Data Structures and Algorithms (Spring2024)

Week09/10/11 (2/2/8/9/9/15-May-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Understanding ADTs and their Classifications

Question: Why are arrays, dynamic arrays, and lists not considered Abstract Data Types (ADTs) like stacks and queues, and how do trees and graphs qualify as ADTs, particularly as non-linear ADTs? Additionally, where does hashing stand in the context of ADTs?

Answer: The classification of data structures as Abstract Data Types (ADTs) revolves around the abstraction of their operations rather than their specific implementation details. An ADT is defined more by the operations it supports and less by how these operations are implemented.

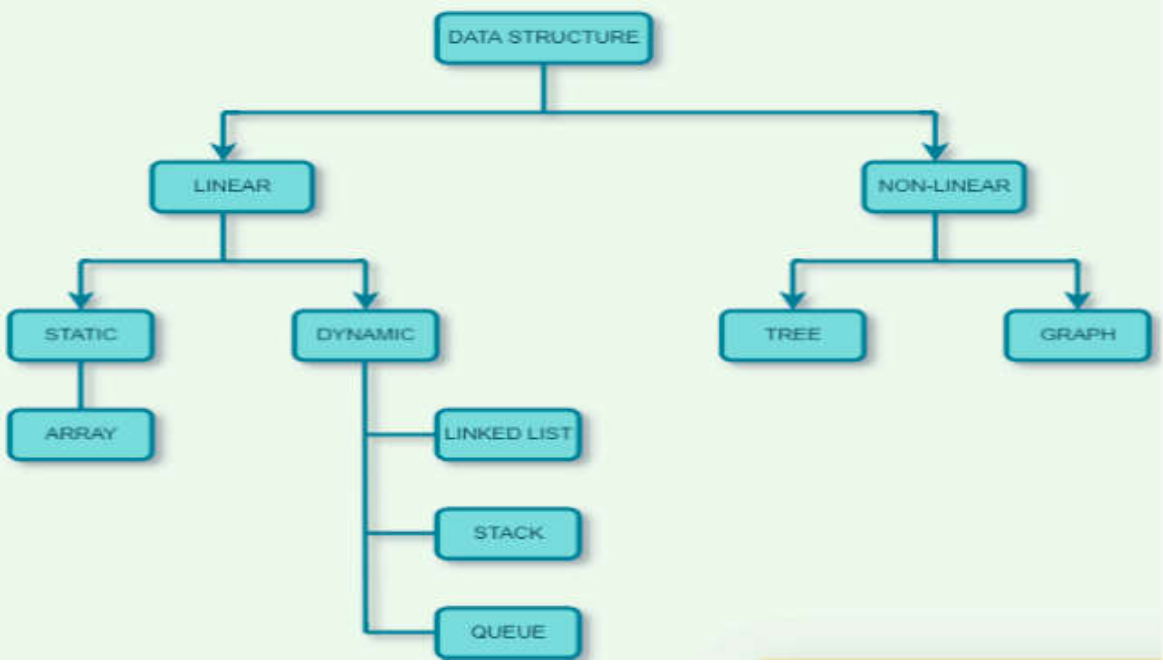
1. **Arrays, Dynamic Arrays, and Lists:**
    - **Not Typically ADTs:** Arrays, dynamic arrays, and lists provide a way to store data in a linear, sequential manner, and they expose specific elements' positions and indexing mechanisms. While they are crucial data structures, they are often not considered ADTs because they deal more with the implementation of data storage—how data is organized in memory—rather than abstracting the operations that can be performed on the data.
    - **Operational Transparency:** These structures lack a level of operational abstraction that hides the implementation details from the user. For example, when you use an array, you often need to manage indices directly.

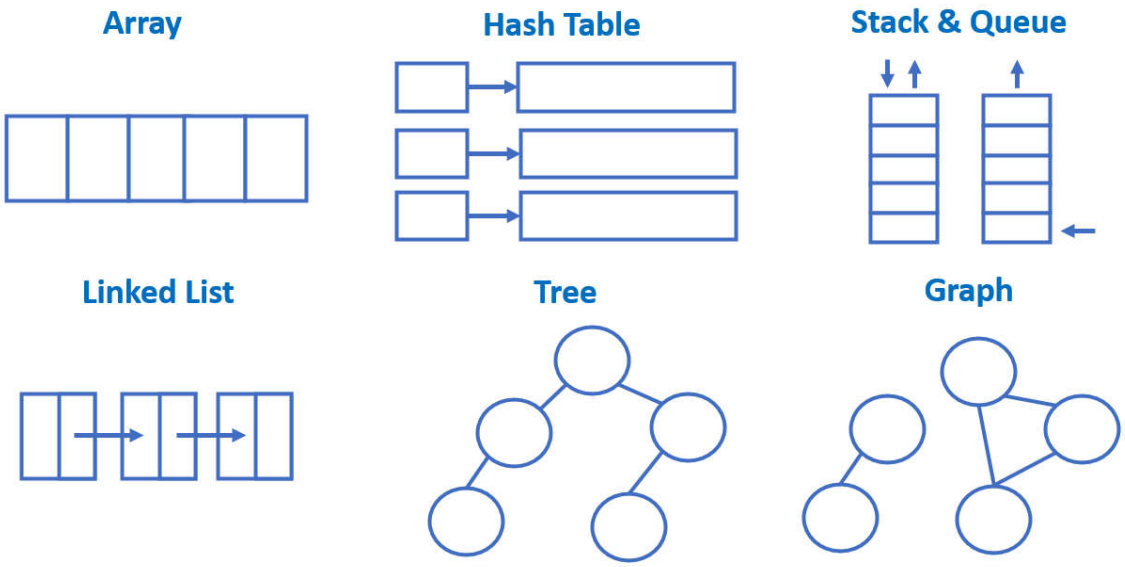
2. **Stacks and Queues:**
    - **Definite ADTs:** Unlike arrays and lists, stacks and queues are considered ADTs because they provide a clear set of operations (like push, pop for stacks and enqueue, dequeue for queues) without exposing how these operations are implemented internally. This abstraction is key to defining ADTs.
    - **Operational Abstraction:** Users of stacks and queues do not need to know about the underlying structure, whether it's an array or a linked list, which supports the ADT philosophy.
3. **Trees and Graphs:**
    - **Non-linear ADTs:** Trees and graphs are also ADTs. They provide a higher level of abstraction for handling hierarchical (trees) or networked (graphs) data. They are classified as non-linear because they do not store data in a sequential manner but in a hierarchical or interconnected way that can branch in various directions.
    - **Abstract Operations:** Operations on trees (like search, insert, delete) and graphs (like add vertex, add edge, find path) are abstracted from the underlying representation, which can vary widely (e.g., nodes and pointers, adjacency lists, adjacency matrices).

4. **Hashing:**

   - **Role in ADTs:** Hashing itself is a technique rather than a data structure. It transforms a given key into an index for an array-based data structure, typically used in hash tables.
   - **Hash Tables as ADTs:** Hash tables, which use hashing, are considered ADTs because they provide operations like insertion, deletion, and retrieval based on keys, abstracting away the details of how hashing or collision resolution is implemented.

In summary, the distinction between what is considered an ADT and what is not often hinges on the level of abstraction in handling operations and hiding implementation details from the end user, a crucial principle in data structure design. Trees and graphs expand this concept into more complex, non-linear domains, while hashing supports the mechanism behind operations in hash tables, exemplifying the abstraction of data retrieval processes.





Comprehensive Definition, Comparison, and Applications of Graphs and Trees

Question: Can we comprehensively define graphs and trees, compare and contrast them, and highlight their various applications?

Answer: Graphs and trees are fundamental non-linear data structures used to represent various forms of data relationships and structures in computing. Understanding their definitions, differences, and applications provides a foundation for leveraging them in appropriate contexts.

Definitions:

1. **Graphs:**
    - A graph is a collection of nodes (or vertices) and edges connecting pairs or groups of nodes. Graphs can be directed (where edges have direction) or undirected (where edges do not have direction). They are highly versatile and can represent complex relationships like networks.
2. **Trees:**
    - A tree is a special form of graph that is hierarchical and does not contain cycles. It consists of nodes connected by edges, with one node designated as the root. Each node (except the root) is connected by exactly one incoming edge, creating a parent-child relationship.

Comparison:

1. **Structure:**
    - **Graphs** can have cycles, multiple paths between nodes, and can be either directed or undirected.
    - **Trees** are acyclic (no cycles), have exactly one path between any two nodes, and inherently have a directional flow from the root to leaves.
2. **Complexity:**
    - **Graphs** are generally more complex due to their versatility and the potential for representing more complex relationships. Handling graphs often requires managing additional properties like direction and weight on edges.
    - **Trees** offer simpler dynamics due to their hierarchical structure, making them easier to manage and traverse.
3. **Root Node:**
    - **Graphs** do not necessarily have a root node.
    - **Trees** always have a root node from which all nodes are accessible.

Applications:

1. **Graphs:**
    - **Networking:** Graphs are used to represent and manage networks like social networks, communication infrastructures, and internet routing.

