EECS402, Fall 2017, Project 5 (aka 4b)

Overview:

This project will focus on developing an event-driven simulation. The primary portion of this project is very different from previous projects, as it is not very detailed in the specifications. Before getting to the simulation itself, though, there is some additional work needed to prepare for its development.

Due Date and Submitting:

This project is due on **Monday, December 4 at 4:00pm**. Early and late submissions are allowed, with corresponding bonus points or penalties, according to the policy described in detail in the course syllabus.

For this project, you must submit several files. You must submit each header file (with extension .h), each source file (with extension .cpp), and each templated class implementation file (with extension .inl) that you create in implementing the project. In addition, you must submit a valid UNIX Makefile, that will allow your project to be built using the command "make" resulting in an executable file named "proj5.exe". Also, your Makefile must have a target named “clean” that removes all of your .o files and your executable (but not your source code!).

When submitting your project, be sure that **every** source file (.h, .cpp, and .inl files!!!) and your valid Makefile are attached to the submission email. The submission system will respond with the number of files accepted as part of your submission, and a list of all the files accepted – it is ***your responsibility*** to ensure all source files were attached to the email and were accepted by the system. If you forget to submit a file on accident, we will not allow you to add the file after the deadline, so please take the time to carefully check the submission response email to be completely sure every single file you intended to submit was received and accepted by the system.

Detailed Description:

In the previous project, you developed some linked data structures to store integer data – a sorted doubly-linked list, a first-in-first-out queue, and a last-in-first-out stack. You will be using your solution for project 4 as a starting point for this project, so if necessary, you should quickly fix any remaining issues with those implementations.

Preparation Work:

To prepare for designing and developing the event-driven simulation, you’ll need some data structures to store simulation-related data. In project 4 you developed the data structures you’ll need, but since they only store integers, they’ll have to be updated. Rather than update them to only store the project 5-specific data types, your first step will be to “templatize” the LinkedNodeClass, the SortedListClass, and the FIFOQueueClass. The LIFOStackClass developed on project 4 will not be needed, so you don’t need to do any further work on that one.

The data structure classes are included below in full – do not add, remove, or change the class interfaces from what is provided. For the most part, the interfaces are exactly as they were in project 4, except data values utilize the placeholder type T instead of int. There are a small number of additional functions that you will need to implement as well. To make those additions stand out, I’ve colored them in red.

Remember, the template function implementations must be in files with extension .inl as opposed to .cpp, and the corresponding .inl file must be included after the class definition in that class' .h file. Also remember, .inl files are treated like .h files and are NOT compiled individually like .cpp files are, so make sure your Makefile is set up appropriately.

LinkedNodeClass:

This templated class will be used to store individual nodes of a doubly-linked data structure. This class should end up being quite short and simple – no significant complexity is needed, desired, or allowed. The interface to the LinkedNodeClass will be ***exactly*** as follows:

```
//The list node class will be the data type for individual nodes of
//a doubly-linked data structure.
template < class T >
class LinkedNodeClass
{
  private:
    LinkedNodeClass *prevNode; //Will point to the node that comes before
                               //this node in the data structure. Will be
                               //NULL if this is the first node.
    T nodeVal;                 //The value contained within this node.
    LinkedNodeClass *nextNode; //Will point to the node that comes after
                               //this node in the data structure. Will be
                               //NULL if this is the last node.
  public:
    //The ONLY constructor for the linked node class - it takes in the
    //newly created node's previous pointer, value, and next pointer,
    //and assigns them.
    LinkedNodeClass(
         LinkedNodeClass *inPrev, //Address of node that comes before this one
         const T &inVal,          //Value to be contained in this node
         LinkedNodeClass *inNext  //Address of node that comes after this one
         );

    //Returns the value stored within this node.
    T getValue(
         ) const;

    //Returns the address of the node that follows this node.
    LinkedNodeClass* getNext(
         ) const;

    //Returns the address of the node that comes before this node.
    LinkedNodeClass* getPrev(
         ) const;

    //Sets the object's next node pointer to NULL.
    void setNextPointerToNull(
```

```
        );

    //Sets the object's previous node pointer to NULL.
    void setPreviousPointerToNull(
        );

    //This function DOES NOT modify "this" node. Instead, it uses
    //the pointers contained within this node to change the previous
    //and next nodes so that they point to this node appropriately.
    //In other words, if "this" node is set up such that its prevNode
    //pointer points to a node (call it "A"), and "this" node's
    //nextNode pointer points to a node (call it "B"), then calling
    //setBeforeAndAfterPointers results in the node we're calling
    //"A" to be updated so its "nextNode" points to "this" node, and
    //the node we're calling "B" is updated so its "prevNode" points
    //to "this" node, but "this" node itself remains unchanged.
    void setBeforeAndAfterPointers(
        );
};
```

SortedListClass:

This templated class will be used to store a doubly-linked list in an always-sorted way, such that the user does not specify where in the list a value should be inserted, but rather the new value is inserted in the correct place to maintain a sorted order. The interface to the SortedListClass will be ***exactly*** as follows:

```
//The sorted list class does not store any data directly.  Instead,
//it contains a collection of LinkedNodeClass objects, each of which
//contains one element.
template < class T >
class SortedListClass
{
  private:
    LinkedNodeClass< T > *head; //Points to the first node in a list, or NULL
                                //if list is empty.
    LinkedNodeClass< T > *tail; //Points to the last node in a list, or NULL
                                //if list is empty.

  public:
    //Default Constructor.  Will properly initialize a list to
    //be an empty list, to which values can be added.
    SortedListClass(
        );

    //Copy constructor.  Will make a complete (deep) copy of the list,
    //such that one can be changed without affecting the other.
    SortedListClass(
        const SortedListClass< T > &rhs
        );

    //Overloaded assignment operator.  Will make a complete (deep) copy
```

```
//of the list, such that one can be changed without affecting
//the other.
void operator=(
     const SortedListClass< T > &rhs
     );

//Clears the list to an empty state without resulting in any
//memory leaks.
void clear(
     );

//Allows the user to insert a value into the list.  Since this
//is a sorted list, there is no need to specify where in the list
//to insert the element.  It will insert it in the appropriate
//location based on the value being inserted.  If the node value
//being inserted is found to be "equal to" one or more node values
//already in the list, the newly inserted node will be placed AFTER
//the previously inserted nodes.
void insertValue(
     const T &valToInsert  //The value to insert into the list
     );

//Prints the contents of the list from head to tail to the screen.
//Begins with a line reading "Forward List Contents Follow:", then
//prints one list element per line, indented two spaces, then prints
//the line "End Of List Contents" to indicate the end of the list.
void printForward(
     ) const;

//Prints the contents of the list from tail to head to the screen.
//Begins with a line reading "Backward List Contents Follow:", then
//prints one list element per line, indented two spaces, then prints
//the line "End Of List Contents" to indicate the end of the list.
void printBackward(
     ) const;

//Removes the front item from the list and returns the value that
//was contained in it via the reference parameter.  If the list
//was empty, the function returns false to indicate failure, and
//the contents of the reference parameter upon return is undefined.
//If the list was not empty and the first item was successfully
//removed, true is returned, and the reference parameter will
//be set to the item that was removed.
bool removeFront(
     T &theVal
     );

//Removes the last item from the list and returns the value that
//was contained in it via the reference parameter.  If the list
//was empty, the function returns false to indicate failure, and
//the contents of the reference parameter upon return is undefined.
//If the list was not empty and the last item was successfully
```

```
    //removed, true is returned, and the reference parameter will
    //be set to the item that was removed.
    bool removeLast(
         T &theVal
         );

    //Returns the number of nodes contained in the list.
    int getNumElems(
         ) const;

    //Provides the value stored in the node at index provided in the
    //"index" parameter.  If the index is out of range, then outVal
    //remains unchanged and false is returned.  Otherwise, the function
    //returns true, and the reference parameter outVal will contain
    //a copy of the value at that location.
    bool getElemAtIndex(
         const int index,
         T &outVal
         );

    //Destructor, which will free up all dynamic memory associated
    //with this list when the list is destroyed (i.e when a statically
    //allocated list goes out of scope or a dynamically allocated list
    //is deleted).
    ~SortedListClass(
         );
};
```

FIFOQueueClass:

