CSE222

HW5

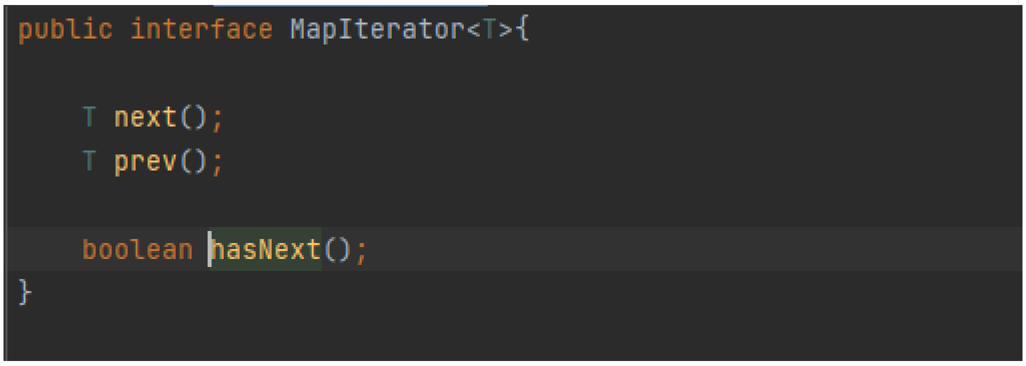
Oğulcan Kalafatoğlu

1801042613

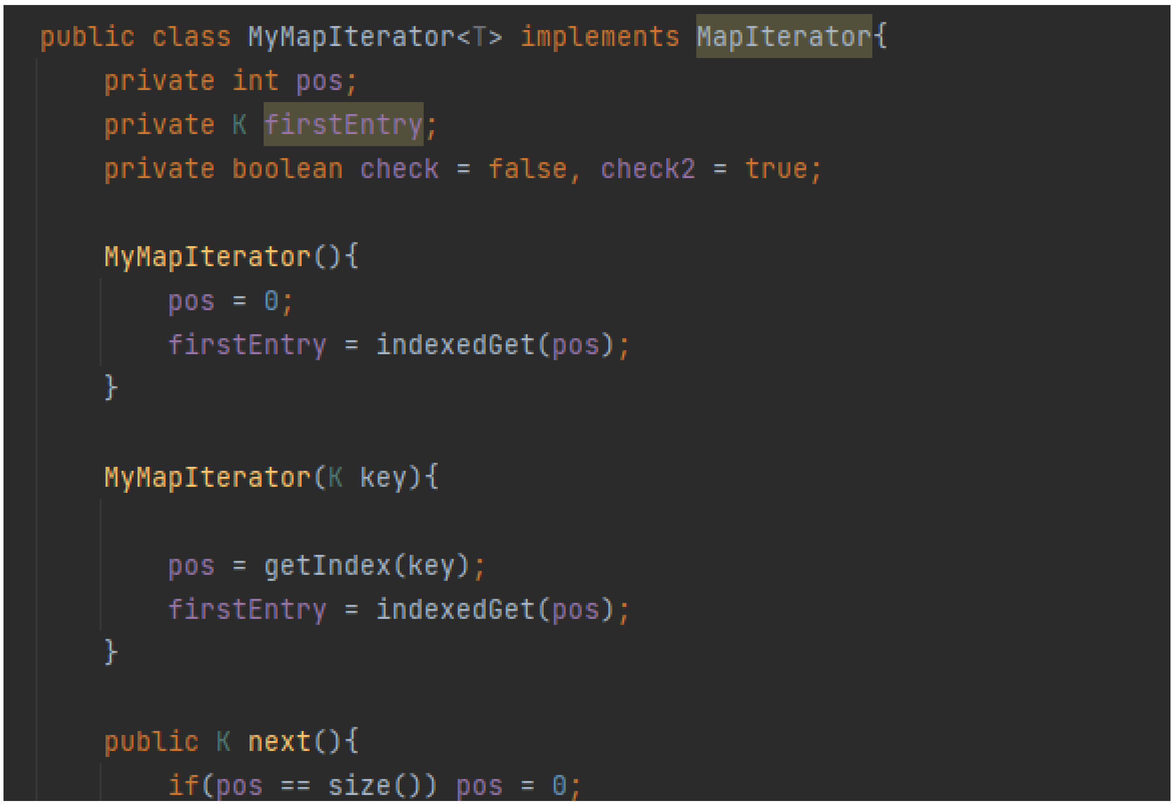
1.Problem Solutions Approach

Part1:

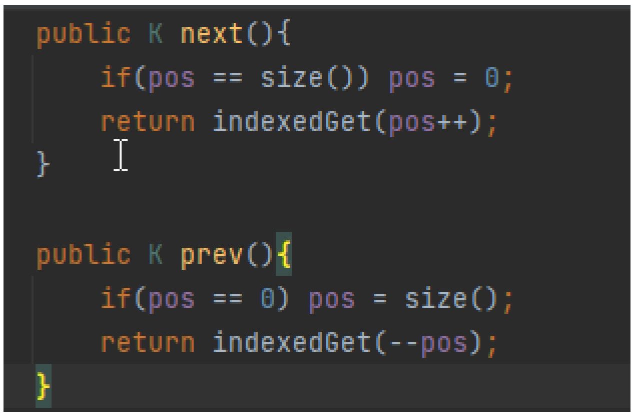
We were asked to write a custom iterator class for HashMap, I extended java hashMap and implemented my generic interface iterator in it.



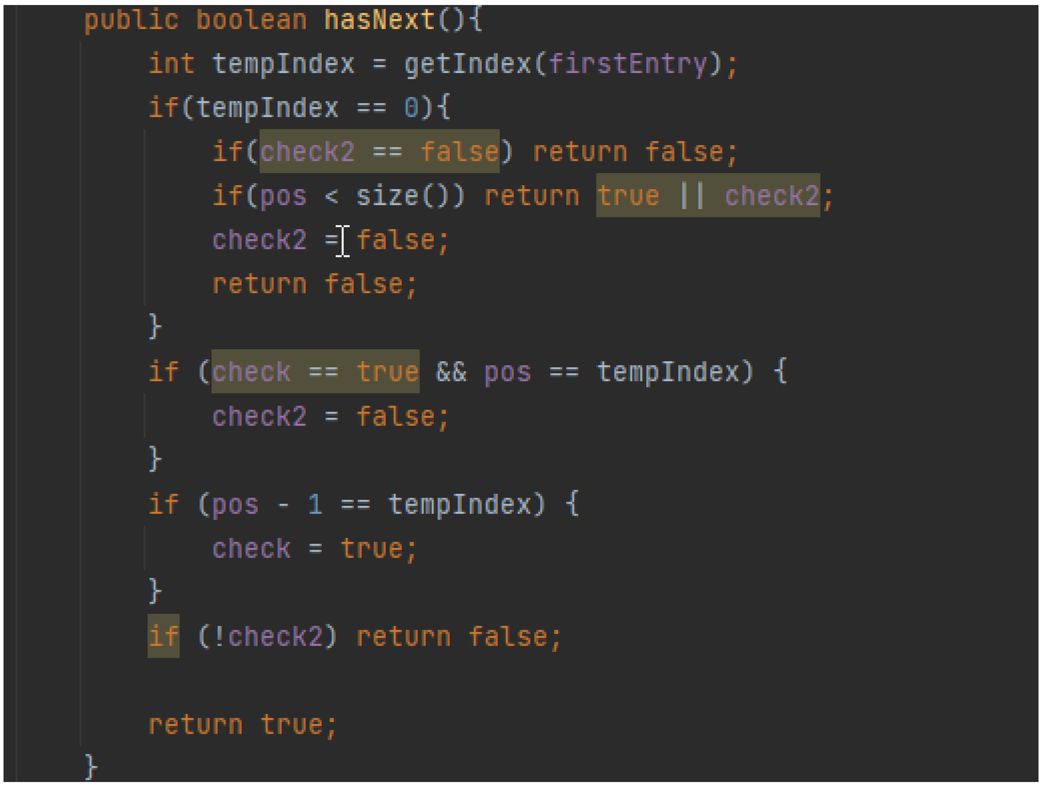
```java
public interface MapIterator<T>{

    T next();
    T prev();

    boolean hasNext();
}
```



```java
public class MyMapIterator<T> implements MapIterator{
    private int pos;
    private K firstEntry;
    private boolean check = false, check2 = true;

    MyMapIterator(){
        pos = 0;
        firstEntry = indexedGet(pos);
    }

    MyMapIterator(K key){

        pos = getIndex(key);
        firstEntry = indexedGet(pos);
    }

    public K next(){
        if(pos == size()) pos = 0;
```



```java
public K next(){
    if(pos == size()) pos = 0;
    return indexedGet(pos++);
}

public K prev(){
    if(pos == 0) pos = size();
    return indexedGet(--pos);
}
```

