Hashing and Binary Search Trees

Dr Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

Slides and examples are available in Moodle

20 februari 2019

Agenda

- Java Collection Classes ⇒ Data structures in the Java Library
  - Available sequential data structures
  - Set implementations in the Java Library
- **Hashing in Java** (e.g., used in `java.util.HashSet`)
- **Binary search trees** (e.g., used in `java.util.TreeSet`)
- Maps

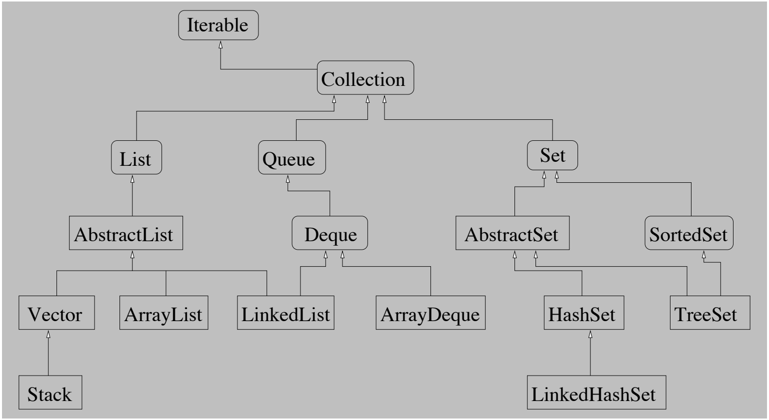
Why study these classes?

- Very useful when you know how to use them
- Examples of common techniques (linked structures, hashing, trees)

As usual, user requirements decide what structure to use ...

- Required properties ⇒ type of structure (e.g., List, Set, Map, Stack)
- Frequently used operations ⇒ type of implementation (e.g., array based)

The Collection Hierarchy



Note: This is a selection of the most important classes. Dropped are:
1) Older collection types saved for backward compability
2) Synchronized data structures ⇒ can only be accessed from one thread at the time.

Java Collections: Basic Information

- All classes belongs to the package `java.util`
- All classes implements the interface `Iterable` ⇒
  - has a method `iterator()` ⇒ access to an iterator of type `Iterator`
  - can be traversed by the simplified for-statement
    `for (int n : list) { ...`
- Sequential Collection Classes: (See Java API doc for more details)
  - Lists: `ArrayList, LinkedList, Vector`
  - Queue and Deque: `ArrayDeque, LinkedList`
  - Stack: `Stack`
- All Collection classes comes in two versions
  1. A **raw** version where all types of data can be stored
     ⇒ type conversion (down-cast) required to access data
     ⇒ uses the class `Object` ⇒ all types of objects can be stored
  2. A **generic** version where only one type of a data can be stored
     ⇒ no type conversion (down-cast) required to access data

The `java.util.Set<T>` Interface

All sets in the Java library implement the *Set* interface

```
public interface Set<T> extends Iterable<T> {
   public boolean add(T t);               // Add t if not already added,
                                          // returns true if added
   public boolean contains(Object o);     // Return true if o contained
   public void remove(Object o);          // Remove o if o contained
   public String toString();              // Print contained elements
   public Iterator<T> iterator();         // Returns iterator over all elements,
   ... and more ...                       // required by Iterable<T>
}
```

Set usage:

- Want to make sure that a collection never contains two identical elements.
  (For example, registered students on a course)
- Want to count how many different XXX you have
  (For example, how many different words a given text has)
  $\Rightarrow$ Add all XXXs to a set and ask the set for its size.
- **We also expect a speedy lookup** $\Rightarrow$ fast on add(), contains(), remove()
- Not sequential $\Rightarrow$ no positions $\Rightarrow$ no get(int index), remove(int index), addAt(...), ...

