Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

CELAL TEMİZ

101044070

Course Assistant: FATMA NUR ESİRCİ

1 Double Hashing Map

1.1 Pseudocode and Explanation

Bu problemin çözümünde kaynak olarak öncelikle ders kitabından faydalanılmıştır. Kitabın kaynak kodları ile linear probing işlemi yapılmaktadır. Bu methodlar içerisinde değişiklikler yapılarak bizden istenen double hashing probleminin çözümü sağlanmıştır .

Map interface' ine ait problemin çözümündeki metodlar (**get(), isEmpty(), put(), remove() ve size()**) **CTMap adında bir interface**' e eklenmiştir. Daha sonra **CTDoubleHashingMap adında bir class** yazılarak, bu interface methodları implement edilmiştir. Bu class içerisinde hash table için key ve value değerlerini tutan table adında bir Entry dizisi, verilen boyut büyüklüğünde table oluşturmak için START_CAPACITY değişkeni, tablodaki eleman sayısını tutmak için numKeys adında bir değişken, silinen eleman sayısını tutmak için numDeletes ve tabloda silinen bölgeyi işaretlemek için DELETED adında bir Entry objesi bulunmaktadır.

Tabloya ait key ve value değerlerini tutmak için **Entry** adında bir **static inner class** bulunmaktadır.

Sınıfın yapısı :

```
public class CTDoubleHashingMap<K,V> implements CTMap<K,V>{

	static class Entry < K, V > {
		K key;
		V value;
	}

}
```

Find(Object key)
{
Set index to key % table.length.

İf index is negative, add table.length.

While table[index] is not empty
Set index to (hash1(key) + k * hash2(key)) % table.length.
İf index is greate than or equal to table.length
Set index to 0.
Return the index.
}

get(Object key)
{
Find the first table element that is empty or the table element that contains the key.

İf the table element found contains the key
Return the value at this table element.
Else
Return **null**.
}

put(K key, V value)
{
Find the first table element that is empty or the table element that contains the key.

if an empty element was found
Insert the new item and increment numKeys.
Check for need to rehash.

The key was found. Replaced the value associated with this table element and return the old value.
}

remove(Object key)
{
Find the first table element that is empty or the table element that contains the key.

if an empty element was not found
return **null**.
Key was found. Remove this table element by setting it to reference DELETED, increment numDeletes, and decrement numKeys.

Return the value associated with this key.
}

rehash()
{
Allocate a new hash table that is double the size and has an odd length
Reset the number of keys and number of deletions to 0.
Reinsert each table entry that has not been deleted in the new hash table.
}

1.2 Test Cases

Aşağıdaki işlemlerin doğruluğu Q1 package içerisinde Main.java içerisinde test edilerek doğruluğu sağlanmıştır. Ek olarak JUNIT testi yazılmıştır.

Hash Table Size = 101

```
put(14, "Ali");      // No Collision - Assign Ali  to 14.slot
put(24, "Veli");     // No Collision - Assign Veli to 24.slot
put(34, "Ayşe");     // No Collision - Assign Ayşe to 34.slot
put(10, "Ece");      // No Collision - Assign Ece to 10.slot
put(10, "Ege");      // Collision 1  - Assign Ege to 59.slot
put(10, "Can");      // Collision 2  - Assign Can to 7.slot
put(10, "Deniz");    // Collision 3  - Assign Deniz to 56.slot
put(10, "Su");       // Collision 4  - Assign Su to 4.slot
put(14, "Caner");    // Collision 5  - Assign Caner to 3.slot
put(3, "Pelin");     // Collision 7  - Assign Pelin to 70.slot
put(3, "Arda");      // Collision 8  - Assign Arda to 25.slot
put(25,"Semih");     // Collision 9  - Assign Semih to 93.slot
put(25,"Berke Can"); // Collision 10 - Assign Berke Can to 26.slot
```

Hash Table Size = 150

```
put(10, 10);             // No Collision - Assign 10 to 10.slot
put(20, 20);             // No Collision - Assign 20 to 20.slot
put(30, 30);             // No Collision - Assign 30 to 30.slot
put(40, 40);             // No Collision - Assign 40 to 40.slot
put(40, 100);            // Collision 1  - Assign 100 to 59.slot
put(59, 150);            // Collision 2  - Assign 150 to 118.slot
put(59, 300);            // Collision 3  - Assign 300 to 27.slot
put(27, 444);            // Collision 4  - Assign 444 to 91.slot
put(27, 55);             // Collision 5  - Assign 55 to 123.slot
```

2 Recursive Hashing Set

2.1 Pseudocode and Explanation

Bu problemim çözümünde ders kitabındaki kaynak kodlardan faydalanılmıştır.

Bu problemin çözümünde **`CTRecursiveHashingSet<E>`** adında bir class yazarak ders kitabındaki Set<E> interface' ine ait methodları **CTSet<E>** interface' ine eklenerek, bu interface' in implementasyonu gerçekleştirilmiştir. Oluşturduğum bu sınıf içerisine hash table' a ait dataları tutmak için **Data<E>** adında bir **static inner class** eklenmiştir. Data değerlerini tutmak için, table adında bir Data<E> dizisi, tabloya eklenen elemanları tutmak için numberOfElements değişkeni, tablodan silinen elemanları tutmak için numberOfDeletedElements değişkeni, bir LOAD_THRESHOLD değeri ve tabloda silinen bölgeyi işaretlemek için DELETED adında bir Data objesi bulunmaktadır.

2.2 Test Cases

Bu problemin çözümünde eksiklikler bulunmaktadır. Bu yüzden yazılan bütün methodlar test edilememiştir.

3 Sorting Algortihms

3.1 MergeSort with DoubleLinkedList

3.1.1 Pseudocode and Explanation

Bu problemin çözümü için **CTMergeSortDoubleLinkedList** adında bir generic **class** yazılmıştır. Node' lara ait dataların büyüklüğünü karşılaştırmak için **Comparable**<> interface' i implement edilmiştir. Node' lara ait dataları, önceki ve sonraki node'ları gösteren linkleri tutmak için **Node<E> inner class '** ı eklenmiştir. Bu listenin başını tutmak için de sınıf içerisine Node<E> header eklenmiştir. Sınıfın yapısı :

```
public class CTMergeSortDoubleLinkedList
   <E extends Comparable <E>>
   implements Comparable<CTMergeSortDoubleLinkedList> {

     static class Node<E> {

          E data;

          Node<E> next;
          Node<E> prev;

     }

}
```

MergeSort( ) {

Create two Node<E> references to left and right nodes. And check the list header.

İf the list header is not null, divide the list to two piece.

Then assign the linkedList header link to left node, header.next link to right node.

While leftNode.next is not equal to null and leftNode.next.next is not equal to null

Look at the all nodes from left to right in this way.

