**COLLECTIONS IN JAVA:**

* Collection in java is a framework which represents, manipulates and stores the group of objects.
* Collection framework have many important interfaces like List, Set, Map and Queue and important classes like ArrayList, LinkedList, PriorityQueue, HashSet, LinkedHashSet and TreeSet classes.
* Collection hierarchy follows as below:

hierarchy of collection framework

* Collection framework classes and interfaces are present in the java.util package.
* There are many methods available in the Collection interface below are the list.

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| public boolean add(Object obj) | This method adds the specified element into current collection. |
| public boolean addAll(Collection c) | Adds all the elements of the specified collection into the invoking collection. |
| public boolean remove(Object obj) | This method removes the specified element from the invoking collection. |
| public boolean removeAll(Collection c) | This method removes all the elements of the specified collection from the invoking collection. |
| Public boolean retainAll(Collection c) | This method removes all the elements of the invoking collection except the specified collection. |
| public int size() | Returns the total number of elements in the collection. |
| public void clear() | Removes all the elements from the collection. |
| public boolean contains(Object obj) | This method is used to search an element in the collection. |
| public boolean containsAll(Collection c) | This method is used to search the specified collection in the invoking collection. |
| public Iterator iterator() | Returns an iterator for the invoking collection. |
| public Object[] toArray() | Converts the collection objects into an array. |
| public boolean isEmpty() | Checks if the collection is empty. |
| public boolean equals(Object obj) | Used to Match the collection elements. |
| public int hashCode() | Returns the hashCode number for collection. |

**Iterator Interface:**

* This interface helps to iterate the elements in the forward direction only and below are the three important methods available in the Iterator interface.

|  |  |
| --- | --- |
| **METHOD NAME** | **DESCRIPTION** |
| public boolean hasNext() | This method returns true if the collection has some more elements. |
| public Object next() | Returns the current element and points the cursor to the next element. |
| public void remove() | Removes the last returned element by the iterator. |

**ARRAYLIST:**

* Below is the declaration of ArrayList class:

**public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable.

* ArrayList internally uses dynamic array for storing the elements.
* ArrayList can contain duplicate elements.
* ArrayList maintains insertion order.
* ArrayList class is non-synchronized.
* ArrayList can be accessed randomly, because it internally uses Array and data can be fetched on the index basis.
* For manipulations ArrayList is slow, because lot of swapping has to be done if one element is deleted from the ArrayList.
* ArrayList can contain null elements as well.

**Below are the methods available in the ArrayList Class:**

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| void add(int index, Object e) | This method adds the object e in the specified index. |
| boolean add(Object e) | Adds the specified object e in the collection at the end. |
| boolean addAll(int index, Collection c) | Adds the specified collection to the invoking collection from the specified index. |
| boolean addAll(Collection c) | Adds the specified collection c to invoking collection. |
| boolean remove(Object o) | Removes the specified element from the invoking collection. |
| boolean removeAll(Collection c) | Removes all the elements of the Specified collection from the invoking collection. |
| boolean contains(Object o) | This method returns true if the specified element is present in the invoking collection. |
| boolean containsAll(Collection c) | This method returns true if all the elements of the specified collection is present in the invoking collection. |
| int size() | Returns the number of elements in the List. |
| Object get(int index) | Returns the object at the specified index. |
| void clear() | Removes all the elements from the Collection. |
| boolean isEmpty() | Returns true if the List is Empty else false. |
| int indexOf(Object o) | Returns the index of the specified object. |
| int lastIndexOf(Object o) | Returns the last index of the specified object. |
| Object clone() | Returns the exact copy of the collection object. |
| int hashCode() | Returns the hashCode for the specified collection. |
| Object[] toArray() | Returns the Collection elements into Object array. |
| boolean equals() | This method matches the two Collection objects and returns true if the number of elements and corresponding elements are same in the two collection objects. |
| boolean retainAll(Collection c) | This method removes all the elements from the invoking collection except the specified collection. |

**LINKEDLIST:**

* LinkedList class in java is used to store the list of objects and manipulate the same.
* LinkedList class uses double LinkedList as shown below to store the elements. As a result we can add the element or remove the elements from both ends.

java LinkedList class using doubly linked list

* Below is the Hierarchy for the LinkedList class in java.

Java LinkedList class hierarchy

* Below are the important point’s w.r.t LinkedList class.
* It can contain duplicate elements, null elements and it maintains insertion order.
* It is non-synchronized class.
* LinkedList class is bit fast as no shifting of the elements are required here.
* LinkedList class can be used as List, Queue and Stack.

Below are the important methods available in the LinkedList class along with the methods in the Collection interface.

|  |  |
| --- | --- |
| **Method Name** | **Method Purpose** |
| void add(int index, Object o) | Adds the specified element at the specified index. |
| boolean add(Object o) | Adds the element at the end of the List. |
| void addFirst(Object o) | Adds the specified element at the beginning of the List. |
| void addLast(Object o) | Adds the specified element at the end of the list. |
| Object getFirst() | Returns the first element of the List. |
| Object getLast() | Returns the Last element of the List. |
| int size() | Returns the size() of the List or returns the total number of elements of the list. |
| Object get(int index) | Returns the element at the specified index. |
| peek() | Returns the first element of the List and does not remove the same. |
| peekFirst() | Returns the first element of the List and returns null if list is empty. |
| peekLast() | Returns the last element of the List and returns null if list is empty. |
| poll() | This method returns the first element and removes the same. |
| pollFirst() | This method returns the first element and removes the same. This method returns null if the list is empty. |
| pollLast() | This method returns the last element and removes the same. Returns null if the list is empty. |
| pop() | This method removes the first element from the List and throws NoSuchElementException if the list is empty. |
| remove() | This method removes the first element and returns the same. |
| push() | Adds the element to the beginning of the List. |

* Remaining methods addAll(), removeAll(), contains(),containsAll(), retainAll(), equals(),hashCode(), toArray(), isEmpty(), clear(),iterator() remains same as of the Collection interface and ArrayList class. Please refer the project CollectionsDemo2-CollectionsDemoLinkedList2 and CollectionsDemoLinkedList.

**Below are the Differences between ArrayList and LinkedList:**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList internally uses Dynamic Array for Storing the elements. | LinkedList internally uses Double Linked List for storing the elements. |
| ArrayList can act as only List as it implements only List interface. | LinkedList can act as both Queue and List as it implements both Queue and List Interfaces. |
| Performance wise if we remove a element from the ArrayList, lot of shiftings has to be done since, it uses Dynamic array. | LinkedList is better than the ArrayList Performance wise, since no shifting’s has to be done after removing the elements. |
| For storing and retrieving the objects, ArrayList is better. | For manipulating the objects LinkedList is better. |

**LIST AND LISTITERATOR INTERFACE:**

* List interface is the sub-interface of the Collection interface. Below is the Hierarchy of the List Interface.
* **public** **interface** List<E> **extends** Collection<E>
* **Below are the important methods in the List Interface.**

|  |  |
| --- | --- |
| Method Name | Purpose |
| void add(int index, Object o) | Adds the specified object at the specified position. |
| void addAll(int index, Collection c) | Adds the specified collection from the specified index in the invoking collection. |
| Object get(int index) | Returns the object present in the specified index. |
| Object set(int index, Object obj) | Overrides the element if already present in the specified index. Otherwise it will insert the element at the specified index. |
| Object remove(int index) | Removes the object in the specified index and returns the same. |
| ListIterator listIterator() | Returns the iterator to the invoking list from the beginning. |
| ListIterator listIterator(int index) | Returns the iterator to the invoking list from the specified index. |

**LISTITERATOR:**

* With ListIterator we can traverse the elements in forward and backward directions.
* Below is the Hierarchy for ListIterator.

**public** **interface** ListIterator<E> **extends** Iterator<E>

* Below are the methods available in the ListIterator.

|  |  |
| --- | --- |
| **Method Name** | **Purpose** |
| hasNext() | Returns true if the if the list for which iterator is returned has some more elements. |
| next() | Returns the current element and moves the cursor to the next element. |
| hasPrevious() | Returns true if the list has some more elements. |
| previous() | Returns the current element and moves the cursor to the previous element. |

**HASHSET:**

* HashSet class is also for creating the collection, in which it uses Hash Table for storing the elements.
* HashSet contains unique elements and does not maintain insertion order and null elements can be present.
* HashSet internally uses hashing mechanism for storing the elements.
* Below is the hierarchy for the HashSet class.
* **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable.
* **Below is the important constructor of HashSet.**
* **HashSet(Collection c). This constructor takes the Collection elements as input.**

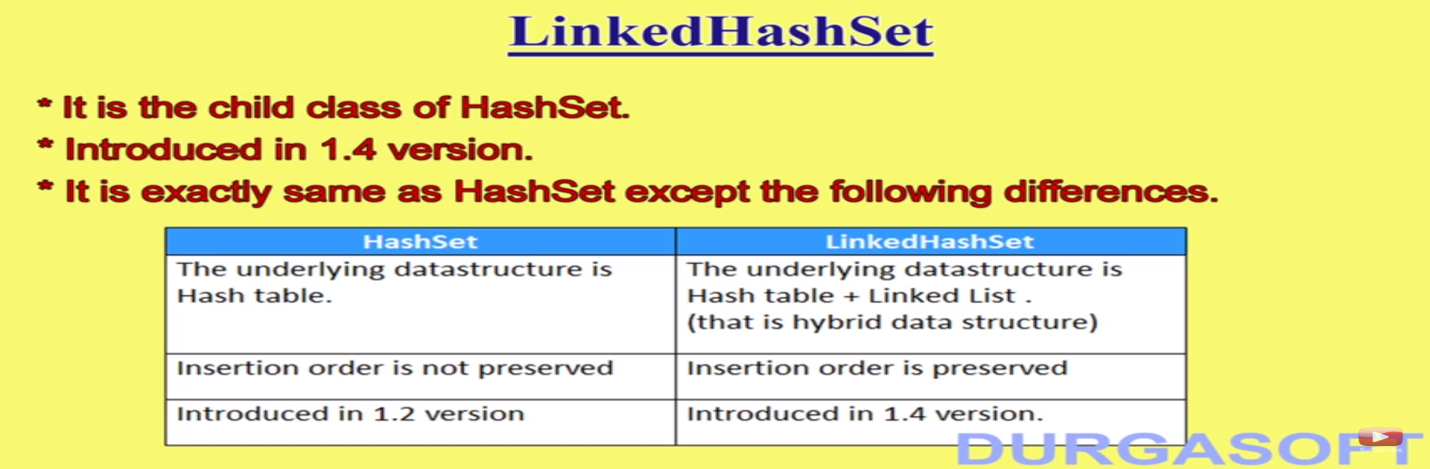
**Below are the important methods available in the HashSet class:**

|  |  |
| --- | --- |
| **Method Name** | **Purpose** |
| void clear() | Clears all the elements of the HashSet. |
| boolean add(Object o) | Adds the specified object to the HashSet if it is not already present. |
| boolean remove(Object o) | Removes the element from the HashSet if it is present and returns true or false based on the action. |
| boolean contains(Object o) | This method returns true if the HashSet contains the specified element. |
| int size() | Returns the total number of elements in the HashSet. |
| Object clone() | Returns the cloned HashSet Object. |
| Iterator iterator() | Returns the iterator for the HashSet. |
| boolean isEmpty() | This method returns true if the HashSet is Empty. |

* **The** main difference between Set and List is Set doesn’t allow duplicate elements whereas List allows duplicate Elements.