java.util.HashSet (HashSetMain.java)

```
Set<String> set = new HashSet<String>(); // Create empty set
// Set<String> set = new TreeSet<String>(); // Alternative

set.add("Jonas"); set.add("Jens");        // Add 4 strings
set.add("Jesper"); set.add("Johan");
System.out.println("Size: "+set.size());    // ==> 4

set.add("Jonas");     // Add duplicates
set.add("Jens"); set.add("Jesper");
System.out.println("Size: "+set.size());   // ==> 4

if (set.contains("Jesper"))
    System.out.println("Contains Jesper");      // OK!
if (set.contains("Maria"))
    System.out.println("Contains Maria");       // Not printed

set.remove("Jesper");
System.out.println("Size: "+set.size());   // ==> 3
```

Sets in the Java Library

- *Set* in Java ⇒ collections with no duplicate elements
  (The second attempt to add an element is ignored.)
- Java sets are **not** mathematical sets with operations like union and intersection
- **Example:** A simple list based implementation

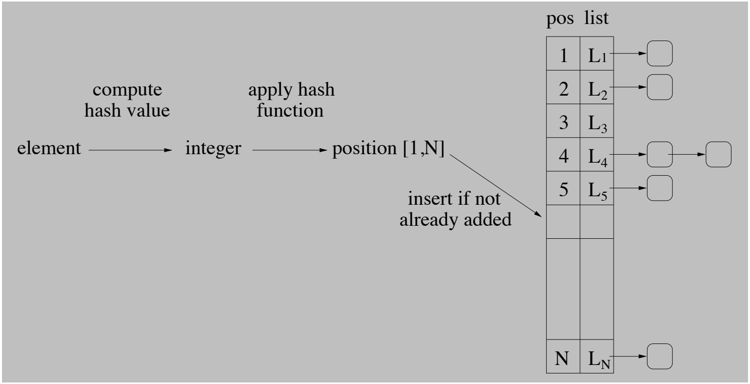
```
public class ListSet implements Set {
   private List list = new ArrayList();

   public void add(Object obj) {   // Time: O(N), N = list size
      if (!list.contains(obj))  list.add(obj);
   }

   public boolean contains(Object obj) {  // Time: O(N), N = list size
      return list.contains(obj);
   }
   ... more methods
```

- **Note** `add`, `contains`, `remove` search the list sequentially
  ⇒ Time proportional to N required where N is the list size
- Sequential search is very costly for larger sets
  ⇒ better implementations are required and available

Three `Set` Classes in the Java Library

1. `java.util.HashSet`
   - Backed up by a hash table ⇒ `add/contains/remove` is very fast, $O(1)$
   - No element ordering ⇒ iteration (e.g., printing) in random order
2. `java.util.TreeSet`
   - Backed up by a binary search tree ⇒ `add/contains/remove` a bit slower than `HashSet`, $O(\log(N))$, but still much faster than a list
   - The elements are ordered using `Comparable` ⇒ iteration (e.g., printing) according to `Comparable`
3. `java.util.LinkedHashSet`
   - Backed up by a hash table and a list ⇒ `add/contains/remove` a tiny bit slower than `HashSet` (but faster than `TreeSet`)
   - The elements are ordered in *insertion* order ⇒ iteration (e.g., printing) in *insertion* order

Hashing – A Brief Presentation



Hashing: Assume table with N *buckets* (A pair position/list)

- Associate each element with a hash value (an integer): `element --> int`
- Apply hash function (maps hash value to a bucket): `int --> bucket`
- Add to the bucket (the list part) if not already added

Hashing – A Concrete Example

A hash table for strings

Assume that ...

- We have a table with 64 *buckets* (current bucket size)
- We compute the hash value for a string by summing up the ASCII codes for each character
- We use a simple modulus operator (`... % 64`) as our hash function

Example

- Adding `"Hello"` ⇒ hash value 500 ($= 72 + 101 + 108 + 108 + 111$)
  ⇒ bucket 52 (since `500 % 64 = 52`)
  ⇒ insert `"Hello"` in bucket 52 (if not already added)
- Adding `"Jonas"` ⇒ hash value 507 ⇒ bucket 59 (`= 507 % 64`)
  ⇒ insert `"Jonas"` in bucket 59 (if not already added)

Hashing – Result

Assume that:

- all elements are evenly distributed across all buckets
  $\Rightarrow$ puts demands on the hash values/functions
- number of elements $\approx$ number of buckets
  $\Rightarrow$ number of buckets increase when the number of elements increase (a process called *rehashing*)

Result: `add/contains/remove` executes in fix number of steps $\Rightarrow O(1)$

Important:

- We must discover when two *similar* (duplicate) elements are added.
- This requires that all *similar* elements:
  - are hashed to the same bucket
    $\Rightarrow$ must be associated with the same hash value
  - must be recognized when traversing the bucket list
- How does these things work in `java.util.HashSet` in the example `HashSetMain.java`?

Hashing in Java

Objects in Java are prepared for hashing.
All objects have the following methods (inherited from `java.lang.Object`)

- `public boolean equals(Object other)`
  DOC: Indicates whether some other object is "equal to" this one.
- `public int hashCode()`
  DOC: Returns a hash code value for the object

Usage in hashing (e.g., in the class `HashSet`)

- `hashCode()` is used to associate each element with a hash value.
- `equals(Object o)` is used to identify identical elements in the linked lists.

Implementation rules for `equals` and `hashCode`:

1. `hashCode()` must always give the same value when called on the same object twice.
2. `o1.equals(o2)` $\Rightarrow$ `o1.hashCode() == o2.hashCode()`
   $\Rightarrow$ two equal elements must have the same hash value.

Similar Objects in java.util.HashSet

Two similar objects can not be added to a set.
(The second attempt will be silently ignored.)

Two objects `o1` and `o2` are considered as *similar* by `HashSet` if

- `o1.hashCode() = o2.hashCode()`, and
- `o1.equals(o2) = true`

Motivation

- Equal `hashCode()` $\Rightarrow$ they are hashed to the same bucket
- `o1.equals(o2) = true` $\Rightarrow$ identified as identical elements in the linked lists.

Example: A class suitable for hashing

```
public class Student {
    private String name;
    private String idNumber;    \\ "YYMMDD-NNNN"

    /* Override Object.equals() */
    public boolean equals(Object other) {
        if (other instanceof Student) {
            Student otherStudent = (Student) other;
            return idNumber.equals(otherStudent.idNumber);  // Compare ID strings
        }
        return false ;
    }

    /* Override Object.hashCode() */
    public int hashCode() {  // Integer based on ID string
        int hc = 0;
        for (int i=0;i<idNumber.length();i++) {
            char c = idNumber.charAt(i);
            hc += Character.getNumericValue(c);   // ASCII number
        }
        return hc;
```

A Simple Hash Set Implementation

```
public class StringHashSet implements StringSet {
   private int sz;                  // Current size
   private Node[] buckets = new Node[8];

   /* Missing methods, will be shown later */

   private class Node {  // Private inner classes
      String value;
      Node next = null;

      public Node(String str) { value = str;   }
      public String toString() {return value;}
   }
}
```

Our simple hash set implementation consists of the following:

- An array of nodes where every element represents a bucket.
  ⇒ array index and node is a bucket (position/list)
- An inner class `Node` that is a linked list.
  (The array `buckets` contains the first element in the lists.)

Implementation: Method add(String str)

```
public void add(String str) {
   int pos = getBucketNumber(str);
   Node node = buckets[pos];   // First node in list
   while (node != null) {      // Search list
      if (node.value.equals(str))
         return;                  // Element found ==> return
      else
         node = node.next;        // Next node in list
   }

   node = new Node(str); // Not found, add new node as first entry
   node.next = buckets[pos];
   buckets[pos] = node;
   sz++;
   if (sz == buckets.length) rehash();   // Rehash if needed
}

private int getBucketNumber(String str) {
   int hc = str.hashCode();        // Use hashCode() from String class
   if (hc < 0) hc = -hc;           // Make sure non-negative
   return hc % buckets.length;     // Simple hash function
```

