Section Handout #5: Linked Lists

Based on handouts by various current and past CS106B/X instructors and TAs.

Extra practice problems: CodeStepByStep – linked list pointers; Textbook – 11.2, 11.7

Recall our definition of the ListNode structure, which you’ll be getting more practice with during this section. For these problems, **do not modify a node’s data field or use auxiliary structures like Vectors, Stacks, Queues, etc.**

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

Also, note that a linked list written as {1, 1, 2, 3, 5, 8, 13}, e.g., really refers to a linked list structured like this:

```
front
↓
1 ––> 1 ––> 2 ––> 3 ––> 5 ––> 8 ––> 13
```

1. **isSorted** *(linked lists)*

Write a function called `isSorted` that accepts a pointer to a ListNode at the front of a linked list and returns true if the list of integers being passed is in sorted (nondecreasing) order and 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?

2. **removeAllThreshold** *(linked lists)*

Write a function called `removeAllThreshold` that accepts a pointer to a ListNode at the front of a list and removes all occurrences of a given integer value, plus or minus a provided threshold value, from that list. For example, if `front` points to the first ListNode in the following list, then calling `removeAllThreshold(front, 3, 2)` would remove all occurrences of 3 +/- 2 from the list (nodes to be removed are underlined):

```
{3, 9, 4, 2, 1, 0, 3, 5, -2, 10} –> {9, 0, -2, 10}
```

3. **doubleList** *(linked lists)*

Write a function `doubleList` that doubles the size of a list of integers by appending a copy of the original sequence to the end of the list. For example, if a list initially stores the sequence below at left, then passing in the front ListNode into `doubleList` would result in the list storing the sequence on the right (new nodes underlined):

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

If the original list contains `N` nodes, then you should dynamically allocate exactly `N` new nodes to be added. You may not use any auxiliary data structures to solve this problem (no vector, stack, queue, string, etc.). Your function should run in `O(N)` time, where `N` is the number of nodes in the list.
4. split (linked lists)

Write a function `split` that accepts a pointer the front `ListNode` in a linked list as a parameter and rearranges the elements of a list of integers so that all negative values appear before all of the non-negatives, with each group in the same relative order. For example, a call to `split` would change the list on the left into the list on the right:

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

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

5a. reverse (linked lists)

Write a function called `reverse` that reverses the order of the elements in a linked list. For example, if a list initially stores the sequence of integers below at left, it should store the sequence on the right after your function is called: {1, 8, 19, 4, 17} -> {17, 4, 19, 8, 1}

**Bonus:** Solve this problem without creating any new nodes.

5b. braid (linked lists)

Now, write a function `braid` that weaves the reverse of that list into the original. For this problem, you will need to create new nodes. Here are a few examples:

{1, 4, 2} -> {1, 2, 4, 4, 2, 1}

{3} -> {3, 3}

{1, 3, 6, 10, 15} -> {1, 15, 3, 10, 6, 6, 10, 3, 15, 1}

**Bonus:** This one also has an interesting recursive solution.

6. drawPolygonalPath (linked lists)

```c
struct PointNode {
    GPoint data;
    PointNode* next;
};
```

Write a function `drawPolygonalPath` that accepts a reference to a `GWindow` object and a pointer to a `PointNode` (see above) that represents the beginning of a "polygonal path," or a series of connected line segments, and draws the path as a series of dots and lines on the window.

The `GPoint` within each `PointNode` contains an x and y value representing the point along the polygonal path, and the `next` field links to the next point in the path. Each line should be 1 pixel thick and each dot should be 2 pixels wide. Recall the `drawLine` function from the `GWindow` class:

```c
gw.drawLine(x1, y1, x2, y2);
```

Note that the polygonal path can be open or closed. In the case of an open polygonal path, the last `PointNode` in the path will have a `next` value of `nullptr`. In the case of a closed polygonal path (in other words, a polygon), the last node in the path will link back to the first node in the path. Note that `drawPolygonalPath` should also be able to draw a single point. Finally, in the case of a null `PointNode` parameter, you should draw nothing. See the above diagrams for examples.