Do left.next.next assign the left node,then right.next assign the right node.

Sort the left sequence and sort the right sequence.

Merge the left sequence and right sequence.

}

3.1.2 Average Run Time Analysis

Q3 package içerisinde Main.java içerisinde testler yazılmıştır. Ek olarak JUNIT testi yazılmıştır.

LinkedList' in iki parçaya bölünmesinin çalışma zamanı **T(N) = θ (logN)** olmaktadır. LinkedList' in elaman sayısı, üzerinde gezilip karşılaştırma yapılan eleman sayısı N olduğunu için buradan da **T(N) = θ (N)** elde edilir.

Sonuç olarak **T(N) = θ (N logN)** olacaktır.

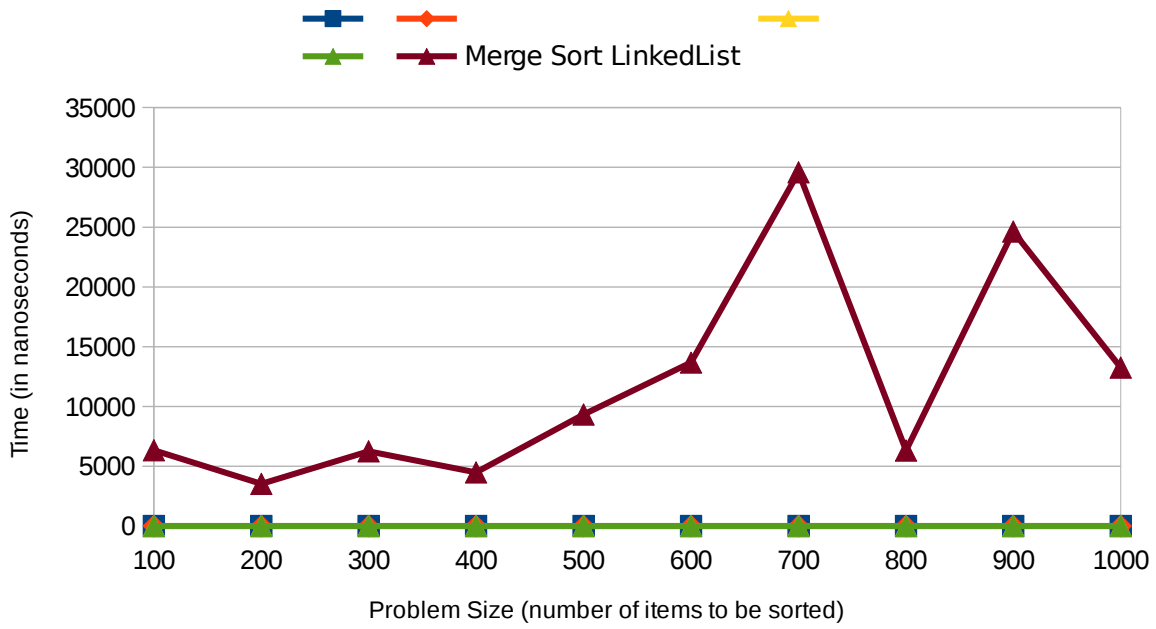

Figure 3.1.2 Merge Sort with Double Linked List algorithm

3.1.3 Wort-case Performance Analysis

Worst – Case Performans Analysis için linkList' in tersten sıralı durumda olması dikkate alınarak analiz yapılmaktadır.

Önce random Integer değerler üretilerek, dizi sıralanmıştır. Daha sonra dizi tersten sıralanarak sort algoritmasına verilmiştir.

Bu durumda **`T( N ) = O (N logN)`** olacaktır.

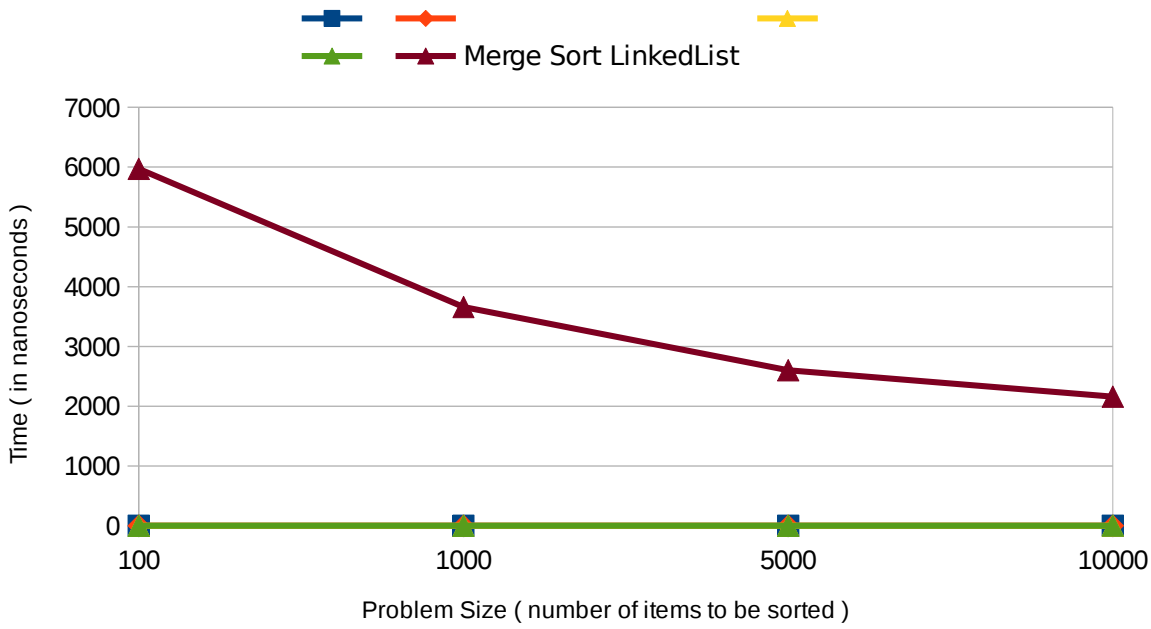

Figure 3.1.3 Merge Sort with Double Linked List algorithm

3.2 MergeSort

3.2.1 Average Run Time Analysis

```
mergeSort(T[] table)                          T( N ) = θ (N logN)
{
  if (table.length > 1) {
    int halfSize = table.length / 2;                          Constant Time
    T[] leftTable = (T[])new Comparable[halfSize];            Constant Time
    T[] rightTable =
        (T[])new Comparable[table.length – halfSize];         Constant Time
    System.arraycopy(table, 0, leftTable, 0, halfSize);       θ ( N )
    System.arraycopy(table, halfSize, rightTable, 0,          θ ( N )
        table.length - halfSize);
    mergeSort(leftTable);              N/2+N/4+...+2+1 =      θ (log N)
    mergeSort(rightTable);             N/2+N/4+...+2+1 =      θ (log N)
    merge(table, leftTable, rightTable);                      θ (N)
  }
}
```

```
merge(T[] outputSequence, T[] leftSequence, T[] rightSequence)
{
  int i = 0,j = 0, k = 0;       Comparison Number : T( N ) = θ ( N )

  while (i < leftSequence.length && j < rightSequence.length) {

    if (leftSequence[i].compareTo(rightSequence[j]) < 0) {
      outputSequence[k++] = leftSequence[i++];
    }
    else {
      outputSequence[k++] = rightSequence[j++];      Constant Time
    }
  }
  while (i < leftSequence.length) {
    outputSequence[k++] = leftSequence[i++];         Constant Time
  }
  while (j < rightSequence.length) {
    outputSequence[k++] = rightSequence[j++];        Constant Time
  }


}
```