Methods `rehash` and `contains`

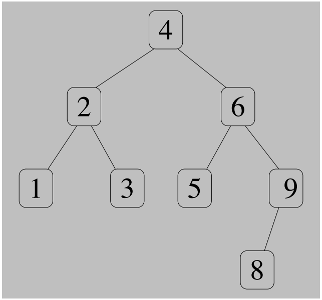
```
private void rehash() {
    Node[] temp = buckets;                       // Copy of old buckets
    buckets = new Node[2*temp.length];  // New empty buckets
    sz = 0;
    for (Node n : temp) {               // Insert old values into new buckets
        if (n == null) continue;        // Empty bucket
        while (n != null) {
            add(n.value);               // Add elements again
            n = n.next;
        }
    }
}

public boolean contains(String str) {
    int pos = getBucketNumber(str);
    Node node = buckets[pos];
    while (node != null) {              // Search list for element
        if (node.value.equals(str))
            return true;        // Found!
        else
            node = node.next;
    }
    return false;    // Not found
}
```

A 10 Minute Break

ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ

Binary Search Trees (BST)

The class `java.util.TreeSet` makes use of *binary search trees*



Note:

- A *tree* consists of *nodes*
- The top-most node (4) is called the *root*
- *Binary trees* ⇒ a maximum of two children for each node
- *Binary search trees* ⇒ left child is always smaller than right child

Question: Where should 7 be placed?

A Simple BST Implementation

```
public class IntBST { // The class accessed by a user
  private BST root = null;

  public void add(int n) {
    if (root==null)
      root = new BST(n);
    else
      root.add(n);
  }
  ... more methods

  private class BST {         // private inner class
    int value;
    BST left = null;
    BST right = null;

    BST(int val) { value = val;}

    void add(int n) { ... recursive add method }
    ... more methods
  }
```

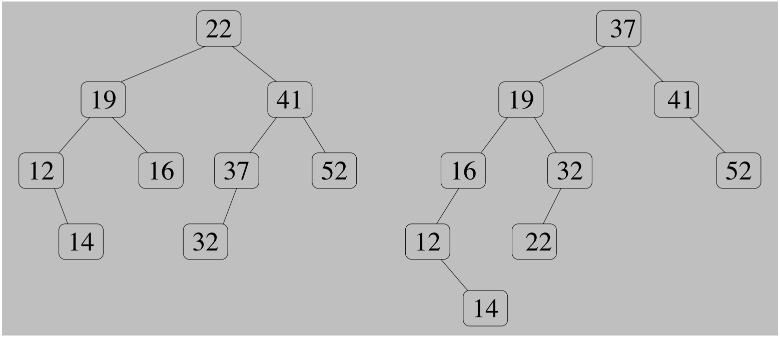
BST: The Recursive Method `add(...)`

```
private class BST {          // private inner class
    int value;
    BST left = null, right = null;

    BST(int val) { value = val; }

    void add(int n) {  // recursive add
        if (n<value) {                // add to left branch
            if (left == null)
                left = new BST(n);
            else
                left.add(n);  // Recursive call
        }
        else if (n>value) {           // add to right branch
            if (right == null)
                right = new BST(n);
            else
                right.add(n);   // Recursive call
        }
    }  // ... more methods
```

Binary Search Trees: Two Examples

Ex1: 22,19,41,52,37,16,12,14,32

Ex2: 37,19,16,12,14,32,41,52,22



Notice:

- **Error in first figure! 16 is at wrong position!**
- Same elements added in different order $\Rightarrow$ two different trees
- No duplicated entries

Recursive method for look-up?

