

SET & MAP COLLECTION SORTING

Instructor:



◇ Map Collection

- ◇ HashMap

- ◇ TreeMap

◇ Set Collection

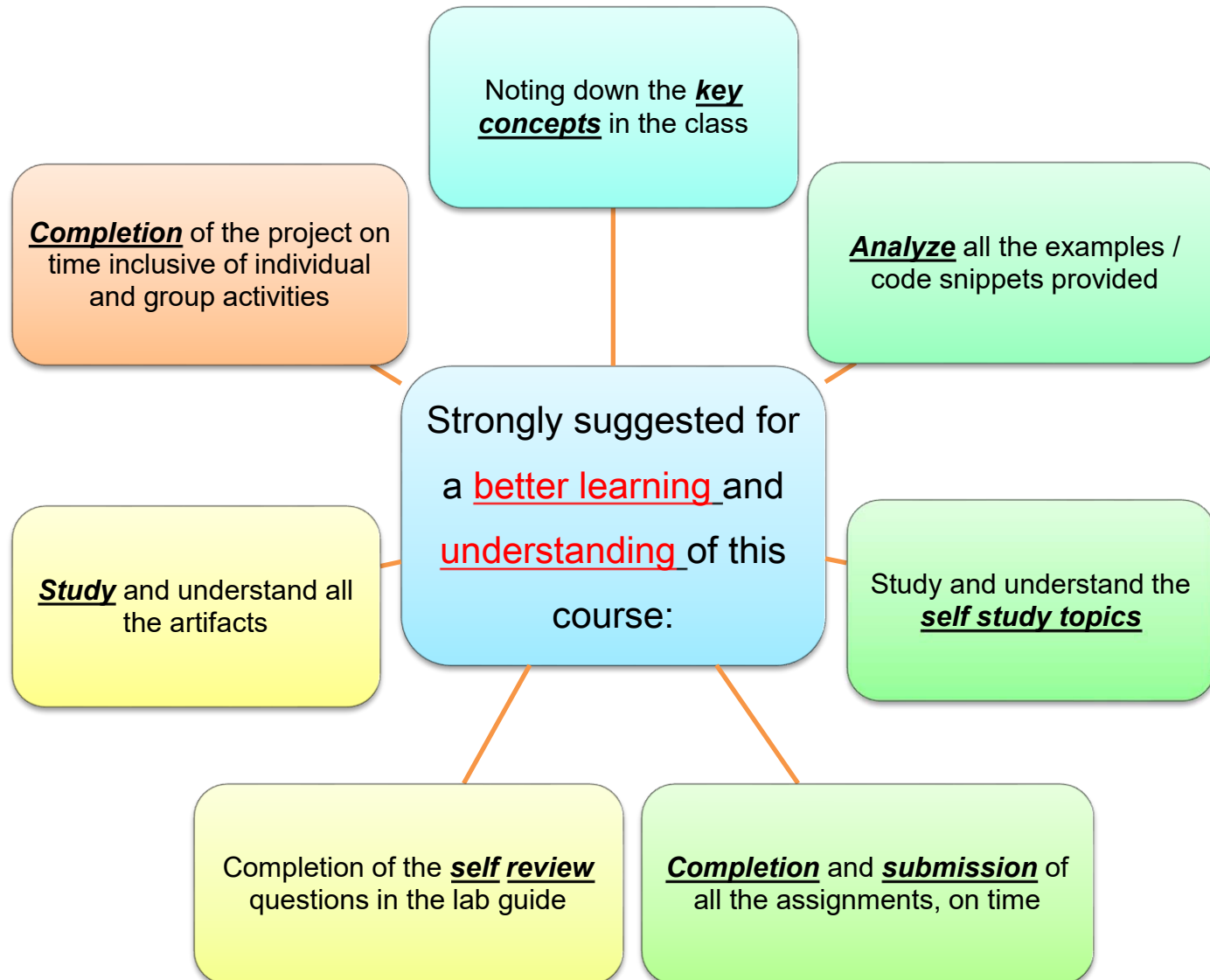
- ◇ HashSet

- ◇ TreeSet

◇ Collection Sorting

- ◇ Comparable

- ◇ Comparator

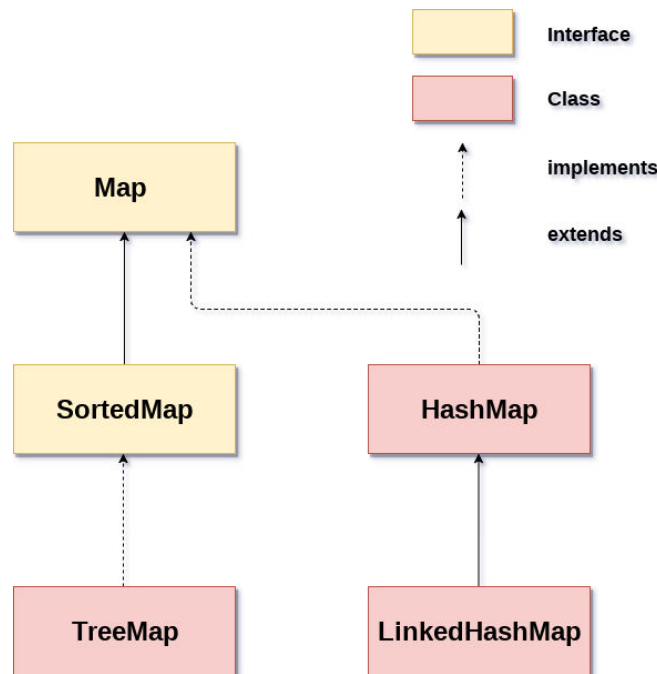


Section 1

MAP COLLECTION

Map Interface

- ❖ A map contains values on the basis of key, i.e. **key** and **value** pair. Each key and value pair is known as an entry. A Map contains unique keys.
- ❖ A Map is useful if you have to search, update or delete elements on the basis of a key.
- ❖ There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: **HashMap**, **LinkedHashMap**, and **TreeMap**. The hierarchy of Java Map is given below:



Map Interface

- ❖ A Map **doesn't allow duplicate keys**, but you **can have duplicate values**.
- ❖ **HashMap and LinkedHashMap allow null keys and values**, but TreeMap doesn't allow any null key or value.
- ❖ A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order .
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order .

- ❖ Java HashMap class implements the Map interface which allows us to store **key** and **value** pair, where **keys should be unique**.
 - ✓ If you try to insert the duplicate key, it will **replace the element** of the corresponding key.
 - ✓ It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the **java.util** package.
- ❖ HashMap in Java is like the legacy Hashtable class, but it is **not synchronized**.
 - ✓ It allows us to store the null elements as well, but there should be **only one null key**.
 - ✓ Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

❖ Points to remember:

- 1 • Java HashMap contains values based on the key.
- 2 • Java HashMap contains only unique keys.
- 3 • Java HashMap may have one null key and multiple null values.
- 4 • Java HashMap is non synchronized.
- 5 • Java HashMap maintains no order.
- 6 • The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Main methods of HashMap

Method	Description
<code>void clear()</code>	It is used to remove all of the mappings from this map.
<code>boolean isEmpty()</code>	It is used to return true if this map contains no key-value mappings.
<code>Set entrySet()</code>	It is used to return a collection view of the mappings contained in this map.
<code>Set keySet()</code>	It is used to return a set view of the keys contained in this map.
<code>V put(Object key, Object value)</code>	It is used to insert an entry in the map.
<code>void putAll(Map map)</code>	It is used to insert the specified map in the map.
