CSM 61B
Fall 2020

# ADTs and BSTs

Mentoring 7: October 3 - October 9, 2020

## 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 Array Lists

3.1 Mallory is designing a resizing `ArrayList` implementation. She needs to decide the amount to resize by. Help her figure out which option provides the best runtime.

Assuming Mallory resizes her `ArrayList` when it's full, what is the average runtime of adding an element to the `ArrayList`?

(a) When full, increase the size of the array by 1 element.

This would be in $\Theta(N)$ since every time we want to add a new element, we will have to create a new array with space for one additional element and copy all the previous elements ($N$ elements) over.

(b) When full, increase the size of array by 10,000 elements.

This would still be in $\Theta(N)$ since the constant, 10,000, does not scale with the size of the array. We're resizing the array every 10,000 inputs, regardless of the size of the array which could be much larger than 10,000.

(c) When full, double the size of the array.

This would be in $\Theta(1)$.

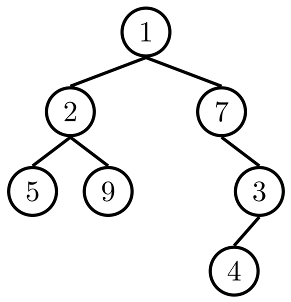
| Operation | Elements Added | Cost | Array Usage |
|---|---|---|---|
| `add()` | 1 | 1 | 1/10 |
| `add()` | 1 | 1 | 2/10 |
| `add()` $\times 8$ | 8 | 8 | 10/10 |
| `add()` | 1 | 11 | 11/20 |
| `add()` $\times 9$ | 9 | 9 | 20/20 |
| `add()` | 1 | 20 | 21/40 |
| `add()` $\times 19$ | 19 | 19 | 40/40 |
| `add()` | 1 | 40 | 41/80 |
| `add()` $\times 39$ | 39 | 39 | 80/80 |

Given a full array of size $\frac{N}{2}$, the runtime of resizing it would be in $\Theta(N)$. This new array of size $N$ would allow us to insert $\frac{N}{2}$ elements before having to resize again. This means the runtime of inserting each one of those elements, amortized, is in $\frac{\Theta(N)}{\frac{N}{2}}$, which is in $\Theta(1)$.

## 4 Stacks and Queues

A **queue** is a data structure that orders items in a first-in-first-out (FIFO) manner, meaning that the first element you `add` will be at the front and the last item you `add` will be at the tail. Elements are `removed` from the front. A **stack** is a data structure that orders items in a last-in-first-out (LIFO) manner, meaning that the first element you `add` will be at the tail and the last item you `add` will be at the front. Elements are `removed` from the front.

4.1
```
public void treeTraversal(Fringe<Node> fringe) {
    fringe.add(root);
    while (!fringe.isEmpty()) {
        Node node = fringe.remove();
        System.out.print(node.value);
        if (node.left != null) {
            fringe.add(node.left);
        }
        if (node.right != null) {
            fringe.add(node.right);
        }
    }
}
```



What would Java display?

(a) `tree.traversal(new Queue<Node>());`

`1275934`

A Queue has a FIFO (First in First Out) nature that is used to perform the traversal.
To be more specific, you add node 1 to the queue. You then pop it and print its value, and add its children, 2 and then 7 to the queue. Since it is FIFO, you would pop 2 out first and add all its children, 5 and then 9 to the queue. Then you would pop 7, and add its child 3 to the queue. Then you would pop 5 and 9, who have no children to add. Next, you pop 3, add 4 to queue. Lastly you pop 4 and you are done!

Make sure you explain the FIFO nature of the Queue and visually represent how the solution 1275934 is arrived at using a Queue.

(b) `tree.traversal(new Stack<Node>());`

`1734295`

A Stack has a LIFO (Last in First Out) nature that is used to perform the traversal.

To be more specific, you add node 1 to the queue. Pop it, add its children, 2 and then 7 to the queue. (Every time you pop a node it gets printed). Since it is LIFO, you would pop 7 out first and add all its children, 3 to the queue. Then you would pop 3, and add its child 4, to the queue. Then you would pop 4, but since it has no children, you would not add anything to the Stack. Once you've popped 4, you would pop 2, and add its children to the Stack - 5 and then 9. Then you would pop 9, but because it has no children, nothing is added to the Stack. Lastly, pop 5!

## 5 Binary Search Trees

5.1 Implement `fromSortedArray` for binary search trees. Given a sorted `int[] array`, efficiently construct a balanced binary search tree containing every element of the array.

```
public class BinarySearchTree<T extends Comparable<T>> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
    public static BinarySearchTree<Integer> fromSortedArray(int[] values) {
        BinarySearchTree<Integer> bst = new BinarySearchTree<>();
        bst.root = bst.fromSortedArray(values, 0, values.length - 1); // setting a new root
        return bst;
    }
    private Node fromSortedArray(int[] values, int lower, int upper) {
       if (lower > upper) {
           return null;
       }
       int middle = lower + ((upper - lower) / 2); // middle index of the array
       Node mid = new Node();
       mid.value = values[middle];
       mid.left = fromSortedArray(values, lower, middle - 1);  // recurse on the left half
       mid.right = fromSortedArray(values, middle + 1, upper); // recurse on the right half
       return mid;
    }
}
```

**Meta**: Be sure to explain the difference between a binary tree and a binary search tree.