Roll No.**24** Exam Seat No._____

VIVEKANANDEDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

Hashu Advani Memorial Complex, Collector's Colony, R. C. Marg,Chembur, Mumbai – 400074. Contact No. 02261532532

Since 1962

CERTIFICATE

Certified that Mr./Miss **NARENDER KESWANI**

of **FYMCA-1B** has satisfactorily completed a course of the necessary experiments in **MCAL11 - Data Structures Lab with C and / C++** under my supervisionin the Institute of Technology in the academic year **2021- 2022.**

Principal Head of Department

Lab In-charge Subject Teacher

V.E.S. Institute of Technology, Collector Colony, Chembur, Mumbai
Department of M.C.A

INDEX

Sr. No	Contents	Date Of Preparation	Date Of Submission	Page Number	Marks	Sign
1	Implementation of different sorting techniques. Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Radix Sort, Quick Sort.	31/12/2021	31/12/2021	1		
2	Implementation of searching algorithms Linear Search and Binary search	21/01/2022	21/01/2022	24		
3	Stack Array implementation, Linked List implementation.	31/01/202	31/01/202	33		
4	Implementation of Stack Applications like: Postfix evaluation Balancing of Parenthesis	31/01/2022	31/01/2022	45		
5	Simple Queue implementation using Linked List Circular Queue implementation using Linked List Double ended Queue implementation using Linked List	18/02/2022	18/02/2022	49		
6	Demonstrate application of queue (Priority Queue)	18/02/2022	18/02/2022	69		
7	Implementation of all types of linked List Insert, Display, Delete, Search, Count Reverse operation on • Singly Linked Lists: • Circular Linked List • Doubly Linked Lists	23/02/2022	23/02/2022	74		

8	**Binary Search Tree Creation and Traversal Operation on BST** • **Largest Node** • **Smallest Node** • **Count number of nodes**	**23/02/2022**	**23/02/2022**	**111**		
9	**Find the minimum spanning tree (using any method Kruskal's Algorithm or Prim's Algorithm)**	**04/03/2022**	**04/03/2022**	**123**		
10	**Implementation of Graph traversal. (DFS and BFS)**	**04/03/2022**	**04/03/2022**	**136**		
11	**HEAP SORT & SPARSE MATRIX**	**08/03/2022**	**08/03/2022**	**148**		
12	**Group Project**	**10/03/2022**	**10/03/2022**			

AIM: Implementation of different sorting techniques.

Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Radix Sort, Quick Sort

A) BUBBLE SORT:

THEORY:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Example:

First Pass:

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

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

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

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

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

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

ALGORITHM:

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)
for all elements of list
    if list[i] > list[i+1]
        swap(list[i], list[i+1])
    end if
end for
return list
end BubbleSort
```

PSEUDOCODE:

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

SOURCE CODE:

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

void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      cout << arr[i] << " ";
   cout << endl;
}

void bubble_sort( int A[ ], int n )
{
   int temp;
   for(int k = 0; k< n-1; k++)
   {
      // (n-k-1) is for ignoring comparisons of elements which have already been compared in earlier iterations
```

```
      for(int i = 0; i < n-k-1; i++)
      {
         if(A[ i ] > A[ i+1] )
         {
            // here swapping of positions is being done.
            temp = A[ i ];
            A[ i ] = A[ i+1 ];
            A[ i + 1] = temp;
         }

      }
      cout << "Pass" <<k <<" : ";
      printArray(A, n);
   }
}

int main()
{
   int n;
   cout << "Enter number of elements to be sorted \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }

   bubble_sort(arr, n);
   cout << "Using Bubble Sort, the sorted array is: \n";
   printArray(arr, n);
   return 0;

}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Desktop\bubblesort.exe"
Enter number of elements to be sorted
5
Enter element 1
-9
Enter element 2
9
Enter element 3
7
Enter element 4
0
Enter element 5
1
Pass0 : -9 7 0 1 9
Pass1 : -9 0 1 7 9
Pass2 : -9 0 1 7 9
Pass3 : -9 0 1 7 9
Using Bubble Sort, the sorted array is:
-9 0 1 7 9

Process returned 0 (0x0)   execution time : 14.820 s
Press any key to continue.
```

B) INSERTION SORT:

THEORY:

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Example:

12, 11, 13, 5, 6

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

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

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

11, 12, 13, 5, 6

i = 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

i = 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:

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be

sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

PSEUDOCODE:

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert

  for i = 1 to length(A) inclusive do:

    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert

  end for

end procedure
```

SOURCE CODE:

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

void printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      cout << arr[i] << " ";
```

```
    cout << endl;
}

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
        cout << "Pass" <<i <<" : ";
        printArray(arr, n);
    }
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    insertionSort(arr, n);
    cout << "Using Insertion Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;
}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Desktop\insertsort.exe"
Enter number of elements to be sorted
7
Enter element 1
-1
Enter element 2
6
Enter element 3
7
Enter element 4
2
Enter element 5
-5
Enter element 6
5
Enter element 7
3
Pass1 : -1 6 7 2 -5 5 3
Pass2 : -1 6 7 2 -5 5 3
Pass3 : -1 2 6 7 -5 5 3
Pass4 : -5 -1 2 6 7 5 3
Pass5 : -5 -1 2 5 6 7 3
Pass6 : -5 -1 2 3 5 6 7
Using Insertion Sort, the sorted array is:
-5 -1 2 3 5 6 7

Process returned 0 (0x0)   execution time : 18.681 s
Press any key to continue.
```

C) SELECTION SORT:

THEORY:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where **n** is the number of items.

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be

useful when memory write is a costly operation.

Example:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

Algorithm:

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

Pseudocode

```
procedure selection sort
  list  : array of items
  n     : size of list

  for i = 1 to n - 1
  /* set current element as minimum*/
    min = i

    /* check the element to be minimum */

    for j = i+1 to n
      if list[j] < list[min] then
        min = j;
      end if
    end for
```

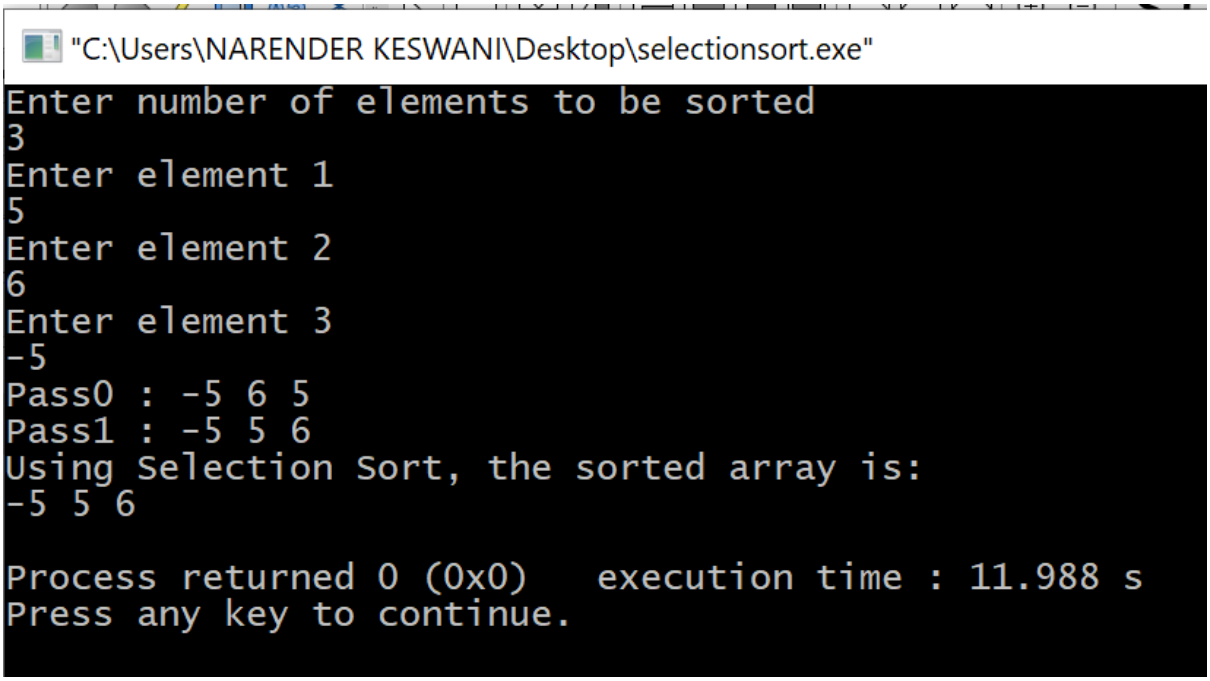
```
    /* swap the minimum element with the current element*/
    if indexMin != i  then
      swap list[min] and list[i]
    end if
  end for

end procedure
```

SOURCE CODE:

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

void printArray(int arr[], int n)
{
   int i;
   for (i=0; i < n; i++)
      cout << arr[i] << " ";
   cout << endl;
}

void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void selectionSort(int arr[], int n)
{
   int i, j, min_idx;
   // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
      {
         if (arr[j] < arr[min_idx])
            min_idx = j;
      }
      // Swap the found minimum element with the first element
      swap(arr[min_idx], arr[i]);
      cout << "Pass" <<i <<" : ";
      printArray(arr, n);
```

```
    }
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    selectionSort(arr, n);
    cout << "Using Selection Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;

}
```

OUTPUT:



D) SHELL SORT:

THEORY:

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. The interval between the elements is reduced based on the sequence used.

Worst Case Complexity: less than or equal to $O(n^2)$
Worst case complexity for shell sort is always less than or equal to $O(n^2)$.
Best Case Complexity: $O(n*\log n)$
Average Case Complexity: $O(n*\log n)$

Shell Sort Applications

Shell sort is used when:

calling a stack is overhead. uClibc library uses this sort.

recursion exceeds a limit. bzip2 compressor uses it.

Insertion sort does not perform well when the close elements are far apart. Shell sort helps in reducing the distance between the close elements. Thus, there will be less number of swappings to be performed.

ALGORITHM:

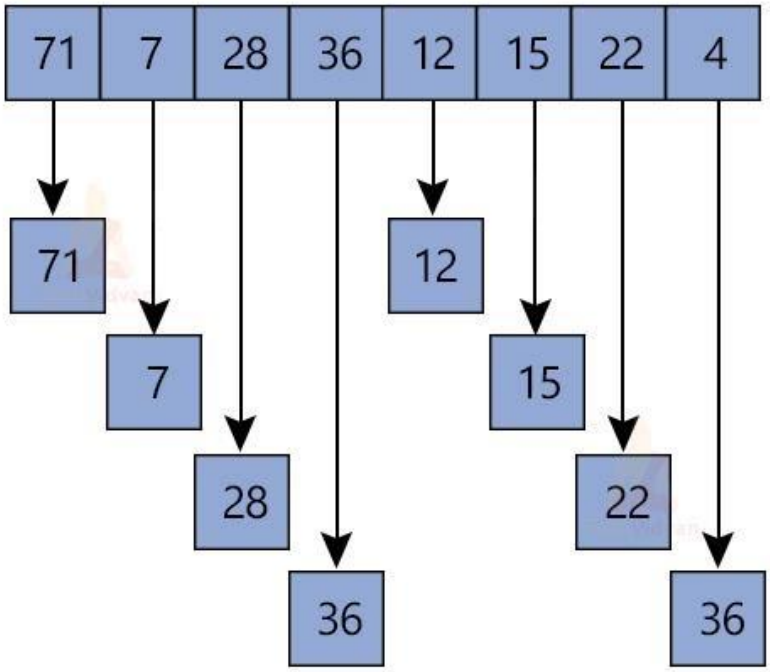
Following is the algorithm for shell sort.

Step 1 − Initialize the value of h
Step 2 − Divide the list into smaller sub-list of equal interval h
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted

Pseudocode

```
procedure shellSort()
  A : array of items

  /* calculate interval*/
  while interval < A.length /3 do:
    interval = interval * 3 + 1
  end while

  while interval > 0 do:

    for outer = interval; outer < A.length; outer ++ do:
```

```
    /* select value to be inserted */
    valueToInsert = A[outer]
    inner = outer;

      /*shift element towards right*/
      while inner > interval -1 && A[inner - interval] >= valueToInsert do:
        A[inner] = A[inner - interval]
        inner = inner - interval
      end while

    /* insert the number at hole position */
    A[inner] = valueToInsert

    end for

  /* calculate interval*/
  interval = (interval -1) /3;

  end while

end procedure
```



SOURCE CODE:

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

void printArray(int arr[], int n)
```

```
{
for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

int shellSort(int arr[], int n)
{

for (int gap = n/2; gap > 0; gap /= 2)
{
        for (int i = gap; i < n; i += 1)
        {

                int temp = arr[i];

                int j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                        arr[j] = arr[j - gap];

                arr[j] = temp;


        }

    cout << "Pass"<<" :";
        printArray(arr,n);
        cout << " \n";
}
return 0;
}

int main()
{
   int n;
   cout << "Enter number of elements to be sorted \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }

   shellSort(arr, n);
   cout << "Using Shell Sort, the sorted array is: \n";
   printArray(arr, n);
   return 0;
}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\shellsort.exe"
Enter number of elements to be sorted
5
Enter element 1
5
Enter element 2
6
Enter element 3
1
Enter element 4
3
Enter element 5
-5
Pass :-5 3 1 6 5
Pass :-5 1 3 5 6
Using Shell Sort, the sorted array is:
-5 1 3 5 6
Process returned 0 (0x0)   execution time : 10.120 s
Press any key to continue.
```

E) QUICK SORT:

THEORY:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

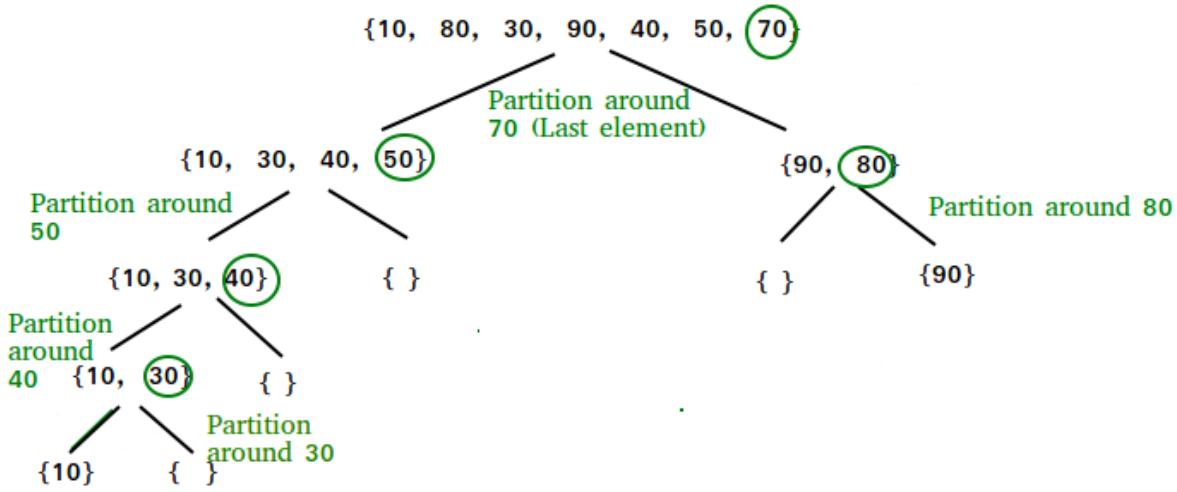
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
procedure quickSort(left, right)

  if right-left <= 0
    return
  else
```

```
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if

end procedure
```



Worst Case Complexity [Big-O]: $O(n^2)$
Best Case Complexity [Big-omega]: O(n*log n)
Average Case Complexity [Big-theta]: O(n*log n)
Space Complexity: O(log n)

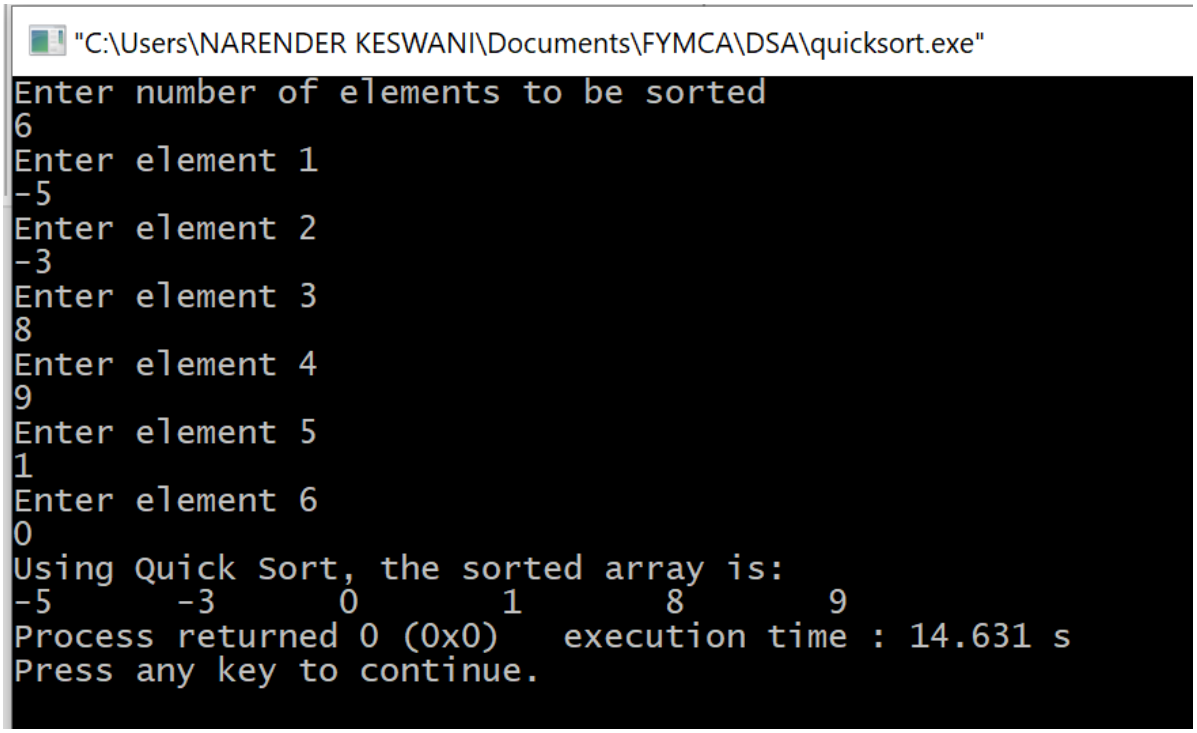
SOURCE CODE:

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

void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      cout<<arr[i]<<"\t";

}

// Swap two elements - Utility function
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
```

```
}

// partition the array using last element as pivot
int partition (int arr[], int low, int high)
{
   int pivot = arr[high];    // pivot
   int i = (low - 1);

   for (int j = low; j <= high- 1; j++)
   {
      //if current element is smaller than pivot, increment the low element
      //swap elements at i and j
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

//quicksort algorithm
void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      //partition the array
      int pivot = partition(arr, low, high);
      //sort the sub arrays independently
      quickSort(arr, low, pivot - 1);
      quickSort(arr, pivot + 1, high);
   }
}

int main()
{
   int n;
   cout << "Enter number of elements to be sorted \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }

   quickSort(arr, 0, n-1);
   cout << "Using Quick Sort, the sorted array is: \n";
   printArray(arr, n);
   return 0;
```

```
}
```

OUTPUT:



F) RADIX SORT:

THEORY:

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort

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 is implemented in places where there are numbers in large ranges.

Time Complexity	
Best	O(n+k)

Worst	O(n+k)
Average	O(n+k)
Space Complexity	O(max)

Example:
Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[*Notice that we keep 802 before 2, because 802 occurred

before 2 in the original list, and similarly for pairs

170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:
[*Notice that 802 again comes before 2 as 802 comes before
2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by the most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802

Algorithm:

radixSort(arr)

1. max = largest element in the given array
2. d = number of digits in the largest element (or, max)
3. Now, create d buckets of size 0 - 9
4. for i -> 0 to d
5. sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

SOURCE CODE:

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

// Print an array
void printArray(int arr[], int n)
{
```

```
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Function to get the largest element from an array
int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countingSort(int arr[], int n, int place)
{
    const int max = 10;
    int output[n];
    int count[max];

    for (int i = 0; i < max; ++i)
        count[i] = 0;

    // Calculate count of elements
    for (int i = 0; i < n; i++)
        count[(arr[i] / place) % 10]++;

    // Calculate cumulative count
    for (int i = 1; i < max; i++)
        count[i] += count[i - 1];


    // Place the elements in sorted order
    for (int i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i] / place) % 10] - 1] = arr[i];
        count[(arr[i] / place) % 10]--;
    }

    for (int i = 0; i < n; i++)
    {
        arr[i] = output[i];
        cout<<"Pass: ";
        printArray(arr, n);
    }
}

// Main function to implement radix sort
```

```
void radixsort(int arr[], int n)
{
   // Get maximum element
   int max = getMax(arr, n);
   cout<<"The max element is "<<max<<"\n";

   // Apply counting sort to sort elements based on place value.
   for (int place = 1; max / place > 0; place *= 10)
   {
      countingSort(arr, n, place);

   }
}

int main()
{
   int n;
   cout << "Enter number of elements to be sorted \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }

   radixsort(arr, n);
   cout << "Using Radix Sort, the sorted array is: \n";
   printArray(arr, n);
   return 0;
}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\radixsort.exe"
Enter number of elements to be sorted
5
Enter element 1
5
Enter element 2
4
Enter element 3
3
Enter element 4
6
Enter element 5
1
The max element is 6
Pass: 1 4 3 6 1
Pass: 1 3 3 6 1
Pass: 1 3 4 6 1
Pass: 1 3 4 5 1
Pass: 1 3 4 5 6
Using Radix Sort, the sorted array is:
1 3 4 5 6

