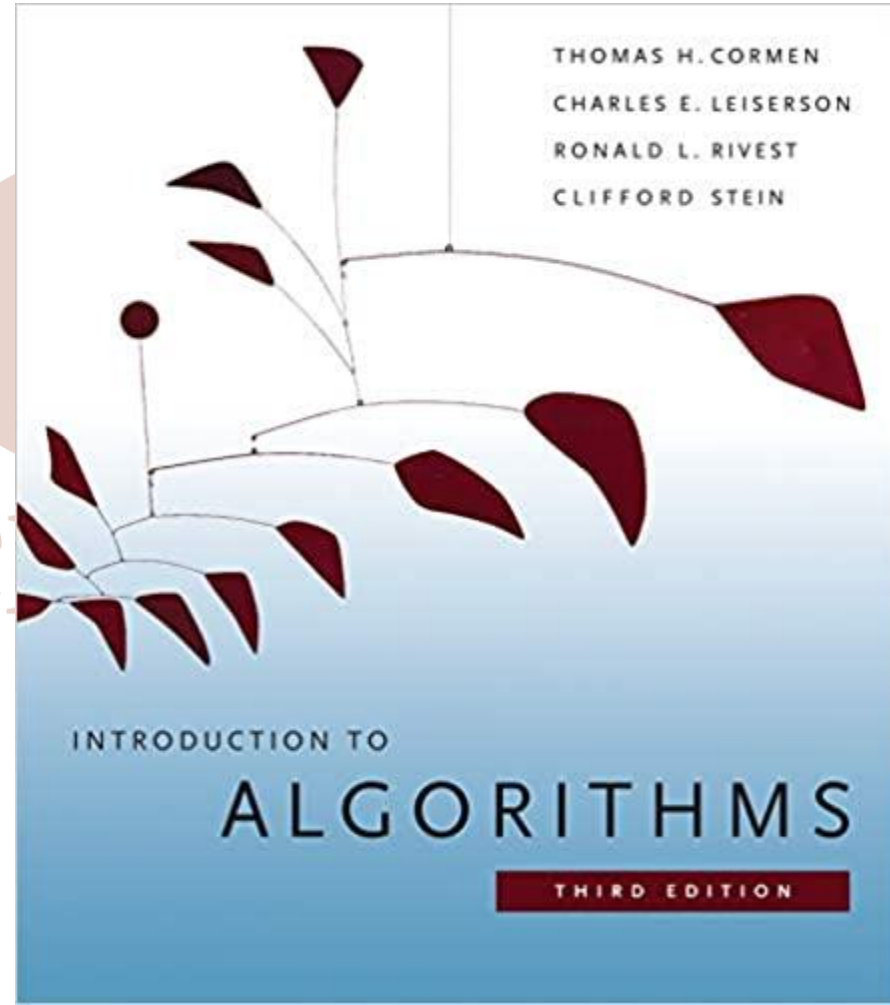


CS 07540 Advanced Design and Analysis of Algorithms

Week 5

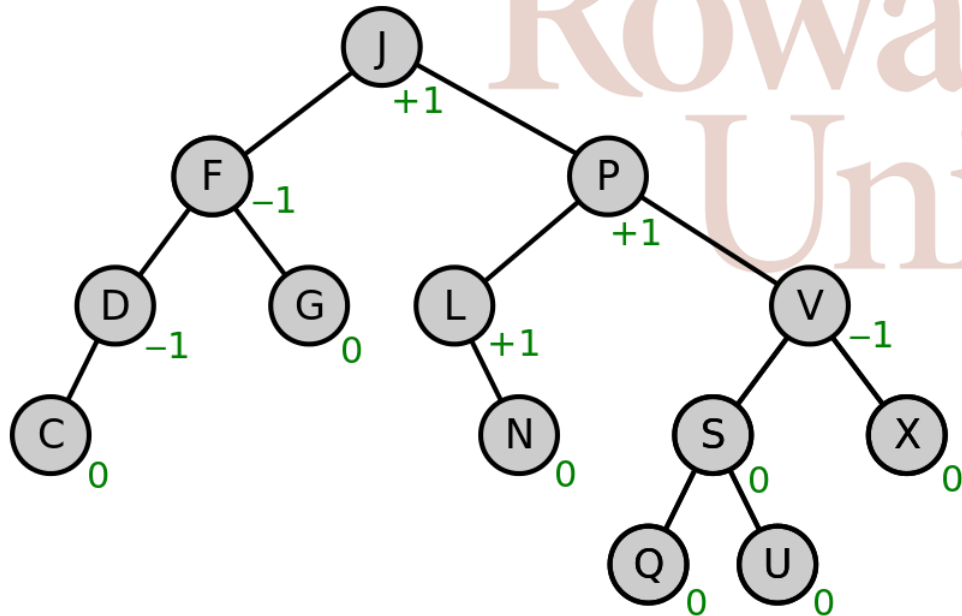
- AVL Trees
 - Terminology
 - Rebalancing
 - Rotations
- Scapegoat Trees
 - Terminology
 - Rebalancing
 - Partial rebuilding
 - Global rebuilding



AVL-Trees

An AVL tree (named after its two inventors Adel'son-Vel'skii and Landis) is a [self-balancing binary search tree](#) where the height of every node and that of its sibling differ by at most 1. AVL trees are ***height-balanced***.

Height can be calculated recursively; however, it is generally faster for each node to keep track of its subtree height during insertions and deletions. While this is also done recursively, only the subtrees with a direct path from a new node to root need to be involved.



```
public class AVLTree<Key extends Comparable<Key>, Value> {  
  
    public class Node<Key, Value> {  
        Key key;  
        Value value;  
        int height;  
        Node<Key, Value> left;  
        Node<Key, Value> right;  
  
        Node(Key key, Value value) {  
            this.key = key;  
            this.value = value;  
            this.height = 0;  
            this.left = null;  
            this.right = null;  
        }  
    }  
}
```

```
private Node<Key, Value> root;  
private final Comparator<Key> comparator;
```



```
public AVLTree() {
    root = null;
    comparator = new Comparator<Key>() {
        @Override
        public int compare(Key o1, Key o2) {
            return o1.compareTo(o2);
        }
    };
}

public AVLTree(Comparator<Key> keyComparator) {
    root = null;
    comparator = keyComparator;
}

public int size() { return size(this.root); }

private int size(Node<Key, Value> node) {
    return node == null ? 0 : 1 + size(node.left) + size(node.right);
}
```

Now we can instantiate an AVL tree as follows:

```
AVLTree<Integer, String> T = new AVLTree<>();
```

Before we begin to fill our AVL tree, let's recognize that we already have a method for searching, because it is just BST search (with generic keys).



Search

```
/**
 * <p>Non-recursive implementation of search for a BST (AVL Tree)</p>
 */
public Value search(Key key) throws FindException {
    Node<Key, Value> node = root;
    while (node != null) {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult == 0) {
            break;
        }
        node = compareResult < 0 ? node.left : node.right;
    }
    if (node != null)
        return node.value;
    else
        throw new FindException("Key not found.");
}
```

Insert

A node n of a tree T is called ***balanced*** if the absolute value of the difference between the heights of its children is at most 1. Otherwise, it is called unbalanced.

