GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 5 Report

CANBERK ARICI

171044062

PROBLEM SOLUTIONS APPROACH

In part 1, I got challenged very much because I could not find how to get hashmap which is created in main then I tried several ways to solve this issue and finally by extending HashMap, I was able to iterate through hash map's keys. I designed methods according to pdf, next method returns next key in map and if there is no not-iterated key in the map then it returns first key. Prev method returns previous key in map and It returns the last key when the iterator is at the first key, hasNext method returns True if there are still not-iterated key/s in the Map, otherwise returns False and if a key is given to MapIterator constructor then iterator will start from the given key otherwise it will start from the first key.

In part 2, firstly I implemented hash table with linked list. I chained items on the same table slot by using linked list, every item is an entry so I have an inner Entry class. I have specified capacity and load threshold in this class. In outer class, get method returns value of key if found index of given key is not null, put method creates a linked list and creates an entry by using given key and value then adds this entry to linked list if found index according to key in hash table is null then checks if load is more than load threshold and if yes then rehashs, if found index according to key in hash table is not null then finds entry and changes its value to given value then returns old value, if cannot find such an entry then it creates a new entry by using given key and value and adds it to the linked list at found index then increases number of keys and checks if load is more than load threshold and if yes then rehashs then returns value of new entry. Remove method finds index according to given key and checks whether found index is null in hash table or not, if null then returns null otherwise finds entry that has same key with given key then removes it and returns its value, if cannot find then returns null. Size method iterates through hash tables' indexes and linked lists in indexes then counts entries and returns size. Show method shows elements in the same slot, it determines slot by using given key. Finally isEmpty method returns whether table is empty or not. Hash table implementation with treeset is kind of similar with linked list but I had to override compareTo method in Entry class because treeset uses it and I overrode equals method and hashCode method to specialize them for Entry class. There is not much difference between implementations with linked list and treeset.

In hash table implementation by using the Coalesced hashing technique, I determined load threshold as 0.75 and used an entry named DELETED to indicate deleted entries to prevent unwanted results of algorithm while searching etc. I used next value in entry to link the colliding elements to each other, next value is set to null as default, it is become to index of entry when collision happens. Show method shows information of entries in table, get method uses quadratic probing to find key and then its value if found index is not null, put method searches for entry according to found index by using given key, I link collided entries by keeping index which entry tries to occupy and entry whose next is null in indexes which entry tries to occupy and lastly occupied index then make occupied index, next of last tried index and entry whose next is null in this index, checking for exceeding load

threshold and changing existent key’s value etc are the same with other implementations of hash table. Rehash method enlarges size by 2 +1. In remove method, I do deletion of a key as in pdf, by linking its next entry to the entry that points the deleted key by replacing deleted entry by the next entry, if next of key which is wanted to be deleted is not null, I change this entry’s information with its next entry’s information then slide array elements to left from next entry to last entry and make last entry null otherwise I slide array elements to left from wanted entry to last entry and make last entry null, finally I return value of deleted key. Size method returns number of keys and isEmpty method returns whether table is empty or not.

TEST CASES

Test Case #	Test Case Description
1	Put 10000 elements in hashmap and iterate forward (Part 1)

```
System.out.println("Put 10000 elements in hashmap and iterate forward\n");
MapIterator<Integer, Integer> nMap3 = new MapIterator<>( key: 5000);
for(int i=0; i<10000; i++){
    int k = i;
    int v = rand.nextInt( bound: 1000);
    nMap3.put(k,v);
}
MapIterator.MapIter iter3 = nMap3.iterator();
for(int i=0; i<10; i++){
    System.out.format("Next: %d ",iter3.next());
}
System.out.println("\n");
```

```
Put 10000 elements in hashmap and iterate forward


Next: 5000 Next: 5001 Next: 5002 Next: 5003 Next: 5004 Next: 5005 Next: 5006 Next: 5007 Next: 5008 Next: 5009
```

Test Case #	Test Case Description
2	Test to iterate backwards (Part 1)

```
System.out.println("Iterate backwards\n");
for (int i=0;i<15; i++){
    System.out.format("Prev: %d\n",iter3.prev());
}
```

```
Iterate backwards

Prev: 5009
Prev: 5008
Prev: 5007
Prev: 5006
Prev: 5005
Prev: 5004
Prev: 5003
Prev: 5002
Prev: 5001
Prev: 5000
Prev: 4999
Prev: 4998
Prev: 4997
Prev: 4996
Prev: 4995
```