Process returned 0 (0x0)   execution time : 7.516 s
Press any key to continue.
```

CONCLUSION:

I have learned the basic sorting algorithms such as quick, shell, bubble, insertion, radix, selection, etc

AIM: Implementation of searching algorithms: Linear Search and Binary search

1) Linear Search:

THEORY:

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found. Here is simple approach is to do Linear Search:

• Start from the leftmost element of array and one by one compare the element we are searching for with each element of the array.

• If there is a match between the element we are searching for and an element of the array, return the index.

• If there is no match between the element we are searching for and an element of the array, return -1.

Linear Search Complexities:

Time Complexity: O(n)

Space Complexity: O(1)

Linear Search Applications:

For searching operations in smaller arrays (<100 items).

Algorithm:

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode:

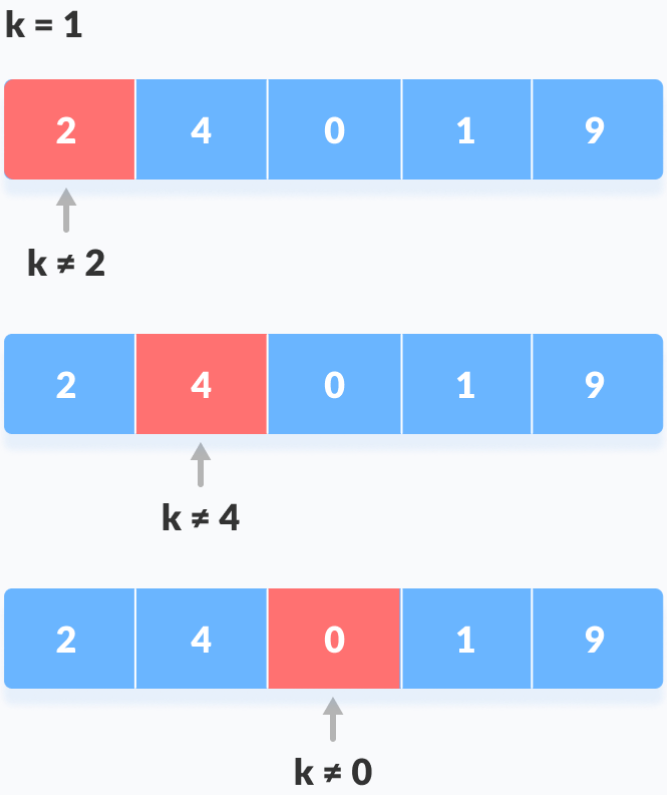
```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
   end for

end procedure
```

Example:

The following steps are followed to search for an element k = 1 in the list below.

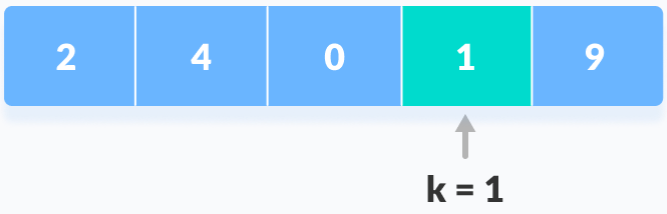
Array to be searched for

1. Start from the first element, compare k with each element x.



Compare with each element

2. If x == k, return the index.



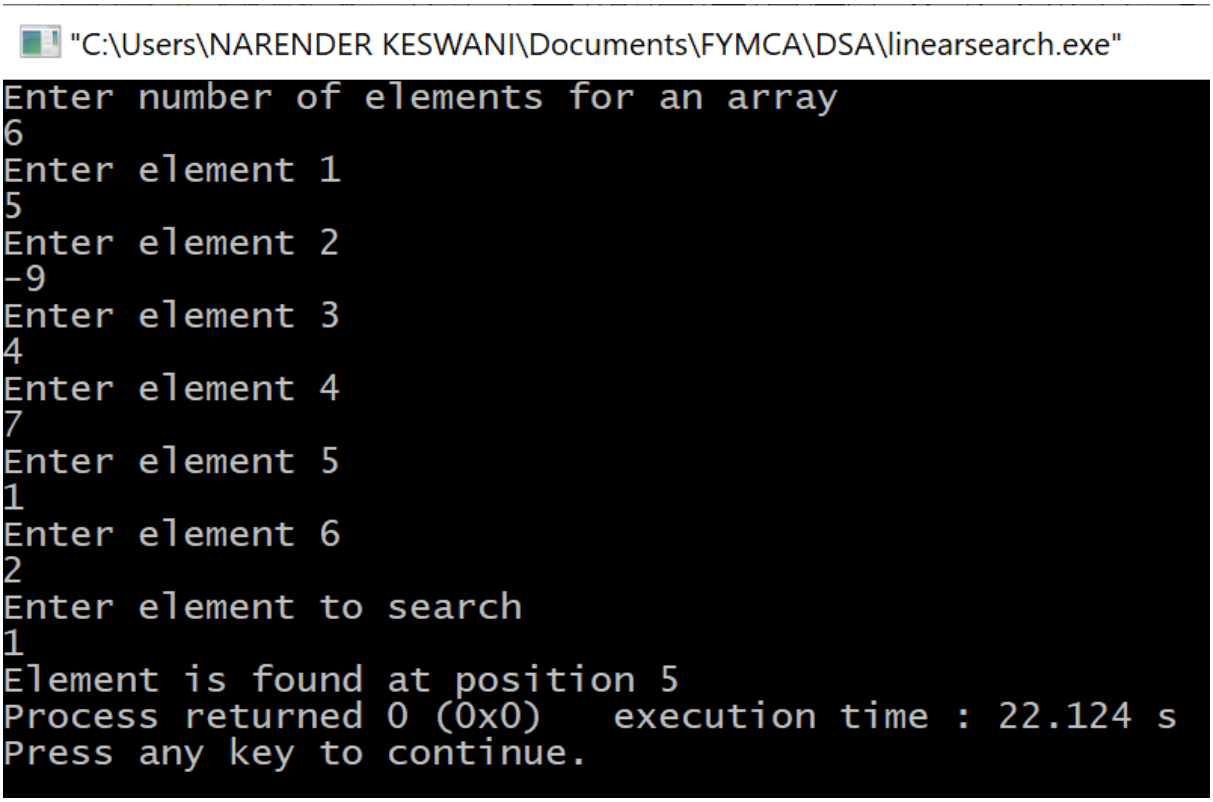
Element found

3. Else, return not found.

SOURCE CODE:

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

int linear_search( int arr[], int n, int s )
{
   for(int i=0; i < n; i++)
   {
      if(arr[i] == s)
      {
         return i;
      }
   }
   return -1;
}

int main()
{
   int n;
   cout << "Enter number of elements for an array \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }
   int s;
   cout<< "Enter element to search \n";
   cin>>s;
   int rs = linear_search(arr, n, s);
   if(rs==-1)
   {
      cout<<"Element is not found!!";
   }
   else
   {
      cout<<"Element is found at position "<<rs+1;
   }
   return 0;
}
```

OUTPUT:



2) **Binary Search:**

THEORY:

Binary Search is a searching algorithm for finding an element's position in a sorted array. The element is always searched in the middle of a portion of an array. Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first. Binary Search Algorithm can be implemented in two ways:

1. Iterative Method
2. Recursive Method

Time Complexities:

- **Best case complexity: O(1)**
- **Average case complexity: O(log n)**
- **Worst case complexity: O(log n)**

Space Complexity: O(1).

Binary Search Applications:

- **In libraries of Java, .Net, C++ STL, while debugging, the binary search is used to pinpoint the place where the error happens.**

Binary Search Algorithm:

1) Iteration Method:

```
do until the pointers low and high meet each other.
   mid = (low + high)/2
   if (x == arr[mid])
      return mid
   else if (x > arr[mid]) // x is on the right side
      low = mid + 1
   else                   // x is on the left side
      high = mid – 1
```

2) Recursive Method:

```
binarySearch(arr, x, low, high)
   if low > high
      return False
   else
      mid = (low + high) / 2
      if x == arr[mid]
         return mid
      else if x > arr[mid]       // x is on the right side
         return binarySearch(arr, x, mid + 1, high)
      else                       // x is on the right side
         return binarySearch(arr, x, low, mid - 1)
```

Example:

The array in which searching is to be performed is:

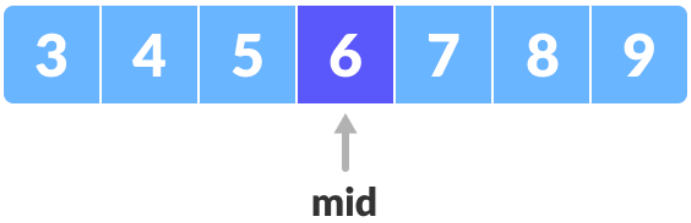
Initial array

Let x = 4 be the element to be searched.

Set two pointers low and high at the lowest and the highest positions respectively.

Setting pointers

Find the middle element mid of the array ie. arr[(low + high)/2] = 6.

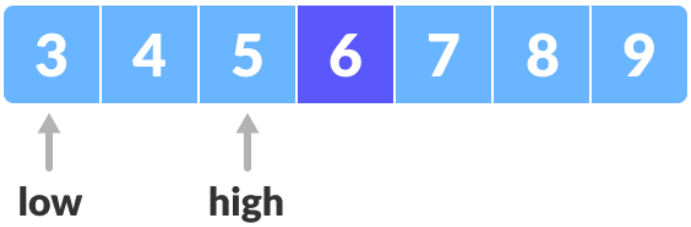


Mid element

If x == mid, then return mid.Else, compare the element to be searched with m.

If x > mid, compare x with the middle element of the elements on the right side of mid. This is done by setting low to low = mid + 1.

Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to high = mid - 1.

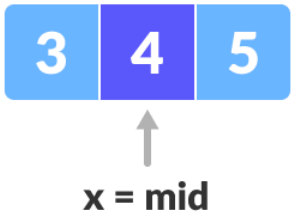


Finding mid element

Repeat steps 3 to 6 until low meets high.

Mid element

x = 4 is found.



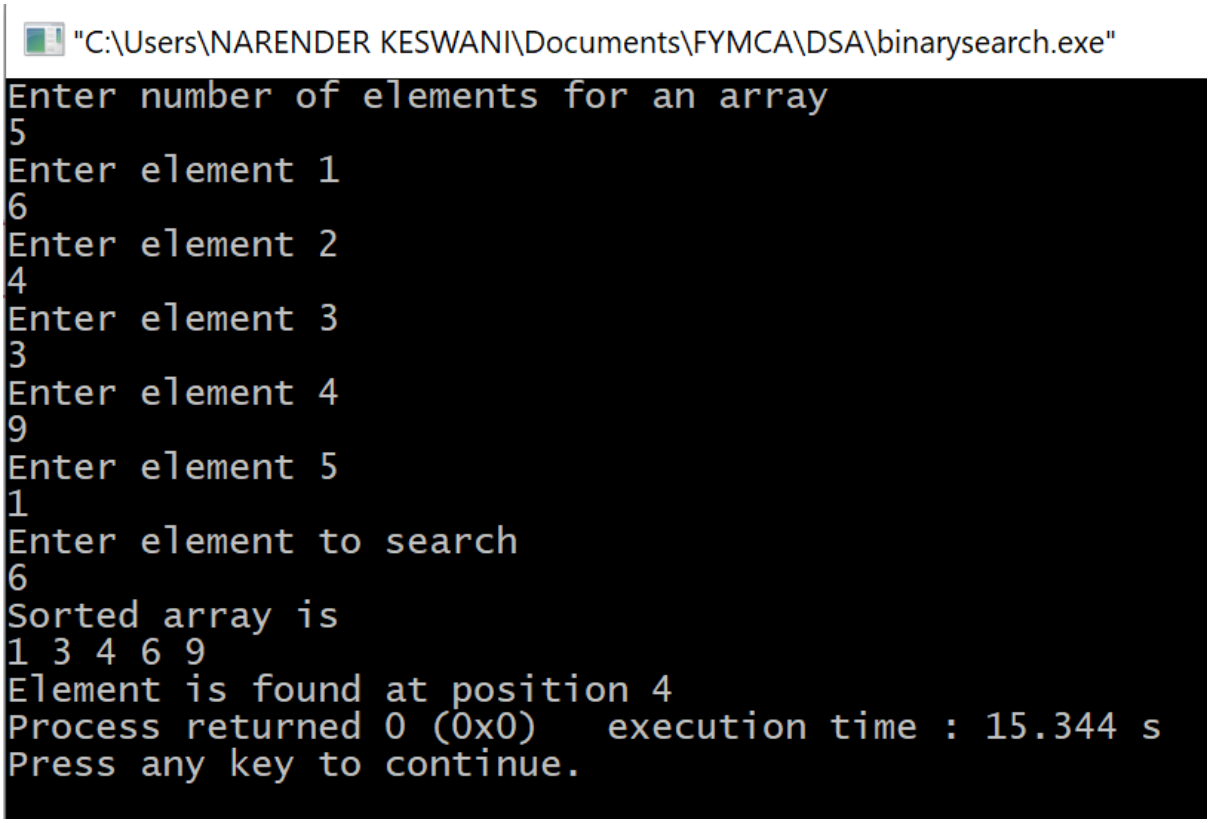
Founded

SOURCE CODE:

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

void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      cout << arr[i] << " ";
   cout << endl;
}

int binary_search( int arr[], int l, int r, int s )
{

   if (r>=1)
   {
      int mid = (l + r)/2;

      if(arr[mid] == s)
      {
         return mid;
      }
      else if(arr[mid] < s)
      {
         return binary_search(arr, mid + 1, r, s);
      }
      else
      {
      return binary_search(arr, l, mid - 1, s);
      }
   }
   else
      {
         return -1;
      }
}

int * bubble_sort( int A[ ], int n )
{
   int temp;
   for(int k = 0; k< n-1; k++)
   {
      // (n-k-1) is for ignoring comparisons of elements which have already been compared in
earlier iterations

      for(int i = 0; i < n-k-1; i++)
      {
         if(A[ i ] > A[ i+1] )
         {
            // here swapping of positions is being done.
```

```
                temp = A[ i ];
                A[ i ] = A[ i+1 ];
                A[ i + 1] = temp;
              }

          }
      }
           return A;
}

int main()
{
   int n;
   cout << "Enter number of elements for an array \n";
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      cout << "Enter element "<<i+1<<"\n";
      cin >> arr[i];
   }
   int s;
   cout<< "Enter element to search \n";
   cin>>s;
   int *ar = bubble_sort(arr,n);
   cout<< "Sorted array is \n";
   printArray(ar,n);
   int rs = binary_search(ar, 0, n-1, s);
   if(rs==-1)
   {
      cout<<"Element is not found!!";
   }
   else
   {
      cout<<"Element is found at position "<<rs+1;
   }
   return 0;

}
```

OUTPUT:



CONCLUSION:

I have learned the searching algorithms such as linear and binary. It is observed that binary search is more optimized than linear search.

AIM: Implement the following stack operations such as push, pop, display, count number of elements in stack, top using Array and Linked List using a Menu driven Format.

THEORY:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- Push :- Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop :- Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top :- Returns top element of stack.
- isEmpty :- Returns true if stack is empty, else false.
- Count: Returns number of elements in a stack.

There are two ways to implement a stack:

- **Using array:**

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. This implementation is very simple. We have to 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.

- **Using linked list:**

A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is the “head” of the stack where pushing and popping items happens at the head of the list. The first node has null in the link field and the second node link has the first node address in the link field and so on and the last node address in “top” pointer.

The main advantage of using linked lists over an array is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.

A) STACK USING ARRAY:

SOURCE CODE:

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

int stack[100],choice,n,top,x,i,val;

void push(int val)
{
   if(top>=n-1)
   {
      cout<<"Stack is Overflow"<<endl;
   }
   else
   {
      cout<<"Enter a value to be pushed:"<<endl;
      top++;
      stack[top] = val;
   }
}

void pop()
{
   if(top<=-1)
   {
      cout<<"Stack is Underflow"<<endl;
   }
   else
   {
      cout<<"The popped elements is"<<endl;
      cout<<stack[top]<<endl;
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      cout<<"The elements in STACK are"<<endl;
      for(i=top; i>=0; i--)
         cout<<stack[i]<<endl;
      cout<<"\n"<<endl;
      cout<<"Press Next Choice \n"<<endl;
   }
   else
   {
      cout<<"The STACK is empty \n"<<endl;
   }
}
```

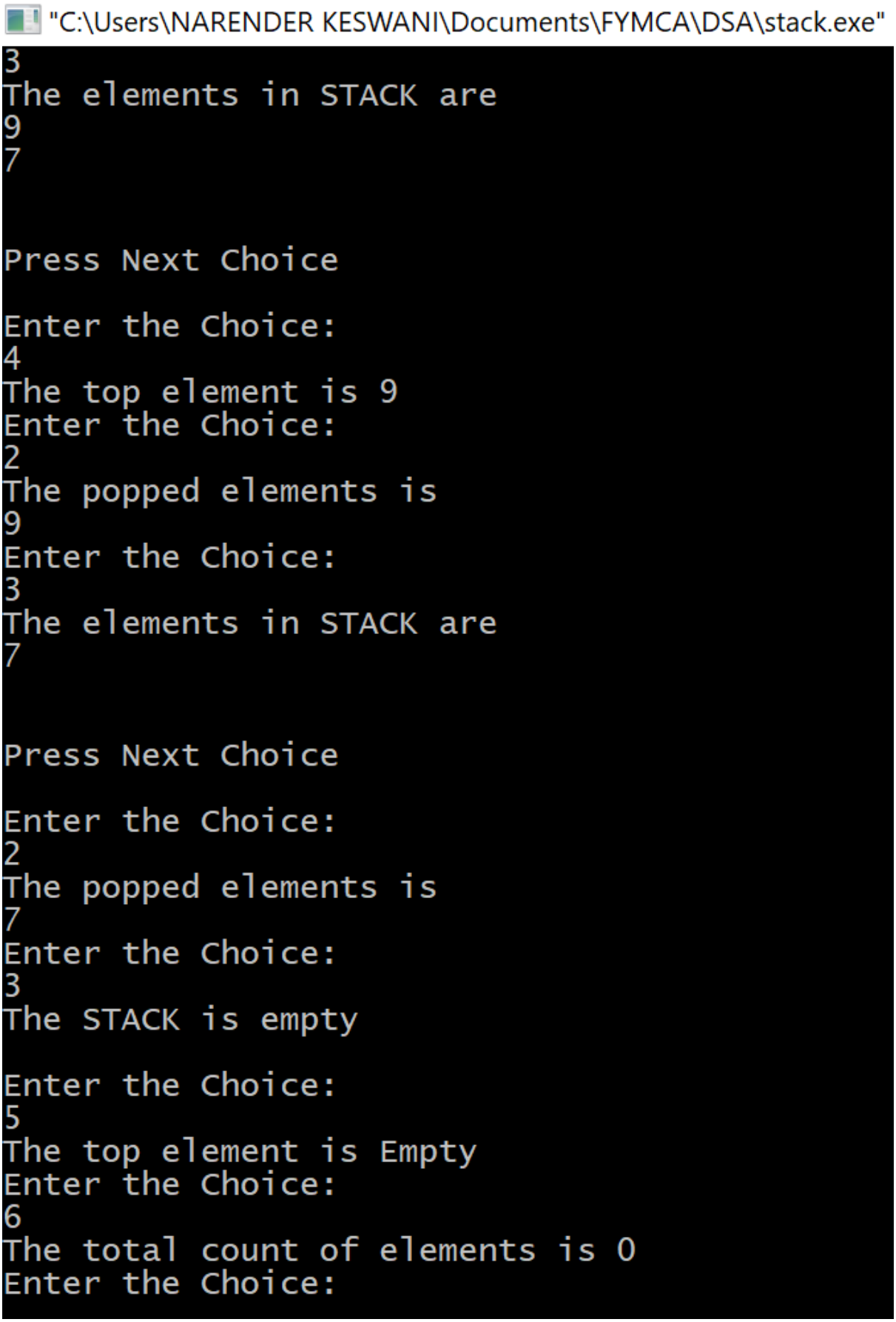
```
void Top()
{
   cout<<"The top element is "<<stack[top]<<"\n";
}

void IsEmpty()
{
   if(top == -1)
   {
      cout<<"The top element is Empty\n";
   }
   else
   {
      cout<<"The top element is NOT Empty\n";
   }
}

void countElements()
{
   cout<<"The total count of elements is "<<top+1<<"\n";
}

int main()
{
   top=-1;
   cout<<"Enter the size of STACK[MAX=100]:"<<endl;
   cin>>n;
   cout<<"STACK OPERATIONS USING ARRAY"<<endl;
   cout<<"\n----------------------------------------------"<<endl;
   cout<<"\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.TOP\n 5.IsEmpty\n 6.COUNT\n 7.EXIT"<<endl;
   do
   {
      cout<<"Enter the Choice:"<<endl;
      cin>>choice;
      switch(choice)
      {
      case 1:
      {
         cout<<"Enter value to be pushed:"<<endl;
         cin>>val;
         push(val);
         break;
      }
      case 2:
      {
         pop();
         break;
      }
      case 3:
      {
         display();
```

```
        break;
      }
      case 4:
      {
        Top();
        break;
      }
      case 5:
      {
        IsEmpty();
        break;
      }
      case 6:
      {
        countElements();
        break;
      }
      case 7:
      {
        cout<<"Exit \n"<<endl;
        break;
      }
      default:
      {
        cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4/5/6/7)"<<endl;
      }
      }
   }
   while(choice!=7);

   return 0;
}
```

OUTPUT:

"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\stack.exe"

```
Enter the size of STACK[MAX=100]:
5
STACK OPERATIONS USING ARRAY

----------------------------------------------

 1.PUSH
 2.POP
 3.DISPLAY
 4.TOP
 5.IsEmpty
 6.COUNT
 7.EXIT
Enter the Choice:
1
Enter value to be pushed:
7
Enter a value to be pushed:
Enter the Choice:
5
The top element is NOT Empty
Enter the Choice:
6
The total count of elements is 1
Enter the Choice:
3
The elements in STACK are
7


Press Next Choice

Enter the Choice:
1
Enter value to be pushed:
9
Enter a value to be pushed:
Enter the Choice:
```


"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\stack.exe"
3
The elements in STACK are
9
7
Press Next Choice
Enter the Choice:
4
The top element is 9
Enter the Choice:
2
The popped elements is
9
Enter the Choice:
3
The elements in STACK are
7
Press Next Choice
Enter the Choice:
2
The popped elements is
7
Enter the Choice:
3
The STACK is empty
Enter the Choice:
5
The top element is Empty
Enter the Choice:
6
The total count of elements is 0
Enter the Choice:

```
7
Exit


Process returned 0 (0x0)   execution time : 81.096 s
Press any key to continue.
```

B) STACK USING LINKED LIST:

SOURCE CODE:

