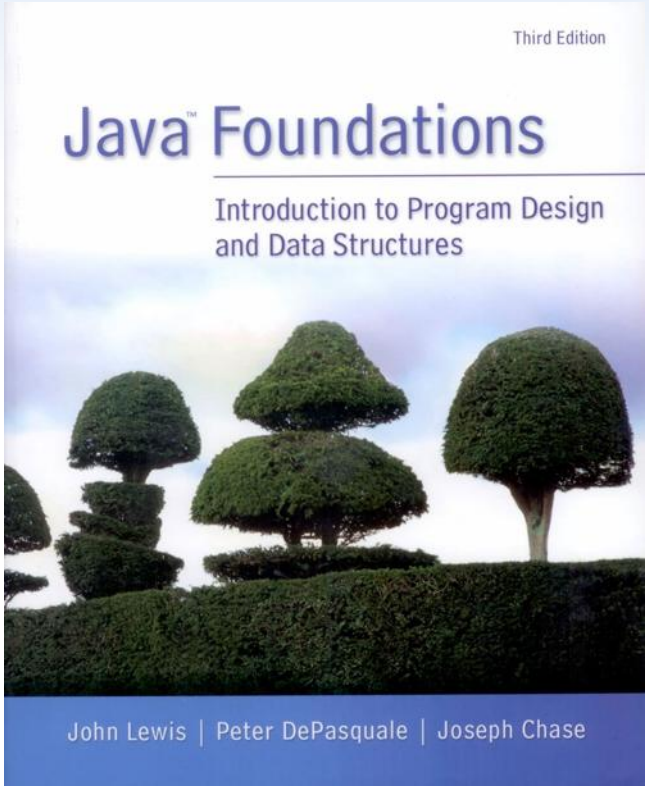


Chapter 20

Binary Search Trees

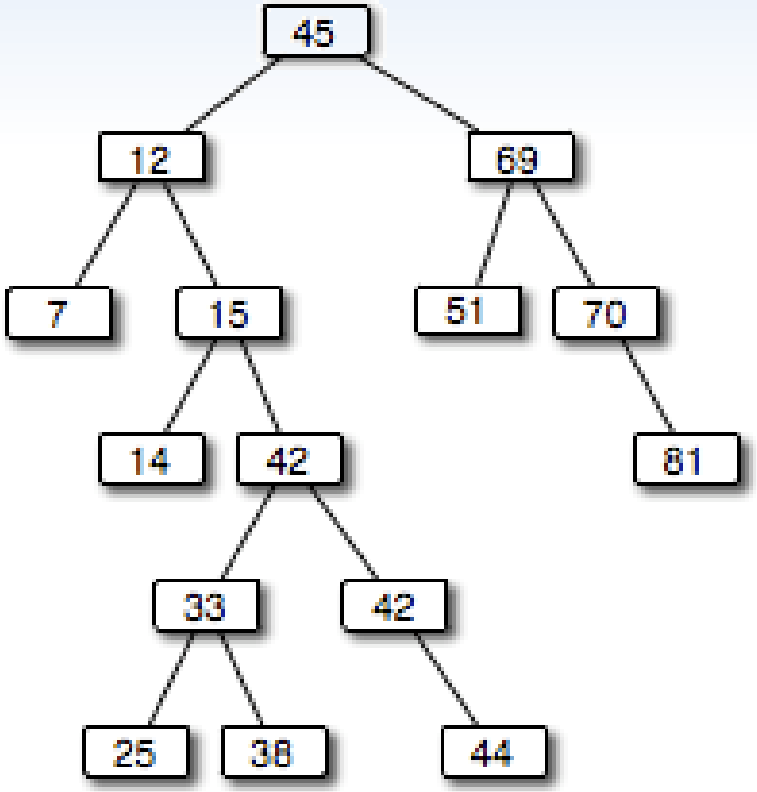
Chapter Scope

- Binary search tree processing
- Using BSTs to solve problems
- BST implementations
- Strategies for balancing BSTs

Binary Search Trees

- A *search tree* is a tree whose elements are organized to facilitate finding a particular element when needed
- A *binary search tree* is a binary tree that, for each node *n*
  - the left subtree of *n* contains elements less than the element stored in *n*
  - the right subtree of *n* contains elements greater than or equal to the element stored in *n*

Binary Search Trees



Binary Search Trees

- To determine if a particular value exists in a tree
  - start at the root
  - compare target to element at current node
  - move left from current node if target is less than element in the current node
  - move right from current node if target is greater than element in the current node
- We eventually find the target or encounter the end of a path (target is not found)

Binary Search Trees

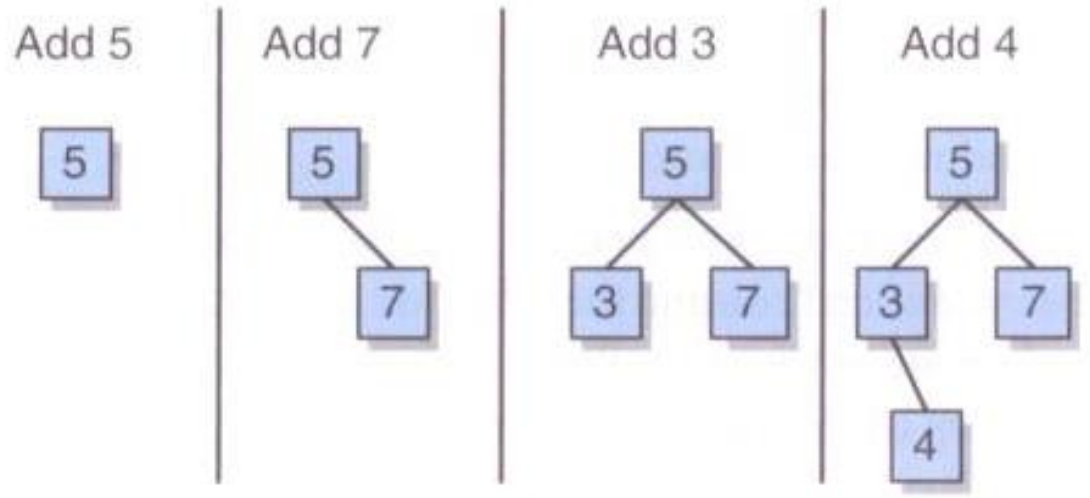
- The particular shape of a binary search tree depends on the order in which the elements are added
- The shape may also be dependant on any additional processing performed on the tree to reshape it
- Binary search trees can hold any type of data, so long as we have a way to determine relative ordering
- Objects implementing the `Comparable` interface provide such capability

Binary Search Trees

- Process of adding an element is similar to finding an element
- New elements are added as leaf nodes
- Start at the root, follow path dictated by existing elements until you find no child in the desired direction
- Then add the new element

Binary Search Trees

- Adding elements to a BST:



Binary Search Trees

- BST operations:

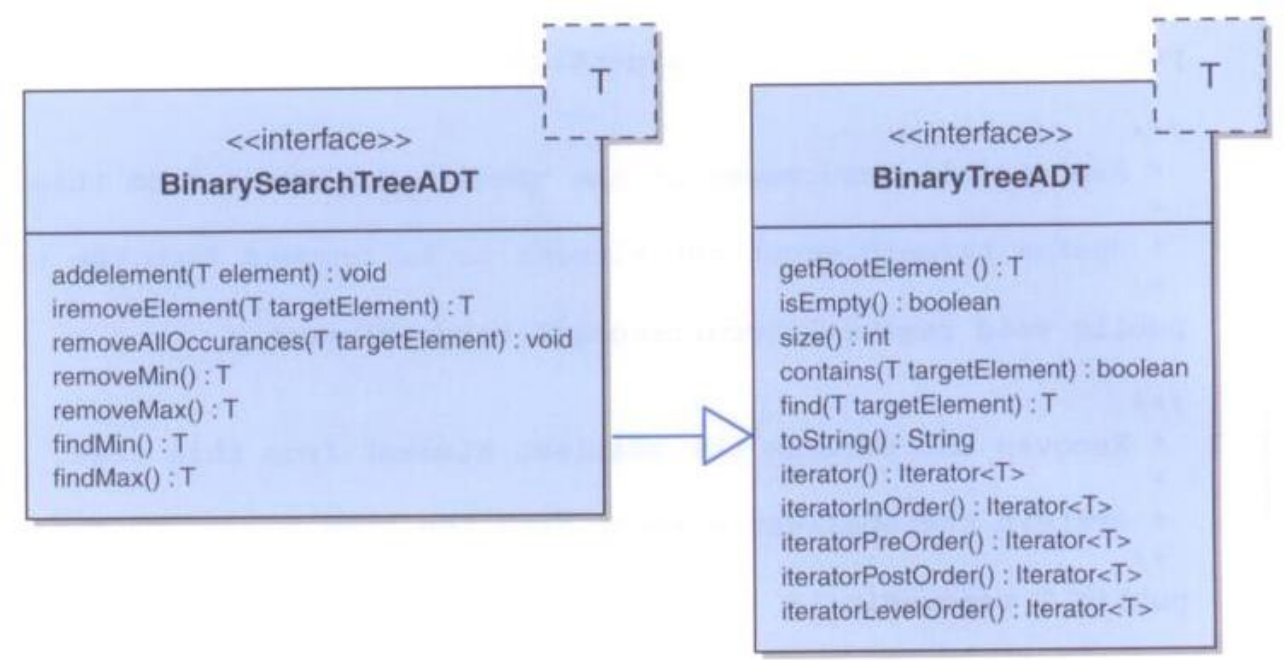
Operation	Description
`addElement`	Add an element to the tree.
`removeElement`	Remove an element from the tree.
`removeAllOccurrences`	Remove all occurrences of element from the tree.
`removeMin`	Remove the minimum element in the tree.
`removeMax`	Remove the maximum element in the tree.
`findMin`	Returns a reference to the minimum element in the tree.
`findMax`	Returns a reference to the maximum element in the tree.

```
package jsjf;

/**
 * BinarySearchTreeADT defines the interface to a binary search tree.
 *
 * @author Java Foundations
 * @version 4.0
 */
public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T>
{
    /**
     * Adds the specified element to the proper location in this tree.
     *
     * @param element the element to be added to this tree
     */
    public void addElement(T element);

    /**
     * Removes and returns the specified element from this tree.
     *
     * @param targetElement the element to be removed from the tree
     * @return the element to be removed from the tree
     */
    public T removeElement(T targetElement);
```

```
/**
 * Removes all occurences of the specified element from this tree.
 *
 * @param targetElement the element to be removed from the tree
 */
public void removeAllOccurrences(T targetElement);

/**
 * Removes and returns the smallest element from this tree.
 *
 * @return the smallest element from the tree.
 */
public T removeMin();

/**
 * Removes and returns the largest element from this tree.
 *
 * @return the largest element from the tree
 */
public T removeMax();
```

```
    /**
     * Returns the smallest element in this tree without removing it.
     *
     * @return the smallest element in the tree
     */
    public T findMin();

    /**
     * Returns the largest element in this tree without removing it.
     *
     * @return the largest element in the tree
     */
    public T findMax();
}
```

Binary Search Trees



```
package jsjf;

import jsjf.exceptions.*;
import jsjf.*;

/**
 * LinkedBinarySearchTree implements the BinarySearchTreeADT interface
 * with links.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class LinkedBinarySearchTree<T> extends LinkedBinaryTree<T>
                                       implements BinarySearchTreeADT<T>
{
    /**
     * Creates an empty binary search tree.
     */
    public LinkedBinarySearchTree()
    {
        super();
    }
```

```
/**
 * Creates a binary search with the specified element as its root.
 *
 * @param element the element that will be the root of the new binary
 *        search tree
 */
public LinkedBinarySearchTree(T element)
{
    super(element);

    if (!(element instanceof Comparable))
        throw new NonComparableElementException("LinkedBinarySearchTree");
}
```

```
/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its natural order.  Note that
 * equal elements are added to the right.
 *
 * @param element the element to be added to the binary search tree
 */
public void addElement(T element)
{
    if (!(element instanceof Comparable))
        throw new NonComparableElementException("LinkedBinarySearchTree");

    Comparable<T> comparableElement = (Comparable<T>)element;

    if (isEmpty())
        root = new BinaryTreeNode<T>(element);
    else
    {
        if (comparableElement.compareTo(root.getElement()) < 0)
        {
            if (root.getLeft() == null)
                this.getRootNode().setLeft(new BinaryTreeNode<T>(element));
            else
                addElement(element, root.getLeft());
        }
```

