Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T

ArrayList

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Method	**Description**
void **add**(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean **add**(E e)	It is used to append the specified element at the end of a list.
boolean **addAll**(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean **addAll**(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void **clear**()	It is used to remove all of the elements from this list.
E **get**(int index)	It is used to fetch the element from the particular position of the list.
boolean **isEmpty**()	It returns true if the list is empty, otherwise false.
Object[] **toArray()**	It is used to return an array containing all of the elements in this list in the correct order.
boolean **contains**(Object o)	It returns true if the list contains the specified element
int **indexOf**(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E **remove**(int index)	It is used to remove the element present at the specified position in the list.
boolean **remove**(Object o)	It is used to remove the first occurrence of the specified element.
E **set**(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void **sort**(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
int **size**()	It is used to return the number of elements present in the list.

Ways to iterate the elements of the collection in Java

There are various ways to traverse the collection elements:

By Iterator interface.

```
Iterator itr=list.iterator();
while(itr.hasNext()) {
  System.out.println(itr.next()); }
```

By for-each loop.

```
for(String fruit:list)
     System.out.println(fruit)
```

By ListIterator interface.

```
ListIterator<String> list1=list.listIterator(list.size());
              while(list1.hasPrevious())
              {
                  String str=list1.previous();
                  System.out.println(str);
              }
```

By for loop. **//**using list.size();

By forEach() method. **//** using lambda function

```
list.forEach(a -> {
                System.out.println(a);       });
```

By forEachRemaining() method **//** using lambda and Iterator

```
Iterator<String> itr=list.iterator();
 itr.forEachRemaining(a ->   {
           System.out.println(a);   });
```

Some methods and examples of Arraylist called 'al'

ArrayList<String> **al**=**new** ArrayList<String>();

If **al2** is another array List

```
//Adding new elements to arraylist
                    al.addAll(al2);
//Removing all the new elements from arraylist
                    al.removeAll(al2);
//Removing elements on the basis of specified condition  using "Lambda Expression"
                             al.removeIf(str -> str.contains("Ajay"));
```

```
//Sorting the list
          Collections.sort(al);
```

Collections is the utility class

Java LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure

The important points about Java LinkedList are:

- Java LinkedList class **can contain duplicate** elements.
- Java LinkedList class **maintains insertion order**.
- Java LinkedList class is **non synchronized**.
- In Java LinkedList class**, manipulation is fast** because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

add(), **add**(int x), **addAll**(Collection<> c), **addAll**(int x, Collection<> c), **contains**(Object o)

void **addFirst**(E e)	It is used to insert the given element at the beginning of a lis
void **addLast**(E e)	It is used to append the given element to the end of a list.
void **clear**()	It is used to remove all the elements from a list.
E **get**(int index)	It is used to return the element at the specified position in a list.
E **getFirst**()	It is used to return the first element in a list.
E **getLast**()	It is used to return the last element in a list.
int **indexOf**(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.

int **lastIndexOf**(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
E **peek**()	It retrieves the first element of a list
E **poll**()	It retrieves **and removes** the first element of a list.
E **pop**()	It pops an element from the stack represented by a list.
void **push**(E e)	It pushes an element onto the stack represented by a list.
E **remove**()	It is used to retrieve and removes the first element of a list.
E **remove**(int index)	It is used to remove the element at the specified position in a list.
E **removeFirst**()	It removes and returns the first element from a list.
boolean **removeFirstOccurrence**(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
boolean **removeLastOccurrence**(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
Iterator<E> **descendingIterator**()	It is used to return an iterator over the elements in a deque in reverse sequential order.

```
LinkedList<String> ll=new LinkedList<String>();
           ll.add("Ravi");
           ll.add("Vijay");
           ll.add("Ajay");

           Iterator i=ll.descendingIterator();
           while(i.hasNext())
           {
               System.out.println(i.next());
           }
```

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Convert an Array to List:

asList() method of **Arrays** class

Eg:-

```
String a[] = new String[] { "A", "B", "C", "D" };

List<String> list = Arrays.asList(a);

// to print an array also we can use Arrays class

System.out.println(Arrays.toString(a));
```

Convert a List to Array:

```
List<String> list = new ArrayList<>();    //add some values to it

String[] array = list.toArray(new String[list.size()]);
```

HashSet

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**.