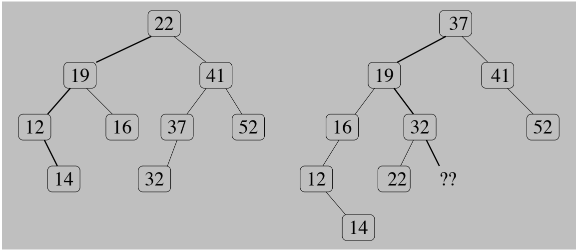
BST: The Recursive Method `contains(...)`

```
private class BST {
   int value;
   BST left = null, right = null;

   /* true if tree contains n */
   boolean contains(int n) { // recursive look-up
      if (n<value) {                    // search left branch
         if ( left == null)
            return false;
         else
            return left.contains(n);
      }
      else if (n>value) {               // search right branch
         if ( right == null)
            return false;
         else
           return right.contains(n);
      }
      return true;                      // Found!
   }
```

Binary Search Trees: Two Examples

Ex1: Search for 14

Ex2: Search for 34



Notice:

- Search 14: completed after 4 steps
- Search 34: completed after 3 steps
- Similar to Binary Search in sorted list
- In general: A search in a tree with N elements requires $\log_2(N)$ steps
  $\Rightarrow$ Time-Complexity for `add`, `remove`, `contains` is $O(\log_2(N))$

Exercise: Find insertion order for `1,2,3,4,5,6,7` that (on average) gives:

- a) the fastest search? b) the slowest search?

Balanced Trees and Speed

- Balanced tree $\Rightarrow$ uniform tree with minimum depth
- $\Rightarrow$ Every level of the tree is full
- A balanced tree with depth n contains $2^{n+1} - 1$ elements
- depth $n \Rightarrow 2^{n+1} - 1$ elements can be searched in n steps
- Examples
  - $n = 10 \Rightarrow$ tree size 2047
  - $n = 15 \Rightarrow$ tree size 65535
  - $n = 20 \Rightarrow$ tree size 2097151
  - $n = 30 \Rightarrow$ tree size 2147483647
  - $n = 40 \Rightarrow$ tree size 2199023255551
- This is very fast compared to sequential search for larger sets
- Microseconds rather than seconds
- More advanced BST algorithms (e.g. Red-Black Trees) always keep the tree balanced $\Rightarrow$ no need to worry about adding elements in a certain order.

Time-complexity for Hashing and BSTs?

Time-complexity for lookup in hash tables and binary search trees?

Hash tables

- Assume number of buckets $\geq$ number of elements and that elements are evenly distributed over all buckets. We can then look up an element in three steps
  1. compute hash value
  2. identify bucket
  3. traverse (very short) list

  $\Rightarrow$ A fix number of computations (independent of table size) $\Rightarrow O(1)$

Binary Search Trees

- 1) Each visited node halves the number of remaining elements, and
  2) The number of operations performed in each node is fix
  $\Rightarrow$ Very similar to binary search $\Rightarrow O(\log_2(N))$

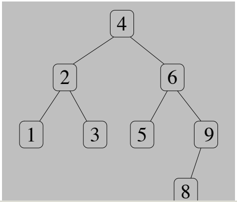
remove(...) – A nightmare, dropped!

```
BST remove(int n) {
    if (n<value) {
        if (left != null) left = left.remove(n);
    }
    else if (n>value) {
        if (right != null) right = right.remove(n);
    }
    else { // remove this node value
        if (left==null) return right;
        else if (right==null) return left;
        else {                          // The tricky part!
            if (right.left == null) {
                value = right.value;
                right = right.right; }
            else
                value = right.delete_min();
        }
    }
    return this;
}
int delete_min() { // more code here ...
```

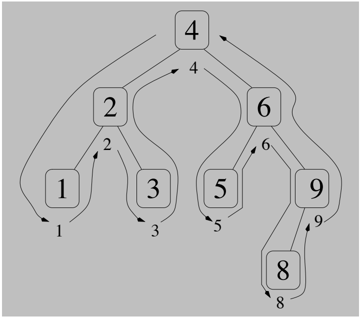
BST – The Method `print()`

```
private class BST {
    int value;
    BST left = null, right = null;

    void print() {
        if (left != null)  // visit left child
            left.print();
        System.out.print(" "+value);  // in-order print value
        if (right != null)  // visit right child
            right.print();
    }
```

Apply algorithm on the following tree: What is printed?



