# List

## Adding and Iterating

**import** **java.io.\***;

**import** **java.util.\***;

**import** **java.util.function.\*** ;

**class** **JavaMain** {

**public** **static** void main(String[] args)

{

Io.initializeIO() ;

List<Integer> list = **new** ArrayList<Integer>() ;

list.add(10) ;

list.add(11) ;

list.add(12) ;

*// Iterate in single direction*

Iterator<Integer> itr = list.iterator() ;

**while**(itr.hasNext()){

Integer num = itr.next() ;

Io.output.println(num) ;

}

*// Iterate in both directions*

ListIterator<Integer> litr = list.listIterator(list.size()) ;

**while**(litr.hasPrevious()){

Integer num = litr.previous() ;

Io.output.println(num) ;

}

Io.output.println(list) ;

Io.closeIO() ;

}

}

## Removing

**class** **JavaMain** {

**public** **static** void main(String[] args)

{

Io.initializeIO() ;

List<Integer> list = **new** ArrayList<Integer>() ;

list.add(10) ;

list.add(11) ;

list.add(12) ;

list.add(13) ;

list.add(14) ;

list.add(15) ;

Io.output.println(list) ;

*// This makes a call to remove(int) and removes element 11*

list.remove(1) ;

Io.output.println(list) ;

*// This makes a call to remove(Object) and removes element 14*

list.remove(Integer.valueOf(14));

Io.output.println(list) ;

Io.closeIO() ;

}

}

## Updating

**class** **JavaMain** {

**public** **static** void main(String[] args)

{

Io.initializeIO() ;

List<Integer> list = **new** ArrayList<Integer>() ;

list.add(10) ;

list.add(11) ;

list.add(12) ;

list.add(13) ;

list.add(14) ;

list.add(15) ;

Io.output.println(list) ;

*// Replacing element at the index 1 with 10 using method set()*

list.set(1 , 10 ) ;

Io.output.println(list) ;

Io.output.println() ;

Io.closeIO() ;

}

}

O/p :

[10, 11, 12, 13, 14, 15]

[10, 10, 12, 13, 14, 15]

## Remove Duplicates

**class** **JavaMain** {

**public** **static** void main(String[] args)

{

Io.initializeIO() ;

List<Integer> list = Arrays.asList(1,2,3,1,1,3,3,2,4,3,5) ;

Io.output.println(list) ;

List<Integer> uniqueList = **new** ArrayList<>();

**for**(Integer num : list ){

**if**(!uniqueList.contains(num)){

uniqueList.add(num) ;

}

}

Io.output.println(uniqueList) ;

Io.output.println() ;

Io.closeIO() ;

}

}

O/p :

[1, 2, 3, 1, 1, 3, 3, 2, 4, 3, 5]

[1, 2, 3, 4, 5]

# Internal working for HashMap in java 8

**Bucket Array:** HashMap internally maintains an array, also called a "bucket array," to store key-value pairs.

Each element of the bucket array represents an individual LinkedList. The LinkedList is used to handle collisions, i.e., when multiple keys have the same hash code and need to be stored in the same bucket.

That means there would be as many linked lists as there are buckets. Initially, it has a bucket size of 16 which grows to 32 when the number of entries on the map crosses the 75%. (That means after inserting in 12 buckets bucket size becomes 32) (0.75 is the load factor)

**Node Class:**

HashMap consists of a Node class that represents a key-value pair. The Node class contains the following objects:

int hash: The hash value of the key.

K key: The key object.

V value: The value object.

Node next: A reference to the next Node in the LinkedList.

**Put Operation :**

Internally HashMap uses a hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket where the new entry can be added.

HashMap uses multiple buckets and each bucket points to a Singly Linked List where the entries (nodes) are stored.

Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket(singly linked list).

If there already exists a key with the same hashCode, then the equals() method is used on the keys. If the equals method returns true, that means there is already a node with the same key and hence the value against that key is overwritten in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.

If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added to the bucket found.

**Get Operation :**

Find the hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket.

Traverse the LinkedList at the calculated index to find the Node with the matching key.

If the Node is found, return its value. If the Node is not found, return null.

**Java 8 Enhancements:**

Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a thresold is crossed (static final int TREEIFY\_THRESHOLD = 8;)

The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup.

Also note that Ordinary binary search trees have pathological cases where they become O(n) [basically BST becomes skewed], but red-black/AVL trees are specifically designed to prevent these cases.

But it seems like with this red-black/AVL tree scheme, even if all the items hashed into the same bucket, we would get O(log(n)) lookup in worst of worst scenario.

# Fail First and Fail Safe Iterators

These iterators immediately throw a ConcurrentModificationException if there is a structural modification (adding, removing, or updating elements) in the collection while a thread is iterating over it.

They operate directly on the original collection and are called "fail-fast" because they quickly expose any concurrent modifications.

Examples of fail-fast iterators are Iterator on ArrayList and HashMap

**Internal working :**

Every fail fast collection has a modCount field, to represent how many times the collection has changed/modified.

So at every modification of this collection we increment the modCount value. For example the modCount is incremented in below cases:

1. When one or more elements are removed.

2. When one or more elements are added.

3. When the collection is replaced with other collection.

4. When the collection is sorted.

So everytime there is some change in the collection structure, the mod count is incremented.

This code is used in most of the iterator methods e.g.

1. next()

2. remove()

3. add()

So if we make any changes to the collection, the modCount will change, and expectedModCount will not be hence equal to the modCount. Then if we use any of the above methods of iterator, the ConcurrentModificationException will be thrown.

**Imporant Note :**

If we remove/add the element using the remove() or add() of iterator instead of collection, then in that case no exception will opccur. It is because the remove/add methods of iterators call the remove/add method of collection internally, and also it reasigns the expectedModCount to new modCount value.

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

al.add(1) ; al.add(2) ; al.add(3) ; al.add(4) ; al.add(5) ;

Iterator<Integer> itr = al.iterator();

**while** (itr.hasNext()) {

**if** (itr.next() == 2) {

*// will not throw Exception*

itr.remove();

}

}

System.out.println(al);

itr = al.iterator();

**while** (itr.hasNext()) {

**if** (itr.next() == 3) {

*// will throw Exception on*

*// next call of next() method*

al.remove(3);

}

}

**Fail-Safe Iterators :**

These iterators do not throw any exceptions if a collection is structurally modified while iterating over it. Instead, they operate on a clone or a separate copy of the collection, which means they are not affected by the modifications in the original collection. This approach ensures that the original collection remains unaffected by the modifications made during iteration. Examples of fail-safe iterators are Iterator on *CopyOnWriteArrayList* and *ConcurrentHashMap*

List<Integer> al

= **new** CopyOnWriteArrayList<Integer>() ;

*// (new Integer[] { 1, 2, 3, 5, 8 });*

al.add(1) ; al.add(2) ; al.add(3) ; al.add(4) ; al.add(5) ;

Iterator<Integer> itr = al.iterator();

System.out.println(al);

itr = al.iterator();

**while** (itr.hasNext()) {

**if** (itr.next() == 3) {

al.remove(Integer.valueOf(3));

}

}

System.out.println(al);