Hashing (Chapter 27) + Sets and Maps (Chapter 21)

Map

- A *map* is a data structure that stores entries. Each entry contains two parts: *key* and *value*.
- The key is also called a *search key*, which is used to search for the corresponding value. For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.
- A map is also called a *dictionary*, a *hash table*, or an associative array.
- The new trend is to use the term map.

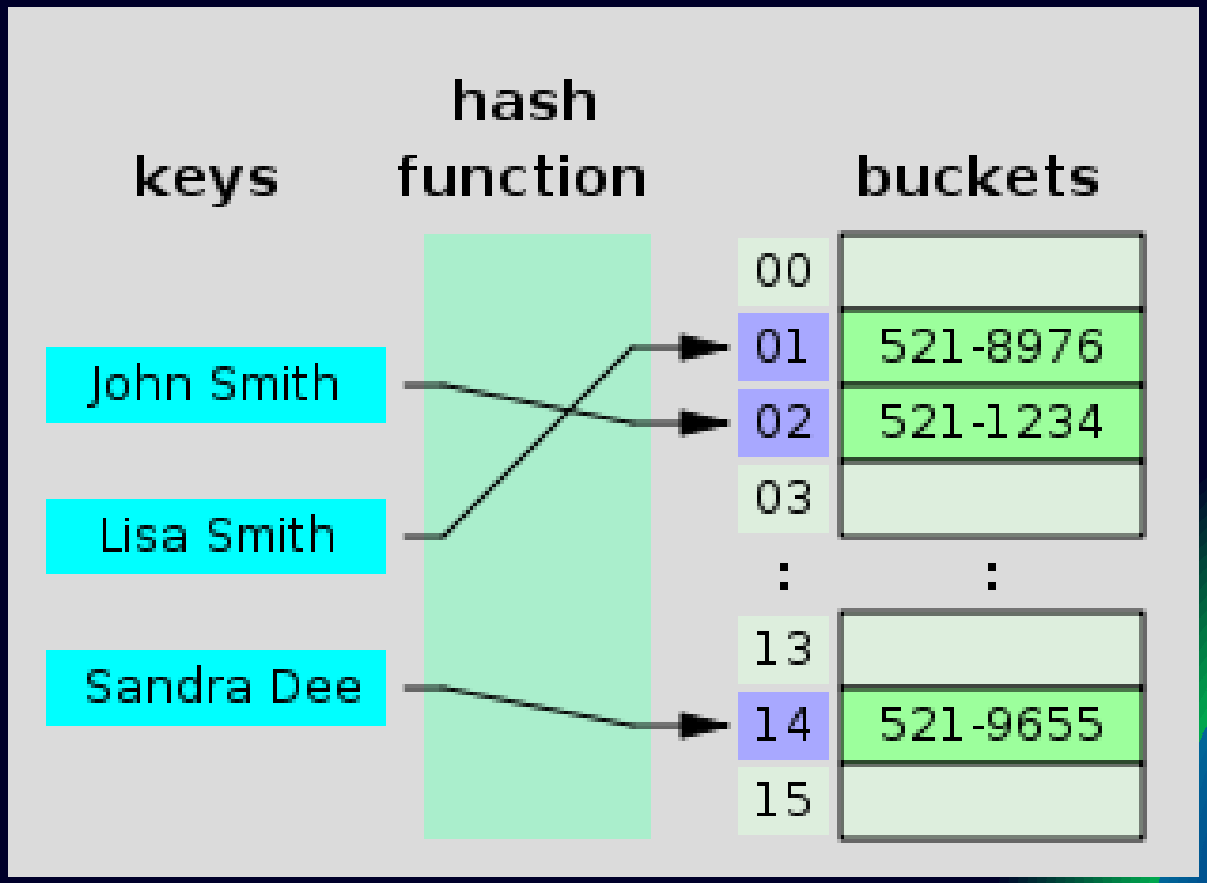
What is Hashing?

- If you know the index of an element in the array, you can retrieve the element using the index very efficiently.
- So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.
- The array that stores the values is called a *hash table*.
- The function that maps a key to an index in the hash table is called a *hash function*.

- *Hashing* is a technique that retrieves the value using the index obtained from key without performing a search.

Hash Function and Hash Codes

A typical hash function first converts a search key to an integer value called a *hash code*, and then compresses the hash code into an index to the hash table.