An insertion in an AVL tree T begins as an **INSERT** operation for a standard binary search tree. This insertion may violate the height-balance property since height of some subtrees may have increased by 1. Those subtrees are exactly the trees that are on the path from the new node to root.

```
public void insert(Key key, Value value) {
    root = insert(root, key, value);
}

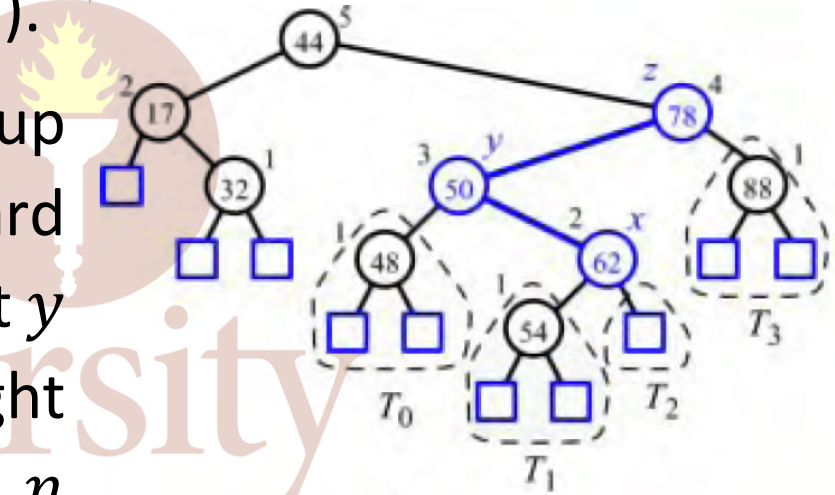
private Node<Key, Value> insert(Node<Key, Value> node, Key key, Value value) {
    if (node == null) {
        return new Node<Key, Value>(key, value);
    } else {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult < 0) {
            node.left = insert(node.left, key, value);
        } else if (compareResult > 0) {
            node.right = insert(node.right, key, value);
        } else {
            // duplicate key
            // may throw runtime exception or quietly overwrite value
            node.value = value;
        }
    }
}

return rebalance(node);
}
```

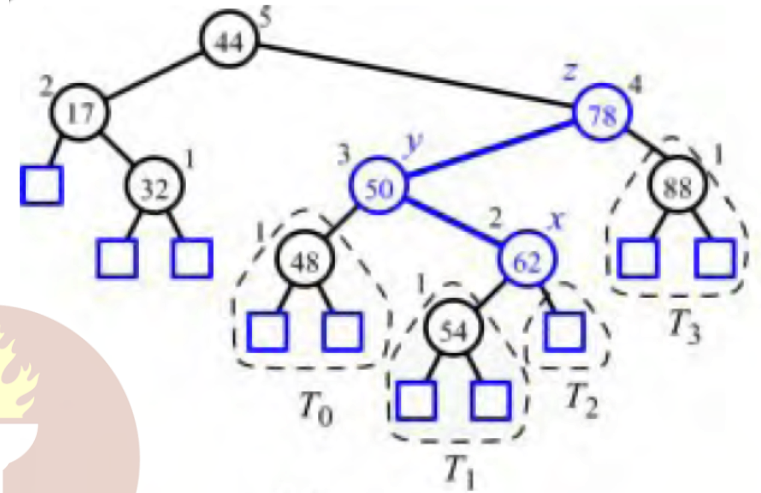

Rebalancing an AVL Tree

We restore the balance of the nodes in the BST T using a search-and-repair strategy. Numbers next to nodes in subsequent pictures are the subtree heights + 1 (counting null children).

Let z be the first node encountered going up from newly inserted node n (key 54) toward the root of T such that z is unbalanced. Let y denote the child of z with higher height (note that y must be an ancestor of n because n added height to an imbalance of **2**). Let x be the child of y with higher height (there cannot be a tie and x must be an ancestor of n , possibly itself, again because of path length from n to z of at least **2**).



The repair is done with a tree rotation, like the Red-Black tree rotations we have already looked at. We need an auxiliary function that checks for imbalance:

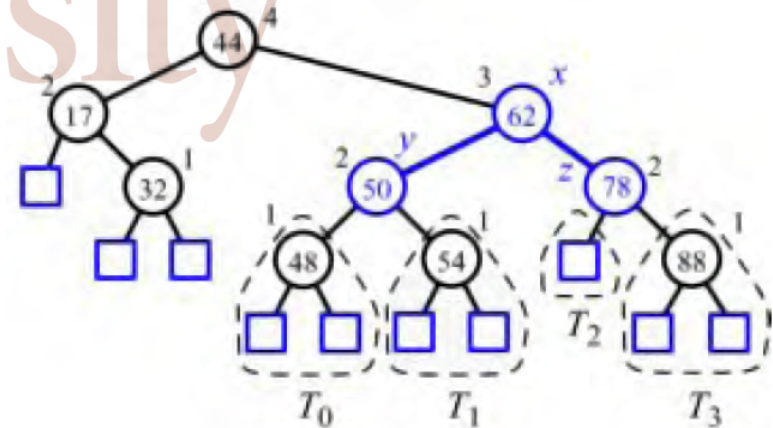
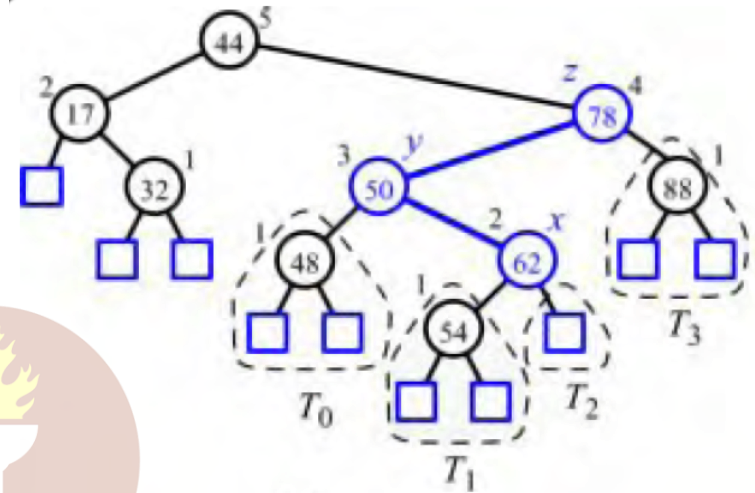


```
Balance (node z)
    return height(z.right) - height(z.left)
```

```

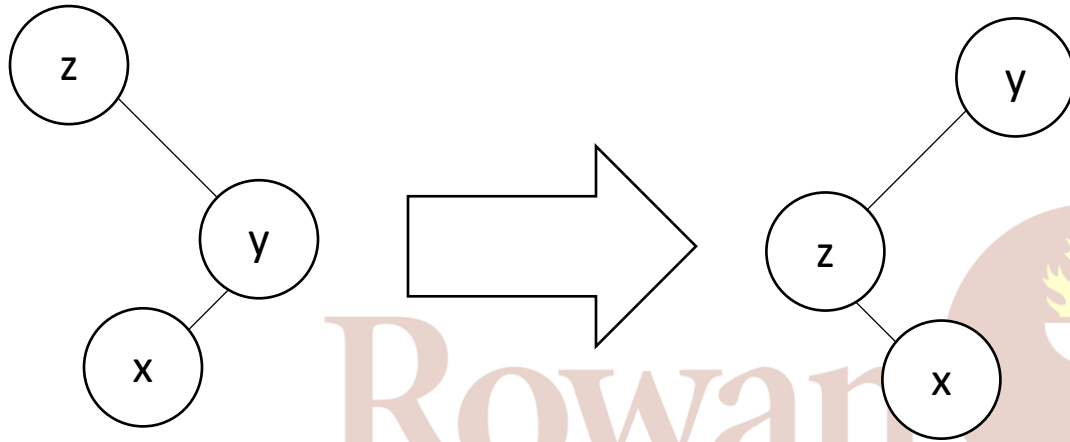
Rebalance(node z)
  if Balance(z) == 2
    y = z.right
    if Balance(y) == 2
      x = y.right
      z = rotateLeft(z)
    else
      x = y.left
      y = rotateRight(y)
      z = rotateLeft(z)
  else
    y = z.left
    if Balance(y) == -2
      x = y.left
      z = rotateRight(z)
    else
      x = y.right
      y = rotateLeft(y)
      z = rotateRight(z)

```



```
private Node<Key, Value> rebalance(Node<Key, Value> z) {
    updateHeight(z);
    int balance = getBalance(z);
    if (balance > 1) {
        if (height(z.right.right) > height(z.right.left)) {
            z = rotateLeft(z);
        } else {
            z.right = rotateRight(z.right);
            z = rotateLeft(z);
        }
    } else if (balance < -1) {
        if (height(z.left.left) > height(z.left.right)) {
            z = rotateRight(z);
        } else {
            z.left = rotateLeft(z.left);
            z = rotateRight(z);
        }
    }
    return z;
}
```

