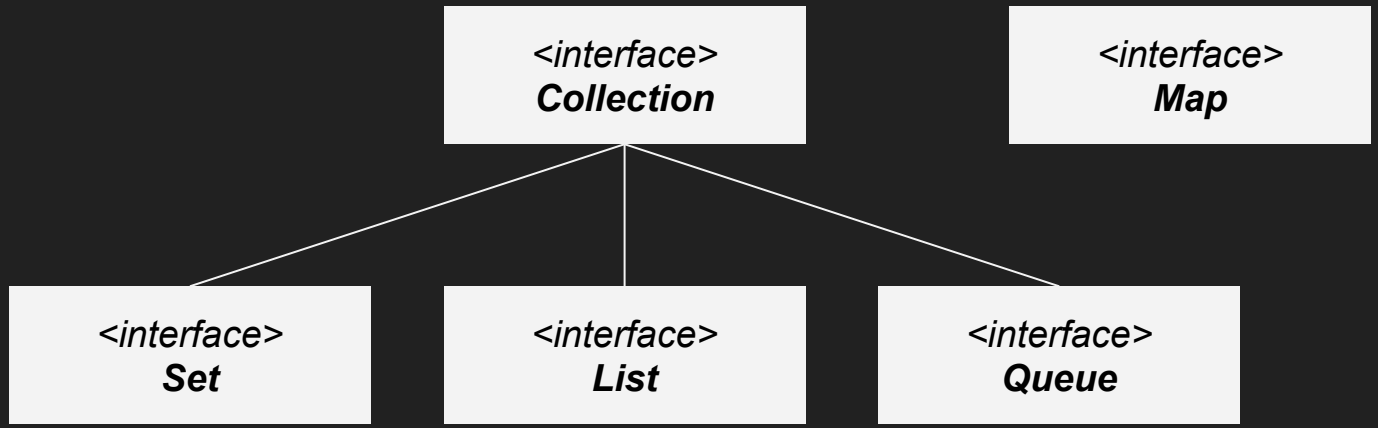
Java - Lesson 3

Collections and Generics

Author: Kirill Volkov
https://github.com/vulko/JL3.Collections
vulkovk@gmail.com

Basic interfaces

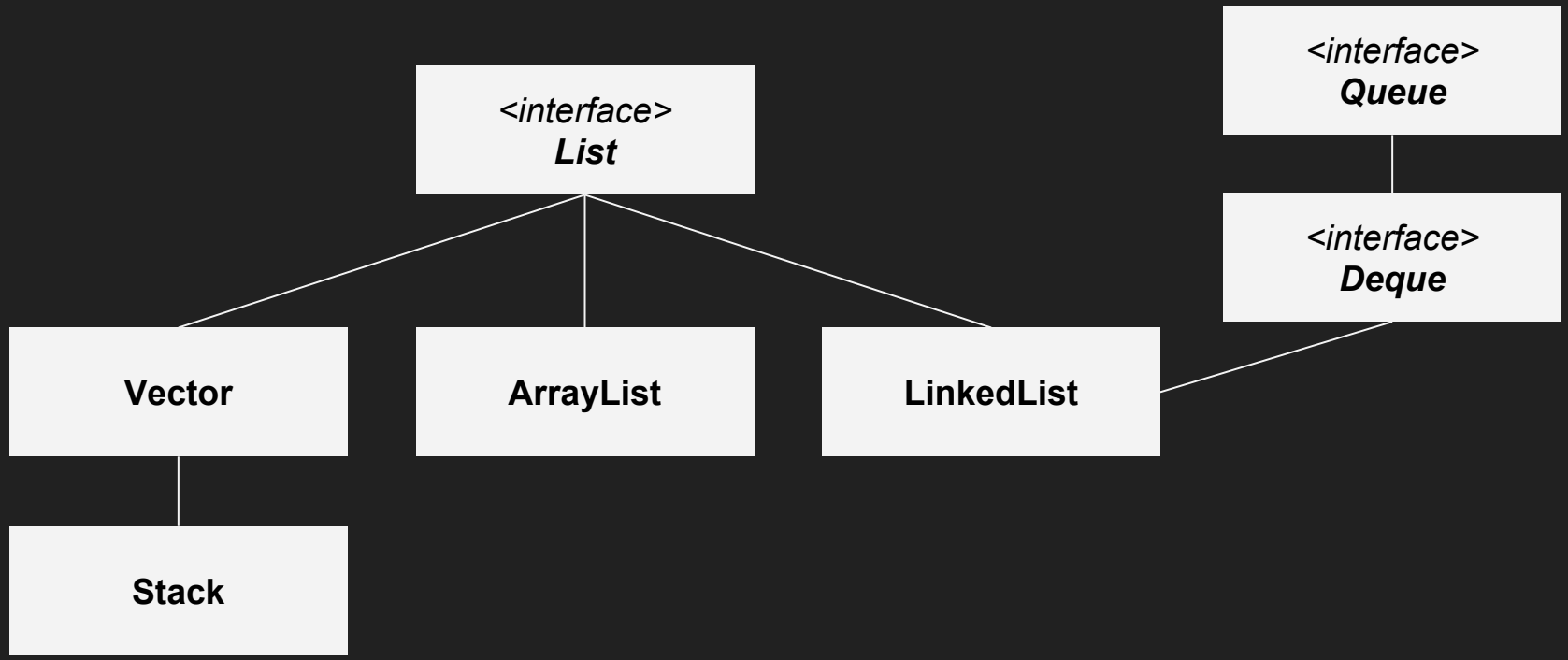
- Collection, items are Values
- Map, items are Key-Value pairs



