Chapter 21

Sets and Maps
23.2 Sets
A Set contains no duplicate elements

Set:
* no duplicates
* values have no specific order (order of insertion is not the same as the order of extraction)
A HashSet uses a hash function to map its elements

* Each value maps to a hashCode
* Two equal values should map to the same hashCode
* Two non equal values could also map to the same hashCode. Try to avoid such collisions
* ints have their value as the hashCode
* chars use their UNICODE value
* strings use: \( s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + ... + s_{n-1} \times 31^0 \), where \( s_i \) is s.charAt(\( i \))
package inclass.collections;

import java.util.*;

public class TestHashSet {
    public static void main(String[] args) {
        Set<String> hashSet = new HashSet<>();

        hashSet.add("London");
        hashSet.add("Paris");
        hashSet.add("New York");
        hashSet.add("San Francisco");
        hashSet.add("Beijing");
        hashSet.add("New York");
System.out.println(hashSet);

[San Francisco, New York, Paris, Beijing, London]
// Method A: use an iterator.
Iterator<String> iterator = hashSet.iterator();

while(iterator.hasNext()) {
    System.out.print(iterator.next().toUpperCase() + " ");
}

System.out.println();

[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON
// Method B: use fast enumerator.
for (String element : hashSet) {
    System.out.print(element + " ");
}

[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON
San Francisco New York Paris Beijing London
public class TestCollectionMethods {
    public static void main(String[] args) {
        Set<String> hashSet1 = new HashSet<>();
        hashSet1.add("London");
        hashSet1.add("Paris");
        hashSet1.add("New York");
        hashSet1.add("San Francisco");
        hashSet1.add("Beijing");

        System.out.println(hashSet1.size() + " elements in hashSet1");
        System.out.println(hashSet1);

        5 elements in hashSet1
        [San Francisco, New York, Paris, Beijing, London]
// Delete an element from hashSet1.
hashSet1.remove("London");

System.out.println("\n" + hashSet1.size() + " elements in hashSet1");
System.out.println(hashSet1);

5 elements in hashSet1
[San Francisco, New York, Paris, Beijing, London]

4 elements in hashSet1
[San Francisco, New York, Paris, Beijing]
Set<String> hashSet2 = new HashSet<>();

    // Add some elements to hashSet2.
    hashSet2.add("London");
    hashSet2.add("Shanghai");
    hashSet2.add("Paris");

    System.out.println("\n" + hashSet2.size() + " elements in hashSet2");
    System.out.println(hashSet2);

4 elements in hashSet1
[San Francisco, New York, Paris, Beijing]

3 elements in hashSet2
[Shanghai, Paris, London]
System.out.println("\nIs Taipei in hashSet2? " +
    hashSet2.contains("Taipei");

3 elements in hashSet2
[Shanghai, Paris, London]
Is Taipei in hashSet2? false
// Set union.
hashSet1.addAll(hashSet2);
System.out.println("UNION: hashSet1 after adding hashSet2 to it\n" + hashSet1);

Is Taipei in hashSet2? false

UNION: hashSet1 after adding hashSet2 to it
[San Francisco, New York, Shanghai, Paris, Beijing, London]
// Set difference.
hashSet1.removeAll(hashSet2);
System.out.println("DIFFERENCE: hashSet1 after removing hashSet2 from it\n" + hashSet1);

UNION: hashSet1 after adding hashSet2 to it
[San Francisco, New York, Shanghai, Paris, Beijing, London]

DIFFERENCE: hashSet1 after removing hashSet2 from it
[San Francisco, New York, Beijing]
// Set intersection.
hashSet1.retainAll(hashSet2);
System.out.println("INTERSECTION: hashSet1 after retaining common elements in hashSet2\n" + hashSet1);

DIFFERENCE: hashSet1 after removing hashSet2 from it
[San Francisco, New York, Beijing]

INTERSECTION: hashSet1 after retaining common elements in hashSet2 []
A `LinkedHashSet` supports ordering

