Chris Gregg
CS106B

Section #6
May 16, 2018

Solution to Section #6

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

1. No, *You're* Out of Order

a. pre-order: 3 5 1 2 4 6
in-order: 1 5 3 4 2 6
post-order: 1 5 4 6 2 3

b. pre-order: 19 47 23 -2 55 63 94 28
in-order: 23 47 55 -2 19 63 94 28
post-order: 23 55 -2 47 28 94 63 19

c. pre-order: 2 1 7 4 3 5 6 9 8
in-order: 2 3 4 5 7 1 6 8 9
post-order: 3 5 4 7 8 9 6 1 2

2. Height

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

3. Count Left Nodes

```
int countLeftNodes(TreeNode *node) {
    if (node == nullptr) {
        return 0;
    } else if (node->left == nullptr) {
       return countLeftNodes(node->right);
    } else {
        return 1 + countLeftNodes(node->left) + countLeftNodes(node->right);
    }
}
```

4. Balanced

```
bool isBalanced(TreeNode *node) {
    if (node == nullptr) {
        return true;
    } else if (!isBalanced(node->left) || !isBalanced(node->right)) {
        return false;
    } else {
        int leftHeight = height(node->left);    // from problem 2
        int rightHeight = height(node->right);
        return abs(leftHeight - rightHeight) <= 1;
    }
}
```

5. Prune a Tree

```
void pruneTree(TreeNode *&node) {
    if (node != nullptr) {
        if (node->left == nullptr && node->right == nullptr) {
            delete node;
            node = nullptr;
        } else {
            pruneTree(node->left);
            pruneTree(node->right);
        }
```

```
    }
}
```

6. Complete To Level

```
void completeToLevel(StringNode *&node, int k) {
    if (k < 1) {
        throw k;
    } else {
        completeToLevelHelper(node, k, 1);
    }
}

void completeToLevelHelper(StringNode *&node, int k, int level) {
    if (level <= k) {
        if (node == nullptr) {
            node = new StringNode;
            node->str = "??";
        }
        completeToLevelHelper(node->left, k, level + 1);
        completeToLevelHelper(node->right, k, level + 1);
    }
}
```

7. Word Trees

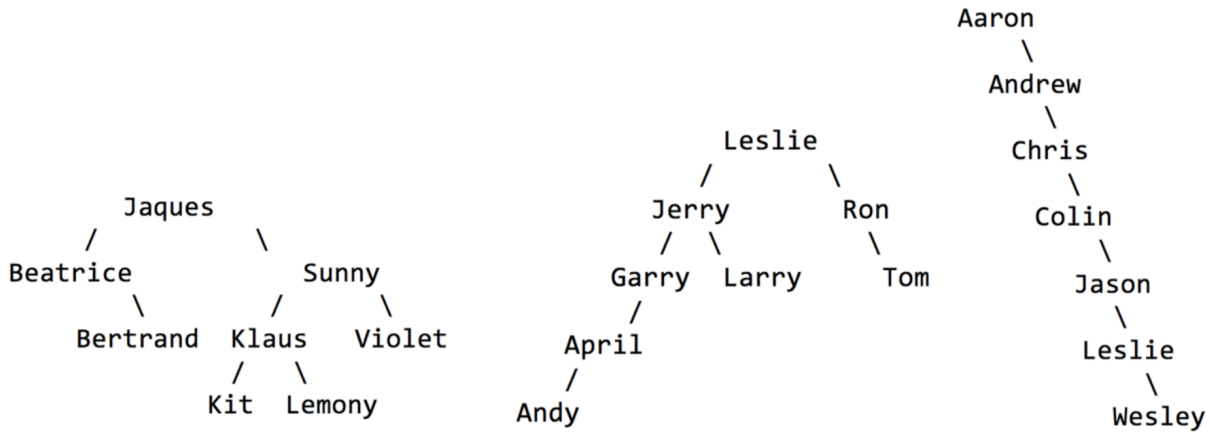
```
bool wordExists(CharNode *node, string str) {
    if (str.empty()) {
        return true;    // the empty tree contains the empty string
    } else if (node == nullptr) {
        return false;
    } else if (suffixExists(node, str)) {
        return true;
    } else {
        return wordExists(node->left, str) || wordExists(node->right, str);
    }
}

// helper to search the given subtree for the rest of the string
bool suffixExists(CharNode *node, string suffix) {
    if (suffix.empty()) {
        return true;
    } else if (node == nullptr) {
        return false;
    } else if (suffix[0] != node->ch) {
        return false;
    } else {
        return suffixExists(node->left, suffix.substr(1)) ||
               suffixExists(node->right, suffix.substr(1));
    }
}
```

