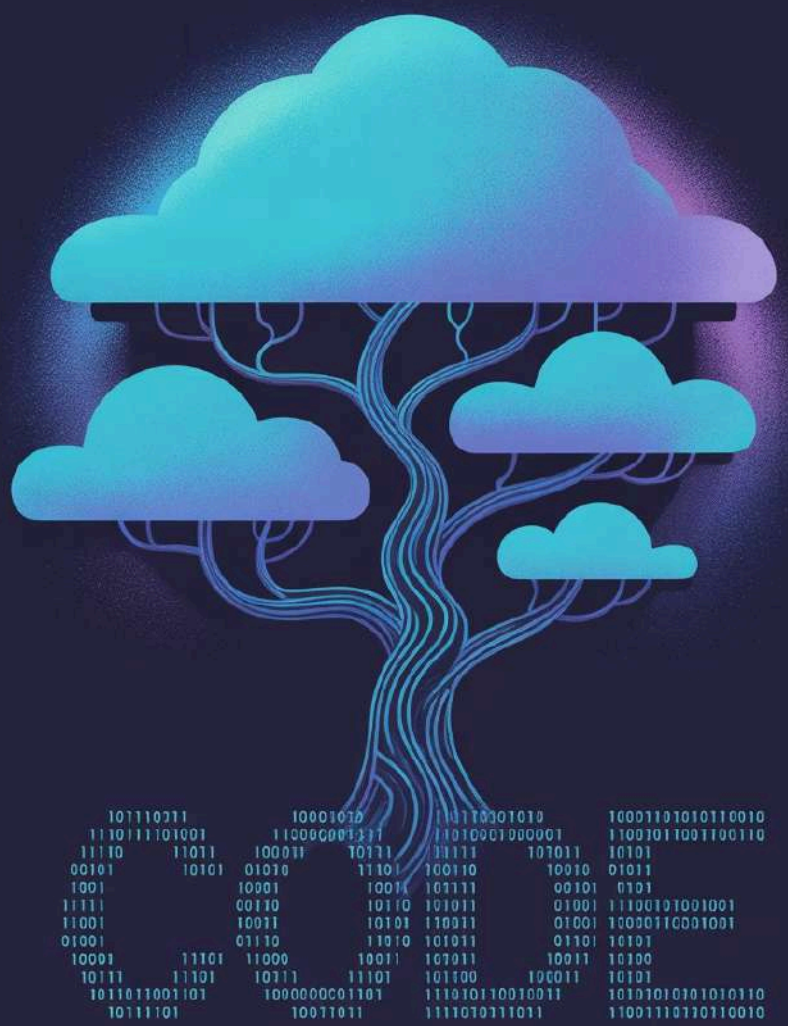


Introduction to Trees. Binary Search Trees

Welcome to the lecture on one of the most important data structures in programming! Today, we will explore trees and their special type — binary search trees.

📘 Lecture 14

Introduction to Trees. Binary Search Trees (BST)

Data structures are the foundation of effective programming. Trees are a powerful tool for organizing and searching for information, which we actively use in modern applications and systems.

Lecture Objectives

Understanding the Structure

To study the hierarchical nature of trees and their basic properties

Practical Skills

To learn how to implement Binary Search Trees (BST) in C++

Applying Knowledge

To understand where and how to use BSTs in real projects

🌳 Part 1

What is a Tree?

A tree is a hierarchical data structure that naturally organizes information on the 'parent-child' principle. Unlike linear structures (arrays, lists), trees allow for the representation of complex relationships between elements.