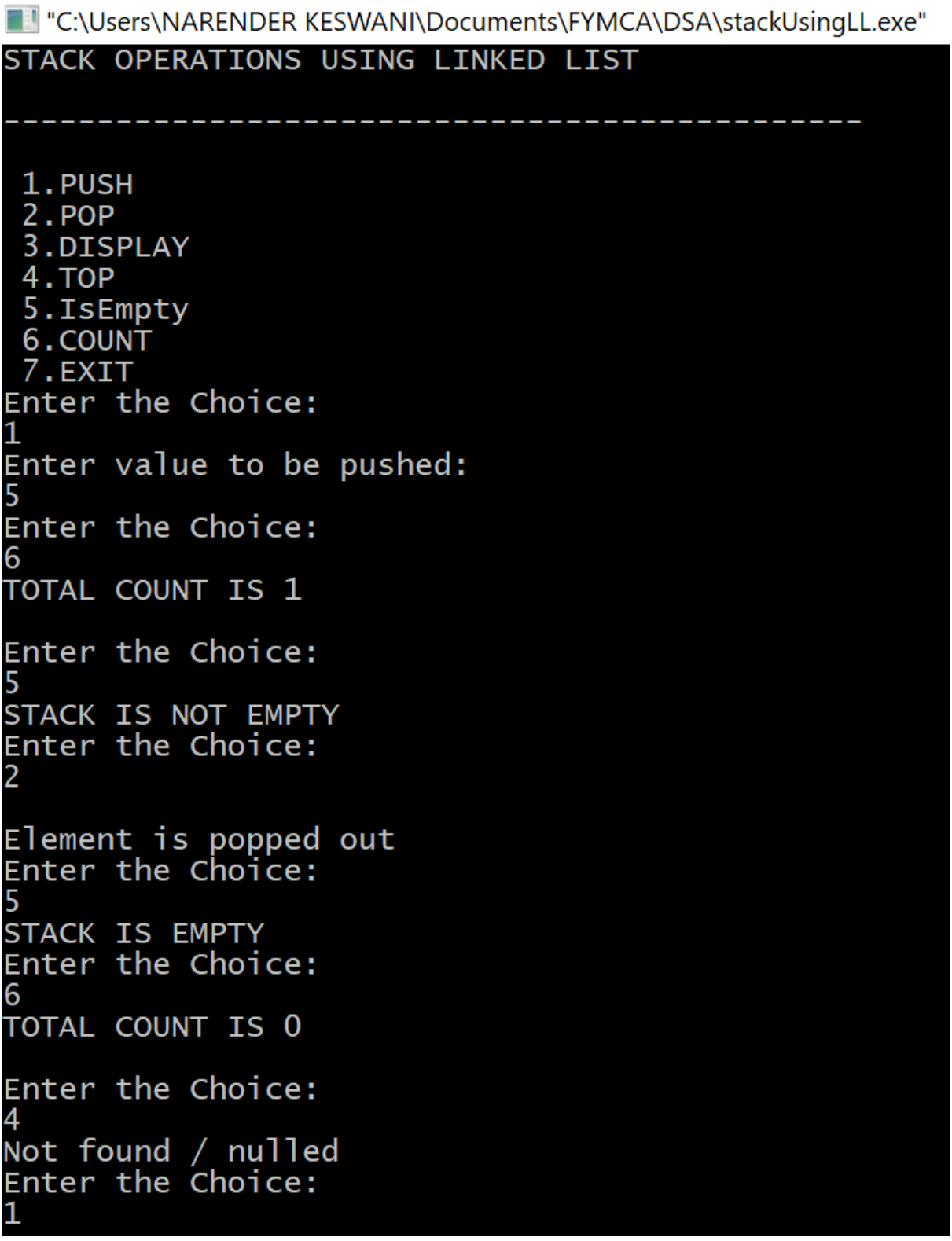
```
#include <bits/stdc++.h>
using namespace std;

struct Node
{
   int data;
   struct Node* link;
};

struct Node* top;

void push(int data)
{

   struct Node* temp;
   temp = new Node();

   if (!temp)
   {
      cout << "\nHeap Overflow";
      exit(1);
   }

   temp->data = data;
   temp->link = top;
   top = temp;
}

void IsEmpty()
{
   if(top == NULL)
   {
      cout<<"STACK IS EMPTY"<<endl;
   }
   else
   {
      cout<<"STACK IS NOT EMPTY"<<endl;
   }
}
```

```
int topElement()
{

   if (top != NULL)
   {
      cout<<"Top Element is "<<top->data<<" \n"<<endl;
   }
   else
   {
      cout<<"Not found / nulled"<<endl;
   }
}

void countElements()
{
   int count = 0;
   Node *temp = top;
   while(temp!=NULL)
   {
      count++;
      temp = temp->link;
   }
   cout<<"TOTAL COUNT IS "<<count<<"\n"<<endl;
}

void pop()
{
   struct Node* temp;

   if (top == NULL)
   {
      cout << "\nStack Underflow" << endl;
   }
   else
   {
      temp = top;
      top = top->link;
      temp->link = NULL;
      free(temp);
      cout << "\nElement is popped out" << endl;
   }
}

void display()
{
   struct Node* temp;

   if (top == NULL)
   {
      cout << "\nStack Underflow";
```

```
      exit(1);
   }
   else
   {
      temp = top;
      cout << "Elements are \n"<<endl;
      while (temp != NULL)
      {
         cout << temp->data<<"\n";
         temp = temp->link;
      }
   }
}

int main()
{
   int choice,val;
   cout<<"STACK OPERATIONS USING LINKED LIST"<<endl;
   cout<<"\n-----------------------------------------------"<<endl;
   cout<<"\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.TOP\n 5.IsEmpty\n 6.COUNT\n 7.EXIT"<<endl;
   do
   {
      cout<<"Enter the Choice:"<<endl;
      cin>>choice;
      switch(choice)
      {
      case 1:
      {
         cout<<"Enter value to be pushed:"<<endl;
         cin>>val;
         push(val);
         break;
      }
      case 2:
      {
         pop();
         break;
      }
      case 3:
      {
         display();
         break;
      }
      case 4:
      {
         topElement();
         break;
      }
      case 5:
      {
         IsEmpty();
```

```
            break;
        }
        case 6:
        {
            countElements();
            break;
        }
        case 7:
        {
            cout<<"Exit \n"<<endl;
            break;
        }
        default:
        {
            cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4/5/6/7)"<<endl;
        }
        }
    }
    while(choice!=7);
    return 0;
}
```

OUTPUT:


"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\stackUsingLL.exe"
STACK OPERATIONS USING LINKED LIST
1.PUSH
2.POP
3.DISPLAY
4.TOP
5.IsEmpty
6.COUNT
7.EXIT
Enter the Choice:
1
Enter value to be pushed:
5
Enter the Choice:
6
TOTAL COUNT IS 1
Enter the Choice:
5
STACK IS NOT EMPTY
Enter the Choice:
2
Element is popped out
Enter the Choice:
5
STACK IS EMPTY
Enter the Choice:
6
TOTAL COUNT IS 0
Enter the Choice:
4
Not found / nulled
Enter the Choice:
1

"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\stackUsingLL.exe"

```
Enter value to be pushed:
5
Enter the Choice:
1
Enter value to be pushed:
6
Enter the Choice:
4
Top Element is 6

Enter the Choice:
3
Elements are

6
5
Enter the Choice:
8
Invalid Input!, Please Enter a Valid Choice(1/2/3/4/5/6/7)
Enter the Choice:
7
Exit


Process returned 0 (0x0)   execution time : 55.781 s
Press any key to continue.
```

CONCLUSION:

I have learned about stacks and their implementation in arrays and linked lists. The main advantage of using linked lists over an array is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.

AIM: Implementation of Stack Applications like: Postfix evaluation & Balancing of Parenthesis

THEORY:

The stack is a linear data structure which follows the last in first out (LIFO) principle. Applications of stack:

1) Balancing of symbols.
2) Infix to postfix /Prefix conversion
3) Redo-undo features at many places like editors, photoshop.
4) Forward and backward feature in web browsers
5) Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
6) Backtracking is one of the algorithm designing technique .Some example of backtracking are Knight-Tour problem, N-Queen problem, find your way through maze and game like chess or checkers in all this problems we dive into someway if that way is not efficient we come back to the previous state and go into some other path. To get back from the current state we need to store the previous state for that purpose we need stack.
7) In Graph Algorithms like topological sorting and Strongly Connected Components.
8) In Memory management any modern computer uses stack as the primary-management for a running purpose. Each program that is running in a computer system has its own memory allocations

- **POSTFIX EVALUATION:**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Following is algorithm for evaluation postfix expressions.

1) Create a stack to store operands (or values).
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.
3) When the expression is ended, the number in the stack is the final answer.

- **BALANCING OF PARENTHESIS:**

Steps to find whether a given expression is balanced or unbalanced:

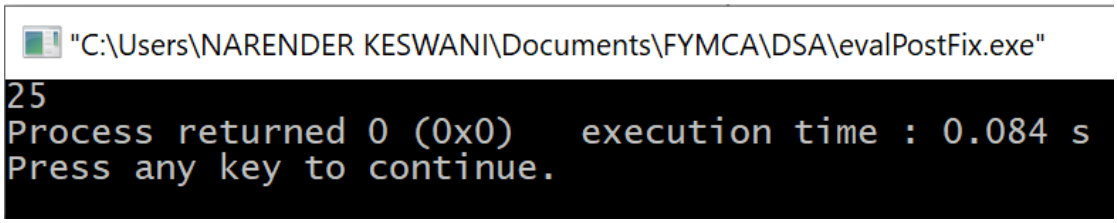
1) Input the expression and put it in a character stack.
2) Scan the characters from the expression one by one.
3) If the scanned character is a starting bracket (' (' or ' { ' or ' ['), then push it to the stack.
4) If the scanned character is a closing bracket (') ' or ' } ' or '] '), then pop from the stack and if the popped character is the equivalent starting bracket, then proceed. Else, the expression is unbalanced.
5) After scanning all the characters from the expression, if there is any parenthesis found in the stack or if the stack is not empty, then the expression is unbalanced.

A) POSTFIX EVALUATION:

SOURCE CODE:

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int evalPostfix(string exp)
{
   stack<int> stack;

   for (char c: exp)
   {
      if (c >= '0' && c <= '9') {
         stack.push(c - '0');
      }
      else {
         int x = stack.top();
         stack.pop();

         int y = stack.top();
         stack.pop();

         if (c == '+') {
            stack.push(y + x);
         }
         else if (c == '-') {
            stack.push(y - x);
         }
         else if (c == '*') {
            stack.push(y * x);
         }
         else if (c == '/') {
            stack.push(y / x);
         }
      }
   }

   return stack.top();
}

int main()
{
   string exp = "138*+";

   cout << evalPostfix(exp);

   return 0;
}
```

OUTPUT:



B) BALANCED PARENTHESIS:

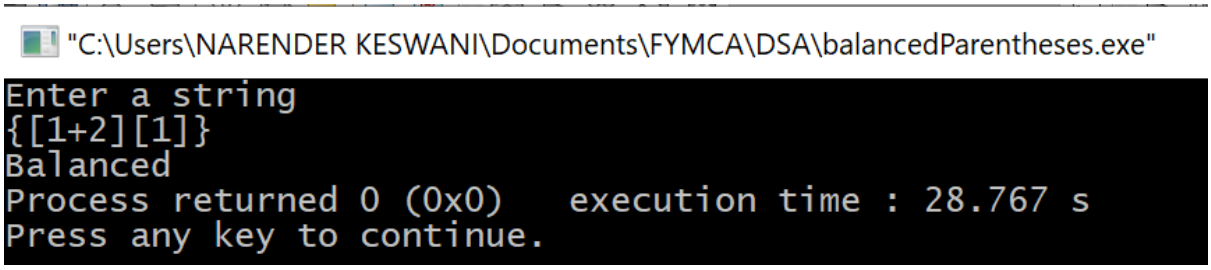
SOURCE CODE:

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

bool checkBalancing(string expr)
{
	stack<char> s;
	char x;

	for (int i = 0; i < expr.length(); i++)
	{
		if (expr[i] == '(' || expr[i] == '['
			|| expr[i] == '{')
		{
			s.push(expr[i]);
			continue;
		}

		if (s.empty())
			return false;

		switch (expr[i]) {
		case ')':

			x = s.top();
			s.pop();
			if (x == '{' || x == '[')
				return false;
			break;

		case '}':

			x = s.top();
			s.pop();
			if (x == '(' || x == '[')
				return false;
			break;

		case ']':
```

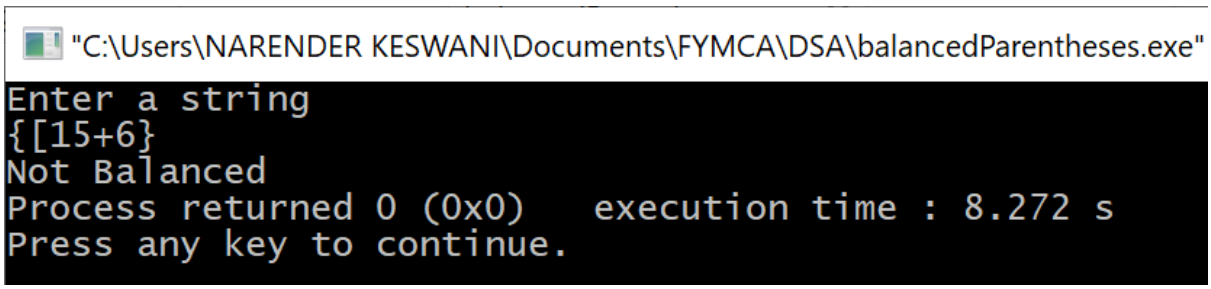
```
                        x = s.top();
                        s.pop();
                        if (x == '(' || x == '{')
                                return false;
                        break;
                }
        }

        return (s.empty());
}

int main()
{
        string expr;
        cout<<"Enter a string \n";
        cin>>expr;

        if (checkBalancing(expr))
                cout << "Balanced";
        else
                cout << "Not Balanced";
        return 0;
}
```

OUTPUT:

CASE-I BALANCED:



CASE-II UNBALANCED:



Queues

A) Simple Queue implementation using Linked List

THEORY:

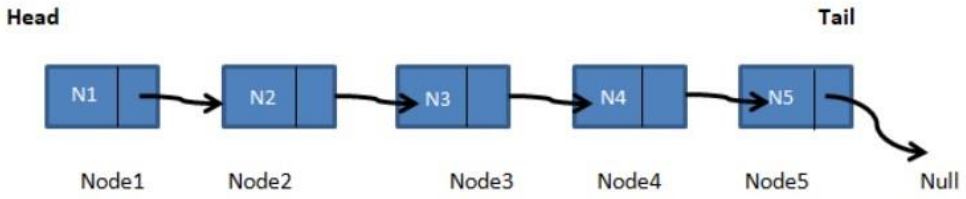
The array implementation cannot be used for the large-scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements 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 i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.



Algorithm :-

Insert operation :-

Step 1 :- Allocate the space for the new node PTR
Step 2 :- SET PTR -> DATA = VAL
Step 3 :- IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
Step 4 :- END

Deletion operation :-

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

SOURCE CODE:

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

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

struct node *front = NULL;
struct node *rear = NULL;
struct node *temp;

void Insert(int val)
{
   cout<<"Inserting \n"<<endl;

   if(rear==NULL)
   {
      rear = new node;
      rear -> next = NULL;
      rear -> data = val;
      front = rear;
   }
   else
   {
      temp = new node;
      rear -> next = temp;
      temp -> data = val;
      temp -> next = NULL;
      rear = temp;

   }
}
```

```
void Delete()
{
   temp = front;
   if(front == NULL)
   {
      cout<<"Underflow \n"<<endl;
      return;
   }
   else
   {
      if(temp->next != NULL)
      {
         temp = temp -> next;
         cout<<front->data<<endl;
         free(front);
         front = temp;
      }
      else
      {
         cout<<"Element deleted from queue is: "<<front->data<<"\n"<<endl;
         free(front);
         front = NULL;
         rear = NULL;

      }
   }
}

void Display()
{
   temp = front;
   if((front == NULL) && (rear == NULL))
   {
      cout<<"Queue is Empty \n"<<endl;
      return;
   }
   cout<<"Queue Elements are \n";
   while(temp!=NULL)
   {
      cout<<temp->data<<" ";
      temp = temp -> next;
      cout<<"\n";
   }
}

int main()
{
   int choice;
   int val;
```

```
    cout<<"QUEUE OPERATIONS USING LINKED LIST"<<endl;
    cout<<"\n-----------------------------------------------"<<endl;
    cout<<"\n 1.ENQUEUE\n 2.DEQUEUE\n 3.DISPLAY\n 4.EXIT"<<endl;
    do
    {
       cout<<"Enter the Choice: \n"<<endl;
       cin>>choice;
       switch(choice)
       {
       case 1:
       {
          cout<<"Enter value to be ENQUEUE: \n"<<endl;
          cin>>val;
          Insert(val);
          break;
       }
       case 2:
       {
          Delete();
          break;
       }
       case 3:
       {
          Display();
          break;
       }
       case 4:
       {
          cout<<"Exit \n"<<endl;
          break;
       }
       default:
       {
          cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4) \n"<<endl;
       }
       }
    }
    while(choice!=4);

    return 0;
}
```

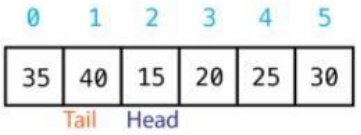
OUTPUT:

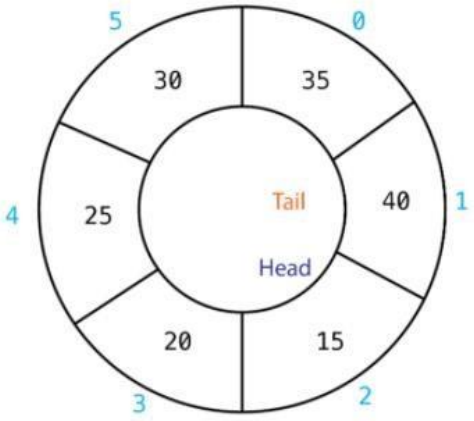
```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\queueUsingLL.exe"
QUEUE OPERATIONS USING LINKED LIST

------------------------------------------------

 1.ENQUEUE
 2.DEQUEUE
 3.DISPLAY
 4.EXIT
Enter the Choice:

3
Queue is Empty

Enter the Choice:

1
Enter value to be ENQUEUE:

3
Inserting

Enter the Choice:

3
Queue Elements are
3
Enter the Choice:

1
Enter value to be ENQUEUE:

5
Inserting

Enter the Choice:

3
Queue Elements are
3
5
Enter the Choice:

2
3
Enter the Choice:

3
Queue Elements are
5
Enter the Choice:

2
Element deleted from queue is: 5

Enter the Choice:

3
Queue is Empty

Enter the Choice:

2
Underflow

Enter the Choice:
```

B) Circular Queue implementation using Linked List

THEORY:

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.
A circular queue is similar to a linear queue as it is also 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. It is also known as a Ring Buffer.






```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
Q.Enqueue(40)
```

Algorithm :-

Insert operation :-

This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps :-

1. Create a new node dynamically and insert value into it.
2. Check if front==NULL, if it is true then front = rear = (newly created node)
3. If it is false then rear=(newly created node) and rear node always contains the address of the front node.

Delete operation :-

This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.

Steps :-

1. Check whether queue is empty or not means front == NULL.
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.

SOURCE CODE:

```
#include <iostream>
using namespace std;
struct Node
{
   int data;
   struct Node *next;
};
Node *front = NULL;
Node *rear = NULL;

void enqueue(int val)
{
   if(front==NULL || rear==NULL)
   {
      Node *newNode;
      newNode = new Node;

      newNode->data = val;
      newNode->next = NULL;

      front = newNode;
      rear = newNode;
   }
   else
   {
      Node *newNode;
      newNode = new Node;

      newNode->data = val;
      rear->next = newNode;


      newNode->next = front;
      rear = newNode;
   }

}

void dequeue()
{

   Node *n;
   n = front;
   if(front == NULL)
   {
      cout<<"Underflow \n"<<endl;
      return;
   }
   else
```

```
    {
      front = front->next;
      delete(n);
    }
}

void display()
{
   Node *ptr;
   ptr = front;
   if(ptr == NULL)
   {
      cout<<"Queue is Empty \n"<<endl;
      return;
   }
   else
   {
      do
      {
         cout<<"\n";
         cout<<ptr->data<<" ";
         ptr = ptr->next;
         cout<<"\n";
      }
      while(ptr != rear->next);
   }
}

int main()
{
   int choice;
   int val;
   cout<<"CIRCULAR QUEUE OPERATIONS USING LINKED LIST"<<endl;
   cout<<"\n-----------------------------------------------"<<endl;
   cout<<"\n 1.ENQUEUE\n 2.DEQUEUE\n 3.DISPLAY\n 4.EXIT"<<endl;
   do
   {
      cout<<"Enter the Choice: \n"<<endl;
      cin>>choice;
      switch(choice)
      {
      case 1:
      {
         cout<<"Enter value to be ENQUEUE: \n"<<endl;
         cin>>val;
         enqueue(val);
         break;
      }
      case 2:
      {
         dequeue();
```

```
            break;
          }
          case 3:
          {
            display();
            break;
          }
          case 4:
          {
            cout<<"Exit \n"<<endl;
            break;
          }
          default:
          {
            cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4) \n"<<endl;
          }
          }
        }
        while(choice!=4);

        return 0;
      }
```

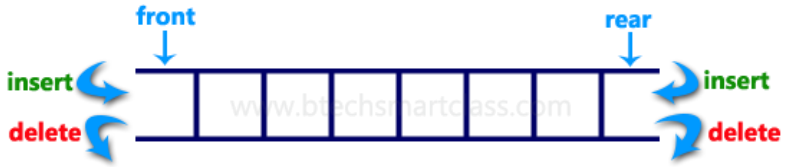
OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\circularQueueUsingLL.exe"
CIRCULAR QUEUE OPERATIONS USING LINKED LIST

-----------------------------------------------

 1.ENQUEUE
 2.DEQUEUE
 3.DISPLAY
 4.EXIT
Enter the Choice:

3
Queue is Empty

Enter the Choice:

1
Enter value to be ENQUEUE:

7
Enter the Choice:

1
Enter value to be ENQUEUE:

9
Enter the Choice:

3

7

9
Enter the Choice:

2
Enter the Choice:

3

9
Enter the Choice:
```

C) **Double ended Queue implementation using Linked List**

THEORY:

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



There are two variants of a **double-ended queue**. They include :-

 Input restricted deque :- In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

 Output restricted deque :- In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

Algorithm :-

Algorithm for Insertion at front end :-

Step-1 : [Check for the front position]
if(front<=1)
Print("Cannot add item at the front");
return;
Step-2 : [Insert at front]
else
front=front-1;
q[front]=no;
Step-3 : Return

Algorithm for Insertion at rear end :-

Step-1: [Check for overflow]
if(rear==MAX)
Print("Queue is Overflow");
return;
Step-2: [Insert Element]
else
rear=rear+1;
q[rear]=no;
[Set rear and front pointer]
if rear=0
rear=1;
if front=0
front=1;
Step-3: return

Algorithm for Deletion from front end:-

Step-1 [Check for front pointer]
if front=0
print(" Queue is Underflow");
return;
Step-2 [Perform deletion]
else
no=q[front];
print("Deleted element is",no);
[Set front and rear pointer]
if front=rear

front=0;
rear=0;
else
front=front+1;
Step-3 : Return

Algorithm for Deletion from rear end :-

Step-1 : [Check for the rear pointer]
if rear=0
print("Cannot delete value at rear end");
return;
Step-2: [perform deletion]
else
no=q[rear];
[Check for the front and rear pointer]
if front= rear

front=0;
rear=0;
else
rear=rear-1;
print("Deleted element is",no);
Step-3 : Return

