

TABLE OF CONTENTS

| No | Date | Title of the Exercises | Marks | Page no | Staff -Sig |
|----|------|-------------------------|-------|---------|------------|
| 1 | | LINEAR SEARCH | | | |
| 2 | | BINARY SEARCH | | | |
| 3 | | INSERTION SORT | | | |
| 4 | | SELECTION SORT | | | |
| 5 | | SHELL SORT | | | |
| 6 | | BUBBLE SORT | | | |
| 7 | | HEAP SORT | | | |
| 8 | | STACK | | | |
| 9 | | QUEUE | | | |
| 10 | | SINGLY LINKED LIST | | | |
| 11 | | STACK USING LINKED LIST | | | |
| 12 | | QUEUE USING LINKED LIST | | | |
| 13 | | DOUBLY LINKED LIST | | | |
| 14 | | BINARY SEARCH TREE | | | |
| 15 | | BINARY TREE TRAVERSALS | | | |
| 16 | | AVL TREE | | | |
| 17 | | DEPTH FIRST SEARCH | | | |
| 18 | | BREATH FIRST SEARCH | | | |

| | |
|---------------|--|
| EX.NO: | |
| DATE : | |

LINEAR SEARCH

AIM

To Write a C Program to search a particular element from the given set of elements using Linear Search.

DESCRIPTION:

- This search process starts comparing search element with the first element in the list.
- If both are matched then result is element found otherwise search element is compared with the next element in the list.
- Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list".

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

list

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are matching. So we stop comparing and display element found at index 5.

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"  
}
```

PROGRAM CODE:

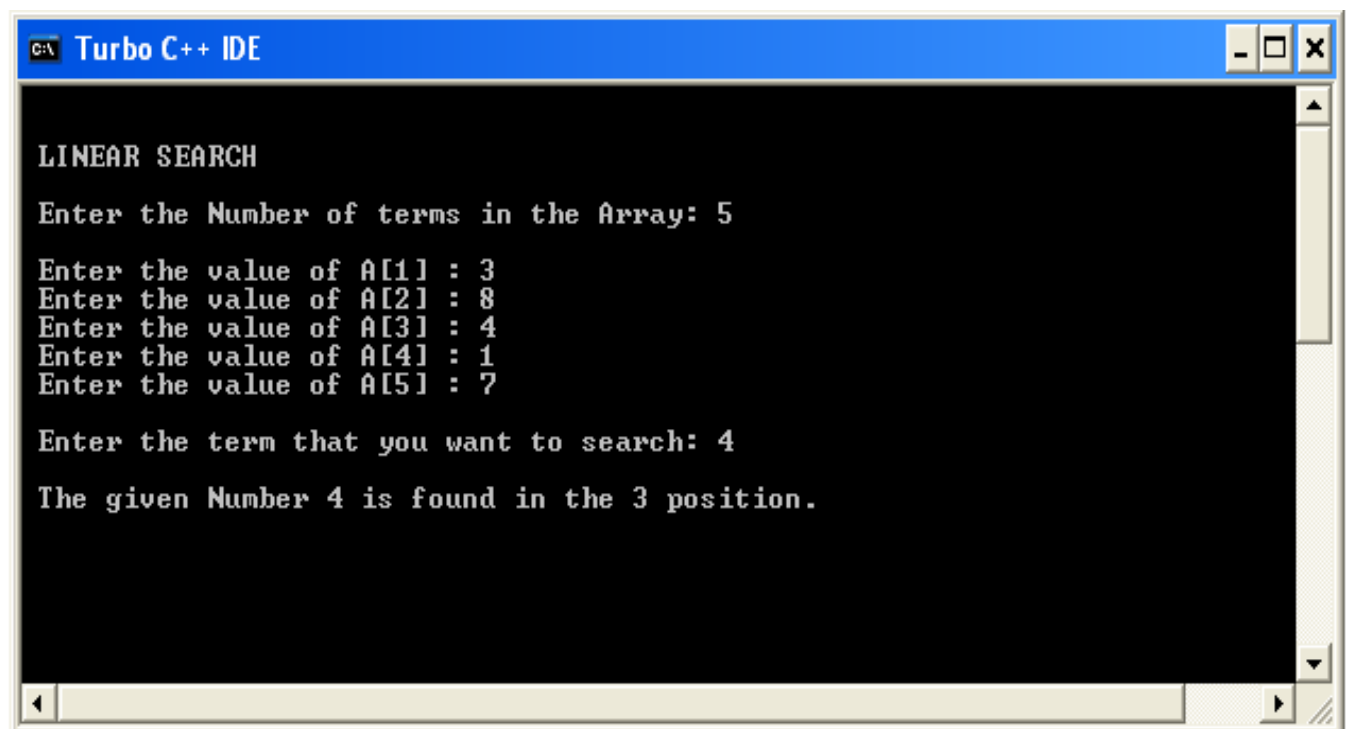
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,a[20],key,flag=0;
    clrscr();
    printf("LINEAR SEARCH");
    //GET THE INPUT
    printf("\n\nEnter the number of terms in the array\n");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\nEnter the term that you want to search\n");
    scanf("%d",&key);
    // PERFORMS SEARCHING
    for(i=0;i<n;i++)
    {
        if(key==a[i])
        {
            // PRINTS THE FOUND ELEMENT & ITS POSITION
            flag=1;
            break;
        }
    }

    // PRINTS IF THE ELEMENT IS FOUND
    if(flag==1)
    {
        printf("\n\nThe given number %d is found in the array \n", key,i+1);
    }
}
```

```
// PRINTS IF THE ELEMENT IS NOT FOUND
else
    printf("\n %d is not found in the array",key);
getch();
}
```

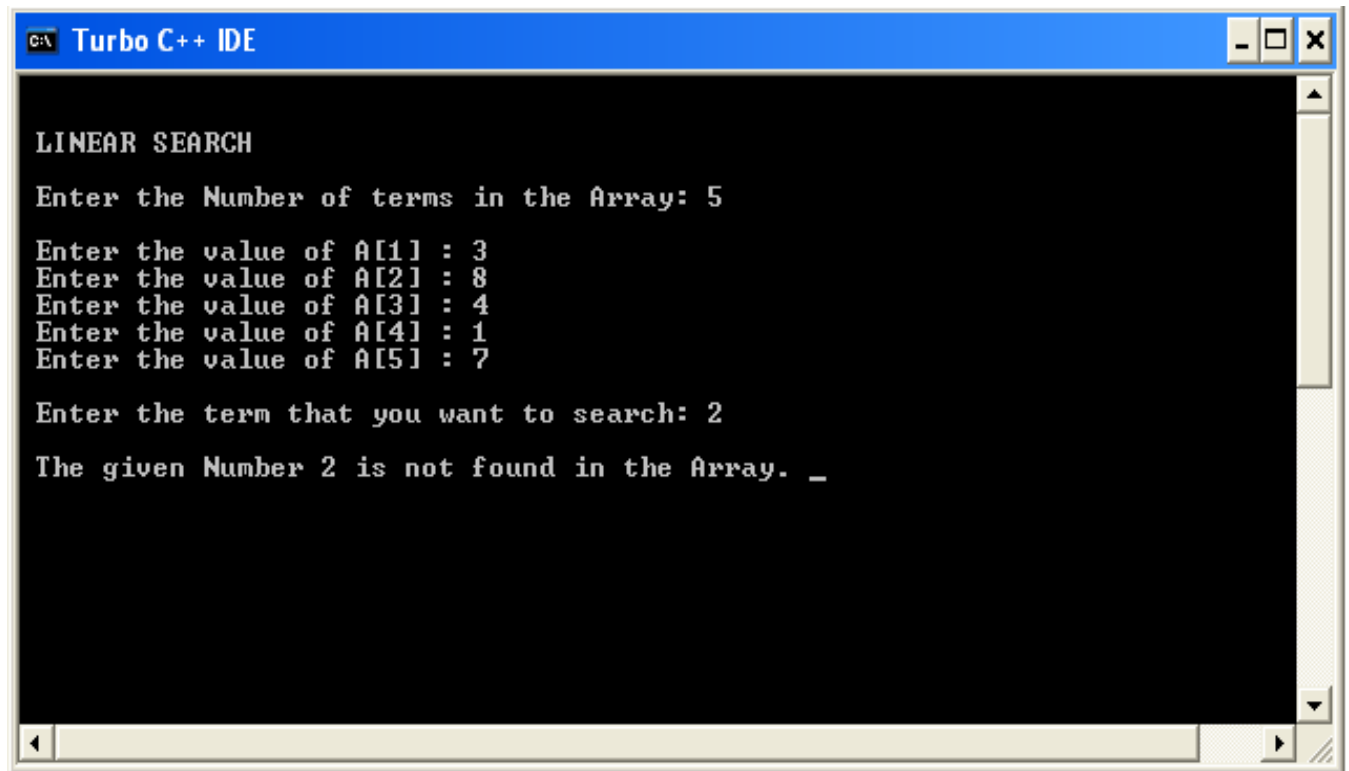
OUTPUT



The screenshot shows a Turbo C++ IDE window with a blue title bar. The main window is black with white text. The text displayed is as follows:

```
LINEAR SEARCH
Enter the Number of terms in the Array: 5
Enter the value of A[1] : 3
Enter the value of A[2] : 8
Enter the value of A[3] : 4
Enter the value of A[4] : 1
Enter the value of A[5] : 7

Enter the term that you want to search: 4
The given Number 4 is found in the 3 position.
```



The screenshot shows the Turbo C++ IDE window with a blue title bar. The main text area is black with white text. The program is titled 'LINEAR SEARCH'. It prompts the user to enter the number of terms in the array (5), then the values of five array elements (A[1] to A[5]). The values entered are 3, 8, 4, 1, and 7 respectively. It then prompts for the term to search (2). The final output is 'The given Number 2 is not found in the Array. _'.

```
Turbo C++ IDE

LINEAR SEARCH

Enter the Number of terms in the Array: 5

Enter the value of A[1] : 3
Enter the value of A[2] : 8
Enter the value of A[3] : 4
Enter the value of A[4] : 1
Enter the value of A[5] : 7

Enter the term that you want to search: 2

The given Number 2 is not found in the Array. _
```

| CONTENTS | MARKS ALLOTTED | MARKS OBTAINED |
|-----------------------|----------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT

Thus the C program is executed, the given key element is identified in the current location and the output is verified.

EXERCISES:

Write a C program to create a student details.

- Enter details of the Student_Reg , Student_name, Student marks of five subjects
- Search the particular student details based on regno in the given list of students using Linear searching technique.

| | |
|---------------|--|
| EX.NO: | |
| DATE : | |

BINARY SEARCH

AIM:

To Write a C Program to search a particular element from the given set of elements using Binary Search algorithm.

DESCRIPTION:

- The binary search algorithm can be used with only a sorted list of elements.
- That means the binary search is used only with a list of elements that are already arranged in an order.
- The binary search can not be used for a list of elements arranged in random order.
- This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list.
- If the search element is smaller, then we repeat the same process for the left sublist of the middle element.
- If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element.
- And if that element also doesn't match with the search element, then the result is "Element not found in the list".

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

search element **12**

Step 1:

search element (12) is compared with middle element (50)

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

Step 2:

search element (12) is compared with middle element (12)

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

12

Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

Step 2:

search element (80) is compared with middle element (65)

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

Step 3:

search element (80) is compared with middle element (80)

list

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

80

Both are not matching. So the result is "Element found at index 7"

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;
        }
    }
}
```

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
//DECLARATION
int i,mid,low,high,n,key,a[20],index;
int binsearch();
void main()
{
    clrscr();
    // GETS THE INPUT
    printf("\nEnter the number of terms in the array :\n");
    scanf("%d",&n);
    printf("\nEnter the array elements in sorted order:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\nEnter the number that you want to search:\n");
    scanf("%d",&key);
    index=binsearch();
    if(index!=-1)
    {
        printf("\nthe given number %d is found in the %d position \n",key,index+1);
    }
    else
    {
        printf("\nThe given number %d not found in the array\n",key);
    }
    getch();
}
// PERFORMS BINARY SEARCH
int binsearch()
{
    low=0;
```

```

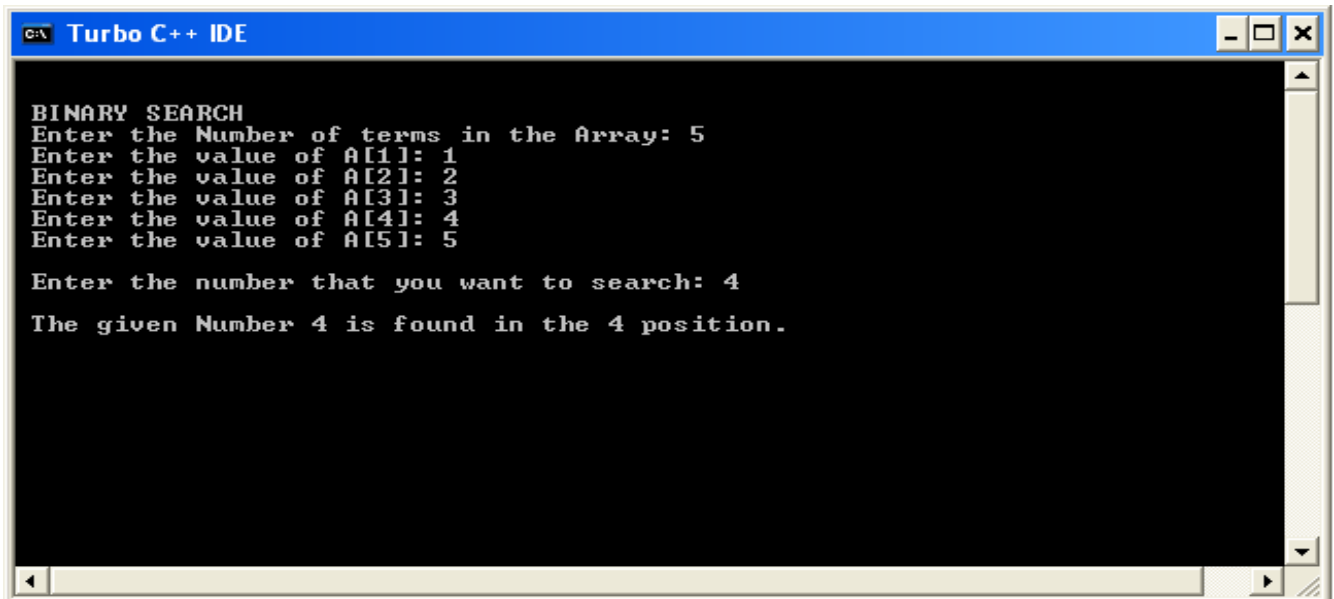
high=n-1;
while (low<high)
{
    mid=(low+high)/2;
    if(key==a[mid])
    {
        return mid;
    }
    else if(key>a[mid])
    {
        low=mid+1;
    }

else if(key<a[mid])
    {
        high=mid-1;
    }
}

return -1;
}

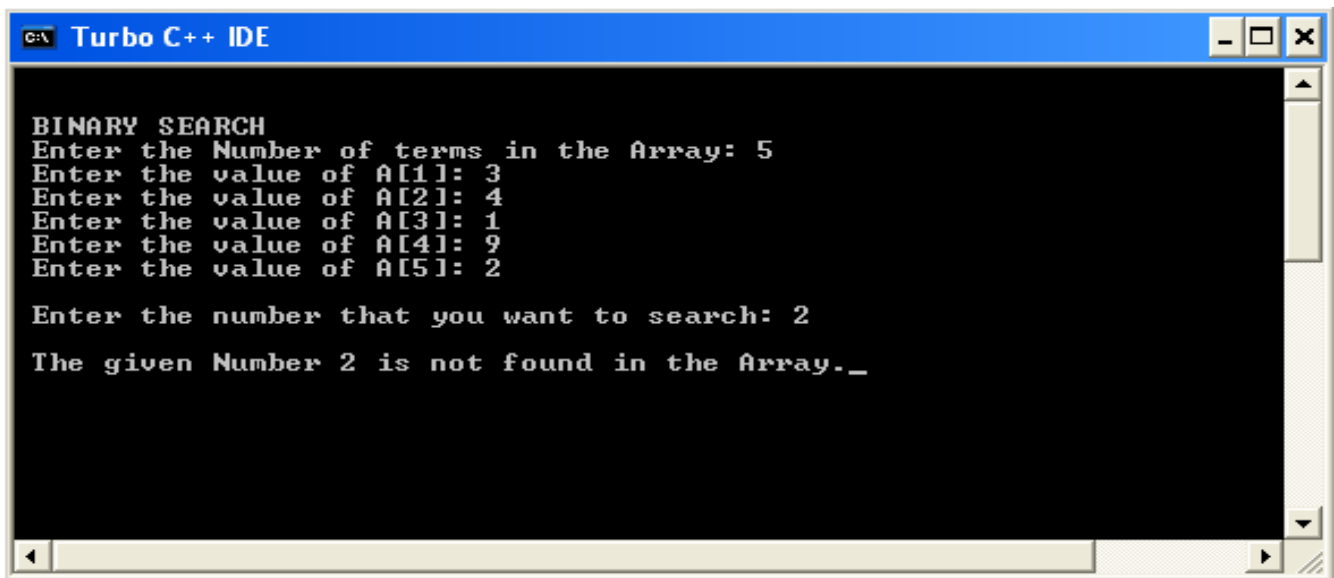
```

OUTPUT:



```
BINARY SEARCH
Enter the Number of terms in the Array: 5
Enter the value of A[1]: 1
Enter the value of A[2]: 2
Enter the value of A[3]: 3
Enter the value of A[4]: 4
Enter the value of A[5]: 5

Enter the number that you want to search: 4
The given Number 4 is found in the 4 position.
```



```
BINARY SEARCH
Enter the Number of terms in the Array: 5
Enter the value of A[1]: 3
Enter the value of A[2]: 4
Enter the value of A[3]: 1
Enter the value of A[4]: 9
Enter the value of A[5]: 2

Enter the number that you want to search: 2
The given Number 2 is not found in the Array._
```

| CONTENTS | MARKS ALLOTTED | MARKS OBTAINED |
|-----------------------|----------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program is executed using binary searching, the given key element is identified in the current location and the output is verified.