<code>V putIfAbsent(K key, V value)</code>	It inserts the specified value with the specified key in the map only if it is not already specified.
<code>V remove(Object key)</code>	It is used to delete an entry for the specified key.
<code>boolean remove(Object key, Object value)</code>	It removes the specified values with the associated specified keys from the map.

Main methods of HashMap

Method	Description
<code>boolean containsValue(Object value)</code>	This method returns true if some value equal to the value exists within the map, else return false.
<code>boolean containsKey(Object key)</code>	This method returns true if some key equal to the key exists within the map, else return false.
<code>V get(Object key)</code>	This method returns the object that contains the value associated with the key.
<code>V replace(K key, V value)</code>	It replaces the specified value for a specified key.
<code>boolean replace(K key, V oldValue, V newValue)</code>	It replaces the old value with the new value for a specified key.
<code>void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)</code>	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
<code>Collection<V> values()</code>	It returns a collection view of the values contained in the map.
<code>int size()</code>	This method returns the number of entries in the map.

HashMap Example

```
public class HashMapExample {  
    public static void main(String args[]) {  
        HashMap<Integer, String> map = new HashMap<Integer, String>(); // Creating  
                                                                           // HashMap  
  
        map.put(1, "Mango"); // Put elements in Map  
        map.put(2, "Apple");  
        map.put(3, "Banana");  
        map.put(1, "Grapes"); // trying duplicate key  
  
        System.out.println("Iterating Hashmap...");  
  
        for (Entry<Integer, String> m : map.entrySet()) {  
            System.out.println(m.getKey() + " " + m.getValue());  
        }  
    }  
}
```

Output:

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

Working of HashMap in Java

- ❖ HashMap uses a technique called Hashing.
- ❖ HashMap contains an array of the nodes, and the node is represented as a class.
- ❖ It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

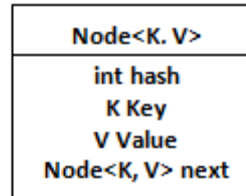


Figure: Representation of a Node

- ❖ Before understanding the internal working of HashMap, you must be aware of `hashCode()` and `equals()` method.
 - ✓ **equals()**: It checks the equality of two objects. It compares the Key, whether they are equal or not.
 - ✓ **hashCode()**: It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.