SOURCE CODE:

```
#include <iostream>
using namespace std;
class Node
{
public:
   int data;
   Node *next;
   Node *prev;

   Node(int d)
   {
      data = d;
      next = NULL;
      prev = NULL;
   }
};

Node *newNode(int x)
{
   Node *node = new Node(x);
   return node;
}
Node *front = NULL;
Node *rear = NULL;
int Size = 0;
void insertFront(int x)
{
   Node *node = newNode(x);
   if (front == NULL)
   {
      front = rear = node;
   }
   else
   {
      node->next = front;
      front->prev = node;
      front = node;
   }
   Size++;
}
void insertEnd(int x)
{
   Node *node = newNode(x);
   if (rear == NULL)
   {
      front = rear = node;
   }
   else
```

```
    {
      node->prev = rear;
      rear->next = node;
      rear = node;
    }
    Size++;
}
void deleteFront()
{
    if (front == NULL)
    {
      cout << "DeQueue is empty" << endl;
      return;
    }
    if (front == rear)
    {
      front = rear = NULL;
    }
    else
    {
      Node *temp = front;
      front = front->next;
      front->prev = NULL;
      delete (temp);
    }
    Size--;
}
void deleteEnd()
{
    if (rear == NULL)
    {
      cout << "DeQueue is empty" << endl;
      return;
    }
    if (front == rear)
    {
      front = rear = NULL;
    }
    else
    {
      Node *temp = rear;
      rear = rear->prev;
      rear->next = NULL;
      delete (temp);
    }
    Size--;
}
int getFront()
{
    if (front != NULL)
```

```
    {
        return front->data;
    }
    return -1;
}
int getEnd()
{
    if (rear != NULL)
    {
        return rear->data;
    }
    return -1;
}
int size()
{
    return Size;
}
bool isEmpty()
{
    if (front == NULL)
    {
        return true;
    }
    return false;
}
void erase()
{
    rear = NULL;
    while (front != NULL)
    {
        Node *temp = front;
        front->prev = NULL;
        front = front->next;
        delete (temp);
    }
    Size = 0;
}
void display()
{

    Node *temp = front;
    if (front == NULL)
    {
        cout << "Queue is empty" << endl;
    }
    else
    {
        cout << "Queue is: \n";
        while (temp != NULL)
        {
```

```
            cout << temp->data << "  ";
            temp = temp->next;
        }
        cout << endl;
    }
}

int main()
{


    int choice;
    int val;
    int n;
    cout<<"QUEUE OPERATIONS USING DOUBLY LINKED LIST"<<endl;
    cout<<"\n-----------------------------------------------"<<endl;
    cout <<
"\n\n1.InsertFront\n2.InsertEnd\n3.DeleteFront\n4.DeleteEnd\n5.GetFront\n6.GetEnd\n7.Is
Empty\n8.SizeOfQueue\n9.Erase\n10.Display\n11.Exit";
    do
    {
        cout<<"\nEnter the Choice: \n"<<endl;
        cin>>choice;
        switch(choice)

        {
        case 1:
            cout << "\nEnter the element to be inserted at Front: ";
            cin >> n;
            insertFront(n);
            break;
        case 2:
            cout << "\nEnter the element to be inserted at End : ";
            cin >> n;
            insertEnd(n);
            break;
        case 3:
            cout << "\nDelete at Front  ";
            deleteFront();
            break;
        case 4:
            cout << "\nDelete at End ";
            deleteEnd();
            break;
        case 5:
            n = getFront();
            if (n == -1)
            {
                cout << "\nDeQueue is empty" << endl;
            }
```

```
            else
            {
               cout << "\nFront element is : " << getFront();
            }
            break;
         case 6:
            n = getFront();
            if (n == -1)
            {
               cout << "DeQueue is empty" << endl;
            }
            else
            {
               cout << "\nEnd element is : " << getEnd();
            }
            break;
         case 7:
            if (isEmpty())
            {
               cout << "\nDeQueue is empty";
            }
            else
            {
               cout << "\nDeQueue is not empty";
            }
            break;
         case 8:
            cout << "\nSize of DeQueue is : " << size();
            break;
         case 9:
            erase();
            break;
         case 10:
            display();
            break;
         case 11:
            exit(0);
            break;
         default:
            cout << "\nSelect Proper Option" << endl;
         }
      }

         while(choice!=11);

         return 0;
      }
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\queueUsingDLL.exe"
QUEUE OPERATIONS USING DOUBLY LINKED LIST

-----------------------------------------------


1.InsertFront
2.InsertEnd
3.DeleteFront
4.DeleteEnd
5.GetFront
6.GetEnd
7.IsEmpty
8.SizeOfQueue
9.Erase
10.Display
11.Exit
Enter the Choice:

3

Delete at Front  DeQueue is empty

Enter the Choice:

4

Delete at End DeQueue is empty

Enter the Choice:

5

DeQueue is empty

Enter the Choice:

6
DeQueue is empty

Enter the Choice:

7

DeQueue is empty
```

```
Enter the Choice:

8

Size of DeQueue is : 0
Enter the Choice:

10
Queue is empty

Enter the Choice:

1

Enter the element to be inserted at Front: 5

Enter the Choice:

2

Enter the element to be inserted at End : 10

Enter the Choice:

10
Queue is:
5  10

Enter the Choice:

1

Enter the element to be inserted at Front: 3

Enter the Choice:

5

Front element is : 3
Enter the Choice:

6

End element is : 10
```

```
Enter the Choice:

7

DeQueue is not empty
Enter the Choice:

8

Size of DeQueue is : 3
Enter the Choice:

10
Queue is:
3  5  10

Enter the Choice:

3

Delete at Front
Enter the Choice:

10
Queue is:
5  10

Enter the Choice:

4

Delete at End
Enter the Choice:

10
Queue is:
5

Enter the Choice:

9

Enter the Choice:

10
Queue is empty
```

CONCLUSION:

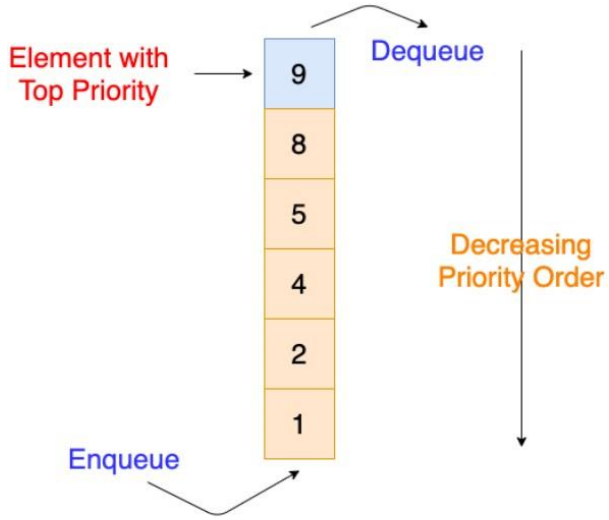
From this practical, I have learned about queue implementation using array, linked list & doubly linked list.

Implement Priority Queue Using Linked list

THEORY:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.



Difference between Priority Queue and Normal Queue is that, In a queue, the first-infirst-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

Algorithm :- Insert operation :-

1. IF((Front == 0)&&(Rear == N-1))
2. PRINT “Overflow Condition”
3. Else
4. IF(Front == -1)
5. Front = Rear =0
6. Queue[Rear] = Data
7. Priority[Rear] = Priority
8. ELSE IF(Rear ==N-1)
9. FOR i=Front;i<=Rear;i++)
10. FOR(i=Front;i<=Rear;i++)
11. Q[i-Front] =Q[i]
12. Pr[i-Front] = Pr[i]
13. Rear = Rear-Front
14. Front = 0
15. FOR(i = r;i>f;i–)
16. IF(p>Pr[i])
17. Q[i+1] = Q[i] Pr[i+1] = Pr[i]

18. ELSE
19. Q[i+1] = data Pr[i+1] = p
20. Rear++

Delete operation :- 1.
IF(Front == -1)
2. PRINT “Queue Under flow condition”
3. ELSE
4. PRINT”Q[f],Pr[f]”
5. IF(Front==Rear)
6. Front = Rear = -1
7. ELSE
8. FRONT++

SOURCE CODE:

```
#include <iostream>
using namespace std;
struct Node
{
   int data;
   int priority;
   struct Node *next;
};
Node *front = NULL;
Node *rear = NULL;

int Size = 0;
void insert(int val, int pr)
{
   Node *q;
   Node *temp = new Node;
   temp->data = val;
   temp->priority = pr;
   if (front == NULL || pr < front->priority)
   {
      temp->next = front;
      front = temp;
   }
   else
   {
      q = front;
      while (q->next != NULL && q->next->priority <= pr)
      {
         q = q->next;
      }
      temp->next = q->next;
      q->next = temp;
   }
```

```
   Size++;
}

void del()
{
   Node *temp;
   if (front == NULL)
   {
      cout << "Priority Queue is empty" << endl;
   }
   else
   {
      temp = front;
      cout << "Element deleted from priority queue is: " << temp->data << endl;
      front = front->next;
      delete temp;
      Size--;
   }
}

void display()
{
   Node *ptr;
   ptr = front;
   if (front == NULL)
   {
      cout << "Priority Queue is empty" << endl;
   }
   else
   {
      cout << "Priority Queue is: \n";
      cout << "Priority      Item\n";
      while (ptr != NULL)
      {
         cout << ptr->priority << "\t" << ptr->data << "\n";
         ptr = ptr->next;
      }
   }
}

int main()
{
 int choice, n, pr;
   int val;
   cout<<"PRIORITY QUEUE OPERATIONS"<<endl;
   cout<<"\n-----------------------------------------------"<<endl;
   cout<<"\n 1.INSERT\n 2.DELETE\n 3.DISPLAY\n 4.EXIT"<<endl;
   do
   {
      cout<<"Enter the Choice: \n"<<endl;
      cin>>choice;
```

```
    switch (choice)
    {
    case 1:
       cout << "Enter the value: ";
       cin >> n;
       cout << "Enter the priority: ";
       cin >> pr;
       insert(n, pr);
       break;
    case 2:
       del();
       break;
    case 3:
       display();
       break;
    case 4:
       exit(0);
    default:
        cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4) \n"<<endl;
    }
  }
  while(choice!=4);

  return 0;
}
```

OUTPUT:

```
Select "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\priorityQueue.exe"
PRIORITY QUEUE OPERATIONS

-----------------------------------------------

 1.INSERT
 2.DELETE
 3.DISPLAY
 4.EXIT
Enter the Choice:

3
Priority Queue is empty
Enter the Choice:

2
Priority Queue is empty
Enter the Choice:

1
Enter the value: 5
Enter the priority: 6
Enter the Choice:

1
Enter the value: 7
Enter the priority: 1
Enter the Choice:

3
Priority Queue is:
Priority        Item
1       7
6       5
Enter the Choice:

3
Priority Queue is:
Priority        Item
1       7
6       5
Enter the Choice:

2
Element deleted from priority queue is: 7

Enter the Choice:

3
Priority Queue is:
Priority        Item
6       5
```

CONCLUSION:

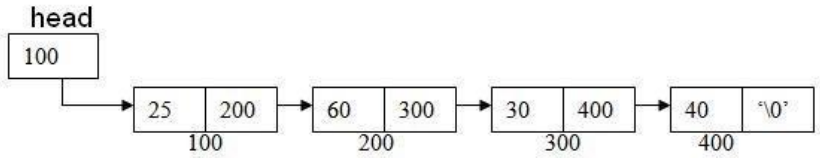
From this practical, I have learned from priority queue implementation.

AIM: **Implementation of all types of linked List Insert, Display, Delete, Search, Count, Reverse operation on Singly Linked Lists , Circular Linked List & Doubly Linked Lists**

THEORY:

A) Singly Linked Lists:

Single linked list is a sequence of elements in which every element has link to its next element in the sequence. In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field.



singly linked list with 4 nodes

Advantages over arrays:

1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So, we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL. Each node in a list consists of at least two parts:

1) data
2) Pointer (Or Reference) to the next node

In CPP, we can represent a node using structures. Below is an example of a linked list node with integer data.

In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

B) Circular Linked List:

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list.

Advantages over singly linked list:

1) A DLL can be traversed in both forward and backward direction.

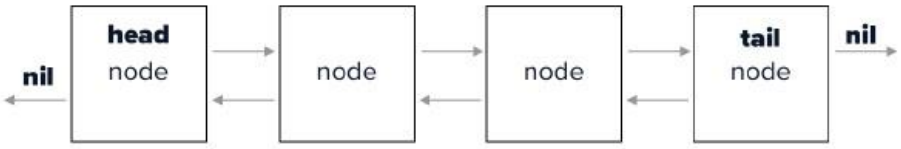
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

3) We can quickly insert a new node before a given node. In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list:

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though.

2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.



C) Doubly Linked Lists:

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element. That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

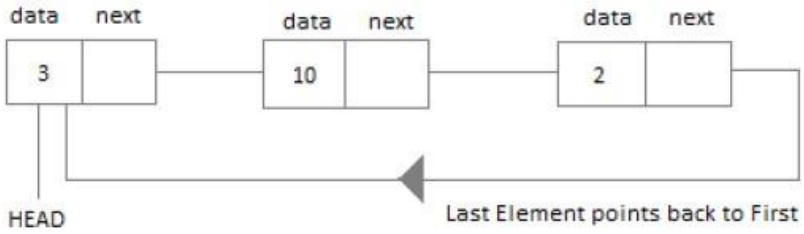
Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.



A) SINGLY LINKED LISTS:

SOURCE CODE:

```
#include<iostream>
using namespace std;
class node
{
public:
   int data;
   node* next;

   node(int val)
   {
      data=val;
      next=NULL;
   }
};

int nodeCount = 0;

void insertStart(node*&head, int val)
{
   nodeCount+=1;
   node* n= new node(val);
   n->next= head;
   head= n;
}

void insertEnd(node* &head, int val)
{
   node* n = new node(val);
   node* temp = head;

   nodeCount+=1;
   if(head==NULL)
   {
      head=n;
      return;
   }

   while(temp->next!=NULL)
   {

      temp=temp->next;
   }
   temp->next=n;
   return;

}

void insertInBetween(node* &head, int data, int n)
```

```
{
   node* temp1 = new node(data);
   temp1->next = NULL;
   if(n == 1)
   {
      nodeCount+=1;
      temp1->next = head;
      head = temp1;
      return;
   }

   node* temp2 = head;

   for(int i=0; i<n-2; i++)
   {
      temp2 = temp2->next;
   }

   nodeCount+=1;
   temp1->next = temp2->next;
   temp2->next = temp1;
}

void search(node* head, int data)
{
   node *curr = head;
   while(curr!= NULL)
   {
      if(curr->data == data)
      {
         cout<<"Found in List\n";
         return;
      }
      curr = curr->next;
   }
   cout<<"Not Found\n";
}

void deleteNode(node **head, int key)
{
   node *temp = *head;
   node *prev = NULL;
   nodeCount-=1;
   if(temp != NULL && temp->data == key)
   {
      *head = temp->next;
      delete temp;
      return;
   }
   else
   {
```

```
        while(temp != NULL && temp->data != key)
        {
           prev = temp;
           temp = temp->next;
        }
        if(temp == NULL)
        {
           return;
        }
        prev->next = temp->next;
        delete temp;
     }
}

void deleteFront(node* &head)
{
   if(head == NULL)
   {
      cout<<"Underflow"<<endl;
      return;
   }
   else
   {
      nodeCount-=1;
      node* temp = head;
      head = head->next;
      delete temp;
      return;
   }
}

void deleteEnd(node* &head)
{
   if (head == NULL)
   {
      cout<<"UnderFlow"<<endl;
      return;
   }

   else if (head->next == NULL)
   {
      nodeCount-=1;
      delete head;
      return;
   }
   else
   {
      node* temp = head;
      nodeCount-=1;
      while (temp->next->next != NULL)
         temp = temp->next;
```

```
        delete (temp->next);

        temp->next = NULL;
        return;
    }


}

void reverseList(node** head)
{
    node* prev = NULL;
    node* curr = *head;
    node* next = NULL;
    while(curr != NULL)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    *head = prev;
}

void display(node *head)
{
    if(head == NULL)
    {
        cout<<"EMPTY LINKED LIST"<<endl;
    }
    while(head != NULL)
    {
        cout<<head->data<<" -> ";
        head = head->next;
    }
    cout<<"\n";
}


int main()
{
    node *head = NULL;
    int choice, data;
    cout<<"SINGLY LINKED LIST"<<endl;
    cout<<"\n-----------------------------------------------"<<endl;
    cout<<"\n1.InsertFront \n2.InsertEnd \n3.Display \n4.Search \n5.Reverse
\n6.DeleteFront \n7.DeleteEnd \n8.Delete (by providing data) \n9.Count \n10.Insert
in between \n11.Exit"<<endl;
    do
    {
```

```
cout<<"\nEnter the Choice: \n"<<endl;
cin>>choice;
switch(choice)
{
case 1:
   cout<<"Enter data to append (start): ";
   cin>>data;
   insertStart(head,data);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 2:
   cout<<"Enter data to append (end): ";
   cin>>data;
   insertEnd(head,data);
   break;
case 3:
   display(head);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 4:
   cout<<"Enter data to Search : ";
   cin>>data;
   search(head,data);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 5:
   reverseList(&head);
   display(head);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 6:
   deleteFront(head);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 7:
   deleteEnd(head);
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 8:
   cout<<"Enter data to Delete :- ";
   cin>>data;
   deleteNode(&head,data);
   cout<<"\n----------------------------------------------"<<endl;
case 9:
   cout<<nodeCount<<"\n";
   cout<<"\n----------------------------------------------"<<endl;
   break;
case 10:
   int data,pos;
   cout<<"Enter data : ";
   cin>>data;
```

```
            cout<<"Enter position : ";
            cin>>pos;
            insertInBetween(head, data, pos);
            cout<<"\n---------------------------------------------"<<endl;
            break;
        case 11:
            exit(0);
            break;
        default:
            cout << "\nSelect Proper Option (1/2/3/4/5/6/7/8/9/10/11)" << endl;
        }
    }
    while(choice!=11);
    return 0;
}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\singlyLL.exe"
SINGLY LINKED LIST

---------------------------------------------

1.InsertFront
2.InsertEnd
3.Display
4.Search
5.Reverse
6.DeleteFront
7.DeleteEnd
8.Delete (by providing data)
9.Count
10.Insert in between
11.Exit
```

INSERT AT START:

```
Enter the Choice:

1
Enter data to append (start): 5

---------------------------------------------

Enter the Choice:

3
5 ->

---------------------------------------------
```

INSERT AT END:

```
Enter the Choice:

2
Enter data to append (end): 10

Enter the Choice:

3
5 -> 10 ->

------------------------------------------------
```

INSERT AT MIDDLE / SPECFIC POSITION:

```
Enter the Choice:

10
Enter data : 7
Enter position : 2

------------------------------------------------

Enter the Choice:

3
5 -> 7 -> 10 ->

------------------------------------------------
```

SEARCH:

```
Enter the Choice:

4
Enter data to Search : 7
Found in List

------------------------------------------------

Enter the Choice:

4
Enter data to Search : 8
Not Found

------------------------------------------------
```

REVERSE:

```
Enter the Choice:

5
10 -> 7 -> 5 ->

------------------------------------------------
```

DELETE FROM FRONT:

```
Enter the Choice:

6

------------------------------------------------

Enter the Choice:

3
7 -> 5 ->

------------------------------------------------
```

DELETE FROM END:

```
Enter the Choice:

7

------------------------------------------------

Enter the Choice:

3
7 ->

------------------------------------------------
```

DELETE SPECFIC DATA:

```
Enter the Choice:

8
Enter data to Delete :- 7

------------------------------------------------
0

------------------------------------------------

Enter the Choice:

3
EMPTY LINKED LIST


------------------------------------------------

Enter the Choice:
```

B) CIRCULAR LINKED LIST:

SOURCE CODE:

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

struct Node
{
  int data;
  Node* next;
};

class LinkedList
{
private:
  Node* head;
public:
  LinkedList()
  {
    head = NULL;
  }

  void InsertAtHead(int newElement)
  {
    Node* newNode = new Node();
    newNode->data = newElement;
    newNode->next = NULL;
    if(head == NULL)
    {
      head = newNode;
      newNode->next = head;
    }
    else
    {
      Node* temp = head;
      while(temp->next != head)
        temp = temp->next;
      temp->next = newNode;
      newNode->next = head;
      head = newNode;
    }
  }


  void InsertAtEnd(int newElement)
  {

    Node* newNode = new Node();

    newNode->data = newElement;

    newNode->next = NULL;
```

```
    if(head == NULL)
    {
       head = newNode;
       newNode->next = head;
    }
    else
    {

       Node* temp = head;
       while(temp->next != head)
          temp = temp->next;

       temp->next = newNode;
       newNode->next = head;
    }
}

void InsertAtPosition(int newElement, int position)
{

    Node* newNode = new Node();
    newNode->data = newElement;
    newNode->next = NULL;
    Node* temp = head;
    int NoOfElements = 0;

    if(temp != NULL)
    {
       NoOfElements++;
       temp = temp->next;
    }
    while(temp != head)
    {
       NoOfElements++;
       temp = temp->next;
    }

    if(position < 1 || position > (NoOfElements+1))
    {
       cout<<"\nInavalid position."<<endl;
    }
    else if (position == 1)
    {

       if(head == NULL)
       {
          head = newNode;
          head->next = head;
       }
       else
       {
          while(temp->next != head)
          {
             temp = temp->next;
```

```
            }
            newNode->next = head;
            head = newNode;
            temp->next = head;
         }
      }
      else
      {


         temp = head;
         for(int i = 1; i < position-1; i++)
            temp = temp->next;
         newNode->next = temp->next;
         temp->next = newNode;
      }
   }

   void display()
   {
      Node* temp = head;
      if(temp != NULL)
      {
         cout<<"The list contains: ";
         while(true)
         {
            cout<<temp->data<<" ";
            temp = temp->next;
            if(temp == head)
               break;
         }
         cout<<endl;
      }
      else
      {
         cout<<"The list is empty.\n";
      }
   }


   void deleteFirst()
   {
      if(head != NULL)
      {


         if(head->next == head)
         {
            head = NULL;
         }
         else
         {
```

```
            Node* temp = head;
            Node* firstNode = head;

            while(temp->next != head)
            {
               temp = temp->next;
            }


            head = head->next;
            temp->next = head;
            free(firstNode);
         }
      }
   }


   void deleteEnd()
   {
      if(head != NULL)
      {

         if(head->next == head)
         {
            head = NULL;
         }
         else
         {

            Node* temp = head;
            while(temp->next->next != head)
               temp = temp->next;


            Node* lastNode = temp->next;
            temp->next = head;
            free(lastNode);
         }
      }
   }

   void DeleteAtPosition(int position)
   {


      Node* nodeToDelete = head;
      Node* temp = head;
      int NoOfElements = 0;

      if(temp != NULL)
      {
         NoOfElements++;
         temp = temp->next;
      }
```

```
    while(temp != head)
    {
       NoOfElements++;
       temp = temp->next;
    }


    if(position < 1 || position > NoOfElements)
    {
       cout<<"\nInavalid position.";
    }
    else if (position == 1)
    {


       if(head->next == head)
       {
          head = NULL;
       }
       else
       {
          while(temp->next != head)
             temp = temp->next;
          head = head->next;
          temp->next = head;
          free(nodeToDelete);
       }
    }
    else
    {

       temp = head;
       for(int i = 1; i < position-1; i++)
          temp = temp->next;
       nodeToDelete = temp->next;
       temp->next = temp->next->next;
       free(nodeToDelete);
    }
}

int countNodes()
{

    Node* temp = head;

    int i = 0;


    if(temp != NULL)
    {
       i++;
       temp = temp->next;
    }
    while(temp != head)
```

```
    {
        i++;
        temp = temp->next;
    }

    return i;
}

void SearchElement(int searchValue)
{

    Node* temp = head;
    int found = 0;
    int i = 0;


    if(temp != NULL)
    {
        while(true)
        {
            i++;
            if(temp->data == searchValue)
            {
                found++;
                break;
            }
            temp = temp->next;
            if(temp == head)
            {
                break;
            }
        }
        if (found == 1)
        {
            cout<<searchValue<<" is found at index = "<<i<<".\n";
        }
        else
        {
            cout<<searchValue<<" is not found in the list.\n";
        }
    }
    else
    {


        cout<<"The list is empty.\n";
    }
}

void reverseList()
{

    if(head != NULL)
    {
```

```
            Node* prevNode = head;
            Node* tempNode = head;
            Node* curNode = head->next;


            prevNode->next = prevNode;

            while(curNode != head)
            {

               tempNode = curNode->next;
               curNode->next = prevNode;
               head->next = curNode;
               prevNode = curNode;
               curNode = tempNode;
            }

            head = prevNode;
         }
      }

};


int main()
{
   int choice, data, location;
   LinkedList cll;

   cout<<"CIRCULAR LINKED LIST"<<endl;
   cout<<"\n-----------------------------------------------"<<endl;
   cout<<"\n1.InsertFront\n2.InsertEnd\n3.InsertAtPosition\n4.Delete First\n5.Delete
End\n6.DeleteAtPosition\n7.Count\n8.Search\n9.Reverse\n10.Display\n11.Exit\n"<<endl;
   do
   {
      cout<<"\nEnter the Choice: \n"<<endl;
      cin>>choice;
      switch(choice)
      {
      case 1:
         cout<<"Enter data to append :- (start)";
         cin>>data;
         cll.InsertAtHead(data);
         cll.display();
         cout<<"\n-----------------------------------------------"<<endl;
         break;
      case 2:
         cout<<"Enter data to append :- (end)";
         cin>>data;
         cll.InsertAtEnd(data);
         cll.display();
         cout<<"\n-----------------------------------------------"<<endl;
         break;
      case 3:
```

```
            cout << "Enter data to be inserted: ";
            cin >> data;
            cout << "Enter location to be inserted into: ";
            cin >> location;
            cll.InsertAtPosition(data,location);
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 4:
            cll.deleteFirst();
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 5:
            cll.deleteEnd();
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 6:
            cout << "Enter location to be inserted into: ";
            cin >> location;
            cll.DeleteAtPosition(location);
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 7:
            cout<<cll.countNodes()<<endl;
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 8:
            cout<<"Enter data to Search :- ";
            cin>>data;
            cll.SearchElement(data);
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 9:
            cll.reverseList();
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 10:
            cll.display();
            cout<<"\n----------------------------------------------"<<endl;
            break;
         case 11:
            exit(1);
            break;
         default:
            cout << "\nSelect Proper Option (1/2/3/4/5/6/7/8/9/10/11)" << endl;
         }
      }
      while(choice!=11);
      return 0;
   }
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\circularLL.exe"
CIRCULAR LINKED LIST

-----------------------------------------

1.InsertFront
2.InsertEnd
3.InsertAtPosition
4.Delete First
5.Delete End
6.DeleteAtPosition
7.Count
8.Search
9.Reverse
10.Display
11.Exit
```

NULL VALIDATION:

```
Enter the Choice:

7
0

-----------------------------------------------

Enter the Choice:

10
The list is empty.

-----------------------------------------------

Enter the Choice:

4
The list is empty.

-----------------------------------------------

Enter the Choice:

5
The list is empty.

-----------------------------------------------
```

```
Enter the Choice:

6
Enter location to be inserted into: 5

Inavalid position.The list is empty.