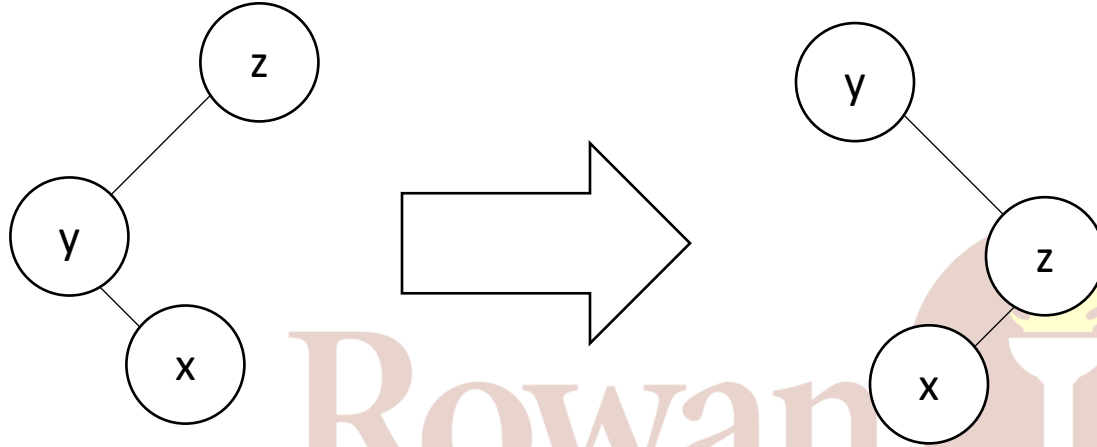
RotateLeft



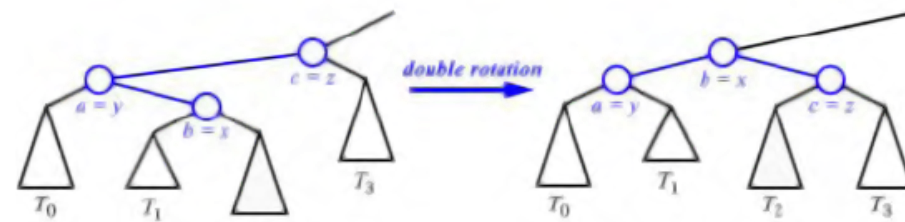
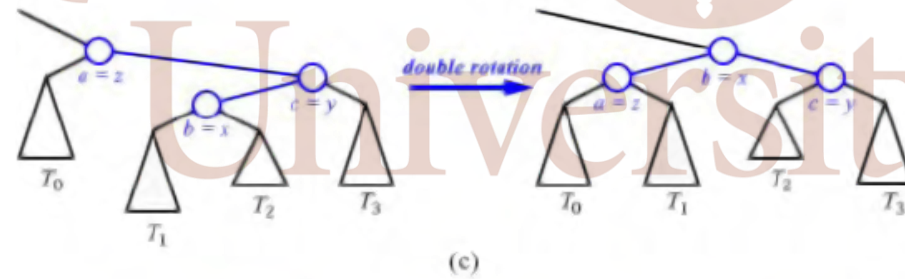
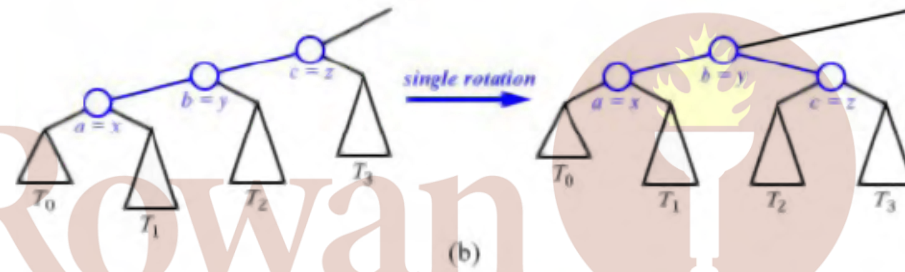
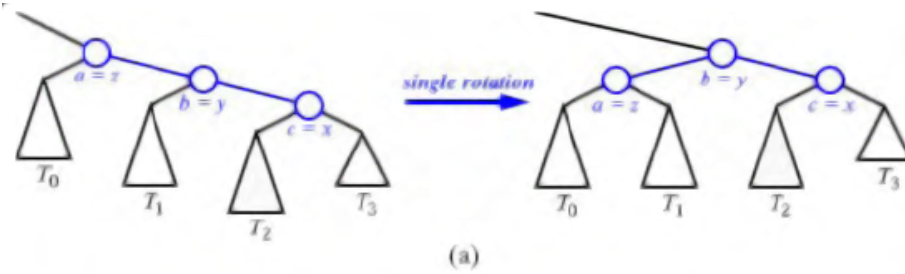
Rowan
University



RotateRight



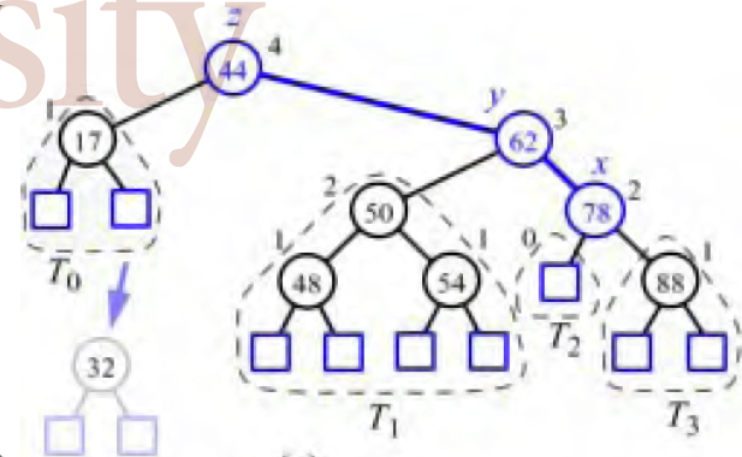
Rowan
University



Delete

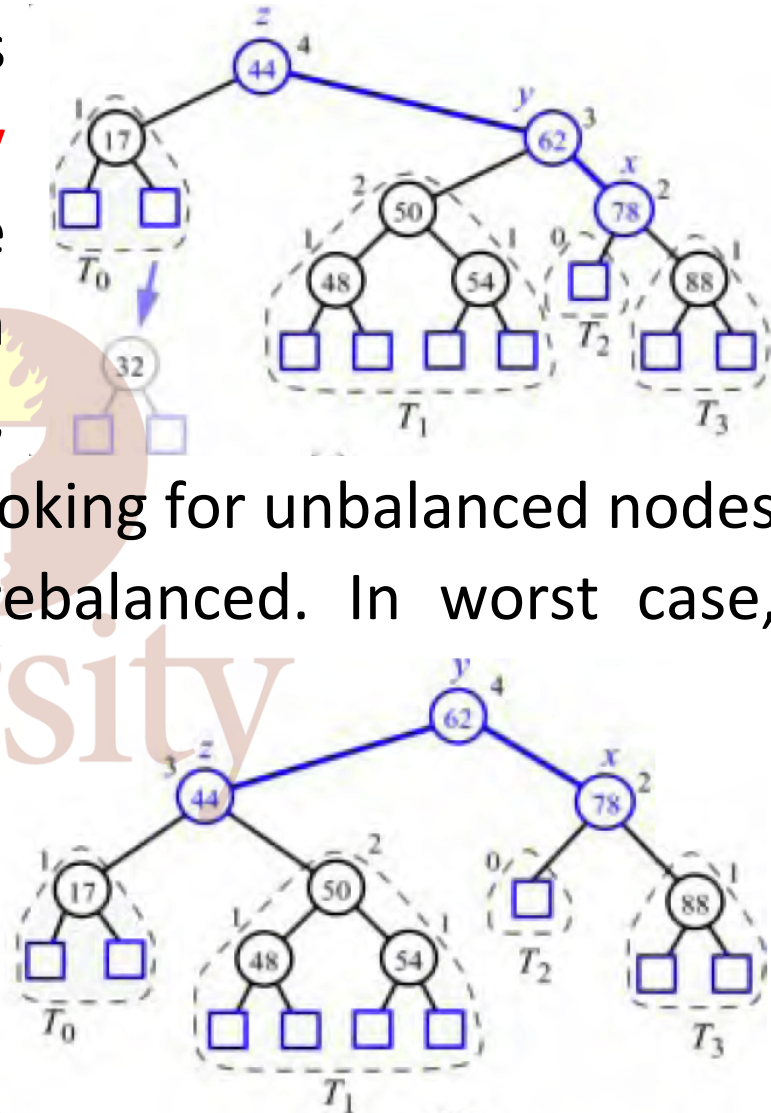
Removal of an element in an AVL tree also starts with the standard BST **DELETE**. Again, it may violate the height-balance property. After removing an internal node and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent node of the previously removed node to the root of T . However, there can be at most one such unbalanced node.

