COMP-SCI 5501 Assignment 4
Application of Tree Structures & Algorithms

Group 2: Bosia N'dri, Joel Vinas, Josh Oursler, Nimisha Chandra, & Sarah Newell

Abstract

Problem 1 (AVL Tree vs Red-Black Tree)

This experiment evaluates the performance differences between AVL Trees and Red-Black Trees using 1000 frequent English words as input. Although both are self-balancing binary search trees with O(log n) complexity, their balancing strategies differ significantly. AVL Trees enforce strict height balancing, minimizing height variations across the tree, while Red-Black Trees maintain a more relaxed balance structure to reduce the number of rotations during insertion and deletion. In this study, both trees were constructed using the same dataset, and their search behavior was tested using 100 randomly sampled words. Comparisons were counted to quantify search efficiency. The results confirm that AVL Trees perform fewer comparisons on average due to their tighter height constraints, while Red-Black Trees—although slightly slower for searching—remain more efficient in insertion and deletion operations because they require fewer rotations to maintain balance.

Problem 2 (B-Tree vs Python List Search)

This experiment investigates the search performance of a B-Tree compared to a built-in Python List when storing and retrieving customer data. A B-Tree of minimum degree t = 3 was constructed using customer first names as keys, while the entire dataset was also stored in a standard Python List. One hundred first names were randomly selected and searched in both structures, tracking the number of key comparisons to measure efficiency. The B-Tree achieved significantly fewer comparisons due to its logarithmic height and multi-key node design, while the List required linear scans. The results demonstrate that B-Trees are considerably more efficient for search operations, validating their widespread use in database indexing and large-scale data management systems.

Problem 3 (k-labeling for Graphs)

This experiment explores vertex k-labeling on a Perfect Ternary Tree in Python to demonstrate how algorithmic graph labeling techniques can satisfy the theoretical properties of k-labeling. The graph was represented in python using dictionaries, and an algorithm was designed to assign vertex labels within the theoretical bounds. Breadth-First Search and Depth-First Search were applied to organize the tree by levels, determine parent-child relationships, and provide a traversal order for labeling. The resulting labels and edge weights were output into a table to be validated against k-labeling properties. The final output confirmed that all edges in the tree

received unique weights spanning within the expected range, demonstrating that the algorithm successfully produced valid k-labeling for a Perfect Ternary Tree.

Introduction

Problem 1

Red-Black Trees are a form of self-balancing Binary Search Tree (BST). A standard BST node has three attributes: the key or value it holds, a pointer to the left child node, and a pointer to the right child node [1]. Red-Black Tree (RBT) nodes have an additional attribute, a flag for the color of the node, which is either red or black. By storing the color of each node, the RBT maintains balance by meeting the following requirements as nodes are added or removed [1]:

1. Nodes must either be red or black
2. The root of the tree must be black
3. A red node cannot have a red parent
4. All paths from a node to its NIL pointers, also known as leaves, must have the same number of black nodes
5. All leaves (NIL pointers) must be black

The requirements use color as a flag to determine when nodes should be rotated to another branch or recolored to prevent a tree from becoming unbalanced [2]. The height of the tree will always be O(log n) while these conditions are met, thus the time complexity of insertion, deletion, and search is also O(log n) [1].

AVL Trees are another form of self-balancing BST with time complexity O(log n) for insertion, deletion and search. (SOURCE) In addition to the standard BST attributes, AVL nodes store the height of the subtree rooted at that node. [3] The AVL Tree maintains balance by requiring that the balance factor, or the height difference between the left and right subtrees, is within [-1, 0, 1]. This means that the heights on each side cannot differ by more than one. [3] If the balance factor falls outside of the allowable range as nodes are inserted or deleted, the subtree is rotated to regain balance [4].

Problem 2

M-ary trees are trees where each internal node has up to *m* children [5]. For example, both of the self-balancing Binary Search Trees we reviewed in Problem 1 are m-ary trees where m = 2. A B-Tree is also a self-balancing m-way tree, however, it can have more than two children. B-Trees can store up to *m-1* keys within a single node, allowing them to have a reduced height and use fewer disk operations [6]. This makes them a strong choice for large data and file management systems [5].

