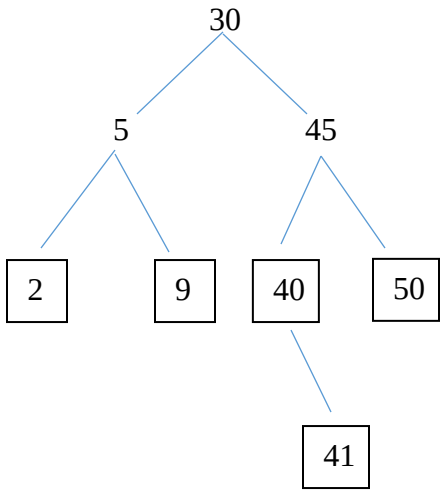
Chapter 12: Tree Maps and Tree Sets

- This chapter introduces another kind of balanced binary tree: the red-black tree.
- Red-Black trees are foundation for two extremely valuable classes: the TreeMap and the TreeSets classes, both of which are part of JCF
- In a TreeMap object, each element has two parts: a Key part, by which the element is compared to and a value part consist of the rest of the element.
- No two elements in a TreeMap object can have the same key.
- A TreeSet objectis a TreeMap object in which all elements have the same value parts.

Red-**Black** Trees

- Recall from Chapter 9, if element A is an ancestor of element B, the **path** from A to B is a sequence elements, starting with A and ending with B, in which each element in sequence (except the last) is the parent of the next element.
- We will be interested in paths from the root to elements with no children or with one child.

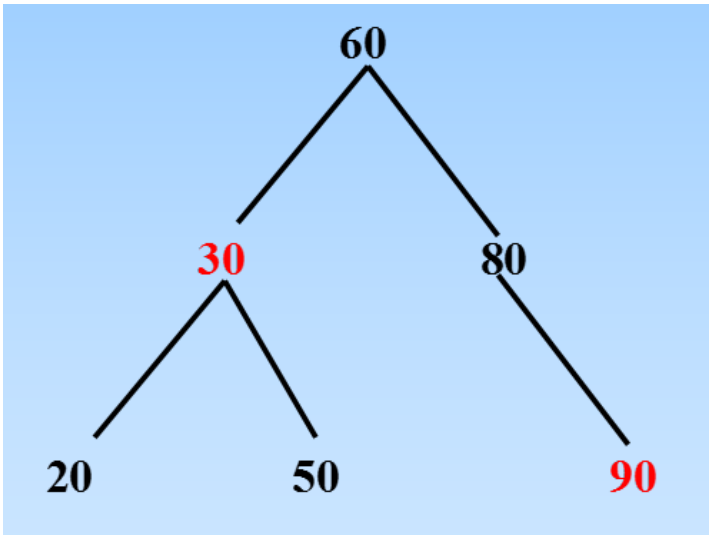


- A red-black tree is a balanced binary search tree.
- A red-black tree is a binary search tree that is empty or in which the root element is colored black, every other element is colored red or black, and

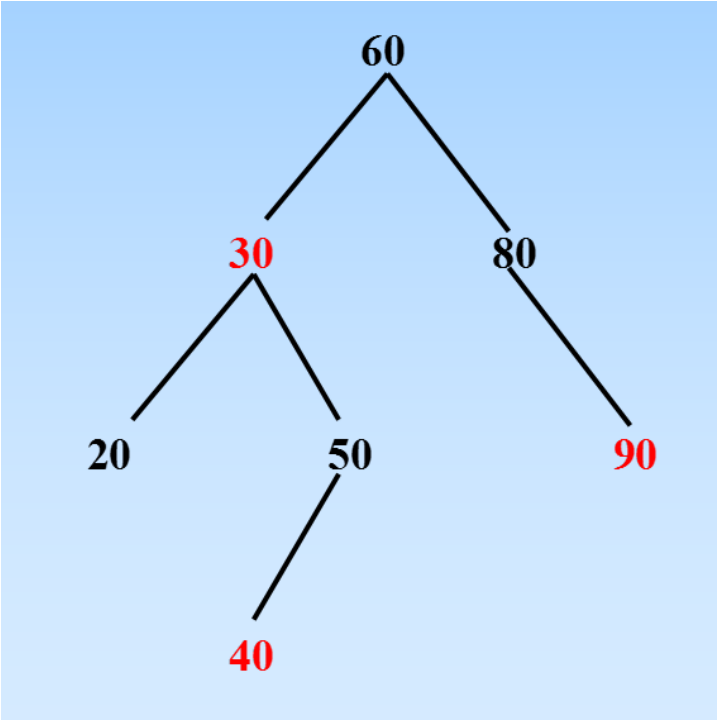
1. (Red rule) A red element cannot have any red children;

2. (Path rule) The number of black elements is the same in any path from the root element to any element with no children or with one child.

