**Beteab Gebru**

**April 06 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 the customers in the queue.

* **A server ADT** is represented by a class **Server Class**

**Data-Members – Private:**

-It can serve a customer for **int Transact\_Time** ( rand()% Max\_Service +1) ticks/sec, which is a random number between 1 - Max\_Service\_Time.

-A server ADT will have another data member (**bool Availability**) indicating **availability of the server**

**Class Functions - Public**

**Server(); ~Server(); //** CONSTRUCTOR && DESTRUCTOR

- **void Set\_Availability(bool status) :**sets the server to available when the customer is done getting a service (indicated by transaction time getting reduced zero)

**-bool Is\_Available() :**returns true when the teller is available for dequeued customers

**- void Set\_Trans\_Time(int Trans\_Time) :**sets the transaction duration; takes time(ticks of the loop) variable as a parameter

**- int Get\_Trans\_Time() :retrieves the transaction time left on the tellers clock**

**- void CLock\_Tick() :used to tick the clock on the teller’s desk when the loop clock ticks**

* **A Line of customers ADT** is represented by a class **Queue**,

-A class Queue represents a linked list of nodes that represent queue of customers in a bank

**Data-Members – Private:**

-**Node** represented a customer queued to get serviced. A node contains **Enqueue\_T** - The time customer walks-in is stored here.

**Enqueue\_T -** The time customer gets dequeued is stored here.

**Next -** Stores the Pointer to the next node

**Item Wait\_Period –** Stores the number of ticks a node is in queue.

-**Node\* Head** and **Node\* Tail** pointers are pointers to the head and tail of the queue

**-int Count -** keeps track of queue length)

**-double Avg\_Wait –** It stores the average wait time of the whole queue. It is done everytime an item is enqueued we add it to the average and/2.

**-int H\_wait –** The highest wait time in the queue

**-int L\_Wait –** The lowest wait time in the Queue

**Class Functions – Public:**

**-Queue(): ~Queue()** ://CONSTRUCTOR && DESTRUCTOR

-**void Enqueue(Item entry)** :This function takes an entry from the program as a parameter and puts it in a node and attaches the node to the tail end of the queue

**-Item Dequeue** **(Item Deque\_T)** :This function returns (node-> **Wait\_Period** ) for the current head node and reduces queue size by 1. In the dequeue function we also calculate the wait time of the node, and the average wait time experienced by the nodes/customer that have been dequeued so far.

**-int Size() const** :Returns the size of the queue

**-void High\_Low\_Set(int Wait\_Period)** : We send the wait period of every node we dequeue and this function keeps track of Lowest and Highest wait-periods.

**-bool Is\_Empty() const** :Reports true of queue is empty

**-double Get\_Avg\_Wait() const** :Returns the average wait period for a queue

**-Node\* Get\_Node() const** :Used to get a Node pointer for use during Enqueueing of an entry, returns a pointer to a node.

**The Main program**

The Client program has three supporting functions that are used in simulation.

**- int Shortest(Queue Lines[], int Num\_Queues)**

This function is passed, an Array of queues and their size as a parameter and traverses the array to find the smallest queue size and returns its index in the array

**-int Queue\_Total(Queue Lines[], int Num\_Queues**

The function above will find the total number of nodes by adding up all the queue sizes and return the sum.

**-void output(Queue Queue1, Item DequeT);**

This function is called to output our Queue after simulation finishes.

Step 1. Get user input for the parameters of simulation.

Step 2. Seed the random number generator with start number

Step 3. Starting from time=0 to time==Simulation\_Duration

-For every tick of the clock we do as follows

3.a If a random probability 1-100 is below user-defined probability We enqueue/customer comes in to the queue. We use a function to select the shortest one.

3.b for all the tellers- if queue !Empty() dequeue(current time) and set their availability to unavailable.

3.c After the first iteration (Elapsed\_Time >0) the loop will start to do the following before steps 3.a and 3.b

*-> will tick the clocks on all the Tellers/Servers to synchronize with our simulation time.*

*->Will check if the transaction time on any of the servers/tellers has become 0, if true ->change availability to available*

### User Document The way to run the program

The program has four files, namely

Main.cpp

Implementation.cpp

Queue.h

Server.h

Using g++ they can be run by calling the two cpp files together:

G++ Main.cpp Implementation.cpp

Or using Microsoft visual studio to run the console application.

The user is expected to enter the simulation parameters to run the simulation.

