EX. NO: 1 a)	IMPLEMENTATION OF NON RECURSIVE LINEAR SEARCH
DATE:	

AIM

To Write a C Program to implement non-recursive linear search.

ALGORITHM

```
Algorithm linearsearch(a,n)
{
        read  key;
        for (i=1 to
        n)
        {
                if (a[i]==key)
                {
                        write"key exist in the i th position";
                }
        }
        write"key does not exist"
}
```

RESULT

Thus, the program for non-recursive linear search is executed and verified successfully.

EX. NO: 1 b)	IMPLEMENTATION OF RECURSIVE LINEAR SEARCH
DATE:	

AIM

To Write a C Program to implement recursive linear search.

ALGORITHM

```
Algorithm linearsearch(a,n)
{
        read  key;
        for (i=1 to
        n)
        {
                if (a[i]==key)
                {
                        write"key exist in the i th position";
                }
        }
        write"key does not exist"
}
```

RESULT

Thus, the program for recursive linear search is executed and verified successfully.

EX. NO: 2 a)	IMPLEMENTATION OF NON RECURSIVE BINARY SEARCH
DATE:	

AIM

To Write a C Program to implement non-recursive Binary search.

ALGORITHM

```
Algorithm binarysearch(a,n)
{
        read key;
        mid =
        (low+high)/2;
        while
        (low<=high)
        {
                if(key==a[mid])
                {
                        Write "key exist in mid position";
                }
                  else if(key<a[mid])
                {
                     high = mid – 1;
                }
                else
                {
                      Low = mid – 1;
                }
           }
```

RESULT

Thus, the program for non-recursive binary search is executed and verified successfully.

EX. NO: 2 b)	IMPLEMENTATION OF RECURSIVE BINARY SEARCH
DATE:	

AIM

To Write a C Program to implement recursive Binary search.

ALGORITHM

```
Algorithm binarysearch(a,n)
{
        read key;
        mid =
        (low+high)/2;
        while
        (low<=high)
        {
                if(key==a[mid])
                {
                        Write "key exist in mid position";
                }
                 else if(key<a[mid])
                {
                    high = mid – 1;
                }
                else
                {
                     Low = mid – 1;
                }
            }
```

RESULT

Thus, the program for recursive binary search is executed and verified successfully.

EX. NO: 3 a)	IMPLEMENTATION OF BUBBLE SORT
DATE:	

AIM

To Write a C Program to sort the given set of elements using Bubble Sort.

ALGORITHM

```
Algorithm bubblesort(a,n)
{
   for(i=1 to n) do
    {
      for(j=1 to n) do
       {
          if(a[j]>a[j+1]) then
           {
             temp=a[j];
             a[j]=a[j+1];
             a[j+1]=temp;
           }
       }
    }
}
```

RESULT

Thus, the program for sorting the elements using bubble sort is executed and verified successfully.

EX. NO: 3 b)	IMPLEMENTATION OF SELECTION SORT
DATE:	

AIM

To Write a C Program to sort the given set of elements using Selection Sort.

ALGORITHM

```
Algorithm selectionsort(a,n)
{
        for(i=1 to n) do
        {
                min=i;
                for(j=i+1 to n) do
                {
                        if(a[j]<a[min]) then
                        {
                                min=j;
                        }
                }
                if(min!=i) then
                {
                        t=a[i];
                        a[i]=a[mi
                        n];
                        a[min]=t;
                }
        }
}
```

RESULT

Thus, the program for sorting the elements using selection sort is executed and verified successfully.

EX. NO: 3 c)	IMPLEMENTATION OF INSERTION SORT
DATE:	

AIM

To Write a C Program to sort the given set of elements using Insertion Sort.

ALGORITHM

```
Algorithm selectionsort(a,n)
{
        for(i=1 to n) do
        {
                min=i;
                for(j=i+1 to n) do
                {
                        if(a[j]<a[min]) then
                        {
                                min=j;
                        }
                }
                if(min!=i) then
                {
                        t=a[i];
                        a[i]=a[mi
                        n];
                        a[min]=t;
                }
        }
}
```

RESULT

Thus, the program for sorting the elements using insertion sort is executed and verified successfully.

EX. NO: 3 d)	IMPLEMENTATION OF SHELL SORT
DATE:	

AIM

To Write a C Program to sort the given set of elements using Shell Sort.