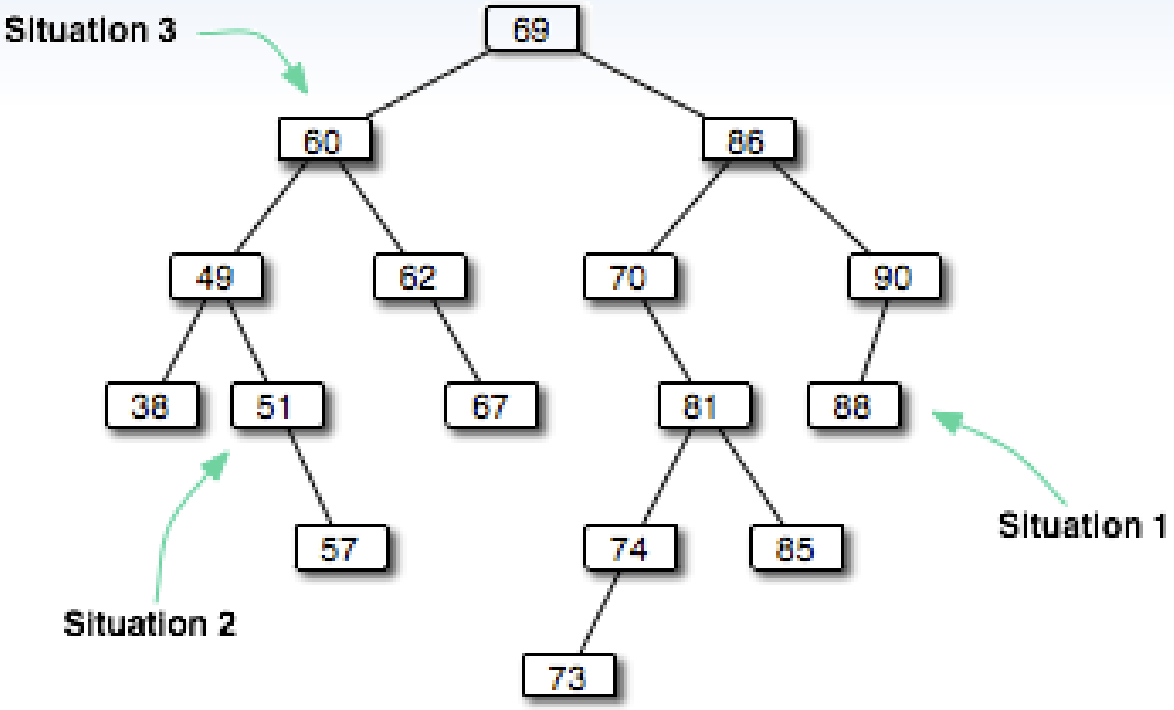
```
            else
            {
                if (root.getRight() == null)
                    this.getRootNode().setRight(new BinaryTreeNode<T>(element));
                else
                    addElement(element, root.getRight());
            }
        }
        modCount++;
    }
```

```
/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its natural order.  Note that
 * equal elements are added to the right.
 *
 * @param element the element to be added to the binary search tree
 */
private void addElement(T element, BinaryTreeNode<T> node)
{
    Comparable<T> comparableElement = (Comparable<T>)element;

    if (comparableElement.compareTo(node.getElement()) < 0)
    {
        if (node.getLeft() == null)
            node.setLeft(new BinaryTreeNode<T>(element));
        else
            addElement(element, node.getLeft());
    }
    else
    {
        if (node.getRight() == null)
            node.setRight(new BinaryTreeNode<T>(element));
        else
            addElement(element, node.getRight());
    }
}
```

BST Element Removal

- Removing a target in a BST is not as simple as that for linear data structures
- After removing the element, the resulting tree must still be valid
- Three distinct situations must be considered when removing an element
  - The node to remove is a leaf
  - The node to remove has one child
  - The node to remove has two children

BST Element Removal



BST Element Removal

- Dealing with the situations
  - Node is a leaf: it can simply be deleted
  - Node has one child: the deleted node is replaced by the child
  - Node has two children: an appropriate node is found lower in the tree and used to replace the node
    - Good choice: *inorder successor* (node that would follow the removed node in an inorder traversal)
    - The inorder successor is guaranteed not to have a left child
    - Thus, removing the inorder successor to replace the deleted node will result in one of the first two situations (it's a leaf or has one child)

```
/**
 * Removes the first element that matches the specified target
 * element from the binary search tree and returns a reference to
 * it.  Throws a ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @throws ElementNotFoundException if the target element is not found
 */
public T removeElement(T targetElement)
                                throws ElementNotFoundException
{
    T result = null;

    if (isEmpty())
        throw new ElementNotFoundException("LinkedBinarySearchTree");
    else
    {
        BinaryTreeNode<T> parent = null;
        if (((Comparable<T>)targetElement).equals(root.element))
        {
            result =  root.element;
            BinaryTreeNode<T> temp = replacement(root);
            if (temp == null)
                root = null;
```

```
            else
            {
                root.element = temp.element;
                root.setRight(temp.right);
                root.setLeft(temp.left);
            }

            modCount--;
        }
        else
        {
            parent = root;
            if (((Comparable)targetElement).compareTo(root.element) < 0)
                result = removeElement(targetElement, root.getLeft(), parent);
            else
                result = removeElement(targetElement, root.getRight(), parent);
        }
    }

    return result;
}
```

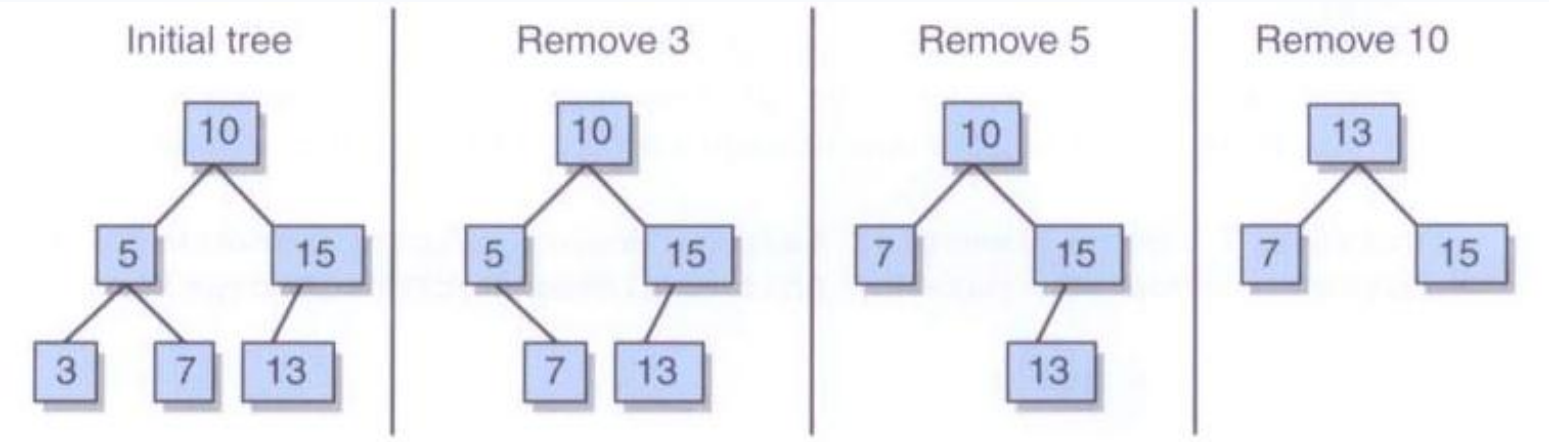
```
/**
 * Removes the first element that matches the specified target
 * element from the binary search tree and returns a reference to
 * it.  Throws a ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @param node the node from which to search
 * @param parent the parent of the node from which to search
 * @throws ElementNotFoundException if the target element is not found
 */
private T removeElement(T targetElement, BinaryTreeNode<T> node, BinaryTreeNode<T>
parent)
throws ElementNotFoundException
{
    T result = null;

    if (node == null)
        throw new ElementNotFoundException("LinkedBinarySearchTree");
    else
    {
        if (((Comparable<T>)targetElement).equals(node.element))
        {
            result =  node.element;
            BinaryTreeNode<T> temp = replacement(node);
            if (parent.right == node)
                parent.right = temp;
```

```
                else
                    parent.left = temp;

                modCount--;
            }
            else
            {
                parent = node;
                if (((Comparable)targetElement).compareTo(node.element) < 0)
                    result = removeElement(targetElement, node.getLeft(), parent);
                else
                    result = removeElement(targetElement, node.getRight(), parent);
            }
        }

        return result;
    }
```

```
/**
 * Returns a reference to a node that will replace the one
 * specified for removal.  In the case where the removed node has
 * two children, the inorder successor is used as its replacement.
 *
 * @param node the node to be removed
 * @return a reference to the replacing node
 */
private BinaryTreeNode<T> replacement(BinaryTreeNode<T> node)
{
    BinaryTreeNode<T> result = null;

    if ((node.left == null) && (node.right == null))
        result = null;

    else if ((node.left != null) && (node.right == null))
        result = node.left;

    else if ((node.left == null) && (node.right != null))
        result = node.right;

    else
    {
        BinaryTreeNode<T> current = node.right;
        BinaryTreeNode<T> parent = node;
```

```
            while (current.left != null)
            {
                parent = current;
                current = current.left;
            }

            current.left = node.left;
            if (node.right != current)
            {
                parent.left = current.right;
                current.right = node.right;
            }

            result = current;
        }

        return result;
    }
```

