Recitation 11: Linked List Basics

Topic

- Singly linked lists. No classes!!!

Recitation Instructions

- Here we are using singly linked lists.
  - An empty list is just a null pointer.
  - There are no “dummy” nodes.
- There are three tasks given for this recitation. All are required.
- It would be good if you ask a lab worker to check off each task before proceeding. Up to you.
- For simplicity, we are providing you with test code!!! And corresponding output. (I hope it is right.)
- Always good for you to make additional test cases. You will find a handy function listBuild that can help you. See the comment in the code for how to use it. Especially, note the use of "curly braces", i.e. "{}", in the function call.
- **Turn in a single file rec11.cpp.**

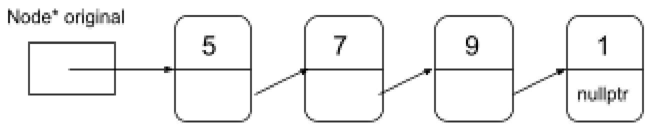
Task One

You will write two functions for this task:

- The first function, `reverse`, will be passed a list and will return a new list that holds all the *values* that are in the original list, but in reverse order. Note that your new list will **consist of all new nodes**. The original list will be unchanged. We will print both lists after calling reverse to be sure.

- The second function, `reverseInPlace`, will not return anything. Instead it will use the nodes in the original list and simply reverse them. **No new nodes** will be created in this function. We will print the list before and after calling reverseInPlace, to see that the original list was changed. Again, this function will **not** create any new nodes.

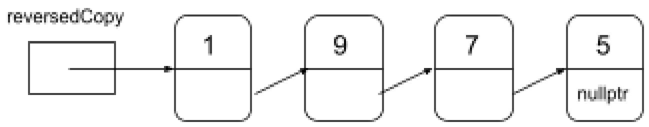
Examples:



Then if we run:

```
Node* reversedCopy = reverse(original);
```

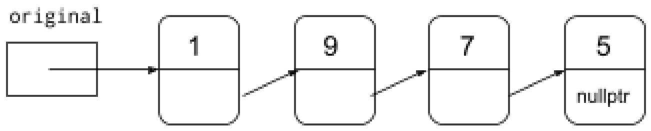
We will have:



If we then run:

```
reverseInPlace(original);
```

We will have:

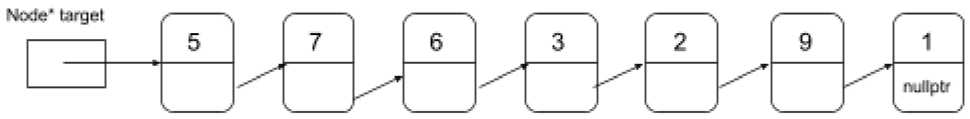


Task Two

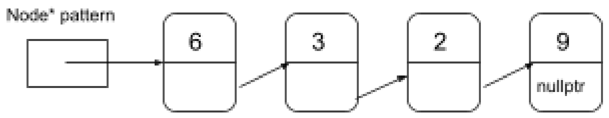
Create and test the function(s) needed for this problem.

Given two lists of ints, is the second list a sublist of the first? E.g.:
<u>Situation: Second list is a sublist of the first.</u>

If the list to be searched is



and the list to be looked for is



the function should return a pointer to the node containing the **6 i**n the list to be searched.

Requirements:

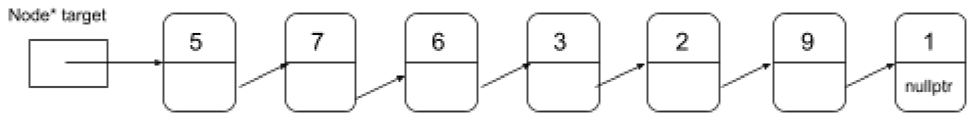
Your function will be named `isSublist`:

- The first argument will be the list you are looking for.

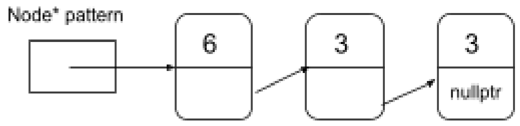
- The second argument will be the list you are looking in.
- The `isSubList` function must return a pointer to the node where the sublist starts in the searched list or nullptr if not found.

