Assignment No: 2
Date: 18/03/2024

ADVANCED DATA STRUCTURES

ASSIGNMENT - 2

COMPARISON OF INSERTION AND SEARCH ALGORITHMS IN BINARY SEARCH, AVL AND SPLAY TREES

Assignment Type of Submission:			
Group **COMP47500 Advanced Data Structures** **Group Assignment- 2**	Yes/No Yes	**List all group members' details:** Student Name: Avantika Sivakumar Student ID: 23205789 Student Name: Ria Rahul Asgekar Student ID: 23203987 Student Name: Jeevan Raju Ravi Govind Raju Student ID: 23207314	**% Contribution Assignment Workload** 33.33% (Code implementation, Result evaluation, video explanation) 33.33% (Code Implementation, material gathered for report, video explanation) 33.33% (Code implementation, material gathered for report, video explanation)

1. **Problem Domain Description:**

   Implementation and comparison of insertion & search performances of Binary Search Tree, AVL Tree and Splay Tree.

   Binary Search Tree (BST):

   - A BST is a binary tree where the left child of a node contains values less than the node, and the right child contains values greater than the node. This structure enables efficient searching, insertion, and deletion operations with average time complexity of O(log n). However, if the tree becomes unbalanced, worst-case time complexity can degrade to O(n).

   AVL Tree:

   - An AVL Tree is a self-balancing binary search tree where the heights of the left and right subtrees of any node differ by at most one. This balance ensures a worst-case time complexity of O(log n) for all operations by performing rotations to maintain balance after insertions and deletions.

   Splay Tree:

   - A Splay Tree is a self-adjusting binary search tree where recently accessed nodes are moved closer to the root. While it doesn't guarantee a specific worst-case time complexity, it typically achieves amortised logarithmic time complexity for operations. This dynamic adjustment makes it suitable for scenarios with dynamic access patterns where certain nodes are accessed more frequently.

   For this assignment, our problem domain includes:

   1.1. Binary Search Tree Implementation:

   - Developing a Java program to implement a BST data structure
   - Ensuring efficient insertion and search operations within the binary search tree.
   - Using nodes keys and values to arrange them in a hierarchical order such that the left child is less than the parent, and the right child is greater.
   - Insertion recursively places a new key in the appropriate position based on its value.
   - Searching recursively traverses the tree to find the desired key.

   1.2. AVL Tree Implementation:

   - Developing a Java program to implement a self-balancing binary search tree called AVL Tree.
   - Ensuring efficient insertion and search operations within the splay tree.
   - Implementing a mechanism to ensure the tree remains balanced, maintaining a logarithmic time complexity for insertion and searching.
   - Performing rotations (right and left) as needed to restore balance.

1.3. Splay Tree Implementation:

- Developing a Java program to implement a splay tree data structure.
- Ensuring efficient insertion, deletion, and search operations within the splay tree.
- Implementing the splaying mechanism to bring frequently accessed files closer to the root, improving access time for subsequent operations.

1.3. Tree Comparisons:

- Test the insertion and search operations of all three tree structures.
- Insert keys into each tree and measure the time taken for comparison of insertion times.
- Search for keys in each tree and measure the time taken for comparison of search times.
- Repeat tests for various input sizes to observe and validate findings.

1.4. File System Representation:

- Designing a representation for files and directories within the file system.
- Associating each file with a unique identifier and metadata.
- Managing file access and directory structures as needed.

1.5. File Access Operations:

- Implementing methods to add files to the file system.
- Providing functionality to search for files by name or unique identifier within the splay tree.
- Handling file access requests efficiently by leveraging the properties of each tree to optimise access time.

1.6. User Interface:

- Offering a user-friendly interface for users to interact with the file system.
- Allowing users to add and search for files using intuitive commands.
- Displaying information about files and directories.

1.7. Error Handling:

- Detecting and handling various types of errors, such as attempting to access non-existent files or directories.
- Providing informative error messages to users to aid in troubleshooting.

1.8. Testing and Validation:

- Conducting thorough testing to ensure the correctness and efficiency of file access operations.
- Validating the implementations of each tree by testing with various scenarios, including edge cases and large file systems.
- Verifying that file access operations perform optimally, demonstrating the performance and ability of each tree.

2. Theoretical Foundations of the Data Structure(s) utilised

Binary Search Trees:

- In the realm of dynamic data structures, Binary Search Trees (BSTs) serve as fundamental structures for efficient storage and retrieval of elements. BSTs organise elements in a hierarchical order, where each node's key is greater than all keys in its left subtree and less than all keys in its right subtree.
- This hierarchical arrangement enables quick insertion, deletion, and searching operations, making BSTs suitable for various applications such as database indexing, symbol tables, and implementing priority queues.
- However, their performance heavily depends on the tree's balance, and unbalanced trees can degrade to linear search time, prompting the need for self-balancing variants like AVL trees.

AVL Trees:

- AVL Trees, named after their inventors Adelson-Velsky and Landis, are self-balancing binary search trees designed to maintain balance automatically.
- Unlike BSTs, where the height of the tree can degrade to O(n) in the worst-case scenario, AVL trees guarantee O(log n) time complexity for insertion, deletion, and searching operations by ensuring that the height difference between the left and right subtrees of any node (balance factor) remains within [-1, 1].
- This self-balancing property allows AVL trees to handle dynamic datasets efficiently, making them ideal for applications like database systems, where consistent performance is paramount.

