326.041 (2015S) – Practical Software Technology

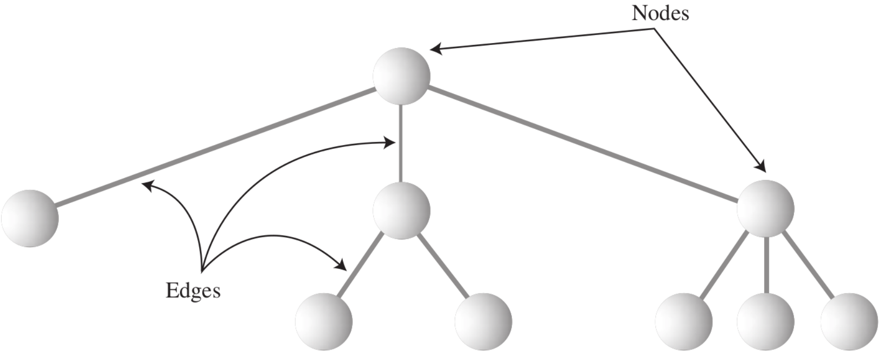
(Praktische Softwaretechnologie)

Binary Search Trees, Red-Black Trees

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

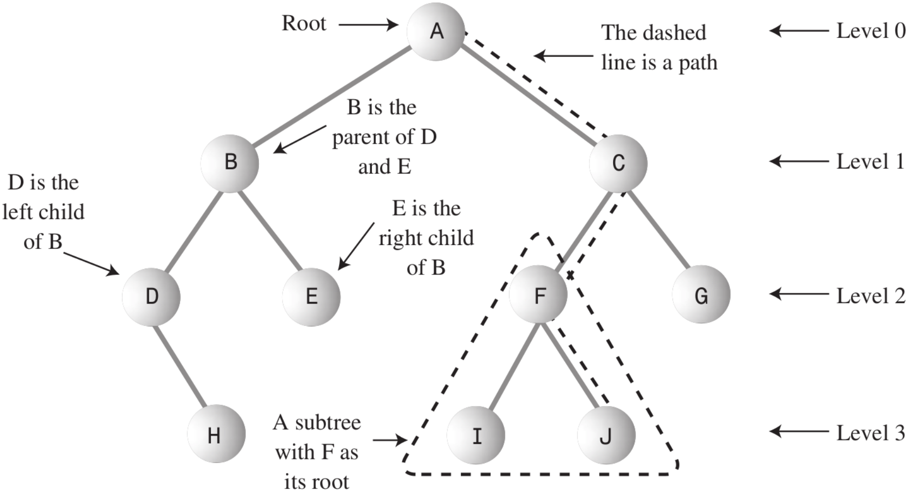
Tree: Nodes and Edges



Definition

- A **tree** T is a set of nodes storing elements such that the nodes have a **parent-child relationship** that satisfies the following properties:
  - If T is nonempty, it has a special node, called the **root** of T, that has no parent.
  - Each node v of T different from the root has a **unique parent** node w; every node with parent w is a child of w.
- A tree can be empty.
- A node without children is a **leaf**.

Basic Notions

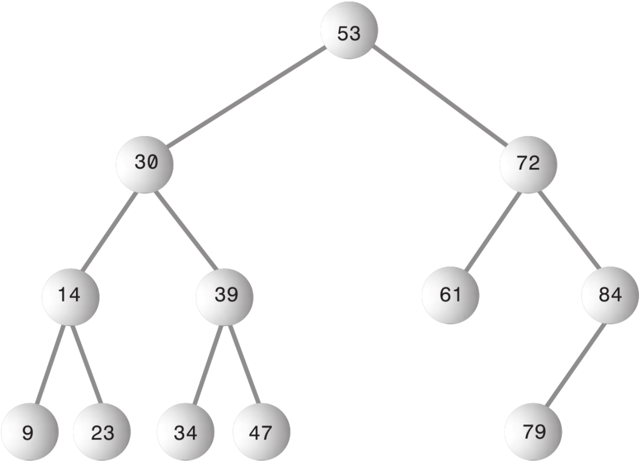


Binary Trees

- A tree where every node has at most **two children** is a **binary tree**.
- Children are called the **left child** and the **right child**.

Binary Search Trees

- A node's **left** child must have a key **less** than its parent.
- A node's **right** child must have a key **greater or equal** to its parent.

