Pseudo code for Expression Tree Construction (Postfix):

constructExpressionTree(postfix)

Input: postfix expression as a list of characters or tokens

Output: root of the constructed expression tree

1. Initialize an empty stack `stack`
2. For each character `token` in the postfix expression:
   a. If `token` is an operand (constant or variable):
      i. Create a new node `Node(token)`
      ii. Push `Node(token)` onto `stack`
   b. Else if `token` is an operator (e.g., +, -, *, /):
      i. Pop the top element from `stack` and call it `rightNode`
      ii. Pop the next top element from `stack` and call it `leftNode`
      iii. Create a new node `Node(token)`
      iv. Set `Node(token).left = leftNode`
      v. Set `Node(token).right = rightNode`
      vi. Push `Node(token)` back onto `stack`
3. After the loop, the single element remaining in `stack` is the root of the expression tree.
4. Return the root node

Tree traversal methods

1. **In-order Traversal** (Left, Root, Right)

Algorithm: inOrderTraversal(node)

Input: node (root of the tree or subtree)

Output: None (prints or processes the nodes in in-order)

If node is not NULL:
   a. Call inOrderTraversal(node.left)   // Visit left subtree
   b. Process the node (e.g., print node.value) // Visit root
   c. Call inOrderTraversal(node.right)   // Visit right subtree

2. **Pre-order Traversal** (Root, Left, Right)

Algorithm: preOrderTraversal(node)

Input: node (root of the tree or subtree)

Output: None (prints or processes the nodes in pre-order)

If node is not NULL:
   a. Process the node (e.g., print node.value) // Visit root
   b. Call preOrderTraversal(node.left)   // Visit left subtree
   c. Call preOrderTraversal(node.right) // Visit right subtree

3. **Post-order Traversal** (Left, Right, Root)

Algorithm: postOrderTraversal(node)

Input: node (root of the tree or subtree)

Output: None (prints or processes the nodes in post-order)

If node is not NULL:
   a. Call postOrderTraversal(node.left)   // Visit left subtree
   b. Call postOrderTraversal(node.right) // Visit right subtree
   c. Process the node (e.g., print node.value) // Visit root

Non-Recursive In-order Traversal (Left, Root, Right)
Algorithm: inOrderTraversal(root)
Input: root (root of the tree)
Output: None (prints or processes nodes in in-order)

1. Initialize an empty stack `stack`
2. Set `current` to root

3. While `current` is not NULL or `stack` is not empty:
   a. While `current` is not NULL:
      i. Push `current` onto the `stack`
      ii. Set `current = current.left` // Traverse left subtree

   b. Pop `current` from the `stack`
   c. Process `current` (e.g., print current.value) // Visit root

   d. Set `current = current.right` // Traverse right subtree

Non-Recursive Pre-order Traversal (Root, Left, Right)
Algorithm: preOrderTraversal(root)
Input: root (root of the tree)
Output: None (prints or processes nodes in pre-order)

1. Initialize an empty stack `stack`
2. Push `root` onto the `stack`

3. While `stack` is not empty:
   a. Pop `current` from the `stack`
   b. Process `current` (e.g., print current.value) // Visit root

   c. If `current.right` is not NULL:
      i. Push `current.right` onto the `stack`

   d. If `current.left` is not NULL:
      i. Push `current.left` onto the `stack`

// Note: Push right before left to ensure left is processed first

Non-Recursive Post-order Traversal (Left, Right, Root)
Algorithm: postOrderTraversal(root)
Input: root (root of the tree)
Output: None (prints or processes nodes in post-order)

1. Initialize two empty stacks `stack1` and `stack2`
2. Push `root` onto `stack1`

3. While `stack1` is not empty:
   a. Pop `current` from `stack1` and push it onto `stack2`

   b. If `current.left` is not NULL:

```
      i. Push `current.left` onto `stack1`

   c. If `current.right` is not NULL:
      i. Push `current.right` onto `stack1`

4. While `stack2` is not empty:
   a. Pop `current` from `stack2`
   b. Process `current` (e.g., print current.value)  // Visit root
```

BST ADT:

A **Binary Search Tree (BST)** ADT supports the following operations:

1. Basic Structure

- Each node in the BST contains:
  - **Key/Value**: The value stored at the node.
  - **Left**: A pointer/reference to the left child node (NULL if no left child).
  - **Right**: A pointer/reference to the right child node (NULL if no right child).

2. Operations of BST ADT

1. **Create (Initialize an Empty BST)**:
   - Creates an empty binary search tree.
   - This operation initializes the root to NULL or empty.

   Pseudocode:

   ```
   text
   Copy code
   createBST()
       root = NULL
   ```

2. **Insert (x)**:
   - Inserts a new key `x` into the BST.
   - The insertion follows the ordering property:
     - Traverse the tree starting from the root.
     - If `x` is less than the current node's value, move to the left subtree.
     - If `x` is greater, move to the right subtree.
     - Insert at the position where the left or right child is NULL.

