# F.Y. MCA

SEMESTER - I(CBCS)

# DATA STRUCTURES LAB USING C/C++

SUBJECT CODE : MCAL11

| | | |
|---|---|---|
| **Programme Co-ordinator** | : | **Shri Mandar Bhanushe**<br>Asst. Prof. cum Asst. Director in Mathematics,<br>IDOL, University of Mumbai, Mumbai |
| **Course Co-ordinator** | : | **Mrs. Reshama Kurkute**<br>Asst. Professor, Dept. of MCA<br>IDOL, University of Mumbai, Mumbai |
| **Course Writers** | : | **Prof. Aman Ansari**<br>Assistant Professor<br>B.N.N. College, Bhiwandi |
| | : | **Prof. Sujata Rizal**<br>Assistant Professor<br>S.M.Shetty College,<br>Mumbai |
| | : | **Prof. Saba Ansari**<br>Assistant Professor<br>J.K College of Science and Commerce<br>Navi Mumbai |

**DECEMBER 2021, Print - I**

| | | |
|---|---|---|
| Published by | : | Director<br>Institute of Distance and Open Learning ,<br>University of Mumbai,<br>Vidyanagari, Mumbai - 400 098. |

| | | |
|---|---|---|
| DTP Composed<br>Printed by | : | Mumbai University Press<br>Vidyanagari, Santacruz (E), Mumbai, |

# CONTENTS

❖❖❖❖

# F.Y. MCA
## SEMESTER - I(CBCS)
## DATA STRUCTURES LAB USING C/C++
## Syllabus

| Module No | Detailed Contents | Hrs |
|---|---|---|
| **01** | **Module: Sorting Techniques:**<br>Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Radix Sort<br>**Self Learning Topics:** Quick sort | **04** |
| **02** | **Module: Searching and Hashing Techniques:**<br>Linear search, Binary search, Methods for Hashing: Modulo Division, Digit Extraction, Fold shift, Fold Boundary, Linear Probe for Collision Resolution.<br>**Self Learning Topics :** Direct and Subtraction hashing | **08** |
| **03** | **Module: Stacks:**<br>Array implementation, Linked List implementation, Evaluation of postfix expression and balancing of parenthesis | **06** |
|  | **Self Learning Topics: Conversion of infix notation to postfix notation** |  |
| **04** | **Module: Queue:**<br>**Linked List implementation of ordinary queue, Array implementation of circular queue, Linked List implementation of priority queue, Double ended queue**<br>**Self Learning Topics : Other queue applications** | **08** |
| **05** | **Module: Linked List:**<br>**Singly Linked Lists, Circular Linked List, Doubly Linked Lists : Insert, Display, Delete, Search, Count, Reverse(SLL), Polynomial Addition**<br>**Self Learning Topics : Comparative study of arrays and linked list** | **10** |

|  |  |  |
|---|---|---|
| **06** | **Module: Trees:**<br><br>**Binary search tree : Create, Recursive traversal: preorder, post order, in order, Search Largest Node, Smallest Node, Count number of nodes, Heap: Min Heap, Max Heap: reheap Up, reheap Down, Delete**<br><br>**Self Learning Topics: Expression Tree, Heapsort** | **08** |
| **07** | **Module: Graphs:**<br><br>**Represent a graph using the Adjacency Matrix, BFS, Find the**<br><br>**minimum spanning tree (using any method Kruskal's Algorithm or Prim's Algorithm)**<br><br>**Self Learning Topics : Shortest Path Algorithm** | **08** |

❖❖❖❖

# Module I

**Practical No: 1**

**Aim: Implement program for Bubble sort.**

**Objective:** To understand working of bubble sort algorithm and sort array elements if they are not in the right order.

**Theory:**

1. Bubble sort is a sorting technique that compares two adjacent array elements and swaps them if they are not in the intended order.
2. It works on the principle of repeatedly swapping adjacent elements in case they are not in the right order. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.
3. In simpler terms, if the input is to be sorted in ascending order, the bubble sort will first compare the first two elements in the array. In case the second one is smaller than the first, it will swap the two, and move on to the next element, and so on.
4. Note that at the end of the first pass, the largest element in the list will be placed at its proper position.

**Example:**

Let us consider an array A[] that has the following elements:

**A[] = {30, 52, 29, 87, 63, 27, 19, 54}**

**Pass 1 :-**

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.

(b) Compare 52 and 29. Since 52 > 29, swapping is done.

**30, 29, 52, 87, 63, 27, 19, 54**

(c) Compare 52 and 87. Since 52 < 87, no swapping is done.

(d) Compare 87 and 63. Since 87 > 63, swapping is done.

**30, 29, 52, 63, 87, 27, 19, 54**

(e) Compare 87 and 27. Since 87 > 27, swapping is done.

**30, 29, 52, 63, 27, 87, 19, 54**

(f) Compare 87 and 19. Since 87 > 19, swapping is done.

**30, 29, 52, 63, 27, 19, 87, 54**

(g) Compare 87 and 54. Since 87 > 54, swapping is done.

**30, 29, 52, 63, 27, 19, 54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

**Pass 2 :-**

(a) Compare 30 and 29. Since 30 > 29, swapping is done.

**29, 30, 52, 63, 27, 19, 54, 87**

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.

**29, 30, 52, 27, 63, 19, 54, 87**

(e) Compare 63 and 19. Since 63 > 19, swapping is done.

**29, 30, 52, 27, 19, 63, 54, 87**

(f) Compare 63 and 54. Since 63 > 54, swapping is done.

**29, 30, 52, 27, 19, 54, 63, 87**

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

**Pass 3 :-**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.

**29, 30, 27, 52, 19, 54, 63, 87**

(d) Compare 52 and 19. Since 52 > 19, swapping is done.

**29, 30, 27, 19, 52, 54, 63, 87**

(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

**Pass 4 :-**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.

**29, 27, 30, 19, 52, 54, 63, 87**

(c) Compare 30 and 19. Since 30 > 19, swapping is done.

**29, 27, 19, 30, 52, 54, 63, 87**

(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

**Pass 5 :-**

(a) Compare 29 and 27. Since 29 > 27, swapping is done.

**27, 29, 19, 30, 52, 54, 63, 87**

(b) Compare 29 and 19. Since 29 > 19, swapping is done.

**27, 19, 29, 30, 52, 54, 63, 87**

(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

**Pass 6 :-**

(a) Compare 27 and 19. Since 27 > 19, swapping is done.

**19, 27, 29, 30, 52, 54, 63, 87**

(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array.
All the array elements are present in sorted order.

**Algorithm:**

**BUBBLE_SORT(A, N)**

```
  Step 1: Repeat Step 2 For I= 0 to N-1 // to keep track of the number of iterations
  Step 2: Repeat For J= 0 to N-I // to compare the elements within the particular iteration
  Step 3: IF A[J] > A[J+1]   // swap if any element is greater than its adjacent element
          SWAP A[J] and A[J+1]
          [END OF INNER LOOP]
          [END OF OUTER LOOP]
  Step 4: EXIT
```

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i, n, temp, j, arr[10];
  printf("Enter the maximum elements you want to store : ");
  scanf("%d", &n);
  printf("Enter the elements \n");
  for(i=0;i<n;i++)
  {
    scanf("%d", & arr[i]);
  }
```

```
    for(i=0;i<n;i++)
    {
      for(j=0;j<n-1;j++)
      {
        if(arr[j]>arr[j+1])
        {
          temp = arr[j];
          arr[j] = arr[j+1];
          arr[j+1] = temp;
        }
      }
    }
    printf("The array sorted in ascending order is :\n");
    for(i=0;i<n;i++)
      printf("%d\t", arr[i]);
    getch();
    return 0;
}
```

**Questions:**

1. Assume that we use Bubble Sort to sort n distinct elements in ascending order. When does the best case of Bubble Sort occur?

   A. When elements are sorted in descending order
   B. When elements are sorted in ascending order
   C. When elements are not sorted by any order
   D. There is no best case for Bubble Sort. It always takes O(n*n) time

2.The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order, using bubble sort is

   A. 11
   B. 12
   C. 13
   D. 10

3. When will bubble sort take worst-case time complexity?

   A. Only the first half of the array is sorted.
   B. Only the second half of the array is sorted.
   C. The array is sorted in descending order.
   D. The array is sorted in ascending order.

4. Sort given array elements using Bubble Sort.

   90 , 25 , 30 , 78 , 86 , 60 , 40 , 8 , 55

## Practical No: 2

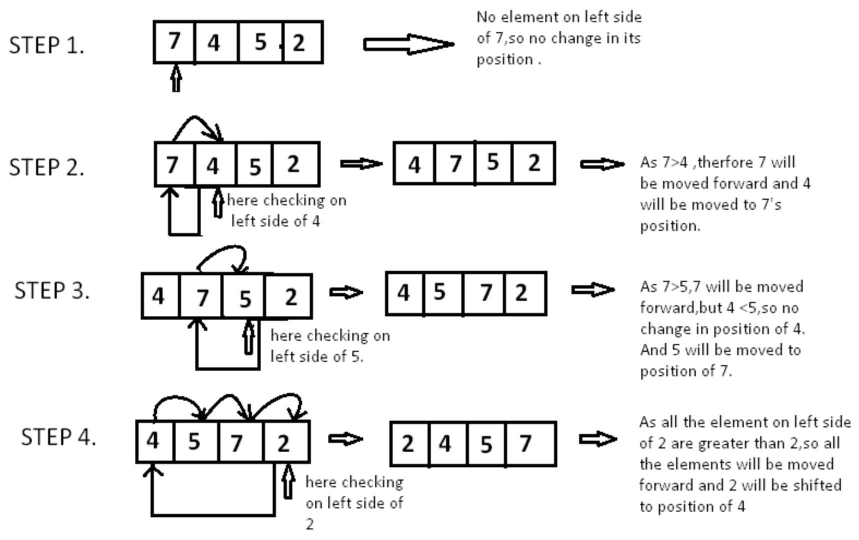
**Aim: Implement program for Insertion sort.**

**Objective:** To understand steps for sorting data using insertion sort algorithm. To implement program for sorting array elements using insertion sort.

**Theory:**

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. The array is virtually split into a sorted and an unsorted part. Elements from the unsorted part are picked and placed at the correct position in the sorted part. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

**Example:**


STEP 1.
7 4 5 2
No element on left side of 7,so no change in its position .
STEP 2.
7 4 5 2
here checking on left side of 4
4 7 5 2
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.
STEP 3.
4 7 5 2
here checking on left side of 5.
4 5 7 2
As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.
STEP 4.
4 5 7 2
here checking on left side of 2
2 4 5 7
As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

**Another Example:**

**12, 11, 13, 5, 6**

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

**Step 1:** Since 11 is smaller than 12, move 12 and insert 11 before 12

**11, 12, 13, 5, 6**

**Step 2:**13 will remain at its position as all elements in A[0..i-1] are smaller than 13

**11, 12, 13, 5, 6**

**Step 3:**5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

**5, 11, 12, 13, 6**

**Step 4:**6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

**5, 6, 11, 12, 13**

**Algorithm:**

**INSERTION-SORT (ARR, N)**

```
Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <=ARR[J]
        SET ARR[J + 1] = ARR[J]
        SET J = J - 1
              [END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main ()
   {
     int i, j, k,temp;
     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
     printf("\nprinting sorted elements...\n");
     for(k=1; k<10; k++)
     {
       temp = a[k];
       j= k-1;
       while(j>=0 && temp <= a[j])
       {
         a[j+1] = a[j];
         j = j-1;
       }
       a[j+1] = temp;
     }
     for(i=0;i<10;i++)
     {
       printf("\n%d\n",a[i]);
     }
     getch();
   }
```

**Questions:**

1. What will be the number of passes to sort the elements using insertion sort?

   14, 12,16, 6, 3, 10

   A. 6
   B. 5
   C. 7
   D. 1

2. For the following question, how will the array elements look like after second pass?

34, 8, 64, 51, 32, 21

   A. 8, 21, 32, 34, 51, 64
   B. 8, 32, 34, 51, 64, 21
   C. 8, 34, 51, 64, 32, 21
   D. 8, 34, 64, 51, 32, 21

3. In C, what are the basic loops required to perform an insertion sort?

   A. do- while
   B. if else
   C. for and while
   D. for and if

4.Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of insertions done while doing insertion sort in the array.

   A. 5 4 3 1 2, 5 4 1 2 3, 5 1 2 3 4, 1 2 3 4 5
   B. 4 5 3 2 1, 3 4 5 2 1, 2 3 4 5 1, 1 2 3 4 5
   C. 4 3 2 1 5, 3 2 1 5 4, 2 1 5 4 3, 1 5 4 3 2
   D. 4 5 3 2 1, 2 3 4 5 1, 3 4 5 2 1, 1 2 3 4 5

5. Which of the following real time examples is based on insertion sort?

   A. Arranging a pack of playing cards
   B. Database scenarios and distributes scenarios
   C. Arranging books on a library shelf
   D. Real-time systems

## Practical No: 3

**Aim: Implement program for Selection Sort.**

**Objective:** Develop a program for sorting array elements using selection sort.

**Theory:**

1. Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it

with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

2. The idea behind this algorithm is that first divide the array into two parts: sorted and unsorted. The left part is sorted subarray and the right part is unsorted subarray. Initially, sorted subarray is empty and unsorted array is the complete given array.Then perform the steps given below until the unsorted subarray becomes empty:

i. Pick the minimum element from the unsorted subarray.
ii. Swap it with the leftmost element and that element becomes a part of sorted subarray and will not be a part of unsorted subarray.
iii. This process continues moving unsorted array boundary by one element to the right.

**Example:**

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.
A = {10, 2, 3, 90, 43, 56}

| **Pass** | **A[0]** | **A[1]** | **A[2]** | **A[3]** | **A[4]** | **A[5]** |
|---|---|---|---|---|---|---|
| 1 | 2 | 10 | 3 | 90 | 43 | 56 |
| 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 3 | 2 | 3 | 10 | 90 | 43 | 56 |
| 4 | 2 | 3 | 10 | 43 | 90 | 56 |
| 5 | **2** | **3** | **10** | **43** | **56** | **90** |

**Algorithm:**

```
SELECTION SORT(ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
            [END OF LOOP]
Step 4: EXIT
```

```
SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N -1
            IF SMALL > ARR[J]
                SET SMALL = ARR[J]
                SET POS = J
            [END OF IF]
            [END OF LOOP]
Step 4: RETURN POS
```

**Program:**

```
#include<stdio.h>
#include<conio.h>
int smallest(int[],int,int);
void main ()
    {
      int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
      int i,j,k,pos,temp;
      for(i=0;i<10;i++)
      {
         pos = smallest(a,10,i);
         temp = a[i];
         a[i]=a[pos];
         a[pos] = temp;
      }
      printf("\nprinting sorted elements...\n");
      for(i=0;i<10;i++)
      {
         printf("%d\n",a[i]);
      }
    }
    int smallest(int a[], int n, int i)
    {
      int small,pos,j;
      small = a[i];
      pos = i;
      for(j=i+1;j<10;j++)
      {
         if(a[j]<small)
         {
            small = a[j];
            pos=j;
        }
    }
    getch();
    return pos;
}
```

**Questions:**

1. The given array is arr = {3,4,5,2,1}. The number of iterations in bubble sort and selection sort respectively are __________.

   A. 5 and 4
   B. 4 and 5
   C. 2 and 4
   D. 2 and 5

2.How many comparisons are needed to sort an array of length 5 if a straight selection sort is used and array is already in the opposite order?

   A. 1
   B. 10
   C. 5
   D. 20

3. Which operation does the Selection sort use to move numbers from the unsorted section to the sorted section of the list?

   A. Swap
   B. Sort
   C. Insert
   D. Merge

4. For each i from 1 to n-1, there are ____________ exchanges for selection sort.

   A. 1
   B. n
   C. n-1
   D. n-3

5. Selection sort the following array. Show the array after each swap that takes place.

   80, 65, 46, 32, 95, 30

## Practical No: 4

**Aim: Implement program for Shell sort.**

**Objective:** To understand working of Shell Sort algorithm for sorting array elements and implement program for the same.

**Theory:**

Shell sort algorithm is invented by Donald shell. Shell sort is a highly efficient sorting algorithm. It is a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. This algorithm avoids large shifts. The idea of ShellSort is to allow exchange of far items. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. We keep reducing the value of interval until it becomes 1.

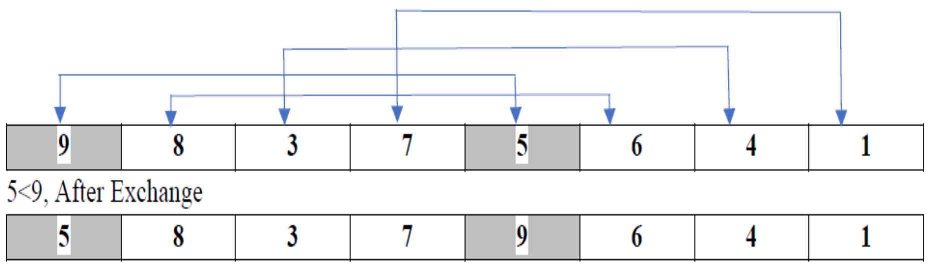
**Example:**

Suppose, we need to sort the following array.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **9** | **8** | **3** | **7** | **5** | **6** | **4** | **1** |

**Step 1:** In the first step, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.

a. The 0th element is compared with the 4th element.
b. If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.



This process goes on for all the remaining elements at N/2 interval i.e., 4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **5** | **8** | **3** | **7** | **9** | **6** | **4** | **1** |
| **5** | **6** | **3** | **7** | **9** | **8** | **4** | **1** |
| **5** | **6** | **3** | **7** | **9** | **8** | **4** | **1** |
| **5** | **6** | **3** | **1** | **9** | **8** | **4** | **7** |

In the second step, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.
Rearrange the elements at n/4 interval

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 6 | 3 | 1 | 9 | 8 | 4 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 9 | 8 | 4 | 7 |

All the elements in the array lying at the current interval are compared.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 5 | 6 | 9 | 8 | 4 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 5 | 6 | 9 | 8 | 4 | 7 |

The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position is also compared. All the elements in the array lying at the current interval are compared. The same process goes on for remaining elements.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 1 | 9 | 8 | 4 | 7 |
| 3 | 1 | 5 | 6 | 9 | 8 | 4 | 7 |
| 3 | 1 | 4 | 6 | 5 | 8 | 9 | 7 |
| 3 | 1 | 4 | 6 | 5 | 8 | 9 | 7 |
| 3 | 1 | 4 | 6 | 5 | 7 | 9 | 8 |

Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 6 | 5 | 7 | 9 | 8 |
| 1 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |
| 1 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |
| 1 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |
| 1 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| 1 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| 1 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| 1 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**Algorithm:**

```
Shell_Sort(Arr, n)
Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3: SET FLAG = 0
Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5: Repeat Step 6 for I = 0 to I < (N -GAP_SIZE)
Step 6: IF Arr[I + GAP_SIZE] > Arr[I]
        SWAP Arr[I + GAP_SIZE], Arr[I]
        SET FLAG = 0
Step 7: END
```

**Program:**

```
#include <stdio.h>
#include <conio.h>
void shellsort(int arr[], int num)
{
	int i, j, k, tmp;
	for (i = num / 2; i > 0; i = i / 2)
	{
	for (j = i; j < num; j++)
	{
		for(k = j - i; k >= 0; k = k - i)
		{
			if (arr[k+i] >= arr[k])
			break;
			else
			{
				tmp = arr[k];
				arr[k] = arr[k+i];
				arr[k+i] = tmp;
			}
		}
	}
	}
}
int main()
{
	int arr[30];
	int k,  num;
	printf("Enter total no. of elements : ");
	scanf("%d", &num);
	printf("\nEnter %d numbers: ", num);
	for (k =  0 ; k < num; k++)
	{
		scanf("%d", &arr[k]);
	}
	shellsort(arr, num);
	printf("\n Sorted array is: ");
	for (k = 0; k < num; k++)
```

```
        printf("%d ", arr[k]);
        return 0;
}
```

**Questions:**

1.Which of the following sorting algorithms is closely related to shell sort?

A. Selection sort
B. Merge sort
C. Insertion sort
D. Bucket sort

2. Why is Shell sort called as a generalization of Insertion sort?

A. Shell sort allows an exchange of far items whereas insertion sort moves elements by one position
B. Improved lower bound analysis
C. Insertion is more efficient than any other algorithms
D. Shell sort performs internal sorting

3. Who invented the shell sort algorithm?

A. John Von Neumann
B. Donald Shell
C. Tony Hoare
D. Alan Shell

4. Shell sort is applied on the elements 27 59 49 37 15 90 81 39 and the chosen decreasing sequence of increments is (5, 3, 1). The result after the first iteration will be ___________.

A. 27 59 49 37 15 90 81 39
B. 27 59 37 49 15 90 81 39
C. 27 59 39 37 15 90 81 49
D. 15 59 49 37 27 90 81 39

5. What is the best-case complexity for shell sort?

A. O(1)
B. O(n)
C. O(logn)
D. O(nlogn)

6. Given an array of the following elements

81,94,11,96,12,35,17,95,28,58,41,75,15.

What will be the sorted order after shell sort?

## Practical No: 5

**Aim: Implement program for Radix sort.**

**Objective:** To understand steps for sorting elements using Radix sort.Develop a program for sorting array elements using Radix sort.

**Theory:**

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit. Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

**Step by Step Process:**

The Radix sort algorithm is performed using the following steps.

**Step 1** - Define 10 queues each representing a bucket for each digit from 0 to 9.

**Step 2** - Consider the least significant digit of each number in the list which is to be sorted.

**Step 3** - Insert each number into their respective queue based on the least significant digit.

**Step 4** - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

**Step 5** - Repeat from step 3 based on the next least significant digit.

**Step 6** - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

**Example:**

Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 345 | | | | | | 345 | | | | |
| 654 | | | | | 654 | | | | | |
| 123 | | | | 123 | | | | | | |
| 567 | | | | | | | | 567 | | |
| 472 | | | 472 | | | | | | | |
| 555 | | | | | | 555 | | | | |
| 808 | | | | | | | | | 808 | |
| 911 | | 911 | | | | | | | | |

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 911 | | 911 | | | | | | | | |
| 472 | | | | | | | | 472 | | |
| 123 | | | 123 | | | | | | | |
| 654 | | | | | | 654 | | | | |
| 345 | | | | | 345 | | | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | | 567 | | | |
| 808 | 808 | | | | | | | | | |

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

| **Number** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **808** | | | | | | | | | **808** | |
| **911** | | | | | | | | | | **911** |
| **123** | | **123** | | | | | | | | |
| **345** | | | | **345** | | | | | | |
| **654** | | | | | | | **654** | | | |
| **555** | | | | | | **555** | | | | |
| **567** | | | | | | **567** | | | | |
| **472** | | | | | **472** | | | | | |

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as
123, 345, 472, 555, 567, 654, 808, 911

**Algorithm:**

```
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS =0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5: SET I = 0 and INITIALIZE buckets
Step 6: Repeat Steps 7 to 9 while I
Step 7: SET DIGIT = digit at Pass^th place in A[I]
Step 8: Add A[I] to the bucket numbered DIGIT
Step 9: INCREMENT bucket count for bucket numbered DIGIT
            [END OF LOOP]
Step 10: Collect the numbers in the bucket
            [END OF LOOP]
Step 11: END
```

**Program:**

```
#include <stdio.h>
#include <conio.h>
int largest(int a[]);
void radix_sort(int a[]);
void main()
{
    int i;
    int a[10]={90,23,101,45,65,23,67,89,34,23};
    radix_sort(a);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
        printf(" %d\t", a[i]);
}

int largest(int a[])
{
    int larger=a[0], i;
    for(i=1;i<10;i++)
    {
        if(a[i]>larger)
        larger = a[i];
    }
    return larger;
}
void radix_sort(int a[])
{
    int bucket[10][10], bucket_count[10];
    int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
    larger = largest(a);
    while(larger>0)
    {
        NOP++;
        larger/=10;
    }
    for(pass=0;pass<NOP;pass++) // Initialize the buckets
    {
        for(i=0;i<10;i++)
        bucket_count[i]=0;
        for(i=0;i<10;i++)
        {
            // sort the numbers according to the digit at passth place
            remainder = (a[i]/divisor)%10;
            bucket[remainder][bucket_count[remainder]] = a[i];
            bucket_count[remainder] += 1;
        }
        // collect the numbers after PASS pass
        i=0;
        for(k=0;k<10;k++)
        {
            for(j=0;j<bucket_count[k];j++)
            {
               a[i] = bucket[k][j];
               i++;
            }
        }
        divisor *= 10;
}
getch();
}
```

**Questions:**

1. Given a number of elements in the range [0….n^3]. which of the following sorting algorithms can sort them in O(n) time?
   A. Counting sort
   B. Bucket sort
   C. Radix sort
   D. Quick sort

2. Suppose we need to sort 10 million 80-character strings representing DNA information from a biological study. Which sorting algorithm should we use?
   A. Bucket Sort
   B. Radix Sort
   C. Quick Sort
   D. Selection Sort

3. Given an array where numbers are in range from 1 to n6, which sorting algorithm can be used to sort these number in linear time?
   A. Not possible to sort in linear time
   B. Radix Sort
   C. Counting Sort
   D. Quick Sort

4. Which of the following is the most suitable definition of radix sort?
   A. It is a non-comparison based integer sort
   B. It is a comparison based integer sort
   C. It is a non-comparison based non integer sort
   D. It is a comparison based non integer sort

5. Which of the following is the distribution sort?
   A. Heap sort
   B. Smooth sort
   C. Quick sort
   D. Radix sort

6. Sort following elements using Radix Sort.
   127, 324, 173, 4, 38, 217, 135

7. Perform Radix sort on following array elements
   10, 21, 17, 34, 44, 11, 654, 123

**Self-Learning Topic:**

**Quick sort:**
Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element.The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.Finally, elements are combined to form a sorted array.

❖❖❖❖

# Module II

**Practical No: 1**

**Aim: Implement program for Linear Search.**

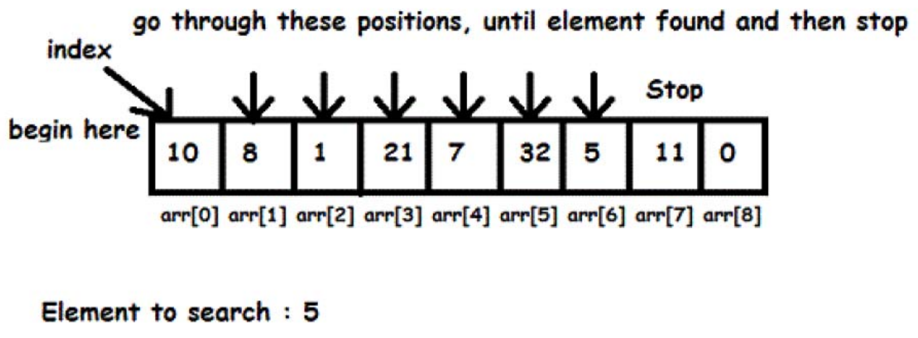
**Objective:** Develop a program for searching an element from array using Linear search.

**Theory:**

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found then location of the item is returned otherwise the algorithm returns no element found. Linear search is mostly used to search an unordered list in which the items are not sorted. As Linear search compares each and every element one by one i.e., it requires more time as compared to other search algorithms.

**Example:**

If an array A[] is declared and initialized as,
int A[] = {10, 8, 1, 21, 7, 32, 5, 11, 0}


go through these positions, until element found and then stop
index
begin here
Stop
10
8
1
21
7
32
5
11
0
arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6] arr[7] arr[8]
Element to search : 5

The value to be searched is VAL = 5, then searching means to find whether the value '5' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 6 (index starting from 0).

**Algorithm:**

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [Initialize] set pos = -1
Step 2: [Initialize] set i = 1
Step 3: Repeat Step 4 while I<=N
Step 4:If a[i] = val
        Set pos = i
```

```
          Print pos
          Go to step 6
          [End of if]
          Set i = i + 1
          [End of loop]

Step 5:If pos = -1
          Print " value is not present  in the array "
          [End of if]
Step 6:Exit
```

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main ()
    {
    int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
    int item, i, flag;
    printf("\nEnter Item which is to be searched\n");
    scanf("%d",&item);
    for (i = 0; i< 10; i++)
    {
        if(a[i] == item)
        {
                flag = i+1;
                break;
        }
        else
        {
                flag = 0;
        }
    }
    if(flag != 0)
    {
        printf("\nItem found at location %d\n",flag);
    }
    else
    {
        printf("\nItem not found\n");
    }
    getch();
    }
```

**Questions:**

1. Which of the following is a disadvantage of linear search?
    - A. Requires more space
    - B. Requires more time for searching
    - C. Not easy to understand
    - D. Not easy to implement

2. The array is as follows: 11,2,30,65,80,100. Given that the number 23 is to be searched. After how many iterations it tells that there is no such element?
    - A. 7
    - B. 9
    - C. 20
    - D. 5

3. A linear search algorithm is also known as a __________.
    - A. Binary search algorithm
    - B. Bubble sort algorithm
    - C. Sequential search algorithm
    - D. Radix search algorithm

4. What will happen in a Linear search algorithm if no match is found?
    - A. It continues to search in a never-ending loop.
    - B. "Item not found" is returned
    - C. Compile-time error
    - D. Run-time error

5. What is an advantage of the Linear search algorithm?
    - A. Is complicated to code
    - B. Can be used on data sets with more than a million elements
    - C. Performs well with small sized data sets
    - D. Difficult to understand

## Practical No: 2

**Aim: Implement program for Binary Search.**

**Objective:** To understand working of Binary search algorithm and to implement program for searching an element using binary search.

**Theory:**

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted. But it cannot be applied to linked list. Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched in the sub-array to the right of the middle item.

**Example:**

Let us consider an array a = {11, 15, 17, 18, 23, 29, 30, 33, 39}. Find the location of the item 33 in the array.

| **Elements** | **11** | **15** | **17** | **18** | **23** | **29** | **30** | **33** | **39** |
|---|---|---|---|---|---|---|---|---|---|
| **Indexes** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**Step 1:**

BEG = 0
END = 8
MID = (BEG+END)/2= (0+8)/2= 4
a[MID] = a[4] = 23 < 33

**Step 2:**

BEG = MID +1 = 4+1= 5
END = 8
MID = (BEG+END)/2= (5+8)/2= 13/2 = 6
a[MID] = a[6] = 30 < 33

**Step 3:**

BEG = MID + 1 = 6+1= 7
END = 8
MID = (BEG+END)/2= (7+8)/2= 15/2 = 7
a[MID] = a[7]
a[7] = 33 = item;
Therefore, the location of the item will be 7.

**Algorithm:**

**Step 1**: Find the middle element in the sorted list.

**Step 2**: Compare the search element with the middle element in the sorted list.

**Step 3**: If both are matched, then display "Given element is found!" and terminate the function.

**Step 4**: If both are not matched, then check whether the search element is smaller or larger than the middle element.

**Step 5**: If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

**Step 6**: If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

**Step 7**: Repeat the same process until we find the search element in the list or until sublist contains only one element.

**Step 8**: If that element also doesn't match with the search element, then display "Element is not found in the list" and terminate the function.

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int first, last, middle, size, i, key, list[100];
        clrscr();
        printf("Enter the size of the list: ");
        scanf("%d",&  size);
        printf("Enter %d integer values in Ascending order\n", size);
        for (i = 0; i < size; i++)
        {
                scanf("%d",&list[i]);
        }
        printf("Enter value to be search: ");
        scanf("%d", &key);
        first = 0;
        last = size - 1;
        middle = (first+last)/2;
        while (first <= last)
        {
                if (list[middle] <key)
                {
                        first = middle + 1;
                }
                else if (list[middle] == key)
                {
                        printf("Element found at index %d.\n",middle);
                        break;
                }
                else
                {
                        last = middle - 1;}
                middle = (first + last)/2;
        }
        if (first > last)
        {
                printf("Element Not found in the list.");
        }
        getch();
}
```

**Questions:**

1. Binary Search can be categorized into which of the following?
   A. Brute Force technique
   B. Divide and conquer
   C. Greedy algorithm
   D. Dynamic programming

2. Given an array S = {50,60,77,88,99} and key = 88; How many iterations are done until the element is found?
   A. 1
   B. 3
   C. 4
   D. 2

3. Binary search algorithm cannot be applied to
   A. Sorted linked list
   B. Sorted binary trees
   C. Sorted linear array
   D. Pointer array

4. Search element 500 from the given array using binary search algorithm.
   50, 60, 150, 280, 320, 400, 500, 600

## Practical No: 3

**Aim: Implement program for Modulo Division.**

**Objective:**To understand modulo division method in hashing with the help of example. To implement program for finding key location of elements using Modulo division.

**Theory:**

Hashing is a technique or process of mapping keys, values into the hash table by using a hash function. It is done for faster access to elements. Modulo Division is the easiest method to create a hash function.

Also known as division remainder, the modulo-division method divides the key by table size and uses the remainder for the address. The hash function can be described as

$$h(k) = k \bmod m$$

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table m using the remainder.

A disadvantage of the division method is that consecutive keys map to consecutive hash values in the hash table. This leads to a poor performance. This algorithm works with any table size, but a table size that is a prime number produces fewer collisions than other table sizes. We should therefore try to make the array size a prime number.

**Example:**

Elements to be placed in a hash table are 42,78,89,64 and let's take table size as 10.

Hash (key) = Elements % table size;

h(k)= k mod m

h(42) = 42 % 10 = 2

h(78) = 78 % 10 = 8

h(89) = 89 % 10 = 9

h(64) = 64 % 10 = 4

The table representation can be seen as below:

| Key | Value |
|---|---|
| 0 | |
| 1 | |
| 2 | 42 |
| 3 | |
| 4 | 64 |
| 5 | |
| 6 | |
| 7 | |
| 8 | 78 |
| 9 | 89 |

**Algorithm:**

Suppose array name is A and n is the size of array.

```
Step 1: Initialize all array values with -1.
Step 2: Specify the values which needs to be inserted.
Step 3: Calculate key address using modulo division method.
            Set key= value % size
Step 4: If A[key] = = -1
        Set A[key] = value   // Insert the value at calculated key or address
```

```
            Else
                        Print: Unable to insert
          [End If]
Step 5: If A[key] = = value
                        Print: Search found
            Else
                        Print: Search not found
          [End If]
Step 6: Repeat while key< n
                        Print: A[key]
          [End while]
Step 7: End
```

**Program:**

```
#include<stdio.h>
#include<conio.h>
#define size 7
int arr[size];
void init()
{
   int i;
   for(i = 0; i < size; i++)
   {
      arr[i] = -1;
   }
}

void insert(int value)
{
   int key = value % size;              //use of modulo division
   if(arr[key] == -1)
   {
      arr[key] = value;
      printf("%d inserted at arr[%d]\n", value,key);
   }
   else
   {
      printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);
      printf("Unable to insert %d\n",value);
   }
}

void search(int value)
{
   int key = value % size;
   if(arr[key] == value)
   {
      printf("Search Found\n");
   }
   else
   {
      printf("Search Not Found\n");
    }
}

void display()
{
   int i;
   for(i = 0; i < size; i++)
   {
      printf("arr[%d] = %d\n",i,arr[i]);
    }
}

int main()
{
   init();
   insert(10); //key = 10 % 7 ==> 3
   insert(4);  //key = 4 % 7  ==> 4
   insert(2);  //key = 2 % 7  ==> 2
   insert(3);  //key = 3 % 7  ==> 3 (collision)

   printf("Hash table\n");
display();
   printf("\n");
   printf("Searching value 4..\n");
   search(4);
   getch();
   return 0;
}
```

**Questions:**

1. What is the hash function used in the division method?
   - A. h(k) = k/m
   - B. h(k) = k mod m
   - C. h(k) = m/k
   - D. h(k) = m mod k

2. What can be the value of m in the division method?

A. Any prime number
B. Any even number
C. 2p – 1
D. 2p

3. Using division method, in a given hash table of size 157, the key of value 172 be placed at position _________.

A. 19
B. 72
C. 15
D. 17

4. In which of the following hash functions, do consecutive keys map to consecutive hash values?

A. Division Method
B. Multiplication Method
C. Folding Method
D. Mid-Square Method

## Practical No: 4

**Aim: Implement program for Digit Extraction.**

**Objective:** To develop program for hashing using digit extraction method.

**Theory:**

Using digit extraction method, selected digits are extracted from the key and used as the address. It is also called a Truncation method. Steps for truncation are as follows.

