

III SEMESTER

UNIT II

Stacks: Definition – operations - applications of stack. Queues: Definition - operations - Priority queues - De queues – Applications of queue. Linked List: Singly Linked List, Doubly Linked List, Circular Linked List, linked stacks, Linked queues, Applications of Linked List

Two Marks

Linked List

1. **What is a linked list? (Dec2014)**
How is an element of linked list called? What will it contain? (May 2015)

Linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the Next Pointer. The last cell's Next pointer points to NULL.



HEAD-It contains the address of the first node in that list.

NULL - It indicates the end of the list structure.

Stack applications

2. **Give some applications of stack. (May 2015)**
List the application of stacks. (Dec2018)
 - Towers of Hanoi
 - Reversing a string
 - Balanced parenthesis
 - Recursion using stack
 - Evaluation of arithmetic expressions
 - Infix to postfix conversion.
3. **Mention Differentiate stack and Queue. (May 2016)**
Bring out the difference between stack and queue. (May2019)

Queue	Stack
Queue is typically FIFO	Stack is LIFO
Elements get inserted at one end of a queue and retrieved from the other	Insertion and removal operations for a stack are done at the same end.

4. What is the application of stack ADT? (May 2017)

A stack is an ordered list of elements in which elements are always inserted and deleted at one end, say the beginning. In the terminology of stacks, this end is called the top of the stack, whereas the other end is called the bottom of the stack. Also the insertion operation is called push and the deletion operation is called pop.

Linear data structures**5. Give some examples for linear data structures? (Dec2014)**

Non primitive data structures are broadly classified into linear and non linear data structures.

Examples for linear data structures are Lists, Stacks and Queues.

Examples for non linear data structures are Trees and Graphs.

6. What capabilities are needed to make linked representation? (May 2016)

The alternative representation for ordered lists which will reduce the time needed for arbitrary insertion and deletion. The solution to this problem of data movement in *sequential* representations is achieved by using *linked* representations. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory.

Array**7. Define array. (Dec 2016)**

An *array* is a data structure that contains a group of elements. Typically these elements are all of the same data type, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

Bubble sort**8. What is meant by bubble sort? (Dec 2016)**

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element.

Priority Queue**9. Define priority Queue. (May 2017)**

Priority queues are a kind of queue in which the elements are dequeued in priority order.

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed.

Circular Queue**10. Illustrate with an algorithm to count the nodes in a circular queue. (Dec 2017)**

Circular queue is another form of a linear queue in which the last position is connected to the first position of the list. In order to count and add an element, it will be necessary to move *rear* one position clockwise, i.e.,

if *rear* = *n*-1 **then** *rear* = 0

else *rear* = *rear*+1

Linked list**11. Write the advantages of using doubly linked list over singly linked list. (Dec 2017)**

Following are advantages/disadvantages of *doubly linked list* over singly linked list. 1) A DLL can be traversed in both forward and backward direction. 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given. 3) We can quickly insert a new node before a given node.

12. List out the application of linked list. (May2018)

Some of the important applications of *linked lists* are

- manipulation of polynomials,
- sparse matrices,
- stacks and queues.

Circular Linked list**13. What is circular linked list? Give example. (May2019)**

In a *circularly-linked list*, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. They can be seen as having no beginning or end.

14. In what way ENQUEUE operation of circular Queue differ from normal Queue. (Dec2019)

In *circular queue*, the new element is added at the very first position of the queue if the last is occupied and space is available. When it comes to linear queue the insertion can be performed only from the rear end and deletion from the front end.

Queue**15. Define queue with example. (Dec2018)**

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue. The principle of Queue is: The first element inserted into a queue is the first element to be removed. Queue is called First in First out (FIFO) list.

16. Write an algorithm to traverse a single linked list. (Dec2019)

Algorithm

STEP 1: SET PTR = HEAD.

STEP 2: IF PTR = NULL.

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL.

STEP 5: PRINT PTR → DATA.

STEP 6: PTR = PTR → NEXT.

STEP 7: EXIT.

17. Evaluate the value of expression $ab + c * d$ using stack. (May2018)

The expression $A + B * C + D$ can be rewritten as $((A + (B * C)) + D)$ to show ... In postfix, the expression would be $A B C * +$ As a final stack example, we will consider the evaluation of an expression that is Pop the operand Stack and return the value.

11 MARKS**Priority Queues****1. Explain the following: (a) Priority queues (b) De Queues (Dec 2016)**

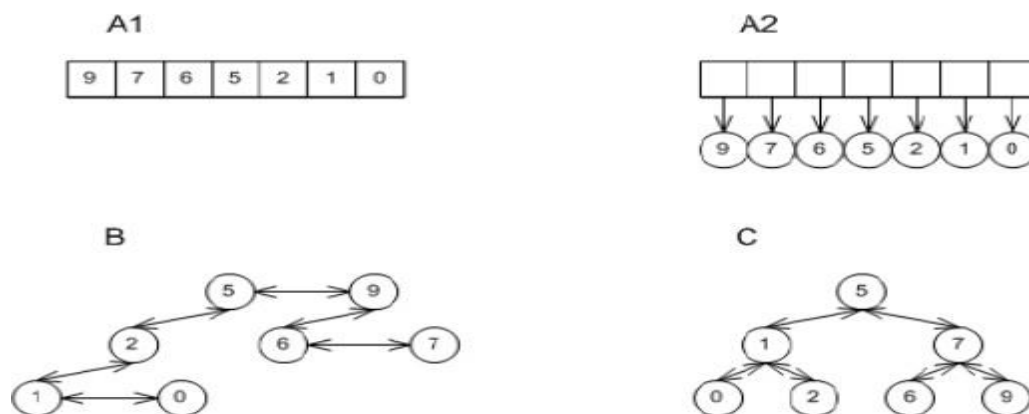
Brief on the following: (a) Priority queues (b) De-queues (c) Applications of queue (May2019)

(a) Priority Queues:

Priority queues are a kind of queue in which the elements are dequeued in priority order.

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed. When using a heap mechanism, a priority queue is quite efficient for this type of operation.

- Each element has a **priority**, an element of a totally ordered set (usually a number).
- More important things come out *first*, even if they were added later.
- Three types of priority : Low priority [10], Normal Priority [5] and High Priority [1].
- There is no (fast) operation to find out whether an arbitrary element is in the queue.

**ALGORITHM:****Priority Queue - Algorithms - Adjust**

Adjust(i)

left = 2i, right = 2i + 1

if left <= H.size and H[left] > H[i]

then largest = left

else largest = i

if right <= H.size and H[right] > H[largest]

then largest = right

if largest != i

then swap $H[\text{largest}]$ with $H[i]$
 Adjust(largest)

Explanation

Adjust works recursively to guarantee the heap property. It compares the current node with its children finding which, if either, has a greater priority. If one of them does, it will swap array locations with the largest child. Adjust is then run again on the current node in its new location.

Priority Queue - Algorithms - Insert

```
Insert(Key)
H.size = H.size + 1
i = H.size
while i > 1 and H[i/2] < Key
  H[i] = H[i/2]
  i = i/2
end while
H[i] = key
```

Explanation

The insert algorithm works by first inserting the new element (key) at the end of the array. This element is then compared with its parent for highest priority. If the parent has a lower priority, the two elements are swapped. This process is repeated until the new element has found its place.

Priority Queue - Algorithms - Extract

```
Extract()
Max = H[1]
H[1] = H[H.size]
H.size = H.size - 1
Adjust(1)
Return Max
```

Explanation

Extract works by removing and returning the first array element, the one of highest priority, and then promoting the last array element to the first. Adjust is then run on the first element so that the heap property is maintained.

Run Time Complexity:

- $\Theta(\log n)$ time for Insert and worst case for Extract (where n is number of elements)
- $\Theta(\log n)$ average time for Insert
- Can construct a Heap from a list in $\Theta(\log n)$ where a Binary Search Tree takes $\Theta(n \log n)$

Space Requirements:

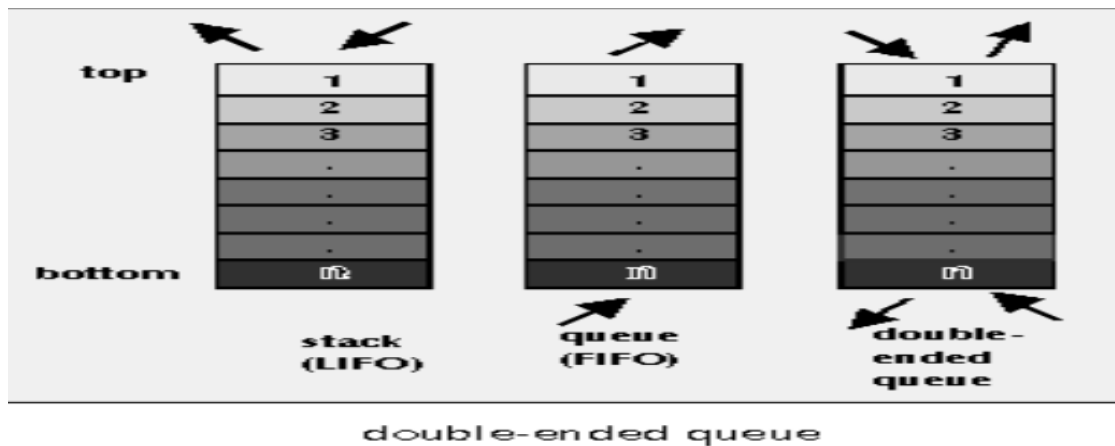
- All operations are done in place - space requirement at any given time is the number of elements, n .

(b) De queues:

Double Ended Queue

A dequeue expanded as *double-ended queue* is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where

elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of dequeues, and can be implemented using dequeues.



Two types of Dequeue are

1. Input Restricted Dequeue
2. Output Restricted Dequeue.

1. Input Restricted Dequeue

Where the input (insertion) is restricted to the rear end and the deletions has the options either end.

2. Output Restricted Dequeue.

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

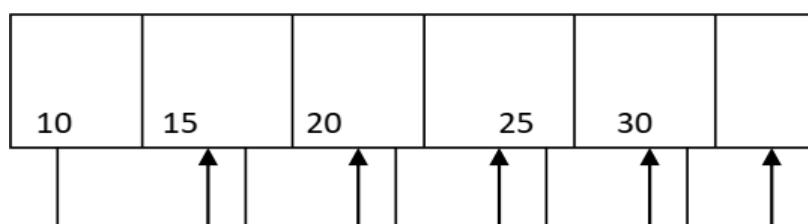
INSERT NEW ELEMENT INTO DEQUEUE:

AT THE END OF THE DEQUEUE:

- First check the dequeue is full or not.
- We have to check whether the element is first to be added or inserted if it is true assign front and rear is 0.
- Insert a new element.

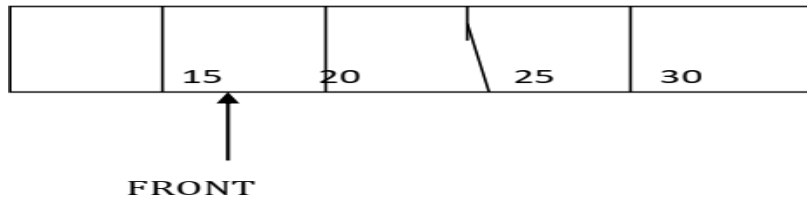
AT THE BEGINNING OF THE DEQUEUE:

- First check the dequeue is full or not.
- We have to check whether the element is first to be inserted if it is true then perform the following steps
 1. Shift the elements from left to right
 2. So that we need the number of elements present in the dequeue, the position of the last element (ie) the position of the rear
 3. Now insert any element in zeroth position.

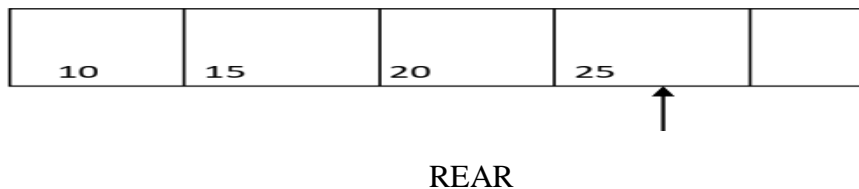


DELETING THE ELEMENT FROM DEQUEUE:**AT THE BEGINNING OF DEQUEUE:**

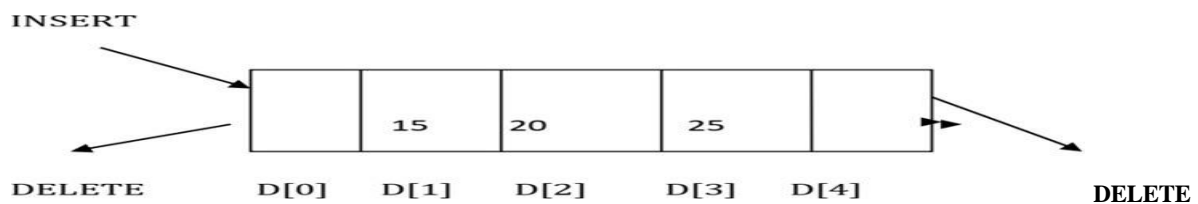
- delete the first element from the front position of the dequeue
- the index of next element is stored in front pointer.
- Increment the front pointer by one.

**AT THE END OF DEQUEUE:**

- The last element is deleted.
- The index of next element is stored in rear pointer.
- Decrement the rear pointer by one.

**INPUT RESTRICTED DEQUEUE:**

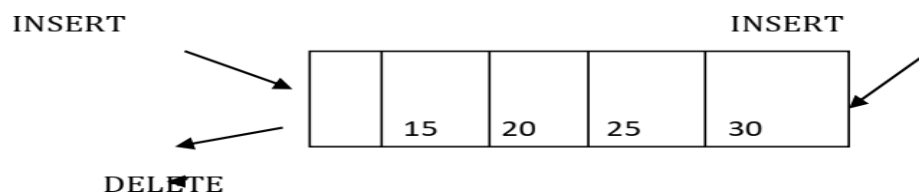
- It means we can insert the element only at one end and delete the elements at both ends.



- The above diagram shows the input restricted dequeue.

OUTPUT RESTRICTED DEQUEUE:

- It means we can insert the elements in both ends and delete the elements at only one end.



- The above diagram shows the output restricted dequeue.

Linked list

2. Briefly explain the linked list. (Dec 2016)

Linked List

❖ Some demerits of array, leads us to use linked list to store the list of items. They are,

1. It is relatively expensive to insert and delete elements in an array.
2. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. For this reason, arrays are called “**dense lists**” and are said to be “**static**” data structures.

❖ A **linked list**, or **one-way list**, is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. That is, each node is divided into two parts:

✓ The first part contains the information of the element i.e. INFO or DATA.

✓ The second part contains the **link field**, which contains the address of the next node in the list.



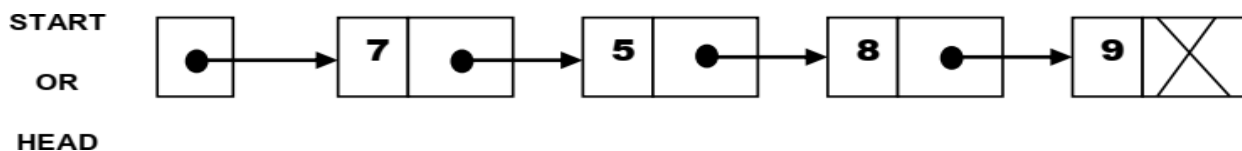
❖ The linked list consists of series of nodes, which are not necessarily adjacent in memory.

❖ A list is a **dynamic data structure** i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.

❖ The pointer of the last node contains a special value, called the **null** pointer, which is any invalid address. This **null pointer** signals the end of list.

❖ The list with no nodes on it is called the **empty list** or **null list**.

Example: The linked list with 4 nodes.

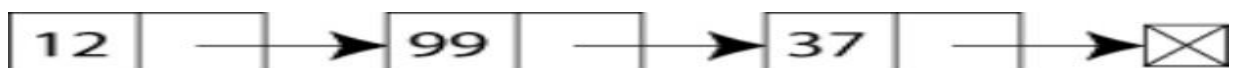


Types of Linked Lists:

- a) Singly Linked List
- b) Circular Linked List
- c) Two-way or doubly linked lists
- d) Circular doubly linked lists

Singly-linked list

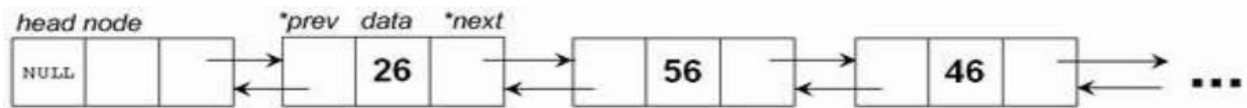
The simplest kind of linked list is a **singly-linked list** or **slist** for short, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.



A singly linked list containing three integer values

Doubly-linked list

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.



An example of a doubly linked list.

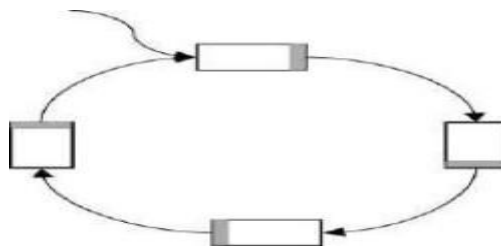
Circularly-linked list

In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list. The pointer pointing to the whole list is usually called the end pointer.

Singly-circularly-linked list

In a **singly-circularly-linked list**, each node has one link, similar to an ordinary *singly-linked list*, except that the next link of the last node points back to the first node.

As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly-circularly linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.



- Note that there is no **NULL** terminating pointer
- Choice of **head** node is arbitrary

Advantages of Linked List:

1. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
2. The Linked List does not waste memory
3. It is not necessary to specify the size of the list, as in the case of arrays.
4. Linked List provides the flexibility in allowing the items to be rearranged.

3. Write detailed notes on singly linked list. (May 2016)

Illustrate the algorithm to create the singly linked list and perform all the operations on the created list. (May2017)

SINGLY / LINEAR LINKED LIST:

In a sequential representation, suppose that the items were implicitly ordered, that is, each item contained within itself the address of the next item, such an implicit ordering gives rise to a data structure known as a Linear linked list or Singly linked list which is shown in figure:



- information field called INFO that holds the actual element on the list and
- a pointer pointing to next element of the list called LINK, ie. It holds the address of the next element. The name of a typical element is denoted by NODE.

INFO	LINK
NODE	

```

graph TD
    AVAIL --> Node1
    Node1 --> Node2
    Node2 --> Node3
    Node3 --> Null
    style Null fill:none,stroke:none
  
```

OPERATIONS:

Inserting a new item, say 'x', into the list has three situations:

1. Insertion at the front of the list
2. Insertion in the middle of the list or in the order
3. Insertion at the end of the list

Algorithm:

1. Obtain space for new node
2. Assign data to the item field of new node
3. Set the next field of the new node to point to the start of the list
4. Change the head pointer to point to the new node.

Function INSERT(X, FIRST)

Variables used:

$X \leftarrow$ New element to be inserted

$FIRST \leftarrow$ Pointer to the first element whose node contains INFO and LINK fields.

$AVAIL \leftarrow$ Pointer to the top element of the availability stack

$NEW \leftarrow$ Temporary pointer variable.

1. [Underflow?]

If $AVAIL = NULL$

Then Write ('AVAILABILITY STACK UNDERFLOW')

Return (FIRST)

2. [Obtain address of next free node]

$NEW \leftarrow AVAIL$

3. [Remove free node from availability stack]

$AVAIL \leftarrow LINK (AVAIL)$

4. [Initialize fields of new node and its link to the list]

$INFO (NEW) \leftarrow X$

$LINK(NEW) \leftarrow FIRST$

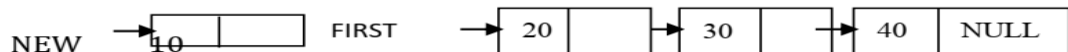
5. [Return address of new node]

Return (NEW)

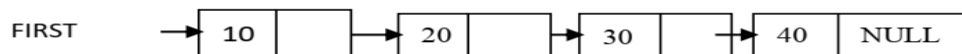
Step 1:



Step 2:



Step 3:



INSERTION IN THE MIDDLE OF THE LIST

Algorithm:

1. Set space for new node x

2. Assign value to the item field of x

3. Search for predecessor node n1 of x

4. Set the next field of x to point to link of node n1 (node N2)

5. Set the next field of N1 to point to x.

Function INSORD(X, FIRST)

Variables used:

$X \leftarrow$ new element

$FIRST \leftarrow$ Pointer to the first element whose node contains INFO and LINK fields.

$AVAIL \leftarrow$ pointer to the top element of the availability stack

$NEW, SAVE \leftarrow$ Temporary pointer variables

1. [Underflow?]

If $AVAIL = NULL$

Then Write ('AVAILABILITY STACK UNDERFLOW')

Return (FIRST)

2. [Obtain address of next free node]

NEW \leftarrow AVAIL

3. [Remove free node from availability stack]

AVAIL \leftarrow LINK (AVAIL)

4. [Copy information contents into new node]

INFO (NEW) \leftarrow X

5. [Is the list empty?]

If FIRST = NULL

Then LINK (NEW) \leftarrow FIRST

6. [Does the new node precede all others in the list?]

If INFO(NEW) \leq INFO(FIRST)

Then LINK(NEW) \leftarrow FIRST

Return(NEW)

7. [Initialize temporary pointer]

SAVE \leftarrow FIRST

8. [Search for predecessor of new node]

Repeat while LINK (SAVE) \neq NULL and INFO (LINK (SAVE)) \leq INFO(NEW)

SAVE \leftarrow LINK(SAVE)

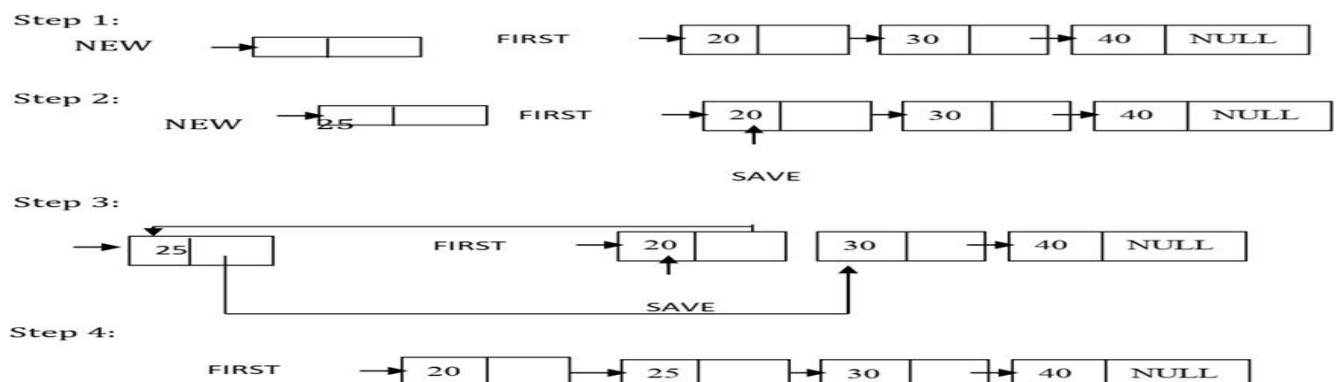
9. [Set link fields of new node and its predecessor]

LINK (NEW) \leftarrow LINK (SAVE)

LINK (SAVE) \leftarrow NEW

10. [Return first node pointer]

Return (FIRST)



INSERTION AT THE END OF THE LIST

Algorithm:

1. Set space for new node x

2. Assign value to the item field of x

4. Set the next field of N2 to point to x

Function INSEND(X,FIRST)

Variables used:

X \leftarrow new element

FIRST \leftarrow Pointer to the first element whose node contains

INFO and LINK fields.

AVAIL \leftarrow pointer to the top element of the availability stack

NEW,SAVE \leftarrow Temporary pointer variables

1. [Underflow?]

If AVAIL = NULL

Then Write('AVAILABILITY STACK UNDERFLOW')

Return(FIRST)

2. [Obtain address of next free node]

NEW \leftarrow AVAIL

3. [Remove free node from availability stack]

AVAIL \leftarrow LINK(AVAIL)

4. [Initialize fields of new node]

INFO(NEW) \leftarrow X

LINK(NEW) \leftarrow NULL

5. [Is the list empty?]

If FIRST = NULL

Then Return(NEW)

6. [Initiate search for the last node]

SAVE \leftarrow FIRST

7. [Search for end of list]

Repeat while LINK(SAVE) \neq NULL

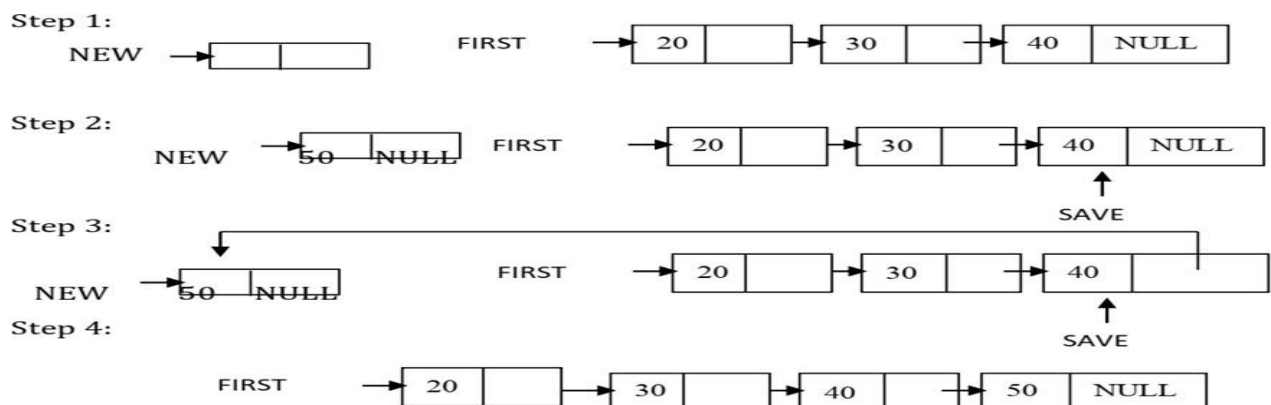
SAVE \leftarrow LINK(SAVE)

8. [Set LINK field of last node to NEW]

LINK(SAVE) \leftarrow NEW

9. [Return first node pointer]

Return(FIRST)



Note that no data is physically moved, as we had seen in array implementation. Only the pointers are readjusted.

DELETING AN ITEM FROM A LIST:

Deleting a node from the list requires only one pointer value to be changed, there we have situations:

1. Deleting the first item
2. Deleting the last item
3. Deleting between two nodes in the middle of the list.

Algorithms for deleting the first item:

1. If the element x to be deleted is at first store next field of x in some other variable y.
2. Free the space occupied by x
3. Change the head pointer to point to the address in y.

Algorithm for deleting the last item:

1. Set the next field of the node previous to the node x which is to be deleted as NULL

2. Free the space occupied by x

Algorithm for deleting x between two nodes N1 and N2 in the middle of the list:

1. Set the next field of the node N1 previous to x to point to the successor field N2 of the node x.

2. Free the space occupied by x

Procedure DELETE(X, FIRST)

Variables used:

X \leftarrow New element to be inserted

FIRST \leftarrow Pointer to the first element whose node contains
INFO and LINK fields.

TEMP \leftarrow To find the desired node

PRED \leftarrow keeps track of the predecessor of TEMP

1. [Empty list?]

If FIRST = NULL

Then Write('UNDERFLOW')

Return

2. [Initialize search for X]

TEMP \leftarrow FIRST

3. [Find X]

Repeat thru step 5 while TEMP \neq X and LINK(TEMP) \neq NULL

4. [Update predecessor marker]

PRED \leftarrow TEMP

5. [Move to next node]

TEMP \leftarrow LINK(TEMP)

6. [End of the list]

If TEMP \neq X

Then Write('NODE NOT FOUND')

Return

7. [Delete X]

If X = FIRST (Is X the first node?)