The following are red-black trees:

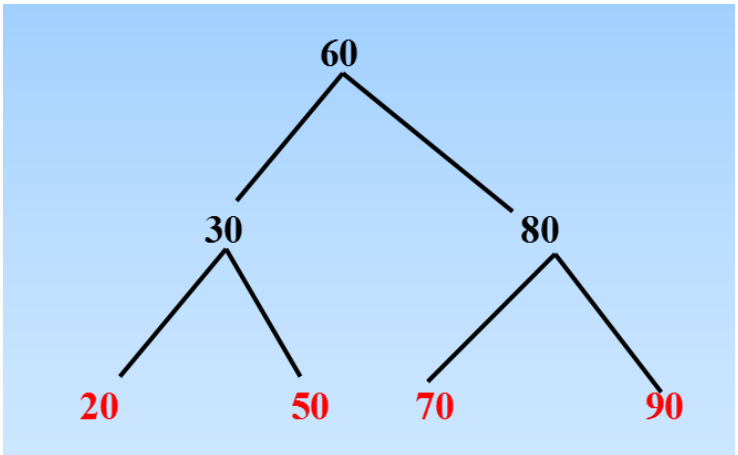


HOW MANY PATHS ARE THERE FROM THE ROOT ELEMENT TO AN ELEMENT WITH NO CHILDREN OR WITH ONE CHILD?

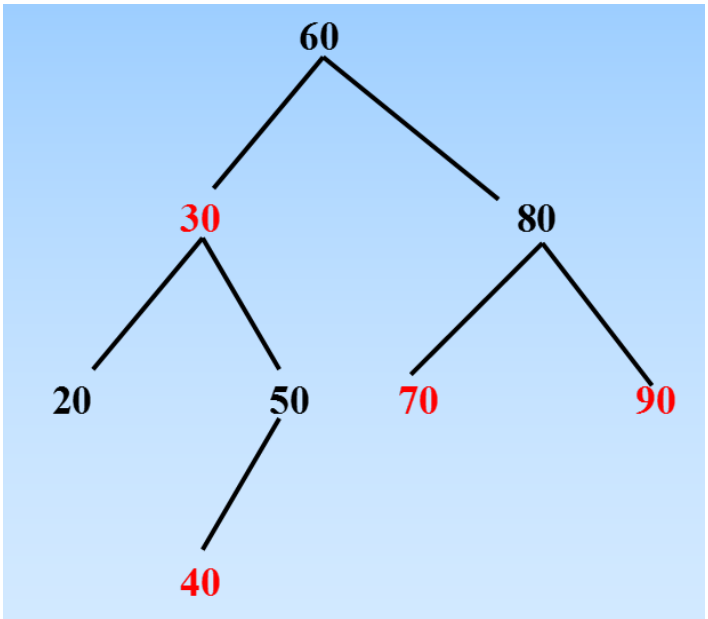


How about this one?

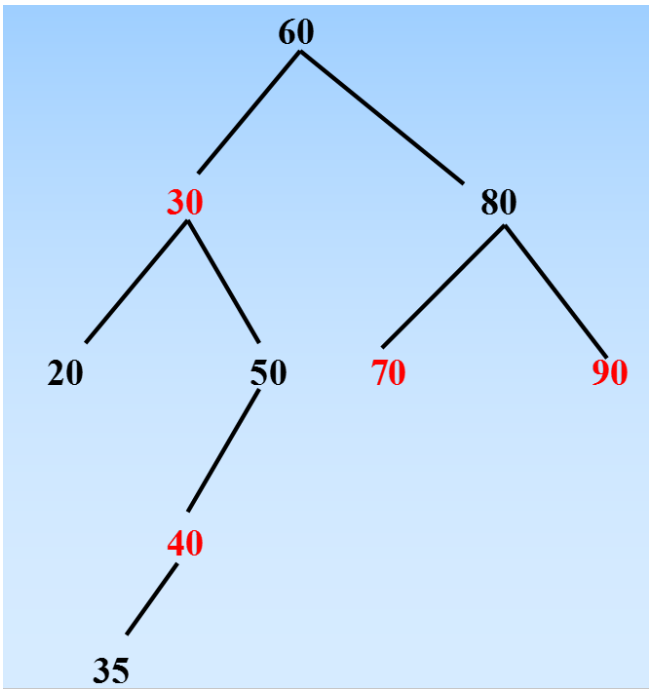
Let look at another example:



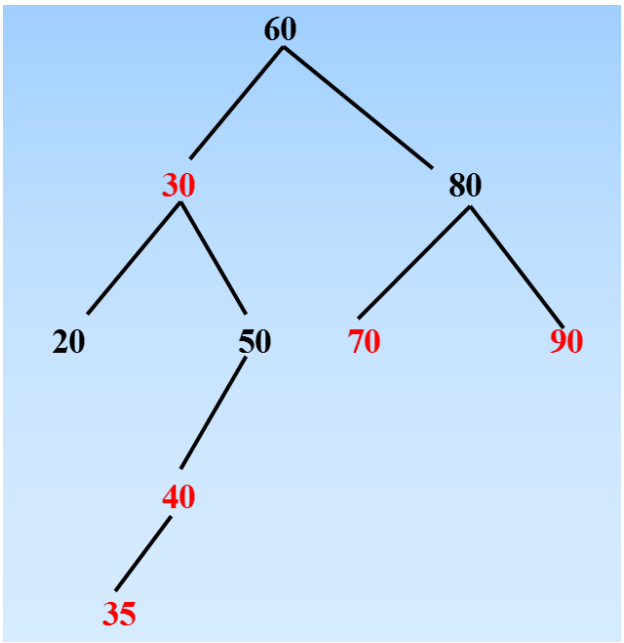
Add 40?



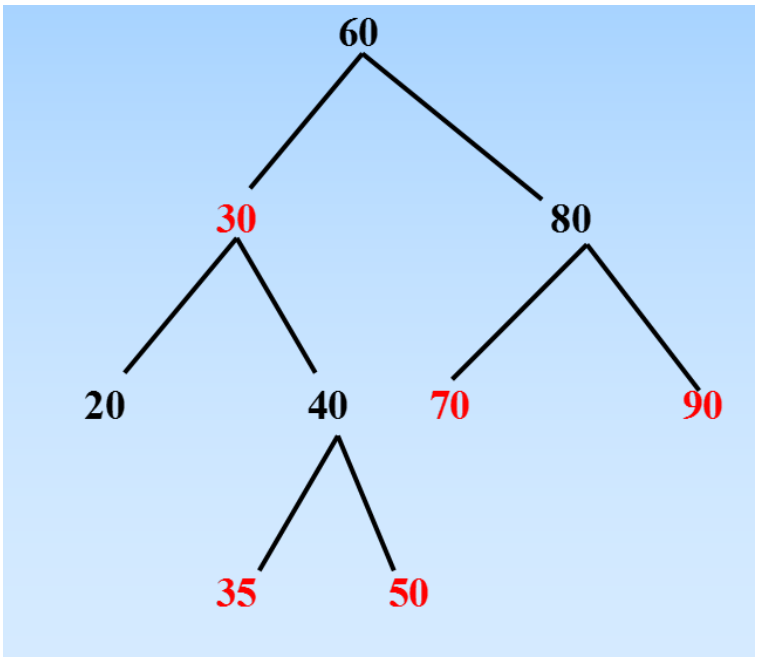
Add 35?



Is this a red-black tree?