Rebalancing



Perform a **Rebalance(x)** operation. This restores the height-balance property **locally** at z . Rebalancing locally may reduce the height of the subtree, which may cause an ancestor to become unbalanced. Hence, after rebalancing z , continue to walk up T looking for unbalanced nodes until there are none or root has been rebalanced. In worst case, $O(\log(n))$ rebalancing operations are needed to restore balance **globally**. Per node a rotation takes $O(1)$ time.

```
public void delete(Key key) {
    root = delete(root, key);
}
```



```
private Node<Key, Value> delete(Node<Key, Value> node, Key key) {
    if (node == null) {
        return node;
    } else {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult < 0) {
            node.left = delete(node.left, key);
        } else if (compareResult > 0) {
            node.right = delete(node.right, key);
        } else {
            if (node.left == null || node.right == null) {
                node = (node.left == null) ? node.right : node.left;
            } else {
                Node<Key, Value> leftMostChild = leftMostChild(node.right);
                node.key = leftMostChild.key;
                node.right = delete(node.right, node.key);
            }
        }
    }
    if (node != null) {
        node = rebalance(node);
    }
    return node;
}
```

Properties of AVL Trees

- Find/Search/Get in an AVL tree is $O(\log(n))$.
- For every internal node of an AVL tree, the heights of the two children subtree differ by at most 1.
- A subtree of an AVL tree is itself an AVL tree.
- The height of an AVL tree storing n entries is $O(\log(n))$.
 - Can be proven with a recursion formula bottom up.

Comparison AVL vs Red-Black Trees

Both red-black trees and AVL trees are balanced BSTs. They support **INSERT**, **DELETE** and **SEARCH** in guaranteed $O(\log(n))$ time.

- AVL trees are more rigidly balanced. They provide faster **SEARCH**.
 - Sibling subtrees differ by $0, \pm 1$ vs path length from root to leaf longest-to-shortest is at most a factor of 2.
- Red-Black trees are better for **INSERT** and **DELETE** intensive tasks.
- AVL trees store the balance factor at each node. This takes $O(n)$ extra space on the tree. Red-Black trees need one extra bit: makes $O(n)$.

Red-Black trees are used in

- current Linux kernel for the [Completely Fair Scheduler](#)
- [HashMap](#) and derived structures in Java 8
- [set](#), [map](#) and derived structures in [C++ STL](#)

```
typedef _Rb_tree<key_type, value_type, _Identity<value_type>,  
               key_compare, _Key_alloc_type> _Rep_type;  
_Rep_type _M_t; // Red-black tree representing set.
```

AVL trees are mostly used for in-memory sorts of sets and dictionaries.

Scapegoat Trees

Recall that plain binary search trees work well if data is inserted in random order, but traversing a BST becomes much slower if data inserted is already ordered or inversely ordered. In that case a BST becomes unbalanced and the ability to quickly find, insert, or delete an item is lost ($O(\log(n))$ to $O(n)$ transition).

A scapegoat tree (SGT) is a self-balancing binary search tree which provides **worst-case** $O(\log(n))$ lookup time, and $O(\log(n))$ **amortized** insertion and deletion time. They were presented in 1993 by Galperin and Rivest. Scapegoat trees are lazily height balanced trees. We already looked at other balanced trees such as 2-3 trees, red-black trees and AVL trees.

Abstract

We present an algorithm for maintaining binary search trees. The amortized complexity per INSERT or DELETE is $O(\log n)$ while the *worst-case* cost of a SEARCH is $O(\log n)$.

Scapegoat trees, unlike most balanced-tree schemes, do not require keeping extra data (e.g. “colors” or “weights”) in the tree nodes. Each node in the tree contains only a key value and pointers to its two children. Associated with the root of the whole tree are the only two extra values needed by the scapegoat scheme: the number of nodes in the whole tree, and the maximum number of nodes in the tree since the tree was last completely rebuilt.

In a scapegoat tree a typical rebalancing operation begins at a leaf, and successively examines higher ancestors until a node (the “scapegoat”) is found that is so unbalanced that the entire subtree rooted at the scapegoat can be rebuilt at zero cost, in an amortized sense. Hence the name.

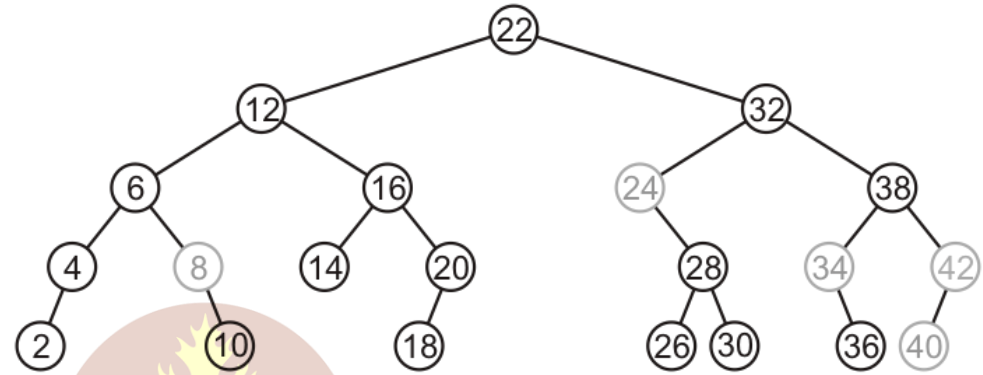
Balancing

Self-balancing BSTs are designed to provide worst-case $O(\log(n))$ lookup time. There are two main balanced BST schemes:

- Height-balanced scheme: the height of the whole tree is $O(\log(n))$, where n is the number of nodes in the tree.
 - Scapegoat trees, Red-black trees and AVL trees
 - Worst-case cost of every operation is $O(\log(n))$ time
- Weight-balanced scheme: the size of subtrees rooted at siblings for every node in the tree is approximately equal. Maintaining weight-balance also maintains height-balance.
 - Scapegoat trees, BB[α] trees
 - Worst-case cost of every operation is $O(\log(n))$ time

Rebalancing Methods

- Global Rebuilding
 - Lazy Deletions
- Partial Rebuilding
 - Insertion



In a lazy-deletion binary search tree nodes are just marked as deleted while remaining in the tree. Runtime would be $O(\log n)$ in worst case. Occasionally, a member method may be called to rebuild (and clean up) all deleted nodes at once. When half the nodes in the tree have been marked, a new perfectly balanced tree containing only the unmarked nodes is rebuilt. The latter takes $O(n)$ using an in-order walk to create an ordered list of nodes/keys from which we can extract median keys recursively. We will do an [amortized analysis](#) in another class meeting.

Partial rebuilding is regular BST insertion followed by rebuilding unbalanced subtrees depending on a parameter α . After inserting a node, we walk back up the tree (using parent pointers) and update the heights and sizes of the nodes on the search path. If there is a node with $\text{height}(\text{node}) > \alpha \cdot \log(\text{size}(\text{node}))$, the subtree rooted at this node is rebuilt into a perfectly balanced tree. This takes $O(\text{size}(\text{node}))$ time. Scapegoat trees use both techniques.

Rebalancing

Unlike most other self-balancing BSTs, scapegoat trees have no additional per-node memory overhead compared to a regular binary search tree: a node stores only a key (and a value) and two pointers to the child nodes (and possibly a parent pointer). This makes scapegoat trees easier to implement and reduces node overhead.

Instead of the incremental rebalancing, scapegoat trees rarely but expensively choose a "scapegoat" and completely rebuild the subtree rooted at the scapegoat into an as complete as possible binary tree. Scapegoat trees have $O(n)$ worst-case update performance.

Scapegoat Trees Advantages

- Scapegoat trees combines ideas of weight- and height-balanced schemes **without** storing either height or weight data in the nodes.
- Scapegoat trees are regular binary search trees
- Scapegoat trees achieve $O(\log(n))$ run-time complexity for all operations (amortized – see later in the semester)
 - **INSERT, DELETE, SEARCH**

Each time the tree becomes “too unbalanced” it is rebuilt it into a fully balanced binary search tree. How can a tree become unbalanced?

