**A Comparative Study of Balanced BST**

**Data Structures.**

**By George Katsonis.**

**Introduction:**

The purpose of this paper is to examine and compare the functionality, speed, and applications of several structures, based on binary search trees (BSTs). A binary search tree is a tree structure characterized by the rule that a parent node can only have two children nodes, of which the left must be smaller, and the right must be larger than the parent. The purpose of the tree data structure is to allow for faster data insertions than ordered arrays, and faster data searches than linked lists. However, a normal BST often becomes unbalanced, leading to O(N) time required for base operations. These alternative expressions of a BST aim to improve this operation time by balancing the data (often leading to O(logN) lookup times). This paper assumes knowledge of data structure terminology and will attempt to retain concision by avoiding redundant over-explanation of such. It should be also noted that this paper attempts to compile knowledge from many different sources and present it in an easy to understand fashion. This means that in text citations will be scarce, due to the factual nature of the information and the fact that throughout different sources, the same information is repeated, although in different formats.

**AVL Trees:**

**Description:** The AVL tree is a BST that does not allow for a difference in height larger than 1 between two siblings. This rule assumes that an empty tree (or sub-tree) has a height of -1, while a tree with only one node has a height of 0. In mathematic terms we need a **balance factor** (B(h)) that is smaller or equal to 1, where B(h) is the height of the left sub-tree of any given node, minus the height of the same node’s right sub-tree. This can be represented as **B(h) = H(TL) – H(TR) <= 1**. In order to achieve this form, the tree uses **rotations**.

**Principle:** Every time we perform an insertion in an AVL tree, we have to calculate the balance factor for all nodes. In case we find a balance factor other that 1,0, or -1, we have to perform a rotation.

It is important to note that a node can be either **left-heavy**, if the balance factor is larger than zero, or **right-heavy**, if the balance factor is smaller than zero.

When we come across a node with a problematic balance factor, we examine whether it is left-heavy or right-heavy. We call this problematic node **Critical Node.**

Depending of the nature of our Critical Node we have the following possibilities:

* If the Critical Node is left-heavy can have:
  + A LL scenario: This means that the left child of the critical node is also left-heavy. In this scenario we perform a single Right rotation.   
    To perform a Right rotation, we set the left child of the critical node as the root of the sub-tree (Critical node-child-grandchild), the critical node becomes the right child, and the grandchild becomes the left child. All subsequent children are positioned according to the rules of a BST.   
    As we can infer from the following example (**1.1)**, this means that the critical node (X), becomes the right child of its previous left child (Y). Furthermore, the right child (Purple) of the previous left child (Y), now becomes the left child of the critical node (X). The rest of the tree is left unchanged. (In this example no rotation was needed but it was used to clearly display the rotation.)
  + A LR scenario: This means that the left child of the critical node is right-heavy. In this scenario we perform two subsequent rotations, first a right rotation, followed by a left rotation. A left rotation operates exactly the same as a right rotation, but instead of considering the left children, we instead consider the right.
* If the Critical Node is right-heavy we have the exact same scenarios mirrored (RR – RL).

**Performance:** A normal BST guarantees to run at O(h) speed, where h is the height of the tree. The AVL balancing process results to an almost balanced tree, where all subtrees cannot differ in height for more than 1, guaranteeing thus, an O(log N) for **lookup**, **insertion**, and **deletion**, as long as we calculate the height of each node during the insertion and not in a separate loop. The major disadvantage of this tree is the required time to perform the necessary rotations and balance the tree. It is important to note that all AVL operations are the executed as in a normal BST, the only difference being the balancing of the tree. Another disadvantage is the need to store at least 4 bits of information per node to maintain balance information.

**2-3 Trees:**

**Description:** A 2-3 tree is actually a form of a B-tree, a subcategory of BSTs. B-trees, unlike BSTs, allow for more than two children, and must be perfectly balanced (As all things should be), meaning that all leaf nodes must be on the same height. The number of maximum children (M) allowed in a B-tree, dictate that tree’s order. Since the 2-3 trees are based on a 3-order B-tree, we can infer that each node can have either 2 children (**2-Node**) or 3 children (**3-Node**). A 2-Node can have one value while a 3-Node can have up to 2. **4-Nodes** also appear during tree computations but they never persist in the tree.