EXERCISES:

Write a C program for search a particular flight on a given date in a given list of flights.

- Enter the flight name , flight no , date of departure.
- Search the particular flight name and flight no on the given date using binary searching technique.

EX.NO:

DATE :

INSERTION SORT

AIM:

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

DESCRIPTION:

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using the following steps...

- **Step 1** - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Consider the following unsorted list of elements...

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 15 | 20 | 10 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

Assume that sorted portion of the list is empty and all elements in the list are in the unsorted portion of the list as shown in the figure below...

| Sorted | Unsorted |
|--------|---------------------|
| 15 | 20 10 30 50 18 5 45 |

Move the first element 15 from unsorted portion to sorted portion of the list.

| Sorted | Unsorted |
|--------|---------------------|
| 15 | 20 10 30 50 18 5 45 |

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

| Sorted | Unsorted |
|--------|------------------|
| 15 20 | 10 30 50 18 5 45 |

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is inserted at its correct position in sorted portion of the list.

| Sorted | Unsorted |
|----------|---------------|
| 10 15 20 | 30 50 18 5 45 |

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

| Sorted | Unsorted |
|-------------|------------|
| 10 15 20 30 | 50 18 5 45 |

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

| Sorted | Unsorted |
|----------------|----------|
| 10 15 20 30 50 | 18 5 45 |

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

| Sorted | Unsorted |
|-------------------|----------|
| 10 15 18 20 30 50 | 5 45 |

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these elements, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

| Sorted | Unsorted |
|---------------------|----------|
| 5 10 15 18 20 30 50 | 45 |

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

| Sorted | Unsorted |
|------------------------|----------|
| 5 10 15 18 20 30 45 50 | |

Unsorted portion of the list has become empty. So we stop the process. And the final sorted list of elements is as follows...

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 18 | 20 | 30 | 45 | 50 |
|---|----|----|----|----|----|----|----|

ALGORITHM:

Algorithm insertionsort(a,n)

```
{  
    for(i=2 to n) do  
        {  
            key=a[i];  
            j=i-1;  
            while((j>0) && (key < a[j])) do  
                {  
                    a[j+1]=a[j];  
                    j=j-1;  
                }  
            a[j+1]=key;  
        }  
}
```

PROGRAM CODE :

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i,j,key,a[200],n;
    clrscr();
    printf("\n\n Insertion Sort");
    printf("\n\n Enter the number of values you have got: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf(" Enter the %d value: ",i+1);
        scanf("%d",&a[i]);
    }
    printf("\n\n Sorted List");
    for(i=1;i<n;i++)    // INSERTION SORT LOGIC
    {
        key=a[i];
        j=i-1;
        while((j>=0)&&(key<a[j]))
        {
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=key;
    }
    for(i=0;i<n;i++)
    {
        printf("\n %d",a[i]);
    }
    getch();
    return 0;
}
```

OUTPUT:

A screenshot of the Turbo C++ IDE window. The title bar reads "Turbo C++ IDE". The main text area has a black background with white text. It displays the output of an Insertion Sort program. The text is as follows:

```
Insertion Sort
Enter the number of values you have got: 5
Enter the 1 value: 3
Enter the 2 value: 6
Enter the 3 value: 2
Enter the 4 value: 9
Enter the 5 value: 0

Sorted List
0
2
3
6
9
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the given elements is been sorted by using insertion sorting technique and the output is executed and verified.

EXERCISES:

Write a C program to implement a telephone directory details.

- Enter the details of the person_name , phone numbers and address.
- And display the details in a sorted alphabetical order using the insertion sorting technique.

EX.NO:

DATE :

SELECTION SORT

AIM:

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

DESCRIPTION :

- In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list.
- If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order.
- Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list.
- If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using the following steps:

- **Step 1** - Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

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[min];
            a[min]=t;
        }
    }
}
```


| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 15 | 20 | 10 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 15 | 20 | 10 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

15 > 20
FALSE

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 15 | 20 | 10 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

15 > 10
TRUE
SWAP

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 10 | 20 | 15 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

10 > 30
FALSE

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 10 | 20 | 15 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

10 > 50
FALSE

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 10 | 20 | 15 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

10 > 18
FALSE

| | | | | | | | |
|----|----|----|----|----|----|---|----|
| 10 | 20 | 15 | 30 | 50 | 18 | 5 | 45 |
|----|----|----|----|----|----|---|----|

10 > 5
TRUE
SWAP

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 20 | 15 | 30 | 50 | 18 | 10 | 45 |
|---|----|----|----|----|----|----|----|

5 > 45
FALSE

List after 1st iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 20 | 15 | 30 | 50 | 18 | 10 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 20 | 30 | 50 | 18 | 15 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 30 | 50 | 20 | 18 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 18 | 50 | 30 | 20 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 18 | 20 | 50 | 30 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 18 | 20 | 30 | 50 | 45 |
|---|----|----|----|----|----|----|----|

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 5 | 10 | 15 | 18 | 20 | 30 | 45 | 50 |
|---|----|----|----|----|----|----|----|

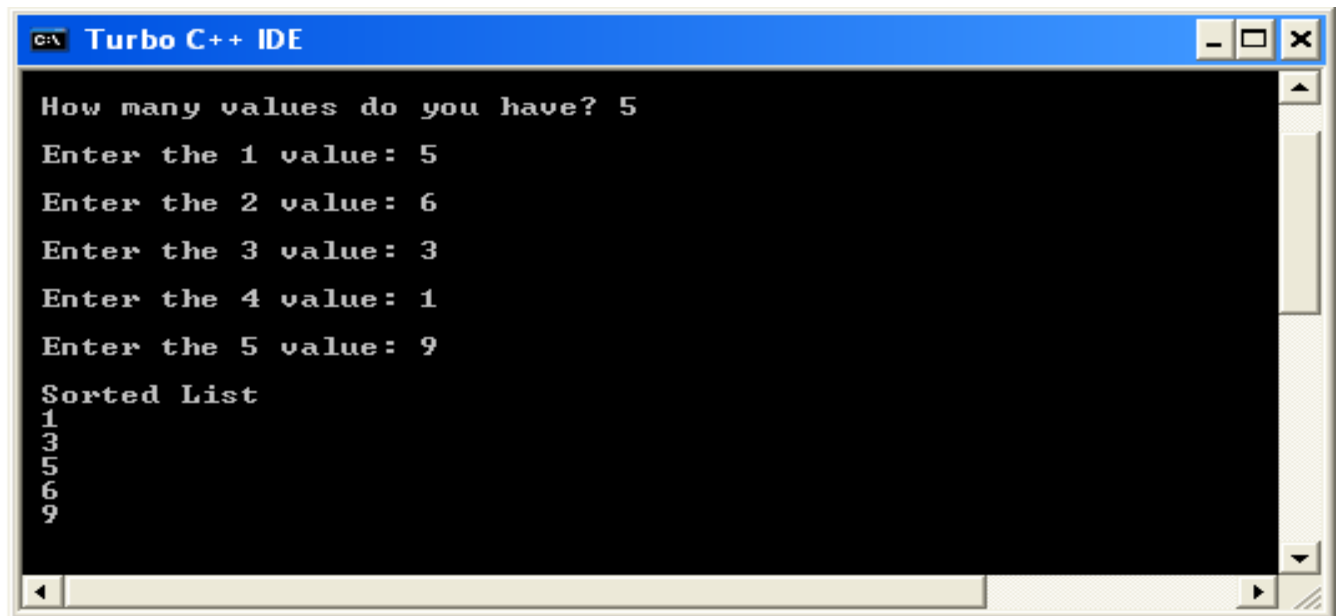
Final sorted list

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j,k,min,n,a[200];
    clrscr();
    printf("\n\n Selection Sort \n ");
    printf("\n How many values do you have? ");
    scanf("%d",&n);
    for(i=0;i<n;i++)                // GETTING THE INPUT FROM THE USER
    {
        printf("\n Enter the %d value: ",i+1);
        scanf("%d",&a[i]);
    }
    printf("\n Sorted List");
    for(i=0;i<n;i++)                // LOGIC FOR THE SELECTION SORT
    {
        k=i;
        min=a[i];
        for(j=i+1;j<n;j++)
        {
            if(a[j]<min)
            {
                min=a[j];
                k=j;
            }
        }
        a[k]=a[i];
        a[i]=min;
        printf("\n %d",a[i]);
    }
    getch();
}
```

}

OUTPUT:

A screenshot of the Turbo C++ IDE window. The title bar reads "Turbo C++ IDE". The main text area shows the following output:

```
How many values do you have? 5
Enter the 1 value: 5
Enter the 2 value: 6
Enter the 3 value: 3
Enter the 4 value: 1
Enter the 5 value: 9
Sorted List
1
3
5
6
9
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the given elements is been sorted by using selection sorting technique and the output is executed and verified.

EXERCISES:

Write a C program to implement a Student details.

- Enter the details of the student name , Regno and Department.
- Sort and display the student details based on the Register number using selection sorting technique.

EX.NO:

DATE :

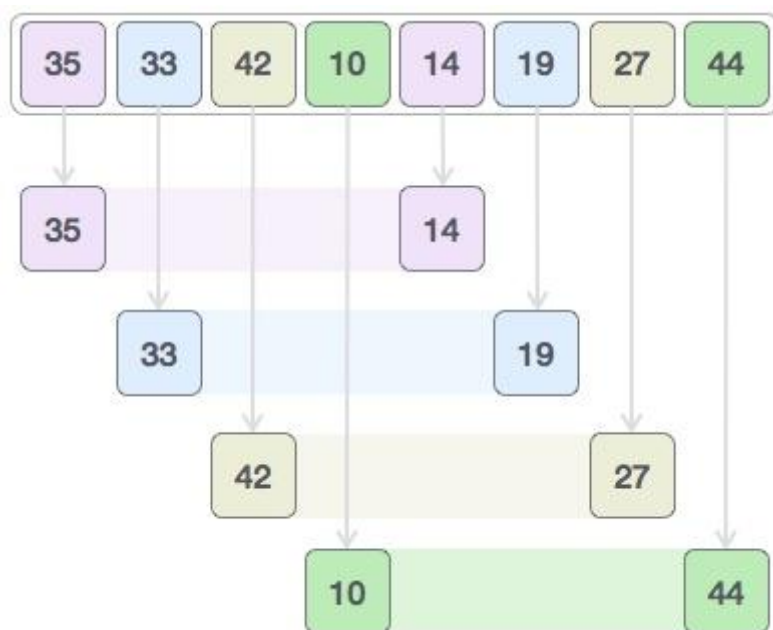
Shell SORT

AIM:

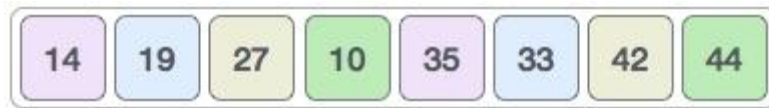
To Write a C Program to sort the given set of elements using Shell Sort.

DESCRIPTION:

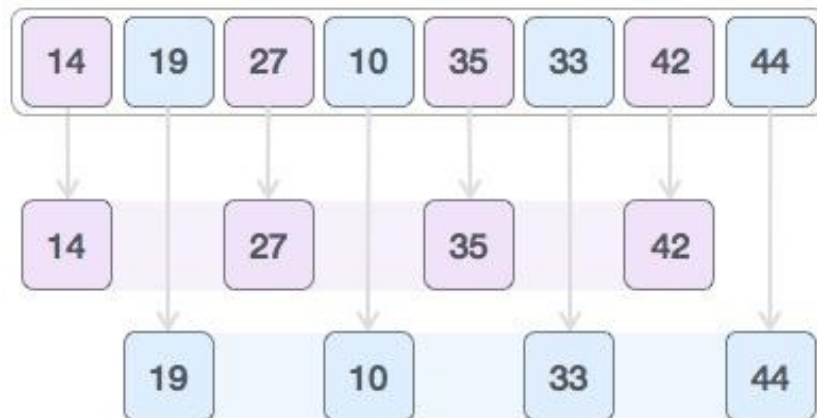
- This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is $O(n)$, where n is the number of items.
- And the worst case space complexity is $O(n)$.
- Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous
- Examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



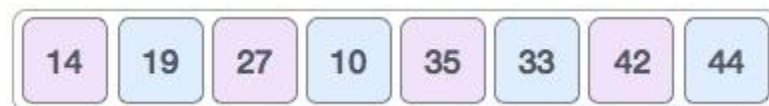
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array. Following is the step-by-step depiction.



ALGORITHM

```

Shlsort(ElementTypeA[ ], int N)
{
    int i, j, increment;
    ElementType Tmp;

```

```

    For(increment=N/2;increment>0;increment++)
        For(i=increment;i<N;i++)
            {
                Tmp=A[i];
                For(j=i;j>=increment;j=increment)
                    If(Tmp<A[j-increment])
                        A[j]=A[j-increment];
                Else
                    Break;
                A[j]=Tmp;
            }
    }

```

PROGRAM CODE:

```

#include<stdio.h>
#include<conio.h>
void shellsort(int a[],int n)
{
    int j,i,k,m,mid;

```

```

for(m = n/2;m>0;m/=2)
{
    for(j = m;j< n;j++)
    {
        for(i=j-m;i>=0;i-=m)
        {
            if(a[i+m]>=a[i])
                break;
            else
            {
                mid = a[i];
                a[i] = a[i+m];
                a[i+m] = mid;
            }
        }
    }
    printf("\n\nPass %d:",m);
    for(k=0;k<n;k++)
        printf("%d\t",a[k]);
}
}

int main()
{
    int a[20],i,n;
    clrscr();
    printf("\t\tSHELL SORT ALGORITHM\n\n\n\n");
    printf("Enter total elements : ");
    scanf("%d", &n);
    printf("\nEnter %d Elements : ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("\nBefore Sorted list by Shell sort :\n ");
    for(i=0;i< n;i++)
        printf("%d\t",a[i]);

```

```
shellsort(a,n);  
printf("\nAfter Sorted list by Shell sort :\n ");  
for(i=0;i< n;i++)  
printf("%d\t",a[i]);  
getch();  
return 0;  
}
```

OUTPUT

```
Turbo C++ IDE

SHELL SORT ALGORITHM

Enter total elements : 8
Enter 8 Elements : 56 85 92 99 67 28 95 15
Before Sorted list by Shell sort :
56    85    92    99    67    28    95    15
Pass 4:56    28    92    15    67    85    95    99
Pass 2:56    15    67    28    92    85    95    99
Pass 1:15    28    56    67    85    92    95    99
After Sorted list by Shell sort :
15    28    56    67    85    92    95    99
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the given elements is been sorted by using shell sorting technique and the output is executed and verified.

EXERCISES:

Write a C program to implement the employee details and to sort the list based on the joining date.

- Enter the employee details like Employee_name , Employee_designation and salary and date of joining .
- Sort and display the employee details based on the joining date using Shell sorting technique.

