3/11/2025

Data Structures and Algorithms Project Documentation

Project Title: Queue Simulator

Debendra Adhikari, Subhash Shrestha &
Abhinaya Gyawali

Contents

Data Structures and Algorithms Project Documentation

Project Title: Queue Simulator

Purpose
The purpose of this project is to implement a queue simulator using the C++ programming language. The simulator models a queue system, such as a checkout queue in a store or a service request queue on a computer server. It demonstrates the use of STL data structures, file I/O operations, and basic statistical analysis.

Project Overview
The Queue Simulator is designed to simulate a queue system with customers arriving at random intervals and being served by one or more servers. The arrival and service times follow an exponential distribution. The project uses STL data structures like std::queue and std::vector, performs file I/O operations to save results, and calculates statistical metrics such as mean, median, and mode of service times. Additionally, it generates a graphical representation of customer service times using GNUplot.

Functional Requirements

1. **Simulate a Queue System:**
    a. The simulator models a queue system with customers arriving at random intervals and being served by one or more servers.
    b. The arrival and service times follow an exponential distribution.
2. **Use of STL Data Structures:**
    a. The project uses the following STL data structures:

i. std::queue to manage the customer queue.
ii. std::vector to store served customers for analysis.

3. **File I/O Operations:**
    a. The simulator saves the simulation results to a CSV file for further analysis.
    b. It generates a data file for plotting a graph using GNUplot.
4. **Statistical Analysis:**
    a. The simulator calculates and displays the mean, median, and mode of the service times.
5. **Graphical Output:**
    a. The simulator generates a graph of customer service times using GNUplot.

Implementation Details
Key Components

1. **Customer Structure:**
   Defined in Customer.h, the Customer structure represents a customer with the following attributes:
    a. id: Unique identifier for the customer.
    b. arrivalTime: Time at which the customer arrives in the queue.
    c. serviceTime: Time taken to serve the customer.
    d. departureTime: Time at which the customer leaves the system.

   The toCsv() function converts customer data to a CSV string.

   struct Customer { int id; double arrivalTime; double serviceTime; double departureTime;
   std::string toCsv() const {
   return std::to_string(id) + "," + std::to_string(arrivalTime) + "," +
   std::to_string(serviceTime) + "," + std::to_string(departureTime);
   }

   };

2. **Simulation Logic:**
   The runSimulation() function implements the core logic of the queue simulator:
   a. Customers arrive at random intervals following an exponential distribution.
   b. Customers are served by a fixed number of servers.
   c. The simulation runs for a predefined time (SIMULATION_TIME).

```
void runSimulation(double arrivalRate, double serviceRate, int numServers)
{
   queue<Customer> customerQueue;
   vector<Customer> servedCustomers;
   vector<double> serverAvailableTimes(numServers, 0.0);

   mt19937 gen(random_device{}());
   int customerId = 1;
   double currentTime = 0.0;

   while (currentTime < SIMULATION_TIME) {
      double arrivalTime = getExponentialRandom(arrivalRate, gen);
      currentTime += arrivalTime;
```

```
        Customer newCustomer = {customerId++, currentTime,
getExponentialRandom(serviceRate, gen), 0.0};
        customerQueue.push(newCustomer);

        for (int i = 0; i < numServers; ++i) {
            if (!customerQueue.empty() && serverAvailableTimes[i] <=
currentTime) {
                Customer customer = customerQueue.front();
                customerQueue.pop();

                customer.departureTime = currentTime + customer.serviceTime;
                serverAvailableTimes[i] = customer.departureTime;
                servedCustomers.push_back(customer);
            }
        }
    }
}
```

3. **Statistical Calculations:**
   The simulator calculates the following statistics:
   a. **Mean:** Average service time.
   b. **Median:** Middle value of service times.
   c. **Mode:** Most frequent service time

```
double calculateMean(const vector& customers) { double sum = 0.0; for
(const auto& customer : customers) { sum += customer.serviceTime; }
return sum / customers.size(); }
double calculateMedian(vector customers) { sort(customers.begin(),
customers.end(), [](const Customer& a, const Customer& b) { return
a.serviceTime < b.serviceTime; });
```

```
size_t size = customers.size();
if (size % 2 == 0) {
   return (customers[size / 2 - 1].serviceTime + customers[size /
2].serviceTime) / 2.0;
} else {
   return customers[size / 2].serviceTime;
}

}
double calculateMode(const vector& customers) { map<double, int>
frequencyMap; for (const auto& customer : customers)
{ frequencyMap[customer.serviceTime]++; }
double mode = 0.0;
int maxFrequency = 0;
for (const auto& pair : frequencyMap) {
   if (pair.second > maxFrequency) {
      mode = pair.first;
      maxFrequency = pair.second;
   }
}
return mode;

}
```

