FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 4 – SUPPORT PRESENTATION

Outline

- Hashing in Java
- Design by Contract – adding custom tags to JavaDoc
- Serialization
- Bibliography

Hashing in Java [1-5]

Hash Table (Overview)

- The hash table is a data structure used to implement an **associative array** (by mapping keys to values) with **constant access time** to its elements.
- Constant access time => no repetitive structures => direct memory access
- The keys will be used as indexes in an array: store the **pair (key, value)** as

bucket[key]=value

- The elements of the array are called **buckets.**
- The **problem** with this approach is the large memory allocated and unused if the key set is sparse.

Hash Table (Overview)

- Solution: define a hash function

hash : Keys -> {1..N}

to reduce the key set to a smaller set of size N.

- The pair (key, value) will be stored as

bucket[hash(key)] = value

- The hash function can lead to collisions when **hash(k1) = hash(k2)**
- In order to save collisions, two techniques are used:
  - **Open Addressing** : probe the next free space from the array in a given sequence
  - **Chaining:** store a list in a bucket. Add all elements with the same hash value in the corresponding list

The Map interface

- There are several data structures in Java that rely on the hash table: HashMap, Hashtable, LinkedHashMap, HashSet
- In order to implement the associative array structure, the Map interface was created.
- A Map in Java holds a collection of pairs key (K) and value (V) defined as: Entry<K,V>
- The various **Map implementations** differ through the underlying data structures:
  - Hash table: HashMap, Hashtable, LinkedHashMap
  - Red-black trees: TreeMap

Hash Map in Java

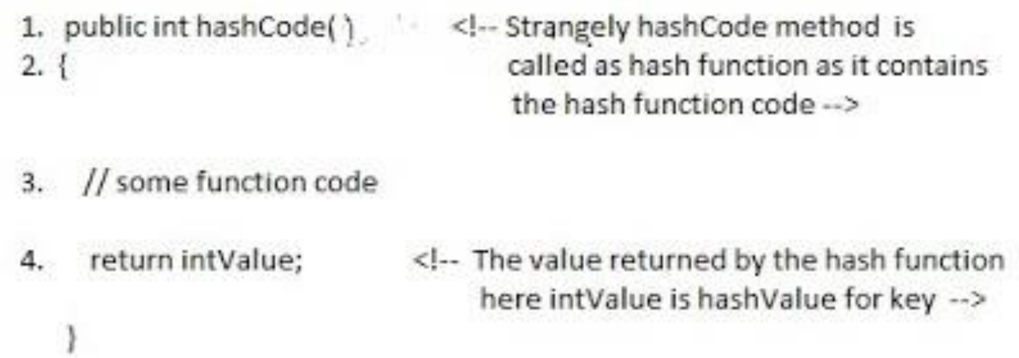
- To understand Hashing in Java , we should understand the following terms :
  - *Hash Function*
  - *Hash Value*
  - *Bucket*
- According to the theory, an associative array/ map contains **key-value** pairs. When implementing a hash table, the key is used to compute an index
- Java is an OOP language. The key is an Object.
- **How can we compute an index (integer) from an object?**

Hashing Elements (1/3)

In order to determine the bucket where to store the Entry<K,V>, two steps are required:

1. Compute a code from the K object

- The Object class defines the method: public int hashCode()
- This method has to be **overridden** for the K object to return an integer computed based on the object's fields
- The hashCode method should return the same integer for two equal, and different integers for different values



```
public int hashCode( )      <!-- Strangely hashCode method  is
{                                called as hash function as it contains
                                 the hash function code -->

   // some function code

    return intValue;        <!-- The value returned by the hash function
                                 here intValue is hashValue for key -->
}
```

Hashing Elements (2/3)