-----------------------------------------------
```

INSERT AT START:

```
Enter the Choice:

1
Enter data to append :- (start)5
The list contains: 5

-----------------------------------------------
```

INSERT AT END:

```
Enter the Choice:

2
Enter data to append :- (end)10
The list contains: 5 10

-----------------------------------------------
```

INSERT AT POSITION:

```
Enter the Choice:

3
Enter data to be inserted: 2
Enter location to be inserted into: 2
The list contains: 5 2 10

-----------------------------------------------
```

COUNT:

```
Enter the Choice:

7
3

------------------------------------------------
```

SEARCH:

```
Enter the Choice:

8
Enter data to Search :- 55
55 is not found in the list.

------------------------------------------------

Enter the Choice:

8
Enter data to Search :- 5
5 is found at index = 1.

------------------------------------------------
```

PRINT:

```
10
The list contains: 5 2 10

------------------------------------------------
```

REVERSE:

```
Enter the Choice:

9
The list contains: 10 2 5

------------------------------------------------
```

DELETE FRONT:

```
Enter the Choice:

4
The list contains: 2 5

------------------------------------------------
```

DELETE END:

```
Enter the Choice:

5
The list contains: 2

------------------------------------------------
```

DELETE AT POINT:

```
Enter the Choice:

1
Enter data to append :- (start)6
The list contains: 6 2

------------------------------------------------

Enter the Choice:

6
Enter location to be inserted into: 2
The list contains: 6

------------------------------------------------

Enter the Choice:

10
The list contains: 6

------------------------------------------------
```

C) **DOUBLY LINKED LIST:**

SOURCE CODE:

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;

struct node
{
   int value;
   struct node* next;
   struct node* prev;
};
struct node* head;
struct node* tail;

void init()
{
   head=NULL;
   tail=NULL;
}

void insertFirst(int element)
{
   struct node* newItem;
   newItem=new node;
   if(head==NULL)
   {
      head=newItem;
      newItem->prev=NULL;
      newItem->value=element;
      newItem->next=NULL;
      tail=newItem;
   }
   else
   {
      newItem->next=head;
      newItem->value=element;
      newItem->prev=NULL;
      head->prev=newItem;
      head=newItem;
   }
}

void insertLast(int element)
{
   struct node* newItem;
   newItem=new node;
   newItem->value=element;
   if(head==NULL)
   {
      head=newItem;
```

```
      newItem->prev=NULL;
      newItem->next=NULL;
      tail=newItem;
    }
    else
    {
      newItem->prev=tail;
      tail->next=newItem;
      newItem->next=NULL;
      tail=newItem;
    }
}

void insertAfter(int old, int element)
{
    struct node* newItem;
    newItem=new node;
    struct node* temp;
    temp=head;
    if(head==NULL)
    {
      return;
    }
    if(head==tail)
    {
      if(head->value!=old)
      {
        return;
      }
      newItem->value=element;
      head->next=newItem;
      newItem->next=NULL;
      head->prev=NULL;
      newItem->prev=head;
      tail=newItem;
      return;
    }
    if(tail->value==element)
    {
      newItem->next=NULL;
      newItem->prev=tail;
      tail->next=newItem;
      tail=newItem;
      return;
    }
    while(temp->value!=old)
    {
      temp=temp->next;
      if(temp==NULL)
      {
        cout<<"Could not insert"<<endl;
        cout<<"Element not found"<<endl;
        return;
      }
```

```
    }

    newItem->next=temp->next;
    newItem->prev=temp;
    newItem->value=element;
    temp->next->prev=newItem;
    temp->next=newItem;
}

void deleteFirst()
{
    if(head==NULL)
    {
        return;
    }
    if(head==tail)
    {
        struct node* cur;
        cur=head;
        head=NULL;
        tail=NULL;
        delete cur;
        return;
    }
    else
    {
        struct node* cur;
        cur=head;
        head=head->next;
        head->prev=NULL;
        delete cur;
    }
}

void deleteLast()
{
    if(head==NULL) return;
    if(head==tail)
    {
        struct node* cur;
        cur=head;
        head=NULL;
        tail=NULL;
        delete cur;
        return;
    }
    else
    {
        struct node* cur;
        cur=tail;
        tail=tail->prev;
        tail->next=NULL;
        delete cur;
    }
```

```
}
void deleteItem(int element)
{
   struct node* temp;
   temp=head;
   if(head==tail)
   {
      if(head->value!=element)
      {
         cout<<"Could not delete"<<endl;
         return;
      }
      head=NULL;
      tail=NULL;
      delete temp;
      return;
   }
   if(head->value==element)
   {
      head=head->next;
      head->prev=NULL;
      delete temp;
      return;
   }
   else if(tail->value==element)
   {
      temp=tail;
      tail=tail->prev;
      tail->next=NULL;
      delete temp;
      return;
   }
   while(temp->value!=element)
   {
      temp=temp->next;
      if(temp==NULL)
      {
         cout<<"Element not found"<<endl;
         return;
      }
   }
   temp->next->prev=temp->prev;
   temp->prev->next=temp->next;
   delete temp;
}

struct node* searchItem(int element)
{
   struct node* temp;
   temp=head;
   while(temp!=NULL)
   {
      if(temp->value==element)
      {
```

```
            return temp;
            break;
        }
        temp=temp->next;
    }
    return NULL;
}

void printList()
{
    struct node* temp;
    temp=head;
    while(temp!=NULL)
    {
        cout<<temp->value<<"->";
        temp=temp->next;
    }
    puts("");
}

void printReverse()
{
    struct node* temp;
    temp=tail;
    while(temp!=NULL)
    {
        cout<<temp->value<<"->";
        temp=temp->prev;
    }
    cout<<endl;
}

void makereverse()
{
    struct node* prv=NULL;
    struct node* cur=head;
    struct node* nxt;
    while(cur!=NULL)
    {
        nxt=cur->next;
        cur->next=prv;
        prv=cur;
        cur=nxt;
    }
    head=prv;
}

int countNodes()
{
    struct node* temp=head;
    int i = 0;
    while(temp != NULL)
    {
        i++;
```

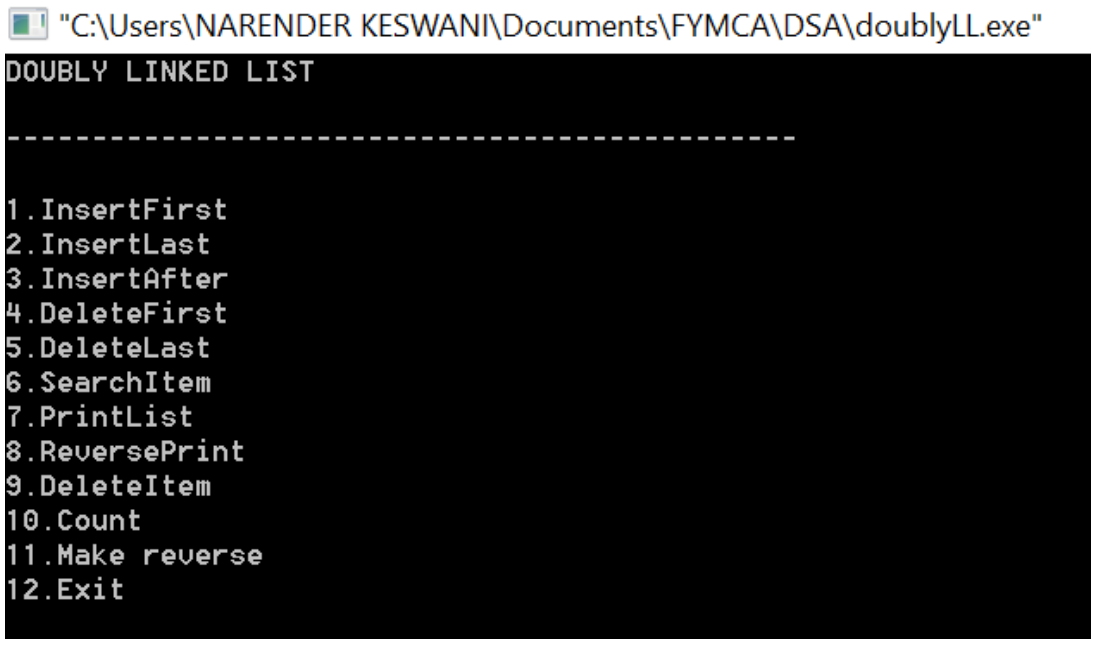
```
      temp = temp->next;
    }
    return i;
}

int dltfrst()
{
    if(head==NULL)
    {
      return 0;
    }
    int prev;
    prev=head->value;
    if(head==tail)
    {
      struct node* cur;
      cur=head;
      head=NULL;
      tail=NULL;
      delete cur;
      return prev;
    }
    else
    {
      struct node* cur;
      cur=head;
      head=head->next;
      head->prev=NULL;
      delete cur;
      return prev;
    }
}
int dltlast()
{
    if(head==NULL) return 0;
    int prev;
    prev=tail->value;
    if(head==tail)
    {
      struct node* cur;
      cur=head;
      head=NULL;
      tail=NULL;
      delete cur;
      return prev;
    }
    else
    {
      struct node* cur;
      cur=tail;
      tail=tail->prev;
      tail->next=NULL;
      delete cur;
      return prev;
```

```
    }
}

int main()
{
    init();
    int choice;

    cout<<"DOUBLY LINKED LIST"<<endl;
    cout<<"\n-----------------------------------------------"<<endl;

cout<<"\n1.InsertFirst\n2.InsertLast\n3.InsertAfter\n4.DeleteFirst\n5.DeleteLast\n6.Sear
chItem\n7.PrintList\n8.ReversePrint\n9.DeleteItem\n10.Count\n11.Make
reverse\n12.Exit\n"<<endl;
    do
    {
        cout<<"\nEnter the Choice: \n"<<endl;
        cin>>choice;
        switch(choice)
        {
        case 1:
            {
            int elementStart;
            cout<<"Enter data to append :- (start)";
            cin>>elementStart;
            insertFirst(elementStart);
            printList();
            cout<<"\n-----------------------------------------------"<<endl;
            break;
            }
        case 2:
            {
            int elementEnd;
            cout<<"Enter data to append :- (end)";
            cin>>elementEnd;
            insertLast(elementEnd);
            printList();
            cout<<"\n-----------------------------------------------"<<endl;
            break;
            }
        case 3:
            {
            int old,newitem;
            cout << "Enter data to be inserted: ";
            cout<<"Enter Old Item_";
            cin>>old;
            cout<<"Enter new Item_";
            cin>>newitem;
            insertAfter(old,newitem);
            printList();
            cout<<"\n-----------------------------------------------"<<endl;
            break;
            }
        case 4:
```

```
    {
    deleteFirst();
    printList();
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 5:
    {
    deleteLast();
    printList();
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 6:
    {
    int item;
    cout<<"Enter Item to Search";
    cin>>item;
    struct node* ans=searchItem(item);
    if(ans!=NULL)
    {
       cout<<"FOUND "<<ans->value<<endl;
    }
    else
    {
       cout<<"NOT FOUND"<<endl;
    }
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 7:
    {
    printList();
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 8:
    {
    printReverse();
    printList();
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 9:
    {
    int element;
    cout<<"Enter element to delete "<<endl;
    cin>>element;
    deleteItem(element);
    printList();
    cout<<"\n-----------------------------------------------"<<endl;
    break;
    }
  case 10:
```

```
            {
            cout<<countNodes()<<endl;
            cout<<"\n-----------------------------------------------"<<endl;
            break;
            }
        case 11:
            {
            makereverse();
            printList();
            cout<<"\n-----------------------------------------------"<<endl;
            break;
            }
        case 12:
            {
            exit(1);
            break;
            }
        default:
            cout << "\nSelect Proper Option (1/2/3/4/5/6/7/8/9/10/11/12)" << endl;
        }
    }
    while(choice!=12);
    return 0;
}
```

OUTPUT:



INSERT FIRST:

```
Enter the Choice:

1
Enter data to append :- (start)5
5->

------------------------------------------------
```

INSERT LAST:

```
Enter the Choice:

2
Enter data to append :- (end)10
5->10->

------------------------------------------------
```

INSERT AFTER SPECFIC VALUE:

```
Enter the Choice:

3
Enter data to be inserted: Enter Old Item_5
Enter new Item_7
5->7->10->

------------------------------------------------
```

SEARCH:

```
Enter the Choice:

6
Enter Item to Search11
NOT FOUND

------------------------------------------------

Enter the Choice:

6
Enter Item to Search7
FOUND 7

------------------------------------------------
```

PRINT / DISPLAY:

```
Enter the Choice:

7
5->7->10->

------------------------------------------------
```

COUNT:

```
Enter the Choice:

10
3

------------------------------------------------
```

DELETE FIRST:

```
Enter the Choice:

4
7->10->

------------------------------------------------
```

DELETE LAST:

```
Enter the Choice:

5
7->

------------------------------------------------
```

DELETE SPECFIC VALUE:

```
Enter the Choice:

9
Enter element to delete
7


------------------------------------------------
```

INSERT & REVERSE:

```
Enter the Choice:

1
Enter data to append :- (start)1
1->

-----------------------------------------------

Enter the Choice:

1
Enter data to append :- (start)2
2->1->

-----------------------------------------------

Enter the Choice:

1
Enter data to append :- (start)3
3->2->1->

-----------------------------------------------

Enter the Choice:

7
3->2->1->

-----------------------------------------------

Enter the Choice:

8
1->2->3->
3->2->1->

-----------------------------------------------

Enter the Choice:

11
1->2->3->

-----------------------------------------------
```

CONCLUSION:

From this practical, I have learned how to implement singly, doubly, circular linked list.

AIM: IMPLEMENT BINARY SEARCH TREE INSERTION AND TRAVERSAL, OPERATION ON BST, LARGEST NODE, SMALLEST NODE, COUNT NUMBER OF NODES

THEORY:

A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree. A binary tree is a type of data structure for storing data such as numbers in an organized way. Binary search trees allow binary search for fast lookup, addition and removal of data items, and can be used to implement dynamic sets and lookup tables. The order of nodes in a BST means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value), and each has two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search property: the key in each node is greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another.

Attributes of Binary Search Tree:

A BST is made of multiple nodes and consists of the following attributes:

- Nodes of the tree are represented in a parent-child relationship
- Each parent node can have zero child nodes or a maximum of two sub-nodes or subtrees on the left and right sides.
- Every sub-tree, also known as a binary search tree, has sub-branches on the right and left of themselves.
- All the nodes are linked with key-value pairs.
- The keys of the nodes present on the left subtree are smaller than the keys of their parent node
- Similarly, the left subtree nodes' keys have lesser values than their parent node's keys.

Operations on Binary Search Tree:

BST primarily offers the following three types of operations for your usage:

- Search: searches the element from the binary tree
- Insert: adds an element to the binary tree
- Delete: delete the element from a binary tree

Each operation has its own structure and method of execution/analysis, but the most complex of all is the Delete operation.

Search Operation:

Always initiate analyzing tree at the root node and then move further to either the right or left subtree of the root node depending upon the element to be located is either less or greater than the root.

Insert Operation:

This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it is lesser than the root, it is added to the left subtree.

Delete Operations:

Delete is the most advanced and complex among all other operations. There are multiple cases handled for deletion in the BST.

- Case 1- Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.
- Case 2 - Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.
- Case 3 Node with two children: this is the most difficult situation, and it works on the following two rules
  - 3a - In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
  - 3b - In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node

SOURCE CODE:

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

using namespace std;

struct treeNode
{
   int data;
   treeNode *left;
   treeNode *right;
};
treeNode* FindMin(treeNode *node)
{
   if(node==NULL)
   {
      return NULL;
   }
   if(node->left)
      return FindMin(node->left);
   else
      return node;
}
treeNode* FindMax(treeNode *node)
{
   if(node==NULL)
   {
      return NULL;
   }
   if(node->right)
      return(FindMax(node->right));
   else
      return node;
}
treeNode *Insert(treeNode *node,int data)
{
   if(node==NULL)
   {
      treeNode *temp;
      temp=new treeNode;
```

```
      temp -> data = data;
      temp -> left = temp -> right = NULL;
      return temp;
   }
   if(data >(node->data))
   {
      node->right = Insert(node->right,data);
   }
   else if(data < (node->data))
   {
      node->left = Insert(node->left,data);
   }
   return node;
}
treeNode * Delet(treeNode *node, int data)
{
   treeNode *temp;
   if(node==NULL)
   {
      cout<<"Element Not Found";
   }
   else if(data < node->data)
   {
      node->left = Delet(node->left, data);
   }
   else if(data > node->data)
   {
      node->right = Delet(node->right, data);
   }
   else
   {

      if(node->right && node->left)
      {
         temp = FindMin(node->right);
         node -> data = temp->data;
         node -> right = Delet(node->right,temp->data);
      }
      else
      {
```

```
            temp = node;
            if(node->left == NULL)
               node = node->right;
            else if(node->right == NULL)
               node = node->left;
            free(temp);
         }
      }
      return node;
}
treeNode * Find(treeNode *node, int data)
{
   if(node==NULL)
   {
      return NULL;
   }
   if(data > node->data)
   {
      return Find(node->right,data);
   }
   else if(data < node->data)
   {
      return Find(node->left,data);
   }
   else
   {
      return node;
   }
}
void Inorder(treeNode *node)
{
   if(node==NULL)
   {
      return;
   }
   Inorder(node->left);
   cout<<node->data<<" ";
   Inorder(node->right);
}
void Preorder(treeNode *node)
{
```

```
   if(node==NULL)
   {
      return;
   }
   cout<<node->data<<" ";
   Preorder(node->left);
   Preorder(node->right);
}
void Postorder(treeNode *node)
{
   if(node==NULL)
   {
      return;
   }
   Postorder(node->left);
   Postorder(node->right);
   cout<<node->data<<" ";
}

int countNodes(treeNode *node)
{
   if(node==0)
      return 0;
   else
      return(countNodes(node->left)+countNodes(node->right)+1);
}
int main()
{
   treeNode *root = NULL,*temp;
   int ch;

   cout<<"BINARY TREE OPERATIONS"<<endl;
   cout<<"\n-----------------------------------------------"<<endl;

cout<<"\n1.Insert\n2.Delete\n3.Inorder\n4.Preorder\n5.Postorder\n6.FindMin\n7.FindMax
\n8.Search\n9.Count\n10.Exit\n";
   do
   {
      cout<<"\nEnter the Choice: \n"<<endl;
      cin>>ch;
      switch(ch)
```

```
{
case 1:
   cout<<"\nEnter element to be insert:";
   cin>>ch;
   root = Insert(root, ch);
   cout<<"\nElements in BST are:";
   Inorder(root);
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 2:
   cout<<"\nEnter element to be deleted:";
   cin>>ch;
   root = Delet(root,ch);
   cout<<"\nAfter deletion elements in BST are:";
   Inorder(root);
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 3:
   cout<<"\nInorder Travesals is:";
   Inorder(root);
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 4:
   cout<<"\nPreorder Traversals is:";
   Preorder(root);
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 5:
   cout<<"\nPostorder Traversals is:";
   Postorder(root);
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 6:
   temp = FindMin(root);
   cout<<"\nMinimum element is :"<<temp->data;
   cout<<"\n-----------------------------------------------"<<endl;
   break;
case 7:
   temp = FindMax(root);
   cout<<"\nMaximum element is :"<<temp->data;
   cout<<"\n-----------------------------------------------"<<endl;
```

```
            break;
        case 8:
            cout<<"\nEnter element to be searched:";
            cin>>ch;
            temp = Find(root,ch);
            if(temp==NULL)
            {
               cout<<"Element is not found"<<endl;
            }
            else
            {
               cout<<"Element "<<temp->data<<" is Found\n";
            }
            cout<<"\n-----------------------------------------"<<endl;
            break;
        case 9:
            cout<<countNodes(root)<<endl;
            cout<<"\n-----------------------------------------"<<endl;
            break;
        case 10:
             exit(0);
            break;
        default:
            cout << "\nSelect Proper Option (1/2/3/4/5/6/7/8/9/10)" << endl;
            break;
        }
    }
    while(ch!=10);
    return 0;
}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\BST.exe"
BINARY TREE OPERATIONS

-----------------------------------------------

1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.FindMin
7.FindMax
8.Search
9.Count
10.Exit
```

INSERT:

```
Enter the Choice:

1

Enter element to be insert:6

Elements in BST are:6
-----------------------------------------------

Enter the Choice:

1

Enter element to be insert:5

Elements in BST are:5 6
-----------------------------------------------

Enter the Choice:

1

Enter element to be insert:9

Elements in BST are:5 6 9
-----------------------------------------------
```

FIND MIN:

```
Enter the Choice:

6

Minimum element is :5
-----------------------------------------------
```

FIND MAX:

```
Enter the Choice:

7

Maximum element is :9
-----------------------------------------------
```

SEARCH:

```
Enter the Choice:

8

Enter element to be searched:9
Element 9 is Found

-----------------------------------------------

Enter the Choice:

8

Enter element to be searched:111
Element is not found

-----------------------------------------------
```

COUNT:

```
Enter the Choice:

9
3

------------------------------------------------
```

INORDER:

```
Enter the Choice:

3

Inorder Travesals is:5 6 9
------------------------------------------------
```

PREORDER:

```
Enter the Choice:

4

Preorder Traversals is:6 5 9
------------------------------------------------
```

POSTORDER:

```
Enter the Choice:

5

Postorder Traversals is:5 9 6
------------------------------------------------
```

DELETE:

```
Enter the Choice:

2

Enter element to be deleted:6

After deletion elements in BST are:5 9
-----------------------------------------

Enter the Choice:

2

Enter element to be deleted:10
Element Not Found
After deletion elements in BST are:5 9
-----------------------------------------
```

CONCLUSION:

From this practical, I have learned about binary trees and their operations.

AIM: FIND THE MINIMUM SPANNING TREE(MST) USING

I) **KRUSKAL'S ALGORITHM**
II) **PRIM'S ALGORITHM**

THEORY:

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

There are many use cases for minimum spanning trees. One example is a telecommunications company trying to lay cable in a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

A) Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

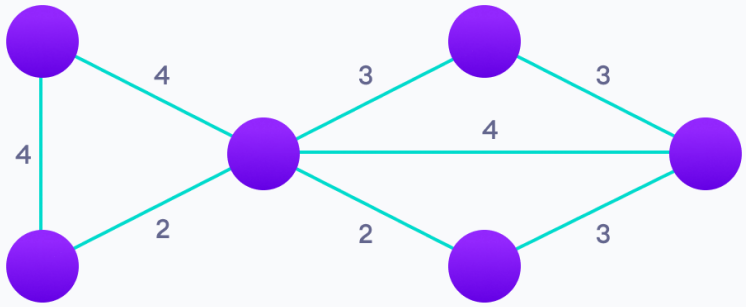
We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high

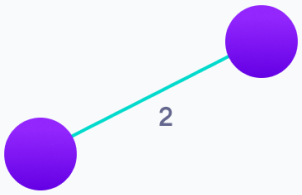
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



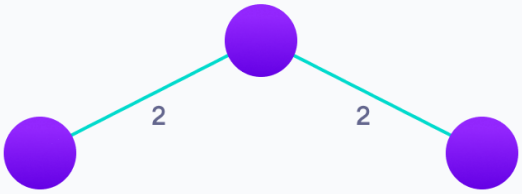
Step: 1

Start with a weighted graph



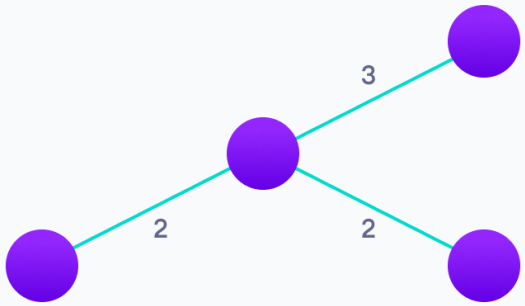
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



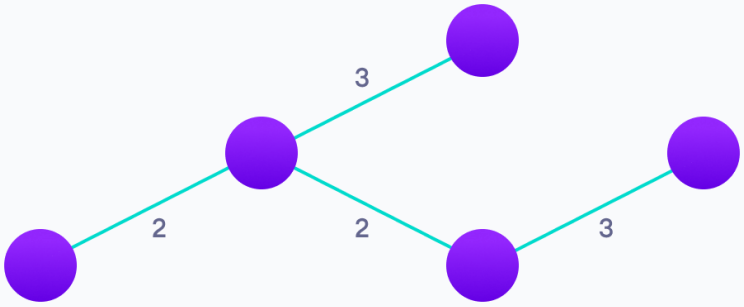
Step: 3

Choose the next shortest edge and add it



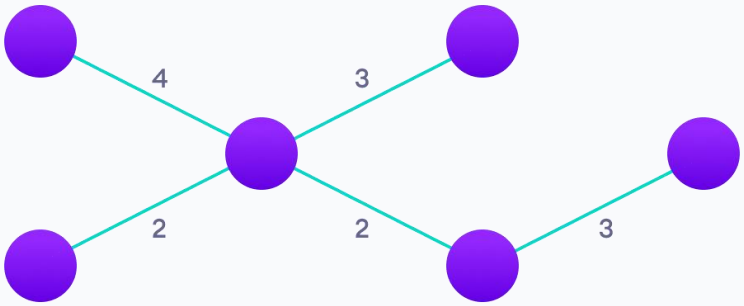
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union FInd. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A ∪ {(u, v)}
   UNION(u, v)
return A
```

Kruskal's vs Prim's Algorithm

Prim's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

Kruskal's Algorithm Complexity

The time complexity Of Kruskal's Algorithm is: O(E log E).

