Map ADT

Interface `Map<K,V>`

Type Parameters:

- `K` - the type of keys maintained by this map
- `V` - the type of mapped values

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Collection Views

The Map interface provides three collection views, which allow a map's contents to be viewed as:

1. set of keys
2. collection of values
3. set of key-value mappings.

The order of a map is defined as the order in which the iterators on the map's collection views return their elements.

Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Concerning `Map` keys

We should be careful if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map.

Concerning constructors

All general-purpose map implementation classes should provide two "standard" constructors:

- a void (no arguments) constructor which creates an empty map
- a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument.

Concerning constructors (Cont.)

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map.

For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Restrictions

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false.

Methods

static interface `Map.Entry<K,V>` : A map entry (key-value pair).

- `void clear()` : Removes all of the mappings from this map (optional operation).
- `boolean containsKey(Object key)` : Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)` : Returns true if this map maps one or more keys to the specified value.
- `Set<Map.Entry<K,V>> entrySet()` : Returns a Set view of the mappings contained in this map.
- `boolean equals(Object o)` : Compares the specified object with this map for equality.

Methods (Cont.)

- `V get(Object key)`: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- `int hashCode()`: Returns the hash code value for this map.
- `boolean isEmpty()`: Returns true if this map contains no key-value mappings.
- `Set<K> keySet()`: Returns a Set view of the keys contained in this map.
- `V put(K key, V value)`: Associates the specified value with the specified key in this map (optional operation).
- `void putAll(Map<? extends K,? extends V> m)`: Copies all of the mappings from the specified map to this map (optional operation).

Methods (Cont.)

- `V remove(Object key)`: Removes the mapping for a key from this map if it is present (optional operation).
- `int size()`: Returns the number of key-value mappings in this map.
- `Collection<V> values()`: Returns a Collection view of the values contained in this map.

Common Data-Structures

- BST-Set
  - Tree of key, value pair
- Hashmap with LinkedList or BST bucket
  - Fixed size array with values being bucket
- Associative Array
  - Array of key, value pairs

Bisection Search

Or binary search is a method to find something in a sorted list

Exercise: Bisection Search

Recursive implementation:

```
public int runBinarySearchRecursively(int[] sortedArray, int key, int low, int high) {
    // TODO: return the index of an array where key is located at
    //       else return -1
    //
    //
}
```

Solution

```
public int runBinarySearchRecursively(int[] sortedArray, int key, int low, int high) {
    int middle = (low + high) / 2;

    if (high < low) {
        return -1;
    }

    if (key == sortedArray[middle]) {
        return middle;
    } else if (key < sortedArray[middle]) {
        return runBinarySearchRecursively(sortedArray, key, low, middle - 1);
    } else {
        return runBinarySearchRecursively(sortedArray, key, middle + 1, high);
    }
}
```

Exercise: Bisection Search

Iterative implementation:

```
public int runBinarySearchIteratively(int[] sortedArray, int key, int low, int high) {
    // TODO: return the index of an array where key is located at
    //       else return -1
    //
}
```

Solution

```
public int runBinarySearchIteratively(int[] sortedArray, int key, int low, int high) {
    int index = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (key > sortedArray[mid]) {
            low = mid + 1;
        } else if (key < sortedArray[mid]) {
            high = mid - 1;
        } else if (sortedArray[mid] == key) {
            index = mid;
            break;
        }
    }

    return index;
}
```

Lab exercise

Binary Search