1. Choose the hash table size.

2. Then the respective right most or left most digits are truncated and used as hash value.

If address is represented by n-bits, the use n digits from the key.

Address = selected digits from key

Using employee number to hash to a 3 digit address we could select first, third & fourth element [from left].

| Key | Address |
|---|---|
| 3 9 7 4 2 5 | 3 7 4 |
| 2 3 5 6 7 8 | 2 5 6 |

**Example:**

Ex: 123,42,56 and Table size = 9

H(123) =1 //First digit i.e. 1 is selected

H(42) = 4 //First digit i.e. 4 is selected

H(56) = 5 //First digit i.e. 5 is selected

| **Address** | **Key** |
|---|---|
| 0 | |
| 1 | 123 |
| 2 | |
| 3 | |
| 4 | 42 |
| 5 | 56 |
| 6 | |
| 7 | |
| 8 | |

**Algorithm:**

```
Step 1: Begin
Step 2: Pass key value ‘Key’ as an argument to digit_extraction().
Step 3: Initialize values
        Set: first_digit=0 and fouth_digit=0 //for extracting digit at first & fourth position
Step 4: For extracting first digit from given no
        Calculate: first_digit= key%10000000;
                   first_digit=first_digit/1000000;
Step 5: For extracting fourth digit from given no
        Calculate: fourth_digit= key%1000;
                   fourth_digit=fourth_digit/100;
Step 6: Display the hashed location where given number will be stored.
        Print: (first_digit, fourth_digit);
Step 7:End
```

**Program:**

```
#include<stdio.h>
int digit_extraction(int key)
{
        int key_length=0;
        int first_digit=0;
        int fourth_digit=0;
        first_digit= key%10000000;
        first_digit=first_digit/1000000;
        fourth_digit= key%1000;
        fourth_digit=fourth_digit/100;
        printf("%d key would be hashed at location %d%d \n",key,first_digit, fourth_digit);
}

int main()
{
        digit_extraction(1347878);   //18
        digit_extraction(1234678);   //16
        return 0;
}
```

**Questions:**

1. In ________ method of hashing, selected digits are extracted from the key and used as the address.

   A. Subtraction
   B. Digit extraction
   C. Rotation
   D. Folding

2. __________ is also known as Digit Extraction.

   A. Folding
   B. Subtraction
   C. Truncation
   D. Collision Resolution

3. If key is 987654 then using the odd-place digits, the index(hash value) would be ________.

   A. 486
   B. 864
   C. 975
   D. 987

4. If key is 356487 then using the even-place digits, the index(hash value) would be ________.

   A. 368
   B. 547
   C. 648
   D. 354

## Practical No: 5

**Aim: Implement program for Fold Shift.**

**Objective:**To understand fold shift method of hash function and to implement program for hashing values using fold shift.

**Theory:**

In fold shift the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part.The folding method works in the following two steps:

**Step 1:** Divide the key value into a number of parts. That is, divide k into parts k1, k2, …, kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

**Step 2:** Add the individual parts. That is, obtain the sum of k1 + k2 + … + kn. The hash value is produced by ignoring the last carry, if any.

**Example:**

Suppose to calculate hash value for X = 5678 and hash table size 100, we need to follow below steps:

Step 1: The X will be divided into two parts each having two digits i.e. k1=56 and k2 = 78

Step 2: Adding all key parts

k1 + k2 i.e.

Key= 56 + 78 = 134

After ignoring the carry 1 (because here only two digits are required as hash value) the resulting hash value for 5678 is 34.

**Algorithm:**

**Step 1:**The folding method is used for creating hash functions starts with the item being divided into equal-sized pieces i.e., the last piece may not be of equal size.

**Step 2:**The outcome of adding these bits together is the hash value, H(x) = (a + b + c) mod M, where a, b, and c represent the preconditioned key broken down into three parts and M is the table size, and mod stands for modulo.

**Step 3:**In other words, the sum of three parts of the preconditioned key is divided by the table size. The remainder is the hash key.

**Program:**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include <math.h>
int count_digits(int key)
{
        int count=0;
        while(key != 0)
        {
                key /= 10;
                ++count;
        }
        return count;
}

int fold_shift(int key, int size)
{
        int key_roll=key;
        int key_sum=0;
        int key_frac=0;
        int key_length=0;
        int fraction = size;
        key_length = count_digits(key_roll);
        while (key_length > 0)
        {
         if (key_length >fraction)
        {
         key_frac = key_roll / (int)pow(10, (key_length - fraction));
        key_sum += key_frac;
         key_roll = key_roll % (int)pow(10, (key_length - fraction));
        key_length = key_length - fraction;
        }
        else
        {
                key_sum += key_roll;
                break;
        }
}
return key_sum % (int)pow(10, (fraction));
}

int main()
{
        clrscr();
        printf("\n\n%d",fold_shift(12789, 3));   //216
        printf("\n\n%d",fold_shift(12345678, 1));    //6
        printf("\n\n%d",fold_shift(5678, 2));   //34
        getch();
        return 0;
}
```

**Questions:**

1. For key 345678123 what will be index in fold shift?
   - A. 146
   - B. 641
   - C. 542
   - D. 678

2. Folding is a method of generating _________.
   - A. A hash function
   - B. Index function for a triangular matrix
   - C. Linear probing
   - D. Chaining

3. Hashing method in which the given key is partitioned into subparts k1,k2,k3....kn is known as
   - A. Mid square method
   - B. Division method
   - C. Partition method
   - D. Folding method

4. If the number is 164257408 and table size is 100 then the location where number will get stored by fold shift method is _________.
   - A. 6
   - B. 3
   - C. 56
   - D. 63

5. If the number is 123456789 and table size is 1000 then address where number will get stored by foldshift method is __________.
   - A. 8
   - B. 138
   - C. 368
   - D. 20

## Practical No: 6

**Aim: Implement program for Fold Boundary.**

**Objective:**To understand fold boundary method of hash function and to develop program for hashing values using fold boundary.

**Theory:**
In fold boundary the left and right numbers are folded on a fixed boundary between them and the center number. The two outside values are thus reversed.

The fold boundary method works in the following two steps:
**Step 1:** Divide the key value into a number of parts i.e., left, right and middle parts.

**Step 2:**Reverse left and right individual parts. Then, obtain the sum of reversed parts and middle part if it exists. The hash value is produced by ignoring the last carry, if any.

**Example:**

Suppose to calculate hash value for Key = 123456789 and size of required address is 3 digits, we need to follow below steps:

**Step 1:** The Key will be divided into two parts

Key = 123 | 456 | 789

**Step 2:**Reverse left and right parts and add it with middle part.

321 (folding applied)+456+987 (folding applied) = 1764(discard 1 or 4)

After we ignore the carry 1 (because here only three digits are required as hash value) the resulting hash value for 123456789 is 764.

**Algorithm:**

**Step 1:** Specify the number which is to be folded and boundary between them.

fold_boundary(int key, int size)

**Step 2:** Initialize all the integer values.

Set: key_sum=0, key_frac=0, middle=0, left=0, right=0, digits=0, key_length=0

Set: key_roll=key & fraction = size

```
Step 3: Calculate key_length
        count_digits(key)
        a. Initialize count with value zero
        b. Repeat while (key!=0)
               Key/=10;
               ++count
               [End while]
        c. Return count
        d. Terminate function
```

**Step 4:** Divide the number in three parts around a fixed boundary on left and right side.

**Step 5:** Compute first three digits of given number. Reverse it and store the reversed value in left variable.

**Step 6:** Compute last three digits of the given number. Reverse it and store the reversed value in right variable.

**Step 7:** Find middle value of the given number and store it in variable middle.

```
Step 8: Calculate key_sum
         key_sum = left + middle + right
Step 9: If carry is generated then ignore carry
         Set: key_sum= key_sum % (int)pow(10, (fraction))
Step 10: Print: key_sum
Step 11: End
```

**Program:**

```
#include<stdio.h>
#include<string.h>
#include <math.h>
int count_digits(int key)
{
        int count=0;
        while(key != 0)
        {
                key /= 10;
                ++count;
        }
return count;
}

int fold_boundary(int key, int size)
```

```
{
    int key_roll=key;
    int key_sum=0;
    int key_frac=0;
    int middle=0;
    int left=0;
    int right=0;
    int digits=0;
    int key_length=0;
    int fraction = size;
    key_length = count_digits(key_roll);
    key_frac = key_roll / (int)pow(10, (key_length - fraction));// start digit
    left=reversDigits(key_frac);
    key_roll = key_roll % (int)pow(10,3);
    right=reversDigits(key_roll);
    digits = (int)log10(key) + 1;
    middle= (int)(key / pow(10, digits/ 2)) % 10;
    key_sum = left +middle+ right;
        return key_sum % (int)pow(10, (fraction)); //ignore carry
}

int reversDigits(int num)
{
        int rev_num = 0;
        while (num > 0)
        {
                rev_num = rev_num * 10 + num % 10;
                num = num / 10;
        }
        return rev_num;
}

int main()
{
        printf("\n\n%d",fold_boundary(3347878, 3));   //318
        printf("\n\n%d",fold_boundary(1234678, 3));    //201
        return 0;
}
```

**Questions:**

1. In which of the following, the left and right numbers are reversed on except the center number?

   A. Division method
   B. fold boundary
   C. fold shift
   D. folding method

2. For key 345678123 what will be index in fold boundary?

   A. 542
   B. 245
   C. 146
   D. 876

3. If number is 15547012 and table size is 100 then address of number by fold boundary method is _________.

   A. 1
   B. 51
   C. 96
   D. 6

4. The types of folding method are:

   A. fold shift
   B. fold boundary
   C. both
   D. none of these

## Practical No: 7

**Aim: Implement program for Linear probe for Collision Resolution.**

**Objective:** To understand linear probing with its example. To develop program for collision resolution using linear probe.

**Theory:**

Linear probing is one of the collision resolution techniques classified under open addressing technique/Closed Hashing which is a method for handling collisions. While hashing if two or more key points to the same hash index under some modulo M it is called as collision. When collision occurs, we linearly probe for the next slot. We keep probing until an empty slot is found. The main problem with linear probing is clustering. Many consecutive elements form groups.

Steps for collision resolution using linear probing are as follows.

**Step 1:** Calculate the hash key.

address = key % size;

**Step 2:** If hashTable[key] is empty, store the value directly. hashTable[key] = data.

If the hash index already has some value, check for next index.

Next address = (key+1) % size;

If the next index is available hashTable[key], store the value. Otherwise try for next index.

h(k) = (key+i) % size; where i= 0,1,2,3,…

**Step 3:**Do the above process till we find the space.

**Example:**

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92. Hash them and if collision occurs resolve it with linear probing.

**1) Initially empty table**

| Key | | | | | | | |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**2) Insert 50:**

h(k)= key % 7, h(50) = 50 % 7 = 1

| Key | | 50 | | | | | |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**3) Insert 700:**

h(k)= key % 7, h(700) = 700 % 7 = 0

| Key | 700 | 50 | | | | | |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**4) Insert 76:**

h(k)= key % 7, h(76) = 76 % 7 = 6

| Key | 700 | 50 | | | | | 76 |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**5) Insert 85:**

h(k)= key % 7, h(85) = 85 % 7 = 1 //Collision occurs as 50 is already present at location 1 i.e. Find next location i.e.

h(k') =(key + 1) % 7, h(85')= (85+1)%7= 2 //insert 85 at location 2

| Key | 700 | 50 | 85 | | | | 76 |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**6) Insert 92:**

h(k)= key % 7 = 92 % 7 = 1 //Collision occurs as 50 is already present at location 1, Find next location i.e.

h(k ') =(key + 1) % 7= (92+1)%7= 2 //Collision occurs as 85 is already present at location 2, Find next location i.e.

h(k ') =(key + 2) % 7= (92+2)%7= 3 //insert 92 at location 3

| Key | 700 | 50 | 85 | 92 | | | 76 |
|---|---|---|---|---|---|---|---|
| Location | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**Algorithm:**

Consider x is original array of elements, n is total no of elements in x, and ht is hash table array.

```
Step 1: Enter total no of elements to store and enter those elements.
        Accept: n and values of x[i]
Step 2: Initialize empty hash table, Repeat while i < size
                Set ht[i]=-1 //empty hash table
Step 3: Repeat while i< n
                Set key= x[i];
                Set address=modulodivision(key)
                If address not equals to -1
                        Set address= linearprobe(address)
                else
                        Set address=key
                [End If]
        [End while]
Step 4: End
```

**modulodivision(key)**

```
Step 1: Calculate address as
                address= key%size+1
Step 2: If address equal to size of hash table
                Set address=0
        Else
                Return address
        [End If]
```

**linearprobe(address)**

```
Step 1:Repeat while address in hash table is not empty
                Find next address
                If address == size //address equals to last address of hash
table
                        Set address = 0 //point address to first address in
                hash table
                [End If]
          [End while]
Step 2: Return Address
```

**Program:**

```
#include <stdio.h>
#include <conio.h>
#define size 10
int ht[size];
void store(int  x[ ], int  n);
int modulodivision(int  key);
int linearprobe(int  address);

void main()
{
   int i, n, x[10] ;
   char ch ;
   clrscr();
   printf("Enter the number of elements: ") ;
   scanf("%d",&n) ;
   printf("Enter the elements:\n") ;
   for(i=0 ; i<n ; i++)
   {
      scanf("%d",&x[i]) ;
    }
   store(x,n) ;
   printf("Hashtable is as shown:\n") ;
   for(i=0 ; i<size ; i++)
   {
      printf("%d ", ht[i]) ;
    }
    getch() ;
}

void store(int  x[ ], int  n)
{
   int i, key, address;
  /* Initializing hash table to empty */
   for(i=0 ; i<size ; i++)
      ht[i]=-1;
  /* Copying elements from original array to hashtable */
   for(i=0 ; i<n ; i++)
      {
         key=x[i];
```

```
            address=modulodivision(key);
            if(ht[address]!=-1)
               address=linearprobe(address);
            ht[address]=key;
        }
}

/* Hash Function */
int modulodivision(int  key)
{
   int address;
   address=key%size+1;
   if(address==size)
   {
      return 0;
    }
   else
   {
      return address;
    }
}

/* Collision Resolution */
int linearprobe(int  address)
{
   while(ht[address]!=-1)
        {
           address++;
           if(address==size)
              address=0;
        }
   return address;
}
```

**Questions:**

1. What is the hash function used in linear probing?
   - A. H(x)= key mod table size
   - B. H(x)= (key+ F(i2)) mod table size
   - C. H(x)= (key+ F(i)) mod table size
   - D. H(x)= X mod 17

2. ___________ is not a theoretical problem but actually occurs in real implementations of probing.

A. Hashing
B. Clustering
C. Rehashing
D. Collision

3. Consider a 13-element hash table for which h(k)= key mod 13 is used with integer keys. Assuming linear probing is used for collision resolution, at which location would the key 103 be inserted, if the keys 661, 182, 24 and 103 are inserted in that order?

A. 0
B. 1
C. 11
D. 12

4. A hash function h defined as h(k) = k mod 7, with linear probing, insert the keys 37, 38, 72, 48, 98, 11, 56 into a table. Key 11 will be stored at the location ______.

A. 3
B. 7
C. 2
D. 5

**Self-Learning Topics:**

**Direct and Subtraction hashing:**

In direct hashing the key is the address without any algorithmic manipulation.

Direct hashing is limited and not suitable for large key values, but it can be very powerful because it guarantees that there are no collisions. In Subtraction hashing a fixed number is subtracted from key. It is suitable for small list.

❖❖❖❖

# Module III

**Practical No: 1**

**Aim: Implement program for Stack using Arrays.**

**Objective:** To understand stack operations and to develop a program for implementing stack using array.

**Theory:**

A Stack is a linear data structure that follows the Last-In-First-Out(LIFO) principle. It can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack. A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. Just define a one-dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable called 'top'. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

**Stack Operations using Array:**

**1. push():** In a stack, push() is used to insert an element into the stack. In a stack, the new element is always inserted at top position. Following are the steps to push an element on to the stack.

**Step 1:** Check whether stack is full. (top == n)

**Step 2:** If it is full, then display "Overflow” and terminate the function.

**Step 3:** If it is not full, then increment top value by one (top+1) and set stack[top] to value (stack[top] = value).

**2. pop():** In a stack, pop() is used to delete an element from the stack. In a stack, the element is always deleted from top position. We can use the following steps to pop an element from the stack.

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

**Step 2:** If it is empty, then display "Underflow" and terminate the function.

**Step 3:** If it is not empty, then delete stack[top] and decrement top value by one (top-1).

**Algorithm:**

```
1. Algorithm for push operation:
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
2. Algorithm for pop operation:
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

**Program:**

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{
   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("***Stack operations using array***");
printf("\n----------------------------------\n");
   while(choice != 4)
   {
      printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
```

```
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        }
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
       printf("Stack is empty");
    }
}
```

**Questions:**

1. What does 'stack underflow' refers to?
    - A. Accessing item from an undefined stack
    - B. Adding items to a full stack
    - C. Removing items from an empty stack
    - D. Index out of bounds exception

2. Array implementation of Stack is not dynamic, which of the following statements supports this argument?
    - A. User unable to give the input for stack operations
    - B. Space allocation for array is fixed and cannot be changed during run-time
    - C. A runtime exception halts execution
    - D. Improper program compilation

3. Which of the following array element will return the top of the stack element for a stack of size n elements?
    - A. S[n-1]
    - B. S[n]
    - C. S[n-2]
    - D. S[n+1]

4. Which one of the following is the process of inserting an element in the stack?
    - A. Insert
    - B. Add
    - C. Push
    - D. Pop

5. What is the meaning of Top == -1?
    - A. Stack is empty
    - B. Overflow condition
    - C. Underflow condition
    - D. Stack is full

**Practical No: 2**

**Aim: Implement program for Stack using Linked List.**

**Objective:** To develop a program for implementing stack using Linked List.

**Theory:**

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. In linked list implementation of a stack, 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.

**Stack operations using linked list**

**1. push():** We can use the following steps to push an element into the stack.
Step 1: Create a newNode with the given data.
Step 2: Check whether the stack is empty (TOP == NULL).
Step 3: If it is empty, then set the pointer of the node to NULL.
Step 4: If it is not empty, then make the node point to TOP.
Step 5: Finally, make the newNode as TOP.

**2. pop():** We can use the following steps to pop an element from the stack.
Step 1: Check whether stack is empty (top == NULL).
Step 2: If it is empty, then display "EMPTY STACK"
Step 3: If it is not empty, then create a temporary node and set it to TOP.
Step 4: Print the data of TOP.
Step 5: Make TOP to point to the next node.
Step 6: Delete the temporary node.

**Algorithm:**

1. Algorithm for push() operation:

```
begin
if (TOP == NULL)    //Check whether stack is Empty
newNode -> next = NULL  //if stack is empty
else
newNode -> next = TOP  //if stack is not empty
```

```
TOP= newNode
end
```

2. Algorithm for pop() operation:

```
begin
if (TOP == NULL)   //Check whether stack is Empty
print "EMPTY STACK"
else
create a temporary node, temp = top   //if stack is not empty
print TOP -> data
TOP = TOP -> next
free(temp)
end
```

**Program:**

```
#include<stdio.h>
#include<stdlib.h>
/* Structure to create a node with data and pointer */
struct Node
{
        int data;
        struct Node *next;
}
*top = NULL; // Initially the list is empty
void push(int);
void pop();
void display();

int main()
{
    int choice, value;
    printf("\nIMPLEMENTING STACKS USING LINKED LISTS\n");
        while(1)
        {
                printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
                printf("\nEnter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("\nEnter the value to insert: ");
                        scanf("%d", &value);
                        push(value);
                        break;
```

```
                    case 2: pop();
                    break;

                    case 3: display();
                    break;

                    case 4: exit(0);
                    break;

                    default: printf("\nInvalid Choice\n");
            }
    }
}

void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value; // get value for the node
    if(top == NULL)
    newNode->next = NULL;
    else
    newNode->next = top; // Make the node as TOP
    top = newNode;
    printf("Node is Inserted\n\n");
}

void pop()
{
    if(top == NULL)
    printf("\nEMPTY STACK\n");
    else{
    struct Node *temp = top;
    printf("\nPopped Element : %d", temp->data);
    printf("\n");
    top = temp->next; // After popping, make the next node as TOP
    free(temp);
}
}
void display()
{
    if(top == NULL)
    printf("\nEMPTY STACK\n");
    else
```

```
        {
                printf("The stack is \n");
                struct Node *temp = top;
                while(temp->next != NULL){
                printf("%d--->",temp->data);
                temp = temp -> next;
        }
        printf("%d--->NULL\n\n",temp->data);
}
}
```

**Questions:**

1. If the size of the stack is 8 and we try to add the 9th element in the stack then the condition is known as __________.

    A. Underflow
    B. Garbage collection
    C. Overflow
    D. Empty

2. If the elements '10', '20', '30' and '40' are added in a stack, so what would be the order for the removal?

    A. 10, 20, 30, 40
    B. 20, 10, 30, 40
    C. 40, 30, 20, 10
    D. 30, 10, 20, 40

3. Stack can be implemented using _________ and _________?

    A. Array and Binary Tree
    B. Linked List and Graph
    C. Array and Linked List
    D. Queue and Linked List

4. TOP == NULL represents:

    A. Stack is full
    B. Stack is empty
    C. Overflow condition
    D. Underflow condition

5. Statement top = temp->next does what in the Linked List?

    A. Make the next node as top
    B. Make the current node as top
    C. Make the predecessor node as top
    D. Pop one element form the linked list

## Practical No: 3

**Aim: Implement program for Evaluation of Postfix Expression.**

**Objective:** To develop program for Evaluation of Postfix Expression.

**Theory:**

The Postfix notation is used to represent algebraic expressions.Reverse Polish notation, also known as Polish postfix notation or simply postfix notation, is a mathematical notation in which operators follow their operands. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Example:
Let us consider the given expression as 2 3 1 * + 9 -
We scan all elements one by one.

| **Step** | **Character Scanned** | **Operation** | **Stack Status** | **Calculation** |
|---|---|---|---|---|
| 1 | 2 | Push | 2 | |
| 2 | 3 | Push | 2,3 | |
| 3 | 1 | Push | 2,3,1 | 3*1=3 |
| 4 | * | Pop 2 elements & evaluate | 2,3 | |
| 5 | + | Pop 2 elements & evaluate | 5 | 2+3=5 |
| 6 | 9 | Push | 5,9 | |
| 7 | - | Pop 2 elements & evaluate | -4 | 5-9= -4 |

Explanation:

1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’.
3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’
4) Scan ‘*’, it’s an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3’.
5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.

6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.

7) Scan ‘-‘, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.

8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

**Algorithm:**

**Step 1:**Create a stack to store operands (or values).

**Step 2:**Scan the given expression and do following for every scanned element.

a) If the element is a number, push it into the stack.

b) If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack.

**Step 3:**When the expression is ended, the number in the stack is the final answer.

**Program:**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 50              //max size defined
int stack[MAX];             //a global stack
char post[MAX];             //a global postfix stack
int top=-1;                 //initializing top to -1
void pushstack(int tmp);      //push function
void evaluate(char c);        //calculate function
void main()
{
  int i,l;
  //clrscr();
  printf("Insert a postfix notation :: ");
  gets(post);                 //getting a postfix expression
  l=strlen(post);             //string length
  for(i=0;i<l;i++)
  {
    if(post[i]>='0' && post[i]<='9')
    {
      pushstack(i);             //if the element is a number push it
    }
    if(post[i]=='+' || post[i]=='-' || post[i]=='*' ||
    post[i]=='/' || post[i]=='^')      //if element is an operator
    {
      evaluate(post[i]);            //pass it to the evaluate
    }
  }                  //print the result from the top
  printf("\n\nResult :: %d",stack[top]);
  getch();
}

void pushstack(int tmp)         //definiton for push
{
  top++;                          //incrementing top
  stack[top]=(int)(post[tmp]-48);     //type casting the string to its integer
value
}

void evaluate(char c)      //evaluate function
{
  int a,b,ans;        //variables used
  a=stack[top];       //a takes the value stored in the top
  stack[top]='\0';     //make the stack top NULL as its a string
  top--;               //decrement top's value
  b=stack[top];         //put the value at new top to b
  stack[top]='\0';      //make it NULL
  top--;               //decrement top
  switch(c)     //check operator been passed to evaluate
  {
    case '+':          //addition
      ans=b+a;
      break;
    case '-':          //subtraction
      ans=b-a;
      break;
    case '*':           //multiplication
      ans=b*a;
      break;
    case '/':          //division
      ans=b/a;
      break;
    case '^':     //power
      ans=b^a;
      break;
    default:
      ans=0;      //else 0
  }
  top++;            //increment top
  stack[top]=ans;       //store the answer at top
}
```

**Questions:**

1. Which of the following is an example for a postfix expression?
   A. a*b(c+d)
   B. abc*+de-+
   C. +ab
   D. a+b-c

2. While evaluating a postfix expression, when an operator is encountered, what is the correct operation to be performed?
   A. Push it directly on to the stack
   B. Pop 2 operands, evaluate them and push the result on to the stack
   C. Pop the entire stack
   D. Ignore the operator

3. What is the result of the following postfix expression?
   ab*cd*+ where a=2,b=2,c=3,d=4.
   A. 16
   B. 12
   C. 14
   D. 10

4. Evaluate and write the result for the following postfix expression
   abc*+de*f+g*+ where a=1, b=2, c=3, d=4, e=5, f=6, g=2.
   A. 61
   B .59
   C. 60
   D. 55

5. What is the other name for a postfix expression?
   A. Normal polish Notation
   B. Reverse polish Notation
   C. Warsaw notation
   D. Infix notation

6. Data Structure required to evaluate postfix expression is __________.
   A. Heap
   B. Stack
   C. Pointer
   D. Queue

## Practical No: 4

**Aim: Implement program for balancing of parenthesis.**

**Objective:** To understand applications of stack in balancing of parenthesis and to implement program for same using stack.

**Theory:**

A stack can be used for syntax verification of the arithmetic expression for ensuring that for each left parenthesis in the expression there is a corresponding right parenthesis.

To accomplish this task the expression is scanned from left to right character by character.

1. Whenever a left parenthesis is encountered, we push it onto the stack. It could be of any type, square brace [, round brace (, or curly brace {.
2. When we encounter a right parenthesis], or), or}, the status of the stack is checked.
   a. If the stack is empty and we have a right parenthesis in the expression that does not have corresponding left parenthesis then there is mistake in expression.
   b. If the stack is not empty, we will pop the topmost element from the stack and compare it with the scanned right parenthesis.
3. If both the parenthesis is not of the same type then it shows a mistake in expression. But if both parentheses are of same type, then same procedure is repeated until the whole expression is scanned and stack is empty

Let us check the order of brackets in an expression I:
I= [(5+6)*7-{7/4}+(3*2)-8]

| Character scanned | Status of stack |
|---|---|
| [ | [ |
| ( | [( |
| ) | [ |
| { | [{ |
| } | [ |
| ( | [( |
| ) | [ |
| ] | Null |

Expression is balanced as every left parenthesis is having corresponding right parenthesis.

**Algorithm:**

**Algorithm to check balanced parenthesis**

**Step 1:** Initialize a character stack. Set top pointer of stack to -1.

**Step 2:** Find length of input string using strlen function and store it in an integer variable "length".

**Step 3:** Using a for loop, traverse input string from index 0 to length-1.

**Step 4:** a.If current character is open parenthesis, then push it inside stack.

b. If current character is closing parenthesis, then pop a character from stack.

c. If stack is empty, then input string is invalid, it means there is no matching opening parenthesis corresponding to closing parenthesis.

**Step 5:** After complete traversal of input string, If stack is empty then input expression is a Valid expression otherwise Invalid.

**Program:**

```
#include<string.h>
#include<conio.h>
#define MAX 20
struct stack
{
        char stk[MAX];
        int top;
}s;

void push(char item)
{
        if (s.top == (MAX - 1))
        printf ("Stack is Full\n");
        else
        {
                s.top = s.top + 1; // Push the char and increment top
                s.stk[s.top] = item;
        }
}

void pop()
{
        if (s.top == - 1)
        {
                printf ("Stack is Empty\n");
        }
        else
        {
        s.top = s.top - 1; // Pop the char and decrement top
        }
}

int main()
{
        char exp[MAX];
        int i = 0;
        s.top = -1;
        printf("\nINPUT THE EXPRESSION : ");
        scanf("%s", exp);
        for(i = 0;i < strlen(exp);i++)
        {
        if(exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
        {
                push(exp[i]); // Push the open bracket
                continue;
        }
        else if(exp[i] == ')' || exp[i] == ']' || exp[i] == '}') // If a closed
bracket is encountered
        {
                if(exp[i] == ')')
                {
                        if(s.stk[s.top] == '(')
                        {
                                pop(); // Pop the stack until closed bracket is
                        found
                        }
                        else
                        {
                                printf("\nUNBALANCED
                        EXPRESSION\n");
                                break;
                        }
                }
                if(exp[i] == ']')
                {
                        if(s.stk[s.top] == '[')
```

```
                    {
                    pop(); // Pop the stack until closed bracket is found
                    }
                    else
                    {
                            printf("\nUNBALANCED
                    EXPRESSION\n");
                            break;
                    }
            }
            if(exp[i] == '}')
            {
                    if(s.stk[s.top] == '{')
                    {
                    pop(); // Pop the stack until closed bracket is found
                    }
                    else
                    {
                            printf("\nUNBALANCED
                    EXPRESSION\n");
                            break;
                    }
            }
    }
    }
    if(s.top == -1)
    {
            printf("\nBALANCED EXPRESSION\n"); // Finally if the
    stack is empty, display that the expression is balanced
    }
    getch();
}
```

**Questions:**

1. In balancing parentheses algorithm, the string is read from?
    A. Right to left
    B. Left to right
    C. Center to right
    D. Center to left

2. Which is the most appropriate data structure for applying balancing of parentheses algorithm?

A. Stack

B. Queue

C. Tree

D. Graph

3. Which of the following does the balancing symbols algorithm include?

A. Balancing double quotes

B. Balancing single quotes

C. Balancing operators and brackets

D. Balancing parentheses, brackets and braces

4. What should be done when an opening parenthesis is read in a balancing symbols algorithm?

A. Push it on to the stack

B. Throw an error

C. Ignore the parentheses

D. Pop the stack

5. If the corresponding end bracket/braces/parentheses is encountered, which of the following is done?

A. Push it on to the stack

B. Pop the stack

C. Throw an error

D. Treated as an exception

6. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. The maximum number of parentheses that appear on the stack at any one time when the algorithm analyzes: (()(())(())) are:

A. 1

B. 2

C. 3

D. 4 or more

**Self-Learning Topic:**

**Conversion of infix notation to postfix notation:**

When the operator is written in between the operands, then it is known as infix notation. The postfix expression is an expression in which the operator is written after the operands. Postfix notation is very easily implemented and does not have overhead of parentheses and there is no complication of precedence of one operator over the other. To convert infix expression to postfix expression stack data structure will be used.

# Module IV

## Data structures

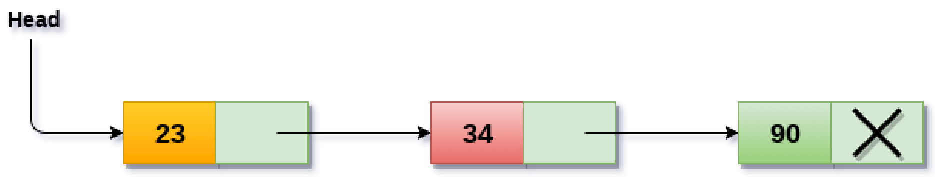
## Form No .: 5

1. **Point:** To implement a singly linked list using C / C ++

**Objective:** Get familiar with the List interface. Understand how to write a matrix based on.

**Theory:**

- The individually linked list can be defined as the collection of ordered sets of elements. The number of items may vary according to the needs of the program. A node in the individual linked list consists of two parts: data part and link part. The data part of the node stores the actual information that will be represented by the node, while the link part of the node stores the address of its immediate successor.
- One-way strings or individually linked lists can only be traversed in one direction. In other words, we can say that each node contains only the next pointer, so we cannot traverse the list in the reverse direction.
- Consider an example where the student's grades in three subjects are stored in a linked list as shown in the figure.



In the figure above, the arrow represents the links. The data part of each node contains the marks obtained by the student in the various subjects. The last node in the list is identified by the null pointer in the address part of the last node. We can have all the elements we need, in the data part of the list.

**Algorithm:**

**Operations on a single linked list**

The following operations are performed on a single linked list

- **Insertion**
- **deletion**
- **Show**

Before implementing the actual operations, we must first set up an empty list. First, perform the following steps before implementing the actual operations.

- Step 1: Include all header files used in the program.
- Step 2: Declare all user-defined functions.
- Step 3: Define a node structure with two-membered data, then
- Step 4: Define a 'head' node pointer and set it to NULL.
- Step 5: Implement the main method by displaying the operations menu and make appropriate function calls in the main method to perform the operation selected by the user.

**Insertion**

In a single linked list, the insert operation can be performed in three ways. Are the following...

1. Insert at the beginning of the list
2. Enter to the end of the list
3. Insert in a specific position in the list

**Insert at the beginning of the list**

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

- Step 1: Create a new node with a certain value.
- Step 2: Check if the list is empty (head == NULL)
- Step 3: If blank, configure **newNode → next** = NULL and head = newNode.
- Step 4: If it's not empty, configure **newNode → next** = head and head = new knot.

**Enter to the end of the list**

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

- Step 1: Create a new node with a certain value e **newNode → next** as NULL.

- Step 2: Check if the list is empty (head == NULL).
- Step 3: If empty, set head = newNode.
- Step 4: If it is not empty, define a node pointer temperature and initialize with head.
- Step 5: Keep moving the temperature to the next node until it reaches the last node in the list (until **temperature → next** equal to NULL).
- Step 6 - Set up **temperature → next** = newNode.

**Insert in a specific position 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 new node with a certain value.
- Step 2: Check if the list is empty (head == NULL)
- Step 3: If blank, configure **newNode → next** = NULL and head = newNode.
- Step 4: If it is not empty, define a node pointer temperature and initialize with head.
- Step 5: Keep moving the temperature on its next node until it reaches the node, after which we want to insert the new node (until **temp1 → data** is equal to location, here location is the value of the node after which we want to insert the newNode).
- Step 6: Check each time the temperature has reached the last knot or not. If the last node is reached, 'The specified node is not in the list! Entry is not possible !!! 'and terminate the function. Otherwise, move the temperature to the next node.
- Step 7 - Finally, configure '**newNode → next** = **temperature → next**'S'**temperature → next** = newNode '

**deletion**

In a single linked list, the delete operation can be performed in three ways. Are the following...