Working of HashMap in Java

- ❖ **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

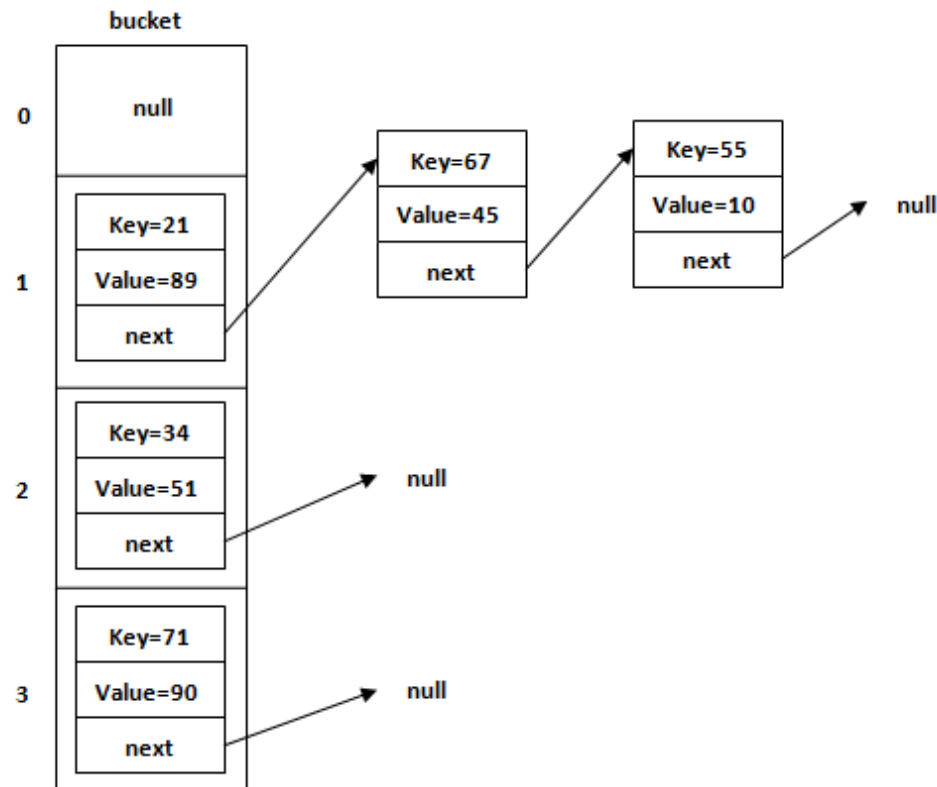


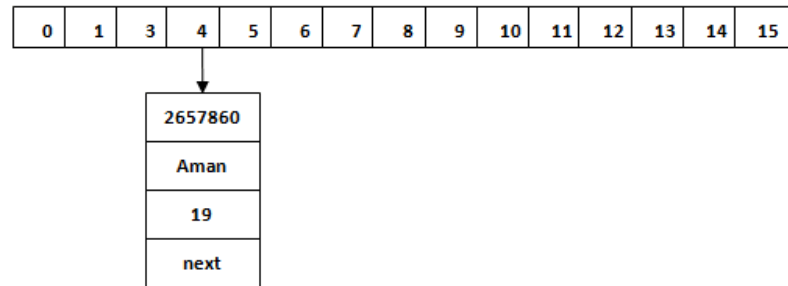
Figure: Allocation of nodes in Bucket

Working of HashMap in Java

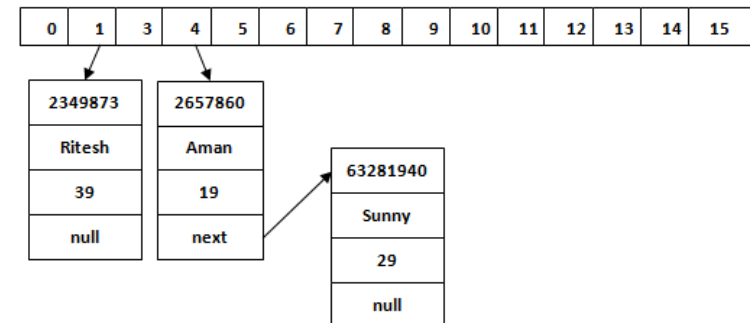
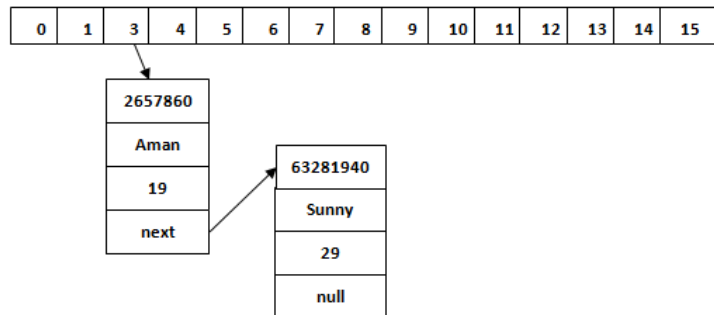
❖ Example

```
HashMap<String, Integer> map = new HashMap<>();  
map.put("Aman", 19);  
map.put("Sunny", 29);  
map.put("Ritesh", 39);
```

