Contents

1 Basic Test Results

```
compiling with
    javac -cp /cs/course/2013/oop/lib/junit4.jar  *.java
tests output :

ChainedHashSet
==============
Perfect!

OpenHashSet
===========
Perfect!
```

2 README

```
roigreenberg


##################
File Description
##################

SimpleSet.java - an interface of the method add, delete, contains and size.
SimpleHashSet.java  - an abstract class implementing SimpleSet
ChainedHashSet.java - a hash-set base on chaining. Extends SimpleHashSet.
ChainedHashSet.java - a hash-set base on open-adressing with quadric probing.
                      Extends SimpleHashSet.
CollectionFacadeSet.java - wrap an object implementing java's Collections
                        interface with a class that has common type with our
                        own implementaions for sets
SimpleSetPerformanceAnalyzer.java - has a main method that measures the run-times
README - this file

########
Design
########

For the design I choose to try  as much as I could to write the code in SimpleHashSet
    class for both hashSet to reduce double in the code.
    the method 'resize' is implement at SimpleHashSet. it change the capacity variable
    then call for 'refillTable' to create the updated table. each implementation for itself
    Also most of the variable is the same in both implementation so they declare there.

Another decision is to seperate the ADD method so the accual adding will be in seperate
    method(addToSEt). the reason is that when rehsing the entire table there are no need to
    check if the key is exist or to check the load factor because we are in state that we
    accualy just passed the load factor and all the keys we rehashing are already added to
    the table before and each of them already unique.

In case of the ChainedHashSets I start as recommand with ArrayList of LinkedList but then
    I find out the using ArrayList instead of the LinkedList return faster result so I
    change to this implementation.
    I also find out that if I create the table this an empty ArrayList from the begining
    it work faster then start with null's and create the array only when needed.

In case of OpenHashSet I used array of String (string[]).
    the main issue here were the deletion mechanithem which I describled below
    Another thing I did in order to reduce the runing time is to calculate the hash only
    once. I saw that if when adding and deleting I already calculte the hashcode to see
    if the key is contain I can use the calculate for adding the or deleting the keys.


All the method are using the principle of the HASH so in avarage it take only O(1)
for every action.



#######################
Implementation Issues
#######################

Once again, the big issue was the deletion mechanithem at OpenHashSet which describled below

A problem were in case of rehashing the table of OpenHashSet. As I wrote, in this case I
```

didn't use contains so it didn't calculte the hash.
to solve it I choose to calculate the hashcode in 'addToSet' method only in case
the index of the hash is 0, which in case of rehashing before adding the keys the
index is set to 0.
Also it mean that at normal add, if the calculate stay with index=0 it also use it
again but this will stop at the first time of the while loop so it wont cause a time

1. I didn't add any extra files.
2. For implementing the ChainedHashSet, I choose as reccomand to use ArrayList of
   LinkedLists (I find out using another ArrayLIst will be faster but I have been
   told that this may cause issues)
   I create an ArrayList full with empty LinkedList(another decision I made after
   seeing the diffrences in running time if I start with null and put the list when
   needed.
3. The deletion mechanism - I choose to use a technict maybe little unordinary.
   Each deleted key is replaced by the string "<del>"(can be any other key)
   so no problem will happend when looking of key.
   To overcome the problem can be if the above string will be the actual key
   need to add to the table, I decided to use spacial term for this situation.
   I create a boolean variable(named "isDelIn") that tell if the string is in the
   table or not, and every time add,delete, or contains is called, if the key is
   this, it will follow the spacial terms.
   contains - searching for the key wont work so it just return the boolean variable.
   delete - just change the boolean to false(no use the replace it with the same key)
   add - add normaly, and change the boolean to true
   When the table need to resize and rehash, first I add every other keys, the if neccery
   I add the spacial key.
   a. I test every possible situation to make sure that solution work. I also pass the auto
      test when this key is "DAST" which is one of the key used in the tests.
   b. I heard about the solution of using == and equals but I couldn't make it work.
      maybe because of the way I do the resizing.

