**Beteab Gebru**

**Mar 29 2017**

**CSCI 301 - Section 02**

**QUEUEING SIMULATION**

**Project 8: Program Documentation**

### Introduction

INTRODUCTION

A **queue** is a sequence of elements, all of the same type, to which elements can be appended at one end (the **rear** of the queue) and from which elements can be removed at the other end (the **front**). Queues are first-in-first-out (FIFO) structures. They mimic the behavior of such systems as people in lines, cars at traffic lights, and files to be printed.

Systems that involve queues are called **queueing systems**. A queueing system consists of one or more queues of elements waiting to be **served** by one or more **servers**. When an element is removed from the front of a queue, a server serves that element. How queues and servers interact and parameters such as the numbers of queues and servers, how often new elements arrive, and when items are dequeued and served determine the behavior of a queueing system.

A **queueing simulation** is a program that simulates a queueing system. A **probabilistic** simulation calls a pseudo-random number generator to determine if events occur at each tick of the simulation's clock and the details of those events.

**Design Document**

The idea is to create an abstract data types that can represent objects in a queue and the server/tellers which serve their needs.

-A server is represented by a class, with capability to serve a customer per a probabilistic randomized service time, which is less than the maximum service time allowed.

-A customer in queue, waiting to be served is represented by a node in Class Queue. The customer has data values of Arrival time and max processing time

-A class Queue represents a linked list of nodes that represent queues customers

The Lines of varying lengths are represented by array of queues declared in the main program. The booths with tellers that serve customers are represented by the array of servers of the same number as lines**.**

**Data structures**

The stack ADT will have the ability to store/push items up to the capacity of the array structure.

**Data Members And Functions of the Stack ADT**

#include <cstdlib> // Provides the type size\_t.

#include <string>

using namespace std;

typedef int Item; //item type def

static const int CAPACITY = 100;//potential to store 100 prime factors

class Stack

{

public:

Stack(); //constructor

~Stack(){{ while (used >= 0) used--; } //desttructor

bool is\_empty() const; //returns true if stack is empty

int get\_used() const;//returns the number of items in stack

int get\_smallest();//returns the number of items in stack

Item peek() const;

void push(const Item entry);

Item pop();

bool IsPrime(int); //returns true if prime

void Prime\_Factors(int); //prints out the prime number

friend std::ostream& operator << (std::ostream& Output, Stack& Chain); //overloads the '<<' operator

private:

Item data[CAPACITY];

Item smallest=0;

int used;

};

**Functions**

**-The Constructor**

The constructor in the program simply initializes the Data-member used=0;

**~The Destructor**

-simply takes used down to 0;

**-Peek()**

This function peeks and returns the topmost item on the Stack. Before the peeking it asserts that we are not dealing with an empty stack

**-get\_used()**

This function returns the number of items on stack

**-Is\_empty()**

This function will return true when used==0

-**get\_smallest()**

This function will return the smallest prime factor of the number. The variable smallest is updated when the first factor is loaded onto the stack

**-POP()**

-This function will be used to extract the topmost item on stack. We use this until we get an empty stack which is represented by used<0. When the last item is poped used will be -1.

**-Is\_Prime()**

This function will check for primality of an item so we don’t waste time looking for prime factors.

**Friend function**

-This is the overloaded output stream function that will be able to take the Stack ADT and display its data elements . In my case I have decided to output the elements in a tabular format

**// Example program**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**srand(1);**

**int random[50];**

**int x=0;**

**while(x<50)**

**{**

**random[x]=rand();**

**int prob = random[x] % 100 ;**

**if ( prob< 25)**

**cout<<"probability < 80%->"<<prob<<endl;**

**x++;**

**}**

**x=0;**

**while(x < 50)**

**{**

**cout<<random[x]<<":";**

**if(x%5==0 && x>0)**

**cout<<endl;**

**x++;**

**}**

**}**

**The Main program**

The main program will simply use the test data we already setup to test program functionality. After it processes test data it will ask for user input for additional user tests. The results of which I have displayed below

### User Document The way to run the program

A program is run from the IDE or simply using G++ and it will prompt for user input after it displays the test data results. The user input is processed and the program displays the prime factorization and also show the smallest prime factor.

### Testing of the program ::Screen shot from my visual studio IDE

|  |  |  |  |
| --- | --- | --- | --- |
|  | Number Input | Prime factorisation | Program output |
| 1 | 1776 | = 37 x 3 x 2 x 2 x 2 x 2 | Same outptut |
| 2 | 6463 | = 281 x 23 | Same outptut |
| 3 | 349856 | = 29 x 29 x 13 x 2 x 2 x 2 x 2 x 2 | Same outptut |
| 4 | 352170 | = 43 x 13 x 7 x 5 x 3 x 3 x 2 | Same outptut |
| 5 | 726345 | = 16141 x 5 x 3 x 3 | Same outptut |
| 6 | 36423479 | = 36423479 | Same outptut |
| 7 | 1 | Nothing instack | Same outptut |
| 8 | 0 | stop | Same outptut |

### Summary

The program works the way it should and displays all prime factors. I found out during testing we can make the program faster by changing the upper limit of the prime factor search as we move along rather than testing all numbers up to the number itself. On average the factors can be found between 2 and sqrt(number) but there are cases where we need to go above that limit to find the next prime factor

**Conclusion**

The main conclusions I can make from this project is that stacks could be a very useful ADT in data processing. The other conclusion is the efficiency improvements can be made by checking if we need to continue with the process. Using product of prime factors variable enables the program to cut short its search for prime factors.