



# Comp 282 Review

Why do we use Binary Search trees?

- Combines the advantage of two other structures on ordered array and linked list.

Slow insertion in an Ordered Array

- Search the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the bottom half. → Object in  $O(\log N)$

```
23 public void insert(int id, double data) {  
24     Node newNode = new Node(id, data);  
25     insert(newNode);  
26 }  
27  
28 //Todo: Implement the Node insertion  
29 public void insert(Node newNode) {  
30     if (root == null) {  
31         root = newNode;  
32         return;  
33     }  
34  
35     Node current = root;  
36     while (current != null) {  
37         if (newNode.id == current.id) {  
38             //No duplicates allowed  
39             throw new ArrayIndexOutOfBoundsException();  
40         }  
41  
42         if (newNode.id < current.id) {  
43             if (current.leftChild == null) {  
44                 current.leftChild = newNode;  
45                 break;  
46             } else {  
47                 current = current.leftChild;  
48             }  
49         } else {  
50             if (current.rightChild == null) {  
51                 current.rightChild = newNode;  
52                 break;  
53             } else {  
54                 current = current.rightChild;  
55             }  
56         }  
57     }  
58 }
```

→ inserting the node  
recursively inserting New node

## Path

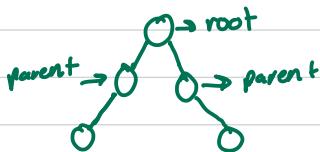
- Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a path.

## Root

- The node at the top of the tree is called a root.
- There is only one root in a tree.

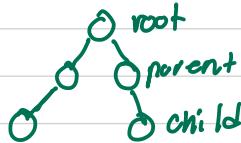
## Parent

- Any node (except the root) has exactly one edge running upward to another node.
- The node above is called the parent



## Child

- Any node may have one or more lines running downward to other nodes.
- These nodes below a given node are called its children.



## leaf

A node that has no children is called a leaf node

## subtree

- Any node can be considered to be the root of a subtree, which consists of its children and its children's children, and so on. Contains all its descendants

## Visiting

- A node visited when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or displaying it.

## Traversing

To traverse a tree means to visit all the nodes in some specific order  
For example you might visit all the nodes in order of ascending key values.

## levels

The levels of a particular node refer to how many generations the node is from the root.

- root = level 0 then its children will be level

1 its grandchildren will be level 2 and so on.

## Keys

- key values are used to search for the item or perform other operation

## Binary trees

- tree can have at most two children, the tree is called Binary tree
- The two children are called left child and Right child

## Finding a node

### Java code to find a node

```
7  public Node find(int key) {  
8      Node current = root;  
9  
10     while (current != null) {  
11         if (key == current.id) {  
12             return current;  
13         } else if (key < current.id) {  
14             current = current.leftChild;  
15         } else {  
16             current = current.rightChild;  
17         }  
18     }  
19  
20     return null;  
21 }  
22  
23 → set the current to the root  
→ while loop as long as the current != null or empty  
→ key must > the current num we will return the key  
→ If < the current we will get current to left child  
→ else current will be right child  
→ if the key not found return null
```

## Inserting a node

```
23  public void insert(int id, double data) {  
24      Node newNode = new Node(id, data);  
25      insert(newNode);  
26  }  
27  
28 → method created to add a new  
node.
```

```
28 //Todo: Implement the Node insertion  
29 public void insert(Node newNode) {  
30     if (root == null) {  
31         root = newNode;  
32         return;  
33     }  
34     → root is the new node  
35     Node current = root; → set current to new node  
36     while (current != null) {  
37         if (newNode.id < current.id) {  
38             → while current does not equal null  
39             //No duplicates allowed → will check if the new node equal the current if so no duplicates  
40             throw new ArrayIndexOutOfBoundsException(); → throw an exception  
41         }  
42  
43         if (newNode.id < current.id) {  
44             if (current.leftChild == null) {  
45                 current.leftChild = newNode; → current left child = new node  
46                 break; → End if statement  
47             } else {  
48                 current = current.leftChild; → Else if current left child is not empty we set it to current node  
49             }  
50         } else {  
51             if (current.rightChild == null) {  
52                 current.rightChild = newNode; → current right child = new node  
53                 break; → Break if else  
54             } else {  
55                 current = current.rightChild; → current = current right child.  
56             }  
57         }  
58     }  
59 }
```

