```
/*======= 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),
            tallyTime_(tallyTime),
            cashPayment_(cashPayment) {}
        double getArrivalTime() { return arrivalTime_; }
        double getTallyTime() { return tallyTime_; }
        bool isCash() { return cashPayment_; }
    private:
        double arrivalTime_, tallyTime_;
        bool cashPayment_; // true for cash, false for card
};

class Event {
    public:
        // CustomerArrival constructor
        Event(double eventTime, double tallyTime, bool cashPayment) :
            eventTime_(eventTime),
            tallyTime_(tallyTime),
            cashPayment_(cashPayment) {
            type_ = -1;
        }
        // ServerFinish constructor
        Event(int type, double eventTime) :
            type_(type), // indicates index of server
            eventTime_(eventTime) {
            tallyTime_ = 0; // unused
            cashPayment_ = false; // unused
        }

        Event(const Event &e);
        Event & operator=(const Event &e);
        friend bool operator<(const Event &lhs, const Event &rhs);
        int getType() { return type_; }
        double getEventTime() { return eventTime_; }
        double getTallyTime() { return tallyTime_; }
        bool isCash() { return cashPayment_; }
    private:
        int type_; // 1 = CustomerArrival, 2 = ServerFinish (indicates the index of the server)
        double eventTime_, tallyTime_;
        bool cashPayment_; // true for cash, false for card
};

inline bool operator<(const Event &lhs, const Event &rhs) { return lhs.eventTime_ < rhs.eventTime_; }

class Server {
    public:
        Server(unsigned int index, double efficiency) : index_(index), efficiency_(efficiency) {
            nCustomersServed = 0;
            serviceTime_ = 0;
        }
        friend bool operator<(const Server &lhs, const Server &rhs);
        double getEfficiency() { return efficiency_; }
        unsigned int getIndex() { return index_; }
        unsigned int getCustomerCount() { return nCustomersServed; }
        double getServiceTime() { return serviceTime_; }
        void addCustomerServed() { ++nCustomersServed; }
        void addServiceTime(double units) { serviceTime_ += units; }
    private:
        unsigned int index_, nCustomersServed;
        double efficiency_, serviceTime_;
};

inline bool operator<(const Server &lhs, const Server &rhs) { return lhs.efficiency_ < rhs.efficiency_; }

class Servers {
    public:
        Servers() { nServers_ = 0; }
        void deleteServers();
        bool isFull() { return nServers_ == MAX_SERVERS; }
        bool isEmpty() { return nServers_ == 0; }
        void printStats(double totalTime);
        void addServer(Server *server);
        Server * getServer(unsigned int index) { return servers_[index]; }
    private:
        Server * servers_[MAX_SERVERS];
        unsigned int nServers_;
};

class IdleServers {
    public:
        IdleServers() { nIdleServers_ = 0; }
        bool isFull() { return nIdleServers_ == MAX_SERVERS; }
        bool isEmpty() { return nIdleServers_ == 0; }
        void enqueue(Server *servers);
        Server * dequeue();
        void swapServer(Server *&a, Server *&b);
        void siftUp(unsigned int i);
        void siftDown(unsigned int i);
    private:
        Server * idleServers_[MAX_SERVERS];
        unsigned int nIdleServers_;
};

class EventQueue {
    public:
        EventQueue() { nEvents_ = 0; }
        bool isFull() { return nEvents_ == MAX_EVENTS; }
        bool isEmpty() { return nEvents_ == 0; }
        void enqueue(Event *event);
        Event * dequeue();
        void swapEvent(Event *&a, Event *&b);
        void siftUp(unsigned int i);
        void siftDown(unsigned int i);
    private:
        Event *events_[MAX_EVENTS];
        unsigned int nEvents_;
};

class CustomerQueue {
    public:
        CustomerQueue() {
            nCustomers_ = 0;
            front_ = 0;
            rear_ = MAX_CUSTOMERS - 1;
            greatestLength_ = 0;
            totalTimeInQueue_ = 0;
        }
        bool isFull() { return nCustomers_ == MAX_CUSTOMERS; }
        bool isEmpty() { return nCustomers_ == 0; }
        void enqueue(Customer *c);
        Customer * dequeue(double currentTime);
        double getAverageLength(double totalTime) { return totalTimeInQueue_ / totalTime; }
        double getAverageTime(unsigned int nCustomersServed) { return totalTimeInQueue_ / nCustomersServed; }
        unsigned int getGreatestLength() { return greatestLength_; }
    private:
        Customer *customers_[MAX_CUSTOMERS];
        unsigned int nCustomers_, front_, rear_, greatestLength_;
        double totalTimeInQueue_;
};

/*----------------------------------------------------------------------------*/

// Driver
int main() {
    ifstream fin;
    fin.open(FILE_NAME);
    if (!fin) {
        cerr << "Could not open " << FILE_NAME << endl;
        return 1;
    }
    CustomerQueue customerQueue;
    EventQueue eventQueue;
    Servers servers;
    IdleServers idleServers;
    unsigned int nServers,
             nCustomersServed = 0,
             noWaitCount = 0;
    double efficiency,
           arrivalTime,
           tallyTime,
           currentTime = 0,
           firstArrivalTime;
    char paymentType[5];
    fin >> nServers;
    for (unsigned int i = 0; i < nServers; ++i) {
        fin >> efficiency;
        Server *server = new Server(i, efficiency);
        servers.addServer(server);
        idleServers.enqueue(server);
    }
    // Read first CustomerArrival event from file and add it to the event queue
    fin >> arrivalTime >> tallyTime >> paymentType;
    eventQueue.enqueue(new Event(arrivalTime, tallyTime, (strcmp(paymentType, "cash") == 0) ? true : false)); // Enqueue CustomerArrival
    firstArrivalTime = arrivalTime;
    while (!eventQueue.isEmpty()) {
        Event *event = eventQueue.dequeue(); // Dequeue the event
        currentTime = event->getEventTime();
        if (event->getType() == -1) {
            // Event is CustomerArrival
            Server *server = idleServers.dequeue(); // Attempt to dequeue an idle server
            if (server != NULL) {
                // Idle server available
                double serviceTime = event->getTallyTime() * server->getEfficiency() + (event->isCash() ? 0.3 : 0.7);
                double finishTime = currentTime + serviceTime;
                server->addServiceTime(serviceTime);
                eventQueue.enqueue(new Event(server->getIndex(), finishTime)); // Enqueue ServerFinish
                ++noWaitCount;
            }
            else {
                // No idle server available
                customerQueue.enqueue(new Customer(currentTime, event->getTallyTime(), event->isCash())); // Enqueue Customer
            }
            // Read in the next customer
            if (fin >> arrivalTime >> tallyTime >> paymentType)
                eventQueue.enqueue(new Event(arrivalTime, tallyTime, (strcmp(paymentType, "cash") == 0) ? true : false)); // Enqueue CustomerArrival
        }
        else {
            // Event is ServerFinish
            ++nCustomersServed;
            servers.getServer(event->getType())->addCustomerServed();
            idleServers.enqueue(servers.getServer(event->getType())); // Enqueue idle server
            if (!customerQueue.isEmpty()) {
                Customer *customer = customerQueue.dequeue(currentTime); // Dequeue Customer
                Server *server = idleServers.dequeue(); // Dequeue idle server
                double serviceTime = customer->getTallyTime() * server->getEfficiency() + (customer->isCash() ? 0.3 : 0.7);
                double finishTime = currentTime + serviceTime;
                server->addServiceTime(serviceTime);
                eventQueue.enqueue(new Event(server->getIndex(), finishTime)); // Enqueue ServerFinish
                delete customer;
            }
        }
        delete event;
    }
    fin.close();
    double totalTime = currentTime - firstArrivalTime;
    cout << "Number of customers served: " << nCustomersServed
         << "\nTime taken to serve all customers: " << totalTime
         << "\nGreatest length reached by the customer queue: " << customerQueue.getGreatestLength()
         << "\nAverage length of the customer queue: " << customerQueue.getAverageLength(totalTime)
         << "\nAverage customer waiting time in queue: " << customerQueue.getAverageTime(nCustomersServed)
         << "\nPercentage of customers with zero waiting time: " << ((double)noWaitCount / nCustomersServed * 100) << "%"
         << "\n"
         << endl;
    servers.printStats(totalTime);
    servers.deleteServers(); // free dynamic memory
    return 0;
}

/*-------------------------------------------------------------------------------
Customers waiting to be served are placed into a FIFO queue, implemented using
modular arithmetic (circular), which keeps track of the rear and front of the queue.
*/

// A new customer is placed into the rear of the queue.
void CustomerQueue::enqueue(Customer *customer) {
    if (isFull()) {
        cerr << "MAX_CUSTOMERS exceeded." << endl;
        exit(1);
    }
    rear_ = (rear_ + 1) % MAX_CUSTOMERS; // modular arithmetic so rear_ "wraps around" to zero when it reaches MAX_CUSTOMERS
    customers_[rear_] = customer;
    ++nCustomers_;
    if (nCustomers_ > greatestLength_)
        greatestLength_ = nCustomers_;
}

// The customer in the front of the queue is removed.
Customer * CustomerQueue::dequeue(double currentTime) {
    if (isEmpty()) {
        cerr << "CustomerQueue is empty." << endl;
        exit(1);
    }
    Customer *customer = customers_[front_];
    double timeInQueue = currentTime - customer->getArrivalTime();
    totalTimeInQueue_ += timeInQueue;
    front_ = (front_ + 1) % MAX_CUSTOMERS; // modular arithmetic so front_ "wraps around" to zero when it reaches MAX_CUSTOMERS
    --nCustomers_;
    return customer;
}

/*-------------------------------------------------------------------------------
CustomerArrival and ServerFinish events are placed into a priority queue,
implemented using a min-heap.
*/

// Inserts a new element into the heap.
// The new element is placed at the end of the heap and siftUp moves it up into the correct position.
void EventQueue::enqueue(Event *event) {
    if (isFull()) {
        cerr << "MAX_EVENTS exceeded." << endl;
        exit(1);
    }
    events_[nEvents_++] = event;
    siftUp(nEvents_ - 1);
}

```

```cpp
// Removes the top element in the heap.
// The top element is replaced with the last element in the heap,
// and siftDown then moves that element back to the bottom of the heap.
Event * EventQueue::dequeue() {
    if (isEmpty()) {
        cerr << "EventQueue is empty." << endl;
        exit(1);
    }
    Event *event = events_[0];
    events_[0] = events_[nEvents_ - 1];
    --nEvents_;
    siftDown(0);
    return event;
}

// Swap the addresses that the pointers are pointing to using reference-to-pointer.
void EventQueue::swapEvent(Event *&a, Event *&b) {
    Event *temp = a;
    a = b;
    b = temp;
}

// Min heap
// Moves element up to its correct position.
void EventQueue::siftUp(unsigned int i) {
    if (i == 0) // then the element is the root
        return;
    unsigned int p = (i - 1) / 2; // integer division to find the parent
    if (*events_[p] < *events_[i]) // parent is smaller, so we will leave it as is
        return;
    else {
        swapEvent(events_[i], events_[p]); // put smallest in parent
        siftUp(p); // and siftUp parent
    }
}

// Min heap
// Moves element down to its correct position.
void EventQueue::siftDown(unsigned int i) {
    unsigned int left = i * 2 + 1; // index of the left child
    if (left >= nEvents_)
        return; // left child does not exist
    unsigned int smallest = left;
    unsigned int right = left + 1; // index of the right child
    if (right < nEvents_) // right child exists
        if (*events_[right] < *events_[smallest]) // right child is smallest child
            smallest = right;
    if (*events_[smallest] < *events_[i]) {
        swapEvent(events_[i], events_[smallest]);
        siftDown(smallest);
    }
}

/*----------------------------------------------------------------------------*/

// Prints the statistics for each server.
void Servers::printStats(double totalTime) {
    cout << left
         << setw(12) << "Server"
         << setw(12) << "Efficiency"
         << setw(12) << "Customers"
         << setw(12) << "Idle Time" << endl;
    for (unsigned int i = 0; i < nServers_; ++i) {
        cout << left
             << setw(12) << i
             << setw(12) << servers_[i]->getEfficiency()
             << setw(12) << servers_[i]->getCustomerCount()
             << setw(12) << totalTime - servers_[i]->getServiceTime() << endl;
    }
}

// Inserts a server into the server array.
```

```cpp
void Servers::addServer(Server *server) {
    if (isFull()) {
        cerr << "MAX_SERVERS exceeded." << endl;
        exit(1);
    }
    servers_[nServers_++] = server;
}

// Frees dynamic memory.
void Servers::deleteServers() {
    while (!isEmpty()) {
        delete servers_[nServers_ - 1];
        --nServers_;
    }
}

/*---------------------------------------------------------------------------------
Idle servers are placed into a priority queue, implemented using a min-heap.
*/

// Inserts a new element into the heap.
// The new element is placed at the end of the heap and siftUp moves it up into the correct position.
void IdleServers::enqueue(Server *server) {
    if (isFull()) {
        cerr << "MAX_SERVERS exceeded." << endl;
        exit(1);
    }
    idleServers_[nIdleServers_++] = server;
    siftUp(nIdleServers_ - 1);
}

// Removes the top element in the heap (fastest idle server, or NULL if there are no idle servers).
// The top element is replaced with the last element in the heap,
// and siftDown then moves that element back to the bottom of the heap.
Server * IdleServers::dequeue() {
    if (isEmpty())
        return NULL;
    Server *server = idleServers_[0];
    idleServers_[0] = idleServers_[nIdleServers_ - 1];
    --nIdleServers_;
    siftDown(0);
    return server;
}

// Swap the addresses that the pointers are pointing to using reference-to-pointer.
void IdleServers::swapServer(Server *&a, Server *&b) {
    Server *temp = a;
    a = b;
    b = temp;
}

// Min heap
// Moves element up to its correct position.
void IdleServers::siftUp(unsigned int i) {
    if (i == 0) // then the element is the root
        return;
    unsigned int p = (i - 1) / 2; // integer division to find the parent
    if (*idleServers_[p] < *idleServers_[i]) // parent is smaller, so we will leave it as is
        return;
    else {
        swapServer(idleServers_[i], idleServers_[p]); // put smallest in parent
        siftUp(p); // and siftUp parent
    }
}

// Min heap
// Moves element down to its correct position.
void IdleServers::siftDown(unsigned int i) {
    unsigned int left = i * 2 + 1; // index of the left child
    if (left >= nIdleServers_)
        return; // left child does not exist
    unsigned int smallest = left;
    unsigned int right = left + 1; // index of the right child
    if (right < nIdleServers_) // right child exists
        if (*idleServers_[right] < *idleServers_[smallest]) // right child is smallest child
            smallest = right;
    if (*idleServers_[smallest] < *idleServers_[i]) {
        swapServer(idleServers_[i], idleServers_[smallest]);
        siftDown(smallest);
    }
}

/*-------------------------------------------------------------------------------

The customer queue (FIFO queue) is implemented as a circular buffer using modular
arithmetic to keep track of the rear and front of the queue. Enqueue and dequeue
is fast because it never needs to be sorted.

The event queue (priority queue) is implemented using a min-heap, so the root
element always contains the event with the smallest event time. When adding an
event, it is placed at the end of the heap and siftUp moves it up into the
correct position. When removing an event, the top element is replaced with the
last element in the heap, and siftDown then moves that element back to the bottom
of the heap [see lecture slides, Week 3, "Priority queues"].

Similarly to events, idle servers are placed into a priority queue implemented
using a min-heap, so the root element always contains the server with the best
efficiency. Servers are removed from the heap if they are busy, and placed back
when they become free (idle).

The swap functions (as used by the siftUp and siftDown heap functions) swap the
memory addresses that the pointers are pointing to, opposed to entire objects.

-------------------------------------------------------------------------------*/
```