01

Nodes

Fundamental elements of a tree, containing data

02

Edges

Connections between nodes, defining the structure

03

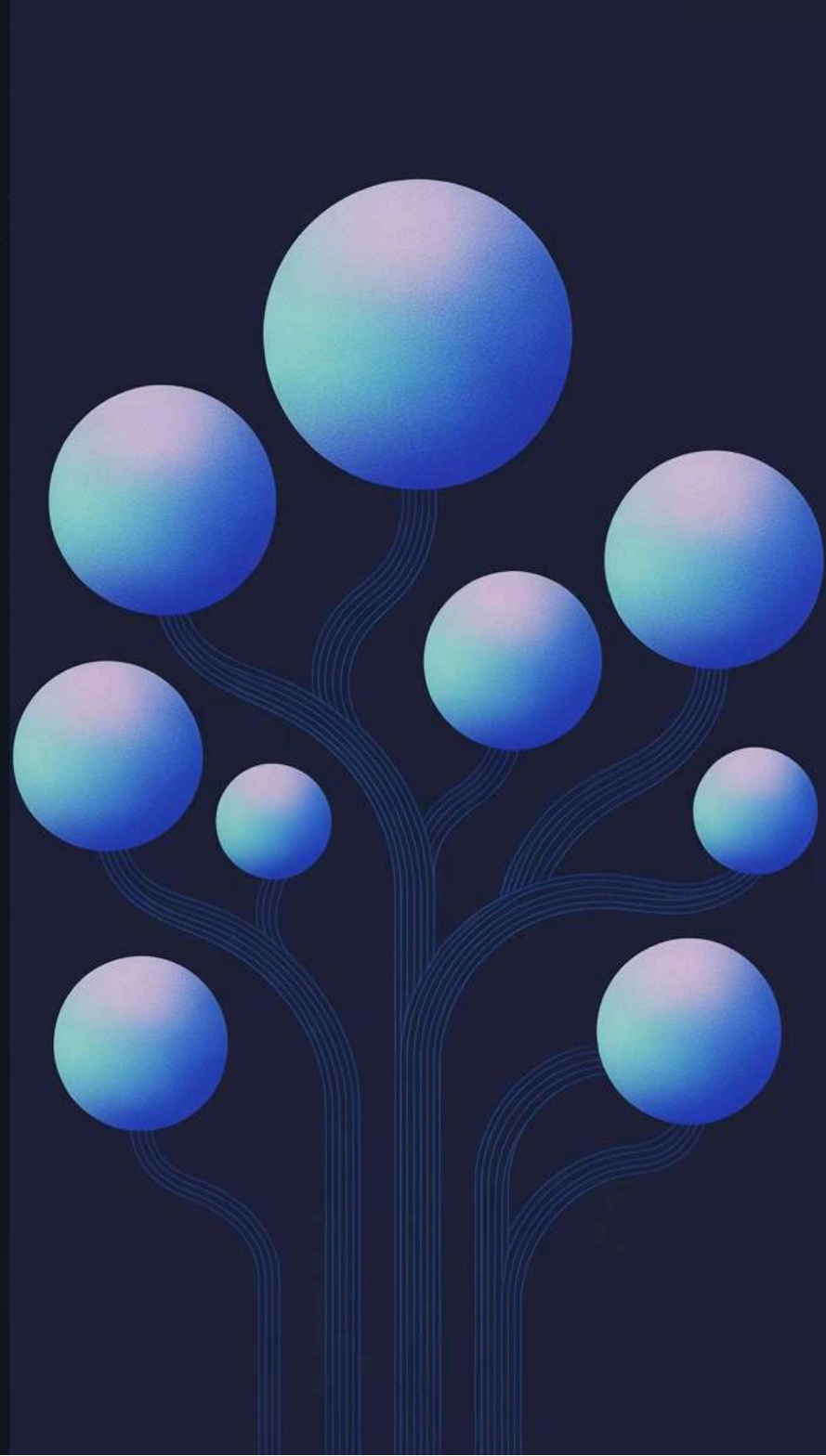
Root

The primary node from which the tree originates

04

Leaves

Nodes without children, the endpoints of branches



Examples of Trees in Everyday Life

File System

Folders contain files and subfolders, forming a tree-like hierarchy. The root folder contains all other system elements.

Organizational Structure

Director → managers → employees. Each management level represents a node in the company's hierarchy.

Family Tree

Parents and descendants form a natural tree-like structure, showing family relationships.

Key Terms

Tree Height

The maximum distance from the root to any leaf. Defines the depth of the longest branch.

