1. Easy tree interview questions

You might be tempted to try to read all of the possible questions and memorize the solutions, but this is not feasible. Interviewers will always try to find new questions, or ones that are not available online. Instead, you should use these questions to practice the **fundamental concepts** of trees.

As you consider each question, try to replicate the conditions you'll encounter in your interview. Begin by writing your own solution without external resources in a fixed amount of time.

If you get stuck, go ahead and look at the solutions, but then try the next one alone again. Don't get stuck in a loop of reading as many solutions as possible! We've analysed dozens of questions and selected ones that are commonly asked and have clear and high quality answers.

Here are some of the easiest questions you might get asked in a coding interview. These questions are often asked during the "phone screen" stage, so you should be comfortable answering them without being able to write code or use a whiteboard.

1.1 Binary tree in-order traversal

- Text guide (LeetCode)
- Video guide (Nick White)
- Code example (LeetCode)

1.2 Symmetric tree

- Text guide (GeeksForGeeks)
- Video guide (Kevin Naughton Jr.)
- Video guide (Back to Back SWE)
- Code example (LeetCode)

1.3 Maximum depth of binary tree

- Text guide (Medium/Sara)
- Video guide (NeetCode)
- Code example (LeetCode)

1.4 Convert sorted array to binary search tree

- Text guide (Medium/Hary)
- Video guide (NeetCode)
- Code example (LeetCode)

1.5 Invert binary tree

- Text guide (After Academy)
- Video guide (Nick White)
- Video guide (Ben Awad)

- Code example (LeetCode)

1.6 Diameter of binary tree

- Text guide (Techie Delight)
- Video guide (NeetCode)
- Code example (LeetCode)

1.7 Merge two binary trees

- Text guide (After Academy)
- Text guide (LeetCode)
- Video guide (Nick White)
- Code example (LeetCode)

1.8 Same tree

- Text guide (LeetCode)
- Video guide (Kevin Naughton Jr.)
- Video guide (NeetCode)
- Code example (LeetCode)

1.9 Balanced binary tree

- Text guide (GeeksForGeeks)
- Video guide (Back to Back SWE)
- Video guide (NeetCode)
- Code example (LeetCode)

1.10 Minimum depth of binary tree

- Text guide (Educative.io)
- Video guide (Terrible Whiteboard)
- Code example (LeetCode)

1.11 Path sum

- Text guide (After Academy)
- Video guide (Kevin Naughton Jr.)
- Video guide (Nick White)
- Code example (LeetCode)

1.12 Binary tree pre-order traversal

- Text guide (Programiz)
- Video guide (Michael Sambol)
- Code example (LeetCode)

1.13 Binary tree post-order traversal

- Text guide (Programiz)
- Video guide (Michael Sambol)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

1.14 Binary tree paths

- Text guide (GeeksForGeeks)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

1.15 Sum of left leaves

- Text guide (GeeksForGeeks)
- Video guide (Kevin Naughton Jr.)
- Video guide (Nick White)
- Code example (LeetCode)

1.16 Find mode in binary search tree

- Text guide (Developpaper)
- Video guide (Nick White)
- Code example (LeetCode)

1.17 Subtree of another tree

- Text guide (Opengenus)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

1.18 N-ary tree pre-order traversal

- Text guide (Dev.to/Seanpgallivan)
- Video guide (Nick White)
- Video guide (Algorithms Made Easy)
- Code example (LeetCode)

1.19 N-ary tree post-order traversal

- Text guide (GeeksForGeeks)
- Video guide (Nick White)
- Code example (LeetCode)

1.20 Average of levels in binary tree

- Text guide (LeetCode)
- Video guide (Nick White)
- Video guide (Timothy H Chang)
- Code example (LeetCode)

1.21 Sum of root to leaf binary numbers

- Text guide (LeetCode)
- Video guide (Knowledge Center)
- Video guide (Timothy H Chang)
- Code example (LeetCode)

1.22 Increasing order search tree

- Text guide (LeetCode)
- Video guide (Algorithms Made Easy)
- Code example (LeetCode)

1.23 Range sum of BST

- Text guide (Andrew Hawker)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

2. Medium tree interview questions

Here are some moderate-level questions that are often asked in a video call or onsite interview. You should be prepared to write code or sketch out the solutions on a whiteboard if asked.

2.1 Validate binary search tree

- Text guide (Baeldung)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

2.2 Binary tree level-order traversal

- Text guide (Educative.io)
- Video guide (NeetCode)
- Video guide (Back to Back SWE)
- Code example (LeetCode)

2.3 Binary tree zigzag level-order traversal

- Text guide (Medium/Hary)
- Video guide (Knowledge Center)
- Code example (LeetCode)

2.4 Construct binary tree from pre-order and In-order traversal

- Text guide (Dev.to/Seanpgallivan)
- Text guide (LeetCode)
- Video guide (NeetCode)

- Code example (LeetCode)

2.5 Populating next right pointers in each node

- Text guide (Medium/Pruthvik)
- Video guide (Amell Peralta)
- Code example (LeetCode)

2.6 Kth smallest element in a BST

- Text guide (Dev.to/Akhil)
- Text guide (LeetCode)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

2.7 Lowest common ancestor of a binary tree

- Text guide (LeetCode)
- Video guide (NeetCode)
- Video guide (Back to Back SWE)
- Code example (LeetCode)

2.8 In-order successor in BST

- Text guide (Techie Delight)
- Video guide (Mycodeschool)
- Code example (GeeksForGeeks)

2.9 Unique binary search trees

- Text guide (GeeksForGeeks)
- Video guide (Back to Back SWE)
- Code example (LeetCode)

