



FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES (CTDL)

Semester 1, 2021/2022

Identifying unique elements



Java

Data Structures
and Algorithms

<http://www.javaguides.net>

Identifying unique elements in HashSet?

```
public static void main(String[] args) {  
    Set<String> set = new HashSet<String>();  
  
    set.add("Dai");  
    set.add("Hoc");  
    set.add("Nong");  
    set.add("Lam");  
    set.add("Nong");  
  
    System.out.println(set);  
}
```

- Output: [Lam, Dai, Hoc, Nong]

Data Structures
and Algorithms

<http://www.javaguides.net>

Identifying unique elements in HashSet? (cont.)

```
public static void main(String[] args) {  
    Set<Integer> set = new HashSet<Integer>();  
  
    set.add(1);  
    set.add(3);  
    set.add(2);  
    set.add(3);  
    set.add(4);  
  
    System.out.println(set);  
}
```

► Output: [1, 2, 3, 4]

Data Structures
and Algorithms

<http://www.javaguides.net>

Identifying unique elements in HashSet? (cont.)

```
public static void main(String[] args) {  
    Set<Circle> set = new HashSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```

<<Java Class>>

 Circle

lec6

 radius: int

 Circle(int)

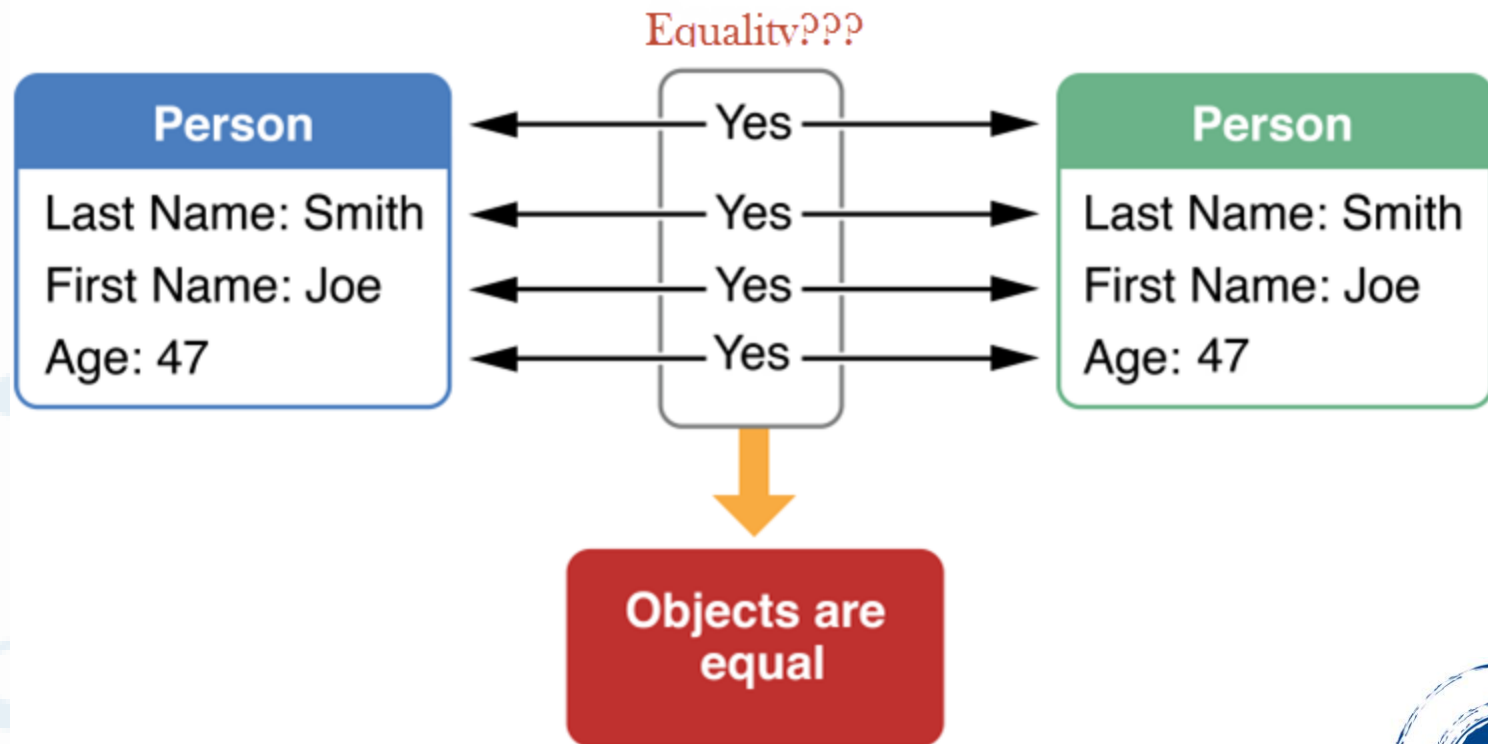
► Output:

[[Circle: 1], [Circle: 2], [Circle: 3], [Circle: 2]]

<http://www.javaguides.net>

How to identify unique elements?

- ▶ Object equality (using equals() method):



Implementing `equals()` method

<<Java Class>>

 **Circle**

lec8

• radius: int

 Circle(int)

 equals(Object):boolean

 toString():String

```
@Override
public boolean equals(Object obj) {
    if (obj == null || obj.getClass() != getClass()) {
        return false;
    } else {
        Circle that = (Circle) obj;
        return (this.radius == that.radius);
    }
}
```

Identifying unique elements in HashSet? (cont.)

```
public static void main(String[] args) {  
    Set<Circle> set = new HashSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```






► Output:

[[Circle: 1], [Circle: 2], [Circle: 3], [Circle: 2]]

<http://www.javaguides.net>

How to identify unique elements?

- ▶ Another approach, using [hashCode\(\)](#) method:

<<Java Class>>	
	Circle
lec6	
	radius: int
	Circle(int)
	hashCode():int
	toString():String

```
@Override
public int hashCode() {
    return Integer.hashCode(this.radius);
}
```

Identifying unique elements in HashSet? (cont.)

```
public static void main(String[] args) {  
    Set<Circle> set = new HashSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```

► Output:

[[Circle: 1], [Circle: 2], [Circle: 2], [Circle: 3]]

<http://www.javaguides.net>

How to identify unique elements?

- ▶ Combine both hashCode() and equals() methods:

```
@Override
public int hashCode() {
    return Integer.hashCode(this.radius);
}


@Override
public boolean equals(Object obj) {
    if (obj == null || obj.getClass() != getClass()) {
        return false;
    } else {
        Circle that = (Circle) obj;
        return (this.radius == that.radius);
    }
}
```

<<Java Class>>


 Circle

lec6

■ radius: int

 Circle(int)

 hashCode():int

 equals(Object):boolean

 toString():String

Identifying unique elements in HashSet? (cont.)

```
public static void main(String[] args) {  
    Set<Circle> set = new HashSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```

► Output:

[[Circle: 1], [Circle: 2], [Circle: 3]]

<http://www.javaguides.net>


Identifying unique elements in TreeSet?

```
public static void main(String[] args) {  
    TreeSet<Circle> set = new TreeSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```

<<Java Class>>

 Circle

lec6

 radius: int






 Circle(int)

- ▶ Output: java.lang.ClassCastException: lec6.Circle cannot be cast to java.lang.Comparable

➔ Implement **Comparable** interface and override **compareTo** method!!!

Identifying unique elements in TreeSet?