Modifier & Type	Method	Description
boolean	**add**(E e)	It is used to **add** the specified element to this set if it is not already present.
void	**clear**()	It is used to **remove all** of the elements from the set.
boolean	**contains**(Object o)	It is used to return true if this set contains the specified element.
boolean	**isEmpty**()	It is used to return true if this set contains no elements.
Iterator<E>	**iterator**()	It is used to return an iterator over the elements in this set.
boolean	**remove**(Object o)	It is used to **remove the specified element** from this set if it is present.
int	**size**()	It is used to return the number of elements in the set.
Spliterator<E>	**spliterator**()	It is used to create a late-binding and fail-fast **Spliterator** over the elements in the set.

```
ArrayList<String> list=new ArrayList<String>();
        list.add("Ravi");          list.add("Vijay");          list.add("Ajay");
HashSet<String> set=new HashSet(list);          set.add("Gaurav");
        Iterator<String> i=set.iterator();
        while(i.hasNext())
        {          System.out.println(i.next());          }
```

HashSet Implementation

- We are achieving uniqueness in Set internally through HashMap
- When we create an object of HashSet, it will create an object of HashMap. We know that each key is unique in the HashMap
- When we add an element in HashSet like hs.add("India"), Java does internally is that it will put that element E here "India" as a key into the HashMap (generated during HashSet object creation). It will also put some dummy value that is Object's object is passed as a value to the key
- The important points about put(key, value) method is that
  If the Key is unique and added to the map, then it will return null
  If the Key is duplicate, then it will return the old value of the key
- When we invoke add() method in HashSet, Java internally checks the return value of map.put(key, value) method with the null value
- If the method map.put(key, value) returns null, then the method map.put(e, PRESENT)==null will return true internally, and the element added to the HashSet
- If the method map.put(key, value) returns the old value of the key, then the method map.put(e, PRESENT)==null will return false internally, and the element will not add to the HashSet

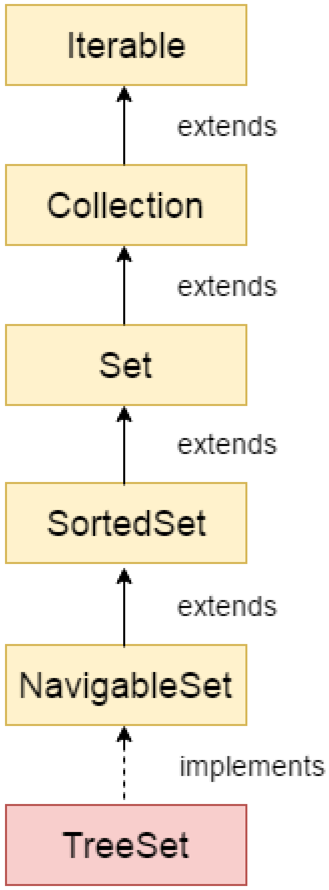
LinkedHashSet class

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

It implements all the Set Interface methods as mentioned above in HashSet

Java TreeSet class



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 quiet fast**.
- Java TreeSet class **doesn't allow null element**.
- Java TreeSet class is **non synchronized**.
- Java TreeSet class **maintains ascending order**.

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	DEFINITION
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 – *throws* ***null pointer exception*** *if compared with a null value*
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.
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.
boolean **contains**(Object o)	It returns true if this set contains the specified element.
boolean **isEmpty**()	It returns true if this set contains no elements.
boolean **remove**(Object o)	It is used to remove the specified element from this set if it is present.
void **clear**()	It is used to remove all of the elements from this set.
Spliterator **spliterator**()	It is used to create a late-binding and fail-fast spliterator over the elements.
Iterator **iterator**()	It is used to iterate the elements in ascending order
Iterator **descendingIterator**()	It is used iterate the elements in descending order.
Comparator<? super E> **comparator**()	It returns comparator that arranged elements in order.
SortedSet **headSet**(E toElement)	It returns the group of elements that are less than the specified element.
SortedSet **tailSet**(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
SortedSet **subSet**(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement

Example:

```
TreeSet<String> set=new TreeSet<String>();
set.add("A");          set.add("B");          set.add("C");          set.add("D");          set.add("E");

    System.out.println("Reverse Set: "+set.descendingSet());     // [E, D, C, B, A]

    System.out.println("Head Set: "+set.headSet("C", true));     //  [A, B, C]

    System.out.println("SubSet: "+set.subSet("A", false, "E", true));   //  [B, C, D, E]

    System.out.println("TailSet: "+set.tailSet("C", false));     //  [D, E]
```

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of a Comparable type**. String and Wrapper classes are Comparable by default**. To add *user-defined objects* in TreeSet, you need to implement the **Comparable interface**

```
import java.util.*;
class Book implements Comparable<Book>{
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
public int compareTo(Book b) {
    if(id>b.id){
        return 1;
    }else if(id<b.id){
        return -1;
    }else{
    return 0;
    }
}
}
```

```
public class TreeSetExample {
public static void main(String[] args) {
    Set<Book> set=new TreeSet<Book>();

    Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
    Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

    set.add(b1);
    set.add(b2);
    set.add(b3);

    for(Book b:set){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Queue Interface

FIFO(First In First Out) manner

Method	Description
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Java Deque Interface

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "**double ended queue".**

ArrayDeque class

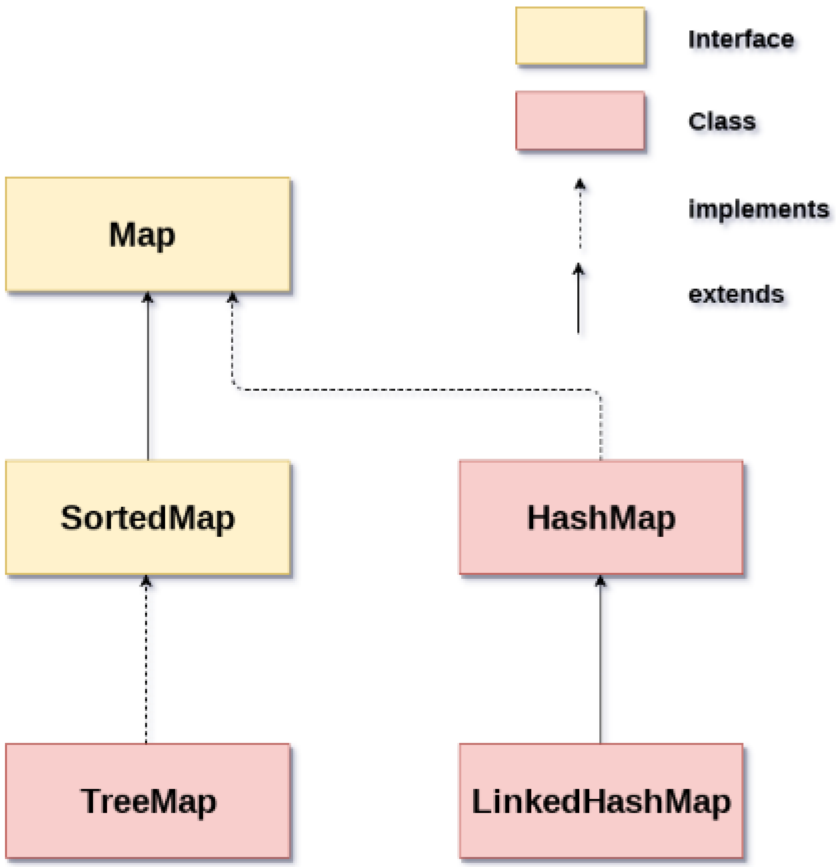
The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are

- Unlike Queue, we can **add or remove elements from both sides**.
- **Null** elements are **not allowed** in the ArrayDeque.
- ArrayDeque is **not thread safe**, in the absence of external synchronization.
- ArrayDeque has **no capacity restrictions**.
- ArrayDeque is **faster than LinkedList and Stack**.

offerFirst()	Add as the 1st element
pollLast()	Removes the last/recently added element

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

- 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.

METHODS of **MAP** INTERFACE

Method	Description
V **put**(Object key, Object value)	It is used to insert an entry in the map.
void **putAll**(Map map)	It is used to insert the specified map in the map.
V **putIfAbsent**(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.

V **remove**(Object key)	It is used to delete an entry for the specified key.
void **clear**()	It is used to reset the map.
boolean **containsValue**(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean **containsKey**(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean **equals**(Object o)	It is used to compare the specified Object with the Map
V **get**(Object key)	This method returns the object that contains the value associated with the key.
V **getOrDefault**(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int **hashCode**()	It returns the hash code value for the Map
boolean **isEmpty**()	This method returns true if the map is empty; returns false if it contains at least one key.
V **replace**(K key, V value)	It **replaces** the specified **value** for a specified key.
Set **keySet**()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> **entrySet**()	It returns the Set view containing all the keys and values.

Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name.
It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value

Method	Description
K **getKey**()	It is used to obtain a **key**.
V **getValue**()	It is used to obtain **value**.
int **hashCode**()	It is used to obtain hashCode.
V **setValue**(V value)	It is used to replace the **value** corresponding to this entry with the specified value.
Boolean **equals**(Object o)	It is used to compare the specified object with the other existing objects.

Example code:

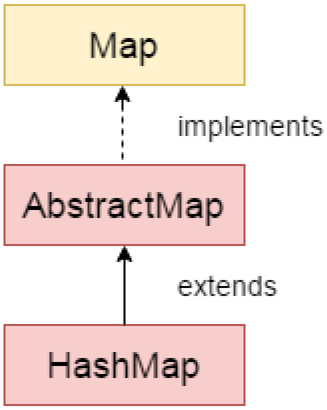
```
Map<Integer,String> map=new HashMap<Integer,String>();
  map.put(100,"Amit");    map.put(101,"Vijay");    map.put(102,"Rahul");

