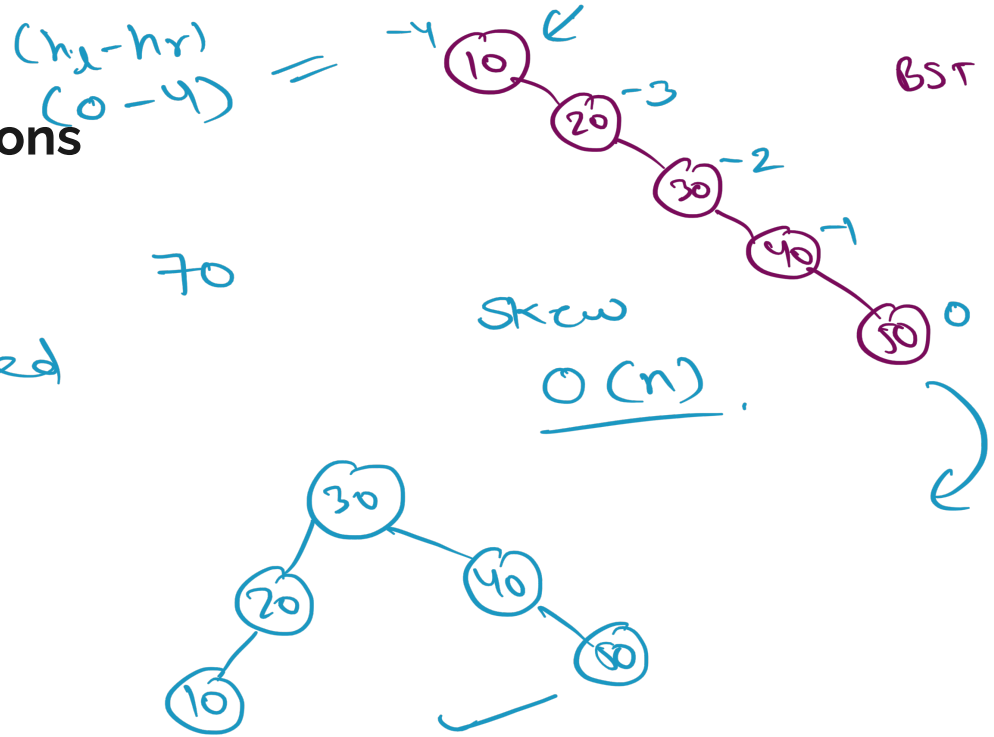
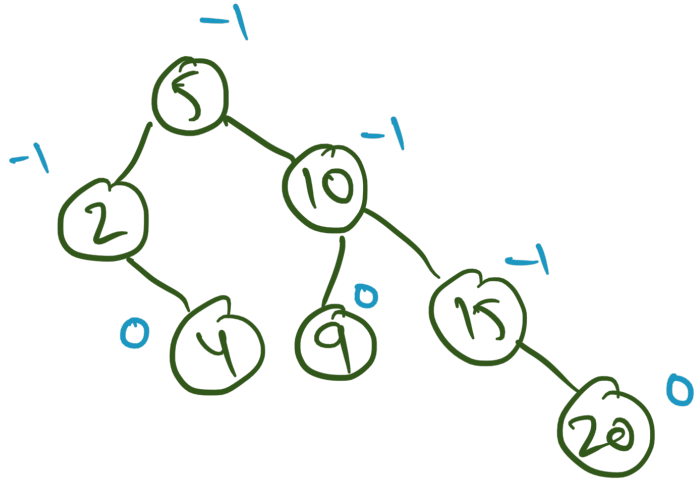
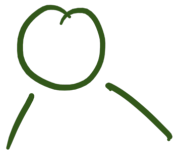
Binary Search Tree - III

AVL Tree Rotations



5, 10, 2, 4, 15, 9, 7, 20

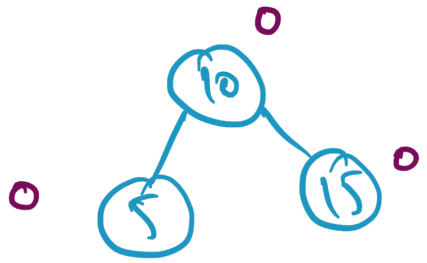


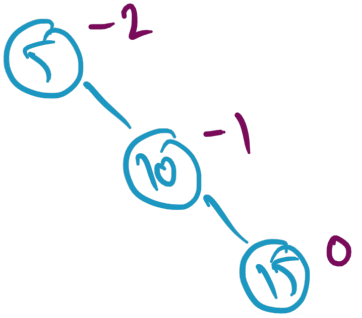


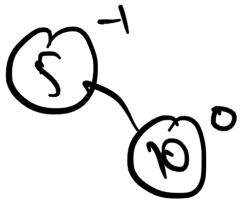
Balanced BST

↳ Every Node should be height-balanced → AVL Tree

$-1 <= h_l - h_r <= 1$ [-1, 0, 1]