Splay Trees:

- Splay Trees, on the other hand, stand out as a self-adjusting variant of binary search trees, specifically optimised for access time. These trees dynamically adapt to access patterns, ensuring that frequently accessed nodes are brought closer to the root, thereby reducing access time for subsequent operations.
- While they may not guarantee strict balance like AVL trees, their ability to optimise access times makes them suitable for scenarios where access patterns change frequently, such as web caching, network routing, and real-time data processing systems.

2.1. Definitions:

- Binary Search Tree:
  A Binary Search Tree (BST) is a hierarchical data structure consisting of nodes in a tree-like form, where each node has at most two children referred to as the left and right subtrees.
- AVL Tree:
  An AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. This balance property ensures that the

tree remains approximately balanced, preventing degeneration into a linked list-like structure and maintaining efficient search, insertion, and deletion operations.

- Splay Tree:
  A splay tree is a self-adjusting binary search tree where frequently accessed elements are moved closer to the root, improving access times for subsequent operations.It maintains the binary search tree property, where elements to the left of a node are smaller, and elements to the right are larger.

2.2. Binary Search Trees:

2.2.1. Structure:

- The structure of a BST is determined solely by the order in which elements are inserted, leading to potential imbalances in the tree. While BSTs provide efficient searching capabilities when balanced, they may exhibit suboptimal performance if skewed towards one side.
- They rely on the sequential insertion of elements to maintain their structure.

2.2.2. Insertion and Search:

- Insertion and search operations in a BST involve traversing the tree recursively based on the key values of nodes. During insertion, elements are placed according to their key values, ensuring that the binary search tree property is maintained. Similarly, searching involves recursively traversing the tree to find the target element based on its key value.

2.2.3. Performance:

- The performance of a BST depends on its balance. In the worst-case scenario, where the tree is highly unbalanced, searching, insertion, and deletion operations may degrade to O(n) time complexity. However, when the tree is balanced, typically achieved through careful insertion strategies or using self-balancing trees like AVL trees, operations maintain their efficient O(log n) time complexity.

2.3. AVL Trees:

2.3.1. Self-Balancing Property:

- The core characteristic of AVL trees is their self-balancing property, which is maintained through rotations performed during insertion and deletion operations. Whenever an insertion or deletion potentially violates the balance property of the tree, rotations are applied to restore balance.
- This self-balancing behaviour ensures that the height of the tree remains logarithmic, optimising the time complexity of operations.

2.3.2. Rotations:

- AVL trees employ four types of rotations to maintain balance: single left rotation, single right rotation, double left rotation (left-right rotation), and double right rotation (right-left rotation).
- These rotations are performed when a subtree becomes unbalanced due to insertion or deletion. Single rotations adjust the tree by rearranging nodes around a pivot point, while double rotations involve performing two single rotations to achieve balance.

2.3.3. Insertion and Search:

- Insertion and search operations in AVL trees are similar to those in standard binary search trees. However, AVL trees include additional steps to ensure balance after insertion.
- During insertion, the tree is traversed recursively to find the appropriate position for the new node.
- After insertion, the balance property of each node along the insertion path is checked, and rotations are applied if necessary to restore balance.

2.3.4. Performance:

- The performance of AVL trees is characterised by their self-balancing property, which guarantees that the height of the tree remains logarithmic relative to the number of nodes.
- While the balancing process incurs some overhead, the benefits of maintaining balance outweigh this cost, making AVL trees suitable for dynamic datasets with frequent updates.

2.4. Splay Trees:

2.4.1. Self-Adjusting Property:

- The core principle of a splay tree is its self-adjusting behaviour, where accessing or manipulating a node causes it to move closer to the root.
- This property ensures that frequently accessed nodes are positioned near the root, reducing access time for future operations.

2.4.2. Rotations:

- Splay trees utilise rotation operations (rightRotate and leftRotate) to restructure the tree during splaying.
- A right rotate operation is performed on a node and its left child. It restructures the tree by making the left child the new root, with the original node as the right child of the new root.
- A left rotate operation is performed on a node and its right child. It restructures the tree by making the right child the new root, with the original node as the left child of the new root.
- Rotations adjust the tree's structure to maintain the binary search tree property and facilitate efficient access to elements.

2.4.3. Splaying:

- Splaying is the process of bringing a target node closer to the root by performing a sequence of rotations.
- It involves traversing the tree from the target node to the root while applying rotations to adjust the tree's structure.
- Splaying ensures that frequently accessed nodes rise towards the root, optimising access times for subsequent operations.

2.4.4. Insertion and Search:

- When inserting or searching for a node in a splay tree, the tree is splayed to bring the target node closer to the root.
- This process improves the efficiency of subsequent operations involving the accessed node.

2.4.5. Performance:

- The performance of a splay tree is influenced by its self-adjusting behaviour, which adapts to access patterns over time.
- While individual operations may not guarantee balanced trees, the amortised time complexity remains competitive, making splay trees suitable for dynamic datasets.

Overall, Binary Search Trees (BSTs) organise data in a binary tree structure, facilitating efficient search operations with logarithmic time complexity but prone to becoming unbalanced, affecting performance. AVL Trees address this issue by maintaining balance through rotations to ensure logarithmic search time, while Splay Trees further optimise access times by dynamically adjusting the tree structure during operations to bring frequently accessed elements closer to the root.