```
public static void main(String[] args) {  
    TreeSet<Circle> set = new TreeSet<Circle>();  
  
    set.add(new Circle(1));  
    set.add(new Circle(2));  
    set.add(new Circle(3));  
    set.add(new Circle(2));  
  
    System.out.println(set);  
}
```

<<Java Class>>	
	Circle
lec8	
	radius: int
	Circle(int)
	toString():String
	compareTo(Circle):int

► Output:

[[Circle: 1], [Circle: 2], [Circle: 3]]

Hash in Java Collection Framework

- ▶ **HashSet**, **LinkedHashSet** are not collections that make use of **equals()** and **hashCode()**.
- ▶ **HashMap**, **Hashtable**, and **LinkedHashMap** also require these methods.

Java
Data Structures
and Algorithms

<http://www.javaguides.net>

How to implement `hashCode()` method?



Java

Data Structures
and Algorithms

<http://www.javaguides.net>

Hashing

- ▶ **hash**: To map a value to a specific integer index.
 - **hash table**: An array that stores elements via hashing.
 - The internal data structure used by `HashSet` and `HashMap`.
 - **hash function**: An algorithm that maps values to indexes.
 - A possible hash function for integers: $HF(i) \rightarrow i \% \text{length}$

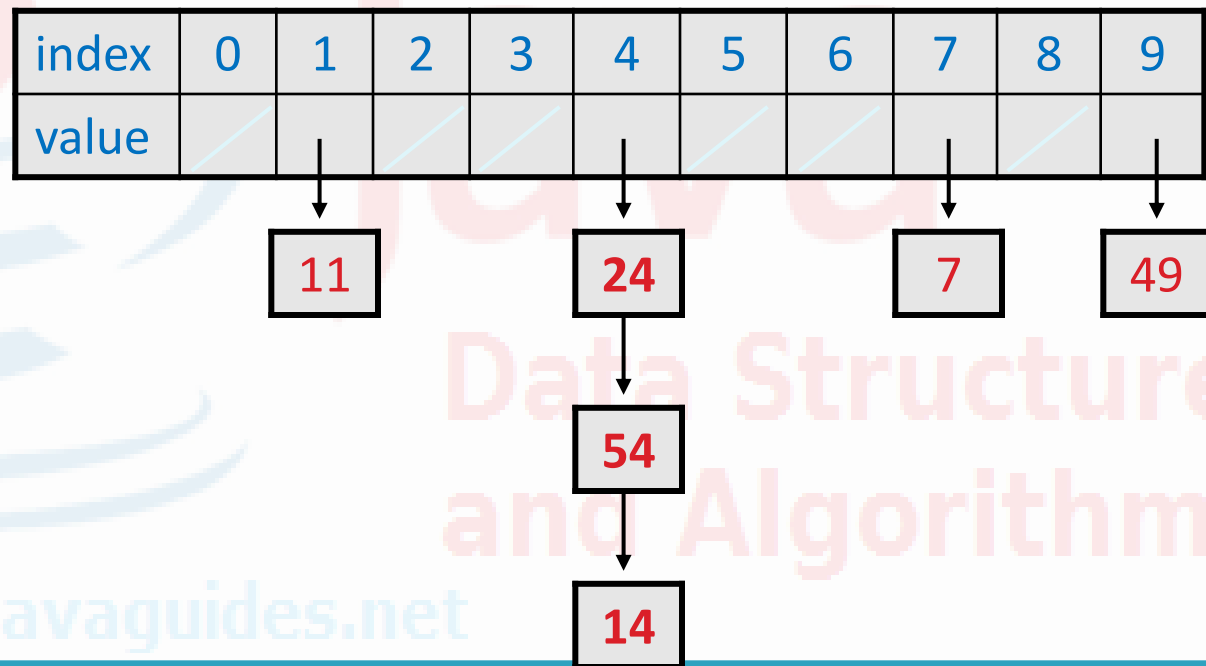
```
set.add(11);           // 11 % 10 == 1
set.add(49);           // 49 % 10 == 9
set.add(24);           // 24 % 10 == 4
set.add(7);            // 7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

- What is a hash function for a string? for other kinds of objects?

Chaining

- ▶ **Chaining**: Resolving collisions by storing a list at each index.
 - Add/search/remove must traverse lists, but the lists are short.
 - Impossible to "run out" of indexes, unlike with probing.
 - Alternative to chaining: *probing* (choosing the next available index).



The **hashCode** method

- ▶ From the **Object** class:

```
public int hashCode()
```

Returns an integer hash code for this object.

- We can call **hashCode** on *any object* to find the index where it "prefers" to be placed in a hash table.

Data Structures
and Algorithms

<http://www.javaguides.net>

Implementing hashCode

- ▶ hashCode's implementation depends on the object's type/state.
 - A String's **hashCode** method adds the ASCII values of its letters.
 - A Point's **hashCode** produces a weighted sum of its x/y coordinates.
 - A Double's **hashCode** converts the number into bits and returns that.
 - A collection's **hashCode** combines the hash codes of its elements.
- ▶ You can override **hashCode** in your classes.
 - *Tip*: Always override **hashCode** when you override **equals**.

hashCode for String object

- ▶ The hash code for a String object is computed as:

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

where :

$s[i]$ – is the i^{th} character of the string

n – is the length of the string, and

\wedge – indicates exponentiation

The hashCode contract

- ▶ The general contract of **hashCode** is that it must be:
 - *Self-consistent* (produces the same results on each call):
 - `o.hashCode() == o.hashCode()`
...so long as `o` doesn't change between the calls
 - *Consistent with equality*.
 - `a.equals(b)` implies that
`a.hashCode() == b.hashCode()`
 - `!a.equals(b)` does NOT necessarily imply that
`a.hashCode() != b.hashCode()` (*why not?*)



hashCode tricks

- ▶ If one of your object's fields is an object, call its **hashCode**:

```
public int hashCode() {           // TimeSpan
    return 65531 * amPm.hashCode() + ...;
}
```

- ▶ To incorporate an array, use **Arrays.hashCode**.

```
private String[] addresses;

public int hashCode() {
    return 3137 * Arrays.hashCode(addresses) + ...
} // also Arrays.deepHashCode for multi-dim arrays
```

hashCode tricks 2

- ▶ To incorporate a **double** or **boolean**, use the **hashCode** method from the **Double** or **Boolean** wrapper classes:

```
public int hashCode() {                // BankAccount
    return 37 * new Double(balance).hashCode() +
           new Boolean(isCheckingAccount).hashCode() + ...;
}
```

- ▶ If your hash code is expensive to compute, consider caching it.

```
private int myHashCode = ...;          // pre-compute once

public int hashCode() {
    return myHashCode;
}
```


Notes on `hashCode()` and `equals()`

➤ Hashcode comparisons

If the <code>hashCode()</code> comparison ...	Then...
returns true	execute <code>equals()</code>
returns false	do not execute <code>equals()</code>