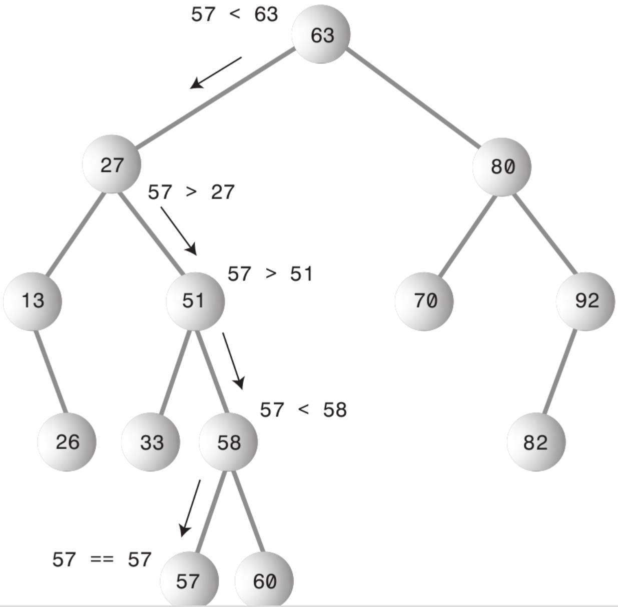


Node.java – Generic Implementation

```java
public class Node<E extends Comparable<E>>
               implements Comparable<Node<E>> {
    E data;
    Node<E> leftChild;
    Node<E> rightChild;

    public int compareTo(Node<E> o) {
        if (data == null)
            return o.data == null ? 0 : -1;
        return o.data==null ? 1 : data.compareTo(o.data);
    }
}
```

Finding a Node



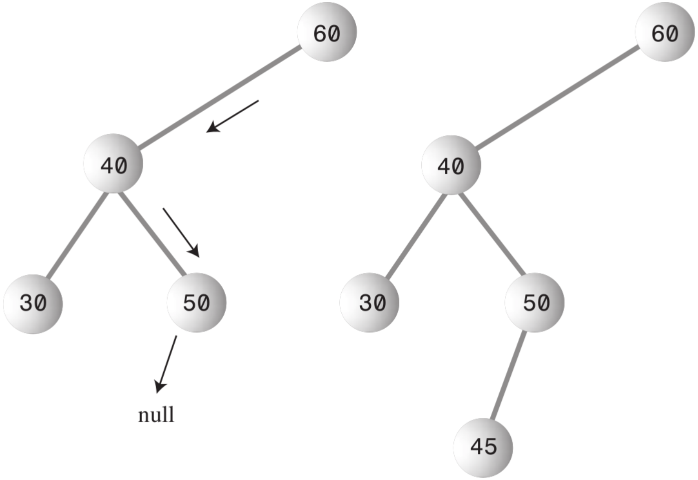
Tree.java – Finding a Node

```
public class Tree<E extends Comparable<E>> {
    Node<E> root;
    ...

    public Node<E> find(E data) {
        Node<E> cur = root;
        while (cur != null && data.equals(cur.getData())){
            if (cur.getData().compareTo(data) > 0)
                cur = cur.leftChild;
            else
                cur = cur.rightChild;
        }
        return cur;
    }
}
```

Inserting a Node

Find an appropriate position to insert a node as leaf:

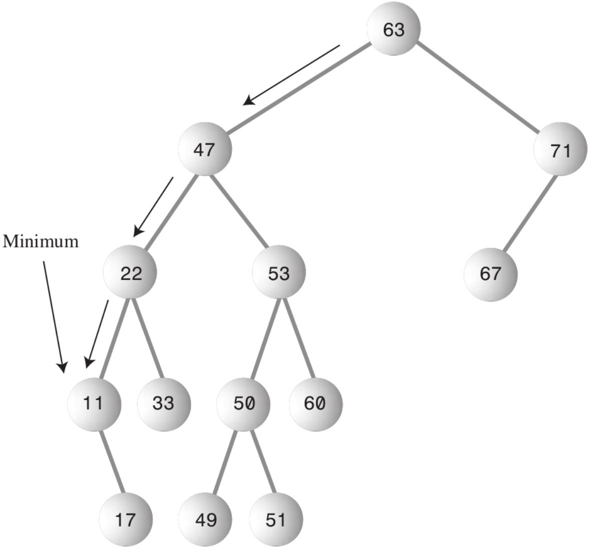


Tree.java – Inserting a Node


```java
public void insert(E data) {
    Node<E> newNode = new Node<>(data);
    if (root == null) {
        root = newNode;
        return;
    }
    for (Node<E> current = root; current != null;) {
        Node<E> parent = current;
        if (current.compareTo(newNode) > 0) {
            current = current.leftChild;
            if (current == null)
                parent.leftChild = newNode;
        } else {
            current = current.rightChild;
            if (current == null)
                parent.rightChild = newNode;
        }
    }
}
```

Find Minimum / Maximum

Follow the left / right child as long as possible.



