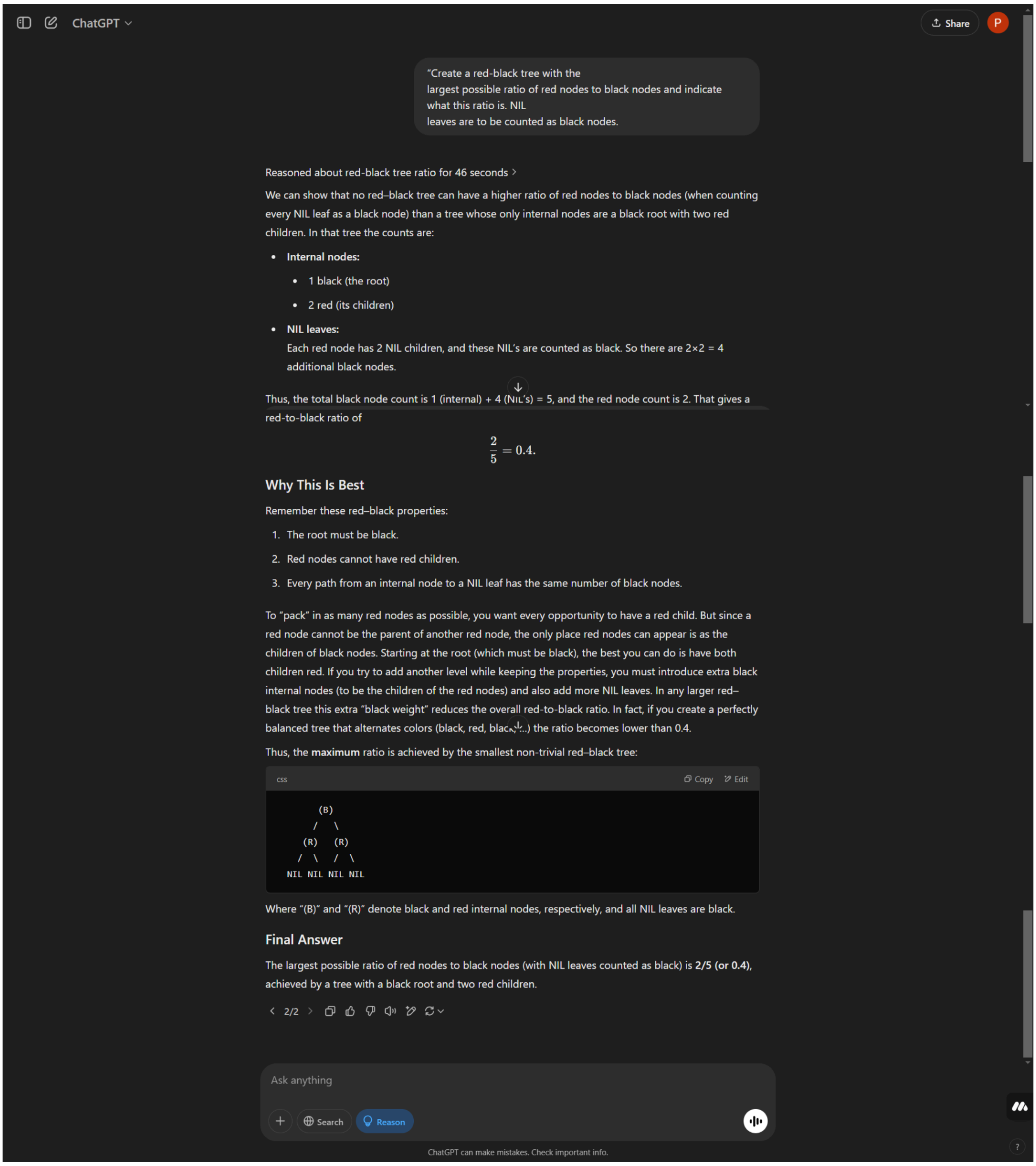
Assignment 1

Question 1: Red-Black Trees

Problem: Create a red-black tree with the largest possible ratio of red nodes to black nodes, with NIL leaves counted as black nodes.

(a) OpenAI-ChatGPT Answer (A1)



ChatGPT

Share

"Create a red-black tree with the largest possible ratio of red nodes to black nodes and indicate what this ratio is. NIL leaves are to be counted as black nodes.

Reasoned about red-black tree ratio for 46 seconds

We can show that no red–black tree can have a higher ratio of red nodes to black nodes (when counting every NIL leaf as a black node) than a tree whose only internal nodes are a black root with two red children. In that tree the counts are:

- **Internal nodes:**
  - 1 black (the root)
  - 2 red (its children)
- **NIL leaves:**
  Each red node has 2 NIL children, and these NIL's are counted as black. So there are 2×2 = 4 additional black nodes.

Thus, the total black node count is 1 (internal) + 4 (NIL's) = 5, and the red node count is 2. That gives a red-to-black ratio of

$$\frac{2}{5} = 0.4.$$

Why This Is Best

Remember these red–black properties:

1. The root must be black.
2. Red nodes cannot have red children.
3. Every path from an internal node to a NIL leaf has the same number of black nodes.

To "pack" in as many red nodes as possible, you want every opportunity to have a red child. But since a red node cannot be the parent of another red node, the only place red nodes can appear is as the children of black nodes. Starting at the root (which must be black), the best you can do is have both children red. If you try to add another level while keeping the properties, you must introduce extra black internal nodes (to be the children of the red nodes) and also add more NIL leaves. In any larger red–black tree this extra "black weight" reduces the overall red-to-black ratio. In fact, if you create a perfectly balanced tree that alternates colors (black, red, black, ...) the ratio becomes lower than 0.4.

Thus, the **maximum** ratio is achieved by the smallest non-trivial red–black tree:

css

Copy Edit

```
       (B)
      /   \
    (R)   (R)
   / \   / \
  NIL NIL NIL NIL
```

Where "(B)" and "(R)" denote black and red internal nodes, respectively, and all NIL leaves are black.

Final Answer

The largest possible ratio of red nodes to black nodes (with NIL leaves counted as black) is **2/5 (or 0.4)**, achieved by a tree with a black root and two red children.