Kruskal's Algorithm Applications

- In order to layout electrical wiring
- In computer network (LAN connection)

B) Prim's Algorithm

In this tutorial, you will learn how Prim's Algorithm works. Also, you will find working examples of Prim's Algorithm in C, C++, Java and Python.

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

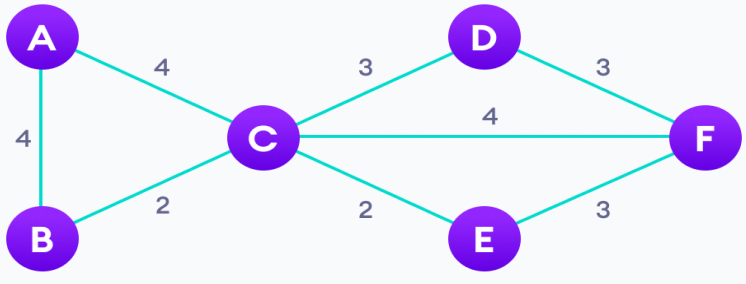
How Prim's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Example of Prim's algorithm

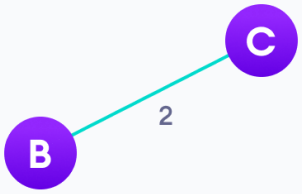


Step: 1

Start with a weighted graph

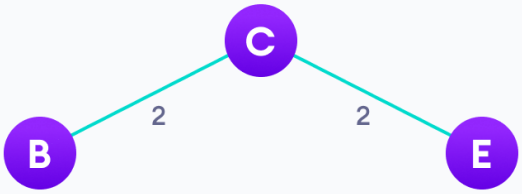
Step: 2

Choose a vertex



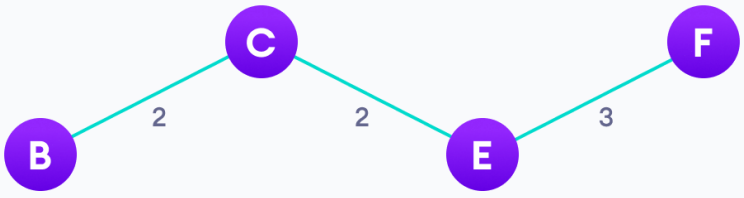
Step: 3

Choose the shortest edge from this vertex and add it



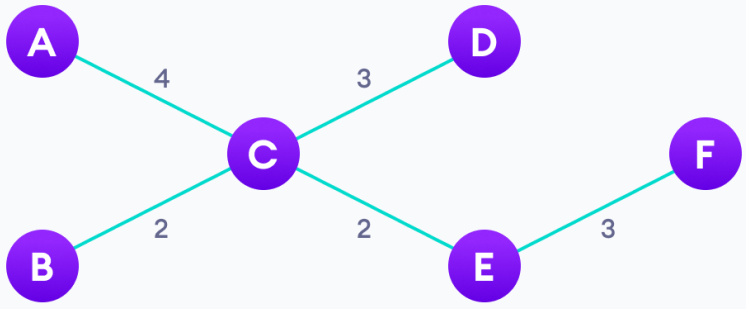
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

```
T = ∅;
U = { 1 };
while (U ≠ V)
   let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
   T = T ∪ {(u, v)}
   U = U ∪ {v}
```

Prim's vs Kruskal's Algorithm

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

Prim's Algorithm Complexity

The time complexity of Prim's algorithm is O(E log V).

Prim's Algorithm Application

- Laying cables of electrical wiring
- In network designed
- To make protocols in network cycles

A) KRUSKAL'S ALGORITHM

SOURCE CODE:

```
#include<iostream>
#include <algorithm>
#include<vector>
using namespace std;

class edge
{

public:
   int s;
   int d;
   int w;

   edge()
   {

   }
   edge(int src,int des,int wei)
   {
      s=src;
      d=des;
      w=wei;
   }
};


bool compare(edge e1,edge e2)
{
   return e1.w<e2.w;
}


int findparent(int i,int* parent )
{
   if(parent[i]==i)
   {
    return i;
   }
   return findparent(parent[i],parent);
}


class graph
{

public:
```

```
int e,n;
edge* v;

graph(int n,int e)
{

   this->n=n;
   this->e=e;
   v=new edge[e];
   for(int i=0; i<e; i++)
   {
      int x,y,w;
      cout<<"ENTER VERTICES AND WEIGHT OF EDGE "<<i+1<<" :          ";
      cin>>x>>y>>w;
      edge e(x,y,w);
      v[i]=e;
   }
}

edge* unionfind()
{
   int* parent=new int[n];
   for(int i=0; i<n; i++)
   {
      parent[i]=i;

   }

   sort(v,v+e,compare);

   edge* output;
   output=new edge[n-1];
   int count=0,i=0;
   while(count!=n-1)
   {
      edge c=v[i];
      int sourceparent=findparent(v[i].s,parent);
      int desparent=findparent(v[i].d,parent);

      if(sourceparent!=desparent)
      {
         output[count]=c;
         parent[sourceparent]=desparent;
         count++;
      }
      i++;

   }

   int sum=0;
   cout<<endl<<"-------MST-------\n";
```

```
    for(int i=0; i<n-1; i++)
    {
      cout<<output[i].s<<"      "<<output[i].d<<"        "<<output[i].w<<endl;
      sum+=output[i].w;
    }
    cout<<"\nWEIGHT OF MST IS "<<sum;
    return output;
  }
};


int main()
{
  int n,e;
  cout<<"KRUSKAL'S ALGORITHM\nENTER NUMBER OF VERTICES :       ";
  cin>>n;
  cout<<"ENTER NUMBER OF EDGEES :  ";
  cin>>e;
  graph g(n,e);
  edge* mst=g.unionfind();
}
```

OUTPUT:

"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\kruskal.exe"

```
KRUSKAL'S ALGORITHM
ENTER NUMBER OF VERTICES :      8
ENTER NUMBER OF EDGEES :        12
ENTER VERTICES AND WEIGHT OF EDGE 1 :   0 1 2
ENTER VERTICES AND WEIGHT OF EDGE 2 :   1 5 5
ENTER VERTICES AND WEIGHT OF EDGE 3 :   1 7 6
ENTER VERTICES AND WEIGHT OF EDGE 4 :   2 7 6
ENTER VERTICES AND WEIGHT OF EDGE 5 :   7 5 6
ENTER VERTICES AND WEIGHT OF EDGE 6 :   1 3 5
ENTER VERTICES AND WEIGHT OF EDGE 7 :   4 5 3
ENTER VERTICES AND WEIGHT OF EDGE 8 :   3 4 10
ENTER VERTICES AND WEIGHT OF EDGE 9 :   4 5 9
ENTER VERTICES AND WEIGHT OF EDGE 10 :  1 6 5
ENTER VERTICES AND WEIGHT OF EDGE 11 :  2 7 6
ENTER VERTICES AND WEIGHT OF EDGE 12 :  1 3 7

-------MST-------
0       1       2
4       5       3
1       5       5
1       3       5
1       6       5
1       7       6
2       7       6

WEIGHT OF MST IS 32
Process returned 0 (0x0)   execution time : 89.553 s
Press any key to continue.
```

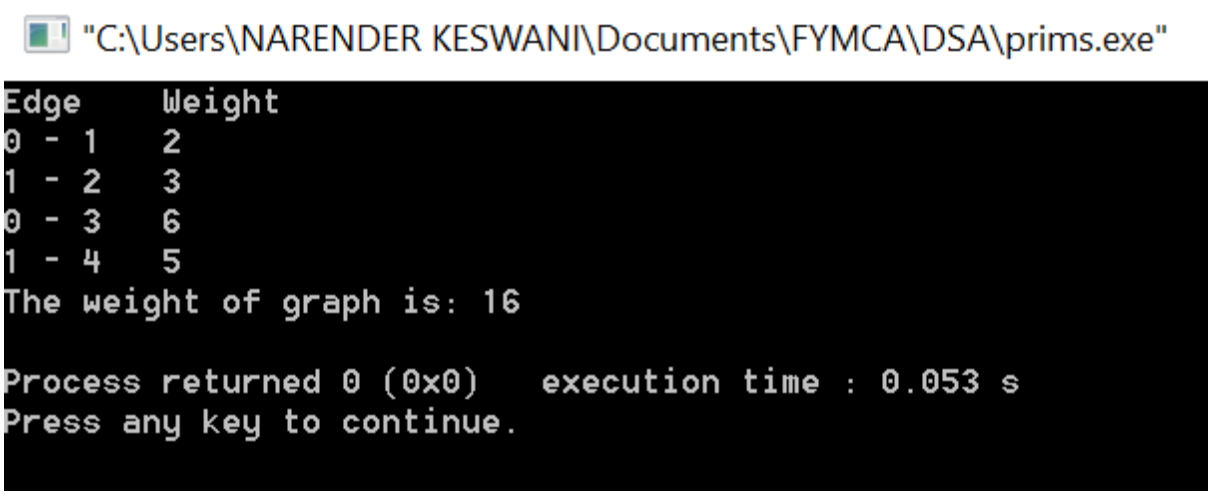
B) PRIM'S ALGORITHM:

SOURCE CODE:

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

#define V 5

int minKey(int key[], bool mstSet[])
{
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
   {

      if (mstSet[v] == false && key[v] < min)
      {
         min = key[v], min_index = v;
      }
   }

   return min_index;
}

void printMST(int parent[], int graph[V][V])
{
   int a = 0;
   cout<<"Edge \tWeight\n";
   for (int i = 1; i < V; i++)
   {
      cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
      a = a +  graph[i][parent[i]];
   }
   cout<<"The weight of graph is: "<<a<<endl;
}

void primMST(int graph[V][V])
{
   int parent[V];

   int key[V];

   bool mstSet[V];

   for (int i = 0; i < V; i++)
   {

      key[i] = INT_MAX, mstSet[i] = false;
   }
```

```
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);

        mstSet[u] = true;


        for (int v = 0; v < V; v++)
        {


            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            {
                parent[v] = u, key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph);
}

int main()
{
    /* Let us create the following graph
            2 3
    (0)--(1)--(2)
    | / \ |
    6| 8/ \5 |7
    | / \ |
    (3)-------(4)
                        9           */
    int graph[V][V] =
    { { 0, 2, 0, 6, 0 },
        { 2, 0, 3, 8, 5 },
        { 0, 3, 0, 0, 7 },
        { 6, 8, 0, 0, 9 },
        { 0, 5, 7, 9, 0 }
    };

    primMST(graph);

    return 0;
}
```

OUTPUT:



CONCLUSION:

From this practical, I have learned how to implement minimum spanning tree using Kruskal & prims algorithms.

AIM: Implementation of Graph traversal. (DFS and BFS)

THEORY:

A) Depth First Search (DFS):

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

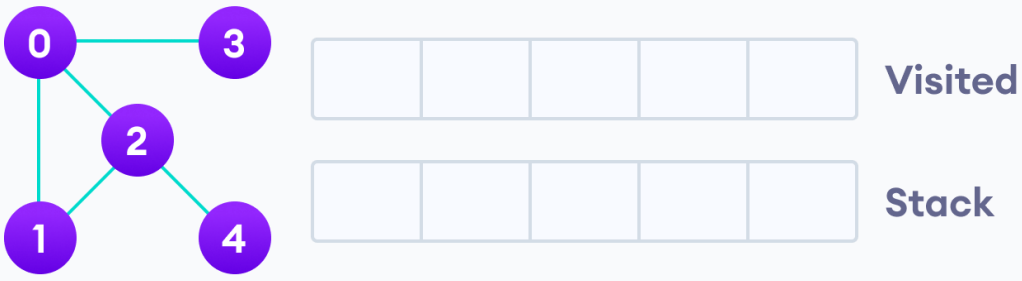
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack 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 top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

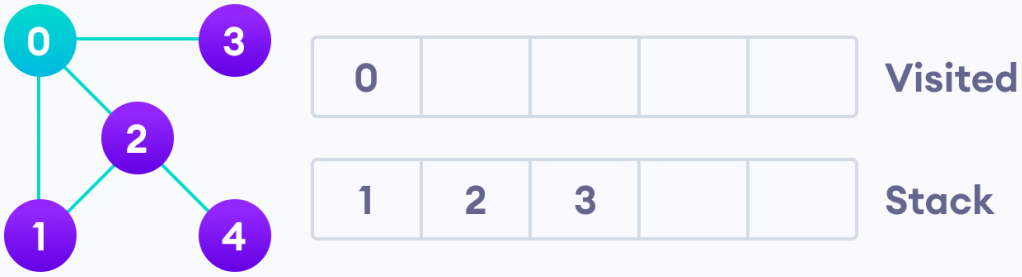
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



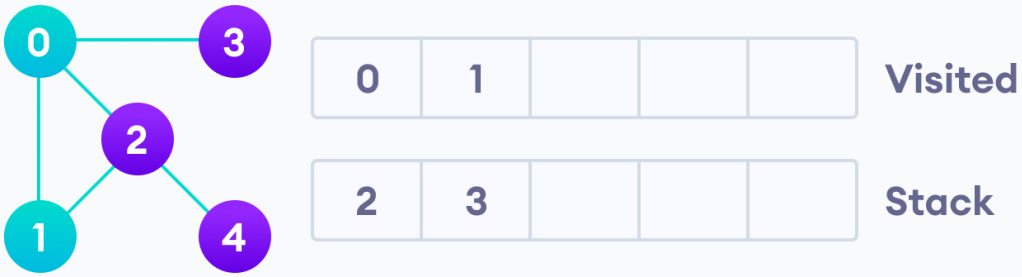
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



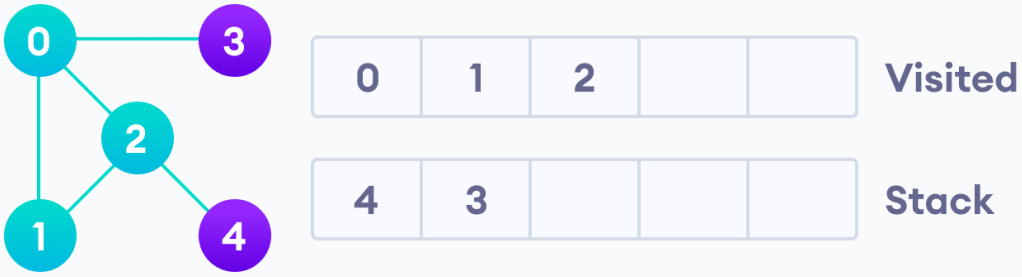
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

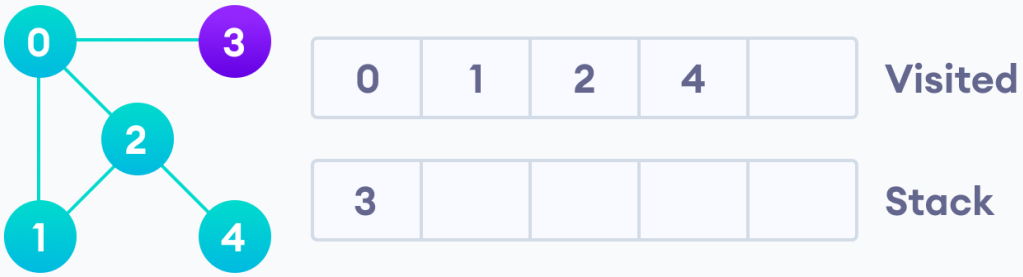


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

B) Breadth First Search (BFS):

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS 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.

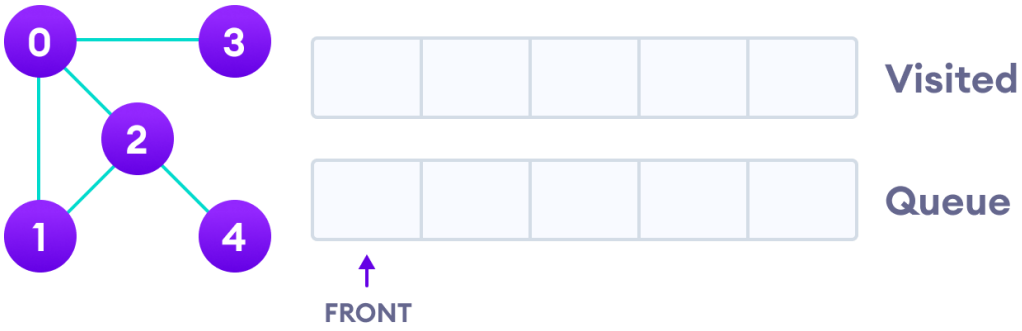
The algorithm works as follows:

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.

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

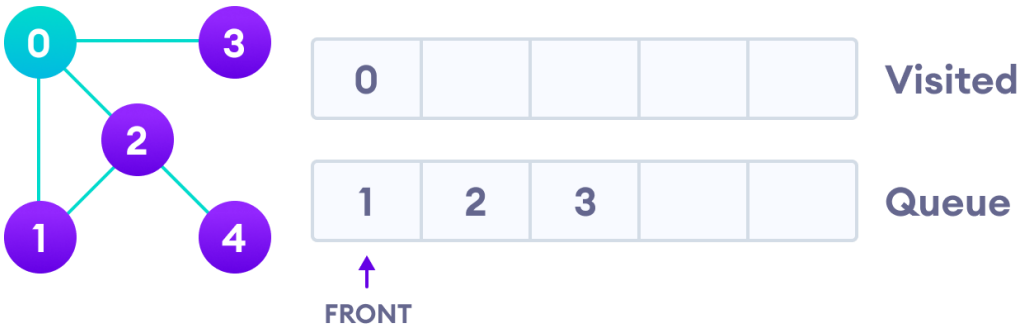
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



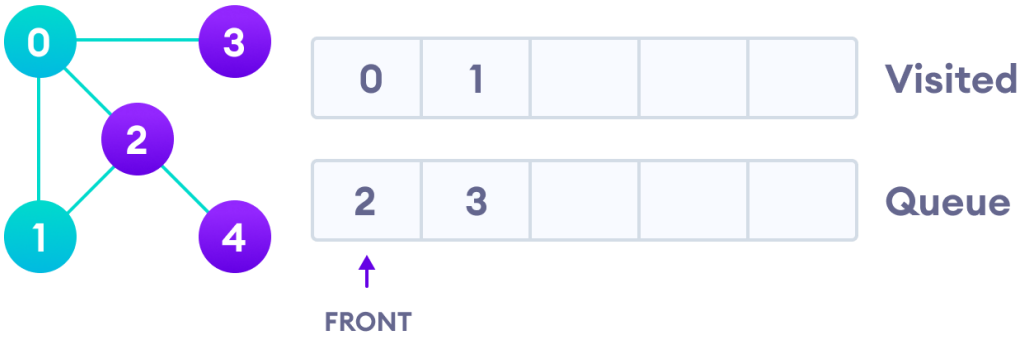
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



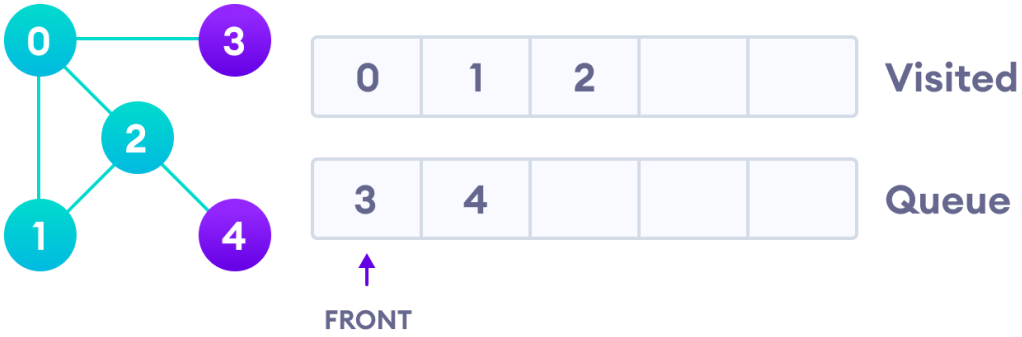
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

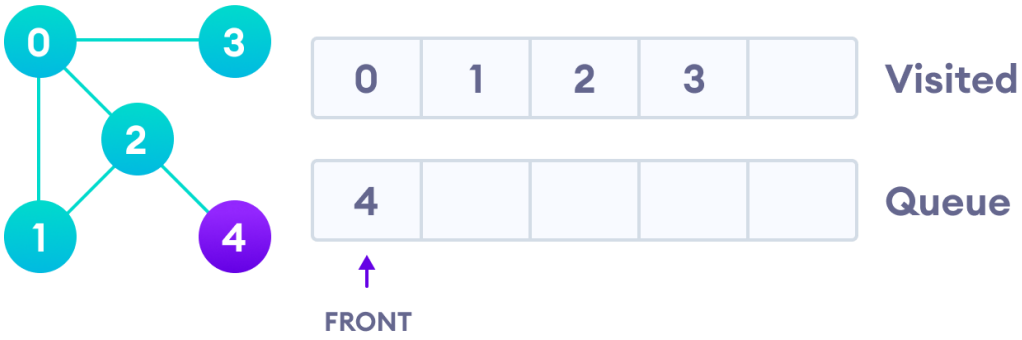


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

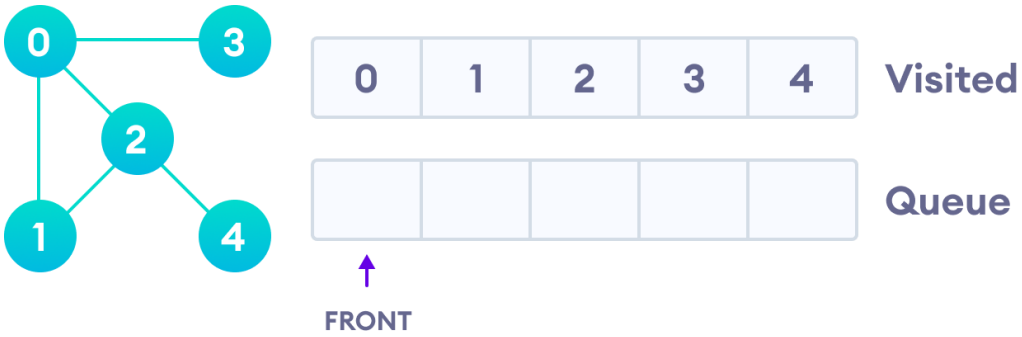


Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the queue to check if it has unvisited neighbours

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

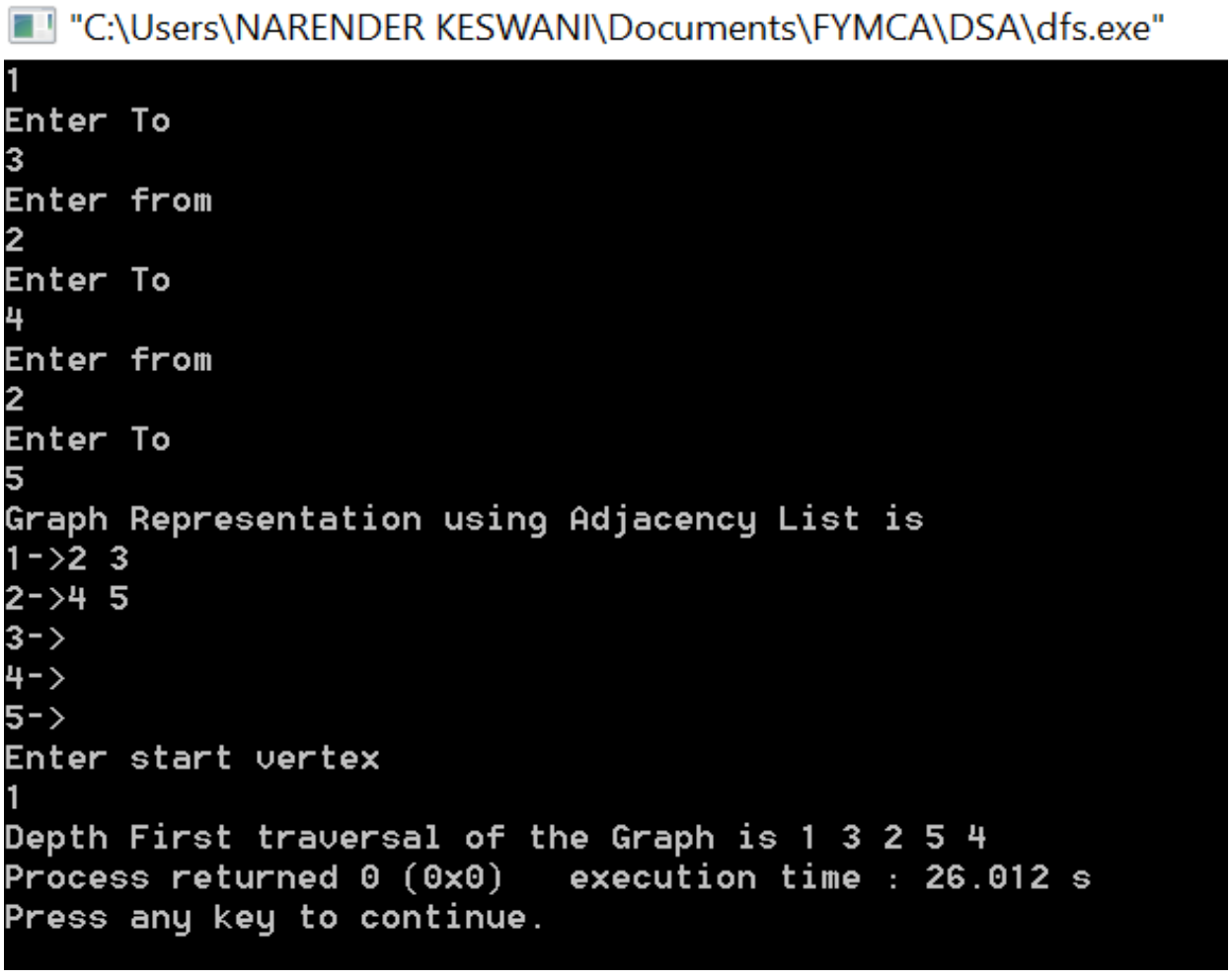
A) Depth First Search (DFS):

SOURCE CODE:

