Algorithms & Datastructures
Laboratory Exercise Sheet 9

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 25th, to be discussed on June 1st

Exercise 1) Questions and Answers Answer the following questions (true or false).

Exercise 1) Solution

- ☑ Quick sort has a worst-case running time of $\Theta(n^2)$.
- ☒ Omitting the pre-processing step of the Quicksort given in the lecture has no impact.
- ☒ Radix sort sorts numbers on their most significant digits.
- ☒ Loop invariants help us understand complexity of algorithms.
- ☑ For the loop invariant we state a condition that must be true before the first iteration of the loop while it remains true before subsequent iteration, and a condition when the loop terminates.
- ☒ Pre-order binary tree traversal, which prints a number of a node key in each step, prints the lowest number first.
- ☒ Binary tree conditions is $p.left.key < p.key > p.right.key$.
- ☑ A minimum $min(p)$ of node p is in a left-most leaf of the $p.left$ subtree.
- ☒ Successor of a node p in a binary tree can always be determined using $min(p.right)$
- ☑ In an AVL tree, for each node x, the heights of the left and right subtrees of x always differ by at most 1.
- ☒ Removing a node from an AVL tree always requires rotation.
- ☒ Rotation in an AVL tree always changes the height of the tree.
- ☑ In a B-tree, an internal node x that contains $x.n$ keys always has $x.n+1$ children.
- ☑ All leaves in a B-tree are in the same depth.
- ☒ Every node in a B-tree with the minimum degree of 2 must have at least 1 key and may contain at most 4 keys (therefore the internal node may have at most 4 children).

Exercise 2) Hashing

Exercise 2) Solution

1. Universal hashing is possible for the Triple-type (assuming that X, Y, and Z use universal hashing), but not for the lists. The reason for the latter is that there is no bound on the list length, and hence, one cannot bound the size of $\mathcal{K}$ by some fixed number.

2. For the Triple-type we used universal hashing where the bit-length is 3*32 = 96. The large number is a generated prime number of 97 bits.

```
import java.math.*;
import java.util.*;

public class Triple<X,Y,Z> {

   private final static BigInteger p = new BigInteger("
       95673531099682755594247122307");
   private final static Random r = new Random();
   private final static BigInteger a = new BigInteger(3*32, r).add(
       BigInteger.ONE);
   private final static BigInteger b = new BigInteger(3*32, r);

   X x;
   Y y;
   Z z;

   public boolean equals(Object other) {
      if (other instanceof Triple) {
         Triple<X,Y,Z> t = (Triple<X,Y,Z>)other;
         return this.x.equals(t.x) && this.y.equals(t.y) && this.z.
             equals(t.z);
      } else {
         return false;
      }
   }

   public int hashCode() {
      int hx = x.hashCode();
      int hy = y.hashCode();
      int hz = z.hashCode();
      BigInteger n = BigInteger.valueOf(hx);
      n = n.shiftLeft(32);
      n = n.or(BigInteger.valueOf(hy));
      n = n.shiftLeft(32);
      n = n.or(BigInteger.valueOf(hz));
      n = a.multiply(n).add(b).mod(p);
      return n.intValue();
   }

}
```

For the list-type we used a generated large integer prime number. We start with a hashvalue of 0, and then, everytime an element is encountered, we multiply the hashvalue by this prime and add the element. In this way, every element influences the hashvalue (and also the position of an element is important), and we use the full range of integers. E.g., even if the list only contains small numbers, the hashvalue will be some arbitrary integer, which would not be the case if we just XOR all numbers.

```
public class List {
   Element root;

   public boolean equals(Object other) {
      if (other instanceof List) {
         Element one = this.root;
```

```
            Element two = ((List)other).root;
            while (one != null && two != null) {
               if (one.x != two.x) {
                  return false;
               }
               one = one.next;
               two = two.next;
            }
            return (one == null && two == null);
         } else {
            return false;
         }
      }


      public int hashCode() {
         Element e = root;
         int hc = 0;
         while (e != null) {
            hc = hc * 2082205841 + e.x; // mult. with prime
         }
         return hc;
      }
   }


   class Element {
      int x;
      Element next;

      Element(int x, Element next) {
         this.x = x;
         this.next = next;
      }
   }
```