BST Element Removal



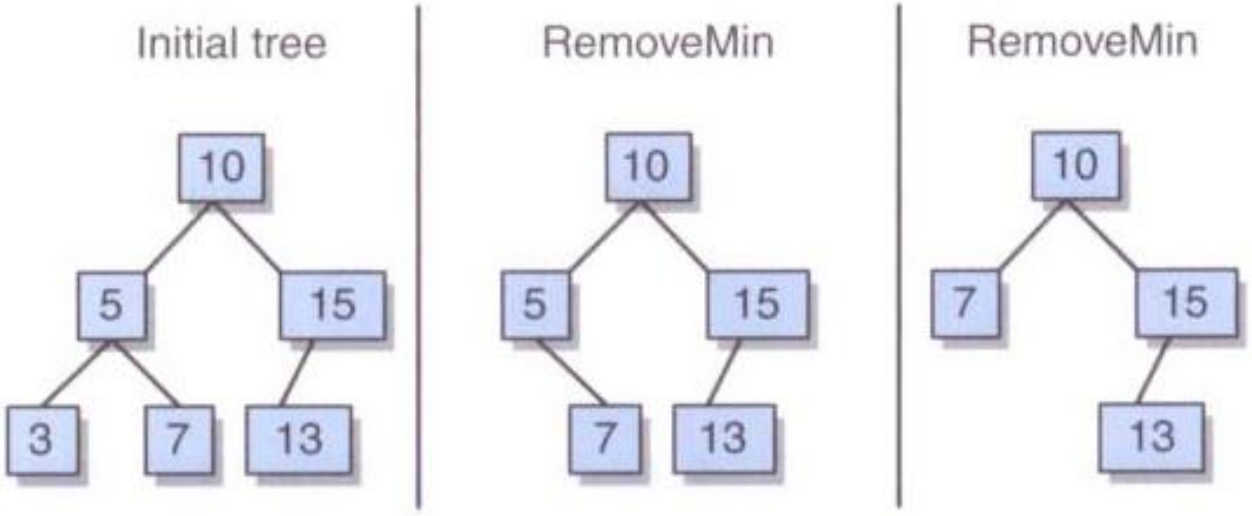
```
/**
 * Removes elements that match the specified target element from
 * the binary search tree. Throws a ElementNotFoundException if
 * the sepcified target element is not found in this tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @throws ElementNotFoundException if the target element is not found
 */
public void removeAllOccurrences(T targetElement)
                throws ElementNotFoundException
{
    removeElement(targetElement);

    try
    {
        while (contains((T)targetElement))
            removeElement(targetElement);
    }

    catch (Exception ElementNotFoundException)
    {
    }
}
```

```
/**
 * Removes the node with the least value from the binary search
 * tree and returns a reference to its element.  Throws an
 * EmptyCollectionException if this tree is empty.
 *
 * @return a reference to the node with the least value
 * @throws EmptyCollectionException if the tree is empty
 */
public T removeMin() throws EmptyCollectionException
{
    T result = null;

    if (isEmpty())
        throw new EmptyCollectionException("LinkedBinarySearchTree");
    else
    {
        if (root.left == null)
        {
            result = root.element;
            root = root.right;
        }
        else
        {
            BinaryTreeNode<T> parent = root;
            BinaryTreeNode<T> current = root.left;
```

```
            while (current.left != null)
            {
                parent = current;
                current = current.left;
            }
            result =  current.element;
            parent.left = current.right;
        }

        modCount--;
    }

    return result;
}
```

BST Element Removal

- Removing the minimum element:



Using BSTs to Implement Ordered Lists

- Recall the list operations:

Operation	Description
`removeFirst`	Removes the first element from the list.
`removeLast`	Removes the last element from the list.
`remove`	Removes a particular element from the list.
`first`	Examines the element at the front of the list.
`last`	Examines the element at the rear of the list.
`contains`	Determines if the list contains a particular element.
`isEmpty`	Determines if the list is empty.
`size`	Determines the number of elements on the list.

- And specifically for ordered lists:

Operation	Description
`add`	Adds an element to the list.

```
package jsjf;

import jsjf.exceptions.*;
import java.util.Iterator;

/**
 * BinarySearchTreeList represents an ordered list implemented using a binary
 * search tree.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class BinarySearchTreeList<T> extends LinkedBinarySearchTree<T>
                      implements ListADT<T>, OrderedListADT<T>, Iterable<T>
{
    /**
     * Creates an empty BinarySearchTreeList.
     */
    public BinarySearchTreeList()
    {
        super();
    }

    /**
     * Adds the given element to this list.
     *
     * @param element the element to be added to the list
     */
    public void add(T element)
    {
        addElement(element);
    }
```

```
    /**
     * Removes and returns the first element from this list.
     *
     * @return the first element in the list
     */
    public T removeFirst()
    {
        return removeMin();
    }

    /**
     * Removes and returns the last element from this list.
     *
     * @return the last element from the list
     */
    public T removeLast()
    {
        return removeMax();
    }

   /**
    * Removes and returns the specified element from this list.
    *
    * @param element the element being sought in the list
    * @return the element from the list that matches the target
    */
    public T remove(T element)
    {
        return removeElement(element);
    }
```

```
    /**
     * Returns a reference to the first element on this list.
     *
     * @return a reference to the first element in the list
     */
     public T first()
     {
         return findMin();
     }

    /**
     * Returns a reference to the last element on this list.
     *
     * @return a reference to the last element in the list
     */
     public T last()
     {
         return findMax();
     }

    /**
     * Returns an iterator for the list.
     *
     * @return an iterator over the elements in the list
     */
     public Iterator<T> iterator()
     {
         return iteratorInOrder();
     }
}
```

