CS 91 USACO

Bronze Division

Unit 2: 1-D Data Structures

LECTURE 8: SETS AND MAPS

DR. ERIC CHOU

IEEE SENIOR MEMBER

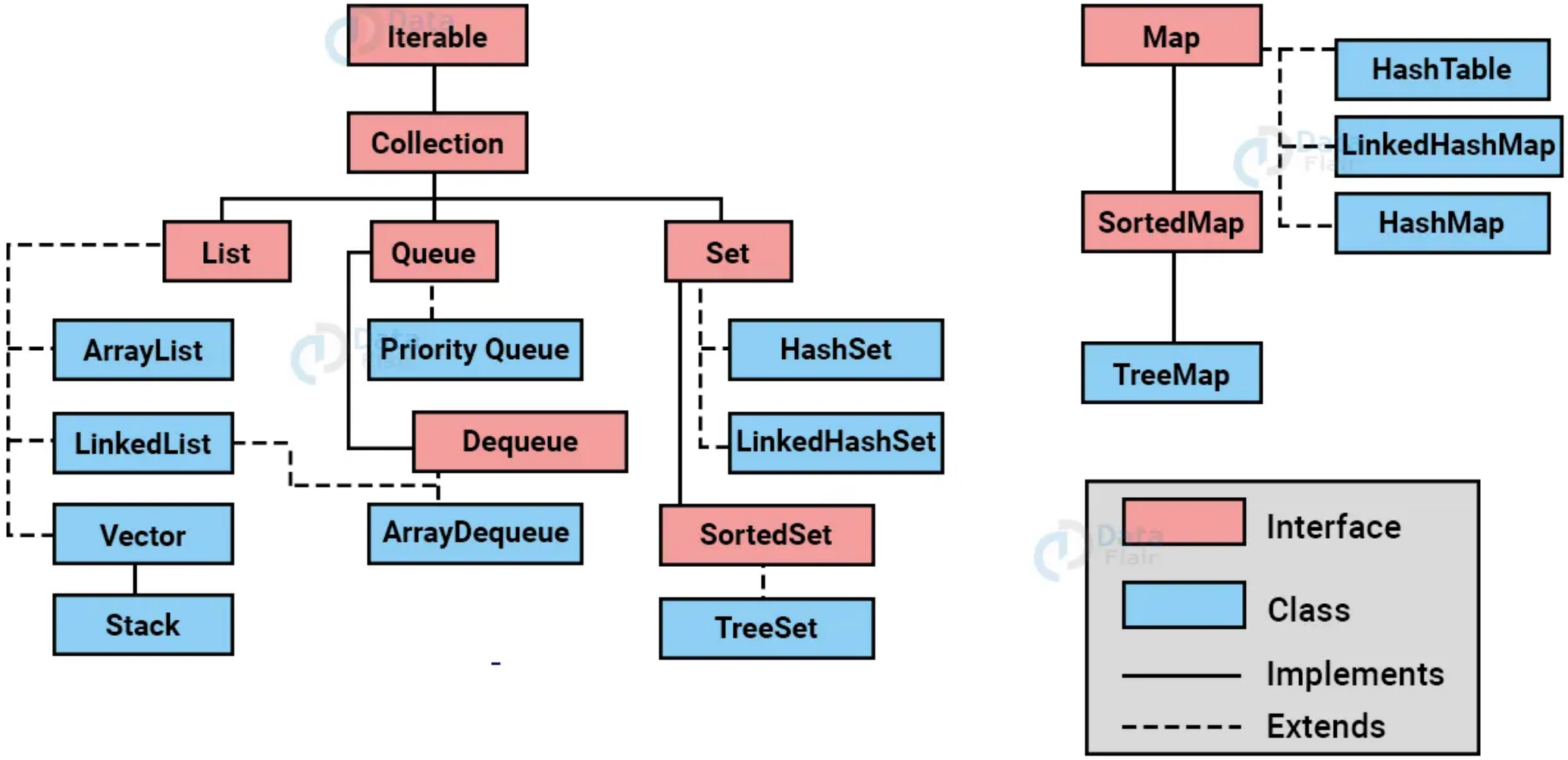
Objectives

- Set, Map
- TreeSet
- HashSet
- HashMap
- Application of Maps and Sets: (1) Content-addressable Memory (Reverse Array), (2) Random Permutation (Random Seating)

Maps

SECTION 1

Hierarchy of Collection Framework in Java

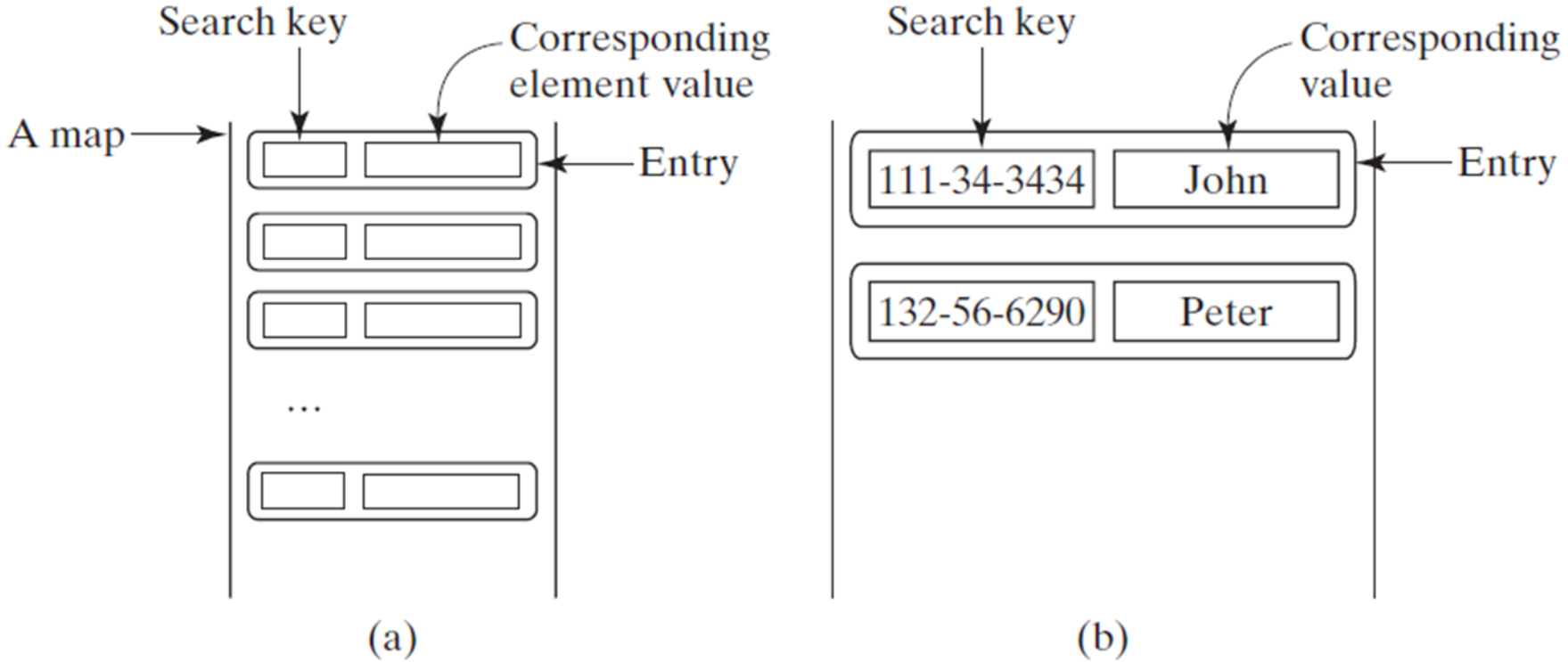


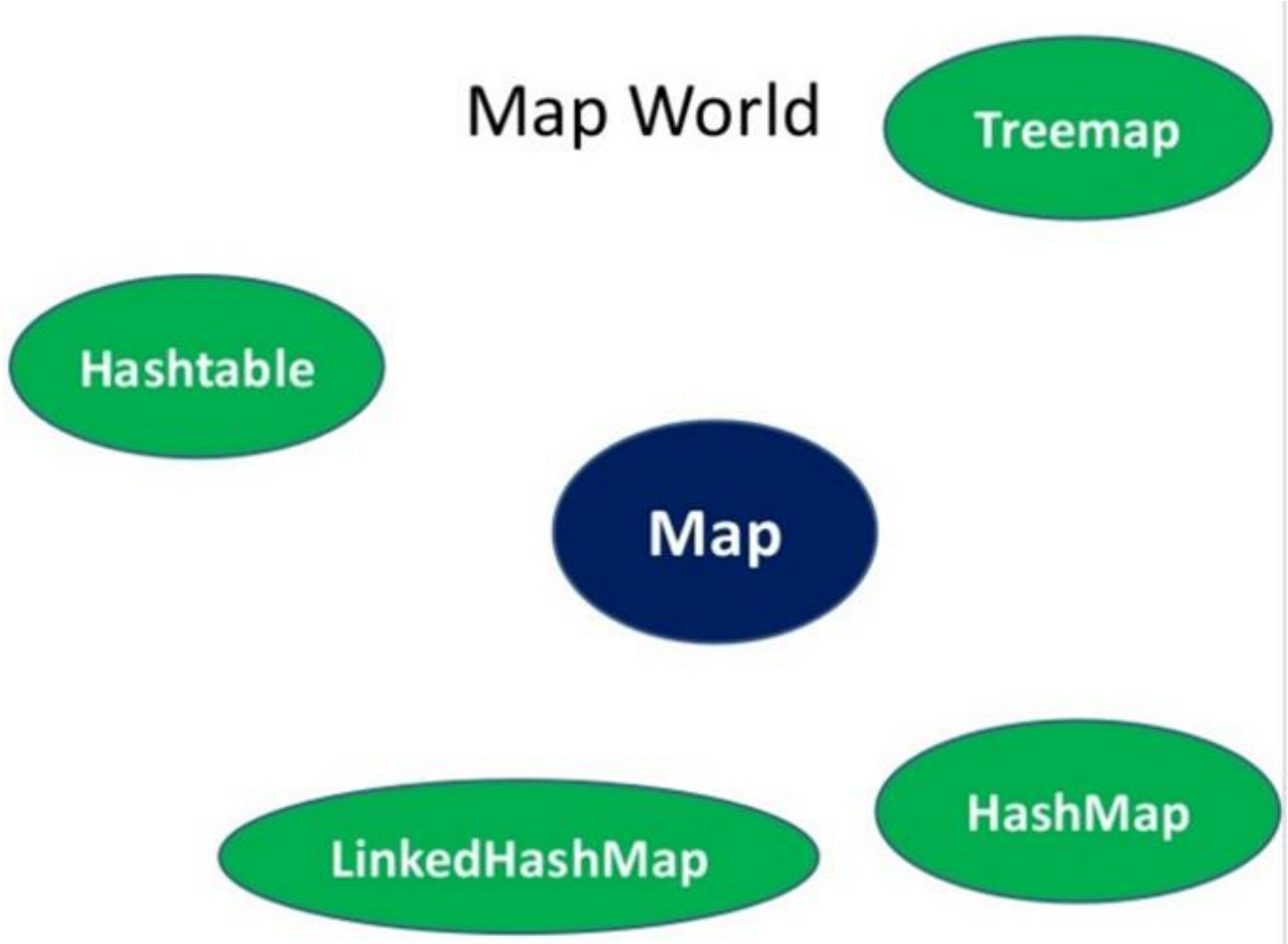
Map<K, V> Interface

lists of (key, value) pairs

Map means Mapping

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.





