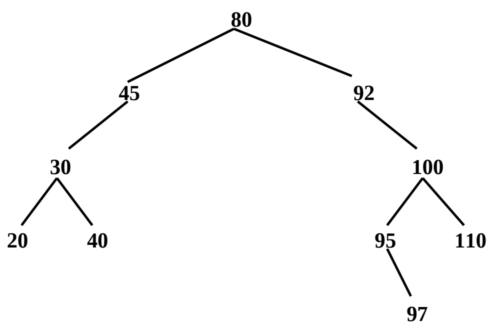
Chapter 10

Binary Search Trees

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**

Here is an example of a binary search tree:



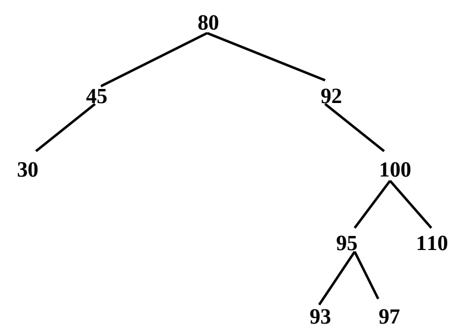
A binary search tree need not be full, complete or a two-tree, but it could be any of those.

If a binary search tree is full or complete, its height is logarithmic in *n*.

If a binary search tree is a chain, its height is linear in *n*.

Even binary search trees that are not chains may have height that is linear in *n*. For example, suppose there are exactly two elements at level 1, level 2, … .

See the following tree:



The BinarySearchTree **Class**

The BinarySearchTree **class implements the** Set **interface, which has the same methods as the** Collection **interface, but does not allow duplicate elements.**

The AbstractSet **class has general-purpose implementations of** isEmpty()**,** toString()**,** clear()**,** toArray()**, …**

```
public class BinarySearchTree<E>
                    implements Set<E>,
                    extends AbstractSet<E>
```

The BinarySearchTree **class is not in the Java collections framework, but it is a much simplified version of the** TreeSet **class, which is in the Java collections framework. The** BinarySearchTree **class has very few defined methods:**

```
// Initializes this BinarySearchTree object to be empty,
// with elements of type E.
public BinarySearchTree( )


// Initializes this BinarySearchTree object to contain a
// copy of otherTree.
public BinarySearchTree (BinarySearchTree<E> otherTree)
```

```
// Returns the number of elements in this
// BinarySearchTree object
public int size( )

// Returns an iterator positioned at the first element
// in this BinarySearchTree object
public Iterator<E> iterator( )
```

```
// Returns true if there is an element equal to obj in this
// BinarySearchTree object.  The averageTime(n) is
//  O(log n), and worstTime(n) is O(n).
public boolean contains (Object obj)
```

```
// Returns false if, before this call, this BinarySearchTree
// object contained an element equal to element.  Otherwise,
// element has been inserted where it belongs in this
// BinarySearchTree object and true has been returned.
// The averageTime(n) is O(log n), and worstTime(n) is O(n).
public boolean add (E element)
```

```
// Returns false if, before this call, this BinarySearchTree
// object did not contain an element equal to obj.
//  Otherwise, an element equal to obj has been
//  removed from this BinarySearchTree object
//  and true has  been returned. The averageTime(n) is
// O(log n), and worstTime(n) is O(n).
public boolean remove (Object obj)
```

Exercise: In a processInput (String s) **method, convert** s **into an int** n**, and then construct a** BinarySearchTree **object** tree **that contains** Integer**s with values 0, 1, …,** n **– 1.**

The following main **method reads words from the input into a** BinarySearchTree **until "***" is read in.**

Then the first word, the last word, and "maybe" are deleted, and after each deletion, the words are printed in alphabetical order.

```
public static void main (String[ ] args)
{
    final String SENTINEL = "***";

    final String PROMPT = "Enter a word, or " + SENTINEL +
                            " to quit: ";

    BufferedReader reader = new BufferedReader
             (new InputStreamReader (System.in));

    BinarySearchTree<String> tree =
             new BinarySearchTree<String>();
```

```
try
{
    while (true)
    {
        System.out.print (PROMPT);

        String word = reader.readLine();
        if (word.equals (SENTINEL))
            break;
        tree.add (word);
    } // while
```

```
        Iterator<String> itr = tree.iterator();
        tree.remove (itr.next());
        System.out.println (tree);

        String save = "";
        for (String word : tree)
            save = word;
        tree.remove (save);
        System.out.println (tree);

        tree.remove ("maybe");
        System.out.println (tree);
    } // try
    catch (IOException e) { }
} // method main
```