**LINKEDHASHSET:**

* LinkedHashSet class is also same as that of the HashSet class except few points as it maintains insertion order, can also contain null values, uses doubly Linked List for storing the elements.
* All the methods in the LinkedHashSet is same as the HashSet class.
* Below is the Hierarchy of the LinkedHashSet Class.
* **public** **class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable.



* *LinkedHashSet is used mainly in cache based applications where insertion order is required and duplicates are not allowed.*
* LinkedHashSet internally uses LinkedHashMap for storing the elements in insertion order.
* LinkedHashSet uses both HashTable and LinkedList Data Structures internally.

**TREESET:**

* **Underlying data structure in TreeSet is Balanced Tree(Red Black Tree).**
* TreeSet class is also for creating the collection and stores the elements in the Tree.
* TreeSet stores only unique elements.
* TreeSet cannot have null values.
* TreeSet stores the elements in the ascending the order.
* Accessing is faster in the TreeSet class.
* All the user defined objects except String and Wrapper class objects that stores in the TreeSet should be of type Comparable. That is every userdefined object should implement Comparable interface.
* Below is the hierarchy for TreeSet class.
* public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>,Clonable, Serializable.
* TreeSet internally uses TreeMap for storing the elements i.e. in addition, removing or retrieval of elements.
* There are four important constructors in TreeSet:
  + TreeSet<> t=new TreeSet<>(); ---creates a tree set with default natural sorting order i.e. numbers in ascending order, string or characters in alphabetical order.
  + TreeSet<> t=new TreeSet<>(Comparator c); ---creates a tree set and sorts the elements in customized sorting order specified by comparator.
  + TreeSet<> t=new TreeSet<>(Collection c); ---creates the tree set with the elements in the specified collection.
  + TreeSet<> t=new TreeSet<>(SortedSet s); ---creates the tree set with the elements in the specified SortedSet.
* Till JDK 1.6 null insertion in TreeSet is possible as a first element only. If we try to insert as any other element then we will get NullPointerException. And its identified as bug and corrected from JDK1.7.
* From JDK 1.7 we cannot have any null element in TreeSet.
* ***If we try to add the elements into the TreeSet that are not of type Comparable then we will get ClassCastException.***
* Whenever we add every element into the TreeSet internally compareTo() of particular class which type we are inserting will be executed. Any operations on null leads to NullPointerException.

TreeSet class hierarchy

* Below are the important methods in the TreeSet class.

|  |  |
| --- | --- |
| **Method Name** | **Purpose** |
| boolean add(Object o) |  |
| boolean remove(Object o) |  |
| boolean addAll(Collection c) |  |
| boolean removeAll(Collection c) |  |
| boolean contains(Object o) |  |
| boolean containsAll(Collection c) |  |
| int size() |  |
| boolean retainAll() |  |
| Object first() | **Returns the first element i.e. least element in the TreeSet.** |
| Object last() | **Returns the last element i.e. highest element in the TreeSet.** |
| boolean retainAll() |  |
| boolean isEmpty() |  |
| Object clone() |  |
| Object[] toArray() |  |
| int hashCode() |  |
| boolean equals(Collection c) |  |

**QUEUE:**

* **QUEUE** is used for creating the collection and stores the elements in the Queue in the FIFO (FirstInFirstOut Manner). The element first inserted will be removed first and element inserted at the last will be removed last.
* **Below is the** Hierarchy of the Queue.
* public interface Queue<E> extends Collection<E>
* Below are the all methods available in the Queue interface.

|  |  |
| --- | --- |
| **Method Name** | **Method Purpose** |
| public boolean add(Object e) | This method adds the element to the Queue and returns true upon success, else will throw IllegalStateExecption if the element cannot be added due to capacity restrictions. |
| boolean offer(Object o) | This method also adds the specified element to the Queue and returns true upon success, if it is not able to add at this time then it will return false. |
| Object remove() | This method retrieves and removes the head of the queue i.e. first element in the queue. It will throw NoSuchElementException if queue is empty. |
| Object poll() | This method retrieves and removes the head of the queue and will return null if queue is empty. |
| Object element() | This method returns the first element i.e. head of the queue and throws NoSuchElementException if queue is empty. |
| Object peek() | This method also returns the first element and does not remove the same and returns null if queue is empty. |

**PRIORITYQUEUE:**

* Java PriorityQueue class is used to create collection and doesn’t maintain FIFO structure.
* PriorityQueue class maintains default natural order, cannot have null values and can contain duplicate elements.
* All the elements inside the PriorityQueue should be of type Comparable, by default String and Wrapper classes objects are of type Comparable.
* The hierarchy of the PriorityQueue is as below.
* public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable.
* The initial capacity of PriorityQueue is 11 and Queue is introduced in JDK 1.5 version.
* **Please refer** CollectionsDemo2 - CollectionDemoQueueAndPriorityQueue.
* All the methods in the Queue interface is overridden in AbstractQueue and PriorityQueue classes.
* \*\*\*\*If we want the default natural sorting order then all the elements in the PriorityQueue should be of type comparable otherwise we will get ClassCastException.
* \*\*\*\* We can provide a comparator if we don’t want the default natural sorting order.

**Below are 5 important constructors in the PriorityQueue class:**

1. PriorityQueue queue=new PriorityQueue(); creates a PriorityQueue of size 11 and maintains default NaturalSortingOrder.
2. PriorityQueue queue=new PriorityQueue(int capacity); creates a PriorityQueue with the specified size and maintains default Natural Sorting order.
3. PriorityQueue queue=new PriorityQueue(int capacity, Comparator comparator); creates a new priorityqueue with the specified capacity and sorts based on the specified compartor sequence.
4. PriorityQueue queue=new PriorityQueue(SortedSet set); A new PriorityQueue is created for the specified sorted set.
5. PriorityQueue queue=new PriorityQueue(Collection c); A new PriorityQueue is created for the specified Collection.

**🡪Some platforms won’t provide support for the PriorityQueue i.e. default natural sorting order is not retained when we iterate the queue or when we print the queue elements. It’s specified in java documentation also.**

**Deque Interface:**

* Deque interface provides the flexibility for us to add and remove the elements in both ends. Name itself suggests Double Ended Queue (Deque).
* Below are the important methods available in the Deque interface which is extends Queue interface.
* Deque interface is implemented by LinkedList and ArrayDeque classes.

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| boolean add(Object o) | Adds the specified object into the queue at the end of the queue. |
| boolean addFirst(Object o) | Adds the specified object at the start of the queue. |
| boolean addLast(Object o) | Adds the specified object at the end of the queue. |
| boolean offer(Object o) | This method also adds the object into the queue and returns false if it is not able to add the element to the queue. |
| boolean offerFirst(Object o) | Adds the element at the head of the queue. |
| boolean offerLast(Object o) | Adds the specified object at the end of the queue. |
| Object remove() | Returns and removes the head of the queue. |
| Object removeFirst() | Returns and removes the head of the queue. |
| Object removeLast() | Returns and removes the tail of the queue. All the remove methods throws NoSuchElementException if the queue is empty. |
| Object poll() | Removes the head of the queue. |
| Object pollFirst() | Removes the head of the queue. |
| Object pollLast() | Removes the tail of the queue. All the forms of poll() returns null if the queue is empty. |
| Object element() | This method only returns the head and don’t remove the same. Throws NoSuchElementException if queue is empty. |
| Object peek() | This method also returns the head of the queue and will return null if queue is empty. |
| Object peekFirst() | This method also returns the head of the queue and will return null if queue is empty. |
| Object peekLast() | This method also returns the last element of the Queue and will not remove the same. It returns null if queue is empty. |
| Object pop() | This method also returns and removes the head of the queue. Similar to removeFirst(). |
| Object push() | This method adds the element to the head of the queue. Similar to addFirst() |

* In Addition to the methods available in the Collection interface, above methods are specific to the Queue.

**ARRAYDEQUE:**

* ArrayDeque uses Deque and Array of Resizable for storing the elements.
* In ArrayDeque we can add and remove the elements from both ends.
* We cannot have null values in the ArrayDeque.
* ArrayDeque class maintains insertion order.
* It is non-synchronized class.
* **It is faster than Stack and LinkedList and ArrayDeque can be used as Stack (LIFO) as well as Queue(FIFO).**
* **Whenever we think Deque implementation we should go for ArrayDeque as it is more faster than the LinkedList as Deque.**
* **Deque internally uses dynamic array and its circular and ArrayList cannot be used as Deque as lots of shifting has to be done and it impacts performance highly.**
* Below is the hierarchy of the ArrayDeque class.
* public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>.