EX.NO:

DATE :

BUBBLE SORT

AIM:

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

DESCRIPTION:

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

First Pass:

(5 1 4 2 8) → (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) → (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) → (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) → (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 2 4 5 8), Swap since $4 > 2$

(1 2 4 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

ALGORITHM

Algorithm bubblesort(a,n)

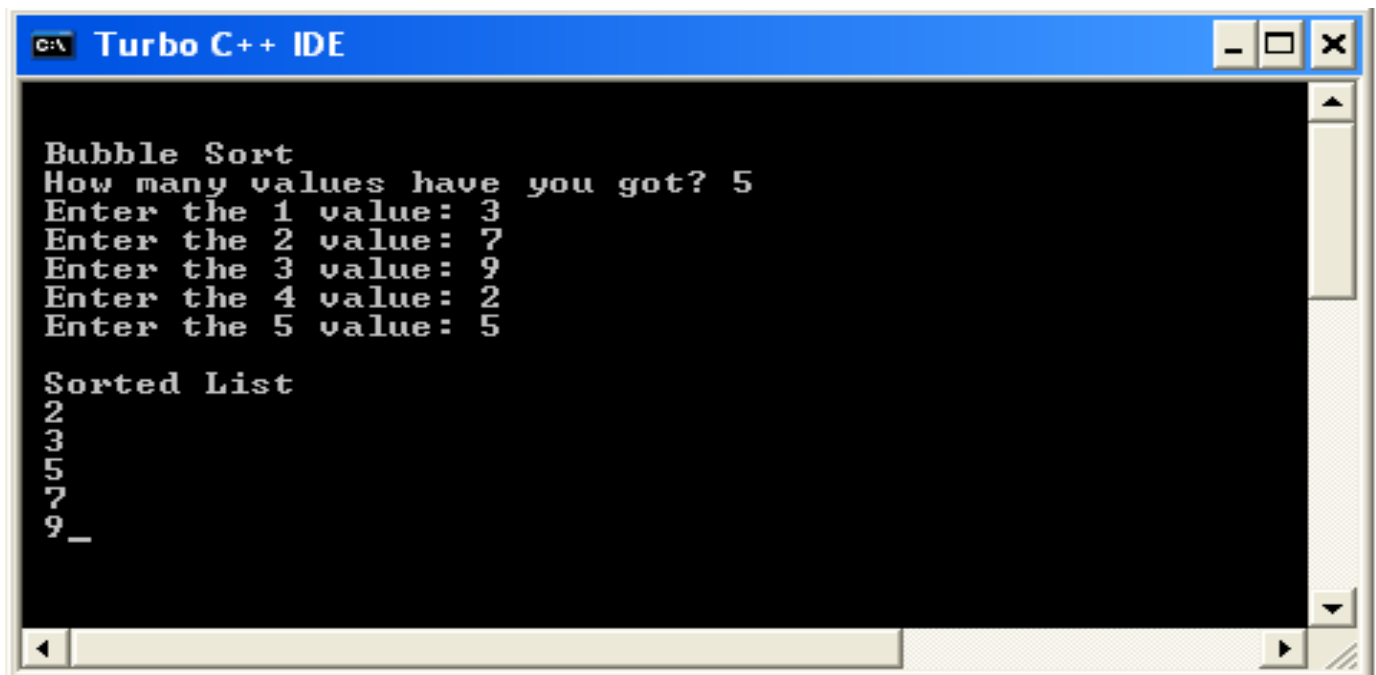
```
{  
    for(i=1 to n) do  
    {  
        for(j=1 to n) do  
        {  
            if(a[j]>a[j+1]) then  
            {  
                temp=a[j];  
                a[j]=a[j+1];  
                a[j+1]=temp;  
            }  
        }  
    }  
}
```

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i,j,temp,n,a[200];
    clrscr();
    printf("\n\n Bubble Sort");
    printf("\n How many values have you got? ");
    scanf("%d",&n);
    for(i=0;i<n;i++)                                // GETTING THE INPUT FROM THE USER
    {
        printf(" Enter the %d value: ",i+1);
        scanf("%d",&a[i]);
    }
    printf("\n Sorted List");
    for(i=0;i<n;i++)                                // BUBBLE SORTING TECHNIQUE
    {
        for(j=0;j<n-1;j++)
        {
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
    for(i=0;i<n;i++)
        printf("\n %d",a[i]);
    getch();
    return 0;
```

}

OUTPUT:



The screenshot shows a Turbo C++ IDE window with a blue title bar. The text inside the window is as follows:

```
Bubble Sort
How many values have you got? 5
Enter the 1 value: 3
Enter the 2 value: 7
Enter the 3 value: 9
Enter the 4 value: 2
Enter the 5 value: 5

Sorted List
2
3
5
7
9_
```

RESULT:

Thus the given elements is been sorted by using bubble sorting technique and the output is executed and verified.

EXERCISES:

Write a c program to sort the given group of people according to their age.

- Get the user name , user id and user age as input and sort the different users based on their age by using Bubble sorting Technique.

EX.NO:

DATE :

HEAP SORT

AIM:

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

DESCRIPTION:

- Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**.
- In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

Step by Step Process

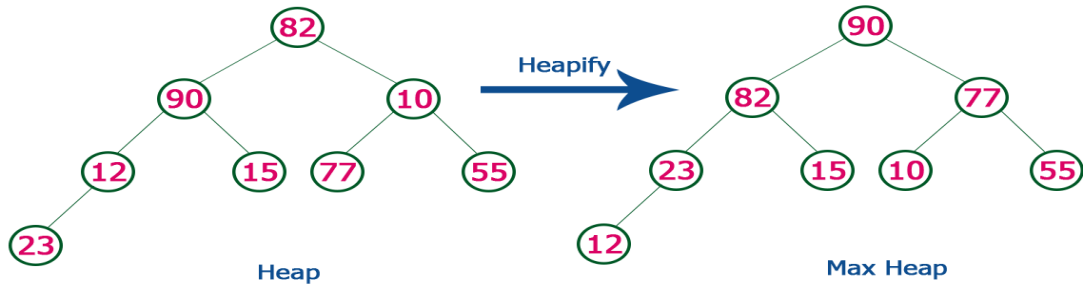
The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1** - Construct a **Binary Tree** with given list of Elements.
- **Step 2** - Transform the Binary Tree into **Min Heap**.
- **Step 3** - Delete the root element from Min Heap using **Heapify** method.
- **Step 4** - Put the deleted element into the Sorted list.
- **Step 5** - Repeat the same until Min Heap becomes empty.
- **Step 6** - Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

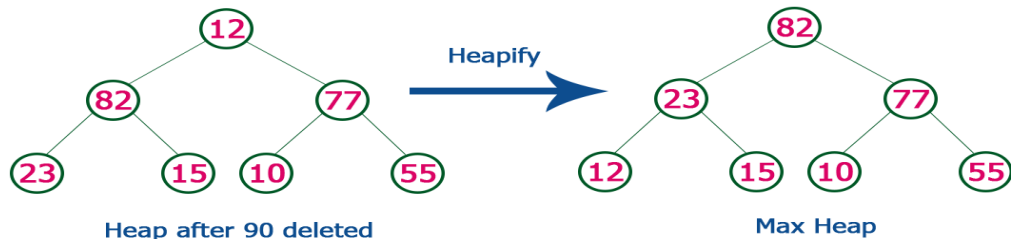
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

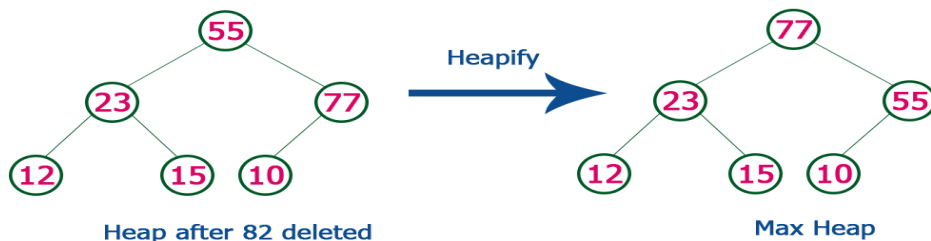
Step 2 - Delete root (90) from the Max Heap. To delete root node it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

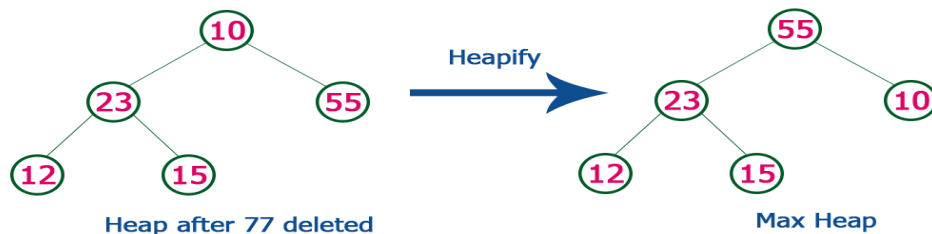
Step 3 - Delete root (82) from the Max Heap. To delete root node it needs to be swapped with last node (55). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

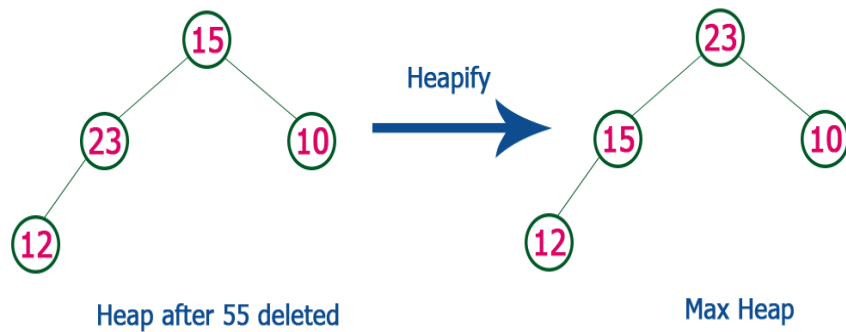
Step 4 - Delete root (77) from the Max Heap. To delete root node it needs to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

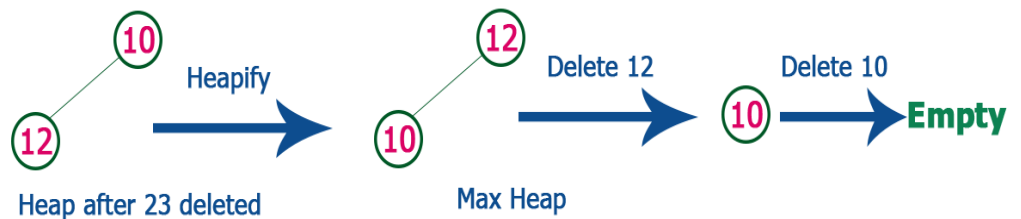
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

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
 3. Exit

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
void create(int []);
void down_adjust(int [],int);
int main()
{
    int heap[30],n,i,last,temp;
    printf("Enter no. of elements:");
    scanf("%d",&n);
    printf("\nEnter elements:");
    for(i=1;i<=n;i++)
        scanf("%d",&heap[i]);
    //CREATE A HEAP
    heap[0]=n;
    create(heap);
    //SORTING
    while(heap[0] > 1)
    {
        //swap heap[1] and heap[last]
        last=heap[0];
        temp=heap[1];
        heap[1]=heap[last];
        heap[last]=temp;
        heap[0]--;
        down_adjust(heap,1);
    }
    //PRINT SORTED DATA
    printf("\nArray after sorting:\n");
    for(i=1;i<=n;i++)
        printf("%d ",heap[i]);

    return 0;
```

```

}
void create(int heap[])
{
int i,n;
n=heap[0]; //no. of elements

for(i=n/2;i>=1;i--)
    down_adjust(heap,i);
}
void down_adjust(int heap[],int i)
{
int j,temp,n,flag=1;
n=heap[0];
while(2*i<=n && flag==1)
{
    j=2*i;    //J Points To Left Child
    if(j+1<=n && heap[j+1] > heap[j])
        j=j+1;
    if(heap[i] > heap[j])
        flag=0;
    else
    {
        temp=heap[i];
        heap[i]=heap[j];
        heap[j]=temp;
        i=j;
    }
}
}
}

```

OUTPUT

Output

```
Enter no. of elements:5
Enter elements:12 8 46 3 7
Array after sorting:
7 8 12 23 46
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the given elements is been sorted by using bubble sorting technique and the output is executed and verified.

EXERCISES:

Write a C program for heap sort, Show the steps followed to create a heap for the given data.

42,23,74,11,65,58,94,36,99,87.

EX.NO:

DATE :

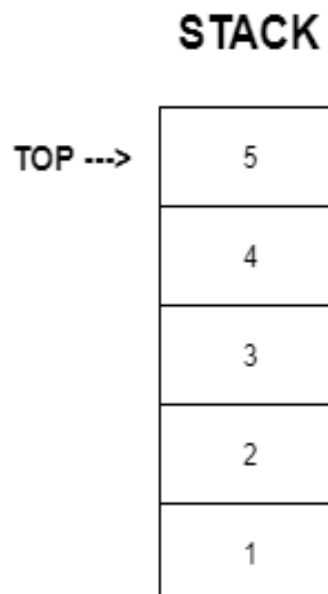
STACK USING ARRAY

AIM:

To Write a C Program A stack can be implemented using array.

DESCRIPTION:

- **Stack** is an abstract data type with a bounded(predefined) capacity.
- It is a simple data structure that allows adding and removing elements in a particular order.
- Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack.
- A **Stack** is a data structure following the LIFO (Last In, First Out) principle.



PUSH Operation

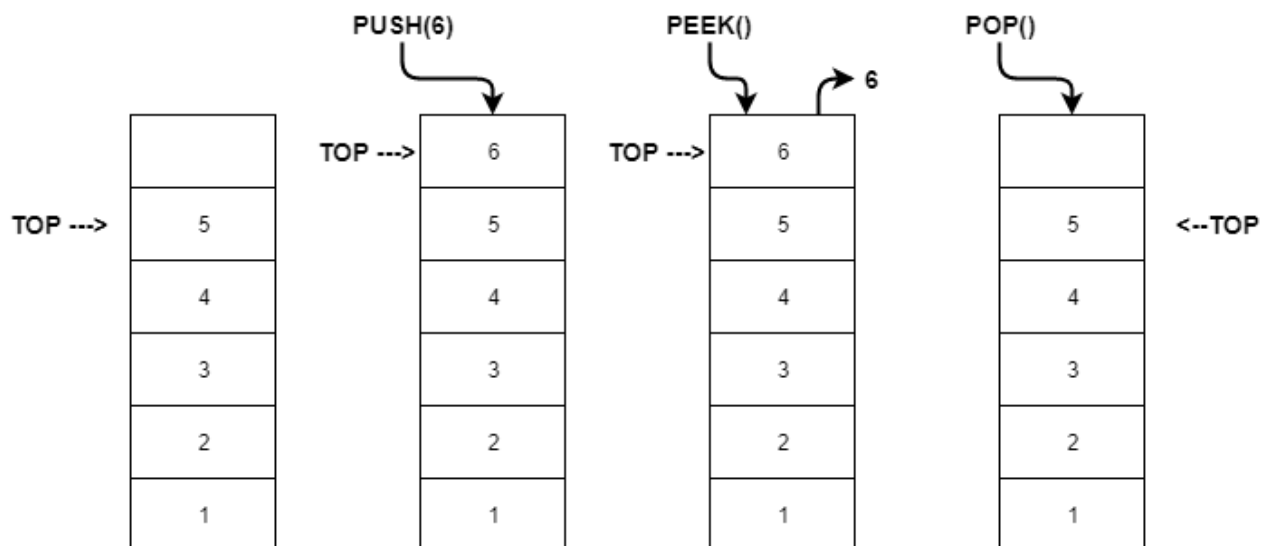
Push operation refers to inserting an element in the stack. Since there's only one position at which the new element can be inserted—Top of the stack, the new element is inserted at the top of the stack.

POP Operation

Pop operation refers to the removal of an element. Again, since we only have access to the element at the top of the stack, there's only one element that we can remove. We just remove the top of the stack. **Note:** We can also choose to return the value of the popped element back, its completely at the choice of the programmer to implement this.

PEEK Operation

Peek operation allows the user to see the element on the top of the stack. The stack is not modified in any manner in this operation.



(i) Different operation of the Stack

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];
    }
}
```

PROGRAM CODE :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define size 5
void push(int);
int pop(void);
void display();
int s[size],top=-1,ele,choice,i;
void main()
{
clrscr();
do
{
printf("\t\t\t\t\t stack operation \n");
printf("\n Enter the choice  \n 1.push \n 2.pop \n 3.display \n 4.exit \n");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the element");
scanf("%d",&ele);
push(ele);
break;
case 2:
ele=pop();
if(ele== -1)
{
printf("Delete the element %d",ele);
}
break;
case 3:
```

```

        display();
break;
case 4:
    exit (0);
break;
default:
    printf("wrong choice");
break;
}
}while(choice!=4);
getch();
}
void push(int el)                                // INSERTING THE ELEMENTS INTO THE STACK
{
if(top==size-1)
{
    printf("Stack is overflow");
}
else
{
    top=top+1;
    s[top]=el;
}
}
int pop()                                        // DELETING THE ELEMENTS FROM THE STACK
{
    if(top== -1)
    {
        printf("Stack is empty \n");
        return -1;
    }
    else
    {
        ele=s[top];

```

```

        top=top-1;
        return ele;
    }

}

void display()                // DISPLAYING THE ELEMENT IN THE STACK
{
    for(i=0;i<=top;i++)
    {
        printf("\n %d",s[i]);
    }
}

```

OUTPUT

```

                                stack operation

Enter the choice
1.push
2.pop
3.display
4.exit
1
Enter the element10
                                stack operation

Enter the choice
1.push
2.pop
3.display
4.exit

```

```

Enter the choice
1.push
2.pop
3.display
4.exit
1
Enter the element50
stack operation