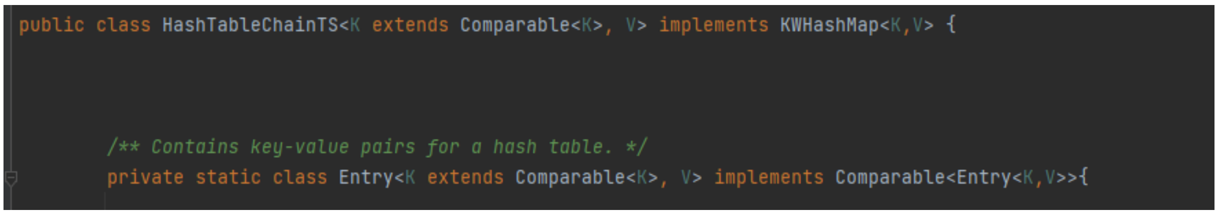


```java
    public boolean hasNext(){
        int tempIndex = getIndex(firstEntry);
        if(tempIndex == 0){
            if(check2 == false) return false;
            if(pos < size()) return true || check2;
            check2 = false;
            return false;
        }
        if (check == true && pos == tempIndex) {
            check2 = false;
        }
        if (pos - 1 == tempIndex) {
            check = true;
        }
        if (!check2) return false;

        return true;
    }
```

Part2:

For chaining using LinkedList part i implemented it using textbook.

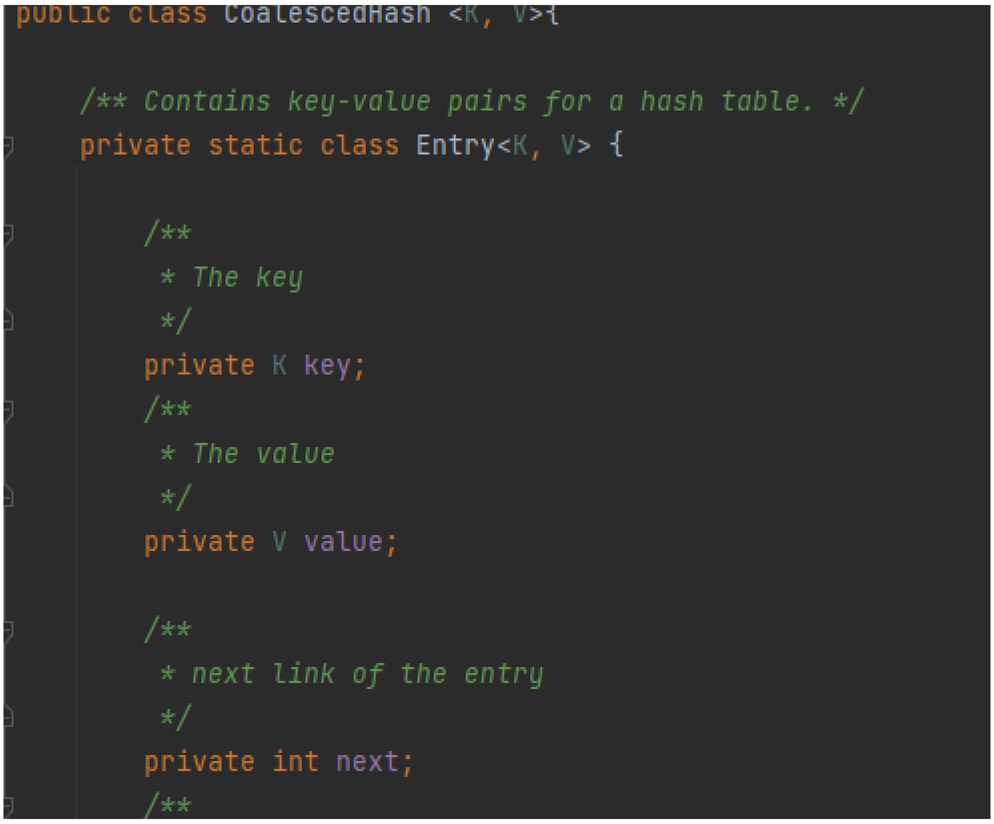
For treeSet part, used treeSet from java, and made Entry class comparable.



```java
public class HashTableChainTS<K extends Comparable<K>, V> implements KWHashMap<K,V> {


        /** Contains key-value pairs for a hash table. */
        private static class Entry<K extends Comparable<K>, V> implements Comparable<Entry<K,V>>{
```

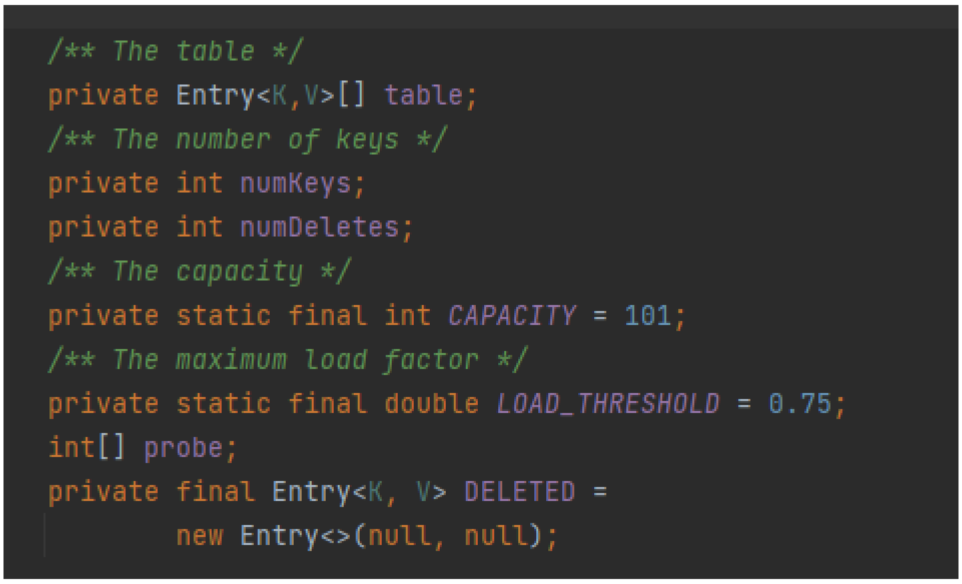
For last part, Implemented it according to homework pdf

I hold next as int inside entry class.



```java
public class CoalescedHash <K, V>{

    /** Contains key-value pairs for a hash table. */
    private static class Entry<K, V> {

        /**
         * The key
         */
        private K key;
        /**
         * The value
         */
        private V value;

        /**
         * next link of the entry
         */
        private int next;
        /**
```