4. Analysis result:
   the time will be in the follow format <milisecond>, <second>, <minutes-if needed>
   exept for LinkedList all the time's of the contains checks are avarage of 50,000 checks.

   a. add text1:
      i.   ChainedHashSet - 32761 mSec, 32.7 sec, 0.5 min
      ii. OpenHashSet   - 164257 mSec, 164 sec, 2.48 min
      iii. TreeSet   - 56 mSec,   0.056 sec
      iv. *LinkedList*   - 5 mSec,   0.005 sec
      v.   HashSet   - 67 mSec,   0.067 sec
   b. add text2:
      i.   ChainedHashSet - 55 mSec, 0.055 sec
      ii. *OpenHashSet* - 13 mSec, 0.013 sec
      iii. TreeSet   - 27 mSec, 0.027 sec
      iv. *LinkedList*   - 13 mSec, 0.017 sec
      v.   HashSet   - 17 mSec, 0.017 sec
   c. text1 vs text2:
      i.   ChainedHashSet - 32761 vs 55 mSec
      ii. OpenHashSet   - 164257 vs 13 mSec
      iii. TreeSet   - 56 vs 27 mSec
      iv. *LinkedList*   - 5 vs 13 mSec
      v.   HashSet   - 67 vs 17 mSec
   d. text1: contains("hi"):
      i.   *ChainedHashSet* - 0.00004 mSec
      ii. *OpenHashSet*   - 0.00004 mSec
      iii. TreeSet   - 0.00032 mSec
      iv. LinkedList   - 6 mSec
      v.   HashSet   - 0.00006 mSec
   e. text1: contains("-13170890158"):
      i.   ChainedHashSet - 0.47 mSec
      ii. OpenHashSet   - 3.32 mSec
      iii. TreeSet   - 0.00026 mSec
      iv. LinkedList   - 4 mSec
      v.   *HashSet*   - 0.0002 mSec

f. text1: "hi" vs "-13170890158":
i. ChainedHashSet - 0.00004 vs 0.47 mSec
ii. OpenHashSet - 0.00004 vs 3.32 mSec
iii. TreeSet - 0.00032 vs 0.00026 mSec
iv. LinkedList - 6 vs 4 mSec
v. HashSet - 0.00006 vs 0.0002 mSec
g. text2: contains("hi"):
i. *ChainedHashSet* - 0.00004 mSec
ii. OpenHashSet - 0.00006 mSec
iii. TreeSet - 0.00012 mSec
iv. LinkedList - 2 mSec
v. HashSet - 0.00008 mSec
e. text2: contains("23"):
i. ChainedHashSet - 0.00004 mSec
ii. *OpenHashSet* - 0.00001 mSec
iii. TreeSet - 0.0001 mSec
iv. LinkedList - 1 mSec
v. *HashSet* - 0.00006 mSec
f. text2: "hi" vs "23":
i. ChainedHashSet - 0.00004 vs 0.00004 mSec
ii. OpenHashSet - 0.00006 vs 0.00001 mSec
iii. TreeSet - 0.00012 vs 0.0001 mSec
iv. LinkedList - 2 vs 1 mSec
v. HashSet - 0.00008 vs 0.00006 mSec

note: the amazing result in (a) and (b) for LinkedList is mostly because it allow duplicates. otherwise as we see 'contain' for linkedlist is very slow...

5. the bad result in chainedHashSet, causes from several reason. because we start with small table capacity, there are many resizing and every time take lot of time. another reason, is the time require to add keys to the same spot again and again. the bigger the ArrayList get, it take more time to add it a new key because it need to look first if the key is already there.
In the OpenHashSet, as before we have the reason of the small starting table. if we will start with much bigger table capacity, the diffrences will be amazing! Also, because of the adentical hash code, every next key will take more time to calculate the proper hash code, the 100000 key will need to calculate 100000 times!
i. ChainedHashSet -
advantages: look for key is very fast, add keys with different hash is fast
cons: very slow in case of same hash keys.
ii. OpenHashSet -
advantages: the fastest in looking for keys, add keys with different hash is fast
cons: very very slow in case of same hash keys.
iii. TreeSet -
advantages: add keys is fast regardless the hashcode diffrences
cons: reletivly slow in looking for keys(compare to other set implementation)
iv. LinkedList -
advantages: add keys *very* fast.
cons: very very slow in looking for keys
v. HashSet -
advantages: add keys is fast regardless the hashcode diffrences. looking for keys is fast

If my need is create a full data structure that allow diplicates, the LinkedList will work best but in case we also need to look for keys fast or no duplicates I will prefer the HashSet.

Between my implementaion, as reflected from the result the ChainedHashSet dealling better with collied keys but the OpenHashSet it way better when the hashcode is different In case of looking for keys both gave the best result even compare to java data structures

In case of adding the same hashcode keys, I didn't had a chance against java hashSet but in different hashcode, my OpenHashSet bit the java structure. Also in case of looking for keys mine worked fastar.