ALGORITHM

```
Algorithm Shellsort(ElementTypeA[ ], int N)
  {
   int I, j, increment;
   ElementType Temp;
   For(increment=N/2;increment>0;increment++)
   For(i=increment; i<N; i++)
   {
     Temp=A[i];
     For(j=I;j>=increment;j=increment)
         If(Temp<A[j-increment])
               A[j]=A[j-increment];
         Else
               Break;
               A[j]=Temp;
    }
 }
```

RESULT

Thus, the program for sorting the elements using shell sort is executed and verified successfully.

EX. NO: 3 e)	IMPLEMENTATION OF HEAP SORT
DATE:	

AIM

To Write a C Program to sort the given set of elements using Heap Sort.

ALGORITHM

- An array A with N elements is given.
- This algorithm sorts the element of A.

1. [Build a heap A ,using a procedure 1]
   Repeat for J=1 to N-1
   Call INSHEAP(A, J, A[J+1])
2. [sort A by repeatedly deleting the root of H, using procedure 2]
   Repeat while N>1:
   Call DELHEAP(A , N,VALUE)
   Set A[n+1]:=value

Exit

PROGRAM

RESULT

Thus, the program for sorting the elements using heap sort is executed and verified successfully.

EX. NO: 4 a)	ARRAY IMPLEMENTATION OF STACK ADT
DATE:	

AIM

To Write a C Program to implement the stack operations using array.

ALGORITHM

```
PUSH()
{
        if(front==(MAX-1)) // stack overflow condition
        else
        {
                // get the input item
                front++;
        }
        stack_arr[front]=pushed_item;
}

POP()
{
        if(front==-1) // stack underflow condition
        else
        {
                // item deleted
                front--;
        }
}

DISPLAY()
{
        if(front==-1) // STACK EMPTY CONDITION
        else
        {
                for(i=front;i>=0;i--)
                Display stack_arr[i]);
        }
}
```

RESULT

Thus, the program for implementation of stack operations using array is executed and verified successfully.

EX. NO: 4 b)	ARRAY IMPLEMENTATION OF QUEUE ADT
DATE:	

AIM

To Write a C Program to implement the queue operations using array.

ALGORITHM

```
INSERT ( )
// pushed _ item is the value to be inserted.
{
        if(rear==(MAX-1))
        // queue overflow condition
else
        {
        if(front==-1)
        front++;
        // input the ITEM into queue array
        rear++;
        queue_arr[rear]=pushed_item;
        }
}

DELETE ( )
{
        if(front>rear)
        // Queue underflow condition
else
        {
        // element removed
        front++;
        }
}

DISPLAY()
//QUEUE_ARR is an array with N locations.
//FRONT and REAR points to the front and rear of the QUEUE
{
        if(front>rear)
        // Queue Underflow
else
        {
        for(i=front;i<=rear;i++)
        Display queue_arr[i]; // Displaying Queue elements
        }
}
```

RESULT

Thus, the program for implementation of queue operations using array is executed and verified successfully.

EX. NO: 5	IMPLEMENTATION OF LIST ADT OPERATIONS
DATE:	

AIM

To Write a C Program to implement the list ADT operations.

ALGORITHM

Insertion

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

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

Inserting at Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list

Step 1 - Create a new Node with given value.
Step 2 - Check whether list is Empty (head == NULL)
Step 3 - If it is Empty then, set new Node→next = NULL and head = newNode.
Step 4 - If it is Not Empty then, set new Node→next = head and head = newNode.

Inserting at End of the list

We can use the following steps to insert a new node at end of the single linked list

Step 1 - Create a newNode with given value and newNode → next as NULL.
Step 2 - Check whether list is Empty (head == NULL).
Step 3 - If it is Empty then, set head = newNode.
Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6 - Set temp → next = newNode.

Inserting at Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list

Step 1 - Create a newNode with given value.
Step 2 - Check whether list is Empty (head == NULL)
Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.
Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

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

Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 - Check whether list is having only one node (temp → next == NULL)
Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
Step 6 - If it is FALSE then set head = temp → next, and delete temp

Deleting from End of the list

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

Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4 - Check whether list has only one Node (temp1 → next == NULL)
Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list

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

Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1→ next == NULL).
Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

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

Step 1 - Check whether list is Empty (head == NULL)
Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

RESULT

Thus, the program for implementation of list ADT operations is executed and verified successfully.