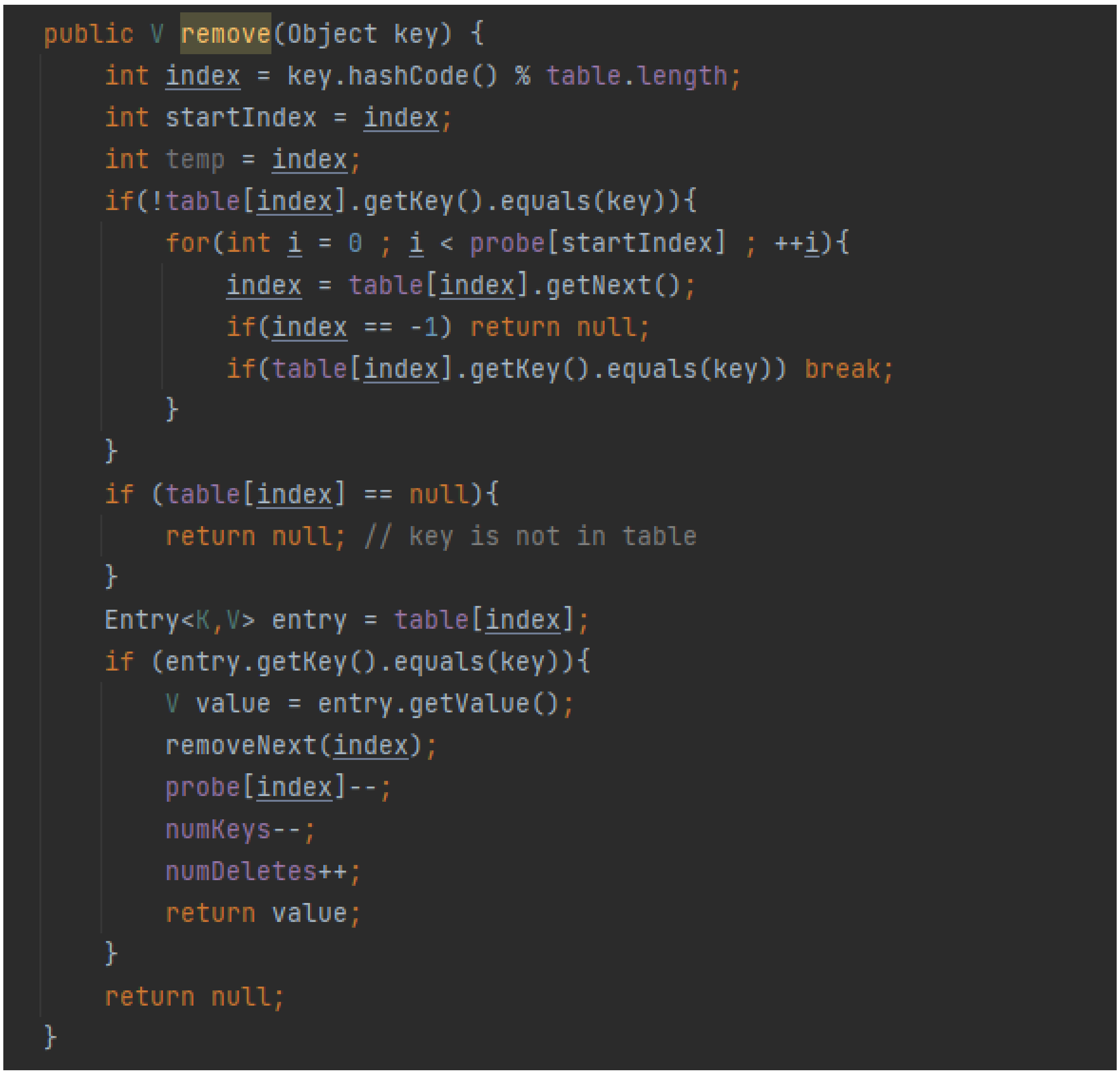
Table is array of Entry, and I hold another integer array sized as same capacity as table, probe array for quadratic probing.



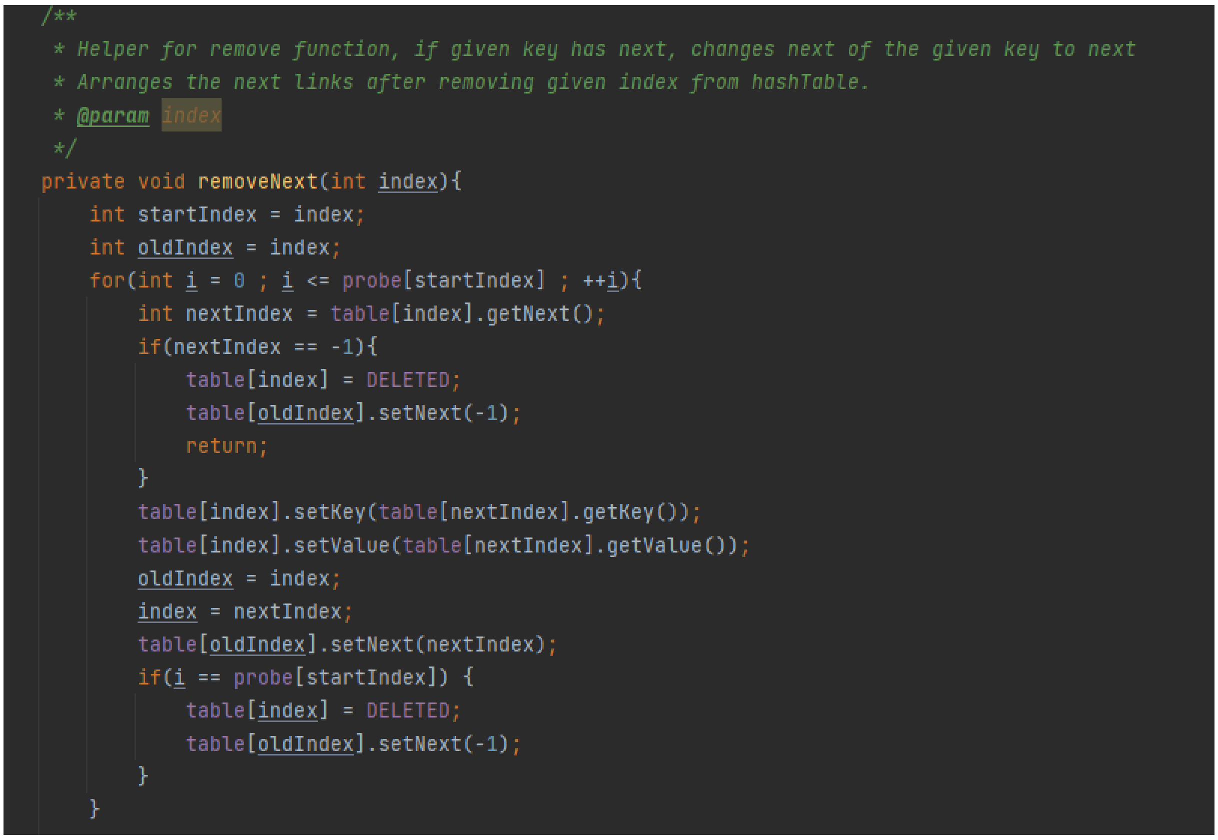
```java
/** The table */
private Entry<K,V>[] table;
/** The number of keys */
private int numKeys;
private int numDeletes;
/** The capacity */
private static final int CAPACITY = 101;
/** The maximum load factor */
private static final double LOAD_THRESHOLD = 0.75;
int[] probe;
private final Entry<K, V> DELETED =
        new Entry<>(null, null);
```

Most of the methods were taken from textbook and modified.

For remove method another private method is created managing linking of entries are done there.