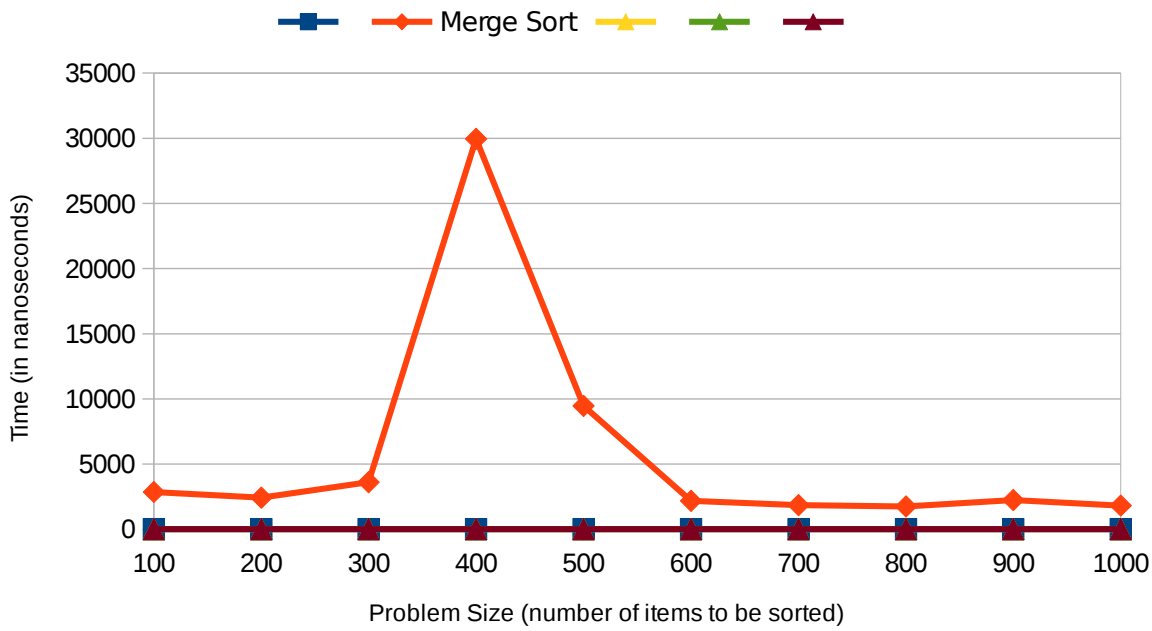

Figure 3.2.1. Merge Sort algorithm

3.2.2 Wort-case Performance Analysis

Worst – Case Performans Analysis için dizinin tersten sıralı durumda olması dikkate alınmıştır.

Önce random Integer değerler üretilerek, dizi sıralanmıştır. Daha sonra dizi tersten sıralanarak sort algoritmasına verilmiştir.

```
mergeSort(T[] table)                    T(N) = O (N log N)
{
  if (table.length > 1) {
    int halfSize = table.length / 2;                        Constant Time
    T[] leftTable = (T[])new Comparable[halfSize];          Constant Time
    T[] rightTable =
        (T[])new Comparable[table.length – halfSize];       Constant Time
    System.arraycopy(table, 0, leftTable, 0, halfSize);     O ( N )
    System.arraycopy(table, halfSize, rightTable, 0,        O ( N )
        table.length - halfSize);
    mergeSort(leftTable);               N/2+N/4+...+2+1  =  O (log N)
    mergeSort(rightTable);              N/2+N/4+...+2+1  =  O (log N)
    merge(table, leftTable, rightTable);                    O (N)
  }
}

merge(T[] outputSequence, T[] leftSequence, T[] rightSequence)
{
  int i = 0,j = 0, k = 0;
  while (i < leftSequence.length && j < rightSequence.length) {
    if (leftSequence[i].compareTo(rightSequence[j]) < 0) {
      outputSequence[k++] = leftSequence[i++];
```

```
    }                                                    Constant Time
    else {
      outputSequence[k++] = rightSequence[j++];
    }
  }
  while (i < leftSequence.length) {                      Constant Time
    outputSequence[k++] = leftSequence[i++];
  }
  while (j < rightSequence.length) {                     Constant Time
    outputSequence[k++] = rightSequence[j++];
  }

     T( N ) = O ( N )
}
```