EX. NO: 6 a)	SINGLY LINKED LIST IMPLEMENTATION OF STACK ADT
DATE:	

AIM

To Write a C Program to implement the stack operations using singly linked list

ALGORITHM

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

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

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

Step 3 - Define a Node pointer 'top' and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

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

Step 1 - Create a newNode with given value.

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

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

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

Step 5 - Finally, set top = newNode.

pop() - Deleting an Element from a Stack

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

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

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

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

Step 4 - Then set 'top = top → next'.

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

display() - Displaying stack of elements

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

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

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

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

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

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

RESULT

Thus, the program for implementation of stack operations using singly linked list is executed and verified successfully.

EX. NO: 6 b)	SINGLY LINKED LIST IMPLEMENTATION OF QUEUE ADT
DATE:	

AIM

To Write a C Program to implement the queue operations using singly linked list

ALGORITHM

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

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

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

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

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

enQueue(value) - Inserting an element into the Queue

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

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

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

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

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

deQueue() - Deleting an Element from Queue

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

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

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

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

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

display() - Displaying the elements of Queue

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

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

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

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

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

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

RESULT

Thus the program for implementation of queue operations using singly linked list is executed and verified successfully.

EX. NO: 7 a)	IMPLEMENTATION OF DOUBLY LINKED LIST
DATE:	

AIM

To Write a C Program to implement the doubly linked list operations.

ALGORITHM

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

Inserting at Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

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

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

Step 3 - If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4 - If it is not Empty then, assign head to newNode →next and newNode to head.

Inserting at End of the list

We can use the following steps to insert a new node at end of the double linked list...

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

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

Step 3 - If it is Empty, then assign NULL to newNode →previous and newNode to head.

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

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

Step 6 - Assign newNode to temp → next and temp to newNode → previous.

Inserting at Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a newNode with given value.

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

Step 3 - If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.

Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.

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

location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step 7 - Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode →next and newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

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

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

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

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

Step 4 - Check whether list is having only one node (temp → previous is equal to temp → next)

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

Step 6 - If it is FALSE, then assign temp → next to head, NULL to head →previous and delete temp

Deleting from End of the list

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

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

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

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

Step 4 - Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7 - Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

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

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

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

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

Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

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

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

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

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

Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.

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

Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).

Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

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

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

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

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

Step 4 - Display 'NULL <--- '.

Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node

Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data --->NULL).

RESULT

Thus the program for implementation of double ended queue ADT using a doubly linked list is executed and verified successfully.

EX. NO: 7 b)	IMPLEMENTATION OF DOUBLE ENDED QUEUE
DATE:	

AIM

To Write a C Program to implement the double ended queue operations.

ALGORITHM

Enqueue(front)

Step 1. Check whether queue is full (f==0 && r==N-1) || (f==r+1)
Step 2. If it is full then display “Queue full, Insertion is not possible” and terminate the function
Step 3. Else if (f==-1 && r==-1) then assign f=r=0 and set dequeue [f]= value
Step 4. Else if (f==0) assign f=n-1 and dequeue[f]= value
Step 5. Else if decrement front value by one(f--) and set dequeue [f]= value

Enqueue(rear)

Step 1. Check whether queue is full
Step 2. If it is full then display “Queue full, Insertion is not possible” and terminate the function
Step 3. Else if (f==-1) && (r==-1) then assign r=0 and set dequeue[r] = value
Step 4. Else if (r==N-1) then assign r=0 and set dequeue[r] = value
Step 5. Else increment rear value by ONE (r++) and set dequeue[r] = value

Dequeue(front)

Step 1. Check whether queue is empty (f==-1 && r==-1)
Step 2. If it is empty then display “Queue is empty, deletion is not possible” and terminate the function
Step 3. Else if (f==r) then print the deleted element and assign f=-1 and r=-1
Step 4. Else if (f==n-1) then print the deleted element and assign f=0
Step 5. Else print the deleted element and increment front value by one(f+1)

Dequeue(rear)

Step 1. Check whether queue is empty (f==-1 && r==-1)
Step 2. If it is empty then display “Queue is empty, deletion is not possible” and terminate the function
Step 3. Else if (f==r) then print the deleted element and assign f=-1 and r=-1
Step 4. Else if (r==0) then print the deleted element and assign r=n-1
Step 5. Else print the deleted element and decrement rear value by one(r-1)

RESULT

Thus the program for implementation of double ended queue ADT using an array is executed and verified successfully.

