Wednesday, 30 December 2015

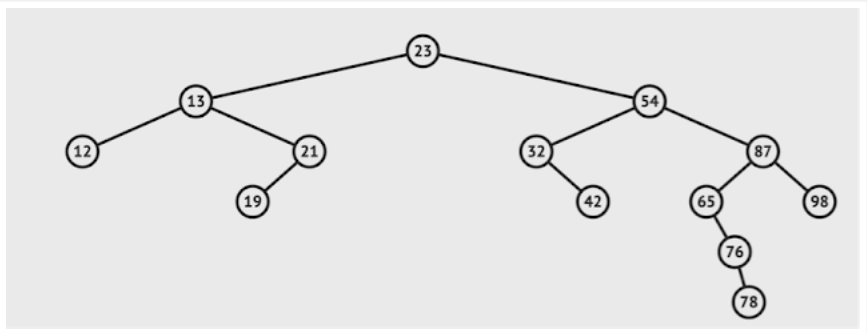
How TreeMap works in Java

In this post I am going to explain the internal working of TreeMap. It is just like HashMap, but is a Red-black tree based implementation. A Comparator provided at map creation time sorts the map according to the natural ordering of its keys.

Before discussing further, I want to explain about Binary search tree, Balanced binary search trees, Red-Black tree etc.,

What is Binary Tree?
Binary Tree is a tree kind of data structure, in which each node has at most two children. As per convention, these two children referred as left child and right child.

Following is the example of Binary Tree. 23 is the root element, 13 is the left child of root element 23 and 54 is the right child of root element 23. Leaf nodes don't have any left and right children. In the following figure 12, 19, 42, 78, 98 are the leaf nodes.



What is Binary Search Tree?
Binary search tree is a binary tree with following property. Assume 'X' represent value at a particular node, all children to the left of the node have values smaller than 'X', and all children to the right of the node have values larger than or equal to 'X'. Tree in above figure maintains binary search tree property. In best and average cases binary search tree guarantees O(log n) time to insert, search and delete element from Binary search tree, n is the number of elements in tree. In worst case Binary search tree takes O(n) time for insertion, search and deletion operations.

For example, to search for an element 78 in above figure, following is the procedure.

Step 1: Start from the root element 23, 78 >= 23, so we should travel right side of the root node.

Step 2: 78 >= 54, we should travel right side of node with element 54.

Step 3: 78 < 87, we should travel left side of the element 87

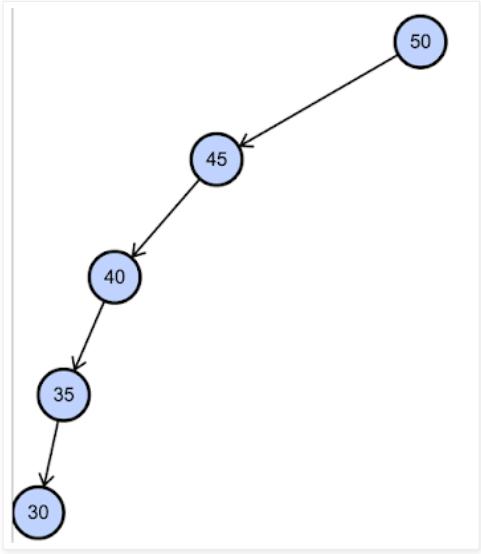
Step 4: 78 >= 65, we should travel right side of node with element 65

Step 5: 78 >= 76, we should travel right side of node with element 76

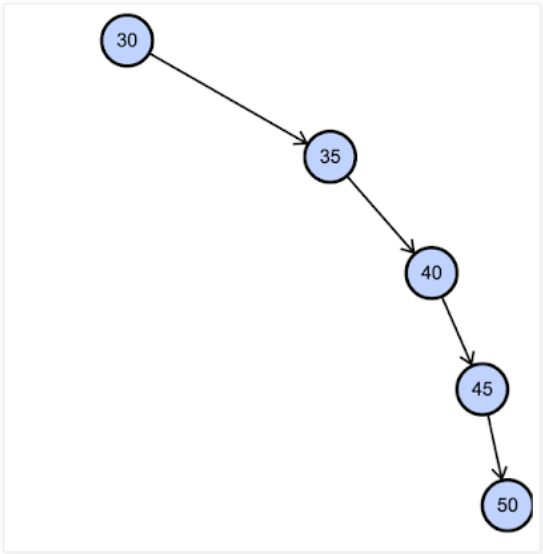
Step 6: Finally we found the element 78, return true.

