GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 5 Report

Yusuf Arslan
200104004112

1. System Requirement

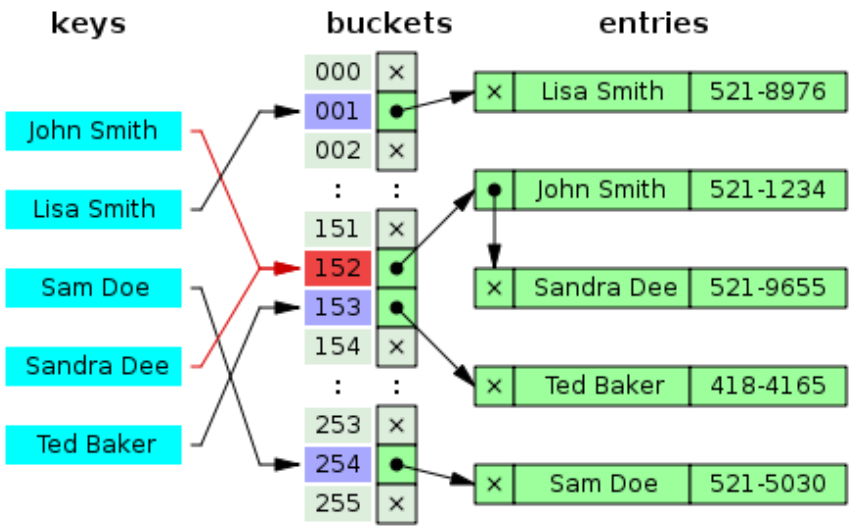
First of all the operating system should have Java virtual machine (JVM), Java development kit (JDK) and ‘at least’ Java runtime environment (JRE) – 11 in a linux distribution.

This application not interactive. User should compile and run program to see result of test of HashTableChainBST HashTableHybrid and sorting classes.
But, user is free to change tests which are in the Driver class.
Note that, due to sorting algorithms durations are too long, the program can take 20 25 seconds to run.

2. Research About Hashing Techniques (Q1.2)

a. Coalesced Hashing

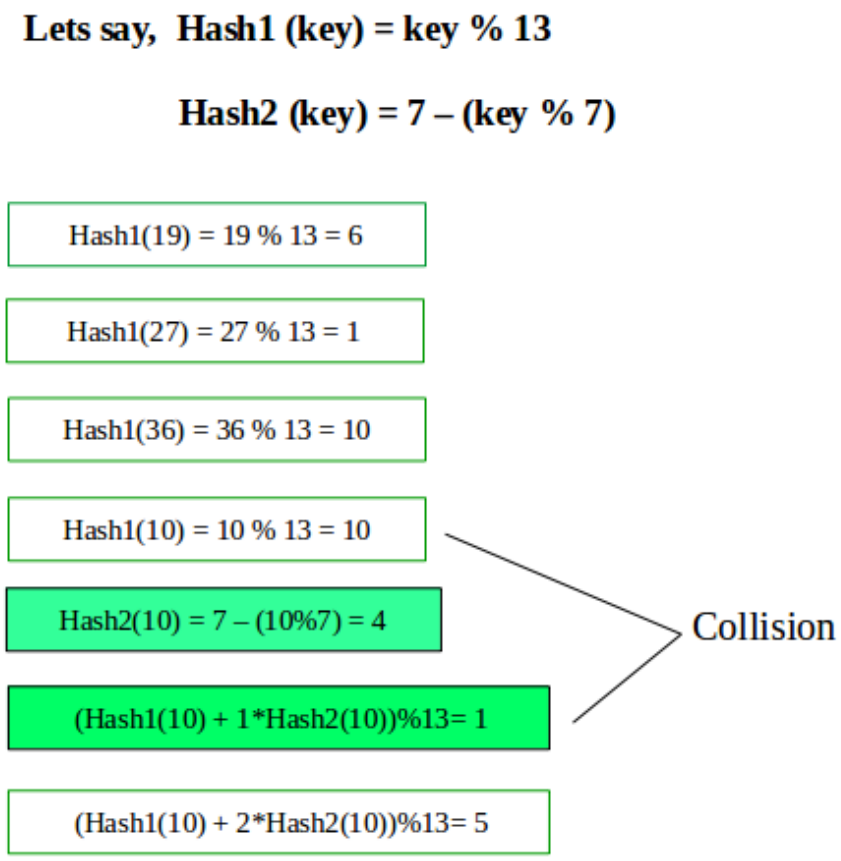
It is a hashing technique that avoid collision when the data is stored in a fixed sized data structure.
It is, combination of seperate chaining and open addressing.
When an element is inserted, a hash value being calculated by modulo and if there is a collision in that index, items are linking each other through the reference of the next index.
Advantage of this technique: In a balanced data set, insertion is fast and calculating hash value from the key is not compilcated (not to many operations) and, once the hash value is being calculated, other items can easyly find through the first element since collided items were linked to the head.
But, removing is hard. Every node, also has an index even they are inserted through the reference. So, another item can be collide to this node. Programmer should keep track of this items before removing a node.Also, to link nodes, there must be extra space in the memory for the references. They are the disadvantages.



