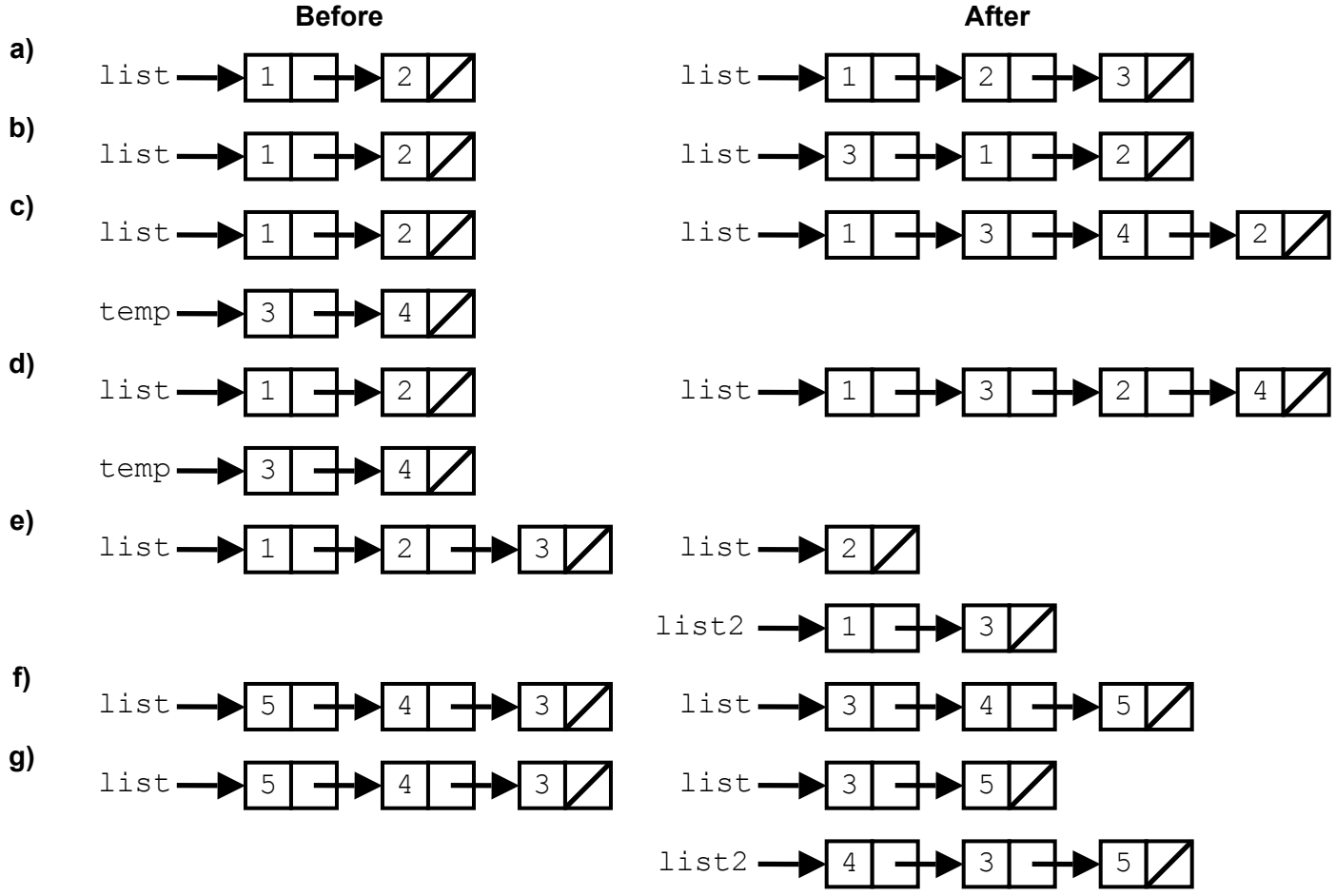
CS106X
Autumn 2015 Cynthia Lee

Section 5 (Week 6) Handout

Problem and solution authors include Marty Stepp and Ilan Goodman

Problem 1: Linked Node Manipulation

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.