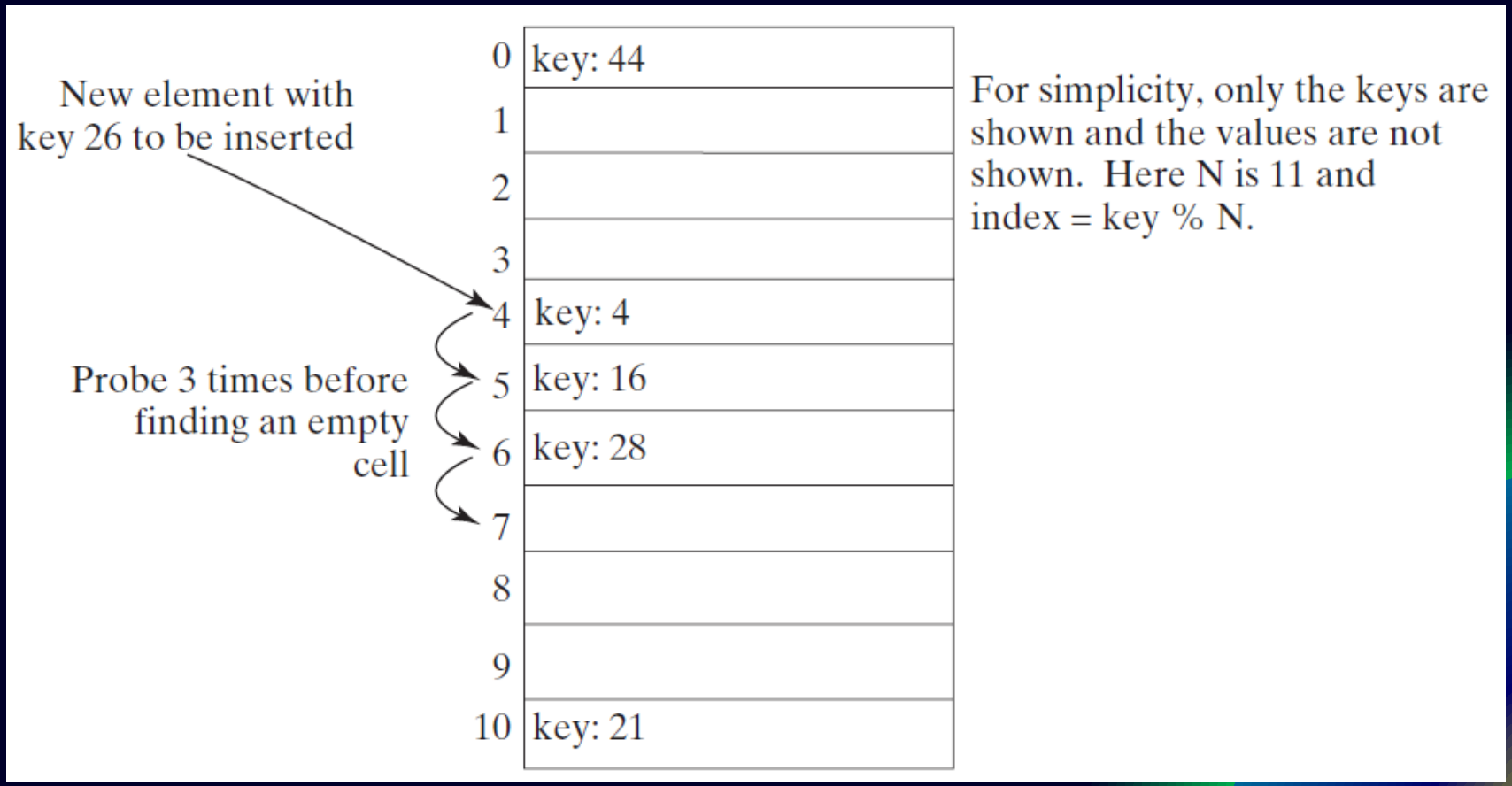


Collisions

- A collision occurs when two keys are mapped to the same index in a hash table.
- Generally, there are two ways for handling collisions: open addressing and separate chaining.
- Open addressing is the process of finding an open location in the hash table in the event of a collision.
- Open addressing has several variations: linear probing, quadratic probing, and double hashing.

