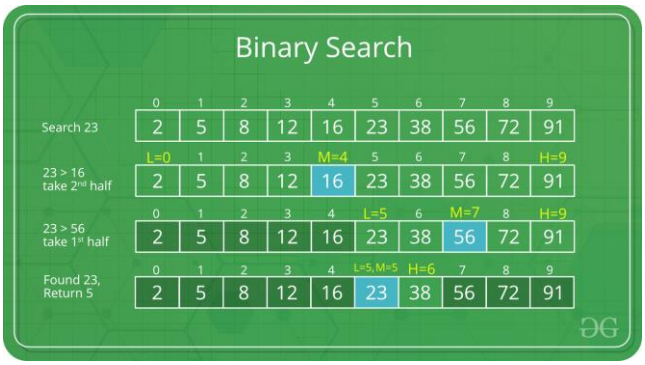
DATA STRUCTURES

10 - MARKS

1. Describe binary search with an example and give its algorithm.

BINARY SEARCH

1. The binary search algorithm is one of the most efficient searching techniques, which requires the list to be sorted in ascending order.
2. To search for an element in the list, the binary search algorithm splits the list and locates the middle element of the list. It is then compared with the search element.
3. If the search element is less than the middle element, the first part of the list is searched else the second part of the list is searched.
4. The algorithm again reduces the list into two halves locate the middle element and compares with the search element. If the search element is less than the middle element the first part of the list is searched.
5. This process continues until the search element is equal to the middle element or the list consists of only one element that is not equal to the search element.



Binary Search Algorithm:

procedure BINSRCH(F, n, i, k)
//Search an ordered sequential file F with records R1, ...,Rn and the keys K1 <= K2 <= ... <= Kn for a record Ri such that Ki = K; i = 0 if there is no such record else Ki = K. Throughout the algorithm, l is the smallest index such that Kl may be K and u the largest index such that Ku may be K//
  l <-- 1; u <-- n
  while *l* <= u **do**
  m <-- [(**l** + u)/2] //compute index of middle record//
  case
  :K > K_m: ***l*** <-- m + 1 //look in upper half//
  :K = K_m: i <-- m; **return**
  :K < K_m: u <-- m - 1 //look in lower half//
  end
  end i <-- 0 //no record with key K//
end BINSRCH
Time required by a binary search to search a list on n records is O(logn)

Best case for binary search happen if the search key is found in the middle array O(1).

Worse case for binary search happens if the search key is not in the list or the searching size equal to 1. The searching time for worst case is O(log2n).

2. Explain about the different types of queue and describe the various operations in queue with its algorithm.

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

Simple Queue

Queue is an ordered collection of elements in which insertions are restricted to one end called the rear end and deletions are restricted to the other end called the front end. Queues are also referred as First-In-First-Out (FIFO).

Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle. The last position is connected back to the first position to make a circle. Circular Queue is used in memory management and scheduling process. It is also called 'Ring Buffer'.

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority.
Insert - inserts an element at the end of the list called the rear. Delete Min - Finds, returns and removes the minimum element in the priority Queue.

Types of priority queues

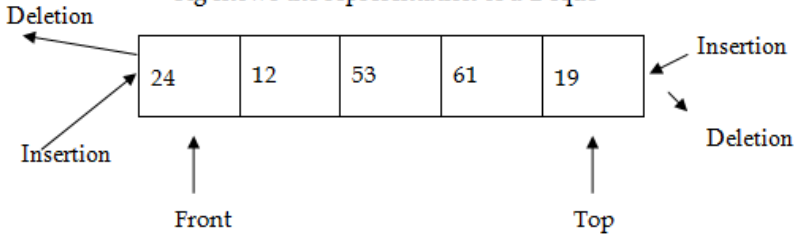
Ascending Priority Queue : It is a collection of items into which items can be inserted arbitrarily and from which the smallest item can be removed.

Descending Priority Queue : It is a collection of items into which items can be inserted arbitrarily and from which the largest item can be removed.

Double Ended Queue

The Deque is a short form of Double ended queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be a made elsewhere in the list. Thus, a Deque is a generalization of both a stack and a queue.

