Overview

- Monday, May 9th 3pm-6pm, Darrin 308,

- Coverage: cumulative, but with substantial emphasis on material covered since the second exam.

- Length: about the same as earlier exams, but we’ll give you an extra 40 minutes.

- Closed-book, closed-notes, but Appendix B of the Koenig and Moo test will be available as a handout. This will be augmented with the `cs2list<T>` class declaration and the stack and queue class public member function declarations.

Important Topics

- Everything from before Test 3. Especially the following:
  - Iterators and indices
  - Order notation
  - Lists, vectors, strings, and maps
  - Classes, including operators, constructors, and destructors
  - Problem solving
  - Program design and choosing containers
  - Recursion
  - Dynamic memory
  - Tests 1, 2 and 3
  - Stacks and queues
  - Our own versions of the vector and string classes.

- Linked lists:
  - Basic structure of linked objects
  - Singly-linked lists: stepping through, insert, remove
  - Doubly-linked lists
Trees:
- Binary trees and binary search trees
- Node class
- Traversal
- Algorithms: finding the largest and smallest value, counting the nodes, height.
- Insert, find and copy.
- `cs2set<T>` class, with its member functions

Practice Problems

After reviewing the lecture notes and texts based on the outline above, you should do practice problems. Work on problems from the notes, from lab, from homework, and from the earlier Below are some additional practice problems covering material after Test 3. Do not consider this list complete. You might even make-up specific examples of the types problems that you think may be on the test and work them yourself.

1. Write a function to create a new singly-linked list that is a COPY of a sublist of an existing list. The prototype is

   ```cpp
   Node<T>* Sublist( Node<T>* head, int low, int high )
   ```

   The `Node` class is:

   ```cpp
   template <class T>
   class Node {
   public:
     T value;
     Node* next;
   }
   ```

   The new list will contain high-low+1 nodes, which are copies of the values in the nodes occupying positions low up through and including high of the list pointed to by head. The function should return the pointer to the first node in the new list. For example, in the following
drawing the original list is shown on top and the new list created by the function when \texttt{low==2} and \texttt{high==4} is shown below.

Original list

\begin{itemize}
  \item head \[3.1 \rightarrow 2.4 \rightarrow 8.7 \rightarrow 9.4 \rightarrow 14.2 \rightarrow 0.9\]
\end{itemize}

New list

\begin{itemize}
  \item nhead \[2.4 \rightarrow 8.7 \rightarrow 9.4\]
\end{itemize}

A pointer to the first node of this new list should be returned. (In the drawing this would be the value of \texttt{nhead}.) You may assume the original list contains at least \texttt{low} nodes. If it contains fewer than \texttt{high} nodes, then stop copying at the end of the original list.

2. Suppose that a monster is holding you captive on a computational desert island, and has a large file containing double precision numbers that he needs to have sorted. If you write correct code to sort his numbers he will release you and when you return home will be allowed to move on to DSA. If you don’t write correct code, he will eventually release you, but only under the condition that you retake CS 1. The stakes indeed are high, but you are quietly confident — you know about the standard library sort function. (Remember, you are supposed to have forgotten all about bubble sort.) The monster startles you by reminding you that this is a computational desert island and because of this the only data structure you have to work with is a queue.

After panicking a bit (or a lot), you calm down and think about the problem. You realize that if you maintain the values in the queue in increasing order, and insert each value into the queue one at a time, then you can solve the rest of the problem easily. Therefore, you must write a function that takes a new double, stored in \texttt{x}, and stores it in the queue. Before the function is called, the values in the queue are in increasing order. After the function ends, the values in the queue must also be in increasing order, but the new value must also be among them.
Here is the function prototype.

```c
void insert_in_order( double x, queue<double>& q )
```

You may only use the public queue interface (member functions) as specified in lab. You may use a second queue as local variable scratch space or you may try to do it in a single queue (which is a bit harder). Give an “O” estimate of the number of operations required by this function.

3. Write a `cs2list<T>` member function called `reverse` that reverses the order of the nodes in the list. The head pointer should point to what was the tail node and the tail pointer should point to what was the head node. All directions of pointers should be reversed. The function prototype is

```c
template <class T>
void cs2list<T>::reverse();
```

The function must NOT create ANY new nodes.

4. Write a `cs2list<T>` member function called `splice` that takes an iterator and a second `cs2list<T>` object and splices the entire contents of the second list between the node pointed to by the iterator and its successor node. The second list must be completely empty afterwards. The function prototype is

```c
template <class T>
void cs2list<T>::splice( iterator itr, cs2list<T>& second );
```

No new nodes should be created by this function AND it should work in $O(1)$ time (i.e. it should be independent of the size of either list).

5. For this question and the next few, consider the following tree node class

```c
template <class T>
class TreeNode {
public:
    TreeNode() : left(0), right(0) {}
    TreeNode(const T& init) : value(init), left(0), right(0) {}
```
T value;
TreeNode* left;
TreeNode* right;

Write a function to find the largest value stored in a binary search tree of integers pointed to by TreeNode<int>* root. Write both recursive and non-recursive versions.

6. Write a recursive function to count the number of nodes stored in the binary tree pointed to by TreeNode<T>* root.

7. Write a new member function of the cs2set<T> class called to_vector that copies all values from the binary search tree implementation of the set into a vector. The resulting vector should be increasing order. You may assume the vector is empty at the start. The function prototype should be

   template <class T>
   void cs2set<T>::to_vector( vector<T>& vec );

8. Write a function called Trim that removes all leaf nodes from a tree, but otherwise retains the structure of the tree. Hint: look carefully at the way the pointers are passed in the insert and erase functions.

9. (Challenge) Write a constructor for a cs2set<T> object that builds the tree underlying the set from a vector that is increasing order. Try to do so as efficiently as possible (i.e. without using insert). The prototype is

   template class<T>
   cs2set<T>::cs2set<T>( vector<T> const& v );