Section Solutions #7

Problem 1: Client-side vs. Implementation-side

a) As the implementer, we can access our own state

```
template <typename Type>
  void Stack<Type>::reverse() {
    for (int lh = 0, rh = elems.size()-1; lh < rh; rh--, lh++)
    {
      Swap(elems[lh], elems[rh]);
    }
  }
```

b) As a client, we can't access **elems**, so we have to try something else.

```
template <typename Type>
  void reverse(Stack<Type> &s) {
    Stack<Type> t;
    while (!s.isEmpty())
    {
      t.push(s.pop());
    }
    s = t;
}
```

Problem 2: Template Class Conversion

(Mob.h):

```
template <typename ElemType>
class Mob {
      public:
            Mob();
            void enqueue(ElemType newElem);
            ElemType dequeue();
            int size();
      private:
            Vector<ElemType> elems;
};
```

(Mob.cpp):

```
template <typename ElemType>
void Mob<ElemType>::enqueue(ElemType newElem)
{
      elems.add(newElem);
}

template <typename ElemType>
ElemType Mob<ElemType>::dequeue()
{
      int elemNum = RandomInteger(0, elems.size()-1);
      ElemType value = elems[elemNum];
```

```
        elems.removeAt(elemNum);
        return value;
}
```

In addition to the changes in the code, the following changes need to be made. Previously, both .h and .cpp files were included in the project, and the .cpp file #included the .h file but not the other way around. Now the .cpp file is not included in the prject, and the .h file #includes the .cpp file at the end of the .h file.

Problem 3: Big-O Detective

a) The general strategy here is to call the contains function on Sets of different sizes and try to use this resulting timing information to determine the Big-O. For example, how does the time change if you double the size? If it is O(N), this should double the time, if it is O(N^2), it should quadruple the time, and so on. Calling contains for elements not in the Set is better for this purpose, since it will almost certainly be worst-case. Asking about an element that is in the Set, on the other hand, might take very little time if it is examined early on in the search. However, you should still make sure to run the test a number of times on each size so that you average out any discrepencies.

b) If you use a callback function for comparisons, you can count every time there is a comparison. However, to do this you have to have access to a persistent variable. You can do this by making a global variable that you increment every time the callback function is called. While in general global variables are poor style, here there is no other way to accomplish what we want since we can't pass information out any other way.

Problem 4: Deques

The easiest way to build a deque is to use a doubly linked list, which makes inserting and deleting from both ends very simple. With a singly linked list, whenever we wanted to insert or remove at the opposite end we would have to traverse the entire list, and keeping an additional pointer is worth reducing these times to O(1). As for the functions, all we really need are push/pop from the front/back (four functions in total), plus a constructor, a destructor, isEmpty, and size. We can use a deque to simulate a stack by only using the Push/Pop from the front, and we can simulate a queue by popping from the front and pushing on the back.

Problem 5: Stutter

```
/**
 * Function: Stutter
 * Usage: Stutter(list)
 * ------------------------
 * Duplicates every cell in a linked list
 */
void Stutter(Node* list)
{
        for (Node* curr = list; curr != NULL; curr = curr->next)
        {
                Node* stuttered = new Node;
                stuttered->value = curr->value;
                stuttered->next = curr->next;
                curr->next = stuttered;
                curr = stuttered;
```

```
	}
}
```

Problem 6: Unstutter

```
/**
 * Function: Unstutter
 * Usage: Unstutter (list)
 * -----------------------
 * Removes any neighboring duplicates found in the passed in linked
 * list.
 */
void Unstutter(Node *list)
{
	for(Node *cur = list; cur != NULL; cur = cur->next)
	{
		if (cur->next != NULL && cur->value == cur->next->value)
		{
			// match!
			Node *duplicate = cur->next;
			cur->next = cur->next->next; // splice out duplicate
			delete duplicate;
		}
	}
}
```