Problem 2: Linked List Member Functions

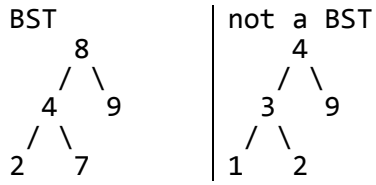
Each of the following subproblems asks you to add a member function to the `LinkedList` class from lecture. In all cases, if your function deletes a node from the list, free the associated memory for the node. Declare member functions as `const` if appropriate, if they do not modify the state of the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

1. 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?
2. 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.
3. Write a member function `reverse` that reverses the order of the elements in the list. For example, if the variable list initially stores the sequence of integers `{1, 8, 19, 4, 17}`, then it should store the following sequence of integers after `reverse` is called: `{17, 4, 19, 8, 1}`.
4. 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 the 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.
5. 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, `ArrayList`s, stacks, queues, etc., to solve this problem.
6. 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}`.

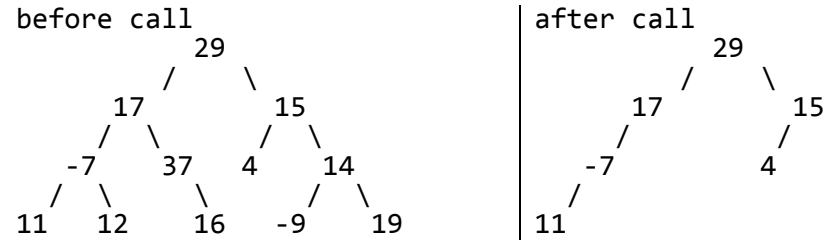
Problem 3: Binary Tree Member Functions

Each of the following subproblems asks you to add a member function to the `BinaryTree` class from lecture. In all cases, if your function deletes a node from the tree, free the associated memory for the node. Declare member functions as `const` if appropriate, if they do not modify the state of the binary tree.

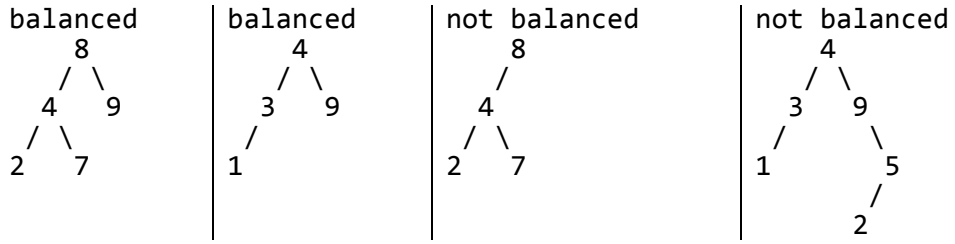
1. **`height`.** Write a member function `height` that returns the height of a tree. The height is defined to be the number of edges along the longest path from the root to a leaf. For example, an empty tree has height -1, a tree of one node has height 0, a node with one or two leaves as children is a tree of height 2, etc.
2. **`isBST`.** Write a member function `isBST` that returns whether or not a binary tree is arranged in valid binary search tree (BST) order. Remember that a BST is a tree in which every node *n*'s left subtree is a BST that contains only values less than *n*'s data, and its right subtree is a BST that contains only values greater than *n*'s data.



