and right, which point to the node's left and right sub-trees, respectively. The same concept is used in an M-way search tree which has M – 1 values per node and M subtrees. In such a tree, M is called the degree of the tree. Note that in a binary search tree M = 2, so it has one value and two sub-trees. In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2.



M-way search tree of order 3

B TREES

- B tree is a specialized M-way tree that is widely used for disk access. B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees. B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.
- B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree.

In addition, it has the following **properties**:

1. Every node in the B tree has at most (maximum) m children.

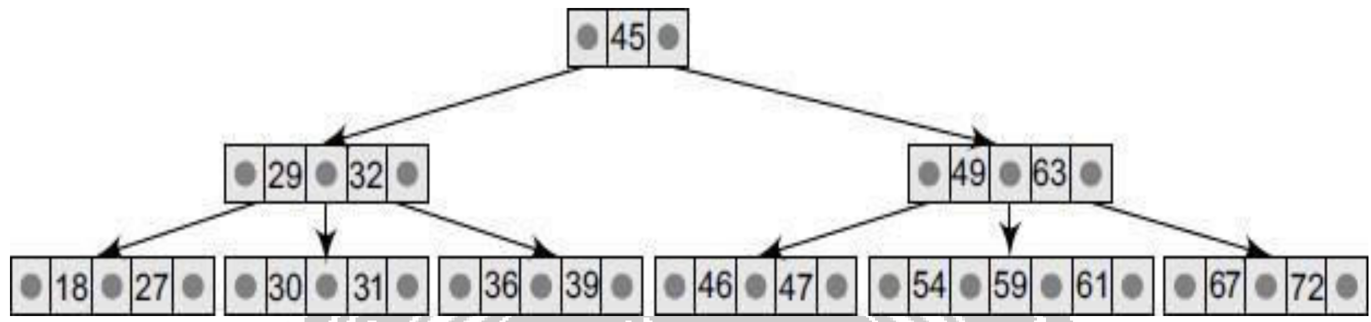
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.

3. The root node has at least two children if it is not a terminal (leaf) node.

4. All leaf nodes are at the same level.

1. Searching for an Element in a B Tree

Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree iven in Fig. To search for 59



Step 1. we begin at the root node. The root node has a value 45 which is less than 59.

Step 2. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since 49 < 59 < 63,

Step 3. We traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.

The running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B tree takes O(logt n) time to execute.

2. Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

Procedure:

Step 1. Search the B tree to find the leaf node where the new key value should be inserted.

Step 2. If the leaf node is not full, that is, it contains less than m–1 key values, then insert the new element in the node keeping the node's elements ordered.

Step 3. If the leaf node is full, that is, the leaf node already contains m–1 key values, then

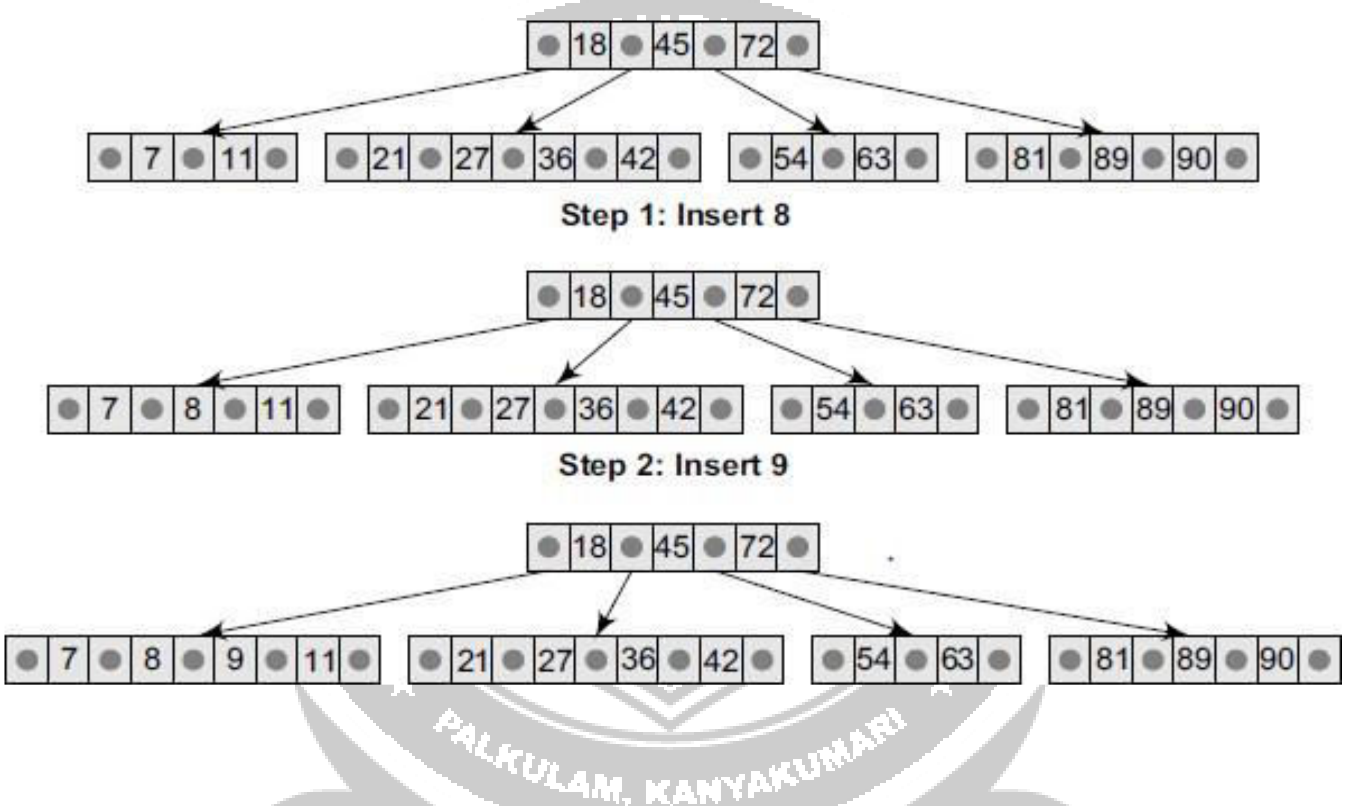
OBSERVE OPTIMIZE OUTSPREAD

(a) insert the new value in order into the existing set of keys,

(b) split the node at its median into two nodes (note that the split nodes are half full), and

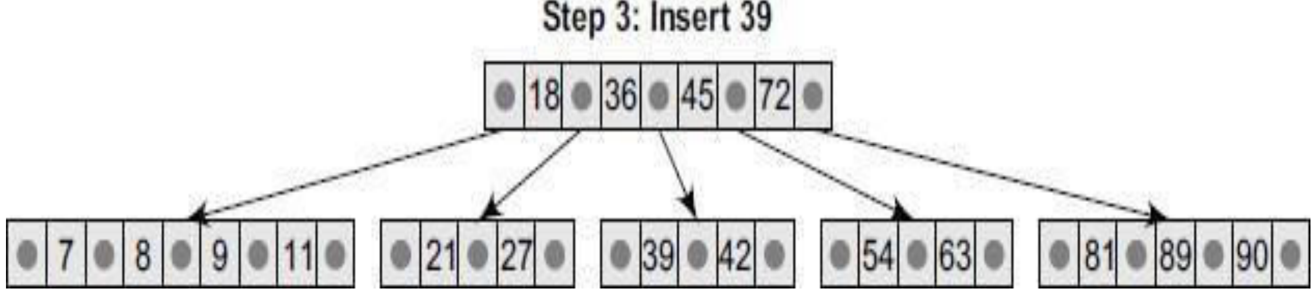
(c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps

Example 1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

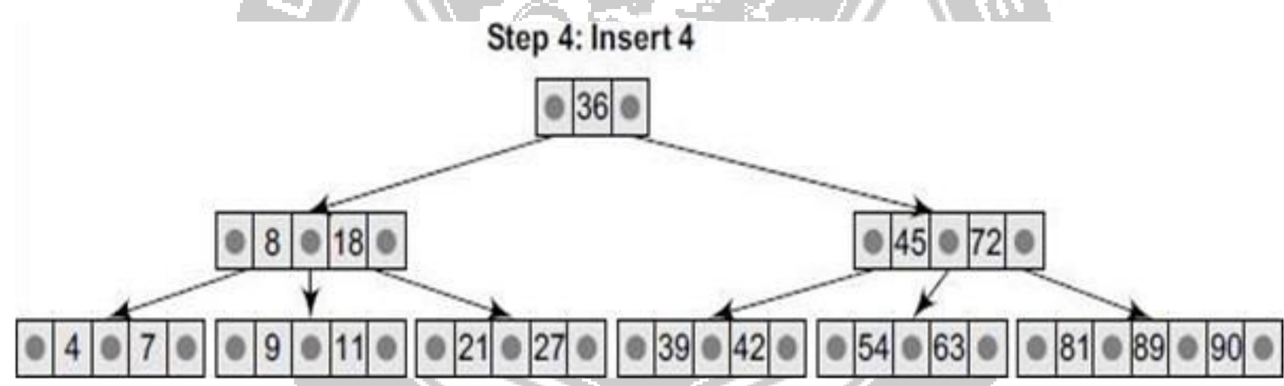


Till now, we have easily inserted 8and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value).

The ordered set of values isgivenas21, 27, 36,39, and42. The median value is 36, so push 36 into its parent's node and split the leaf nodes



Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.



3. Deleting an Element from a B Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

Step 1. Locate the leaf node which has to be deleted.

Step 2. If the leaf node contains more than the minimum number of key values (more than m/2 elements), then delete the value.

Step 3. Else if the leaf node does not contain m/2 elements, then fill the node by taking an element

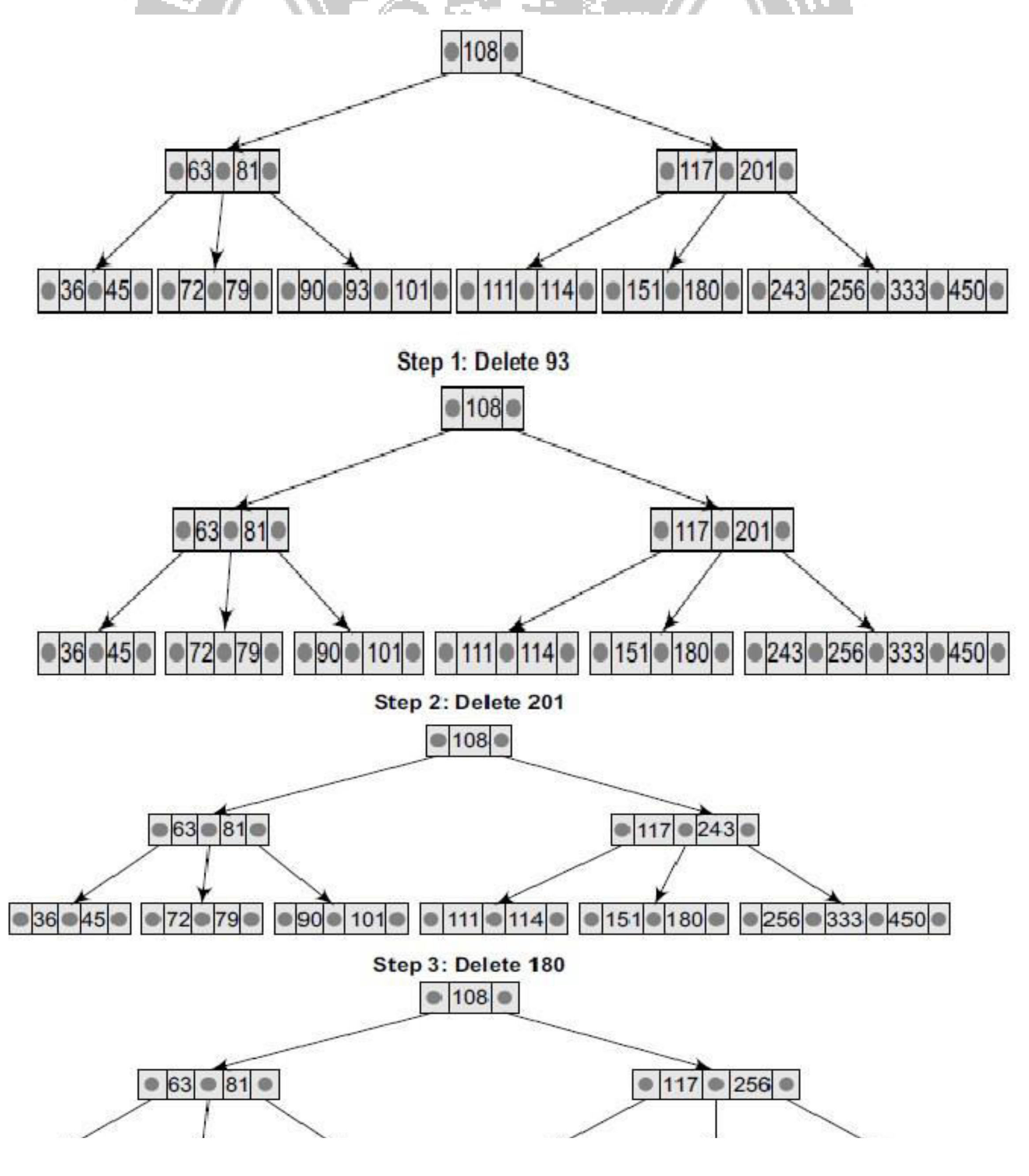
either from the left or from the right sibling.

(a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

(b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

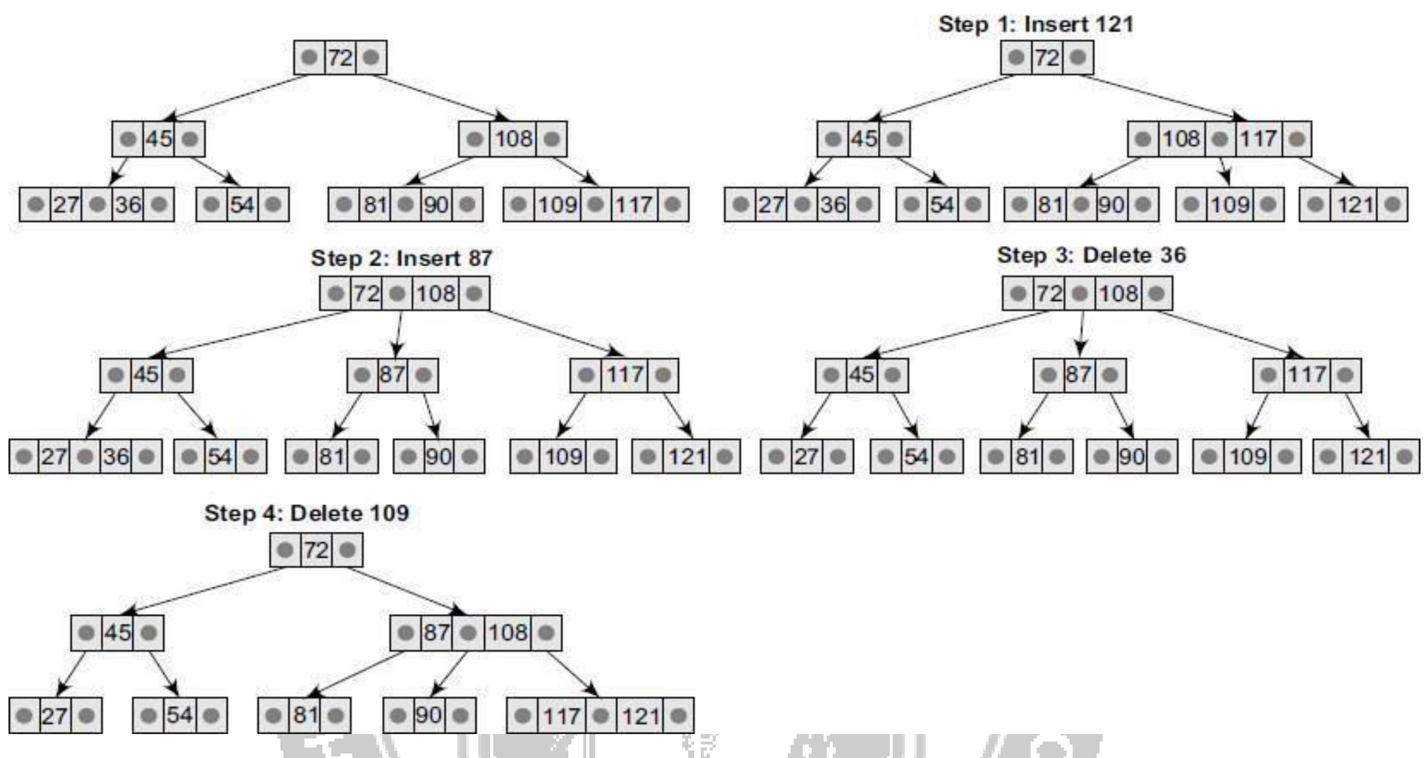
Step 4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent

Example 1 Consider the following B-tree of order 5 and delete values 93, 201, 180, and 72 from it

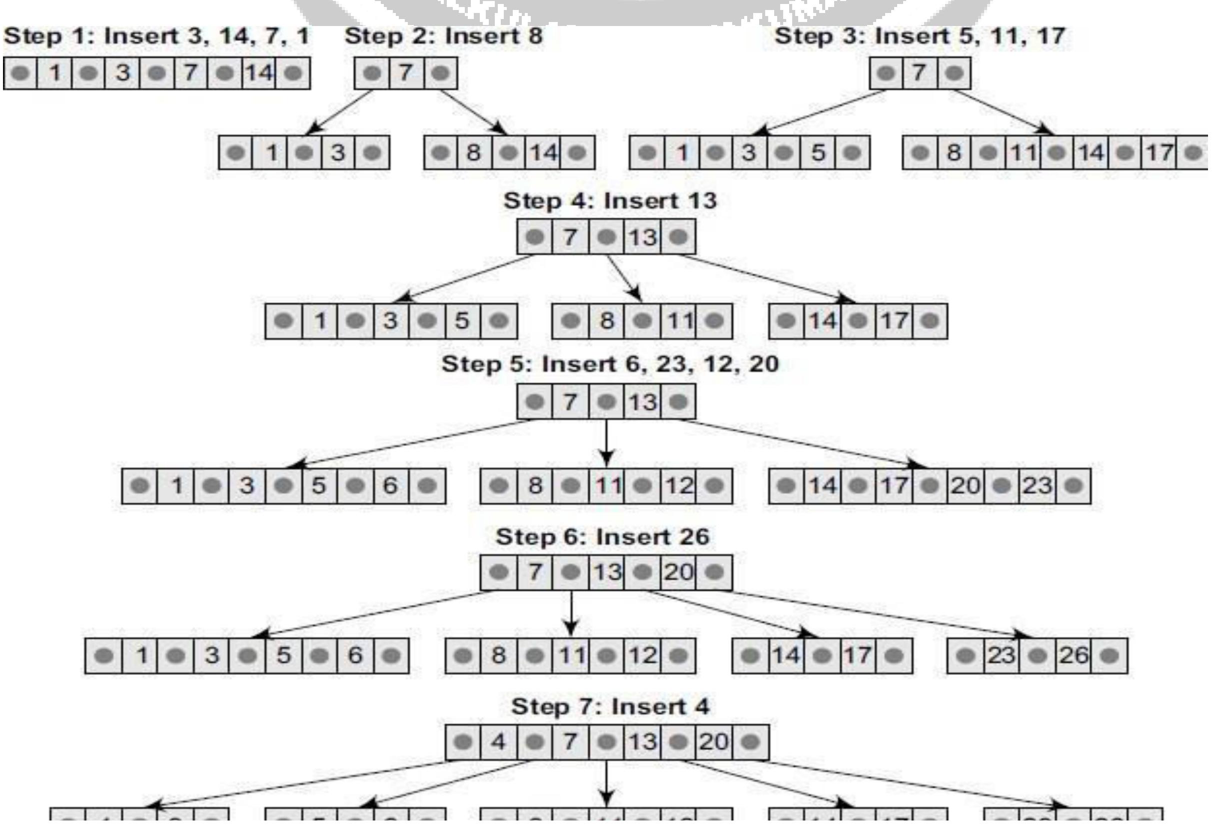


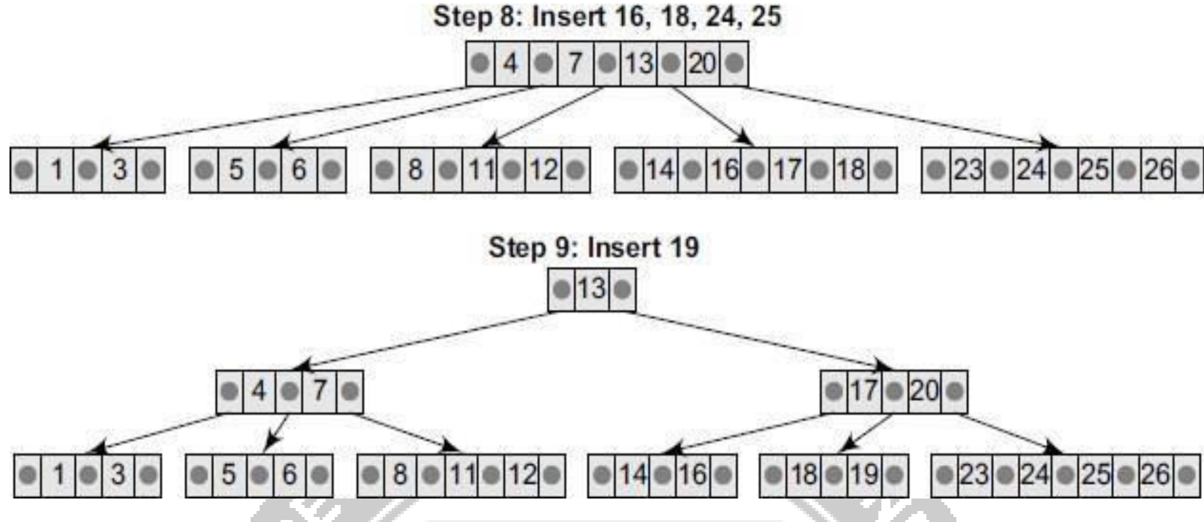
Example 2 Consider the B tree of order 3 given below and perform the following operations:

(a) insert 121, 87 and then (b) delete 36, 109.



