Activity 5 - Discussion Points

1. What is an AVL tree? How is it different from a BST?

An AVL tree is a self-balancing binary search tree where the height difference between left and right subtrees is at most 1. Unlike a regular BST, it automatically performs rotations to stay balanced, ensuring faster operations.

2. Why is it important to keep the tree height balanced?

Keeping the tree height balanced ensures that search, insertion, and deletion operations run in (O(log n)) time. Without balancing, a tree can become skewed and degrade to (O(n)) performance.

3. What is a B-Tree? How is it different from an AVL tree?

A B-Tree is a self-balancing tree designed to store multiple keys per node and is commonly used in databases and file systems. Unlike AVL trees, B-Trees minimize disk reads by being shallow and wide rather than tall and binary.

4. What is the fundamental problem that balanced trees aim to solve?

Balanced trees aim to prevent performance degradation caused by unbalanced or skewed trees. They ensure predictable and efficient data access times regardless of insertion order.

5. Compare and contrast different types of balanced trees (AVL, red-black, splay, B-tree).

AVL trees are strictly balanced and provide fast lookups but require frequent rotations. Red-black trees allow looser balancing for faster updates, splay trees self-adjust based on access patterns, and B-Trees are optimized for disk-based storage systems.

6. Discuss the challenges of implementing balanced trees and the importance of ensuring correctness during rotations.

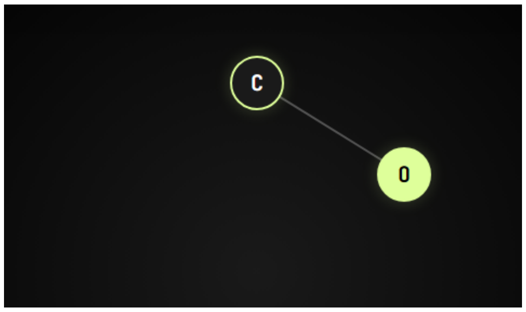
Implementing balanced trees is complex due to the need to handle many edge cases during rotations. Incorrect rotations can violate tree properties, leading to incorrect searches or corrupted structures.

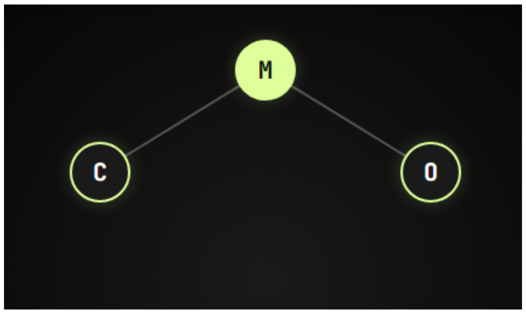
7. Brainstorm real-world scenarios where balanced trees would be the most suitable data structure choice.

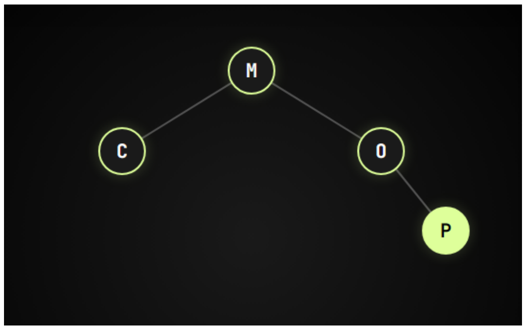
Balanced trees are ideal for database indexing, file systems, and memory management systems. They are also useful in applications requiring fast searching and frequent updates, such as compilers and operating systems.