BST: In-order visit



Print-out: `1,2,3,4,5,6,8,9,` ⇒ BST are sorted in principle.
Find min/max:

- Always pick the left-most child ⇒ the lowest added number
- Always pick the right-most child ⇒ the highest added number

Binary Tree Visiting Strategies

```
Left-to-Right, In-order
   visit left subtree (if exist)
   visit node                   ( Do something, e.g., print node value)
   visit right subtree (if exist)

Right-to-Left, Post-order
   visit right subtree (if exist)
   visit left subtree (if exist)
   visit node
```

- `Left-to-Right, Right-to-Left` ⇒ traversal strategies ⇒ decides in which order we visit the children ⇒ a left or right traversal around the tree
- `Pre-order, In-order, Post-order` ⇒ decides when we do something in the node ⇒ before (pre), in between (in), or after (post) we visit the children.

BST –The `TreeSet` Class

- Elements are placed in a special order
  ⇒ `add,contains,remove` is much faster than in a list
- Most efficient if elements are added in random order (non-sorted)
  ⇒ gives a *balanced* tree with a low depth
- A BST is always sorted (in principle)
  ⇒ in-order traversal gives sorted print-out.
- Easy to find minimum/maximum value

Hashing – The `HashSet` Class

- The hash function must provide an even distribution of the elements over the buckets
- Faster than BST on `add,contains,remove`
- No ordering among the elements ⇒
  - print-outs and iteration order not decided
  - Min/max requires that all elements are visted

The `java.util.LinkedHashSet` Class

- Uses a hash table and a linked list to store the elements. Roughly

```
public class LinkedHashSet {
  private Set set = new HashSet();
  private List list = new LinkedList();

  public boolean add(Object obj) {
     if ( set.add(obj) ) {  // true ==> not already added
        list.add(obj);
        return true;
     }
     else
        return false;
  }
}
```

- the hash set ⇒ `add`, `contains` very fast
- the list ⇒ iteration is fast and in *insertion order*
- Disadvantages
  - Elements stored twice ⇒ uses twice as much memory
  - Updating both ⇒ operations are just a bit slower

The Point Class

A simple class suitable for hashing and sorting

```
public class Point implements Comparable<Point> {
   private final int X, Y;

   public Point(int x, int y) {X = x; Y = y;}

   public String toString() {return "("+X+","+Y+")"; }
   public boolean equals(Object other) {
      if (other instanceof Point) {          // Override Object methods
         Point p = (Point) other;
         return p.X==X && p.Y==Y;
      }
      else
         return false;
   }
   public int hashCode() {return X^Y;}  // bitwise XOR

   public int compareTo(Point p) {       // Implement comparable.
      if (X == p.X) return Y-p.Y;        // Used in TreeSet
      else return X-p.X;
   }
```

Using Point in Sets

```
//Set<Point> points = new HashSet<Point>();
Set<Point> points = new TreeSet<Point>();
//Set<Point> points = new LinkedHashSet<Point>();

points.add(new Point(3,3)); points.add(new Point(3,2));
points.add(new Point(5,4)); points.add(new Point(1,3));
points.add(new Point(1,2));

/* Add duplicates */
points.add(new Point(3,2)); points.add(new Point(5,4));
points.add(new Point(1,3));

/* Print results */
System.out.println("Set implementation: "+points.getClass().getName());
System.out.println("Size: "+points.size());    // ==>  Size: 5
System.out.print("Content:");
for (Point p : points)
   System.out.print(" "+p);
-------------------------  Content print-outs ----------------------------
HashSet (Content)   TreeSet (ordered)           LinkedHashSet (insertion order)
random              (1,2)(1,3)(3,2)(3,3)(5,4)  (3,3)(3,2)(5,4)(1,3)(1,2)
```