Basic interfaces: Collection

```
boolean          add(E e)
boolean          addAll(Collection<? extends E> c)

void             clear()
boolean          isEmpty()
int              size()

boolean          contains(Object o)
boolean          containsAll(Collection<?> c)

boolean          remove(Object o)
boolean          removeAll(Collection<?> c)
boolean          removeIf(Predicate<? super E> filter)

Object[]         toArray()
<T> T[]          toArray(T[] a)
```

Basic interfaces: List



Basic interfaces: List

- **Vector**

  Dynamic array implementation. Accepts null values. Synchronized, so avoid using it!

- **Stack**

  Based on Vector as a LIFO (last-in-first-out) structure.

  Synchronized [besides put()], better use Deque.

- **ArrayList**

  Dynamic array implementation. Accepts null values.

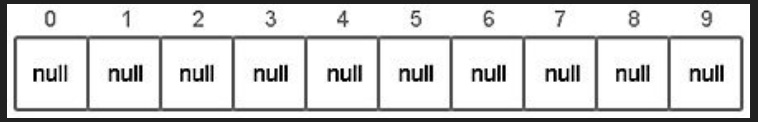
  Access by index is O(1). Insert and remove items in the middle is O(n)

- **LinkedList**

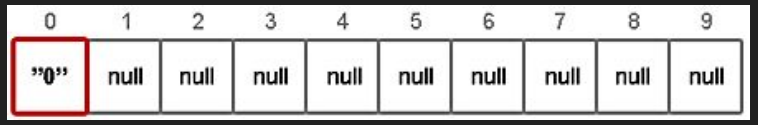
  Double-linked list. Add/remove by index is O(n), but for front and end it's O(1).

Lists: ArrayList

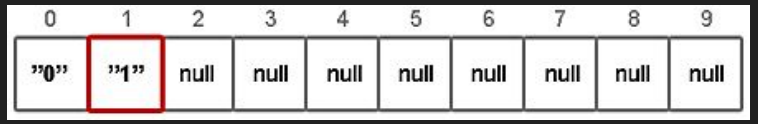
```
List<String> mList = new ArrayList<>(10);
```

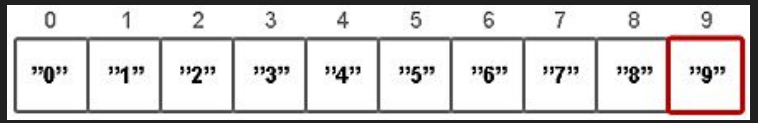



```
mList.add("0");
```

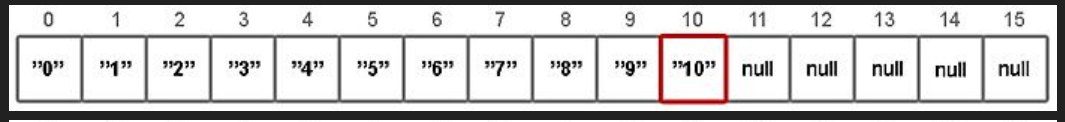


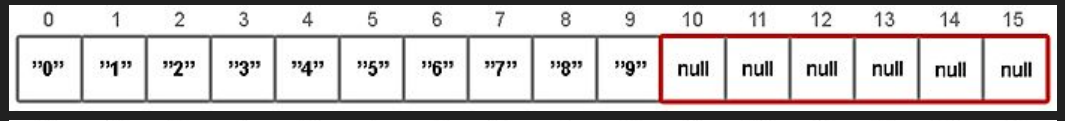

```
for (int i = 1; i < 9; ++i) {
    mList.add(String.valueOf(i));
}
```

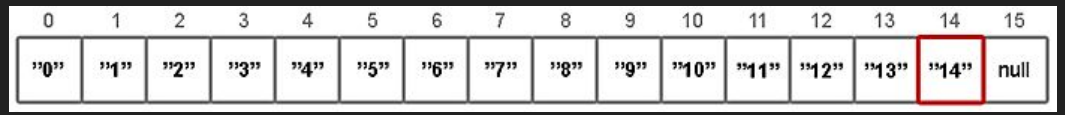





```
// this will create a copy of array using
// System.arrayCopy(), so it has enough
// capacity to add new elements

for (int i = 10; i < 14; ++i) {
    mList.add(String.valueOf(i));
}
```