Enter the choice
1.push
2.pop
3.display
4.exit
1
Enter the element60
Stack is overflow
stack operation

Enter the choice
1.push
2.pop
3.display
4.exit

```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program to implement various operations of the stack is executed and the output is verified.

EXERCISES:

Write a C program to check whether a given string is a Palindrome or not using Stack.

EX.NO:

DATE :

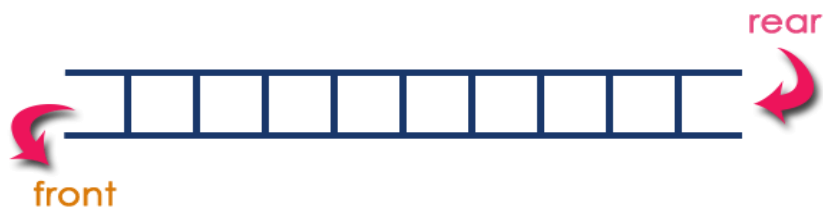
QUEUE USING ARRAY

AIM:

To write a C program to implement Queue and its various operations using array.

DESCRIPTION :

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure.
- The insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



After Inserting five elements...



Operations on a Queue

The following operations are performed on a queue data structure...

1. enQueue(value) - (To insert an element into the queue)
2. deQueue() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)

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
}
}
```

PROGRAM CODE:

```
# include<stdio.h>
#include<conio.h>
# define MAX 5
void inq();
void deq();
void display();
int queue[MAX];
int rear = -1,front = -1;
int item;
void main()
{
    int choice;
    clrscr();
    do
    {
        printf("\tMENU\n1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.EXIT\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1 :
                inq();
                display();
                break;

            case 2 :
                deq();
                display();
                break;

            case 3:
                display();
                break;

            case 4:
```

```

                                exit(0);

                        default:

                                printf("Wrong choice\n");
                                break;

                }
        }while(choice<=4);
        getch();
}

void inq()
{

        if(rear==MAX-1)                                // QUEUE IS EMPTY OR NOT
        {
                printf("Queue Overflow\n");
        }
        else
        {
                if (front==-1)                                // INSERTING THE ELEMENT IN THE QUEUE
                {
                        front=0;

                }
                printf("Enter the element: ");
                scanf("%d", &item);
                rear=rear+1;
                queue[rear] = item ;

        }
}

void deq()                                // REMOVING THE ELEMENTS FROM THE QUEUE
{
        if(front==-1||front>rear)
        {
                printf("Queue Underflow\n");
                getch();

        }
}

```

```

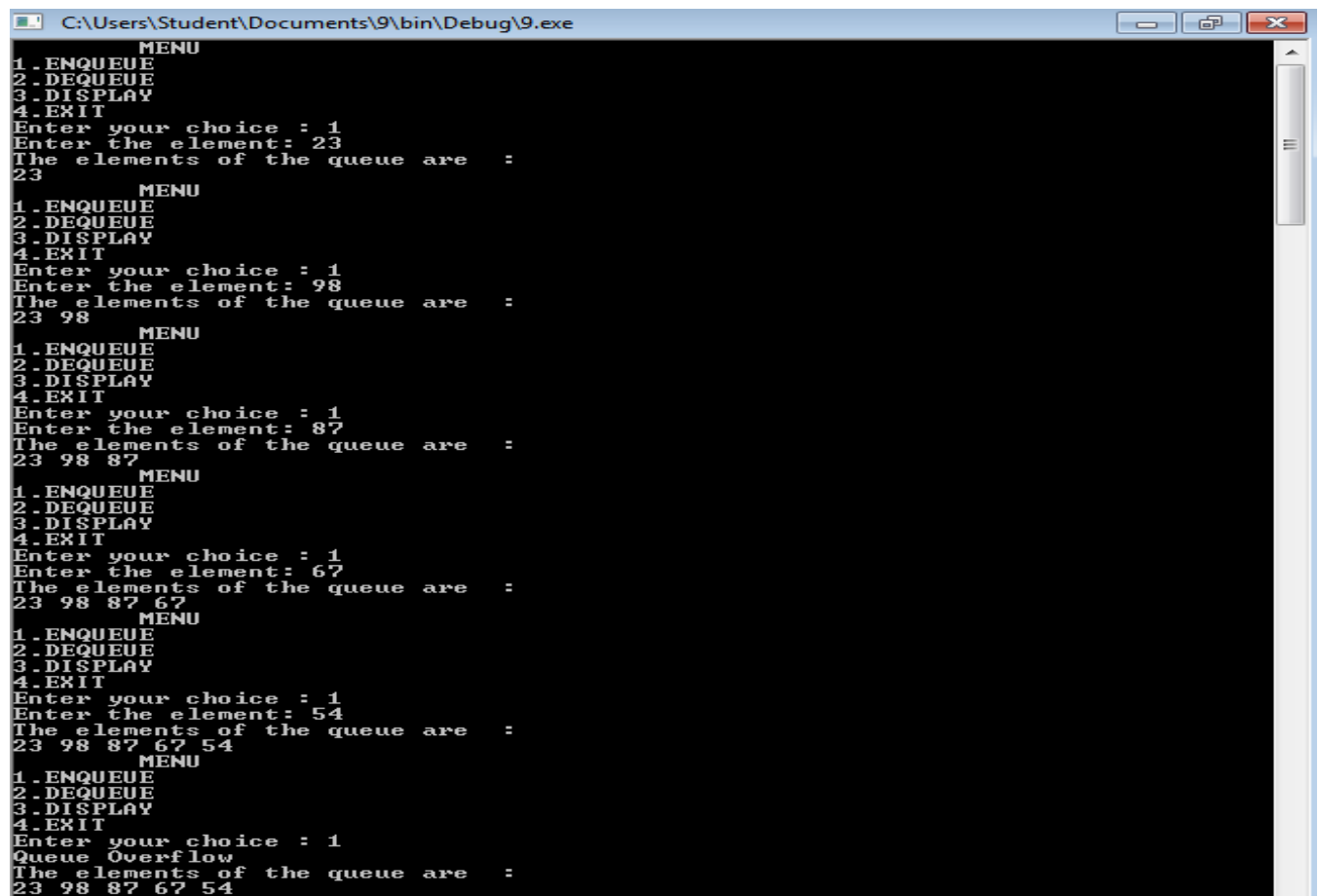
        else
        {
            printf("The Element deleted is : %d\n", queue[front]);
            front=front+1;
        }
    }

void display()                // DISPLAY THE ELEMENTS OF THE QUEUE
{
    int i;
    if(front==-1||front>rear)
    {
        printf("Queue is empty\n");
        getch();
    }

    else
    {
        printf("The elements of the queue are :\n");
        for(i=front;i<= rear;i++)
        {
            printf("%d ",queue[i]);
        }
        printf("\n");
    }
}

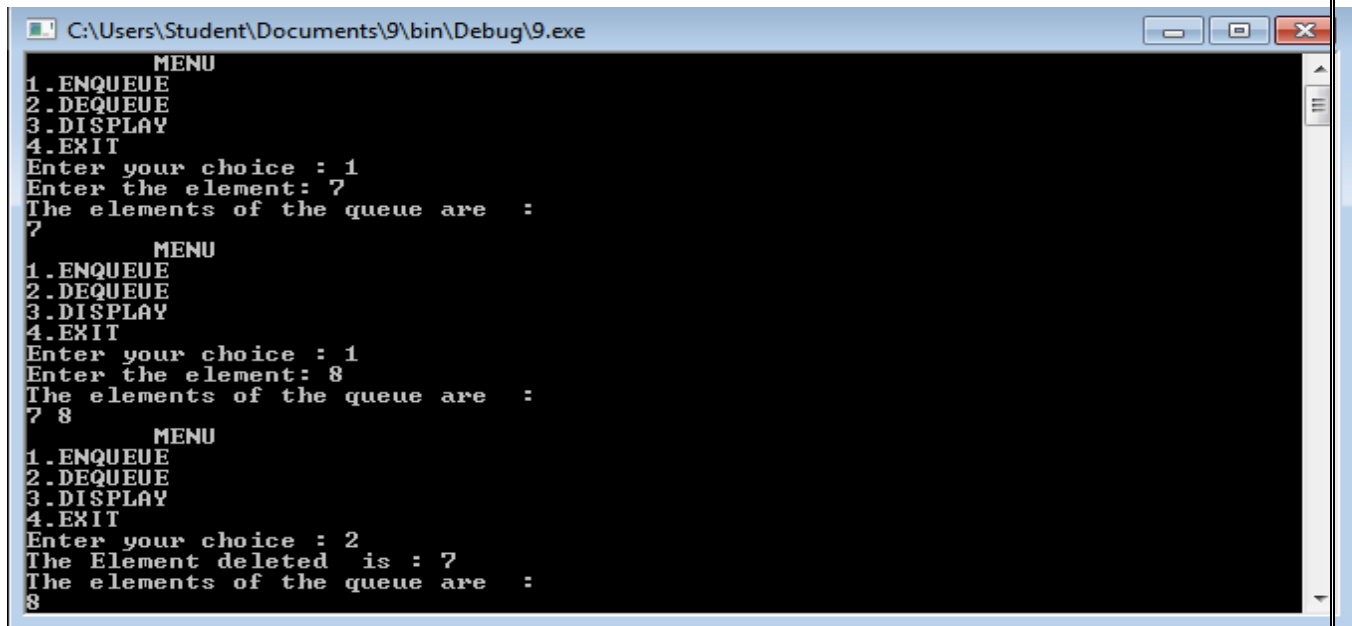
```


OUTPUT:



```

C:\Users\Student\Documents\9\bin\Debug\9.exe
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 23
The elements of the queue are :
23
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 98
The elements of the queue are :
23 98
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 87
The elements of the queue are :
23 98 87
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 67
The elements of the queue are :
23 98 87 67
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 54
The elements of the queue are :
23 98 87 67 54
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Queue Overflow
The elements of the queue are :
23 98 87 67 54
```



```
C:\Users\Student\Documents\9\bin\Debug\9.exe
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 7
The elements of the queue are :
7
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 1
Enter the element: 8
The elements of the queue are :
7 8
MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice : 2
The Element deleted is : 7
The elements of the queue are :
8
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT

Thus the C program is executed, the elements are inserted and deleted in the queue and the output is verified.

Exercises:

Write a C program to perform railway ticket reservation , where the number of people is standing in the linear queue. Implement the enqueue and dequeue operations of the user who are standing in the queue.

- Get the user name, age and ticket details as input and perform various operations.
- Inserting a new user in the queue.
- Deleting the Existing user from the queue.
- Display the details.

EX.NO:

DATE :

SINGLY LINKED LIST

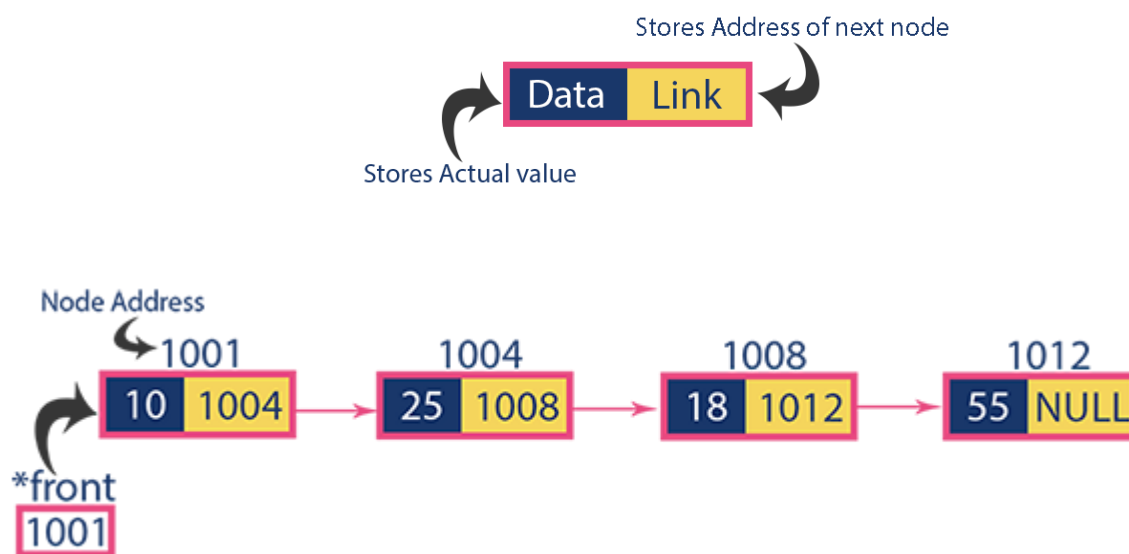
AIM

To develop a program to perform singly linked list operations.

DESCRIPTION:

- In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field.
- The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

The graphical representation of a node in a single linked list is as follows...



Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

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

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**).

PROGRAM CODE :

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

struct node                                // CREATING A NODE
{
    int info;
    struct node *link;
}*start;

void create(int);
void display();
void insert_end(int);
void insert_beg(int);
void insert_pos(int,int);
void delete_beg();
void delete_pos(int);
void delete_end();

main()
{
    int choice,data,choice1,choice2,pos,no_of_node;
    start=NULL;
    while(1)
    {
        clrscr();
        printf("1.Create\n");
        printf("2.Display\n");
        printf("3.Insert node\n");
        printf("4.Delete node\n");
        printf("5.Exit\n");
        printf("Enter ur choice:\n");
        scanf("%d",&choice);
        switch(choice)
        {
```

```

case 1: printf("Enter no. of node to be created:");
        scanf("%d",&no_of_node);
        create(no_of_node);
        break;
case 2: display();
        break;
case 3: clrscr();
        printf("1.Insert node at beginning\n");
        printf("2.Insert node at specific position\n");
        printf("3.Insert node at end of list\n");
        printf("4.Previous menu\n");
        printf("Enter choice:");
        scanf("%d",&choice1);
        switch(choice1)
        {
            case 1: printf("Enter data for node");
                    scanf("%d",&data);
                    insert_beg(data);
                    break;
            case 2: printf("Enter data for node:");
                    scanf("%d",&data);
                    printf("Enter the position to insert\n");
                    scanf("%d",&pos);
                    insert_pos(data,pos);
                    break;
            case 3:      printf("Enter data for node:");
                        scanf("%d",&data);
                        insert_end(data);
                        break;
            case 4: break;
        }break;
case 4: clrscr();
        printf("1.Delete node at beginning\n");
        printf("2.Delete node at specific position\n");

```

```

printf("3.Delete node at end of list\n");
printf("4.Previous menu\n");
printf("Enter choice:");
scanf("%d",&choice2);
switch(choice2)
{
    case 1:delete_beg();
        break;
    case 2:printf("Enter the position to insert\n");
        scanf("%d",&pos);
        delete_pos(pos);
        break;
    case 3:delete_end(data);
        break;
    case 4: break;
}break;
case 5: exit(1);
default: printf("Invalid entry");
}
}
}

```

```

void create(int no)                                // CREATING A NODE
{
    int i,data;
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    for(i=0;i<no;i++)
    {
        if(start==NULL)
        {

            printf("Enter data for node %d:",i);
            scanf("%d",&data);
            temp->info=data;

```

```

        temp->link=NULL;
        start=temp;
    }
    else
    {
        printf("Enter Data for node %d:",i);
        scanf("%d",&data);
        insert_end(data);
    }
}
if(no>0)
printf("List created");
else
printf("List not created");
getch();
clrscr();
}

void display()                                // DISPLAYING A NODE
{
    struct node *ptr;
    ptr=start;
    clrscr();
    if(start==NULL)
    {
        printf("List is empty");
        getch();
        return;
    }
    printf("Linked List\n");
    while(ptr!=NULL)
    {
        printf("%d->",ptr->info);
        ptr=ptr->link;
    }
}

```

```

    printf("End_of_list");
    getch();
    clrscr();
}
void insert_end(int data)                                // INSERTING AT THE END

```

```

{
    struct node *ptr,*tempnode;
    ptr=start;
    while(1)
    {
        if(ptr->link!=NULL)
            ptr=ptr->link;
        else
            break;
    }
    tempnode = (struct node *)malloc(sizeof(struct node));
    tempnode->info=data;
    tempnode->link=NULL;
    ptr->link=tempnode;
}

```

```

void insert_beg(int data)                                // INSERTING AT THE BEGINNING

```

```

{
    struct node *tempnode;
    tempnode = (struct node *)malloc(sizeof(struct node));
    tempnode->info=data;
    tempnode->link=start;
    start = tempnode;
}

```

```

void insert_pos(int data,int pos)

```

```

{
    int i;
    struct node *tempnode,*ptr;
    ptr=start;
    for(i=1;i<pos;i++)