Fig shows the representation of a Deque



There are two variations of a Deque. These are:
Input – Restricted Deque
Output – Restricted Deque

An input restricted Deque restricts the insertion of elements at one end only, but the deletion of elements can be done at both the ends of a Deque.

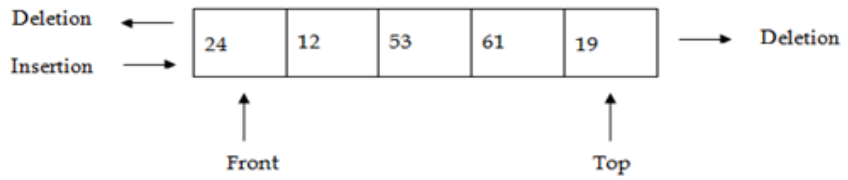


Fig shows the representation of input – Restricted Deque

An output restricted Deque restricts the deletion of elements at one end only, but the insertion of elements can be done at both the ends of a Deque.

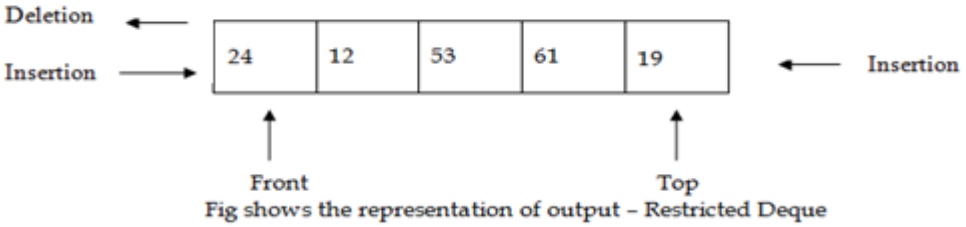


Fig shows the representation of output – Restricted Deque

Operations involved in a queue:

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

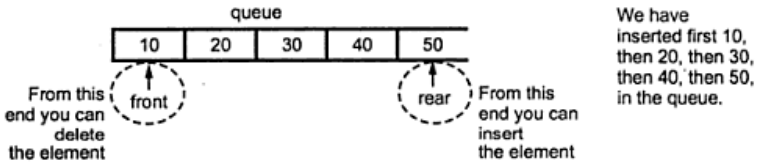
INSERT OPERATION:

- Insertion of any element takes place from the rear end.
- An attempt to push an element onto the queue, when the array is full, causes an overflow.

Insert operation involves:

1. Check whether the array is full before attempting to insert another element.
2. If so halt execution.
3. Increment the rear pointer.

Example:



Implementation in c

```
void addqueue(int q[], int n)

{

        if(rear == n-1)

                printf("Queue full");

        } else{

                rear = rear + 1;

                q[rear] = element;

        } }
```

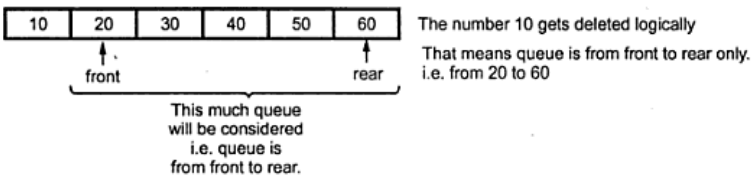
DELETE OPERATION:

- Delete operation removes an element from the queue.
- It takes place at the front end always. An attempt to remove an element from the stack, when the array is empty, causes an underflow.

Delete operation involves:

1. Check whether the array is empty before attempting to remove another element.
2. If so halt execution.
3. Decrement the front pointer.
4. Remove the element from the point pointed by the front pointer of the queue.

Example:



Implementation in c

```
void deletequeue()
{     if (front == rear)
        {          printf("Queue empty");        }
      else {
                  front = front + 1;
                  item = q[front];
                  printf("Deleted element is %d", item);
}     }
```

3. Discuss the insertion and deletion operations in a doubly linked list with an example along with their respective algorithms.

Following are the basic operations in doubly linked list.

