Gebze Teknik Üniversitesi
Bilgisayar Mühendisliği

CSE 222- 2018 Bahar

ÖDEV 5 RAPORU

GULZADA IISAEVA
131044085

1 Double Hashing Map

Bu bölüm için iki tane sınıf yazıldı.

Map sınıfı: kitapta verilen metodların implement edilmesi için

Metotları:

```
V get(Object key);
V put(K key, V value);
V remove(Object key);
int size();
boolean isEmpty();
```

HTOpenAddressing sınıfı: Map sınıfını "open addressing" metodunu kullanarak implement eder. Çarpışmalar "double hashing ile çözüldü

1.1 Pseudocode ve Açıklama

HTOpenAddressing sınıfının implement edilmesi:
Kullanılan kaynak: Kitabın HashtableOpen.java sınıfı kullanıldı.

- Ilk olarak Key ve Value'yu tutmak için Entry<K,V> sınıfı yazıldı
  Private datalar:
  - `Entry < K, V > [] table:` Dataları tutan array yapısı
  - `private double LOAD_THRESHOLD = 0.75;` Kullanılacak alan
  - `private int numKeys;` Eklenen eleman sayısı
  - `private int numDeletes;` Silinen eleman sayısı
  - `private final Entry < K, V > DELETED = new Entry < K, V > (null, null);`
- Tablo sayısı asal sayı olarak 101 verildi.
- `V put(K key, V value)`

```
public V put(K key, V value) {
    int hashIndex = doubleHashing(key);
    int stepSize = doubleHashing2(key);

    if (hashIndex < 0)
        hashIndex += table.length;

    while (table[hashIndex] != null
            && (!key.equals(table[hashIndex].key))) {
        hashIndex += stepSize;
        hashIndex %= table.length;
    }
    if (table[hashIndex] == null) {
        table[hashIndex] = new Entry < K, V > (key, value);
        numKeys++;

        double loadFactor =
                (double) (numKeys + numDeletes) / table.length;
        if (loadFactor > LOAD_THRESHOLD){
            rehash();
            return null;}
    }

    V oldVal = table[hashIndex].value;
    table[hashIndex].value = value;
    return oldVal;
}
```

```
private int doubleHashing(Object key) {
    return key.hashCode() %  table.length; }
```

Eklenecek eleman indexi

```
private int doubleHashing2(Object key) {
    return 5 - key.hashCode() % 5; }
```

Eklenecek step sayısı

pozitif yapmak

Eleman boş olana kadar ve Key eşit olmadığı sürece, hashcodeları hesaplayacak. Eğer ilk bulunan index boş değilse step sayısını ekleyip bir daha hashcode bulunacak

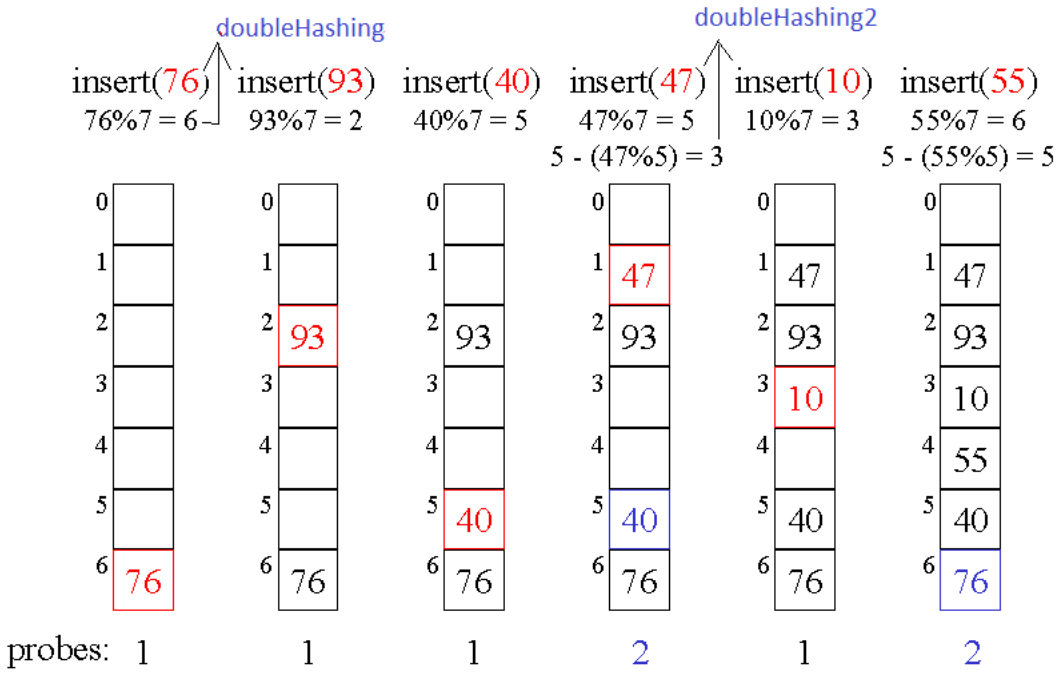
Eğer eleman hiç yoksa, yeni ekleme

Eğer eklenen eleman ve silinen sayılar kullanılabilecek alandan büyükse rehash() yapılacak