Test Case #	Test Case Description
3	Test to iterate from given key (Part 1)

```
MapIterator<Integer, Integer> nMap2 = new MapIterator<>( key: 2);
nMap2.put(1, 42);
nMap2.put(2, 53);
nMap2.put(3, 1);
nMap2.put(4, 58);
/* Test with giving key */
MapIterator.MapIter iter2 = nMap2.iterator();
/* Iterate forward */
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.println("\n");
```

```
Next: 2
Next: 3
Next: 4
Next: 1
Next: 2
Next: 3
Next: 4
```

Test Case #	Test Case Description
4	Test to iterate backwards with parameter constructor (Part 1)

```
MapIterator<Integer, Integer> nMap2 = new MapIterator<>( key: 2);
nMap2.put(1, 42);
nMap2.put(2, 53);
nMap2.put(3, 1);
nMap2.put(4, 58);
/* Test with giving key */
MapIterator.MapIter iter2 = nMap2.iterator();
/* Iterate forward */
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.format("Next: %d\n",iter2.next());
System.out.println("\n");

/* Iterate backwards */
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.format("Prev: %d\n",iter2.prev());
System.out.println("\n");
```

```
Next: 2
Next: 3
Next: 4
Next: 1
Next: 2
Next: 3
Next: 4


Prev: 4
Prev: 3
Prev: 2
Prev: 1
Prev: 4
Prev: 3
Prev: 2
```

Test Case #	Test Case Description
5	Test to add elements to hash table (linked list) and show linked elements on same slot when table size is 10

```
HashTableLL<String, Integer> hL = new HashTableLL<>();
hL.put("Izmir", 1);
hL.put("Foca", 4);
hL.put("Buca", 5);
hL.put("Istanbul", 7);
hL.put("Adana", 2);
hL.put("Edirne", 6);
hL.put("Ankara", 3);
hL.put("Kirikkale", 8);
hL.put("Trabzon", 9);
hL.put("Ordu", 10);
hL.put("Rize", 11);
hL.show( key: "Adana");
System.out.println("\n");
```

```
Chained elements:
Buca
Adana
Edirne
Kirikkale
```

Test Case #	Test Case Description
6	Test to remove element from chained elements and show chained elements again(hash table linked list) when table size is 10

```
HashTableLL<String, Integer> hL = new HashTableLL<>();
hL.put("Izmir", 1);
hL.put("Foca", 4);
hL.put("Buca", 5);
hL.put("Istanbul", 7);
hL.put("Adana", 2);
hL.put("Edirne", 6);
hL.put("Ankara", 3);
hL.put("Kirikkale", 8);
hL.put("Trabzon", 9);
hL.put("Ordu", 10);
hL.put("Rize", 11);
hL.show( key: "Adana");
System.out.println("\n");
hL.remove( key: "Buca");
System.out.println("After deletion of an element from chained elements\n");
hL.show( key: "Adana");
```

```
Chained elements:
Buca
Adana
Edirne
Kirikkale


After deletion of an element from chained elements

Chained elements:
Adana
Edirne
Kirikkale
```

Test Case #	Test Case Description
7	Test of accessing nonexistent element(hash table linked list)

```
System.out.format("\nResult of accessing non-existent element: %d\n",hL.get("aaa"));
```

```
Result of accessing non-existent element: null
```

Test Case #	Test Case Description
8	Test of accessing existent element(hash table linked list)

```
System.out.format("\nResult of accessing existent element: %d\n",hL.get("Izmir"));
```

```
Result of accessing existent element: 1
```

Test Case #	Test Case Description
9	Access values in hash table(linked list) with 100 elements

```
HashTableLL<Integer, Integer> hL2 = new HashTableLL<>();
for(int i=0; i<100; i++){
    int key = rand.nextInt( bound: 100);
    int value = rand.nextInt( bound: 100);
    hL2.put(key,value);
}
System.out.format("\nAccess elements in hash table with 100 elements\n");
long sTime = System.nanoTime();
for(int i=0; i<hL2.size(); i++)
    hL2.get(rand.nextInt( bound: 100));
long fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 100 elements
Time to access elements in hash table with 100 elements: 1911197 ns
```

Test Case #	Test Case Description
10	Remove elements from hash table(linked list) with 100 elements

```
System.out.format("\nRemove element from hash table with 100 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hL2.size(); i++)
    hL2.remove(rand.nextInt( bound: 100));
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 100 elements
Time to remove element from hash table with 100 elements: 491511 ns
```