Tree.java – Finding the Minimum


```java
public Node<E> minimum() {
    Node<E> last = null;
    for (Node<E> n = root; n != null; n = n.leftChild)
        last = n;
    return last;
}
```

Maximum is similar.

Alexander.Baumgartner@risc.jku.at

Traversing the Tree: Depth-First

Depth-first-traversal:

- **Preorder:**
  - Visit the node,
  - Traverse the nodes left subtree,
  - Traverse the nodes right subtree.
- **Inorder:**
  - Traverse the nodes left subtree,
  - Visit the node,
  - Traverse the nodes right subtree.
- **Postorder:**
  - Traverse the nodes left subtree,
  - Traverse the nodes right subtree,
  - Visit the node.

Tree.java – Inorder Traversal I


```java
// Inner class has access to the type variable
public abstract class Visitor {
    public abstract void visit(Node<E> node);
}

public void inOrder(Visitor visitor) {
    inOrder(root, visitor);
}

private void inOrder(Node<E> current, Visitor visitor) {
    if (current == null)
        return;
    inOrder(current.leftChild, visitor);
    visitor.visit(current);
    inOrder(current.rightChild, visitor);
}
```

Tree.java – Inorder Traversal II


```
public class Tree<E extends Comparable<E>> {
    ...
    // Inner class has access to the type variable
    public abstract class Visitor {
        public abstract void visit(Node<E> node);
    }
```

```
Tree<Integer> t = new Tree<>();
t.insert(5);
t.insert(11);
...

final StringBuilder sb = new StringBuilder();
t.inOrder(t.new Visitor() {
    public void visit(Node<Integer> node) {
        sb.append(node.getData()).append(' ');
    }
});
System.out.println(sb.toString());
```

Traversing the Tree: Breadth-First

Breadth-first-traversal:

- **Levelorder:**
  Use a Queue to go through the tree level-by-level.
  1. Start with the root node and visit it (Level 0).
  2. Visit the left child, unless it is null.
     - Put it into the Queue.
  3. Visit the right child, unless it is null.
     - Put it into the Queue.
  4. Poll the next node from the Queue and go to 2.

Tree.java – Levelorder Traversal


```
public void levelOrder(Visitor visitor) {
    Queue<Node<E>> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node<E> current = queue.poll();
        visitor.visit(current);
        if (current.leftChild != null)
            queue.add(current.leftChild);
        if (current.rightChild != null)
            queue.add(current.rightChild);
    }
}
```

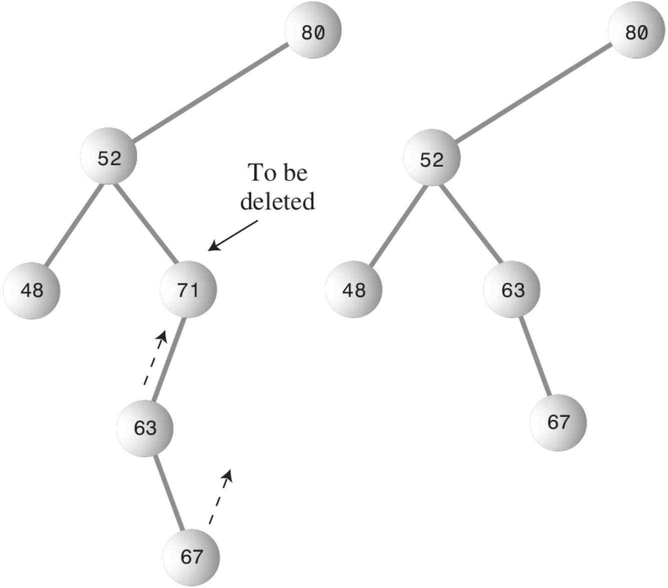
Deleting a Node I

Three cases:

- The node to be deleted is a leaf.
- The node to be deleted has one child.
- The node to be deleted has two children.