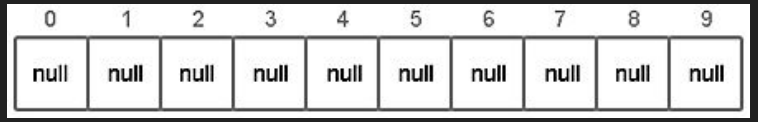




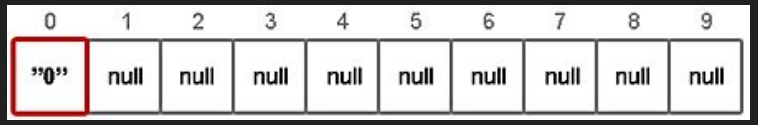


Lists: ArrayList

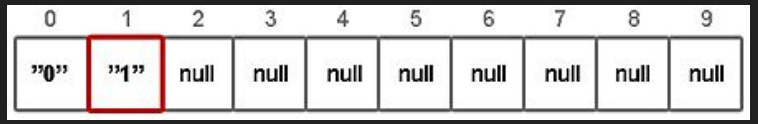
```
List<String> mList = new ArrayList<>(10);
```

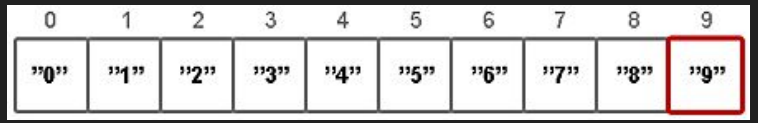



```
mList.add("0");
```

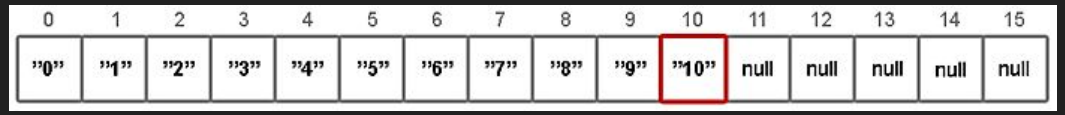


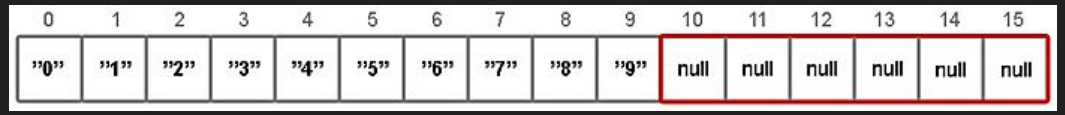

```
for (int i = 1; i < 9; ++i) {
    mList.add(String.valueOf(i));
}
```

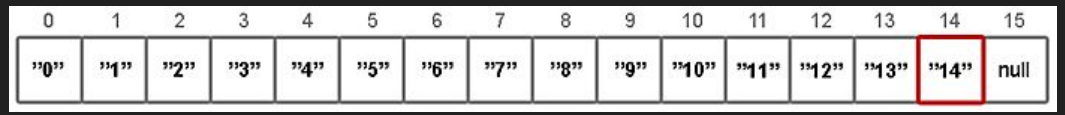





```
// this will create a new array using and copy
// old one using System.arrayCopy(),
// so it has enough capacity to add
// new elements

for (int i = 10; i < 14; ++i) {
    mList.add(String.valueOf(i));
}
```