The number of queue/server pairs:

The probability that a customer arrives in one tick (%):

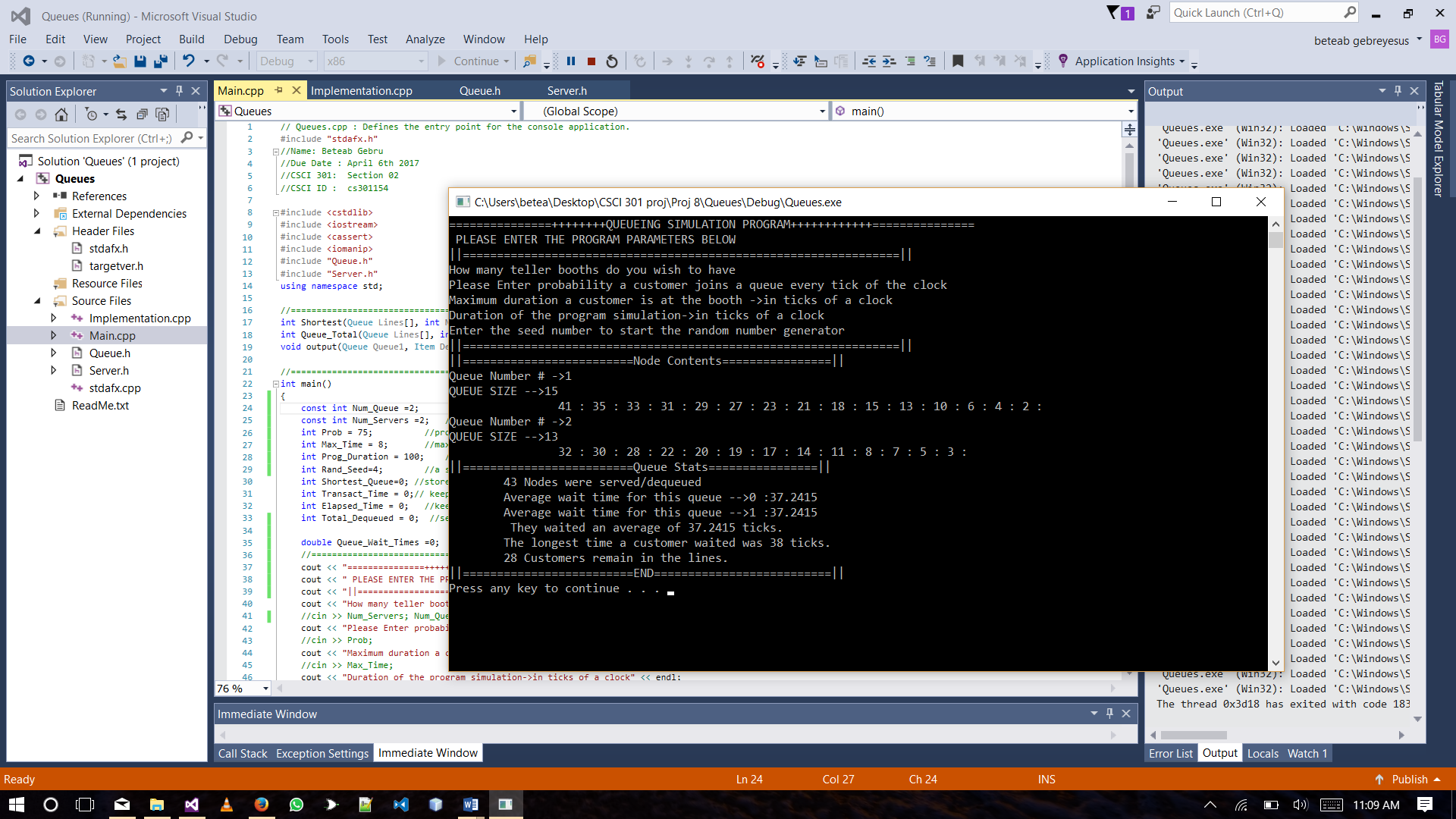
The maximum duration of a transaction in ticks:

The duration of the simulation in ticks:

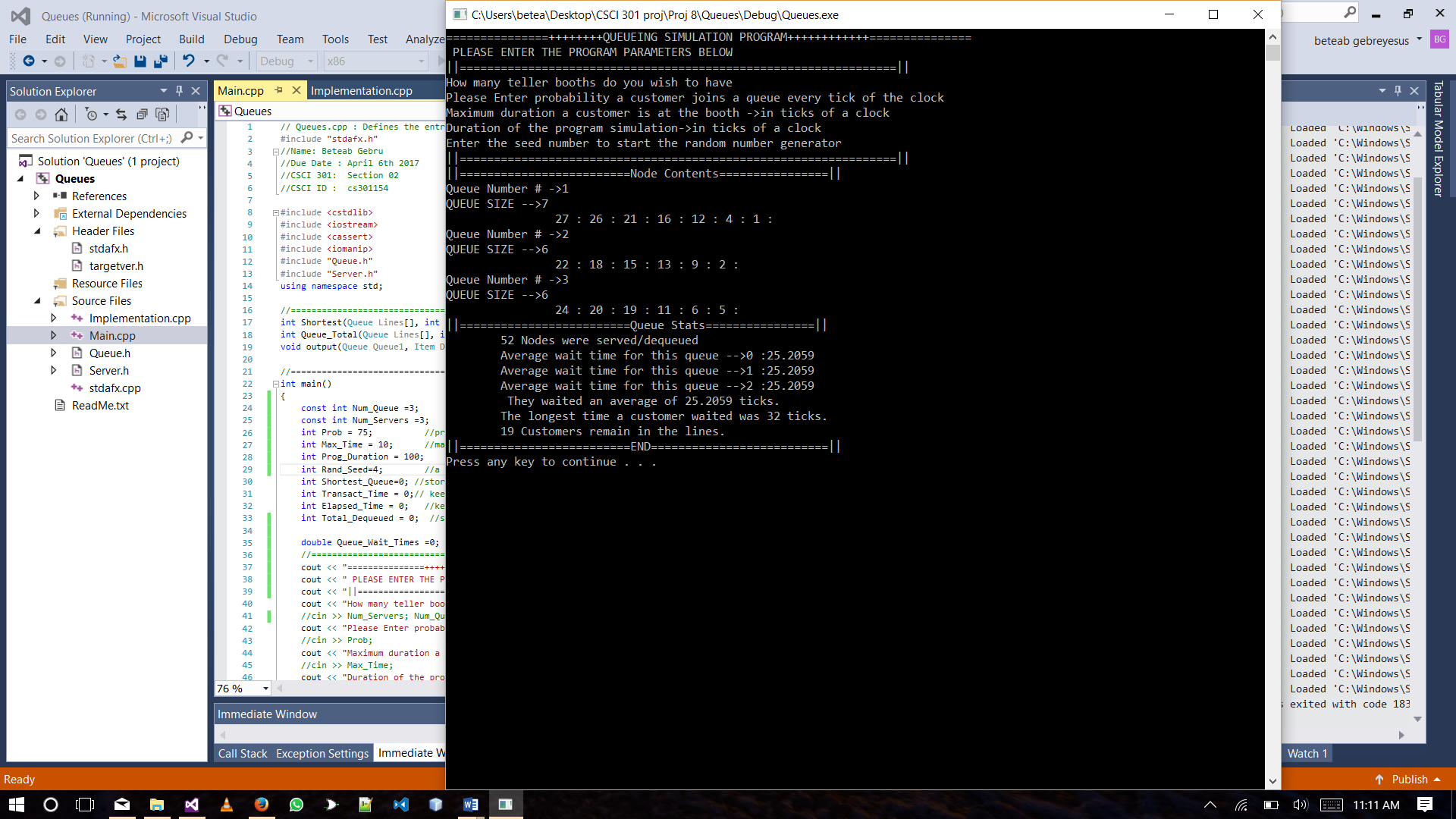
Enter a random number seed:

**Test Runs of the program**

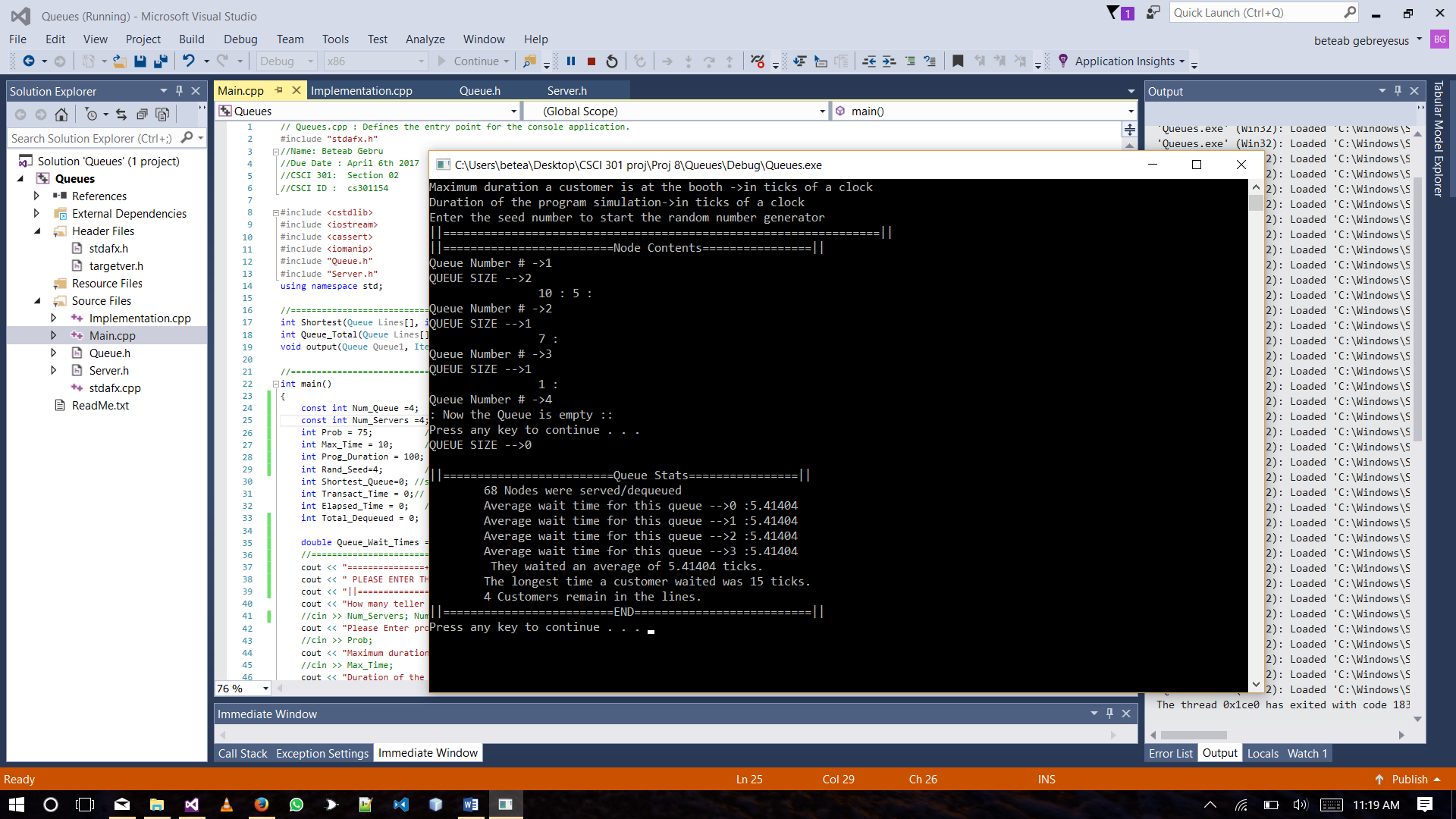
|  |  |
| --- | --- |
| Parameters | |
| **queue/server pairs:** | 2 |
| **Prob/one tick arrival** | 70% |
| **Max\_Trans\_Time** | 8 |
| **Simulation\_Duration** | 100 |
| **Rand(seed)** | 4 |