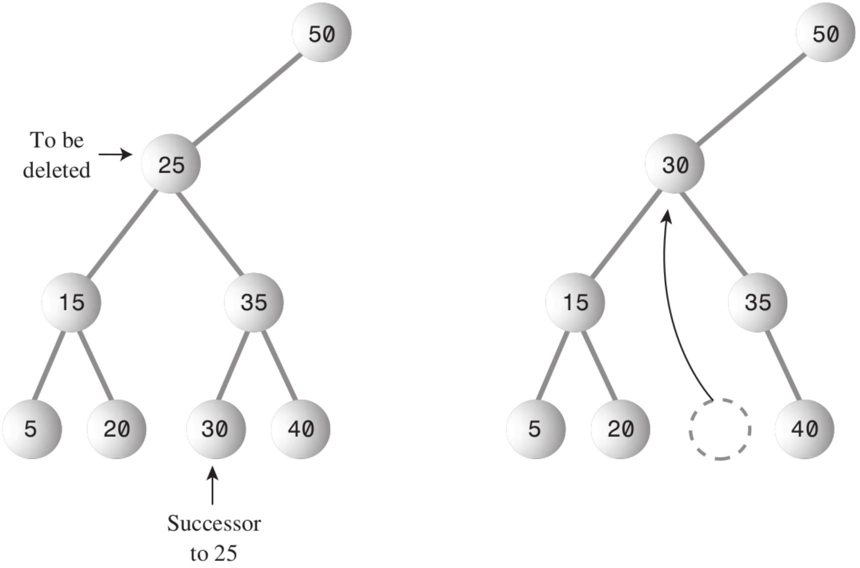
The first case is trivial.

Deleting a Node II

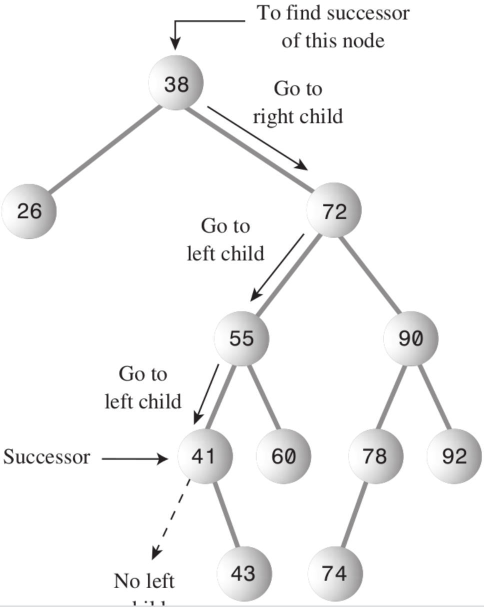


Alexander.Baumgartner@risc.jku.at

Deleting a Node III



Deleting a Node IV



Tree.java – Finding the Successor


```java
private Node<E> getSuccessor(Node<E> delNode) {
    Node<E> successorParent = delNode;
    Node<E> successor = delNode;
    // go to right child
    for (Node<E> n = delNode.rightChild; n != null;) {
        successorParent = successor;
        successor = n;
        n = n.leftChild;
    }
    if (successor != delNode.rightChild) {
        successorParent.leftChild = successor.rightChild;
        successor.rightChild = delNode.rightChild;
    }
    return successor;
}
```

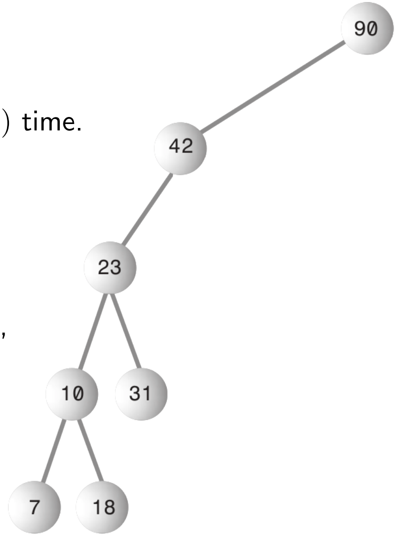
Tree.java – Deleting a Node


```java
public class Tree<E extends Comparable<E>> {
    ...
    /**
    * Returns true in case of success and false
    * if the given node was not found.
    */
    public boolean delete(E node) {
        see Exercise 8
    }
    private Node<E> getSuccessor(Node<E> delNode) {
    ...
}
```