Test Case #	Test Case Description
11	Access values in hash table(linked list) with 1000 elements

```
HashTableLL<Integer, Integer> hL3 = new HashTableLL<>();
for(int i=0; i<1000; i++){
    int key = rand.nextInt( bound: 1000);
    int value = rand.nextInt( bound: 1000);
    hL3.put(key,value);
}
System.out.format("\nAccess elements in hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hL3.size(); i++)
    hL3.get(rand.nextInt( bound: 1000));
fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 1000 elements
Time to access elements in hash table with 1000 elements: 30856085 ns
```

Test Case #	Test Case Description
12	Remove elements from hash table(linked list) with 1000 elements

```
System.out.format("\nRemove element from hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hL3.size(); i++)
    hL3.remove(rand.nextInt( bound: 1000));
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 1000 elements
Time to remove element from hash table with 1000 elements: 1919398 ns
```

Test Case #	Test Case Description
13	Test to add elements to hash table (treeset) and show linked elements on same slot when table size is 10

```
System.out.println("\n ***Hash table implementation with treeset ***\n");
HashTableTree<String, Integer> hT = new HashTableTree<>();
hT.put("Izmir", 1);
hT.put("Foca", 4);
hT.put("Buca", 5);
hT.put("Istanbul", 7);
hT.put("Adana", 2);
hT.put("Edirne", 6);
hT.put("Ankara", 3);
hT.put("Kirikkale", 8);
hT.put("Trabzon", 9);
hT.put("Ordu", 10);
hT.put("Rize", 11);
hT.show( key: "Adana");
System.out.println("\n");
```

```
 ***Hash table implementation with treeset ***

Adana
Buca
Edirne
Kirikkale
```

Test Case #	Test Case Description
14	Test to remove element from chained elements and show chained elements again(hash table treeset) when table size is 10

```
hT.remove( key: "Edirne");
// **** I tried here when table size is 10 and put results in report then made table size 101
System.out.println("After deletion of an element from chained elements\n");
```

```
 ***Hash table implementation with treeset ***

Adana
Buca
Edirne
Kirikkale


After deletion of an element from chained elements

Adana
Buca
Kirikkale
```

Test Case #	Test Case Description
15	Test of accessing existent element(hash table treeset)

```
System.out.format("\nResult of accessing existent element: %d\n",hT.get("Trabzon"));
```

```
Result of accessing existent element: 1
```

Test Case #	Test Case Description
15	Test of accessing nonexistent element(hash table treeset)

```
System.out.format("\nResult of accessing non-existent element: %d\n",hT.get("aadsgasa"));
```

```
Result of accessing non-existent element: null
```

Test Case #	Test Case Description
16	Access values in hash table(treeset) with 100 elements

```
HashTableTree<Integer, Integer> hT2 = new HashTableTree<>();
for(int i=0; i<100; i++){
    int key = rand.nextInt( bound: 100);
    int value = rand.nextInt( bound: 100);
    hT2.put(key,value);
}
System.out.format("\nAccess elements in hash table with 100 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hT2.size(); i++)
    hT2.get(rand.nextInt( bound: 100));
fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 100 elements
Time to access elements in hash table with 100 elements: 1936792 ns
```

Test Case #	Test Case Description
17	Remove elements from hash table(treeset) with 100 elements

```
System.out.format("\nRemove element from hash table with 100 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hT2.size(); i++)
    hT2.remove(rand.nextInt( bound: 100));
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 100 elements
Time to remove element from hash table with 100 elements: 504651 ns
```

Test Case #	Test Case Description
18	Access values in hash table(treeset) with 1000 elements

```
HashTableLL<Integer, Integer> hT3 = new HashTableLL<>();
for(int i=0; i<1000; i++){
    int key = rand.nextInt( bound: 1000);
    int value = rand.nextInt( bound: 1000);
    hT3.put(key,value);
}
System.out.format("\nAccess elements in hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hT3.size(); i++)
    hT3.get(rand.nextInt( bound: 1000));
fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 1000 elements
Time to access elements in hash table with 1000 elements: 2173460 ns
```

Test Case #	Test Case Description
19	Remove elements from hash table(treeset) with 1000 elements

```
System.out.format("\nRemove element from hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hT3.size(); i++)
    hT3.remove(rand.nextInt( bound: 1000));
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 1000 elements
Time to remove element from hash table with 1000 elements: 1321519 ns
```