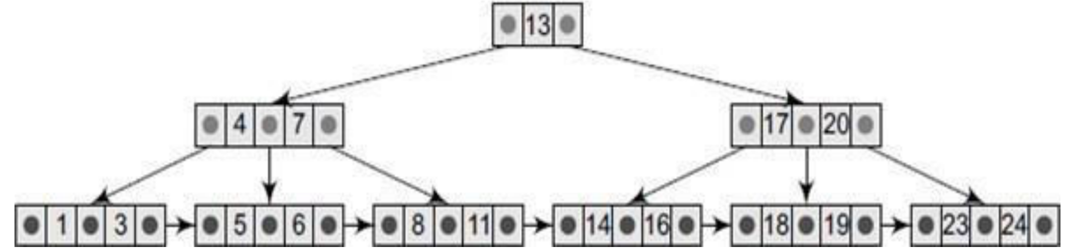
Example 11.4 Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19





B+ TREES

- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes.
- The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient.
- Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.
- B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called index nodes or i-nodes and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item. Figure shows a B+ tree of order 3.



Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index.

The advantages of B+ trees can be given as follows:

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

1. Inserting a New Element in a B+ Tree

- A new element is simply added in the leaf node if there is space for it. But if the data node in the tree where insertion has to be done is full, then that node is split into two nodes. This calls for adding a new index value in the parent index node so that future queries can arbitrate between the two new nodes.
- However, adding the new index value in the parent node may cause it, in turn, to split.
- In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node. If the root node splits, a new leaf node is created and the tree grows by one level.

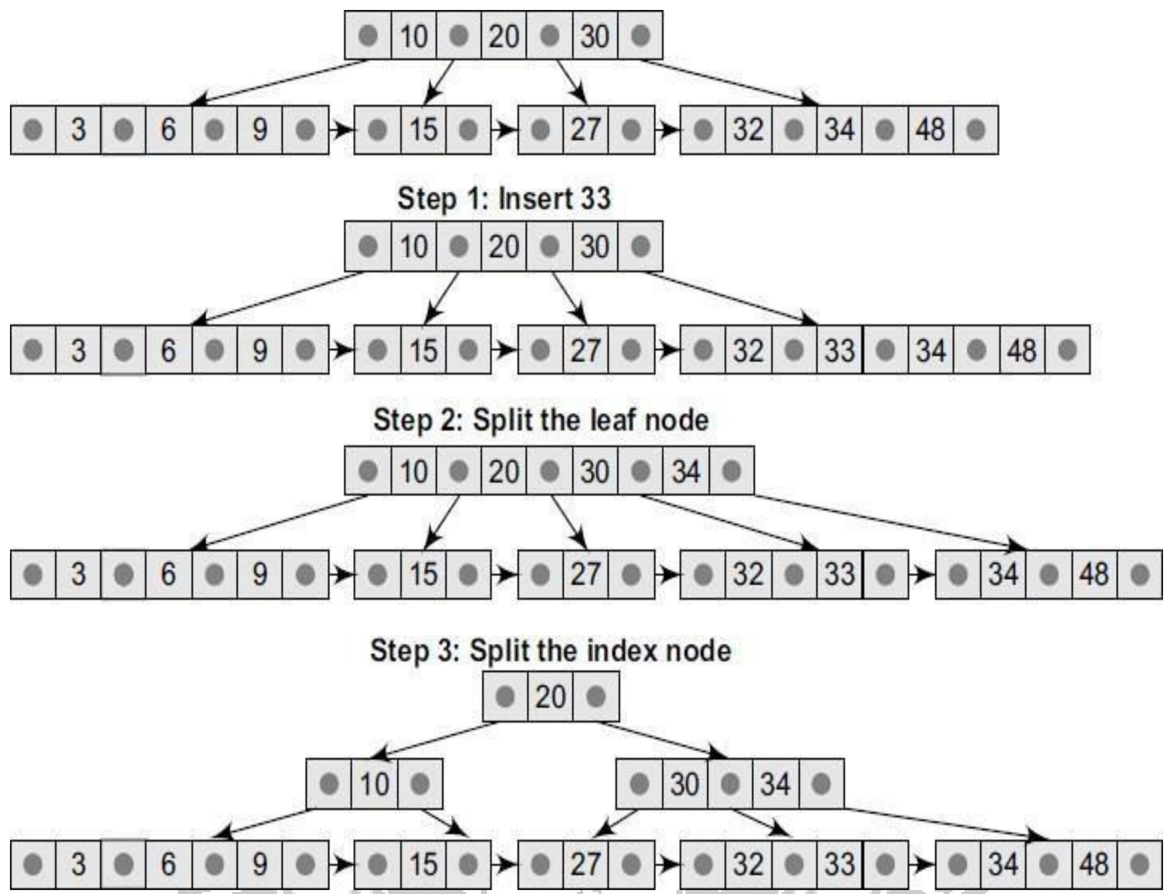
The steps to insert a new node in a B+ Tree are summarized below

Step 1: Insert the new node as the leaf node.

Step 2: If the leaf node overflows, split the node and copy the middle element to next index node.

Step 3: If the index node overflows, split that node and move the middle element to next index page.

Example 1. Consider the B+ tree of order 4 given and insert 33 in it.



2 Deleting an Element from a B+ Tree

- As in B trees, deletion is always done from a leaf node. If deleting a data element leaves that node empty, then the neighboring nodes are examined and merged with the under full node.
- This process calls for the deletion of an index value from the parent index node which, in turn, may cause it to become empty. Similar to the insertion process, deletion may cause a merge-delete wave to run from a leaf node all the way up to the root. This leads to shrinking of the tree by one level.

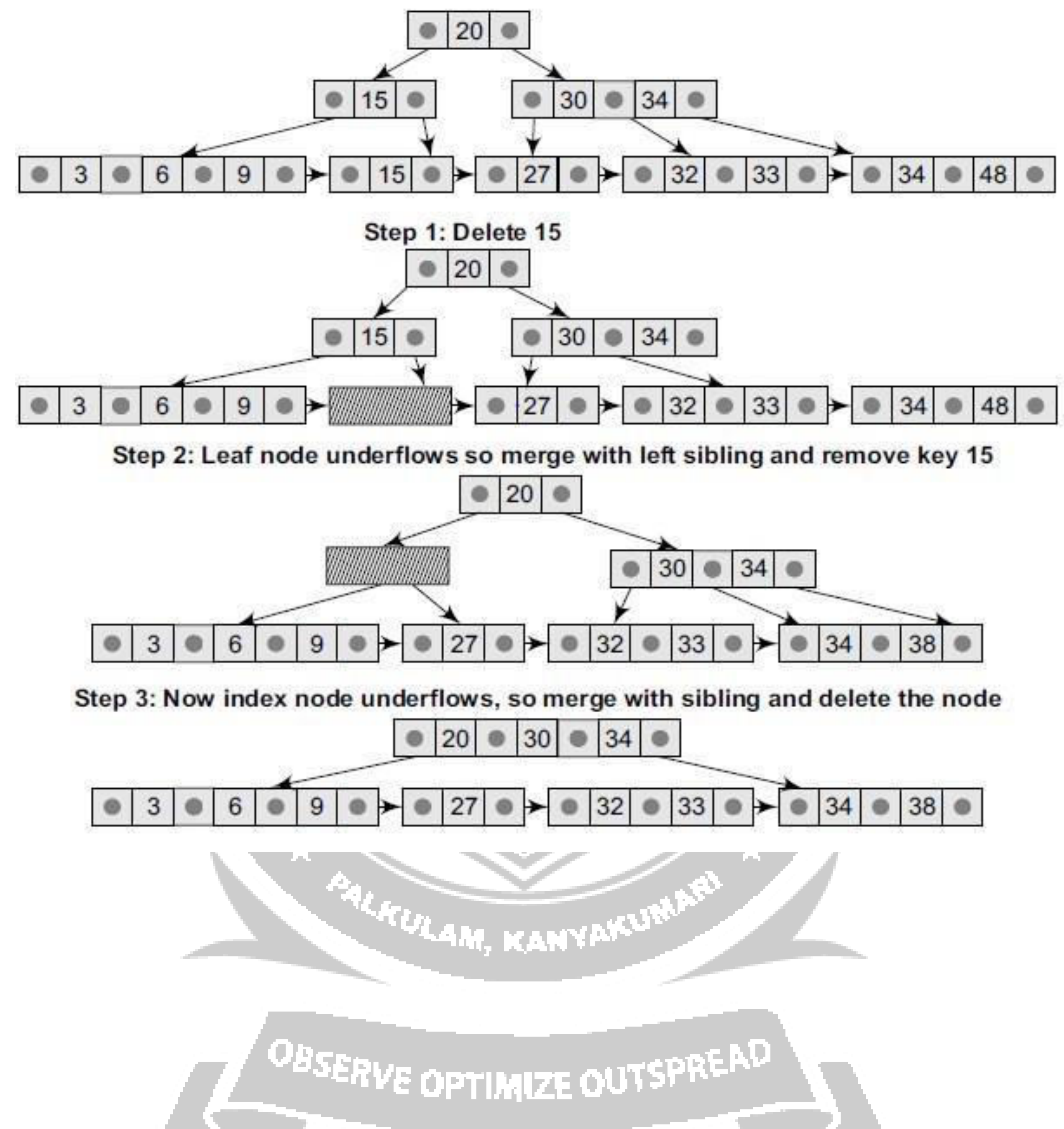
The steps to delete a node from a B+ tree are summarized below

Step 1: Delete the key and data from the leaves.

Step 2: If the leaf node underflows, merge that node with the sibling and delete the key in between them.

Step 3: If the index node underflows, merge that node with the sibling and move down the key in between them.

Example 1: Consider the B+ tree of order 4 given below and delete node 15 from it.

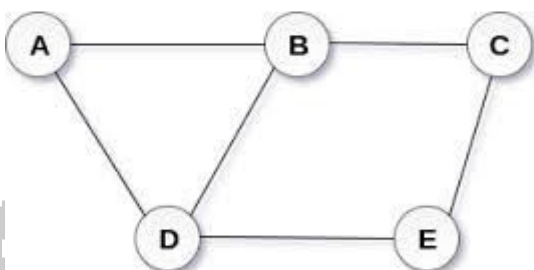


PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

GRAPHS

DEFINITION

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

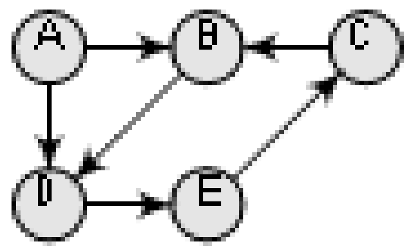


In the above graph, V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C,E)}.

There are five vertices or nodes and six edges in the graph.

DIRECTED AND UNDIRECTED GRAPHS

- In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
- In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).



Directed Graphs

OBSERVE OPTIMIZE OUTSPREAD

GRAPH TERMINOLOGY

Adjacent nodes or neighbours

For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end- points and are said to be the adjacent nodes or neighbours.

Degree of a node

Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Closed path

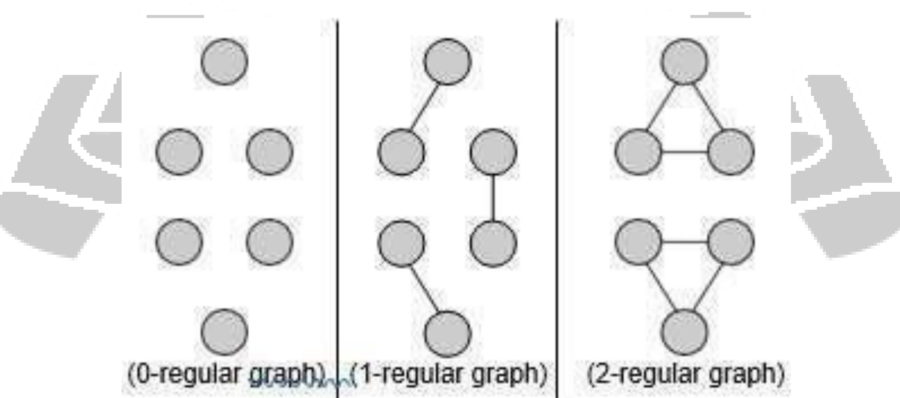
A path P is known as a closed path if the edge has the same end-points. That is, if vo=vn.

Simple path

A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0= vn then the path is called a closed simple path.

Cycle

A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).



Regular Graph

Types of Graphs

Connected graph

A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.

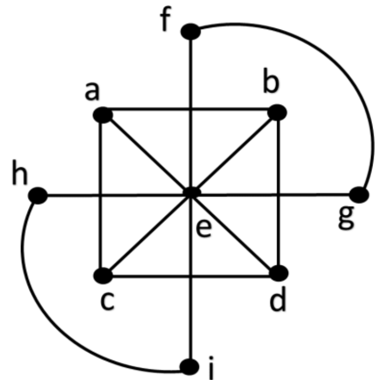


Fig: Connected graph

Complete graph

A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.

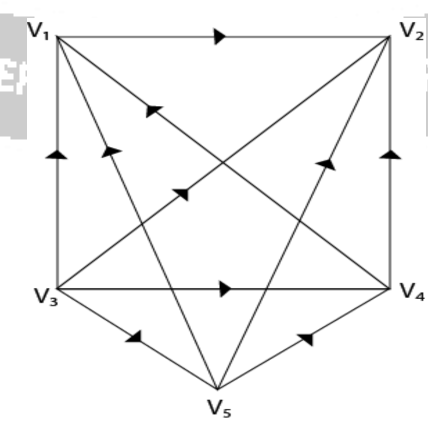


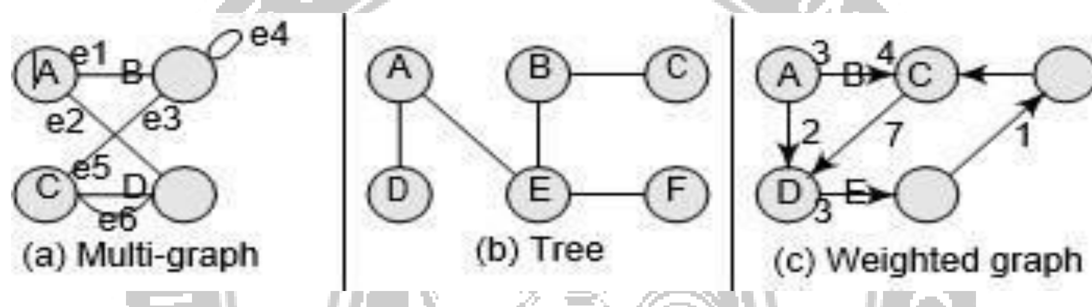
Fig: K_5

Clique

In a undirected graph G=(V, E), clique is a subset of the vertex, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled graph or weighted graph

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.



Multiple edges

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop

An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph

A graph with multiple edges and/or loops is called a multi-graph.

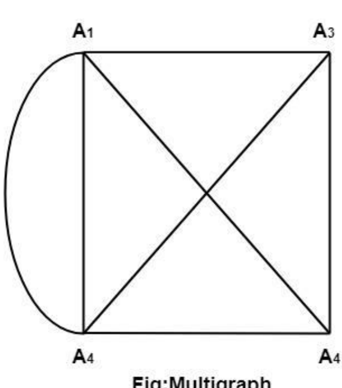


Fig:Multigraph

OBSERVE OPTIMIZE OUTSPREAD
PALKULAM, KANYAKUMARI

Size of a graph

The size of a graph is the total number of edges in it.

DIRECTED GRAPHS

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

Terminology of a Directed graph

Out-degree of a node - The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.

In-degree of a node - The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

Degree of a node - The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).

Isolated vertex - A vertex with degree zero. Such a vertex is not an end-point of any edge.

Pendant vertex (also known as leaf vertex) - A vertex with degree one.

Cut vertex - A vertex which when deleted would disconnect the remaining graph.

Source - A node u is known as a source if it has a positive out-degree but a zero in-degree.

Sink - A node u is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability -A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Strongly connected directed graph

A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally connected graph

A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

Weakly connected digraph

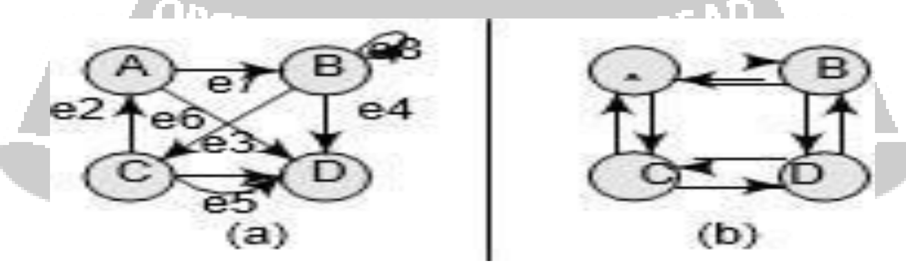
A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Simple directed graph

A directed graph G is said to be a simple directed graph if and only if it has no parallel edges



a) Directed acyclic graph and b) Strongly connected directed graph

REPRESENTATION OF GRAPH

There are three common ways of storing graphs in the computer's memory. They are:

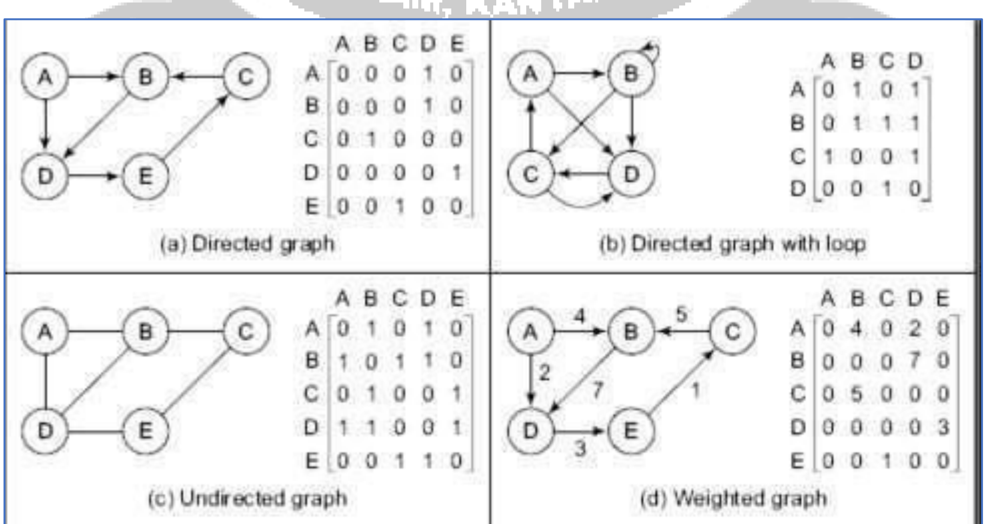
- Sequential representation by using an adjacency matrix.
- Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.
- Adjacency multi-list which is an extension of linked representation

Adjacency matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n ¥ n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero

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

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. therefore, a change in the order of nodes will result in a different adjacency matrix.



(a) Directed graph (b) Directed graph with loop (c) Undirected graph (d) Weighted graph

From the above examples, we can draw the following conclusions:

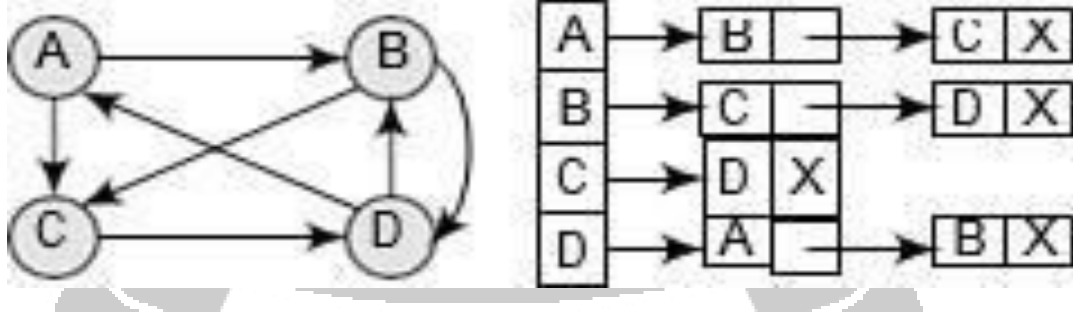
- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.

- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

ADJACENCY LIST REPRESENTATION

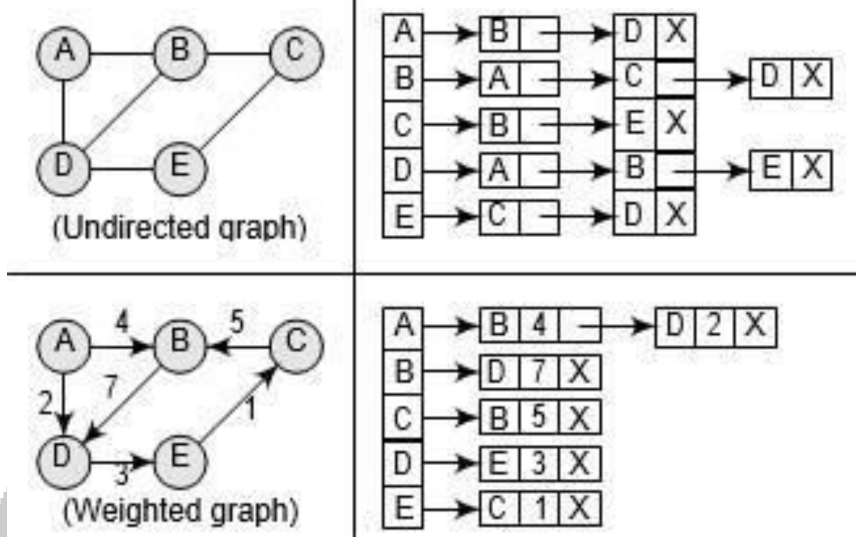
An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.



Graph G and its Adjacency List

Adjacency list for an undirected graph and a weighted graph



ADJACENCY MULTI-LIST REPRESENTATION

A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node i points to the adjacency list for node i.

In a multi-list representation, the information about an edge (vi , vj) of an undirected graph can be stored using the following attributes:

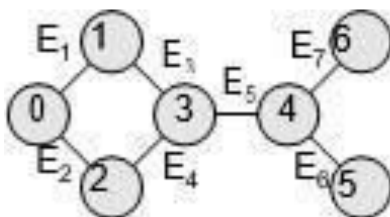
M: A single bit field to indicate whether the edge has been examined or not.

Vj : A vertex in the graph that is connected to vertex vi by an edge.

Vi : A vertex in the graph that is connected to vertex vi by an edge.

Link i for vj : A link that points to another node that has an edge incident on v .

Link j for vi : A link that points to another node that has an edge incident on vi .



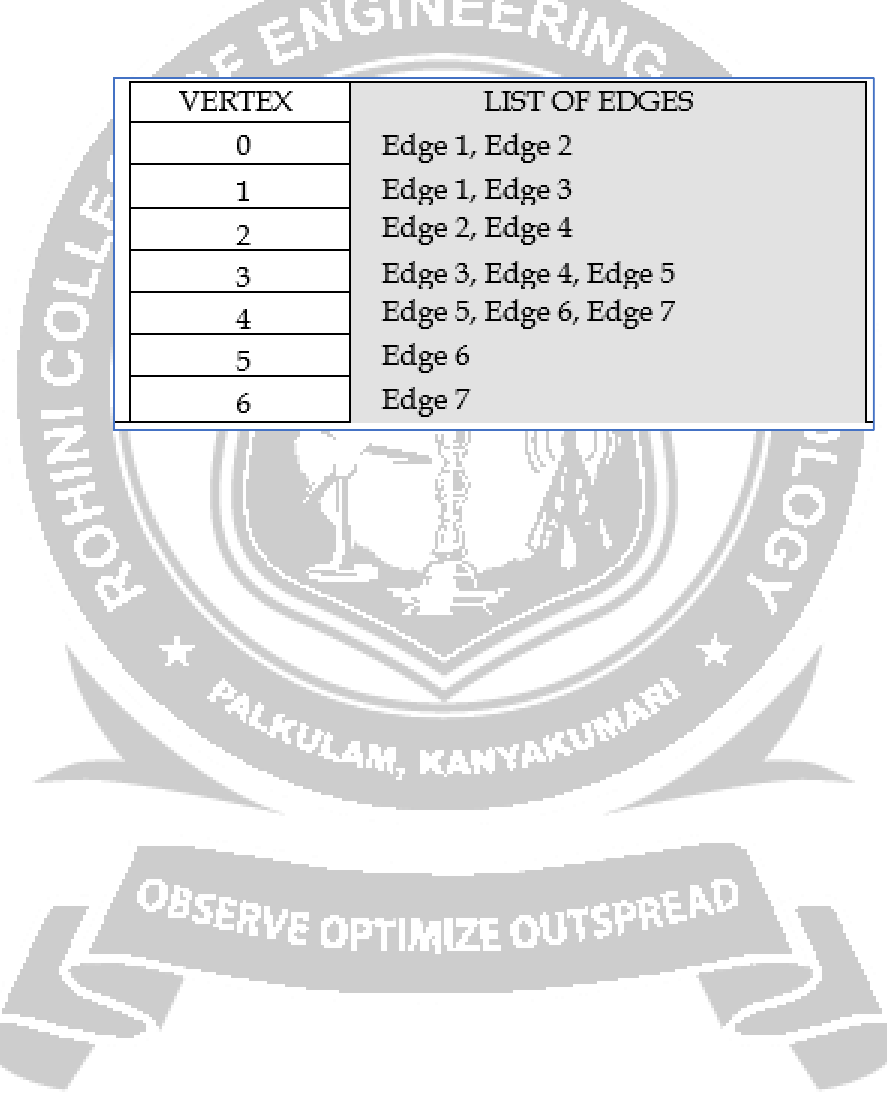
Undirected Graph

The adjacency multi-list for the graph can be given as:

Edge 1	0	1	Edge 2	Edge 3
Edge 2	0	2	NULL	Edge 4
Edge 3	1	3	NULL	Edge 4
Edge 4	2	3	NULL	Edge 5
Edge 5	3	4	NULL	Edge 6
Edge 6	4	5	Edge 7	NULL
Edge 7	4	6	NULL	NULL

Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

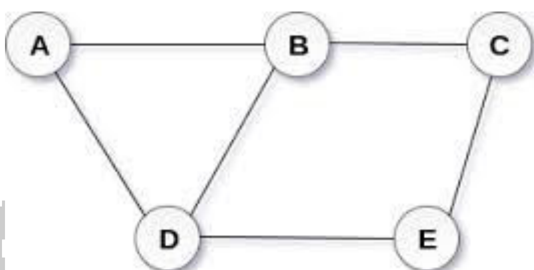
VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7



GRAPHS

DEFINITION

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

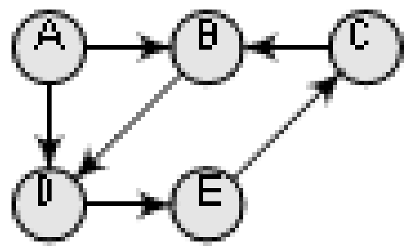


In the above graph, V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C,E)}.

There are five vertices or nodes and six edges in the graph.

DIRECTED AND UNDIRECTED GRAPHS

- In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
- In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).



Directed Graphs

GRAPH TERMINOLOGY

Adjacent nodes or neighbours

For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end- points and are said to be the adjacent nodes or neighbours.

Degree of a node

Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Closed path

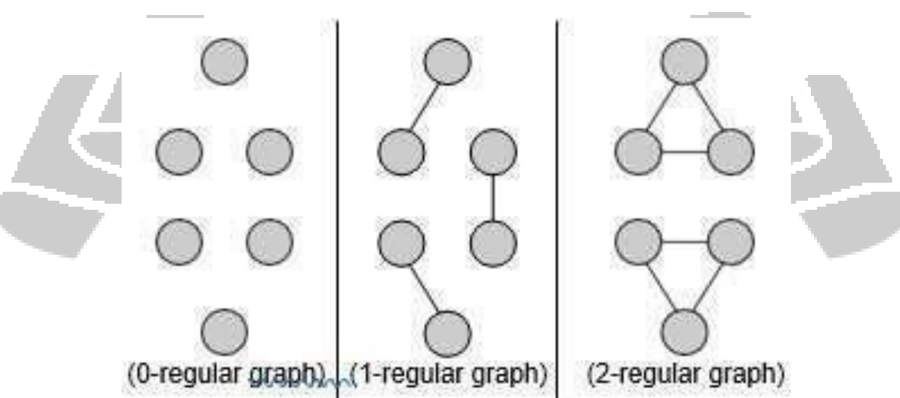
A path P is known as a closed path if the edge has the same end-points. That is, if vo=vn.

Simple path

A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0= vn then the path is called a closed simple path.

Cycle

A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).



Regular Graph

Types of Graphs

Connected graph

A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.

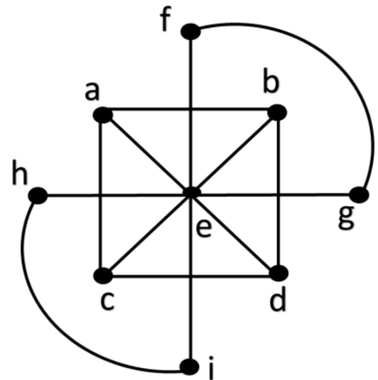


Fig: Connected graph

Complete graph

A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.

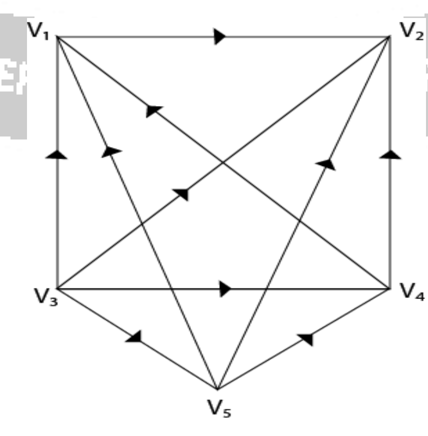


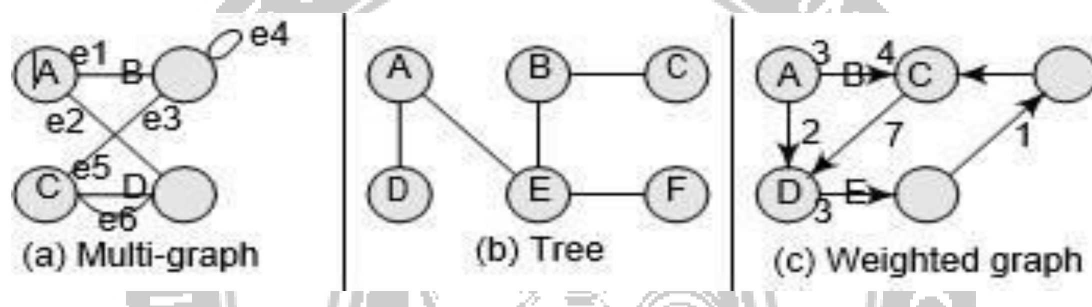
Fig: K_5

Clique

In a undirected graph G=(V, E), clique is a subset of the vertex, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled graph or weighted graph

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.



Multiple edges

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop

An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph

A graph with multiple edges and/or loops is called a multi-graph.

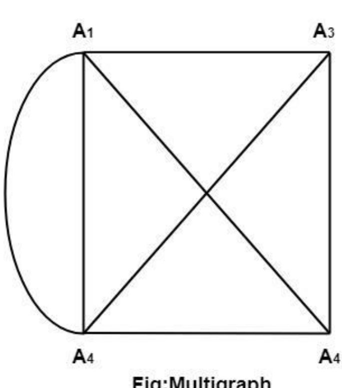


Fig:Multigraph

OBSERVE OPTIMIZE OUTSPREAD
PALKULAM, KANYAKUMARI

Size of a graph

The size of a graph is the total number of edges in it.

DIRECTED GRAPHS

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

Terminology of a Directed graph

Out-degree of a node - The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.

In-degree of a node - The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

Degree of a node - The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).

Isolated vertex - A vertex with degree zero. Such a vertex is not an end-point of any edge.

Pendant vertex (also known as leaf vertex) - A vertex with degree one.

Cut vertex - A vertex which when deleted would disconnect the remaining graph.

Source - A node u is known as a source if it has a positive out-degree but a zero in-degree.

Sink - A node u is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability -A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

Strongly connected directed graph

A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally connected graph

A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

Weakly connected digraph

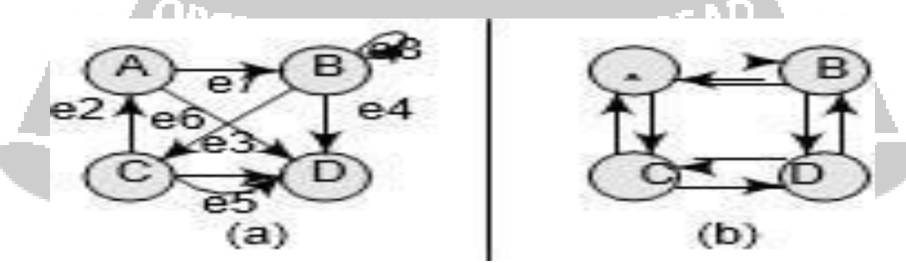
A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Simple directed graph

A directed graph G is said to be a simple directed graph if and only if it has no parallel edges



a) Directed acyclic graph and b) Strongly connected directed graph

REPRESENTATION OF GRAPH

There are three common ways of storing graphs in the computer's memory. They are:

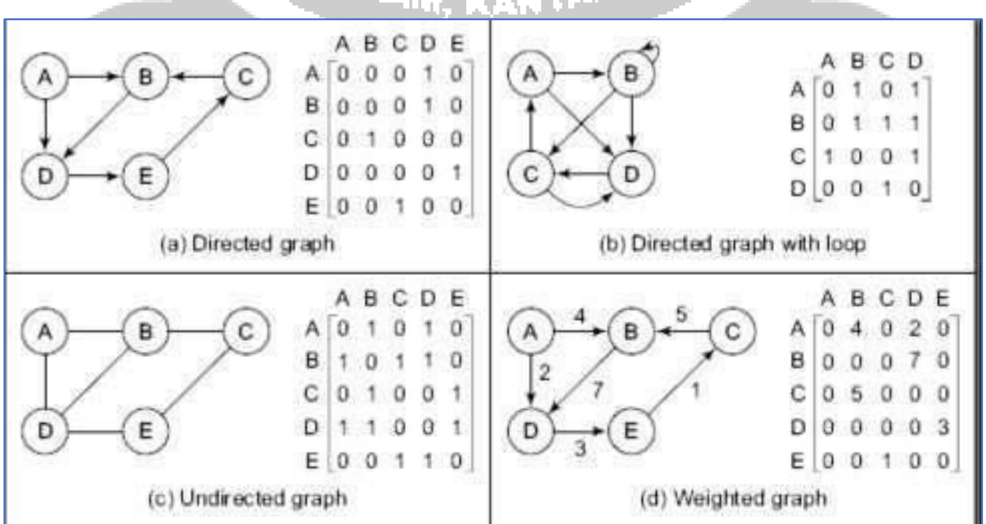
- Sequential representation by using an adjacency matrix.
- Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.
- Adjacency multi-list which is an extension of linked representation

Adjacency matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n ¥ n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero

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

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. therefore, a change in the order of nodes will result in a different adjacency matrix.



From the above examples, we can draw the following conclusions:

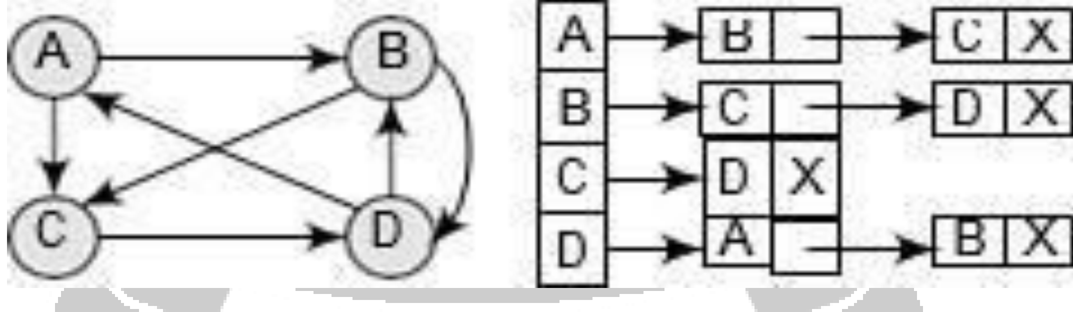
- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.

- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

ADJACENCY LIST REPRESENTATION

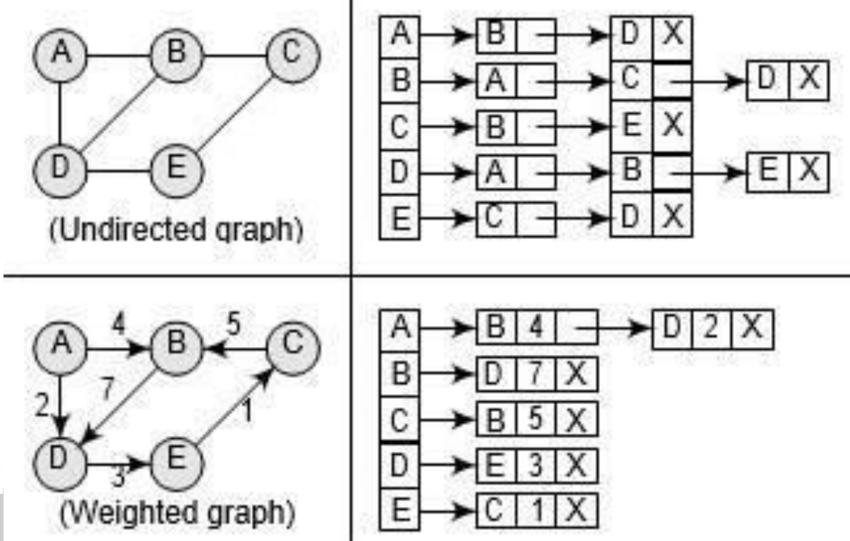
An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.



Graph G and its Adjacency List

Adjacency list for an undirected graph and a weighted graph



ADJACENCY MULTI-LIST REPRESENTATION

A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node i points to the adjacency list for node i.

In a multi-list representation, the information about an edge (vi , vj) of an undirected graph can be stored using the following attributes:

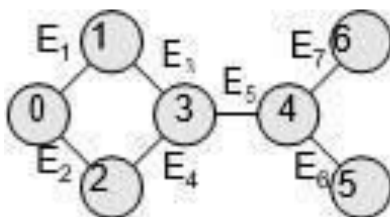
M: A single bit field to indicate whether the edge has been examined or not.

Vj : A vertex in the graph that is connected to vertex vi by an edge.

Vi : A vertex in the graph that is connected to vertex vi by an edge.

Link i for vj : A link that points to another node that has an edge incident on v .

Link j for vi : A link that points to another node that has an edge incident on vi .



