AVL & Red-Black Tree Comparison

Aiden Williams

372001L

Department of Artificial Intelligence

Faculty of ICT

University of Malta

June 2021

Contents

Table of Figures

Introduction

This assignment is focused on the implementation and comparison of two balanced binary search trees: The AVL Tree and Red-Black Tree. This report will go over the implementation of these two trees by providing pseudo code for the main elements of any tree: being insertion, deletion, and search. As will be explained in the report, these trees also make use of rotation algorithms that facilitate their self-balancing behaviour, these too shall be explained. Both trees were implemented using raw Python (3.8) the code for which is attached with this submission. Also attached is a IPython notebook Comparison.pynb (and its html variant, included for readability), where both trees where tested and evaluated based off the project requirements. Throughout this report Python styled pseudo-code is used.

AVL Tree

Introduction

The AVL tree is named after its inventors Georgy Adelson-Velsky and Evgenii Landis who in 1962 published a paper titled "An algorithm for the organization of information" defining the tree [1]. The tree is the first self-balancing tree proposed, where given a root has two children, the children's height can only differ by 1. If after an insertion or deletion the tree becomes unbalanced several rotations can be done to re-balance the tree.

Properties

Balance

Balance is maintained by checking the Balance Factor of a particular tree. The balance factor is defined as:

$$BF(Node) := Height(RightChild(Node)) - Height(LeftChild(Node))$$

Where height is defined as:

The largest number of edges of the root node to its last child.

However, for the tree to be considered an AVL Tree the BF must be defined as such:

$$BF(Node) \in \{-1, 0, 1\}$$

I.e., the BF can either be:

- -1, Meaning the tree is left leaning
- 1, Meaning the tree is right leaning
- 0, Meaning the tree is perfectly balanced

Big-O Notation

The big O notation cases were taken from [2]

AVL Tree	Average Case	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Applications

The AVL tree is best used when insertions and deletions are not as frequent, but searches are. While researching the tree, it was found that railway systems can use AVL trees in their system, as train purchases, which would be result in an insertion, and replacement, which would result in a deletion, happen at the most frequent, yearly while searches are done very frequently all the time.

Implementation

Rotations

Rotations are used for balancing the AVL tree. In general balancing actions need to be taken when the BF of a tree is no longer within the set as defined above. In this implementation rotations only occur after an insertion or a deletion of a node. For both insertion and deletion, a number of cases dictate in which order the rotations and number of which need to happen.

Left Rotation

```
leftRotate(AVLN):
AVLNR := AVLN.right
AVLNL := AVLNR.left
AVLNR.left := AVLN
AVLN.right := AVLNL
AVLNR is returned
```

Right Rotation

```
rightRotate(AVLN):
AVLNL := AVLN.left
AVLNR := AVLNL.right
AVLNL.right := AVLN
AVLN.left := AVLNR
AVLNL is returned
```

Insertion

AVL Tree insertion has similar functionality to standard BST insertion and in fact uses the same algorithm, but has the added case checking for self-balancing, which are considered after insertion. For this reason, the below pseudo-code and documentation has been split into two sections, one detailing the standard BST insertion and the other details the cases for rotation.

Standard BST Insertion

```
Insert (AVLN, Key):
If AVLN is empty:
      AVLN := AVLNode(Key).
      AVLN is returned.
If AVLN.left.key > Key:
      AVLN.left := Insert(AVLN.left, Key).
If AVLN.right.key > Key:
      AVLN.right := Insert(AVLN.right, Key).
```

Balancing Cases

BF is the Balancing Factor of the current sub-tree

Case 1 – Right Rotation

```
BF > 1 and Key < AVLN.left.key.
```

Case 2 – Left Rotation

```
BF < -1 and Key > AVLN.right.key.
```

Case 3 – Left Right Rotations

```
BF > 1 and Key > AVLN.left.key.
```

Case 4 – Right Left Rotations

```
BF < -1 and Key < AVLN.left.key.
```

Deletion

Much like insertion, AVL Tree deletion implements standard BST deletion but has the added functionality for the rotation cases which are considered after deletion. Here again, the pseudo-code and documentation is split into these two sections of the deletion algorithm.