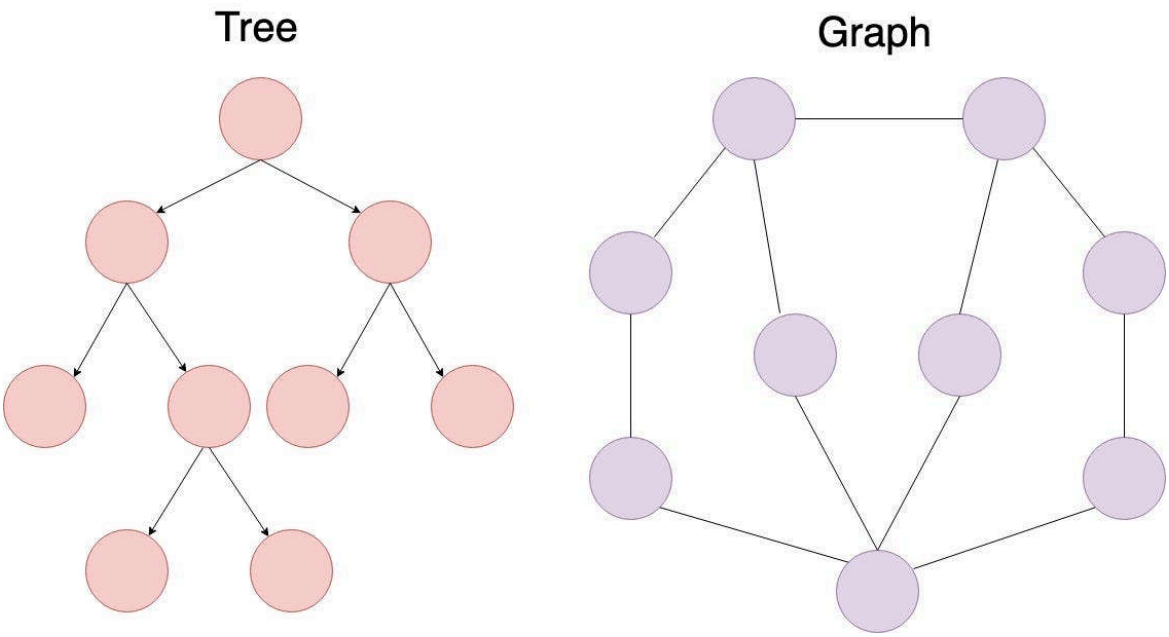
- **Urban Planning:** Used in city and transportation planning, especially in modeling traffic flows and public transport networks.
- **Problem Solving:** Essential in algorithms like searching (DFS, BFS), shortest path finding (Dijkstra's, Bellman-Ford), and network flow problems.
- **Games and Puzzles:** Used to model different states in games, such as puzzles like Rubik's Cube or strategic games like chess.

2. **Trees:**

- **File Systems:** Trees are used to represent and manage hierarchical structures in file systems.
- **Databases:** Binary Search Trees, AVL Trees, and B-trees are used in databases and file storage systems to manage large data sets efficiently.
- **Decision Processes:** Decision trees are a popular method in decision analysis, helping to visualize and strategically analyze decisions.
- **UI Organization:** Used in managing and displaying hierarchical data in user interfaces, like menu structures.

Conclusion:

While both graphs and trees are used to represent relationships and structures, their applications differ based on their structural properties. Trees provide a more straightforward, hierarchical structure ideal for systems that require clear, single-pathed decision hierarchies. In contrast, graphs are suitable for complex, interconnected systems where relationships are not hierarchically organized, allowing for a more flexible representation of connections.



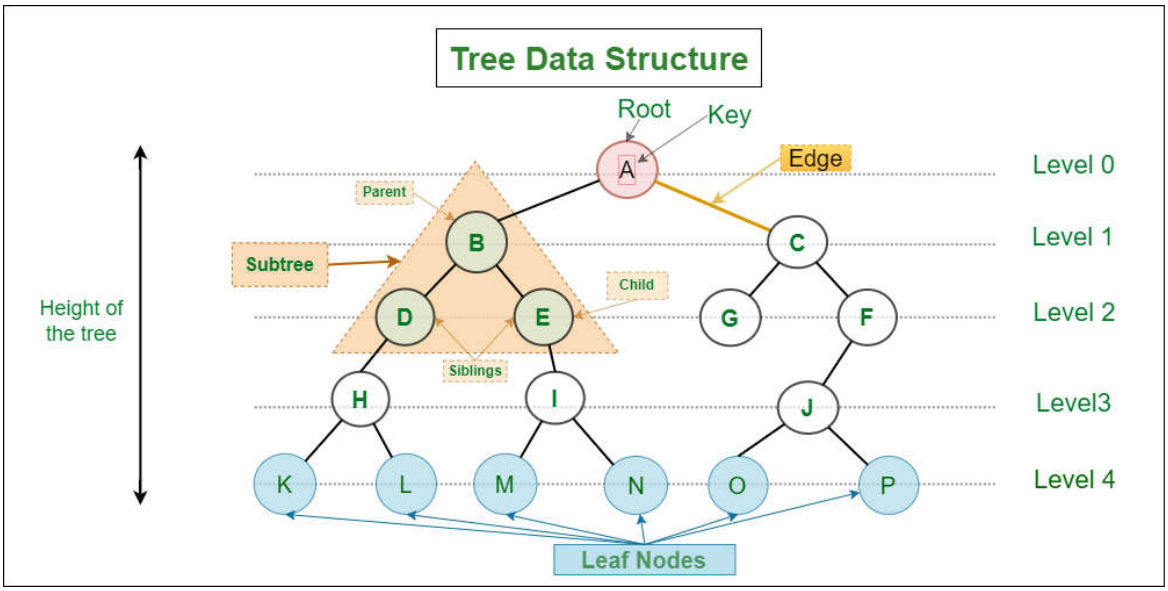
Introduction to Trees and Terminology

Objective: To provide a foundational understanding of tree data structures, including basic terminology and concepts.

Definition of a Tree

A **tree** is a nonlinear data structure that simulates a hierarchical tree structure, with a set of linked nodes. The basic characteristics of a tree include:

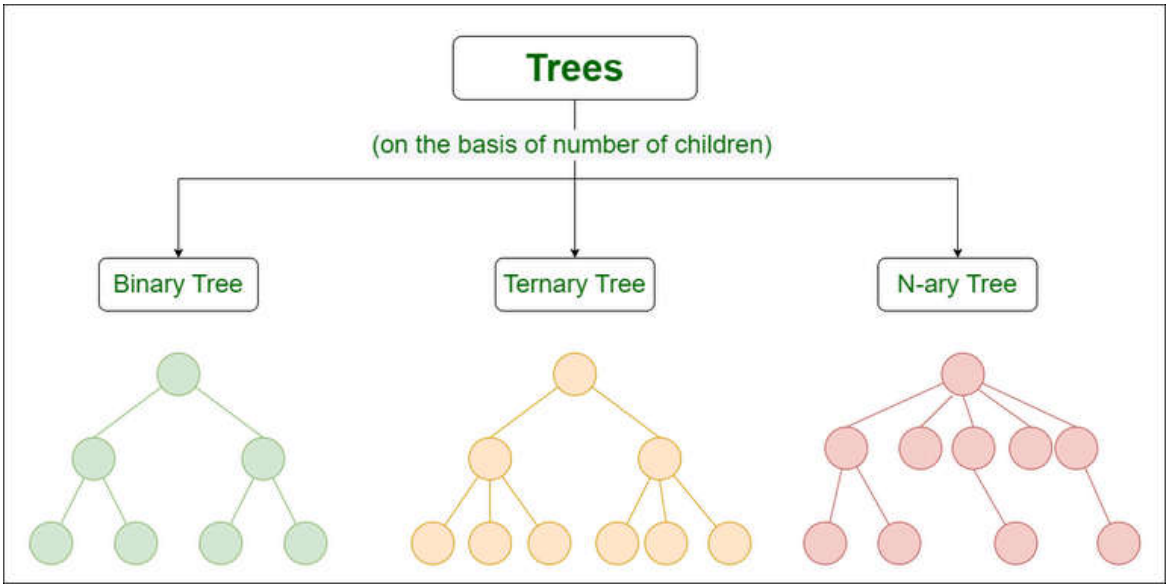
- **Nodes:** Each element in a tree is called a node.
- **Root:** The top node in a tree, from which all nodes descend.
- **Edge:** The connection between one node and another.
- **Children:** Nodes linked to and descended from a parent node.
- **Parent:** The converse notion of a child.
- **Leaf (or terminal node):** A node that does not have children.
- **Subtree:** A tree entirely contained within another tree.
- **Path:** A sequence of nodes and edges connecting a node with a descendant.
- **Level:** The level of a node is defined by 1 + (the number of connections between the node and the root). The root is at level 1.
- **Height of Node:** The height of a node is the number of edges on the longest path from the node down to a leaf. A leaf node will have a height of 0.
- **Depth of Node:** The depth of a node is the number of edges from the node to the tree's root node.
- **Height of Tree:** The height of the tree is the height of the root node.

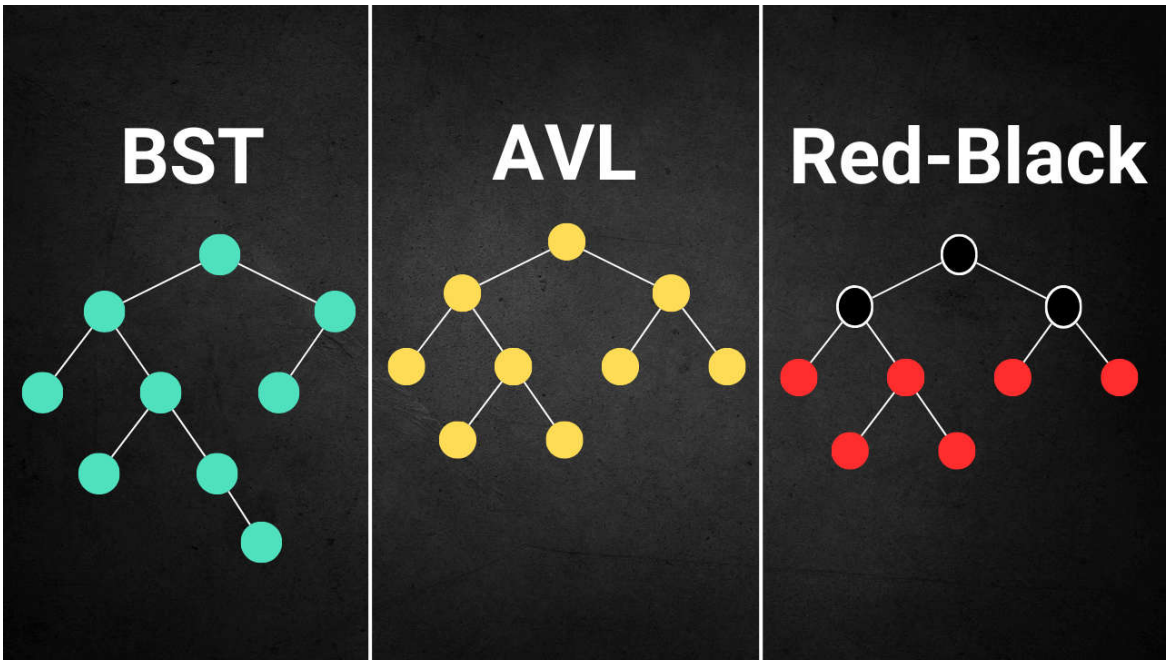


Types of Trees

- **Binary Tree:** A tree where each node has up to two children, often referred to as the left and right children.
- **Binary Search Tree (BST):** A binary tree in which each node has a comparable key (and an associated value) and satisfies the constraint that the key in any node is larger than the keys in all nodes in its left subtree and smaller than the keys in all nodes in its right subtree.