Fields and Implementation of the BinarySearchTree Class

We assume that the elements in a BinarySearchTree are objects in a class that implements the Comparable interface:

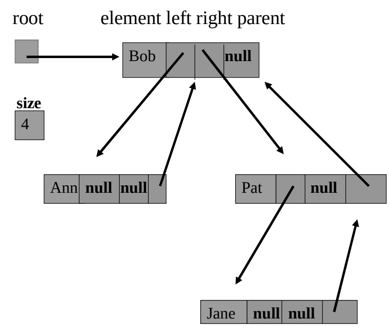
```
public interface Comparable
{
    int compareTo(Object obj);
} // interface Comparable
```

```
String s = "mellow";

System.out.println (s.compareTo ("minty"));
```

The output will be < 0 because "mellow" is, lexicographically, less than "minty". In general, the int returned will be < 0, = 0, or > 0 depending on whether the calling object is less than, equal to, or greater than the argument.

```
Entry<E> root;

int size;
```

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

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

    // Initializes this Entry object from element
    // and parent.
    protected Entry (E element, Entry<E> parent)
    {
       this.element = element;
       this.parent = parent;
    } // constructor
} // class Entry
```

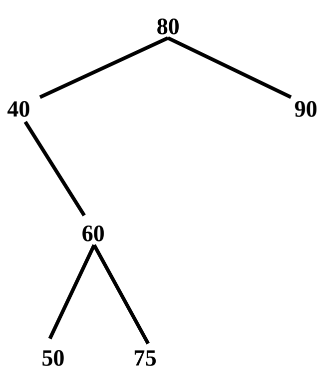



```
public Iterator<E> iterator()
{
      return new TreeIterator();
} // method iterator
```

For the contains**, add, and** remove **methods,**

Keep in mind that the only element immediately accessible is the root element: root.element**.**

Each element in the left subtree is less than the root element, and each element in the right subtree is greater than the root element.



contains (75)? contains (73)?

```
public boolean contains (Object obj)  {
      Entry<E> temp = root;
      int comp;
      while (temp != null) {
          comp =  ((Comparable)obj).compareTo
                                       (temp.element);
          if (comp == 0)
               return true;
          if (comp < 0)
               temp = temp.left;
          else
               temp = temp.right;
      } // while
      return false;
} // contains
```

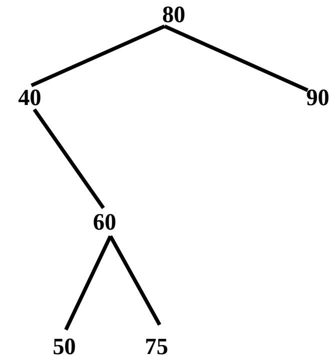
The averageTime(*n*) for a successful search:

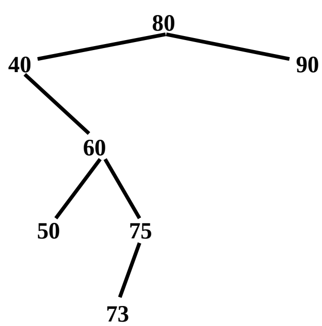
The average height of a binary search tree is logarithmic in *n*.

So averageTime(*n*) is O(log *n*). In fact, averageTime(*n*) is logarithmic in *n*.

The worstTime(*n*) occurs if the tree is a chain. So worstTime(*n*) is ????

add (73);





Will the inserted element always be a leaf?