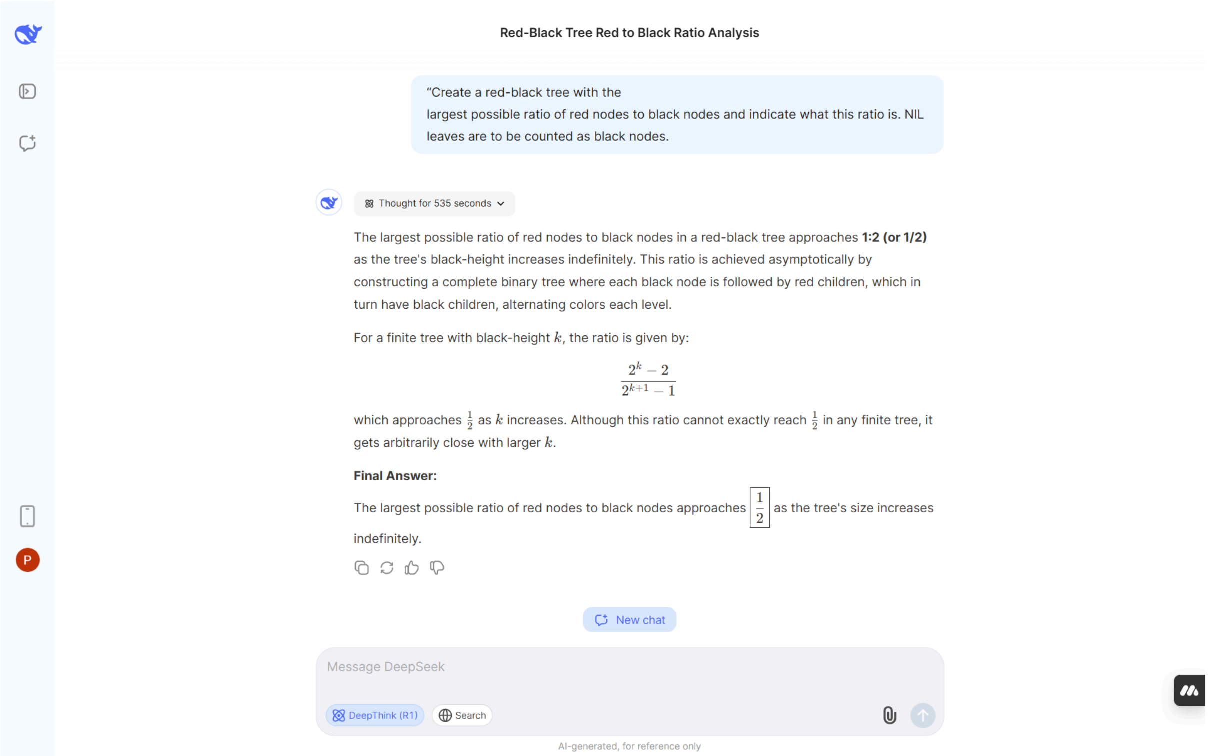
2/2

Ask anything

Search Reason

ChatGPT can make mistakes. Check important info.

(b) DeepSeek R1 Answer (A2)



(c) Critical Evaluation of A1 and A2

Evaluation of A1 (OpenAI-ChatGPT)

Answer Provided: A1 claims that the largest possible ratio of red nodes to black nodes in a red-black tree is 2/5 (0.4), achieved by a tree with a black root, two red children, and four black NIL leaves (1 black internal + 4 black NILs = 5 black nodes, 2 red nodes).

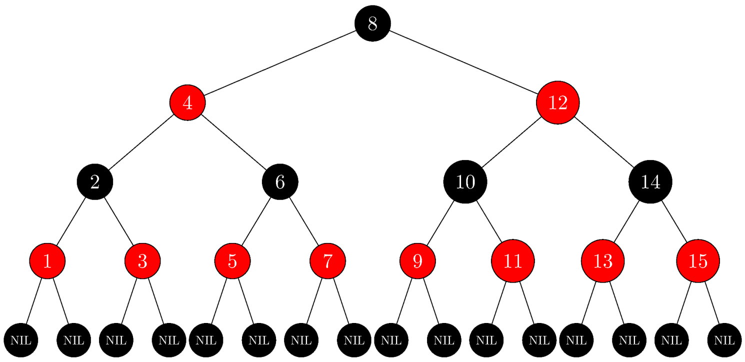
Correctness: Incorrect.

Positives:

- Provides a specific, concrete example of a red-black tree: a black root with two red children, each having two black NIL leaves.
- Correctly computes the ratio for this tree: 2 red nodes / 5 black nodes = 2/5 = 0.4.
- Lists the red-black tree properties and uses them to argue why this configuration might maximize the ratio.

Negatives:

- Incorrectly asserts that this is the maximum possible ratio. Larger valid red-black trees can achieve higher ratios (e.g., 10/21 ≈ 0.4762), contradicting the claim that adding more levels reduces the ratio below 0.4.

   - e.g., 

- Lacks rigorous proof to show that no tree can exceed a ratio of 2/5. The argument about “extra black weight” is intuitive but not substantiated with a general analysis.
- Does not consider larger trees systematically, limiting the scope to a small, specific case without exploring the asymptotic behavior.

Evaluation of A2 (DeepSeek R1)

Answer Provided: A2 states that the largest possible ratio approaches 1/2 as the tree’s black-height increases indefinitely, with a formula for a finite tree of black-height k: $\frac{2^k-2}{2^{k+1}-1}$, which approaches 1/2 as k grows.

Correctness: Correct.

Positives:

- Accurately identifies that the ratio approaches 1/2 as the tree size increases, aligning with the theoretical upper bound for large red-black trees.
- Provides a mathematical formula to quantify the ratio for a given black-height, showing an understanding of the asymptotic behaviour.
- Acknowledges that the exact value of 1/2 is not achievable in a finite tree, which is a subtle but correct observation.

Negatives:

- Does not provide a specific tree structure or construction method to achieve the stated ratio, making it less concrete than A1.
- The formula $\frac{2^k-2}{2^{k+1}-1}$ is presented without derivation or a clear explanation of the tree configuration it corresponds to, reducing clarity.
- Lacks a detailed justification or example to connect the formula to red-black tree properties explicitly.

Summary:

- **A1** is incorrect because it underestimates the maximum ratio by fixing it at 2/5, despite evidence of higher ratios in larger trees.
- **A2** is correct in stating that the ratio approaches 1/2 asymptotically, though it could be improved with a specific construction and derivation.

(d) Correct Answer with Justification

Correct Answer: The largest possible ratio of red nodes to black nodes in a red-black tree approaches 1/2 as the tree's size increases indefinitely. For a finite tree, the ratio can be made arbitrarily close to 1/2 but never reaches it exactly. This is achieved by constructing a complete binary tree with alternating levels of black and red internal nodes, starting with a black root and ending with red nodes before the black NIL leaves.

Justification:

Red-Black Tree Properties

To find the maximum ratio, we must adhere to the following properties:

1. Every node is either red or black.
2. The root is black.
3. All NIL leaves are black (and counted as black nodes in this problem).
4. If a node is red, both its children are black (no two red nodes can be adjacent).
5. Every path from the root to a NIL leaf has the same number of black nodes (the black-height).

The goal is to maximize the number of red nodes relative to black nodes, subject to these constraints.

Construction of the Tree

Consider a complete binary tree where:

- Internal nodes are arranged in levels, alternating colours: level 0 (black), level 1 (red), level 2 (black), level 3 (red), and so on.
- The root (level 0) is black.
- The last internal level is red, with NIL leaves (black) as their children.
- Each internal node has exactly two children, ensuring a complete binary structure.

For a tree with $2m$ levels of internal nodes (indexed 0 to $2m - 1$):

- Black internal nodes are at even levels: 0, 2, 4, ..., $2m - 2$.
- Red internal nodes are at odd levels: 1, 3, 5, ..., $2m - 1$.
- NIL leaves are attached to the nodes at level $2m - 1$ (red), forming level $2m$.

Example:

- $m = 1$ (2 internal levels):
  - Level 0: 1 black (root).
  - Level 1: 2 red.
  - Level 2: 4 NILs (black).
  - Black nodes = 1 (internal) + 4 (NILs) = 5, Red nodes = 2, Ratio = 2/5 = 0.4.
- $m = 2$ (4 internal levels):
  - Level 0: 1 black.