3. Applications:

3.1. Binary Search Tree:

- Symbol Tables: Binary Search Trees are widely used to implement symbol tables in compilers, interpreters, and other language processing tools for efficient storage and retrieval of identifiers.
- File Systems: They are utilised in file systems for directory organisation and efficient searching, enabling quick lookup of files and directories based on their names or metadata.
- Database Systems: In database management systems, BSTs are employed for indexing, facilitating fast retrieval of records based on indexed attributes, enhancing query performance.
- Networking: BSTs are used in network routing algorithms for efficient packet routing and forwarding, where they help in quickly determining the next hop for data packets.
- Genetic Algorithms: In bioinformatics and computational biology, BSTs play a role in representing and searching genetic data efficiently, aiding in tasks such as sequence alignment and phylogenetic analysis.

3.2. AVL Tree:

- Database indexing: AVL trees are used to efficiently index and retrieve data in databases, speeding up query execution.
- File systems: AVL trees help maintain directory structures and optimise file access in file systems by providing efficient searching and insertion.
- Network routers: AVL trees are employed in network routers for managing routing tables, facilitating fast lookup and routing decisions.
- Language compilers: AVL trees are utilised in language compilers to implement symbol tables, aiding in efficient variable and function lookup during compilation.
- Priority queues: AVL trees can be used to implement priority queues, where elements with higher priority (lower key) are processed first.
- Dynamic programming: AVL trees find applications in dynamic programming algorithms, such as optimal binary search trees, where they help achieve optimal time complexity for certain problems.

3.3. Splay Trees:

- Expression Evaluation: Splay trees find applications in expression evaluation tasks where efficient access to frequently accessed elements is crucial, such as in managing and accessing files in a file system based on access patterns.
- Cache Management: Splay trees can be utilised in cache management systems to optimise access times by keeping frequently accessed data closer to the root.
- Database Indexing: Splay trees can serve as efficient indexing structures in databases, where quick access to frequently queried data is essential for performance.
- Network Routing: Splay trees can be applied in network routing algorithms to optimise routing decisions by prioritising frequently accessed routes.

4. Time Complexity:

Insertion Time Complexity:

- Binary Search Tree: In the worst-case scenario, insertion in a BST has a time complexity of O(n), where n is the number of elements in the tree. This occurs when the tree becomes skewed, leading to a linear search.
- AVL Tree: AVL trees ensure balance after every insertion, resulting in an average and worst-case time complexity of O(log n), where n is the number of elements. This is achieved by performing rotations to maintain balance.
- Splay Tree: Splay trees also have an amortised insertion time complexity of O(log n), where n is the number of elements. However, individual insertions might take O(n) in the worst case if the tree becomes unbalanced due to splaying.

Search Time Complexity:

- Binary Search Tree: In a balanced BST, search operations have a time complexity of O(log n), where n is the number of elements. However, in the worst-case scenario where the tree is skewed, search time complexity degrades to O(n).
- AVL Tree: AVL trees maintain balance, ensuring that search operations have a time complexity of O(log n) in both average and worst-case scenarios.
- Splay Tree: Splay trees exhibit an amortised search time complexity of O(log n), where n is the number of elements. Similar to insertion, individual search operations might take O(n) in the worst case if the tree becomes unbalanced due to splaying.

To summarise, AVL trees provide guaranteed logarithmic time complexity for both insertion and search operations due to their self-balancing property. BSTs offer O(log n) time complexity for balanced trees but can degrade to O(n) in the worst case. Splay trees offer amortised logarithmic time complexity, but individual operations might become linear in the worst-case scenario.

5. Advantages:

5.1 Binary Search Trees:

5.1.1. Efficient Search Operation: BSTs offer efficient searching with a time complexity of O(log n) on average, making them suitable for applications requiring quick retrieval of data.

5.1.2. Simple Implementation: BSTs are relatively straightforward to implement compared to more complex data structures like AVL trees or Red-Black trees, making them a popular choice for various applications.

5.1.3. Memory Efficiency: BSTs typically require less memory overhead compared to other self-balancing trees, as they only store the key value and pointers to left and right children.

5.1.4. Flexible Data Storage: BSTs allow for flexible storage of data with the ability to insert, delete, and search for elements in sorted order, providing versatility in applications ranging from databases to symbol tables.

5.1.5. Easy Traversal: BSTs facilitate easy traversal of elements in sorted order, which can be beneficial in tasks like printing data in ascending or descending order, range queries, and finding predecessor/successor elements.

5.1.6. Adaptability: While BSTs do not self-balance automatically like AVL or Red-Black trees, they can be rebalanced if necessary, offering a balance between simplicity and performance in certain scenarios.

5.2. AVL Tree:

5.2.1. Balanced Structure: AVL trees maintain balance after each insertion and deletion, ensuring that the height of the tree remains logarithmic. This balanced structure facilitates efficient search, insertion, and deletion operations, resulting in consistent performance.

5.2.2. Guaranteed Time Complexity: With their balanced nature, AVL trees guarantee logarithmic time complexity for search, insertion, and deletion operations in both average and worst-case scenarios. This predictability makes them suitable for applications requiring consistent performance.

5.2.3. Self-Balancing Property: The self-balancing property of AVL trees eliminates the need for manual rebalancing operations. As nodes are inserted or removed, AVL trees automatically adjust their structure to maintain balance, simplifying tree management.

5.2.4. Optimised Performance: Due to their balanced nature, AVL trees offer optimised performance for dynamic datasets where frequent insertions and deletions occur. They are commonly used in scenarios where efficient search and modification operations are crucial, such as database indexing and compiler implementations.

5.3. Splay Tree:

5.3.1. Efficient Access Patterns:

- Splay trees optimise file access patterns by rearranging nodes based on the frequency of access. Frequently accessed files are brought closer to the root, reducing access time for subsequent operations.
- This dynamic adjustment ensures that commonly accessed files remain easily accessible, leading to improved performance in file retrieval tasks.