2.10 Flatten binary tree to linked list

- Text guide (After Academy)
- Video guide (Fit Coder)
- Video guide (Nick White)
- Code example (LeetCode)

2.11 Binary tree right side view

- Text guide (Medium/Annamariya)
- Text guide (GeeksForGeeks)
- Video guide (NeetCode)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

2.12 House robber III

- Text guide (Medium/Mollishree)
- Video guide (NeetCode)
- Code example (LeetCode)

2.13 Path sum III

- Text guide (LeetCode)
- Video guide (Code and Coffee)
- Code example (Medium/Len Chen)

2.14 Recover binary search tree

- Text guide (After Academy)
- Video guide (IDeserve)
- Code example (LeetCode)

2.15 Construct binary tree from in-order and post-order traversal

- Text guide (GeeksForGeeks)
- Video guide (Jenny's lectures)
- Video guide (Timothy H Chang)
- Code example (LeetCode)

2.16 Convert sorted list to binary search tree

- Text guide (Dev.to/seanpgallivan)
- Video guide (Algorithms Made Easy)
- Code example (LeetCode)

2.17 Path sum II

- Text guide (Medium/Len Chen)
- Video guide (Kevin Naughton Jr.)
- Code example (LeetCode)

2.18 Populating next right pointers in each node II

- Text guide (Medium/Nerd For Tech)
- Video guide (babybear4812)
- Video guide (Algorithms Made Easy)
- Code example (LeetCode)

2.19 Sum root to leaf numbers

- Text guide (GeeksForGeeks)
- Video guide (NeetCode)
- Video guide (TECH DOSE)

- Code example (LeetCode)

2.20 Binary search tree iterator

- Text guide (Medium/Deeksha Sharma)
- Video guide (Algorithms Made Easy)
- Video guide (The Code Mate)
- Code example (LeetCode)

2.21 Count complete tree nodes

- Text guide (Medium/Algo.Monster)
- Video guide (TECH DOSE)
- Code example (LeetCode)

2.22 Count univalue subtrees

- Text guide (Daily Coding Problem)
- Video guide (Yusen Zhang)
- Code example (GeeksForGeeks)

2.23 Delete node in a BST

- Text guide (GeeksForGeeks)
- Video guide (Mycodeschool)
- Code example (LeetCode)

3. Hard tree interview questions

Similar to the medium section, these more difficult questions may be asked in an onsite or video call interview. You will likely be given more time if you are expected to create a full solution.

3.1 Binary tree maximum path sum

- Text guide (LeetCode)
- Text guide (After Academy)
- Video guide (Michael Muinos)
- Code example (GeeksForGeeks)

3.2 Serialize and deserialize binary tree

- Text guide (Educative.io)
- Video guide (Back to Back SWE)
- Video guide (NeetCode)
- Code example (LeetCode)

3.3 Binary tree cameras

- Text guide (Dev.to/Seanpgallivan)
- Video guide (Algorithms Made Easy)
- Video guide (happygirlzt)
- Code example (LeetCode)

3.4 Sum of distances in tree

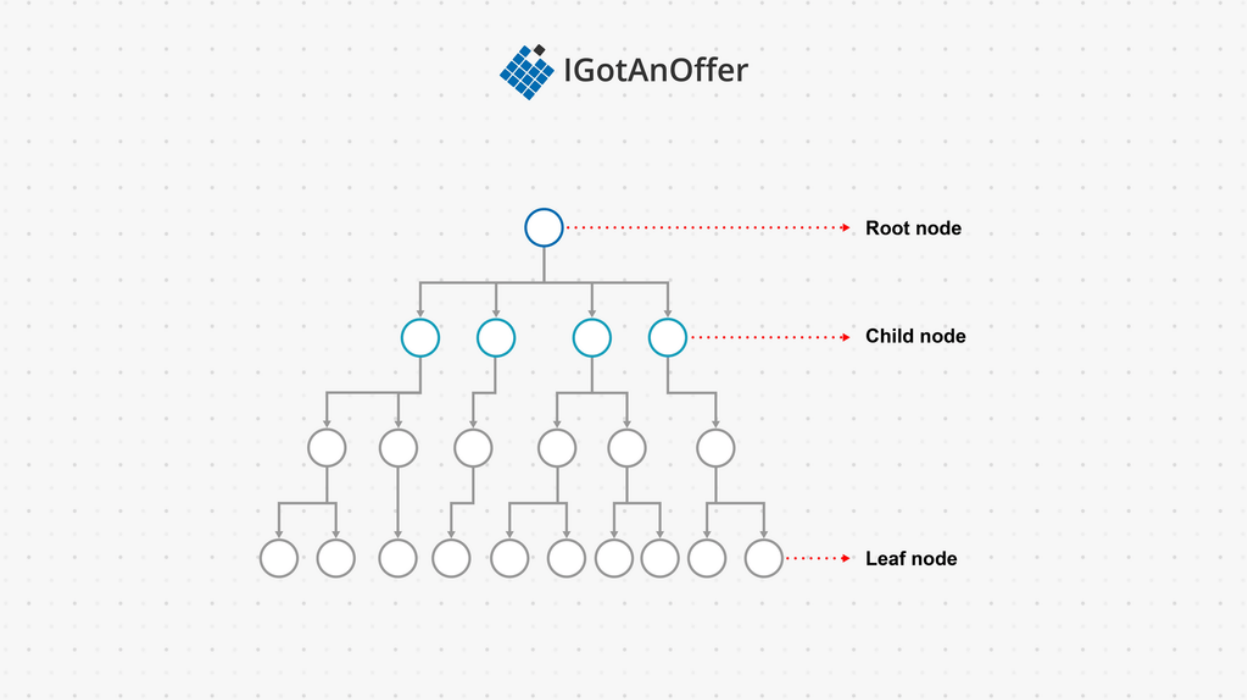
- Text guide (LeetCode)
- Video guide (happygirlzt)
- Code example (LeetCode)