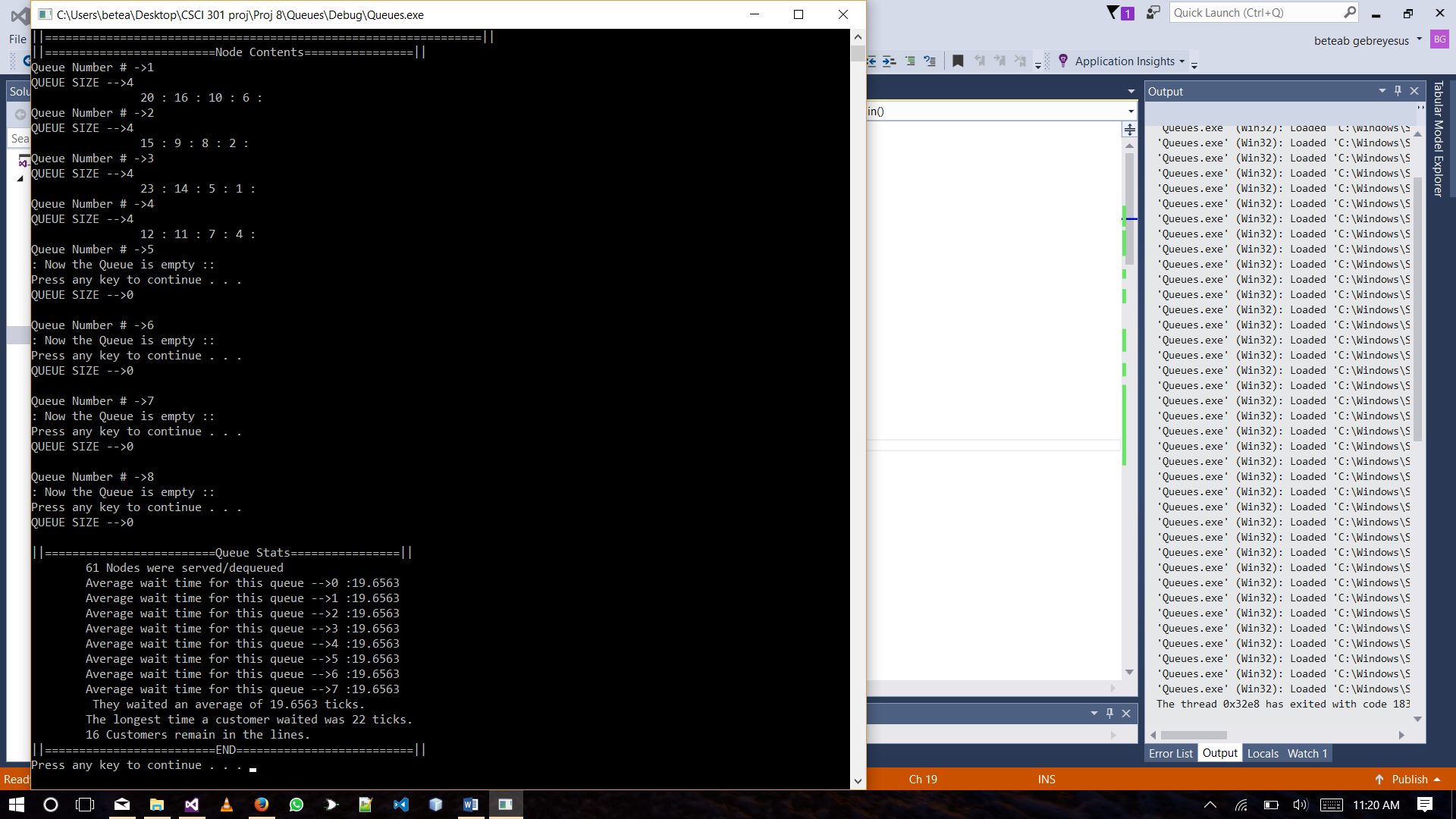
|  |  |  |
| --- | --- | --- |
| 2 | **queue/server pairs:** | 3 |
| **Prob/one tick arrival** | 75% |
| **Max\_Trans\_Time** | 10 |
| **Simulation\_Duration** | 100 |
| **Rand(seed)** | 4 |



|  |  |  |
| --- | --- | --- |
| 3 | **queue/server pairs:** | 4 |
| **Prob/one tick arrival** | 75% |
| **Max\_Trans\_Time** | 10 |
| **Simulation\_Duration** | 100 |
| **Rand(seed)** | 4 |



|  |  |  |
| --- | --- | --- |
| 3 | **queue/server pairs:** | 8 |
| **Prob/one tick arrival** | 75% |
| **Max\_Trans\_Time** | 12 |
| **Simulation\_Duration** | 100 |
| **Rand(seed)** | 4 |



### Summary

The program simulates a queue quite well. The interesting thing however is that the rand() number generator is not as random as it looks because the queue ends up becoming more or less even. In the very many runs of the simulation there was not a case where the difference between the array of queues’ sizes did not exceed two which indicates the number produces numbers with uniformity.

**Conclusion**

I run the program many times to check how many server we need to keep the average wait time to lower than 10 ticks. I discovered that for One tick customer walking prob of 75%,Max\_T of 8 ticks, it takes servers to keep the wait time below 10.