CSM 61B
Fall 2017

Sequences & ADTs

Mentoring 7: October 9, 2017

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 An `array` contains all the numbers from 0 to n except for some number, k. Define a procedure, `missingNo`, which returns k given the input `array`.

```
public static int missingNo(int[] array) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        seen.add(value);
    }
    for (int i = 0; i <= array.length; i++) {
        if (!seen.contains(i)) {
            return i;
        }
    }
    return -1;
}
```

Or, using the sum of the arithmetic series.

```
public static int missingNo(int[] array) {
    int sum = 0;
    for (int value : array) {
        sum += value;
    }
    int expectedSum = (array.length * (array.length + 1)) / 2;
    return expectedSum - sum;
}
```

2.3 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 Amortized Analysis

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 Finding Duplicates *Extra Practice*

4.1 Define `findDuplicatesWithinK`, a procedure which, when given an `int[] array` and an boundary range k, returns a `Set` of all duplicates within k indices of each other.

```
findDuplicatesWithinK([1, 2, 3, 1, 4, 3], 3) // {1, 3}
findDuplicatesWithinK([1, 2, 3, 1, 4, 3], 2) // {}
```

```
public static Set<Integer> findDuplicatesWithinK(int[] array, int k) {
    Map<Integer,Integer> seen = new HashMap<>();
    Set<Integer> duplicates = new HashSet<>();
    for (int i = 0; i < array.length; i++) {
        if (!seen.containsKey(array[i])) {
            seen.put(array[i], 0);
        }
        if (seen.get(array[i]) > 0) {
            duplicates.add(array[i]);
        }
        seen.put(array[i], seen.get(array[i]) + 1);
        if (i - k >= 0) {
            seen.put(array[i - k], seen.get(array[i - k]) - 1);
        }
    }
    return duplicates;
}
```

The question is asking for all duplicates in all "sliding windows" of k values in the array. The solution keeps track of all duplicates in the set `duplicates`. The map `seen` changes as we iterate through the array. Before processing index `i`, its keys are the elements in the array seen before index `i`, and the corresponding values are the number of times they key has been seen on and after the index `i-k`.

At every index in the iteration, we check if the value of the key corresponding to the element of the array we are currently looking at is greater than `0`. If it is, then it was previously seen within the current `k`-element sliding window, and we therefore need to add that element to our set of `duplicates`. We then decrement the value of the key corresponding to the `i-k`th element in the array by one before moving on to the next index.