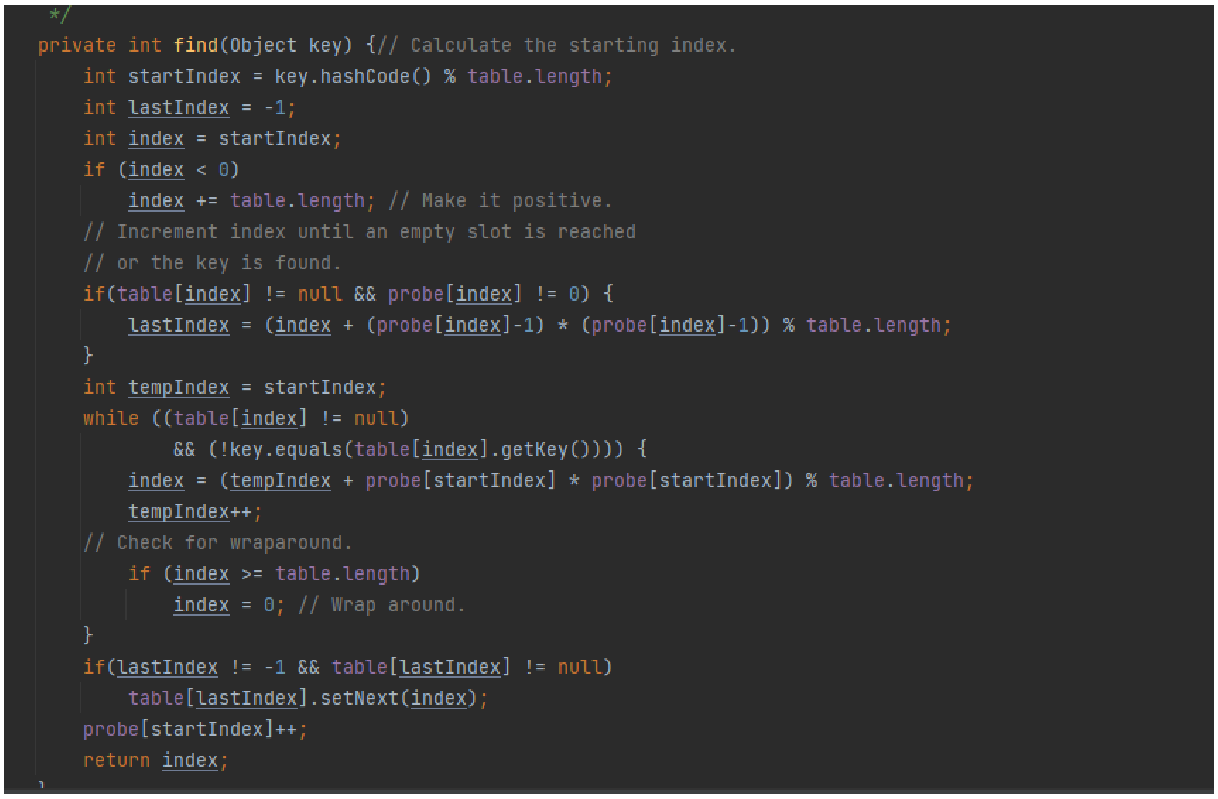


```java
public V remove(Object key) {
    int index = key.hashCode() % table.length;
    int startIndex = index;
    int temp = index;
    if(!table[index].getKey().equals(key)){
        for(int i = 0 ; i < probe[startIndex] ; ++i){
            index = table[index].getNext();
            if(index == -1) return null;
            if(table[index].getKey().equals(key)) break;
        }
    }
    if (table[index] == null){
        return null; // key is not in table
    }
    Entry<K,V> entry = table[index];
    if (entry.getKey().equals(key)){
        V value = entry.getValue();
        removeNext(index);
        probe[index]--;
        numKeys--;
        numDeletes++;
        return value;
    }
    return null;
}
```



```java
/**
 * Helper for remove function, if given key has next, changes next of the given key to next
 * Arranges the next links after removing given index from hashTable.
 * @param index
 */
private void removeNext(int index){
    int startIndex = index;
    int oldIndex = index;
    for(int i = 0 ; i <= probe[startIndex] ; ++i){
        int nextIndex = table[index].getNext();
        if(nextIndex == -1){
            table[index] = DELETED;
            table[oldIndex].setNext(-1);
            return;
        }
        table[index].setKey(table[nextIndex].getKey());
        table[index].setValue(table[nextIndex].getValue());
        oldIndex = index;
        index = nextIndex;
        table[oldIndex].setNext(nextIndex);
        if(i == probe[startIndex]) {
            table[index] = DELETED;
            table[oldIndex].setNext(-1);
        }
    }
```

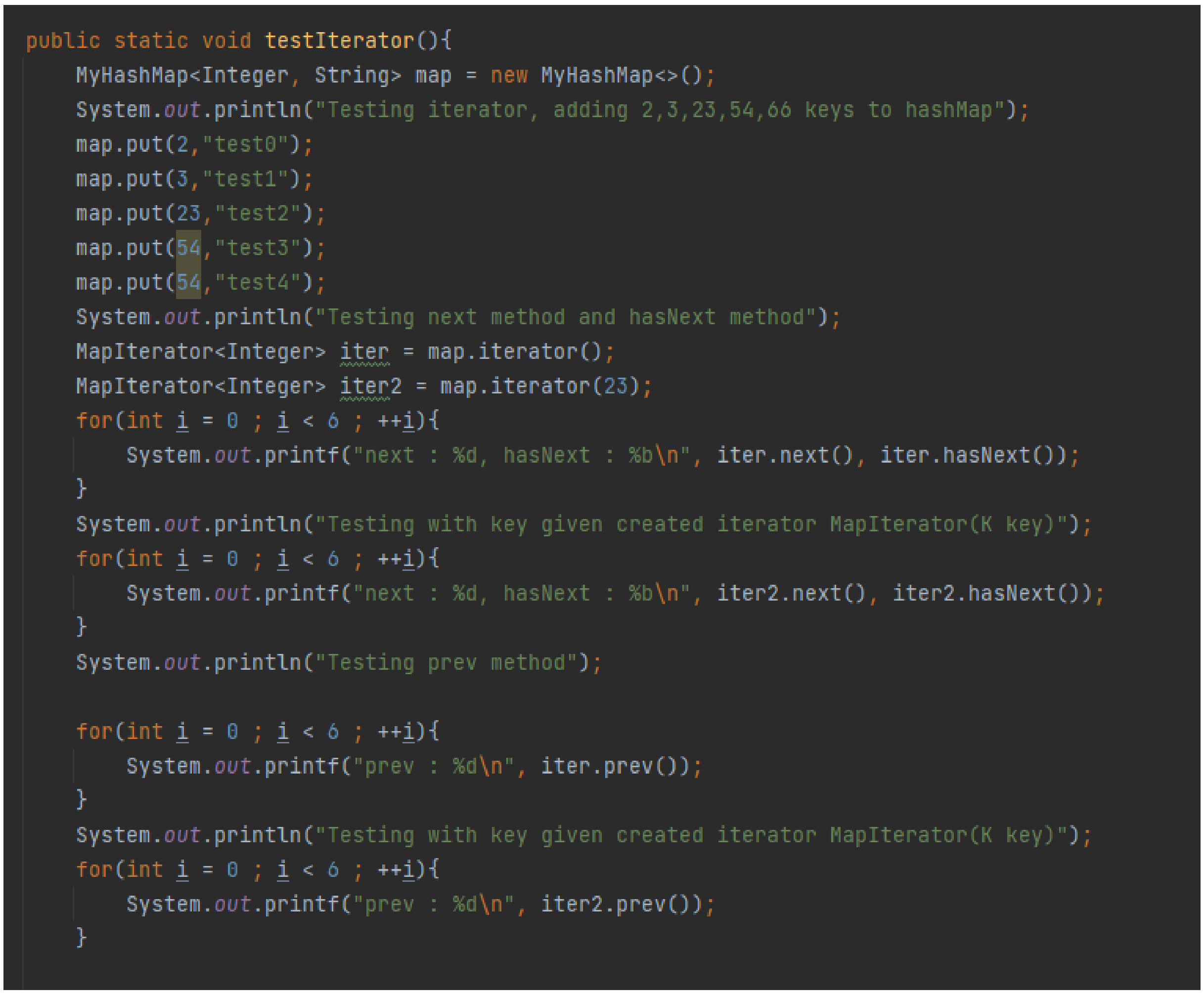
For inserting, find method is changed and indexing is done by quadratic probing.



```java
 */
private int find(Object key) {// Calculate the starting index.
    int startIndex = key.hashCode() % table.length;
    int lastIndex = -1;
    int index = startIndex;
    if (index < 0)
        index += table.length; // Make it positive.
    // Increment index until an empty slot is reached
    // or the key is found.
    if(table[index] != null && probe[index] != 0) {
        lastIndex = (index + (probe[index]-1) * (probe[index]-1)) % table.length;
    }
    int tempIndex = startIndex;
    while ((table[index] != null)
            && (!key.equals(table[index].getKey()))) {
        index = (tempIndex + probe[startIndex] * probe[startIndex]) % table.length;
        tempIndex++;
    // Check for wraparound.
        if (index >= table.length)
            index = 0; // Wrap around.
    }
    if(lastIndex != -1 && table[lastIndex] != null)
        table[lastIndex].setNext(index);
    probe[startIndex]++;
    return index;
```

