

AVL Tree | Set 2 (Deletion)

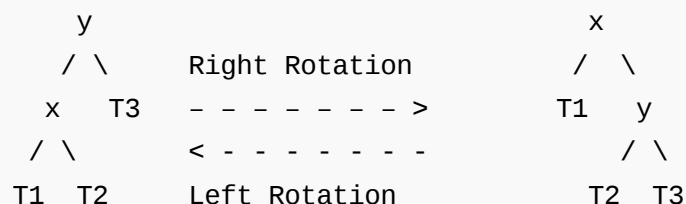
We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

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 ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

- 1) Left Rotation
- 2) Right Rotation

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



Keys in both of the above trees follow the following order

$$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$$

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

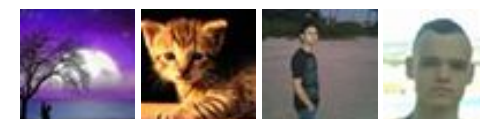
Google™ Custom Search



GeeksforGeeks



52,731 people like [GeeksforGeeks](#).



[Interview Experiences](#)

[Advanced Data Structures](#)

[Dynamic Programming](#)

[Greedy Algorithms](#)

[Backtracking](#)

[Pattern Searching](#)

[Divide & Conquer](#)

[Mathematical Algorithms](#)

[Recursion](#)

[Geometric Algorithms](#)

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.

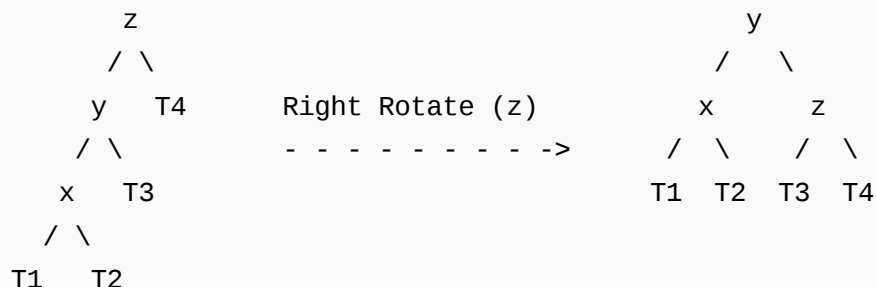
Following are the possible 4 arrangements:

- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)
- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)

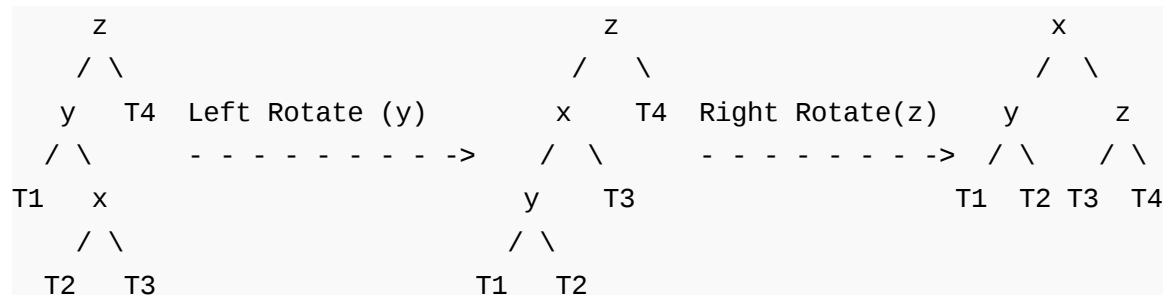
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

a) Left Left Case

T1, T2, T3 and T4 are subtrees.



b) Left Right Case



c) Right Right Case



Popular Posts

[All permutations of a given string](#)

[Memory Layout of C Programs](#)

[Understanding "extern" keyword in C](#)

[Median of two sorted arrays](#)

[Tree traversal without recursion and without stack!](#)

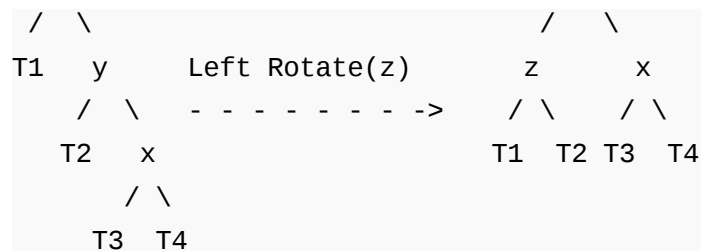
[Structure Member Alignment, Padding and Data Packing](#)