```

```

        {
            if(ptr==NULL)
            {
                printf("Invalid Position Entered");
                getch();
                return;
            }
            ptr=ptr->link;
        }
        if(ptr->link==NULL)
        {
            insert_end(data);
        }
        else
        {
            temnode = (struct node *)malloc(sizeof(struct node));
            temnode->info=data;
            temnode->link=ptr->link;
            ptr->link = temnode;
        }
    }

void delete_beg()                                // DELETING AT THE BEGINNING
{
    struct node *ptr;
    ptr = start;
    if(start==NULL)
    {
        printf("List is empty");
        getch();
        return;
    }
    start = ptr->link;
    free(ptr);
}

```

```
void delete_pos(int pos)
```

```
// DELETING AT THE POSITION
```

```
{  
int i;  
    struct node *tempnode,*ptr;  
    ptr=start;  
    for(i=1;i<pos;i++)  
    {  
        if(ptr==NULL)  
        {  
            printf("Invalid Position Entered");  
            getch();  
            return;  
        }  
        ptr=ptr->link;  
    }  
    if(ptr->link==NULL)  
    {  
        delete_end();  
    }  
    else  
    {  
        tempnode = ptr->link;  
        ptr->link = ptr->link->link;  
        free(tempnode);  
    }  
}
```

```
void delete_end()
```

```
// DELETING AT END OF THE LIST
```

```
{  
    struct node *ptr,*prvptr;  
    ptr=start;  
    while(1)  
    {  
        if(ptr->link!=NULL)
```



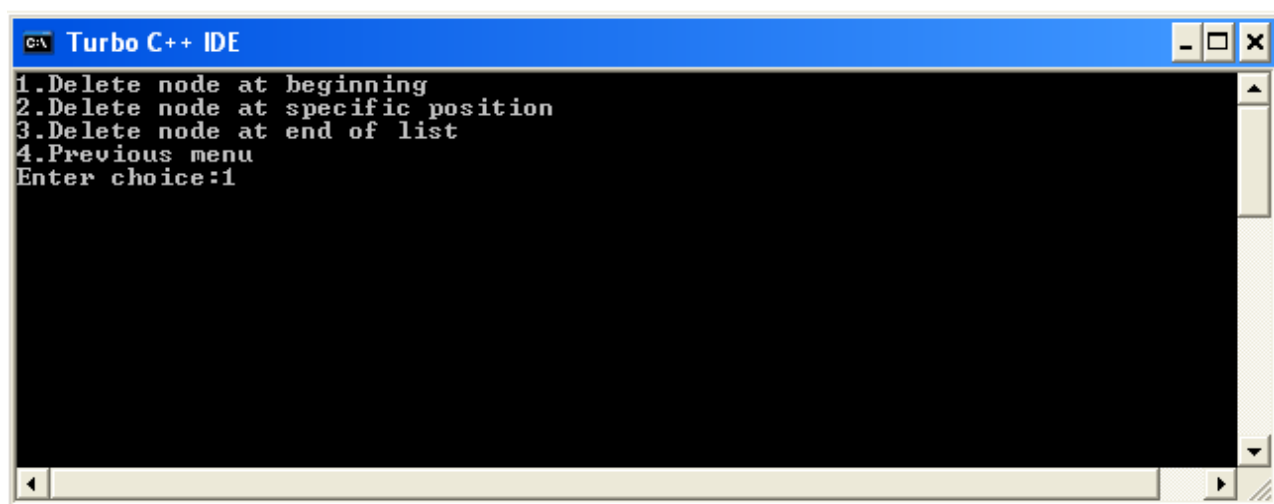
```
        {  
            prvptr=ptr;  
            ptr=ptr->link;  
        }  
        else  
            break;  
    }  
    prvptr->link=NULL;  
    free(ptr);  
}
```

OUTPUT:

```
Turbo C++ IDE
1.Create
2.Display
3.Insert node
4.Delete node
5.Exit
Enter ur choice:
1
Enter no. of node to be created:5
Enter data for node 0:5
Enter Data for node 1:8
Enter Data for node 2:10
Enter Data for node 3:15
Enter Data for node 4:25
List created
```

```
Turbo C++ IDE
Linked List
5->8->10->15->25->End_of_list_
```

```
Turbo C++ IDE
1.Insert node at beginning
2.Insert node at specific position
3.Insert node at end of list
4.Previous menu
Enter choice:1
Enter data for node50_
```



The image shows a screenshot of the Turbo C++ IDE window. The title bar reads "C:\ Turbo C++ IDE". The main text area contains a menu with four options: "1.Delete node at beginning", "2.Delete node at specific position", "3.Delete node at end of list", and "4.Previous menu". Below the menu, it says "Enter choice:1". The IDE has a standard Windows-style window with a blue title bar and a scroll bar on the right.

```
C:\ Turbo C++ IDE
1.Delete node at beginning
2.Delete node at specific position
3.Delete node at end of list
4.Previous menu
Enter choice:1
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the various operations of the linked list is been executed and the output is verified.

Exercises:

Write a C program to create a Single Linked List and perform the various operations which is listed below.

- Searching a node in the single linked list
- Sorting the values in the linked list in ascending order.
- Reversing the Elements In single linked list.

EX.NO:

DATE :

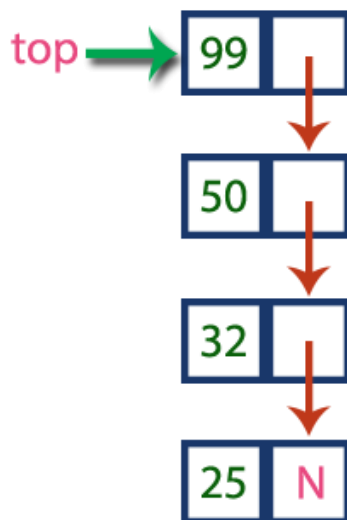
STACK USING LINKED LIST

AIM

To write a program to create a Stack and manipulate it using linked list

DESCRIPTION:

- A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. In linked list implementation of a stack, every new element is inserted as '**top**' element.
- That means every newly inserted element is pointed by '**top**'.
- Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list.
- The **next** field of the first element must be always **NULL**.



ALGORITHM

Stack Operations using Linked List

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**'.

PROGRAM CODE :

```
#include <stdio.h>

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

    struct node                // CREATING A NODE
    {
        int info;
        struct node *link;
    }*top = NULL;

int item;

void main()
{
    int ch;
    do
    {
        clrscr();
        printf("\n\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1: push();
                    break;
```

```

        case 2: pop();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("Invalid choice. Please try again.\n");
    }
} while(1);
getch();}

void push()                                // INSERTING A ELEMENT IN THE TOP OF THE LIST
{
    struct node *ptr;
    printf("\n\nEnter ITEM: ");
    scanf("%d", &item);
    if (top == NULL)
    {
        top = (struct node *)malloc(sizeof(struct node));
        top->info = item;
        top->link = NULL;
    }
    else
    {
        ptr = top;
        top = (struct node *)malloc(sizeof(struct node));
        top->info = item;
        top->link = ptr;
    }
    printf("\nItem inserted: %d\n", item);
}

void pop()                                // DELETING AN ELEMENT IN THE TOP OF THE LIST
{
    struct node *ptr;
    if (top == NULL)
        printf("\n\nStack is empty\n");

```

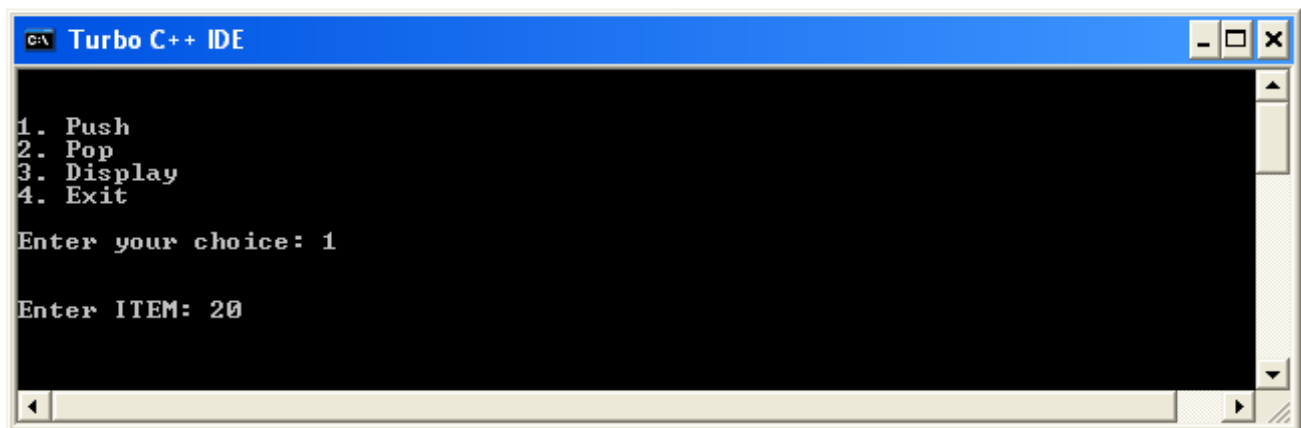
```

else
{
    ptr = top;
    item = top->info;
    top = top->link;
    free(ptr);
    printf("\n\nItem deleted: %d", item);
}
}

void display()                // DISPLAYING AN ELEMENT IN THE TOP OF THE LIST
{
    struct node *ptr;
    if (top == NULL)
        printf("\n\nStack is empty\n");
    else
    {
        ptr = top;
        while(ptr != NULL)
        {
            printf("\n\n%d", ptr->info);
            ptr = ptr->link;
        }
    }
    getch();
}

```

OUTPUT



The screenshot shows the Turbo C++ IDE window with a blue title bar. The main text area has a black background and displays the following output:

```

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 1

Enter ITEM: 20

```

The window includes standard Windows-style controls (minimize, maximize, close) in the top right corner and a scroll bar on the right side.

```
C:\ Turbo C++ IDE

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

50
40
30
20
10
```

```
C:\ Turbo C++ IDE

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

40
30
20
10_
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the PUSH and POP is been performed in the stack using the linked list and the output is verified.

EXERCISES:

Write a C program library information system. Where the user can PUSH and POP the book from the given list of books by performing stack operation in the linked list (LINKED STACK).

EX.NO:

DATE :

QUEUE USING LINKED LIST

AIM:

To write a program in C to implement Queue using Linked List.

DESCRIPTION:

- Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure.
- The queue which is implemented using a linked list can work for an unlimited number of values.
- That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).
- The Queue implemented using linked list can organize as many data values as we want.

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



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**'.

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
void inq();
void deq();
void display();
struct node                                // CREATING A NODE
{
    int data;
    struct node *link;
}*front=NULL,*rear=NULL;
int item;
void main()
{
    int n;
    clrscr();
    printf("\tMENU\n1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.EXIT\n");
    do
    {
        printf("\nEnter your choice\n");
        scanf("%d",&n);
        switch(n)
        {
            case 1:
                inq();
                display();
                break;

            case 2:
                deq();
                display();
                break;

            case 3:
```

```

        display();
        break;
    case 4:
        exit(0);
    default:
        printf("Invalid choice\n");
        break;
    }
}
while(n<=4);
getch();
}

void inq()                                // PERFORMING ENQUEUE OPERATION
{
    struct node *temp;
    printf("Enter the item\n");
    scanf("%d",&item);
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=item;
    temp->link=NULL;
    if(rear==NULL)
    {
        front=temp;
        rear=temp;
    }
    else
    {
        rear->link=temp;
        rear=temp;
    }
}

void deq()                                // PERFORMING DEQUEUE OPERATION
{
    int item;

```

```

    if(front==NULL)
    {
        printf("Queue is empty\n");
    }
    else
    {
        item=front->data;
        printf("The element deleted = %d\n",item);
    }
    if(front==rear)
    {
        front=NULL;
        rear=NULL;
    }
    else
    {
        front=front->link;
    }
}

void display()                                // PERFORMING DISPLAY OPERATION
{
    struct node *ptr;
    if(front==NULL)
    {
        printf("Queue is empty\n");
    }
    else
    {
        ptr=front;
        printf("The elements of the queue are :\n");
        while(ptr!=NULL)
        {
            printf("%d\t",ptr->data);
            ptr=ptr->link;
        }
    }
}

```

```

    }
}
}

```

OUTPUT:

```

      MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT

Enter your choice
1
Enter the item
20
The elements of the queue are :
20
Enter your choice
1
Enter the item
30
The elements of the queue are :
20      30
Enter your choice

```

```

      MENU
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT

Enter your choice
1
Enter the item
20
The elements of the queue are :
20
Enter your choice
1
Enter the item
30
The elements of the queue are :
20      30
Enter your choice
2
The element deleted = 20
The elements of the queue are :
30
Enter your choice

```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT

Thus the C program to implement the queue operation using linked list is implemented and the output is verified.

EXERCISES:

Write a C program to perform various operation using the Circular Linked List.

- Inserting a node in the beginning of the list.
- Deleting a node in the beginning of the list.
- Inserting a node at the specific position of the list.
- Checking the list is empty.
- Deleting a node at the specific position of the list.
- Finding the total elements in the List.

EX.NO:

DATE :

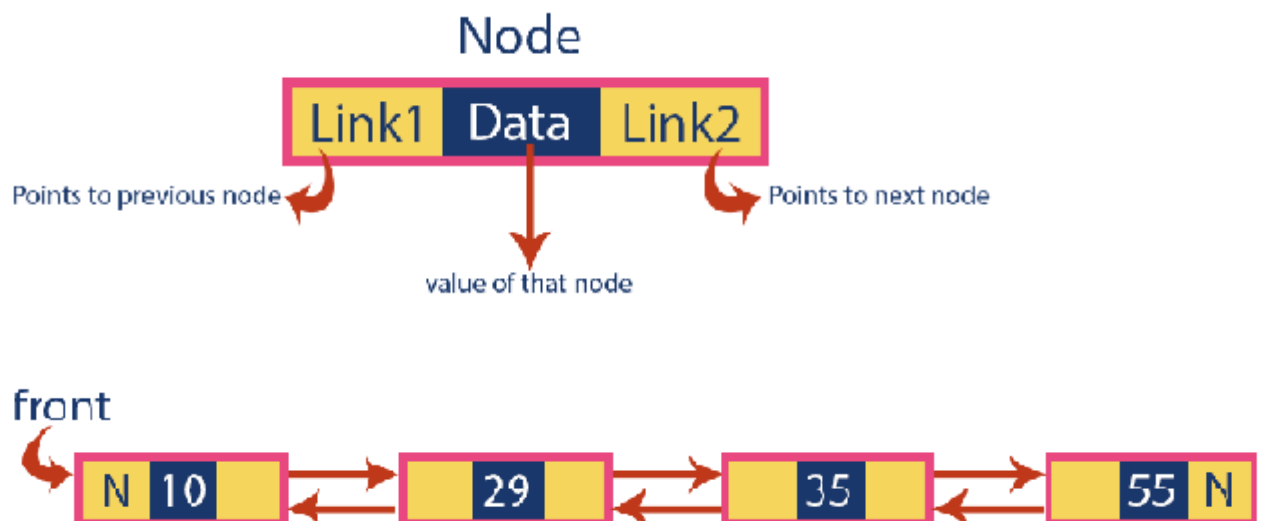
DOUBLY LINKED LIST

AIM:

To write a program in C to implement Doubly Linked List.

DESCRIPTION:

- In a double linked list, every node has a link to its previous node and next node.
- So, we can traverse forward by using the next field and can traverse backward by using the previous field.
- Every node in a double linked list contains three fields and they are shown in the following figure...



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

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 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 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**).

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<malloc.h>
struct node
{
int info;
struct node *nxt,*prv;
}*start;
void create(int);
void display();
void insert_end(int);
void insert_beg(int);
void insert_pos(int,int);
void delete_beg();
void delete_end();
void delete_pos(int);
main()
{
int ch,data,ch1,ch2,nof,pos;
start=NULL;
while(1)
{
clrscr();
printf("\n1.Create\n2.Display\n3.Insert\n4.Delete\n5.Exit\n");
printf("\nEnter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nNumber of nodes to be created\n");
scanf("%d",&nof);
create(nof);
```

```

break;
case 2:
display();
break;
case 3:
clrscr();
printf("\n1.Insert node at beginning\n2.Insert node at specific position\n3.Insert node at
end\n4.Previous menu\n");
printf("\nEnter your choice\t");
scanf("%d",&ch1);
switch(ch1)
{
case 1:
printf("\nEnter the data for node\n");
scanf("%d",&data);
insert_beg(data);
break;
case 2:
printf("\nEnter the data for node\n");
scanf("%d",&data);
printf("\nEnter the position to insert\n");
scanf("%d",&pos);
insert_pos(data,pos);
break;
case 3:
printf("\nEnter the data for node\n");
scanf("%d",&data);
insert_end(data);
break;
case 4:
break;
}break;
case 4:
clrscr();

```

```

printf("\n1.Delete node at beginning\n2.Delete node at specific position\n3.Delete node at
end\n4.Previous menu\n");
printf("\nEnter your choice\t");
scanf("%d",&ch2);
switch(ch2)
{
case 1:
    delete_beg();
break;
case 2:
    printf("\nEnter the position to delete\n");
    scanf("%d",&pos);
    delete_pos(pos);
break;
case 3:
    delete_end();
break;
    case 4:
break;
}break;
    case 5:
    exit(1);
default:
printf("Invalid entry");
}
}
}
void create(int no)
{
int i,data;
struct node *temp;
clrscr();
temp=(struct node *)malloc(sizeof(struct node));
for(i=0;i<no;i++)