B-Trees maintain balance by meeting the following requirements at each insertion and deletion [6]:

1. All leaf nodes are at the same level of the tree

2. The keys of each node are stored in ascending order
3. Each Node may contain at most *m-1* keys and at most *m* children
4. The root must either be a leaf or, if internal, have at least two children
5. All internal nodes must have a minimum of *m/2* children
6. All nodes, except the root, must have a minimum of *m/2 - 1* keys and at most *m-1* keys

These properties guarantee a height-balanced tree where each node has a bounded number of keys, maintaining O(log n) time complexity for insertion, deletion and search [6].

Problem 3

A Ternary Tree is an m-ary tree where *m = 3*, or each node has up to three children. In a Perfect Ternary Tree, each internal node has exactly three children, all nodes are populated, and all leaves live at the same level [5]. This results in a balanced and symmetrical tree structure. The height *h* of the Perfect Ternary Tree is calculated as:

$$h = log_m n$$

The total number of nodes in the tree, *n*, is calculated as:

$$n = \frac{(3^{h+1}-1)}{2}$$

A graph is a set of vertices (nodes) and edges that connect any two nodes. A tree is a type of graph that is fully connected, has no cycles, and has exactly one path between any two nodes. Thus, a tree can be expressed as a graph $G = (V, E)$, where *V* is the number of vertices and *E* is the number of edges. The number of edges is always $|E| = |V| - 1$. [5]

Representing a tree as a connected acyclic graph allows us to apply graph labeling methods to the structure. Vertex k-labeling, the method of focus in Problem 3, is a graph labeling technique that assigns an integer to each vertex *v* through labeling function:

$$\phi : V(G) \rightarrow \{1, 2, ..., k\}$$

After labels are assigned, the weight (w_ϕ) of each edge of the graph is calculated as the sum of the labels assigned to its endpoint vertices. For example, if an edge is connected to nodes x and y, w_ϕ is computed as:

$$w_\phi(xy) = \phi(x) + \phi(y)$$

K-labeling requires that no two edges have the same weight. The minimum value of *k,* or the size of the label set, is referred to as the edge irregularity strength, es(G) [7]:

$$es(G) \geq max\left\{\left\lceil \frac{|E(G)|+1}{2}, \Delta(G)\right\rceil\right\}$$

K-labeling is useful because it assigns distinct weights to all edges while using the smallest possible label range. The resulting edge weights act as unique identifiers, while the minimized label set contributes to efficient storage and computation.

Objectives

Problem 1

Both RBT and AVL Trees are forms of self-balancing binary search trees, each with their own strengths. AVL Trees maintain a strict balance in height between the right and left subtrees of any node, enabling faster performance than RBT. However, more rotations are required to maintain the strict balance in AVL Trees, meaning inserting and deleting nodes can be slower in AVL Trees than RBT. [5] Problem 1 challenges us to prove that AVL trees are more efficient for searching, while RBT are faster for insertion and deletion.

Problem 2

B-Trees are self-balancing m-ary trees that maintain O(log n) time complexity for insertion, deletion, and search, making them useful in handling large amounts of data [6]. By storing multiple sorted keys per node and requiring equal leaf depth, B-Trees reduce the height that would be needed to store the same data in a Binary Search Tree. This results in an efficient use of storage space and less key comparisons needed during search [5]. The goal of Problem 2 is to validate that searching for a key within a B-Tree is faster than searching a built-in Python Collection, such as a List.

Problem 3

Perfect Ternary Trees are connected, acyclic graphs, allowing for graph labeling techniques to be applied. Problem 3 tasks us with applying vertex k-labeling to a Perfect Ternary Tree, demonstrating that an algorithm can assign labels such that all edge weights remain unique while satisfying the minimum edge irregularity strength bound. The challenge problem asked similar questions of a Homogeneous Amalgamated Star rather than a m-ary tree.

Experiment

Problem 1

To prove that AVL trees are more efficient for searching, while RBT are faster for insertion and deletion, we stored the same list of 1000 words into both an AVL and Red-Black Trees in Python. We then randomly selected 100 words from the list. Each tree's search function is

applied to the same 100 words. The time taken to find each word, along with the number of comparisons, is recorded to determine which type of tree is faster for searching.