source: wikipedia

b. Double Hashing

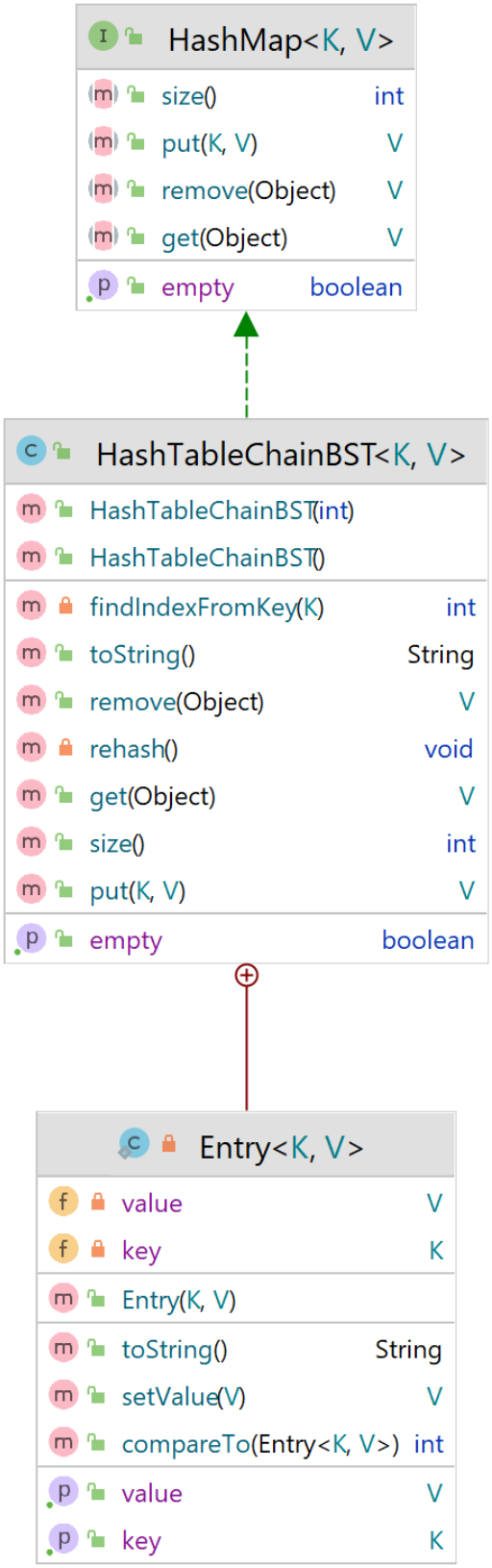
In double hashing technique, a hash value is calculated from the key and, if there is a collision, according to the number of collisions other hash values are being calculated until an empty space is found.
This technique does not requires extra memory for the references and also, removing (replacing removed element by dummy value) and insertion is simple, but since the size of data structure is fixed, resize operation will be performed when it is almost full.
It effects the efficiency. And also, calculating new hash values requires a few operation (sum, division…) is also makes the hashing complicated. But, still it is a effective method for the hashing.



