Assignment #02



|  |  |
| --- | --- |
| **Submitted By:** | **Burhan Riaz** |
| **Roll Number:** | **SP20-MCS-036** |
| **Lab Instructor:** | **Nashit Ali** |
| **Subject:** | **Data Structure** |

A ASSIGNMENT SUBMITTED FULFILMENT OF THE REQUIERMENT FOR THE DEGGRE OF MACTER OF COMPUTER SCINECE (November 13, 2020)

Instructor: Mr. Nashit Ali

Department of Computer science

COMSATAS University Islamabad, Vehari Campus, Pakistan

Table of Contents

[Stack Data Structure: 3](#_Toc55940214)

[Stack Syntax: 3](#_Toc55940215)

[Member Types: 3](#_Toc55940216)

[Operations in Stack 4](#_Toc55940217)

[Complete code of Stack: 4](#_Toc55940218)

[Infix To Prefix & Postfix Conversion 7](#_Toc55940219)

[Introduction 7](#_Toc55940220)

[Infix Notation 7](#_Toc55940221)

[Postfix Notation 7](#_Toc55940222)

[Prefix Notation 7](#_Toc55940223)

[Algorithm (For Code/Manual Calculation) 7](#_Toc55940224)

[Queue Data Structure 9](#_Toc55940225)

[What is a queue? 9](#_Toc55940226)

[Types of Queues in Data Structure 9](#_Toc55940227)

[Simple Queue: 9](#_Toc55940228)

[Circular Queue: 9](#_Toc55940229)

[Priority Queue: 9](#_Toc55940230)

[Dequeue (Doubly Ended Queue): 10](#_Toc55940231)

[Basic operations performed on a Queue: 10](#_Toc55940232)

[Complete Code of Queue: 10](#_Toc55940233)

Stack Data Structure:

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).

The std::stack class is a container adapter. Container objects hold data of a similar data type. You can create a stack from various sequence containers. If no container is provided, the deque containe will be used by default. Container adapters don't support iterators, so it can't be used to manipulate data.

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

Stack Syntax:

To create a stack, we must include the <stack> header file in our code. We then use this syntax to define the std::stack:

template <class Type, class Container = deque<Type> > class stack;

What is Database SQL

* **Type** – is the Type of element contained in the std::stack. It can be any valid C++ type or even a user-defined type.
* **Container** – is the Type of underlying container object.

Member Types:

Here are stack member types:

* value\_type- The first template parameter, T. It denotes the element types.
* container\_type- The second template parameter, Container. It denotes the underlying container type.
* size\_type- Unsigned integral type.

Operations in Stack

* push – It adds/pushes an item into the stack.
* pop – It removes/pops an item from the stack.
* peek – Returns the top item of the stack without removing it.
* isFull – Checks whether a stack is full.
* isEmpty – Checks whether a stack is empty.

Complete code of Stack:

/\* C++ program to implement basic stack

operations \*/

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

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);

}

// Driver program to test above functions

int main()

{

class Stack s;

s.push(10);

s.push(20);

s.push(30);

cout << s.pop() << " Popped from stack\n";

return 0;

}

Infix To Prefix & Postfix Conversion

Introduction

Operation: Any Expression of algebraic format (Example : A + B)

Operands: A and B or 5 & 6 are operands

Operators: +. -, %,\*,/ etc are operators

Infix Notation

Infix is the day to day notation that we use of format A + B type. The general form can be classified as (a op b) where a and b are operands(variables) and op is Operator.

Example 1: A + B

Example 2: A \* B + C / D

Postfix Notation

Postfix is notation that compiler uses/converts to while reading left to right and is of format AB+ type. The general form can be classified as (ab op) where a and b are operands (variables) and op is Operator.

Example 1: AB+

Example 2: AB\*CD/+

Prefix Notation

Prefix is notation that compiler uses/converts to while reading right to left (some compilers can also read prefix left to right) and is of format +AB type. The general form can be classified as (op ab) where a and b are operands (variables) and op is Operator.

Example 1: +AB

Example 2: +\*AB/CD

Algorithm (For Code/Manual Calculation)

1. First Start scanning the expression from left to right
2. If the scanned character is an operand, output it, i.e. print it
3. Else

* If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'(‘), then push operator in the stack
* Else, Pop all the operators, that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)

1. If the scanned character is an ‘(‘, push it to the stack.
2. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
3. Now, we should repeat the steps 2 – 6 until the whole infix i.e. whole characters are scanned.
4. Print output
5. Do the pop and output (print) until stack is not empty

Queue Data Structure

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.

****What is a queue?****

A Queue is a FIFO (First In First Out) data structure where the element that is added first will be deleted first. The basic queue operations are in enqueue (insertion) and dequeue (deletion). Enqueue is done at the front of the queue and dequeue is done at the end of the queue. The elements in a queue are arranged sequentially and hence queues are said to be linear data structures.

Basic operations performed on a Queue:

|  |  |
| --- | --- |
| **Operations** | **Description** |
| enqueue() | This function defines the operation for adding an element into queue. |
| dequeue() | This function defines the operation for removing an element from queue. |
| init() | This function is used for initializing the queue. |
| Front | Front is used to get the front data item from a queue. |
| Rear | Rear is used to get the last item from a queue. |

****Types of Queues in Data Structure****

Queue in data structure is of the following types

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

****Simple Queue:****

The simple queue is a normal queue where insertion takes place at the **FRONT** of the queue and deletion takes place at the **END** of the queue.

****Circular Queue:****

* In a circular queue, the last node is connected to the first node.
* Circular queue is also called as**Ring Buffer.**
* Insertion in a circular queue happens at the **FRONT** and deletion at the **END** of the queue.

****Priority Queue:****

* In a priority queue, the nodes will have some predefined priority.
* Insertion in a priority queue is performed in the order of arrival of the nodes.
* The node having the least priority will be the first to be removed from the priority queue.

****Dequeue (Doubly Ended Queue):****

In a Double Ended Queue, insertion and deletion operations can be done at both**FRONT** and**END** of the queue.

Example:

Complete Code of Queue:

#include<iostream>

using namespace std;

#define SIZE 10

class Queue

{

int a[SIZE];

int rear; //same as tail

int front; //same as head

public:

Queue()

{

rear = front = -1;

}

//declaring enqueue, dequeue and display functions

void enqueue(int x);

int dequeue();

void display();

};

// function enqueue - to add data to queue

void Queue :: enqueue(int x)

{

if(front == -1) {

front++;

}

if( rear == SIZE-1)

{

cout << "Queue is full";

}

else

{

a[++rear] = x;

}

}

// function dequeue - to remove data from queue

int Queue :: dequeue()

{

return a[++front]; // following approach [B], explained above

}

// function to display the queue elements

void Queue :: display()

{

int i;

for( i = front; i <= rear; i++)

{

cout << a[i] << endl;

}

}

// the main function

int main()

{

Queue q;

q.enqueue(10);

q.enqueue(100);

q.enqueue(1000);

q.enqueue(1001);

q.enqueue(1002);

q.dequeue();

q.enqueue(1003);

q.dequeue();

q.dequeue();

q.enqueue(1004);

q.display();

return 0;

}