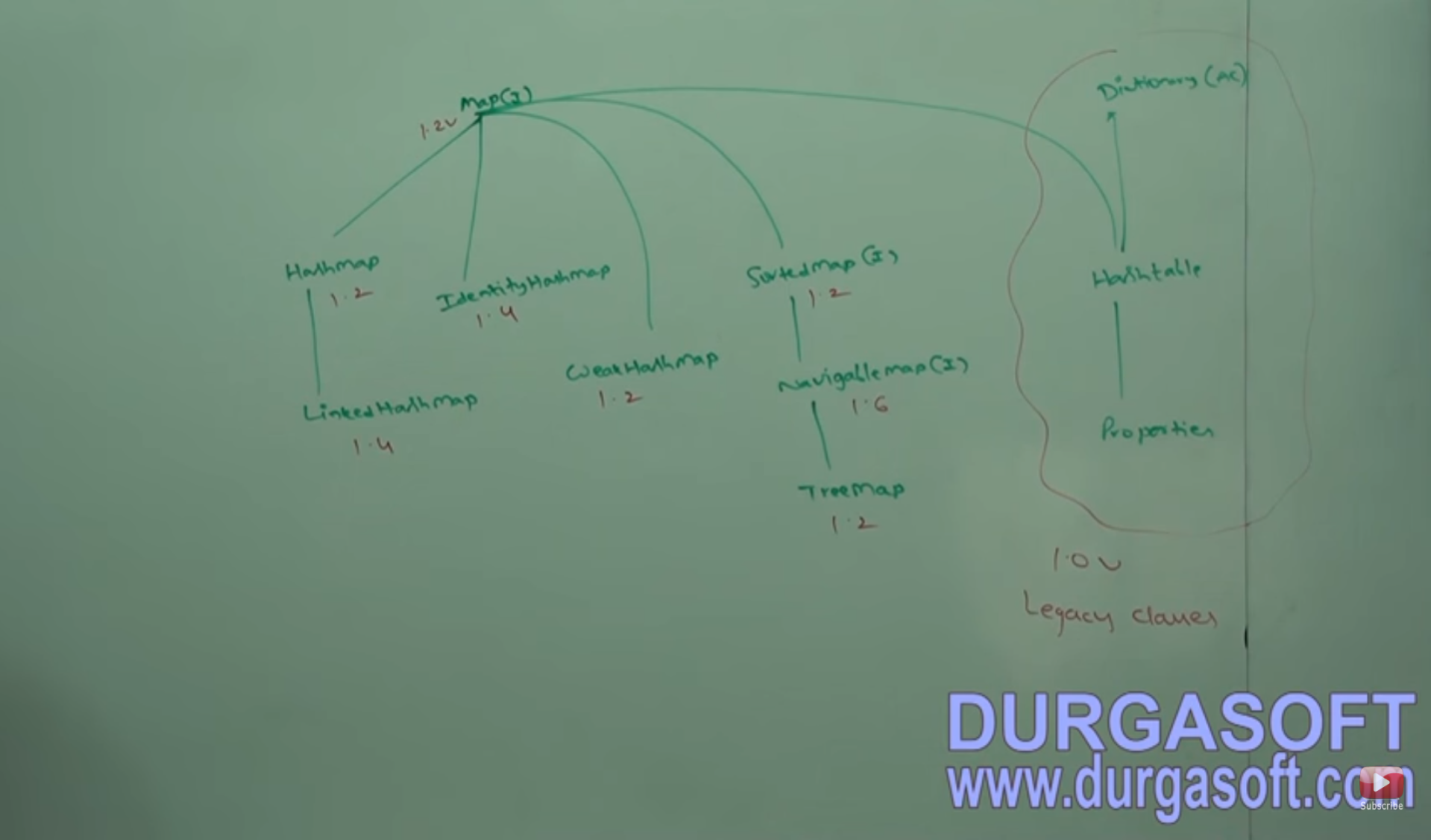
java arraydeque hierarchy

**Below is the Difference among List, Set and Queue.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Difference** | **List** | **Set** | **Queue** |
| Insertion Order | Yes | HashSet-No  LinkedHashSet-Yes  TreeSet-Ascending order | PriorityQueue- No  ArrayDeque-Yes |
| Null elements | Allowed | TreeSet- Not Allowed and in HashSet and LinkedHashSet allowed. | Not allowed |
| Duplicates | Allowed | No | Allowed |
| Does Objects should be of the comparable type | No | TreeSet objects should be of type Comparable. | PriorityQueue Objects should be of type Comparable. By default String and Wrapper class objects are of type comparable. |
| Internal Implementation | ArrayList- Dynamic Array  LinkedList- DoublyLinkedList  Vector- Dynamic Array.  Stack- Stack | HashSet- HashTable(HashMap)  LinkedHashSet-(LinkedHashMap)  TreeSet- Tree(Red Black Tree).(TreeMap) | PriorityQueue – Does not use Queue.  ArrayDeque –Uses Resizable Array and Deque. |
| Specific Methods | indexOf(), lastIndexOf(), add(int i, Object o), addAll(int i, Collection c), get(int i), set(int i, Object o), listIterator(), | first(), last() | offer(), offerFirst(), offerLast(), remove(), removeFirst(), removeLast(), poll(), pollFirst(), pollLast(), addFirst(), addLast(), getFirst(), getLast(), peek(), peekFirst(), peekLast(), push(), pop(), element(). |
| Default Initial Capacity | ArrayList- 10  Vector- 10 | HashSet -16  LinkedHashSet-16 | PriorityQueue- 11  ArrayDeque -16 |

**MAP INTERFACE:**

* Map stores the values based on the keys.
* (Key + Value) is defined as the Entry.
* Map contains unique keys.
* Map is separate interface which doesn’t inherit Collection interface.
* Map doesn’t maintain insertion order.



* Below are the important methods available in the Map Interface.

|  |  |
| --- | --- |
| **Method Name** | **Method Purpose** |
| Object put(Object key, Object value) | This method is for adding an entry into the Map and returns null if the inserting key is not present otherwise returns Replaced value(old value) will be returned. |
| void putAll(Map map) | This method adds the specified Map into the invoking Map. |
| Object get(Object key) | Returns the value for the specified key. |
| Object remove(Object key) | Returns and Removes the entry for the specified key. |
| boolean containsKey(Object key) | This method returns true if the specified key is present in the map. |
| Set keySet() | This method returns the set view of the keys in a map. |
| Set entrySet() | This method returns the set view of the entries in a map. |
| int size() | Returns the number of entries in the Map. |
| boolean isEmpty() | Checks if the Map is empty or not. |
| void clear() | Removes all the entries of the Map. |
| boolean containsValue(Object o) | Returns true if the Map contains the specified value. |

**ENTRY:**

* This is a sub interface present in the Map interface and can be accessed by Map.Entry. Entry interface is for the purpose of the getting the keys and values from the Map.

**Below are two important methods available in the Entry interface:**

|  |  |
| --- | --- |
| **Method Name** | **Method description** |
| Object getKey() | Returns the key. |
| Object getValue() | Returns the value. |

**HASHMAP CLASS:**

* HashMap class stores the values based on the keys.
* HashMap contains unique keys but can have duplicate values.
* HashMap doesn’t maintain insertion order and it is stored based on the hashcode of the keys.
* HashMap can have one null key and multiple null values.
* HashMap internally uses HashTable datastructure for storing keys and values.
* If our frequent operation is search then we should go for HashMap.

**Below are HashMap constructors:**

1. **HashMap map=new HashMap(); ---Creates an empty hash map with default initial capacity 16 and load factor- 0.75.**
2. **HashMap map=new HashMap(int initialcapacity)—customizes the initial capacity of the HashMap.**
3. **HashMap map=new HashMap(int initialcapacity, float loadfactor)—Customizes the initial capacity and load factor.**
4. **HashMap map=new HashMap(Map m); converts the specified map into the HashMap.**

* Below is the Hierarchy of the HashMap class.
* public class HashMap extends AbstractMap and implements Map.
* If we insert the model class object as a key inside the map, in this case we need override the equals () and hashCode() in model class otherwise duplicates keys can exist.
* **Below are some important methods in HashMap class along with the implementations of the methods in the Map interface.**

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| Object clone() | This method returns the exact copy of the Map. |
| Collection values() | This method returns the collection view of the values in the Map. |

**LINKEDHASHMAP CLASS:**

* LinkedHashMap class is same as HashMap class instead this maintains insertion order.
* LinkedHashMap internally uses Hash table and LinkedList implementation of the Map.
* Below is the Hierarchy of LinkedHashMap class.
* LinkedHashSet and LinkedHashMap are predominantly used in cache based applications where insertion order is important and duplicates not allowed.
* public class LinkedHashMap extends HashMap implements Map.

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| HashMap internally uses Hashtable data structure. | LinkedHashMap internally uses Hashtable and LinkedList Datastruture |
| HashMap does not maintain insertion order. | LinkedHashMap maintains insertion order. |
| It is introduced in 1.2 version | It is introduced in 1.4 Version. |
| HashMap inherits AbstractMap | LinkedHahMap inherits HashMap |

**IDENTITYHASHMAP:**

* IdentityHashMap is same as the HashMap except the below difference.

|  |  |
| --- | --- |
| **HashMap** | **IdentityHashMap** |
| For identifying duplicate keys equals() which does the content comparison is used. | For Identifying the duplicate keys == operator is used which does the reference comparison. |

**WEAKHASHMAP:**

* WeakHashMap is same as the HashMap except the below difference.

|  |  |
| --- | --- |
| **HashMap** | **WeakHashMap** |
| If the object does not have any reference and is only associated with HashMap then GC cannot remove the object from HashMap. Here HashMap dominates GC. | In WeakHashMap if the Object does not have any reference and is only associated with WeakHashMap, then GC collects the object for destruction and calls finalize () before destroys. Here GC dominates WeakHashMap. |

**SORTEDMAP:**

* If we want entries in the map to be stored in a certain sorting order then we will go for SortedMap.
* SortedMap is child interface of the Map interface.
* Below are the specific methods w.r.t SortedMap.

|  |  |
| --- | --- |
| **Method Name** | **Functionality** |
| Object firstKey() | Returns the firstKey() of the SortedMap. |
| Object lastKey() | Returns the lastKey() of the SortedMap |
| SortedMap headMap(Object key) | Returns the SortedMap < than the specified key. |
| SortedMap tailMap(Object key) | Returns the SortedMap >= specified key. |
| SortedMap subMap(Object k1, Object k2) | Returns the SortedMap >=k1 and lessthan k2. |
| Comparator comparator() | This method returns null if the elements are in default Natural Sorting order. |

**TREEMAP CLASS:**

* TreeMap class stores the data in the form of key and value pair.
* TreeMap stores the data in the ascending order and it is based on the key.
* TreeMap internally uses RED BLACK TREE for storing the data.
* TreeMap should not contain null keys, and can have multiple null values. Till 1.6 version null key is allowed as first entry in TreeMap if TreeMap is empty. From 1.7 onwards null keys are not allowed at any place in TreeMap.
* TreeMap contains unique elements.
* If we are depending on the default natural sorting order then all the keys in the TreeMap should be of type Comparable. Otherwise ClassCastException we will get. If we don’t need DNSO then keys need not be of type Comparable. There is no restriction for the values.

**Important Constructors of TreeMap:**

* **TreeMap m=new TreeMap(); --- Creates an Empty TreeMap with Default Natural Sorting order.**
* **TreeMap m=new TreeMap(Comparator c); --- Creates a TreeMap with Customized Sorting order.**
* **TreeMap m=new TreeMap(SortedMap m);-- Creates a TreeMap for the specified SortedMap.**
* **TreeMap m=new TreeMap(Map m);--- Creates a TreeMap for the specified Map.**
* Below is the Hierarchy of the TreeMap.

Java TreeMap class hierarchy

* public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>.
* Below 2 are the specific methods to the TreeMap class.

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| Object firstKey() | Returns the first Key of the TreeMap. |
| Object lastKey() | Returns the last key of the TreeMap. |

**Differences between HashMap and TreeMap:**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| No insertion order | Ascending order is maintained. |
| Can contain one null key | No Null key is allowed. |
| No restrictions for the keys in HashMap. | All the keys should be of type Comparable. |

**HASHTABLE CLASS:**

* Hashtable stores the data in the format of key and value pair.
* Hashtable class contains unique elements.
* Hashtable class cannot have null keys or null values.
* It internally uses hashtable datastructure for storing the data.
* It stores data as array of list and each list is called a bucket.
* It also uses the hashing technique for storing the unique elements and it is synchronized as well.
* It is introduced in 1.0 and we will go for Hashtable class if our frequent operation is search.
* **All the elements in Hashtable are stored based on the hashCode of the Keys.**

**Below are very important Constructors in Hashtable class:**

* **Hashtable t=new Hashtable();--- creates an empty Hashtable with default initial capacity 11 and load factor 0.75.**
* **Hashtable t=new Hashtable(int initialCapacity); --- creates an empty Hashtable with specified initial capacity.**
* **Hashtable t=new Hashtable(int initialCapacity, float loadFactor); --- creates the Hashtable with specified initialCapacity and loadFactor.**
* **Hashtable t=new Hashtable(Map m)—Creates a Hashtable with the specified Map.**
* Please refer HashtableClassDemo.java for more details on Hashtable class.
* public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>.
* Below are the few specific methods to the Hashtable class.