Map Interface and Map Classes

Class Diagram for Map<K, V>

java.util.Map<K, V>	
+clear(): void	Removes all mappings from this map.
+containsKey(key: Object): boolean	Returns true if this map contains a mapping for the specified key.
+containsValue(value: Object): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set	Returns a set consisting of the entries in this map.
+get(key: Object): V	Returns the value for the specified key in this map.
+isEmpty(): boolean	Returns true if this map contains no mappings.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts a mapping in this map.
+putAll(m: Map): void	Adds all the mappings from m to this map.
+remove(key: Object): V	Removes the mapping for the specified key.
+size(): int	Returns the number of mappings in this map.
+values(): Collection<V>	Returns a collection consisting of the values in this map.

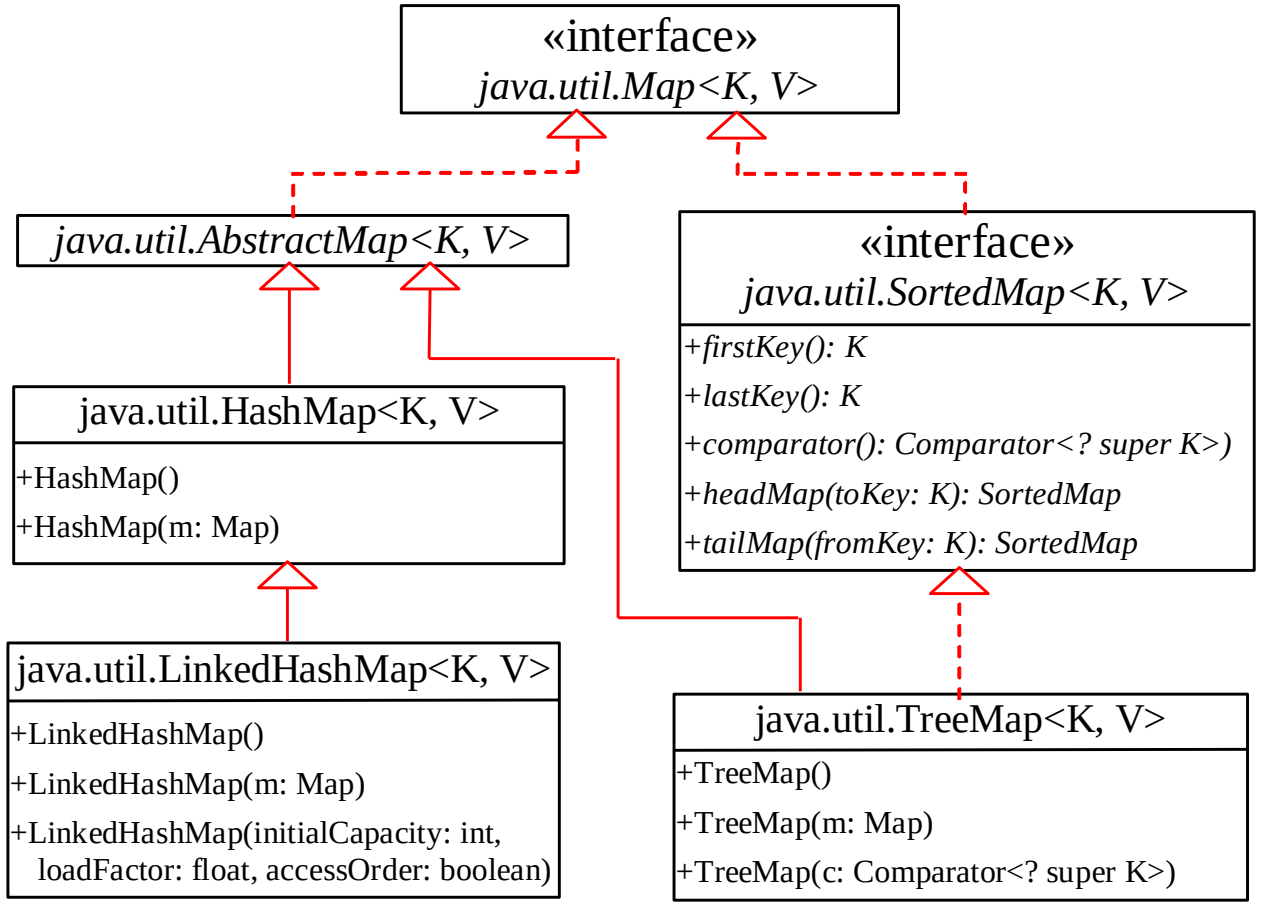
Dictionary Class (abstract class)

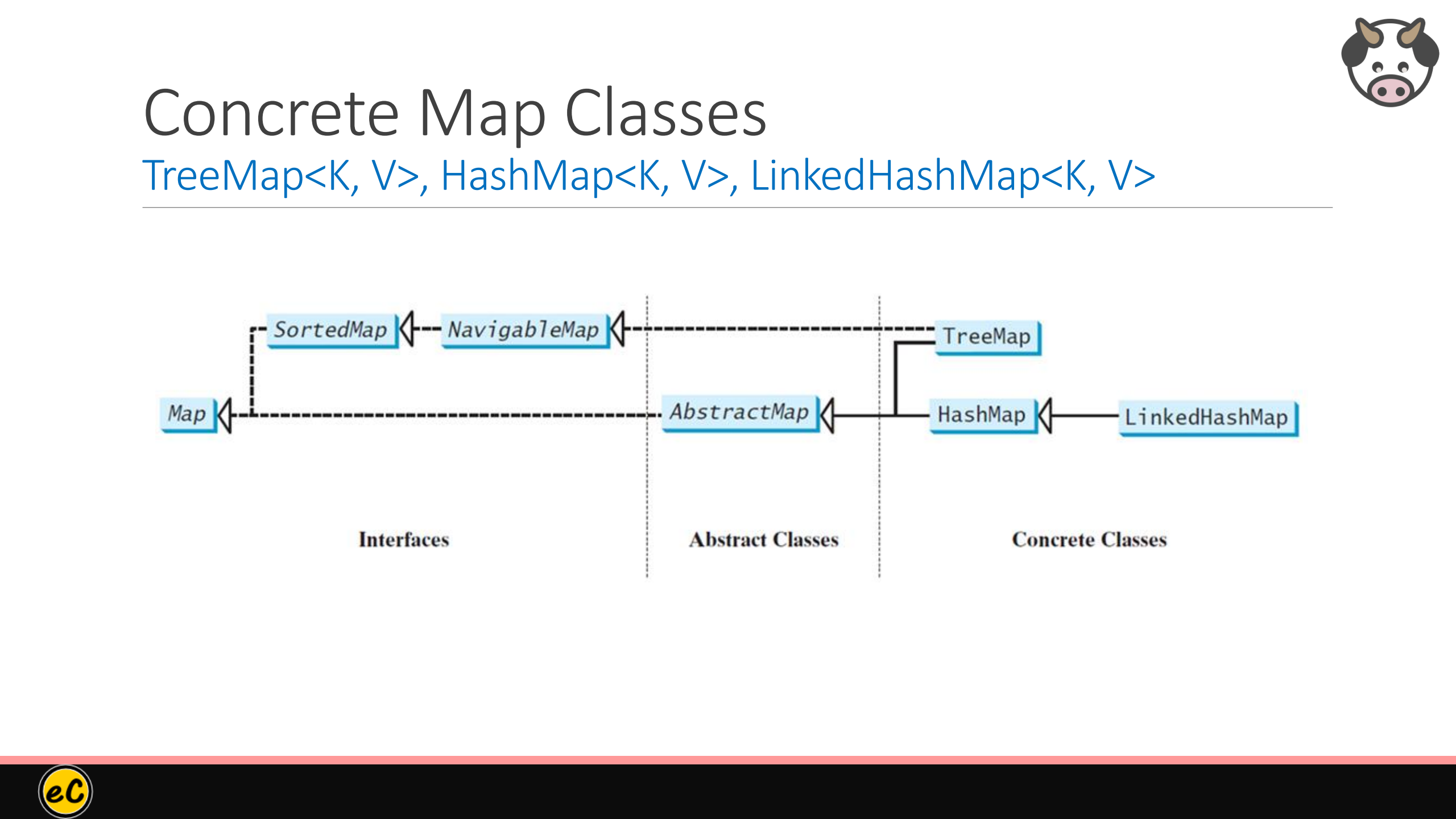
Dictionary<K, V> // java.util.Dictionary Out-Of-Date (legacy class)

- Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
- Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