- Level 1: 2 red.
- Level 2: 4 black.
- Level 3: 8 red.
- Level 4: 16 NILs (black).
- Black nodes = 1 + 4 (internal) + 16 (NILs) = 21, Red nodes = 2 + 8 = 10, Ratio = 10/21 ≈ 0.4762.

General Formula

For $m \geq 1$:

- **Number of black internal nodes**: Levels 0, 2, ..., $2m - 2$.
  - Level $2k$ has 2^{2k} nodes (level 0 has $2^0 = 1$, level 2 has $2^2 = 4$, etc.).
  - Total = $\sum_{k=0}^{m-1} 2^{2k} = 1 + 4 + 16 + \dots + 4^{m-1} = \frac{4^m-1}{4-1} = \frac{4^m-1}{3}$.
- **Number of red internal nodes**: Levels 1, 3, ..., $2m - 1$.
  - Level $2k + 1$ has 2^{2k+1} nodes (level 1 has $2^1 = 2$, level 3 has $2^3 = 8$, etc.).
  - Total = $\sum_{k=0}^{m-1} 2^{2k+1} = 2 + 8 + 32 + \dots + 2^{2m-1} = 2\sum_{k=0}^{m-1} 4^k = 2 \cdot \frac{4^m-1}{3}$.
- **Number of NILs**: The last level ($2m - 1$) has 2^{2m-1} red nodes, each with 2 NIL children.
  - Total NILs = $2 \cdot 2^{2m-1} = 2^{2m} = 4^m$.
- **Total black nodes**: $\frac{4^m-1}{3} + 4^m = \frac{4^m-1+3\cdot4^m}{3} = \frac{4\cdot4^m-1}{3} = \frac{4^{m+1}-1}{3}$.
- **Ratio**:
  - Ratio = $\frac{Red\ nodes}{Black\ nodes} = \frac{2\cdot\frac{4^m-1}{3}}{\frac{4^{m+1}-1}{3}} = \frac{2(4^m-1)}{4^{m+1}-1}$.

Asymptotic Behavior

As m increases:

- Let $a = 4^m$.
- Ratio = $\frac{2(a-1)}{4a-1}$.
- For large a, $\frac{2(a-1)}{4a-1} \approx \frac{2a}{4a} = \frac{1}{2}$.
- Examples:
  - $m = 1$: $a = 4$, $\frac{2(4-1)}{16-1} = \frac{6}{15} = \frac{2}{5} = 0.4$.
  - $m = 2$: $a = 16$, $\frac{2(16-1)}{64-1} = \frac{30}{63} = \frac{2}{5} = 0.4762$.
  - $m = 3$: $a = 64$, $\frac{2(64-1)}{256-1} = \frac{126}{255} = \frac{42}{85} = 0.4941$.

The ratio approaches 1/2 from below $m \to \infty$ but never equals 1/2 in a finite tree.

Verification of Red-Black Properties

- **Root is black**: Level 0 is black.
- **No adjacent red nodes**: Red levels follow black levels, and NILs (black) follow red nodes.
- **Equal black-height**: Count black nodes per path:
    - For m black internal levels (0, 2, ..., $2m - 1$), plus 1 NIL, black-height = $m + 1$.
    - E.g., $m = 2$: Path = black (0) - red (1) - black (2) - red (3) - NIL, has 3 black nodes (levels 0, 2, NIL).
- All paths have the same black height, satisfying the properties.

Comparison with A1 and A2

- **A1**: Claims 2/5 is the maximum, but our construction shows higher ratios (e.g., 10/21 > 2/5), disproving the assertion that larger trees reduce the ratio.
- **A2**: Correctly states the ratio approaches 1/2. Its formula $\frac{2^k-2}{2^{k+1}-1}$ may correspond to a different construction, but $k = m + 1$, it yields different values (e.g., $k = 2$: 2/7 ≠ 2/5). Our construction provides a clearer, verifiable ratio.

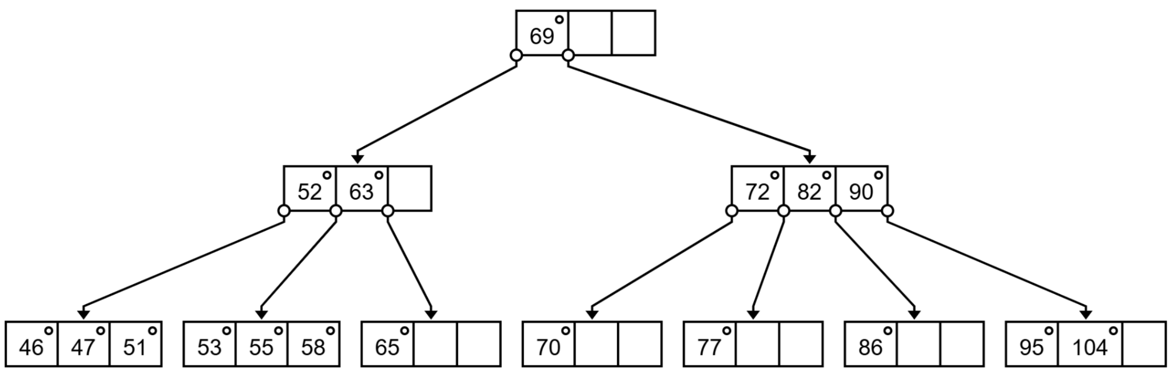
Conclusion: The largest possible ratio approaches 1/2 as \(m \) increases. For any finite tree, the ratio is less than 1/2 but can be made arbitrarily close to it by increasing the number of levels.

Question 2: B-Trees

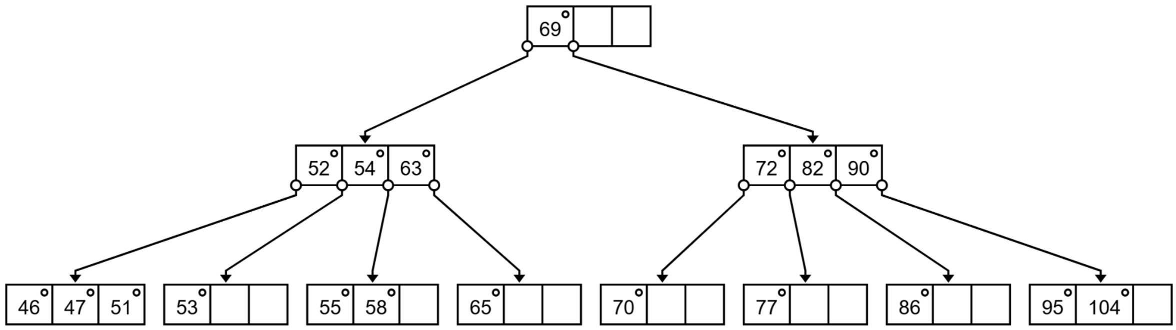
Given: A B-tree with minimum degree $t = 2$ (max 3 keys per node).