8. List to Tree

```
TreeNode *listToBinaryTree(ListNode *front) {
    if (front == nullptr) return nullptr;
    TreeNode *root = new TreeNode;
    root->data = front->data;
```

```
    root->left = listToBinaryTree(front->next);
    root->right = listToBinaryTree(front->next);
    return root;
}
```

9. Binary Search Tree Insertion



(a) (b) (c)

10. Is it a BST?

```
bool isBST(TreeNode *node) {
    TreeNode *prev = nullptr;
    return isBSTHelper(node, prev);
}

bool isBSTHelper(TreeNode *node, TreeNode *&prev) {
    if (node == nullptr) {
        return true;
    // check the left, and then that we are greater than the left
    } else if (!isBSTHelper(node->left, prev) ||
               (prev && node->data <= prev->data)) {
        return false;
    } else {
        prev = node;
        return isBSTHelper(node->right, prev);
    }
}
```

Alternate solution that uses min and max bounds (thanks to our Section Leader Anand!):

```
bool isBST(TreeNode* head) {
    return isBSTHelper(head, INT_MIN, INT_MAX);
}

bool isBSTHelper(TreeNode* curr, int min, int max) {
    if (curr == nullptr) {
        return true;
    }

    if (curr->data < min || curr->data > max) {
        return false;
    }
```

```
    // The left nodes must all be smaller, and the right nodes larger
    return isBSTHelper(curr->left, min, curr->data - 1) &&
               isBSTHelper(curr->right, curr->data + 1, max);
}
```

11. Level-Order Heaps

```
void levelOrderTraversal(int *heap, int size) {
    for (int i = 0; i < size; i++) {
        cout << heap[i];
        if (i < size) {
            cout << " ";
        }
    }
    cout << endl;
}
```

12. Quad Trees [Challenge Problem]

```
QuadTreeNode *gridToQuadTree(Grid<bool> &image) {
    return gridToQuadTree(image, 0, image.numCols(), 0, image.numRows());
}

QuadTreeNode *gridToQuadTree(Grid<bool> &image, int minX, int maxX, int minY,
                             int maxY) {
    QuadTreeNode *qt = new QuadTreeNode;
    qt->minX = minX;
    qt->maxX = maxX - 1;
    qt->minY = minY;
    qt->maxY = maxY - 1;

    if (allPixelsAreTheSameColor(image, minX, maxX, minY, maxY)) {
        qt->isBlack = image[minX][minY];
        for (int i = 0; i < 4; i++) {
            qt->children[i] = nullptr;
        }
    } else {
        int midX = (maxX - minX) / 2 + minX;
        int midY = (maxY - minY) / 2 + minY;

        qt->children[0] = gridToQuadTree(image, minX, midX, minY,midY); // NW
        qt->children[1] = gridToQuadTree(image, midX, maxX, minY,midY); // NE
        qt->children[2] = gridToQuadTree(image, midX, maxX, midY,maxY); // SE
        qt->children[3] = gridToQuadTree(image, minX, midX, midY,maxY); // SW
    }

    return qt;
}

bool allPixelsAreTheSameColor(Grid<bool> &image, int minX, int maxX, int
                              minY, int maxY) {
    bool value = image[minX][minY];
    for (int x = minX; x < maxX; x++) {
        for (int y = minY; y < maxY; y++) {
            if (image[x][y] != value) {
                return false;
            }
```

```
        }
    }

    return true;
}
```