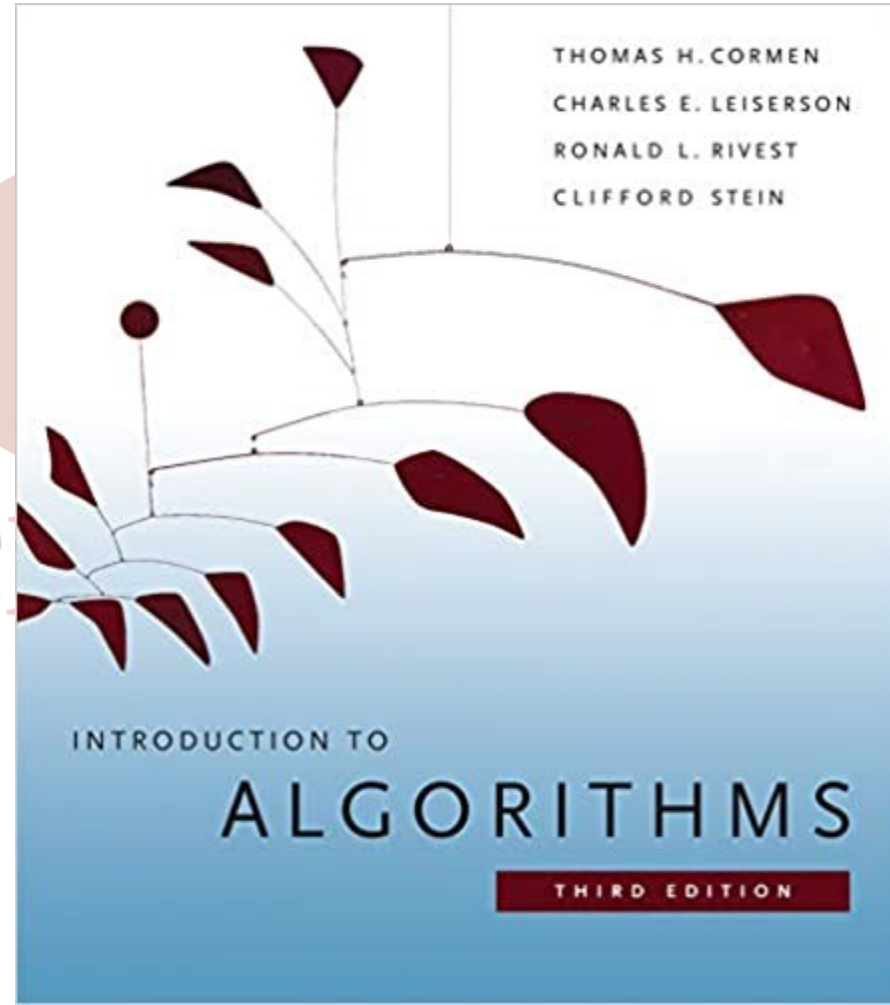


CS 07540 Advanced Design and Analysis of Algorithms

Week 4

- [Scapegoat Trees](#)
 - Data structure
- [2-3 Trees](#)
 - Data structure
 - [Runtime analysis](#)
- [Red-Black Trees](#)
 - Data structure
 - [Runtime analysis](#)

(Using additional material from [Galperin & Rivest, Scapegoat Trees](#) and from *Sedgewick & Wayne, Algorithms 4e*)



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 weight (and hence height) balanced trees. Other height balanced trees we will look at are 2-3 trees and red-black trees. We have already seen 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.

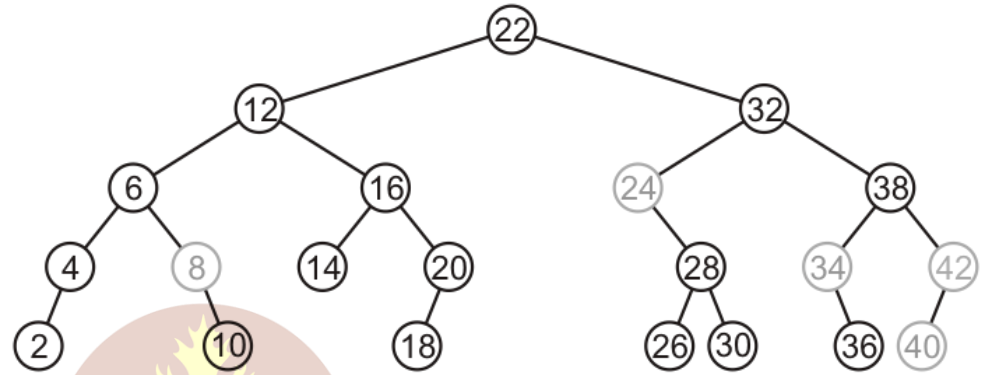
Balancing

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

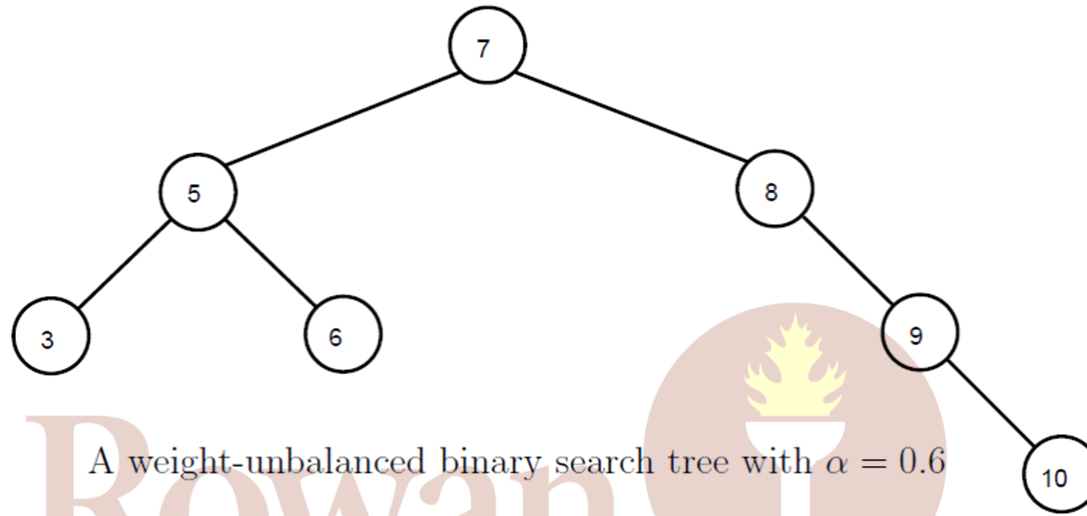
A binary-tree search (sub)tree rooted at **node** is **α -weight-balanced** for some α , $\frac{1}{2} \leq \alpha < 1$ if

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

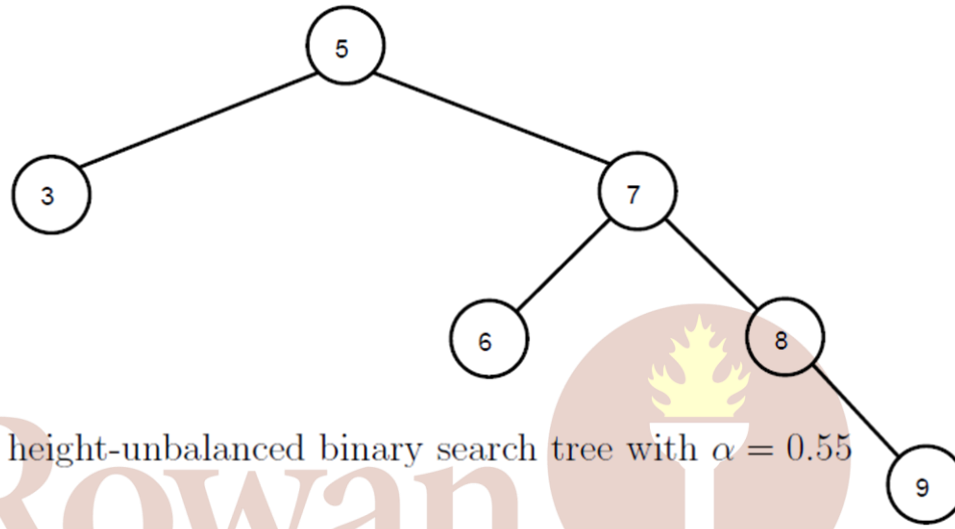
A tree is **α -weight-balanced** if for a given value α , $\frac{1}{2} \leq \alpha < 1$, all the nodes in the tree are α -weight-balanced. Intuitively, a tree is α -weight-balanced if for any subtree, the sizes of its left and right subtree are approximately equal.

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

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



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

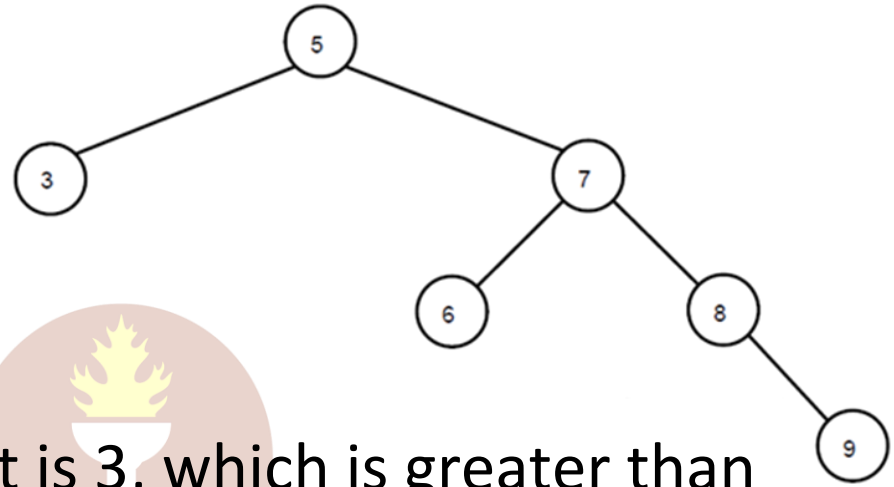
$$\text{depth}(\text{node}) > T.h_\alpha$$

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.