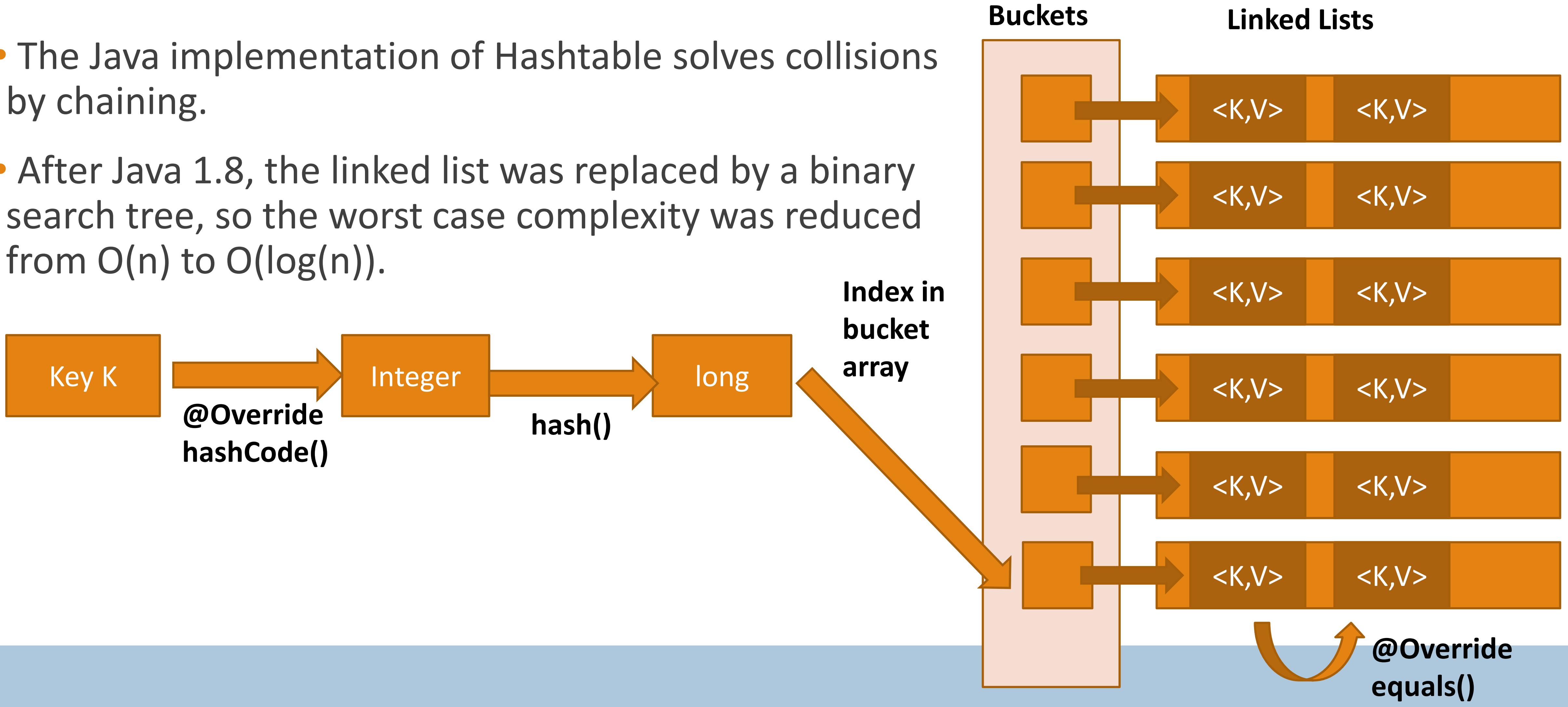
In order to determine the bucket where to store the Entry<K,V>, two steps are required:

2. Apply the hash function on the code to determine the index

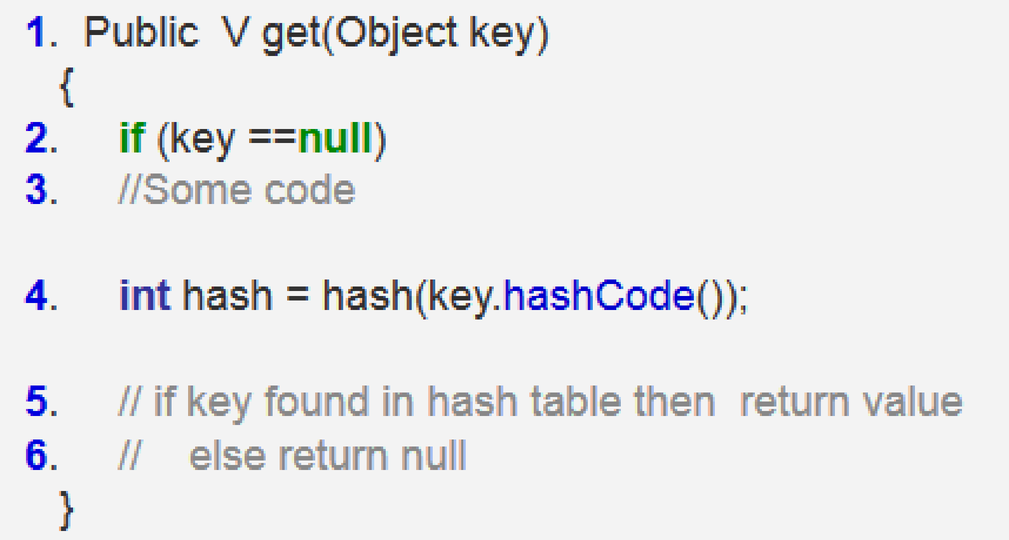
- The code computed by the hashCode method is passed to the internal Java hash function that will compute the index of the bucket that stores the pair
- Index = hash(hashCode(K))
- The hash function implementation is kept internal in the java Collection framework for several reasons:
  - Performance
  - Automatic resize of the hash

Hashing Elements (3/3)

- Each bucket in Java contains a LinkedList.
- The Java implementation of Hashtable solves collisions by chaining.
- After Java 1.8, the linked list was replaced by a binary search tree, so the worst case complexity was reduced from O(n) to O(log(n)).



Retrieving an element from the Map



```
1.  Public  V get(Object key)
    {
2.     if (key ==null)
3.     //Some code

4.     int hash = hash(key.hashCode());

5.     // if key found in hash table then  return value
6.     //    else return null
    }
```

- The hash stores pairs of form Entry<K,V>
- The get method returns the corresponding value V to the given key K
- If key is null (line 1) it returns the element from position 0 (HashMap allows only one null key but several null values)
- Otherwise, it applies the internal static hash function to the hash code of the key to obtain the index of the bucket

- Each **bucket[index]** contains a list with the elements **Entry<K,V>** with the property **hash(hashCode(K))=index**
- When returning the value V corresponding to the key K, the list is iterated and each key stored is compared with K using the **equals** method defined in K’s class.
- If a match is determined, the value V is returned. Otherwise, null is returned.

Adding an element to the Map

The method has the following signature: V put(K key, V value)

1. The index of the bucket is computed as
   - index = hash(hashCode(K))
2. The linked list from bucket[index] is traversed and each element K is **compared using equals** with key
   - If a stored element **equals** with the key then the corresponding value is overriden
   - If no element is found, the pair <key, value> is added to the list stored in bucket[index]

Implementing the Key Object

- Taking into account the mechanisms for get and put, the class that is the type of the Key must override the following methods:
  - **hashCode** – in order to generate a number for each Object. This number will be the input of the internal hash function of the HashMap.
  - **equals** - in order to compare key equality for **get operation** or to check if the object exists in the map, in case of **put operation**.

Complexity of operations

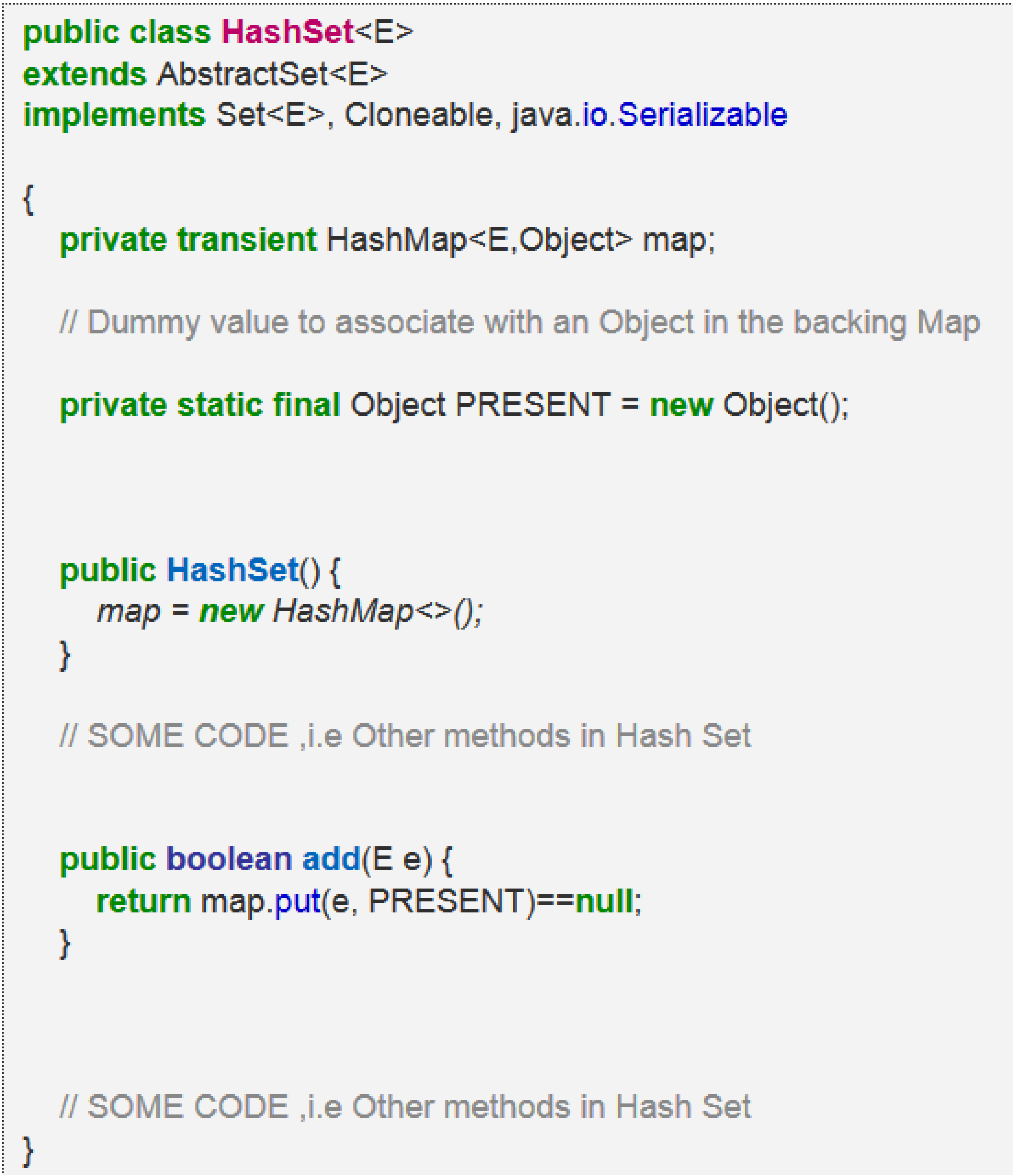
- The average complexity of get and put methods is O(1)
- The worst case complexity of get and put methods is O(n)
- After Java 1.8, the linked list from the buckets is replaced with a binary tree, so the worst case complexity is reduced to O(log(n))

Data structures comparison

Property	HashMap	HashTable	LinkedHashMap	TreeMap
Synchronization or Thread Safe	No	Yes	No	No
Null keys and null values	One null key and any number of null values	No	One null key and any number of null values	Only values
Iterating the values	Iterator	Enumerator	Iterator	Iterator
Iterator type	Fail fast iterator	Fail safe iterator	Fail fast iterator	Fail fast iterator
Interfaces	Map	Dictionary	Map	Map, NavigableMap, SortedMap
Internal implementation	Hashtable with buckets	Hashtable with buckets	Hashtable with double-linked buckets	Red-Black Tree
Get/Put average Complexity	O(1)	O(1)	O(1)	O(log(n))
Get/Put worst complexity	O(n)	O(n)	O(n)	O(log(n))
Space Complexity	O(n)	O(n)	O(n)	O(n)
Order	No guarantee that order will remain constant over time	No guarantee that order will remain constant over time	Insertion-order	Sorted according to natural ordering of the keys

Hash Set

- Does not allow duplicates in the Collection
- Implemented using a HashMap
- The add method returns **false** if the element already exists
- Internally it calls the **put** method of the **Map:**
  - If the element e has not been added yet to the map, the put returns null, thus the add method returns true
  - If the element e is a key in the underlying map, the put method returns the value PRESENT, thus the add method returns false



```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, java.io.Serializable

{
   private transient HashMap<E,Object> map;

   // Dummy value to associate with an Object in the backing Map

   private static final Object PRESENT = new Object();



   public HashSet() {
      map = new HashMap<>();
   }

   // SOME CODE ,i.e Other methods in Hash Set


   public boolean add(E e) {
      return map.put(e, PRESENT)==null;
   }



   // SOME CODE ,i.e Other methods in Hash Set
}
```

HashSet vs TreeSet

Property	Hash Set	Tree Set
Ordering	No	Natural Ordering
Null values	Yes	No
Average Complexity	O(1)	O(log(n))
Worst Complexity	O(n)	O(log(n))
Internal implementation	Hashtable with buckets	Red-Black Trees
Comparison method	equals()	compareTo()

Design By Contract – Adding Custom Tags to JavaDoc

Adding custom tags to javadoc

Reference: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#tag

-tag tagname:Xaoptcmf:"taghead"

Placement of tags: Xaoptcmf

X (disable tag)

a (all)

o (overview)

p (packages)

t (types, that is classes and interfaces)

c (constructors)

m (methods)

f (fields)

Serialization [6-8]

Overview

- Serialization
  - Mechanism of converting the state of an object into a byte stream
- Deserialization
  - The byte stream is used to recreate the actual Java object in memory
- To make a Java object serializable we implement the **java.io.Serializable** interface
- The **ObjectOutputStream** class contains **writeObject**() method for **serializing** an Object
  - public final void writeObject(Object obj) throws IOException
- The **ObjectInputStream** class contains **readObject**() method for **deserializing** an object
  - public final Object readObject() throws IOException, ClassNotFoundException

Overview

- SerialVersionUID
  - a version number associated with each Serializable class
  - used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization
  - if the receiver has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an InvalidClassException

Example – object serialization

```
SerializableClass object = new SerializableClass(); // SerializableClass implements Serializable

FileOutputStream file = new FileOutputStream (filename);
ObjectOutputStream out = new ObjectOutputStream(file);
out.writeObject(object); // Method for serialization of object
out.close();
file.close();
```

Example – object deserialization

```
SerializableClass object = new SerializableClass(); // SerializableClass implements Serializable
 FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
object = (SerializableClass)in.readObject(); // Method for deserialization of object
in.close();
file.close();
```

Bibliography

[1] http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html

[2] http://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html

[3] http://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html

[4] https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

[5] https://en.wikipedia.org/wiki/Hash_table

[6] https://www.baeldung.com/java-serialization

[7] https://www.geeksforgeeks.org/serialization-in-java/

[8] https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html