```
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
```

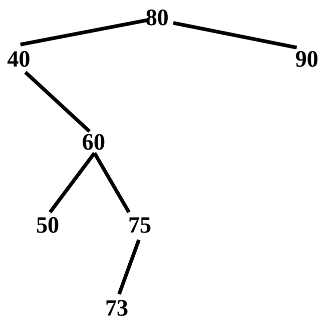
For adding an element, what is the worst case?
What is the worst height?

The worstTime (n) is linear in n.

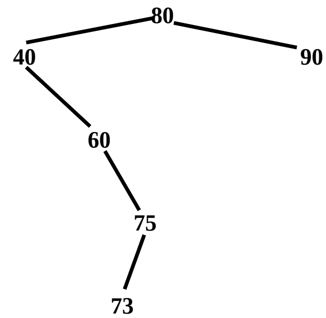
What is the average height?

The averageTime (n) is logarithmic in n.

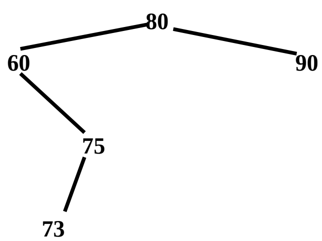
remove (50);



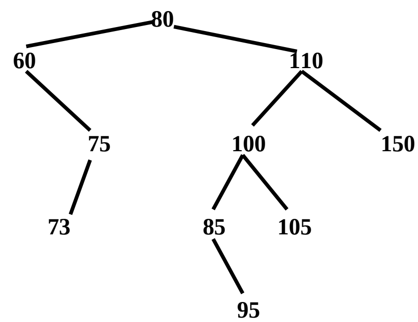
remove (40);



After removing 40:



remove (80);



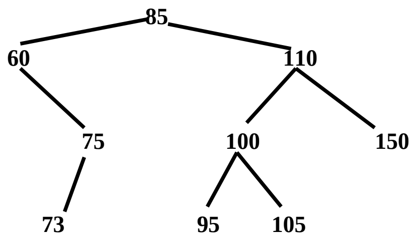
The element 80 has two children, so we cannot simply unlink 80 from the tree: that would create a hole.

Of the elements already in the tree, two could replace 80 (and then have the original deleted) without destroying the binary search tree properties. Which two?

We can replace 80 with either its predecessor, 75, or its successor, 85. We'll choose its successor because we will need the same successor **method later (where?). The successor of an element is the leftmost element in the right subtree.**

Replace 80 with 85, and then remove 85.

After removing 80:

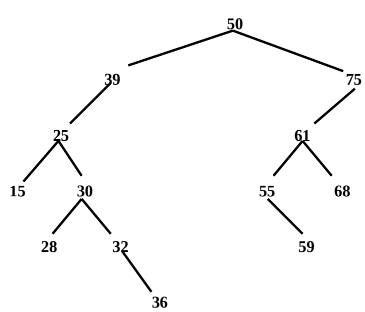


Can removing the successor get complicated?
Can the successor have two children?

What is worstTime(*n*)?

What is averageTime(*n*)?

```
// Returns the successor Entry of e, if e has a successor.
// Otherwise, returns null.  The averageTime(n) is constant,
and // worstTime(n) is O(n).
protected Entry<E> successor (Entry<E> e)
```



Successor of 36? Successor of 50?

```
protected Entry<E> successor (Entry<E> e)
{
      if (e == null)
            return null;
      else if (e has a right child)
            // successor is leftmost Entry in right subtree of e
      else
             // go up the tree to the left as far as possible,
            then go up // to the right.

} // method successor
```

The TreeIterator **Class**

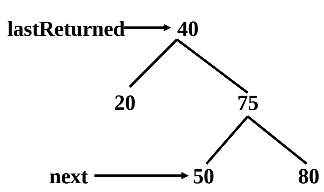
```
protected class TreeIterator implements Iterator<E>
{
    protected Entry<E> lastReturned = null,
                       Entry<E> next;
```

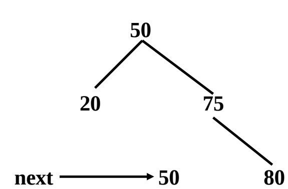
Default Constructor:

Where should we start iterating? Root or smallest element?

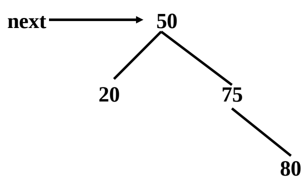
```
public E next( ) {

    lastReturned = ?

    next = ?

    return ?

