CIS 5511. Programming Techniques

Self-balancing BST

Since in a Binary Search Tree (BST) all major operations have worst-case costs proportional to the height of the tree, a "balanced" BST will be close to the best case, in which each node has two subtrees with similar heights. However, to keep this balance, structure adjustments are needed after certain insertions and deletions. Self-balancing BSTs have this capability, though they achieve it in different ways.

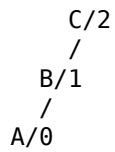
1. AVL tree

For each node in an AVL tree, its balance factor is $h_L - h_R$, where h_L and h_R is the height of its left and right subtree, respectively. In a balanced AVL tree, all balance factors are +1, -1, or 0.

After an insertion or deletion, the balance factor of some nodes may become +2 or -2, then the tree is "rebalanced" by changing its structure.

Let's start with BSTs with 3 nodes. There are 5 possible shapes, but only one of the them is balanced.

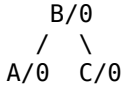
In the following case, each node shows its key over its balance factor



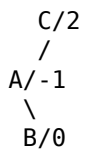
This is considered a "*LL*" situation, as the unbalanced node C is "left heavy" (+2) and so is its left child B (+1).

To rebalance the tree without changing the relative order of nodes (as defined in BST), a *rotation* is used.

The result of a right rotation of the top 2 nodes leads to



Another case:



This is a "*LR*" situation, as the unbalanced node C is "left heavy" (+2), and its left child A is "right heavy" (-1). After a left rotation of the bottom 2 nodes:

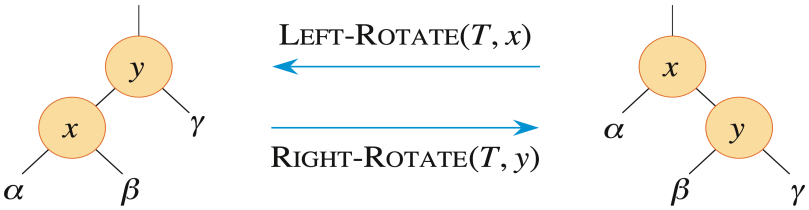
```
    C/2
    /
  B/1
  /
A/0
```

which is the "LL" case, then after a right rotation the tree is balanced

```
   B/0
  /  \
A/0  C/0
```

There are two mirror-image cases: "*RR*" and "*RL*".

The above process can be extended into general situations in BSTs (not limited to AVL trees). In such a situation, a rebalancing is still achieved by rotations, which change certain balance factors, while keeping the order of all the nodes involved.

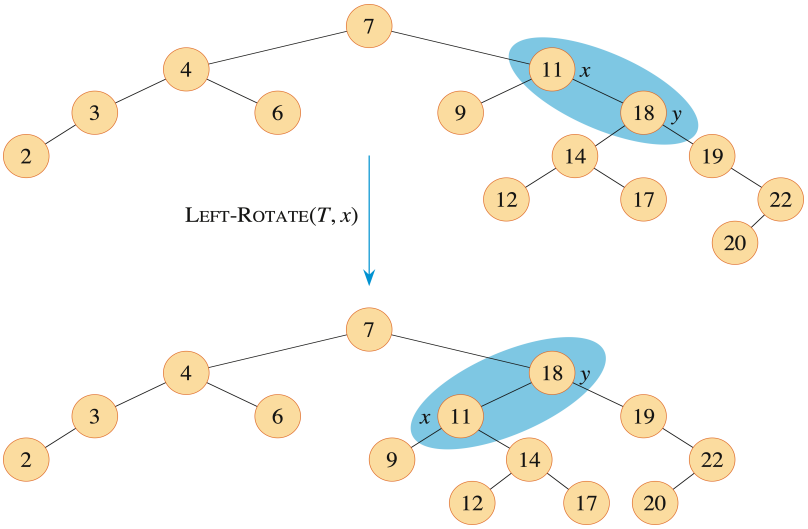


The algorithm for a left rotation at node x in tree T:

LEFT-ROTATE(T, x)

```
1  y = x.right
2  x.right = y.left       // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil      // if y's left subtree is not empty ...
4       y.left.p = x      // ... then x becomes the parent of the subtree's root
5  y.p = x.p              // x's parent becomes y's parent
6  if x.p == T.nil        // if x was the root ...
7       T.root = y        // ... then y becomes the root
8  elseif x == x.p.left   // otherwise, if x was a left child ...
9       x.p.left = y      // ... then y becomes a left child
10 else x.p.right = y     // otherwise, x was a right child, and now y is
11 y.left = x             // make x become y's left child
12 x.p = y
```

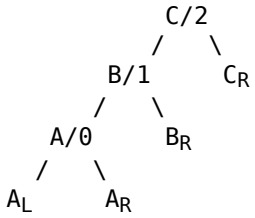
The algorithm is used in the following BST:



The right rotation algorithm is symmetric.