- **Balanced Tree:** A tree in which no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to maintain the tree balanced.
- **AVL Tree:** A self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- **Red-Black Tree:** A type of self-balancing binary search tree where each node has an extra bit for denoting the color of the node, either red or black. A balanced path from the root to the farthest leaf is no more than twice as long as the shortest path from the root to the nearest leaf.





Basic Operations on Trees

- **Traversal:** The process of visiting all nodes in some order. Common types of traversal include:
  - **In-order Traversal (LNR):** Traverse the left subtree, visit the root, traverse the right subtree.
  - **Pre-order Traversal (NLR):** Visit the root, traverse the left subtree, traverse the right subtree.

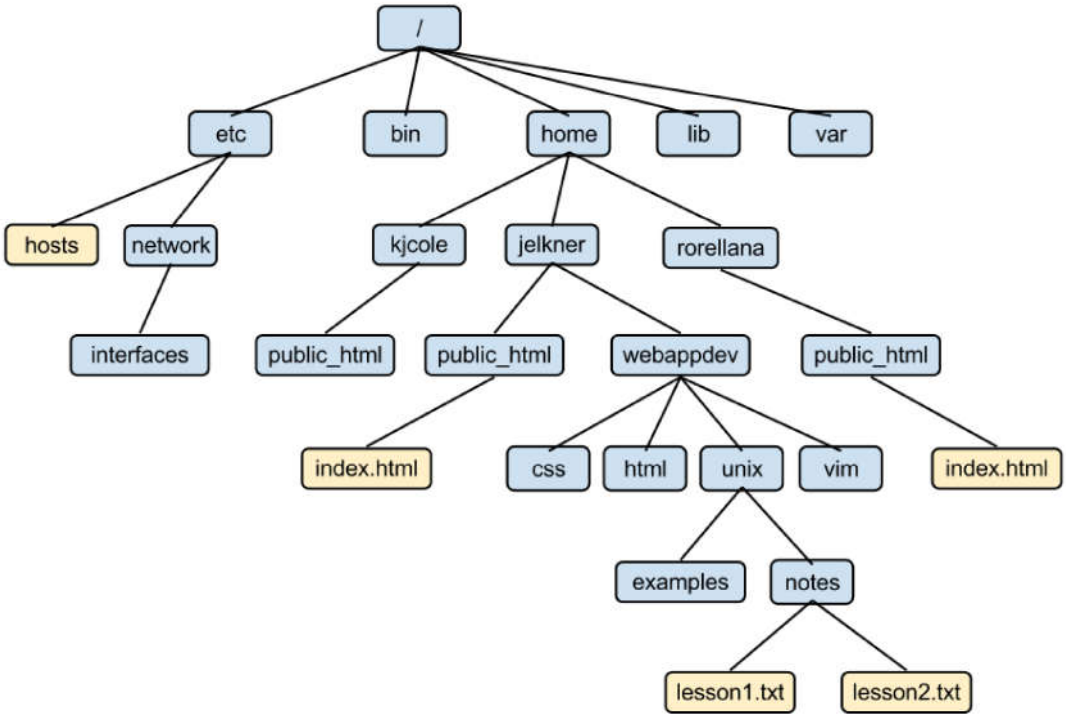
    - **Post-order Traversal (LRN):** Traverse the left subtree, traverse the right subtree, visit the root.
    - **Level-order Traversal:** Visit the nodes on each level from left to right or right to left.
- **Insertion:** Adding a node to the tree in a location that maintains the properties of the tree.
- **Deletion:** Removing a node from the tree while maintaining the properties of the tree.
- **Searching:** Finding a node within a tree based on a condition or value.

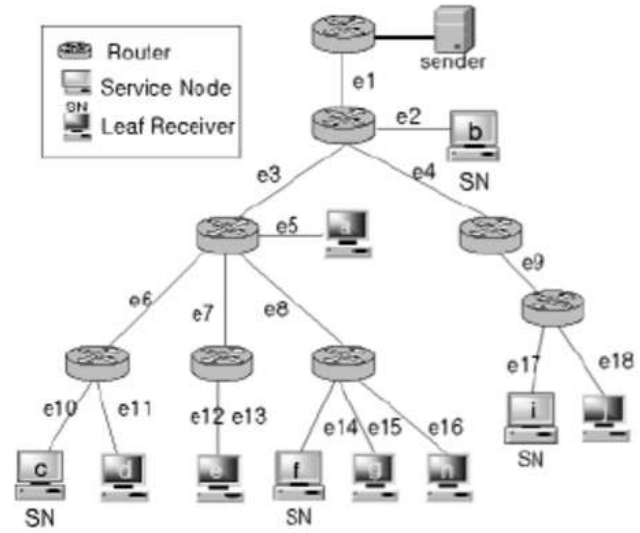
Applications of Trees

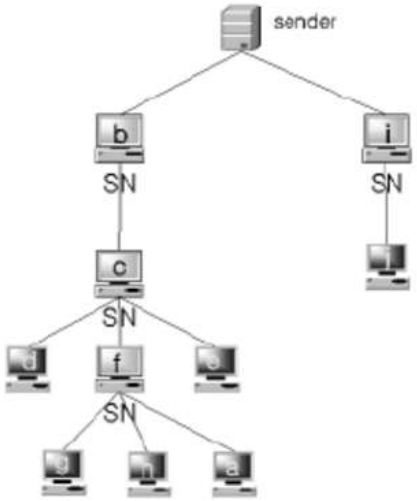
Trees are used in various applications such as:

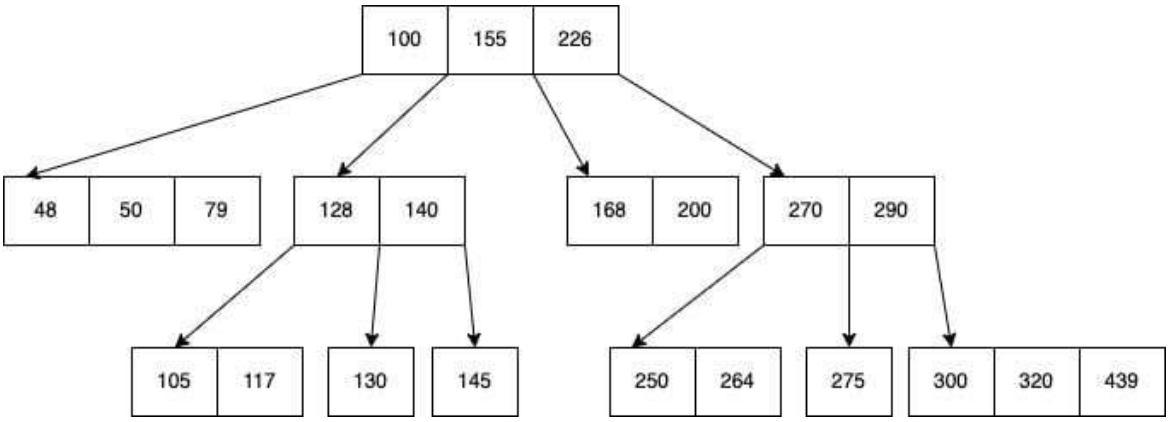
- Organizing data in hierarchical form (e.g., file systems).
- Facilitating fast search, insert, and delete operations (e.g., binary search trees).
- Implementing priority queues (e.g., binary heaps).
- Supporting efficient form of data and network routing in databases and operating systems.

This introduction provides the basic building blocks for understanding trees, their properties, and operations, setting the stage for more detailed discussions on specific types of trees and their applications in subsequent lectures.









Demonstrating Tree Traversals

Objective: To demonstrate the four primary tree traversals (in-order, pre-order, post-order, and level-order) on a complex tree structure and discuss their importance and applications.

Example Tree Structure

For our demonstration, consider the following binary tree:

```
      10
     /  \
    5    20
   / \   / \
  3   7 15  30
     /     /
    6     25
```

Tree Traversals Demonstrations

1. **In-order Traversal (LNR)**

   - **Process:** Traverse the left subtree, visit the root, traverse the right subtree.
   - **Traversal:** 3, 5, 6, 7, 10, 15, 20, 25, 30

- **Applications:** In-order traversal is particularly useful in binary search trees (BSTs) where this traversal results in nodes being visited in their non-decreasing order. It is commonly used for sorting and retrieving sorted data.

2. **Pre-order Traversal (NLR)**

   - **Process:** Visit the root, traverse the left subtree, traverse the right subtree.
   - **Traversal:** 10, 5, 3, 7, 6, 20, 15, 30, 25
   - **Applications:** Pre-order traversal is useful for creating a copy of the tree or when saving the tree structure to a file or another storage system in a way that allows it to be reconstructed later. It reflects the way a tree is constructed top-down.

3. **Post-order Traversal (LRN)**

   - **Process:** Traverse the left subtree, traverse the right subtree, visit the root.
   - **Traversal:** 3, 6, 7, 5, 15, 25, 30, 20, 10
   - **Applications:** Post-order traversal is useful for deleting or freeing nodes and space of the tree from the bottom up. It is also used in mathematical expressions stored in trees (expression trees) to evaluate the expression.

4. **Level-order Traversal**

   - **Process:** Visit the nodes on each level from left to right.
   - **Traversal:** 10, 5, 20, 3, 7, 15, 30, 6, 25
   - **Applications:** Level-order traversal is particularly useful in breadth-first search applications, such as finding the shortest path in unweighted graphs, or when a breadth-first view of the tree is required, for example in networking applications and hierarchical level data representation.

Importance of Traversal Methods

Tree traversals are fundamental operations that enable the processing of tree data structures in various ways, depending on the specific application:

- **Sorting and retrieval:** In-order traversal ensures elements are processed in their natural order, critical for sorted data access.
- **Structure reconstruction:** Pre-order traversal captures the structure of the tree, making it ideal for serialize/deserialize operations.
- **Resource deallocation:** Post-order traversal is suited for safely deallocating resources used by the tree.
- **Level-wise processing:** Level-order traversal is essential for scenarios where actions need to be executed level by level, such as in certain algorithmic or networking contexts where a breadth-first approach is optimal.