- Execution of operations **INSERT** and **DELETE**

We will need to create new insert, delete, and rebuild methods.

Scapegoat Tree Additional Attributes

Each node **node** in an SGT maintains the following attributes:

- **node.key** – the value of the key stored at the node
- **node.left** – pointer to the left child
- **node.right** – pointer to the right child of x

The tree T maintains the following attributes:


- **T.root** – pointer to the root
- **T.size** – the number of nodes (needs to be updated upon **insert** and **delete**)
- **T.maxSize** – the maximum value of **T.size** since the rebuilt

Definitions for Pseudocode

- **height (node)** is the height of the subtree rooted at **node** or the longest distance in terms of edges from **node** to a leaf. It is a method.
- **depth (node)** is the depth of **node** or the distance in terms of edges from the root to **node**. It is a method.
- **node.parent** is the parent node of **node**
- **node.sibling** is the child, other than **node**, of parent of **node**
- **size (node)** is the number of nodes of subtree rooted at **node**. It is a method.
- **T.height** is the height of T or the longest path in terms of edges from **T.root** to a leaf in T . It can be a method or an attribute.

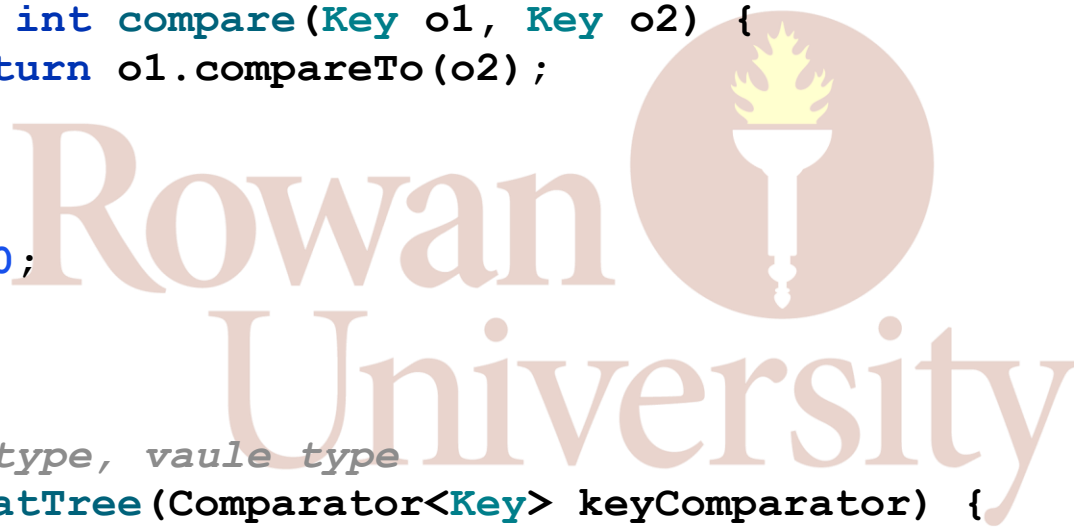
We want to allow for generic keys and values. Note that keys need to be **comparable** for the binary search tree property.

```
public class ScapegoatTree<Key extends Comparable<Key>, Value> {  
  
    static private class Node<Key, Value> {  
        Key key;  
        Value value;  
        Node<Key, Value> left, right, parent;  
  
        Node(Key key, Value value) {  
            this.key = key;  
            this.value = value;  
            this.parent = null;  
        }  
    }  
  
    private Node<Key, Value> root;  
    private Comparator<Key> comparator;  
    private int size;  
    private int maxSize;
```



```
// constructor
// @param key type, vaule type
public ScapegoatTree() {
    root = null;
    comparator = new Comparator<Key>() {
        @Override
        public int compare(Key o1, Key o2) {
            return o1.compareTo(o2);
        }
    };
    size = 0;
    maxSize = 0;
}
```

```
// constructor
// @param key type, vaule type
public ScapegoatTree(Comparator<Key> keyComparator) {
    root = null;
    comparator = keyComparator;
    size = 0;
    maxSize = 0;
}
```



Balancing Parameter

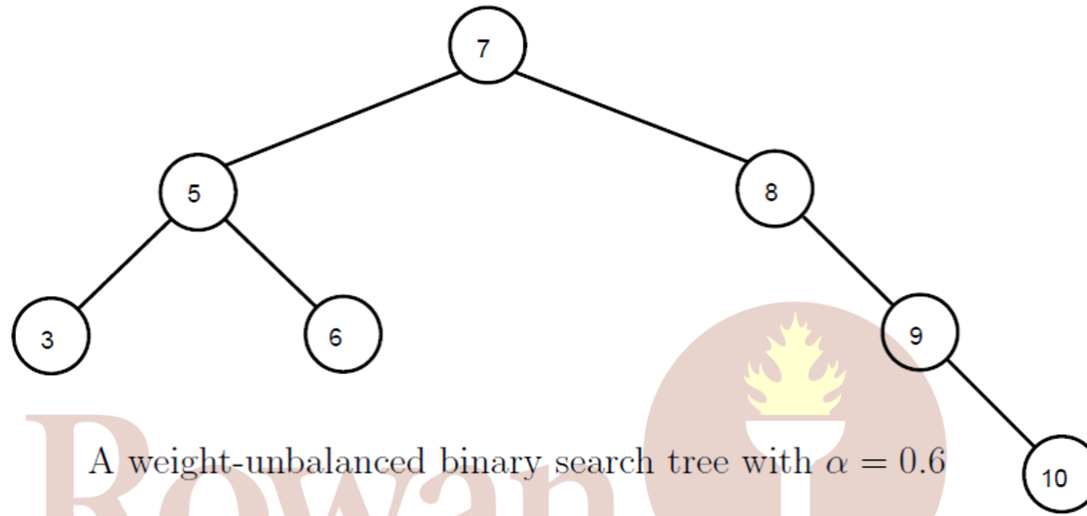
A binary-tree search (sub)tree rooted at **node** is **α -weight-balanced** for some α , $\frac{1}{2} \leq \alpha < 1$ if

- $\text{size}(\text{left}[\text{node}]) \leq \alpha \cdot \text{size}(\text{node})$, and
- $\text{size}(\text{right}[\text{node}]) \leq \alpha \cdot \text{size}(\text{node})$.

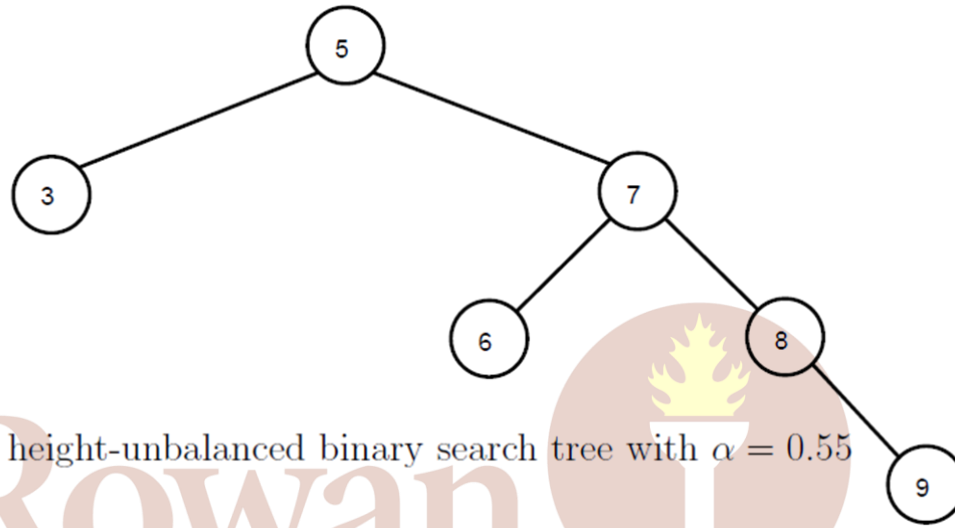
A tree is **α -weight-balanced** if for a given value α , $\frac{1}{2} \leq \alpha < 1$, all the nodes in the tree are α -weight-balanced. Intuitively, a tree is α -weight-balanced if for any subtree, the sizes of its left and right subtree are approximately equal.

$$x.h_{\alpha} = \left\lceil \log_{(1/\alpha)} x.size \right\rceil$$

$$T.h_{\alpha} = \left\lceil \log_{(1/\alpha)} T.size \right\rceil$$



Node 8 is not 0.6-weight-balanced because its right subtree has 2 nodes, and that is greater than $0.6 \cdot 3 = 1.8$ where 3 is the total number of nodes of the subtree rooted at 8. Hence the tree is not 0.6-weight-balanced binary search tree.