2-Test cases

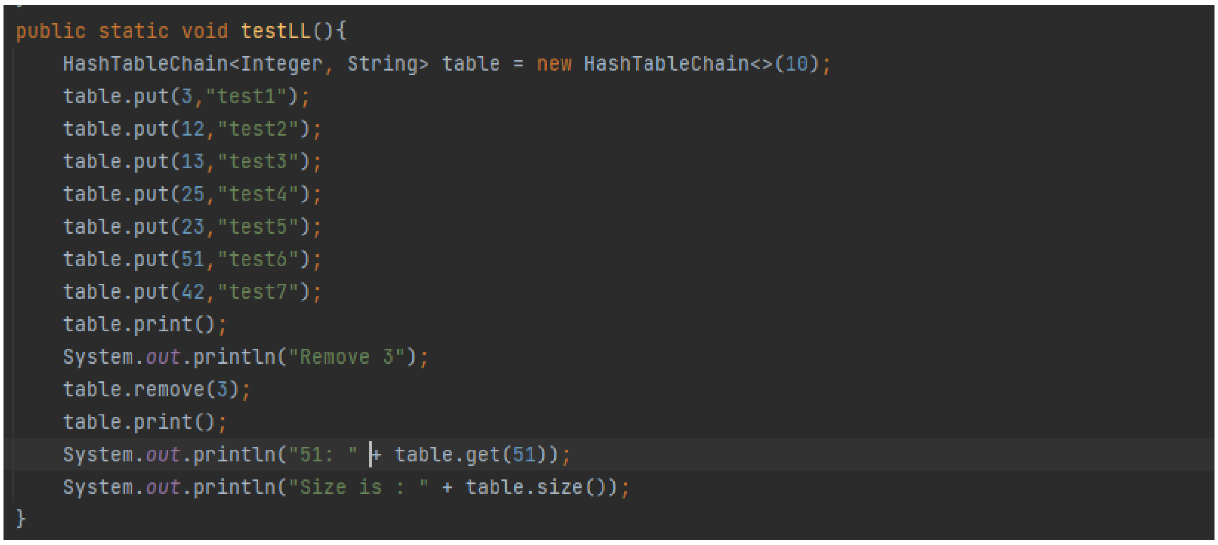
part1



```java
public static void testIterator(){
    MyHashMap<Integer, String> map = new MyHashMap<>();
    System.out.println("Testing iterator, adding 2,3,23,54,66 keys to hashMap");
    map.put(2,"test0");
    map.put(3,"test1");
    map.put(23,"test2");
    map.put(54,"test3");
    map.put(54,"test4");
    System.out.println("Testing next method and hasNext method");
    MapIterator<Integer> iter = map.iterator();
    MapIterator<Integer> iter2 = map.iterator(23);
    for(int i = 0 ; i < 6 ; ++i){
        System.out.printf("next : %d, hasNext : %b\n", iter.next(), iter.hasNext());
    }
    System.out.println("Testing with key given created iterator MapIterator(K key)");
    for(int i = 0 ; i < 6 ; ++i){
        System.out.printf("next : %d, hasNext : %b\n", iter2.next(), iter2.hasNext());
    }
    System.out.println("Testing prev method");

    for(int i = 0 ; i < 6 ; ++i){
        System.out.printf("prev : %d\n", iter.prev());
    }
    System.out.println("Testing with key given created iterator MapIterator(K key)");
    for(int i = 0 ; i < 6 ; ++i){
        System.out.printf("prev : %d\n", iter2.prev());
    }
```

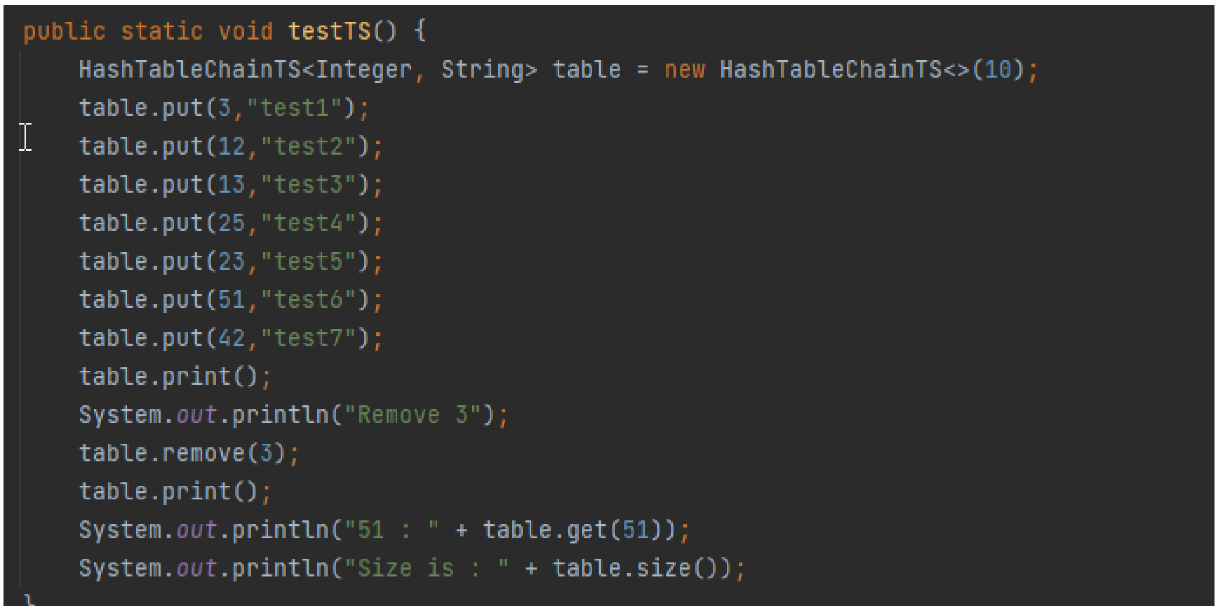
part2

Chaining with linkedList test cases



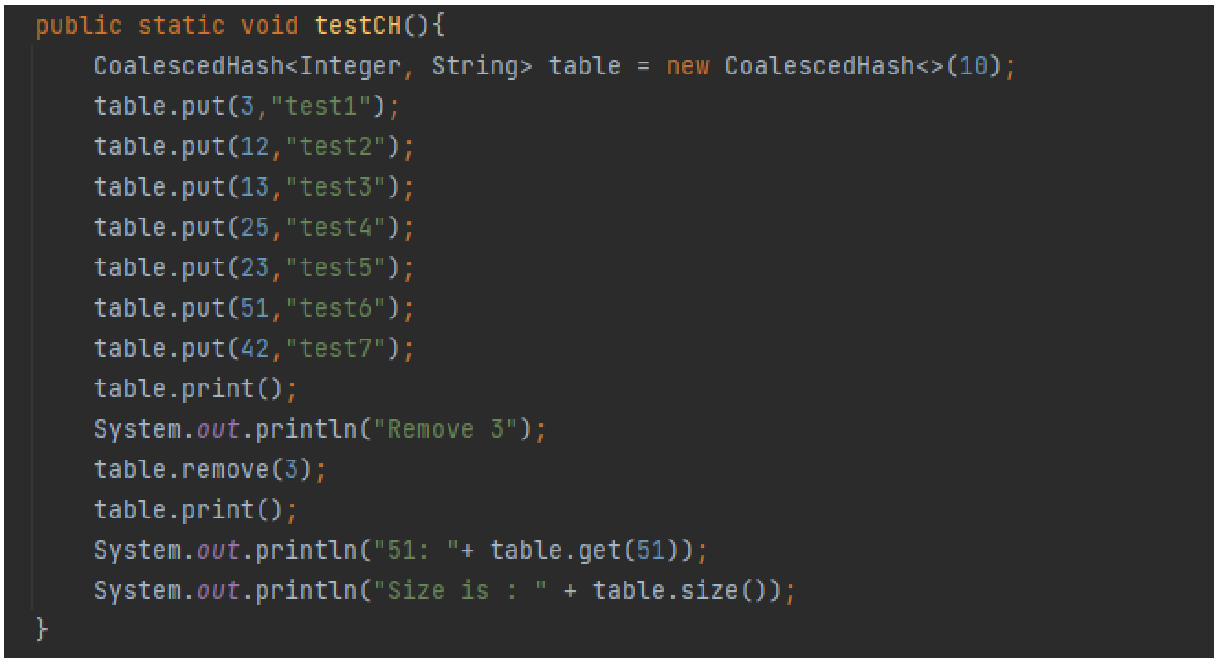
```java
public static void testLL(){
    HashTableChain<Integer, String> table = new HashTableChain<>(10);
    table.put(3,"test1");
    table.put(12,"test2");
    table.put(13,"test3");
    table.put(25,"test4");
    table.put(23,"test5");
    table.put(51,"test6");
    table.put(42,"test7");
    table.print();
    System.out.println("Remove 3");
    table.remove(3);
    table.print();
    System.out.println("51: " + table.get(51));
    System.out.println("Size is : " + table.size());
}
```

Chaining with TreeSet test cases



```java
public static void testTS() {
    HashTableChainTS<Integer, String> table = new HashTableChainTS<>(10);
    table.put(3,"test1");
    table.put(12,"test2");
    table.put(13,"test3");
    table.put(25,"test4");
    table.put(23,"test5");
    table.put(51,"test6");
    table.put(42,"test7");
    table.print();
    System.out.println("Remove 3");
    table.remove(3);
    table.print();
    System.out.println("51 : " + table.get(51));
    System.out.println("Size is : " + table.size());
```

CoalescedHash test cases.

