Recitation 11: Linked List Basics

Topic

- linked lists

Recitation Instructions

There are three tasks given for this recitation. The first two are required. The third need only be done if you have time *during* lab.

It might be good if you ask a lab worker to check off the first task before proceeding to the second, but is not required.

Do, however, have the first two tasks checked before moving on to task three.

For simplicity, please put all tasks in a single program. Make sure that the output *clearly* identifies what is being tested. Consider using something like the format of the Sample Output below. (I don't want to require that format in case you have a better / more readable idea.)

And of course, you should *free up* any nodes created in this exercise. To help out you will see a `listClear` function in the attached file of list functions. In task three you will have to be more resourceful, as the lists may have shared nodes.

Provided Code

We have provided some code for your use that you may have covered in lecture:

- `Node` struct with a constuctor
- `listAddHead`
- `listClear`

We have also provided a function `listBuild` that may be useful with building your test cases. See the comment in the code for how to use it. Especially, note the use of "curly braces", i.e. "{}", in the function call.
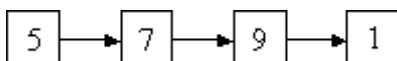
Task One

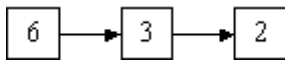Create and test the function(s) needed for this problem.

Splice a singly linked list of `ints` into another list of `ints` given a pointer to the node that you will insert after.
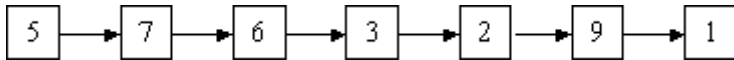
E.g.:

If the original list was

and the list to splice in was

6 → 3 → 2

and if the function is passed a pointer to node containing the 7 in the original list, the resultant list would be

5 → 7 → 6 → 3 → 2 → 9 → 1

Note that the 6 → 3 → 2 list was spliced into the original list after the node containing the 7 .

Requirements:

- Your function will be named **splice**.
  - The first argument will be the list that you are adding.
  - The second argument will be the location where you are splicing the first argument in.
- you must use this attached provided code to work from
- Do **not** create any new Nodes.
- Do **not** define any additional types. You only need the Node struct. In particular you should *not* define a "List" class.
- the splice-into function must be void

Assumptions:

- the function will not attempt to splice before the first node in the original list

Considerations:

- Do not use any other code from the course's sample code
- can we use the code that splices a list between two nodes to splice a list after the last node?

Testing:

- print both lists before the splice and print the original list after splicing.

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.:

Situation: Second list is a sublist of the first.

If the list to be searched is 5 → 7 → 6 → 3 → 2 → 9 → 1
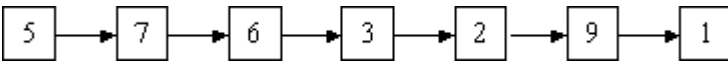
and the list to be looked for is 6 → 3 → 2 → 9

the function should return a pointer to the node 6 in 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
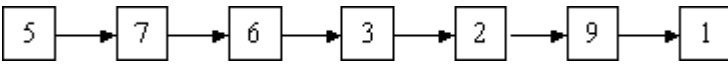
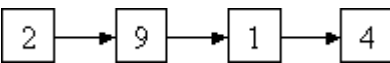Situation: Second list is not a sublist of the first.

If the list to be searched is

| 5 | → | 7 | → | 6 | → | 3 | → | 2 | → | 9 | → | 1 |

and the list to be looked for is

| 6 | → | 2 | → | 3 |

the function should return `nullptr`.

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

If the list to be searched is

| 5 | → | 7 | → | 6 | → | 3 | → | 2 | → | 9 | → | 1 |

and the list to be looked for is

| 2 | → | 9 | → | 1 | → | 4 |

the function should return `nullptr`.

Considerations:

- what if there is more than one match of the sublist

Testing:

- print the lists
- print the list returned by the isSubList function

Sample Output:

```
Part One:

L1: 5 7 9 1
L2: 6 3 2
Target: 7 9 1
Splicing L2 at target in L1
L1: 5 7 6 3 2 9 1
L2: 6 3 2 9 1
=====================

Part two:

Target: 1 2 3 2 3 2 4 5 6

Attempt match: 1
1 2 3 2 3 2 4 5 6

Attempt match: 3 9
Failed to match
```

```
Attempt match: 2 3
2 3 2 3 2 4 5 6

Attempt match: 2 4 5 6
2 4 5 6

Attempt match: 2 3 2 4
2 3 2 4 5 6

Attempt match: 5 6 7
Failed to match

Attempt match: 6
6
```

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, we mean that actual nodes from one list appear in the other. Naturally these shared nodes *can only appear* as a *suffix* 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.
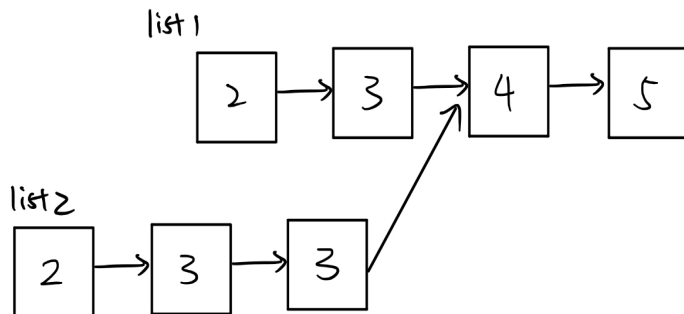
Without the help of an auxiliary data structure, this might be fairly time consuming. I.e. if the two lists are named L1 and L2, and we denote the length of a list L by |L|, then a brute force approach would cost $O(|L1| * |L2|)$. I would like you to first implement a solution of this form as your first 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 are likely to be implemented.) This will be your second function: `sharedListUsingSet`.

The class we will use is called `unordered_set`. Like 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 easily use to 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 methods 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 expected time O(1) 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. (It wouldn't be particularly shorter.)

Note that there is yet another approach. It will run in the same time as the second version, but will take ony *constant* space. You used O(n) space for the `unordered_set`.

This approach is based on the idea that if you know the lengths of the two lists, then you know 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 there first node. And then you are done! Neat, eh? Can you code this one up? (Many thanks to Yuxi for pointing this out.)