```
#include<iostream>
#include<vector>
#include<stack>
using namespace std;
void addEdge(int u,int v,vector<int>* V)
{
        V[u].push_back(v);
}
void DFS(int s,vector<int>* adj,int n)
{
  int visited[n+1]={0};
  stack<int> stack;
  stack.push(s);
  vector<int>::iterator i;
  while (!stack.empty())
  {
    s = stack.top();
    stack.pop();

    if (!visited[s])
    {
      cout << s << " ";
      visited[s] = true;
    }
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
      if (!visited[*i])
        stack.push(*i);
  }
}
int main()
{
        int n,e,u,v,start;
        cout<<"Enter no of vertices"<<endl;
        cin>>n;
        cout<<"Enter no of Edges"<<endl;
        cin>>e;
        int copy=n;
        vector<int> V[n+1];
        for(int i=0;i<e;i++)
        {
                cout<<"Enter from"<<endl;
                cin>>u;
                cout<<"Enter To"<<endl;
                cin>>v;
                addEdge(u,v,V);
        }
        cout<<"Graph Representation using Adjacency List is"<<endl;
        vector<int>::iterator it;
```

```
        for(int i=1;i<=n;i++)
        {
                cout<<i<<"->";
                for(it=V[i].begin();it!=V[i].end();it++)
                {
                        cout<<*it<<" ";
                }
                cout<<endl;
        }
        cout<<"Enter start vertex"<<endl;
        cin>>start;
        cout<<"Depth First traversal of the Graph is ";
        DFS(start,V,n);
        return 0;
}
```

OUTPUT:



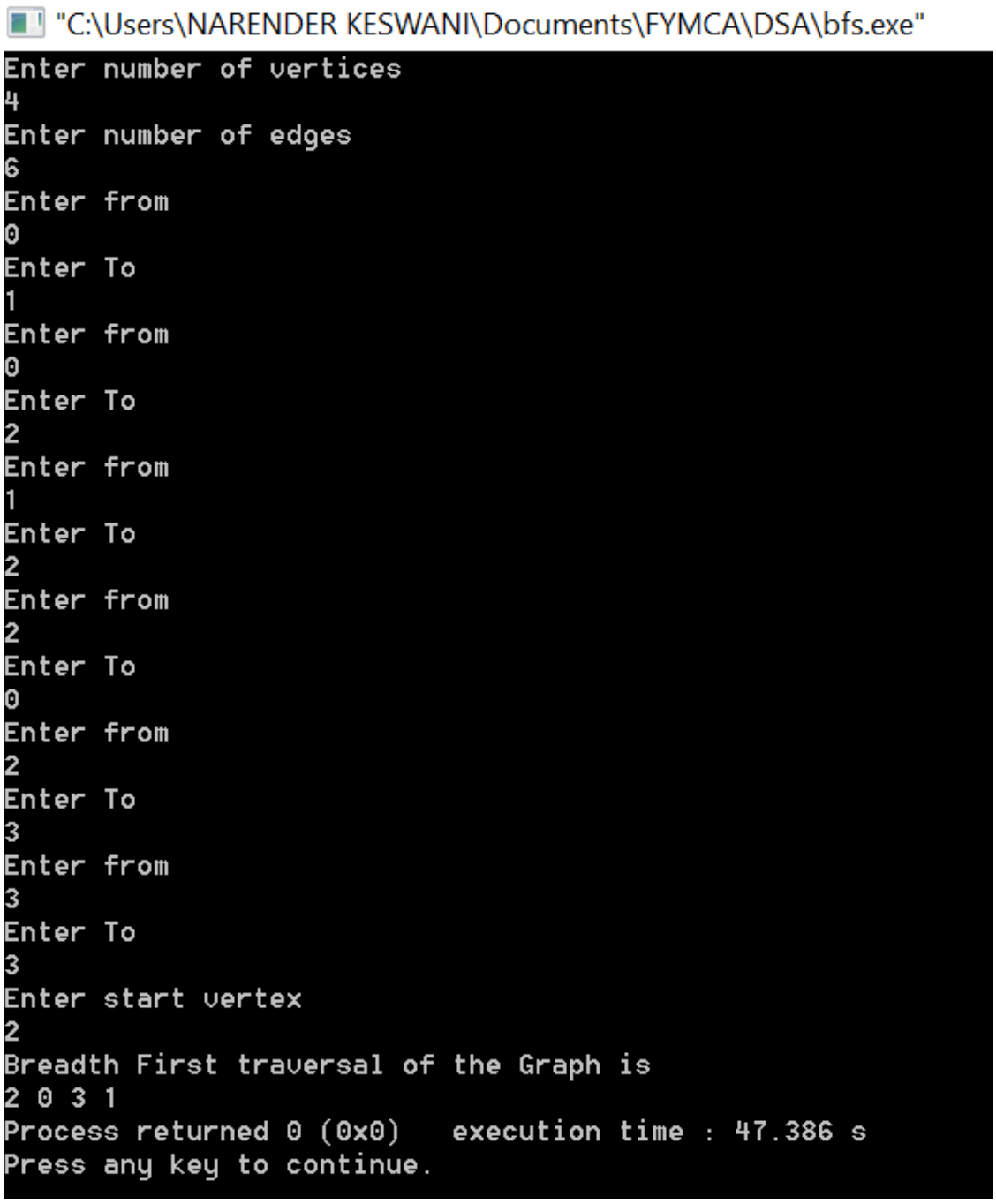
B) Breadth First Search (BFS):

SOURCE CODE:

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

class Graph
{
    int V;
    list<int> *adj;
public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::BFS(int s)
{
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
            visited[i] = false;

    list<int> queue;
    visited[s] = true;
    queue.push_back(s);
    list<int>::iterator i;

    while(!queue.empty())
    {
            s = queue.front();
            cout << s << " ";
            queue.pop_front();

            for (i = adj[s].begin(); i != adj[s].end(); ++i)
            {
                    if (!visited[*i])
                    {
                            visited[*i] = true;
```

```
                                queue.push_back(*i);
                            }
                        }
                    }
                }

int main()
{
   int vt,e,u,v,start;
   cout<<"Enter number of vertices"<<endl;
    cin>>vt;
    Graph g(vt);
   cout<<"Enter number of edges"<<endl;
    cin>>e;
    for(int i=0;i<e;i++)
    {
            cout<<"Enter from"<<endl;
            cin>>u;
            cout<<"Enter To"<<endl;
            cin>>v;
            g.addEdge(u,v);
    }
    cout<<"Enter start vertex"<<endl;
    cin>>start;
    cout<<"Breadth First traversal of the Graph is "<<endl;
    g.BFS(start);
    return 0;
}
```

OUTPUT:



CONCLUSION:

From this practical, I have learned how to implement the DFS & BFS in c++.

AIM: HEAP SORT & SPARSE MATRIX

THEORY:

Heap Sort:

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees. Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

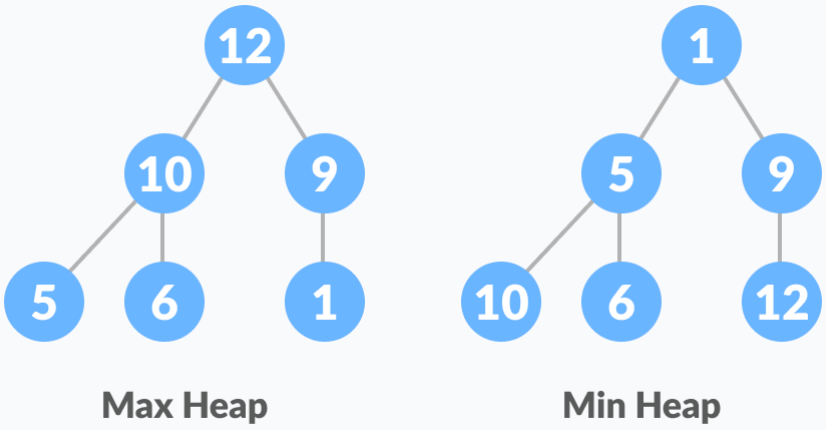
If the index of any element in the array is `i`, the element in the index `2i+1` will become the left child and element in `2i+2` index will become the right child. Also, the parent of any element at index `i` is given by the lower bound of `(i-1)/2`.

What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

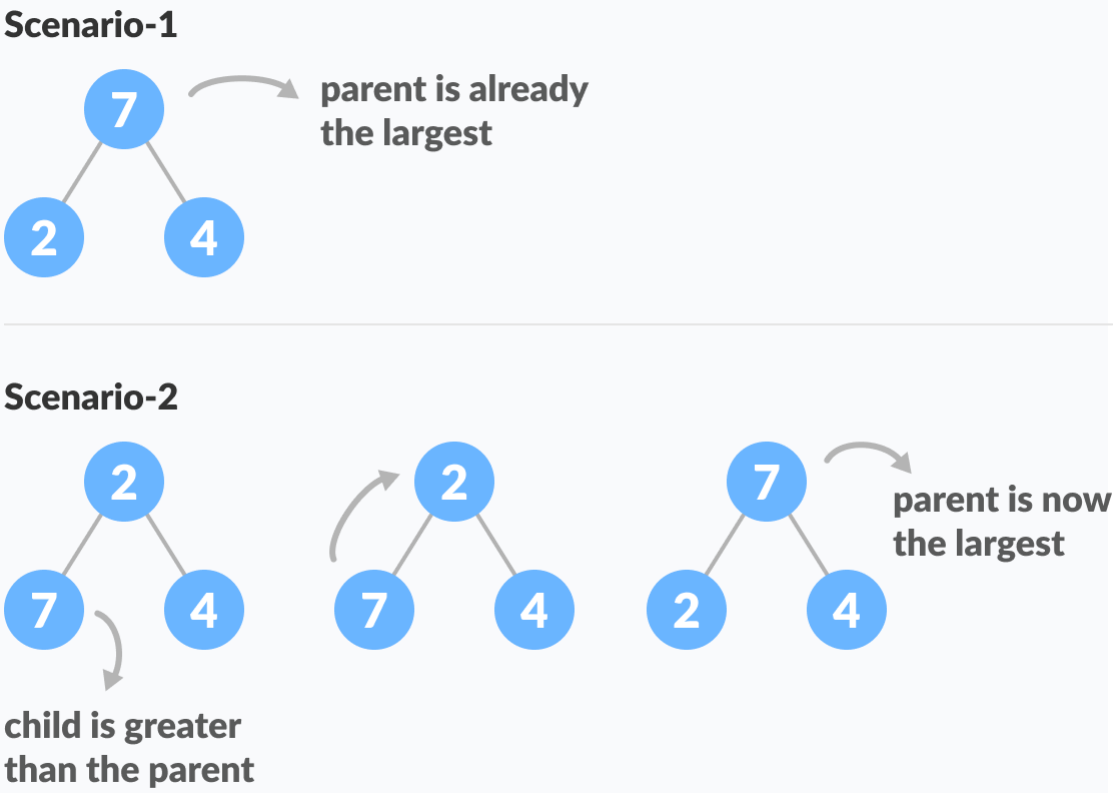
- it is a complete binary tree
- All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.



How to "heapify" a tree

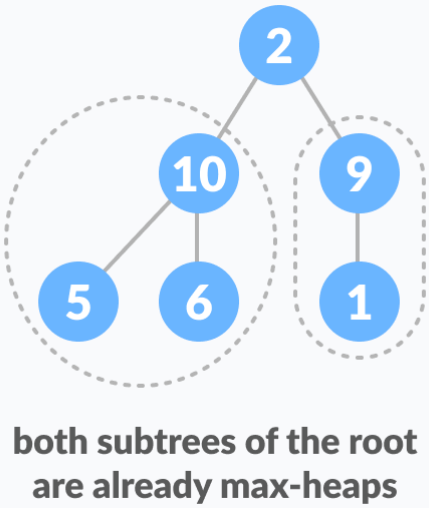
Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap. Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.



The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.

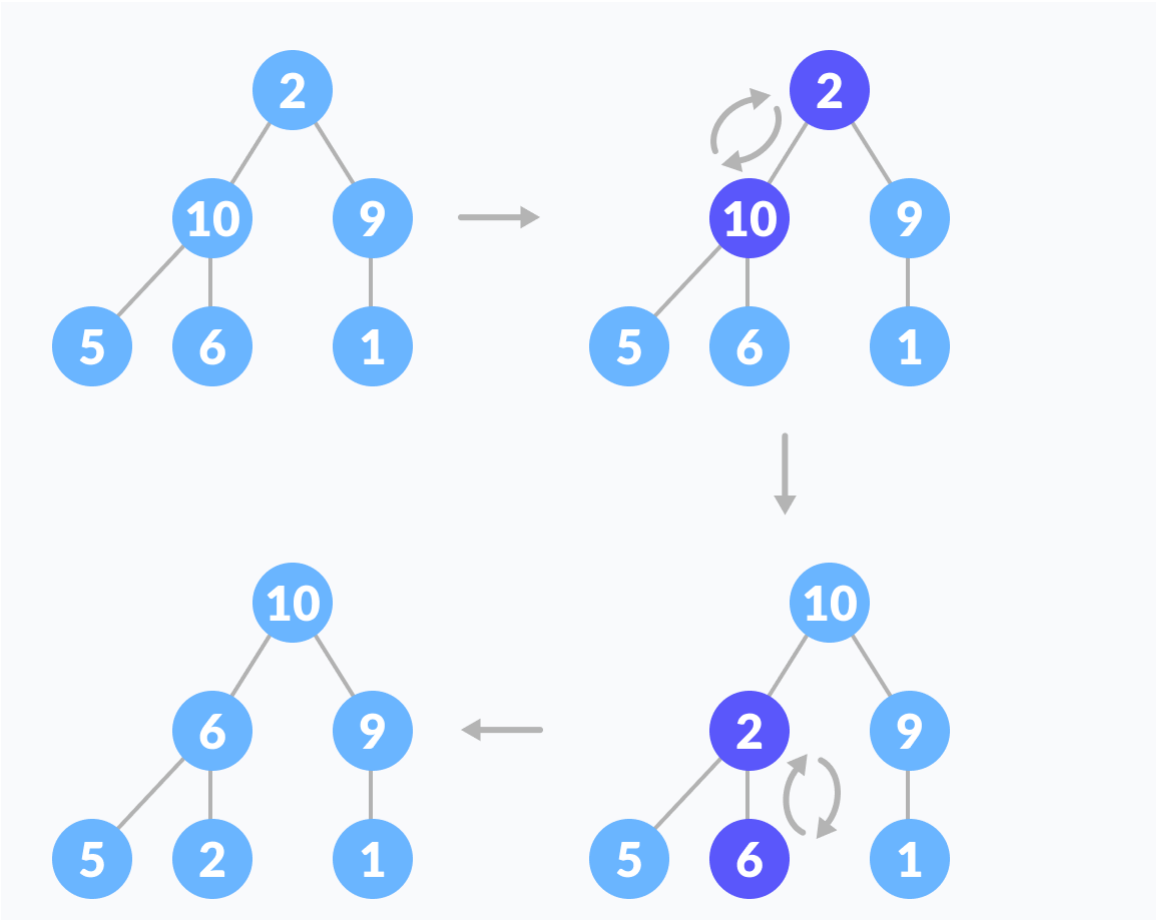


both subtrees of the root are already max-heaps

How to heapify root element when its subtrees are already max heaps

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



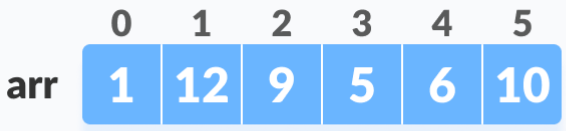
Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

Build max-heap

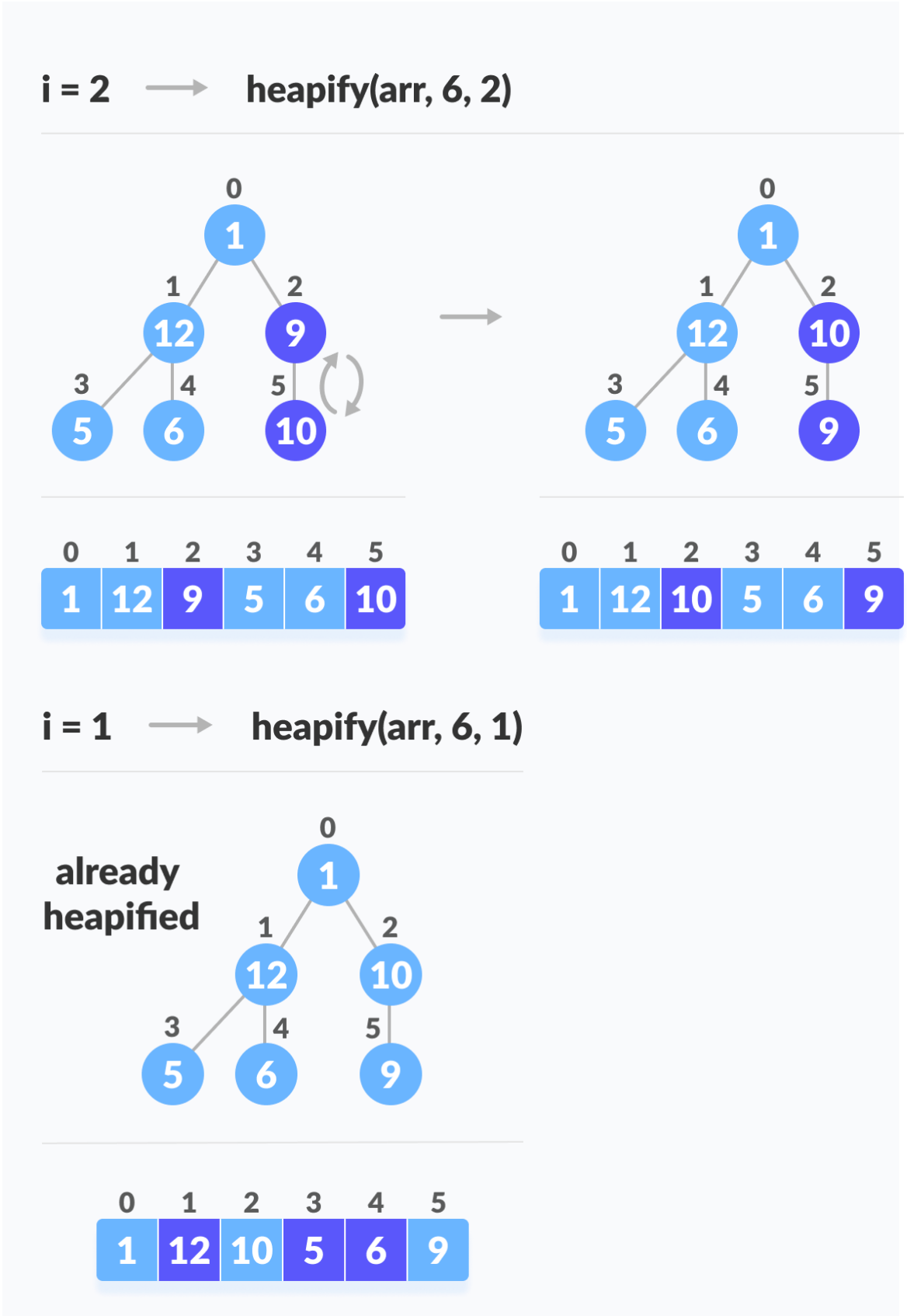
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