Concrete Map Classes

TreeMap<K, V>, HashMap<K, V>, LinkedHashMap<K, V>



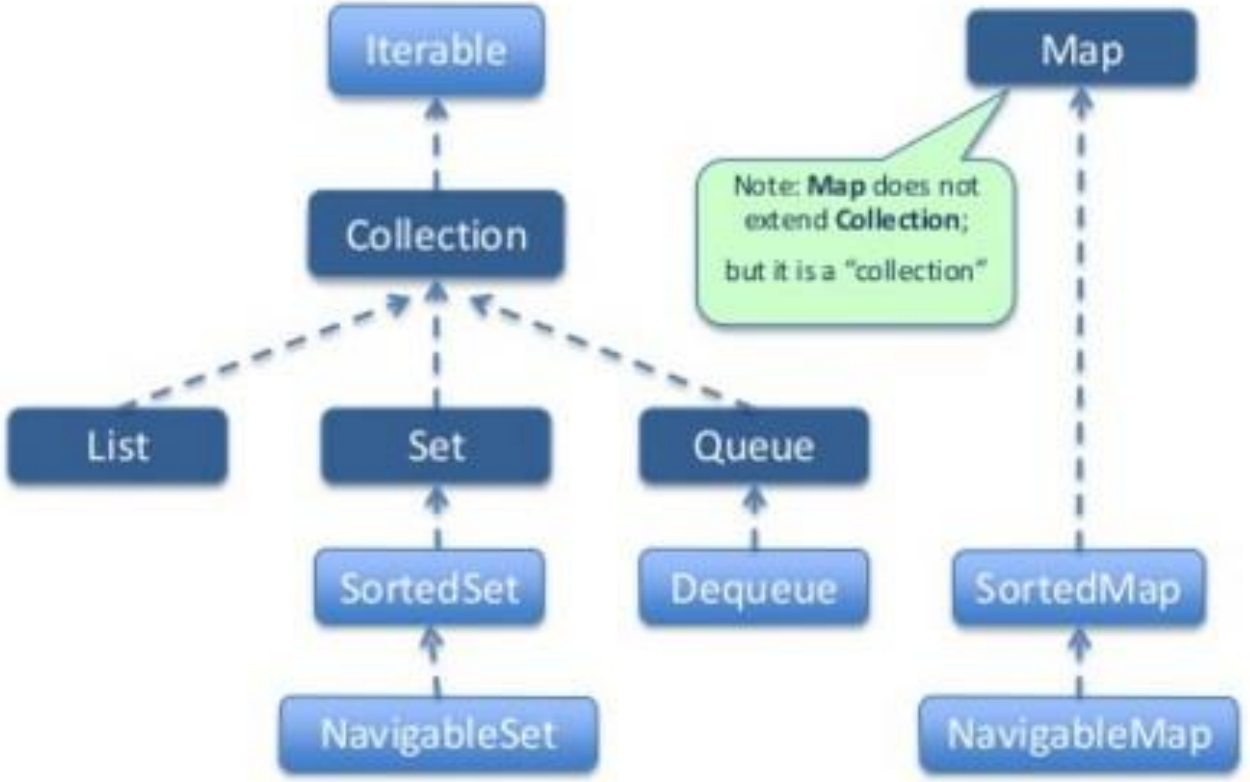


Concrete Map Classes

TreeMap<K, V>, HashMap<K, V>, LinkedHashMap<K, V>

Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	Sorted according to natural **order of keys**	Sorted according to the **insertion order**.	Random
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use Collection.synchronizedMap(new HashMap())			Yes but prefer to use ConcurrentHashMap
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

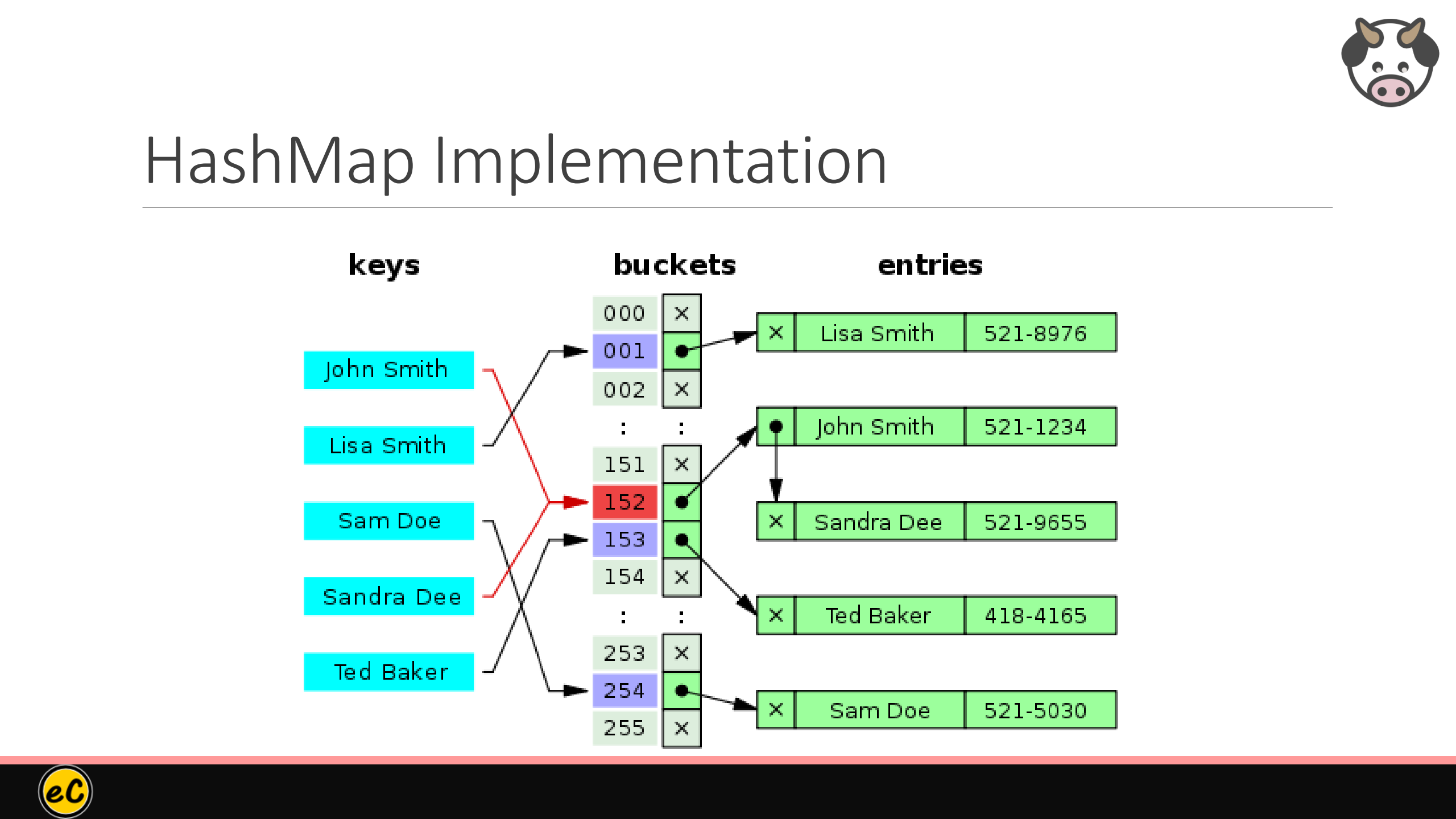
Sub-Interfaces of Map



HashMap, TreeMap and LinkedHashMap

Different Implementations (buckets, R-B Tree, LinkedList)

- The **HashMap** and **TreeMap** classes are two **concrete** implementations of the Map interface. The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping. The **TreeMap** class, implementing **SortedMap**, is efficient for traversing the keys in a sorted order.
- **LinkedHashMap** extends HashMap with a linked list implementation that supports an ordering of the entries in the map. The entries in a **HashMap** are not ordered, but the entries in a LinkedHashMap can be retrieved in the order in which they were inserted into the map (known as the insertion order), or the order in which they were last accessed, from least recently accessed to most recently (access order). The no-arg constructor constructs a LinkedHashMap with the insertion order. To construct a LinkedHashMap with the access order, use the **LinkedHashMap(initialCapacity, loadFactor, true).**



HashMap Implementation

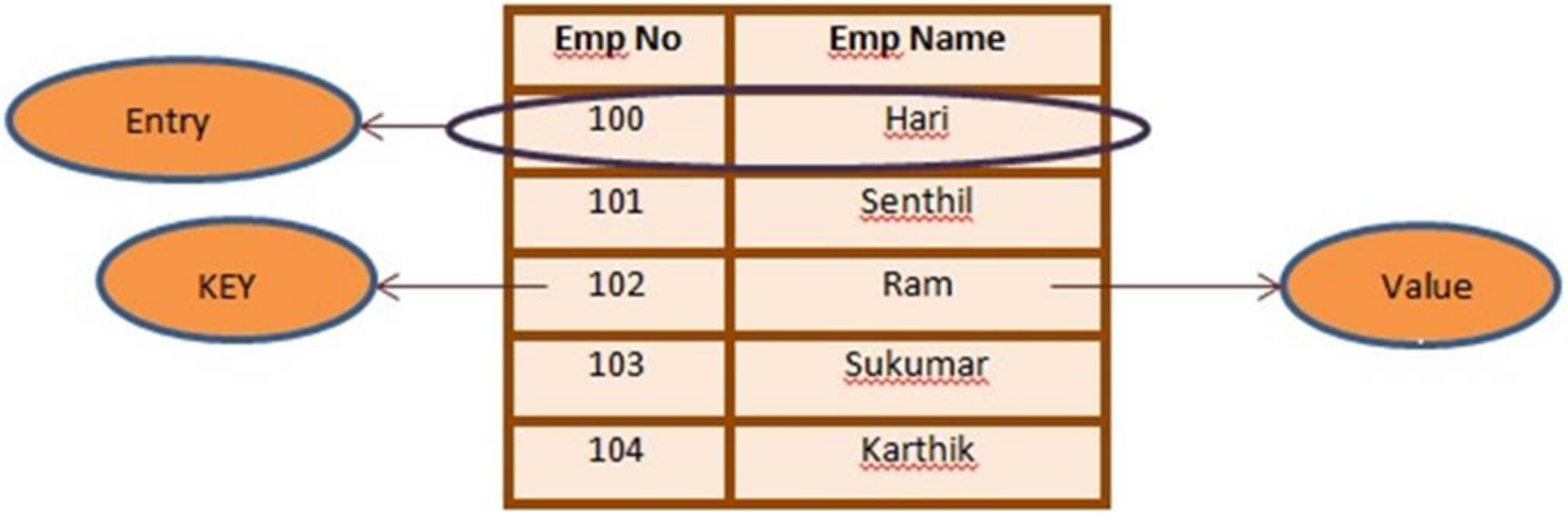
Using HashMap and TreeMap

TreeMap can be used for Sorting

- This example creates a hash map that maps borrowers to mortgages. The program first creates a hash map with the borrower's name as its key and mortgage as its value. The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

Demo Program:

TestMap.java



Demonstration Program

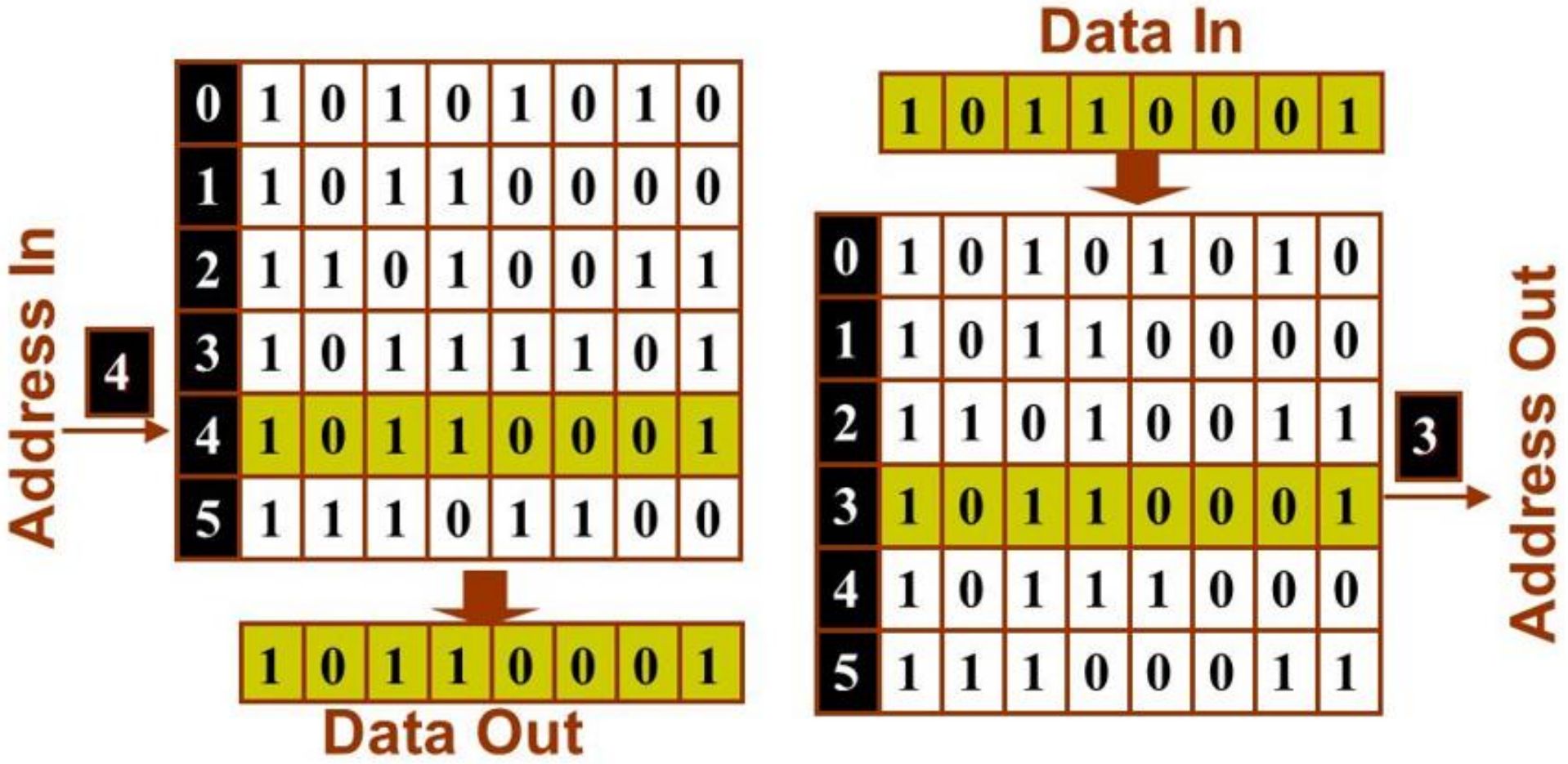
TESTMAP.JAVA

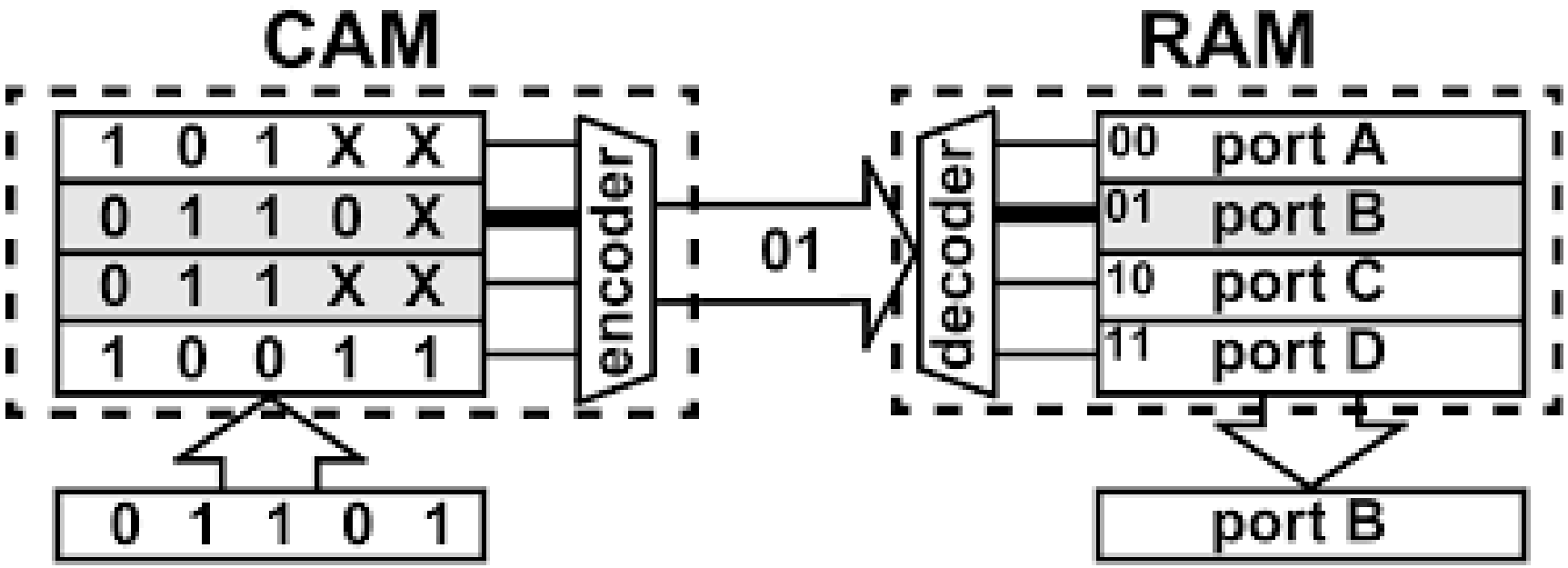
Content-Addressable Memory

SECTION 2

CAM: Introduction

- CAM vs. RAM




CAM
RAM
1 0 1 X X
0 1 1 0 X
0 1 1 X X
1 0 0 1 1
encoder
01
decoder
00 port A
01 port B
10 port C
11 port D
0 1 1 0 1
port B

Content Addressable Memory

- Use content to find the data location by a hash function is ***O(1)*** search algorithm.
- Therefore, it is possible to use Hash Function as a search function to quickly located a data in a data structure.
- Then, we have HashSet, HashMap.
- Both are **Abstract Data Types** that can be implemented by ArrayList (Vecotr), LinkedList or HeapTree.

Hash Table

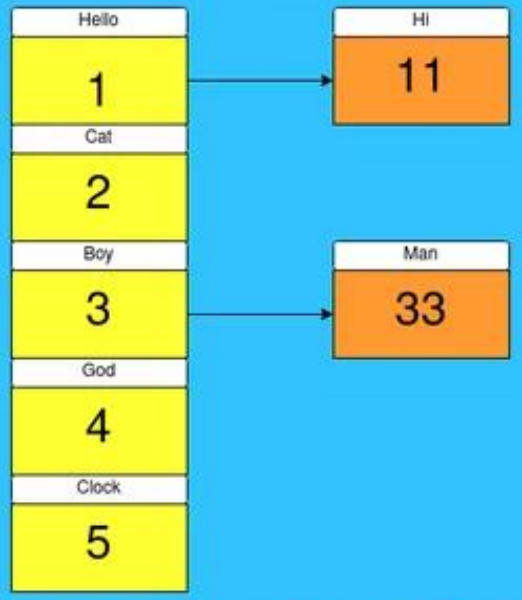
SECTION 3

Hash Table

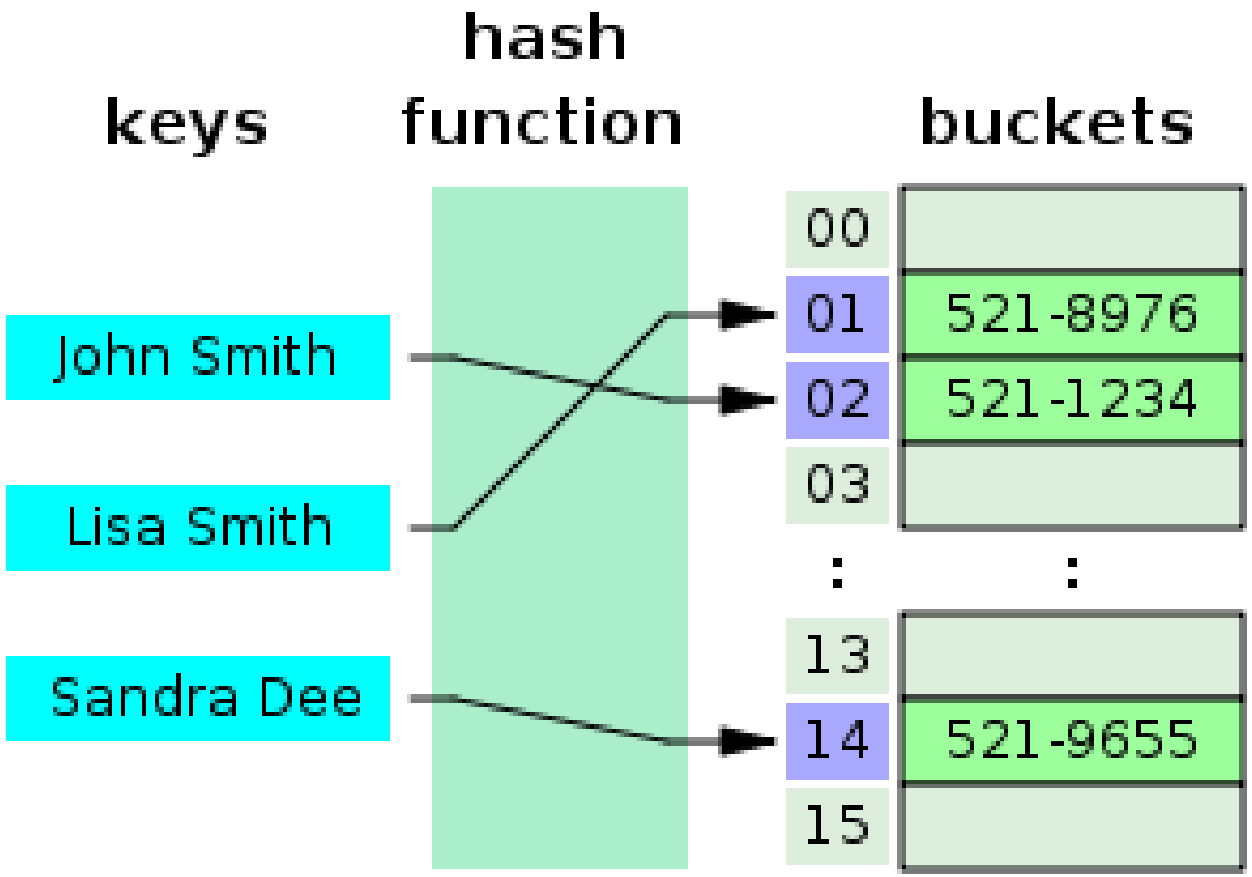
- A **Hash Table** in C/C++ (Associative array) is a data structure that maps keys to values. This uses a hash function to compute indexes for a key.
- Based on the Hash Table index, we can store the value at the appropriate location.
- If two different keys get the same index, we need to use other data structures (buckets) to account for these collisions.