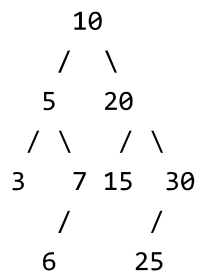
Understanding these traversal techniques is crucial for leveraging the full potential of trees in computer science and related fields.

Operations on Trees - Insertion, Deletion, and Searching

Objective: To demonstrate how to perform insertion, deletion, and searching operations on a **binary search tree (BST)** using a specific example.

Example Tree Structure

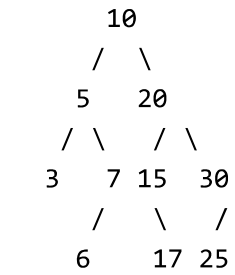
Let's continue with the binary search tree used in the previous examples:



Operations Demonstrations

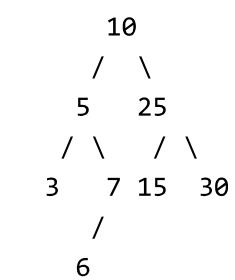
1. **Insertion**

- **Process:** To insert a value, start from the root and compare the value to be inserted with the value of the current node. Move left if it's less, right if it's more. Insert when a leaf node is reached.
- **Example Insertion of '17':**
  - Start at 10; 17 > 10, go right.
  - Compare with 20; 17 < 20, go left.
  - 15 already has no right child; insert 17 there.
- **Resulting Tree:**



2. **Deletion**

- **Process:** To delete a node, find the node, then if:
  - **Node has no children:** Simply remove the node.
  - **Node has one child:** Replace the node with its child.
  - **Node has two children:** Use the in-order successor (smallest in the right subtree) or predecessor (largest in the left subtree) to replace the node, then delete the successor/predecessor.
- **Example Deletion of '20':**
  - Find 20. It has two children.
  - The in-order successor is 25. Replace 20 with 25.
  - Delete 25 from its previous position.
- **Resulting Tree:**



3. **Searching**

   - **Process:** Start from the root and compare the search value with the current node's value. Go left if the search value is less, right if more. Repeat until the value is found or a leaf is reached.
   - **Example Searching for '7':**
     - Start at 10; 7 < 10, go left.
     - Compare with 5; 7 > 5, go right.
     - 7 is found.
   - **Search Outcome:** The value '7' is found in the tree.

Importance of These Operations

- **Insertion** is crucial for building the tree and adding new values while maintaining the BST property.
- **Deletion** is necessary for removing nodes, whether to manage the data stored or to maintain tree health and balance.
- **Searching** is fundamental to retrieve values and is pivotal in operations like lookup, which is extensively used in database systems and other applications where data retrieval is frequent.

These operations ensure that the tree remains an efficient and dynamic data structure capable of adapting to various data manipulations and queries. This demonstration not only shows how these operations are performed but also highlights their practical importance in real-world applications.

In-depth Applications of Trees

Objective: To explore various practical applications of trees in depth, demonstrating their significance in different domains and systems.

1. File Systems

- **Overview:** Trees are fundamental in representing file systems. Each file or directory is a node, with directories containing files or other directories as children.
- **Applied Aspect:**
  - **Hierarchical Structure Management:** Trees facilitate efficient management of the hierarchical structure of directories and files. The tree's root represents the root directory, and branches represent subdirectories.
  - **Path Resolution:** Traversal algorithms allow for efficient path resolution and searching within the file system, making operations like opening a file or listing a directory efficient.
  - **Permissions and Metadata:** Each node (file or directory) can hold metadata, including permissions, which can be efficiently updated or checked using tree traversals.

2. Database Indexing

- **Overview:** Trees, especially binary search trees, AVL trees, and B-trees, are extensively used in database indexing to speed up data retrieval.

- **Applied Aspect:**
  - **Quick Access:** Trees provide logarithmic time complexity for search, insert, and delete operations, crucial for databases with large volumes of data.
  - **Balancing Large Datasets:** AVL trees and B-trees automatically balance themselves to maintain low height, ensuring the tree operations remain efficient even as data grows.
  - **Range Queries:** Trees like segment trees and interval trees support efficient range queries, which are essential for querying continuous chunks of data.

3. Networking Algorithms

- **Overview:** Trees are used in networking, particularly in routing and spanning tree algorithms.
- **Applied Aspect:**
  - **Routing Protocols:** Algorithms like Shortest Path First (used in OSPF) utilize trees to calculate the most efficient routing paths.
  - **Network Redundancy:** Spanning trees are critical in preventing loops in networks, especially in bridged networks. The spanning tree protocol ensures a loop-free topology for Ethernet networks.

4. Artificial Intelligence and Machine Learning

- **Overview:** Decision trees are used in AI for building classification and regression models.
- **Applied Aspect:**
  - **Model Training:** Trees make decisions by learning decision boundaries from data. They are non-parametric, meaning they don't make assumptions about the space distribution and the form of the function that models the predictions.
  - **Handling Both Numerical and Categorical Data:** Trees easily handle features that are categorical or numerical, often without needing preprocessing that other algorithms might require.
  - **Feature Importance:** Trees provide insights into which features are most effective at predicting the outcome, aiding in feature selection.

5. Game Theory

- **Overview:** Trees represent possible moves in games like chess or poker, helping in the computation of optimal strategies.
- **Applied Aspect:**
  - **Minimax Algorithm:** Trees are used to simulate all possible moves in adversarial games, with the minimax algorithm used to find the best move by minimizing the possible loss in a worst-case scenario.
  - **Efficient Pruning Techniques:** Alpha-beta pruning is used to eliminate branches in the tree that cannot possibly affect the final decision, significantly speeding up the computation.

6. GUI Applications

- **Overview:** Many graphical user interfaces (GUIs) use trees to manage the hierarchy of elements and render them accordingly.
- **Applied Aspect:**
  - **Widget Hierarchy:** In frameworks like SwiftUI or Android's UI toolkit, the UI components are managed in a tree structure where the containment hierarchy dictates the rendering process.
  - **Event Propagation:** Events like clicks or touches are propagated through the tree, from parent to child or vice versa, allowing for intuitive event handling and delegation.

These examples underscore the versatility and power of trees across a broad spectrum of applications. By abstracting complex systems into manageable hierarchical models, trees enable efficient operations, deep analytics, and intuitive structuring, proving essential in both

Detailed Discussion on Binary Trees

Objective: To provide an in-depth understanding of binary trees, covering essential terminology, characteristics, and variations.

Definition of a Binary Tree

A **binary tree** is a tree data structure in which each node has at most two children, referred to as the left child and the right child. This structure creates a hierarchical organization of nodes, ideal for various computing problems.

Essential Terminology and Features

1. **Node:** The basic unit of a binary tree, which contains storage for data and two pointers or references to its left and right children.
2. **Root:** The top node of the tree from which all other nodes descend. The root node does not have a parent.
3. **Leaf (or Terminal Node):** Nodes that do not have any children. They are the endpoints of the tree.
4. **Subtree:** A subtree is any node in a tree along with its descendants. A binary tree can be split into left and right subtrees.
5. **Depth of a Node:** The number of edges from the node to the tree's root. The root node has a depth of zero.

6. **Height of a Node:** The number of edges on the longest path from the node to a leaf. A leaf node will have a height of zero.
7. **Height of the Tree:** The height of the root node, which is the longest path from the root to any leaf.
8. **Level:** The set of nodes at a particular depth. For example, all nodes with the same depth are said to be on the same level.
9. **Full Binary Tree:** Every node other than the leaves has two children.
10. **Complete Binary Tree:** All levels, except possibly the last, are completely filled, and all nodes are as left as possible.
11. **Balanced Binary Tree:** A tree where the height of the left and right subtrees of any node differ by no more than one.

Types of Binary Trees

1. **Binary Search Tree (BST):** A binary tree where for each node, the values of all the nodes in the left subtree are lesser, and the values of all the nodes in the right subtree are greater.
2. **AVL Tree:** A self-balancing binary search tree where the heights of two child subtrees of any node differ by no more than one.
3. **Red-Black Tree:** Another type of self-balancing binary search tree, where each node stores an extra bit for denoting the color of the node, either red or black, to ensure the tree remains balanced during insertions and deletions.
4. **Segment Tree:** Used for storing intervals or segments, and optimized to find which of the stored segments contain a given point.
5. **Trie (Prefix Tree):** A special type of tree used to store associative data structures. A common application of tries is storing a predictive text or autocomplete dictionary.

Applications of Binary Trees

- **Hierarchical Data Representation:** Trees are perfect for representing hierarchical data, such as file systems or organizational structures.
- **Database Indexing:** Binary search trees are widely used in database indexing to allow quick retrieval of information.
- **Priority Queues:** Binary heaps are a type of binary tree used in implementing priority queues, where the root node always has the highest or lowest priority.
- **Expression Evaluation:** Binary trees, specifically expression trees, are used to evaluate expressions in compilers.
- **Routing Algorithms:** Trees such as tries are extensively used in networking for routing algorithms, especially in IP routing and subnetting.

Binary trees are versatile data structures that are foundational to many computer science concepts and applications. Their various specializations allow them to be tailored to address specific problems efficiently, making them indispensable tools in algorithm design and systems implementation.

Implementing a Simple Binary Tree in C++

Objective: To provide a complete implementation of a simple binary tree in C++, including basic operations like insertion, traversal, and node definition.

