```
/*======= CSCI203/803 ASSIGNMENT-2 MARKING (out of 10 Marks) ============

========================== YOUR OUTPUT ====================================

Number of customers served: 494
Time taken to serve all customers: 276.886
Greatest length reached by the customer queue: 33
Average length of the customer queue: 12.4289
Average customer waiting time in queue: 6.96637
Percentage of customers with zero waiting time: 1.417%

Server      Efficiency  Customers   Idle Time
0           1.4         77          1.78
1           0.5         177         4.421
2           1.1         82          5.447
3           1.1         89          6.378
4           1.5         69          3.776

======== MARKING & FEEDBACK ON YOUR OUTPUT (2 marks)
====================================

Your output is correct.

========= MARKING & FEEDBACK ON YOUR REPORT (2 marks)
====================================
// Note: to get full marks the report should list all the data structures used in
// your code and explain at least three optimisations done to improve the speed.

Your report is ok and consistent with the code.

========= MARKING & FEEDBACK ON YOUR CODE (6 marks)
====================================
// Note: to get full marks the code should be correct and have three optimisations.
e.g.:
// 1. Heaped event queue; 2. heaped idle server selection; 3. customer circular queue.

Correct implementation of the code.

---------------------------------------------------------------------
TOTAL MARKS FOR ASSIGNMENT 2 STEPS 2 to 5:  10 MARKS (OUT OF 10)
---------------------------------------------------------------------

============================ YOUR CODE ========================================*/

/*
Assignment 2
Ben Malen
bm365

This program models the operation of a proposed supermarket by using Discrete Event
Simulation.
*/

#include <iostream>
#include <fstream>
#include <string>
#include <cstring> // strcmp
#include <iomanip> // setw
using namespace std;

const char FILE_NAME[9] = "ass2.txt";
const unsigned int MAX_SERVERS = 20;
const unsigned int MAX_CUSTOMERS = 500;
const unsigned int MAX_EVENTS = 100;

/*-----------------------------------------------------------------------------*/
// Class declarations

class Customer {
    public:
        Customer(double arrivalTime, double tallyTime, bool cashPayment) :
            arrivalTime_(arrivalTime),
```

```
68              tallyTime_(tallyTime),
69              cashPayment_(cashPayment) {}
70          double getArrivalTime() { return arrivalTime_; }
71          double getTallyTime() { return tallyTime_; }
72          bool isCash() { return cashPayment_; }
73      private:
74          double arrivalTime_, tallyTime_;
75          bool cashPayment_; // true for cash, false for card
76  };
77
78  class Event {
79      public:
80          // CustomerArrival constructor
81          Event(double eventTime, double tallyTime, bool cashPayment) :
82              eventTime_(eventTime),
83              tallyTime_(tallyTime),
84              cashPayment_(cashPayment) {
85              type_ = -1;
86          }
87          // ServerFinish constructor
88          Event(int type, double eventTime) :
89              type_(type), // indicates index of server
90              eventTime_(eventTime) {
91              tallyTime_ = 0; // unused
92              cashPayment_ = false; // unused
93          }
94
95          Event(const Event &e);
96          Event & operator=(const Event &e);
97          friend bool operator<(const Event &lhs, const Event &rhs);
98          int getType() { return type_; }
99          double getEventTime() { return eventTime_; }
100         double getTallyTime() { return tallyTime_; }
101         bool isCash() { return cashPayment_; }
102     private:
103         int type_; // 1 = CustomerArrival, 2 = ServerFinish (indicates the index of
            the server)
104         double eventTime_, tallyTime_;
105         bool cashPayment_; // true for cash, false for card
106 };
107
108 inline bool operator<(const Event &lhs, const Event &rhs) { return lhs.eventTime_ <
    rhs.eventTime_; }
109
110 class Server {
111     public:
112         Server(unsigned int index, double efficiency) : index_(index),
            efficiency_(efficiency) {
113             nCustomersServed = 0;
114             serviceTime_ = 0;
