Trees: A Comparison of Algorithms and Applications of Treaps, Red-Black Trees, and Ternary Search Trees

Nicole Hessner, Tyler Kepler, Matthew Thibault  
CS622 – Discrete Math and Algorithms for Computing,

Masters of Science in Computer Science,

School of Technology & Computing, City University of Seattle

[hessnernicole@cityuniversity.edu](mailto:hessnernicole@cityuniversity.edu), [keplertyler@cityuniversity.edu](mailto:keplertyler@cityuniversity.edu), [thibaultmatthew@cityuniversity.edu](mailto:thibaultmatthew@cityuniversity.edu)

**Abstract**

In mathematics and computing, trees are a data structure family that use hierarchy to represent and organize data. During this project, we will compare three specific types of tree data structures: Treaps, Red-Black Trees, and Ternary Search Trees. The three types of trees will be compared based on suitability for various applications, similarities and differences to each other structurally and algorithmically, and time and memory (Big O) complexity. Each data structure will be tested against both hard-coded and randomized data to determine which is most efficient for various operations. Since common applications for tree data structures include storing hierarchal information, searching stored data, and making decisions using mathematical probabilities, the applications researched and tested will be reflective of those three categories.

**Keywords:** computational complexity, tree data structure, treap, red-black tree, ternary search

# INTRODUCTION

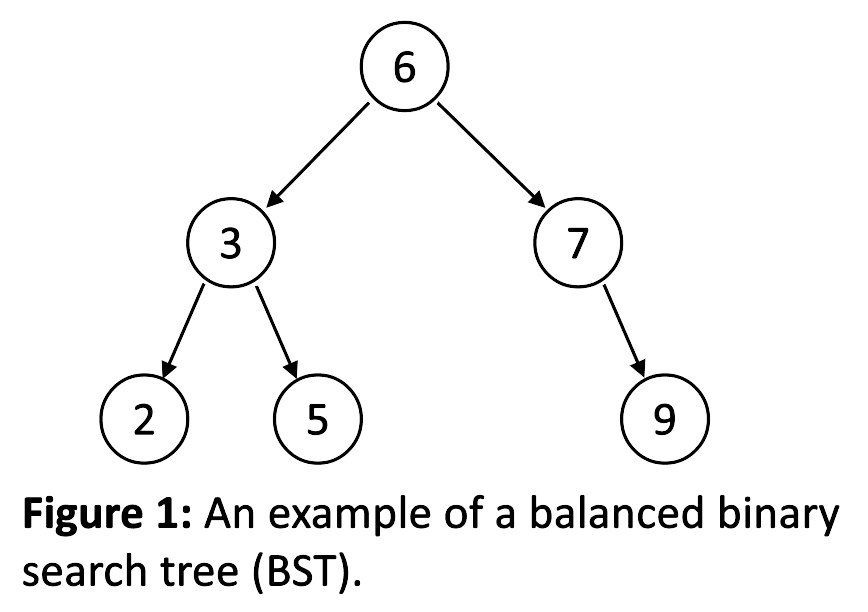
In graph theory, a tree is a type of graph in which all nodes are connected and there exists only one simple path between any two nodes. Trees are used in computing as a type of hierarchical data structure made up of nodes containing data points and pointers between nodes, but the overall structure is the same as in graph theory. Typically, each node contains a single data point and some number of pointers that denote the children of the node. The primary purpose of trees is to store data in an efficient manner such that updating the nodes of the tree and retrieving data from the tree is carried out as quickly as possible. While there are a wide variety of trees with more specific functions and unique features, this paper is primarily concerned with treaps, red-black trees, and ternary search trees.

