Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

SERKAN SORMAN
151044057

Course Assistant: Fatma Nur ESİRCİ

1 Double Hashing Map

This part about Question1 in HW5

1.1 Pseudocode and Explanation

HashMap interfacesi implement edilerek, put,get,remove,size,isEmpty metodları override edildi ve helper metodlar kullanıldı. Hash tablosunda tutulacak elemanlar için Entry classı kullanıldı. Bu class her elemana ait key ve value değerlerini içeriyor. Entry referans tipinde bir array oluşturuldu ve hash tablosu olarak kullanıldı. Tablonun kapasitesi 5 den büyük olacak şekilde constructur ile alınarak set edildi. find() metodu kullanılarak istenen keye ait index veya tabloda bulunan boş bir index değeri elde edildi. find() metodu içinde yapılan arama işlemi sırasında collision ile karşılaşıldığında ikinci hash metodu olan secondHash() metodu ile yeni bir index oluşturuldu ve collision engellendi.

put() metodunda verilen key,value değeri ile bir Entry oluşturulup find() metodu ile elde edilen index değeri kullanılarak tabloya eklendi. Tablonun doluluk oranı sınırı aşılmış ise tablo boyutu rehash() metodu ile büyütüldü. put() metodu ile tabloda daha önce bulunan bir indexe yeni bir value set edildiğinde yeni value set edilerek eski value return edildi.

Remove() metodu ile verilen keye ait Entry boolean isDeleted ile belirtilerek key ve value ya null atandı böylece bu keye ait Entry değeri tablodan kaldırılmış oldu. Rehash() metodu ile tablo boyutu büyütülüp deletedlar referansları hariç tüm Entryler tabloya yeniden eklendi. showTable() metodu ile tüm tablo ekranda gösterildi. isEmpty() metodu ile tablodaki key sayısına göre tablonun boş olup olmadığı, size() metodu ile tabloda bulunan key sayısı elde edildi.

```
find(key)
        index = key's hashcode % tableSize
        if index less than zero
                index = index + tableSize
        while table[index] != null AND key != key of table[index]
                index = secondHash(index,key)
        return index
```

```
secondHash(index,key)
        index = index + (5 – key's hashcode % 5 )
        return index % tableSize


get(key)
        index = find(key)
        if(table[index] != null)
                return value of table[index]
        else
                return null


put(key,value)
        index = find(key)
        if(table[index] equals to null)
                Construct new Entry for table[index] with key and value
                İncrement numberOfKeys by One
                Calculate loadFactor
                if  loadFactor  > loadTheresold
                        rehash()
                return null
        set value to Entry
        return oldValue of Entry


remove(key)
        index = find(key)
        if(table[index] equals to null)
                return null
        table[index] set deleted
        decrement numberOfKeys by One
        increment numberOfDeletes by One
        return value of table[index]
```

size()

return numberOfKeys

isEmpty

return size equals to zero

rehash()

oldTable = table

Increment capacity of table

numberOfKeys = 0

numberOfDeletes = 0

for i=0 to tableSize

if oldTable[i] != null and oldTable != deleted

put Entry to table

1.2 Test Cases

7 Sizelık birinci tabloya key 14 index 0a eklendi ve ardından key 21 eklenerek 0 ıncı indexte collision gerçekleşmesi sağlandı. Double hashing yapılarak 0 +(5 – 21 % 5) ile key 21 index 4e yerleştirildi. Ardından key 15 tabloya eklenerek index 1e yerleştirildi. Key 29 eklenerek index 1de tekrardan collision oluşturuldu ve key 29 index 2 ye eklendi. Key 12 collision olmadan index 5 e yerleştirildi.

```
########### Table One ##########

INDEX KEYS VALUES
--------------------
0 ==> [14,Ali]
1 ==> [15,Ayse]
2 ==> [29,Burak]
3 ==> null
4 ==> [21,Fatma]
5 ==> [12,Mert]
6 ==> null

Table size: 5

Key 15 is removed
Key 29 is removed

INDEX KEYS VALUES
--------------------
0 ==> [14,Ali]
1 ==> DELETED
2 ==> DELETED
3 ==> null
4 ==> [21,Fatma]
5 ==> [12,Mert]
6 ==> null

Table size: 3
```

Key 33 tabloya eklenerek tablo doluluk sınırı aşıldı ve rehash işlemi gerçekleştirildi. Tüm entryler tekrardan 15 sizelık tabloya collision gerçekleşmeden eklendi.

```
Key 33 is added and Table is rehashing...

INDEX KEYS VALUES
--------------------
0 ==> null
1 ==> null
2 ==> null
3 ==> [33,Mustafa]
4 ==> null
5 ==> null
6 ==> [21,Fatma]
7 ==> null
8 ==> null
9 ==> null
10 ==> null
11 ==> null
12 ==> [12,Mert]
13 ==> null
14 ==> [14,Ali]

Table size: 4
```

Tüm entryler tablodan silindi ve DELETED olarak gösterildi. Ardından tablonun boş olup olmadığı check edildi.

```
Key 21 is removed
Key 12 is removed
Key 14 is removed
Key 33 is removed

INDEX KEYS VALUES
--------------------
0 ==> null
1 ==> null
2 ==> null
3 ==> DELETED
4 ==> null
5 ==> null
6 ==> DELETED
7 ==> null
8 ==> null
9 ==> null
10 ==> null
11 ==> null
12 ==> DELETED
13 ==> null
14 ==> DELETED

Table size: 0
Table is empty !!!
```

11 sizelık ikinci tabloya key 25 index 3 e eklendi. Daha sonra key 47 eklenmeye çalışılarak index 3te collision gerçekleştirildi. Double hashing kullanılarak 3 += (5- 47 % 5) ile key 47 index 6 ya eklendi. Key 12 index 1e eklendi. Key 56 eklenirken index 1 de collision gerçkleşti ve 1+= (5-56%5) ile index 5 e yerleştirildi.Key 36 index 3 e eklenirken collision gerçekleşti ve 3+=(5-36%5) ile index 7 ye eklendi. 33 index 0 a eklendi. 22 index 0 a eklenirken collision gerçekleşti ve 0+=(5-

