CS106B Supplement to Section 4 (Week 5)

ListNode structure

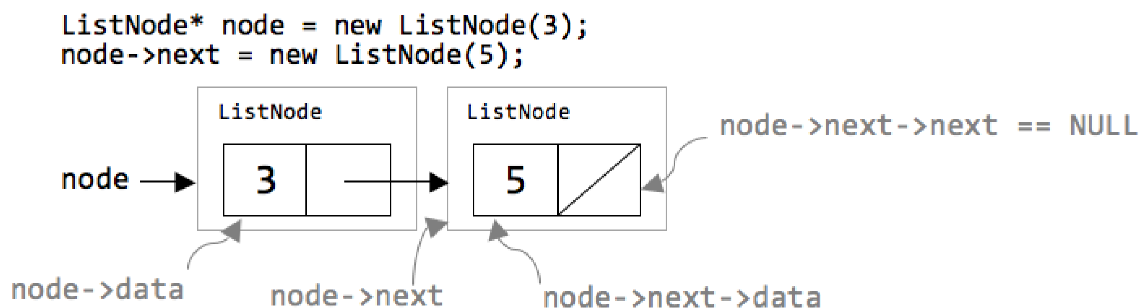
(represents a single data value in a linked list, and a link to the next node)

```
struct ListNode {
    int data;          // data stored in this node
    ListNode* next;    // a link to the next node in the list

    // Constructs a node with the given data and a NULL next link.
    ListNode(int data) {
        this->data = data;
        this->next = NULL;
    }

    // Constructs a node with the given data and the given next link.
    ListNode(int data, ListNode* next) {
        this->data = data;
        this->next = next;
    }
};
```

Here is a diagram of two ListNodes that result from running the two lines of code below. Notice what the different arrows point to (whether it is the object instances of ListNode or the data inside).



LinkedList class

(represents a chain of many list nodes, keeping a pointer to the front node only)

```
class LinkedList {
public:
    void add(int value);
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    int size() const;
    string toString() const;
    ...

private:
    ListNode* m_front;   // NULL if list is empty
};
```

CS106B Supplement to Section 4 (Week 5)

1. Linked nodes (1).

Draw a picture of what the given nodes would look like after the code executes.

Before / Code | **After**

a)

list → [1] → [2 /]

```
list->next = new ListNode(3);
```

b)

list → [1] → [2 /]

```
list->next = new ListNode(3, list->next);
```

c)

list → [1] → [2] → [3 /]

```
list = new ListNode(4, list->next->next);
```

d)

list → [1] → [2] → [3 /]

```
list->next->next = NULL;
```

2. Linked nodes (2).

Write the code that will produce the given "after" result from the given "before" starting point by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but do NOT change any existing node's data field value. If a variable does not appear in the "after" picture, it doesn't matter what value it has after the changes are made.

	Before	After
a)	list → [1] → [2 /]	list → [1] → [2] → [3 /]
b)	list → [1] → [2 /]	list → [3] → [1] → [2 /]
c)	list → [1] → [2 /] temp → [3] → [4 /]	list → [1] → [3] → [4] → [2 /]
d)	list → [1] → [2 /] temp → [3] → [4 /]	list → [1] → [3] → [2] → [4 /]
e)	list → [1] → [2] → [3 /]	list → [2 /] list2 → [1] → [3 /]
f)	list → [5] → [4] → [3 /]	list → [3] → [4] → [5 /]
g)	list → [5] → [4] → [3 /]	list → [3] → [5 /] list2 → [4] → [3] → [5 /]

CS106B Supplement to Section 4 (Week 5)

Each of the following problems asks you to add a member function to the `LinkedList` class from lecture. In all cases, if your function deletes a node from the list, free the associated memory for the node. Declare member functions as const if appropriate, if they do not modify the state of the linked list. Generally, you should try to do these problems without calling other member functions.

---

3. min.