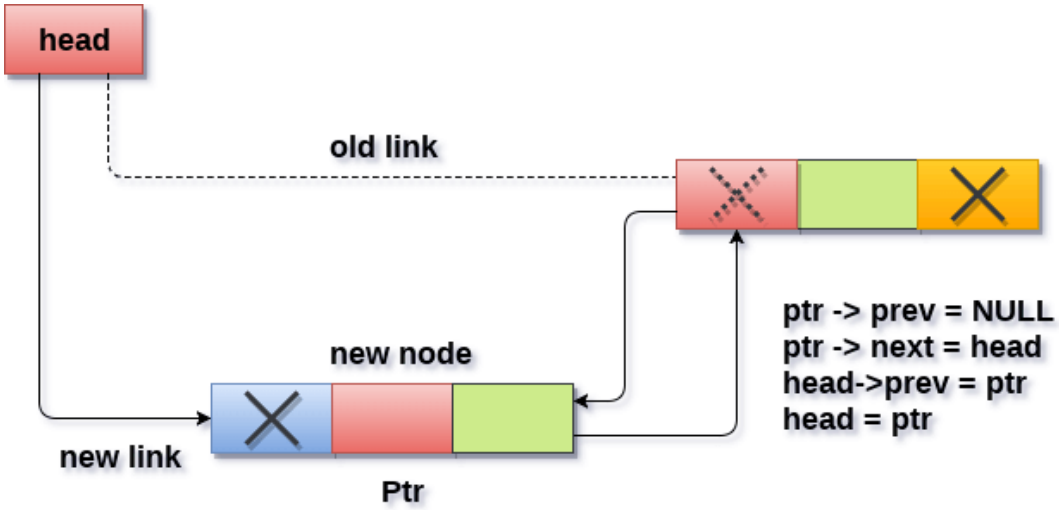
1. Insertion − Adds an element at the beginning of the list.
2. Deletion − Deletes an element at the beginning of the list.
3. Insert Last − Adds an element at the end of the list.
4. Delete Last − Deletes an element from the end of the list.
5. Insert After − Adds an element after an item of the list.
6. Delete − Deletes an element from the list using the key.

Insertion at the Beginning

In this operation, we create a new node with three compartments, one containing the data, the others containing the address of its previous and next nodes in the list. This new node is inserted at the beginning of the list.

Steps

1. START
2. Create a new node with three variables: prev, data, next.
3. Store the new data in the data variable
4. If the list is empty, make the new node as head.
5. Otherwise, link the address of the existing first node to the next variable of the new node, and assign null to the prev variable.
6. Point the head to the new node.
7. END



Insertion into doubly linked list at beginning

ALGORITHM

Insertion at beginning:

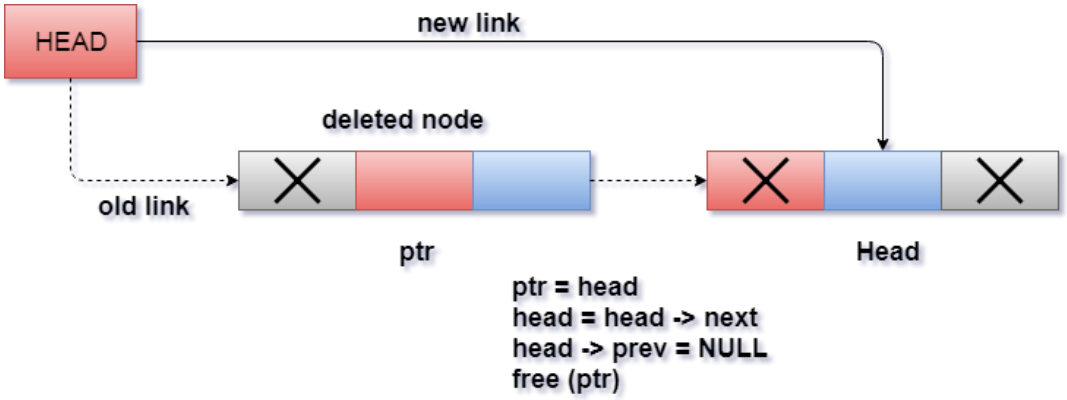
```
Void insertbeg()
{
p=(node*)malloc(sizeof(node));
p->data=x
p->prev=NULL
p->next=start;
if( start!=NULL)
        start ->prev=p;
}
```

Deletion at the Beginning

This deletion operation deletes the existing first nodes in the doubly linked list. The head is shifted to the next node and the link is removed.

