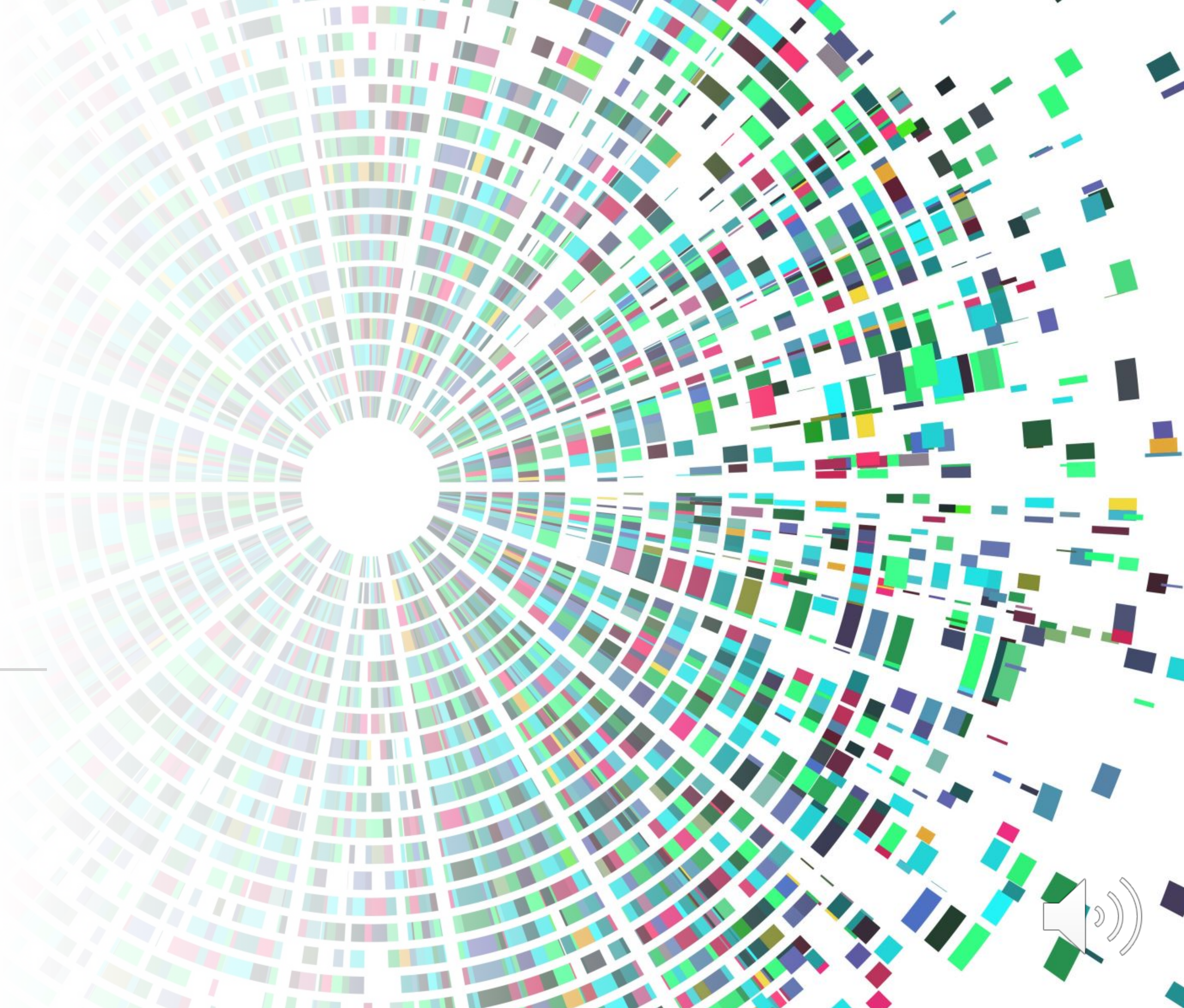
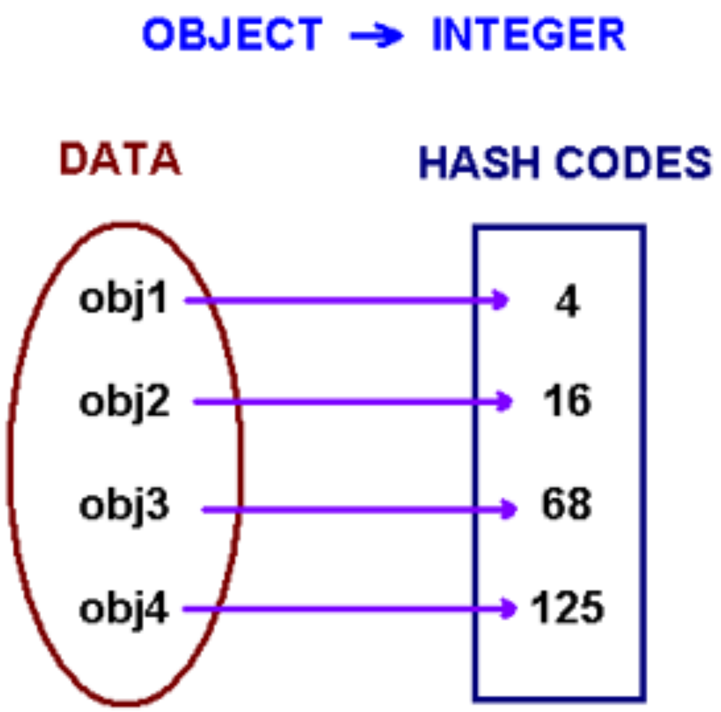
Collections IV

