Northeastern University
College of Engineering
Department of Electrical & Computer Engineering

EECE 2560: Fundamentals of Engineering Algorithms

Spring 2020 - Homework 4

Instructions

- For programming problems:
  1. Your code must be well commented by explaining what the lines of your program do. Have at least one comment for every 4 lines of code.
  2. You are not allowed to use any advanced C++ library unless it is clearly allowed by the problem. For example, you cannot use a library function to sort a list of data if the problem is asking you to implement an algorithm to sort the list.
  3. At the beginning of your source code files write your full name, students ID, and any special compiling/running instruction (if any).
  4. Test your code on the COE Linux server before submitting it:
     a. If your program does not compile in the COE server due to incompatible text encoding format, then before uploading your source code file to the server make sure it is saved with Encoding Unicode (UTF-8). In visual studio, Save As -> Click on the arrow next to Save -> Save with Encoding -> Yes -> Unicode (UTF-8) -> Ok
     b. Compile using g++ -std=c++11 <filename>

- Submit the following to the homework assignment page on Blackboard:
  1. Your homework report developed by a word processor (no handwritten or drawn contents are acceptable) and submitted as one PDF file. The report includes the following (depending on the assignment contents):
     a. Answers to the non-programming problems.
     b. A summary of your approach to solve the programming problems.
     c. A summary of the skills you acquired and challenges you faced by implementing the programs.
     d. Your recommendations of extension to the programming problems.
     e. The screen shots of the sample run(s) of your program(s)
  2. Your well-commented programs source code files (i.e., the .cc or .cpp files).

  Do NOT submit any files (e.g., the PDF report file and the source code files) as a compressed (zipped) package. Rather, upload each file individually.

Note: Yon can submit multiple attempts for this homework, however, only your last submitted attempt will be graded.

Problem 1 (100 Points)

In this problem you will test and analyze the following sorting algorithms: Selection, Bubble, Insertion, Merge, and Quick. You can use the code provided in the lecture slides after modifying it to meet the following requirements:

1. Analyze the algorithms by sorting, in ascending order, arrays of 1000 integers (i.e., no need to use template functions as in the lecture slides).
2. Create three arrays of 1000 integers: BST, AVG, and WST. Where

a. BST has 1000 integers already sorted in ascending order (e.g., 10, 20, 30, …etc.).
b. AVG has 1000 randomly generated integers, where each integer is between 0 and 100,000.
c. WST has 1000 integers sorted in reverse order (e.g., 1000, 990, 980, …etc.).

3. Test the five sorting algorithms with three 1000-integer arrays: tBST, tAVG, and tWST. Before testing each algorithm, these three arrays (tBST, tAVG, and tWST) must be re-initialize as copies from BST, AVG, and WST respectively (you can write a function to carry out the copy task). This is to make sure that all algorithms are tested with an identical set of arrays.
4. Define two global integer variables: `moves` and `comps`. Re-initialize these variables to zero before each call of the five sorting algorithms. Add any necessary code to the algorithms implementation to carry out the following:
   a. Increment `moves` with every movement of any of the elements to be sorted from one "place" to another. However, the following is not considered a move: making a "copy" of an element that will not result eventually of that element being copied to another location in the array (i.e., the element stays in its place). Consider that the swap operation requires three movements of two elements.
   b. Increment `comps` with each comparison operation on the elements to be sorted.
   c. After each call to the sorting algorithms functions (5 algorithms × 3 arrays = 15 calls), write to a text file, `sort.txt`, the values of `moves` and `comps`.
5. After calling an algorithm function to sort an array, verify that the array is sorted. One way to do that is to confirm that every element *i* in the sorted array ≤ element *i*+1 (write a function for this task and correct the algorithm implementation code that results in a false verification).
6. After running your program, use the numbers in the text file, `sort.txt`, to generate two graphs in an excel sheet. One graph compares the number of moves needed by the five algorithms for the three cases: best, average, and worst. Another graph does the same but for the number of comparisons performed by the algorithms.

In your homework report, answer the following:

i. Why do the Selection, Bubble, and Insertion sort algorithms must result in zero moves when sorting an already sorted array (the best case)?
ii. Why do the Bubble and Insertion sort algorithms must result in 999 comparisons when sorting an already sorted array (the best case)?
iii. Why does the Selection sort algorithm must result in 1500 moves when sorting a reverse sorted array (the worst case)?
iv. Why does the Merge sort algorithm result in the same number of moves to sort the three arrays?
v. From your two excel graphs, comment on the performance of the algorithms under different scenarios (best, average, and worst).