Steps

1. START
2. Check the status of the doubly linked list
3. If the list is empty, deletion is not possible
4. If the list is not empty, the head pointer is shifted to the next node.
5. END

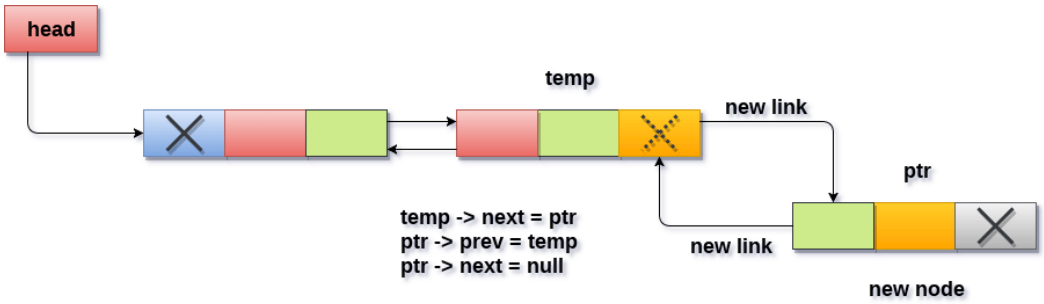


Deletion in doubly linked list from beginning

Insertion at the End

In this insertion operation, the new input node is added at the end of the doubly linked list; if the list is not empty. The head will be pointed to the new node, if the list is empty.
Steps

1. START
2. If the list is empty, add the node to the list and point the head to it.
3. If the list is not empty, find the last node of the list.
4. Create a link between the last node in the list and the new node.
5. The new node will point to NULL as it is the new last node.
6. END



Insertion into doubly linked list at the end

ALGORITHM

Insertion at end:

```
Void insertend()
{
        p=(node*)malloc(sizeof(node));
        p->data=x
```

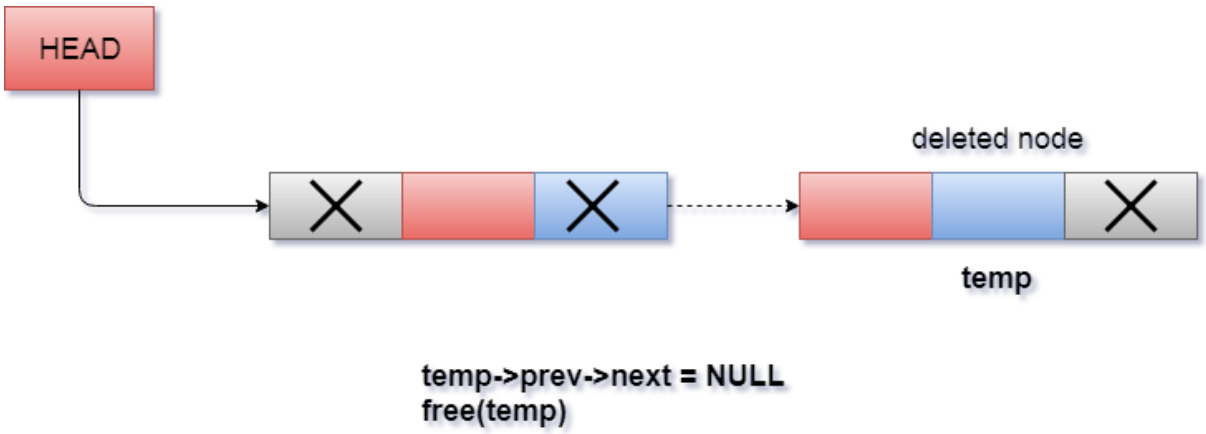
```
        p->next=NULL;
q=start;
while(q->next!=NULL)
        q=q->next;
p->prev=q;
q->next=p
}
```

Deletion at the End

In this deletion operation, the last node is deleted at the end of the doubly linked list; if the list is not empty.