Node Definition and Constructor

We start by defining the structure for a tree node, which includes data and pointers to the left and right child nodes.

```
struct TreeNode {
    int value; // The value or key of the node
    TreeNode *left; // Pointer to the left child
    TreeNode *right; // Pointer to the right child
```

Insertion in a Binary Tree

For simplicity, this insertion function adds nodes level by level to keep the tree relatively balanced manually. We will use a queue to help with level-order insertion.

```
#include <queue>

void insert(TreeNode *&root, int value) {
    TreeNode *newNode = new TreeNode(value);
    if (root == nullptr) {
        root = newNode;
        return;
    }



    std::queue<TreeNode*> queue;
    queue.push(root);

    while (!queue.empty()) {
        TreeNode *current = queue.front();
        queue.pop();

        if (current->left == nullptr) {
            current->left = newNode;
            return;
        } else {
            queue.push(current->left);
        }

        if (current->right == nullptr) {
            current->right = newNode;
            return;
        } else {
            queue.push(current->right);
        }
    }
}
```

Traversal Methods

We'll implement three common traversal methods: in-order, pre-order, and post-order.

```
// In-Order Traversal: Left, Root, Right
void inOrder(TreeNode *node) {
    if (node != nullptr) {
        inOrder(node->left);
        std::cout << node->value << " ";
```

```
// Pre-Order Traversal: Root, Left, Right
void preOrder(TreeNode *node) {
    if (node != nullptr) {
        std::cout << node->value << " ";
        preOrder(node->left);
        preOrder(node->right);
    }
}

// Post-Order Traversal: Left, Right, Root
void postOrder(TreeNode *node) {
    if (node != nullptr) {
        postOrder(node->left);
        postOrder(node->right);
        std::cout << node->value << " ";
    }
}
```

Complete Binary Tree Example Usage

Finally, let's see how these components are used together in a simple test function.

```
#include <iostream>

int main() {
    TreeNode *root = nullptr; // Start with an empty binary tree

    // Insert nodes into the binary tree
    insert(root, 10);
    insert(root, 5);
    insert(root, 15);
    insert(root, 3);
    insert(root, 7);
    insert(root, 12);
    insert(root, 18);
```

```
    // Perform different tree traversals
    std::cout << "In-Order Traversal: ";
    inOrder(root);
    std::cout << std::endl;

    std::cout << "Pre-Order Traversal: ";
    preOrder(root);
    std::cout << std::endl;

    std::cout << "Post-Order Traversal: ";
    postOrder(root);
    std::cout << std::endl;

    return 0;
}
```

Explanation

- **Node Definition:** Each node in the tree contains an integer value and pointers to its left and right children.
- **Insertion:** Nodes are added level by level to maintain a simple structure.
- **Traversals:** Different traversal methods are used to visit all the nodes in specific orders, which can be useful for various applications such as expression evaluation, making copies of the tree, or simply outputting the tree's contents.

This implementation provides a basic but complete setup for managing simple binary trees in C++, showcasing foundational operations necessary for more advanced tree manipulation and algorithms.

Tree Traversal Methods

Objective: To explore all the common tree traversal methods, including their implementations, applications, and associated complexities.

Types of Tree Traversals

1. **In-Order Traversal (LNR)**
    - **Implementation:** For binary trees, this traversal visits the left subtree first, then the root, and finally the right subtree.
    - **C++ Implementation:**

```
void inOrder(TreeNode* root) {
    if (root != nullptr) {
        inOrder(root->left);
        std::cout << root->value << " ";
        inOrder(root->right);
    }
}
```

- **Applications:** In binary search trees (BSTs), in-order traversal retrieves data in sorted order. This property is used for in-order displays in GUIs or converting a BST into a sorted array.
- **Complexity:** O(n), where n is the number of nodes in the tree, as each node is visited once.

2. **Pre-Order Traversal (NLR)**

- **Implementation:** This method visits the root first, followed by the left subtree, then the right subtree.
- **C++ Implementation:**

```
void preOrder(TreeNode* root) {
    if (root != nullptr) {
        std::cout << root->value << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

- **Applications:** Pre-order is used to export a tree structure so that it can be reconstructed later. This is also useful for creating deep copies of the tree and for prefix notation of expressions.
- **Complexity:** O(n), as it visits each node exactly once.

3. **Post-Order Traversal (LRN)**

- **Implementation:** This method processes the left subtree, then the right subtree, and finally the root.
- **C++ Implementation:**

```
void postOrder(TreeNode* root) {
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        std::cout << root->value << " ";
    }
}
```

- **Applications:** Post-order traversal is useful for deleting or freeing nodes and space of the tree efficiently. It's also used for post-fix expression evaluation and certain graph algorithms where dependencies need to be resolved.
- **Complexity:** O(n), with each node being visited once.

4. **Level-Order Traversal (Breadth-First Search)**

- **Implementation:** This traversal visits nodes level by level from top to bottom.
- **C++ Implementation:**

```
#include <queue>
void levelOrder(TreeNode* root) {
    if (root == nullptr) return;
    std::queue<TreeNode*> queue;
    queue.push(root);

    while (!queue.empty()) {
        TreeNode* current = queue.front();
        queue.pop();
        std::cout << current->value << " ";
        if (current->left != nullptr) queue.push(current->le
ft);
        if (current->right != nullptr) queue.push(current->r
ight);
    }
}
```

- **Applications:** Level-order traversal is used in many real-time applications like breadth-first search in games, decision making processes, and whenever it is necessary to explore all siblings before moving to children (layer by layer exploration).

Explanation and Analysis

Each traversal serves different purposes depending on the required application:

- **In-Order Traversal** is inherently designed to work with binary search trees to access elements in sorted order.
- **Pre-Order Traversal** is particularly useful for operations where the tree needs to be explored as soon as it is encountered.
- **Post-Order Traversal** is suitable for scenarios where operations need to be performed after all descendants are processed.
- **Level-Order Traversal** is crucial for tasks that require processing one entire level of the tree before moving to the next.

The choice of traversal method depends largely on the specific requirements of the application, such as the need for sorted data, the reconstruction of trees, or the mode of exploration preferred in algorithms. All these traversal methods share a time complexity of O(n), which means they are efficient in terms of visiting each node once, but they differ in their usage of additional memory, especially in level-order traversal which requires additional space proportional to the breadth of the tree layer being traversed.

Binary Search Trees (BSTs)

Objective: To provide a comprehensive overview of Binary Search Trees (BSTs), including their definition, necessity, operations, and applications.

Definition of BSTs

A **Binary Search Tree (BST)** is a binary tree in which each node has a key (and possibly associated data), and it satisfies the following properties:

- The key in any node is greater than the keys in all nodes in its left subtree.
- The key in any node is less than the keys in all nodes in its right subtree.

This property makes BSTs efficient for searching, which gives them a significant advantage over other data structures like arrays or linked lists for certain applications.

Why We Need BSTs

BSTs are particularly valued for their ability to maintain a dynamically changing dataset in sorted order, allowing for quick lookup, addition, and removal of items. Here are key reasons for using BSTs:

- **Efficient Search:** BSTs offer efficient search operations, close to O(log n) time complexity in balanced scenarios, compared to O(n) in unsorted arrays or linked lists.
- **Ordered Data:** BSTs inherently maintain data in sorted order, which makes in-order traversal very efficient and useful for applications that require sorted data.
- **Flexibility in Data Manipulation:** Unlike arrays, BSTs are linked structures and are more flexible in handling dynamic data where insertions and deletions happen frequently.

Operations on BSTs

1. **Insertion**
    - To insert a new key, the tree is traversed starting from the root. The binary search property guides whether to go left or right, and the new key is inserted where the appropriate child link is null.
    - **Complexity:** O(h), where h is the height of the tree. In the best case (balanced tree), this is O(log n).
2. **Searching**
    - Searching starts at the root and traces a path downward, choosing left or right by comparing the key to be found with the current node.
    - **Complexity:** O(h). For balanced trees, it's about O(log n).
3. **Deletion**
    - To delete a node with a given key, there are three cases to consider:
        - **No Child:** Remove the node.
        - **One Child:** Replace the node with its child.
        - **Two Children:** Find the in-order successor (smallest key in the right subtree) or the in-order predecessor (largest key in the left subtree), copy its data to the node, and delete the successor or predecessor.
    - **Complexity:** O(h), where h is the height of the tree.
4. **Traversal**
    - Traversal operations (in-order, pre-order, post-order, and level-order) allow visiting all the nodes in various orders. In-order traversal of a BST will yield keys in a sorted sequence.

Applications of BSTs

1. **Database Systems:**

- BSTs are widely used in database systems to index data, allowing quick retrieval, addition, and deletion of records.

2. **Symbol Tables in Compilers:**
   - Used in compilers for storing identifiers and their attributes in an organized manner, where quick lookup and insertion are often required.
3. **Network Routing Algorithms:**
   - BSTs can help in managing sorted lists of IP addresses and routing tables efficiently.
4. **User Interface Management:**
   - Sorted data structures like BSTs are useful in applications where large sets of items need to be added, removed, and queried frequently, such as contact managers or reservation systems.

Conclusion

BSTs are a fundamental data structure that offer efficient performance for operations involving sorted data, provided the tree remains balanced. They are a staple in many systems where quick search and ordered data are required, although care must be taken to manage the tree's balance to prevent performance degradation. This is why self-balancing BSTs like AVL trees or Red-Black trees are often preferred in practice, as they automatically ensure the tree remains balanced after every insertion and deletion.

Implementing a Binary Search Tree (BST) in C++

Objective: To demonstrate the implementation of a Binary Search Tree (BST) in C++, including basic operations such as insertion, search, deletion, and traversal, along with a discussion on their complexities.

BST Node Structure and Class Definition

Let's start by defining the `TreeNode` structure and the `BST` class that will encapsulate our tree operations.

```
#include <iostream>

struct TreeNode {
    int value;
    TreeNode *left, *right;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr)
{}
};
```

```
class BST {
    private:
        TreeNode* root;

        TreeNode* insert(TreeNode* node, int value) {
            if (node == nullptr) {
                return new TreeNode(value);
            }
            if (value < node->value) {
                node->left = insert(node->left, value);
            } else if (value > node->value) {
                node->right = insert(node->right, value);
            }
            return node;
        }



        TreeNode* search(TreeNode* node, int value) {
            if (node == nullptr || node->value == value) {
                return node;
            }
            if (value < node->value) {
                return search(node->left, value);
            } else {
                return search(node->right, value);
            }
        }
```

```
TreeNode* deleteNode(TreeNode* node, int value) {
    if (node == nullptr) return node;

    if (value < node->value) {
        node->left = deleteNode(node->left, value);
    } else if (value > node->value) {
        node->right = deleteNode(node->right, value);
    } else {
        if (node->left == nullptr) {
            TreeNode* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            TreeNode* temp = node->left;
            delete node;
            return temp;
        }

        TreeNode* temp = minValueNode(node->right);
        node->value = temp->value;
        node->right = deleteNode(node->right, temp->value);
    }
    return node;
}



TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}

void inOrder(TreeNode* node) {
    if (node != nullptr) {
        inOrder(node->left);
        std::cout << node->value << " ";
        inOrder(node->right);
    }
}
```