$u = 44$ **(last two digits of 200144X)**

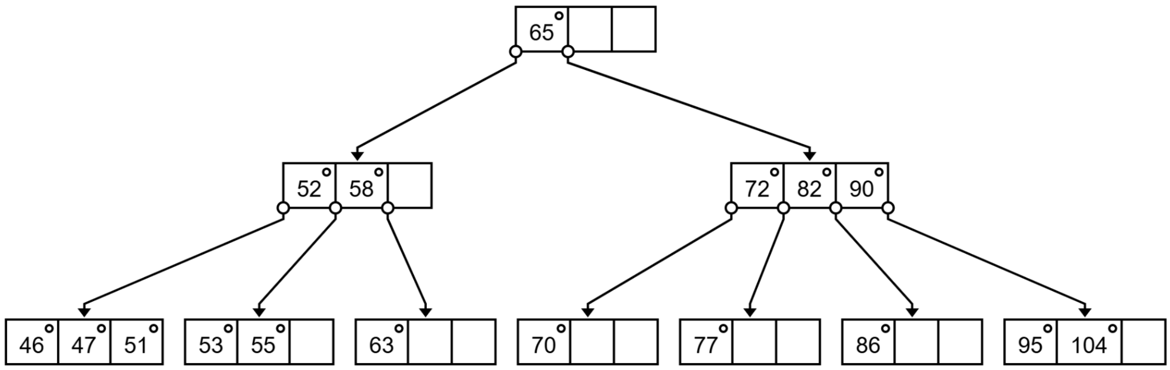
(a) Redraw with u Added



(b) Insert $10 + u = 54$



(c) Delete $25 + u = 69$



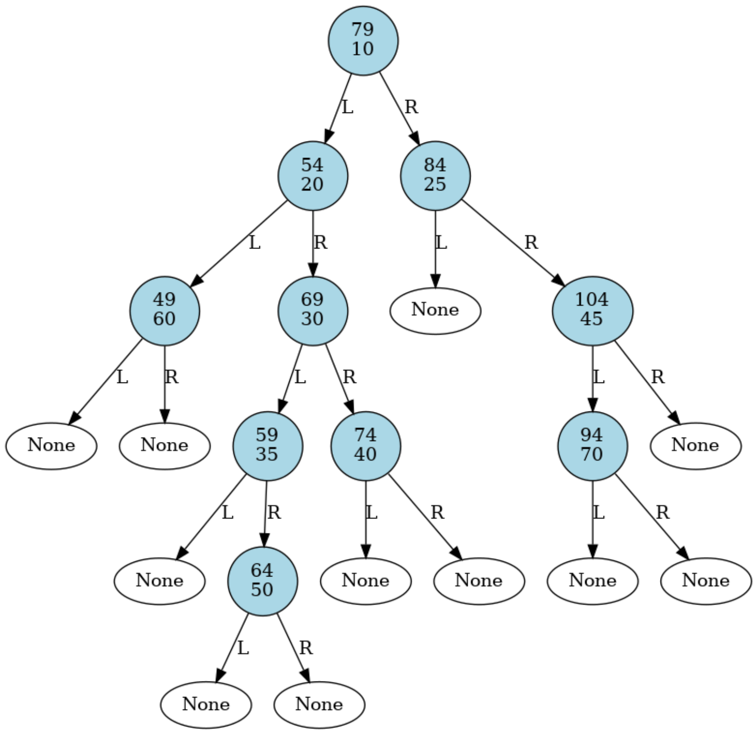
Question 3: Treaps

$u = 44$

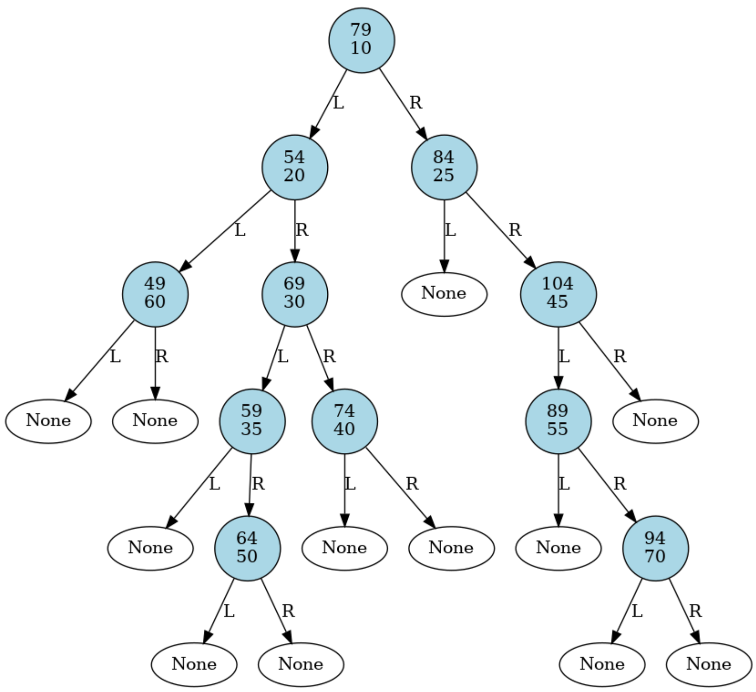
Keys: 59, 74, 54, 49, 64, 69, 79, 84, 94, 104

Priorities: 35, 40, 20, 60, 50, 30, 10, 25, 70, 45

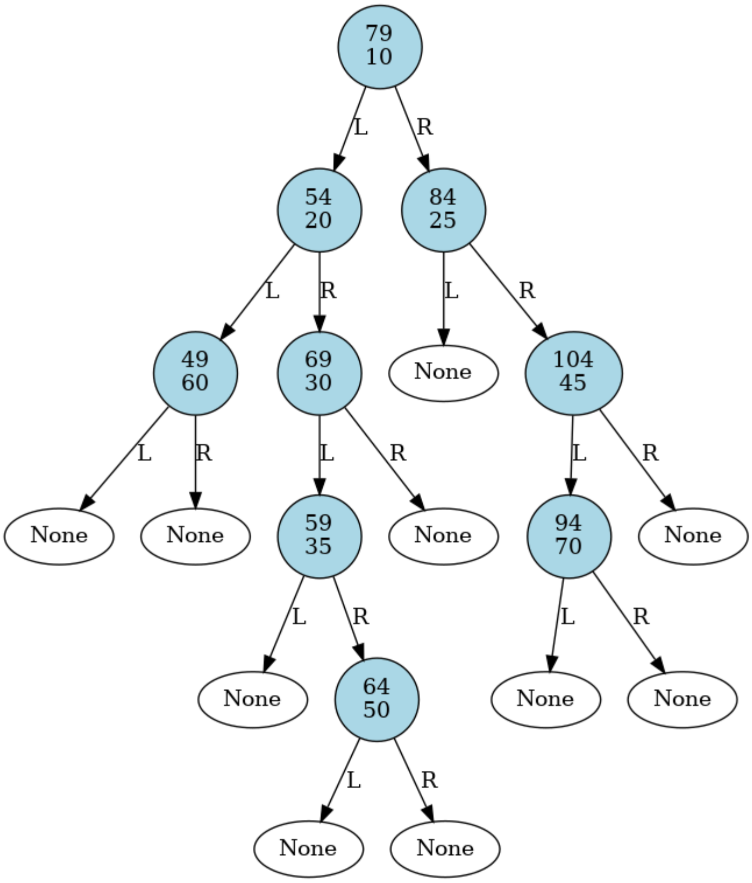
(a) Build Treap



(b) Insert (89, 55)