Then FIRST \leftarrow LINK(FIRST)

Else LINK(PRED) \leftarrow LINK(X)

8. [Return node to availability area]

LINK(X) \leftarrow AVAIL

AVAIL \leftarrow X

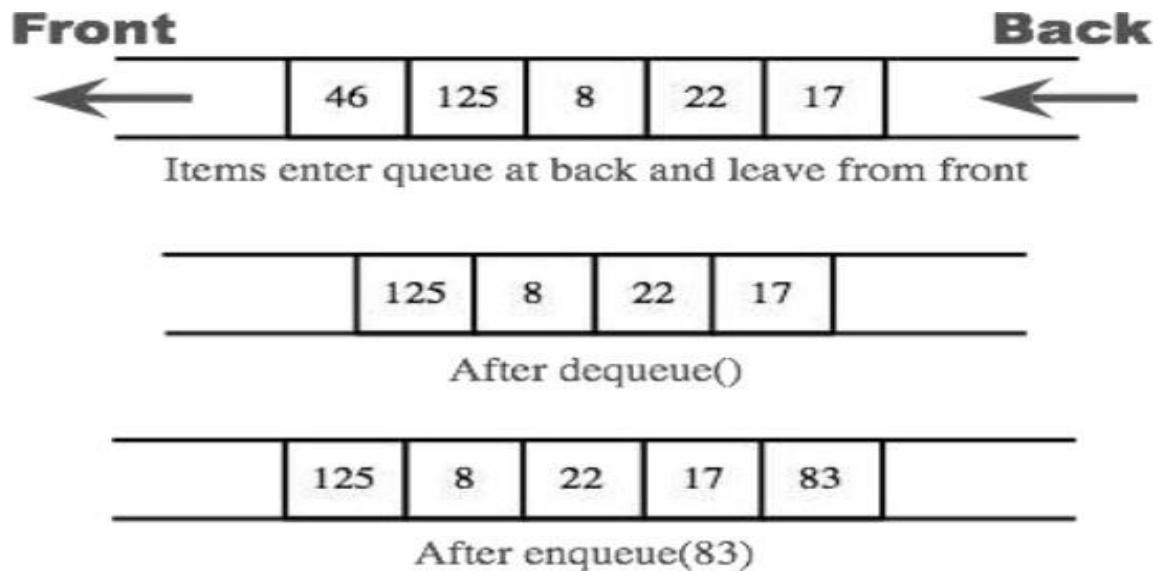
Return

Queue operation**4. Explain the operation performed on Queue in detail. (May2017)**

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

PRINCIPLE

The first element inserted into a queue is the first element to be removed. Queue is called First In First Out (FIFO) list.

BASIC OPERATIONS INVOLVED IN A QUEUE:

1. Create a queue
2. Check whether a queue is empty or full
3. Add an item at the rear end
4. Remove an item at the front end
5. Read the front of the queue
6. Print the entire queue

INSERTION OPERATION:

An attempt to push an item onto a queue, when the queue is full, causes an overflow.

1. Check whether the queue is full before attempting to insert another element.
2. Increment the rear pointer & 3. Insert the element at the rear pointer of the queue.

ALGORITHM:

Rear – Rear end pointer, Q – Queue, N – Total number of elements & Item – The element to be inserted

1. if(Rear=N) [Overflow?]

Then Call QUEUE_FULL

Return

2. Rear<-Rear+1 [Increment rear pointer]

```
3. Q[Rear]<-Item [Insert element]
End INSERT
```

DELETION OPERATION:

An attempt to remove an element from the queue when the queue is empty causes an underflow.

Deletion operation involves:

1. Check whether the queue is empty.
2. Increment the front pointer.
3. Remove the element.

ALGORITHM:

Q – Queue, Front – Front end pointer, Rear – Rear end pointer & Item – The element to be deleted.

1.if (Front=Rear) [Underflow?]

Then Call QUEUE_EMPTY

2. Front<-Front+1 [Incrementation]

3. Item<-Q [Front] [Delete element]

Thus queue is a dynamic structure that is constantly allowed to grow and shrink and thus changes its size, when implemented using linked list.

5. Describe the linked list implementation of queue with insertion, deletion operation. (Dec 2017)

Like stacks, queues are lists. With a queue, however, insertion is done at one end, whereas deletion is performed at the other end. The basic operations on a queue are enqueue, which inserts an element at the end of the list called the rear, and dequeue, which deletes and returns the element at the start of the list known as the front.

- Figure shows the abstract model of a queue. The queues can be implemented using arrays or pointers.
- Queue is implemented using Singly Linked Lists, where the next pointer of the element at the rear end is made to point to NULL.

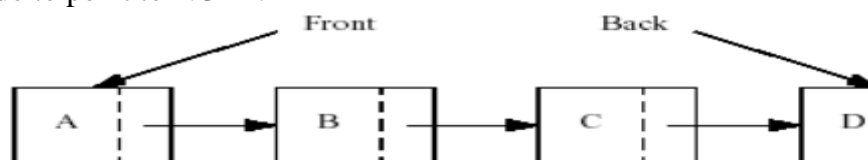


Fig: Linked List Implementation of Queue

Queue Model

The basic operations on a queue are *enqueue*, which inserts an element at the end of the list (called the rear), and *dequeue*, which deletes (and returns) the element at the start of the list (known as the front). Figure shows the abstract model of a queue.

Structure definition

Each node contains two fields. Every last node points to null.

```
struct node
{
    element type element;
    ptr to node*next;
```



```
};
typedef node_ptr queue;
```

Function to check whether the queue is empty or not

This function is empty checks whether the queue is empty or not, It returns true, if the queue is empty. It makes the q->next to point to NULL in case if the queue is empty.

```
int is_Empty (queue q)
{
    return q->next == NULL;
}
```

Creating a Queue

It allocates memory for the given structure (header of the queue. It points to the element at the front of the queue). The queue is emptied by calling makeempty() function.

```
queue create queue (void)
{
    queue q;
    q=malloc (sizeof(struct node));
    if(q == null)
        fatal error("OUT OF SPACE");
    makeempty(q);
    return q;
}
```

Function to empty the queue

If the queue is not empty, the elements in the queue are removed until the queue becomes empty. In other words, the queue exists but the queue is empty.

```
void make empty(queue q)
{
    if(q == null)
        error("Must use create queue first");
    else
        while (!isempty (q));
        delete(q);
}
```

6. Write a C++ code to perform subtraction of two polynomials using linked list. (May2018)

```
void sub ( )
{
    poly *ptr1, *ptr2, *newnode;
    ptr1 = list1 ;
    ptr2 = list 2;
    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 (list 3, newnode);
    ptr1 = ptr1 → next;
    ptr2 = ptr2 → next;
}
else
{
    if (ptr1 → power > ptr2 → power)
    {
        newnode → coeff = ptr1 → coeff;
        newnode → power = ptr1 → power;
        newnode → next = NULL;
        list 3 = create (list 3, newnode);
        ptr1 = ptr1 → next;
    }
}
}

```

7. What is doubly linked list? Write an algorithm to implement the following expression.
(Dec2018)

(a) Insertion (all cases)

(b) Traversal (Forward and Backward)

In the singly linked list ,we traverse the list in one dimension .In many application it is required to traverse a list in both direction .This 2 way traverse a list in both direction .This 2 way traverse can be realized by maintaining 2 link fields in each node instead of one .Each element of a doubly linked list structure has three fields

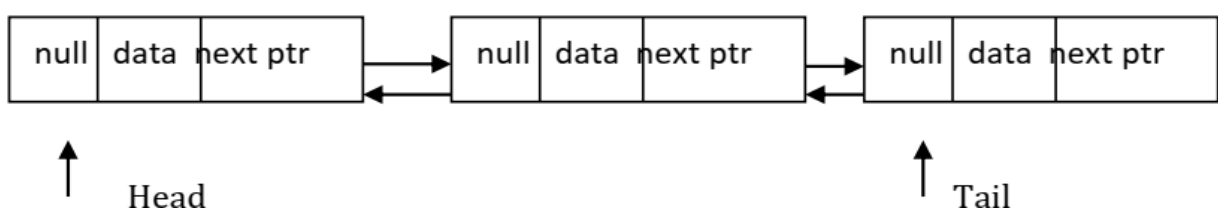
-data value

-a link to its successor

-a link to its predecessor

The predecessor link is called left link the successor link is known as right link

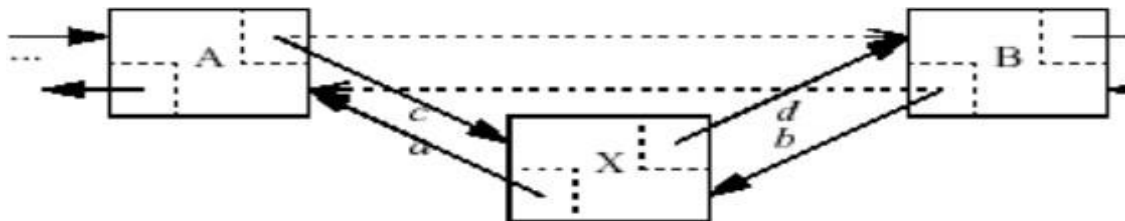
Thus the traversal of the list can be in any direction. Such a structure is shown in following figure:



Insertion of a node:

To insert a node into a doubly linked list to the right of a specified node, we have to consider several case, these are as follows:

1. If the list is empty, insert a new node and make left and right link of new node to be set to nil.
2. If there is a predecessor and a successor to the given node .In such a case need to readjust link of the specified node and its successor node



Insertion into a doubly linked list by getting new node and then changing pointers in the order indicated:

Procedure DOUBINS (L,R,M,X)**Variables used:**

$L \leftarrow$ left most node address
 $R \leftarrow$ right most node address
 $NEW \leftarrow$ New node address to be inserted
 $LPTR \leftarrow$ Left link of a node
 $RPTR \leftarrow$ Right link of a node
 $INFO \leftarrow$ Information field of the node
 $NODE \leftarrow$ Name of an element of the list
 $M \leftarrow$ Pointer variable
 $X \leftarrow$ It contains information to be entered in the node
 1. [Obtain new node from availability stack]
 $NEW \leftarrow NODE$
 2. [Copy information field]
 $INFO(NEW) \leftarrow X$
 3. [Insertion into an empty list?]
 If $R = NULL$
 Then $LPTR(NEW) \leftarrow RPTR(NEW) \leftarrow NULL$
 $L \leftarrow R \leftarrow NEW$
 Return
 4. [Left most insertion?]
 If $M = L$
 Then $LPTR(NEW) \leftarrow NULL$
 $RPTR(NEW) \leftarrow M$
 $LPTR(M) \leftarrow NEW$
 $L \leftarrow NEW$
 Return
 5. [Insert in middle]
 $LPTR(NEW) \leftarrow LPTR(M)$
 $RPTR(NEW) \leftarrow M$
 $LPTR(M) \leftarrow NEW$

RPTR(LPTR(NEW)) \leftarrow NEW

Return

3. If the insertion has to be done after the right most node in this list. In such a case only the right link of the specified node ie, the right most is to be changed.

DELETION OF A NODE:

Similar to insertion of a node ,we have to consider several case for deletion of a node.

Procedure DOUBDEL (L, R, OLD)

Variables used:

L \leftarrow Left most node

R \leftarrow Right most node

LPTR \leftarrow Left link of a node

RPTR \leftarrow Right link of a node

OLD \leftarrow The address of the node to be deleted

1. [Underflow?]

If R = NULL

Then Write('UNDERFLOW')

Return

2. [Delete node]

If L = R (Single node in list)

Then L \leftarrow R \leftarrow NULL

Else IF OLD = L (Left most node being deleted)

Then L \leftarrow RPTR(L)

LPTR(L) \leftarrow NULL

Else If OLD = R (Right most node being deleted)

Then R \leftarrow LPTR(R)

RPTR(R) \leftarrow NULL

Else RPTR(LPTR(OLD)) \leftarrow RPTR(OLD)

LPTR(RPTR(OLD)) \leftarrow LPTR(OLD)

3. [Return deleted node]

Restore(OLD)

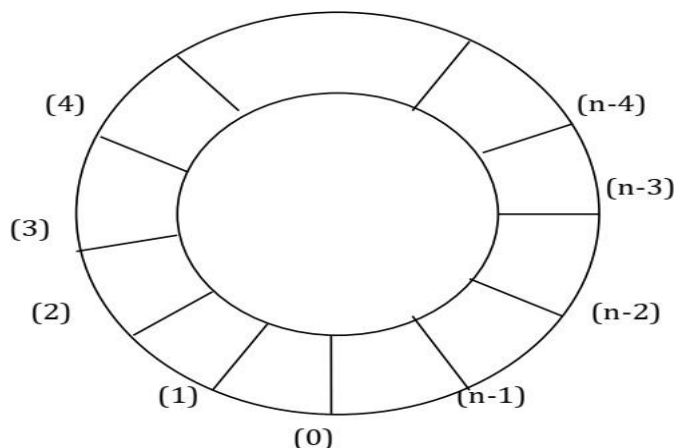
Return

8. **Write an algorithm for inserting an element into a circular Queue and deleting an element from a circular Queue. (Dec2018)**

CIRCULAR QUEUE

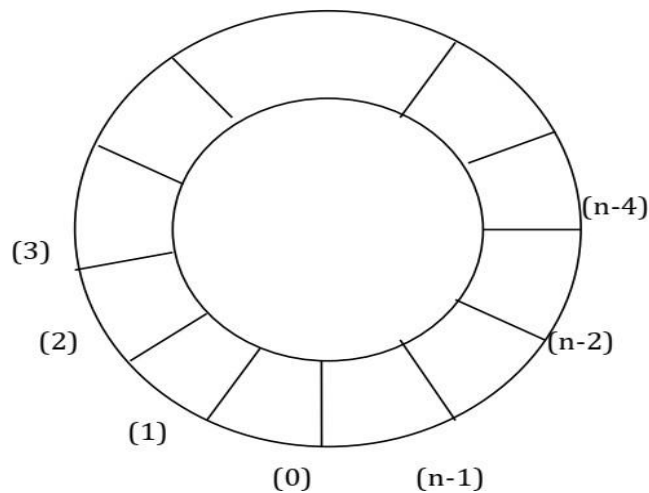
- Initially in linear queue, the front and rear ends are in same position (i.e., -1).
- While inserting elements, the rear pointer moves one by one, until the last index position is reached.
- Beyond this the element cannot be inserted irrespective of the position of the front pointer.
- When deleting the elements, the front pointer moves one by one until the rear point is reached.
- If the front pointer reaches the rear pointer, both their positions are initialized to -1, and the queue is said to be empty.
- This is the main disadvantage of the linear queues, which is overcome in the circular queues.
- **Circular queue** is another form of a linear queue in which the last position is connected to the first position of the list.

- The circular queue is similar to linear queue has two ends, the front end and the rear end.
- The rear end is where we insert the elements and the front end is where we delete the elements.
- The traversing is in only one direction (i.e., from front to rear).
- Initially the front and rear ends are at same position(i.e., -1);
- While inserting elements the rear pointer moves one by one (where front pointer doesn't change) until the front end is reached.
- If the next position of the rear is front, the queue is said to be fully occupied.
- Beyond this insertion is not done.
- But if we delete any data, we can insert the element accordingly.
- While deleting the elements the front pointer moves one by one (where as the rear point doesn't change) until the rear point is reached.
- If the front pointer reaches the rear pointer, both the ir positions are initialized to -1, and the queue is said to be empty.
- A more efficient queue representation is obtained by the circular array.
- It becomes more convenient to declare the array as $Q[n-1]$.
- When $rear = n-1$, the next element is entered at $Q[0]$ in case that position is free.
- Using the same conventions, front will always point one position counterclockwise from the first element in the queue.
- Again, $front=rear$ if and only if the queue is empty. Initially we have $front = rear = 1$.
- The following figure illustrates some of the possible configurations for a circular queue containing the four elements with $n>4$.



Front= 0; Rear =4

- The assumption of circularity changes the ADD and DELETE algorithm slightly.
- In order to add an element, it will be necessary to move *rear* one position clockwise, i.e., if $rear = n-1$ then $rear = 0$ else $rear = rear+1$.
- Using modulo operator which computes remainders, this is just $rear=(rear+1)\bmod n$.
- Similarly, it will be necessary to move front one position clockwise each time a deletion is made.
- Again, using the modulo operation, this can be accomplished by $front=(front+1)\bmod n$.
- An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or $O(1)$.



Front = n-4 ; Rear = 0

Procedure *ADDQ* (*item*, *Q*, *n*, *front*, *rear*)

//insert item in the circular queue stored in Q (0: n-1);

rear points to the last item and *front* is one position counterclockwise from the first item in Q//

rear = (*rear*+1)mod *n* //advance rear clockwise//

if *front* = *rear* then call QUEUE-FULL

Q(*rear*)= *item* //insert new item //

end *ADDQ*.

Procedure *DELETEQ* (*item*, *Q*, *n*, *front*, *rear*)

//removes the front element of the queue Q (0: n-1)//

if *front* = *rear* then call QUEUE-EMPTY

front = (*front* + 1)mod *n* //advance front clockwise//

item = *Q*(*front*) //set item to front of queue//

end *DELETEQ*

9. Write an algorithm that takes the first node in a linked list, reverse it and return the first node in the resulting linked list without recursion and with recursion. (Dec 2017)

Given single linked list containing set of data. From this (Dec2019)

(a) Reverse the direction of links.

(b) For the list count the number of nodes.

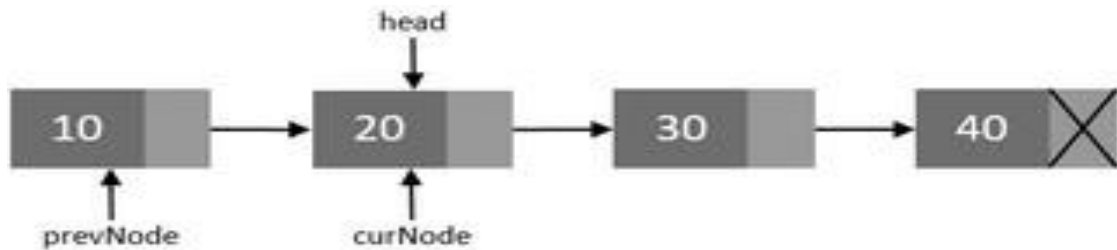
Steps to reverse a Singly Linked List

1. Create two more pointers other than head namely prevNode and curNode that will hold the reference of previous node and current node respectively.

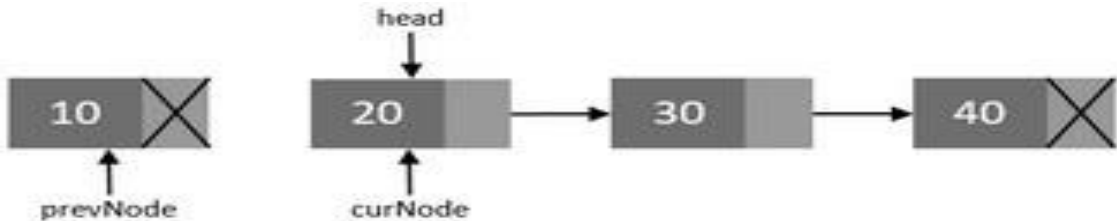
Make sure that prevNode points to first node i.e. prevNode = head.

head should now point to its next node i.e. the second node head = head->next.

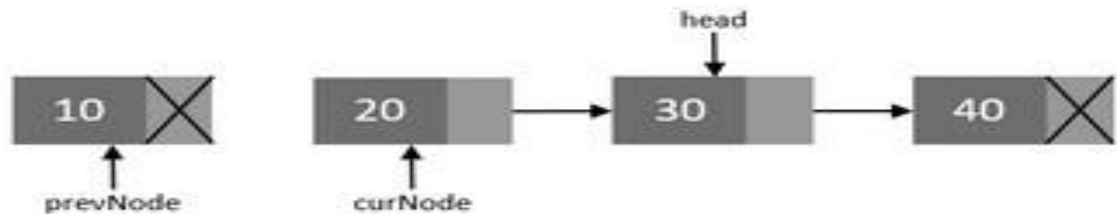
CurNode should also points to the second node i.e. curNode = head.



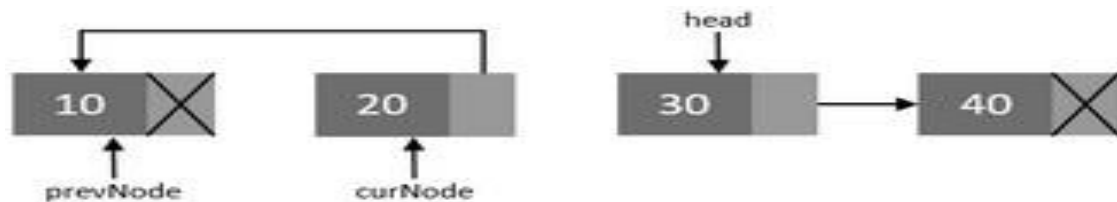
2. Now, disconnect the previous node i.e. the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation $\text{prevNode} \rightarrow \text{next} = \text{NULL}$.



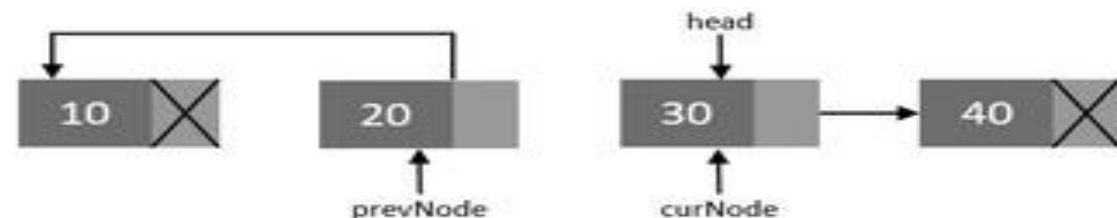
3. Move head node to its next node i.e. $\text{head} = \text{head} \rightarrow \text{next}$.



4. Now, re-connect the current node to its previous node i.e. $\text{curNode} \rightarrow \text{next} = \text{prevNode}$



5. Point the previous node to current node and current node to head node. Means they should now point to $\text{prevNode} = \text{curNode}$; and $\text{curNode} = \text{head}$.



6. Repeat steps 3-5 till head pointer becomes NULL

For the list count the number of nodes:

Algorithm to count number of nodes in Singly Linked List

%%Input : *head* node of the linked list

Begin:

$count \leftarrow 0$

If (*head* != NULL) then

$temp \leftarrow head$

While (*temp* != NULL) do

$count \leftarrow count + 1$

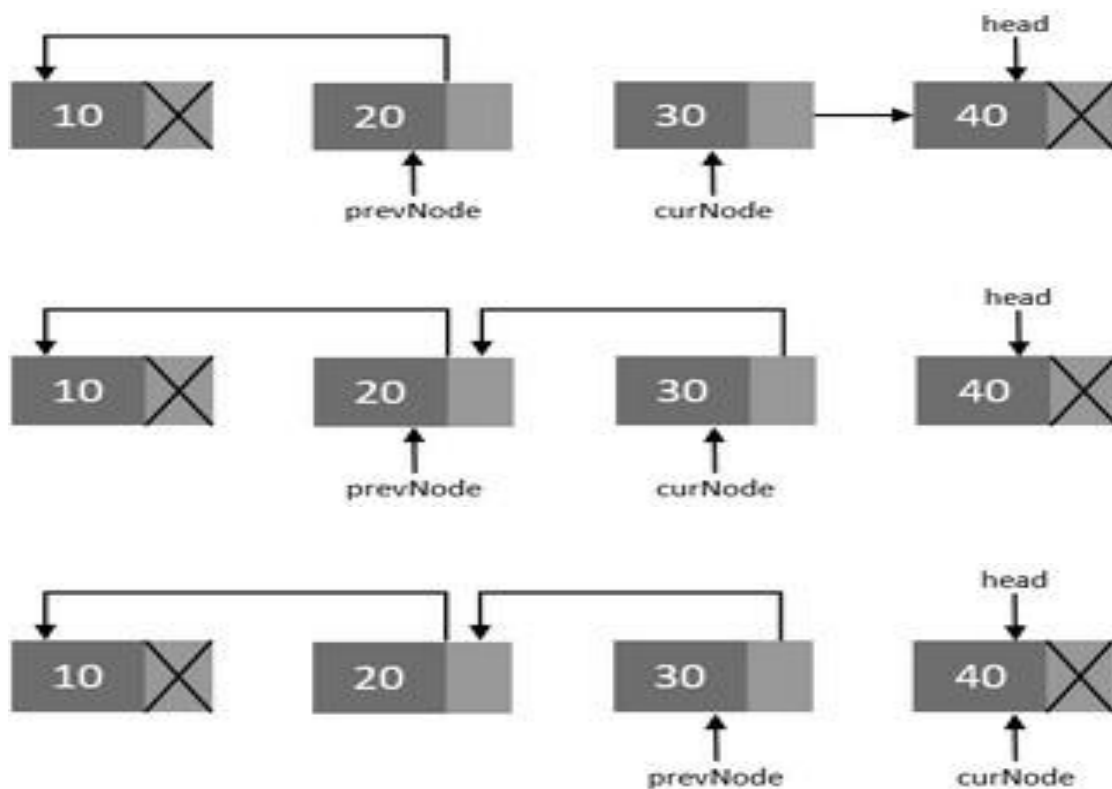
$temp \leftarrow temp.next$

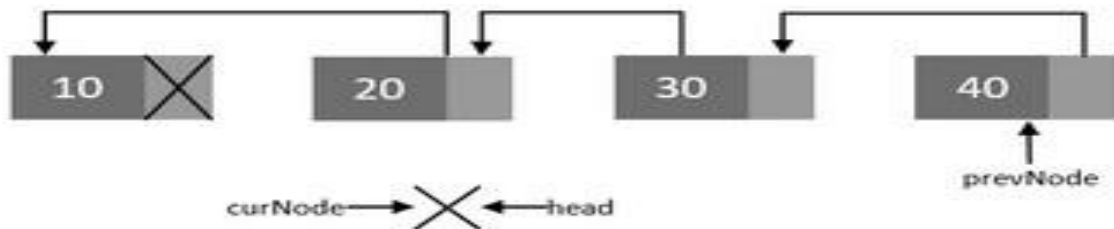
End while

End if

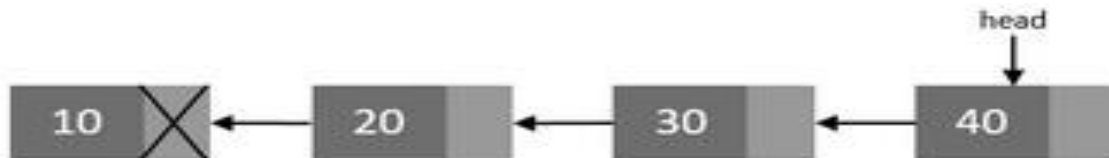
write ('Total nodes in list =' + *count*)

End





7. Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the head pointer should point to prevNode pointer. Perform head = prevNode;. Finally you end up with a reversed linked list of its original.



Develop procedures to perform push and pop operations of stack in an array and evaluate postfix expression $1 + 2 \ 3 \ -4 * 5 /$ using it. (May2018)

10. **Develop procedures to perform push and pop operations of stack in an array and evaluate postfix expression $1 + 2 \ 3 \ -4 * 5 /$ using it. (May2018)**

PUSH OPERATION

Stack can be implemented using arrays and pointers.

Array Implementation

In this implementation each stack is associated with a pop pointer, which is -1 for an empty stack

- To push an element X onto the stack, Top Pointer is incremented and then set Stack [Top] = X
- To pop an element, the stack [Top] value is returned and the top pointer is decremented.
- pop on an empty stack or push on a full stack will exceed the array bounds.

