University of Glasgow

Systems Programming

AVL Trees | 15th of October 2018

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk

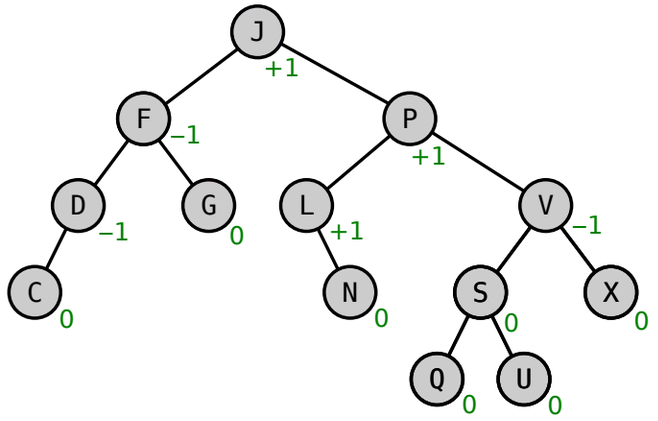
AVL Trees

- An AVL tree is a *self-balancing binary search tree*
- The `balance` factor is the difference between the height of the left and right subtree:

```
node.balance = height(node.right) - height(node.left);
```

- A node is *balanced* if the height of left and right subtrees differ at most by one (balance factor is 0, -1, or +1)
- A binary search tree is an *AVL tree* if every node in the tree is balanced

AVL tree:



Inserting nodes

- When inserting into an AVL tree we have to *rebalance* the tree if the balancing property is violated

```
public boolean insert(int key) {
    if (root == null) { root = new Node(key, null); return true; } // inserting in empty tree

    Node n = root;
    while (true) {
        if (n.key == key) { return false; } // node with same key is already in the tree

        Node parent = n;

        boolean goLeft = n.key > key;
        n = goLeft ? n.left : n.right;

        if (n == null) { // once we have reached the leafs of the tree we insert
            if (goLeft) { parent.left = new Node(key, parent);  }
            else        { parent.right = new Node(key, parent); }
            rebalance(parent); // now we have to rebalance the parent node!
            break;       // and exit the tree traversal
        }
    }
    return true;
}
```

Rebalancing

- There are four different types of *rotations* to be performed for different cases

```
private void rebalance(Node n) {
    setBalance(n); // we first compute the balance of the node

    // check if we have violated the balancing property
    if (n.balance == -2) {          // the left subtree is to high
        if (height(n.left.left) >= height(n.left.right))   { n = rotateRight(n);         }
        else                                               { n = rotateLeftThenRight(n); }
    } else if (n.balance == 2) {    // the right subtree is to high
        if (height(n.right.right) >= height(n.right.left)) { n = rotateLeft(n);          }
        else                                               { n = rotateRightThenLeft(n); }
    }

    // after this node has been rebalanced we work the tree upwards and continue to rebalance
    if (n.parent != null) { rebalance(n.parent); }
    else                  { root = n;            }
}
```

Rotating Left

- We are looking at one case `rotateLeft`. The `rotateRight` case is similar

```
private Node rotateLeft(Node x) {
    Node z = x.right;
    z.parent = x.parent;

    x.right = z.left;

    if (x.right != null) { x.right.parent = x; }

    z.left = x;
    x.parent = z;

    if (z.parent != null) {
        if (z.parent.right == x) { z.parent.right = z; }
        else                     { z.parent.left  = z; }
    }

    setBalance(x);
    setBalance(z);
    return z;
}
```

Combining rotations

- `rotateRightThenLeft`

```
private Node rotateRightThenLeft(Node n) {
    n.right = rotateRight(n.right);
    return rotateLeft(n);
}
```

- `rotateLeftThenRight`

```
private Node rotateLeftThenRight(Node n) {
    n.left = rotateLeft(n.left);
    return rotateRight(n);
}
```

Deleting nodes

- Rebalancing has also to be performed when deleting nodes
- I leave the details to the Java implementation posted on moodle

Systems Programming

AVL Trees | 15th of October 2018

Michel Steuwer | http://michel.steuwer.info | michel.steuwer@glasgow.ac.uk