} // method next
```

```
public void remove( ) {

    BASICALLY:

    remove (lastReturned.element);
    lastReturned = null;
```





Uggh!



Ahhh!

Exercise: Draw the tree and determine the contents of the BinarySearchTree object myTree after the following:

```
BinarySearchTree<String> myTree =
        new BinarySearchTree<String>();
```

```
myTree.add ("C");
myTree.add ("O");
myTree.add ("N");
myTree.add ("G");
myTree.add ("R");
myTree.add ("A");
myTree.add ("T");
myTree.add ("U");
myTree.add ("L");
myTree.add ("A");
myTree.add ("T");
myTree.add ("I");
myTree.add ("O");
myTree.add ("N");
myTree.add ("S");
myTree.remove ("C");
Iterator<String> itr = myTree.iterator( );
itr.next( );
itr.next( );
itr.next( );
itr.remove( );
itr.next( );
System.out.println (itr.next( ));
```

The Problem:

For the contains**,** add**, and** remove **methods in the** BinarySearchTree **class, the bad news is that worstTime(*n*) is linear in *n* (for example, if the tree is a chain).**

The good news is that averageTime(*n*) is logarithmic in *n* for those methods.

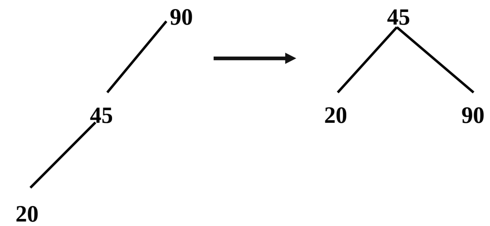
A tree-oriented data structure is *balanced* if its height is logarithmic in *n*.

For any balanced binary search tree, searching, inserting and deleting have worstTime(*n*) that is logarithmic in *n*.

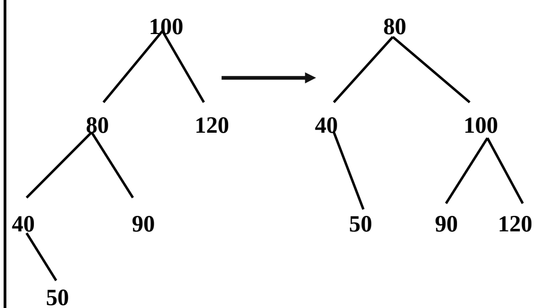
The balance is maintained through rotations.

A *rotation* is an adjustment to the tree, around an element, that maintains the required ordering of elements.

Here is a right rotation around 90:

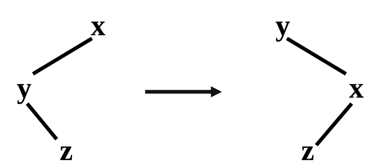


Here is a right rotation around 100:

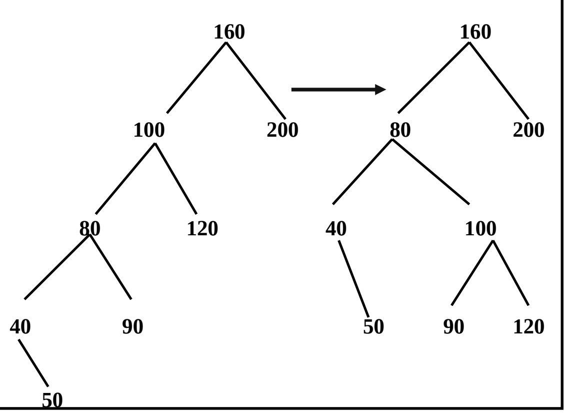


Notice that 90 is now in the right subtree.

In general, for any right rotation around element x, the right subtree of x's left child becomes the left subtree of x.



Here is a right rotation around 100:



In a rotation around x, the only restructuring is to the subtree rooted at x.

Let p **(for parent) be a reference to an** Entry **object, and let** l **(for left child) be a reference to the left child of** p**.**

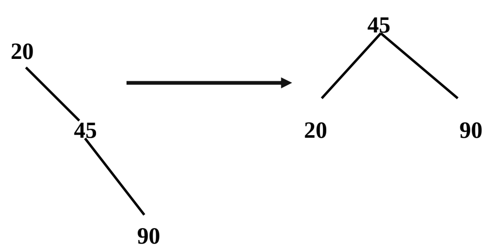
For a right rotation around p:

```
p.left = l.right;

l.right = p;
```

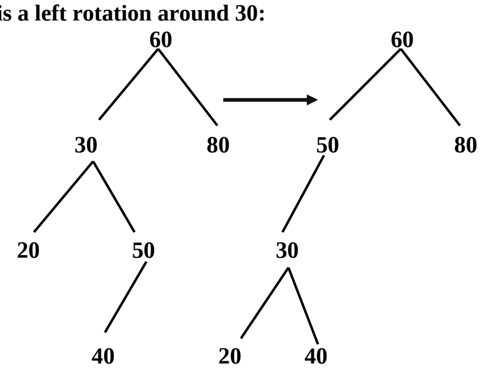
The complete method also adjusts parents:

```
private void rotateRight(Entry<E> p) {
    Entry<E> l = p.left;
    p.left = l.right;          ← From previous slide
    if (l.right != null) l.right.parent = p;
    l.parent = p.parent;
    if (p.parent == null)
        root = l;
    else if (p.parent.right == p)
        p.parent.right = l;
    else p.parent.left = l;
    l.right = p;               ← From previous slide
    p.parent = l;
}
```

A left rotation around 20:

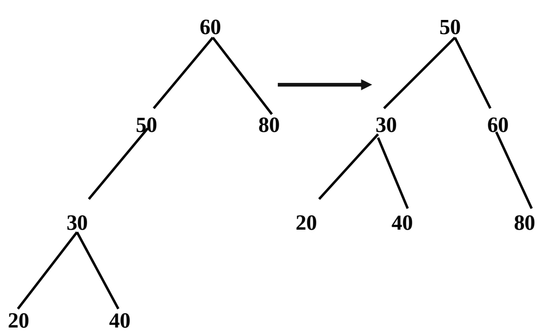


Here is a left rotation around 30:



The height of the tree is still 3. What now?

Now a right rotation around 60:



- ✓ **There are four kinds of rotation:**
  1. **A left rotation;**
  2. **A right rotation;**
  3. **A left rotation around the left child of an element, followed by a right rotation around the element itself;**
  4. **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 the rotation.

- ✓ Elements not in the subtree of the element rotated about are unaffected by the rotation.
- ✓ A rotation takes constant time.

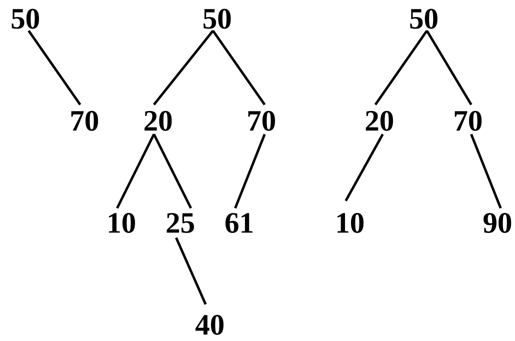
- ✓ Elements not in the subtree of the element rotated about are unaffected by the rotation.
- ✓ A rotation takes constant time.
- ✓ Before and after a rotation, the tree is still a binary search tree.

- ✓ Elements not in the subtree of the element rotated about are unaffected by the 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 "left" and "right."

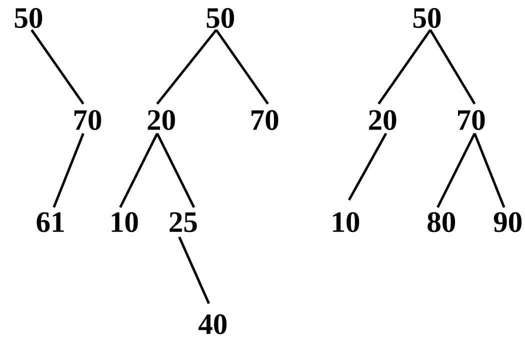
AVL Trees

An *AVL tree* is a binary search tree that either is empty or in which:

1. **The heights of the left and right subtrees differ by at most 1;**
2. **The left and right subtrees are AVL trees.**



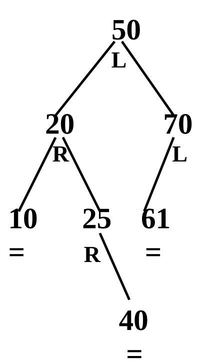
Some AVL Trees



Some binary trees that are not AVL trees

If the heights of the left and right subtrees of a binary search tree are the same, must the tree be an AVL tree?

```
public class AVLTree<E> extends BinarySearchTree<E>
{
    // Override the add and deleteEntry method definitions.


    protected static AVLEntry<E>
                    extends BinarySearchTree.Entry<E>

        protected char balanceFactor = '=';

        // definition of constructor

     } // embedded class AVLEntry

} // class AVLTree
```



Exercise: Create an AVL tree of height four that has as few elements as possible. Include balance factors.