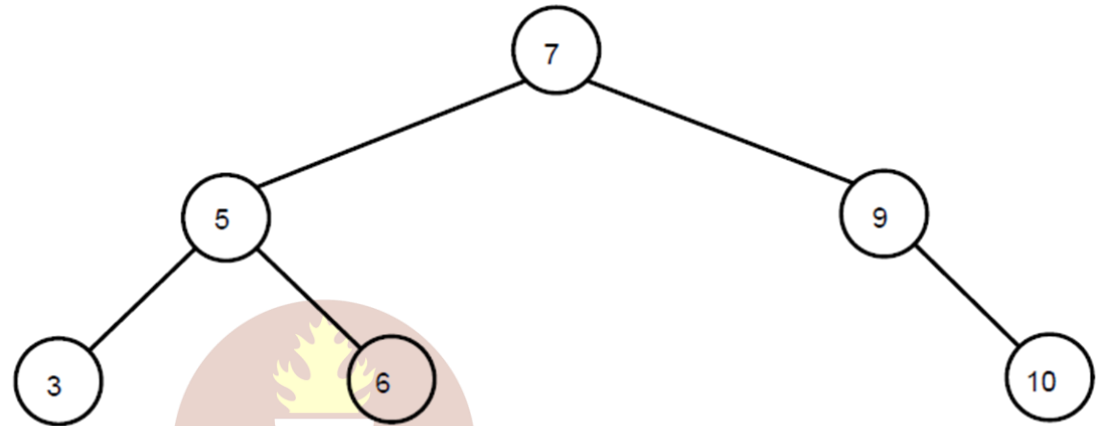
The tree T is an example of loosely 0.55-height-balanced tree because $T.\text{height} = 3$ and

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

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.

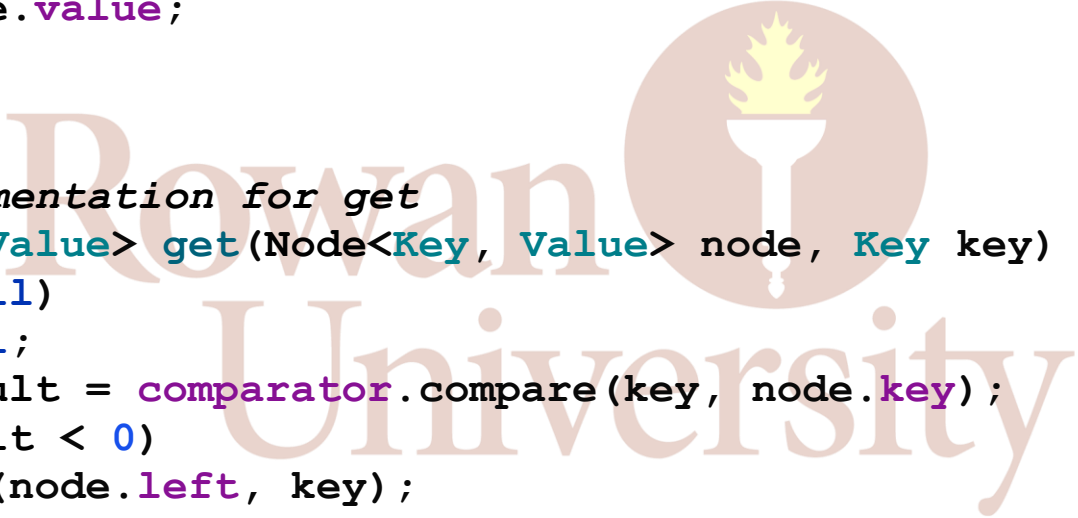
Input: **root/node** of some subtree T to search for an integer key **k**

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
```

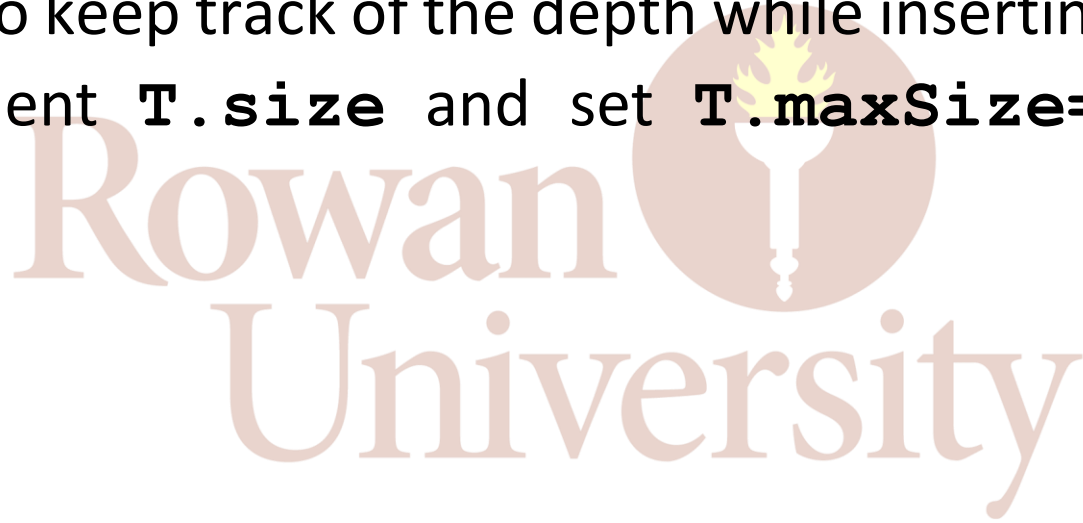
```
// public interface for get
// @param void
// @return node.value containing key
public Value get(Key key) {
    Node<Key, Value> node = get(this.root, key);
    if (node != null)
        return node.value;
    return null;
}

// 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

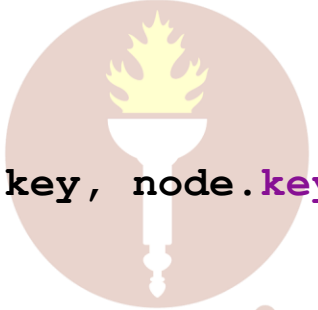
Insertion is done the same way as it is in a regular BST unless the inserted node is a deep node. Then we need to rebalance the tree. To detect deep nodes we need to keep track of the depth while inserting a node, and we need to increment **T.size** and set **T.maxSize=max(T.size, T.maxSize)**.