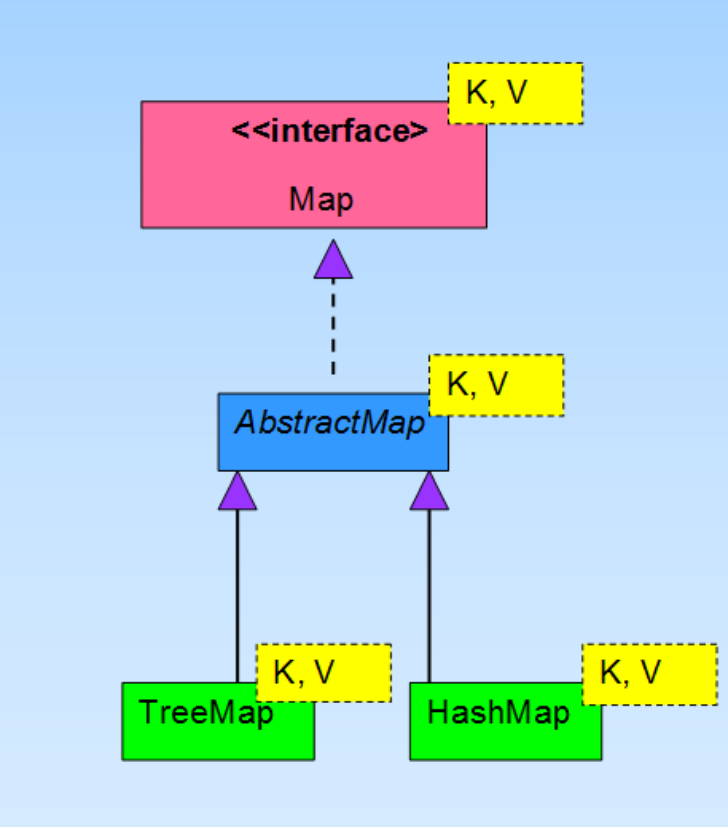
How about now?


60
30
80
20
40
70
90
35
50

The Map interface

- A **map** is a collection in which each element has two parts: a unique **key** part and a **value** part.
- For example, we can create a map of students, in which each key is the student ID, and each value is the student's GPA.
- Just as with the LinkedList and BinarySearchTree classes, each element is stored in an Entry object. The public Map.Entry interface has getKey() and getValue() methods.
- The Java Collection Framework's TreeMap class stores a map in a red-black tree, ordered by keys. In the class heading, the type parameters K and V stand for "Key"And "Value".

  public class TreeMap<K, V> implements SortedMap<K, V> extends AbstractMap<K, V>



Note: The TreeMap class does not implement the Collection interface – because many of the methods are key-value oriented.

Example:

```
TreeMap<String, Double> students =  new TreeMap<String, Double>( );

students.put (“L00000000”, 3.7);
students.put (“L11111111”, 2.0);
students.put (“L22222222”, 3.5);
students.put (“L44444444”, 3.7);
students.put (“L33333333”, 4.0);
students.put (“L22222222”, 3.8);
```

Now the GPA for L22222222 is 3.8

```
/**
 *  Ensures that there is an element in this TreeMap object with the specified key&value pair. If
 this TreeMap object had an element with the specified key before this method was called, the
 previous value associated with that key has been returned.  Otherwise, null has been returned.
 *  The worstTime (n) is O (log n).
 *  @param key – the specified key
 *  @param value – the specified value
 *  @return the previous value associated with key, if there was such a mapping; otherwise, null.
* @throws ClassCastException – if key cannot be compared with the keys currently in the map.
* @throws NullPointerException – if key is null and this Map object uses the natural order, or
the comparator does not allow null keys.
*/
```

public V put (K key, V value)

```
/**
* Determines if this TreeMap object contains a mapping
* with a specified key.
 * The worstTime (n) is O (log n).
 *
* @param key – the specified key
*
 * @return true – if this TreeMap object contains a mapping
 *              with the specified key; otherwise, false.
* @throws ClassCastException – if key cannot be compared
*              with the keys currently in the map.
* @throws NullPointerException – if key is null and this Map
*              object uses the natural order, or the comparator
 *              does not allow null keys.
 */
```

public boolean containsKey (Object key)

System.out.println (students.containsKey (“L11111111”)); // output: true

```
/**
* Determines if this TreeMap object contains a mapping
* with a specified value.
 * The worstTime (n) is O (n).
 *
* @param value – the specified value
*
 * @return true – if this TreeMap object contains a mapping
 *              with the specified value; otherwise, false.
 */
```

public boolean containsValue (Object value)

System.out.println (students.containsValue (3.4)); // output: false

```
/**
 *  Ensures that there is no mapping in this TreeMap object
 *  with the specified key. If this TreeMap object had such
*  a mapping before this method was called, the value
 *   has been returned.  Otherwise, null has been returned.
 *  The worstTime (n) is O (log n).
 *
 *  @param key – the specified key
 *  @return the value associated with key, if
 *              there was such a mapping; otherwise, null.
* @throws ClassCastException – if key cannot be compared
*              with the keys currently in the map.
* @throws NullPointerException – if key is null and this Map
*              object uses the natural order, or the comparator
*              does not allow null keys.
*/
```

public V remove (Object key)

