Quiz #2: Linked Lists and Recursion ICS-46 Fall 2016

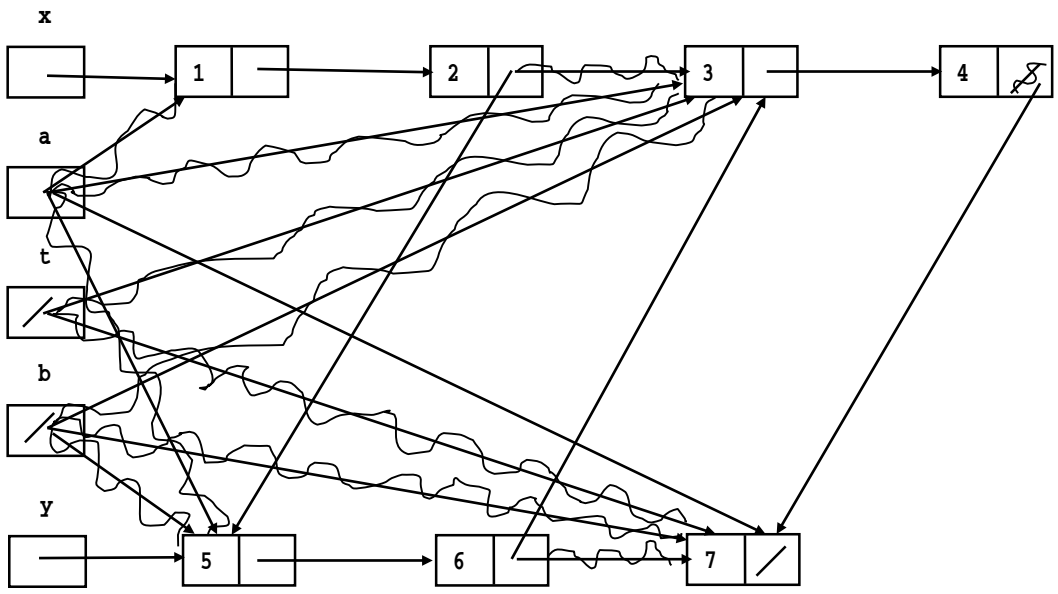
Name _____

When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download/unzip the **`q2helper`** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **`q2solution.hpp`** file online by Thursday, 11:30pm; submit this sheet at the **start of class** on Friday. I will post my solutions to EEE reachable via the **Solutions** link on Friday afternoon.

Write your answer to Problem 1 on this sheet. Print just the first page of this document and submit it in class.

1. (7 pts) Examine the **`mystery`** method and hand simulate the call **`mystery(x,y);`** using the linked list below. Use the LN class from the lecture notes. **Cross out** ALL references that are replaced and **Write in** new references: don't erase any references. It will look a bit messy, but be as neat as you can. Probably it is best to do this problem first on another sheet of paper, , then copy EVERYTHING here. **If an assignment doesn't change a pointer (e.g., x = x;) , just leave it as is: don't cross it out and redraw it.**

```
void mystery(LN<int>* a, LN<int>* b) {
  while (b != nullptr) {
    LN<int>* t = a->next->next;
    a->next->next = b;
    a = b;
    b = t;
  }
}
```



2. (6 pts) Define a **recursive** function named `relation`; it is passed two `std::string` arguments; it returns one `char` value: `'<'`, `'='`, or `'>'` indicating the relationship between the first and second parameters. Hint: my code used no local variables other than the parameters, compared strings only to the empty string, and compared only one character in one string to one character in the other. You may use C++relational operators **only** to compare **single characters**, otherwise this function would be trivial to write.

Given the standard templated declaration for LN:

```
template<class T>
class LN {
  public:
    LN ()                          : next(nullptr){}
    LN (const LN<T>& ln)           : value(ln->value), next(ln->next){}
    LN (T v, LN<T>* n = nullptr)   : value(v), next(n){}
    T      value;
    LN<T>* next;
};
```

The `remove_ascending` functions are `void` and have one parameter: a linked list of integers whose values appear in any order (and may be repeated). Whenever two adjacent values are ascending, the first value is removed from the linked list. For example, if `l` is a linked list printing as `5->7->4->8->nullptr`; and we call `remove_ascending(l);` then printing the linked list `l` produces `7->8->nullptr`. The `5` is removed (it is smaller than `7`), then the `7` is not removed (it is not smaller than `4`), then the `4` is removed (it is smaller than `8`), and there is nothing after `8` to compare. If `l` were `5->3->2->7->8->4->2->nullptr` the result would be `5->3->8->4->2->nullptr`.

The driver/Googletests call these functions for duplicate ascending values at the front/middle/rear of the list, including duplicates in all those locations.

3. (6 points) Write the `remove_ascending_i` function (`remove_ascending` **iteratively)**. Its parameter should be a reference to a pointer.

There are many ways to implement this function iteratively (including using a local variable that is a pointer to a pointer). I wrote several iterative solutions that were as large as 12 lines and as small as 7 lines (well, formatted, not counting lines with just a closing brace).

4. (6 points) Write the `remove_ascending_r` function (`remove_ascending` **recursively)**. Its parameter should be a reference to a pointer.

You might want to try writing this function first, because it can be written very clearly. I wrote the recursive solution in 10 lines (well formatted, not counting lines with just a closing brace).