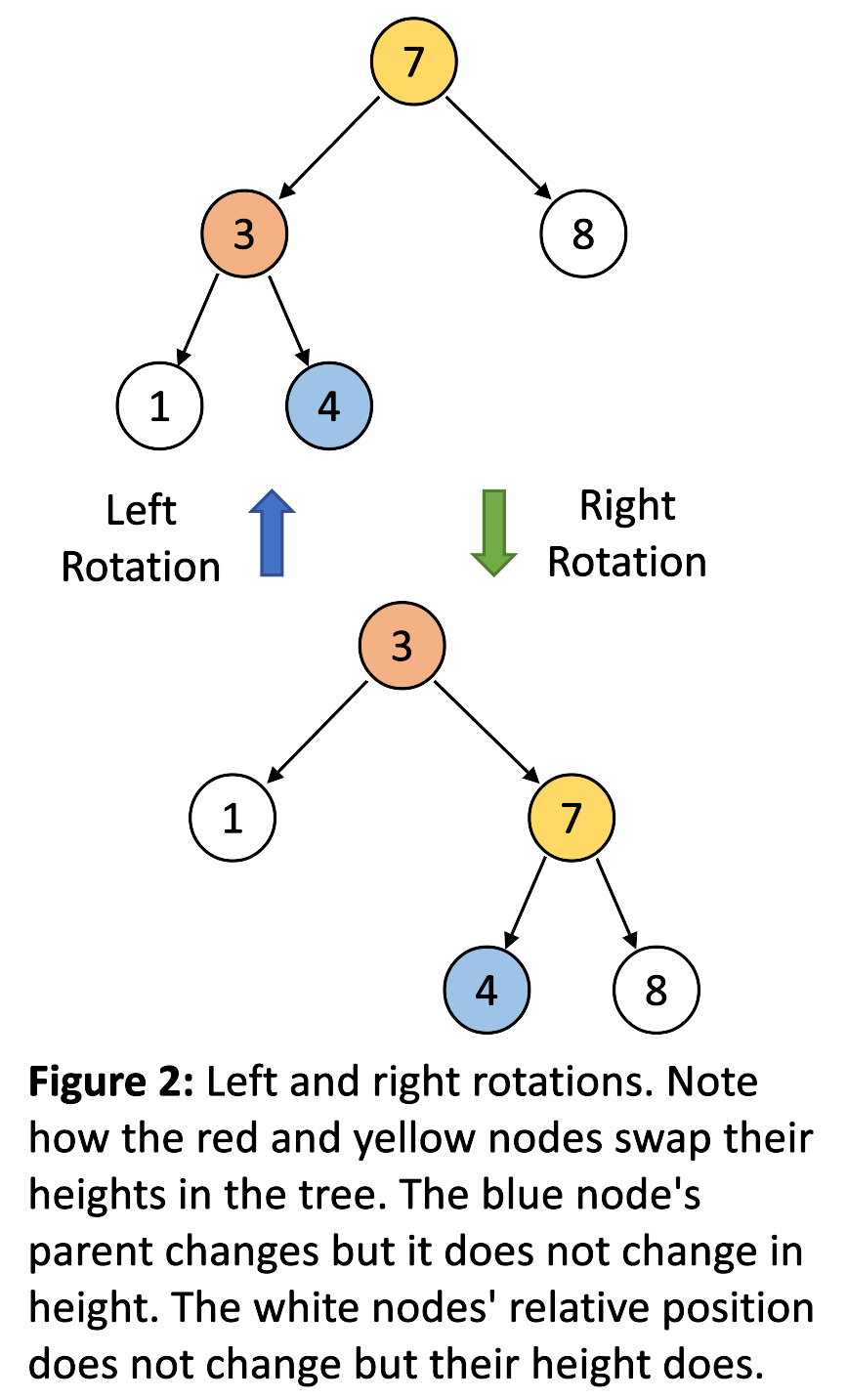
# 2. LITERATURE REVIEW

## Binary Search Trees

Binary search trees (BSTs) are binary tree data structures that follow three rules. First, the left subtree of any node must only have descendants with values that are less than the node. Second, the right subtree of any node must only have descendants with values that are greater than the node. These first two rules are what make a binary search tree searchable in an efficient manner. The final rule states that both subtrees of each node must also be BSTs (GeeksforGeeks, 2023). The ‘binary’ part of binary search tree refers to the additional criteria that each node has at most two children. An example of a binary search tree is shown in Figure 1.

All BSTs need to support some basic functions in order to be useful. In the simplest case, these are insertion, deletion, and search. New nodes in BSTs are always inserted at the leaves. Deletion can occur at any node and there are three cases. If the node to be deleted has no children then simply delete it. If the node to be deleted has 1 child, replace the deleted node with its child. If the node to be deleted has 2 children then some restructuring of the tree is required, specifically the deleted node must be replaced by its inorder successor. Note that the successor is treated as being deleted from its previous location and so the deletion algorithm should recursively continue from that node position until one of the 2 base cases is reached.



The primary goal of a BST is efficiency. Due to the nature of the tree structure, the worst-case time to search a BST for a value is O(log(n)) where n is the number of nodes in the tree. However, this is only true if the tree is balanced. A binary tree is considered balanced if for every node, the height difference of the left and right subtrees is no more than 1. The need to keep binary trees balanced introduces a new class of operations that are commonly called rotations. Rotations allow a BST to reorganize locally related nodes in order to make the tree more balanced. The two basic rotations are the left rotation and right rotation and are shown in Figure 2. However, these operations create a complication: if efficiency is the goal then making a BST very fast at searching (balanced) means that excessive restructuring may occur each time a node is inserted or deleted (rotations). To balance these two competing interests, additional criteria are required to improve the efficiency of the restructuring process while maintaining balance.