Standard BST Deletion

```
Delete (AVLN, Key):
If AVLN is empty:
      None is returned.
If Key is < the AVLN.Key:
      AVLN.left := Delete (Key, AVLN.left).
If Key is > the AVLN.Key:
      AVLN.right := Delete (Key, AVLN.right).
Otherwise (Key is AVLN.Key):
      If AVLN.left is empty:
            AVLN.right is returned.
      If AVLN.right is empty:
            AVLN.left is returned.
      temp := The node with minimum value in AVLN.left.
      AVLN.key := temp.key.
      AVLN.left := Delete (AVLN.right, temp.key).
```

Balancing
BF is the Balancing Factor of the current sub-tree

Case 1 – Right Rotation

```
BF > 1 and AVLN.left.key BF is >= 0.
```

Case 2 – Left Rotation

```
BF < -1 and AVLN.right.key BF is <= 0.
```

Case 3 – Left Right Rotations

```
BF > 1 and AVLN.left.key BF is < 0.
```

Case 4 – Right Left Rotations

```
BF < -1 and AVLN.left.key BF is > 0.
```

Search
The standard BST search algorithm is used for AVL Trees.

```
Search (AVLN, Key):
If AVLN is empty:
      None is returned.
If AVLN.key is > Key:
      Search (AVLN.right, key) is returned.
If AVLN.key is < Key:
      Search (AVLN.left, key) is returned.
If AVLN.key is Key
      AVLN is returned.
```

Red-Black Tree

Introduction

The RB Tree is named after the colours used to ensure balance in the tree. It is the product of multiple works and contributions from various people most recently, Robert Sedgewick's "Left Leaning Red-Black Tree" [3]. The RB Tree is a self-balancing tree, that follows a set of rules. If after an insertion or deletion the tree becomes unbalanced several rotations and colourings can be done to re-balance the tree. In my implementation I followed closely Sedgewick's paper.

Properties

These are the set of rules that constrain RB Trees.

1. Every Node can either be Red or Black.
2. All Null(empty) nodes are considered Black.
3. If a Node is Red, then both of its children must be Black.
4. Every path from a given Node to any of its descendant Null nodes would contain the same number of Black nodes.

In some versions a fifth rule is applied, that the root node is black, however Sedgewick does not apply this rule.

Big-O Notation

The big O notation cases were taken from [4]

AVL Tree	Average Case	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Applications

RB trees are used in several applications ranging from Java TreeMap [5], a school phone lookup system [6] and the Linux Completely Fair Scheduler [7]. RB trees are used when balancing, and hence retrieval, is important. The tree is also useful for frequent insertion and deletion due to the retrieval property.

Implementation

In this section I go over my implementation for the 4 main parts of the tree.

Rotations and Colour Flipping

Rotation and colour flipping operations are used to balance the RB tree. In this implementation, rotations are used to transform a 3-Node-Subtree with a red node that leans to the left to a 3-Node-Subtree with a red node that leans to the right and vide-versa. Due to the added colour property of the RB tree, a colour flip operation is also used to flip a 3-Node-Subtree's colours.

Left Rotation

```
rotateLeft(Node):
x := Node.right.
Node.right := x.left.
x.left := Node.
x.colour := Node.colour.
Node.colour := Red.
x is returned.
```

Right Rotation

```
rotateRight(Node):
x := Node.left.
Node.left := x.right..
x.right := Node.
x.colour := Node.colour.
Node.colour := Red.
x is returned.
```

Colour Flip

```
flipColours(Node):
if Node.colour is Red:
      Node.colour := Black.
Otherwise:
      Node.colour := Red.
If Node.left Exists:
      If Node.left.colour is Red:
            Node.left.colour := Black.
      Otherwise:
            Node.left.colour := Black.
If Node.right Exists:
      If Node.right.colour is Red:
            Node.right.colour := Black.
      Otherwise:
            Node.right.colour := Red.
```

Insertion

The RB Tree insertion algorithm can be split into three sections: colour flip case, standard BST insertion and the rotation cases. Much like the AVL Tree, RB Tree insertion revolves around standard BST insertion, but for a proper implementation, cases are added that to maintain balance according to the tree's properties, as outlined in the properties section.