Data Structures
and Algorithms

<http://www.javaguides.net>

Notes on hashCode() and equals()

➤ Object comparison with hashCode()

When the hashCode comparison returns ...	The equals() method should return ...
true	true or false
false	false

java
Data Structures
and Algorithms

<http://www.javaguides.net>

Notes on `hashCode()` and `equals()`

➤ Object comparison with `equals()`

When the <code>equals()</code> method returns ...	The <code>hashCode()</code> method should return ...
true	true
false	true or false

java
Data Structures
and Algorithms

<http://www.javaguides.net>

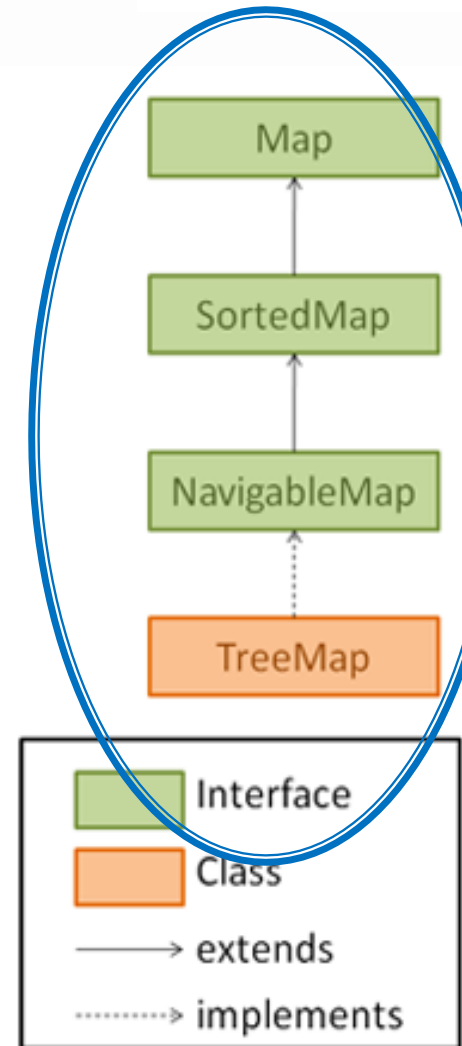
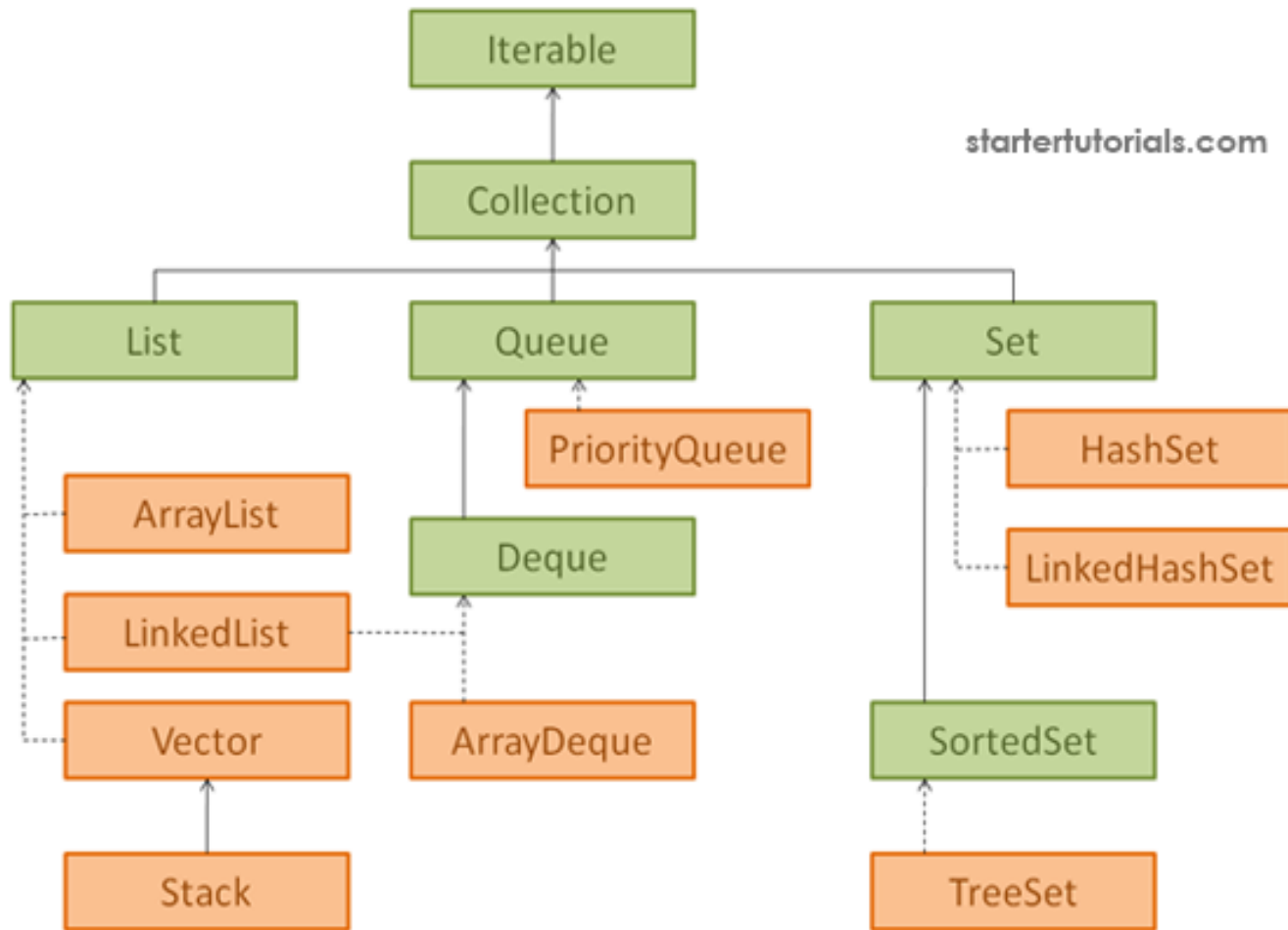
Section complete

