EECS402, Winter 2013, Project 4

Overview:
This project will focus on linked data structures and an event-driven simulation. Some parts of this project are very different from previous projects, as they may not be very detailed in the specifications. Also, the project is divided in phases. It is strongly recommended that you complete phase I before moving on two phase II, and then complete phase II before moving onto phase III. This will result in later phases being much easier to debug.

Due Date and Submitting:
This project is due on **Friday, March 29 at 5:30pm**. Late submissions will not be accepted for any reason.

For this project, you must submit several files. You must submit each header file (with extension .h) and each source file (with extension .cpp) 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 "proj4.exe". When submitting your project, all these files must be in the single tar file emailed. Also, your Makefile must have a target named clean that removes all your .o files and your executable.

Phase I:
In phase I, you will develop a basic generic data structure that you will use later in the project. Phase I is fully specified in significant detail, and you must implement exactly what is given in these specifications.

The first class to develop is a node class that will be one element of a linked list data structure. In order to develop your list as generically as possible, this node class will be a templated class, allowing you to use it to store any type of data.

Following is the complete and exact specification of the linked list node class. Do not change anything, regardless of how minor it may seem.

```
template < class T >
class ListNodeClass
{
  private:
    ListNodeClass *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.
    ListNodeClass *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 list node class - it takes in the
    //newly created node's previous pointer, value, and next pointer,
    //and assigns them.
    ListNodeClass(
         ListNodeClass *inPrev,  //Address of node that comes before this one
         const T &inVal,//Value to be contained in this node
         ListNodeClass *inNext   //Address of node that comes after this one
         );

    //Sets the "next" pointer of a node.  No return value.
```

```
    void setNextPointer(
         ListNodeClass *newNextPtr
         );

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

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

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

    //Sets the object's previous node pointer to the value passed in.
    void setPreviousPointer(
         ListNodeClass *newPrevPtr
         );

    //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 updates so its "prevNode" points
    //to "this" node, but "this" node itself remains unchanged.
    void setBeforeAndAfterPointers(
         );
};
```

Now that you have a node to store your data, you'll implement the linked list class. For this project, you will implement a sorted doubly-linked list. This means that when you insert an element, the class will decide where the proper location within the list is. The use of the list will not specify where a value is to be inserted, just that a value is to be inserted in a sorted way.

Following is the complete and exact specification of the linked list node class. Do not change anything, regardless of how minor it may seem.

```
template < class T >
class SortedListClass
{
  private:
    ListNodeClass< T > *head;  //Points to the first node in a list, or NULL
                           //if list is empty.
    ListNodeClass< 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(
```

```
     );

//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
     );

//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
     );
```

That's it for phase I. What you should do now is test your sorted list functionality using multiple data types. Write your own main that creates list of different data types, inserts elements, prints the lists in both directions, removes elements, etc. If you have exhaustively tested your phase I functionality you should be able to use it during later phases without having to come back to it.

Phase II:
For this phase you will develop two more data structures. First, you will create a first-in-first-out queue data structure. You must use the list node class you developed in phase I. Develop this data structure as generically as possible (i.e. make a templated class) and test it thoroughly. You should restrict the member functions to operations that can be performed on a FIFO queue. In other words, you should NOT have a member function that inserts values in the middle or takes items off the back, etc. Use function names that are standard to queues, like enqueue and dequeue.

The rest is up to you. Please note: This phase should be VERY short and simple. If you find it is time consuming or very challenging, it is likely you are doing more than is expected.

Phase III:
Once you have phases I and II done, you will develop an event driven simulation. Create an event class that will be inserted into a SortedListClass (from phase I) 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 in the future, and the simulation will advance much like the airport example demonstrated in class. Note: you must use the data structures you developed in phases I and II – do ***not*** use any STL containers when implementing this phase.

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.

One 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 should 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).

Your simulation must provide enough output to follow 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, 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.

Random Number Code:
Here are some functions you should just copy/paste into your project to get uniform and/or normal random numbers.

```
#include <cstdlib>
using namespace std;

void setSeed(
     int seedVal
     )
{
  srand(seedVal);
}

int getUniform(
     int min,
     int max
     )
{
  int uniRand;

  uniRand = rand() % ((max + 1) - min) + min;

  return (uniRand);
}

int getNormal(
     float mean,
     float stdDev
```

```
    )
{
  const int NUM_UNIFORM = 12;
  const int MAX = 1000;
  const float ORIGINAL_MEAN = NUM_UNIFORM * 0.5;
  float sum;
  int i;
  float standardNormal;
  float newNormal;
  int uni;

  sum = 0;

  for (i = 0; i < NUM_UNIFORM; i++)
  {
    uni = rand() % (MAX + 1);

    sum += uni;
  }

  sum = sum / MAX;

  standardNormal = sum - ORIGINAL_MEAN;

  newNormal = mean + stdDev * standardNormal;

  if (newNormal < 0)
  {
    newNormal *= -1;
  }

  return ((int)newNormal);
}
```

Project Grading:

As you've noticed much of this project is left to you to design and develop. Because of this, grading is expected to be very difficult, as each student can come up with their own input and output scheme. For this reason, I will meet with each student to grade his/her project. We will spend a fair amount of time testing your data structures you developed in phases I and II, and then I will ask you to provide some inputs for your simulation. I'll modify the inputs a bit and look at your simulation's output. We'll also look through the code and discuss style issues and agree on a grade for the project. Hopefully, each grading session should take no more than 15 minutes.