Alexander.Baumgartner@risc.jku.at

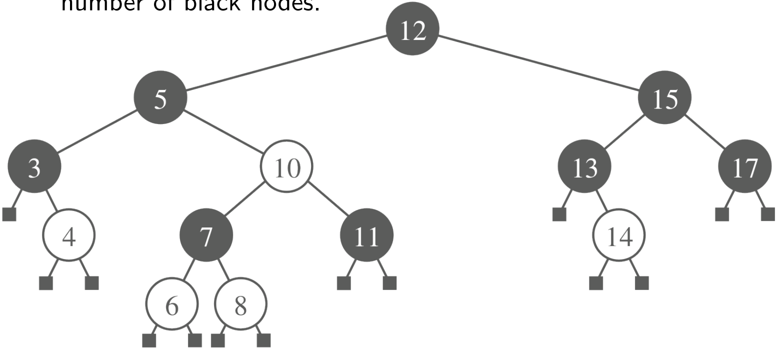
Out of Balance?

- If the binary tree is balanced, find, insert, delete needs $O(\log n)$ time.
- What happens if the values to be inserted are already ordered?
- Binary trees might **become unbalanced** over time.
- The ability to quickly find, insert, delete a given element is lost.



Solution: Red-Black Trees

- A **red-black tree** is a binary search tree with **colored nodes**.
- It uses O(1) structural changes after an update to stay **balanced**.
- The height of a red-black tree storing n entries is $O(\log n)$.
  1. Every node is either red or black.
  2. The root is always black.
  3. If a node is red, its children must be black.
  4. Every path from the root to a "null child", must contain the same number of black nodes.



Keeping the Balance

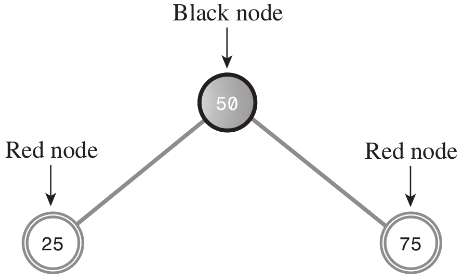
The rules ensure that the height is bound by $O(\log n)$.

1. Every node is either red or black.
2. The root is always black.
3. If a node is red, its children must be black.
4. Every path from the root to a "null child", must contain the same number of black nodes.

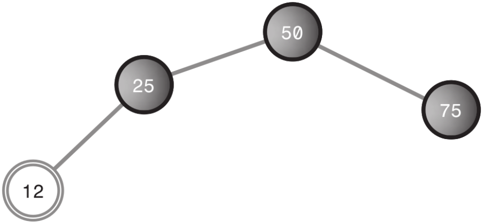
To keep this rules intact:

- You can **change the colors** of nodes.
- You can **perform rotations**. Rotations must do two things at once:
  - Raise some nodes and lower others to help balance the tree.
  - Ensure that the characteristics of a binary search tree are not violated.

Example: Change the Color

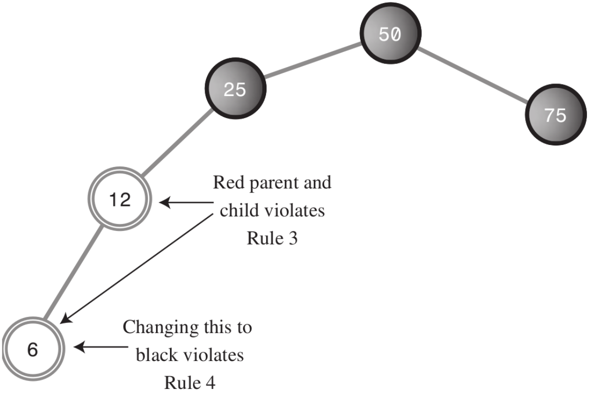


To insert a node 12, the colors need to be changed:



Example: Rotations are Required

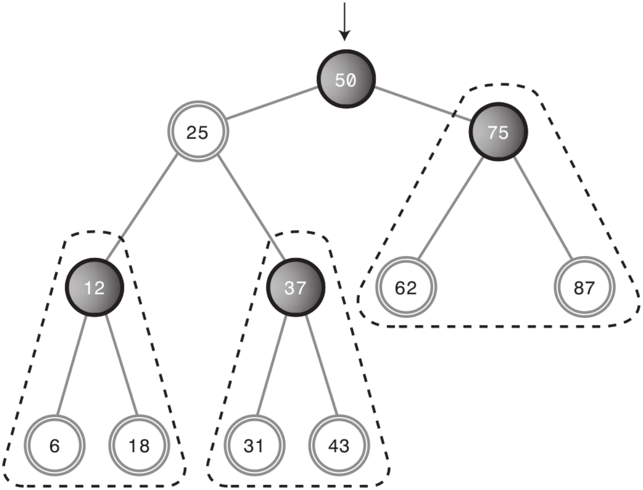
The node 6, cannot be inserted without rotations:



Solution: Rotate right such that 25 is the new root.