```

```

{
if(start==NULL)
{
    printf("\nEnter the data for node %d:",i);
    scanf("%d",&data);
    temp->info=data;
    temp->nxt=NULL;
    temp->prv=NULL;
    start=temp;
}
else
{
    printf("\nEnter the data for node %d:",i);
    scanf("%d",&data);
    insert_end(data);
}
}
if(no>0)
    printf("List created");
else
    printf("List is not created");
    getch();
clrscr();
}
void display()
{
    struct node *ptr;
    ptr=start;
    clrscr();
    if(start==NULL)
    {
        printf("\nList is empty");
        getch();
    }
    return;
}

```



```

}
printf("\nLinked list");
while(ptr!=NULL)
{
    printf("\t%d->",ptr->info);
    ptr=ptr->nxt;
}
printf("\nEnd of list");
getch();
clrscr();
}
void insert_end(int data)
{
    struct node *ptr,*tempnode;
    ptr=start;
    while(1)
    {
        if(ptr->nxt!=NULL)
            ptr=ptr->nxt;
        else
            break;
    }
    tempnode=(struct node *)malloc(sizeof(struct node));
    tempnode->info=data;
    tempnode->nxt=NULL;
    tempnode->prv=ptr;
    ptr->nxt=tempnode;
}
void insert_beg(int data)
{
    struct node *tempnode;
    tempnode=(struct node *)malloc(sizeof(struct node));
    tempnode->info=data;
    tempnode->nxt=start;

```

```

start=tempnod;
}
void insert_pos(int data,int pos)
{
int i;
struct node *tempnod,*ptr;
ptr=start;
for(i=1;i<pos;i++)
{
    if(ptr==NULL)
    {
        printf("Invalid position");
        getch();
        return;
    }
ptr=ptr->nxt;
}
if(ptr->nxt==NULL)
insert_end(data);
else
{
    tempnod=(struct node *)malloc(sizeof(struct node));
    tempnod->info=data;
    tempnod->nxt=ptr->nxt;
    tempnod->prv=ptr;
    ptr->nxt=tempnod;
    (tempnod->nxt)->prv=tempnod;
}
}
void delete_beg()
{
struct node*ptr;
ptr=start;
if(start==NULL)

```

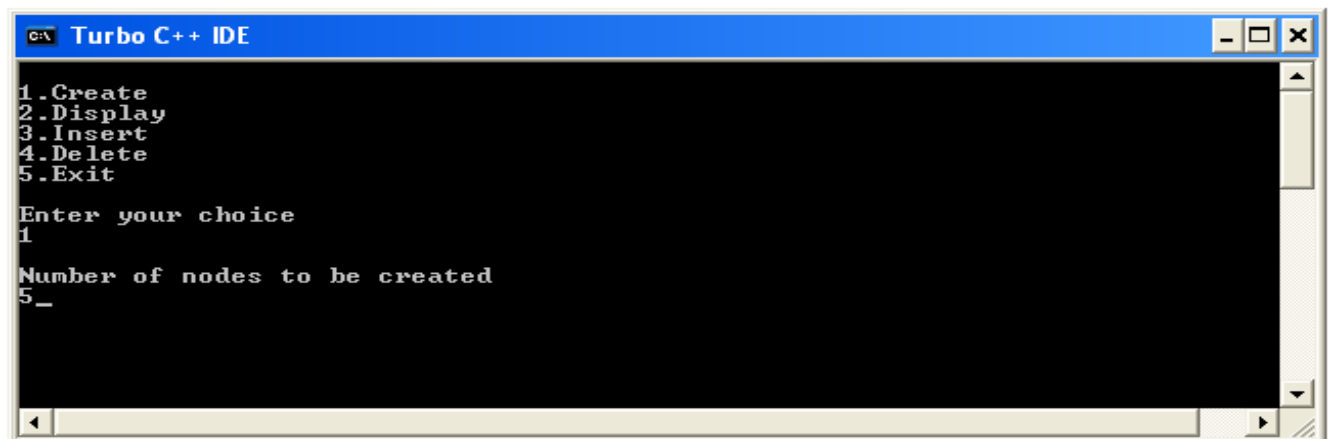
```

{
printf("List is empty");
getch();
return;
}
start=ptr->nxt;
free(ptr);
}
void delete_pos(int pos)
{
int i;
struct node *tempnode,*ptr;
ptr=start;
for(i=1;i<pos;i++)
{
if(ptr==NULL)
{
printf("Invalid position");
getch();
return;
}
ptr=ptr->nxt;
}
if(ptr->nxt==NULL)
delete_end();
else
{
tempnode=ptr->nxt;
ptr->nxt=tempnode->nxt;
free(tempnode);
}
}
void delete_end()
{

```

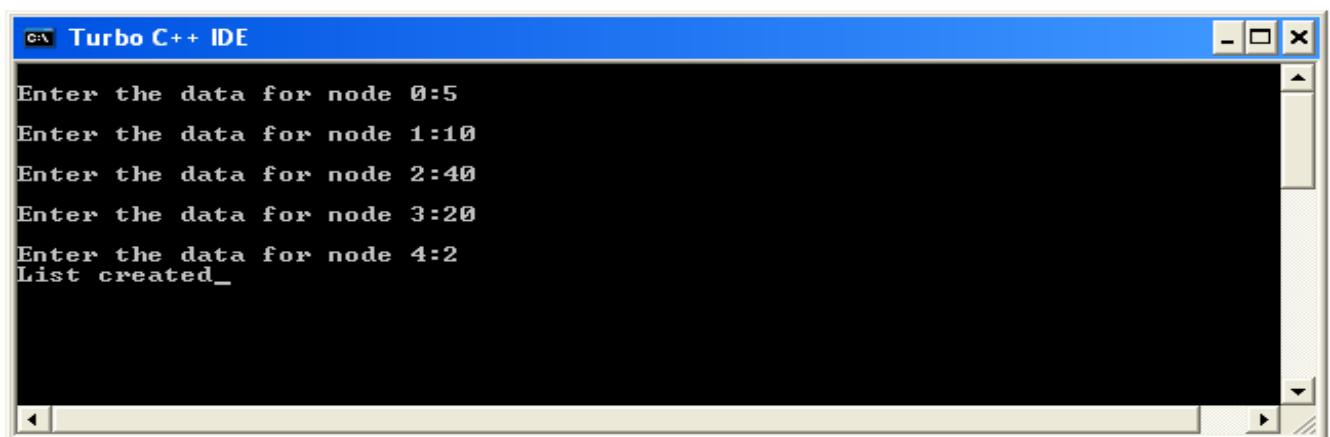
```
struct node *ptr;
ptr=start;
while(1)
{
if(ptr->nxt!=NULL)
{
    ptr=ptr->nxt;
}
else
break;
}
(ptr->prv)->nxt=NULL;
free(ptr);
}
```

OUTPUT



The screenshot shows the Turbo C++ IDE window with a blue title bar. The text area contains a menu with five options: 1.Create, 2.Display, 3.Insert, 4.Delete, and 5.Exit. Below the menu, it prompts the user to 'Enter your choice'. The user has entered '1'. Then, it prompts for 'Number of nodes to be created' and the user has entered '5_'. The IDE has a black background and a yellow border.

```
C:\ Turbo C++ IDE
1.Create
2.Display
3.Insert
4.Delete
5.Exit
Enter your choice
1
Number of nodes to be created
5_
```



The screenshot shows the Turbo C++ IDE window with a blue title bar. The text area shows the program prompting for data for five nodes. The prompts are: 'Enter the data for node 0:5', 'Enter the data for node 1:10', 'Enter the data for node 2:40', 'Enter the data for node 3:20', and 'Enter the data for node 4:2'. The user has entered the data for each node. The final prompt is 'List created_'. The IDE has a black background and a yellow border.

```
C:\ Turbo C++ IDE
Enter the data for node 0:5
Enter the data for node 1:10
Enter the data for node 2:40
Enter the data for node 3:20
Enter the data for node 4:2
List created_
```

```
Turbo C++ IDE
Linked list      5->      10->      40->      20->      2->
End of list_
```

```
Turbo C++ IDE
1.Insert node at beginning
2.Insert node at specific position
3.Insert node at end
4.Previous menu
Enter your choice      2
Enter the data for node
22
Enter the position to insert
3_
```

```
Turbo C++ IDE
1.Delete node at beginning
2.Delete node at specific position
3.Delete node at end
4.Previous menu
Enter your choice      2
Enter the position to delete
4
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT

Thus the C program for the double linked list and its operations is executed and the output is verified.

EXERCISES:

Write a C program for creating a employee records, where the employee id can be inserted and deleted in the given list by using double linked list.

- Attributes to be created Emp_name , Emp_no, Emp_designation.
- Display the total no of employees.
- Display the no of employees deleted from the list.

EX.NO:

DATE :

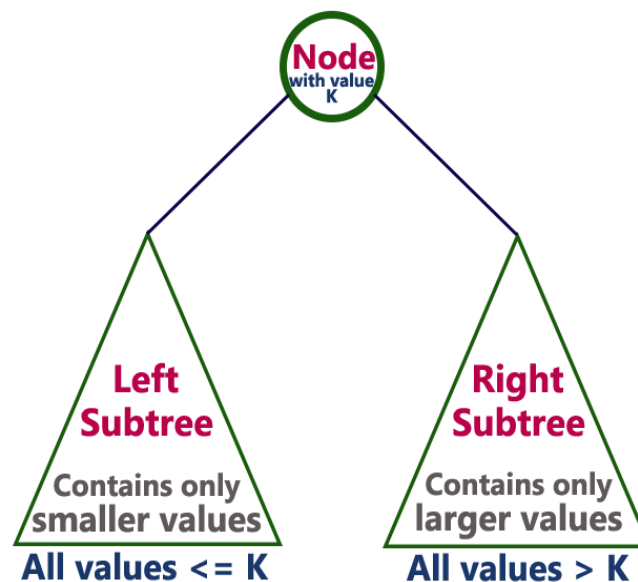
BINARY SEARCH TREE

AIM:

To develop a program to implement Binary Search.

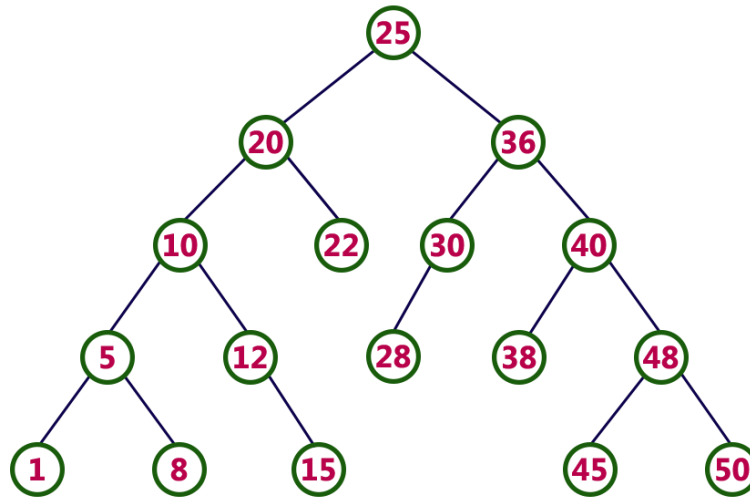
DESCRIPTION:

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

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.

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
void inorder(struct node *root)
{
    if(root)
    {
        inorder(root->left);
        printf(" %d",root->data);
        inorder(root->right);
    }
}
int main()
{
    int n , i;
    struct node *p , *q , *root;
    printf("Enter the number of nodes to be insert: ");
    scanf("%d",&n);
    printf("\nPlease enter the numbers to be insert: ");
    for(i=0;i<i++)
    {
        p = (struct node*)malloc(sizeof(struct node));
        scanf("%d",&p->data);
        p->left = NULL;
        p->right = NULL;
        if(i == 0)
        {
            root = p; // root always point to the root node
        }
        else
        {
            q = root; // q is used to traverse the tree
```

```

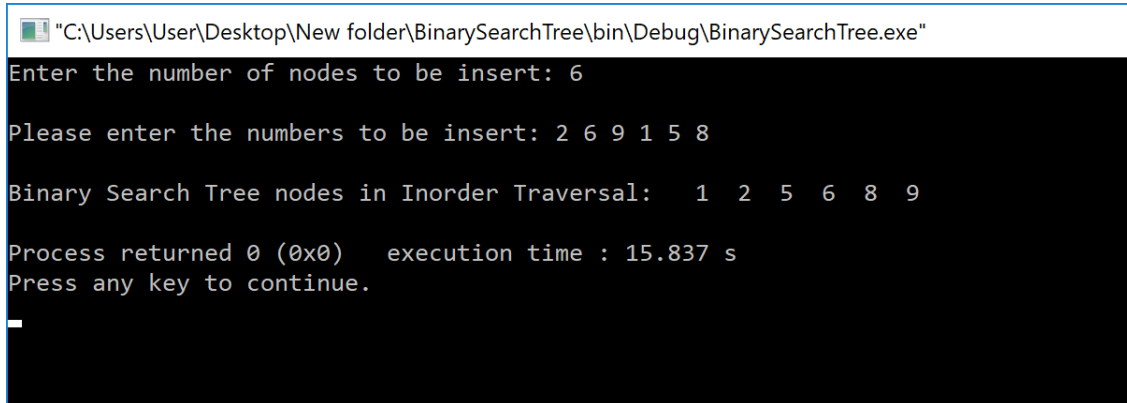
        while(1)
        {
            if(p->data > q->data)
            {
                if(q->right == NULL)
                {
                    q->right = p;
                    break;
                }
                else
                    q = q->right;
            }
            else
            {
                if(q->left == NULL)
                {
                    q->left = p;
                    break;
                }
                else
                    q = q->left;
            }
        }
    }

}

printf("\nBinary Search Tree nodes in Inorder Traversal: ");
inorder(root);
printf("\n");
return 0;
}

```

OUTPUT:



```
"C:\Users\User\Desktop\New folder\BinarySearchTree\bin\Debug\BinarySearchTree.exe"
Enter the number of nodes to be insert: 6
Please enter the numbers to be insert: 2 6 9 1 5 8
Binary Search Tree nodes in Inorder Traversal: 1 2 5 6 8 9
Process returned 0 (0x0) execution time : 15.837 s
Press any key to continue.
_
```

| CONTENTS | MARKS ALLOTTED | MARKS OBTAINED |
|-----------------------|----------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program to implement the binary search tree is implemented and the output is verified.

EXERCISES :

Write a C program for the binary search tree

- a) Search a particular node in the binary search tree
- b) Find the minimum node in the binary search tree.
- c) Find the maximum node in the binary search tree.

EX.NO:

DATE :

BINARY TREE TRAVERSAL

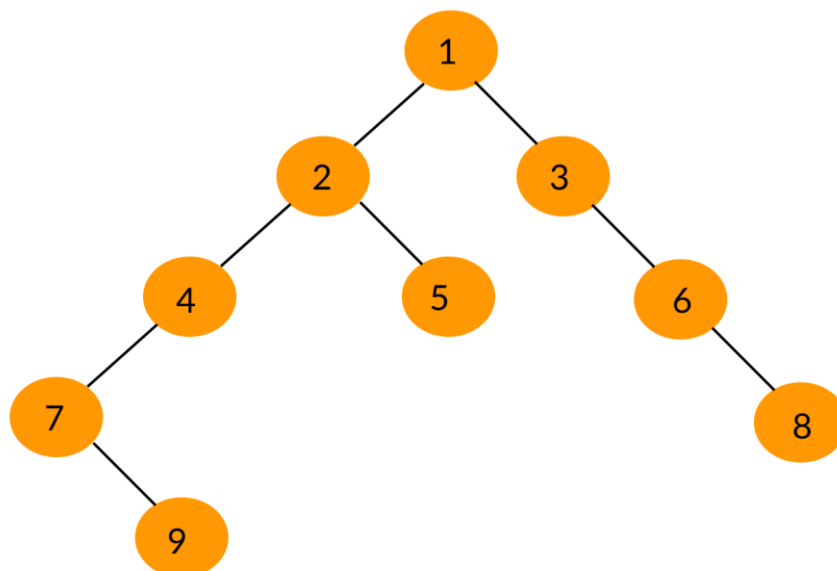
AIM

A program to create a Binary tree and perform the 3 different types of traversals.
Inorder, Preorder and Postorder

ALGORITHM

- Binary tree Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree
 - Inorder traversal
 - Preorder traversal
 - Postorder traversal

Consider the given binary tree,



Inorder Traversal: For binary search trees (BST), Inorder Traversal specifies the nodes in non-descending order. In order to obtain nodes from BST in non-increasing order, a variation of inorder traversal may be used where inorder traversal is reversed.

