Binary Search Tree's

This doc contains tons of definitions and subsequently some topics that you should think of as example test question topics. For some visualization of basic BST https://www.programiz.com/dsa/binary-search-tree

A **Binary Search Tree (BST)** is a fundamental data structure in computer science, used to store data in a way that enables efficient searching, insertion, and deletion operations. It's a specific type of binary tree where each node has at most two children, referred to as the left child and the right child.

Definitions and Components:

Trees

- **Tree**: A hierarchical structure consisting of nodes, with one node designated as the root, where each node (except the root) is connected by an edge from exactly one other node known as its parent.
- **Subtree**: A subtree is a tree consisting of a node (referred to as the subtree's root) and all of its descendants. In addition to the subtree's root, it includes its children, the children's children, and so on. Each node in a tree can be the root of its own subtree with respect to the larger tree it is a part of.
- **Binary Tree**: A tree data structure in which each node has at most two children, referred to as the left child and the right child.
- **Binary Search Tree (BST)**: A binary tree where for every node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node, facilitating efficient searching, insertion, and deletion operations.

Other Tree Terms

- **Ancestor**: A node reachable by repeated proceeding from child to parent. An node's ancestors include its parent, its parent’s parent, and so on, up to the root.
- **Balanced**: The height of the left subtree subtracted from the height of the right subtree differs by no more than one.
- **Child**: A node that is directly connected to another node when moving away from the root. Each node can have zero, one, or more children.
- **Depth (of a node)**: The number of edges from the root to the node.
- **Descendant**: A node reachable by repeated proceeding from parent to child. A node’s descendants include its children, its children’s children, and so on.
- **Height (of a Node)**: The number of edges on the longest downward path between that node and a leaf. The height of the tree itself is the height of the root.
- **Height (of a node)**: The number of edges on the longest downward path between the node and a leaf. The height of the tree is the height of the root.
- **Inner Node**: A node that has at least one child (i.e., it is not a leaf).
- **Inorder Predecessor**: The inorder predecessor of a node in a binary search tree is the node with the largest key smaller than the key of the given node. In terms of inorder traversal, it is the node that appears immediately before the given node.

- **Inorder Successor**: The inorder successor of a node in a binary search tree (BST) is the node with the smallest key greater than the key of the given node. In terms of inorder traversal, it is the node that appears immediately after the given node.
- **Leaf**: A node with no children.
- **Level**: The depth of a node plus one. Levels are numbered from the root (level 1) downwards.
- **Node**: An individual element of a tree that contains data and may also include references to other nodes (children). This is the fundamental unit of a tree.
- **Parent**: The converse notion of a child. For any node except the root, the parent is the node that is one level above.
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Root**: The topmost node of a tree, which has no parent.
- **Sibling**: Nodes that share the same parent.
- **Width**: The maximum number of nodes at any level in the tree.

BST Operations

A binary search tree has the typical operations that many data structures have:

- **Insert**: Add a data element to the tree by placing a new node containing the data element into its proper location.
- **Find (search)**: Locates a node by traversing the BST and comparing each nodes data element to the "key" or search value.
- **Delete**: Removes a node from the BST.

Cost

The **cost** of each operation is ***O(log N)***. However we are only guaranteed this good complexity if our tree is balanced.

`Inserting` or `finding` a value in a BST follow somewhat similar steps but there is a fundamental difference. Could you describe or explain the differences?

`Deleting` a node from a BST requires assistance from the `Find` method, but then you are left with different scenarios (cases) in which you must deal with. Identify each case and explain its solution.

Why BSTs are represented with linked structures and not arrays:

BSTs are typically represented with linked structures (i.e., each node contains data and pointers to its children) rather than arrays due to several advantages:

1. **Dynamic Size**: Linked structures allow the tree to expand as new nodes are added without the need for reallocating or resizing an array, making insertion and deletion operations more efficient.
2. **Memory Efficiency**: Unlike arrays, there's no need to allocate memory for unused elements, as memory is allocated only for the actual nodes in the tree.
3. **Ease of Modifications**: Adding or removing nodes only requires updating a few pointers, without the need to shift elements as in an array.

Why recursive functions work so well with linked implementations of BSTs:

Recursive functions naturally align with the structure of BSTs for several reasons:

1. **Self-Similarity**: Each subtree of a BST is itself a BST. Recursive functions exploit this property by applying the same logic to a node and its subtrees, simplifying code complexity.
2. **Natural Base Case**: Leaf nodes (nodes without children) naturally provide a base case for recursive functions, often corresponding to the termination condition of the recursion.
3. **Divide and Conquer**: Many operations on BSTs, such as searching, insertion, and traversal, can be viewed as divide-and-conquer algorithms where the tree is divided into smaller subtrees that are handled independently. Recursive functions are well-suited for these types of algorithms.