Hash Table in C/C++

- Uses a Hash Function to find index
- Stores {Key : Value} pairs



- Collisions can be handled using different algorithms
- Separate Chaining is one common choice


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

Hash Table

Hash Table

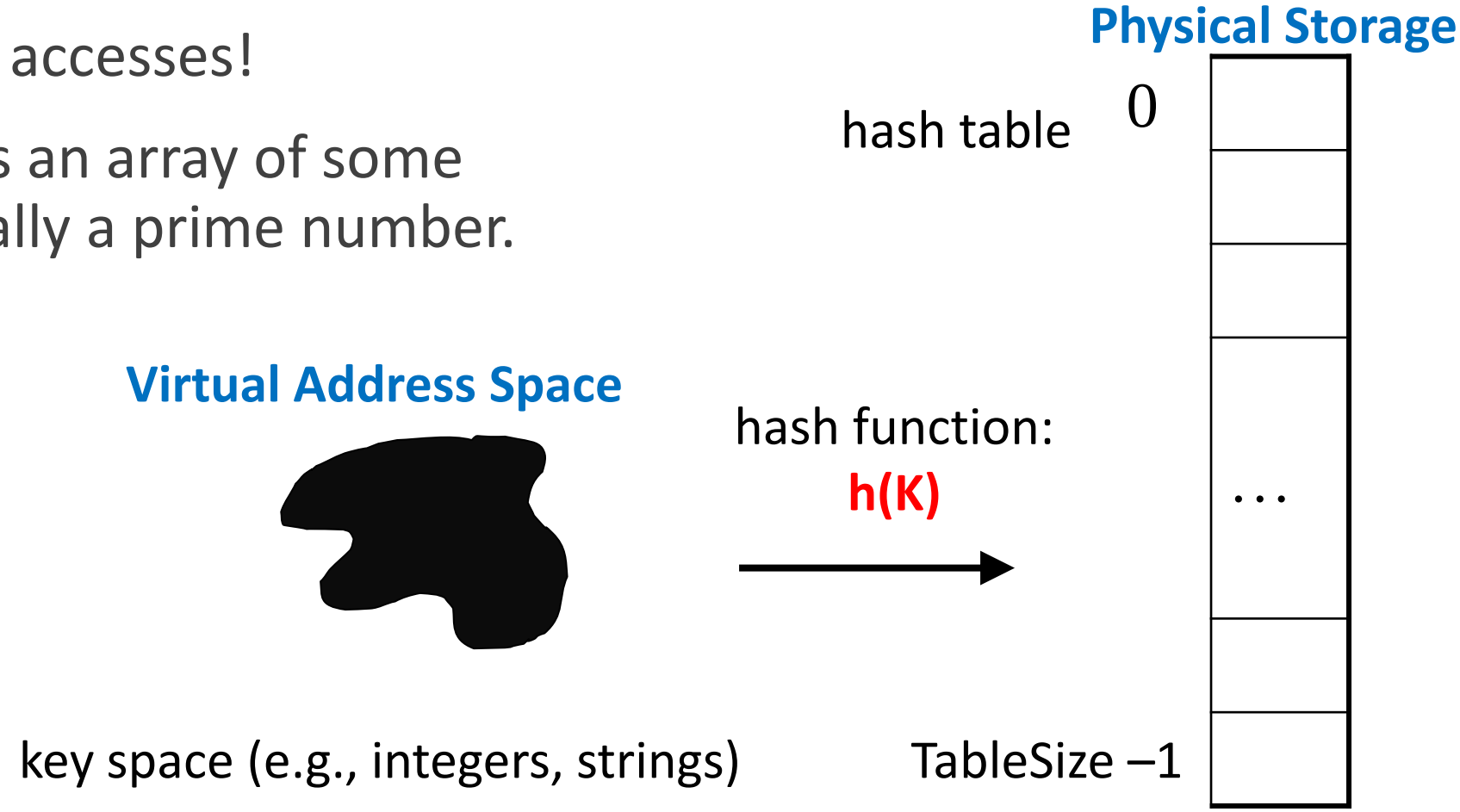
- The whole benefit of using a Hash Table is due to it's very **fast access time**. While there can be a collision, if we choose a very good hash function, this chance is almost zero.
- So, on average, the time complexity is a constant **O(1)** access time. This is called **Amortized Time Complexity**.
- The C++ STL (Standard Template Library) has the `std::unordered_map()` data structure which implements all these hash table functions.
- However, knowing how to construct a hash table from scratch is a crucial skill, and that is indeed what we aim to show you.

Hash Table

- Let us understand more about the implementation details.
  - Any Hash Table implementation has the following three components:
  - A good Hash function to map keys to values
  - A Hash Table Data Structure that supports insert, search and delete operations.
  - A Data Structure to account for collision of keys

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



Hash Tables

key space = integers

TableSize = 10

h(K) = K mod 10

Insert: 7, 18, 41, 94

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

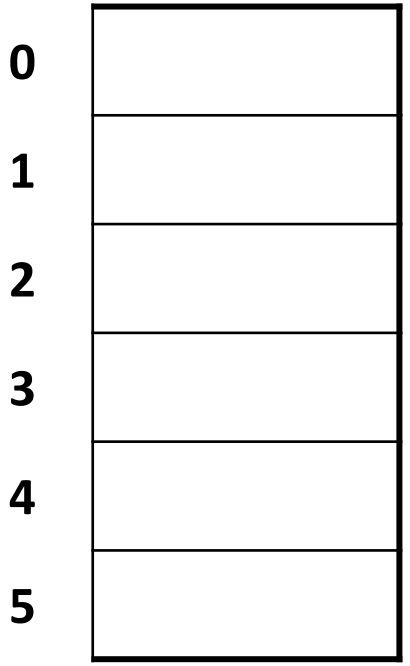
Another Example

key space = integers

TableSize = 6

h(K) = K mod 6

Insert: 7, 18, 41, 34



Hash Function

SECTION 4

Hash Functions

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Perfect Hash function:

Sample Hash Functions:

- key space = strings
- $s = s_0\ s_1\ s_2 \ldots s_{k-1}$

1. $h(s) = s_0 \bmod \text{TableSize}$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i \right) \bmod \text{TableSize}$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \bmod \text{TableSize}$

Collision Resolution

Collision: when two keys map to the same location in the hash table.

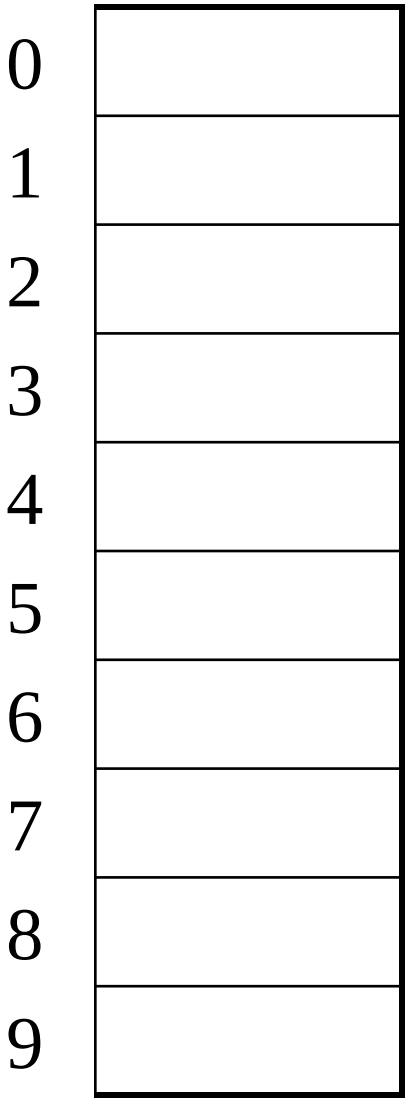
Two ways to resolve collisions:

1. Separate Chaining (Set)
2. Open Addressing (linear probing, quadratic probing, double hashing)

Hash Set with Buckets

SECTION 5

Separate Chaining



Insert:
10
22
107
12
42

- **<u>Separate chaining</u>**:
  All keys that map to the same hash value are kept in a list (or “bucket”).

Analysis of find

- Define: The load factor, λ, of a hash table is the ratio:

$$\frac{\mathrm{N}}{\mathrm{M}}$$

$\mathrm{N} \leftarrow$ no. of elements

$\mathrm{M} \leftarrow$ table size

For separate chaining, λ = average # of elements in a bucket

- Unsuccessful find:

- Successful find:

How big should the hash table be?