Node Depth

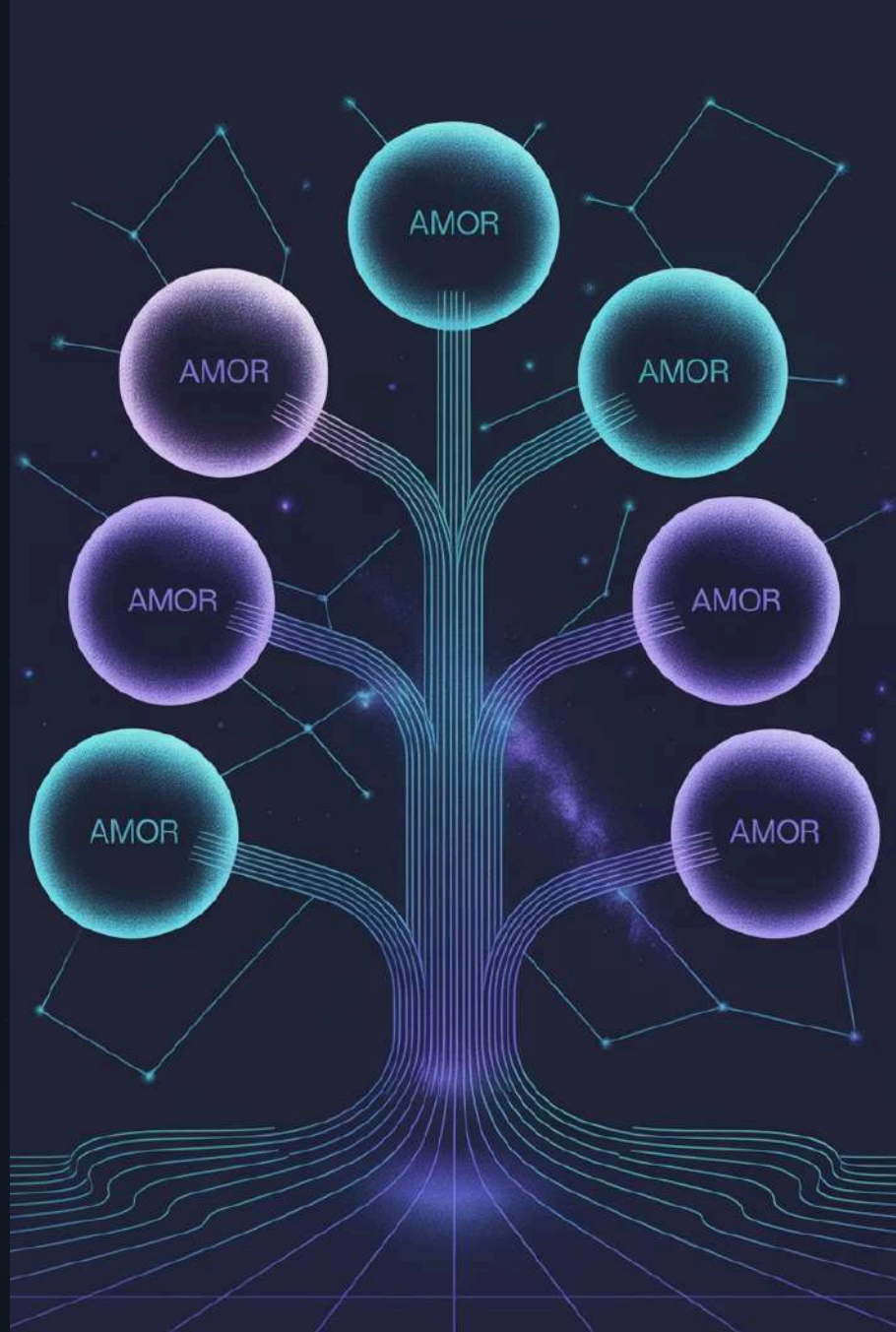
The distance from the root to a specific node. The root has a depth of 0.

Subtree

A part of a tree that is itself a tree with its own root and descendants.

Node Degree

The number of direct children of a node. In a binary tree, the degree does not exceed 2.



Types of Trees

Binary Tree

Each node has no more than 2 children

Balanced Tree

The heights of the left and right subtrees differ by no more than 1

Binary Search Tree

An ordered tree with special properties for fast searching

Complete Binary Tree

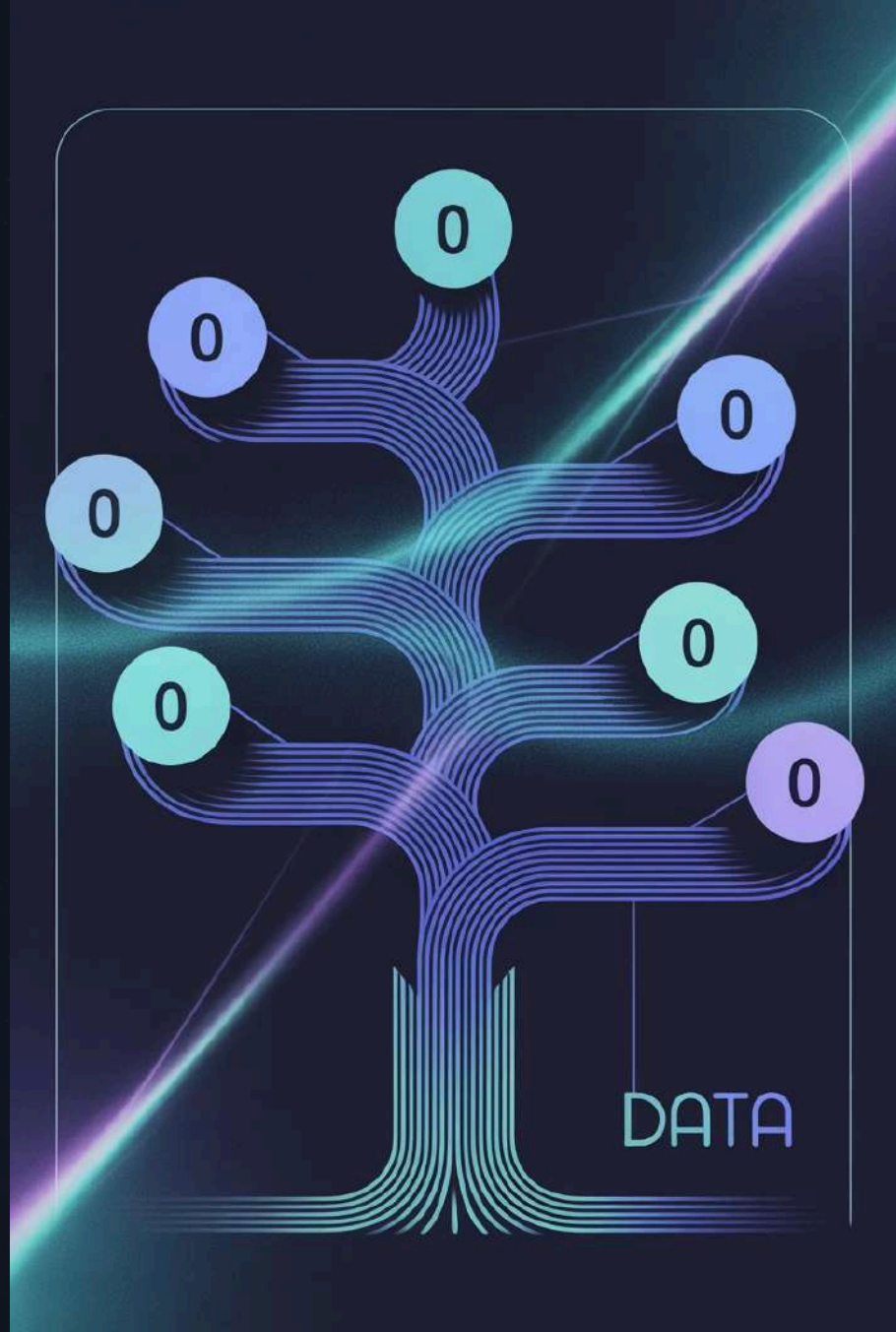
All levels are filled, except possibly the last one