When the worst case occurs for Binary search trees?
When elements are inserted in sorted order, then binary search tree goes to worst case. When elements are inserted in Ascending order, it becomes Skewed Right binary search tree. When elements are inserted in Descending order, then it becomes Skewed Left binary search tree.

If you insert 50, 45, 40, 35, 30 it becomes left skewed binary search tree.



If you insert 30, 35, 40, 45, 50 it becomes right skewed binary search tree.



Balanced Binary Search Trees

Balanced binary search trees are used to solve the worst-case (Left skewed, Right skewed trees) scenario of Binary search trees. For n number of elements, balanced binary search trees maintains a height of O(log n).

Following are popular Balanced Binary Search tree implementations.

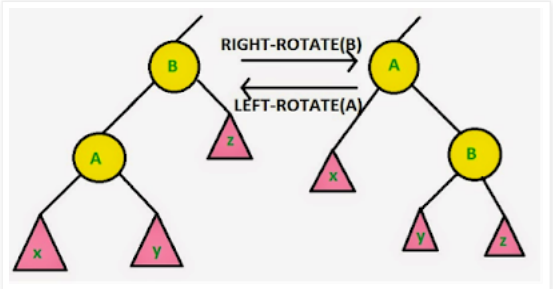
a. 2-3 tree
b. AA tree
c. AVL tree
d. Red-black tree
e. Scapegoat tree
f. Splay tree
g. Treap

Red-black tree

Java TreeMap is a Red-black tree implementation. Following are the properties of Red-black tree.

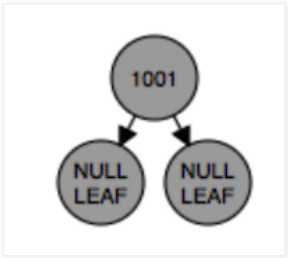
a. Every node is either red or black.
b. Every leaf is a NIL node, and is colored black.
c. If a node is red, then both its children are black. That is a red node can't have a red child node.
d. Every simple path from a node to a descendant leaf contains the same number of black nodes.
e. The root node is always black

Rotation in Red-black tree is like below.

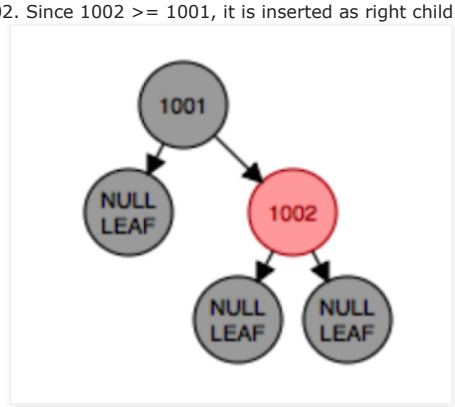


I am going to insert the numbers 1001, 1002, 1003, 1004, 1005, 1006 into a Red-black tree.

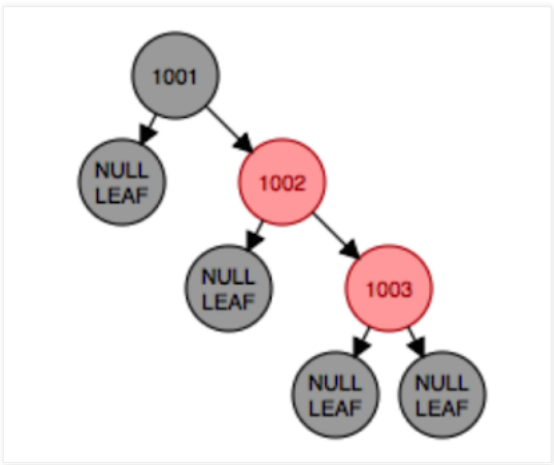
Step 1: Insert the number 1001



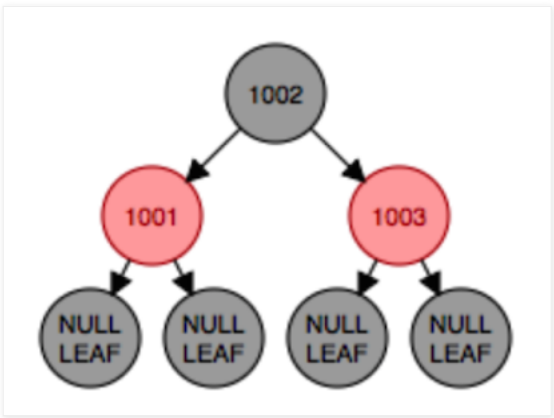
Step 2: Insert the number 1002. Since 1002 >= 1001, it is inserted as right child to 1001.