4. Tree basics

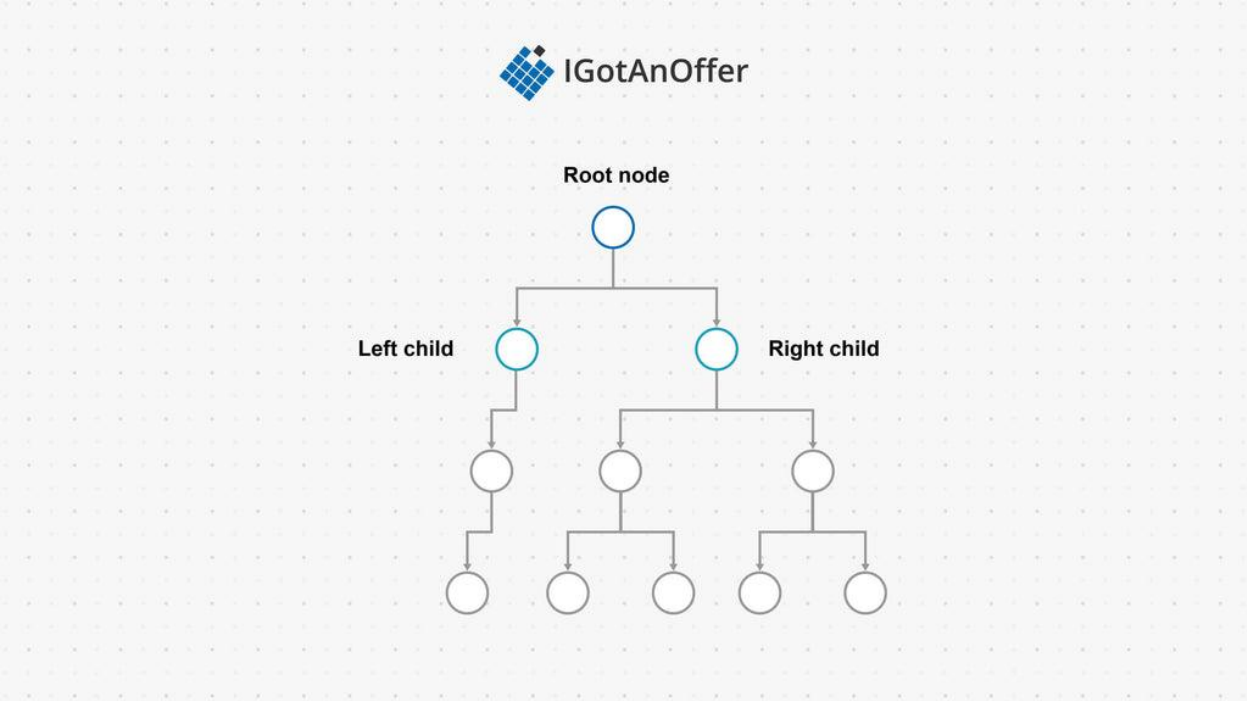
In order to crack the questions above and others like them, you'll need to have a strong understanding of trees, how they work, and when to use them. Let's get into it.

4.1 What are trees?

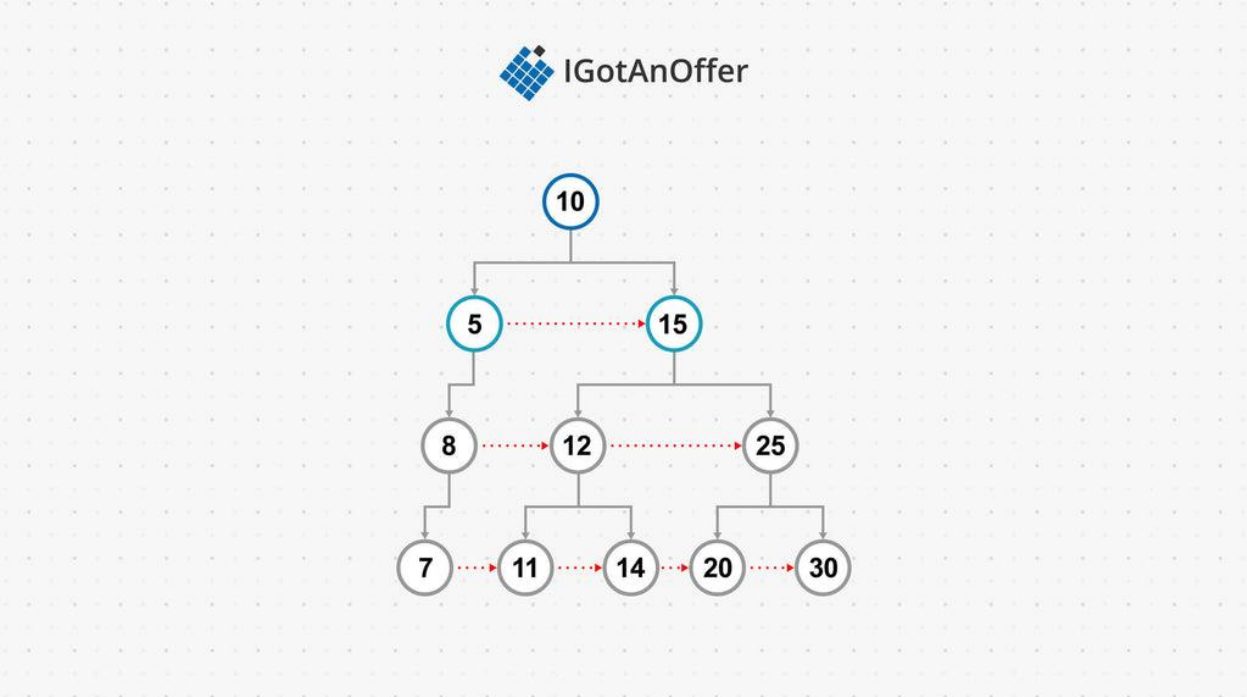
A tree is an abstract hierarchical data structure. It is represented by a group of linked nodes, with a single root node. Each node can have zero or multiple children. A leaf is a node with no children.