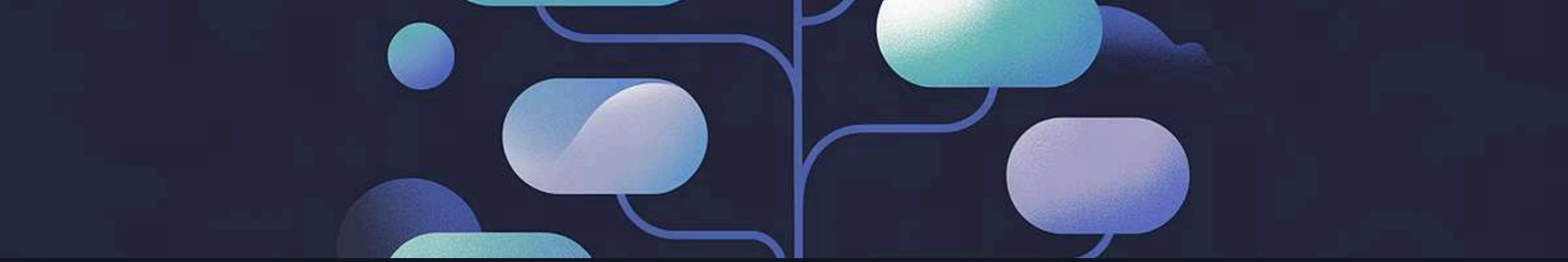
🌲 Part 2

Binary Search Trees (BSTs)

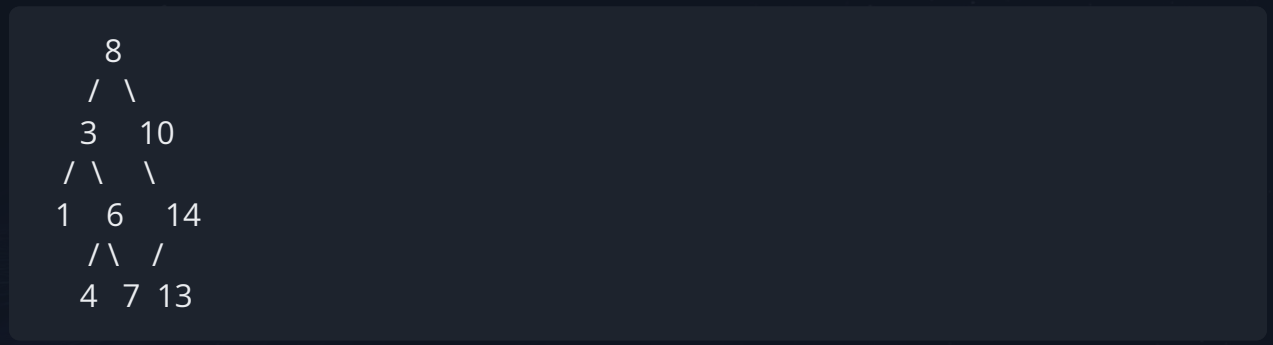
A Binary Search Tree (BST) is a special type of binary tree where elements are arranged in a specific order. This key property makes search, insertion, and deletion operations very efficient.

