EECS402, Fall 2025, Project 4

Overview:

In this project you will develop three common linked data structures: a sorted doubly-linked list, a simple queue, and a simple stack. This is one half of the final project, and project 5 will fully utilize the data structures developed in this project. Therefore, you'll need to make sure your implementations are complete and bug-free so that you can focus on project 5-specific functionality during that time rather than debugging project 4 functionality.

This project is an "infrastructure"-type project again (i.e. the classes you're developing are the point of the project), so you will not be responsible for developing a specific main function of your own that performs some interesting functionality. However, you will want to write a main function to run tests on your implementation, and you will need to submit a main function for build purposes. Write your main function in the placeholder main function provided along with these specs. Do NOT remove or modify any of the preprocessor directives in the provided code (the lines that start with '#'). You may freely modify the main function in there to perform your own tests. When you submit, while the actual contents of your main function are not important, they must be syntactically correct to allow the project to build, and they must be styled consistently with the rest of your project implementation. When we run your program, we will be using our own main function.

Due Date and Submitting:

This project is due on **Thursday, November 20, 2025 at 4:30pm**. Early submissions are allowed, with corresponding bonus points, 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) 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". Also, your Makefile must have a target named "clean" that removes all of your .o files and your executable (but not your source code!). You must also submit a typescript file showing that you ran valgrind and considered its output. This type of project is where valgrind really shows its usefulness, so make sure you run valgrind on a main function that thoroughly tests the required functionality!

When submitting your project, be sure that **every** source file (both .h and .cpp files!!!), your valid Makefile, and your required typescript file 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 accepted by the system.

Detailed Description:

The classes being developed in project 4 will be completely specified – please be sure to follow these specs *exactly* - do not deviate from them in *any* way, including, but not limited to: changing the class name, method names, attribute names, types or orders of method parameters, etc.; adding functionality; neglecting functionality; changing the purpose or functionality of the methods, etc… We *will* be testing your implementation using our own main

function, which will assume everything was developed exactly as specified, so it is absolutely critical that your implementation be written exactly according to these specs.

All functions must be fully implemented "from scratch" – that is, you may not use any libraries (standard or non-standard) to accomplish the required functionality of the specified data structures, with the one exception of using <iostream> in order to print the required output.

The primary data structure will be a doubly-linked list that is always maintained in a sorted order, along with a simple node class that will be able to be used by the sorted list. Keep in mind that, while the list will fully utilize the node class, the node class really shouldn't know anything about a list – it is just a node, and it might be used in other data structures (like the others being developed for this project!).

LinkedNodeClass:

This 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.
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.
    int 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 int &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.
    int 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 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.
class SortedListClass
{
  private:
    LinkedNodeClass *head; //Points to the first node in a list, or NULL
                           //if list is empty.
    LinkedNodeClass *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 &rhs
        );
```

```
//Destructor.  Responsible for making sure any dynamic memory
//associated with an object is freed up when the object is
//being destroyed.
~SortedListClass(
     );

//Assignment operator. Will assign one list (on left hand side of
//operator) to be a duplicate of the other (on the right hand side
//of operator).
SortedListClass& operator=(
     const SortedListClass &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 int &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(
     int &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(
          int &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
     //0-based "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,
          int &outVal
          ) const;
};
```

Additional Data Structures

After you've completed implementing and fully testing the above classes, move on to implementing these additional data structures, which should be able to be completed very quickly due to their restricted functionality.

FIFOQueueClass:

This will be a "first-in-first-out queue" data structure. You should be able to use the LinkedNodeClass you developed above to very quickly develop and test this data structure. If written correctly, this class should be *very short* and simple, and should not require a significant chunk of the time to implement. Since the FIFO queue has such restricted functionality, it is quite straight forward to develop, especially since the bidirectional LinkedNodeClass is already available and can be used to make this a very short and simple data structure to implement.

This 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:

```
class FIFOQueueClass
{
  private:
    LinkedNodeClass *head; //Points to the first node in a queue, or NULL
                           //if queue is empty.
    LinkedNodeClass *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(
         );

    //NOTE: This class does NOT have a copy ctor or an overloaded
    //      assignment operator - therefore, using either of those
    //      things will result in a shallow copy. Users should not
    //      attempt to copy a FIFOQueueClass object using either of
    //      these approaches!

    //Destructor.  Responsible for making sure any dynamic memory
    //associated with an object is freed up when the object is
    //being destroyed.
    ~FIFOQueueClass(
         );

    //Inserts the value provided (newItem) into the queue.
    void enqueue(
         const int &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(
         int &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. Values will
    //be printed such that the next value that would be dequeued
    //is printed first.
    void print(
         ) const;

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

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

LIFOStackClass:

This will be a "last-in-first-out stack" data structure. You should be able to use the LinkedNodeClass you developed above to very quickly develop and test this data structure. If written correctly, this class should be *very short* and

simple, and should not require a significant chunk of the time to implement. Since the LIFO stack has such restricted functionality, it is quite straight forward to develop, especially since the bidirectional LinkedNodeClass is already available and can be used to make this a very short and simple data structure to implement.

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

```
class LIFOStackClass
{
  private:
    LinkedNodeClass *head; //Points to the first node in a stack, or NULL
                           //if stack is empty.
    LinkedNodeClass *tail; //Points to the last node in a stack, or NULL
                           //if stack is empty.

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

    //NOTE: This class does NOT have a copy ctor or an overloaded
    //      assignment operator - therefore, using either of those
    //      things will result in a shallow copy. Users should not
    //      attempt to copy a LIFOStackClass object using either of
    //      these approaches!

    //Destructor.  Responsible for making sure any dynamic memory
    //associated with an object is freed up when the object is
    //being destroyed.
    ~LIFOStackClass(
         );

    //Inserts the value provided (newItem) into the stack.
    void push(
         const int &newItem
         );

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

    //Prints out the contents of the stack.  All printing is done
    //on one line, using a single space to separate values, and a
    //single newline character is printed at the end. Values will
    //be printed such that the next value that would be popped
    //is printed first.
```

```
    void print(
         ) const;

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

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

"Specific Specifications"

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. We have not necessarily covered all the topics listed, so if you don't know what each of these is, it's not likely you would "accidentally" use them in your solution. Those types of restrictions are put in place mainly for students who know some of the more advanced topics and might try to use them when they're not expected or allowed. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: No
- Use of Friend Functions / Classes: No
- Use of Structs: No
- Use of Classes: Yes – required!
- Public Data In Classes: No (all data members must be private)
- Use of Inheritance / Polymorphism: No
- Use of Arrays: No
- Use of C++ "string" Type: No
- Use of C-Strings: No
- Use of Pointers: Yes – required!
- Use of STL Containers: **No**
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): No
- Use of overloaded operators: No
- Use of float type: No