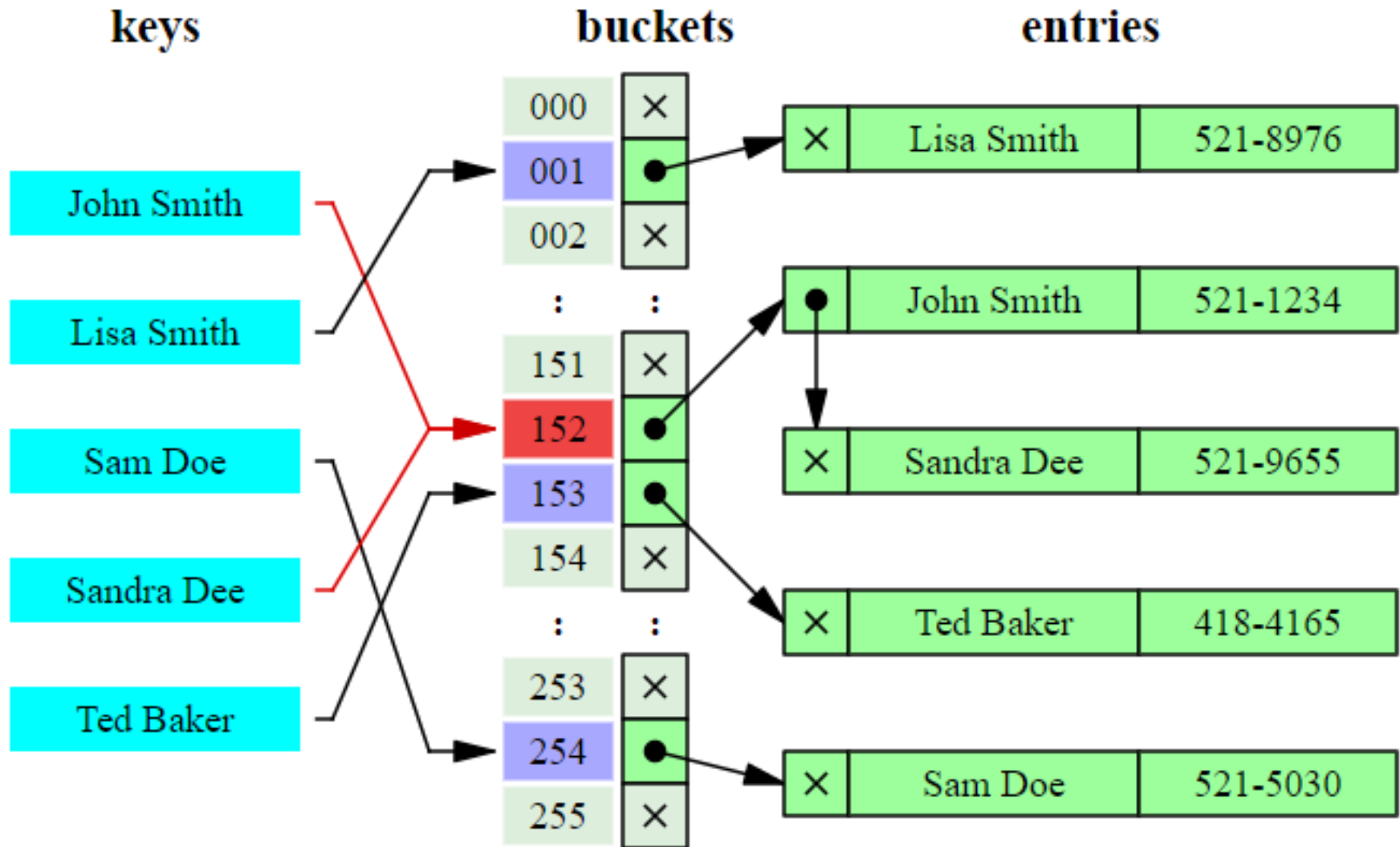
- ✓ When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is **2657860**. To store the Key in memory, we have to calculate the index.
- ✓ Index minimizes the size of the array. The Formula for calculating the index is: **Index = hashCode(Key) & (n-1) → Index = 2657860 & (16-1) = 4** (Where *n* is the size of the array).



- ✓ Suppose the hash code for "Sunny" is **63281940**. Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873.




Working of HashMap in Java



- ❖ Java TreeMap class is a **red-black tree** based implementation.
- ❖ It provides an efficient means of storing key-value pairs in **sorted order**.
- ❖ The important points about Java TreeMap class are:

- 1 • Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- 2 • Java TreeMap contains only unique elements.
- 3 • Java TreeMap cannot have a null key but can have multiple null values.
- 4 • Java TreeMap is non synchronized.
- 5 • Java TreeMap maintains ascending order.