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;
    }
}
}
}
}
```

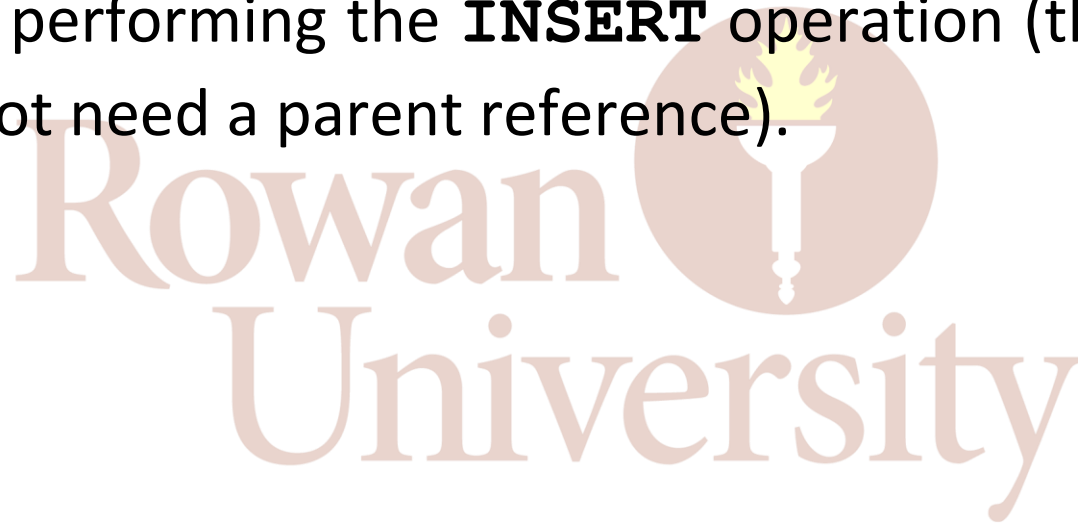
Input: The integer key **k**

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 size 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.

2-3 Trees

We set out to create guaranteed balanced search trees. The primary step to get the flexibility that we need to guarantee balance in search trees is to allow the nodes in our trees to hold more than one key. Specifically, referring to the nodes in a standard BST as 2-nodes (they hold two links and one key), we will now also allow 3-nodes, which hold three links and two keys. Both 2-nodes and 3-nodes have one link for each of the intervals subtended by its keys.

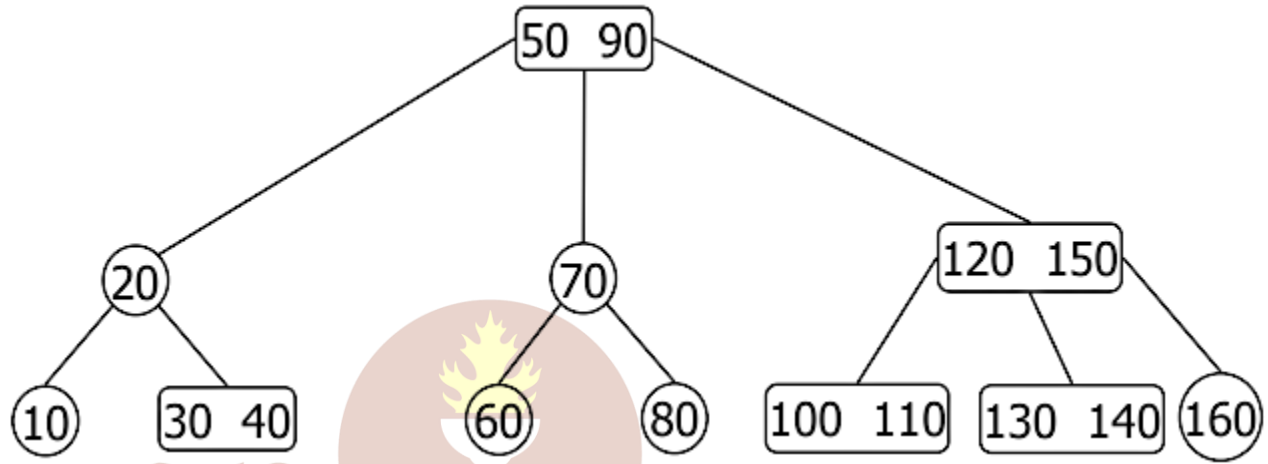
Unlike standard BSTs, which grow top down, 2-3 trees will grow up from the bottom. Its balancing algorithm will rely on ***splits*** and ***merges*** of nodes (unlike Red-Black trees and AVL trees, which use ***rotations***).

A **2-3 Search Tree** (2-3 tree) is a tree that is either empty or

- a 2-node, with one key (and associated value) and two links,
 - a left link to a 2-3 search tree with smaller keys, and
 - a right link to a 2-3 search tree with larger keys
- a 3-node, with two keys (and associated values) and three links
 - a left link to a 2-3 search tree with smaller keys,
 - a middle link to a 2-3 search tree with keys between the node's keys, and
 - a right link to a 2-3 search tree with larger keys.

Note that links could be **null**!

A perfectly balanced 2-3 search tree is one whose leaves are all at the same distance from root:



- each interior node has either 2 or 3 children (possibly **null**)
- **all the leaves are at the same level** (as part of construction)

Each interior node contains one or two values:

- $L[v]$... largest value in the left subtree
- $M[v]$... largest values in the middle or second subtree

2-3 Tree Methods Overview

We are interested in the following operations:

SEARCH, MIN, MAX, INSERT, (DELETE, CONCATENATE, and SPLIT)



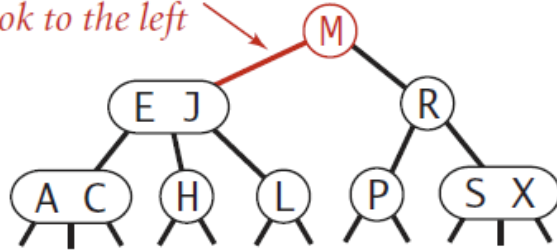
Search

The search algorithm for keys in a 2-3 tree directly generalizes the search algorithm for BSTs. To determine whether a key is in the tree, we

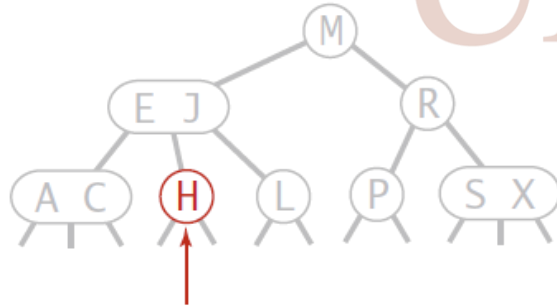
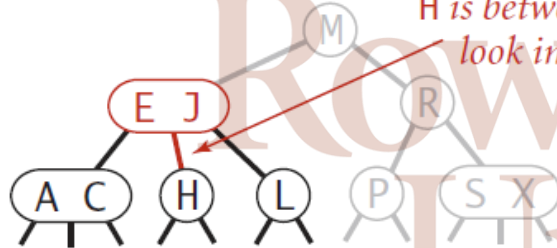
- compare it against the keys at the root. If it is equal to any of them, we have a search hit; otherwise,
- follow the link from root to the subtree corresponding to the interval of key values that could contain the search key.
 - if that link is null, we have a search miss; otherwise
 - recursively (or while loop) search in that subtree.

successful search for H

*H is less than M so
look to the left*



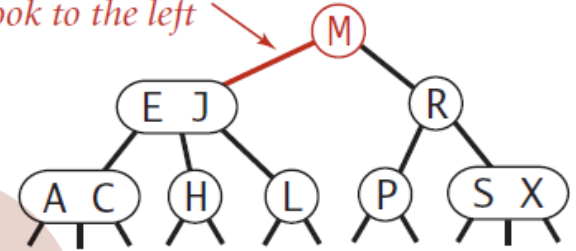
*H is between E and J so
look in the middle*



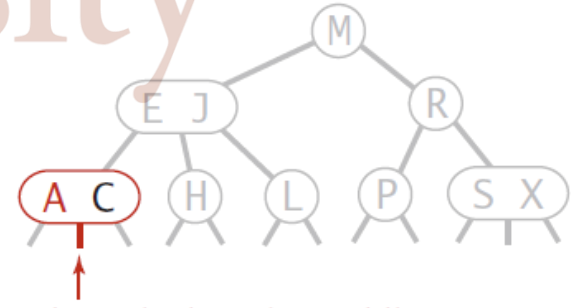
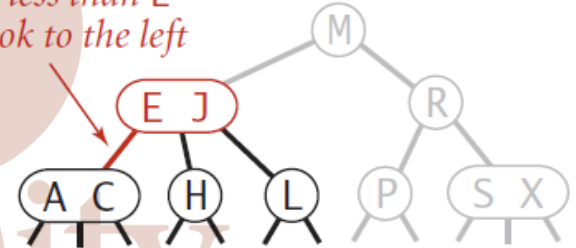
found H so return value (search hit)

unsuccessful search for B

*B is less than M so
look to the left*



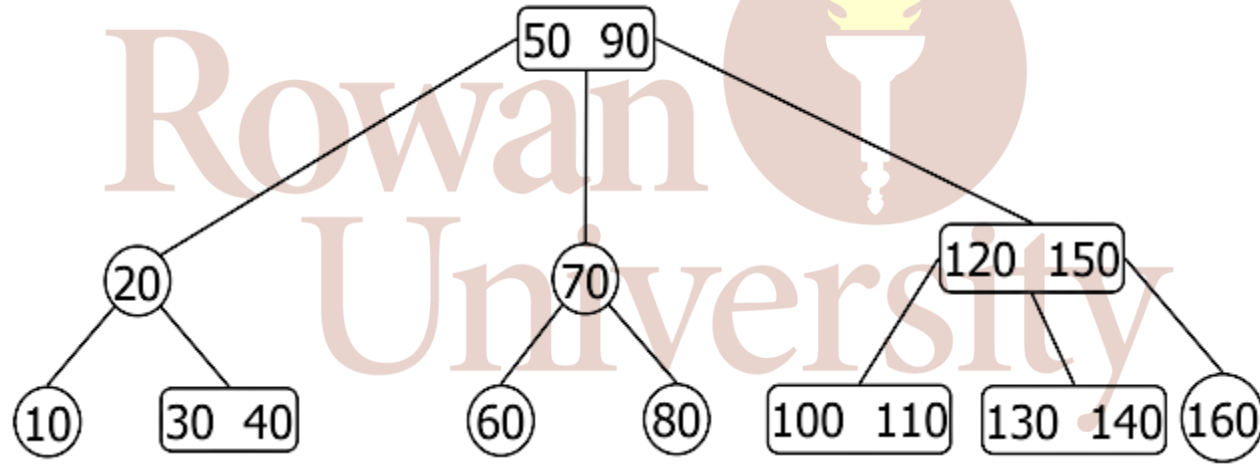
*B is less than E
so look to the left*



*B is between A and C so look in the middle
link is null so B is not in the tree (search miss)*

Min and Max

Travel through the 2-3 tree to reach to the leftmost or the rightmost leaf. Cost is $O(\log(n))$. Alternatively, we can keep (and update) pointers to largest and smallest elements (cost $O(1)$).

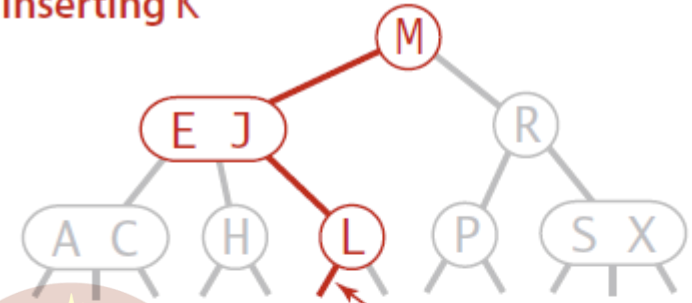


Insert (2-node)

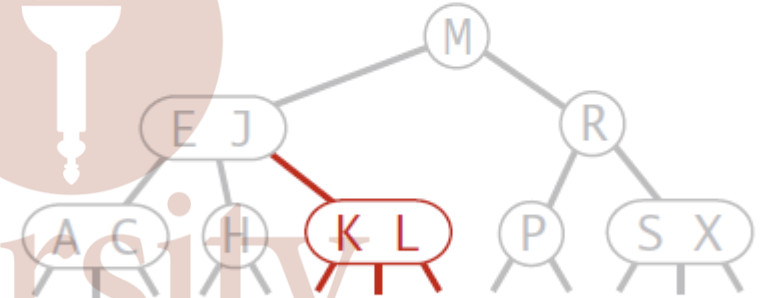
In general, to insert a new node in a 2-3 tree, we might do an unsuccessful search and then add a node at the bottom, as we did with BSTs. However, then the new tree would not remain perfectly balanced. We will need to consider cases!

