

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.

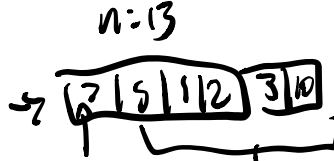
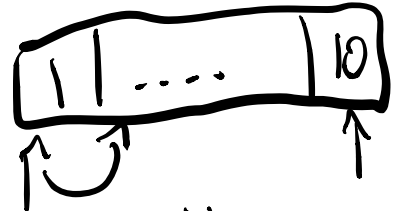
```
interface Map<K,V> {  
    V put(K key, V value);  
    V get(K key);  
    boolean containsKey(Object key);  
    Set<K> keySet();  
}
```

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) {
```

OCN) `Set<Integer> seen = new HashSet<>();`
`for (int value : array) {`
 `if (seen.contains(n - value)) {`
 `return true; // O(1)`
 `}`
 `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) {
```

- use map
 - loop through strings,
 - letter = key
 - # of occurrences = value



```
Map<Character, Integer> cache = new HashMap<>();
```

```
for (Char c : s1.toCharArray()) {
```

```
    if (cache.containsKey(c)) {
```

```
        cache.put(c, cache.get(c) + 1);
```

```
    } else {
```

```
        cache.put(c, 1);
```

```
    }
```

```
for (Char c : s2.toCharArray()) {
```

```
    if (!cache.containsKey(c)) {
```

```
        return false;
```

```
    } else {
```

```
        cache.put(c, cache.get(c) - 1);
```

```
    }
```

```
for (Char c : cache.keySet())
```

```
    if (cache.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.

n additions: $1 + 2 + 3 + 4 \Rightarrow \underline{\underline{\theta(N^2)}}$
 divide by N additions $\rightarrow N$
 $\theta(N)$

resizing

- create new arraylist
- copy over element

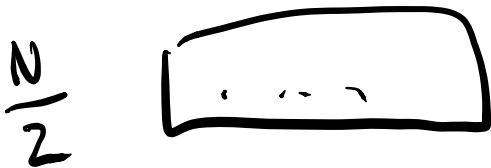
(b) When full, increase the size of array by 10,000 elements.

$\theta(N)$

$\theta\left(\frac{N}{10000}\right) \Rightarrow \theta(N)$
 $\nearrow \approx 10000$
 $\theta\left(\frac{N/2}{10000}\right)$

$\frac{3N}{10000}$

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

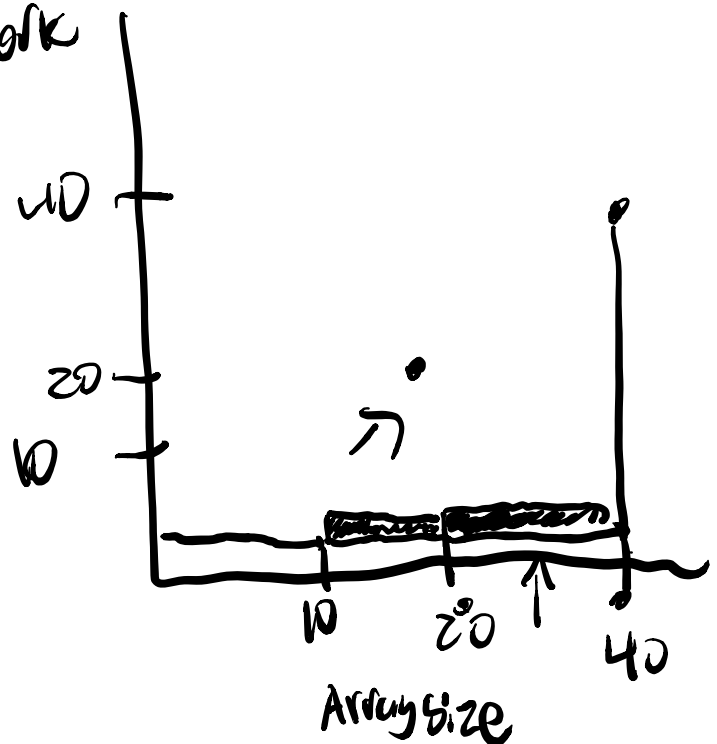


resizing $\theta(N)$

initial size = n

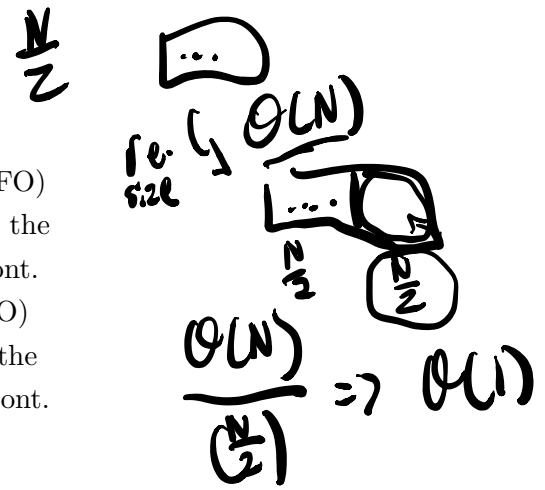


$\frac{\theta(N)}{\left(\frac{N}{2}\right)} \Rightarrow \theta(1)$

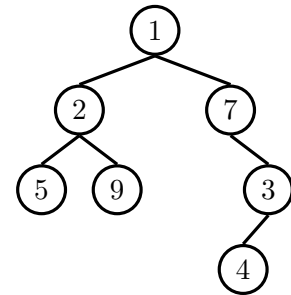


4 Stacks and Queues

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>());`

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

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);
        return bst;
    }
    private Node fromSortedArray(int[] values, int lower, int upper) {

```