

## Section #6 Solutions

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

### 1. Traversals (binary trees)

a)	b)	c)
Pre-order: 3 5 1 2 4 6	Pre-order: 19 47 23 -2 55 63 94 28	Pre-order: 2 1 7 4 3 5 6 9 8
In-order: 1 5 3 4 2 6	In-order: 23 47 55 -2 19 63 94 28	In-order: 2 3 4 5 7 1 6 8 9
Post-order: 1 5 4 6 2 3	Post-order: 23 55 -2 47 28 94 63 19	Post-order: 3 5 4 7 8 9 6 1 2

### 2. BST Insertion (binary trees)

a)	b)	c)
<pre>           Leia          /   \       Boba       R2D2         \       /       Darth     Luke         /   \     Chewy     Han         \       Jabba   </pre>	<pre>           Meg          /   \       Joe       Stewie         /   \   /     Brian     Lois   Peter         \           \       Cleveland   Quagmire   </pre>	<pre>           Peralta          /   \       Doyle      Santiago         /   \     Diaz     Hitchcock         \       Terry         \       Holt         \       Scully   </pre>

### 3. countLeftNodes (binary trees)

```

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. isBST (binary trees)

```

// approach 1 -- set bounds on valid data values and restrict them while walking down tree
bool isBSTHelper(TreeNode* node, int min, int max) {
    if(node == nullptr) return true;
    if(node->data < min || node->data > max) return false;
    return isBSTHelper1(node->left, min, node->data - 1) &&
           isBSTHelper1(node->right, node->data + 1, max);
}

bool isBST(TreeNode* root) {
    return isBSTHelper(root, INT_MIN, INT_MAX);
} // another approach is shown on the next page
  
```

```

// approach 2 -- use prev pointers to walk back up and across the tree
bool isBSTHelper(TreeNode* node, TreeNode*& prev) {    // an in-order walk
    if (node == nullptr) {                                // of the tree, storing the
        return true;                                     // last visited node in 'prev'
    } else if (!isBSTHelper(node->left, prev) || (prev && node->data <= prev->data)) {
        return false;
    } else {
        prev = node;
        return isBSTHelper(node->right, prev);
    }
}

```

## 5. removeLeaves (*binary trees*)

```

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

```

## 6. completeToLevel (*binary trees*)

```

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

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

```

## 7. wordExists (*binary trees*)

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

bool wordExists(TreeNodeChar* node, const 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);
    }
}
```