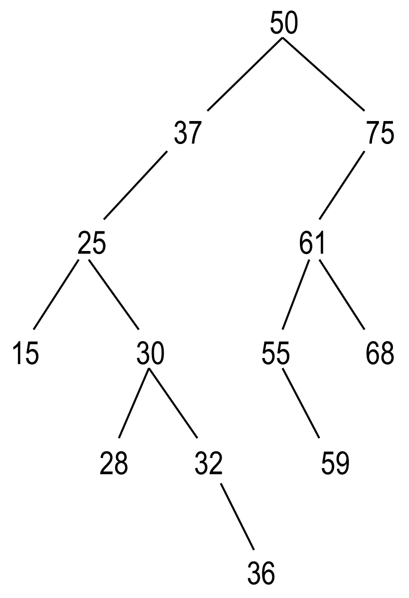
# Chapter 10 Binary Search Trees

- The BinarySearchTree class is not part of the Java Collection Framework. However, framework already includes the TreeSet class, which has a logarithmic time for inserting, removing and searching even in worst case.
- The BinarySearchTree class should be viewed as a “toy” class and it is used for the purpose of better understanding TreeSet class.
- The BinarySearchTree class requires linear time in n for inserting, removing, and searching.
- The implementation of the TreeSet class is based on a kind of “balanced” binary search tree, namely, the red-black tree.

## 10.1 Binary Search Tree

- Recursive definition of a binary search tree:

  A binary search tree t is a binary tree such that either t is empty or:
  1. Each element in **leftTree(t)** is less than the root element of **t**;
  2. Each element in **rightTree(t)** is greater than the root element of **t**;
  3. Both **leftTree(t)** and **rightTree(t)** are binary search trees.



- As we have defined a binary search tree, duplicates are not permitted.
- inOrder trasversal of the above binary search tree will produce following sequence:
  15 25 28 30 32 36 37 50 55 59 61 68 75
- A binary search tree need not be full, complete or a two-tree, but it could be any of those.
- If a binary search tree is full or complete, its height is logarithmic in n.

### 10.1.1 The BinarySearchTree Implements of the Set Interface

- The BinarySearchTree class is not the part of the Java Collection Framework. However, we will study binary search tree data type through the method specification of this class.
- The BinarySearchTree implements Set Interface, which slightly extends Collection interface.
- The Set interface does not provide any new methods, the only difference between Collection and Set interface is that duplicate elements are not allowed in a Set interface.
- The BinarySearchTree class has different specifications for default constructor and add method.
- Also, BinarySearchTree class must maintain order of its elements. Thus, BinarySearchTree class implements Comparable interface.
  Page 403. Comparable interface
  Page 406. A UML diagram that includes part of the BinarySearchTree class.

### 10.1.2 Implementation of the BinarySearchTree class

- The only methods we need to implement are seven methods:
  - **public** BinarySearchTree()
  - **public** BinarySearchTree( BinarySearchTree <? Extends E> otherTree)
  - **public int** size()
  - **public** Iterator <E> iterator()
  - **public boolean** contains (Object obj)
  - **public boolean** add( E element)
  - **public boolean** remove ( Object obj)

- However, in order to implement iterator method, we will have to develop a class that implements the Iterator interface, with hasNext(), next(), and remove() methods.
- TreeIterator class will be embedded in BinarySearchTree class.
- The definitions of default constructor, size(), iterator, and hasNext() are one-linear.
- However, methods contains, add, and next will differ from those method definitions we studied in chapter 7, LinkedList class.