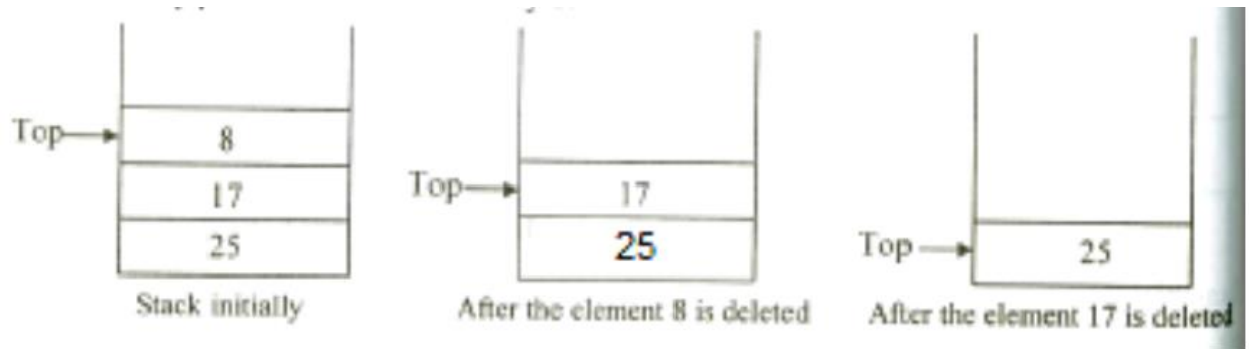
ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int x, Stack S)
{
    if(IsFull (S))
        Error ("Full Stack");
    else
    {
        Top = Top + 1;
        S[Top] = X;
    }
}

int IsFull (Stack S)
{
    if (Top == Arraysize)
        return (1);
}
```

POP

The process of deleting an element from the top of stack is called pop operation. After every pop operation the top pointer is decremented by 1.



POP OPERATION

ROUTINE TO POP AN ELEMENT FROM THE STACK

```
void pop (Stack S)
{
    if (IsEmpty (S))
        Error ("Empty Stack");
    else
    {
        X = S [Top];
        Top = Top - 1;
    }
}
```

EXCEPTIONAL CONDITIONS

Over Flow: - Attempt to insert an element when the stack is full is said to be overflow.

Under Flow: - Attempt to delete an element, when the stack is empty is said to be underflow.

11. Write a C program to add two polynomials. (Dec 2017)

```

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 = ptr1 -> 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;
            }
        }
    }
}

```

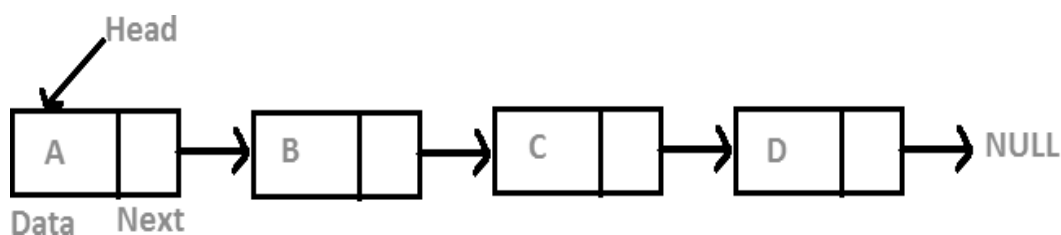
```

else
{
    newnode → coeff = ptr2 → coeff;
    newnode → power = ptr2 → power;
    newnode → next = NULL;
    list3 = create (list3, newnode);
    ptr2 = ptr2 → next;
}
}
}

```

12. Explain the linked list and stack implementation of stack ADT, write program to implement Stack operations? (Dec 2014)

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

The Stack ADT stores arbitrary objects and insertions and deletions of this ADT follow the last-in-first-out (LIFO) scheme like a spring-loaded plate dispenser.

Main stack operations:

- *void push(object)*: inserts an element
- *object pop()*: removes and returns the last inserted element

Auxiliary stack operations:

- *object top()*: returns the last inserted element without removing it
- *integer size()*: returns the number of elements stored
- *boolean isEmpty()*: indicates whether no elements are stored

Array-based stack

This is a simple way of implementing the Stack ADT using an array. Here, elements are added from left to right and a variable keeps track of the index of the top element.

PROGRAM TO IMPLEMENT STACK OPERATION

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

```
int stack[10],choice,n,top,x,i; // Declaration of variables
```

```
void push();
```

```
void pop();
```

```
void display();
```

```
int main()
```

```
{
```

```
    top = -1; // Initially there is no element in stack
```

```
    printf("\n Enter the size of STACK : ");
```

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

```
    printf("\nSTACK IMPLEMENTATION USING ARRAYS\n");
```

```
    do
```

```
{
```

```
    printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n");
```

```
    printf("\nEnter the choice : ");
```

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

```
    switch(choice)
```

```
{
```

```
case 1:
```

```
{
```

```
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
break;
}
default:
{
printf ("\nInvalid Choice\n");
}}}
while(choice!=4);
return 0;
}

void push()
{
if(top >= n - 1)
{
printf("\nSTACK OVERFLOW\n");
}
else
{
```

```
printf("Enter a value to be pushed : ");
scanf("%d",&x);
top++;      // TOP is incremented after an element is pushed
stack[top] = x; // The pushed element is made as TOP
}}
```

```
void pop()
{
    if(top <= -1)
    {
        printf("\nSTACK UNDERFLOW\n");
    }
    else
    {
        printf("\nThe popped element is %d",stack[top]);
        top--; // Decrement TOP after a pop
    }
}
```

```
void display()
{
    if(top >= 0)
    {
        // Print the stack
        printf("\nELEMENTS IN THE STACK\n\n");
        for(i = top ; i >= 0 ; i--)
            printf("%d\t",stack[i]);
    }
    else
    {
        printf("\nEMPTY STACK\n");
    }
}
```

13. What are the various operations that can be performed on stacks and queues? Explain the algorithm for each. (Dec 2014)

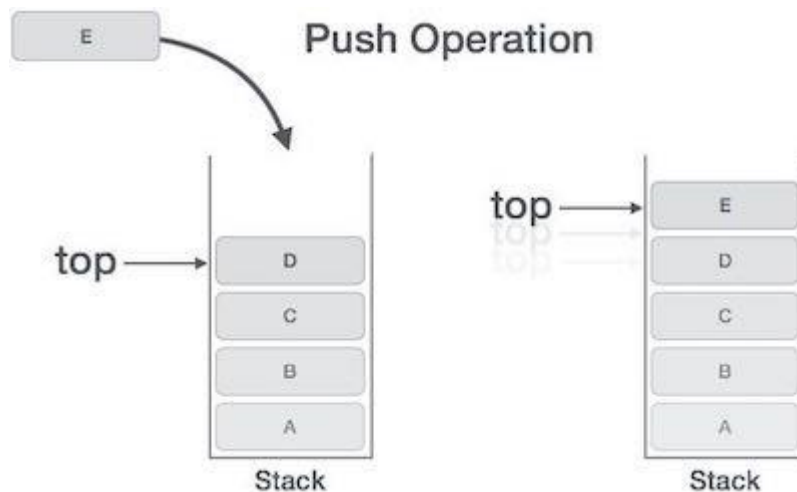
A stack is an abstract data type that serves as a collection of elements, with two main principal operations:

- Push, which adds an element to the collection, and
- Pop, which removes the most recently added element that was not yet removed.

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.
- Step 5 – Returns success.



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

Algorithm for PUSH Operation

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

```
begin procedure push: stack, data
```

```
    if stack is full
    return null
    endif
```

```
    top ← top + 1
    stack[top] ← data
```

```
end procedure
```


Implementation of this algorithm in C, is very easy. See the following code –

Example

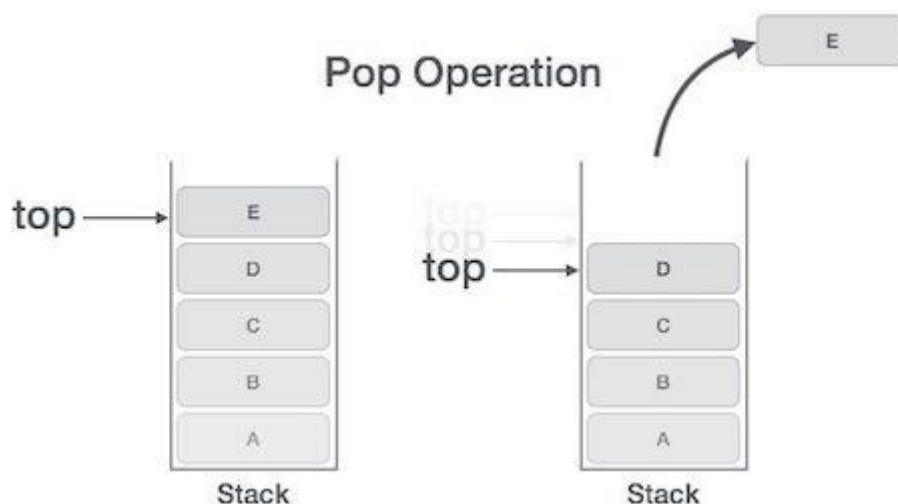
```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

Pop Operation

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

A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
```

```
    if stack is empty
    return null
endif
```

```
    data ← stack[top]
    top ← top - 1
    return data
```

```
end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data) {
    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
```

Operations on Queue:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```

begin procedure peek
    return queue[front]
end procedure

```

Implementation of peek() function in C programming language –

Example

```

int peek() {
    return queue[front];
}

```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```

begin procedure isfull

    if rear equals to MAXSIZE
return true
    else
return false
endif

end procedure

```

Implementation of isfull() function in C programming language –

Example

```

bool isfull() {
    if(rear == MAXSIZE - 1)
return true;
    else
return false;
}

```

isempty()

Algorithm of isempty() function –

Algorithm

```

begin procedure isempty

    if front is less than MIN OR front is greater than rear
return true
    else
return false
endif

end procedure

```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

```

bool isempty() {
    if(front < 0 || front > rear)
        return true;
    else
        return false;
}

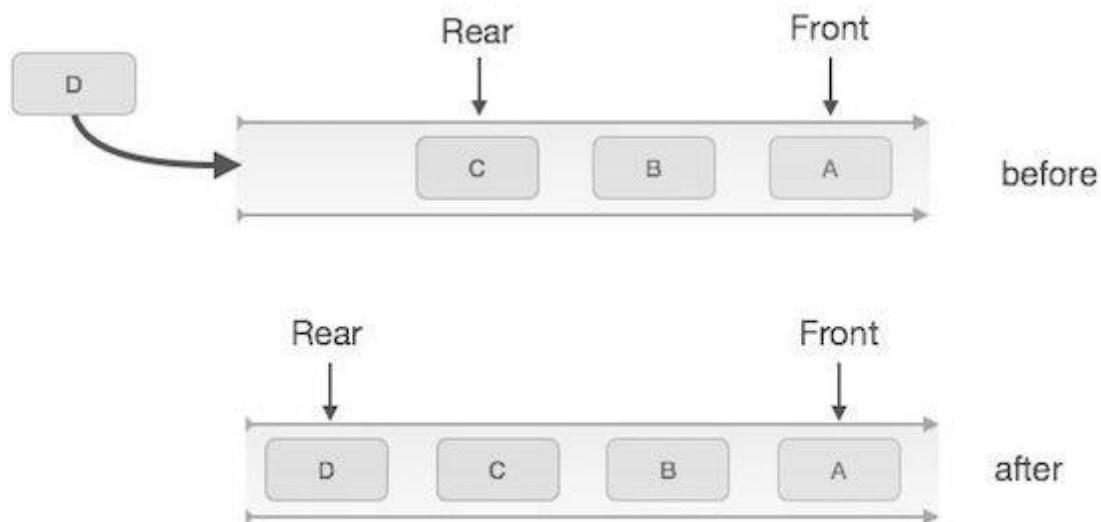
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.

**Queue Enqueue**

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```

Procedure enqueue(data)

    if queue is full
        return overflow
    endif

    rear ← rear + 1
    queue[rear] ← data
    return true

end procedure

```

Implementation of enqueue() in C programming language –

Example

```

int enqueue(int data)
if(isfull())
    return 0;

rear = rear + 1;
queue[rear] = data;

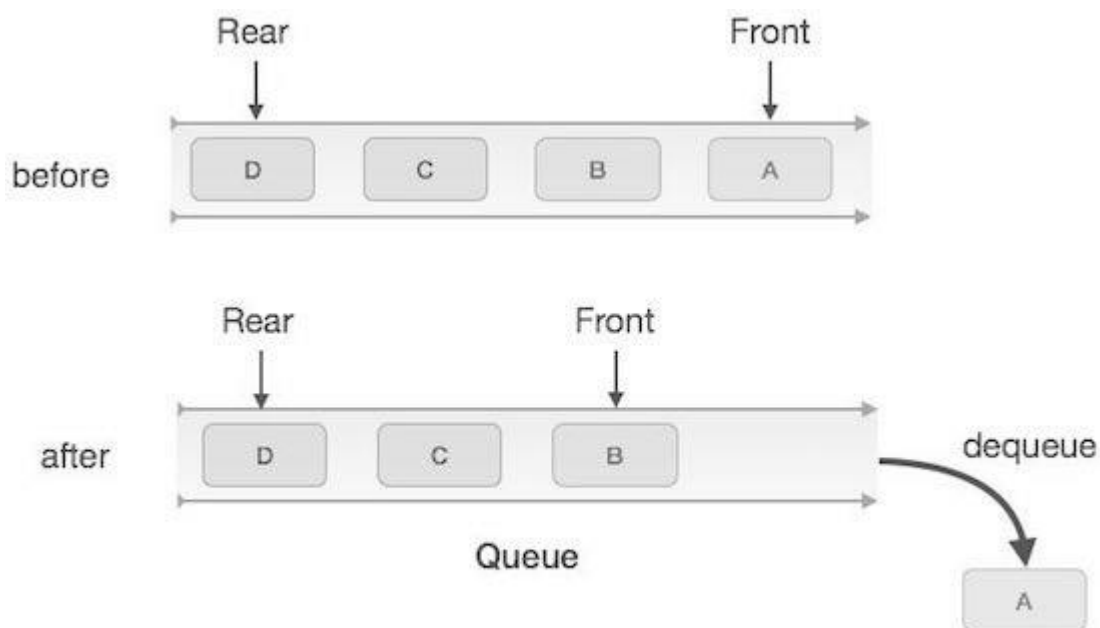
return 1;
end procedure

```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.

**Queue Dequeue****Algorithm for dequeue operation**

```

procedure dequeue

if queue is empty
    return underflow
end if

data = queue[front]

```

```
front ← front + 1  
return true  
  
end procedure
```

Implementation of dequeue() in C programming language –

Example

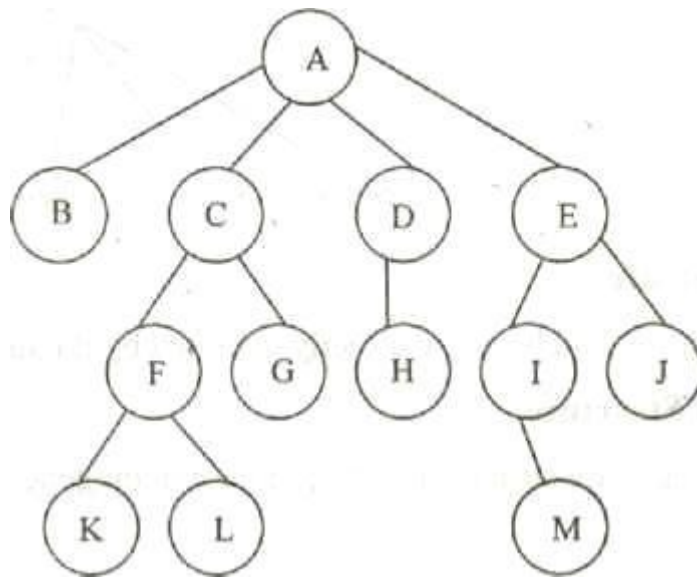
```
int dequeue() {  
    if(isempty())  
        return 0;  
  
    int data = queue[front];  
    front = front + 1;  
  
    return data;  
}
```

UNIT III : DYNAMIC STORAGE MANAGEMENT

Trees: Binary tree, Terminology, Representation, Traversals, Applications. **Graph:** Terminology, Representation, Traversals – Applications - spanning trees, shortest path. Introduction to **Hash tables**.

TREE :

A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T_1, T_2, \dots, T_k , each of whose roots are connected by a directed edge from Root R.



ROOT A node which doesn't have a parent. In the above tree. The Root is A.

NODE Item of Information.

LEAF A node which doesn't have children is called leaf or Terminal node.

Here B, K, L, G, H, M, J are leafs.

SIBLINGS Children of the same parents are said to be siblings,

Here B, C, D, E are siblings, F, G are siblings.

Similarly I, J & K, L are siblings.

PATH A path from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, n_3, \dots, n_k$ such that

n_i is the parent of n_{i+1} . for $1 < i < k$ There is exactly only one path from each node to root.

In fig 3.1.1 path from A to L is A, C, F, L. where A is the parent for C, C is the, parent of F and F is the parent of L.

LENGTH : The length is defined as the number of edges on the path. In fig the length for the path A to L is 3.

DEGREE

The number of subtrees of a node is called its degree.

Degree of A is 4

Degree of C is 2

Degree of D is 1

Degree of H is 0.

The degree of the tree is the maximum degree of any node in the tree.

In fig the degree of the tree is 4.

LEVEL

The level of a node is defined by initially letting the root be at level one, if a node is at level L then its children are at level L + 1.

Level of A is 1.

Level of B, C, D, is 2.

Level of F, G, H, I, J is 3

Level of K, L, M is 4.

DEPTH

For any node n, the depth of n is the length of the unique path from root to n. The depth of the root is zero.

In fig 3.1.1 Depth of node F is 2.

Depth of node L is 3.

HEIGHT

For any node n, the height of the node n is the length of the longest path from n to the leaf.

The height of the leaf is zero

In fig 3.1.1 Height of node F is 1.

Height of L is 0.

BINARY TREE

Binary Tree is a tree in which no node can have more than two children. Maximum number of nodes at Level i of the binary tree is 2^i

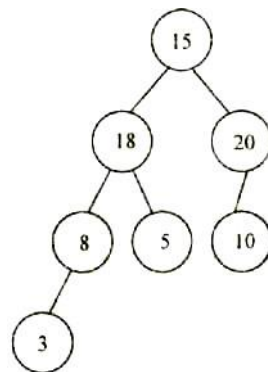


Fig. 3.2.1 Binary Tree

At level 0 $2^0 = 1$ maximum of 1 node at zeroth level

At level 3 $2^3 = 8$ maximum of 8 node at level three

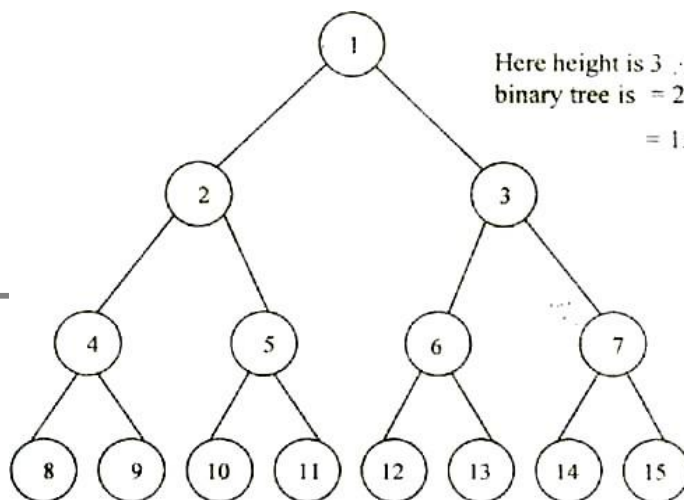


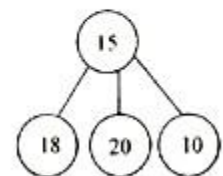
Fig. 3.2.2 A Full Binary Tree

Here height is 3 \therefore No. of nodes in full binary tree is $= 2^{h+1} - 1$
 $= 15$ nodes.

GENERAL TREE

General Tree

* General Tree has any number of children.



FULL BINARY TREE :- A full binary tree of height h has $2^{h+1} - 1$

nodes.

COMPLETE BINARY TREE :

A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. In the bottom level the elements should be filled from left to right.

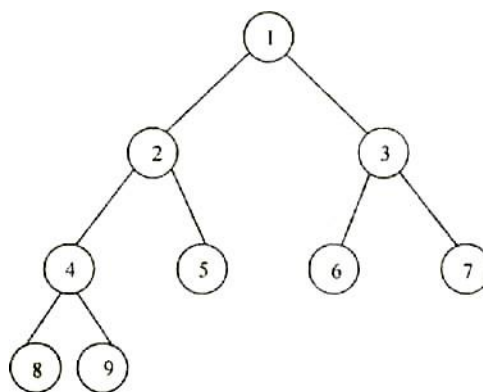


Fig. 3.2.3 A Complete Binary Tree.

Note : A full binary tree can be a complete binary tree, but all complete binary tree is not a full binary tree.

BINARY TREE REPRESENTATION:

LINEAR REPRESENTATION OF A BINARY TREE

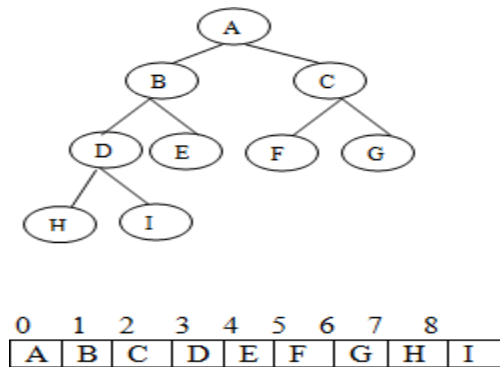
The linear representation method of implementing a binary tree uses a one-dimensional array of size $((2^{d+1})-1)$ where d is the depth of the tree.

Once the size of the array has been determined the following method is used to represent the tree.

1. Store the root in 1st location of the array.
2. If a node is in location n of the array store its left child at location $2n$ and its right

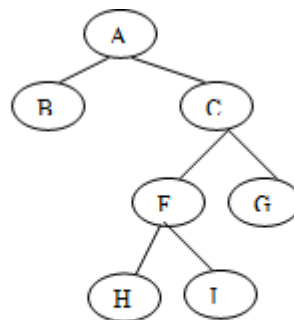
child at $(2n+1)$.

In c, arrays start at position 0; therefore instead of numbering the trees nodes from 1 to n, we number them from 0 to n-1. The two child of a node at position P are in positions $2P+1$ and $2P+2$. The following figure illustrate arrays that represent the almost complete binary trees.

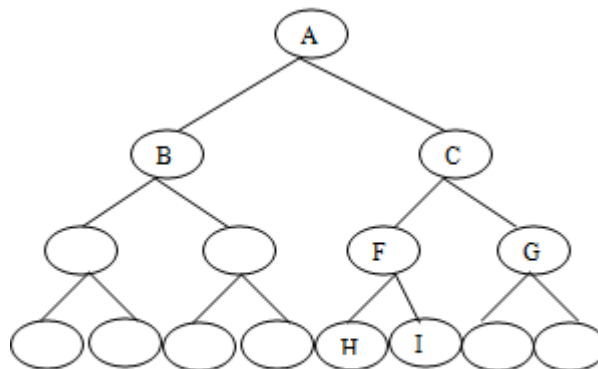


We can extend this array representation of almost complete binary trees to an array representation of binary trees generally.

The following fig(A) illustrates binary tree and fig(B) illustrates the almost complete binary tree of fig(a). Finally fig(C) illustrates the array implementation of the almost complete binary tree.



Fig(A) Binary tree



Fig(B) Almost a complete binary tree

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			F	G					H	I

Fig (C) Array Representation

ADVANTAGES:

1. Given a child node , its parent node can be determined immediately. If a child node is at location N in the array, then its parent is at location $N/2$.
2. It can be implemented easily in languages in which only static memory allocation is directly available.

DISADVANTAGES:

1. Insertion or deletion of a node causes considerable data movement up and down the array, using an excessive amount of processing time.
2. Wastage of memory due to partially filled trees.

LINKED LIST REPRESENTATION:

Linked lists most commonly represent binary trees. Each node can be considered as having 3 elementary fields : a data field, left pointer, pointing to left sub-tree and right pointer pointing to the right sub-tree.

The following figure is an example of linked storage representation of a binary tree.

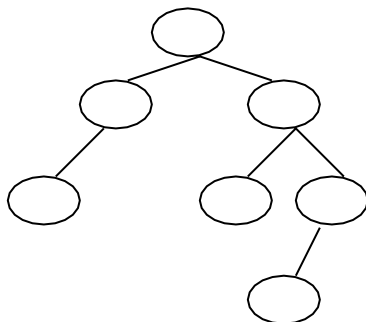


FIG: Binary tree

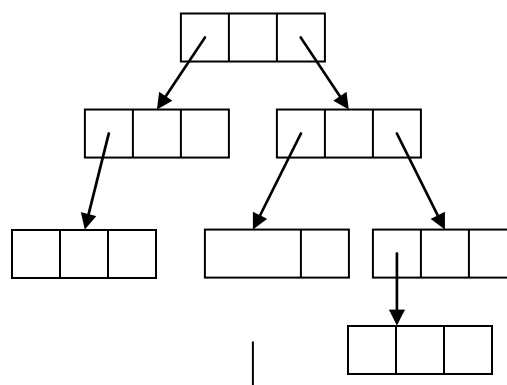


FIG: Linked representation of a binary tree

Although for most purposes the linked representation of a binary tree is efficient, it does have certain disadvantages. Namely,

1. Wasted memory space is well pointers.

2. Given a node, it is difficult to determine its parent.

3. Its implementation algorithm is more difficult in languages that do not offer dynamic storage techniques.

BINARY TREE TRAVERSAL

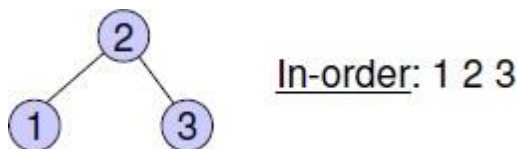
In a traversal of a binary tree, each element of the binary tree is visited exactly once. During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

IN-ORDER

This can be summed up as

- Traverse to left sub tree
- Visit root node (generally output this)
- Traverse to right sub tree

Example 1 :



Example 2:

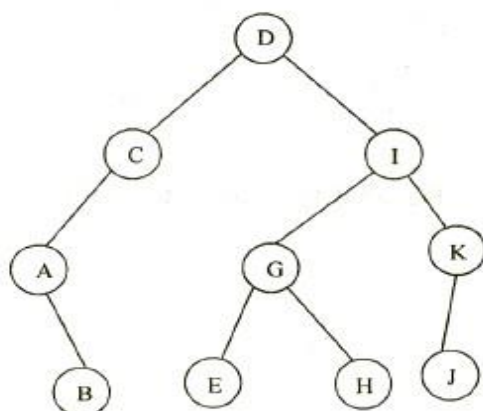


Fig. 3.5.2 A B C D E G H I J K

RECURSIVE ROUTINE FOR INORDER TRAVERSAL

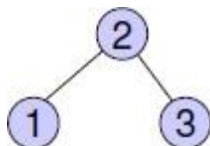
```
void Inorder (Tree T)
{
    if (T != NULL)
    {
        Inorder (T → left);
        printElement (T → Element);
        Inorder (T → right);
    }
}
```

POST-ORDER

This can be summed up as

- Traverse to left sub tree
- Traverse to right sub tree
- Visit root node (generally output this)

Example : 1



Post-order: 1 3 2

Example : 2

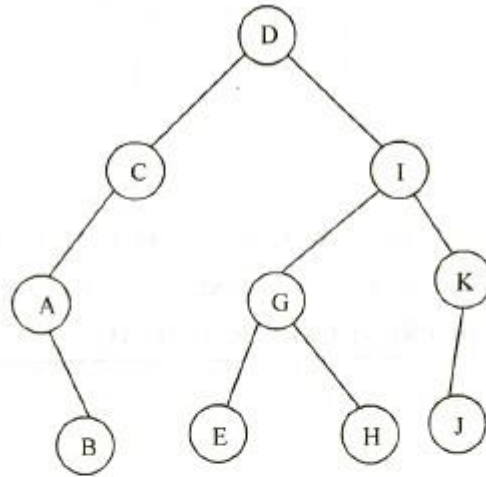


Fig. 3.5.6 Post order : - B A C E H G J K I D

The postorder traversal of the binary tree for the given expression gives in postfix form.

RECURSIVE ROUTINE FOR POSTORDER TRAVERSAL