If the search terminates at a 2-node, we can do insertions and still maintain perfect balance.

inserting K



search for K ends here



replace 2-node with
new 3-node containing K

Insert into a 2-node

Insert (tree is single 3-node)

Before we consider the general case, suppose we have a simplistic 2-3 tree consisting of just one 3-node (root). Such a tree has two keys, but no room for a new key in its one node.

- Temporarily create a 4-node
- Split 4-node into 2-3 tree with root and two 2-nodes

This works for this artificial example, but it **increases the height of the tree!**

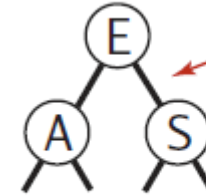
inserting S



← no room for S



← make a 4-node



← split 4-node into this 2-3 tree

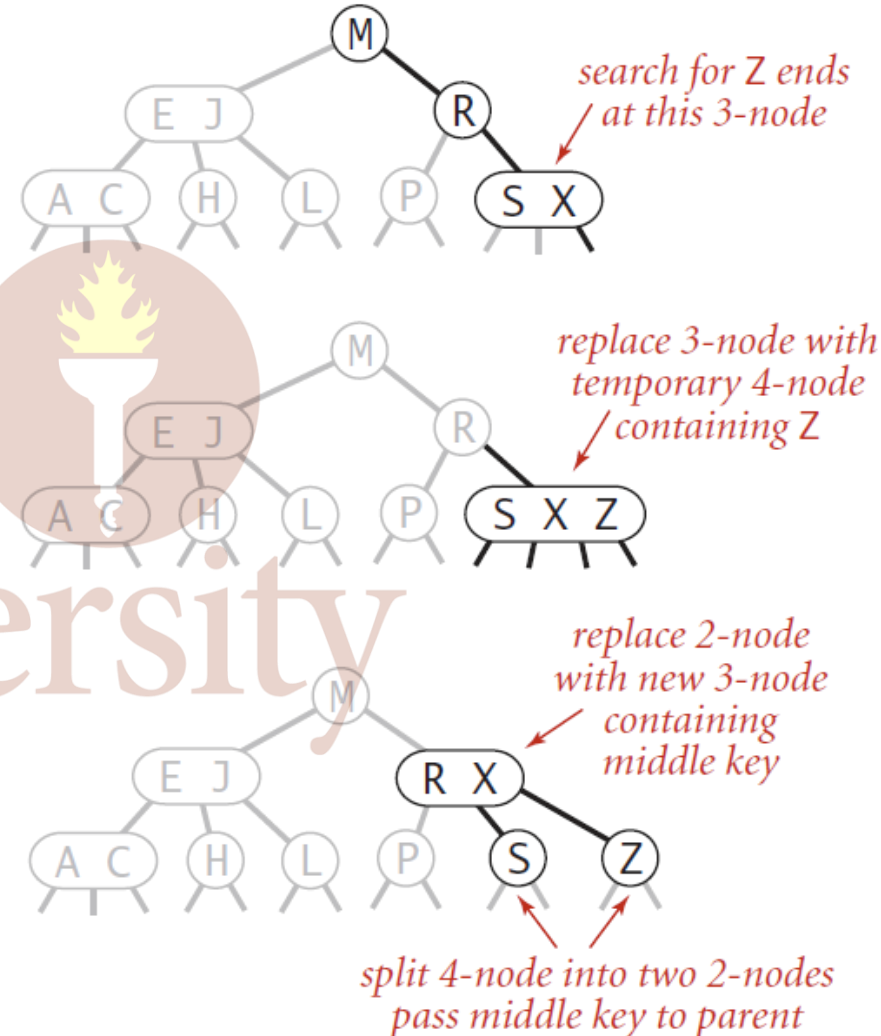
Insert into a single 3-node

Insert (3-node with parent 2-node)

We can make room for the new key while maintaining balance in the tree: make a temporary 4-node and then split the 4-node by moving the middle key to the node's parent.

- Temporarily create 4-node
- Split 4-node by moving middle key to parent (at the correct position). Make left and right key of 4-node new 2-node children.

inserting Z



We should notice a theme here: As long as there is some 2-node above the insertion, we should be able to make room for a new key in a 3-node by moving a key from the insertion 3-node up. Eventually this will fail when all nodes are 3-nodes along an insertion path, though.

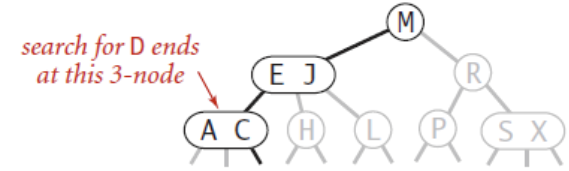


Insert (3-node with parent 3-node)

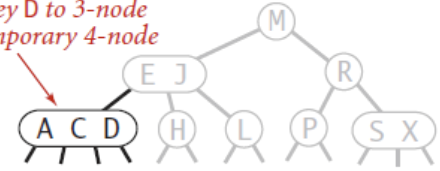
- Make a temporary 4-node at insertion
- Split insertion 4-node and add middle key to parent
 - Repeat recursively, if possible, until no splitting is necessary, or root is turned into a 4-node

If root is turned into a 4-node, we need to split root and increase the height of the tree.

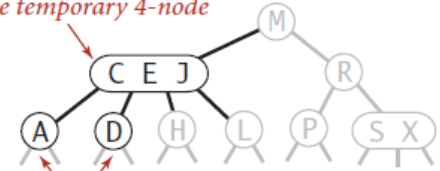
inserting D



add new key D to 3-node to make temporary 4-node

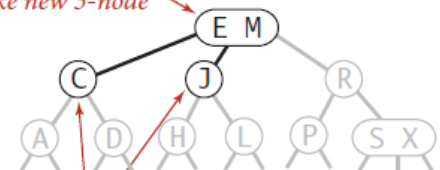


add middle key C to 3-node to make temporary 4-node



split 4-node into two 2-nodes
pass middle key to parent

add middle key E to 2-node to make new 3-node



split 4-node into two 2-nodes
pass middle key to parent

Insert into a 3-node whose parent is a 3-node

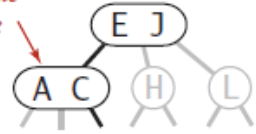
Insert (Splitting the root)

If we have 3-nodes along the whole path from the insertion point to root, we end up with a temporary 4-node at root. We already looked at this case in our first attempt at 3-nodes.

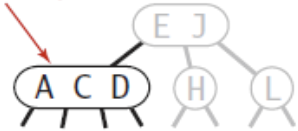
Now we have a theoretical procedure for inserts into a perfectly balanced 2-3 tree. These trees will maintain balance while growing and guarantee $O(\log(n))$ runtime. What about **INSERT** runtime?

inserting D

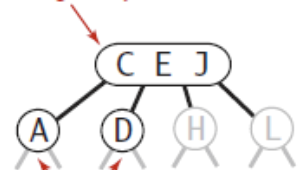
search for D ends
at this 3-node



add new key D to 3-node
to make temporary 4-node

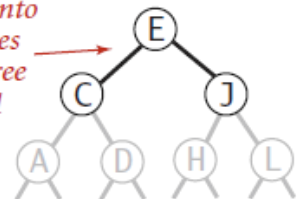


add middle key C to 3-node
to make temporary 4-node



split 4-node into two 2-nodes
pass middle key to parent

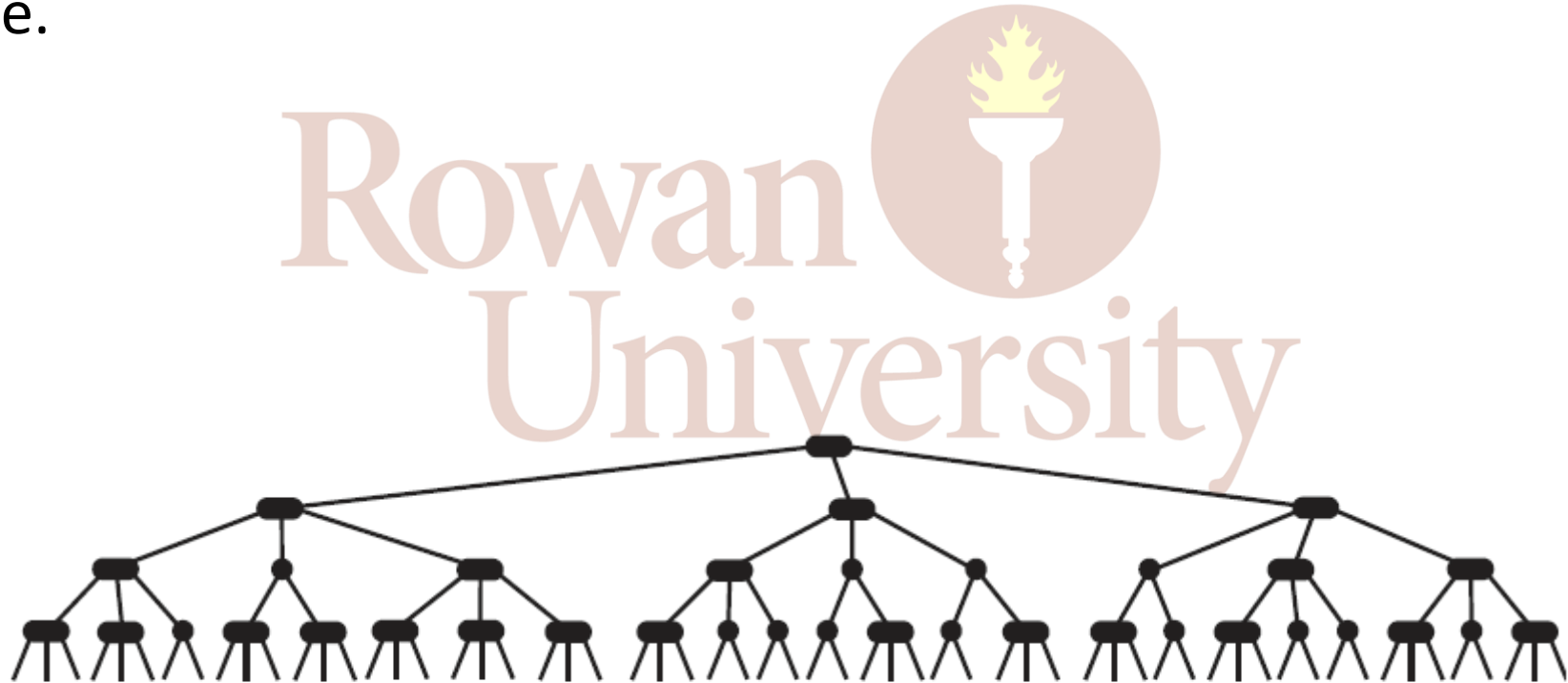
split 4-node into
three 2-nodes
increasing tree
height by 1



Splitting the root

Example

We have seen in class that an ordered set inserted into a BST will create a linked list. Insert the ordered set of keys $\{1,2,3,4,5,6,7\}$ into an empty 2-3 tree.



Typical 2-3 tree built from random keys

DelMin

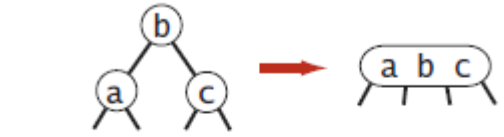
Deletion of elements is more involved since we want to maintain balance. Let's consider deleting the minimum instead of an arbitrary key.