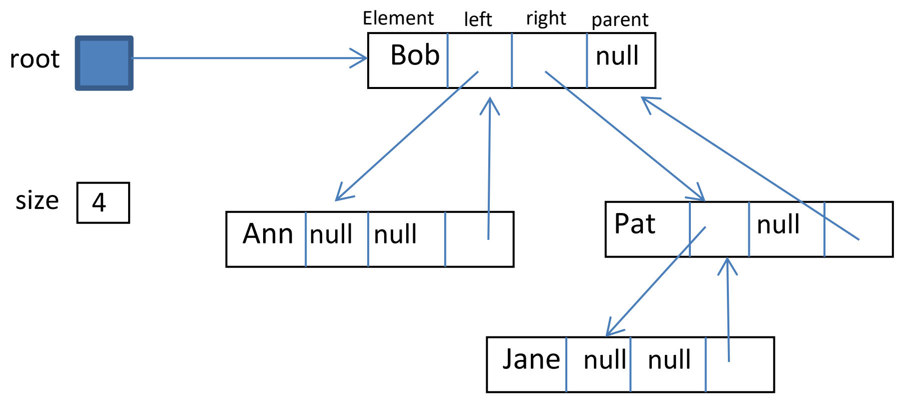
#### 10.1.2.1 Fields and Nested Classes in the BinarySearchTree Class

- Some of the embedded classes that would be used in the BinarySearchTree class will be TreeIterator class and Entry class.
- The only fields that BinarySearchTree class will have are:

  **protected** Entry <E> root;
  **protected int** size;

- The Entry class that we will use in BinarySearchTree class will have four fields:
- An **element** field of type (reference to) **E**
- **left** field and **right** field of type (reference to) **Entry <E>**

- **parent** filed of type (reference to) **Entry<E>** that would facilitate going back up the tree during an iteration.



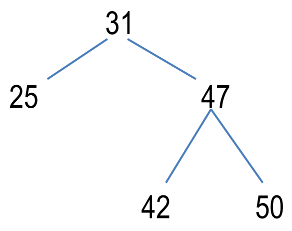
Here is the nested Entry class:

```
protected static class Entry<E>
{
        protected E element;

        protected Entry <E> left = null,
                                right = null,
                                parent;

        public Entry()
        {

        }

        public Entry (E element, Entry <E> parent)
        {
                this.element = element;
                this.parent = parent;
        }
}
```

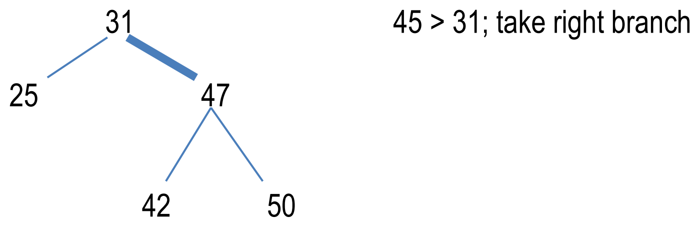
### 10.1.2.4 Definition of the add Method

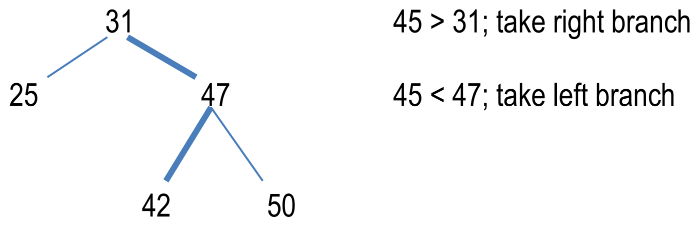
- The add method starts at the root and branches down the tree searching for the element; if the search fails, the element is inserted as a leaf.

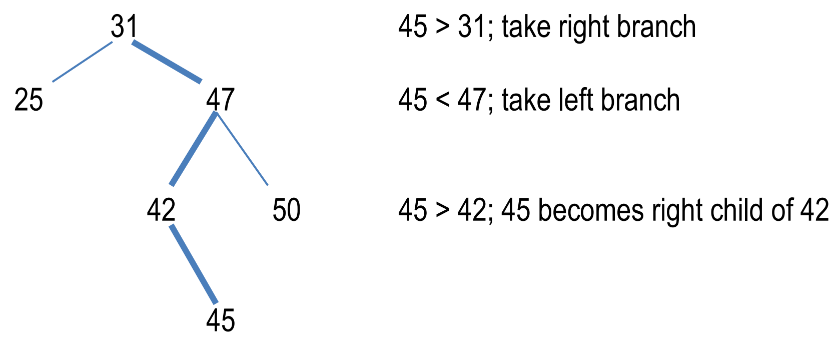
- For example inserting element 45 into the binary search tree:



- The insertion is made in a loop that starts by comparing 45 and 31, the root element.
- Since 45 > 31, we advance to 47, the right child of 31.
- Because 45 < 47, we advance to 42, the left child of 47.
- Then, 45 > 42 so we would advance to the right child of 42 if 42 had a right child.
- It does not, so 45 is inserted as the right child to 42.