A height-unbalanced binary search tree with $\alpha = 0.55$

The tree is a 0.55-height-unbalanced BST because it has 6 nodes and its height is 3 while $\left\lfloor \log_{\frac{1}{0.55}}(6) \right\rfloor = 2$ (it is close: $\log_{\frac{1}{0.55}}(6) = 2.998$).

Scapegoat Nodes

A **node** in a BST T is a deep node for α such that $1/2 \leq \alpha < 1$ if

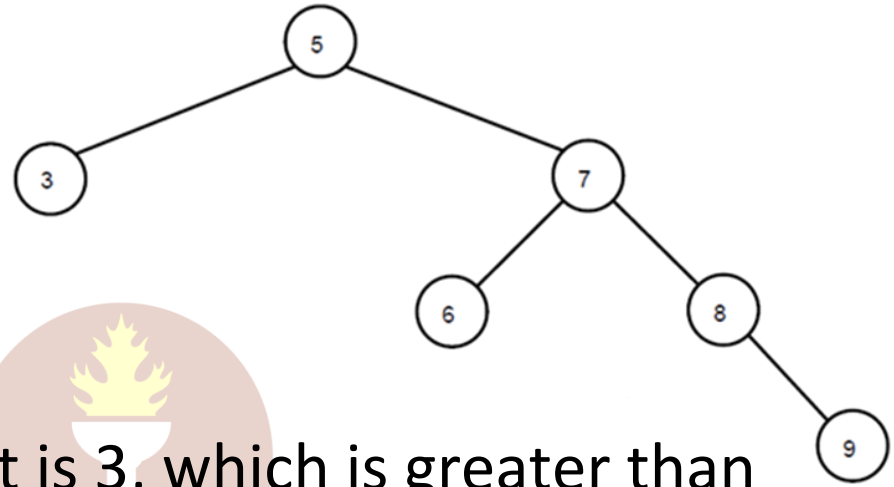
$$\mathbf{depth}(\mathbf{node}) > T.h_{\alpha}$$

A node **s** is called Scapegoat node of a newly inserted deep **node** if **s** is an ancestor of **node** and subtree rooted at **s** is not α -weight-balanced.

A BST is a loosely α -height-balanced if

$$T.\text{height} \leq (T.h_{\alpha} + 1)$$

Scapegoat Examples



Node 9 is a deep node because its height is 3, which is greater than

$$T.h_{\alpha} = \left\lfloor \log_{\frac{1}{0.55}}(6) \right\rfloor = 2$$

Root 5 would be chosen as Scapegoat node.

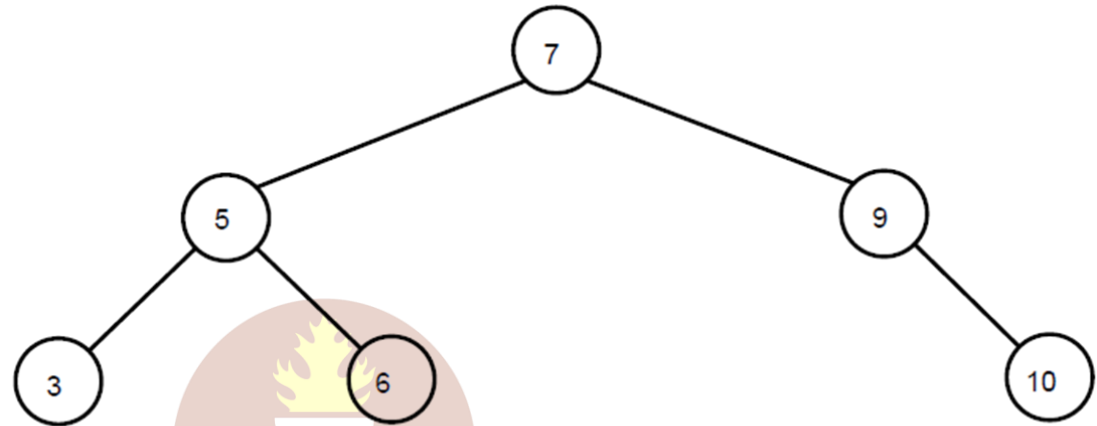
The tree T is an example of loosely 0.55-height-balanced tree because $T.\text{height} = 3$ and

$$3 \leq (T.h_{\alpha} + 1) = \left(\left\lfloor \log_{1/0.55}(6) \right\rfloor \right) + 1 = 3$$

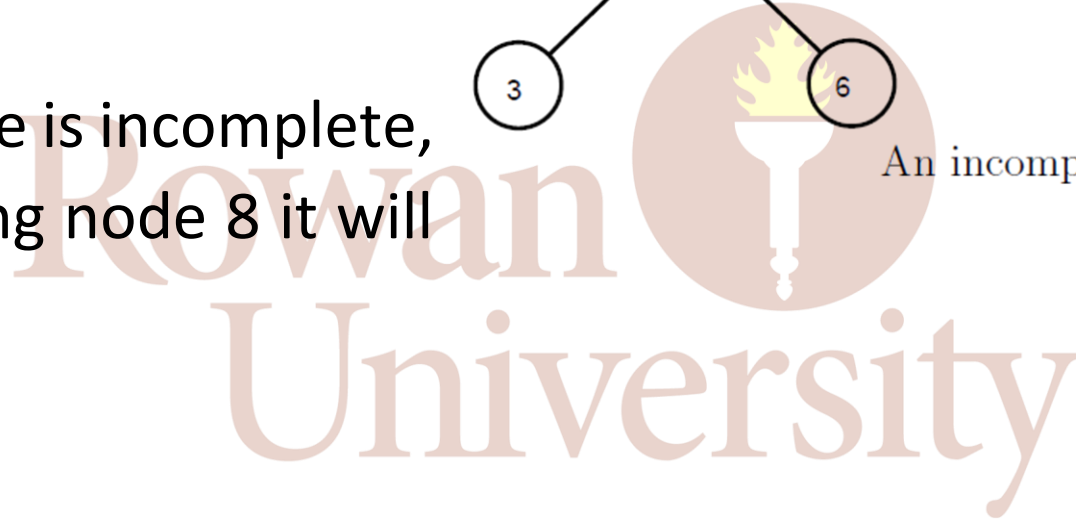
Incomplete vs Complete

A BST T is complete if inserting any node into T increases its height.

This example tree is incomplete, but after inserting node 8 it will be complete.



An incomplete binary search tree



Back to Scapegoat Tree Rebalancing

A Scapegoat tree is a regular BST in which the tree height is always loosely α -height-balanced. The balance is maintained by rebuilding the tree upon

- an **INSERT** operation of a deep node or
- a **DELETE** operation that leads to $T.size < \alpha \cdot T.maxSize$.

In the case of inserting a deep node, a Scapegoat node **s** will be detected. Both cases will result in rebuilding the subtree rooted at **s** into a 1/2-weight-balanced BST.

Scapegoat Tree Search

An SGT search is performed the same way as in BSTs.