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: Preorder traversal will create a copy of the tree. Preorder Traversal is also used to get the prefix expression of an expression.

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Preorder Traversal: 1 2 4 7 9 5 3 6 8

Postorder Traversal: Postorder traversal is used to get the postfix expression of an expression given

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

Postorder Traversal: 9 7 4 5 2 8 6 3 1

ALGORITHM:

PREORDER

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;  
}
```

PROGRAM CODE:

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

struct node                                // CREATING A NODE
{
    int info;
    struct node *lchild;
    struct node *rchild;
} *root, *p, *q;

typedef struct node NODE;
struct node *make(int y)
{
    struct node *newnode;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->info=y;
    newnode->rchild=newnode->lchild=NULL;
    return(newnode);
}

void left(struct node*r,int x)
{
    if(r->lchild!=NULL)
        printf("\n invalid");
    else
        r->lchild=make(x);
}

void right(struct node*r,int x)
{
    if(r->rchild!=NULL)
        printf("\n invalid");
    else
        r->rchild=make(x);
}

void insertBST(struct node *r,int no)
```



```

{
    printf("Enter another element");
    scanf("%d",&no);
    p=r;
    q=r;
    while(no!=p->info&&q!=NULL)
        {
            p=q;
            if(no<p->info)
                q=p->lchild;
            else
                q=p->rchild;
        }
    if(no<p->info)
        {
            printf("\n left branch of %dis %d",p->info,no);
            left(p,no);
        }
    else
        {
            right(p,no);
            printf("\n right branch of %d is %d",p->info,no);
        }
    }
NODE *search_BST(int elt,NODE*T)
{
    if(T==NULL)
        return NULL;
    else if(elt<T->info)
        {
            return search_BST(elt,T->lchild);
        }

    else if(elt>T->info)

```

```

        {
            return search_BST(elt,T->rchild);
        }
    else
        return T;
    }
NODE *findmin(NODE *T)
{
    if(T==NULL)
        return NULL;
    if(T->lchild==NULL)
        return T;
    else
        return findmin(T->lchild);
}
NODE *delete_BST(int elt, NODE*T)
{
    NODE *minelt;
    if(T==NULL)
        printf("elt not found");
    else if (elt<T->info)
        T->lchild=delete_BST(elt,T->lchild);
    else if(elt>T->info)
        T->rchild=delete_BST(elt,T->rchild);
    else if(T->lchild!=NULL && T->rchild!=NULL)
    {
        minelt=findmin(T->rchild);
        T->info=minelt->info;
        T->rchild=delete_BST(T->info,T->rchild);
    }

    else
    {

```

```

        minelt=T;
        if(T->lchild==NULL)
            T=T->rchild;
        if(T->rchild==NULL)
            T=T->lchild;
        free(minelt);
    }
    return T;
}

void inorder_traversal(NODE *T)
{
    if(T!=NULL)
    {
        inorder_traversal(T->lchild);
        printf("%d->",T->info);
        inorder_traversal(T->rchild);
    }
}

void preorder_traversal(NODE *T)
{
    if(T!=NULL)
    {
        printf("%d->",T->info);
        preorder_traversal(T->lchild);
        preorder_traversal(T->rchild);
    }
}

void postorder_traversal(NODE *T)
{
    if(T!=NULL)
    {
        postorder_traversal(T->lchild);
        postorder_traversal(T->rchild);
    }
}

```

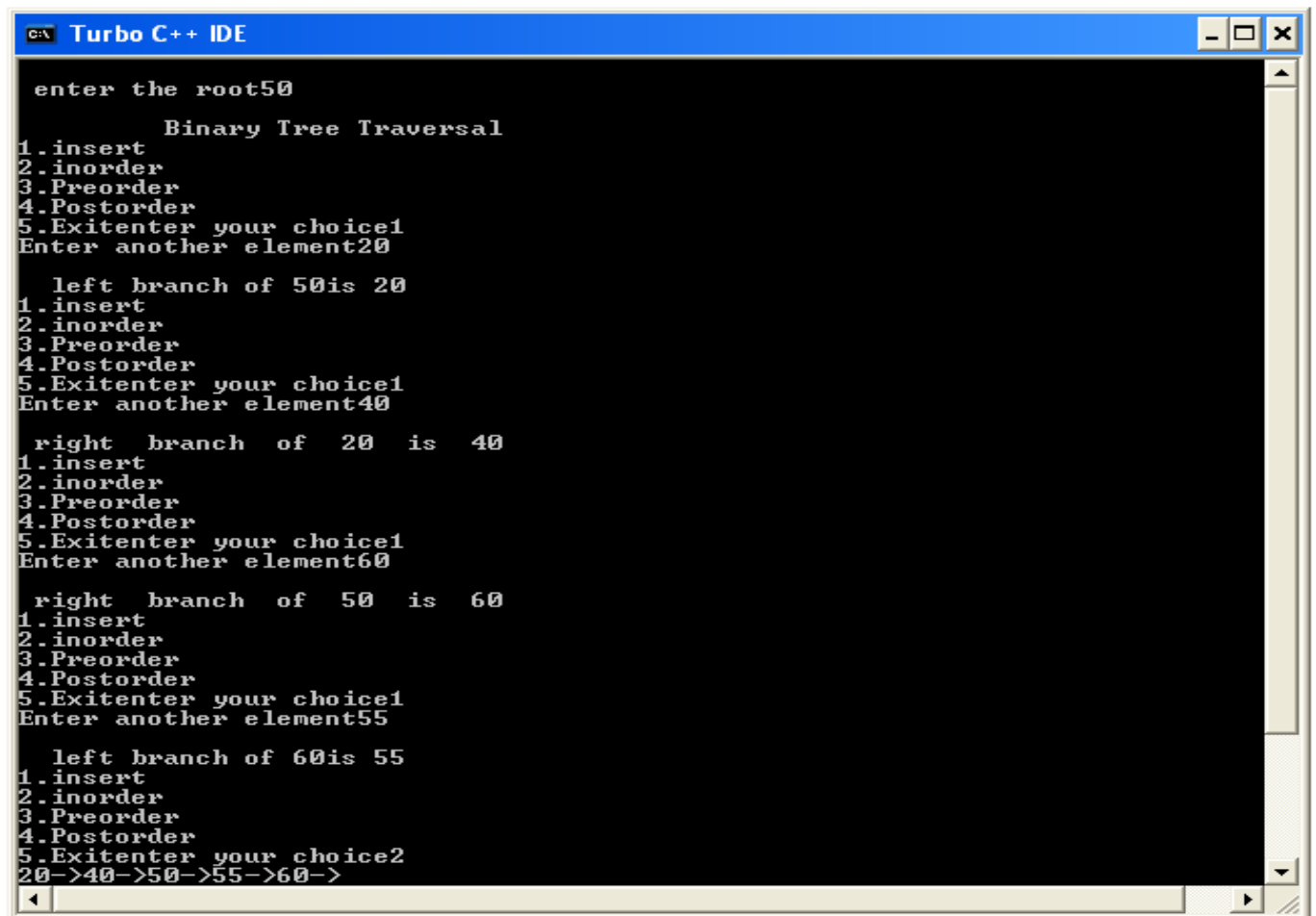
```

        printf("%d->",T->info);
    }
}
void main()
{
    int ch,elt,no;
    struct node *ptr,*T;
    clrscr();
    printf("\n enter the root");
    scanf("%d",&no);
    root=make(no);
    p=root;
    printf("\n\t Binary Tree Traversal");
    do
    {
        printf("\n1.insert");
        printf("\n2.inorder");
        printf("\n3.Preorder");
        printf("\n4.Postorder");
        printf("\n5.Exit");
        printf("enter your choice");
        scanf("%d",&ch);
        switch (ch)
        {
            case 1:
                insertBST(root,elt);
                break;
            case 2:
                inorder_traversal(root);
                break;
            case 3:
                preorder_traversal(root);
                break;
            case 4:

```

```
        postorder_traversal(root);
    break;
    case 5:
    exit(1);
    break;
    default :printf("ivalid choice");
}
}while(ch!=5);
getch();
}
```

OUTPUT:



```
Turbo C++ IDE

enter the root50

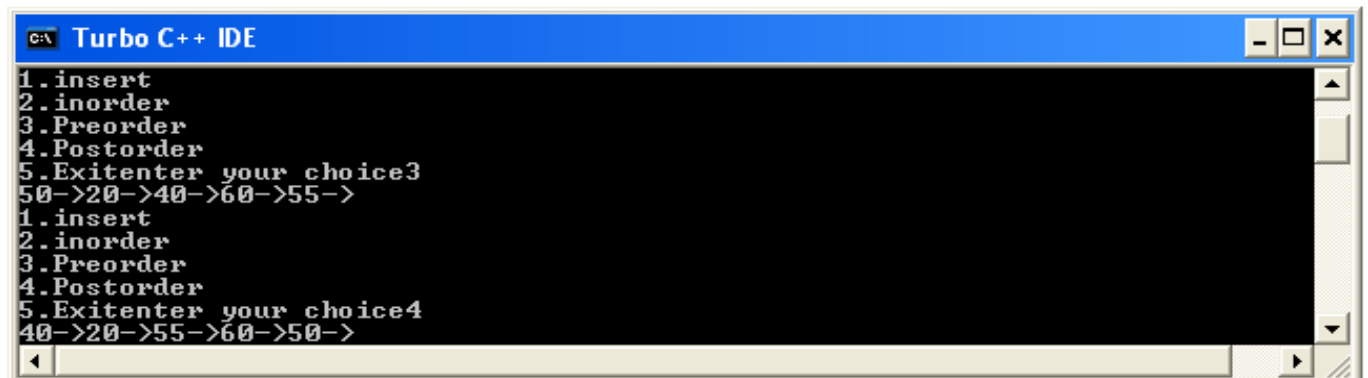
      Binary Tree Traversal
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice1
Enter another element20

      left branch of 50is 20
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice1
Enter another element40

      right branch of 20 is 40
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice1
Enter another element60

      right branch of 50 is 60
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice1
Enter another element55

      left branch of 60is 55
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice2
20->40->50->55->60->
```



```
Turbo C++ IDE

1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice3
50->20->40->60->55->
1.insert
2.inorder
3.Preorder
4.Postorder
5.Exitenter your choice4
40->20->55->60->50->
```

| CONTENTS | MARKS ALLOTTED | MARKS OBTAINED |
|-----------------------|----------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program to implement the different traversal technique is executed and the output is verified.

EXERCISES:

Write a C Program to insert a node in a binary tree and to perform the

- I) INORDER TRAVERSAL
- II) PREORDER TRAVERSAL
- III) POST ORDER TRAVERSAL.

EX.NO:

DATE :

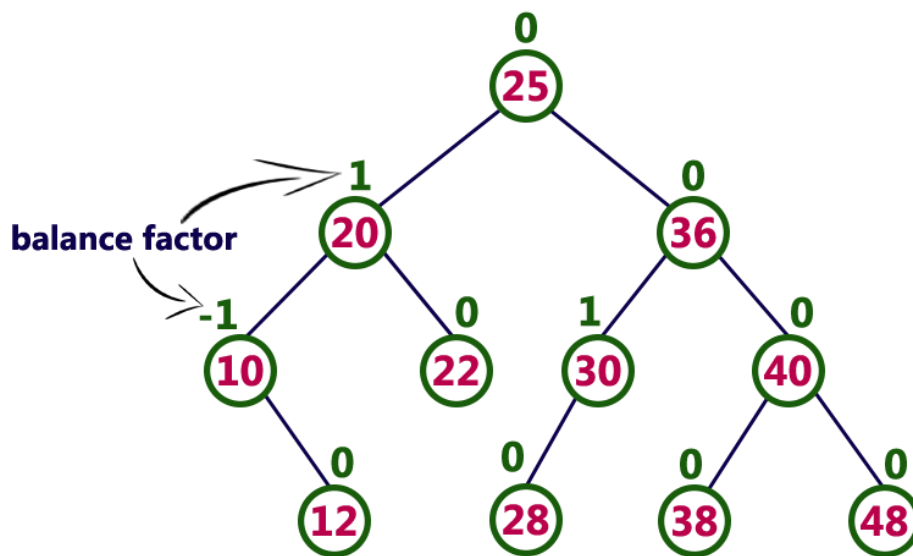
AVL TREE

AIM

To develop a program to implement AVL Tree.

DESCRIPTION:

- AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1.
- In an AVL tree, every node maintains an extra information known as balance factor.
- The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.
- Balance factor of a node is the difference between the heights of the left and right subtrees of that node.
- The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion.

ALGORITHM:

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.

PROGRAM CODE:

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

```
#include<stdlib.h>
```

```
// An AVL tree node
```

```
struct Node
```

```
{
```

```
    int key;
```

```
    struct Node *left;
```

```
    struct Node *right;
```

```
    int height;
```

```
};
```

```
// A utility function to get maximum of two integers
```

```
int max(int a, int b);
```

```
// A utility function to get the height of the tree
```

```
int height(struct Node *N)
```

```
{
```

```
    if (N == NULL)
```

```
        return 0;
```

```
    return N->height;
```

```
}
```

```
// A utility function to get maximum of two integers
```

```
int max(int a, int b)
```

```
{
```

```
    return (a > b)? a : b;
```

```
}
```

```
/* Helper function that allocates a new node with the given key and
```

```
    NULL left and right pointers. */
```

```
struct Node* newNode(int key)
```

```
{
```

```
    struct Node* node = (struct Node*)
```

```
        malloc(sizeof(struct Node));
```

```
    node->key  = key;
```

```
    node->left = NULL;
```

```
    node->right = NULL;
```

```
node->height = 1; // new node is initially added at leaf
return(node);
}
```

// A utility function to right rotate subtree rooted with y

// See the diagram given above.

```
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
    // Return new root
    return x;
}
```

// A utility function to left rotate subtree rooted with x

// See the diagram given above.

```
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;
    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    // Return new root
    return y;
}
```

// Get Balance factor of node N

```

int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.

struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                          height(node->right));

    /* 3. Get the balance factor of this ancestor
node to check whether this node became
unbalanced */

    int balance = getBalance(node);
    // If this node becomes unbalanced, then
// there are 4 cases
// Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
// Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

```

// Left Right Case

```
if (balance > 1 && key > node->left->key)
{
    node->left = leftRotate(node->left);
    return rightRotate(node);
}
```

// Right Left Case

```
if (balance < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

/* return the (unchanged) node pointer */

```
return node;
```

```
}
```

// A utility function to print preorder traversal**// of the tree.****// The function also prints height of every node**

```
void preOrder(struct Node *root)
```

```
{
```

```
    if(root != NULL)
```

```
    {
```

```
        printf("%d ", root->key);
```

```
        preOrder(root->left);
```

```
        preOrder(root->right);
```

```
    }
```

```
}
```

/* Driver program to test above function*/

```
int main()
```

```
{
```

```
    struct Node *root = NULL;
```

/* Constructing tree given in the above figure */