  for(Map.Entry m : map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue()); }
```

comparingByKey(Comparator <> cmp)	It returns a comparator that compare the objects by key using the given Comparator.
comparingByValue(Comparator< > cmp)	It returns a comparator that compare the objects by value using the given Comparator.

```
Map<Integer,String> map=new HashMap<Integer,String>();
 map.put(100,"Amit");
 map.put(101,"Vijay");
 map.put(102,"Rahul");
                map.entrySet()
                      .stream()
                      .sorted(Map.Entry.comparingByKey())
                      .forEach(System.out::println);

// Output: 100=Amit  101=Vijay 102=Rahul

/*
                      .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))

                      .sorted(Map.Entry.comparingByValue())

                      .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
*/
```

HashMap

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



It allows us to store the null elements as well, but there should be **only one null key**

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

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the 'value'.

Map.put(1,"Tony");   Map.put(1,"Stark");

It will consider as {1=Stark}

Working of HashMap in Java

Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches

HashMap

- It uses a technique called Hashing.
- It implements the map interface.
- It stores the data in the pair of Key and Value.
- 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.
- The default size of HashMap is 16 (0 to 15).

- **equals():** It checks the equality of two objects. It **compares** the **Key**, whether they are equal or not. It is a method of the Object class. ***It can be overridden***. If you override the equals() method, ***then it is mandatory to override the hashCode() method.***

- **hashCode():** This is the method of the object class. 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**.
- **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

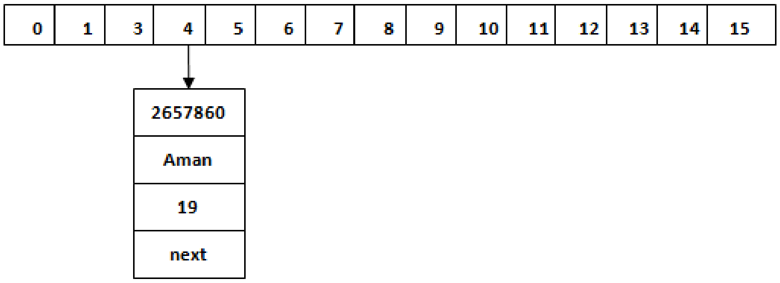
Calculating Index of HashMap

Index minimizes the size of the array. The Formula for calculating the index is:

Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

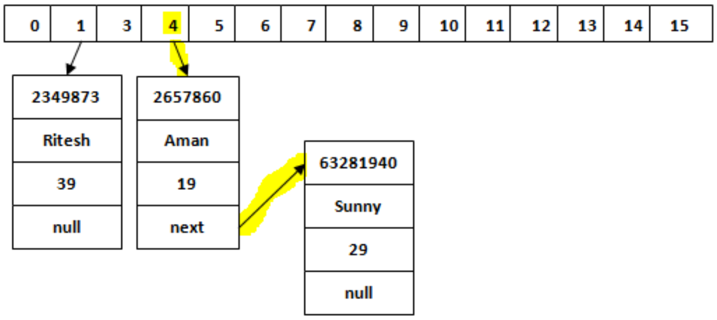
Index = 2657860 & (16-1) = 4



Hash Collision (when the calculated index value is the same for two or more Keys)

In this case, equals() method check that both Keys are equal or not.

If Keys are same, replace the value with the current value
Otherwise, connect this node object to the existing node object through the LinkedList



get() method