   Pseudocode:

   ```
   text
   Copy code
   insert(node, x)
       if node == NULL
           node = createNewNode(x)
       else if x < node.value
           node.left = insert(node.left, x)
       else
           node.right = insert(node.right, x)
       return node
   ```

3. **Search (x)**:
   - Searches for the key `x` in the BST.
   - Starts at the root and traverses the tree based on the BST property:
     - If `x` is less than the current node's value, move to the left subtree.
     - If `x` is greater, move to the right subtree.
     - If `x` matches the current node's value, return the node or TRUE (if only checking existence).

Pseudocode:

```
text
Copy code
search(node, x)
    if node == NULL or node.value == x
        return node  // Return node or TRUE/FALSE if only checking
existence
    else if x < node.value
        return search(node.left, x)
    else
        return search(node.right, x)
```

4. **Delete (x)**:
   - Deletes a node with key `x` from the BST.
   - The deletion operation has three cases:
     1. **Node with no children**: Simply remove the node.
     2. **Node with one child**: Replace the node with its child.
     3. **Node with two children**: Find the in-order successor (smallest node in the right subtree) or the in-order predecessor (largest node in the left subtree), replace the node's value with the successor/predecessor, and then delete the successor/predecessor.

   Pseudocode:

```
text
Copy code
delete(node, x)
    if node == NULL
        return NULL

    if x < node.value
        node.left = delete(node.left, x)
    else if x > node.value
        node.right = delete(node.right, x)
    else  // Node found
        if node.left == NULL
            return node.right
        else if node.right == NULL
            return node.left

        // Node with two children
        successor = findMin(node.right)  // Find in-order successor
        node.value = successor.value
        node.right = delete(node.right, successor.value)

    return node
```

5. **Find Minimum**:
   - Returns the node with the smallest key in the tree.
   - In a BST, the smallest key is found by following the left child pointers from the root until reaching a node with no left child.

   Pseudocode:

```
text
```

```
Copy code
findMin(node)
    while node.left != NULL
        node = node.left
    return node
```

6. **Find Maximum**:
   - Returns the node with the largest key in the tree.
   - In a BST, the largest key is found by following the right child pointers from the root until reaching a node with no right child.

   Pseudocode:

```
text
Copy code
findMax(node)
    while node.right != NULL
        node = node.right
    return node
```

7. **In-order Traversal**:
   - Traverses the tree in **in-order** (left, root, right) to get the elements in sorted order.

   Pseudocode:

```
text
Copy code
inOrderTraversal(node)
    if node != NULL
        inOrderTraversal(node.left)
        process(node.value)  // Example: print(node.value)
        inOrderTraversal(node.right)
```

8. **Pre-order Traversal**:
   - Traverses the tree in **pre-order** (root, left, right).

   Pseudocode:

```
text
Copy code
preOrderTraversal(node)
    if node != NULL
        process(node.value)  // Example: print(node.value)
        preOrderTraversal(node.left)
        preOrderTraversal(node.right)
```

9. **Post-order Traversal**:
   - Traverses the tree in **post-order** (left, right, root).

   Pseudocode:

```
text
Copy code
postOrderTraversal(node)
```

```
    if node != NULL
        postOrderTraversal(node.left)
        postOrderTraversal(node.right)
        process(node.value)  // Example: print(node.value)
```

10. **Height of the Tree**:
    - The height of a BST is the number of edges on the longest path from the root to a leaf.

    Pseudocode:

```
text
Copy code
height(node)
    if node == NULL
        return -1  // or 0, depending on definition (leaf height = 0
or -1)
    else
        leftHeight = height(node.left)
        rightHeight = height(node.right)
        return max(leftHeight, rightHeight) + 1
```

Summary of BST ADT:

Operation	Time Complexity (Average Case)	Time Complexity (Worst Case)
Insert	O(log n)	O(n)
Search	O(log n)	O(n)
Delete	O(log n)	O(n)
Find Min/Max	O(log n)	O(n)
In-order Traversal	O(n)	O(n)
Pre-order Traversal	O(n)	O(n)
Post-order Traversal	O(n)	O(n)
Height	O(log n)	O(n)

These time complexities assume the tree is balanced. In the worst case, when the tree becomes a linked list (i.e., highly unbalanced), the time complexities can degrade to `O(n)`.

Threaded Binary Tree (TBT)

The **Threaded Binary Tree (TBT)** is a variation of the binary tree data structure that addresses the issue of wasted space in binary trees due to null pointers. In a standard binary tree, many of the pointers in the nodes are null, especially in the leaf nodes, since those nodes do not have left or right children. A threaded binary tree makes use of these null pointers to store additional information, improving traversal efficiency.

Key Significance of Threaded Binary Trees:

1. **Efficient In-order Traversal Without Stack or Recursion:** In a normal binary tree, in-order traversal typically requires either recursion or a stack to keep track of the nodes, which can add overhead in terms of both memory and processing. In a threaded binary tree, the null pointers are replaced with "threads" that point to the in-order predecessor or successor, allowing the tree to be traversed efficiently without the need for extra memory or stack.
2. **Space Optimization:** Threaded binary trees help reduce the amount of memory wasted on null pointers. In a complete binary tree, almost half of the pointers are null. By using these null pointers as threads, we can store useful information, reducing memory overhead.
3. **Improved Search and Insertion Operations:** With the presence of threads, navigating between nodes (for searching, inserting, or deleting) becomes quicker since we can directly access the in-order predecessor or successor without recalculating the path from the root or using extra memory structures.
4. **Faster Access to Parent Nodes:** In some implementations, the threading can be extended to include a link to the parent node. This allows faster upward traversal, which is useful in various operations, such as balancing the tree or performing reverse traversals.
5. **Simplified Tree Traversals:** Traversing the tree in an in-order fashion is made simpler, as it can be done iteratively without a stack or recursion. This is particularly useful for systems with limited memory or environments where recursion is not efficient.

Types of Threaded Binary Trees:

1. **Single Threaded Binary Tree:** Only one of the pointers (either left or right) is replaced with a thread, usually pointing to the in-order predecessor (for the left pointer) or successor (for the right pointer).
2. **Double Threaded Binary Tree:** Both the left and right pointers can be threaded, where the left pointer points to the in-order predecessor and the right pointer points to the in-order successor.

Applications:

- Threaded binary trees are useful in applications that require frequent traversal of the tree without modifying its structure.
- They are used in environments with memory constraints, where the overhead of recursion and additional data structures (like stacks) needs to be minimized.

Pseudo Code for Threaded Binary Tree Construction

1. **Node Structure:**

```
Node {
    int data
    Node *left, *right
    bool isLeftThread, isRightThread
}
```

2. **In-order Threaded Binary Tree Construction:**

```
createInOrderThreadedTree(root):
    prev = null    # Initialize a pointer to store the previous node

    inorderThreading(root, prev)call a recursive function to thread the
tree


inorderThreading(root, prev):
    if root is not null:
        # Step 1: Thread the left subtree
        inorderThreading(root.left, prev)

        # Step 2: Handle the current node's left pointer
        if root.left is null:
            root.left = prev  # Set left thread to the predecessor
            root.isLeftThread = true

        # Step 3: Handle the previous node's right pointer
        if prev is not null and prev.right is null:
            prev.right = root  # Set right thread of prev to the current
node
            prev.isRightThread = true

        # Update prev to the current node
        prev = root

        # Step 4: Thread the right subtree
        inorderThreading(root.right, prev)
```

In this algorithm:

- The `inorderThreading` function threads the tree by replacing null pointers with links to in-order predecessors and successors.
- The `prev` variable keeps track of the previously visited node during the in-order traversal to update the threads.

Pseudo Code for In-order Traversal of a Threaded Binary Tree

In an **in-order threaded binary tree**, the traversal can be done iteratively by following the threads, without the need for a stack or recursion.

```
inorderTraversal(root):
    # Step 1: Start at the leftmost node
    current = root
```

```
    while current is not null and current.isLeftThread is false:
        current = current.left

    # Step 2: Traverse the threaded tree
    while current is not null:
        # Visit the current node
        print(current.data)

        # Step 3: If the right pointer is a thread, follow it
        if current.isRightThread:
            current = current.right
        else:
            # Otherwise, go to the leftmost node in the right subtree
            current = current.right
            while current is not null and current.isLeftThread is false:
                current = current.left
```

Pseudo Code for Pre-order Traversal of a Threaded Binary Tree

Pre-order traversal in a threaded binary tree also benefits from threads, but it requires a slightly different approach:

```
preorderTraversal(root):
    current = root

    # Step 1: Traverse the tree using threads
    while current is not null:
        # Step 2: Visit the current node
        print(current.data)

        # Step 3: If there is a left child, move to the left subtree
        if current.isLeftThread is false:
            current = current.left
        else:
            # Otherwise, follow the thread to the right
            current = current.right
```

Explain the difference between a normal binary tree and a threaded binary tree.

Define the two types of threads used in threaded binary trees. What are in-order predecessor and successor threads?

Given the numbers `[25, 15, 50, 10, 22, 35, 70, 4, 12, 18, 24, 31, 44, 66, 90]`, build the corresponding Binary Search Tree. Perform the deletion of nodes `22`, `50`, and `10` in sequence. After each deletion, draw the tree and explain how the structure changes.