# GTU Department of Computer Engineering
# CSE 222/505 - Spring 2022
# Homework #06 Report

**Ali Kaya**

**1901042618**

**a.kaya2019@gtu.edu.tr**

**1.Detailed System Requirements**

*#For HashTableBinaryChaining<K , V>*

Implements the chaining technique for hashing. However, it uses binary search trees to chain items mapped on the same table slot.

To use this class, the user must specify two generics types. let's call them K andV.

```
public class HashTableBinaryChaining<K , V> implements KWHashMap<K , V>{
```

*#For HashTableCombine<K , V>*

Implements a hashing technique that is a combination of the double hashing and coalesced hashing techniques.

Again , To use this class, the user must specify two generics types. let's call them K and V.

```
public class HashTableCombine<K , V> implements KWHashMap<K, V>{
```

*#For MergeSort*

To use this class' sort method, the user must specify tree parameter.

int[] arr => our initial array

int l => our start index (0)        int r => our last index  (arr.length-1)

```
public void sort(int[] arr , int l , int r){
```

*#For QuickSort*

To use this class' sort method, the user must specify tree parameter.

int[] arr => our initial array

int low => our start index (0)        int high => our last index  (arr.length-1)

```
public void quickSort(int[] arr, int low, int high) {
```

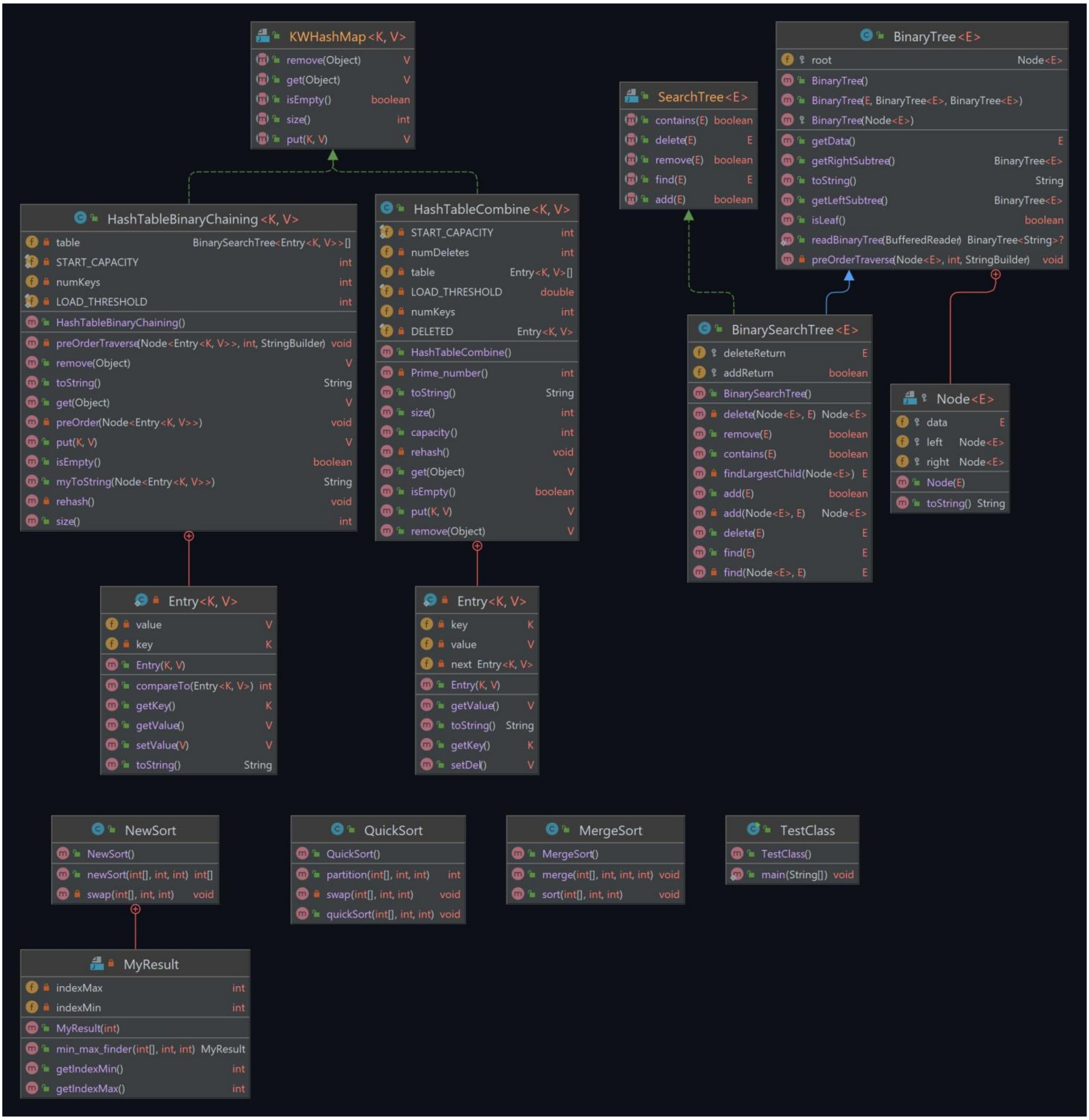
*#For NewSort*

To use this class' sort method, the user must specify tree parameter.