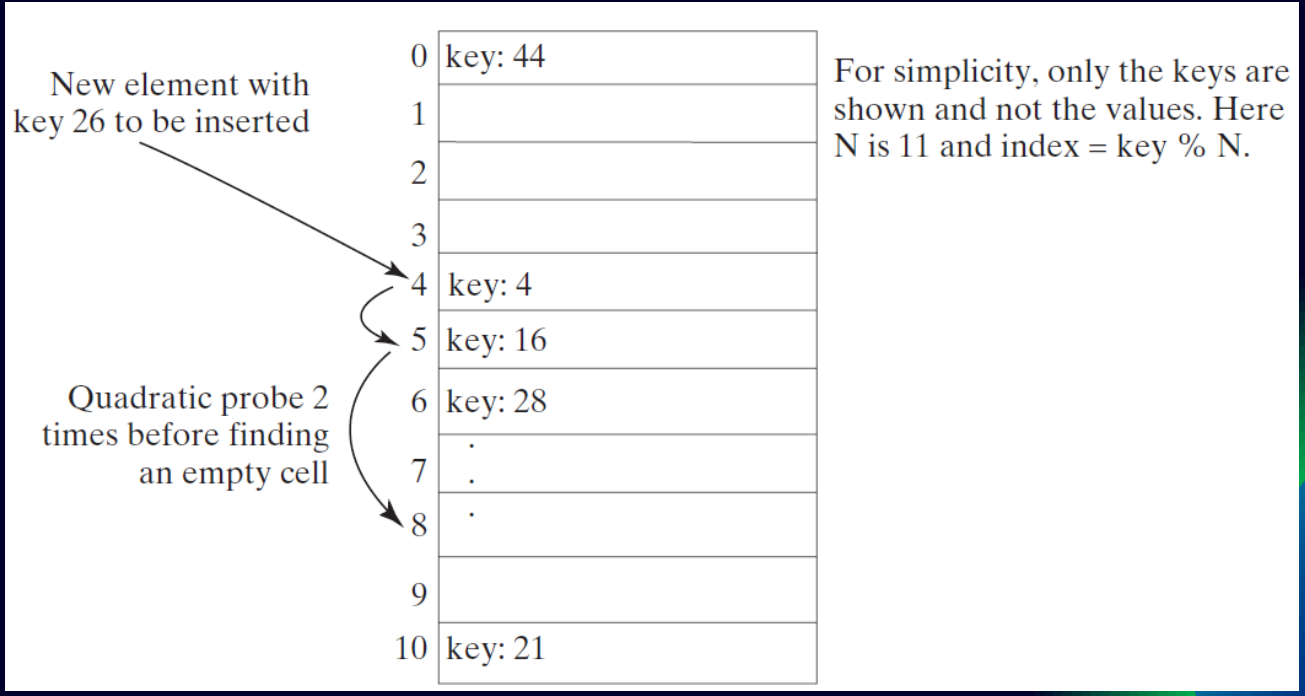
Linear Probing

When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially.



Quadratic Probing

Quadratic probing can avoid the clustering problem that can occur in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + 1) \% N$, $(k + 4) \% N$, $(k + 9) \% N$.

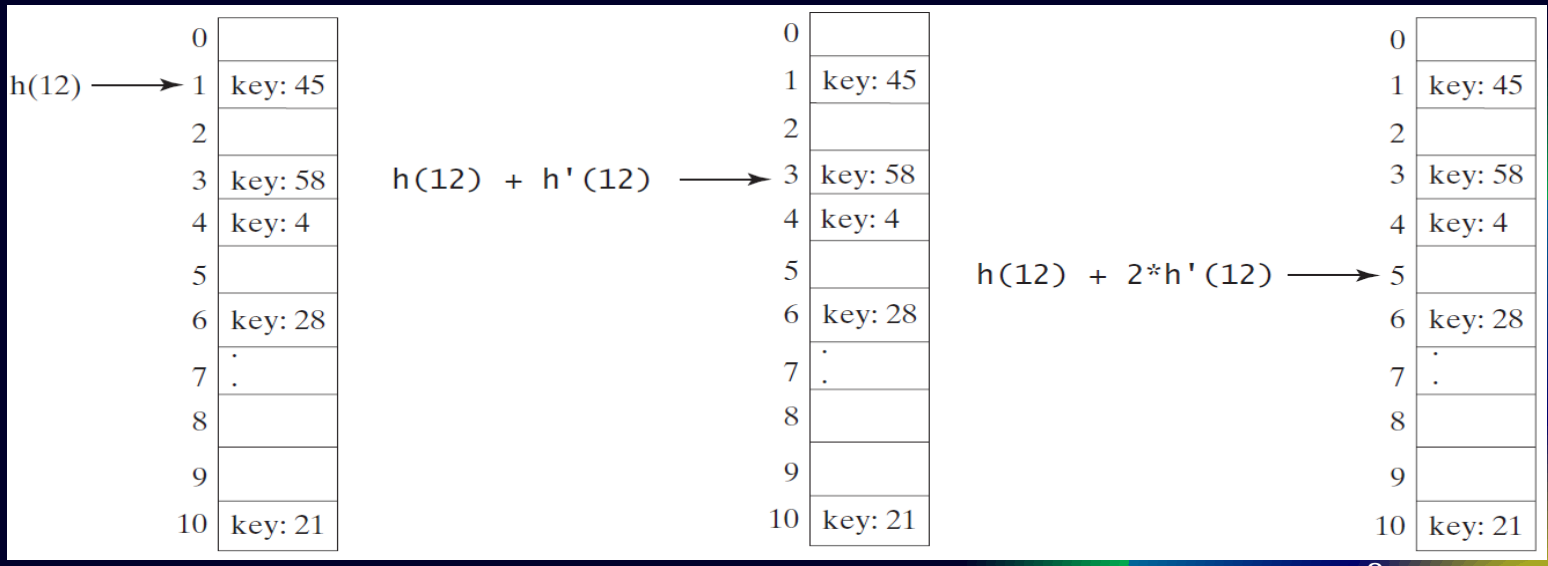


Double Hashing

- Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem. Specifically, double hashing looks at the cells at indices $(k + j * h'(key)) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + h'(key)) \% N$, $(k + 2 * h'(key)) \% N$, $(k + 3 * h'(key)) \% N$, and so on.