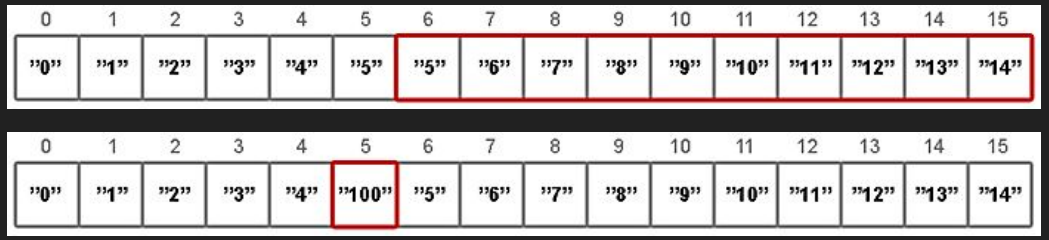






Lists: ArrayList

```
// now add item into the middle
list.add(5, "100");
```



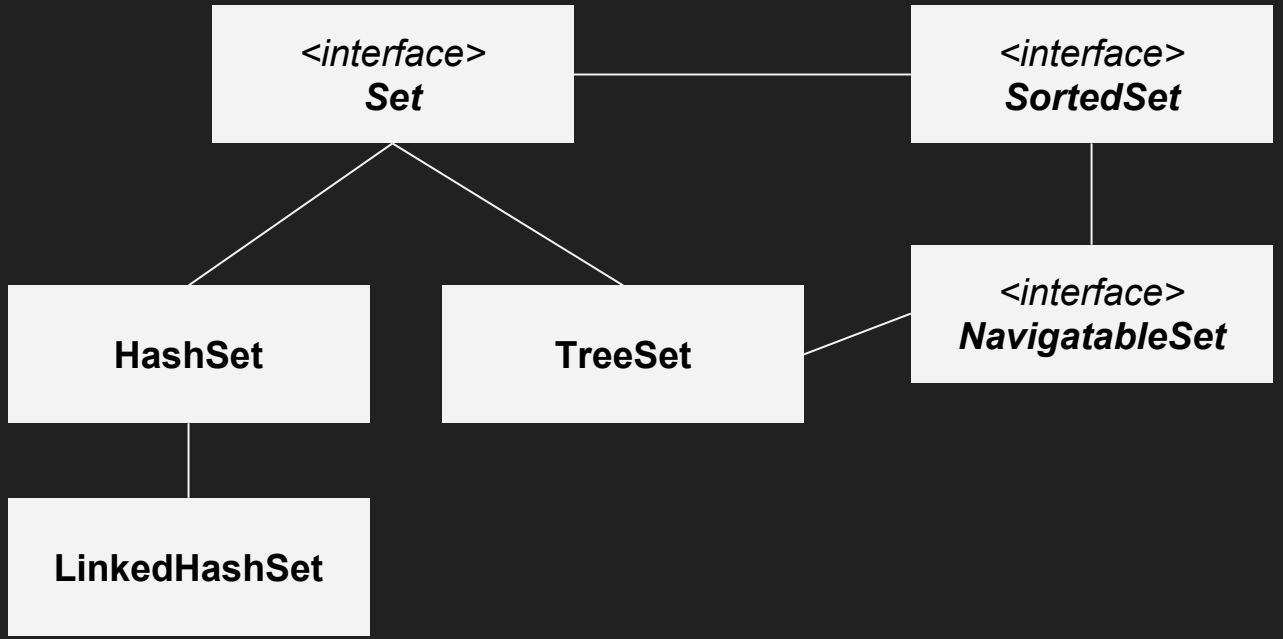
Good for:

- Add element to an empty place O(1), if enough capacity, O(n) otherwise

Bad for:

- Add/remove element in/from the middle O(n)
- Access element by value O(n)

Basic interfaces: Set



Basic interfaces: Set

- **HashSet**

  Based on HashMap.

  Key is an added element, value is an empty item (new Object()).

  No default ordering.

- **LinkedHashSet**

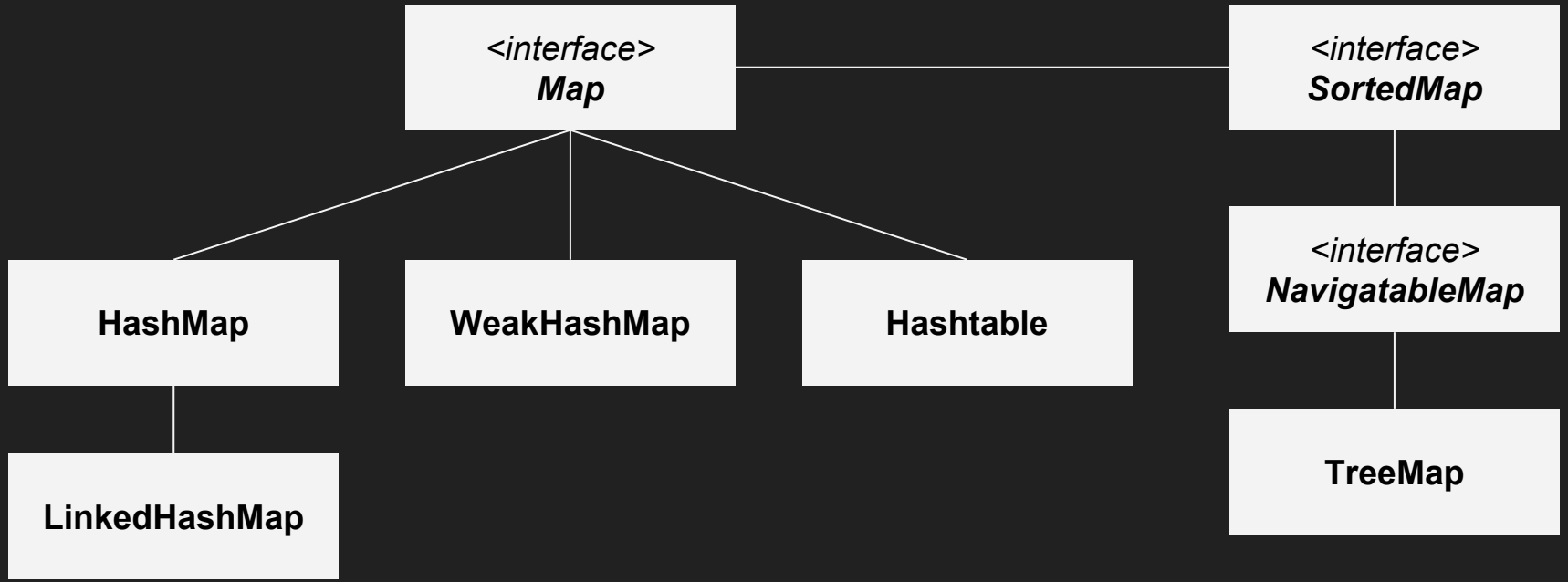
  Based on LinkedHashMap. Has natural ordering.

- **TreeSet**

  Based on NavigableMap.

  Allows to use Comparator to order elements or use natural one instead.

