**Data Structures Lab 8**

**Course:** Data Structures (CL2001) **Semester:** Fall 2022

**Instructor:** Muhammad Monis  **T.A:** N/A

**Note:**

* + - * Lab manual cover following below Stack and Queue topics

**{Stack with Array and Linked list ,Queue with Array and Linked, Tree, BST, Design and implement classes for binary tree nodes and nodes for general tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}**

* 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;

    }

}

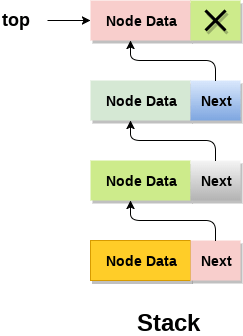
bool Stack::isEmpty()

{

    return (top < 0);

}

**Stack with Linked list**



**Sample Code of Stack in Linked List**

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-1:**

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

### 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

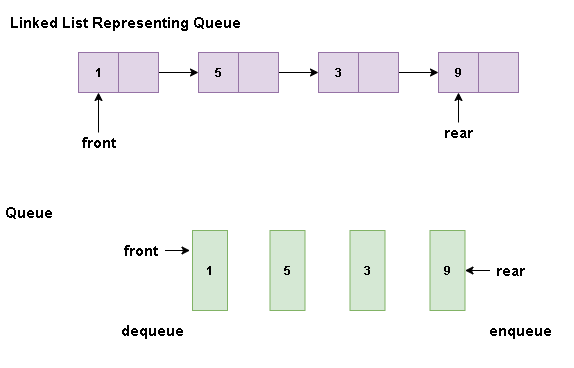
queue->rear = capacity - 1;

queue->array = new int[queue->capacity];

return queue;

}

### 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-2:**

1. Use the Upper code snippet implement the following utility function in the Link based Queue
2. Write a function **ADDMember** when a new integer value is added in the linkedlist
3. Write a function **RemoveMember** when any data member is removing from the queue

**BINARY SEARCH TREE**

**KeyPoint**: A Binary Search Tree (BST) is a binary tree with the following properties:

* The left subtree of a particular node will always contain nodes whose keys are less than that node’s key.
* The right subtree of a particular node will always contain nodes with keys greater than that node’s key. The left and right subtree of a particular node will also, in turn, be binary search trees

**BST Insertion**

**Sample Code of class Nodes**

Create class Nodes

class Node { private:

int key;

string name;

Node leftChild;

Node rightChild; public:

Node(int key, string name) {

this.key = key;

this.name = name;

}

string toString() {

return cout<<name<< " has the key " <<key<<endl;

} };

**Task-3: Complete the following Code:**

**Create class BinaryTree and create a function which add nodes in BST**

class BinaryTree {

private: Node root;

public:

void addNode(int key, string name) {

-----------------------// Create a new Node and initialize it

// If there is no root this becomes root

if (root == NULL) {

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

} else {

// Set root as the Node we will start with as we traverse the tree

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

// Future parent for new Node

Node parent;

while (true) {

// root is the top set the parent node to the root node

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

// Check if the new node should go on

// the left side of the parent node

Key is compared with that of root. If the key is less than root, it is compared with root’s left child key. If greater, it is compared with the root's right child. Continue this process until the new node is compared with a leaf node and added either on the right or left child depending on its key.

}

**Implement main.cpp for the code provided such that a given array is passed to form a BST{ 15, 10, 20, 8, 12, 16, 25 }**

**Tree Traversals: Inorder, PreOrder, PostOrder**

**Pseudo code For Inorder Traversal (iteration)**

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

a) Pop the top item from stack.

b) Print the popped item, set current = popped\_item->right

c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

**Task-4:**

**Write recursive algorithms that perform preorder and inorder tree walks.**

**Preorder Traversal approach.**

1. Visit Node.
2. Traverse Node’s left sub-tree.
3. Traverse Node’s right sub-tree

**BST Deletion**

**BST Deletion**

**1) *Node to be deleted is the*** ***leaf:*** Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

**2) *Node to be deleted has only one child:*** Copy the child to the node and delete the child

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

**3) *Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

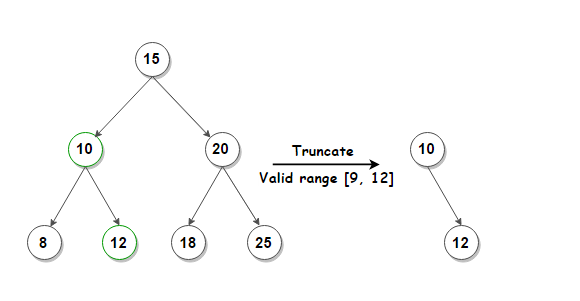
40 70 ---------> 40 70

/ \ \

60 80 80

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

**Task:5**Given a BST and a range of keys(values), remove nodes from BST that have keys outside the given range



### Some points to Note:

### 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**

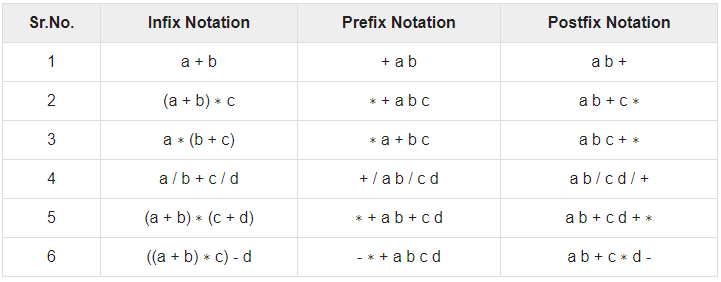
We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly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



**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](https://www.geeksforgeeks.org/stack-data-structure/)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 :**

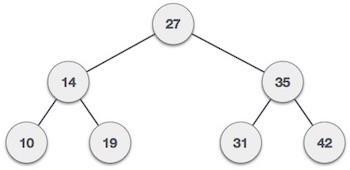
* Priority Queue container processes the element with the highest priority, whereas no priority exists in a queue.
* Queue follows First-in-First-out (FIFO) rule, but in the priority queue highest priority element will be deleted first.
* If more than one element exists with the same priority, then,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

### Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

* Each node has a value.
* The key value of the left child of a node is less than to the parent's key value.
* The key value of the right child of a node is greater than (or equal) to the parent's key value.
* And these properties hold true for every node in the tree.



* **Subtree**: any node in a tree and its descendants.
* **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
* **Depth of a tree**: the maximum depth of any of its leaves.
* **Height of a node**: the length of the longest downward path to a leaf from that node.
* **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
* **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
* **Traversal**: an organized way to visit every member in the structure.

**Traversals**

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.   
The algorithms are described below, with Node initialized to the tree’s root.

• **Preorder Traversal**

1. Visit Node.
2. Traverse Node’s left sub-tree.
3. Traverse Node’s right sub-tree.

**• In-order Traversal**

1. Traverse Node’s left sub-tree.
2. Visit Node.
3. Traverse Node’s right sub-tree

**• Post-order Traversal**

1. Traverse Node’s left sub-tree.
2. Traverse Node’s right sub-tree.
3. Visit Node

### Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k,* TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

TREE-SEARCH (*x, k*)

1 **if** *x* = NIL or *k = key*[*x*]

2 **then return** *x*

3 **if** *k < key*[*x*]

4 **then return** TREE-SEARCH *(left*[*x*], *k)*

5 **else return** TREE-SEARCH *(right*[*x*], *k)*

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node *x* it encounters, it compares the key *k* with *key*[*x*]*.* If the two keys are equal, the search terminates. If *k* is smaller than *key*[*x*]*,* the search continues in the left subtree of *x,* since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *key*[*k*]*,* the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is *O*(*h*)*,* where *h* is the height of the tree.