After an insertion or deletion in an AVL tree, one or more "out-of-balance" nodes (with balance factor 2 or -2) may appear in the path from the root to the insertion/deletion location, but cannot occur outside the path. If we check the "out-of-balance" node that is the "deepest" (the farthest from the root), there are only 4 canonical forms.

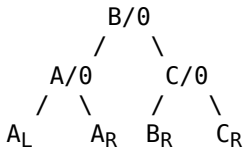
LL case:



Here A_L, A_R, B_R, and C_R have the same height.

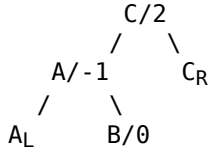
By the definition of BST, there is $A_L < A < A_R < B < B_R < C < C_R$

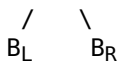
This requires a right rotation. B will become the root of the (local) tree. Its right subtree will be C. C will no longer have B as the root of its right subtree but will have B_L instead. B and C have balance factors of 0. The balance factors of the other nodes remain the same as before.



where we still have the order $A_L < A < A_R < B < B_R < C < C_R$.

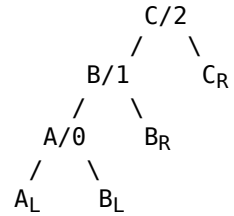
LR Case:



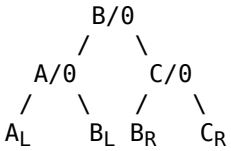


Assume that all the subtrees, A_L, B_L, B_R, and C_R, have the same height. For the order among the nodes, here we have $A_L < A < B_L < B < B_R < C < C_R$.

A left rotation at A:



then a right rotation at C:



where we still have $A_L < A < B_L < B < B_R < C < C_R$.

The *RL* and *RR* cases are symmetric.

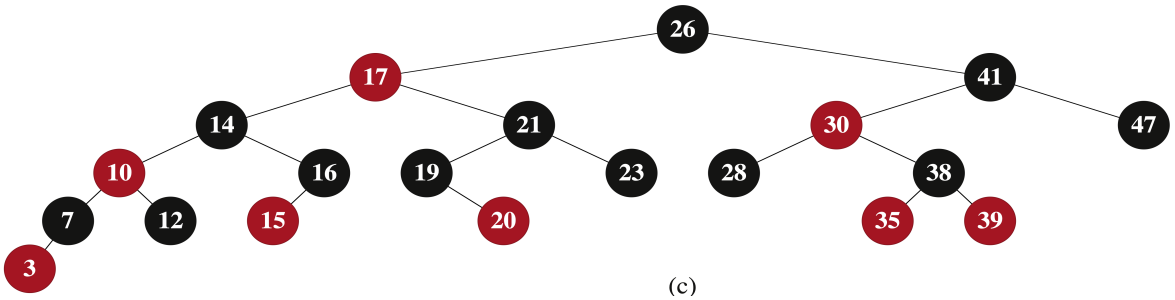
These four cases cover all possibilities. For the deepest out-of-balance node (i.e., with balance factor 2 or -2), if its child on the heavy side has balance factor 1 or -1, it directly maps into one of the above four cases: 2/1 to LL, 2/-1 to LR, -2/-1 to RR, and -2/1 to RL. If the third node (A in LL, B in LR and RL, and C in RR) has a balance factor 1 or -1, we can still use the above procedure to rebalance the tree. If the heavy-side child has balance factor 0 (which can only happen after a deletion, but not after an insertion), it is treated as LL or RR, and the tree will be balanced properly.

An AVL tree is maintained by making some changes to the algorithms for BST insertion and deletion. We can visualize the change as retracing the path from the root to the insertion/deletion position and checking to see whether the change caused any node in the path to become unbalanced. If necessary, a rebalance action is taken.

In a balanced BST, the time complexity of search, insertion, and deletion is O(log n), so it is usually more efficient than BSTs and linear data structures (array and linked list, sorted or not) in the major operations.

2. Red-Black Tree

Red-Black Tree is another self-balancing BST. It has a lower standard for "balance" (so is more efficient), though is less intuitive than AVL tree.



(c)

In a Red-Black tree, A node is either black (regular) or red (extra). The tree has the following properties:

1. The root is black.
2. Every path from a node to a descendant leaf contains the same number of black nodes.
3. A red node cannot have a red child.

It follows from the above definition that in a Red-Black tree, no leaf is more than twice as far from the root as any other, which defines a form of "balance". A red-black tree with n internal nodes has height at most `2log(n+1)`.

Similar to in AVL tree, an insertion in a Red-Black tree is an insertion in a BST, sometimes followed by a rotation to keep the balance, plus the re-coloring of certain nodes to keep the above properties. The rebalancing algorithm is much more complicated than the one for AVL tree.

In a Red-Black tree, the major operations take O(log N) time in the worst cases, as in an AVL tree. However, since in Red-Black trees the notion of "balance" allows more shapes than in AVL trees, rotations happen less often, so it is considered more efficient.

The Java TreeMap class is based on a Red-Black tree.

3. Augmenting Data Structures

Very often, a practical application requires additional functionality that is not provided by classical data structures. The general procedure to augment a data structure:

1. Choose an underlying data structure.
2. Determine additional information to maintain in the underlying data structure.
3. Verify that the additional information is maintained for the basic modifying operations on the underlying data structure.
4. Develop new operations.

One attribute that can be added into each node of a BST is the number of nodes in the tree rooted at the node (including itself), that is, the size of the local tree. Using this information, an "Order-Statistic Tree" can be built, in which algorithm OS-SELECT(x, i) retrieves an element in the tree with (local) root x with a given rank i.

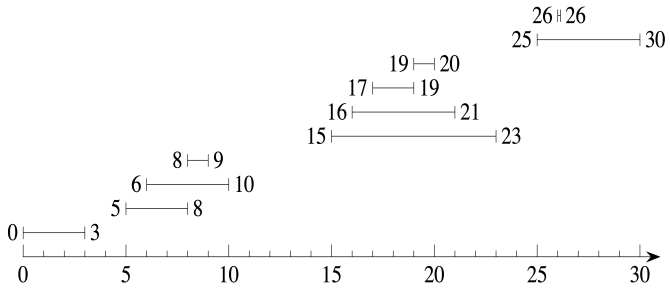
```
OS-SELECT(x, i)
1  r = x.left.size + 1    // rank of x within the subtree rooted at x
2  if i == r
3        return x
4  elseif i < r
5        return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

Algorithm OS-RANK(T, x) calculates the rank of a given node x in tree T by counting the nodes "on its left", both below and above.

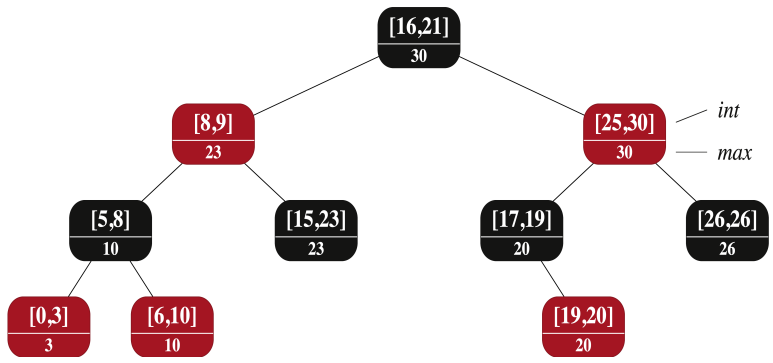
```
OS-RANK(T, x)
1  r = x.left.size + 1          // rank of x within the subtree rooted at x
2  y = x                        // root of subtree being examined
3  while y ≠ T.root
4        if y == y.p.right                  // if root of a right subtree ...
5              r = r + y.p.left.size + 1    // ... add in parent and its left subtree
6        y = y.p                            // move y toward the root
7  return r
```

Red-black trees can also be augmented to support operations on dynamic sets of intervals, which can be open or closed. The key of each interval *x* is the low endpoint, *x.int.low*, of the interval. In addition to the intervals themselves, each node *x* contains a value *x.max*, which is the maximum value of all interval endpoints stored in the subtree rooted at *x*.

(a)



(b)



Algorithm INTERVAL-SEARCH(T, i) searches the tree *T* for the given interval *i*, and use the additional information *max* to decide if a subtree should be searched.

INTERVAL-SEARCH(T, i)

```
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left    // overlap in left subtree or no overlap in right subtree
5      else x = x.right  // no overlap in left subtree
6  return x
```