Week 5 Presentation 5



Hashcodes

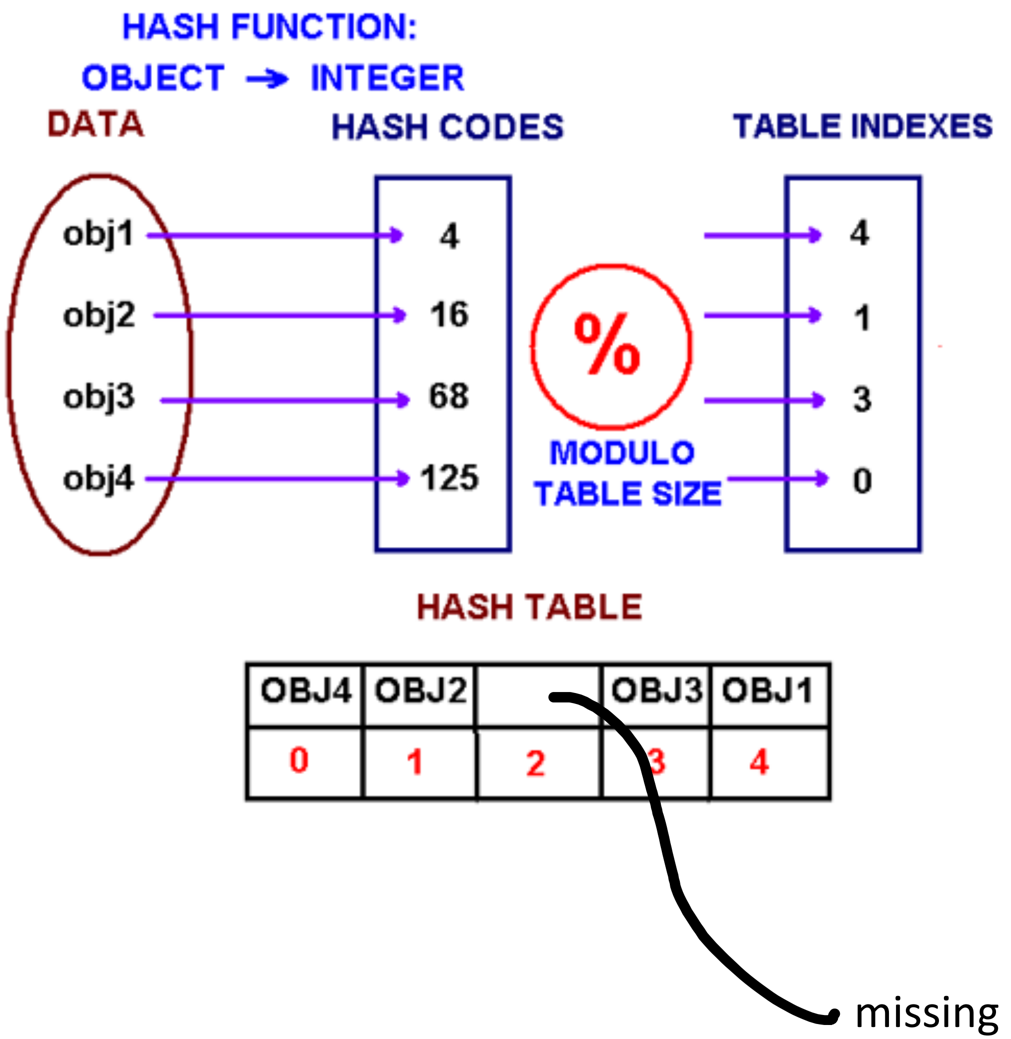
- Instead of adding items to the collection as they come, hashmaps compute a number (called a hash code) for each object, and derive the storage location from this number.
- On a related topic, all Java classes extend the Object class, and this has a method called hashcode() that returns an int.
- Whenever the method is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.



