CSM 61B Spring 2019

ADTs, Trees

Mentoring 6: March 4, 2019

1 Abstract Data Types

A **list** is an ordered sequence of items: like an array, but without worrying about the length or size.

```
interface List<E> {
    boolean add(E element);
    void add(int index, E element);
    E get(int index);
    int size();
}
```

A **set** is an unordered collection of unique elements.

```
interface Set<E> {
    boolean add(E element);
    boolean contains(Object object);
    int size();
    boolean remove(Object object);
}
```

A **map** is a collection of key-value mappings, like a dictionary in Python. Like a set, the keys in a map are unique.

```
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
    boolean containsKey(Object key);
    Set<K> keySet();
}
```

2 Interview Questions

2.1 Define a procedure, `sumUp`, which returns **`true`** if any two values in the `array` sum up to `n`.

```
public static boolean sumUp(int[] array, int n) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        if (seen.contains(n - value)) {
            return true;
        }
        seen.add(value);
    }
    return false;
}
```

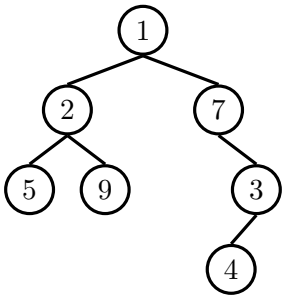
2.2 Define a procedure, `isPermutation`, which returns **`true`** if a string `s1` is a permutation of `s2`. For example, `"atc"` and `"tac"` are permutations of `"cat"`.

```
public static boolean isPermutation(String s1, String s2) {
    Map<Character,Integer> characterCounts = new HashMap<>();
    for (char c : s1.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count + 1);
    }
    for (char c : s2.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count - 1);
    }
    for (char c : characterCounts.keySet()) {
        if (characterCounts.get(c) != 0 ) {
            return false;
        }
    }
    return true;
}
```

3 Binary Trees

3.1 Define a procedure, `height`, which takes in a `Node` and outputs the height of the tree. Recall that the height of a leaf node is 0.

```
public class BinaryTree<T> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
}
```




```
private int height(Node node) {
    if (node == null) {
        return -1;
    }
    return 1 + Math.max(height(node.left), height(node.right));
}
```

Meta: That this Binary Tree class is an encapsulated binary tree. This means that in a recursive function, we want to recurse on that particular node.

What is the runtime of `height`?

$\Theta(N)$, where N is the number of nodes in the tree. We visit every node once and at each node perform a constant amount of work (`null` check). The actual "work" that contributes to the order of growth is done in the recursion, where we repeatedly step down through every node in the tree.

3.2 Define a procedure, `isBalanced`, which takes a `Node` and outputs whether or not the tree is balanced. A tree is **balanced** if the left and right branches differ in height by at most one and are themselves balanced.

```
private boolean isBalanced(Node node) {
    if (node == null) {
        return true;
    } else if (Math.abs(height(node.left) - height(node.right)) <= 1) {
        return isBalanced(node.left) && isBalanced(node.right);
    }
    return false;
}
```

What is the runtime of `isBalanced`?

$\Theta(N)$ in the best case, $\Theta(N \log N)$ in the worst case. This can also be read as $\Omega(N), O(N \log N)$ overall.

The best case is if the tree is unbalanced at the root, meaning that the difference in the height of the root's left branch and the root's right branch is greater than one. In this case, we just call `height` twice, once on the left branch and once on the right subtree. After these two calls, we can immediately see that the tree is unbalanced, so we return false. This leads to a runtime of $\Theta(N)$ in the best case.

The worst case is if the tree is perfectly balanced. In this case, first, we will call `height` on `node.left` and `node.right`. Each of these nodes has a sub-tree of roughly $N/2$ nodes, and so at this level, 2 height calls are made, each of which costs $N/2$. The total work done on this level is $\Theta(N)$. Next, `node.left` will call `height` on its left and right children, and `node.right` will do the same. These children are now on the third level of the tree (the root node being the first level). Note that these third-level children now have a subtree of roughly $N/4$ nodes each. 4 `height` calls are made at this level, for a total cost of $\Theta(N)$ at this level too. As we keep going, each level will do $\Theta(N)$ work. Note that the bottom-most level (leaf-level) of such a perfectly balanced tree would have roughly $N/2$ nodes, and each height call would take constant time, for a total of $\Theta(N/2) = \Theta(N)$ work at the leaf-level too.

Now that we have established that each level does $\Theta(N)$ work, all that we need to figure out is how many levels there are in our worst case situation. This tree has $\log n$ levels, since a perfectly balanced tree has $\log n$ levels. Therefore, the total runtime cost for the worst case is $\Theta(N \log N)$, which can also be read as $O(N \log N)$

3.3 Define `isSymmetric` which checks whether the binary tree is a mirror of itself.

```
public boolean isSymmetric() {
    if (root == null) {
        return true;
    }
    return isSymmetric(root.left, root.right); // use helper method
}

private boolean isSymmetric(Node left, Node right) {
    if (left == null) {
        return right == null; // if left is null, right must also be null
    } else if (right == null) {
        return false;         // left is not null but right is null, so not symmetric
    } else if (!left.value.equals(right.value)) {
        return false;         // left value and right value are unequal
    } else {
        return isSymmetric(left.right, right.left) &&
               isSymmetric(left.left, right.right);
    }
}
```

Meta: We can use a helper function here to create a new method that takes in two parameters, the left and the right branch of the current tree we are rooted at. This allows us to more easily compare the content of the two branches to see if they are the same.

4 Binary Search Trees

4.1 Provide tight asymptotic runtime bounds in terms of N, the number of nodes in the tree, for the following operations and data structures.

Operations	Binary Search	Balanced Search
`boolean contains(E e);`	$\Omega(1)$ $O(N)$	$\Omega(1)$ $O(\log N)$
`boolean add(E e);`	$\Omega(1)$ $O(N)$	$\Theta(\log N)$

For `boolean contains(E e)`, the best case is always $\Theta(1)$ for both binary search trees and balanced search trees. This is the case if the node we are looking for is at the root of the tree. The worst case for binary search trees is if the tree is completely unbalanced (a spindly tree), and the node we are looking for is a leaf, yielding a time complexity of $\Theta(N)$ because we have to visit every node.

However, for a balanced search tree, we know that the height of the tree will never exceed $\Theta(\log N)$. Thus, even if the node we are looking for is a leaf, we will never have to search more than $\Theta(\log N)$ nodes.

For `boolean add(E e)`, we always have to insert nodes as a a leaf. For a binary search tree, imagine a right-leaning spindly tree (where all of the root's children are in its right subtree). To insert a node in the root's left subtree, this would simply be a constant time operation because there are no nodes in the left subtree, so we would just need to set the root's left child to the node we are inserting. This is the best case for binary search trees, so the runtime is $\Theta(1)$ in this case. However, to insert a node in the root's right subtree, we would have to move all the way down the spindly tree, passing every node, to reach the leaf node to insert the new node. This is the worst case for binary search trees, so the runtime is $\Theta(N)$.

For balanced binary search trees, the height will always be $\Theta(\log N)$ nodes. This means that every insertion will always have to visit $\Theta(\log N)$ nodes to reach a leaf. Thus, the runtime is always in $\Theta(\log N)$.