Additional Tree Definitions & Concepts:

- **Traversal** (generic): Tree traversal is the process of visiting each node in a tree data structure, exactly once, in a systematic way.
- **Traversal** (ordered): BSTs support several types of traversals to access their elements, including in-order (left-root-right), pre-order (root-left-right), post-order (left-right-root), and level-order (breadth-first) traversal.
- **Balance**: The efficiency of operations on a BST depends on its height. A perfectly balanced tree (where each node's left and right subtrees' heights differ by at most one) allows for operations in logarithmic time complexity. However, a completely unbalanced tree (e.g., a linear chain of nodes) degrades to linear time complexity, similar to a linked list. This has led to the development of self-balancing BSTs (e.g., AVL trees and red-black trees) that maintain a more balanced structure to ensure operations remain efficient.
- **Use Cases**: BSTs are used in many applications, including database indices, filesystems, and in-memory data structures for efficient lookup and modification operations.

Traversals

In the context of Binary Search Trees (BSTs) and binary trees in general, there are several primary traversal strategies, each useful for different scenarios:

1. In-Order Traversal (Left-Root-Right)

In in-order traversal, the left subtree is visited first, followed by the root node, and finally the right subtree. For BSTs, this traversal outputs the nodes in non-decreasing order because of the BST property that left child ≤ parent and parent < right child.

Use Cases:

- **Sorting**: In-order traversal of a BST effectively sorts the data.
- **Binary Search Trees to Arrays**: Convert a BST to a sorted array.
- **Detecting BST Properties**: Helpful in checking if a binary tree is a BST by verifying if the output is sorted.

2. Pre-Order Traversal (Root-Left-Right)

Pre-order traversal visits the root node first, then recursively visits the left subtree, and finally the right subtree.

Use Cases:

- **Tree Copy**: Pre-order traversal is naturally suited for cloning a tree.
- **Expression Trees**: Used to print a copy of the tree as an expression (prefix notation).
- **Directory Structures**: Pre-order traversal can simulate the depth-first search algorithm, useful for exploring file systems or any hierarchical structure.

3. Post-Order Traversal (Left-Right-Root)

In post-order traversal, the tree is traversed first on the left subtree, then the right subtree, and finally the root node is visited.

Use Cases:

- **Tree Deletion**: Safely delete or free nodes from the bottom up.
- **Postfix Expression**: Evaluate postfix expressions in expression trees.
- **Dependency Trees**: Determine which dependencies to load first in dependency trees, where a node must be processed after its children.

4. Level-Order Traversal (Breadth-First)

Level-order traversal visits the nodes of the tree level by level, starting from the root. This traversal requires a queue to keep track of the nodes.

Use Cases:

- **Shortest Path**: In unweighted graphs represented by trees, level-order traversal can find the shortest path to a node.
- **Min/Max Depth**: Quickly determine the minimum or maximum depth of a tree.
- **Web Crawling**: In scenarios akin to web crawling where a breadth-first search is preferred for exploring resources.

Implementing Tree Traversals

Here are brief snippets for implementing in-order, pre-order, and post-order traversals in C++ (assuming a simple BST structure):

```
struct Node {
    int value;
    Node* left, *right;
};

// In-Order Traversal
void inOrderTraversal(Node* node) {
    if (node == nullptr) return;
    inOrderTraversal(node->left);
    std::cout << node->value << " ";
    inOrderTraversal(node->right);
}

// Pre-Order Traversal
```

```
void preOrderTraversal(Node* node) {
    if (node == nullptr) return;
    std::cout << node->value << " ";
    preOrderTraversal(node->left);
    preOrderTraversal(node->right);
}

// Post-Order Traversal
void postOrderTraversal(Node* node) {
    if (node == nullptr) return;
    postOrderTraversal(node->left);
    postOrderTraversal(node->right);
    std::cout << node->value << " ";
}
```

Each traversal method serves different purposes and is chosen based on the specific requirements of the task at hand. Understanding these traversal strategies is key to manipulating and leveraging tree data structures effectively in algorithms and applications.

Serialization

Serialization is the process of converting a data structure or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (over a network) and reconstructed later. The goal of serialization is to enable the persistence of complex data structures or the transfer of data between different components of a system, possibly across different computing environments.

In regards to BST's, the goal of serialization is to preserve the tree's structure and data so that the original tree can be accurately reconstructed resulting in the exact same tree (structure and data).

Best Methods for Serializing and Reconstructing a BST

1. Serialize Using Pre-Order or Post-Order Traversal

One effective approach is to use either pre-order or post-order traversal for serialization. This method ensures that you maintain the relative structure of the tree. The root node is serialized first, which helps during deserialization to reconstruct the tree correctly by knowing the root upfront.