I was very surprise to see the huge diffrance between the time took to add text1 to my implementation compare to java's.
I wasn't surprise that most of the result of "Contains" take no time and cause me to

do it 50000(!) time to get comparable results. the reason is because it use hash,
if check a spesific location(s) so even full table can be search fast.

As I said the result of java structures ws very surprising. espacialy hashSet.
explanation for this, exept of better implementation as expected from better programers
is that every hash function return diffrent value and it very possible that the function
they use does not give the same value to all the keys in text1 which make the copmare
"unfair".

Using the advises from appendix A indeed make the run-time fasters.
Since I also change many other things all the time I can't tell how much this thing helped.

3 ChainedHashSet.java

```
1  import java.util.*;
2
3
4  /**
5   *
6   * @author RoiGreenberg
7   */
8  public class ChainedHashSet extends SimpleHashSet{
9      ArrayList<ArrayList<String>> tempTable;
10     ArrayList<ArrayList<String>> table = new ArrayList<ArrayList<String>>();
11
12
13
14     /**
15      * A default constructor.
16      * Constructs a new, empty table with default initial capacity (16),
17      * upper load factor (0.75) and lower load factor (0.25).
18      */
19     public ChainedHashSet(){
20         super (DEFAULT_UPPER,DEFAULT_LOWER);
21         createEmptyTable();
22     }
23     /**
24      * Constructs a new, empty table with the specified load factors,
25      * and the default initial capacity (16).
26      * @param upperLoadFactor The upper load factor of the hash table.
27      * @param lowerLoadFactor The lower load factor of the hash table.
28      */
29     public ChainedHashSet(float upperLoadFactor,
30               float lowerLoadFactor){
31         super (upperLoadFactor,lowerLoadFactor);
32
33         createEmptyTable();
34     }
35     /**
36      * Data constructor - builds the hash set by adding the elements one by one.
37      *  Duplicate values should be ignored. The new table has the default values
38      *   of initial capacity (16), upper load factor (0.75),
39      *   and lower load factor (0.25).
40      * @param data Values to add to the set.
41      */
42     public ChainedHashSet(java.lang.String[] data){
43         super (DEFAULT_UPPER,DEFAULT_LOWER);
44         createEmptyTable();
45         for (String singleData: data)
46             this.add(singleData);
47     }
48     /**
49      * Add a specified element to the set.
50      * check if the value not in the table, add it and resize the table if needed
51      * @param newValue - New value to add to the set
52      * @return False iff newValue already exists in the set
53      */
54     @Override
55     public boolean add(String newValue) {
56
57         if (!this.contains(newValue)){
58
59             size++;
```

```
60              addToSet(newValue);
61              if ((float)this.size()/currentCapacity>upperLoadFactor){
62                  resize("increase");
63              }
64              return true;
65          }
66          return false;
67      }
68
69      /**
70       * Add a specified element to the set.
71       * find the right hashcode and insert the value to the table
72       * @param newValue - New value to add to the set
73       */
74      protected void addToSet(String newValue) {
75          hashIndex = hash(newValue);
76          table.get(hashIndex).add(newValue);
77      }
78
79
80      /**
81       * the function copy the table then recreate the table with current
82       * capacity then readd the values from the copied table.
83       */
84      protected void refillTable() {
85          tempTable = table;
86          createEmptyTable();
87          for (ArrayList<String> arr: tempTable){
88              for (String value: arr){
89                  this.addToSet(value);
90              }
91          }
92      }
93
94
95
96      /**
97       * Look for a specified value in the set.
98       * @param searchVal - Value to search for
99       * @return True iff searchVal is found in the set
100      */
101     @Override
102     public boolean contains(String searchVal) {
103         hashIndex = hash(searchVal);
104         return (table.get(hashIndex).contains(searchVal));
105
106     }
107     /**
108      * Remove the input element from the set.
109      * @param toDelete - Value to delete
110      * @return True iff toDelete is found and deleted
111      */
112     @Override
113     public boolean delete(String toDelete) {
114
115         if (this.contains(toDelete)){
116             hashIndex = hash(toDelete);
117             table.get(hashIndex).remove(toDelete);
118             size--;
119
120             if ((double)this.size()/currentCapacity<lowerLoadFactor){
121
122                 resize("decrease");
123             }
124             return true;
125         }
126         return false;
127     }
```

```
128         /**
129          * @return The number of elements currently in the set
130          */
131         @Override
132         public int size() {
133 
134             return this.size;
135         }
136         /**
137          * @return The current capacity (number of cells) of the table.
138          */
139         public int capacity(){
140             return this.currentCapacity;
141         }
142 
143         /**
144          * reset the table according to new capacity
145          */
146         protected void createEmptyTable() {
147             table = new ArrayList<ArrayList<String>>();
148             for (int i=0;i<currentCapacity;i++)
149                 table.add(i, new ArrayList<String>());
150         }
151 
152         /**
153          * return the hash-key for the value
154          * @param value - the value needed to be hashed
155          * @return the hash key
156          */
157         private int hash(String value){
158             int capacity = currentCapacity - 1;
159             return Math.abs((value.hashCode())&(capacity));
160         }
161 
162     }
```

4 CollectionFacadeSet.java

```java
/**
 * This class is used to wrap java collection so they could use together with
 * SimpleSet classes.
 * @author RoiGreenberg
 */
public class CollectionFacadeSet extends java.lang.Object implements SimpleSet {
    java.util.Collection<java.lang.String> collection;
    /**
     *
     * @param javaCollection - collection data structure
     */
    public CollectionFacadeSet(java.util.Collection<java.lang.String> javaCollection){
        this.collection = javaCollection;
    }

    public boolean add(java.lang.String newValue){
        return collection.add(newValue);
    }

    public boolean contains(java.lang.String searchVal){
        return collection.contains(searchVal);
    }

    public boolean delete(java.lang.String toDelete){
        return collection.remove(toDelete);
    }

    public int size(){
        return collection.size();
    }

}
```

-5/-5 Bad javadoc. (code='bad_javadoc')

-2/-2 Your facade does not verify that a newly added value is not already in the collection. (code='facade_no_contains_check')

5 OpenHashSet.java

```java
/**
 *
 * @author RoiGreenberg
 */
public class OpenHashSet extends SimpleHashSet{

    private int hashIndex;
    private String[] table, tempTable;
    private static final String DELETED = "<del>";
    private int index;

    /**
     * A default constructor.
     * Constructs a new, empty table with default initial capacity (16),
     * upper load factor (0.75) and lower load factor (0.25).
     */
    public OpenHashSet(){
        super (DEFAULT_UPPER,DEFAULT_LOWER);


        table = new String[currentCapacity];

    }
    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     * @param upperLoadFactor The upper load factor of the hash table.
     * @param lowerLoadFactor The lower load factor of the hash table.
     */
    public OpenHashSet(float upperLoadFactor,
              float lowerLoadFactor){
        super (upperLoadFactor,lowerLoadFactor);

        table = new String[currentCapacity];


    }
    /**
     * Data constructor - builds the hash set by adding the elements one by one.
     *  Duplicate values should be ignored. The new table has the default values
     *   of initial capacity (16), upper load factor (0.75),
     *   and lower load factor (0.25).
     * @param data Values to add to the set.
     */
    public OpenHashSet(java.lang.String[] data){
        super (DEFAULT_UPPER,DEFAULT_LOWER);
        table = new String[currentCapacity];
        for (String singleData: data)
            this.add(singleData);
    }
    /**
     * Add a specified element to the set.
     * @param newValue - New value to add to the set
     * @return False iff newValue already exists in the set
     */
    @Override
    public boolean add(String newValue) {

        if (!this.contains(newValue)){
```

```
60              size++;
61              addToSet(newValue);
62              if ((float)this.size()/currentCapacity>upperLoadFactor){
63                  resize("increase");
64              }
65              // in case the added key is same as DELETED string
66              if (newValue.equals(DELETED))
67                  isDelIn =true;
68              return true;
69          }
70          return false;
71      }
72
73
74      protected void addToSet(String newValue) {
75          if (index==0){
76              hashIndex = hash(newValue, index);
77              while (table[hashIndex]!=null  && (newValue != DELETED)) {
78                  index++;
79                  hashIndex = hash(newValue, index);
80              }
81          }
82          table[hashIndex] = newValue;
83      }
84
85
86      /**
87       * Look for a specified value in the set.
88       * @param searchVal - Value to search for
89       * @return True iff searchVal is found in the set
90       */
91      @Override
92      public boolean contains(String searchVal) {
93          // in case the key is same as DELETED string
94          if (searchVal.equals(DELETED)){
95              return isDelIn;
96          }
97
98
99          index = 0;
100         hashIndex = hash(searchVal, index);
101
102             while (table[hashIndex]!=null) {
103
104                 if (table[hashIndex].equals(searchVal)) {
105                     return true;
106                 } else {
107                     index++;
108                     hashIndex = hash(searchVal ,index);
109                 }
110             }
111
112         return false;
113
114     }
115
116     /**
117      * Remove the input element from the set.
118      * @param toDelete - Value to delete
119      * @return True iff toDelete is found and deleted
120      */
121     @Override
122     public boolean delete(String toDelete) {
123         if (this.contains(toDelete)){
124             size--;
125             table[hashIndex] = DELETED;
126             if ((double)this.size()/currentCapacity<lowerLoadFactor){
127                 resize("decrease");
```

```
            }
            // in case the deleted key is same as DELETED string
            if (toDelete.equals(DELETED))
                isDelIn = false;
            return true;
        }
        return false;
    }
    /**
     * @return The number of elements currently in the set
     */
    @Override
    public int size() {
        return this.size;
    }
    /**
     * @return The current capacity (number of cells) of the table.
     */
    public int capacity(){
        return this.currentCapacity;
    }


    /**
     * the function copy the table then recreate the table with current
     * capacity then add the values from the copied table to the table.
     */
    protected void refillTable() {
        tempTable = table;

        table = new String[currentCapacity];
        for (String value: tempTable){
            if ((value != null) && (value != DELETED)){
                index = 0;
                this.addToSet(value);
            }
        }
        // add the DELETED string if the same key is at the table
        if (isDelIn){
            index = 0;
            this.addToSet(DELETED);
        }

    }



    /**
     * calculate the hash value
     * @param value - the value need to hash
     * @param i the counter for the probing
     * @return the hash value
     */
    private int hash(String value,int i){
        int capacity = currentCapacity - 1;
        if (i==0)
            return Math.abs((value.hashCode())&capacity);
        return Math.abs((hashIndex+(i+i*i)/2)&capacity);
    }


}
```

6 SimpleHashSet.java

```java
/**
 *
 * @author RoiGreenberg
 */
public abstract class SimpleHashSet extends java.lang.Object implements SimpleSet{
    protected float lowerLoadFactor;
    protected float upperLoadFactor;
    protected int currentCapacity;
    protected int hashIndex;
    protected int size = 0;
    protected static final int INITIAL_CAPACITY = 16;
    protected static final float DEFAULT_LOWER =  (float) 0.25;
    protected static final float DEFAULT_UPPER =  (float) 0.75;
    protected int oldCapacity;
    protected boolean isDelIn = false;
    /**
     * A default constructor.
     * Constructs a new, empty table with default initial capacity (16),
     * upper load factor (0.75) and lower load factor (0.25).
     */
    public SimpleHashSet(){
        currentCapacity = INITIAL_CAPACITY;
        upperLoadFactor = DEFAULT_UPPER;
        lowerLoadFactor = DEFAULT_LOWER;
    }
    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     * @param upperLoadFactor The upper load factor of the hash table.
     * @param lowerLoadFactor The lower load factor of the hash table.
     */
    public SimpleHashSet(float upper, float lower){
        currentCapacity = INITIAL_CAPACITY;
        upperLoadFactor = upper;
        lowerLoadFactor = lower;
    }
    /**
     * @return The current capacity (number of cells) of the table.
     */
    public abstract int capacity();


    /**
     * Resizing the table capacity if needed
     * @param change - determine if to increase of decrease table size
     */
    protected void resize(String change){

        oldCapacity = currentCapacity;
        switch (change){
            case "increase":
                currentCapacity <<=1;

                break;
            case "decrease":

                currentCapacity >>=1;
                break;

```

```
60          }
61
62          refillTable();
63
64      }
65      /**
66       * the function copy the table then recreate the table with current
67       * capacity then add the values from the copied table to the table.
68       */
69      protected void refillTable(){}
70  }
```

7 SimpleSetPerformanceAnalyzer.java

```java
import java.util.*;



/**
 * THis class is used to analayzed time-run of several data structure
 * @author RoiGreenberg
 */
public class SimpleSetPerformanceAnalyzer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int N = 5;
        HashSet<String> h01 = new HashSet<String>();
        TreeSet<String> t01 = new TreeSet<String>();
        LinkedList<String> l02 = new LinkedList<String>();
        HashSet<String> h02 = new HashSet<String>();
        TreeSet<String> t02 = new TreeSet<String>();
        LinkedList<String> l01 = new LinkedList<String>();
        ChainedHashSet c1 = new ChainedHashSet();
        OpenHashSet o1 = new OpenHashSet();
        CollectionFacadeSet h1 = new CollectionFacadeSet(h01);
        CollectionFacadeSet t1 = new CollectionFacadeSet(t01);
        CollectionFacadeSet l1 = new CollectionFacadeSet(l01);

        ChainedHashSet c2 = new ChainedHashSet();
        OpenHashSet o2 = new OpenHashSet();
        CollectionFacadeSet h2 = new CollectionFacadeSet(h02);
        CollectionFacadeSet t2 = new CollectionFacadeSet(t02);
        CollectionFacadeSet l2 = new CollectionFacadeSet(l02);
        SimpleSet[] testSets1;
        testSets1 = new SimpleSet[] {l1, h1, t1, c1, o1};

        SimpleSet[] testSets2;
        testSets2 = new SimpleSet[] {l2, h2, t2, c2, o2};
        String[] setsName;
        double[] add1 = new double[5];
        double[] add2 = new double[5];
        double[] containf1 = new double[5];
        double[] containt1 = new double[5];
        double[] containf2 = new double[5];
        double[] containt2 = new double[5];
        setsName = new String[] {"LinkedList", "treeSet", "hashSet", "chainedHashSet", "openHashSet"};
        String TEST1 = "hi";
        String TEST2 = "-13170890158";
        String TEST3 = "23";

        String[] text1 = Ex4Utils.file2array("E:\\Documents\\oop\\Ex4\\src\\data1.txt");
        String[] text2 = Ex4Utils.file2array("E:\\Documents\\oop\\Ex4\\src\\data2.txt");


        long timeBefore;
        long timeAfter;
        for (int i = 0; i < N; i++){
            timeBefore = new Date().getTime();
            for (String w: text1)
                testSets1[i].add(w);
```

```
60              timeAfter = new Date().getTime();
61              add1[i] = (double) (timeAfter-timeBefore);
62
63              timeBefore = new Date().getTime();
64              for (String w: text2)
65                  testSets2[i].add(w);
66              timeAfter = new Date().getTime();
67              add2[i] = (double)(timeAfter-timeBefore);
68              int M1 = 50000;
69              int M2 = 50000;
70
71              timeBefore = new Date().getTime();
72              if (i==0)testSets1[i].contains(TEST1);
73              else
74                  for (int j = 0; j < M1; j++)
75                      testSets1[i].contains(TEST1);
76              timeAfter = new Date().getTime();
77              containf1[i] =(double) (timeAfter-timeBefore);
78
79
80              timeBefore = new Date().getTime();
81              if (i==0)testSets1[i].contains(TEST2);
82              else
83                  for (int j = 0; j < M1; j++)
84                      testSets1[i].contains(TEST2);
85              timeAfter = new Date().getTime();
86              containt1[i] = (double)(timeAfter-timeBefore);
87
88
89              timeBefore = new Date().getTime();
90              if (i==0)testSets1[i].contains(TEST3);
91              else
92                  for (int j = 0; j < M2; j++)
93                      testSets2[i].contains(TEST3);
94              timeAfter = new Date().getTime();
95              containt2[i] =(double) (timeAfter-timeBefore);
96
97              timeBefore = new Date().getTime();
98              if (i==0)testSets1[i].contains(TEST1);
99              else
100                 for (int j = 0; j < M2; j++)
101                     testSets2[i].contains(TEST1);
102             timeAfter = new Date().getTime();
103             containf2[i] = (double)(timeAfter-timeBefore);
104
105
106
107         }
108
109         for (int i = 0; i < N; i++){
110             System.out.print("Name: "+setsName[i]+"  ");
111             System.out.print("add1: "+ add1[i]+"  ");
112             System.out.print("containf1 "+ containf1[i]+"  ");
113             System.out.print("containt1 "+ containt1[i]+"  ");
114             System.out.print("size: "+ testSets1[i].size()+"  ");
115             System.out.print("add2: "+ add2[i]+"  ");
116             System.out.print("containf2 "+ containf2[i]+"  ");
117             System.out.print("containt2 "+ containt2[i]+"  ");
118             System.out.print("size: "+ testSets2[i].size()+"  ");
119             System.out.println("");
120         }
121     }
122 }
```