PRACTICAL 1

Problem statement: A. Implementation of Searching Algorithm

a. Linear Search

b. Binary Search

Practical 1.A.a

Aim :- To implement Linear Search Algorithm.

Theory :- Linear search is a method of finding a target element by checking each element one by one from the beginning of the list until the desired element is found or the end of the list is reached.

Algorithm for Linear Search :-

Step 1: Repeat Steps 2 and 3 For i = 1 To n

Step 2: If S[i] = Data

Print "Element found at index "

Exit

Step 3: Set i = i + 1

[End Loop]

Step 4: Print "Desired element Data is not found in the array"

Step 5: Exit

Time And Space Complexity:-

Time Complexity:-

- Best Case: O(1) (element found at first position)
- Worst Case: O(n) (element at last or not present)

Space complexity:-

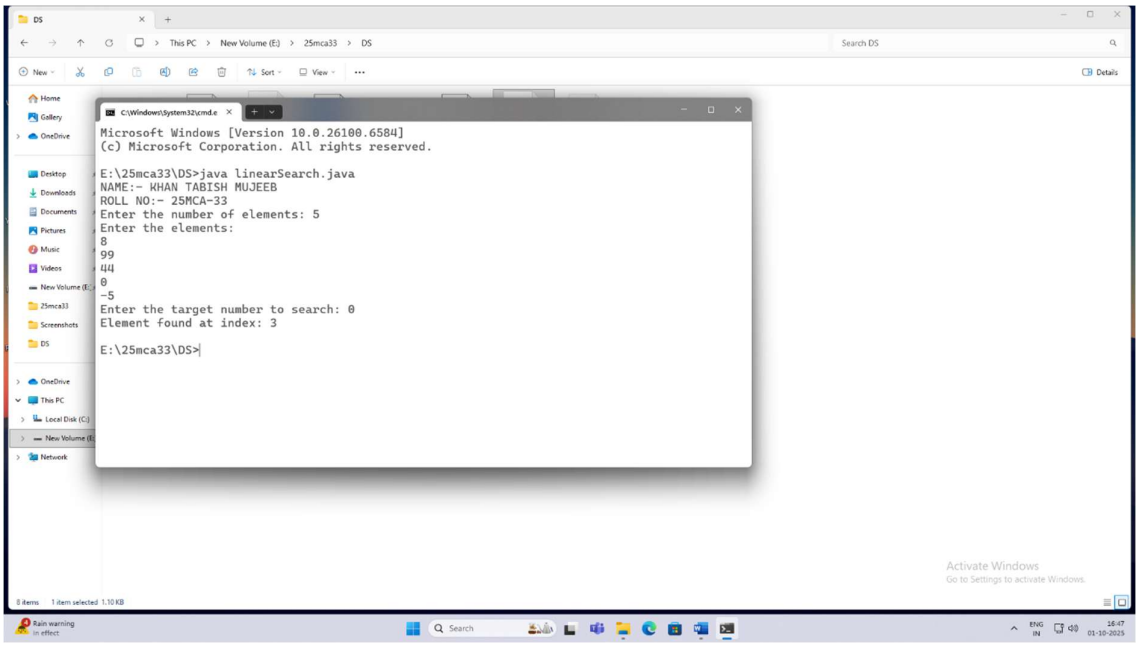
- O(1) — no extra space used apart from variables.

Programming code for Linear Search:-

```
import java.util.Scanner;

class linearSearch {

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);

      System.out.print("NAME:- KHAN TABISH MUJEEB\n");

      System.out.print("ROLL NO:- 25MCA-33\n");

      System.out.print("Enter the number of elements: ");
```

```
        int n = scanner.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        System.out.print("Enter the target number to search: ");
        int target = scanner.nextInt();
        int result = linearSearch(arr, target);
        if (result == -1) {
            System.out.println("Element not found.");
        } else {
            System.out.println("Element found at index: " + result);
        }
        scanner.close();
    }
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

Implementation (Output of the program):-



Limitations of Linear Search:-

1.Inefficient for Large Data Sets

Linear search checks each element one by one, so for large lists or arrays, it becomes slow and time-consuming.

2.Slower Than Other Searching Techniques

When compared to binary search, hashing, or indexed searching, linear search performs poorly in terms of speed.

Conclusion:-Linear search is the simplest and most straightforward searching technique that works on both sorted and unsorted data. However, it is inefficient for large datasets because it checks each element one by one, resulting in high time complexity (O(n)). While it is easy to implement and understand, more efficient algorithms like binary search or hashing are preferred for faster searching in large collections.

Practical 1.A.b

Aim:- To implement a Binary Search algorithm.

Theory:- Binary Search is an efficient searching algorithm used to find the position of a target element in a sorted list or array by repeatedly dividing the search interval in half.

Algorithm for Binary Search:-

Step 1: Set Start = lb, End = ub

Step 2: Repeat Steps 3 to 5 While Start <= End

Step 3: Set Middle = Integer(Start+End/2) //round off value

Step 4: If S[Middle] = Data Then

Print : "Element found at position" : Middle

Exit

[End if]

Step 5: If S[Middle] < Data Then

Set Start = Middle + 1;

Else

Set End = Middle – 1;

[End Loop]

Step 6: Print : "Element does not exists in the array"

Step 7 : Exit

Time And Space Complexity:-

Time Complexity:-

- Best Case: O(1) (element found at first position)
- Worst Case: O(log n) (element at last or not present)

Space Complexity:-

- O(1) — no extra space used apart from variables.

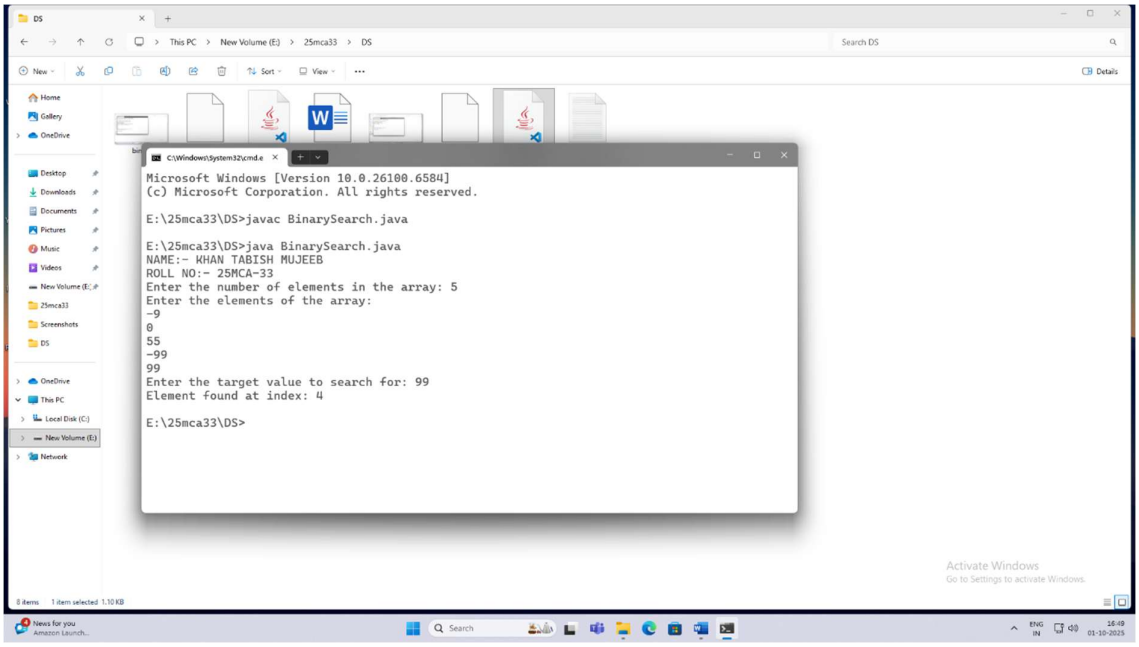
Programming code for Binary Search:-

```
import java.util.Arrays;

import java.util.Scanner;

public class BinarySearch {

   public static int binarySearch(int[] arr, int target) {
```

```
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid;
            }
             else if (arr[mid] > target) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        Arrays.sort(arr);
        System.out.print("Enter the target value to search for: ");
        int target = scanner.nextInt();
        int result = binarySearch(arr, target);
```

```
        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " + result);
        }
        scanner.close();
    }
}
```

Implementation (Output of the program):-



Limitations of Binary Search:-

1.Requires Sorted Data:–

Binary search can only be applied to a list or array that is already sorted. It doesn't work on unsorted data.

2.Difficult for Dynamic Data:–

If the dataset changes frequently (insertions or deletions), maintaining a sorted order becomes difficult and costly.

Conclusion:-Binary search is a fast and efficient algorithm for finding elements in a sorted list. It significantly reduces the number of comparisons by dividing the search range in half each time, making it much faster than linear search for large datasets.

PRACTICAL 1

Program Statement : B. Implementation Of Sorting Techniques:

a. Bubble Sort
b. Insertion Sort
c. Selection Sort
d. Shell Sort
e. Radix Sort
f. Quick Sort

Practical 1.B.a

Aim:- To Implement Bubble Sort Algorithm

Theory:- Bubble Sort is a simple sorting algorithm that repeatedly compares adjacent elements in a list and swaps them if they are in the wrong order. This process continues until the entire list is sorted in ascending or descending order.

Algorithm For Bubble Sort:-

Step 1: Start

Step 2: Set Pass = 1

Step 3: Repeat Step 4 to 7 while Pass ≤ n-1

Step 4: Set i = 0

Step 5: Repeat Steps 6 to 7 while i ≤ n - Pass - 1

If A[i] > A[i + 1]

then

Swap A[i] and A[i + 1]

[End If]

Set i = i + 1 [End of Inner Loop]

Step 6: Set Pass = Pass + 1 [End of Outer Loop]

Step 7: Print “Array is sorted in ascending order”

Step 8: Exit

Time And Space Complexity:-

Time Complexity

- Best Case:O(n)(the array is already sorted)
- Worst Case:$O(n^2)$(the array is sorted in reverse order)

Space Complexity

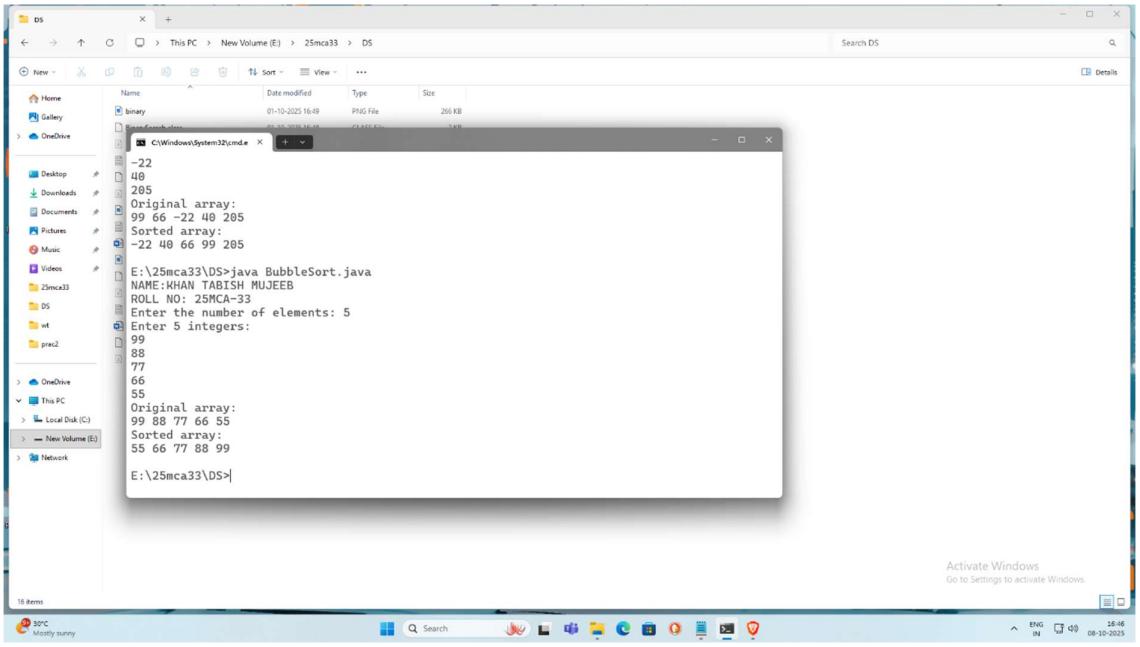
- O(1) - In-place sorting algorithm

Programming Code for Bubble Sort:

```
import java.util.Scanner;
public class BubbleSort {
   public static void bubbleSort(int[] arr) {
      int n = arr.length;
      boolean swapped;
      for (int i = 0; i < n - 1; i++) {
         swapped = false;
         for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
               int temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
               swapped = true;
            }
         }
         if (!swapped) break;
      }
   }
   public static void printArray(int[] arr) {
      for (int num : arr) {
         System.out.print(num + " ");
      }
      System.out.println();
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("NAME:KHAN TABISH MUJEEB");
      System.out.println("ROLL NO: 25MCA-33");
      System.out.print("Enter the number of elements: ");
```

```
        int n = scanner.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter " + n + " integers:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        System.out.println("Original array:");
        printArray(arr);
        bubbleSort(arr);
        System.out.println("Sorted array:");
        printArray(arr);
        scanner.close();
    }
}
```

Implementation (Output Of The Code)



Limitations Of Bubble Sort:

1.High Time Complexity:

It takes $O(n^2)$ time in the average and worst cases, making it unsuitable for real-time or large-scale applications.

2.Requires Many Swaps:

The algorithm performs a large number of swaps, which increases processing time and can reduce efficiency.

Conclusion: Bubble Sort is the simplest sorting algorithm to understand and implement. It works by repeatedly comparing and swapping adjacent elements until the list is sorted. Although it is useful for small datasets and for learning the concept of sorting, it is inefficient for large data due to its $O(n^2)$ time complexity.

Practical 1.B.b

Aim:- To Implement Insertion Sort Algorithm

Theory:- Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It works by taking one element from the unsorted part of the list and inserting it into its correct position in the sorted part.

Algorithm For Bubble Sort:-

Step 1: Start

Step 2: Set i = 1 (Assume the first element A[0] is already sorted)

Step 3: Repeat Steps 4 to 7 while i < n

Step 4: Set temp = A[i] and j = i - 1

Step 5: Repeat while j >= 0 and A[j] > temp

Set A[j + 1] = A[j]

Set j = j - 1 [End of inner loop]

Step 6: Set A[j + 1] = temp

Step 7: Set i = i + 1

[End of Outer Loop]

Step 8: Print" Array is sorted in ascending order"

Step 9: Exit

Time And Space Complexity:

Time Complexity:

- Best Case: O(n)(Array is already sorted)
- Worst Case :$O(n^2)$(Array is sorted in reverse order)

Space Complexity:

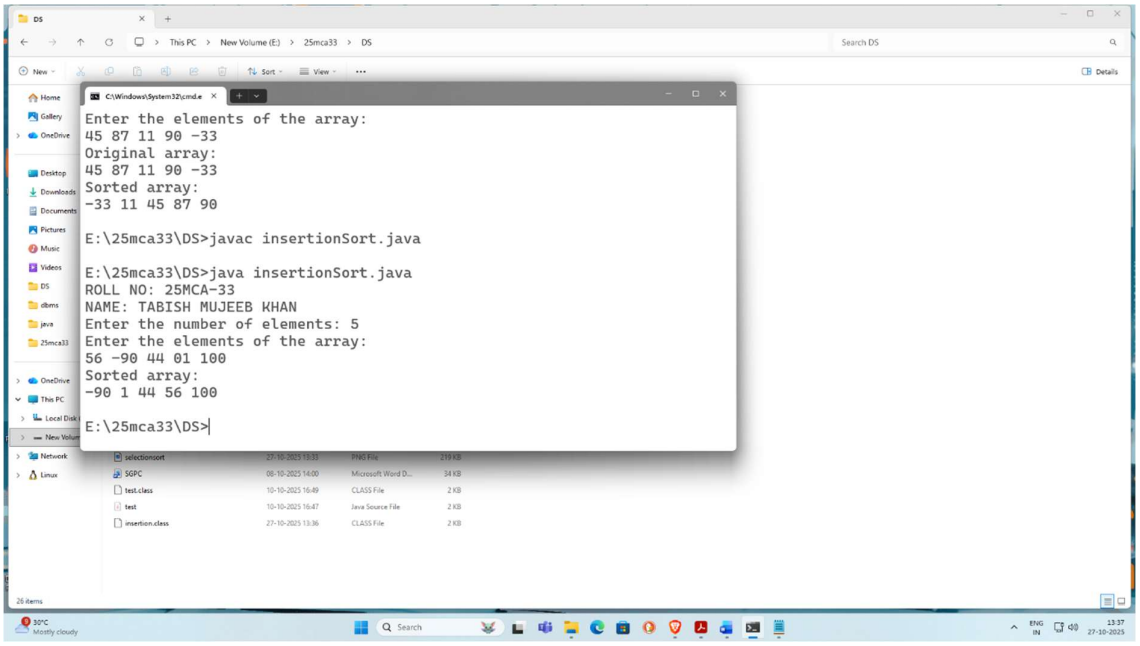
- O(1) - In-place sorting algorithm

Programming Code For Insertion Sort:

```
import java.util.*;

class insertion {

   public static void insertionSort(int[] array) {

      for (int i = 1; i < array.length; i++) {

         int key = array[i];
```

```
            int j = i - 1;
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }
    public static void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("ROLL NO: 25MCA-33\n");
        System.out.print("NAME: TABISH MUJEEB KHAN\n");
        System.out.print("Enter the number of elements: ");
        int n = scanner.nextInt();
        int[] array = new int[n];
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            array[i] = scanner.nextInt();
        }
        insertionSort(array);
        System.out.println("Sorted array:");
        printArray(array);
        scanner.close();
    }
}
```

Implementation (Output Of The Code)



Limitation Of Insertion Sort:

- ❖ Inefficient for Large Data Sets:

Insertion sort becomes very slow when the number of elements increases because it makes many comparisons and shifts.

- ❖ Less Efficient on Random or Reverse Data:

When the data is completely unsorted or in reverse order, it performs the maximum number of comparisons and shifts.

Conclusion: Insertion sort is a simple, stable, and in-place sorting algorithm that is easy to understand and implement. It works efficiently for small datasets and nearly sorted arrays, making it suitable for situations where simplicity is more important than speed.

Practical 1.B.c

Aim:- To Implement Selection Sort Algorithm

Theory: Selection Sort is a simple comparison-based sorting algorithm that works by repeatedly finding the smallest (or largest) element from the unsorted part of the list and placing it at the beginning. It divides the list into two parts: a sorted part and an unsorted part. In each step, the minimum element from the unsorted portion is selected and swapped with the first element of the unsorted part, gradually expanding the sorted portion until the entire list is sorted.

Algorithm for Selection Sort:

Sort an array 'A' with 'n' elements using Selection Sort.

Step 1: Repeat Step 2 to 5 Fo i =1 to n -1

Step 2: Set Min= A[i] and Flag= False

Step 3: Repeat Step 4 For j= i+1 to n

Step 4: If A[i]< Min Then

Set Min=A[j]

Set Pos=j And Flag = True

[End If]

[End Loop]

Step 5: If Flag=True Then

Set Temp=A[i]

Set A[i]= A[Pos]

Set A[Pos]= Temp

[End If]

[End Loop]

Step 6: Exit

Time Complexity

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Space Complexity

- Space Complexity: O(1)

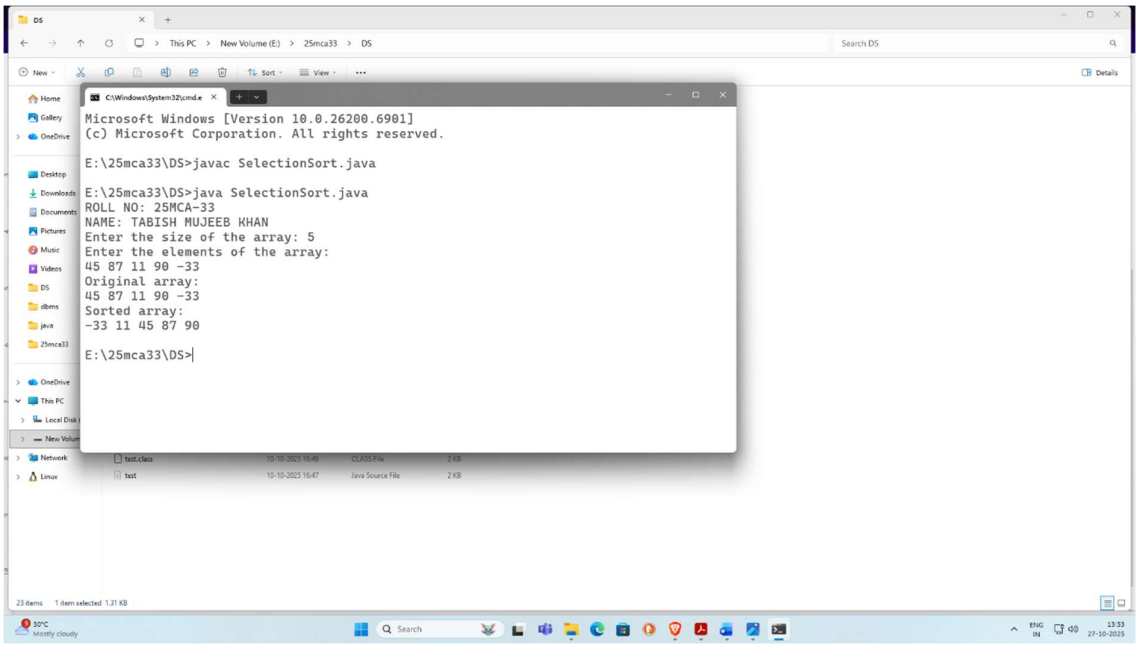
Programming Code for Selection Sort:

```
import java.util.*;
```

```
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("ROLL NO: 25MCA-33\n");
        System.out.print("NAME: TABISH MUJEEB KHAN\n");
        System.out.print("Enter the size of the array: ");
        int size = scanner.nextInt();
        int[] arr = new int[size];
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < size; i++) {
            arr[i] = scanner.nextInt();
        }
```

```
        System.out.println("Original array:");
        printArray(arr);
        selectionSort(arr);
        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

Implementation (Output Of The Code):



Limitations of Selection Sort:

- Inefficient for Large Data Sets:
  Selection sort always performs the same number of comparisons regardless of the input order, making it slow for large datasets.
- Not Adaptive:
  It does not take advantage of already sorted or partially sorted data.
- Not Stable (in basic form):
  Selection sort may change the relative order of equal elements.

Conclusion: Selection sort is a simple, easy-to-understand, and in-place sorting algorithm. It is useful for small datasets and situations where memory usage must be minimal. However, due to its $O(n^2)$ time complexity, it is not suitable for large datasets or performance-critical applications.

Practical 1.B.d

Theory: Shell Sort is an efficient comparison-based sorting algorithm that is an improvement over Insertion Sort. It works by sorting elements that are far apart first and gradually reducing the gap between elements to be compared. The list is divided into sublists based on a gap value, and each sublist is sorted using insertion sort. As the gap decreases, the array becomes more sorted, and when the gap reaches 1, the algorithm performs a final insertion sort, resulting in a fully sorted list.

Algorithm for Shell Sort:

Step 1: Start

Step 2: Read the number of elements n

Step 3: Read the array elements A[0 … n−1]

Step 4: Set gap = n / 2

Step 5: While gap > 0, repeat Steps 6 to 10

Step 6: For i = gap to n − 1, do Steps 7 to 9

Step 7: Store A[i] in temp

Step 8: Set j = i

Step 9: While j ≥ gap and A[j − gap] > temp

Set A[j] = A[j − gap]

Set j = j − gap

Step 10: Set A[j] = temp

Step 11: Reduce gap = gap / 2

Step 12: Display the sorted array

Step 13: Stop

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: $O(n^2)$

Space Complexity:

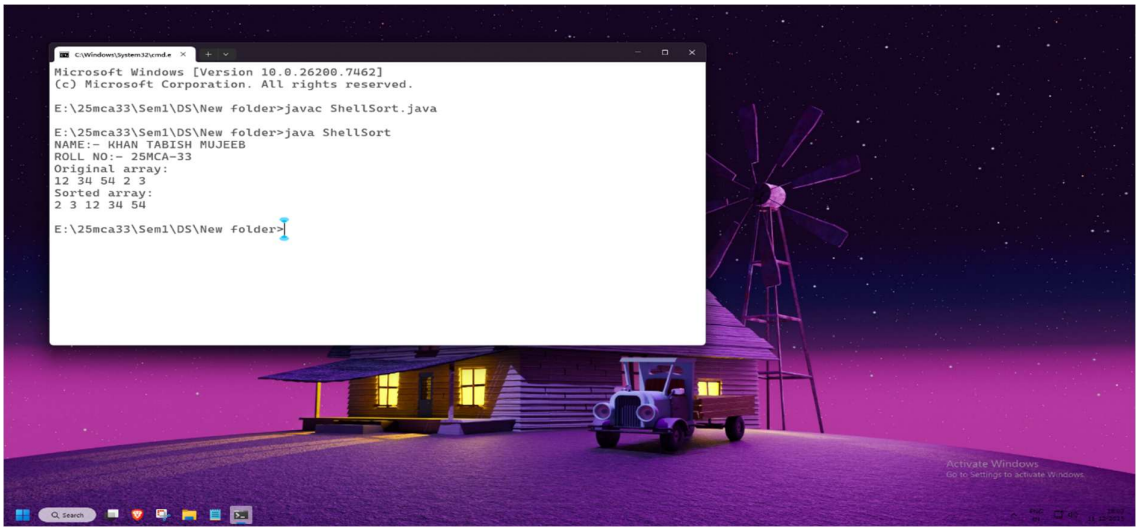
- Space Complexity: O(1)

Programming Code for Shell Sort:

```
class ShellSort {
   public static void shellSort(int[] arr) {
      int n = arr.length;
```

```
        for (int gap = n / 2; gap > 0; gap /= 2) {
            for (int i = gap; i < n; i++) {
                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;
            }
        }
    }
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        int[] arr = {12, 34, 54, 2, 3};
        System.out.println("Original array:");
        printArray(arr);
        shellSort(arr);
        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

Implementation(Output Of The Code):



Limitations of Shell Sort:

- Not Stable:
  Shell sort may change the relative order of equal elements.
- Gap Sequence Dependent:
  Performance varies depending on the chosen gap sequence.
- Complex to Analyze:
  Its time complexity is harder to analyze compared to simpler algorithms.

Conclusion:

Shell sort is an efficient in-place sorting algorithm that performs better than insertion sort for medium-sized datasets. By reducing large displacements early, it significantly improves performance. However, it is not stable and its efficiency depends on the chosen gap sequence.

Practical 1.B.e

Theory: Radix Sort is a non-comparison-based sorting algorithm that sorts elements by processing individual digits of the numbers. It works by grouping numbers according to the value of their digits at a particular position, starting from the least significant digit (or most significant digit) and moving to the next position. At each step, a stable sorting method is used to maintain the relative order of elements with the same digit. This process is repeated for all digit positions until the entire list is sorted.

Algorithm For Radix Sort:

Step 1: Start

Step 2: Read the number of elements n

Step 3: Read the array elements A[0 … n−1]

Step 4: Find the maximum element max in the array

Step 5: Set exp = 1 (exp represents digit position: 1, 10, 100, …)

Step 6: While (max / exp) > 0, repeat Steps 7 to 11

Step 7: Initialize count array count[0 … 9] with 0

Step 8: For i = 0 to n − 1

- Find digit = (A[i] / exp) % 10
- Increment count[digit]

Step 9: Modify count[] so that it contains cumulative positions

Step 10: Build output array output[] by placing elements in correct order according to current digit

Step 11: Copy output[] back to array A[]

Step 12: Set exp = exp × 10

Step 13: Display the sorted array

Step 14: Stop

Time Complexity:

- Best Case: O(nk)
- Average Case: O(nk)
- Worst Case: O(nk)

Space Complexity:

- Space Complexity: O(n + k)

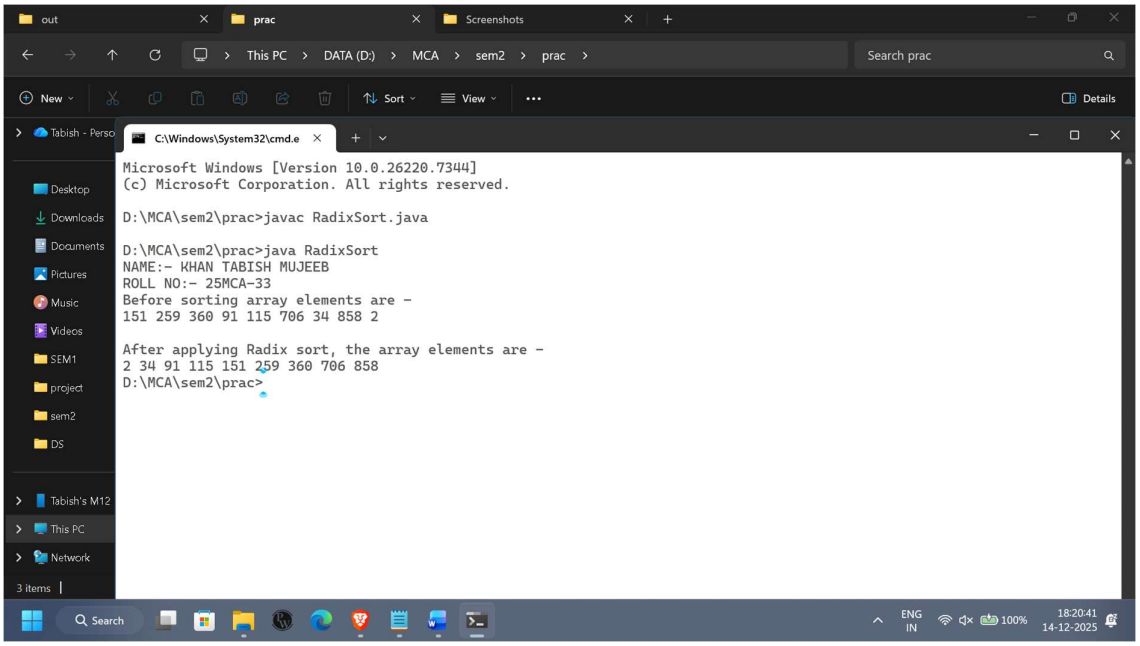
Programming Code for Radix Sort:

```
class RadixSort

{

int getMax(int a[], int n)
```

```
{
int max = a[0];
for(int i = 1; i<n; i++)
{
if(a[i] > max)
max = a[i];
}
return max;
}
void countingSort(int a[], int n, int place)
{
int[] output = new int[n+1];
int[] count = new int[10];
for (int i = 0; i < n; i++)
count[(a[i] / place) % 10]++;
for (int i = 1; i < 10; i++)
count[i] += count[i - 1];
for (int i = n - 1; i >= 0; i--)
{
output[count[(a[i] / place) % 10] - 1] = a[i];
count[(a[i] / place) % 10]--;
}
for (int i = 0; i < n; i++)
a[i] = output[i];
}
void radixsort(int a[], int n)
{
int max = getMax(a, n);
for (int place = 1; max / place > 0; place *= 10)
countingSort(a, n, place);
}
void printArray(int a[], int n)
```

```
{
for (int i = 0; i < n; ++i)
System.out.print(a[i] + " ");
}
public static void main(String args[])
{
  System.out.print("NAME:- KHAN TABISH MUJEEB\n");
       System.out.print("ROLL NO:- 25MCA-33\n");
int a[] = {151, 259, 360, 91, 115, 706, 34, 858, 2};
int n = a.length;
RadixSort r1 = new RadixSort();
System.out.print("Before sorting array elements are - \n");
r1.printArray(a,n);
r1.radixsort(a, n);
System.out.print("\n\nAfter applying Radix sort, the array elements are -\n");
r1.printArray(a, n);
}
}
```

Implementation(Output Of The Code):



Limitations of Radix Sort:

- Extra Space Required:
  Radix sort requires additional memory for auxiliary arrays.
- Restricted to Certain Data Types:
  It works efficiently only with integers or fixed-length strings.
- Not In-Place:
  The algorithm does not sort within the original array only.

Conclusion: Radix sort is an efficient sorting algorithm for large datasets with small digit lengths. Since it does not rely on comparisons, it can outperform comparison-based sorting algorithms in specific cases. However, it requires extra memory and is limited to certain data types.

Practical 1.B.f

Theory: Quick Sort is an efficient comparison-based sorting algorithm that follows the divide-and-conquer approach. It works by selecting a pivot element and partitioning the array into two subarrays: elements smaller than the pivot and elements greater than the pivot. The pivot is placed in its correct sorted position, and the same process is recursively applied to the left and right subarrays. This continues until all subarrays are sorted, resulting in a fully sorted list.

Algorithm For Quick Sort:

Step 1: Start

Step 2: Read the number of elements n

Step 3: Read the array elements A[0 … n−1]

Step 4: Call QUICKSORT(A, 0, n−1)

Procedure: QUICKSORT(A, low, high)

Step 5: If low < high, then do Steps 6 to 8

Step 6: Set p = PARTITION(A, low, high)

Step 7: Call QUICKSORT(A, low, p−1)

Step 8: Call QUICKSORT(A, p+1, high)

Procedure: PARTITION(A, low, high)

Step 9: Select pivot = A[high]

Step 10: Set i = low − 1

Step 11: For j = low to high − 1

- If A[j] ≤ pivot
    - Increment i
    - Swap A[i] and A[j]

Step 12: Swap A[i+1] and A[high]

Step 13: Return i + 1 (pivot position)

Step 14: Display the sorted array

Step 15: Stop

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: $O(n^2)$

Space Complexity:

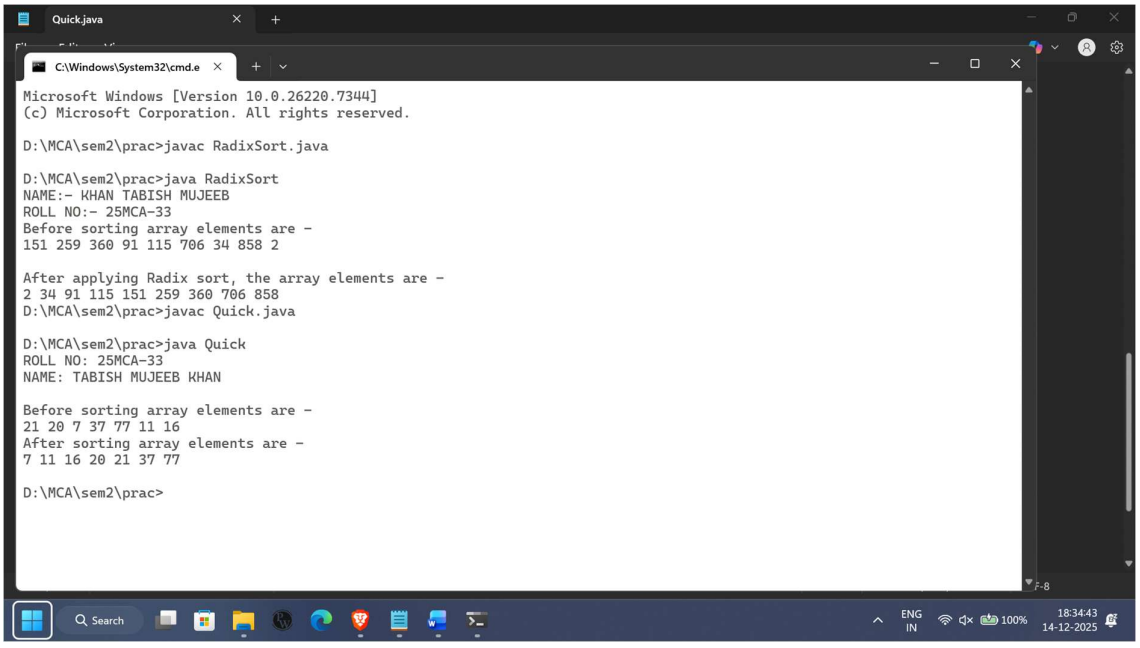
- Space Complexity: O(log n)

Programming Code for Quick Sort:

```
public class Quick
{
int partition (int a[], int start, int end)
{
int pivot = a[end];
int i = (start - 1);
for (int j = start; j <= end - 1; j++)
{
if (a[j] < pivot)
{
i++;
int t = a[i];
a[i] = a[j];
a[j] = t;
}
}
int t = a[i+1];
a[i+1] = a[end];
a[end] = t;
return (i + 1);
}
void quick(int a[], int start, int end)
{
if (start < end)
{
int p = partition(a, start, end);
quick(a, start, p - 1);
quick(a, p + 1, end);
}
}
void printArr(int a[], int n)
{
```

```
int i;
for (i = 0; i < n; i++)
System.out.print(a[i] + " ");
}
public static void main(String[] args)
{
 System.out.print("ROLL NO: 25MCA-33\n");
       System.out.print("NAME: TABISH MUJEEB KHAN\n");
int a[] = { 21, 20, 7, 37, 77, 11,16 };
int n = a.length;
System.out.println("\nBefore sorting array elements are - ");
Quick q1 = new Quick();
q1.printArr(a, n);
q1.quick(a, 0, n - 1);
System.out.println("\nAfter sorting array elements are - ");
q1.printArr(a, n);
System.out.println();
}
}
```

Implementation(Output Of The Code):



Limitations of Quick Sort:

- **Worst-Case Performance:**
  If poor pivot selection occurs, time complexity degrades to $O(n^2)$.
- **Not Stable:**
  Quick sort does not preserve the relative order of equal elements.
- **Recursive Overhead:**
  Uses recursion which may cause stack overflow for very large datasets.

Conclusion: Quick sort is one of the fastest and most widely used sorting algorithms. It performs efficiently for large datasets and has excellent average-case performance. However, its worst-case time complexity and instability are its main drawbacks.

Practical No .2

Aim: To perform various **Hashing Techniques** using **Linear Probing** as the collision resolution scheme..

Theory: Hashing is a technique used to store and retrieve data efficiently by mapping keys to specific locations in a hash table using a hash function. The position generated by the hash function is called the hash address. When two or more keys map to the same hash address, a collision occurs. To handle collisions, collision resolution techniques are used.
Linear Probing is an open addressing method where, in case of a collision, the next available location in the hash table is searched sequentially until an empty slot is found.

Hash Function:

A commonly used hash function is:

$$H(key) = key mod m$$

Where:

- key = value to be stored
- m = size of the hash table

Collision Resolution Technique – Linear Probing:

In Linear Probing, if a collision occurs at position H(key), the algorithm checks the next position sequentially using the formula:

$$H(key, i) = (H(key) + i) mod m$$

Where i = 0, 1, 2, ... until an empty slot is found.

Algorithm for Hashing with Linear Probing:

Step 1: Initialize the hash table of size m with all locations empty.
Step 2: Read the key to be inserted.
Step 3: Compute the hash address using H(key) = key mod m.
Step 4: If the calculated position is empty, insert the key.
Step 5: If a collision occurs, apply Linear Probing:
Check (H(key) + 1) mod m, (H(key) + 2) mod m, and so on.
Step 6: Insert the key at the first available empty position.
Step 7: Repeat Steps 2 to 6 for all keys.
Step 8: Exit.

Time Complexity:

- Best Case: O(1)
- Average Case: O(1)
- Worst Case: O(n)

Space Complexity:

- Space Complexity: O(m)

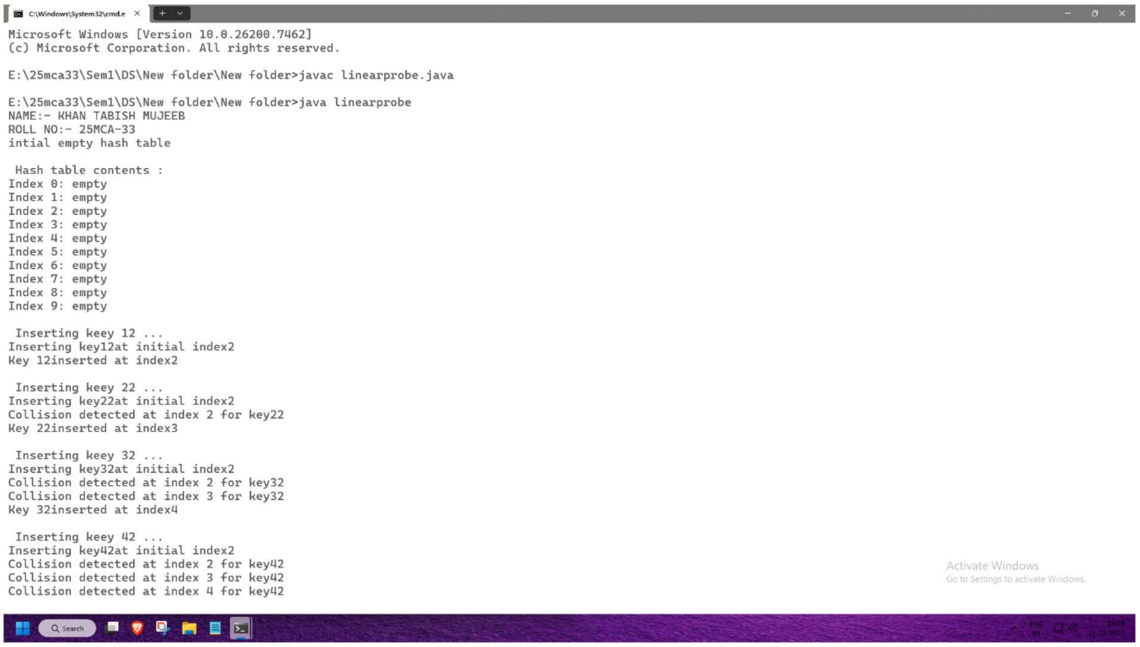
Programming Code for Linear Probing Collision:

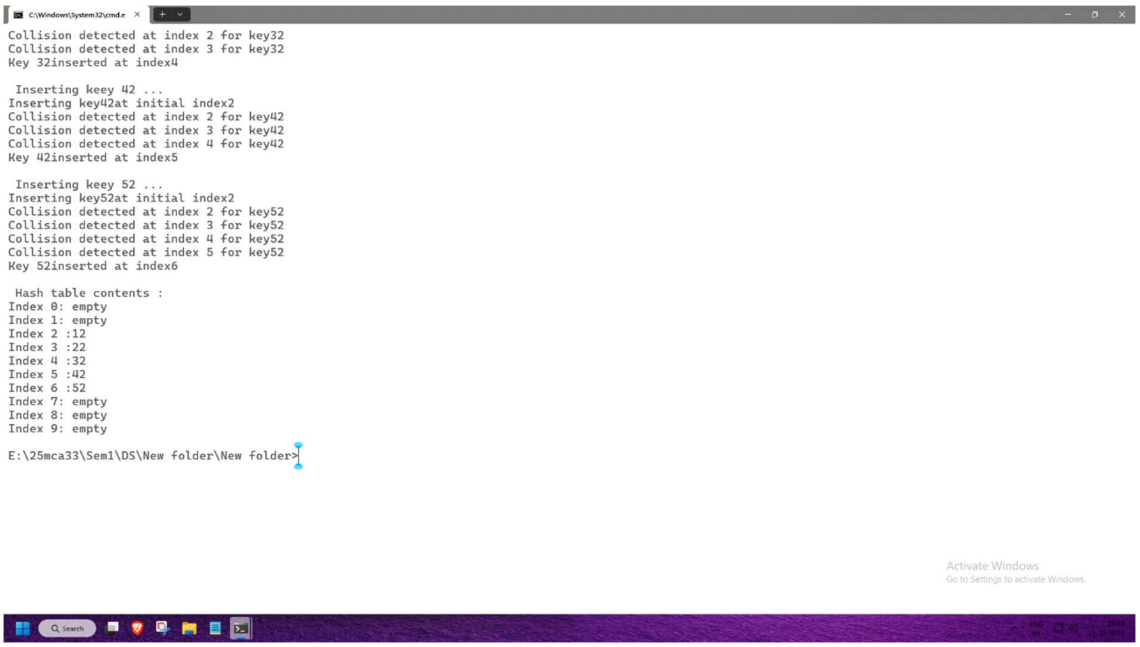
```
class HashTable
{
    private Integer[] table;
     private int size;
     private int capacity;
      public HashTable(int capacity)
      {
           this.capacity = capacity;
           this.size = 0;
           table = new Integer[capacity];
            for (int i = 0 ;i<capacity; i++)
               {
                  table[i] = null;
                  }
               }
             private int hash(int key)
                {
                  return key % capacity;
                  }
              public void insert(int key)
                 {
                   if (size == capacity)
                      {
                       System.out.println("Hash Table is full.can't insert" +key);
                       return;
                       }
                  int index = hash(key);
                System.out.println("Inserting key"+key+"at initial index" + index);
                while (table[index] != null  )
                {
                 System.out.println("Collision detected at index " + index +" for key" +key);
```

```
                index = (index +1) %capacity;
               }
              table[index] = key;
              size++;
              System.out.println("Key " +key + "inserted at index" + index);
               }
        public void display()
          {
            System.out.println("\n Hash table contents :");
            for (int i = 0; i <capacity; i++)
             {
                if(table[i] != null)
                {
                System.out.println("Index " +i +" :" +table[i]);
                 }
                 else
                   {
                 System.out.println("Index " + i +": empty");
               }
             }
           }
        }
    public class linearprobe
   {
          public static void main (String[] args)
                {
 System.out.print("NAME:- KHAN TABISH MUJEEB\n");
     System.out.print("ROLL NO:- 25MCA-33\n");
               HashTable ht = new HashTable(10);
                System.out.println("intial empty hash table");
ht.display();
System.out.println("\n Inserting keey 12 ...");
```

```
ht.insert(12);
System.out.println("\n Inserting keey 22 ...");
ht.insert(22);
System.out.println("\n Inserting keey 32 ...");
ht.insert(32);
System.out.println("\n Inserting keey 42 ...");
ht.insert(42);
System.out.println("\n Inserting keey 52 ...");
ht.insert(52);
ht.display();
}
}
```

Implementation(Output Of The Code):





Advantages of Linear Probing:

- Simple and easy to implement.
- Efficient when the hash table is sparsely filled.
- Uses the entire hash table (no extra memory for pointers).

Limitations of Linear Probing:

- Primary Clustering:
  Consecutive occupied slots form clusters, increasing search time.
- Performance Degradation:
  As the load factor increases, insertion and search become slower.

Conclusion: Hashing with Linear Probing is an efficient method for resolving collisions in hash tables. It provides fast access to data when the load factor is low. However, clustering can reduce performance, making it important to choose an appropriate hash function and table size.

Practical No. 3

AIM:- Implementation of Queue Using arrays

A. Ordinary Queue &

B. Circular queue

Practical.3.a

Theory:A Queue is a linear data structure that follows the **FIFO (First In First Out)** principle. The element that is inserted first is removed first. In an Ordinary Queue, insertion takes place at the rear end, and deletion takes place from the front end. Once the rear reaches the end of the array, no new elements can be inserted even if there are empty spaces at the beginning.

Basic Operations:

- Enqueue: Insert an element into the queue
- Dequeue: Remove an element from the queue
- Peek: View the front element
- IsEmpty / IsFull: Check queue status

Algorithm for Ordinary Queue:

Enqueue Operation:

1. If rear == MAX − 1, display Queue Overflow.
2. If front == −1, set front = 0.
3. Increment rear = rear + 1.
4. Insert the element at queue[rear].

Dequeue Operation:

1. If front == −1 or front > rear, display Queue Underflow.
2. Remove the element at queue[front].
3. Increment front = front + 1.

Programming Code for Ordinary Queue:

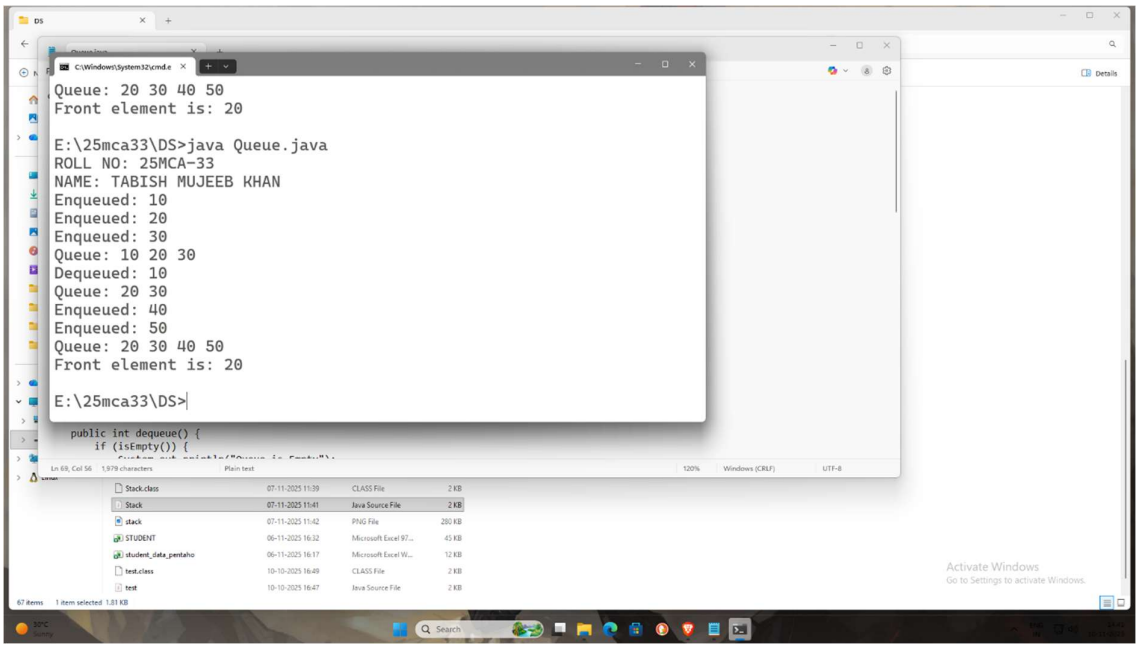
```
class Queue {
   private int[] arr;
   private int front;
   private int rear;
   private int capacity;
   private int count;
   public Queue(int size) {
      arr = new int[size];
```

```
    capacity = size;
    front = 0;
    rear = -1;
    count = 0;
}
public void enqueue(int item) {
    if (isFull()) {
        System.out.println("Queue is full");
        return;
    }
    rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
    System.out.println("Enqueued: " + item);
}
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is Empty");
        return -1;
    }
    int item = arr[front];
    front = (front + 1) % capacity;
    count--;
    System.out.println("Dequeued: " + item);
    return item;
}
public int peek() {
    if (isEmpty()) {
        System.out.println("Queue is Empty");
        return -1;
    }
    return arr[front];
```

```
    }
    public boolean isEmpty() {
        return (count == 0);
    }
    public boolean isFull() {
        return (count == capacity);
    }
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        System.out.print("Queue: ");
        for (int i = 0; i < count; i++) {
            System.out.print(arr[(front + i) % capacity] + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN");
        Queue queue = new Queue(5);
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.display();
        queue.dequeue();
        queue.display();
        queue.enqueue(40);
        queue.enqueue(50);
        queue.display();
        System.out.println("Front element is: " + queue.peek());
```

```
    }
}
```

Implementation(Output Of The Code):



Limitations of Ordinary Queue:

- ❖ Wastage of memory due to unused spaces.
- ❖ Not efficient for continuous insertions and deletions.

Practical 3.b

Theory: A Circular Queue is an improved version of the ordinary queue. The last position of the queue is connected to the first position, forming a **circular structure**.

This allows efficient utilization of memory because once the rear reaches the end, it can wrap around to the beginning if there is space.

Basic Operations:

- Enqueue
- Dequeue
- Peek
- IsEmpty / IsFull

Algorithm used for CircularQueue:

Enqueue Operation

Step 1: Check if (rear + 1) % MAX == front.
Step 2: If true, display **Queue Overflow** and exit.
Step 3: If front == −1, set front = rear = 0.
Step 4: Else set rear = (rear + 1) % MAX.
Step 5: Insert the element at Queue[rear].
Step 6: Exit.

Dequeue Operation

Step 1: Check if front == −1.
Step 2: If true, display **Queue Underflow** and exit.
Step 3: Delete the element from Queue[front].
Step 4: If front == rear, set front = rear = −1.
Step 5: Else set front = (front + 1) % MAX.
Step 6: Exit.

Time Complexity:

- Enqueue: O(1)
- Dequeue: O(1)

Programming Code for Circular Queue:

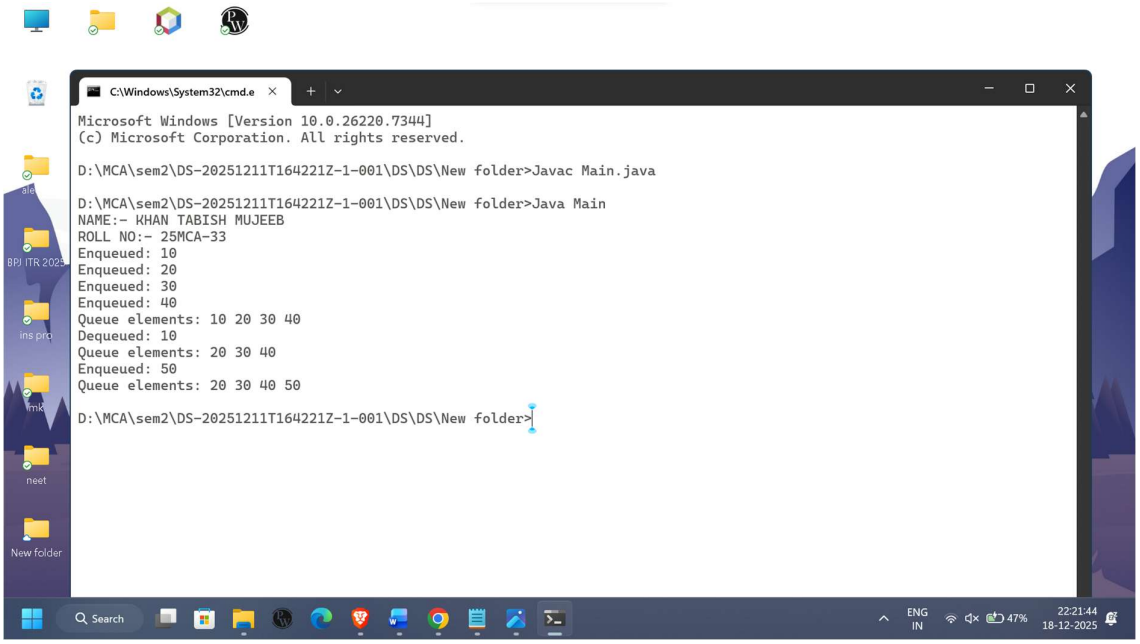
```
class CircularQueue {
    private Node rear;
    private int size;
    private static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
```

```
    }
  }
  public CircularQueue() {
    this.rear = null;
    this.size = 0;
  }
  public void enqueue(int data) {
    Node newNode = new Node(data);
    if (rear == null) {
      rear = newNode;
      rear.next = rear;
    } else {
      newNode.next = rear.next;
      rear.next = newNode;
      rear = newNode;
    }
    size++;
    System.out.println("Enqueued: " + data);
  }
  public int dequeue() {
    if (rear == null) {
      System.out.println("Queue is empty");
      return -1;
    }
    Node front = rear.next;
    int data = front.data;
    if (rear == front) {
      rear = null;
    } else {
      rear.next = front.next;
    }
    size--;
```

```
        System.out.println("Dequeued: " + data);
        return data;
    }
    public boolean isEmpty() {
        return rear == null;
    }
    public int getSize() {
        return size;
    }
    public void display() {
        if (rear == null) {
            System.out.println("Queue is empty");
            return;
        }
        Node current = rear.next;
        System.out.print("Queue elements: ");
        do {
            System.out.print(current.data + " ");
            current = current.next;
        } while (current != rear.next);
        System.out.println();
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        CircularQueue q = new CircularQueue();
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
```

```
        q.display();
        q.dequeue();
        q.display();
        q.enqueue(50);
        q.display();
    }
}
```

Implementation(Output Of The Code):



Advantages of Circular Queue:

- Efficient use of memory.
- No wastage of space.
- Suitable for buffering and scheduling applications.

Conclusion: An Ordinary Queue is simple to implement but inefficient in memory utilization. A Circular Queue overcomes this limitation by reusing vacant spaces, making it more efficient for real-time applications.

Practical No.4

AIM:- Implementation of Stack Applications like:

A. Infix to Postfix
B. Postfix evaluation
C. Balancing of Parenthesis

Practical No.4.a

A. Infix to Postfix Conversion

Theory: Infix expressions have operators **between operands** (e.g., A + B).
Postfix expressions have operators **after operands** (e.g., A B +).

Using a **stack**, we can convert an infix expression to postfix by following **operator precedence** rules.

Algorithm:

Step 1: Initialize an empty stack for operators.
Step 2: Read the infix expression from left to right.
Step 3: If the symbol is an operand, append it to the postfix expression.
Step 4: If the symbol is '(', push it onto the stack.
Step 5: If the symbol is ')', pop from the stack and append to postfix until '(' is encountered. Discard '(' from the stack.
Step 6: If the symbol is an operator, pop and append operators from the stack that have **higher or equal precedence**, then push the current operator onto the stack.
Step 7: Repeat Steps 2–6 until the end of the infix expression.
Step 8: Pop and append any remaining operators from the stack to the postfix expression.
Step 9: Exit.

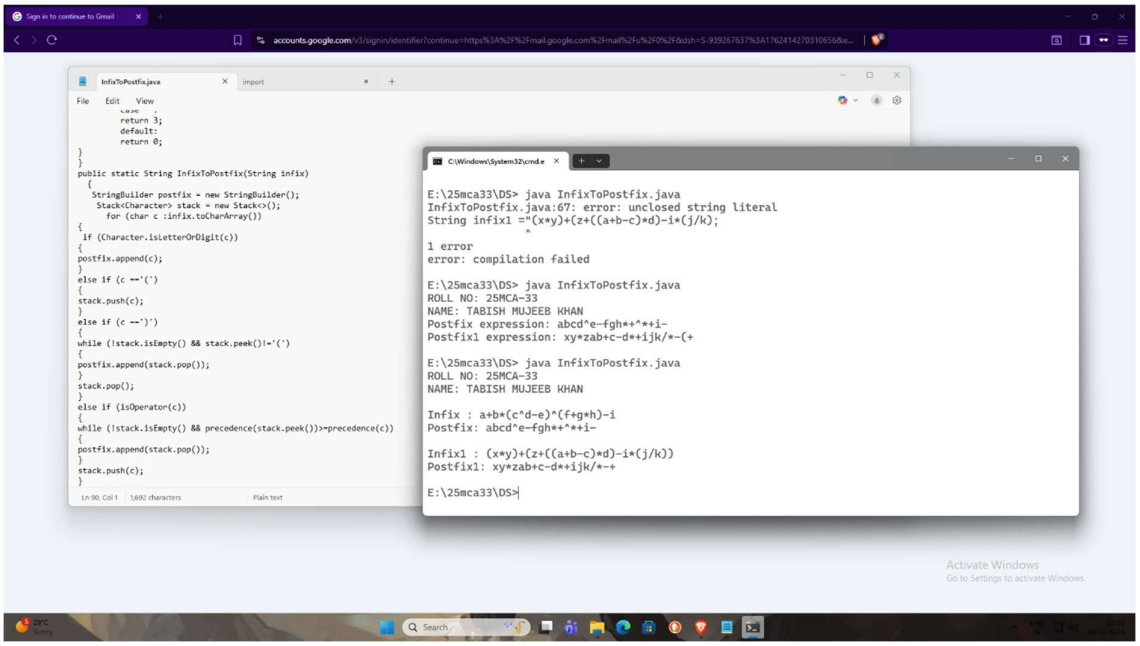
Programming Code for Infix to Postfix :

```
import java.util.*;

class InfixToPostfix

{

     private static boolean isOperator(char c)

     {

       return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';

      }

    private static int precedence(char op)

    {

     switch(op)

     {

       case'+':

       case'-':

       return 1;
```

```
        case'*':
        case'/':
        return 2;
        case'^':
        return 3;
        default:
        return 0;
}
}
public static String InfixToPostfix(String infix)
  {
   StringBuilder postfix = new StringBuilder();
    Stack<Character> stack = new Stack<>();
     for (char c :infix.toCharArray())
{
 if (Character.isLetterOrDigit(c))
{
postfix.append(c);
}
else if (c =='(')
{
stack.push(c);
}
else if (c ==')')
{
while (!stack.isEmpty() && stack.peek()!='(')
{
postfix.append(stack.pop());
}
stack.pop();
}
else if (isOperator(c))
```

```
{
while (!stack.isEmpty() && precedence(stack.peek())>=precedence(c))
{
postfix.append(stack.pop());
}
stack.push(c);
}
}
while(!stack.isEmpty())
{
postfix.append(stack.pop());
}
return postfix.toString();
}
public static void main(String[] args){
  System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN\n");
        String infix = "a+b*(c^d-e)^(f+g*h)-i";
        String infix1 = "(x*y)+(z+((a+b-c)*d)-i*(j/k))";
        String postfix = InfixToPostfix(infix);
        String postfix1 = InfixToPostfix(infix1);
        System.out.println("Infix : " + infix);
        System.out.println("Postfix: " + postfix);
        System.out.println("\nInfix1 : " + infix1);
        System.out.println("Postfix1: " + postfix1);
}
}
```

Implementation(Output Of The Code):



Practical No.4.b

B. Postfix Evaluation

Theory: Postfix expressions can be evaluated using a **stack**. Operands are pushed onto the stack, and operators pop the required number of operands, perform the operation, and push the result back.

Algorithm:

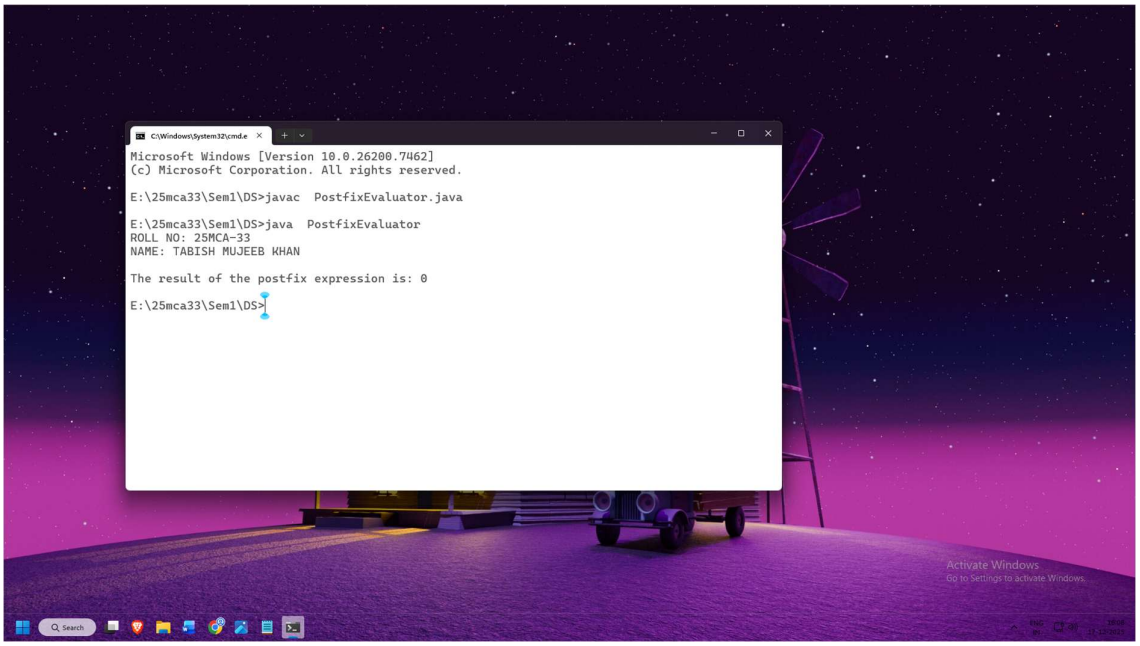
Step 1: Initialize an empty stack for operands.
Step 2: Read the postfix expression from left to right.
Step 3: If the symbol is an operand, push it onto the stack.
Step 4: If the symbol is an operator, pop the required operands from the stack.
Step 5: Perform the operation and push the result back onto the stack.
Step 6: Repeat Steps 2–5 until the end of the expression.
Step 7: The value remaining on the stack is the result.
Step 8: Exit.

Programming Code for Postfix Evaluation:

```
import java.util.Stack;

class PostfixEvaluator

{
        public static int evaluatePostfix(String expression)
        {
                Stack<Integer> stack = new Stack<>();
                for (char ch : expression.toCharArray())
                {
                        if (Character.isDigit(ch))
                        {
                                stack.push(ch - '0');
                        }
                        else
                        {
                                int b = stack.pop();
                                int a = stack.pop();
                                int result = 0;
                                switch (ch)
                                {
                                 case '+':
                                        result = a + b;
```

```
                                        break;
                                case '-':
                                        result = a - b;
                                        break;
                                case '*':
                                        result = a * b;
                                        break;
                                case '/':
                                        result = a / b;
                                        break;
                                }
                                stack.push(result);
                        }
                }
                return stack.pop();
        }
        public static void main(String[] args)
        {
                 System.out.println("ROLL NO: 25MCA-33");
      System.out.println("NAME: TABISH MUJEEB KHAN\n");
                String postfixExpression = "23*54+9-";
                int result = evaluatePostfix(postfixExpression);
                System.out.println("The result of the postfix expression is: " + result);
        }
}
```

Implementation(Output Of The Code):



Practical No.4.c

C. Balancing of Parentheses

Theory: A stack can be used to check whether an expression has **balanced parentheses**. Every opening bracket '(' must have a corresponding closing bracket ')'.

Algorithm:

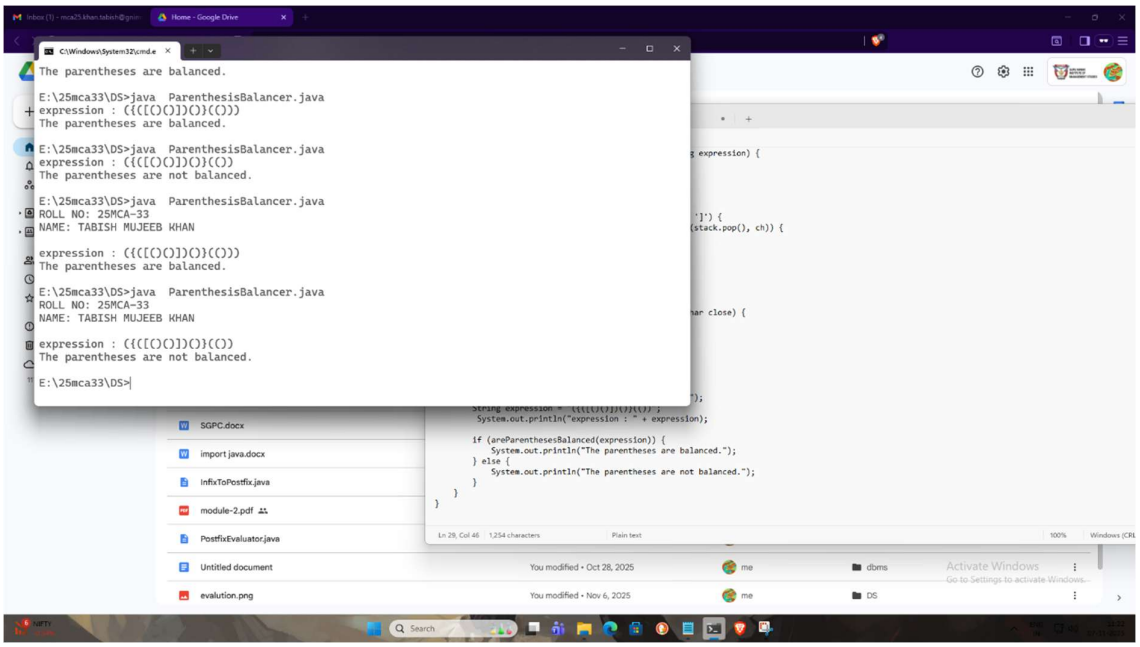
Step 1: Initialize an empty stack.
Step 2: Read the expression from left to right.
Step 3: If the symbol is an opening bracket '(', push it onto the stack.
Step 4: If the symbol is a closing bracket ')', check if the stack is empty.
  Step 4a: If empty, expression is **unbalanced**.
  Step 4b: If not empty, pop from the stack.
Step 5: Repeat Steps 2–4 until the end of the expression.
Step 6: If the stack is empty after processing, the expression is **balanced**; otherwise, it is **unbalanced**.
Step 7: Exit.

Programming Code for Balancing of Parenthesis:

```
import java.util.Stack;

class ParenthesisBalancer {

   public static boolean areParenthesesBalanced(String expression) {

      Stack<Character>stack = new Stack<>();

      for (char ch : expression.toCharArray()) {

         if (ch == '(' || ch == '{' || ch == '[') {

            stack.push(ch);

         } else if (ch == ')' || ch == '}' || ch == ']') {

            if (stack.isEmpty() || !isMatchingPair(stack.pop(), ch)) {

               return false;

            }

         }

      }

      return stack.isEmpty();

   }

   private static boolean isMatchingPair(char open, char close) {

      return (open == '(' && close == ')') ||

            (open == '{' && close == '}') ||

            (open == '[' && close == ']');

   }
```

```
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN\n");
        String expression = "({([()()])()}(()))";
        System.out.println("expression : " + expression);
        if (areParenthesesBalanced(expression)) {
            System.out.println("The parentheses are balanced.");
        } else {
            System.out.println("The parentheses are not balanced.");
        }
    }
}
```

Implementation(Output Of The Code):



Practical No.4.d

Aim: To implement **Infix to Prefix conversion** using **Stack**.

Theory: Infix notation is the common arithmetic expression format where operators are placed **between operands**
Example: A + B

Prefix notation (also called Polish notation) places the operator **before operands**
Example: + A B

Computers find it easier to evaluate prefix expressions because **operator precedence and parentheses are not required**.

Rules for Infix to Prefix Conversion:

1. Reverse the infix expression.
2. Replace (with) and vice versa.
3. Convert the modified expression to **Postfix**.
4. Reverse the postfix expression to get **Prefix**.

Algorithm for Infix to Prefix Conversion:

Step 1: Read the infix expression
Step 2: Reverse the infix expression
Step 3: Replace (with) and) with (
Step 4: Initialize an empty stack
Step 5: Scan the expression from left to right

- If operand → add to result
- If (→ push to stack
- If) → pop from stack until (is found
- If operator →
    - Pop operators from stack with higher precedence
    - Push current operator

Step 6: Pop remaining operators from stack
Step 7: Reverse the result to get prefix expression
Step 8: Print the prefix expression

Time and Space Complexity:

Time Complexity:

- **O(n)** — each character is processed once

Space Complexity:

- **O(n)** — stack and result storage

Programming Code for Infix to Prefix :

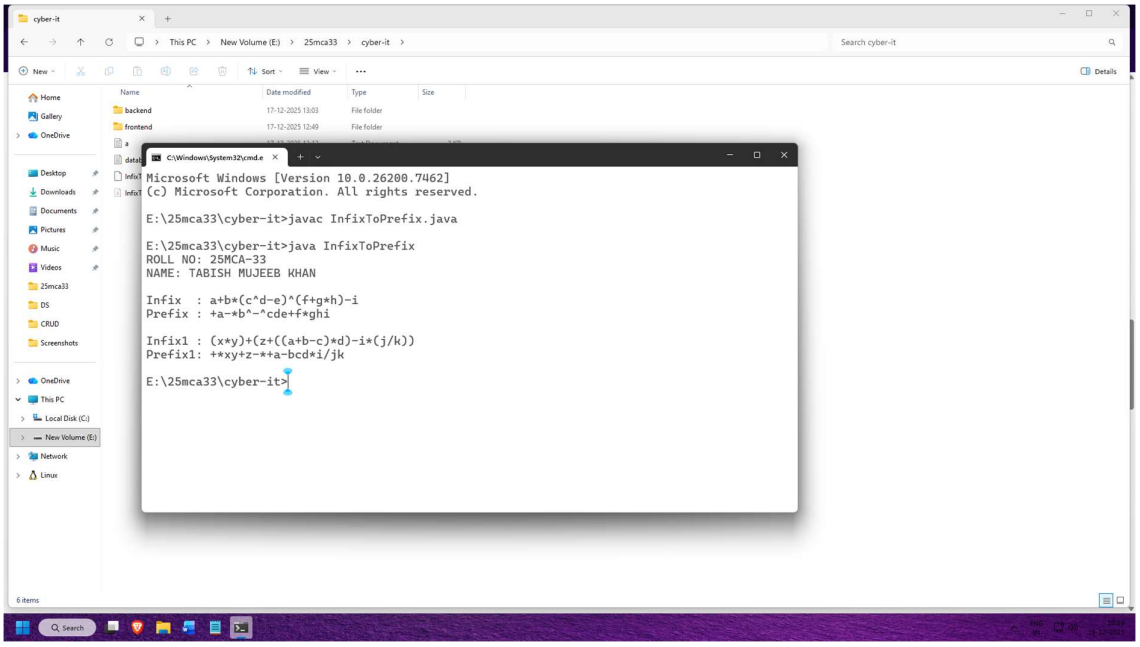
```
import java.util.*;
```

```
class InfixToPrefix {
    private static boolean isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
    }
    private static int precedence(char op) {
        switch (op) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            case '^':
                return 3;
            default:
                return 0;
        }
    }
    public static String infixToPrefix(String infix) {
        StringBuilder reversed = new StringBuilder(infix).reverse();
        for (int i = 0; i < reversed.length(); i++) {
            if (reversed.charAt(i) == '(')
                reversed.setCharAt(i, ')');
            else if (reversed.charAt(i) == ')')
                reversed.setCharAt(i, '(');
        }
        String postfix = infixToPostfix(reversed.toString());
        return new StringBuilder(postfix).reverse().toString();
    }
    private static String infixToPostfix(String infix) {
        StringBuilder postfix = new StringBuilder();
        Stack<Character> stack = new Stack<>();
```

```
        for (char c : infix.toCharArray()) {
            if (Character.isLetterOrDigit(c)) {
                postfix.append(c);
            }
            else if (c == '(') {
                stack.push(c);
            }
            else if (c == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    postfix.append(stack.pop());
                }
                stack.pop();
            }
            else if (isOperator(c)) {
                while (!stack.isEmpty() &&
                    (precedence(stack.peek()) > precedence(c) ||
                    (precedence(stack.peek()) == precedence(c) && c != '^'))) {
                    postfix.append(stack.pop());
                }
                stack.push(c);
            }
        }
        while (!stack.isEmpty()) {
            postfix.append(stack.pop());
        }
        return postfix.toString();
    }
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN\n");
        String infix = "a+b*(c^d-e)^(f+g*h)-i";
        String infix1 = "(x*y)+(z+((a+b-c)*d)-i*(j/k))";
```

```
        System.out.println("Infix  : " + infix);
        System.out.println("Prefix : " + infixToPrefix(infix));
        System.out.println("\nInfix1 : " + infix1);
        System.out.println("Prefix1: " + infixToPrefix(infix1));
    }
}
```

Implementation(Output Of The Code):



Limitations of Infix to Prefix Conversion:

1. Complex Implementation

Compared to linear search, infix to prefix conversion requires **stack handling and precedence rules**, making it harder to implement.

2. Difficult for Beginners

Understanding operator precedence, associativity, and stack operations can be confusing for new learners.

Conclusion: The above experiments helped in understanding the practical application of **stack data structure** in solving various expression-related problems. Converting **infix expressions to postfix and prefix** simplifies expression evaluation by eliminating the need for parentheses and operator precedence rules. **Postfix evaluation** demonstrates how stacks are used to evaluate expressions efficiently, while **balancing of parentheses** ensures the correctness of expressions in programming and mathematical computations. These experiments highlight the importance of stacks in compiler design and expression processing. Although stack-based algorithms are easy to understand and implement, careful handling of operators and operands is required. Overall, these experiments strengthen the understanding of stack operations and their real-world applications in data structures.

PRACTICAL 5

Problem Statement:

Aim:- Implementation of various Linked List

a. Singly Linked List

b. Doubly Linked List

c. Header Linked List

1. Grounded Header Linked List

2.Two-way Header Linked List

3.Circular Header Linked List

4.Circular Two-way Header Linked List

d. Doubly Ended Linked List

Practical 5.a

Aim: Implementing various operations on singly linked list

Theory: A Singly Linked List is a linear data structure in which elements, called nodes, are connected sequentially using pointers.

Each node contains two parts:

1. Data – stores the actual information.

2. Next – a pointer (or reference) to the next node in the list.

Algorithm for Singly Linked List:

- ❖ Algorithm To Traverse A Singly Linked List

Step 1: Start

Step 2: Set temp = head

Step 3: Repeat while temp != NULL

Print temp -> data

Set temp = temp ->next

Step 4: Exit

- ❖ Algorithm To Insert A Node At The Beginning

Step 1: Start

Step 2: Create a new node and set newnode -> data = value

Step 3: Set newnode -> next = head

Step 4: Set head = newnode

Step 5: Exit

❖ Algorithm To Insert A Node At The End

Step 1: Start

Step 2: Create a new node and set newnode -> data = value and newnode -> next = NULL

Step 3: If head == NULL,

set head = newnode

[Exit]

Step 4: Else set temp = head

Step 5: Repeat while temp -> next != null

Set temp = temp -> next

Step 6: Set temp -> next = newnode

Step 7: Exit

❖ Algorithm To Delete A Node From Beginning

Step 1: Start

Step 2: If head == NULL ,

print “List is empty”

Exit

Step 3: Set temp = head

Step 4: Set head = head ->

Step 5: Free temp

Step 6: Exit

❖ Algorithm To Delete A Node From The End

Step 1: Start

Step 2: If head == NULL,

Print”List is empty”

Exit

Step 3: If head -> next == NULL, free head and set head = NULL

Step 4: Else set temp = head

Step 5: Repeat while temp -> next -> next != NULL

Set temp = temp -> next

Step 6: Free temp -> next and set temp -> next = NULL

Step 7: Exit

Time And Space Complexity

Time Complexity:

- Best: O(1) (insert/delete at beginning)
- Worst: O(n) (traverse/search)

Space Complexity:

O(n) (for storing n nodes)

Programming for Singly Linked List:

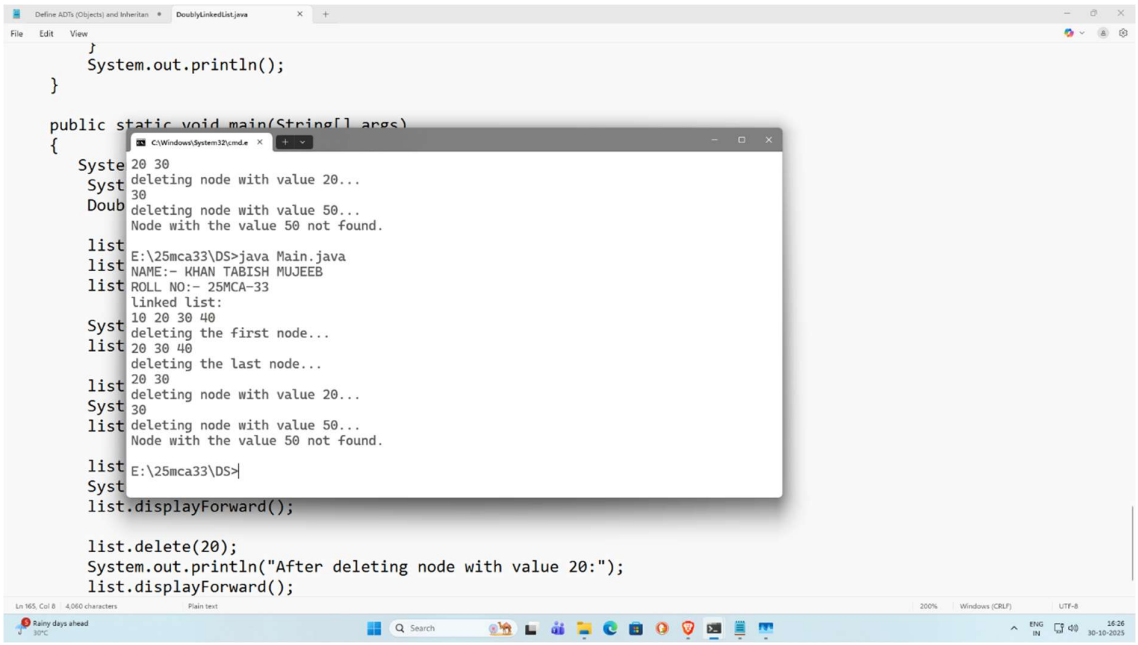
```
class Node {
   int data;
   Node next;
   public Node(int data) {
      this.data = data;
      this.next = null;
   }
}
class LinkedList {
   Node head;
   public LinkedList() {
      head = null;
   }
   public void insert(int data) {
      Node newNode = new Node(data);
      if (head == null) {
         head = newNode;
      } else {
         Node temp = head;
         while (temp.next != null) {
            temp = temp.next;
         }
         temp.next = newNode;
      }
   }
   public void display() {
      if (head == null) {
```

```
            System.out.println("The list is empty.");
            return;
        }
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
    public void deleteFirst() {
        if (head == null) {
            System.out.println("This list is empty. No node to delete.");
            return;
        }
        head = head.next;
    }
    public void deleteLast() {
        if (head == null) {
            System.out.println("The list is empty. No node to delete.");
            return;
        }
        if (head.next == null) {
            head = null;
            return;
        }
        Node temp = head;
        while (temp.next != null && temp.next.next != null) {
            temp = temp.next;
        }
        temp.next = null;
    }
```

```
    public void delete(int value) {
        if (head == null) {
            System.out.println("The list is empty. No node to delete.");
            return;
        }
        if (head.data == value) {
            head = head.next;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            if (temp.next.data == value) {
                temp.next = temp.next.next;
                return;
            }
            temp = temp.next;
        }
        System.out.println("Node with the value " + value + " not found.");
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        LinkedList list = new LinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.insert(40);
        System.out.println("linked list:");
        list.display();
        System.out.println("deleting the first node...");
```

```
        list.deleteFirst();
        list.display();
        System.out.println("deleting the last node...");
        list.deleteLast();
        list.display();
        System.out.println("deleting node with value 20...");
        list.delete(20);
        list.display();
        System.out.println("deleting node with value 50...");
        list.delete(50);
    }
}
```

Implementation(Output Of The Code):



Limitations Of Singly Linked List:

- One-way traversal only: Nodes can only be traversed in the forward direction since each node points to the next one only. Backward traversal is not possible without extra data structures or reversing the list.
- Higher memory usage: Each node needs extra memory for a pointer (reference to the next node).Compared to arrays, this increases overhead, especially for small data elements.

Conclusion: A singly linked list is a fundamental dynamic data structure that allows efficient insertion and deletion operations, especially at the beginning of the list. It is simple to implement and ideal for situations where memory reallocation (as in arrays) is undesirable.

Practical 5.b

Aim: Implementing various operations on doubly linked list

Theory: A doubly linked list (DLL) is a type of linked data structure in which each node contains three components:

- Data – The actual value stored in the node.
- Next pointer – A reference (or link) to the next node in the list.
- Previous pointer – A reference (or link) to the previous node in the list.

Because of these two pointers, a doubly linked list allows traversal in both directions — forward (using the next pointer) and backward (using the prev pointer).

Algorithm For Doubly Linked List

- Algorithm To Traverse A Doubly Linked List:

Step 1: If End = NULL Then

Print:"Linked List is empty"

Exit [End If]

Step 2: Set Pointer = End

Step 3: Repeat while Pointer != NULL

Process Pointer-> Info

Set Pointer = Pointer -> Pre [End Loop]

Step 4: Exit

- Algorithm To Insert A Node In The Beginning Of The List

Step 1: Create a new node NEW.

Step 2: Set NEW -> INFO = ITEM.

Step 3: Set NEW -> PREV = NULL.

Step 4: Set NEW -> NEXT = START.

Step 5: If START != NULL Then

Set START -> PREV = NEW

[End If]

Step 6: Set START = NEW.

Step 7: Exit.

- Algorithm To Insert A Node At The End Of The List:

Step 1: Create a new node NEW.

Step 2: Set NEW -> INFO = ITEM.

Step 3: Set NEW -> NEXT = NULL.

Step 4: If START = NULL Then

Set NEW -> PREV = NULL Set START = NEW

Exit

[End If]

Step 5: Set PTR = START.

Step 6: Repeat while PTR -> NEXT != NULL Set PTR = PTR -> NEXT

[End Loop]

Step 7: Set PTR -> NEXT = NEW. Step 8: Set NEW -> PREV = PTR.

Step 9: Exit.

Programming for Doubly Linked List:

```
class Node
{
   int data;
   Node next;
   Node prev;
   public Node(int data)
   {
      this.data = data;
      this.next = null;
      this.prev = null;
   }
}
class DoublyLinkedList
{
   Node head;
   Node tail;
   public DoublyLinkedList()
   {
      head = null;
      tail = null;
   }
   public void insertAtBeginning(int data)
   {
```

```
    Node newNode = new Node(data);
    if (head == null)
    {
        head = tail = newNode;
    }
    else
    {
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }
}
public void insertAtEnd(int data)
{
    Node newNode = new Node(data);
    if (head == null)
    {
        head = tail = newNode;
    }
    else
    {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    }
}
public void deleteFirst()
{
    if (head == null)
    {
        System.out.println("The list is empty.");
        return;
```

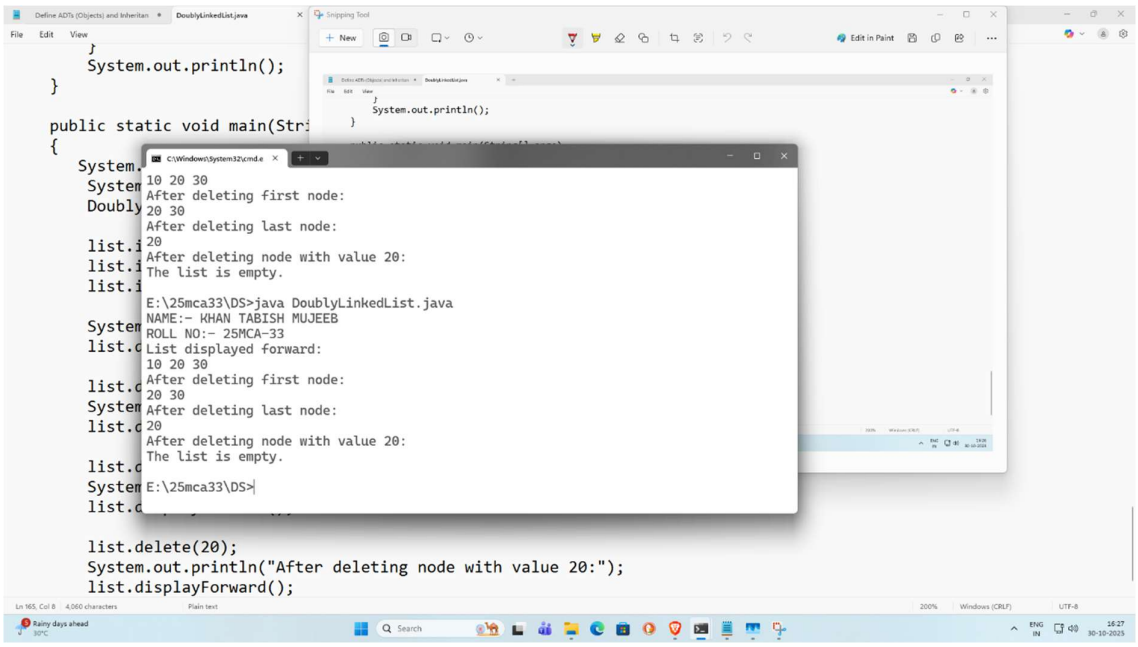
```
    }
    if (head == tail)
    {
        head = tail = null;
    }
    else
    {
        head = head.next;
        head.prev = null;
    }
}
public void deleteLast()
{
    if (head == null)
    {
        System.out.println("The list is empty.");
        return;
    }
    if (head == tail)
    {
        head = tail = null;
    }
    else
    {
        tail = tail.prev;
        tail.next = null;
    }
}
public void delete(int value)
{
    if (head == null)
    {
```

```
            System.out.println("The list is empty.");
            return;
        }
        Node temp = head;
        while (temp != null)
        {
            if (temp.data == value)
            {
                if (temp == head)
                {
                    deleteFirst();
                }
                else if (temp == tail)
                {
                    deleteLast();
                }
                else
                {
                    temp.prev.next = temp.next;
                    if (temp.next != null)
                    {
                        temp.next.prev = temp.prev;
                    }
                }
                return;
            }
            temp = temp.next;
        }
        System.out.println("Node with value " + value + " not found.");
    }
    public void displayForward()
    {
```

```
    if (head == null)
    {
        System.out.println("The list is empty.");
        return;
    }
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}
public void displayBackward()
{
    if (tail == null)
    {
        System.out.println("The list is empty.");
        return;
    }
    Node temp = tail;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.prev;
    }
    System.out.println();
}
public static void main(String[] args)
{
   System.out.print("NAME:- KHAN TABISH MUJEEB\n");
    System.out.print("ROLL NO:- 25MCA-33\n");
```

```
        DoublyLinkedList list = new DoublyLinkedList();
        list.insertAtBeginning(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
        System.out.println("List displayed forward:");
        list.displayForward();
        list.deleteFirst();
        System.out.println("After deleting first node:");
        list.displayForward();
        list.deleteLast();
        System.out.println("After deleting last node:");
        list.displayForward();
        list.delete(20);
        System.out.println("After deleting node with value 20:");
        list.displayForward();
    }
}
```

Implementation(Output Of The Code):



Practical 5.c.1

Aim :- To implement a **Grounded Header Linked List** and perform basic operations on it.

Theory :- A **Grounded Header Linked List** is a linked list that contains a **header node** at the beginning. The header node does **not store actual data**; it only stores information such as the address of the first node. The last node of the list points to **NULL**, hence it is called *grounded*.

This type of list simplifies insertion and deletion operations because special cases (empty list) are easier to handle.

Algorithm for Insertion in Grounded Header Linked List :-

Step 1: Create a header node and set its link to NULL
Step 2: Create a new node
Step 3: Store data in the new node
Step 4: Set new node's link to header → link
Step 5: Update header → link to new node
Step 6: Exit

Time And Space Complexity :-

Time Complexity :-

- Insertion at beginning: **O(1)**
- Traversal: **O(n)**

Space Complexity :-

- **O(n)** — memory required for n nodes

Programming for Grounded Header Linked List:

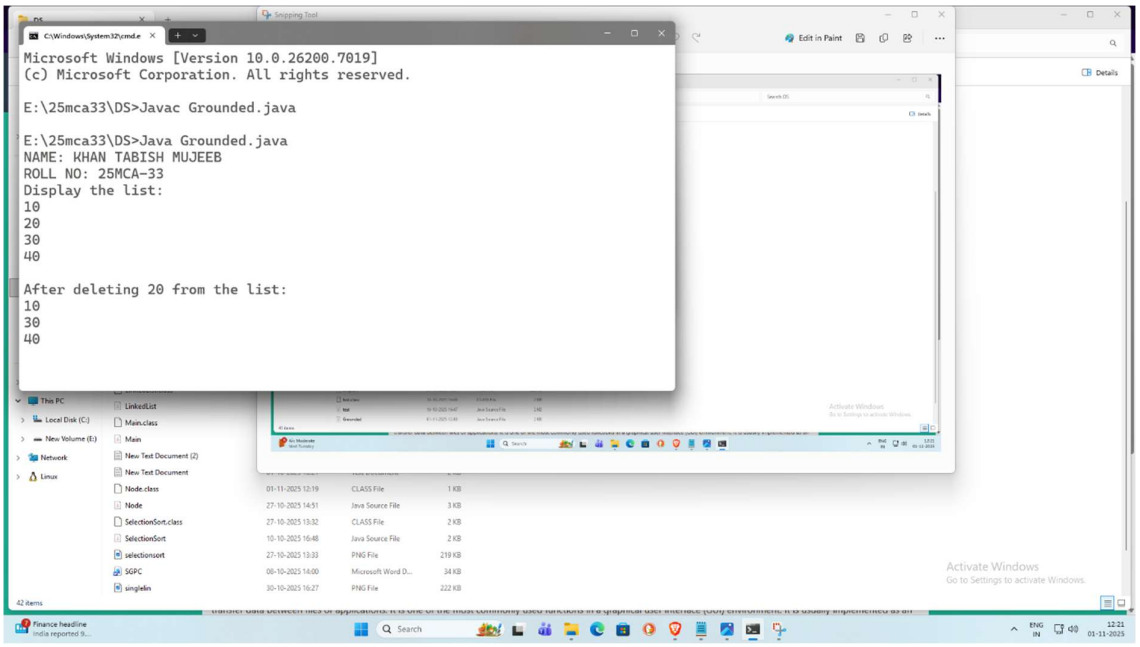
```
class Node
{
	int data;
	Node next;
	public Node(int data)
	{
		this.data = data;
		this.next = null;
	}
}
class HeaderLinkedList1
{
	Node header;
	public HeaderLinkedList1()
```

```
{
        header = new  Node(-1);
}
public void insert(int data)
{
        Node newNode = new Node(data);
        Node current = header;
        while (current.next != null)
        {
                current = current.next;
        }
        current.next = newNode;
}
public void delete(int data)
{
        Node current = header;
        while(current.next != null && current.next.data != data)
        {
                current = current.next;
        }
        if(current.next != null)
        {
                current.next = current.next.next;
        }
        else
        {
                System.out.println("element not found");
        }
}
public void display()
{
        Node current = header.next;
```

```
                if (current == null)
                {
                        System.out.println("List is empty");
                        return;
                }
                while( current != null)
                {
                        System.out.println(current.data +" ");
                        current = current.next;
                }
                System.out.println();
        }
        public boolean search(int data)
        {
                Node current = header.next;
                while(current != null)
                {
                        if(current.data == data)
                        {
                                return true;
                        }
                        current = current.next;
                }
                return false;
        }
}
public class Grounded
{
        public static void main(String[] args)
        {
                System.out.println("NAME: KHAN TABISH MUJEEB");
          System.out.println("ROLL NO: 25MCA-33");
```

```
        HeaderLinkedList1 list = new HeaderLinkedList1();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.insert(40);
        System.out.println("Display the list:");
        list.display();
        System.out.println("After deleting 20 from the list:");
        list.delete(20);
        list.display();
    }
}
```

Implementation(Output Of The Code):



Limitations of Grounded Header Linked List :-

1. Extra memory is required for the header node
2. Traversal is slower compared to arrays
3. Random access is not possible

Conclusion :- Grounded Header Linked List simplifies insertion and deletion operations by using a header node. It is useful when frequent modifications are required, but it is inefficient for direct access and large datasets.

Practical 5.c.2

Aim :-To implement a **Two-Way Header Linked List** and perform traversal operations.

Theory :- A **Two-Way Header Linked List** (Doubly Linked List) contains a **header node** and each node has **two links**:
• One pointing to the previous node
• One pointing to the next node

This structure allows traversal in **both forward and backward directions**, making it more flexible than a singly linked list.

Algorithm for Insertion in Two-Way Header Linked List :-

Step 1: Create a header node
Step 2: Create a new node
Step 3: Store data in the new node
Step 4: Set new node's next to header → next
Step 5: Set new node's previous to header
Step 6: Update header → next and previous links
Step 7: Exit

Time And Space Complexity :-

Time Complexity :-

- Insertion: **O(1)**
- Traversal: **O(n)**

Space Complexity :-

- **O(n)** — extra memory for previous and next pointers

Programming Code for Two-Way Header Linked List

```
class HeaderLinkedList {
   Node header;
   Node tail;
   public HeaderLinkedList() {
      header = new Node(-1);
      tail = null;
      header.next = null;
      header.prev = null;
   }
   public void insertAtBeginning(int data) {
      Node newNode = new Node(data);
      if (header.next == null) {
         header.next = newNode;
```

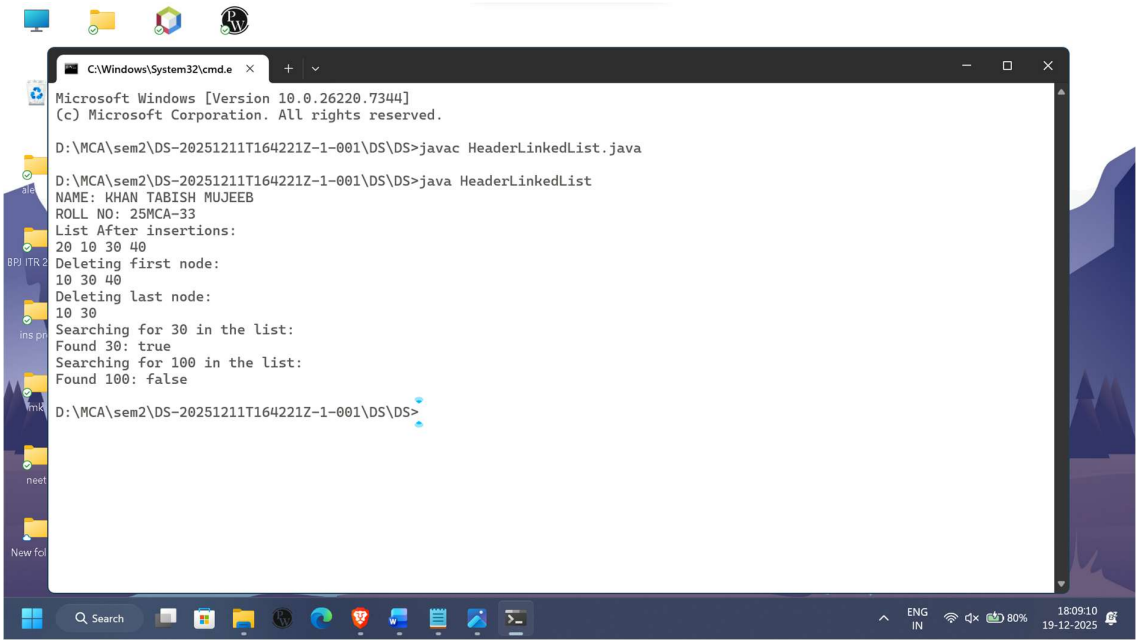
```
            newNode.prev = header;
            tail = newNode;
        } else {
            newNode.next = header.next;
            header.next.prev = newNode;
            header.next = newNode;
            newNode.prev = header;
        }
    }
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (header.next == null) {
            header.next = newNode;
            newNode.prev = header;
            tail = newNode;
        } else {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        }
    }
    public void deleteFirst() {
        if (header.next == null) {
            System.out.println("The list is empty.");
            return;
        }
        Node firstNode = header.next;
        header.next = firstNode.next;
        if (header.next != null) {
            header.next.prev = header;
        } else {
            tail = null;
```

```
    }
  }
  public void deleteLast() {
    if (header.next == null) {
      System.out.println("The list is empty.");
      return;
    }
    if (header.next == tail) {
      header.next = null;
      tail = null;
      return;
    }
    tail = tail.prev;
    tail.next = null;
  }
  public void display() {
    if (header.next == null) {
      System.out.println("The list is empty.");
      return;
    }
    Node temp = header.next;
    while (temp != null) {
      System.out.print(temp.data + " ");
      temp = temp.next;
    }
    System.out.println();
  }
  public boolean search(int value) {
    Node temp = header.next;
    while (temp != null) {
      if (temp.data == value) {
        return true;
```

```
            }
            temp = temp.next;
        }
        return false;
    }
    class Node {
        int data;
        Node next;
        Node prev;
        public Node(int data) {
            this.data = data;
            this.next = null;
            this.prev = null;
        }
    }
    public static void main(String[] args) {
        System.out.println("NAME: KHAN TABISH MUJEEB");
        System.out.println("ROLL NO: 25MCA-33");
        HeaderLinkedList list = new HeaderLinkedList();
        list.insertAtBeginning(10);
        list.insertAtBeginning(20);
        list.insertAtEnd(30);
        list.insertAtEnd(40);
        System.out.println("List After insertions:");
        list.display();
        System.out.println("Deleting first node:");
        list.deleteFirst();
        list.display();
        System.out.println("Deleting last node:");
        list.deleteLast();
        list.display();
        System.out.println("Searching for 30 in the list:");
```

```
        System.out.println("Found 30: " + list.search(30));
        System.out.println("Searching for 100 in the list:");
        System.out.println("Found 100: " + list.search(100));
    }
}
```

Implementation(Output Of The Code):



Limitations of Two-Way Header Linked List :-

1. Requires more memory than singly linked list
2. Implementation is more complex
3. Extra pointer increases overhead

Conclusion :-

Two-Way Header Linked List allows efficient forward and backward traversal and simplifies deletion operations. However, it consumes more memory and is more complex compared to singly linked lists.

Practical.5.c.3

Aim:- To study and understand the working of a **Circular Header Linked List**.

Theory :- A **Circular Header Linked List** is a linked list that contains a **header node** and the last node of the list points back to the **header node instead of NULL**. The header node does not store actual data but is used to store information about the list such as the starting address. Since the list is circular, traversal can continue from the last node back to the header node, making it useful for applications that require repeated cycling through data.

Algorithm for Traversal of Circular Header Linked List :-

Step 1: Start from the node pointed by header → next
Step 2: Visit the current node and display its data
Step 3: Move to the next node
Step 4: Repeat Steps 2 and 3 until the header node is reached again
Step 5: Stop traversal
Step 6: Exit

Time And Space Complexity :-

Time Complexity :-

- Traversal: **O(n)**
- Insertion/Deletion: **O(1)** (at beginning)

Space Complexity :-

- **O(n)** — memory required for n nodes

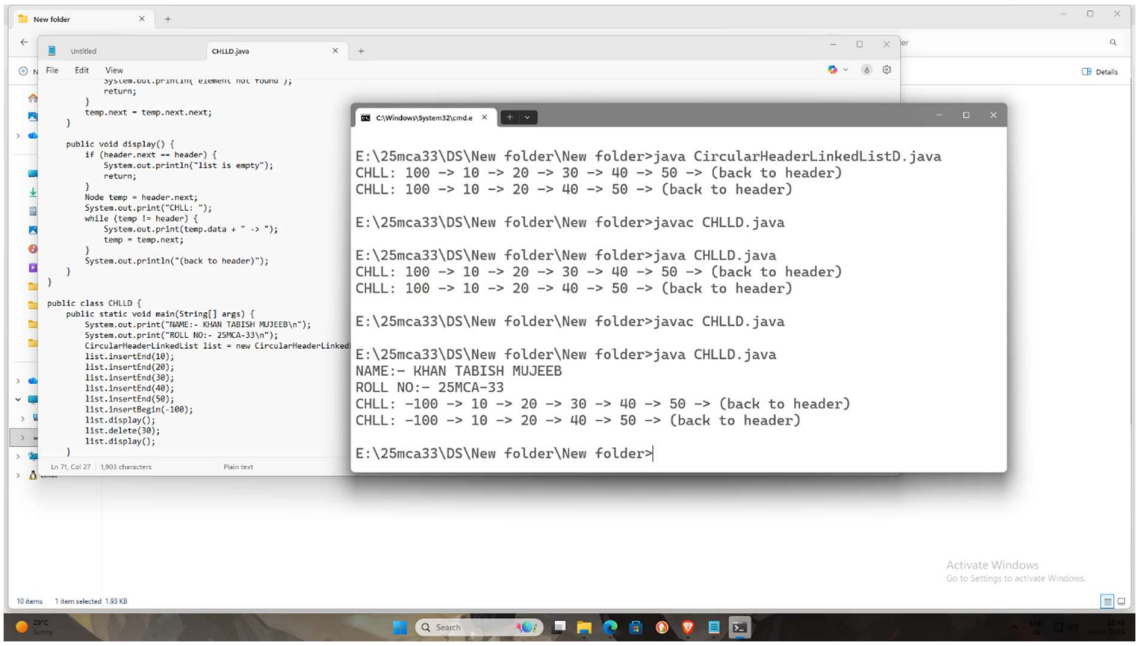
Programming for Circular Header Linked List:

```
class Node {
   int data;
   Node next;
   Node(int data) {
      this.data = data;
      this.next = null;
   }
}
class CircularHeaderLinkedList {
   Node header;
   public CircularHeaderLinkedList() {
      header = new Node(-1);
      header.next = header;
   }
   public void insertEnd(int data) {
```

```
        Node newNode = new Node(data);
        Node temp = header;
        while (temp.next != header) {
            temp = temp.next;
        }
        temp.next = newNode;
        newNode.next = header;
    }
    public void insertBegin(int data) {
        Node newNode = new Node(data);
        newNode.next = header.next;
        header.next = newNode;
    }
    public void delete(int key) {
        Node temp = header;
        while (temp.next != header && temp.next.data != key) {
            temp = temp.next;
        }
        if (temp.next == header) {
            System.out.println("element not found");
            return;
        }
        temp.next = temp.next.next;
    }
    public void display() {
        if (header.next == header) {
            System.out.println("list is empty");
            return;
        }
        Node temp = header.next;
        System.out.print("CHLL: ");
        while (temp != header) {
```

```
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("(back to header)");
    }
}
public class CHLLD {
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        CircularHeaderLinkedList list = new CircularHeaderLinkedList();
        list.insertEnd(10);
        list.insertEnd(20);
        list.insertEnd(30);
        list.insertEnd(40);
        list.insertEnd(50);
        list.insertBegin(-100);
        list.display();
        list.delete(30);
        list.display();
    }
}
```

Implementation(Output Of The Code):



Limitations of Circular Header Linked List :-

1. Implementation is more complex than linear linked lists
2. Infinite loop may occur if traversal condition is not handled properly
3. Extra memory is required for the header node

Conclusion :-

Circular Header Linked List efficiently handles continuous traversal and simplifies list operations using a header node. However, care must be taken during traversal to avoid infinite loops.

Practical5.c.4

Aim :- To study and understand the working of a **Circular Two-Way Header Linked List**.

Theory :- A **Circular Two-Way Header Linked List** (Circular Doubly Linked List with Header) contains a header node and each node has **two pointers**:
• One pointing to the previous node
• One pointing to the next node

The last node's next pointer points to the header node, and the header's previous pointer points to the last node, forming a complete circular structure. This allows traversal in both **forward and backward directions**.

Algorithm for Traversal of Circular Two-Way Header Linked List :-

Step 1: Start from header → next
Step 2: Display the data of the current node
Step 3: Move to the next node
Step 4: Repeat Steps 2 and 3 until the header node is reached again
Step 5: Exit

Time And Space Complexity :-

Time Complexity :-

- Traversal: **O(n)**
- Insertion/Deletion: **O(1)**

Space Complexity :-

- **O(n)** — extra memory for two pointers per node

Programming for Circular Two-Way Header Linked List:

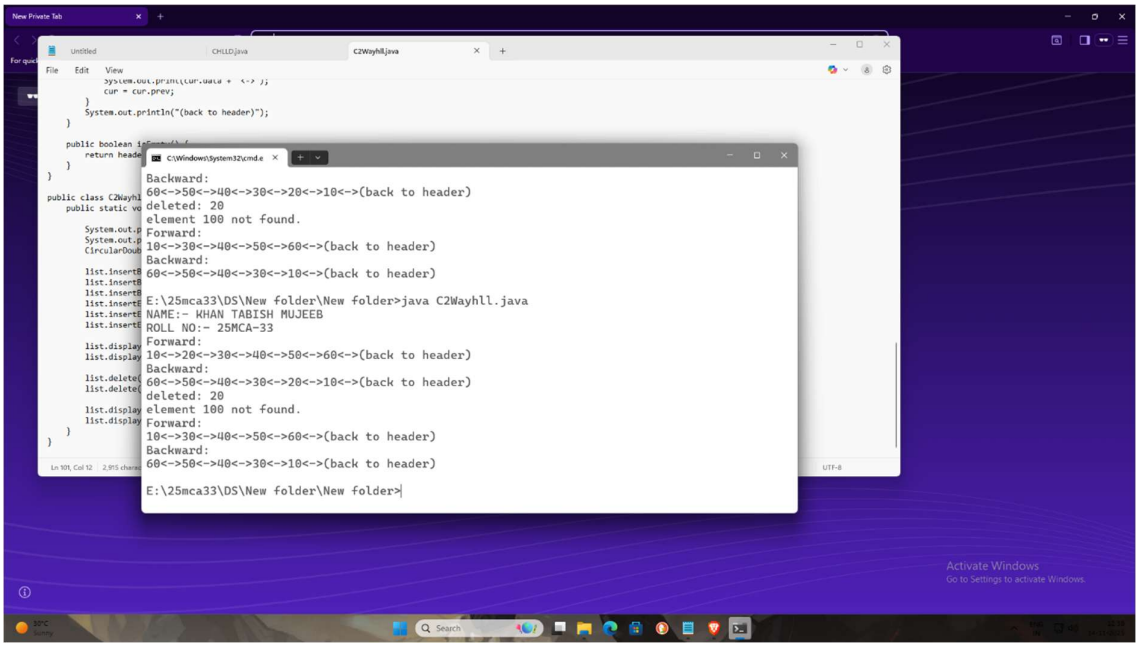
```
class Node {
   int data;
   Node next;
   Node prev;
   Node(int data) {
      this.data = data;
      this.next = null;
      this.prev = null;
   }
}
class CircularDoublyHeaderList {
   private Node header;
   public CircularDoublyHeaderList() {
      header = new Node(-1);
```

```
        header.next = header;
        header.prev = header;
    }
    public void insertBegin(int data) {
        Node first = header.next;
        Node newNode = new Node(data);
        newNode.next = first;
        newNode.prev = header;
        header.next = newNode;
        first.prev = newNode;
    }
    public void insertEnd(int data) {
        Node last = header.prev;
        Node newNode = new Node(data);
        newNode.next = header;
        newNode.prev = last;
        last.next = newNode;
        header.prev = newNode;
    }
    public void delete(int key) {
        if (header.next == header) {
            System.out.println("list is empty. cannot delete " + key);
            return;
        }
        Node cur = header.next;
        while (cur != header && cur.data != key) {
            cur = cur.next;
        }
        if (cur == header) {
            System.out.println("element " + key + " not found.");
            return;
        }
```

```
        cur.prev.next = cur.next;
        cur.next.prev = cur.prev;
        cur.next = null;
        cur.prev = null;
        System.out.println("deleted: " + key);
    }
    public void displayForward() {
        if (header.next == header) {
            System.out.println("List is empty");
            return;
        }
        System.out.println("Forward:");
        Node cur = header.next;
        while (cur != header) {
            System.out.print(cur.data + "<->");
            cur = cur.next;
        }
        System.out.println("(back to header)");
    }
    public void displayBackward() {
        if (header.prev == header) {
            System.out.println("list is empty");
            return;
        }
        System.out.println("Backward:");
        Node cur = header.prev;
        while (cur != header) {
            System.out.print(cur.data + "<->");
            cur = cur.prev;
        }
        System.out.println("(back to header)");
    }
```

```
    public boolean isEmpty() {
        return header.next == header;
    }
}
public class C2Wayhll{
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        CircularDoublyHeaderList list = new CircularDoublyHeaderList();
        list.insertBegin(30);
        list.insertBegin(20);
        list.insertBegin(10);
        list.insertEnd(40);
        list.insertEnd(50);
        list.insertEnd(60);
        list.displayForward();
        list.displayBackward();
        list.delete(20);
        list.delete(100);
        list.displayForward();
        list.displayBackward();
    }
}
```

Implementation(Output Of The Code):



Practical 5D

Aim :-To study and understand the working of a **Doubly Ended Linked List**.

Theory :- A **Doubly Ended Linked List** is a linked list that maintains **two pointers**:
• One pointer points to the **first node (front)**
• Another pointer points to the **last node (rear)**

Each node contains data and a link to the next node.Because both ends of the list are directly accessible, insertion and deletion operations can be efficiently performed at **both the beginning and the end** of the list.

Algorithm for Traversal of Doubly Ended Linked List :-

Step 1: Start from the first node using the front pointer
Step 2: Display the data of the current node
Step 3: Move to the next node
Step 4: Repeat Steps 2 and 3 until the last node is reached
Step 5: Stop traversal
Step 6: Exit

Time And Space Complexity :-

Time Complexity :-

- Insertion at beginning: **O(1)**
- Insertion at end: **O(1)**
- Traversal: **O(n)**

Space Complexity :-

- **O(n)** — memory required for n nodes

Programming for Doubly Ended Linked List:

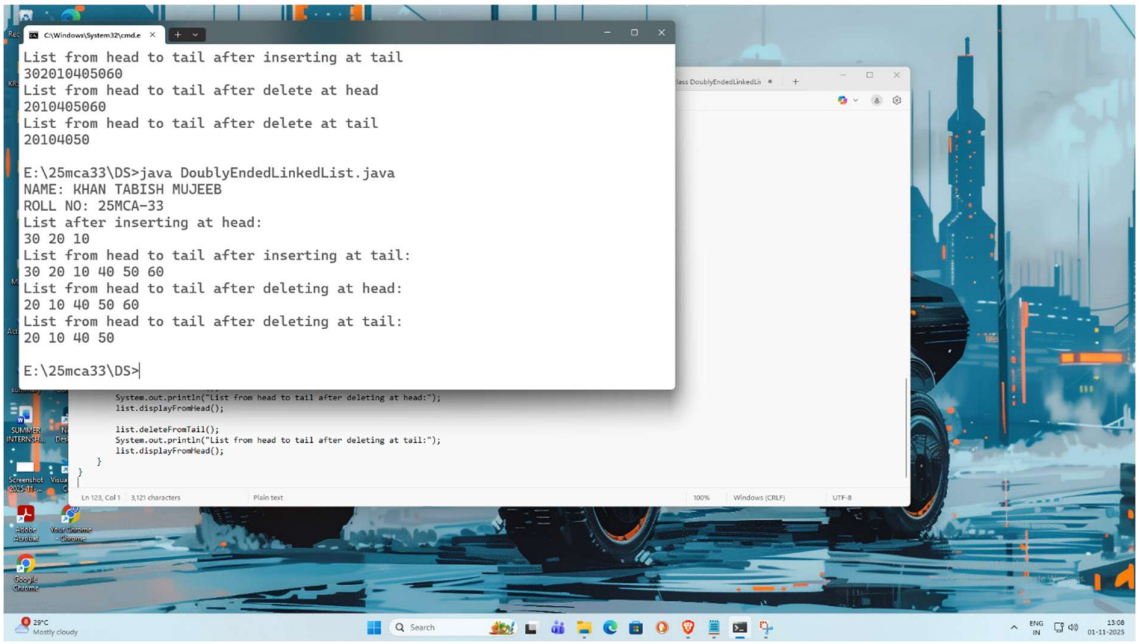
```
class DoublyEndedLinkedList {
   class Node {
      int data;
      Node next;
      Node prev;
      public Node(int data) {
         this.data = data;
         this.next = null;
         this.prev = null;
      }
   }
   private Node head, tail;
   public DoublyEndedLinkedList() {
```

```
    this.head = null;
    this.tail = null;
  }
  public void insertAtHead(int data) {
    Node newNode = new Node(data);
    if (head == null) {
      head = tail = newNode;
    } else {
      newNode.next = head;
      head.prev = newNode;
      head = newNode;
    }
  }
  public void insertAtTail(int data) {
    Node newNode = new Node(data);
    if (tail == null) {
      head = tail = newNode;
    } else {
      tail.next = newNode;
      newNode.prev = tail;
      tail = newNode;
    }
  }
  public void deleteFromHead() {
    if (head == null) {
      System.out.println("List is empty. Cannot delete.");
      return;
    }
    if (head == tail) {
      head = tail = null;
    } else {
      head = head.next;
```

```
            head.prev = null;
        }
    }
    public void deleteFromTail() {
        if (tail == null) {
            System.out.println("List is empty. Cannot delete.");
            return;
        }
        if (head == tail) {
            head = tail = null;
        } else {
            tail = tail.prev;
            tail.next = null;
        }
    }
    public void displayFromHead() {
        if (head == null) {
            System.out.println("List is empty.");
            return;
        }
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
    public void displayFromTail() {
        if (tail == null) {
            System.out.println("List is empty.");
            return;
        }
```

```
        Node current = tail;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.prev;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.print("NAME: KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO: 25MCA-33\n");
        DoublyEndedLinkedList list = new DoublyEndedLinkedList();
        list.insertAtHead(10);
        list.insertAtHead(20);
        list.insertAtHead(30);
        System.out.println("List after inserting at head:");
        list.displayFromHead();
        list.insertAtTail(40);
        list.insertAtTail(50);
        list.insertAtTail(60);
        System.out.println("List from head to tail after inserting at tail:");
        list.displayFromHead();
        list.deleteFromHead();
        System.out.println("List from head to tail after deleting at head:");
        list.displayFromHead();
        list.deleteFromTail();
        System.out.println("List from head to tail after deleting at tail:");
        list.displayFromHead();
    }
}
```

Implementation(Output Of The Code):



Limitations of Doubly Ended Linked List :-

1. Extra memory is required to store the rear pointer
2. More complex than a singly linked list
3. Random access of elements is not possible

Conclusion :- Doubly Ended Linked List allows efficient insertion and deletion at both ends of the list. It improves performance for queue-like operations but is still slower than arrays for accessing elements randomly.

Practical 6

Demonstrate application of Linked List:

1. Polynomial addition, (Using DLL)

2. Sparse matrix

3. Stack

4. Queue

5. Priority and Double ended Queue

Practical 6.1

Aim :- To study and understand **Polynomial Addition using Doubly Linked List (DLL)**.

Theory :- A polynomial consists of terms with coefficients and exponents. In **Polynomial Addition using Doubly Linked List**, each node represents a term of the polynomial containing:

- Coefficient
- Exponent
- Pointer to previous node
- Pointer to next node

Two polynomials are traversed simultaneously. If exponents are equal, coefficients are added. If exponents are unequal, the term with the higher exponent is copied to the result polynomial.

Algorithm used for Polynomial Addition:

Step1: Initialize pointers p1 → Poly1, p2 → Poly2, and create an empty list Result.

Step2: While p1 ≠ NULL and p2 ≠ NULL:

- Case 1: If p1.exp == p2.exp:
- Compute sumCoeff = p1.coeff + p2.coeff.
- If sumCoeff ≠ 0, create a new node with (sumCoeff, p1.exp) and append to Result.
- Move p1 → p1.next, p2 → p2.next.
- Case 2: If p1.exp > p2.exp:
- Copy the term (p1.coeff, p1.exp) to Result.
- Move p1 → p1.next.
- Case 3: If p1.exp < p2.exp:
- Copy the term (p2.coeff, p2.exp) to Result.
- Move p2 → p2.next.

Step3: Append remaining terms of Poly1 (if p1 ≠ NULL) to Result.

Step4: Append remaining terms of Poly2 (if p2 ≠ NULL) to Result.

Step5: Return Result.

Programming for Polynomial Addition:

```
public class Polynomial {

   private Node head;
```

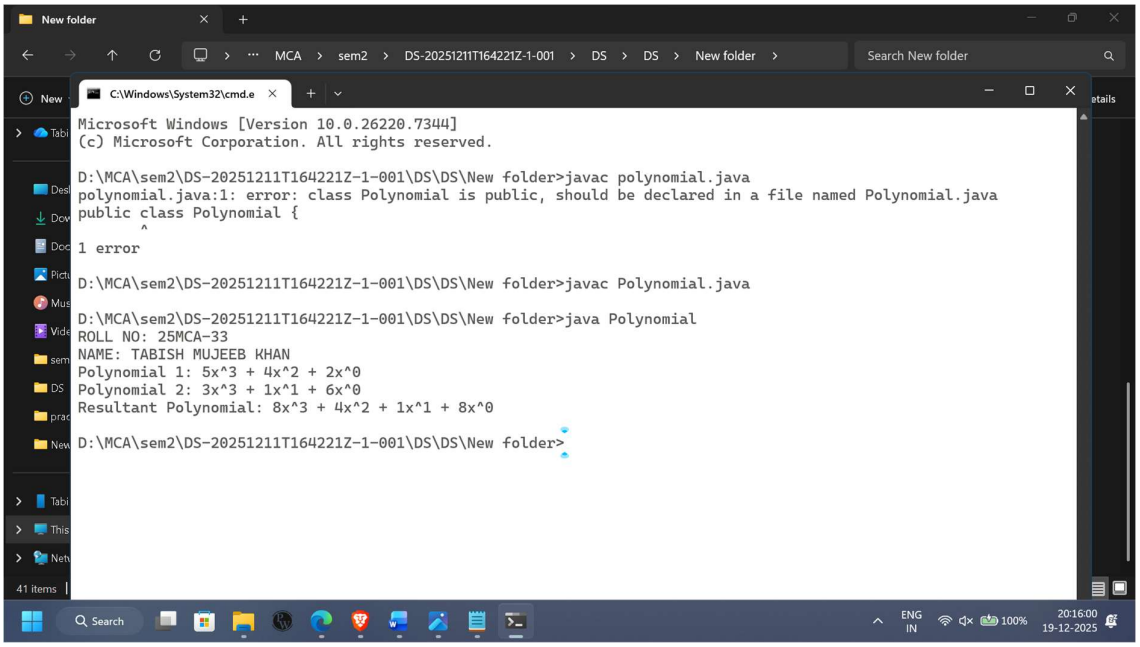
```
private static class Node {
    int coeff;
    int exp;
    Node prev;
    Node next;
    Node(int coeff, int exp) {
        this.coeff = coeff;
        this.exp = exp;
    }
}
public void addTerm(int coeff, int exp) {
    if (coeff == 0) return;
    Node newNode = new Node(coeff, exp);
    if (head == null) {
        head = newNode;
        return;
    }
    Node current = head;
    Node prev = null;
    while (current != null && current.exp > exp) {
        prev = current;
        current = current.next;
    }
    if (current != null && current.exp == exp) {
        current.coeff += coeff;
        if (current.coeff == 0) {
            deleteNode(current);
        }
    } else {
        newNode.next = current;
        newNode.prev = prev;
        if (current != null)
```

```
            current.prev = newNode;
        if (prev != null)
            prev.next = newNode;
        else
            head = newNode;
    }
}
private void deleteNode(Node node) {
    if (node.prev != null)
        node.prev.next = node.next;
    else
        head = node.next;

    if (node.next != null)
        node.next.prev = node.prev;
}
public static Polynomial addPolynomials(Polynomial p1, Polynomial p2) {
    Polynomial result = new Polynomial();
    Node n1 = p1.head;
    Node n2 = p2.head;
    while (n1 != null && n2 != null) {
        if (n1.exp > n2.exp) {
            result.addTerm(n1.coeff, n1.exp);
            n1 = n1.next;
        } else if (n1.exp < n2.exp) {
            result.addTerm(n2.coeff, n2.exp);
            n2 = n2.next;
        } else {
            result.addTerm(n1.coeff + n2.coeff, n1.exp);
            n1 = n1.next;
            n2 = n2.next;
        }
```

```
        }
        while (n1 != null) {
            result.addTerm(n1.coeff, n1.exp);
            n1 = n1.next;
        }
        while (n2 != null) {
            result.addTerm(n2.coeff, n2.exp);
            n2 = n2.next;
        }
        return result;
    }
    public void display() {
        if (head == null) {
            System.out.println("0");
            return;
        }
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.coeff + "x^" + temp.exp);
            if (temp.next != null)
                System.out.print(" + ");
            temp = temp.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN");
        Polynomial poly1 = new Polynomial();
        poly1.addTerm(5, 3);
        poly1.addTerm(4, 2);
        poly1.addTerm(2, 0);
```

```
        Polynomial poly2 = new Polynomial();
        poly2.addTerm(3, 3);
        poly2.addTerm(1, 1);
        poly2.addTerm(6, 0);
        System.out.print("Polynomial 1: ");
        poly1.display();
        System.out.print("Polynomial 2: ");
        poly2.display();
        Polynomial result = Polynomial.addPolynomials(poly1, poly2);
        System.out.print("Resultant Polynomial: ");
        result.display();
    }
}
```

Implementation(Output Of The Code):



Limitations :-

1. Requires extra memory for pointers
2. More complex than array representation
3. Slower for small polynomials

Conclusion :- Polynomial addition using DLL is efficient for handling large polynomials and dynamic data, though it requires more memory and careful pointer management.

Practical 6.2

Aim :- To study and understand the representation of a **Sparse Matrix**.

Theory :- A **Sparse Matrix** is a matrix in which most of the elements are zero. To save memory, only **non-zero elements** are stored along with their row and column positions. This reduces memory usage and improves performance.

Algorithm used for Sparse Matrix:

Step 1:Input the matrix dimensions (rows, columns).

Step 2: Initialize head = NULL.

Step 3: FOR i = 0 to rows-1:

FOR j = 0 to columns-1:

IF matrix[i][j] != 0:

Create a new node with:

row = i, column = j, value = matrix[i][j]

next = NULL

IF head == NULL:

head = newNode

ELSE:

Append newNode at the end of the list

Step 4: Initialize temp = head.

Step 5: WHILE temp != NULL:

Print (temp.row, temp.column, temp.value)

Move temp = temp.next

Step 6**:** Exit

Time And Space Complexity :-

- **Time Complexity:** O(rows × columns)
- **Space Complexity:** O(non-zero elements)

Programming for Sparse Matrix:

```
class SparseMatrix
{
   private Node head;
   private static class Node
   {
      int row;
```

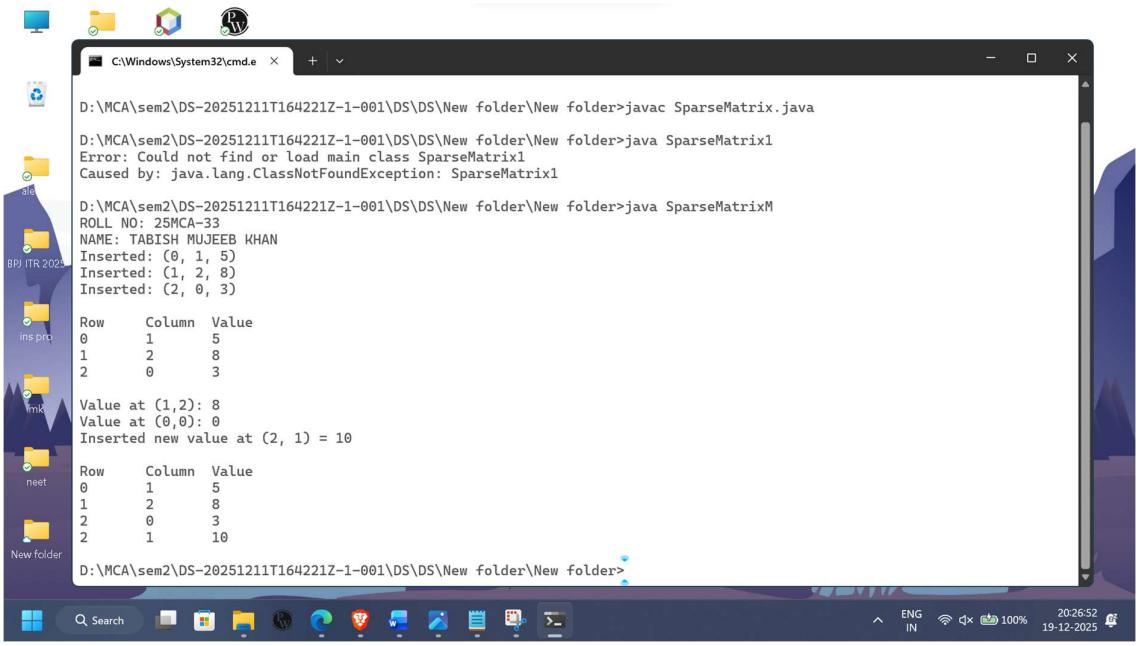
```
    int col;
    int value;
    Node next;
    Node(int row, int col, int value)
    {
        this.row = row;
        this.col = col;
        this.value = value;
        this.next = null;
    }
}
public SparseMatrix()
{
    head = null;
}
public void insert(int row, int col, int value)
{
    if (value == 0) return;
    Node newNode = new Node(row, col, value);
    if (head == null)
    {
        head = newNode;
    }
    else
    {
        Node current = head;
        while (current.next != null)
        {
            current = current.next;
        }
        current.next = newNode;
    }
```

```
        System.out.println("Inserted: (" + row + ", " + col + ", " + value + ")");
    }
    public void display()
    {
        if (head == null)
        {
            System.out.println("Sparse matrix is empty");
            return;
        }
        Node current = head;
        System.out.println("\nRow\tColumn\tValue");
        while (current != null)
        {
            System.out.println(current.row + "\t" + current.col + "\t" + current.value);
            current = current.next;
        }
    }
    public int getValue(int row, int col)
    {
        Node current = head;
        while (current != null)
        {
            if (current.row == row && current.col == col)
            {
                return current.value;
            }
            current = current.next;
        }
        return 0;
    }
    public void setValue(int row, int col, int value)
    {
```

```
        if (value == 0)
        {
            System.out.println("Setting a zero value is ignored in sparse matrix.");
            return;
        }
        Node current = head;
        Node prev = null;
        while (current != null)
        {
            if (current.row == row && current.col == col)
            {
                current.value = value;
                System.out.println("Updated value at (" + row + ", " + col + ") to " + value);
                return;
            }
            prev = current;
            current = current.next;
        }
        Node newNode = new Node(row, col, value);
        if (prev == null)
        {
            head = newNode;
        }
        else
        {
            prev.next = newNode;
        }
        System.out.println("Inserted new value at (" + row + ", " + col + ") = " + value);
    }
}
class SparseMatrixM
{
```

```
    public static void main(String[] args)
    {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN");
        SparseMatrix sparseMatrix = new SparseMatrix();
        sparseMatrix.insert(0, 1, 5);
        sparseMatrix.insert(1, 2, 8);
        sparseMatrix.insert(2, 0, 3);
        sparseMatrix.display();
        System.out.println("\nValue at (1,2): " + sparseMatrix.getValue(1, 2));
        System.out.println("Value at (0,0): " + sparseMatrix.getValue(0, 0));
        sparseMatrix.setValue(2, 1, 10);
        sparseMatrix.display();
    }
}
```

Implementation(Output Of The Code):



Practical 6.3

Aim :-To study and understand the working of a **Stack data structure**.

Theory :-A **Stack** is a linear data structure that follows **LIFO (Last In First Out)** principle. Insertion is called **Push** and deletion is called **Pop**. All operations are performed at one end called the **top**.

Algorithm used for Stack:

Step 1: Input the matrix dimensions (rows, columns).

Step 2: Initialize head = NULL.

Step 3: FOR i = 0 to rows-1:

FOR j = 0 to columns-1:

IF matrix[i][j] != 0:

Create a new node with:

row = i, column = j, value = matrix[i][j]

next = NULL

IF head == NULL:

head = newNode

ELSE:

Append newNode at the end of the list

Step 4: Initialize temp = head.

Step 5: WHILE temp != NULL:

Print (temp.row, temp.column, temp.value)

Move temp = temp.next

Step 6: exit

Time And Space Complexity :-

- **Time Complexity:** O(1)
- **Space Complexity:** O(n)

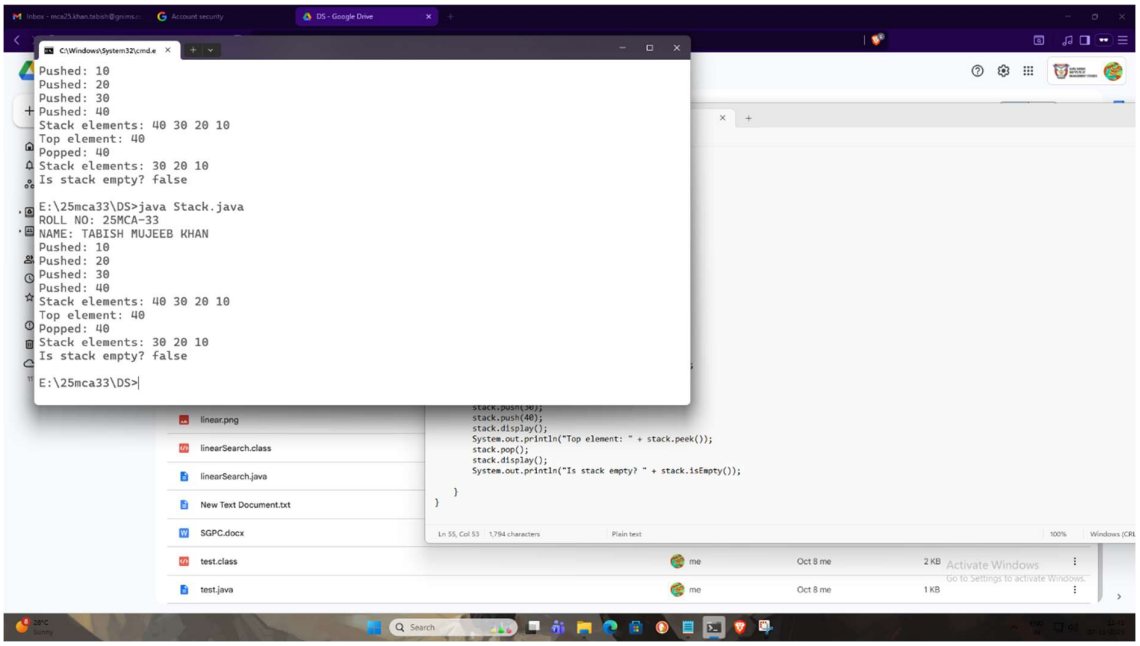
Programming for Stack:

```
class Stack {
   private Node top;
   private static class Node {
      int data;
      Node next;
      Node(int data) {
         this.data = data;
```

```
            this.next = null;
        }
    }
    public Stack() {
        this.top = null;
    }
    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
        System.out.println("Pushed: " + data);
    }
    public int pop() {
        if (top == null) {
            System.out.println("Stack is empty");
            return -1;
        }
        int data = top.data;
        top = top.next;
        System.out.println("Popped: " + data);
        return data;
    }
    public int peek() {
        if (top == null) {
            System.out.println("Stack is empty");
            return -1;
        }
        return top.data;
    }
    public boolean isEmpty() {
        return top == null;
    }
```

```
    public void display() {
        if (top == null) {
            System.out.println("Stack is empty");
            return;
        }
        Node current = top;
        System.out.print("Stack elements: ");
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN");
        Stack stack = new Stack();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
        stack.display();
        System.out.println("Top element: " + stack.peek());
        stack.pop();
        stack.display();
        System.out.println("Is stack empty? " + stack.isEmpty());
    }
}
```

Implementation(Output Of The Code):



Practical 6.4

Aim :- To study and understand the working of a **Queue data structure**.

Theory :- A **Queue** is a linear data structure that follows **FIFO (First In First Out)** principle. Insertion is done at the **rear**, and deletion is done from the **front**.

Algorithm used for Queue :

Step 1: If Front = 1 And Rear = n Then

Print "Queue is Full, Overflow Condition"

Exit

[End If]

Step 2: If Front = Rear + 1 Then

Print: "Queue is full, Overflow condition"

Exit

[End If]

Step 3: If Rear = Null Then

Set Front = 1 And Rear = 1

Else If Rear = n Then

Set Rear = 1

Else

Set Rear = Rear + 1

[End If]

Step 4: Set Q[Rear] = Data

Step 5: Exit

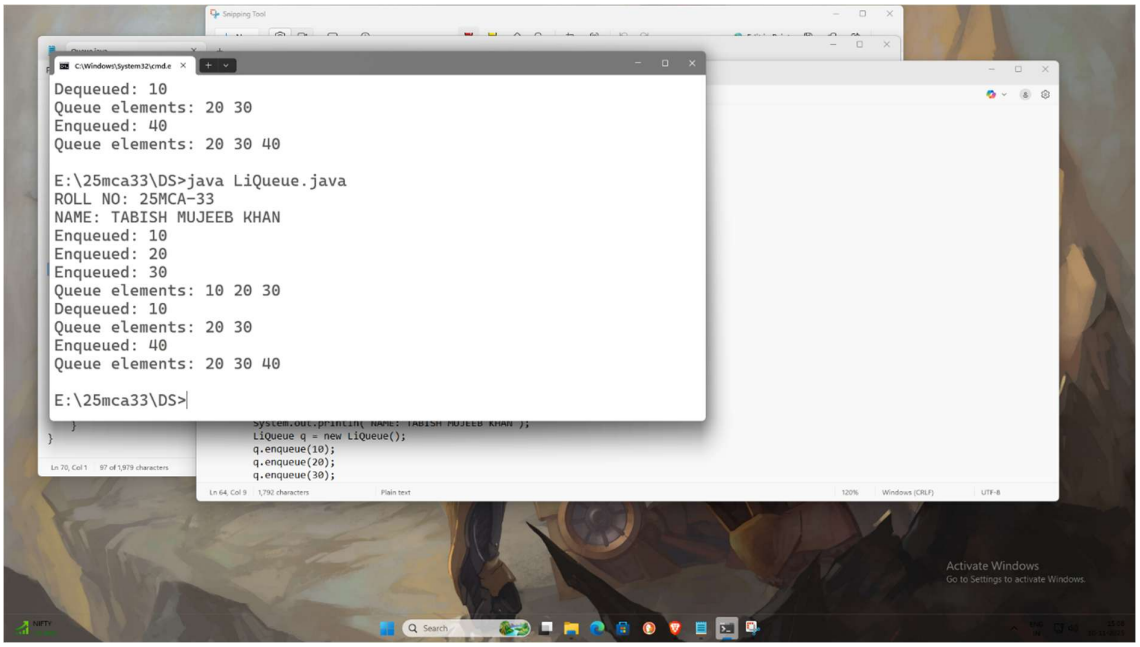
Programming for Queue :

```
class LiQueue {
   private Node front;
   private Node rear;
   private static class Node {
      int data;
      Node next;
      Node(int data) {
         this.data = data;
         this.next = null;
```

```
    }
}
public LiQueue() {
    this.front = this.rear = null;
}
public void enqueue(int data) {
    Node newNode = new Node(data);
    if (rear == null) {
        front = rear = newNode;
        System.out.println("Enqueued: " + data);
        return;
    }
    rear.next = newNode;
    rear = newNode;
    System.out.println("Enqueued: " + data);
}
public int dequeue() {
    if (front == null) {
        System.out.println("Queue is empty");
        return -1;
    }
    int data = front.data;
    front = front.next;
    if (front == null) {
        rear = null;
    }
    System.out.println("Dequeued: " + data);
    return data;
}
public boolean isEmpty() {
    return front == null;
}
```

```
    public void display() {
        if (front == null) {
            System.out.println("Queue is empty");
            return;
        }
        Node current = front;
        System.out.print("Queue elements: ");
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        System.out.println("ROLL NO: 25MCA-33");
        System.out.println("NAME: TABISH MUJEEB KHAN");
        LiQueue q = new LiQueue();
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.display();
        q.dequeue();
        q.display();
        q.enqueue(40);
        q.display();
    }
}
```

Implementation(Output Of The Code):



Practical 6.5

Aim :- To study **Priority Scheduling using Doubly Ended Queue (Deque)**.

Theory :- A **Priority Queue** processes elements based on priority rather than insertion order. Using a **Doubly Ended Queue**, insertion and deletion can be performed at both ends. High-priority tasks are inserted at the front, and low-priority tasks at the rear.

Algorithm :-

Step 1: Assign priority to each task
Step 2: Insert high-priority task at front
Step 3: Insert low-priority task at rear
Step 4: Remove task from front for execution
Step 5: Repeat
Step 6: Exit

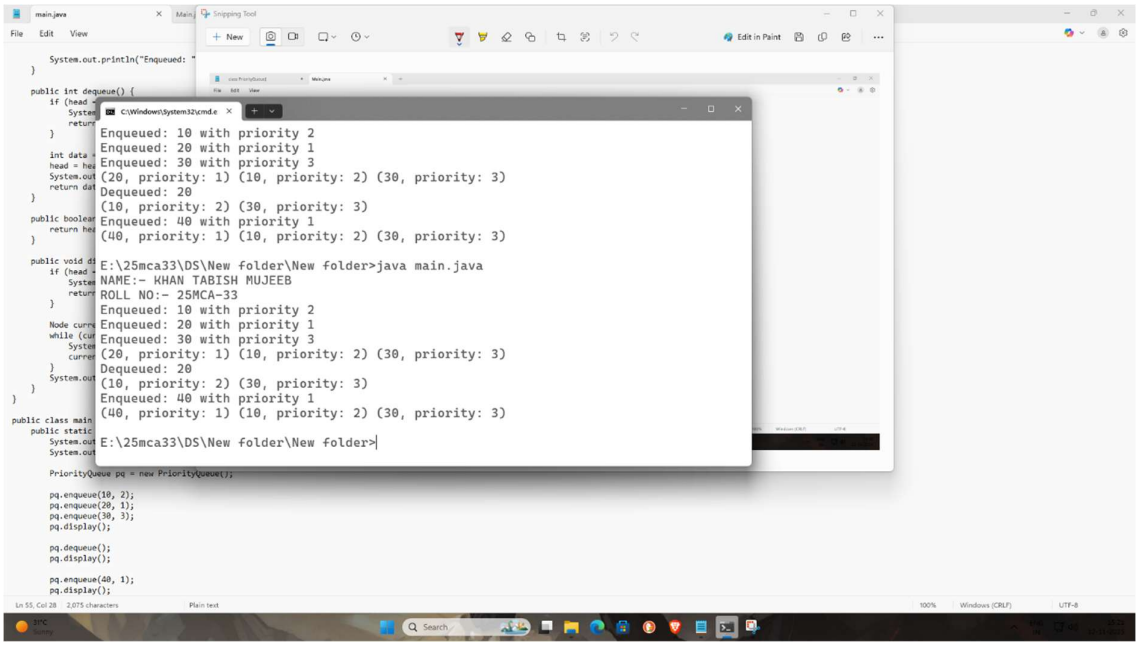
Programming for Priority and Double ended Queue:

```
class PriorityQueue {
    private Node head;
    private static class Node {
        int data;
        int priority;
        Node next;
        Node(int data, int priority) {
            this.data = data;
            this.priority = priority;
            this.next = null;
        }
    }
    public PriorityQueue() {
        this.head = null;
    }
    public void enqueue(int data, int priority) {
        Node newNode = new Node(data, priority);
        if (head == null || head.priority > priority) {
            newNode.next = head;
            head = newNode;
        } else {
            Node current = head;
```

```
            while (current.next != null && current.next.priority <= priority) {
                current = current.next;
            }
            newNode.next = current.next;
            current.next = newNode;
        }
        System.out.println("Enqueued: " + data + " with priority " + priority);
    }
    public int dequeue() {
        if (head == null) {
            System.out.println("Queue is empty");
            return -1;
        }
        int data = head.data;
        head = head.next;
        System.out.println("Dequeued: " + data);
        return data;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void display() {
        if (head == null) {
            System.out.println("Queue is empty");
            return;
        }
        Node current = head;
        while (current != null) {
            System.out.print("(" + current.data + ", priority: " + current.priority + ") ");
            current = current.next;
        }
        System.out.println();
```

```
    }
}
public class main {
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        PriorityQueue pq = new PriorityQueue();
        pq.enqueue(10, 2);
        pq.enqueue(20, 1);
        pq.enqueue(30, 3);
        pq.display();
        pq.dequeue();
        pq.display();
        pq.enqueue(40, 1);
        pq.display();
    }
}
```

Implementation(Output Of The Code):



PRACTICAL NO 7

Aim: Create and perform various operations on BST. (Binary Search Tree)

- Insertion
- Traversal:
    a. Pre-order,
    b. In-order,
    c. Post-Order
- Deletion

Practical 7.a

Theory: In Pre-order traversal, nodes are visited in the following order:

Root → Left Subtree → Right Subtree

Algorithm: To Traverse a binary tree in Pre-order manner recursively:

Step 1: If Root = Null Then

Print: "Tree is empty" Return

Else

Print Root → Info

[End If]

Step 2: If Root → Left ≠ Null Then

Call RecPreTraversal(Root → Left) [End If]

Step 3: If Root → Right ≠ Null Then

Call RecPreTraversal (Root → Right) [End If]

Step 4: Return

Time and Space Complexity :-

- **Time Complexity:** O(n)
- **Space Complexity:** O(h) (h = height of the tree, due to recursion stack)

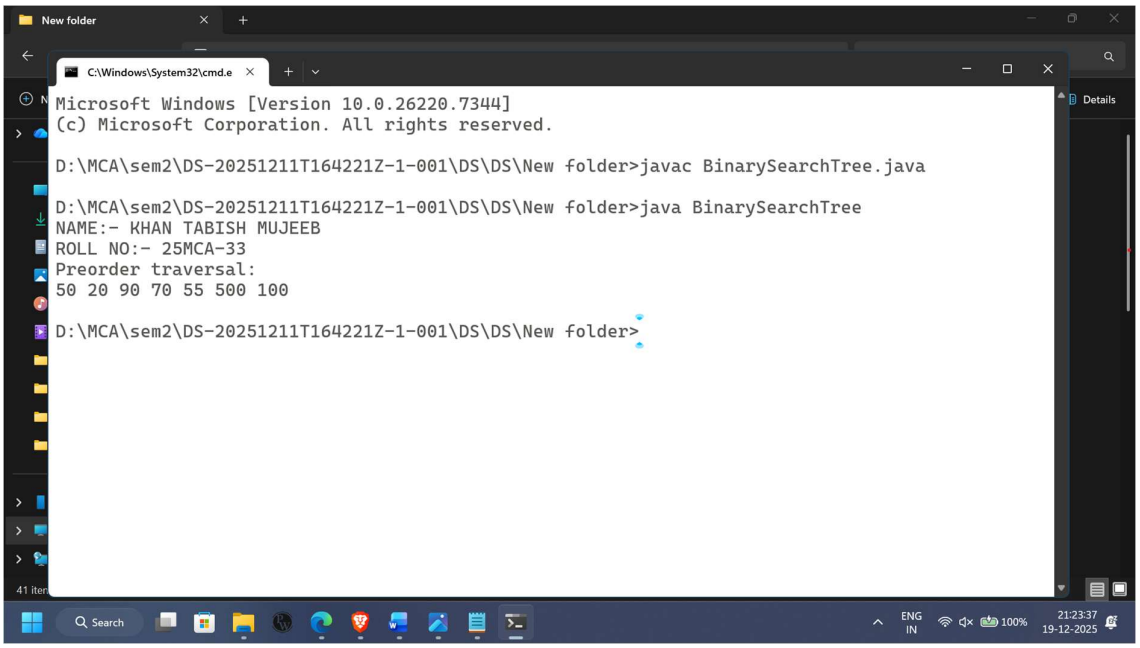
Programming for Pre-Order traversal:

```
class BinarySearchTree {
   static class Node {
      int data;
      Node left, right;
      public Node(int item) {
         data = item;
         left = right = null;
      }
```

```
}
static class BST {
    Node root;
    public BST() {
        root = null;
    }
    public void insert(int data) {
        root = insertRec(root, data);
    }
    private Node insertRec(Node root, int data) {
        if (root == null) {
            return new Node(data);
        }
        if (data < root.data) {
            root.left = insertRec(root.left, data);
        } else if (data > root.data) {
            root.right = insertRec(root.right, data);
        }
        return root;
    }
    public void preorder() {
        System.out.println("Preorder traversal:");
        preorderRec(root);
        System.out.println();
    }
    private void preorderRec(Node root) {
        if (root != null) {
            System.out.print(root.data + " ");  // FIXED
            preorderRec(root.left);
            preorderRec(root.right);
        }
    }
```

```
    }
    public static void main(String[] args) {
 System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        BST tree = new BST();
        tree.insert(50);
        tree.insert(20);
        tree.insert(90);
        tree.insert(500);
        tree.insert(100);
        tree.insert(70);
        tree.insert(55);
        tree.preorder();
    }
}
```

Implementation(Output Of The Code):



Practical 7.b

Aim: To implement a Binary Search Tree (BST) and perform Insertion, Deletion, and In-Order Traversal operations on it.

1.Insertion

2.Deletion

B.In-order traversal

Theory:A Binary Search Tree (BST) is a non-linear data structure where:

- Each node contains data, left child, and right child.
- Left subtree contains values smaller than the root.
- Right subtree contains values greater than the root.
- No duplicate values are allowed.

1. Insertion in BST

Insertion always maintains the BST property.

Process:

Compare new data with root.

If data < root → go to left subtree.

If data > root → go to right subtree.

When a null position is found → insert node there.

Time Complexity: O(log n) average

2. Deletion in BST

Deletion of a node depends on three cases:

Case 1: Node has no child (Leaf Node)

Directly remove the node.

Case 2: Node has one child

Replace the node with its child.

Case 3: Node has two children

Find In-order Successor (minimum node in the right subtree)

Replace the node's value with successor value

Delete the successor

This ensures the BST property remains intact.

3. In-Order Traversal of BST

Traversal means visiting all nodes in a specific order.

In-Order Traversal = Left → Root → Right

For BST, in-order traversal always gives elements in ascending sorted order.

Example:

If the tree has: 50, 30, 70, 20, 40

In-Order Output → 20 30 40 50 70

Algorithm used for Insertion in BST:

Step 1:If Free = Null Then

Print: "No space is available for the node to insert" Exit

Else

Allocate memory to new node for insertion (New Free And Free = Free → Right)

Set New Info = Item

Set New Left = Null And New Right = Null [End If]

Step 2:If Root = Null Then Set Root New Exit

[End If]

Step 3:If Item ≥ Root → Info Then Set Pointer Root → Right

Set PointerP = Root Else

Set Pointer = Root → Left Set PointerP = Root

[End If]

Step 4:Repeat step 5 While Pointer + Null

Step 5:If Item ≥ Pointer Info Then Set PointerP = Pointer

Set Pointer Pointer Right Else

Set PointerP = Pointer Set Pointer Pointer Left [End If]

[End Loop]

Step 6:If Item PointerP→ Info Then

Set PointerP → Left = New Else

Set PointerP → Right = New [End If]

Step 7:Exit

Algorithm used for Deletion in BST:

Step 1:Call BS

Step 2:If Position = Null Then Print: "Item not found in the tree"

Exit [End If]

Step 3:If Position ND Position → Right Null Then →Left Null AND Call Delete2(Root, Position, Parent)

Else

Call Deletel(Root, Position, Parent) [End If]

Step 4:Deallocate memory held by node Position (Set Position → Right = Free And Free = Position)

Step 5:Exit

Algorithm used for Inorder Traversal in BST: (Left → Root → Right)

IF node ≠ NULL:

Call InorderTraversal(node.left).

Process node.data (Visit the node).

Call InorderTraversal(node.right).

Programming for Insertion ,In-order, Deletion:

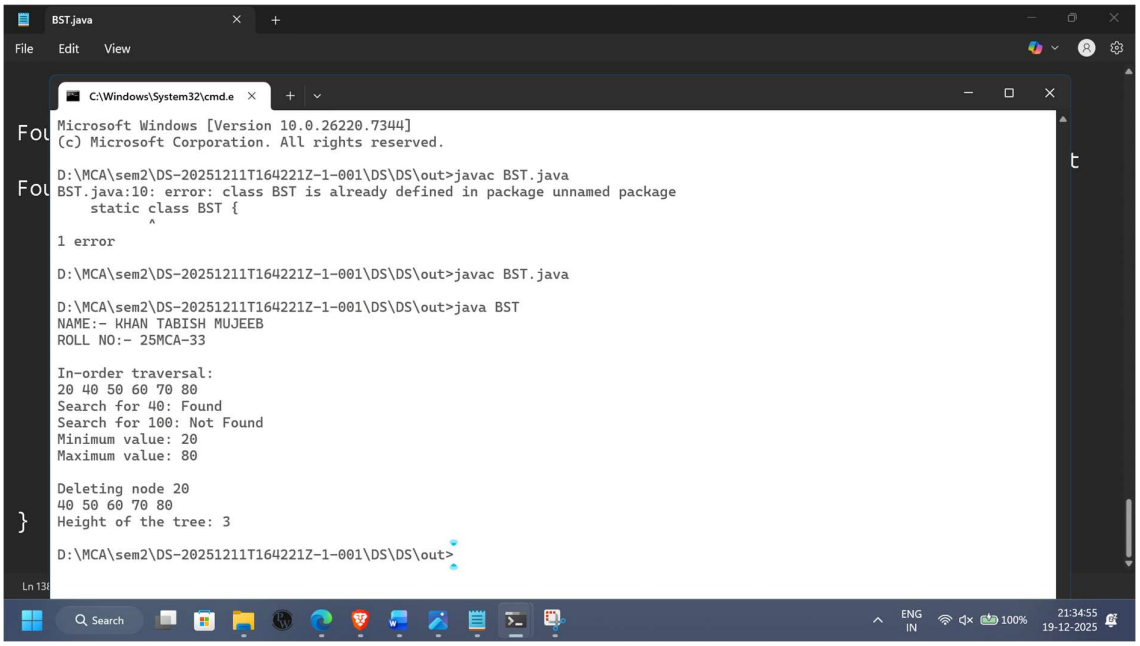
```
public class BST {
   static class Node {
      int data;
      Node left, right;
      public Node(int item) {
         data = item;
         left = right = null;
      }
   }
   Node root;
   public BST() {
      root = null;
   }
   public void insert(int data) {
      root = insertRec(root, data);
   }
   private Node insertRec(Node root, int data) {
      if (root == null) {
         root = new Node(data);
         return root;
      }
      if (data < root.data) {
```

```
            root.left = insertRec(root.left, data);
        } else if (data > root.data) {
            root.right = insertRec(root.right, data);
        }
        return root;
    }
    public void inorder() {
        inorderRec(root);
    }
    private void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.data + " ");
            inorderRec(root.right);
        }
    }
    public boolean search(int key) {
        return searchRec(root, key);
    }
    private boolean searchRec(Node root, int key) {
        if (root == null) return false;
        if (root.data == key) return true;
        if (key > root.data) return searchRec(root.right, key);
        return searchRec(root.left, key);
    }
    public int findMin() {
        Node current = root;
        while (current.left != null) current = current.left;
        return current.data;
    }
    public int findMax() {
        Node current = root;
```

```
    while (current.right != null) current = current.right;
    return current.data;
}
public void delete(int key) {
    root = deleteRec(root, key);
}
private Node deleteRec(Node root, int key) {
    if (root == null) return null;
    if (key < root.data) root.left = deleteRec(root.left, key);
    else if (key > root.data) root.right = deleteRec(root.right, key);
    else {
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        root.data = minValue(root.right);
        root.right = deleteRec(root.right, root.data);
    }
    return root;
}
private int minValue(Node root) {
    int minValue = root.data;
    while (root.left != null) {
        minValue = root.left.data;
        root = root.left;
    }
    return minValue;
}
public int height() {
    return heightRec(root);
}
private int heightRec(Node root) {
    if (root == null) return 0;
    int leftHeight = heightRec(root.left);
```

```
        int rightHeight = heightRec(root.right);
        return Math.max(leftHeight, rightHeight) + 1;
    }
    public static void main(String[] args) {
        System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        BST tree = new BST();
        tree.insert(50);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);
        System.out.println("\nIn-order traversal:");
        tree.inorder();
        System.out.println();
        System.out.println("Search for 40: " + (tree.search(40) ? "Found" : "Not Found"));
        System.out.println("Search for 100: " + (tree.search(100) ? "Found" : "Not Found"));
        System.out.println("Minimum value: " + tree.findMin());
        System.out.println("Maximum value: " + tree.findMax());
        System.out.println("\nDeleting node 20");
        tree.delete(20);
        tree.inorder();
        System.out.println();
        System.out.println("Height of the tree: " + tree.height());
    }
}
```

Implementation(Output Of The Code):



Practical 7.c

C.Post-order traversal

Aim: To implement Post-Order Traversal of a Binary Search Tree (BST) and observe how nodes are visited in Left–Right–Root order.

Theory:

Post-order traversal is a depth-first traversal technique used in Binary Trees and Binary Search Trees.

In this traversal, each node is visited in the following order:

Post-Order = Left Subtree → Right Subtree → Root Node

Meaning:

First visit left child

Then visit right child

Finally visit current node (root)

Algorithm used for Postorder Traversal in BST: (Left → Right → Root)

IF node ≠ NULL:

Call PostorderTraversal(node.left).

Call PostorderTraversal(node.right).

Process node.data (Visit the node).

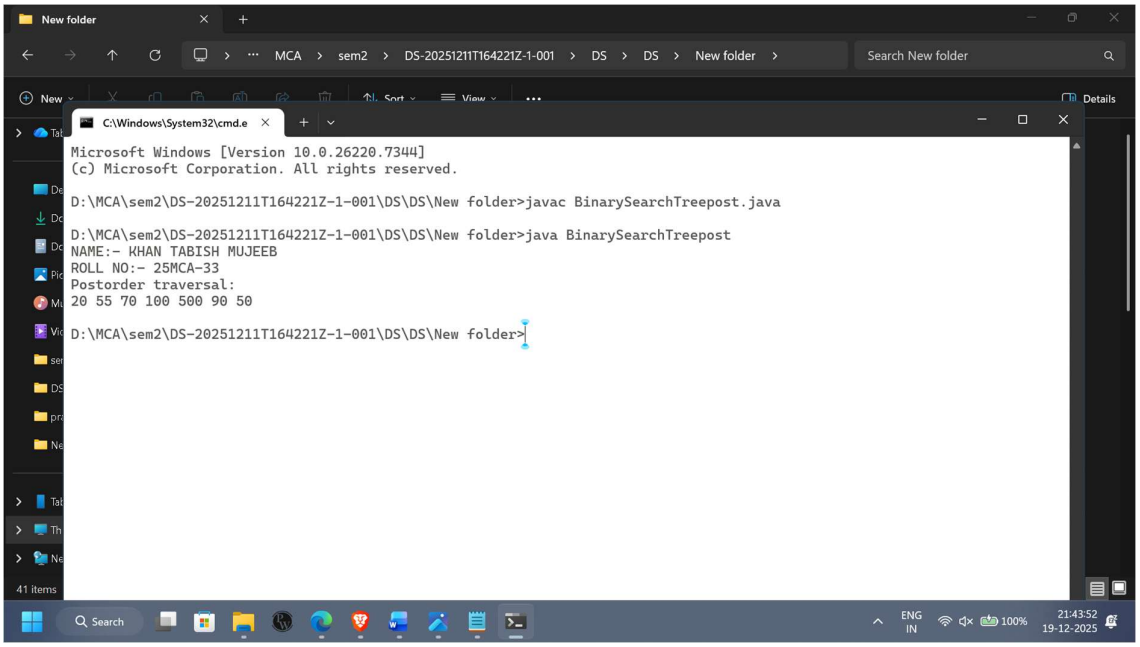
Programming for Post-Order Traversal:

```
class BinarySearchTreepost {
   static class Node {
      int data;
      Node left, right;
      public Node(int item) {
         data = item;
         left = right = null;
      }
   }
   static class BST {
      Node root;
      public BST() {
         root = null;
      }
      public void insert(int data) {
```

```
        root = insertRec(root, data);
    }
    private Node insertRec(Node root, int data) {
        if (root == null) {
            return new Node(data);
        }
        if (data < root.data) {
            root.left = insertRec(root.left, data);
        } else if (data > root.data) {
            root.right = insertRec(root.right, data);
        }
        return root;
    }
    public void postorder() {
        System.out.println("Postorder traversal:");
        postorderRec(root);
        System.out.println();
    }
    private void postorderRec(Node root) {
        if (root != null) {
            postorderRec(root.left);
            postorderRec(root.right);
            System.out.print(root.data + " ");
        }
    }
}
  public static void main(String[] args) {
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
    System.out.print("ROLL NO:- 25MCA-33\n");
    BST tree = new BST();
    tree.insert(50);
    tree.insert(20);
```

```
        tree.insert(90);
        tree.insert(500);
        tree.insert(100);
        tree.insert(70);
        tree.insert(55);
        tree.postorder();
    }
}
```

Implementation(Output Of The Code):



Practical No.8

Aim: Implementing Heap with different operations performed. (Heap)

- ReheapUp (Insertion)
- ReheapDown (Deletion)
- Delete

Theory: A **Heap** is a **complete binary tree** that satisfies the **heap property**. It is primarily used to implement **priority queues** and is a key component of **heap sort** algorithms.

There are two types of heaps:

1. **Max Heap**: In this type, the value of each parent node is **greater than or equal** to the values of its children. The largest element is always at the **root**.
2. **Min Heap**: In this type, the value of each parent node is **less than or equal** to the values of its children. The smallest element is always at the **root**.

Both types of heaps are **complete binary trees**, meaning:

1. All levels of the tree are fully filled except for possibly the last level.
2. The last level is filled from left to right.

Algorithm used for Heap:

Step1:Set Pos = 1

Step2: Set Item H[Pos]

Step 3:Set Temp H[n] And n = n-1

Step 4:Set Left 2x Pos And Right = 2 x Pos + 1 d by CamScanner

Step 5:Repeat steps 6 to 8 While Right ≤ n

Step 6:If Temp≥ H[Left] AND Temp ≥ H[Right] Set H[Pos] Return Temp [End If]

Step 7:If H[Left] ≥ H[Right] Then

Set H[Pos] = H[Left] And Pos = Left Set H[Pos]H[Right] And Pos = Right Else [End If]

Step 8:Set Left 2x Pos And Right = 2 x Pos + 1 [End While]

Step 9:If Left AND Temp < H[Left] Then Set H[Pos]H[Left] And Pos = Left [End If]

Step 10:Set H[Pos] Temp And Return Item

Programming for Heap:

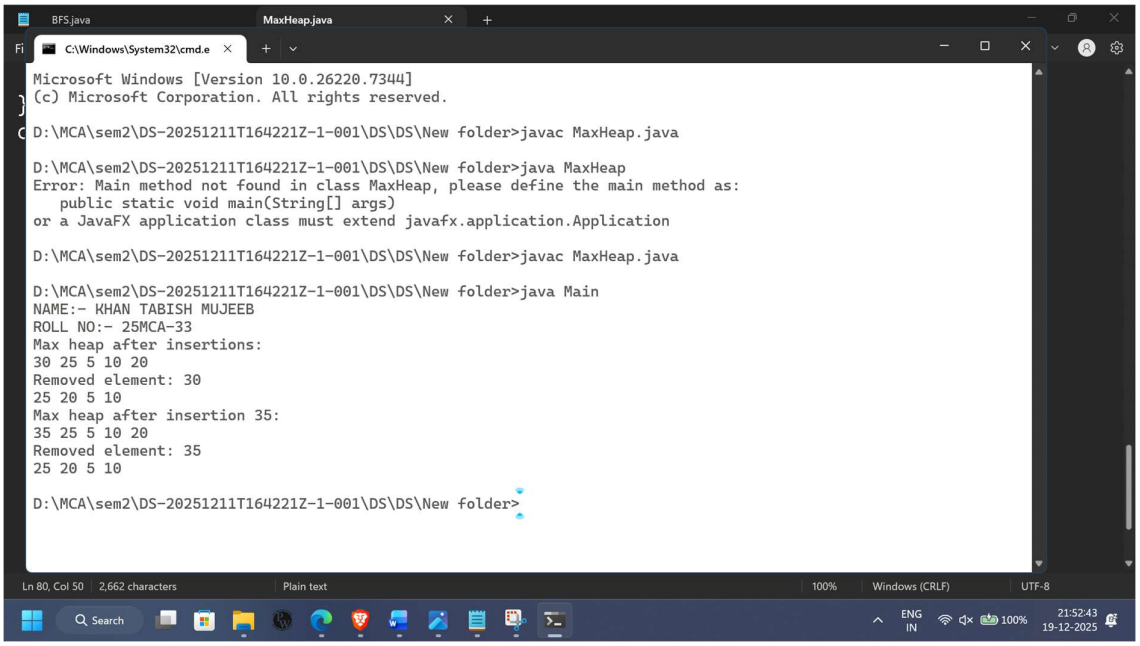
```
import java.util.*;
class MaxHeap {
   private int[] heap;
   private int size;
```

```
private int capacity;
public MaxHeap(int capacity) {
    this.capacity = capacity;
    heap = new int[capacity];
    size = 0;
}
private int parent(int index) {
    return (index - 1) / 2;
}
private int leftChild(int index) {
    return 2 * index + 1;
}
private int rightChild(int index) {
    return 2 * index + 2;
}
public void insert(int value) {
    if (size == capacity) {
        throw new IllegalStateException("heap is full");
    }
    heap[size] = value;
    size++;
    reheapUp(size - 1);
}
private void reheapUp(int index) {
    while (index > 0 && heap[parent(index)] < heap[index]) {
        swap(index, parent(index));
        index = parent(index);
    }
}
public int remove() {
    if (size == 0) {
        throw new IllegalStateException("heap is empty");
```

```
        }
        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
        reheapDown(0);
        return root;
    }
    private void reheapDown(int index) {
        int largest = index;
        int left = leftChild(index);
        int right = rightChild(index);
        if (left < size && heap[left] > heap[largest]) {
            largest = left;
        }
        if (right < size && heap[right] > heap[largest]) {
            largest = right;
        }
        if (largest != index) {
            swap(index, largest);
            reheapDown(largest);
        }
    }
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
    public void printHeap() {
        for (int i = 0; i < size; i++) {
            System.out.print(heap[i] + " ");
        }
        System.out.println();
```

```
    }
    public boolean isEmpty() {
        return size == 0;
    }
    public int getSize() {
        return size;
    }
}
class Main {
    public static void main(String[] args) {
 System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        MaxHeap mh = new MaxHeap(10);
        mh.insert(10);
        mh.insert(20);
        mh.insert(5);
        mh.insert(30);
        mh.insert(25);
        System.out.println("Max heap after insertions:");
        mh.printHeap();
        System.out.println("Removed element: " + mh.remove());
        mh.printHeap();
        mh.insert(35);
        System.out.println("Max heap after insertion 35:");
        mh.printHeap();
        System.out.println("Removed element: " + mh.remove());
        mh.printHeap();
    }
}
```

Implementation(Output Of The Code):



PRACTICAL NO.9

Aim:- Create a Graph storage structure (e.g., Adjacency matrix)

Theory: An Adjacency Matrix is a 2-D array of size V × V, where V is the number of vertices in the graph.

Each cell matrix[i][j] represents whether an edge exists between vertex i and vertex j.

An adjacency matrix is a matrix used to represent a finite graph where:

matrix[i][j] = 1 → if there is an edge from vertex i to vertex j

matrix[i][j] = 0 → if no edge exists between the vertices

For weighted graphs:

matrix[i][j] = weight of edge (i, j)

0 or ∞ represents no edge

Algorithm used for Adjacency matrix:

Step 1: Initialize the graph:

Input number of vertices V and edges E.

Create a 2D matrix adjMatrix[V][V] initialized to 0.

Step 2: Add edges:

FOR each edge (u, v) in the graph:

Set adjMatrix[u][v] = 1 (for unweighted, or set to edge weight for weighted).

If undirected graph, set adjMatrix[v][u] = 1.

Step 3: Display the adjacency matrix: FOR i = 0 to V-1:

FOR j = 0 to V-1:

PRINT adjMatrix[i][j]

Step 4: To check if there is an edge between u and v: IF adjMatrix[u][v] != 0:

PRINT "Edge exists between u and v." ELSE:

PRINT "No edge between u and v."

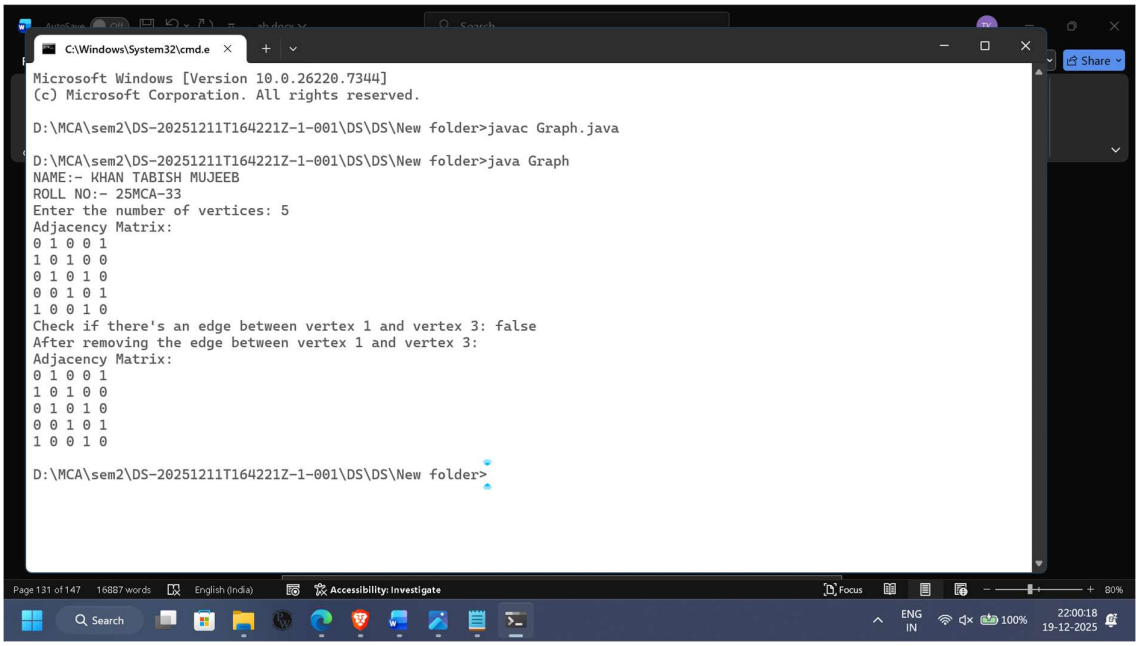
Programming for Adjacency matrix:

```
import java.util.*;
public class Graph {
   private int[][] adjMatrix;
   private int numVertices;
   public Graph(int numVertices) {
      this.numVertices = numVertices;
```

```
    adjMatrix = new int[numVertices][numVertices];
  }
  public void addEdge(int v1, int v2) {
    if (v1 >= numVertices || v2 >= numVertices) {
      System.out.println("Invalid vertex number");
      return;
    }
    adjMatrix[v1][v2] = 1;
    adjMatrix[v2][v1] = 1;
  }
  public void removeEdge(int v1, int v2) {
    if (v1 >= numVertices || v2 >= numVertices) {
      System.out.println("Invalid vertex number");
      return;
    }
    adjMatrix[v1][v2] = 0;
    adjMatrix[v2][v1] = 0;
  }
  public void printGraph() {
    System.out.println("Adjacency Matrix:");
    for (int i = 0; i < numVertices; i++) {
      for (int j = 0; j < numVertices; j++) {
        System.out.print(adjMatrix[i][j] + " ");
      }
      System.out.println();
    }
  }
  public boolean hasEdge(int v1, int v2) {
    if (v1 >= numVertices || v2 >= numVertices) {
      System.out.println("Invalid vertex number");
      return false;
    }
```

```
        return adjMatrix[v1][v2] == 1;
    }
    public static void main(String[] args) {
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int vertices = scanner.nextInt();
        Graph graph = new Graph(vertices);
        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);
        graph.addEdge(4, 0);
        graph.printGraph();
        System.out.print("Check if there's an edge between vertex 1 and vertex 3: ");
        System.out.println(graph.hasEdge(1, 3));
        graph.removeEdge(1, 3);
        System.out.println("After removing the edge between vertex 1 and vertex 3:");
        graph.printGraph();
    }
}
```

Implementation(Output Of The Code):



Conclusion:

Graphs are a versatile and powerful data structure used to represent various real-world systems involving relationships and connections. They are fundamental in **networking**, **social media**, **pathfinding algorithms**, and many other applications.
Understanding graph traversal techniques like **BFS** and **DFS**, as well as graph representations, is essential for efficiently working with complex systems that require relationship modeling.

Practical No 10

Demonstrate application of queue

a. Priority Queue (Task Scheduling using Priority Queue)

b. BFS (Breadth First Search)

c. DFS (Depth First Search)

Practical 10.a

Aim: Demonstrate application of Priority Queue

Theory: A Priority Queue is a special type of queue in which each element has a priority, and elements are removed based on their priority rather than their insertion order.

In a normal queue → FIFO (First In First Out)

In a priority queue → Highest (or lowest) priority element is removed first

Priority queues are commonly implemented using Heaps because they provide efficient operations:

Insertion: O(log n)

Deletion (remove highest priority): O(log n)

Search top element: O(1)

Algorithm used for Priority Queue:

Step1:taskName (String) priority (int)

Step 2: Define a constructor for the class to initialize taskName and priority.

Step 3: Provide getter methods getTaskName() and getPriority() to retrieve task details. Step 4: Override the toString() method to return a string representation of the task in the format: Task{name='taskName', priority=priority}.

Main Algorithm used for Priority Queue:

Step 1: Define a Comparator<Task> called taskComparator to compare tasks based on their priority in ascending order using Integer.compare().

Step 2: Create a PriorityQueue<Task> called taskQueue and pass taskComparator to it. Step 3: Add tasks to the taskQueue using the add() method with different priorities.

Step 4: Print "Task Execution Order:" to indicate the start of task execution. Step 5: Repeat Step 6 while the queue is not empty:

Step 6: Retrieve and remove the task with the highest priority using the poll() method, and print the task details.

Step 7: Exit

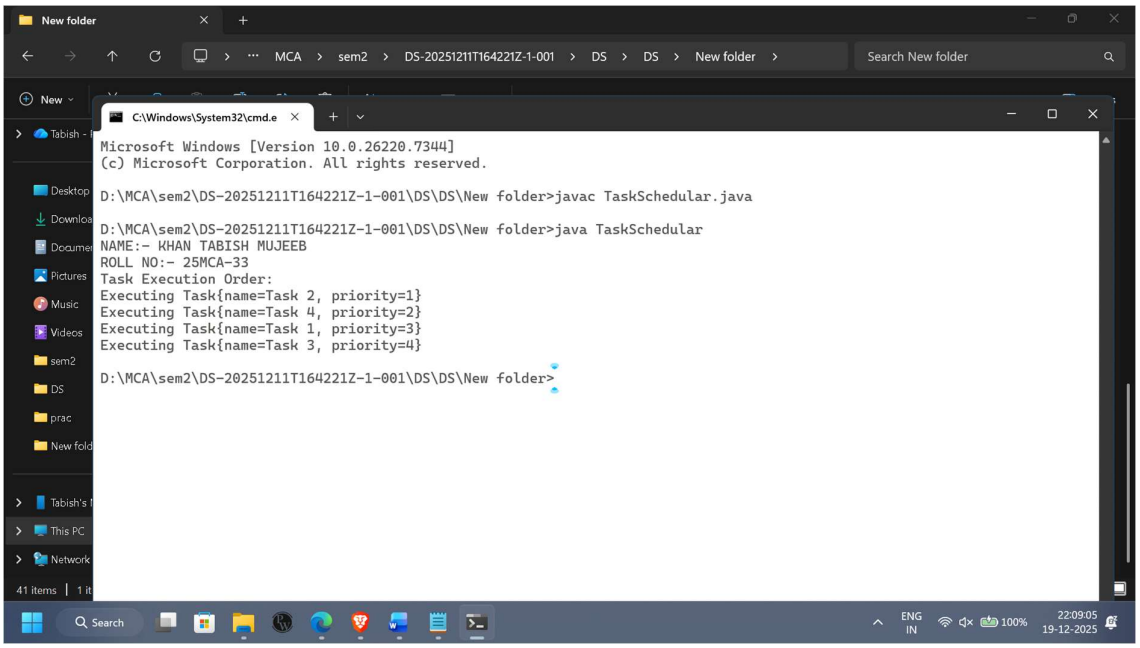
Programming for Priority Queue:

```
import java.util.*;
```

```
public class TaskSchedular
{
   static class Task
{
      private String taskName;
      private int priority;
      public Task(String taskName, int priority) {
         this.taskName = taskName;
         this.priority = priority;
      }
      public String getTaskName()
{
         return taskName;
      }
      public int getPriority()
{
         return priority;
      }
      public String toString()
 {
         return "Task{name=" + taskName + ", priority=" + priority + "}";
      }
   }
   public static void main(String[] args)
{
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
      System.out.print("ROLL NO:- 25MCA-33\n");
      Comparator<Task> taskComparator = new Comparator<Task>() {
         public int compare(Task task1, Task task2) {
            return Integer.compare(task1.getPriority(), task2.getPriority());
         }
      };
```

```
        PriorityQueue<Task> taskQueue = new PriorityQueue<>(taskComparator);
        taskQueue.add(new Task("Task 1", 3));
        taskQueue.add(new Task("Task 2", 1));
        taskQueue.add(new Task("Task 3", 4));
        taskQueue.add(new Task("Task 4", 2));
        System.out.println("Task Execution Order:");
        while (!taskQueue.isEmpty()) {
            Task task = taskQueue.poll();
            System.out.println("Executing " + task);
        }
    }
}
```

Implementation(Output Of The Code):



Practical 10.b

Aim: To implement Breadth-First Search (BFS) traversal of a graph using a Queue and demonstrate how BFS explores all vertices level-by-level.

Theory: Breadth-First Search (BFS) is a graph traversal algorithm that explores nodes in levels.

It starts from a given source vertex, visits all its neighbors first, then moves to the next level neighbors.

Key Concept

BFS uses a Queue (FIFO) to keep track of the nodes to be visited.

Because of FIFO behavior, BFS always visits nodes in the order they were discovered.

Working of BFS

Choose a starting vertex.

Insert it into the queue and mark it as visited.

While the queue is not empty:

Remove (dequeue) the front vertex.

Visit it.

Add (enqueue) all unvisited neighbors of that vertex into the queue.

Continue until all reachable nodes are visited.

Algorithm used for BFS:

BFS(Graph, start):

Step 1: Initialize Queue Q.

Step 2: Initialize visited[] with False for all vertices.

Step 3: Enqueue start vertex into Q. Step 4: Mark visited[start] = True. Step 5:WHILE Q is not empty:

Dequeue vertex u from Q.

Process vertex u (print or store it).

FOR each neighbor v of u: IF visited[v] is False:

Enqueue v into Q. Mark visited[v] = True.

Step 6:End.

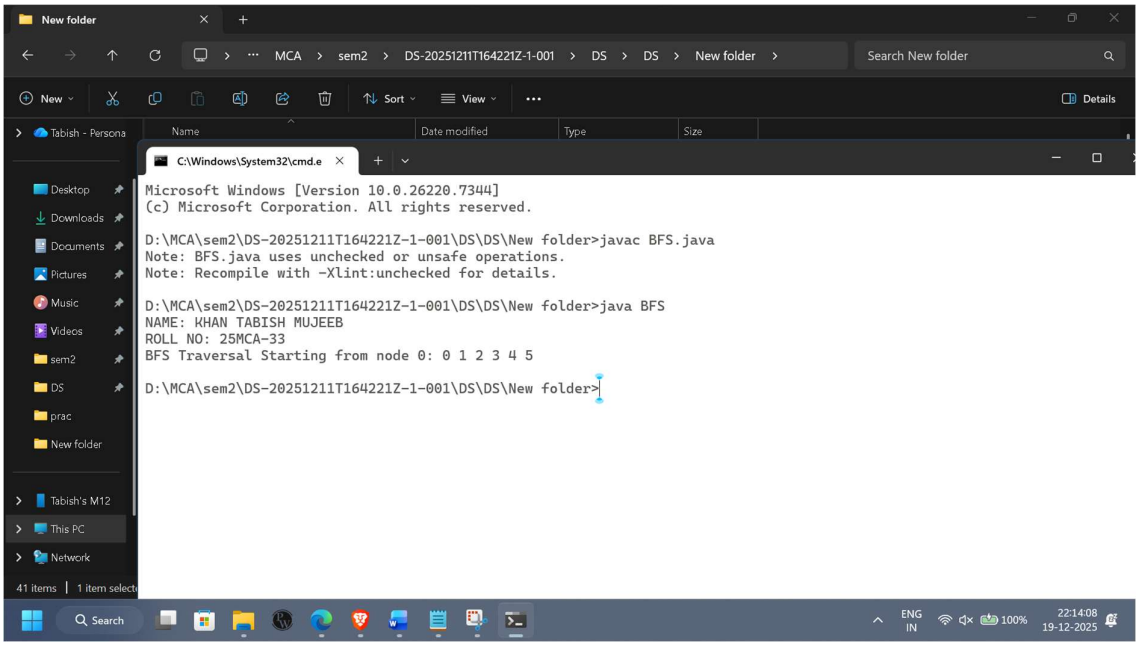
Programming for BFS:

```
import java.util.*;
class Graph {
   private int vertices;
```

```
    private LinkedList<Integer>[] adjList;
    public Graph(int vertices) {
        this.vertices = vertices;
        adjList = new LinkedList[vertices];
        for (int i = 0; i < vertices; i++) {
            adjList[i] = new LinkedList<>();
        }
    }
    public void addEdge(int source, int destination) {
        if (source >= 0 && source < vertices && destination >= 0 && destination < vertices) {
            adjList[source].add(destination);
            adjList[destination].add(source); // For undirected graph
        } else {
            System.out.println("Invalid edge: " + source + " -> " + destination);
        }
    }
    public void bfs(int start) {
        if (start < 0 || start >= vertices) {
            System.out.println("Invalid start node");
            return;
        }
        boolean[] visited = new boolean[vertices];
        Queue<Integer> queue = new LinkedList<>();
        visited[start] = true;
        queue.add(start);
        System.out.print("BFS Traversal Starting from node " + start + ": ");
        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");
            for (Integer neighbor : adjList[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
```

```
                queue.add(neighbor);
            }
        }
    }
    System.out.println();
  }
}
public class BFS {
  public static void main(String[] args) {
    System.out.println("NAME: KHAN TABISH MUJEEB");
    System.out.println("ROLL NO: 25MCA-33");
    Graph gh = new Graph(6);
    gh.addEdge(0, 1);
    gh.addEdge(0, 2);
    gh.addEdge(1, 3);
    gh.addEdge(1, 4);
    gh.addEdge(2, 5);
    gh.bfs(0);
  }
}
```

Implementation(Output Of The Code):



Practical 10.c

Aim: DFS is another fundamental graph traversal algorithm, which explores as far as possible along each branch before backtracking.

Theory of DFS: Depth-First Search (DFS) is a graph traversal algorithm that starts at a source vertex and explores as far as possible along each branch before backtracking. DFS can be implemented using a **stack** (LIFO), which means that DFS explores the most recently discovered vertex that has not yet been fully explored.

Key Concept:

- DFS uses a **stack** (or recursion) to keep track of the nodes to be visited.
- In the case of recursion, the function calls are automatically managed by the system stack.
- DFS explores each vertex's descendants before visiting its neighbors on the same level.

Working of DFS:

1. Choose a starting vertex.
2. Push it onto the stack and mark it as visited.
3. While the stack is not empty:
    - Pop the top vertex.
    - Visit the vertex (process it).
    - Push all unvisited neighbors of that vertex onto the stack.
4. Continue until all reachable nodes are visited.

Algorithm for DFS:

Step 1: Initialize stack S.

Step 2: Initialize visited[] with False for all vertices.

Step 3: Push start vertex onto stack S.

Step 4: Mark visited[start] = True.

Step 5: WHILE stack S is not empty:

a. Pop vertex u from S.

b. Process vertex u (print or store it).

c. FOR each neighbor v of u:

IF visited[v] is False:

Push v onto stack S.

Mark visited[v] = True.

Step 6: End.

Programming for DFS:

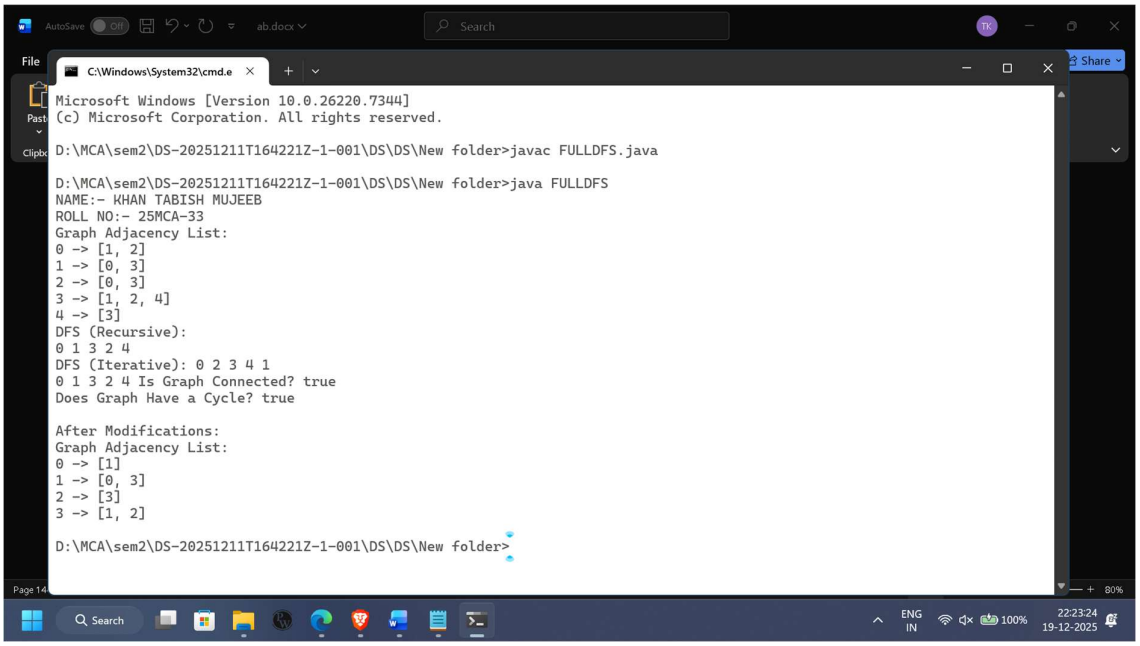
```
import java.util.*;
class Graph {
    private int vertices;
    private Map<Integer, List<Integer>> adj;
    public Graph() {
        this.vertices = 0;
        this.adj = new HashMap<>();
    }
    public void addVertex(int v) {
        if (!adj.containsKey(v)) {
            adj.put(v, new ArrayList<>());
            vertices++;
        }
    }
    public void removeVertex(int v) {
        if (!adj.containsKey(v)) return;
        adj.remove(v);
        vertices--;
        for (int key : adj.keySet()) {
            adj.get(key).remove(Integer.valueOf(v));
        }
    }
    public void addEdge(int src, int dest) {
        addVertex(src);
        addVertex(dest);
        adj.get(src).add(dest);
        adj.get(dest).add(src);
    }
    public void removeEdge(int src, int dest) {
        if (adj.containsKey(src))
            adj.get(src).remove(Integer.valueOf(dest));
        if (adj.containsKey(dest))
```

```
        adj.get(dest).remove(Integer.valueOf(src));
    }
    public void display() {
        System.out.println("Graph Adjacency List:");
        for (int v : adj.keySet()) {
            System.out.println(v + " -> " + adj.get(v));
        }
    }
    public void dfsRecursive(int start) {
        Set<Integer> visited = new HashSet<>();
        System.out.println("DFS (Recursive):");
        dfsUtil(start, visited);
        System.out.println();
    }
    private void dfsUtil(int node, Set<Integer> visited) {
        visited.add(node);
        System.out.print(node + " ");
        for (int neighbor : adj.get(node)) {
            if (!visited.contains(neighbor)) {
                dfsUtil(neighbor, visited);
            }
        }
    }
    public void dfsIterative(int start) {
        Set<Integer> visited = new HashSet<>();
        Stack<Integer> stack = new Stack<>();
        stack.push(start);
        System.out.print("DFS (Iterative): ");
        while (!stack.isEmpty()) {
            int node = stack.pop();
            if (!visited.contains(node)) {
                visited.add(node);
```

```
            System.out.print(node + " ");
        }
        for (int neighbor : adj.get(node)) {
            if (!visited.contains(neighbor)) {
                stack.push(neighbor);
            }
        }
    }
    System.out.println();
}
public Boolean isConnected(int start) {
    Set<Integer> visited = new HashSet<>();
    dfsUtil(start, visited);
    return visited.size() == vertices;
}
public Boolean hasCycle() {
    Set<Integer> visited = new HashSet<>();
    for (int vertex : adj.keySet()) {
        if (!visited.contains(vertex)) {
            if (detectCycleDFS(vertex, -1, visited))
                return true;
        }
    }
    return false;
}
private Boolean detectCycleDFS(int current, int parent, Set<Integer> visited) {
    visited.add(current);
    for (int neighbor : adj.get(current)) {
        if (!visited.contains(neighbor)) {
            if (detectCycleDFS(neighbor, current, visited))
                return true;
        } else if (neighbor != parent) {
```

```
                return true;
            }
        }
        return false;
    }
}
public class FULLDFS {
    public static void main(String[] args) {
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        Graph gh = new Graph();
        gh.addEdge(0, 1);
        gh.addEdge(0, 2);
        gh.addEdge(1, 3);
        gh.addEdge(2, 3);
        gh.addEdge(3, 4);
        gh.display();
        gh.dfsRecursive(0);
        gh.dfsIterative(0);
        System.out.println("Is Graph Connected? " + gh.isConnected(0));
        System.out.println("Does Graph Have a Cycle? " + gh.hasCycle());
        gh.removeEdge(0, 2);
        gh.removeVertex(4);
        System.out.println("\nAfter Modifications:");
        gh.display();
    }
}
```

Implementation(Output Of The Code):



PRACTICAL NO.11

Create a Minimum Spanning Tree using any method

- Prim's Algorithm
- Kruskal's Algorithm

Practical 11.a

Aim: Create a Minimum Spanning Tree using Prim's Algorithm

Theory: A Minimum Spanning Tree (MST) of a connected, weighted, undirected graph is a subset of edges that:

Connects all vertices

Contains no cycles

Has the minimum possible total edge weight

Concept

Start with any vertex.

Repeatedly choose the smallest weight edge that connects a selected vertex to an unselected vertex.

Continue until all vertices are included in the MST.

Algorithm Used for Prim's Algorithm

Step 1: Let N be the set of selected nodes in the tree Initially N()

Step 2. Let M be the set of selected edges in the tree. Initially M()

Step 3: Let 5 be the set of graph edges.

Step 4:Initialize N with the initial vertex (starting node)

Step 5:Repeat While (N] No. of nodes in N

Step 6:Let (u, v) be the least cost edge in 5 such that u & NAND not e Le the edge (u, v) does not create any cycle

Step 7:If there is no such edge Then Goto Step 10 [End 11

Step 8:Add the node to N

Step 9:Add Edge (u, v) to M and Set 55 (u,v) [End Loop)

Step 10: IF IN Then

Print: "M contains the edges of MST Exit

Else

Print: "Graph is not connected and hence no MST can be formed [End If

Step 11: Exit

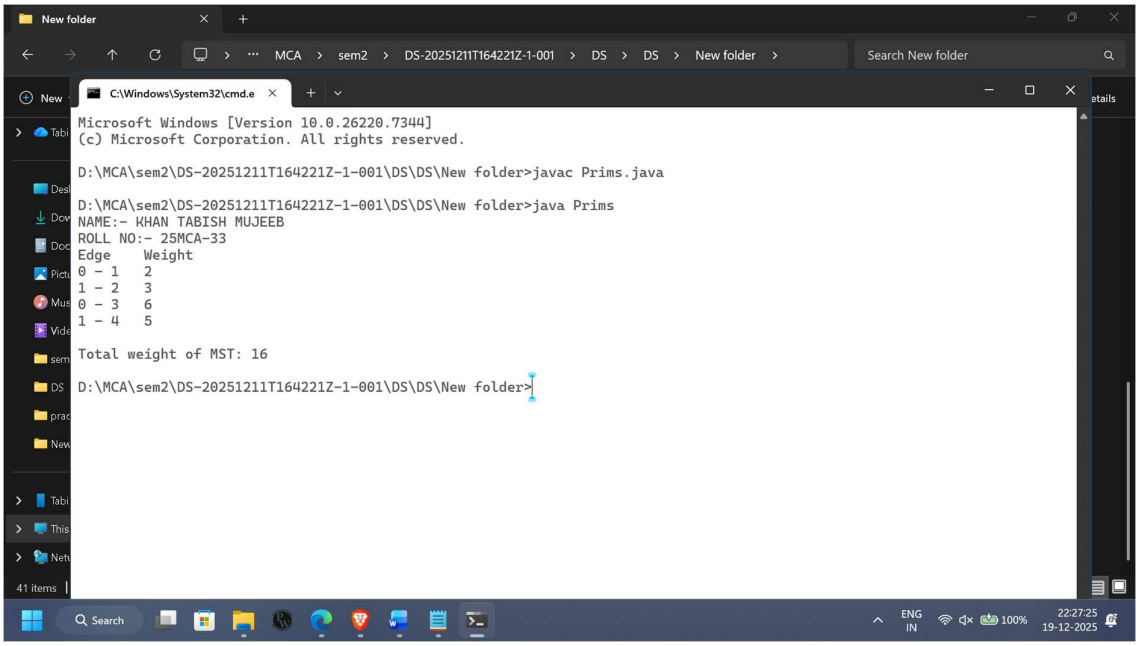
Programming for Prim's Algorithm

```
import java.util.*;
class Graph {
    private int V;
    private int[][] adjMatrix;
    public Graph(int v) {
        this.V = v;
        adjMatrix = new int[V][V];
    }
    public void addEdge(int u, int v, int weight) {
        adjMatrix[u][v] = weight;
        adjMatrix[v][u] = weight;
    }
    public void primMST() {
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node ->
node.weight));
        int[] parent = new int[V];
        int[] key = new int[V];
        boolean[] inMST = new boolean[V];
        Arrays.fill(key, Integer.MAX_VALUE);
        Arrays.fill(inMST, false);
        key[0] = 0;
        parent[0] = -1;
        pq.add(new Node(0, 0));
        while (!pq.isEmpty()) {
            int u = pq.poll().vertex;
            inMST[u] = true;
            for (int v = 0; v < V; v++) {
                if (adjMatrix[u][v] != 0 && !inMST[v] && adjMatrix[u][v] < key[v]) {
                    key[v] = adjMatrix[u][v];
                    parent[v] = u;
                    pq.add(new Node(v, key[v]));
                }
```

```
            }
        }
        printMST(parent);
    }
    private void printMST(int[] parent) {
        System.out.println("Edge \tWeight");
        int totalWeight = 0;
        for (int i = 1; i < V; i++) {
            System.out.println(parent[i] + " - " + i + "\t" + adjMatrix[i][parent[i]]);
            totalWeight += adjMatrix[i][parent[i]];
        }
        System.out.println("\nTotal weight of MST: " + totalWeight);
    }
    static class Node {
        int vertex;
        int weight;
        public Node(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }
    }
}
public class Prims{
    public static void main(String[] args) {
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        Graph graph = new Graph(5);
        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 3, 6);
        graph.addEdge(1, 2, 3);
        graph.addEdge(1, 3, 8);
        graph.addEdge(1, 4, 5);
```

```
        graph.addEdge(2, 4, 7);
        graph.addEdge(3, 4, 9);
        graph.primMST();
    }
}
```

Implementation(Output Of The Code):



Practical 11.b

Aim: Create a Minimum Spanning Tree using Kruskal's Algorithm

Theory: Kruskal's Algorithm is also a greedy algorithm, but it works by sorting all edges first and then adding edges to the MST in increasing order of weight, ensuring no cycle is formed.

Concept

Sort all edges in ascending order.

Pick the smallest weight edge.

Add it to the MST only if it does not form a cycle.

Cycle detection is done using Disjoint Set (Union-Find).

Repeat until MST contains V – 1 edges.

Algorithm Used for Kruskal's Algorithm:

Step 1: Let M be the set of selected edges that will make the MST. Initially M = (φ)

Step 2: Let S be the set graph edges.

Remaining Edges are Null in S or No. of Edges in M aren - 1

Step 3: Repeat while S AND Mn-1

Let (uv) be the least cost edge in S

Set S S - (uv)

// Delete the edge (uv) from S

If (uv) does not create cycle in M Put the edge (uv) into M

[End If] [End Loop]

Step 4:If Mn-1 Then Print: "M is the MST"

//M contains n-1 edges Else

Print: "Graph is not connected and hence no MST can be formed [End If]

Step 5: Exit

Programming for Kruskal's Algorithm:

```
// Kruskal MST Implementation in Java
import java.util.*;
class DisjointSet {
   private int[] parent, rank;
   public DisjointSet(int n) {
      parent = new int[n];
```

```
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }
    public int find(int u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }
    public void union(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);
        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
}
class Edge implements Comparable<Edge> {
    int u, v, weight;
    public Edge(int u, int v, int weight) {
        this.u = u;
        this.v = v;
```

```
        this.weight = weight;
    }
    @Override
    public int compareTo(Edge other) {
        return Integer.compare(this.weight, other.weight);
    }
}
public class KruskalMST {
    public static List<Edge> kruskal(int n, List<Edge> edges) {
        Collections.sort(edges);
        DisjointSet ds = new DisjointSet(n);
        List<Edge> mst = new ArrayList<>();
        for (Edge edge : edges) {
            int u = edge.u;
            int v = edge.v;
            if (ds.find(u) != ds.find(v)) {
                ds.union(u, v);
                mst.add(edge);
            }
        }
        return mst;
    }
    public static void main(String[] args) {
System.out.print("NAME:- KHAN TABISH MUJEEB\n");
        System.out.print("ROLL NO:- 25MCA-33\n");
        int n = 4;
        List<Edge> edges = new ArrayList<>();
        edges.add(new Edge(0, 1, 10));
        edges.add(new Edge(0, 2, 6));
        edges.add(new Edge(0, 3, 5));
        edges.add(new Edge(1, 3, 15));
        edges.add(new Edge(2, 3, 4));
```

```
        List<Edge> mst = kruskal(n, edges);
        System.out.println("Minimum Spanning Tree (MST): ");
        int totalWeight = 0;
        for (Edge edge : mst) {
            System.out.println("Edge: " + edge.u + " - " + edge.v + " | Weight: " + edge.weight);
            totalWeight += edge.weight;
        }
        System.out.println("\nTotal weight of MST: " + totalWeight);
    }
}
```

Implementation(Output Of The Code):