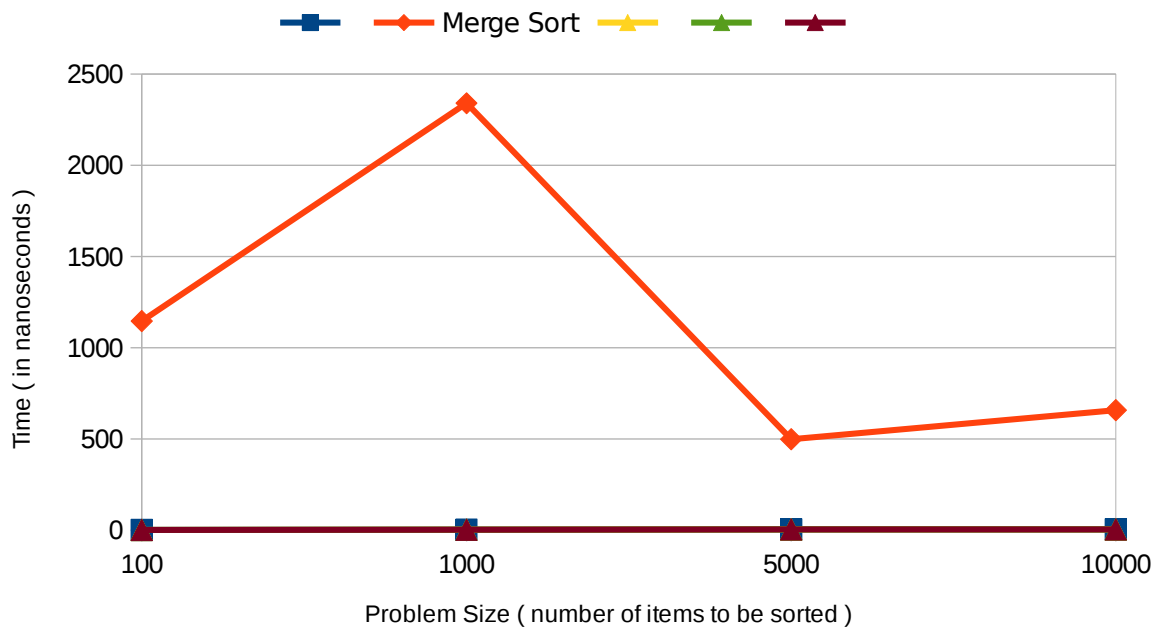

Figure 3.2.2. Merge Sort algorithm

3.3 Insertion Sort

3.3.1 Average Run Time Analysis

```
insertionSort(T[] table)                          T ( N ) = θ ( N^2 )
{
  for (int nextPos = 1; nextPos < table.length; nextPos++)
 {
    insert(table, nextPos);
  }
}

insert(T[] table, int nextPos)
{
  T nextVal = table[nextPos];          Comparison Number :
  while (nextPos > 0 &&                1+2+3+...+N-1    = N*(N-1)/2
                                                         = θ ( N^2 )
     nextVal.compareTo(table[nextPos - 1]) < 0) {
    table[nextPos] = table[nextPos - 1];                  Constant Time
    nextPos--;                                            Constant Time
  }
  table[nextPos] = nextVal;                               Constant Time
}
```

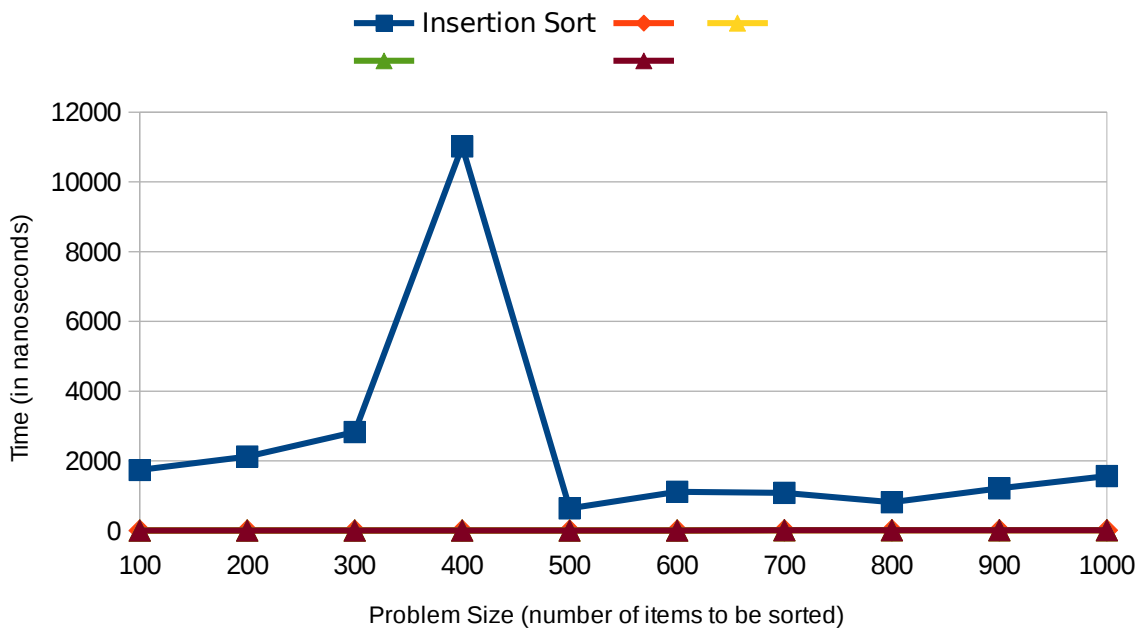

Figure 3.3.1. Insertion Sort algorithm

3.3.2 Wort-case Performance Analysis

Worst – Case Performans Analysis için dizinin tersten sıralı durumda olması dikkate alınmıştır.

Önce random Integer değerler üretilerek, dizi sıralanmıştır. Daha sonra dizi tersten sıralanarak sort algoritmasına verilmiştir.

```
insertionSort(T[] table)
{                                                    T ( N ) = O ( N^2 )
  for (int nextPos = 1; nextPos < table.length; nextPos++)
 {
    insert(table, nextPos);
  }
}

insert(T[] table, int nextPos)
{
  T nextVal = table[nextPos];          Comparison Number:
  while (nextPos > 0 &&                1+2+3+...+N-1  = ((N-1) *N)/2
                                                        = O ( N^2 )
     nextVal.compareTo(table[nextPos - 1]) < 0) {
    table[nextPos] = table[nextPos - 1];              Constant Time
    nextPos--;                                        Constant Time
  }
  table[nextPos] = nextVal;                           Constant Time
}
```

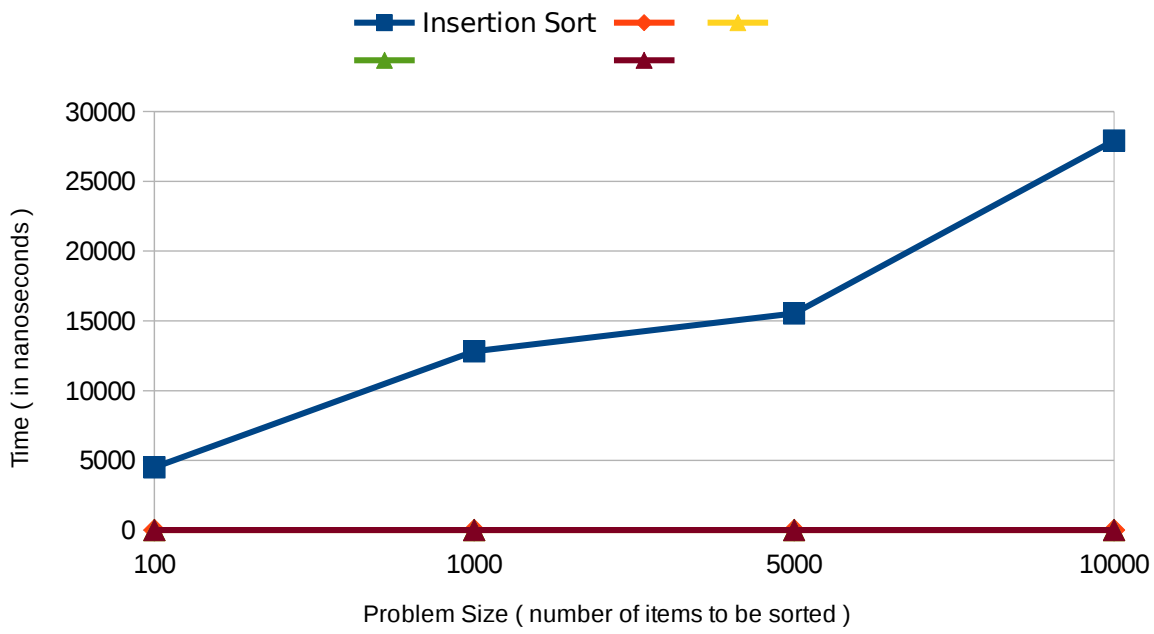

Figure 3.3.2 Insertion Sort algorithm

3.4 Quick Sort

3.4.1 Average Run Time Analysis

```
QuickSort(T[] table) {                          T (N) = θ (N log N)

  quickSort(table, 0, table.length - 1);
}

quickSort(T[] table, int first, int last) {
  if (first < last) {                                    Constant Time
    int pivIndex = partition(table, first, last);        θ (N)
    quickSort(table, first, pivIndex – 1);Recursive Div. θ (log N)
    quickSort(table, pivIndex + 1, last); Recursive Div. θ (log N)
  }
}

partition(T[] table,  int first, int last) {
  T pivot = table[first];
  int up = first;
  int down = last;
  do {                                          Comparison Number :
                                                 θ (N)

    while ( (up < last) && (pivot.compareTo(table[up]) >= 0)) {
      up++;
    }

    while (pivot.compareTo(table[down]) < 0) {
      down--;
    }

    if (up < down) {
      swap(table, up, down);                    Constant Time
    }
  }
  while (up < down);
  swap(table, first, down);                     Constant Time
  return down;
}

swap(T[] table, int i, int j) {                 Constant Time
  T temp = table[i];
  table[i] = table[j];
  table[j] = temp;
}
```

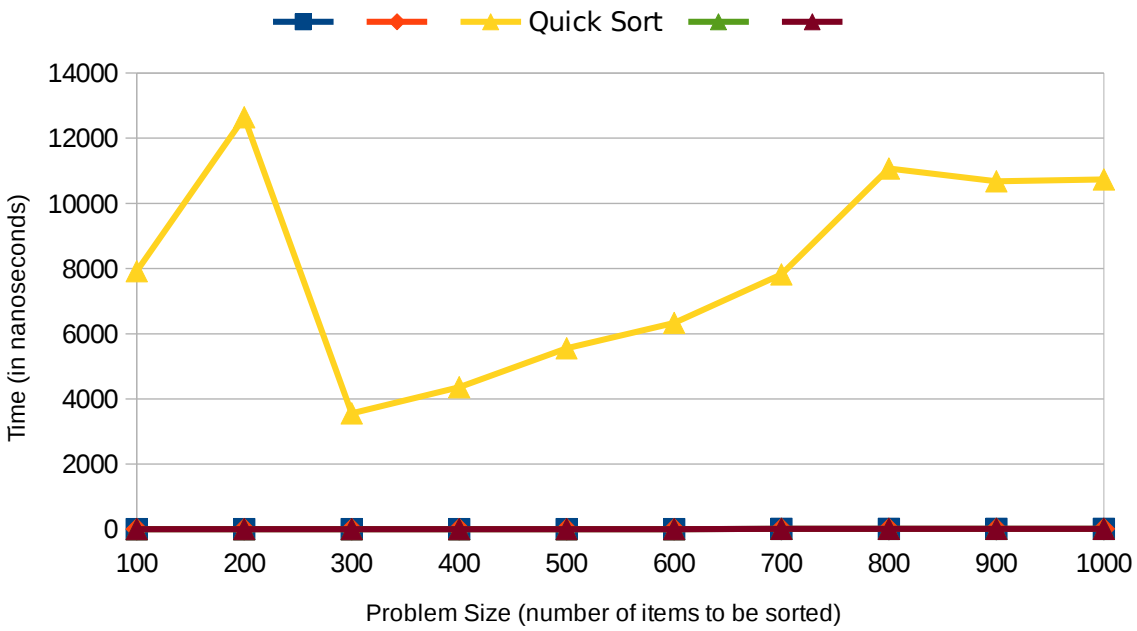

Figure 3.4.1. Quick Sort algorithm

3.4.2 Wort-case Performance Analysis

Pivot seçimi yapılırken dizideki en büyük ya da en küçük elemanın pivot eleman olarak seçilmesi durumunda worst case durumu oluşur.

Önce random Integer değerler üretilerek, dizi sıralanmıştır. Daha sonra dizi tersten sıralanarak sort algoritmasına verilmiştir.

Bu durumda partion methodunda pivot' a göre yapılan toplam karşılaştırma sayısında değişiklik olmaktadır. Bu yüzden performans aşağıdaki gibi olmaktadır.

T (N) = N-1 + N-2 + N-3 + ... + 1

T (N) = (N-1 * N) / 2

T (N) = O(N^2)

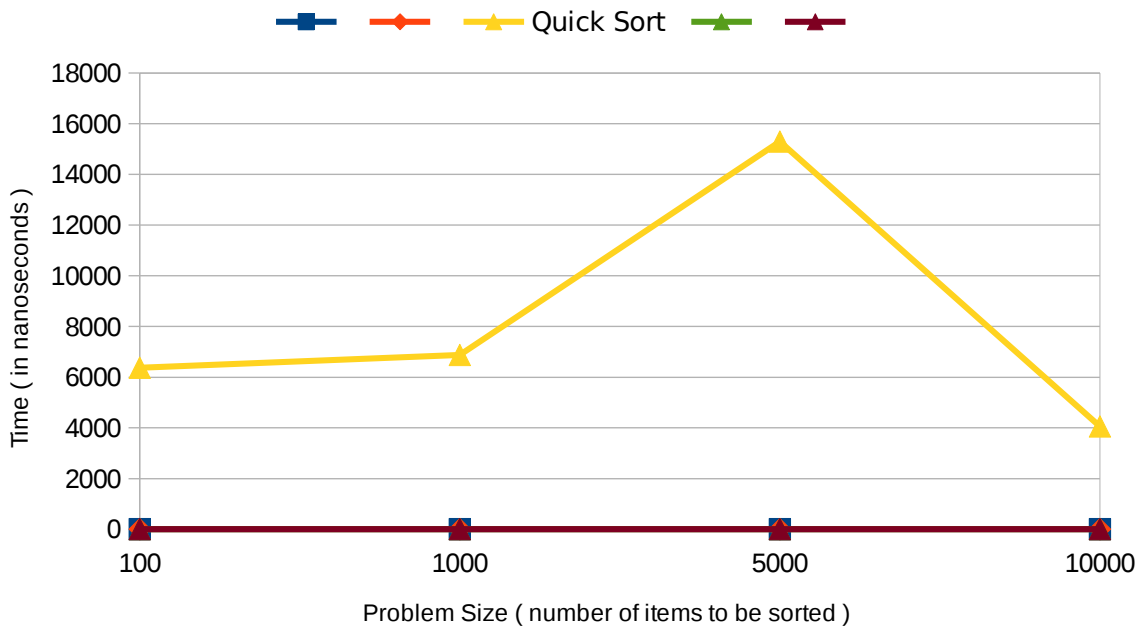

Figure 3.4.2 Quick Sort algorithm

3.5 Heap Sort

3.5.1 Average Run Time Analysis

```
heapSort(T[] table) {
  buildHeap(table);
  shrinkHeap(table);
}                                                    T (N) = θ (N log N)

buildHeap(T[] table) {
  int n = 1;
  while (n < table.length) {
    n++;
    int child = n – 1;                                    Constant Time
    int parent = (child - 1) / 2;                         Constant Time
    while (parent >= 0
       && table[parent].compareTo(table[child]) < 0) {  θ (N)
      swap(table, parent, child);                         Constant Time
      child = parent;                                     Constant Time
      parent = (child - 1) / 2;                           Constant Time
    }
  }
}

shrinkHeap(T[] table) {
  int n = table.length;                                   Constant Time
  while (n > 0) {
    n--;
    swap(table, 0, n);                                    Constant Time
    int parent = 0;
```

```
    while (true) {
      int leftChild = 2 * parent + 1;                    Constant Time
      if (leftChild >= n) {
        break;
      }
      int rightChild = leftChild + 1;                    Constant Time
      int maxChild = leftChild;
      if (rightChild < n
          && table[leftChild].compareTo(table[rightChild]) < 0) {
        maxChild = rightChild;                           θ (log N )
      }

      if (table[parent].compareTo(table[maxChild]) < 0) {
        swap(table, parent, maxChild);                   θ (log N )
        parent = maxChild;
      }
      else {
        break;
      }
    }
  }
}

swap(T[] table, int i, int j) {
  T temp = table[i];
  table[i] = table[j];
  table[j] = temp;
}
```

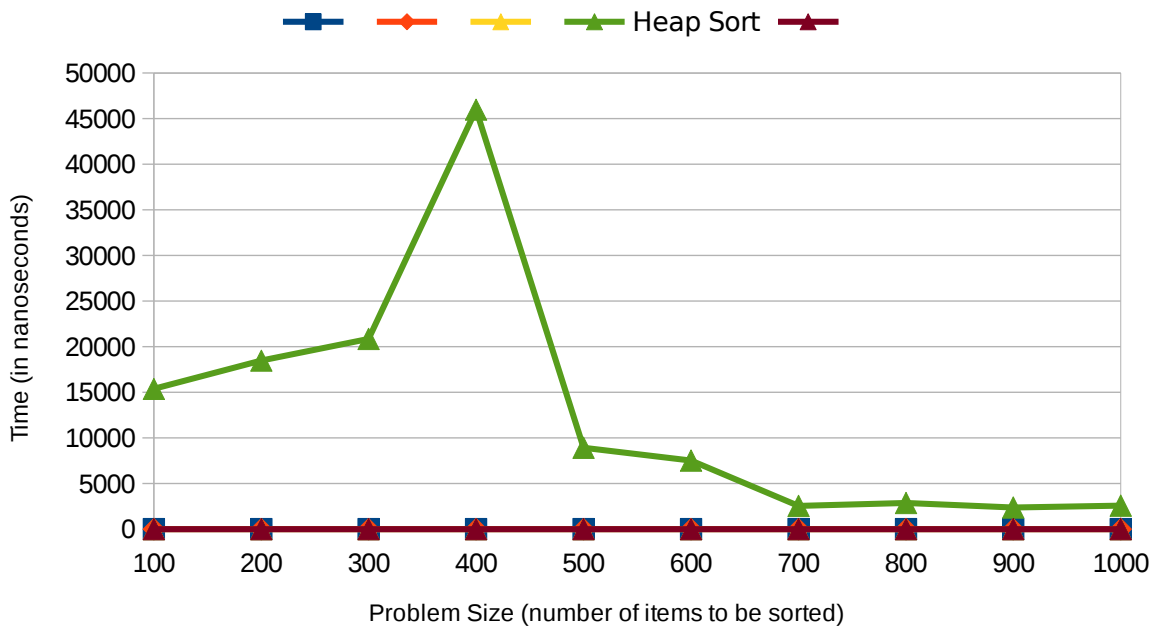

Figure 3.5.1. Heap Sort algorithm

3.5.2 Wort-case Performance Analysis

Önce random Integer değerler üretilerek, dizi sıralanmıştır. Daha sonra dizi tersten sıralanarak sort algoritmasına verilmiştir.

Worst – case durumunda kökteki eleman en büyük eleman ise, en alttaki bir yaprakla yerini değiştirmemiz gerekecektir. Bu yüzden ağacın yüksekliği kadar karşılaştırma ve parent – child arasında yer değiştirme işlemleri gerçekleşecektir.

Buradaki işlemler $O(\log N)$ zaman alacaktır.

Daha sonra heap oluşturma ise $O(N)$ zaman alacaktır.

Sonuç olarak worst case performans analizi $T(N) = O(N \log N)$ olmaktadır.

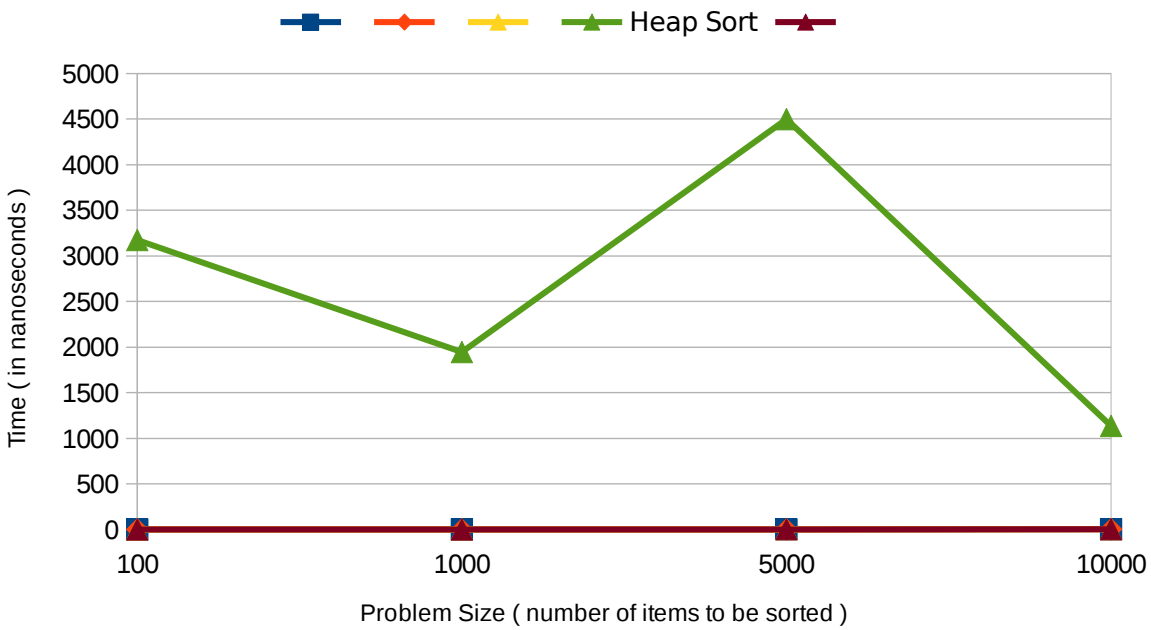

Figure 3.5.2. Quick Sort algorithm

4 Comparison the Analysis Results

5 sorting algorithm worst-case analysis cases.

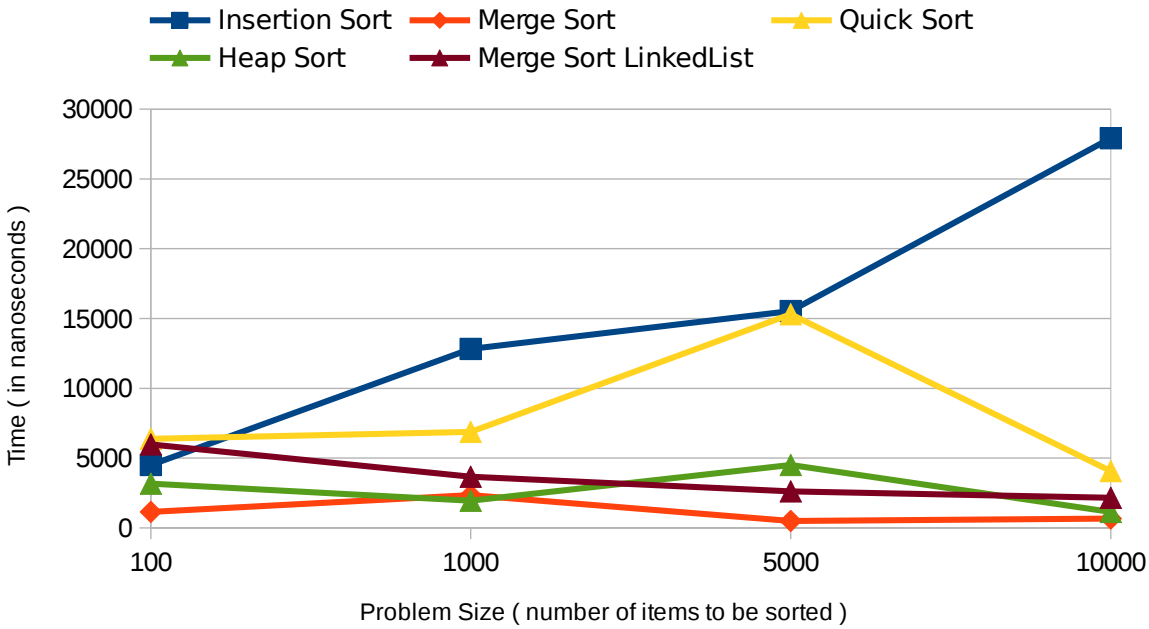

Figure 4.1. Comparison of sorting algorithms

5 Screen Shots

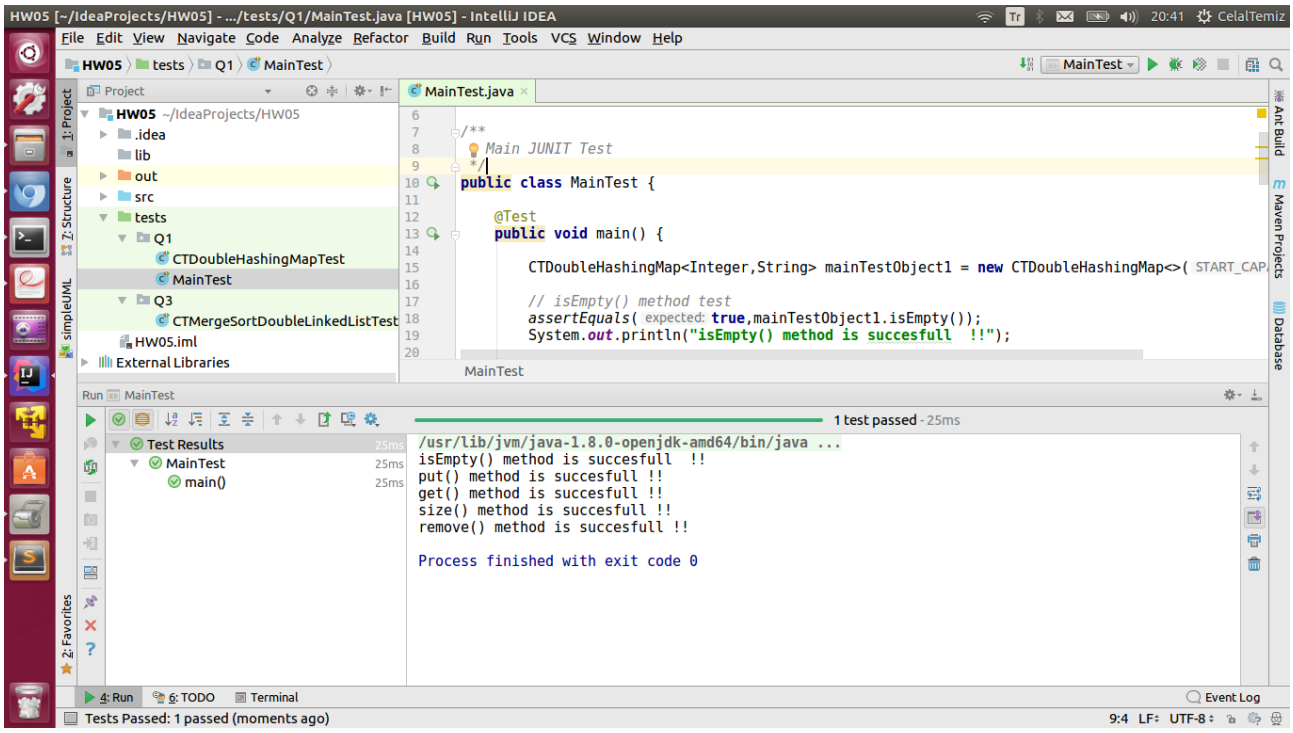

Figure 5.1. Q1 junit main test

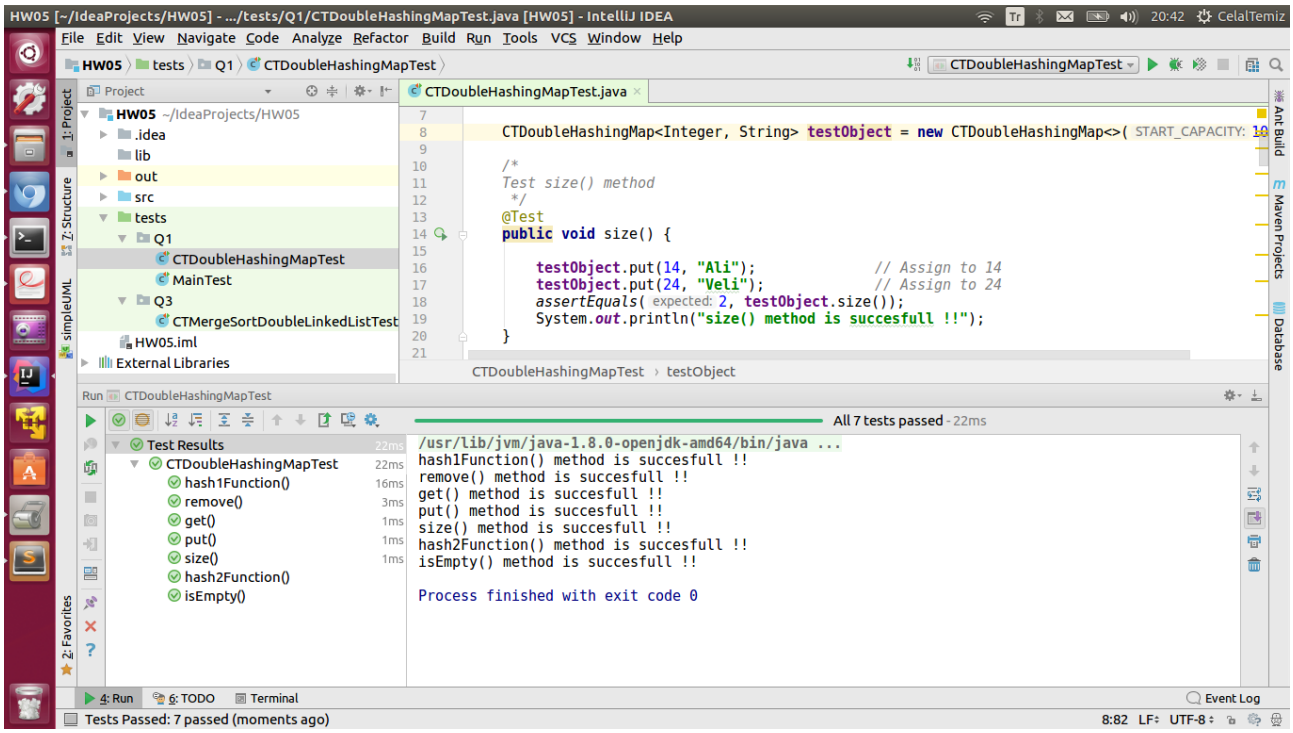

Figure 5.2. Q1 junit unit test

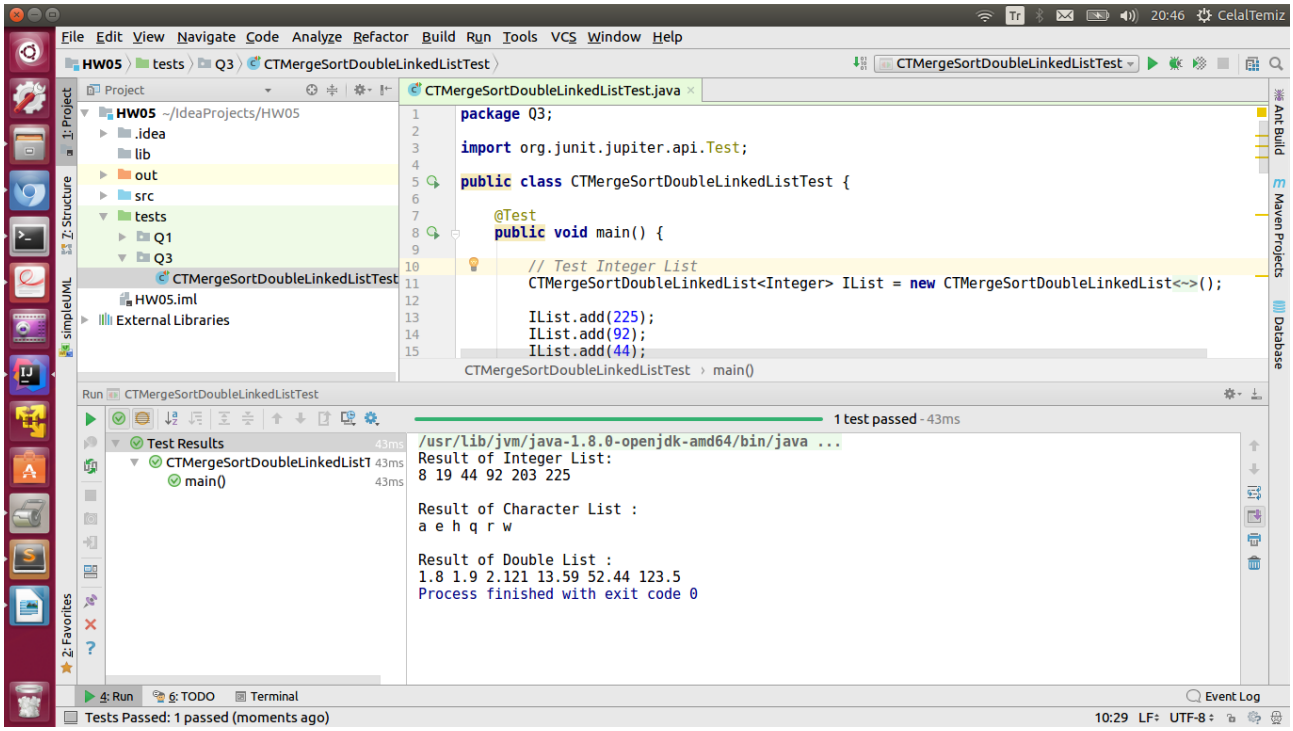

Figure 5.3. Q3 junit main test