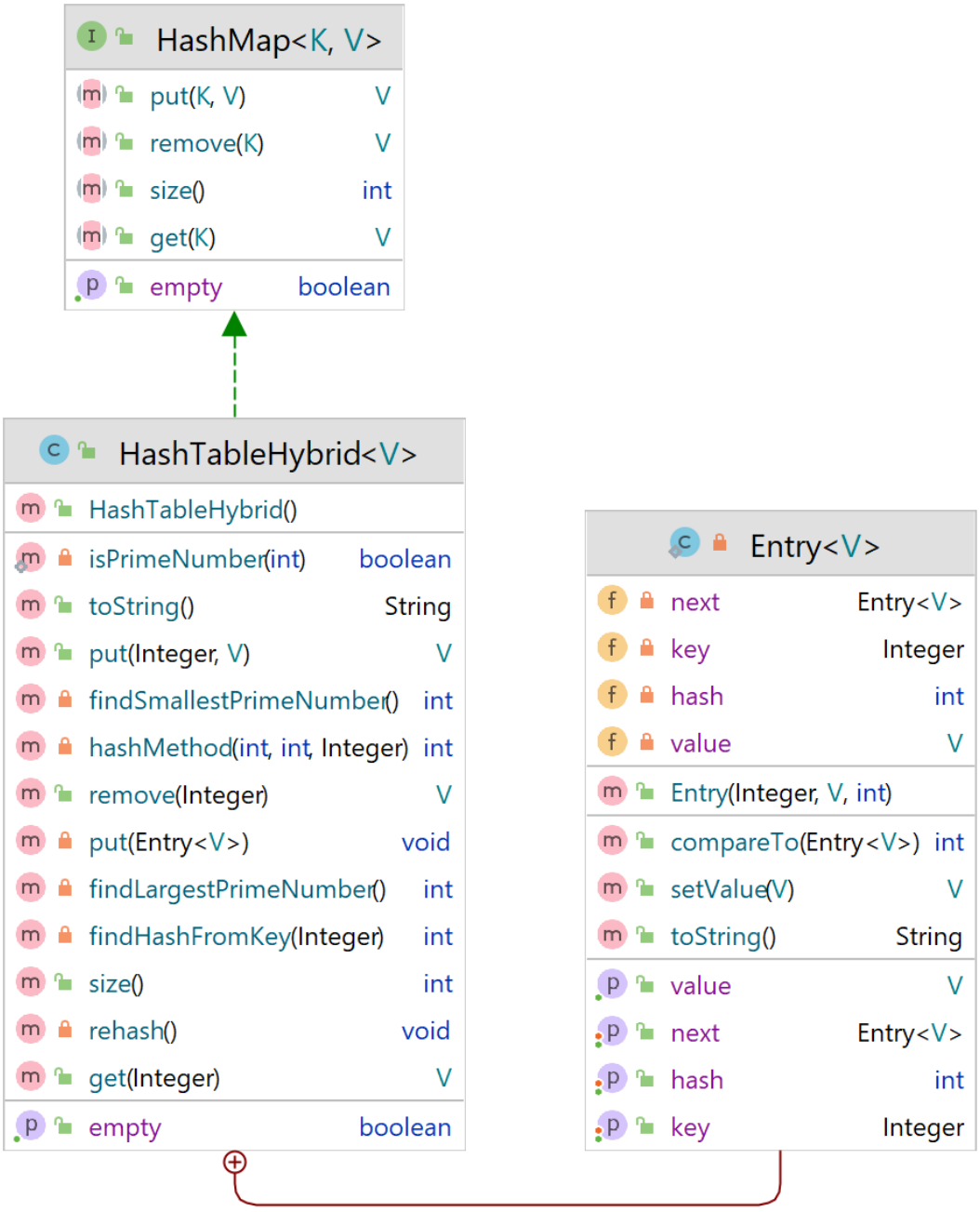
source: geeksforgeeks

3. Class Diagrams

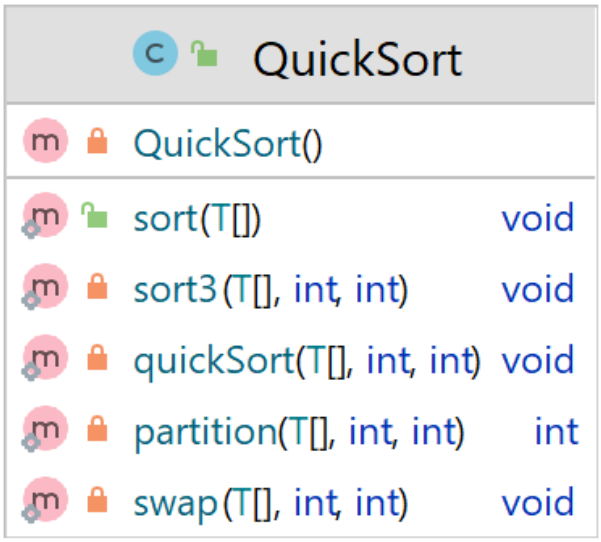
1. Class diagram of HashTableChainBST (Q1.1)

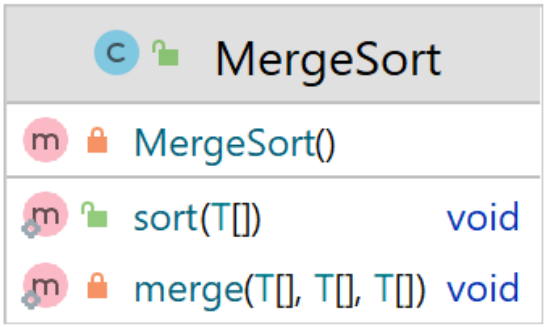


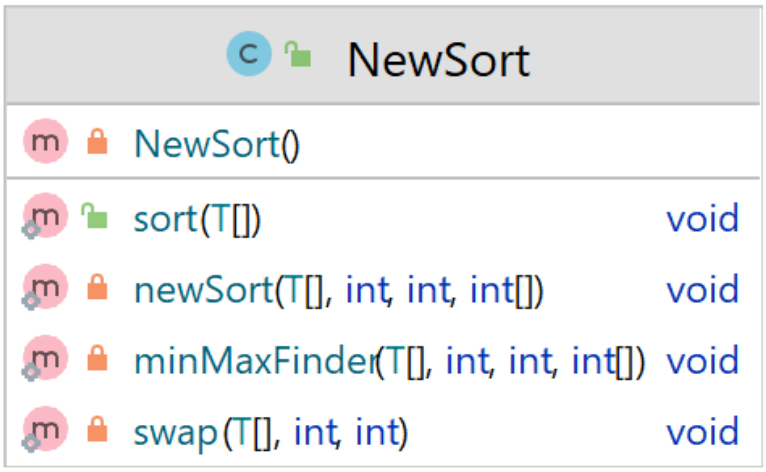
2. Class diagram of HashTableHybrid (Q1.2)



3. Class diagram of Sorting Classes (Q2)







4. Problem Solution Approaches

1. HashTableChainBST (Q1.1)

I implement a hash table from the KWHashMap interface which is perform operations like (put, remove, get…).
It is a simple chain hash table but instead of using linked lists, we used binary search tree as data fields during the scope of the homework.
According to the key value, the index is calculated by modulo operation to the length of the array. If the current place is occupied, collided items will be add to the binary search tree through the link.

2. HashTableHybrid (Q1.2)

I implement a hash table from the KWHashMap interface which is perform operations like (put, remove, get…).
It is a hybrid hash table, that is a combination of the double hashing and coalesced hashing techniques (which is a hybrid of open addressing and chaining).
I calculated the index by the given formula, if the place is occupied, I linked other items to the first element.
To get an item, after first index is calculated by the hash method, items can be search through the head of probes.

3. NewSort (Q2.3)

I impelement the NewSort class which the pseudo codes of it was given the scope of the homework.
I it finds min and max of the array in given range, then set the min to the head and max to the tail, then shrinks the range from both side.

