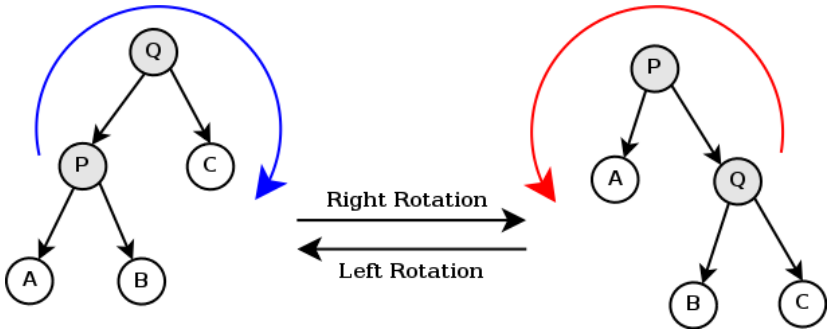
09: binary trees #2

Tree Rotations | DSW Algorithm | AVL Trees | Splay Trees

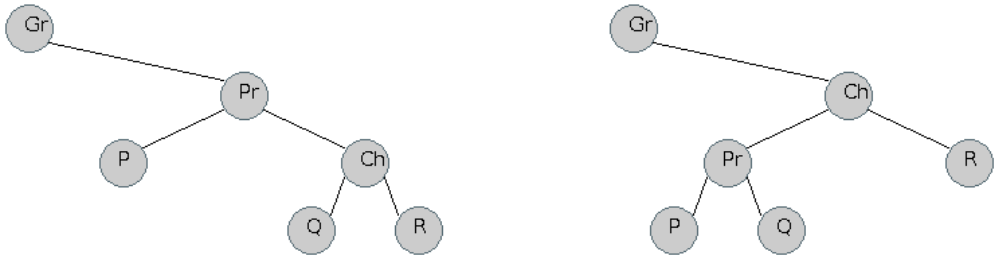
DSW Algorithm

DSW is based on rotations. It takes a possibly unbalanced tree and balances it all at once. Other algorithms keep a tree balanced after every insertion/deletion.



(Image from wikipedia [1])

- Rotate Left with Labels:



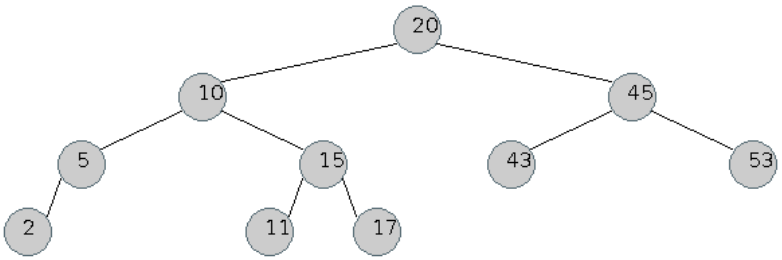
- Rotate Right with Labels:


```
1:  // rotations.cpp - download here
2:
3:  void rotateRight(TreeNode * gr, TreeNode * par, TreeNode * ch){
4:    //grandparent child becomes ch
5:    if(gr->getRight() == par){
6:      gr->setRight(ch);
7:    } else {
8:      gr->setLeft(ch);
9:    }
10:
11:   TreeNode * ch_right = ch->getRight();
12:   par->setLeft(ch_right);
13:   ch->setRight(par);
14: }
15:
```

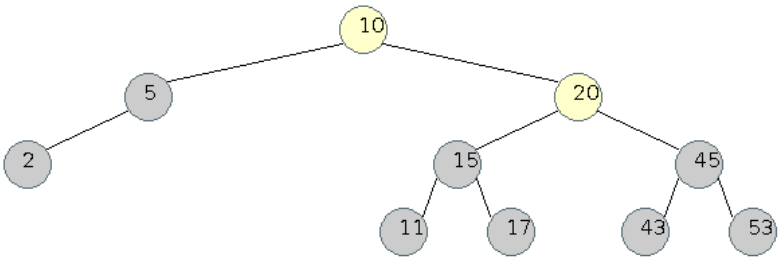
```
16: void rotateLeft(TreeNode * gr, TreeNode * par, TreeNode * ch){
17:   //grandparent child becomes ch
18:   if(gr->getRight() == par){
19:     gr->setRight(ch);
20:   } else {
21:     gr->setLeft(ch);
22:   }
23:
24:   TreeNode * ch_left = ch->getLeft();
25:   par->setRight(ch_left);
26:   ch->setLeft(par);
27: }
```

First we create the backbone (this takes O(n) worst case)

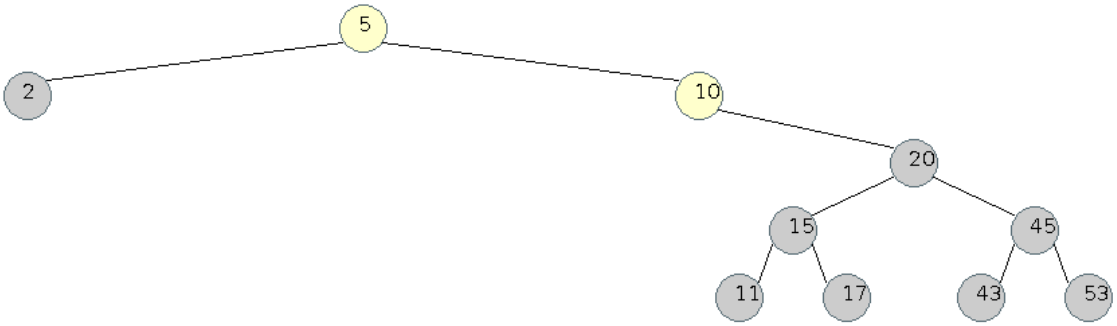
1. Start:



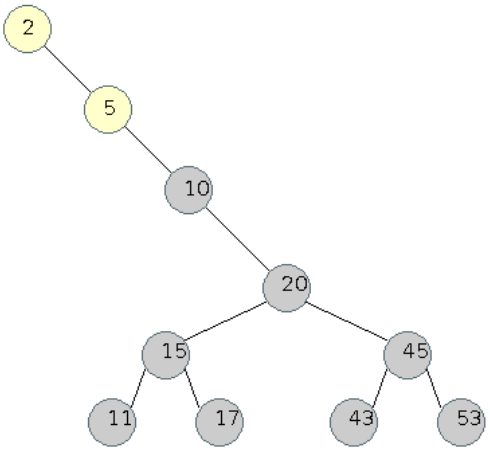
2. Rotate right (20, 10):



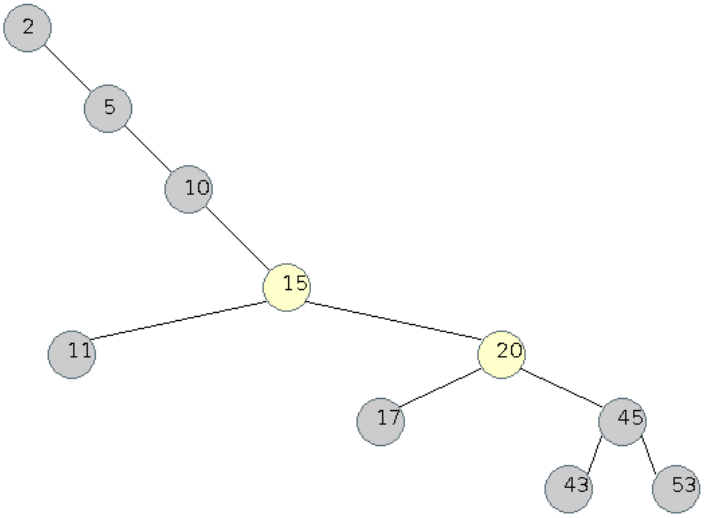
3. Rotate right (10, 5):



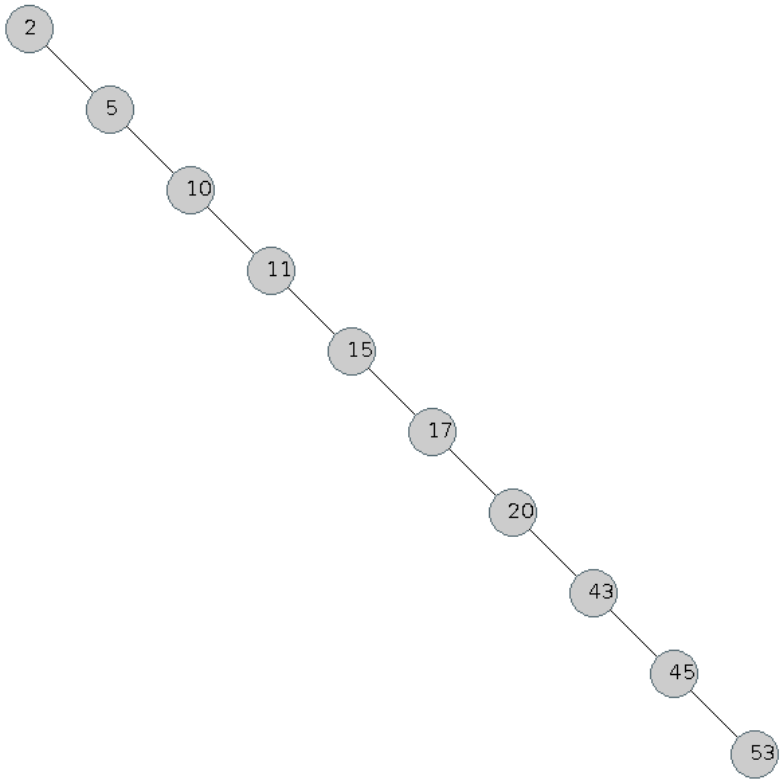
4. Rotate right (5, 2):



5. Rotate right (20, 15):

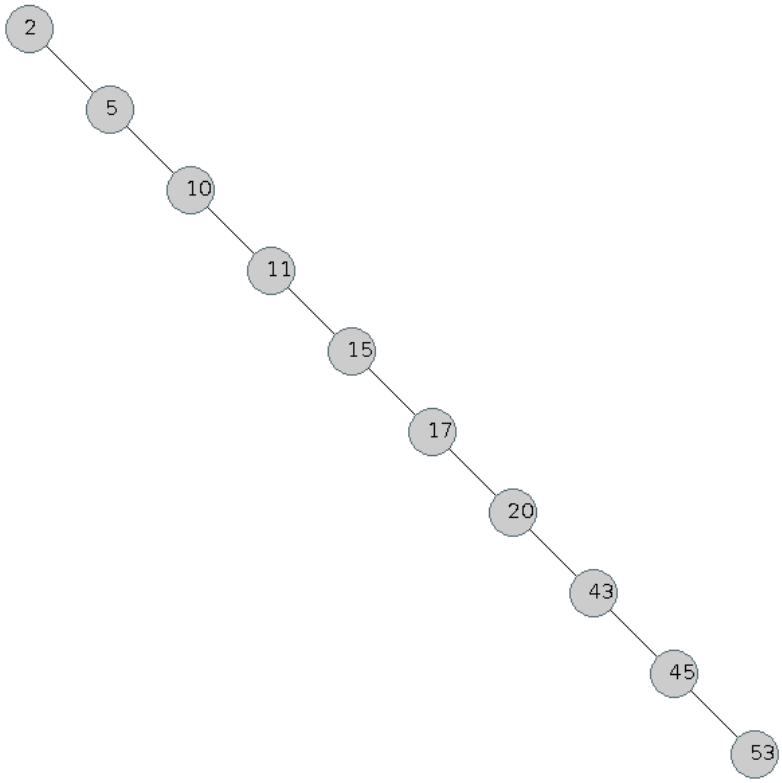


6. Apply algorithm to remainder:

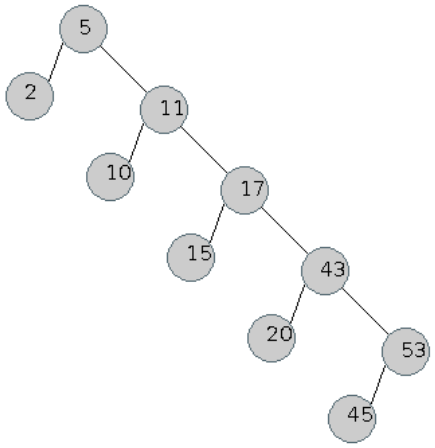


Then we create the tree (this also takes O(n) worst case). Execute rotations repeatedly for every other node.

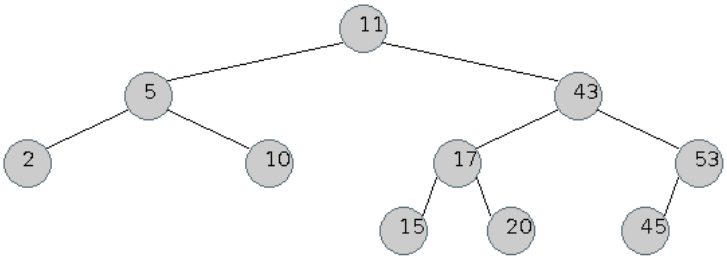
1. Start:



2. Rotate left at 5 and 11, 17, 43, 53



3. Rotate left at 11, 43. We are done because the maximum height difference is one.



(approximate) DSW Algorithm:

```
1:  algorithm createBackbone(root)
2:    tmp = root;
3:    while(tmp != NULL)
4:      if tmp has a left child
5:        rotate right about tmp and it's left child;
6:        set tmp to the left child that just became the parent;
7:      else
8:        set tmp to the right child
9:
10: algorithm createPerfectTree(n)
11:   make n/2 left rotations starting from the top;
12:   while (n > 1)

13:     n = n / 2;
14:     make n left rotations starting from the top;
```

AVL Trees

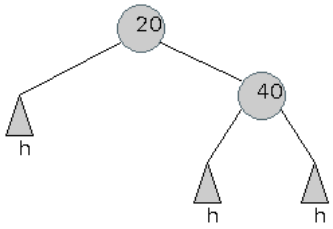
AVL Trees were the first self balancing binary tree invented [2].

- In AVL Trees each node has a balance factor that specify the difference in height between the left and right subtrees (balance = right_height - left_height).
- The balance factors should always be -1, 0 or +1. Otherwise rebalancing is needed
- AVL Trees do not guarentee that the tree is always perfectly balanced.
- AVL Trees are similar to Red-Black Trees.
  - AVL Trees have faster lookup (empirically) than Red-Black Trees because the balancing is more rigid
  - Red-Black Trees have faster insertion (empirically) and removal because the balancing is less rigid
- AVL Tree Time Complexities (Worst Case):
  - Search: O(lg(n))
  - Insert: O(lg(n))
  - Delete: O(lg(n))

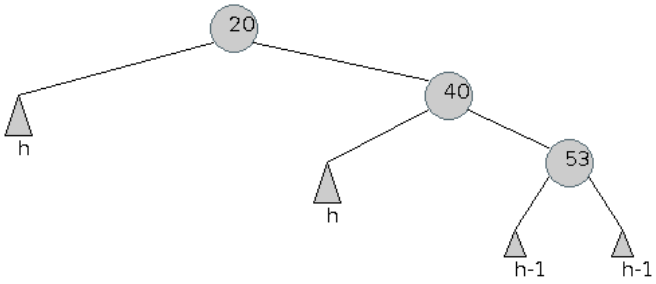
AVL Tree Insertion

After insertion, the balance factors need to be checked. There are six cases. Content from this section is adapted from [2].

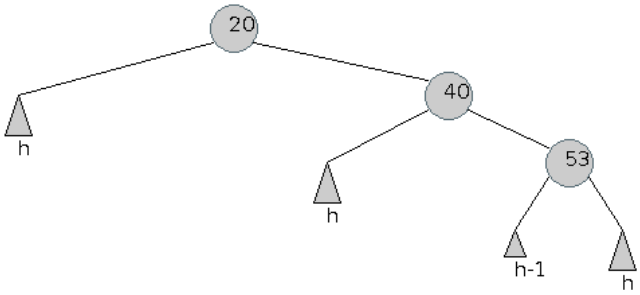
1. Case: +2, +1 (parent is 2 and signs are the same)
   1. Start:



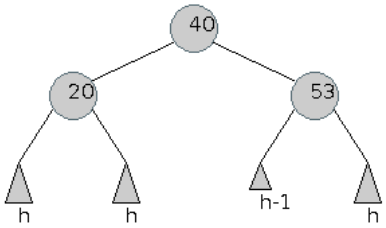
2. Show Node 53



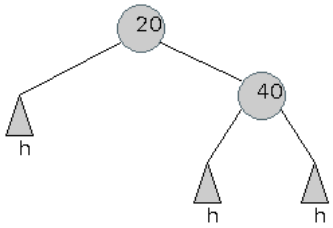
3. Insert value



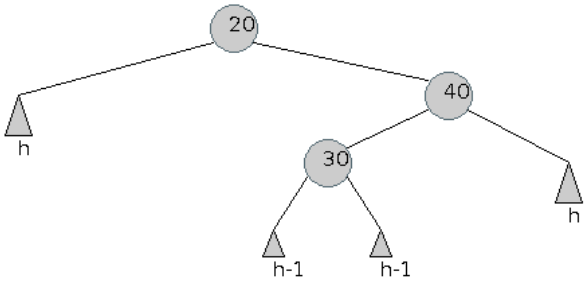
4. Right Rotation around 20 and 40



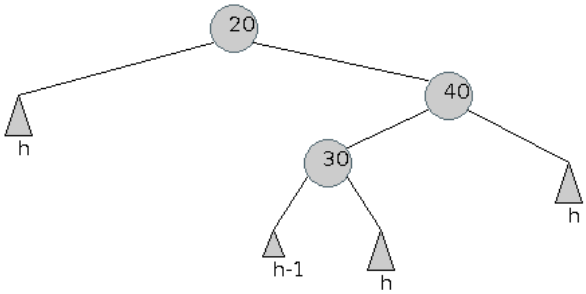
2. Case: +2, -1 (parent is 2 and signs are the different)
    1. Start:



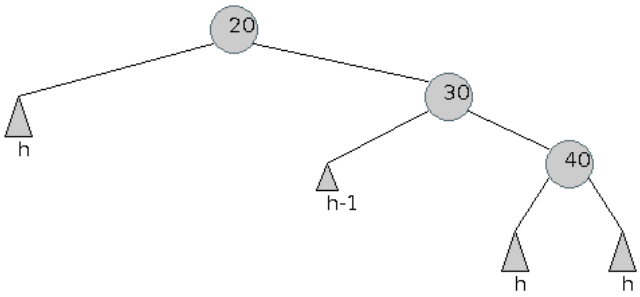
2. Show Node 30



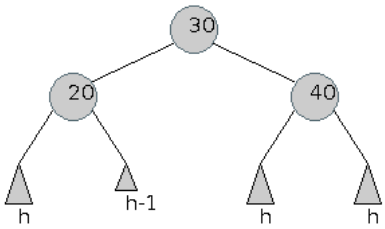
3. Insert value



4. Left Rotation around 30 and 40



5. Right Rotation with Parent as Root



3. There are mirror images of step 3 not shown here

AVL Tree Deletion

- Deletion by copying is prefered because there is less balancing needed.

Splay Trees

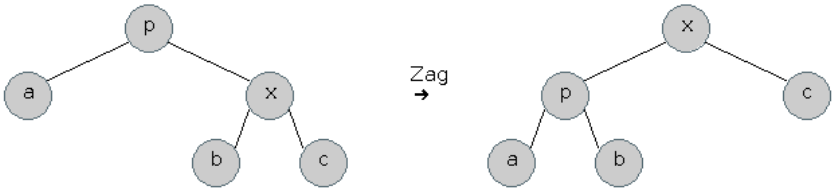
Splay Trees are like self organizing lists, except for trees. The most recently accessed element is moved to the root using tree rotations. [3]

Splay Tree Search and Insertion

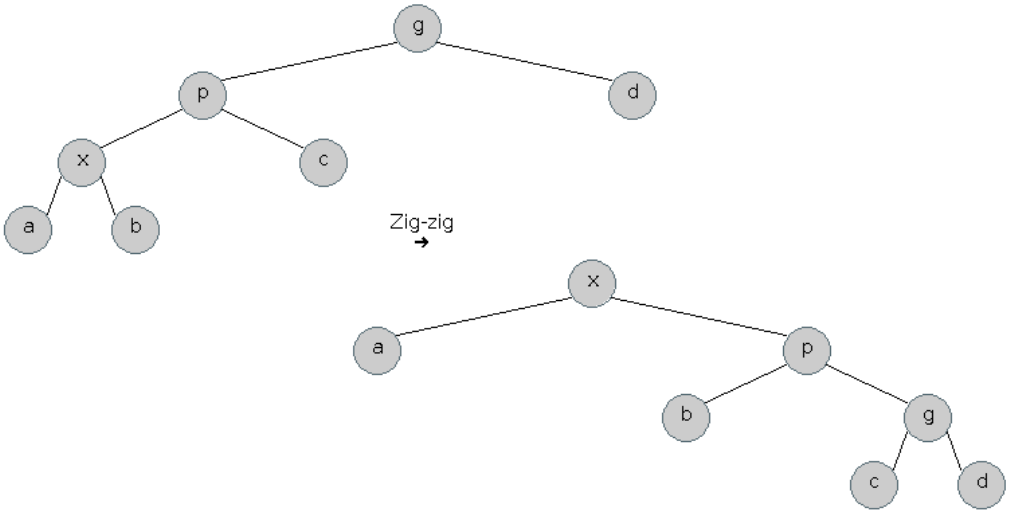
A basic binary search is used to find the node in a search or find the place to insert in insertion. On the way down the recursion, 6 types of rotations are used.

1. Zig Step: The parent is the root and the found node is the left child. A right rotation is done at the root

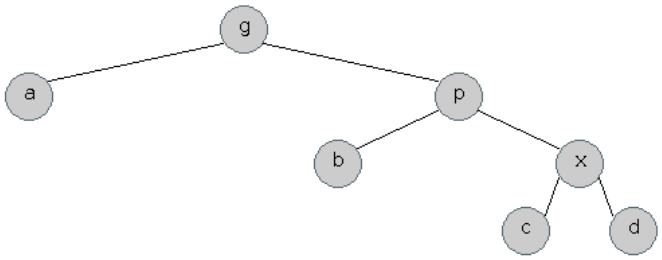
2. Zag Step: The parent is the root and the found node is the right child. A left rotation is done at the root



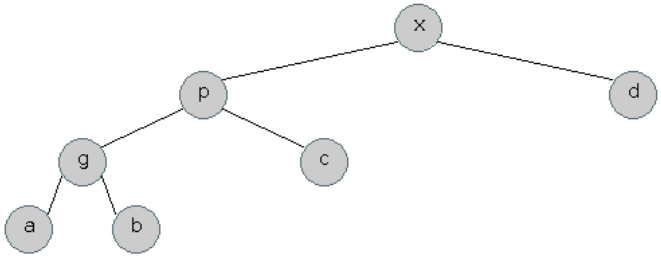
3. Zig-zig Step: The path from the grandparent to the found node was two left pointers. A right rotation is done at the grandparent and parent



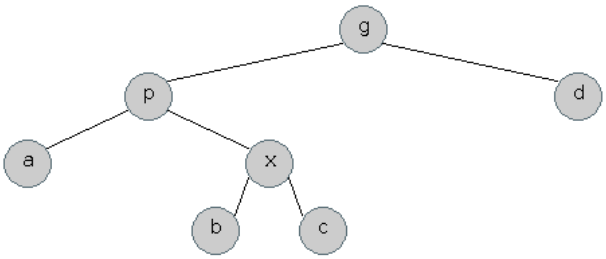
4. Zag-zag Step: The path from the grandparent to the found node was two right pointers. A left rotation is done at the grandparent and parent



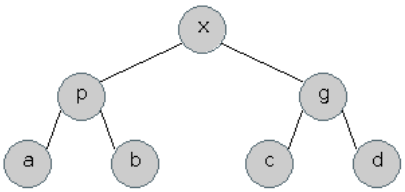
Zag-zag



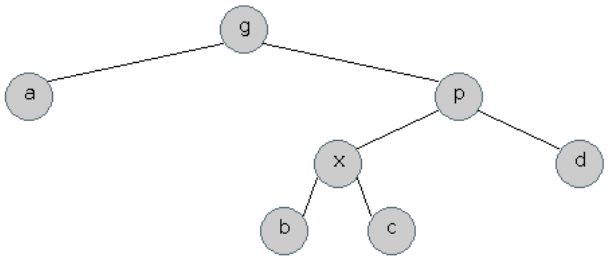
5. Zag-zig Step: The path from the grandparent to the found node was a left pointer followed by a right pointer. A left rotation is done at the parent and a right at the grandparent



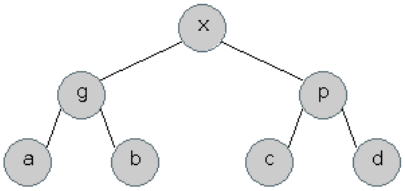
Zag-zig



6. Zig-zag Step: The path from the grandparent to the found node was a right pointer followed by a left pointer. A right rotation is done at the parent and a left at the grandparent



Zig-zag
➔



Splay Tree Deletion

One method of Splay Tree Deletion is as follows:

1. Splay the node to be deleted to the top
2. Make the left child the root
3. Make the left child's right sub-tree and attach it to the leftmost node of the right child
4. Make the right child the new right child of the new root

References

1. http://en.wikipedia.org/wiki/Tree_rotation
2. http://en.wikipedia.org/wiki/AVL_tree
3. http://en.wikipedia.org/wiki/Splay_tree