Situation: Second list is not a sublist of the first.

If the list to be searched is



and the list to be looked for is



the function should return nullptr.

Some other possibly cases?

- What if there is more than one match of the sublist? Return the start of the first match.
- What should you do if the sublist is empty? Return the first list.

Testing:
- print the lists
- print the list returned by isSubList.

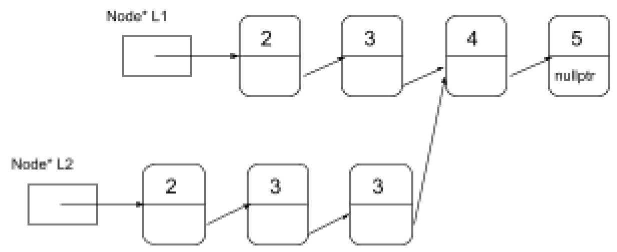
Task Three

In this task you will write two functions, the first, `sharedListBruteForce`, and then, sharedListUsingSet. Both are passed two linked lists.

It is possible that the two lists "share" a sublist. By share, here we mean that actual nodes from one list appear in the other. Naturally these shared nodes can only appear as a suffix, i.e. end, of the two lists. Your goal is to return a pointer to the first node that the two lists share, or to return a nullptr if there is none.

Your test code should then print the shared list.

For example, consider the two lists below. Note that the two lists share the nodes that have the data values 4 and 5. It is not just that they have nodes that hold the same data values, but that they are the same nodes. Your function would return the address of the shared node that is holding the 4, and your test code would print the list starting there.



Without the help of an auxiliary data structure, this might be fairly time consuming. I.e. if the two lists are named L1 and L2, and we denote the length of a list L by |L|, then a brute force approach would cost O(|L1| * |L2|). I would like you to first implement a solution of this form as your first function: `sharedListBruteForce`.

If your two lists were the same length, say n, then your solution with sharedListBruteForce was O(n^2), i.e. quadratic.

Of course, we can do much better using some form of set data structure. C++ provides two of them, one that keeps the items "in order" and the second that does not. (From your data structures course, you hopefully remember how these might be implemented.) This will be your second function: `sharedListUsingSet`.

The class we will use is called `unordered_set`. As when we use the vector class, we will need to include a header file: `#include <unordered_set>;`

Also, as with the vector class, the unordered_set is a "generic" type, meaning that it is defined with the type of thing that it holds. So if we wanted to define an unordered set of ints called "stuff", the definition would look like:

```
unordered_set<int> stuff;
```

unordered_set has three methods that you can use to easily solve this problem:

- `insert`: which takes a single value to be added to the set. If the item is already in there, the item will not be added.
- `find`: which locates a value in the set. We are only concerned with what it returns if the element is not there, which is the next method:
- end. This method takes no arguments. All you need to know for now is that the method find will return the same value as end, if the item passed to find is not present in the set.

Given this data structure, and realizing that find and insert methods will be constant time operations, then you can solve the problem in O(|L1| + |L2|) time.

For those who might be wondering, yes, we could write this without the find method, but you would have to know a little bit more about how data structures like this work in C++, and I don't want to bother you with those details right now. We will cover them shortly. (The code wouldn't actually be particularly shorter.)

Optional

Note that **there is yet another approach**. (No, you don't have to do this one.) It will run in the same time as the second version, but will take *only constant space*. You used O(n) space for the unordered_set.

This third approach is based on the idea that if you know the lengths of the two lists, then you realize that you only have to compare as many nodes as there are in the shorter list. So, just read down the longer list until you get to the point where you have the number of nodes left as are in the shorter list. (Yes, you had to find out the lengths of the two lists first, so there is some trade off of time and space, as usually occurs.) Once you get to the point where you have the same number of nodes in each list to examine, you can walk down the rest of the longer list along with the shorter list, till you get to the point where they are sharing their first node. And then you are done! Neat, eh? Can you code this one up? (Many thanks to Yuxi for pointing this out.)