```
    public:
        BST() : root(nullptr) {}

        void insert(int value) {
            root = insert(root, value);
        }

        void searchAndPrint(int value) {
            TreeNode* result = search(root, value);
            if (result == nullptr) {
                std::cout << "Value " << value << " not found in th
e BST." << std::endl;
            } else {
                std::cout << "Value " << value << " found in the BS
T." << std::endl;
            }
        }

        void deleteValue(int value) {
            root = deleteNode(root, value);
        }

        void displayInOrder() {
            inOrder(root);
            std::cout << std::endl;
        }
};
```

Operations and Their Complexities

1. **Insertion**
   - **Method:** Recursively place the new value in the correct position based on the BST property.
   - **Complexity:** Average case O(log n), worst case O(n) (skewed tree).
2. **Search**
   - **Method:** Recursively search for a value, moving left or right based on comparison.
   - **Complexity:** Average case O(log n), worst case O(n).
3. **Deletion**
   - **Method:** Handle three cases for a node with the target value: no child, one child, or two children (using in-order successor).
   - **Complexity:** Average case O(log n), worst case O(n).
4. **In-Order Traversal**
   - **Method:** Traverse left child, visit node, then right child.
   - **Complexity:** O(n) as it visits every node.

Example Usage in Main Function

```
int main() {
    BST tree;
    tree.insert(15);
    tree.insert(10);
    tree.insert(20);
    tree.insert(8);
    tree.insert(12);
    tree.insert(17);
    tree.insert(25);



    tree.displayInOrder();  // Should display the numbers in sorted
order

    tree.searchAndPrint(10);
    tree.deleteValue(10);
    tree.displayInOrder();  // 10 should be missing now