Steps

1. START
2. If the list is not empty, delete the last node in the list.
3. Now the last node will point to NULL as it is the new last node.
4. END



Deletion in doubly linked list at the end

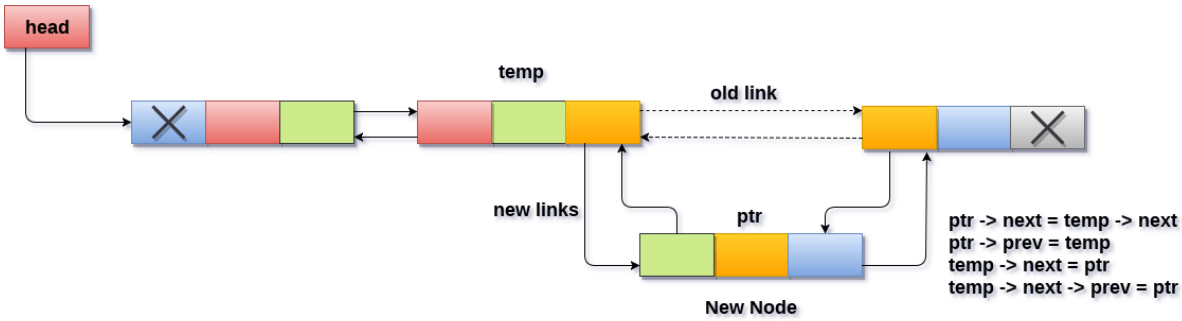
Insertion After a Node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Steps

1. START
2. Allocate the memory for the new node.

3. Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
4. The temp would point to the specified node at the end of the for loop.
5. The new node needs to be inserted after this node therefore we need to make a pointer adjustments here.
6. Make the next pointer of ptr point to the next node of temp.
7. Make the prev of the new node ptr point to temp.
8. Make the next pointer of temp point to the new node ptr.
9. Make the previous pointer of the next node of temp point to the new node.
10. END



Insertion into doubly linked list after specified node

ALGORITHM

Insertion at middle:

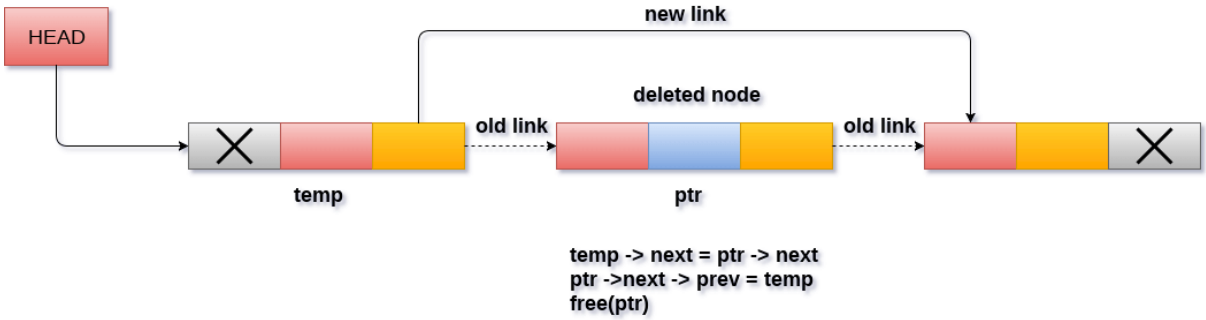
```
Void insertmid()
{
p=(node*)malloc(sizeof(node));
p->data=3;
p->next=q->next;
p->prev=q;
q->next=p;
if(p->next!=NULL)
        p->next->prev=p;
}
```

Deletion After a Node

In order to delete a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Steps

1. START
2. Copy the head pointer into a temporary pointer temp..
3. Traverse the list until we find the desired data value.
4. Check if this is the last node of the list. If it is so then we can't perform deletion.
5. Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
6. Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.
7. END

