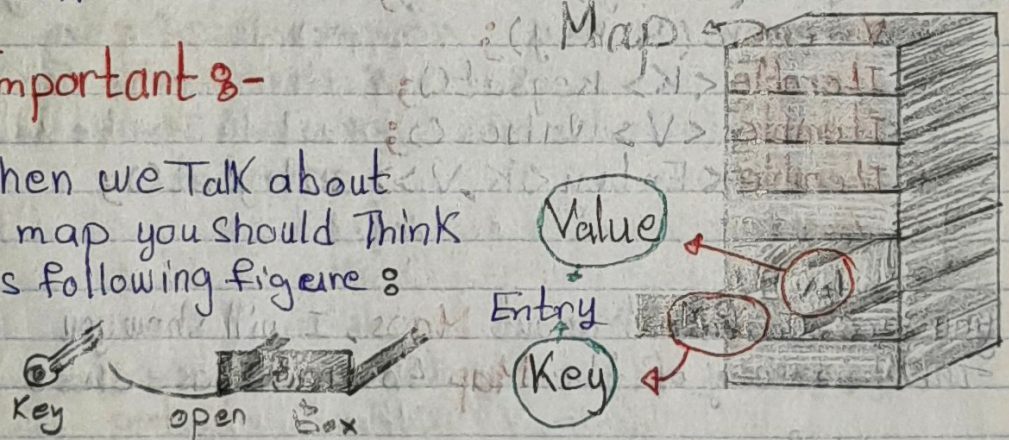
Maps, Hash Tables, and Skip Lists :-

Maps :

Def: A map is an abstract dataType designed to efficiently store and retrieve values based on a uniquely identifying Search-Key for each

Important :-

when we Talk about a map you should Think as following figure :



and here Some examples on a map Structures :

ex: 1. A university's information System relies on Some for of Student ID as a Key that is mapped to that student's AssociationRecord serving as a Value.

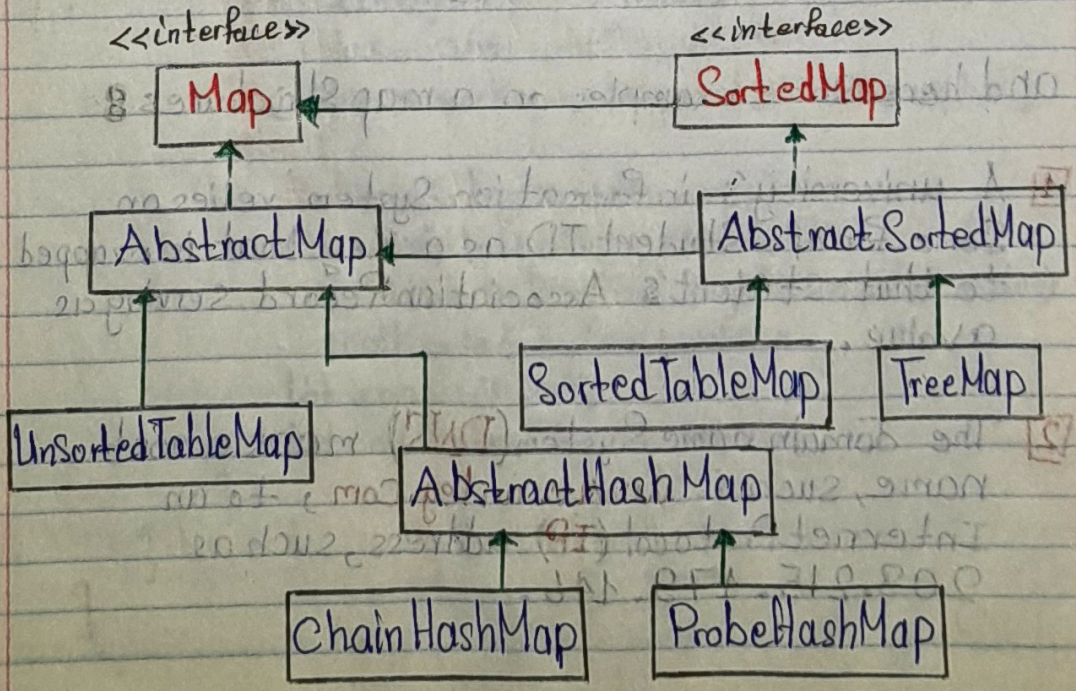
ex: 2. The domain name System (DNS) maps a host name, such as www.wiley.com, to an InternetProtocol (IP) address, such as 208.215.179.146.