## inorder traversal

- all nodes will be visited in ascending order based on their key values

## Pre-order and post-order traversal

```
private void inOrder(Node root) {
    if (root == null) return;
    inOrder(root.left);
    System.out.print(" " + root.value);
    inOrder(root.right);
}

private void preOrder(Node root) {
    if (root == null) return;
    System.out.print(" " + root.value);
    preOrder(root.left);
    preOrder(root.right);
}

private void postOrder(Node root) {
    if (root == null) return;
    postOrder(root.left);
    postOrder(root.right);
    System.out.print(" " + root.value);
}
```

// traverse methods

## Finding max and min values

min values go to left child of root; then go left child of that child until you come to the

child that has no left child this node is the min

```
public Node getMinimum() {
    if (root == null)  $\Rightarrow$  if root equals null return
        return null;
    ...
    Node current = root;  $\Rightarrow$  current equals root
    while (current.leftChild != null)  $\Rightarrow$  while current.leftChild != null
        current = current.leftChild;  $\Rightarrow$  current = current.leftChild
    ...
    return current;  $\Rightarrow$  return current
}
```

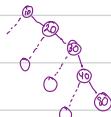
```
public Node getMaximum() {
    if (root == null)  $\Rightarrow$  root equals null will return null
        return null;
    ...
    Node current = root;  $\Rightarrow$  node current = root
    while (current.rightChild != null)  $\Rightarrow$  current = root
        current = current.rightChild;  $\Rightarrow$  current = current.rightChild;
    ...
    return current;  $\Rightarrow$  return current
}
```

## Deleting a node

```
private void deleteNode(Node root, int key) {
    if (root == null) return;
    if (key < root.value)  $\Rightarrow$  current equals root
        deleteNode(root.left, key);
    else if (key > root.value)  $\Rightarrow$  current equals root
        deleteNode(root.right, key);
    else if (root.leftChild == null & root.rightChild == null)  $\Rightarrow$  current.equals(root)
        return;
    else if (root.leftChild != null & root.rightChild == null)  $\Rightarrow$  current.equals(root)
        current = root.leftChild;
    else if (root.leftChild == null & root.rightChild != null)  $\Rightarrow$  current.equals(root)
        current = root.rightChild;
    else {  $\Rightarrow$  current.equals(root)
        if (root.leftChild == null)  $\Rightarrow$  current.leftChild = null
            current.parent.leftChild = null;
        else if (root.leftChild != null & root.rightChild != null)  $\Rightarrow$  current.equals(root)
            current = findMin(root.right);
        else if (root.leftChild != null & root.rightChild == null)  $\Rightarrow$  current.equals(root)
            current = root.leftChild;
        current.parent.leftChild = current;
        current.parent = current.parent.parent;
    }
}
```

## chapter 9) Red - Black Trees

## Balanced and Unbalanced Trees



- During insertion balance is achieved

### Rest - Blank Tree characteristic

- Q The nodes are colored

- During insertion and deletion rules are followed that

preserves various arrangement of those solar

### Selected Studies

- TA: A red-block - every node is either black or red

~~Red - Black Rules~~

- Every node is either red or Black

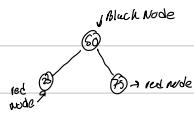
- The next is always black.

- z. B. werden die ersten 10 Minuten durch die Block*

- Even with five other kids to care for, we could just capture the ones

NUMBER OF BLACKS IN U.S.

### *Tension, two and three*



## Rotation

```

private void rotateLeft(RedBlackNode top, RedBlackNode parent) {
    if (top.rightChild == null) //right child
        return;
    RedBlackNode nextTop = top.rightChild; //new black node
    top.rightChild = nextTop.leftChild; //new red node
    nextTop.leftChild = top; //parent
    if (parent == null)
        root = nextTop;
    else if (parent.leftChild == top)
        parent.leftChild = nextTop;
    else
        parent.rightChild = nextTop;
}

private void rotateRight(RedBlackNode top, RedBlackNode parent) {
    if (top.leftChild == null) //left child
        return;
    RedBlackNode nextTop = top.leftChild; //new black node
    top.leftChild = nextTop.rightChild; //new red node
    if (top == root)
        parent = null;
    else if (parent.leftChild == top)
        parent.leftChild = nextTop;
    else
        parent.rightChild = nextTop;
}

```

## Color flips

```

private void flipColors(RedBlackNode current) {
    if (current.leftChild == null || current.rightChild == null)
        return;
    if (current.leftChild.color == current.rightChild.color) {
        current.leftChild.color = !color;
        current.rightChild.color = !color;
        current.parent.color = !color;
        if (current.parent == null)
            root.color = !color;
        else
            current.parent.color = true;
    }
}

