CS106X
Autumn 2015 Cynthia Lee

Section 5 (Week 6) – SOLUTION

Problem and solution authors include Marty Stepp and Ilan Goodman

Problem 1 Solution: Linked Node Manipulation

a)
```
list->next->next = new ListNode(3, NULL); // 2 -> 3
```
b)
```
list = new ListNode(3, list);             // 3 -> 1  and  list -> 3
```
c)
```
temp->next->next = list->next;            // 4 -> 2
list->next = temp;                        // 1 -> 3
```
d)
```
list->next->next = temp->next;            // 2 -> 4
temp->next = list->next;                  // 3 -> 2
list->next = temp;                        // 1 -> 3
```
e)
```
ListNode* list2 = list;                   // list2 -> 1
list = list->next;                        // list -> 2
list2->next = list2->next->next;          // 1 -> 3
list->next = NULL;                        // 2 /
```
f)
```
ListNode* temp = list->next->next;        // temp -> 3
temp->next = list->next;                  // 3 -> 4
list->next->next = list;                  // 4 -> 5
list->next->next->next = NULL;            // 5 /
list = temp;                              // list -> 3
```
g)
```
list->next->next->next = list;            // 3 -> 5
list = list->next->next;                  // list -> 3
ListNode* list2 = list->next->next;       // list2 -> 4
list->next->next = NULL;                  // 5 /
```

Problem 2 Solution: Linked List Member Functions

1.
```
bool LinkedList::isSorted() const {
    if (m_front != NULL) {
        ListNode* current = m_front;

        while (current->next != NULL) {
            if (current->data > current->next->data) {
                return false;
            }

            current = current->next;
        }
    }

    return true;
}
```

2.
```
int LinkedList::deleteBack() {
    if (m_front == NULL) {
        throw "empty list";
    }

    int result = 0;

    if (m_front->next == NULL) {
        result = m_front->data;
        delete m_front;
        m_front = NULL;
    } else {
        ListNode* current = m_front;

        while (current->next->next != NULL) {
            current = current->next;
        }

        result = current->next->data;
        delete current->next;
        current->next = NULL;
    }

    return result;
}
```

3.
```
void LinkedList::reverse() {
    ListNode* current = m_front;
    ListNode* previous = NULL;

    while (current != NULL) {
        ListNode* nextNode = current->next;
        current->next = previous;
        previous = current;
        current = nextNode;
    }

    m_front = previous;
}
```

4.
```
void LinkedList::doubleList() {
    if (m_front != NULL) {
        ListNode* half2 = new ListNode(m_front->data);
        ListNode* back = half2;
        ListNode* current = m_front;

        while (current->next != NULL) {
            current = current->next;
            back->next = new ListNode(current->data);
            back = back->next;
        }

        current->next = half2;
    }
}
```

5.
```
void LinkedList::split() {
    if (m_front != NULL) {
        ListNode* current = m_front;

        while (current->next != NULL) {
            if (current->next->data < 0) {
                ListNode* temp = current->next;
                current->next = current->next->next;
                temp->next = m_front;
                m_front = temp;
            } else {
                current = current->next;
            }
        }
    }
}
```

6.
```
void LinkedList::stutter() {
    ListNode* current = m_front;
    while (current != NULL) {
        current->next = new ListNode(current->data,
                                     current->next);
        current = current->next->next;
    }
}
```

Problem 3 Solution: Binary Tree Member Functions

1.
```
int BinaryTree::height() {
    return height(root);
}

int BinaryTree::height(const TreeNode*& node) {
    if (node == NULL) {
        return 0;
    } else {
        return 1 + max(height(node->left), height(node->right));
    }
}
```

2.
```
bool BinaryTree::isBST() {
    TreeNode* prev = NULL;
    return isBST(root, prev);
}

// An in-order walk of the tree, storing the last visited node in
// 'prev'
bool BinaryTree::isBST(const TreeNode*& node, TreeNode*& prev) {
    if (node == NULL) {
        return true;
    } else if (!isBST(node->left, prev) ||
               (prev && node->data <= prev->data)) {
        return false;
    } else {
        prev = node;
        return isBST(node->right, prev);
    }
}
```

3.
```
void BinaryTree::limitPathSum(int max) {
    limPathSum(root, max, 0);
}

void BinaryTree::limPathSum(TreeNode*& node, int max, int sum) {
    if (node != NULL) {
        sum += node->data;

        if (sum > max) {
            deleteTree(node); // frees subtree rooted at node
            node = NULL;
        } else {
            limPathSum(node->left,  max, sum);
            limPathSum(node->right, max, sum);
        }
    }
}
```

4.
```
bool BinaryTree::isBalanced() {
    return isBalanced(root);
}

bool BinaryTree::isBalanced(const TreeNode*& node) {
    if (node == NULL) {
        return true;
    } else if (!isBalanced(node->left) ||
               !isBalanced(node->right)) {
        return false;
    } else {
        int leftHeight  = height(node->left);
        int rightHeight = height(node->right);
        return abs(leftHeight - rightHeight) <= 1;
    }
}
```

5.
```
void BinaryTree::completeToLevel(int k) {
    if (k < 1) {
        throw k;
    }

    completeToLevel(root, k, 1);
}

void BinaryTree::completeToLevel(TreeNode*& node,
                                 int k,
                                 int level) {
    if (level <= k) {
        if (node == NULL) {
            node = new TreeNode(-1);
        }

        completeToLevel(node->left, k, level + 1);
        completeToLevel(node->right, k, level + 1);
    }
}
```

```
6. int BinaryTree::countLeftNodes()  {
       return countLeftNodes(root);
   }
   int BinaryTree::countLeftNodes(TreeNode* node) {
       if (node == NULL) {
           return 0;
       } else if (node->left == NULL) {
           return countLeftNodes(node->right);
       } else {
           return 1 + countLeftNodes(node->left)
                    + countLeftNodes(node->right);
       }
   }
```

Problem 4 Solution: Aesop's Algorithms

```
// Actually runs in O(N) time (prove this)
bool hasACycle(const ListNode*& head) {
    // Two pointers = O(1) auxiliary space
    ListNode* hare = head;
    ListNode* tortoise = head;

    // Loop until we reach the end of an acyclic list
    while (hare != NULL) {
        hare = hare->next;

        if (hare == NULL) {
            return false;
        }

        hare = hare->next;
        tortoise = tortoise->next;

        // We have found a cycle!
        if (hare == tortoise) {
            // To find the length of the loop, just let the hare do a
            // victory lap here until she runs into the tortoise again
            return true;
        }
    }

    return false;
}
```