int[] arr => our initial array

int head => our start index (0)        int tail => our last index  (arr.length-1)

## 2. Class Diagrams



## 3. Problem solutions approach

*#For HashTableBinaryChaining<K , V>*

This class Implements the chaining technique for hashing. However, it uses binary search trees to chain items mapped on the same table slot.

```
private BinarySearchTree< Entry<K , V> >[] table;
```

```
----------------------
33
    11
        0
            null
            null
        22
            null
            null
    55
        44
            null
            null
        66
            null
            null
----------------------
34
    12
        1
            null
            null
        23
            null
            null
    56
        45
            null
            null
        67
            null
            null
----------------------
[empty table slot]
----------------------
[empty table slot]
----------------------
```

I used the implementations in the book for BinarySearchTree.

If the values from the hash method of the two values are the same, these values are sorted according to their keys in the bst in the same slot.

When the class is opened, the initial capacity is certain, and the threshold value for the load factor is certain.

```
private static final int START_CAPACITY = 11;
private static final int LOAD_THRESHOLD = 3;
```

Each time the put method is used, a comparison is made between the loadFactor and the LOAD_THRESHOLD value for the table. if necessary, rehash() method is called.

```
if (numKeys > (LOAD_THRESHOLD * table.length)){
    rehash();
}
```

```
private void rehash() {

    BinarySearchTree< Entry < K, V >>[] oldTable = table;
    table = new BinarySearchTree[2 * oldTable.length + 1];

    numKeys = 0;
    for (int i = 0; i < oldTable.length; i++) {
        if (oldTable[i] != null) {
            preOrder(oldTable[i].root);
        }
    }
}
```

Although the remove process is difficult in Coalesced Chaining, this chaining model avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithm for separate chaining. If the chains are short, this strategy is very efficient and can be highly condensed, memory-wise. As in open addressing, deletion from a coalesced hash table is awkward and potentially expensive, and resizing the table is terribly expensive and should be done rarely, if ever.

Double hashing is also a collision resolution technique when two different values to be searched for produce the same hash key. As an advantage double hashing finally overcomes the problems of the clustering issue but as an disadvantage double hashing is more difficult to implement than any other.

*#For HashTableCombine<K , V>*

This class Implements a hashing technique that is a combination of the double hashing and coalesced hashing techniques.

```
public class HashTableCombine<K , V> implements KWHashMap<K, V>{
```

During an insertion operation, the probe positions for the colliding item are calculated by using the double hashing function.

It uses the following hash function to calculate probe positions.

```
int i = 1;
int Prime_number = Prime_number();
int Hash1 = key.hashCode() % table.length;
int Hash2 = Prime_number - (key.hashCode() % Prime_number);
int index;

while(true){

    index = (Hash1 + (i*Hash2)) % table.length;
```

The colliding items are linked to each other through the pointers as in the coalesced hashing technique.

Following code segments links nodes that has same hash value.

```
table[index] = new Entry<K , V>(key, value);
numKeys++;
if(i != 1) {
    int tempIndex = (Hash1 + ((i-1)*Hash2)) % table.length;
    if (tempIndex < 0) tempIndex += table.length;
    table[tempIndex].next = table[index] ;
}
```

```
[0]   19   null
[1]   18   37
[2]   17   null
```

37(it indicates key not index) is the next of 18

Again this class is opened, the initial capacity is certain, and the threshold value for the load factor is certain.

```
private static final int START_CAPACITY = 10;
private final double LOAD_THRESHOLD = 0.75;
```

Each time the put method is used, a comparison is made between the loadFactor and the LOAD_THRESHOLD value for the table. if necessary, rehash() method is called.

```
double loadFactor = (double) (numKeys+numDeletes)/ table.length;
if (loadFactor > LOAD_THRESHOLD)
    rehash();
```

This class also calculates largest prime number that specifies appropriate conditions for Hash Function.

- Hash1 = key % tablesize (10 in our case)
- Hash2 = Prime_number – (key % Prime_number)
- Hash function = ( Hash1 + (*i* * Hash2) ) % tablesize    for the *ith* probe.

Prime_number = the largest prime number smaller than 0.8*table.length

*#For NewSort algorithm*

```java
public int[] newSort (int[] arr, int head, int tail) {
    if(head > tail)
        return arr;
    else
    {
        MyResult result = new MyResult(tail);
        result = result.min_max_finder(arr, head, tail);
        swap(arr , head , result.getIndexMin());
        swap(arr , tail , result.getIndexMax());
        return newSort(arr, head: head + 1, tail: tail -1);
    }
}
```

The min_max_finder() is a recursive function that returns the indices of minimum and maximum items between the given head and tail items in a single execution together.

min_max_finder() method returns a MyResult reference that contains minIndex and maxIndex.

I had to use class to find indexes with both minimum and maximum values.

(next page->)

```java
private static class MyResult {
    private int indexMax;
    private int indexMin;

    public MyResult(int second) {
        this.indexMin = second;
        this.indexMax = second;
    }

    public int getIndexMax() {
        return indexMax;
    }

    public int getIndexMin() {
        return indexMin;
    }

    public MyResult min_max_finder(int[] arr, int head, int tail){
        if(head == tail){
            return this;
        }
        else{
            if (arr[head] > arr[getIndexMax()]){
                indexMax = head;
            }
            if (arr[head] < arr[getIndexMin()]){
                indexMin = head;
            }
            return this.min_max_finder(arr , head: head+1 , tail);
        }
    }
}
```

**4)Test Cases**

| | | |
|---|---|---|
| [0] | 19 | null |
| [1] | 18 | 37 |
| [2] | 17 | null |
| [3] | 16 | null |
| [4] | 15 | null |
| [5] | 14 | null |
| [6] | 13 | null |
| [7] | 12 | null |
| [8] | 11 | null |
| [9] | 10 | null |
| [10] | 9 | null |
| [11] | 8 | null |
| [12] | 7 | null |
| [13] | 6 | null |
| [14] | | null |
| [15] | 4 | null |
| [16] | 3 | null |
| [17] | 2 | null |
| [18] | 1 | null |
| [19] | 0 | 52 |
| [20] | | null |
| [21] | | null |
| [22] | | null |
| [23] | | null |
| [24] | | null |
| [25] | | null |
| [26] | 37 | null |
| [27] | | null |
| [28] | | null |
| [29] | 52 | null |
| [30] | | null |
| [31] | 5 | 19 |
| [32] | | null |
| [33] | | null |
| [34] | | null |
| [35] | | null |
| [36] | | null |
| [37] | | null |
| [38] | 24 | null |
| [39] | | null |
| [40] | | null |
| [41] | | null |
| [42] | | null |

Indicates nextKey not nextIndex

remove(0) =>

remove(18) =>

remove(5) =>

| | | |
|---|---|---|
| [0] | DL | null |
| [1] | 37 | null |
| [2] | 17 | null |
| [3] | 16 | null |
| [4] | 15 | null |
| [5] | 14 | null |
| [6] | 13 | null |
| [7] | 12 | null |
| [8] | 11 | null |
| [9] | 10 | null |
| [10] | 9 | null |
| [11] | 8 | null |
| [12] | 7 | null |
| [13] | 6 | null |
| [14] | | null |
| [15] | 4 | null |
| [16] | 3 | null |
| [17] | 2 | null |
| [18] | 1 | null |
| [19] | 52 | null |
| [20] | | null |
| [21] | | null |
| [22] | | null |
| [23] | | null |
| [24] | | null |
| [25] | | null |
| [26] | DL | null |
| [27] | | null |
| [28] | | null |
| [29] | DL | null |
| [30] | | null |
| [31] | 19 | null |
| [32] | | null |
| [33] | | null |
| [34] | | null |
| [35] | | null |
| [36] | | null |
| [37] | | null |
| [38] | 24 | null |
| [39] | | null |
| [40] | | null |
| [41] | | null |
| [42] | | null |

```java
myHashTable2.put(24 , 24);
myHashTable2.put(52 , 52);
for(int i = 0; i < 20; i++){
    myHashTable2.put(i, i);
}
myHashTable2.put(37 , 37);
System.out.println(myHashTable2.toString());

myHashTable2.remove( key: 5);
myHashTable2.remove( key: 18);
myHashTable2.remove( key: 0);
System.out.println(myHashTable2.toString());
```

```
________________TESTING MergeSort class__________

Initial Array
12 11 13 5 6 7
Sorted array
5 6 7 11 12 13
________________TESTING QuickSort class__________

Initial Array
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
________________TESTING NewSort class____________

Initial Array
5 10 3 8 2 4
Sorted array:
2 3 4 5 8 10
```

```
----------------------
33
    11
        0
            null
            null
        22
            null
            null
    55
        44
            null
            null
        66
            null
            null
----------------------
34
    12
        1
            null
            null
        23
            null
            null
    56
        45
            null
            null
        67
            null
            null
----------------------
[empty table slot]
----------------------
[empty table slot]
```

```
myHashTable1.remove( key: 11);
myHashTable1.remove( key: 56);
```

```
----------------------
33
    0
        null
        22
            null
            null
    55
        44
            null
            null
        66
            null
            null
----------------------
34
    12
        1
            null
            null
        23
            null
            null
    45
        null
        67
            null
            null
----------------------
[empty table slot]
----------------------
[empty table slot]
----------------------
```

*Testing Average Time results for both classes=>>*

*100*small =>*

```
Average Time result for HashTableBinaryChaining: 1.60815E-4
Average Time result for HashTableCombine: 0.001240903
```

*100*medium =>*

```
Average Time result for HashTableBinaryChaining: 0.001677759
Average Time result for HashTableCombine: 0.1331464
```

*100*large =>*

```
Average Time result for HashTableBinaryChaining: 0.00559083
Average Time result for HashTableCombine: 0.207291299
```

*TESTING Average Time results for Sort Algorithms=>>*

*1000*small=>*

```
Average Time result for MergeSort: 1.0377E-5
Average Time result for QuickSort: 1.0675E-5
Average Time result for NewSort: 8.407E-6
```

*1000*medium=>*

```
Average Time result for MergeSort: 3.94368E-4
Average Time result for QuickSort: 0.012457824
Average Time result for NewSort: 0.016298853
```

*1000*large=>*

```
Average Time result for MergeSort: 7.1657E-5
Average Time result for QuickSort: 5.853E-4
Average Time result for NewSort: 5.80401E-4
```

Merge sort => **$T(n) = 2T(n/2) + \theta(n)$**

QuickSort => **$T(n) = T(k) + T(n-k-1) + \theta(n)$**

NewSort => **$T(n) = T(n-2)\ T(n-k-2) + \theta(1)$**

## 5) Running command and results

```
System.out.println("Then let's add some data in it with put(key , value) method");
myHashTable1.put(33 , 33);
myHashTable1.put(11 , 11);
myHashTable1.put(55 , 55);
myHashTable1.put(0 , 0);
myHashTable1.put(22 , 22);
myHashTable1.put(44 , 44);
myHashTable1.put(66 , 66);

myHashTable1.put(34 , 34);
myHashTable1.put(12 , 12);
myHashTable1.put(56 , 56);
myHashTable1.put(1 , 1);
myHashTable1.put(23 , 23);
myHashTable1.put(45 , 45);
myHashTable1.put(67 , 67);
System.out.println("Let's print it with toString() method\n");
System.out.println(myHashTable1.toString());
```

```
-----------------------
33
    11
        0
            null
            null
        22
            null
            null
    55
        44
            null
            null
        66
            null
            null
-----------------------
34
    12
        1
            null
            null
        23
            null
            null
    56
        45
            null
            null
        67
            null
            null
-----------------------
[empty table slot]
-----------------------
[empty table slot]
-----------------------
[empty table slot]
-----------------------
[empty table slot]
```

```java
myHashTable1.remove( key: 11);
myHashTable1.remove( key: 56);
System.out.println(myHashTable1.toString());
```

```
-----------------------
33
    0
        null
        22
            null
            null
    55
        44
            null
            null
        66
            null
            null
-----------------------
34
    12
        1
            null
            null
        23
            null
            null
    45
        null
        67
            null
            null
-----------------------
[empty table slot]
-----------------------
[empty table slot]
-----------------------
[empty table slot]
-----------------------
[empty table slot]
-----------------------
```

```java
System.out.println("Testing get(key) methods;");
System.out.print("get(712) => ");
System.out.println(myHashTable1.get(712));
System.out.print("get(33) => ");
System.out.println(myHashTable1.get(33));
System.out.println();

System.out.println("Testing size() method:");
System.out.print("size() => " + myHashTable1.size() + "\n");
System.out.println();

System.out.println("Testing isEmpty method:");
System.out.print("isEmpty() => " + myHashTable1.isEmpty() + "\n");
System.out.println();
```

```
Testing get(key) methods;
get(712) => null
get(33) => 33

Testing size() method:
size() => 12

Testing isEmpty method:
isEmpty() => false
```

```java
System.out.println("Then let's add some data in it with put(key , value) method");
myHashTable2.put(24 , 24);
myHashTable2.put(52 , 52);
for(int i = 0; i < 20; i++){
    myHashTable2.put(i, i);
}
myHashTable2.put(37 , 37);

System.out.println("Let's print it with toString() method\n");
System.out.println(myHashTable2.toString());
```

```
[1]   18   37
[2]   17   null
[3]   16   null
[4]   15   null
[5]   14   null
[6]   13   null
[7]   12   null
[8]   11   null
[9]   10   null
[10]   9    null
[11]   8    null
[12]   7    null
[13]   6    null
[14]        null
[15]   4    null
[16]   3    null
[17]   2    null
[18]   1    null
[19]   0    52
[20]        null
[21]        null
[22]        null
[23]        null
[24]        null
[25]        null
[26]   37   null
[27]        null
[28]        null
[29]   52   null
[30]        null
[31]   5    19
[32]        null
[33]        null
[34]        null
[35]        null
[36]        null
[37]        null
[38]   24   null
[39]        null
[40]        null
[41]        null
[42]        null
```

```java
System.out.println("Let's remove some elements with remove(key) method");
System.out.println("Program automatically fixes the next current relation between nodes after remove method");
myHashTable2.remove( key: 5);
myHashTable2.remove( key: 18);
myHashTable2.remove( key: 0);
System.out.println(myHashTable2.toString());
```

```
[0]   DL   null
[1]   37   null
[2]   17   null
[3]   16   null
[4]   15   null
[5]   14   null
[6]   13   null
[7]   12   null
[8]   11   null
[9]   10   null
[10]   9    null
[11]   8    null
[12]   7    null
[13]   6    null
[14]        null
[15]   4    null
[16]   3    null
[17]   2    null
[18]   1    null
[19]   52   null
[20]        null
[21]        null
[22]        null
[23]        null
[24]        null
[25]        null
[26]   DL   null
[27]        null
[28]        null
[29]   DL   null
[30]        null
[31]   19   null
[32]        null
[33]        null
[34]        null
[35]        null
[36]        null
[37]        null
[38]   24   null
[39]        null
[40]        null
[41]        null
[42]        null
```

```java
System.out.println("Testing get(key) methods;");
System.out.print("get(990) => ");
System.out.println(myHashTable2.get(990));
System.out.print("get(37) => ");
System.out.println(myHashTable2.get(37));
System.out.println();

System.out.println("Testing size() method:");
System.out.print("size() => " + myHashTable2.size() + "\n");
System.out.println();

System.out.println("Testing isEmpty method:");
System.out.print("isEmpty() => " + myHashTable2.isEmpty() + "\n");
System.out.println();
```

```
Testing get(key) methods;
get(990) => null
get(37) => 37

Testing size() method:
size() => 20

Testing isEmpty method:
isEmpty() => false
```

```java
int[] myArr = { 12, 11, 13, 5, 6, 7 };

System.out.println("Initial Array");
for (int i = 0; i < myArr.length; ++i)
    System.out.print(myArr[i] + " ");

MergeSort testMerge = new MergeSort();
testMerge.sort(myArr, l: 0, r: myArr.length - 1);

System.out.println("\nSorted array");
for (int i = 0; i < myArr.length; ++i)
    System.out.print(myArr[i] + " ");
System.out.println();
```

```
________________TESTING MergeSort class________________

Initial Array
12 11 13 5 6 7
Sorted array
5 6 7 11 12 13
```

```java
int[] myArr2 = { 10, 7, 8, 9, 1, 5 };

System.out.println("Initial Array");
for (int i = 0; i < myArr2.length; ++i)
    System.out.print(myArr2[i] + " ");

QuickSort testQuick = new QuickSort();
testQuick.quickSort(myArr2, low: 0, high: myArr2.length - 1);

System.out.println("\nSorted array: ");
for (int i = 0; i < myArr2.length; ++i)
    System.out.print(myArr2[i] + " ");
System.out.println();
```

```
________________TESTING QuickSort class________________

Initial Array
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```

```java
int[] myArr3 = {5, 10, 3, 8, 2, 4 };

System.out.println("Initial Array");
for (int i = 0; i < myArr3.length; ++i)
    System.out.print(myArr3[i] + " ");

NewSort testNew = new NewSort();
testNew.newSort(myArr3, head: 0, tail: myArr3.length - 1);

System.out.println("\nSorted array: ");
for (int i = 0; i < myArr3.length; ++i)
    System.out.print(myArr3[i] + " ");
System.out.println();
```

```
________________TESTING NewSort class________________

Initial Array
5 10 3 8 2 4
Sorted array:
2 3 4 5 8 10
```