[Intersection point of two Linked Lists](#)

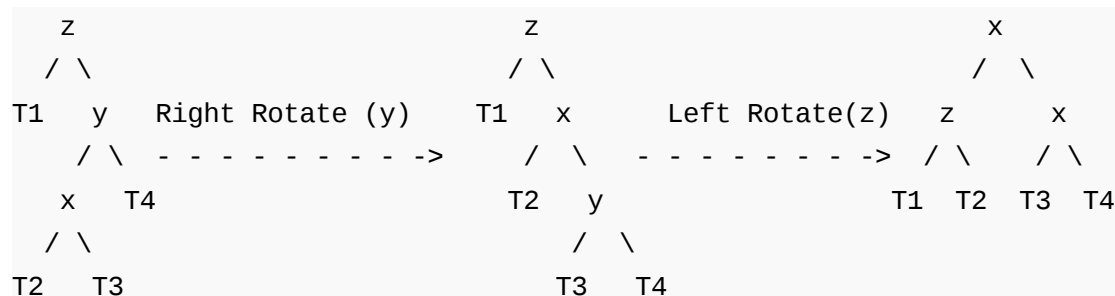
[Lowest Common Ancestor in a BST.](#)

[Check if a binary tree is BST or not](#)

[Sorted Linked List to Balanced BST](#)



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.

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.

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct 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(struct 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. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct 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.
struct node *rightRotate(struct node *y)
```

Recent Comments

affiszerv Your example has two 4s on row 3,
that's why it...

[Backtracking | Set 7 \(Sudoku\)](#) · 37 minutes ago

RVM Can someone please elaborate this Qs
from above...

[Flipkart Interview | Set 6](#) · 57 minutes ago

Vishal Gupta I talked about as an Interviewer
in general,...

[Software Engineering Lab, Samsung Interview | Set 2](#) · 57 minutes ago

@meya Working solution for question 2 of
4f2f round....


[Amazon Interview | Set 53 \(For SDE-1\)](#) · 1 hour ago

sandeep void rearrange(struct node *head)
{...

Given a linked list, reverse alternate nodes and
append at the end · 3 hours ago

Neha I think that is what it should return as,
in...

Find depth of the deepest odd level leaf node · 3
hours ago

AdChoices 

[► Binary Tree](#)

[► Java Tree](#)

[► Tree Balancing](#)

```

{
    struct node *x = y->left;
    struct 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.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct 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(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

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


```

AdChoices 

[► Red Black Tree](#)

[► Tree Structure](#)

[► Ancestor Tree](#)

AdChoices 

[► Tree Root](#)

[► Tree Trees](#)

[► Node](#)

```

        return (newNode (key) );

    if (key < node->key)
        node->left = insert (node->left, key);
    else
        node->right = insert (node->right, key);

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

    /* 3. Get the balance factor of this ancestor node to check whethe
       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. */
struct node * minValueNode (struct node* node)
{

```

```

struct node* current = node;

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

return current;
}

struct node* deleteNode(struct 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) )
        {
            struct 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 (small
        // in the right subtree)
        struct 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 = max(height(root->left), height(root->right)) + 1;

// 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(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main()
{
    struct 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
 */

    printf("Pre order 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
*/

printf("\nPre order traversal after deletion of 10 \n");
preOrder(root);

return 0;
}

```

Output:

Pre order traversal of the constructed AVL tree is

9 1 0 -1 5 2 6 10 11

Pre order 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 $O(\log n)$. So time complexity of AVL delete is $O(\log n)$.

References:

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.



Better Than Hadoop.

HPCC Systems is Big Data Processing and Analytics
Open Source. Proven. Trusted.

 LexisNexis® [Learn More](#) 

Related Topics:

- [Print a Binary Tree in Vertical Order | Set 2 \(Hashmap based Method\)](#)
- [Print Right View of a Binary Tree](#)
- [Red-Black Tree | Set 3 \(Delete\)](#)
- [Construct a tree from Inorder and Level order traversals](#)
- [Print all nodes at distance k from a given node](#)
- [Print a Binary Tree in Vertical Order | Set 1](#)
- [Interval Tree](#)
- [Check if a given Binary Tree is height balanced like a Red-Black Tree](#)



10



Tweet 0



2

Writing code in comment? Please use [ideone.com](#) and share the link here.

16 Comments

GeeksforGeeks

Sort by Newest ▼





...on the discussion...



ArafatX · 4 months ago

If anyone here want to have the complete source code with this functionality, p
twitter.com/arafatx and pm me there.

[img] <http://codegix.com/imagex/avll...> [/img]

Credit to geeksforgeeks to with the precious explanation in order to create this

^ | v · Reply · Share ›



mallard · 4 months ago

can't we check the check left->left->right etc case of deleted nodes just like w

^ | v · Reply · Share ›



ArafatX · 4 months ago

I believe this code has error but I'm not sure what (trying to figure out) For exam
sequence 10, 20, 30, 40, 50, 60, 70, 80, 90, 99. And we delete 40 (the root), Th
30 , 20 , 10 , 80 , 60, 50, 70, 90, 99. But the result from your code: 50, 20, 10, :

Edit: It's ok. Now I understand. There are 2 ways of deleting the AVL tree. by c

^ | v · Reply · Share ›



Guest → **ArafatX** · 4 months ago

Pardon me, now I understand. There are 2 ways of deleting. :)

^ | v · Reply · Share ›



Dimitris S. · 5 months ago

Hey, one small question:

What do the different height values stand for?

- Does height == 1 mean "the node is balanced" e.g. a node with no children c

themselves.

- What height would a node that is left-balanced have, for example a root with
- In relation to the upper bulletpoint, what value of height would a root with just

^ | v • Reply • Share ›



Castle Age → Dimitris S. • 2 months ago

Height is the height of the current subtree. Leaf nodes always have a height of 1. When the difference between the height of the left subtree and the height of the right subtree is -1 and 1.

You need to reread every thing about trees and their definitions.

^ | v • Reply • Share ›



Jaini • 8 months ago

No doubt, this tutorial is awesome but to make it more self explanatory, should we include subtrees in diagrams?

^ | v • Reply • Share ›



abhishek08aug • a year ago

Intelligent :D

^ | v • Reply • Share ›



Pranshu Tomar • a year ago

not bad...

^ | v • Reply • Share ›



Doubt • a year ago

When you delete the node , don't you need to reset it's parent's pointer which |

/* Paste your code here (You may delete these lines if not writing code)

^ | v • Reply • Share ›



GeeksforGeeks → Doubt • a year ago

This is handled in as we have recursive code and returned value is as:
a closer look at following lines

```
// 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);
```

^ | v • Reply • Share ›



Julian • a year ago

Your deletion code seems to only check for imbalance on the root node and do not check for all ancestors of the replaced node. However, wikipedia says you need to do it for all ancestors of the replaced node. Is this? because I'm not seeing it.

3 ^ | v • Reply • Share ›



Castle Age → Julian • 2 months ago

The power of recursion. For every recursive call, it checks the balance of the subtree. If it is unbalanced and then move up.

^ | v • Reply • Share ›



mallard → Julian • 4 months ago

we are backtracking when we have deleted the node, hence we are clear and do appropriate

^ | v • Reply • Share ›



BlackMath • 2 years ago

Something wrong with the code.

At the line while deleting the node, in the case of two children :

```
// node with two children: Get the inorder successor (smallest
// in the right subtree)
struct node* temp = minValueNode(root->right);
```

What if the right subtree is null ?

when the right subtree of a node is null and we want to find the successor, we parent which is a left child of its parent.

Code for minValueNode is not handling this case and if we call minValueNode will throw null pointer exception :

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

Please correct me is i am wrong.

```
/* Paste your code here (You may delete these lines if not writing cor
```

```
/* Paste your code here (You may delete these lines if not writing cor
```

^ | v • Reply • Share ›



GeeksforGeeks → BlackMath • 2 years ago

@BlackMath:

We travel up to find the inorder successor only when right subtree of the node is not null. In the `minValueNode()` function, we only find the left child of the node until we reach a node with no left child. This node is the inorder predecessor, not the successor. To find the inorder successor, we need to find the left child of the right child of the node.

Following link can also be useful:

<http://www.geeksforgeeks.org/archives/9999/comment-page-1#comment-9999>

^ | v • Reply • Share ›



Subscribe



Add Disqus to your site