Next Section ►►

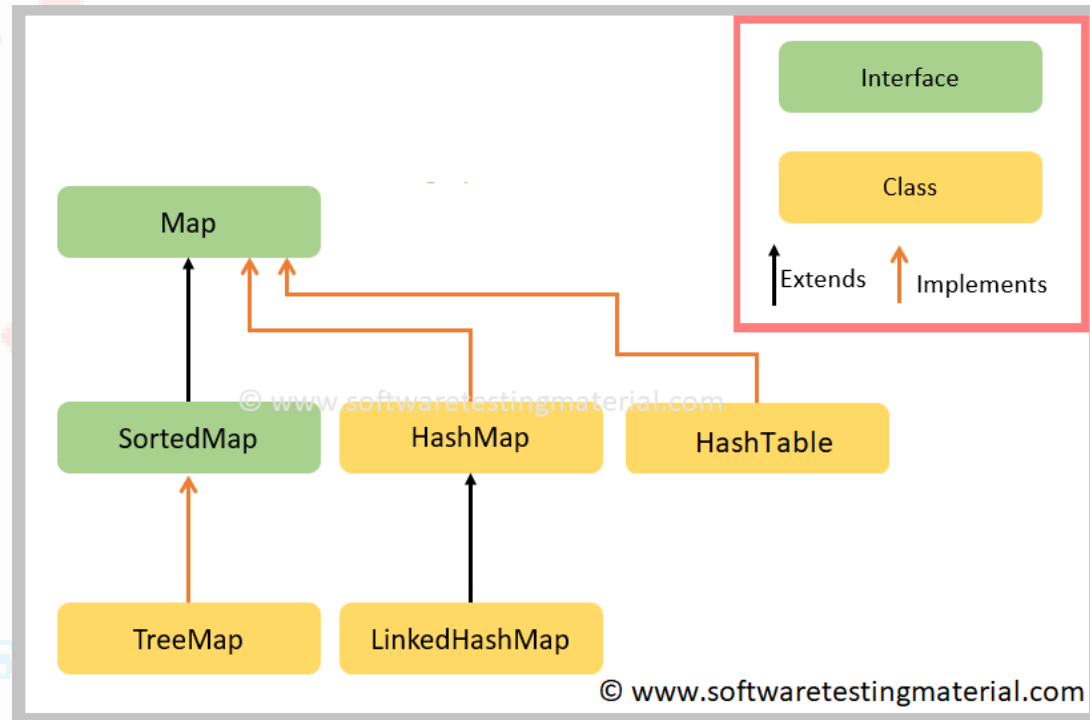
Data Structures
and Algorithms

<http://www.javaguides.net>

Java Collection framework



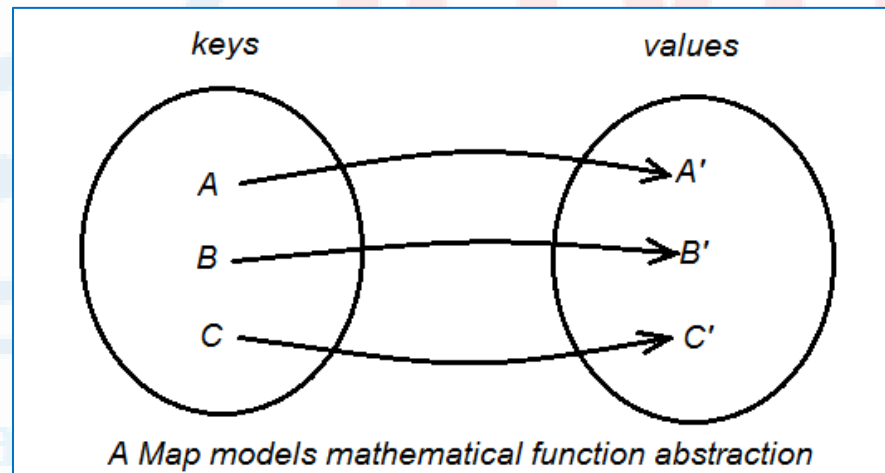
MAP



© www.softwaretestingmaterial.com

Definition

- ▶ A set is a collection that efficiently stores and retrieves values based upon a uniquely identifying *search key* for each.
- ▶ A map stores **(k,v) pairs**, *entries*, where
 - *k* is the key, and required to be *unique*
 - *v* is its corresponding value.



Methods

`size()`: Returns the number of entries in M .

`isEmpty()`: Returns a boolean indicating whether M is empty.

`get(k)`: Returns the value v associated with key k , if such an entry exists; otherwise returns null.

`put(k, v)`: If M does not have an entry with key equal to k , then adds entry (k, v) to M and returns null; else, replaces with v the existing value of the entry with key equal to k and returns the old value.

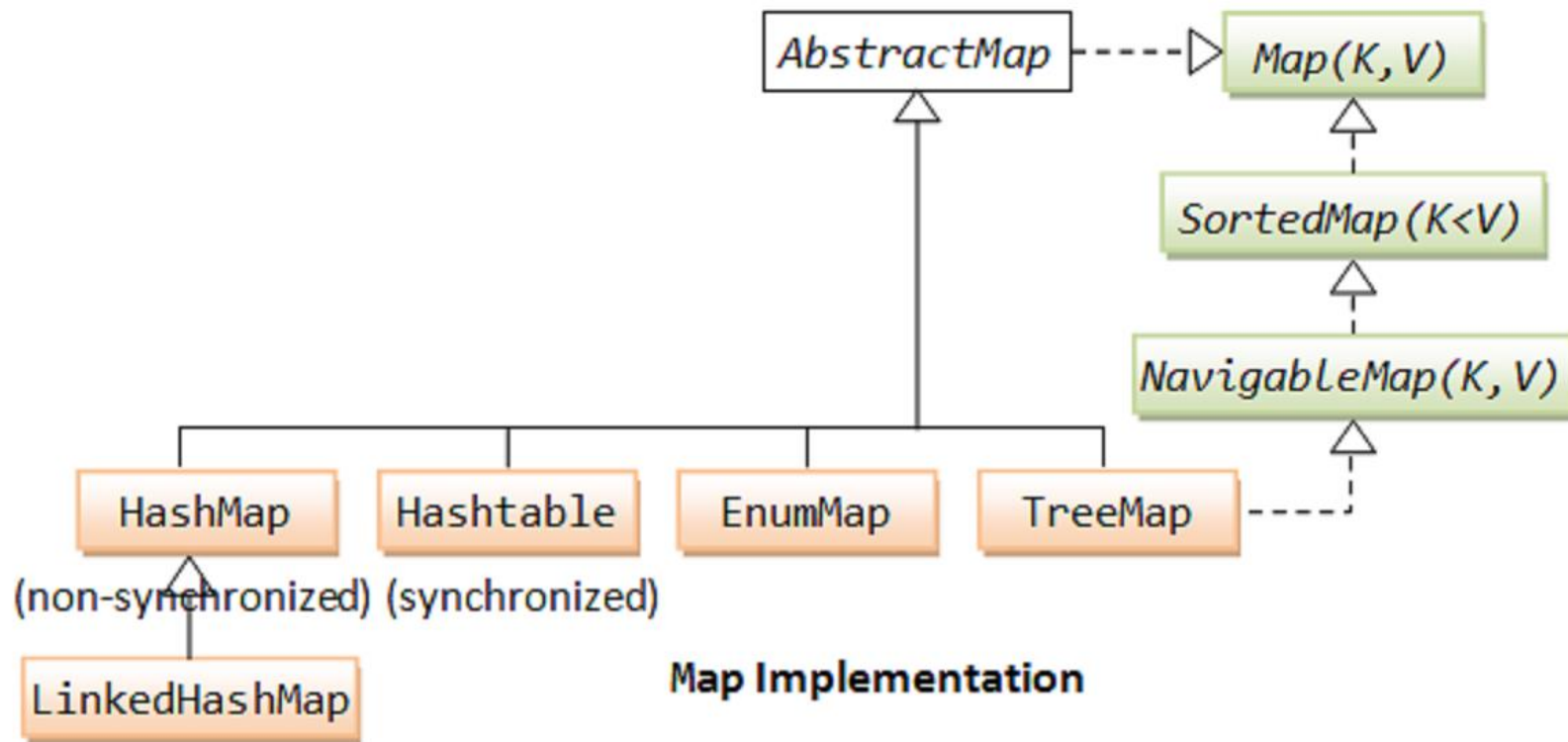
`remove(k)`: Removes from M the entry with key equal to k , and returns its value; if M has no such entry, then returns null.