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| rehash() | Increases the size of the hashtable. |
| contains(Object value) | Returns true if the hashtable contains specified value. |

**Differences between HashMap and Hashtable:**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| HashMap is non-synchronized and hence it is fast | Hashtable is synchronized and hence it is slow. |
| It is intro in 1.2 | It is a legacy class |
| HashMap can have one null key multiple null values. | Hashtable cannot have null keys and null values. |
| HashMap inherits AbstractMap class. | Hashtable inherits Dictionary class. |
| HashMap can be converted to the synchronized by calling Map m=Collections.synchronizedMap(hashMap). | Hashtable is internally synchronized and cannot be converted to non-synchronized. |
| We can traverse HashMap with the Iterator. | We can traverse Hashtable with Iterator and Enumerator both. |
| Iterator in HashMap is fail-fast. | Enumerator in Hashtable is not fail-fast. |

**COLLECTIONS CLASS IN JAVA VERY IMPORTANT\*\*\*:**

* Collections class is present in the java.util.\* package.
* Collections class provides the functionalities that operate on the Collection Objects.
* Functionalities like searching, sorting and accessing random values in the collection are possible with the Collections class methods.
* We will get NullPointerException if the input to the Collections class methods is null. All the Collections class methods are static.
* Below is the Hierarchy of the Collections class.
* public class Collections extends Object.
* There are many predefined methods in the Collections class. Below are few of them.

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| static addAll(Collection c, T elements) | This method adds the all the specified elements to the specified collection c. |
| static void reverse(Collection c) | This method reverses all the elements in the specified list. |
| boolean replaceAll(Collection c, T oldValue, T newValue) | Replaces all the oldValues with the newValues in the specified list. |
| max(Collection c) | Returns the Maximum element in the specified collection. |
| min(Collection c) | Returns the Minimum element in the specified collection. |
| Queue<T> asLifoQueue(Deque d) | This method returns the view of LIFO Queue where last inserted element will be the one to be removed first. |
| int binarySearch(List e, K key) | This method searches the specified key in the specified List and returns the index of the specified key. Otherwise returns -1. Before we make a call to this method Collections.sort() has to be invoked otherwise we will get erroneous results. It uses binary search Algorithm. |
| void sort(List l) | This method sorts the specified list in the ascending order. |
| Collection checkedCollection(Collection c, Class type) | This method returns the dynamic typesafe view of the specified collection. Here TypeSafe view is nothing but we are not allowed to perform any illegal operations until it is permitted. |
|  |  |

* **Pending for Notes. Collections class methods like copy, disjoint, list, synchronizedCollection, enumeration, sort, fill.**
* **Comparable interface -> compareTo()**
* **Comparator interface ->compare() and equals().**

**Differences between Comparable and Comparator interfaces.**

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| This is meant for default natural sorting order. | This is meant for customized sorting order. |
| It is present in java.lang package. | This is present in java.util pacakage. |
| This class provides only one method. compareTo() | This class provides two methods. compare() and equals(). |
| All the wrapper classes and String class implements Comparable. | Only one class implements Comparator interface i.e. Collator abstract class. |
| Comparable object compares the objects of its own type. The comparable object has to implement java.lang.Comparable. Hence we can achieve single sorting sequence. | Comapartor object compares the objects of any type. The comparator object has to implement java.util.Comparator. Hence we can achieve multi sorting sequence. |
| In Comparable interface we can sort the collection elements based on one of the parameter i.e. salary or empid or DOJ. Hence we can achieve only single sorting sequence. | In Comparator interface we can sort the collection elements based on all parameters like salary, empid and DOJ. Hence we can achieve multiple sorting sequence. |

**Differences between ArrayList and Vector:**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList is non-synchronized and hence fast. | Vector is synchronized and hence slow. |
| It increases the size of the array by 50% when the capacity of the current Array exceeds. | It increases the size of the array by 100% when the capacity of the current Array exceeds. |
| It is introduced in 1.2 | It is a Legacy class. |
| We can traverse the elements of the ArrayList with Iterator and ListIterator and for-each loop. | Along with ListIterator, for-each loop and Iterator we can traverse Vector elements with Enumeration also. |
| Iterators in point 4 are fail-fast and we will get ConcurrentModificationException if we try to modify the data once iterator is returned. | Here Enumeration is not fail-fast. |

**Vector Class:**

* Vector class is a legacy class present in the java.util package.
* It maintains insertion order.
* We can have duplicates in Vector and null values are allowed.
* Vector also internally uses Dynamic Array for storing the elements.
* Vector’s Dynamic array grows or shrinks based upon whether we add elements or remove elements from the Vector.
* If we don’t need thread-safe implementations then we can use ArrayList for storing and retrieving elements.
* In Vector iterators returned by iterator() is fail-fast i.e. when we make changes to vector once iterator is returned then we will get ConcurrentModificationException, but it cannot be guaranteed.
* Vector uses Enumeration which is not fail-fast by calling elements().
* Since vector uses internally array we can access the elements with the index.
* Below is the hierarchy of the Vector class:

public class Vector<E> extends [AbstractList](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html)<E>

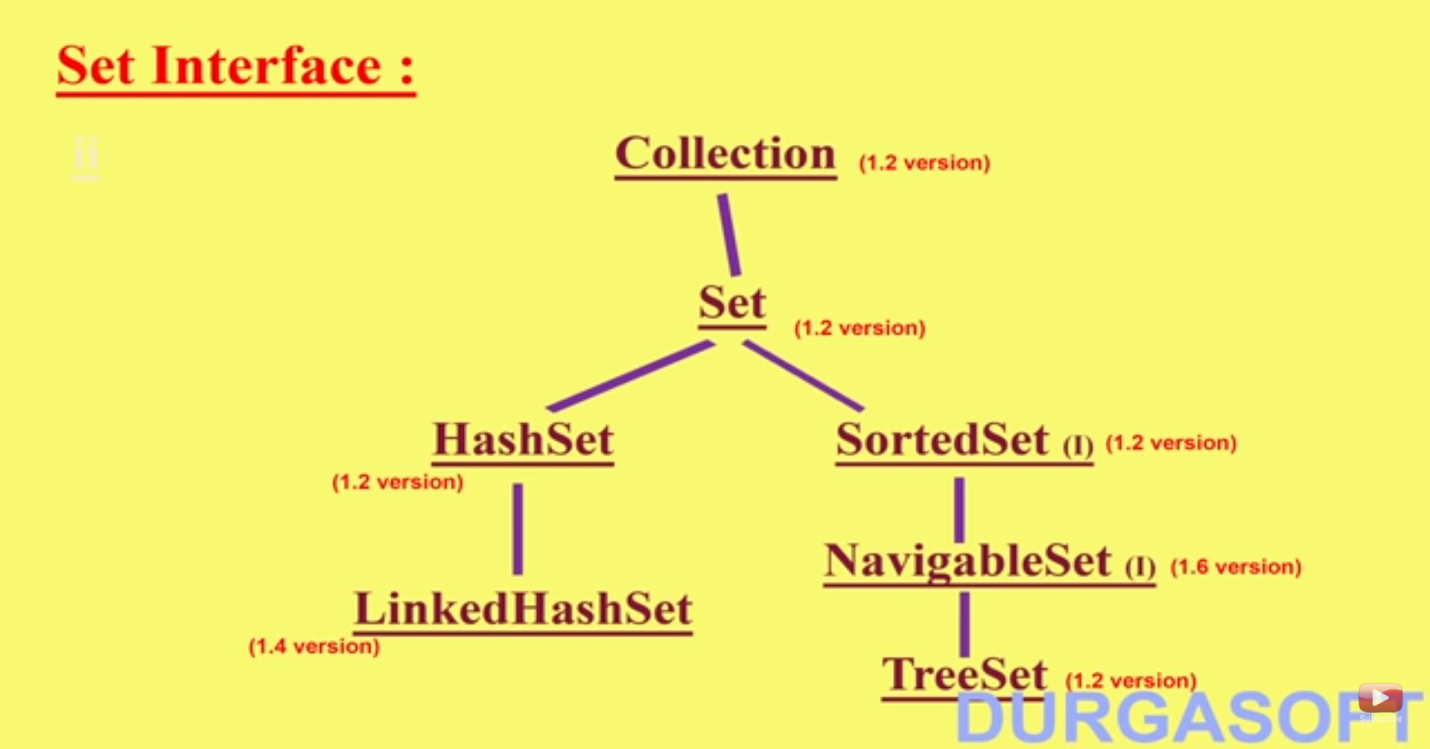
implements [List](https://docs.oracle.com/javase/7/docs/api/java/util/List.html)<E>, [RandomAccess](https://docs.oracle.com/javase/7/docs/api/java/util/RandomAccess.html), [Cloneable](https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

**STACK CLASS:**

* Stack class is also used for storing the Collection Objects.
* Stack class can contain duplicates, maintains insertion order and can have null elements.
* Stack class inherits Vector class.
* Stack class elements also can be traversed by Iterator, ListIterator and Enumeration interface.
* Stack stores the values in the format of the LIFO order and it has push, pop, search, empty, peek important methods.

**SET INTERFACE HISTORY BELOW AND HASH SET INTERNAL IMPLEMENTATION:**

* **HashSet internally uses HashTable data structure for storage.**
* HashSet does not allow duplicates, it will just return false for add().
* HashSet insertion is based on the hashCode of the objects.
* HashSet is best for searching the elements as the storage with hashCode is efficient.



* HashSet set=new HashSet();---- A new empty HashSet will be created with the initial capacity 16 and Load Factor or Fill Ratio- 0.75.
* Here Fill Ratio or load factor is like if the capacity gets filled by 75% then a new object will be created in HashSet.
* **HashSet internally uses HashMap.**

**Very Important Constructors in HashSet that are commonly used for all Hashing Data Structures.**

* HashSet hashSet=new HashSet(); creates a new HashSet with defalut initial capacity of 16 and load factor of 0.75.
* HashSet set2=new HashSet(int initialCapacity); customizes the initial capacity of the HashSet.
* HashSet set3=new HashSet(int initialCapacity, float loadFactor); customizes the capacity and load Factor of the HashSet.
* HashSet set4=new HashSet(Collection c);-- Takes collection as input.

**Internal Implementation of HashSet:**

* HashSet stores elements based on the hashing technique.
* It won't store the elements based on key/value pairs.
* As HashSet internally uses HashMap for storing the elements, it cannot have the duplicates in it as HashMap allows unique keys and only one null key.
* When we call set.add(“Ravi”); it internally calls map.put(setvalue, PRESENT); i.e. the value that has to be added in HashSet is passed as key to HashMap and PRESENT one dummy object is passed as value to the map.
* Below is the internal implementation of add() in HashSet class.
  + boolean add(Element e){

return map.put(e,PRESENT)==null;

}

* PRESENT is dummy object defined as private static final Object PRESENT=new Object();
* add() returns true if the key is not present already in the HashMap if it contains will return false.
* Since HashMap allows unique keys, hence HashSet contains unique elements.

**How HashSet remove() works:**

* Internally hashSet.remove() is as below

Public boolean remove(Object o){

return map.remove(o)==PRESENT;

}

* We know that map.remove(key) will return the value associated with that key, since for all the keys in HashMap the value is PRESENT only.
* Hence remove() return true or false in HashSet.