System.out.println (students.remove (“L22222222”)); // output: 3.8

System.out.println (students.remove (“L23456789”)); // output: null

We can view a TreeMap object as a set of entries, as a set of keys, or as a collection of values, and then iterate accordingly.

```
/**
 *  @return a Set view of the mappings in this
 *              TreeMap object.
 */
```
public Set entrySet( )

Example: To print each student whose GPA is above 3.5:

```
for (Map.Entry<String, Double> entry : students.entrySet())
        if (entry.getValue() > 3.5)
                System.out.println (entry);
```

or

```
Iterator<Map.Entry<String, Double>> itr = students.entrySet().iterator();

while (itr.hasNext())
{
    Map.Entry<String, Double> entry = itr.next();
        if (entry.getValue() > 3.5)
                System.out.println (entry);
} // while
```

```
private transient Entry<K, V> root = null;

private transient int size = 0;

private transient int modCount = 0;

private Comparator<? super K> comparator = null;
```

Recall that transient means that the field itself will not be saved if the instance is serialized (saved to disk).

Example: Suppose we construct a TreeMap collection; the keys will be of type String and the values will be of type Integer.

```
TreeMap<String, Integer> myMap = new TreeMap<String, Integer>();
```

The int returned by

x.compareTo (y)

Is < 0, if x is less than y;

Is = 0, if x is equal to y;

Is > 0, if x is greater than y.

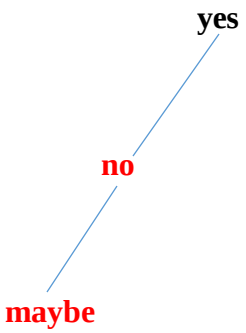
Since myMap was initialized with the default constructor, the keys are compared by the String class's compareTo method. That performs a lexicographical (≈ alphabetical) comparison:

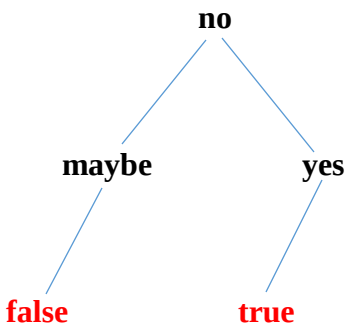
```
myMap.put ("yes", 1);
myMap.put ("no",  1);
myMap.put ("maybe",  1);
myMap.put ("true",  1);
myMap.put ("false",  1);
```

Here, step-by-step, is the red-black tree of keys; Note: When a key is inserted, it is initially colored red.

yes

Re-color black because the root must be black.





System.out.println (myMap.keySet());

The output will be:
[false, maybe, no, true, yes]

The TreeSet Class

A TreeSet is an ordered Collection in which duplicate elements are not allowed.

The TreeSet class has all of the methods in the Collection interface (add, remove, size, contains,…) plus toString (inherited from AbstractCollection) and several constructors.

The SortedSet interface extends the Set interface in two ways:

1. By stipulating that its iterator must traverse the Set in order of ascending elements.
2. By including a few new methods relating to the ordering, such as first(), which returns the smallest element in the instance, and last(), which returns the largest element in the instance.

The TreeSet class's contain, add and remove methods, the worstTime(n) is log(n).

public class TreeSet<E> extends AbstractSet<E> implements SortedSet<E>, Cloneable, java.io.Serializable

Here are a pair of TreeSet declarations and a few messages. The ByLength class was defined earlier.

```
TreeSet<Integer> tree1 = new TreeSet<Integer>( );
TreeSet<String> tree2 = new TreeSet<String> (new ByLength( ));

    tree1.add (83);
    tree1.add (74);
    tree1.add (83);
    tree1.add (92);
    if (tree1.remove (55))
            System.out.println ("How did 55 get there?");
    else
            System.out.println ("size of tree1 = " + tree1.size());
    System.out.println (tree1);

    tree2.add ("yes");
    tree2.add ("no");
    tree2.add ("maybe");
    tree2.add ("true");
    tree2.add ("false");
    System.out.println (tree2);
```

The output is:

```
size of tree1 = 3
[74, 83, 92]
[no, yes, true, false, maybe]
```