Sets and Maps

Weiss, sections 6.7 and 6.8 and from section 6.10 to the end of chapter 6 (pages 261–274 and 277–292)
Set

• A collection of elements without duplicates
• Start from an empty set and add elements to it
  \[
  \{ \} \rightarrow \{ 1, 4, 2, 9, 8 \}
  \]
• The set does not change if you add 4 to it.
• Like list, set supports adding, removing, and iterating over elements
• Unlike list, set is not a sequence. Thus, elements do not have indices.
• \textit{contains} for a set is performed faster than O(N).
Set Operations

`boolean Add(T v)`: adds v to a set if it’s not already present. It returns a boolean to indicate if the v was added.

`boolean remove(T v)`: removes v from the set. It returns a boolean that indicates whether v was present.

`boolean contains(T v)`: returns true if v is in the set and false otherwise.

`int size()`: returns the size of the set.
SortedSet

• Maintains items in *natural* ordering order or by specifying a *comparator* when creating a SortedSet.
  – Iterates through the elements in the sorted order
• Supports finding the *smallest* and *largest* elements
• TreeSet implements SortedSet
```java
public class MyTreeSetComparator implements Comparator<String> {
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
}

public class TestMyTreeSet {
    public static void main(String[] args) {
        Set<String> ts = new TreeSet<String>(new MyTreeSetComparator());
        ts.add("B");
        ts.add("A");
        ts.add("C");

        for(String element:ts) {
            System.out.print(element + " ");
        }
    }
}

Output: [C B A]
HashSet

• Offers no ordering guarantee.
• Contains only unique elements. To do this, HashSet uses the `equals` method to check if elements are the same.
• HashSet uses an internal array to store elements. The internal array is called *hash table*.
• Hash table places elements into specific positions based on *integers* called *hash codes* (array indices).
• Every object has a hash code. It is generated by `hashCode` method.
• Hash codes, obtained from hashing, provide *constant* time for searching, inserting, and removing elements.
HashSet

Set<String> stooges= new HashSet<>();
stooges.add(“Larry”);
stooges.add(“Moe”);
stooges.add(“Curly”);
stooges.add(“Moe”); //duplicate, won’t be added

Iterator<String> itr = stooges.iterator();
while(itr.hasNext()) {
    String stoog = itr.next();
    System.out.print(stoog + “ “);
}
Output: [Moe, Larry, Curly]
HashSet

• Conversion constructor converts a Collection $c$ (e.g., array list or linked list) to a Set.

```java
List<String> c = new ArrayList<String>();
c.add("Bob");
c.add("Joe");
c.add("Bob"); // duplicate
Set<String> noDups = new HashSet<String>(c); // contains Bob and Joe
```
Mapping Structure

• Defines mapping relationship between elements.
  • For example, a dictionary maps words to definitions. A bank account maps account numbers to balances.
• Map data structure maps keys of type K to values of type V.
• \((k, v)\) is a key-value pair. It is an entry in the map.
• For dictionary, keys are words and values are definitions.
• The types of keys and values can be different.
• A map does not contain duplicate keys.
• A key maps to at most one value.
MAP Operations

V put(K k, V v): Puts an entry for key k and associates it with value v. It returns the *previous* value associated with key k, or null if there was not a mapping.

V remove(K k): removes the entry for key k and returns the corresponding value if the key exists.

int size(): returns the number of entries in the map.

boolean containsKey(K k): returns true if the map contains an entry for key k.

V get(K k): returns a value v associated with key k, or null if k does not exist.
MAP Operations

Set\langle k \rangle \text{ keySet}(): \text{return}s a \textit{set} of keys for this map.
Collection\langle v \rangle \text{ values}(): \text{return}s a collection of values (duplicates are allowed).

- Map allows \textit{null} values. If \textit{get} returns null, it’s not clear whether null refers to an \textit{unsuccessful} search or null \textit{value}.
- \textit{containsKey} can be used if a map contains null values.
Association List

• Implement Map as a *list* of key-value pairs.
• Represent pairing using a class. The first component is key and the second component is its value.

```java
public class Pair<T,U> {
    private T fst,
    private U snd;
    public Pair(T fst, U snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```
Association List

• Define a mapping from strings to integers. Integer is the size of string.

[]
[(“dog”, 5)]
[(“dog”, 5), (“doghouse”, 8), (“cat”, 3)]

• To put a key-value pair, a Map checks if key already exists. If so, it deletes the old key-value pair. Then, re-adds the key with the new value

[(“doghouse”, 8), (“cat”, 3), (“dog”, 3)]

• size returns 3.

• get and containsKey traverse the list to find if a pair has the key.

• The running time of the above operations is similar to the List operations. E.g., put takes O(N) time (checks if a key exists in the list)
Tree Maps

• Sort map entries by key. Keys need to be comparable.
• This is done through a binary search tree called tree map.
• The elements of the search tree are key-value pairs.

```
(“dog”, 5)       (”dog”, 5)
    /         \
(“apple”, 5)    (”mouse”, 5)
```

• Traverse tree map in-order to obtain the list of keys in the sorted order
• The running time of Map operations is $O(\log N)$, which is faster than the list-based map.