Bits and Pieces of Code

My personal place to share projects and code tutorials.

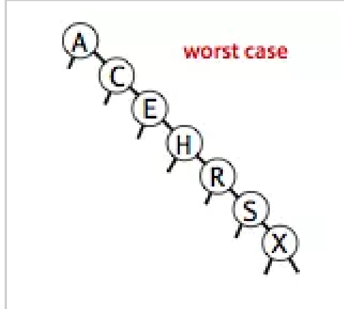
AVL Tree in C#

Karim Oumghar / September 16, 2014

AVL Tree is a self balancing binary tree data structure. It has a very efficient Insert, Delete, and Find times. In terms of the depth of an AVL tree on both sides, it differs at most by 1 level. At any other time where difference in height/depth is greater than 1 or less than -1, rebalancing occurs. In terms of space it has a O(n) complexity. With time complexity it has O(log n) for all cases (worst, average, best). Comparing this with the commonly known Red-Black Tree, the AVL Tree is more rigidly balanced than the RB Tree, thus while having fast retrieval times, the RB Tree is more efficient in insertion & deletion times. Nonetheless, both have a runtime of O(log n) and are self balancing. The name AVL comes from the creators of this algorithm (Adelson-Velskii and Landis).

Why the need to balance?

Consider a regular binary tree or a binary search tree. We know that in the worst case retrieval and insertion is O(n), when the tree looks like a linked list, and traversal is pretty much like that of a linked list. This is quite inefficient and costs time. To remedy and eliminate this problem, we introduce the idea of a self balancing tree; through height checking and rotations, maintains a more balanced structure; thus less time to lookup some data.



In the worst case, a regular BST or Binary Tree takes the shape resembling a linked list.

The algorithm of an AVL Tree is as follows:

- Get the height difference from both sides of the tree, using recursion and the difference in balance is the height of the left side minus height of the right side
- If the balance is greater 1 or less than -1, rotations must occur to balance the tree, if the balance is -1,0,or 1, then no rotations are needed.
- Nodes in the AVL Tree also store their height, for example, nodes at the top are higher than nodes at the bottom therefore the root would store the highest height while leaf nodes at the bottom would store a height of 1

In this kind of self balanc … alled rotations. The data struct

- There are 4 cases for rotatio … right
- Right-Right: All nodes are to … comes the new parent/root and original p … pivot
- Right-Left: Pivot is the right c
- Left-Left: All nodes are to the
- Left-Right: Pivot is the left ch
- To go even further with how r
  - A generic rotation in pseu
  - Pivot = Parent.L
    Parent.L = Pivot.L
    Pivot.R = Parent
    return Pivot
  - Illustrations:

Rotation Illustrations

Methods:

- Insert(): after inserting a new node using normal procedure (recursive or non recursive), its necessary to check each of the nodes ancestors for an unbalance in the tree, therefore calling the Balance() method, basically, insert and then a small fix.
  - To go into further detail, we have private and public Insert methods. The private method inserts recursively and it takes a new node object, and a node reference/pointer, and it is here where we call Balance_Tree(). In the public method, we call the private recursive insert method and we need to set our root pointer/reference equal to the method call because the private method returns type Node. Also because of the recursion, when we perform a rotation from calling the Balance_Tree() method, we need to recurse up one level and make the necessary re-connections of parent to pivot nodes. [The best way to visualise this recursion is to draw a stack frame of calls in order to see the process better.]
  - In short, our base case is that if our current node we use to traverse the tree to insert is null, current = new node and return current. That would go to our next statement which recurses up one level and sets current->left/right to the newly added node. Then we balance our tree by calling Balance_Tree. Once rotations have been done, we return our pivot node and re-check the balance factor once again to make sure we have no imbalances. Recurse up a level once more and reconnect the rotated nodes to the parent node.

- Search(): Searching is more optimized since things are more balanced, therefore normal implementation in this function is sufficient.

- Delete(): Just like Insert(), after Deletion occurs we have to call Balance() to check each of the nodes for any unbalance in the tree, we have a public Delete() and a private recursive Delete() that does the actual work
- RotateRR(), RotateLL(), RotateLR(), and RotateRL() all take in a node pointer/reference argument, and return a pivot node with the rotation
- GetHeight(): takes a node reference/pointer argument, and returns the height. More info here on why we add 1 to the height.
- Balance_Factor(): takes a Node reference as an argument, this will recursively get the heights for both sides and return an integer (left height – right height)
- Balance_Tree(): This method takes a node pointer/reference passed into it. When we balance the tree, the algorithm in goes something like this:
  - If balance factor is 2, we first check if we have a left-left case, if we do then we perform that rotation, else, we perform a left-right rotation
  - If balance factor is -2, we first check if we have a right-right case, if we do then we perform a right right rotation, else, we perform a right-left rotation

Implementation in C#

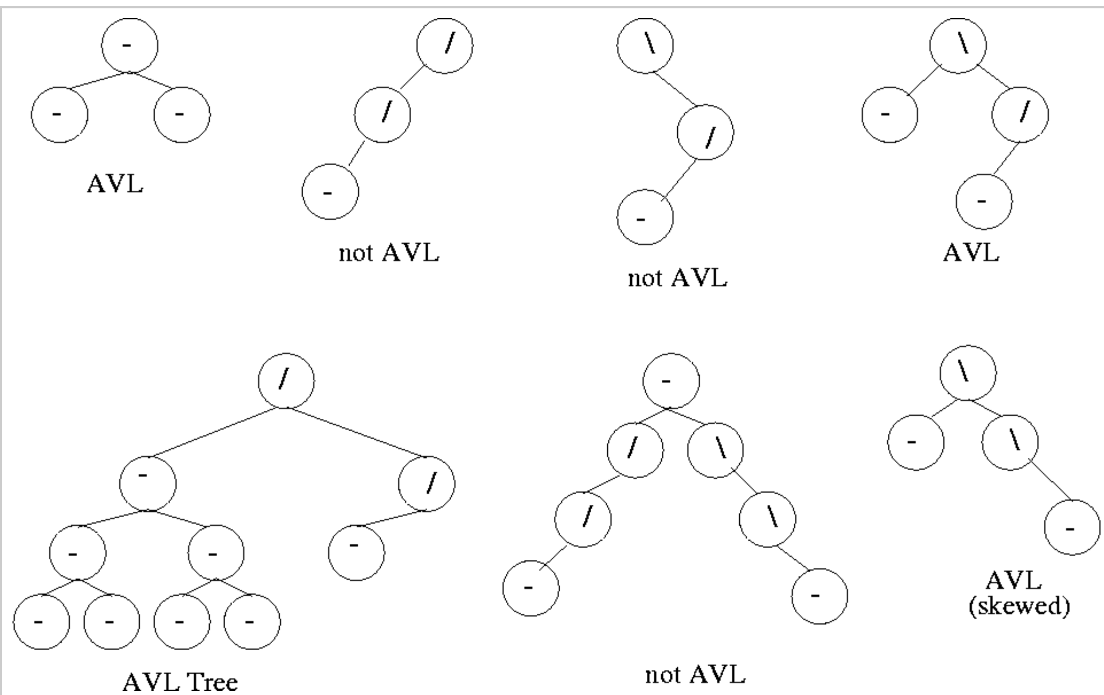
```
class Program
    {
        static void Main(string[] args)
        {
            AVL tree = new AVL();
            tree.Add(2);
            tree.Add(1);
            tree.Add(0);
            tree.Add(-1);
            tree.Add(-2);
            tree.Add(3);
            tree.Add(5);
            tree.Add(4);

            tree.DisplayTree();
        }
    }
    class AVL
    {
        class Node
        {
            public int data;
            public Node left;
            public Node right;
            public Node(int data)
            {
                this.data = data;
            }
        }
        Node root;
        public AVL()
        {
        }
        public void Add(int data)
        {
            Node newItem = new Node(data);
            if (root == null)
            {
```

```
                root = newItem;
            }
            else
            {
                root = RecursiveInsert(root, newItem);//root = 
            }
        }
        private Node RecursiveInsert(Node current, Node n)
        {
            if (current == null)//base case, we reach this when
            {
                current = n;
                return current;
            }
            else if (n.data < current.data)//if the new node is
            {
                current.left = RecursiveInsert(current.left, n)
                current = balance_tree(current);//calling balan
            }
            else if (n.data > current.data)//if the new node is
            {
                current.right = RecursiveInsert(current.right, 
                current = balance_tree(current);
            }
            return current;
        }
        private Node balance_tree(Node current)
        {
            int b_factor = balance_factor(current);
            if (b_factor > 1)
            {
                if (balance_factor(current.left) > 0)
                {
                    current = RotateLL(current);
                }
                else
                {
                    current = RotateLR(current);
                }
            }
            else if (b_factor < -1)
            {
                if (balance_factor(current.right) > 0)
                {
                    current = RotateRL(current);
                }
                else
                {
                    current = RotateRR(current);
                }
            }
            return current;
        }
        public void Delete(int target)
        {
             Delete(root, target);
        }
        public void Find(int key)
        {
            if (Find(key, root).data == key)
            {
                Console.WriteLine("{0} was found!", key);
            }
            else
            {
```

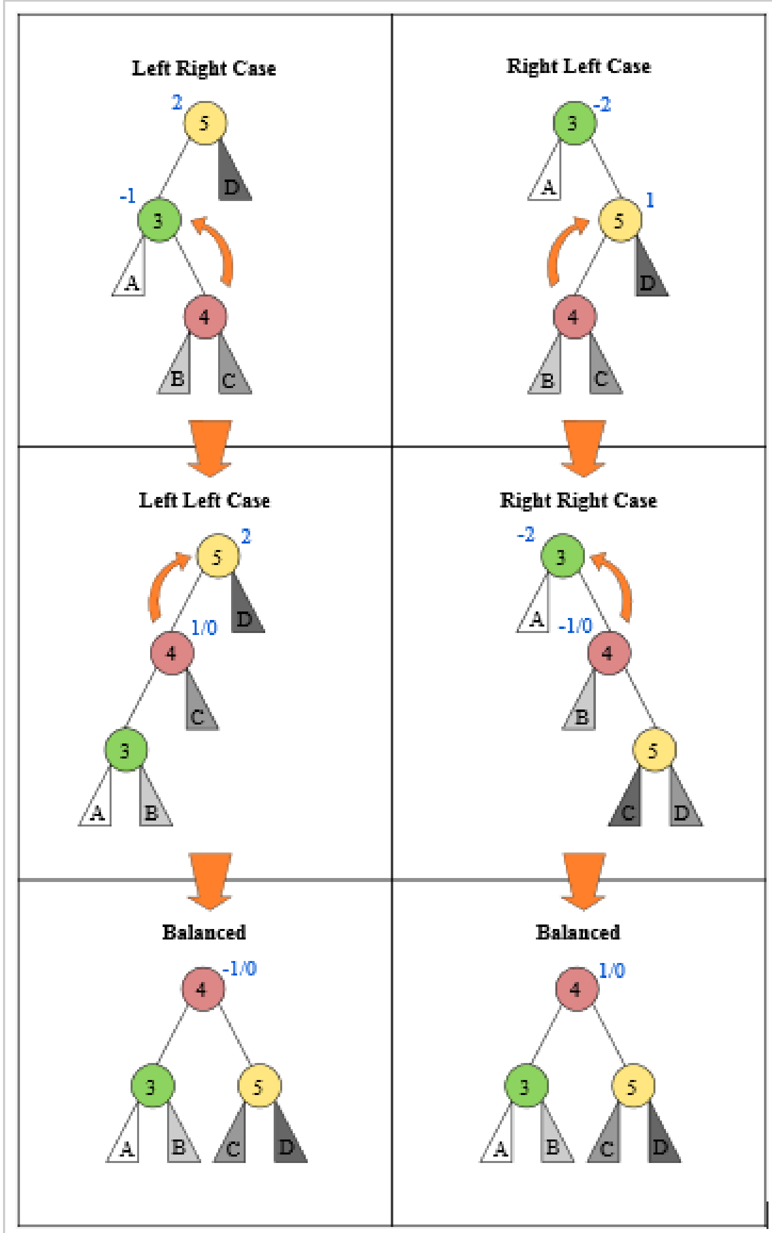
```
                Console.WriteLine("Nothing found!");
            }
        }
        private Node Find(int target, Node current)
        {

                if (target < current.data)
                {
                    if (target == current.data)
                    {
                        return current;
                    }
                    else
                    return Find(target, current.left);
                }
                else
                {
                    if (target == current.data)
                    {
                        return current;
                    }
                    else
                    return Find(target, current.right);
                }

        }
        public void DisplayTree()
        {
            InOrderDisplayTree(root);
            Console.ReadLine();
        }
        private void InOrderDisplayTree(Node current)
        {
            if (current != null)
            {
                InOrderDisplayTree(current.left);
                Console.Write("({0}) ", current.data);
                InOrderDisplayTree(current.right);
            }
        }
        private int max(int l, int r)//returns maximum of two i
        {
            return l > r ? l : r;
        }
        private int getHeight(Node current)
        {
            int height = 0;
            if (current != null)
            {
                int l = getHeight(current.left);
                int r = getHeight(current.right);
                int m = max(l, r);
                height = m + 1;
            }
            return height;
        }
        private int balance_factor(Node current)
        {
            int l = getHeight(current.left);
            int r = getHeight(current.right);
            int b_factor = l - r;
            return b_factor;
        }
        private Node RotateRR(Node parent)
        {
```

```
            Node pivot = parent.right;
            parent.right = pivot.left;
            pivot.left = parent;
            return pivot;
        }
        private Node RotateLL(Node parent)
        {
            Node pivot = parent.left;
            parent.left = pivot.right;
            pivot.right = parent;
            return pivot;
        }
        private Node RotateLR(Node parent)
        {
            Node pivot = parent.left;
            parent.left = RotateRR(pivot);
            return RotateLL(parent);
        }
        private Node RotateRL(Node parent)
        {
            Node pivot = parent.right;
            parent.right = RotateLL(pivot);
            return RotateRR(parent);
        }
        private Node Delete(Node current, int target)
        {
            Node parent;
            if (current == null)
            { return null; }
            else
            {
                //left subtree
                if (target < current.data)
                {
                    current.left = Delete(current.left, target)
                    if (balance_factor(current) == -2)
                    {
                        if (balance_factor(current.left) <= 0)
                        {
                            current = RotateRR(current);
                        }
                        else
                        {
                            current = RotateRL(current);
                        }
                    }
                }
                //right subtree
                else if (target > current.data)
                {
                    current.right = Delete(current.right, targe
                    if (balance_factor(current) == 2)
                    {
                        if (balance_factor(current.right) <= 0)
                        {
                            current = RotateLL(current);
                        }
                        else
                        {
                            current = RotateLR(current);
                        }
                    }
                }
                //if target is found
                else
```

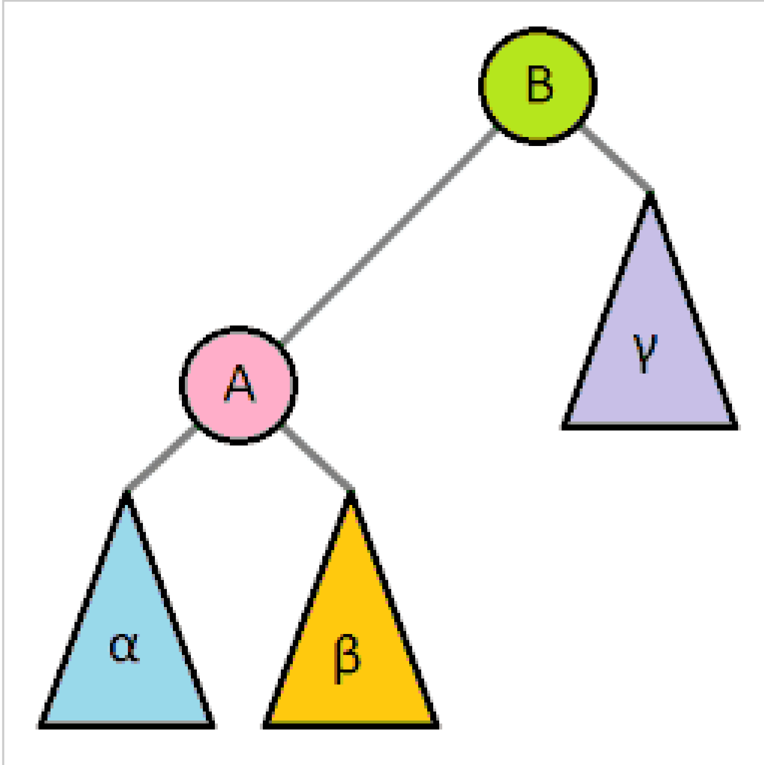
```
                {
                    if (current.right != null)
                    {
                        //delete its inorder successor
                        parent = current.right;
                        while (parent.left != null)
                        {
                            parent = parent.left;
                        }
                        current.data = parent.data;
                        current.right = Delete(current.right, pa
                        if (balance_factor(current) == 2)//rebal
                        {
                            if (balance_factor(current.left) <=
                            {
                                current = RotateLL(current);
                            }
                            else { current = RotateLR(current);
                        }
                    }
                    else
                    {
                        return current.left;
                    }
                }
            }
            return current;
        }
    }
```



AVL Tree depictions



AVL Tree (unbalanced and balanced tree process)



Tree rotation animation

[Interactive AVL Tree Applet demo.]

Twitter 3 Facebook Google Email Print LinkedIn
Reddit Tumblr Pinterest Pocket

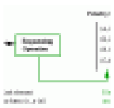
Like

Be the first to like this.

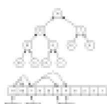
September 16, 2014 in C#, Data Structures, Java, Searching & Sorting. Tags: algorithm, avl tree, big-o, binary search tree, bst, c programming, c#, c/c++, computer science, data structures, how to implement avl tree, java, node, recursive, searching, self balancing, sorting, tree, tree rotation, tutorial

Related posts

Generic Programming in C#.Net



Priority Queue Tutorial (C#, C++, Java)



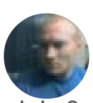
The Heap Data Structure (C++, Java, C#)

[← jQuery simple example](#)

[Installing Infinity Conky Theme for Ubuntu/Linux/Fedora →](#)

4 thoughts on “AVL Tree in C#”

Pingback: [Red-Black Tree in C# | Bits and Pieces of Code](#)



GoodNPlenty333 (@GoodNPlenty333)

July 3, 2015 at 3:44 pm

You are recalculating the height of each node recursively, which is very costly. You can gain significant performance improvements by storing the height in each node.

★ Like

[Reply](#)

Karim Oumghar July 7, 2015 at 5:39 pm

I am aware of this. At the moment I am busy with other things however I have kept a note of this and will update this tutorial soon. Thanks for your feedback.

★ Like

Reply

aljensen September 22, 2015 at 5:12 pm

Reblogged this on .Net Programming with Al Jensen.

★ Like

Reply

Leave a Reply

Enter your comment here...

Search …

Recent Posts

A Snake Game in JavaScript

Generic Programming in C#.Net

Android: How to make a phone call

Priority Queue Tutorial (C#, C++, Java)

JavaScript Animations Tutorial

Categories

Articles

C#

C++

Data Structures

Database

GDI+

HTML/CSS

Java

Javascript

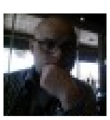
OOP

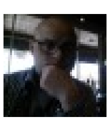
Searching & Sorting

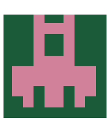
Uncategorized

 **Karim Oumghar**

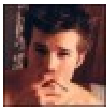
Recent Comments

 aljensen on AVL Tree in C#

 aljensen on The Heap Data Structure (C++,…

 Nikita on Priority Queue Tutorial (C#, C…

 Karim Oumghar on Priority Queue Tutorial (C#, C…



dimezis on Priority Queue Tutorial (C#, C…

GitHub

My GitHub Profile

LinkedIn

LinkedIn

Follow Blog via Email

Enter your email address to follow this blog and receive notifications of new posts by email.

Enter your email address

Follow

Follow Bits and Pieces of Code

Live Traffic Stats