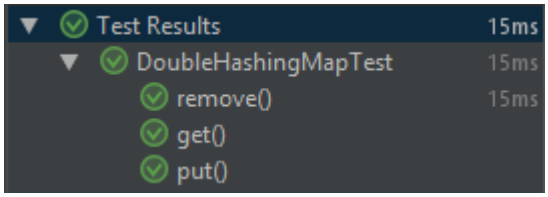
22%5) ile index 3 e eklenmeye çalışıldı fakat tekrar collision meydana geldi. 3+=(5-22%5) ile index 6 da tekrar collision gerçekleşti. 6 += (5 – 22 % 5) ile key 22 index 9 a eklendi.

```
INDEX KEYS VALUES
--------------------
0 ==> [33,F]
1 ==> [12,C]
2 ==> null
3 ==> [25,A]
4 ==> null
5 ==> [56,D]
6 ==> [47,B]
7 ==> [36,E]
8 ==> null
9 ==> [22,G]
10 ==> null

Table size: 7

Key 22 is removed
Key 77 is added
New value added to key 47

INDEX KEYS VALUES
--------------------
0 ==> [33,F]
1 ==> [12,C]
2 ==> null
3 ==> [25,A]
4 ==> [77,X]
5 ==> [56,D]
6 ==> [47,Z]
7 ==> [36,E]
8 ==> null
9 ==> DELETED
10 ==> null

Table size: 7
```

Key 22 remove edilerek key 77 tablodaki 4. indexe eklendi. Böylece remove edilen indexe(isDeleted = true) yeni bir keyin eklenemeyeceği test edilmiş oldu. Key 47 ye yeni bir value set edildi ve doğrulandı.

Unit testler

Unit testlerde get,put ve remove metodların karşılaşabileceği durumlar test edildi ve doğrulandı.



2 Recursive Hashing Set

This part about Question2 in HW5

2.1 Pseudocode and Explanation

HashSet interfacesi implement edilerek add,contains,remove,size,isEmpty metodları override edildi ve helper metodlar kullanıldı. Hash tablosunda tutulacak elemanlar için Entry classı kullanıldı. Bu class her elemana ait key ve value değerlerini içerir ayrıca içinde collision durumunda oluşturulmak üzere farklı bir hash tablosu tutar. Bu hash tablosu da classda bulunan Entry referans tipindeki array ile aynı yapıdadır. Oluşturulacak tabloların sizelarının belirlenmesi için prime numberlardan oluşan bir Queue veri yapısı kullanıldı. Tablonun kapasitesi 5 den büyük eşit olacak şekilde constructur ile alınarak set edilebilir.

Add() metodunda parametre ile alınan key daha önce tabloda yok ise ve oluşturulan index boş ise eklenir. Aksi durumda yani collision gerçekleştiğinde Queueda bulunan ilk prime number değeri pool edilerek bu sizeda collision gerçekleşen indexteki Entrye bağlı yeni bir hash tablosu oluşturulur ve yeni tabloya collision gerçekleşen key eklenir. Aynı indexe yeni collisionlar olduğunda keyler bu bağlı hash tablosuna eklenir. Bu işlem recursive bir şekilde iç içe hash tabloları oluşturularak devam ettirilir. Elde edilen hash tabloları birbirinden farklı sizelarda ve collisionu engellemeye yönelik bir şekilde oluşturulur.

Remove() metodunda parametrede verilen key tabloda var ise bu keye sahip Entrynin isDeleted ı true yapılır,key ve valuesine null atanır. Eğer kendsine bağlı bir hash tablosu varsa bu referansa aktarılır. Böylece silinen keye bağlı olan tablo korunmuş olur.

Contains() metodu ile verilen key tabloda bulunuyorsa ve varsa bağlı tabloları içinde recursive bir şekilde aranır ve boolean return edilir. Ayrıca ana hash tablosunu ve buna bağlı metodların gösterilmesi için showTable() ve showNextTable() metodları kullanıldı. Belirtilen bu tüm metodlar wrapper bir metod içinde recursive olarak gerçekleştirilir.

```
add(key)
    if key in the table
        return false
    return add(key,table)


add(key,nexTable)
    index = hashcode of keys % nextTableSize
    if(index < 0)
        index = index + nextTableSize
```

```
    if(table[index] equals to null)
        Construct entry for  table[index]
        İncrement numberOfKeys by one
        if(numberOfKeys > final capacity
            rehash()
        return true
    else if (nextTable of table[index] equals to null)
        Construct table with first prime number of queue as a size
    Add(key, nextTable of table[index])



remove(key)
    return Is key in the table AND remove(key,table)


remove(key,table)
    if key of table[index] equals to key
        set delete to entry
        set null to key and value
        decrement numberOfDeleted by one
        return true
  return remove(key, nextTable of table[index])


contains(key)
     return contains(key,table)


contains(key,table)
    index = hashcode of keys % tableSize
     if(index < 0)
        index = index + tableSize
    if(table[index] equals to null)
        return false
```

```
else if( table[index] != deleted AND  key of table[index] equals to key)
        return true
else if(nextTable of table[index] equals to null)
        return false
return contains(key, nextTable of table[index])
```

```
rehash()
    oldTable = table
    Increment capacity of table
    numberOfKeys = 0
    for i=0 to tableSize
        if oldTable[i] != null and oldTable != deleted
            table[i] = oldTable[i]
```