31
25
47
42
50
45
45 > 31; take right branch
45 < 47; take left branch
45 > 42; 45 becomes right child of 42

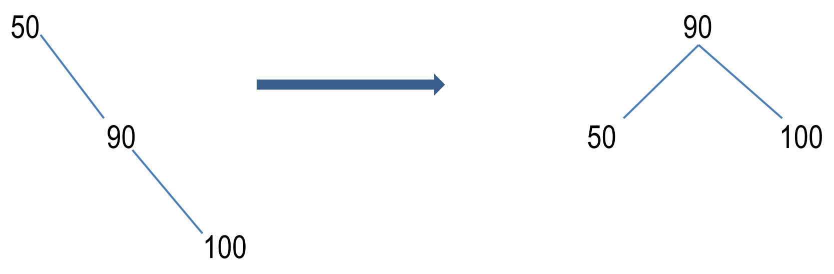
- The inserted element always becomes a leaf in the tree. The advantage of this is that the tree is not re-organized after an insertion.
- Definition of one-parameter add method:

```
public boolean add (E element)
{
     if (root == null)
    {
        root = new Entry (element, null);
        size++;
        return true;
     } // empty tree
     else
    {
        Entry<E> temp = root;

        int comp;

          while (true)
          {
                comp =  ((Comparable)element).compareTo (temp.element);
           if (comp == 0)
                       return false;
            if (comp < 0)
                              if (temp.left != null)
                                      temp = temp.left;
                              else
                               {
                                        temp.left = new Entry<E> (element, temp);
                                        size++;
                                        return true;
                                } // temp.left == null
             else if (temp.right != null)
               temp = temp.right;
            else { /* Insert as right child and leaf */ }
    } // while
```

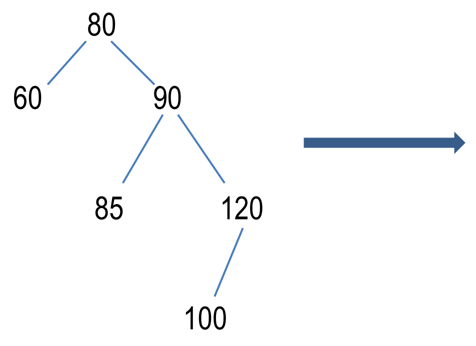
## 10.2 Balanced Binary Search Trees

- Keep in mind that the height of a BinarySearchTree is determined factor in estimating the time to insert, remove or search.
- In the average case, the height of a BinarySearchTree object is logarithmic in n, so inserting, removing and searching takes only logarithmic in n.
- But in the worst case, a BinarySearchTree object ‘s height can be linear in n, which leads to linear in n worstTime(n) for inserting, removing, and searching.
- However, the height of a tree for TreeMap and TreeSet classes is always logarithmic in n, so the worst time for inserting, removing, and searching is also logarithmic in n.
- A binary search tree is **balanced** if its height is logarithmic in n, the number of elements in the tree.
- Three widely known data structures in this category of balanced binary search trees are AVL trees, red-black trees, and splay trees.
- The basic mechanism for binary search tree is rotation.
- A **rotation** is an adjustment to the tree around an element such that the adjustment maintains the required ordering of elements.
- The ultimate goal of rotating is to restore some balance property has temporarily been violated due to the insertion or removal.
- One of those properties is that the height of the left subtree and right subtree of any element should differ by at most 1.
- In a **left rotation**, some adjustments are made to the element’s parent, left subtree and right subtree.
- The main effect is that the element becomes the left child of what had been the element ‘s right child.

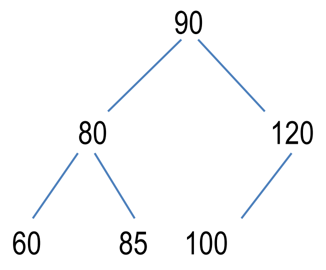
  For example left rotation around element 50.