Input: **root/node** of some subtree T to search for an integer key **k**

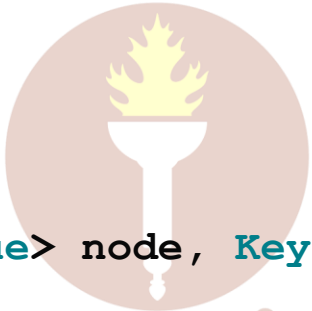
Output: **node** is a node in T such that **node.key = k** or **null** if there is no such n

```
if root = null or root.key = k then
    return root
else if  $k \leq \text{root.left.key}$  then
    return Search(root.left, k)
else
    return Search(root.right, k)
end if
```



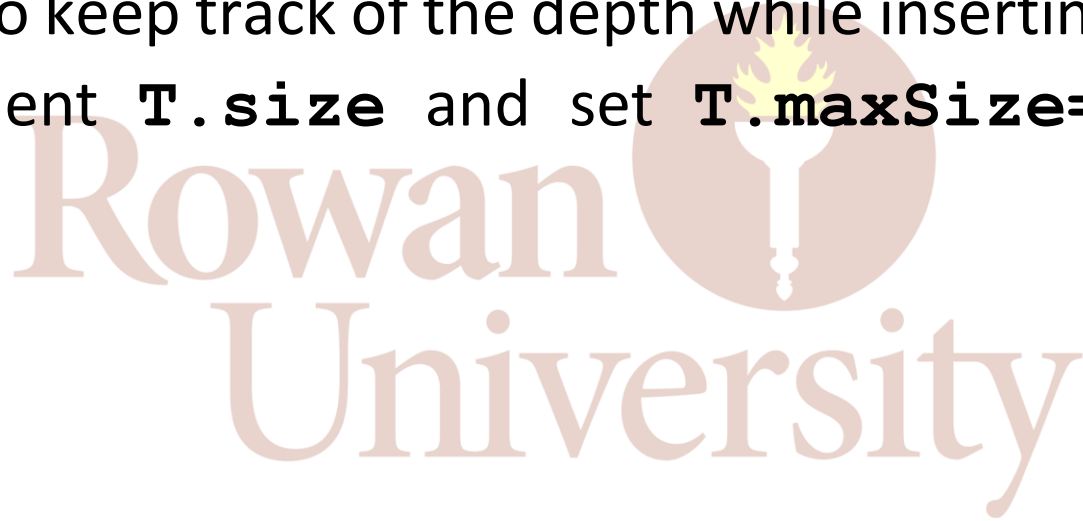
```
// public interface for get
// @param void
// @return node.value containing key
public Value get(Key key) {
    Node<Key, Value> node = get(this.root, key);
    if (node != null)
        return node.value;
    return null;
}

// recursive implementation for get
private Node<Key, Value> get(Node<Key, Value> node, Key key) {
    if (node == null)
        return null;
    int compareResult = comparator.compare(key, node.key);
    if (compareResult < 0)
        return get(node.left, key);
    else if (compareResult > 0)
        return get(node.right, key);
    else {
        return node;
    }
}
```



Scapegoat Tree Insert

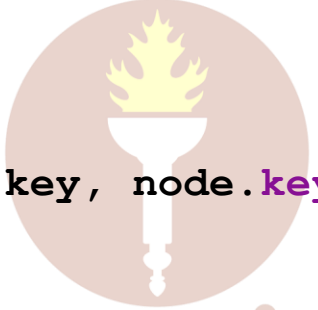
Insertion is done the same way as it is in a regular BST unless the inserted node is a deep node. Then we need to rebalance the tree. To detect deep nodes we need to keep track of the depth while inserting a node, and we need to increment **T.size** and set **T.maxSize=max(T.size, T.maxSize)**.



Recall BST Insert

```
public void insert(Key key, Value value){
    root = insert(root, key, value);
}

private Node<Key,Value> insert(Node<Key,Value> node, Key key, Value value){
    if(node == null) {
        node = new Node<>(key, value);
        return node;
    }
    int compareResult = comparator.compare(key, node.key);
    if(compareResult == 0){
        node.value = value;
        return node;
    }
    if(compareResult < 0) {
        node.left = insert(node.left, key, value);
        node.left.parent = node;
    }
    else {
        node.right = insert(node.right, key, value);
        node.right.parent = node;
    }
    return node;
}
```



We need to check for a newly inserted deep node and a possible scapegoat after the

```
if (node == null) {  
    node = new Node<>(key, value);  
    return node;  
}
```

statement.

Since we also need to keep track of parent pointers and left and right, it is better to implement a non-recursive traversal using a while loop.

```
// non-recursive insert
public void insert(Key key, Value value) {
    if (root == null) {
        root = new Node<>(key, value);
        size++;
    } else {
        Node<Key, Value> node = root;
        Node<Key, Value> parent = node.parent;
        boolean isLeftChild = false;
        while (true) {
            if (node == null) {
                Node<Key, Value> newNode = new Node<>(key, value);
                newNode.parent = parent;
                this.size++;
                this.maxSize = Math.max(size, maxSize);
                if (isLeftChild) {
                    newNode.parent.left = newNode;
                } else {
                    newNode.parent.right = newNode;
                }
            }
        }
    }
}
```

```
    if (isDeepNode(newNode)) {  
        // new node is in its place but tree needs to be rebalanced  
        // from an ancestor of node called scapegoat  
        Node<Key, Value> scapegoat = findScapegoat(newNode);  
        Node<Key, Value> scapegoatParent = scapegoat.parent;  
        boolean scapegoatOnParentsLeft = scapegoatParent != null &&  
scapegoatParent.left == scapegoat;  
        // connect root of balanced tree to parent  
        Node<Key, Value> balancedTreeRoot =  
buildTree(flattenTree(scapegoat), 0, size(scapegoat) - 1);  
        if (scapegoatParent != null) {  
            if (scapegoatOnParentsLeft) {  
                scapegoatParent.left = balancedTreeRoot;  
            } else {  
                scapegoatParent.right = balancedTreeRoot;  
            }  
        }  
        // if parent is null then scapegoat must be root  
        if (scapegoat == root) {  
            root = balancedTreeRoot;  
        }  
        maxSize = size;  
    }  
    return;  
}
```

```
int compareResult = comparator.compare(key, node.key);
if (compareResult == 0) {
    node.value = value;
    if (parent != null) {
        if (isLeftChild) {
            node.parent.left = node;
        } else {
            node.parent.right = node;
        }
    }
    return;
} else {
    parent = node;
    if (compareResult < 0) {
        node = node.left;
        isLeftChild = true;
    } else {
        node = node.right;
        isLeftChild = false;
    }
}
}
}
}
```

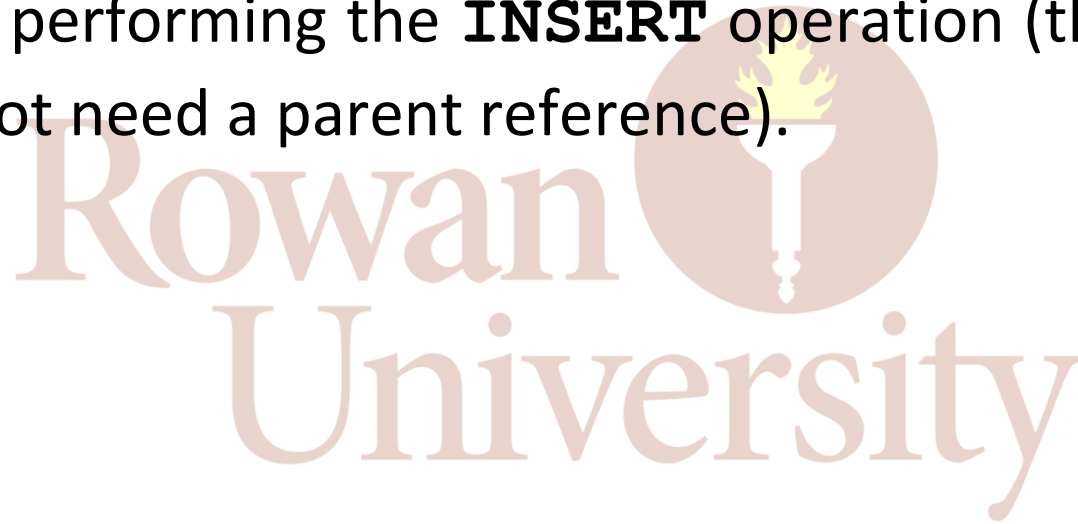
Input: The integer key **k**

Output: none

```
height = height(k)
if height = -1 then
    return
else if height > T.hα then
    scapegoat = FINDSCAPEGOAT (SEARCH (T.root,k) )
    REBUILDTREE (scapegoat.size() , scapegoat)
end if
return
```


Find a Scapegoat

Given a newly inserted deep node n_0 , climb up the tree until we find node n_i that is not α -weight-balanced. Store references to the parent on a stack while performing the **INSERT** operation (that way the node structure does not need a parent reference).