> **Main BST Property:** For any given node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value.





Example BST Structure



This example demonstrates how the fundamental property of BSTs is maintained: left children are less than the parent, and right children are greater.

Let's verify the property:

- Node 8: left subtree {1,3,4,6,7} < 8
- Right subtree {10,13,14} > 8
- Node 3: 1 < 3 < 6
- Node 10: 10 < 14

Applications of BST in Programming

Fast Search

The average search complexity of O(log n) makes BST ideal for frequently performed search operations in large datasets.

Databases

Database indexes are often implemented using balanced search trees for quick record access.

Dictionaries and Sets

STL containers std::set and std::map in C++ use balanced search trees for efficient operation.

Implementing a Tree Node in C++

Let's start by defining the basic node structure. Each node contains data and pointers to its left and right children.

```
struct Node {
    int key;          // Значение узла
    Node* left;       // Указатель на левого потомка
    Node* right;      // Указатель на правого потомка

    // Конструктор для инициализации узла
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};
```

Important: Initializing pointers with nullptr prevents access errors to undefined memory.

Inserting Elements into a BST

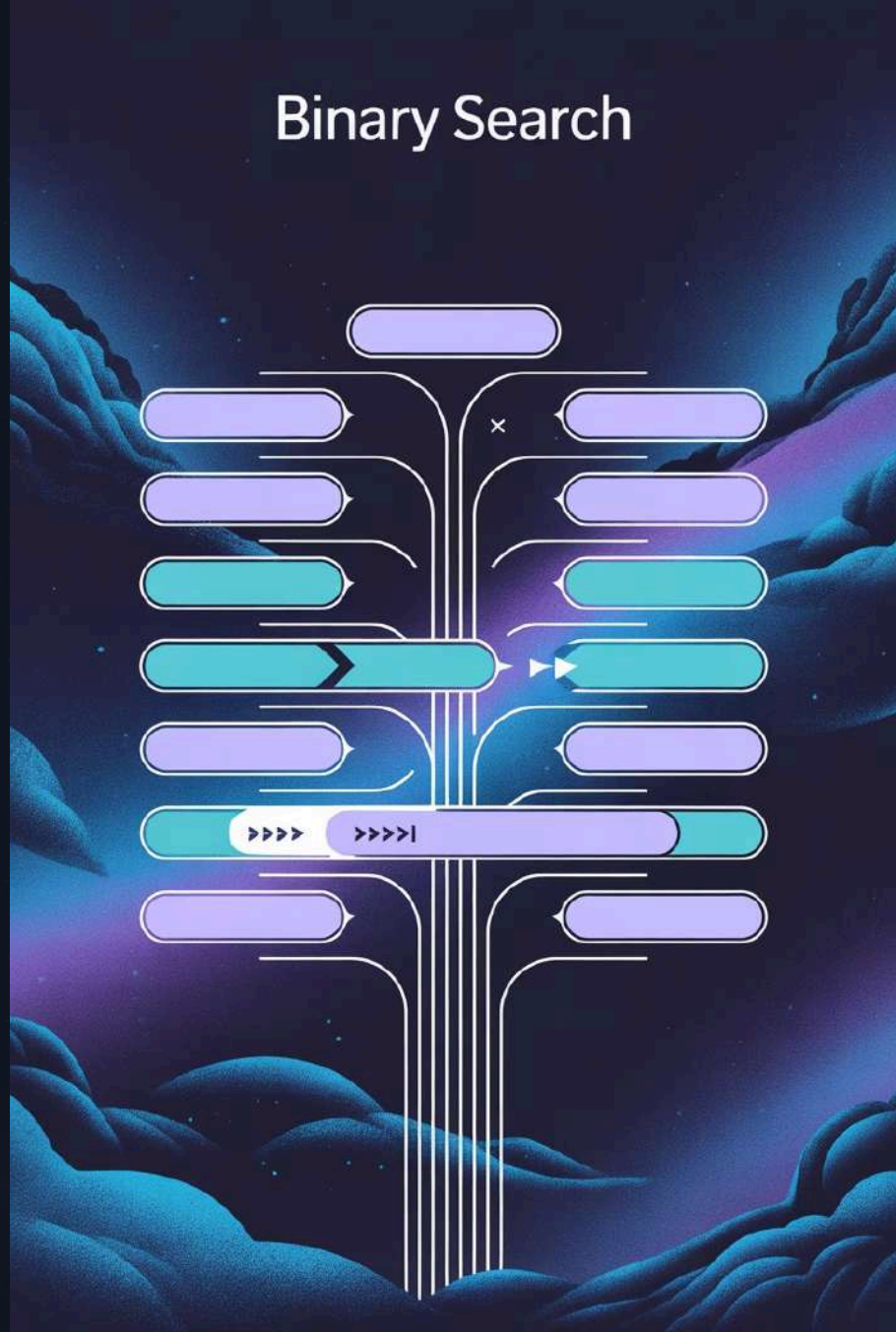
Insertion Algorithm:

1. If the tree is empty, create the root.
2. Compare with the current node.
3. Go left if less.
4. Go right if greater.
5. Repeat until an empty spot is found.

```
Node* insert(Node* root, int key) {
 // Base case: create a new node
 if (!root)
 return new Node(key);

 // Recursively insert into the
appropriate subtree
 if (key < root->key)
 root->left = insert(root->left, key);
 else if (key > root->key)
 root->right = insert(root->right, key);

 return root; // Return the root
}
```



Searching Elements in BST

Searching in a BST utilizes its ordered property to efficiently find elements. At each step, we eliminate half of the tree from consideration.

```
Node* search(Node* root, int key) {
    // Базовые случаи: дерево пусто или элемент найден
    if (!root || root->key == key)
        return root;

    // Если искомое значение меньше - идём влево
    if (key < root->key)
        return search(root->left, key);

    // Иначе идём вправо
    return search(root->right, key);
}
```

1

Start Search

Compare with root

2

Choose Direction

Left or Right

3

Result

Found or Not Found

Tree Traversal

Tree traversal is the systematic visiting of all nodes. Different types of traversals yield different orders of elements.

Pre-order

Root → Left → Right

Used for copying a tree

In-order

Left → Root → Right

Yields sorted values!

Post-order

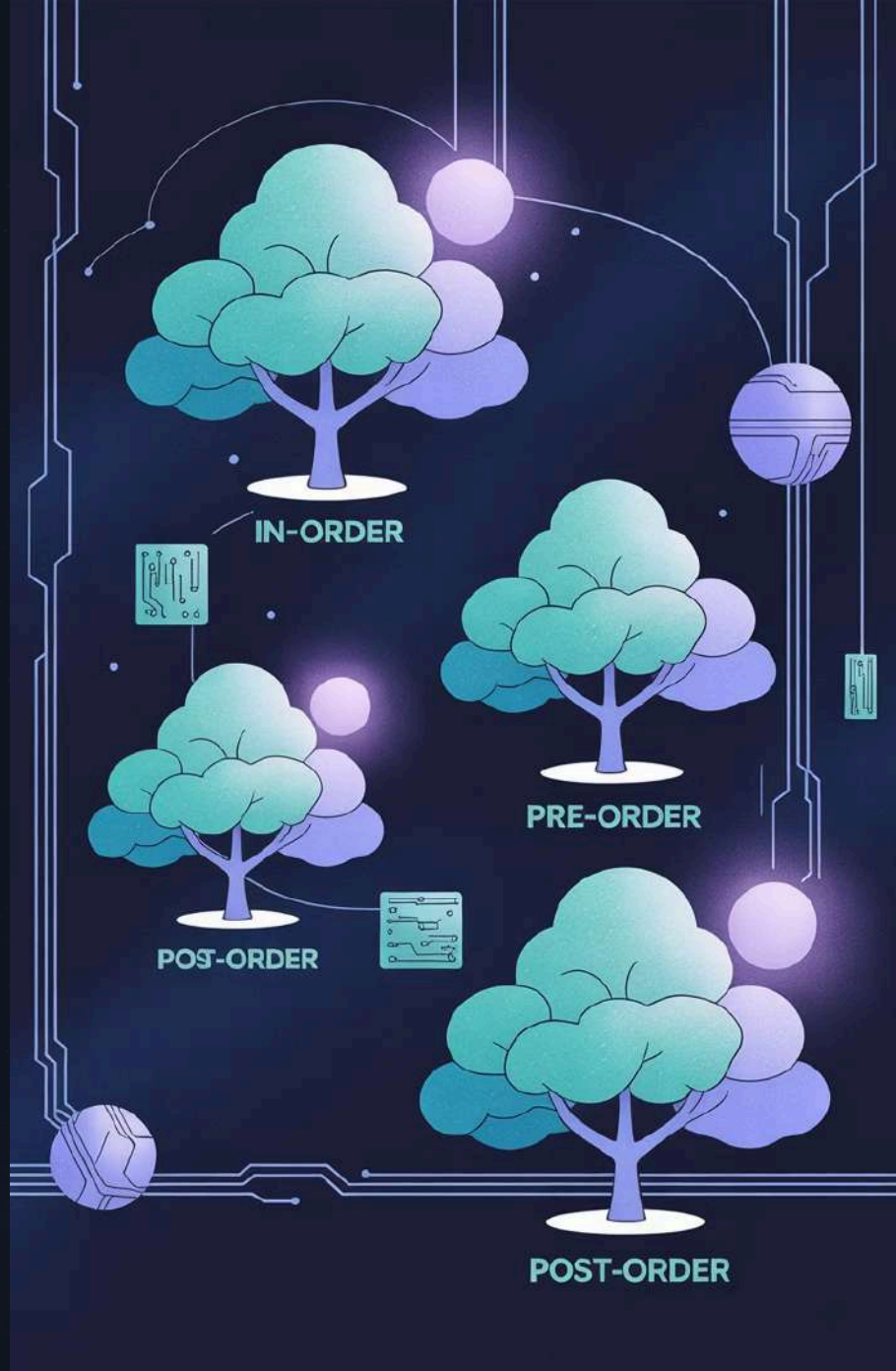
Left → Right → Root

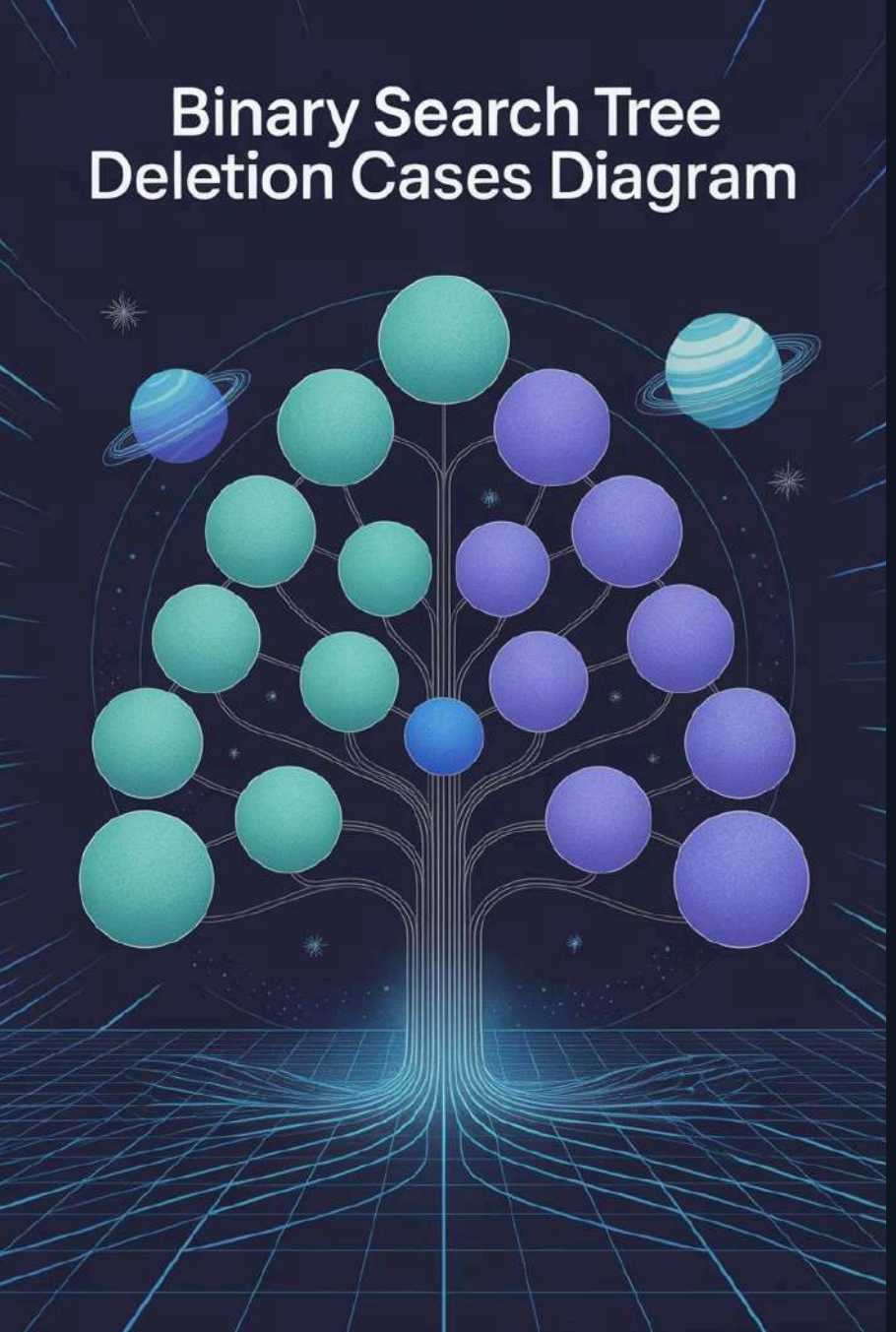
Used for deleting a tree

In-order Traversal Example Implementation:

```
void inorder(Node* root) {
 if (!root) return; // Базовый случай

 inorder(root->left); // Обходим левое поддерево
 cout << root->key << " "; // Выводим текущий узел
 inorder(root->right); // Обходим правое поддерево
}
```





Deleting elements from BST

Deletion is the most complex operation in a BST. It is necessary to preserve the tree's properties after deleting a node.