h(k) = k % 11

h'(k) = 7 – k % 7

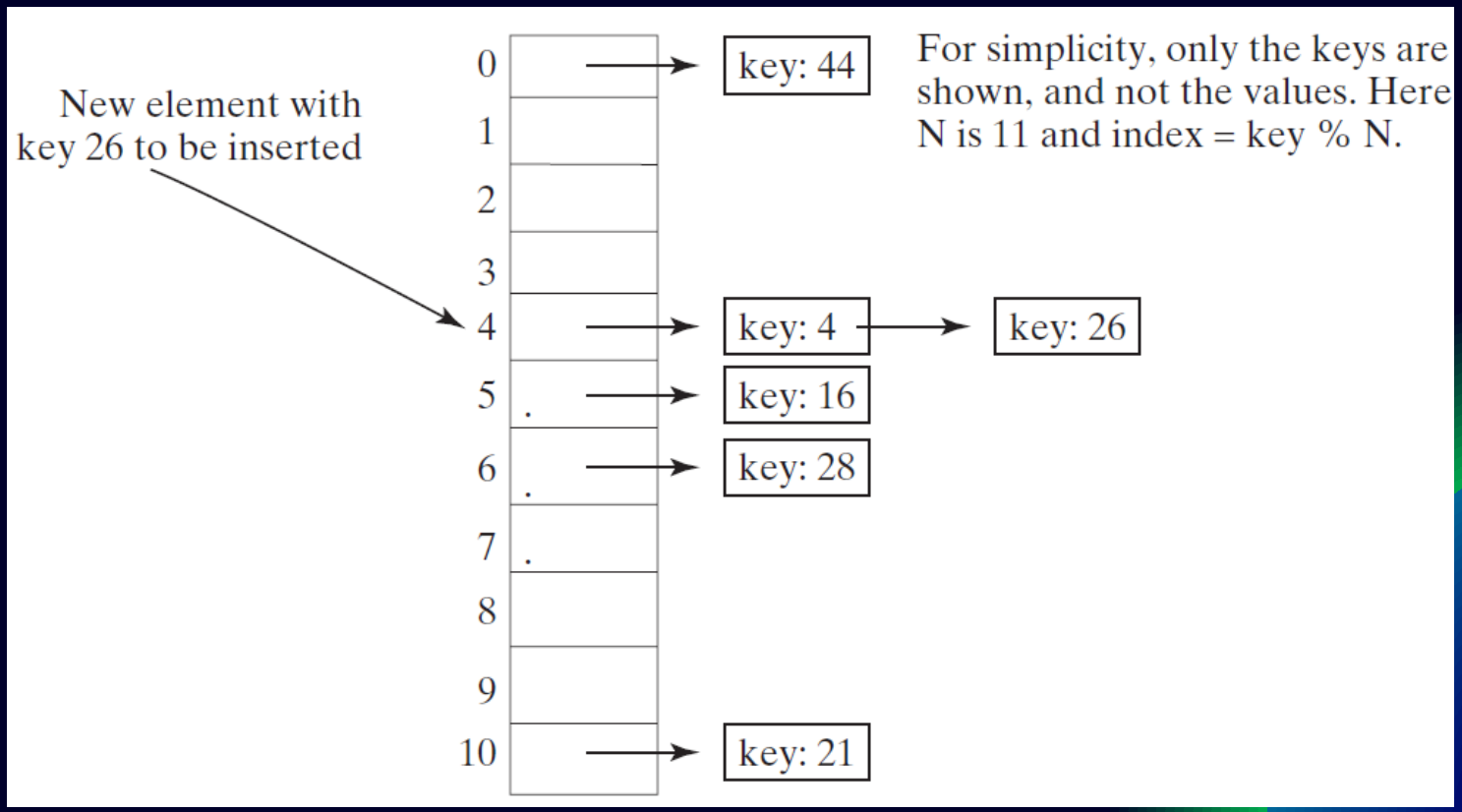


Handling Collisions Using Separate Chaining

The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations.

Each location in the separate chaining scheme is called a *bucket*.

A bucket is a container that holds multiple entries.



Sets and Maps

- A set is an efficient data structure for storing and processing non-duplicate elements.
- A map is like a dictionary that provides a quick lookup to retrieve a value using a key.