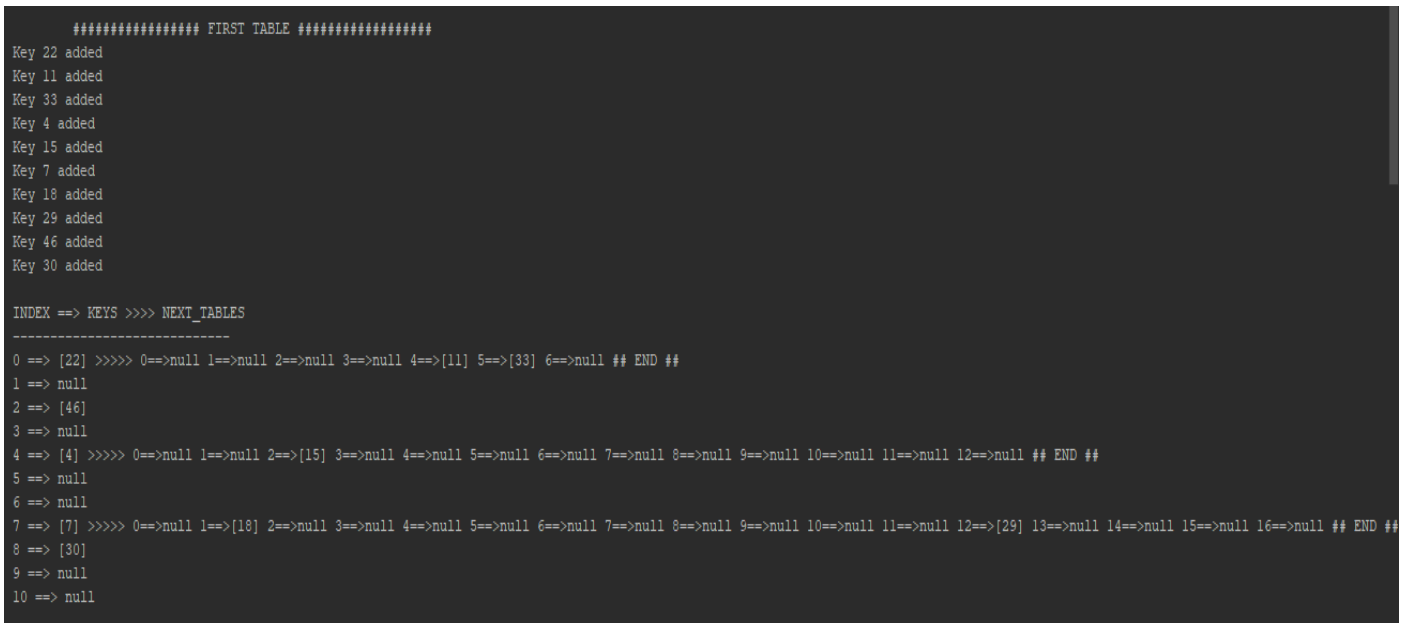
2.2 Test Cases

Key 22 index 0 a eklendi. Key 11 index 0a eklenirken collision oluştu ve Key 11 e bağlı 7 sizeda yeni bir hash tablosu oluşturuldu (>>>> yeni tabloya olan bağı ifade eder). Key 11 bu yeni tablonun 4. İndexine eklendi. Key 33 index 0 a eklenirken collision oluştu ve key 33 index 0 a bağlı olan hash tablosundaki index 5 e eklendi.

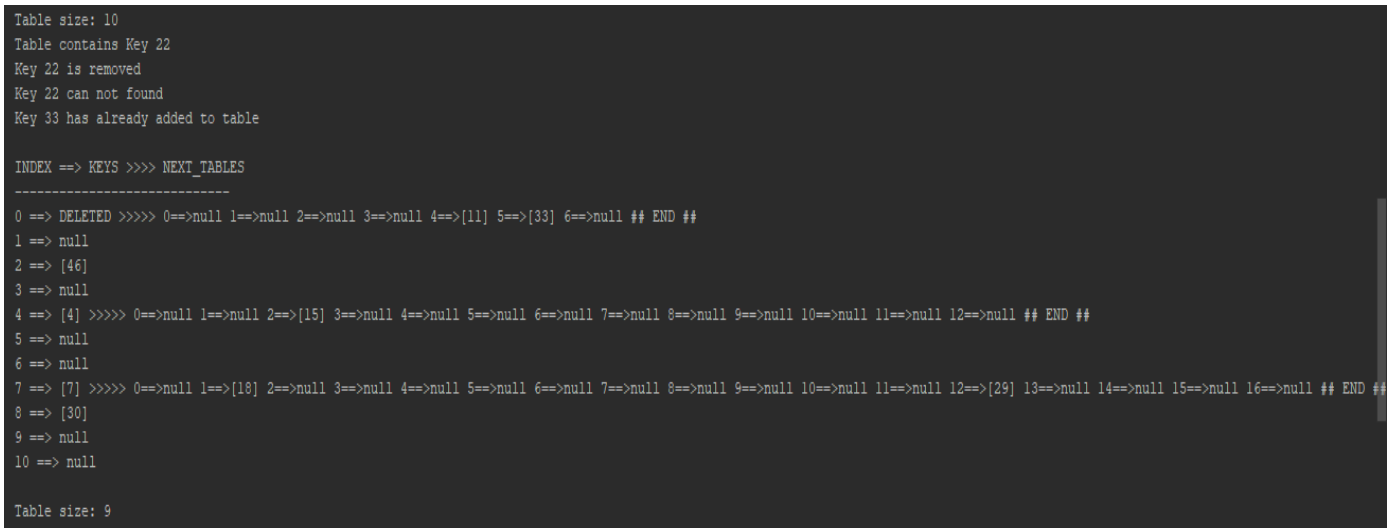
Key 4 ün index 4 e eklenmesi ardından key 15 eklenirken index 4 te collision gerçekleşti ve Key 4 e bağlı 13 sizelı tablo oluşturulup key 15 bu tablodaki index 2 ye eklendi.

Key 7 index 7 ye eklendi.Key 18 ile collision yaşandı ve key 7 ye bağlı 17 sizelı hash tablosu oluşturulup index 1 e eklendi.

Key 29 eklenirken collision oldu ve key 7 ye bağlı tablonun index 12sine eklendi.Key 46 index 2 ye ve key 30 index 8e eklendi.



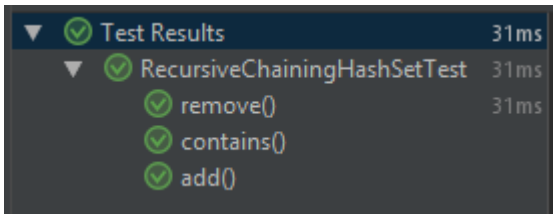
Tablodaki key 22 check edildi. Key 22 silindi ve tekrar check edildi. Tabloda var olan key 33 tekrar tabloya eklenmeye çalışıldı. Key 22 nin silinmesinin ardından hala ona bağlı olan hash tablosu gösterildi.



Tablo 2 de Tablo 1 den farklı olarak iç içe olan hash tablolarında collision gerçekleşme durumu test edildi. Key 12 nin eklenmesinin ardından Key 22,42 eklenerek üst üste collisionlar yapıldı ve Key 12 ye bağlı 7 sizelık bir tablo oluşması sağlandı. Ardından Key 112 eklenmeye çalışıldı Key 12 ile collision olduğu görüldü ve ona bağlı olan tabloya eklenmeye çalışıldı fakat burada da 0. İndexteki key 42 ile collision gerçekleşti ve görüldüğü gibi key 42 ye bağlı bir hash tablosu oluşturulup key 112 bu tablonun index 2 sine eklendi. Görüldüğü gibi 0➔[42] >>>>> 0➔null ile yeni bir tablo başlangıcı belirtilmiş ve bu tablonun sonu ## END ## ile gösterilip bu tabloyu kapsayan tablo elemanlarından devam edilmiştir. ## END ## 1 ➔[22] şeklinde.

```
       ################ SECOND TABLE #################
Key 12 added
Key 22 added
Key 42 added
Key 112 added
Key 15 added
Key 20 added
Key 10 added
Key 9 added
Key 24 added
Key 64 added

INDEX ==> KEYS >>>> NEXT_TABLES
----------------------------
0 ==> [15] >>>>> 0==>null 1==>null 2==>null 3==>null 4==>null 5==>null 6==>null 7==>[20] 8==>null 9==>null 10==>[10] 11==>null 12==>null ## END ##
1 ==> null
2 ==> [12] >>>>> 0==>[42]  >>>>> 0==>null 1==>null 2==>[112] 3==>null 4==>null 5==>null 6==>null 7==>null 8==>null 9==>null 10==>null ## END ## 1==>[22] 2==>null 3==>null 4==>null 5==>n
3 ==> null
4 ==> [9] >>>>> 0==>null 1==>null 2==>null 3==>null 4==>null 5==>null 6==>null 7==>[24] 8==>null 9==>null 10==>null 11==>null 12==>null 13==>[64] 14==>null 15==>null 16==>null ## END ##

Table size: 10
```