5. Code Snippets and Analysis

1. HashTableChainBST (Q1.1)

a. Put()

```java
@Override
public V put(K key, V value)
{
    int index = findIndexFromKey(key);

    if (table[index] == null)
    {
        // Creating a new BinarySearchTree object and assigning it to
        // the index of the table if current position is null.
        table[index] = new BinarySearchTree<>();
    }

    // Adding the key and value to the BinarySearchTree object.
    // If the key is already in the tree, the value is replaced.
    Entry<K, V> entry = table[index].find(new Entry<>(key, value));

    if (entry != null)
    {
        // If the key and value already exist in the BinarySearchTree object,
        // then return the value.
        V oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    }
    else
    {
        // If the key and value does not exist in the BinarySearchTree object,
        // then add the key and value to the BinarySearchTree object.
        table[index].add(new Entry<>(key, value));
        numKeys++;
        if (numKeys > table.length * LOAD_THRESHOLD)
        {
            // If the load factor is greater than the load threshold,
            // then increase the size of the table by the initial capacity.
            rehash();
        }
        return null;
    }
}
```

Put method takes a key and a value.
It finds index (simple modulo operation), if the element is already in the list, it replace the its value with the current value and returns the previous element. Otherwise, puts the new entry to the binary search tree.

b. Rehash()

```
private void rehash()
{
    // Creating a new table with the size of the current table plus the old capacity.
    BinarySearchTree<Entry<K, V>>[] oldTable = table;
    table = new BinarySearchTree[oldTable.length * 2 + 1];
    numKeys = 0;
    for (int i = 0; i < oldTable.length; i++)
    {
        if (oldTable[i] != null)
        {
            // Iterating through the BinarySearchTree object
            // and adding the key and value to the new table.
            Iterator<Entry<K, V>> iterator = oldTable[i].iterator();
            while (iterator.hasNext())
            {
                Entry<K, V> entry = iterator.next();
                put(entry.getKey(), entry.getValue());
            }
        }
    }
}
```

If the table is out of the load threshold, rehash method is performing.
It iterates all elements and creates a new table.

c. Get()

```java
public V get(Object key)
{
    if (key == null)
    {
        throw new IllegalArgumentException("Key cannot be null.");
    }

    int index = findIndexFromKey((K) key);
    if (table[index] == null)
    {
        return null;
    }
    else
    {
        // Find the value associated with the key.
        // If there is no value associated with the key, return null.
        Entry<K, V> entry = table[index].find(new Entry<>((K) key, null));
        if (entry != null)
        {
            return entry.getValue();
        }
        else
        {
            return null;
        }
    }
}
```

Find the first hash and then find the requested element from the binary search tree.

d. Remove()

```java
@Override
public V remove(Object key)
{
    if (key == null)
    {
        throw new IllegalArgumentException("Key cannot be null.");
    }
    int index = findIndexFromKey((K) key);
    if (table[index] == null)
    {
        return null;
    }
    else
    {
        // Removing the key and value from the BinarySearchTree object.
        Entry<K, V> entry = table[index].find(new Entry<>((K) key, null));
        if (entry != null)
        {
            V value = entry.getValue();
            table[index].remove(entry);

            // If the BinarySearchTree object is empty, then
            // remove the BinarySearchTree object from the table.
            if ("".equals(table[index].toString()))
            {
                table[index] = null;
            }
            numKeys--;
            return value;
        }
        else
        {
            return null;
        }

    }
}
```

Find the first hash result, and remove the item from the corresponded binary search tree element.

2. HashTableHybrid (Q1.2)

a. Put()

```java
public V put(Integer key, V value)
{
    V oldValue = null;
    int hash = findHashFromKey(key);
    Entry<V> entry = new Entry<>(key, value, hash);
    if (table[hash] != null)
    {
        oldValue = table[hash].getValue();
    }
    put(entry);
    return oldValue;
}
```

Find the place for the value that want to putted to the table.
If there is already an element in that place, first call private put method and return the previous value.
Private `put()` method:

```java
private void put(Entry<V> entry)
{
    if (entry.getHash() > table.length)
    {
        rehash();
        entry.setHash(findHashFromKey(entry.getKey()));
        put(entry);
    }
    else if (table[entry.getHash()] == null)
    {
        table[entry.getHash()] = entry;
        if (prevIndex != -1 && table[prevIndex] != null)
        {
            table[prevIndex].setNext(entry);
        }
        numKeys++;
        if (numKeys >= table.length * PRIME_NUMBER_CONSTANT)
        {
            rehash();
        }
    }
    else
    {
        table[entry.getHash()] = entry;
    }
}
```

It sets the new its corresponding place which is find in `findHashFromKey()` method. If the key already in the table, it just replace the previous value with the new value.

`findHashFromKey()` method:

```java
private int findHashFromKey(Integer key)
{
    int primeNumber = findLargestPrimeNumber();
    int hash1 = key % table.length;
    int hash2 = primeNumber - (key % primeNumber);

    return hashMethod(hash1, hash2, key);
}
```

Finds first and second hash value and invokes `hashMethod()`:

```java
private int hashMethod(int hash1, int hash2, Integer key)
{
    int i = 1;
    int hash = (hash1 + (i * hash2)) % table.length;
    prevIndex = -1;
    boolean isPrevSettable = true;
    while (table[hash] != null && table[hash].getKey().compareTo(key) != 0)
    {
        if (isPrevSettable)
            prevIndex = hash;
        i++;
        hash = (hash1 + (i * hash2)) % table.length;

        if (table[hash] != null && table[hash].getNext() == null)
        {
            isPrevSettable = false;
        }
    }
    return hash;
}
```