(c) Delete 74



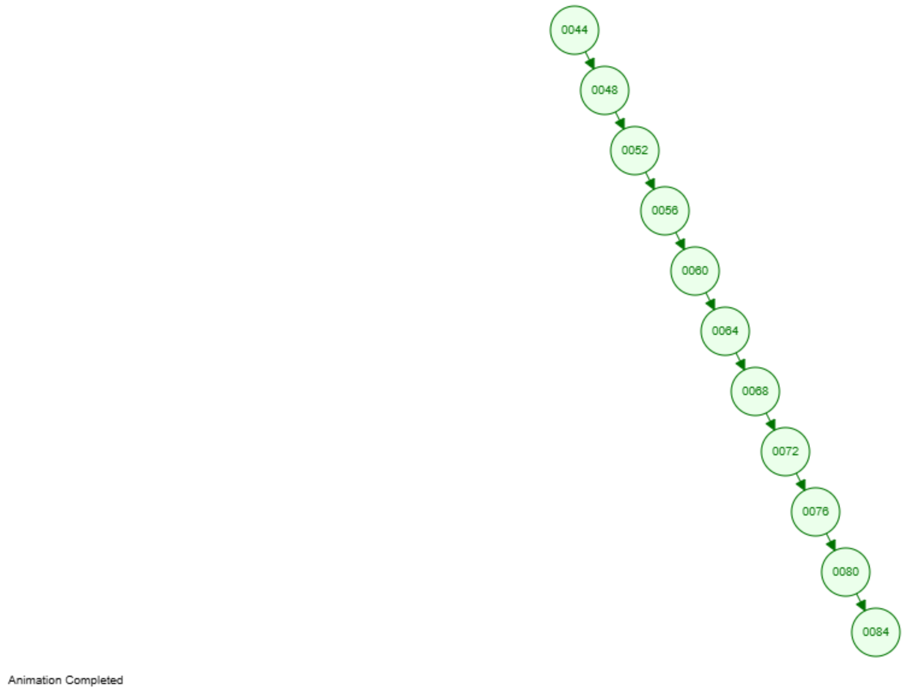
Question 4: BST Visualizations

Sequence: 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84

(a) BST, (b) AVL, (c) Red-Black, (d) Splay: Perform insertions, deletions (44, 64, 84), insertions (85, 63, 45), and for splay, searches (smallest, largest, 65). Use the website to generate screenshots.

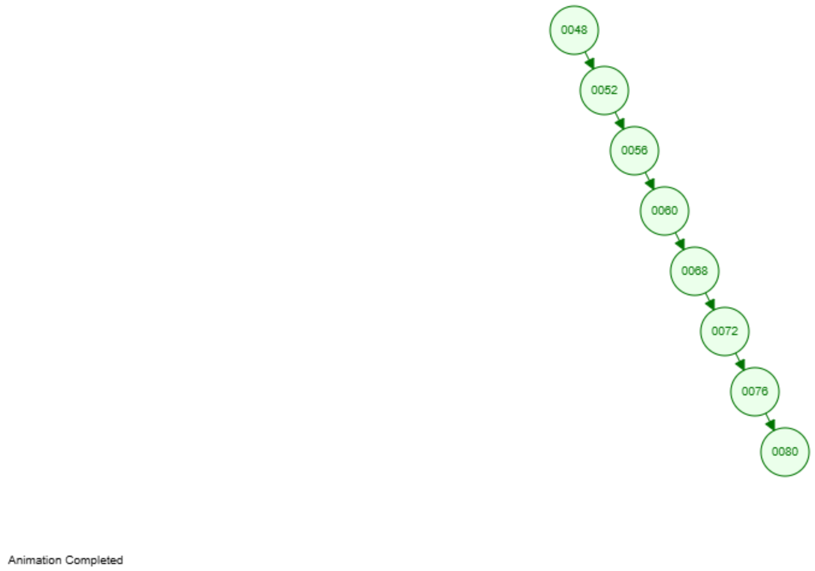
(a) BST

(i)



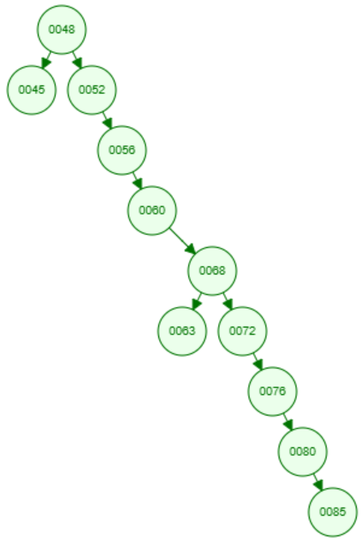
0044
0048
0052
0056
0060
0064
0068
0072
0076
0080
0084
Animation Completed

(ii)



0048
0052
0056
0060
0068
0072
0076
0080
Animation Completed

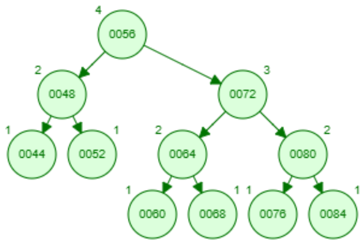
(iii)



Animation Completed

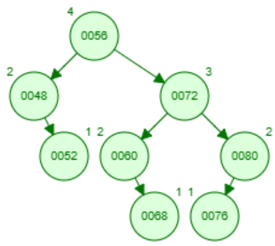
(b) AVL

(i)



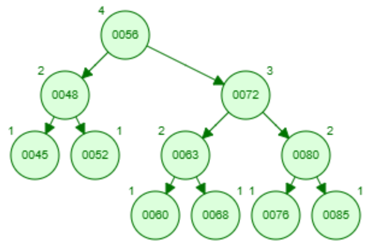
Animation Completed

(ii)



Animation Completed

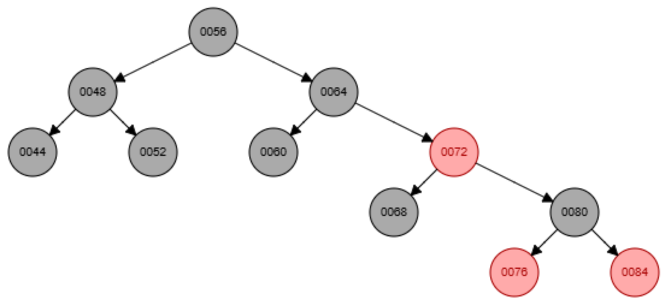
(iii)



Animation Completed

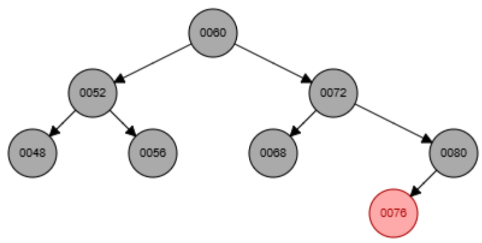
(c) Red-Black

(i)



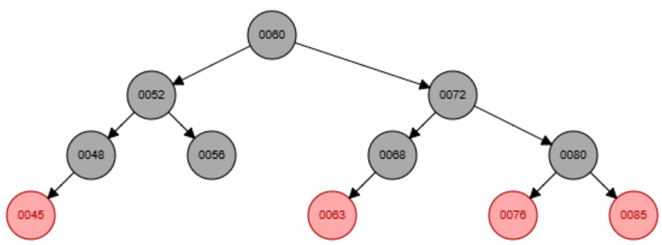
Animation Completed

(ii)