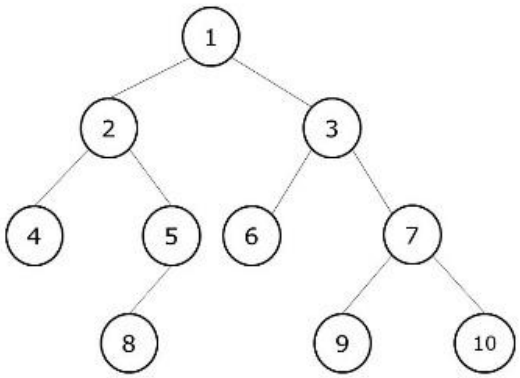


Deletion of a specified node in doubly linked list

DELETION OF ANY NODE :

```
void Delete (int X, List L)
{
        position P;
        P = Find (X, L);
        If ( IsLast (P, L))
        {
                Temp = P;
                P →Blink →Flink = NULL;
                free (Temp);
        }
        else
        {
                Temp = P;
                P →Blink→ Flink = P→Flink;
                P →Flink →Blink = P→Blink;
                free (Temp);
        }
}
```

4. Explain about the binary tree traversal techniques with its algorithm for the given tree.



Traversing a tree means processing the tree such that each node is visited only one. Traversal of a binary tree is useful in many applications. For example, in searching for particulars nodes compilers commonly build a binary trees in the process of scanning, parsing, generating code and evaluation of arithmetic expression.

Let T be a binary tree, there are a number of a different ways to proceed. The methods differ primarily in the order in which they visit the nodes.

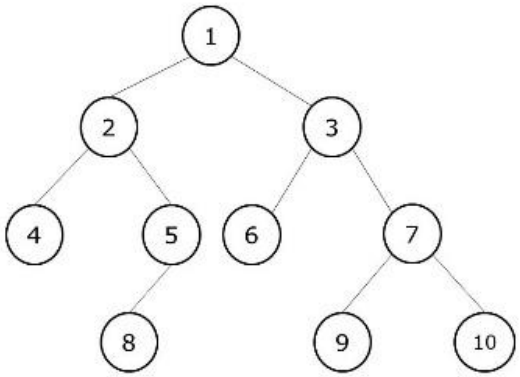
The three different traversals of T are

- Inorder traversal
- Preorder traversal
- Postorder traversal

1. Inorder traversal

In this Traversal ,the left sub-tree of the root is visited, then the root node and after that the right sub tree of the root node is visited. visiting both the sub tree is in the same fashion as the tree itself.

- Left subtree
- Root
- Right subtree



Solution : 4 -> 2 -> 8 -> 5 -> 1 -> 6 -> 3 -> 9 -> 7 -> 10

ALGORITHM:

Ptr=root

If(ptr≠NULL)then

Inorder(ptr->LC)

Visit(ptr)

Inorder(ptr->RC)

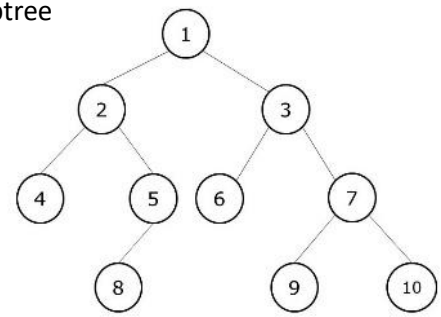
Endif

Stop

2. Preorder traversal

In this traversal, the root is visited first, then the left sub-tree in preorder fashion, and then the right sub-tree in preorder fashion. such a traversal can be defined as follow:

- Root
- Left subtree
- Right subtree



Solution : 1 -> 2 -> 4 -> 5 -> 8 -> 3 -> 6 -> 7 -> 9 -> 10

ALGORITHM:

Ptr=root

If(ptr≠NULL)then

Visit(ptr)

Preorder(ptr->LC)

Preorder(ptr->RC)

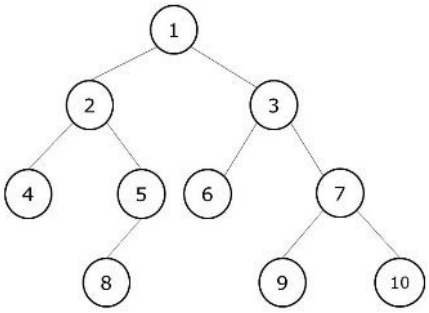
Endif

stop

3. Postorder traversal

In this Traversal, the left sub-tree of the root is visited, then the right sub tree of the root node is visited, after that root is visited last. visiting both the sub tree is in the same fashion as the tree itself.