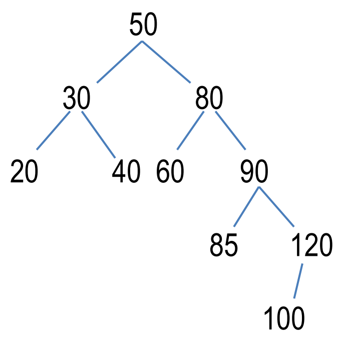
Another example left rotation around element 80, it reduces the height of the tree from 3 to 2.

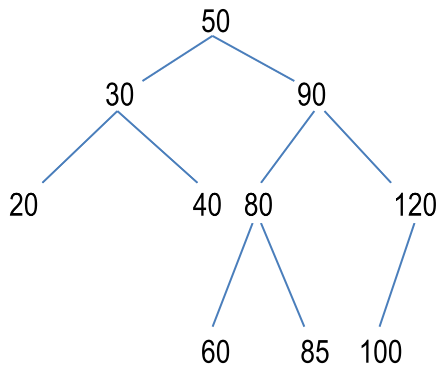


- Note that 85, which was in the right subtree of the before-rotation, ends up in the right subtree of the after-rotation tree.



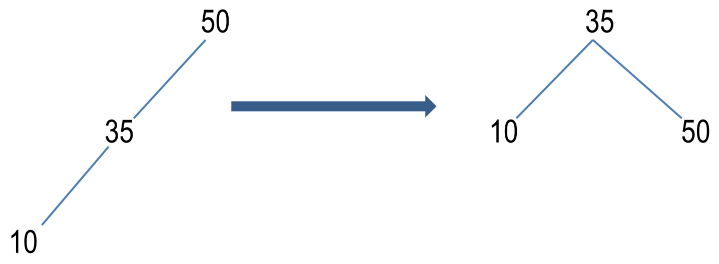
- How about rotation that is not root element?
  Here is another example:



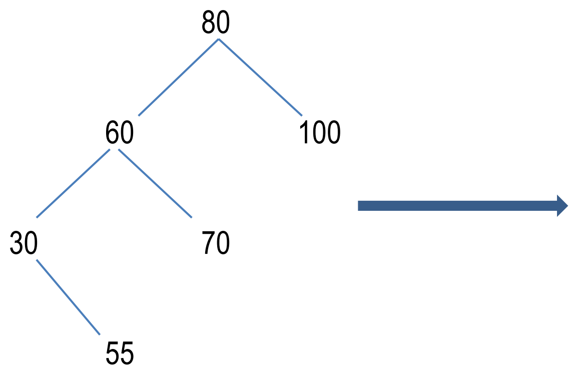


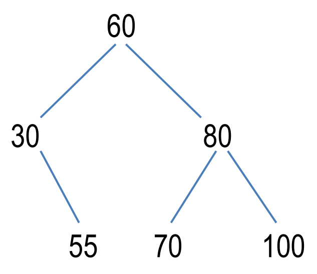
- Another aspect of all rotations: all the elements that are not in the rotated element's subtree are unaffected by the rotation.

  What about right rotation?

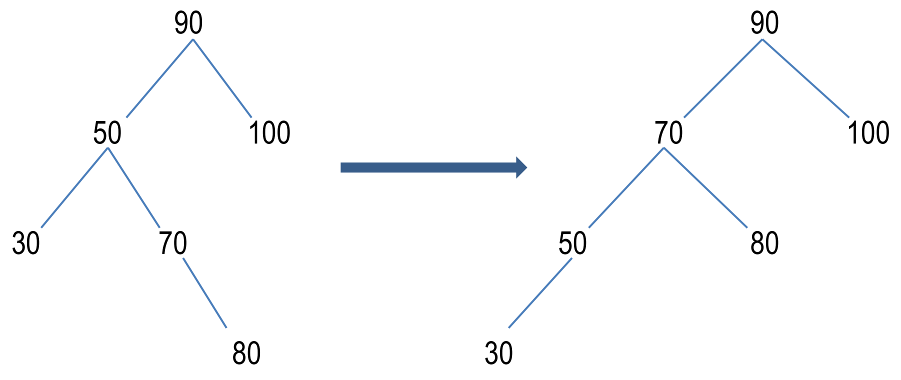


- In general, if you perform a left rotation around an element, and then a right rotation around the new parent, you will end up with the tree you started with.