Undirected Graph

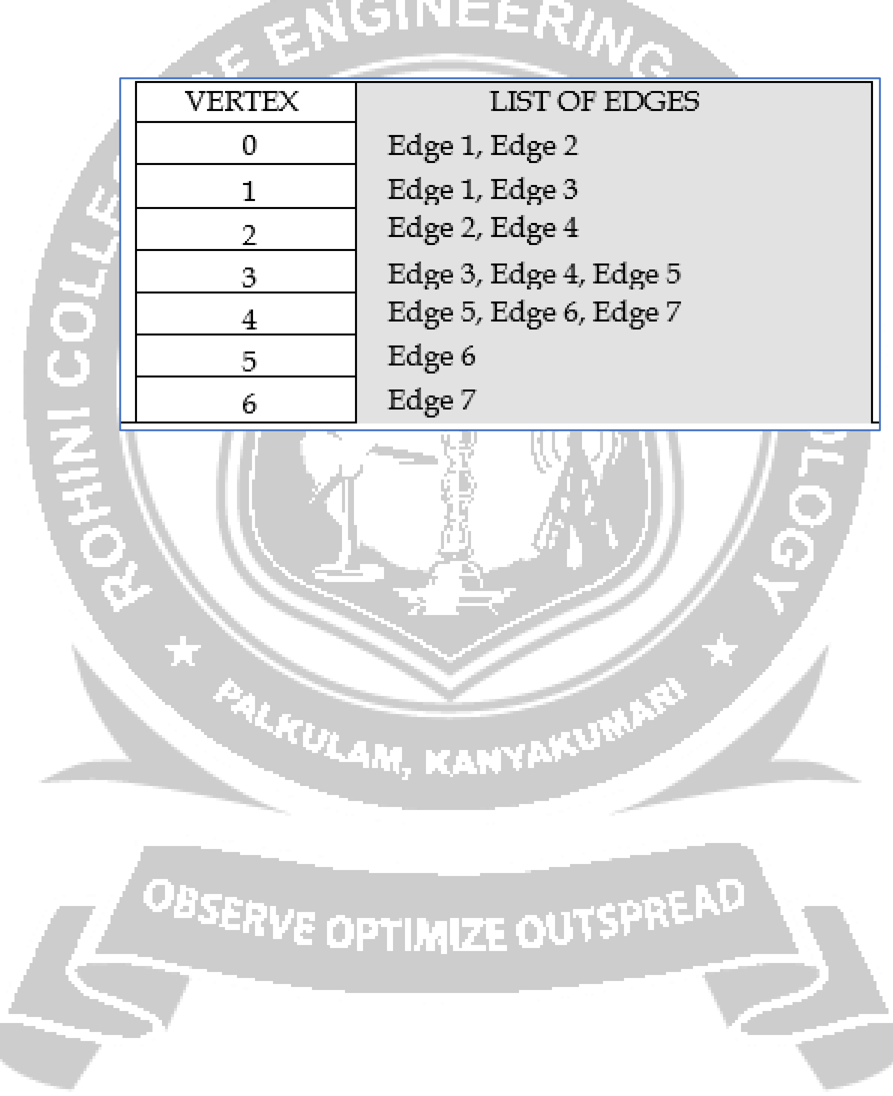
The adjacency multi-list for the graph can be given as:

Edge 1	0	1	Edge 2	Edge 3
Edge 2	0	2	NULL	Edge 4
Edge 3	1	3	NULL	Edge 4
Edge 4	2	3	NULL	Edge 5
Edge 5	3	4	NULL	Edge 6
Edge 6	4	5	Edge 7	NULL
Edge 7	4	6	NULL	NULL

Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown

below:

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7



GRAPH TRAVERSAL ALGORITHMS

There are two standard methods of graph traversal, they are

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state.

BREADTH-FIRST SEARCH ALGORITHM

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Algorithm to breadth-first Search

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

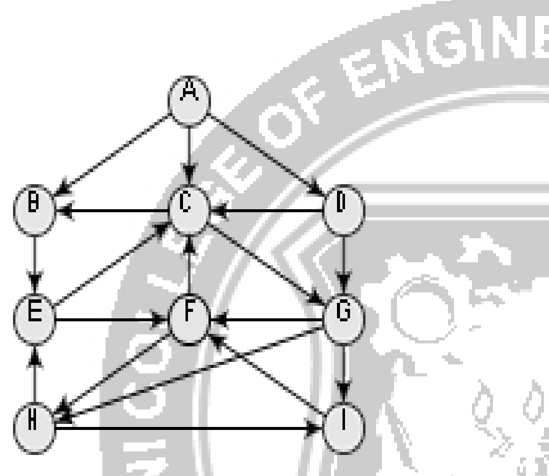
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

Example

Consider the graph G given. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.



Adjacency List
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:

QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed,

ORIG is used to keep track of the origin of each edge.

Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE =	A	B	C	D
REAR = 3	ORIG	\0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE =	A	B	C	D	E
REAR = 4	ORIG	\0	A	A	A	B

d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE =	A	B	C	D	E	G
REAR = 5	ORIG	\0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE =	A	B	C	D	E	G
REAR = 5	ORIG	\0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE =	A	B	C	D	E	G	F
REAR = 6	ORIG	\0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT=6	QUEUE =	A	B	C	D	E	G	F	H	I
REAR = 9	ORIG	\0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A -> C -> G -> I

Features of Breadth-First Search Algorithm

Space complexity

- In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).
- If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity

In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness

Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality

Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph

DEPTH-FIRST SEARCH ALGORITHM

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead- end, we backtrack to find another path P¢. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges.

STEPS

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

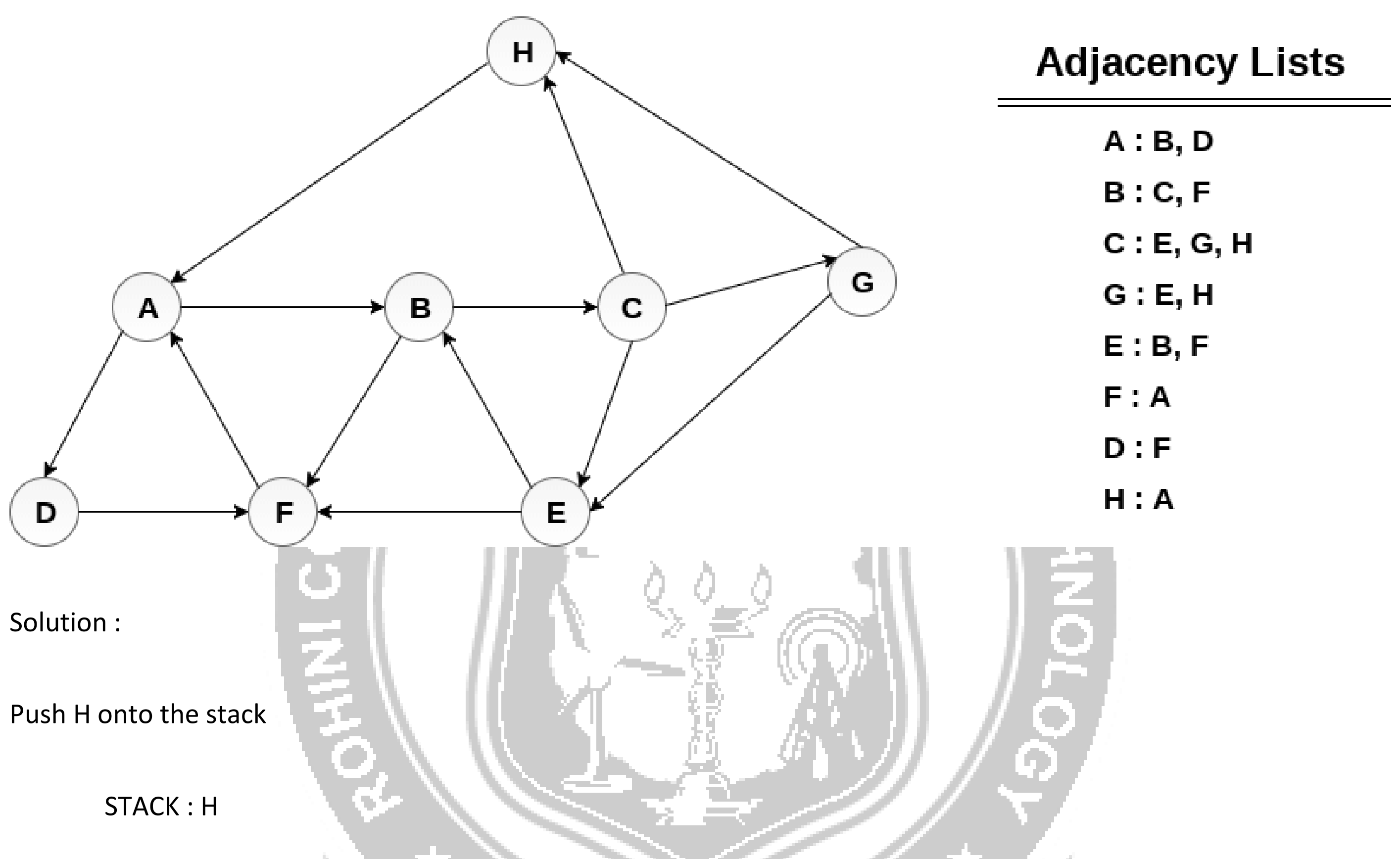
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

EXAMPLE

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search **(DFS) algorithm.**



Solution :

Push H onto the stack

STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

Print H

STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

Print A

Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

Print D

Stack : B, F

OBSERVE OPTIMIZE OUTSPREAD

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

Print F

Stack : B

Pop the top of the stack i.e. B and push all the neighbours

Print B

Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

Print E

Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.

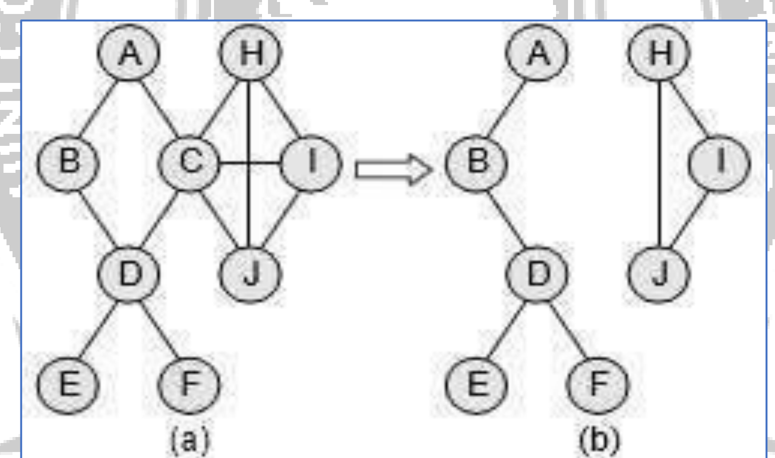
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

BI- CONNECTIVITY

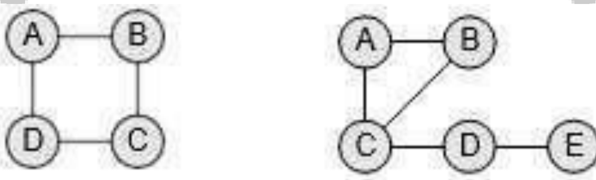
A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

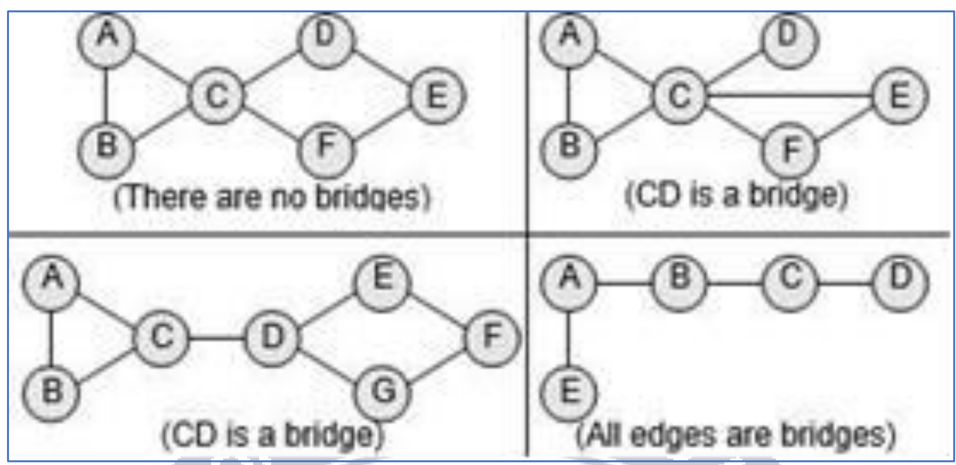
- A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.
- In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.



Note that the graph is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph



Biconnected Graph



Graph with Bridges

As for vertices, there is a related concept for edges. An edge in a graph is called a bridge if removing that edge results in a disconnected graph. Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge.

CUT VERTEX

Articulation point

The vertices whose removal would disconnect the graph are known as articulation points.

Steps to find Articulation point

1. Find DFS spanning tree

2. Number the vertex in the order in which they are visited. This number is referred as Num(v)

3. Compute the lowest numbered vertex for every vertex v in the DFS spanning tree which we call as Low(w)(i.e) reachable from v by taking 0 or more tree edges and then possible one back edge. By definition Low(v) is the

a. Minimum of Num(v)

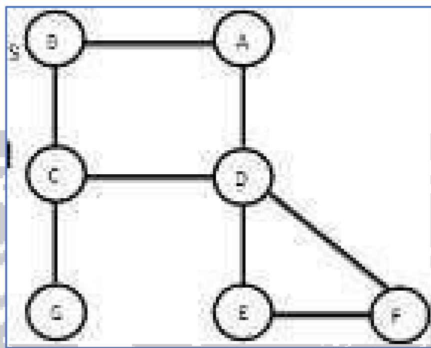
b. The lowest Num(w) among all back edges

c. The lowest Low(w) among all tree edges (i.e) min(Num(v),Num(w),Low(w))

Use post order traversal to calculate Low(v). The root is articulation if and only if it has more than 2 children. Any vertex V other than root is an articulation point if and only if V has some children such that Low(w)≥Num(v).

Example

Checking whether finding the articulation point



Low(v)=min(Num(v),Num(w),Low(w))

Low(F)=min(Num(F),Num(D),Low(D)) =min(6,4) =4

Low(E)=min(Num(E),Num(F),Low(F)) =min(5,6,4) =4

Low(D)=min(Num(D),Num(E),Low(E),Num(A),Low(A)) =min(4,5,4,1) =1

Low(G)=min(Num(G)) =min(7) =7

Low(C)=min(Num(C),Num(D),Low(D),Num(G),Low(G)) =min(3,4,1,7,7) =1

Low(B)=min(Num(B),Num(C),Low(C)) =min(2,3,1) =1

Low(A)=min(Num(A),Num(B),Low(B)) =min(1,2,1) =1

At vertex F Low(W) ≥Num(V) 1 ≥ 6

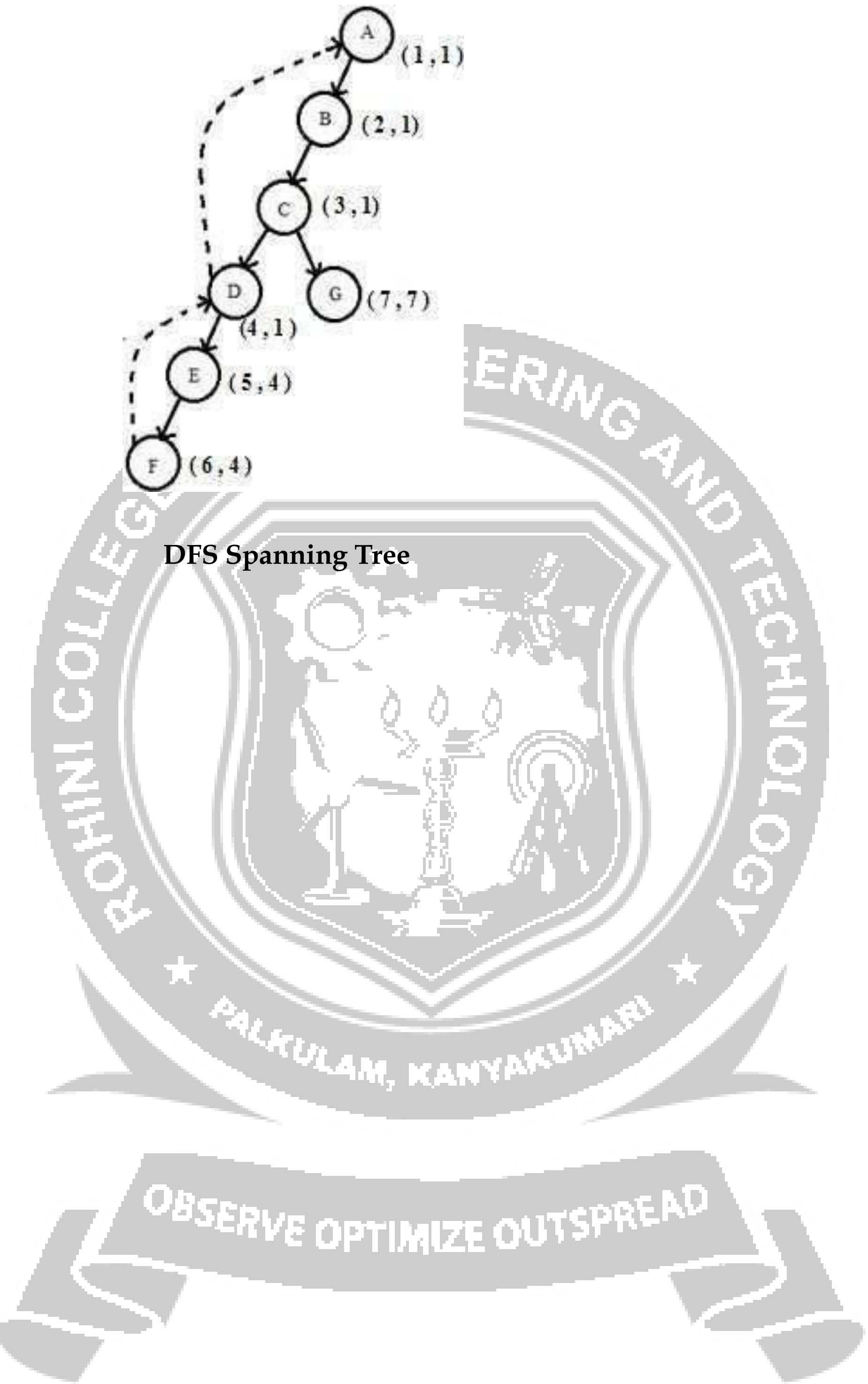
At vertex D Low(A) ≥ Num(D) 1 ≥ 4

Low(E) ≥ Num(D) 4 ≥ 4(articulation point)

At vertex E Low(F) ≥ Num(E) 4 ≥5

At vertex C Low(D) ≥ Num(C) 1 ≥3

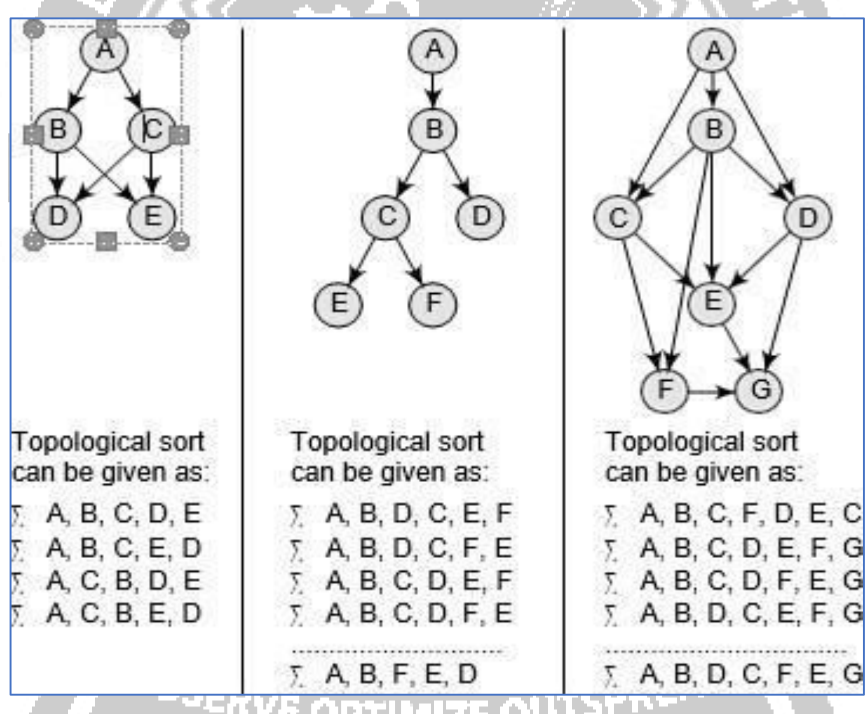
Low(G) ≥Num(C) 6 ≥ 3(articulation point)



DFS Spanning Tree

TOPOLOGICAL SORTING

- Topological sort of a directed acyclic graph (DAG) G is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.
- A topological sort of a DAG G is an ordering of the vertices of G such that if G contains an edge (u, v), then u appears before v in the ordering. Note that topological sort is possible only on directed acyclic graphs that do not have any cycles. For a DAG that contains cycles, no linear ordering of its vertices is possible.
- In simple words, a topological ordering of a DAG G is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.



Topological Sorting

Algorithm

The two main steps involved in the topological sort algorithm include:

- Selecting a node with zero in-degree
- Deleting N from the graph along with its edges

Step 1: Find the in-degree INDEG(N) of every node in the graph

Step 2: Enqueue all the nodes with a zero in-degree

Step 3: Repeat Steps 4 and 5 until the QUEUE is empty

Step 4: Remove the front node N of the QUEUE by setting FRONT = FRONT + 1

Step 5: Repeat for each neighbour M of node N:

a) Delete the edge from N to M by setting INDEG(M) = INDEG(M) – 1

b) IF INDEG(M) = , then Enqueue M, that is, add M to the rear of the queue

[END OF INNER LOOP]

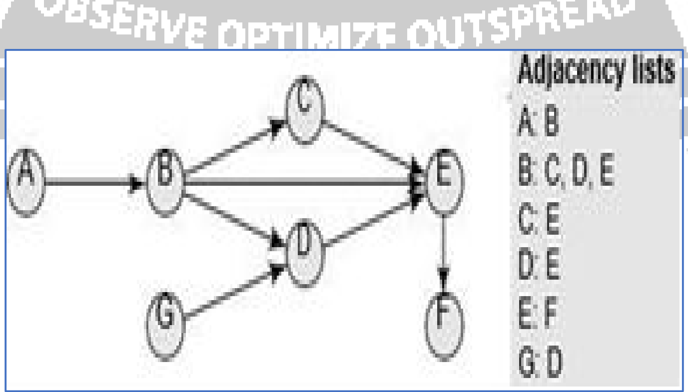
[END OF LOOP]

Step 6: Exit

We will use a QUEUE to hold the nodes with zero in-degree. The order in which the nodes will be deleted from the graph will depend on the sequence in which the nodes are inserted in the QUEUE. Then, we will use a variable INDEG, where INDEG(N) will represent the in-degree of node N.