The minimum is the item we reach following left pointers from root. We can easily delete a key from a 3-node at the bottom of the tree, but not from a 2-node. Deleting the key from a 2-node leaves a node with no keys, and we cannot simply leave it as null since that violates the balance. What we could do *on the path from root down to the minimum* is to make sure every node visited is a 3-node or temporary 4-node by performing appropriate transformations to the subtree. This way, a 2-node leaf can borrow from its parent using a ***split***. If the split is not needed (because the minimum is actually in a 3-node), we can ***merge on our way back up***.

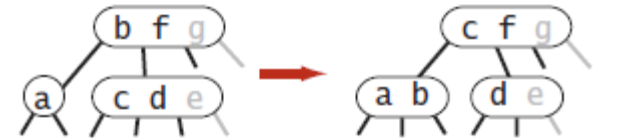
At root, there are two possibilities for a 2-node:

- root is a 2-node and both children are 2-nodes
 - convert the three nodes to a 4-node
- root is a 2-node and at least one child is a 3-node
 - borrow from the right sibling if necessary to ensure that the left child of the root is not a 2-node

at the root



on the way down



at the bottom

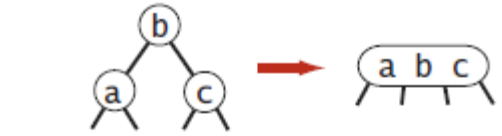


Transformations for delete the minimum

On the way down to the minimum, one of the following must hold:

- If the left child of the current node is not a 2-node, there is nothing to do.
- If the left child is a 2-node and its immediate sibling is not a 2-node, move a key from the sibling to the left child.
- If the left child and its immediate sibling are 2-nodes, then combine them with the smallest key in the parent to make a 4-node, changing the parent from a 3-node to a 2-node or from a 4-node to a 3-node.

at the root



on the way down



at the bottom



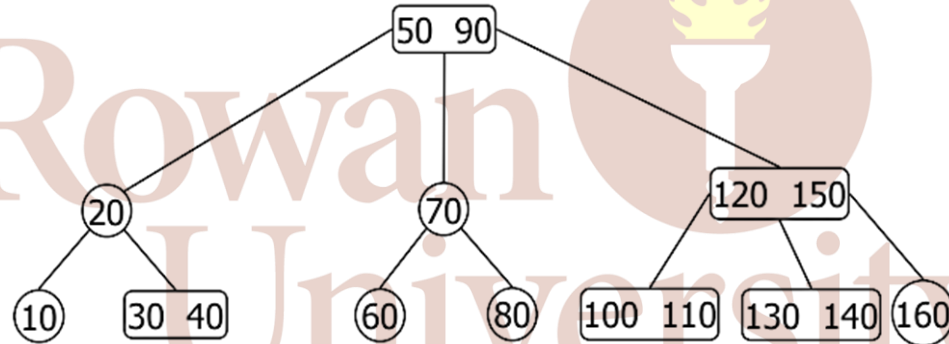
Transformations for delete the minimum

Following this process as we traverse left links to the bottom, we wind up on a 3-node or a 4-node with the smallest key, so we can just remove it, converting the 3-node to a 2-node or the 4-node to a 3-node. On the way up we split any unused temporary 4-nodes.

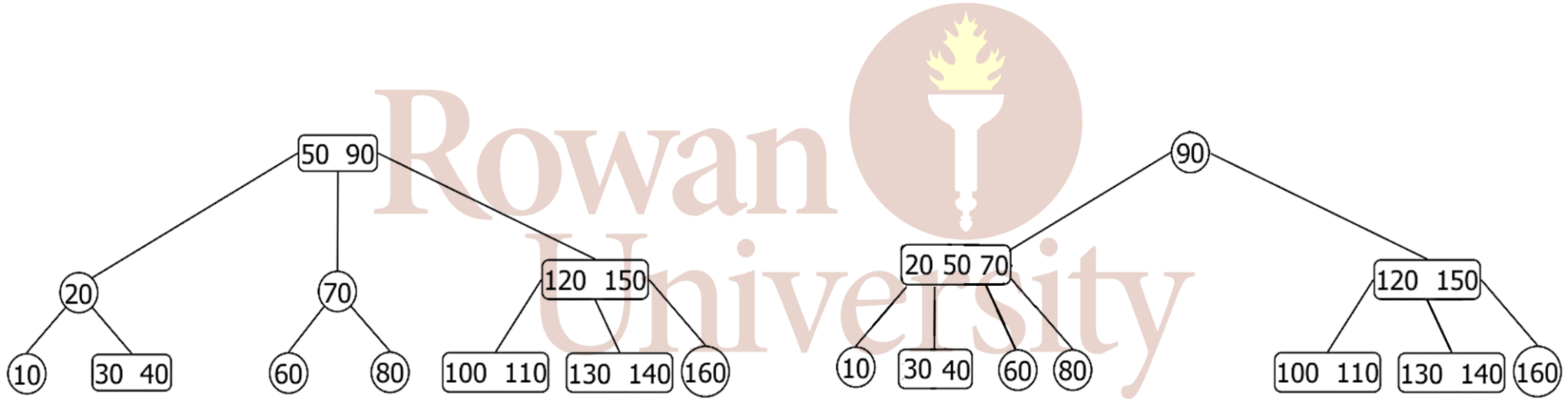


Example

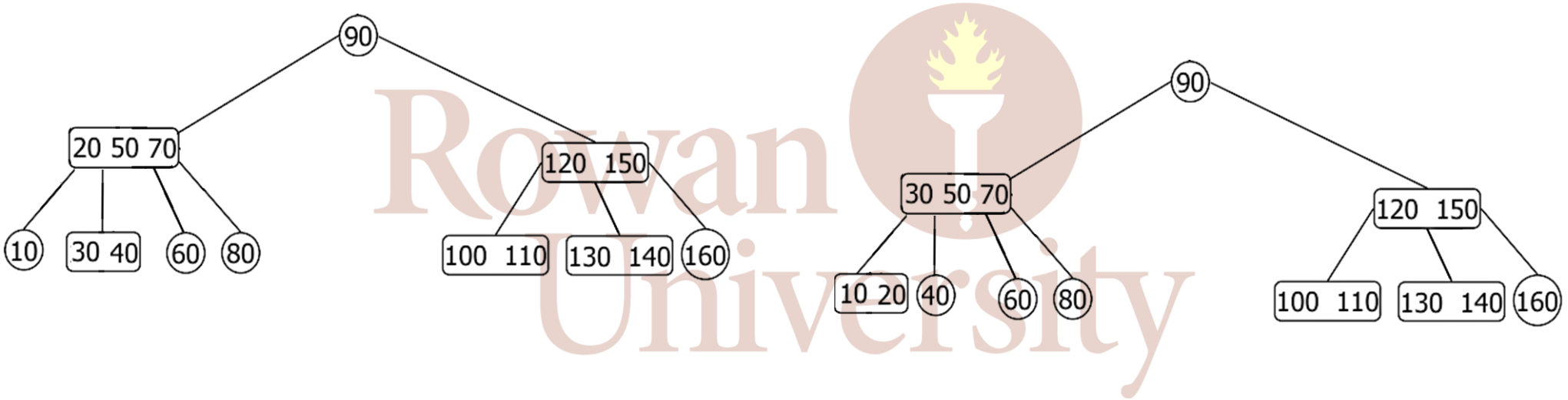
Delete the minimum in the given 2-3 tree using the algorithm just outlined. Merge while moving from root to minimum, and split (if necessary) moving back from new minimum to root.



Step 1: Left child of root is a 2-node, so we need to bubble down a key (data) to the left. From root (which here is a 3-node) to left child, bubble down left key and temporarily create a 4-node.



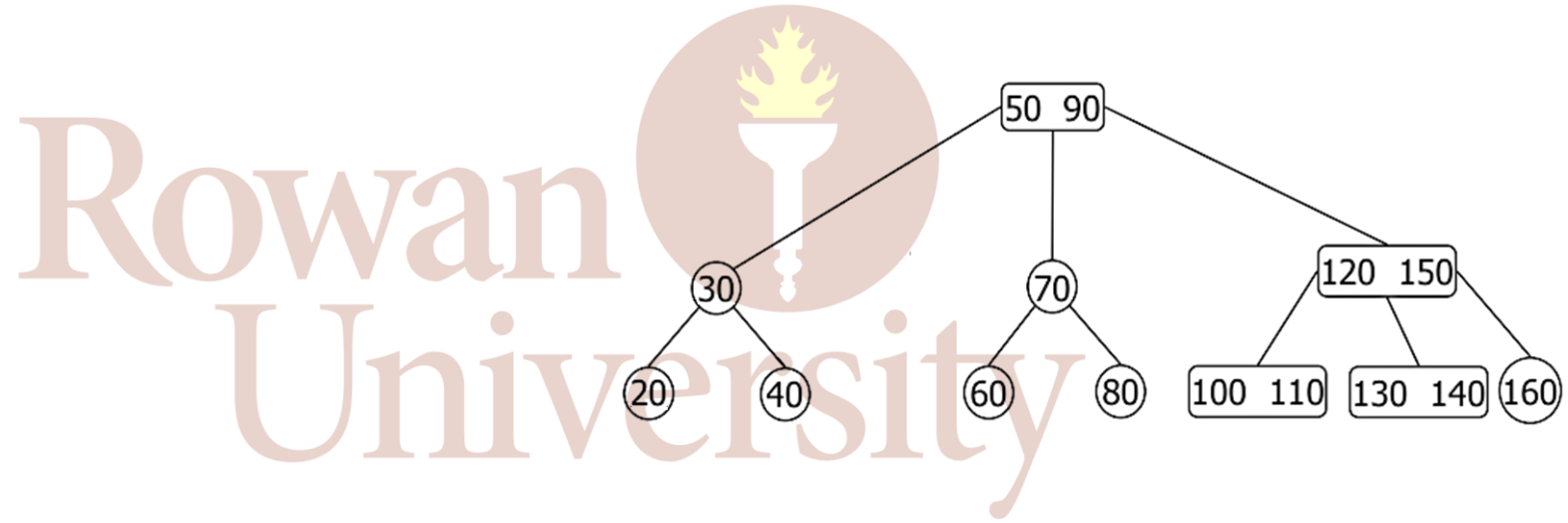
Step 2: From the current node (20, 50, 70), its left child is a 2-node. Bubble down key 20 and move up key 30.



Step 3: Delete minimum which is a 3-node.



Step 4: On the way from minimum towards root, parent of current node (20) is a 4-node. Split like we inserted a node. Middle key (50) bubbles up. Parent (90) becomes a 2-node with right sibling (70) going to the middle.



The same transformations along the search path just described for deleting the minimum are effective to ensure that the current node is not a 2-node during a search for any key. If the search key is at the bottom, we can just remove it. If the key is not at the bottom, then we have to exchange it with its successor as in regular BSTs. Then, since the current node is not a 2-node, we have reduced the problem to deleting the minimum in a subtree whose root is not a 2-node, and we can use the procedure just described for that subtree. After the deletion, we split any remaining 4-nodes on the search path on the way up the tree.

Delete

1. Find the element to be deleted.
2. Delete the element
 - a. If the subtree is left with 2 leaves then
 - i. Update internal nodes. Done
 - b. If the subtree is left with only one leaf then
 - i. Check the sibling (left or right)