Animation Completed

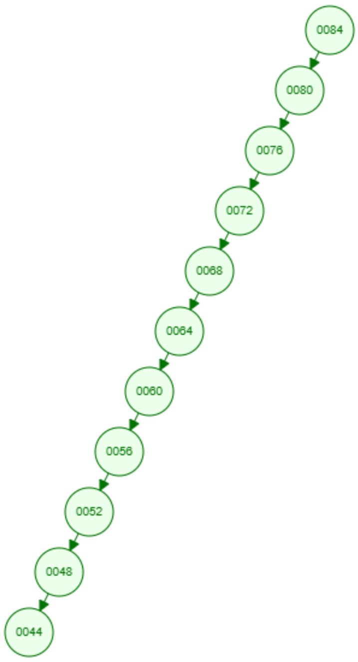
(iii)



Animation Completed

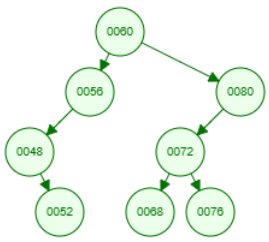
(d) Splay

(i)



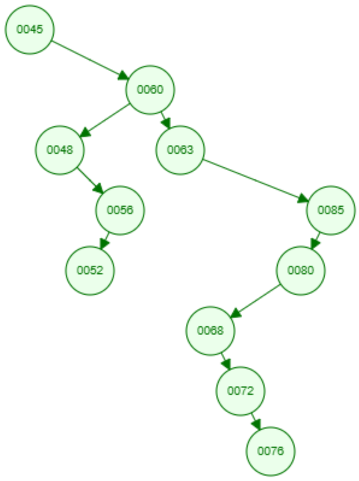
Animation Completed

(ii)



Animation Completed

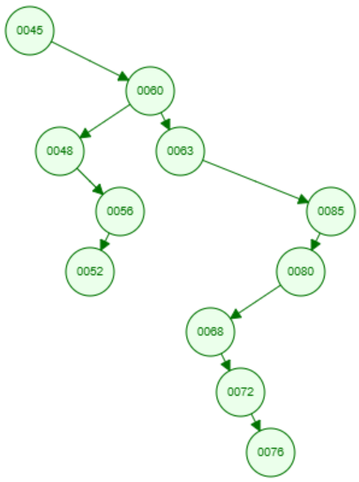
(iii)



Animation Completed

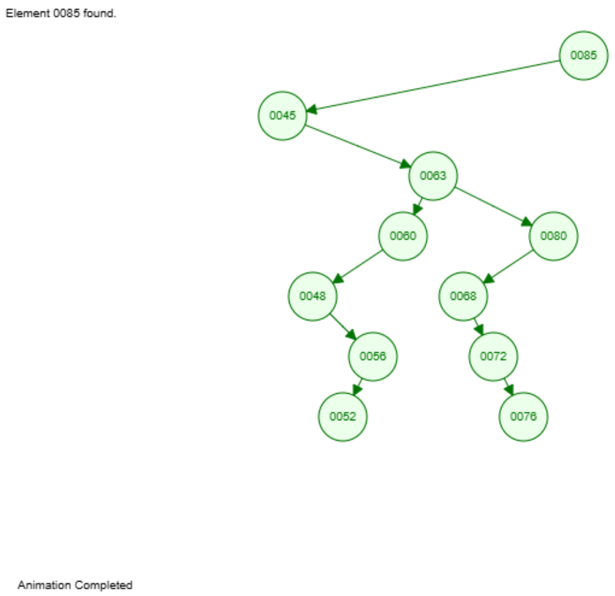
(iv)

Element 0045 found.

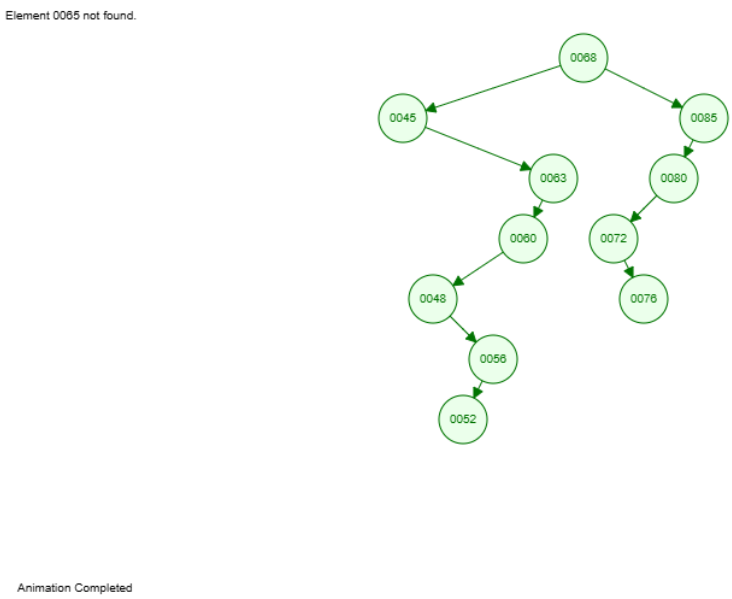


Animation Completed

(v)



(vi)



(e) Observations

Balanced trees were designed to overcome the worst-case performance issues of an unbalanced BST. In our experiments, the traditional BST's shape was entirely dictated by the insertion order, which sometimes resulted in a long, skewed tree. This can severely degrade performance (O(n) in the worst case) for search, insertion, and deletion operations.

In contrast, AVL trees enforce a strict balance by ensuring that the heights of the left and right subtrees differ by at most one. This guarantees a height of O(log n), leading to faster searches. However, maintaining this balance comes at a cost: AVL trees tend to perform more rotations during insertions and deletions, which can introduce overhead, especially in workloads with frequent updates.

Red-black trees adopt a more relaxed balancing approach by imposing a set of colour and structural properties that allow the tree to remain “almost balanced.” Although the worst-case height is slightly higher than that of AVL trees, red-black trees generally perform fewer rotations during updates. This balance between strictness and efficiency is one reason why they’re widely used in many standard libraries.

Splay trees offer a different philosophy altogether. Instead of enforcing a strict balance after every operation, splay trees “self-adjust” by moving recently accessed nodes to the root. This behaviour is particularly advantageous when there is a non-uniform access pattern, as frequently accessed items become quicker to reach over time. However, the downside is that individual operations might occasionally be more expensive, even though the amortized cost remains O(log n).

Regarding the visualization tool itself, it provides an intuitive, hands-on way to observe these dynamic operations. It clearly shows how the trees restructure themselves during insertions, deletions, and searches, which is invaluable for understanding the mechanics behind rotations and rebalancing. That said, the tool could be further improved by:

• Offering step-by-step animations with detailed annotations to indicate which rotation (e.g., left-right, right-left) is being performed.
• Providing side-by-side comparisons of different tree types for the same sequence of operations, which would help highlight the efficiency and structure differences more directly.
• Including performance metrics such as node heights, balance factors, and counts of rotations to give users quantitative insights into how balancing affects performance.

Overall, while unbalanced BSTs are simpler, they risk degenerating into inefficient structures. Balanced trees like AVL and red-black trees maintain optimal heights and ensure consistent performance, whereas splay trees adapt to access patterns, making them particularly useful in scenarios with locality of reference. The visualization tool serves as an excellent educational resource, even though incorporating more detailed analytics and user controls would enhance its instructional value further.