Eğer Key varsa Valuesunu değiştirmek

-

- **`rehash()`-** tablonun boyutu iki kat arttırılacak , **`DELETED`** ve null elemenlar hariç kopyalanacak



- `V get(Object key)` metodu için index key'in hashcode u hesaplanarak bulunacak
- `V remove(Object key)`: elemanın hashcode u hesaplanarak index bulunacak,yerine **`DELETED`** koyulacak. Aynı zamanda numKeys azaltılacak,numDeletes arttırılacak

1.2 Testler

```
"C:\Program Files\Java\jdk1.8.0_111\bin\java" ...
############################################################
########  TESTING Q1:DOUBLE HASHING MAP      #############
########  READS COUNTRIES/CAPITALS FROM FILE #############
########  Key: COUNTRY, Value: CAPITAL       #############
############################################################

isEmpty(): true
```

Başta boş

```
Value :K�bil
Value :Berlin
Value :Luanda
Value :Tiran
Value :Canberra
Value :Nassau
Value :Manama
Value :Dhaka
Value :Bridgetown
Value :Br�ksel
Value :Belmopan
Value :Thimphu
Value :Sucre
Value :Gaborone
Value :Brezilya
Value :Sofya
Value :Bujumbura
Value :Cezayir
Value :Cibuti
Value :Roseau
Value :Quito
Value :Jakarta
Value :Asmara
Value :Tallinn
Value :Rabat
Value :Suva
Value :Manila
Value :Vatikan
Value :Karakas
Value :Hanoi
Value :Sanaa
Value :Atina
Value :Lusaka
Value :Harare
```

Key : Ülkeler
Value: Başkentler

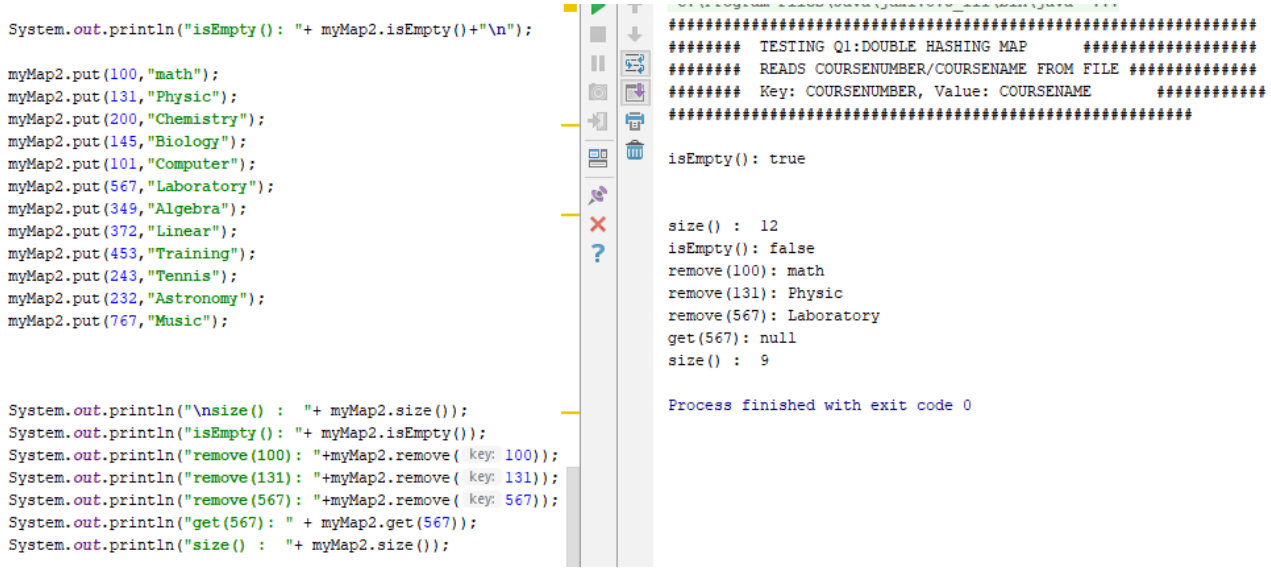
şekilde dosyadan okuyup put ile eklendi

```
isEmpty(): false
size() :  136
remove(Almanya): Berlin
remove(Kenya): Nairobi
remove(Fas): Rabat
get(Fas): null
size() :  133
```