now let me show you the Map ADT interface as following Syntax :

Codes

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    V get(K key); //get value at a key
    V put(K key, V value); //add value with a key
    V remove(K key); //remove value of a key
    Iterable<K> keySet(); // return Iterable keys
    Iterable<V> values(); //return Iterable values
    Iterable<Entry<K,V>> entrySet();
}   //return Iterable entries (K,V)
```

you are confused about Maps, I will show you the hierarchy of the Map Abstract base Class :



now let's implement AbstractMap class as following Syntax:

Code:

```
public abstract class AbstractMap<K,V>
  implements Map<K,V>{
  public boolean isEmpty(){ return size()==0;}
  protected static class MapEntry<K,V>
  implements Entry<K,V>{
      private K k;
      private V v;
      public MapEntry(K key, V value){
          K = key;
          v = value;}
      public K getKey(){ return K;}
      public V getValue(){ return v;}
      protected void setKey(K key){ K = key;}
      protected V setValue(V value){
          V old = V;
          v = value;
          return old;}
  }
  private class KeyIterator implements Iterator<K>
  {
      private Iterator<Entry<K,V>> entries
          = entrySet().iterator();
      public boolean hasNext(){entries.hasNext();}
      public K next(){return entries.next().getKey();}
      public void remove(){throw new UnsupportedOperation
          Exception();
  }
}
```

EntryClass →

KeyIterator Class →

```
private class KeyIterable implements Iterable<K>{
    public Iterator<K>iterator(){
      return new KeyIterator();}
}
public Iterable<K> KeySet(){
   return new KeyIterable();
}
private class ValueIterator Implements Iterator<V>
{
   privat Iterator<Entry<K,V>> entries
       = entrySet().Iterator();
    public boolean hasNext(){return entries.hasNext();}
    public V next(){return entries.next().getValue();}
    public void remove(){
        throw new UnsupportedOperationException();
    }
}
private class ValueIterable implements Iterable<V>{
    public Iterator<V> iterator(){
      return new ValueIterator();
    }
}
public Iterable <V> values(){
   return new ValueIterable();
}
}
```

In the previous abstract Class for Map D.S any Map should extends That abstract Class.

Each of the fundamental methods get(K), put(K,V), and remove(K) requires an initial scan of the array to determine whether an entry with key equal to K exists. For this reason, we provide a nonpublic utility, findIndex(key), that returns the index at which such an entry is found, or -1 if no such entry is ~~found~~ found.

Because of scan process in the map all fundamental methods of an UnSortedTableMap class aren't very efficient such that each one takes O(n) for the scanning process.

now Leb's implement the UnsortedTableMap class as following Syntax:

Code:

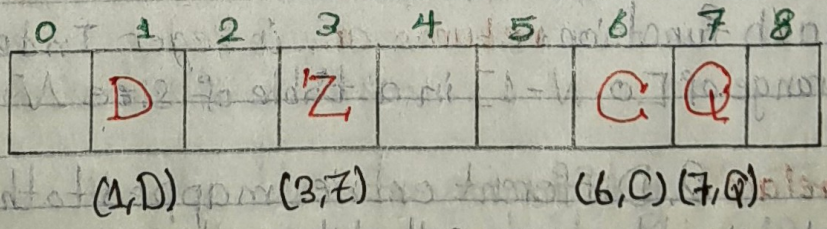
```
public class UnsortedTableMap <K,V>
    extends AbstractMap <K,V> {
  private ArrayList <MapEntry<K,V>> table
    = new ArrayList <>();
  public UnsortedTableMap(){}
  private int findIndex (K key) {
    int n = table.size();
    for (int j=0; j < n; j++)
      if (table.get(j).getKey().equals(key))
        return j;
    return -1;
  }
  public int size(){retun table.size();}
```

```
public V get(K key){
    int j = findIndex(key);
    if (j == -1) return null;
    return table.get(j).getValue();
}
public V put(K key, V value) {
    int j = findIndex(key);
    if (j == -1) {
        table.add(new EntryMap<>(key, value));
        return null;
    } else return table.get(j).setValue(value);
}
public V remove(K key) {
    int j = findIndex(key);
    int n = size();
    if (j == -1) return null;
    V answer = table.get(j).getValue();
    if (j != n-1)
        table.set(j, table.get(n-1));
    table.remove(n-1);
    return answer;
}
private class EntryIterator implements
    Iterator<Entry<K,V>> {
    private int j = 0;
    public boolean hasNext() {
        return j < table.size(); }
    public Entry<K,V> next() {
        if (j == table.size)
            throw new NoSuchElementException();
        return table.get(j++);
    }
```