This templated class will be used to store a simple first-in-first-out queue data structure. It's full and complete specification is as follows, and you must implement this ***exactly*** as specified:

```
template < class T >
class FIFOQueueClass
{
  private:
    LinkedNodeClass< T > *head; //Points to the first node in a queue, or NULL
                                //if queue is empty.
    LinkedNodeClass< T > *tail; //Points to the last node in a queue, or NULL
                                //if queue is empty.

  public:
    //Default Constructor.  Will properly initialize a queue to
    //be an empty queue, to which values can be added.
    FIFOQueueClass(
         );

    //Inserts the value provided (newItem) into the queue.
    void enqueue(
         const T &newItem
         );
```

```
    //Attempts to take the next item out of the queue.  If the
    //queue is empty, the function returns false and the state
    //of the reference parameter (outItem) is undefined.  If the
    //queue is not empty, the function returns true and outItem
    //becomes a copy of the next item in the queue, which is
    //removed from the data structure.
    bool dequeue(
         T &outItem
         );

    //Prints out the contents of the queue.  All printing is done
    //on one line, using a single space to separate values, and a
    //single newline character is printed at the end.
    void print(
         ) const;

    //Destructor, which will free up all dynamic memory associated
    //with this queue when the list is destroyed (i.e when a statically
    //allocated queue goes out of scope or a dynamically allocated queue
    //is deleted).
    ~FIFOQueueClass(
         );
};
```

Event-Driven Simulation:

Once your data structures are implemented and tested, you will develop an event driven simulation. Create an event class that will be inserted into a SortedListClass in a sorted way based on the time that the event is scheduled to occur. Then, handle one event at a time, as discussed in lecture. As you handle certain events, new events will be generated to occur at a future time (randomly drawn from a specified distribution), and the simulation will advance much like the airport example demonstrated in class. Note: you must use the data structures you developed in project 4 and "templatized" in this project – do not use any STL containers when implementing this project.

The simulation to implement will be a server simulation at a fast food restaurant. Customers should come into the restaurant on a pseudo-random basis, where each customer enters the restaurant some amount of time after the previous customer. The amount of time between customers should be drawn from a *uniform* distribution, which has specified min and max values. Your restaurant will have only a single server, in order to make things much simpler. If the server is not currently waiting on a customer, a new customer can immediately get served. Otherwise, the customer will have to wait in a first-in-first-out queue and wait their turn. The amount of time it takes for the server to wait on the customer will be drawn from a *normal* distribution with a specified mean and standard deviation.

This simulation ***MUST*** be implemented as an *event-driven simulation* as described in lecture. If your simulation is implemented as a time-driven simulation (or any variant on a time-driven simulation) or you otherwise generate and handle events in a way that is not the way described in lecture for an event-driven simulation, you will not receive credit for your simulation.

Another very important requirement - you must generate new objects only when you need them. That is, do NOT generate a full list of customers and their arrival times at the beginning of the simulation. Instead, generate a single customer to arrive at a determined time to start off the simulation. Then, when handling that arrival event, determine when the next customer will arrive and generate a new arrival event. That is (please note!):

- There must only be at most one customer arrival event in the event list at any given moment
- At the time you handle a customer's arrival event, determine how far in the future the next arrival event occurs using BOTH a minimum and a maximum (do NOT assume a minimum of 0).
    - For example, if a customer's arrival time is 100, then, when you are handling that event, you will generate the next arrival event. If the uniform distribution has a min of 10 and a max of 20, then the next arrival event must be within the range 110 to 120 (as opposed to 100 to 120).

Similarly, do not compute a customer's service time until that customer begins being served (in other words, do not compute all the timing for a customer right when the customer is first created – instead, compute the duration of the customer's service time only when the customer reaches the front of the line and you're able to compute the time the service will end).

Your simulation must provide enough console output to allow a user to easily follow and understand what is happening at the restaurant. For example, you should print a description when a customer enters the restaurant, whether the customer is served or has to wait in line, how many people are in front of the customer in the line, when the server finished serving a customer, etc., etc., etc. I should be able to look at your simulation's output and "see" what is happening in the restaurant and follow a customer through the process. The simulation should run from time = 0 to a specified time, after which the simulation ends and maintained statistics are output.

Finally, you should keep track of some basic statistics. At a minimum, you must keep track of:

- Total number of customers simulated
- What percentage of time the server was busy helping customers
- What percentage of customers had to wait in line
- The longest the line was throughout the simulation

Come up with some more interesting stats to keep track of and print those out as well. Provide the four statistics above and *at least three more.*

See the assignment page on Canvas to obtain the random number generation code you should use in your program.