Test Case #	Test Case Description
20	Test to add elements to hash table (Coalesced hashing) and show hash table when table size is 10

```
HashTableCoa<Integer, Integer> hC = new HashTableCoa<>();
hC.put(3, 1);
hC.put(12, 4);
hC.put(13, 5);
hC.put(25, 7);
hC.put(23, 2);
hC.put(51, 6);
hC.put(42, 3);
System.out.println("\n");

hC.show();
```

```
Index: 1, Key: 51, Next: null
Index: 2, Key: 12, Next: 6
Index: 3, Key: 3, Next: 4
Index: 4, Key: 13, Next: 7
Index: 5, Key: 25, Next: null
Index: 6, Key: 42, Next: null
Index: 7, Key: 23, Next: null
```

Test Case #	Test Case Description
21	Test to remove element from hash table (Coalesced hashing) and show hash table when table size is 10

```
hC.remove( key: 13);
System.out.println("\nAfter removal: ");
hC.show();
```

```
After removal:
Index: 1, Key: 51, Next: null
Index: 2, Key: 12, Next: 6
Index: 3, Key: 3, Next: 4
Index: 4, Key: 23, Next: null
Index: 5, Key: 25, Next: null
Index: 6, Key: 42, Next: null
Index: 7, Key: null, Next: null
```

Test Case #	Test Case Description
22	Test of accessing existent element in hash table(Coalesced hashing)

```
System.out.format("\nResult of accessing existent element: %d\n",hC.get(13));
```

```
Result of accessing existent element: 5
```

Test Case #	Test Case Description
23	Test of accessing nonexistent element in hash table(Coalesced hashing)

```
System.out.format("\nResult of accessing nonexistent element: %d\n",hC.get(999));
```

```
Result of accessing nonexistent element: null
```

Test Case #	Test Case Description
24	Test to remove element from hash table and show elements in hash table(Coalesced hashing) when table size is 10

```
hC.show();
hC.remove( key: 13);
hC.remove( key: 25);
hC.remove( key: 23);
System.out.println("\nAfter removal: ");
hC.show();
```

```
Index: 3, Key: 3, Next: null
Index: 12, Key: 12, Next: null
Index: 13, Key: 13, Next: null
Index: 23, Key: 23, Next: null
Index: 25, Key: 25, Next: null
Index: 42, Key: 42, Next: null
Index: 51, Key: 51, Next: null

After removal:
Index: 3, Key: 3, Next: null
Index: 7, Key: null, Next: null
Index: 12, Key: 12, Next: null
Index: 13, Key: null, Next: null
Index: 23, Key: null, Next: null
Index: 25, Key: null, Next: null
Index: 42, Key: 42, Next: null
Index: 51, Key: 51, Next: null
```

Test Case #	Test Case Description
25	Access values in hash table(Coalesced hashing) with 100 elements

```
HashTableCoa<Integer, Integer> hC2 = new HashTableCoa<>();
for(int i=0; i<100; i++){
    int key = rand.nextInt( bound: 100);
    int value = rand.nextInt( bound: 100);
    hC2.put(key,value);
}
System.out.format("\nAccess elements in hash table with 100 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hC2.size(); i++)
    hC2.get(i);
fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 100 elements
Time to access elements in hash table with 100 elements: 28882 ns
```

Test Case #	Test Case Description
26	Removing elements from hash table(Coalesced hashing) with 100 elements

```
System.out.format("\nRemove element from hash table with 100 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hC2.size(); i++){
    hC2.remove(rand.nextInt( bound: 100));
}
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 100 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 100 elements
Time to remove element from hash table with 100 elements: 173170 ns
```

Test Case #	Test Case Description
27	Access values in hash table(Coalesced hashing) with 1000 elements

```
System.out.format("\nAccess elements in hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hC3.size(); i++)
    hC3.get(i);
fTime = System.nanoTime();
System.out.format("Time to access elements in hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Access elements in hash table with 1000 elements
Time to access elements in hash table with 1000 elements: 215850 ns
```

Test Case #	Test Case Description
28	Removing elements from hash table(Coalesced hashing) with 1000 elements

```
System.out.format("\nRemove element from hash table with 1000 elements\n");
sTime = System.nanoTime();
for(int i=0; i<hC3.size(); i++)
    hC3.remove(rand.nextInt( bound: 1000));
fTime = System.nanoTime();
System.out.format("Time to remove element from hash table with 1000 elements: %d ns\n",fTime - sTime);
```

```
Remove element from hash table with 1000 elements
Time to remove element from hash table with 1000 elements: 774206 ns
```