```

## Poletion

```

import java.util.*;

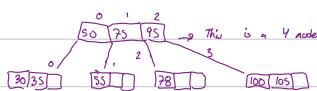
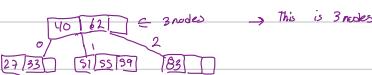
public class RedBlackTree {
    private RedBlackNode root;
    private RedBlackNode current;
    private RedBlackNode previous;
    private RedBlackNode parent;
    private RedBlackNode nextTop;
    private RedBlackNode next;
    private RedBlackNode leftChild;
    private RedBlackNode rightChild;
    private boolean color;
    private int height;
    private int maxDepth;

    public RedBlackTree() {
        root = null;
        current = null;
        previous = null;
        parent = null;
        nextTop = null;
        next = null;
        leftChild = null;
        rightChild = null;
        color = true;
        height = 0;
        maxDepth = 0;
    }
}

```

## Chapter 10) 2-3-4 Trees and External Storage

- A node with one data item always has two children
- A node with two data items always has three children.
- A node with three data items always has four children



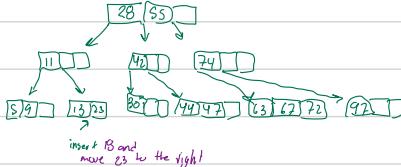
### Searching 2-3-4 trees

Same as binary search you go thru every child consequently

### Insertion

```
public void insert(int key) {
    if (isFull()) {
        System.out.println("Tree is full");
        return;
    }
    if (isRootEmpty())
        root = new Node(key);
    else
        insertInRoot(key);
}

private void insertInRoot(int key) {
    if (root.isFull()) {
        System.out.println("Root is full");
        Node parent = new Node();
        parent.setLeftChild(createNewNode());
        parent.setRightChild(createNewNode());
        parent.setLeftChild(root);
        parent.setRightChild(new Node(key));
        root = parent;
    } else if (key < root.data)
        root.setLeftChild(insertInLeftSubtree(key));
    else if (key > root.data)
        root.setRightChild(insertInRightSubtree(key));
}
```



## Node splits



### CHAPTER 10 2-3-4 Trees and External Storage

- Data item B is moved into the parent of the node being split.
- Data item A remains where it is.
- The rightmost two children are disconnected from the node being split and connected to the new node.

An example of a node split is shown in Figure 10.5. Another way of describing a node split is to say that a 4-node has been transformed into two 2-nodes.

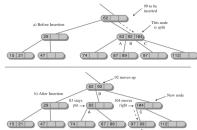


FIGURE 10.5 Splitting a node.

Note that the effect of the node split is to move data up and to the right. It is this rearrangement that keeps the tree balanced.

Here the insertion required only one node split, but more than one full node may be encountered on the path to the insertion point. When this is the case, there will be multiple splits.



## Splitting the Root



- Data item B is moved into the new sibling.
- Data item A is inserted where it is.
- The two rightmost children of the node being split are disconnected from it and connected to the new right child node.

Figure 10.6 shows the root being split. This process creates a new root that is at a higher level than the old one. Thus, the overall height of the tree is increased by one. Note that in this diagram the root node is split into two nodes.

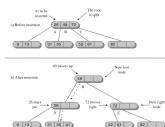


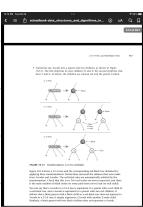
FIGURE 10.6 Splitting the root.

Remember that when the root node splits, the insertion path continues down the tree. In Figure 10.6, the data item with a key of 41 is inserted into the appropriate leaf.

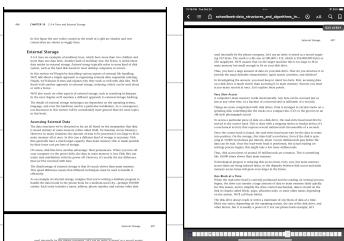
### Splitting on the Way Down



## Transforming from 2-3-4 to Red-Black



## External storage



```
private boolean isBalancedByNodeCount(){  
    int leftHand;  
    int rightHand;
```

```
    if(root == null){  
        return true;  
    }  
    leftHand = height(leftChild);  
    rightHand = height(rightChild);
```

```
    if(Math.abs(leftHand - rightHand) <= 1  
    && leftChild.isBalancedByNodeCount() &&  
    rightChild.isBalancedByNodeCount()){  
        return true;  
    }  
    return false;  
}
```

```
int height(Node node){
```

```
    if(node == null){  
        return 0;  
    }  
}
```

```
    if(height(leftChild) > height(rightChild)){  
        return 1 + height(leftChild);  
    }  
    else{  
        return 1 + heiaght(rightChild);  
    }  
}
```

```
12:21 PM Tue Oct 24 *** 92%  
BinarySearchTreeCheckBalanced.java  
Switch To Dark Mode  