It finds place for the key, If it need to be link it sets `prevIndex.`

b. Rehash()

```java
private void rehash()
{
    Entry<V>[] oldTable = table;
    int size = findSmallestPrimeNumber();
    table = new Entry[size];

    for (int i = 0; i < oldTable.length; i++)
    {
        if (oldTable[i] != null)
        {
            oldTable[i].setHash(findHashFromKey(oldTable[i].getKey()));
            oldTable[i].setNext(null);
            put(oldTable[i]);
        }
    }
}
```

If the table size is greater then the 80% of size * constant(0.8) rehash the table and create a larger table with a prime number.

c. Get()

```java
public V get(Integer key)
{
    int primeNumber = findLargestPrimeNumber();

    int hash1 = key % table.length;
    int hash2 = primeNumber - (key % primeNumber);
    int hash = (hash1 + (1 * hash2)) % table.length;

    Entry<V> current = table[hash];
    while (current != null && current.getKey().compareTo(key) != 0)
    {
        current = current.getNext();
    }
    if (current != null)
        return current.getValue();
    return null;
}
```

Find first hash and get the requested element from the links.

d. Remove()

```java
public V remove(Integer key)
{
    int primeNumber = findLargestPrimeNumber();
    int hash1 = key % table.length;
    int hash2 = primeNumber - (key % primeNumber);
    int hash = (hash1 + (1 * hash2)) % table.length;

    if (table[hash] == null)
        return null;

    V value;

    if (table[hash].getKey().compareTo(key) == 0 && table[hash].getNext() == null)
    {
        value = table[hash].getValue();
        table[hash] = null;
    }
    else if (table[hash].getKey().compareTo(key) == 0 && table[hash].getNext() != null)
    {
        // Go to the last element and set it to the as first element.
        Entry<V> current = table[hash];
        value = current.getValue();

        while (current.getNext().getNext() != null)
        {
            current = current.getNext();
        }
        table[hash].setValue(current.getNext().getValue());
        table[hash].setKey(current.getNext().getKey());

        int removedHash = current.getNext().getHash();
        current.setNext(null);
        table[removedHash] = null;
    }
    else
    {
        Entry<V> current = table[hash];
        while (current.getNext() != null && current.getNext().getKey().compareTo(key) != 0)
        {
            current = current.getNext();
        }
        if (current.getNext() == null)
            return null;

        value = current.getNext().getValue();
        int removedHash = current.getNext().getHash();
        current.setNext(current.getNext().getNext());
        table[removedHash] = null;
    }
    numKeys--;
    return value;
}
```

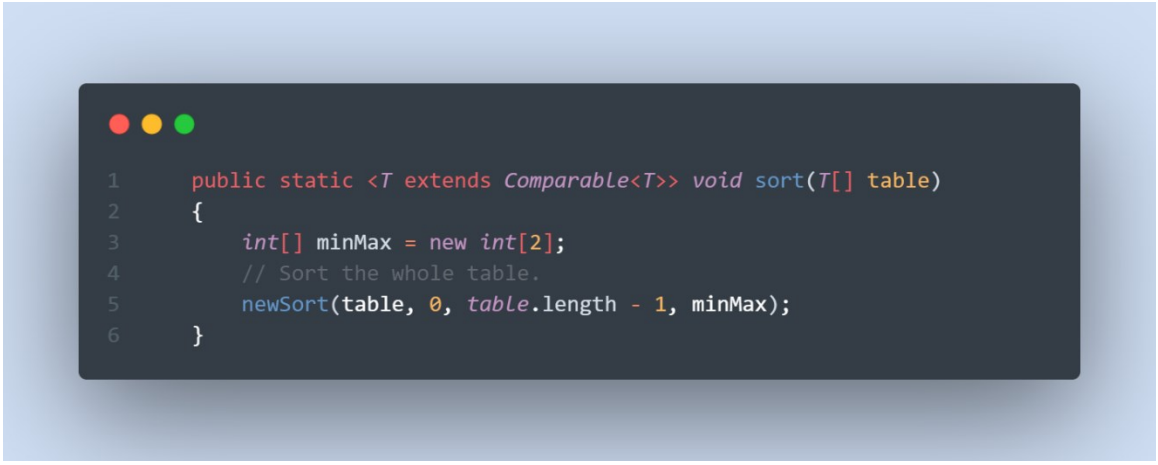
Find first element, if it’s only element, then simply set it to the null, if it is the first element of the linked elements, then, go to to the last element and set it as the first element.

If it is any other place on the list, remove it just set the next

element to the next of removed element.
Finally return the removed value if removing is performed, null otherwise.