Problem 2

To evaluate the search efficiency of a B-Tree compared to a Python built-in Collection, we stored a file of customer contact information into both a B-Tree and a List. The B-Tree uses the first name of the customer as the key value. Similar to the approach in Problem 1, we randomly selected 100 first-names from the file to search for in both the B-Tree and List. Based on the number of comparisons performed in each search, we can conclude whether a B-Tree or List is faster for searching.

Problem 3

To demonstrate how k-labeling can be applied to a Perfect Ternary Tree through an algorithmic approach, we first stored the graph into two dictionaries. The first dictionary stores each vertex as a key and a list of its adjacent vertices. The second dictionary stores a list of edges. We then created a function in Python to create a Perfect Ternary Tree of $T_{3,5}$ with 40 vertices and 39 edges. Based on the properties of a Perfect Ternary Tree, we know $k=\lceil V/2 \rceil$ or in this scenario k = 20 [7]. We then built a function to apply k-labeling so that the size of the label set is equal to 20 while maintaining unique edge weights, satisfying the edge irregularity constraint of k-labeling.

The next component of Problem 3 is to apply traversing methods to the tree. Both Breadth-First Search and Depth-First Search are applied. BFS organized the vertices by level to establish parent-child relationships, while DFS provided the order used to add labels onto the tree.

Results

Problem 1

A set of 100 random words was generated for search comparisons. The performance metrics collected during insertion, searching, and deletion are summarized below:

- **AVL Tree**
    - Time to load (insertion): **6970 µs**
    - Total comparisons during search: **913**
    - Average comparisons per search: **9.13**
    - Time taken for 100 searches: **98 µs**

    - Time for deletion: **758 µs**

- **Red-Black Tree**
    - Time to load (insertion): **1960 µs**
    - Total comparisons during search: **922**
    - Average comparisons per search: **9.22**
    - Time taken for 100 searches: **123 µs**

```
Source File downloaded successfully to: /tmp/1000 Frequent Words.txt
Inserted all words into both trees successfully!
Generated 100 random words for searching.

===== Search Comparison Results (100 random words) =====
Time taken to load AVL Tree:        6970.000 microseconds
Total comparisons in AVL Tree:       913
Average comparisons per search AVL:  9.13
Time taken for AVL Search:           98.000 microseconds
Time taken for AVL Deletion:           758.000 microseconds
Time taken to load Red-Black Tree:        1960.000 microseconds
Total comparisons in Red-Black Tree: 922
Average comparisons per search RB:   9.22
Time taken for Red-Black Search:      123.000 microseconds
```

Problem 2

A B-Tree (minimum degree *t* = *3*) was constructed using customer first names as keys. The same data was stored in a Python List.
Then, 100 first names were randomly selected and searched in both structures, with comparison counts recorded.

```
B-Tree: hits=100/100, avg=5.30, min=1, max=10
List:   hits=100/100, avg=47.34, min=1, max=100
First 10 B-Tree comparisons: [8, 7, 4, 1, 5, 3, 5, 10, 8, 6]
First 10 List comparisons: [3, 24, 96, 7, 83, 34, 14, 26, 57, 38]
```

Problem 3

K-labeling was applied to the Perfect Ternary Tree, and the resulting edges and labels were recorded in a comparison table, shown below.

Comparison Table:

Vertex	Label	Adjacency List	Edge List
0	1	[1, 2, 3]	['(1, 1, 2)', '(1, 10, 11)', '(1, 20, 21)']
1	1	[0, 4, 5, 6]	['(1, 1, 2)', '(1, 2, 3)', '(1, 6, 7)', '(1, 11, 12)']
2	10	[0, 7, 8, 9]	['(10, 1, 11)', '(10, 6, 16)', '(10, 10, 20)', '(10, 15, 25)']
3	20	[0, 10, 11, 12]	['(20, 1, 21)', '(20, 20, 40)', '(20, 16, 36)', '(20, 12, 32)']
4	2	[1, 13, 14, 15]	['(2, 1, 3)', '(2, 2, 4)', '(2, 3, 5)', '(2, 4, 6)']
5	6	[1, 16, 17, 18]	['(6, 1, 7)', '(6, 2, 8)', '(6, 3, 9)', '(6, 4, 10)']
6	11	[1, 19, 20, 21]	['(11, 1, 12)', '(11, 2, 13)', '(11, 3, 14)', '(11, 4, 15)']
7	6	[2, 22, 23, 24]	['(6, 10, 16)', '(6, 11, 17)', '(6, 12, 18)', '(6, 13, 19)']
8	10	[2, 25, 26, 27]	['(10, 10, 20)', '(10, 12, 22)', '(10, 13, 23)', '(10, 14, 24)']
9	15	[2, 28, 29, 30]	['(15, 10, 25)', '(15, 11, 26)', '(15, 12, 27)', '(15, 13, 28)']
10	20	[3, 31, 32, 33]	['(20, 20, 40)', '(20, 19, 39)', '(20, 18, 38)', '(20, 17, 37)']
11	16	[3, 34, 35, 36]	['(16, 20, 36)', '(16, 19, 35)', '(16, 18, 34)', '(16, 17, 33)']
12	12	[3, 37, 38, 39]	['(12, 20, 32)', '(12, 19, 31)', '(12, 18, 30)', '(12, 17, 29)']
13	2	[4]	['(2, 2, 4)']
14	3	[4]	['(3, 2, 5)']
15	4	[4]	['(4, 2, 6)']
16	2	[5]	['(2, 6, 8)']
17	3	[5]	['(3, 6, 9)']
18	4	[5]	['(4, 6, 10)']
19	2	[6]	['(2, 11, 13)']
20	3	[6]	['(3, 11, 14)']
21	4	[6]	['(4, 11, 15)']
22	11	[7]	['(11, 6, 17)']
23	12	[7]	['(12, 6, 18)']
24	13	[7]	['(13, 6, 19)']
25	12	[8]	['(12, 10, 22)']
26	13	[8]	['(13, 10, 23)']
27	14	[8]	['(14, 10, 24)']
28	11	[9]	['(11, 15, 26)']
29	12	[9]	['(12, 15, 27)']
30	13	[9]	['(13, 15, 28)']
31	19	[10]	['(19, 20, 39)']
32	18	[10]	['(18, 20, 38)']
33	17	[10]	['(17, 20, 37)']
34	19	[11]	['(19, 16, 35)']
35	18	[11]	['(18, 16, 34)']
36	17	[11]	['(17, 16, 33)']
37	19	[12]	['(19, 12, 31)']
38	18	[12]	['(18, 12, 30)']
39	17	[12]	['(17, 12, 29)']

Graph Statistics:
Total edges (E): 39
Minimum weight: 2
Maximum weight: 40

For the Challenge problem, similar results to the m-ary tree were found with slight alterations based on the new graph layout:

Comparison Table:

Vertex	Label	Adjacency List	Edge List
0	1	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]	['(1, 1, 2)', '(1, 3, 4)', '(1, 5, 6)', '(1, 7, 8)', '(1, 10, 11)', '(1, 12, 13)', '(1, 14, 15)', '(1, 16, 17)', '(1, 19, 20)', '(1, 21, 22)', '(1, 23, 24)', '(1, 25, 26)']
1	1	[0, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]	['(1, 1, 2)', '(1, 2, 3)', '(1, 4, 5)', '(1, 6, 7)', '(1, 8, 9)', '(1, 9, 10)', '(1, 11, 12)', '(1, 13, 14)', '(1, 15, 16)', '(1, 17, 18)', '(1, 18, 19)', '(1, 20, 21)', '(1, 22, 23)']
2	3	[0, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36]	['(3, 1, 4)', '(3, 22, 25)', '(3, 24, 27)', '(3, 25, 28)', '(3, 26, 29)', '(3, 27, 30)', '(3, 28, 31)', '(3, 29, 32)', '(3, 30, 33)', '(3, 31, 34)', '(3, 32, 35)', '(3, 33, 36)', '(3, 34, 37)']
3	5	[0, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48]	['(5, 1, 6)', '(5, 33, 38)', '(5, 34, 39)', '(5, 35, 40)', '(5, 36, 41)', '(5, 37, 42)', '(5, 38, 43)', '(5, 39, 44)', '(5, 40, 45)', '(5, 41, 46)', '(5, 42, 47)', '(5, 43, 48)', '(5, 44, 49)']
4	7	[0]	['(7, 1, 8)']
5	10	[0]	['(10, 1, 11)']
6	12	[0]	['(12, 1, 13)']
7	14	[0]	['(14, 1, 15)']
8	16	[0]	['(16, 1, 17)']
9	19	[0]	['(19, 1, 20)']
10	21	[0]	['(21, 1, 22)']
11	23	[0]	['(23, 1, 24)']
12	25	[0]	['(25, 1, 26)']
13	2	[1]	['(2, 1, 3)']
14	4	[1]	['(4, 1, 5)']
15	6	[1]	['(6, 1, 7)']
16	8	[1]	['(8, 1, 9)']
17	9	[1]	['(9, 1, 10)']
18	11	[1]	['(11, 1, 12)']
19	13	[1]	['(13, 1, 14)']
20	15	[1]	['(15, 1, 16)']
21	17	[1]	['(17, 1, 18)']
22	18	[1]	['(18, 1, 19)']
23	20	[1]	['(20, 1, 21)']
24	22	[1]	['(22, 1, 23)']
25	22	[2]	['(22, 3, 25)']
26	24	[2]	['(24, 3, 27)']
27	25	[2]	['(25, 3, 28)']
28	26	[2]	['(26, 3, 29)']
29	27	[2]	['(27, 3, 30)']
30	28	[2]	['(28, 3, 31)']
31	29	[2]	['(29, 3, 32)']
32	30	[2]	['(30, 3, 33)']
33	31	[2]	['(31, 3, 34)']
34	32	[2]	['(32, 3, 35)']
35	33	[2]	['(33, 3, 36)']
36	34	[2]	['(34, 3, 37)']
37	33	[3]	['(33, 5, 38)']
38	34	[3]	['(34, 5, 39)']
39	35	[3]	['(35, 5, 40)']
40	36	[3]	['(36, 5, 41)']
41	37	[3]	['(37, 5, 42)']
42	38	[3]	['(38, 5, 43)']
43	39	[3]	['(39, 5, 44)']
44	40	[3]	['(40, 5, 45)']
45	41	[3]	['(41, 5, 46)']
46	42	[3]	['(42, 5, 47)']
47	43	[3]	['(43, 5, 48)']
48	44	[3]	['(44, 5, 49)']

```
Graph Statistics:
Total edges (E): 48
Minimum weight: 2
Maximum weight: 49
```

Explanation

Problem 1 (AVL vs RBT Search Performance)

From the measurements, the **Red-Black Tree** inserts significantly faster than the **AVL Tree**, requiring only **1960 µs** compared to **6970 µs**. This aligns with theoretical expectations: AVL trees maintain stricter balancing, resulting in more rotations during insertion, which increases the insertion cost.

In terms of search operations, both trees exhibit nearly identical comparison counts—**9.13 vs. 9.22** on average—since both are height-balanced and maintain O(logn)O(\log n)O(logn) search complexity. However, the **AVL tree still achieves slightly faster search time (98 µs)** than the Red-Black Tree (123 µs), again consistent with theory: AVL trees are more strictly balanced, often yielding shallower heights and hence fewer pointer traversals in practice.

Deletion in the AVL tree required **758 µs**. While a Red-Black deletion measurement wasn't included in the displayed output, AVL deletions are generally more rotation-heavy, so the observed value appears reasonable.

Overall the result is AVL trees incur more overhead during updates but reward with faster lookups; Red-Black trees prioritize moderately faster updates with slightly slower searches.

Problem 2 (B-Tree vs List Search Performance)

The B-Tree efficiently organizes keys in a multi-level, height-balanced structure where each node contains multiple sorted keys and multiple children. This greatly reduces the height of the tree and allows searches to skip large portions of the dataset at each step. Each search operation involves:

1. Checking a small number of keys within a node, and

2. Deciding which single child branch to descend into.

This structure ensures **O(log n)** performance and typically only a handful of comparisons per search.
Your B-Tree results (avg 5.30 comparisons) reflect exactly this behavior.

In contrast, a Python List performs a **linear search**. Each item must be checked sequentially until the key is found. This results in:

- **O(n)** search time,

- High variance (sometimes 3 comparisons, sometimes 96),

- A worst case of 100 comparisons for 100 records.

The average of **47.34 comparisons per search** matches expectations for scanning a list of 100 items.

Thus, the experiment directly demonstrates how B-Trees outperform Lists in lookup efficiency, especially as dataset size grows.