Example

Consider a directed acyclic graph G given below. We use the algorithm given above to find a topological sort T of G. The steps are given as below



Step 1: Find the in-degree INDEG(N) of every node in the graph

INDEG(A) = 0 INDEG(B) = 1 INDEG(C) = 1 INDEG(D) = 2

INDEG(E) = 3 INDEG(F) = 1 INDEG(G) = 0

Step 2: Enqueue all the nodes with a zero in-degree

FRONT = 1 REAR = 2 QUEUE = A, G

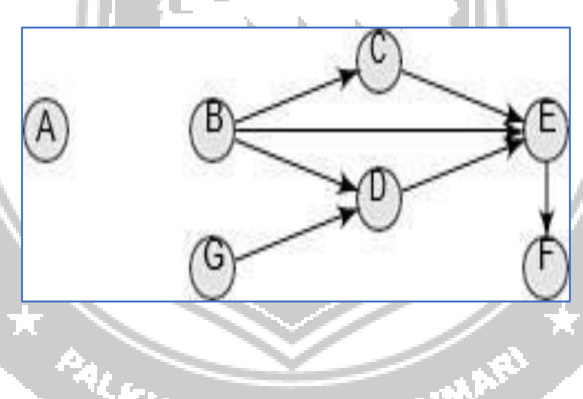
Step 3: Remove the front element A from the queue by setting FRONT = FRONT + 1, so

FRONT = 2 REAR = 2 QUEUE = A, G

Step 4: Set INDEG(B) = INDEG(B) – 1, since B is the neighbour of A. Note that INDEG(B) is 0, so add it on the queue. The queue now becomes

FRONT = 2 REAR = 3 QUEUE = A, G, B

Delete the edge from A to B. The graph now becomes as shown in the figure below



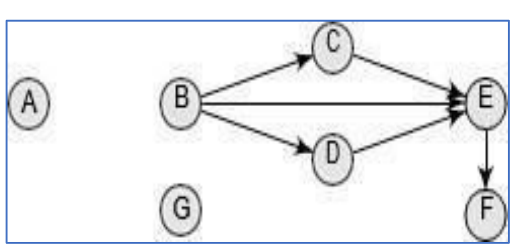
Step 5: Remove the front element B from the queue by setting FRONT = FRONT + 1, so

FRONT = 2 REAR = 3 QUEUE = A, G, B

Step 6: Set INDEG(D) = INDEG(D) – 1, since D is the neighbour of G. Now, INDEG(C) = 1

INDEG(D) = 1 INDEG(E) = 3 INDEG(F) = 1

Delete the edge from G to D. The graph now becomes as shown in the figure below



ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

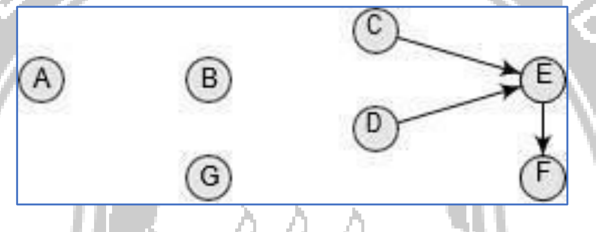
Step 7: Remove the front element B from the queue by setting FRONT = FRONT + 1, so

FRONT = 4 REAR = 3 QUEUE = A, G, B

Step 8: Set INDEG(C) = INDEG(C) – 1, INDEG(D) = INDEG(D) – 1, INDEG(E) = INDEG(E) – 1, since C, D, and E are the neighbours of B. Now, INDEG(C) = 0, INDEG(D) = 1 and INDEG(E) = 2

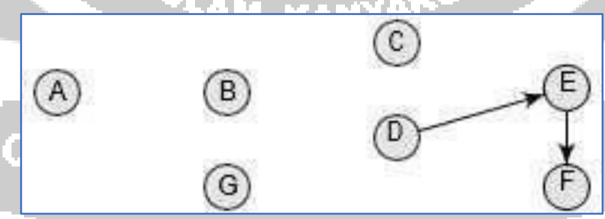
Step 9: Since the in-degree of node c and D is zero, add C and D at the rear of the queue. The queue can be given as below:

FRONT = 4 REAR = 5 QUEUE = A, G, B, C, D The graph now becomes as shown in the figure below



Step 10: Remove the front element C from the queue by setting FRONT = FRONT + 1, so FRONT = 5 REAR = 5 QUEUE = A, G, B, C, D

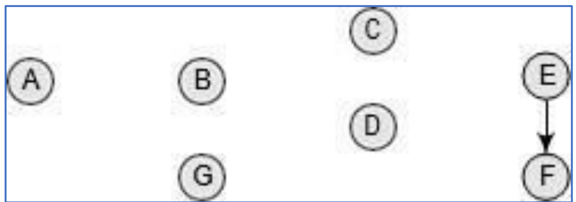
Step 11: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of C. Now, INDEG(E) = 1 The graph now becomes as shown in the figure below



Step 12: Remove the front element D from the queue by setting FRONT = FRONT + 1, so FRONT = 6 REAR = 5 QUEUE = A, B, G, C, D

Step 13: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of D. Now, INDEG(E) = 0, so add E to the queue. The queue now becomes. FRONT = 6 REAR = 6 QUEUE = A, G, B, C, D, E

Step 14: Delete the edge between D an E. The graph now becomes as shown in the figure below



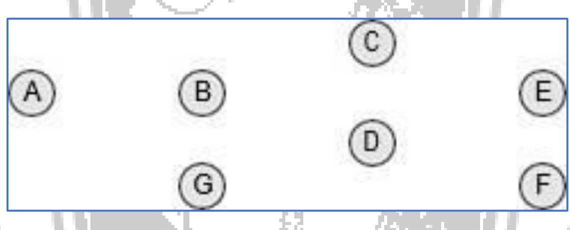
Step 15: Remove the front element D from the queue by setting FRONT = FRONT + 1, so FRONT = 7 REAR = 6

QUEUE = A, G, B, C, D, E

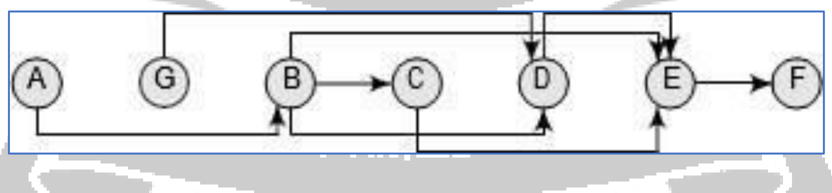
Step 16: Set INDEG(F) = INDEG(F) – 1, since F is the neighbour of E. Now INDEG(F) = 0, so add F to the queue. The queue now becomes,

FRONT = 7 REAR = 7 QUEUE = A, G, B, C, D, E, F

Step 17: Delete the edge between E and F. The graph now becomes as shown in the figure below



There are no more edges in the graph and all the nodes have been added to the queue, so the topological sort T of G can be given as: A, G, B, C, D, E, F. When we arrange these nodes in a sequence, we find that if there is an edge from u to v, then u appears before v.



Topological Sort of G

SHORTEST PATH ALGORITHMS

An algorithm to find the shortest distance path between the source and destination vertices is called the shortest path algorithm.

Types of shortest path problem

i. Single source shortest path

Given an input graph G = (V,E) and a distinguished vertex S, find the shortest path from S to every other vertex in G. Example: Dijkstra�even‟s algorithm (weighted graph and unweighted graph).

ii. All pairs shortest path problem

Given an input graph G = (V,E). Find the shortest path from each vertex to all vertices in a graph.

Dijkstra‟s algorithm

Weighted Graph

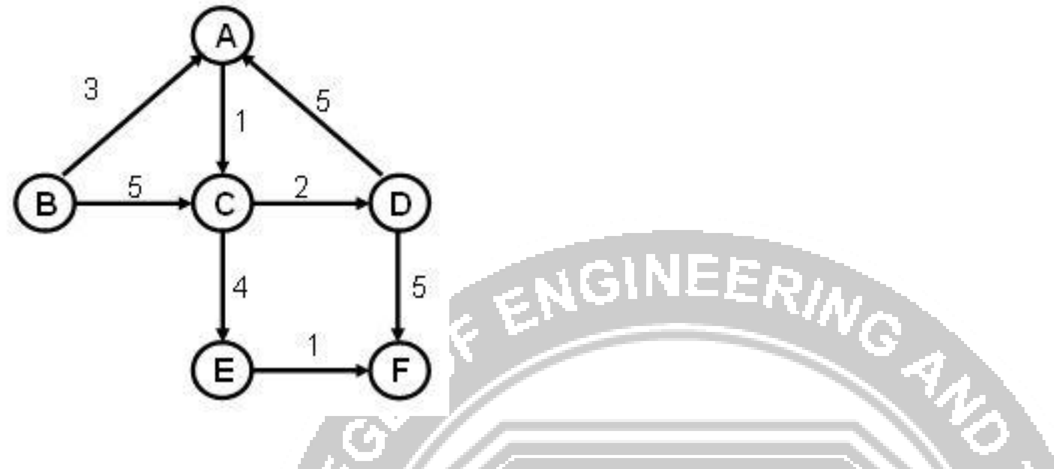
The general method to solve the single source shortest path problem is known as Dijkstra‟s algorithm. It applied to weighted graph.

Procedure

- It uses greedy technique.
- It proceeds in stages.
- It selects a vertex v, which has the smallest dv among all the unknown vertices and declares the shortest path from s to v is known.
- The remainder consists of updating the value of dw.
- We should set dw = dv + Cv, w, if the new value for dw would an improvement.

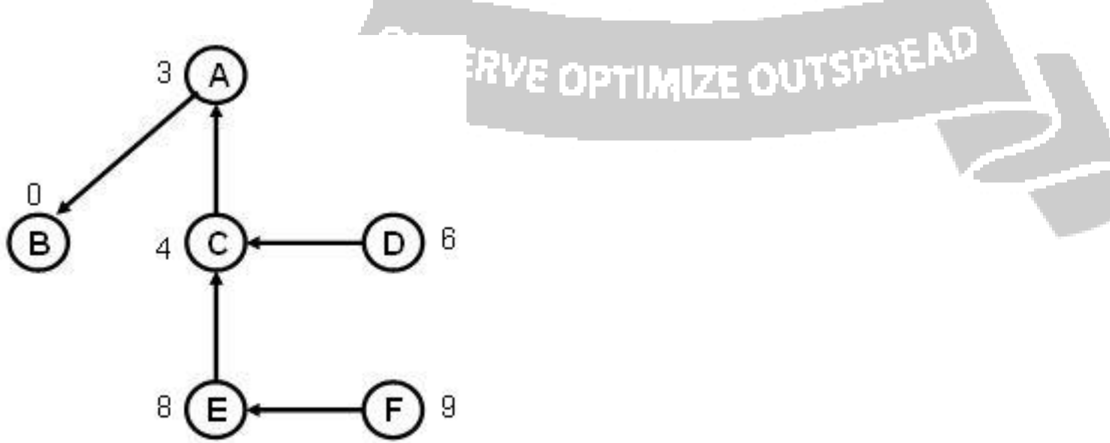
Example: Find the shortest path for the following graph.

Tracing Dijkstra's algorithm starting at vertex B:



Pass:	initially	1	2	3	4	5	6	Shortest distance	Predecessor
Active vertex:		B	A	C	D	E	F		
A	∞	3						3	B
B	0							0	-
C	∞	5	4					4	A
D	∞	∞	∞	6				6	C
E	∞	∞	∞	8	8			8	C
F	∞	∞	∞	∞	11	9		9	E

The resulting vertex-weighted graph is:



Algorithm Analysis

Time complexity of this algorithm $O(|E| + |V|^2) = O(|V|^2)$

Table Initialization routine

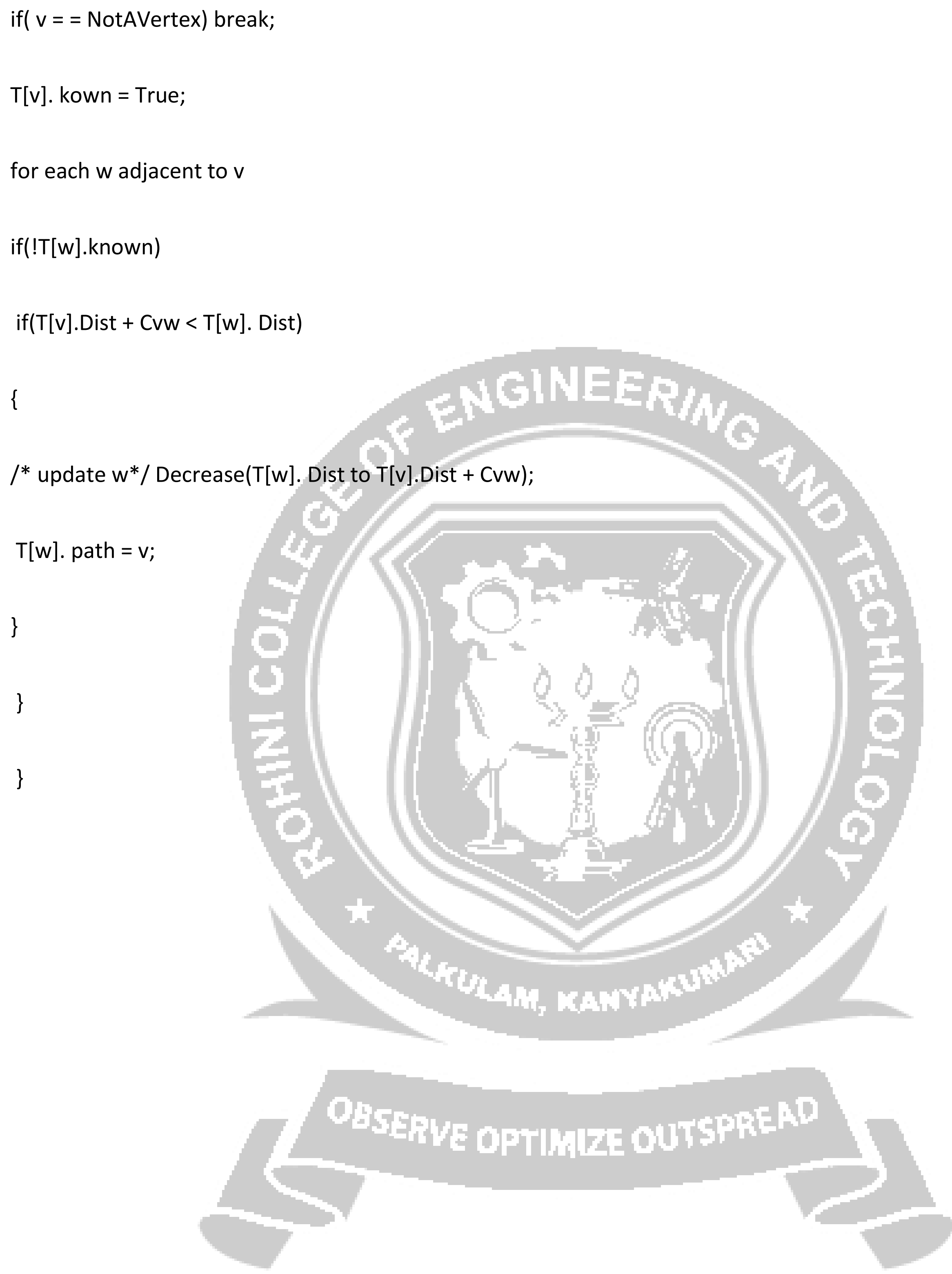
```
void InitTable(Vertex Start, Graph G, Table T)

{

int i;

ReadGraph(G,T);

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

{

T[i].known = False;

T[i]. Dist = Infinity;

T[i]. Path = NotAVertex;

}

T[Start]. Dist = 0;

}
```

Pseudocode for Dijkstra's algorithm

```
void Dijkstra(Table T)

{

Vertex v, w;

for( ; ;)

{

v = smallest unknown distance vertex;
```

OBSERVE OPTIMIZE OUTSPREAD

```
if( v = = NotAVertex) break;

T[v]. kown = True;

for each w adjacent to v

if(!T[w].known)

 if(T[v].Dist + Cvw < T[w]. Dist)

{

/* update w*/ Decrease(T[w]. Dist to T[v].Dist + Cvw);

 T[w]. path = v;

}

 }

 }
```



MINIMUM SPANNING TREES

- A Spanning tree of an undirected graph, G is a tree formed from graph edges that connects all vertices of G.
- A Minimum Spanning tree of an undirected graph, G is a tree formed from graph edges that connects all vertices of G at lowest cost.
- A minimum spanning tree exists if and only if G is connected. The number of edges in the minimum spanning tree is |V| -1.
- The minimum spanning tree is a tree because it is acyclic, it is spanning because it covers every vertex, and it is minimum because it covers with minimum cost.
- The minimum spanning tree can be created using two algorithms, that is prim's algorithm and kruskal's algorithm.

PRIM'S ALGORITHM

In this method, minimum spanning tree is constructed in successive stages. In each stage, one node is picked as a root and an edge is added and thus an associated vertex is added to the tree.

The Strategy

- One node is picked as a root node (u) from the given connected graph.
- At each stage choose a new vertex v from u, by considering an edge (u,v) with minimum cost among all edges from u, where u is already in the tree ad v is not in the tree.

The prims algorithm table is constructed with three parameters.

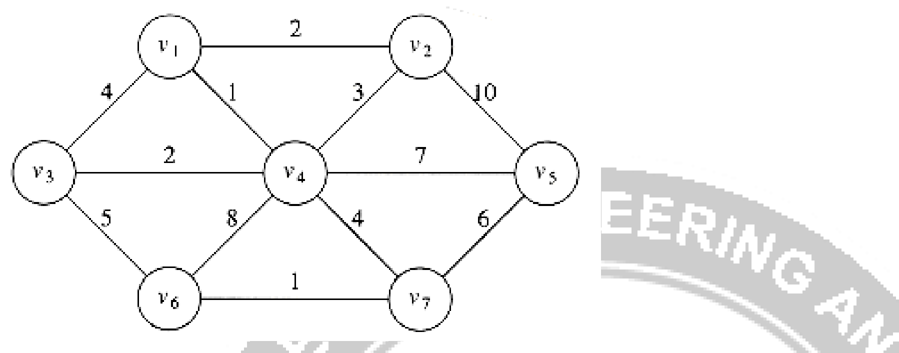
They are

- known – known vertex i.e., processed vertex is indicated by 1. Unknown vertex is indicated by zero.
- dv - Weight of the shortest edge connecting v to the known vertex.
- pv - It contains last vertex to cause a change in dv.

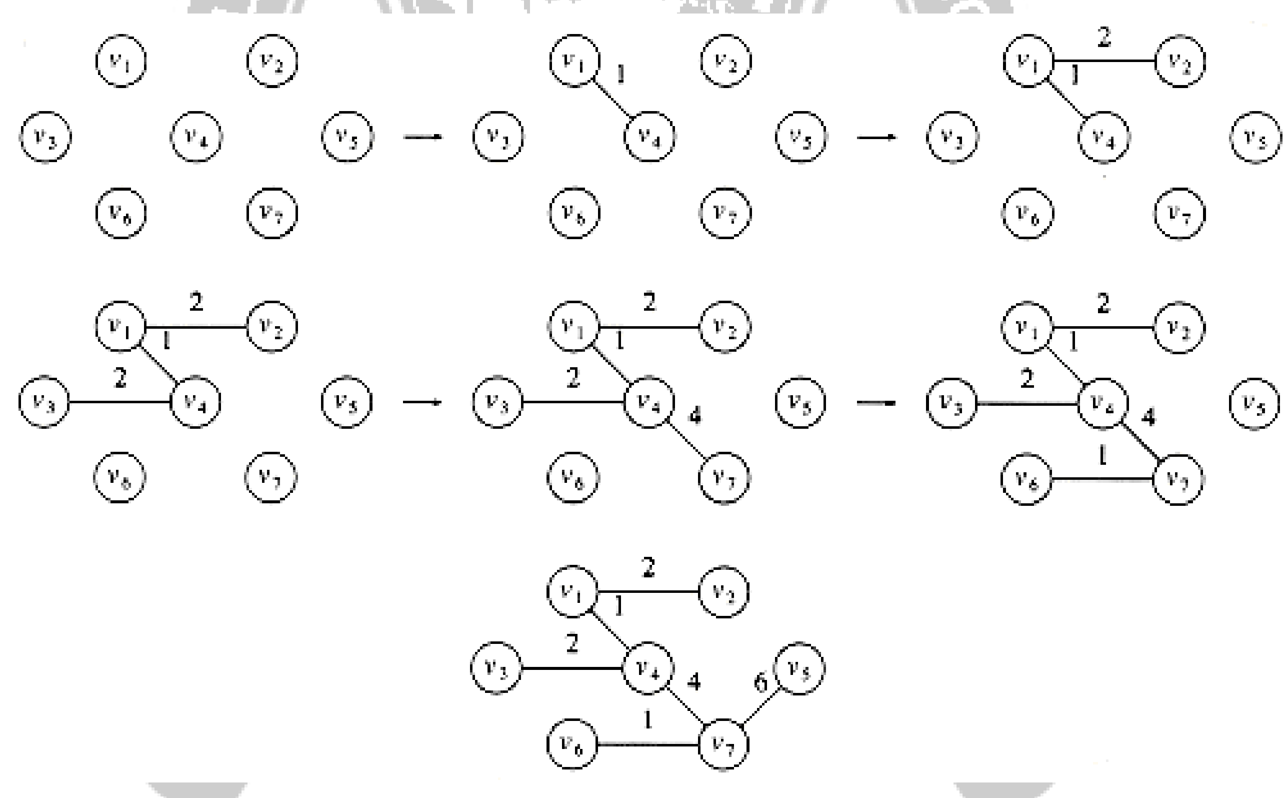
- After selecting the vertex v, the update rule is applied for each unknown w adjacent to v. The rule is

  dw = min (dw , Cw,v).