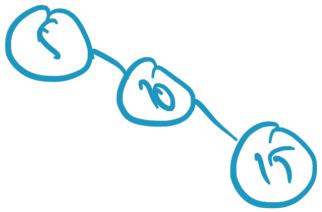




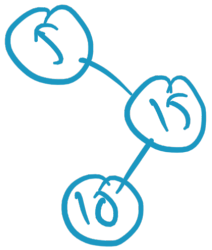


5, 10, 15 : Permutations

5, 10, 15

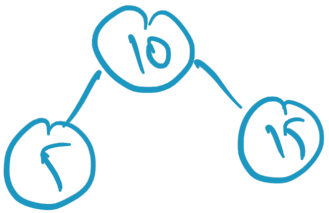


5, 15, 10,

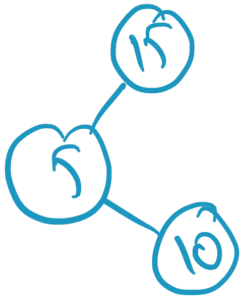


// 10, 5, 15

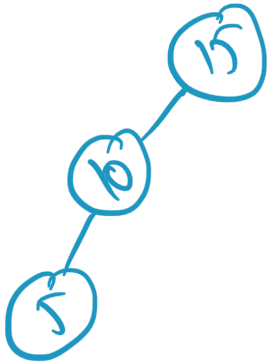
// 10, 15, 5



15, 5, 10,

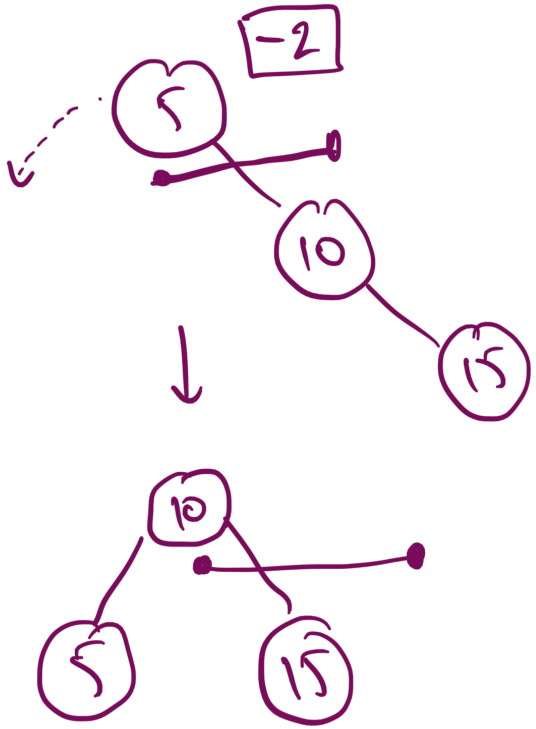


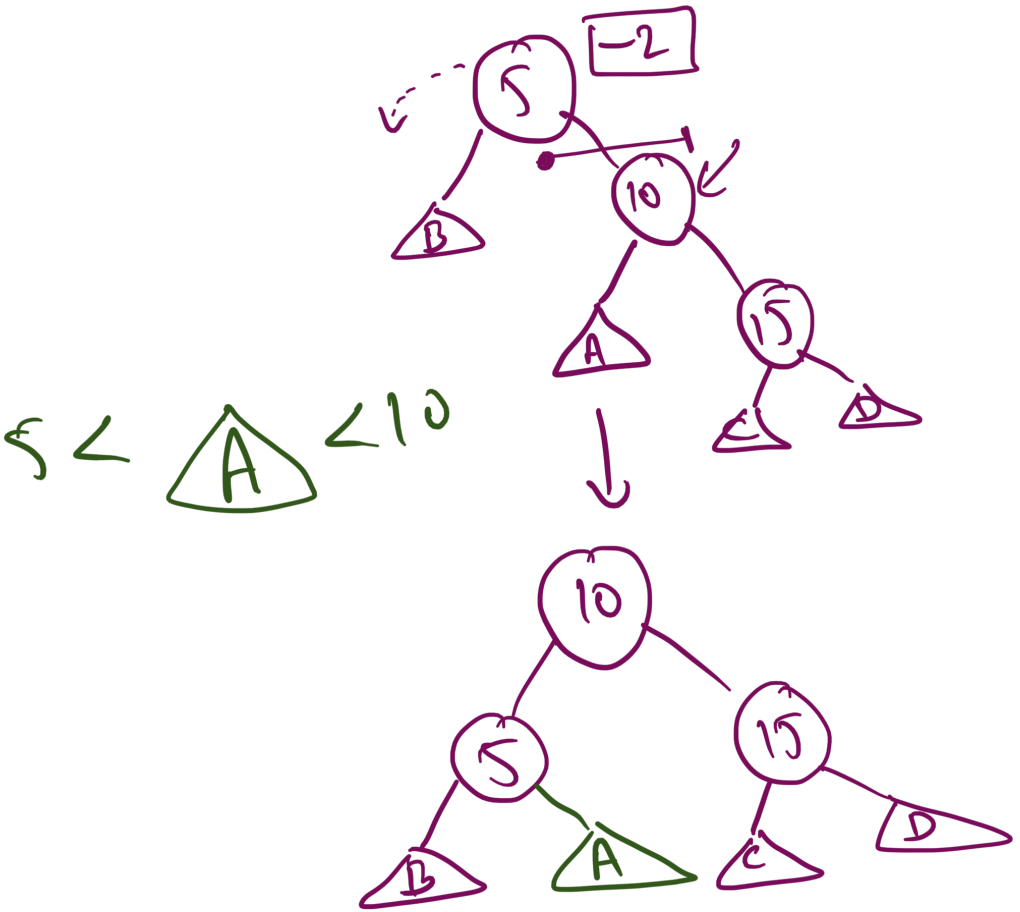
15, 10, 5

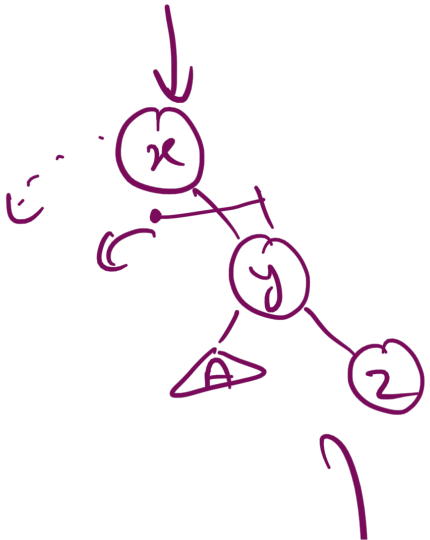


Rotations

Left Rotation







leftRotate (Node x) {

Node y = x.right

Node A = y.left

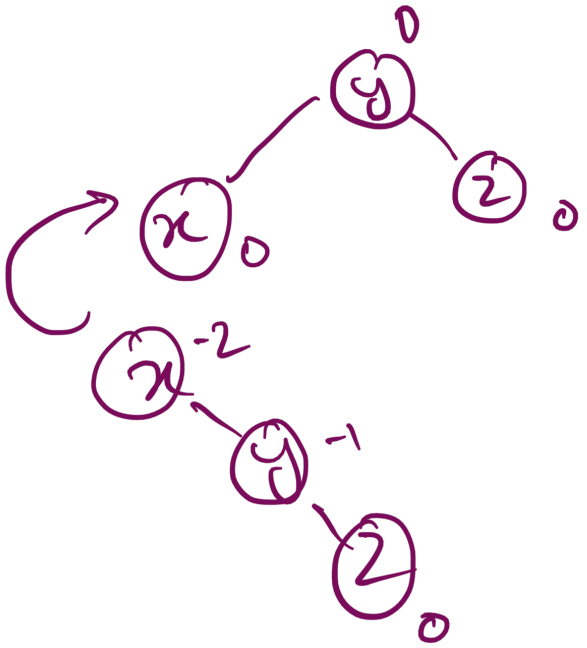
y.left = x

x.right = A

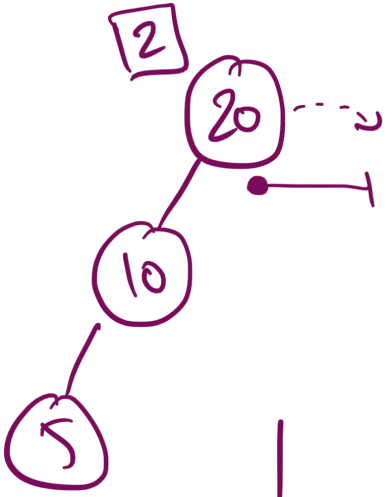
return y;

