Data Structures Lab
Session 8

Course: Data Structures (CS2001) **Semester:** Fall 2022
Instructor: Eman Shahid **T.A:** N/A

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, Tree, Binary Tree, Tree Traversal }
- 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 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;
}
```

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;
}
```

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;
```

}}

Tree Data Structure

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

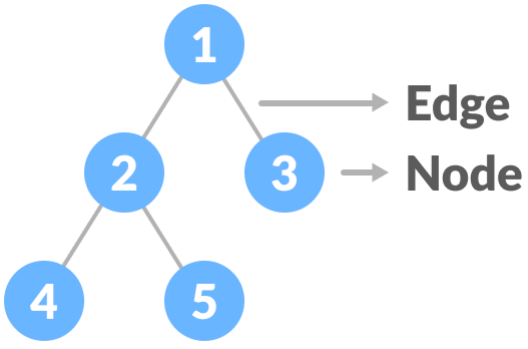
Node

A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes.



Nodes and edges of a tree

Root

It is the topmost node of a tree.

Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

Height and depth of each node in a tree
Creating forest from a tree

You can create a forest by cutting the root of a tree.

Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

To learn more, please visit tree traversal.

Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.

- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

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.

Lab8: 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		
Task# 6		