Example:



Prim's Algorithm after each stage



Steps

v1 is selected as initial node and construct initial configuration of the table.

v	Known	dv	pv
V1	0	0	0
V2	0	∞	0

V3	0	∞	0
V4	0	∞	0
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

ii. v1 is declared as known vertex. Then its adjacent vertices v2, v3, v4 are updated.

T[v2].dist = min(T[v2].dist, Cv1,v2) = min (∞ ,2) = 2

T[v3].dist = min(T[v3].dist, Cv1,v3) = min (∞ ,4) = 4

T[v4].dist = min(T[v4].dist, Cv1,v4) = min (∞ ,1) = 1

v	Known	dv	pv
V1	1	0	0
V2	0	2	V1
V3	0	4	V1
V4	0	1	V1
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

iii. Among all adjacent vertices V2, V3, V4. V1 -> V4 distance is small. So V4 is selected and declared as known vertex.

Its adjacent vertices distance are updated.

V1 is not examined because it is known vertex.

No change in V2 , because it has dv = 2 and the edge cost from V4 -> V2 = 3.

T[v3].dist = min(T[v3].dist, Cv4,v3) = min (4 ,2) = 2

T[v5].dist = min(T[v5].dist, Cv4,v5) = min (∞ ,7) = 7

T[v6].dist = min(T[v6].dist, Cv4,v6) = min (∞ ,8) = 8

T[v7].dist = min(T[v7].dist, Cv4,v7) = min (∞ ,4) = 4

v	Known	dv	pv
V1	1	0	0
V2	0	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4
V7	0	4	V4

iv. Among all either we can select v2, or v3 whose dv = 2, smallest among v5, v6 and v7.

v2 is declared as known vertex.

Its adjacent vertices are v1, v4 and v5. v1, v4 are known vertex, no change in their dv value.

T[v5].dist = min(T[v5].dist, Cv2,v5) = min (7 ,10) = 7

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

v. Among all vertices v3's dv value is lower so v3 is selected. v3�even's adjacent vertices are v1, v4 and v6. No changes in v1 and v4.

T[v6].dist = min(T[v6].dist, Cv3,v6) = min (8 ,5) = 5

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	5	V3
V7	0	4	V4

vi. Among v5, v6, v7, v7‟s dv value is lesser, so v7 is selected. Its adjacent vertices are v4, v4, and v6. No change in v4.

T[v5].dist = min(T[v5].dist, Cv7,v5) = min (7,6) = 6

T[v6].dist = min(T[v6].dist, Cv7,v6) = min (5,1) = 1

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4

vii. Among v5 and v6, v6 is declared as known vertex. v6‟s adjacent vertices are v3, v4, and v7, no change in dv value, all are known vertices.

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	1	1	V7
V7	1	4	V4

viii. Finally v5 is declared as known vertex. Its adjacent vertices are v2, v4, and v7, no change in dv value, all are known vertices.

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	1	6	V7
V6	1	1	V7
V7	1	4	V4

The minimum cost of spanning tree is 16.

Algorithm Analysis

The running time is O(|V|2) in case of adjacency list and O(|E| log |V|) in case of binary heap.

Kruskal's Algorithm

A second greedy strategy is repeatedly to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

Steps:

- Initially, there are |V| single-node trees. Adding an edge merges two trees into one.
- When the algorithm terminates, there is only one tree, and this is the minimum spanning tree.
- The algorithm terminates when enough edges are accepted.

The strategy

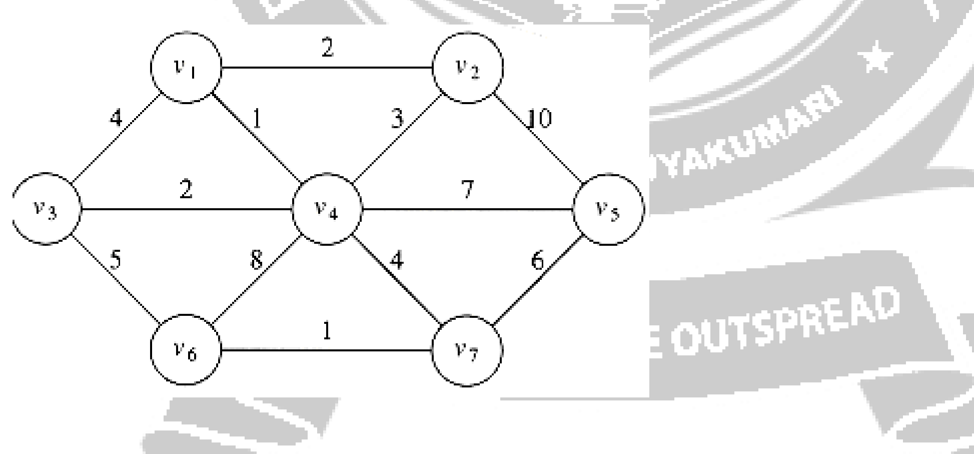
i. The edges are built into a minheap structure and each vertex is considered as a sigle node tree.

ii. The deletemin operation is used to find the minimum cost edge (u,v).

iii. The vertices u and v are searched in the spanning tree set S and if the returned sets are not same then (u,v) is added to the set s with the constraint that adding (u,v) will not create a cycle in spanning tree set S.

iv. Repeat step (ii) and (iii) until a spanning tree is constructed with |V| - 1 edges.

Example



i. Initially all the vertices are single node trees.

ii. Select the smallest edge v1 to v4, both the nodes are different sets, it does not form cycle.

iii. Select the next smallest edge v6 to v7. These two vertices are different sets; it does not form a cycle, so it is included in the MST.

iv. Select the next smallest edge v1 to v2. These two vertices are different sets; it does not form a cycle, so it is included in the MST.

v. Select the next smallest edge v3 to v4. These two vertices are different sets; it does not form a cycle, so it is included in the MST.

vi. Select the next smallest edge v2 to v4 both v2 and v4 are same set, it forms cycle so v2 – v4 edge is rejected.

vii. Select the next smallest edge v1 to v3, it forms cycle so v1 – v3 edge is rejected.

viii. Select the next smallest edge v4 to v7, it does not form a cycle so it is included in the tree.

ix. Select the next smallest edge v3 to v6, it forms a cycle so v3 – v6 edge is rejected.

x. Select the next smallest edge v5 to v7, it does not form a cycle so it is included in the tree.

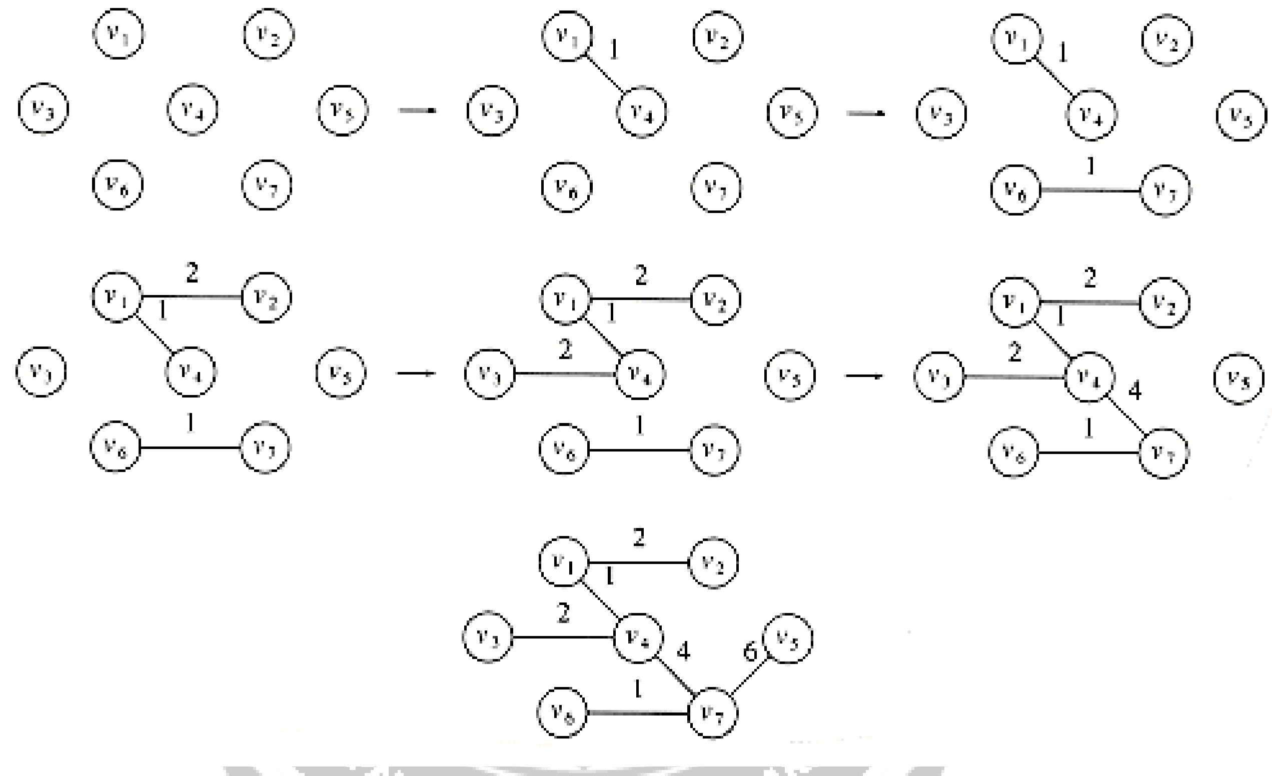
Edge	Weight	Action
(v1,v4)	1	Accepted
(v6,v7)	1	Accepted
(v1,v2)	2	Accepted
(v3,v4)	2	Accepted
(v2,v4)	3	Rejected
(v1,v3)	4	Rejected
(v4,v7)	4	Accepted
(v3,v6)	5	Rejected
(v5,v7)	6	Accepted

(v3,v6) 5 Rejected

(v5,v7) 6 Accepted

Figure: Action of Kruskal's algorithm on G

All the nodes are included. The cost of minimum spanning tree = 16 (2 + 1+ 2 + 4 + 1 + 6).



ALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

Routine for kruskals algorithm

```
void kruskal( graph G )

{

int EdgesAccepted;

DisjSet S;

PriorityQueue H;

vertex u, v;

SetType uset, vset;
```

```
Edge e;

Initialize( S ); // form a single node tree ReadGraphIntoHeapArray( G, H );

BuildHeap( H );

EdgesAccepted = 0;

while( EdgesAccepted < NumVertex-1 )

{

e = DeleteMin( H ); // Selection of minimum edge

uset = Find( u, S );

vset = Find( v, S );

if( uset != vset ) { /* accept the edge */ EdgesAccepted++; SetUnion( S, uset, vset );

}

}

}
```

The appropriate data structure is the union/find algorithm

The worst-case running time of this algorithm is O(|E| log |E|), which is dominated by the heap operations. Notice that since |E| = O(|V|2), this running time is actually O(|E| log |V|).

OBSERVE OPTIMIZE OUTSPREAD

Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Binary Search

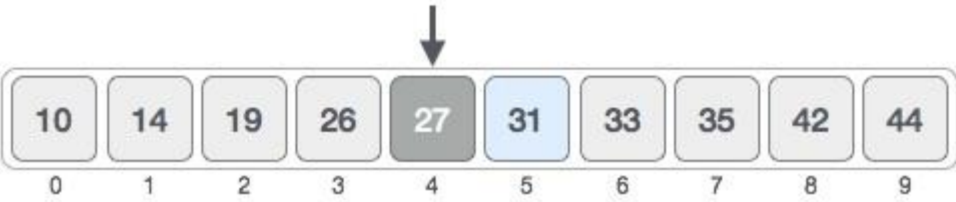
Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

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

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

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



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is

greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

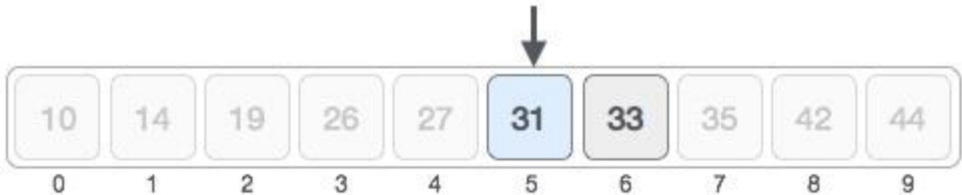
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

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

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

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



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

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

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

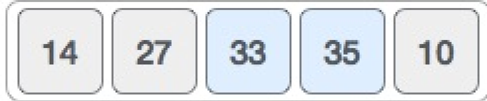
Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

SELECTION SORT

This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worstcase complexities are of $O(n^2)$, where **n** is the number of items.

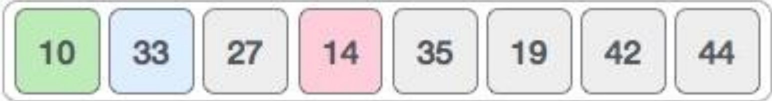
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

10 14 27 33 35 19 42 44

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –

10 14 27 33 35 19 42 44

10 14 27 33 35 19 42 44

10 14 19 33 35 27 42 44

10 14 19 33 35 27 42 44

10 14 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

Algorithm

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

INSERTION SORT

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worstcase complexity are of $O(n^2)$, where **n** is the number of items.

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. N

Algorithm

Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

SHELL SORT

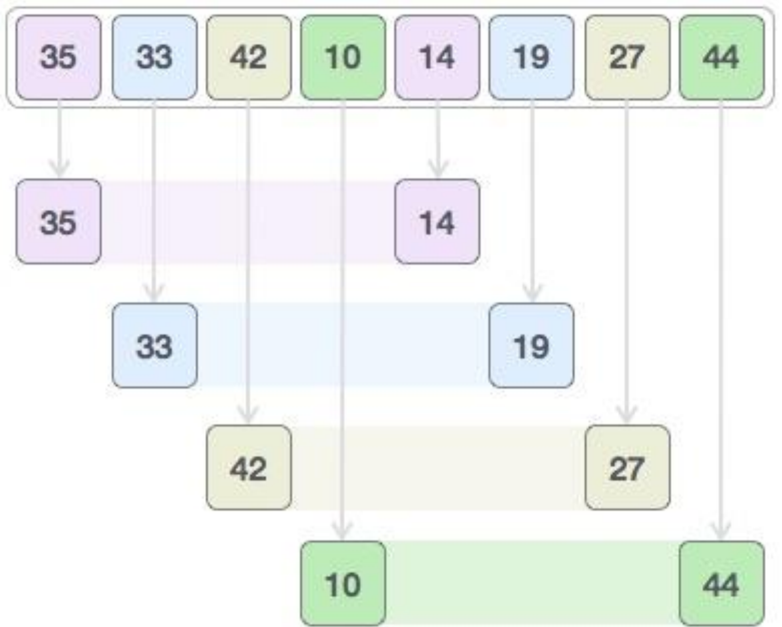
Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as –

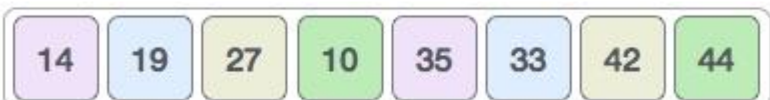
```
h = h * 3 + 1
where –
  h is interval with initial value 1
```

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worstcase space complexity is O(n).

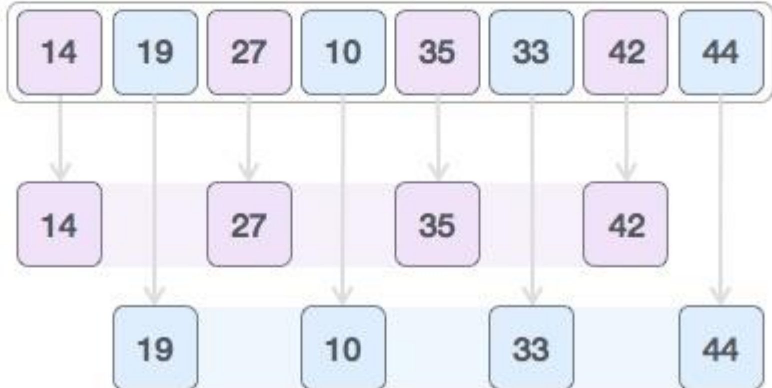
Let us consider the below elements. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



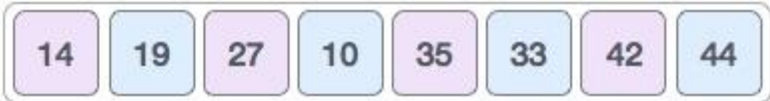
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction –

We see that it required only four swaps to sort the rest of the array.

Algorithm

Step 1 – Initialize the value of *h*
Step 2 – Divide the list into smaller sub-list of equal interval *h*
Step 3 – Sort these sub-lists using **insertion sort**
Step 3 – Repeat until complete list is sorted

MERGE SORT

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

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

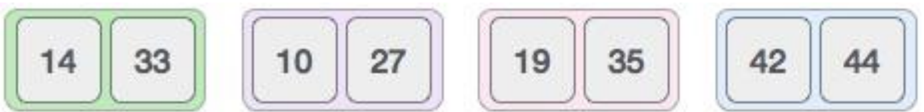
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

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

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

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

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



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

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

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 – if it is only one element in the list it is already sorted, return.
Step 2 – divide the list recursively into two halves until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order.

HASHING

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

The two types of hashing are separate chaining and open addressing

RE HASHING

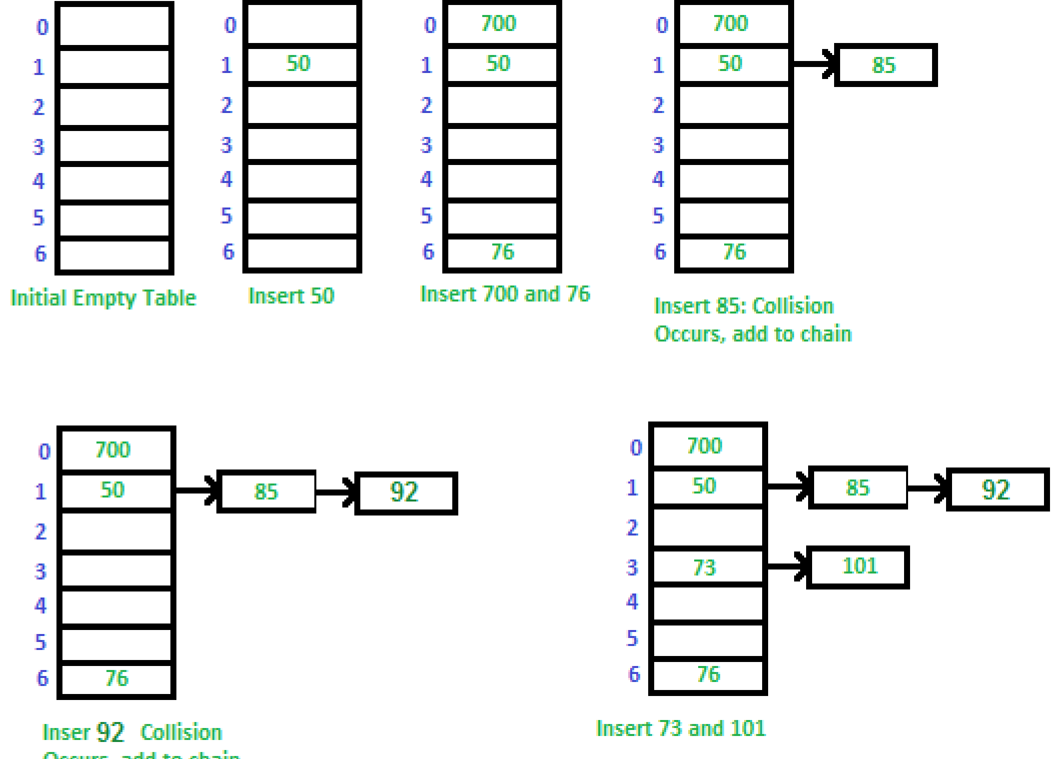
Rehashing is the process of recalculating the hash code of previously stored entries (key values pair) in order to shift them to larger size hash map when the threshold is reached or crossed.