```
    root = insert(root, 10);
```

```
    root = insert(root, 20);
```

```

root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);
/* The constructed AVL Tree would be
        30
       / \
      20 40
     / \  \
    10 25 50
*/
printf("Preorder traversal of the constructed AVL  " tree is \n");
preOrder(root);
return 0;
}

```


OUTPUT:

Preorder traversal of the constructed AVL tree is

30 20 10 25 40 50

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program is been exceuted and the output is verified.

EXERCISES:

Write a C program to insert the elements in the AVL tree in the Following order.

Mar,may,nov,aug,apr,jan,dec,jul,feb,jun,oct,sep

EX.NO:

DATE :

GRAPH TRAVERSAL – DEPTH FIRST SEARCH

AIM

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

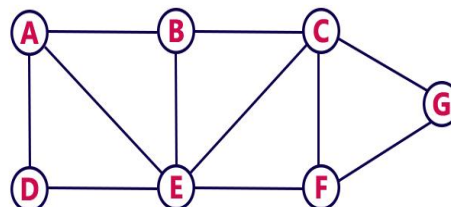
DESCRIPTION:

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

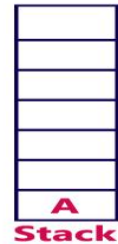
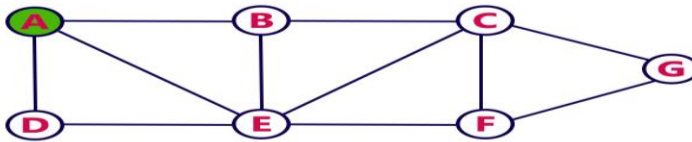
- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph
- **Back tracking** is coming back to the vertex from which we reached the current vertex.

Consider the following example graph to perform DFS traversal

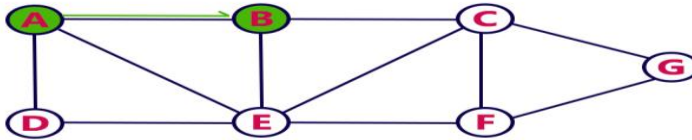


Step 1:

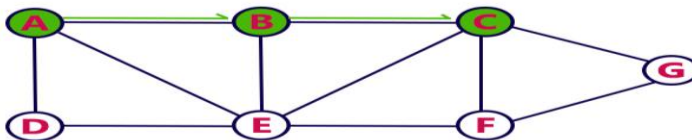
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

**Stack****Step 2:**

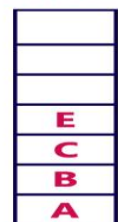
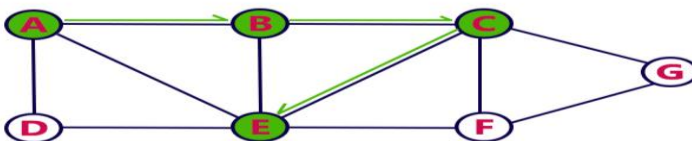
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

**Stack****Step 3:**

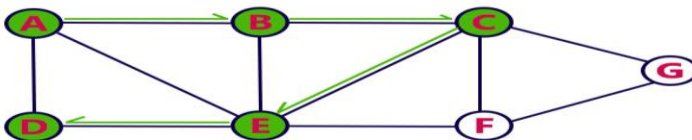
- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.

**Stack****Step 4:**

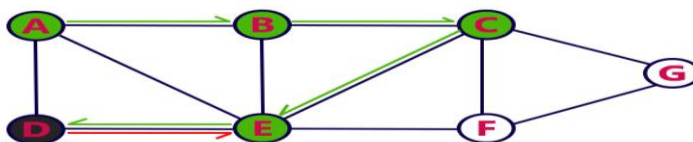
- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack.

**Stack****Step 5:**

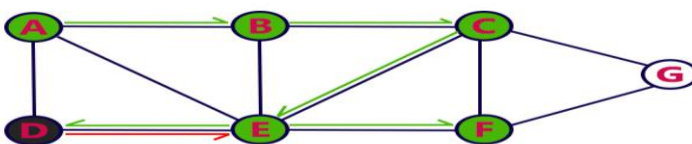
- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack.

**Stack****Step 6:**

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.

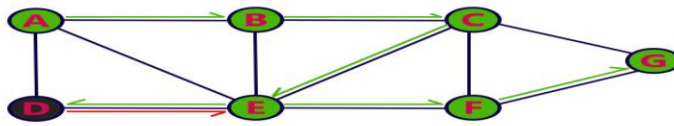
**Stack****Step 7:**

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

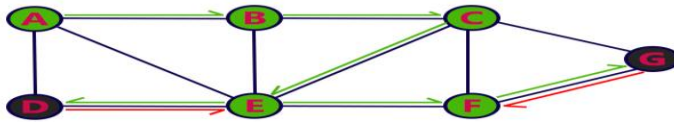
**Stack**

Step 8:

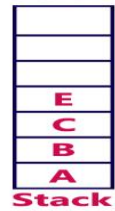
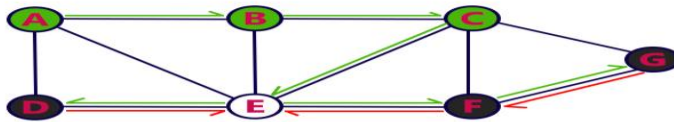
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

**Step 9:**

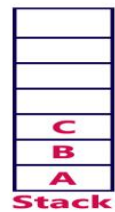
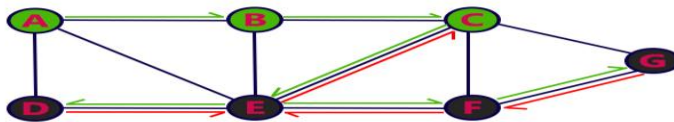
- There is no new vertex to be visited from **G**. So use back track.
- Pop **G** from the Stack.

**Step 10:**

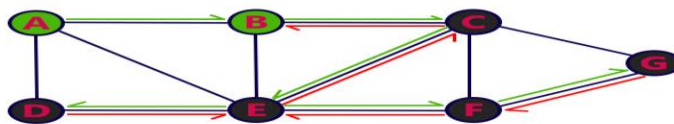
- There is no new vertex to be visited from **F**. So use back track.
- Pop **F** from the Stack.

**Step 11:**

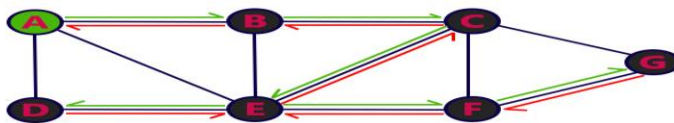
- There is no new vertex to be visited from **E**. So use back track.
- Pop **E** from the Stack.

**Step 12:**

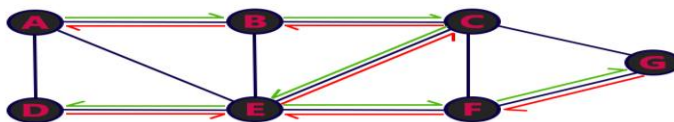
- There is no new vertex to be visited from **C**. So use back track.
- Pop **C** from the Stack.

**Step 13:**

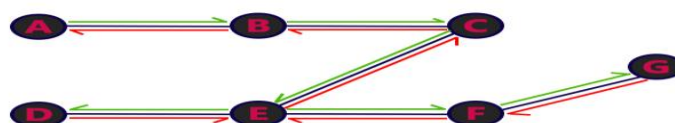
- There is no new vertex to be visited from **B**. So use back track.
- Pop **B** from the Stack.

**Step 14:**

- There is no new vertex to be visited from **A**. So use back track.
- Pop **A** from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



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];
            }
        }
    }
}
```

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
    int i;
    reach[v]=1;
    for(i=1;i<=n;i++)
    if(a[v][i] && !reach[i])
    {
        printf("\n %d->%d",v,i);
        dfs(i);
    }
}
void main()
{
    int i,j,count=0;
    clrscr();
    printf("Depth First Search");
    printf("\n Enter number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        reach[i]=0;
        for(j=1;j<=n;j++)
            a[i][j]=0;
    }
    printf("\n Enter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        scanf("%d",&a[i][j]);
```



```
        dfs(1);
        printf("\n");
    for(i=1;i<=n;i++)
    {
        if(reach[i])
            count++;
    }
    if(count==n)
        printf("\n Graph is connected");
    else
        printf("\n Graph is not connected");
    getch();
}
```

OUTPUT

```
Turbo C++ IDE
Depth First Search
Enter number of vertices:5

Enter the adjacency matrix:
0 1 1 1 0
1 0 0 1 0
1 0 0 0 1
1 1 0 0 1
0 0 1 1 0

1->2
2->4
4->5
5->3

Graph is connected
```

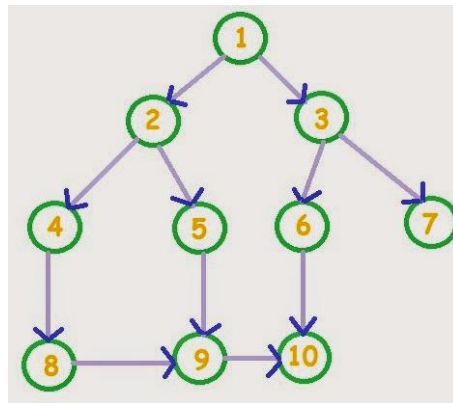
| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program to implement the Depth first search is executed and the output is verified.

EXERCISES:

Write a C program to find traverse the all the vertices of the given graph using Depth first search.



EX.NO:

DATE :

GRAPH TRAVERSAL – BREATH FIRST SEARCH

AIM

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

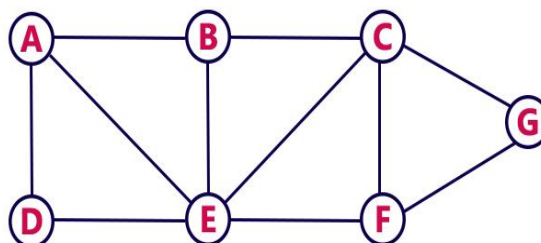
DESCRIPTION:

- BFS traversal of a graph produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without loops.
- We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

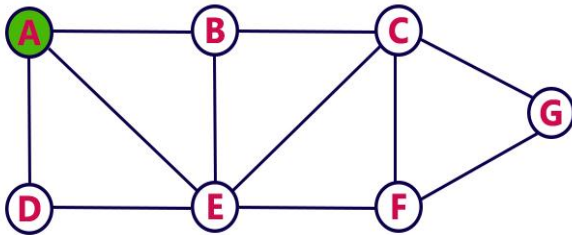
- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform DFS traversal

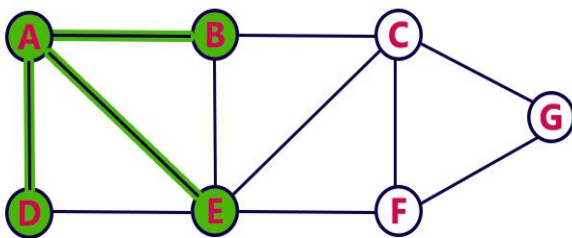


Step 1:

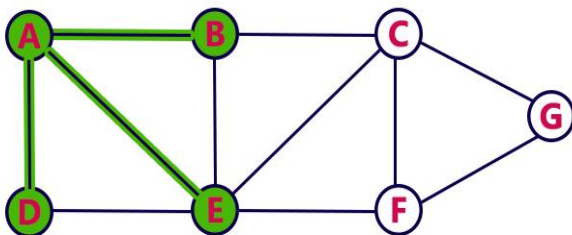
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

**Queue****Step 2:**

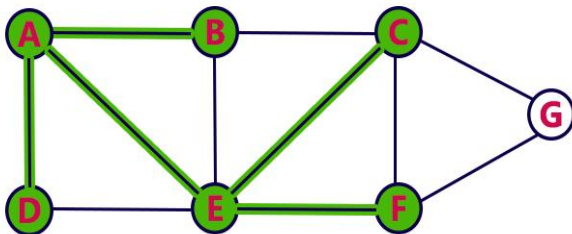
- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

**Queue****Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

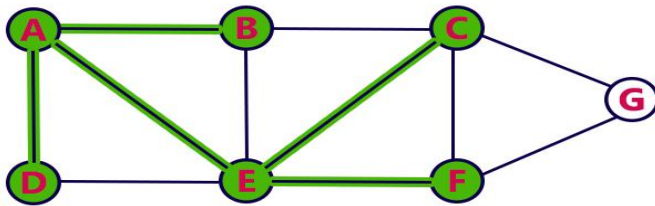
**Queue****Step 4:**

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

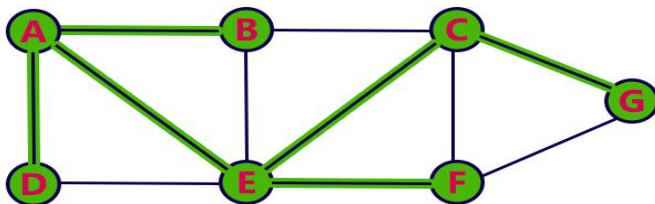
**Queue**

Step 5:

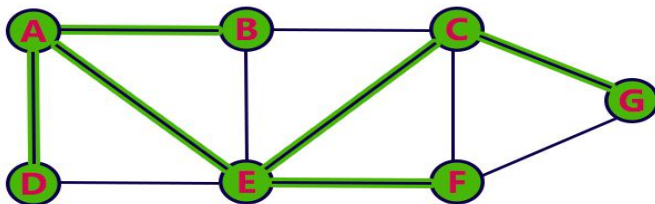
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

**Queue****Step 6:**

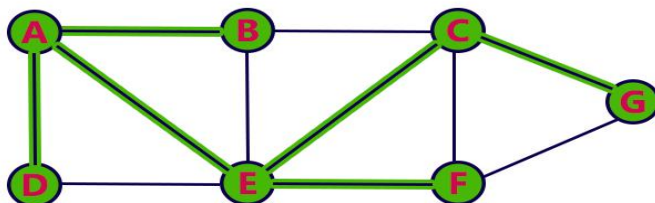
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

**Queue****Step 7:**

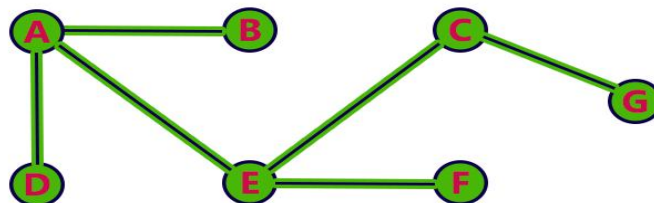
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

**Queue****Step 8:**

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

**Queue**

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



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

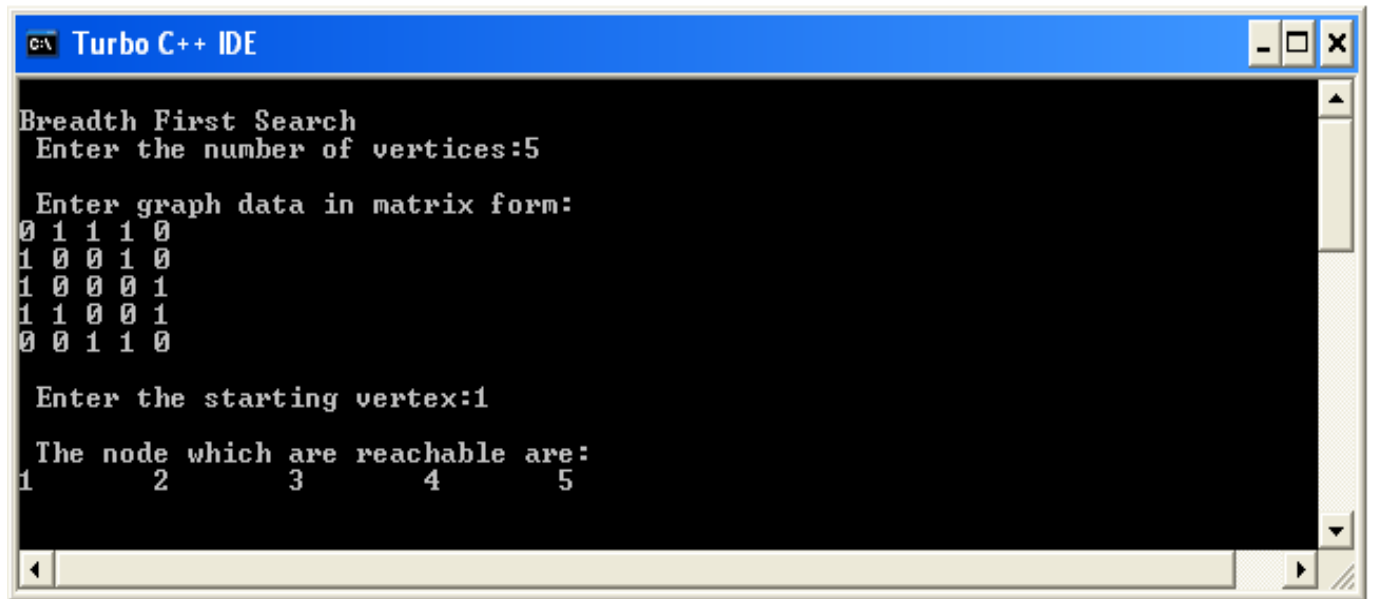
    write v;
    visited[v]=1;
    front=rear=0;
    while(front==rear)
        {
        for(i=1 to n) do
        {
        for(j=1 to n) do
        {
        if (edge[v,i]==1 && visited[i]==0) then
        {
        write i;
        visited[i]=1;
        queue[rear++]=i;
        }
        }
        }
        }
        v=queue[front];
    }
```


PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
for(i=1;i<=n;i++)
if(a[v][i] && !visited[i])
    q[++r]=i;
if(f<=r)
{
    visited[q[f]]=1;
    bfs(q[f++]);
}
}
void main()
{
int v;
clrscr();
printf("\nBreadth First Search");
printf("\n Enter the number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
    q[i]=0;
    visited[i]=0;
}
printf("\n Enter graph data in matrix form:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
```

```
printf("\n Enter the starting vertex:");  
scanf("%d",&v);  
bfs(v);  
printf("\n The node which are reachable are:\n");  
for(i=1;i<=n;i++)  
if(visited[i])  
    printf("%d\t",i);  
else  
    printf("\n Bfs is not possible");  
getch();  
}
```

OUTPUT



```
C:\ Turbo C++ IDE

Breadth First Search
Enter the number of vertices:5

Enter graph data in matrix form:
0 1 1 1 0
1 0 0 1 0
1 0 0 0 1
1 1 0 0 1
0 0 1 1 0

Enter the starting vertex:1

The node which are reachable are:
1      2      3      4      5
```

| CONTENTS | MARKS ALLOTED | MARKS OBTAINED |
|-----------------------|---------------|----------------|
| PROGRAM AND EXECUTION | 15 | |
| VIVA-VOCE | 10 | |
| TOTAL | 25 | |

RESULT:

Thus the C program to implement the Breadth first search is implemented and the output is verified.

EXERCISES:

Write a C program to find traverse the all the vertices of the given graph using Breadth first search.

