MTH 4300
Spring 2025
E. Fink

Homework 8

1. I've provided the code for the LinkedList class from lecture. Let's modify this in a few ways.

   a. First, add a private int length member, which maintains the number of Nodes in the list. Go through the existing member functions, and update them appropriately.

   b. Implement void insert_after(const T& data, int n), where T is the type of the list. This should insert a (heap-allocated) Node into the list after the nth Node (with 1-based indexing), if there are at least n Nodes. If n is greater than the number of Nodes, or less than 1, then the function should throw a std::runtime_error – see the provided code for pop_front() and get_front().

   c. In class, we were lazy and prohibited copy construction using = delete;. Let's not be lazy anymore: implement LinkedList(const LinkedList&), the copy constructor. When used, this should create a separate-but-identical LinkedList – that is, a *deep copy* of the list, with a bunch of entirely new heap-allocated Nodes.

   There are several tricky bits:

   - You have to be careful about the case of copying an *empty* list, which will probably be simple but work differently from non-empty lists. I suggest using an if-else, with the code for handling the empty list case in the if-block, and most of the code of the function lying in the else block.
   - For non-empty lists, you'll probably want to treat the very first Node a little differently than subsequent Nodes.
   - You'll almost certainly need to maintain two "current" pointers, one that traverses the list you're copying from, and one that moves through the list you are actually constructing.
   - Don't forget about length!

   d. You don't have to do anything for this part, but please read the provided implementation for LinkedList& operator=(const LinkedList&), the copy assignment. This uses an extremely clever technique. First, instead of rewriting all the code from part c for producing a deep copy, you simply use the copy constructor to create a new copied object, and store it in a local LinkedList variable.

   The brilliant idea: trade the head pointers of the assigned-to (left-hand-side) list with the local copy. Now, the assigned-to list points to the newly-produced Nodes, while the local variable has the *old* contents of the left-hand-side. The best part is that when the local variable gets destructed at the end of the function call, these old contents will automatically get deallocated! Neat.

2. A *queue* is a data structure that represents something like a ticket line: initially, the line is empty; periodically people enter the line, at the back; every once in a while, the person at the front of the line gets called to the counter, and gets removed from the line. So, this models a "first in, first out" data structure.

   Implement a Queue template class from scratch, as follows. The underlying data should be stored in a linked list, so you will need the Node class we've become accustomed to:

```
struct Node<T> {
public:
   T data;
   Node *next;
   Node(T d, Node* n = nullptr): data{d}, next{n} {}
};
```

   The Queue class will have the following declaration:

```
template<typename T>
class Queue {
private:
    Node<T> *front;
    Node<T> *back;
public:
    Queue();
    bool is_empty();
    void push_back(const T&);
    T pop_front();
    void print_queue();
    ~Queue();
    void operator=(const Queue&) = delete;
    Queue(const Queue&) = delete;
};
```

The attributes front and back will point to the first and last elements of the Queue respectively, or to nullptr if the Queue contains no elements. New elements added to the Queue will be added to the back; when we remove elements, we remove from the front.

Implement the methods as follows:

- the constructor should just initialize a Queue with no elements. So front and back should point to nullptr. Easy.
- is_empty() should return whether or not there are any Nodes in the Queue.
- push_back(const T &s) should create a Node with data s, and add that to the end of the Queue. front and/or back should be updated appropriately. Take special care considering what should be done if the Queue is currently empty!
- pop_front() should remove a Node from the front of the Queue, AND return its contents; if the Queue is empty, a std::runtime_error should be thrown as before.
- print_queue() should print out the data currently stored in the Queue, on a single line, and print a newline at the end.
- the destructor should deallocate all remaining Nodes.
- We'll return to being lazy again when it comes to copying.

Please place the declaration for the Queue class (as I've written it above), along with the implementation, in the marked area in hw8_q2.cpp.

Your code should make my code in main() work, and print

```
After A, B, and C are pushed, here's what's in the queue: A B C
Now, a call to .pop_front() yields: A
And now the queue contains: B C
Finally, at the end, the contents of the queue are: C D E
```

3. The Standard Template Library has its own stack class, which can accommodate any type of data. To use them,

   - #include <stack>
   - to declare an empty stack, you would write a line like

     std::stack<int> my_int_stack;

     where int could be replaced by any data type you like
   - my_int_stack.push(15); would put the value 15 at the top of the stack
   - my_int_stack.top() would return the value of the top element of the stack, WITHOUT removing it
   - my_int_stack.empty() would return whether or not the stack is empty
   - my_int_stack.size() would return the number of entries in the stack.
   - my_int_stack.pop(); would remove the top value from the stack (but would NOT return the removed value) – this one will cause a segmentation fault if called when the stack is empty!

   See the example in stack_ex.cpp.

   In hw8_q3.cpp, I have several string variables which contain written expressions mathematical expressions which contains pairs of round parentheses () and square brackets []. The first three are *balanced*: every open parentheses has a matching closed parentheses which comes later, the same hold for brackets, and there are not misalignments like ([)] (where the open [occurs within a pair of parentheses, and its matching] occurs outside of those parentheses). The last four are *unbalanced*.

   Write a function called balanced() which receives a string composed of digits, operators, and open/closed parentheses and brackets. The function should return true if the parentheses and brackets contained within are balanced, and false if they are not.

Use a stack. Here's the idea: go through the string character by character; push open parentheses/brackets onto the stack as you encounter them; and pop the stack when you encounter close brackets and parentheses. As you do this, and at the end, you should perform appropriate checks.

Warnings: be careful when popping – the stack has to be non-empty! Also, if you use a loop to loop through the string, I suggest you make your loop test be something like `i < str.size();` rather than `i <= str.size() - 1;`, as the latter doesn't do well when `str` is the empty string.