 - ii. If the sibling has only two children then remove the parent and combine the child with the children of the sibling. If again the tree is left with only one child then repeat the same process recursively until either obtained a normal 2-3 tree or reached the root. If reached the root then delete it. Update internal nodes as you travel up.
 - iii. If the sibling has three children then take rightmost/leftmost child of the sibling and add it to this node's parent.

Insert Runtime Analysis

Method	Amortized	Worst case
Search	$O(\log(n))$	$O(\log(n))$
Insert	$O(\log(n))$	$O(\log(n))$
Delete	$O(\log(n))$	$O(\log(n))$

- Find insertion/deletion position is $O(\log(n))$ due to balance.
- We need to split at most $O(\log(n))$ times since there are only $\log(n)$ levels and we do constant amount of work at each level.

Splitting a 4-node is a local transformation which preserves order and perfect balance. Moreover, these local transformations preserve the global properties that the tree is ordered and perfectly balanced. The height of the tree is between $\log_3(n)$ and $\log_2(n)$.

2-3 trees are usually not implemented using the operations we described since we would need to track two different types of nodes. The overhead of constantly testing what type of node requires what type of case might eat up all the speedup.



2-3 Tree Summary

2-3 trees are strictly speaking not BSTs as they have 2-nodes and 3-nodes, but they are equivalent to Red-Black trees, which are BSTs. Their worst case run-time complexity for **SEARCH/INSERT/DELETE** is $O(\log(n))$.

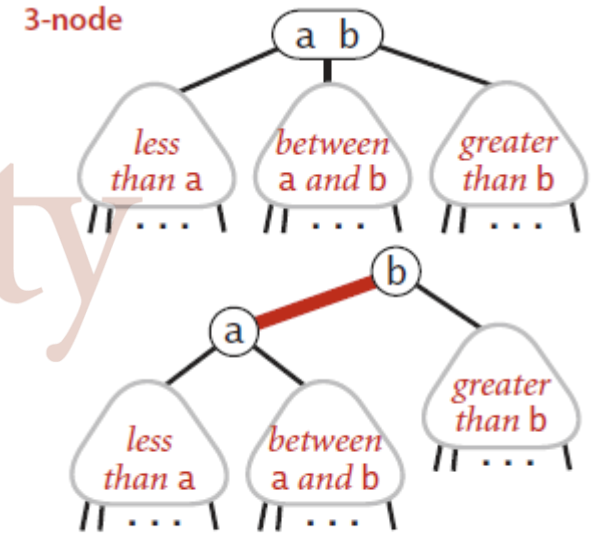
- Nodes have one (2-node) or two (3-node) keys.
- Trees grow bottom up.
- Height is between $\log_2(n)$ and $\log_3(n)$.
- Every path from the root to a leaf is of the same length, meaning all leaves are at the same level.
- Balancing uses splits and merges.

We will see later that amortized (average) run-time complexity for **SEARCH/INSERT/DELETE** is also $O(\log(n))$.

Red-Black Trees

A red-black tree is a **binary search tree** with one extra bit of storage per node: its color, which can be either red or black. By constraining the node colors on any simple path from the root to a leaf, **red-black trees ensure that no such path is more than twice as long as any other**, so that the tree is approximately balanced.

Red-black BSTs can also be used to implement 2-3 trees by coloring links rather than nodes. This is equivalent to coloring nodes (the parent pointer being the color). The exact balance of 2-3 trees turns into the known (for red-black trees) approximate balance due to additions in height.

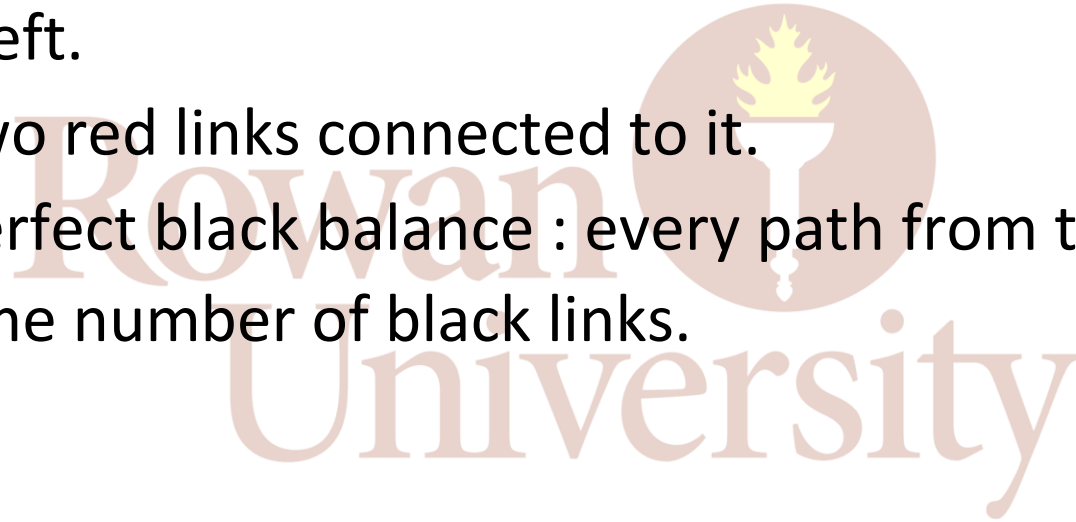


Encoding a 3-node with two 2-nodes connected by a left-leaning red link

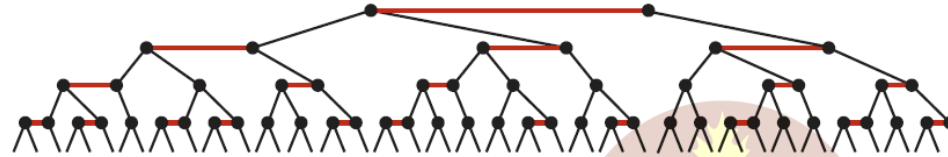
Red-Black Trees for 2-3 Trees

Red-black trees as an implementation for 2-3 trees are BSTs with red and black links with the following restrictions:

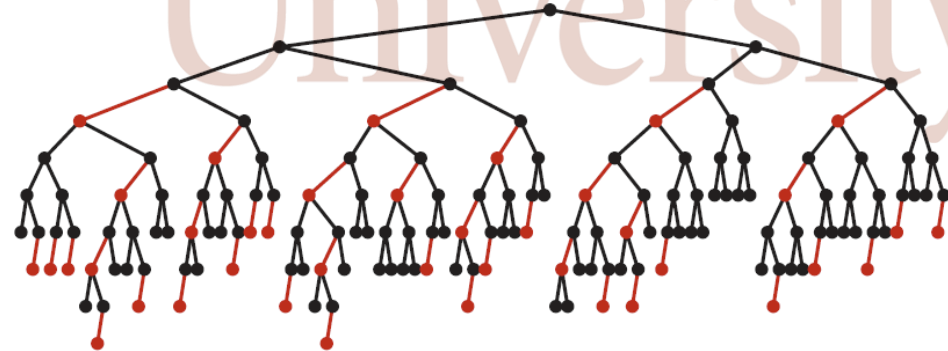
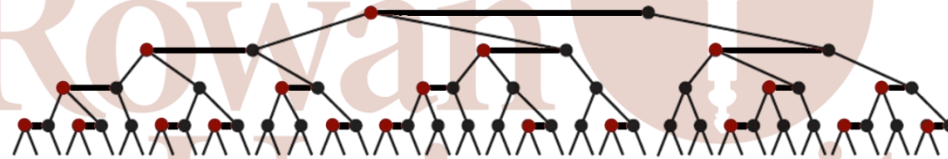
- Red links lean left.
- No node has two red links connected to it.
- The tree has perfect black balance : every path from the root to a null link has the same number of black links.



There is a 1-1 correspondence between red-black BSTs defined this way and 2-3 trees: Red links correspond to 3-nodes in a 2-3 tree.

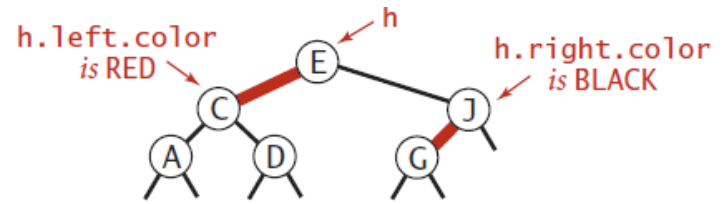


A red-black tree with horizontal red links is a 2-3 tree



Typical red-black BST built from random keys (null links omitted)

Since red/black is a boolean distinction, we can encode color with one bit. This can be both interpreted as the color of the node or the color of the parent link.



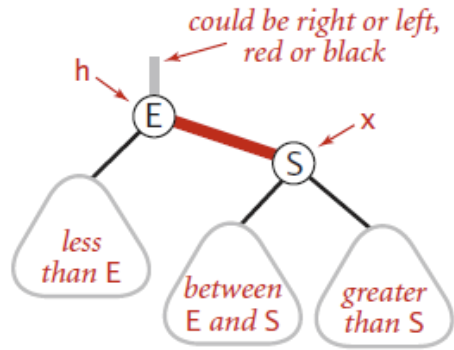
```
private static final boolean RED    = true;
private static final boolean BLACK  = false;

