CSCI-1200 Computer Science II — Spring 2006
Final Exam Practice Problem Solutions

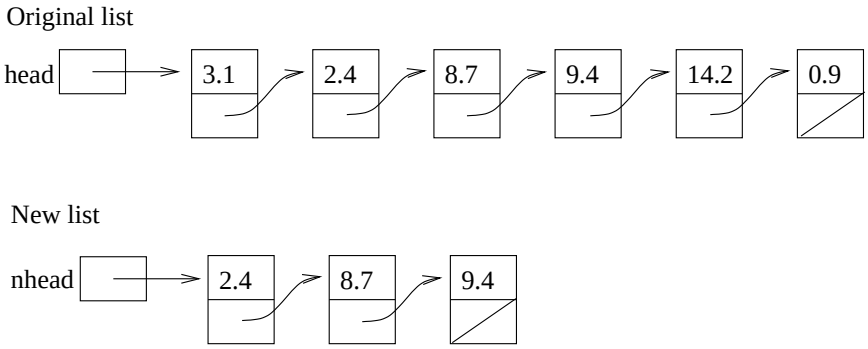
1. Write a function to create a new singly-linked list that is a *copy* of a sublist of an existing list. The prototype is:

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

The `Node` class is:

```
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 `low==2` and `high==4` is shown below.



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

Solution:

```
Node<T>* Sublist(Node<T>* head, int low, int high) {
  // Skip over the first low-1 nodes in the existing list
  Node<T>*p = head;
  int i;
  for (i=1; i<low; ++i) p = p-> next;
  // Make the new head node and make a pointer to the last node in list
  Node<T>* new_head = new Node<T>;
  new_head->value = p->value;
  Node<T> * last = new_head;
  // Copy the remaining nodes, one at a time
  for (++i, p = p->next; i<=high && p; ++i, p = p->next) {
    last->next = new Node<T>;
    last->next->value = p->value;
    last = last->next;
  }
  last->next = 0;
}
```

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 `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:

```
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.

   Solution: Here is the version with a scratch queue

```
void insert_in_order(double x, queue<double>& q) {
  if (q.empty())
    q.push(x);
  else {
    queue<double> temp(q);           // copy q;
    while (!q.empty()) q.pop();      // empty q out
    while (!temp.empty() && x > temp.front()) {
      double item = temp.front();
      temp.pop();
      q.push(item);
    }
    q.push(x);    // insert x in its proper position
    while (!temp.empty()) {
      double item = temp.front()
      temp.pop();
      q.push(item);
    }
  }
}
```

   This function requires $O(n)$ operations. Copying the queue initially and emptying the queue requires $O(n)$ time each. The second and third while loops, combined, touch each entry in the queue and therefore require $O(n)$ operations. Since none of these loops are nested we add the results and get $O(n)$ time overall.

Here is a version without a scratch queue:

```
void insert_in_order(int x, queue<double>& q) {
  int n = q.size();
  int position = 0;
  // Find the position for x in the queue, copying all values
  // less than x to the back of the queue
  while (position < n && x < q.front()) {
    q.push(q.front());      // copy the front to the back
    q.pop();                // remove the front
    ++position;
  }
  q.push(x);                // put x in position

  // The first n-position entries on the queue haven't been
  // touched and are greater than or equal to the value
  // stored in x.  They need to move to the back of the queue
  int i = position;
  while (i < n) {
    q.push(q.front());      // copy the front to the back
    q.pop();                // remove the front
    ++i;
  }
}
```

The two while loops, combined touch each entry in the queue once and therefore require $O(n)$ operations. Since none of these loops are nested we add the results and get $O(n)$ time overall.

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:

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

The function must NOT create ANY new nodes.

Solution:

```
template <class T>
void cs2list<T>::reverse() {
  // Handle empty or single node list
  if (head_ == tail_) return;
  // Swap pointers at each node of the list, using a temporary
  // pointer q to remember where to go next.
  Node<T>* p = head_;
  while (p) {
    Node<T>*q = p->next_;
    p->next_ = p->prev_;
    p->prev_ = q;
    p = q;
  }
  // Swap head and tail pointers
  p = head_;
  head_ = tail_;
  tail_ = p;
}
```

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:

```
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).

Solution:

```
template <class T>
void cs2list<T>::splice(iterator itr, cs2list<T>& second) {
  if (second.empty()) return;
  second.head_->prev_ = itr.ptr_;
  second.tail_->next_ = itr.ptr_->next_;
  if (itr.ptr_->next_) {
    itr.ptr_->next_->prev_ = second.tail_;
  } else { // itr.ptr_ is the tail, so it must be reset
    this->tail_ = second.tail_;
  }
  itr.ptr_->next_ = second.head_;
  this->size_ += second.size_;
  second .size_ = 0;
  second.head_ = second.tail_ = 0;
}
```

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

```
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.

Recursive Solution:

```
int FindLargest(TreeNode<int>* root) {
  if (! root->right)
    return root->value;
  else
    return FindLargest(root->right)
}
```

Non-recursive Solution:

```
int FindLargest(TreeNode<int>* root) {
  while (root->right)
    root = root->right;
  return root->value;
}
```

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

   Solution:

   ```
   int Count(TreeNode<T>* root) {
     if (! root)
       return 0;
     else
       return 1 + Count(root->left) + Count(root->right);
   }
   ```

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);
   ```

   Solution:

   ```
   template <class T>
   void cs2set<T>::to_vector(vector<T>& vec) {
     to_vector(this->root_, vec);
   }

   template <class T>
   void cs2set<T>::to_vector(TreeNode<T>* p, vector<T>& vec) {
     if (p) {
       to_vector(p->left, vec);
       vec.push_back(p->value);
       to_vector(p->right, 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.

   Solution:

   ```
   template <class T>
   void Trim(TreeNode<T> *& root) { // Passing by reference is crucial here
     if (root) { // Only do something for non-empty trees
       if (!root->left && !root->right) { // Leaf
         delete root;
         root = 0;     // This sets the appropriate pointer in the parent node to 0.
       } else {
         Trim(root->left);
         Trim(root->right);
       }
     }
   }
   ```

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(vector<T> const& v);
```

Solution: The solution has the structure of mergesort, without the merge step: the vector is split into approximately even-sized subintervals, each becomes a subtree. For any interval, the value in the center is placed at the root (of the subtree) and the lower and upper subintervals become the left and right subtrees. The bulk of the work is done in `build_from_vector`.

```
template class<T> cs2set<T>::cs2set(vector<T> const& v) {
  root_ = this->build_from_vector(0, v.size(), v);
}

template class<T>
TreeNode<T>* cs2set<T>::build_from_vector(int low, int high, vector<T> const& v) {
  if (low > high)  // No more values, so return a null pointer.
    return 0;
  else {
    // Form a root (of the subtree) from the middle value and make
    // left and right subtrees from the left and right subintervals.
    int mid = (low+high)/2;
    TreeNode<T>* p = new TreeNode<T>(v[mid]);
    p->left = build_from_vector(low, mid-1, v);
    p->right = build_from_vector(mid+1, high, v);
    return p;
  }
}
```