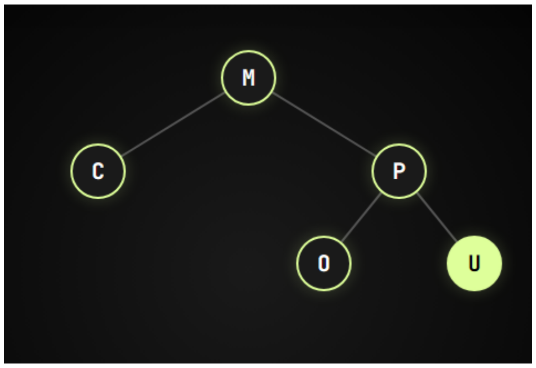
Activity 6 – Questions

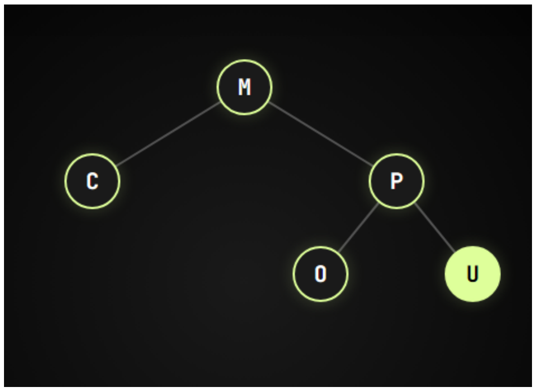
1. **COMPUTING (LEFT TO RIGHT SEQUENCE)**

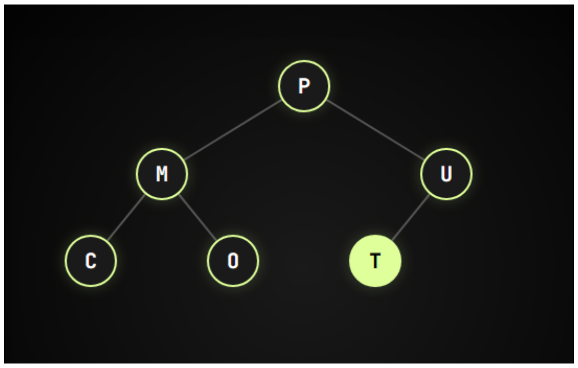


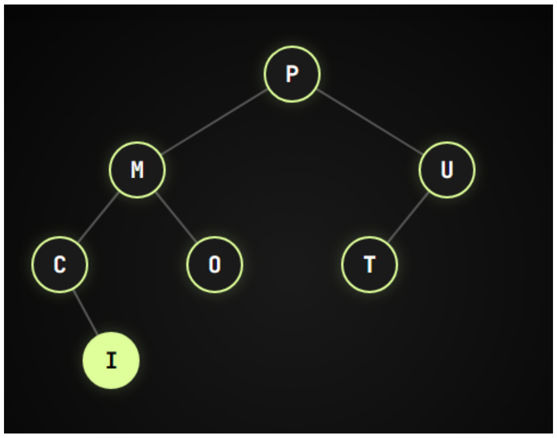


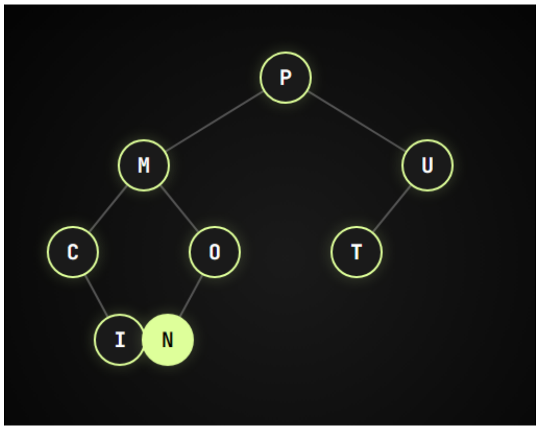




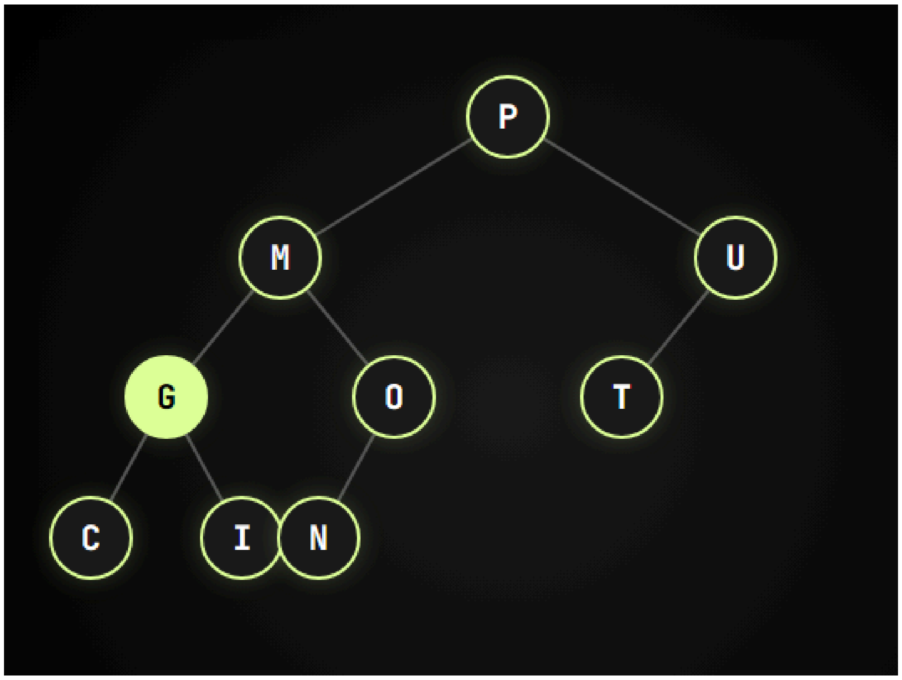




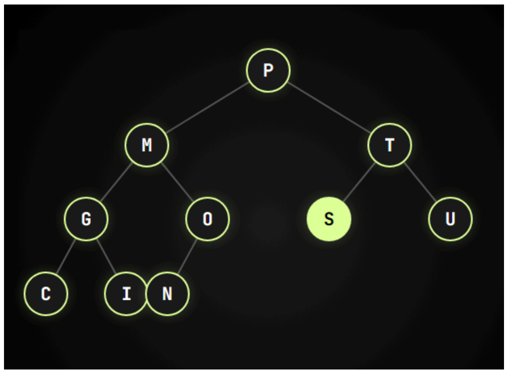


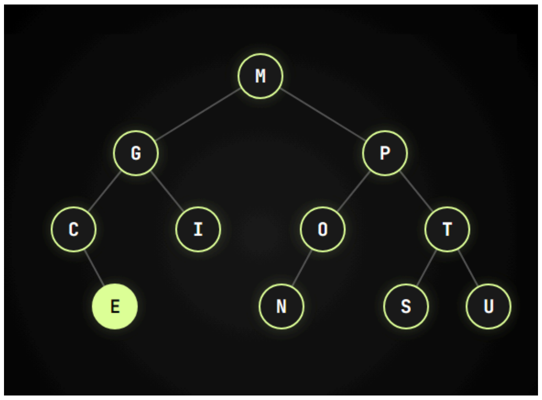


