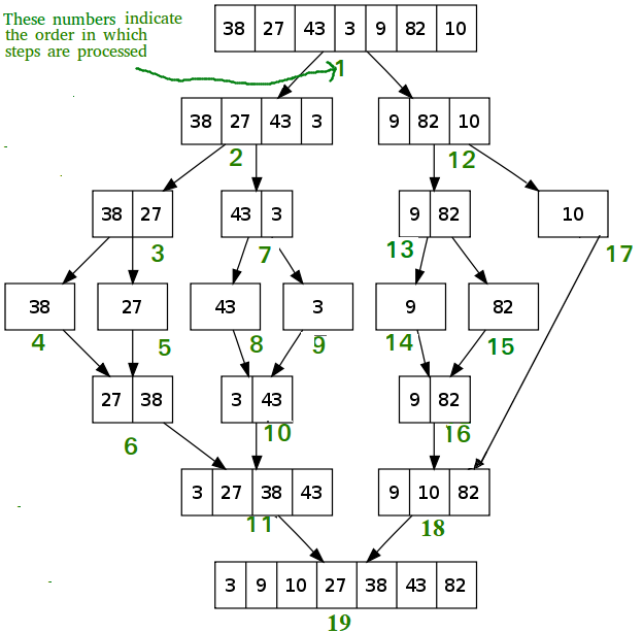
```
void merge(int array[], int const left, int const mid,
           int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0, // Initial index of first sub-array
        indexOfSubArrayTwo = 0; // Initial index of second sub-array
    int indexOfMergedArray = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne
           && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is  right index of the sub-array of arr to be sorted
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```



Pseudocodestep 1: start

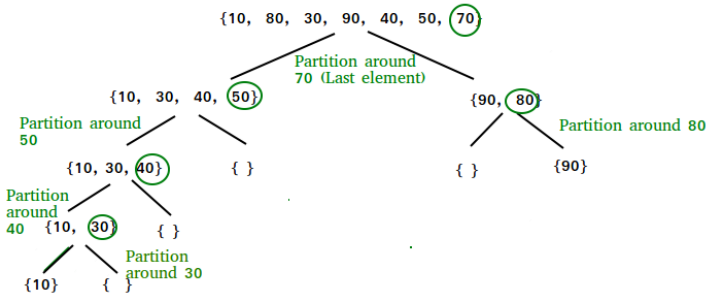
step 2: declare array and left, right, mid variable

step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)

step 4: Stop

Quick Sort

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

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all
smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
           - 1); // Index of smaller element and indicates the right position of pivot found so far

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

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```



Interpolation Search vs binary search

```
// C++ program to implement interpolation search
#include<bits/stdc++.h>
using namespace std;

// If x is present in arr[0..n-1], then returns
// index of it, else returns -1.
int interpolationSearch(int arr[], int n, int x)
{
 // Find indexes of two corners
 int lo = 0, hi = (n - 1);

 // Since array is sorted, an element present
 // in array must be in range defined by corner
 while (lo <= hi && x >= arr[lo] && x <= arr[hi])
 {
  if (lo == hi)
  {
   if (arr[lo] == x) return lo;
   return -1;
  }
  // Probing the position with keeping
  // uniform distribution in mind.
  int pos = lo + (((double)(hi - lo) /
   (arr[hi] - arr[lo])) * (x - arr[lo]));

  // Condition of target found
  if (arr[pos] == x)
   return pos;

  // If x is larger, x is in upper part
  if (arr[pos] < x)
   lo = pos + 1;

  // If x is smaller, x is in the lower part
  else
   hi = pos - 1;
 }
 return -1;
}

// Driver Code
int main()
{
 // Array of items on which search will
 // be conducted.
 int arr[] = {10, 12, 13, 16, 18, 19, 20, 21,
    22, 23, 24, 33, 35, 42, 47};
 int n = sizeof(arr)/sizeof(arr[0]);

 int x = 18; // Element to be searched
 int index = interpolationSearch(arr, n, x);

 // If element was found
 if (index != -1)
  cout << "Element found at index " << index;
 else
  cout << "Element not found.";
 return 0;
}
```

Data Structures Lab

Session 9

Note:

- Lab manual cover following below Stack and Queue topics
  {Stack with Array and Linked list , Application of Stack, Queue with Array and Linked List , Application of Queue }
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Stack with Array

Sample Code of Stack in Array

```
class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
```

```
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}
```

Task-1:

A. Design a Main class of upper code which perform the below task

1. Insert 10 Integers values in the stack
2. If the Insert input reach the Highest index of Array display the message Stack overflow
3. Remove the Inserted values till the Last value and print the message that stack is empty

Stack with Linked list

Sample Code of Stack in Array

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

struct Node* top;

// Utility function to add an element
// data in the stack insert at the beginning
void push(int data)
{

   // Create new node temp and allocate memory
   struct Node* temp;
   temp = new Node();

   // Check if stack (heap) is full.
   // Then inserting an element would
   // lead to stack overflow
   if (!temp)
   {
      cout << "\nHeap Overflow";
      exit(1);
   }

   // Initialize data into temp data field
   temp->data = data;

   // Put top pointer reference into temp link
   temp->link = top;

   // Make temp as top of Stack
   top = temp;
}
```

Task-2:

A. Design a Main class of upper code which perform the below task

1. Insert 10 Integers values in the stack
2. Write a utility function for upper code to display all the inserted integer values in the linked list in forward and reverse direction both
3. Write utility function to pop top element from the stack

Application of Stack (convert infix expression to postfix)

Sample Pseudocode

```
Begin
  initially push some special character say # into the stack
  for each character ch from infix expression, do
    if ch is alphanumeric character, then
      add ch to postfix expression
    else if ch = opening parenthesis (, then
      push ( into stack
    else if ch = ^, then          //exponential operator of higher precedence
      push ^ into the stack
    else if ch = closing parenthesis ), then
      while stack is not empty and stack top ≠ (,
        do pop and add item from stack to postfix expression
      done

      pop ( also from the stack
    else
      while stack is not empty AND precedence of ch <= precedence of stack top element, do
        pop and add into postfix expression
      done

      push the newly coming character.
  done

  while the stack contains some remaining characters, do
    pop and add to the postfix expression
  done
  return postfix
End
```

Code Snippet

```
#include<bits/stdc++.h>
using namespace std;
//Function to return precedence of operators
int prec(char c)
{
if(c == '^')
return 3;
```

```
else if(c == '*' || c == '/')
return 2;
else if(c == '+' || c == '-')
return 1;
else
return -1;
}



int main()
{
   string exp = "a+b*(c^d-e)^(f+g*h)-i";
   infixToPostfix(exp);
   return 0;
}
```

Task-3:

A. Use the Upper code snippet implement the utility function with the help of array based stack **infixToPostfix** by using sample pseudocode

Queue with Array

```
using namespace std;

// A structure to represent a queue
class Queue {
public:
   int front, rear, size;
   unsigned capacity;
   int* array;
};

// function to create a queue
// of given capacity.
// It initializes size of queue as 0
Queue* createQueue(unsigned capacity)
{
   Queue* queue = new Queue();
   queue->capacity = capacity;
   queue->front = queue->size = 0;

   // This is important, see the enqueue
   queue->rear = capacity - 1;
   queue->array = new int[queue->capacity];
   return queue;
}
```

Task-4:

A. Use the Upper code snippet implement the following utility function in the Array based Queue
    1. Write a function **QueueCapacity** when the Queue is Full
    2. Write a function **ADDMember** when a new integer value is added in the array
    3. Write a function **RemoveMember** when any data member is remove from the queue

Queue with Linked list

Sample Code

```
#include <iostream>
using namespace std;
struct node {
  int data;
  struct node *next;};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert() {
  int val;
  cout<<"Insert the element in queue : "<<endl;
  cin>>val;
  if (rear == NULL) {
    rear = (struct node *)malloc(sizeof(struct node));
    rear->next = NULL;
    rear->data = val;
    front = rear;
  } else {
    temp=(edlist struct node *)malloc(sizeof(struct node));
    rear->next = temp;
    temp->data = val;
    temp->next = NULL;
  rear = temp;
  }}
```

Task-5:

B. Use the Upper code snippet implement the following utility function in the Link based Queue
    4. Write a function **QueueCapacity** when the Queue is Full
    5. Write a function **ADDMember** when a new integer value is added in the linkedlist
    6. Write a function **RemoveMember** when any data member is remove from the queue

Application of Queue

Task-6: Read the Web link https://algs4.cs.princeton.edu/24pq/ .Implement the Elementary implementation of Priority queue using linked list as explain in Figure "Sequence of Operation in Priority Queue" .
Note: Your Base code will be in C++

Reference:

Stack with Array and Linked list

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

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

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

Application of Stack

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Table-1

Queue with Array and Linked list

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Following are the Operations on Queue

1.Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

2.Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

3.Front: Get the front item from queue.

4.Rear: Get the last item from queue.

Applications of Queue

A priority queue in c++ is a type of container adapter, which processes only the highest priority element, i.e. the first element will be the maximum of all elements in the queue, and elements are in decreasing order.

Difference between a queue and priority queue :

- Priority Queue container processes the element with the highest priority, whereas no priority exists in a queue.
- Queue follows First-in-First-out (FIFO) rule, but in the priority queue highest priority element will be deleted first.
- If more than one element exists with the same priority, then, in this case, the order of queue will be taken.

1. **empty()** – This method checks whether the priority_queue container is empty or not. If it is empty, return true, else false. It does not take any parameter.

2. **size()** – This method gives the number of elements in the priority queue container. It returns the size in an integer. It does not take any parameter.

3. **push()** – This method inserts the element into the queue. Firstly, the element is added to the end of the queue, and simultaneously elements reorder themselves with priority. It takes value in the parameter.

4. **pop()** – This method delete the top element (highest priority) from the priority_queue. It does not take any parameter.

5. **top()** – This method gives the top element from the priority queue container. It does not take any parameter.

6. **swap()** – This method swaps the elements of a priority_queue with another priority_queue of the same size and type. It takes the priority queue in a parameter whose values need to be swapped.

7. **emplace()** – This method adds a new element in a container at the top of the priority queue. It takes value in a parameter.

Lab3: Stack and Queue (using both Arrays and Linked List)		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		