Programming Hints for Problem 1

- The following C++ code shows you how to create a text file in which you can save the output of your program:

```
#include <iostream>
#include <fstream>
#include <stdexcept>

using namespace std;

int main() {
	ofstream outf;

	outf.open("sort.txt");
	if (outf.fail()) {
		cerr << "Error: Could not open output file\n";
		exit(1);
	}

	outf << "\t\t Hello World \n";

	outf.close(); //Close the file at the end of your program.
	return 0;
}
```

- The following C++ code shows you how to generate 5 random integers. Each integer is between 0 and 100.

```
#include <chrono>

int main() {
	int A[5];

	srand(time(NULL)); //call this only once at the beginning to
	                   // allow rand() to generate a different
	                   // succession of random values.

	for (int i = 0; i < 5 ; i++)
		A[i] = rand() % 100;

	return 0;
}
```

Problem 2 (100 Points)

In this problem you will implement the event-driven simulation of a bank customer service that is described in the lecture slides. In this simulation you need to maintain the customers' arrival and departure events in a priority queue, prioritized by the time of the event (the sooner the time the higher the priority). Another regular queue must be used to maintain the bank waiting line where the already arrived customers wait to be serviced. The customer at the front of this queue is serviced once a teller is available. The input to your simulation will be from a text file that contains the customers' arrival and transaction times. Each line of the file contains the arrival time and required transaction time for each customer.

Your program must collect the required data so that after the last event is processed, the program must store the following statistics in an output text file:

3. Customers count.
4. The average and maximum customers' waiting time.
5. The maximum length (in terms of number of customers) of the waiting line.

In the output file you need also to write a trace of the events executed (i.e., the time at which each arrival and departure event is processed).

Code Requirements:

Write the C++ code to implement the bank simulation with the following classes:
Class *BankCustomer* that maintains the data of each customer arrives to the bank. For simplicity, all attributes are defined as public.

BankCustomer	

public:	`int ID;`
	`int ArrivalTime;`
	`int ServiceStartTime;`
	`int TransactionLength;`

Notes: The following statistics can be calculated from the above attributes as: - A customer's waiting time = ServiceStartTime – ArrivalTime - A customer's departure time = ServiceStartTime + TransactionLength	

Class *Event* that represents the simulation customers' arrival and departure events. For simplicity, all attributes are defined as public.

Event	

public:	`char EventType; //'A' for arrival and 'D' for departure`
	`int CustID; // ID of the customer who created the event`
	`int ADTime; //Arrival or Departure time`

Notes: - Arrival events are all created at the beginning of the simulation from the input text files. - Departure events are created during the simulation. One departure event is created for every customer that starts receiving his/her service form a bank teller.	

Class *PQueue* for the priority queue that maintains the simulation events.

PQueue	

private:	`int  QCapacity; //Max size of the queue`
	`Event* events; //Pointer to the array that holds arrival & departure events`
	`int  count; //Number of events currently in the queue`
	`bool listInsert(Event ev); //insert a new event to a sorted list`
	`bool listDelete(); //Delete the event at the end of the list`
	`Event listPeek(); //Get the event at the end of the list`

public:	`PQueue(int c); //c for the Qcapacity`
	`~PQueue();`
	`bool isEmpty();`
	`bool enqueue(Event ev); //calls listInsert to insert a new event`
	`bool dequeue();//calls listDelete to remove the queue's front event`
	`Event  peekFront();//calls listPeek to return the queue's front event`

Notes: - The array contains events sorted by their ADTime in **descending** order. Therefore, the next event to be processed is the last event in the list, which has the least ADTime (i.e., the "front" of the priority queue is in the last current index of the sorted list). - If an arrival event and a departure event have the same ADTime, arrange them in this queue so that the "arrival" event is processed first. - The constructor sets the queue capacity and creates, dynamically, the empty events array. - The destructor deletes the memory allocated to the events array. - Separating the queue methods from the list methods is a good practice in case if we want to implement the queue with a different data structure in the future.	

Class *ArrayQueue* for the FIFO queue that maintains the bank waiting line. It is implemented as a circular array.

ArrayQueue	

private:	`int QCapacity;`
	`int* IDs;     // Array of queue items (the customers' IDs in this case)`
	`int front; // Index to front of queue`
	`int back;  // Index to back of queue`
	`int count; // Number of items currently in the queue`

public:	`ArrayQueue(int c);`
	`~ArrayQueue();`
	`bool isEmpty();`
	`bool enqueue(const int newID);`
	`bool dequeue();`
	`int peekFront();`
	`int queueLength(); //returns the current number of items in the queue`

Notes: - The constructor initializes the queue parameters and creates, dynamically, the empty IDs array. - The destructor deletes the memory allocated to the IDs array.	

Class *BankTellerService* simulates the bank teller service.

<table>
<tr><th colspan="2">BankTellerService</th></tr>
<tr><td>private:</td><td>

```
bool BusyTeller; //indicates if the teller is currently busy or available
int CustomersNum; //the total number of customers in the simulation
ifstream inf; //text file with the arrival and transaction times
ofstream otf; //text file with the trace messages and the final statistics
ArrayQueue WaitingQ = ArrayQueue(waitingQCapacity);
PQueue     EventsQ = PQueue(eventsQCapacity);
BankCustomer* Customers; //array of the customer objects in the simulation
int waitingQMax = 0; //the maximum length of the waiting queue
```

</td></tr>
<tr><td>public:</td><td>

```
BankTellerService(string infS, string otfS);
~BankTellerService();
void readCustomersInfo();//reads the customers' arrival and transaction times
       //from the input file. For each arrival data, add one customer object
       //in the Customers array and insert an arrival event in the EventsQ.
void serveCustomers();//executes a loop that continues as long as there are
              //events in the EventsQ. Serve each event in order and according
              //to the event type (Arrival or Departure). A customer who
              //receives service from an available teller, will result in
              //a departure event to be inserted in the EventQ. An arrival
              //event with a busy teller will result in adding the
              //corresponding customer ID in the WaitingQ. A departure event
              //results in serving a Customer from the WaitingQ or mark the
              //teller as available (if the WaitingQ is empty)
void getStatistics();//calculates the needed statistics based on the
                     //updated data of the customers and stores these
                     //statistics in the output text file.
```

</td></tr>
<tr><td colspan="2"><u>Notes</u>:
- <code>waitingQCapacity</code> and <code>eventsQCapacity</code> are constants defined in the program.
- The constructor initializes the class parameters, creates dynamically the Customers array, and opens both text files, while checking for any error exceptions with opening these files.
- The destructor deletes the memory allocated to the Customers array and closes the text files.
- For simplicity, assume, for each Customer object, a customer ID equals his/her index in the Customers array.</td></tr>
</table>

Testing your Code

The following `main()` function is an example of testing the above classes.

```
int main() {
      BankTellerService myBank("input.txt", "output.txt");
      myBank.readCustomersInfo();
      myBank.serveCustomers();
      myBank.getStatistics();
      return 0;
}
```

Test your code with the following contents of `input.txt` (where each line in the file contains, for each customer, his/her arrival time and his/her required transaction time):

```
1 5
2 5
4 5
20 5
22 5
24 5
26 5
28 5
30 5
88 3
```

For the above input data, the expected content of `output.txt` will be:

```
Simulation traces:
Processing an arrival event at time <-- 1
Processing an arrival event at time <-- 2
Processing an arrival event at time <-- 4
Processing a departure event at time -->6
Processing a departure event at time -->11
Processing a departure event at time -->16
Processing an arrival event at time <-- 20
Processing an arrival event at time <-- 22
Processing an arrival event at time <-- 24
Processing a departure event at time -->25
Processing an arrival event at time <-- 26
Processing an arrival event at time <-- 28
Processing an arrival event at time <-- 30
Processing a departure event at time -->30
Processing a departure event at time -->35
Processing a departure event at time -->40
Processing a departure event at time -->45
Processing a departure event at time -->50
Processing an arrival event at time <-- 88
Processing a departure event at time -->91

Final Statistics:
Total number of customers processed: 10
Average waiting time = 5.6      Maximum waiting time = 15
Maximum waiting queue length = 4
```

Extra Credit (20 Points)

Modify the simulation so that it accounts for any number of tellers. Verify that the customers waiting time and maximum queue length will be zero if the number of tellers equal the expected maximum waiting queue length. Analyze the simulation for different data input and different number of tellers. Support your analysis with graphs.

Programming Hints for Problem 2

- The following C++ code shows you how to read integers from a text file and store them in an array:

```
#include <iostream>
#include <fstream>
#include <stdexcept>

#define maxints 100
using namespace std;

int main() {
	ifstream inf;
	int count = 0;
	int x;
	int list[maxints];

	inf.open("c:\\temp\\ints.txt");
	if (inf.fail())
	{
		cerr << "Error: Could not open input file\n";
		exit(1);
	}
	//activate the exception handling of inf stream
	inf.exceptions(std::ifstream::failbit | std::ifstream::badbit);

	while (count < maxints) { //keep reading until reading maxints or
				  //until a reading failure is caught.
		try {
			inf >> x;
		}
		//Check for reading failure due to end of file or
		// due to reading a non-integer value from the file.
		catch (std::ifstream::failure e) {
			break;
		}
		list[count++]=x;
	}
		for(int i = 0; i < count; i++)
			cout << list[i] << endl;

		inf.close(); //Close the file at the end of your program.
		return 0;
}
```