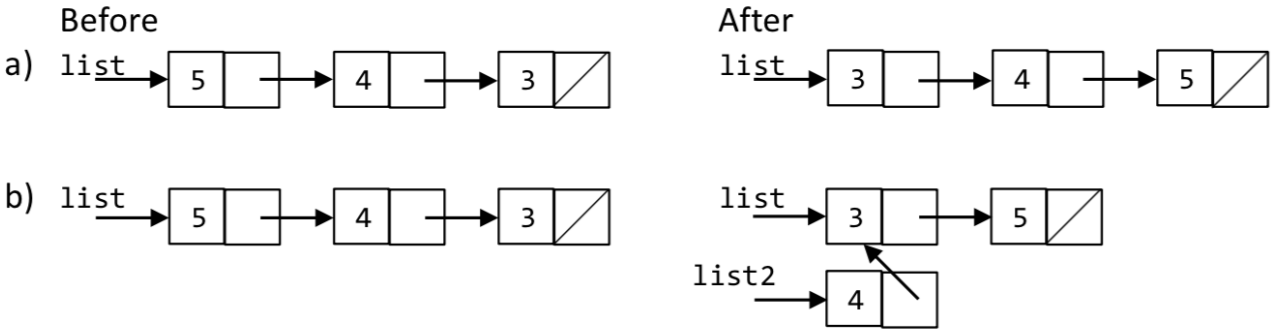
Section #6

Tyler Conklin and Kate Rydberg (Based on handouts by previous CS106B instructors and TAs, especially those of Ashley Taylor and Shreya Shankar.)

Linked Lists

1. Linked Nodes Before/After

Write the code that will produce the given after result from the given before starting point by modifying links between the nodes shown. Do NOT change any existing nodes data field value.



2. Is Sorted

Write a function `isSorted` that accepts a pointer to a `ListNode` representing the front of a linked list. Your function should return true if the list of integers being passed is in sorted (nondecreasing) order and should return false otherwise. An empty list is considered to be sorted.

Bonus: Solve this problem both recursively and non-recursively. Which solution do you prefer?

3. Insert

Write a function named `insert` that accepts a reference to a `ListNodeString` pointer representing the front of a linked list, along with an index and a string value. Your function should insert the given value into a new node at the specified position of the list.

For example, suppose the list passed to your function contains the following sequence of values:

`{ "Tyler", "Kate" }`

The call of `insert(front, 1, "Nick")` should change the list to store the following:

`{ "Tyler", "Nick", "Kate" }`

The other values in the list should retain the same order as in the original list. You may assume that the index passed is between 0 and the existing size of the list, inclusive.

Constraints: Do not modify the data field of existing nodes; change the list by changing pointers only. Do not use any auxiliary data structures to solve this problem (no array, Vector, Stack, Queue, string, etc).

What happens if the list is not passed by reference?

4. Merge

Write a recursive function `merge` that accepts two pointers to two sorted linked lists and returns a pointer to a merged, sorted list.

For example, suppose your function is given the following lists:

{`1,4,5,10,11`}

{`2,4,6,7,8`}

A call to your function should return a pointer to the following list:

`{ 1, 2, 4, 4, 5, 6, 7, 8, 10, 11 }`

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

5. Remove All

Write a function named `removeAll` that accepts two parameters: a reference to a pointer to a `ListNode` representing the front of a linked list, and an integer value. Your function should remove all occurrences of that value from the list. You must preserve the original order of the remaining elements of the list.

For example, if a variable named front points to the front of a list containing the following values:

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

then the call of `removeAll(front, 3)` would remove all occurrences of the value 3 from the list:

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

Constraints: Do not modify the data field of existing nodes; change the list by changing pointers only. Do not construct any new `ListNode` objects in solving this problem (though you may create as many `ListNode*` pointer variables as you like). Do not use any auxiliary data structures to solve this problem (no array, vector, stack, queue, string, etc).

6. Split

Write a function `split` that rearranges the elements of a list of integers so that all negative values appear before all of the non-negatives. The negatives should also appear in the opposite relative order that they appeared in the original list.

For example, suppose a list stores the following values:

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

After a call to your function, the list's contents would be:

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

Constraints: 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.