When talking about trees, we often mean binary trees. Binary trees are a type of tree where each node has at maximum two child nodes - a left child and a right child.



A particular type of binary tree is a binary search tree, or BST. In a BST, the nodes are ordered by their keys, making it efficient for search and inserting data. The key of any node is >= the value of its left child key, and <= the value of its right child key.



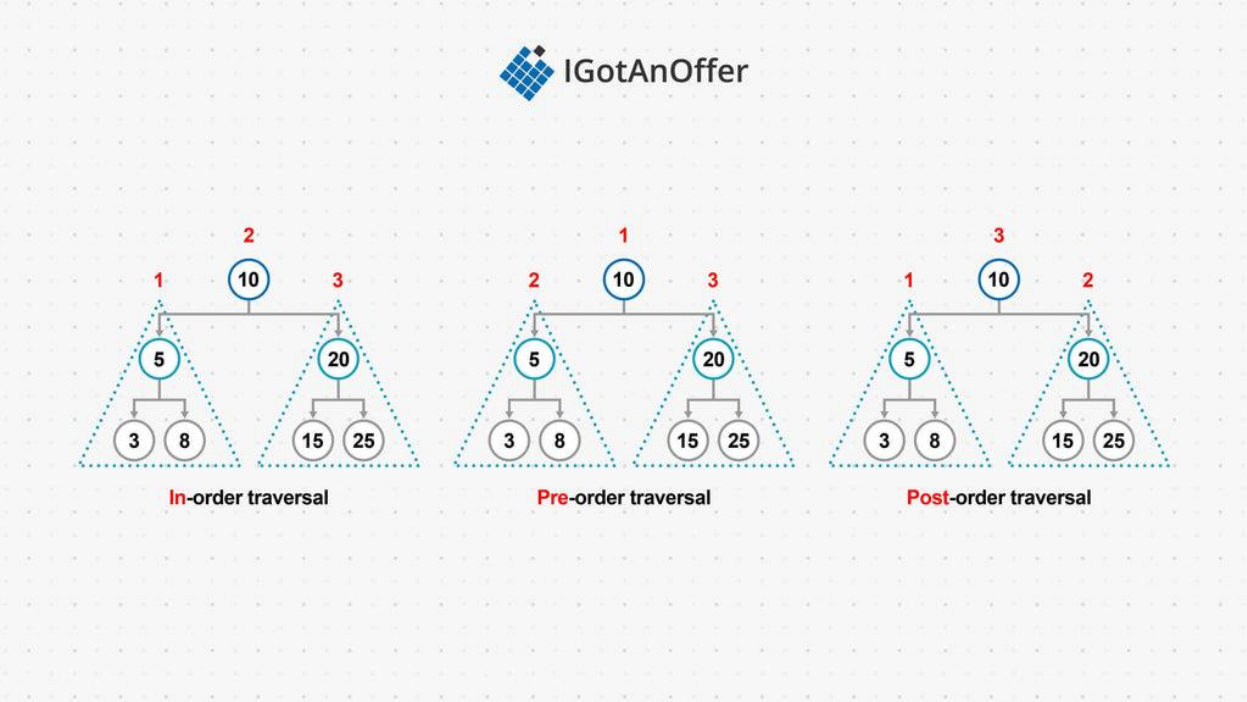
When elements are stored in a binary search tree, search, insert and delete times can be logarithmic, as up to half of the tree's elements can be eliminated at each node when searching

the tree. This means BSTs offer the advantages of a linked list (large size, quick removal and adding of elements), along with the search time advantages of a sorted array.

Searching, inserting, and deleting from the tree are similar processes. Searching is a recursive process, by checking if the desired key is equal to the current node's key. If it is, the node is returned. If the desired key is less than or greater than the current node's key, then the operation is run on the current node's left or right child respectively. Inserting an element into a BST involves searching the tree to find an empty leaf where the new element's node can be added. Deleting involves searching for an element, and then removing it from the graph by updating the links to and from the node.

Tree traversal is a common operation that visits every node in the tree in order to, for example, list the tree's contents, or for serialization. There are two main traversal strategies: depth-first, and breadth-first. Depth-first traversal visits nodes in order from root down to leaf recursively. Breadth-first traversal visits nodes of the same level before going to nodes of the next depth level.

Depth-first traversal has three sub-types: in-order, pre-order and post-order traversal. These refer to the order in which a node and its child nodes are visited.