`keySet()`: Returns an iterable collection containing all the keys stored in M .

`values()`: Returns an iterable collection containing all the *values* of entries stored in M (with repetition if multiple keys map to the same value).

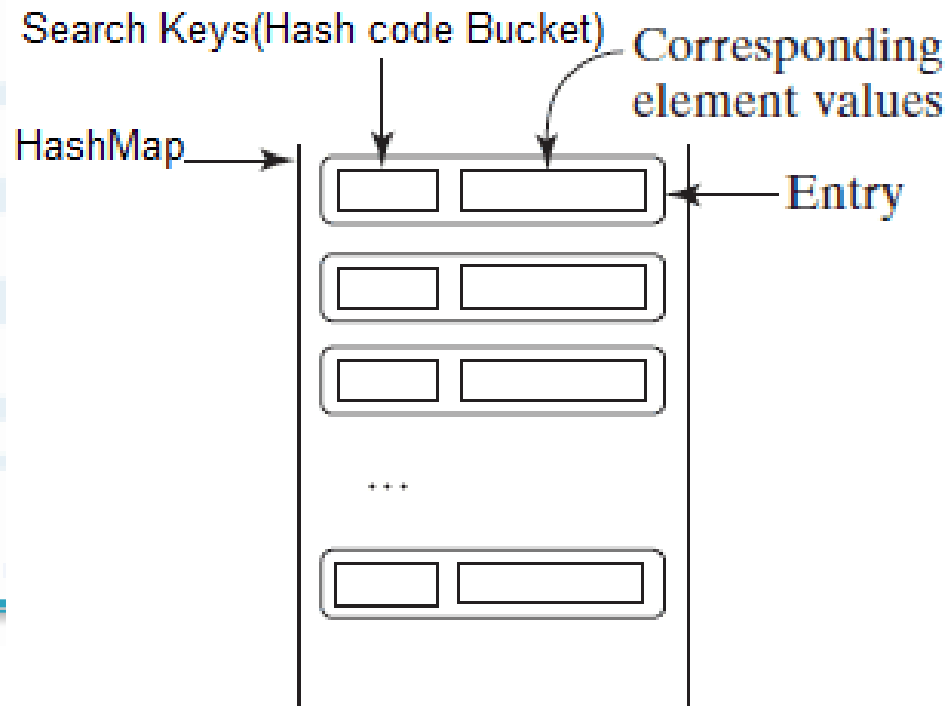
`entrySet()`: Returns an iterable collection containing all the key-value entries in M .

Map in Java Collection Framework



HashMap

- ▶ A hash map hashes the keys.
- ▶ Hashing is a bit faster, and it is the preferred choice if you don't need to visit the keys in sorted order.
- ▶ **Keys** and **values** in a HashMap are **actually** objects.



HashMap (cont.)

► Constructors:

◦ `HashMap()`:

- It is the default constructor which creates an instance of `HashMap` with initial capacity 16 and load factor 0.75.

◦ `HashMap(int initialCapacity)`:

- It creates a `HashMap` instance with specified initial capacity and load factor 0.75.

◦ `HashMap(int initialCapacity, float loadFactor)`:

- It creates a `HashMap` instance with specified initial capacity and specified load factor.

◦ `HashMap(Map map)`:

- It creates instance of `HashMap` with same mappings as specified map.

Methods

Sr.No.	Method & Description
1	void clear() Removes all mappings from this map.
2	Object clone() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
3	boolean containsKey(Object key) Returns true if this map contains a mapping for the specified key.
4	boolean containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
5	Set entrySet() Returns a collection view of the mappings contained in this map.
6	Object get(Object key) Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.

Methods (cont.)

Sr.No.	Method & Description
7	boolean isEmpty() Returns true if this map contains no key-value mappings.
8	Set keySet() Returns a set view of the keys contained in this map.
9	Object put(Object key, Object value) Associates the specified value with the specified key in this map.
10	putAll(Map m) Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
11	Object remove(Object key) Removes the mapping for this key from this map if present.
12	int size() Returns the number of key-value mappings in this map.
13	Collection values() Returns a collection view of the values contained in this map.

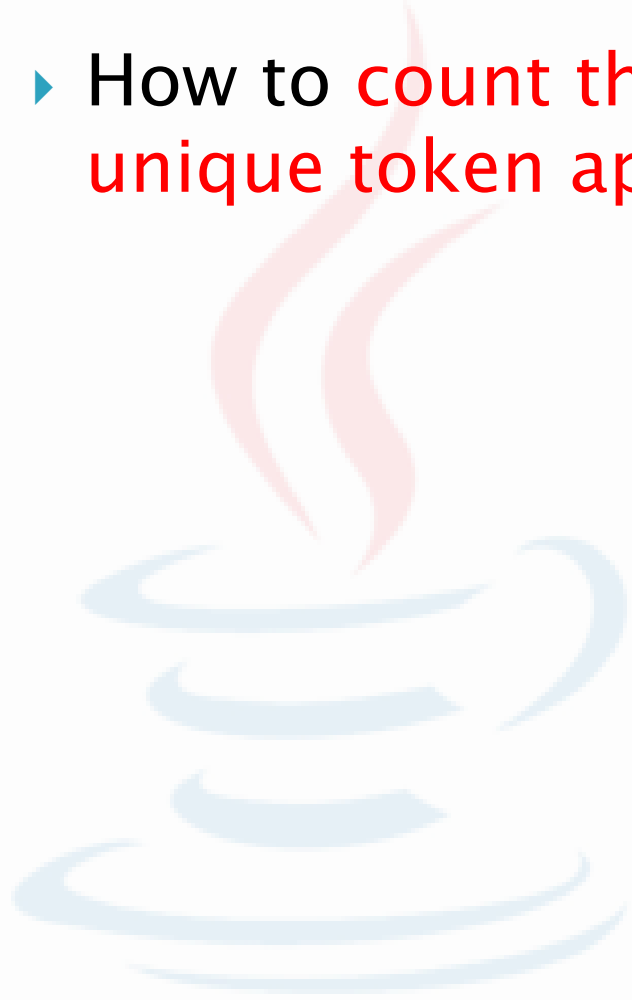
Using HashMap

```
public static void main(String[] args) {  
    HashMap<String, Integer> map = new HashMap<String, Integer>();  
  
    map.put("Dai", 1);  
    map.put("Hoc", 1);  
    map.put("Nong", 2);  
    map.put("Lam", 2);  
    map.put("Dai", 2);  
  
    System.out.println("Size: " + map.size()); //?? 4 OR 5??  
    System.out.println("HashMap: " + map);  
}
```

- ▶ Size: 4
- ▶ HashMap: {Dai=2, Hoc=1, Lam=2, Nong=2}

Example

- ▶ How to count the number of times each unique token appears in a given text file?



Java

Data Structures
and Algorit



<http://www.javaguides.net>

Notes on HashMap

- ▶ HashMap contains values based on the key.
- ▶ HashMap contains only unique keys.
- ▶ HashMap may have one null key and multiple null values.
- ▶ HashMap is non synchronized.
- ▶ HashMap maintains no order.
- ▶ The initial default capacity is 16 with a load factor of 0.75.

Java
Data Structures
and Algorithms

<http://www.javaguides.net>

LinkedHashMap



Java

Data Structures
and Algorithms

<http://www.javaguides.net>

LinkedHashMap

- ▶ **LinkedHashMap** is just like **HashMap** with an additional feature of **maintaining an order of elements inserted into it**.
- ▶ This implementation **differs from HashMap in that it maintains a doubly-linked list** running through all of its entries.

Data Structures
and Algorithms

<http://www.javaguides.net>

LinkedHashMap (cont.)

▶ **LinkedHashMap():**

- construct a default LinkedHashMap constructor.

▶ **LinkedHashMap(int capacity):**

- initialize a particular LinkedHashMap with a specified capacity.

▶ **LinkedHashMap(Map m_a_p):**

- initialize a particular LinkedHashMap with the elements of the specified map.

<http://www.javaguides.net>

LinkedHashMap (cont.)

▶ **LinkedHashMap(int capacity, float loadFactor):**

- initialize both the capacity and load factor for a LinkedHashMap.

▶ **LinkedHashMap(int capacity, float loadFactor, boolean order):**

- initialize both the capacity and load factor for a LinkedHashMap along with whether to follow the insertion order or not.
 - True: access order.
 - False: insertion order.

<http://www.javaguides.net>

Using LinkedHashMap

```
public static void main(String[] args) {  
    LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();  
    lhm.put("one", "Dai");  
    lhm.put("two", "Hoc");  
    lhm.put("four", "Nong");  
    lhm.put("five", "Lam");  
  
    // It prints the elements in same order  
    // as they were inserted  
    System.out.println(lhm);  
}
```

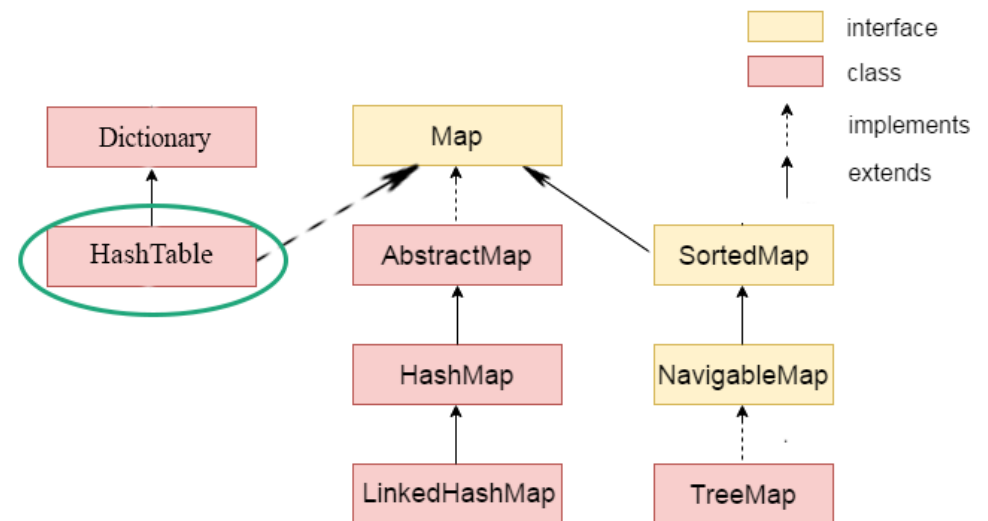
► Output:

{one=Dai, two=Hoc, four=Nong, five=Lam}

Notes on LinkedHashMap

- ▶ LinkedHashMap contains values based on the key.
- ▶ LinkedHashMap contains unique elements.
- ▶ LinkedHashMap may have **one null key** and multiple null values.
- ▶ LinkedHashMap is **non synchronized**.
- ▶ LinkedHashMap maintains **insertion order**.
- ▶ The initial default capacity is 16 with a load factor of 0.75.

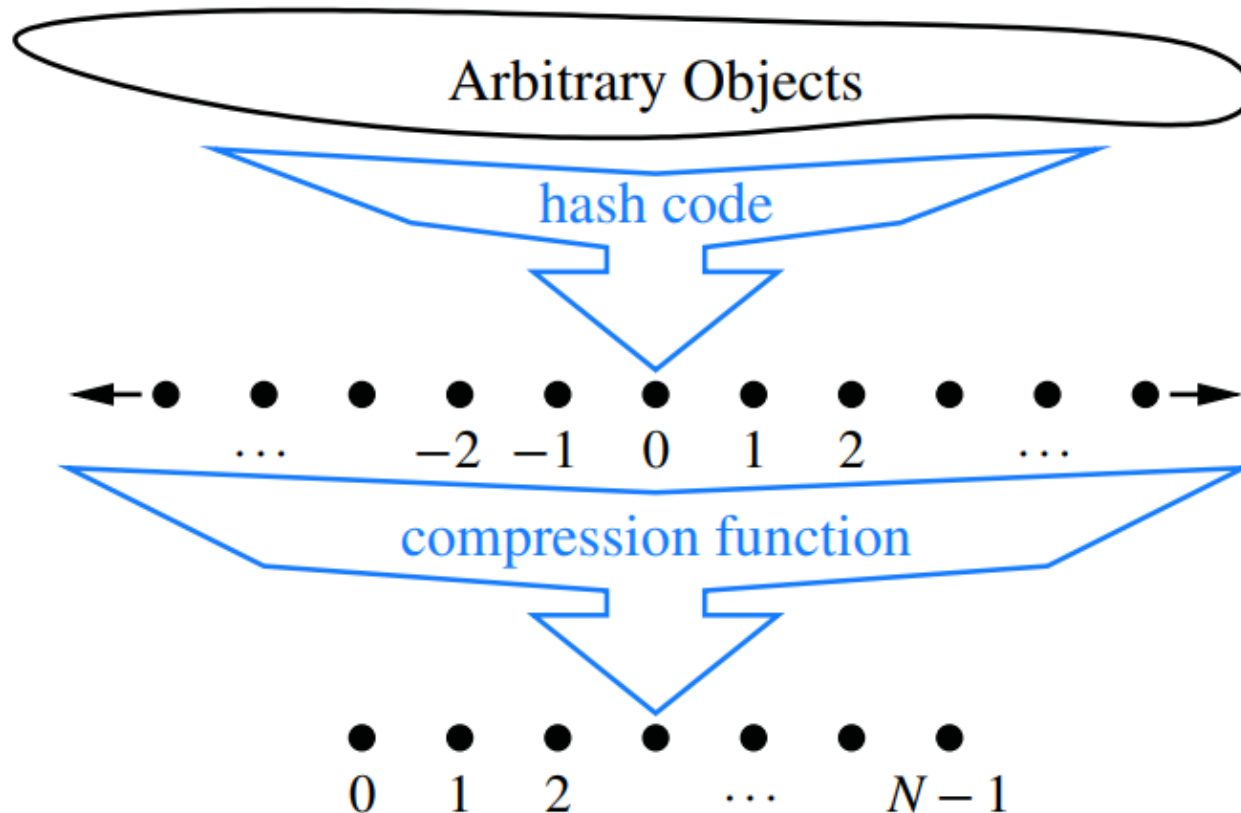
Hashtable



Hashtable (cont.)

- ▶ The fundamental idea of a hash table is to **store entries in an array**.
- ▶ If the **physical size** of array is n , and h is a hash function, then an entry with key k is stored at index $h(k) \bmod n$.
- ▶ Then, looking up key k just involves computing $h(k) \bmod n$ again and looking at the index in the array → **very fast**.
- ▶ It is possible for $h(k_1) \bmod n$ and $h(k_2) \bmod n$ to be the same for two different keys k_1 and k_2 . That is called a **collision**.
- ▶ There are several ways of dealing with collisions.
(i.e. **Chaining**)

Hash function



Common hash functions

Division method: $h(k) = k \bmod m$

Example: $m=1024$, $k = 2058 \rightarrow h(k) = 10$

- don't use a power of 2
 $m = 2^p \rightarrow h(k)$ depends only on the p least significant bits
- use $m = \text{prime number}$, not near any power of two

Multiplication method: $h(k) = \lfloor m (kA \bmod 1) \rfloor$

1. $0 < A < 1$ is a constant
 2. compute kA and extract the fractional part
 3. multiply this value with m and then take the floor of the result
- Advantage: choice of m is not so important, can choose $m = \text{power of } 2$

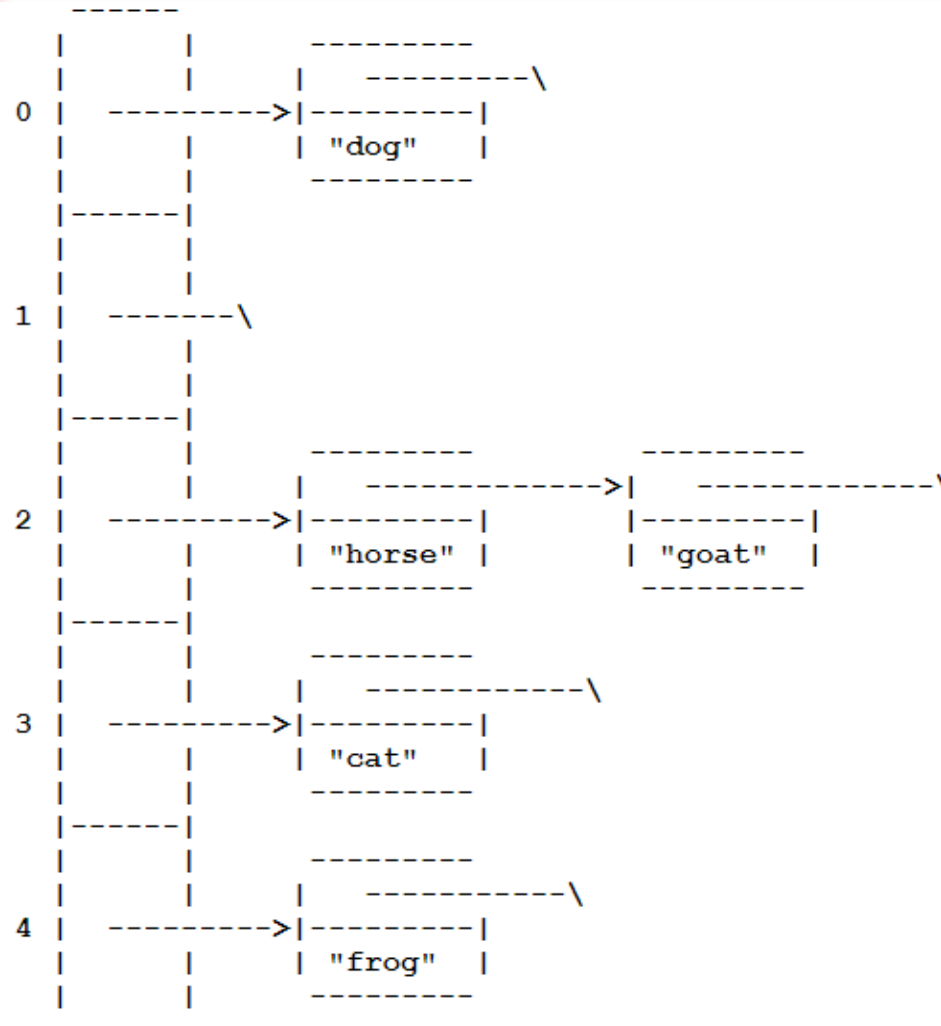
```
Math.floor(60984.1)=60984.0  
Math.floor(-497.99)=-498.0  
Math.floor(0)=0.0
```

Dealing with collisions by chaining

- ▶ Storing a linked list of values at each index in the array
- ▶ For example, suppose that the physical size of the array is 5. Keys: "cat", "dog", "frog", "goat" and "horse".
- ▶ The hash function yields the following values:
 - $h(\text{"cat"}) = 63$
 - $h(\text{"dog"}) = 35$
 - $h(\text{"frog"}) = 79$
 - $h(\text{"goat"}) = 72$
 - $h(\text{"horse"}) = 117$