**Principle:** To find a node in a 2-3 tree we need to take into consideration the existence of 3-nodes. Since 2-3 trees follow the same ordering rules as a normal BST, moving left or right remains the same. However, in case we have to travel through a 3-node, we need to consider both numbers and thus have three possible travel paths. Left for smaller, right for bigger, and middle for in-between.

To insert, we follow the rules of the BST until we reach a leaf node. We examine the node and identify its type. If it is a 2-Node, we add our value to it and transform it into a 3-Node, ending the process. If it is a 3-Node, we will merge our value, temporarily transforming it to a 4-Node, bust since this is a 2-3 tree, we will need to perform a split.

When performing a split, the middle value of the 4-Node moves up one level, while the two outer values will split, becoming separate nodes but maintaining their relevant positions (**1.2**).

This procedure ripples if required, meaning that if the parent node was already a 3-Node, it will split again. If the ripple reaches the root, a final split will happen and thus the height of the tree will have increased by one. The only way for the height to increase, thus, is through the root.

Deletion also operates according to the rules of a BST, with the same re-balancing method used for insertion, going through the nodes and moving values or performing splits and merges as necessary.

**Performance:** Since 2-3 trees follow the same operation procedures as a BST, the fact that they are always perfectly balanced, leads to time O(log N) needed for **lookup**, **insertion** and **deletion**.

The major drawback of 2-3 trees is the difficulty of implementation, since we need to account for different types of nodes. Thus, the most common practice is to convert a 2-3 tree to a Red-Black tree. Deletions are also very expensive due to the need to maintain perfect balance, making 2-3 trees ideal for search operations but no so much for deletions.

**RED-BLACK Trees:**

**Description:** Red-Black tree nodes contain an extra bit of information that describes their color. As the name implies, there can be either red nodes or black nodes, and to maintain approximate balance in the tree, they follow a small set of rules**. First rule** is that the root of the tree will always be black. **Second rule** is that red nodes can only have black children. The **third rule** is a bit more complicated. It states that for every node with at least one null child, there must be the exact same number of black nodes, in the path that starts from the root and leads to that null child. These rules are established to maintain the balanced structure of the tree, while at the same time retaining all qualities of a BST.

**Principle:** Insertion in a Red-Black tree will proceed the same as in a normal BST, where every newly inserted leaf node will be initially painted red. This means that by default, that the insertion of a new node cannot affect the first rule, since we are not dealing with the root, or the third rule, since by only inserting red nodes, we do not alter the number of black nodes in that specific path. Issues, however might arise when it comes to the second rule. To tackle second rule violations, we have two main tools at our disposal, **restructuring**, and **recoloring**.

**Restructuring** follows the exact same rules and methodology as the rotations explained above, with the added rule that the new root of the rotated sub-tree must now be painted black, while the children are now painted red. This is clearly illustrated in example **1.3**. Naturally, the rotating condition is no longer height balance, but instead color balance. But the weight of a subtree and the need of a certain case of restructuring (R, L, RL, LR), exactly mimic those of the rotations.

We use restructuring in the case where the second rule is violated and the sibling of the parent of the leaf node we added, is either black or null. We can only have one rebalance per insertion.

**Recoloring** is much simpler. This method entails a shift in color for the parent of the newly added leaf node, the parent’s sibling, and also the grandparent. This is a bit hard to visualize but a quick look at example **1.4** should make things clear. We do not recolor the root of the tree in any case, in order to preserve the integrity of the first rule.

We use recoloring in the case where the second rule is violated and the sibling of the parent of the leaf node we added, is red. We can have multiple recolors per insertion.

Naturally, both these methods can create different violations of the rules on higher levels, but the rebalancing ripples as needed, much like in the AVL trees.