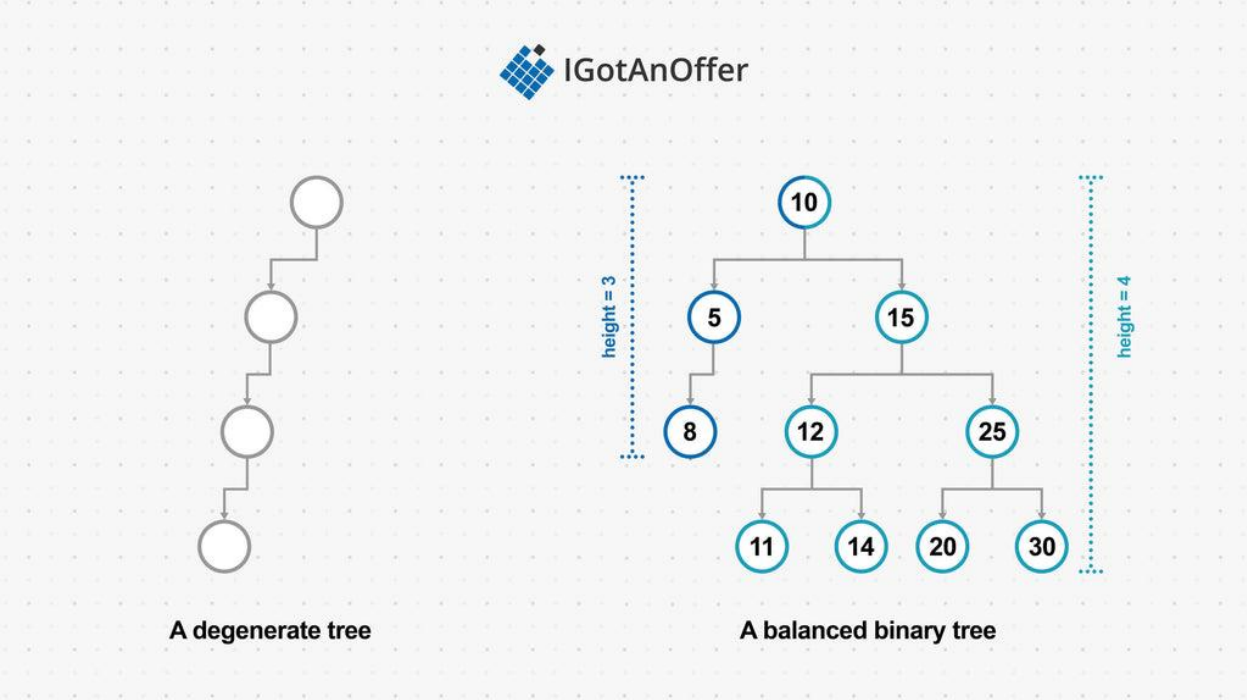
In-order traversal visits the node's left subtree, then the parent node, then the right subtree. This happens recursively. In this method, a BST's keys will be visited in ascending order.

Pre-order traversal visits the current node first, then the left subtree, then the right subtree. This happens recursively.

Post-order traversal visits the left subtree, then the right subtree, and finally the parent node. This happens recursively.

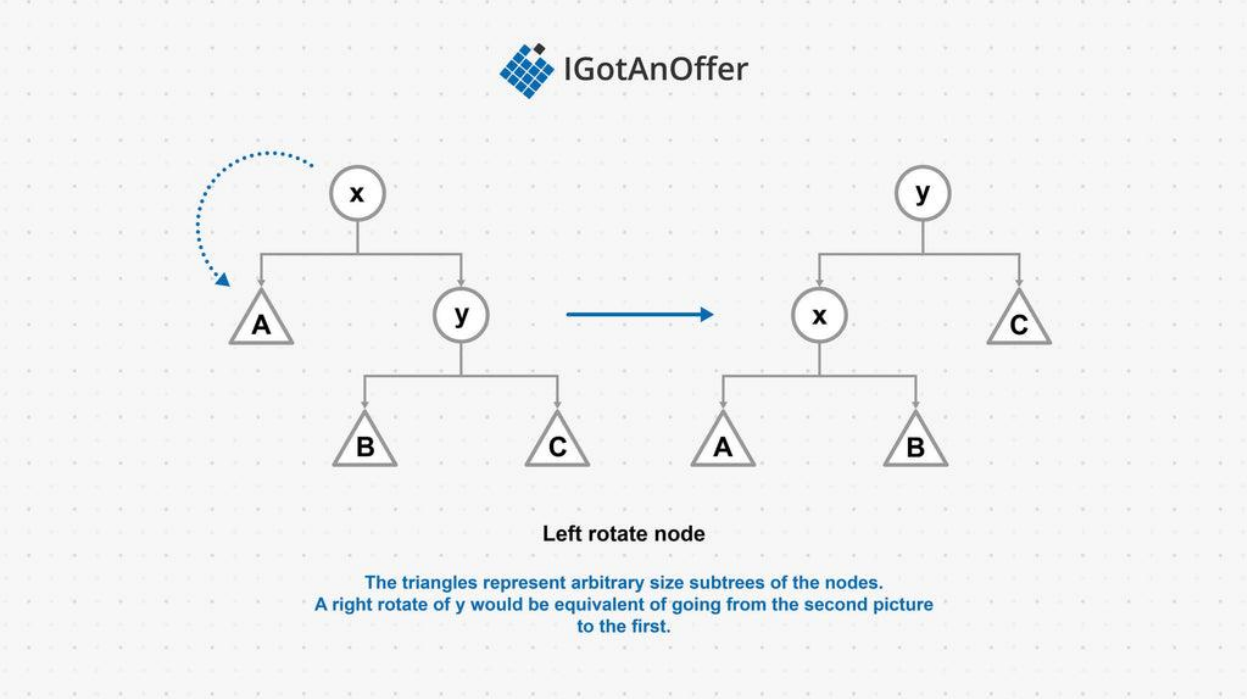
4.1.1 Types of trees (Java, Python, C++)

Binary trees can take on various shapes depending on how the nodes are distributed. A tree with only one child per node is called a **degenerate** tree, as it has degenerated into a linked list. A **height-balanced tree** is one where the left and right subtrees of every node have a height difference of not more than one. The height of a node is the length of the longest path from the node to a leaf.



BSTs can become unbalanced as elements are added or removed. In order to keep search and insert times logarithmic, binary search trees should be kept balanced. The more unbalanced a tree is (to the point of a degenerate tree, or linked list), the closer the search and update times are to linear.

A key operation used to balance trees is known as a rotation. This is a way to reorganize nodes in order to balance a tree. There are two types of rotations: left rotation and right rotation.



Self-balancing trees are BSTs that automatically keep their maximum height difference as close to one as possible. Two well known self-balancing BST's are red-black trees and AVL trees.

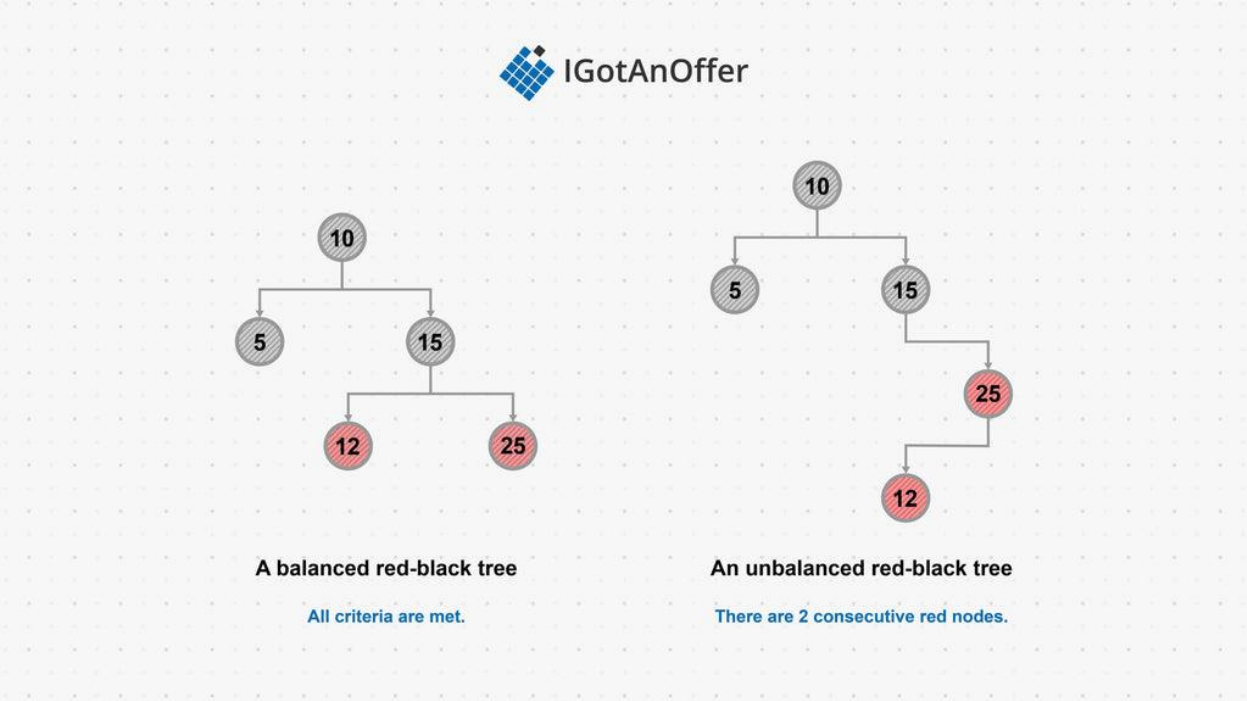
Red-black trees

A red-black tree is a binary search tree in which each node has an additional property to mark it as a “red node” or “black node.” A red-black tree is considered valid if the following properties apply to it:

- Every node is red or black.
- The root node is black.
- Every possible path through the tree from root to leaf has the same number of black nodes.
- There are no two consecutive red nodes in a path (there can be any number of consecutive black nodes in a path).

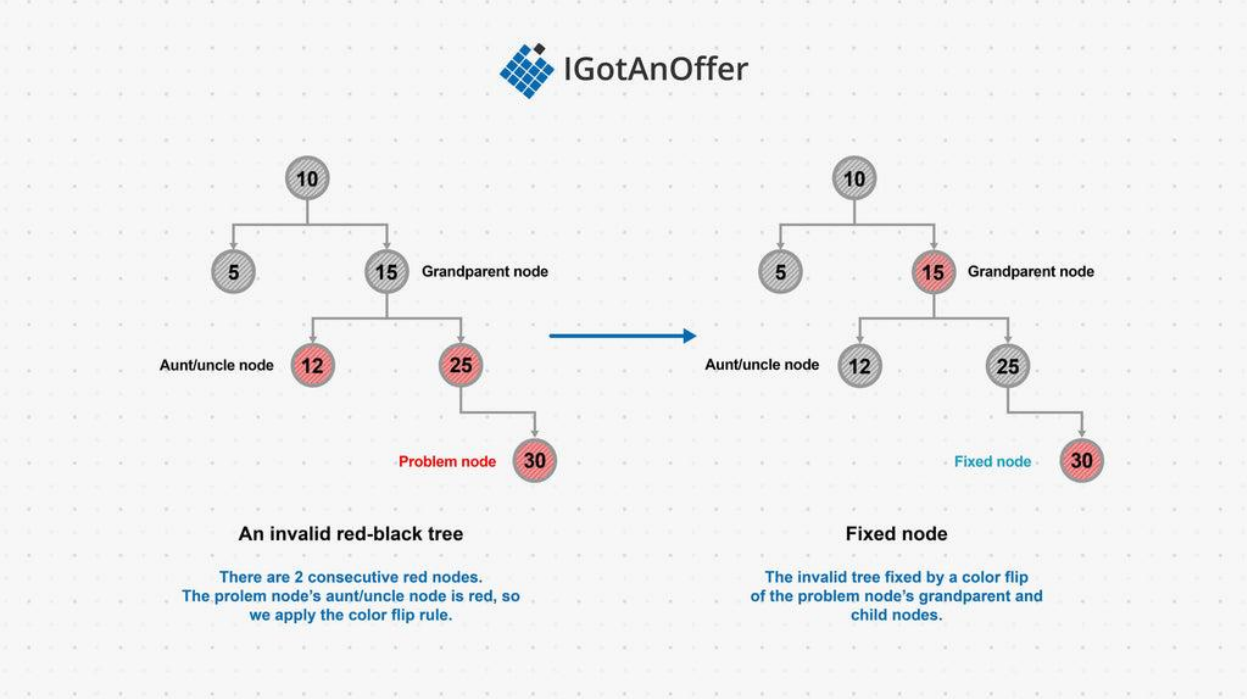
When operating on the tree, the following rules also apply:

- Any new node added is red.
- Any null (empty) node is considered black.

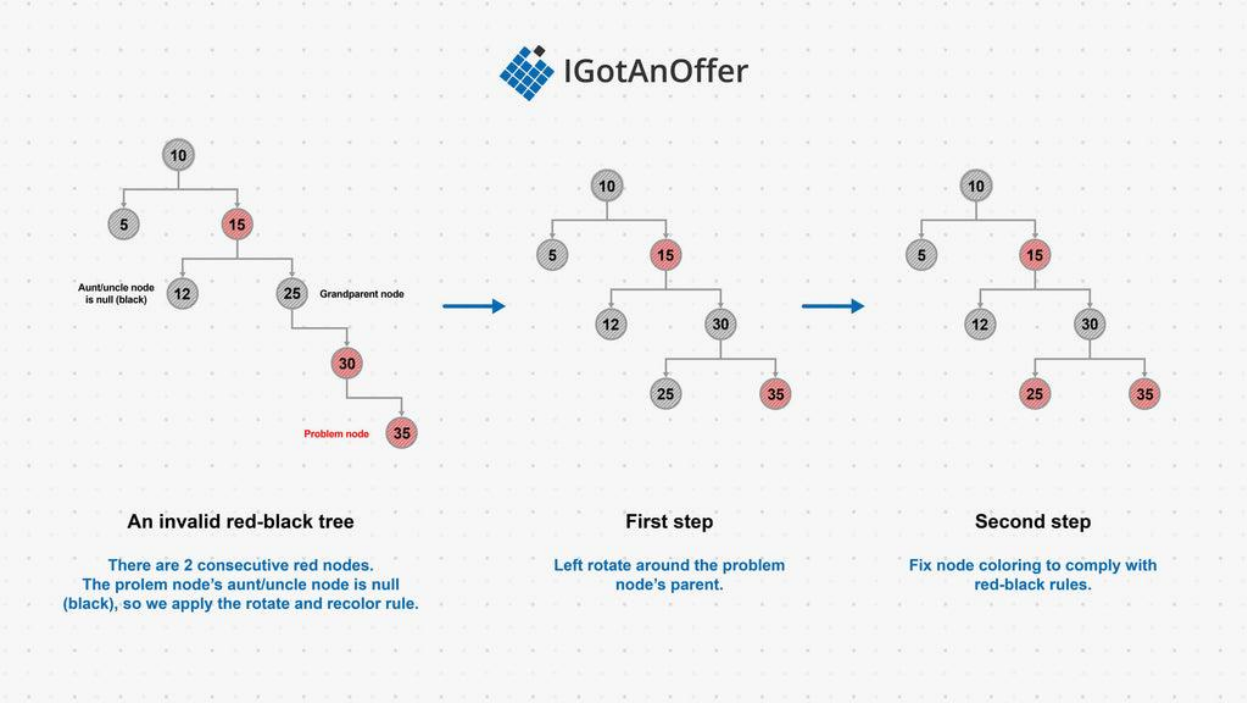


If a tree no longer meets the above criteria following an insertion or deletion, either a color flip or a rotation operation must be performed to rebalance the tree. The rules to determine which operation needs to occur are:

- If the node with an issue has a red aunt/uncle node, do a color flip on the node's grandparent and immediate children (make black grandparent or children nodes red, and red grandparent or children nodes black).
- If the node with an issue has a black aunt/uncle node, rotate around that node's grandparent. After the rotation, fix the affected node colors to comply with red-black tree properties.



An example of the steps to fix an invalid tree is shown below.

