CS 202 Summer 2022 - Assignment 5

Linked Lists and Templates

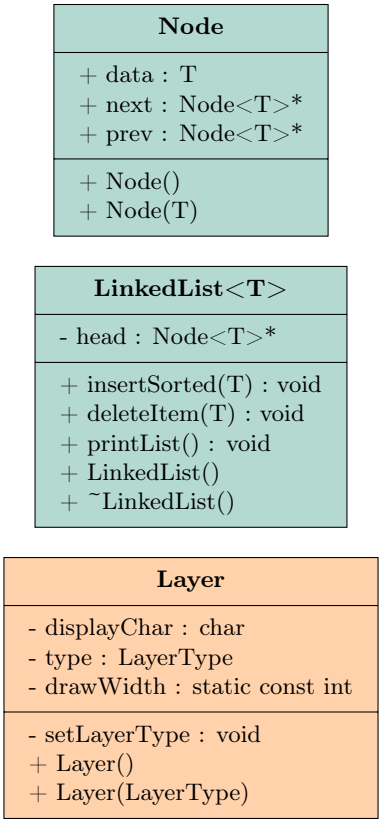
Overview

Liquids of different densities will sort themselves into Layers when poured into a container. In this assignment, we will simulate this phenomenon using a sorted Linked List. To simulate this, we will "pour" layers into the container by starting at the top of the linked list and allowing the layer to sink until it finds a spot where everything after it is more dense.

In order to do this, we will implement two major classes: A sorted doubly linked list that will sort items as it inserts and a Layer class to represent a layer of liquid like we might see in the image above. A doubly linked Node class will also be created for the Linked List to use. As items are inserted into the LinkedList, they should start at the head and work there way through the list until either a Node that they cannot go after is hit or the end of the list is reached, in which case the Node will be placed at the end. Cases where the head must be modified should also be considered - if the list is empty or the new element needs to go before the head.

In order to test the sorting process, the list will first be tested with floats. Operator overloads will be provided so that the Layers may be sorted just like the floats would. It is recommended to work things out using floats first (i.e. how do we tell that they are in the right order).

UML Diagrams



Important Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, **functions implemented for you already are in blue**, and **functions that are abstract that will be implemented later are in magenta**.

Node

The Node struct will represent a doubly linked Node for our LinkedList class. This means it has both a pointer to the next Node as well as the previous one. This struct is templated so that any type of data can be stored in it.

- ***T data*** - The data stored within this Node
- ***Node<T>* next*** - Points to the next Node in the linked list
- ***Node<T>* prev*** - Points to the previous Node in the linked list
- **Node()** - Default constructor that just sets pointers to nullptr
- **Node(T data)** - Sets the pointers to nullptr and the data to the passed parameter

LinkedList<T>

The LinkedList class implements a sorted doubly linked list with a head and the ability to insert and delete items.

- ***Node<T>* head*** - A pointer to the first Node in our list. This will be nullptr when the list is empty.

- **void insertSorted(T new_item)** - Inserts the new_item into the LinkedList in a sorted order. As an example, if we had a linked list of ints **4->5->7** and wanted to insert 6, it would end up between the 4 and 5: **4->5->6->7**. This function needs to address 3 cases, one where the list is empty, one where the new element needs to be inserted at the head, and one where the node needs to be put somewhere in the middle of the list. The first two cases may be merged, but don't have to be. Start by creating a new node to store the item being added to the list. If the list is empty, simply make the new node the head. If the new item would be the first one in the list, set the head to be that node and have it point at the old head. Otherwise, go through the list and find the correct spot to put the new node at so that it ends up in sorted order. There are multiple ways to go about this, but as a hint, try traversing the list the standard way and insert the node once either the end of the list is reached or we find that the data of the next node would be greater than the data being inserted. From the example earlier, when inserting the 6, it would not come before the 4 or 5, but it would come before the 7, because $7 > 6$. Be sure to set any next and prev pointers when inserting.
- **deleteItem(T search_item)** - This function should look through the list and delete the first node containing the search_item as its data. If no node is found containing the data, none are deleted. Recall that delete has a special case when deleting the head where the node after the deleted head must be the new head and the prev of the new head should be nullptr. Otherwise, traverse the list until the Node to delete is found, adjust the next and prev pointers of the Nodes to the left and right of it if they exist, and then delete the Node.
- **void printList()** - Prints the contents of the list with a space between each datum. Prints a new line after printing all items.
- **LinkedList()** - Sets the head to nullptr to represent an empty list
- **~LinkedList()** - The destructor should deallocate all of the Nodes in the LinkedList. Start at the head Node and deallocate each node as you traverse the list. It is recommended to keep two pointers: one to traverse the list and another to keep a temporary reference to the last Node seen so that it may be deleted without causing a memory leak. Be careful of the order; make sure to advance any pointers before they are lost from deleting a Node.