Silinenleri return etmesi lazım

Silindikten sonra

- 2.test için de <Integer,String> tipinde obje oluşturarak put işlemleri ve diğer metotlar test edildi



```
System.out.println("isEmpty(): "+ myMap2.isEmpty()+"\n");

myMap2.put(100,"math");
myMap2.put(131,"Physic");
myMap2.put(200,"Chemistry");
myMap2.put(145,"Biology");
myMap2.put(101,"Computer");
myMap2.put(567,"Laboratory");
myMap2.put(349,"Algebra");
myMap2.put(372,"Linear");
myMap2.put(453,"Training");
myMap2.put(243,"Tennis");
myMap2.put(232,"Astronomy");
myMap2.put(767,"Music");



System.out.println("\nsize() :  "+ myMap2.size());
System.out.println("isEmpty(): "+ myMap2.isEmpty());
System.out.println("remove(100): "+myMap2.remove( key: 100));
System.out.println("remove(131): "+myMap2.remove( key: 131));
System.out.println("remove(567): "+myMap2.remove( key: 567));
System.out.println("get(567): " + myMap2.get(567));
System.out.println("size() :  "+ myMap2.size());
```

```
############################################################
########  TESTING Q1:DOUBLE HASHING MAP      ###############
########  READS COURSENUMBER/COURSENAME FROM FILE ##########
########  Key: COURSENUMBER, Value: COURSENAME       ########
############################################################

isEmpty(): true


size() :  12
isEmpty(): false
remove(100): math
remove(131): Physic
remove(567): Laboratory
get(567): null
size() :  9

Process finished with exit code 0
```

2 Recursive Hashing Set

Bu bölümde Set interfaceini chaining hash table kullanarak implement edilmesi istenildi. Eleman eklerken ortaya çıkan çakışmaları o elemana bağlı farklı boyutta hash table oluşturarak recursive şekilde devam edilmesi istenildi.

2.1 Pseudocode ve Açıklama

Bu bölüm için Java Set interfaceinin kitabımızdaki metodlarını implement etmek için kendim Set interface tanımladım.

- Metotlar:

```
boolean add(E obj);
boolean addAll(Collection<? extends E> coll);
boolean contains(Object obj);
boolean containsAll(Collection<?> coll);
boolean isEmpty();
Iterator<E> iterator();
boolean remove(Object obj);
boolean removeAll(Collection<?> coll);
boolean retainAll(Collection<?> coll);
int size();
void clear();
```

Bu Set interfaceini implement etmek için “HashtableChain.java” sınıfı yazıldı.

- İç sınıf:

```
private static class Table<E> {
    private   E data;
    private int size;
    private Table [] nextTable =null;
```

  Bu sınıf her bir dataya bağlı ayrı tablo oluşması için nextTable değişkeni var. Çarpışma olduğu zaman 101 de küçük olan asal sayıyı bulup o kadarlık yeni hashtable oluşturuluyor.

Asal sayıyı bulmak için:

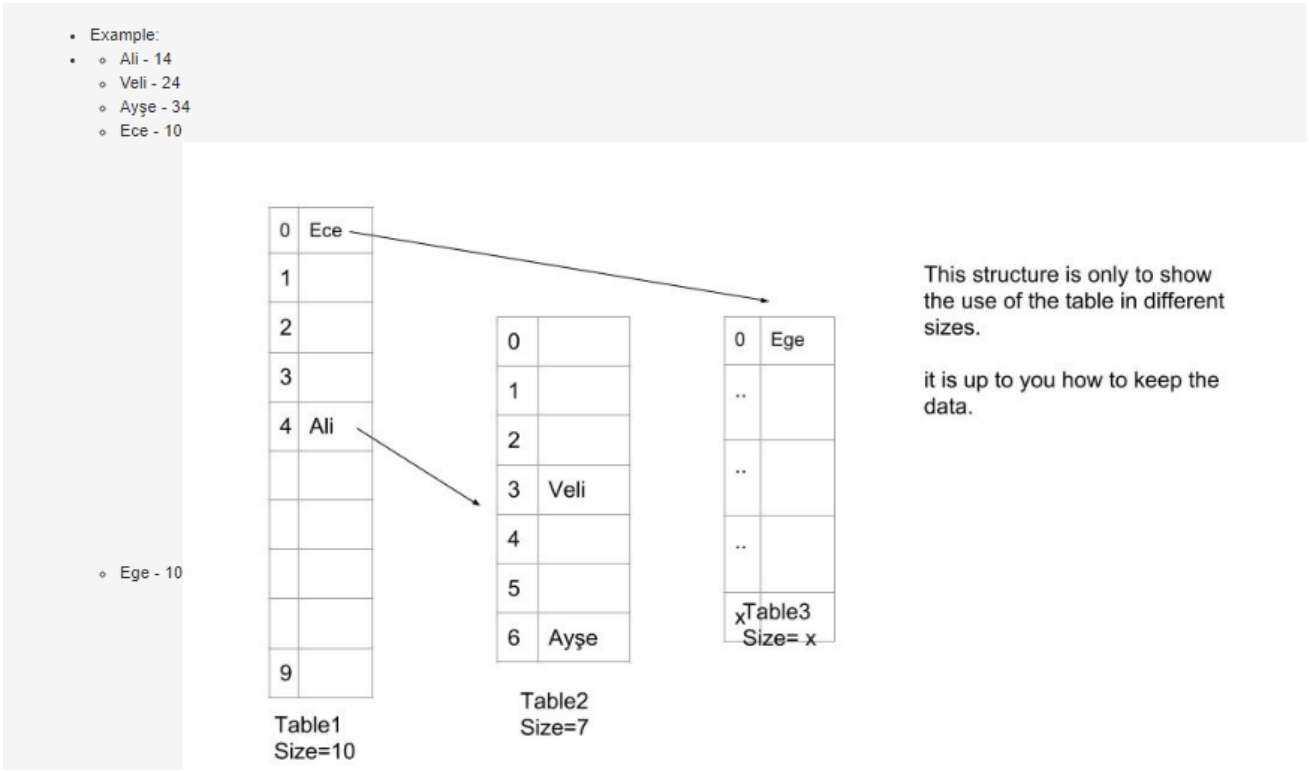
```
private int primeNumber(int n)
private static boolean isPrime(int n)
```

metodları yazıldı.