- Uses a linked list implementation
- Elements can be retrieved in the order they were added
public class TestLinkedHashSet {
    public static void main(String[] args) {
        Set<String> linkedHashSet = new LinkedHashSet<>();

        linkedHashSet.add("London");
        linkedHashSet.add("Paris");
        linkedHashSet.add("New York");
        linkedHashSet.add("San Francisco");
        linkedHashSet.add("Beijing");
        linkedHashSet.add("New York");

        System.out.println(linkedHashSet);
    }
}

A TreeSet offers increasing/decreasing ordering

- Uses a linked list implementation
- Elements can be retrieved in the order they were added
package inclass.collections;

import java.util.*;

public class TestTreeSet {
    public static void main(String[] args) {
        Set<String> hashSet = new HashSet<>();
        hashSet.add("London");
        hashSet.add("Paris");
        hashSet.add("New York");
        hashSet.add("San Francisco");
        hashSet.add("Beijing");
        hashSet.add("New York");
TreeSet<String> treeSet = new TreeSet<>(hashSet);
System.out.println("Sorted tree set (ascending)" unlaw); System.out.println(treeSet);

Sorted tree set (ascending)
Application Deconstructed
< TestTreeSet.java >

// Use methods in the SortedSet interface.
System.out.println("first(): " + treeSet.first());
System.out.println("last(): " + treeSet.last());
System.out.println("headSet("New York"): " +
    treeSet.headSet("New York");
System.out.println("tailSet("New York"): " +
    treeSet.tailSet("New York");

Sorted tree set (ascending)

first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
// Use methods in the NavigableSet interface.
System.out.println("lower("Paris"): " + treeSet.lower("Paris");
System.out.println("higher("Paris"): " + treeSet.higher("Paris");
System.out.println("floor("Paris"): " + treeSet.floor("Paris");
System.out.println("ceiling("Paris"): " + treeSet.ceiling("Paris");

first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]

lower("Paris"): New York
higher("Paris"): San Francisco
floor("Paris"): Paris
ceiling("Paris"): Paris

lower vs floor: lower is less than, floor is less than or equal
higher vs ceiling: higher is greater than, ceiling is greater than or equal
Application Deconstructed
< TestTreeSet.java >

// Remove the first and last elements.
System.out.println("pollFirst(): " + treeSet.pollFirst());
System.out.println("pollLast(): " + treeSet.pollLast());

System.out.println("New tree set: " + treeSet);

lower("Paris"): New York
higher("Paris"): San Francisco
floor("Paris"): Paris
ceiling("Paris"): Paris

pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
23.5 Maps
Maps are efficient at accessing data

- Uses a linked list implementation
- Elements can be retrieved in the order they were added
A Map stores unique key/value pairs

- Uses a linked list implementation
- Elements can be retrieved in the order they were added
The three Map classes of the JCF

- Uses a linked list implementation
- Elements can be retrieved in the order they were added
package inclass.collections;

import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        // Create a HashMap.
        Map<String, Integer> hashMap = new HashMap<>();

        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
System.out.println("HashMap entries:");
System.out.println("  " + hashMap + "\n");

HashMap entries:
{Smith=30, Lewis=29, Anderson=31, Cook=29}
// Create a TreeMap from the previous HashMap.
Map<String, Integer> treeMap = new TreeMap<>(hashMap);

System.out.println("TreeMap entries in ascending key order:");
System.out.println("  " + treeMap + 

HashMap entries:
{Smith=30, Lewis=29, Anderson=31, Cook=29}

TreeMap entries in ascending key order:
{Anderson=31, Cook=29, Lewis=29, Smith=30}
// Create a LinkedHashMap:
boolean accessOrder = true;
Map<String, Integer> linkedHashMap =
    new LinkedHashMap<>(16, 0.75f, accessOrder);

linkedHashMap.put("Smith", 30);
linkedHashMap.put("Anderson", 31);
linkedHashMap.put("Lewis", 29);
linkedHashMap.put("Cook", 29);

System.out.println("LinkedHashMap entries in access-order:");
System.out.println(linkedHashMap);

If accessOrder is set to false then the order is based on insert order.
System.out.println("\nThe age for " + "Lewis is " +
linkedHashMap.get("Lewis").intValue() );

LinkedHashMap entries in access-order:
{Smith=30, Anderson=31, Cook=29, Lewis=29}

The age for Lewis is 29