```
map.get(new Key("Aman"));
```

1. get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key
2. When get(K Key) method is called, it calculates the hash code of the Key.
3. Now calculate the index value by using index formula
4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists
5. It returns null if the next of the node is null

LinkedHashMap class

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order

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

Collection<V> **values**()	It returns a Collection view of the **values** contained in this map.
Set<K> **keySet**()	It returns a Set view of the **keys** contained in the map
Set<Map.Entry<K,V>> **entrySet**()	It returns a Set view of the **mappings** contained in the map.

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();

map.put(100,"Amit");        map.put(101,"Vijay");        map.put(102,"Rahul");

 System.out.println("Keys: "+map.keySet());
  // Keys: [100, 101, 102]
  System.out.println("Values: "+map.values());
  // Values: [Amit, Vijay, Rahul]
  System.out.println("Key-Value pairs: "+map.entrySet());
 //Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]
```

Java TreeMap class

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:

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

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<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.
boolean **containsKey**(Object key)	It returns true if the map contains a mapping for the specified key.
boolean **containsValue**(Object value)	It **returns** true if the map maps one or more keys to the specified value.

Map.Entry **firstEntry**()	It **returns** the key-value pair having the least key.
Map.Entry<K,V> **lastEntry**()	It **returns** the key-value pair having the greatest 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
K **firstKey**()	It is used to return the first (lowest) key currently in this sorted map
K **lastKey**()	It is used to return the last (highest) key currently in the sorted map.
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.
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

As the Tree map is sorted in ascending order, we use these 2 methods exclusively to fetch Descending values

NavigableSet<K> **descendingKeySet**()	It returns a reverse order NavigableSet view of the **keys** contained in the map.
NavigableMap<K,V> **descendingMap**()	It returns the specified **key-value pairs** in descending order

SortedMap<K,V> **subMap**(K fromKey, K toKey)	It returns key-value pairs whose keys range *from fromKey*, inclusive, *to toKey*, exclusive.
SortedMap<K,V> **headMap**(K toKey)	It returns the key-value pairs whose keys are strictly *less than toKey*
SortedMap<K,V> **tailMap**(K fromKey)	It returns key-value pairs whose keys are *greater than or equal to fromKey*.

Example program for the above 3 methods:

```
NavigableMap<Integer,String> map=new TreeMap<Integer,String>();

map.put(100,"Amit"); map.put(102,"Ravi");  map.put(101,"Vijay"); map.put(103,"Rahul");

    System.out.println("descendingMap: "+map.descendingMap());
           //descendingMap: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

    System.out.println("headMap: "+map.headMap(102,true));
           //headMap: {100=Amit, 101=Vijay, 102=Ravi}

    System.out.println("tailMap: "+map.tailMap(102,true));
           //tailMap: {102=Ravi, 103=Rahul}

    System.out.println("subMap: "+map.subMap(100, false, 102, true));
          //subMap: {101=Vijay, 102=Ravi}
```

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