**How to retrieve values from HashSet:**

* We don’t have any get() in HashSet for retrieving the elements. Hence HashSet uses iterator for fetching the values of HashSet.
* Internally iterator() of HashSet is as below:

Public Iterator<E> iterator(){

return map.keySet().iterator();

}

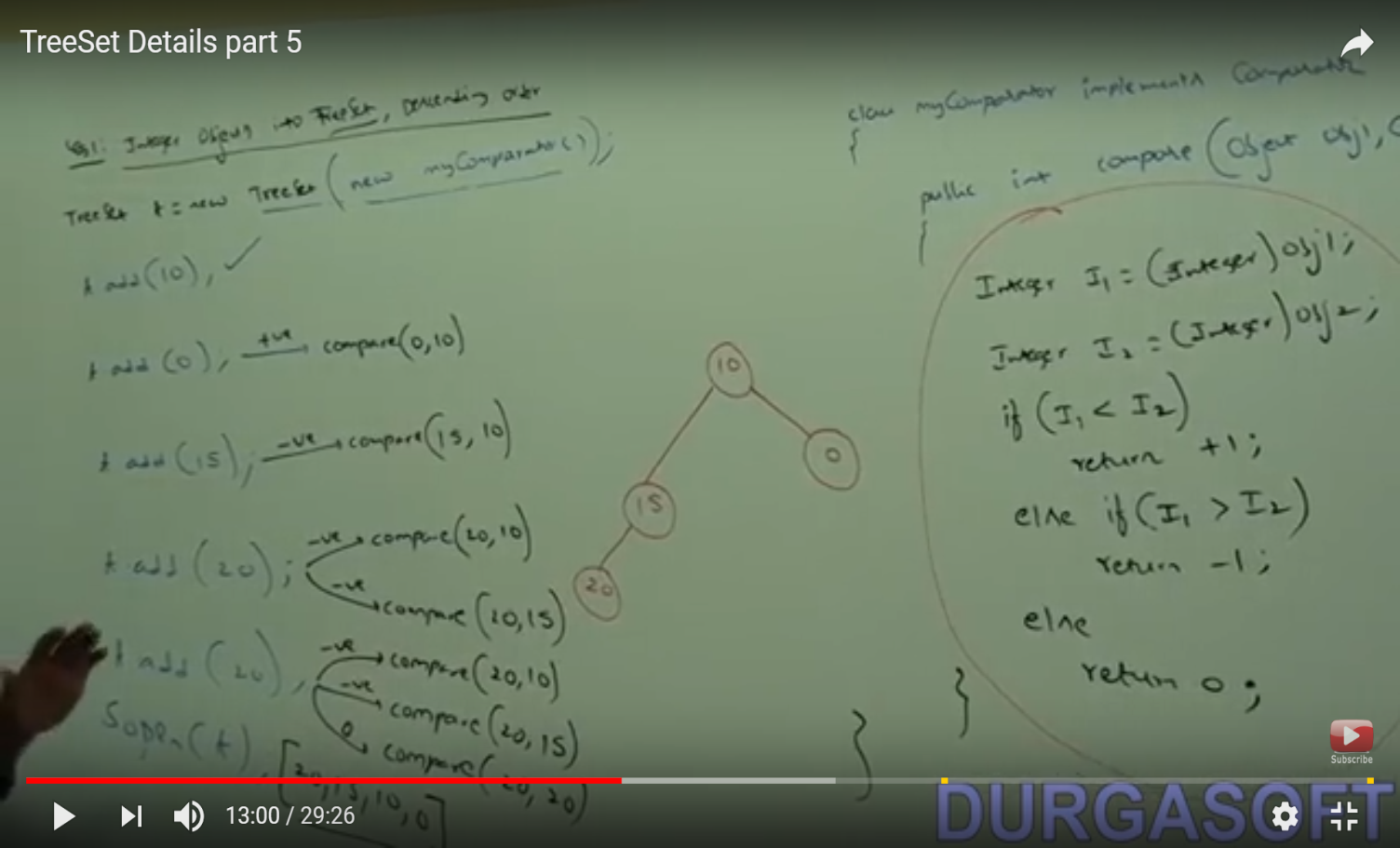
* Iterator will return all the keys in the HashMap in Set view as values of set are keys in HashMap.

**Comparable Interface** \*\*\*\*\*\*\*\*\*:

* This interface is present in java.lang package.
* This interface has only one method compareTo() and below is the behavior of the compareTo(). Below is declaration and behavior of .
  + public int compareTo(Object obj);
  + obj1.compareTo(obj2) ---If obj1>obj2 then returns positive.
  + obj1.compareTo(obj2) ---If obj1<obj2 then returns negative.
  + obj1.compareTo(obj2) ---If obj1==obj2 returns 0.
* Comparable is meant for default natural sorting order.
* **Please refer TreeSetWithComparable.java.**

**Comparator Interface \*\*\*\*\*\*\*\*\*:**

* If we are not satisfied with the default natural sorting or if some objects does not implement Comparable interface then we will go for the Comparator available in java.util package.
* Comparator is meant for customized sorting order.
* Comparator interface has two methods public int compare(Object obj1, Object obj2); public boolean equals();
* Whenever we implement Comparator interface we should provide implementation for the compare() and for equals() is optional.
* Providing implementation for equals() is optional because, our class by default extends Object class and provides implementation for the equals() and by default all parent class methods are available to child class.
* compare() is also having same behavior as the compareTo() except it takes two objects as input.
* **Please refer TreeSetWithComparator.java.**
* The First element inserted into the TreeSet will be treated as root and based upon the values that is inserted into TreeSet new elements are assigned left or right of the root.
* *\*\*\*\*\*\*\*How Comparison happens in compare(Object o1, Object o2) of Comparator interface. For General behavior if o1>o2 return positive, if o1<o2 return negative, o1==o2 return 0 this is applicable good for ascending order. If we need some custom descending order for TreeSet then the value we return will be changed in compare() as o1>o2 return negative, o1<o2 then return positive, o1==o2 return 0.*
* Below diagram represents sorting will happen in descending order for TreeSet where we use balanced tree.

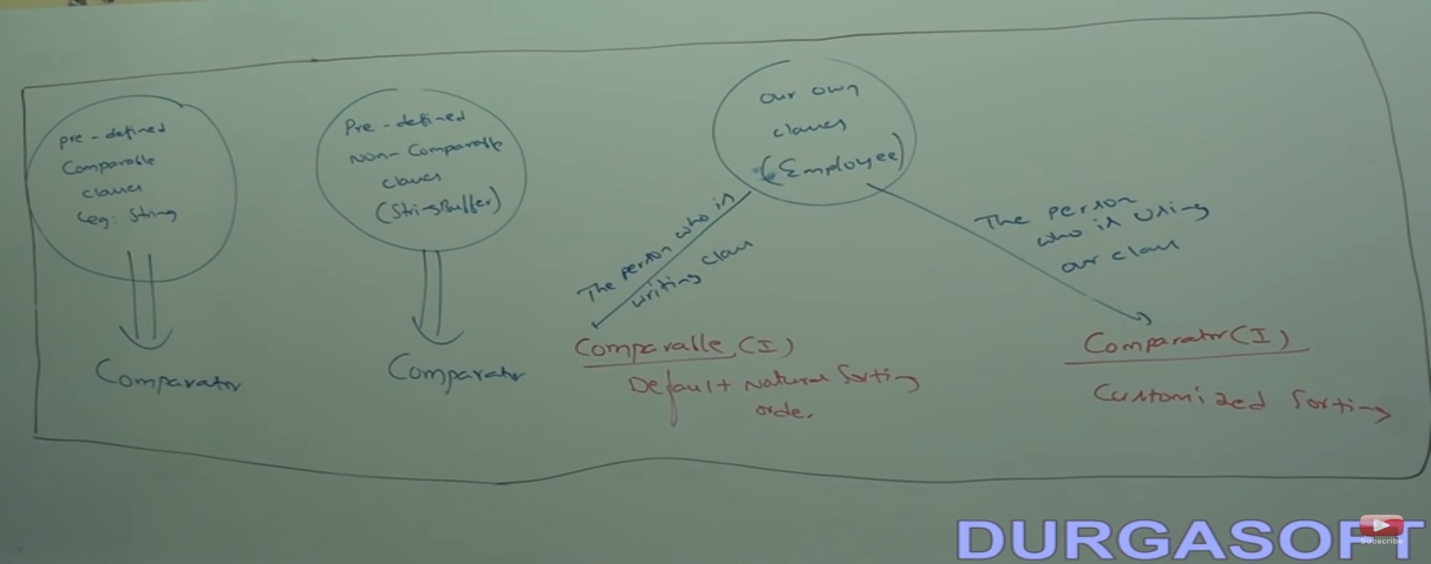




* ***If we define our own sorting by using comparator, then the objects inside the TreeSet need not be of type Comparable.***
* We can insert heterogeneous elements also into the TreeSet by specifying the common interface implemented by the class objects that we are inserting i.e. if we depend on Default Natural Sorting order then the objects should be of type Comparable and Homogeneous. If we are using our customized sorting order then we don’t need to have homogenous objects to be inserted in TreeSet.

**When we should go for Comparator and when we should go for Comparable\*\*\*\*\*\*\*\*\*\*\*\*:**

* All the predefined Comparable classes like String, Wrapper Classes( Integer, Long etc..) have already default natural sorting order, if we are not satisfied with the default natural sorting order then we should go for the customized sorting order by implementing Comparator interface.
* All the predefined Non-Comparable classes like StringBuffer, StringBuilder will not have default natural sorting order, hence we have to provide custom sorting order by implementing Comparator interface and implementing compare().
* All the customized or user defined classes have two scenarios :
  + Person who is writing should provide the support for Default Natural Sorting order by implementing Comparable interface. For ex: Employee objects in majority of cases is ordered by empid.
  + Person who is using the above class if they are satisfied with the Default Natural Sorting order then can implement Comparator interface and provide implementation for compare().



* **The above diagram show diagrammatically when to use Comparator and when to use Comparable.**
* **The core difference between Comparable and Comparator is:** 
  + **Suppose If I want some class objects to be sorted which I got from some library and does not implement Comparable interface then we can go ahead and implement the Anonymous inner class implements Comparable and Comparator interfaces.**
  + **If we are not satisfied with the sorting which is already implemented then we can ahead and implement the Comparator interface and we can do customized sorting based on the any property.**

**Difference between List, Queue and Set:**

|  |  |  |
| --- | --- | --- |
| **List** | **Queue** | **Set** |
| In List we can store the elements in a particular order, and we can insert or remove or retrieve any element at any point in collection. We can have duplicates also here. | In Queue also we can store the elements in ordered format. Here we can insert only at the end of the queue i.e. tail of queue and can retrieve and remove the elements only from the front i.e. head of the queue. | In Set we cannot have duplicates and no order is followed here. |

**HashTable data structure internal implementation:**

* **Hash Table is a data structure that is mainly used for very fast retrieval of the data irrespective of the amount of data.**
* **The application of the Hash Table is in the DataBase indexing, compilers, password authentication, caching etc.**
* **Suppose we have a one dimensional array of Strings as below with size 10 and filled with values.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ram** | **Sam** | **Tim** | **Ant** | **Bat** | **Cat** | **Hat** | **Rat** | **Pot** | **Toy** |

* For Accessing Cat if we know the index then it will be very faster to access the element. If the array size is very big and if you don’t know the index of an element then we need to do linear search for the element in the array which will take more time than accessing directly the element.
* With Hashing technique below, we can store the elements in the array in a different format as below.