    return 0;
}
```

This implementation provides a robust framework for understanding and utilizing

binary search trees, focusing on the key operations required to manage and interact with this type of data structure effectively. The complexities discussed highlight the importance of tree balance, motivating further exploration into self-balancing trees like AVL or Red-Black trees for more consistent performance.

Comparative Table: Time Complexity of Operations Across Data Structures

Below is a table summarizing the time complexities for common operations—insertion, deletion, and search—across various data structures:

Data Structure	Insertion	Deletion	Search
Array (Fixed-size)	O(n) (append at end)	O(n) (remove at end)	O(n) (linear search)
Dynamic Array	O(1) amortized	O(1) (remove last)	O(n) (linear search)
Single Linked List	O(1) (insert at head)	O(1) (remove head)	O(n)
Double Linked List	O(1) (insert at head)	O(1) (remove head)	O(n)
Circular Linked List	O(1) (insert at head)	O(1) (remove head)	O(n)
Stack (Array-based)	O(1) (push, amortized)	O(1) (pop)	N/A (stacks don't search)
Stack (List-based)	O(1) (push)	O(1) (pop)	N/A (stacks don't search)
Queue (Array-based)	O(1) (enqueue, amortized)	O(1) (dequeue)	N/A (queues don't search)
Queue (List-based)	O(1) (enqueue)	O(1) (dequeue)	N/A (queues don't search)
Binary Search Tree	O(logn) avg O(n) worst	O(logn) avg O(n) worst	O(logn) avg O(n) worst

Notes on Complexities:

- **Arrays and Dynamic Arrays:** Searching is linear unless the array is sorted, in which case binary search can reduce the complexity to O(log n).
- **Linked Lists (all types):** Insertion and deletion are O(1) only when operating at the head or with direct access to the node (O(1) does not account for the time to traverse to a specific position).
- **Stacks and Queues:** These data structures are designed for specific access patterns (LIFO and FIFO, respectively), and thus typical search operations are not applicable.
- **Binary Search Trees:** The complexities assume that the tree is reasonably balanced. For a skewed tree (degenerate to a linked list), these operations can degrade to O(n).

Highlighting the Need for Improvement

Problems with Basic BSTs:

- The performance of BST operations is heavily dependent on the tree's height. In the worst case, if the tree becomes skewed, the operations can degenerate to O(n), similar to that of a linked list.

Improvements with Balanced Trees:

1. **AVL Trees:** Self-balancing binary search trees where the heights of two child subtrees of any node differ by no more than one. Rebalancing is done through rotations which ensure that the tree remains balanced after every insertion and deletion, maintaining O(log n) complexity for all operations.
2. **Red-Black Trees:** Another type of self-balancing BST, where each node stores an extra bit representing "color" (red or black). These trees ensure that the tree remains balanced with constraints on node colors and the structure, also maintaining O(log n) complexities.

Way Forward:

- Using balanced trees like AVL or Red-Black trees is essential when dealing with data that frequently changes. They provide faster access times compared to unbalanced BSTs due to their guaranteed logarithmic height.
- For specific use cases where operations are highly dynamic and require consistent performance, exploring further optimized tree structures such as B-trees (common in databases) and Splay Trees might also be beneficial.

These balanced structures are crucial in real-world applications where performance can significantly impact usability and efficiency, making them a vital area of focus in both academic study and practical implementation in software development.

Introduction to Balanced Trees

1.1 Definition and Concept A balanced tree is a type of advanced data structure that ensures the tree remains efficient for operations such as search, insert, and delete. The primary goal of a balanced tree is to maintain its height as minimal as possible while adding or removing nodes to keep operation costs proportional to the logarithm of the number of elements within the tree (`O(log n)`).

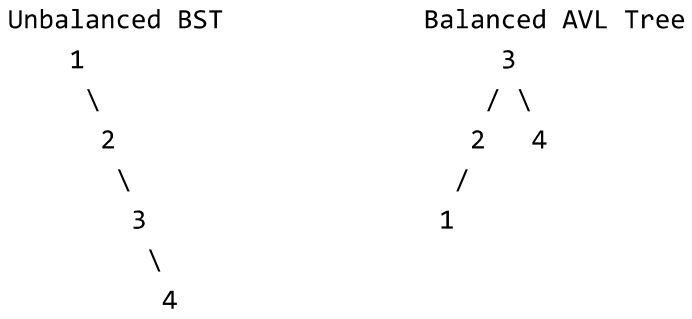
1.2 Importance of Balance The balance in a tree is crucial for maintaining high performance in dynamic data sets where insertions and deletions are frequent. In balanced trees, the depth of all leaf nodes is kept within a certain range relative to each other. This constraint helps avoid the degeneration of the tree into a linked list, which can happen in unbalanced

1.3 Performance Comparison: Balanced vs. Unbalanced Trees

- **Balanced Trees:**
  - Operations such as insertion, deletion, and search have time complexities of `O(log n)`. This efficiency is achieved by dynamically maintaining tree height as operations are performed.
- **Unbalanced Trees:**
  - While simple to implement, unbalanced trees can have a worst-case performance of `O(n)` for basic operations if the tree becomes skewed. For example, inserting sorted data into a regular binary search tree (BST) would result in a tree where each node has only one child, effectively becoming a linked list.

1.4 Visual Illustration Consider the following simple example for clarity:

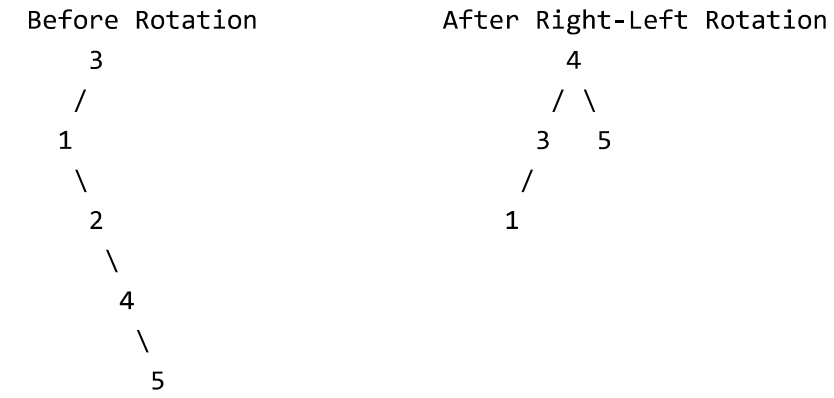
- **Unbalanced BST:** Inserting elements [1, 2, 3, 4, 5] into a BST leads to a structure where each node has only a right child, increasing the height to 5.
- **Balanced BST:** Inserting the same elements into an AVL tree, for instance, would result in a tree structure that maintains a height of 3, preserving faster search times.



Types of Balanced Trees

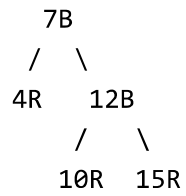
2.1 AVL Trees

- **Definition:** AVL trees are self-balancing binary search trees where the heights of the two child subtrees of any node differ by no more than one. This tight balancing ensures that AVL trees remain close to perfectly balanced.
- **Properties:**
  - AVL trees use rotations as their primary mechanism for maintaining tree balance after insertions and deletions.
  - Rotations include single rotations (right-right and left-left) and double rotations (right-left and left-right) to ensure the balance factor (difference between heights of left subtree and right subtree) remains within -1, 0, or +1.
- **Applications:** Highly suitable for lookup-intensive applications, such as database indices where frequent queries are made.



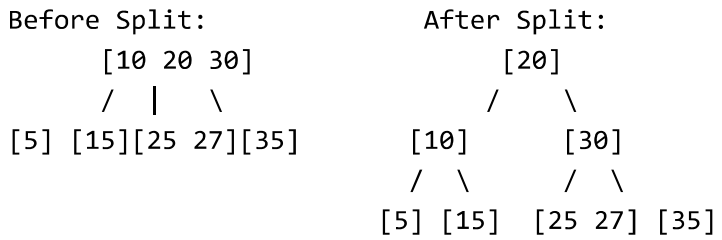
2.2 Red-Black Trees

- **Definition:** A red-black tree is a type of self-balancing binary search tree where each node contains an extra bit for denoting the color of the node, either red or black. This color attribute ensures the tree remains balanced during insertions and deletions.
- **Properties:**
  - The tree satisfies certain properties that ensure the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.
  - These properties include each node being either red or black, the root being black, all leaves (NIL nodes) being black, no two red nodes being adjacent, and every path from a given node to its descendant NIL nodes having the same number of black nodes.
- **Applications:** Commonly used in programming language libraries such as the C++ STL (Standard Template Library) for implementing map, multimap, multiset, and set.



2.3 B-Trees

- **Definition:** B-Trees are a generalization of binary search trees in which a node can have more than two children. This structure is particularly useful for storage systems that read and write large blocks of data.
- **Properties:**
  - B-Trees maintain balance by keeping all leaf nodes at the same depth.
  - B-Trees use node splits and merges as their primary balancing mechanism, which allows them to maintain a balanced state efficiently, even in the face of high volumes of insertions and deletions.
- **Applications:** Extensively used in databases and file systems to allow information to be stored, retrieved, and maintained in a way that allows for efficient scaling in large databases or systems that require frequent data modifications.



2.4 Splay Trees

- **Definition:** Splay trees are a type of self-adjusting search binary tree. Any operation on the tree – including search, insert, and delete – is followed by a splay operation.
- **Properties:**
  - The splay operation brings the node involved to the root of the tree through a series of rotations, thereby allowing recently accessed elements to be accessed quickly again.
  - The structure of a splay tree changes dramatically with each access, but it remains balanced by keeping frequently accessed elements near the root.
- **Applications:** Ideal for caches, memory management systems, and applications where locality of reference is prevalent, implying that recently accessed items may be accessed again soon.

Mechanisms for Maintaining Balance in Trees

3.1 Rotations and Their Rules

- **Purpose:** Rotations are fundamental operations in balanced trees that realign the nodes and update their heights to maintain tree balance.
- **Types:**
  - **Single Rotations:** These include right-right (RR) and left-left (LL) rotations, typically used when the tree becomes unbalanced by being too 'heavy' on one side.
  - **Double Rotations:** These include left-right (LR) and right-left (RL) rotations, used when the tree becomes unbalanced in a zigzag pattern, requiring a two-step rotation to restore balance.
- **Mechanism:**
  - In a rotation, the positions of the nodes are switched in such a way that balance is restored while preserving the order properties of the binary search tree.
  - For instance, in a left rotation on node X with right child Y, Y becomes the new root of the subtree, X becomes the left child of Y, and Y's left child becomes the right

child of X.

3.2 Node Splits and Merges (B-Trees)

- **Purpose:** Splits and merges in B-Trees handle node overflow and underflow, maintaining balance across the tree's multiple levels.
- **Node Split:**
  - Occurs when a node exceeds the maximum number of children or keys. The node is split into two, and the median key is moved up to the parent, helping to balance the tree.
  - This process might propagate up to the root, potentially raising the tree's height but maintaining its balanced nature.
- **Node Merge:**
  - Performed when a node falls below the minimum number of children or keys after a deletion. It involves merging with a sibling node and possibly borrowing a key from another sibling through a redistribution process, thus ensuring that all leaf nodes remain at the same depth.

3.3 Self-adjusting Techniques (Splay Trees)

- **Purpose:** Self-adjusting techniques in splay trees ensure that frequently accessed elements are quicker to access again, which indirectly helps maintain a balanced tree by keeping the most accessed elements near the root.
- **Splaying:**
  - Consists of moving a accessed node to the root of the tree using a series of rotations. The specific type of rotation (zig, zig-zig, or zig-zag) depends on the node's position relative to its parent and grandparent.
  - This operation not only optimizes access times for frequently accessed nodes but also works toward balancing the tree by spreading the access evenly over time.

Algorithmic Complexity of Balanced Trees

4.1 Overview of Complexities Balanced trees are designed to optimize the efficiency of basic operations such as insertions, deletions, and searches by maintaining a structured balance throughout the tree. This section explores the time complexities associated with these operations across different types of balanced trees.

4.2 AVL Trees

- **Time Complexity:** The insertions, deletions, and searches in AVL trees all operate in O(log n) time.
- **Explanation:**
  - AVL trees ensure that the heights of two child subtrees of any node differ by no more than one. The strict balancing rule necessitates few rotations (usually at most double rotations) to restore balance after updates, thus maintaining logarithmic height and ensuring operations complete in logarithmic time.

4.3 Red-Black Trees

- **Time Complexity:** Similar to AVL trees, Red-Black trees guarantee O(log n) time for insertions, deletions, and searches.
- **Explanation:**

- Although Red-Black trees do not maintain as strict a balance as AVL trees (thus potentially being taller), they simplify the balancing operations by using color changes and at most three rotations to restore properties after updates. This flexibility allows them to still achieve logarithmic operation times while often being faster in practice due to fewer rotations required per update.

4.4 B-Trees

- **Time Complexity:** B-Trees also provide O(log n) time complexity for insertions, deletions, and searches.
- **Explanation:**
  - B-Trees are multi-way trees that can have a large number of children per node, greatly reducing the tree's height and the path length for traversing the tree. The node splits and merges handle overflows and underflows effectively, maintaining tree balance and depth proportional to the log of the number of elements.

4.5 Splay Trees

- **Time Complexity:** Splay trees offer an amortized time complexity of O(log n) for insertions, deletions, and searches.
- **Explanation:**
  - Splay trees do not guarantee balanced tree height after every operation but adjust themselves over time through splaying. This means that individual operations can occasionally be expensive, but the tree compensates by making frequently accessed elements faster to access over time.

4.6 Comparative Analysis

- **Worst-case vs. Average-case:**
  - In AVL and Red-Black Trees, the worst-case time complexities are tightly bound to O(log n) due to strict structural properties.
  - In Splay Trees, while the worst-case time can be as bad as O(n) for a single operation, the amortized time complexity for a sequence of operations stays O(log n).
- **Unbalanced Trees:**
  - For comparison, operations in unbalanced binary search trees can degrade to O(n) in the worst case, such as when the elements are inserted in a sorted order, leading to tree degeneration into a linked list.

Applications of Balanced Trees

5.1 Database Indexing

- **Usage:** Balanced trees, particularly B-Trees and their variants like B+ Trees, are extensively used in database systems for indexing data.
- **Benefits:** They allow for efficient retrieval, insertion, and deletion of records within a database, significantly speeding up query response times even with very large datasets. The tree structure helps in maintaining sorted data, facilitating range queries and ordered data traversal.
- **Example:** Most modern relational database management systems (RDBMS) employ B-Trees for indexing tables and ensuring quick searchability.

5.2 Programming Language Libraries

- **Usage:** Data structures such as AVL Trees and Red-Black Trees are commonly implemented in standard programming libraries.
- **Benefits:** These trees provide the backbone for efficient implementations of maps, sets, and multimap containers, which are critical for many applications where key-value storage and retrieval are necessary.
- **Example:** The C++ Standard Template Library (STL) and Java's Collections Framework use Red-Black Trees to implement various associative containers like `std::map`, `std::set`, `TreeMap`, and `TreeSet`.

5.3 Memory Management Systems

- **Usage:** Balanced trees are used in operating systems and memory management systems to keep track of memory usage and allocation.
- **Benefits:** They help manage blocks of memory by keeping an organized structure, allowing the system to quickly find, allocate, and deallocate memory spaces without significant overhead.
- **Example:** AVL Trees are sometimes used in the implementation of memory allocators to manage free memory segments efficiently.

5.4 Network Applications

- **Usage:** Splay Trees find applications in network routers and switches for managing routing tables.
- **Benefits:** The self-adjusting property of Splay Trees is particularly useful in network routing where certain routes are accessed more frequently than others, allowing for faster access times to these routes.
- **Example:** Some advanced network systems use Splay Trees to enhance the performance of route lookups and modifications.

5.5 Real-Time Computing

- **Usage:** The need for consistent and predictable performance in real-time systems makes balanced trees a suitable choice.
- **Benefits:** Since operations on balanced trees like AVL or Red-Black Trees have predictable worst-case time bounds, they are ideal for scenarios where timing is critical.
- **Example:** Real-time gaming servers, financial trading systems, and high-frequency trading algorithms use balanced trees to manage user sessions and transaction orders efficiently.

5.6 Graphical User Interfaces

- **Usage:** Balanced trees are utilized in graphical user interface (GUI) frameworks to manage hierarchical objects and render them efficiently.
- **Benefits:** They help in quick visibility toggling, searching, and rearranging of UI components, ensuring that the interface remains responsive even with complex layouts.
- **Example:** Some advanced UI frameworks use tree structures to manage layers of graphical objects, where objects need to be frequently updated and reordered.

Summary Balanced trees are versatile data structures whose applications span across various domains from databases and system programming to network management and real-time applications. Their ability to maintain sorted data efficiently and perform quick insertions

Here's a table that lists various types of balanced trees and summarizes their key features as discussed:

Balanced Tree Type	Definition and Balancing Mechanism	Key Features	Typical Applications
AVL Trees	Self-balancing binary search trees with node-based height factors.	- Uses rotations to maintain balance. - Balance factor is always -1, 0, or +1. - Provides O(log n) complexity for insertions, deletions, and searches.	- Ideal for databases and systems where read operations are frequent and need to be fast.
Red-Black Trees	Binary trees with an extra color bit for balancing.	- Ensures no path is more than twice as long as any other. - Uses less strict balancing rules compared to AVL, allowing for faster insertions and deletions.	- Widely used in programming languages' libraries for implementing associative arrays, maps, and sets.
B-Trees	Multi-way trees with nodes containing multiple keys and children.	- Nodes can split or merge to maintain balance. - Depth of tree remains low by maintaining multiple keys per node, which reduces disk accesses.	- Commonly used in databases and file systems for efficient data storage and retrieval.
Splay Trees	Self-adjusting binary search trees.	- Frequently accessed elements are moved to the root. - Uses splaying to adjust structure, optimizing future accesses. - Offers amortized O(log n) complexity.	- Useful in network routing and caches where recently accessed items are likely to be accessed again soon.

This table provides a concise summary of each type of balanced tree, outlining their unique balancing mechanisms, key features, and typical applications, offering a clear comparison across different tree structures.

Red Black Tree Implementation

```
#include <iostream>

enum Color {RED, BLACK};

struct Node {
    int data;
    bool color;
    Node *left, *right, *parent;

    Node(int data): data(data), color(RED), left(nullptr), right(nu
llptr), parent(nullptr) {}
};
```

```
class RBTree {
    private:
        Node *root;

    protected:
        void rotateLeft(Node *&node) {
            Node *right_child = node->right;
            node->right = right_child->left;

            if (node->right != nullptr)
                node->right->parent = node;

            right_child->parent = node->parent;



            if (node->parent == nullptr)
                root = right_child;
            else if (node == node->parent->left)
                node->parent->left = right_child;
            else
                node->parent->right = right_child;

            right_child->left = node;
            node->parent = right_child;
        }



        void rotateRight(Node *&node) {
            Node *left_child = node->left;
            node->left = left_child->right;

            if (node->left != nullptr)
                node->left->parent = node;

            left_child->parent = node->parent;

            if (node->parent == nullptr)
                root = left_child;
            else if (node == node->parent->left)
                node->parent->left = left_child;
            else
                node->parent->right = left_child;

            left_child->right = node;
            node->parent = left_child;
        }
```

```
        void fixViolation(Node *&node) {
            Node *parent = nullptr;
            Node *grandparent = nullptr;

            while ((node != root) && (node->color != BLACK) && (nod
e->parent->color == RED)) {
                parent = node->parent;
                grandparent = node->parent->parent;