Main methods of TreeMap

Method	Description
Map.Entry<K,V> ceilingEntry(K key)	It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
K ceilingKey(K key)	It returns the least key, greater than the specified key or null if there is no such key.
Map.Entry firstEntry()	It returns the key-value pair having the least key.
Map.Entry<K,V> floorEntry(K key)	It returns the greatest key, less than or equal to the specified key, or null if there is no such key.
Map.Entry<K,V> higherEntry(K key)	It returns the least key strictly greater than the given key, or null if there is no such key.
K higherKey(K key)	It is used to return true if this map contains a mapping for the specified key.
Set keySet() 	It returns the collection of keys exist in the map.
Map.Entry<K,V> lastEntry()	It returns the key-value pair having the greatest key, or null if there is no such key.

Main methods of TreeMap

Method	Description
Map.Entry<K,V> lowerEntry(K key)	It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K lowerKey(K key)	It returns the greatest key strictly less than the given key, or null if there is no such key.
Map.Entry<K,V> pollFirstEntry()	It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> pollLastEntry()	It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
V put(K key, V value)	It inserts the specified value with the specified key in the map.
void putAll(Map<? extends K, ? extends V> map)	It is used to copy all the key-value pair from one map to another map.
V replace(K key, V value)	It replaces the specified value for a specified key.

Main methods of TreeMap

Method	Description
K firstKey() 	It is used to return the first (lowest) key currently in this sorted map.
V get(Object key) 	It is used to return the value to which the map maps the specified key.
K lastKey() 	It is used to return the last (highest) key currently in the sorted map.
V remove(Object key) 	It removes the key-value pair of the specified key from the map.
Set<Map.Entry<K,V>> entrySet() 	It returns a set view of the mappings contained in the map.
int size() 	It returns the number of key-value pairs exists in the hashtable.
Collection values() 	It returns a collection view of the values contained in the map.

TreeMap Example

```
public class TreeMapExample {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
  
        TreeMap<Integer, String> map = new TreeMap<Integer, String>();  
        map.put(100, "Java");  
        map.put(102, "Net");  
        map.put(101, "Test");  
        map.put(103, "C++");  
        map.put(101, "Automation Test");  
  
        System.out.println("Before invoking remove() method");  
        for (Entry<Integer, String> m : map.entrySet()) {  
            System.out.println(m.getKey() + " " + m.getValue());  
        }  
  
        map.remove(102);  
        System.out.println("After invoking remove() method");  
        for (Entry<Integer, String> m : map.entrySet()) {  
            System.out.println(m.getKey() + " " + m.getValue());  
        }  
  
        map.put(104, "Android");  
        map.put(105, "iOS");  
  
        // Returns key-value pairs whose keys are less than the specified key.  
        System.out.println("headMap: " + map.headMap(104));  
        // Returns key-value pairs whose keys are greater than or equal to the  
        // specified key.  
        System.out.println("tailMap: " + map.tailMap(104, false));  
        // Returns key-value pairs exists in between the specified key.  
        System.out.println("subMap: " + map.subMap(100, 104));  
  
    }  
}
```

Difference between HashMap and TreeMap

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap cannot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Section 3

SORTING COLLECTION

- ❖ Java collection class is used exclusively with **static methods** that operate on or return collections. It inherits Object class.
- ❖ The important points about Java Collections class are:
 - ✓ Java Collection class supports the **polymorphic algorithms** that operate on collections.
 - ✓ Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.
- ❖ **Example:**

```
public class CollectionsExample {  
    public static void main(String a[]) {  
        List<String> list = new ArrayList<String>();  
        list.add("C");  
        list.add("Core Java");  
        list.add("Advanced Java");  
  
        System.out.println("Initial collection value: " + list);  
  
        Collections.addAll(list, "Servlet", "JSP");  
        System.out.println("After adding elements collection value: " + list);  
  
        String[] strArr = { "C#", "Net" };  
        Collections.addAll(list, strArr);  
        System.out.println("After adding array collection value: " + list);  
  
        Collections.sort(list);  
        System.out.println("After sorting array collection value: " + list);  
  
        System.out.println("Value of maximum element from the collection: " + Collections.max(list));  
    }  
}
```

Sorting in Collection

- ❖ We can sort the elements of:
 - ✓ String objects
 - ✓ Wrapper class objects
 - ✓ User-defined class objects
- ❖ **Collections** class provides static methods for sorting the elements of a collection.
- ❖ Example to sort string objects in reverse order

```
public class CollectionsSortExample {  
  
    public static void main(String[] args) {  
        ArrayList<Integer> al = new ArrayList<Integer>();  
  
        al.add(12);  
        al.add(3);  
        al.add(6);  
        al.add(37);  
        al.add(4);  
        al.add(9);  
  
        Collections.sort(al, Collections.reverseOrder());  
        System.out.println(al);  
    }  
}
```

Output: [37, 12, 9, 6, 4, 3]

Sorting user-defined class objects

- ❖ **Comparable** and **Comparator** both are interfaces and can be used to sort collection elements.
- ❖ They are used to order the objects of the user-defined class. These interfaces are found in **java.lang** package.
- ❖ **Comparable.compareTo(Object obj)** method: It is used to compare the current object with the specified object.
 - ✓ positive integer, if the current object is greater than the specified object.
 - ✓ negative integer, if the current object is less than the specified object.
 - ✓ zero, if the current object is equal to the specified object.
- ❖ **Comparator** contains 2 methods **compare(Object obj1, Object obj2)** and **equals(Object element)**.

ArrayList: Sort by Arrays

```
class A implements Comparable<A>{    // implement Comparable<T>
    int i;
    public int compareTo(A another){ // implement compareTo(T t)
        if (i == another.i) return 0;
        if (i < another.i) return -1; // sort theo tăng dần
        return 1;
    }
}

Object[] arA = alA.toArray();    // convert to array
Arrays.sort(arA);                // using Arrays.sort
for (Object a: arA){
    A a1 = (A)a;                 // revert to original type
    System.out.println(a1.i);
}
```

Sorting user-defined class objects

❖ Example:

```
public class Course {  
    private String courseId;  
    private String subjectId;  
    private String courseCode;  
    private String courseTitle;  
    private int numOfCredits;  
  
    public Course() {  
  
    }  
  
    public Course(String courseId, String subjectId,  
        String courseCode,  
        String courseTitle,  
        int numOfCredits) {  
        super();  
        this.courseId = courseId;  
        this.subjectId = subjectId;  
        this.courseCode = courseCode;  
        this.courseTitle = courseTitle;  
        this.numOfCredits = numOfCredits;  
    }  
  
    @Override  
    public int compareTo(Course c) {  
  
        return (this.numOfCredits - c.numOfCredits);  
    }  
}
```

Sorting user-defined class objects

❖ Example:

```
public class CourseExample {  
  
    public static void main(String[] args) {  
        List<Course> courses = new ArrayList<>();  
  
        Course c1 = new Course("11111", "CSCI", "1301", "Introduction to Java I", 4);  
        Course c2 = new Course("11112", "CSCI", "1302", "Introduction to Java II", 3);  
        Course c3 = new Course("11115", "MATH", "2750", "Calculus I", 5);  
        Course c4 = new Course("11117", "EDUC", "1111", "Reading", 3);  
        Course c5 = new Course("11118", "ITEC", "1344", "Database Administration", 3);  
  
        courses.add(c1);  
        courses.add(c2);  
        courses.add(c3);  
        courses.add(c4);  
        courses.add(c5);  
  
        courses.forEach(c -> System.out.println(c));  
  
        Collections.sort(courses);  
  
        System.out.println("After sorting:");  
  
        courses.forEach(c -> System.out.println(c));  
    }  
}
```

- ❖ Sorting the elements of list on the basis of **courseTitle** using Comparator.

```
public class CourseTitleCompare implements Comparator<Course> {  
  
    @Override  
    public int compare(Course o1, Course o2) {  
  
        return o1.getCourseTitle().compareTo(o2.getCourseTitle());  
    }  
  
}
```

Difference between Comparable and Comparator

- **Comparable** and **Comparator** both are interfaces and can be used to sort collection elements.
- But there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides single sorting sequence . In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides multiple sorting sequence . In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable affects the original class i.e. actual class is modified.	Comparator doesn't affect the original class i.e. actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
4) Comparable is found in java.lang package.	Comparator is found in java.util package.
5) We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List,Comparator) method.

Section 2

SET COLLECTION

- ❖ **Set** interface in Java is present in *java.util* package.
 - ✓ It extends the Collection interface.
 - ✓ It represents the **unordered** set of elements which doesn't allow us to store the duplicate items.
 - ✓ We can store at most one null value in Set.
 - ✓ Set is implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.

- ❖ **Set can be instantiated as:**

```
Set<data-type> s1 = new HashSet<data-type>();  
Set<data-type> s2 = new LinkedHashSet<data-type>();  
Set<data-type> s3 = new TreeSet<data-type>();
```

- ❖ Java **HashSet** class is used to create a collection that uses a hash table for storage.
 - ✓ It inherits the AbstractSet class and implements Set interface.
- ❖ The **important points** about Java HashSet class are:
 - ✓ HashSet stores the elements by using a mechanism called **hashing**.
 - ✓ HashSet contains **unique elements** only.
 - ✓ HashSet **allows null** value.
 - ✓ HashSet class is **non synchronized**.
 - ✓ HashSet **doesn't** maintain the **insertion order**. Here, elements are inserted on the basis of their hashcode.
 - ✓ HashSet is the best approach for search operations.
 - ✓ The initial default capacity of HashSet is 16, and the load factor is 0.75.

❖ Give **Course** class:

```
public class Course implements Comparable<Course> {  
    private String courseId;  
    private String subjectId;  
    private String courseCode;  
    private String courseTitle;  
    private int numOfCredits;  
  
    public Course() {  
    }  
  
    public Course(String courseId, String subjectId, String courseCode,  
        String courseTitle, int numOfCredits) {  
        super();  
        this.courseId = courseId;  
        this.subjectId = subjectId;  
        this.courseCode = courseCode;  
        this.courseTitle = courseTitle;  
        this.numOfCredits = numOfCredits;  
    }  
}
```

❖ Consider the following snippet code:

```
public class HashSet2 {  
  
    public static void main(String[] args) {  
        HashSet<Course> courses = new HashSet<>();  
  
        Course c1 = new Course("1122", "J001", "JSE",  
                                "Java SE Programming Essentials", 10);  
        Course c2 = new Course("1124", "S004", "SQLE", "SQL", 5);  
        Course c3 = new Course("1123", "F003", "FEE", "Front End Essentials", 10);  
        Course c4 = new Course("1124", "S004", "SQLE", "SQL", 5);  
        courses.add(c1);  
        courses.add(c2);  
        courses.add(c3);  
        courses.add(c4);  
  
        System.out.println("Size of set: " + courses.size());  
    }  
}
```

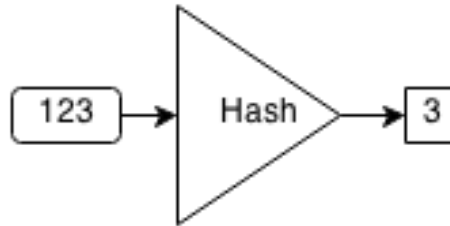
Output:

Size of set: 4

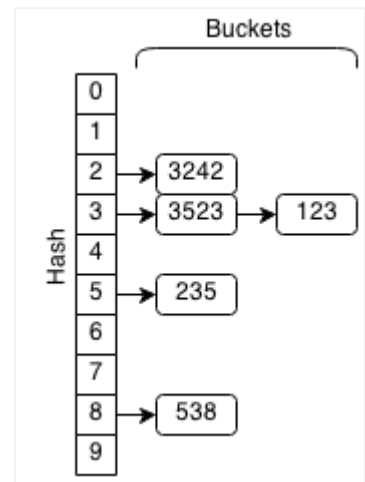
Why? c1 and c4 are the same values.
Set doesn't allow us to store the duplicate items.

Working of HashSet in Java

- ❖ **HashSet** uses hashing algorithms to **store**, **remove**, and **retrieve** its elements.
 - ✓ When an object is added to the Set, its hash code is used to choose a “bucket” into which to place the object.
 - ✓ Example: `set.add (123);`

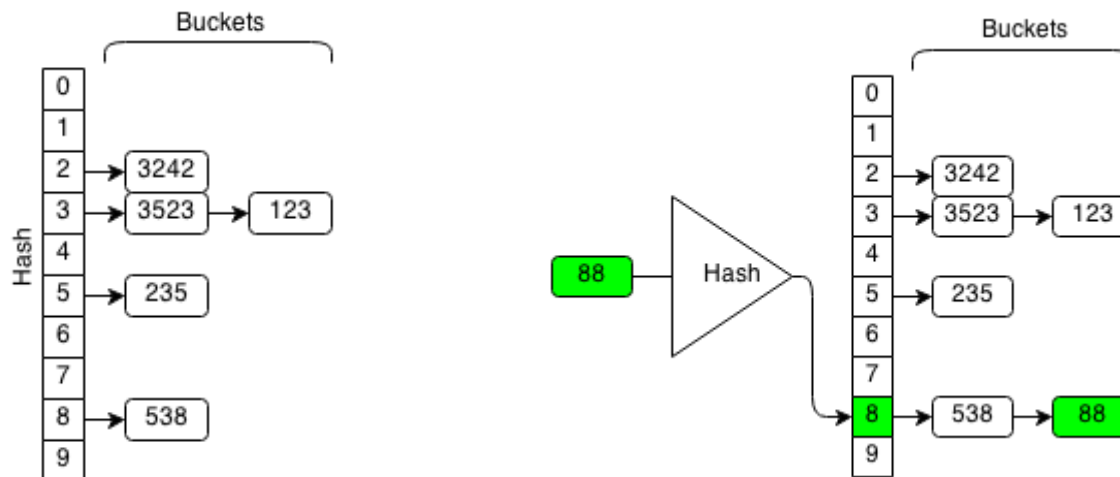


- ❖ For every element in a hash set, the hash is computed and elements with the same hash are grouped together. This group of similar hashes is **called a bucket** and they are usually stored as linked lists.