## Red-Black Trees

Red-Black Trees are a subset of binary search trees. A binary search tree is a red-black tree if it meets four additional criteria: each node in the tree must be either red or black; each leaf is black; for red nodes, each child is black; and lastly, every simple path from any node to any of its descendant leaves must have the same number of black nodes (Whitney, 1995). The first rule indicates that there are two kinds of nodes. To distinguish between these node types, each node must contain a new bit of information that determines its color. Since we are dealing with only 2 colors a simple 0 or 1 is enough. The leaves indicated in the second rule refer to the null nodes at the end of each branch of the tree, and not the last node in the tree to contain a value. The third rule says that the children of red nodes must be black; however, black nodes may have children of either color. The last criteria must hold true for all paths that descend from the right subtree or the left subtree: they must all have the same number of black nodes. This value is known as the black height. An example of a red-black tree and these rules is shown in Figure 3. For a red-black tree, the longest path from the root to a leaf will be no more than twice as long as the shortest path. This is because the shortest path may consist of only black nodes, while the longest path must have an equal number of black nodes (per rule 4) but may alternate between red and black nodes. Thus, at most there are an equal number of red and black nodes along the longest path (Woltmann, 2021).

Diagram

Description automatically generated

Red-black trees are among the fastest BSTs available and are not excessively difficult to implement, particularly if the implementation utilizes a simpler version called a left-leaning red-black tree (Sedgewick & Wayne, 2008). The difficulty with red-black trees in comparison to other BSTs is due to the management of the red and black nodes such that they don’t break rules 3 and 4. As nodes are added and deleted from the tree, some scanning, recoloring, and rotating of nearby nodes may also be required to manage the balance of the tree. For both insertion and deletion there are multiple cases that can occur that each require their own algorithms.

## Ternary Search Trees

Ternary Search Trees are a combination of digital search tries and binary search trees. They take the time efficiency from digital tries and the space efficiency from binary trees to create a tree structure that is faster than hashing for most search problems and supports a wider range of problems. Each node in a ternary search tree only contains three pointers, instead of the standard trie data structure where each node contains 26 pointers.

The node structure of a ternary search tree can be seen in figure 4 below.

**Figure 4: Structure of a Ternary Search Tree**

The five fields of a node are as follows; the “Data” field contains the character to be stored. The “EOS” field contains the end of string bit which allows the tree to determine if that is the end of a valid string. The left pointer points to the node whose value is less than the value in the current node. The equal (middle) pointer points to the node whose value is equal to the value in the current node. The right most pointer points to the node whose value is greater than the value in current node.

**Figure 5: Ternary Search Tree Example**

An example of implementing a ternary search tree can be seen above in figure 5. The tree starts with inserting the word “she” which can be seen by noticing the EOS field. Next the word “shell” is inserted which just adds existing nodes to the “she” tree.

Then the word “sell” is added, this can be seen by following the left pointer on “H”, “E” is less than “H” so the left pointer gets updated to point at “E” with the rest of the characters added as normal. In contrast the word “swap” is added, since “W” is greater than “H” the right pointer on “H” gets updated to point to “W” with the rest of the characters added as normal. Finally the word “sea” is added this is shown by following “S” to “H” but following the “E” branch where the left pointer on “E” gets updated to point to “A” and the EOS bit gets flagged.

Ternary search trees are a unique data structure and can grow very quickly as words and strings are added however this plays to the strengths of a ternary search tree due to its space complexity O(n). The average run time for insertion, search and deletion in big O is O(logn) while the worst case is linear time O(n) (GreeksForGeeks, 2023). Using big O notation is a little misleading when talking about the average run time because it is proportional to the height of the tree and that constant gets dropped in bigO. It is extremely important to create a balanced tree to avoid the worst case runtime, one method to create a balanced tree is inserting the words in a random order. Some practical applications of ternary search trees are auto-complete, spell checking, and near neighbor searching.

## Treaps

Treaps combine a binary search tree with heap elements to help maintain balance. Binary search trees have the best average performance for insert, remove, search, and min/max operations. Heaps are better for tracking priorities in a tree-like structure, however, and binary heaps are a sub-type of binary trees. Treaps can only be implemented on datasets with multiple parts, because unidimensional data cannot be constrained both horizontally by the BST structure and vertically by the heap structure at the same time. In situations where each entry has at least two values, one value (the key) can be treated like a BST, and the other (the priority) can be treated like a heap. Nodes on the same path from root to leaves are ordered with respect to their priority (La Roca, 2021).

One benefit of a treap is that it is easy to query by key, and the highest priority item will always be at the root of the tree. However, insertion and deletion operations are more complicated, because the both the BST and heap constraints must be met by the new node placement. Similarly to red-black trees, rotations can be used to maintain the constraints of a treap. In a rotation, the goal is to make the child node become the parent node and the parent node the child node. Another benefit of a treap is that it tends to produce a balanced tree, guaranteeing the longest path is O(log(n)) rather than O(n), as is possible with an unbalanced tree (La Roca, 2021).