SEPARATE CHAINING

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



OPEN ADDRESSING

It inserts the data into the hash table itself. The size of the hash table should be larger than the number of keys.

There are three types of open addressing. They are

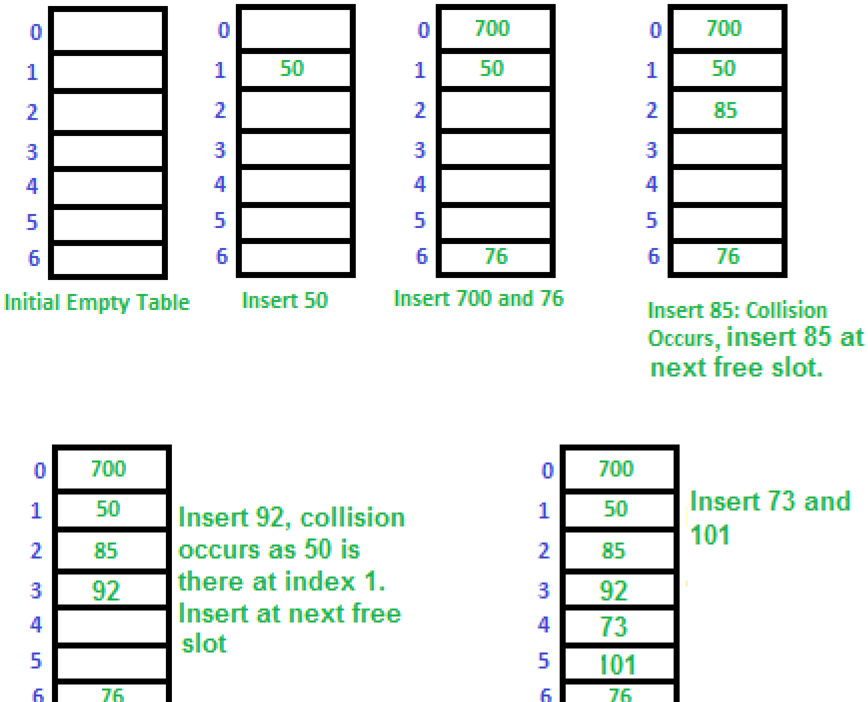
1. Linear probing
2. Quadratic probing
3. Double hashing

LINEAR PROBING

In this technique, if collision occurs, the element has to be inserted into the next free slot.

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.



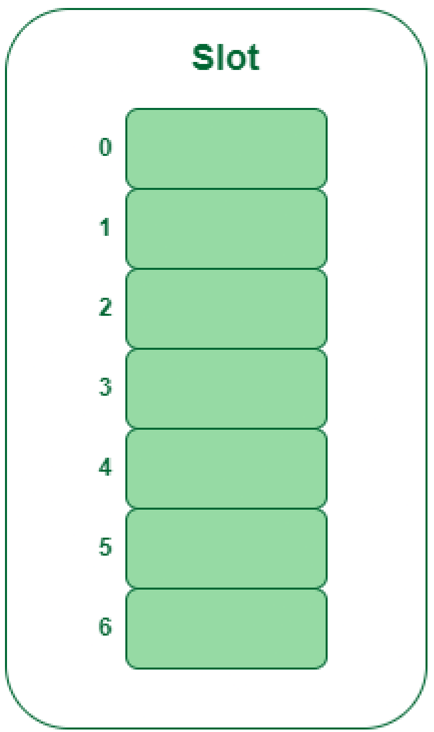
QUADRATIC PROBING

This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.
let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*

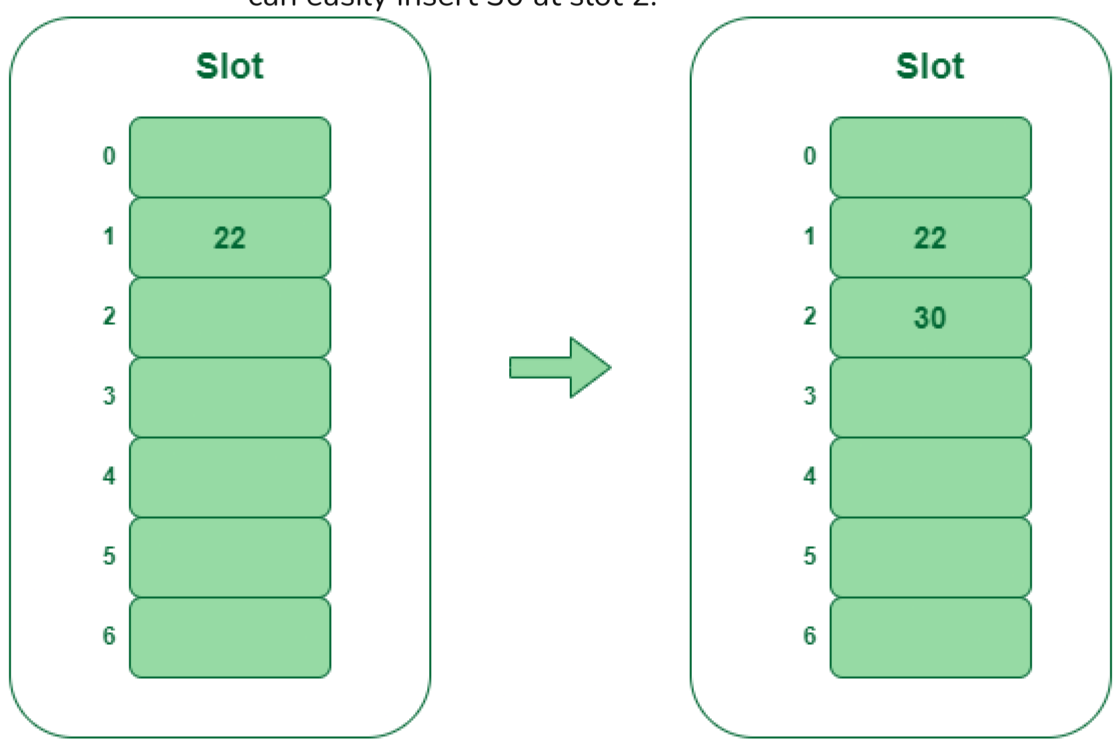
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.

- **Step 1:** Create a table of size 7.



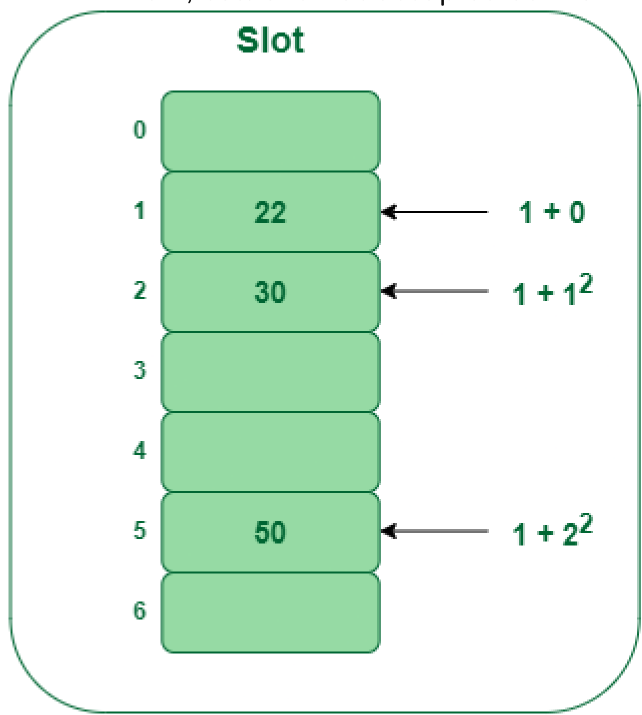
Hash table

- **Step 2** – Insert 22 and 30
    - Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
    - Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

- **Step 3:** Inserting 50
  - Hash(50) = 50 % 7 = 1
  - In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
  - Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
  - Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

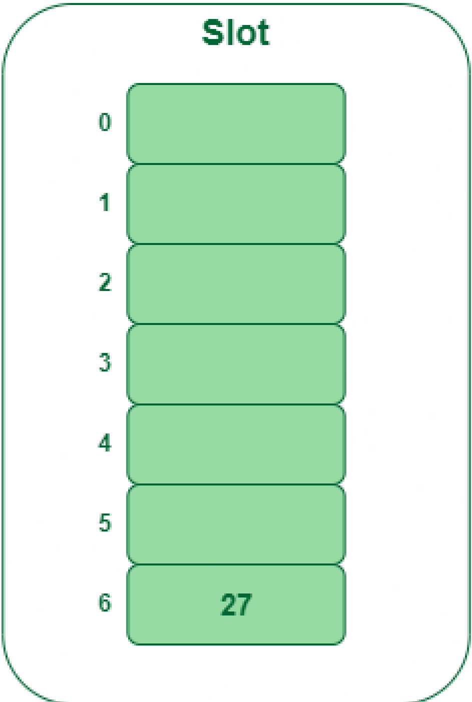
DOUBLE HASHING

In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i**^th^ rotation.
let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

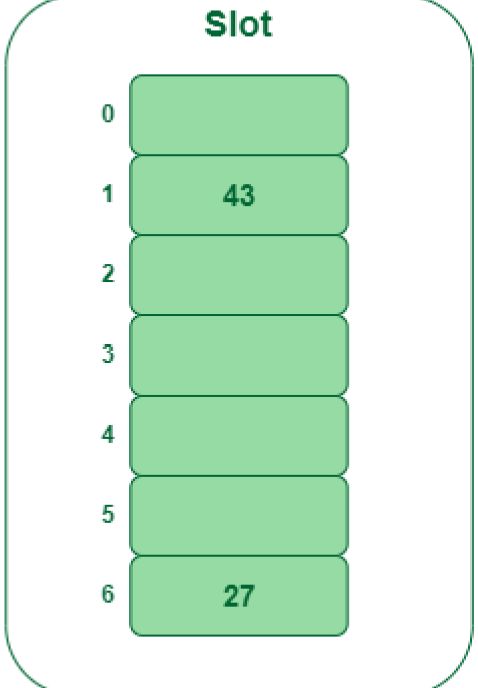
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
  - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

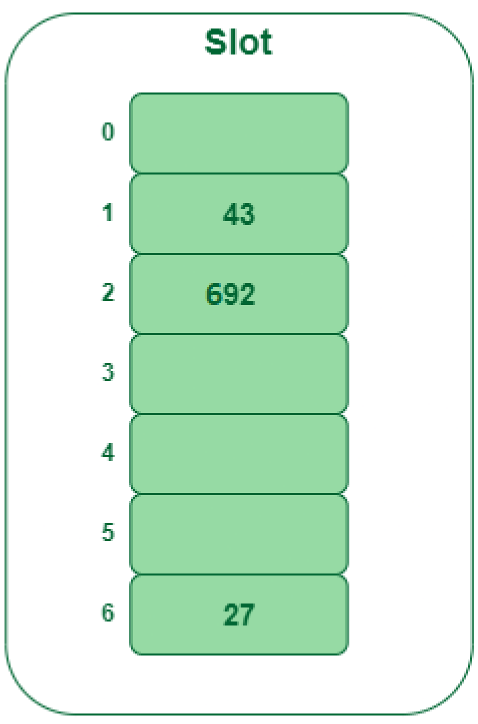
- **Step 2:** Insert 43
  - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



Insert key 43 in the hash table

- **Step 3:** Insert 692
    - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
    - So we need to resolve this collision using double hashing.

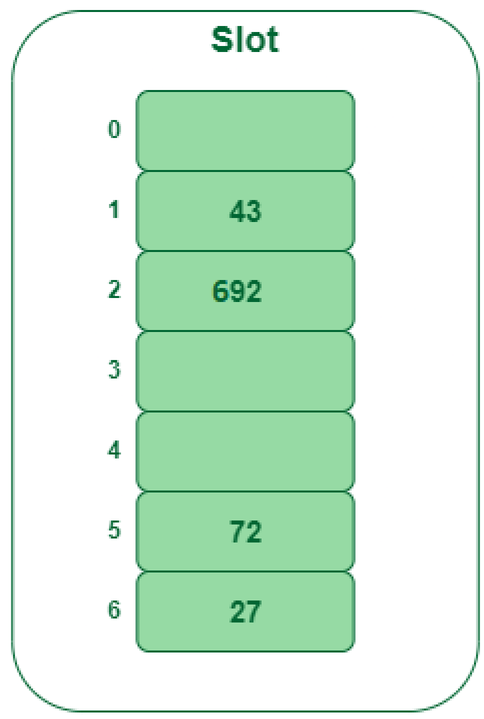
*h_{new} = [h1(692) + i * (h2(692)] % 7*
*= [6 + 1 * (1 + 692 % 5)] % 7*
= 9% 7
= 2
Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.



Insert key 692 in the hash table

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
= 5 % 7
= 5,
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.


Slot
0
1
43
2
692
3
4
5
72
6
27

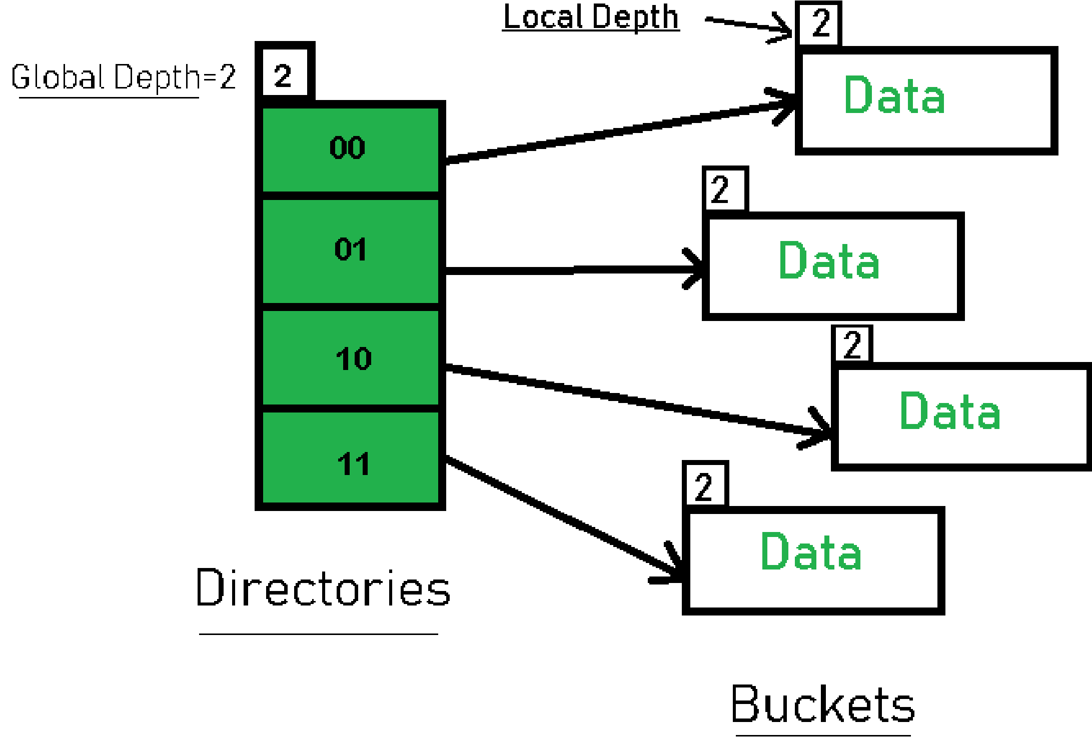
EXTENDIBLE HASHING

It is a dynamic hashing method.

The main terms in hashing technique are:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointer to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth

The basic structure of extendible hashing

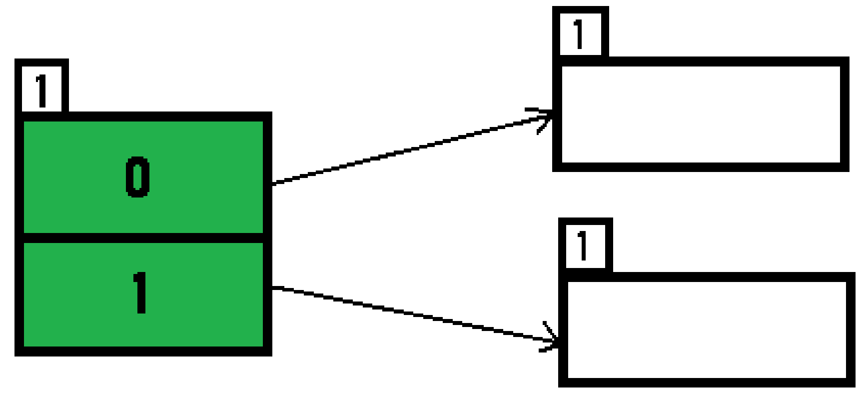


Extendible Hashing

Example

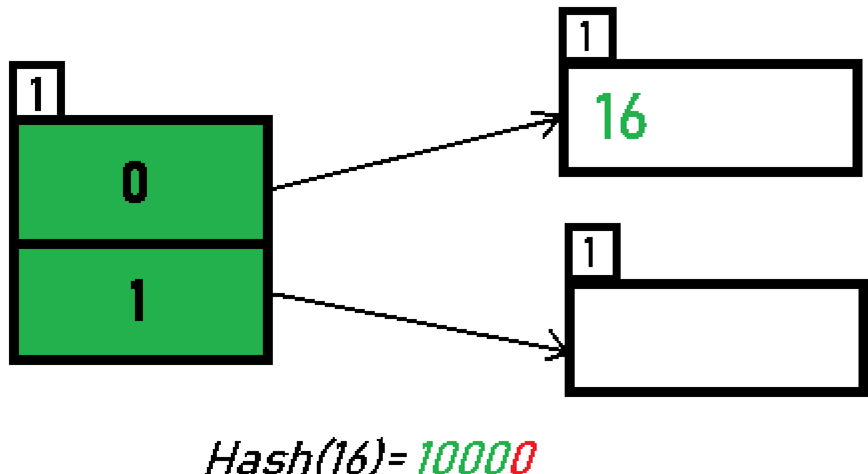
Now, let us consider a example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

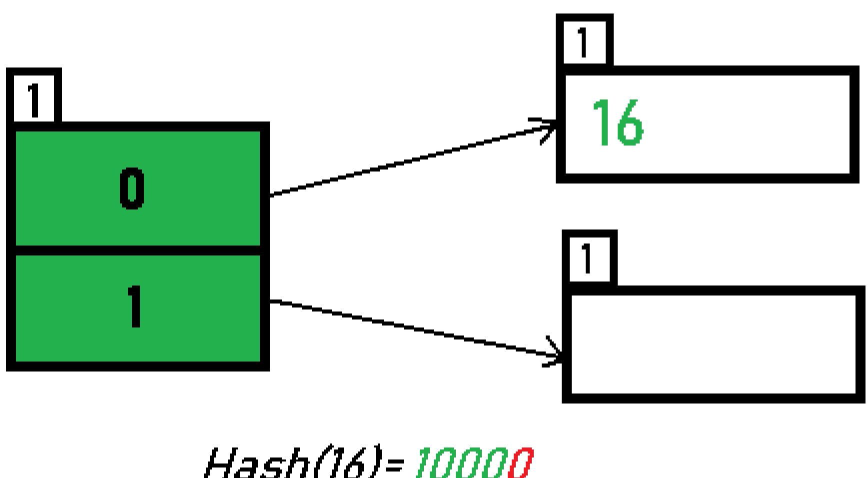
- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 11010
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



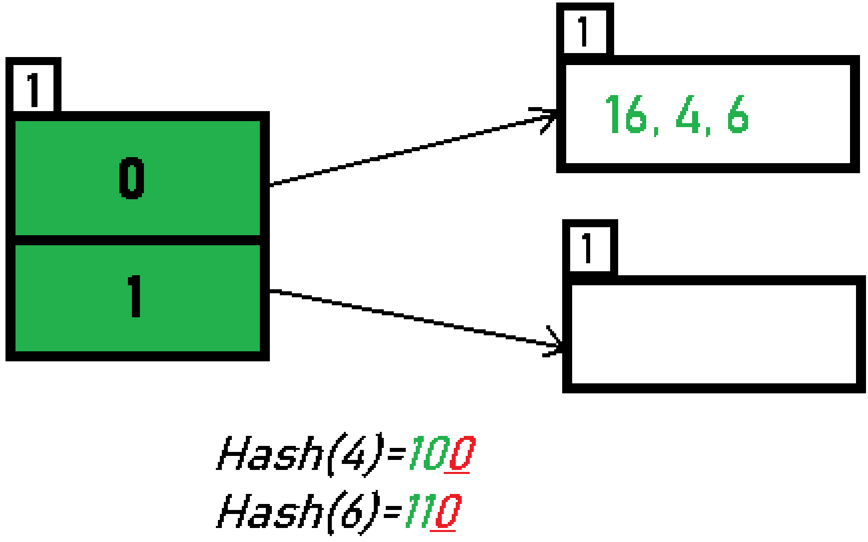
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the

directory with id=0.