```
public class BinarySearchTreeCheckBalanced {  
    private Node root;  
  
    ****  
    Implement this method and replace the return statement below with your code.  
    * Definition of Balance tree : On page 372 of book:  
    * An internal node is balanced if most of the nodes are on one side of the root or the other.  
    ****  
    private boolean isBalancedByNodeCount() {  
  
        return false;  
    }  
  
    public static void main(String args[]) {  
        int[] values1 = {50, 25, 12, 23, 75, 65, 85};  
        int[] values2 = {50, 25, 12, 53, 75, 65, 85};  
  
        BinarySearchTree tree1 = new BinarySearchTree();  
        tree1.buildTreeFromArray(values1);  
        if (tree1.isBalancedByNodeCount())  
            System.out.println("Tree No. 1 is Balanced.");  
        else  
            System.out.println("Tree No. 1 is NOT Balanced.");  
  
        BinarySearchTree tree2 = new BinarySearchTree();  
        tree2.buildTreeFromArray(values2);  
        if (tree2.isBalancedByNodeCount())  
            System.out.println("Tree No. 2 is Balanced.");  
        else  
            System.out.println("Tree No. 2 is NOT Balanced.");  
    }  
  
    private void buildTreeFromArray(int[] values) {  
        for (int i = 0; i < values.length; i++) {  
            double d = values[i] * 2.5;  
            insert(values[i], d);  
        }  
    }  
  
    ****  
    Code below this is copied from our implementation in class ****  
    ****  
    private void insert(int key, double data) {  
        Node n = new Node(key, data);  
  
        if (root == null)  
            root = n;  
        else {  
            Node current = root;  
            Node parent;  
  
            while (true) {  
                parent = current;  
                if (key < current.key) {  
                    current = current.leftChild;  
                    if (current == null) {  
                        parent.leftChild = n;  
                        return;  
                    }  
                } else {  
                    current = current.rightChild;  
                    if (current == null) {  
                        parent.rightChild = n;  
                        return;  
                    }  
                }  
            }  
        }  
    }  
}
```


```

```
12:21 PM Tue Oct 24 *** 92%  
BinarySearchTreeCheckBalanced.java  
Switch To Dark Mode  


```
int[] values2 = {50, 25, 12, 53, 75, 65, 85};  
BinarySearchTree tree1 = new BinarySearchTree();  
tree1.buildTreeFromArray(values1);  
if (tree1.isBalancedByNodeCount())  
    System.out.println("Tree No. 1 is Balanced.");  
else  
    System.out.println("Tree No. 1 is NOT Balanced!");  
  
BinarySearchTree tree2 = new BinarySearchTree();  
tree2.buildTreeFromArray(values2);  
if (tree2.isBalancedByNodeCount())  
    System.out.println("Tree No. 2 is Balanced.");  
else  
    System.out.println("Tree No. 2 is NOT Balanced!");  
  
private void buildTreeFromArray(int[] values) {  
    for (int i = 0; i < values.length; i++) {  
        double d = values[i] * 2.5;  
        insert(values[i], d);  
    }  
}  
  
****  
Code below this is copied from our implementation in class ****  
****  
private void insert(int key, double data) {  
    Node n = new Node(key, data);  
  
    if (root == null)  
        root = n;  
    else {  
        Node current = root;  
        Node parent;  
  
        while (true) {  
            parent = current;  
            if (key < current.key) {  
                current = current.leftChild;  
                if (current == null) {  
                    parent.leftChild = n;  
                    return;  
                }  
            } else {  
                current = current.rightChild;  
                if (current == null) {  
                    parent.rightChild = n;  
                    return;  
                }  
            }  
        }  
    }  
}  
  
class Node {  
    int key;  
    double data;  
    Node leftChild;  
    Node rightChild;  
}  
Node(int key, double data) {  
    this.key = key;  
    this.data = data;  
}  
void displayNode() {  
    System.out.println("(" + key + ", " + data + ")");  
}
```


```