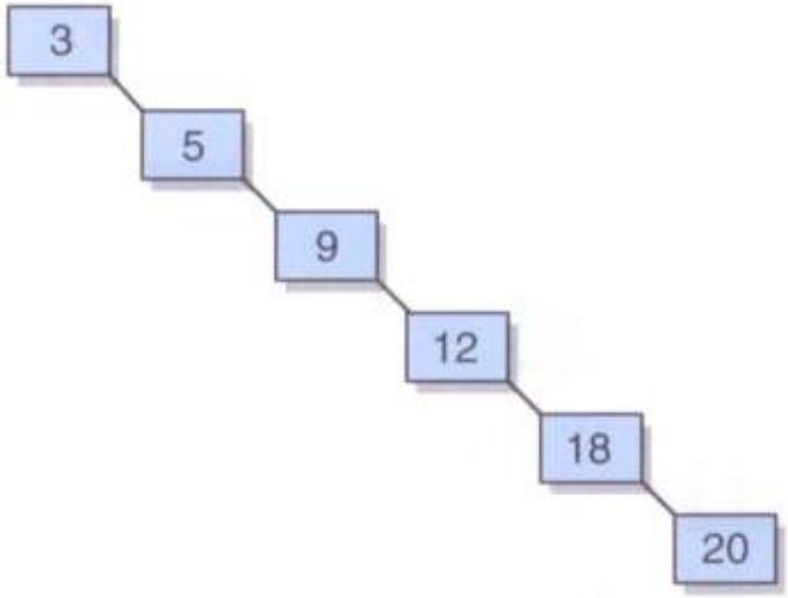
Ordered List Analysis

- Comparing operations for both implementations of ordered lists:

Operation	LinkedList	BinarySearchTreeList
`removeFirst`	O(1)	O(log n)
`removeLast`	O(n)	O(log n)
`remove`	O(n)	O(log n)*
`first`	O(1)	O(log n)
`last`	O(n)	O(log n)
`contains`	O(n)	O(log n)
`isEmpty`	O(1)	O(1)
`size`	O(1)	O(1)
`add`	O(n)	O(log n)*
*both the `add` and `remove` operations may cause the tree to become unbalanced		

Balancing BSTs

- As operations are performed on a BST, it could become highly unbalanced (a *degenerate tree*)



Balancing BSTs

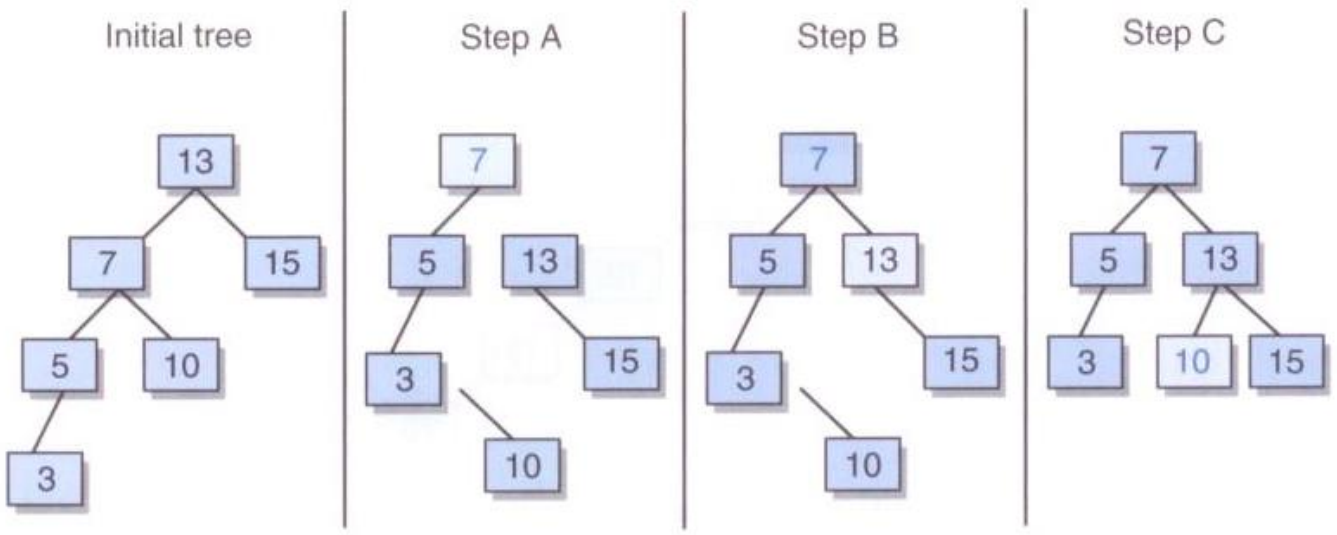
- Our implementation does not ensure the BST stays balanced
- Other approaches do, such as AVL trees and red/black trees
- We will explore *rotations* – operations on binary search trees to assist in the process of keeping a tree balanced
- Rotations do not solve all problems created by unbalanced trees, but show the basic algorithmic processes that are used to manipulate trees

Balancing BSTs

- A *right rotation* can be performed at any level of a tree, around the root of any subtree
- Corrects an imbalance caused by a long path in the left subtree of the left child of the root
- To correct the imbalance
  - make the left child element of the root the new root element
  - make the former root element the right child element of the new root
  - make the right child of what was the left child of the former root the new left child of the former root

Balancing BSTs

- A right rotation:

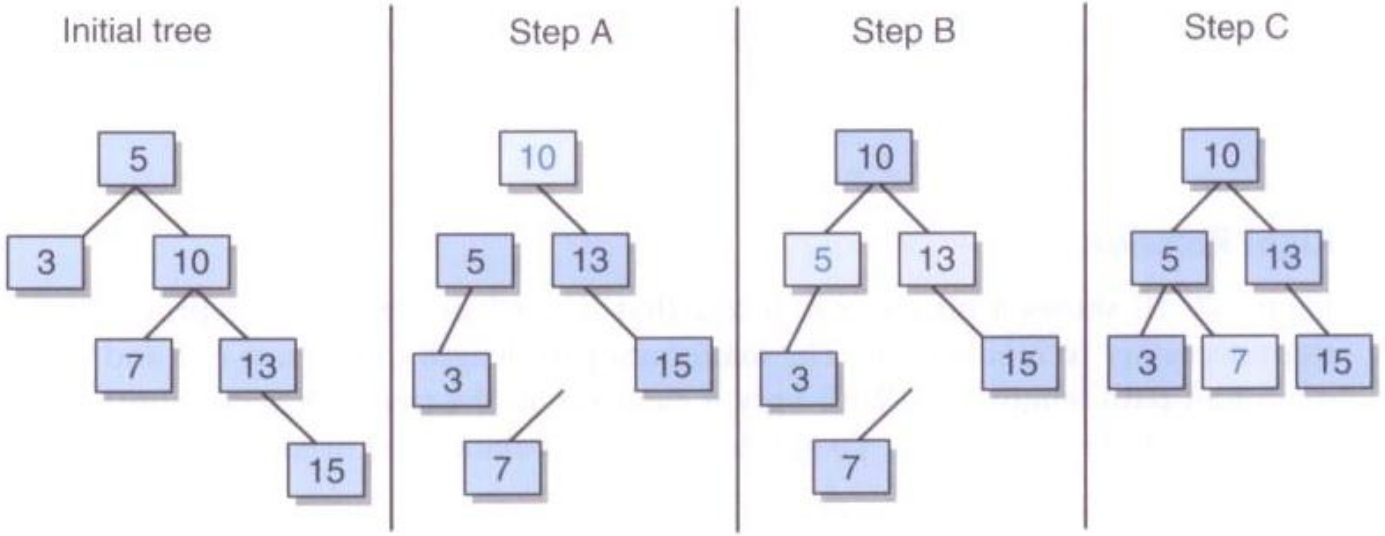


Balancing BSTs

- A *left rotation* can be performed at any level of a tree, around the root of any subtree
- Corrects an imbalance caused by a long path in the right subtree of the left child of the root
- To correct the imbalance
  - make the right child element of the root the new root element
  - make the former root element the left child element of the new root
  - make the left child of what was the right child of the former root the new right child of the former root