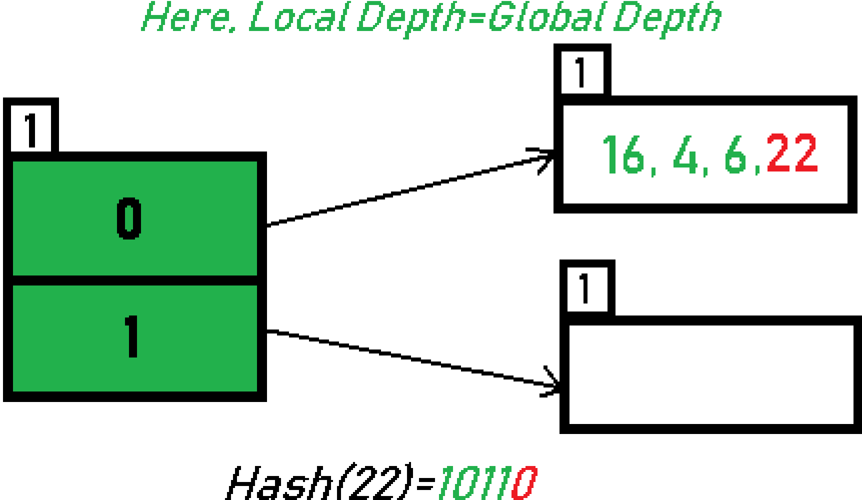




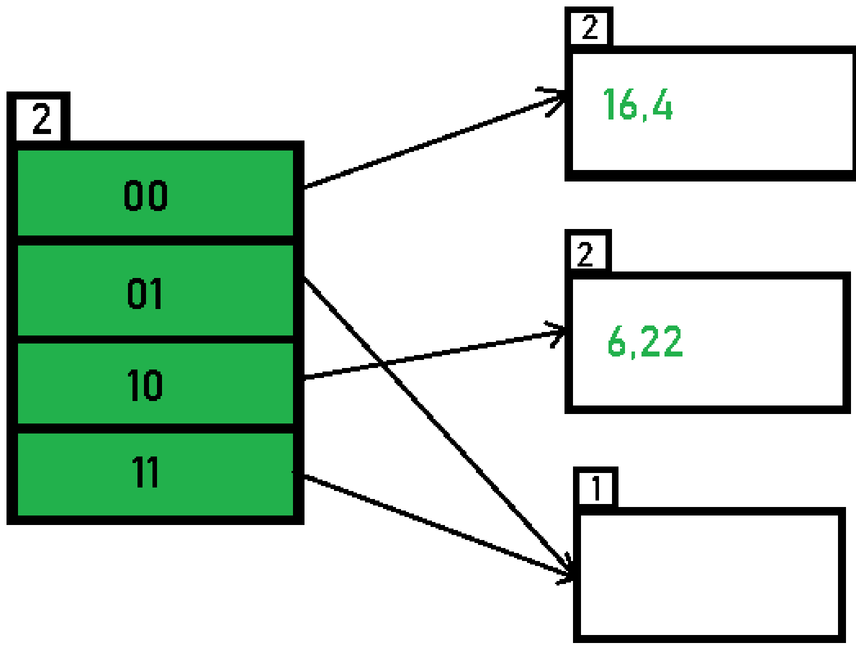
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



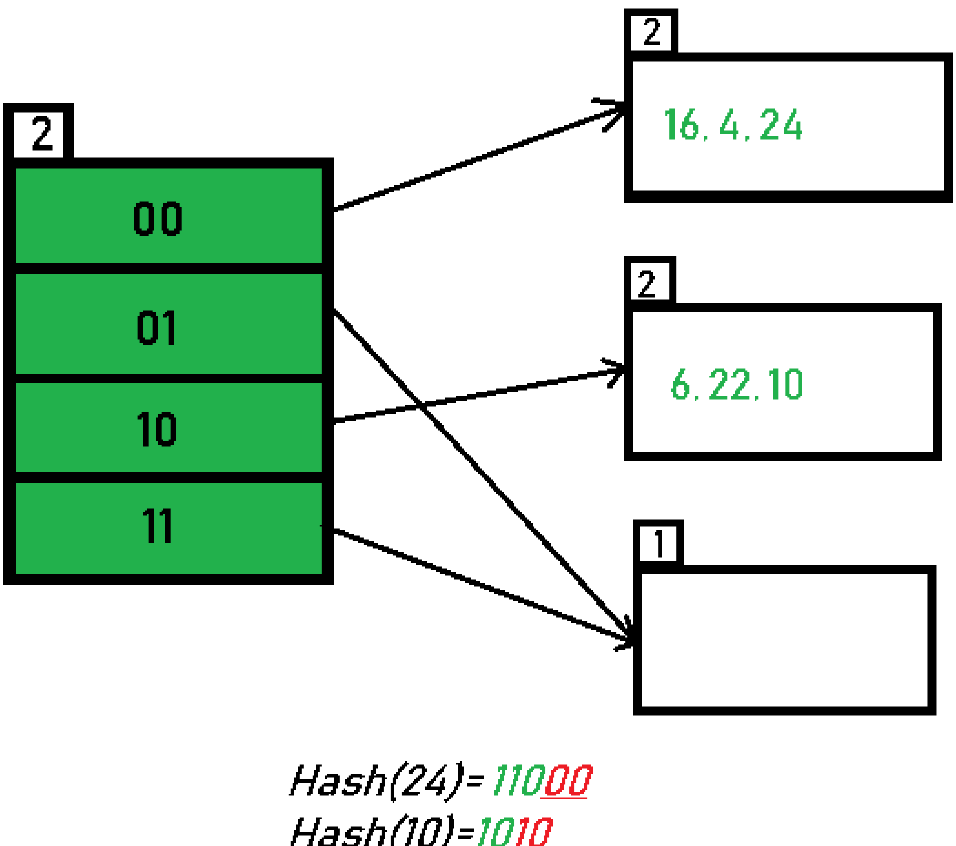
- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



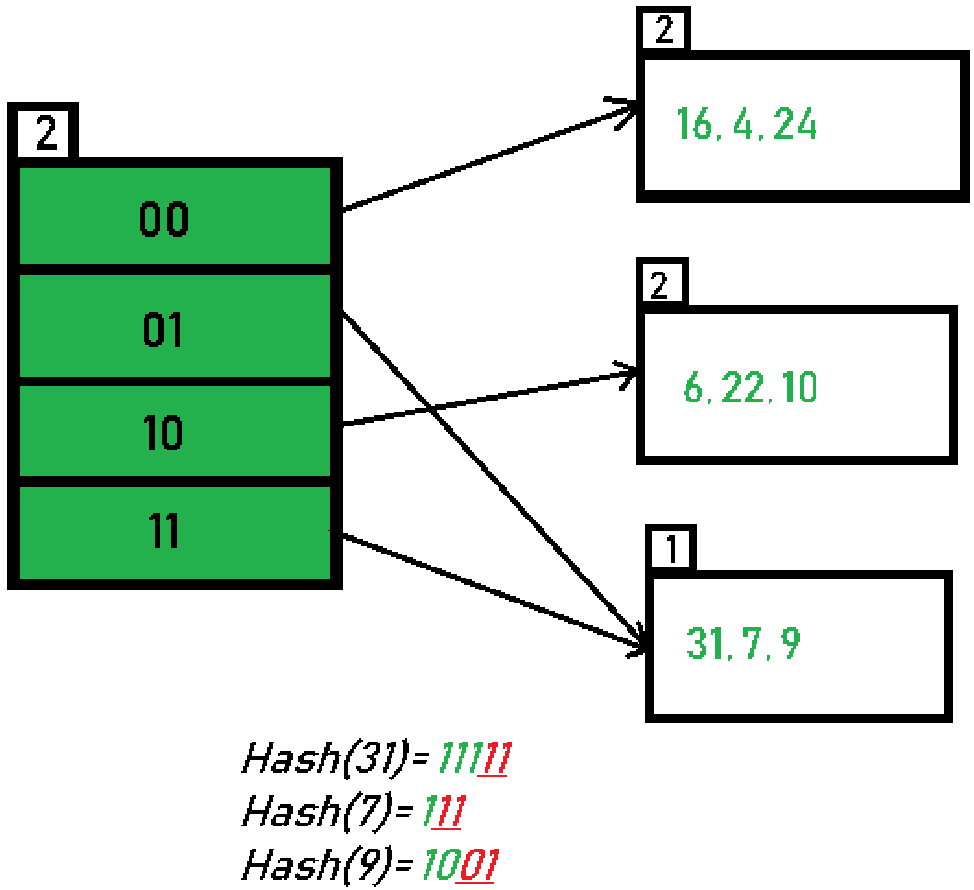
- Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]



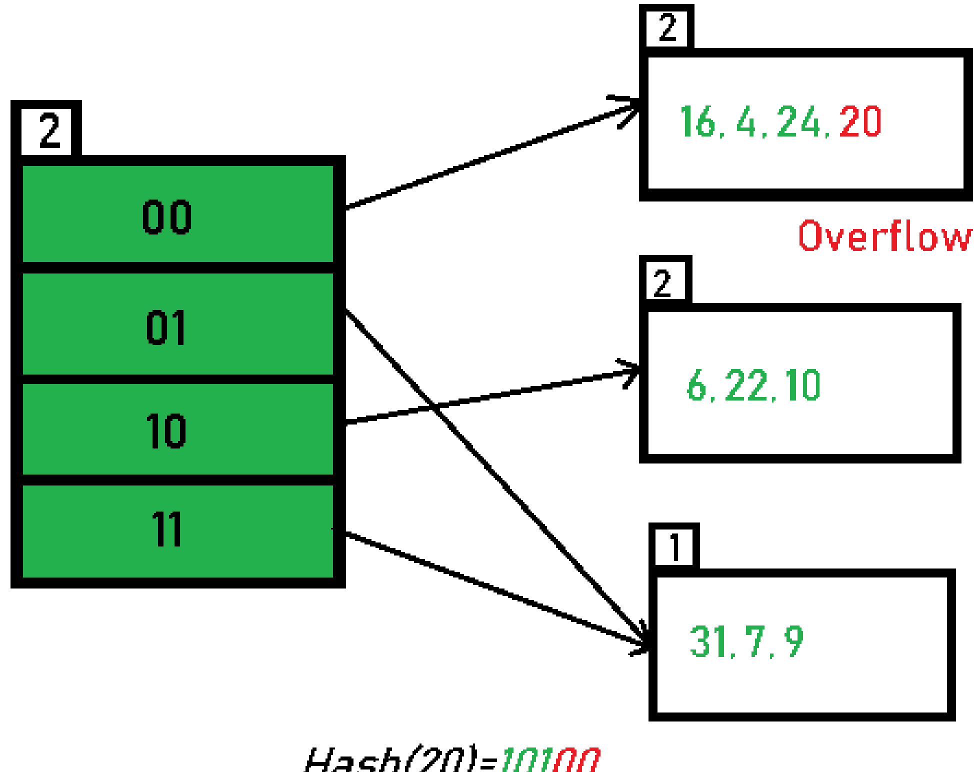
Inserting 24 and 10: 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



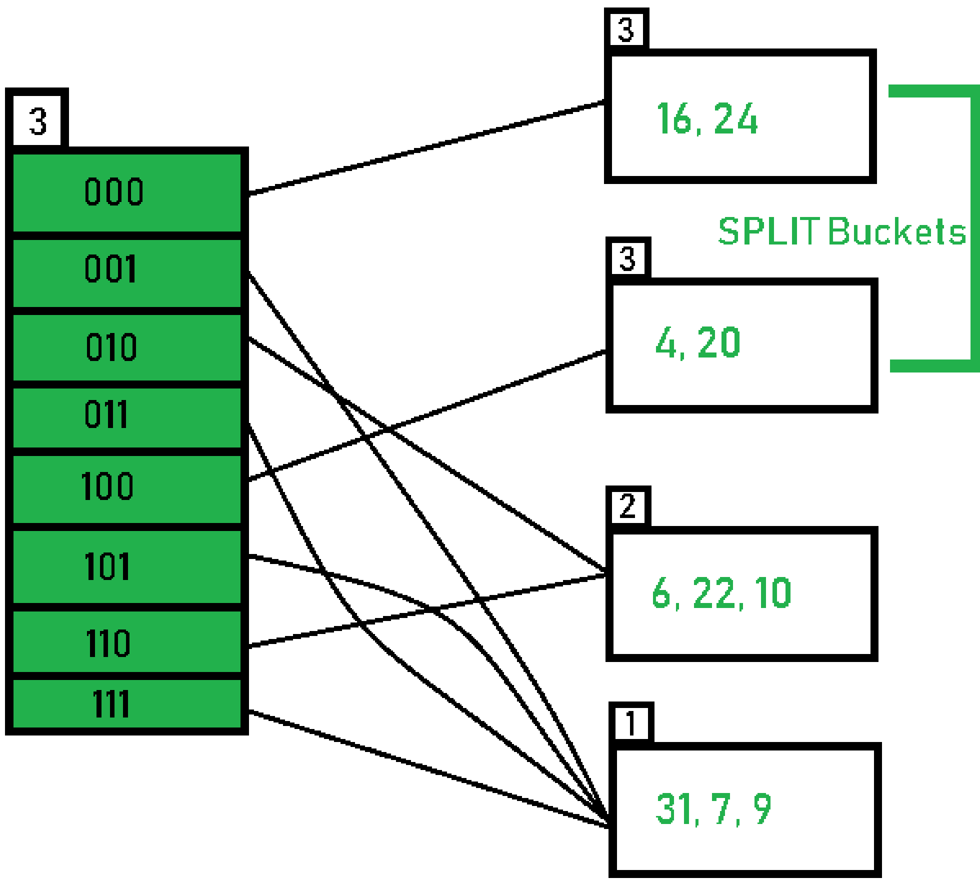
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



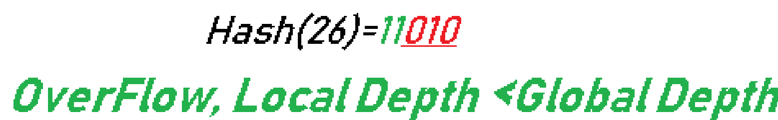
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.

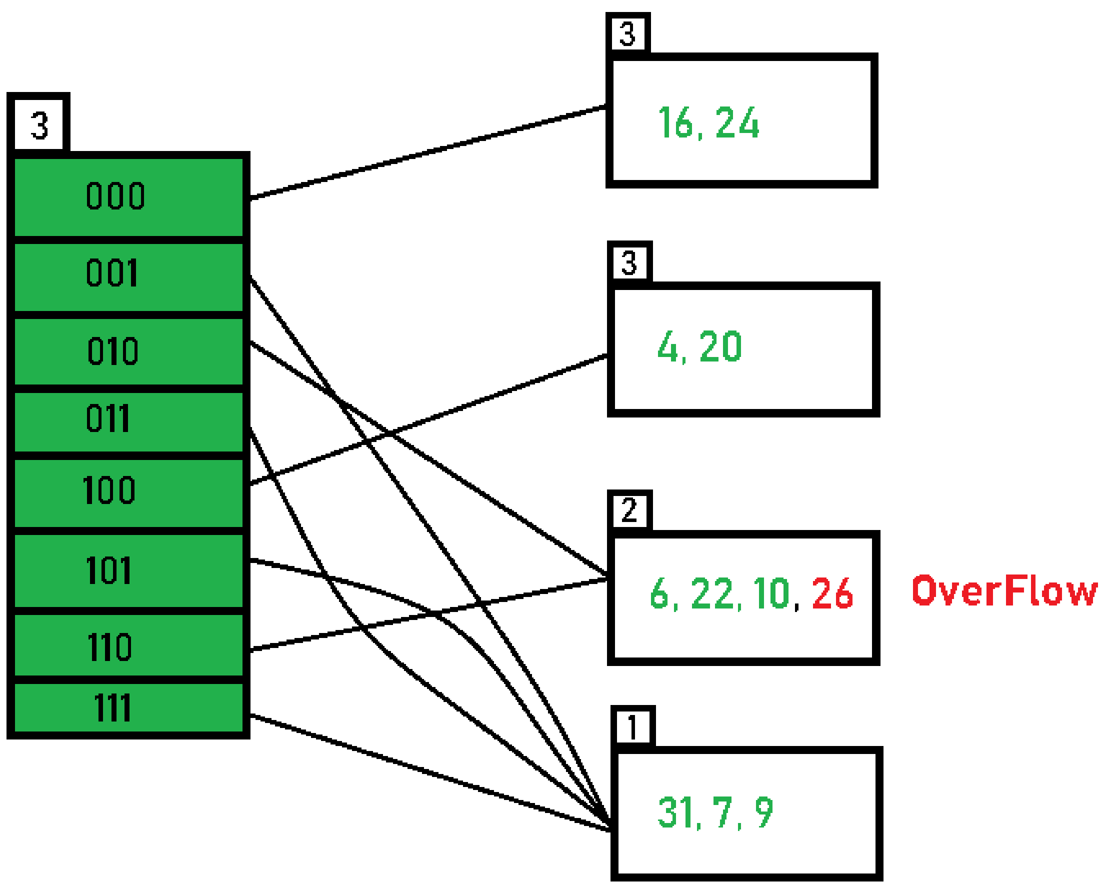


- 20 is inserted in bucket pointed out by 00. Since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



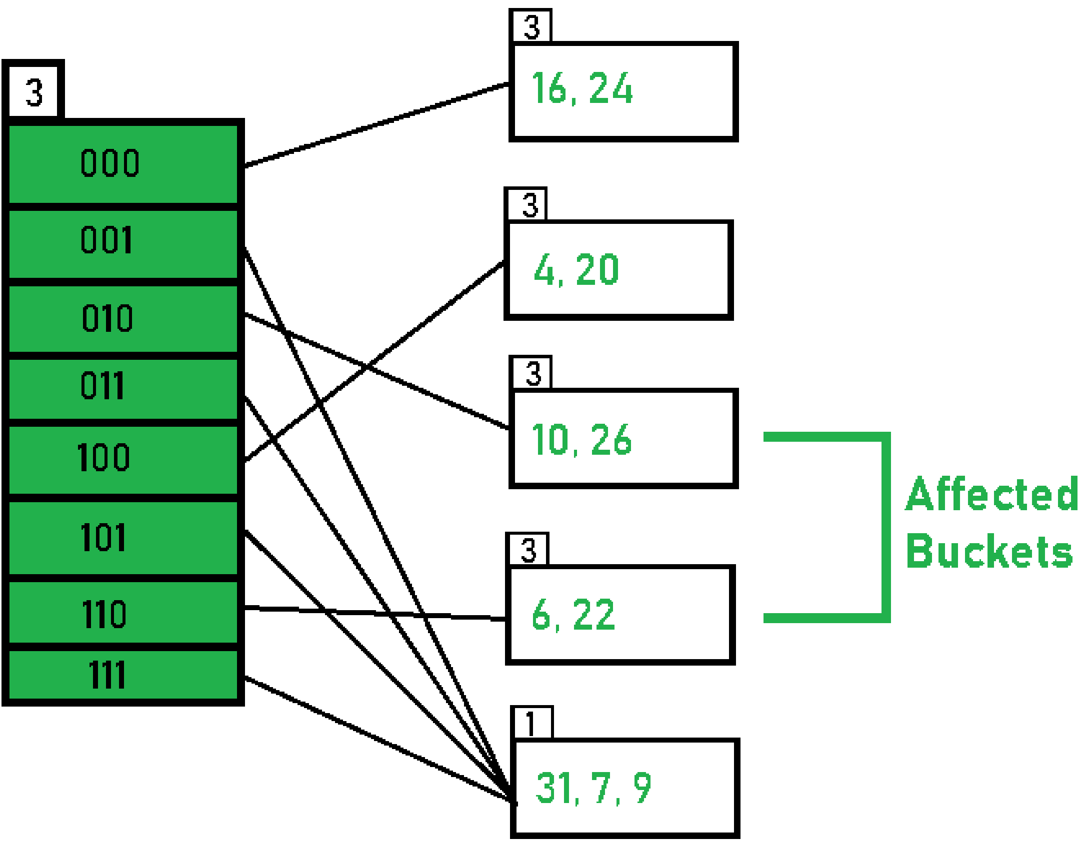
- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.





- The bucket overflows, and, since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.

  Finally, the output of hashing the given list of numbers is obtained.



- **Hashing of 11 Numbers is thus Completed.**