3. NewSort (Q2.3)

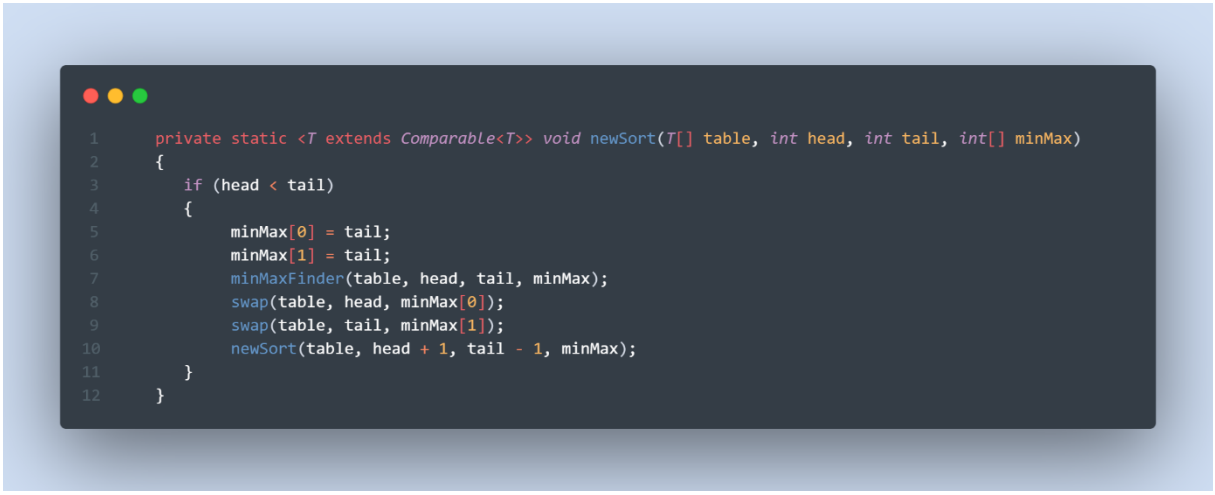
a. Sort()



```java
public static <T extends Comparable<T>> void sort(T[] table)
{
    int[] minMax = new int[2];
    // Sort the whole table.
    newSort(table, 0, table.length - 1, minMax);
}
```

Takes a table and invoke `newSort()` method to sort it.

b. NewSort()



```java
private static <T extends Comparable<T>> void newSort(T[] table, int head, int tail, int[] minMax)
{
   if (head < tail)
   {
        minMax[0] = tail;
        minMax[1] = tail;
        minMaxFinder(table, head, tail, minMax);
        swap(table, head, minMax[0]);
        swap(table, tail, minMax[1]);
        newSort(table, head + 1, tail - 1, minMax);
   }
}
```

Sets min and max value to the tail (any element), finds min and max value by `minMaxFinder()` method and swaps head with min value and tail with max value, then shrinks the range from 2 side.

c. MinMaxFinder()

```
private static <T extends Comparable<T>> void minMaxFinder(T[] table, int head, int tail, int[] minMax)
{
    if (head < tail)
    {
        int middle = (head + tail) / 2;
        if (table[middle].compareTo(table[minMax[0]]) < 0)
        {
            minMax[0] = middle;
        }
        if (table[middle].compareTo(table[minMax[1]]) > 0)
        {
            minMax[1] = middle;
        }
        minMaxFinder(table, head, middle, minMax);
        minMaxFinder(table, middle + 1, tail, minMax);
    }
}
```

Find the middle of the array, if the value in the middle is greater then the current largest element set max to middle, if it is smaller then the current smallest element then set min to the middle.

Then divide problem into 2 part, check both left and right side of the array until no elements left.

6. Test Cases (Class Tests)

1. HashTableChainBST (Q1.1)

a. Test 1:

Put elements without collision.

b. Test2:

Put element with collision.

c. Test3:

Remove existent element and try to remove non-existent element.

d. Test4:

Get existent element and try to get non-existent element.

2. HashTableHybrid (Q1.2)
    a. Test 1:
       Put elements without collision.
    b. Test2:
       Put element with collision.
    c. Test3:
       Remove existent element and try to remove non-existent element.
    d. Test4:
       Remove from head of the link, remove from middle of the link.
    e. Test5:
       Get an existent element and try to get non-existent element.
    f. Test6:
       Get an element after break the link.