Colour Flipping

```
If h.left is Red and h.right is Red:
      h := flipColours(h).
```

Standard BST Insertion

```
If Key is h.key:
      Continue.
If Key < h.key:
      h.left := insert(h.left, Key).
Otherwise:
      h.right := insert(h.right, Key).
```

Rotation Cases

There are two rotation cases in the insert algorithm:

Case 1 – Left Rotation

```
h.left is Black and h.right is Red.
```

Case 2 – Right Rotation

```
h.left is Red and h.left.left is Red.
```

Deletion

Deletion works in the direct opposite way insertion does. However, the implementation is more complex than insertion. In this implementation several delete functions are used as well as added rotation and colour changing functionalities.

Public Delete

The public delete function acts as the entry point to the more complex _delete function. Here the input key is first checked to see whether it appears in the tree or not.

```
delete(tree, Key):
res = search(Key)
If res is None:
      False is returned.
If tree.root.left is Black and tree.root.right is Black:
      tree.root.colour := Red.
If tree.root Exists:
      tree.root := _delete(tree, tree.root, Key).
If tree is not Empty:
      tree.root.colour := Black.
```

Private Recursive Delete

In _delete, the different cases for standard rotations and red node rotations are checked for. It was decided to document this function in its entire form and then provide extra documentation for any functionality used, as there are several cases within at various stages and depending on previous stages which would have made the different style of documentation convoluted.

```
_delete(tree, h, Key):
If Key < h.key:
      If h.left is Black and h.left Exists and h.left.left is Black:
            h := moveRedLeft(h, tree).
      h.left := _delete(h.left, Key).
Otherwise:
      If h.left is Red:
            h := rotateRight(h).
      If Key is h.key and h.right is None:
            None is returned.
      If h.right is Black and h.right Exists and h.right.left is Black:
            h := moveRedRight(tree, h).
      If Key is h.key:
            h.key := min(tree, h.right).
            h.right := deleteMin(tree, h.right).
      Otherwise:
            h.right := _delete(tree, h.right, key).
fixUp(tree, h) is returned.
```

moveRedLeft

The moveRedLeft function is an RB rotation case where there are two sequential left Black nodes.

```
moveRedLeft(node, tree):
tree := flipColours(tree).
If node.right and node.right.left is Red:
      node.right := rotateRight(node.right).
      node := rotateLeft(node).
      tree := flipColours(tree).
node is returned.
```

moveRedRight
The moveRedLeft function is a RB rotation case where there are two sequential right Black nodes.

```
moveRedRight(node, tree):
tree := flipColours(tree).
If node.left Exists and node.left.left is Red:
      node := rotateRight(node).
      tree := flipColours(tree).
node is returned.
```

min (Node)
The min function returns the key of the smallest node in a tree, by searching the left node recursively.

```
min(node, tree):
If node.left is None:
      node.key is returned.
Otherwise:
      min(node.left, tree) is returned.
```

min (tree)
The min function uses the Node min function to returns the smallest key in the tree.

```
min(node):
If tree.root is None:
      None is returned.
Otherwise:
      min(tree.root , tree) is returned.
```

deleteMin (node)
The Node deleteMin function deletes the smallest node in the current tree, the moveRedLeft case is also checked while traversing the tree.

```
deleteMin(node, tree):
If self.left is None:
      None is returned.
If node.left is Black and node.left Exists and node.left.left is Black:
      node := moveRedLeft(node, tree).
node.left := deleteMin(node.left , tree).
```

deleteMin (tree)
The tree deleteMin function uses the Node deleteMin function to delete the smallest node in the tree.

```
deleteMin(tree):
        tree.root := deleteMin(tree.root , tree)
        tree.root.colour := BLACK.
```

FixUp
The FixUp function represents the rotation cases checked on the way up of the recursive _delete calls.

```
fixUp(node, tree):
If node.right is Red:
      node := rotateLeft(node).
If node.left is Red and node.left Exists and node.left.left is Red:
      node := rotateRight(self).
If noed.left is Red and noed.right is Red:
      tree := flipColours(tree).
node is returned.
```

Search

The standard BST search algorithm is used for RB Trees however here (compared to the AVL implementation) it is implemented iteratively instead of recursively.

```
Search (Key):

x := tree.root.

Loop until x Exists:

      If key is x.key:

            x.key is returned.

      If key < x.key:

            x := x.left.

      If key > x.key:

            x := x.right.
```

Comparison Analysis

General Comparison

As detailed in the report above, both the AVL Tree and RB Tree are self-balancing binary trees. As they are based on BST, both trees have several similarities, including the implementation of the BST insertion, deletion and search algorithms within the respective trees' algorithms. When the trees diverge from the BST implementation both use the same tool for balancing, being rotations. While the implementation and use of the Left or Right rotations is slightly different, the resulting tree after the operation of a few rotations is the same, being a balanced tree. Here at balancing the trees have their differences become more apparent.

In AVL trees a balance factor, as shown in the AVL Tree Properties section, is used to tell if a child node is out of balance whereas RB Trees use the colours of parent or child nodes as indicators. It is due to this that the maximum height difference for an AVL Tree is a factor of 1 whereas for an RB Tree this is 2.

Between the two, AVL Tree balance is stricter due to less rules, as described in the AVL Tree Properties section. In fact, every AVL Tree can be considered an RB Tree after one simple operation, where each node is given an alternating colour, either black or red, starting from assigning the root of the tree with a black colour.

Test Result Comparison

Table 1: Insertion Results

AVL		RBT	
Rotations	2,611	Rotations	3,866
Height	12	Height	18
Comparisons	15,233,522	Comparisons	302,826
Nodes	2,621	Nodes	2,621

2621 items where inserted, of which all of them where successfully inserted into the trees.

Table 2: Deletion Results

AVL		RBT	
Rotations	1	Rotations	1,956
Height	12	Height	18
Comparisons	6,214,874	Comparisons	81,769
Nodes	2,381	Nodes	2,381

564 items where put for deletion, off which only 240 were present in the trees.

Table 3: Search Results

AVL		RBT	
Rotations	0	Rotations	0
Height	12	Height	18
Comparisons	66,598	Comparisons	72,747
Nodes	2,381	Nodes	2,381

1749 items were searched for.

Comparing the results of the tests done on the trees in Comparison.ipynb, the RB Tree performs a larger number of rotations for insertion and deletion but has significantly fewer comparison checks. The stricter balancing rules for the AVL Tree are also apparent as its height is smaller than the RB Tree after both insertion and deletion. Something that must be noted is the very small number of rotations performed in deletion by the AVL Tree. In search, both trees performed similarly, which is expected since both implement the same standard BST search algorithm.

Conclusion

After implementing and testing the AVL Tree and the RB Tree I can conclude that both trees can be used interchangeably for most applications requiring a balanced binary tree. The case for one or the other must be made on whether the system the tree will be implemented on is weak on comparisons or memory movement. If the former than an RB Tree implementation is ideal as the number of comparisons is much lower, especially on deletion, whereas if it is the latter, an AVL implementation would be ideal since the number of rotations performed, and hence memory movement, is much lower, again especially in the case of deletion.

Bibliography

[1] G. . M. Adel'son-Vel'skii and E. M. Landis, “An algorithm for the organization of information,” *Soviet Mathematics Doklady,* vol. 3, pp. 1259-1263, 1962.

[2] E. Alexander, “AVL Trees,” 31 07 2019. [Online]. Available: https://web.archive.org/web/20190731124716/https://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html. [Accessed 02 06 2021].

[3] R. Sedgewick, “Left-leaning Red-Black Trees,” 2008.

[4] J. Paton, “Red-Black Trees,” [Online]. Available: http://pages.cs.wisc.edu/~paton/readings/Red-Black-Trees/. [Accessed 02 06 2021].

[5] Oracle, “Class TreeMap<K,V>,” Oracle, 2020. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html. [Accessed 02 06 2021].

[6] jawardell, A. Hufstetler, omarbatyah and KDTrey, “github,” 18 04 2018. [Online]. Available: https://github.com/jawardell/red_black_tree. [Accessed 02 06 2021].

[7] M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler,” IBM, 19 09 2018. [Online]. Available: https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler. [Accessed 02 06 2021].

Statement of Completion

AVL Tree	Yes
Red-Black Tree	Yes
Discussion Comparing AVL to RBT	Yes