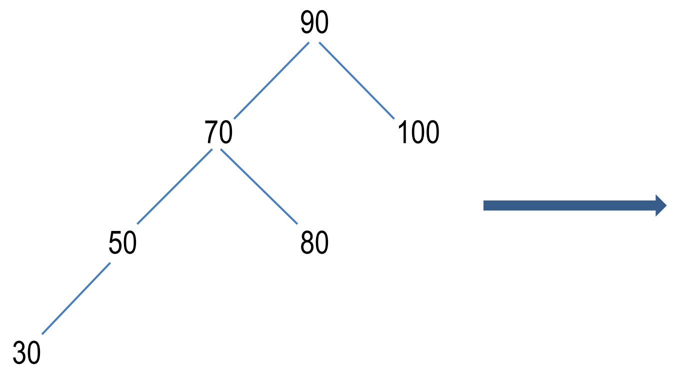


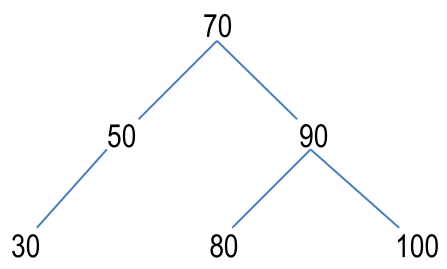


- All rotations we have seen so far, the height of the tree was reduced by 1.
- Actually, this is the main motivation behind rotations.
- However, it is not necessary that every rotation reduce the height of the tree.



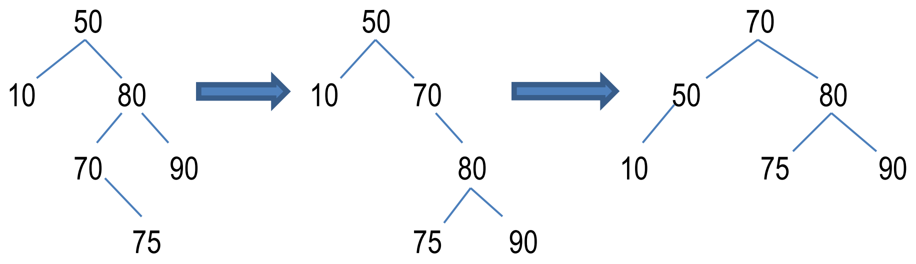
- A left rotation around 50. The height of the tree is still 3 after the rotation.
- Now in order to reduce the height of the tree we will have to perform right rotation around 90.





- The right rotation around 90. The height of the tree has been reduced from 3 to 2.
- This previous example, where we performed left rotation around 50 followed by the right rotation around 90, is called **double rotation**.

- Following is another kind of double rotation: a right rotation around the right child of 50, followed by left rotation around 50.



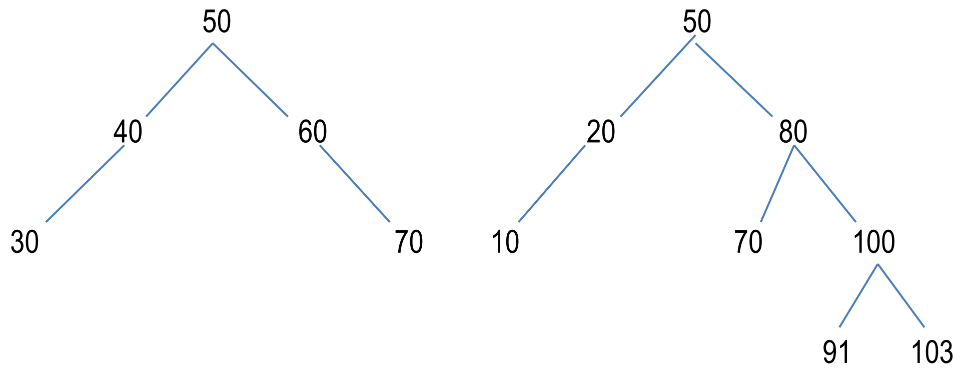
**Major features of rotations**:

- There are four kinds of rotation:
    a. Left rotation
    b. Right rotation
    c. A left rotation around the left child of an element, followed by a right rotation around the element itself.
    d. A right rotation around the right child of an element, followed by a left rotation around the element itself.
- Elements not in the subtree of the element rotated about are unaffected by rotation.
- A rotation takes constant time.
- Before and after a rotation, the tree is still a binary search tree.
- The code for a left rotation is symmetric to the code for a right rotation: simply swap the words “left” and “right”

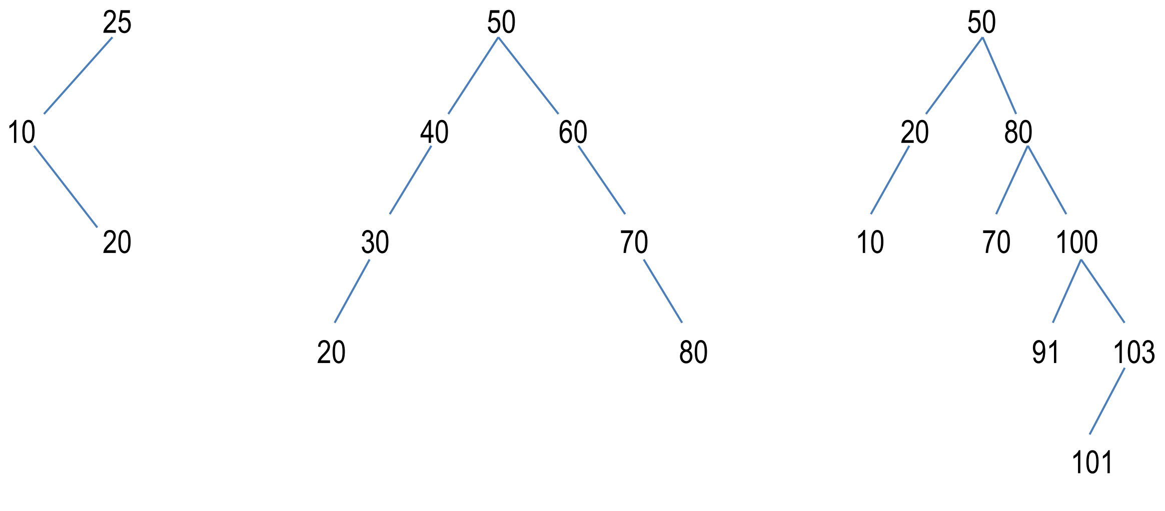
### 10.2.1 AVL Trees

- An AVL tree is a binary search tree that either is empty or in which :
  1. The height of the root's left and right subtrees differ by at most 1
  2. And root's left and right subtrees are AVL trees.

- AVL trees are named after the two Russian mathematicians, Adelson-Velski and Landis, who invented them in 1962.



- Binary search trees that are not AVL trees



- First tree is not AVL because its left subtree has height 1 and right subtree has height of -1
- Second tree is not AVL because left or right subtrees are not AVL trees.
- Third subtree is not AVL because left subtree has height of 1 while right subtree has a height of 3.

**Summary**

- A binary search tree t is a binary tree such that either t is empty or
    a. Each element in leftTree(t) is less than the root element t;
    b. Each element in rightTree(t) is greater than the root element of t;
    c. Both leftTree abd rightTree(t) are binary search trees.

- A balanced search tree is balanced if its height is logarithmic in, the number of elements in the tree.
- The balancing is maintained with rotations.
- A rotation is an adjustment to the tree around an element such that the adjustment maintains the required ordering of elements.
- An AVL tree is a binary search tree that either is empty or in which:
    d. The height of the root's left and right subtrees differ by at most 1
    e. And root's left and right subtrees are AVL trees.