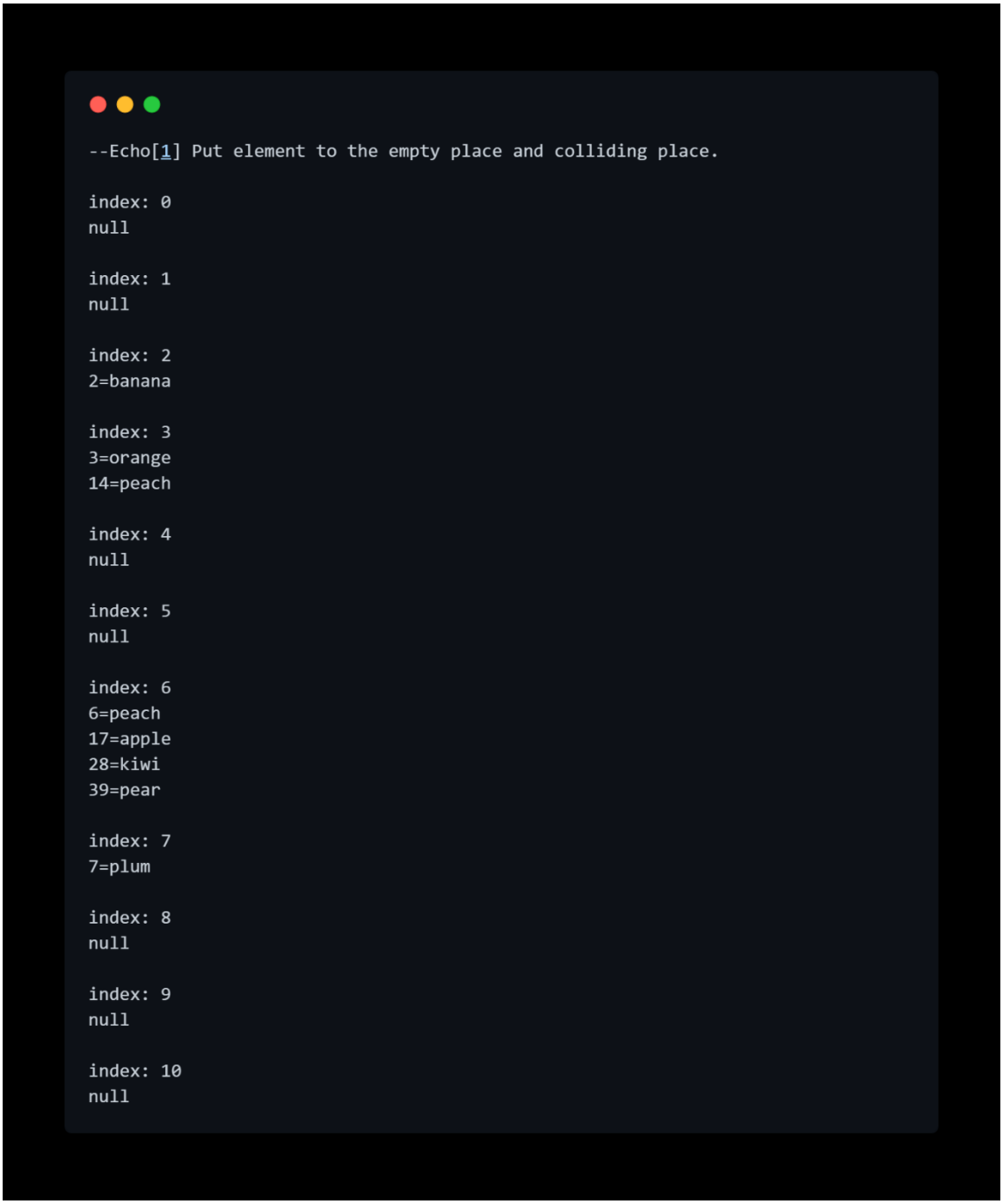
7. Test Cases (Time Tests)

For the HashTableChainBST and HashTableHybrid create 100 randomly generated data sets and for each different set sizes (small (size = 100), medium (size = 1000), and large (size = 10000)).

For sorting algorithms create 1000 randomly generated arrays for each problem size (small (size = 100), medium (size = 1000), and large (size = 10000)).

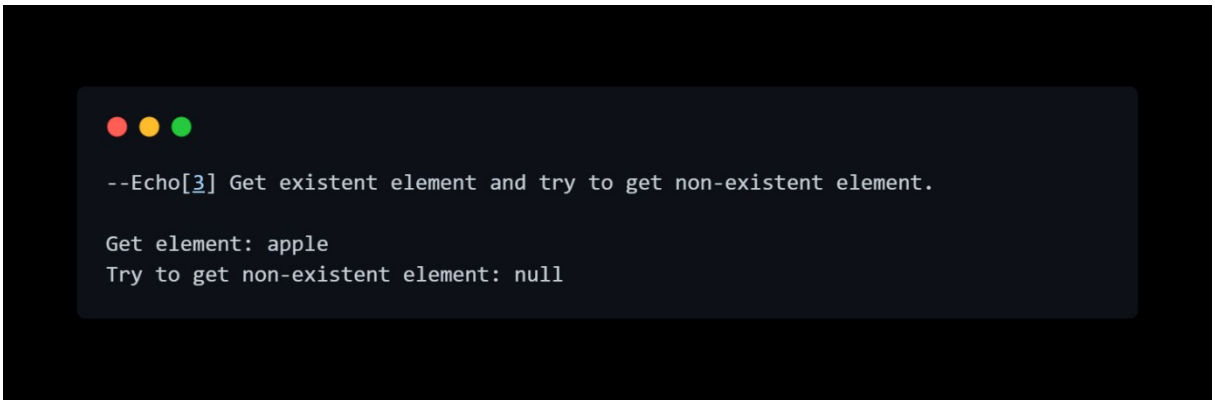
8. Running and Results (Class Tests)

1. HashTableChainBST (Q1.1)



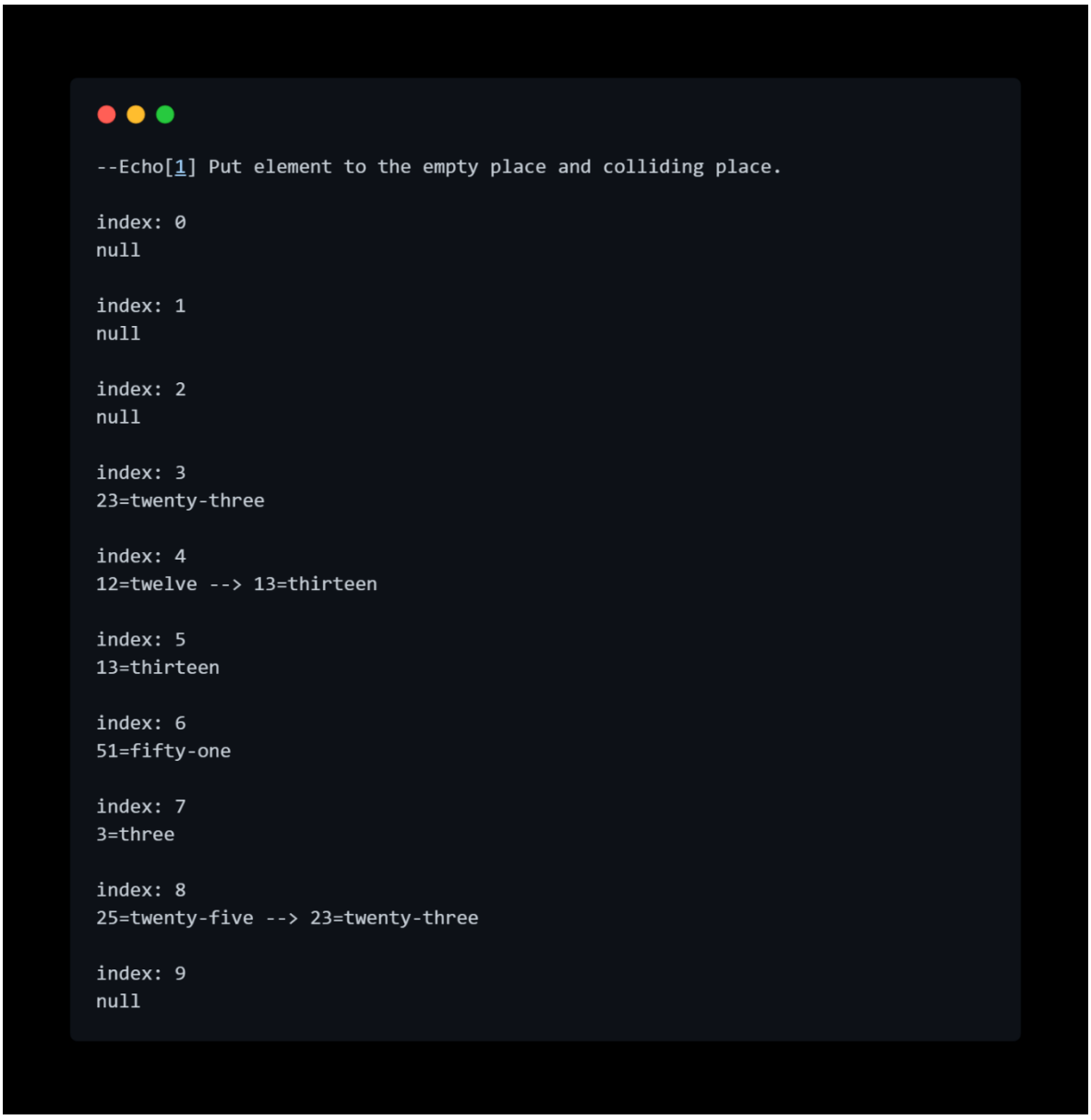
```
--Echo[1] Put element to the empty place and colliding place.

index: 0
null

index: 1
null

index: 2
2=banana

index: 3
3=orange
14=peach

index: 4
null

index: 5
null

index: 6
6=peach
17=apple
28=kiwi
39=pear

index: 7
7=plum

index: 8
null

index: 9
null

index: 10
null
```

```
--Echo[2] Remove existent element and try to remove non-existent element.

Remove element: banana
Try to remove non-existent element: null

index: 0
null

index: 1
null

index: 2
null

index: 3
3=orange
14=peach

index: 4
null

index: 5
null

index: 6
6=peach
17=apple
28=kiwi
39=pear

index: 7
7=plum

index: 8
null

index: 9
null

index: 10
null
```



```
--Echo[3] Get existent element and try to get non-existent element.

Get element: apple
Try to get non-existent element: null
```

2. HashTableHybrid (Q1.2)



```
--Echo[1] Put element to the empty place and colliding place.

index: 0
null

index: 1
null

index: 2
null

index: 3
23=twenty-three

index: 4
12=twelve --> 13=thirteen

index: 5
13=thirteen

index: 6
51=fifty-one

index: 7
3=three

index: 8
25=twenty-five --> 23=twenty-three

index: 9
null
```

```
--Echo[2] Remove existent element and try to remove non-existent element.

Remove element with key 25 (Head of a link)): twenty-five
Remove element with key 12 (Middle of a link): thirteen
Remove element with key 100 (Non-existent): null

index: 0
null

index: 1
null

index: 2
null

index: 3
null

index: 4
12=twelve

index: 5
null

index: 6
51=fifty-one

index: 7
3=three

index: 8
23=twenty-three

index: 9
null
```

```
--Echo[3] Get an existent element and try to get a non-existent element.

Get element with key 23 (existent - after break a link): twenty-three
Get element with key 100 (Non-existent): null
```

9. Running and Results (Time Tests)

1. HashTableChainBST

2. HashTableHybrid

Yetistiremedim…

3. NewSort

```
Array size:     100x100
------------------------
Merge Sort:     0.06ms
Quick Sort:     0.05ms
New Sort:   0.11ms


Array size:     100x1000
------------------------
Merge Sort:     0.98ms
Quick Sort:     0.68ms
New Sort:   1.67ms


Array size:     100x10000
------------------------
Merge Sort:     3.27ms
Quick Sort:     1.18ms
New Sort:   161.62ms


Merge Sort  Quick Sort  New Sort
----------  ----------  --------
0.06ms      0.05ms      0.11ms
0.98ms      0.68ms      1.67ms
3.27ms      1.18ms      161.62ms
```

Time complexity of MergeSort: O(nlogn)

Time complexity of QuickSort (Average): O(nlogn)

Time complexity of NewSort: $O(n^2)$

Experimental results and theoretical results are consistent.