private class Node
{
    Key key;           // key
    Value val;         // associated data
    Node left, right;  // subtrees
    int N;             // # nodes in this subtree
    boolean color;     // color of link from
                      // parent to this node

    Node(Key key, Value val, int N, boolean color)
    {
        this.key    = key;
        this.val    = val;
        this.N      = N;
        this.color   = color;
    }
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

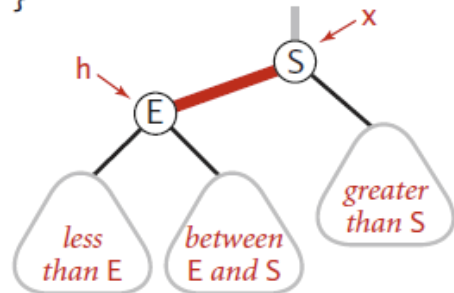
Node representation for red-black BSTs



```

Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}

```



Left rotate (right link of h)

Rotations

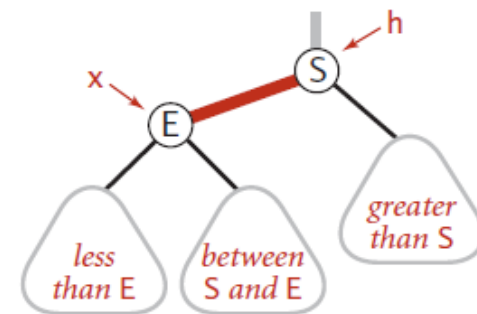
During insertions (and deletions), we may end up with right-leaning red links or two red links in a row. Red-black trees solve this problem through **rotations**, which switch the orientation of red links.

- Use left/right rotations to ensure the
 - Red links lean left.
 - No node has two red links connected to it.
 - The tree has perfect black balance.

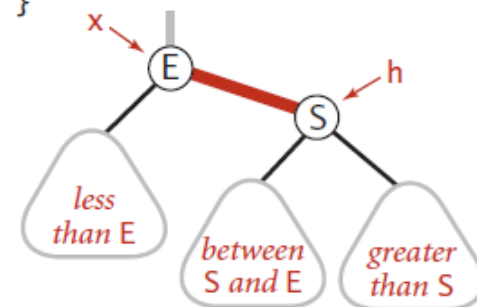
The recursive implementations of BST methods make this easier than iterative programming.

- Left and right rotations preserve order of keys.
- Perfect black balance (number of black nodes) is maintained if it had this property to begin with since pathways through E-S (in our picture) turn into pathways through S-E.

The other two properties need to be preserved when inserting or deleting.



```
Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```

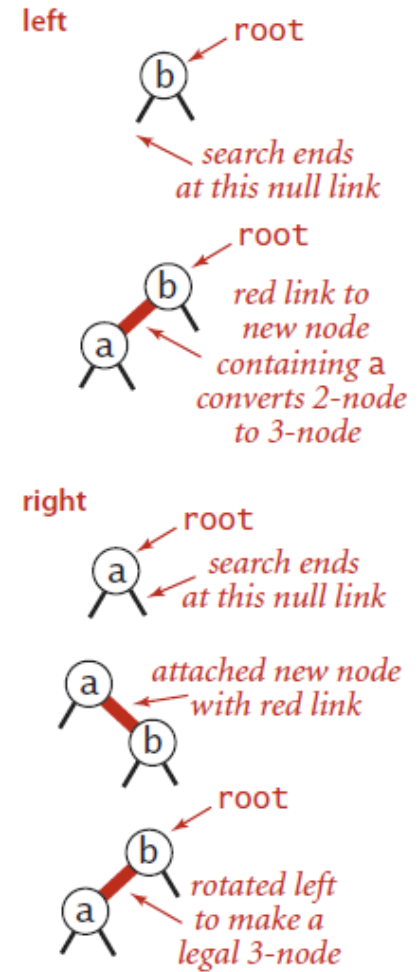


Right rotate (left link of h)

Insert (2-node)

A red-black BST with 1 key is just a single 2-node. In a 2-3 tree we would create a 3-node. This corresponds to two cases when translated to a binary tree.

- Inserted key is smaller than root key.
 - Insert on left with red link (3-node).
- Inserted key is larger than root key.
 - Insert on right with red link (3-node) and rotate left to make it a standard red-black tree “3-node”.



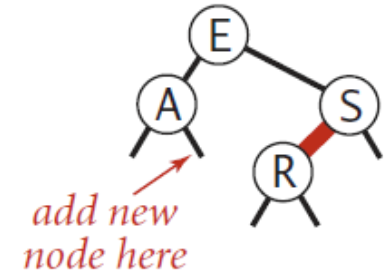
Insert into a single
2-node (two cases)

Insert (2-node at bottom)

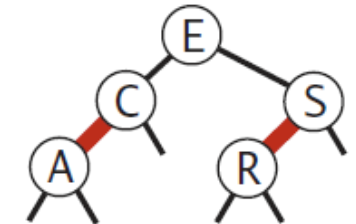
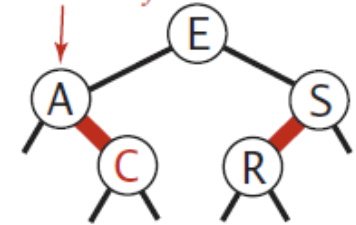
- Insert key as in a regular BST but connect with red link
- If parent (A) is a 2-node (link is black), rotate as in previous case to obtain proper coloring. Creates a 3-node.



insert C



right link red
so rotate left

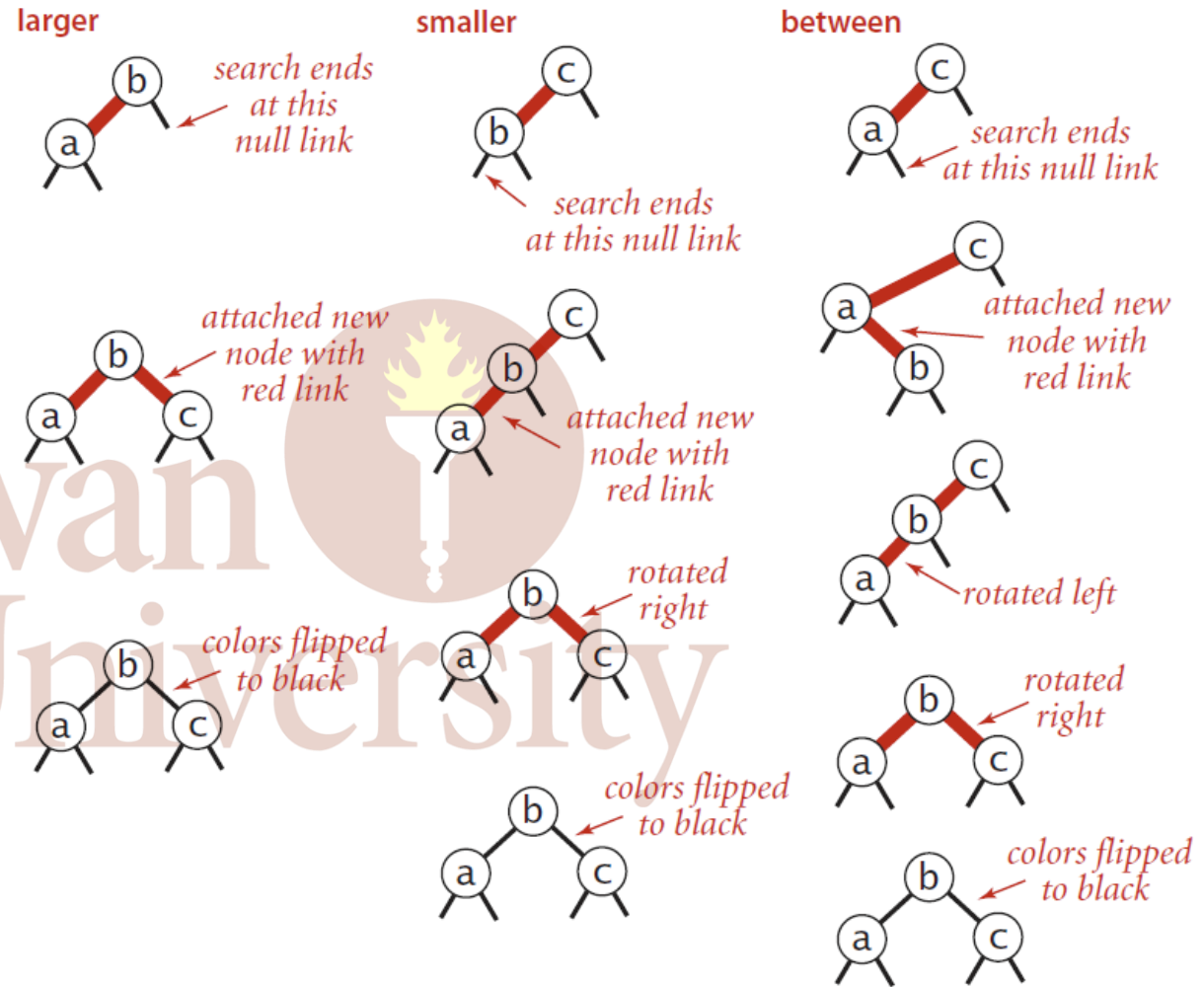


Insert into a 2-node
at the bottom

Insert (3-node)

A 3-node is a pair of two keys connected with a red edge. Hence its parent **must** be connected in black. There are three subcases:

- New key is larger than the two existing keys
- New key is smaller than both existing keys
- New key is in between

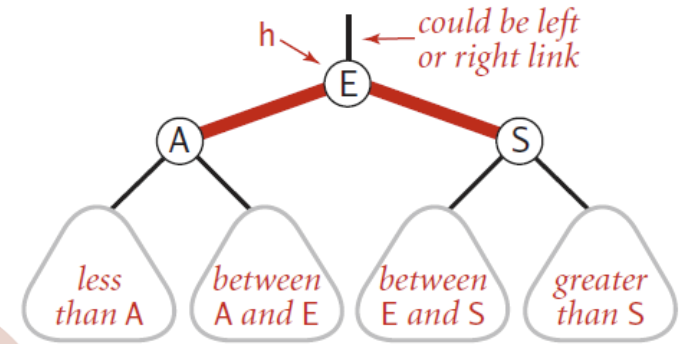


Insert into a single 3-node (three cases)

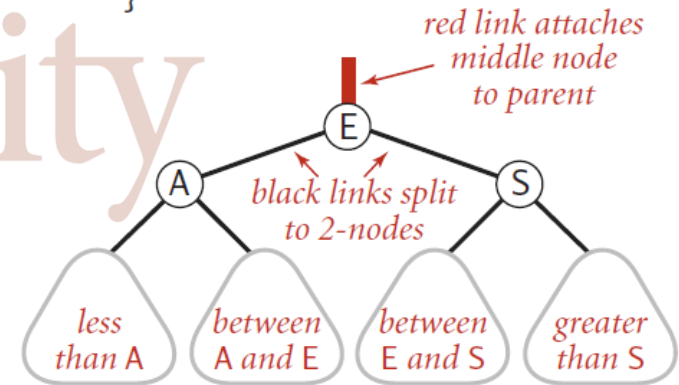
Flipping colors

Several of the insertion cases created two children with red links. In these cases, we flipped both colors from red to black. In addition to flipping the colors of the children, we also flip the color of the parent from black to red.

This local transformation maintains the global perfect black balance in the tree. Any path through the root node of the subtree in question during insertion will have the same number of black links.



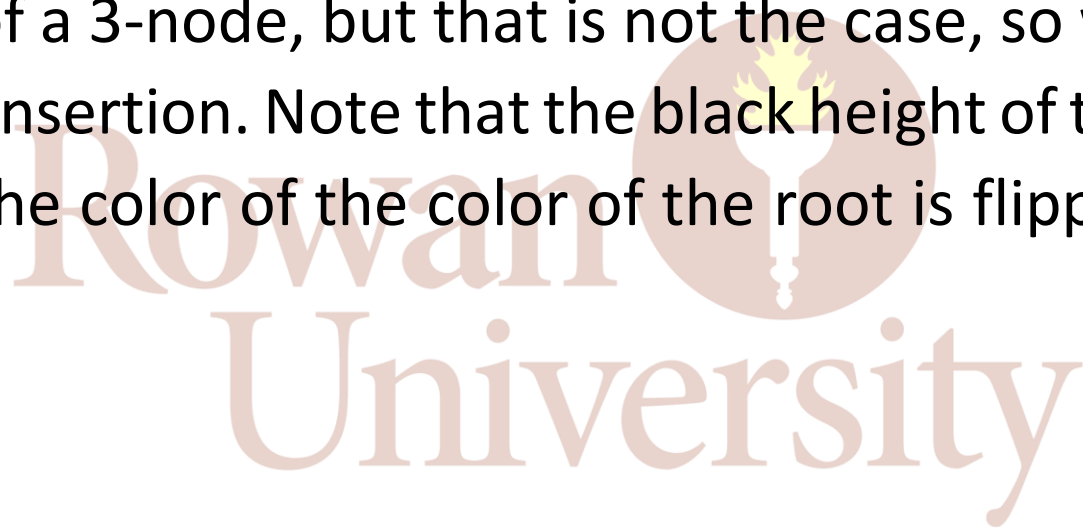
```
void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

