CS241
Homework 2: Vectors, Lists, Stacks, Queues, Sorting

1. Recall that project 2 involved writing a CoolVector class with lots of extra methods. Suppose we want to add a method to the CoolVector class called `duplicate` that takes an argument called `times`. This function creates a new CoolVector that consists of all the elements of the original CoolVector duplicated in order the number of times requested.

   Example of use:
   ```
   CoolVector<int> coolvec;
   coolvec.append(1);
   coolvec.append(2);
   coolvec.append(3);
   CoolVector<int> coolvec2 = coolvec.duplicate(3);
   // coolvec2 is now[1 2 3 1 2 3 1 2 3]
   ```

   Write C++ code for this function. Your code should be as efficient as reasonably possible (don't re-allocate or copy the CoolVector more than necessary, don't loop over the CoolVector more than necessary, etc). Do not call any other functions you wrote for the project; write all of your code inside this function. In other words, don't call `insert` or append or anything like that.

   Here's the method signature:

   ```
   CoolVector CoolVector::duplicate(int times)
   ```

   You may assume that `times` is an integer >= 0.

2. Suppose we have a singly-linked list class set up like this:

   ```
   struct node {
     int data;
     node * next;
   };

   class SList {
     private:

     node * head;
   }
   ```

   Write a member function for `SList` called `removeMultiple(int start, int howmany)` that removes "howmany" elements from the list, starting at the "`start`" element (indexed from zero, like arrays/vectors). So if we had a linked list called `lst`, then `lst.removeMultiple(2, 3)` would remove items at positions 2, 3, and 4 from the linked list.

   Your code should be as efficient as possible (within reason), which means only traversing the list once to find the right spot to start removing elements, and not linking any node pointers that will just immediately be unlinked. Do not call any other functions from `removeMultiple` (in other words, don't rely on the existence of a `removeOne()` function or something like that).

   Here's the method signature:

   ```
   void SList::removeMultiple(int start, int howmany)
   ```

You may assume `start` and howmany have legal integer values (no error checking is needed in your code).

3. Assume we have an empty stack that stores single characters. For the following sequence of stack operations, **list the order that the characters are popped off the stack**. Also, for every operation marked with a star (*), **draw the stack after that operation has completed** (you may draw the stack vertically or horizontally, just make it clear where the top & bottom of the stack are.)

   Push A
   Push B
   Push C
   Push D *
   Pop *
   Push E *
   Pop
   Pop *
   Push F *
   Pop
   Pop
   Push G *
   Pop
   Pop

4. Assume we have an empty queue that stores single characters. Run the same sequence of operations as the previous stack question, but change all the pushes to enqueues and all the pops to dequeues. **Show the queue after each operation marked with a star** (draw the queue horizontally with the front at the left and back at the right).

5. Recall our discussion of postfix arithmetic expressions in class: these are mathematical expressions where the operator (like +, -, *, /) comes after the operands (numbers), rather than before. So instead of writing "3 + 4", you'd write "3 4 +."

   Recall the algorithm for evaluating a postfix expression which involves a stack:

   > Make a new, empty stack
   > Examine the expression from left to right, and for each item in the expression:
   > if the item is an operand (a number), push it onto the stack
   > if the item is an operator, pop the stack twice to get the two operands, perform the desired operation on the two operands, and push the result back onto the stack
   > There should be one operand left on the stack after examining the entire staring expression, and this operand is guaranteed to be the answer.

   For example, running this algorithm on the expression 3 4 + 2 – results in the following:

Actions	Current stack
Push 3	3
Push 4	3 4
See +, Pop 4, Pop 3, Evaluate 3+4, Push 7	7
Push 2	7 2
See -, Pop 2, Pop 7, Evaluate 7-2, Push 5	5

   So 5 is the answer.

Run this algorithm on the following postfix expression: `3 4 2 / + 5 3 * 6 - * 8 -`

Show the stack of operands as you are running the algorithm.

Important: With operators like – and / where the result depends on order, the first number you pop off the stack becomes the 2nd operator, and the second number you pop off becomes the 1st operator. In other words, if you are evaluating the expression "10 3 –", you push 10, then push 3 (so 3 is at the top of the stack), then when you see the minus, you pop 3, then pop 10, but the evaluation becomes 10-3, not 3-10.

6. Suppose we have the following array of six numbers:

   3 2 4 5 1

   Recall that each of the quadratic sorting algorithms (selection sort, bubble sort, insertion sort) has an inner loop and an outer loop. For each of the algorithms, run the algorithm on the starting array above, and *show the state of the array after each iteration of the outer loop* as the array is being sorted. Even if the

   For instance, for selection sort, you would begin with the following:

   Starting array: 3 2 4 5 1
   After iteration #1: 1 2 4 5 3
   [more lines here]

7. In class we studied the properties of selection sort, bubble sort, and insertion sort for various starting configurations of elements in the array. For instance, we examined how the algorithms would operate if given an array that was already sorted or an array sorted in reverse order.

   Suppose we want to sort an array of length *n* that has the following starting configuration: it always begins with a sequence of *n*-1 sorted numbers, but then ends with one that is less than all the previous numbers. For instance, an array that looks like this would be [2 3 4 5 1].

   Describe the performance of selection sort, bubble sort, and insertion sort on an array of this type. Tell me explicitly, in terms of big-oh, the overall running time, and then break this down into the big-oh for the number of comparisons each algorithm makes, and the number of swaps each algorithm makes. Explain your reasoning for each part (this will get you partial credit of your big-oh times are wrong).