- For Separate Chaining:

tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10

    data hashes to 0, 3, 0, 5, 1, 0, 0
  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, 9, 7

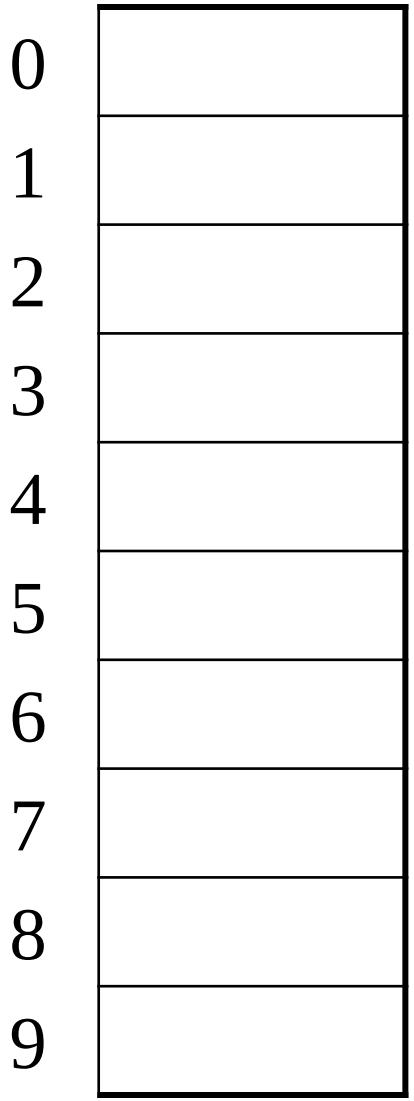
Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

Probing Methods (Open Addressing)

SECTION 6

Open Addressing



Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Terminology

	“**Open** Hashing”	“Closed Hashing”
	equals	equals
Weiss	“Separate Chaining”	“**Open** Addressing”

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

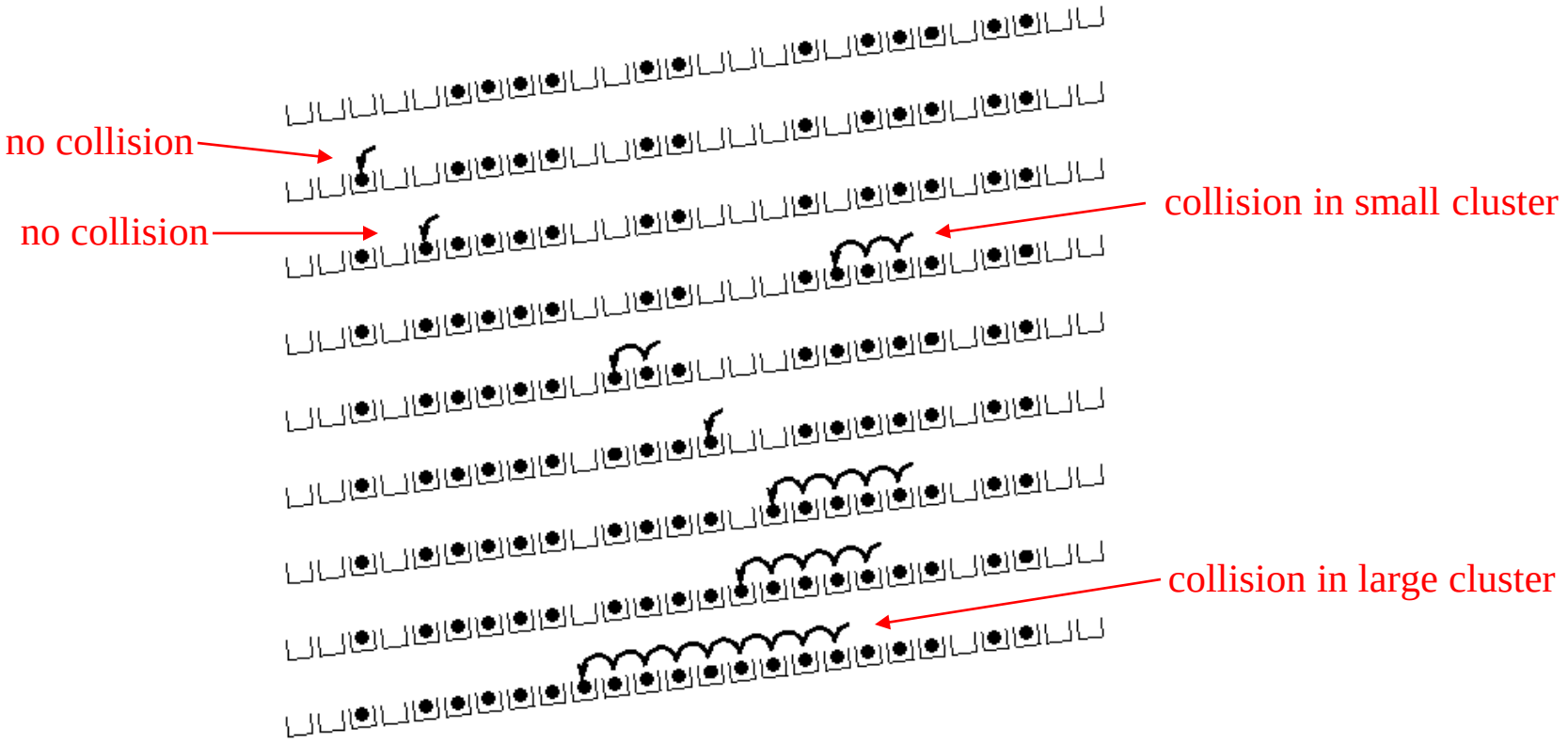
  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 2) mod TableSize

  . . .

  i^{th} probe = (h(k) + i) mod TableSize

Linear Probing – Clustering



Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$
  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$

Quadratic Probing

$$f(i) = i^2$$

Less likely to encounter Primary Clustering

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

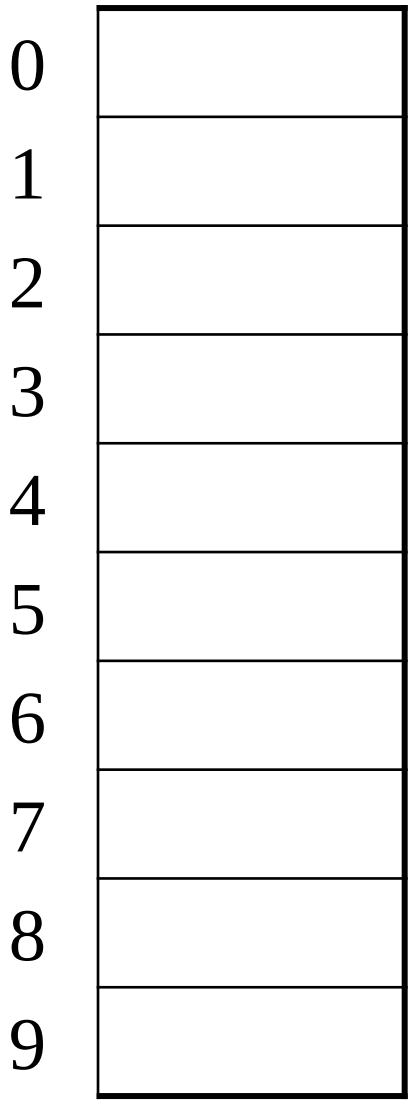
  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = (h(k) + i^2) mod TableSize

Quadratic Probing



Insert:
89
18
49
58
79

Quadratic Probing Example

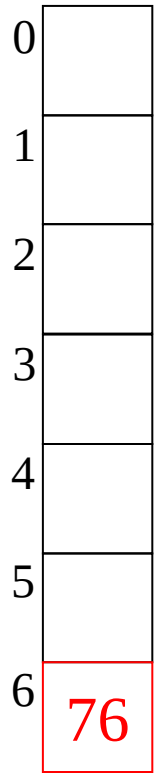
insert(76)
76%7 = 6

insert(40)
40%7 = 5

insert(48)
48%7 = 6

insert(5)
5%7 = 5

insert(55)
55%7 = 6



But… insert(47)
47%7 = 5

Quadratic Probing: Success guarantee for $\lambda < ½$

- If size is prime and $\lambda < ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all $0 \le i,j \le size/2$ and $i \ne j$

    $(h(x) + i^2) \bmod size \ne (h(x) + j^2) \bmod size$
  - by contradiction: suppose that for some $i \ne j$:

    $(h(x) + i^2) \bmod size = (h(x) + j^2) \bmod size$

    $\Rightarrow \; i^2 \bmod size = j^2 \bmod size$

    $\Rightarrow (i^2 - j^2) \bmod size = 0$

    $\Rightarrow [(i + j)(i - j)] \bmod size = 0$

Because size is prime **(i-j)** or **(i+j)** must be zero, and neither can be

Quadratic Probing: Properties

- For *any* $\lambda < ½$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad

•But what about keys that hash to the same *spot*?

- ***Secondary Clustering!***

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

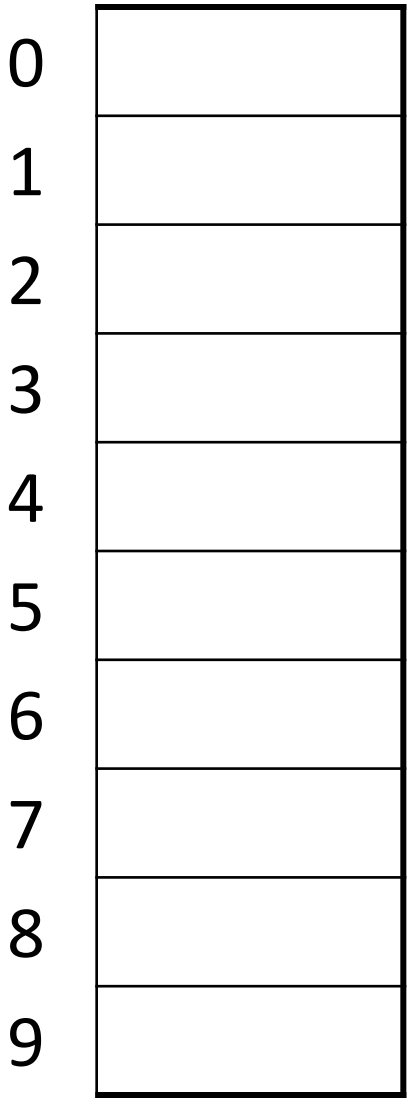
  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Resolving Collisions with Double Hashing



Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

M =

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Java HashCode

SECTION 7

Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically “significant” fields of objects

Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically “significant” fields of objects

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.