EX. NO: 8. a)	IMPLEMENTATION OF BINARY SEARCH TREE
OPERATIONS	

DATE:

AIM

To Write a C Program to implement Binary Search Tree.

ALGORITHM

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6-** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node
- **Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

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

- **Step 1 -** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2 -** Check whether tree is Empty.
- **Step 3 -** If the tree is **Empty**, then set **root** to **newNode**.
- **Step 4 -** If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5 -** If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

- **Step 6-** Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).
- **Step 7 -** After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity.

Deleting a node from Binary search tree includes following three cases...

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

RESULT

Thus the program for implementation of binary search tree operations is executed and verified successfully.

EX. NO: 8 b)	IMPLEMENTATION OF BINARY TREE TRAVERSAL
DATE:	

AIM

To develop a C program to create a Binary tree and perform the 3 different types of traversals: Inorder, Preorder and Postorder.

ALGORITHM

```
Struct tree
{
        int data;
        Struct tree, *left, *right;
}
*newnode

Algorithm createnode()
{
        newnode= new node;
        read value;
        newnode-> data= value;
        newnode-> left=NULL;
        newnode-> right=NULL;
}

Algorithm insert(Struct tree *root, value)
{
        if(root==NULL)
{
root = newnode;
}
else
{
        node=root;
        if value= node□data
        {
                Write —duplicate value exists‖;
                return;
        }
        If value < node□data
        {
                if (node->left = NULL)
```

```
                {
                        node→left = newnode;
                }
                else
                {
                        insert(node→left, value);
                }
        }
        else
        {
                if (node→right = NULL)
                {
                        node→right = newnode;
                }
                else
                {
                        insert(node→right, value);
                }               }       }
}
Algorithm inorder (Struct tree, *node)
{
        if (node==NULL)
        {
                return;
        }
        inorder(node→left);
        write node→data;
        inorder(node→right);
}
Algorithm preorder (Struct tree, *node)
{
        if (node==NULL)
        {
                return;
```

```
        }
        write node→data;
        inorder(node→left);
        inorder(node→right);
}
Algorithm postorder (Struct tree, *node)
{
        if (node==NULL)
        {
                return;
        }
        inorder(node→left);
        inorder(node→right);
        write node→data;
}
```

RESULT

Thus, the program for implementation of binary search tree operations is executed and verified successfully.

EX.NO: 9	
DATE :	AVL TREE

AIM

To develop a program to implement AVL Tree.

ALGORTIHM:

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

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

- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

RESULT

Thus, the program for implementation of AVL tree is executed and verified successfully.

EX.NO: 10 a)	
DATE :	GRAPH TRAVERSAL – DEPTH FIRST SEARCH

AIM

To write program to implement the concept of depth first search travel in graph.

ALGORITHM

```
Algorithm DFS(v)
{
        write v;
        visited[v]=1;
        for i=1 to n
        {
                for j=1 to n
                {
                        if edge[v,i]==1 && visited[i]==0
                        {
                                write [i];
                                visited[i]=1;
                                DFS[i];
                        }
                }
        }
}
```

RESULT

Thus, the program for implementation of Depth First Search is executed and verified successfully.

EX.NO: 10 b)	
DATE :	**GRAPH TRAVERSAL – BREADTH FIRST SEARCH**

AIM

To write program to implement the concept of breadth first search travel in graph.

ALGORITHM

```
Algorithm BFS(n)
{
        \\Given undirected graph G=(V,E) with n vertices
        \\Edge[][]- adjacency matrix of given graph
        \\visited[] initially set to zero for all vertices
write v;
visited[v]=1;
front=rear=0;
while(front==rear)
          {
 for(i=1 to n) do
 {
 for(j=1 to n) do
 {
 if (edge[v,i]==1 && visited[i]==0) then
 {
 write i;
 visited[i]=1;
 queue[rear++]=i;
 }
 }
 }
 }
 v=queue[front];
 }
```

RESULT

Thus, the program for implementation of Breadth First Search is executed and verified successfully.