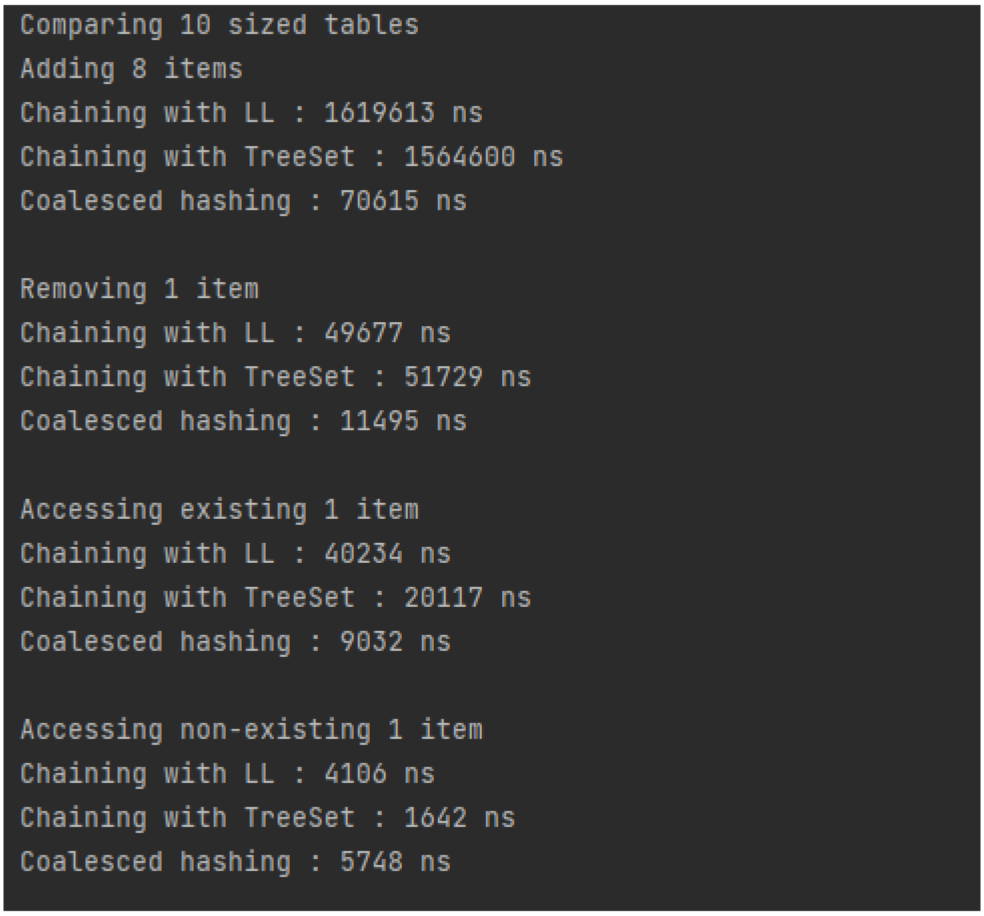


```java
public static void testCH(){
    CoalescedHash<Integer, String> table = new CoalescedHash<>(10);
    table.put(3,"test1");
    table.put(12,"test2");
    table.put(13,"test3");
    table.put(25,"test4");
    table.put(23,"test5");
    table.put(51,"test6");
    table.put(42,"test7");
    table.print();
    System.out.println("Remove 3");
    table.remove(3);
    table.print();
    System.out.println("51: "+ table.get(51));
    System.out.println("Size is : " + table.size());
}
```

3- Running Commands and Results

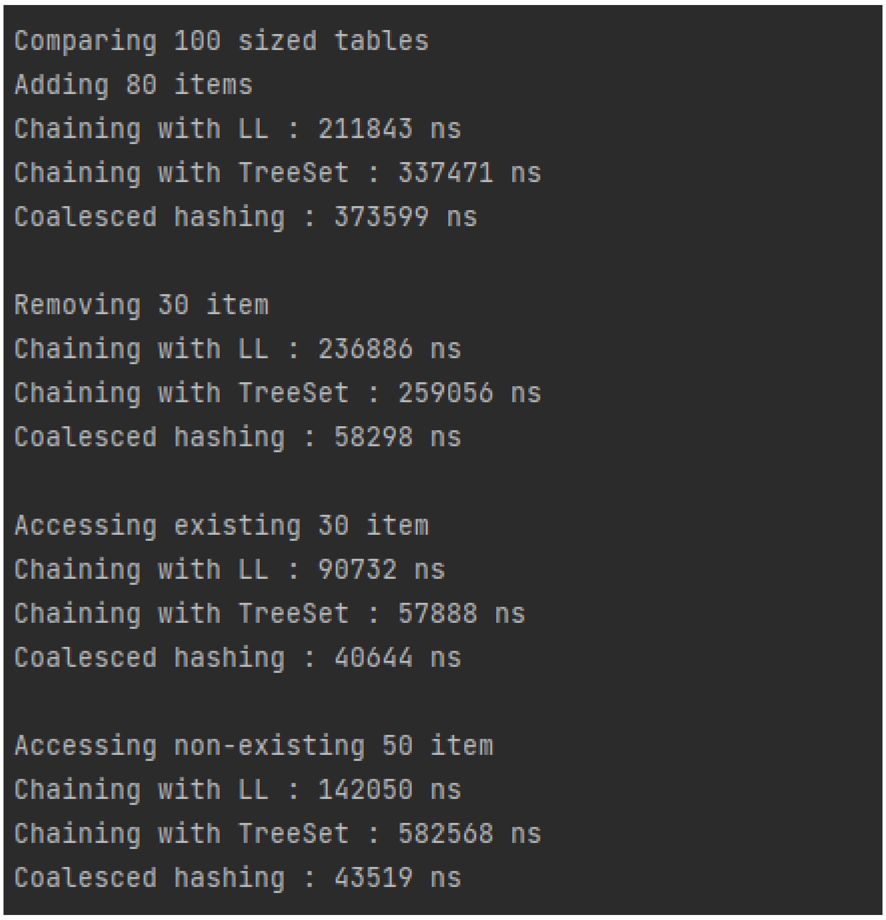
Main – Performance comparision

10 sized tables



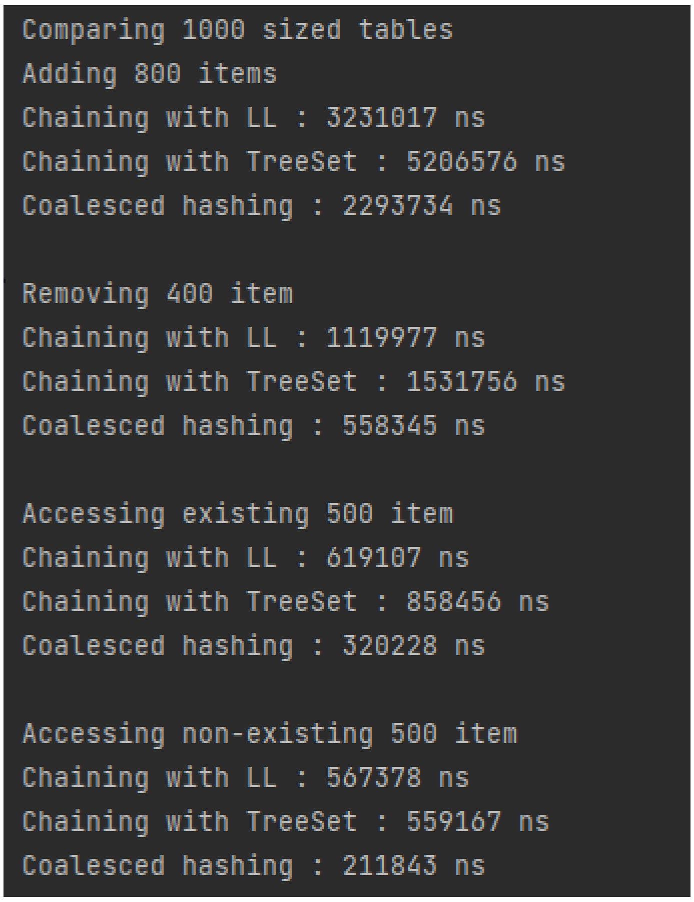
```
Comparing 10 sized tables
Adding 8 items
Chaining with LL : 1619613 ns
Chaining with TreeSet : 1564600 ns
Coalesced hashing : 70615 ns

Removing 1 item
Chaining with LL : 49677 ns
Chaining with TreeSet : 51729 ns
Coalesced hashing : 11495 ns

Accessing existing 1 item
Chaining with LL : 40234 ns
Chaining with TreeSet : 20117 ns
Coalesced hashing : 9032 ns

Accessing non-existing 1 item
Chaining with LL : 4106 ns
Chaining with TreeSet : 1642 ns
Coalesced hashing : 5748 ns
```

100 sized tables



```
Comparing 100 sized tables
Adding 80 items
Chaining with LL : 211843 ns
Chaining with TreeSet : 337471 ns
Coalesced hashing : 373599 ns

Removing 30 item
Chaining with LL : 236886 ns
Chaining with TreeSet : 259056 ns
Coalesced hashing : 58298 ns

Accessing existing 30 item
Chaining with LL : 90732 ns
Chaining with TreeSet : 57888 ns
Coalesced hashing : 40644 ns

Accessing non-existing 50 item
Chaining with LL : 142050 ns
Chaining with TreeSet : 582568 ns
Coalesced hashing : 43519 ns
```

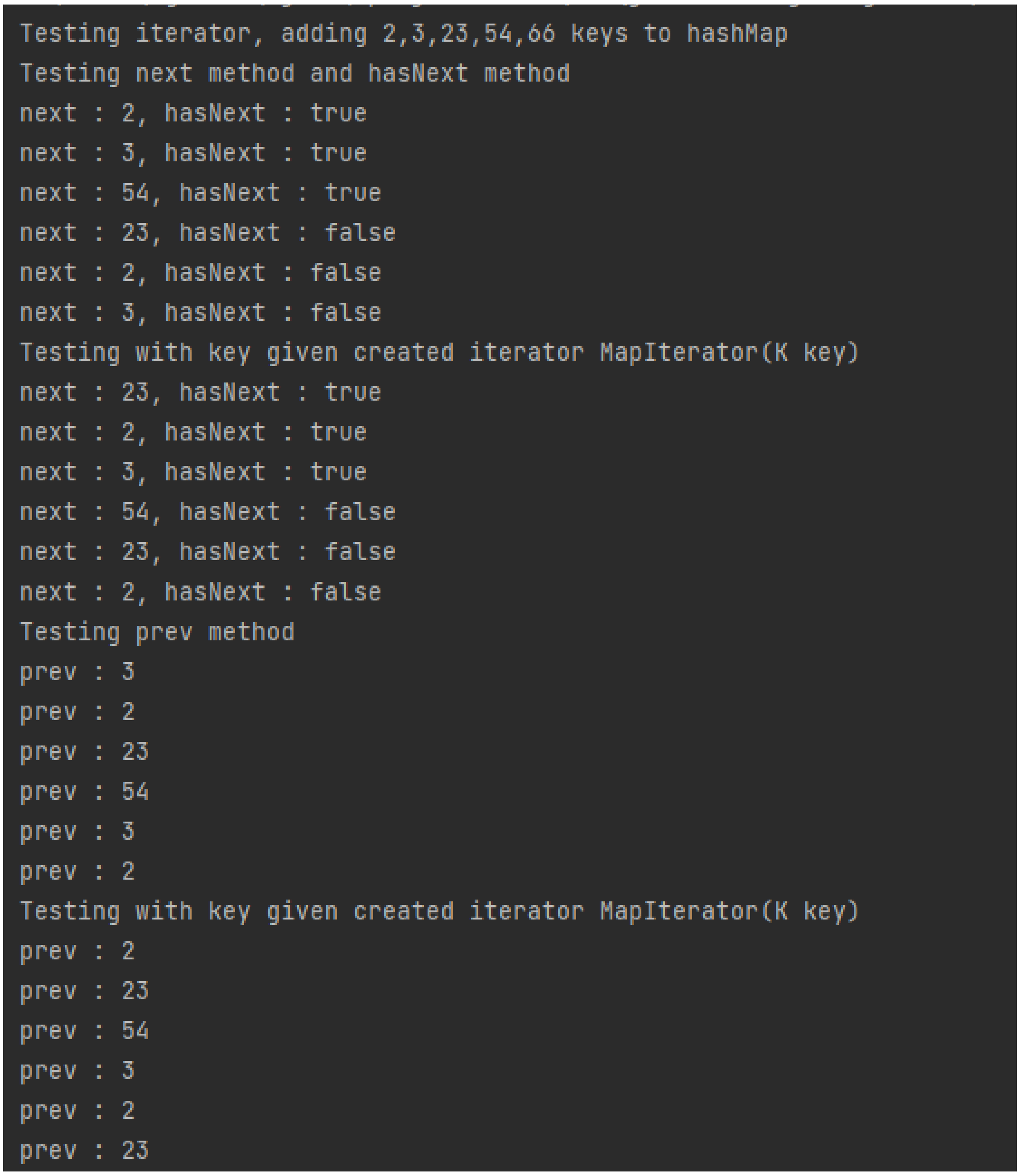
1000 sized tables



```
Comparing 1000 sized tables
Adding 800 items
Chaining with LL : 3231017 ns
Chaining with TreeSet : 5206576 ns
Coalesced hashing : 2293734 ns

Removing 400 item
Chaining with LL : 1119977 ns
Chaining with TreeSet : 1531756 ns
Coalesced hashing : 558345 ns

Accessing existing 500 item
Chaining with LL : 619107 ns
Chaining with TreeSet : 858456 ns
Coalesced hashing : 320228 ns

Accessing non-existing 500 item
Chaining with LL : 567378 ns
Chaining with TreeSet : 559167 ns
Coalesced hashing : 211843 ns
```

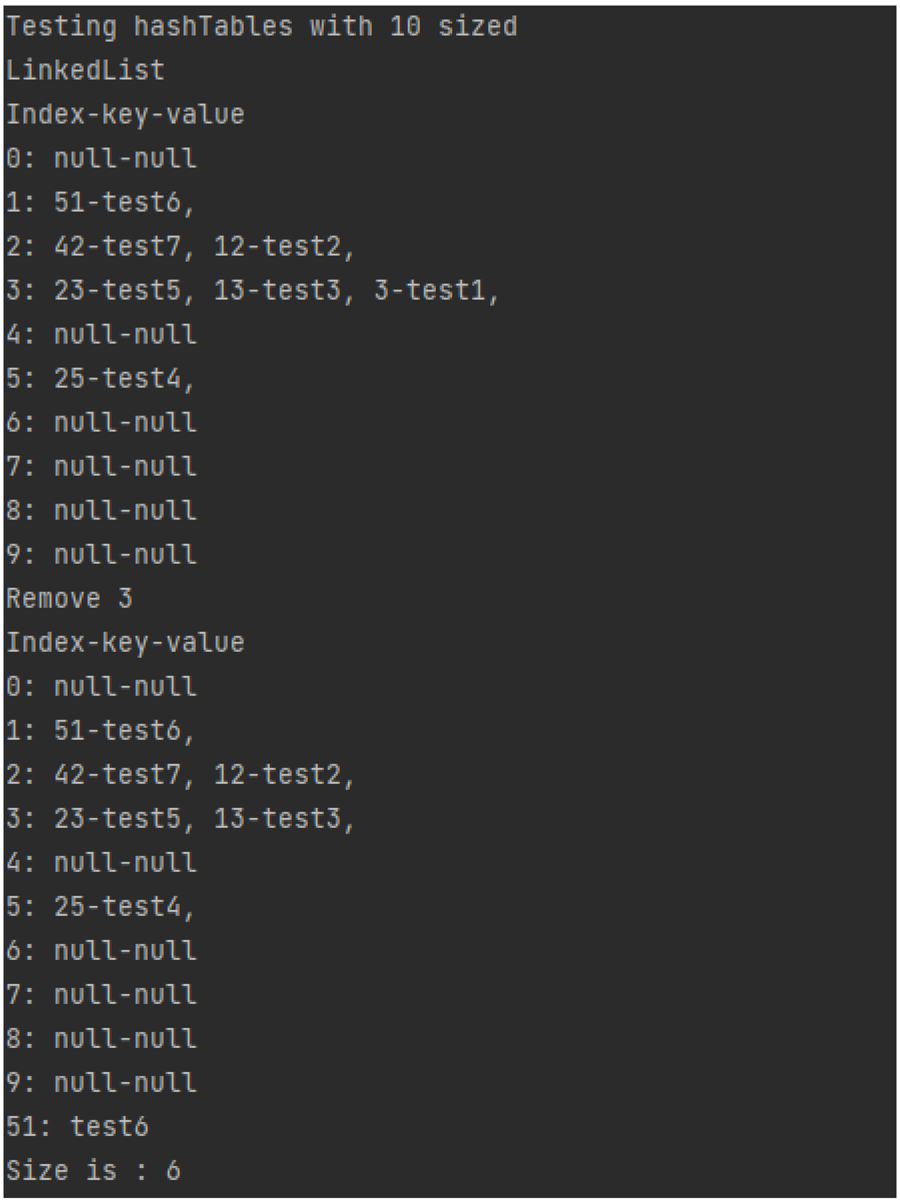
Test case results:

Iterator test-cases



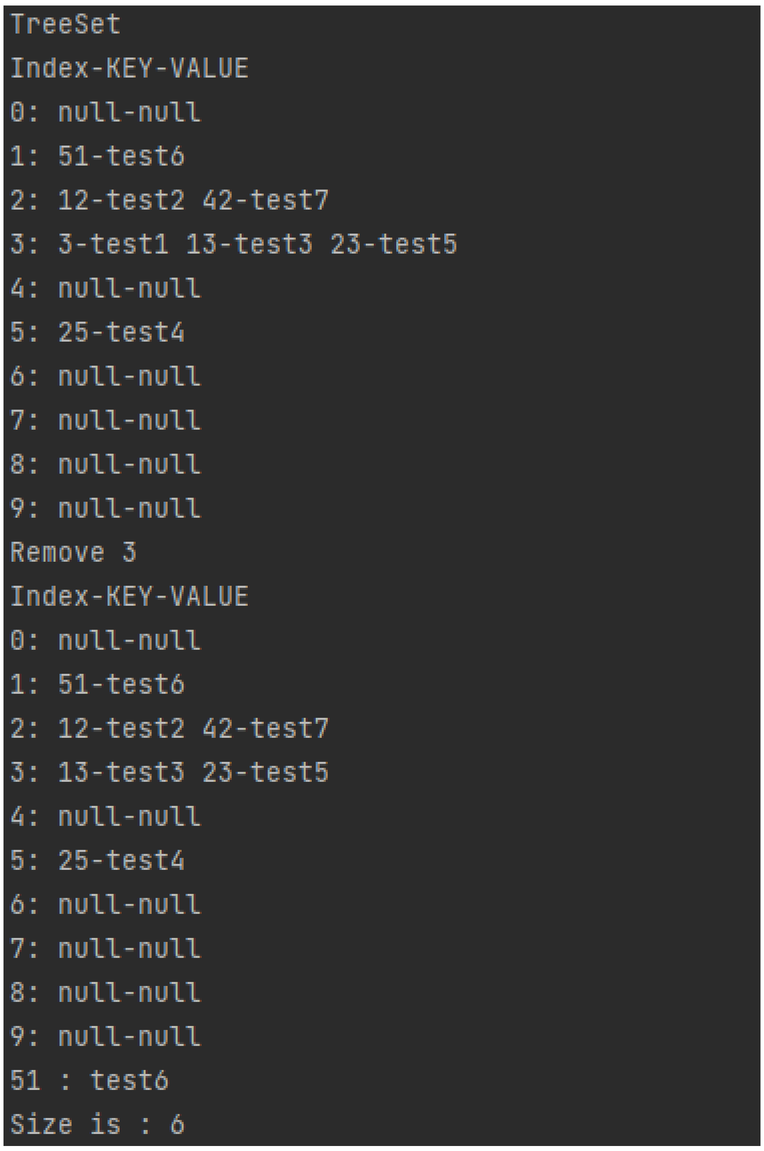
```
Testing iterator, adding 2,3,23,54,66 keys to hashMap
Testing next method and hasNext method
next : 2, hasNext : true
next : 3, hasNext : true
next : 54, hasNext : true
next : 23, hasNext : false
next : 2, hasNext : false
next : 3, hasNext : false
Testing with key given created iterator MapIterator(K key)
next : 23, hasNext : true
next : 2, hasNext : true
next : 3, hasNext : true
next : 54, hasNext : false
next : 23, hasNext : false
next : 2, hasNext : false
Testing prev method
prev : 3
prev : 2
prev : 23
prev : 54
prev : 3
prev : 2
Testing with key given created iterator MapIterator(K key)
prev : 2
prev : 23
prev : 54
prev : 3
prev : 2
prev : 23
```

HashTable chaining done with LinkedList(10 sized)



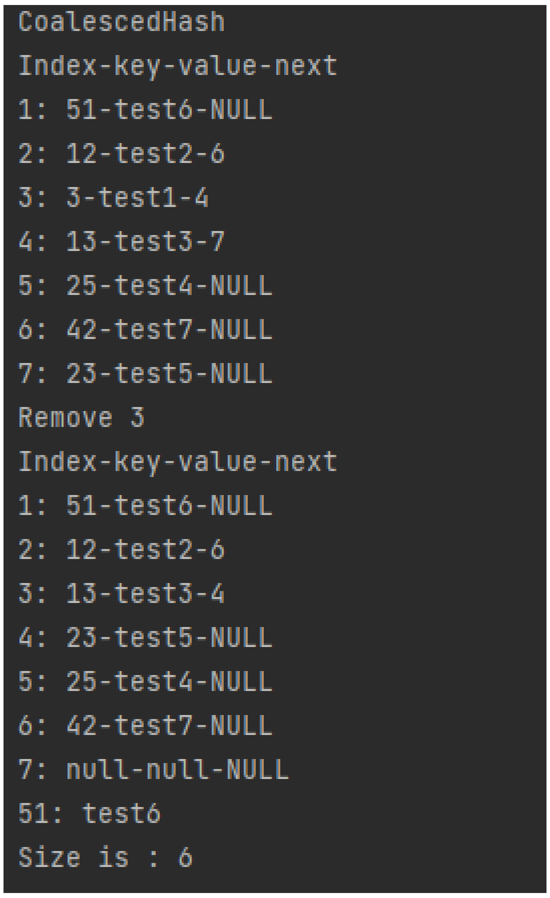
```
Testing hashTables with 10 sized
LinkedList
Index-key-value
0: null-null
1: 51-test6,
2: 42-test7, 12-test2,
3: 23-test5, 13-test3, 3-test1,
4: null-null
5: 25-test4,
6: null-null
7: null-null
8: null-null
9: null-null
Remove 3
Index-key-value
0: null-null
1: 51-test6,
2: 42-test7, 12-test2,
3: 23-test5, 13-test3,
4: null-null
5: 25-test4,
6: null-null
7: null-null
8: null-null
9: null-null
51: test6
Size is : 6
```

TreeSet test cases result



```
TreeSet
Index-KEY-VALUE
0: null-null
1: 51-test6
2: 12-test2 42-test7
3: 3-test1 13-test3 23-test5
4: null-null
5: 25-test4
6: null-null
7: null-null
8: null-null
9: null-null
Remove 3
Index-KEY-VALUE
0: null-null
1: 51-test6
2: 12-test2 42-test7
3: 13-test3 23-test5
4: null-null
5: 25-test4
6: null-null
7: null-null
8: null-null
9: null-null
51 : test6
Size is : 6
```

CoalescedHash test case results



```
CoalescedHash
Index-key-value-next
1: 51-test6-NULL
2: 12-test2-6
3: 3-test1-4
4: 13-test3-7
5: 25-test4-NULL
6: 42-test7-NULL
7: 23-test5-NULL
Remove 3
Index-key-value-next
1: 51-test6-NULL
2: 12-test2-6
3: 13-test3-4
4: 23-test5-NULL
5: 25-test4-NULL
6: 42-test7-NULL
7: null-null-NULL
51: test6
Size is : 6
```