Algorithms & Datastructures
Laboratory Exercise Sheet 6

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>

April 27th, to be discussed on May 4th

Exercise 1) Tree Traversal The tree iterator from the lecture only supports traversal of trees, but did not allow to modify the underlying data-structure. For this exercise you should also implement the `remove()`-method in the iterator with the following conventions (adapted from the Java Iterator-Interface).

- `remove` may only be invoked after a call to `next`.
- If `remove` is called, then the last element that has been returned by `next` is removed from the tree. It is not allowed to call `remove` twice without an intermediate call to `next`.
- If `remove` is called wrongly, an `IllegalStateException` should be thrown.
- After a (legal) call to `remove`, the iterator should proceed normally, i.e., in total, all elements must be traversed and no element should be returned twice.

For example, if there is a tree with elements 4,7,8,10,11 then after calling `next` twice, the iterator may return first the 4 and then the 8 (or other values, depending on the structure of the tree and the traversal-order). A call of `remove` should now delete the 8 from the tree. Two more calls to `next` return the values 11 and 7. Another call to `remove` is possible and removes the 7. Finally, one more call to `next` is possible and it returns the element 10.

1. In the sources to this exercise sheet you already find a slightly simplified binary tree implementation from the lecture where the iterator only supports inorder-traversal. Modify this implementation such that the iterator supports the `remove`-operation. (You might also need to change other methods, please indicate which methods have been changed.)
2. Would your implementation also work correct for preorder-traversal or for postorder-traversal? If not, try to illustrate the problem (think about removal of nodes with two children).

Exercise 1) Solution

1. Essentially, we store the node that was previously returned by `next`. If there is such a node we remove it by the `removeNode`-method from the `BinTree`-class and set the previously returned node to `null`. Otherwise, `remove` was called wrongly and we throw an exception.

   However, to be able to call the `removeNode`-method, we have to also store the tree in the iterator. Therefore, the signature of the constructor of the `TreeIterator` has to be changed, too, and likewise the invocation of this constructor within the `BinTree`-class.

   In total, we have the following changes (in lines 4,5,7,8,9,11,19,26-31,37):

```
1  public class TreeIterator<D> implements Iterator<D> {
2     int direction;   // from which direction ...
3     Node<D> current; // ... did we enter current node
4     Node<D> previous; // the node we previously returned by next()
5     BinTree<D> tree; // the tree we are traversing
6
7     public TreeIterator(BinTree<D> tree) {
8        this.current = tree.root;
9        this.tree = tree;
10       this.direction = UP;
11       this.previous = null;
12       walkNext(INORDER);
13    }
14    public Entry<D> next() {
15       if (current == null) {
16          throw new NoSuchElementException();
17       } else {
18          Entry<D> entry = new Entry<D>(current.key, current.data);
19          this.previous = this.current;
20          walkNext(NONE);
21          return entry;
22       }
23    }
24
25    public void remove() {
26       if (this.previous == null) {
27          throw new IllegalStateException();
28       } else {
29          this.tree.removeNode(this.previous);
30          this.previous = null;
31       }
32    }
33
34 }
```

```
35 public class BinTree<D> implements Dictionary<D> {
36    public Iterator<D> iterator() {
37       return new TreeIterator<D>(this);
38    }
39
40 }
```

2. The iterator does not work correctly for the preorder traversal. Consider a tree that is obtained after inserting values 3,1,2,4 in that order. Then an initial call to `next` will return the 3 and the `current`-reference is at the node with the 1. Now calling `remove` will remove the node with the 3 which has two children, so the 3 and the 2 are swapped and then the 3 is deleted (where now the 2 is the root of the node). As the iterator has already output

the root-node, the following calls to `next` will result in 1 and 4, so the 2 is never returned by the iterator.

The iterator will however work correctly for the postorder traversal: whenever a value in a node is returned then the iterator will not revisit and output nodes in the corresponding subtree. Hence, it is safe to remove such a node. (A similar reasoning also shows that there are no problems using the inorder-traversal.)

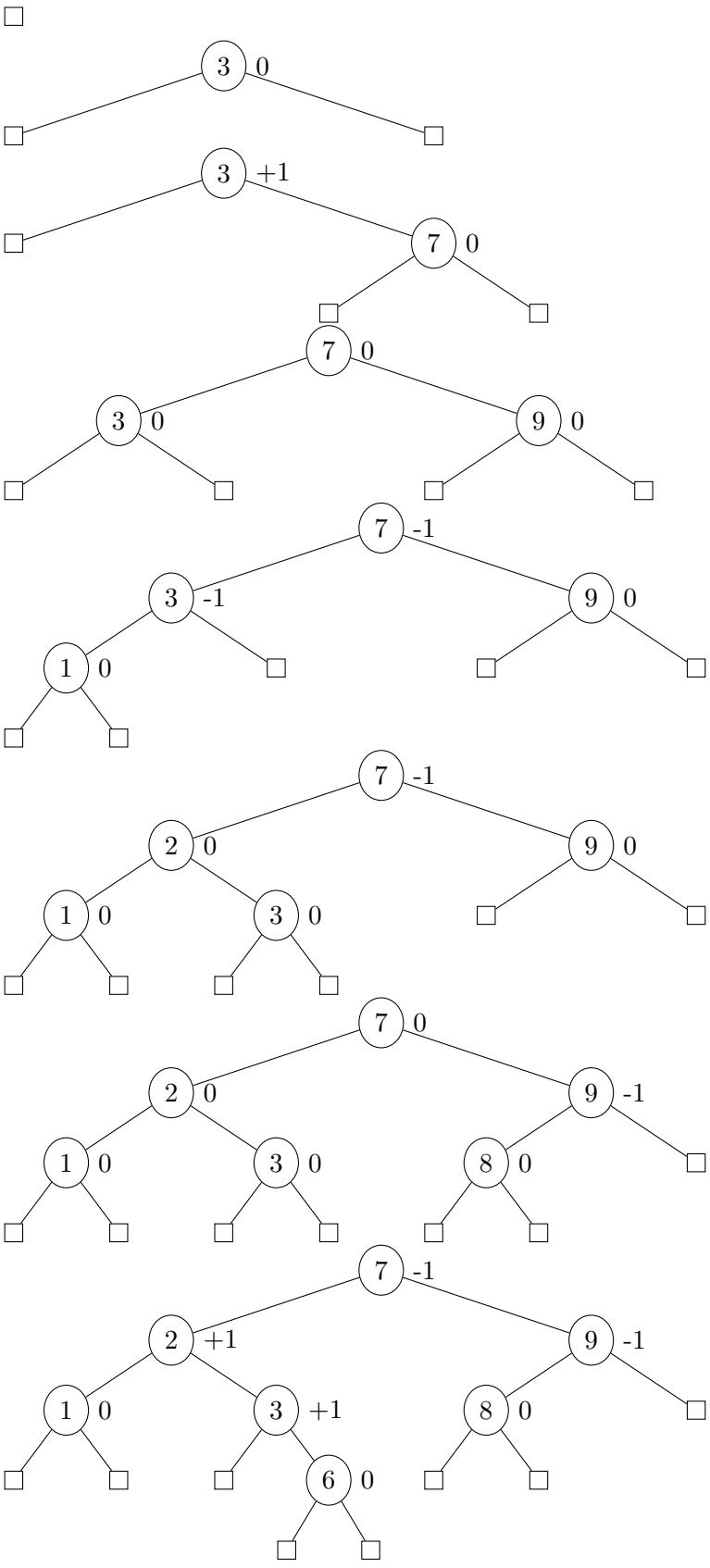
Exercise 2) AVL Trees

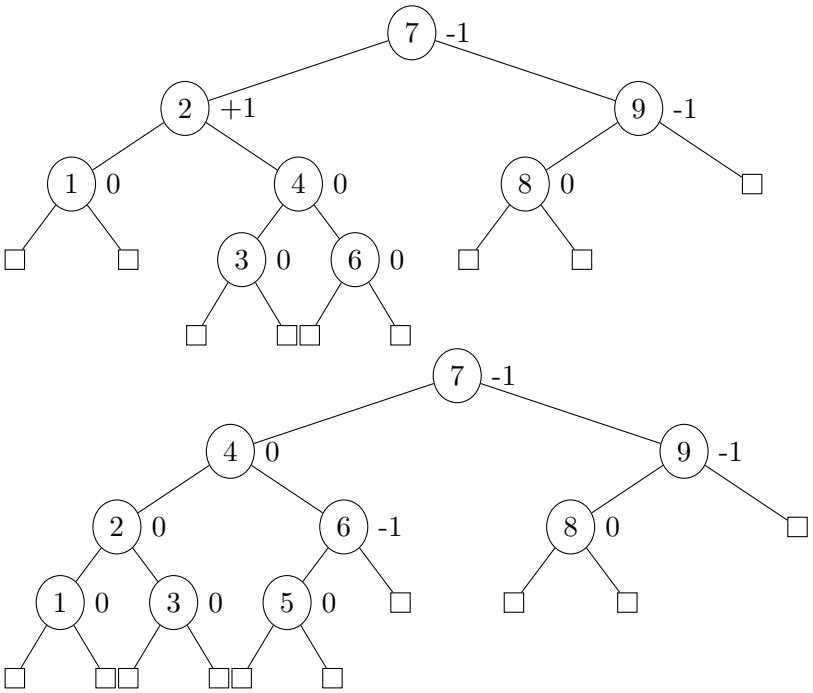
1. Insert the values 3,7,9,1,2,8,6,4,5 into an empty AVL-tree (in the given order). Give the tree after each insertion.
2. In the picture on slide 163 the new `hdiff`-values for nodes `n` and `p` are missing. Determine these values by only looking at the other `hdiff`-values, but without computing the heights of the trees.
3. Implement the method `rotateLeft` for AVL-tree nodes. This method should update all references to children, parents, etc., but should ignore the `hdiff`-values.
4. Implement `balGrow` where you can omit the symmetric cases where `n = p.right`. Moreover, you can assume that both `rotateLeft` and `rotateRight` are available.

An incomplete implementation of AVL-trees is given in the sources of this exercise sheet.

Exercise 2) Solution

1.





2.-4.

```java
public class AVLTree<D> implements Dictionary<D> {
   AVLNode<D> root;
   /**
    * does not update the balance-informations
    */
   void rotateRight(AVLNode<D> p) {
      AVLNode<D> l = p.left;
      l.parent = p.parent;
      p.left = l.right;
      l.right = p;
      p.parent = l;
      if (p.left != null) {
         p.left.parent = p;
      }
      if (l.parent == null) {
         this.root = l;
      } else if (l.parent.left == p) {
         l.parent.left = l;
      } else {
         l.parent.right = l;
      }
   }

   /**
    * does not update the balance-informations
    */
   void rotateLeft(AVLNode<D> p) {
      AVLNode<D> r = p.right;
      r.parent = p.parent;
      p.right = r.left;
      r.left = p;
      p.parent = r;
      if (p.right != null) {
         p.right.parent = p;
      }
      if (r.parent == null) {
         this.root = r;
```

```
38          } else if (r.parent.left == p) {
39              r.parent.left = r;
40          } else {
41              r.parent.right = r;
42          }
43      }
44
45      void balGrow(AVLNode<D> n) {
46          while (n.parent != null) {
47              AVLNode<D> p = n.parent;
48              if (p.left == n) {
49                  if (p.hdiff == 1) {
50                      p.hdiff = 0;
51                      return;
52                  } else if (p.hdiff == 0) {
53                      p.hdiff = -1;
54                      n = p;
55                  } else if (n.hdiff == -1) { // and p.hdiff = -1
56                      rotateRight(p);
57                      p.hdiff = 0;
58                      n.hdiff = 0;
59                      return;
60                  } else { // n.hdiff = 1, p.hdiff = -1
61                      AVLNode<D> r = n.right;
62                      rotateLeft(n);
63                      rotateRight(p);
64                      n.hdiff = r.hdiff ==  1 ? -1 : 0;
65                      p.hdiff = r.hdiff == -1 ?  1 : 0;
66                      r.hdiff = 0;
67                      return;
68                  }
69              } else { // p.right = n
70                  if (p.hdiff == -1) {
71                      p.hdiff = 0;
72                      return;
73                  } else if (p.hdiff == 0) {
74                      p.hdiff = 1;
75                      n = p;
76                  } else if (n.hdiff == 1) { // and p.hdiff = 1
77                      rotateLeft(p);
78                      p.hdiff = 0;
79                      n.hdiff = 0;
80                      return;
81                  } else { // n.hdiff = -1, p.hdiff = 1
82                      AVLNode<D> l = n.left;
83                      rotateRight(n);
84                      rotateLeft(p);
85                      n.hdiff = l.hdiff == -1 ?  1 : 0;
86                      p.hdiff = l.hdiff ==  1 ? -1 : 0;
87                      l.hdiff = 0;
88                      return;
89                  }
90              }
91          }
92      }
93  }
```