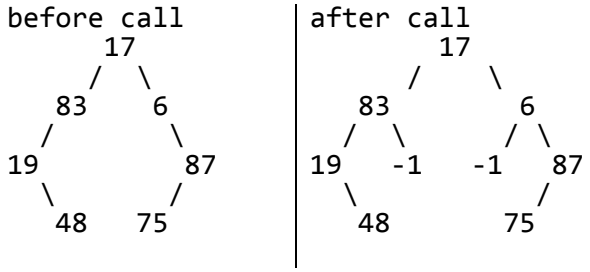
3. **`limitPathSum`.** Write a member function `limitPathSum` that accepts an integer value representing a maximum, and removes tree nodes to guarantee that the sum of values on any path from the root to a node does not exceed that maximum. For example, if variable `t` refers to the tree below at left, the call of `t.limitPathSum(50);` will require removing node 12 because the sum from the root down to that node is more than 50 (29 + 17 + -7 + 12 = 51). Similarly, we have to remove node 37 because its sum is (29 + 17 + 37 = 83). When you remove a node, you remove anything under it, so removing 37 also removes 16. We also remove the node with 14 because its sum is (29 + 15 + 14 = 58). If the data stored at the root is greater than the given maximum, remove all nodes, leaving an empty (`NULL`) tree. Free memory as needed, but only remove nodes when necessary.



4. **`isBalanced`.** Write a member function `isBalanced` that returns whether or not a binary tree is balanced. A tree is balanced if its left and right subtrees are *also balanced trees* whose heights differ by at most 1. The empty (`NULL`) tree is balanced by definition. You may call solutions to other section exercises to help you.



5. **`completeToLevel.`** Write a member function `completeToLevel` that accepts an integer *k* as a parameter and adds nodes with value -1 to a tree so that the first *k* levels are complete. A level is complete if every possible node at that level is non-`NULL`. We will use the convention that the overall root is at level 1, its children are at level 2, and so on. Preserve any existing nodes in the tree. For example, if a variable called `t` refers to the tree below and you make the call of `t.completeToLevel(3);` you should fill in nodes to ensure that the first 3 levels are complete. Notice that level 4 of this tree is not complete. Keep in mind that you might need to fill in several different levels. You should throw an integer exception if passed a value for *k* that is less than 1.



6. **`countLeftNodes.`** Write a member function `countLeftNodes` that returns the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. For example, the tree in Problem 1 (a) above has 3 left children (the nodes storing the values 5, 1, and 4).

Problem 4: Aesop's Algorithms

(N.B.: This is a common coding interview question, so take a moment to think about it on your own before discussing it with a neighbor.)

Write a function `hasACycle(const ListNode*& head)` that takes in a pointer to the head of a linked list and returns `true` if the linked list contains a cycle and `false` otherwise. For full credit, you are only allowed to have a constant space complexity (you are only allowed *O(1)* auxiliary space for your algorithm). What is the time complexity of your algorithm, in big-O notation? Is there a better algorithm?

Bonus problem: find the length of the loop (if one exists).

Linked List Reference Sheet

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

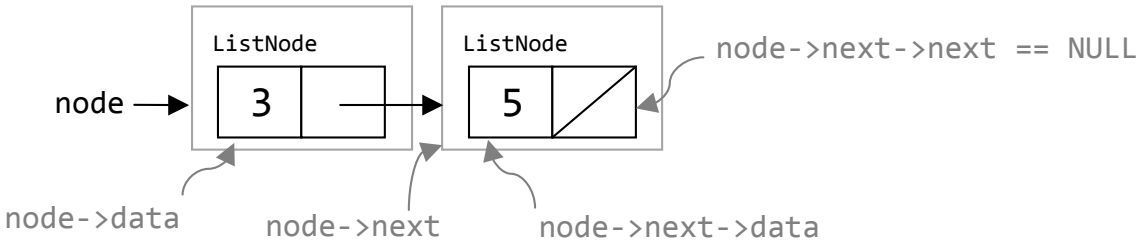
```
struct ListNode {
    int data;          // data stored in this node
    ListNode* next;    // a link to the next node in the list

    // 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).

```
ListNode* node = new ListNode(3);
node->next = new ListNode(5);
```



`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
};
```

Binary Tree Reference Sheet

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    ...
};


class BinaryTree {
public:
    member functions;
private:
    TreeNode* root; // NULL if empty
};
```