- **Serialization**: Perform a pre-order or post-order traversal of the BST and write each node's value to a string or file, separating values with a delimiter (e.g., comma). For null pointers (to represent the end of a branch), you can use a special marker (e.g., `#` or `null`).
- **Deserialization**: Use the serialized string or file to reconstruct the tree by recursively creating nodes in the same order as they were serialized. The key is to maintain a pointer or iterator that moves through the serialized data, creating nodes based on the encountered values and reconstructing the tree structure accurately.

Advantages

- Straightforward to implement.
- The serialized data is relatively compact.

Example (Using Pre-Order Traversal)

```
// Assuming Node is defined as:
struct Node {
    int value;
    Node* left;
    Node* right;
    Node(int x) : value(x), left(NULL), right(NULL) {}
};

void serialize(Node* root, std::string& out) {
    if (!root) {
        out += "#,";
        return;
    }
    out += std::to_string(root->value) + ",";
    serialize(root->left, out);
    serialize(root->right, out);
}

Node* deserialize(std::string& data) {
    std::stringstream ss(data);
    std::string item;
    std::queue<std::string> elements;
    while (std::getline(ss, item, ',')) {
        elements.push(item);
    }
    return reconstruct(elements);
}

Node* reconstruct(std::queue<std::string>& elements) {
    std::string val = elements.front();
    elements.pop();
    if (val == "#") return nullptr;
    Node* node = new Node(std::stoi(val));
    node->left = reconstruct(elements);
    node->right = reconstruct(elements);
    return node;
}
```

2. Level-Order Traversal (Breadth-First)

Another approach is to serialize the tree using level-order traversal. This method involves traversing the tree level by level and serializing each node's value. This approach is intuitive for deserialization because it follows the natural top-down and left-right construction of the tree.

- **Serialization**: Perform a level-order traversal and serialize each node's value. Use a queue to track nodes and a special marker for null children.
- **Deserialization**: Read the serialized data sequentially to reconstruct the tree level by level, using a queue to keep track of nodes whose children have yet to be added.

Advantages

- More intuitive reconstruction process, especially for balanced trees.
- Ensures all levels are processed from top to bottom.

Conclusion

Both methods have their use cases and advantages. Pre-order or post-order based serialization is more space-efficient for sparse trees since it doesn't need to encode many null markers for missing children. However, level-order traversal can be more intuitive for deserialization and might be preferred for complete or nearly complete trees. The choice depends on the specific requirements and characteristics of the BST you're working with.

Stack or Queue Assisted Traversal

Let's break down the scenarios based on using a stack or a queue and the order of selecting children (smallest or largest value first) to see which of the standard tree traversals (pre-order, in-order, post-order, or level-order) we might mimic under these conditions.

Using a Stack

A stack is a Last-In, First-Out (LIFO) data structure, which means the last element added is the first to be removed. In the context of tree traversal, using a stack and always choosing the child with the smallest value first is akin to a depth-first search strategy.

- **Smallest Value First**: If you consistently choose the smallest value first and use a stack, you're essentially performing a variant of in-order traversal but in reverse. Normally, in-order traversal for a BST visits the left (smaller) child, then the root, and finally the right (larger) child. However, since you're using a stack and choosing the smallest value first, the actual traversal order is affected by how you insert the nodes into the stack (you would need to insert the right child followed by the left child to mimic in-order when popped).

- **Largest Value First**: Conversely, if you choose the largest value first with a stack, it doesn't directly mimic the standard in-order, pre-order, or post-order traversal as traditionally defined. You would be traversing the tree in a depth-first manner but prioritizing the larger values (right children in a BST) to be visited before the smaller ones (left children).

Using a Queue

A queue is a First-In, First-Out (FIFO) data structure, where the first element added is the first to be removed. This setup is naturally suited for level-order traversal (or breadth-first search) of a tree.

- **Smallest Value First or Largest Value First**: Regardless of whether you choose the smallest or largest child value first, using a queue inherently leads to a level-order traversal because you're visiting nodes level by level. However, the order in which nodes at the same level are visited will vary based on your child selection strategy. Typically, level-order traversal doesn't prioritize nodes based on their values but rather visits them from left to right. Selecting children based on their values (smallest or largest first) while using a queue doesn't directly mimic any of the standard depth-first traversal orders (pre-order, in-order, post-order) but rather influences the visitation order within each level.

Conclusion

- Using a stack with a depth-first approach and choosing the smallest value first can be manipulated to mimic a form of in-order traversal in reverse, depending on how you insert the nodes.
- Using a queue inherently leads to a level-order traversal pattern, though choosing nodes based on their values (smallest or largest first) alters the visitation order within each level rather than mimicking the standard depth-first traversal patterns.