C Code For AVL Tree Insertion & Rotation (LL, RR, LR & RL Rotation)

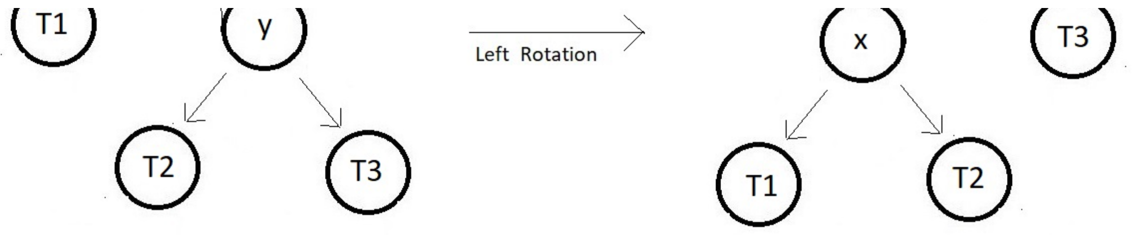
In the last lecture, we saw some good examples and complex ones to fix violations that arise after insertion operation using different rotation techniques. We learned using all the different types of strategies we use in AVL trees to balance an unbalanced tree caused after the four types of rotation situations. Today, we'll see the implementation of the same in C language.

So, before we proceed with the programming implementation of the things we learned, let's first give ourselves a very quick revision of a few things.

The two basic Rotate operations were:

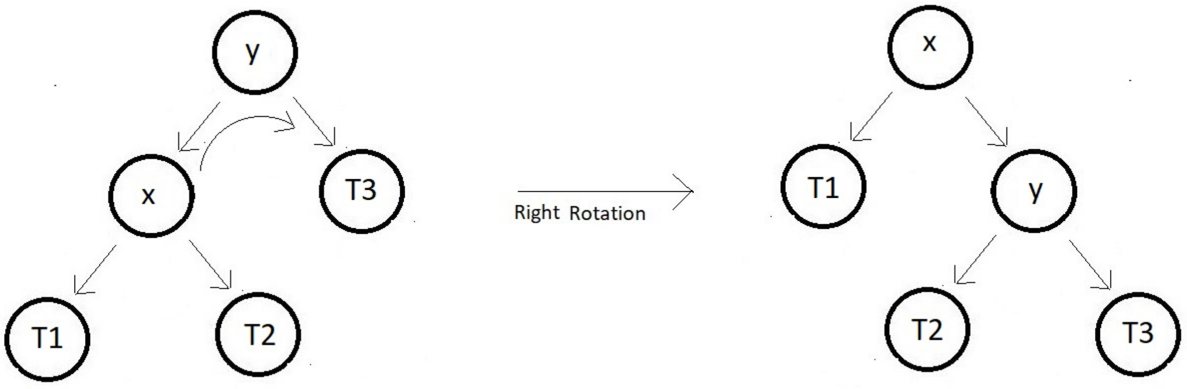
1. Left Rotate Operations.

Here, we move our unbalanced node to the left. Follow the example below.



2. Right Rotate Operation

Here, we move our unbalanced node to the right. Follow the example below.



We'll use the same tree we have illustrated above to guide us through our implementation of left and right rotation functions. We have already learnt in the last lecture the situations we had to use these two rotations techniques in. When you study the code, keep these illustrations in mind.

So, let's just move on to our editors without any further ado. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. The first thing would be to create a struct Node which we earlier used to create a node in a tree. So, create a struct Node. Inside the structure, we would have four embedded elements, first is an integer variable *data* to store the data of the node, second is a struct Node pointer called *left* to store the address of the left child node, and third is again a struct Node pointer called *right* to store the address of the right child node. Fourth is an integer variable storing the height of that node for the ease of calculating the absolute balance factor. Absolute balance factor, if you remember, should not exceed 1, otherwise, that node becomes imbalanced.

```
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};
```

Code Snippet 1: Creating the struct Node

Creating the getHeight function:

2. Create an integer *getHeight,* and pass the pointer to the struct node you want to get the height of as the only parameter. Check if the struct node pointer is not NULL, return 0 if it is NULL, otherwise return the height element of the struct node. And that would be it.

```
int getHeight(struct Node *n){
    if(n==NULL)
        return 0;
    return n->height;
}
```

Code Snippet 2: Creating the getHeight function

Creating the createNode function:

3. Create a struct Node* function *createNode,* and pass the data/key for the node as the only parameter. Create a struct Node pointer *node*. Reserve a memory in the heap for the *node* using malloc. Make the left and the right element of the struct *node* point to NULL, and fill the data in the data element. Now, set the height element of this *node* as 1, because any new node would be a leaf node that has a height of 1. And now, return the pointer *node* you created. This would simply create a node as a part of the tree.

```
struct Node *createNode(int key){
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}
```

Code Snippet 3: Creating the createNode function

Here come the three most important functions of all when we implement the AVL tree. First is the getBalanceFactor, the second is the leftRotate and the third is the rightRotate.