115         }
116         friend bool operator<(const Server &lhs, const Server &rhs);
117         double getEfficiency() { return efficiency_; }
118         unsigned int getIndex() { return index_; }
119         unsigned int getCustomerCount() { return nCustomersServed; }
120         double getServiceTime() { return serviceTime_; }
121         void addCustomerServed() { ++nCustomersServed; }
122         void addServiceTime(double units) { serviceTime_ += units; }
123     private:
124         unsigned int index_, nCustomersServed;
125         double efficiency_, serviceTime_;
126 };
127
128 inline bool operator<(const Server &lhs, const Server &rhs) { return lhs.efficiency_
    < rhs.efficiency_; }
129
130 class Servers {
131     public:
132         Servers() { nServers_ = 0; }
133         void deleteServers();
134         bool isFull() { return nServers_ == MAX_SERVERS; }
135         bool isEmpty() { return nServers_ == 0; }
```

```
136         void printStats(double totalTime);
137         void addServer(Server *server);
138         Server * getServer(unsigned int index) { return servers_[index]; }
139     private:
140         Server * servers_[MAX_SERVERS];
141         unsigned int nServers_;
142 };
143
144 class IdleServers {
145     public:
146         IdleServers() { nIdleServers_ = 0; }
147         bool isFull() { return nIdleServers_ == MAX_SERVERS; }
148         bool isEmpty() { return nIdleServers_ == 0; }
149         void enqueue(Server *servers);
150         Server * dequeue();
151         void swapServer(Server *&a, Server *&b);
152         void siftUp(unsigned int i);
153         void siftDown(unsigned int i);
154     private:
155         Server * idleServers_[MAX_SERVERS];
156         unsigned int nIdleServers_;
157 };
158
159 class EventQueue {
160     public:
161         EventQueue() { nEvents_ = 0; }
162         bool isFull() { return nEvents_ == MAX_EVENTS; }
163         bool isEmpty() { return nEvents_ == 0; }
164         void enqueue(Event *event);
165         Event * dequeue();
166         void swapEvent(Event *&a, Event *&b);
167         void siftUp(unsigned int i);
168         void siftDown(unsigned int i);
169     private:
170         Event *events_[MAX_EVENTS];
171         unsigned int nEvents_;
172 };
173
174 class CustomerQueue {
175     public:
176         CustomerQueue() {
177             nCustomers_ = 0;
178             front_ = 0;
179             rear_ = MAX_CUSTOMERS - 1;
180             greatestLength_ = 0;
181             totalTimeInQueue_ = 0;
182         }
183         bool isFull() { return nCustomers_ == MAX_CUSTOMERS; }
184         bool isEmpty() { return nCustomers_ == 0; }
185         void enqueue(Customer *c);
186         Customer * dequeue(double currentTime);
187         double getAverageLength(double totalTime) { return totalTimeInQueue_ /
        totalTime; }
188         double getAverageTime(unsigned int nCustomersServed) { return
        totalTimeInQueue_ / nCustomersServed; }
189         unsigned int getGreatestLength() { return greatestLength_; }
190     private:
191         Customer *customers_[MAX_CUSTOMERS];
192         unsigned int nCustomers_, front_, rear_, greatestLength_;
193         double totalTimeInQueue_;
194 };
195
196 /*--------------------------------------------------------------------------*/
197
198 // Driver
199 int main() {
200     ifstream fin;
201     fin.open(FILE_NAME);
202     if (!fin) {
203         cerr << "Could not open " << FILE_NAME << endl;
204         return 1;
205     }
```

```
206         CustomerQueue customerQueue;
207         EventQueue eventQueue;
208         Servers servers;
209         IdleServers idleServers;
210         unsigned int nServers,
211                      nCustomersServed = 0,
212                      noWaitCount = 0;
213         double efficiency,
214                arrivalTime,
215                tallyTime,
216                currentTime = 0,
217                firstArrivalTime;
218         char paymentType[5];
219         fin >> nServers;
220         for (unsigned int i = 0; i < nServers; ++i) {
221             fin >> efficiency;
222             Server *server = new Server(i, efficiency);
223             servers.addServer(server);
224             idleServers.enqueue(server);
225         }
