Algorithms & Datastructures
Laboratory Exercise Sheet 8

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

May 11th, to be discussed on May 18th

Exercise 1) B-trees The B-Tree is one of the major data structures in computer science, especially for databases. It is not optimized on comparison or swap complexity, but on block-read complexity as it is very costly to search for a block (up to multiple magnitudes against all other concerning operations), aswhile the subsequential read of a big size (a block) does not take much time.

k-B-Trees are similar to the brother trees in the lecture and consist of "lines" of keys/references as the nodes. The nodes contain $2 * k$ sorted keys and $2 * k + 1$ references to children. All leaves are at the same level. Though in general it can contain the same element twice, we are concerning only dictionaries with unique elements.

The size of k is chosen in that way, that a node fills (at best) exactly one (or a multiple of) blocks on the hard-disk. (It is one parameter to optimize databases.) Some of the interesting aspects are, the height is of small magnitude and there will be few (costly) re-balancing.

To summarize a k-B-Tree is defined as follows:

- The root may consist of 0 (sometimes 1) up to $2 * k$ keys
- All other nodes consist of at least k up to $2 * k$ keys
- All leaves are at the same level.

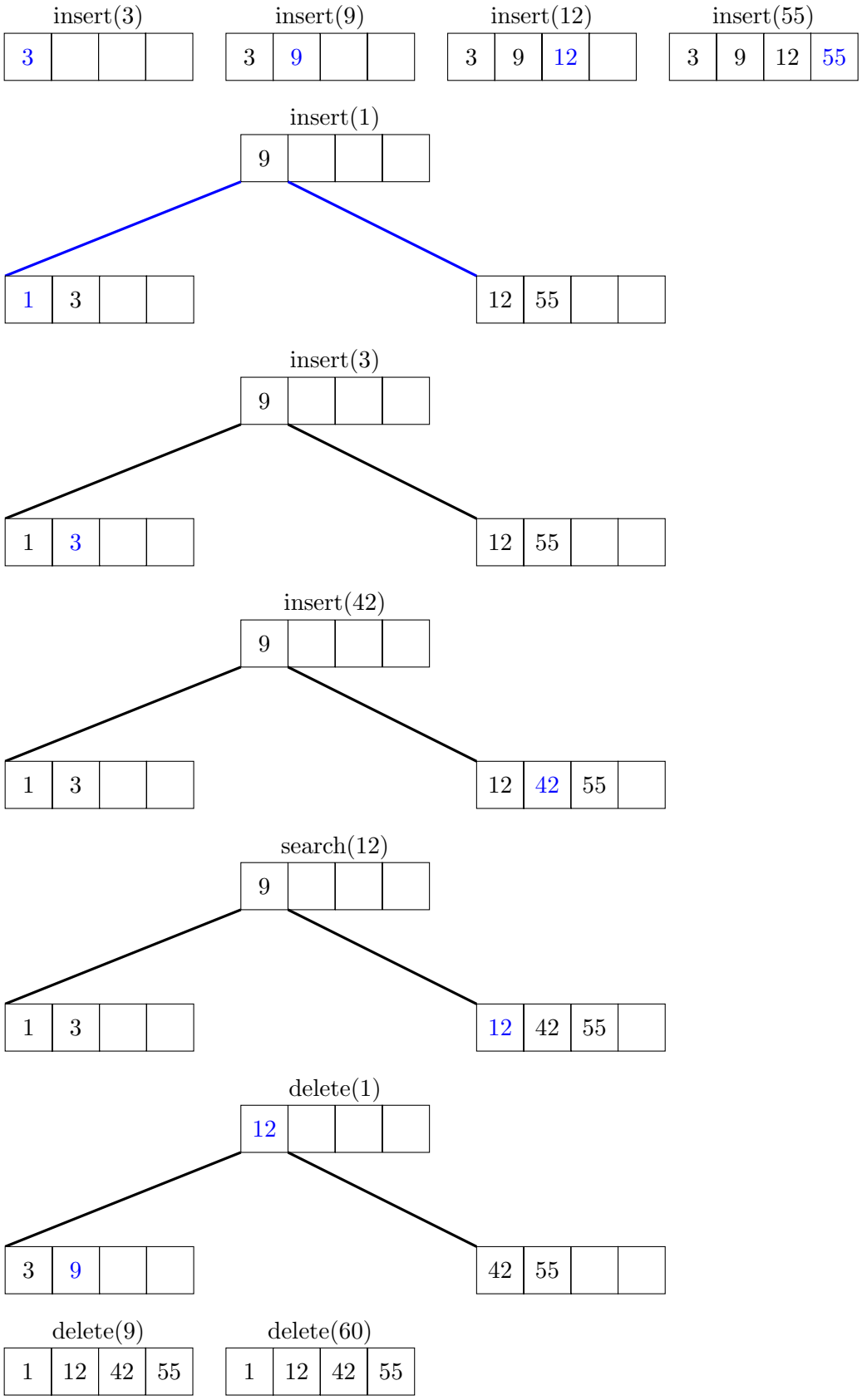
Insertion goes as follows: You insert at a leaf. If the leaf get size $2k + 1$, you divide it into two leafes of size k. The median is inserted into the parent node. This is recursed upwards, as the parent node could also reach size $2k + 1$. If the root exceeds its maximal size, you generate a new root of size 1.

Exercises:

1. Let $k = 2$. Do the following operations: (give graphs)
   `insert(3)`, `insert(9)`, `insert(12)`, `insert(55)`, `insert(1)`, `insert(3)`, `insert(42)`, `search(12)`, `delete(3)`, `delete(9)`, `delete(60)`
2. Proof by induction:
   k-B-Trees with height $h > 0$ (height(root) is 0) have at least $2 * (k + 1)^{(h-1)}$ leaves.

Exercise 1) Solution

1. Graphs:



2. Proof: k-B-Trees with height $h > 0$ (height(root) is 0) have at least $2 * (k+1)^{(h-1)}$ leaves.

   Induction start:
   $h = 1$: $2 * (k+1)^0 = 2$, the root has at least 1 element, so 2 childs.
   $h = 2$: $2 * (k+1)^1 = 2k + 2$, root has at least 2 childs, these have $(k+1)$ childs

   Induction assumption:
   For height h we have at least $2 * (k+1)^{(h-1)}$ leaves. All leaves have same depth.

   Induction step $h \mapsto h+1$:
   $2 * (k+1)^{(h+1-1)} = 2 * (k+1) * (k+1)^{(h-1)} = \#leaves_h * (k+1)$ correct, as each leaf has at least $(k+1)$ children.

Exercise 2) Amortized costs

1. Do an amortized cost analysis for the lecture's ArrayStack implementation. (Push)
2. Why one could be interested in an amortized analysis? It is already in O(N), so what!?

Exercise 2) Solution

1. Do an amortized cost analysis...

```
public void push (int n) {   //k = this.arr.length
  if (this.size == this.arr.length) {                    //+1
    int[] large = new int [this.arr.length * 2 ];        //+2n
    System.arraycopy(this.arr, 0, large, 0, this.size);  //+n
    this.arr = large;                                    //+1
  }
  this.arr[size] = n;                                    //+1
  size++;                                                //+1
}
```

   We prove: each push has an amortized cost of $10 \in \mathcal{O}(1)$ where we choose $G = 6 \cdot (n - \frac{\ell}{2})$ where $n =$ `this.arr.size` is the number of elements stored in the array, and $\ell =$ `this.arr.length`. So, in essence, we count a multiple of the number of elements that are stored in the second half of the array.

   If the array is not extended, the real costs are 3. Moreover, we get $G_{after} - G_{before} = 6 \cdot (n + 1 - \frac{\ell}{2}) - 6 \cdot (n - \frac{\ell}{2}) = 6$. Thus, the amortized costs are $3 + 6 \leq 10$.

   If a new array is created then $n = \ell$. Hence, $G_{before} = 6 \cdot \frac{n}{2} = 3n$. Moreover, $G_{after} = 6$ as there is only element in the second half of the new array. And since the real costs are $4 + 3n$, we end up in the amortized costs $(4 + 3n) + 6 - 3n = 10$.

   In the beginning we have an empty stack with $G_0 = -3$ (0 elements stored, array has length 1). After performing $m > 0$ pushes, we obtain $G_m \geq 0$, as the arrays are always at least half-filled. Hence, $G_m \geq G_0$ for all m, and thus, the real costs of a sequence of m pushes are at most $m \cdot \mathcal{O}(1) = \mathcal{O}(m)$.

2. You need some heavy weighted analysis as you want to be as exact as possible, so that you can measure the efficiency of your algorithm and/or datastructure. Many algorithm are quite better through such an analysis, some analysis guide you which algorithm to choose for your specific problem and so on.

Exercise 3) Hashing

1. Do the following in the given order both for (a) closed hashing (open addressing) with linear probing, (b) open hashing. Each have an array size 10 and are using the modulo operator for locating the actual array block to insert into.
   Assume the given objects have got the following hashes, and they are equal if and only if these hashes are the same. (why is this a special case?).

   - Insert: 50, 2, 34, 22, 48, 54, 5, 17, 26
   - Delete: 2, 33, 54
   - Insert: 12, 33

   At each step, state clearly the status of the datastructure.

2. There are open hash and closed hash data structures. List the differences.

3. What kind of hash (open/closed/linear probing/...) is used in Java (`java.util.HashMap<K,V>`, `java.util.HashTable<K,V>`, `java.util.LinkedHashMap<K,V>`)?

4. Hashing can be used in some unexpected situations. This is one used case.
   Write a program which saves a list of Strings into some files decided on its hash. The files are some kind of dictionaries, it is faster to have several files.

   (a) Is this open or closed hashing?

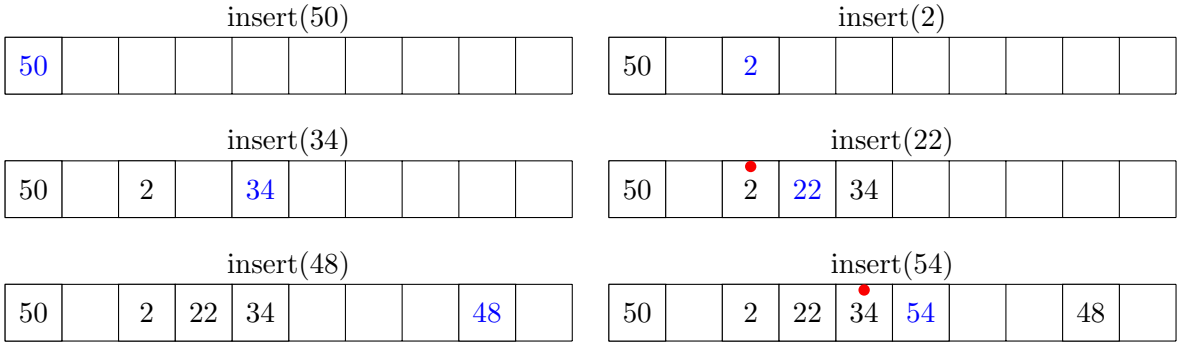
   (b) Design such a write-only data structure in Java.
   Notes: You don't need threads. For files, please have a look into PrintWriter (`http://www.java2s.com/Tutorial/Java/0180__File/WritelinesoftexttofileusingaPrintWriter.htm`).
   This time there is no skeleton given, because the task is to build up one.

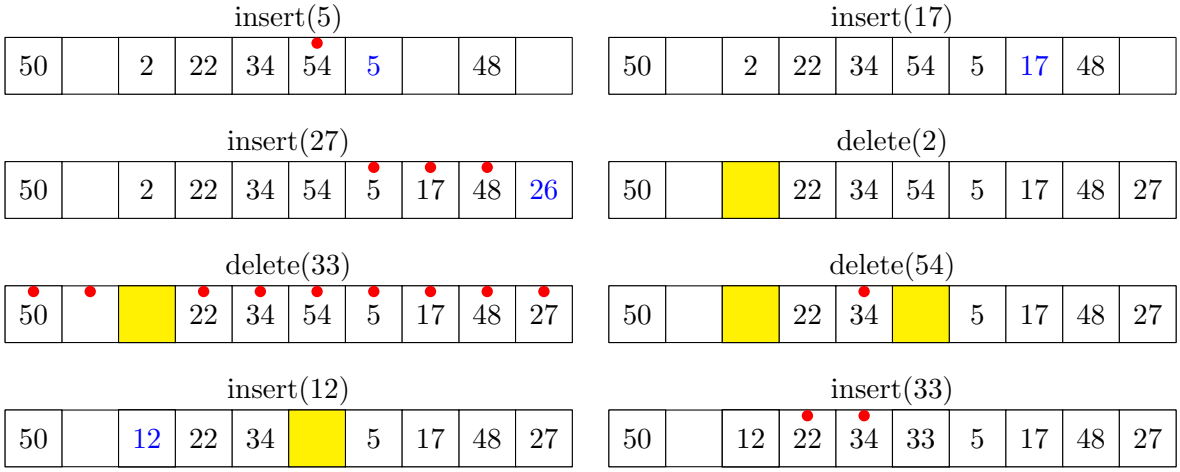
   (c) Write several values into the files and explain what happens.

   Background: If you want to introduce thread parallelism in file-based dictionaries, you will need several files (because you need several so-called locks, in this case they are file-based). You can decide on which file to read/write by using a method like above.

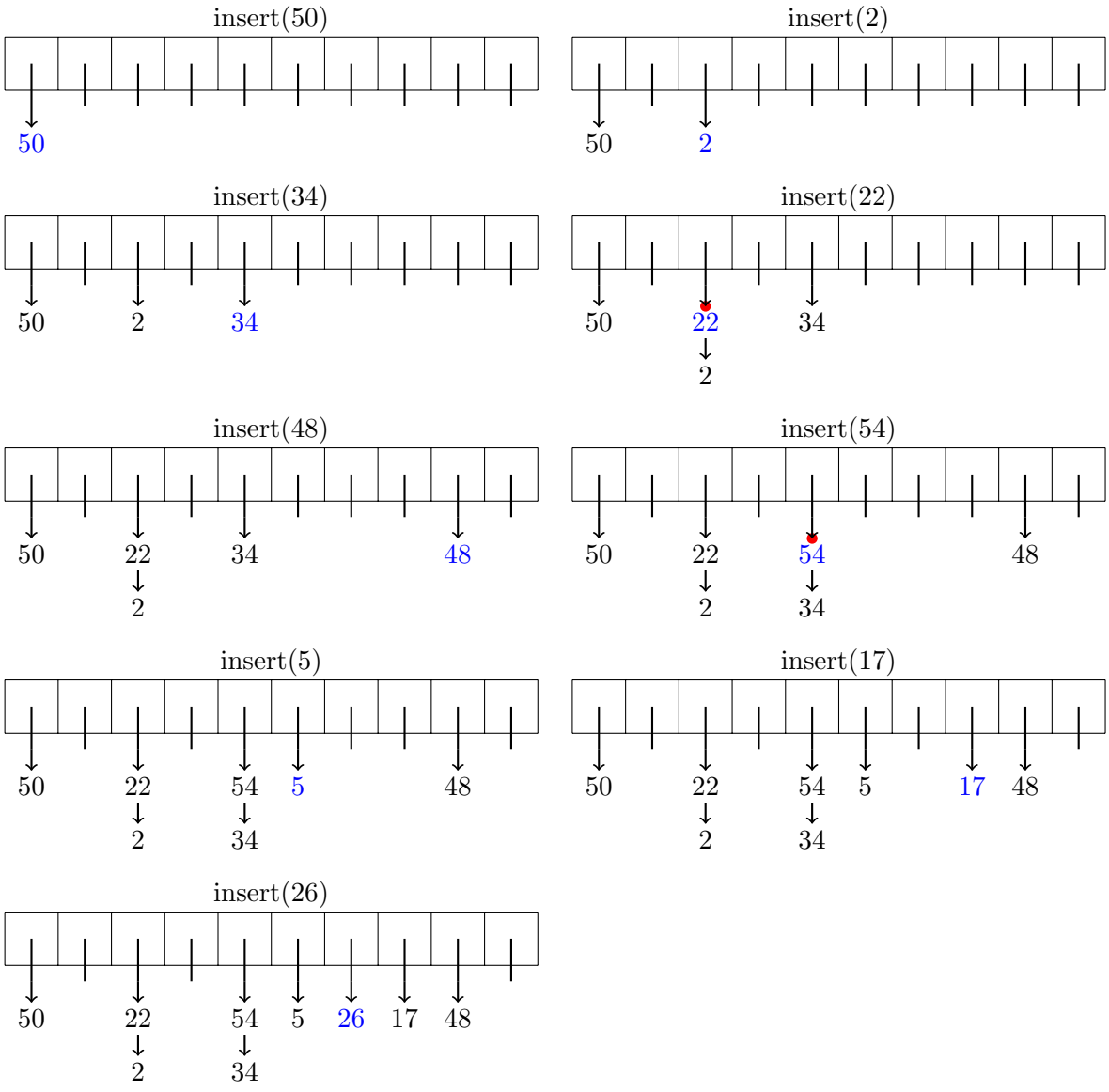
Exercise 3) Solution

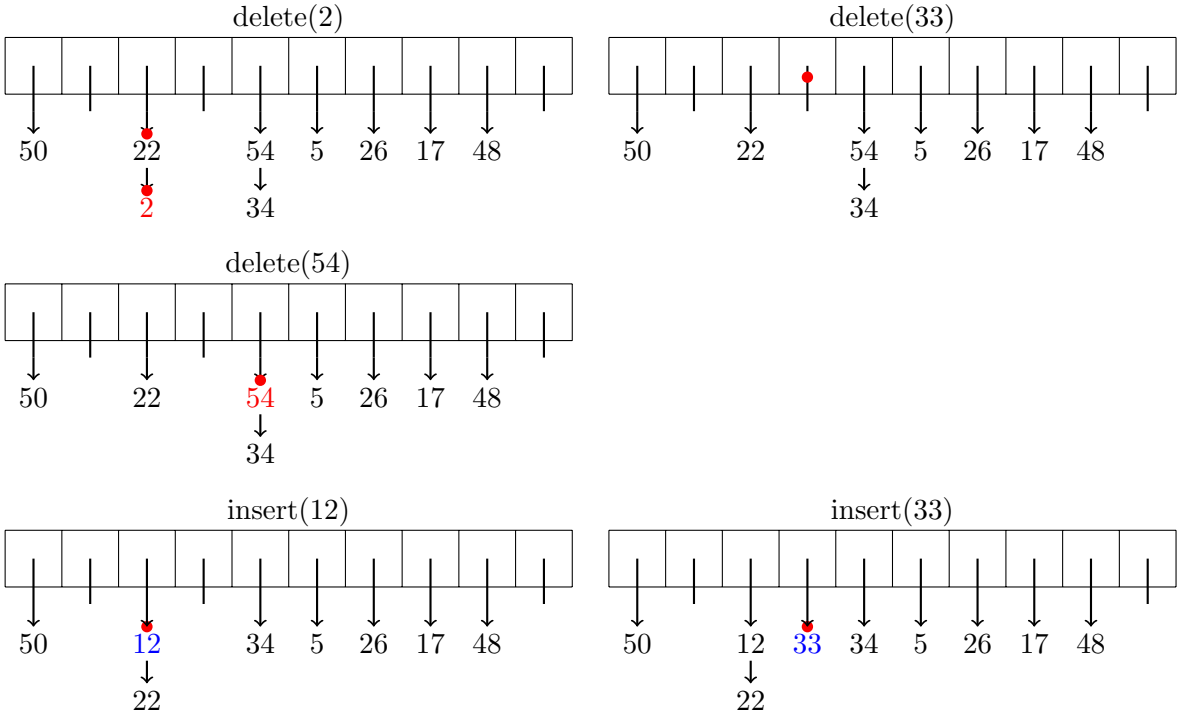
1a) Use closed hashing (open addressing) with linear probing and array size 10. Use the modulo operator for array-size.
Assume objects having same hashes are equal (why is this a special case?).
Special Case: Two elements with the same hash can be different in real world.





1b) Use open hashing with array size 10 and modulo operator for array-size
Assume objects having same hashes are equal.





2 There are open hash and closed hash data structures. List the differences.

(a) Open hashes save the real items seperately (e.g. in a list), closed hashes save them in the hash array-

(b) Open hashes can take more elements than array size, closed hashes can't

(c) For many usage cases open hashes should have an array to be of element count size, sometimes it is fulfilling to have a far smaller array and have an additional data structure. Closed hashes must have a relatively big array to have at least possible double hits.

(d) For open hashes it is not neccessary to have the very best hash function, as long as the elements are clustered in a region (not only one element). At closed hashes it is worst to have a small region where every element is matched to.

(e) Closed hashes are slow on not found elements (with high load up to O(n)!), while open hashes needs at maximum time O(#samehash).

(f) ... (student input)

3 What kind of hash (open/closed/linear probing/...) is used in Java (`java.util.HashMap<K,V>`, `java.util.HashTable<K,V>`, `java.util.LinkedHashMap<K,V>`)?

(a) http://java.sun.com/j2se/1.5.0/docs/api/java/util/Hashtable.html

Note that the hash table is open: in the case of a hash collision, a single bucket stores multiple entries, which must be searched sequentially.

(b) http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html

(The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.)

(c) http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedHashMap.html

[...], assuming the hash function disperses elements properly among the buckets.

(d) ... (student input)

(e) Question to the students: Why it was decided like this?

4 It's open hashing

```
import java.io.*;

//Prints out Strings in different files due to their Hashes
public class HashDS {
    private final int HASH_SIZE = 10;
    PrintWriter[] writers = new PrintWriter[HASH_SIZE];

    /** Constructor which creates the PrintWriters.
      * @param prename String which is added in front of filename.
    */
    HashDS(String prename) {
        try {
            for(int i=0; i<HASH_SIZE; i++) {
                writers[i] = new PrintWriter(new FileWriter(
                        prename+i+".txt", true));
            }
        } catch (IOException e) {
            //Give a good hint and then forget about it...
            e.printStackTrace();
        }
    }

    /** Writes obj.toString() into the file.
      * Using the hashCode to determine which one.
      * @param obj The object to write.
    */
    public void addObject(Object obj) {
        if (obj==null) return;
        int hash = obj.hashCode();
        writers[hash % HASH_SIZE].write(obj.toString());
        writers[hash % HASH_SIZE].flush();
    }
}
```

```
class MyString {
    private String elem;

    MyString(String e) {
        elem = e;
    }

    /** Computes a hash code out of the first two chars.
      * @return Some hash code.
    */
    public int hashCode() {
        if (elem.length()<=1) return 0;
        return elem.charAt(0) + elem.charAt(1);
    }

    /** For the output.
      * @return the elem.
    */
    public String toString() {
        return elem;
    }
}
```

```
public class TestHashDS {

    /** Adds the program parameters into a HashDS.
    */
    public static void main(String[] args) {
        HashDS hds = new HashDS("test");
        for(int i = 0; i<args.length; i++) {
            hds.addObject(new MyString(args[i]));
        }
        //Test Ist Ein Fall overkill
    }
}
```