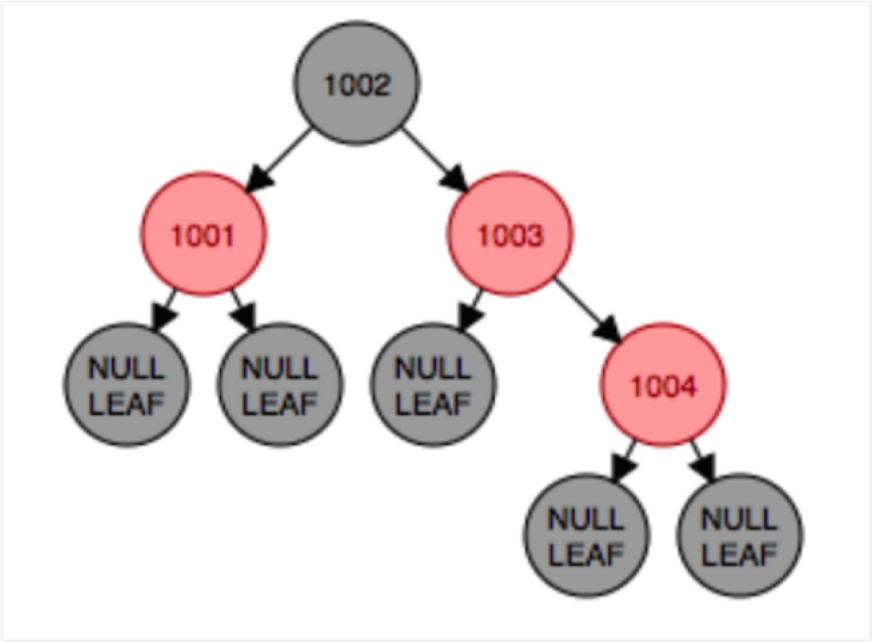
Step 3: Insert the number 1003. 1003 >= 1001, looking at right sub tree of 1001, 1003 >= 1002 insert at right sub tree of 1002.



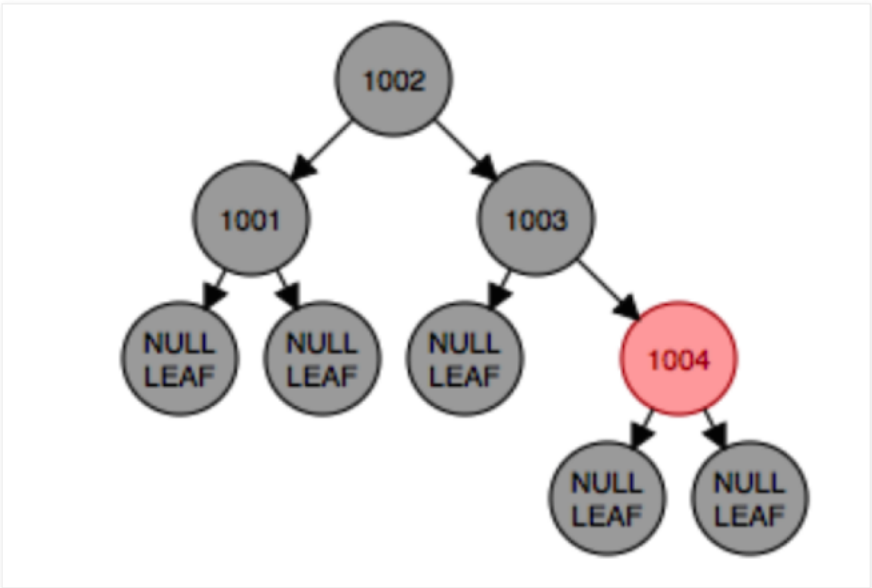
Node (1003) and parent node (1002) both are red. Node is right child, parent is right child, we can fix this extra redness with a single rotation, perform single rotate left.



