

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 5 – Basic Linked List

(Week 7, starting 29 February 2016)

1. Creating New Data Structure

We are going to create a new type of linked list in which any node can be “locked”. A locked node can never be removed from the list. Any attempt to remove a locked node has no effect, not even an error message! This new data structure will inherit from the `BasicLinkedList<E>` class covered in lectures.

- (a) First, define a `LockListNode<E>` class that extends `ListNode<E>`. The `LockListNode` class contains data to indicate whether the node is locked or not. A `LockListNode` is unlocked when first created. Write as little extra code as possible by taking advantage of inheritance.
- (b) Implement the `LockList<E>` class, overriding just enough methods to **ensure** that **LockListNodes are always used** in LockLists, and that nodes which are **locked cannot be removed** from the list. Assume that `BasicLinkedList.head` and `num_nodes` have the **protected** access modifier.

Ask yourself, how do you treat a `ListNode` as a `LockListNode`?

- (c) Implement the additional method shown below, within the `LockList<E>` class. Do NOT change the method signature of `lockNode()`. Since `node` is a parameter passed in from an unknown source, do ensure that the node to be locked really refers to a `LockListNode`, and not any other `ListNode`.

```
// permanently lock a node
public void lockNode(ListNode<E> node) { ... }
```

Ask yourself, how do you first check whether a node is really a `LockListNode`, before treating it as such?

2. Add Functionality to Existing Data Structure

We are now going to add new functionality to the `BasicLinkedList<E>` class:

```
public BasicLinkedList<E> split() // part (a)

public static BasicLinkedList<Integer> merge( // part (b)
    BasicLinkedList<Integer> firstList,
    BasicLinkedList<Integer> secondList)
```

Restrictions for BOTH parts (a) and (b):

- You are NOT allowed to traverse the list repeatedly, but you may maintain a few node references
- You are NOT allowed to create any new nodes
- You may not use the `size()` method, or otherwise keep a count of the number of nodes in the list

Since there is the third restriction, you may assume, just for this question, that the `BasicLinkedList` class does NOT keep track of the number of nodes.

- (a) Implement the `split()` method to split the `LinkedList` down the middle, and return the second list. The first list is either the same size, or one larger, than the second list.

As an example:

this list before split: [1 2 3 4 5 6 7 8 9]

this list after split: [1 2 3 4 5]

Returned list: [6 7 8 9]

- (b) Implement the `merge()` class method. Given two linked lists in which all elements are sorted, create a new linked list in which all elements are in **sorted** order. Where there is a draw, always merge the element from the first list. Although a third linked list is created, be reminded that **NO new nodes** are to be created.

As an example:

firstList before merge: [1 3 4 5 5 7]

secondList before merge: [2 2 3 3 5 6]

New returned list: [1 2 2 3 3 3 4 5 5 5 6 7]

firstList after merge: []

secondList after merge: []

As the original lists will be corrupted, and could potentially corrupt the new list, the method should **clean up** the original lists by emptying them. To simplify this question, you may assume both lists are non-empty.

3. Manipulating Linked Lists

There are basically 3 ways to manipulate the contents of a linked list. For example, if we have found the element 4 in the list [1 2 3 4 5], and we want to swap it with the element at the front, we could:

- Manipulate next pointers, so that no node is created, and no node is removed
- Create/instantiate two new nodes
- Manipulate the items in two of the nodes, i.e. swap references to 1 and 4 **WITHIN** the two nodes

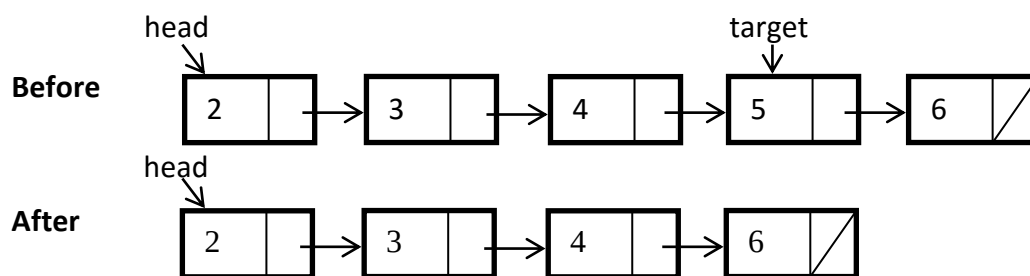
The first approach may seem difficult, but is necessary when we have more complicated reference-based data structures. Therefore, it is good to gain practice tackling a problem using all three ways.

What is the best way to solve the following problem?

```
class BasicLinkedList<E> implements LinkedListInterface<E> {  
    ...  
    public void logicalRemove(ListNode<E> target) { ... }  
}
```

Suppose we have a reference to a node, `target`, which is part of a singly-linked list. The node pointed to is guaranteed NOT to be the last node in the list. We do NOT have references to ANY other nodes, except by calls to `getNext()` in each node. Assume that `ListNode` has getters and setters for both instance variables.

How do we remove the element stored in the node pointed to by `target`, while ensuring that the contents of the list remain accurate?



=p

Draw out your algorithm
Draw your data structures