**Lets take first element Ram—ASCII of R-82, a-97, m-109 ---288%size of array ==288%10===8==Put Ram in 8th Position.**

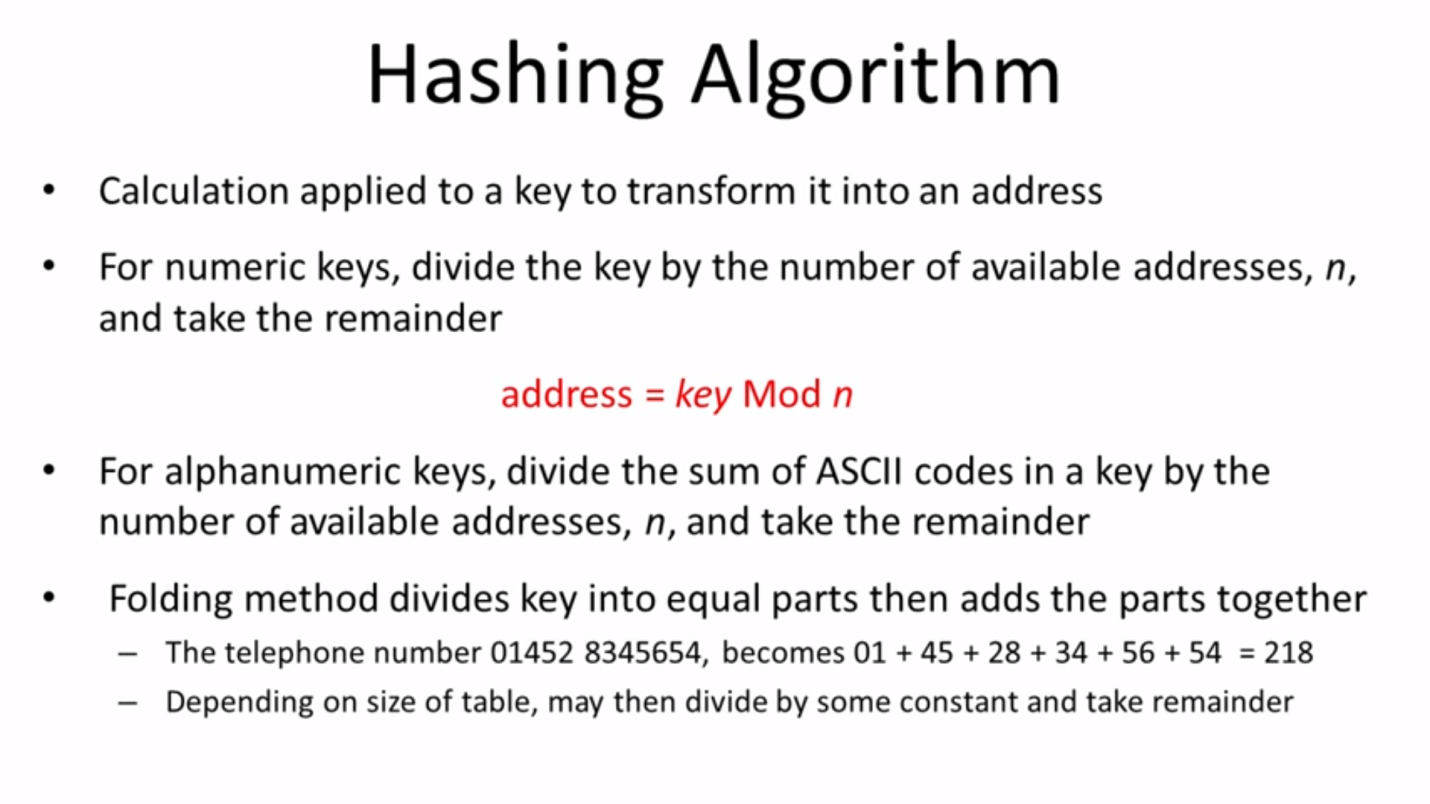
**Sam—289%10**

**Tam—84+97+109**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tam** |  |  |  |  |  |  |  | **Ram** | **Sam** |

**0 1 2 3 4 5 6 7 8 9**

**The same hashing approach applies for all the elements in the array insertion— and during retrieval of Tam—84+97+109 --- 0 Tam is at 0 index hence we will fetch the value directly from the array with indexing hence retrieval is very faster.**

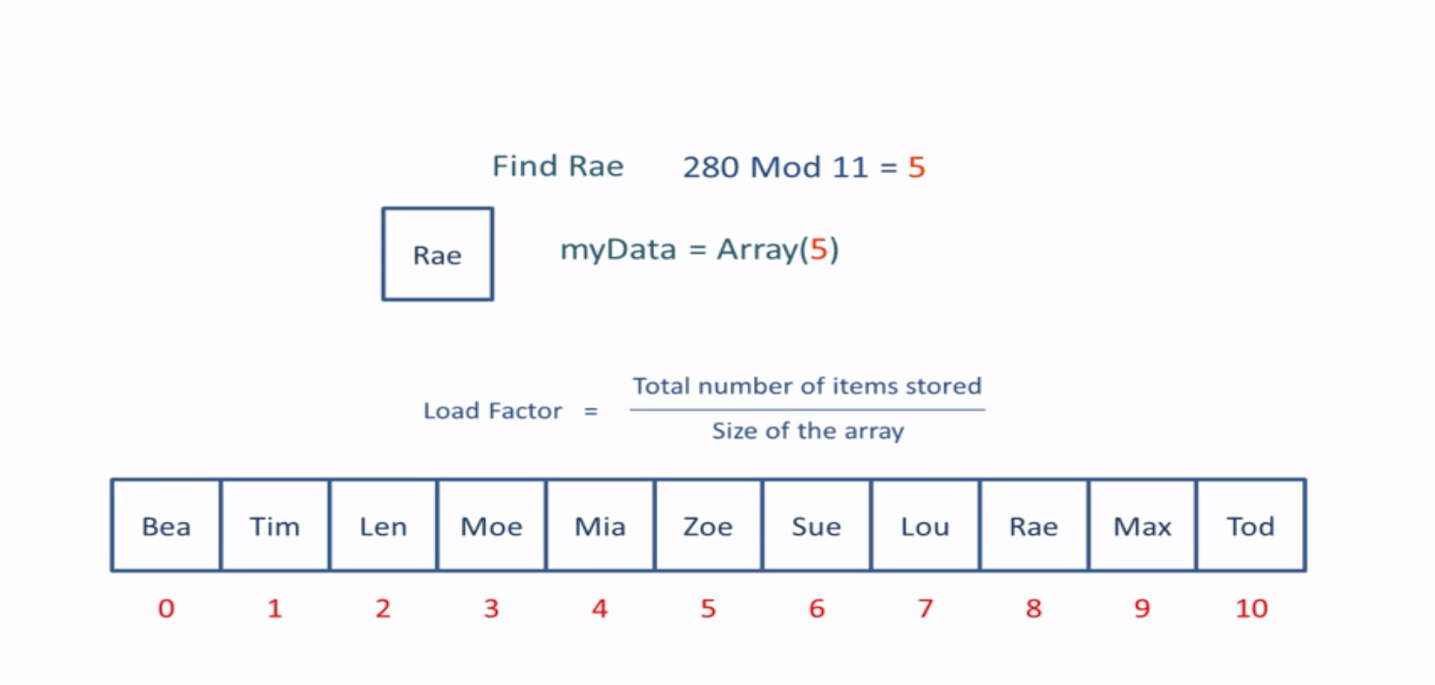


* Hashing algorithm is also known as hash function. There are various types of Hashing algorithms depending upon type of data we will choose the one.
* Hashing is applied to an individual element which will be divided by the total number of addresses in the memory which will give the index where the element has to be stored. The index is called address. This can be applied for both numeric and alphanumeric as directly dividing the numeric%total addresses or ASCII sum%totaladdresses.

**COLLISION:**

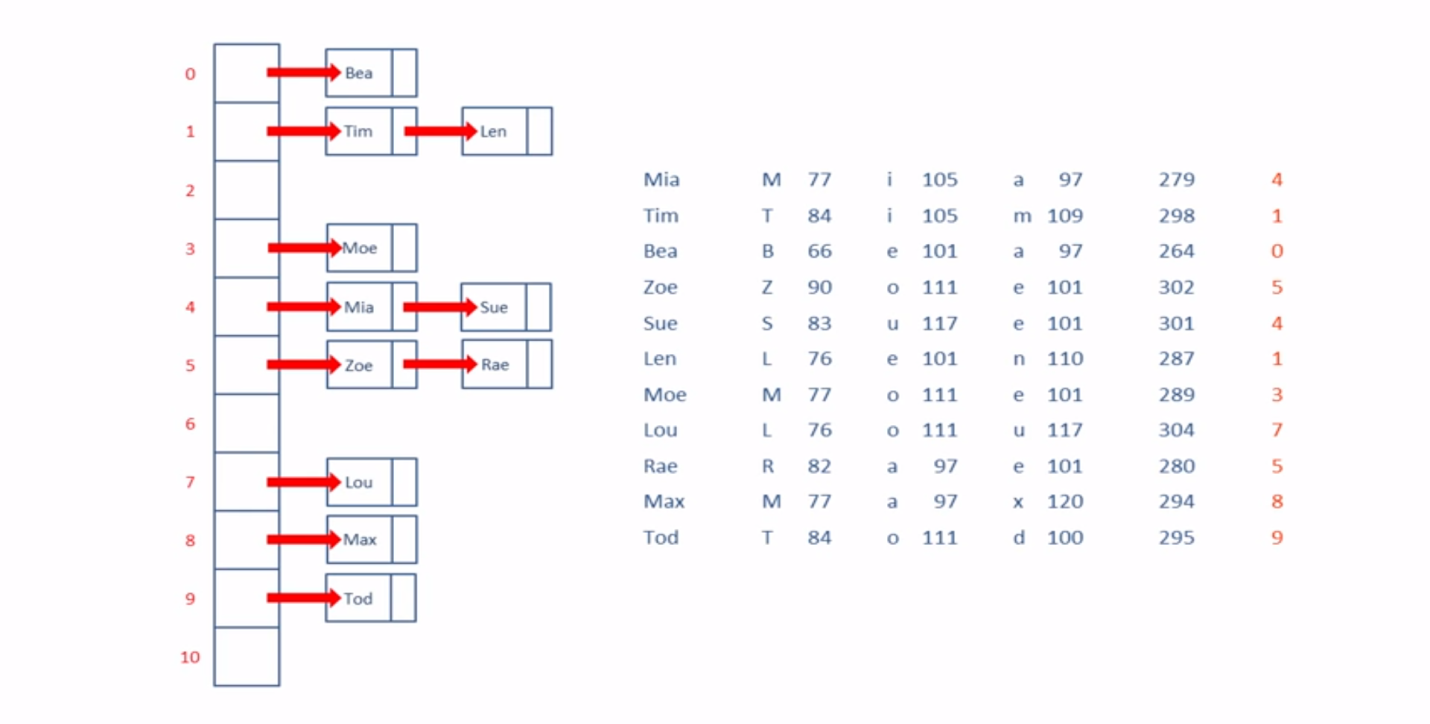
* Whenever we are inserting the elements into a Hash Table there is more likely that hash function may return same index for more than one key. This is called collision.
* To avoid collision, we have a technique called open addressing or linear probing. Suppose in the below scenario for the elements Mia and Sue have the same index returned by the hash function. In this scenario when inserting Sue, as the element is already there in index 4 then we will be searching next adjacent empty addresses for storing if not found till the end of the Hash Table then it will probe from the start of the table till it finds the address. In the below scenario we have 6th position empty and hence Sue is inserted in the same address, the same is followed during the retrieval also.
* One better approach for avoiding Collision is having the more addresses than the number of elements and setting the load factor to 70% after which the elements reaches the Hash Table should dynamically increase its size.