- Dataları:

```
private Table<E> [] table; Dataları tutacak bir tablo
private int numKeys;       Eklenen eleman sayısı
private static final int CAPACITY = 101; Tablo boyutu
```

Table tipinde array oluşturuldu. Bu şekilde her arrayin datasının tablosu oluşturulabilecek recursive şeklinde tasarlandı. Bu algoritmayı bize verilen örnekten yola çıkarak tasarladım.



İmplement edilen metodlar:

```
@Override
public boolean add(E obj){

 int index = obj.hashCode() % table.length;
 if (index < 0)
     index += table.length;

 if (table[index] == null) {
     numKeys++;
     table[index] = new Table<E>(obj);
     return true;
 }

 if(table[index].getData().equals(obj))
     return false;


 addRecursive(table[index].nextTable,primeNumber( n: table.length-1),obj);

 return true;
}
```

Hashcode hesaplama

Pozitif yapmak

Eğer bulunan index boş ise eleman eklenir

Set olduğu için eğer verilen data önceden varsa false döndürür

indexi bulana kadar tüm tablolara recursive şekilde bakacak ve ekleyecek

```
public void addRecursive(Table<E> []node, int size,E obj)
{
  int index = obj.hashCode() % size;

  if(node==null)
  {
      node=new Table[size];
  }
  if(node[index].getData()==obj){
      int newSize=primeNumber( n: size-1);
      addRecursive(node[index].nextTable,newSize,obj);
  }
  else{
      node[index].setData(obj);
      numKeys++;
  }
}
```

hashcode hesaplama

Eğer gelen tablo null ise yer ayırılacak

Eğer gelen tablodaki index de boş değilse ona bağlı tablo tekrardan gönderilecek recursive şekilde bakılacak

Eğer index boş ise eleman eklenecek, eklenen eleman sayısı arttırılacak

- **`boolean`** `addAll(Collection<?` **`extends`** `E> c)` : add metodu kullanarak gelen collection-onun tüm elemanları eklenecek

```java
@Override
public boolean contains(Object obj){
    int index = obj.hashCode() % table.length;
    if(table[index]==null)
        return false;
    if(table[index].getData().equals(obj))
        return true;
    else
        return containsHelper(table[index].nextTable,obj,primeNumber( n: table.length-1));

}

public boolean containsHelper(Table<E>[] node, Object obj,int size){
    int index = obj.hashCode() % size;
    if(node[index]==null)
        return false;
    if(!node[index].getData().equals(obj))
        containsHelper(node[index].nextTable,obj,primeNumber( n: size-1));

    return true;
}
```

index boş ise false

indexteki elemanla aynı ise true

İndexte başka eleman varsa ona bağlı tablo gönderilir

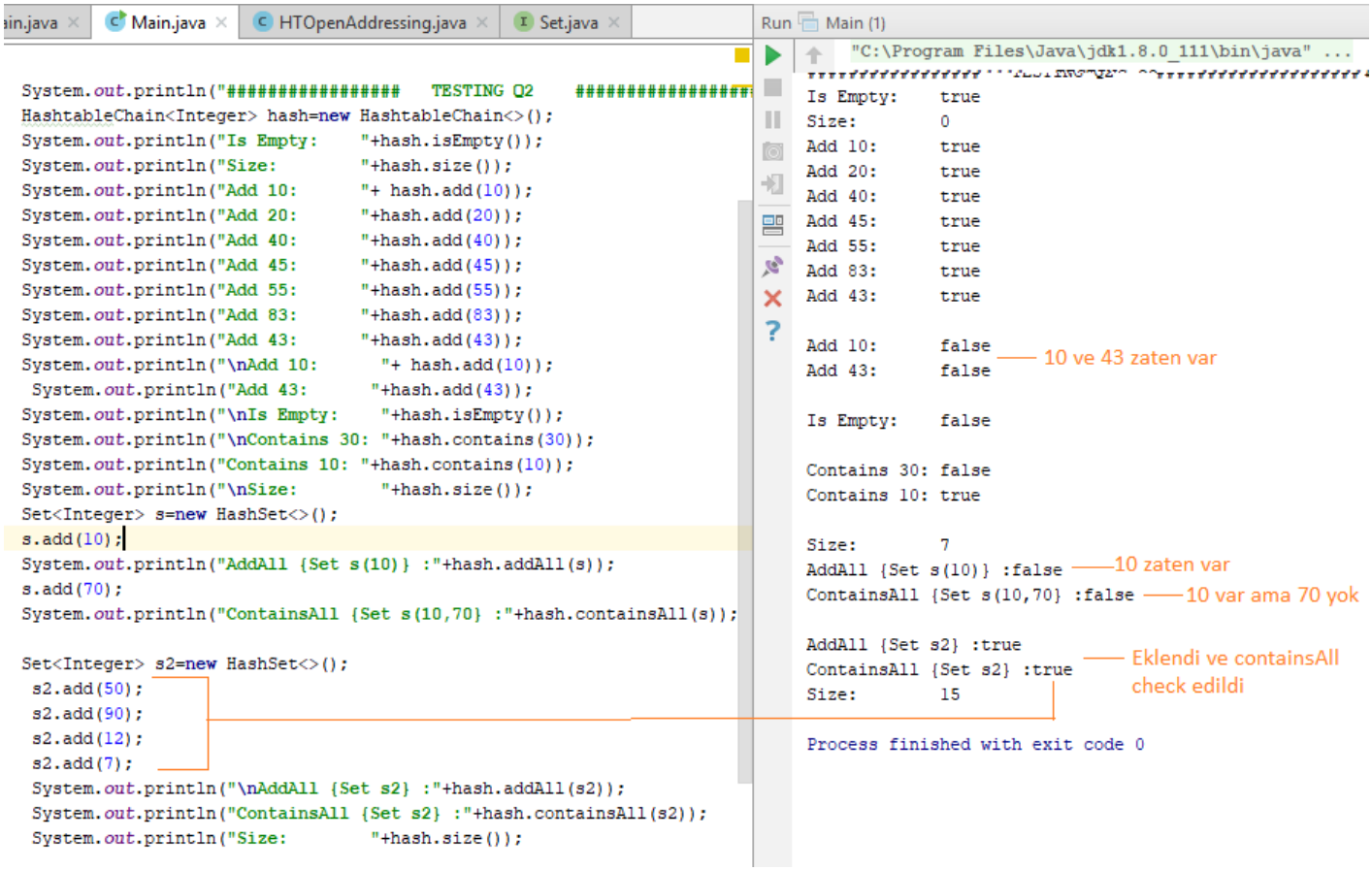
Eğer index boş ise false

Eğer indexte başka eleman varsa recursive şekilde devam edecek

sağlamıyorsa eşit ve true

- **boolean** containsAll(Collection<?> c) : contains() metotunu kullanarak tüm elemanlara bakılacak.
- **int** size(){**return numKeys**;} : Eklenen eleman sayısı
- **boolean** isEmpty(): size() sıfır ise true, değilse false

2.2 Testler



3 Sıralama Algoritmaları

3.1 MergeSort with DoubleLinkedList

Bu bölümde DoubleLinkedList i sıralayan Merge sort yazılması istenildi.
Bunun için :

DoubleLinkedList.java
Node.java
MergeSortForDLL.java

sınıfları yazıldı

3.1.1 Pseudocode ve Açıklama

- Node.java

```
public class Node<E
        extends Comparable < E >> {
    public E data;
    public Node<E> prev;
    public Node<E> next;
```

Hem başlangıcı hem kuyruğu tutan node yapısı

- DoubleLinkedList.java

```
public Node<E> root;
```

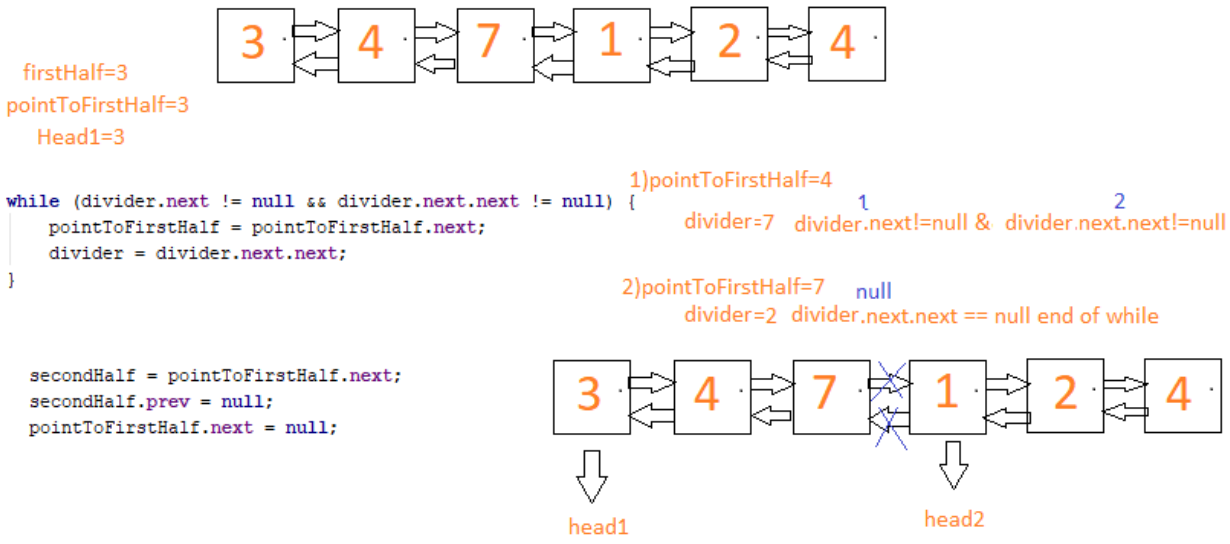
Merge sort için listeye eleman eklemesi yeterli olduğu için sadece `void add(E data) ve toString()` metodu yazıldı.

`void add(E data)`: Eğer boş root boş ise roota ekler, değilse rootun nexti boş olana kadar dolaşır ve ekler. Her ekledikten sonra prev ve next değişir

- MergeSortForDLL.java

`void sort(DoubleLinkedList<E> dll)`: sort metodu DoubleLinkedList alır.

DoubleLinkedListi null olana kadar recursive olarak ikiye böler. En son karşılaştırır ilk eleman diğerinden büyükse yer değiştirir , tekrar recursive olarak merge eder.



3.1.2 Ortalama Çalışma Süresi

Bu bölüm için 10 farklı boyutta (200,500,1000,1500,2000,2500,3000,3500,4000,4500) array oluşturdum ve random sayılar ürettim.

Sorts(parametreler) metodu yazıldı. 200 lük arraye random sayı üretererek, aynı anda aynı boyutta başka bir array göndererek tüm algoritmayı aynı metod içinde 10 kere denedim. Başka boş array göndermenin sebebi de orijinal random array değişmemesi için ve tüm algoritmaya aynı arrayi göndermek için boş arraye kopyalayıp bu array sort edildi

```
startTime = System.nanoTime();
merge.sort(sort);
```

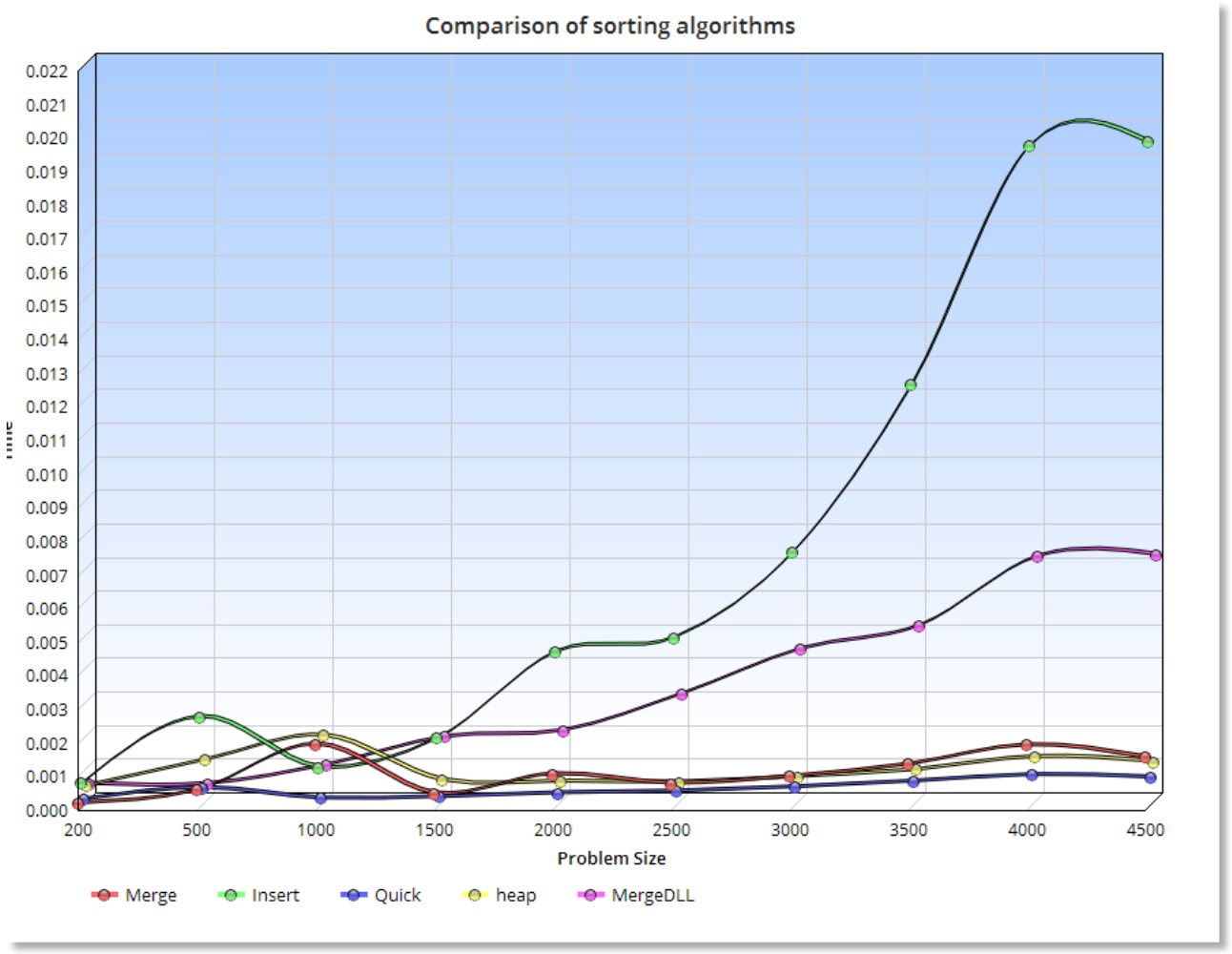
```
endTime = System.nanoTime();
duration = (endTime - startTime);
```

Bu şekilde çalışma zamanı bulundu. Mu sort metod diğer boyuttaki arrayler için de çağırıldı. Her algoritmanın çalışma zamanı bir vectorde tutularak en son işlem bittikten sonra algoritmaların her dizi boyutu için bulunan zamanları toplayıp 10a bölerek nano cinsinden buldum.

```
"C:\Program Files\Java\jdk1.8.0_111\bin\java" ...
##################   TESTING Q4    ######################


#####################   Array size: 200  ##################################

Average time of Merge Sort:      264835    nanosecond
Average time of Insert Sort:     1107936     nanosecond
Average time of Quick Sort:      381255    nanosecond
Average time of Heap Sort:       1211460    nanosecond
Average time of MergeDLL Sort:   772908    nanosecond


#####################   Array size: 500  ##################################

Average time of Merge Sort:      3178694    nanosecond
Average time of Insert Sort:     2872583     nanosecond
Average time of Quick Sort:      614141    nanosecond
Average time of Heap Sort:       1144526    nanosecond
Average time of MergeDLL Sort:   489421    nanosecond


#####################   Array size: 1000  ##################################

Average time of Merge Sort:      3063479    nanosecond
Average time of Insert Sort:     1211281     nanosecond
Average time of Quick Sort:      179695    nanosecond
Average time of Heap Sort:       1332031    nanosecond
Average time of MergeDLL Sort:   1181340    nanosecond
```

Array Sorting Algorithms

Algorithm	Time Complexity
	Average
Quicksort	O(n log(n))
Mergesort	O(n log(n))
Heapsort	O(n log(n))
Insertion Sort	O(n^2)



3.1.3 En Kötü Durum Performans Analizi

Sort algoritmasnın en kötü zamanda çalışabilecek durumu bir arrayin tersten sıralı olmasıdır. Bunu test etmek için 4 farklı boyutta (100,1000,5000,10000,15000) array oluşturdum. for döngüsü ile size dan sıfıra kadar sayıları ekledim. Tersten sıralı sayılar eklenmiş oluyor. Her bir arrayi tüm algoritmalar için denedim.

```
##############   Size of array: 100 ######################
Time of Merge  Sort:    349395
Time of Insert Sort:    1227570
Time of Quick Sort:     580094
Time of Heap Sort:      306558
Time of MergeDLL Sort:  200356

##############   Size of array: 1000 ######################
Time of Merge  Sort:    1854964
Time of Insert Sort:    16804893
Time of Quick Sort:     14792411
Time of Heap Sort:      2298067
Time of MergeDLL Sort:  641228

##############   Size of array: 5000 ######################
Time of Merge  Sort:    4300732
Time of Insert Sort:    124127292
Time of Quick Sort:     100619184
Time of Heap Sort:      16034706
Time of MergeDLL Sort:  1630512

##############   Size of array: 10000 ######################
Time of Merge  Sort:    3788017
Time of Insert Sort:    257221954
Time of Quick Sort:     130660495
Time of Heap Sort:      6668410
Time of MergeDLL Sort:  780896

##############   Size of array: 15000 ######################
Time of Merge  Sort:    3050405
Time of Insert Sort:    442441234
Time of Quick Sort:     395984139
Time of Heap Sort:      36021194
Time of MergeDLL Sort:  1863442
```

3.2 MergeSort

3.2.1 Ortalama Çalışma Süresi

Bölüm 3.1.2 de anlatıldı

3.2.2 En Kötü Durum Performans Analizi

Bölüm 3.1.3 de anlatıldı

3.3 Insertion Sort

3.3.1 Ortalama Çalışma Süresi

Bölüm 3.1.2 de anlatıldı

3.3.2 En Kötü Durum Performans Analizi

Bölüm 3.1.3 de anlatıldı

3.4 Quick Sort

3.4.1 Ortalama Çalışma Süresi

Bölüm 3.1.2 de anlatıldı

3.4.2 En Kötü Durum Performans Analizi

Bölüm 3.1.3 de anlatıldı

3.5 Heap Sort

3.5.1 Ortalama Çalışma Süresi

Bölüm 3.1.2 de anlatıldı

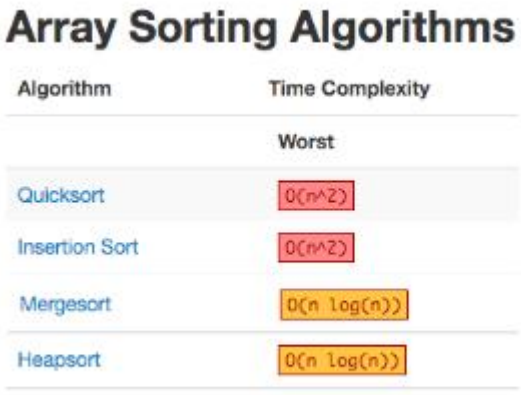
3.5.2 En Kötü Durum Performans Analizi

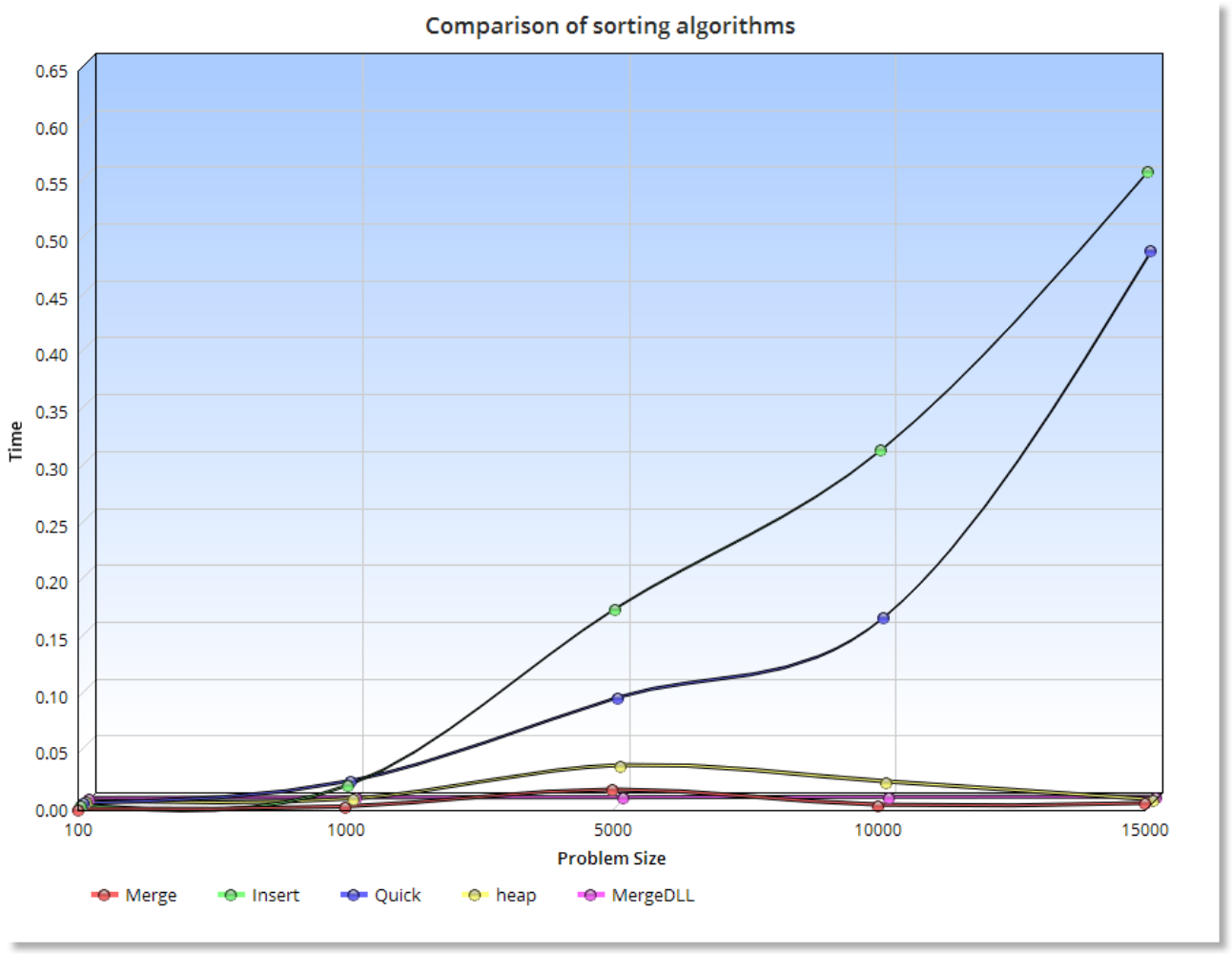
Bölüm 3.1.3 de anlatıldı

4 Analiz Sonuçlarının Karşılaştırılması

Bölüm 3.1.3 deki ekran çıktısındaki değerler nano cinsinden olduğu için saniyeye çevirdim. Oluşturduğum grafik algoritmaların worst-case complexityne göre doğru çıktı

X Axis	Y Axis	Y Axis	Y Axis	Y Axis	Y Axis
100	0.000594823	0.000975011	0.000788041	0.000304774	0.000278001
1000	0.002499777	0.018950469	0.020400267	0.002358768	0.000905846
5000	0.017437751	0.174228916	0.09372423	0.031684962	0.000995538
10000	0.004034806	0.314230701	0.164754127	0.017188309	0.001406961
15000	0.005481035	0.559186524	0.48658456	0.001406961	0.001572512




Comparison of sorting algorithms
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Time
100
1000
5000
10000
15000
Problem Size
Merge
Insert
Quick
heap
MergeDLL