Dealing with collisions by chaining

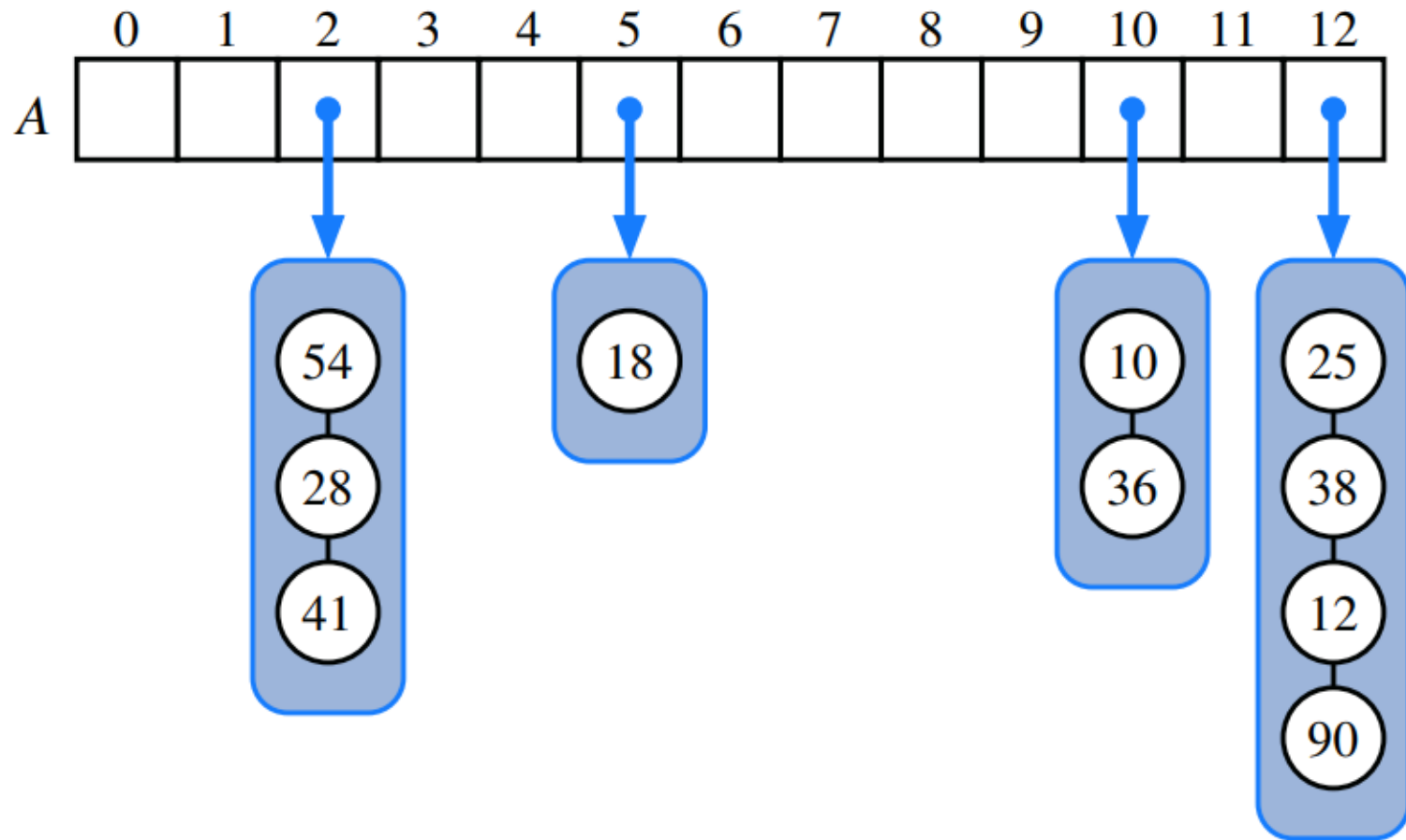
- ▶ Then the **hashtable** stores the keys as follows.



captures
ithms

Hash function

Example:



Working of Hashtable

- ▶ **Hashtable performance** depends on various parameters:
 - Initial capacity, load factor, size and threshold value are the parameters which affect Hashtable performance.
- ▶ **Initial capacity:**
 - This is the capacity of Hashtable to store number of key value pairs when it is instantiated. **Default capacity is 11.**
- ▶ **Load Factor:**
 - A parameter responsible to determine when to increase size of Hashtable. **Default load factor is 0.75.**
- ▶ **Size:**
 - Number of key value pairs in Hashtable.
- ▶ **Threshold value:**
 - When number of key value pairs is more than threshold value, then Hashtable is resized

Hashtable:constructor

- ▶ **Hashtable():**
 - This is the default constructor to the Hashtable it instantiates the Hashtable class.
- ▶ **Hashtable(int size):**
 - It creates a hash table that has initial size specified by size..
- ▶ **Hashtable(int size, float loadFactor):**
 - This creates a Hashtable that has an initial size specified by size and a load factor specified by loadFactor.
 - This factor must be between 0.0 and 1.0, and it determines how full the Hashtable can be before it is resized upward.
- ▶ **Hashtable(Map < ? extends k, ? extends v > t):**
 - This constructs a Hashtable with the given mappings.

Using Hashtable

```
public static void main(String[] args) {  
    // creating a hash table  
    Hashtable<Integer, String> h = new Hashtable<Integer, String>();  
  
    h.put(3, "Dai");  
    h.put(2, "Hoc");  
    h.put(1, "Nong");  
    h.put(4, "Lam");  
  
    System.out.println(h);  
}
```

Output:

{4=Lam, 3=Dai, 2=Hoc, 1=Nong}

Notes on Hashtable

- ▶ It is synchronized.
- ▶ Hashtable may **have not have any null key or value.**
- ▶ Hashtable is slow as it is synchronized.
- ▶ Hashtable is internally synchronized and can't be unsynchronized.
- ▶ Hashtable is **traversed by Enumerator and Iterator.**
- ▶ Hashtable inherits **Dictionary** class
- ▶ We can make the HashMap as synchronized by calling this code:

```
Map m =Collections.synchronizedMap (hashMap);
```

SortedMap



Java

Data Structures
and Algorithms

<http://www.javaguides.net>

Methods

- ▶ **firstEntry()**:
 - Returns the entry with smallest key value (or null, if the map is empty).
- ▶ **lastEntry()**:
 - Returns the entry with largest key value (or null, if the map is empty).
- ▶ **ceilingEntry(k)**:
 - Returns the entry with the least key value greater than or equal to k (or null, if no such entry exists).
- ▶ **floorEntry(k)**:
 - Returns the entry with the greatest key value less than or equal to k (or null, if no such entry exists).

Methods (cont.)

▶ **lowerEntry(k):**

- Returns the entry with the greatest key value strictly less than k (or null, if no such entry exists).

▶ **higherEntry(k):**

- Returns the entry with the least key value strictly greater than k (or null if no such entry exists).

▶ **subMap(k1, k2):**

- Returns an iteration of all entries with key greater than or equal to k1, but strictly less than k2.

TreeMap

► Constructors:

- **TreeMap():**

- Constructs an empty tree map that will be sorted by using the natural order of its keys.

- **TreeMap(Comparator comp):**

- Constructs an empty tree-based map that will be sorted by using the Comparator comp.

- **TreeMap(Map m):**

- Initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys.

- **TreeMap(SortedMap sm):**

- Initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

Using TreeMap

```
public static void main(String[] args) {  
    TreeMap<String, Integer> map = new TreeMap<String, Integer>();  
  
    map.put("Dai", 1);  
    map.put("Hoc", 1);  
    map.put("Nong", 2);  
    map.put("Lam", 2);  
    map.put("Dai", 2);  
  
    System.out.println("Size: " + map.size()); //?? 4 OR 5??  
    System.out.println("TreeMap: " + map);  
}
```

- ▶ Size: 4
- ▶ HashMap: {Dai=2, Hoc=1, Lam=2, Nong=2}

<http://www.javaguides.net>

Notes on TreeMap

- ▶ TreeMap contains values based on the key.
- ▶ TreeMap contains only **unique elements**.
- ▶ TreeMap cannot have **a null key but can have multiple null values**.
- ▶ TreeMap is **non synchronized**.
- ▶ TreeMap **maintains ascending order**.

<http://www.javaguides.net>

HashMap vs TreeMap

Property \ Map	HashMap	TreeMap
Ordering	not guaranteed	sorted, natural ordering
get / put / remove complexity	$O(1)$	$O(\log(n))$
Inherited interfaces	Map	Map NavigableMap SortedMap
NULL values / keys	allowed	only values

and Algorithms

<http://www.javaguides.net>

Initial capacity and Load factor

▶ Initial capacities:

- ArrayList-10
- Vector-10
- HashSet-16
- HashMap-16
- Hashtable-11

▶ Load factor: 0.75

Java

Data Structures
and Algorithms

<http://www.javaguides.net>



FACULTY OF INFORMATION TECHNOLOGY