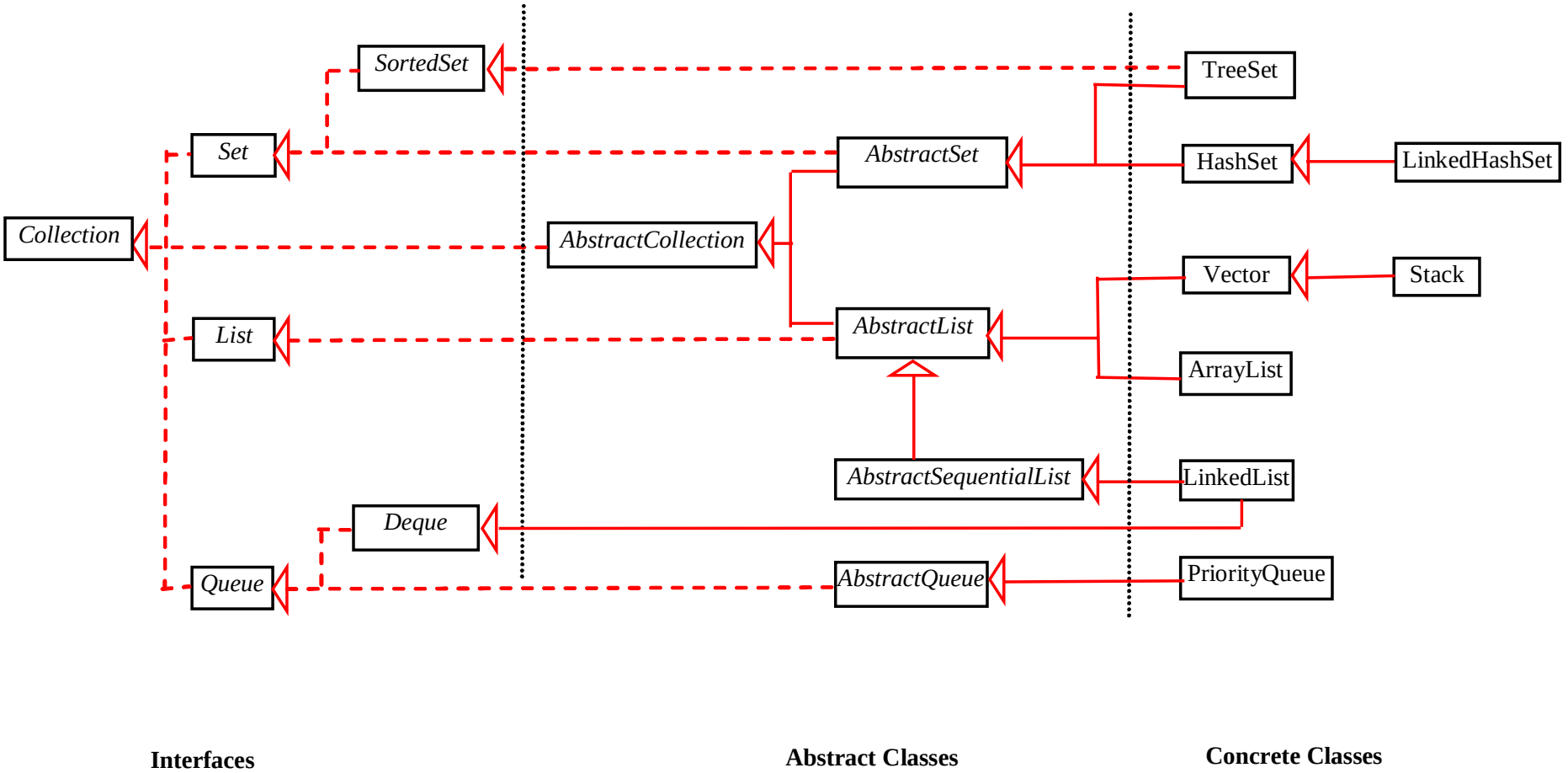
Examples

The "**No-Fly**" **list** is a list, created and maintained by the U.S. government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a *set*.

Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key. A *map* is an efficient data structure for such a task.

Review of Java Collection Framework hierarchy

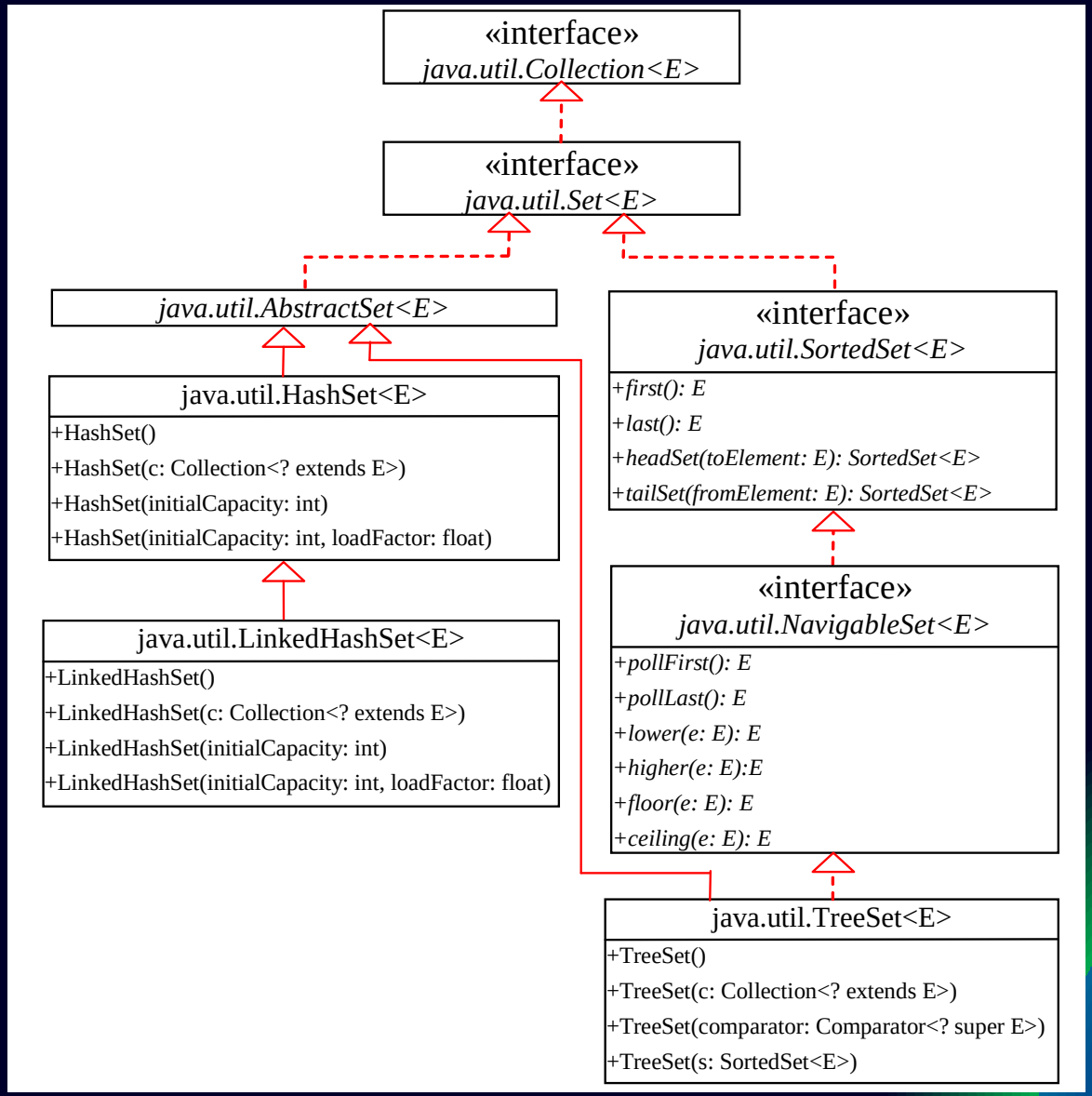
Set is a subinterface of Collection. You can create a set using one of its three concrete classes: HashSet, LinkedHashSet, or TreeSet.