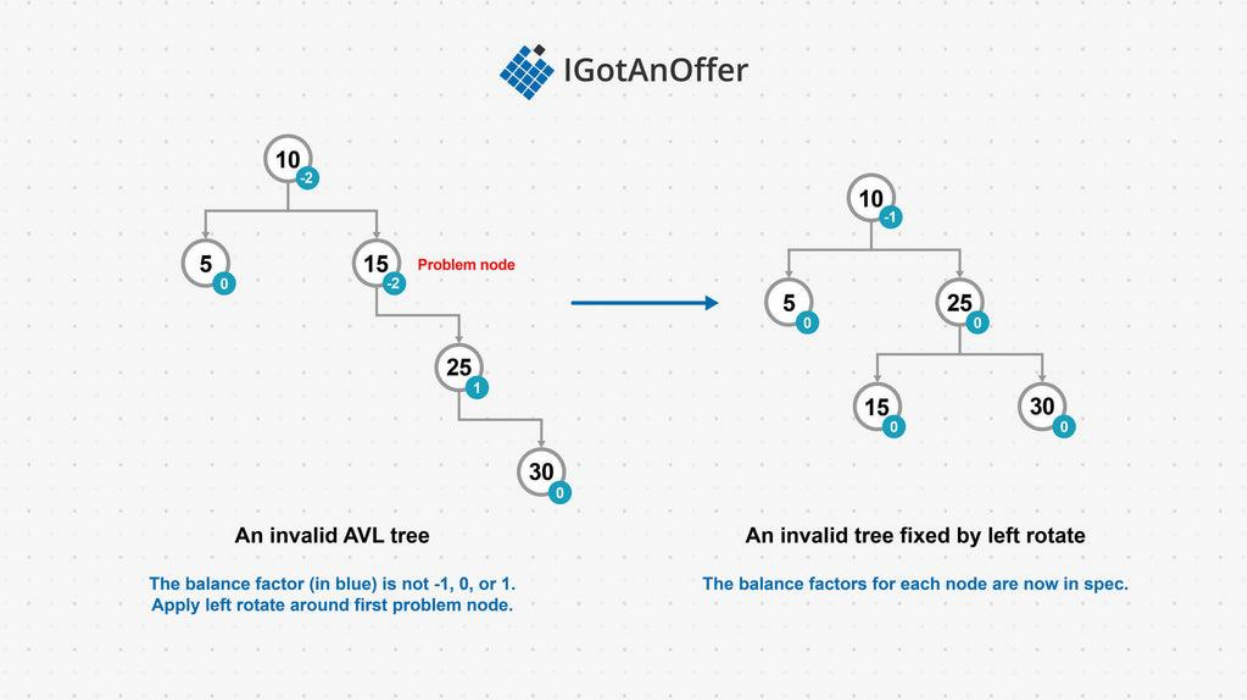


AVL trees

AVL (Adelson-Velski/Landis) trees were the first self-balancing binary search trees. AVL trees work by recording the height difference between the left and right subtrees of every node,

calculated as left_height - right_height. This height difference is known as the balance factor. For an AVL tree to be valid, the balance factor at each node must be -1, 0, or 1.

After modifying an AVL tree by inserting or deleting a node, the balance factors at each node upstream from the modification must be updated. If an imbalance is found at a node, i.e. the balance factor is not -1, 0, or 1, rotations must be performed to rebalance that node.



Other types of trees

Splay trees are adaptive, roughly balanced binary trees, i.e. not completely balanced. Splay trees optimize by keeping the most frequently accessed nodes near the top of the tree for faster access to those nodes. After a node is searched or added to a splay tree, the tree is *splayed* so that the node becomes the root of the whole tree. The splaying operation is performed by one rotation or multiple rotations until the most recently accessed node becomes the root. Because more frequently and recently accessed nodes will be moved nearer to the root of the tree, and therefore quicker to access again, splay trees are suited to cache implementations. It is possible for splay trees to become completely unbalanced, depending on the pattern of node access.

A Cartesian tree is a binary tree built from an ordered list of keys. When the tree is traversed in-order, it will produce the ordered list it was derived from. The tree must have the additional property that the parent of any node is smaller than the node itself.

B-trees are search trees designed for large amounts of data, typically on disk or in storage. They are optimized to reduce the number of disk access operations needed to retrieve a node, as disk access is very slow relative to RAM or processor speed. B-trees are not binary trees, as they may have more than two children per node. They are characteristically "fat," or wide.

Most languages do not have directly usable tree implementations. Generally, trees are implemented based on the particular use case. However, some classes, such as the Set class in the C++ STL, use a red-black tree implementation.

4.1.2 How trees store data

The primary structure in a tree is the node. The node contains the data element being stored, as well as references to the node's child nodes.

4.1.3 How trees compare to other data structures

Trees are used to store and retrieve data elements, so in that sense they can be compared to arrays and linked lists. Like linked lists, trees can store larger amounts of data more flexibly than arrays, as they are not reliant on contiguous memory. However, trees with ordered nodes, such as BSTs, are capable of faster search times than linked lists (O(log n) compared to O(n) for linked lists).

Trees are also a subset of graphs.

5. Tree cheat sheet

Trees Cheat Sheet

(Space-time complexity)

IGotAnOffer TECH

Time complexity:

		Worst Case Scenario	Average Case Scenario	Best Case Scenario
Binary Search Tree, Cartesian Tree, KD Tree	Delete	O(n)	O(log n)	O(log n)
	Insert	O(n)	O(log n)	O(log n)
	Search	O(n)	O(log n)	O(log n)
B-Tree, Red-Black Tree, Splay Tree, AVL Tree	Delete	O(log n)	O(log n)	O(log n)
	Insert	O(log n)	O(log n)	O(log n)
	Search	O(log n)	O(log n)	O(log n)
Traversal		O(n)	O(n)	O(n)

Algorithm Complexity:

	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Depth-First Search (In-order, pre-order, and post-order traversal)	O(n)	O(n)	O(n)	O(n)
Breadth-First Search (Level-order traversal)	O(n)	O(n)	O(n)	O(n)
Tree Sort	$O(n^2)$	O(n log n)	O(n log n)	O(n)
Splaysort	O(n log n)	O(n log n)	O(n)	O(n)
Cartesian Tree Sort	O(n log n)	O(n log n)	O(n)	O(n)

You can download the cheat sheet here.