A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects, it is only possible only if you know (or have a good estimate) the number of objects to be stored.

The Java hashing function has the following properties:

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

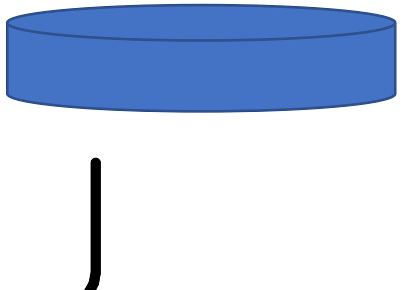


Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. The grouping is a **hash table**.

Can take the hash value mod table size to allocate the object to a slot in an array…

How to choose a hash function (to get hash codes)? We can use Java's *hashCode()* method. The hashCode() method is implemented in the Object class and therefore each class in Java inherits it. The hash code provides a numeric representation of an object (this is somewhat similar to the toString method that gives a text representation of an object).

Hashcodes



```
public static void main(String[] args) {
        ArrayList<City> cities = newArrayList<City>();
        int cityCount = load(cities);
        System.out.println(Integer.toString(cityCount)
                + " cities loaded");
        Collections.sort(cities);

        for (int i = 0; i < cities.size(); i++) {
               System.out.println(cities.get(i) + "\thash = "
                + Integer.toString(cities.get(i).hashCode()));
        }
}
```

```
$ java HashCode
161 cities loaded
Abidjan(ci) – 4765000          hash = 20186994554
Addis Ababa(et) - 3103673      hash = 1311053135
Adelaide(au) – 1316779         hash = 1550089733
Ahmedabad(in) – 5570585        hash = 865113938
Alexandria(eg) - 4616625       hash = 1442407170
Ankara(tr) - 5271000           hash = 1975012498
….
```

hash values (here they are actually just random values!)

We can take a modulo to store each object in one particular position of an array. % table_size.

The method hashCode has different implementation in different classes. In the String class, hashCode is computed by the following formula

$s.charAt(0) * 31^{n-1} + s.charAt(1) * 31^{n-2} + \ldots + s.charAt(n-1)$

EG:

"ABC" = 'A' * 31^2 + 'B' * 31 + 'C' = 65 * 31^2 + 66 * 31 + 67 = 64578

PROBLEM

Different items can hash into the same value.

Collision

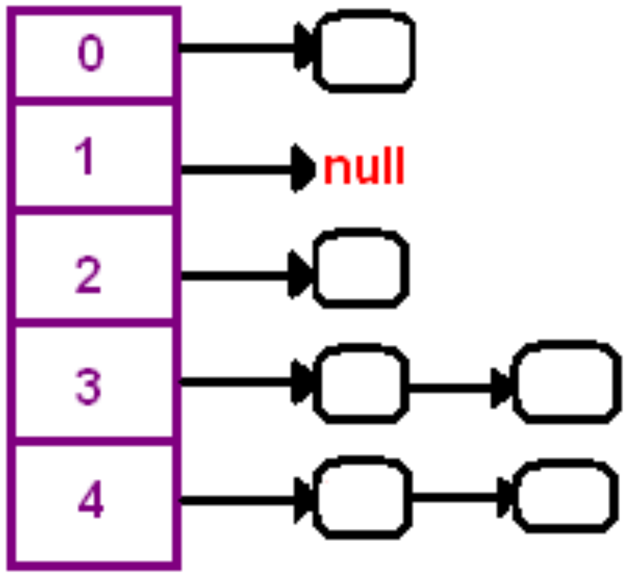
One Solution: Make each array slot a linked list of items hashing to the same value.

Collisions

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key:

"Aa" = 'A' * 31 + 'a' = 2112
"BB" = 'B' * 31 + 'B' = 2112

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.



As you can see, here the hash table is a combination of array and list.
If it's properly designed, it combines the advantages of both (you find quickly each small list, that can be easily updated)

HashMap

(Key, Value) pairs

A HashMap uses this kind of structure to store pairs of objects. The hashcode() value for the key is used for finding the value.

```
import java.util.HashMap;

HashMap<K,V> hm = new HashMap<K,V>();

    hm.put(k, v);
    v = hm.get(k);
    n = hm.size();
    hm.keySet()
    hm.remove(k);
    hm.clear();
```

Pairs

Map Iterator

•

```
import java.util.Iterator;
import java.util.Set;
import java.util.Map;
Set set = hm.entrySet();
Iterator it = set.iterator();
while (it.hasNext()) {
  Map.Entry en = (Map.Entry)it.next();
  System.out.println("key: "
   + en.getKey() + ", value: "
   + en.getValue());
}
```

Hashmaps aren't really designed for accessing the whole set as are optimised by design to access objects one by one. However, as keys are unique, all keys together fit the Set requirement and can be returned as a Set. Then you can get an Iterator on the set, fetch keys one by one and retrieve associated values.

Implicit Iterator: for loop

As with all collections Map type groups also allow accessing heir elements with a `for` loop that uses an implicit iterator.

```
import java.util.Set;
import java.util.Map;

Set set = hm.entrySet();
for (Map.Entry en: set) {
        System.out.println("key: "
                + en.getKey() + ", value: "
                + en.getValue());
}
```

HashMaps

Very good when data is dynamic
Very efficient search
No order, no chronology

The weak spot of hashmaps is that, as location depends only on value, you have no idea (unless you store time in objects) of when you added objects. Contrast with lists, where recent additions are usually at one end.

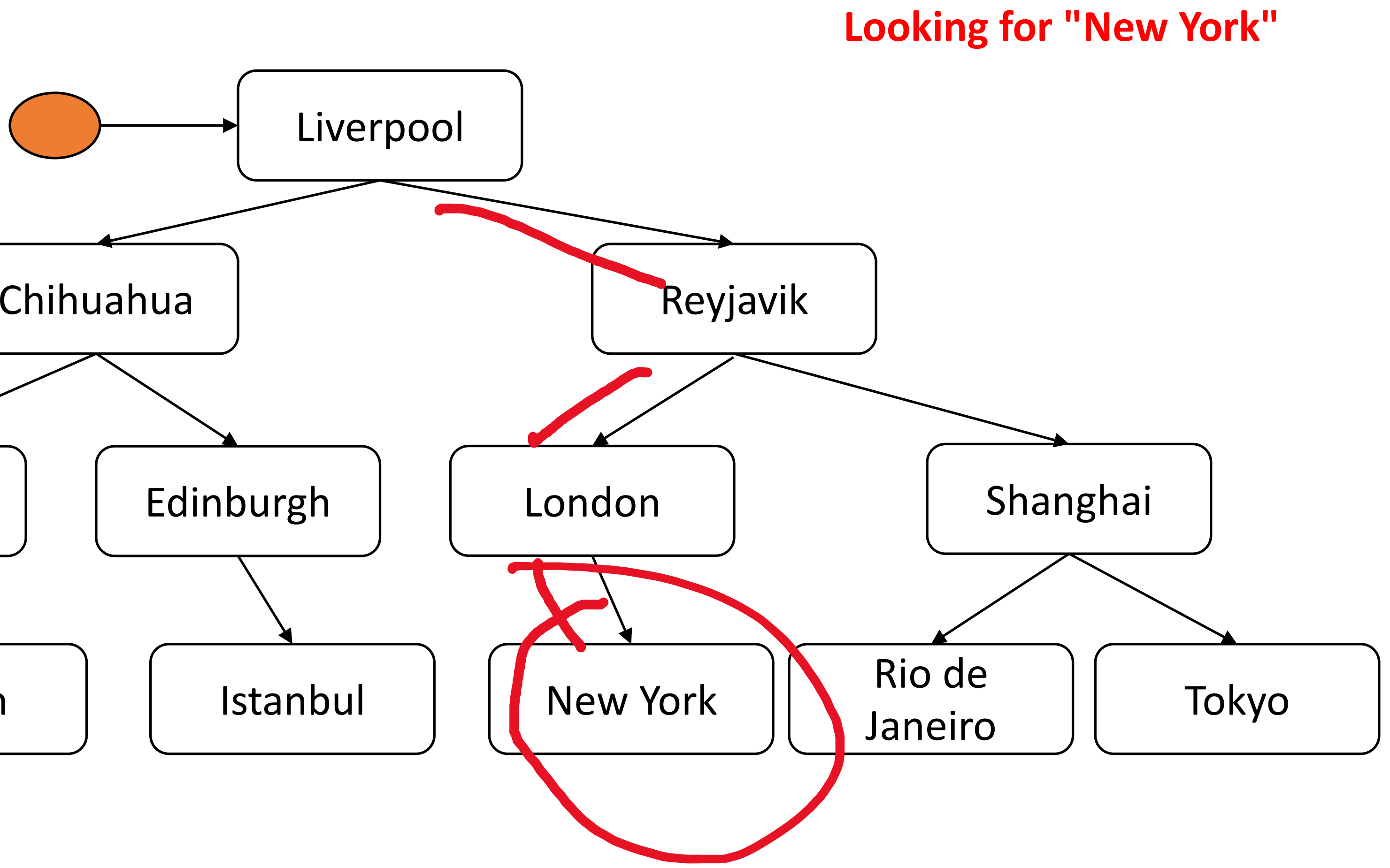
HashSets

= Set implemented with HashMaps

As keys occur only once, a Hashmap can also be used to implement a Set.

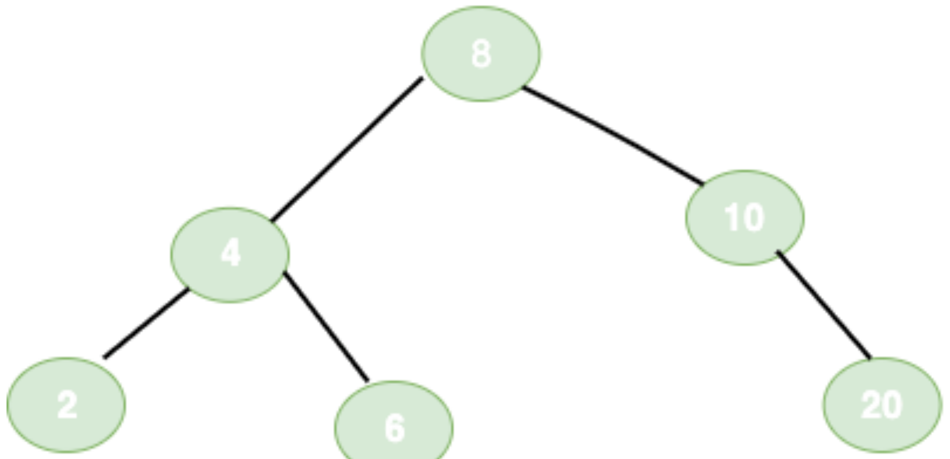
You can also find
some special
classes for
special needs.
Check the docs.

For example – there is a LinkedHashMap class
which returns objects in the order of insertion
when you iterate. It's not the case with a HashMap.

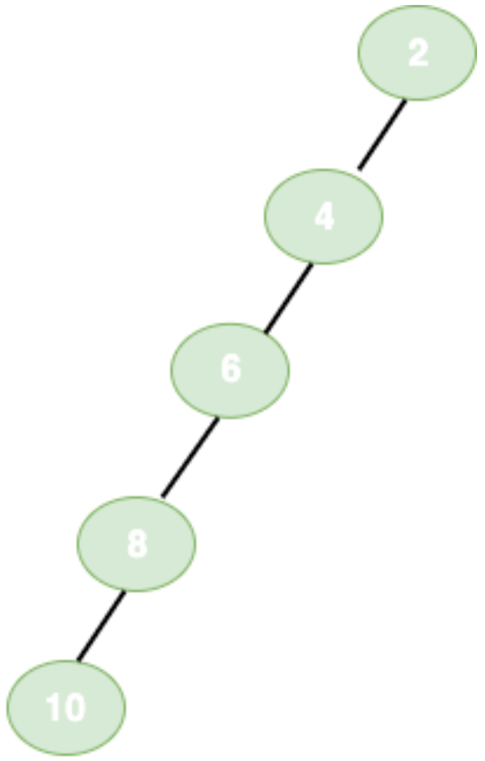


A tree uses reference from object to object
as a list, but each item (usually called a node or a leaf) is linked to more than one other item. You navigate one branch or the other depending on how the value you are looking for compares to the one in the node. As efficient as a binary search.

Trees and Tree Maps



Balanced Binary Search Tree



Unbalanced

Balanced and unbalanced trees

Tree Maps

Also have tree sets (unique values not associations)

Very good when data is dynamic

Efficient search

Ordered by construct

No chronology

Like hash tables, trees don't keep track of time of insertion.

Resizable Array
Linked List
Hash Table
Tree

List
Queue/Deque
Set
Map

Because of the requirements of interfaces, and the limits of implementations, some combinations work very well, and others not at all. Because you lack the time information in a tree or hash table, it makes no sense to implement a Queue/Deque with them. However, arrays and lists are very good for that. When it comes to maps and sets, it's search performance that matters and there it's the opposite − arrays and lists don't really work, hash tables and trees are excellent. It's your requirements and the methods you'll need that dictate your choice of a collection (there are often several possibilities).