- Left subtree
- Right subtree
- Root



Solution : 4 -> 8 -> 5 -> 2 -> 6 -> 9 -> 10 -> 7 -> 3 -> 1

ALGORITHM:

Ptr=root

If(ptr≠NULL)then

Postorder(ptr->LC)

Postorder(ptr->RC)

Visit(ptr)

Endif

Stop

5. Discuss in detail various graph traversal techniques.

There are two types of graph traversals

- Breadth First Search (BFS)
- Depth First Search (DFS)

1. Breadth First Search (BFS)

In BFS, we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.

This traversal is very similar to the level-by-level traversal of a tree. Here, any vertex in the level I will be visited only after the visit of all the vertices in its preceding level, that is at i-1.

A Queue can be used to maintain the track of all paths in breadth first search

Algorithm BFS_LL

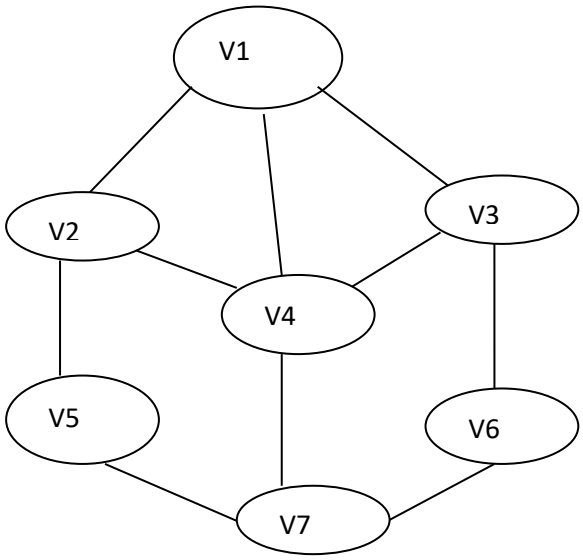
Input: V is the starting vertex

Output: A list VISIT giving the order of visit of vertices during the traversal

Algorithm:

```
If (GPTR = NULL) Then
        Print "Graph is empty"
        Exit
EndIf
u=V
OPENQ.ENQUEUE(u)
While(OPENQ.STATUS( )≠ EMPTY)do
        U=OPENQ.DEQUEUE( )
        If (search_SL(VISIT,u) = FALSE) then
                InsertEnd_SL(VISIT,u)
                Ptr = Gptr[u]
                While (ptr->LINK≠ NULL) do
                        Vptr = ptr ->LINK
                        OPENQ.ENQUEUE(vptr->LABEL)
                EndWhile
        EndIf
EndWhile
Return (VISIT)
Stop
```

Example



1) Let us select starting vertex as v1

The vertices adjacent to v1 are v2,v3,v4, enqueue these adjacent vertices onto the queue

V2	V3	V4			

Visited vertices: v1

2) De queue the Queue for the next vertex to be visited, here v2 is removed and its adjacent vertices v5 and v4 are inserted onto the queue

V3	V4	V5			

Visited vertices: v1,v2

3) De queue the Queue for the next vertex to be visited, here v3 is removed and its adjacent vertices inserted onto the queue

V4	V5	V6			

Visited vertices: v1,v2,v3

4) De queue the Queue for the next vertex to be visited, here v4 is removed and its adjacent vertices inserted onto the queue

V5	V6	V7			

Visited vertices: v1,v2,v3,v4

5) De queue the Queue for the next vertex to be visited, here v5 is removed and its adjacent vertices inserted onto the queue

V6	V7				

Visited vertices: v1,v2,v3,v4,v5

6) De queue the Queue for the next vertex to be visited, here v6 is removed and its adjacent vertices inserted onto the queue

V7					

Visited vertices: v1,v2,v3,v4,v5,v6

7) De queue the Queue for the next vertex to be visited, here v6 is removed and its adjacent vertices inserted onto the queue

Visited vertices: v1,v2,v3,v4,v5,v6,v7

The output of Breadth First Search(BFS) is **v1,v2,v3,v4,v5,v6**

2. Depth First Search (DFS)

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

Approach behind DFS traversal, starting from the given node, this traversal visits all the nodes up to the deepest level and so on

(i) visit the vertex **v** then the vertex immediate adjacent to **V**, let it be $\mathbf{V_x}$.