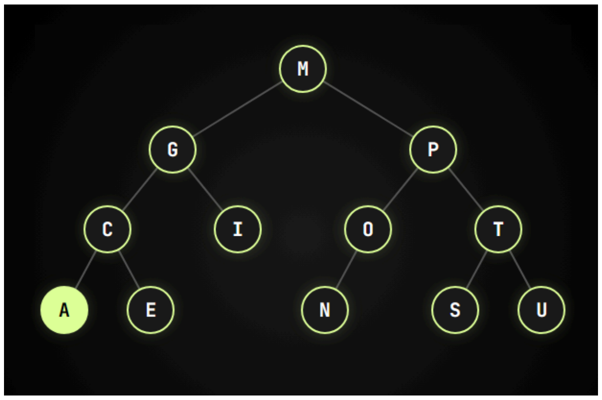
COMPUTING AVL TREE



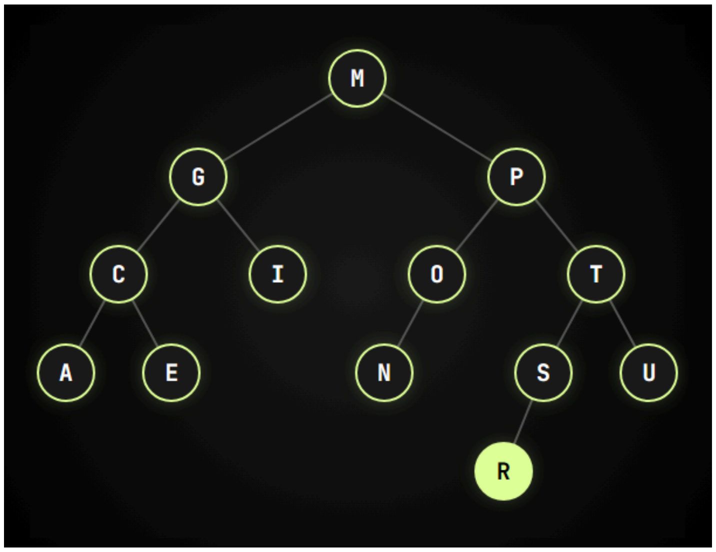
2. ADD S E A R



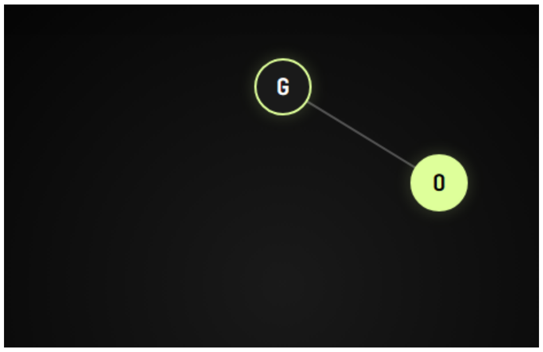


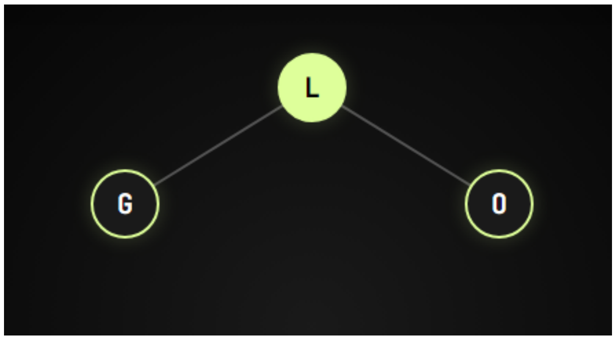


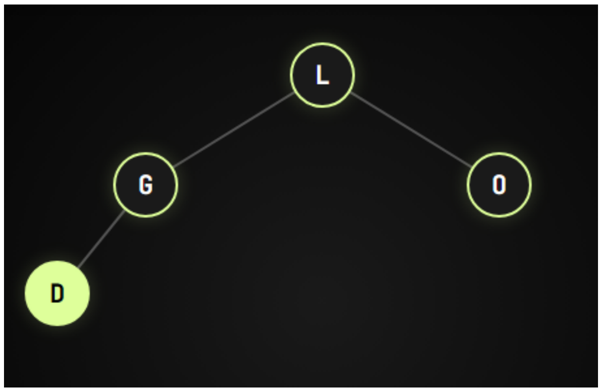
FINAL TREE

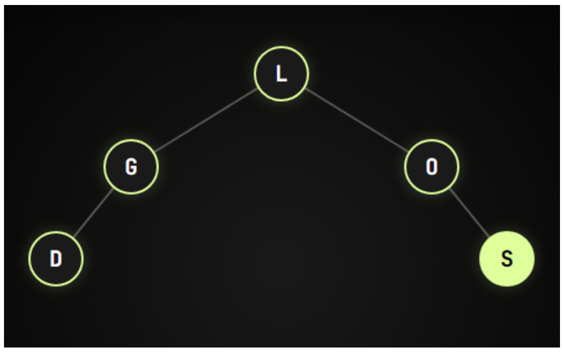


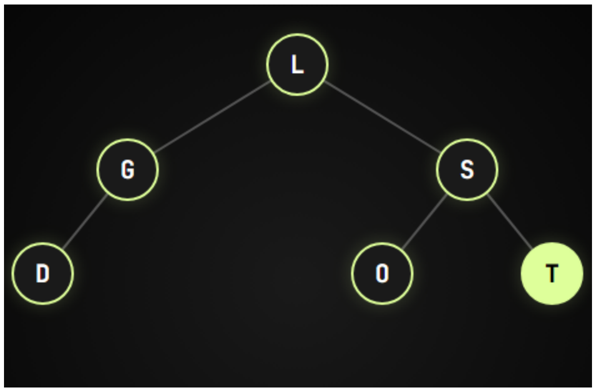
3. **GOLDSTAR**

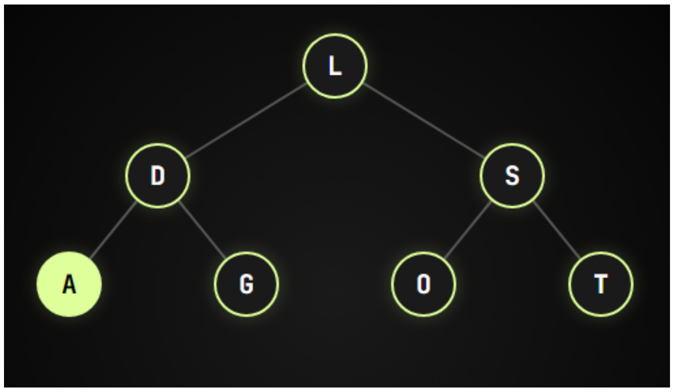




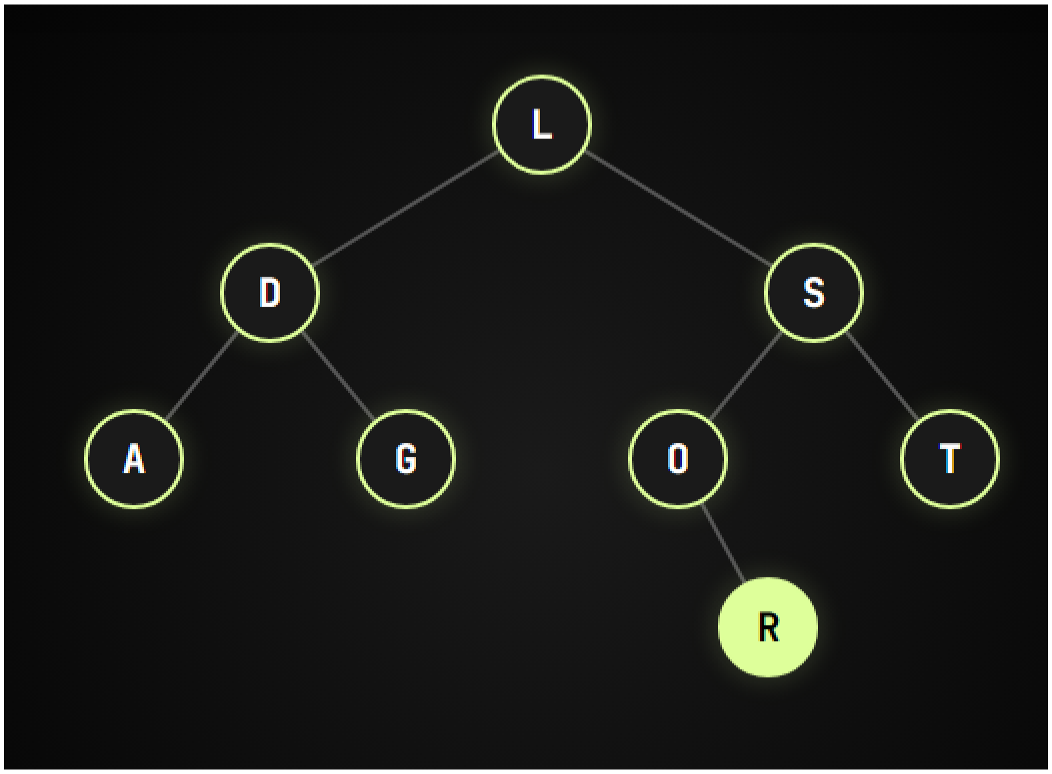




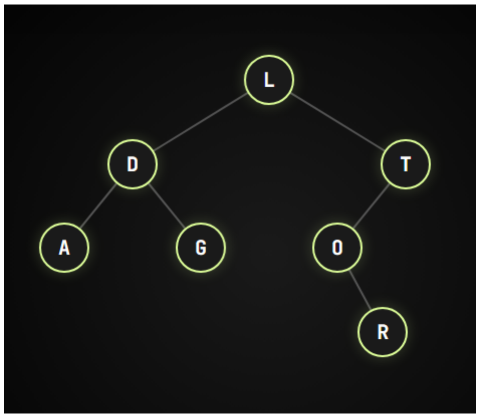


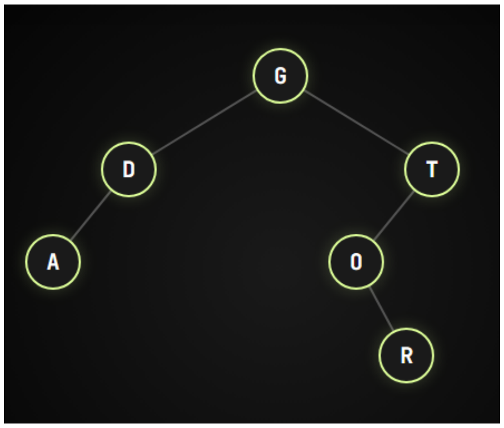


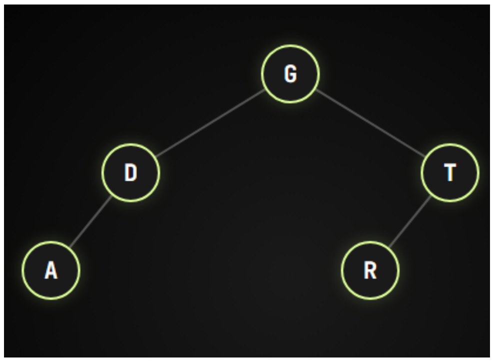
FINAL TREE

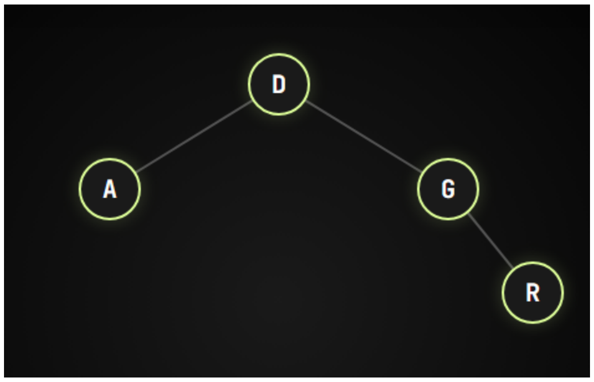


4. DELETE SLOT

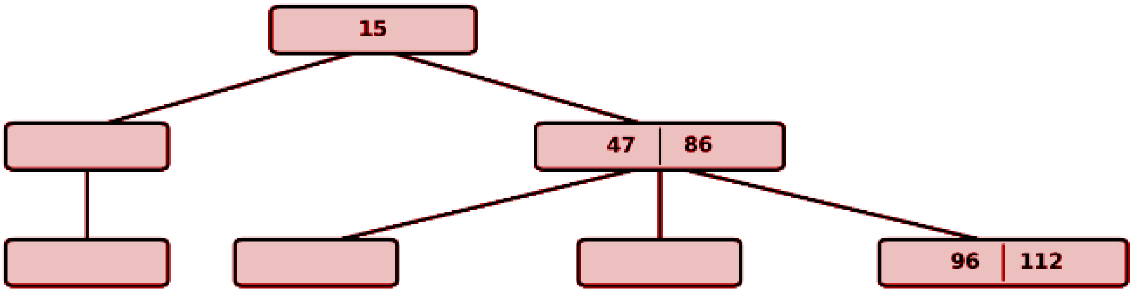




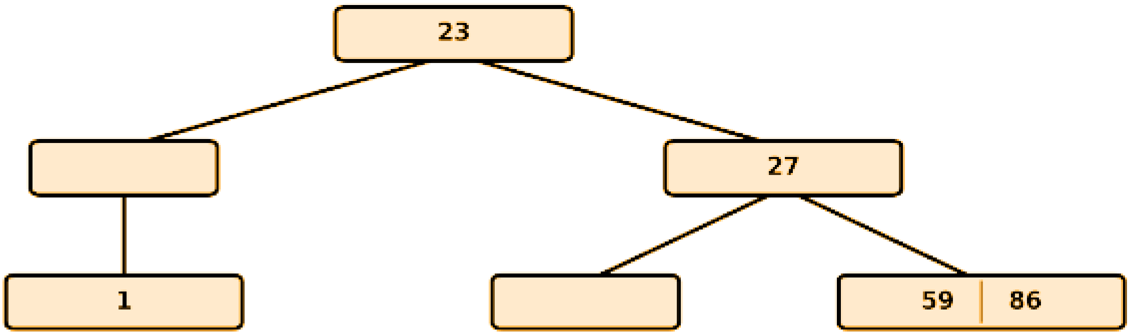




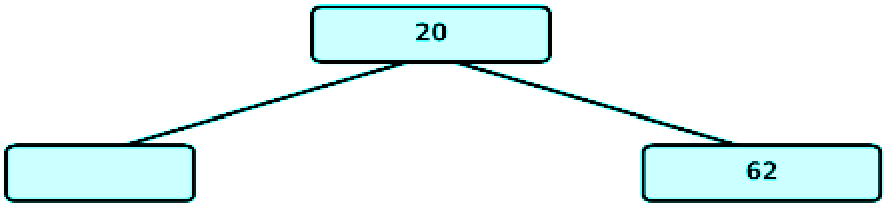
5a



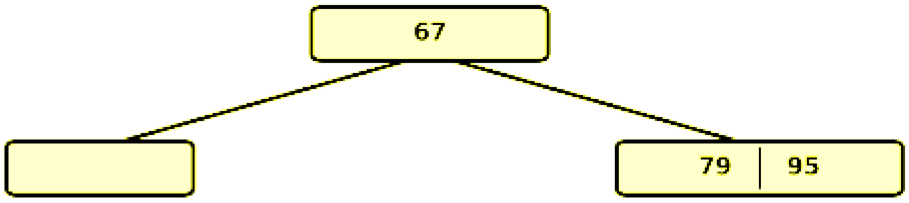
5b



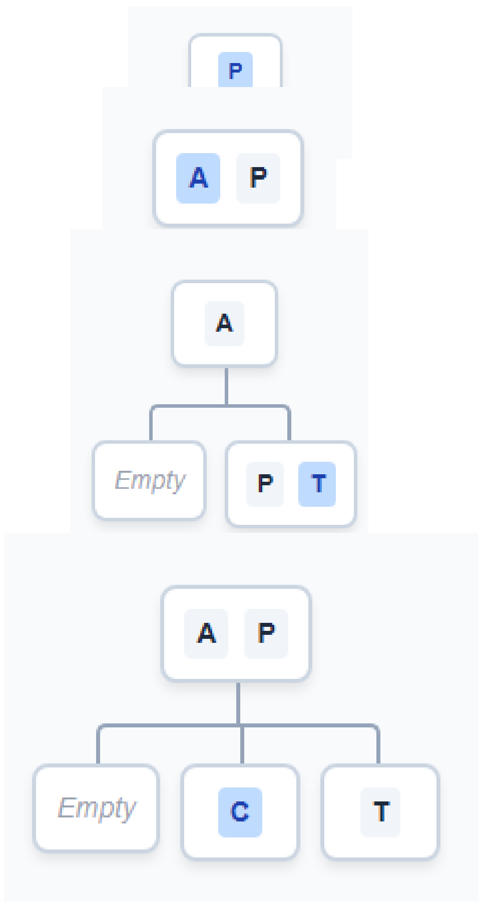
5c

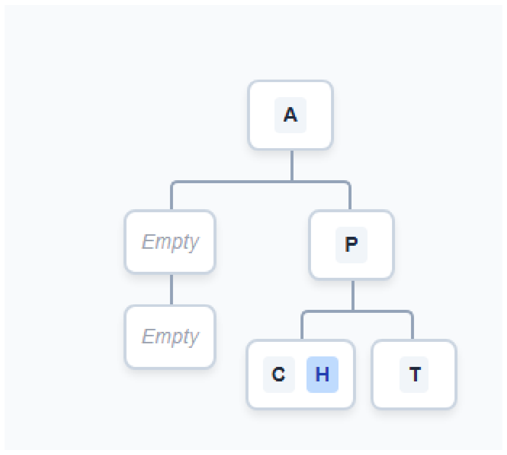


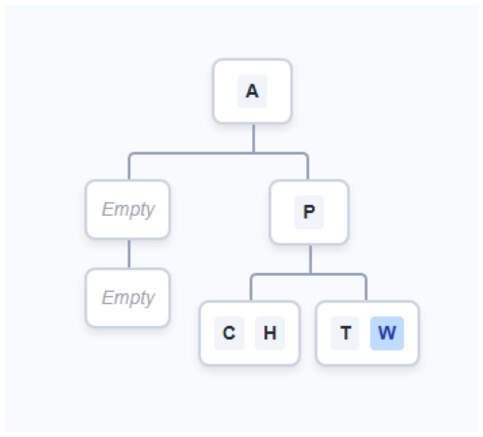
5d

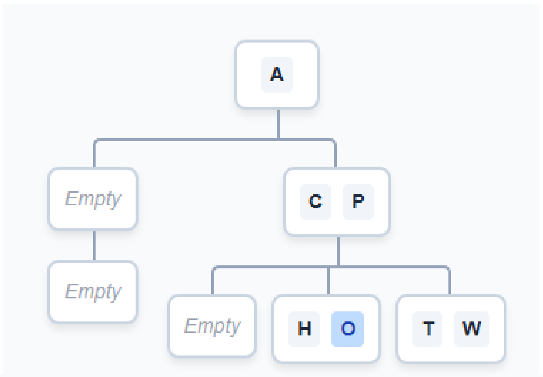


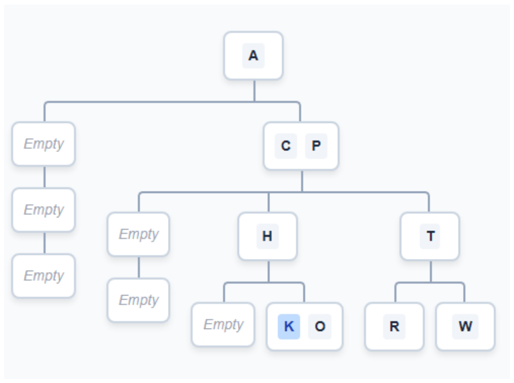
6. PATCHWORKS

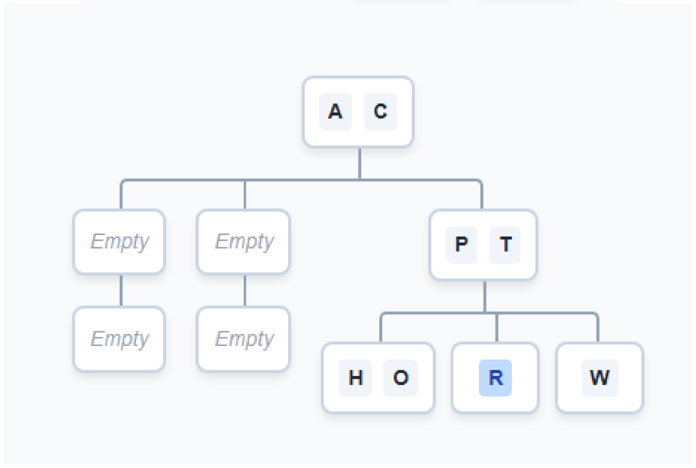




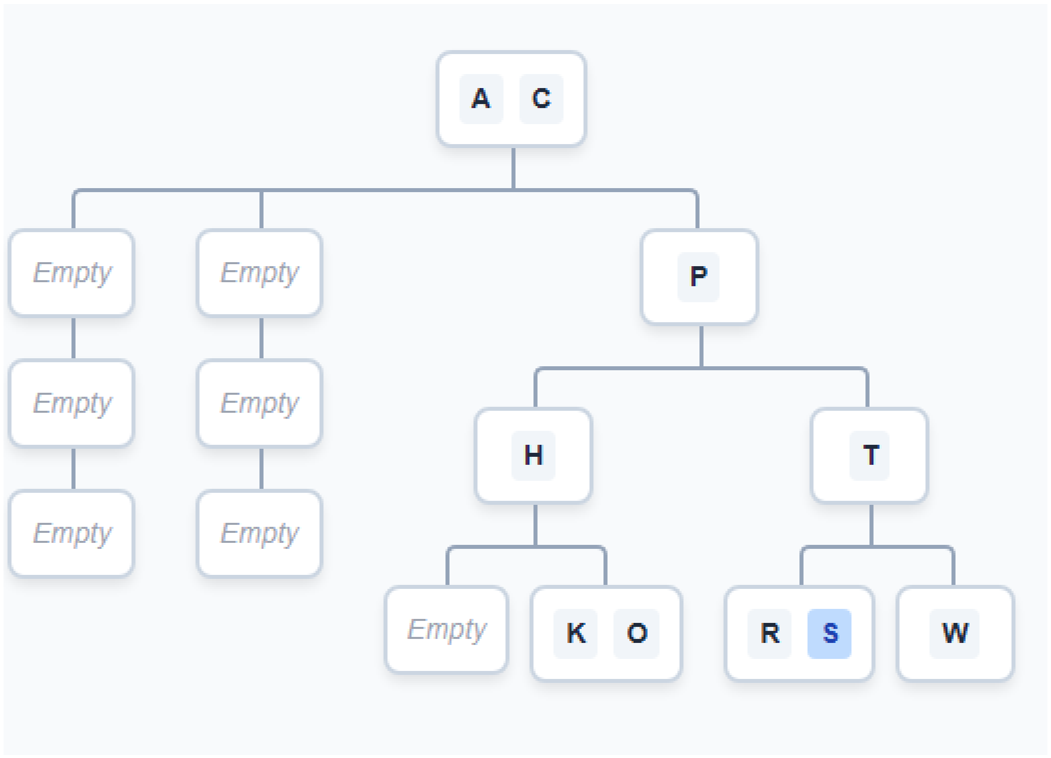



A
Empty
Empty
C P
Empty
H O
T W


A
Empty
Empty
Empty
C P
Empty
Empty
H
T
Empty
K O
R
W


A C
Empty
Empty
Empty
Empty
P T
H O
R
W

FINAL B-TREE


A C
Empty
Empty
Empty
Empty
Empty
Empty
P
H
T
Empty
K O
R S
W

JAVIER, SALVADOR VINCENT R.
BAES, FRANZ EMMANUEL F.
BSIT 2 - 1

Data Structures and Algorithm

Repository: github.com/slvdrvncntjvr/DataStrucProg

Activity 7 - Experiments

1. Compare the performance of a balanced tree (AVL or Red-Black tree) with an unbalanced binary search tree (BST) for operations like insertion, deletion, and searching on various datasets (random, sorted, reverse sorted). Analyze the time complexity and observe how balanced trees offer more consistent performance.

   › **Performance Comparison**

2. Implement either an AVL tree or a Red-Black tree in your preferred programming language. This involves implementing insertion, deletion, searching, and rotation operations. Emphasize the importance of maintaining balance throughout these operations.