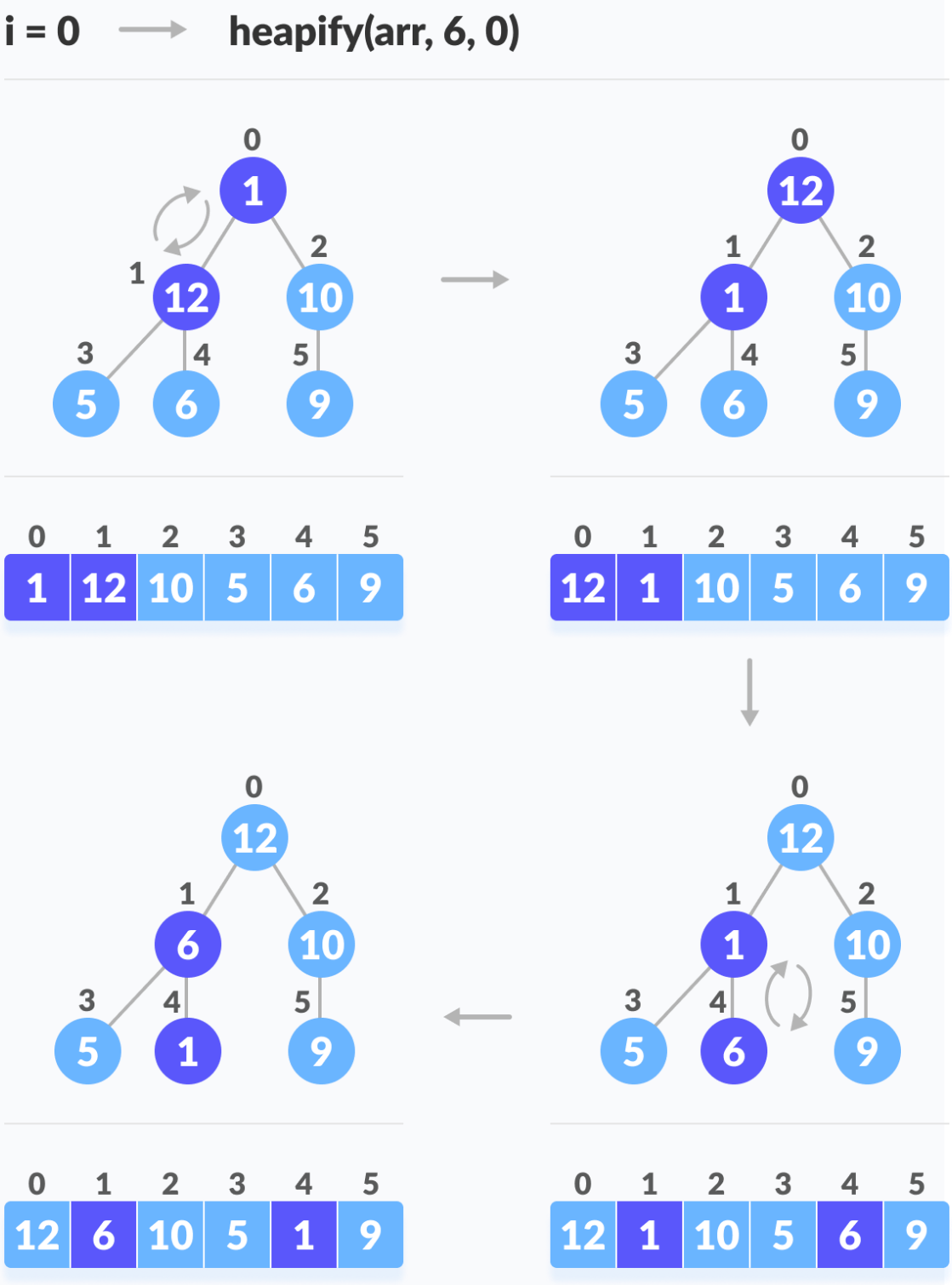
In the case of a complete tree, the first index of a non-leaf node is given by `n/2 - 1`. All other nodes after that are leaf-nodes and thus don't need to be heapified.



n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0


i = 2 → heapify(arr, 6, 2)
0 1 2 3 4 5
1 12 9 5 6 10
0 1 2 3 4 5
1 12 10 5 6 9
i = 1 → heapify(arr, 6, 1)
already heapified
0 1 2 3 4 5
1 12 10 5 6 9



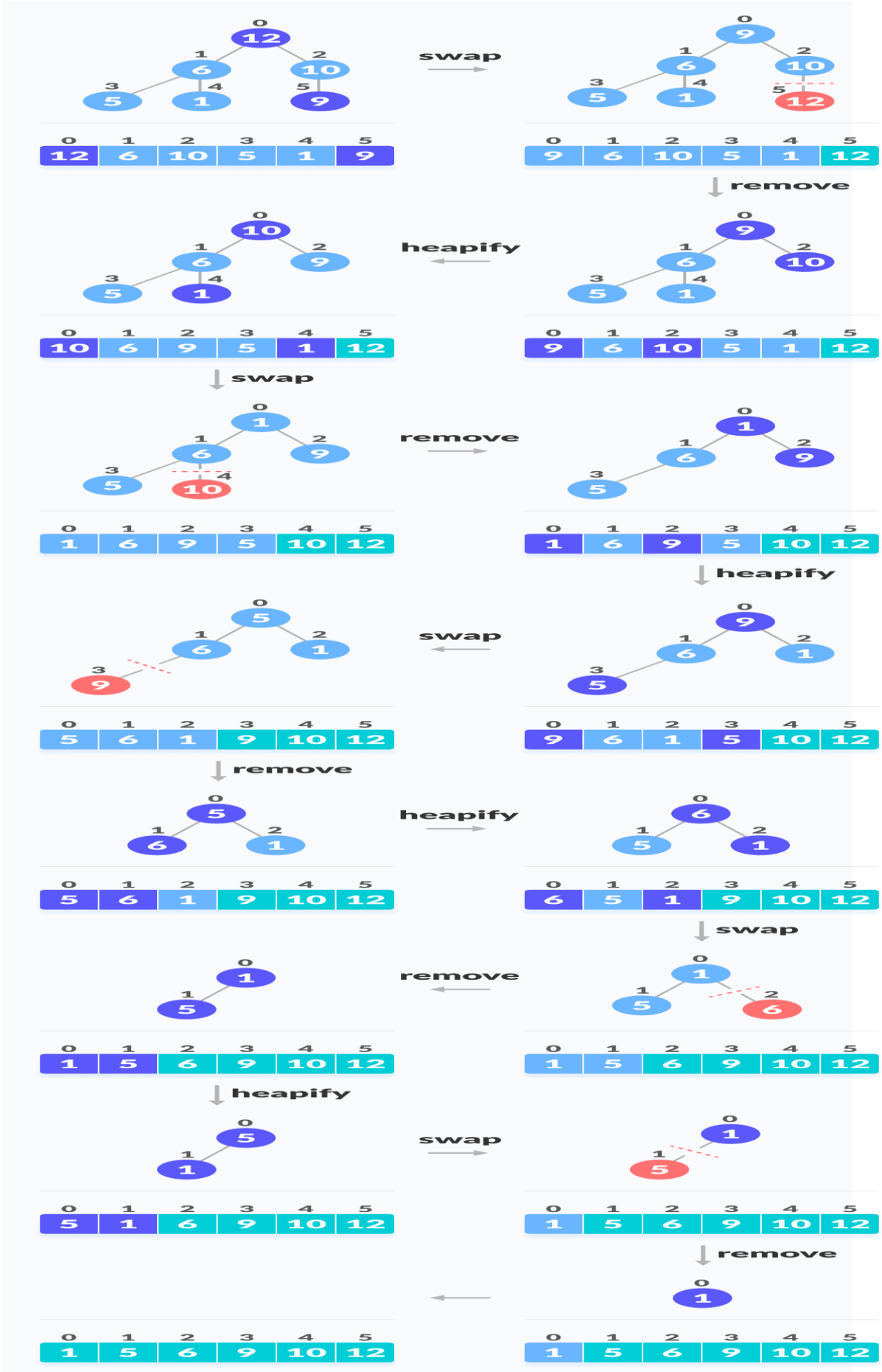
Steps to build max heap for heap sort

As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.


swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap
remove

Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)
Stability	No

Sparse Matrix:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

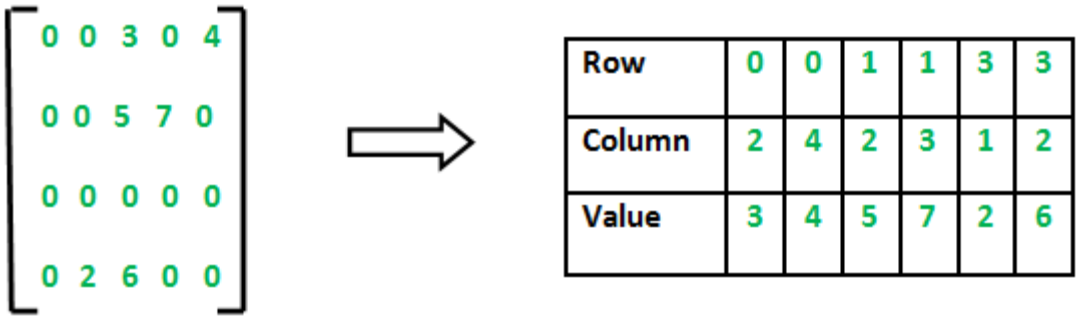
- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:
2D array is used to represent a sparse matrix in which there are three rows named as

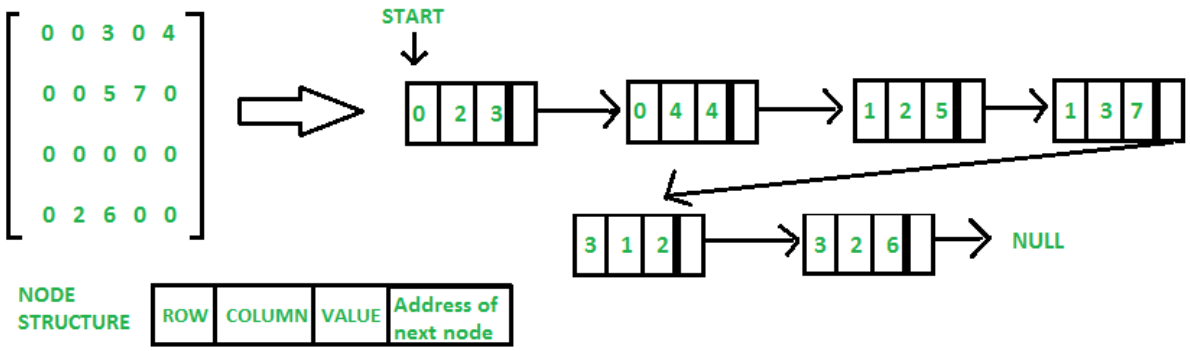
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



1) Implementation of Min Heap and Max Heap Application: Heap Sort

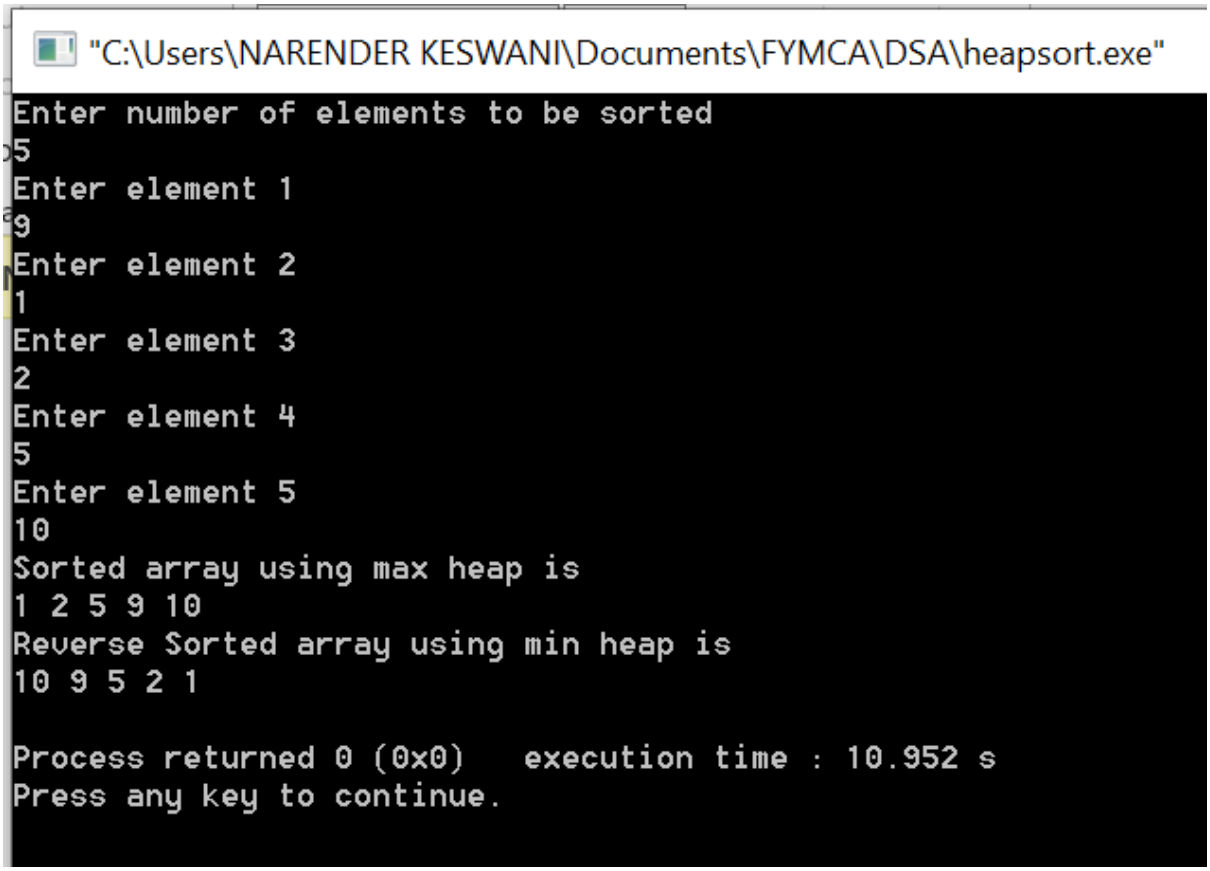
SOURCE CODE:

```
#include <iostream>

using namespace std;
void maxHeapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   if (l < n && arr[l] > arr[largest])
   {
      largest = l;
   }

   if (r < n && arr[r] > arr[largest])
   {
      largest = r;
   }


   if (largest != i)
   {
      swap(arr[i], arr[largest]);
      maxHeapify(arr, n, largest);
   }
}


void minHeapify(int arr[], int n, int i)
{
   int smallest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   if (l < n && arr[l] < arr[smallest])
   {
      smallest = l;
   }

   if (r < n && arr[r] < arr[smallest])
   {
      smallest = r;
   }

   if (smallest != i)
   {
      swap(arr[i], arr[smallest]);
```

```
      minHeapify(arr, n, smallest);
    }
}

void maxHeapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
      maxHeapify(arr, n, i);
    }

    for (int i = n - 1; i > 0; i--)
    {
      swap(arr[0], arr[i]);
      maxHeapify(arr, i, 0);
    }
}

void minHeapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
      minHeapify(arr, n, i);
    }

    for (int i = n - 1; i >= 0; i--)
    {
      swap(arr[0], arr[i]);
      minHeapify(arr, i, 0);
    }
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
      cout << arr[i] << " ";
    }
    cout << "\n";
}


int main()
{
    int n;
    cout<<"Enter number of elements to be sorted"<<endl;
    cin>>n;
    int arr[n];

    for(int i=0; i<n; i++)
    {
```

```
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    cout << "Sorted array using max heap is \n";
    maxHeapSort(arr, n);
    printArray(arr, n);

    cout << "Reverse Sorted array using min heap is \n";
    minHeapSort(arr, n);
    printArray(arr, n);
}
```

OUTPUT:



2) Demonstrate application of linked list - Sparse matrix

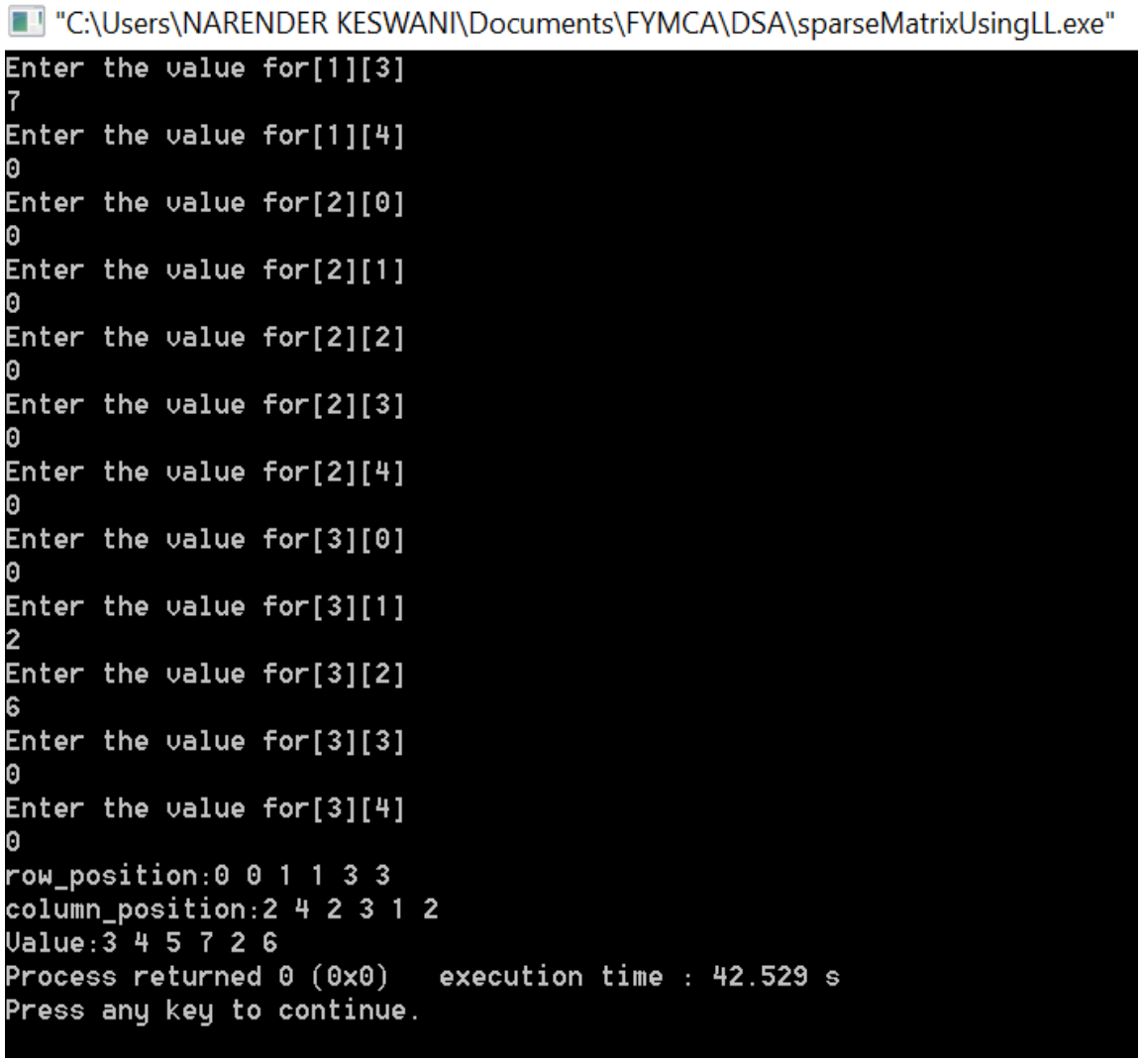
SOURCE CODE:

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

class Node
{
        public:
        int row;
        int col;
        int data;
        Node *next;
};

void create_new_node(Node **p, int row_index,
                                        int col_index, int x)
{
        Node *temp = *p;
        Node *r;

        if (temp == NULL)
        {
                temp = new Node();
                temp->row = row_index;
                temp->col = col_index;
                temp->data = x;
                temp->next = NULL;
                *p = temp;
        }

        else
        {
                while (temp->next != NULL)
                        temp = temp->next;

                r = new Node();
                r->row = row_index;
                r->col = col_index;
                r->data = x;
                r->next = NULL;
                temp->next = r;
        }
}

void printList(Node *start)
{
        Node *ptr = start;
        cout << "row_position:";
        while (ptr != NULL)
        {
```

```
                cout << ptr->row << " ";
                ptr = ptr->next;
        }
        cout << endl;
        cout << "column_position:";

        ptr = start;
        while (ptr != NULL)
        {
                cout << ptr->col << " ";
                ptr = ptr->next;
        }
        cout << endl;
        cout << "Value:";
        ptr = start;

        while (ptr != NULL)
        {
                cout << ptr->data << " ";
                ptr = ptr->next;
        }
}

int main()
{
        int row, col;

        cout<<"Enter number of rows for sparse matrix"<<endl;
        cin>>row;
        cout<<"Enter number of columns for sparse matrix"<<endl;
        cin>>col;

        int sparseMatrix[row][col];

   for(int i = 0; i < row; i++)
        {
                for(int j = 0; j < col; j++)
                {
                   cout<<"Enter the value for["<<i<<"]["<<j<<"]"<<endl;
                   cin>>sparseMatrix[i][j];
                }
        }

        Node *first = NULL;
        for(int i = 0; i < row; i++)
        {
                for(int j = 0; j < col; j++)
                {
                        if (sparseMatrix[i][j] != 0)
        {
                                create_new_node(&first, i, j, sparseMatrix[i][j]);
```

```
            }
                    }
            }
            printList(first);

            return 0;
}
```

OUTPUT:



CONCLUSION:

From this practical, I have learned how to implement heap sort using max and min heap, also learned about sparse matrix implementation using linked lists.