5.3.2. Self-Adjusting Behaviour:

- Splay trees exhibit self-adjusting behaviour, automatically adapting to changes in access patterns over time.
- Unlike static data structures, splay trees continuously optimise themselves based on recent access history, making them well-suited for dynamic environments where access patterns evolve.

5.3.3. Adaptive Performance:

- The performance of splay trees improves with repeated access to specific files, as frequently accessed files are splayed closer to the root, reducing traversal distances.
- This adaptive behaviour ensures that the most accessed files experience faster access times, enhancing overall system performance.

5.3.4. Simplified Management:

- Splay trees simplify file access management by automatically organising files based on access frequency.
- Manual intervention for optimising file access patterns is minimised, as the tree structure adapts to access patterns autonomously.

5.3.5. Reduced Access Overhead:

- Splay trees minimise access overhead by prioritising frequently accessed files, thereby reducing the time required for file retrieval operations.
- Files that are accessed less frequently are naturally positioned farther from the root, mitigating the impact of infrequent accesses on overall system performance.

5.3.6. Versatility:

- Splay trees find applications beyond file access optimization, such as in caching mechanisms, database indexing, and network routing.
- Their ability to dynamically adjust to access patterns makes them suitable for various scenarios where efficient data access is essential.

In conclusion, each type of tree structure offers distinct advantages: Binary Search Trees provide simplicity and ease of implementation, AVL trees ensure consistent logarithmic time complexity for operations, and Splay trees dynamically adapt to access patterns for improved performance. The choice of tree structure depends on the specific requirements of the application, balancing factors such as simplicity, performance guarantees, and adaptability to access patterns.

6. History:

6.1. Binary Search Trees (BST):

Theoretical Foundations: Binary Search Trees were first introduced in the 1960s as a fundamental data structure for organising and searching data. The concept is rooted in the binary tree structure, where each node has at most two children. BST maintains a hierarchical order, ensuring that elements to the left of a node are smaller, and elements to the right are larger. This ordering facilitates efficient searching, insertion, and deletion operations, with average time complexities of O(log n) for balanced trees.

- Practical Applications: BSTs find applications in various domains, including database systems, compilers, and file systems. They are commonly used for implementing symbol tables, dictionary data structures, and in-memory databases due to their fast search times and ease of implementation.

- Legacy and Impact: BSTs have had a significant impact on computer science and software engineering since their inception. They form the basis for more complex data structures and algorithms, such as AVL trees and Red-Black trees, which aim to address the limitations of BSTs in terms of balancing and performance.

- Further Research and Development: Ongoing research in BSTs focuses on improving their performance and efficiency, particularly in scenarios with skewed data distributions. Techniques such as self-balancing trees and adaptive data structures continue to be explored to optimise the trade-offs between search time, space complexity, and ease of maintenance.

6.2. AVL Trees:

- Theoretical Foundations: AVL trees, developed by Adelson-Velsky and Landis in 1962, introduced the concept of self-balancing binary search trees. These trees enforce strict balance conditions to ensure that the height difference between subtrees remains within a limited range, typically logarithmic in the number of nodes. AVL trees achieve this balance through rotations during insertions and deletions, maintaining an overall height of O(log n).

- Practical Applications: AVL trees are widely used in database indexing, where maintaining balanced trees is crucial for efficient search and retrieval operations.

They also find applications in compilers, where symbol tables need to be quickly searched and updated.

- Legacy and Impact: AVL trees represent a significant milestone in the development of self-balancing data structures. Their impact extends to subsequent research in tree-based data structures, such as Red-Black trees and B-trees, which further refine balancing techniques for optimal performance in different scenarios.

- Further Research and Development: Ongoing research in AVL trees focuses on optimising their performance for specific applications, such as real-time systems and embedded databases. Techniques like AVL tree variants and hybrid data structures aim to address the trade-offs between balancing constraints and practical usability in modern computing environments.

6.3. Splay Trees:

- Theoretical Foundations: Splay trees were introduced by Sleator and Tarjan in 1985 as a self-adjusting binary search tree designed to optimise access times dynamically. Unlike AVL trees, which enforce strict balance, splay trees prioritise frequent access to elements by dynamically adjusting the tree structure based on recent operations. This self-adjusting mechanism, achieved through splaying—a sequence of tree rotations—optimises access times by bringing frequently accessed nodes closer to the root.

- Practical Applications: Splay trees excel in scenarios where access patterns are unpredictable or skewed, such as caching systems, web servers, and network routing algorithms. They are commonly used in memory management systems and database query optimization, where efficient access to frequently accessed data is essential.

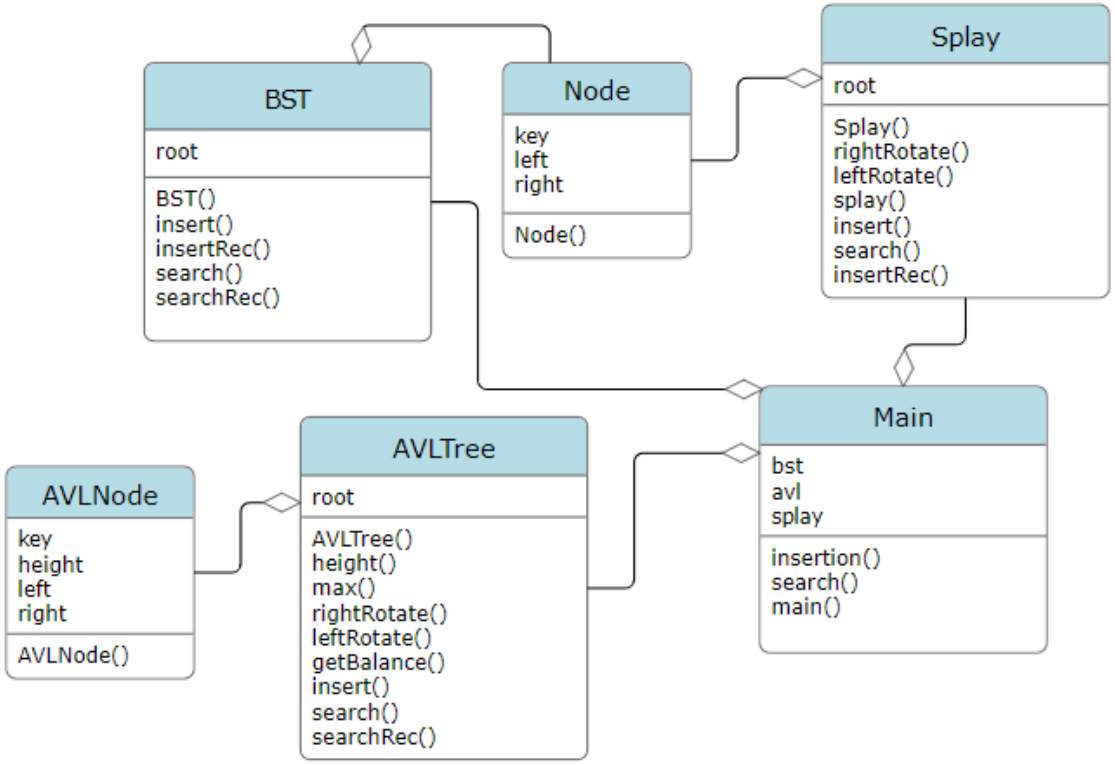
- Legacy and Impact: Splay trees represent a paradigm shift in tree-based data structures, introducing the concept of self-adjusting trees to optimise access times. Their impact extends to various fields, including algorithm design, distributed systems, and operating systems, where efficient data access is crucial for performance.

- Further Research and Development: Ongoing research in splay trees focuses on refining their self-adjusting mechanisms and optimising their performance for modern computing environments. Techniques such as cache-aware splay trees and adaptive data structures aim to address the trade-offs between access times, space complexity, and scalability in large-scale systems.

.

7. Analysis/Design (UML Diagram(s))

The UML diagram representing the SplayTree class and its internal structure:



BST Insertion:

1. Initialise a pointer 'current' to the root of the BST.
2. If the BST is empty (root is null):
   - Create a new node with the given key and set it as the root.
   - Return the root.
3. Otherwise, do the following while 'current' is not null:
   a. If 'key' is less than the key of 'current':
      - If 'current' has no left child:
         * Create a new node with the given key and set it as the left child of 'current'.
         * Break from the loop.
      - Otherwise, update 'current' to its left child.
   b. Else if 'key' is greater than the key of 'current':
      - If 'current' has no right child:
         * Create a new node with the given key and set it as the right child of 'current'.
         * Break from the loop.
      - Otherwise, update 'current' to its right child.
   c. Else (key is equal to the key of 'current'):
      - No duplicate keys allowed in BST. You can either handle this case as per requirement or ignore duplicates.
4. Return the root of the BST.

BST Search:

1. If 'root' is null or the key of 'root' is equal to 'key':
    - Return true if 'root' is not null (indicating the key is found), otherwise return false.
2. If 'key' is greater than the key of 'root':
    - Recursively search in the right subtree by calling searchRec(root.right, key).
3. Otherwise (key is less than the key of 'root'):
    - Recursively search in the left subtree by calling searchRec(root.left, key).
4. Return the result obtained from the recursive calls.

AVL Insertion:

1. If 'node' is null:
    - Create a new AVL node with the given key and return it.
2. If 'key' is less than the key of 'node':
    - Recursively insert the key into the left subtree by calling insert(node.left, key).
3. Otherwise, if 'key' is greater than the key of 'node':
    - Recursively insert the key into the right subtree by calling insert(node.right, key).
4. Otherwise (key is equal to the key of 'node'):
    - Return 'node' without any changes (no duplicate keys allowed in AVL tree).
5. Update the height of 'node' as 1 plus the maximum of the heights of its left and right subtrees.
6. Calculate the balance factor of 'node' by subtracting the height of its right subtree from the height of its left subtree.
7. If the balance factor is greater than 1 and 'key' is less than the key of the left child of 'node':
    - Perform a right rotation on 'node' by calling rightRotate(node).
8. If the balance factor is less than -1 and 'key' is greater than the key of the right child of 'node':
    - Perform a left rotation on 'node' by calling leftRotate(node).
9. If the balance factor is greater than 1 and 'key' is greater than the key of the left child of 'node':
    a. Perform a left rotation on the left child of 'node' by calling leftRotate(node.left).
    b. Perform a right rotation on 'node' by calling rightRotate(node).
10. If the balance factor is less than -1 and 'key' is less than the key of the right child of 'node':
    a. Perform a right rotation on the right child of 'node' by calling rightRotate(node.right).
    b. Perform a left rotation on 'node' by calling leftRotate(node).
11. Return the updated 'node'.

AVL Search:

1. If 'root' is null or the key of 'root' is equal to 'key':
    - Return true if 'root' is not null (indicating the key is found), otherwise return false.
2. If 'key' is greater than the key of 'root':

- Recursively search in the right subtree by calling searchRec(root.right, key).
3. Otherwise (key is less than or equal to the key of 'root'):
- Recursively search in the left subtree by calling searchRec(root.left, key).
4. Return the result obtained from the recursive calls.

Splay Insertion:

1. If 'root' is null:
- Create a new node with the given key and return it.
2. Perform splaying operation on the tree rooted at 'root' with the key 'key' by calling splay(root, key).
3. If the key of the root node after splaying is equal to the key being inserted:
- Return the root node.
4. Create a new node 'newNode' with the given key.
5. If the key of the root node after splaying is greater than the key being inserted:
- Set 'newNode' as the right child of the root node.
- Set the left child of 'newNode' as the left subtree of the root node.
- Set the left subtree of the root node to null.
6. Otherwise (the key of the root node after splaying is less than the key being inserted):
- Set 'newNode' as the left child of the root node.
- Set the right child of 'newNode' as the right subtree of the root node.
- Set the right subtree of the root node to null.
7. Return 'newNode' as the new root node of the Splay Tree.

Splay Search:

1. If 'root' is null or the key of 'root' is equal to 'key':
- Return 'root'.
2. If the key of 'root' is greater than 'key':
a. If the left child of 'root' is not null:
- If the key of the left child of 'root' is greater than 'key':
- Recursively perform splaying on the left-left subtree by calling splay(root.left.left, key).
- Perform a right rotation on 'root' by calling rightRotate(root).
- Otherwise, if the key of the left child of 'root' is less than 'key':
- Recursively perform splaying on the left-right subtree by calling splay(root.left.right, key).
- If the left-right subtree is not null, perform a left rotation on the left child of 'root' by calling leftRotate(root.left).
- Perform a right rotation on 'root' by calling rightRotate(root).
- Return 'root' after the rotation.
3. If the key of 'root' is less than 'key':
a. If the right child of 'root' is not null:
- If the key of the right child of 'root' is greater than 'key':
- Recursively perform splaying on the right-left subtree by calling splay(root.right.left, key).

- If the right-left subtree is not null, perform a right rotation on the right child of 'root' by calling rightRotate(root.right).
- Perform a left rotation on 'root' by calling leftRotate(root).
- Otherwise, if the key of the right child of 'root' is less than 'key':
- Recursively perform splaying on the right-right subtree by calling splay(root.right.right, key).
- Perform a left rotation on 'root' by calling leftRotate(root).
- Return 'root' after the rotation.

8. Time and Space Complexities:

The time and space complexities of operations in each tree are crucial for understanding its efficiency and performance characteristics. Below are the time and space complexities of various operations in a Binary Search Tree, AVL Tree and Splay Tree:

8.1 Binary Search Tree (BST):

Insertion:

- Time Complexity: In the average case, the time complexity of insertion in a BST is O(h), where h is the height of the tree. However, in the worst case, when the tree becomes unbalanced (e.g., skewed), the time complexity can degrade to O(n), where n is the number of nodes in the tree.
- Space Complexity: The space complexity of insertion in a BST is O(1) for each insertion operation, as it only requires creating a new node to store the inserted element.

Searching:

- Time Complexity: Similar to insertion, the average time complexity of searching in a BST is O(h), where h is the height of the tree. In the worst case, the time complexity can degrade to O(n) for skewed trees.
- Space Complexity: The space complexity of searching in a BST is O(1) for each search operation, as it does not require any additional space beyond the iterative traversal of the tree.

8.2. AVL Tree:

Insertion:

- Time Complexity: The time complexity of insertion in an AVL tree is O(log n), where n is the number of nodes in the tree. This is because AVL trees maintain balance during insertion, ensuring that the height of the tree remains logarithmic.
- Space Complexity: The space complexity of insertion in an AVL tree is O(log n) for each insertion operation, as it involves recursive calls to adjust the tree structure to maintain balance.

Searching:

- Time Complexity: Similar to insertion, the time complexity of searching in an AVL tree is O(log n), where n is the number of nodes in the tree. AVL trees maintain balance, resulting in efficient searching operations.
- Space Complexity: The space complexity of searching in an AVL tree is O(1) for each search operation, as it does not require any additional space beyond the iterative traversal of the tree.

8.3. Splay Tree:

Insertion:

- Time Complexity: In a splay tree, the time complexity of insertion is amortised O(log n), where n is the number of nodes in the tree. This is because splay trees dynamically adjust their structure to maintain balance, ensuring efficient insertion operations over time.
- Space Complexity: The space complexity of insertion in a splay tree is O(1) for each insertion operation, as it only requires creating a new node to store the inserted element.

Searching:

- Time Complexity: Similar to insertion, the time complexity of searching in a splay tree is amortised O(log n), where n is the number of nodes in the tree. Splay trees adapt their structure based on access patterns, optimising search times.
- Space Complexity: The space complexity of searching in a splay tree is O(1) for each search operation, as it does not require any additional space beyond the iterative traversal of the tree.

Splaying:

- Time Complexity: The time complexity of the splaying operation depends on the depth of the node being splayed and the structure of the tree. In the worst-case scenario, splaying on a splay tree may take O(n) time, but in practice, it is amortised to O(log n) over multiple operations.
- Space Complexity: Similar to other tree operations, the space complexity of splaying on a splay tree is O(1), as it operates directly on the existing nodes of the tree without requiring additional space.

9. IMPLEMENTATION:

Binary Search Trees, AVL Trees, and Splay trees have all been implemented in various programming languages and utilised in numerous applications due to their various different advantages and versatility. Here's an overview of different implementations and their applications for each tree:

9.1. Binary Search Tree (BST):

- Python: In Python, a BST can be implemented using classes and methods to represent nodes and operations like insertion, deletion, and searching. Python's object-oriented features make it easy to define a Node class with left and right pointers, along with methods for insertion, deletion, and searching recursively.
- C++: C++ allows for the creation of BST implementations using pointers and structures. The implementation typically involves defining a Node structure with left and right pointers, along with methods for insertion, deletion, and searching recursively or iteratively.
- Java: In Java, a BST implementation can be achieved using classes and objects. The Node class can have left and right references, and methods can be defined for insertion, deletion, and searching recursively or iteratively.

9.2. AVL Tree:

- Python: Implementing an AVL tree in Python involves creating classes for nodes and the AVL tree itself. Rotation operations like leftRotate and rightRotate are defined along with balancing methods to maintain AVL tree properties during insertion and deletion.
- C++: In C++, AVL tree implementation requires defining Node structures with balance factors along with methods for balancing the tree after insertion and deletion operations. Rotation functions are implemented to perform left and right rotations.
- Java: Java implementations of AVL trees are similar to BSTs but include additional logic to ensure tree balance. Methods for balancing the tree, such as rebalancing after insertion or deletion, are defined along with rotation operations.

9.3. Splay Tree:

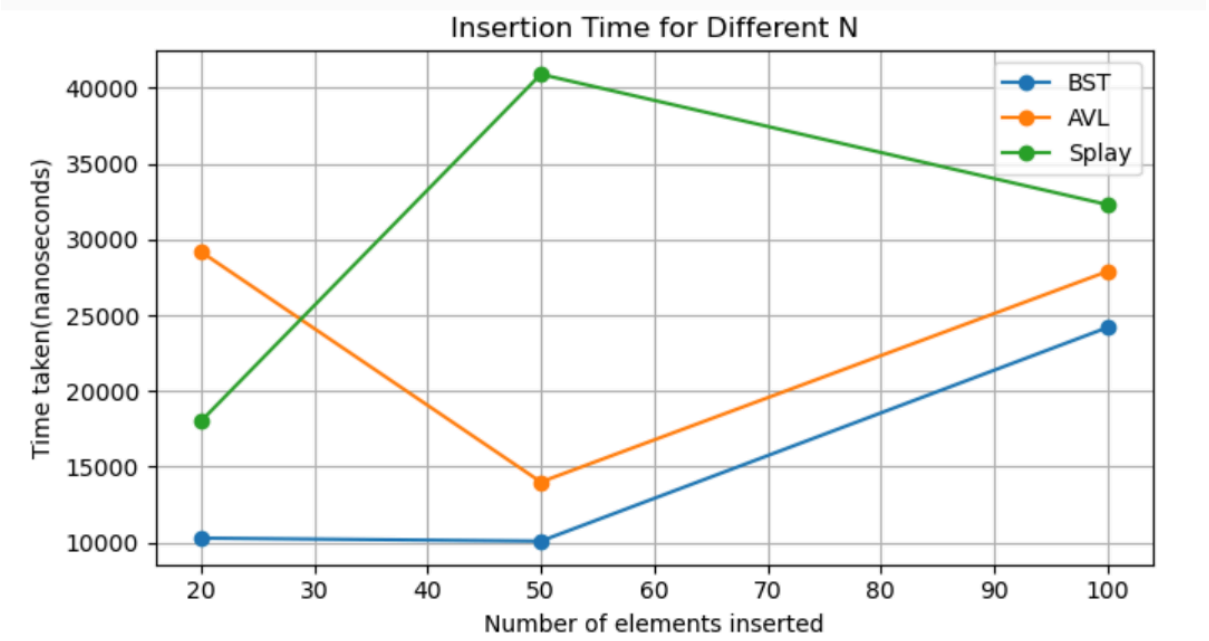
- Python: Implementing a splay tree in Python involves defining classes for nodes and the tree itself. The splaying operation can be implemented recursively or iteratively, bringing the accessed node to the root while maintaining the tree's structure.
- C++: C++ implementations of splay trees require defining node structures and methods for splaying operations. The splay operation can be implemented recursively or iteratively to bring the accessed node to the root.
- Java: Java implementations of splay trees are similar to other tree structures but include splaying logic to adjust the tree structure during search operations. The splaying operation can be implemented recursively or iteratively to move the accessed node closer to the root.

10. RUN:

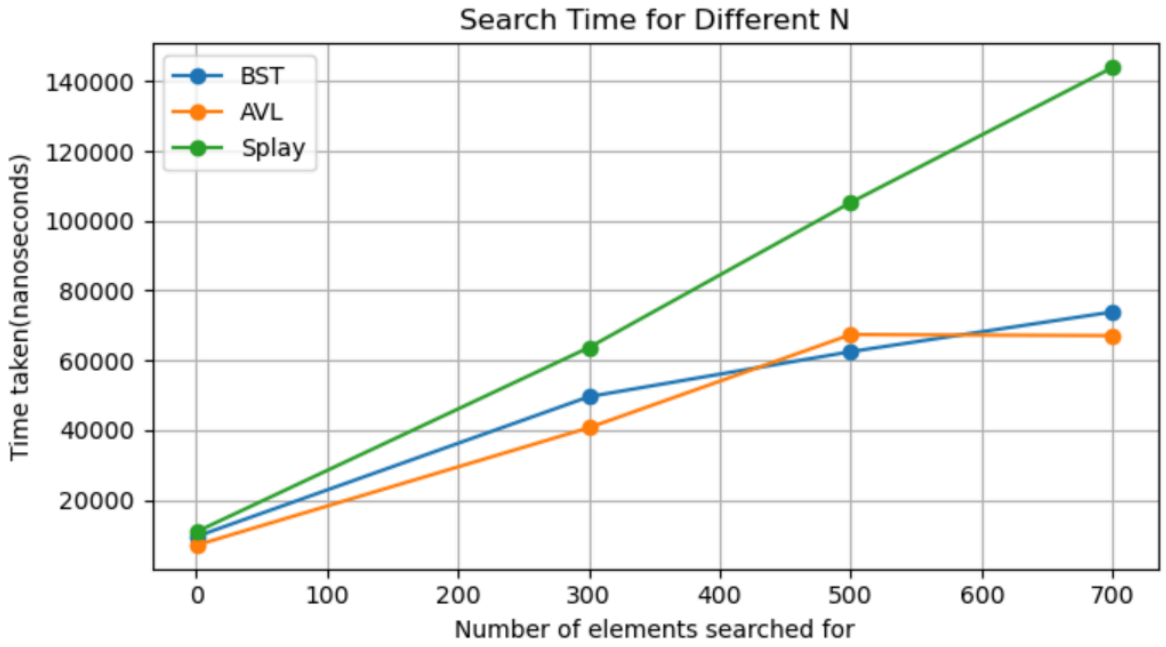
```
N           BST         AVL         SPLAY
--------INSERTION---------
1           590300      481900      5300
20          13400       44200       21800
20          10300       35700       20300
20          39000       32600       18800
20          11700       29200       18000
20          18700       44600       20000
50          12500       73800       59500
50          10800       75100       51800
50          11200       52800       41200
50          10100       29700       52700
50          11600       14000       40900
50          12100       18700       49000
50          10800       14000       44000
100         27300       29200       79400
100         24200       27900       40700
100         30200       29200       32300
100         32400       54300       44000
100         33600       39600       44000
--------SEARCH---------
1           9600        7100        11000
300         200200      196500      120700
300         106800      92600       128000
300         70200       49000       96100
300         49600       40700       65000
300         60400       45700       63700
500         79900       67400       105300
500         98000       83600       125000
500         125700      112900      166100
500         144300      95500       146300
500         62500       77500       110600
500         71800       90800       107000
500         73900       70000       147700
700         81800       76700       144000
700         81100       67100       157900
700         111800      158100      232600
700         142500      94900       180800
700         153800      124500      276000
```

Results:

The experiments test the performances of the insertion and search functions of the 3 trees. The time taken is tabulated and plotted in the below graphs for better visualisation. The first data point is omitted from the graph as that test case takes extra time, since it is the first iteration of the loop and the code is optimised by the JIT compiler during the initial runs.



1. Comparing insertion times, it is observed that overall, splay trees seem to perform much worse than the other two. The time taken improves slightly as the size of the tree increases beyond a certain value.
2. BST insertion is the most efficient overall, and consistently takes less time than the other two trees.
3. Insertion in AVL trees is slightly longer than in BST, but still much quicker than in splay trees.



1. Search times for all 3 trees increases linearly with an increase in the number of elements searched for, with this increase being much more significant in splay trees.
2. Searching in BST and AVL trees has similar performance times.

3. This aligns with our understanding of Splay Trees, that only recently accessed items have faster access times.

Code Implementation GitHub (link):
https://github.com/RiaAsgekar/COMP47500_Assignment2

Set of Experiments run and results:
https://github.com/RiaAsgekar/COMP47500_Assignment2/blob/main/Main.java

Video of the Implementation running
Zoom (link & password):
https://ucd-ie.zoom.us/rec/share/gSTKauWKCFlyG3NrjVHM0AYsUntgodUgaDV866UQZ4AHFbbjJ8azBI6SYW6bryXg.W4VoTonMHvEwxnXk?startTime=1710764130000

Passcode: **6aa?WACk**

Comments:
Jeevan gives an overview of our chosen problem domain, explains BST Trees, AVL Trees and Splay Trees, principle and applications involved, program structure. Ria explains about the BST and AVL classes and their functions. Avantika explains about the Splay and Main class and talks about the test cases involved.

References:

1. Albers, S., & Karpinski, M. (2002). Randomized splay trees: Theoretical and experimental results. Information Processing Letters, 81(4), 213-221.
2. Iacono, J. (2001, January). Alternatives to splay trees with O (log n) worst-case access times. In Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (pp. 516-522).
3. Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. Journal of the ACM (JACM), 32(3), 652-686.
4. https://www.geeksforgeeks.org/introduction-to-splay-tree-data-structure/
5. https://www.geeksforgeeks.org/binary-search-tree-data-structure/
6. https://www.geeksforgeeks.org/introduction-to-avl-tree/
7. Tree concept in data structure by Rubi Dhankhar , Sapna Kamra , Vishal Jangra (2014 IJIRT(Volume 1 Issue 7, ISSN: 2349-6002)
8. https://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec08-splay/rec08.html
9. https://www.programiz.com/dsa/trees
10. https://www.javatpoint.com/splay-tree
11. https://www.javatpoint.com/binary-search-tree-vs-avl-tree#:~:text=In%20BST%2C%20there%20is%20no,balanced%20or%20an%20unbalanced%20tree