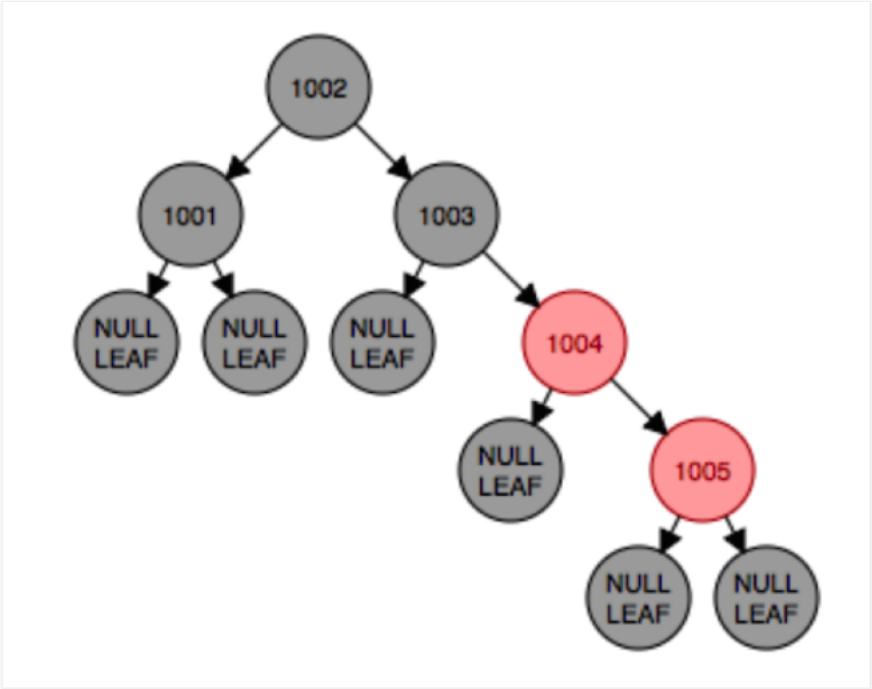
Step 4: Insert element 1004. 1004 >= 1002, looking at right sub tree of 1002, 1004 >= 1003 insert the element 1004 at right sub tree of 1003.



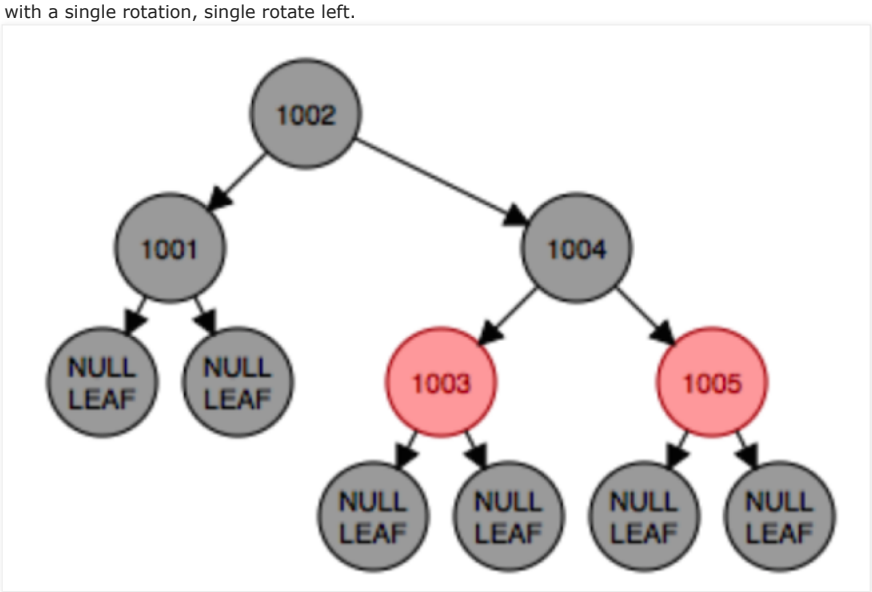
Node (1004) and parent node (1003) both are red, Uncle node 1001 is also red. Push blackness down from grandparent.



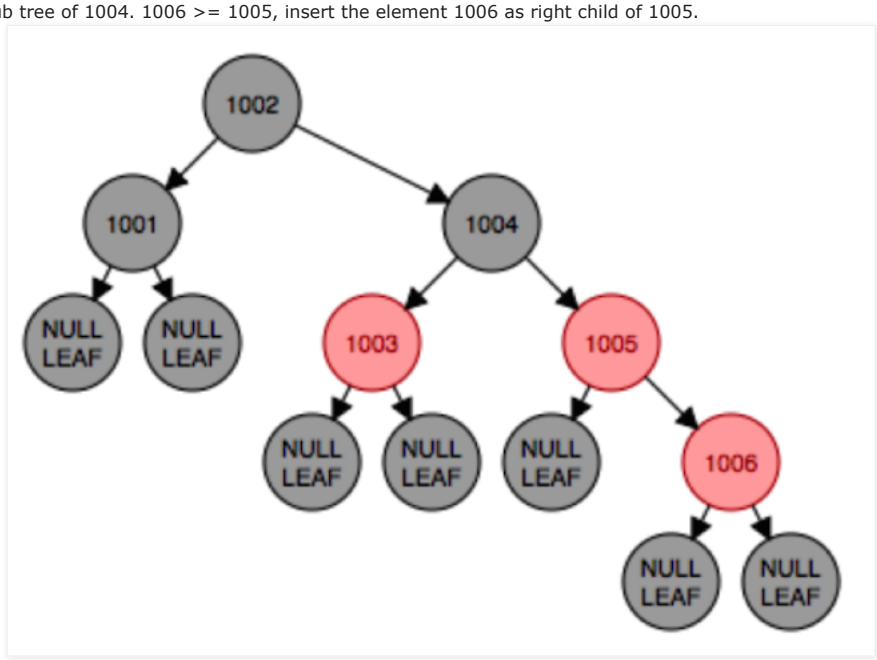
Step 5: Insert the element 1005. 1005 >= 1002, looking at right sub tree of 1002. 1005 >= 1003, looking at right sub tree of 1003. 1005 >= 1004, inserting 1005 as right child of 1004.