A null tree is of course handled as an exceptional case, where we create the root node and simply color it black.

**Performance:** **Insertion** happens the same as in a BST but since Red-Black trees are balanced, we once more have a time of O(log N) for this operation, Restructuring is O(1) since it can only happen once per insertion, but recolors can happen for the entire length of a path in a worse-case scenario, thus having a time of O(log N). **Deletion** operates very similarly to insertion, and although more complex in implementation, it also has a O(log N) time complexity. Most other tree operations like **lookup** (O(log N)) and even more specialized BST operations, do not affect the structure of the tree and thus do not induce the need for rebalancing, making them exactly as time complex as in a balanced BST.

**SPLAY Trees**

**Description:** A SPLAY tree is perhaps the more interesting of the trees we have discussed. While it does not strive to balance it self like the rest, it still provides the overall best performance when it comes to commercial use. This is due to the fact that a SPLAY tree acknowledges that 80% of all tree operations reference only 20% of the data in a real-world scenario. This fact stands due to the excessive use of reoccurring data in real life applications. Thus, the SPLAY tree operates in a fashion that makes visited data more accessible for subsequent visits. Most uses of these trees involve caches as a result.

**Principle:** SPLAY trees operate around the idea that after any of the three main operations (lookup, insert or delete), the node we accessed (or the parent of a deleted node) is promoted to the root through an operation called SPLAY. This operation performs a number of rotations very similar than the ones we have already examined, but of course, with different names.

We perform a **Zig** when the accessed node is the child of the root. A zig can be either a left or right rotation, depending on the positioning of the nodes. (**1.5**)

If the accessed node is either a left child of a left child or a right child of a right child, we perform a **Zig-Zig**, the equivalent of a RR or LL rotation.

Finally, if the accessed node is the right child of a left child (or vice-versa), we perform a **Zig-Zag** (yes those really are the names), that predictably is the equivalent of a LR (or RL) rotation.

Since we do not explicitly care about balancing the tree, there is neither a color bit nor a balance factor to take into consideration when performing these rotations. The only goal is taking the accessed node to the root. This process however, generally halves the depth of the tree anyway.

**Performance:** This is where things get a bit more complicated. Due to the nature of a SPLAY tree’s operations and structure, we cannot use traditional worst-case analysis, but instead we use amortized analysis.

Since depending on the data used and the number of SPLAYS performed, can change the time complexity from O(log N) all the way to O(N), we instead use amortized notation and say that a single operation in a SPLAY tree costs Θ(N).

Theta notation considers a large number of operations and creates a time complexity paradigm that resides between two boundaries. Since high level mathematics are outside the scope of this paper it will be easier to illustrate it through graph **1.6**.

Another way resources display this time is with O(k Log N) where N is of course the number of nodes in the tree but k represents the number of operations in a sequence.

**Comparison:**

Most of the structures we covered in this paper are largely used by modern commercial applications. Their comparison, thus, must not be on a single basis of advantages and disadvantages, but rather on the reasons that call for their separate commercial use.

Due to the fact that their height is limited to 1.44 log2 N, while the height of a Red-Black tree goes up to 2 log2 N, AVL trees are better when it comes to quick searching.

However, due to the required time to perform the extended height balancing, they are much slower than Red-Black trees when it comes to insertions or deletions. Remember, Red-Black trees only allow one reconstruction (the equivalent of a rotation) per insertion, whereas AVLs need to perform multiple rotations to maintain balance.

2-3 trees are a special case, since they have the same advantages and disadvantages as a Red-Black tree, but they are much harder to implement. Due to this fact, 2-3 trees are rarely used commercially since the development of Red-Black trees.

B-trees in general, however, are used for handling data located in external storage according to Lafore (2003). Alas, one must consider the differences in speed when it comes to external drive reading and writing operations from the time Lafore’s book was published and until today.

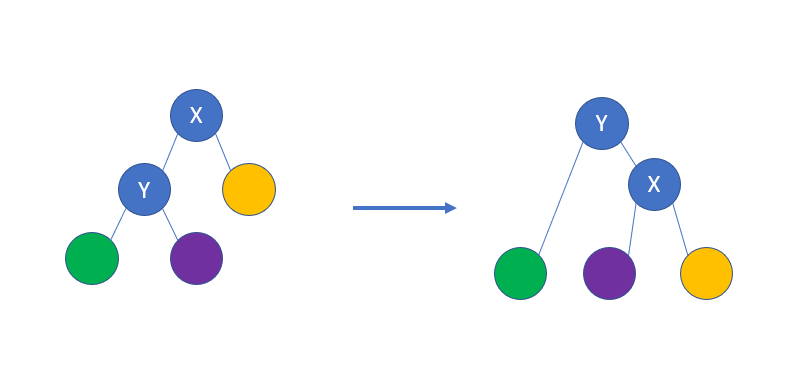
Finally, SPLAY trees, while being the least reliable of all the previous data structures when it comes to guaranteeing O(log N) operations, are also the most widely used. Since their consideration of the need to enable easy repeating access to reoccurring data, is a massively useful property for many network and enterprise applications.

Overall, it is important to note the extreme variety of existing data structures that are made to accommodate the extreme variety of specific needs. In this paper, we covered some of the most well known and widely used algorithms, but we have barely scratched the surface of modern day’s engineering ingenuity.

One, however, should commend these four data structures, for maintaining their positions as some of the most important tools in data science for so long, and for being the building blocks of all modern infrastructure.

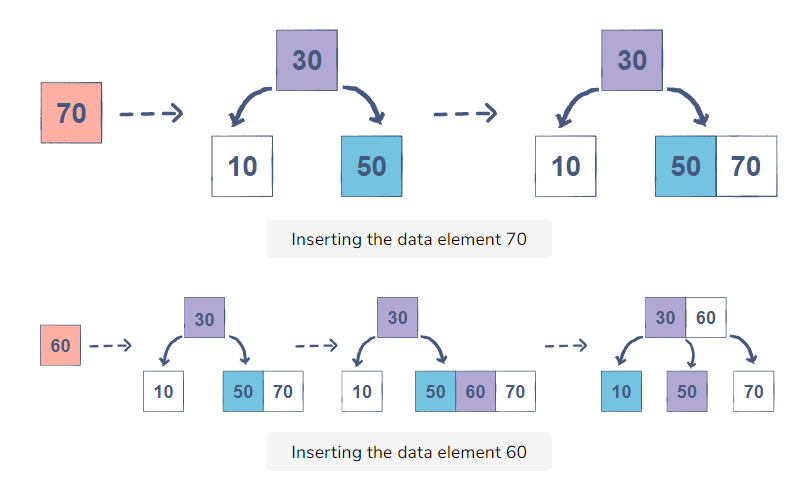
**Appendix:**

**1.1**



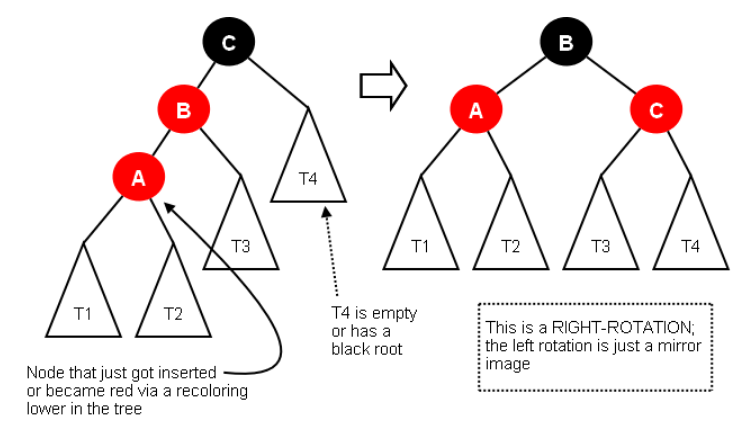
(Back To Back SWE - Platform, 2020)

**1.2**



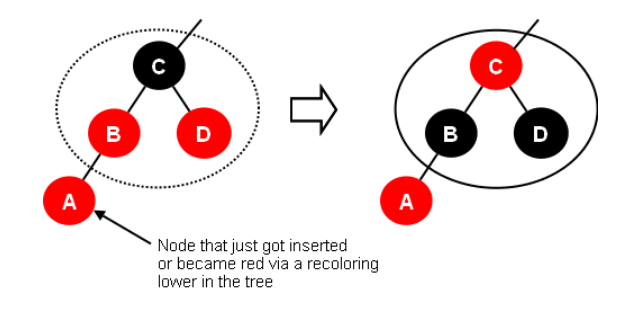
(What is a 2-3 Tree?, 2020)

**1.3**



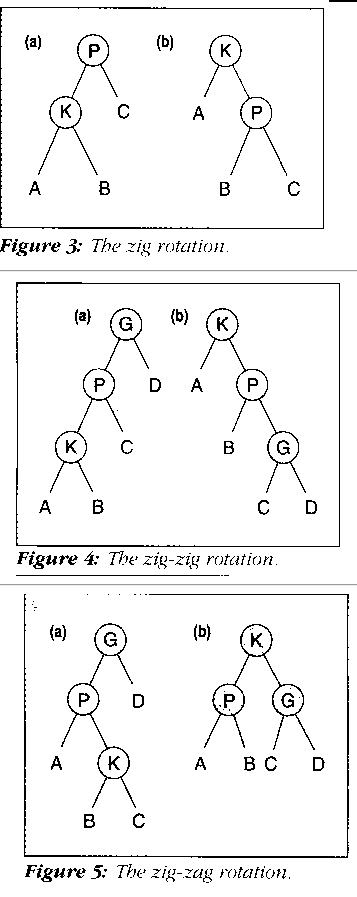
(redblacktrees, 2020)

**1.4**



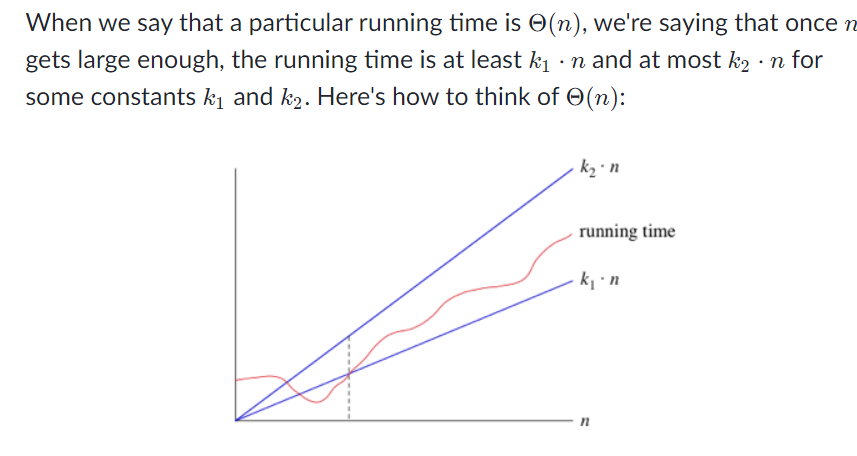
(redblacktrees, 2020)

**1.5**



(Figures 3, 4, and 5, 2020)

**1.6**



(Big-θ (Big-Theta) notation (article) | Khan Academy, 2020)

**References:**

* Algorithm Tutor. 2020. *2-3 Trees*. [online] Available at: <https://algorithmtutor.com/Data-Structures/Tree/2-3-Trees/> [Accessed 5 December 2020].
* Backtobackswe.com. 2020. *Back To Back SWE - Platform*. [online] Available at: <https://backtobackswe.com/platform/content/avl-trees-rotations/video> [Accessed 5 December 2020].
* Coursera. 2020. *2−3 Search Trees - Balanced Search Trees | Coursera*. [online] Available at: <https://www.coursera.org/lecture/algorithms-part1/2-3-search-trees-wIUNW?redirectTo=%2Flearn%2Falgorithms-part1%3Faction%3Denroll> [Accessed 5 December 2020].
* Coursera. 2020. *Red-Black Bsts - Balanced Search Trees | Coursera*. [online] Available at: <https://www.coursera.org/lecture/algorithms-part1/red-black-bsts-GZe13> [Accessed 5 December 2020].
* Cs.cornell.edu. 2020. *Lecture 11: Balanced Binary Trees: Red-Black Trees*. [online] Available at: <https://www.cs.cornell.edu/courses/cs3110/2009sp/lectures/lec11.html> [Accessed 5 December 2020].
* Cs.cornell.edu. 2020. *Splay Trees*. [online] Available at: <https://www.cs.cornell.edu/courses/cs3110/2009fa/Recitations/rec-splay.html> [Accessed 5 December 2020].
* Cs.iit.edu. 2020. *DEC92: SPLAY TREES*. [online] Available at: <http://www.cs.iit.edu/~cs535/splay.htm> [Accessed 5 December 2020].
* Cs.iit.edu. 2020. *Figures 3, 4, And 5*. [online] Available at: <http://www.cs.iit.edu/~cs535/figure345.htm> [Accessed 5 December 2020].
* Cs.lmu.edu. 2020. *Redblacktrees*. [online] Available at: <https://cs.lmu.edu/~ray/notes/redblacktrees/> [Accessed 5 December 2020].
* Cs.lmu.edu. 2020. *Splaytrees*. [online] Available at: <https://cs.lmu.edu/~ray/notes/splaytrees/> [Accessed 5 December 2020].
* Educative: Interactive Courses for Software Developers. 2020. *What Is A 2-3 Tree?*. [online] Available at: <https://www.educative.io/edpresso/what-is-a-23-tree> [Accessed 5 December 2020].
* GeeksforGeeks. 2020. *2-3 Trees | (Search And Insert) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/2-3-trees-search-and-insert/> [Accessed 5 December 2020].
* GeeksforGeeks. 2020. *AVL Tree | Set 1 (Insertion) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/> [Accessed 5 December 2020].
* GeeksforGeeks. 2020. *Difference Between Binary Tree And B-Tree - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/difference-between-binary-tree-and-b-tree/> [Accessed 5 December 2020].
* GeeksforGeeks. 2020. *Red-Black Tree | Set 1 (Introduction) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/> [Accessed 5 December 2020].
* GeeksforGeeks. 2020. *Splay Tree | Set 1 (Search) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/splay-tree-set-1-insert/> [Accessed 5 December 2020].
* Khan Academy. 2020. *Big-Θ (Big-Theta) Notation (Article) | Khan Academy*. [online] Available at: <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation> [Accessed 5 December 2020].
* Lafore, R. and Lafore, R., 2003. *LIBRARY USE ONLY Data Structures & Algorithms In Java*. Indianapolis, Ind.: Sams.
* OpenGenus IQ: Learn Computer Science. 2020. *Introduction To 2-3 Trees*. [online] Available at: <https://iq.opengenus.org/2-3-trees/> [Accessed 5 December 2020].
* Pages.cs.wisc.edu. 2020. *2-3 Trees*. [online] Available at: <http://pages.cs.wisc.edu/~vernon/cs367/notes/10.23TREE.html> [Accessed 5 December 2020].
* Pages.cs.wisc.edu. 2020. *AVL Trees*. [online] Available at: <http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html> [Accessed 5 December 2020].
* Pages.cs.wisc.edu. 2020. *Red-Black Trees*. [online] Available at: <http://pages.cs.wisc.edu/~hasti/cs367-common/readings/Red-Black-Trees/index.html> [Accessed 5 December 2020].
* Zaki, A., 1983. A comparative study of 2-3 trees and AVL trees. *International Journal of Computer & Information Sciences*, 12(1), pp.13-33.