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.

The Set Interface Hierarchy



The AbstractSet Class

- The AbstractSet class is a convenience class that extends AbstractCollection and implements Set.
- The AbstractSet class provides concrete implementations for the equals method and the hashCode method.
- The hash code of a set is the sum of the hash code of all the elements in the set.

The HashSet Class

- The HashSet class is a concrete class that implements the Set interface backed by a hash table.
- It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.
- 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.

The HashSet Class

- You can create an empty hash set using its no-arg constructor or create a hash set from an existing collection.
- By default, the initial capacity is 16 and the load factor is 0.75.
- If you know the size of your set, you can specify the initial capacity and load factor in the constructor.
- The load factor is a value between 0.0 and 1.0 and measures how full the set is allowed to be before its capacity is increased.

The HashSet Class Methods

- boolean add(E e) - Adds the specified element to this set if it is not already present.
- void clear() - Removes all of the elements from this set.
- Object clone() - Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
- boolean contains(Object o) - Returns true if this set contains the specified element.
- boolean isEmpty() - Returns true if this set contains no elements.
- Iterator<E> iterator() - Returns an iterator over the elements in this set.
- boolean remove(Object o) - Removes the specified element from this set if it is present.
- int size() - Returns the number of elements in this set (its cardinality

Example: Using HashSet

```
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }
  }
}
```

LinkedHashSet Class

- Hash table and linked list implementation of the Set interface, with predictable iteration order.
- This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries.
- This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order).
- Note that insertion order is not affected if an element is re-inserted into the set.

Example: Using LinkedHashSet

```
import java.util.*;

public class TestLinkedHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new LinkedHashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String element: set)
      System.out.print(element.toLowerCase() + " ");
  }
}
```

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.

The SortedSet Interface and the TreeSet Class, cont.

- 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*.

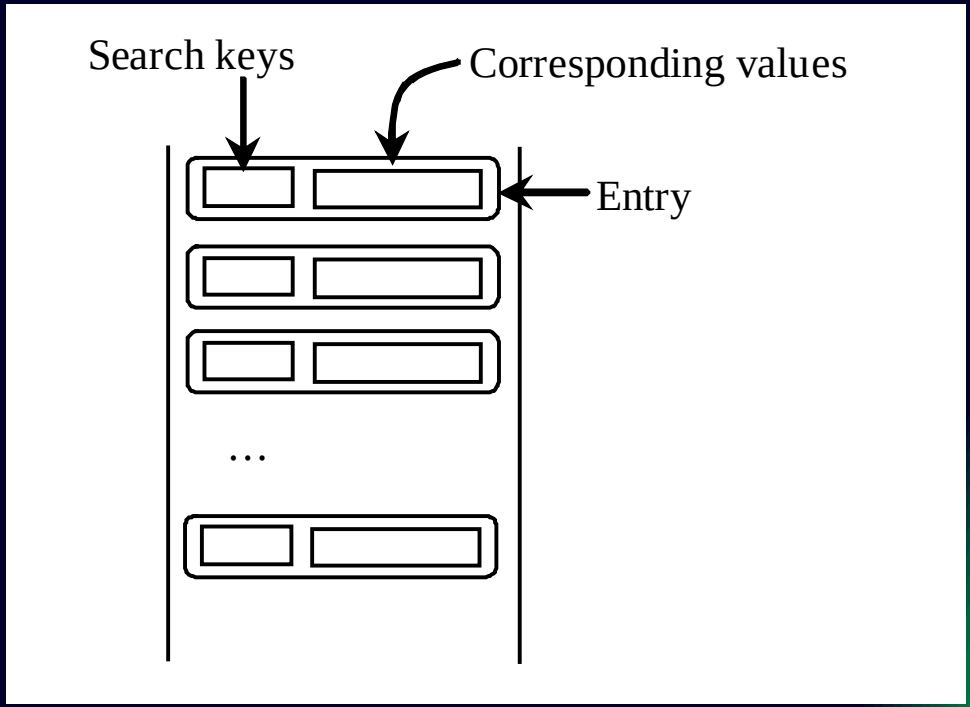
https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