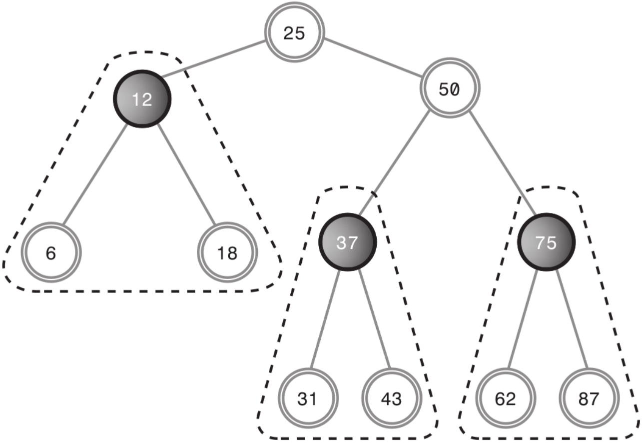
Rotations

- One node is chosen as the “top” of the rotation.
- If we’re doing a right rotation, this “top” node will move down and to the right, into the position of its right child.



Rotations

- One node is chosen as the "top" of the rotation.
- If we're doing a right rotation, this "top" node will move down and to the right, into the position of its right child.



- **Inorder** traversal yields the **same result!**

Inserting a Node

```
public class Node<E extends Comparable<E>>
             implements Comparable<Node<E>> {
    boolean black;
    ...
}
```

On the way down to the insertion point:

- If the current node is black and its two children are both red then:
  1. Color the children black. Color the current red, unless it is the root.
  2. Check that there are no violations of Rule 3 (Children of red must be black).
  3. If so, perform the appropriate rotations. (At most 2 are needed.)
- When you reach a leaf node, insert the new node with color red.
- Check again for red-red conflicts, and perform any necessary rotations.

Alexander.Baumgartner@risc.jku.at

Advantages

- The space complexity is $O(n)$, where n is the size of the input.
- The tree-height is bound by $O(\log n)$.
- Searching is done in $O(\log n)$ for the worst case time.
- Insertion is done in $O(\log n)$ for the worst case time.
- Deletion is done in $O(\log n)$ for the worst case time.

Deleting a Node

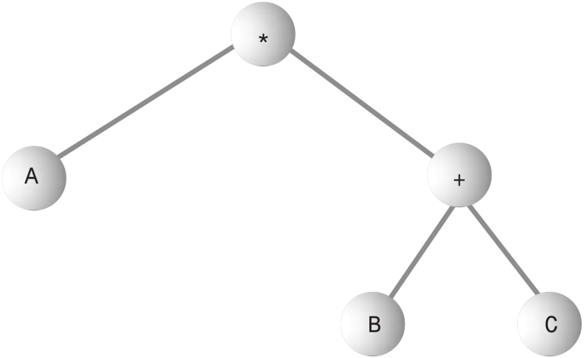
It can be done in $O(\log n)$ time, but:

- If deletion is not performed frequently, then a "deleted flag" can be used to increase the performance.
- Store a boolean value for each node and **mark it as deleted** instead of recoloring nodes and rotating the tree.

Alexander.Baumgartner@risc.jku.at

Other Examples for Trees

- Of course, not every tree is a search tree.
- Trees can be used for data compression (Huffman-Tree).
- Trees can be used to represent algebraic expression.




```
Infix: A*(B+C)
Prefix: *A+BC
Postfix: ABC+*
```

Exercise

- Implement the delete method for the binary tree.
- Use the method getSuccessor to find the successor like discussed during the lecture.

```java
public class Tree<E extends Comparable<E>> {
    ...
    /**
    * Returns true in case of success and false
    * if the given node was not found.
    */
    public boolean delete(E node) {
        // TODO: Implement this algorithm.
    }
    private Node<E> getSuccessor(Node<E> delNode) {
    ...
}
```

See the guidance for this exercise on the Moodle page.