Computer Science II — CSci 1200 — Sections 6-10
Final Review

Overview

- Monday, May 6, 3-6pm Amory Mids
- Coverage: cumulative, but with emphasis on material covered since the second exam.
- Length: about the same as earlier exams
- Closed-book, closed-notes, but the same handout will be available as for Test 2. This will be augmented with the \texttt{cs2list<T>} class declaration and the stack and queue class public member function declarations.

Important Topics

- Everything from before Test 2. Especially the following:
  - Iterators and indices
  - The generic find function
  - Maps
  - Classes, including operators, constructors, and destructors
  - Program design and choosing containers
  - Tests 1 and 2

  Remember that one question from Test 2 will be repeated on the final. Understanding difficulties that you had in other questions will also help on the final.

- Recursion:
  - How recursion works
  - Writing recursive functions.
  - Examples: binary search, merge sort (especially important), word search

- Order notation:
– Intuition
– The meaning of different orders: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, etc.
– Analyzing (relatively) simple algorithms: loops, nested loops, etc.

• Linked lists:
  – Basic structure of linked objects
  – Singly-linked lists: stepping through, insert, remove
  – Doubly-linked lists
  – cs2list<T> class, with its member functions and iterators

• Stacks and queues:
  – basic operations
  – stacks are LIFO, queues are FIFO — think about what this means in terms of computation

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, from projects, and from the first two tests. Below are some additional practice problems covering material after Test 2. 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 recursive function to multiply two positive integers using only addition, subtraction and comparison operations. No loops are allowed in the function. The function prototype should be

   int Multiply( int m, int n)

2. Consider the mergesort function discussed in detail in lecture (the code is posted on the web and would be provided for you if this question appear on an exam). Suppose the vector passed to mergesort has 7 items in it. Specify the EXACT set of function calls that are made and the exact order that they are made. In specifying this, you do not need to show the contents of the vector, just the values of low and high (and mid, for calls to merge).
3. Give an “O” estimate of the number of operations required by \texttt{merge}?
Let \( n = \text{high} - \text{low} + 1 \).

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

\[
\text{Node<T}^{*} \text{ Sublist( Node<T}^{*} \text{ head, int low, int high })}
\]

The \texttt{Node} class is:

\[
\begin{array}{l}
\text{ template <class T> class Node { } } \\
\text{ public: } \\
\text{ \hspace{1cm} T value; } \\
\text{ \hspace{1cm} Node* next; } \\
\text{ }); }
\end{array}
\]

The new list will contain \( \text{high-low+1} \) nodes, which are copies of the values in the nodes occupying positions \( \text{low} \) up through and including \( \text{high} \) of the list pointed to by \( \text{head} \). 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 \( \text{low}==1 \) and \( \text{high}==3 \) is shown below.

<table>
<thead>
<tr>
<th>Original list</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
</tr>
<tr>
<td>3.1 2.4 8.7 9.4 14.2 0.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New list</th>
</tr>
</thead>
<tbody>
<tr>
<td>nhead</td>
</tr>
<tr>
<td>2.4 8.7 9.4</td>
</tr>
</tbody>
</table>

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 \( \text{low} \) nodes. If it contains fewer than \( \text{high} \) nodes, then stop copying at the end of the original list.
5. 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 you 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 settle down and think about the problem. You realize that if you maintain the values in the queue in increasing order, and insert each new 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 \( x \), 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.

\[
\text{void insert\_in\_order( int x, queue<double>& q )}
\]

You may only use the public queue interface (member functions) as specified in the lecture notes. 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.

6. Write a function that takes a \texttt{queue<int>} and determines if it is a palindrome. The only other container you can use is a \texttt{stack<int>}.

You will need to be careful of the possibility that the queue has an odd number of values.

7. Write a \texttt{cs2list<T>} member function called \texttt{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
template <class T>
void cs2list<T>::reverse();

The function must NOT create ANY new nodes.

8. Write a cslist<T> member function called splice that takes an iterator and a second cslist<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

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).