```

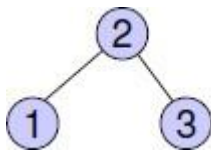
void Postorder (Tree T)
{
    if (T != NULL)
    {
        Postorder (T → Left);
        Postorder (T → Right);
        PrintElement (T → Element);
    }
}
  
```

PRE-ORDER

This can be summed up as

- Visit the root node (generally output this)
- Traverse to left subtree
- Traverse to right subtree

Example :1



Pre-order: 2 1 3

Example : 2

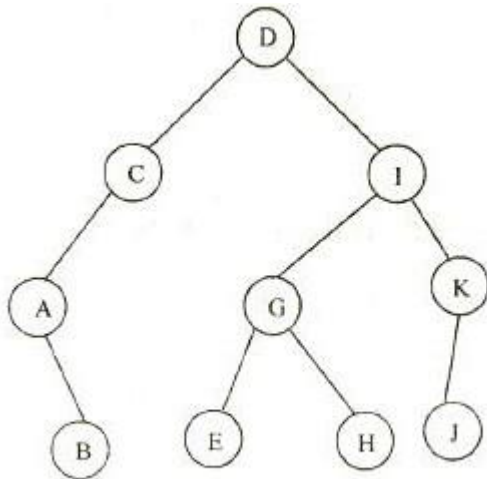


Fig. 3.5.4 Preorder D C A B I G E H K J

RECURSIVE ROUTINE FOR PREORDER TRAVERSAL

```

void Preorder (Tree T)
{
    if (T != NULL)
    {
        printElement (T → Element);
        Preorder (T → left);
        Preorder (T → right);
    }
}
  
```

APPLICATIONS OF BINARY TREE :-

Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.

Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.

Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.

Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* (*path-finding algorithm used in AI applications, including robotics and video games*). Also used in heap-sort.

Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.

Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.

T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

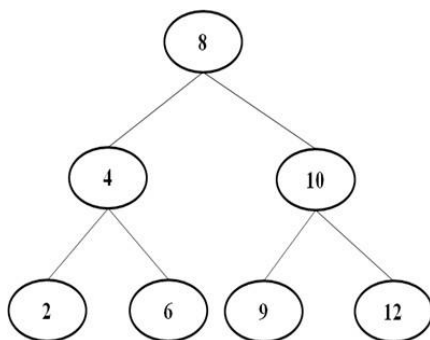
APPLICATION 1 : BINARY SEARCH TREE :-

Binary search tree is a binary tree in which for every node X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

Note : * Every binary search tree is a binary tree.

* All binary trees need not be a binary search tree.

BINARY SEARCH TREE – EXAMPLE



Note: left value < parent < right value

Algorithm

TREE-SEARCH(x, k)

```

. if  $x = \text{NIL}$  or  $k = \text{key}[x]$ 

then return  $x$ 

if  $k < \text{key}[x]$ 

then return TREE-SEARCH( $\text{left}[x], k$ )