4. **File I/O:**
   a. Results are saved to a CSV file (simulation_results.csv).
   b. Service times are written to a data file (service_times.dat) for generating a graph using GNUplot.

```
void saveResultsToCsv(const vector<Customer>& customers, double
mean, double median, double mode, const string& filename) {
   ofstream file(filename);
   if (!file.is_open()) {
      cerr << "Error opening file: " << filename << endl;
```

```
        return;
    }

    file << "ID,ArrivalTime,ServiceTime,DepartureTime\n";
    for (const auto& customer : customers) {
        file << customer.toCsv() << "\n";
    }

    file << "\nStatistics:\n";
    file << "Mean Service Time: " << mean << "\n";
    file << "Median Service Time: " << median << "\n";
    file << "Mode Service Time: " << mode << "\n";

    file.close();
}

void writeServiceTimesToFile(const vector<Customer>& customers,
const string& filename) {
    ofstream file(filename);
    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return;
    }

    for (const auto& customer : customers) {
        file << customer.serviceTime << "\n";
    }

    file.close();
}
```

5. **Graphical Output:**
   The simulator uses GNUplot to generate a graph of customer service times.

   void generateGraphWithGNUplot(const string& dataFilename) { string command = "gnuplot -e "set terminal png; set output 'service_times_graph.png'; plot '" + dataFilename + "' with linespoints""; system(command.c_str()); }

User Manual

How to Run the Simulator

1. **Compile the Code:**
   Ensure you have a C++ compiler installed (e.g., g++).
   Compile the code using the following command:
   g++ -o queue_simulator main.cpp
2. **Run the Executable:**
   Execute the compiled program:
   ./queue_simulator
3. **Input Parameters:**
   When prompted, enter the following parameters:
   a. **Arrival Rate:** Average number of customers arriving per unit time (e.g., 2).
   b. **Service Rate:** Average number of customers served per unit time (e.g., 3).
   c. **Number of Servers:** Number of available servers (e.g., 4).
4. **View Results:**
   The simulator will display the results in a table format.
   It will generate the following files:
   a. simulation_results.csv: Contains detailed simulation results.
   b. service_times.dat: Contains service times for plotting.
   c. service_times_graph.png: Graph of customer service times.

Terminal Output

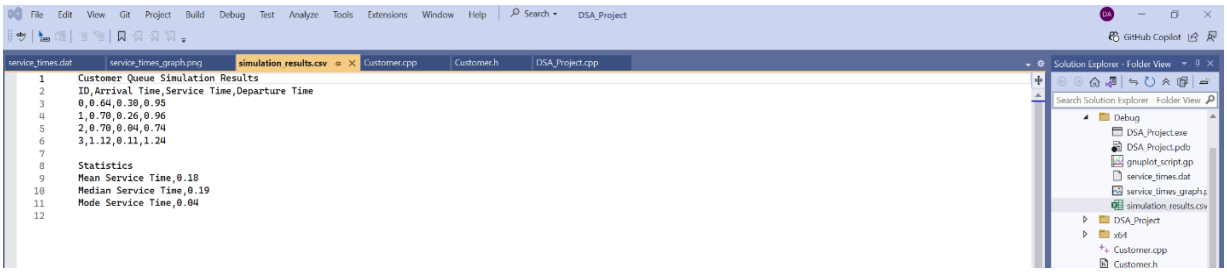
When you run the program, the terminal will display the following output:

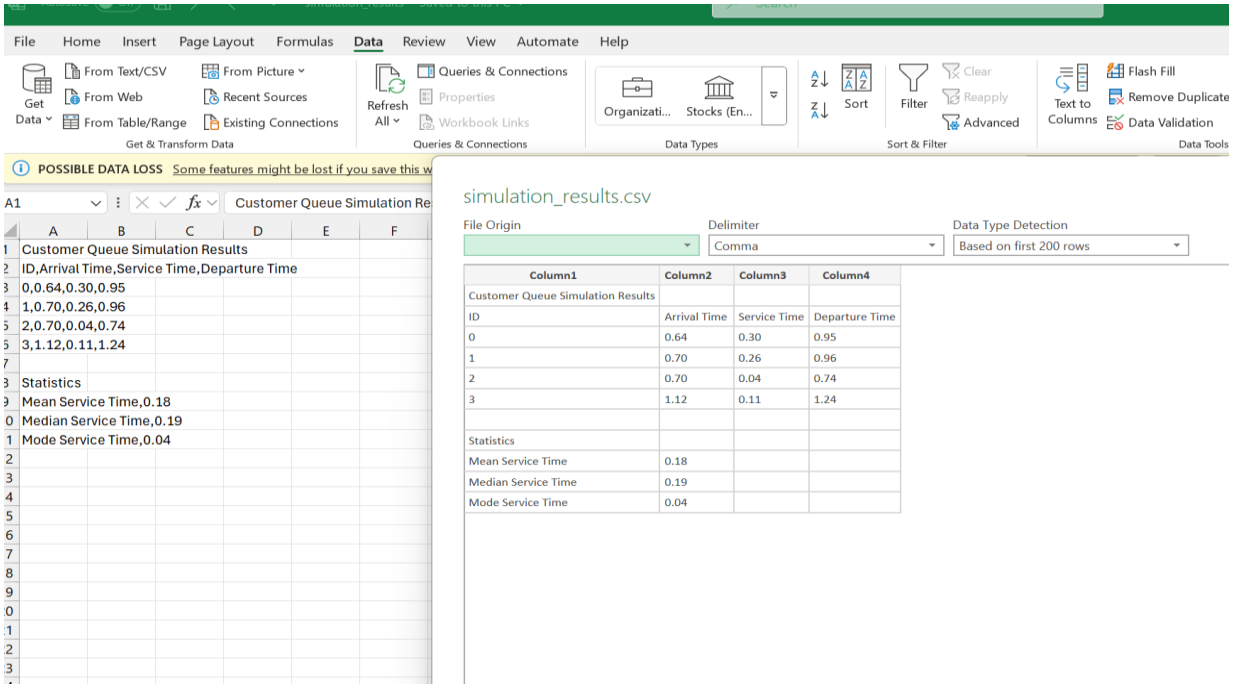
```
Enter arrival rate (customers per unit time): 2
Enter service rate (services per unit time): 3
Enter number of servers: 4
```

```
Enter arrival rate (customers per unit time): 2
Enter service rate (services per unit time): 3
Enter number of servers: 4

Simulation Results:
+-----+--------------+--------------+-----------------+
| ID  | Arrival Time | Service Time | Departure Time  |
+-----+--------------+--------------+-----------------+
|   0 |         0.64 |         0.30 |            0.95 |
|   1 |         0.70 |         0.26 |            0.96 |
|   2 |         0.70 |         0.04 |            0.74 |
|   3 |         1.12 |         0.11 |            1.24 |
+-----+--------------+--------------+-----------------+

+--------------------+
|     Statistics     |
+--------------------+
| Mean Service Time:  |       0.18 |
| Median Service Time:|       0.19 |
| Mode Service Time:  |       0.04 |
+--------------------+
Service times written to service_times.dat
Graph generated as 'service_times_graph.png'
Results saved to simulation_results.csv
Simulation complete. Results saved to file.
Press any key to continue . . .
```

Generated Output Files

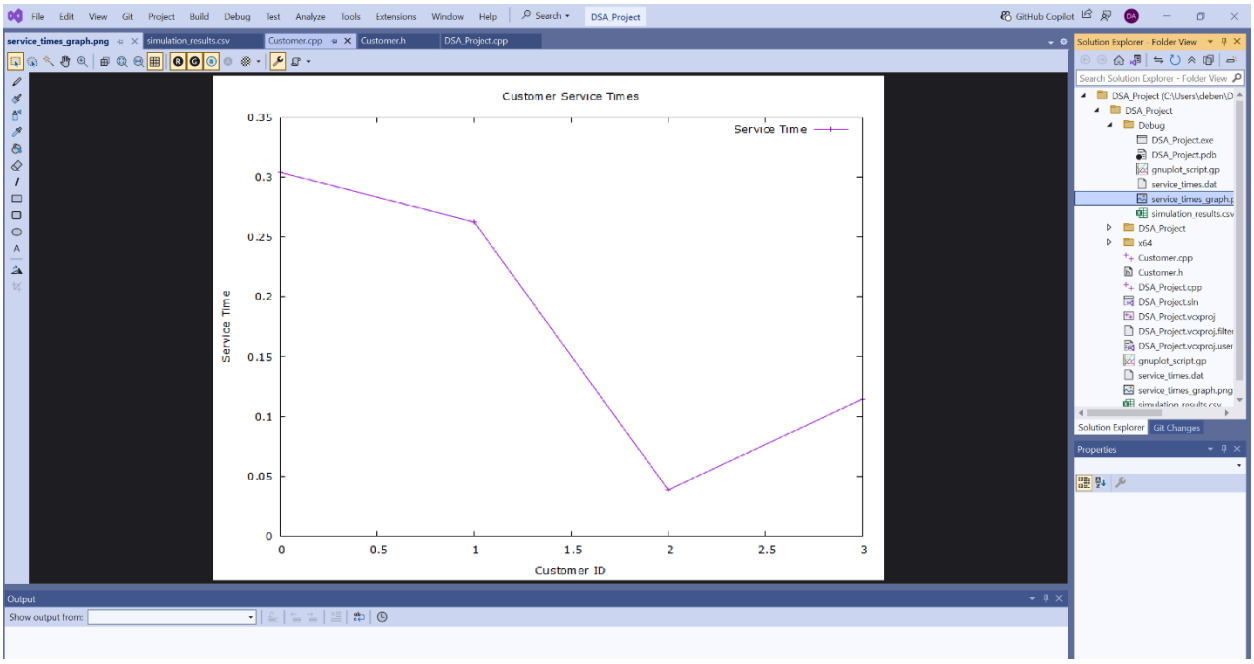
1. **simulation_results.csv:**
   This file contains the detailed simulation results in CSV format. Here's an example of its contents:





2. **service_times.dat:**
   This file contains the service times of all customers, which are used for plotting the graph. Here's an example of its contents:

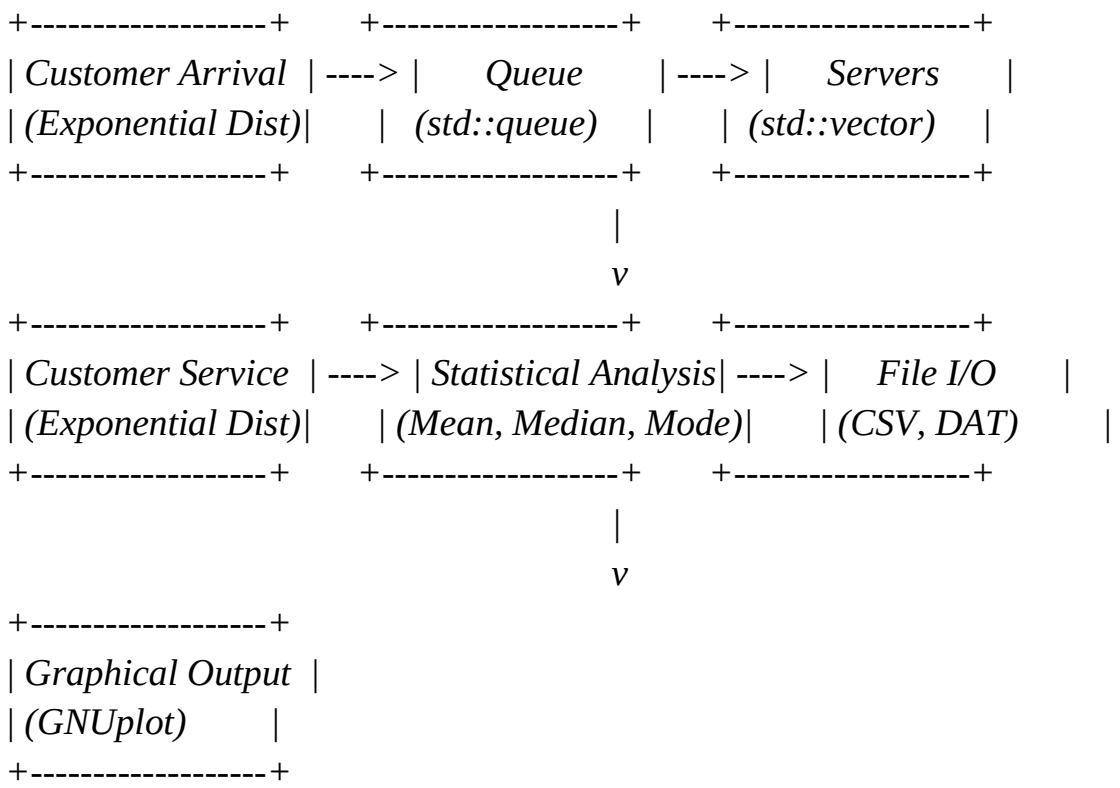
3. **service_times_graph.png:**
   This is the graph generated using GNUplot, which visualizes the service times of customers.



Conclusion

This project demonstrates the implementation of a queue simulator using C++. It leverages STL data structures, file I/O operations, and statistical analysis to model and analyze a queue system. The simulator is modular, efficient, and ready for further extension.

TEXT REPRESENTATION:



VISUAL REPRESENTATION:

[Start Simulation] --> [Customer Arrival] --> [Queue] --> [Servers] --> [Customer Service]

[Statistical Analysis] <-- [File I/O] <-- [Graphical Output] <-- [End Simulation]