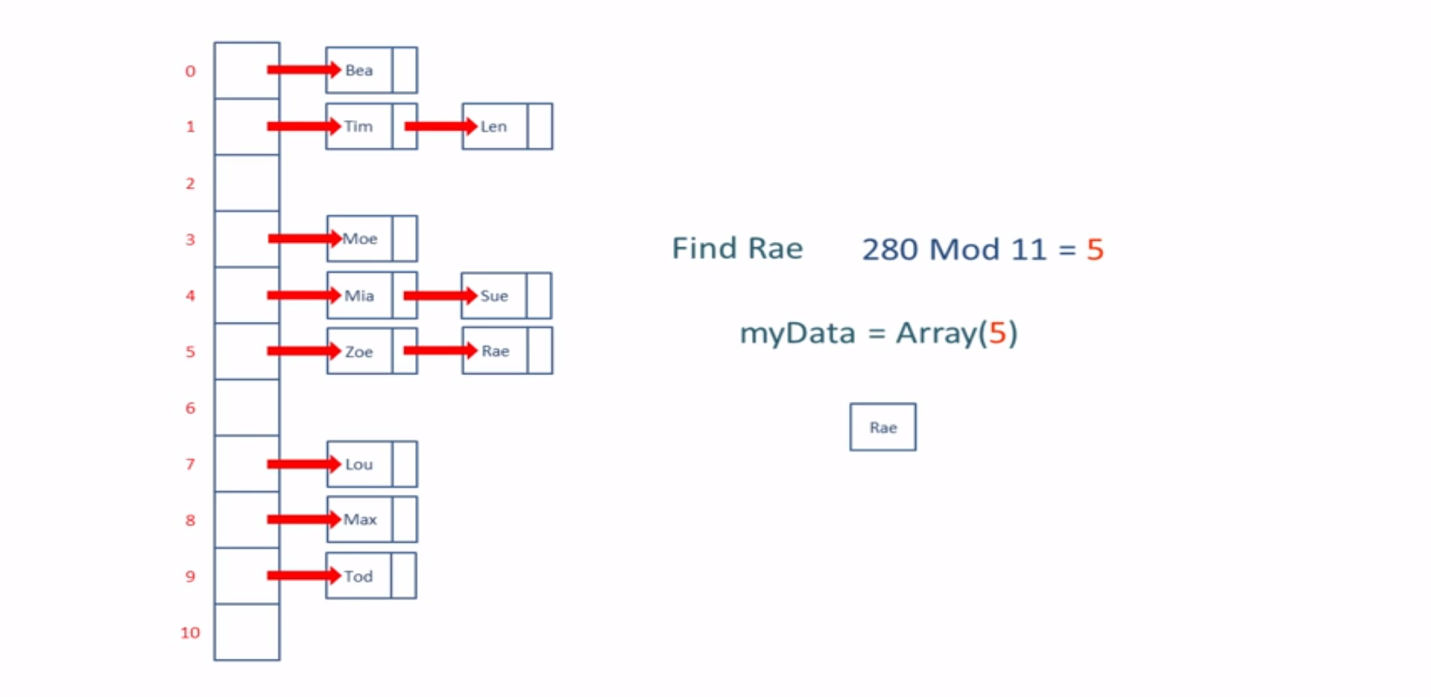




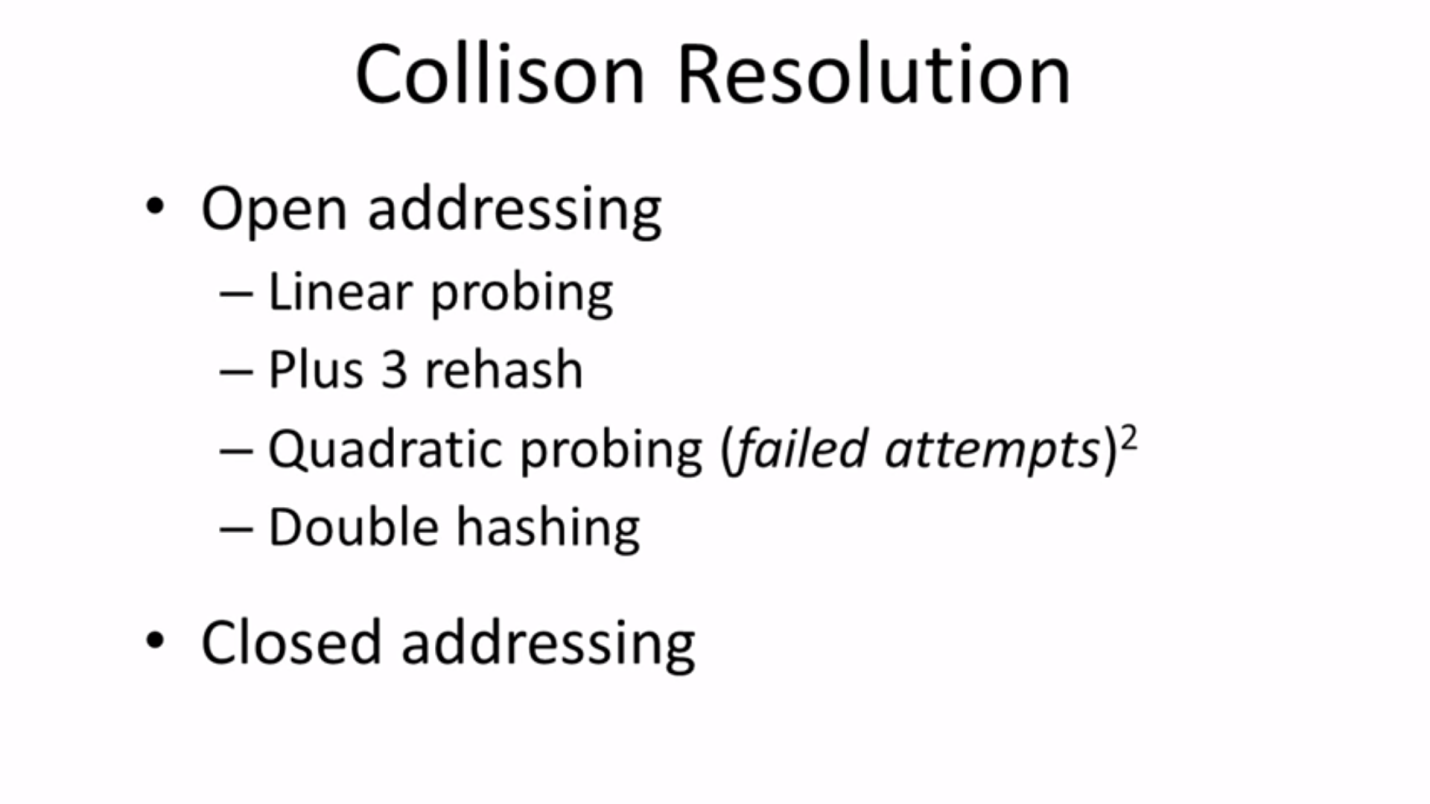
**2nd Approach for Collision Resolution: (Closed Addressing or Chaining):**

* **In this approach we will be using one data structure that will be pointed from the particular address. For example in the below we have used LinkedList. Sue is collisional and added a node to the linkedlist and Mia pointed to that Node. The same will be applied for all collisions. During retrieval the Linked List is traversed for fetching the elements.**