else return TREE-SEARCH( $\text{right}[x], k$ )

```

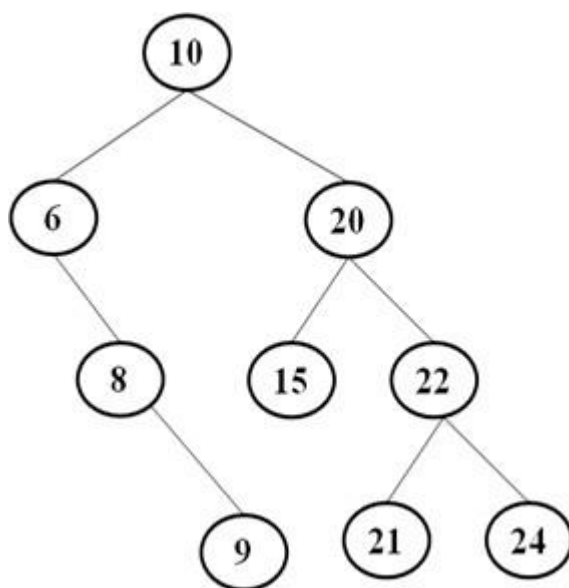
END TREE-SEARCH

OPERATIONS ON BINARY SEARCH TREE

The *basic operations* on which can be performed on *binary search tree* are:

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

INSERTION OF A NODE IN BINARY SEARCH TREE

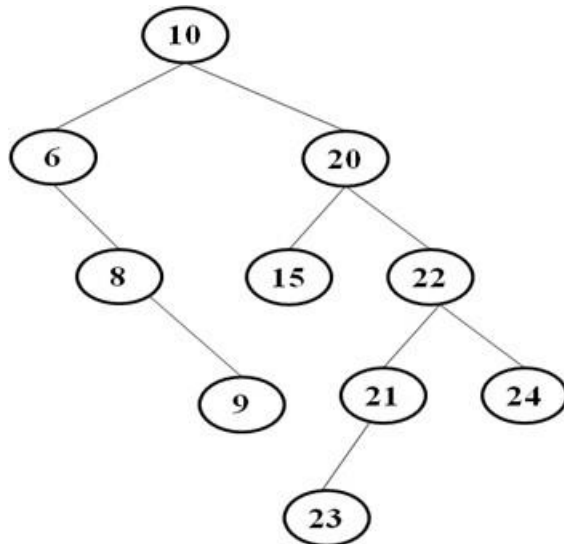


To insert 23 in the figure.

- Start comparing 23 with the root i.e 10.
- As $23 > 10$, move right.

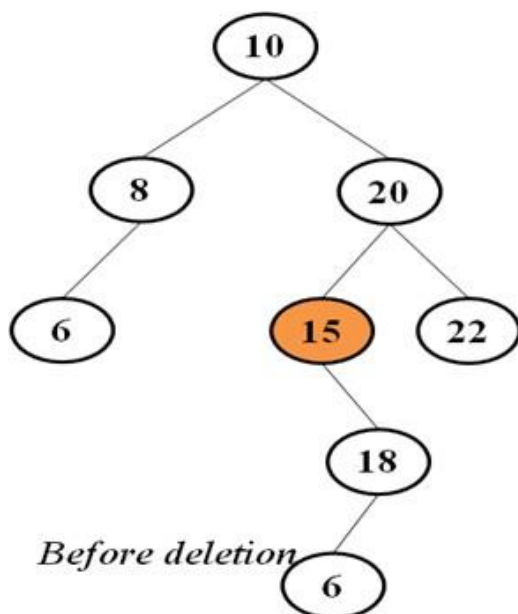
- Then compare 23 with 20, again $23 > 20$, so move right.
- Then compare 23 with 24, $23 < 24$ so insert 23 as the left child of 24.

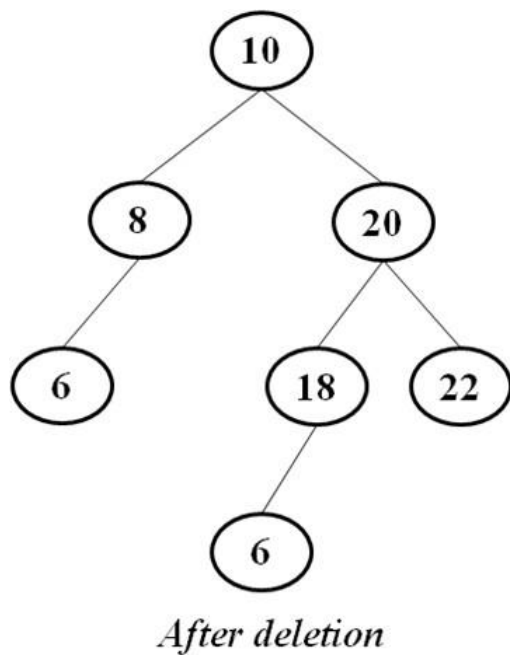
AFTER INSERTION



DELETION OF A NODE IN BINARY SEARCH TREE

If we want to delete the node 15, then we will simply copy node 18 at place of 15 and then set the node free.





APPLICATION 2 : HEAPSORT :-

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

Start deleteMin operations, storing each deleted element at the end of the heap array.

After performing sorting, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heaptree in descending order - the greatest element will have the highest priority.

Algorithm :-

HEAPIFY(A, i)

$L \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $L \leq \text{heap-size}[A]$ and $A[L] > A[i]$

then largest $\leftarrow L$

else largest $\leftarrow r$

if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

then largest $\leftarrow r$

if largest $\neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY(A, largest)

END HEAPIFY

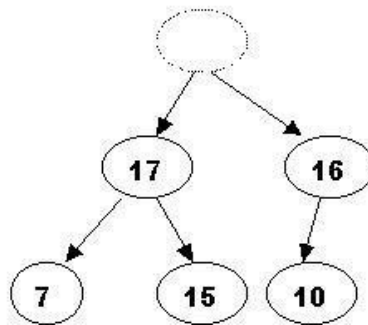
SORTING - PERFORMING DELETETEMAX OPERATIONS:

1. Delete the top element 19.

Store 19 in a temporary place. A hole is created at the top



19



Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array

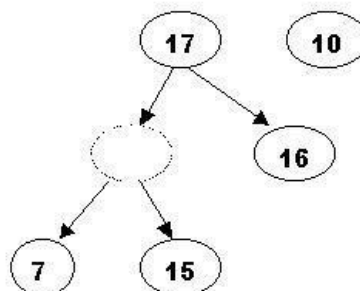


10

Percolate down the hole



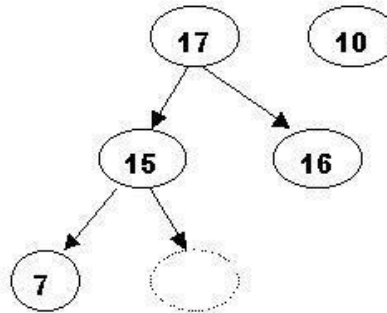
10



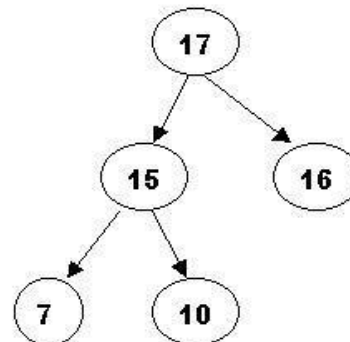
Percolate once more (10 is less than 15, so it cannot be inserted in the previous hole)



10



Now 10 can be inserted in the hole

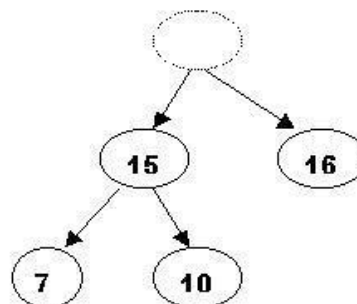


2. DeleteMax the top element 17

Store 17 in a temporary place. A hole is created at the top

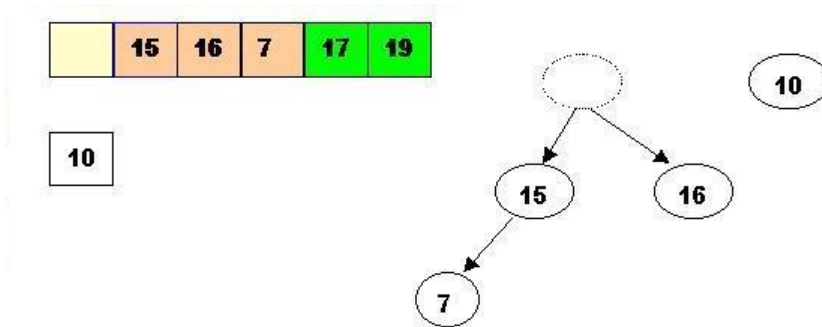


17

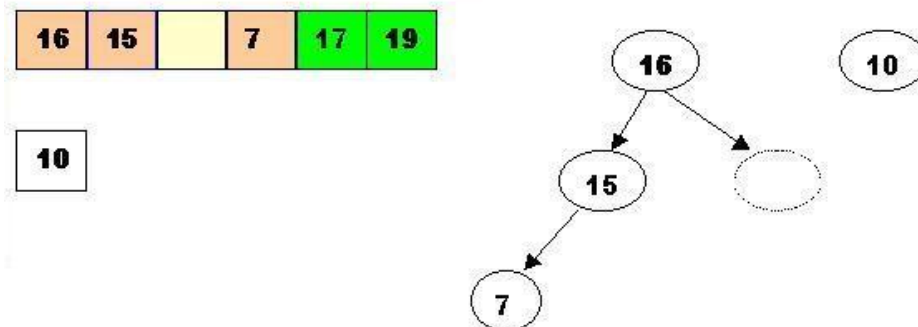


Swap 17 with the last element of the heap.

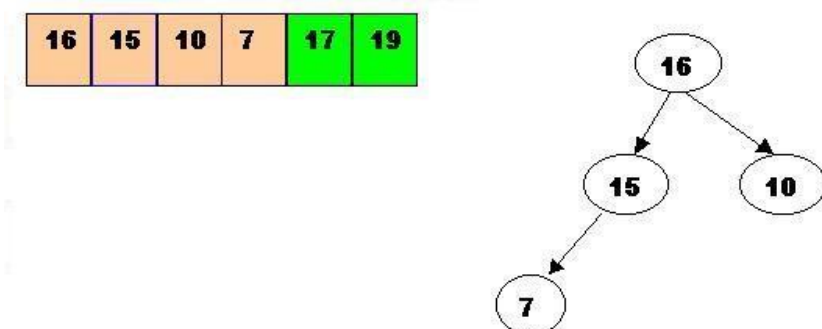
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



The element 10 is less than the children of the hole, and we percolate the hole down:



Insert 10 in the hole



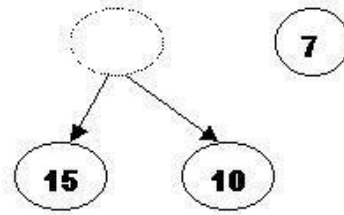
3. DeleteMax 16

Store 16 in a temporary place. A hole is created at the top

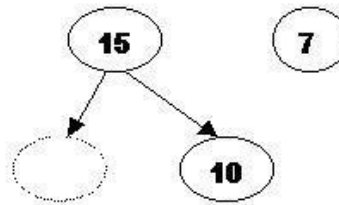


Swap 16 with the last element of the heap.

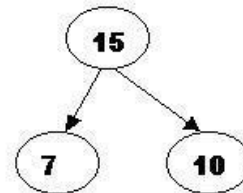
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)

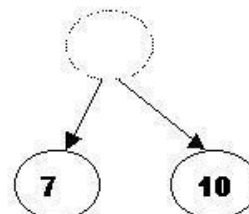


Insert 7 in the hole



4. DeleteMax the top element 15

Store 15 in a temporary location. A hole is created.



Swap 15 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a position from the sorted array



Store 10 in the hole (10 is greater than the children of the hole)



5. DeleteMax the top element 10.

Remove 10 from the heap and store it into a temporary location.

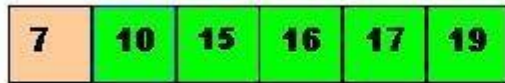


Swap 10 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



Store 7 in the hole (as the only remaining element in the heap)



7 is the last element from the heap, so now the array is sorted

GRAPH

DEFINING GRAPH:

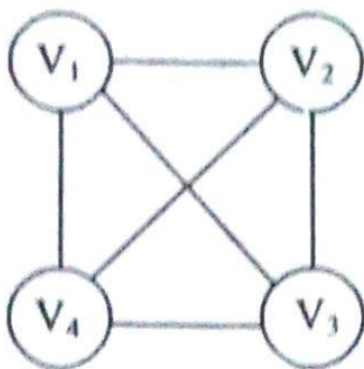
A graph G consists of a set V of vertices (nodes) and a set E of edges (arcs). We write $G=(V,E)$. V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges. Therefore,

$V(G)$, read as V of G , is a set of vertices and $E(G)$, read as E of G is a set of edges.

An edge $e=(v, w)$ is a pair of vertices v and w , and to be incident with v and w .

A graph $G = \{V, E\}$ consists of a set of vertices V and set of edges E .

Vertices are referred to as nodes in graph and the line joining the two vertices are referred to as Edges.



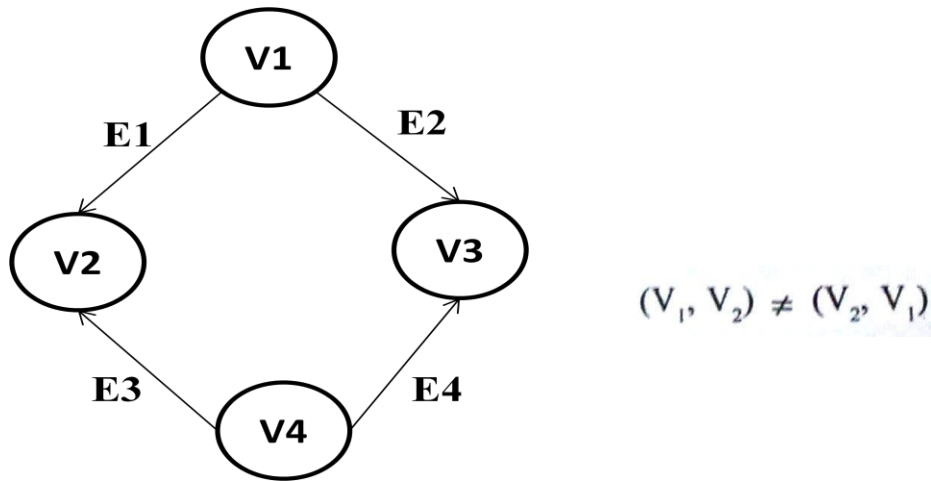
Types of Graphs

Graphs are of two types:

- Directed graphs.
- Undirected Graphs.

DIRECTED GRAPHS

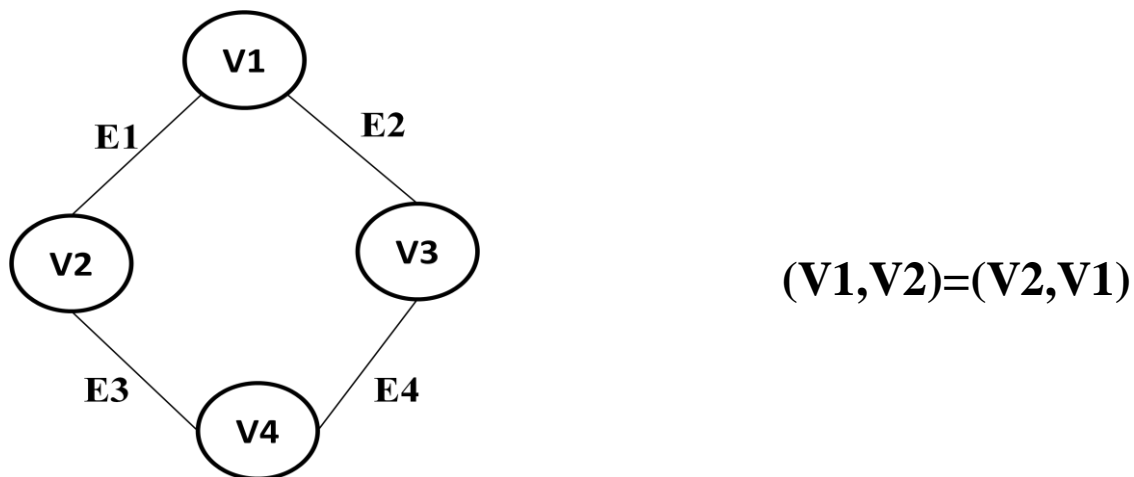
Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then $(v, w) \neq (w, v)$



- In Directed graph, the edges between the vertices are ordered.
- E1 is the edge between the vertices V1 and V2.
- V1 is called the Head and V2 is called the Tail. So, E1 is a set of $(V1, V2)$ and not of $(V2, V1)$.

UNDIRECTED GRAPHS:

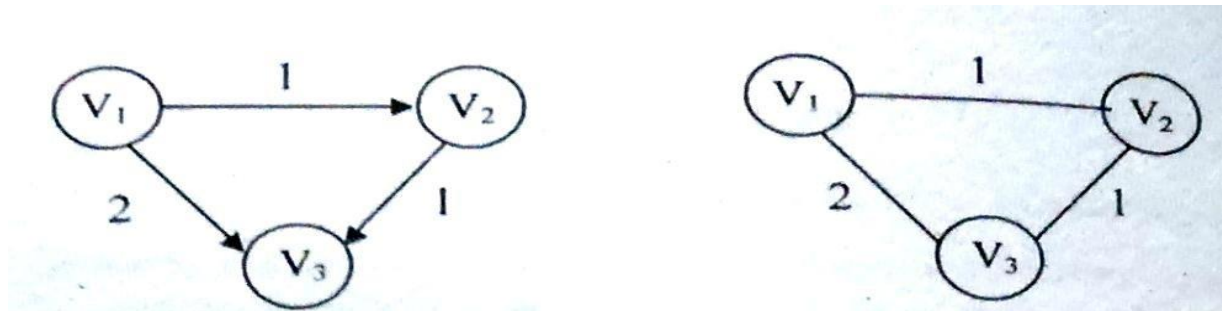
Undirected graph is a graph, which consists of *undirected edges*.



In Undirected graph, the edges between the vertices are not ordered. So, E1 is a set of $(V1, V2)$ or $(V2, V1)$.

WEIGHTED GRAPH:

A Graph is said to be weighted graph if every edge in the graph is assigned a weight or value .It can be directed or undirected graph.

**COMPLETE GRAPH:**

A complete graph is a graph in which there is an edge between every pair of vertices. A complete graph with n vertices will have $\frac{n(n-1)}{2}$ edges.



In above diagram,

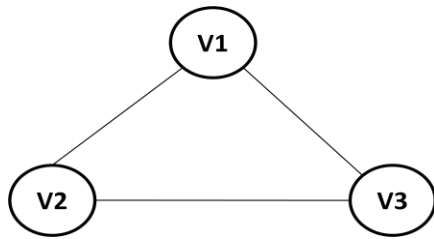
Number of vertices is 4

Number of edges is 6

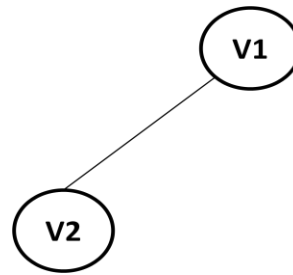
(i.e) There is a path from every vertex to every other vertex. A complete digraph is a strongly connected graph.

SUB GRAPH:

A sub graph G' of G is a graph G such that the set of vertices and set of edges of G' are proper subset of the set of edges of G .



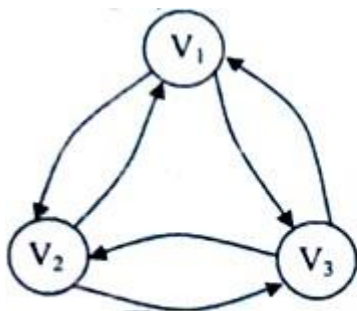
G



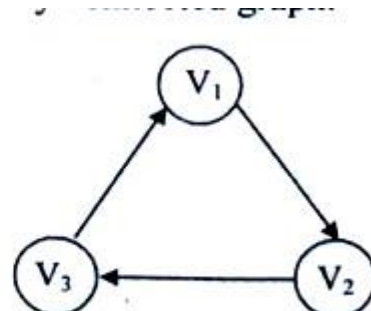
G''

STRONGLY CONNECTED GRAPH :-

If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise it is said to be weakly connected graph.



Strongly Connected Graph



Weakly Connected Graph

PATH :

A Path in a graph is a sequence of vertices $m_1, m_2, m_3, \dots, m_n, m_{n+1} \in E$ for $1 \leq i \leq N$

LENGTH :

- The length of the path is the number of edges on the path, which is equal to $N-1$ where N represents the number of vertices.
- The length of the above path V_1 to V_3 is 2 (i.e) $(V_1, V_2), (V_2, V_3)$
- If there is a path from a vertex to itself, with no edges, then path length is 0.

LOOP :

If the graph contains an edge (v, v) from vertex to itself, then the path is referred to as a loop

SIMPLE PATH :

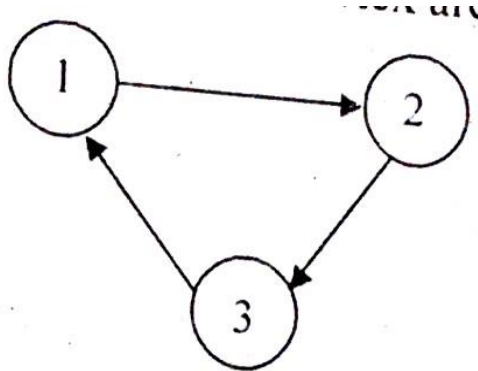
A Simple path is a path such that all vertices on the path, except possibly the first and last are distinct.

A simple cycle is the simple path of length atleast one that begins and ends at the

same vertex.

CYCLE :

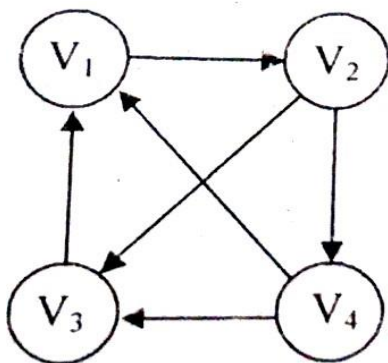
A cycle in a graph is a path in which first and last vertex are same



A graph which has cycles is referred to as cyclic graph

DEGREE:

- The number of edges incident on a vertex determines its degree.
- The indegree of the vertex V is the number of edges entering into the vertex V.
- The Outdegree of the vertex V is the number of edges leaving or exiting from that vertex V.



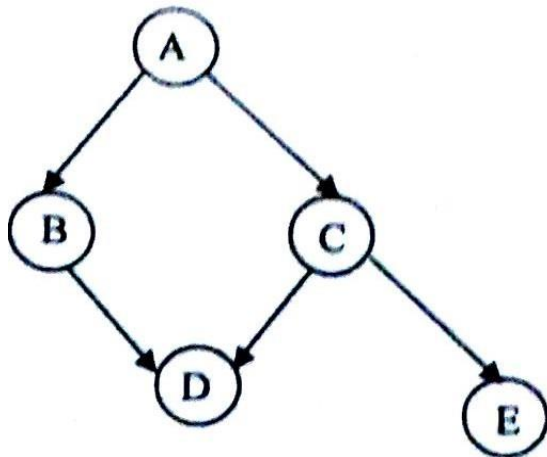
In above fig,

Indegree of V1 is 2	Outdegree of V1 is 1
Indegree of V2 is 1	Outdegree of V2 is 2
Indegree of V3 is 2	Outdegree of V3 is 1

Indegree of V4 is 1	Outdegree of V4 is 2
---------------------	----------------------

ACYCLIC GRAPH :

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG (Directed Acyclic Graph)



REPRESENTATIONS OF GRAPHS:-

A graph can be represented commonly by two ways:

- i) Adjacency matrix representation
- ii) Adjacency list representation.

ADJACENCY MATRIX REPRESENTATION

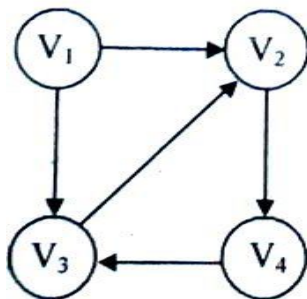
One simple way to represent a graph is Adjacency Matrix

The Adjacency matrix A for a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix, such that

$$A_{ij} = 1, \text{ if there is an edge } V_i \text{ to } V_j$$

$$A_{ij} = 0, \text{ if there is no edge.}$$

1. Adjacency Matrix for Directed Graph:-



	V ₁	V ₂	V ₃	V ₄
V ₁	0	1	1	0
V ₂	0	0	0	1
V ₃	0	1	0	0
V ₄	0	0	1	0

Example $V_{1,2} = 1$ Since there is an edge V_1 to V_2

Similarly $V_{1,3} = 1$, there is an edge V_1 to V_3

$V_{1,4} = 0$, there is no edge.

2. Adjacency Matrix for undirected Graph :-

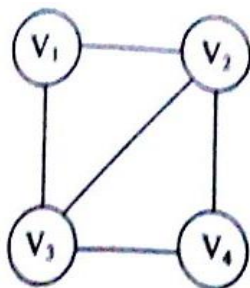


Fig. 7.2.3

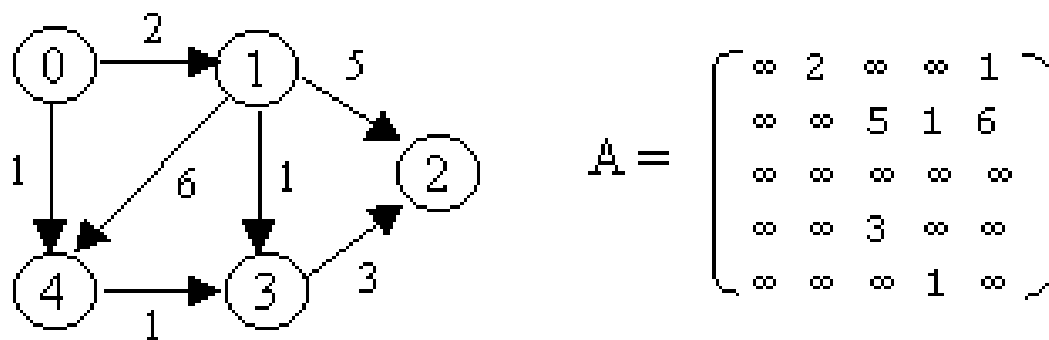
	V ₁	V ₂	V ₃	V ₄
V ₁	0	1	1	0
V ₂	1	0	1	1
V ₃	1	1	0	1
V ₄	0	1	1	0

3. Adjacency Matrix for a Weighted Graph

Adjacency matrix can be constructed as

$A_{ij} = C_{ij}$, if there exists an edge from V_i to V_j

$A_{ij} = 0$, if there is no edge & $i = j$

**Advantage :**

Simple to implement

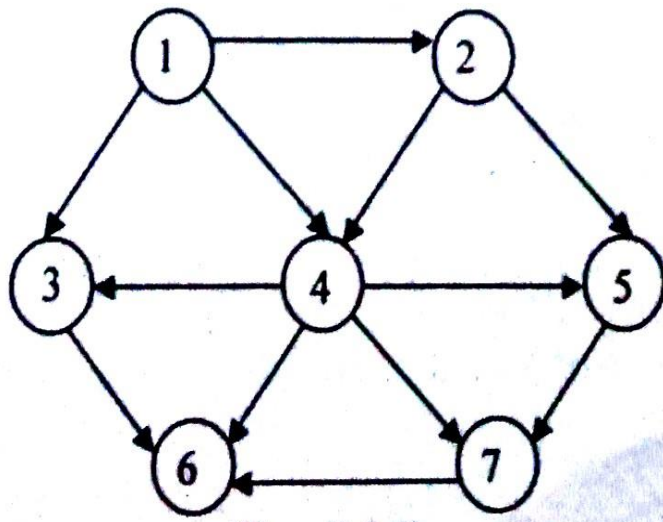
Disadvantage :

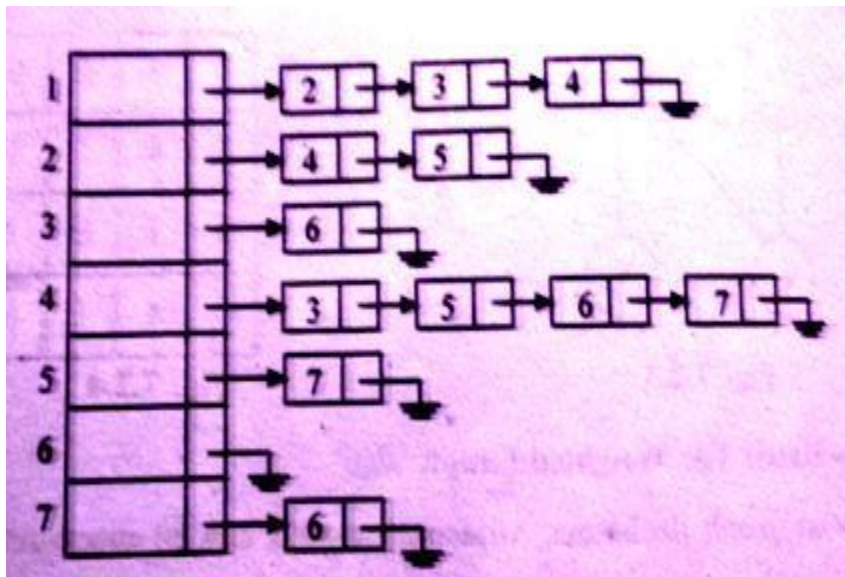
Take $O(n^2)$ space to represent the graph

Take $O(n^2)$ time to solve the most of the problems

ADJACENCY LIST REPRESENTATION

In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices.

**Adjacency List:-**



Disadvantage :

It takes $O(n)$ time to determine whether there is an arc from vertex i to vertex j . Since there can be $O(n)$ vertices on the adjacency list for vertex i .

GRAPH TRAVERSALS

Some applications require the graph to be traversed . This means that starting from some designated vertex the graph is walked around in a symmetric way such that vertex is visited only once. Two algorithms are commonly used:

1. Depth-First Search
2. Breadth-First Search

1. DEPTH FIRST SEARCH

1. DFS stands for “Depth First Search”.
2. DFS is not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.
3. DFS starts the traversal from the root node and explore the search as far as possible

from the root node i.e. depth wise.

4. Depth First Search can be done with the help of Stack i.e. LIFO implementations.
5. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
6. DFS is more faster than BFS.
7. DFS require less memory compare to BFS.

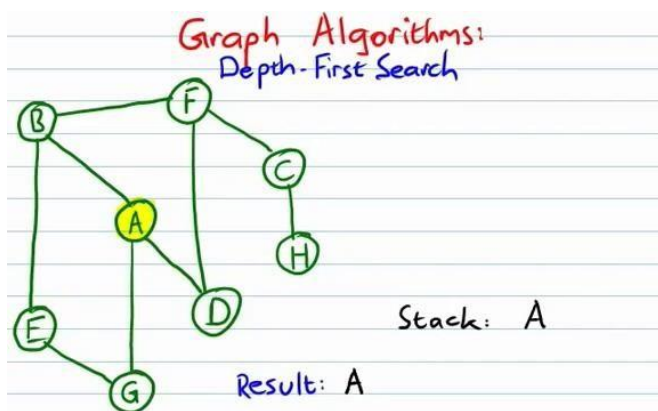
Applications of DFS

1. Useful in Cycle detection
2. In Connectivity testing
3. Finding a path between V and W in the graph
4. useful in finding spanning trees & forest.

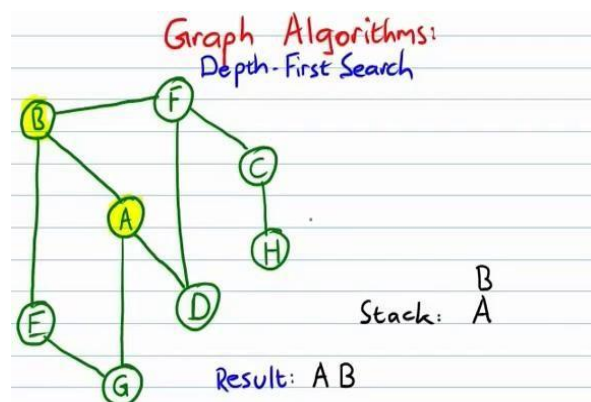
Algorithm: Depth First Search

1. PUSH the starting node in the Stack.
2. If the Stack is empty, return failure and stop.
3. If the first element on the Stack is a goal node g , return succeed and stop otherwise.
4. POP and expand the first element and place the children at the front of the Stack.
5. Go back to step 2

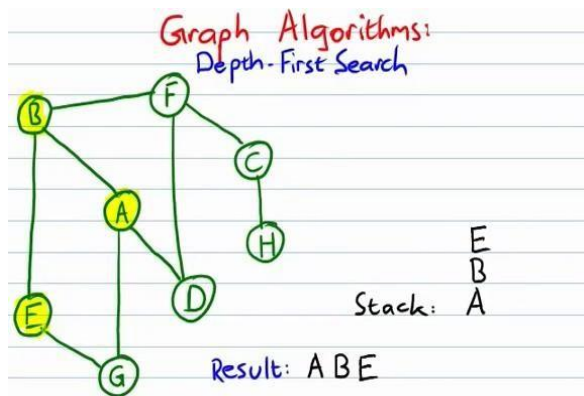
Step 1



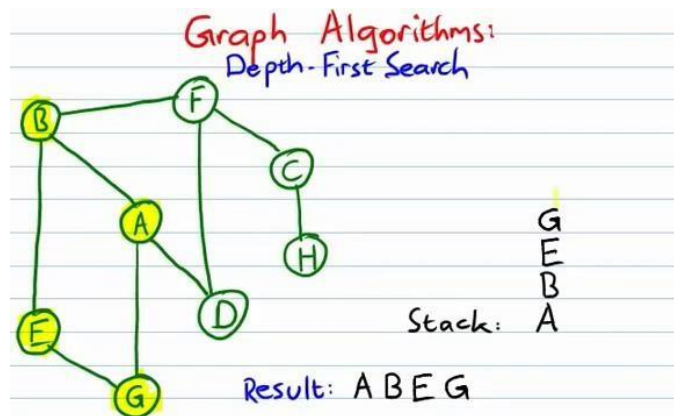
Step 2



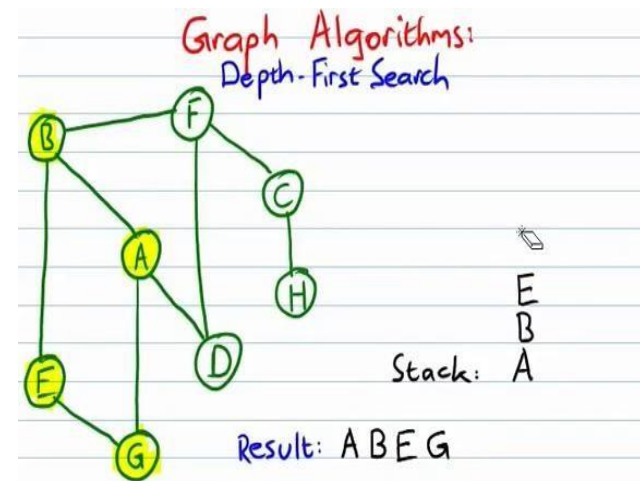
Step 3



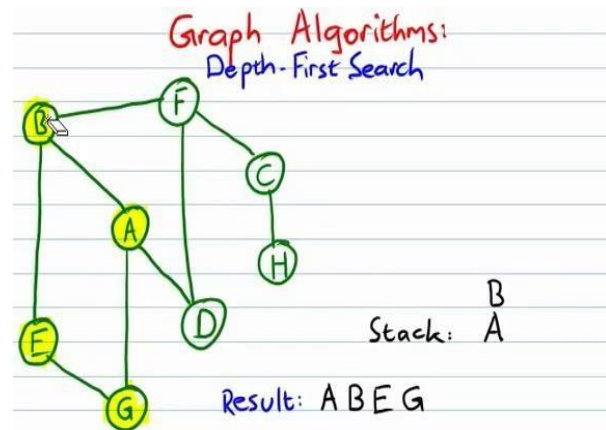
Step 4



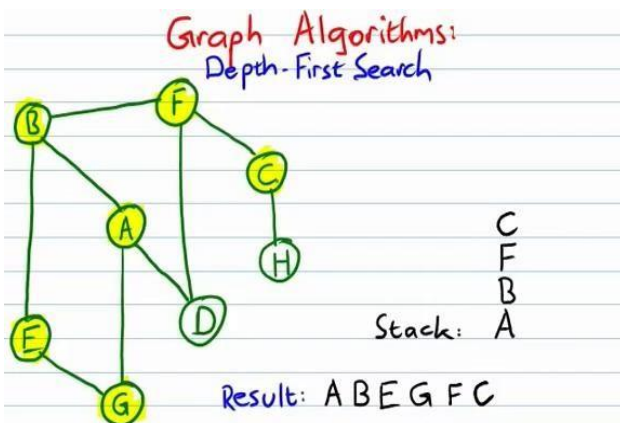
Step 5



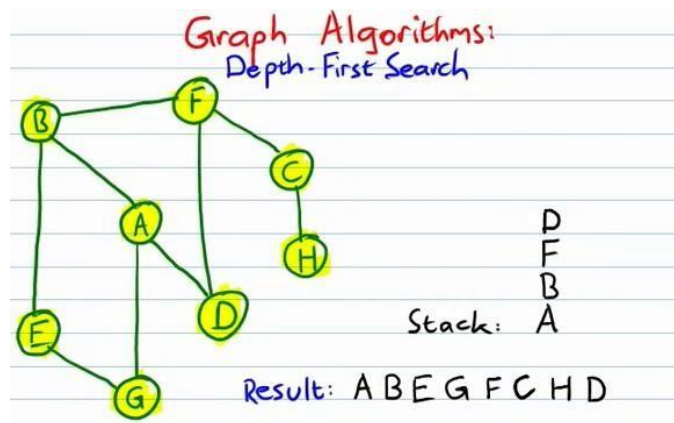
Step 6



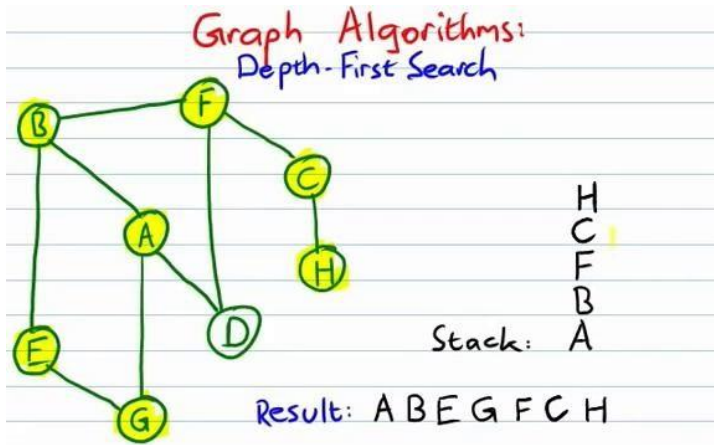
Step 7



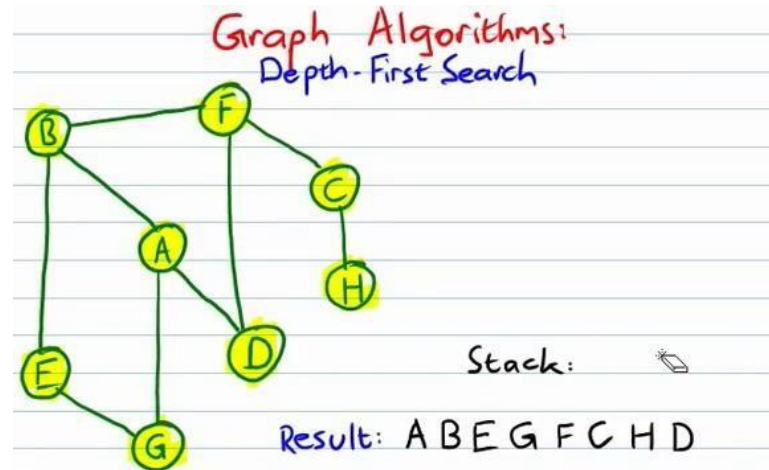
Step 8



Step 9



Step 10



2. BREADTH FIRST SEARCH:-

1. BFS Stands for “Breadth First Search”.
2. BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.
3. BFS is useful in finding shortest path. BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.
4. Breadth First Search can be done with the help of queue i.e. FIFO implementation.
5. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.
6. BFS is slower than DFS.
7. BFS requires more memory compare to DFS.

Applications of BFS

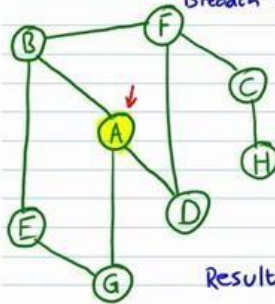
1. To find Shortest path
2. Single Source & All pairs shortest paths
3. In Spanning tree
4. In Connectivity

ALGORITHM: BREADTH - FIRST SEARCH

- 1. Place the starting node on the queue.**
- 2. If the queue is empty return failure and stop.**
- 3. If the first element on the queue is a goal node, return success and stop otherwise.**
- 4. Remove and expand the first element from the queue and place all children at the end of the queue in any order.**
- 5. Go back to step 1.**

Step 1

Graph Algorithms:
Breadth-First Search

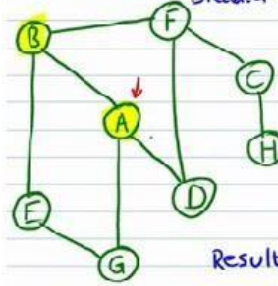


Queue:

Result: A :

Step 2

Graph Algorithms:
Breadth-First Search

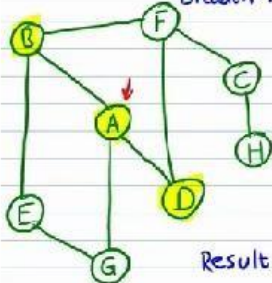


Queue: B

Result: A B

Step 3

Graph Algorithms:
Breadth-First Search

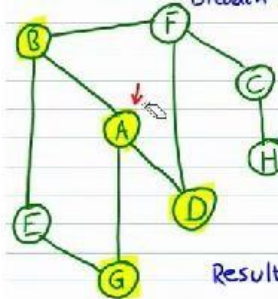


Queue: B
D

Result: A B D

Step 4

Graph Algorithms:
Breadth-First Search

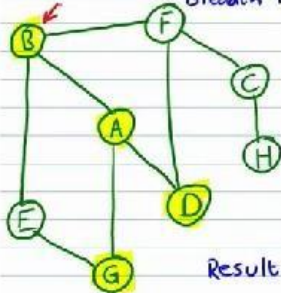


Queue: B
D
G

Result: A B D G

Step 5

Graph Algorithms:
Breadth-First Search

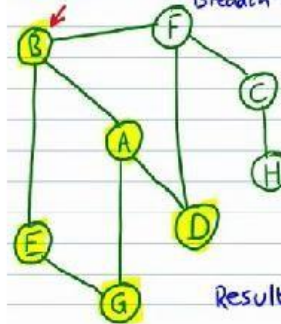


Queue: ~~B~~
D
G

Result: A B D G

Step 6

Graph Algorithms:
Breadth-First Search

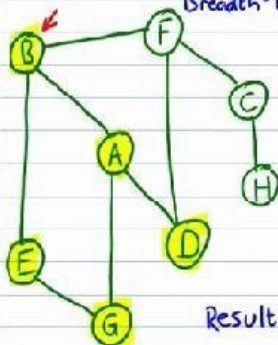


Queue: ~~B~~
D
G
E

Result: A B D G E

Step 7

Graph Algorithms:
Breadth-First Search

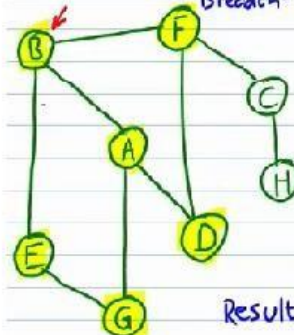


Queue: ~~B~~
D
G
E

Result: A B D G E

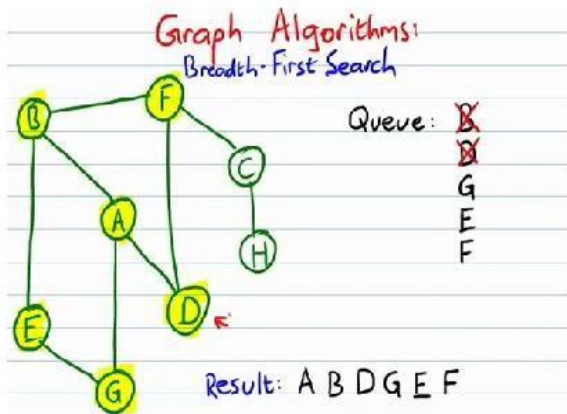
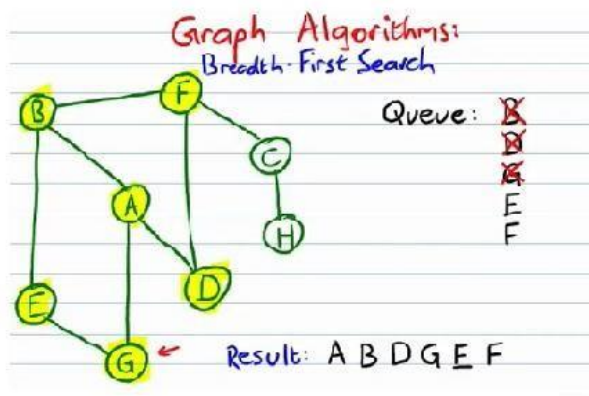
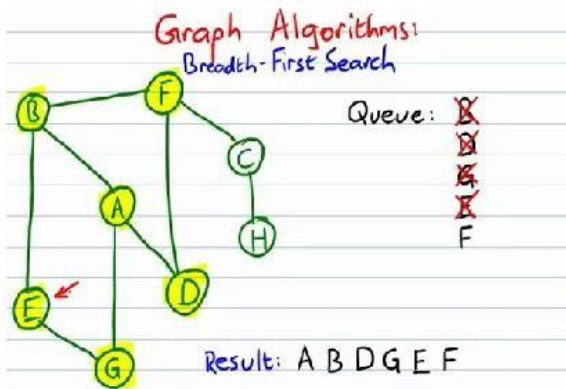
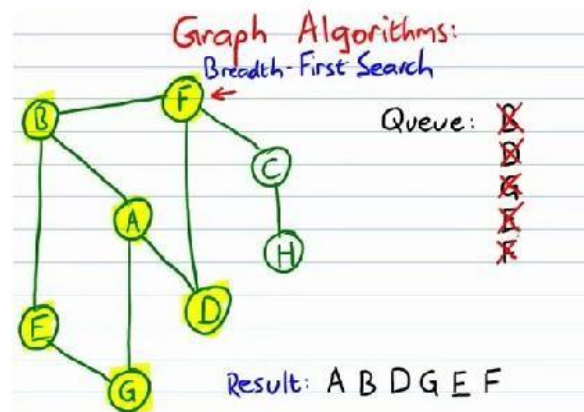
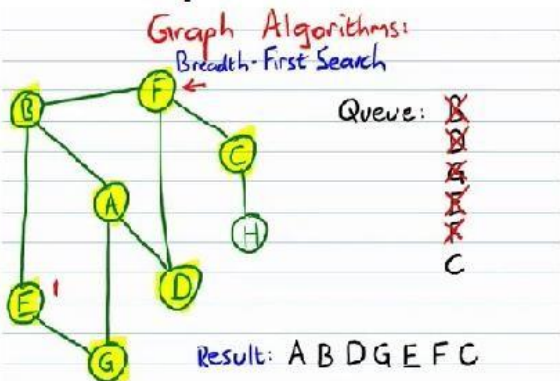
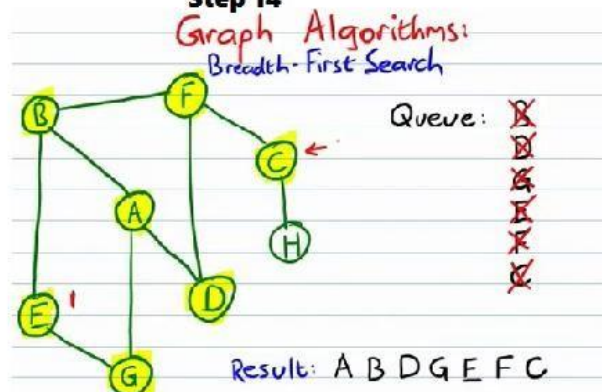
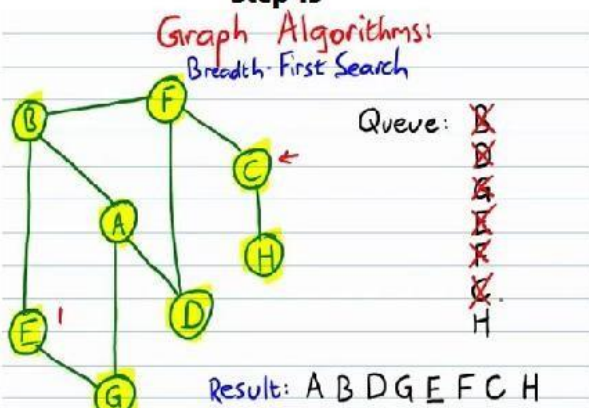
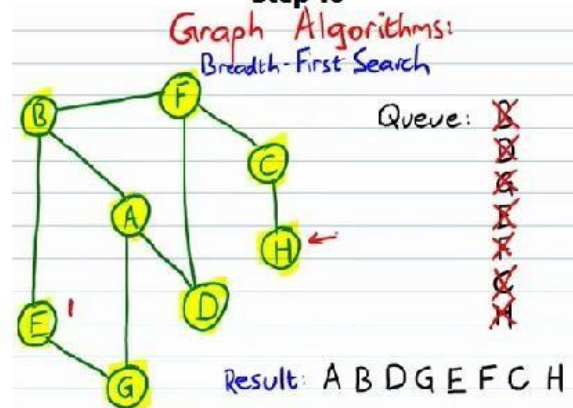
Step 8

Graph Algorithms:
Breadth-First Search



Queue: ~~B~~
D
G
E
F

Result: A B D G E F

Step 9**Step 10****Step 11****Step 12****Step 13****Step 14****Step 15****Step 16**

APPLICATIONS OF GRAPH TRAVERSAL:

Two simple applications of graph traversal are:

- 1.finding the component of a graph and*
- 2.finding a spanning tree of a connected graph*

1. CONNECTED COMPONENTS:

If G is an undirected graph, we can determine whether it is connected or not connected by making a call to either DFS or BFS and then determining if there is any unvisited vertex.

The time to do this $O(n^2)$ if adjacency matrices are used and $O(e)$ if adjacency lists are used., All the connected components of a graph is obtained by making repeated calls to either DFS (V) or BFS (V) with V as vertex not and yet visited

The following algorithm determines all the connected components of G.

The algorithm uses DFS . Instead BFS may be used. The computing time is not affected.

ALGORITHM:

```

Procedure COMP(G, n)
/ determine the connected components of G. G has  $n > 1$  vertices. VISITED is now a local
array /
  for  $i \leftarrow 1$  to  $n$  do
     $VISITED(i) \leftarrow 0$  / initialise all vertices as unvisited /
  end
  for  $i \leftarrow 1$  to  $n$  do
    if  $VISITED(i) = 0$  then / call DFS(i) / find a component / output all newly visited vertices
    together with all edges incident to them/
    end
  end
end COMP

```

2.MINIMUM SPANNING TREE

- A spanning tree S is a sub set of Tree T in which all the vertices of tree T are present but it may not contain all the edges.
- A minimum-cost spanning tree is a spanning tree of least cost.

Two algorithms are used to obtain a minimum-cost spanning tree:

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

1. PRIM'S ALGORITHM

- Prim's algorithm constructs the minimum-cost spanning tree edge by edge.
- Prim's algorithm begins with a tree T that contains a single vertex.
- Then add the least cost edge (u,v) to T such that $T \cup \{(u,v)\}$ is also a tree.
- This edge-addition step is repeated until T contains $n-1$ edges.
- The edges (u,v) is always such that exactly one of u and v is in T .

Algorithm :-

//Assume that G has at least one vertex.

$TV = \{0\}$; //Start with vertex 0 and no edges.

for ($T = \text{Null}$; T contains fewer than $n-1$ edges; add (u,v) to T)

{

let (u,v) be a least-cost edge such that $u \in TV$ and $v \notin TV$;

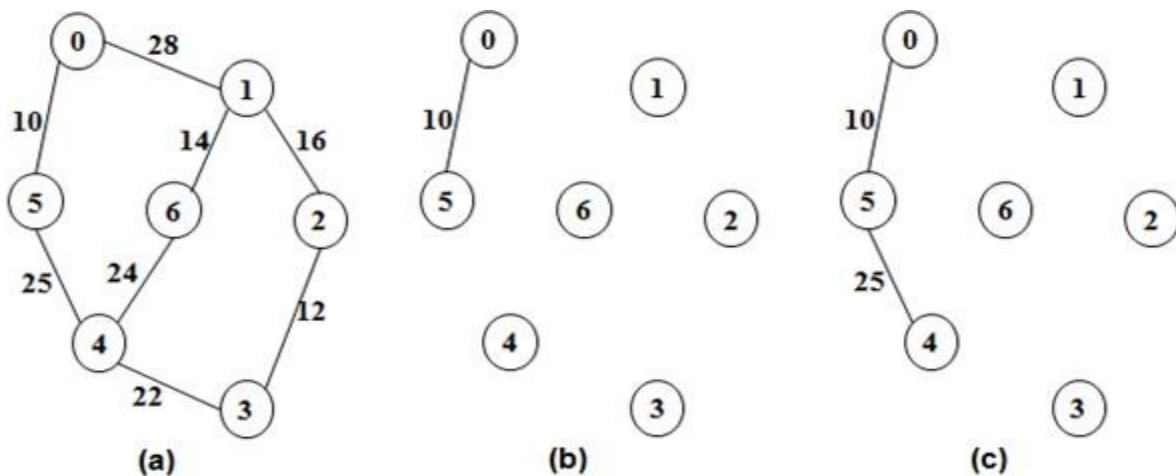
if (there is no such edge) break;

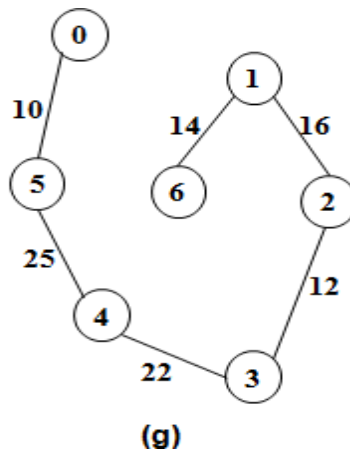
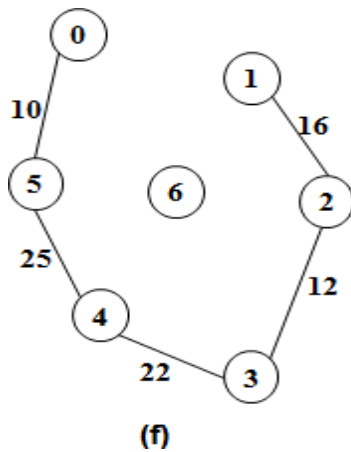
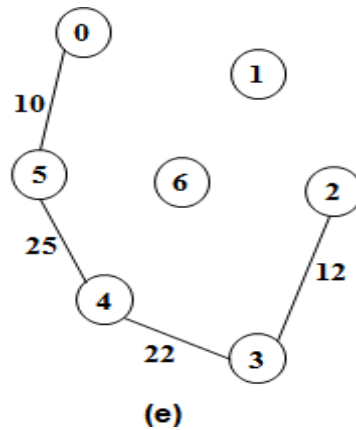
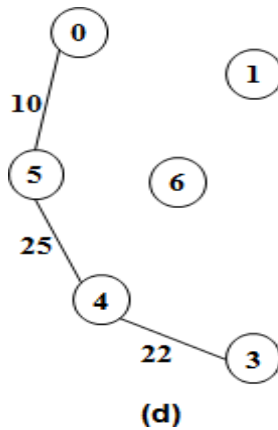
add v to TV ;

}

if (T contains fewer than $n-1$ edges) cout<< "No Spanning Tree";

Different Stages of Prim's Algorithm :-





2. KRUSKAL'S ALGORITHM

- Kruskal's algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time.
- The algorithm selects the edges for inclusion in T in increasing order of their cost.
- An edge is added to T if it does not form a cycle. With the edges that are already in T .
- Exactly $n-1$ edges will be selected for inclusion in T .

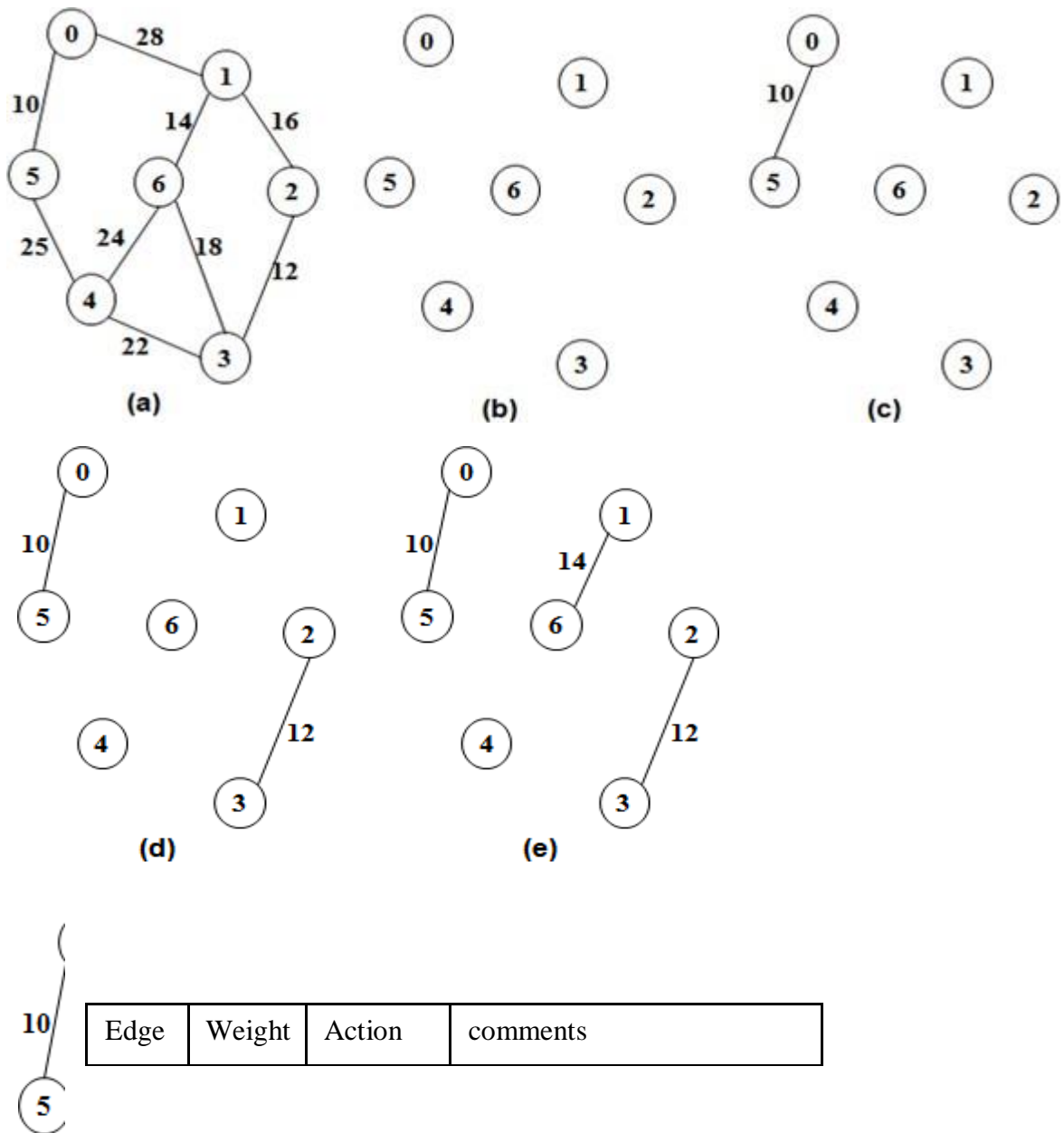
Algorithm :-

```

t = 0;
while((t contains less than n-1 edges)&&(e not empty))
{
  choose an edge(v,w) from e of lowest cost;
  delete (v,w) from e;
  if ((v,w) does not create a cycle in t) add (v,w) to t;
  else discard(v,w);
}
if ( t contains fewer than n-1 edges) cout << " no spanning tree" << endl;

```

Different Stages In Kruskal's Algorithm



Action – Kruskal's Algorithm

(A,F)	10	Accepted	An edge with minimum cost.
(C,D)	12	Accepted	
(B,G)	14	Accepted	
(B,C)	16	Accepted	
(D,E)	22	Accepted	
(E,G)	24	Rejected	Forms a cycle
(F,E)	25	Accepted	
(A,B)	28	Rejected	Forms a cycle

SHORTEST PATH ALGORITHM

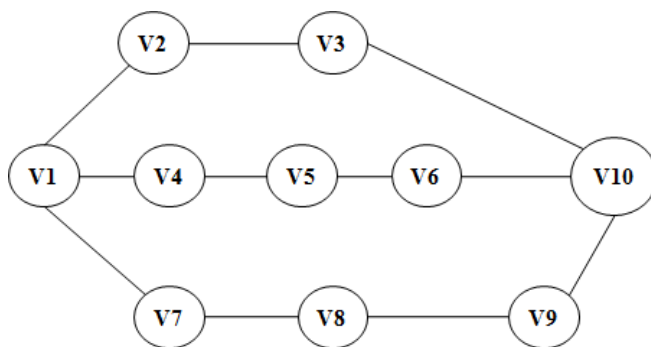
Shortest path Algorithm can be used to obtain the minimum distance between two nodes.

Two types of shortest path problems, exist namely,

1. **Single source shortest path algorithm:** finds the minimum cost from single source vertex to all other vertices. Dijkstra's algorithm is used to solve this problem which follows the greedy technique.
2. **All pairs shortest path algorithm:** finds the shortest distance from each vertex to all other vertices. To solve this problem dynamic programming technique known as floyd's algorithm is used.

1. SINGLE SOURCE SHORTEST PATH PROBLEM(Unweighted Shortest Paths)

The length of the path is equal to the number of edges travelled from source to destination.



S .No	Path	Number of Edges
1	V1 – V2 – V3 – V10	3

2	V1 – V4 – V5 – V6 – V10	4
3	V1 – V7 – V8 – V9 – V10	4

The path V1 – V2 – V3 – V10 is the shortest path from V1 to V10 as it contains only 3 edges.

DIJKSTRA'S ALGORITHM(Shortest path algorithm)

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s , it grows a tree, T , that ultimately spans all vertices reachable from S . Vertices are added to T in order of distance i.e., first S , then the vertex closest to S , then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

INITIALIZE SINGLE-SOURCE (G, s)

$S \leftarrow \{ \}$ // S will ultimately contains vertices of final shortest-path weights from s

Initialize priority queue Q i.e., $Q \leftarrow V[G]$

while priority queue Q is not empty do

$u \leftarrow \text{EXTRACT_MIN}(Q)$ // Pull out new vertex

$S \leftarrow S \cup \{u\}$

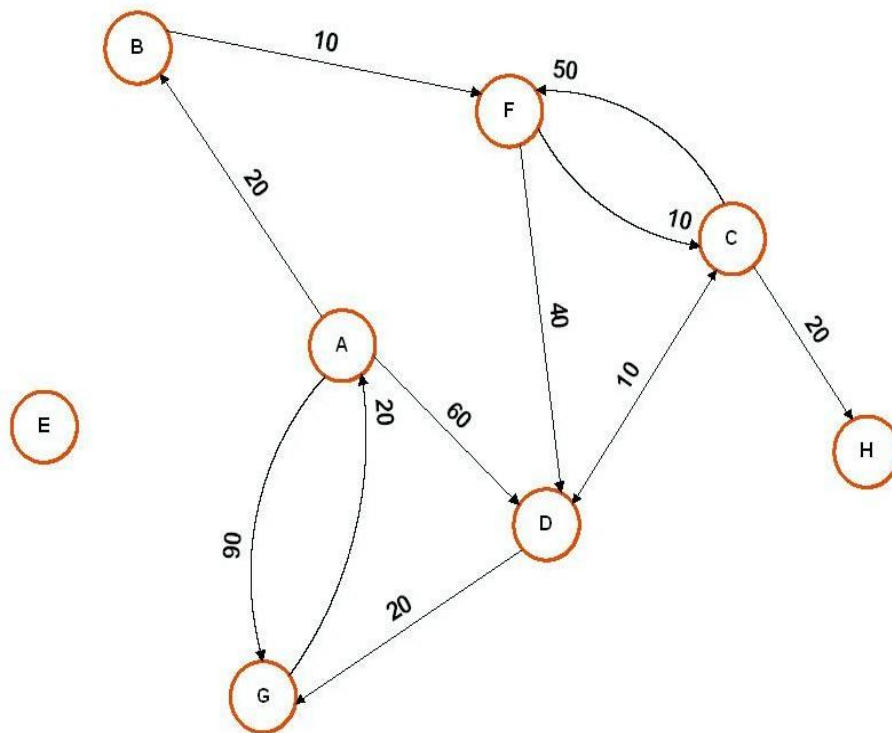
 // Perform relaxation for each vertex v adjacent to u

 for each vertex v in $\text{Adj}[u]$ do

 Relax (u, v, w)

END DIJKSTRA

Example 1 :



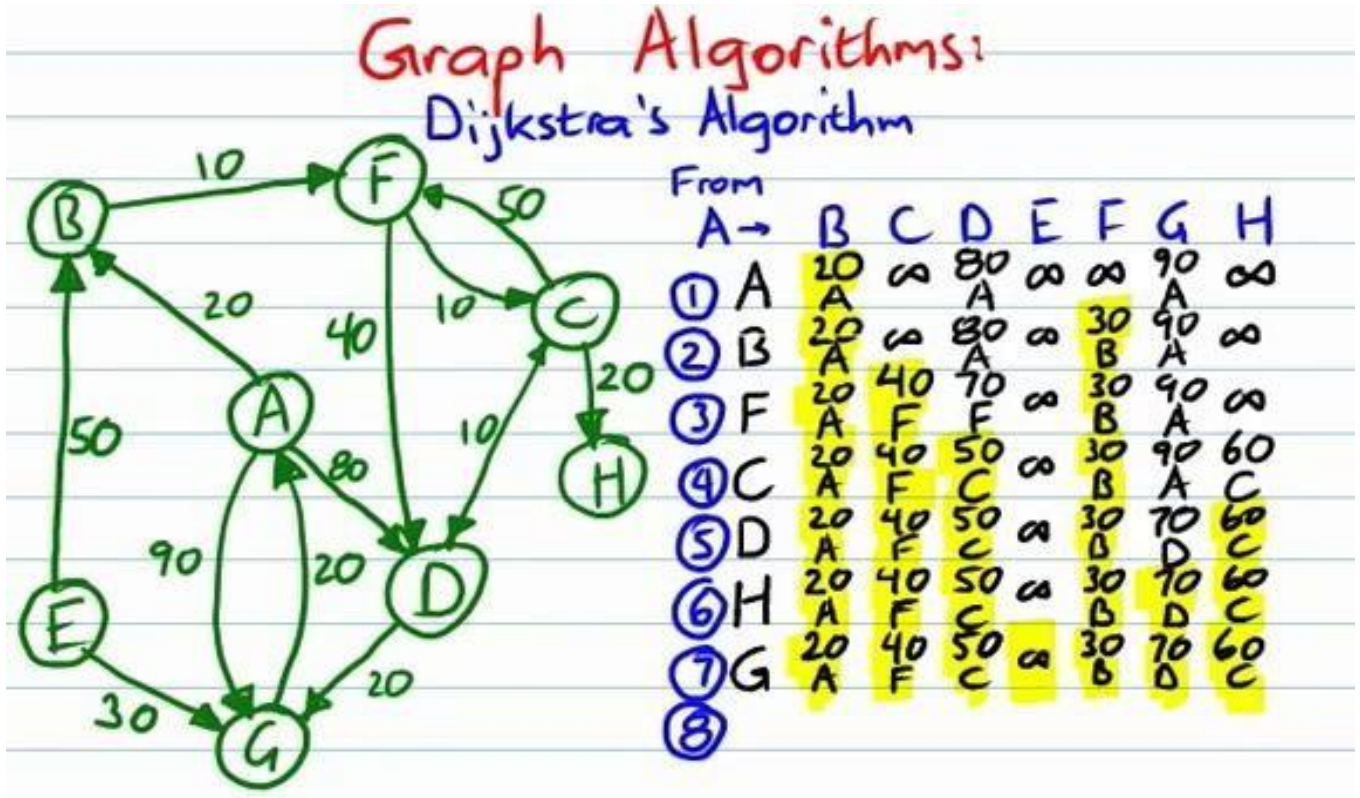
From		A	B	C	D	E	F	G	H
1	A → B	20			80			90	
2	A	A	∞	A	∞	∞	A	∞	
3	B	A	∞	A	∞	B	A	∞	
4	F	A	F	F	∞	B	A	∞	
5	C	A	F	C	∞	B	A	C	
6	D	A	F	C	∞	B	D	C	
7	H	A	F	C	∞	B	D	C	
8	G	A	F	C	∞	B	D	C	

- So with this „Graph Algorithm“ we found our best lowest cost route in this interconnected Vertex.
- And the best lowest cost path is given below:

A -> B -> F -> C -> D -> (H) -> G

- So total cost from „A“ to „G“ vertex is „70“ which is lowest cost from other Vertex.

Example 2 : -



2.ALL PAIRS SHORTEST PATH PROBLEM(Floyd's Algorithm)

Computes shortest distance between all pairs of nodes, and saves P to enable finding shortest paths.

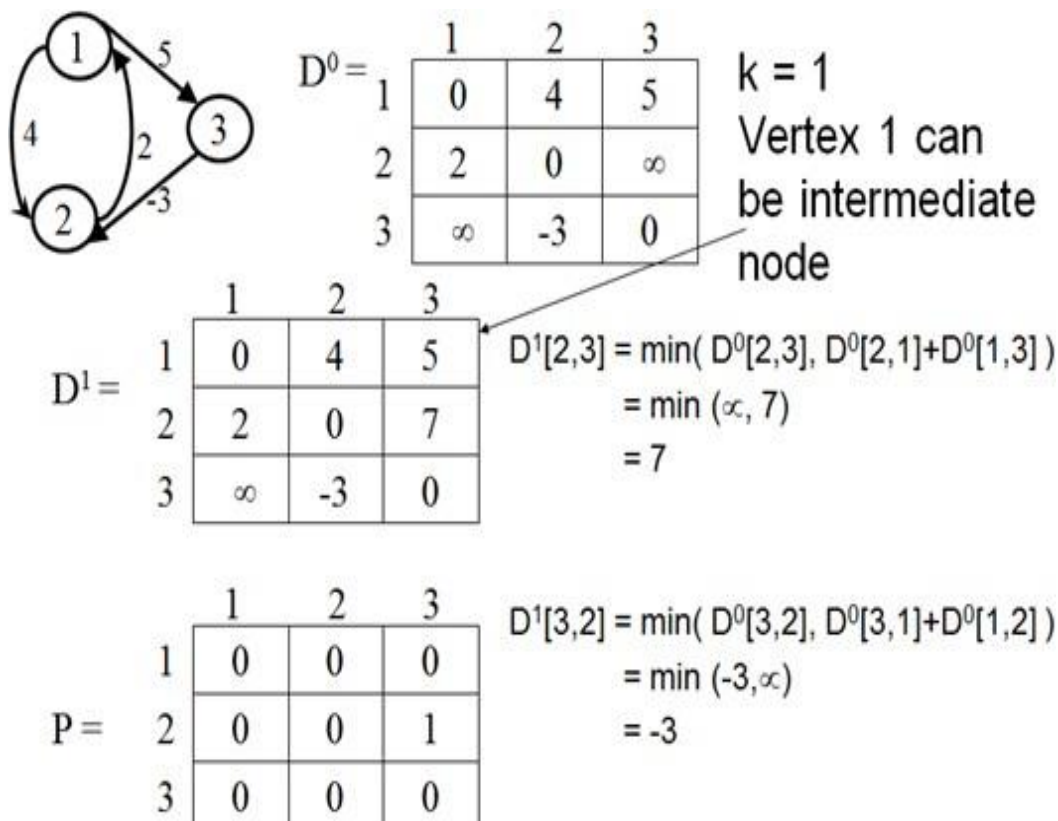
Algorithm:

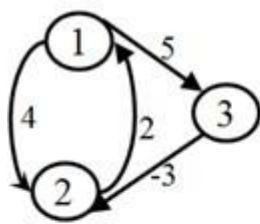
```

1.  $D^0 \leftarrow W$  // initialize  $D$  array to  $W[ ]$ 
2.  $P \leftarrow 0$  // initialize  $P$  array to  $[0]$ 
3. for  $k \leftarrow 1$  to  $n$ 
4.   do for  $i \leftarrow 1$  to  $n$ 
5.     do for  $j \leftarrow 1$  to  $n$ 
6.       if ( $D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j]$ )
7.         then  $D^k[i, j] \leftarrow D^{k-1}[i, k] + D^{k-1}[k, j]$ 
8.          $P[i, j] \leftarrow k$ ;
9.       else  $D^k[i, j] \leftarrow D^{k-1}[i, j]$ 

```

Demonstrate an example:





$$D^1 =$$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$k = 2$
Vertices 1, 2
can be
intermediate

$$D^2 =$$

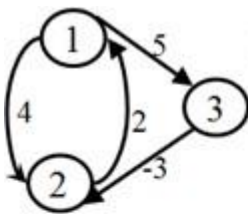
	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$$\begin{aligned} D^2[1,3] &= \min(D^1[1,3], D^1[1,2] + D^1[2,3]) \\ &= \min(5, 4+7) \\ &= 5 \end{aligned}$$

$$P =$$

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$$\begin{aligned} D^2[3,1] &= \min(D^1[3,1], D^1[3,2] + D^1[2,1]) \\ &= \min(\infty, -3+2) \\ &= -1 \end{aligned}$$



$$D^2 =$$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$k = 3$
Vertices 1, 2, 3
can be
intermediate

$$D^3 =$$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$$\begin{aligned} D^3[1,2] &= \min(D^2[1,2], D^2[1,3] + D^2[3,2]) \\ &= \min(4, 5+(-3)) \\ &= 2 \end{aligned}$$

$$P =$$

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

$$\begin{aligned} D^3[2,1] &= \min(D^2[2,1], D^2[2,3] + D^2[3,1]) \\ &= \min(2, 7+(-1)) \\ &= 2 \end{aligned}$$

Hashing

Having an insertion, find and removal of $O(\log(N))$ is good but as the size of the table becomes larger, even this value becomes significant. We would like to be able to use an algorithm for finding of $O(1)$. This is when hashing comes into play!

Hashing using Arrays

When implementing a hash table using arrays, the nodes are not stored consecutively, instead the location of storage is computed using the key and a *hash* function. The computation of the array index can be visualized as shown below:

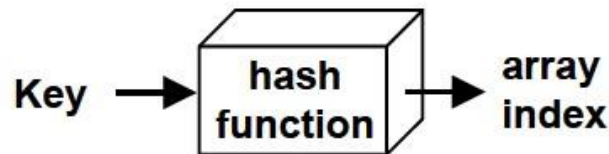


Figure 5. Array Index Computation

The value computed by applying the hash function to the key is often referred to as the hashed key. The entries into the array, are scattered (not necessarily sequential) as can be seen in figure below.

	key	entry
4	<key>	<data>
10	<key>	<data>
123	<key>	<data>

Figure 6. Hashed Array

The cost of the insert, find and delete operations is now only $O(1)$. Can you think of why?

Hash tables are very good if you need to perform a lot of search operations on a relatively stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).

One the other hand, if traversals (covering the entire table), insertions, deletions are a lot more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

Selecting Hash Functions

The hash function converts the key into the table position. It can be carried out using:

- Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques.

For Example: $\text{index} := \text{key} \text{ MOD } \text{table_size}$

- Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table.

For Example: If student id's are the key 928324312 then select just the last three digits as the index i.e. 312 as the index. \Rightarrow the table size has to be atleast 999. Why?

- Folding: Partition the key into several pieces and then combine it in some convenient way.

For Example:

- For an 8 bit integer, compute the index as follows:
 $\text{Index} := (\text{Key}/10000 + \text{Key} \text{ MOD } 10000) \text{ MOD } \text{Table_Size}.$
- For character strings, compute the index as follows:

Hashing Performance

There are three factors the influence the performance of hashing:

- Hash function
 - should distribute the keys and entries evenly throughout the entire table
 - should minimize collisions
- Collision resolution strategy
 - Open Addressing: store the key/entry in a different position
 - Separate Chaining: chain several keys/entries in the same position
- Table size
 - Too large a table, will cause a wastage of memory
 - Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
 - The size should be appropriate to the hash function used and should typically be a prime number. Why? (We discussed this in class).


```

Index := 0
For I in 1.. length(string)
Index := Index + ascii_value(String(I))

```

Collision

Let us consider the case when we have a single array with four records, each with two fields, one for the key and one to hold data (we call this a *single slot bucket*). Let the hashing function be a simple modulus operator i.e. array index is computed by finding the remainder of dividing the key by 4.

$$\text{Array Index} := \text{key MOD } 4$$

Then key values 9, 13, 17 will all hash to the same index. When two(or more) keys hash to the same value, a **collision** is said to occur.

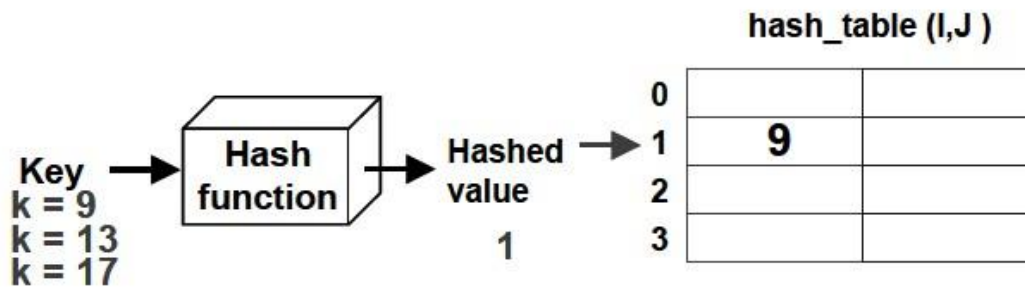


Figure 7. Collision Using a Modulus Hash Function

Collision Resolution

The hash table can be implemented either using

- **Buckets:** An array is used for implementing the hash table. The array has size $m \cdot p$ where m is the number of hash values and $p (\geq 1)$ is the number of slots (a slot can hold one entry) as shown in figure below. The *bucket* is said to have p slots.

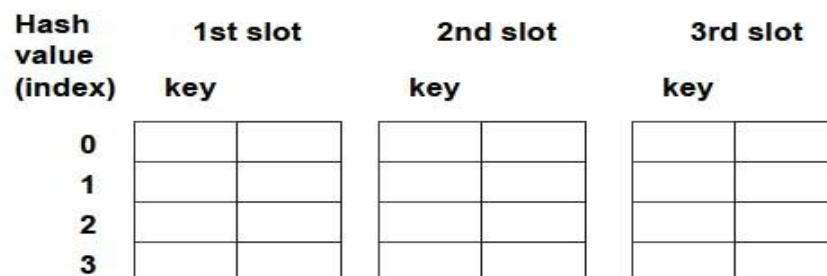


Figure 8. Hash Table with Buckets

Chaining: An array is used to hold the key and a pointer to a linked list (either singly or doubly linked) or a tree. Here the number of nodes is not restricted (unlike with buckets). Each node in the chain is large enough to hold one entry as shown in figure below.

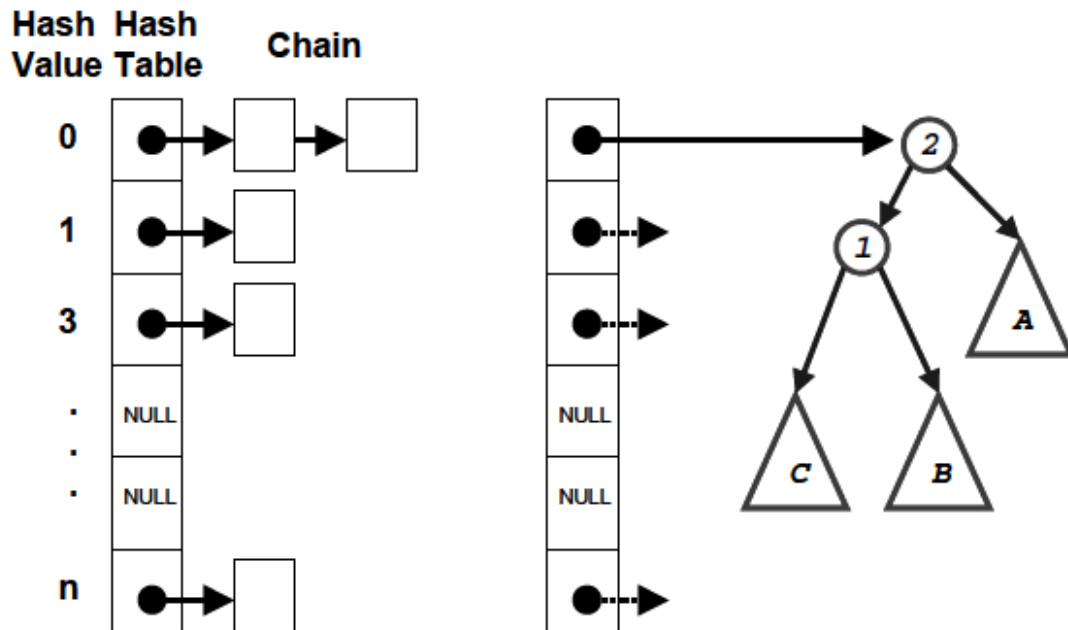


Figure 9. Chaining using Linked Lists / Trees

Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.

There are three schemes commonly used for probing:

- Linear Probing: The linear probing algorithm is detailed below:

```

Index := hash(key)
While Table(Index) Is Full do
    index := (index + 1) MOD Table_Size
if (index = hash(key))
    return table_full
else
    Table(Index) := Entry
  
```

- Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16,

- Double Hash: compute the index as a function of two different hash functions.

Chaining

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

The advantages of using chaining are

- Insertion can be carried out at the head of the list at the index
- The array size is not a limiting factor on the size of the table

The prime disadvantage is the memory overhead incurred if the table size is small.

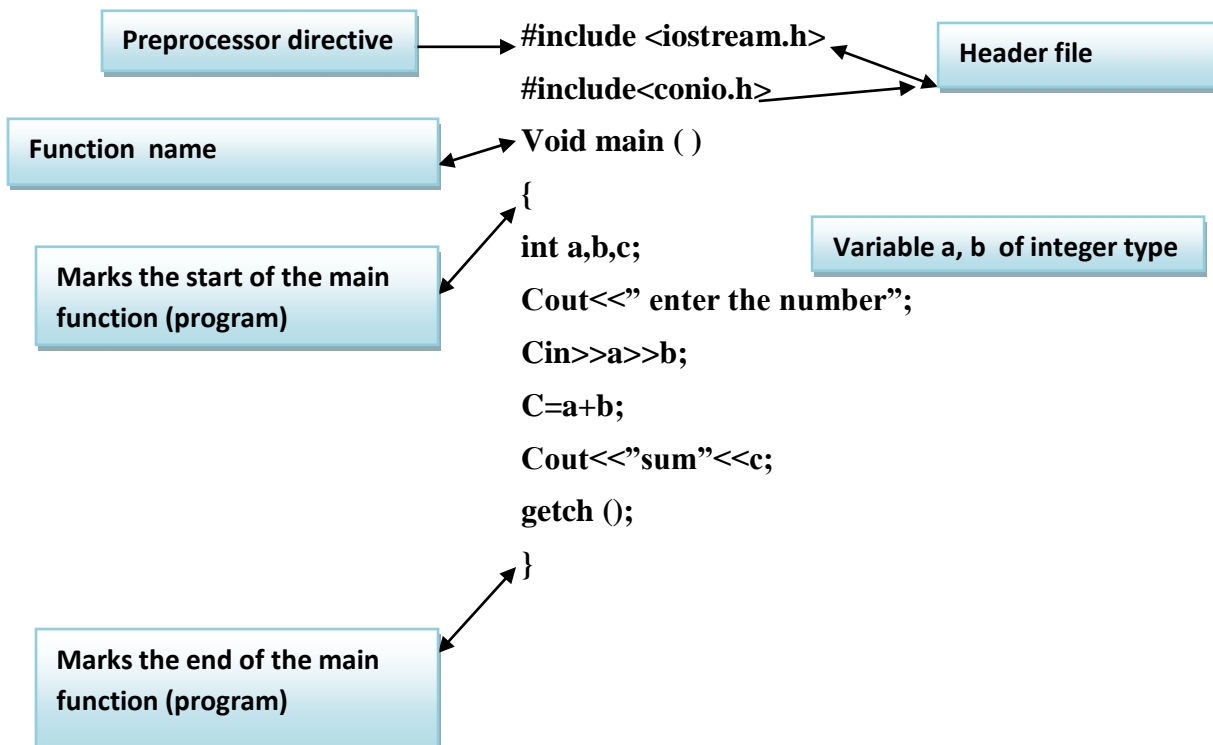
UNIT IV: PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

Principles of Object Oriented Programming - Beginning with C++ - Tokens-Expressions-control Structures – Functions in C++ - classes and objects-constructors and destructors -operators overloading and type Conversions

The history of C++:

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. C++ is an object oriented programming language, it implements “data abstraction” using a concept called “classes” along with some other features of oop, a part of the c++ program are easily reusable and extensible code is easily modifiable without actually having to change the code . The “++” is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C .It contains all features of oops.

A simple program:- A simple program for the addition of two numbers-



What is meant by object-oriented programming?

- OOPs is the new concept of programming parallel to Procedure oriented programming.
- It were introduced in late 80's. It considers the programming simulated to real world objects.
- It helps in programming approach in order to build robust user friendly and efficient softwares and provide the efficient way to maintain real world softwares.
- OOPs is an Object Oriented Programming language which is the extension of Procedure Oriented Programming language.
- OOPs reduce the code of the program because of the extensive feature of Polymorphism. OOPs have many properties such as Data Hiding Inheritance
- Data Abstraction Data Encapsulation and many more. OOPs is Object oriented programming language.

- The main aim is to creating an Object to The Entire program and that to we can control entire program using the Object.
- the main features of OPPS is Polymorphism, Multiple Inheritance, abstraction and encapsulation.

Basic concepts of oops:

The different concepts of OOPs are as follows

- (a) **Encapsulation:** It is used to hide the data as well as the binding of a data members and member functions.
- (b) **Inheritance:** It is the process by which one class inherits the properties of another Class.
- (c) **Polymorphism:** poly means many and morphs means form, so polymorphism Means one name multiple form. there are two types of polymorphism : compile time and run time polymorphism.
- (d) **Data hiding:** This is the property in which some of the members are restricted from Outside access. This is implemented by using private and protected access specifies.
- (e) **Data abstraction:** This is the property in which only necessary information is Extracted. This property is implemented by using the class in C++.
- (f) **Class:** It is a user defines data type which contains data member & member function.
It is collection of various kind of object. It is define by class keyword. It also an Important feature of object oriented programming language. For ex-fruit is a class And apple, mango, banana are its object.
- (g) **Object:** An object is a basic run time entity. Object represents/resembles a Physical/real entity.

An object is simply something you can give a name.

All the objects have some characteristics and behaviour. the state of an object represent all the information held within it and behavior of an object is the set of action that it can perform to change the state of the object.

All real world object have three characteristics:

- State : How object react?
- Behaviour : what we can do with this object?
- Identity : difference between one object to another object?

For ex- our bike

- State : (gear, speed, fuel)
- Behaviour : (changing speed, applying brakes)
- Identity : (registration number, engine number)

- (h) **Overloading:** Adding a new method with the same name in same/derived class but With different number/types of parameters. It implements Polymorphism.

Write the merits and demerits of object-oriented language as compared to procedure-oriented language.

Ans: We can compare the procedure-oriented programming(c) with the object-oriented (c++) .

- pop focus more on function
- oop focus on data
- oop deals with real world object
- In pop error detection is difficult as we can't know which Variable is associated with which function
- In oop we can specify with the object that which variable is Associated with which function
- objects in oop creates many modules of program which is Flexible and easier to execute and also understand
- OOP provides inheritance in which features can be added to Existing classes without modification

1. In pop importance is given for doing things. In oop importance is given on data rather than procedure.
2. Pop, most of function share global data. In oop, data structure are designed such that the characteristics the object function that operate on the data of an object which are tied together in the data structure
3. Pop, larger programs are divided into smaller programs known as function. In oop, programs are divided into smaller programs known as objects.
4. In pop security is not provided for object. In oop security is provided to data.
5. In pop top down approach. In oop bottom up approach.

Benefits of OOPs

- 1.Reusability
- 2.Code Sharing
- 3.Data Hiding
- 4.Reduced complexity of a Program
- 5.Prototyping
- 6.Message Passing
- 7.Extendability

Applications of OOPs

1. Used in Computer Animation
2. Used to design Compiler
3. Used in Logical Network Designing
4. To Develop Software administrative tools, system tools, etc...
5. Used to Access Relational Databases.
6. Used in Simulation and Modeling
7. Used to Develop expert System Software
8. Used to develop Computer Games
9. Used in electrical distribution design Systems

Tokens - The smallest individual units in a program are known as tokens, c++ has the following tokens:

Example of tokens:- } , {, “ “, int

- Keywords
- Identifiers
- Constants

Identifier:

In our everyday, we give names to different things so they can be Referred easily. Similarly, in C+, we use identifiers to name user created entities Which may be?

- Variable
- Function
- Type e.g. a class

Every thing has some restrictions and exceptions along with many permissible things. So, does C++ by putting some restrictions on how we can name these entities.

Let us see these rules in details:

1. An identifier can be combination of letters, numbers, and underscores with following restrictions:
 - a) It should start with a letter or underscore. E.g. height, my_height, _myHeight are allowed but not is Good
 - b) If it starts with a underscore then the first letter should not be capital because such names are reserved for implementation. E.g. _Height not allowed
2. It should be unique in a program taking care that C++ is case sensitive. E.g. age and Age are different variables
3. A keyword cannot be used as an identifier.
4. There is no restriction on length of the identifier. E.g. h and h_represents_my height are both valid.

Besides restrictions, there are certain guidelines which you should follow:

- a.) Use meaningful descriptive names. E.g. int Age is better than int a.
 - If description makes identifier name too long then put a comment before identifier and make identifier shorter
- b.) Be consistent in your naming convention.
 - Use small letters for single word identifier name.
 - For multiword identifiers, either use underscore separated or intercepted notation. E.g. get_my_height () or getMyHeight ()
- c.) Use Hungarian notation. E.g. double dFlowRate, int value, bool check.
- d.) Don't use similar names in a program like Speed, speed, and Speedy
- e.) Don't use capitalized version of a keyword like Return

Keywords:

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.

Keyword is a word that the compiler already knows, i.e. when the compiler sees a keyword somewhere in the program it knows what to do automatically.

For example, when the compiler encounters the keyword „int“, it knows that „int“ stands for an integer. Or if the compiler reads a „break“, then it knows that it should break out of the current loop. Some common keywords are-

auto const double float int short struct unsigned
 break continue else for long signed switch void
 case default enum goto register sizeof typedef volatile
 char do extern if return static union while

Constant :

As the name suggests, a variable is something whose value can be changed throughout the program. It is not fixed. On the other hand, a constant is one whose value remains the same (constant) throughout the program.

- **Variable:** A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore.
- **Variable declaration**

declaration : int a;
 declaration means here a is declared as integer variable

Declaring and defining variables

A variable in C++ must be declared (the type of variable) and defined (values assigned to a variable) before it can be used in a program. Following shows how to declare a variable.

Rules of variable declaration

- A variable name can have one or more letters or digits or underscore, for example character _.
- White space, punctuation symbols or other characters are not permitted to denote variable name. .
- A variable name must begin with a letter.
- Variable names cannot be keywords or any reserved words of the C++ programming language.
- C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters.
- For example, the variable name CIST differs from the variable name cist.

Variable Definition vs Declaration

Definition Ex - int a=5	Tell the compiler about the variable: its type and name, as well as allocated a memory cell for the variable
Declaration Ex- int a	Describe information ``about" the variable, doesn't allocate memory cell for the variable

Operators: operators play a great role in any languages, the operations are represented by operators and the objects of the operation are referred to as operands. There are four types of operators.

- Arithmetic
- Relational
- Logical
- Short hand assignment
- Conditional Operators
- Bitwise
- Scope Resolution
- Member dereferencing operators
- Memory Release operator
- Line Feed
- Memory allocation operator
- Field Width operator

In addition, there are some special operators for special tasks.
Operator can be

- **unary (involve 1 operand) ,**
- **binary(involve 2 operands),and**

- ternary(involve 3 operands).

➤ **Arithmetic operator**

In any language, there are some operators to perform arithmetic, logical and control operations. The basic operators which are used to perform arithmetic operations on integers are as follows:-

<u>Operator</u>	<u>Operation</u>
+	Addition, also called unary addition
-	Subtraction, also called unary subtraction
*	Multiplication
/	Division
%	Modulus (remainder after division)
++	Increment
--	Decrement

The operators +, -, * and / perform the same operation as they do in other languages. The operators + and - are unary operators and can take one or two operands. The multiply and divide are called binary operators as they take two operands.

Integer division will always give the result which is of integer type and truncates the remainder. The output of integer division will be the quotient. For example, 5/2 will give the answer 2 and not 2.5. The modulus operator gives the remainder after the integer division. For example, 5/2 will give the answer 1, which is the remainder.

Example-

```
# include<iostream.h>
#include<conio.h>
Void main( )
{
  Clrscr ( );
  int a=20,b=30,d,e,f,g;
  d=a+b;
  e=a-b;
  f=a*b;
  g=a%b;
  cout<<d<<endl;
  cout<<e<<endl;
  cout<<f<<endl;
  cout<<g<<endl;
  getch( );
}
```

Result/output:

d=50, e=-10, f=600, g=0.

- **Relational operator:** The relational operator, refer to the relationship that value can have with one another, it use the Boolean values contains true or false. 0 means false and 1 means true.

Operator

Operation

>	Greater than
>=	Greater than equal to
<	Less than
<=	Less than equal to
==	Equal to
!=	Not equal to

- **Logical operator :**

Operator

Operation

&&	And
	OR
!	NOT

Table for && (And operator): The operator && corresponds with Boolean logical operation *AND*. This operator returns the value of true if both its operands are true or if it returns false. The following table reflects the value of && operator

p	q	P&&q
0	0	0
0	1	0
1	0	0
1	1	1

Table for || (OR operator): The operator || corresponds with Boolean logical operation *OR*. The operator produces a true value if either one of its two operands are true and produces a false value only when both operands are false. The following table reflects the value of || operator:

p	q	P q
1	1	1
1	0	1
0	1	1
0	0	0

- **Bitwise operator :** These operators are used to perform bitwise operations, these operations are performed on the bits of the binary pattern of the number. bitwise operators refer to

testing, setting or shifting the actual bits in a byte or word, which correspond to the char and int data types. you cannot use bitwise operator on float, double, long double, void and other complex operators.

<u>Operator</u>	<u>Operation</u>
&	And
	OR
~	NOT
^	XOR
>>	shift right
<<	shift left

Table for ^ (Exclusive OR):

p	q	P&&q
0	0	0
1	0	1
1	1	0
0	1	1

Conditional Operator

The conditional operator evaluates an expression returning a value if that expression is true and a different value if the expression is evaluated as false. The syntax is:

```
condition ? value1 : value2
```

For example: In

```
7 > 5 ? x : y
```

Since 7 is greater than 5, true is returned and hence the value x is returned.

Comma Operator

This is denoted by, and it is used to separate two or more expressions. For example:

```
exfor = (x=5, x+3);
```


Here value of 5 is assigned to x and then the value of x+3 is assigned to the variable exfor. Hence, value of the variable exfor is 8.

sizeof() Operator

This operator accepts a single parameter and this can be a variable or data type. This operator returns the size in bytes of the variable or data type.

For example:

```
x = sizeof (char);
```

This returns the size of char in bytes. In this example, the size is 1 byte which is assigned to variable x.

Data types: These are the basic data types :-

int:

This int keyword is used to declare integers, whole numbers either positive or negative. Most of the compilers treat this with a size of 2 bytes. Its range is -32768 to 32767. Integer (2, 3, 4)

Char:

This keyword is used to declare characters. The size of each character is 8 bits. i.e., 1 byte. The characters that can be used with this data type are ASCII characters. Its range is -128 to 128. character values (a, b, c, d).

float:

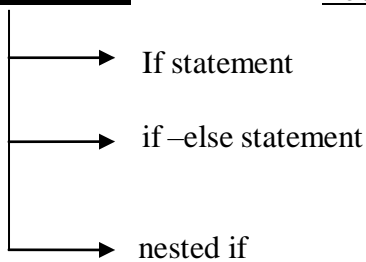
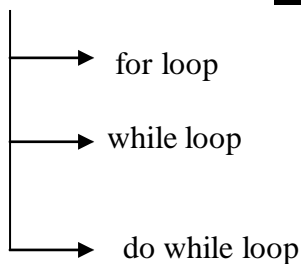
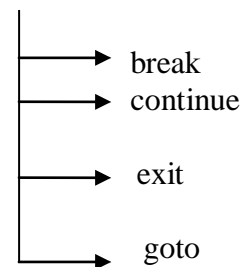
This keyword float is used to declare floating point decimal numbers. The size of each float is 4 byte. its range is -3.4E to 3.4E.

Float value(2.4,3.6,6.5).

long:

This long keyword is used for declaring longer numbers. i.e., numbers of length 32 bits.

keyword	Range (low)	(high)	Bytes of memory(size)
Char	-128	127	1
int	-32768	32767	2
long	-2147483648	2147483647	4
float	3.4 x 10 ⁻³⁸	3.4 x 10 ³⁸	4
double	1.7 x 10 ⁻³⁰⁸	1.7 x 10 ³⁰⁸	8

Control structure:**Conditional****Looping****Breaking**

If statement:- The *if* statement provides a selection control structure to execute a section of code if and only if an explicit run-time condition is met. The condition is an expression which evaluates to a boolean value, that is, either true or false.

Syntax

```

if ( <expression> )
{
    statement
}
  
```

Semantics

- The if statement provides selection control.
- The expression is evaluated first.
- If the expression evaluates to true, the statement part of the if statement is executed.
- If the expression evaluates to false, execution continues with the next statement after the if statement.
- A boolean false, an arithmetic 0, or a null pointer are all interpreted as false.
- A boolean true, an arithmetic expression not equal to 0, or a non-null pointer are all interpreted as true.

Example:

```

#include<iostream.h>
#include<conio.h>
Void main()
{
    Int a ;
    Cout<<"enter the no";
    Cin>>a;
    If(n%2==0)
    Cout<<"it is even no.";
    getch();
}
  
```

If-else: In this statement ,if the expression evaluated to true,the statement or the body of if statement is executed,otherwise the body of if statement is skipped and the body of else statement is executed

```
if (condition)
{
    statement1;
}
else
{
    statement2;
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
Void main()
{
    clrscr();
    Int n;
    Cout<<"enter the no";
    Cin>>n;
    If (n%2==0)
    Cout<<"it is even no";
    else
    Cout<<"it is odd no";
    getch();
}
```

Switch - it provide multiple branch selection statement .if –else provide only two choices for selection and switch statement provide multiple choice for selection.

Syntax-

```
switch(expression)
{
    Case :exp 1
    First case body;
    break;
    Case :exp2
    Second case of body;
    break;
    Default:
    Default case body;
}
```

Example:- #include<iostream.h>
#include<conio.h>
int a ;
Cout<<"enter the no";
Cin>>a;
Switch(a)
{

```

Case1:cout<<"Sunday\n";
Break;
Case1:cout<<"Sunday\n";
break;
Case2:cout<<"monday\n";
break;
Case3:cout<<"tuesday\n";
break;
Case4:cout<<"wednesday\n";
break;
Case5:cout<<"thrusday\n";
break;
Case6:cout<<"friday\n";
break;
Case7:cout<<"Satday\n";
break;
Default:
Cout<<"wrong option";
}
getch();
}

```

For loop- In this, first the expression or the variable is initialized and then condition is checked. if the condition is false ,the loop terminates

Syntax-

for (initillization;condition;increment)

EXAMPLE program to print the no from 1 to 100.

```

#include<iostream.h>
# include<conio.h>
Void main ()
{
int i;
for(i=1;i<=100;i++)
Cout<<i<<"\n";
getch ();
}

```

While loop: This loop is an entry controlled loop and is used when the number of iteration to be performed are known in advance. The statement in the loop is executed if the test condition is true and execution continues as long as it remains true.

Syntax-

```

initialization;
While (condition)
{
Statement;
increment;
}

```

```
}
```

Example writes a program to print a table?

```
#include<iostream.h>
#include<conio.h>
Void main ()
{
Int n,i;
Cout<<"enter the no whose table is to be printed ";
Cin>>n;
i=1;
While (i<=10)
{
Cout<<n<<"x"<<i<<"="<<nxi<<"\n";
i++;
}
getch();
}
```

Do-while - It is bottom controlled loop. This that a do-while loop always execute at least once.

Syntax-

```
initillization
Do
{
Statement ;
Increment;
}
While(condition);
```

Example: Write a program to print the table ?

```
# include<iostream.h>
#include<conio.h>
Void main()
{
int n;
Cout<<"enter the no. whose table is to be printed";
Cin>>n;
I=1;
do
{
Cout<<n<<"x"<<i<<"="<<nxi<<"\n";
getch();
}
```

Break statement- The term break means breaking out of a block of code. The break statement has two use,you can use it to terminate a case in the switch statement, and you can also use it to force immediate termination of loop,bypassing the normal loop condition test.

Example:-

```
#include<iosttream.h>
#include<conio.h>
int a ;
```

```

Cout<<"enter the no";
Cin>>a;
Switch(a)
{
Case1:cout<<"Sunday\n";
Break;
Case1:cout<<"Sunday\n";
break;
Case2:cout<<"monday\n";
break;
Case3:cout<<"tuesday\n";
break;
Case4:cout<<"wednesday\n";
break;
Case5:cout<<"thrusday\n";
break;
Case6:cout<<"friday\n";
break;
Case7:cout<<"Satday\n";
break;
Default:
Cout<<"wrong option";
}
getch();
}

```

Exit statement- Exit is a function defined in the stdlib library means (stdlib.h).

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

```
exit (exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened

Example

```

#include<iuostream.h>
#include<conio.h>
#include<stdlib.h>
Void main ()
{
Int n;
Cout<<"enter the no";
Cin>>n;
If (n%2==0)
{
Cout<<"it is even no";
else
Cout<<"it is odd no";
exit(0);
}
getch();

```

}

Continue:- The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

Example

```
#include <iostream>
#include<conio.h>
Void main ()
{
    for (int n=10; n>0; n--)
    {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "FIRE!\n";
    getch();
}
```

Output- 10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

Goto statement: allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

<pre>// goto loop example #include <iostream> using namespace std; int main () { int n=10; loop: cout << n << ", "; n--; if (n>0) goto loop; cout << "FIRE!\n"; return 0; }</pre>	10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
--	--------------------------------------

FUNCTION

A complex program contains a large list of instructions which is not easy to manage, therefore such programs are generally decompose into different modules containing small sets of instruction that perform specific task. These modules are called function.

There are two types of function:

- Library function

- User define function

Library function-The function that already defines or predefines in the language is known as library function.

User defines function- The functions which are designed by user on the basis of requirement of a programmer are known as user defines function.

In c++ three terms are always associated with the function are:

- Function Prototype(declaration)(use semicolon;)
- Function calling (use semicolon;)
- Function definition

Syntax of function declaration: type function name (type parameter name);

Simple program of addition with function:

```
#include<iostream>
#include<conio.h>
Void main()
{
  Clrscr();
  Int a,b,c;
  Int add (int a,int b); // function decleration
  Cout<<"enter the no";
  Cin>>a>>b;
  C=add(a,b); // function calling
  Cout<<c;
  getch();
}
Int add (int x,int y) // function definition
{
  Int z;
  Z=x+y;
  return z;
}
```

What are the different types of parameters?

There are two types of parameters associated with functions. they are:

(a) Actual parameter: The parameters associated with function call are called

actual parameters.

(b) Formal parameter: The parameters associated with the function definition are called formal parameters.

DIFFERENT TYPES OF PARAMETER PASSING.

There are three types of parameter passing in C++. They are:

(a) Call by value: In this method, the actual parameters are copied into the

Formal parameters and the change in the formal parameters do not affect the actual parameter. In this passing only the value so that copy of the value is sent to function, original value will not change.

(b) Call by reference: In this mode of parameter passing instead of passing the

Value to a function, a reference or an alias to the actual parameter is passed. The changes made in the formal parameters are reflected in the actual parameters. In this, address of the value is passed, so the original value will be change .

Program call by value:- swapping program (passing the value)

```
#include<iostream.h>
# include<conio.h>
Void main( )
{
Int x,y;
Void swap (int a,int b);
Cout<<"enter the value ";
Cin>>x>>y;
Cout<<"\n the original value of x and y"<<x<<"and" <<y;
Swap(x,y);
Cout<<"\n after swap value of x and y:"<<x<<"and" <<y;
getch( ) ;
}
Void swap (int a , int b)
{
Int c;
c=b;
b=a;
a=c;
cout<<"\n swapped x and y:"<<a<<b;
```

```
}
```

Call by reference: swapping program(passing the address)

```
#include<iostream.h>
# include<conio.h>
Void main( )
{
Int x,y;
Void swap (int &a,int &b);
Cout<<"enter the value ";
Cin>>x>>y;
Cout<<"\n the original value of x and y"<<x<<"and" <<y;
Swap(x,y);
Cout<<"\n after swap value of x and y:"<<x<<"and"<<y;
getch( ) ;
}
Void swap (int &a , int &b)
{
Int c;
c=b;
b=a;
a=c;
cout<<"\n swapped x and y:"<<a<<b;
}
```

What is Inline function?

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Reason for the need of Inline Function:

- Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call.
- These concepts of function saved program space and memory space are used because the function is stored only in one place and is only executed when it is called.
- This concept of function execution may be time consuming since the registers and other processes must be saved before the function gets called.

- The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function.

The general format of inline function is as follows:

Inline data type function name (arguments)

The keyword inline specified in the above example, designates the function as inline function.

Program of addition of two values with inline function.

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

Inline Int  add (int a,int b); // function decleration with keyword inline.

Void main()
{
  Clrscr();
  Int a,b,c;
  Cout<<"enter the no";
  Cin>>a>>b;
  C=add(a,b); // function calling
  Cout<<c;
  getch();
}

Int  add (int x,int y) // function definition
{
  Int z;
  Z=x+y;
  return z;
}
```

What is a friend function?

Friend function is a special function which can access the private and protected

Members of a class through the object of the same class. Friend functions are not the member functions of a class and they can be declared under any access specify.

Program of friend function -

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

Class car
{
```

```

Private :
Int speed ;
Char color[20];
Public:
Void input( )
{
Cout<<"enter the color";
Cin>>color;
Cout<<"enter the speed";
Cin>>speed;
}
Friend void display (car);
};
Void display(car x)
{
Cout<<"\n the color of the car is :"<<x.color;
Cout<<"\nthe speed of car is"<<x.speed;
}
Void main( )
{
Car c;
c.input( );
display(c);
garch ( );
}

```

FUNCTION OVERLOADING

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. **Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types or arguments.**

Program to find the volume of cube, volume of cylinder, volume of rectangular box, using function overloading?

```

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

```

```

Int volume(int);

float volume(float , int);

long int volume(long int,int,int);

void main( )
{
    Cout<<"volume of cube"<<volume (10);
    Cout<<"volume of cylinder"<<volume (4.5, 5);
    Cout<<"volume of Rectangular box"<<volume (8, 7, 3);
    getch ( );
}

Int volume(int a)
{
    return(a*a*a);
}

float volume(float r,int h)
{
    return (3.14 * r * r * h);
}

long int volume(long int l, int b, int h);
{
    return(l*b*h);
}

```

Operator overloading: Allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability reduce maintenance costs).

Virtual function

- Virtual functions are functions with the same function prototype that are defined throughout a class hierarchy.
- At least the base class occurrence of the function is preceded by the keyword virtual.
- Virtual functions are used to enable generic processing of an entire class hierarchy of objects through a base class pointer.

CLASSES AND OBJECTS

Class: it is an important concept of object oriented programming. It is a user defines data type which contains data member & member function. It is collection of various kind of object. It is define by class keyword. It also an Important feature of object oriented programming language. For ex-fruit is a class And apple, mango, banana are its object.

It contains data members and member function which are declared under class. There are three types of data members declare in class-

- **Public:** In this data members and member function are accessible outside the class.
- **Private:** Data members and member function are not accessible outside the class.
- **Protected:** Data members and member function are only available to derived class.

Program to add two values using class?

```
#include<iostream.h>
#include<conio.h>
Class add
{
Public:
int a,b,c;
Void input ();
Void output ();
};

Void add:: input ()
{
Cout<<"enter the no";
Cin>>a>>b;
}
Void add:: output ()
{
C=a+b;
Cout<<c;
}
Void main ()
{
Clrscr ();
Add d;
d.input ();
d.output ();
getch ();
}
```

DIFFERENCES BETWEEN A STRUCTURE AND A CLASS

Ans: There is no difference in structure and a class except that class members are **private** by default and structure members are **public** by default. And in class a keyword **class** is used and in structure **struct** keyword is used

Structure program:

To print the student name, branch, roll no and age?

```
#include<iostream.h>
#include<conio.h>
Struct student
{
Char name[10];
Char branch[10];
Int rollno;
Int age;
};
Void main( )
{
Student s;
Cout<<"\n enter the name of student";
Cin>>s.name;
Cout<<"\n enter the branch of student";
Cin>>s.branch;
Cout<<"\n enter the age";
Cin>>s.age;
getch();
}
```

CONSTRUCTOR

Constructor :- The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible. This will be explained in later sections of this tutorial.

General Syntax of Constructor

A constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

```
{ arguments};
```

The default constructor for a class X has the form

```
X::X()
```

In the above example, the arguments are optional.

The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the “new” operator.

Any class which does not contain any constructor then compiler from itself supplies a constructor but it is hidden. For programmer these constructors are default

class fact

```
{
}
```

Constructors are automatically called even when not declared, at that time default constructors are called. Default constructors are destroyed as soon as we declare constructor

DEFAULT CONSTRUCTOR:-

A constructor with no argument is said to be default constructor

Example :-

```
class fact
{
    int n;
    long int f;
public:
    fact()
    {
        f=1;
    }
    void getno();
    void calculate( );
    void display( );
};
void fact::getno( )
{
    cout << "enter a no";
    cin >> n;
}
void fact::calculate( )
{
    int i;
    for (i=1; i<=n; i++)
        f= f * i;
}
void fact::display( )
{
    cout << "no=" <<n<<endl;
    cout << "factorial=" <<f;
}
void main( )
```



```

{
    fact obj;
    obj.getno( );
    obj.calculate( );
    obj.display( );
}

```

PARAMETERIZED CONSTRUCTOR

A Constructor with one or more argument(s) is said to be Parameterized Constructor

Example :

```

class Emp
{
    int age;
    char name[20];
    float sal;
public:
    Emp (int, char *, float);
    void show( );
};
Emp : : Emp (int i, char *j, float k)
{
    age =i;
    strcpy (name, j);
    sal=k;
}
void Emp : : show( )
{
    cout << age << " " << name<< ",  ",<<sal;
}

void main( )
{
    Emp e(101,"Amit",6000.00) ;
    e.show( );
    getch( );
}

```

COPY CONSTRUCTOR

This constructor takes one argument, also called one argument constructor. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. The copy constructor allows the programmer to create a new object from an existing one by initialization.

For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2);
```

or

```
Exforsys e3=e2;
```

Both the above formats can be used to invoke a copy constructor.

For Example:

```
#include <iostream>
using namespace std;
class Exforsys
{
    private:
        int a;
    public:
        Exforsys()
        { }
        Exforsys(int w)
        {
            a=w;
        }
        Exforsys(Exforsys& e)
        {
            a=e.a;
            cout << " Example of Copy Constructor";
        }
        void result()
        {
            cout<< a;
        }
};
void main()
{
    Exforsys e1(50);
    Exforsys e3(e1);
    cout<< "e3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference.

DESTRUCTOR

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release **dynamic** allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

```
~ classname();
```

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example:

```
class Exforsys
{
    private:
        ...
    public:
        Exforsys()
        { }
        ~Exforsys()
        { }
}
```

COMPARISION BETWEEN CONSTRUCTOR & DESTRUCTOR

CONSTRUCTOR	DESTRUCTOR
<ol style="list-style-type: none"> 1. are special member fⁿ of a class having same name as that of the class. 2. constructors are automatically called as soon as object of class is created i.e. their calling is implicit 3. constructors can be parameterized. 4. since constructors accepts parameters they can be overloaded & thus a class can have multiple constructors. 5. constructors are called in the orders in which objects are created. 6. constructors can not be inherited. 7. constructors can not be declared as virtual. 	<ol style="list-style-type: none"> 1. are special member function of class having same name as that of the class but preceded with the symbol of tilde 2. a destructor is also automatically called but whenever an object is about to be destructor or goes out of scope thus these calling is implicit. 3. we can not pass parameters to a destructor. 4. as they don't accept parameters we can not overload then, thus a class can not have multiple destructor 5. calling of destructor is always done in reverse or des of the creation of objects. 6. inheriting of destructor is also not possible 7. we can have virtual destructor in a class.

OPERATOR OVERLOADING

It is a mechanism using which a programmer can make built in operator of C++ act on objects of user defined classes much in the same way like they act on variables of primitive data type. Thus we can say by using operator overloading a programmer can enhance the working range of built in operator from primitive type to non primitive type also.

The major advantage offered by operator overloading is the simplicity in readability of the call i.e. function call which are given using operator overloading are much more easy to interpret as compared to conventional function calls.

Two Types of Operator Overloading are

1. Unary Operator Overloading

2. Binary Operator Overloading

Operator Overloading can be done

1. Making use of member function
2. making use of friend function.

Operators Which Can Not Be Overloaded

1. .(dot operator)
2. :: (scope resolution operator)
3. ?: (conditional Operator)
4. *, → Pointer to member operator
5. Size of operator(sizeof)

Syntax:-

<ret-type> operator <op-symbol> (arguments);

friend (ret-type) operator <op-symbol> (<arguments>);

UNARY OPERATOR OVERLOADING

The operation involves one operand. Overloading Of Unary Operator As Member Function Of The class

```
class counter
{
    int count;
public:
    counter( )
    {
        count =0;
    }
    counter (int c)
    {
        count = c
    }
    void operator ++( );
    void show( )
    {
        cout << count <<endl;
    }
};

void counter :: operator ++ ( )
{
    ++ count;
}
```

```

void main( )
{
    counter a = 10;  // Single parameterize constructor
    a.show( );
    ++ a;           // a. operator ++( );
    a. show( );
}

```

OVERLOAD UNARY USING FRIEND FUNCTION

```

class counter
{
    int count;
public:
    friend counter operator ++ (counter &);
    counter
    {
        count =0;
    }
    counter (int i)
    {
        count = i;
    }
    void show( )
    {
        cout << count << endl;
    }
};

counter operator ++(counter &c)
{
    counter temp;
    temp.count=++c.ount;
    return(temp);
}

```

```

void main()
{
    counter c1=10,c2;
    c1.show();
    c2=++c1;
    c1.show();
    c2.show();
    getch();
}

```

OVERLOADING BINARY OPERATOR

In this the operation involve two operand .hence it is called as binary operator overloading

Example :-

$$D3 = D1 + D2;$$

D3 = D1. operator + (D2);

```

class Distance
{
    int feet, inches;
public:
    void get( )
    {
        Count << "enter feet and inches";
        cin >> feet >> inches;
    }
    void show( )
    {
        cout << feet << inches;
    }
    Distance operator + (Distance);
};
Distance Disttadne : : operator + (Distance P)
{
    Distance t;
    t.feet = feet + P. feet
    t. inches = inches + P. inches;
    if (t.inches>= /12)
    {
        t.feet += t.inches/12;
        t.inches % = 12;
    }
    Return (t);
}
void main ( )
{
    Distance D1, D2, D3;
    D1, get( );
    D2, get( );
    D3 = D1+D2;
    D3. show( );
    Getch( );
}

```

OVERLOADING BINARY OPERATOR USING FRIEND FUNCTION

```

class Distance
{
    int feet, inches;
public:
    void get( )
    {
        cout << "enter feet and inches":
        cin >> feet>> inches;
    }
    void show( )

```

```

    {
        cout << feet<< inches;
    }
    friend Distance operator + (Distance, Distance);
};
Distance Operator + (Distance p, Distance Q)
{
    Distance t;
    t.feet = P.feet +Q.feet;
    t.inches = P.inches+ Q.inches;
    if (t. feet>=12)
    {
        t.feet = t.feet +t.inches/12;
        t.inches % = 12;
    }
    Return (t);
}
void main ( )
{
    Distance D1, D2, D3;
    D1.get( );
    D2.get( );
    D3=D1+d2;
    D3.show( );
}

```

C++ Type Conversions

What is type conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

Type conversion is the process using a programmer can convert value from primitive to non primitive and vice versa as well as from one object of class to another object of different class.

Thus type conversion falls in three categories:-

1. Basic to User Defined
2. User Defined to Basic (Primitive)
3. User Defined to User Defined

Conversion Between Basic TO User Defined

```

Constructor (Basic type
{
    // steps to convert basic to user defined
}

```

Conversion Between User Defined TO Basic

```

Operator primitive type of C++ ( ) return type
{

```

```

        // steps to convert basic to user defined
        Return (<basic – val>);
    }

```

Example User To Basic & Basic To User

```

class Meter
{
    float length;

public :
    Meter ( )
    {
        Length = 0.0;
    }
    Meter (float cm)
    {
        Length = CM/100;
    }
    Operator float( )
    {
        Flaot ans= length * 100.0;
        Return (ans);
    }
    void accept meters( )
    {
        cout << “enter length in (int meters)” ;
        cin >> length ;
    }
    void show meter( )
    {
        cout << “In meter = “ << length ;
    }
};

void main ( )
{
    Meter M1 = 250;
    Meter M2;
    M2.accept Meter( );
    float cm = m2; // float CM= M2. operator float( );
    cout << “250 Meters when converted”;
    M1. show Meter( );   M2. show Meter( );
    cout “ when converted to cms =” << cm
}

```


UNIT V: ADVANCED OBJECT ORIENTED PROGRAMMING

Inheritance: Extending classes-Pointers-Virtual functions and polymorphism, File Handling Operations

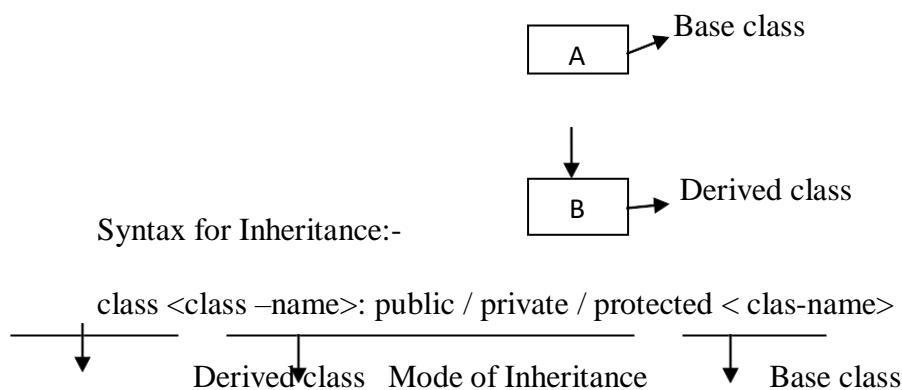
INHERITANCE: Inheritance is the process by which new classes called derived class are created from existing class called base class. The derived class has all the features of base class and the programmer can choose to add new features specific to the newly created derived class.

For example: a programmer can create a base class “fruit” and derived class as “mango”, “apple”, “banana”, “orange”.

In this, each of these derived class has all the features of base class (fruit) with additional attributes or specific to these newly created derived class. In this way mango would have its own features and apple would have their own.

Types of inheritance:**(a) Single-Level Inheritance**

When one class is derived only from one base class then such inheritance is called single-level inheritance. The single-level inheritance is shown below.

**Example:**

```

class Box
{
    int l, b, h;
public:
    void get( )
    {
        cout << "enter l, band";
        cin >> l>>b>>h;
    }
    void show( )
    {
        cout <<l<< " " << b<< " " << h;
    }
};
class carton : public Box
{

```

```

        char type[20];
public:
    void set( )
    {
        cout << "enter material name";
        cin.getline (type, 20);
    }
    void display ( )
    {
        cout << "material =" << type;
    }
};
void main( )
{
    carton obj;
    obj. get( );
    obj. set( );
    obj. show( );
    obj. display( );
}

```

Accessibility in Public Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	yes

Accessibility in Protected Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	no

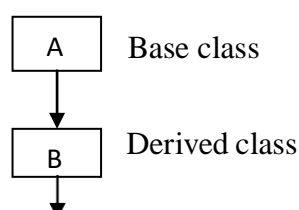
Accessibility in Private Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	no

(b) Multilevel Inheritance

When the single-level inheritance is extended to more levels then it is called multilevel inheritance. In this inheritance one class is derived from another derived class and the level of derivation can be extended to any number of levels.

For example, class C is derived from another derived class B which itself is derived from class A.



C

Derived class

Example :

```

class Num
{
    protected:
        int a, b;
    public:
        void get( )
        {
            cout << "enter a nad b":
            cin >> a >> b;
        }
        void show( )
        {
            cout << "a=" <<a<<endl;
            cout << "b=" <<b<<endl;
        }
};

class AddNum : public Num
{
    protected :
        int c;
    public :
        void set( )
        {
            get( );
        }
        void display( )
        {
            show( );
            cout << "sum=" <<c;
        }
        void add( )
        {
            c=a+b;
        }
};

class DiffNum : public AddNum
{
    int d;
    public:
        void accept( )
        {
            set( );
        }
        void diff( )
        {
            d= a - b;
        }
        void print( )
        {
            display ( );
            cout << "Difference=" <<d;
        }
};

```

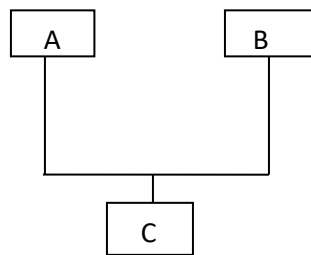
```

    }
};
void main( )
{
    DiffNum obj;
    obj. accept( );
    obj. add( );
    obj. diff( );
    obj. print( );
}

```

(c) Multiple Inheritance

When single class inherits the properties from more than one base class, it is called the multiple inheritance. In other words we can say that multiple inheritance means that one class can have more than one base class. It allows us to combine features of several existing classes into a single class as shown below



Example :

```

class base1
{
    protected:
        int a;
    public:
        void get( )
        {
            cout << "enter a =";
            cin >>a;
        }
        void show( )
        {
            cout <<a << endl;
        }
};
class base2
{
    protected:
        int b;
    public:
        void set( )
        {
            cout << "enter b=";

```

```

        cin >> b;
    }
    void display( )
    {
        cout <<b << endl;
    }
};

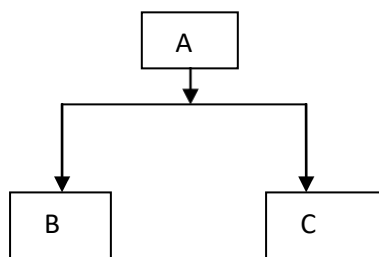
class drv : public base1, public base2
{
    int c;
public :
    void accept ( )
    {
        get();
        set( );
    }
    void add ( )
    {
        c = a+b;
    }
    void print( )
    {
        show( );
        display( );
        cout << "sem =" <<c;
    }
};

void main( )
{
    drv obj;
    obj. accept();
    obj. add();
    obj. print( );
}

```

(d) Hierarchical Inheritance

When many subclasses inherit properties from a single base class, it is called as hierarchical inheritance. The base class contains the features that are common to the subclass and a subclass can inherit all or some of the features from the base class as shown below



```

class Num
{
    protected :
        int a, b;
    public :
        Num (int i, int j)
        {
            a = i;
            b = j;
        }
        void show( )
        {
            cout << "a =" << a;
            cout << "b=" << b;
        }
};

class AddNum : public Num
{
    int c;
public:
    AddNum (int i, int j) : Num (i, j)
    {
        c=a+b;
    }
    void show( )
    {
        Num : : show( );
        cout << " sum =" <<c;
    }
};

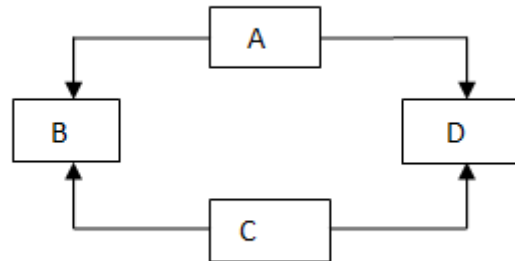
class Diffnum : public Num
{
    int d;
public :
    DiffNum (int x, int y) : Num (x, y)
    {
        d= a - b;
    }
    void show( )
    {
        Num : : show( );
        cout << " Difference =" << d;
    }
};

void main( )
{
    AddNum addobj (10, 20);
    DiffNUM diffobj (30, 70);
    addobj. show( );
    diffobj. Show( );
}

```

(e) Hybrid Inheritance

It is a combination of multiple inheritances and the hybrid inheritance. In hybrid Inheritance a class inherits from a multiple base class which itself inherits from a single base class. This form of inheritance is known as hybrid inheritance. It is shown below



```

class Base
{
    public:
        int a;
};
class drv1 : virtual public Base
{
    public:
        int b;
};
class drv2: virtual public Base
{
    public:
        int c;
};
class drv3 : public drv1, public drv2
{
    public:
        int d;
};
void main( )
{
    drv3 obj;
    obj. a =10;
    obj. b = 20
    obj. c =30;
    obj.d= obj a + obj.b + obj. c;
    cout << "sum =" << obj.d;
}
  
```

POINTER

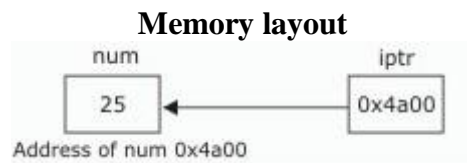
A pointer is a variable that holds a memory address, usually the location of another variable in memory.

Defining a Pointer Variable

```
int *iptr;
iptr can hold the address of an int
```

Pointer Variables Assignment:

```
int num = 25;
int *iptr;
iptr = &num;
```



To access num using iptr and indirection operator *

```
cout << iptr;    // prints 0x4a00
cout << *iptr;   // prints 25
```

Similarly, following declaration shows:

```
char *cptr;
float *fptr;
cptr is a pointer to character and fptr is a pointer to float value.
```

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators ++, --
- Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=

Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

If “p” is a character pointer then “p++” will increment “p” by 1 byte.

If “p” were an integer pointer its value on “p++” would be incremented by 4 bytes.

Pointers and Arrays

Array name is base address of array

```
int vals[] = {4, 7, 11};
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
```

Lets takes an example:

```
int arr[]={4,7,11};
int *ptr = arr;
```


What is ptr + 1?

It means (address in ptr) + (1 * size of an int)

```
cout << *(ptr+1); // displays 7
```

```
cout << *(ptr+2); // displays 11
```

Array Access

Array notation arr[i] is equivalent to the pointer notation *(arr + i)

Assume the variable definitions

```
int arr[]={4,7,11};
```

```
int *ptr = arr;
```

Examples of use of ++ and --

```
ptr++; // points at 7
```

```
ptr--; // now points at 4
```

Character Pointers and Strings

Initialize to a character string.

```
char* a = "Hello";
```

a is pointer to the memory location where „H“ is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

```
char* a = "Hello";
```

a gives address of „H“

*a gives „H“

a[0] gives „H“

a++ gives address of „e“

*a++ gives „e“

Pointers as Function Parameters

A pointer can be a parameter. It works like a reference parameter to allow change to argument from within function

```
#include<iostream>
```

```
using namespace std;
```

```
void swap(int *, int *);
```

```
int main()
```

```
{
    int a=10,b=20;
    swap(&a, &b);
    cout<<a<<" "<<b;
    return 0;
}
```

```
void swap(int *x, int *y)
```

```
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

output:

20 10

Pointers to Constants and Constant Pointers

Pointer to a constant: cannot change the value that is pointed at

Constant pointer: address in pointer cannot change once pointer is initialized

Pointers to Structures

We can create pointers to structure variables

```
struct Student {int rollno; float fees;};
Student stu1;
Student *stuPtr = &stu1;
(*stuPtr).rollno= 104;
```

-or-

Use the form ptr->member:

```
stuPtr->rollno = 104;
```

Static allocation of memory

In the static memory allocation, the amount of memory to be allocated is predicted and preknown. This memory is allocated during the compilation itself. All the declared variables declared normally, are allocated memory statically.

Dynamic allocation of memory

In the dynamic memory allocation, the amount of memory to be allocated is not known. This memory is allocated during run-time as and when required. The memory is dynamically allocated using new operator.

We can allocate storage for a variable while program is running by using new operator

To allocate memory of type integer

```
int *iptr=new int;
```

To allocate array

```
double *dptr = new double[25];
```

To allocate dynamic structure variables or objects

```
Student sptr = new Student; //Student is tag name of structure
```

Releasing Dynamic Memory

Use delete to free dynamic memory

```
delete iptr;
```

To free dynamic array memory

```
delete [] dptr;
```

To free dynamic structure

```
delete Student;
```

Memory Leak

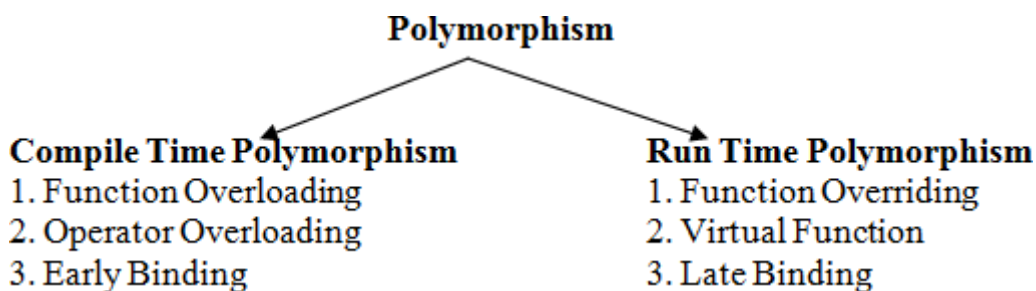
If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks. These orphaned memory blocks when increase in number, bring adverse effect on the system. This situation is called memory leak

Self Referential Structure

The self referential structures are structures that include an element that is a pointer to another structure of the same type.

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

POLYMORPHISM



The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

```
class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    int area()
    {
        cout << "Parent class area :" << endl;
    }
}
```

```

        return 0;
    }
};
class Rectangle: public Shape{
public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Rectangle class area :" <<endl;
        return (width * height);
    }
};
class Triangle: public Shape{
public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Triangle class area :" <<endl;
        return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);

    // store the address of Rectangle
    shape = &rec;
    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;
    // call triangle area.
    shape->area();

    return 0;
}

```

When the above code is compiled and executed, it produces the following result:

```

Parent class area
Parent class area

```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```

class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    virtual int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};

```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

Rectangle class area
Triangle class area

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

Each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

OVERLOADING

The function overriding is a term which is used when a derived class contains a function with the same prototype as its base class. In other words, fn provided by base class has same prototype in derived class but in different body.

Overriding	Overloading
Scope must be different By the classes related by Inheritance	always in same class means at single level
Prototype of functions Must be same .	prototype must be different

Virtual Function:

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

virtual void f() = 0;

We can change the virtual function area() in the base class to the following:

```
class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Abstract Class

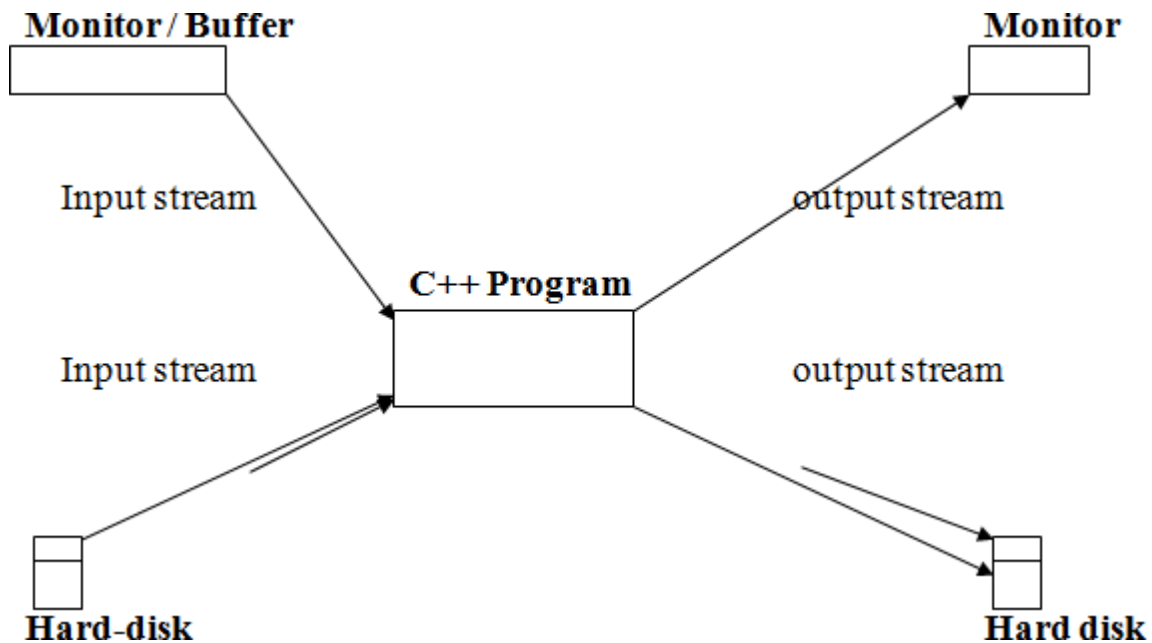
Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

C++ FILES AND STREAMS

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.



This requires another standard C++ library called **fstream**, which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included in your C++ source file.

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing and **ifstream** object is used to open a file for reading purpose only.

Following is the standard syntax for `open()` function, which is a member of **fstream**, **ifstream**, and **ofstream** objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File:

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File:

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
#include <iostream>
using namespace std;
```

```
int main ()
```



```

{

char data[100];

// open a file in write mode.
ofstream outfile;
outfile.open("afile.dat");

cout << "Writing to the file" << endl;
cout << "Enter your name: ";
cin.getline(data, 100);

// write inputted data into the file.
outfile << data << endl;

cout << "Enter your age: ";
cin >> data;
cin.ignore();

// again write inputted data into the file.
outfile << data << endl;

// close the opened file.
outfile.close();

// open a file in read mode.
ifstream infile;
infile.open("afile.dat");

cout << "Reading from the file" << endl;
infile >> data;

// write the data at the screen.
cout << data << endl;

// again read the data from the file and display it.
infile >> data;
cout << data << endl;

// close the opened file.
infile.close();

return 0;
}

```

When the above code is compiled and executed, it produces the following sample input and output:

```

$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara 9

```

Above examples make use of additional functions from cin object, like `getline()` function to read the line from outside and `ignore()` function to ignore the extra characters left by previous read statement.

File Position Pointers:

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to `seekg` and `seekp` normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );
```

```
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
```

```
// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );
```

```
// position at end of fileObject
fileObject.seekg( 0, ios::end );
```