Summary: Hashing and BST

Hashing

- Very fast, $O(1)$, but not ordered
- Requires good `hashCode()` and that $\mathit{BucketSize} \approx \mathit{NoOfElements}$
- Available as `java.util.HashSet` in the Java Library
- `HashSet` uses to `equals(..)` and `hashCode()` to identify similar elements
  $\Rightarrow$ Override these methods in your classes before storing them in `HashSets`

Binary Search Trees

- Fast, $O(\log_2(N))$, and elements are ordered
- Requires a balanced three and fast "size comparison" to be efficient.
  Advanced approaches (e.g. Red-Black Trees can balance the three themself.)
- Available as `java.util.TreeSet` in the Java Library
- Uses the Red-Black approach $\Rightarrow$ no need to worry about balanced or not
- `TreeSet` uses interface `Comparable` for size comparison
  $\Rightarrow$ Let your class implement Comparable before storing them in `TreeSets`

The Program MapMain.java

```
Map<Integer,String> map = new HashMap<Integer,String>();
map.put(1,"Jonas");
map.put(8,"Jesper");
map.put(64,"Jens");
map.put(4,"Johan");

System.out.println("Value for 8: "+map.get(8)); // Jesper
System.out.println("Value for 4: "+map.get(4)); // Johan

map.put(8,"Johanna"); // Replaces Jesper as value for key 8
System.out.println("Value for 8: "+map.get(8)); // Johanna

Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    int key = it.next();
    String value = map.get(key);
    System.out.println("\t"+key+"\t"+value);
}
```

The `java.util.Map` Interface

```
V put(K key, V value) // Associates the specified value with the specified key
V get(Object key)     // The value to which the specified key is mapped

boolean containsKey(Object key)      // True if map contains a mapping for key.
boolean containsValue(Object value)  // True if map maps one or more keys to value.

boolean isEmpty()   // True if this map contains no key-value mappings.
Set<K> keySet()     // Returns a Set view of the contained keys.
int size()          // The number of key-value mappings in this map.
```

Notera:

- `put(key,value)` ⇒ adds a pair `(key,value)` to the mapping
- `put(k,v1)` followed by `put(k,v2)` ⇒ replaces the first mapping with `(k,v2)`
- `get(key)` returns the value associated with `key` (or `null` if no mapping found for `key`)

Three `Map` Classes

1. `java.util.HashMap`
   - Backed up by a hash table ⇒ `put, get, containsKey`, is very fast, $O(1)$
   - No key/value pair ordering ⇒ iteration (e.g., printing) in random order
2. `java.util.TreeMap`
   - Backed up by a binary search tree ⇒ `put, get, containsKey`, a bit slower than `HashMapt`
   - The key/value pairs are ordered using `Comparable` on the keys ⇒ iteration (e.g., printing) according to `Comparable` on the keys
3. `java.util.LinkedHashMap`
   - Backed up by a hash table and a list ⇒ `put, get, containsKey`, a bit slower than `HashMap` (but faster than `TreeMap`)
   - The key/value pairs are ordered in *insertion* order ⇒ iteration (e.g., printing) in *insertion* order

Exercise: Count Words

Count different words in text file using four different approaches

1. Transform arbitrary text file into a file `words.txt` that only contains words $\Rightarrow$ remove digits, brackets, commas, ... $\Rightarrow$ a sequence of words
2. Creat class `Word` representing one word. Methods `equals()`, `hashCode()`, `compareTo()` should consider `hello`, `Hello`, `HELLO` as all equal.
3. For each word in word.txt: 1) Create a `Word` object, and add it to 2) a HashSet and 3) a TreeSet. The size of the two sets should be the number of different words in `word.txt`. Iteration over TreeSet should give words in alphabetical order.
4. Create two classes `WordHashSet` and `WordTreeSet` that implements a given WordSet interface.
5. Repeat 3 using `WordHashSet` and `WordTreeSet`. The result should be the same.