                /* Case A:
                   Parent of node is left child of Grand-parent of
node */
                if (parent == grandparent->left) {
                    Node *uncle = grandparent->right;



                    /* Case 1:
                       The uncle of node is also red
                       Only Recoloring required */
                    if (uncle != nullptr && uncle->color == RED) {
                        grandparent->color = RED;
                        parent->color = BLACK;
                        uncle->color = BLACK;
                        node = grandparent;
                    }
                    else {
                        /* Case 2:
                           node is right child of its parent
                           Left-rotation required */
                        if (node == parent->right) {
                            rotateLeft(parent);
                            node = parent;
                            parent = node->parent;
                        }
```

```
                        /* Case 3:
                           node is left child of its parent
                           Right-rotation required */
                        rotateRight(grandparent);
                        std::swap(parent->color, grandparent->colo
r);
                        node = parent;
                    }
                }
                /* Case B:
                   Parent of node is right child of Grand-parent of
node */
                else {
                    Node *uncle = grandparent->left;



                    /* Case 1:
                       The uncle of node is also red
                       Only Recoloring required */
                    if ((uncle != nullptr) && (uncle->color == RE
D)) {
                        grandparent->color = RED;
                        parent->color = BLACK;
                        uncle->color = BLACK;
                        node = grandparent;
                    }
                    else {
                        /* Case 2:
                           node is left child of its parent
                           Right-rotation required */
                        if (node == parent->left) {
                            rotateRight(parent);
                            node = parent;
                            parent = node->parent;
                        }
```

```
                        /* Case 3:
                           node is right child of its parent
                           Left-rotation required */
                        rotateLeft(grandparent);
                        std::swap(parent->color, grandparent->colo
r);
                        node = parent;
                    }
                }
            }

            root->color = BLACK;
        }



    public:
        RBTree() : root(nullptr) {}

        void insert(const int &data) {
            Node *node = new Node(data);
            root = BSTInsert(root, node);
            fixViolation(node);
        }

        Node* BSTInsert(Node* root, Node* node) {
            if (root == nullptr)
                return node;



            if (node->data < root->data) {
                root->left = BSTInsert(root->left, node);
                root->left->parent = root;
            }
            else if (node->data > root->data) {
                root->right = BSTInsert(root->right, node);
                root->right->parent = root;
            }

            return root;
        }
```

```
        void inorder() {
            inorderHelper(root);
        }

        void inorderHelper(Node *root) {
            if (root == nullptr)
                return;
            inorderHelper(root->left);
            std::cout << root->data << " ";
            inorderHelper(root->right);
        }
};



int main() {
    RBTree tree;
    tree.insert(7);
    tree.insert(3);
    tree.insert(18);
    tree.insert(10);
    tree.insert(22);
    tree.insert(8);
    tree.insert(11);
    tree.insert(26);

    tree.inorder();  // Inorder traversal of created tree
    return 0;
}
```

Reflection MCQs

1. **What type of data structure is a tree?**
   - A) Linear
   - B) Hierarchical
   - C) Tabular
   - D) Sequential
   - **Correct Answer: B**
2. **Which tree property allows AVL trees to maintain balance?**
   - A) Color of nodes
   - B) Number of children
   - C) Height balance factor
   - D) Depth of nodes
   - **Correct Answer: C**
3. **In a binary search tree, the left child of a node contains a value which is:**
   - A) Greater than the node value
   - B) Less than the node value

- C) Equal to the node value
- D) None of the above
- **Correct Answer: B**

4. **Which of the following operations has a time complexity of O(log n) in a balanced binary search tree?**

   - A) Insertion
   - B) Deletion
   - C) Searching
   - D) All of the above
   - **Correct Answer: D**
5. **What is the worst-case time complexity of searching in an unbalanced binary search tree?**

   - A) O(1)
   - B) O(log n)
   - C) O(n)
   - D) O(n log n)
   - **Correct Answer: C**
6. **Which of the following is not a type of binary tree?**

   - A) Red-Black Tree
   - B) Splay Tree
   - C) Circular Tree
   - D) AVL Tree
   - **Correct Answer: C**

7. **What does the Red-Black Tree use to ensure balance?**

   - A) Number of nodes
   - B) Depth levels
   - C) Node colors
   - D) Node values
   - **Correct Answer: C**
8. **B-Trees are particularly good for:**

   - A) CPU caching
   - B) Database indexing
   - C) Recursion optimization
   - D) Quick sorting
   - **Correct Answer: B**
9. **In a Red-Black Tree, the root must always be which color?**

   - A) Red
   - B) Black
   - C) Green
   - D) Blue
   - **Correct Answer: B**

10. **Which operation is not typically supported by a stack data structure implemented using a linked list?**

- A) Push
- B) Pop
- C) Peek
- D) Search
- **Correct Answer: D**

11. **Which type of tree automatically adjusts itself based on the access patterns to speed up future operations?**

    - A) Red-Black Tree
    - B) Splay Tree
    - C) AVL Tree
    - D) B-Tree
    - **Correct Answer: B**

12. **Node splits and merges are characteristic operations of which tree type?**

    - A) Binary Tree
    - B) AVL Tree
    - C) Red-Black Tree
    - D) B-Tree
    - **Correct Answer: D**

13. **Which of the following is a double rotation in AVL trees?**

    - A) Left-Left
    - B) Right-Right
    - C) Left-Right
    - D) None of the above
    - **Correct Answer: C**

14. **Splay Trees provide what type of time complexity for search operations?**

    - A) Constant time, O(1)
    - B) Linear time, O(n)
    - C) Logarithmic time, O(log n) on average
    - D) Quadratic time, O(n^2)
    - **Correct Answer: C**

15. **Which statement is true about inserting a new node in a Red-Black Tree?**

    - A) It requires checking and potentially changing the colors of a constant number of nodes.
    - B) It may require changing the tree structure globally.
    - C) It does not require any rotations to balance the tree.
    - D) It only involves a single rotation at most.
    - **Correct Answer: A**

16. **AVL trees are more balanced compared to Red-Black Trees because they:**

    - A) Allow only one child per node.
    - B) Use color properties to balance.
    - C) Keep the height differences between left and right subtrees to at most one.
    - D) Are less complex to implement.
    - **Correct

    Answer: C**

17. **In which tree is the property "every path from a node to its descendant NIL nodes has the same number of black nodes" found?**

    - A) AVL Tree
    - B) Splay Tree
    - C) Red-Black Tree
    - D) B-Tree
    - **Correct Answer: C**
18. **For a B-Tree of order m, each internal node can hold a maximum of how many keys?**

    - A) m-1
    - B) m
    - C) m+1
    - D) 2m-1
    - **Correct Answer: D**

19. **Which balanced tree type uses rotations for balancing that involve a grandparent node?**

    - A) B-Tree
    - B) Splay Tree
    - C) AVL Tree
    - D) Red-Black Tree
    - **Correct Answer: D**
20. **Which of the following trees does not guarantee balancing of nodes after every insertion and deletion?**

    - A) AVL Tree
    - B) Red-Black Tree
    - C) B-Tree
    - D) Splay Tree
    - **Correct Answer: D**
21. **What is the primary reason for using tree rotations in balancing operations?**

    - A) To make the tree look more aesthetically pleasing.
    - B) To ensure that the depth of the tree is minimized.
    - C) To reduce the complexity of tree operations.
    - D) To increase the data capacity of the tree.
    - **Correct Answer: B**

22. **In the context of balanced trees, what does the term "amortized complexity" refer to?**

    - A) The complexity of the most frequent operation
    - B) The average complexity over a series of operations
    - C) The worst-case complexity of any operation
    - D) The best-case scenario complexity
    - **Correct Answer: B**
23. **Which operation in a Red-Black Tree might trigger a "recoloring" or a "rotation"?**

    - A) Searching for a node
    - B) Accessing the root node

- C) Inserting a node
- D) Traversing from root to leaf
- **Correct Answer: C**

24. **In AVL Trees, what is the maximum number of rotations needed to balance the tree after an insertion?**

    - A) 1
    - B) 2
    - C) 3
    - D) 4
    - **Correct Answer: B**

25. **Which of the following is not a direct application of B-Trees?**

    - A) Implementing filesystems
    - B) Routing protocols in networks
    - C) Database indexing
    - D) Storing large blocks of data
    - **Correct Answer: B**

26. **In B-Trees, node splitting is triggered when a node's key count exceeds:**

    - A) The maximum defined key count.
    - B) Half of its maximum capacity.
    - C) Its parent's key count.
    - D) The tree's total number of keys.
    - **Correct Answer: A**

27. **Why might Splay Trees be preferred in a scenario where certain elements are accessed repeatedly?**

    - A) Because they decrease the overall size of the tree.
    - B) Because they adapt to access patterns by moving frequently accessed elements closer to the root.
    - C) Because they use less memory than other trees.
    - D) Because they do not require any balancing.
    - **Correct Answer: B**

28. **What distinguishes the balancing approach of Red-Black Trees from AVL Trees?**

    - A) Red-Black Trees balance the tree after every single operation without exceptions.
    - B) Red-Black Trees use a less rigid balancing approach, which allows faster insertions and deletions.
    - C) Red-Black Trees only use single rotations.
    - D) Red-Black Trees do not use color coding.
    - **Correct Answer: B**

29. **Which balanced tree structure is particularly effective for read-heavy applications due to its tight balancing properties?**

    - A) Red-Black Tree
    - B) B-Tree
    - C) AVL Tree
    - D) Splay Tree
    - **Correct Answer: C**

30. **Which property is essential for maintaining the structure of a Red-Black Tree?**

- A) The tree must be a perfect binary tree.
- B) No two red nodes can be adjacent.
- C) Every node must have two children.
- D) The tree must have exactly as many red nodes as black nodes.
- **Correct Answer: B**

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind