JOHANNES KEPLER
UNIVERSITÄT LINZ

Algorithms and Data Structures 2
Winter term 2019
Stefan Grünberger, Dari Trendafilov

Assignment 05

Deadline: **Mon. 3.12.2019, 23:59**
Submission via: **www.pervasive.jku.at/Teaching/**

Trees

1. AVL tree — 24 points

Implement the **insertion and removal** of node in an AVL tree according to the procedure presented in lecture and exercise. Use the following class **AVLTree**:

```
public class AVLTree {

  protected AVLNode root;
  protected int size;

  /**
   * Default constructor.
   * Initializes the AVL tree.
   */
  public AVLTree() {. . .}

  /*
   * @return the root node of the AVL tree
   */
  public AVLNode getRoot() {. . .}

  /**
   * Retrieves tree height.
   * @return -1 in case of empty tree, current tree height otherwise.
   */
  public int height() {. . .}

  /**
   * Inserts a new node into AVL tree.
   * @param key Key of the new node. May not be null. Elements with the same key are not allowed,
   * in this case false is returned.
   * @param elem Data of the new node. May not be null.
   * @return true if insert was successful, false otherwise.
   */
  public boolean insert (Integer key, String elem) throws IllegalArgumentException {. . .}

  /**
   * Removes the first node with given key.
   * @param key Key of node to remove. May not be null.
   * @return true, if element was found and deleted.
   */
  public boolean remove (Integer key) throws IllegalArgumentException {. . .}

  /**
   * Returns value of a first found node with given key.
   * @param key Key to search. May not be null.
   * @return Corresponding value if key was found, null otherwise.
   */
  public String find (Integer key) throws IllegalArgumentException {. . .}


  /**
   * Returns the number of key/value pairs in the tree.
   * @return Number of key/value pairs.
   */
  public int size() {. . .}

  /**
   * Returns an array representation of the data elements (pre-ordered).
   * @return Array representation of the tree.
   */
  public Object[] toArray() {. . .}
  . . .
}
```

When searching for nodes x, y, and z (see slides of exercise 05), go up from the node you just inserted. The nodes store a reference to their parent node:

```
public class AVLNode {
```

Assignment 05


```
    public AVLNode parent = null;
    public AVLNode left = null;
    public AVLNode right = null;

    public Integer key;
    public String elem;

    public int height = 0; // To determine node height in O(1)

    public AVLNode(Integer key,  String elem) {
        this.key = key;
        this.elem = elem;
    }

    public AVLNode(Integer key) {
        this.key = key;
    }

@Override
public String toString() {
        return key + " " + elem;
    }
}
```

Hints:

You can build on the methods of the binary search tree of the last assignment and extend them to implement the AVL tree. For the implementation of `insert` an auxiliary function

```
/**
 * Implements cut & link restructure operation.
 * @param n Node to start restructuring with.
 */
private void restructure (AVLNode n) {. . .}
```

might be useful, which performs a one-time **Cut&Link restructuring** (if necessary) starting from a given node n upwards. This method is called after insertion of a node.
In contrast, after removing an element with `remove`, the restructuring – starting with the parent node of the removed Inorder successor – has to be continued **up to the root nod**e, since a restructuring can violate the balancing on higher levels.

Consider if further auxiliary methods are useful for improving the readability of `restructure` e.g.:

```
/**
 * Checks AVL integrity.
 * @param n Node to check integrity for.
 * @return true If AVL integrity is sane, false otherwise.
 */
private boolean isAVLTree (AVLNode n) {. . .}
```

to check if a given subtree is balanced. To achieve a query of the height in O(1), an update of the heights of all affected nodes must be made in `insert`. Herefore an auxiliary method

```
/**
 * Updates node heights starting from given node.
 * @param n Node to start update height operation with.
 */
private void updateHeights (AVLNode n) {. . .}
```

might be useful, which updates the height starting from a given node n and the nodes above. Furthermore, you are free to declare and use other (private) data structure, such as a class that manages x,y,z nodes, e.g.:

```
/**
 * Helper class to manage three AVL nodes.
 */
private class NodeGroup {
  AVLNode cur, parent, grandparent = null;
}
```