<u>Problem 3 and Challenge Problem</u>

The output confirms that all 40 vertices were assigned valid labels consistent with the rules of k-labeling. K = 20, and all 39 edges have unique weights, satisfying the edge irregularity strength condition.

The challenge problem was merely a derivative of the m-ary tree previously implemented. With minor changes to the creation of the graph, an m-ary tree can be easily altered to a homogeneous amalgamated star graph. By altering the labeling function based on the new graph format, the implementation created the k-labels for the vertices based on the stated parameters. For a star with m=12 and n=4, all of the edges had been created with unique weights.

Graph Statistics:
Total edges (E): 48
Minimum weight: 2
Maximum weight: 49

Conclusion

Problem 1 (AVL vs RBT Search Performance)

The experimental results reinforce the theoretical behavior of AVL and Red-Black trees.

- **AVL Trees**:
    - Slower insertions due to strict balancing
    - Slightly faster searches because of better height uniformity
- **Red-Black Trees**:
    - Faster insertion and rebalancing
    - Search performance very close to AVL but marginally slower

Based on these findings, AVL trees are preferable when search performance is the priority, while Red-Black trees remain advantageous for workloads that involve frequent insertions or updates. The comparison shows that although both structures maintain logarithmic efficiency, their real-world performance reflects predictable trade-offs in balancing strategies.

Problem 2 (B-Tree vs List Search Performance)

The experiment confirms that **B-Trees provide significantly faster search performance** compared to sequential List searching. While the List required nearly 47 comparisons per lookup, the B-Tree needed only about 5. The B-Tree's balanced structure and multi-key nodes ensured that search time remained consistently low, even in the worst case.

These results validate the theoretical advantages of B-Trees:

- Superior search time due to logarithmic tree height
- Efficient node structure for minimizing comparisons
- Stable performance for large datasets

Problem 3

The experiment successfully implemented vertex k-labeling on a Perfect Ternary Tree, confirming that an algorithmic approach can achieve the edge irregularity strength required. The

labeling algorithm, supported by both Breadth-First Search and Depth-First Search traversal, produced a valid set of vertex labels with unique edge weights within the expected range. The computed statistics, 39 edges with weights from 2 to 40, align with the theoretical properties of k-labeling ternary trees. Overall, the results validate that the chosen graph, traversal, and labeling method are an effective way to generate k-labeling programmatically.

Colab Notebooks

Problem 1

https://colab.research.google.com/drive/1FLzmvHP2LBytsADXRTmLQmORh_djLU-V?usp=sharing

Problem 2

https://colab.research.google.com/drive/1vbVmkiVlM6sigAPSMw__YE0uH1JQNhAa?usp=sharing

Problem 3

https://colab.research.google.com/drive/1bQ5L8nvtSJwybE6p2hOXM9Wy73AHBMl3?usp=sharing

Challenge Problem

https://colab.research.google.com/drive/19b1oOx1duYECBSplC2hf7lLHZO9ovlrm?usp=sharing

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[3] E. Demaine, “Lecture 06: AVL Trees, AVL Sort,” *Introduction to Algorithms*, MIT OpenCourseWare, Massachusetts Institute of Technology, 2005. [Online]. Available: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/lecture-6-avl-trees-avl-sort/

[4] “AVL Tree Data Structure,” *GeeksforGeeks*, https://www.geeksforgeeks.org/avl-tree-data-structure/ (accessed Nov 24, 2025).

[5] Dr. A. Asim, “Non-Linear Data Structures”, Advanced Computational Thinking & Programming, UMKC, Kansas City, MO, 2025 [PowerPoint slides]

[6] “Introduction of B Tree.” GeeksforGeeks. https://www.geeksforgeeks.org/dsa/introduction-of-b-tree-2/ (accessed Nov. 24, 2025).

[7] A. Asim, *Application of Tree Structure & Algorithms in Different Fields of Sciences (Assignment 4)*, ACT & Programming 5501, Univ. of Missouri–Kansas City, 2025.

[8] M. Shahzad, “Edge Irregular k-labeling of Homogeneous Amalgamated Star S 12,4,” *Computing Edge Irregularity Strength of Star and Banana Trees Using Algorithmic Approach*, Aug. 2024. Available: https://www.researchgate.net/figure/Edge-Irregular-k-labeling-of-Homogeneous-Amalgamated-Star-S-12-4_fig1_383433013 (researchgate.net)