```
        public void remove() {
            throw new UnSupportedOperationException();
        }
    }
    private class EntryIterable implements
        Iterable<Entry<K,V>> {
        public Iterator<Entry<K,V>> iterator(){
            return new EntryIterator();
        }
    }
    public Iterable<Entry<K,V>> entrySet() {
        return new EntryIterable();
    }
}
```

Hash Tables:

Def: It's an implemented DataStructure and most efficient application for the Map. we can represent it in the following figure:



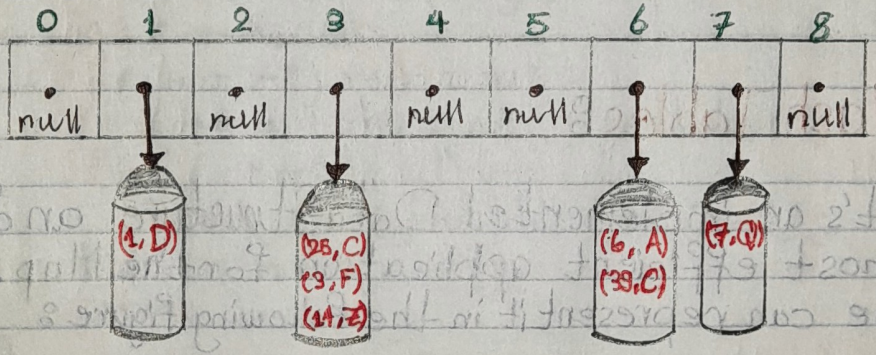
we called it a Hash Table which use something called Hash function to map keys of the Map To the Coresponding indices in the table and it takes O(1) in the worst case.

but, we have 2 problems

First: we may not wish to devote an array of length N if it in case that $N >> n$.

Second: The novel concept for a hash table using hash function may map more than 1 key to the same index.

As a result we conceptualize our table as a Bucket Array such that every index in the table holds a list of entries $(i, h(i))$ and we can represent it as following figure:

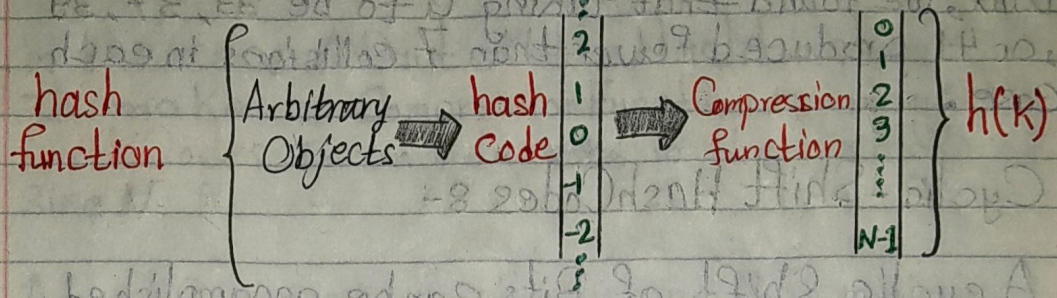


The Hash function returns an integer Index in the range of $[0, N-1]$ in a table of size N

notes: If there are 2 Different entries mapped to the same index then we called it Collision.

notes: We say that a hash function is good if it maps the keys in our map so as to sufficiently minimize Collisions.

The following figure illustrates the components of the Hash function :



notes — The Hash Code Independent of a specific hash Table size but, The Compression function depends upon the table size.

we have 3 approaches for generating Hash Codes :

[1] Treating the Bit Representation as an Integer :-

A simple implementation is to add 2 components as 32-Bit numbers or take the XOR of the 2 components as following formulas :

$$h.c = \sum_{i=0}^{N-1} X_i = X_0 \hat{} X_1 \hat{} X_2 \hat{} \cdots \hat{} X_{n-1}$$

[2] Polynomial Hash Codes :-

If we denote the Component by X_i Then we choose a non-zero Constant $a \neq 1$ Such That the hash Code can be Generated by the following Polynomial :

$$h.c = X_0 a^{n-1} + X_1 a^{n-2} + \cdots + X_{n-1}$$

note: In a list of 50,000 English words formed as Union of the word lists provided in two variants of Unix, we found that taking a to be 33, 37, 39, or 41 produced fewer than 7 collisions in each case.

[3] Cyclic-Shift HashCodes :-

A cyclic shift of Bits can be accomplished through careful use of the BitWise Shift operators as following Syntax:

Code:

```
static int hashCode(String s){
  int h = 0;
  for(int i=0; i<s.length(); i++){
      h = (h<<5) | (h>>>27);   // Unsigned Right Shift
      h += (int) s.charAt(i);
  }
  return h;
}
```

mentioned below

note: In an Experiment of using Cyclic Shift in hashing 230,000 English word by 5-Bits we found that 190 from 230,000 will make collisions at least with another word.

The Object class provide a default version of hashCode function that returns 32-Bit integer drived from the object's (memory address).

notes: WrapperClasses have their own implementation of hashCode function in addition to StringClass.

in the second Part of the HashFunction is To make a CompressionFunction To map hashCodes To an index in the HashTable of size N

A good CompressionFunction is one that minimizes The number of Collisions for a given set of distinct HashCodes and we will introduce Two methods for that behaviour :

1. The Division Method :-

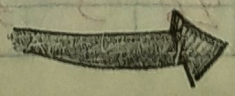
   $i \longrightarrow i \bmod N$ N: size of the bucket Array

2. Multiply_Add and Divide: (MAD)

   $i \longrightarrow [(ai+b) \bmod p] \bmod N$

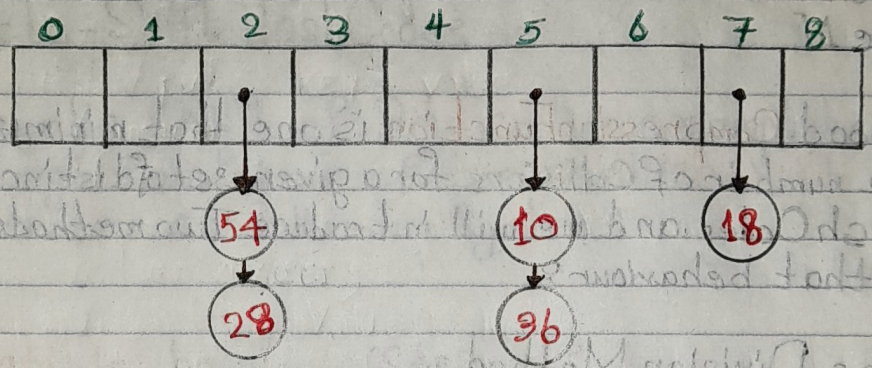
   - N: size of the bucket Array
     p: Prime number greater than N
     a,b: integers chosen randomly in the interval $[0, P-1]$ with $a > 0$

now we are handling the Collisions by Two approaces Separate chaining and OpenAddressing :



[1] Separate Chaining :-

each bucket A[j] store it's own secondary container, holding all entries (k, v) such that $h(k) = j$.

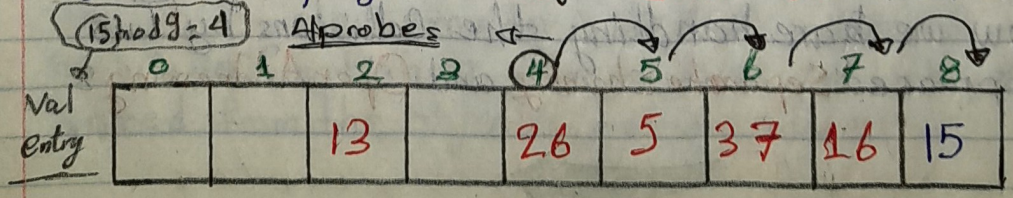


if we need to insert n entries into hashTable of ~~size~~ size N then the Loadfactor of the HashTable $\lambda = \frac{n}{N}$ should be bounded by a small constant preferably below 1 such that the core map operations run in $O(\lambda) = O(1)$.

[2] Open Addressing :-

Is to try find an empty index in the hash-Table and we have 3 strategies to do that

a. Linear Probing : start search from A[h(k)] until find an empty slot A[j + i] mod N. i = 1, 2, 3, ~~N-1~~.



b. Quadratic probing : tries iteratively the buckets $A[h(k) + f(i)]$, $i = 0, 1, 2, \ldots$ $f(i) = i^2$ until finding an empty bucket.

c. Double Hashing : we choose a secondary hash func. h', and if h maps some key k to a bucket $A[h(k)]$ that is already occupied, then we iteratively try buckets $A[(h(k) + f(i)) \bmod N]$ next, $i = 1, 2, 3, \ldots$ $f(i) = i \cdot h'(k)$ where $h'(k) = q - k \bmod q$, $q < N$, q, N are primes.

notes Separate chaining has a load factor $\lambda < 0.9$.

now let's implement the Abstract HashMap Class as following syntax :

Codes