Unit Tests



3 Sorting Algortihms

3.1 MergeSort with DoubleLinkedList

This part about Question3 in HW5

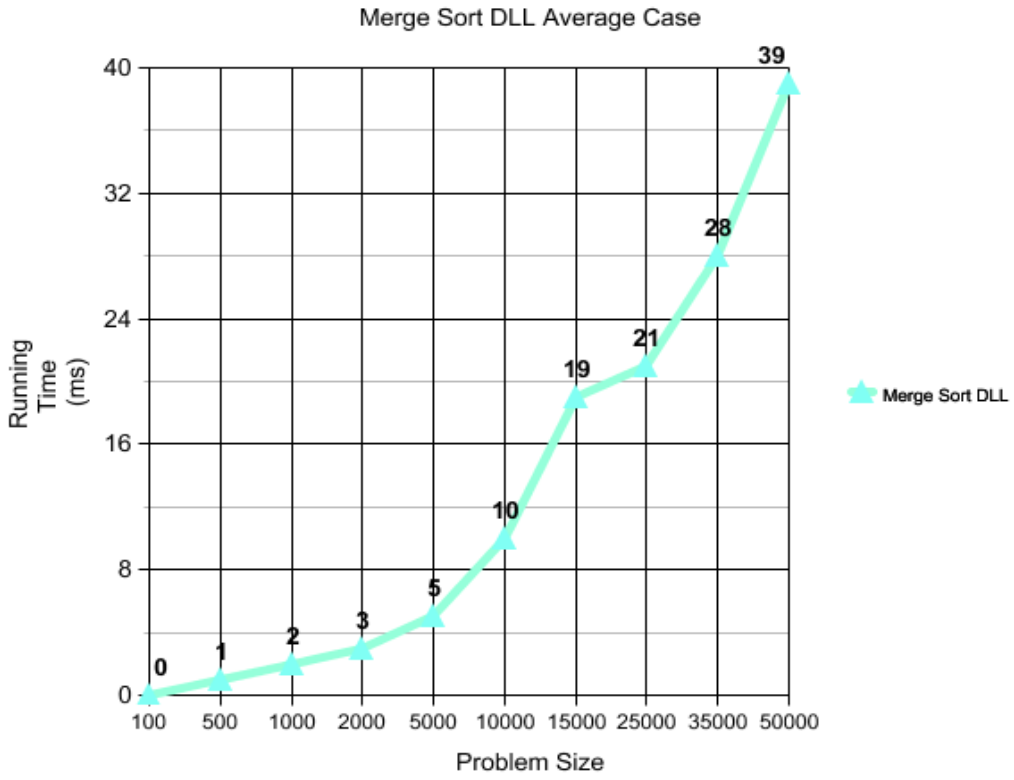
3.1.1 Pseudocode and Explanation

Merge sort algoritması generic Double Linked List üzerinde test edildi. Birleştirme ve sıralama işleminin yapıldığı iki ayrı metod kullanıldı. Sort() metodunda öncelikle gelen liste ikiye bölündü ve bu iki liste kendi içlerinde sıralanmak üzere sort() metoduyla recursive call yapıldı. Listenin ayrı iki listeye aktarılması işlemi sırasında liste üzerinde constant bir şekilde gezebilmek için ListIterator yapısı kullanıldı. Ayrılan tüm listeler 1 eleman kalıncaya dek bölündü ve ardından merge() metodu ile gelen listeyi oluşturan her iki listenin elemanları listelerden silinip karşılaştırılıp ana listeye eklendi ve sonuç olarak listenin sıralı hali elde edilmş oldu.

```
Merge(list, leftList ,rightList)
        while leftList is not empty AND rightList is not empty
                if  first element of leftList <  if  first element of rightList
                        add first element of leftList to list
                        remove first element of leftList
                else
                        add first element of righttList to list
                        remove first element of rightList

        while leftList is not empty
                add first element of leftList to list
                remove first element of leftList

        while rightList is not empty
                add first element of righttList to list
                remove first element of rightList




Sort(list)
        İf listSize > 1
                Construct leftList
                Construct rightList
                Construct list iterator
                halfSize = listSize / 2
                for i = 0 to halfSize
                        add next of list iterator to leftList
                while list iterator has next element
                        add next of list iterator to rightList

                clear the list
                sort(leftList)
                sort(rightList)
                merge(list,leftList,rightList)
```

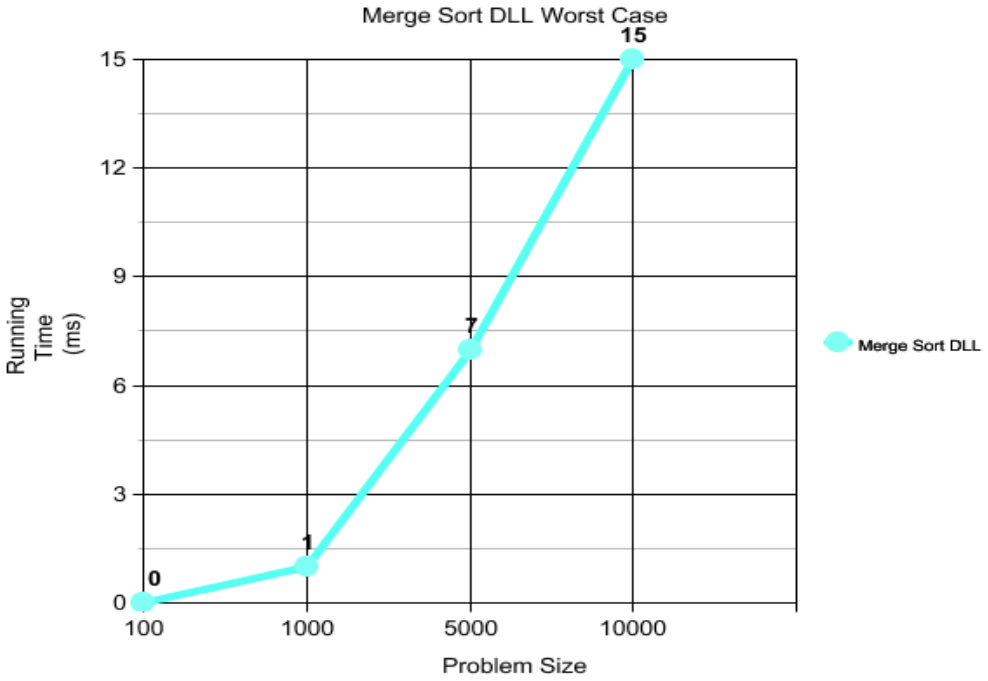
3.1.2 Average Run Time Analysis

Sıralanacak n boyutlu rastgele dizili DLL nin 1 elemanlık listeler haline dönüşünceye kadar yapılan ayırma işlemi T(n) = θ (logn) süre alırken. Ayrılmış bu listelerin sıralanıp birleştirilme işlemi T(n) = θ (n) zaman alır. Sonuç olarak Average T(n) = θ (nlogn) elde edilir bu zaman listenin nasıl dizili olduğundan bağımsız olarak bu şekilde çıkar. Constant time da elemanlara erişim için get() metodu yerine iterator kullanılmıştır. Ayrıca yine constant time olan ve listenin ilk elemanını silen remove() metoduna ek olarak constant timeda ekleme yapan add metodu kullanılmıştır. Normal array sort eden merge sort ile constant time işlem farkı içerir



3.1.3 Wort-case Performance Analysis

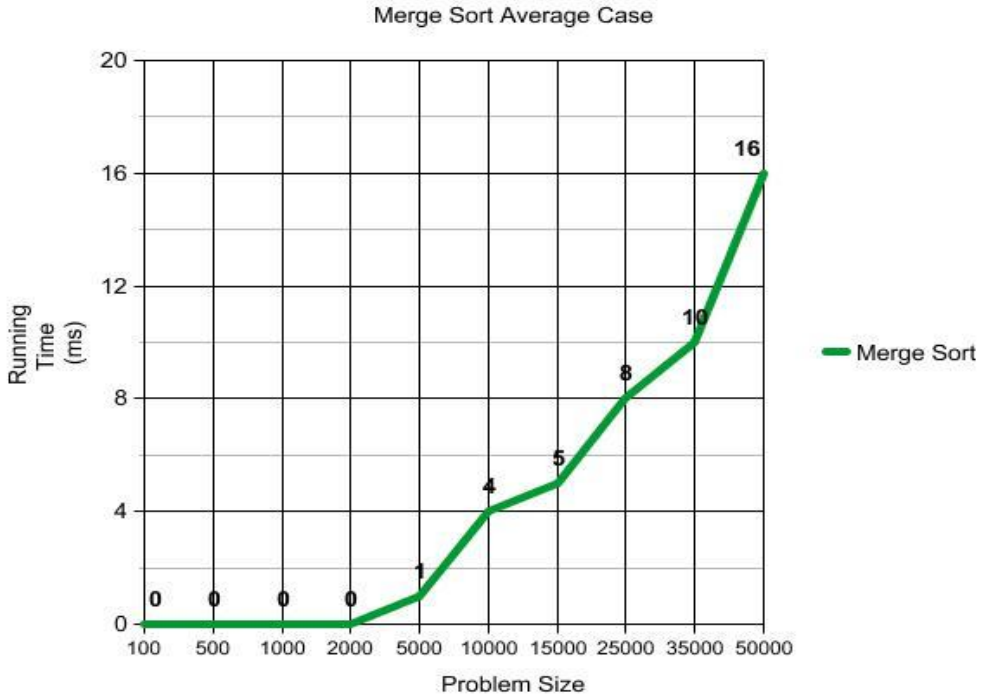
MergeSort DLL için worst case nin best ve average caseden farklı olmadığı görülmüştür çünkü her koşulda 1 elemanlık listeler haline dönüşünceye kadar yapılan ayırma işlemi T(n) = θ (logn) süre aldığı ve ayrılmış bu listelerin sıralanıp birleştirilme işlemi T(n) = θ (n) zaman aldığı görülmüştür. Sonuç olarak Worst case T(n) = θ (nlogn) olduğu anlaşılır.



3.2 MergeSort

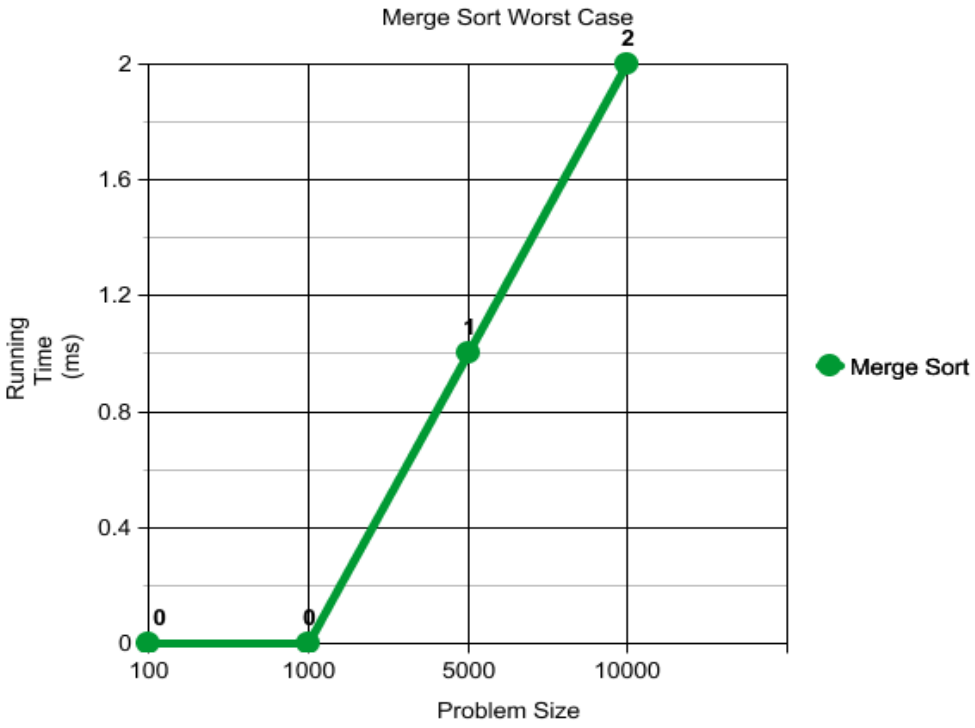
3.2.1 Average Run Time Analysis

Sıralanacak n boyutlu rastgele dizili arrayin 1 elemanlık diziler haline dönüşünceye kadar yapılan ayırma işlemi T(n) = θ (logn) süre alırken. Ayrılmış bu arraylerin sıralanıp birleştirilme işlemi T(n) = θ (n) zaman alır. Sonuç olarak Average T(n) = θ (nlogn) elde edilir bu zaman arrayin nasıl dizili olduğundan bağımsız olarak bu şekilde çıkar.



3.2.2 Wort-case Performance Analysis

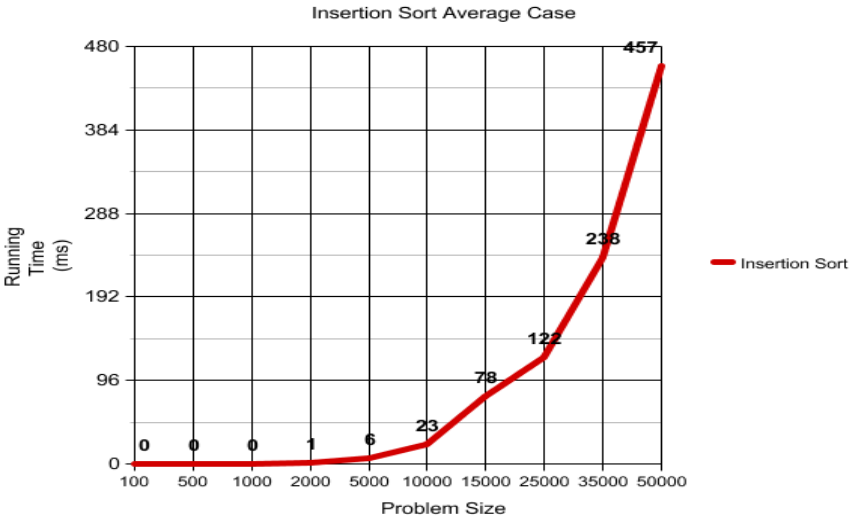
MergeSort DLL ile aynı şekilde ve aynı sebeplerden ötürü worst case nin best ve average caseden farklı olmadığı görülmüştür. Sonuç olarak Worst case T(n) = θ (nlogn) dir.



3.3 Insertion Sort

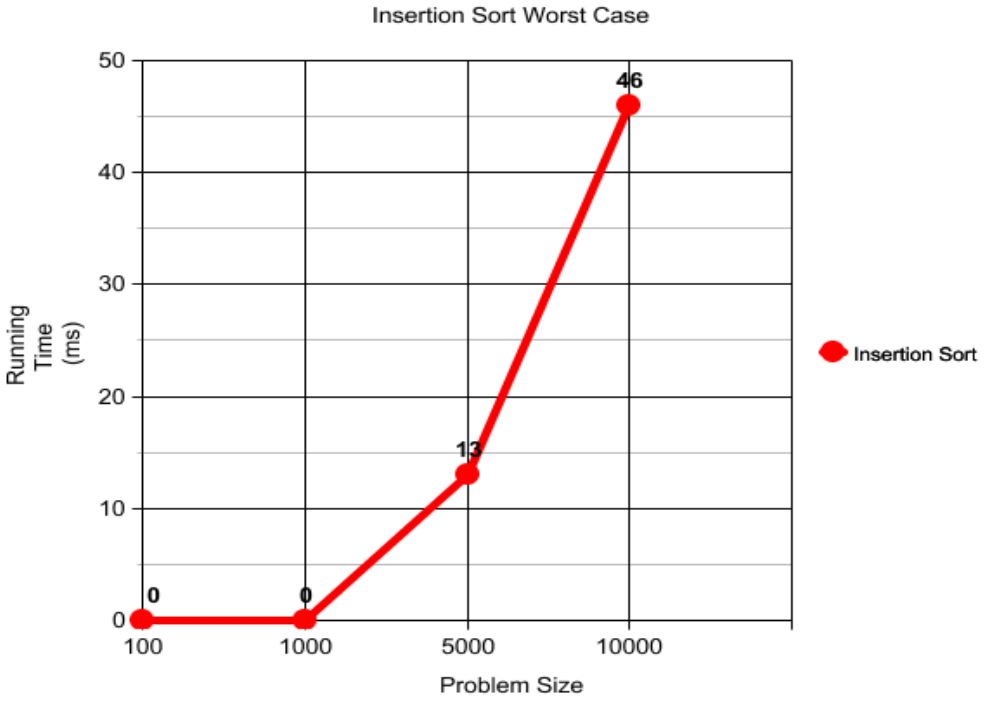
3.3.1 Average Run Time Analysis

Rastgele sırada elemanlar ile doldurulmuş herhangi bir dizinin sıralaması insertion sort ile yapıldığında T(n) = θ (n^2) olarak elde edilir. Arrayler ile yapılan denemeler sonucu Best case durumu (dizinin sıralı olması durumu θ (n)) dışındaki tüm array dizilişlerinde quadratik bir zaman değişimi elde edilmiştir.



3.3.2 Wort-case Performance Analysis

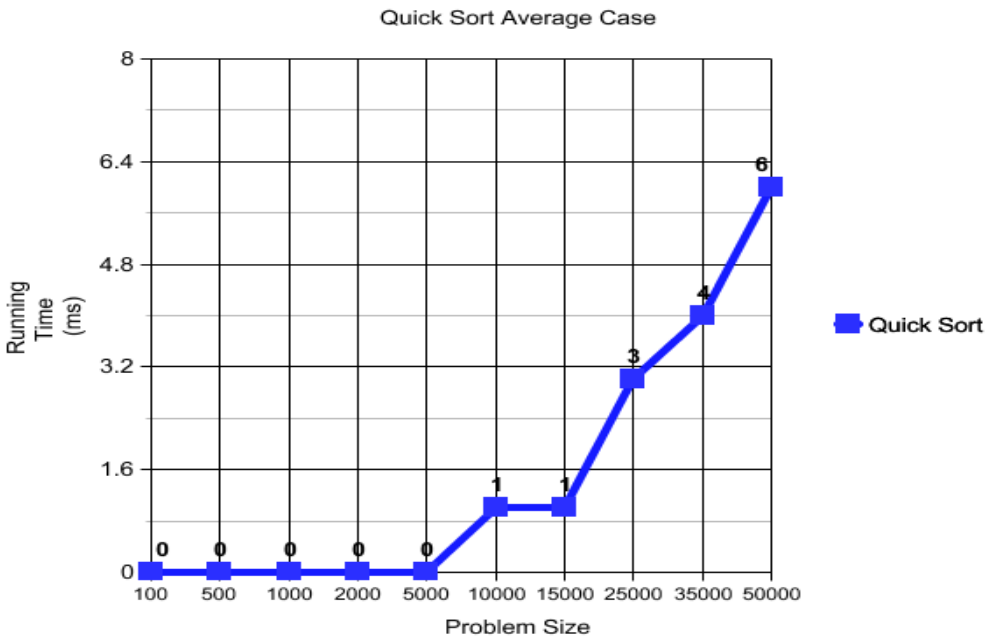
Array elemanlarının ters sıralı olarak yollanması durumunda T(n) = θ (n^2) olarak worst casenin gerçekleştiği görülmüştür. Average casesi ile aynı T(n) e sahiptir.



3.4 Quick Sort

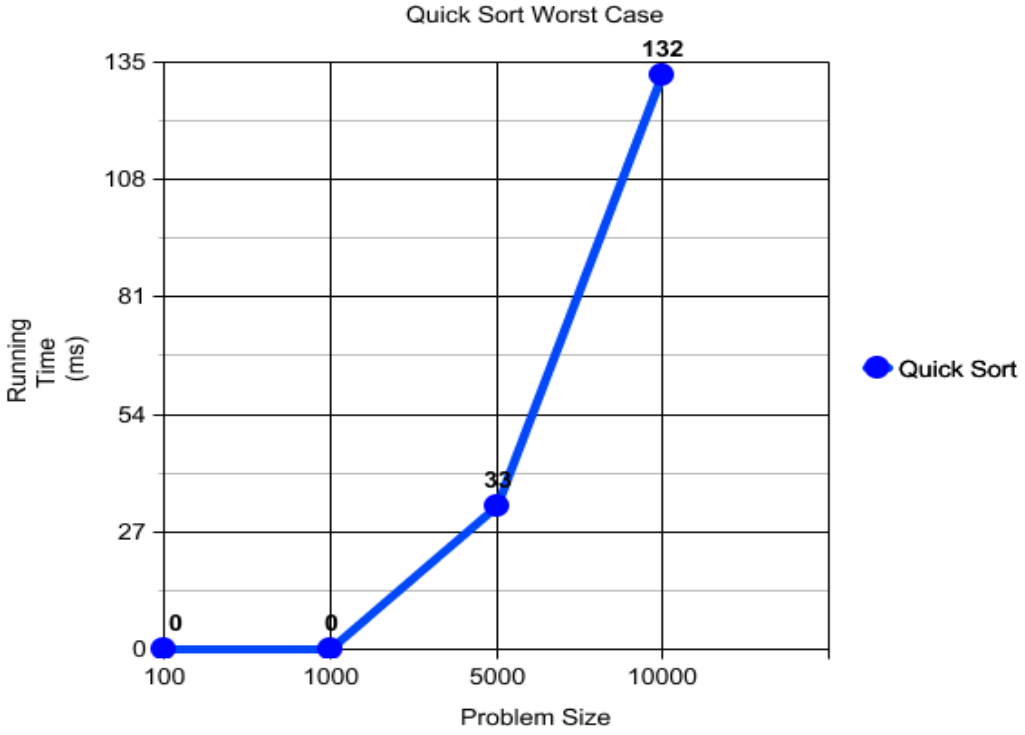
3.4.1 Average Run Time Analysis

Rastgele sırada elemanlar ile doldurulmuş herhangi bir dizinin sıralaması quick sort ile yapıldığında T(n) = θ (nlogn) olarak elde edilir. Arrayler ile yapılan denemeler sonucu Worst case durumu (dizinin son elemanının en küçük olup pivot olarak seçilmesi θ (n^2)) dışındaki tüm array dizilişlerinde θ (nlogn) time complexity elde edilmiştir.



3.4.2 Wort-case Performance Analysis

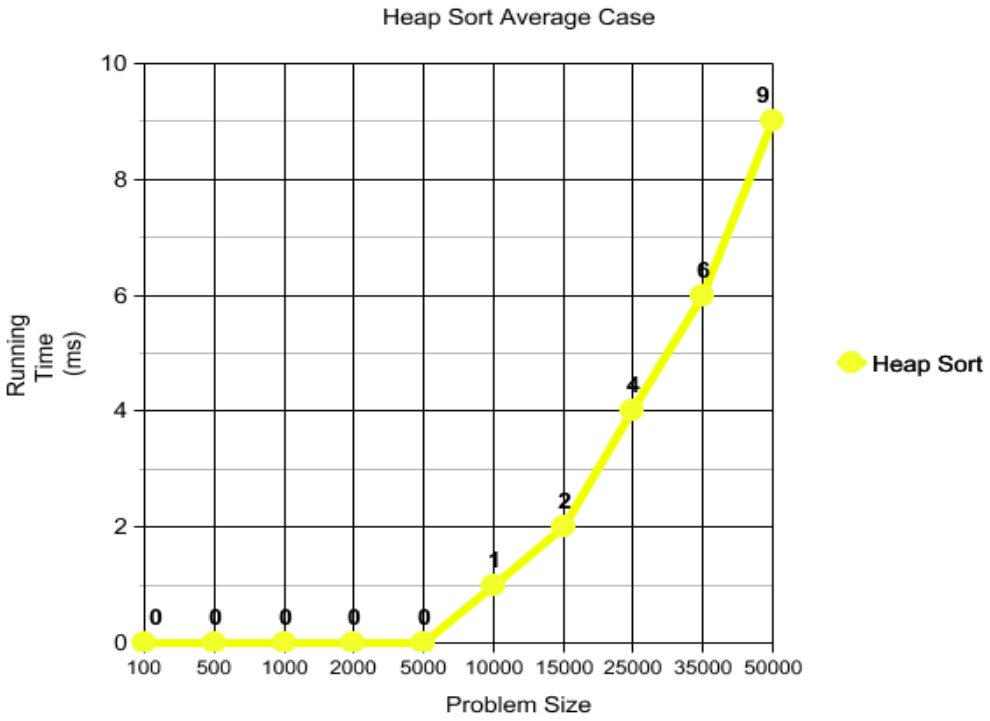
Arrayin son elemanının en küçük eleman olduğu arrayler yollanarak test edilmiştir. Worst case T(n) = θ(n^2) olduğu görülür çünkü böyle bir durumda pivotun solundaki tüm elemanların küçük olması durumu sağlanma işlemi quadratik zaman alır.



3.5 Heap Sort

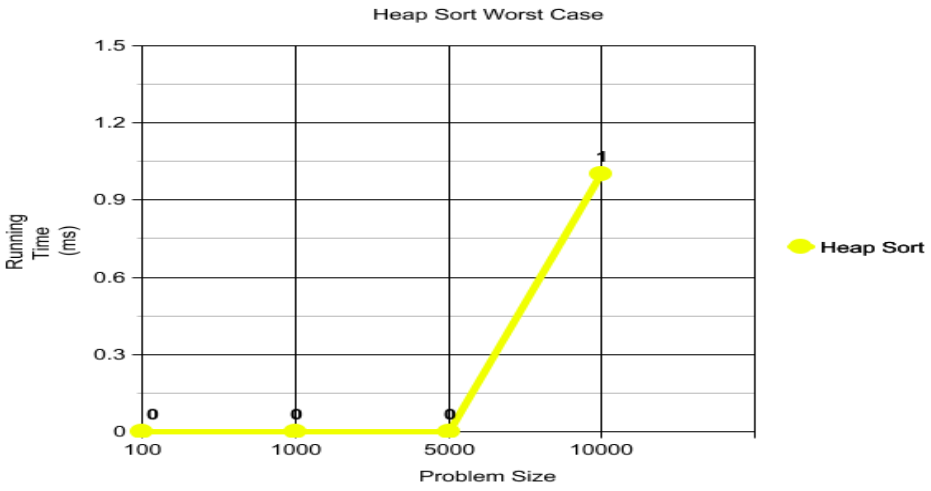
3.5.1 Average Run Time Analysis

Sıralanacak n boyutlu rastgele dizili arrayin bir heap haline dönüştürülmesi θ (n) zaman alırken oluşturulmuş heap üzerinde yapılan sort işlemleri θ (logn) zaman alır. Herhangi bir dizilişe sahip dizi için heap sort Average T(n) = θ (nlogn) zamana sahiptir.



3.5.2 Wort-case Performance Analysis

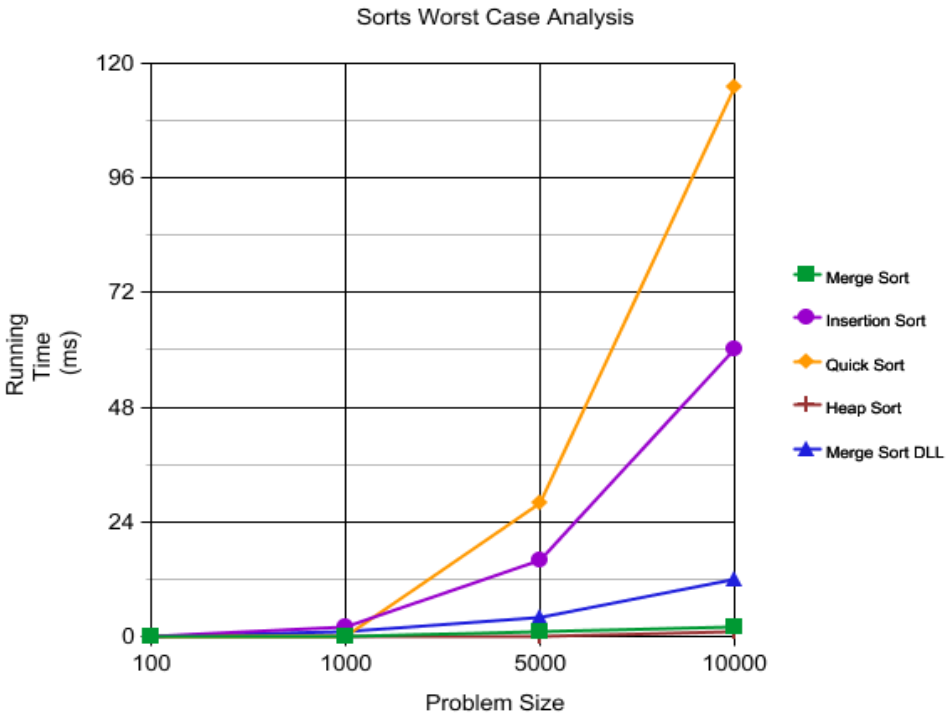
Heap sort için worst case nin best ve average caseden farklı olmadığı görülmüştür. Çünkü her koşulda bir heap oluşturma T(n) = θ(n) zaman aldığı ve oluşturulmuş heap üzerinde yapılan sort işlemleri T(n) = θ (n) zaman aldığı görülmüştür.

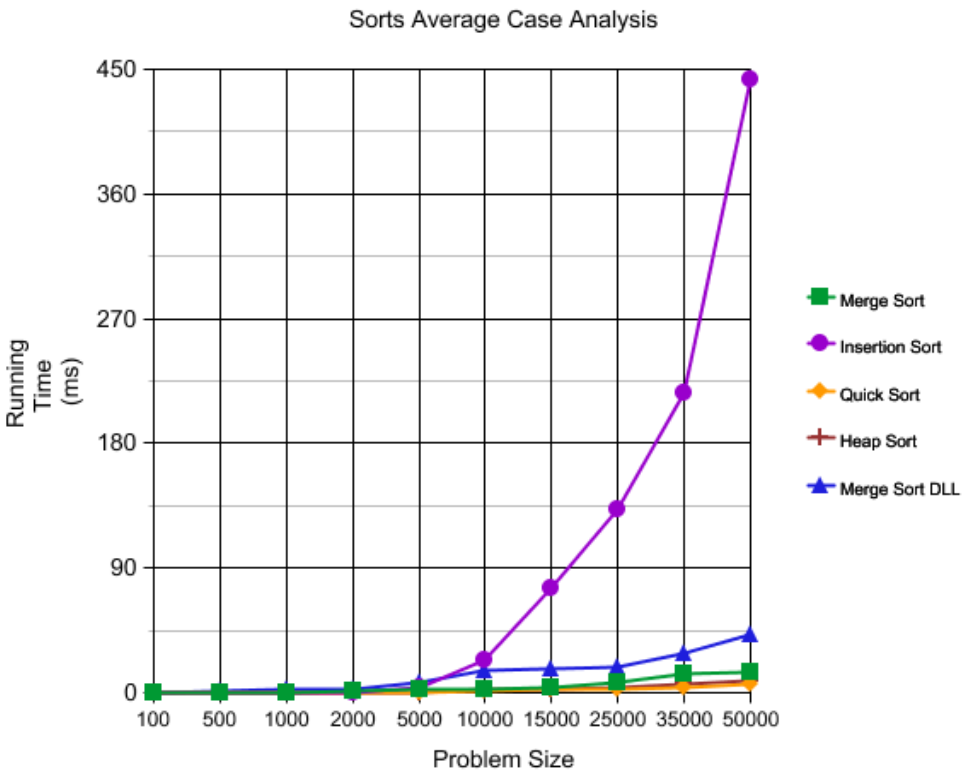


4 Comparison the Analysis Results

Merge Sort için worst case T(n) = θ(nlogn)
Insertion sort için wort case T(n) = θ(n^2)
Quick sort için worst case T(n) = θ(n^2)
Heap sort için worst case T(n) = θ(nlogn)
Merge sort için worst case T(n) = θ(nlogn)

Insertion sort ve Quick sort worst caseleri quadratik zaman alır ve nlogn zamana sahip heap,merge sort ve merge sort DLL den daha büyük time complexitye sahiptir.




Sorts Average Case Analysis
Running Time (ms)
450
360
270
180
90
0
100
500
1000
2000
5000
10000
15000
25000
35000
50000
Problem Size
Merge Sort
Insertion Sort
Quick Sort
Heap Sort
Merge Sort DLL