Example: Using TreeSet to Sort Elements in a Set

```
import java.util.*;

public class TestTreeSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    TreeSet<String> treeSet = new TreeSet<String>(set);
    System.out.println("Sorted tree set: " + treeSet);

    // Use the methods in SortedSet interface
    System.out.println("first(): " + treeSet.first());
    System.out.println("last(): " + treeSet.last());
    System.out.println("headSet(\"New York\"): " +
      treeSet.headSet("New York"));
    System.out.println("tailSet(\"New York\"): " +
      treeSet.tailSet("New York"));

    // Use the methods in NavigableSet interface
    System.out.println("lower(\"P\"): " + treeSet.lower("P"));
    System.out.println("higher(\"P\"): " + treeSet.higher("P"));
    System.out.println("floor(\"P\"): " + treeSet.floor("P"));
    System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
    System.out.println("pollFirst(): " + treeSet.pollFirst());
    System.out.println("pollLast(): " + treeSet.pollLast());
    System.out.println("New tree set: " + treeSet);
  }
}
```

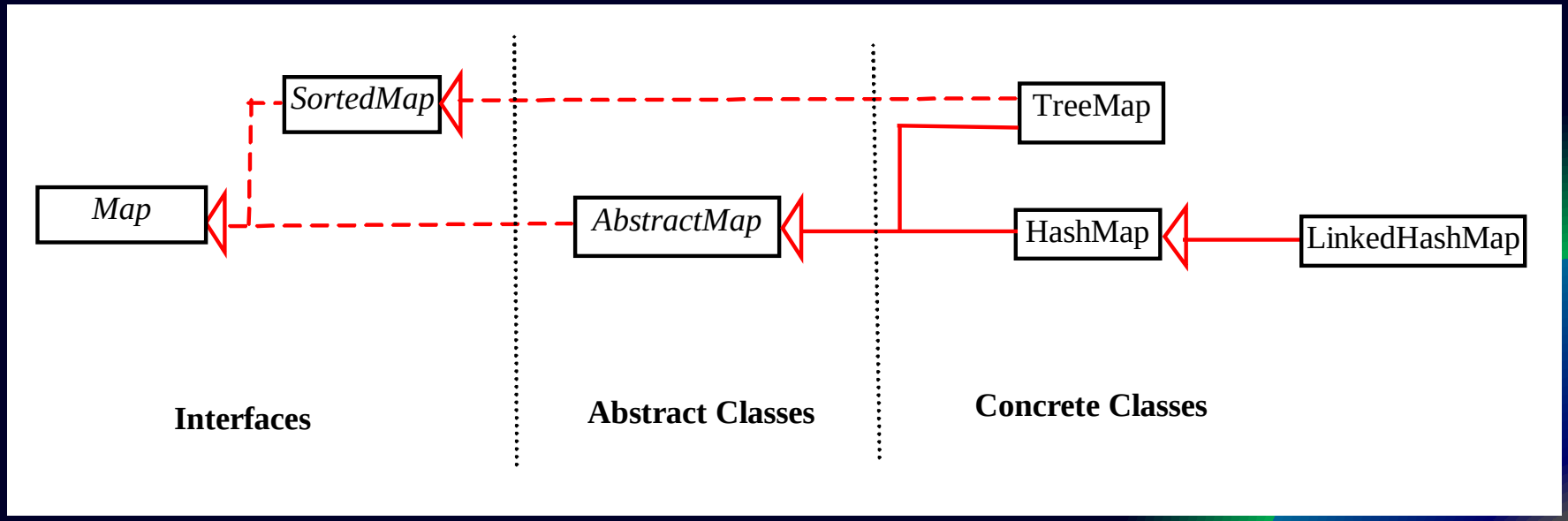
The Map Interface

- The Map interface maps keys to the elements. The keys are like indexes.
- The keys can be any objects.



Map Interface and Class Hierarchy

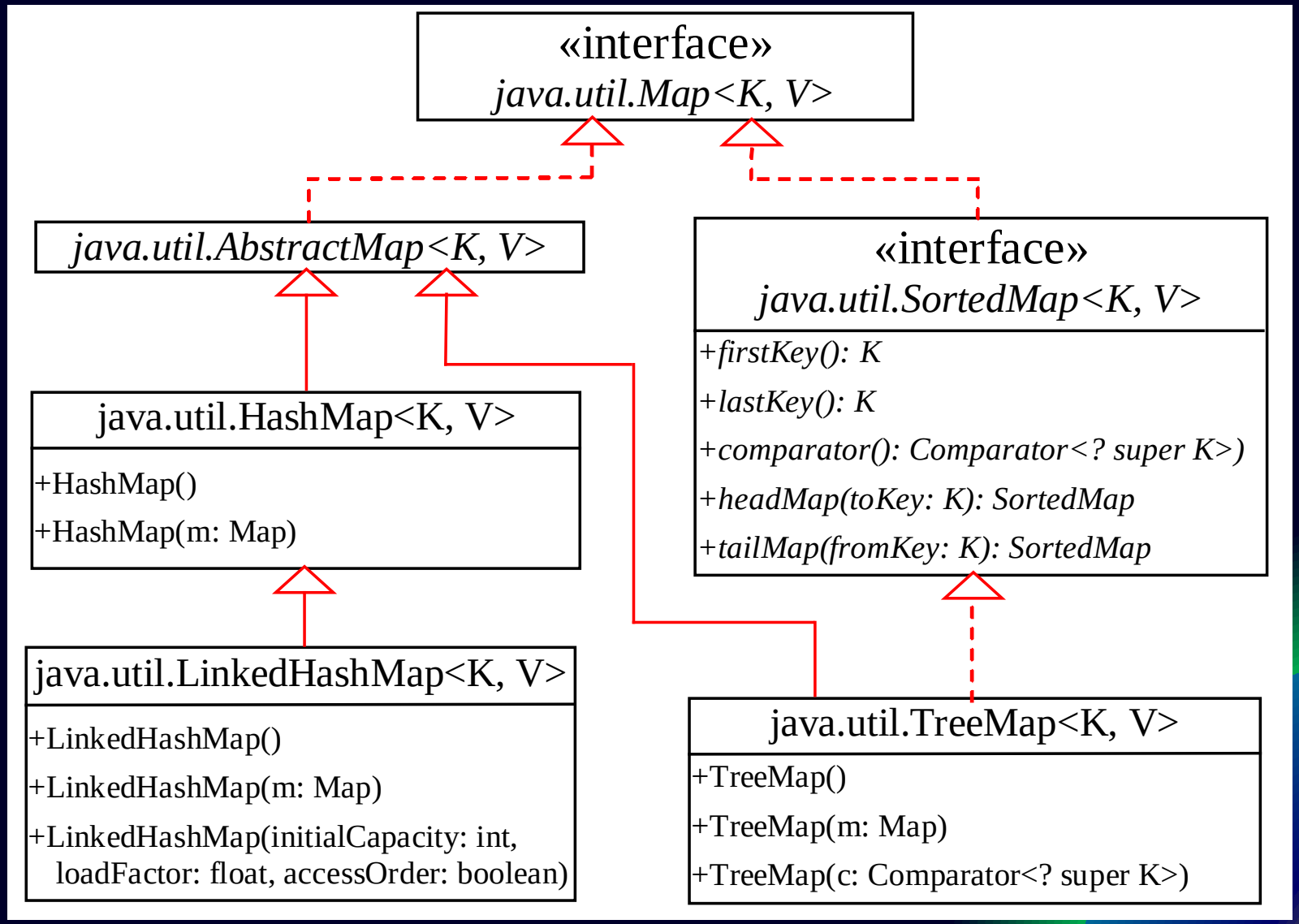
- An instance of Map represents a group of objects, each of which is associated with a key.
- You can get the object from a map using a key, and you have to use a key to put the object into the map.



The Map Interface UML Diagram

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.

Concrete Map Classes



HashMap and TreeMap

- 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

- 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).

Example: Using HashMap and TreeMap

```
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<String, Integer>();
    hashMap.put("Smith", 30);
    hashMap.put("Anderson", 31);
    hashMap.put("Lewis", 29);
    hashMap.put("Cook", 29);

    System.out.println("Display entries in HashMap");
    System.out.println(hashMap + "\n");

    // Create a TreeMap from the preceding HashMap
    Map<String, Integer> treeMap =
      new TreeMap<String, Integer>(hashMap);
    System.out.println("Display entries in ascending order of key");
    System.out.println(treeMap);

    // Create a LinkedHashMap
    Map<String, Integer> linkedHashMap =
      new LinkedHashMap<String, Integer>(16, 0.75f, true);
    linkedHashMap.put("Smith", 30);
    linkedHashMap.put("Anderson", 31);
    linkedHashMap.put("Lewis", 29);
    linkedHashMap.put("Cook", 29);

    // Display the age for Lewis
    System.out.println("\nThe age for " + "Lewis is " +
      linkedHashMap.get("Lewis"));

    System.out.println("Display entries in LinkedHashMap");
    System.out.println(linkedHashMap);
  }
}
```

```
Display entries in HashMap
{Lewis=29, Smith=30, Cook=29, Anderson=31}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}

The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

Programming Challenge

Perform set operations on hash sets:

Create two linked hash sets {**"George"**,**"Jim"**, **"John"**, **"Blake"**, **"Kevin"**, **"Michael"**} and {**"George"**, **"Katie"**, **"Kevin"**, **"Michelle"**, **"Ryan"**} and find their union, difference, and intersection.

(You can clone the sets to preserve the original sets from being changed by these set methods.)