Basic interfaces: Maps



Basic interfaces: Map

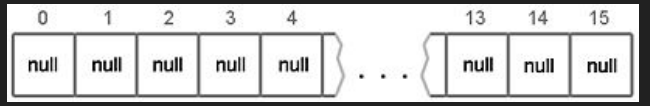
```
void                    clear()
boolean                 isEmpty()
Int                     size()

boolean                 containsKey(Object key)
boolean                 containsValue(Object value)

Set<Map.Entry<K,V>>     entrySet()
Set<K>                  keySet()
Collection<V>           values()

V                       get(Object key)
V                       getOrDefault(Object key, V defaultValue)

V                       put(K key, V value
void                    putAll(Map<? extends K,? extends V> m)
V                       putIfAbsent(K key, V value)

V                       remove(Object key)
boolean                 remove(Object key, Object value)
V                       replace(K key, V value)
boolean                 replace(K key, V oldValue, V newValue)
```

Basic interfaces: Maps

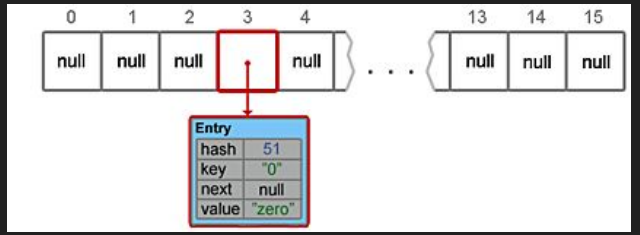
- **Hashtable**
  Doesn’t accept NULL key/values, almost all methods are synchronized. Better avoid using it.
- **HashMap**
  Accepts NULL key/values. Not synchronized. Adding an element is O(1), removing element is ammortized O(1).
- **LinkedHashMap**
  Same as HashMap, but elements within same key cell are stored in double-linked lists in the order they were added.
- **TreeMap**
  This is a Map based on Red-Black-Tree structure. By default "natural ordering" is used, but can be specified with a Comparator implementation, specified during construction.
- **WeakHashMap**
  HashMap where elements are stored as WeakRefs.

Maps: HashMap

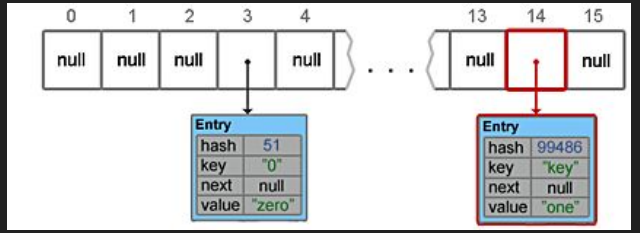
```
Map<String, String> mHashMap = new HashMap<>();
```



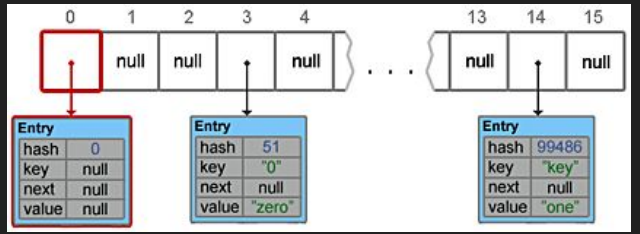

```
mHashMap.put("0", "zero");
```




```
mHashMap.put("key", "zero");
```

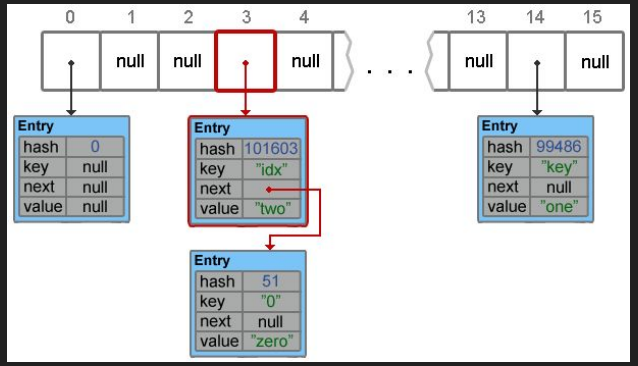



```
mHashMap.put(null, null);
```

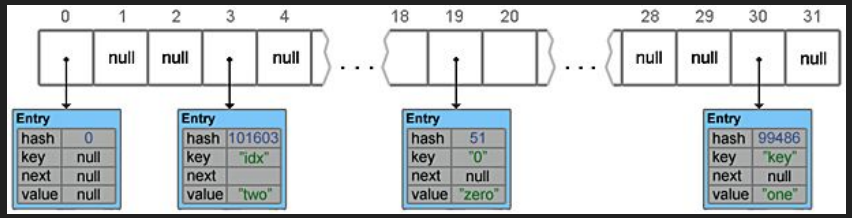


Maps: HashMap

```
mHashMap.put("idx", "two");
```




```
mHashMap.resize(32);
```



Maps: HashMap

- **Iterators are FAIL-FAST !!!** don’t change collection during iteration!

```
// 1.
for (Map.Entry<String, String> entry : hashmap.entrySet())
    System.out.println(entry. getKey() + " = " + entry. getValue());

// 2.
for (String key : hashmap.keySet())
    System.out.println(hashmap. get(key));

// 3.
Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();
while (itr.hasNext())
    System.out.println(itr .next());
```

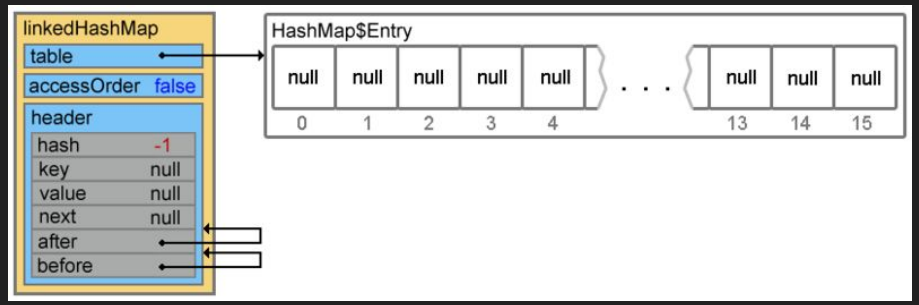
Good for:

- Add element is O(1) *amortized*
- Remove element is O(1) *amortized*
- Find element is O(1) *amortized*

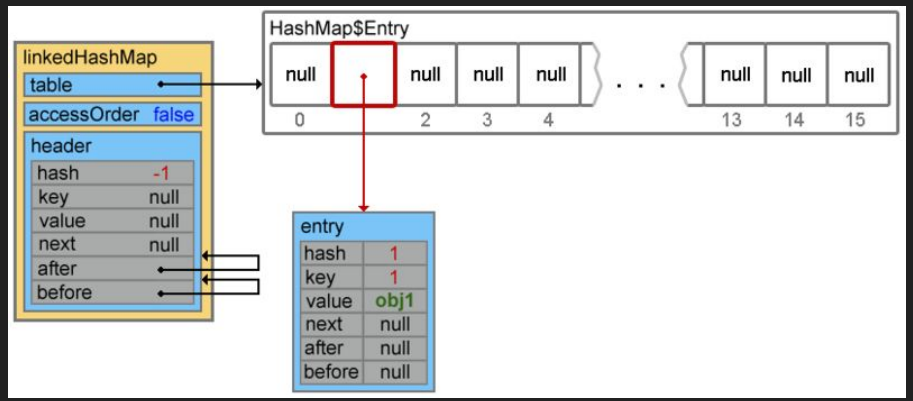
NOTE: in case of bad balancing these operations might take O(n), when all stored under single KEY.

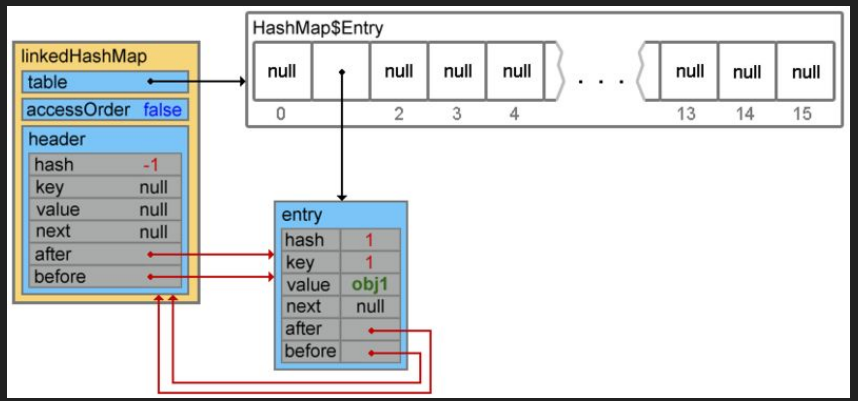
Maps: LinkedHashMap

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



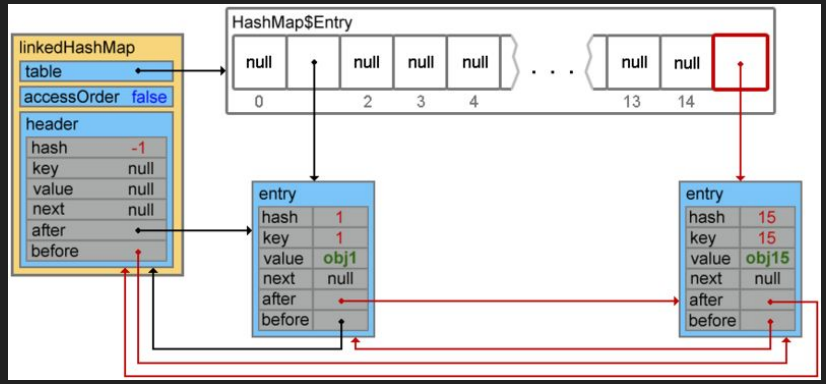

```
mLinkedHashMap.put(1, "obj1");
```



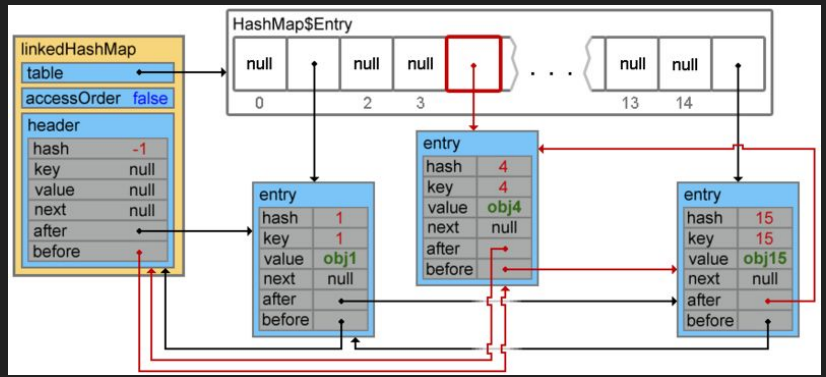


Maps: LinkedHashMap

```
mLinkedHashMap.put(15, "obj15");
```

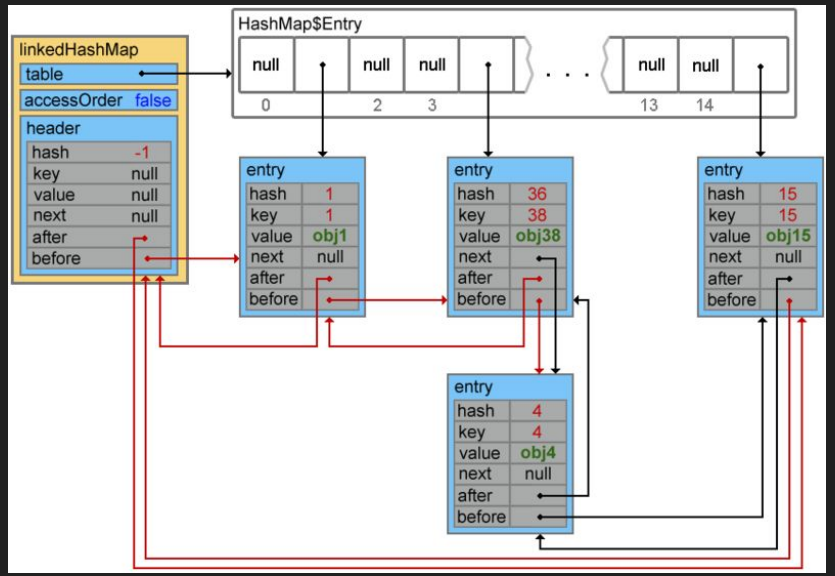


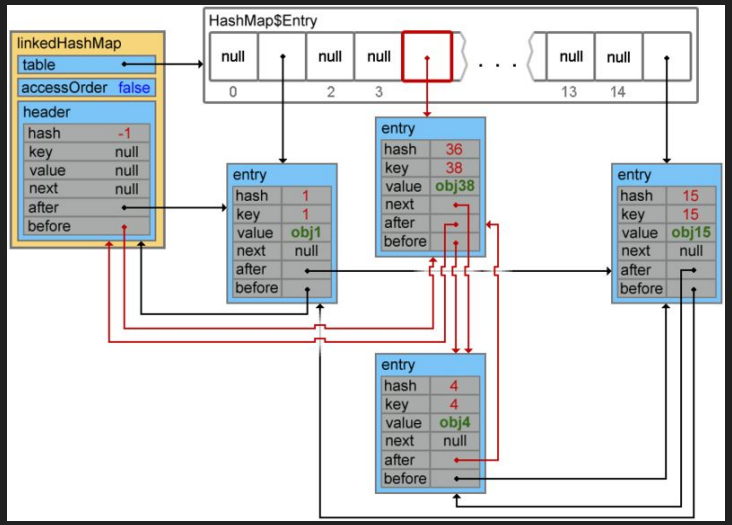

```
mLinkedHashMap.put(4, "obj4");
```



Maps: LinkedHashMap with accessOrder set to **true**

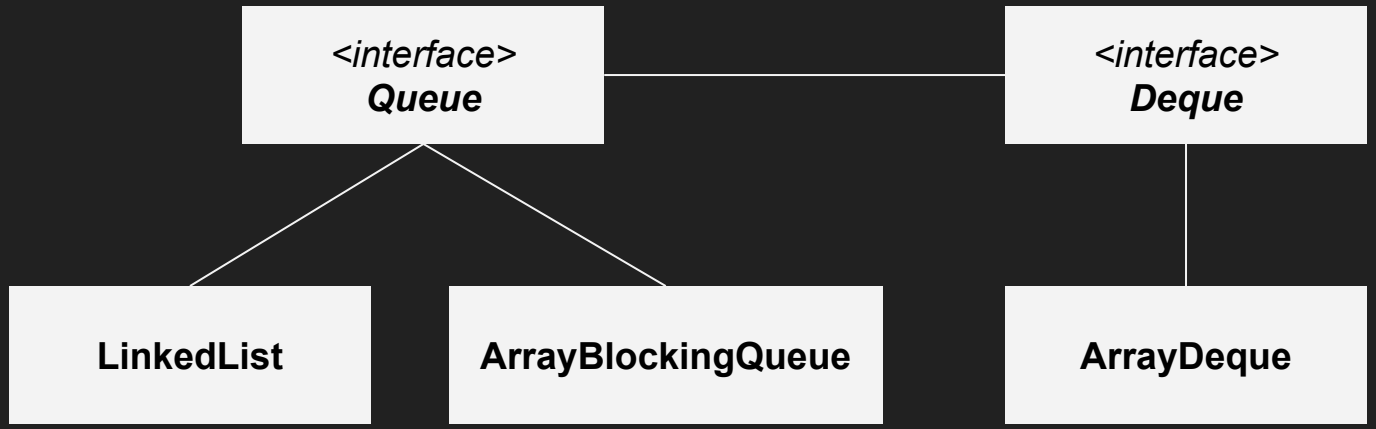
```
mLinkedHashMap.put(38, "obj38");
```






```
Map<Integer, String> linkedHashMap =
          new LinkedHashMap<Integer, String>(15, 0.75f, true) {{
                  put(1, "obj1");
                  put(15, "obj15");
                  put(4, "obj4");
                  put(38, "obj38");
              }};
    // {1 = obj1, 15 = obj15, 4 = obj4, 38 = obj38}

linkedHashMap.get(1); // or linkedHashMap.put(1, "Object1");
    // {15 = obj15, 4 = obj4, 38 = obj38, 1 = obj1}
```

Basic interfaces: Queue, Deque



Basic interfaces: Queue, Deque

- **Hashtable**
  Doesn’t accept NULL key/values, almost all methods are synchronized. Better avoid using it.
- **HashMap**
  Accepts NULL key/values. Not synchronized. Adding an element is O(1), removing element is ammortized O(1).
- **LinkedHashMap**
  Same as HashMap, but elements within same key cell are stored in double-linked lists in the order they were added.
- **TreeMap**
  This is a Map based on Red-Black-Tree structure. By default "natural ordering" is used, but can be specified with a Comparator implementation, specified during construction.
- **WeakHashMap**
  HashMap where elements are stored as WeakRefs.

Summary

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

Generics

Generics allow to abstract over types

Without type generalization:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

With type generalization:

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Using generics

A type generalization is defined as follows:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Any issues with the following code snippet?!

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;

lo.add(new Object());
String s = ls.get(0);
```

Using generics

A type generalization is defined as follows:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Any issues with the following code snippet?!

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;

lo.add(new Object());
String s = ls.get(0);
```

Wildcards

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Using a wildcard

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Wildcards

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());                          // Compile time error
```

Since we don't know what the element type of c is, we cannot add objects to it.

The add() method takes arguments of type E, the element type of the collection.

When the actual type parameter is ?, it stands for some unknown type.

Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in but **null**, which is a member of every type.

On the other hand, given a List<?>, we can call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

Wildcards

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { … }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { … }
}
```

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
   }
}

public void drawAll(List<? extends Shape> shapes) {
    ...
}
```

Generic methods

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o);                          // compile-time error
    }
}


// GENERIC METHOD

static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);                          // Correct
    }
}
```

```
Object[] objectArray = new Object[100];
Collection<Object> collection = new ArrayList<Object>();

// T inferred to be Object
fromArrayToCollection(objectArray, collection);

String[] stringArray = new String[100];
Collection<String> stringCollection = new ArrayList<String>();

// T inferred to be String
fromArrayToCollection(stringArray, stringCollection);

// T inferred to be Object
fromArrayToCollection(stringArray, collection);

Integer[] intArray = new Integer[100];
Float[] floatArray = new Float[100];
Number[] numberArray = new Number[100];
Collection<Number> numberCollection = new ArrayList<Number>();

// T inferred to be Number
fromArrayToCollection(intArray, numberCollection);

// T inferred to be Number
fromArrayToCollection(floatArray, numberCollection);

// T inferred to be Number
fromArrayToCollection(numberArray, numberCollection);

// T inferred to be Object
fromArrayToCollection(numberArray, collection);

// compile-time error
fromArrayToCollection(numberArray, stringCollection);
```

Advanced usage of wildcards

```
interface Sink<T> {
    flush(T t);
}

public static <T> T writeAll(Collection<T> coll, Sink<T> snk) {
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}
...
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // Illegal call.


public static <T> T writeAll(Collection<? extends T>, Sink<T>) {...}
...
// Call is OK, but wrong return type.
String str = writeAll(cs, s);


public static <T> T writeAll(Collection<T> coll, Sink<? super T> snk) {
    ...
}
String str = writeAll(cs, s); // Yes!
```

Advanced usage of wildcards

```
interface Comparator<T> {
    int compare(T fst, T snd);
}



public static <T extends Comparable<T>>
        T max(Collection<T> coll)



class Foo implements Comparable<Object> {
    ...
}



Collection<Foo> cf = ... ;
Collections.max(cf); // Should work.

public static <T extends Comparable<? super T>>
        T max(Collection<T> coll)



// ACTUAL JDK VERSION:

public static <T extends Object & Comparable<? super T>>
        T max(Collection<? extends T> coll)
```