Keeping the root black

Inserting into a single 3-node, the color flip will color the root red. This can also happen in larger trees. Strictly speaking, a red root implies that the root is part of a 3-node, but that is not the case, so we color the root black after each insertion. Note that the black height of the tree increases by 1 whenever the color of the root is flipped from black to red.

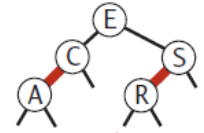
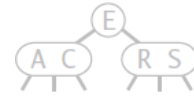


Insert (3-node at bottom)

Inserting a new key into a 3-node at the bottom (node with red parent link) amounts to passing up a key to its parent in a 2-3 tree. We have to move up the tree recursively to fix links by rotating.

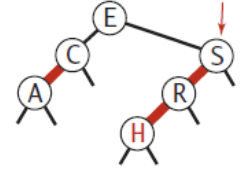
The 2-3 tree insertion algorithm calls for us to split the 3-node, passing the middle key up to be inserted into its parent, continuing until encountering a 2-node or the root.

inserting H

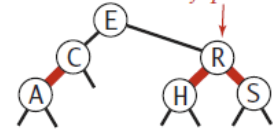


add new node here

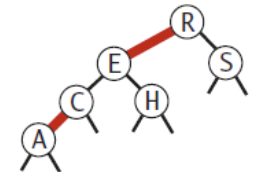
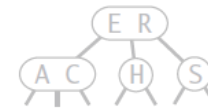
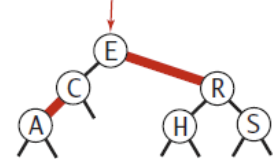
two lefts in a row so rotate right



both children red so flip colors



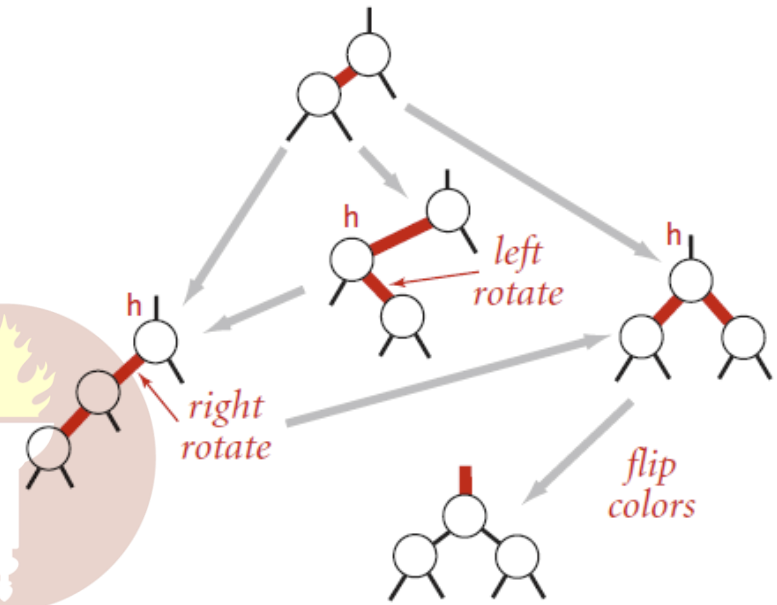
right link red so rotate left



Insert into a 3-node at the bottom

In every case we have considered, we precisely accomplish this objective:

- If the right child is red and the left child is black, rotate left.
- If both the left child and its left child are red, rotate right.
- If both children are red, flip colors.



Passing a red link up a red-black BST

After necessary rotations, we flip colors, which turns the middle node red. From the point of view of the parent of that node, that link becoming red can be handled in the same manner as if the red link came from attaching a new node: pass up a red link to the middle node.

Implementation Considerations

Since the balancing operations are to be performed on the way up the tree from the point of insertion, implementing them in a standard recursive implementation means executing them after the recursive calls.



Insert for red-black BSTs

```
public class RedBlackBST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node // BST node with color bit
    {
        private boolean isRed(Node h)
        private Node rotateLeft(Node h)
        private Node rotateRight(Node h)
        private void flipColors(Node h)

        private int size()

        public void put(Key key, Value val)
        { // Search for key. Update value if found; grow table if new.
            root = put(root, key, val);
            root.color = BLACK;
        }

        private Node put(Node h, Key key, Value val)
        {
            if (h == null) // Do standard insert, with red link to parent.
                return new Node(key, val, 1, RED);

            int cmp = key.compareTo(h.key);
            if (cmp < 0) h.left = put(h.left, key, val);
            else if (cmp > 0) h.right = put(h.right, key, val);
            else h.val = val;

            if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
            if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
            if (isRed(h.left) && isRed(h.right)) flipColors(h);

            h.N = size(h.left) + size(h.right) + 1;
            return h;
        }
    }
}
```

Delete

Deferred.



Red-Black Tree Summary

Red-black trees are used for implementing sets. Their worst case **SEARCH/INSERT/DELETE** are all $O(\log(n))$.

- Nodes are colored red or black.
- No parent-child is red-red.
- Every path from a node to a leaf goes through the same number of black nodes (**null** considered black).
- Longest path from root to a leaf is at most twice as long as shortest path from root to a leaf.
- Balancing uses rotations.

We will see later that amortized (average) run-time complexity for **SEARCH** is $O(\log(n))$ while **INSERT/DELETE** can be $O(1)$.