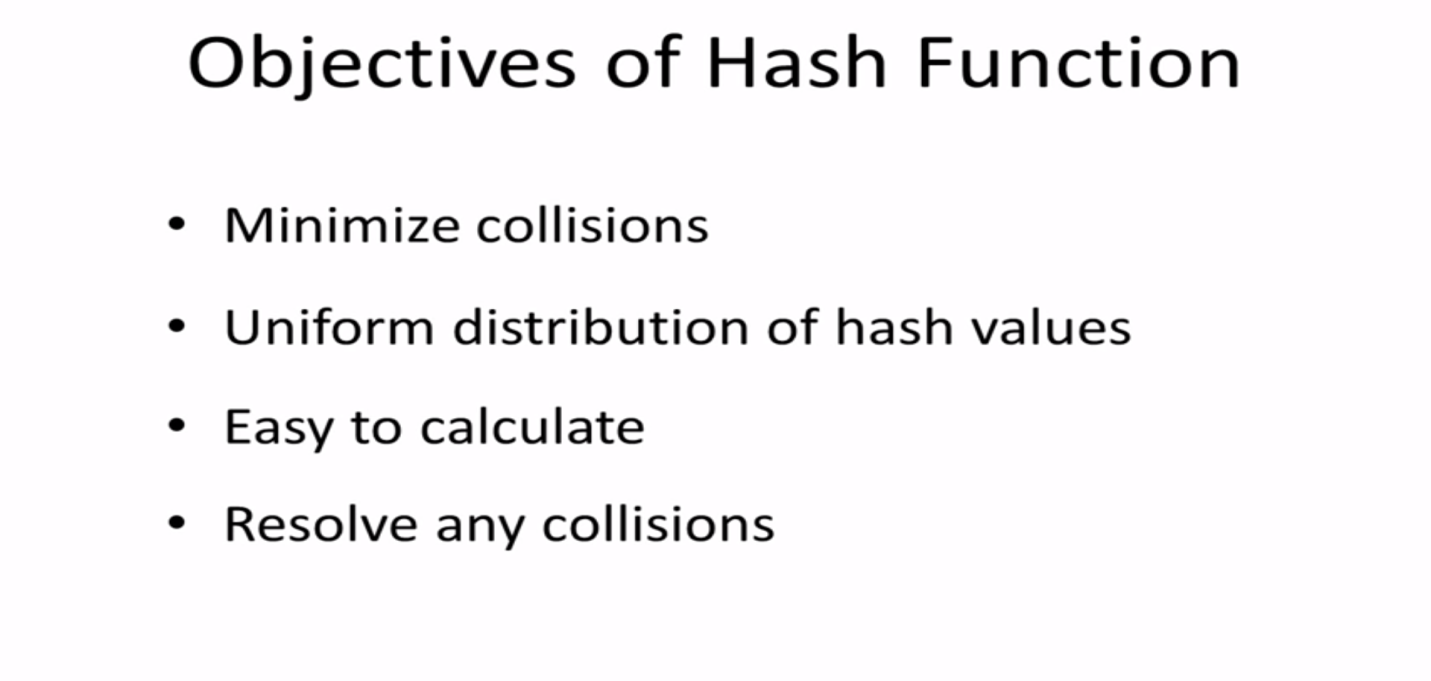


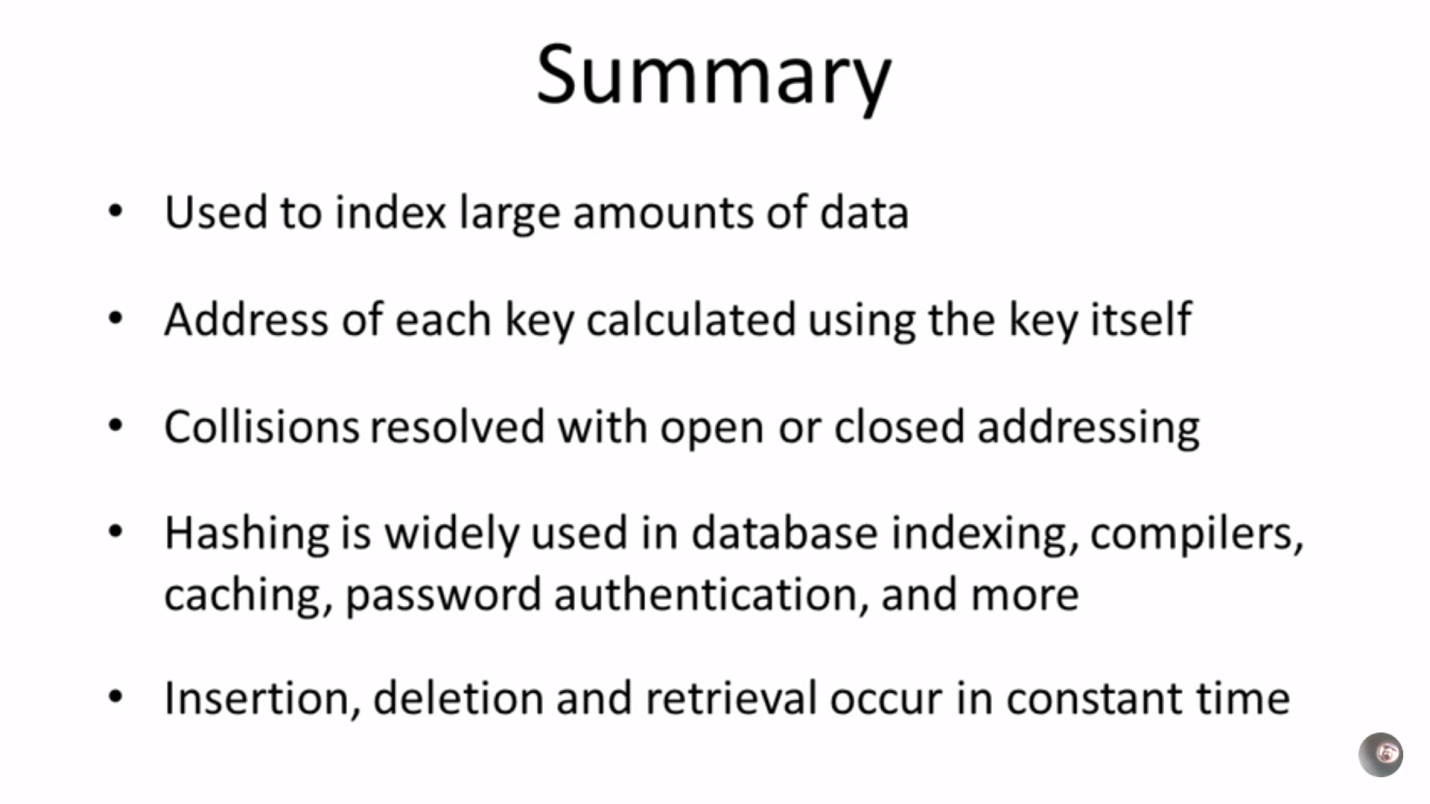


**Collision Resolution is possible with all the below ways.**



* **When a Hash function is created then it should meet all the below objectives to be efficient.**





**Properties class in java.util package:**

* Properties class is introduced in JDK 1.0.
* Properties class object holds the data in the key, value pair.
* Properties class objects holds the data which is fetched from the property file in an application in the Key, Value pair.
* In Properties class both key and value should be of type String.
* Properties class creates an initial capacity of 11 size bucket by default.
* For Properties class no null keys or no null values are allowed. If we try to add then we will get NullPointerException.

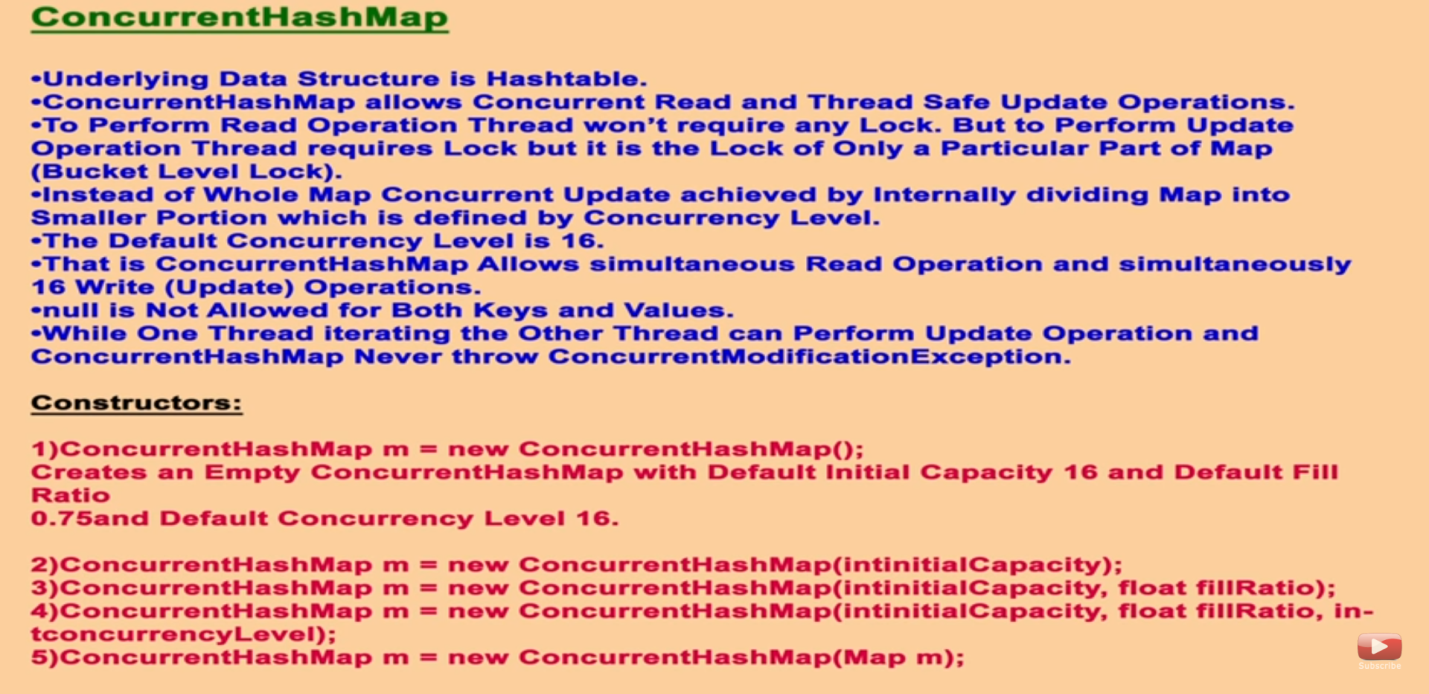
**Constructors in Properties class:**

* **Properties p=new Properties();**

**Methods in Properties class:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| String getProperty(String propertyName) | Returns the value associated with the specified property name. If specified property not available this method returns null. |
| String setProperty(String propertyName, String propertyValue) | Adds the entry into the Properties class. |
| Enumeration propertyNames() | Returns all the property names in Properties object in the Enumeration. |
| void load(InputStream is) | To load all the properties from the property file into Properties class object. |
| void store(OutputStream os, String comments) | To write all the properties in the Properties class object to the property file. |

**CONCURRENTHASHMAP:**



* *ConcurrentHashMap implements ConcurrentMap which extends Map interface.*
* ConcurrentHashMap have lot of advantages of when compared to Hashtable and HashMap.
* When we compare ConcurrentHashMap with HashMap, With HashMap we cannot achieve thread-safe operations but with ConcurrentHashMap we can achieve thread-safe operations along with performance.
* When we compare ConcurrentHashMap with Hashtable, Hashtable is thread-safe and once a thread acquires the object lock all other threads has to wait till the lock is released and hence performance will be very poor.
* Suppose if we need thread-safe operations along with the performance then we can go with the ConcurrentHashMap which allows concurrent read operations and 16 simultaneous write or update operations on a map.
* Default concurrency level of ConcurrentHashMap is 16 but it is not required to be same all the times.
* In ConcurrentHashMap at a time any number of concurrent read operations can be made, instead of locking the entire object in concurrentHashMap bucket level locking is provided i.e. if concurrency level is 16 and initial capacity is 16 then 16 buckets are available and 16 locks for each bucket is available. Since Default initial capacity of the ConcurrentHashMap is 16 its concurrency level is also by default 16.
* In ConcurrentHashMap null keys or null values not allowed.
* ConcurrentHashMap internally uses Hashtable datastructure for storing the elements.
* For read operation in ConcurrentHashMap lock is not required, for write or update operation lock is mandatory i.e. not the entire map object instead bucket lock. Here number of buckets will be decided by the concurrency level.
* If one thread is iterating a map, if other thread updates the map then ConcurrentHashMap never throws ConcurrentModificationException. Not only ConcurrentHashMap all the ConcurrentCollections never throw ConcurrentModificationException.
* We have five important constructors in ConcurrentHashMap which are specified in the above diagram.

**Below are three specific methods w.r.t ConcurrentMap:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| Object putIfAbsent(Object k, Object v) | This method adds the entry into the ConcurrentHashMap object if the specified key is not already present in the Map. Otherwise it will return the value associated with key and don’t add the entry. |
| boolean remove(Object k, Object v) | This method returns true if the specified key and value is present in map and removed. |
| boolean replace(Object k, Object v, Object newValue) | This method replaces the value for the specified key with new value if present and returns true if the replacement is successful otherwise false. |

**INTERNAL IMPLEMENTATION OF HASHMAP \*\*\*\*\*\*\*\*\*\*\*:**

* When we create a HashMap as below:

HashMap m=new HashMap(); Then below are the internal attributes and its values.

1. table-- null -> In table attribute inside HashMap all the entries will be stored.
2. size—0 ->As we did not yet insert any entries in HashMap.
3. loadFactor—0.75 -> Once the HashMap reaches 75% of its size then table will be doubles i.e. from 16 to 32 it will be increased.
4. Threshold—12 i.e. totalcapacity\*loadFactor=threshold. In our case it will be 12.
5. As HashMap default capacity is 16, 16 buckets will be created when we insert the first entry into HashMap. And each bucket internally is a LinkedList.
6. Each LinkedList or bucket will store the following values key, value, hashCode, next. Key is the key in the entry of HashMap, value is value in HashMap, hashCode is generated with the key itself. Based on the hashCode only a bucket will be assigned to the each entry in map. Next is the address of the next LinkedList if there is any collision occurred because of same hashCode generated.
7. If a map has null key then it will be always stored in the 0th bucket only irrespective of HashMap Capacity.
8. \*\*\*\*If we are trying to insert Duplicate Keys then internally HashMap does the content comparison of the keys with equals() and oldValue will be replaced by newValue and oldValue will be returned and duplicate keys will not be inserted inside the HashMap. This is how HashMap doesn’t allow duplicate keys and allows duplicate values.
9. When we are retrieving the values from the HashMap we will use get(Key k). For retrieval hashCode will be generated for key k and index will retrieved for the key. In the index that we got we will check the hashCode and key associated with it, if it matches then we will be retrieving the value associated with the specified key.