# 3. METHODOLOGY

Having learned about our three data structures, we will create a simple program for each one, treating them as classes with associated functions. The classes will all include standard functions such as insert, delete, min, max, and search, as well as functions for storing hierarchal information, searching stored data, and making decisions using mathematical probabilities. Each operation will be performed against a small data set that is hard coded by the team, and a larger, randomized data set that is algorithmically generated. Each data structure’s algorithms will then be compared on time and memory bases to determine which structures operate best overall in the time category and the efficiency category (total combined time of all operations), as well as for each specific operation.

# 4. RESULTS

We chose to compare the time for the insertion and search functions in the Red-Black Tree and the Ternary Search Tree, since they are the two most comparable algorithms of the three we chose. Since the Treap would be more apt to searching for other characteristics based on a key, rather than searching for an item itself, we chose to leave it out in favor of a more direct comparison.

We used test code based on one of our assignments during the quarter, Programming Exercise 09, in which 1,000 words were randomly generated using the RandomWords module in Python. This set of words was what was used for inserting and searching.

Text

Description automatically generated

**Figure 6: Ternary Search Tree vs Red-Black Tree Results**

The insert function was 373% faster for the Red-Black Tree and search was 44% faster for the Red-Black Tree in our trial.

# 5. CONCLUSION

Red-Black Trees, Ternary Search Trees, and Treaps are effective ways to sort data. A Red-Black Tree is a refined Binary Search Tree that produces a consistently balanced tree, limiting the length of tree that needs to be traversed for insertion, deletion, and search operations. Ternary Search Trees take significantly more time in those operations than Red-Black Trees. Treaps combine functionality of both Binary Search Trees and Heaps, which allows for multi-dimensional sorting. This is most useful when there is both a key and some prioritization criteria that are both important for the sorting of data.

# 6. WORKLOAD ASSIGNMENT

## Nicole Hessner

Contributed literature review on treaps and methodology. Edited and formatted paper for submission.

## Tyler Kepler:

Researched ternary search trees and added findings to document.

## Matthew Thibault:

Contributed introduction and literature review on binary search trees and red-black trees.

# 7. REFERENCES

Anand, H. S., & Vinodchandra, S. S. (2016). Association rule mining using treap. International Journal of Machine Learning and Cybernetics, 9(4), 589–597. https://doi.org/10.1007/s13042-016-0546-7

Blelloch, G. E., & Reid-Miller, M. (1998). Fast set operations using treaps. Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures  - SPAA ’98. https://doi.org/10.1145/277651.277660Bentley, J., & Sedgewick, B. (1999). Ternary search trees. Computer Science Department. Retrieved January 29, 2023, from https://www.cs.upc.edu/~ps/downloads/tst/tst.html

Carpanese, I. (2022, May 18). A Visual Introduction to Treap Data Structure (Part I: The Basics). Medium. https://medium.com/carpanese/a-visual-introduction-to-treap-data-structure-part-1-6196d6cc12ee

GeeksforGeeks. (2023, January 27). Introduction to Tree – Data Structure and Algorithm Tutorials. <https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/>

GeeksforGeeks. (2023, January 18). Ternary search tree. GeeksforGeeks. Retrieved January 29, 2023, from <https://www.geeksforgeeks.org/ternary-search-tree/>

GeeksforGeeks. (2023, February 15). *Binary*  *search tree*. https://www.geeksforgeeks.org/binary- search-tree-data-structure/

La Rocca, M. L. (2021). Advanced Algorithms and Data Structures. Manning Publications Co. https://learning.oreilly.com/library/view/advanced-algorithms-and/9781617295485/OEBPS/Text/ch03.htm

Pfaff, B. (2004, June). Performance Analysis of BSTs in System Software. Stanford. Retrieved January 29, 2023, from <https://web.stanford.edu/~blp/papers/libavl.pdf>

Sedgewick, R., & Wayne, K. (2008, October 13). Balanced Trees. Princeton University. Retrieved January 29, 2023, from https://www.cs.princeton.edu/courses/archive/fall08/cos226/lectures/10BalancedTrees-2x2.pdf

Ternary search trees. Top Website Designers, Developers, Freelancers for Your Next Project. (n.d.). Retrieved January 29, 2023, from <https://www.topcoder.com/thrive/articles/ternary-search-trees>

Whitney, R. (1995, August 17). Red-Black and B Trees. CS660: Combinatorial Algorithms. Retrieved January 29, 2023, from <http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html>

Woltmann, S. (2021, September 29). *Red-black*  *tree (fully explained, with Java code)* HappyCoders. https://www.happycoders.eu/algorithms/red-black-tree-java/