Layer

A class to represent "layers" with different densities similar to the picture from the first page of the rubric. This class is implemented for you, but you are welcome to play around with values to see how templates, operator overloading, and enums can make modifying programs very simple. With the default implementation, can represent either water, oil, or rocks. Adding new enums to the top of layers.h will change the way they get sorted.

- ***char displayChar*** - The char to use to display this layer to the screen
- ***LayerType type*** - An enum used to represent what kind of thing this layer is. Used for the purpose of sorting (recall that enums are just ints with special names)
- ***static const int drawWidth*** - Width to use when printing a layer
- **setLayerType(LayerType)** - Sets the Layer's ***type*** member and the ***displayChar*** based on what the type is
- **Layer()** - The default constructor does nothing and exists so this type will work with other templated functions
- **Layer(LayerType type)** - Initializes class members via the **setLayerType** function

Global Functions

- **int main()** - Main creates two LinkedLists then adds and removes elements and prints the results for viewing. The first LinkedList contains floats. Floats are inserted into the list, with the minimum element added into the list halfway through, requiring there to be a new head. After all floats are added, the list is printed. Some of the floats are then deleted and the list is printed again to check for successful deletion. Finally, a LinkedList housing our Layer class is made and layers are gradually poured into the list, with more dense layers sinking towards the bottom/end of the list. The Layers are read from the layers.txt file and then printed to show what they look like after settling. For the sake of debugging, preprocessor directives have been added to the main.cpp file. While you cannot use these on CodeGrade, if you comment out the line that defines the DEBUG symbol at the top, the program will contain a few more prints that may give you a better idea of where your program is having issues if it doesn't seem to be working correctly.
- **istream& operator » (istream& in, LinkedList<T>& list)** - The » operator should read a single item from the stream and then insert it into the list. This version of the operator is templated and will be generated for every version of the LinkedList. Since the list contains items of type T, this operation should read a single item of type T into a temp variable and then insert that into the List. Afterwards, return the stream.
- **bool operator >(Layer left, Layer right)** - Tells which of two Layers are greater. If the left is more dense than the right, returns true. This is based off of the order of the LayerType enum. This is implemented for the sake of the Layer class working with the LinkedList template and using > to put items in sorted order
- **bool operator <(Layer left, Layer right)** - Similar to the > overload, used for sorting Layers
- **ostream& operator « (ostream& out, const Layer& layer)** - Prints the layer to the given out stream. Prints the Layer's ***displayChar*** drawWidth many times, then returns the stream. Exists so that the Layer class can be used with the **printList** function
- **istream& operator » (istream& in, Layer& layer)** - Reads an int from the stream and then constructs a Layer with the equivalent type by casting the int to a LayerType. Exists so that the LinkedList » overload works with the Layer class

Compiling

The source code for this program consists mostly of header files with the only cpp file needing to be compiled being the two main files: float_main.cpp and layer_main.cpp. You can compile either cpp in order to generate a program. When submitting, you need only to turn in the linked_list.h file that will be included in the mains. When working with templates, code is generally limited to just header files, which is why the classes here are implemented in just .h files.
g++ -std=c++11 float_main.cpp
g++ -std=c++11 layer_main.cpp

Sample Run

```
------ Sorted Float List ------
 -2.7 4 5 5.2 6
------ Removing some floats ------
 5 5.2 6
------ Sorted Layer List ------
 --------------------
 --------------------
 --------------------
 ^^^^^^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^^^^
 OOOOOOOOOOOOOOOOOOOO
 OOOOOOOOOOOOOOOOOOOO
 OOOOOOOOOOOOOOOOOOOO
```