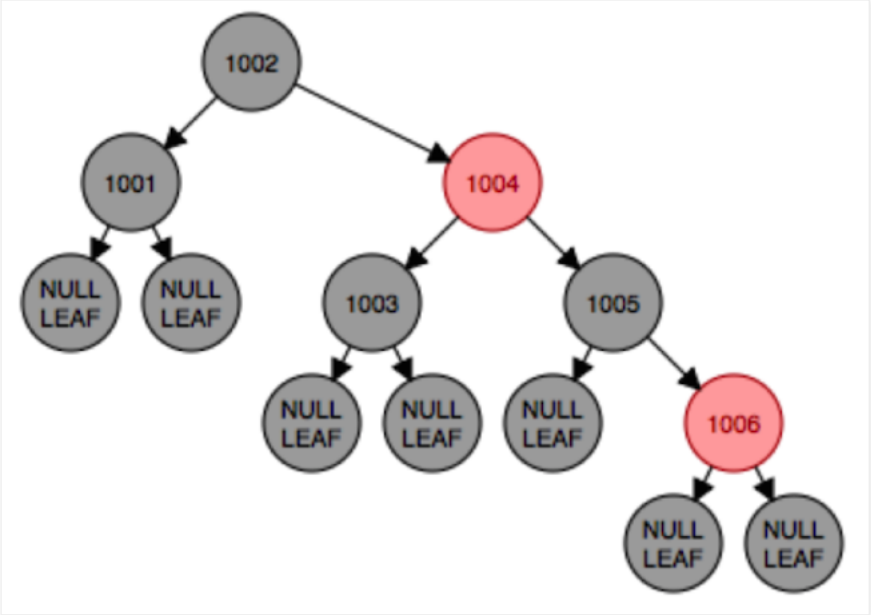
Node (1005) and parent node (1004) both are red. Node is right child, parent is right child, we can fix this extra redness with a single rotation, single rotate left.



Step 6: Insert the element 1006. 1006 >= 1002, looking at right sub tree of 1002. 1006 >= 1004, looking at right sub tree of 1004. 1006 >= 1005, insert the element 1006 as right child of 1005.



Node (1006) and parent node (1005) both are red, Uncle node 1003 is also red. Push blackness down from grandparent.



Same procedure is repeated for n number of elements to maintain the tree balanced. TreeMap maintain Red-black tree data structure to store <Key, Value> entries, so all the insertion, deletion and search operations are performed in O(log n) time.

TreeMap provides following extra functionality, which are not provided by HashMap.

Function	Description
pollFirstEntry()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
tailMap (K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
tailMap (K fromKey, boolean inclusive)	Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.
firstEntry()	Get the First Entry in TreeMap
firstKey()	Get the First Key currently in the map
floorEntry (K key)	Get the Floor Entry of this key
floorKey (K key)	Get the Floor key specific to this key
headMap (K toKey)	Get sorted map whose keys are strictly less than toKey
headMap (K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
higherEntry (K key)	Get the Mapping with least key strictly greater than the given key
higherKey (K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
lastEntry()	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
lastKey()	Returns the last (highest) key currently in this map.
lowerEntry (K key)	Get mapping associated with the greatest key strictly less than this key
lowerKey (K key)	Get greatest key strictly less than given key
subMap (K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	Returns a view of the portion of this map whose keys range from fromKey to toKey.
subMap (K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.

You may like
How HashMap works in Java
How to override hashCode method
Object Creation And Destruction
Lambda Expressions
Garbage Collection

Home

Posted by Krishna at 05:42