Question 5: BDDs (10 marks)

(a) Function at Node u

$f_u = \overline{x_i} \cdot f_{x_i=0} + x_i \cdot f_{x_i=1} = \overline{x_i} \cdot 1 + x_i \cdot (\overline{x_j} + x_k) = \overline{x_i} + x_i(\overline{x_j} + x_k)$

(b) ROBDD for $x_1 + x_2 + x_3 + x_4$

Starting with the root node for x_1:

- If $x_1 = 1$, then $f = 1 + x_2 + x_3 + x_4 = 1$, so the 1-edge points to the 1-node.

- If $x_1 = 0$, then $f = 0 + x_2 + x_3 + x_4 = x_2 + x_3 + x_4$, so the 0-edge points to a node representing $x_2 + x_3 + x_4$.

Next, for the node representing $x_2 + x_3 + x_4$ (label it with x_2):

- If $x_2 = 1$, then $f = 1 + x_3 + x_4 = 1$, so the 1-edge points to the 1-node.
- If $x_2 = 0$, then $f = 0 + x_3 + x_4 = x_3 + x_4$, so the 0-edge points to a node representing $x_3 + x_4$.

For the node representing $x_3 + x_4$ (label it with x_3):

- If $x_3 = 1$, then $f = 1 + x_4 = 1$, so the 1-edge points to the 1-node.
- If $x_3 = 0$, then $f = 0 + x_4 = x_4$, so the 0-edge points to a node representing x_4.

For the node representing x_4 (label it with x_4):

- If $x_4 = 1$, then $f = 1$, so the 1-edge points to the 1-node.
- If $x_4 = 0$, then $f = 0$, so the 0-edge points to the 0-node.

Now, describe the structure:

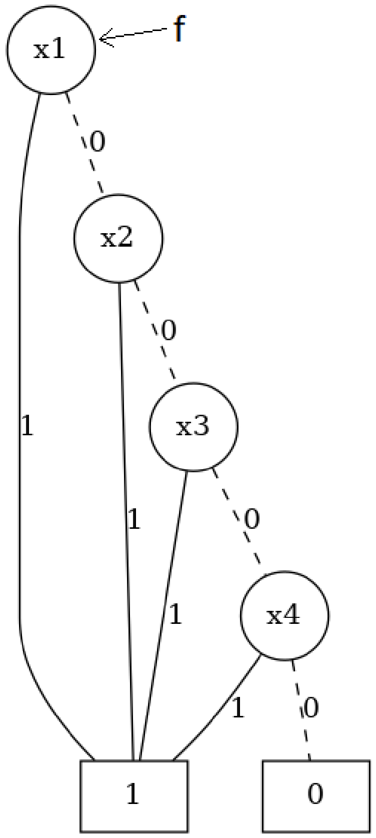
- **Node u_1 (variable x_1)**: High (1-edge) to 1-node, Low (0-edge) to node u_2.
- **Node u_2 (variable x_2)**: High to 1-node, Low to node u_3.
- **Node u_3 (variable x_3)**: High to 1-node, Low to node u_4.
- **Node u_4 (variable x_4)**: High to 1-node, Low to 0-node.
- **Leaf nodes**: 0 and 1.

Reduction Check:

- **No redundant nodes**: A node is redundant if its 0-edge and 1-edge point to the same child. Here, each node has distinct children (e.g., u_1 has High to 1 and Low to u_2, which are different).
- **No isomorphic subgraphs**: Each subtree is unique due to different variables and children.

Thus, this is the ROBDD.

- $x_1 \rightarrow$ (0: x_2, 1: 1)
- $x_2 \rightarrow$ (0: x_3, 1: 1)
- $x_3 \rightarrow$ (0: x_4, 1: 1)
- $x_4 \rightarrow$ (0: 0, 1: 1)



(c) $g = \overline{x_2 + x_3 + x_4}$ in the Same BDD Structure with Complement Edges

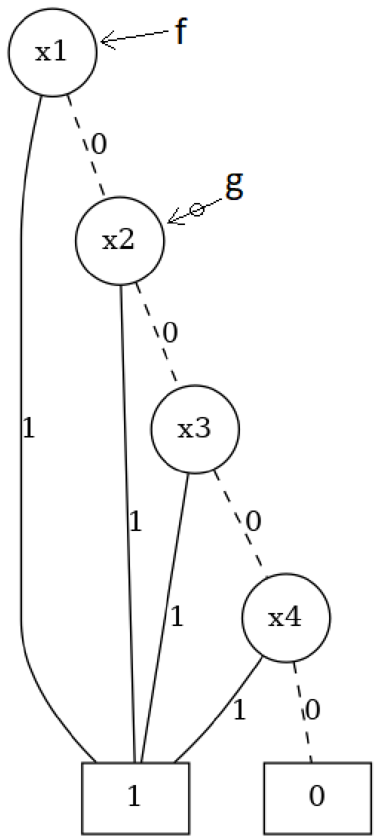
The above structure represents $f = x_1 + x_2 + x_3 + x_4$. Specifically, the subdiagram rooted at the node u_2 represents the function $x_2 + x_3 + x_4$.

Now, we need to represent $g = \overline{x_2 + x_3 + x_4}$ within this same BDD structure, utilizing complement edges. A complement edge in a BDD indicates the negation of the function represented by the node it points to. Since the node u_2 already represents $x_2 + x_3 + x_4$, the negation of this function, $\overline{x_2 + x_3 + x_4}$, is precise g.

Thus, we can represent g without constructing a separate BDD or adding new nodes. Instead, we use the existing structure and define g as follows:

The function g is represented by a pointer to the node u_2 with a complement edge.

This means that $g = \overline{u_2}$ where u_2 is the node for x_2 in the BDD for f, and the complement edge negates the function $x_2 + x_3 + x_4$ to yield $\overline{x_2 + x_3 + x_4}$.



Question 6: Java Code Timing

In this question, we analyze the performance of four binary search tree implementations—Basic BST, Splay Tree, Red-Black (RB) Tree, and AVL Tree—by measuring the time taken for insertion, search, and deletion operations using the provided data sets. The modified Java source files (`AVLTree.java`, `BST.java`, `RedBlackBST.java`, and `SplayBST.java`) include `main()` methods that read data from files in the `data.zip` folder, perform the specified operations, and record the execution times in microseconds. The results are reported below, followed by detailed discussions.

(a) Time Taken to Insert (in Microseconds)

The table below summarizes the time taken to insert data from the files in the `insert` folder into each tree type for two data sets, `set1` and `set2`, each containing three files: `data_1.txt`, `data_2.txt`, and `data_3.txt`.

Data Set	Time taken to **Insert** (in microseconds)

		Basic BST	Splay Tree	RB-Tree	AVL Tree
set1	data_1	39,622	38,426	45,710	29,751
	data_2	61,591	105,078	85,587	62,178
	data_3	153,899	259,208	198,367	169,512
set2	data_1	1,085,094 (SO)	2,188	10,801	18,639
	data_2	1,002,485 (SO)	4,267	42,470	19,534
	data_3	976,728 (SO)	6,137	53,470	23,978

Note: "SO" indicates a StackOverflowError occurred during the operation.