```
public abstract class AbstractHashMap <K, V>
    extends AbstractMap<K,V> {
    protected int n=0;
    protected int capacity;
    private int prime;
    private long scale, shift;
    public AbstractHashMap(int cap, int p){
        prime = p;
        capacity = cap;
        Random rand = new Random();
        scale = rand.nextInt(Prime-1) + 1;
        shift = rand.nextInt(Prime);
        createTable();
    }
```

```
public AbstractHashMap(int cap) {
    this(cap, 109345121); }
public AbstractHashMap() { this(17); }
public int size() { return n; }
public V get(K key) { return bucketGet(hashValue(key), key); }
public V remove(K key) {
    return bucketRemove(hashValue(key), key); }
public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key);
    if (n > capacity / 2)
        resize(2 * capacity - 1);
    return answer; }
private int hashValue(K key) {
    return (int)((Math.abs(key.hashCode() * scale + shift)
            % prime) % capacity); }
private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer
        = new ArrayList<>(n);
    for(Entry<K,V> e : entrySet())
        buffer.add(e);
    capacity = newCap;
    createTable();
    n = 0;
    for(Entry<K,V> e : buffer)
        put(e.getKey(), e.getValue()); }
protected abstract void createTable()
protected abstract V bucketGet(int h, K k);
protected abstract V bucketPut(int h, K k, V v);
protected abstract V bucketRemove(int h, K k);
}
```

Sorted Maps:

In this Section we will introduce an extension as The SortedMap ADT That includes all behaviours of the standard map, plus the following:

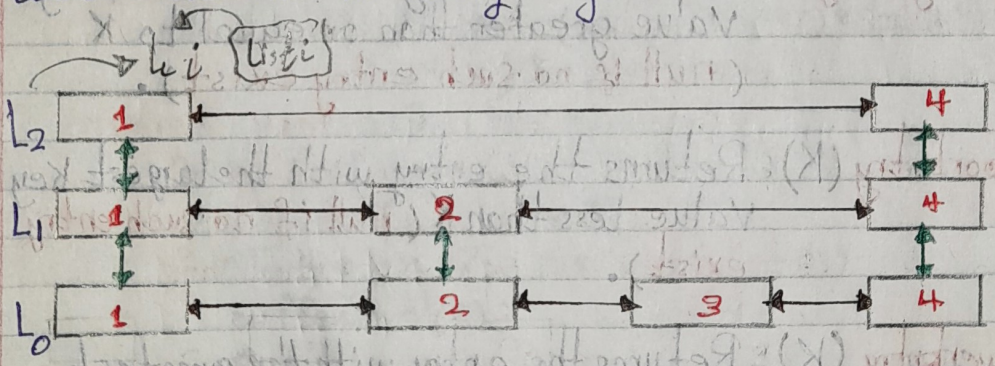
1. firstEntry(): Returns the entry with the smallest Key value (null if the map is empty).
2. LastEntry(): Returns the largest Key value entry (or null if The map is empty).
3. ceilingEntry(K): Returns the entry with the least Key value greater than or equal to K (null if no such entry exist).
4. floorEntry(K): Returns the entry with the largest Key value less than K (null if no such entry exist).
5. LowerEntry(K): Returns the entry with the greatest Key Value strictly less than K (null if no such entry exist).
6. higherEntry(K): Returns the entry with Least Key Value strictly greater than K (null if no such entry exist).
7. SubMap(K_1, K_2): Returns an iteration of all entries with key greater or equal to K_1 but strictly less than K_2.

Note: All the preceding methods included within the java.util.NavigableMap interface which extends the simpler java.util.SortedMap interface.

SkipLists:

Def: It's a two dimensional collection of positions arranged horizontally into Levels and vertically into towers.

Each level is a list S_i and each tower contains positions storing the same entry accross consecutive lists as the following figure:



and for traversing a skiplist we have the following Operations:

1. next(P): Returns the position following P on the same level.
2. prev(P): Returns the position preceding P on the same level.
3. above(P): Returns the position above P in the same level.
4. below(P): Returns the position below P in the same level.

The Skip list is a powerful approach for searching in a Map (→ SortedMaps) by the following algorithms:

// O(log n) Time Complexity