Creating the getBalanceFactor function:

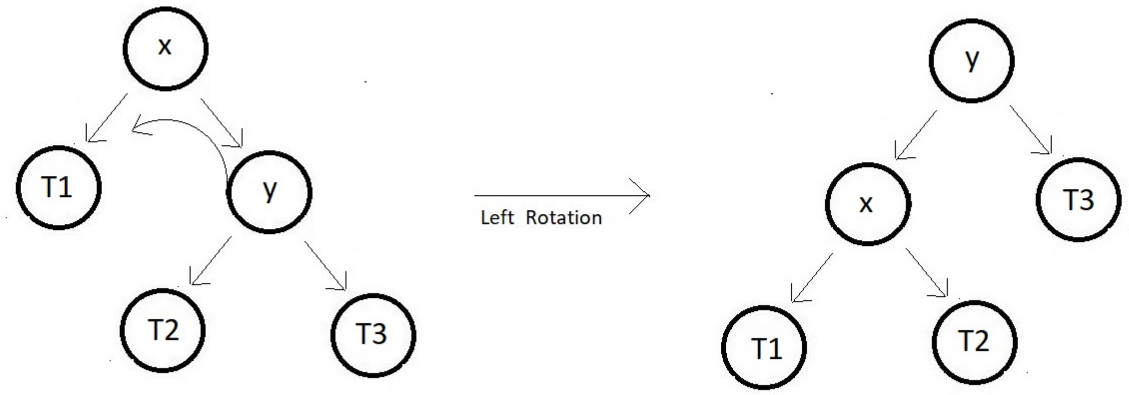
4. Create a struct Node* function *getBalanceFactor,* and pass the pointer to the struct node you want to get the balance factor of as the only parameter. Now the balance factor of a node is just the value we receive after subtracting the height of the right child with that of the left child of the node. So, first check whether the node is not NULL, if it is, simply returns 0. Otherwise, return the value received after subtracting the height of the right child with that of the left child. This was a utility function to find the balance factor for a node.

```
int getBalanceFactor(struct Node * n){
    if(n==NULL){
        return 0;
    }
    return getHeight(n->left) - getHeight(n->right);
}
```

Code Snippet 4: Creating the getBalanceFactor function

Follow the illustrations below while we write the left and the right rotate functions respectively.

Creating the leftRotate function:

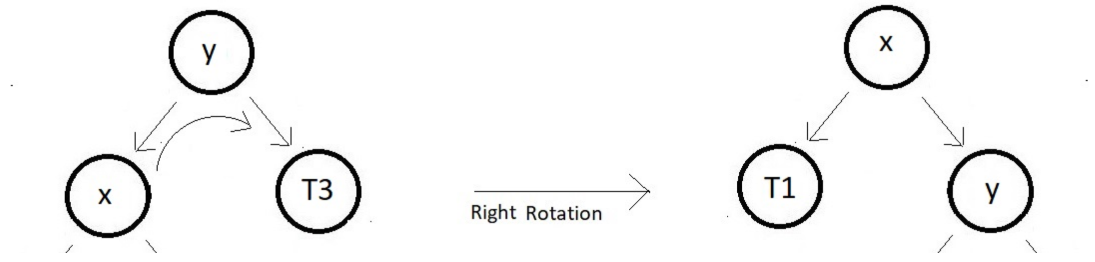


5. Create a struct Node* function leftRotate*,* and pass the pointer to the imbalanced struct node you want to left rotate as the only parameter. Consider this node *x* as represented in the figure above. As you could see, the left child of *x* is *y.* So, define a struct node pointer variable y and initialize it with *x->right*. As seen in the figure, we would make *x* the left child of *x,* and T2 the right child of *x.* So, we would also store *y->left* in another struct node pointer variable T2.
Now, simply assign *x* to the left element of *y,* and T2 to the right element of *x.* And this would finish our left rotation function. We must, however, remember to change the new heights of the nodes that have been shifted, which are *x and y*. Height of *x* becomes one added to the maximum of the heights of both of its children. Similarly, the height of *y* becomes one added to the maximum of the heights of both of its children. And now simply return *y,* since *y* becomes the new root for the subtree. Don't forget to explicitly define a max function before using it.

```
struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return y;
}
```

Code Snippet 5: Creating the leftRotate function

Creating the rightRotate function:



6. Create a struct Node* function rightRotate, and pass the pointer to the imbalanced struct node you want to right rotate as the only parameter. Consider this node *y* as represented in the figure above. As you could see, the left child of *y* is *x*. So, define a struct node pointer variable x and initialize it with *y->left*. As seen in the figure, we would make *y* the right child of *x*, and T2 the left child of *y*. So, we would also store *x->right* in another struct node pointer variable T2.
Now, simply assign *y* to the right element of *x*, and T2 to the left element of *y*. And this would finish our right rotation function. In addition, we must remember to adjust the new height for the nodes that have been shifted, which are *x and y*. Height of *x* becomes one added to the maximum of the heights of both of its children. Similarly, the height of *y* becomes one added to the maximum of the heights of both of its children. And now simply return *x*, since *x* becomes the new root for the subtree.

```
struct Node* rightRotate(struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return x;
}
```

Code Snippet 6: Creating the rightRotate function

Creating the insert function:

7. Create a struct Node* function *insert*, and pass the pointer to the root of the AVL tree and the data we want to insert as two of its parameters. Now, we would follow

the simple Binary Search Tree insertion operation which we had already covered in our past lectures. There, we would just recursively iterate to the best-fit position where the new data should be inserted, and once found, we create a new node, and return the pointer to this node.
And since we have recursively gone down the tree, we would backtrack after inserting. While backtracking, we update the heights of all the nodes we followed to reach the apt position.
Once this new node gets inserted, the balance of the tree might get disturbed, so we would first create an integer variable *bf*, and store in it the balance factor of the current node we are on while backtracking. And if this node gets imbalanced, four of the possible cases arise.
Left-Left case:
Here, the new node would have been inserted on the left to the left child of the current node. So, if our *bf* has a value greater than 1, and the new node has data less than the data of the left child of the node itself, it is the case of left-left rotation, and hence we call the rightRotate function once to fix this disturbance.

Right-Right case:
Here, the new node would have been inserted on the right to the right child of the current node. So, if our *bf* has a value less than -1, and the new node has data greater than the data of the right child of the node itself, it is the case of right-right rotation, and hence we call the leftRotate function once to fix this disturbance.
Left-Right case:
Here, the new node would have been inserted on the right to the left child of the current node. So, if our *bf* has a value greater than 1, and the new node has data greater than the data of the left child of the node itself, it is the case of left-right rotation, and hence we call first the leftRotate function passing the left subtree and then call the rightRotate function on this updated node to fix this disturbance.
Right-Left case:
Here, the new node would have been inserted on the left to the right child of the current node. So, if our *bf* has a value less than -1, and the new node has data less than the data of the right child of the node itself, it is the case of right-left rotation, and hence we call first the rightRotate function passing the right subtree and then

call the leftRotate function on this updated node to fix this disturbance.
And then finally return this node.
8. The Last step would be to fetch the preOrder traversal function from our previous programming lecture, to actually see how our tree would get all balanced after all the insertions we would do.

```
struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return  createNode(key);

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

    node->height = 1 + max(getHeight(node->left), getHeight(node->ri
    int bf = getBalanceFactor(node);

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

Code Snippet 7: Creating the insert function

Here is the whole source code:

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

struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

int getHeight(struct Node *n){
    if(n==NULL)
        return 0;
    return n->height;
}

struct Node *createNode(int key){
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

int max (int a, int b){
    return (a>b)?a:b;
}

int getBalanceFactor(struct Node * n){
    if(n==NULL){
        return 0;
    }
    return getHeight(n->left) - getHeight(n->right);
}

struct Node* rightRotate(struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return x;
}

struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return y;
}
```

```
struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return  createNode(key);

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

    node->height = 1 + max(getHeight(node->left), getHeight(node->ri
    int bf = getBalanceFactor(node);

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

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

int main(){
    struct Node * root = NULL;


    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);
    preOrder(root);
    return 0;
}
```

Code Snippet 8: Implementing AVL tree and its insertion operation

And now having finished all the functions we needed, we can create an AVL tree in main and verify if all these functions actually work. So, simply create a struct node pointer root and initialize it with NULL. And now we are good to insert as many nodes as we want. We would insert 6 of them. And then call the preOrder traversal on the same.

```
    struct Node * root = NULL;


    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);
    preOrder(root);
```

Code Snippet 9: Creating an AVL tree and using the insert function

The output we received was:

```
4 2 1 3 5 6
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, our AVL tree is actually working all fine. And this was the implementation of the rotations we learned. I know this was something actually I would grade tough, but giving respect to the complexity of the topic, I would recommend you all to go through these again and practice writing the code yourselves.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist on DSA yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial. Till then keep coding.

Previous Next

CodeWithHarry