

CSE 232, Lab Exercise 13

The Problem

You are going to work on pointer manipulation in the context of a templated singly linked list

Some Background

In the lab directory is a definition of a template `SingleLink` list in `lab13_singlelink.h`. It is slightly non-standard in that this version maintains two pointers in the list; the `begin_` pointer, which points to the beginning node in the list and the `last_` pointer, which points to the last node in the list. Below is the interface as it stands in the `lab13_singlelink.h` file

Note:

- private data members: `Node* next_` and `T data_`.
- default constructor (done for you)
- 1 arg data constructor (done for you)


SingleLink

- `Node<T>* head_` and `Node<T>* tail_`, pointers to the first and last node of the list
 - should both point to `nullptr` if the list is empty
 - should both point to the same node if only one element in the list
- `SingleLink`. constructor, no args, default constructor
- `SingleLink(Node<T> n)` 1 parameter constructor. Adds Node as the first in the list.

What's missing?

Pretty much everything, but this is an opportunity to work through some things and get a feel for how it has to go. You probably won't get it all but work through from the beginning. Project 11 requires you get the hang of this so work it through, even after class if necessary.

Example 21.2 should be a huge help. Look it over

1. Copy the very incomplete `lab13_singlelink.h` in your directory. Also copy `lab13_main.cpp` to your working directory. You are going to modify these two files.
 - a. `main` in particular doesn't have much to it. You are going to have to write some tests to get your code to run. This will help you in the long run.
2. `void append_back(T dat)`, member function, no return. Creates a Node with `data_ = dat` and appends that Node the end of the list.
 - a.  Before you write code, you must identify all of the cases you must deal with to do a `append_back`. List them below and show them to your TA before you write any code.

Having listed the special cases, move on to implementation

- b. modify `lab13_singlelink.h` and `lab13_main.cpp` to test your code
 - c. test `lab13_singlelink.h` on Mimir
3. A friend function to print out the list. You're going to need that so do that early on
- a. `ostream& operator<<(ostream &out, SingleLink<T>& s)`, friend function, returns `ostream&`. Prints the `data_` part of each node in the list. Node is a struct with public members, so you can indeed do that.
 - b. **Modify lab13 main.cpp** and show that your method works before you move on.
 - i. Think about the special cases again
 - c. test `lab13_singlelink.h` on Mimir
4. Your second task is a `del` member function. (can't call it delete, that's a keyword)



Before you write code, you must identify all of the cases you must deal with to do a `del`. List them below and show them to your TA **before you write any code.**

Having listed the special cases, now you should implement the following:

5. `bool del(T val)`. The function:
- a. searches through the list for the first Node that has the same `data_` value as the parameter `val`. if found, deletes the node and returns true otherwise returns false
 - b. implement the `del` method
 - c. modify main to show that it works
 - d. test `lab13_singlelink.h` on Mimir
6. Now define an `operator[]` for `SingleLink`. The following is the declaration of the operator.

```
Node<T>& SingleLink<T>::operator[](size_t index)
```

On a call, such as `sl[3]`, the argument 3 is assigned to the parameter `index`. The intent is to return the 4th element in the list (assuming an index starts at 0). The return value is a **reference** to a Node so that the Node can be modified (can show up on either side of an assignment operator).

You have to search the list (from the beginning) for the index-th Node. Return a reference to that Node or throw an exception `out_of_range` exception if the index is smaller than 0 or larger than the last element in the list. Remember how exceptions work

```

#include<stdexcept>
using std::out_of_range; // standard error for a bad index
...
if (badindex)           // somehow this is a bad index
    throw out_of_range("Index out of range");

```

Again, modify the lab13_main.cpp to show that your code works, including catching an out_of_range error. Example of a try - catch is below

```

try{
    result = my_list[-1];
}
catch (out_of_range err){
    cout << "Error, message follows: ";
    cout << err.what() << endl;
}

```

a. test lab13_singlelink.h on Mimir

7. None of the rule of 3 stuff is there.
 - a. if you get time, it would be good if you implemented those.
 - b. it would be good practice for project 11
 - c. No mimir test cases for this, just give it a try.