PERFORMANCE COMPARISON

Hash table implementation with linked list

Time to access elements(100 elements): 2059732 ns

Time to remove element(100 elements): 703079 ns

Time to access elements(1000 elements): 25744278 ns

Time to remove element(1000 elements): 7622490 ns

Hash table implementation with treeset

Time to access elements(100 elements): 2851478 ns

Time to remove element(100 elements): 503521 ns

Time to access elements(1000 elements): 1923787 ns

Time to remove element(1000 elements): 1231185 ns

Hash table implementation with Coalesced hashing technique

Time to access elements(100 elements): 27708 ns

Time to remove element(100 elements): 173170 ns

Time to access elements(1000 elements): 215850 ns

Time to remove element(1000 elements): 774206 ns

As it can be seen, Coalesced hashing technique is by far faster.

RUNNING COMMAND AND RESULTS

1) Put elements in hashmap and iterate forward

```
MapIterator<Integer, Integer> nMap = new MapIterator<>();
nMap.put(1, 42);
nMap.put(2, 58);
nMap.put(3, 1);
nMap.put(4, 58);
MapIterator.MapIter iter = nMap.iterator();
/* Iterate forward */
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.println("\n");
```

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 1
Next: 2
Next: 3
```

2) Iterate backward

```
MapIterator<Integer, Integer> nMap = new MapIterator<>();
nMap.put(1, 42);
nMap.put(2, 58);
nMap.put(3, 1);
nMap.put(4, 58);
MapIterator.MapIter iter = nMap.iterator();
/* Iterate forward */
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.format("Next: %d\n",iter.next());
System.out.println("\n");

/* Iterate backwards */
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
System.out.format("Prev: %d\n",iter.prev());
```

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 1
Next: 2
Next: 3


Prev: 3
Prev: 2
Prev: 1
Prev: 4
Prev: 3
Prev: 2
Prev: 1
```

3) Get element(element's value) from hash table(linked list)

```
System.out.println("\nGet element's value from hash table(linked list)");
System.out.format("Element: %d\n", hL3.get(rand.nextInt( bound: 100)));
```

```
Get element's value from hash table(linked list)
Element: 969
```

4) See chained elements in same slot with 2 (hash table by using linked list)

```
System.out.println("See chained elements in same slot with 2");
hL3.show( key: 2);
```

```
See chained elements in same slot with 2
Chained elements:
409
816
```

5) Remove element from hash table(linked list)

```
System.out.format("Removed element's value from hash table(linked list): %d\n\n",hL.remove( key: "Rize"));
```

```
Removed element's value from hash table(linked list): 11
```

6) Get element(element's value) from hash table(treeset)

```
System.out.println("\nGet element's value from hash table(treeset)");
System.out.format("Element: %d\n", hT3.get(rand.nextInt( bound: 100)));
```

```
Get element's value from hash table(treeset)
Element: 348
```

7) See chained elements in same slot with 68 (hash table by using treeset)

```
System.out.println("See chained elements in same slot with 68(treeset)");
hT3.show( key: 68);
```

```
See chained elements in same slot with 68(treeset)
Chained elements:
882
```

8) Remove element from hash table(treeset)

```
System.out.format("Removed element's value from hash table(treeset): %d\n\n",hT.remove( key: "Ordu"));
```

```
Removed element's value from hash table(treeset): 10
```

9) Get element(element's value) from hash table(Coalesced hashing technique)

```
System.out.println("\nGet element's value from hash table(Coalesced hashing technique)");
System.out.format("Element: %d\n", hC.get(23));
```

```
Get element's value from hash table(Coalesced hashing technique)
Element: 2
```

10) See hash table(Coalesced hashing technique) elements

```
hC.show();
```

```
Index: 3, Key: 3, Next: null
Index: 12, Key: 12, Next: null
Index: 13, Key: 13, Next: null
Index: 23, Key: 23, Next: null
Index: 25, Key: 25, Next: null
Index: 42, Key: 42, Next: null
Index: 51, Key: 51, Next: null
```

11) Remove element from hash table(Coalesced hashing technique)

```
System.out.format("\nRemoved element's value from hash table(Coalesced hashing technique): %d\n\n",hC.remove( key: 42));
```

```
Removed element's value from hash table(Coalesced hashing technique): 3
```