226         // Read first CustomerArrival event from file and add it to the event queue
227         fin >> arrivalTime >> tallyTime >> paymentType;
228         eventQueue.enqueue(new Event(arrivalTime, tallyTime, (strcmp(paymentType,
            "cash") == 0) ? true : false)); // Enqueue CustomerArrival
229         firstArrivalTime = arrivalTime;
230         while (!eventQueue.isEmpty()) {
231             Event *event = eventQueue.dequeue(); // Dequeue the event
232             currentTime = event->getEventTime();
233             if (event->getType() == -1) {
234                 // Event is CustomerArrival
235                 Server *server = idleServers.dequeue(); // Attempt to dequeue an idle
                    server
236                 if (server != NULL) {
237                     // Idle server available
238                     double serviceTime = event->getTallyTime() * server->getEfficiency()
                        + (event->isCash() ? 0.3 : 0.7);
239                     double finishTime = currentTime + serviceTime;
240                     server->addServiceTime(serviceTime);
241                     eventQueue.enqueue(new Event(server->getIndex(), finishTime)); //
                        Enqueue ServerFinish
242                     ++noWaitCount;
243                 }
244                 else {
245                     // No idle server available
246                     customerQueue.enqueue(new Customer(currentTime,
                        event->getTallyTime(), event->isCash())); // Enqueue Customer
247                 }
248                 // Read in the next customer
249                 if (fin >> arrivalTime >> tallyTime >> paymentType)
250                     eventQueue.enqueue(new Event(arrivalTime, tallyTime,
                        (strcmp(paymentType, "cash") == 0) ? true : false)); // Enqueue
                        CustomerArrival
251             }
252             else {
253                 // Event is ServerFinish
254                 ++nCustomersServed;
255                 servers.getServer(event->getType())->addCustomerServed();
256                 idleServers.enqueue(servers.getServer(event->getType())); // Enqueue
                    idle server
257                 if (!customerQueue.isEmpty()) {
258                     Customer *customer = customerQueue.dequeue(currentTime); // Dequeue
                        Customer
259                     Server *server = idleServers.dequeue(); // Dequeue idle server
260                     double serviceTime = customer->getTallyTime() *
                        server->getEfficiency() + (customer->isCash() ? 0.3 : 0.7);
261                     double finishTime = currentTime + serviceTime;
262                     server->addServiceTime(serviceTime);
263                     eventQueue.enqueue(new Event(server->getIndex(), finishTime)); //
                        Enqueue ServerFinish
264                     delete customer;
265                 }
266             }
```

```
267             delete event;
268         }
269         fin.close();
270         double totalTime = currentTime - firstArrivalTime;
271         cout << "Number of customers served: " << nCustomersServed
272              << "\nTime taken to serve all customers: " << totalTime
273              << "\nGreatest length reached by the customer queue: " <<
                 customerQueue.getGreatestLength()
274              << "\nAverage length of the customer queue: " <<
                 customerQueue.getAverageLength(totalTime)
275              << "\nAverage customer waiting time in queue: " <<
                 customerQueue.getAverageTime(nCustomersServed)
276              << "\nPercentage of customers with zero waiting time: " <<
                 ((double)noWaitCount / nCustomersServed * 100) << "%"
277              << "\n"
278              << endl;
279         servers.printStats(totalTime);
280         servers.deleteServers(); // free dynamic memory
281         return 0;
282     }
283
284     /*-------------------------------------------------------------------------
285     Customers waiting to be served are placed into a FIFO queue, implemented using
286     modular arithmetic (circular), which keeps track of the rear and front of the queue.
287     */
288
289     // A new customer is placed into the rear of the queue.
290     void CustomerQueue::enqueue(Customer *customer) {
291         if (isFull()) {
292             cerr << "MAX_CUSTOMERS exceeded." << endl;
293             exit(1);
294         }
295         rear_ = (rear_ + 1) % MAX_CUSTOMERS; // modular arithmetic so rear_ "wraps
            around" to zero when it reaches MAX_CUSTOMERS
296         customers_[rear_] = customer;
297         ++nCustomers_;
298         if (nCustomers_ > greatestLength_)
299             greatestLength_ = nCustomers_;
300     }
301
302     // The customer in the front of the queue is removed.
303     Customer * CustomerQueue::dequeue(double currentTime) {
304         if (isEmpty()) {
305             cerr << "CustomerQueue is empty." << endl;
306             exit(1);
307         }
308         Customer *customer = customers_[front_];
309         double timeInQueue = currentTime - customer->getArrivalTime();
310         totalTimeInQueue_ += timeInQueue;
311         front_ = (front_ + 1) % MAX_CUSTOMERS; // modular arithmetic so front_ "wraps
            around" to zero when it reaches MAX_CUSTOMERS
312         --nCustomers_;
313         return customer;
314     }
315
316     /*-------------------------------------------------------------------------
317     CustomerArrival and ServerFinish events are placed into a priority queue,
318     implemented using a min-heap.
319     */
320
321     // Inserts a new element into the heap.
322     // The new element is placed at the end of the heap and siftUp moves it up into the
        correct position.
323     void EventQueue::enqueue(Event *event) {
324         if (isFull()) {
325             cerr << "MAX_EVENTS exceeded." << endl;
326             exit(1);
327         }
328         events_[nEvents_++] = event;
329         siftUp(nEvents_ - 1);
330     }
331
```

```
332 // Removes the top element in the heap.
333 // The top element is replaced with the last element in the heap,
334 // and siftDown then moves that element back to the bottom of the heap.
335 Event * EventQueue::dequeue() {
336     if (isEmpty()) {
337         cerr << "EventQueue is empty." << endl;
338         exit(1);
339     }
340     Event *event = events_[0];
341     events_[0] = events_[nEvents_ - 1];
342     --nEvents_;
343     siftDown(0);
344     return event;
345 }
346
347 // Swap the addresses that the pointers are pointing to using reference-to-pointer.
348 void EventQueue::swapEvent(Event *&a, Event *&b) {
349     Event *temp = a;
350     a = b;
351     b = temp;
352 }
353
354 // Min heap
355 // Moves element up to its correct position.
356 void EventQueue::siftUp(unsigned int i) {
357     if (i == 0) // then the element is the root
358         return;
359     unsigned int p = (i - 1) / 2; // integer division to find the parent
360     if (*events_[p] < *events_[i]) // parent is smaller, so we will leave it as is
361         return;
362     else {
363         swapEvent(events_[i], events_[p]); // put smallest in parent
364         siftUp(p); // and siftUp parent
365     }
366 }
367
368 // Min heap
369 // Moves element down to its correct position.
370 void EventQueue::siftDown(unsigned int i) {
371     unsigned int left = i * 2 + 1; // index of the left child
372     if (left >= nEvents_)
373         return; // left child does not exist
374     unsigned int smallest = left;
375     unsigned int right = left + 1; // index of the right child
376     if (right < nEvents_) // right child exists
377         if (*events_[right] < *events_[smallest]) // right child is smallest child
378             smallest = right;
379     if (*events_[smallest] < *events_[i]) {
380         swapEvent(events_[i], events_[smallest]);
381         siftDown(smallest);
382     }
383 }
384
385 /*-----------------------------------------------------------------------------*/
386
387 // Prints the statistics for each server.
388 void Servers::printStats(double totalTime) {
389     cout << left
390          << setw(12) << "Server"
391          << setw(12) << "Efficiency"
392          << setw(12) << "Customers"
393          << setw(12) << "Idle Time" << endl;
394     for (unsigned int i = 0; i < nServers_; ++i) {
395         cout << left
396              << setw(12) << i
397              << setw(12) << servers_[i]->getEfficiency()
398              << setw(12) << servers_[i]->getCustomerCount()
399              << setw(12) << totalTime - servers_[i]->getServiceTime() << endl;
400     }
401 }
402
403 // Inserts a server into the server array.
```

```
404  void Servers::addServer(Server *server) {
405      if (isFull()) {
406          cerr << "MAX_SERVERS exceeded." << endl;
407          exit(1);
408      }
409      servers_[nServers_++] = server;
410  }
411
412  // Frees dynamic memory.
413  void Servers::deleteServers() {
414      while (!isEmpty()) {
415          delete servers_[nServers_ - 1];
416          --nServers_;
417      }
418  }
419
420  /*-----------------------------------------------------------------------------
421  Idle servers are placed into a priority queue, implemented using a min-heap.
422  */
423
424  // Inserts a new element into the heap.
425  // The new element is placed at the end of the heap and siftUp moves it up into the
     correct position.
426  void IdleServers::enqueue(Server *server) {
427      if (isFull()) {
428          cerr << "MAX_SERVERS exceeded." << endl;
429          exit(1);
430      }
431      idleServers_[nIdleServers_++] = server;
432      siftUp(nIdleServers_ - 1);
433  }
434
435  // Removes the top element in the heap (fastest idle server, or NULL if there are no
     idle servers).
436  // The top element is replaced with the last element in the heap,
437  // and siftDown then moves that element back to the bottom of the heap.
438  Server * IdleServers::dequeue() {
439      if (isEmpty())
440          return NULL;
441      Server *server = idleServers_[0];
442      idleServers_[0] = idleServers_[nIdleServers_ - 1];
443      --nIdleServers_;
444      siftDown(0);
445      return server;
446  }
447
448  // Swap the addresses that the pointers are pointing to using reference-to-pointer.
449  void IdleServers::swapServer(Server *&a, Server *&b) {
450      Server *temp = a;
451      a = b;
452      b = temp;
453  }
454
455  // Min heap
456  // Moves element up to its correct position.
457  void IdleServers::siftUp(unsigned int i) {
458      if (i == 0) // then the element is the root
459          return;
460      unsigned int p = (i - 1) / 2; // integer division to find the parent
461      if (*idleServers_[p] < *idleServers_[i]) // parent is smaller, so we will leave
         it as is
462          return;
463      else {
464          swapServer(idleServers_[i], idleServers_[p]); // put smallest in parent
465          siftUp(p); // and siftUp parent
466      }
467  }
468
469  // Min heap
470  // Moves element down to its correct position.
471  void IdleServers::siftDown(unsigned int i) {
472      unsigned int left = i * 2 + 1; // index of the left child
```

```
473         if (left >= nIdleServers_)
474             return; // left child does not exist
475         unsigned int smallest = left;
476         unsigned int right = left + 1; // index of the right child
477         if (right < nIdleServers_) // right child exists
478             if (*idleServers_[right] < *idleServers_[smallest]) // right child is
                smallest child
479                 smallest = right;
480         if (*idleServers_[smallest] < *idleServers_[i]) {
481             swapServer(idleServers_[i], idleServers_[smallest]);
482             siftDown(smallest);
483         }
484     }
485
486     /*-----------------------------------------------------------------------------
487
488     The customer queue (FIFO queue) is implemented as a circular buffer using modular
489     arithmetic to keep track of the rear and front of the queue. Enqueue and dequeue
490     is fast because it never needs to be sorted.
491
492     The event queue (priority queue) is implemented using a min-heap, so the root
493     element always contains the event with the smallest event time. When adding an
494     event, it is placed at the end of the heap and siftUp moves it up into the
495     correct position. When removing an event, the top element is replaced with the
496     last element in the heap, and siftDown then moves that element back to the bottom
497     of the heap [see lecture slides, Week 3, "Priority queues"].
498
499     Similarly to events, idle servers are placed into a priority queue implemented
500     using a min-heap, so the root element always contains the server with the best
501     efficiency. Servers are removed from the heap if they are busy, and placed back
502     when they become free (idle).
503
504     The swap functions (as used by the siftUp and siftDown heap functions) swap the
505     memory addresses that the pointers are pointing to, opposed to entire objects.
506
507     -----------------------------------------------------------------------------*/
508
```