Input: n is a node and $n \neq \text{null}$

Output: A node s which is a parent of n and is a Scapegoat node

```
size = 1, height = 0
while (n.parent <> null) do
    height = height + 1
    totalSize = 1 + size + n.sibling.size()
    if height >  $\lceil \log_{1/\alpha}(\text{totalSize}) \rceil$  then
        return n.parent
    end if
    n = n.parent, size = totalSize
end while
```

```
private Node<Key,Value> findScapegoat(Node<Key,Value> node) {
    int size = 1;
    int height = 0;
    int subtreeSize = 0;
    while (node.parent != null) {
        height++;
        subtreeSize = 1 + size + size(getSibling(node));
        if (height > (int) Math.floor(Math.log(subtreeSize) /
Math.log(1.0/alpha))) {
            return node.parent;
        }
        node = node.parent;
        size = subtreeSize;
    }
    // something went wrong
    return root;
}
```

Rebalancing by Rebuilding

Flatten the SGT into the list of nodes in order of their keys (use BST **INORDER** walk). From the given list build a new tree:

- Divide the list into three parts:
 - the first half of the nodes,
 - the second half of the nodes, and
 - a root

Recursively build a new $\frac{1}{2}$ -weight-balanced trees.

- The middle node is the root
- Left subtree is the $\frac{1}{2}$ -weight-balanced tree built from the first half of nodes.
- Right subtree is the $\frac{1}{2}$ -weight-balanced tree built from the second half of nodes.

Input: Root of tree T

Output: The list of all nodes in T in order of their keys

```
FLATTENTREE(root, head)
if root == null then
    return head
end if
root.right = FLATTENTREE(root.right, head)
return FLATTENTREE(root.left, root)
```

A straightforward way of rebuilding a tree is to use a stack of logarithmic size to traverse the tree in-order in linear time and copy its nodes to an auxiliary array. Then build the new 1/2-weight-balanced tree using a “divide and conquer” method. This yields $O(n)$ time and space complexity. Our methods improve the space complexity to logarithmic.

The original paper by Galperin and Rivest mentions stacks, so let's use a stack.

```
private List<Node<Key,Value>> flattenTree(Node<Key,Value> node) {  
    List<Node<Key,Value>> nodes = new ArrayList<>();  
    Stack<Node<Key,Value>> stack = new Stack<>();  
    // flatten tree without recursion with in-order walk  
    // using stack described in GalparinRivest93  
    Node<Key,Value> currentNode = node;  
    while(true) {  
        if(currentNode != null){  
            stack.push(currentNode);  
            currentNode = currentNode.left;  
        }  
        else {  
            if(!stack.isEmpty()) {  
                currentNode = stack.pop();  
                nodes.add(currentNode);  
                currentNode = currentNode.right;  
            }  
            else {  
                return nodes;  
            }  
        }  
    }  
}
```



Input: List of **size** nodes in order of their keys headed by node **head**

Output: A $\frac{1}{2}$ -weight-balanced tree built from given list. The last node of the list will be returned.

```
BUILDTREE(size, head)
```

```
if (size == 1) then return head
```

```
else if (size == 2) then  
    (head.right).left = head
```

```
    return head.right
```

```
end if
```

```
root = (Build-Tree([(size - 1)/2], head)).right
```

```
last = Build-Tree([(size - 1)/2], root.right)
```


```
root.left = head
```

```
return last
```

Input: A scapegoat node, the size of the subtree rooted at scapegoat

Output: A $\frac{1}{2}$ -weight-balanced subtree built from all the nodes of the subtree rooted at scapegoat. The root of the rebuilt subtree will be returned.

```
REBUILDTREE(size, scapegoat)
head = FLATTENTREE(scapegoat, null)
BUILDTREE(size, head)
while (head.parent != null) do
    head = head.parent
end while
return head
```

The logo of Rowan University is visible in the background. It features a circular emblem with a yellow torch and flame, set against a reddish-brown background. The words "Rowan University" are written in a large, light-colored serif font across the middle of the image.


```
private Node<Key,Value> rebuildTree(Node<Key,Value> scapegoat, int treeSize) {
    List<Node<Key,Value>> nodes = new ArrayList<>();

    // flatten tree without recursion with in-order walk
    Node<Key,Value> currentNode = scapegoat;
    boolean done = false;
    Stack<Node<Key,Value>> stack = new Stack<>();
    while (!done) {
        if (currentNode != null) {
            stack.push(currentNode);
            currentNode = currentNode.left;
        } else {
            if (!stack.isEmpty()) {
                currentNode = stack.pop();
                nodes.add(currentNode);
                currentNode = currentNode.right;
            } else {
                done = true;
            }
        }
    }

    // build tree from flattened list of nodes
    return buildTree(nodes, 0, treeSize - 1);
}
```

```
private Node<Key,Value> buildTree(List<Node<Key,Value>> nodes, int start, int
end) {
    int median = (int) Math.ceil(((double)(start + end)) / 2.0);
    if (start > end) {
        return null;
    }

    // median becomes root of subtree instead of scapegoat
    Node<Key,Value> node = nodes.get(median);

    // recursively get left and right nodes
    Node<Key,Value> leftNode = buildTree(nodes, start, median - 1);
    node.left = leftNode;
    if (leftNode != null) {
        leftNode.parent = node;
    }
    Node<Key,Value> rightNode = buildTree(nodes, median + 1, end);
    node.right = rightNode;
    if (rightNode != null) {
        rightNode.parent = node;
    }
    return node;
}
```

Delete

Delete the node as we would in a regular BST.

- If $(T.size < \alpha T.maxSize)$ then
 - Rebuilt the whole tree into $\frac{1}{2}$ -weight-balanced BST
 - $T.maxSize = T.size$

Intuition: *if* $(T.size < \alpha T.maxSize)$

- T probably is not α -weight-balanced and
- T probably is not loosely α -height-balanced either.
- To maintain property of being α -height-balanced or loosely α -height-balanced we need to rebuilt T.

Input: The integer key k

Output:

DELETE (k)

deleted = DELETEKEY(k)

if (deleted) then

if ($T.size < (T.\alpha \times T.maxSize)$) then

BUILDTREE($T.size$, $T.root$)

end if

end if

4.3 Deleting from a scapegoat tree. Deletions are carried out by first deleting the node as we would from an ordinary binary search tree, and decrementing $size[T]$. Then, if

$$(4.7) \quad size[T] < \alpha \cdot max_size[T]$$

we rebuild the whole tree, and reset $max_size[T]$ to $size[T]$.

```
public void delete(Key key) {
    root = delete(root, key);
    if (size < alpha * maxSize) {
        // rebuild tree completely
        root = buildTree(flattenTree(root), 0, size);
        maxSize = size;
    }
}
```

```
private Node<Key, Value> delete(Node<Key, Value> node, Key key) {
    if (node == null) {
        return null;
    }
    int compareResult = comparator.compare(key, node.key);
    if (compareResult < 0) {
        node.left = delete(node.left, key);
    }
    else {
        if (compareResult > 0) {
            node.right = delete(node.right, key);
        }
        else {
            // node to be deleted. need to adjust parent pointer!
            // three cases: 1. no left child
            //                  2. no right child
            //                  3 two children
        }
    }
}
```

```
if (node.left == null) {
    if (node.right != null) {
        node.right.parent = node.parent;
    }
    size--;
    return node.right;
}
else {
    if (node.right == null) {
        node.left.parent = node.parent;
        size--;
        return node.left;
    }
    else {
        // two children
        Node<Key, Value> successor = minimum(node.right);
        node.key = successor.key;
        node.value = successor.value;
        node.right = delete(node.right, node.key);
        size--;
    }
}
}
}
return node;
}
```

Properties of Scapegoat Trees

- Find/Search/Get in a Scapegoat tree is $O(\log(n))$.
- Scapegoat trees store the height of the whole tree in the root node.
- Scapegoat trees have no overhead compared to other self-balancing trees.
- Scapegoat trees are α -weight-balanced binary search trees and because of that also α -height-balanced. For a Scapegoat tree, $\frac{1}{2} \leq \alpha \leq 1$ where $1/2$ is a (as much as possible) perfectly balanced binary tree.
- Scapegoat nodes are ancestors of deep nodes.
- The height of a Scapegoat tree storing n entries is $O(\log(n))$.
 - Can be proven recursively.