Simulated Hashing Function

SECTION 8

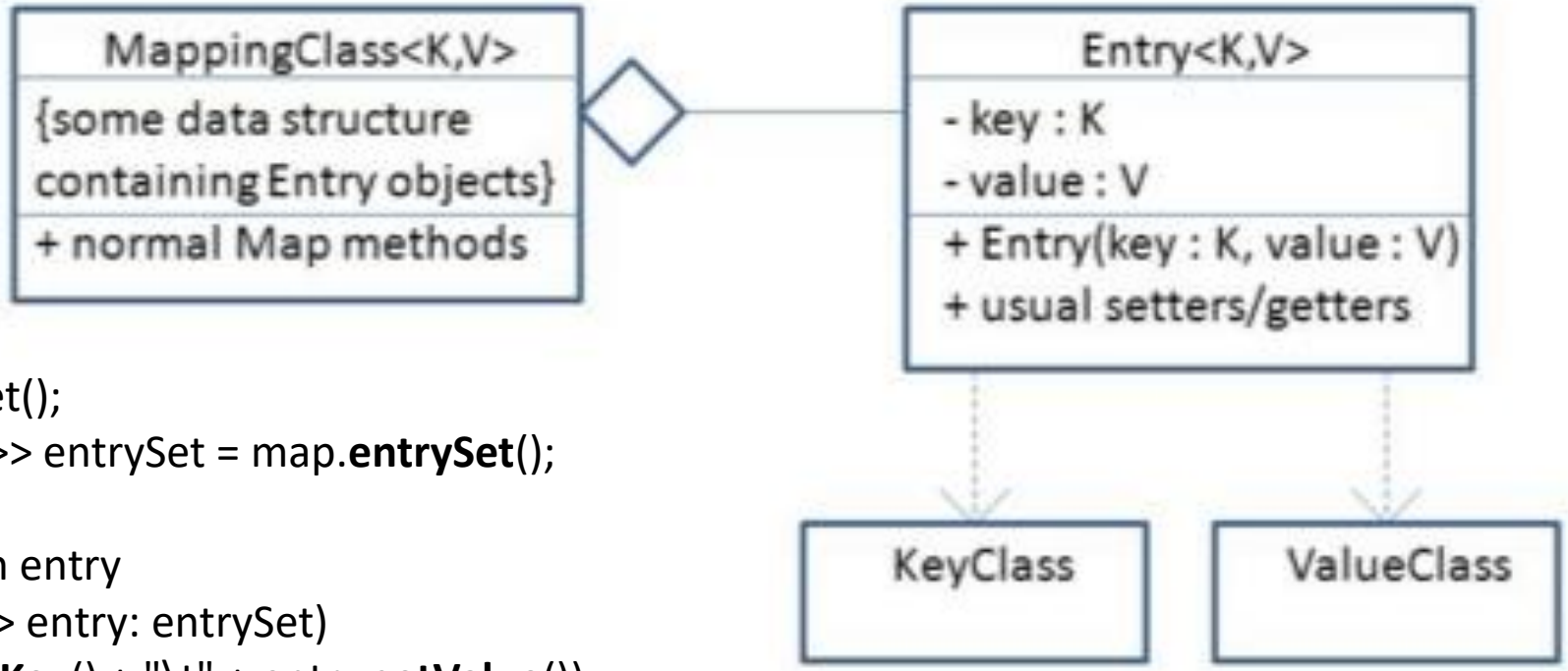
Demonstration Program

SIMULATEDHASHFUNCTION.JAVA

Word Occurrence

SECTION 9

Map Entry Class




```
Set<String> keyset = map.keyset();
Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

 // Get key and value from each entry
 for (Map.Entry<String, Integer> entry: entrySet)
    System.out.println(entry.getKey() + "\t" + entry.getValue());
```

Counting the Occurrences of Words in a Text

- This program counts the occurrences of words in a text and displays the words and their occurrences in ascending order of the words. The program uses a hash map to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase the value for the word (key) by 1 in the map. To sort the map, convert it to a tree map.

Demonstration Program

COUNTOCCURRENCEOFWORDS.JAVA

Random Sequence Generation

SECTION 10

Three way to Generate a Random Sequence

1. Regular assignment and, then, random shuffling.
2. Busy probing
3. Data Chain random removal and addition.

Demonstration Program

RANDOMSEATING.JAVA

Content-Addressable Memory (Decoder Design)

SECTION 11

Content-Addressable Memory

- Usually, we use index to find the element when we use array.
- There are situations that we want to find index from the element, this we call it content-addressable memory or reverse-access of an array.
- Typical example is that we have an array which has index and its related code. And, we put the reverse relationship to a map<String, Integer>. Then, we will be able to find the index based on the content (String).

Demonstration Program

DECODER.JAVA

Random Sequence Generation

1. A Sequence that contains all characters.

2. The sequence can never repeat and must be random.

3. The sequence is considered to be one valid permutation of the symbols.

```
static String source = "piquerasmwpqutpqqwjasdfhagnvznxcn".toUpperCase();
public static void randomEncoder(String[] c){
    ArrayList<String> chain = new ArrayList<String>();
    for (int data=0; data<26; data++) chain.add((char)('A'+data)+"");
    int p=0;
    while (chain.size()>0){ // O(n)
        int i = (int)(Math.random()*chain.size());
        c[p++] = chain.remove(i);
     }
 }
```

```
public static void main(String[] args){
    System.out.print("\f");
    String[] encoder = new String[26];
    randomEncoder(encoder);
    System.out.println("Source  Data: "+source);
    String encoded = "";
    for (int i=0; i<source.length(); i++){
          encoded += encoder[source.charAt(i)-'A'];
      }
    System.out.println("Encoded Data: "+encoded);
    Map<String, Integer> m = new HashMap<String, Integer>();
    for (int i=0; i<encoder.length; i++){
          m.put(encoder[i],i);
      }
    String decoded = "";
    for (int i=0; i<encoded.length(); i++){
         decoded += (char)('A'+m.get(encoded.substring(i, i+1)))+"";
      }
    System.out.println("Decoded Data: "+decoded);
 }
```

Source Data: PIQUERASMWPQUTPQQWJASDFHAGNVZNXCN
Encoded Data: OQWSLMRGYCOWSNOWWCKRGIBHRTPEFPAXP
Decoded Data: PIQUERASMWPQUTPQQWJASDFHAGNVZNXCN