Working of HashSet in Java

- ❖ If have same hashcode then the **equals()** method will be called.
 - ✓ return **true**: the call leaves the set unchanged and returns false.
 - ✓ Return **false**: grouped together of similar hashes is **called a bucket** and they are usually stored as linked lists.
- ❖ **Example**, if we want to insert **88** in the following hash set:
 - ✓ We compute the hash of 88 which is 8, and we insert it to the end of the bucket with hash 8.



equals() and hashCode()

- ❖ To store not duplicate the user-defined class objects into HashSet: you must overriding equals() and hashCode() methods:

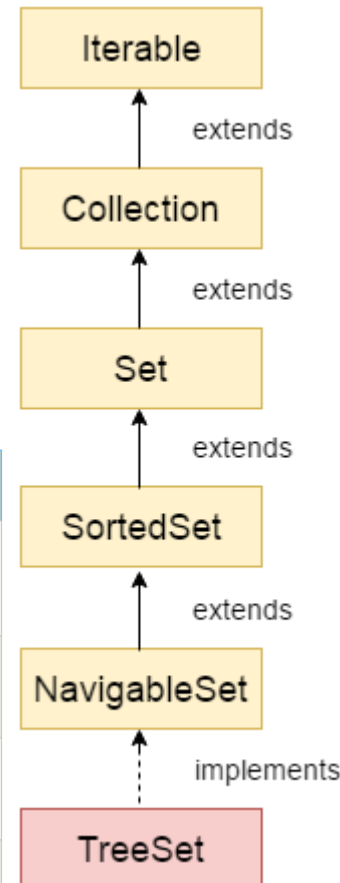
```
@Override
public int hashCode() {
    System.out.println("hashCode");
    final int prime = 31;
    int result = 1;
    result = prime * result
        + ((courseCode == null) ?
            0 : courseCode.hashCode());
    result = prime * result + ((courseId == null) ?
        0 : courseId.hashCode());
    result = prime * result
        + ((courseTitle == null) ?
            0 : courseTitle.hashCode());
    result = prime * result + numOfCredits;
    result = prime * result + ((subjectId == null) ?
        0 : subjectId.hashCode());
    return result;
}
```

```
@Override
public boolean equals(Object obj) {
    System.out.println("equals(): " + obj);
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Course other = (Course) obj;
    if (courseCode == null) {
        if (other.courseCode != null)
            return false;
    } else if (!courseCode.equals(other.courseCode))
        return false;
    if (courseId == null) {
        if (other.courseId != null)
            return false;
    } else if (!courseId.equals(other.courseId))
        return false;
    return true;
}
```

- ❖ Java **TreeSet** class implements the Set interface that **uses a tree for storage**. It inherits *AbstractSet* class and implements the *NavigableSet* interface. The objects of the TreeSet class are stored in *ascending order*.
- ❖ The **important points** about Java TreeSet class are:
 - ✓ Java TreeSet class contains **unique elements only** like HashSet.
 - ✓ Java TreeSet class **access and retrieval times are quite fast**.
 - ✓ Java TreeSet class **doesn't allow null element**.
 - ✓ Java TreeSet class is **non synchronized**.
 - ✓ Java TreeSet class maintains **ascending order**.

❖ Constructors:

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.



Methods of Java TreeSet class

Method	Description
boolean add (E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll (Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E ceiling (E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Iterator descendingIterator ()	It is used iterate the elements in descending order.
NavigableSet descendingSet ()	It returns the elements in reverse order.
E floor (E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet (E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet (E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher (E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.

Methods of Java TreeSet class

Method	Description
E lower (E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst ()	It is used to retrieve and remove the lowest(first) element.
E pollLast ()	It is used to retrieve and remove the highest(last) element.
SortedSet tailSet (E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet (E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
E first ()	It returns the first (lowest) element currently in this sorted set.
E last ()	It returns the last (highest) element currently in this sorted set.
int size ()	It returns the number of elements in this set.

TreeSet Example

```
public class TreeSetTest {  
  
    public static void main(String[] args) {  
        // Creating HashSet and adding elements  
        TreeSet<String> set = new TreeSet<>();  
        set.add("A");  
        set.add("C");  
        set.add("B");  
        set.add("E");  
        set.add("D");  
        Iterator<String> i = set.iterator();  
        while (i.hasNext()) {  
            System.out.println(i.next());  
        }  
    }  
}
```

Output:

A
B
C
D
E

TreeSet Example

```
public class TreeSetTest2 {  
  
    public static void main(String[] args) {  
        TreeSet<Integer> set = new TreeSet<Integer>();  
        set.add(24);  
        set.add(66);  
        set.add(12);  
        set.add(15);  
        System.out.println("Initial Set: " + set);  
  
        System.out.println("Reverse Set: " + set.descendingSet());  
  
        System.out.println("Highest Value: " + set.pollFirst());  
        System.out.println("Lowest Value: " + set.pollLast());  
    }  
}
```

Output:

Initial Set: [12, 15, 24, 66]

Reverse Set: [66, 24, 15, 12]

Highest Value: 12

Lowest Value: 66

◇ Set Collection

- ◇ HashSet

- ◇ TreeSet

◇ Map Collection

- ◇ HashMap

- ◇ TreeMap

◇ Collection Sorting

- ◇ Comparable

- ◇ Comparator

Thank you