1. Delete from the top of the list
2. Remove from the end of the list
3. Delete a specific node

**Delete from the top of the list**

We can use the following steps to remove a node from the beginning of the unique linked list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and the function ends.
- Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.
- Step 4: Check if the list has only one node (**temperature → next** == NULL)
- Step 5: If TRUE, set head = NULL and remove the temperature (setting the conditions of the empty list)
- Step 6: If FALSE, set head = **temperature → next**and eliminates temp.

**Remove from the end of the list**

We can use the following steps to remove a node from the end of the unique linked list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and the function ends.
- Step 3: If it is not empty, define two pointers to the node "temp1" and "temp2" and initialize "temp1" with head.
- Step 4: Check if the list has only one node (**temp1 → next** == NULL)
- Step 5: if it is TRUE. Then set head = NULL and remove temp1. And finish the show. (Setting the empty list condition)
- Step 6: If it is FALSE. Then set 'temp2 = temp1' and move temp1 to your next node. Repeat the same until you reach the last node in the list. (as far as**temp1 → next** == NULL)
- Step 7 - Finally, configure **temp2 → next** = NULL and remove temp1.

**Remove a specific node from the list**

We can use the following steps to remove a specific node from the unique linked list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and the function ends.

- Step 3: If it is not empty, define two pointers to the node "temp1" and "temp2" and initialize "temp1" with head.
- Step 4: Continue moving temp1 until you reach the exact node to be removed or the last node. And each time, set 'temp2 = temp1' before moving 'temp1' to your next node.
- Step 5: If the last node is reached, display 'The specified node is not in the list! Cancellation is not possible !!! '. And finish the show.
- Step 6: If the exact node we want to remove is reached, check if the list has only one node or not
- Step 7: If the list has only one node and this is the node to remove, set head = NULL and remove temp1 (free (temp1)).
- Step 8: If the list contains multiple nodes, check if temp1 is the first node in the list (temp1 == head).
- Step 9: If temp1 is the first node, move your head to the next node (**head = head → forward**) and remove temp1.
- Step 10: If temp1 is not the first node, check if it is the last node in the list (**temp1 → next == NULL**).
- Step 11: If temp1 is the last node, configure **temp2 → next** = NULL and remove temp1 (free (temp1)).
- Step 12: If temp1 is not the first node and not the last node, set **temp2 → next** = **temp1 → next** and remove temp1 (free (temp1)).

**View a single linked list**

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

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display "List is empty!" and terminate the function.
- Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.
- Step 4: Keep showing **temperature → data** with an arrow (--->) until the temperature reaches the last knot
- Step 5 - Finally it shows **temperature → data** with the arrow pointing to NULL (**temp → data ---> NULL**).

**Program:**

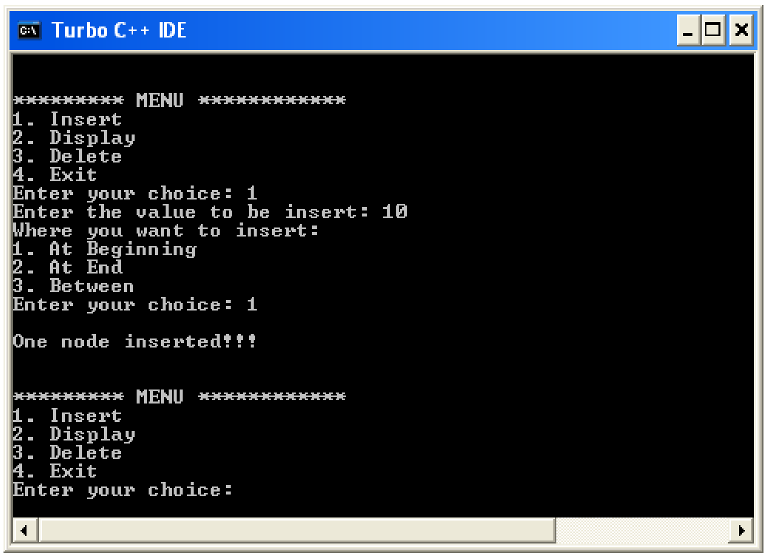
```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void insertAtBeginning (int);
void insertAtEnd (int);
void insertBetween (int, int, int);
blank display ();
void removeBeginning ();
void removeEnd ();
void removeSpecific (int);
Structure node
{
 int data;
 struct Node * next;
} * head = NULL;
main vacuum ()
{
 int choice, value, choice1, loc1, loc2;
 clrscr ();
 while (1) {
 mainMenu: printf ("\ n \ n ****** MENU ****** \ n1. Insert \ n2. Show \
n3. Delete \ n4. Exit \ nEnter your choice:");
 scanf ("% d", & option);
 change (choice)
 {
 case 1: printf ("Insert the value to insert:");
                      scanf ("% d", & value);
                      while (1) {
printf ("Where do you want to enter: \ n1. At the beginning \ n2. At the
end \ n3. Enter \ nPlease enter your choice:");
```

```
                        scanf ("% d", & choice1);
                        switch (option 1)
                        {
                         case 1: insertAtBeginning (value);
                                         break time;
                         case 2: insertAtEnd (value);
                                         break time;
                case 3: printf ("Insert the two values where you want to insert:");
                                         scanf ("% d% d", & loc1, & loc2);
                                         insertBetween (value, loc1, loc2);
                                         break time;
                         default: printf ("\ nIncorrect entry! Try again! \ n \ n");
                                         go to the main menu;
                        }
                        go to subMenuEnd;
                        }
                        submenu End:
                        break time;
        case 2: display ();
                        break time;
        case 3: printf ("How do you want to remove: \ n1. From the beginning \
       n2. From the end \ n3. Specific \ nPlease enter your choice:");
                        scanf ("% d", & choice1);
                        switch (option 1)
                        {
                         case 1: removeBeginning ();
                                         break time;
                         case 2: removeEnd ();
                                         break time;
                         case 3: printf ("Enter the value you want to remove:");
                                         scanf ("% d", & loc2);
```

```
                                        removeSpecific (loc2);
                                        break time;
                     default: printf ("\ nIncorrect entry! Try again! \ n \ n");
                                        go to the main menu;
                     }
                     break time;
 case 4: exit (0);
 default: printf ("\ nIncorrect entry !!! Try again !! \ n \ n");
 }}}
void insertAtBeginning (int value)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode-> data = value;
 yes (head == NULL)
 {
 newNode-> next = NULL;
 head = newNode;
 }
 the rest
 {
 newNode-> next = head;
 head = newNode;
 }
 printf ("\ nA node entered !!! \ n");
}
void insertAtEnd (int value)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode-> data = value;
 newNode-> next = NULL;
 yes (head == NULL)
         head = newNode;
 the rest
 {
 struct Node * temp = head;
 while (temp-> next! = NULL)
         temp = temp-> next;
 temp-> next = newNode;
 }
 printf ("\ nA node entered !!! \ n");
}
void insertBetween (int value, int loc1, int loc2)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode-> data = value;
 yes (head == NULL)
 {
 newNode-> next = NULL;
 head = newNode;
 }
 the rest
 {
 struct Node * temp = head;
 while (temp-> data! = loc1 && temp-> data! = loc2)
         temp = temp-> next;
 newNode-> next = temp-> next;
 temp-> next = newNode;
 }
 printf ("\ nA node entered !!! \ n");
}

void removeBeginning ()
{
 yes (head == NULL)
           printf ("\ n \ nThe list is empty!");
 the rest
 {
 struct Node * temp = head;
 yes (header-> next == NULL)
 {
            head = NULL;
            free (temperature);
 }
 the rest
 {
           head = temp-> next;
           free (temperature);
           printf ("\ nA node removed !!! \ n \ n");
 }}}
void removeEnd ()
{
 yes (head == NULL)
 {
 printf ("\ nThe list is empty! \ n");
 }
 the rest
 {
```

```
 struct Node * temp1 = head, * temp2;
 yes (header-> next == NULL)
          head = NULL;
 the rest
 {
           while (temp1-> next! = NULL)
           {
           temp2 = temp1;
           temp1 = temp1-> next;
           }
           temp2-> next = NULL;
 }
 free (temp1);
 printf ("\ nA node removed !!! \ n \ n");
 }
}
void removeSpecific (int delValue)
{
 struct Node * temp1 = head, * temp2;
 while (temp1-> data! = delValue)
 {
 if (temp1 -> next == NULL) {
          printf ("\ nNo nodes found in the list !!!");
          goto functionEnd;
 }
 temp2 = temp1;
 temp1 = temp1 -> next;
 }
 temp2 -> next = temp1 -> next;
 free (temp1);
 printf ("\ nA node removed !!! \ n \ n");
 End function:
}
blank screen ()
{
 yes (head == NULL)
 {
 printf ("\ nThe list is empty \ n");
 }
 the rest
 {
 struct Node * temp = head;
 printf ("\ n \ nThe elements of the list are - \ n");
 while (temp-> next! = NULL)
 {
         printf ("% d --->", temp-> data);
         temp = temp-> next;
 }
 printf ("% d ---> NULL", temp-> data);
 }}
```

**Production:**



```
********* MENU ************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Where you want to insert:
1. At Beginning
2. At End
3. Between
Enter your choice: 1

One node inserted!!!


********* MENU ************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice:
```

## 1) Search in an individually linked list

The search is performed to find the position of a particular item in the list. Searching for any item in the list requires you to scroll through the list and compare each item in the list with the specified item. If the element matches one of the elements in the list, the function returns the position of the element.

**Algorithm:**

- Step 1: SET PTR = HEAD
- Step 2: Set I = 0
- STEP 3: IF PTR = NULL
- WRITE "EMPTY LIST" GOTO STEP 8 END OF YES STEP 4: REPEAT STEPS 5 TO 7 UNTIL PTR! = NULL

- STEP 5: if ptr → data = item
- write i + 1 End of IFSTEP 6: I = I + 1

- STEP 7: PTR = PTR → NEXT
- [END OF LOOP] STEP 8: EXIT

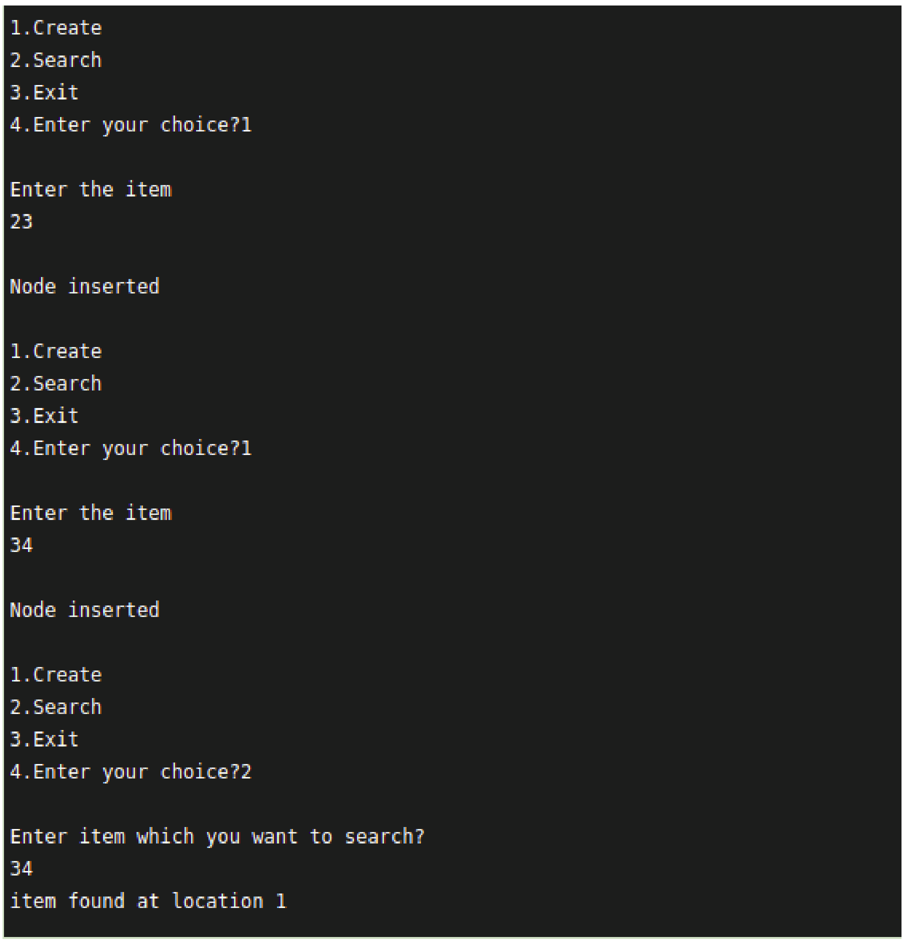
**Program:**

```
#include <stdio.h>
#include <stdlib.h>
empty create (int);
empty Research();
structure node
{
 int data;
 next structure node *;
};
knot structure * head;
empty principal ()
{
 int choice, object, loc;
 do
 {
 printf ("\ n1.Create \ n2.Search \ n3.Sexit \ n4.Enter your choice?");
```

```
scanf ("% d", & option);
change (choice)
{
Case 1:
printf ("\ nInsert element \ n");
scanf ("% d", & element);
create (element);
break time;
case 2:
Research();
case 3:
output (0);
break time;
default:
printf ("\ nPlease enter a valid option \ n");
}
} while (choice! = 3);
}
create empty (int element)
{
structure node * ptr = (structure node *) malloc (size of (structure node
*));
yes (ptr == NULL)
{
printf ("\ nOVERFLOW \ n");
}
the rest
{
ptr-> data = element;
ptr-> next = head;
head = ptr;
printf ("\ nNode inserted \ n");
```

```
}}
empty Research()
{
 struct node * ptr;
 int element, i = 0, flag;
 ptr = head;
 yes (ptr == NULL)
 {
 printf ("\ nEmpty list \ n");
 }
 the rest
 {
 printf ("\ nPlease enter the element you want to search for? \ n");
 scanf ("% d", & element);
 while (ptr! = NULL)
 {
 if (ptr-> data == element)
 {
 printf ("article found at location% d", i + 1);
 flag = 0;
 }
 the rest
 {
 flag = 1;
 }
 i ++;
 ptr = ptr -> next;
 }
 yes (flag == 1)
 {
 printf ("Item not found \ n");
 }}}
```

**Production:**



```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
34

Node inserted

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
34
item found at location 1
```

**2) Individually count the total number of nodes in the linked list**

**Algorithm:**

**%%Entrance** : parent node of the linked list

**Start:**

```
 count 0
 If (test! = NULL) then
 head temperature
 While (temp! = NULL) do
 count ← count + 1
temperature ← temperature. following
 Finish in the meantime
 It will end if
 write ('Total nodes in list =' + count)
end
```

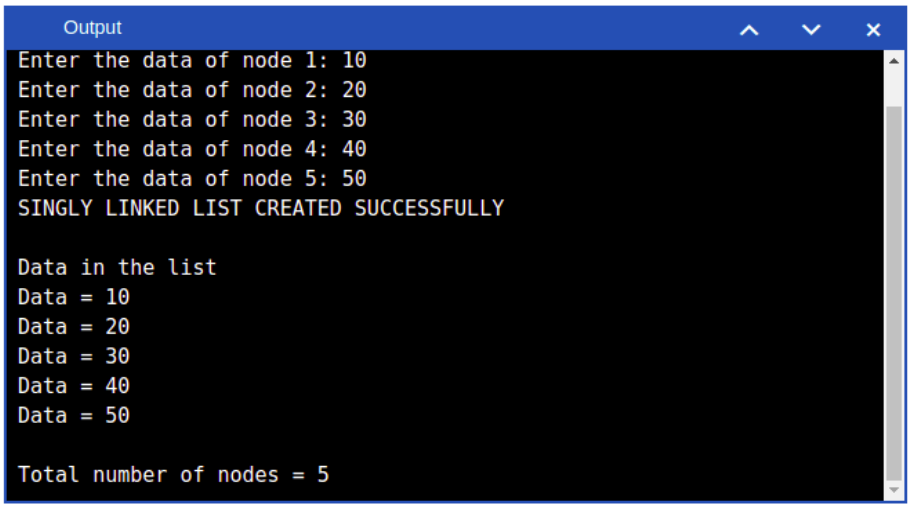
**Program:**

```
#include <stdio.h>
#include <stdlib.h>
/ * Structure of a node * /
node structure {
int data; // Data
next structure node *; // Address
}*head;
void createList (int n);
int countNodes ();
void displayList ();
main integer ()
{
 int n, total;
 / *
 * Create a linked list of n nodes
 * /
 printf ("Enter the total number of nodes:");
 scanf ("% d", & n);
 createList (n);
 printf ("\ nData in the list \ n");
 displayList ();
 / * Counts the number of nodes in the list * /
 total = countNodes ();
 printf ("\ nTotal number of nodes =% d \ n", total);
 returns 0;
}
/ *
 * Create a list of n nodes
 * /
```

```
empty createList (int n)
{
 struct node * newNode, * temp;
 int data, i;
 head = (structure node *) malloc (size of (structure node));
 / *
 * If you cannot allocate memory for the head root node
 * /
 yes (head == NULL)
 {
 printf ("Unable to allocate memory");
 }
 the rest
 {
 / *
 * Read data from user node
 * /
 printf ("Enter data for node 1:");
 scanf ("% d", & data);
head-> data = data; // Link the data field with the data
head-> next = NULL; // Map the address field to NULL
 temperature = head;
 / *
 * Create n nodes and add to linked list
 * /
 for (i = 2; i <= n; i ++)
 {
 newNode = (structure node *) malloc (size of (structure node));
 / * If no memory is allocated for newNode * /
 yes (newNode == NULL)
 {
 printf ("Unable to allocate memory");
 break time;
 }
 the rest
 {
 printf ("Insert data for node% d:", i);
 scanf ("% d", & data);
newNode-> data = data; // Associate the data field of newNode with the
data
newNode-> next = NULL; // Associate the address field of newNode with
NULL
temp-> next = newNode; // Associate the previous node, ie temporarily
with the new node
 temp = temp-> next;
 }}
 printf ("SIMPLE CONNECTED LIST CREATED SUCCESSFULLY \
n");
 }}
/ *
 * Counts the total number of nodes in the list
 * /
int countNodes ()
{
 int count = 0;
 struct node * temp;
 temperature = head;
 while (temp! = NULL)
 {
 count ++;
 temp = temp-> next;
 }
 counting of returns;
}
/ *
```

```
 * Show the complete list
 * /
empty displayList ()
{
 struct node * temp;
 / *
 * If the list is empty, ie head = NULL
 * /
 yes (head == NULL)
 {
 printf ("The list is empty");
 }
 the rest
 {
 temperature = head;
 while (temp! = NULL)
 {
printf ("Data =% d \ n", temp-> data); // Print the data of the current node
temp = temp-> next; // Go to the next node
 }}}
```

**Production:**



```
Enter the data of node 1: 10
Enter the data of node 2: 20
Enter the data of node 3: 30
Enter the data of node 4: 40
Enter the data of node 5: 50
SINGLY LINKED LIST CREATED SUCCESSFULLY

Data in the list
Data = 10
Data = 20
Data = 30
Data = 40
Data = 50

Total number of nodes = 5
```

**3) Reverse a linked list**

Given a pointer to the parent node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing the links between the nodes.

**Examples of**:

**Entrance**: Head of the next linked list

1-> 2-> 3-> 4-> NULL

**Production**: the linked list must be changed to,

4-> 3-> 2-> 1-> NULL

**Entrance**: Head of the next linked list

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

**Production**: the linked list must be changed to,

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

**Entrance**: NOTHING

**Production**: NOTHING

**Entrance**: 1-> NULL

**Production**: 1-> NULL

**Iterative method**

1. Initializes three pointers prev as NULL, curr as head and next as NULL.
2. Iterate through the linked list. In a loop, do the following. // Before changing the next one from the current one, // memorize the next node next = curr-> next // Now change the next one from the current one // This is where the actual inversion takes place curr-> next = previous // Move previous and current one step forward previous = curr curr = next

   **Program:**

```
#include <iostream>

using the std namespace;

/ * Link List Node * /

node structure {

 int data;

 struct Node * next;

 Node (int data)
```

```
{
 this-> data = data;
 next = NULL;
 }
};
struct LinkedList {
 Knot * head;
LinkedList () {head = NULL; }

 / * Function to invert the linked list * /
 reverse vacuum ()
 {
 // Initialize current, previous and
 // next pointers
 Current * node = head;
 Node * previous = NULL, * next = NULL;
 while (current! = NULL) {
 // Memorize the next one
 next = current-> next;
 // Pointer to the current inverse node
 current-> next = previous;
 // Move pointers one position forward.
 previous = current;
 current = next;
 }
 head = front;
 }
 / * Function to print the linked list * /
 blank print ()
 {
```

```
 struct Node * temp = head;
 while (temp! = NULL) {
 cout << temp-> data << "";
 temp = temp-> next;
 }
 }

 empty push (int data)
 {
 Node * temp = new Node (data);
 temp-> next = head;
 head = temperature;
 }
};

/ * Driver code * /
main integer ()
{
 / * Start with empty list * /
 LinkedList ll;
 ll.push (20);
 ll. press (4);
 ll.push (15);
 ll.push (85);

 cout << "Linked list given \ n";
 ll.print ();
 ll.reverse ();
 cout << "\ nInverted linked list \ n";
 ll.print ();
 returns 0;
}
```

**Production:**

```
Linked list given
85 15 4 20
Inverted linked list
20 4 15 85
```

**Question:**

1. What does the following function do for a given Linked List with first node as head?

```
void fun1(struct node* head)
{
if(head == NULL)
return;
fun1(head->next);
printf("%d ", head->data);
}
```

1. A linear collection of data elements where the linear node is given by means of a pointer is called?

   A. linked list

   B. node list

   C. primitive list

   D. None of these

2. What is the output of the following function for starting pointing to the first node of the following linked list? 1->2->3->4->5->6

```
void fun(struct node* start)
{
if(start == NULL)
return;
printf("%d ", start->data);
if(start->next != NULL )
fun(start->next->next);
printf("%d ", start->data);
}
```

A. 1 4 6 6 4 1

B. 1 3 5 1 3 5

C. 1 2 3 5

D. 1 3 5 5 3 1

3. Linked lists are not suitable for the implementation of?

A. Insertion sort

B. Radix sort

C. Polynomial manipulation

D. Binary search

4. Which of these is an application of linked lists?

A. To implement file systems

B. For separate chaining in hash-tables

C. To implement non-binary trees

D. All of the mentioned

**2. Objective:** To implement a circular linked list using C / C ++

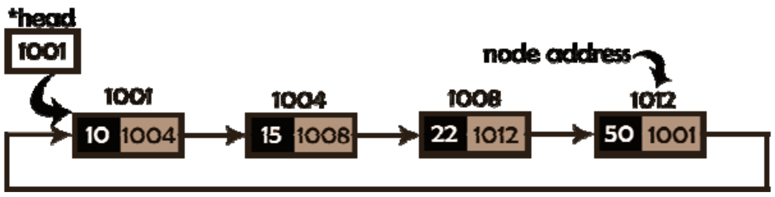
**Objective:**To make it convenient for the operating system to use a circular list so that when you reach the end of the list you can scroll to the top of the list.

**Theory:**

**What is a circular linked list?**

- In a single linked list, each node points to the next node in the sequence and the last node points to NULL. But in a circular linked list, each node points to the next node in the sequence, but the last node points to the first node in the list.
- A circular linked list is a sequence of elements where each element has a link to the next element in the sequence and the last element has a link to the first element.
- This means that the circular linked list is similar to the single linked list, except that the last node points to the first node in the list.

**Example**



**Operations**

In a circular linked list, we do the following ...

1. Insertion
2. deletion
3. Show

Before implementing the actual operations, we must first set up an empty list. First, perform the following steps before implementing the actual operations.

- Step 1: Include all header files used in the program.
- Step 2: Declare all user-defined functions.
- Step 3: Define a node structure with two-membered data, then
- Step 4: Define a 'head' node pointer and set it to NULL.
- Step 5: Implement the main method by displaying the operations menu and make appropriate function calls in the main method to perform the operation selected by the user.

**Insertion**

In a circular linked list, the insert operation can be performed in three ways. Are the following...

1. Insert at the beginning of the list
2. Enter to the end of the list
3. Insert in a specific position in the list

**Insert at the beginning of the list**

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

- Step 1: Create a new node with a certain value.
- Step 2: Check if the list is empty (head == NULL)
- Step 3: If empty, set head = newNode and **newNode → next** = head.
- Step 4: If it is not empty, define a pointer to the 'temp' node and initialize with 'head'.
- Step 5: Keep moving 'temp' to the next node until it reaches the last node (up to '**temperature → next** == head ').
- Step 6 - Set '**newNode → next** = head ',' head = newNode 'e'**temperature → next** = head '.

**Enter to the end of the list**

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

- Step 1: Create a new node with a certain value.
- Step 2: Check if the list is empty (head == NULL).
- Step 3: If empty, set head = newNode and **newNode → next** = head.
- Step 4: If it is not empty, define a node pointer temperature and initialize with head.
- Step 5: Keep moving the temperature to the next node until it reaches the last node in the list (until **temperature → next** == head).
- Step 6 - Set up **temperature → next** = newNode e **newNode → next** = head.

**Insert in a specific position in the list (after a node)**

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

- Step 1: Create a new node with a certain value.

- Step 2: Check if the list is empty (head == NULL)
- Step 3: If empty, set head = newNode and **newNode → next** = head.
- Step 4: If it is not empty, define a node pointer temperature and initialize with head.
- Step 5: Keep moving the temperature on its next node until it reaches the node after which we want to insert the new node (up to **temp1 → data** is equal to location, here location is the value of the node after which we want to insert the newNode).
- Step 6: Check each time if the temperature has been reached or not until the last node. If the last node is reached, 'The specified node is not in the list! Insertion not possible !!! 'and terminate the function. Otherwise, move the temperature to the next node.
- Step 7: If the temperature you reach the exact node after which you want to insert the newNode and then check if it is the last node (temp → next == head).
- Step 8: If the temperature is the last knot, set **temperature → next** = newNode e **newNode → next** = head.
- Step 8: If the temperature is not the last knot, set **newNode → next** = **temperature → next** Yup **temperature → next** = newNode.

**deletion**

In a circular linked list, the delete operation can be performed in three ways, which are as follows ...

1. Delete from the top of the list
2. Remove from the end of the list
3. Delete a specific node

**Delete from the top of the list**

We can use the following steps to remove a node from the beginning of the circular linked list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.
- Step 3: If it is not empty, define two node pointers "temp1" and "temp2" and initialize both "temp1" and "temp2" with head.
- Step 4: Check if the list has only one node (**temp1 → next** == head)

- Step 5: If TRUE, set head = NULL and remove temp1 (setting conditions of empty list)
- Step 6: If it is FALSE, move temp1 until it reaches the last node. (as far as**temp1 → next** == head)
- Step 7: Then set the head = **temp2 → next**, **temp1 → next** = test and clear temp2.

**Remove from the end of the list**

We can use the following steps to remove a node from the end of the linked circular list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.
- Step 3: If it is not empty, define two pointers to the node "temp1" and "temp2" and initialize "temp1" with head.
- Step 4: Check if the list has only one node (**temp1 → next** == head)
- Step 5: if it is TRUE. Then set head = NULL and remove temp1. And finish the show. (Setting the empty list condition)
- Step 6: If it is FALSE. Then set 'temp2 = temp1' and move temp1 to your next node. Repeat the same until temp1 reaches the last node in the list. (as far as**temp1 → next** == head)
- Step 7 - Set up **temp2 → next** = test and delete temp1.

**Remove a specific node from the list**

We can use the following steps to remove a specific node from the linked circular list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.
- Step 3: If it is not empty, define two pointers to the node "temp1" and "temp2" and initialize "temp1" with head.
- Step 4: Continue moving temp1 until you reach the exact node to be removed or the last node. And each time, set 'temp2 = temp1' before moving 'temp1' to your next node.
- Step 5: If the last node is reached, display 'The specified node is not in the list! It cannot be canceled !!! '. And finish the show.

- Step 6: If the exact node we want to remove is reached, check if the list has only one node (**temp1 → next** == head)
- Step 7: If the list has only one node and this is the node to remove, set head = NULL and remove temp1 (free (temp1)).
- Step 8: If the list contains multiple nodes, check if temp1 is the first node in the list (temp1 == head).
- Step 9: If temp1 is the first node, set temp2 = head and keep moving temp2 to the next node until temp2 reaches the last node. Then set**head = head → forward**, **temp2 → next**t = head and delete temp1.
- Step 10: If temp1 is not the first node, check if it is the last node in the list (**temp1 → next == head**).
- Step 1 1- If temp1 is the last node, configure **temp2 → next** = head and remove temp1 (free (temp1)).
- Step 12: If temp1 is not the first node and not the last node, set **temp2 → next** = **temp1 → next** and remove temp1 (free (temp1)).

**View a circular linked list**

We can use the following steps to view the elements of a circular linked list ...

- Step 1: Check if the list is empty (head == NULL)
- Step 2: If blank, show 'List is empty !!!' and terminate the function.
- Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.
- Step 4: Keep showing **temperature → data** with an arrow (--->) until the temperature reaches the last knot
- Step 5 - Finally it shows **temperature → data** with arrow pointing to **head → data**.

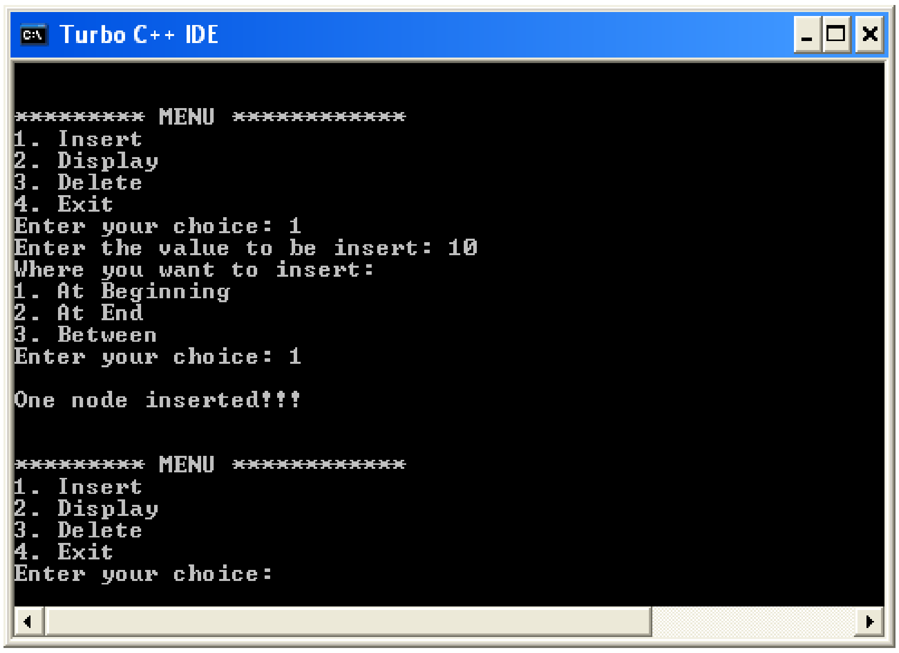
**Program:**

```
#include <stdio.h>

#include <conio.h>

void insertAtBeginning (int);

void insertAtEnd (int);

void insertAtAfter (int, int);

void deleteBeginning ();
void deleteEnd ();
void deleteSpecific (int);
blank display ();
Structure node
{
 int data;
 struct Node * next;
} * head = NULL;
main vacuum ()
{
 int choice1, choice2, value, position;
 clrscr ();
 while (1)
 {
 printf ("\ n *********** MENU ************* \ n");
 printf ("1. Enter \ n2. Delete \ n3. Screen \ n4. Exit \ nEnter your
choice:");
 scanf ("% d", & choice1);
 Change ()
 {
 case 1: printf ("Insert the value to insert:");
 scanf ("% d", & value);
 while (1)
 {
printf ("\ nSelect from the following insert options \ n");
printf ("1. At the beginning \ n2. At the end \ n3. After a node \ n4. Cancel
\ nEnter your choice:");
 scanf ("% d", & choice2);
 switch (option 2)
 {
 case 1: insertAtBeginning (value);
 break time;
 case 2: insertAtEnd (value);
 break time;
 case 3: printf ("Enter the position after which you want to insert:");
 scanf ("% d", & location);
 insertAfter (value, position);
 break time;
 case 4: go to EndSwitch;
 default: printf ("\ nSelect the correct insert option! \ n");
 }
 }
 case 2: while (1)
 {
printf ("\ nSelect from the following delete options \ n");
printf ("1. At the beginning \ n2. At the end \ n3. Specific node \ n4.
Cancel \ nEnter your choice:");
 scanf ("% d", & choice2);
 switch (option 2)
 {
 case 1: deleteBeginning ();
 break time;
 case 2: deleteEnd ();
 break time;
 case 3: printf ("Enter the value of the node to be removed:");
 scanf ("% d", & location);
 deleteSpecic (location);
 break time;
 case 4: go to EndSwitch;
 default: printf ("\ nSelect the correct delete option! \ n");
 }
 }
 Limit switch: pause;
 case 3: display ();
 break time;
 case 4: exit (0);
 default: printf ("\ nSelect the correct option!");
 }}}
void insertAtBeginning (int value)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode -> data = value;
 yes (head == NULL)
 {
 head = newNode;
 newNode -> next = head;
 }
 the rest
 {
 struct Node * temp = head;
 while (temp -> next! = head)
 temp = temp -> next;
 newNode -> next = head;
 head = newNode;
 temp -> next = head;
 }
 printf ("\ nThe entry was successful!");
}
void insertAtEnd (int value)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode -> data = value;
 yes (head == NULL)
 {
 head = newNode;
 newNode -> next = head;
 }
 the rest
 {
 struct Node * temp = head;
 while (temp -> next! = head)
 temp = temp -> next;
 temp -> next = newNode;
 newNode -> next = head;
 }
 printf ("\ nThe entry was successful!");
}
void insertAfter (int value, int location)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode -> data = value;
 yes (head == NULL)
 {
 head = newNode;
 newNode -> next = head;
 }
 the rest
 {
 struct Node * temp = head;
 while (temp -> data! = location)
 {
 yes (temp -> next == head)
 {
 printf ("The indicated node is not in the list !!!");
 go to EndFunction;
 }
 the rest
 {
 temp = temp -> next;
 }
 }
 newNode -> next = temp -> next;
 temp -> next = newNode;
 printf ("\ nThe entry was successful!");
 }
 Final function:
}
void deleteBeginning ()
{
 yes (head == NULL)
 printf ("The list is empty! Could not delete it!");
 the rest
 {
 struct Node * temp = head;
 yes (temp -> next == head)
 {
 head = NULL;
 free (temperature);
 }
 the rest{
 head = head -> forward;
 free (temperature);
 }
 printf ("\ n Deletion successful !!!");
 }
}
void deleteEnd ()
{
 yes (head == NULL)
 printf ("The list is empty! Could not delete it!");
 the rest
 {
 struct Node * temp1 = head, temp2;
 yes (temp1 -> next == head)
 {
 head = NULL;
 free (temp1);
 }
 the rest{
 while (temp1 -> next! = head) {
 temp2 = temp1;
 temp1 = temp1 -> next;
 }
 temp2 -> next = head;
 free (temp1);
 }
 printf ("\ n Deletion successful !!!");
 }
}
```

```
void deleteSpecific (int delValue)
{
 yes (head == NULL)
 printf ("The list is empty! Could not delete it!");
 the rest
 {
 struct Node * temp1 = head, temp2;
 while (temp1 -> data! = delValue)
 {
 yes (temp1 -> next == head)
 {
 printf ("\ nThe indicated node is not in the list !!!");
 go to FineFunction;
 }
 the rest
 {
 temp2 = temp1;
 temp1 = temp1 -> next;
 }
 }
 if (temp1 -> next == head) {
 head = NULL;
 free (temp1);
 }
 the rest{
 yes (temp1 == head)
 {
 temp2 = head;
 while (temp2 -> next! = head)
 temp2 = temp2 -> next;
 head = head -> forward;
 temp2 -> next = head;
 free (temp1);
 }
 the rest
 {
 yes (temp1 -> next == head)
 {
 temp2 -> next = head;
 }
 the rest
 {
 temp2 -> next = temp1 -> next;
 }
 free (temp1);
 }
 }
 printf ("\ n Deletion successful !!!");
 }
 End function:
}
blank screen ()
{
 yes (head == NULL)
 printf ("\ nList is empty !!!");
 the rest
 {
 struct Node * temp = head;
 printf ("\ nThe elements of the list are: \ n");
 while (temp -> next! = head)
```

```
{
printf ("% d --->", temp -> data);
}
printf ("% d --->% d", temp -> data, test -> data);
}}
```

**Production**



```
********** MENU *************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Where you want to insert:
1. At Beginning
2. At End
3. Between
Enter your choice: 1

One node inserted!!!

********** MENU *************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice:
```

**1) Search the individually linked circular list**

Searching in an individually linked circular list must traverse the list. The element to find in the list matches the data for each node in the list once, and if a match is found, the position of that element is returned, otherwise -1 is returned.

**Algorithm:**

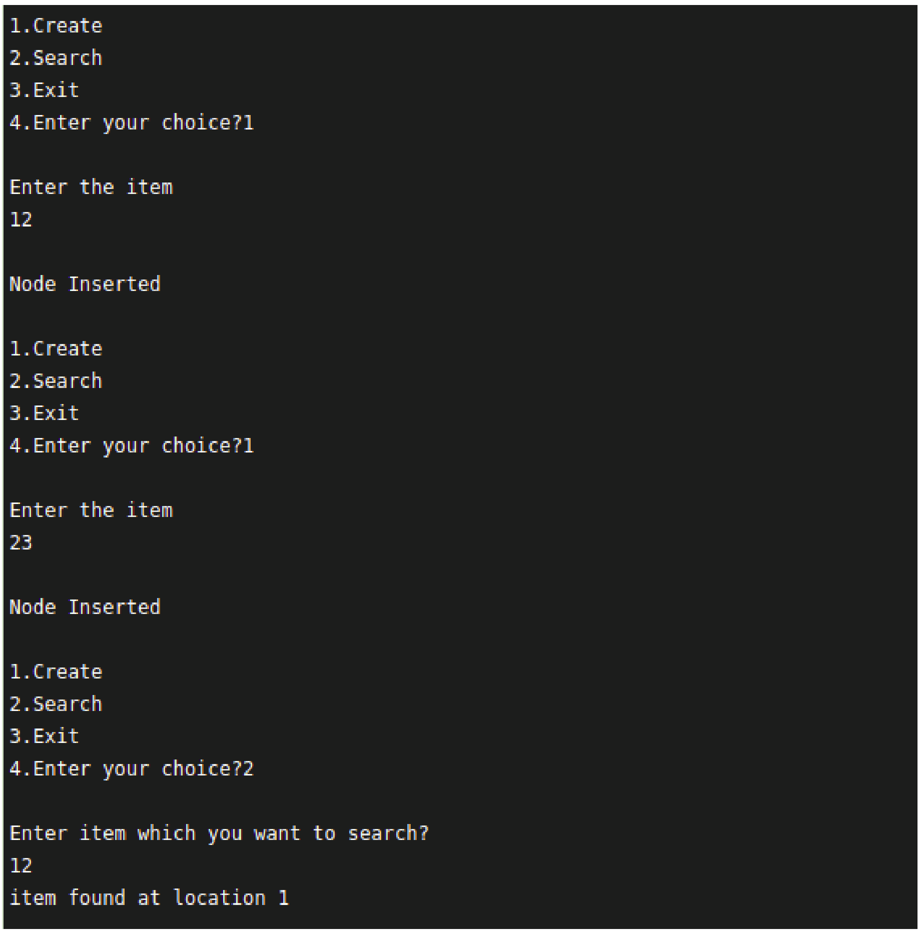
- Step 1: SET PTR = HEAD
- Step 2: Set I = 0
- STEP 3: IF PTR = NULL
- WRITE "EMPTY LIST" GO TO STEP 8 END YES PHASE 4: IF HEAD → DATA = ARTICLE
- WRITE i + 1 RETURN [END OF S] STEP 5: REPEAT STEPS 5 TO 7 UNTIL PTR-> next! = Head
- STEP 6: if ptr → date = article
- write i + 1 RETURN of IF STEP 7: I = I + 1
- STEP 8: PTR = PTR → NEXT
- [END OF LOOP] STEP 9: EXIT

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
create void (int);
empty search ();
structure node
{
 int data;
 next structure node *;
};
knot structure * head;
main vacuum ()
{
 int choice, object, loc;
 do
 {
 printf ("\ n1.Create \ n2.Search \ n3.Sexit \ n4.Enter your choice?");
 scanf ("% d", & option);
 change (choice)
 {
 Case 1:
 printf ("\ nInsert element \ n");
 scanf ("% d", & element);
 create (element);
 break time;
 case 2:
 Research();
 case 3:
 output (0);
 break time;
 default:
 printf ("\ nPlease enter a valid option \ n");
 }
 } while (choice! = 3);
}
create empty (int element)
{
 structure node * ptr = (structure node *) malloc (size of (structure node));
 struct node * temp;
 yes (ptr == NULL)
 {
 printf ("\ nOVERFLOW \ n");
 }
 the rest
 {
 ptr-> data = element;
 yes (head == NULL)
 {
 head = ptr;
 ptr -> next = head;
 }
 the rest
 {
 temperature = head;
 while (temp -> next! = head)
 {
 temp = temp -> next;
 }
 temp -> next = ptr;
 ptr -> next = head;
 }
 printf ("\ nNode inserted \ n");
 }}
stop searching ()
```

```
{
struct node * ptr;
int element, i = 0, flag = 1;
ptr = head;
yes (ptr == NULL)
{
printf ("\ nEmpty list \ n");
}
the rest
{
printf ("\ nPlease enter the element you want to search for? \ n");
scanf ("% d", & element);
if (head -> data == element)
{
printf ("article found at location% d", i + 1);
flag = 0;
come back;
}
the rest
{
while (ptr-> next! = head)
{
if (ptr-> data == element)
{
printf ("article found at location% d", i + 1);
flag = 0;
come back;
}
the rest
{
flag = 1;
}
i ++;
ptr = ptr -> next;
}}
yes (flag! = 0)
{
printf ("Item not found \ n");
come back;
}}}
```

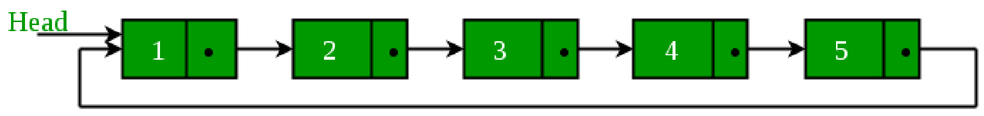
**Production:**



```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
12

Node Inserted

1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node Inserted

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
12
item found at location 1
```

## 2) Count the nodes in a circular linked list

Given a circular linked list, count the number of nodes it contains. For example, the output is 5 for the following list.



**Program:**

```
#include <bit / stdc ++. h>
using the std namespace;
 / * structure for a node * /
node structure {
 int data;
 Next node *;
 Node (int x)
 {
 data = x;
 next = NULL;
 }};
/ * Function to insert a node at the beginning
of a circular linked list * /
struct Node * push (struct Node * last, int data)
{
 if (last == NULL) {
 struct Node * temp
 = (struct Node *) malloc (sizeof (struct Node));
 // Assign the data.
 temp-> data = data;
 last = temperature;
// Note: The list was empty. We connect a single node
 // Furthermore.
 temp-> next = last;
 come back last;
 }
 // Dynamic creation of a node.
 struct Node * temp
 = (struct Node *) malloc (sizeof (struct Node));
 // Assign the data.
 temp-> data = data;
 // Adjust the links.
 temp-> next = last-> next;
 last-> next = temp;
 come back last;
}

/ * Function to count the nodes in a given Circular
linked list * /
int countNodes (node * head)
{
 Node * temp = head;
 int result = 0;
 if (head! = NULL) {
 do {
 temp = temp-> next;
 result ++;
 } while (temp! = head);
 }
 return the result;
```

```
}
 / * Controller program to test the above functions * /
main integer ()
{
 / * Initialize lists as empty * /
 Node * head = NULL;
 head = thrust (head, 12);
 head = thrust (head, 56);
 head = thrust (head, 2);
 head = thrust (head, 11);
 cout << countNodes (head);
 returns 0;
}
```

**Production:**

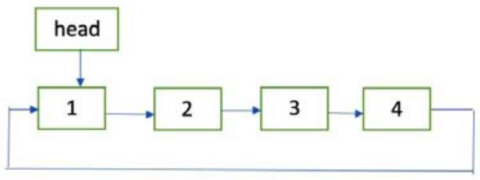
**4**

**3) Reverse a circular linked list**

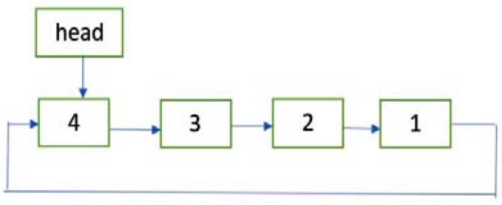
Given a linked circular of size n. The problem is to reverse the given circular linked list by changing the links between nodes.

**Examples:**

**ENTRANCE:**



**PRODUCTION:**



**Program:**

```
#include <bit / stdc ++. h>
 using the std namespace;

// Node of the linked list
node structure {
 int data;
 Next node *;
};

// function to get a new node
Node * getNode (int data)
{
 // allocate memory for the node
 Node * newNode = new Node;

 // put the data
 newNode-> data = data;
 newNode-> next = NULL;
 returns newNode;
}

// Function to invert the circular linked list
reverse void (node ** head_ref)
{
 // if the list is empty
 yes (* head_ref == NULL)
 come back;
 // inverse procedure equal to inverse a
 // list linked individually
 Node * prev = NULL;
 Current * node = * head_ref;
 Next node *;
 do {
 next = current-> next;
 current-> next = previous;
 previous = current;
 current = next;
 } while (current! = (* head_ref));

 // adjusting the bindings so that the
 // the last node points to the first node
 (* head_ref) -> next = previous;
 * head_ref = previous;
}

// Function to print a circular linked list
empty printList (Node * head)
{
 yes (head == NULL)
 come back;

 Node * temp = head;
 do {
 cout << temp-> data << "";
 temp = temp-> next;
 } while (temp! = head);
}

// Controller program to test above
```

```
main integer ()
{
 // Create a circular linked list
 // 1-> 2-> 3-> 4-> 1
 Node * head = getNode (1);
 head-> next = getNode (2);
 head-> next-> next = getNode (3);
 head-> next-> next-> next = getNode (4);
 head-> next-> next-> next-> next = head;
 cout << "Given list of linked circulars:";
 printList (head);
 reverse (and head);
 cout << "\ nInverted circular linked list:";
 printList (head);
 returns 0;
}
```

**Production:**

```
Given the linked circular list: 1 2 3 4
Inverted circular linked list: 4 3 2 1
```

**Question:**

1. What differentiates a circular linked list from a normal linked list?

a) You cannot have the ‘next’ pointer point to null in a circular linked list

b) It is faster to traverse the circular linked list

c) You may or may not have the ‘next’ pointer point to null in a circular linked list

d) Head node is known in circular linked list

2. Which of the following application makes use of a circular linked list?

a) Undo operation in a text editor

b) Recursive function calls

c) Allocating CPU to resources

d) Implement Hash Tables

3. Which of the following is false about a circular linked list?

a) Every node has a successor

b) Time complexity of inserting a new node at the head of the list is O(1)

c) Time complexity for deleting the last node is O(n)

d) We can traverse the whole circular linked list by starting from any point

4. Consider a small circular linked list. How to detect the presence of cycles in this list effectively?

a) Keep one node as head and traverse another temp node till the end to check if its 'next points to head

b) Have fast and slow pointers with the fast pointer advancing two nodes at a time and slow pointer advancing by one node at a time

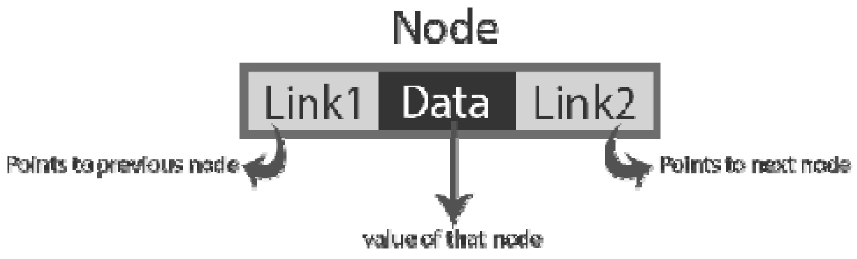
c) Cannot determine, you have to pre-define if the list contains cycles

d) Circular linked list itself represents a cycle. So no new cycles cannot be generated

**3. Objective:** To implement the doubly linked list using C / C ++

**Theory:**

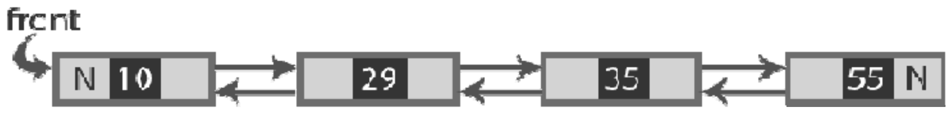
- In a single linked list, each node has a link to the next node in the sequence. So we can only cross from node to node in one direction and we cannot cross backwards. We can solve this type of problem by using a list of double bonds. A list of double bonds can be defined as follows ...
- The double-linked list is a sequence of items where each item has links to the previous item and the next item in the sequence.
- In a double-linked list, each node has a link to its previous node and to the next node. So, we can go forward using the next field and we can go back using the previous field.



**Program**

**Important points to remember**

- In a double-linked list, the first node must always point towards the head.
- The previous field of the first node must always be NULL.
- The next field of the last node must always be NULL.



**Doubly linked list operations**

- Insertion
- deletion
- Show

**Insertion**

In a list of double bonds, the insert operation can be performed in three ways as follows ...

- Insert at the beginning of the list
- Enter to the end of the list
- Insert in a specific position in the list

**Insert at the beginning of the list**

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

Step 1: Create a newNode with the specified value and newNode → above as NULL.

Step 2: Check if the list is empty (head == NULL)

Step 3: If empty, set NULL to newNode → next and newNode to header.

Step 4: If it's not empty, set head to newNode → next and newNode to head.

**Enter to the end of the list**

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

Step 1: Create a newNode with the specified value and newNode → next as NULL.

Step 2: Check if the list is empty (head == NULL)

Step 3: If empty, set NULL to newNode → previous and newNode to header.

Step 4: If it is not empty, define a node pointer temperature and initialize with head.

Step 5: Keep moving the temperature to the next node until it reaches the last node in the list (until temp → next equals NULL).

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

**Insert in a specific position in the list (after a node)**

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

Step 1: Create a new node with a certain value.

Step 2: Check if the list is empty (head == NULL)

Step 3: If empty, set NULL to newNode → previous and newNode → next and set newNode as header.

Step 4: If it is not empty, define two pointers to node temp1 and temp2 and initialize temp1 with head.

Step 5: keep moving temp1 to your next node until you reach the node after which we want to insert the newNode (as long as temp1 → data is equal to the position, here the position is the value of the node after which we want to insert the newNode ).

Step 6: Check every time that temp1 has been reached on the last node. If the last node is reached, 'The specified node is not in the list! Insertion not possible !!! 'and terminate the function. Otherwise, move temp1 to the 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 list of double bonds, the delete operation can be performed in three ways as follows ...

- Delete from the top of the list
- Remove from the end of the list
- Delete a specific node

**Delete from the top of the list**

We can use the following steps to remove a node from the beginning of the double bond list ...

Step 1: Check if the list is empty (head == NULL)

Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.

Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.

Step 4: Check if the list has only one node (temp → previous equal to temp → next)

Step 5: If TRUE, set the head to NULL and remove the temperature (setting the conditions of the empty list)

Step 6: If FALSE, assign temp → next to header, NULL to header → above and remove temp.

**Remove from the end of the list**

We can use the following steps to remove a knot from the end of the double bond list ...

Step 1: Check if the list is empty (head == NULL)

Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.

Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.

Step 4: Check if the list has only one node (temp → previous and temp → next are both NULL)

Step 5: If TRUE, assign NULL to the header and remove the temperature. And finish the show. (Setting the empty list condition)

Step 6: If FALSE, keep moving the temperature until it reaches the last node in the list. (until the next → temperature equal to NULL)

Step 7: NULL at temp → previous → next and remove temp.

**Remove a specific node from the list**

We can use the following steps to remove a specific node from the double bond list ...

Step 1: Check if the list is empty (head == NULL)

Step 2: If blank, display 'List is empty! Deletion is not possible and terminate the function.

Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.

Step 4: Keep moving the temperature until you reach the exact knot to remove or the last knot.

Step 5: If the last node is reached, display 'The specified node is not in the list! It cannot be canceled !!! 'and terminate the function.

Step 6: If the exact node we want to remove is reached, check if the list has only one node or not

Step 7: If the list has only one node and this is the node to remove, set the header to NULL and remove temp (free (temp)).

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

Step 9: If temp is the first node, move the head to the next node (head = head → next), set the head of the previous one to NULL (head → previous = NULL) and remove the temperature.

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

Step 11: If temp is the last node, set temp from previous or next to NULL (temp → previous → next = NULL) and remove temp (free (temp)).

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

**Visualization of a double linked list**

We can use the following steps to view the elements of a list of double bonds ...

Step 1: Check if the list is empty (head == NULL)

Step 2: If blank, show 'List is empty !!!' and terminate the function.

Step 3: If it is not empty, define a pointer to the 'temp' node and initialize with head.

Step 4: Show 'NULL <---'.

Step 5: Continue viewing the temperature → data with an arrow (<===>) until the temperature reaches the last node

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

**Program**

```
#include <stdio.h>

#include <conio.h>

void insertAtBeginning (int);

void insertAtEnd (int);

void insertAtAfter (int, int);

void deleteBeginning ();

void deleteEnd ();

void deleteSpecific (int);

blank display ();

Structure node

{

 int data;

 struct Node * previous, * next;

} * head = NULL;

main vacuum ()

{

 int choice1, choice2, value, position;

 clrscr ();

 while (1)

 {

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

 printf ("1. Enter \ n2. Delete \ n3. Screen \ n4. Exit \ nEnter your
choice:");

 scanf ("% d", & choice1);

 Change ()
 {
 case 1: printf ("Insert the value to insert:");
 scanf ("% d", & value);
 while (1)
 {
printf ("\ nSelect from the following insert options \ n");
printf ("1. At the beginning \ n2. At the end \ n3. After a node \ n4. Cancel
\ nEnter your choice:");
 scanf ("% d", & choice2);
 switch (option 2)
 {
 case 1: insertAtBeginning (value);
 break time;
 case 2: insertAtEnd (value);
 break time;
 case 3: printf ("Enter the position after which you want to insert:");
 scanf ("% d", & location);
 insertAfter (value, position);
 break time;
 case 4: go to EndSwitch;
 default: printf ("\ nSelect the correct insert option! \ n");
 }
 }
 case 2: while (1)
 {
printf ("\ nSelect from the following delete options \ n");
printf ("1. At the beginning \ n2. At the end \ n3. Specific node \ n4.
Cancel \ nEnter your choice:");
 scanf ("% d", & choice2);
 switch (option 2)
 {
```

```
case 1: deleteBeginning ();
break time;
case 2: deleteEnd ();
break time;
case 3: printf ("Enter the value of the node to be removed:");
scanf ("% d", & location);
deleteSpecic (location);
break time;
case 4: go to EndSwitch;
default: printf ("\ nSelect the correct delete option! \ n");
}
}
Limit switch: pause;
case 3: display ();
break time;
case 4: exit (0);
default: printf ("\ nSelect the correct option!");
}}}
void insertAtBeginning (int value)
{
struct Node * newNode;
newNode = (struct Node *) malloc (sizeof (struct Node));
newNode -> data = value;
newNode -> previous = NULL;
yes (head == NULL)
{
newNode -> next = NULL;
head = newNode;
}
the rest
```

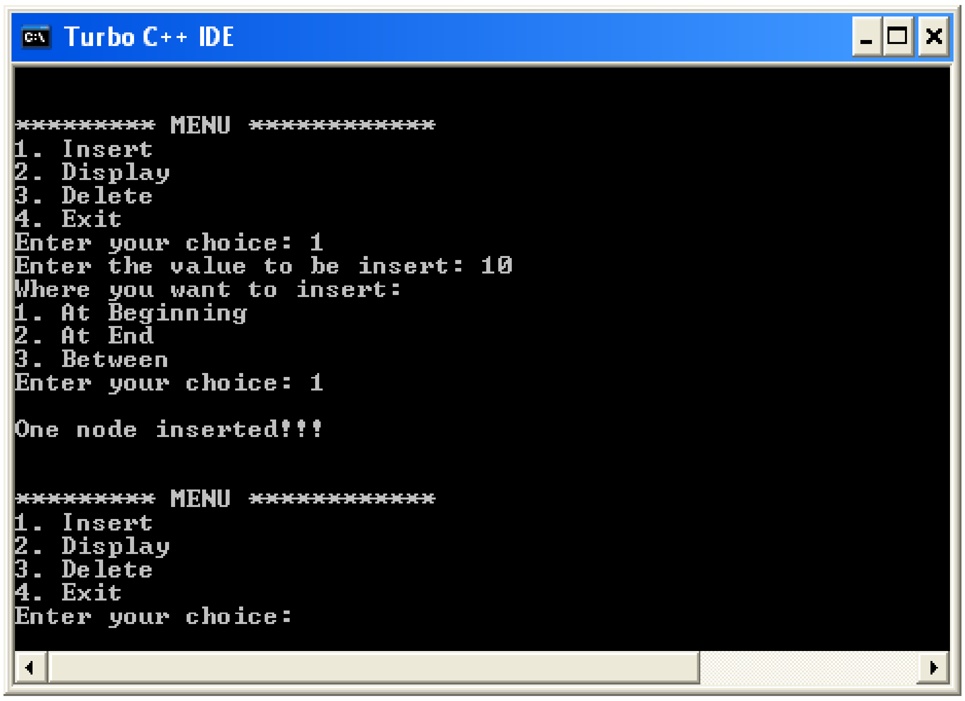
```
 {
 newNode -> next = head;
 head = newNode;
 }
 printf ("\ nThe entry was successful!");
}
void insertAtEnd (int value)
{
 struct Node * newNode;
 newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode -> data = value;
 newNode -> next = NULL;
 yes (head == NULL)
 {
 newNode -> previous = NULL;
 head = newNode;
 }
 the rest
 {
 struct Node * temp = head;
 while (temp -> next! = NULL)
 temp = temp -> next;
 temp -> next = newNode;
 newNode -> previous = temp;
 }
 printf ("\ nThe entry was successful!");
}
void insertAfter (int value, int location)
{
 struct Node * newNode;
```

```
newNode = (struct Node *) malloc (sizeof (struct Node));
newNode -> data = value;
yes (head == NULL)
{
newNode -> previous = newNode -> next = NULL;
head = newNode;
}
the rest
{
struct Node * temp1 = head, temp2;
while (temp1 -> data! = position)
{
yes (temp1 -> next == NULL)
{
printf ("The indicated node is not in the list !!!");
go to EndFunction;
}
the rest
{
temp1 = temp1 -> next;
}}
temp2 = temp1 -> next;
temp1 -> next = newNode;
newNode -> previous = temp1;
newNode -> next = temp2;
temp2 -> previous = newNode;
printf ("\ nThe entry was successful!");
}
Final function:
}
```

```
void deleteBeginning ()
{
 yes (head == NULL)
 printf ("The list is empty! Could not delete it!");
 the rest
 {
 struct Node * temp = head;
 yes (temp -> previous == temp -> next)
 {
 head = NULL;
 free (temperature);
 }
 the rest{
 head = temp -> next;
 head -> front = NULL;
 free (temperature);
 }
 printf ("\ n Deletion successful !!!");
 }}
void deleteEnd ()
{
 yes (head == NULL)
 printf ("The list is empty! Could not delete it!");
 the rest
 {
 struct Node * temp = head;
 yes (temp -> previous == temp -> next)
 {
 head = NULL;
 free (temperature);
```

```
}
the rest{
while (temp -> next! = NULL)
temp = temp -> next;
temp -> previous -> next = NULL;
free (temperature);
}
printf ("\ n Deletion successful !!!");
}}
void deleteSpecific (int delValue)
{
yes (head == NULL)
printf ("The list is empty! Could not delete it!");
the rest
{
struct Node * temp = head;
while (temp -> data! = delValue)
{
yes (temp -> next == NULL)
{
printf ("\ nThe indicated node is not in the list !!!");
go to FineFunction;
}
the rest
{
temp = temp -> next;
}
}
yes (temp == head)
{
head = NULL;
free (temperature);
}
the rest
{
temp -> previous -> next = temp -> next;
free (temperature);
}
printf ("\ n Deletion successful !!!");
}
End function:
}
blank screen ()
{
yes (head == NULL)
printf ("\ nList is empty !!!");
the rest
{
struct Node * temp = head;
printf ("\ nThe elements of the list are: \ n");
printf ("NULL <---");
while (temp -> next! = NULL)
{
printf ("% d <===>", temp -> data);
}
printf ("% d ---> NULL", temp -> data);
}}
```

**Production**



```
********** MENU *************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Where you want to insert:
1. At Beginning
2. At End
3. Between
Enter your choice: 1

One node inserted!!!

********** MENU *************
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice:
```

## 1) Count the nodes in the doubly linked list

```
#include <iostream>
using the std namespace;
// structure of the node
node structure {
 int data;
 Next node *;
 Previous node *;
};
class LinkedList {
 private:
 Knot * head;
 public:
 Linked List () {
 head = NULL;
 }
 // Add a new item to the end of the list
```

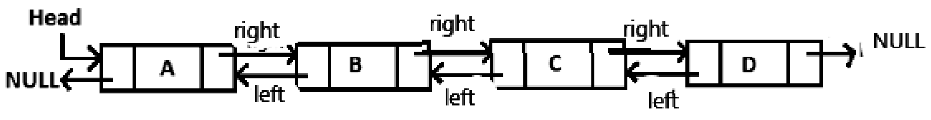
```
void push_back (int newElement) {
Node * newNode = new Node ();
newNode-> data = newElement;
newNode-> next = NULL;
newNode-> prev = NULL;
if (head == NULL) {
head = newNode;
} the rest {
Node * temp = head;
while (temp-> next! = NULL)
temp = temp-> next;
temp-> next = newNode;
newNode-> prev = temp;
}
}
// count the nodes in the list
int countNodes () {
Node * temp = head;
int i = 0;
while (temp! = NULL) {
i ++;
temp = temp-> next;
}
I return;
}
// show the contents of the list
void PrintList () {
Node * temp = head;
if (temp! = NULL) {
cout << "The list contains:";
```

```
 while (temp! = NULL) {
 cout <<temp-> data << "";
 temp = temp-> next;
 }
 cout << endl;
 } the rest {
 cout << "The list is empty. \ n";
 }}};
// test the code
int main () {
 LinkedList MyList;
 // Add four items to the list.
 MyList.push_back (10);
 MyList.push_back (20);
 MyList.push_back (30);
 MyList.push_back (40);
 // Show the contents of the list.
 MyList.PrintList ();
 // number of nodes in the list
 cout << "No. of nodes:" << MyList.countNodes ();
 returns 0;
}
}
```

**Production:**

```
The list contains: 10 20 30 40
Number of nodes: 4
```

## 2) Find an item in a doubly linked list



Given a doubly linked list (DLL) containing N nodes and an integer X, the task is to find the position of the entire X in the doubly linked list. If no such position is found, print -1.

**Examples:**
Input: 15 <=> 16 <=> 8 <=> 7 <=> 13, X = 8
**Production:** 3
**Explanation:** X (= 8) is present at the third node of the doubly linked list.
Therefore, the required output is 3
Input: 5 <=> 3 <=> 4 <=> 2 <=> 9, X = 0
Exit: -1
**Explanation:** X (= 0) is not present in the doubly linked list.
Therefore, the required output is -1
**Approach:** Follow the steps below to fix the problem:

- Initialize a variable, say pos, to store the position of the node that contains the X data value in the doubly linked list.
- Initializes a pointer, such as temp, to store the parent node of the doubly linked list.
- Iterate over the linked list and for each node check if the data value of that node equals X or not. If determined to be true, print pos.
- Otherwise, press -1.

**Program:**

```
#include <bit / stdc ++. h>

using the std namespace;


// Structure of a node

// the doubly linked list

node structure {


 // Store the data value

 // of a node

 int data;
 // Store the pointer
 // to the next node
 Next node *;

 // Store the pointer
 // to the previous node
 Previous node *;
};

// Function to insert a node into the
// start of the doubly linked list
push empty (node ** head_ref, int new_data)
{

 // Allocate memory for a new node
 Node * new_node
 = (Node *) malloc (size of (Node structure));

 // Enter the data
 new_node-> data = new_data;

 // As the node is added to the
 // start, prev is always NULL
 new_node-> prev = NULL;

 // Connect the previous list to the new node
 new_node-> next = (* head_ref);

 // If the head pointer is not NULL
 if ((* head_ref)! = NULL) {
```

```
 // Change the previous header
 // from node to new node
 (* head_ref) -> prev = new_node;
 }

 // Move your head to point to the new node
 (* head_ref) = new_node;
}

// Function to find the position of
// an integer in a doubly linked list
search int (node ** head_ref, int x)
{

 // Store the root node
 Temp node = head_ref;

 // Stores the position of the integer
 // in the doubly linked list
 int position = 0;

 // Loop through the doubly linked list
 while (temp-> data! = x
&& temp-> forward! = NULL) {

 // Update position
 position ++;
 // Update temperature
 temp = temp-> next;
 }

 // If the integer is not present
 // in the doubly linked list
 if (temp-> data! = x)
 return -1;

 // If the integer present in
 // the doubly linked list
 return (pos + 1);
}

// Driver code
main integer ()
{
 Node * head = NULL;
 intX = 8;
 // Create the doubly linked list
 // 18 <-> 15 <-> 8 <-> 9 <-> 14
 push (and head, 14);
 push (and head, 9);
 push (and head, 8);
 push (and head, 15);
 push (and head, 18);
 cout << search (& testa, X);
 returns 0;
}
```

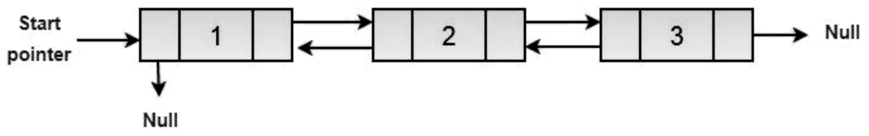
**Production:** 3

**1) Reverse a doubly linked list**

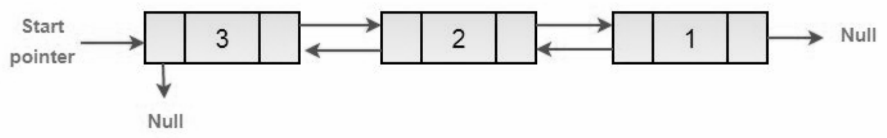
Given a doubly linked list, the task is to reverse the given doubly linked list.

See the following diagrams, for example.

**(a) Double linked original list**



**(b) Reverse doubly linked list**



Here is a simple method to reverse a doubly linked list. All we need to do is swap the previous and next pointers for all nodes, change the previous header (or start), and change the header pointer to the end.

**Program:**

```
#include <bit / stdc ++. h>
using the std namespace;

/ * a node from the doubly linked list * /
Node class
{
 public:
 int data;
 Next node *;
 Previous node *;
};

/ * Function to invert a doubly linked list * /
```

```
reverse void (node ** head_ref)
{
 Node * temp = NULL;
 Current * node = * head_ref;

 / * swaps next and previous for all nodes
 doubly linked list * /
 while (current! = NULL)
 {
 temp = current-> previous;
 current-> previous = current-> next;
 current-> next = temp;
 current = current-> previous;
 }

 / * Before changing the head, check that the speakers are empty
 list and list with a single node * /
 yes (temp! = NULL)
 * head_ref = temp-> previous;
}

/ * UTILITY FUNCTIONS * /
/ * Function to insert a node in the
start of list doubly linked * /
push empty (node ** head_ref, int new_data)
{
 / * assign node * /
 Node * new_node = new node ();

 / * I enter the data * /
 new_node-> data = new_data;
 / * since we are adding at the beginning,
 prev is always NULL * /
 new_node-> prev = NULL;

 / * associate the list above with the new node * /
 new_node-> next = (* head_ref);

 / * change the old main node to a new node * /
 if ((* head_ref)! = NULL)
 (* head_ref) -> prev = new_node;

 / * move head to point to new node * /
 (* head_ref) = new_node;
}

/ * Function to print the nodes in a given doubly linked list
This function is the same as printList () of an individually linked list * /
empty printList (node * node)
{
 while (node! = NULL)
 {
 cout << node-> data << "";
 node = node-> next;
 }
}
/ * Driver code * /
main integer ()
{
 / * Start with empty list * /
 Node * head = NULL;
```

```
/ * We create an ordered linked list to test the functions
 The created linked list will be 10-> 8-> 4-> 2 * /
 press (& testa, 2);
 push (and head, 4);
 push (and head, 8);
 push (and head, 10);

 cout << "Original linked list" << endl;
 printList (head);

 / * Double linked list inverse * /
 reverse (and head);

 cout << "\ nInverted linked list" << endl;
 printList (head);

 returns 0;
}
```

**Production:**

```
Original linked list
10 8 4 2
The inverted linked list is
2 4 8 10
```

**Questions:**

1. Which of the following is false about a doubly linked list?

    a) We can navigate in both the directions

    b) It requires more space than a singly linked list

    c) The insertion and deletion of a node take a bit longer

    d) Implementing a doubly linked list is easier than singly linked list

2. What is a memory efficient doubly linked list?

    a) Each node has only one pointer to traverse the list back and forth

    b) The list has breakpoints for faster traversal

    c) An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list

    d) A doubly linked list that uses bitwise AND operator for storing addresses

3. How do you calculate the pointer difference in a memory efficient double linked list?

    a) head xor tail

    b) pointer to previous node xor pointer to next node

    c) pointer to previous node – pointer to next node

    d) pointer to next node – pointer to previous node

4. the following doubly linked list: head-1-2-3-4-5-tail. What will be the list after performing the given sequence of operations?

```
Node temp = new Node(6,head,head.getNext());
head.setNext(temp);
temp.getNext().setPrev(temp);
Node temp1 = tail.getPrev();
tail.setPrev(temp1.getPrev());
temp1.getPrev().setNext(tail);
```

a) head-6-1-2-3-4-5-tail

b) head-6-1-2-3-4-tail

c) head-1-2-3-4-5-6-tail

d) head-1-2-3-4-5-tail

**4. Objective:** Add two polynomials using a linked list

**Theory:**

Given two polynomial numbers represented by a linked list. Writing a function that aggregates these lists means adding the coefficients that have the same variable powers.

**Example:**

**Entrance:**

1st number = 5x2 + 4x1 + 2x0

2nd number = -5x1 - 5x0

**Production:**

5x2-1x1-3x0

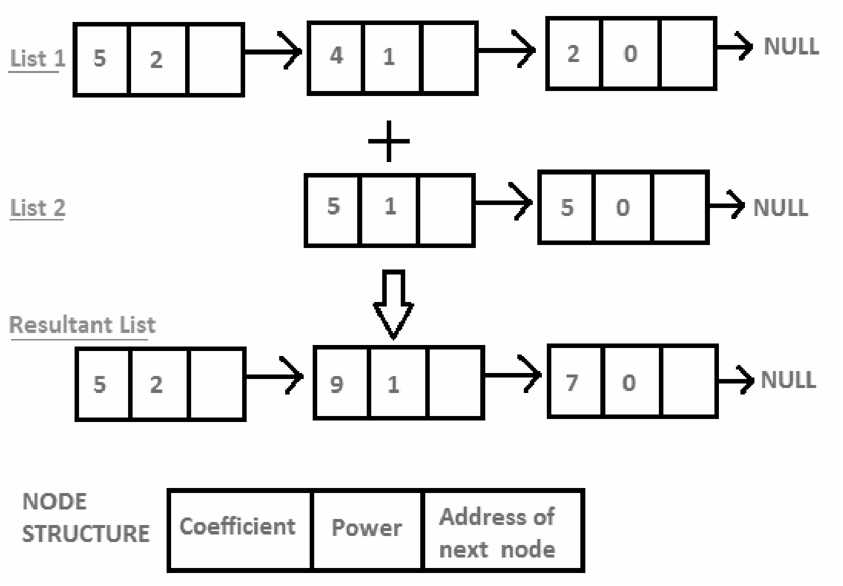
**Entrance:**

1st number = 5x3 + 4x2 + 2x0

2nd number = 5x ^ 1 - 5x ^ 0

**Production:**

5x3 + 4x2 + 5x1 - 3x0



**Program:**

```
// use linked lists
#include <bit / stdc ++. h>
using the std namespace;

// Structure of the node containing power and coefficient of
// variable
node structure {
 int coeff;
 internal power;
 struct Node * next;
};

// Function to create a new node
empty create_node (int x, int y, struct Node ** temp)
{
 Structure node * r, * z;
 z = * temperature;
 if (z == NULL) {
 r = (struct Node *) malloc (sizeof (struct Node));
 r-> coeff = x;
 r-> pow = y;
 * temperature = r;
 r-> next = (struct Node *) malloc (sizeof (struct Node));
 r = r-> next;
 r-> next = NULL;
 }
 the rest {
 r-> coeff = x;
 r-> pow = y;
 r-> next = (struct Node *) malloc (sizeof (struct Node));
```

```
r = r-> next;
r-> next = NULL;
}
}

// Sum function of two polynomial numbers
void polyadd (struct Node * poly1, struct Node * poly2,
 Structure node * poles)
{
 while (poly1-> next && poly2-> next) {
 // If the power of the first polynomial is greater than the second,
 // then store the first one as is and move its pointer
 if (poly1-> pow> poly2-> pow) {
 poli-> pow = poli1-> pow;
 poly-> coeff = poly1-> coeff;
 poly1 = poly1-> next;
 }

 // If the power of the second polynomial is greater than the first,
 // then store the second as is and move its pointer
 else if (poli1-> pow <poli2-> pow) {
 poli-> pow = poli2-> pow;
 poly-> coeff = poli2-> coeff;
 poly2 = poly2-> next;
 }

 // If the power of both polynomials is the same, then
 // add their coefficients
 the rest {
 poli-> pow = poli1-> pow;
 poly-> coeff = poly1-> coeff + poli2-> coeff;
 poly1 = poly1-> next;
 poly2 = poly2-> next;
 }

 // Dynamically create a new node
 poly-> next
 = (struct Node *) malloc (sizeof (struct Node));
 poli = poly-> next;
 poly-> next = NULL;
 }
 while (poly1-> next || poly2-> next) {
 if (poly1-> next) {
 poli-> pow = poli1-> pow;
 poly-> coeff = poly1-> coeff;
 poly1 = poly1-> next;
 }
 if (poly2-> next) {
 poli-> pow = poli2-> pow;
 poly-> coeff = poli2-> coeff;
 poly2 = poly2-> next;
 }
 poly-> next
 = (struct Node *) malloc (sizeof (struct Node));
 poli = poly-> next;
 poly-> next = NULL;
 }
}

// Show linked list
empty show (knot structure * knot)
{
 while (node-> next! = NULL) {
 printf ("% dx ^% d", node-> coeff, node-> pow);
 node = node-> next;
 if (node-> coeff> = 0) {
 if (node-> next! = NULL)
 printf ("+");
 }
 }
}

// Controller code
main integer ()
{
 struct Node * poly1 = NULL, * poly2 = NULL, * poly = NULL;

 // Create the first list of 5x ^ 2 + 4x ^ 1 + 2x ^ 0
 create_node (5, 2, & poly1);
 create_node (4, 1, & poly1);
 create_node (2, 0, & poly1);

 // Create a second list of -5x ^ 1 - 5x ^ 0
 create_node (-5, 1 and poly2);
 create_node (-5, 0 and poly2);

 printf ("1st number:");
 show (poly1);

 printf ("\ n2nd number:");
 show (poly2);
 poly = (struct Node *) malloc (sizeof (struct Node));

 // The function adds two polynomial numbers
 polyadd (poly1, poly2, poly);

 // Show the list of results
 printf ("\ nPolynomial added:");
 show (poles);

 returns 0;
}
```

**Production:**

```
1st number: 5x ^ 2 + 4x ^ 1 + 2x ^ 0
Second number: -5x ^ 1-5x ^ 0
Aggregate polynomial: 5x ^ 2-1x ^ 1-3x ^ 0
```

❖❖❖❖

# Module V

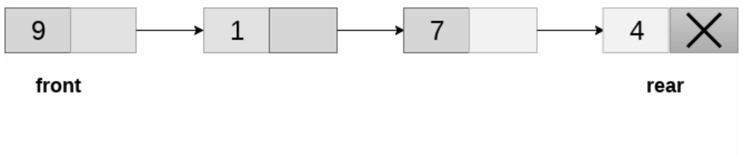
1. **Point:** Implementation of queued linked lists

**Objective:**

Use the queues for basic time simulations. Be able to recognize the properties of the problem where stacks, queues and deque are suitable data structures. Being able to implement the abstract data type list as a linked list using the reference node and model.

**Theory:**

- Due to the disadvantages discussed in the previous section of this tutorial, the matrix implementation cannot be used for large-scale applications where queues are implemented. One of the array implementation alternatives is the queue linked list implementation.
- The storage requirement of the linked representation of a queue with n items is o (n) while the time requirement for operations is o (1).
- In a linked queue, each node of the queue consists of two parts, namely the data part and the association part. Each item in the queue points to the next immediate item in memory.
- There are two pointers in memory in the linked queue: front pointer and back pointer. The front pointer contains the address of the starting element in the queue, while the back pointer contains the address of the last element in the queue.
- Insertion and removal are performed at the back and front respectively. If both the front and back are NULL, it indicates that the queue is empty.
- The linked representation of the queue is shown in the following figure.



Linked Queue

**Queued operation linked**

There are two basic operations that can be implemented on linked queues. The operations are Insert and Delete.

**Operation entry**

The insert operation adds the queue by adding an item to the end of the queue. The new item will be the last item in the queue.

First, allocate the memory for the new ptr node using the following declaration.

1. Ptr = (structure node *) malloc (size of (structure node));

There can be two scenarios of putting this new ptr node into the linked queue.

In the first scenario, we put an item in an empty queue. In this case, the front = NULL condition becomes true. Now the new element will be added as the only element in the queue and the next front and back pointer will point to NULL.

```
ptr -> data = element;
 yes (front == NULL)
 {
 front = ptr;
 posterior = ptr;
 front -> next = NULL;
 rear -> next = NULL;
 }
```

In the second case, the queue contains more than one element. The front = NULL condition becomes false. In this scenario, you need to update the final trailing pointer so that the next trailing pointer points to the new ptr node. Since this is a linked queue, we also need to make the back pointer point to the newly added ptr node. We also need to make the pointer to the next end point NULL.

```
posterior -> next = ptr;
 posterior = ptr;
 back-> next = NULL;
```

This puts the item in the queue. The algorithm and implementation of C are shown below.

**Algorithm**

Step 1: Allocate space for the new PTR node

Step 2: SET PTR -> DATA = VAL

Step 3: IF FRONT = NULLSET FRONT = REAR = PTRSET FRONT -> NEXT = REAR -> NEXT = NULLELSESET REAR -> NEXT = PTRSET REAR = PTRSET REAR -> NEXT = NULL [END OF IF]

**Step 4: FINISH**

**Program:**

```
insert empty (struct node * ptr, int element;)
{
 ptr = (structure node *) malloc (size of (structure node));
 yes (ptr == NULL)
 {
 printf ("\ nOVERFLOW \ n");
 come back;
 }
 the rest
 {
 ptr -> data = element;
 yes (front == NULL)
 {
 front = ptr;
 posterior = ptr;
 front -> next = NULL;
 rear -> next = NULL;
 }
 the rest
 {
 posterior -> next = ptr;
 posterior = ptr;
 back-> next = NULL;
 }}}
```

**deletion**

- The delete operation removes the item inserted first of all items in the queue. First of all, we need to check if the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case we simply write underflow in the console and exit.
- Otherwise, we will remove the element that the front pointer points to. To do this, copy the node pointed to by the front pointer to the ptr pointer. Now, move the front pointer, point to your next node and release the node pointed to by the ptr node. This is done using the following statements.

```
ptr = in front;
 front = front -> forward;
 free (ptr);
```

**Algorithm**

```
Step 1: IF FRONT = NULL Type "Underflow" Go to step 5 [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE RPP
Step 5: FINISH
```

**Program:**

```
remove void (structure node * ptr)
{
 yes (front == NULL)
 {
 printf ("\ nUNDERFLOW \ n");
 come back;
 }
 the rest
 {
```

```
 ptr = in front;
 front = front -> forward;
 free (ptr);
 }}
```

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
structure node
{
 int data;
 next structure node *;
};
front structure knot *;
knot back structure *;
blank insert ();
cancel cancel ();
blank display ();
main vacuum ()
{
 int choice;
 while (choice! = 4)
 {
 printf ("\ n ************************ Main menu
***************** ***********\North");
 printf ("\ n ==========================================
= =================== \ n ");
 printf ("\ n1.insert an item \ n2. Delete an item \ n3. Show queue \
n4.Sexit \ n");
 printf ("\ nPlease enter your choice?");
 scanf ("% d", & option);
 change (choice)
 {
 Case 1:
 to insert();
 break time;
 case 2:
 To remove();
 break time;
 case 3:
 show();
 break time;
 case 4:
 output (0);
 break time;
 default:
 printf ("\ nPlease enter a valid choice ?? \ n");
 }
 }
}
blank insert ()
{
 struct node * ptr;
 int element;
 ptr = (structure node *) malloc (size of (structure node));
 yes (ptr == NULL)
 {
 printf ("\ nOVERFLOW \ n");
 come back;
 }
 the rest
 {
```

```
 printf ("\ nPlease insert value? \ n");
 scanf ("% d", & element);
 ptr -> data = element;
 yes (front == NULL)
 {
 front = ptr;
 posterior = ptr;
 front -> next = NULL;
 rear -> next = NULL;
 }
 the rest
 {
 posterior -> next = ptr;
 posterior = ptr;
 back-> next = NULL;
 }
 }
}
delete delete ()
{
 struct node * ptr;
 yes (front == NULL)
 {
 printf ("\ nUNDERFLOW \ n");
 come back;
 }
 the rest
 {
 ptr = in front;
 front = front -> forward;
 free (ptr);
 }
}
blank screen ()
{
 struct node * ptr;
 ptr = in front;
 yes (front == NULL)
 {
 printf ("\ n Empty queue \ n");
 }
 the rest
 {printf ("\ nprint values ..... \ n");
 while (ptr! = NULL)
 {
 printf ("\ n% d \ n", ptr -> data);
 ptr = ptr -> next;
 }}}
```

**Production:**

```
***********Main menu**********
==============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 1
Enter value?
123
***********Main menu**********
==============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 1
Enter value?
90
***********Main menu**********
==============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 3
print values .....
123
90
***********Main menu**********
==============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 2
***********Main menu**********
==============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 3
print values .....
```

```
90
**********Main menu*********
=============================
1.Enter a subject
2.Remove an item
3. Show the queue
4.exit
Enter your choice? 4
```

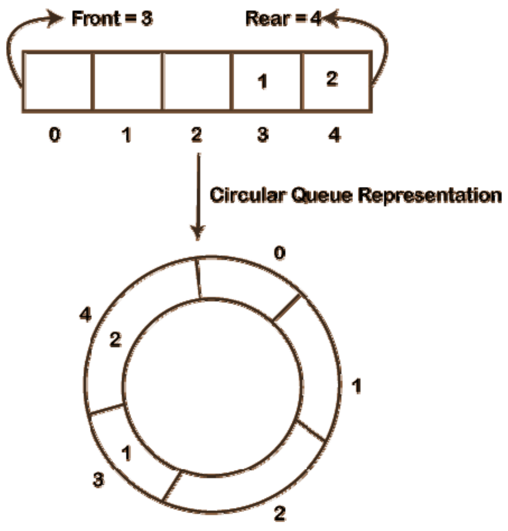
**2. Objective:** Implementation of the circular queue array

**Objective:**

The circular queue solves the main limitation of the normal queue. In a normal queue, after some insertion and deletion, there will be blank, unusable space.

**Theory:**

There was a limitation in the implementation of the Queue. If the back reaches the final position of the queue, there may be gaps at the beginning that cannot be used. So, to overcome these limitations, the concept of a circular queue was introduced.



As we can see in the image above, the back is in the last position of the queue and the front is pointing somewhere instead of position 0. In the array above, there are only two elements and three other positions are empty. . The rear is in the last position of the tail; if we try to insert the element, it will show that there are no empty spaces in the queue. There is a solution to avoid such a waste of memory space by moving both elements to the left and adjusting the front and rear ends accordingly. This is not a practically good approach because changing all the elements will take time. The effective approach to avoiding memory waste is to use the data structure of the circular queue.

**What is a circular queue?**

A circular queue is similar to a linear queue in that it too is based on the FIFO (First in, first out) principle except that the last position is connected to the first position in a circular queue that forms a circle. Also known as Ring Buffer.

**Circular queued operations**

The operations that can be performed on a circular queue are as follows:

- Front: used to get the front element of the tail.
- Back - Used to retrieve the back item from the queue.
- enQueue (value): This function is used to insert the new value into the queue. The new element is always inserted from the back.
- deQueue (): This function removes an item from the queue. Deletion in a queue is always done from the front end.

**Circular glue applications**

The circular queue can be used in the following scenarios:

- Memory Management: The circular queue provides memory management. As we have already seen in linear queues, memory is not handled very efficiently. But in the case of a circular queue, memory is managed efficiently by placing items in an unused location.
- CPU scheduling: The operating system also uses the circular queue to insert processes and then run them.
- Traffic System: In a computer controlled traffic system, semaphores are one of the best examples of a circular queue. Each traffic light turns on one by one after each time interval. As the red light is on for one minute, then the yellow light for one minute and then the green light. After the green light, the red light turns on.

**Queued operation**

The stages of the queuing operation are as follows:

- First, we will check if the queue is full or not.
- Initially, front and rear are set to -1. When we put the first element in a queue, both the front and the back are set to 0.
- When we insert a new element, the back is increased, i.e. back = back + 1.

**Scenarios for inserting an element**

There are two scenarios where the queue is not full:

- If rear! = Max - 1, the back will increase to mod (maxsize) and the new value will be inserted at the end of the queue.
- Yes in front! = 0 and back = max - 1, it means the queue is not full, so set the back value to 0 and put the new element there.

There are two cases in which the element cannot be inserted:

- When front == 0 && rear = max-1, it means the front is in the first position of the tail and the back is in the last position of the tail.
- front == rear + 1;

**Algorithm:**

```
Phase 1: S (REAR + 1)% MAX = FRONT
Write "OVERFLOW"
Go to step 4
[End OF S]
Step 2: IF FRONT = -1 and BACK = -1
SET FRONT = REAR = 0
OTHERWISE IF REAR = MAX - 1 and FRONT! = 0
REAR ADJUSTMENT = 0
THE REST
REAR ADJUSTMENT = (REAR + 1)% MAX
```

```
[END OF S]
Step 3: CONFIGURE QUEUE [REAR] = VAL
Step 4: EXIT
```

**Tail pull operation**

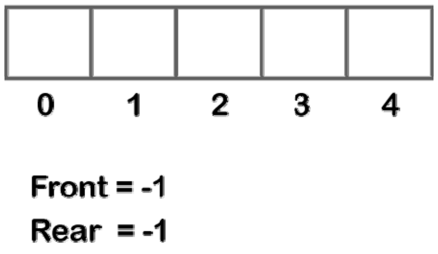
The steps of the remove the queue operation are as follows:

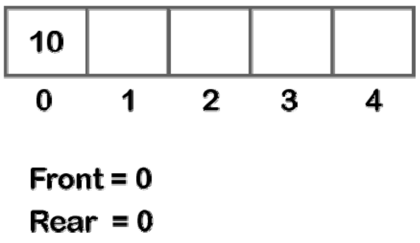
- First, let's check if the queue is empty or not. If the queue is empty, we cannot perform the queue cancel operation.
- When the element is removed, the front value is reduced by 1.
- If there is only one element left that needs to be removed, the front and back are reset to -1.

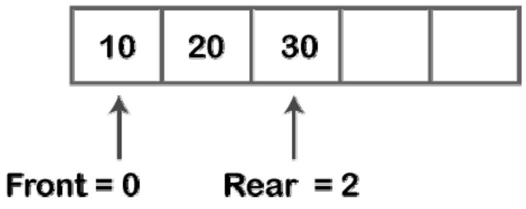
**Algorithm:**

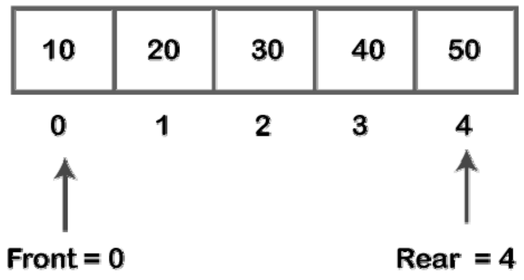
```
Step 1: IF FRONT = -1
Enter "UNDERFLOW"
Go to step 4
[END of S]
Step 2: SET VAL = QUEUE [FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
THE REST
IF FRONT = MAX -1
FRONT SET = 0
THE REST
SET FRONT = FRONT + 1
[END of S]
[END OF S]
Step 4: EXIT
```

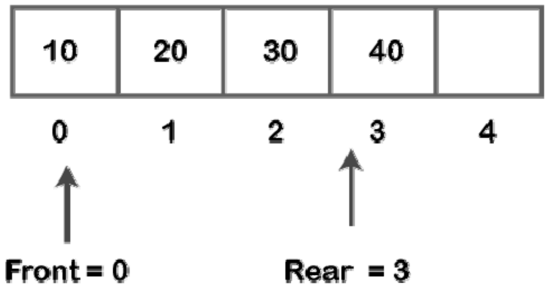
We understand the queuing and unblocking operation via the schematic representation.

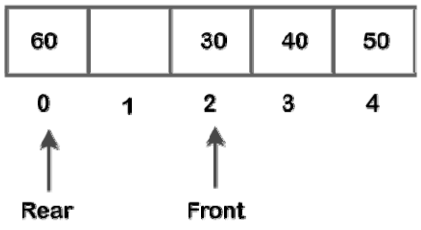

0
1
2
3
4
Front = -1
Rear = -1

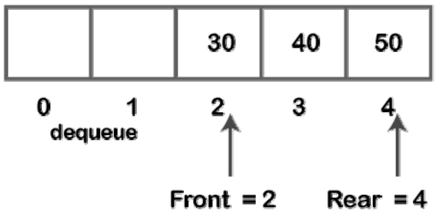

10
0
1
2
3
4
Front = 0
Rear = 0

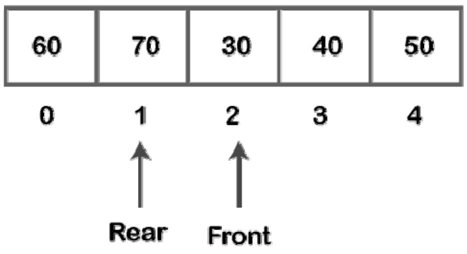

10
20
30
Front = 0
Rear = 2


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4









**Program:**

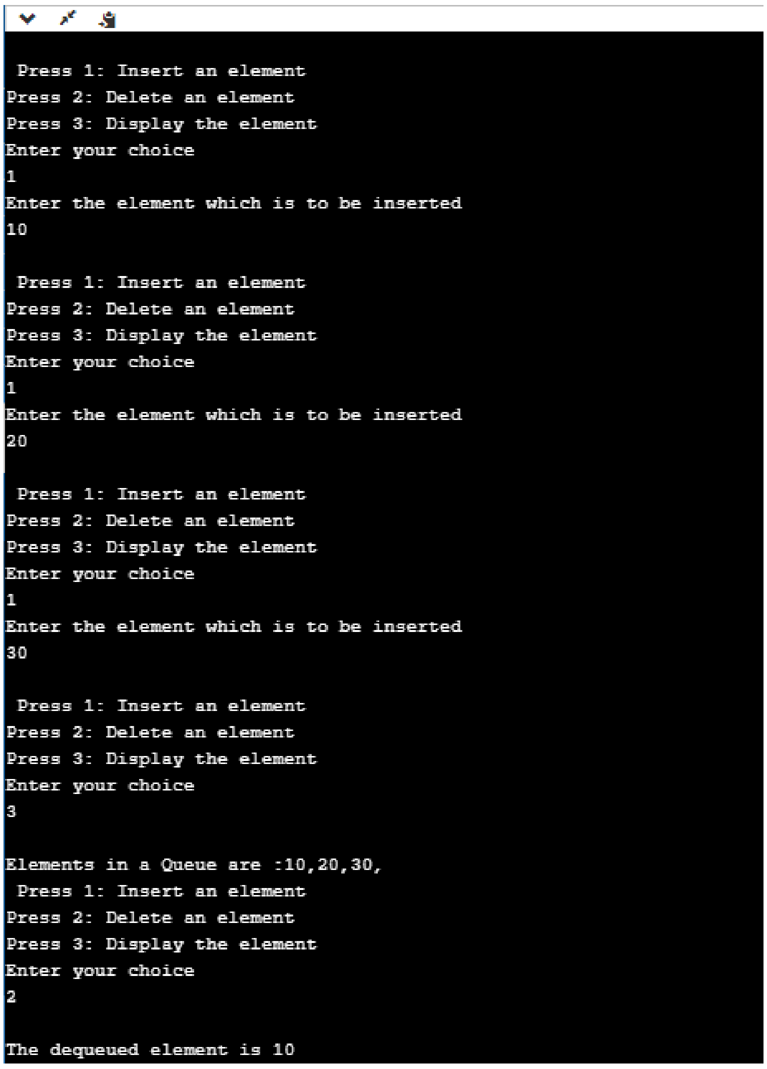
**Program:**

```
#include <stdio.h>

# define maximum 6
```

```
int queue [max]; // declaration of the array
front int = -1;
back int = -1;
// function to put an item in a circular queue
empty queuing (int element)
{
 if (front == - 1 && rear == - 1) // condition to verify that the queue is
empty
 {
 front = 0;
 rear = 0;
 tail [back] = element;
 }
 else if ((rear + 1)% max == front) // condition to check that the queue is
full
 {
 printf ("The queue is overflowing ..");
 }
 the rest
 {
rear = (rear + 1)% max; // the rear is increased
tail [back] = element; // assign a value to the rearmost queue.
 }}
// function to remove the item from the queue
int dequeue ()
{
 if ((front == - 1) && (rear == - 1)) // condition to verify that the queue is
empty
 {
 printf ("\ nQueue is underflow ..");
 }
 else if (front == rear)
{
 printf ("\ nThe removed item from the queue is% d", queue [front]);
 front = -1;
 rear = -1;
}
the rest
{
 printf ("\ nThe removed item from the queue is% d", queue [front]);
 front = (front + 1)% max;
}}
// function to display the elements of a queue
blank screen ()
{
 int i = front;
 yes (front == - 1 && rear == - 1)
  {
 printf ("\ n The queue is empty ..");
  }
 the rest
  {
 printf ("\ nThe elements of a queue are:");
 while (i <= behind)
  {
 printf ("% d,", tail [i]);
 i = (i + 1)% maximum;
  }}}
main integer ()
{
int choice = 1, x; // declaration of variables
```

```
while (option <4 && option! = 0) // while loop
{
printf ("\ n Press 1: Insert an element");
printf ("\ nPress 2: Delete an item");
printf ("\ nPress 3: Show item");
printf ("\ nPlease enter your choice");
scanf ("% d", & option);

change (choice)
{
Case 1:
printf ("Insert the element to insert");
scanf ("% d", & x);
tail (x);
break time;
case 2:
dequeue ();
break time;
case 3:
show();
}}
returns 0;
}
```

**Production:**



```
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

**Questions:**

1. Which of the following properties is associated with a queue?

a) First In Last Out

b) First In First Out

c) Last In First Out

d) Last In Last Out

2. In a circular queue, how do you increment the rear end of the queue?

a) rear++

b) (rear+1) % CAPACITY

c) (rear % CAPACITY)+1

d) rear–

3. What is the term for inserting into a full queue known as?

a) overflow

b) underflow

c) null pointer exception

d) program won't be compiled

4. What is the need for a circular queue?

a) effective usage of memory

b) easier computations

c) to delete elements based on priority

d) implement LIFO principle in queues

**3. Objective:** Priority queue using the linked list in C

**Objective:**

The priority queue (also known as a stripe) is used to keep track of unexplored paths, where a lower bound on the total path length is lower and has the highest priority.

**Theory:**

The queue is a FIFO data structure where the element that is inserted first is the first to be removed. A priority queue is a type of queue in which items can be inserted or removed based on priority. It can be implemented using a linked queue, stack, or list data structure. The priority queue is implemented following these rules:

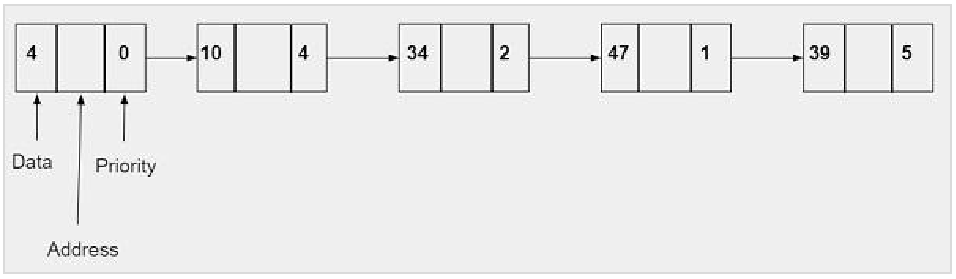
- The data or items with the highest priority will run before the data or items with the lowest priority.
- If two items have the same priority, they will be executed in the sequence in which they are added to the list.

A node in a linked list to implement the priority queue will contain three parts:

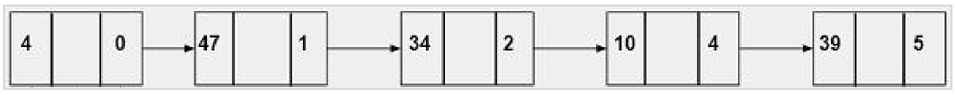
- Data: will store the integer value.
- Address: will store the address of a subsequent node
- Priority: will store the priority, which is an integer value. It can range from 0 to 10, where 0 represents the highest priority and 10 represents the lowest priority.

**Example**

**Entrance**



**Production**



**Algorithm:**

```
Start
Step 1-> Declare a tree node
 Declare data, priority
 Declare a structure node * below
Step 2-> In the Node function * newNode (int d, int p)
 Set Node * temp = (Node *) malloc (size of (Node))
 Set temp-> data = d
 Set temp-> priority = p
 Set temperature-> next = NULL
 Return temperature
Step 3-> In the int peek function (Node ** head)
 return (* head) -> data
Step 4-> In the pop void function (Node ** head)
 Set node * temp = * head
 Set (* head) = (* head) -> next
 free (temporary)
Step 5-> In the push function (Node ** head, int d, int p)
 Set node * start = (* head)
 Set Node * temp = new Node (d, p)
 If (* head) -> priority> p then,
 Set temperature-> next = * head
 Set (* head) = temp
 The rest
Loop While start-> next! = NULL && start-> next-> priority <p
 Set start = start-> next
 Set temperature-> next = start-> next
 Set start-> next = temperature
Step 6-> In the int isEmpty (Node ** head) function
 Return (* head) == NULL
Step 7-> In the int main () function
 Set node * pq = newNode (7, 1)
 Push call function (& pq, 1, 2)
 Push call function (& pq, 3, 3)
 Push function call (& pq, 2, 0)
 While loop (! IsEmpty (& pq))
 Print the results obtained by peek (& pq)
 Call function pop (& pq)
To stop
```

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
// priority node
typedef struct node {
 int data;
 int priority;
 next structure node *;
} Node;
Node * newNode (int d, int p) {
 Node * temp = (Node *) malloc (size of (Node));
 temp-> data = d;
 temp-> priority = p;
 temp-> next = NULL;
 return temperature;
}
int peek (knot ** head) {
 return (* header) -> data;
}
empty pop (knot ** head) {
 Node * temp = * head;
 (* head) = (* head) -> next;
 free (temperature);
}
push empty (node ** head, int d, int p) {
 Knot * start = (* head);
 Node * temp = newNode (d, p);
 if ((* head) -> priority> p) {
 temp-> next = * head;
 (* head) = temperature;
 } the rest {
 while (start-> next! = NULL &&
 start-> next-> priority <p) {
 start = start-> next;
 }
 // Or at the end of the list
 // or in the requested location
 temp-> next = start-> next;
 start-> next = temp;
 }
}
```

```
// Function to verify that the queue is empty
int isEmpty (node ** head) {
 return (* head) == NULL;
}
// main function
int main () {
 Node * pq = newNode (7, 1);
 press (& pq, 1, 2);
 press (& pq, 3, 3);
 press (& pq, 2, 0);
 while (! isEmpty (& pq)) {
 printf ("% d", see (& pq));
 pop (& pq);
 }
 returns 0;
}
```

**Production**

**2 7 1 3**

**Questions:**

1. With what data structure can a priority queue be implemented?

a) Array

b) List

c) Heap

d) Tree

2. Which of the following is not an application of priority queue?

a) Huffman codes

b) Interrupt handling in operating system

c) Undo operation in text editors

d) Bayesian spam filter

3. What is not a disadvantage of priority scheduling in operating systems?

a) A low priority process might have to wait indefinitely for the CPU

b) If the system crashes, the low priority systems may be lost permanently

c) Interrupt handling

d) Indefinite blocking

4. Which of the following is not an advantage of a priority queue?

a) Easy to implement

b) Processes with different priority can be efficiently handled

c) Applications with differing requirements

d) Easy to delete elements in any case

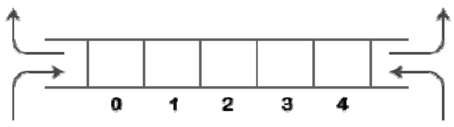
**4. Objective:** Implement double-ended queuing in C / C ++

**Objective:**

Double Ended Queue is also a queue data structure where insert and delete operations are performed on both ends (front and back). This means that we can insert both in the front and back position and we can remove in both the front and back position.

**Theory:**

Dequeue stands for Double Ended Tail. In the tail, the insertion takes place from one end while the removal from the other end. The end where the insertion takes place is known as the posterior end, while the end where the removal takes place is known as the anterior end.

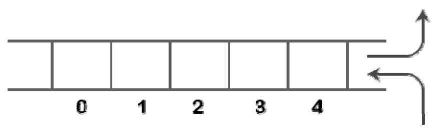


Deque is a linear data structure in which insert and delete operations are performed from both ends. We can say that deque is a generalized version of the coda.

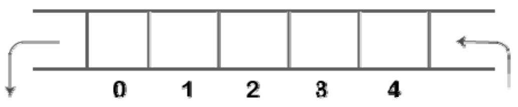
Let's take a look at some properties of deque.

- Deque can be used both as a stack and as a queue, as it allows insert and delete operations at both ends.

Then the insert and delete operation can be performed from the side. The stack follows the LIFO rule in which both insertion and removal can be done from one end; therefore, we conclude that deque can be considered as a stack.

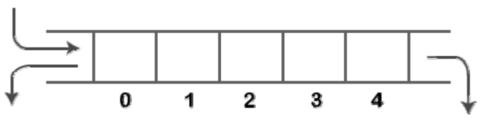


Then the insertion can be done from one end and the removal can be done from the other end. The queue follows the FIFO rule where the element is inserted on one side and removed on the other. Therefore, we conclude that the deque can also be regarded as the tail.

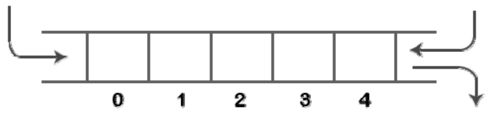


There are two types of queues, restricted entry queue and restricted exit queue.

1. Restricted Input Queue: Restricted Input Queue means that some restrictions are applied to the input. In the restricted input queue, insert is applied to one end while removal is applied to both ends.



2. Restricted Exit Queue: Restricted Exit Queue means that some restrictions apply to the delete operation. In a restricted exit queue, removal can only be applied from one end, while insertion is possible from both ends.



**Operations in Deque**

The operations applied in deque are as follows:

- Insert in the front
- Delete from the end
- insert in the back
- remove from the back

In addition to insertion and removal, we can also perform deque inspection operations. Through the peek operation, we can get the front and rear element of the tail.

We can perform two more operations on removing the queue:

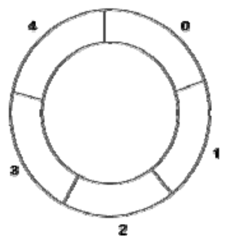
- isFull (): this function returns true if the stack is full; otherwise, it returns a false value.
- isEmpty (): this function returns true if the stack is empty; otherwise, it returns a false value.

**Memory representation**

The deque can be implemented using two data structures, namely a circular array and a doubly linked list. To implement the deque using a circular matrix, we first need to know what a circular matrix is.

**What is a circular matrix?**

An array is said to be circular if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4 and the array is full but the first position of the array is empty. If we want to insert the array element, it will not show any overflow condition since the last element is connected to the first element. The value we want to insert will be added to the first position of the array.



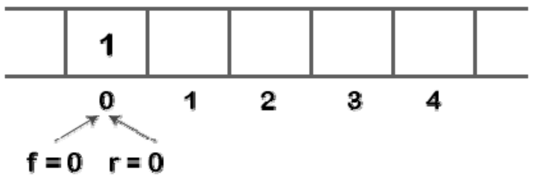
**Deque implementation using a circular array**

Below are the steps to perform the operations on the Deque:

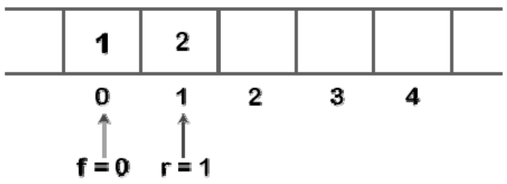
**Queued operation**

1. Initially, we are considering that the deque is empty, so both the front and the back are set to -1, i.e. f = -1 and r = -1.

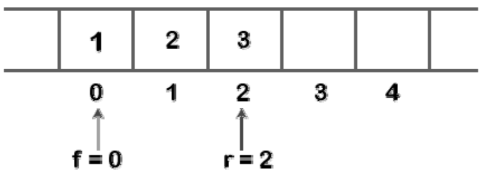
2. Since the deque is empty, inserting an element from the front or back would be the same. Suppose we entered element 1, so front is equal to 0 and back is also equal to 0.



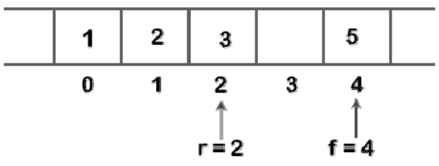
3. Suppose we want to insert the next element from behind. To insert the element from the back, we must first increase the back, that is, back = back + 1. Now, the back points to the second element and the front points to the first element.



4. Suppose we reinsert the element from the back. To insert the element, we will first increment the back and now the back points to the third element.

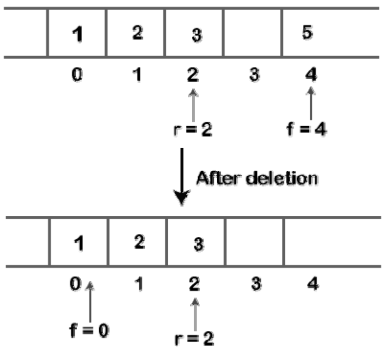


5. If we want to insert the element from the front and insert an element from the front, we need to decrease the value of the front by 1. If we decrease the front by 1, then the front points to position -1, which is not a position valid in an array. Then, we set the edge as (n -1), which is equal to 4 since n is 5. Once we set the edge, we will enter the value as shown in the following figure:

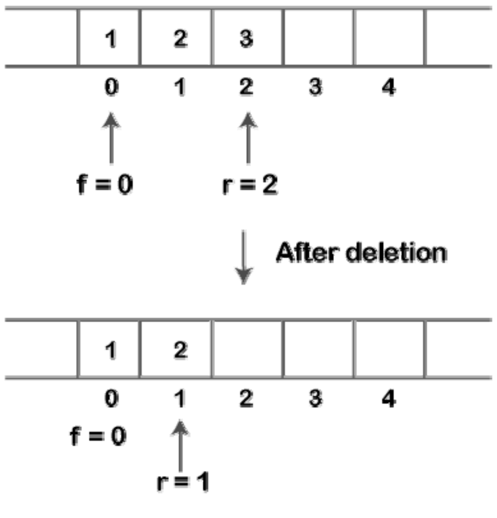


**Tail pull operation**

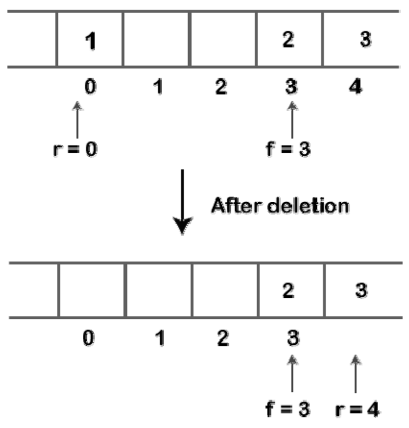
1. If the front points to the last element of the array and we want to perform the delete operation from the front. To remove any elements from the front, we must set front = front + 1. Currently the value of the front is equal to 4, and if we increase the value of the front, it becomes 5, which is not a valid index. So we conclude that if front points to the last element, then front is set to 0 in case of a delete operation.



2. If we want to remove the element from the back, we have to decrease the value of the back by 1, that is, back = back-1 as shown in the following figure:



3. If the back points to the first element and we want to remove the element from the back, then we need to set rear = n-1 where n is the size of the array as shown in the figure below:



Let's create a deque program.

The following are the six functions we used in the following program:

- enqueue_front (): is used to insert the element from the front-end.
- enqueue_rear (): used to insert the element from the back.
- dequeue_front (): used to remove the interface element.
- dequeue_rear (): is used to delete the element from the back.
- getfront (): used to return the front element of the deque.
- getrear (): used to return the element after the deque.

**Program:**

```
#defined size 5
#include <stdio.h>
int deque [size];
int f = -1, r = -1;
// the enqueue_front function will insert the value from the front
void enqueue_front (int x)
{
 if ((f == 0 && r == size-1) || (f == r + 1))
 {
 printf ("what is full");
 }
 plus if ((f == - 1) && (r == - 1))
 {
```

```
f = r = 0;
deque [f] = x;
}
yes no (f == 0)
{
f = dimension-1;
deque [f] = x;
}
the rest
{
f = f-1;
deque [f] = x;
}}
// the enqueue_rear function will insert the value from behind
void enqueue_rear (int x)
{
if ((f == 0 && r == size-1) || (f == r + 1))
{
printf ("what is full");
}
plus if ((f == - 1) && (r == - 1))
{
r = 0;
deque [r] = x;
}
plus if (r == size-1)
{
r = 0;
deque [r] = x;
}
the rest
{
r ++;
deque [r] = x;
```

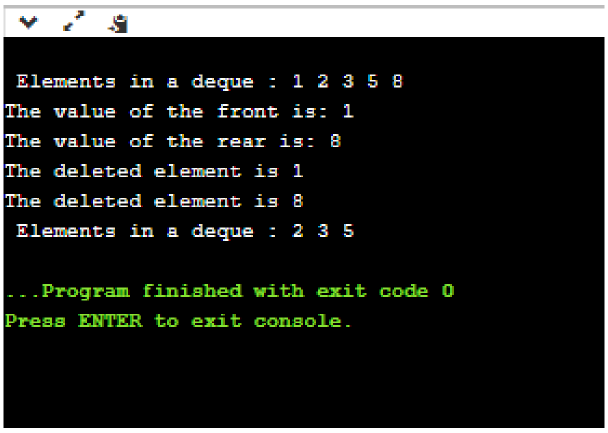
```
 }}
 // the display function prints the entire value of deque.
blank screen ()
{
 int i = f;
 printf ("\ n Elements in a deque:");
 while (i! = r)
 {
 printf ("% d", deque [i]);
 i = (i + 1)% of the dimension;
 }
 printf ("% d", deque [r]);
}
 // The getfront function retrieves the first value of the deque.
empty getfront ()
{
 if ((f == - 1) && (r == - 1))
 {
 printf ("What is empty");
 }
 the rest
 {
 printf ("\ nThe value of the front is:% d", deque [f]);
 }}
 // The getrear function retrieves the last value of the deque.
getrear empty ()
{
 if ((f == - 1) && (r == - 1))
 {
 printf ("What is empty");
 }
 the rest
 {
 printf ("\ nThe bottom value is:% d", deque [r]);
```

```
}}
// The dequeue_front () function removes the element from the front
empty dequeue_front ()
{
if ((f == - 1) && (r == - 1))
{
printf ("What is empty");
}
if not (f == r)
{
printf ("\ nThe deleted item is% d", deque [f]);
f = -1;
r = -1;
}
plus if (f == (size-1))
{
printf ("\ nThe deleted item is% d", deque [f]);
f = 0;
}
the rest
{
printf ("\ nThe deleted item is% d", deque [f]);
f = f + 1;
}}
// the dequeue_rear () function removes the element from the back
dequeue_rear empty ()
{
if ((f == - 1) && (r == - 1))
{
printf ("What is empty");
}
if not (f == r)
{
printf ("\ nThe deleted item is% d", deque [r]);
```

```
 f = -1;
 r = -1;
 }
 if not (r == 0)
 {
 printf ("\ nThe deleted item is% d", deque [r]);
 r = dimension-1;
 }
 the rest
 {
 printf ("\ nThe deleted item is% d", deque [r]);
 r = r-1;
 }}
 main integer ()
{
 // entering a value from the front.
 enqueue_front (2);
 // entering a value from the front.
 enqueue_front (1);
 // entering a value from behind.
 rear_tail (3);
 // entering a value from behind.
 enqueue_rear (5);
 // entering a value from behind.
 enqueue_rear (8);
 // Call the view function to retrieve the deque values
 show();
 // Get the front value front
 getfront ();
// Get the return value.
 getrear ();
// remove a foreground value
dequeue_front ();
// clear a value from the back
```

```
dequeue_rear ();
 // Call the view function to retrieve the deque values
 show();
 returns 0;
}
```

**Production:**



```
 Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
 Elements in a deque : 2 3 5

...Program finished with exit code 0
Press ENTER to exit console.
```

**Question:**

1. What is a dequeue?

a) A queue with insert/delete defined for both front and rear ends of the queue

b) A queue implemented with a doubly linked list

c) A queue implemented with both singly and doubly linked lists

d) A queue with insert/delete defined for front side of the queue

2. What are the applications of dequeue?

a) A-Steal job scheduling algorithm

b) Can be used as both stack and queue

c) To find the maximum of all sub arrays of size k

d) To avoid collision in hash tables

3. What is the time complexity of deleting from the rear end of the dequeue implemented with a singly linked list?

a) O(nlogn)

b) O(logn)

c) O(n)

d) O(n2)

❖❖❖❖

# Module VI (Tree)

## Experiment No-1

**Aim:**

Creating Binary search tree.

**Objective:**

Writing C++ program to create binary search tree.

**Theory:**

A binary tree is a special type of tree that can only have up to two children. This means that a particular node in a binary tree can have no child, one child, or two children but not more. A Binary search tree is a type of binary tree.

- Binary Search Tree is a data structure in which nodes are arranged in a specific order.
- It has one root node (topmost node in hierarchy).
- The left subtree of a root node contains value lesser than the root node's value.
- The right subtree of a root node contains value greater than the root node's value.
- The left and right subtree of the root node each must also be a binary search tree.

Some of the important terms are as follows:

**Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent.

**Child node:** If the node is a descendant of any node, then the node is known as a child node.

**Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.

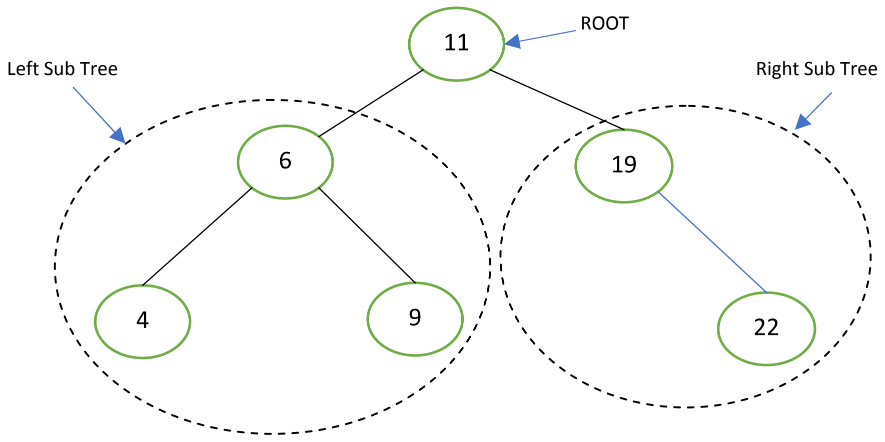
**Sibling:** The nodes that have the same parent are known as siblings.

**Leaf Node**: The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

**Internal nodes:** A node that has atleast one child node is known as an internal node.

**Ancestor node:** An ancestor of a node is any predecessor node on a path from the root to that node.

**Descendant:** The immediate successor of the given node is known as a descendant of a node.



Binary search Tree Example

**Algorithm:**

Inserting node in BST

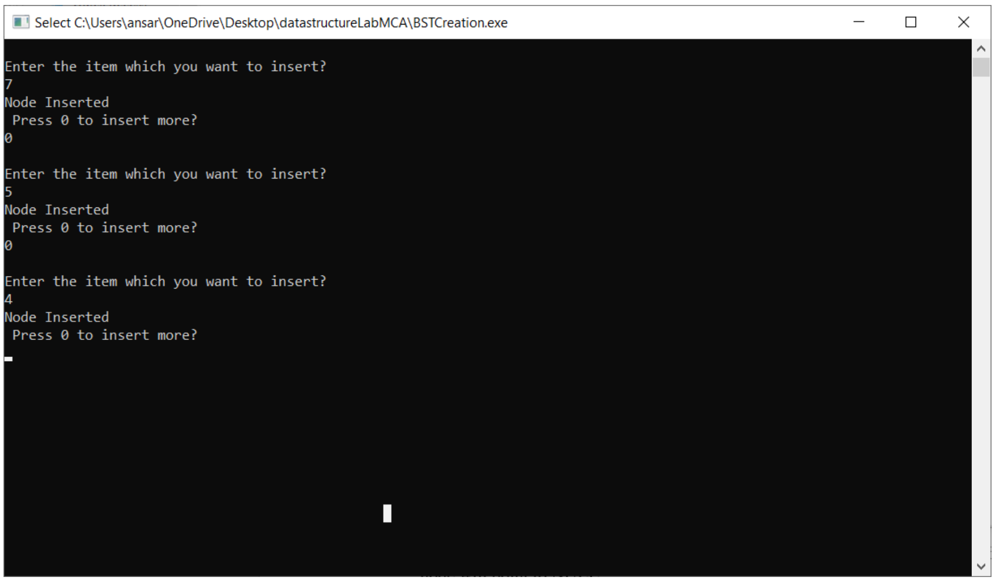
1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted is the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check whether the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left sub tree of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

**Program:**

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
using namespace std;
void insert(int);
struct node
{
int data;
struct node *left;
struct node *right;
};
struct node *root;
int main ()
{
int choice,item;
do
{
cout<<"\nEnter the item which you want to insert?\n";
cin>>item;
insert(item);
cout<<"\n Press 0 to insert more? \n";
cin>>choice;
}while(choice == 0);
return 0;
}
void insert(int item)
{
struct node *ptr, *parentptr , *nodeptr;
ptr = (struct node *) malloc(sizeof (struct node));
if(ptr == NULL)
{
```

```
cout<<"cannot insert";
}
else
{
ptr -> data = item;
ptr -> left = NULL;
ptr -> right = NULL;
if(root == NULL)
{
root = ptr;
root -> left = NULL;
root -> right = NULL;
}
else
{
parentptr = NULL;
nodeptr = root;
while(nodeptr != NULL)
{
parentptr = nodeptr;
if(item < nodeptr->data)
{
nodeptr = nodeptr -> left;
}
else
{
nodeptr = nodeptr -> right;
}
}
if(item < parentptr -> data)
{
parentptr -> left = ptr;
}
else
{
parentptr -> right = ptr;
}
}
cout<<"Node Inserted";
}
```

**OUTPUT:**



Select C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\BSTCreation.exe

```
Enter the item which you want to insert?
7
Node Inserted
 Press 0 to insert more?
0

Enter the item which you want to insert?
5
Node Inserted
 Press 0 to insert more?
0

Enter the item which you want to insert?
4
Node Inserted
 Press 0 to insert more?
```

**Question:**

1) The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?

   A. 2

   B. 3

   C. 5

   D. 4

2) Construct a Binary search tree by inserting the following numbers from left to right:

   11,6,8,19,4,10,5,18,43,49,31

3) A binary search tree is generated by inserting in order the following integers:

   50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

   The number of nodes in the left subtree and right subtree of the root respectively is

   A. (4, 7)

   B. (7, 4)

   C. (8, 3)

   D. (3, 8)

**Experiment No2**

**Aim:**

Traversal of Binary Search Tree

**Objective:**

Writing c++ program to traverse through the Binary Search tree using all the three method.

**Theory:**

Traversal refers to the process of visiting each node in a tree (binary search tree). Because, all nodes are connected via edges (links), traversing always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways in which we can traverse a binary search tree −

1) In-order Traversal
2) Pre-order Traversal
3) Post-order Traversal

**1) In Order Traversal** (left-root-right)

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. If a binary tree is traversed in in-order, the output will produce sorted key values in an ascending order.

Example:

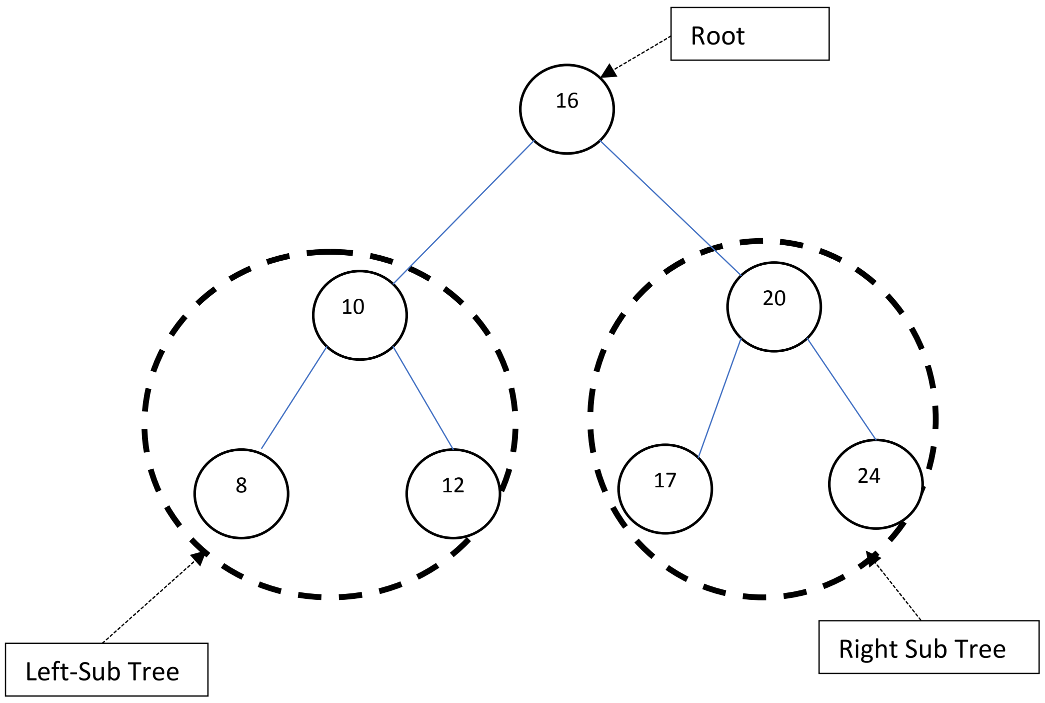


**Fig:2.1**

If the above binary search tree is traversed using In-Order Traversal method then the node traverse will be in the following order:-

**8-10-12-16-17-20-24**

In the above example we start from **16**, and following in-order traversal, we move to its left subtree **10**. **10** is also traversed in-order. The process goes on until all the nodes are visited.

**2) Pre-Order Traversal (Root-Left-Right)**

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Consider the fig 2.1, if this binary search tree is traversed using Pre-Order Traversal method, then the node traverse will be in the following order: -

**16-10-8-12-20-17-24**

We start from **16**, and following pre-order traversal, we first visit **16** itself and then move to its left subtree **10**. **10** is also traversed pre-order. The process goes on until all the nodes are visited.

### 3) Post-Order Traversal (Left-Right-Root)

In this traversal method, the root node is visited last, hence it is called as Post Order Traversal. First, we traverse the left subtree, then the right subtree and finally the root node.

Consider the fig 2.1, if this binary search tree is traversed using Post-Order Traversal method, then the node traverse will be in the following order: -

**8-12-10-17-24-20-16**

We start from **16**, and following post-order traversal, we first visit the left subtree **10**. **10** is also traversed post-order. The process goes on until all the nodes are visited.

Algorithm:

```
Inorder(tree)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Preorder (tree)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Postorder(tree)
  1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  2. Traverse the right subtree, i.e., call Postorder(right-subtree)
  3. Visit the root.
```

Program:

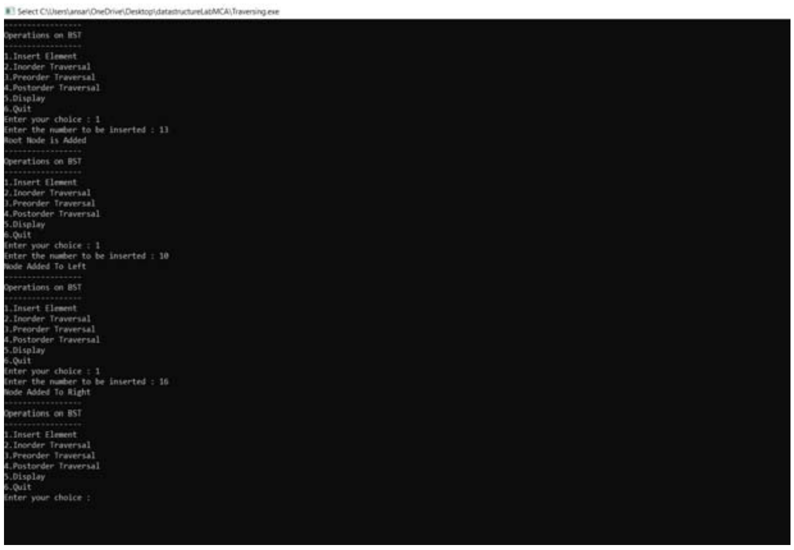
```
# include <iostream>

# include <cstdlib>

using namespace std;


// Binary Search Tree Node Declaration

struct node

{

   int info;

   struct node *left;
```

```
    struct node *right;
}*root;

// Class Declaration for Binary Search Tree
class BST
{
    public:

        void insert(node *,node *) ;
        void preorder(node *);
        void inorder(node *);
        void postorder(node *);
        void display(node *, int);
        BST()
        {
            root = NULL;
        }
};

// Main Contains Menu
int main()
{
    int choice, num;
    BST bst;
    node *temp;
    while (1)
    {
// Main menu for Binary Search Tree Operations
        cout<<"-----------------"<<endl;
        cout<<"Operations on BST"<<endl;
        cout<<"-----------------"<<endl;
        cout<<"1.Insert Element "<<endl;
```

```
cout<<"2.Inorder Traversal"<<endl;
cout<<"3.Preorder Traversal"<<endl;
cout<<"4.Postorder Traversal"<<endl;
cout<<"5.Display"<<endl;
cout<<"6.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
  case 1:
    temp = new node;
    cout<<"Enter the number to be inserted : ";
    cin>>temp->info;
    bst.insert(root, temp);
    break;
                case 2:
    cout<<"Inorder Traversal of BST:"<<endl;
    bst.inorder(root);
    cout<<endl;
    break;
  case 3:
    cout<<"Preorder Traversal of BST:"<<endl;
    bst.preorder(root);
    cout<<endl;
    break;
  case 4:
    cout<<"Postorder Traversal of BST:"<<endl;
    bst.postorder(root);
    cout<<endl;
    break;
  case 5:
    cout<<"Display BST:"<<endl;
```

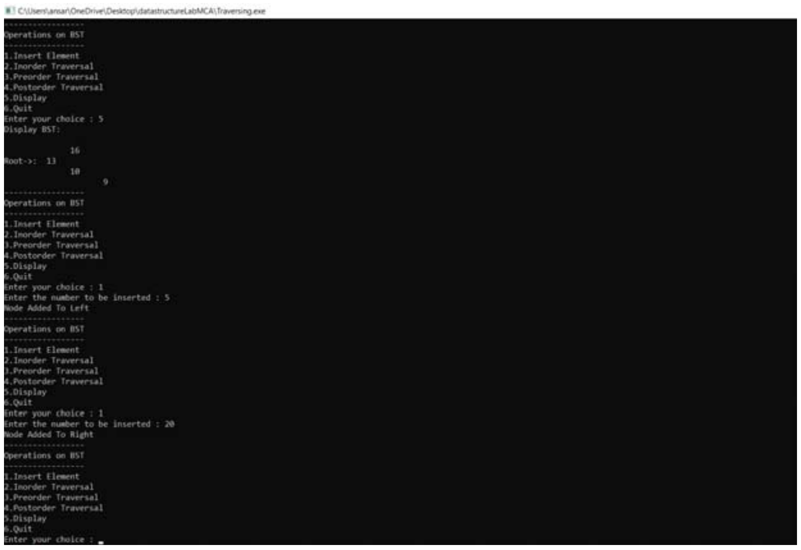
```
                bst.display(root,1);
                cout<<endl;
                break;
            case 6:
                exit(1);
            default:
                cout<<"Wrong choice"<<endl;
        }
    }
}

// Inserting Element into the Binary Search Tree
void BST::insert(node *tree, node *newnode)
{
    if (root == NULL)
    {
        root = new node;
        root->info = newnode->info;
        root->left = NULL;
        root->right = NULL;
        cout<<"Root Node is Added"<<endl;
        return;
    }
    if (tree->info == newnode->info)
    {
        cout<<"Element already in the tree"<<endl;
        return;
    }
    if (tree->info > newnode->info)
    {
        if (tree->left != NULL)
        {
            insert(tree->left, newnode);
        }
        else
        {
            tree->left = newnode;
            (tree->left)->left = NULL;
            (tree->left)->right = NULL;
            cout<<"Node Added To Left"<<endl;
            return;
        }
    }
    else
    {
        if (tree->right != NULL)
        {
            insert(tree->right, newnode);
        }
        else
        {
            tree->right = newnode;
            (tree->right)->left = NULL;
            (tree->right)->right = NULL;
            cout<<"Node Added To Right"<<endl;
            return;
        }
    }
}

// Pre Order Traversal
void BST::preorder(node *ptr)
{
    if (root == NULL)
    {
        cout<<"Tree is empty"<<endl;
        return;
    }
    if (ptr != NULL)
    {
        cout<<ptr->info<<"  ";
        preorder(ptr->left);
        preorder(ptr->right);
    }
}

// In Order Traversal
void BST::inorder(node *ptr)
{
    if (root == NULL)
    {
        cout<<"Tree is empty"<<endl;
        return;
    }
    if (ptr != NULL)
    {
        inorder(ptr->left);
        cout<<ptr->info<<"  ";
        inorder(ptr->right);
    }
}

// Postorder Traversal
void BST::postorder(node *ptr)
{
    if (root == NULL)
    {
        cout<<"Tree is empty"<<endl;
        return;
    }
    if (ptr != NULL)
    {
        postorder(ptr->left);
        postorder(ptr->right);
        cout<<ptr->info<<"  ";
    }
}

// Display Binary Search Tree Structure
void BST::display(node *ptr, int level)
{
    int i;
    if (ptr != NULL)
    {
        display(ptr->right, level+1);
        cout<<endl;
        if (ptr == root)
            cout<<"Root->:  ";
        else
        {
            for (i = 0;i < level;i++)
                cout<<"       ";
        }
        cout<<ptr->info;
        display(ptr->left, level+1);
    }
}
```

**Output:**



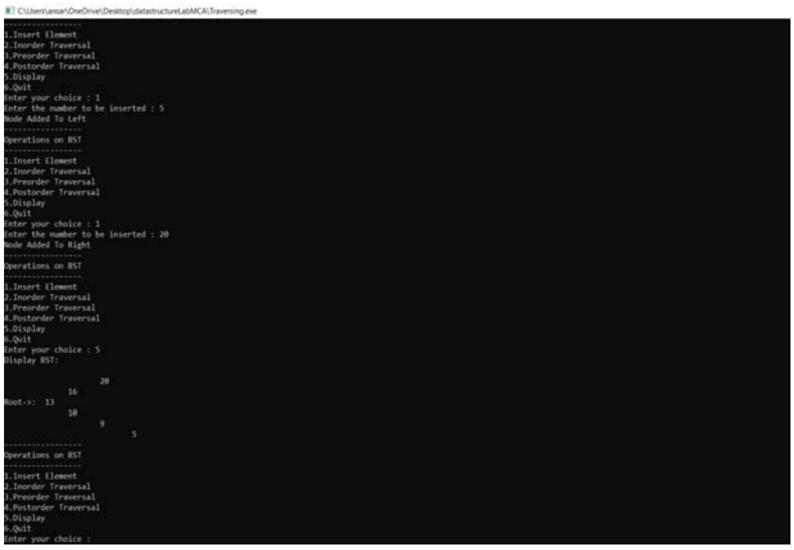
Select C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\Traversing.exe

```
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 13
Root Node is Added
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 10
Node Added To Left
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 16
Node Added To Right
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice :
```



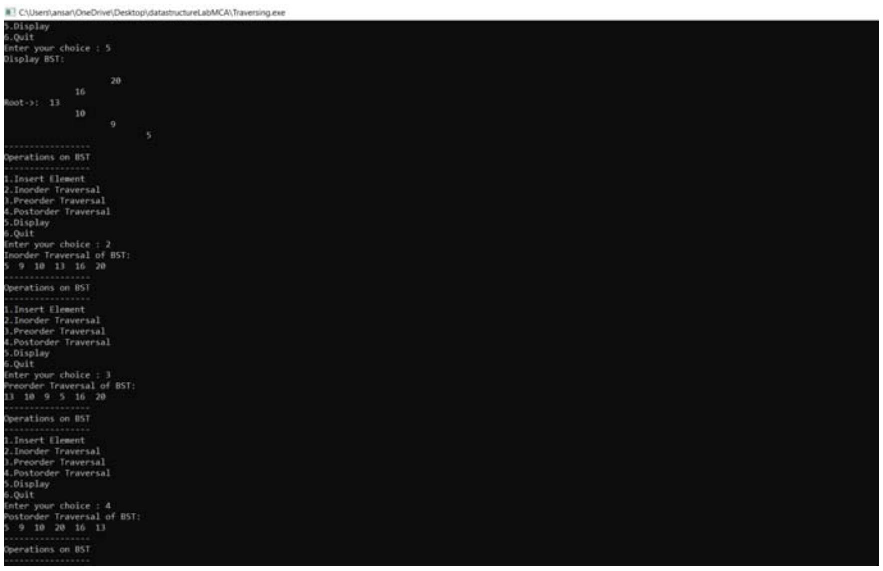
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\Traversing.exe

```
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 5
Display BST:

             16
Root->:  13
             10
                   9
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 5
Node Added To Left
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 20
Node Added To Right
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : _
```



C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\Traversing.exe

```
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 5
Node Added To Left
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 1
Enter the number to be inserted : 20
Node Added To Right
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 5
Display BST:

                   20
             16
Root->:  13
             10
                   9
                         5
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice :
```



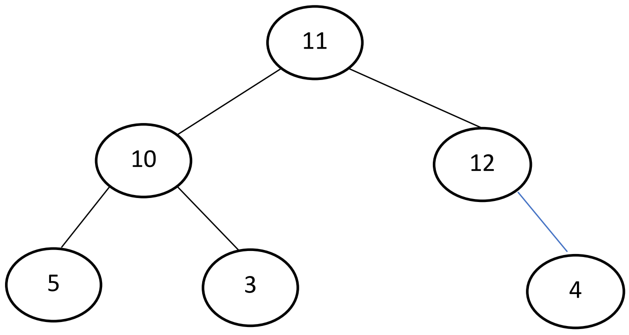
```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\Traversing.exe
5.Display
6.Quit
Enter your choice : 5
Display BST:

                    20
             16
Root->:  13
             10
                    9
                           5
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 2
Inorder Traversal of BST:
5  9  10  13  16  20
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 3
Preorder Traversal of BST:
13  10  9  5  16  20
-----------------
Operations on BST
-----------------
1.Insert Element
2.Inorder Traversal
3.Preorder Traversal
4.Postorder Traversal
5.Display
6.Quit
Enter your choice : 4
Postorder Traversal of BST:
5  9  10  20  16  13
-----------------
Operations on BST
-----------------
```

Question:

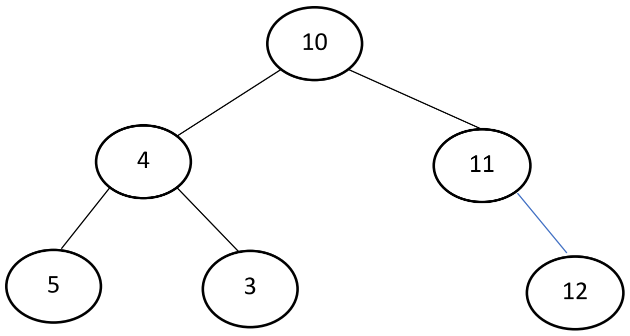
1) What is the specialty about the in-order traversal of a binary search tree?
    a) It traverses in a non-increasing order
    b) It traverses in an increasing order
    c) It traverses in a random fashion
    d) It traverses based on priority of the node

2) The in-order and preorder traversal of a binary tree are d b e a f c g and a b d e c f g, respectively. The post-order traversal of the binary tree is:
    a) d e b f g c a
    b) e d b g f c a
    c) e d b f g c a
    d) d e f g b c a

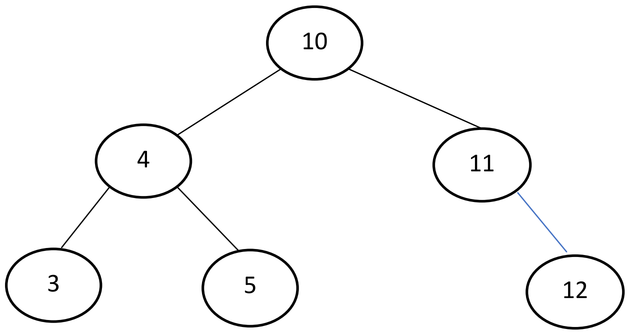
3) Construct a binary search tree with the below information.
   The preorder traversal of a binary search tree 10, 4, 3, 5, 11, 12.



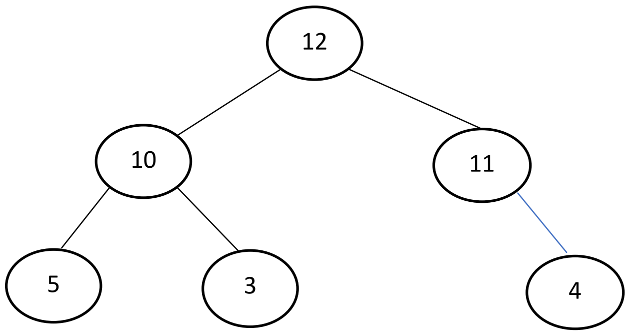
a)



b)



c)



d)

**Experiment No**:3

**Aim:**

Finding Maximum and Minimum Node of the binary search tree.

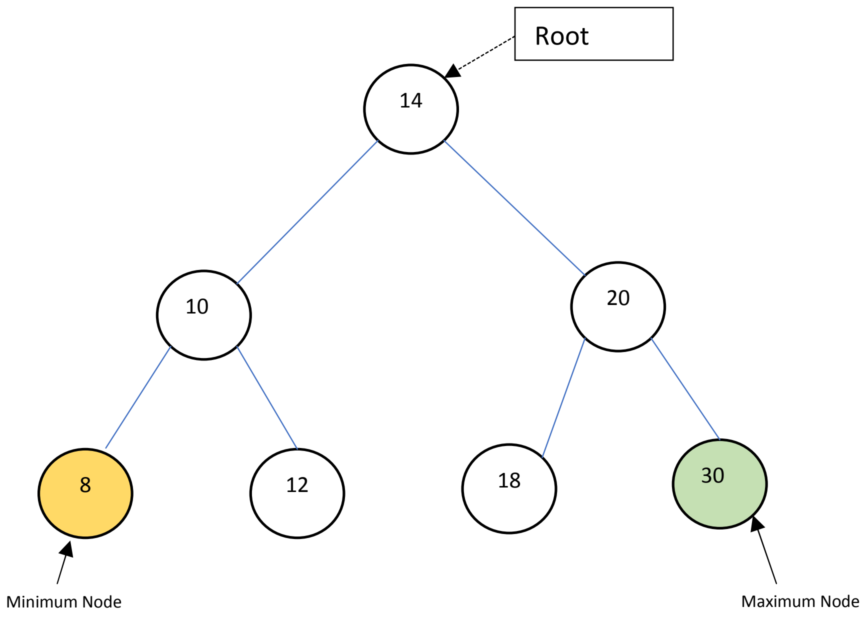
**Objective:**

Writing c++ program to find maximum and minimum node of the binary search tree.

**Theory:**

As in the binary search tree, nodes less than root node goes to the left and nodes greater than root node goes to the right. So, in Binary Search Tree, we can find maximum node by traversing right pointers until we reach the rightmost node. Similarly, we can find minimum node by traversing left pointer until we reach the leftmost node.

For example, consider the following Binary Search Tree,



As shown in the above figure, the minimum node is obtained by traversing left sub tree and maximum node is obtained by traversing right sub tree.

In the above example **8** is the minimum node and **30** is the maximum node of the binary search tree.

**Algorithm:**

**Finding Minimum Node:**

1. Starting from the root node go to its left child.
2. Keep traversing the left children of each node until a node with no left child is reached. That node is a node with minimum value.

**Finding Maximum Node:**

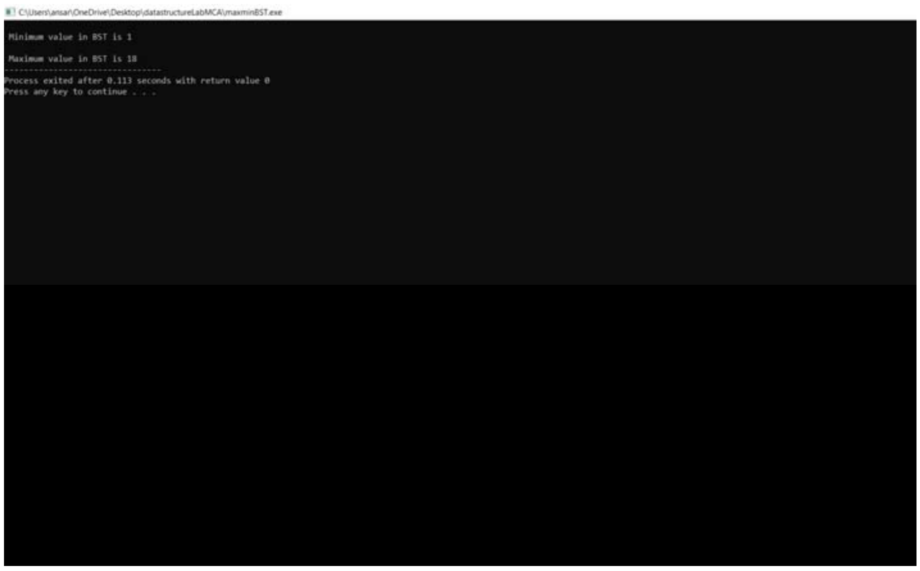
1. Starting from the root node go to its right child.
2. Keep traversing the right children of each node until a node with no right child is reached. That node is a node with maximum value.

**Program:**

```
// C++ program to find maximum or minimum element in binary search
tree
 # include <iostream>
# include <cstdlib>

using namespace std;

struct node
{
   int key;
   struct node *left, *right;
};

// A  function to create a new BST node
struct node *newNode(int item)
{
   struct node *temp =  (struct node *)malloc(sizeof(struct node));
   temp->key = item;
   temp->left = temp->right = NULL;
   return temp;
}

/* A  function to insert a new node with given value or key in BST */
struct node* insert(struct node* node, int key)
{
   struct node *newNode(int );
   /* If the tree is empty, return a new node */
   if (node == NULL) return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);

   /* return the (unchanged) node pointer */
   return node;
}
/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. */
int minValue(struct node* node)
{
   struct node* current = node;
```

```
    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
    {
        current = current->left;
    }
    return(current->key);
}
/* Given a non-empty binary search tree,
return the maximum data value found in that
tree. Note that the entire tree does not need
to be searched. */
int maxValue(struct node* node)
{
    struct node* current = node;

    while (current->right != NULL)
    {
        current = current->right;
    }
    return(current->key);
}

int main()
{
    int maxValue(struct node* );
    struct node* insert(struct node* , int );
    int minValue(struct node* );
    struct node *root = NULL;
    root = insert(root, 8);
    insert(root, 3);
    insert(root, 10);
    insert(root, 1);
    insert(root, 6);
    insert(root, 4);
    insert(root, 7);
    insert(root, 5);
    insert(root, 10);
    insert(root, 9);
    insert(root, 13);
    insert(root, 11);
    insert(root, 18);
    insert(root, 12);
    insert(root, 2);

    cout << "\n Minimum value in BST is " << minValue(root)<<endl;
    cout << "\n Maximum value in BST is " << maxValue(root);
    return 0;
}
```

Output:



```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\maxminBST.exe

 Minimum value in BST is 1

 Maximum value in BST is 18
--------------------------------
Process exited after 0.113 seconds with return value 0
Press any key to continue . . .
```

Question:

1) What will be the minimum element of the binary search tree if no left subtree exist?
2) Which of the following is false about a binary search tree?
    a) The left child is always lesser than its parent
    b) The right child is always greater than its parent
    c) The left and right sub-trees should also be binary search trees
    d) In order sequence gives decreasing order of elements
3) What are the worst case and average case complexities of a binary search tree?
    a) O(n), O(n)
    b) O(logn), O(logn)
    c) O(logn), O(n)
    d) O(n), O(logn)

**Experiment:4**

**Aim:**

Counting number of nodes in binary search tree.

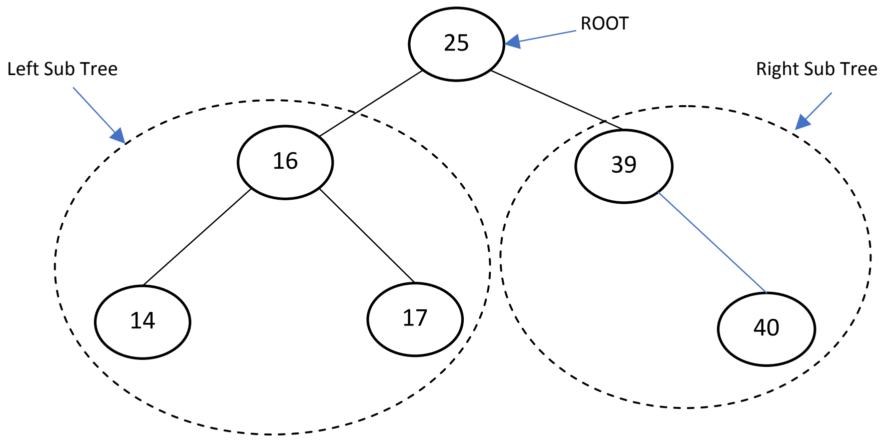
**Objectives:**

Writing c++ program to count number of nodes in the binary search tree.

**Theory:**

To count number of nodes in the binary search tree we can use the following formula: -

Total No. of nodes in BST=Total No. of nodes in left sub-tree + Total no. of node in right sub-tree + 1



Binary search Tree

In the above binary search tree, the total number of Nodes is **6.**
No. of nodes in Left Sub-tree=3
No. of nodes in Right sub-tree=2
No. of Root Node=1
So, the total no of nodes will be = 3+2+1=6.

**Algorithm:**

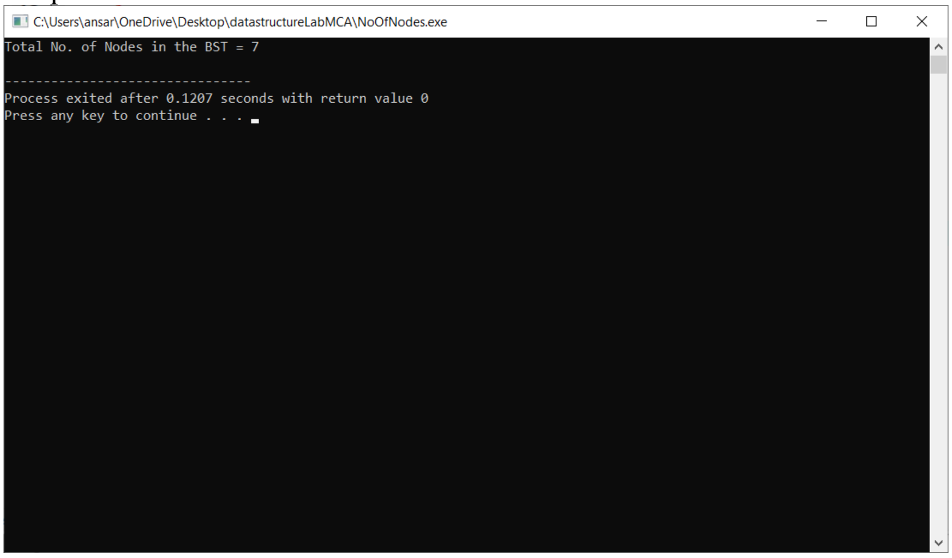
```
1. Initialize "count" variable as 1.
2. If root is NULL, return 0.
3. Else, count = count + countNodes(root -> left) and
    count = count + countNodes(root -> right).
4. Then, return count.
5. End If
```

**Program:**

```
#include<iostream>
using namespace std;
int n=1;
struct node
{
	int data;
	node* left;
	node* right;
};

struct node* getNode(int data)
{
	node* newNode=new node();
	newNode->data=data;
	newNode->left=NULL;
	newNode->right=NULL;
	return newNode;
}

struct node* Insert(struct node* root, int data)
{
	if (root == NULL)
		return getNode(data);

	if (data < root->data)
		root->left  = Insert(root->left, data);
	else if (data > root->data)
		root->right = Insert(root->right, data);

	return root;
}


int CountNodes(node*root)
{
	if(root==NULL)
		return 0;
	if(root->left!=NULL)
	{
		n=n+1;
```

```
            n=CountNodes(root->left);
        }
        if(root->right!=NULL)
        {
            n=n+1;
            n=CountNodes(root->right);
        }
        return n;
}

int main()
{
        node* root=NULL;
        root=Insert(root,3);
        Insert(root,4);
        Insert(root,2);
        Insert(root,5);
        Insert(root,1);
        Insert(root,6);
        Insert(root,8);

        cout<<"Total Number of Nodes in the BST =
"<<CountNodes(root)<<endl;

        return 0;
}
```

Output:



C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\NoOfNodes.exe

```
Total No. of Nodes in the BST = 7

--------------------------------
Process exited after 0.1207 seconds with return value 0
Press any key to continue . . .
```

**Questions:**

1) What are sub trees in binary search tree?
2) How to calculate height of the binary search tree?
3) What are the various application of binary search tree?

**Experiment No-5**

**Aim:**

Creating Max Heap

**Objective:**

Writing c++ program to create max heap data structure.

**Theory:**

**Heap**: A Heap is a special Tree-based data structure in which the tree is a complete binary tree, that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

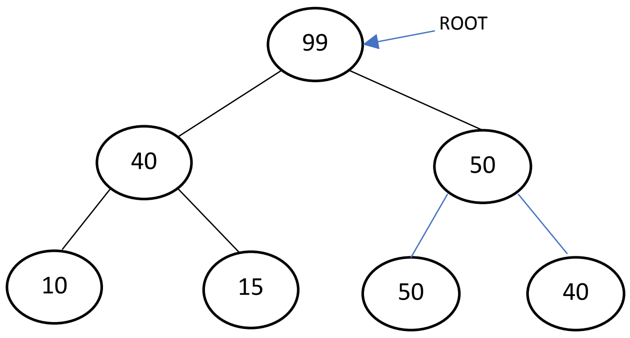
Generally, Heaps can be of two types:

**Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Note: Heap data Structure is implemented using Arrays in Programming Languages.**

Example of Max-Heap



MaxHeap

In the above figure , 99 is the root node and it is the highest node in the tree.In left sub tree 40 is parent of 10 and 15, which is again greater.In right sub tree 50 is the parent of 50 and 40,which is also greater.

| 99 | 40 | 50 | 10 | 15 | 50 | 40 |
|---|---|---|---|---|---|---|
| **Index 0** | **1** | **2** | **3** | **4** | **5** | **6** |

**Array Representation of the above max-heap:**

**Note: In Heap data structure two nodes can have same value, that is repetition of the value is allowed in heap data structure.**

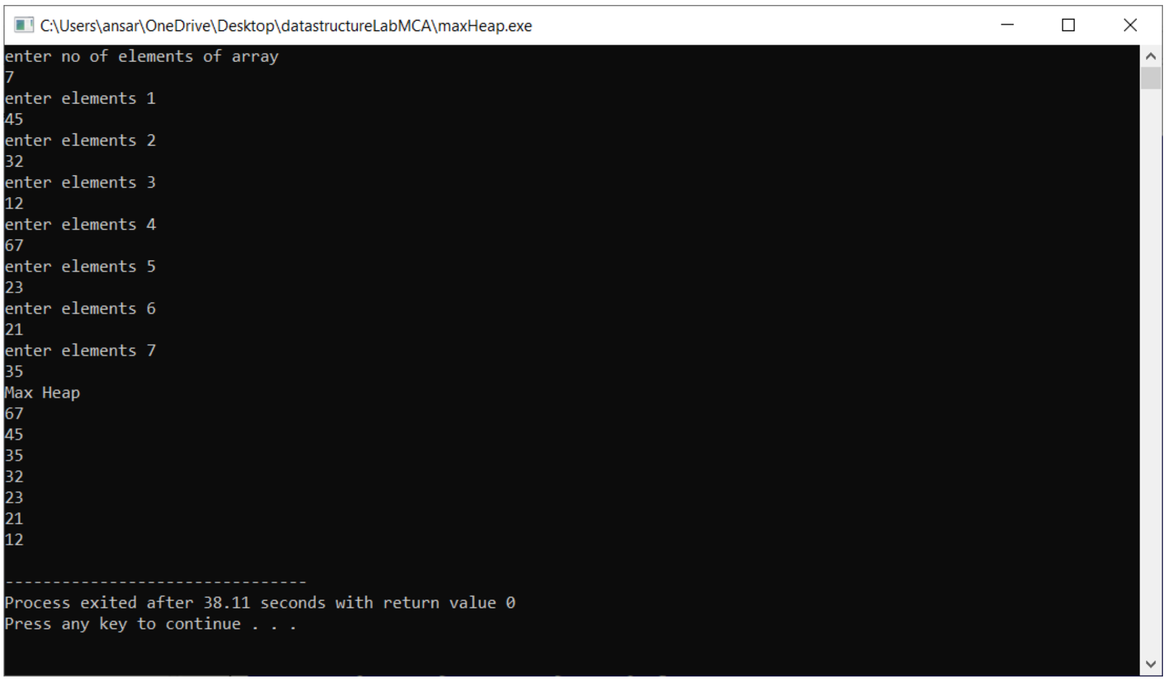
**Algorithm:**

1. Create a new node at the end of heap.
2. Assign new value to the node.
3. Compare the value of this child node with its parent.
4. If value of parent is less than child, then swap them.
5. Repeat step 3 & 4 until Heap property holds (i.e. parent >= child).

**Program:**

```
#include <iostream>
using namespace std;
void max_heap(int *a, int m, int n) {
  int j, t;
  t = a[m];
  j = 2 * m;
  while (j <= n) {
    if (j < n && a[j+1] > a[j])
      j = j + 1;
    if (t > a[j])
      break;
    else if (t <= a[j]) {
      a[j / 2] = a[j];
      j = 2 * j;
    }
  }
  a[j/2] = t;
  return;
}
void build_maxheap(int *a,int n) {
  int k;
  for(k = n/2; k >= 1; k--) {
    max_heap(a,k,n);
  }
}
int main() {
  int n, i;
  cout<<"enter no of elements of array\n";
  cin>>n;
  int a[30];
  for (i = 1; i <= n; i++) {
    cout<<"enter elements"<<" "<<(i)<<endl;
    cin>>a[i];
  }
  build_maxheap(a,n);
  cout<<"Max Heap\n";
  for (i = 1; i <= n; i++) {
    cout<<a[i]<<endl;
  }
}
```

**Output:**


C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\maxHeap.exe

```
enter no of elements of array
7
enter elements 1
45
enter elements 2
32
enter elements 3
12
enter elements 4
67
enter elements 5
23
enter elements 6
21
enter elements 7
35
Max Heap
67
45
35
32
23
21
12

--------------------------------
Process exited after 38.11 seconds with return value 0
Press any key to continue . . .
```

**Questions:**

1. Consider a binary max-heap implemented using an array. Which one of the following arrays represents a binary max-heap? (GATE CS 2009)
    a) 25,12,16,13,10,8,14
    b) 25,12,16,13,10,8,14
    c) 25,14,16,13,10,8,12
    d) 25,14,12,13,10,8,16
2. When do you need to use a heap?
3. Construct a max heap using [12,10,9,8,5,2].

**Experiment No-6**

**Aim:** Creating Min Heap

**Objective:**

Writing c++ program to create min heap data structure.

**Theory:**

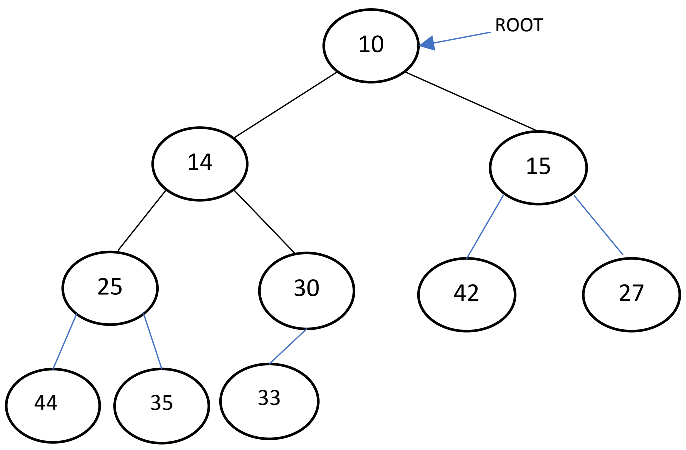
**Heap**: A Heap is a special Tree-based data structure in which the tree is a complete binary tree, that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

Generally, Heaps can be of two types:

**Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Example of Min-Heap



MinHeap

In the above figure, 10 is the root node and it is the smallest node in the tree. Same is true for every sub node.In min heap,the value of the root node is less than or equal to either of its children.

**Array Representation of the above min-heap:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **14** | **15** | **25** | **30** | **42** | **27** | **44** | **35** | **33** |
| **Index 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |

**Note: In Heap data structure two nodes can have same value, that is repetition of the value is allowed in heap data structure.**

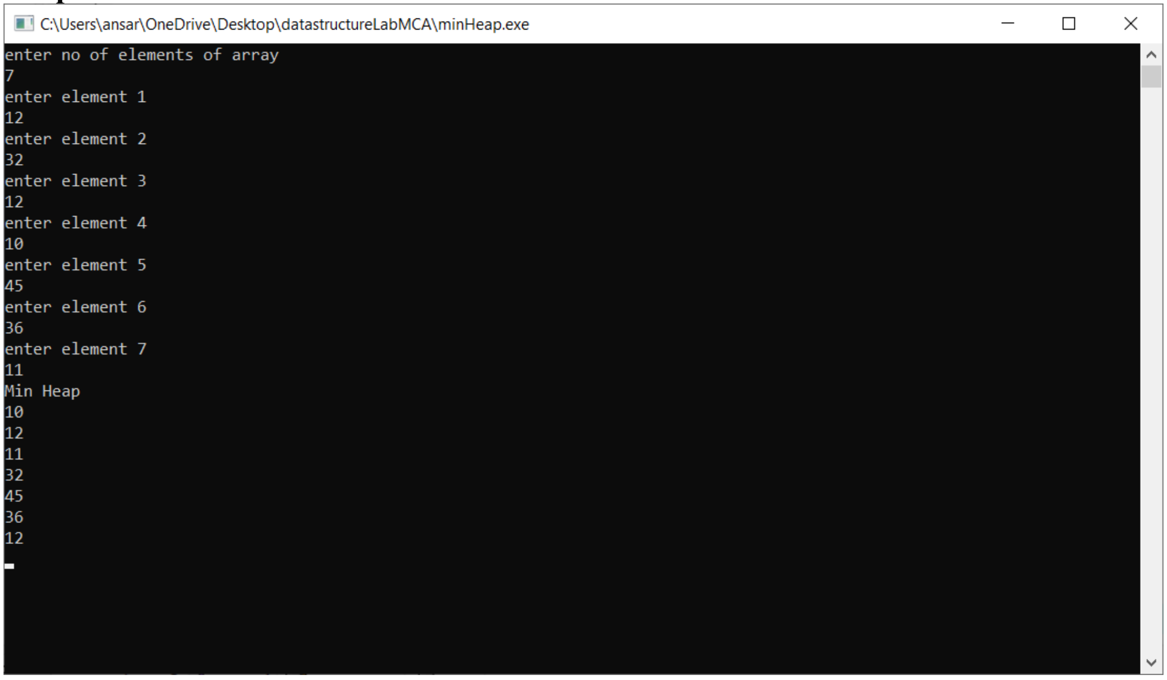
**Algorithm:**

1. Create a new node at the end of heap.
2. Assign new value to the node.
3. Compare the value of this child node with its parent.
4. If value of parent is greater than child, then swap them.
5. Repeat step 3 & 4 until Heap property holds(i.e parent <= child).

**Program:**

```
#include <iostream>
#include <conio.h>
using namespace std;
void min_heap(int *a, int m, int n){
  int j, t;
  t= a[m];
  j = 2 * m;
  while (j <= n) {
    if (j < n && a[j+1] < a[j])
      j = j + 1;
    if (t < a[j])
      break;
    else if (t >= a[j]) {
      a[j/2] = a[j];
      j = 2 * j;
    }
  }
  a[j/2] = t;
  return;
}
void build_minheap(int *a, int n) {
  int k;
  for(k = n/2; k >= 1; k--) {
    min_heap(a,k,n);
  }
}
int main() {
  int n, i;
  cout<<"enter no of elements of array\n";
  cin>>n;
  int a[30];
  for (i = 1; i <= n; i++) {
    cout<<"enter element"<<" "<<(i)<<endl;
    cin>>a[i];
  }
  build_minheap(a, n);
  cout<<"Min Heap\n";
  for (i = 1; i <= n; i++) {
    cout<<a[i]<<endl;
  }
  getch();
}
```

**Output:**


C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\minHeap.exe

```
enter no of elements of array
7
enter element 1
12
enter element 2
32
enter element 3
12
enter element 4
10
enter element 5
45
enter element 6
36
enter element 7
11
Min Heap
10
12
11
32
45
36
12
```

**Questions:**

1. In a min-heap, element with the lowest key is always in which node?
    a) Leaf node
    b) Root node
    c) First node of left sub tree
    d) First node of right sub tree

2. Which one of the following array elements represents a binary min heap?
    a) 12 10 8 25 14 17
    b) 8 10 12 25 14 17
    c) 25 17 14 12 10 8
    d) 14 17 25 10 12 8

3. The ascending heap property is ___________
    a) A[Parent(i)] =A[i]
    b) A[Parent(i)] <= A[i]
    c) A[Parent(i)] >= A[i]
    d) A[Parent(i)] > 2 * A[i]

**Experiment No:7**

**Aim:**

Performing Reheap Up operation on max-Heap and min Heap.

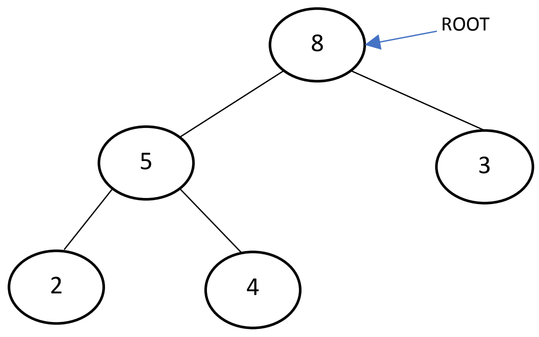
**Objective:**

Writing c++ program to perform reheap operation i.e insert new element in the Heap.

**Theory:**

Reheap Up is used when inserting new element in the heap. Whenever new element is added in the heap it is always added to the first empty leaf at the bottom level from the leftmost side. After that heap is rearranged so that newly added element reached its proper place. For max heap swapping is done, if new value is greater than previous/parent node. For min heap swapping is done, if new value is smaller than previous/parent node.

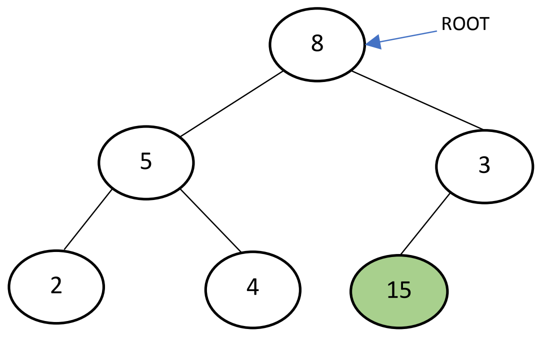
Suppose the Heap is a Max-Heap as shown below:



The new element to be inserted is 15.

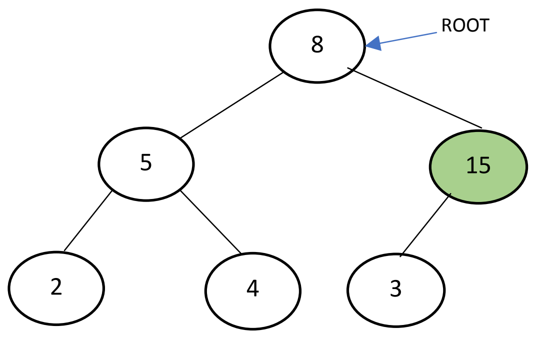
Process:

Step 1: Insert the new element at the end.

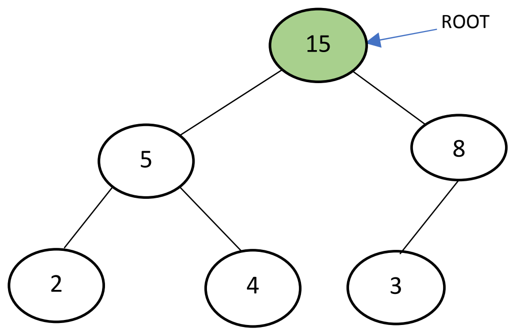


Step 2: Heapify the new element following bottom-up approach.

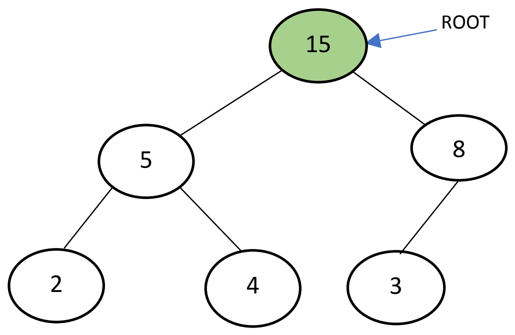
15 is more than its parent 3, swap them.



15 is again more than its parent 8, swap them.



Therefore, the final heap after insertion is:



**This Entire process is Known as ReHeap Up.** The process of insertion to Min-heap is similar to that of Max-heap but only one difference is that the value of parent node is always lesser than the child node.

**Algorithm:**

To insert an element into heap we need to follow 2 steps:

1. Insert the element at last position of the heap and increase the size of heap n to n+1.
2. Recursively test and swap the new value with previous/parent node as long as the heap property is not satisfied. For max heap swap if new value is greater than previous/parent node. For min heap swap if new value is smaller than previous/parent node.

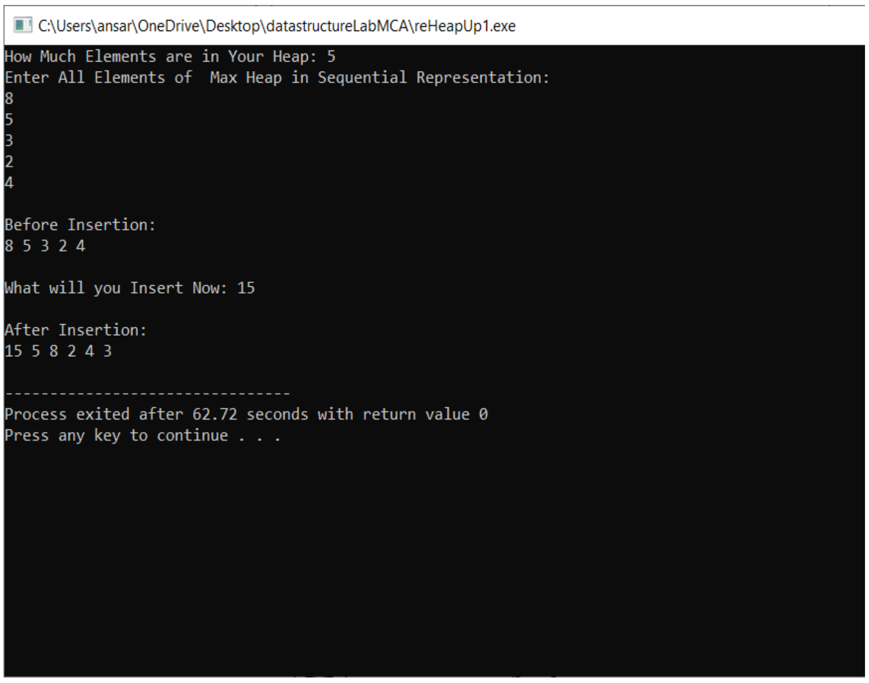
**Program:**

**C++ program to insert new element in Max-Heap:**

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

int main(){
   int n,i,m;        //n is no of elements,i is normal integer and m is used to
not change n
   cout<<"How Much Elements are in Your Heap: ";
   cin>>n;

   long H[n+5];       //H is Array to take input the elements of heap
   cout<<"Enter All Elements of Max Heap in Sequential
Representation:"<<endl;
   for(i=1;i<=n;i++){
      cin>>H[i];
   }
   cout<<endl;

   cout<<"Before Insertion:"<<endl;
   for(i=1;i<=n;i++){
      cout<<H[i]<<" ";
   }
   cout<<endl<<endl;

   cout<<"What will you Insert Now: ";
   cin>>H[n+1];              //Inserting new element at the last of heap tree
   cout<<endl;
   m=n=n+1;                  //Increasing total element number in heap

   while(H[m]>H[m/2]){        //Taking in exact position the inserted
element
      swap(H[m],H[m/2]);
      m=m/2;
   }
```

```
    cout<<"After Insertion:"<<endl;
    for(i=1;i<=n;i++){
        cout<<H[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

**Output:**

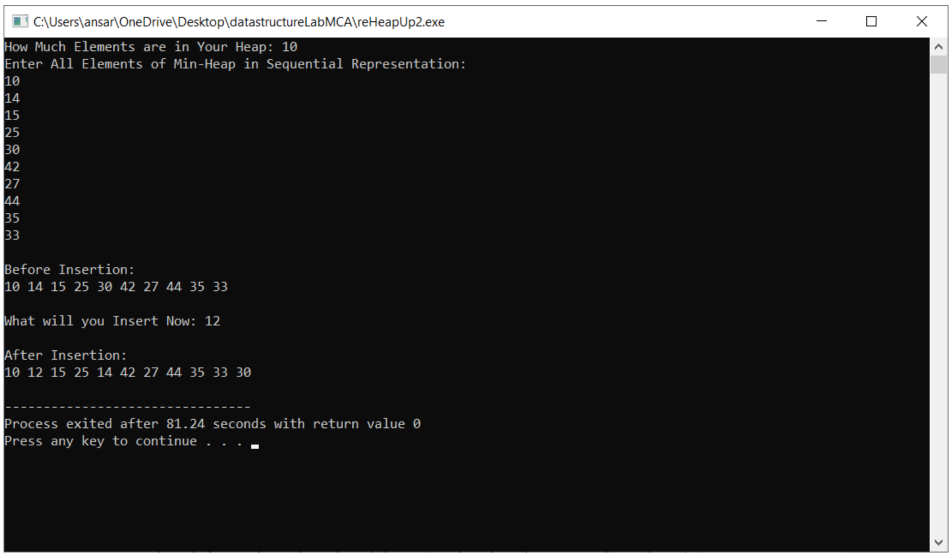


```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\reHeapUp1.exe
How Much Elements are in Your Heap: 5
Enter All Elements of  Max Heap in Sequential Representation:
8
5
3
2
4

Before Insertion:
8 5 3 2 4

What will you Insert Now: 15

After Insertion:
15 5 8 2 4 3

--------------------------------
Process exited after 62.72 seconds with return value 0
Press any key to continue . . .
```

**C++ program to insert new element in Min-Heap:**

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n,i,m;          //n is no of elements,i is normal integer and m is used to
not change n
    cout<<"How Much Elements are in Your Heap: ";
    cin>>n;

    long H[n+5];        //H is Array to take input the elements of heap
    cout<<"Enter All Elements of Min-Heap in Sequential
Representation:"<<endl;
    for(i=1;i<=n;i++){
        cin>>H[i];
    }
    cout<<endl;

    cout<<"Before Insertion:"<<endl;
    for(i=1;i<=n;i++){
        cout<<H[i]<<" ";
    }
    cout<<endl<<endl;

    cout<<"What will you Insert Now: ";
    cin>>H[n+1];                //Inserting new element at the last of heap tree
    cout<<endl;
    m=n=n+1;                    //Increasing total element number in heap

    while(H[m]<H[m/2]){         //Taking in exact position the inserted
element
        swap(H[m],H[m/2]);
        m=m/2;
    }

    cout<<"After Insertion:"<<endl;
    for(i=1;i<=n;i++){
        cout<<H[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

**Output:**


C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\reHeapUp2.exe

```
How Much Elements are in Your Heap: 10
Enter All Elements of Min-Heap in Sequential Representation:
10
14
15
25
30
42
27
44
35
33

Before Insertion:
10 14 15 25 30 42 27 44 35 33

What will you Insert Now: 12

After Insertion:
10 12 15 25 14 42 27 44 35 33 30

--------------------------------
Process exited after 81.24 seconds with return value 0
Press any key to continue . . .
```

**Question:**

1. What is the correct way of adding new element to the heap?
2. Explain reheap up process in detail?
3. What is mean by Heapify?

**Experiment No:8**

**Aim:**

Performing Reheap down operation on max-Heap and min Heap.

**Objective:**

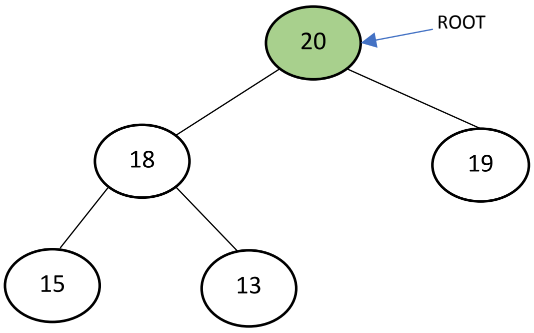
Writing c++ program to perform reheap down i.e deletion of element from the Heap.

**Theory:**

Reheap down is used when deleting an element from the heap. The standard deletion operation on Heap is to delete the element present at the root node of the Heap. That is if it is a Max Heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

Suppose the Heap is a Max-Heap as:

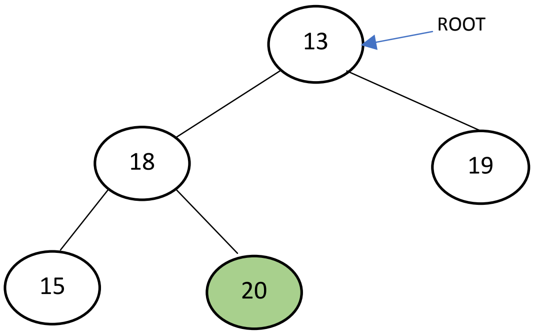


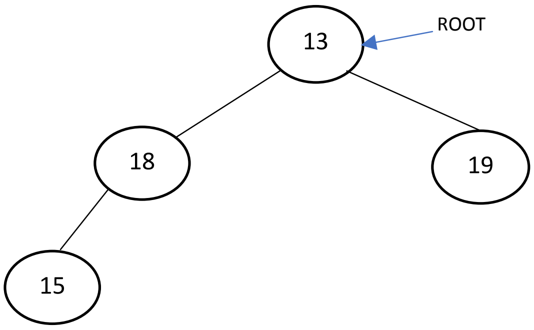
The element to be deleted is root, i.e. 20.

Process:

The last element is 13.

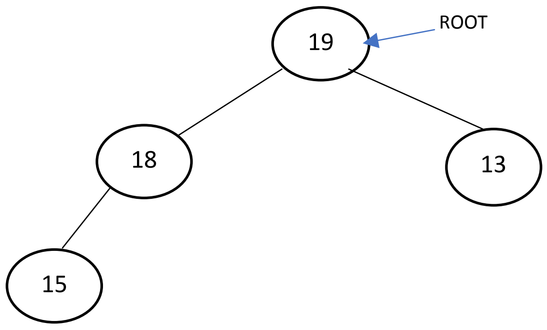
Step 1: Replace the last element with root, and delete it.





Step 2: Heapify root.

Final Heap:



**This Entire process is Known as ReHeap Down.**

**Algorithm:**

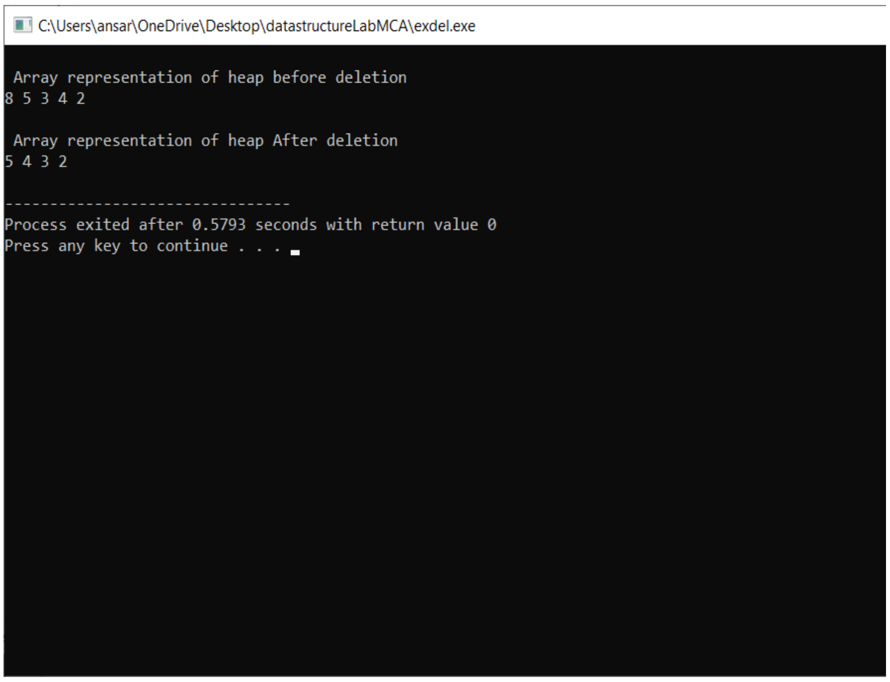
1. Replace the root or element to be deleted by the last element.
2. Delete the last element from the Heap and decrease the size of heap by 1.
3. Since, the last element is now placed at the position of the root node or at deleted element place. Recursively test and swap the replaced value with next/child nodes as long as the heap property is not satisfied. For max heap swap if replaced value is smaller than next/child nodes. For min heap swap if replaced value is greater than next/child nodes.

**Program:**

```
// C++ program for implement deletion in Heaps
#include <iostream>
using namespace std;
// To heapify a subtree rooted with node i which is
// an index of arr[] and n is the size of heap
void heapify(int arr[], int n, int i)
{
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2
        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
```

```
                largest = l;
        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
                largest = r;
        // If largest is not root
        if (largest != i) {
                swap(arr[i], arr[largest]);

                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
        }
}

// Function to delete the root from Heap
void deleteRoot(int arr[], int& n)
{
        // Get the last element
        int lastElement = arr[n - 1];

        // Replace root with last element
        arr[0] = lastElement;

        // Decrease size of heap by 1
        n = n - 1;

        // heapify the root node
        heapify(arr, n, 0);
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
        for (int i = 0; i < n; ++i)
                cout << arr[i] << " ";
        cout << "\n";
}

int main()
{
        // Array representation of Max-Heap
        //          8
        // / \
        // 5 3
        // / \
        // 2 4
        int arr[] = { 8, 5, 3, 4, 2 };

        int n = sizeof(arr) / sizeof(arr[0]);
       cout<<"\n Array representation of heap before deletion\n";
        printArray(arr, n);

        deleteRoot(arr, n);
        cout<<"\n Array representation of heap After deletion\n";
        printArray(arr, n);
        return 0;
}
```

**Output:**



C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\exdel.exe

```
 Array representation of heap before deletion
8 5 3 4 2

 Array representation of heap After deletion
5 4 3 2

--------------------------------
Process exited after 0.5793 seconds with return value 0
Press any key to continue . . .
```

**Question:**

1. How many arrays are required to perform deletion operation in a heap?
   a. 1
   b. 2
   c. 3
   d. 4

2. What is the time taken to perform a delete min operation?
    a. O(N)
    b. O(N log N)
    c. O(log N)
    d. O(N2)

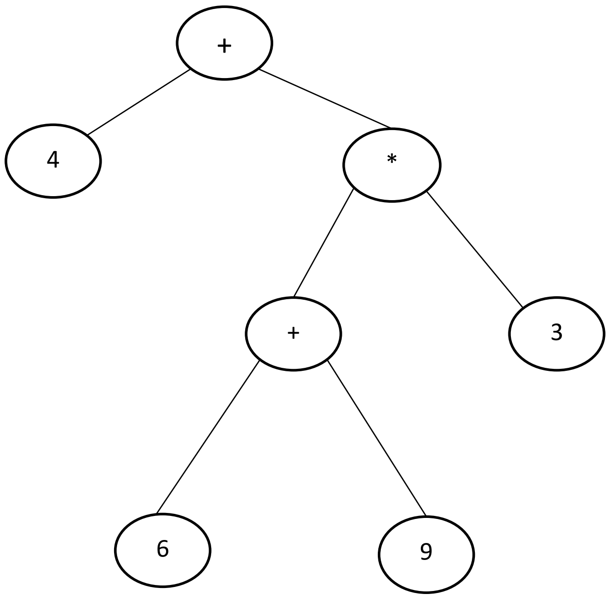
3. Which element from the heap can be deleted?

**Self-Learning Topics:**

**Expression Tree:**

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand. So, for example, **expression tree** for

**4 + ((6+9) *3)** would be:



**Heap Sort:**

Heap sort is a comparison-based sorting technique based on Heap data structure Heap sort involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.

❖❖❖❖

# Module VII

**Experiment No:1**

**Aim:**

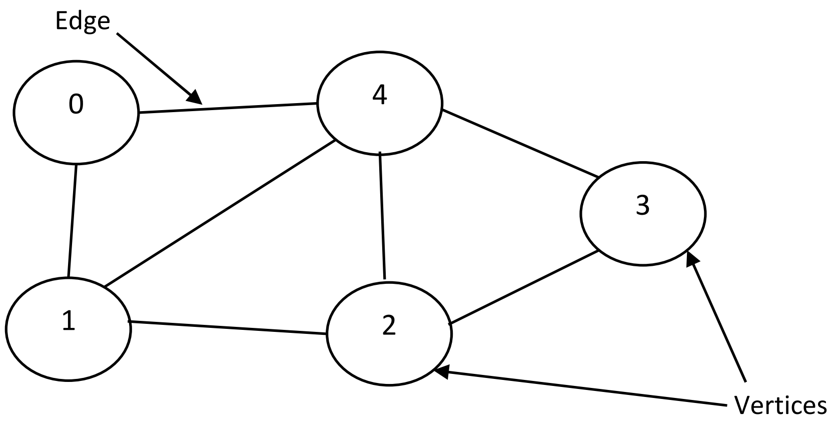
Graph Creation using Adjacency matrix

**Objective:**

Writing C++ program for representation of a graph using adjacency matrix.

**Theory:**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. In other words, we can say that, A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



Graph

**In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 40, 14,24}.**

**Graph Terminology**

**Adjacency:** A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 0 are not adjacent because there is no edge between them.

**Path:** A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1-2-4 is a path from vertex 0 to vertex 4.

**Directed Graph:** The edges in such a graph are represented by arrows to show the direction of the edge.

**Graph Representation:**

The following are the two most commonly used representations of a graph.

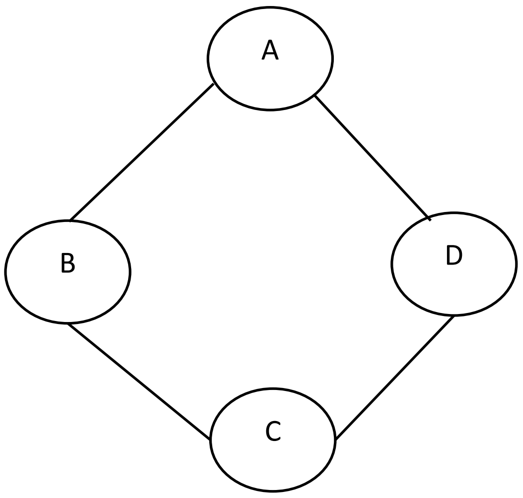
1. **Adjacency Matrix**
2. **Adjacency List**

**Adjacency matrix representation of the Graph:**

An adjacency matrix is used to represent adjacent nodes in the graph. Two nodes are said to be adjacent if there is an edge connecting them. We represent graph in the form of matrix in Adjacency matrix representation. For a graph G, if there is an edge between two vertices a and b then we denote it 1 in matrix. If there is no edge then denote it with 0 in matrix.

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj [][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

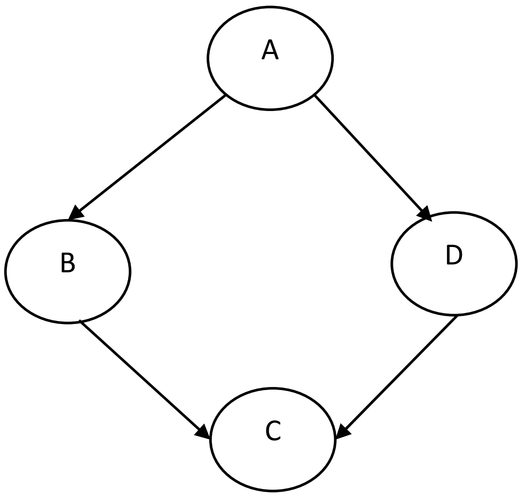
Consider the following Graph and its Adjacency matrix Representation:



Undirected Graph

Adjacency matrix representation of the above undirected graph:

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 |
| B | 1 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 1 |
| D | 1 | 0 | 1 | 0 |



Directed Graph

Adjacency matrix representation of the above Directed graph:

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 |
| B | 0 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |

**Algorithm:**

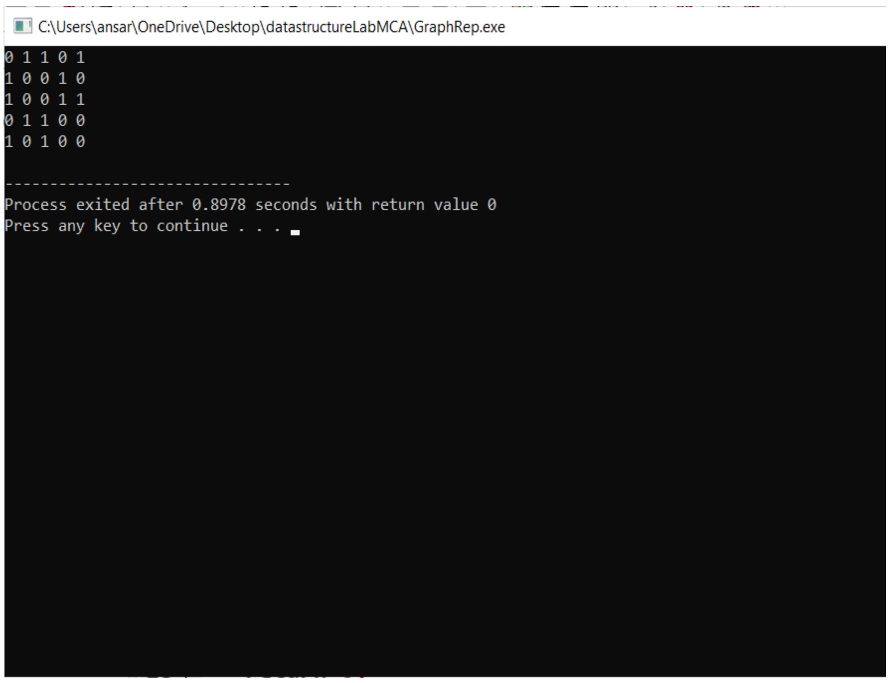
1. Create a 2D array (say Adj[N+1] [N+1]) of size NxN and initialize all value of this matrix to zero. Store all the edges of the graph in arr [] [].

2. For each edge in the arr [][], Update value at Adj[X][Y] and Adj[Y][X] to 1, denotes that there is a edge between X and Y.
3. Display the Adjacency Matrix after the above operation for all the pairs in arr[][] is completed.

**Program:**

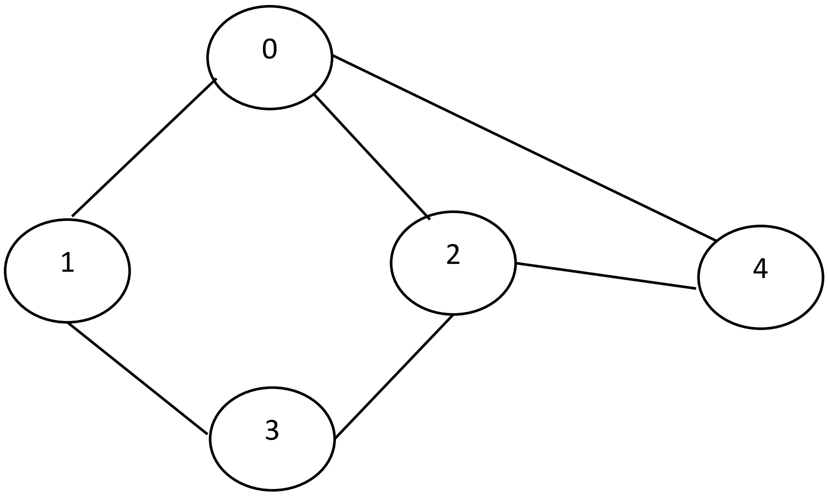
```
#include<iostream>
using namespace std;
int vertArr[20][20]; //the adjacency matrix initially 0
int count = 0;
void displayMatrix(int v) {
  int i, j;
  for(i = 0; i < v; i++) {
    for(j = 0; j < v; j++) {
      cout << vertArr[i][j] << " ";
    }
    cout << endl;
  }
}
void add_edge(int u, int v) {      //function to add edge into the matrix
  vertArr[u][v] = 1;
  vertArr[v][u] = 1;
}
 int main() {
  int v = 5;    //there are 6 vertices in the graph
  add_edge(0, 1);
  add_edge(0, 2);
  add_edge(0, 4);
  add_edge(1, 3);
  add_edge(3, 2);
  add_edge(2, 4);
  displayMatrix(v);
  return 0;
}
```

**Output:**



```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\GraphRep.exe
0 1 1 0 1
1 0 0 1 0
1 0 0 1 1
0 1 1 0 0
1 0 1 0 0

--------------------------------
Process exited after 0.8978 seconds with return value 0
Press any key to continue . . .
```

**Note: The above output is for the following graph:**



**Question:**

1) Give various application of graph data structure in our daily life.
2) Given below the Adjacency matrix representation of the graph,Draw the Graph:

|  | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 0 | 0 | 0 | 1 |
| C | 0 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 0 |

3) Which of the following statements for a simple graph is correct?
   a) Every path is a trail
   b) Every trail is a path
   c) Every trail is a path as well as every path is a trail
   d) None of the mentioned

**Experiment No:2**

**Aim:**

Performing Breadth First Search (BFS) traversal on Graph data structure.

**Objective:**

Writing C++ program to perform BFS traversal on Graph.

**Theory:**

**Graph Traversal:**

There are two types of graph traversal algorithms. These are called the **Breadth First Search (BFS)and Depth First Search (DFS).**

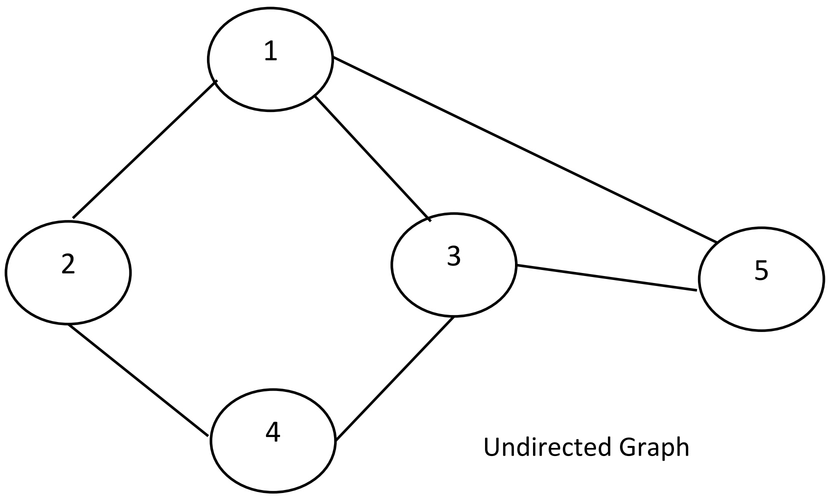
**Breadth First Search (BFS)**

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again. This process will continue until all nodes are visited.

Here, are important rules for using BFS algorithm:

- A queue (FIFO-First in First Out) data structure is used by BFS.
- You mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
- Removes the previous vertex from the queue in case no adjacent vertex is found.
- BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.
- There are no loops caused by BFS during the traversing of data from any node.

Consider the following graph:



Traversal of the above graph Using BFS will be:**1-2-3-5-4.**

According to BFS traversal method, first step is to visit any vertex, so we have visited **1**. Next step is to explore that visited vertex, that means we have to visit adjacent vertex of 1, so we have visited **2,3 and 5**. Again we have to visit any unvisited vertex, so we have visited **4**.

**Algorithm:**

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

**The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.**

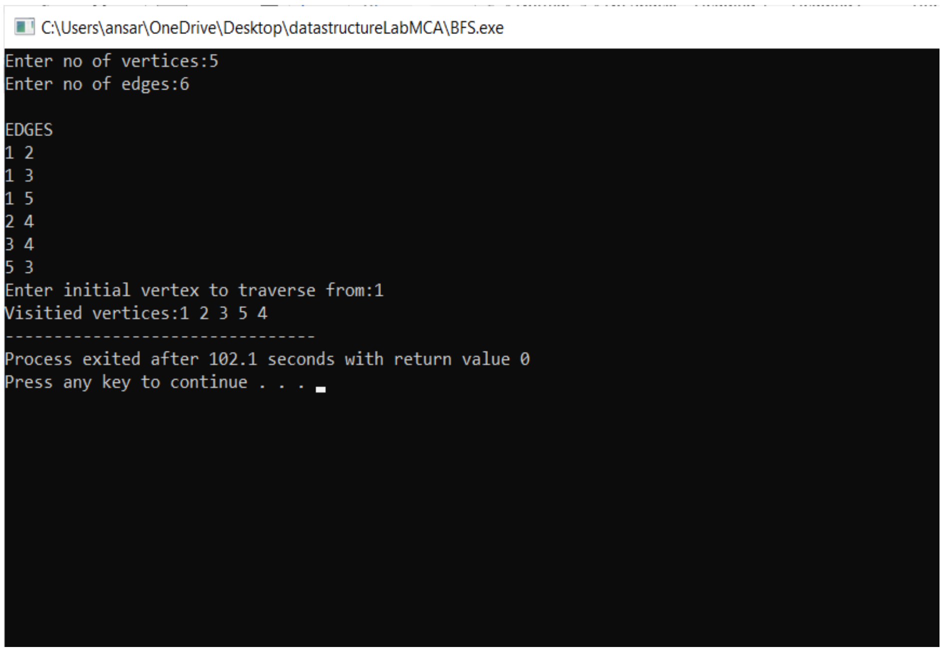
**The algorithm is as follows:**

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.
5. The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

**Program:**

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,qu[10],front,rare,v,visit[10],visited[10];
int main()
{
  int m;
  cout <<"Enter no of vertices:";
  cin >> n;
  cout <<"Enter no of edges:";
  cin >> m;
  cout <<"\nEDGES \n";
  for(k=1; k<=m; k++)
  {
    cin >>i>>j;
    cost[i][j]=1;
  }
  cout <<"Enter initial vertex to traverse from:";
  cin >>v;
  cout <<"Visitied vertices:";
  cout <<v<<" ";
  visited[v]=1;
  k=1;
  while(k<n)
  {
    for(j=1; j<=n; j++)
       if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
       {
          visit[j]=1;
          qu[rare++]=j;
       }
    v=qu[front++];
    cout<<v <<" ";
    k++;
    visit[v]=0;
    visited[v]=1;
  }
  return 0;
}
```

**Output:**



```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\BFS.exe
Enter no of vertices:5
Enter no of edges:6

EDGES
1 2
1 3
1 5
2 4
3 4
5 3
Enter initial vertex to traverse from:1
Visitied vertices:1 2 3 5 4
--------------------------------
Process exited after 102.1 seconds with return value 0
Press any key to continue . . .
```

**Note:** In the above program,while entering edges give space between two vertexes of the edges.

**Questions:**

1. The Date structure used in standard implementation of Breadth First Search is?
   a) Stack
   b) Queue
   c) LinkedList
   d) Tree

2. In BFS,how many times a node is visited?
   a) Once
   b) Twice
   c) Equivalent to number of indegree of the node
   d) Thrice

3. Who describe the Best First Search algorithm using heuristic evaluation rule?
   a) Judea Pearl
   b) Max Bezzel
   c) Franz Nauck
   d) Alan Turing

**Experiment No:3**

**Aim:**

Finding Minimum Spanning tree using Kruskal's algorithm.

**Objective:**

Writing c++ program to find minimum spanning tree using Kruskal's algorithm from a given graph.

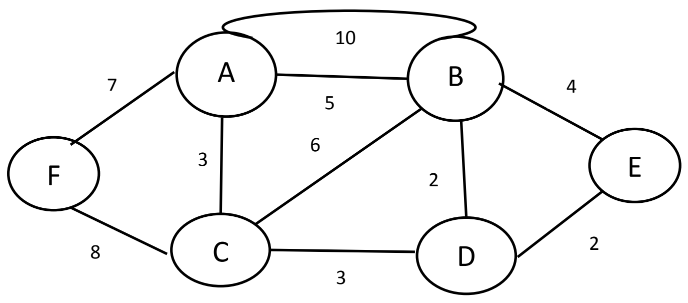
**Theory:**

For any connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees.

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Consider the following weighted graph.

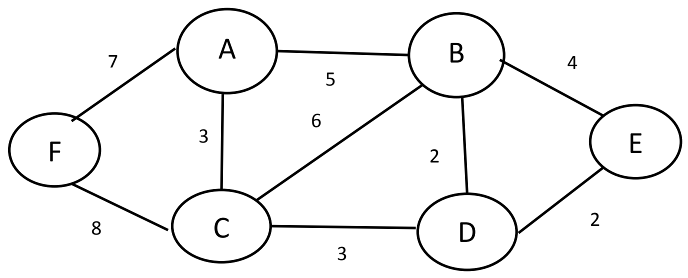


Weighted Graph

To find the MST using Kruskal's algorithm we have to follow the following steps:

1) Remove any loop present in the graph. Check for any parallel edges in the graph and remove any one (Edge with maximum weight will be removed).

   In above graph, there is two edges from A-B, they are called as parallel edges, one with weight 5 and other with weight 10. So, the edge with weight 10 will be removed.



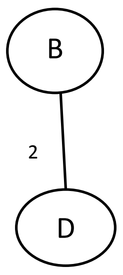
2) Next, we have to sort the edges according to their weight in the ascending order.

So, for our example it will be:

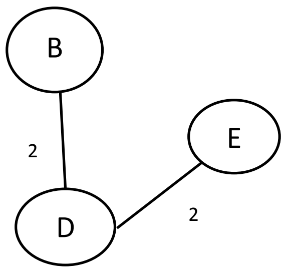
| Weight | Edge |
|---|---|
| 2 | BD |
| 2 | DE |
| 3 | CD |
| 3 | AC |
| 4 | BE |
| 5 | AB |
| 6 | CB |
| 7 | AF |
| 8 | FC |

3) Now pick all edges one by one from the sorted list of edges

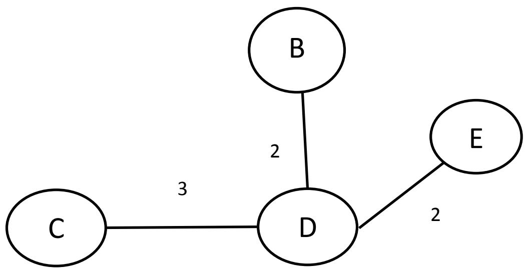
Pick edge B-D: No cycle is formed, include it.



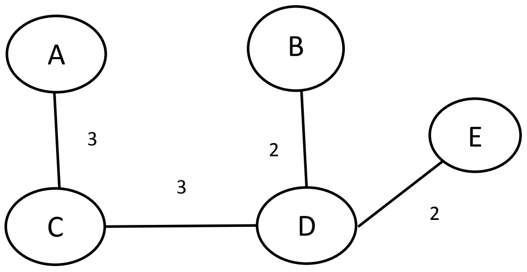
Pick edge D-E: No cycle is formed, include it.



Pick edge C-D: No cycle is formed, include it.



Pick edge A-C: No cycle is formed, include it.

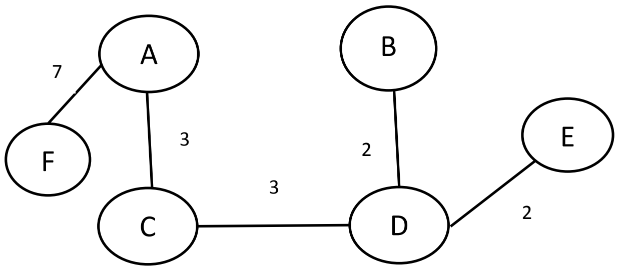


Pick edge B-E: Since including this edge results in the cycle, discard it.

Pick edge A-B: Since including this edge results in the cycle, discard it.

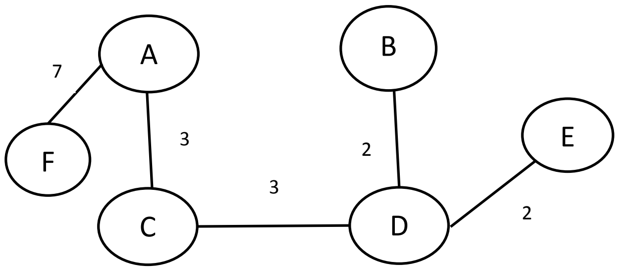
Pick edge C-B: Since including this edge results in the cycle, discard it.

Pick edge A-F: No cycle is formed, include it.



Pick edge F-C: Since including this edge results in the cycle, discard it.

So, the final minimum spanning tree using Kruskal’s algorithm will be:



**Algorithm:**

Kruskal’s Algorithm

This algorithm will create spanning tree with minimum weight, from a given weighted graph.

1. Begin
2. Remove loop and parallel edge.
3. Create the edge list of given graph, with their weights.
4. Sort the edge list according to their weights in ascending order.

5. Draw all the nodes to create skeleton for spanning tree.
6. Pick up the edge at the top of the edge list (i.e., edge with minimum weight).
7. Remove this edge from the edge list.
8. Connect the vertices in the skeleton with given edge. If by connecting the vertices, a cycle is created in the skeleton, then discard this edge.
9. Repeat steps 5 to 7, until n-1 edges are added or list of edges is over.
10. Return

**Program:**

```
// C++ program for Kruskal's algorithm
// to find Minimum Spanning Tree of a
// given connected, undirected and weighted
// graph
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a
// weighted edge in graph
class Edge {
public:
        int src, dest, weight;
};

// a structure to represent a connected,
// undirected and weighted graph
class Graph {
public:

        // V-> Number of vertices, E-> Number of edges
        int V, E;

        // graph is represented as an array of edges.
        // Since the graph is undirected, the edge
        // from src to dest is also edge from dest
        // to src. Both are counted as 1 edge here.
        Edge* edge;
};
```

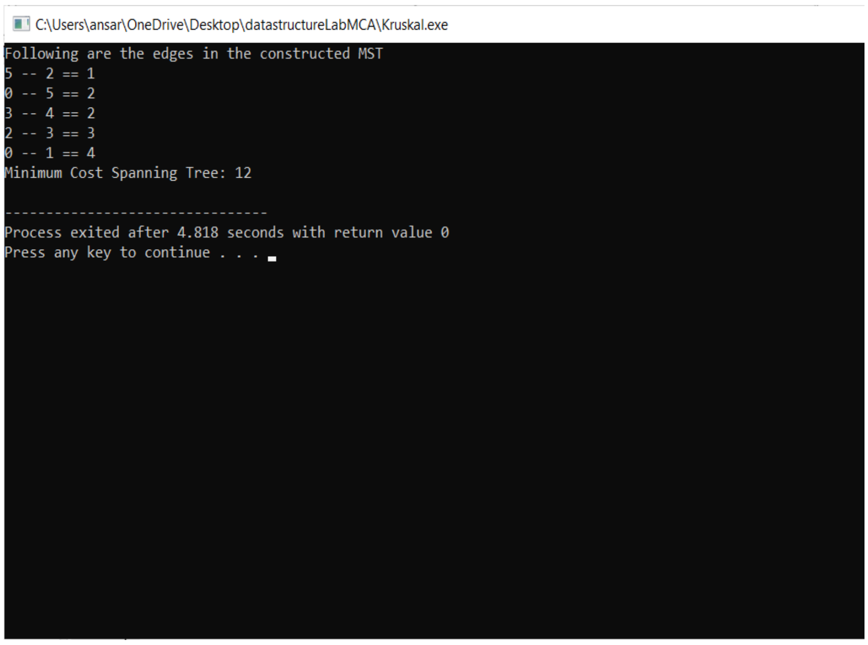
```
// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
        Graph* graph = new Graph;
        graph->V = V;
        graph->E = E;

        graph->edge = new Edge[E];

        return graph;
}

// A structure to represent a subset for union-find
class subset {
public:
        int parent;
        int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
        // find root and make root as parent of i
        // (path compression)
        if (subsets[i].parent != i)
                subsets[i].parent
                        = find(subsets, subsets[i].parent);

        return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        // Attach smaller rank tree under root of high
        // rank tree (Union by Rank)
        if (subsets[xroot].rank < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;

        // If ranks are same, then make one as root and
        // increment its rank by one
        else {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
        Edge* a1 = (Edge*)a;
        Edge* b1 = (Edge*)b;
        return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's
// algorithm
void KruskalMST(Graph* graph)
{
        int V = graph->V;
        Edge result[V]; // Tnis will store the resultant MST
        int e = 0; // An index variable, used for result[]
        int i = 0; // An index variable, used for sorted edges

        // Step 1: Sort all the edges in non-decreasing
        // order of their weight. If we are not allowed to
        // change the given graph, we can create a copy of
        // array of edges
        qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
                myComp);

        // Allocate memory for creating V ssubsets
        subset* subsets = new subset[(V * sizeof(subset))];
```

```
// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
        subsets[v].parent = v;
        subsets[v].rank = 0;
}

// Number of edges to be taken is equal to V-1
while (e < V - 1 && i < graph->E)
{
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y) {
                result[e++] = next_edge;
                Union(subsets, x, y);
        }
        // Else discard the next_edge
}

// print the contents of result[] to display the
// built MST
cout << "Following are the edges in the constructed "
                "MST\n";
int minimumCost = 0;
for (i = 0; i < e; ++i)
{
        cout << result[i].src << " -- " << result[i].dest
                << " == " << result[i].weight << endl;
        minimumCost = minimumCost + result[i].weight;
}

cout << "Minimum Cost Spanning Tree: " << minimumCost
```

```
                    << endl;
}

int main()
{
        int V = 6; // Number of vertices in graph
        int E = 8; // Number of edges in graph
        Graph* graph = createGraph(V, E);

        // add edge 0-1
        graph->edge[0].src = 0;
        graph->edge[0].dest = 1;
        graph->edge[0].weight = 4;

        // add edge 0-5
        graph->edge[1].src = 0;
        graph->edge[1].dest = 5;
        graph->edge[1].weight = 2;

        // add edge 1-2
        graph->edge[2].src = 1;
        graph->edge[2].dest = 2;
        graph->edge[2].weight = 6;

        // add edge 2-3
        graph->edge[3].src = 2;
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 3;
        // add edge 3-4
        graph->edge[4].src = 3;
        graph->edge[4].dest = 4;
        graph->edge[4].weight =2;

// add edge 4-5
        graph->edge[5].src = 4;
        graph->edge[5].dest =5;
        graph->edge[5].weight =4;

// add edge 5-1
        graph->edge[6].src = 5;
        graph->edge[6].dest =1;
        graph->edge[6].weight =5;
```

```
// add edge 5-2
        graph->edge[7].src = 5;
        graph->edge[7].dest =2;
        graph->edge[7].weight =1;

        // Function call
        KruskalMST(graph);

        return 0;
}
```

**Output:**



```
C:\Users\ansar\OneDrive\Desktop\datastructureLabMCA\Kruskal.exe
Following are the edges in the constructed MST
5 -- 2 == 1
0 -- 5 == 2
3 -- 4 == 2
2 -- 3 == 3
0 -- 1 == 4
Minimum Cost Spanning Tree: 12

--------------------------------
Process exited after 4.818 seconds with return value 0
Press any key to continue . . .
```

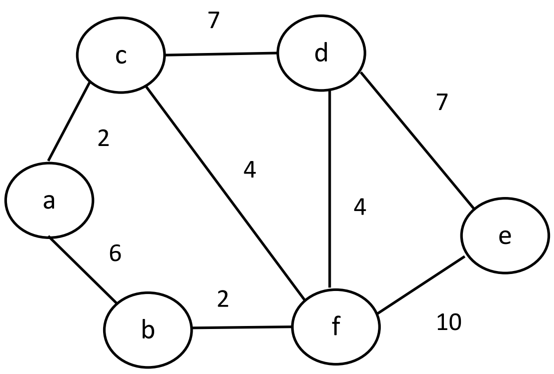
**Questions:**

1. Every graph has only one minimum spanning tree.
    a) True
    b) False

2. Which of the following is not the algorithm to find the minimum spanning tree of the given graph?
    a) Boruvka's algorithm
    b) Prim's algorithm
    c) Kruskal's algorithm
    d) Bellman–Ford algorithm

3. Kruskal’s algorithm is a ______
   a) divide and conquer algorithm
   b) dynamic programming algorithm
   c) greedy algorithm
   d) approximation algorithm

4. Consider the given graph.



What is the weight of the minimum spanning tree using the Kruskal’s algorithm?

a) 24

b) 23

c) 15

d) 19

**Self-Learning Topic:**

**Shortest Path algorithm:**

In computer networks, the shortest path algorithms aim to find the optimal paths between the network nodes so that routing cost is minimized.

Some common shortest path algorithms are −

1. Bellman Ford’s Algorithm
2. Dijkstra’s Algorithm
3. Floyd Warshall’s Algorithm

❖❖❖❖