Sets

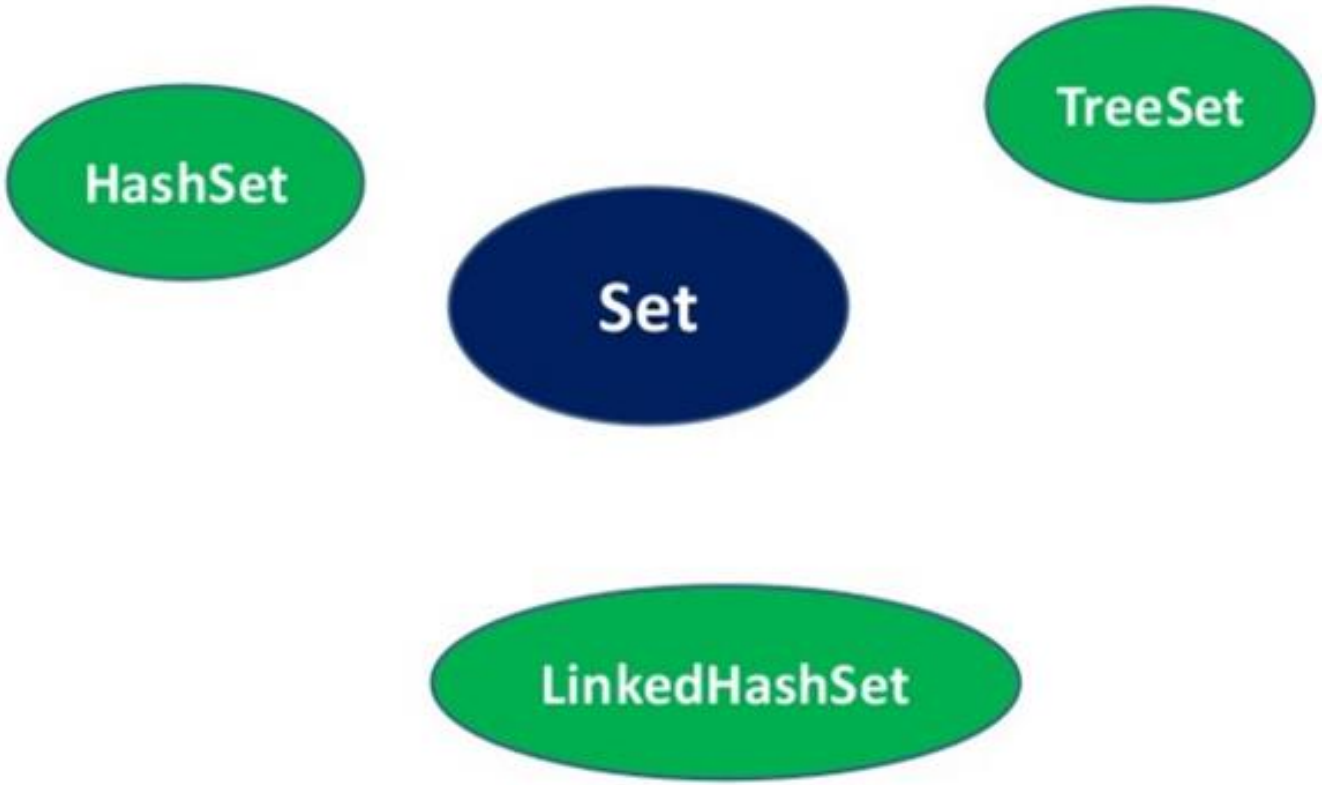
HashSet and LinkedHashSet

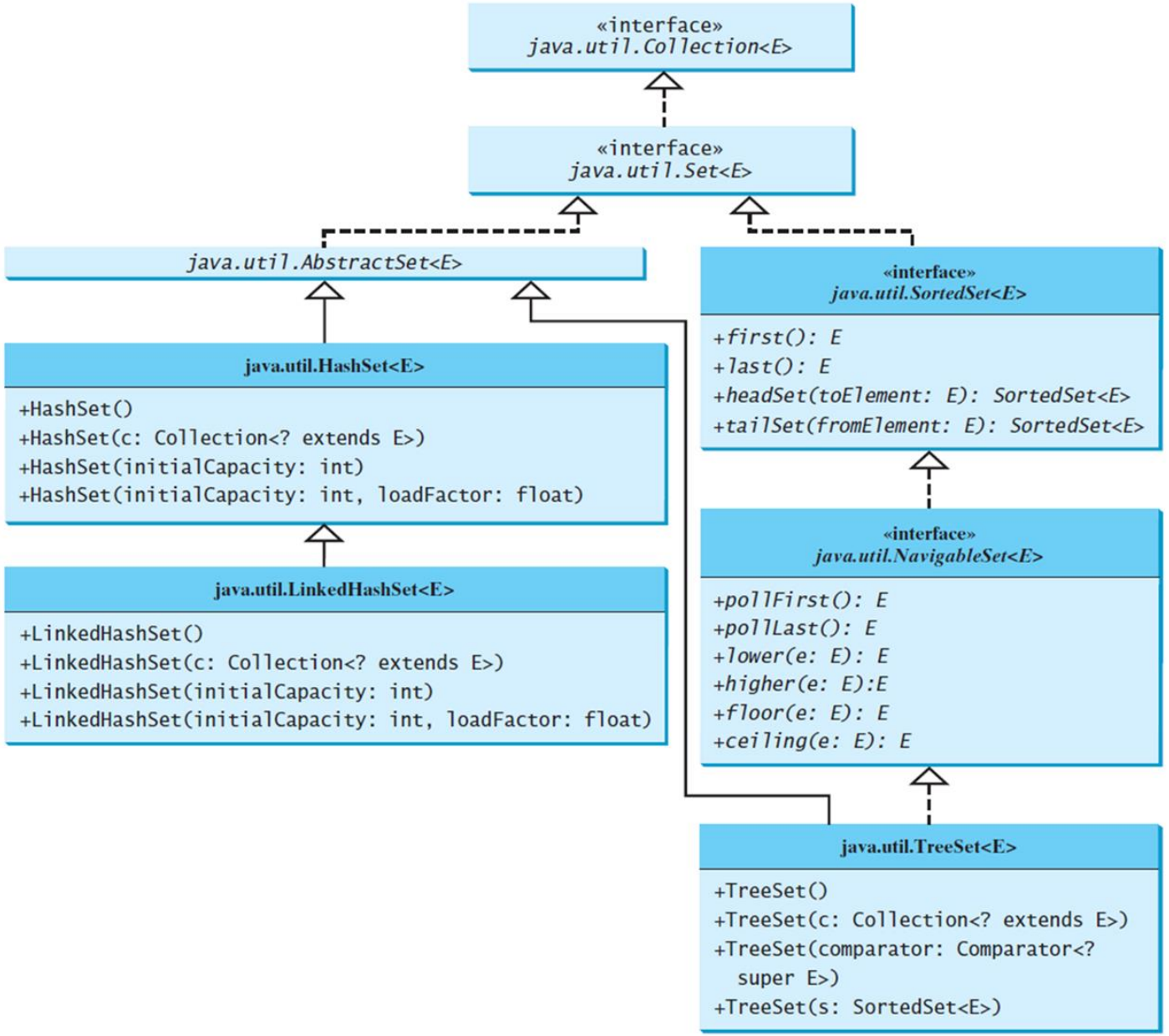
SECTION 12

The Set Interface

The Set interface extends the Collection interface. It does not introduce new methods or constants, but it stipulates that an instance of **Set contains no duplicate elements**. The concrete classes that implement Set must ensure that no duplicate elements can be added to the set. That is **no two elements e1 and e2 can be in the set such that e1.equals(e2) is true.**

Set Interface and Concrete Classes




«interface»
java.util.Collection<E>
«interface»
java.util.Set<E>
java.util.AbstractSet<E>
«interface»
java.util.SortedSet<E>
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
java.util.HashSet<E>
+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)
«interface»
java.util.NavigableSet<E>
+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E
java.util.LinkedHashSet<E>
+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)
java.util.TreeSet<E>
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<?
super E>)
+TreeSet(s: SortedSet<E>)

HashSet vs LinkedHashSet vs EnumSet vs TreeSet in Java

	Data Structure	Sorting	Iterator	Nulls?
HashSet	Hash table	No	Fail-fast	Yes
Linked HashSet	Hash table + linked list	Insertion Order	Fail-fast	Yes
EnumSet	Bit vector	Natural Order	Weakly consistent	No
TreeSet	Red-black tree	Sorted	Fail-fast	Depends
CopyOnWrite ArraySet	Array	No	Snapshot	Yes
Concurrent SkipListSet	Skip list	Sorted	Weakly consistent	No

Data Structure

ArrayList: Dynamic Array (Work like Array)

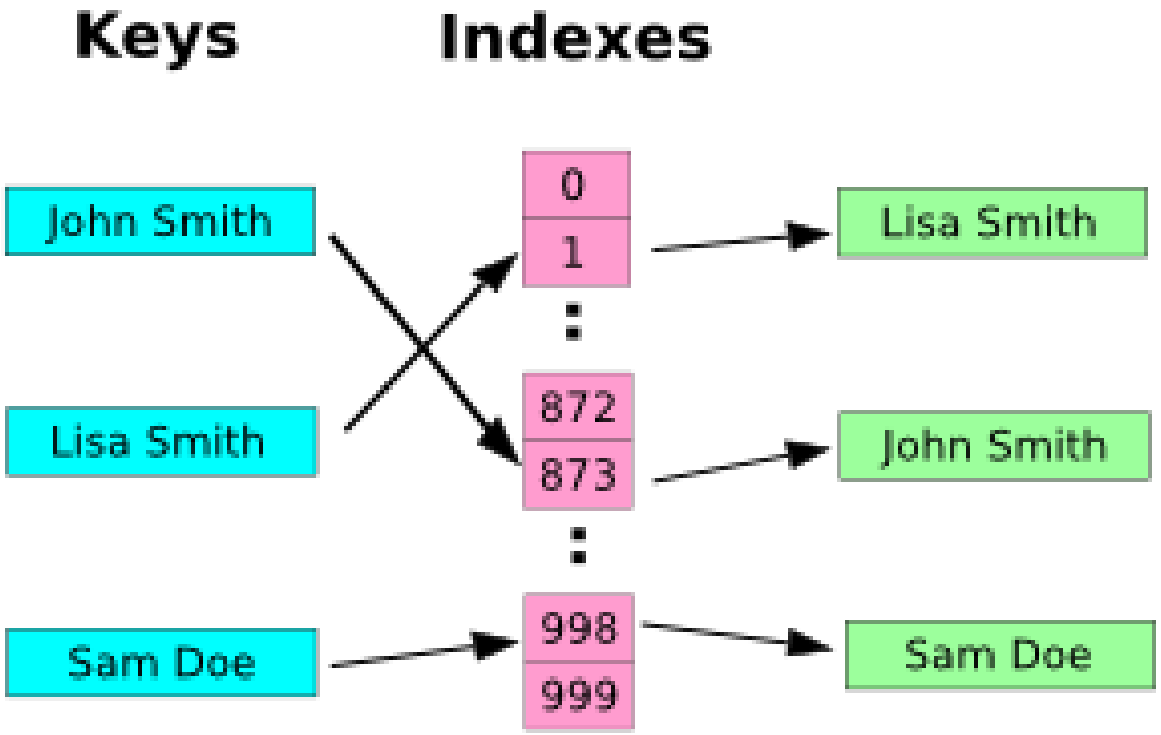
LinkedList: List linked by Nodes (Random Insertion/Deletion)

Set: Non-Recurring data set. (TreeSet: ordered) (Used for the Non-Recurring Word)

Map: (Key, Value) pair like dictionary (TreeMap: ordered) (Used for the Occurrence Count)

HashSet

One to One Mapping (key-index-memory)



The HashSet class is a concrete class that implements Set. It can be used to store duplicate-free elements. For efficiency, objects added to a hash set need to implement the hashCode method in a manner that properly disperses the hash code.

Demo Program:

TestHashSet.java Using HashSet and Iterator

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

Demonstration Program

TESTHASHSET.JAVA USING HASHSET AND ITERATOR

Demo Program:

TestLinkedHashSet.java Using LinkedHashSet

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

Demonstration Program

TESTLINKEDHASHSET.JAVA USING LINKEDHASHSET

Set (TreeSet)

SECTION 13

The SortedSet Interface and the TreeSet Class

- SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.
- One way is to use the Comparable interface.
- The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo method in the class that implements the Comparable interface. This approach is referred to as order by comparator.

Using TreeSet to Sort Elements in a Set

This example creates a hash set filled with strings, and then creates a tree set for the same strings. The strings are sorted in the tree set using the **compareTo** method in the Comparable interface. The example also creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

Demo Program:

TestTreeSet.java

Demonstration Program

TESTTREESET.JAVA

The Using Comparator to Sort Elements in a Set

Write a program that demonstrates how to sort elements in a tree set using the Comparator interface. The example creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

Demo Program:

TestTreeSetWithComparator.java

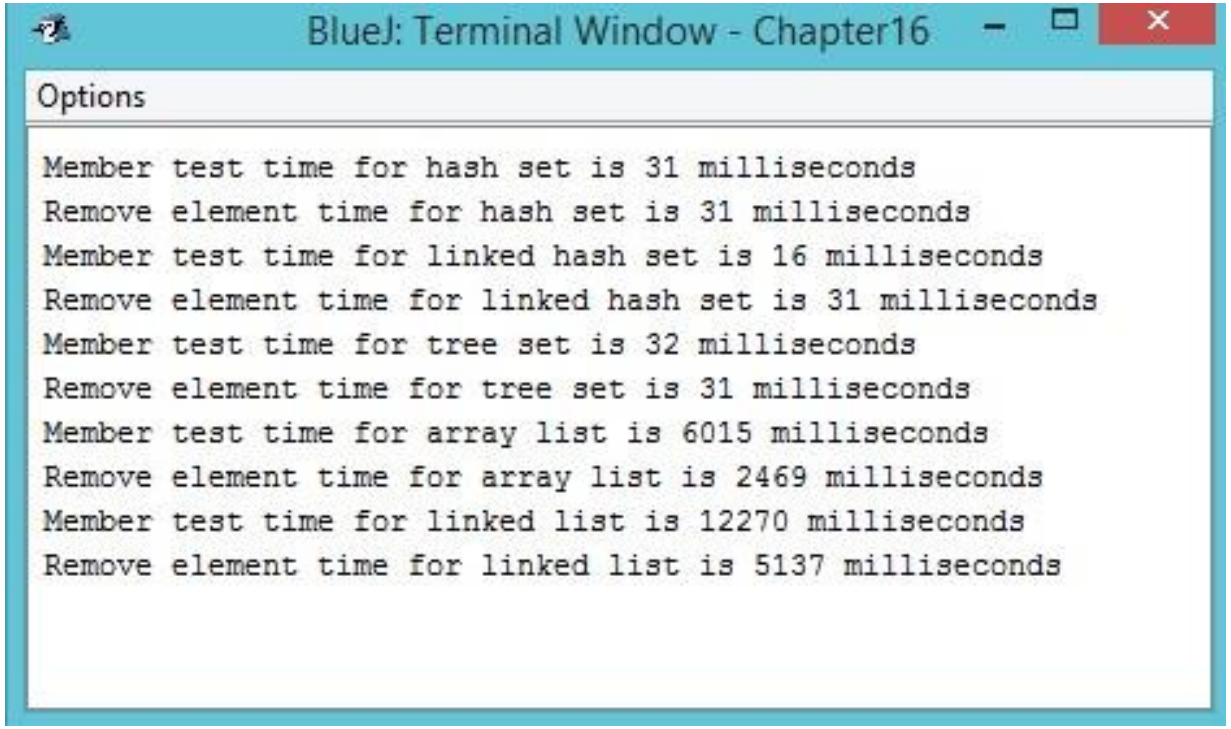
Performance of Sets and Lists

Demo Program:

SetListPerformanceTest.java

Demo Program:

CountKeywords.java



Demonstration Program

SETLISTPERFORMANCETEST.JAVA

COUNTKEYWORDS.JAVA