```
Algorithm SkipSearch(K):
    p = s
    while below(p) ≠ null do
        p = below(p)
        while K ≥ key(next(p)) do
            p = next(p)
    return p
```

and for insertions we have the following Algorithm:

// O(log n) Time Complexity

```
Algorithm SkipInsert(K, V):
    p = SkipSearch(K)
    q = null
    i = -1
    repeat
        i = i + 1
        if i ≥ h then
            h = h + 1
            t = next(s)
            s = insertAfterAbove(null, s, (-∞, null))
                insertAfterAbove(s, t, (+∞, null))
        q = insertAfterAbove(p, q, (K,V))
        while above(p) == null do
            p = prev(p)
        p = above(p)
    until coinFlip() == tails
    n = n + 1
    return q
```

Head of the list ↑

Tail of the list ↓

Sets, Multisets, and Multimaps:

In this chapter Specifically this section we will introduce 3 dataStructures have The same MapStructure:

1| Set ADT:-

Def: A set is an unordered collection of elements without duplicates, that typically supports efficient membership tests.

In essence, elements of a set are like Keys of a map, but without any auxiliary values.

now let's show the java.util.set interface :

Code:

```
public interface Set<E> {
    boolean add(E e);
    boolean remove(E e);
    boolean contains(E e);
    Iterator<E> iterator();
    boolean addAll(Set<E> S); // S∪T
    boolean retainAll(Set<E>S); // S∩T
    boolean removeAll(Set<E>S); // S-T
}
```

and for the sorted sets we have another SortedSet ADT which can be viewed as following Syntax :

1. first(): Returns the smallest element in S.
2. Last(): Returns the largest element in S.
3. ceiling(e): Returns the smallest element greater than or equal to e.
4. floor(e): Returns the largest element less than or equal to e.
5. Lower(e): Returns the largest element strictly less than e.
6. higher(e): Returns the smallest element strictly greater than e.
7. subSet(e_1, e_2): Returns an iteration of all elements greater than or equal to e_1 but strictly less than e_2.
8. pollFirst(): Returns and removes the smallest element in S.
9. pollLast(): Returns and removes the largest element in S.

and we have Three implementations for a Set Data structure

- java.util.HashSet; // Unsorted
- java.util.Concurrent.ConCurrentSkipListSet;
- java.util.TreeSet; // Sorted

2 MultiSet ADT :-

Def: A multiset also known as a Bag is a set-like Container that allows duplicates.

Let's introduce the behaviours :

1. add(e): adds a single ocurrences of e to the multiset.
2. Contains(e): Returns true if the maltiset contains an element equal to e.
3. count(e): Returns the number of occurences of e in the multiset.

4. remove(e): Remove a single occurrence of e in the multiset.
5. remove(e, n): ~~Returns~~ Removes n occurrences of e in the multiset.
6. size(): Returns the number of elements in the multiset.
7. iterator(): Returns an iteration of the all elements in the multiset.

3 MultiMap ADT:-

Defs A multimap is similar to a traditional Map, in that it associates values with keys; however in multimap the same key can be mapped to mutiple values.

Now let's introduce the behaviours of the multimap:

1. get(K): Returns a collection of all values associated with key K in the multimap.
2. put(K, V): Adds a new entry to the multimap associating with key K with value V, without overwriting any existing mappings for key K.
3. remove(K, V): Removes an entry mapping key K to value V from the multimap.
4. removeAll(K): Removes all entries having key equal to K from the multimap.
5. size(): Returns the number of entries of the multiset.
6. entries(): Returns a collection of the all entries in the multimap.
7. keys(): Returns a collection of all keys in the multimap with duplicates.
8. keySet(): Returns a non-duplicative Collection of keys in the multimap.

9 Values(): Returns a collection of values for all entries in the multimap.