Write a member function **min** that returns the minimum value in a linked list of integers. If the list is empty, it should throw a string exception.

---

4. isSorted.

Write a member function **isSorted** that returns true if the list is in sorted (nondecreasing) order and returns false otherwise. An empty list is considered to be sorted. Bonus: solve this problem both recursively and non-recursively. Which solution do you like better?

---

5. countDuplicates.

Write a member function **countDuplicates** that returns the number of duplicates in a sorted list. The list will be in sorted order, so all of the duplicates will be grouped together. For example, if a variable list stores the sequence of values below, the call should return 7 because there are 2 duplicates of 1, 1 duplicate of 3, 1 duplicate of 15, 2 duplicates of 23 and 1 dupe of 40:

```
{1, 1, 1, 3, 3, 6, 9, 15, 15, 23, 23, 23, 40, 40}
```

---

6. stutter.

Write a member function **stutter** that doubles the size of a list by replacing every integer with two of that integer. For example, if a variable list stores {1, 8, 19, 4, 17}, afterward a call to this method, it should store {1, 1, 8, 8, 19, 19, 4, 4, 17, 17}.

---

7. deleteBack.

Write a member function **deleteBack** that deletes the last value (the value at the back of the list) and returns the deleted value. If the list is empty, your method should throw a string exception.

---

8. split.

Write a member function split that rearranges the elements of a list so that all negative values appear before all of the non-negatives. For example, suppose a variable list stores the following sequence of values:

```
{8, 7, -4, 19, 0, 43, -8, -7, 2}
```

One possible arrangement (but certainly not the only one) after a call to this method would be:

```
{-4, -8, -7, 8, 7, 19, 0, 43, 2}
```

Do not swap data fields or create any new nodes to solve this problem; you must rearrange the list by rearranging the links of the list. You also may not use auxiliary structures like arrays, ArrayLists, stacks, queues, etc, to solve this problem.

CS106B Supplement to Section 4 (Week 5)

9. removeAll.

Write a member function removeAll that takes an integer and removes all occurrences of that value. For example, if a variable list contains the following values:

```
{3, 9, 4, 2, 3, 8, 17, 4, 3, 18}
```

The call of list.removeAll(3); would remove all occurrences of the value 3 from the list, yielding the following values:

```
{9, 4, 2, 8, 17, 4, 18}
```

If the list is empty or the value doesn't appear in the list at all, then the list should not be changed by your method. You must preserve the original order of the elements of the list.

10. doubleList.

Write a member function doubleList that doubles the size of a list by appending a copy of the original sequence to the end of the list. For example, if a variable list stores this sequence of values:

```
{1, 3, 2, 7}
```

and then we call this method, it should store the following values after the call:

```
{1, 3, 2, 7, 1, 3, 2, 7}
```

If the original list contains N nodes, then you should construct exactly N nodes to be added. You may not use any auxiliary data structures to solve this problem (no array, Vector, stack, queue, string, etc). Your method should run in O(N) time where N is the number of nodes in the list.

11. rotate.

Write a member function rotate that moves the value at the front of a list of integers to the end of the list. For example, if a variable called list stores the following sequence of values:

```
{8, 23, 19, 7, 45, 98, 102, 4}
```

The call of this method should move the value 8 from the front of the list to the back of the list, yielding this sequence of values:

```
{23, 19, 7, 45, 98, 102, 4, 8}
```

The other values in the list should retain the same order as in the original list. If the method is called for a list of 0 or 1 elements it should have no effect on the list. You must solve the problem by rearranging the links of the list, so do not construct any new nodes to solve this problem or change any of the integer values stored in the nodes.

12. reverse.

Write a member function reverse that reverses the order of the elements in the list. For example, if the variable list initially stores this sequence of integers:

```
{1, 8, 19, 4, 17}
```

It should store the following sequence of integers after reverse is called:

```
{17, 4, 19, 8, 1}
```