Observations and Discussion

- **Basic BST:**
  - For `set1`, insertion times increase with the size of the data file (e.g., 39,622 μs for `data_1` to 153,899 μs for `data_3`), reflecting the expected behaviour where tree height can grow linearly in the worst case (O(n) time complexity) if the data is unbalanced.
  - For `set2`, insertion results in StackOverflowErrors across all files, with times exceeding 1 million microseconds before crashing. This suggests that `set2` contains sorted or nearly sorted data, causing the tree to degenerate into a linked list-like structure, leading to excessive recursion depth.
- **Splay Tree:**
  - In `set1`, insertion times are relatively high (e.g., 259,208 μs for `data_3`), likely due to the splaying operation performed on every insertion, which reorganizes the tree to bring the newly inserted node to the root. This incurs additional overhead (amortized O(log n) per operation).

    - In set2, insertion times are remarkably low (e.g., 2,188 µs for data_1), indicating that the splaying mechanism adapts exceptionally well to the data distribution in set2, possibly benefiting from the locality of access or a pattern that reduces tree height effectively.
- **RB-Tree and AVL Tree:**
    - Both balanced trees show consistent performance across set1 and set2. For set1, AVL Tree is slightly faster (e.g., 29,751 µs vs. 45,710 µs for data_1), while RB-Tree performs better in set2 for smaller files (e.g., 10,801 µs vs. 18,639 µs for data_1).
    - Their self-balancing mechanisms (rotations in AVL, colour flips and rotations in RB) ensure logarithmic time complexity (O(log n)), preventing the extreme degradation seen in Basic BST.

Key Points

- **Balanced Trees:** RB-Tree and AVL Tree maintain stable insertion times across both data sets, making them reliable for general-purpose use, especially with potentially skewed data.
- **Splay Tree Variability:** The Splay Tree's performance varies significantly, with high costs in set1 but exceptional efficiency in set2, suggesting it thrives in scenarios with specific access patterns.
- **Basic BST Limitations:** The Basic BST fails catastrophically in set2 due to its inability to self-balance, highlighting its unsuitability for sorted or near-sorted inputs.

(b) Time Taken to Search (in Microseconds)

The table below presents the time taken to perform search operations using data from the search folder on the trees constructed in part (a).

Data Set		Time taken to **Insert** (in microseconds)			
		Basic BST	**Splay Tree**	**RB-Tree**	**AVL Tree**
set1	data_1	78	38	77	92
	data_2	46	9	18	43

	data_3	59	35	50	42
set2	data_1	1,478 (SO)	5	12	118
	data_2	1,328 (SO)	6	8	17
	data_3	1,480 (SO)	15	12	4

Observations and Discussion

- **Basic BST:**
  - Search times in `set1` are low (e.g., 78 μs for `data_1`), indicating a reasonably balanced tree for this data set.
  - In `set2`, search operations result in StackOverflowErrors, with times around 1,300–1,480 μs, reflecting the same imbalance issue seen in insertion, where the tree height becomes O(n).
- **Splay Tree:**
  - Search times are exceptionally low across both sets, especially in `set2` (e.g., 5 μs for `data_1`). The splaying operation, which brings accessed nodes to the root, optimizes subsequent searches, particularly when certain elements are frequently accessed.
  - In `set1`, times are slightly higher but still competitive (e.g., 9 μs for `data_2`), showcasing its adaptive nature.
- **RB-Tree and AVL Tree:**
  - Both exhibit consistent search times across sets, with RB-Tree slightly faster in `set2` (e.g., 8 μs vs. 17 μs for `data_2`). AVL Tree shows a minor increase for `data_1` in `set2` (118 μs), but overall performance remains stable (O(log n)).
  - Their balanced structures ensure efficient search operations regardless of data distribution.

Explanation of Behavior

- **Splay Tree Advantage:** The Splay Tree's superior search performance stems from its self-adjusting nature, making it ideal for workloads with repeated accesses, as seen in `set2`.
- **Balanced Trees:** RB-Tree and AVL Tree provide reliable logarithmic search times, unaffected by the data skew that cripples Basic BST in `set2`.
- **Basic BST Failure:** The Basic BST's poor performance in `set2` confirms its vulnerability to unbalanced structures, leading to linear-time searches.

(c) Time Taken to Delete (in Microseconds)

The table below shows the time taken to perform deletion operations using data from the `delete` folder on the trees constructed in part (a).

Data Set		Time taken to **Insert** (in microseconds)			
		Basic BST	**Splay Tree**	**RB-Tree**	**AVL Tree**
set1	data_1	110	48	163	100
	data_2	142	10	104	46
	data_3	55	30	122	52
set2	data_1	1,095 (SO)	5	82	67
	data_2	1,004 (SO)	4	38	13
	data_3	1,802 (SO)	6	44	5

Observations and Discussion

- **Basic BST:**
  - Deletion times in `set1` are moderate (e.g., 55 μs for `data_3`), but in `set2`, they spike dramatically with StackOverflowErrors (e.g., 1,802 μs for `data_3`), consistent with its imbalance issues.
- **Splay Tree:**
  - Deletion times are extremely low across both sets (e.g., 4 μs for `data_2` in `set2`), reflecting efficient tree reorganization via splaying, which minimizes depth for subsequent operations.

- **RB-Tree and AVL Tree:**
    - Both maintain consistent deletion times, with AVL Tree outperforming RB-Tree in `set2` for larger files (e.g., 5 μs vs. 44 μs for `data_3`). In `set1`, RB-Tree has slightly higher times (e.g., 163 μs for `data_1`).

Explanation of Behavior

- **Splay Tree Efficiency:** The Splay Tree's low deletion times result from its ability to adapt the tree structure dynamically, making it highly effective for specific access patterns.
- **Balanced Trees:** RB-Tree and AVL Tree ensure logarithmic deletion times through their balancing mechanisms, with AVL Tree showing a slight edge in some cases due to stricter height balance.
- **Basic BST Instability:** The Basic BST's performance collapses in `set2`, reinforcing its unsuitability for unbalanced data.

Overall Discussion

- **Basic BST:** Its performance degrades severely in `set2` due to lack of balance, leading to StackOverflowErrors and high operation times. It is unsuitable for sorted or nearly sorted data.
- **Splay Tree:** Exhibits variable performance—high insertion times in `set1` but exceptional search and deletion times, especially in `set2`. It excels in scenarios with frequent access to specific elements.
- **RB-Tree and AVL Tree:** Both offer consistent logarithmic performance across all operations and data sets, making them reliable for general use. AVL Tree has a slight advantage in insertion and deletion, while RB-Tree edges out in search for some cases.

Conclusion

For general-purpose applications with unpredictable data distributions, RB-Tree and AVL Tree are the most reliable choices due to their consistent performance. Splay Tree is advantageous in specific scenarios with frequent access patterns, while Basic BST should be avoided when data balance cannot be guaranteed.

Appendix Q2

Tree Description for (a)

69
52, 63 / 72, 82, 90
46, 47, 51 / 53, 55, 58 / 65 / 70 / 77 / 86 / 95, 104

Tree Description for (b)

63
52, 55 / 72, 82, 90
46, 47, 51 / 53, 54 / 58, 65 / 70 / 77 / 86 / 95, 104

Tree Description for (c)

65
52, 58 / 72, 82, 90
46, 47, 51 / 53, 55 / 63 / 70 / 77 / 86 / 95, 104

References

[1] B-Sketcher - Project
[2] CS4523 - Advanced Algorithms - Assignment 1 - GitHub Repository