Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

FERHAT ŞİRİN
161044080

Course Assistant:

1 Double Hashing Map

1.1 Pseudocode and Explanation

```
 hash1
  hash =key.hashCode()%table.length;
```

Hash1 fonksiyonu key’in kendi hash kodunun ürettiği değerin tablonun uzunluğuna modunu alıp döndürür.

```
hash2
  hash =key.hashCode();
  prime =getGreatestPrime();
  hash =hash % prime;
```

Hash2 fonksiyonu key’in kendi hash kodunun ürettiği değeri tablo uzunluğundan küçük en büyük asal sayıya göre modunu alıp döndürür.

```
getGreatestPrime(){

  for(int i=table.length-1; i>1; --i){
    found =false;
    for(int j =2; j<i;++j){
      if(i%j == 0){
        found =true;
        break;
      }
    }
```

Tablo uzunluğundan küçük en büyük asal sayıyı döndürür.

```
public V put(K key, V value) {
  int count =0;
  int hash =(hash1(key)+count*hash2(key))%table.length;

  while(table[hash] !=null){

    if(table[hash].key.equals(key)){
      break;
    }
    ++count;
    hash =(hash1(key)+count*hash2(key))%table.length;

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

Ekleme yaparken önce hash1 koddan faydalanılır. Eğer bölge doluysa hash2 kodun 1. katı alınıp hash1 değeri ile toplanıp modu alınır eğer tekrar dolu çıkarsa bu sefer hash2 kodun 2. katı alınarak yer buluna kadar devam edilir. Ekleme yapıldıktan sonra load factor değerine bakılır. Önceden belirlenen 0.75 değerinden büyükse tablo boyutu arttırılır ve rehash yapılır.

```
private void rehash(){
  System.out.println("Rehashing table new size "+(2*table.length+1));
  Entry<K,V>[] oldTable =table;
  table =new Entry[2*table.length+1];
  for(int i=0;i<oldTable.length;++i){
    if(oldTable[i] != null && oldTable[i] != DELETED){
      put(oldTable[i].key,oldTable[i].value);
    }
```

rehashing de tablonun boyutu 2 katına çıkarılır ve elemanlar yeniden yeni hash değerine göre eklenir eğer daha önceden silinenler var ise onlar eklenmez.

```
public V remove(Object key) {
  if(isEmpty())
    return null;
  K keyV =(K)key;
  int count =0;
  int hash =(hash1(keyV)+count*hash2(keyV))%table.length;
  if(hash < 0)
    hash +=table.length;
  while(table[hash] !=null){
    if(table[hash] != DELETED && table[hash].key.equals(keyV)){
      V old =table[hash].value;
      table[hash] =DELETED;
      --numKeys;
      ++numDeletes;
      return old;
    }
    ++count;
    hash =(hash1(keyV)+count*hash2(keyV))%table.length;
    if(hash < 0)
      hash +=table.length;
```

Bir elemanı silmek için öce onun olması gereken yere yani hash kodundan indexi bulunur. Olası index değerlerine bakılarak elemanın olup olmadığına bakılır eğer null bir değere denk gelinirse eleman yoktur denilir. Eleman bulunduğunda DELETED özel değeri atanır.

1.2 Test Cases

Test durumları DoubleHashMapTest classının main fonksiyonuna yazılmıştır.

2 Recursive Hashing Set

2.1 Pseudocode and Explanation

```
private static class Node<E>{
  E data;
  Node<E>[] nextTable;
  public Node(E d){
    data =d;
    nextTable =null;
  }
  public void setTable(int size){
    System.out.println("New table created with size "+size);
    nextTable = new Node[size];
  }
}
```

HashingSet classında veriler Node iç classında bu şekilde tutulur. E tipinde bir veri ve eğer E değeri kullanılmışsa yeni bir tablo yapmak için bir Node<E> arrayi tutulur. İlk olarak veriler HashingSet classnın içinde yapılan theTable arrayında tutulur. Her bir array indexi başka bir tablo açma özelliğine sahiptir. Eğer tablonun kullanılması istenen indexi dolu ise o index üzerinden ulaşılan yeni bir tablo yapılabilir.

```
public boolean add(E item) {
  if(!contains(item)){
    return addRecursive(theTable,item);
  }
  return false;
}
public boolean addRecursive(Node<E>[] table,E item){
  int hash =item.hashCode()%table.length;
  if(hash < 0){
    hash +=table.length;
  }
  if(table[hash] == null){
    table[hash] =new Node<>(item);
    ++numKeys;
    System.out.println(item +" adding at index "+hash);
    double load =(double)numKeys/(double)table.length;
    if(load > LOAD_FACTOR ){
      rehash();
    }
    return true;
  }
  else{
    if(table[hash].nextTable == null){
      table[hash].setTable(getGreatestPrime(table.length));
    }
    return addRecursive(table[hash].nextTable,item);
  }
```

HashingSet de her tablonun elemanı kendi içinde başka bir tablo barındırma ihtimali olduğu için recursive bir şekilde yazıldı. Add methodu recursiveAdd methodunu çağırıp ilk tablo olarak theTablo dan başlar ve uygun bulunan yere eleman eklenmeye çalışılır. Ekleme yapmak için key'in hash kodundan dönen değerin tablo uzunluğuna göre modu alınır.Eğer eklenmek istenen yer boş ise değer eklenir fakat yer dolu ise tablonun her bir indexi kendi içinde başka bir tablo tutabildiği için yeni bir tablo oluşturulur ve ekleme buraya yapılmaya çalışılır eğer orasıda dolu ise tekrar yeni bir tablo oluşturulur. Ekleme yapıldıktan sonra load factore bakılır eğer önceden belirlenen 0.75 değerini aşılmışsa rehash işlemi yapılır.

```
private void rehash(){
  System.out.println("Rehashing table new size is
"+(2*theTable.length+1));
  HashingSet<E> oldTable = new HashingSet<>(this);
  theTable =new Node[2*theTable.length+1];
  numKeys =0;
  Iterator iter = oldTable.iterator();
  while(iter.hasNext()){
    add((E) iter.next());
  }
}
```

Rehash işleminde tablonun büyüklüğü 2 kat arttırılır ve elemanlar yeni hash değerine göre yeniden eklenir. Eğer tablonun indexlerinin alt tabloları varsa hepsi sıfırlanır ve ekleme en baştan yapılır.

```
public boolean remove(Object o) {
  return removeRecursive(theTable,o);
}
private boolean removeRecursive(Node<E>[] nodeTable,Object o){
  if(nodeTable != null){
   int hash =o.hashCode()%nodeTable.length;
   if(hash < 0){
     hash +=nodeTable.length;
   }
   if(nodeTable[hash] != null && nodeTable[hash].data.equals(o)){
     if(nodeTable[hash].nextTable != null) {
       nodeTable[hash].data = DELETED;
     }
     else{
       nodeTable[hash] =null;
     }
     --numKeys;
     return true;
   }
   else if(nodeTable[hash] != null) {
     return removeRecursive(nodeTable[hash].nextTable, o);
   }
   else{
     return false;
   }
  }
  return false;
}
```

Bir elemanı çıkarmak için önce olması gereken yere yani hash koduna bakılır ve indexi alınır eğer index null ise eleman yoktur, eğer index dolu ve eleman orda ise o bölgeye DELETED özel değeri atanır ve bir sonraki rehash işleminde tablo yeniden düzenlendiğinde bu değerde kaldırılır. Eğer index dolu fakat eleman orda yok ise o zaman o indexteki alt tabloya bakılır. Bu durum için remove methodu recursive şekilde yapılmıştır. Eleman indexte bulunmadığı takdirde alt tablolara bakılır. Alt tablolara göre hash alınıp index bakılır ve bulunduğunda DELETED değeri atanır.

```
public boolean contains(Object o) {
  if(isEmpty())
    return false;
  return findRecursive(theTable,o);
}
private boolean findRecursive(Node<E>[] nodeTable,Object o){
  while(nodeTable !=null){
    int hash =o.hashCode()%nodeTable.length;
    if(hash < 0){
      hash +=nodeTable.length;
    }
    if(nodeTable[hash] != null && nodeTable[hash].data.equals(o)){
      return true;
    }
    else if(nodeTable[hash]!= null){
      return findRecursive(nodeTable[hash].nextTable,o);
    }
    else{
      return false;
    }
  }
  return false;
}
```

Bir elemanın olup olmadığını bulmak için önce eğer o eleman var ise onun olması gereken yere yani hash kodundan indexi bulunarak o bölgeye bakılır. Eğer bölge boş ise false değeri döndürülür. Eğer bölge dolu fakat eleman ona eşit değilse alt tabloların olup olmadığına bakılır. İndexin alt tablosu var ise alt tablolara inilir ve o bölgeye göre hash kodundan index alınır ve eleman bulunmaya çalışılır. Her bir indexte alt tabloya inme ihtimali olduğu için method recursive bir şekilde yazılmıştır.

```
public Iterator<E> iterator() {
  return new Iterator<E>() {
    int index =0;
    Node<E>[] table =theTable;
    Stack<Node<E>[]> backTable =new Stack<>();
    Stack<Integer> backIndex =new Stack<>();
    @Override
    public boolean hasNext() {
      if(table !=null) {
        while (index < table.length && table[index] == null) {
          ++index;
        }
        if (index < table.length ) {
          if(table[index].data.equals(DELETED)){
            backTable.push(table);
            backIndex.push(index+1);
            table =table[index].nextTable;
            index =0;
            return hasNext();
          }
          else {
            return true;
          }
        }
        else {
          try {
            table = backTable.pop();
            index = backIndex.pop();
            return hasNext();
          }
          catch(EmptyStackException ex){
            return false;
          }
        }
      }
      return false;
    }
```

Iterator ile tablo hızlı bir şekilde gezilebilir. Iterator önce ana tablo theTable ile başlar. İlk indexten başlayıp aşağı doğru inilir. Eğer bir indexte alt tablo var ise o indexin değerinden sonra o alt tabloya inilir ve o tablo bittikten sonra ana tabloya devam edilir bunun için stack yapısı kullanılmıştır. Eğer o indexin alt tablosu yoksa index değeri arttırılarak diğer değerlere bakılır.

2.2 Test Cases

Test durumları HashingSetTest classının main fonksiyonuna yazılmıştır.

3 Sorting Algortihms

3.1 MergeSort with DoubleLinkedList

3.1.1 Pseudocode and Explanation

```
public static  <E extends Comparable<E>> void
doubleMergeSort(LinkedList<E> list) {
  // A table with one element is sorted already.
  if (list.size() > 1) {
    // Split table into halves.
    int halfSize = list.size() / 2;
    LinkedList<E> leftList = new LinkedList<>();
    LinkedList<E> rightList = new LinkedList<>();
    leftList.addAll(list.subList(0, halfSize));
    rightList.addAll(list.subList(halfSize, list.size()));
    // Sort the halves.
    doubleMergeSort(leftList);
    doubleMergeSort(rightList);
    // Merge the halves.
    merge(list, leftList, rightList);
  }
}
```

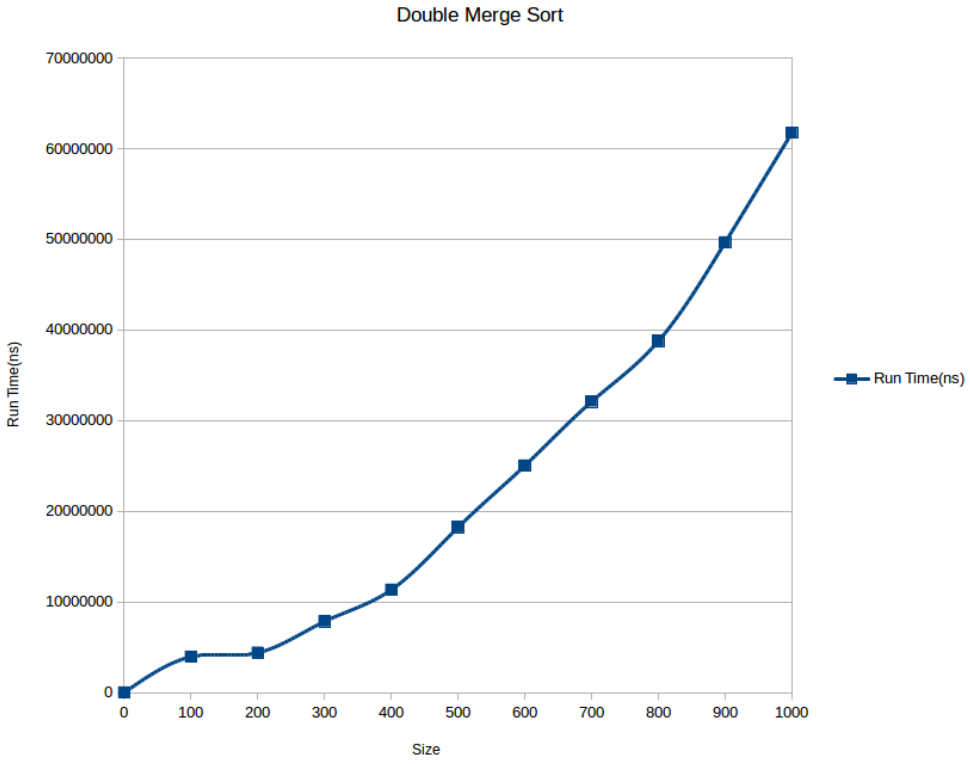
Kullanıcı doubleMergeSortu çağırdığında gönderdiiği liste içindeki değerler sıralı hale getirilir. Bunun için önce listedeki elemanların ilk yarısı sol liste ve ikinci yarısı sağ liste konulur. Bu işlem listete 1 eleman kalıncaya kadar devam eder. Bu işlem sonucunda elde edilen listeler merge fonksiyonuna gönderilir buradan list sort edilmiş halde döner ve tekrardan listeye sort edilmiş halde eklenir.

```
private static <E extends Comparable<E>> void merge(LinkedList<E>
outputList, LinkedList<E> leftList, LinkedList<E> rightList) {
  int i = 0; // Index into the left input list.
  int j = 0; // Index into the right input list.
  int k = 0; // Index into the output list.
  // While there is data in both input sequences
  while (i < leftList.size() && j < rightList.size()) {
    // Find the smaller and insert it into the output list.
    if (leftList.get(i).compareTo(rightList.get(j)) < 0) {
      outputList.set(k,leftList.get(i));
      ++i; ++k;
    } else {
      outputList.set(k,rightList.get(j));
      ++k; ++j;
    }
  }
  // assert: one of the list has more items to copy.
  // Copy remaining input from left list into the output.
  while (i < leftList.size()) {
    outputList.set(k, leftList.get(i));
    ++k; ++i;
  }
  // Copy remaining input from right list into output.
```

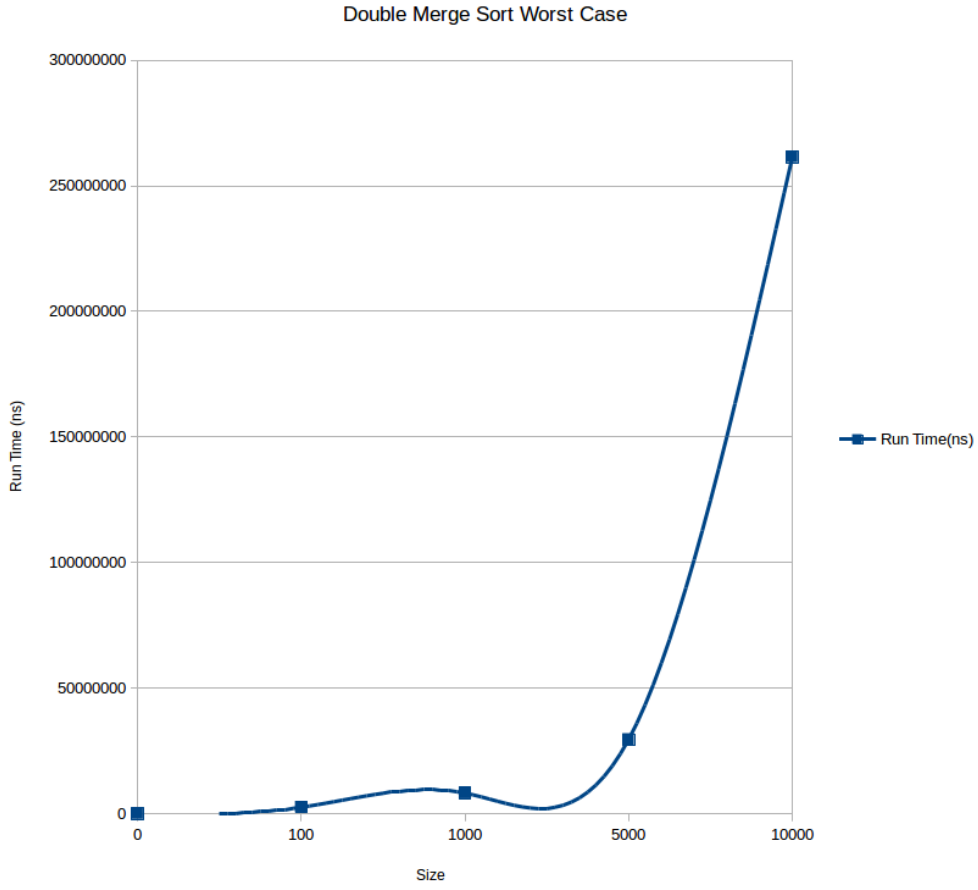
```
  while (j < rightList.size()) {
    outputList.set(k,rightList.get(j));
    ++k; ++j;
  }
}
```

Merge fonksiyonunda gelen sağ ve sol liste sıralı bir şekilde birleştirilerek ana liste ortaya çıkarılır. İlk önce sağ ve sol listenin ilk elemanlarına bakılır ve küçük olan ana listenin ilk yerine set edilir daha sonra listedeki diğer elemanlara da aynı işlem yapılır. Eğer işlem sonunda sağ veya sol list daha fazla elemana sahipse ve daha eklenmemiş elemanı kaldıysa o elemanlar listeye sırasıyla eklenir. Bu işlemler sonunda elde edilen list sıralı bir şekildedir.

3.1.2 Average Run Time Analysis

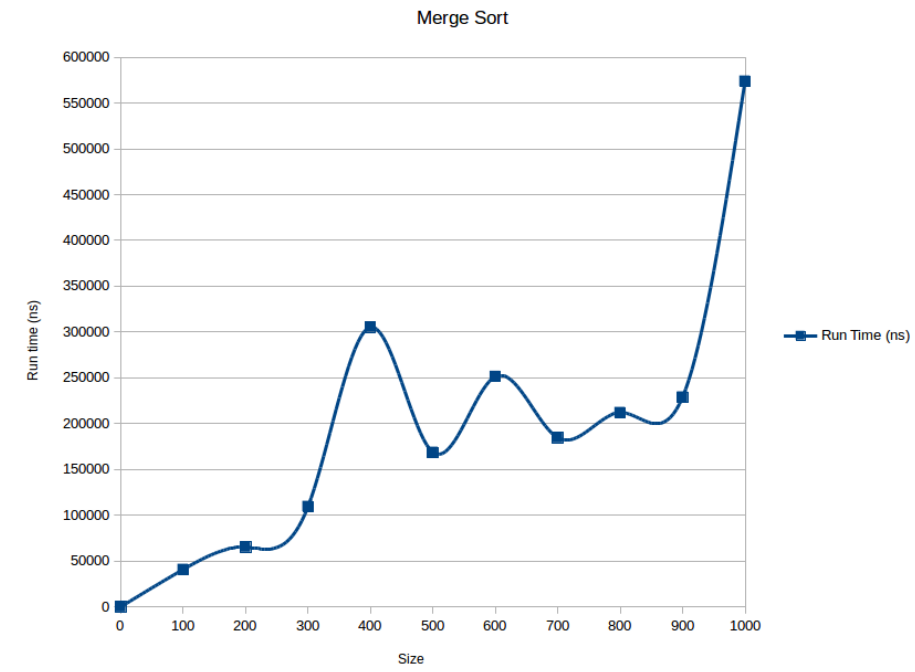


3.1.3 Wort-case Performance Analysis

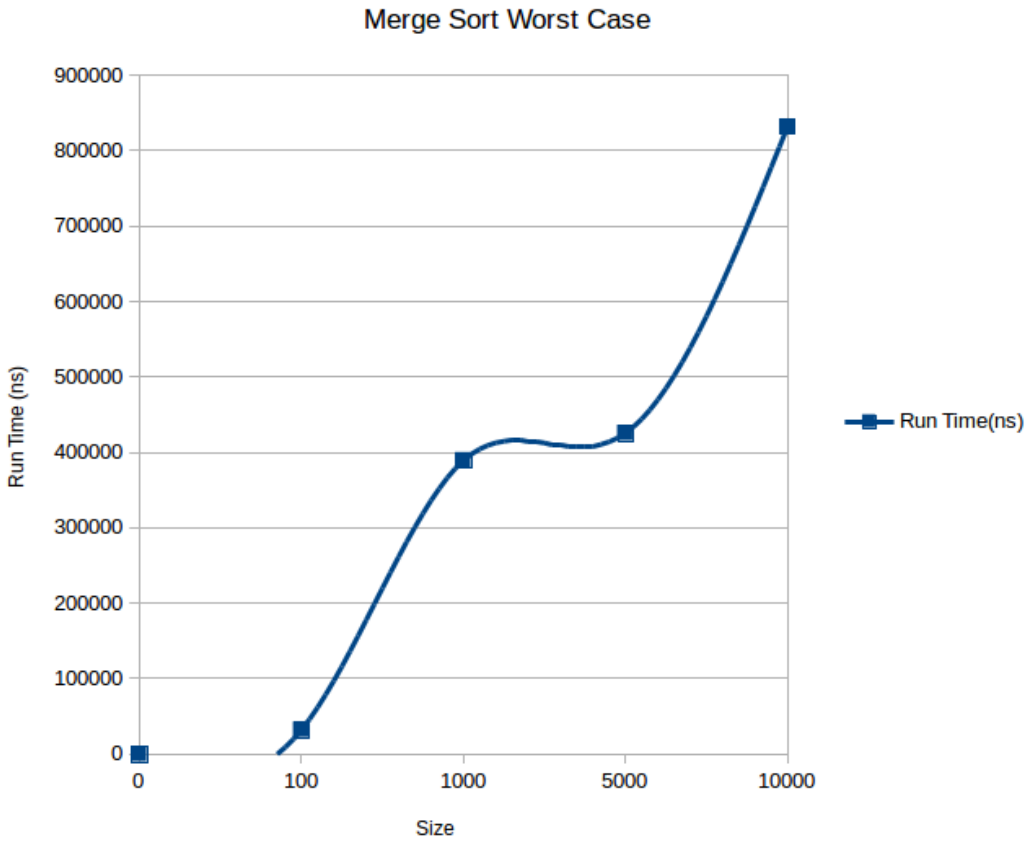


3.2 MergeSort

3.2.1 Average Run Time Analysis

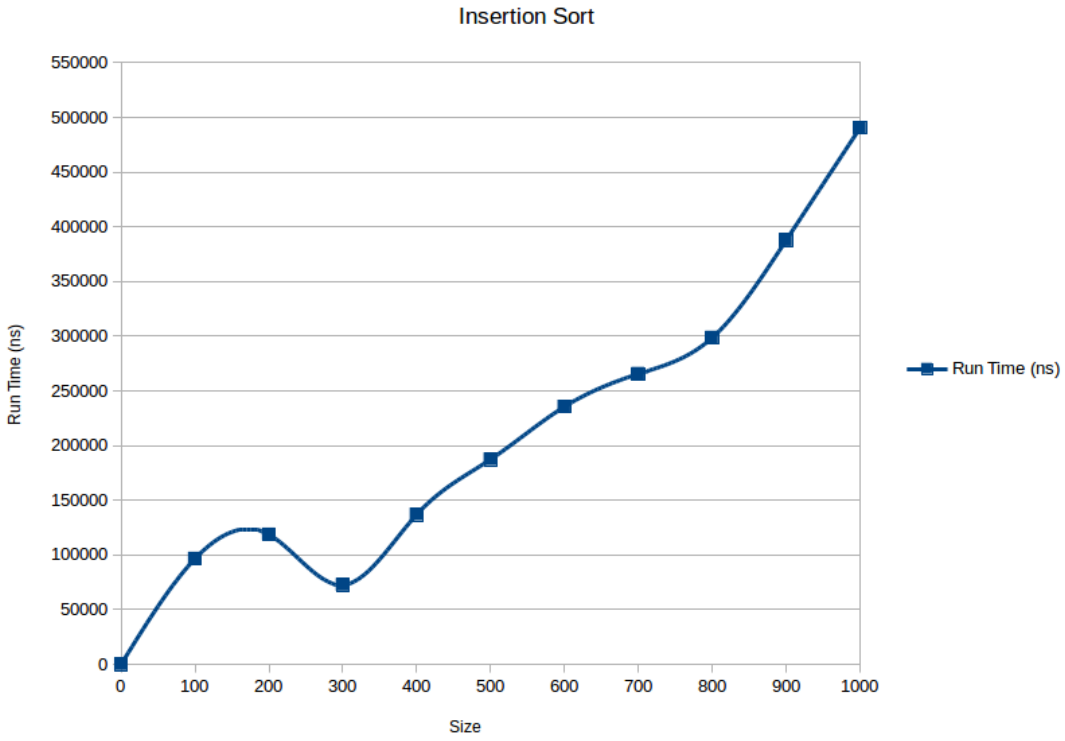


3.2.2 Wort-case Performance Analysis

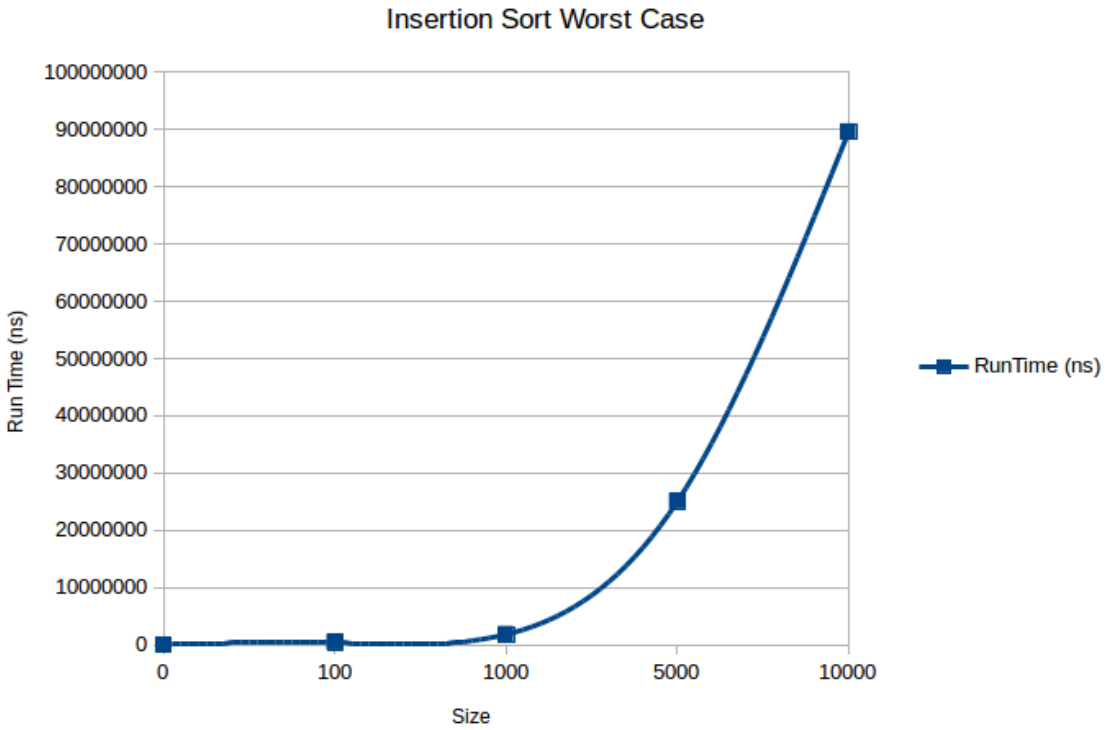


3.3 Insertion Sort

3.3.1 Average Run Time Analysis

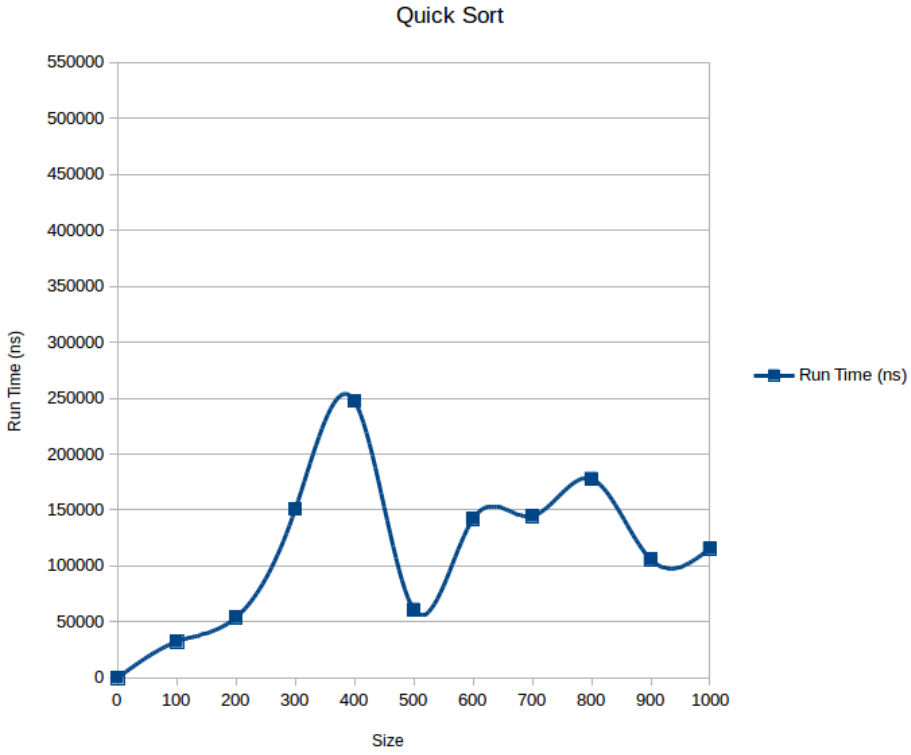


3.3.2 Wort-case Performance Analysis

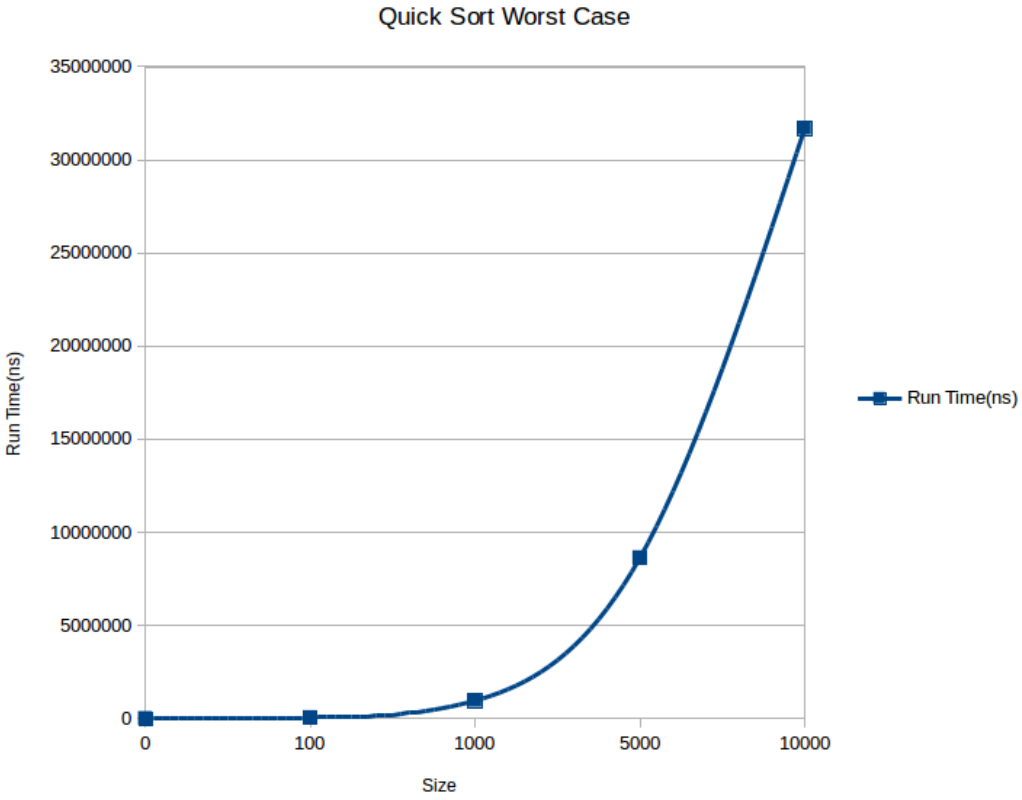


3.4 Quick Sort

3.4.1 Average Run Time Analysis

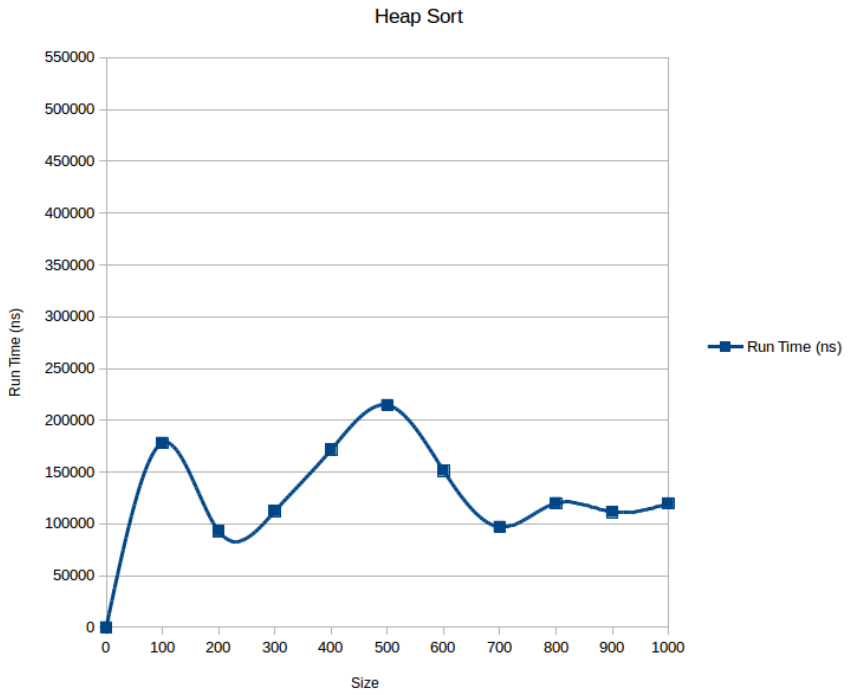


3.4.2 Wort-case Performance Analysis

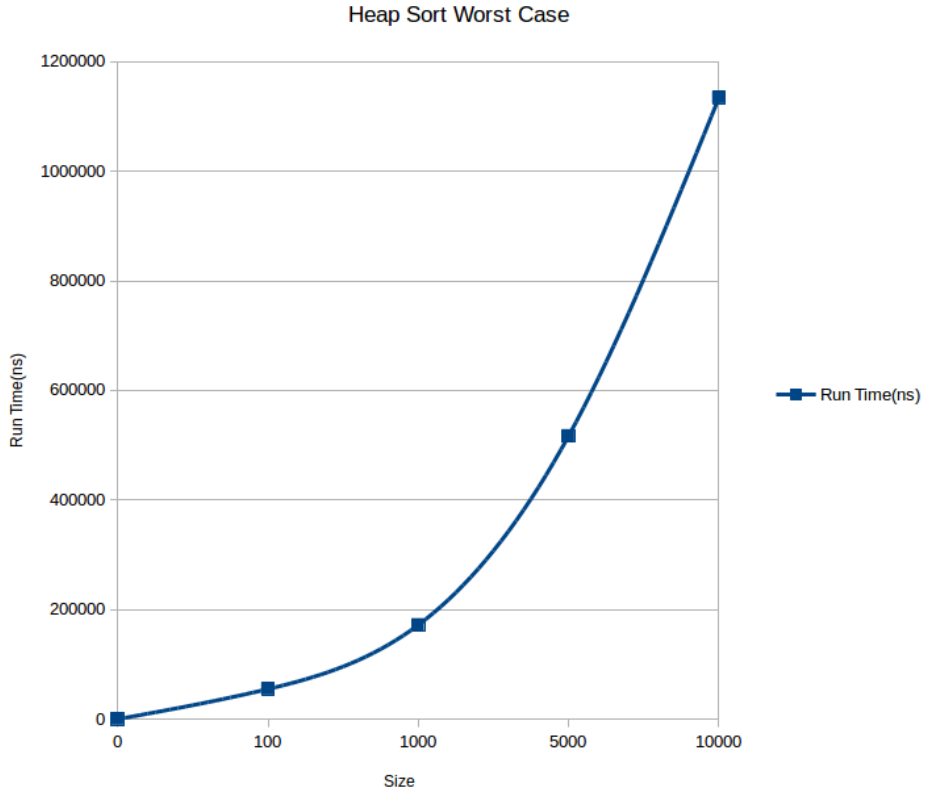


3.5 Heap Sort

3.5.1 Average Run Time Analysis



3.5.2 Wort-case Performance Analysis



4 Comparison the Analysis Results