Balancing BSTs

- A left rotation:

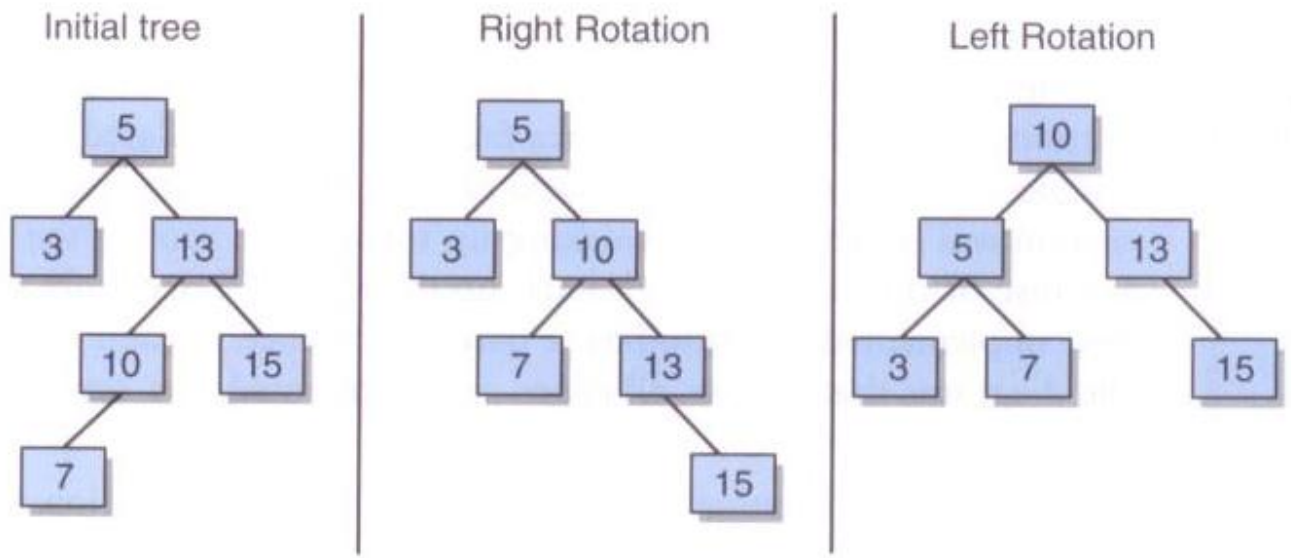


Balancing BSTs

- If the imbalance is caused by a long path in the left subtree of the right child of the root we can address it by performing a *rightleft rotation:*
  - performing a right rotation around the offending subtree
  - and then performing a left rotation around the root

Balancing BSTs

- A rightleft rotation:

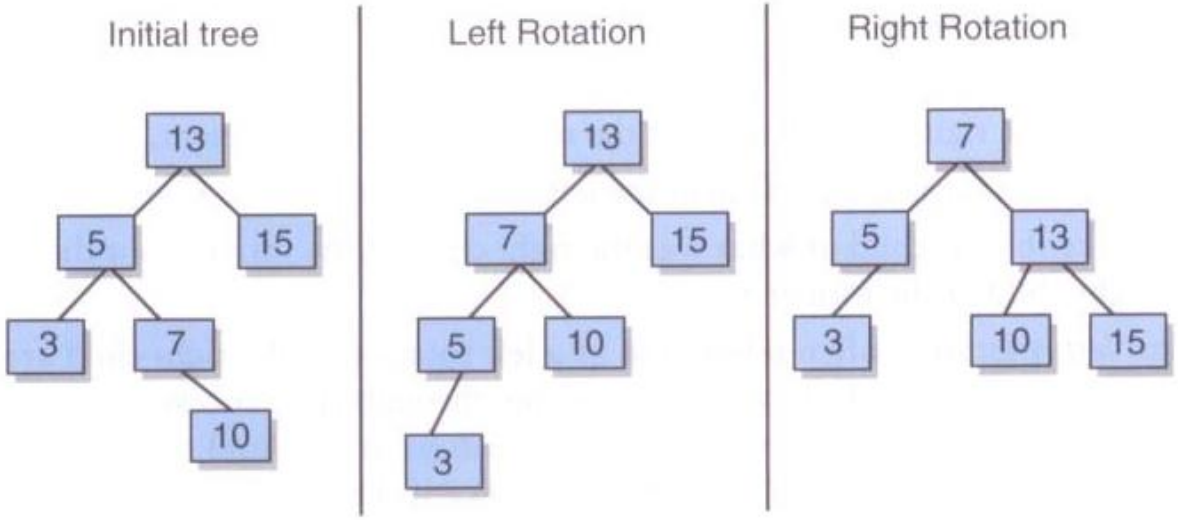


Balancing BSTs

- If the imbalance is caused by a long path in the right subtree of the left child of the root we can address it by performing a *leftright rotation*:
  - performing a left rotation around the offending subtree
  - and then performing a right rotation around the root

Balancing BSTs

- A leftright rotation:

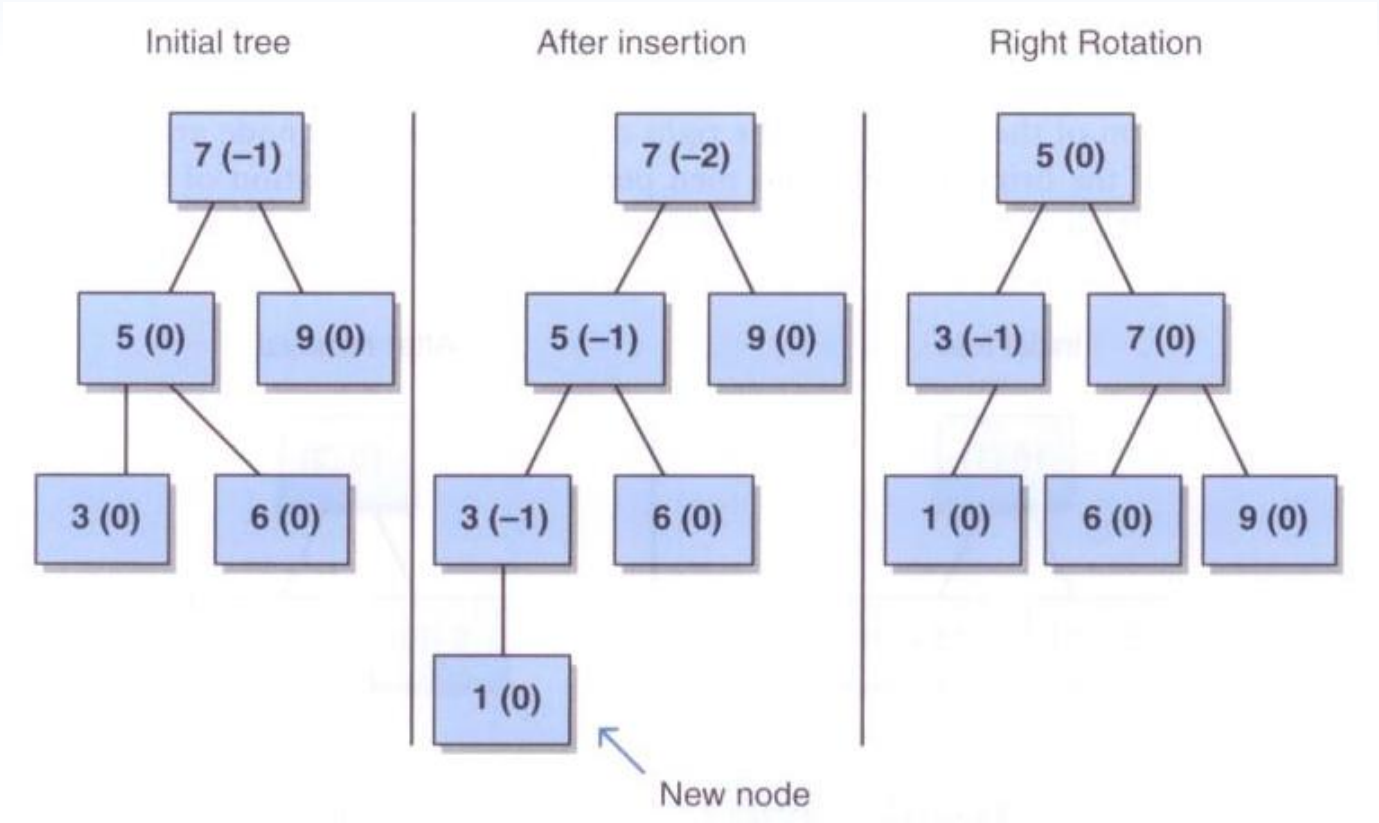


AVL Trees

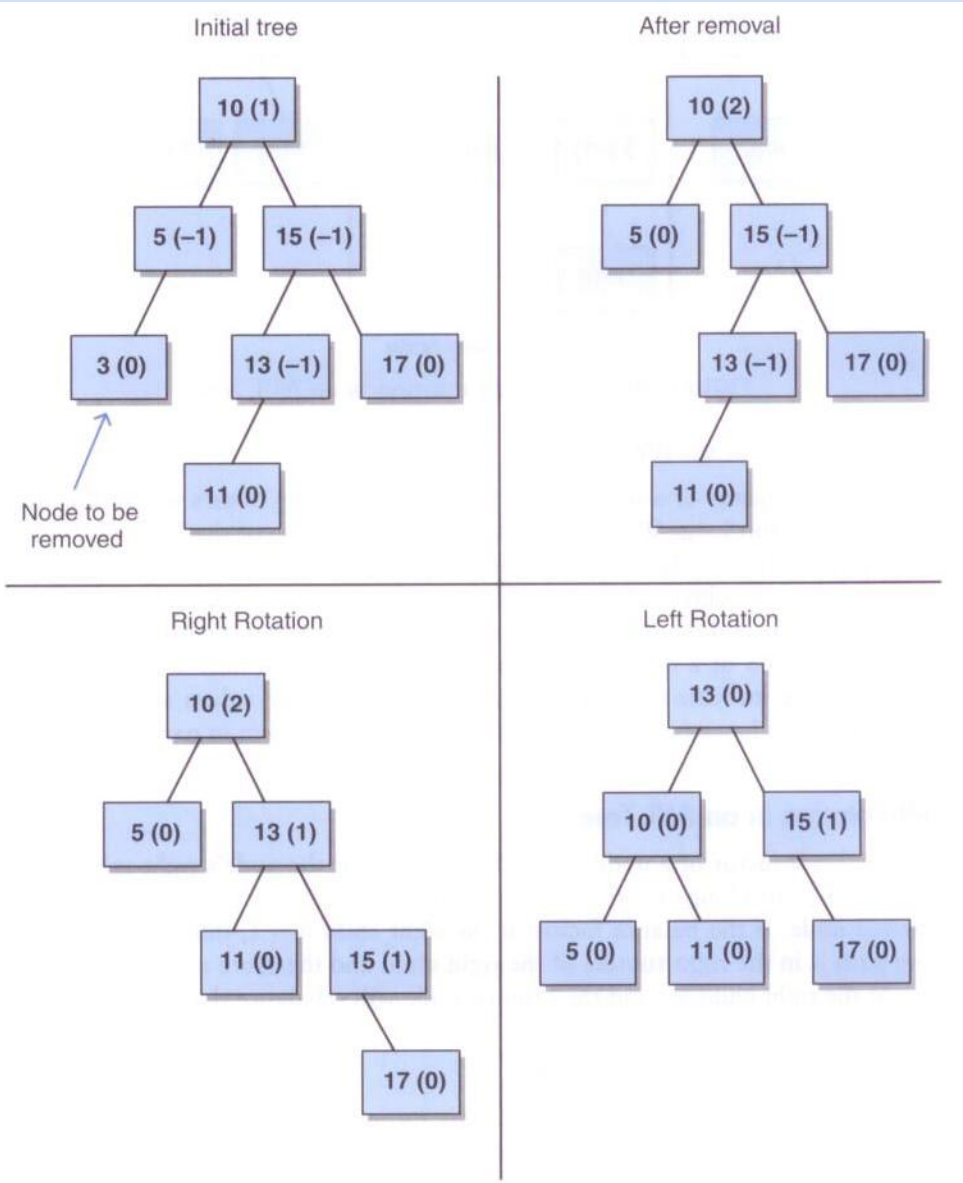
- An AVL tree (named after the creators) ensures a BST stays balanced
- For each node in the tree, there is a numeric *balance factor* – the difference between the heights of its subtrees
- After each add or removal, the balance factors are checked, and rotations performed as needed

AVL Trees

- A right rotation in an AVL tree:



- A rightleft rotation in an ALV tree:

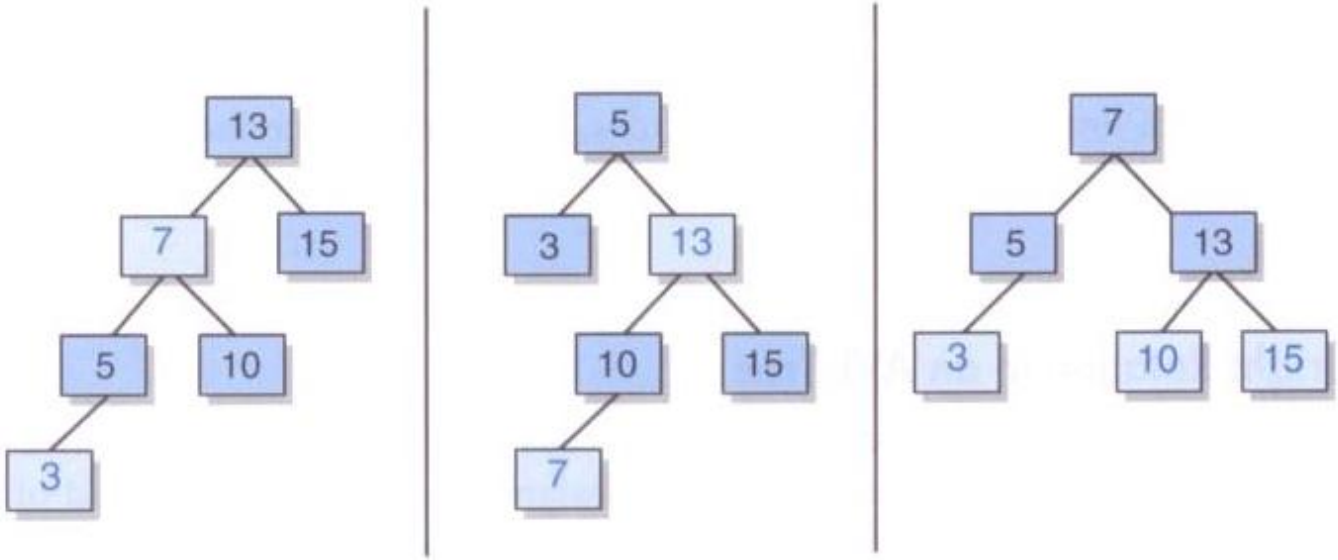


Red/Black Trees

- Another balanced BST approach is a red/black tree
- Each node has a color, usually implemented as a boolean value
- The following rules govern the color of a node:
  - the root is black
  - all children of red nodes are black
  - every path from the root to a leaf contains the same number of black nodes

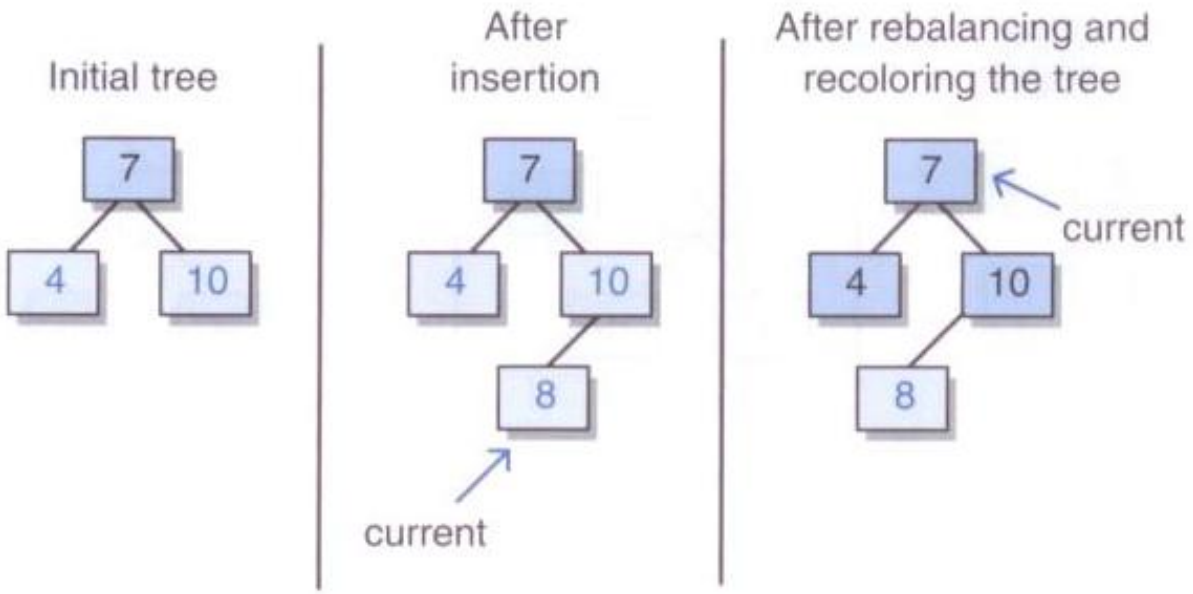
Red/Black Trees

- Examples of red/black trees (light shading = red)



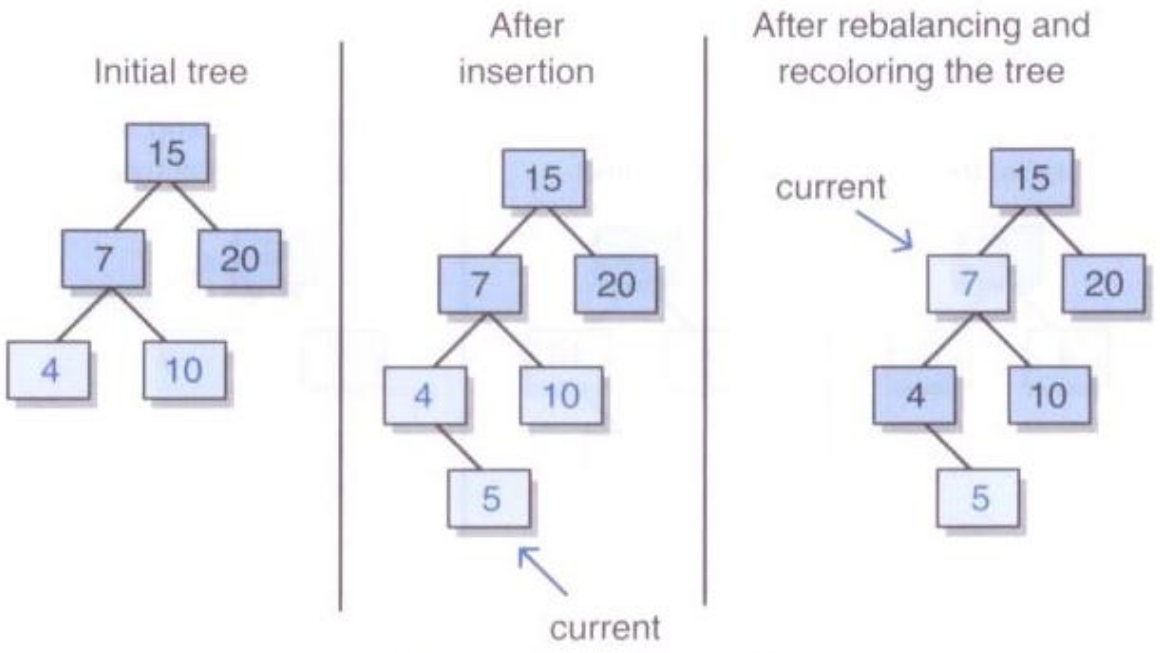
Red/Black Trees

- A red/black tree after an insertion:



Red/Black Trees

- A red/black tree after insertion:



Red/Black Trees

- A red/black tree removal: