NAME: SHEHRYAR KHAN
ROLL NO: 12394
LAB TASK: 13
SUBMITTED TO: SIR JAMAL

Learning Objectives:

- Define and understand the concept of binary trees.
- Implement basic tree operations in Python: node creation, insertion, traversals.
- Explore different tree traversal methods (preorder, inorder, postorder).
- Analyze the time and space complexity of tree operations.
- Distinguish between binary trees and binary search trees (BSTs).
- Implement BST operations: search, insertion, deletion.
- Solve problems using tree structures.

Lab Structure:

1. Introduction to Binary Trees:
    - Definition and terminology (node, root, leaf, level, height, depth)
    - Visual representations of binary trees
    - Applications of binary trees

Definition and terminology:

Here's a breakdown of key terms related to binary trees:

Node:

- The fundamental building block of a binary tree.
- Stores a data value and has at most two child nodes, referred to as the left child and right child.

Root Node:

- The topmost node in the tree, with no parent node.
- It serves as the starting point for accessing other nodes in the tree.

Leaf Node:

- A node without any children, representing the end of a branch in the tree.

Level:

- The position of a node in terms of its distance from the root node.
- The root node is at level 0, its children are at level 1, its grandchildren are at level 2, and so on.

Height:

- The length of the longest path from the root to a leaf node.
- It represents the maximum number of levels in the tree.

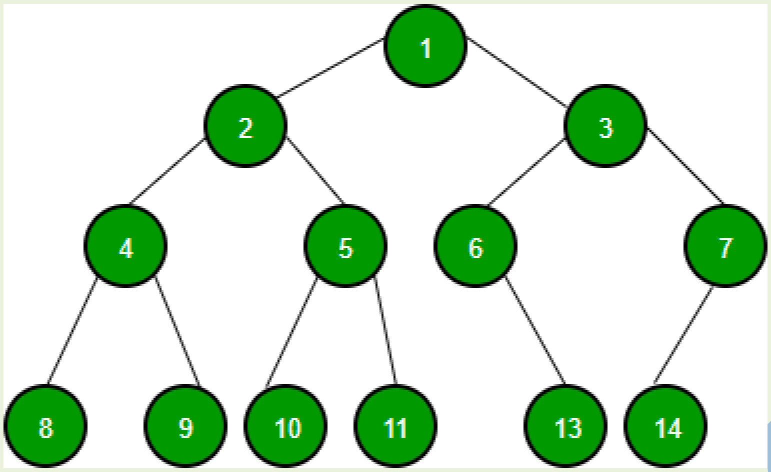
Depth:

- The length of the path from the root to a specific node.
- It indicates how far down a node is located in the tree's hierarchy.

Key Points:

2. Each node in a binary tree can have at most two children, one on the left and one on the right.
3. The arrangement of nodes creates a hierarchical structure.
4. The root node is the only node without a parent.
5. Leaf nodes are terminal nodes with no children.

6. Height and depth are measures of a tree's structure and are used in various tree operations and algorithms.



Applications of Binary Tree:

Here are some common applications of binary trees, along with illustrative examples:

1. Search Trees:

- Binary Search Trees (BSTs): Organize data in a sorted manner, allowing for efficient search, insertion, and deletion operations.

  - Used in databases, search engines, and file systems.

  - Example: A dictionary app stores words in a BST for fast lookups.

2. Expression Trees:

- Represent arithmetic or logical expressions, facilitating evaluation and parsing.

    - Used in compilers and interpreters for programming languages.

    - Example: A calculator app uses an expression tree to evaluate formulas.

    [Image of an expression tree representing an arithmetic expression]

3. Heaps:

- Specialized binary trees that maintain a specific order property (min-heap or max-heap).

    - Used for priority queues, sorting algorithms (Heap Sort), and graph algorithms (Dijkstra's algorithm).

    - Example: A task scheduler uses a min-heap to prioritize urgent tasks.

4. Huffman Coding:

- Construct optimal binary trees for data compression.

    - Used in file compression algorithms like ZIP and JPEG.

    - Example: A compression tool uses Huffman trees to reduce file sizes.

5. Decision Trees:

- Represent decision-making processes, used for classification and prediction.

    - Used in machine learning algorithms for tasks like spam detection and medical diagnosis.

    - Example: A spam filter uses a decision tree to classify emails as spam or not spam.

6. Trie (Prefix Tree):

- Store strings efficiently for fast retrieval and pattern matching.

    - Used in auto-complete features, spell checkers, and routing algorithms.

    - Example: A search engine uses a trie to quickly suggest search terms.

7. Game Trees:

- Represent possible moves and states in games for decision-making.

    - Used in game AI for games like chess, checkers, and Go.

    - Example: A chess-playing program uses a game tree to evaluate potential moves.

8. File Systems:

- Represent hierarchical directory structures.

    - Used in operating systems to manage files and folders.

    - Example: Your computer's file explorer uses a tree structure to display files and folders.

9. Syntax Trees:

- Represent the syntactic structure of programming languages.

    - Used in compilers and interpreters for code analysis and translation.

    - Example: A compiler uses a syntax tree to analyze the structure of code before generating machine code.

Creating a Node Class:

- Define a `Node` class with `data`, `left`, and `right` attributes

```
class Node:
    def __init__(self, data):
        self.data = data
```

```
        self.left = None    # Initially, left and right children are None
        self.right = None
```

Instantiate nodes with data values

```
root = Node(10)  # Create a node with data 10
print(root.data)   # Output: 10
print(root.left)   # Output: None (initially)
root.left = Node(5)   # Assign a left child node
root.right = Node(15) # Assign a right child node
```

7. **Tree Insertion:**

Implement a function to insert a node into a binary tree

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def insert(root, data):
    """Inserts a new node with the given data into the binary tree."""

    if root is None:
        return Node(data)  # Create a new root node if the tree is empty

    if data < root.data:
        root.left = insert(root.left, data)  # Recursively insert into the left subtree
    else:
        root.right = insert(root.right, data)  # Recursively insert into the right subtree

    return root  # Return the updated root node
```

Discuss strategies for choosing insertion positions (e.g., left-biased):

Choosing an insertion position in a binary tree can impact its structure and its search/retrieval efficiency. While various strategies exist, two common approaches are:

1. Left-Biased Insertion:

- Always insert new nodes into the left subtree if possible.
- This strategy tends to create trees with smaller right subtrees and a deeper left subtree.

  Advantages:

  - Improves average search performance, as more nodes are concentrated in the left subtree, leading to faster comparisons during search.
  - Can reduce tree height compared to right-biased insertion, potentially leading to faster operations.

  Disadvantages:

  - May result in an unbalanced tree in the worst case, especially with heavily skewed data distribution.
-

2. Right-Biased Insertion:

- Always insert new nodes into the right subtree if possible.
- This strategy creates trees with larger left subtrees and a shallower right subtree.

  Advantages:

  - Can provide better worst-case search performance compared to left-biased insertion.
  - May be beneficial for specific data distributions where smaller tree height is important.

  Disadvantages:

  - Can worsen average search performance due to potentially deeper left subtrees.
  - Might lead to unbalanced trees in the worst case with skewed data.
-

Other Strategies:

- Randomized Insertion: Choose the insertion direction randomly (left or right) to avoid favoring any subtree and potentially maintain better overall balance.
- Adaptive Strategies: Analyze the tree's current structure and choose the direction that minimizes tree height or some other desired metric.

Choosing the Best Strategy:

- The best strategy depends on the specific data distribution, performance requirements, and application context.

- For general-purpose binary tree implementations, left-biased or randomized insertion often offer a good balance between average and worst-case performance.
- For cases where worst-case search performance is critical, right-biased or adaptive strategies might be considered.

Remember:

- Choosing an insertion strategy can impact the efficiency of your tree structure.
- Analyze your specific needs and data characteristics to select the most appropriate approach.
- Consider further research and experimentation to refine your choice for optimal performance.

8. Tree Traversals:

    - Implement preorder, inorder, and postorder traversal algorithms

```
def insert(root, data):
    """Inserts a new node with the given data into a
binary tree."""

    if root is None:
        # If the tree is empty, create a new root
node
        return Node(data)

    if data < root.data:
        # Recursively insert into the left subtree
        root.left = insert(root.left, data)
    else:
        # Recursively insert into the right subtree
        root.right = insert(root.right, data)
```

```
    return root  # Return the updated root node
# Create an initial tree
root = Node(10)
insert(root, 5)
insert(root, 15)

# Tree now looks like:
#       10
#      / \
#    5   15
```

- Visualize the traversals for different tree structures

I'm unable to create visual representations directly, but I can provide clear descriptions and examples to guide you through visualizing tree traversals for different structures:

Consider this tree:

```
     A
    /   \
  B      C
 / \    / \
D   E F   G
```

Preorder Traversal:

- Visit the root first.
- Traverse the left subtree recursively.
- Traverse the right subtree recursively.

Visualization:

- Imagine a path starting at the root, going left as far as possible, then backtracking up and going right at each node.

Traversal sequence: A, B, D, E, C, F, G

Inorder Traversal:

- Traverse the left subtree recursively.
- Visit the root.
- Traverse the right subtree recursively.

Visualization:

- Imagine a path "zigzagging" down the left side of the tree, visiting the root of each subtree, then "zigzagging" back up the right side.

Traversal sequence: D, B, E, A, F, C, G

Postorder Traversal:

- Traverse the left subtree recursively.
- Traverse the right subtree recursively.
- Visit the root last.

Visualization:

- Imagine visiting all the nodes in the subtrees before visiting the root.

Traversal sequence: D, E, B, F, G, C, A

Key Points:

- Preorder is often used for creating a copy of a tree or for evaluating expressions.
- Inorder often used for printing a binary search tree in sorted order.
- Postorder is often used for deleting a tree or for evaluating expressions in reverse Polish notation.

Tips for Visualization:

9. Draw the tree on paper and trace the paths for each traversal.
10. Use online tree visualization tools or libraries to create interactive visualizations.
11. Practice visualizing different tree structures and traversals to solidify your understanding.

- Explain the applications of each traversal method

Here are the common applications of each tree traversal method:

Preorder Traversal:

- Creating a copy of a tree: Visit the root first, then recursively copy the subtrees to preserve the structure.
- Evaluating expressions: For prefix notation (operator before operands), visit the operator first, then evaluate the operands.
- Serializing a tree: Store the tree's structure in a linear format for later reconstruction.
- Creating a hierarchical representation of data: Visit parent nodes before children to establish hierarchy.

Inorder Traversal:

- Printing a binary search tree (BST) in sorted order: In a BST, inorder traversal visits nodes in ascending order.
- Converting a tree to a sorted array or list: Store node values during inorder traversal to create a sorted sequence.
- Finding the k-th smallest element in a BST: Stop traversal after visiting k nodes to get the k-th smallest element.
- Generating infix expressions: Produce a human-readable expression by placing operators between operands.

Postorder Traversal:

- Deleting a tree: Visit children before the root to ensure children are deleted before the parent, avoiding dangling pointers.
- Evaluating expressions in reverse Polish notation (RPN): Postorder aligns with RPN's operand-operand-operator structure.
- Calculating memory space for a tree: Visit children before the root to determine memory requirements for subtrees.
- Creating a bottom-up processing of tree structures: Perform operations on children before the parent.

Key Points:

12. Each traversal method visits all nodes in the tree but in different orders.
13. The choice of traversal depends on the specific problem or operation being performed.
14. Understanding the applications of each traversal is essential for effective tree manipulation in various algorithms and data structures.

15. Binary Search Trees (BSTs):

- **Introduction to the concept of BSTs and their properties (order)**

Binary Search Trees (BSTs): Balancing Order and Search Efficiency

A Binary Search Tree (BST) is a specialized type of binary tree with a crucial property: it maintains order within its structure. This order allows for efficient searching, insertion, and deletion operations.

Key Properties of a BST:

1. Ordered Data: Each node in a BST stores a data value. For any node `N`:
   - All values in its left subtree are less than the value in `N`.
   - All values in its right subtree are greater than or equal to the value in `N`.
2.
3. Binary Tree Structure: Each node can have at most two children: a left child and a right child.

4. Unique Values: No two nodes in a BST can have the same data value.

Benefits of Order:

- Efficient Searching: Due to the ordered structure, you can quickly find a specific value by comparing it to the data at each node as you traverse the tree. This leads to logarithmic time complexity (O(log n)) on average for successful searches.

- Organized Insertion and Deletion: Maintaining order guides where to insert new values and which nodes to remove during deletion, ensuring efficient operations with logarithmic time complexity.

Real-World Applications of BSTs:

- Dictionaries: BSTs form the backbone of many dictionary implementations, allowing for fast lookups of keywords and associated values.
- Auto-complete features: Search suggestions in web browsers or text editors often utilize BSTs for quick retrieval of potential completions.
- Priority queues: Tasks can be prioritized in a BST based on their urgency, ensuring high-priority tasks are processed first.
- Routing algorithms: Some routing algorithms employ BSTs to efficiently find the optimal path between network nodes.

Challenges and Balancing:

- Maintaining Order: Inserting or deleting elements can potentially disrupt the tree's order property. Various techniques, like rotations, are used to balance the tree and restore order efficiently.

- Worst-Case Scenario: Degenerate BSTs, where nodes are inserted in a specific order, result in linear time complexity for operations similar to a linked list. This highlights the importance of balancing techniques.

Further Exploration:

- Explore different self-balancing BSTs like AVL trees and Red-Black trees, which automatically maintain balance during operations.
- Analyze the time and space complexity of BST operations to understand their performance characteristics.
- Implement a BST in your preferred programming language to gain practical experience with handling ordered data structures.

    - Implement search, insertion, and deletion operations in a BST

1. Search Operation:

```
def search(root, data):

    """Searches for a node with the given data in the
BST."""



    if root is None or root.data == data:

        return root  # Found or not found



    if data < root.data:
```

```
        return search(root.left, data)  # Search in the left subtree

    else:

        return search(root.right, data)  # Search in the right subtree
```

2. Insertion Operation:

```
def insert(root, data):

    """Inserts a new node with the given data in the BST."""



    if root is None:

        return Node(data)  # Create a new root node



    if data < root.data:

        root.left = insert(root.left, data)  # Insert in the left subtree

    else:

        root.right = insert(root.right, data)  # Insert in the right subtree



    return root  # Return the updated root node
```

3. Deletion Operation:

```
def delete(root, data):
    """Deletes the node with the given data from the BST."""

    if root is None:
        return root  # Data not found

    if data < root.data:
        root.left = delete(root.left, data)  # Delete from the left subtree
    elif data > root.data:
        root.right = delete(root.right, data)  # Delete from the right subtree
    else:
        # Node to be deleted found
        if root.left is None:
            temp = root.right
            root = None
```

```python
            return temp

        elif root.right is
            None:

                temp = root.left

            root = None

            return temp

        # Node has two children

        temp = find_min_value_node(root.right)

        root.data = temp.data

        root.right = delete(root.right, temp.data)

    return root   # Return the updated root node

def find_min_value_node(node):

    """Finds the node with the minimum value in a BST."""

    current = node

    while current.left is not None:
```

```
            current = current.left

    return current
```

Key Points:

16. All operations use recursion to navigate the tree's structure.
17. Search and insertion have average time complexity of O(log n) for a balanced BST.
18. Deletion can be more complex, potentially requiring tree rebalancing to maintain order.
19. The provided implementation handles basic deletion scenarios.
20. For efficient self-balancing BSTs, explore AVL trees or Red-Black trees.

  - Discuss the time complexity of BST operations (O(log n) for balanced trees)

Here's a discussion of the time complexity of BST operations, focusing on the ideal O(log n) performance for balanced trees:

Time Complexity:

- Search:
  - Best Case: O(1) if the target value is found at the root.
  - Average Case: O(log n) for a balanced BST, as each comparison eliminates half of the remaining nodes.
  - Worst Case: O(n) for a degenerate BST (like a linked list), where every node is visited.

- Insertion:
    - Average Case: O(log n) for a balanced BST, as finding the insertion point involves a similar logarithmic search.
    - Worst Case: O(n) for a degenerate BST, where traversal to a leaf node is required.
- Deletion:
    - Average Case: O(log n) for a balanced BST, similar to search and insertion.
    - Worst Case: O(n) for a degenerate BST, or when rebalancing is needed.

Balanced vs. Unbalanced Trees:

- Balanced BSTs: Maintain a roughly even height between subtrees, ensuring O(log n) performance for most operations.
- Unbalanced BSTs: Can become skewed, leading to worst-case linear time complexity.

Factors Affecting Time Complexity:

- Tree Balance: The primary factor determining efficiency.
- Implementation: Choice of programming language and data structures can impact performance.
- Input Data: The order of insertions and deletions can influence balance.

Achieving Balanced Trees:

- Self-Balancing BSTs: AVL trees and Red-Black trees automatically maintain balance during operations, ensuring O(log n) time complexity even in worst cases.

Key Points:

21. BSTs offer efficient search, insertion, and deletion operations, primarily due to their ordered structure.
22. Balanced BSTs exhibit logarithmic time complexity, making them valuable for managing large datasets.
23. Self-balancing BSTs ensure consistent performance by proactively addressing imbalances.
24. Understanding time complexity is crucial for selecting appropriate data structures and algorithms for different tasks.

25. Lab Exercises:

    - Implement functions to:
        - Calculate the height of a binary tree
        - Count the number of nodes in a binary tree
        - Determine if a tree is a BST
        - Find the lowest common ancestor (LCA) of two nodes
        - Find the k-th smallest element in a BST
        - Perform a level order traversal
26. Challenges:

    - Create a complete binary tree from a given array
    - Construct an expression tree from a postfix expression
    - Serialize and deserialize a binary tree
    - Explore self-balancing trees (AVL trees, Red-Black trees)

Additional Tips:

- Use clear and concise code with meaningful variable names.
- Add comments to explain the logic behind your functions.
- Test your functions thoroughly with various inputs.
- Visualize your trees using diagrams or graphing libraries.
- Explore different tree variations and applications.