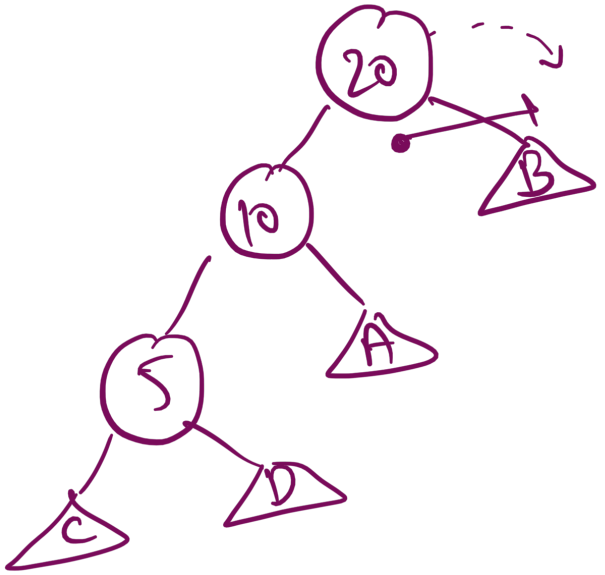
}

rightRotate (x)

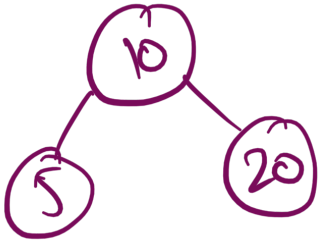


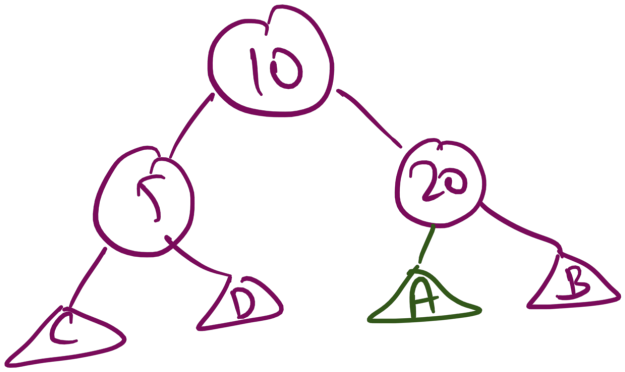
Right Rotation

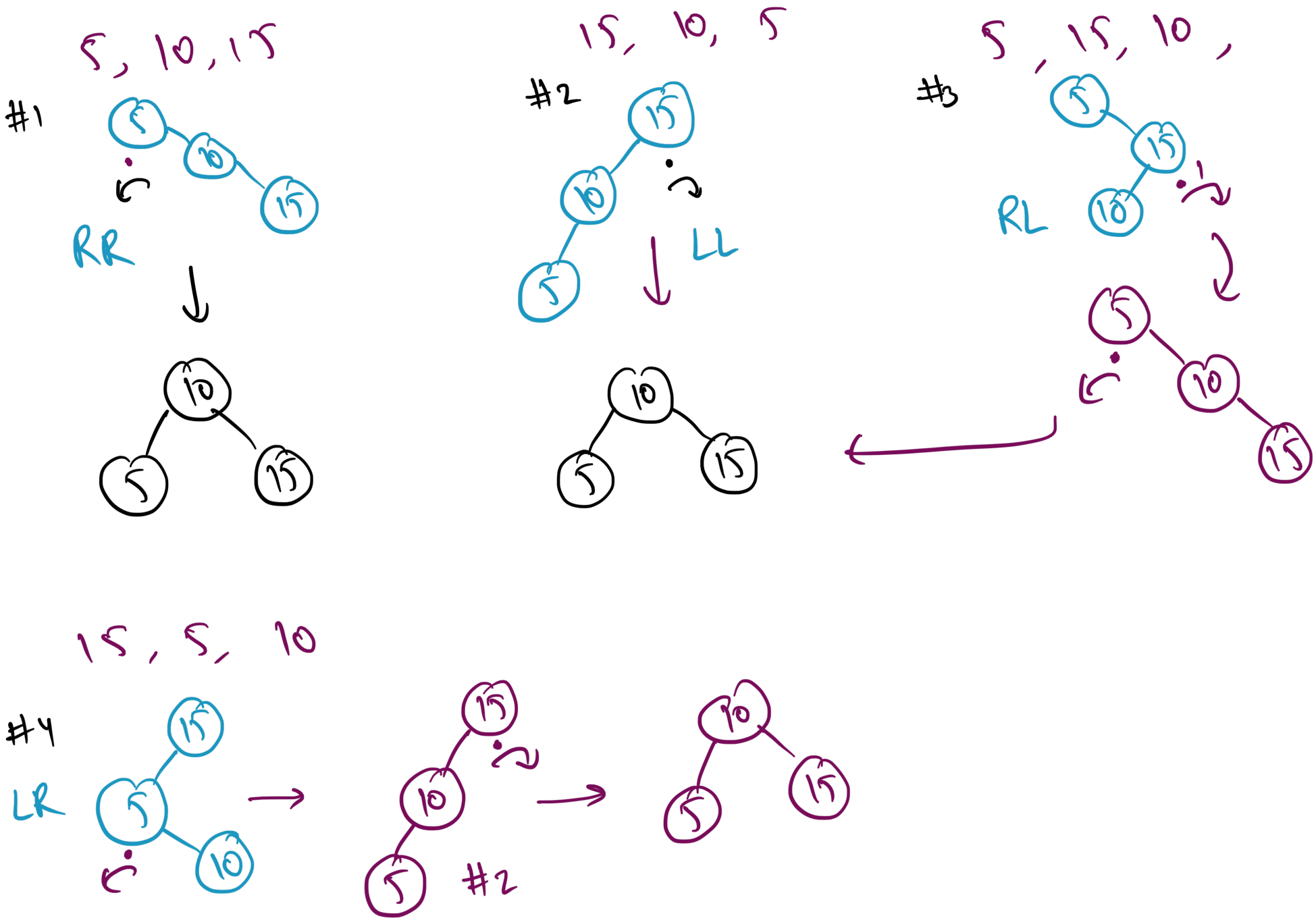


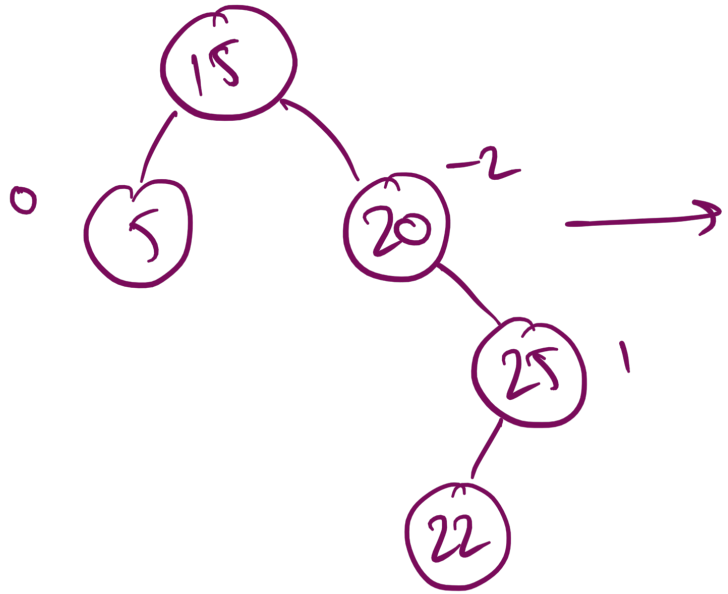


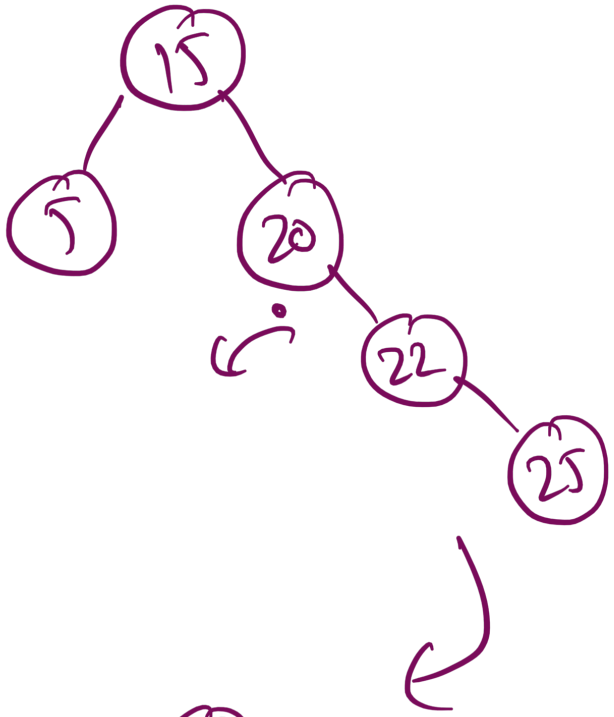
$10 < A < 20$



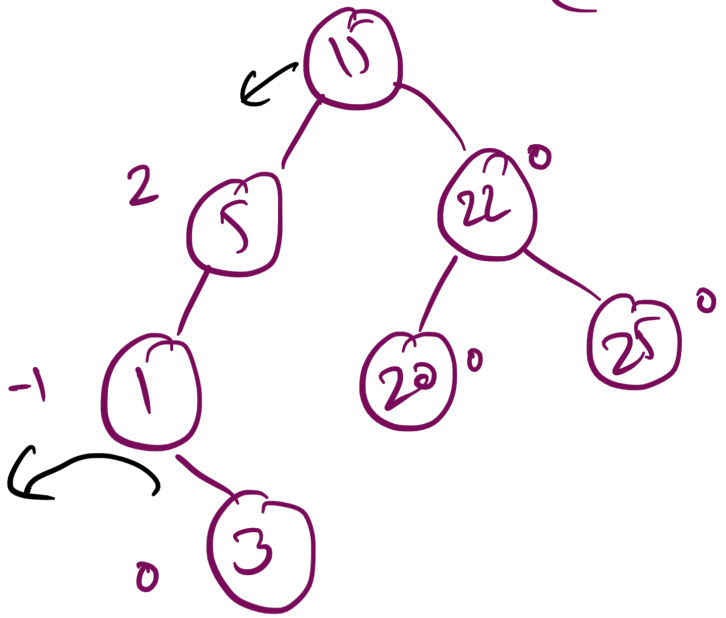



5, 10, 15
#1
RR
15, 10, 5
#2
LL
5, 15, 10
#3
RL
15, 5, 10
#4
LR
#2

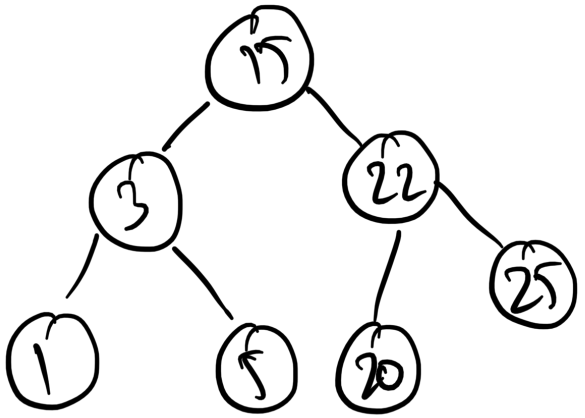




rightRotate (25)
leftRotate (20);



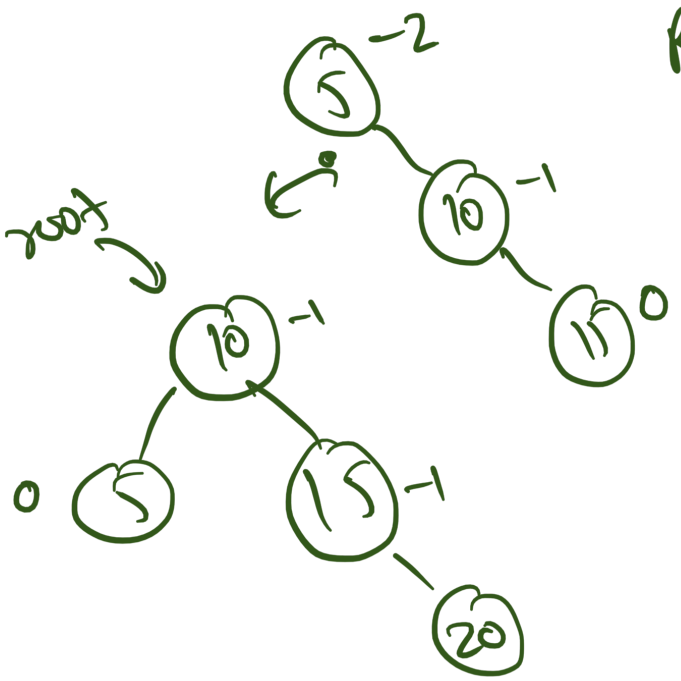
leftRotate (1)
rightRotate (5)



insert ( )

5 , 10, 15 , 20

```
Node {
    int data, height;
    Node left, right;
}
```



RR → Left (Node)

LL → Right (Node)

RL → Right (Node.right)
Left (Node)

LR → Left (Node.left)
Right (Node)

AVL Tree Insertion

AVL Tree Deletion

Practice Problems

1. https://www.interviewbit.com/courses/programming/tree-data-structure
2. https://www.geeksforgeeks.org/binary-search-tree-data-structure/

AVL Tree | Set 1 (Insertion)

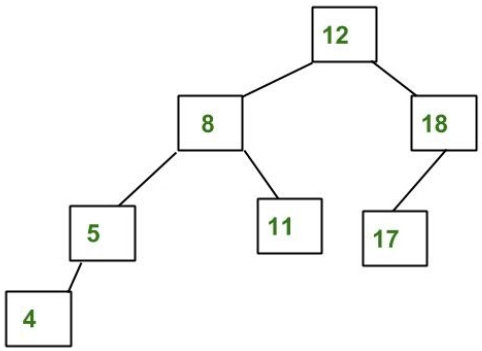
Difficulty Level : Hard • Last Updated : 22 Aug, 2022

Read Discuss

AVL Tree:

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

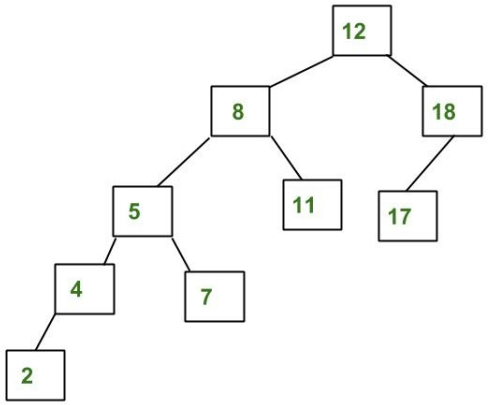
Example of AVL Tree:



No compatible source was found for this media.

The above tree is AVL because the differences between heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

*Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.*

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.
Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

```
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side)

     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation             T2  T3

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```


Steps to follow for insertion:

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:

Following are the possible 4 arrangements:

- y is the left child of z and x is the left child of y (Left Left Case)
- y is the left child of z and x is the right child of y (Left Right Case)
- y is the right child of z and x is the right child of y (Right Right Case)
- y is the right child of z and x is the left child of y (Right Left Case)

*Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to **re-balance** the subtree rooted with **z** and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with **z** becomes the same as it was before insertion.*

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

2. Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - - ->  / \    / \
T1   x                         y    T3                    T1  T2 T3  T4
    / \                       / \
  T2   T3                   T1   T2
```

3. Right Right Case

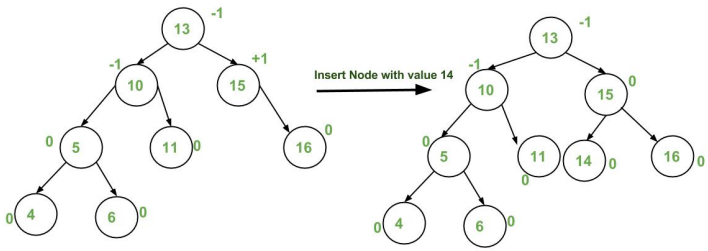
```
  z                                y
 /  \                            /   \
```

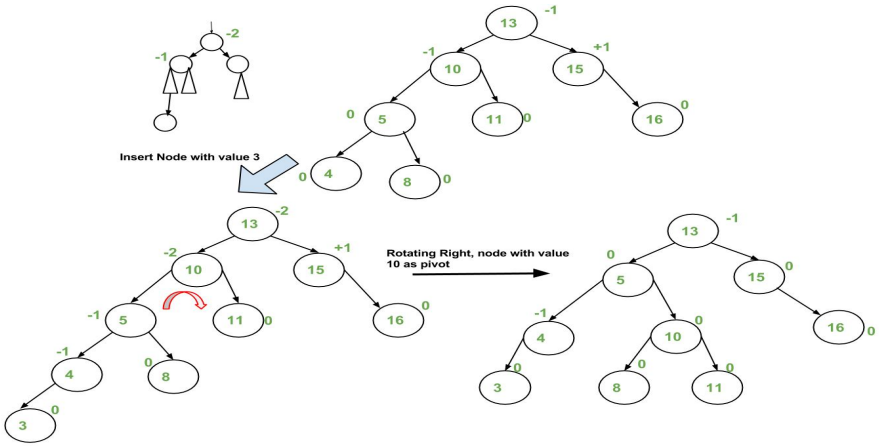
```
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - - ->    / \    / \
   T2   x                      T1  T2 T3  T4
       / \
     T3  T4
```

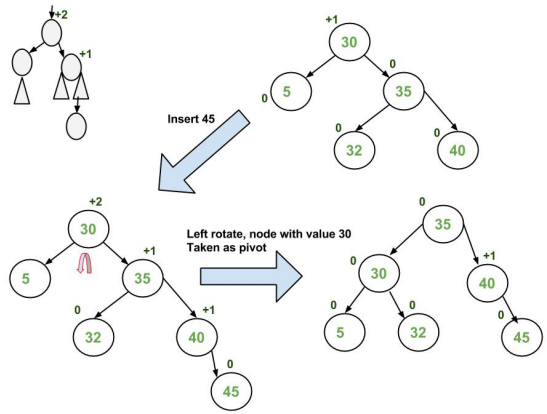
4. Right Left Case

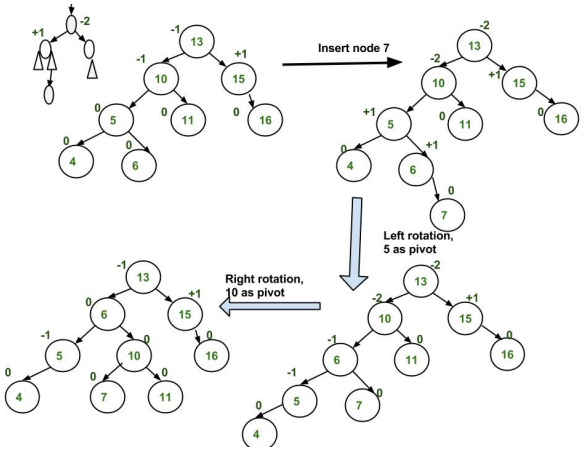
```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - - ->  / \    / \
   x   T4                      T2   y                   T1  T2  T3  T4
  / \                              /  \
T2   T3                          T3   T4
```

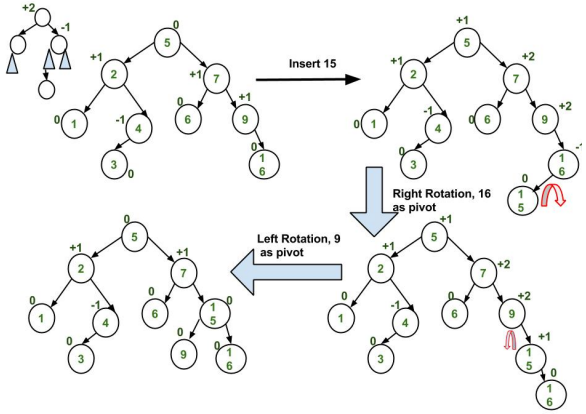
Illustration of Insertion at AVL Tree




Insert Node with value 3
Rotating Right, node with value
10 as pivot


Insert 45
Left rotate, node with value 30
Taken as pivot





Approach:

> *The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.*

Follow the steps mentioned below to implement the idea:

- Perform the normal BST insertion.

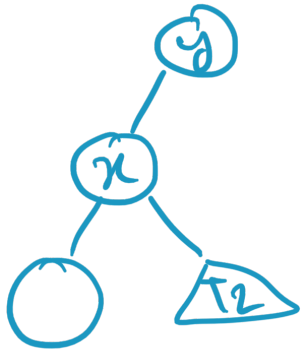
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.
- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.

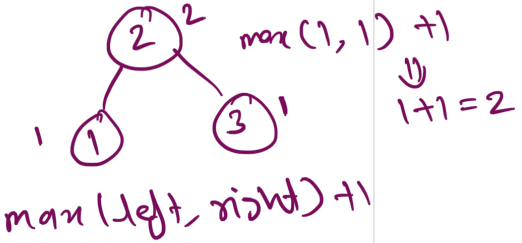
Below is the implementation of the above approach:

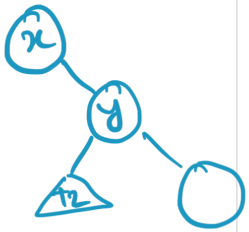
C++ C Java Python3 C# Javascript

```
// C++ program to insert a node in AVL tree
#include<bits/stdc++.h>
using namespace std;

// An AVL tree node
class Node
{
    public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// A utility function to get maximum
// of two integers
int max(int a, int b);

// A utility function to get the
// height of the tree
int height(Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum
// of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}
```

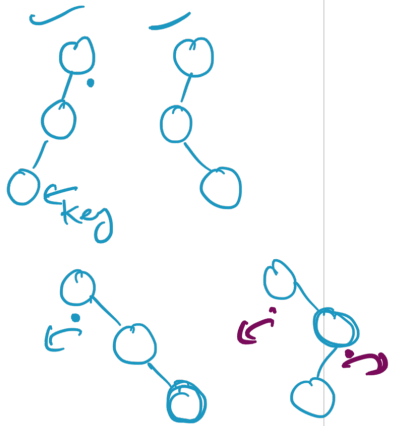
```
/* Helper function that allocates a
   new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return(node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;
```







```
    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key
// in the subtree rooted with node and
// returns the new root of the subtree.
Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));

    /* 3. Get the balance factor of this ancestor
        node to check whether this node became
        unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```



```
    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main()
{
    Node *root = NULL;

    /* Constructing tree given in
    the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
                30
            / \
            20 40
            / \ \
        10 25 50
    */
    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);

    return 0;
}
```

```
// This code is contributed by
// rathbhupendra
```

Output:

```
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50
```

Complexity Analysis

Time Complexity: O(log(n)), For Insertion
Auxiliary Space: O(1)

The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the AVL insert remains the same as the BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Comparison with Red Black Tree:

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then the AVL tree should be preferred over Red Black Tree.

Following is the post for deletion in AVL Tree:
AVL Tree | Set 2 (Deletion)

Previous
Threaded Binary Search Tree | Deletion

Next
How to handle duplicates in Binary Search Tree?

RECOMMENDED ARTICLES

Page : 1 2 3

19, Jan 18

04 **Red Black Tree vs AVL Tree**
27, May 18

08 **How to insert Strings into an AVL Tree**
11, Oct 21

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Hard

Easy | Normal | Medium | Hard | Expert

Improved By : princiraj1992, rathbhupendra, Akanksha_Rai, sohamshinde04, nocturnalstoryteller, rdtank, kaiwenzheng644, hardikkoriintern, garvitpr1hev

Article Tags : Amazon, AVL-Tree, Citicorp, Informatica, MakeMyTrip, Morgan Stanley, Oracle, Oxigen Wallet, Self-Balancing-BST, Snapdeal, Advanced Data Structure, Binary Search Tree, Tree

Practice Tags : Amazon, Citicorp, Informatica, MakeMyTrip, Morgan Stanley, Oracle, Oxigen Wallet, Snapdeal, AVL-Tree, Binary Search Tree, Tree

Improve Article | Report Issue

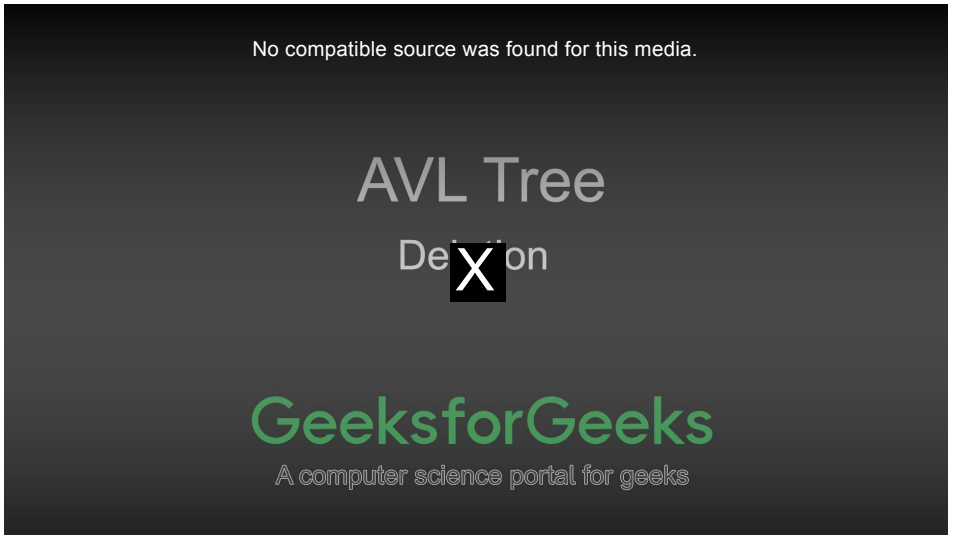
AVL Tree | Set 2 (Deletion)

Difficulty Level : Hard • Last Updated : 17 Jun, 2022

Read Discuss

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).



1. Left Rotation

```
of x (on right side)
               y                                  x
              / \     Right Rotation             /  \
             x   T3   - - - - - - - >          T1   y
            / \       < - - - - - - -               / \
           T1  T2     Left Rotation               T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

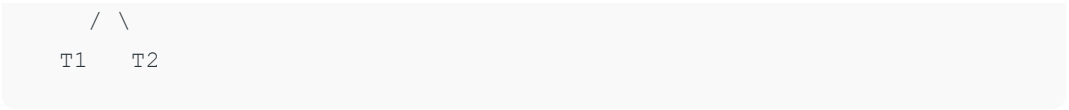
Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   1. y is left child of z and x is left child of y (Left Left Case)
   2. y is left child of z and x is right child of y (Left Right Case)
   3. y is right child of z and x is right child of y (Right Right Case)
   4. y is right child of z and x is left child of y (Right Left Case)

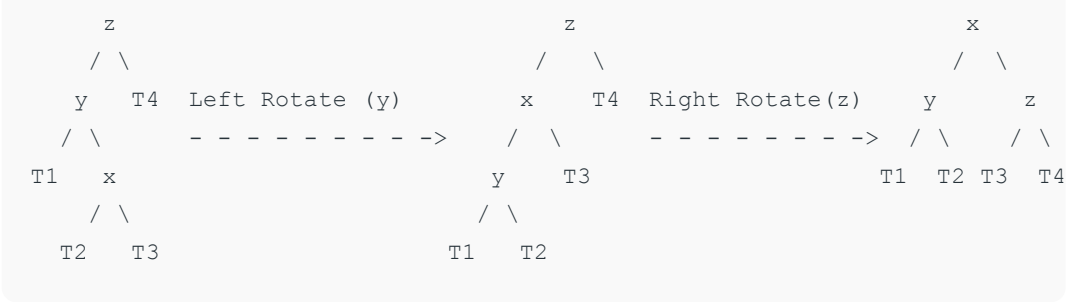
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See this video lecture for proof)

a) Left Left Case

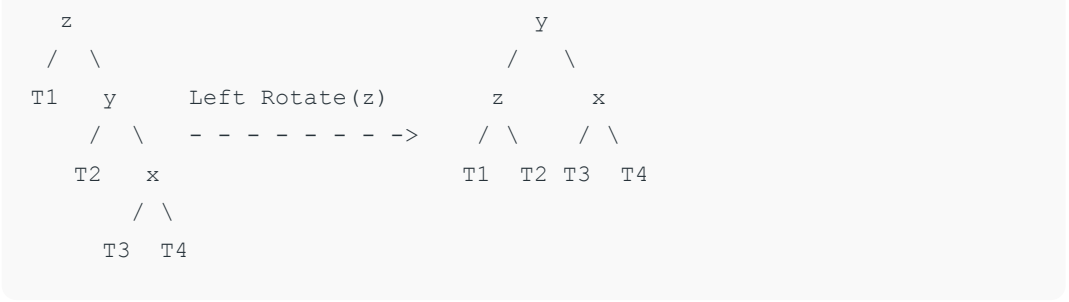
```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
```



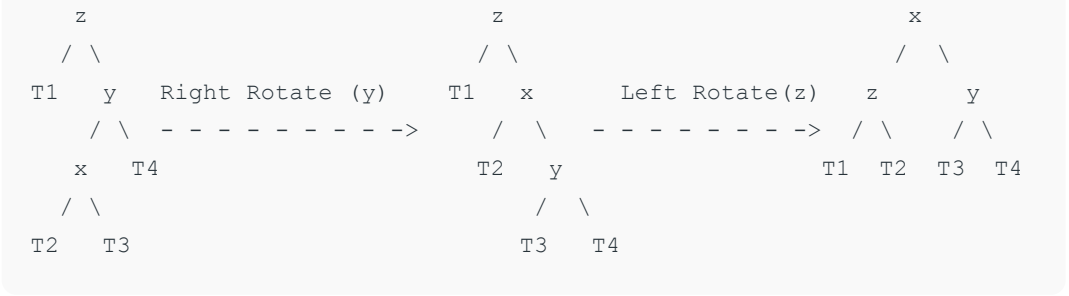
b) Left Right Case



c) Right Right Case



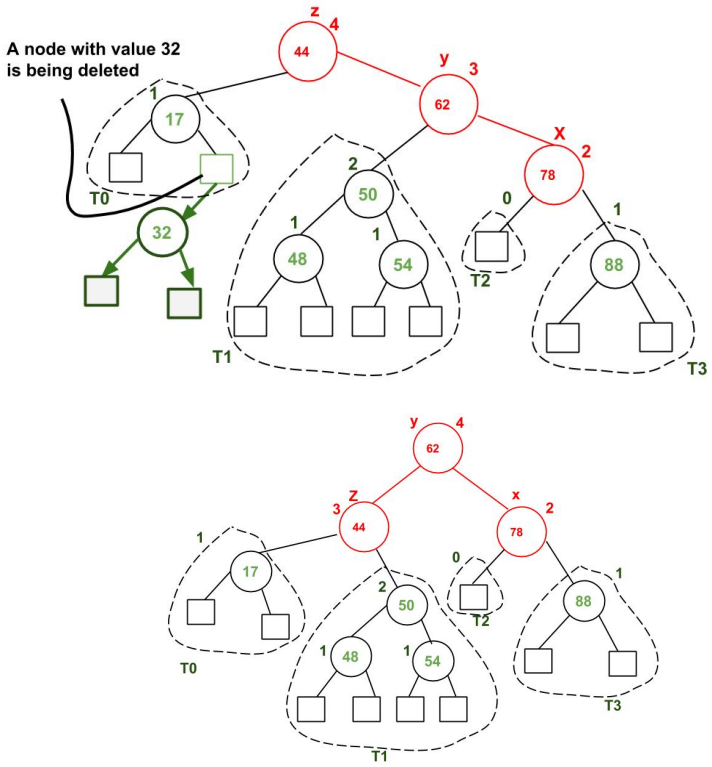
d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example:

Example of deletion from an AVL Tree:



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

Recommended Practice

AVL Tree Deletion

Try It!

C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to

all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.
4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

C++ C Java Python3 C# Javascript

```cpp
// C++ program to delete a node from AVL Tree
#include<bits/stdc++.h>
using namespace std;

// An AVL tree node
class Node
{
    public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// A utility function to get maximum
// of two integers
int max(int a, int b);

// A utility function to get height
// of the tree
int height(Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
```

```
// A utility function to get maximum
// of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a
   new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return(node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
```

```
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) -
           height(N->right);
}

Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));

    /* 3. Get the balance factor of this
        ancestor node to check whether
        this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
```

```
    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree,
return the node with minimum key value
found in that tree. Note that the entire
tree does not need to be searched. */
Node * minValueNode(Node* node)
{
    Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node
// with given key from subtree with
// given root. It returns root of the
// modified subtree.
Node* deleteNode(Node* root, int key)
{

    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller
    // than the root's key, then it lies
    // in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater
    // than the root's key, then it lies
    // in right subtree
    else if( key > root->key )
```

```
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then
    // This is the node to be deleted
    else
    {
        // node with only one child or no child
        if( (root->left == NULL) ||
            (root->right == NULL) )
        {
            Node *temp = root->left ?
                         root->left :
                         root->right;

            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
            *root = *temp; // Copy the contents of
                           // the non-empty child
            free(temp);
        }
        else
        {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            Node* temp = minValueNode(root->right);

            // Copy the inorder successor's
            // data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right,
                                     temp->key);
        }
    }

    // If the tree had only one node
    // then return
    if (root == NULL)
    return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF
    // THIS NODE (to check whether this
    // node became unbalanced)
    int balance = getBalance(root);
```

```
    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 &&
        getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 &&
        getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 &&
        getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 &&
        getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main()
{
Node *root = NULL;

    /* Constructing tree given in
    the above figure */
```

```
        root = insert(root, 9);
        root = insert(root, 5);
        root = insert(root, 10);
        root = insert(root, 0);
        root = insert(root, 6);
        root = insert(root, 11);
        root = insert(root, -1);
        root = insert(root, 1);
        root = insert(root, 2);

        /* The constructed AVL Tree would be
                9
            / \
            1 10
            / \ \
        0 5 11
        / / \
        -1 2 6
        */

        cout << "Preorder traversal of the "
                "constructed AVL tree is \n";
        preOrder(root);

        root = deleteNode(root, 10);

        /* The AVL Tree after deletion of 10
                1
            / \
            0 9
            / / \
        -1 5     11
            / \
            2 6
        */

        cout << "\nPreorder traversal after"
             << " deletion of 10 \n";
        preOrder(root);

        return 0;
    }

    // This code is contributed by rathbhupendra
```

Output:

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
```

```
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n).

Advantages Of AVL Trees

- It is always height balanced
- Height Never Goes Beyond LogN, where N is the number of nodes
- It give better search than compared to binary search tree
- It has self balancing capabilities

Summary of AVL Trees

- These are self-balancing binary search trees.
- Balancing Factor ranges -1, 0, and +1.
- When balancing factor goes beyond the range require rotations to be performed
- Insert, delete, and search time is O(log N).
- AVL tree are mostly used where search is more frequent compared to insert and delete operation.

Previous
Threaded Binary Search Tree |

Next
How to handle duplicates in

Deletion | Binary Search Tree?

RECOMMENDED ARTICLES

Page : 1 2 3

01 Insertion, Searching and Deletion in AVL trees containing a parent node pointer
14, Jul 21

02 Complexity of different operations in Binary tree, Binary Search Tree and AVL tree
19, Jan 18

03 Red Black Tree vs AVL Tree
27, May 18

04 AVL Tree | Set 1 (Insertion)
23, Feb 12

05 Minimum number of nodes in an AVL Tree with given height
30, Oct 18

06 Optimal sequence for AVL tree insertion (without any rotations)
30, Oct 18

07 Implementation of AVL Tree using graphics in C++
11, Jul 21

08 How to insert Strings into an AVL Tree
11, Oct 21

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Hard

Easy | Normal | Medium | Hard | Expert

Improved By : AnkushRodewad, KP1975, princiraj1992, rathbhupendra, KickSam, arijitroy003, decode2207, krishnanand3, guptavivek0503, hardikkoriintern

Article Tags : Amazon, AVL-Tree, MakeMyTrip, Morgan Stanley, Oracle, Oxigen Wallet, Self-Balancing-BST, Snapdeal, Advanced Data Structure, Tree

Practice Tags : Amazon, MakeMyTrip, Morgan Stanley, Oracle, Oxigen Wallet, Snapdeal, AVL-Tree, Tree

Improve Article | Report Issue