(ii) if $\mathbf{V_x}$ has an immediate adjacent, say, v_y then visit it and so on, till there is a 'dead end'. Thus it result in path, **P** ($v\text{-}v_x\text{-}v_y$....)

Dead end means a vertex which does not have an immediate adjacent or its immediate adjacent, already been visited.

(iii) After coming to 'dead end', we backtrack along **P** to **V** to see if it has another adjacent vertex other than v_x ant then continue the same from it else from the adjacent of the adjacent(which is not visited earlier)

A stack can be used to maintain the track of all paths from any vertex so as to help backtracking.

Algorithm

Input: v, the starting vertex

Output: A list VIST giving the order of vertices during traversal

```
If Gptr = NULL then
        Print "Graph is empty"
        Exit
EndIf
U= V
Open.Push(u)
While (open.Top ≠ NULL) do
```

```
        u= OPEN.POP()
        If (search_SL(VISIT,u) = FALSE) then
                InsertEnd_SL(VISIT, u)
                Ptr = GPTR[u]
                While(ptr -> LINK ≠ NULL) do
                        vptr = ptr->LINK
                        OPEN.PUSH(vptr->LABEL)
                EndWhile
        EndIF
EndWhile
Return(VISIT)
Stop
```

Example

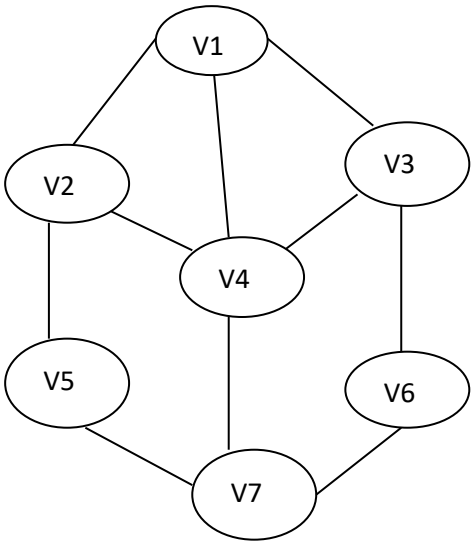


Figure: Graph

1) Let us select start with vertex v1

The vertices adjacent to v1 are v2, v3, v4, push these adjacent vertices on to the stack

V4
V3
V2

Visited vertices: v1

2) Pop the stack for the next vertex to be visited, here v4 is popped and its adjacent vertices v1 is already visited, so v2, v3 and v7 are pushed onto the stack

V7
V3
V2
V3
V2

Visited vertices: v1, v4

3) Pop the stack for the next vertex to be visited, here v7 is popped and its adjacent vertices V4 is already visited, so v5 and v6 pushed on to the stack.

V6
V5
V3
V2
V3
V2

Visited vertices: v1, v4, v7

4) Pop the stack for the next vertex to be visited, here v6 is popped and its adjacent vertices v7 is already visited, so v3 pushed on to the stack

V3
V5
V3
V2
V3
V2

Visited vertices: v1,v4,v7,v6

5) Pop the stack for the next vertex to be visited, here v3 is popped and its adjacent vertices v1, v4 and v7 are already visited, so no elements pushed on to the stack

V5
V3
V2
V3
V2

Visited vertices: v1,v4,v7,v6,v3

6) Pop the stack for the next vertex to be visited, here v5 is popped and its adjacent vertices v7 is already visited, so v2 is pushed on to the stack

V2
V3
V2
V3
V2

Visited vertices: v1,v4,v7,v6,v3,v5

7) Pop the stack for the next vertex to be visited, here v2 is popped and its adjacent vertices v1, v4 and v5 are already visited, so no elements pushed on to the stack

V3
V2
V3
V2

Visited vertices: v1,v4,v7,v6,v5,v3,v2

The remaining vertices in stack are all already visited

The output of depth first traversal is **v1,v4,v7,v6,v5,v3,v2**

No.	**BFS**	**DFS**
1.	Backtracking is not possible	Backtracking is possible
2.	Vertices to be explored are organized	Vertices from which exploration is incomplete
3.	Search is done in same level order	Search is done in depth order