   › **AVL Tree Implementation**

3. Investigate the use of B-trees and B+ trees in database indexing. Understand their structure, properties, and how they address the challenges of disk-based storage compared to in-memory trees.

   › B-trees are self-balancing m-ary search trees that maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time. They permit a high branching factor (many children per node) so that each tree level corresponds to one disk page, drastically reducing the number of I/O operations required for large datasets (GeeksforGeeks, 2023). All leaves in a B-tree lie at the same depth, and nodes are kept partially full (at least half full) to balance the tree; this ensures operations run in O(log n) time even as the tree grows. B+ trees are an extension used in databases: they store all records in the leaf level and keep internal nodes as key guides only. Leaf nodes are linked sequentially, allowing efficient range and ordered scans (GeeksforGeeks, 2023). This structure is ideally suited for disk storage: by packing many keys in nodes sized to disk blocks, these trees minimize costly seeks and exploit spatial locality, making database indexing fast and scalable (PlanetScale, 2023; Wikipedia, 2023).

4. Identify and analyze the use of balanced trees in various modern technologies like databases, search engines, file systems, and programming language compilers. Explore how their self-balancing property contributes to efficient data management and retrieval.

› Balanced search trees underpin modern data systems. All major database management systems use B-tree or B+ tree indexes to locate records quickly by key. Many file systems use B+ tree variants to index directories and file metadata (for example, NTFS, ext4, XFS, and APFS all rely on such trees). In memory, self-balancing binary trees are widespread: language libraries' ordered maps and sets commonly use red-black trees, and even operating system components (e.g., schedulers) use balanced tree structures (GeeksforGeeks, 2023). Search engines similarly employ sorted index structures (such as B-trees or inverted indexes) to quickly match queries, and compilers use balanced trees (e.g., in symbol tables) to keep identifier lookup efficient. The critical advantage of any self-balancing tree is that automatic rotations or splits keep its height logarithmic, so lookups, insertions, and deletions consistently run in $O(\log n)$ time; this guarantees fast, scalable data management and retrieval even as datasets grow.

Activity 8 - Programming Projects

AVL Tree implementation

Implement a fully functional AVL tree from scratch.

Requirements:

- Implement insertion, deletion, and search operations.
- Handle rotations (single and double) to maintain balance.
- Include tree traversal methods (inorder, preorder, postorder).

AVL TREE IMPLEMENTATION
Date: November 12, 2025
Students: JAVIER, SALVADOR VINCENT R. • BAES, FRANZ EMMANUEL F.
Repository: github.com/slvdrvncntjvr/DataStrucProg

Requirement 1: Implement insertion, deletion, and search operations
A. Insertion Operation

- Location: lines 323–435
  Method: `insert(key: int) -> None`
- Features: Recursive BST insertion, automatic balance maintenance, height updates, triggers rotations when needed

```
avl = AVLTree()
avl.insert(10)
avl.insert(20)
avl.insert(30)
```

B. Deletion Operation

- Location: lines 444–570
- Method: `delete(key: int) -> None`
- Features: Handles leaf, one child, two children; rebalances automatically

```
avl.delete(10)
```

C. Search Operation

- Location: lines 575–623
- Method: `search(key: int) -> Optional[AVLNode]`

```
node = avl.search(20)
```

Requirement 3: Handle rotations (single and double) to maintain balance

- **LL Rotation:** Lines 154–196 → `rotate_right()`
- **RR Rotation:** Lines 198–240 → `rotate_left()`
- **LR Rotation:** Lines 242–268 → `rotate_left_right()`

- **RL Rotation:** Lines 270–293 → `rotate_right_left()`

Requirement 4: Include tree traversal methods (inorder, preorder, postorder)

- **Inorder:** Lines 794–809 → Returns sorted output
- **Preorder:** Lines 811–826 → Root appears first
- **Postorder:** Lines 828–843 → Root appears last
- **Bonus:** Level-order traversal (lines 845–862)

TEST RESULTS

Basic Functionality Test

- Insertion: SUCCESS
- Search: FOUND
- Deletion: SUCCESS
- Traversals: All correct

Rotation Test

All rotation types (LL, RR, LR, RL) verified → Tree balanced in each case

FILE STRUCTURE

```
Programming Project/
├── avl_tree.py (1,014 lines)
└── demo_traversals.py (312 lines)
```

CODE QUALITY METRICS

Organization

- Classes: AVLNode, AVLTree
  Type hints and docstrings (500+ lines)
- Logical method grouping

Algorithm Correctness

- BST property maintained (left < root < right)
- Balance factor |BF| ≤ 1

- All rotations preserve order

Performance

- Insertion: O(log n)
- Deletion: O(log n)
- Search: O(log n)
- Rotations: O(1)

FINAL VERIFICATION

Requirement	Status	Evidence
AVL tree from scratch	COMPLETE	avl_tree.py
Insertion	COMPLETE	Lines 323–435
Deletion	COMPLETE	Lines 444–570
Search	COMPLETE	Lines 575–623
Rotations (LL, RR, LR, RL)	COMPLETE	Lines 154–293
Traversals (in, pre, post)	COMPLETE	Lines 794–843