Case 1: Leaf Node

A node with no children is simply deleted without additional operations.

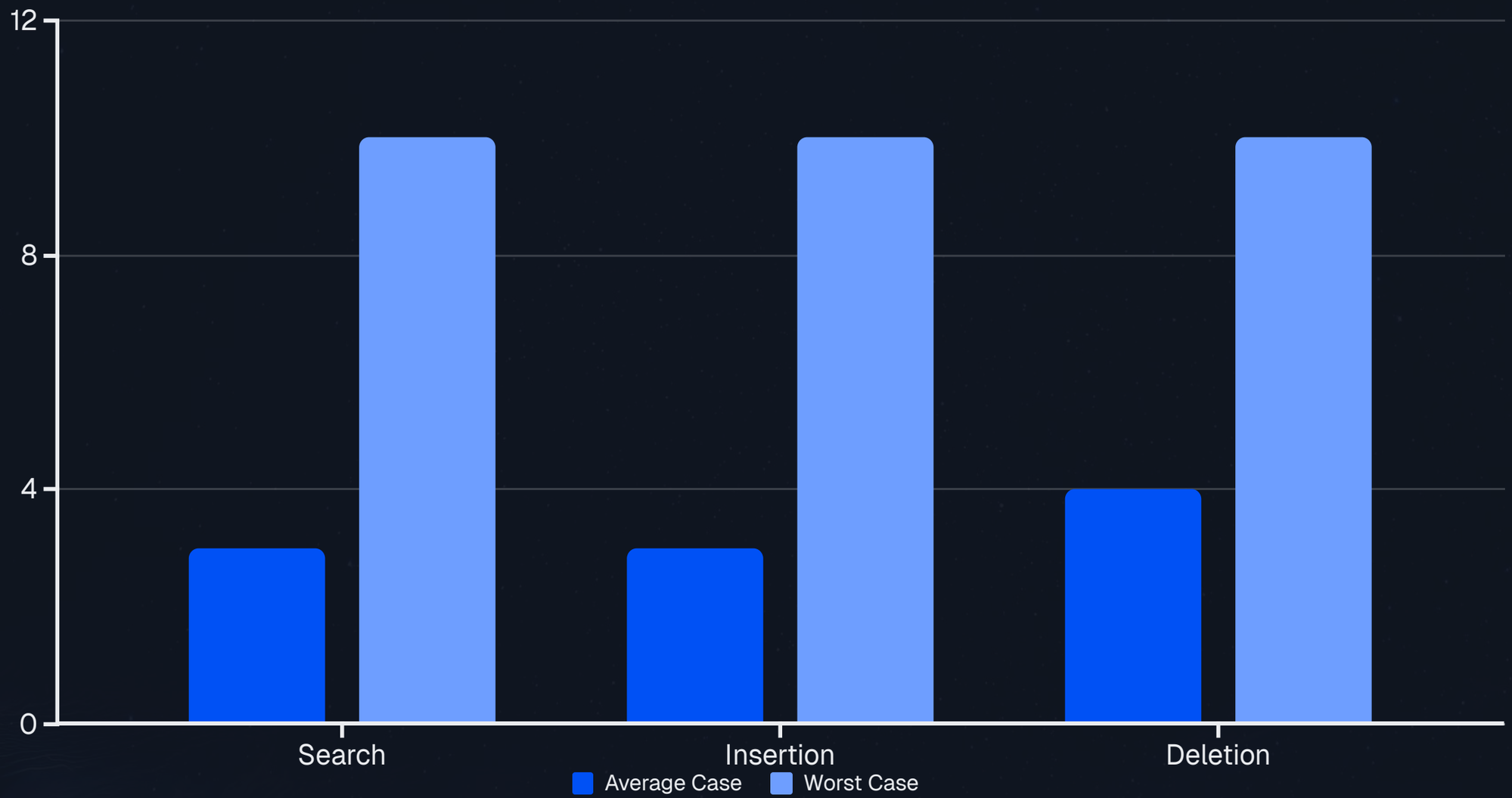
Case 2: One Child

The node is replaced by its only child, and connections are re-established.

Case 3: Two Children

The node is replaced by the minimum element from its right subtree (or maximum from its left).

Analysis of BST Operation Complexity



Average Case: O(log n)

In a balanced tree, the height is proportional to the logarithm of the number of elements.

Worst Case: O(n)

A degenerate tree (similar to a linked list) requires traversing all elements.

Key Takeaways

1

Hierarchical Structure

Trees naturally represent hierarchical relationships between data, making them indispensable in many programming areas.

2

Foundation for Complex Structures

BSTs serve as a foundation for more advanced structures: AVL trees, Red-Black trees, and B-trees are used in real-world systems.

3

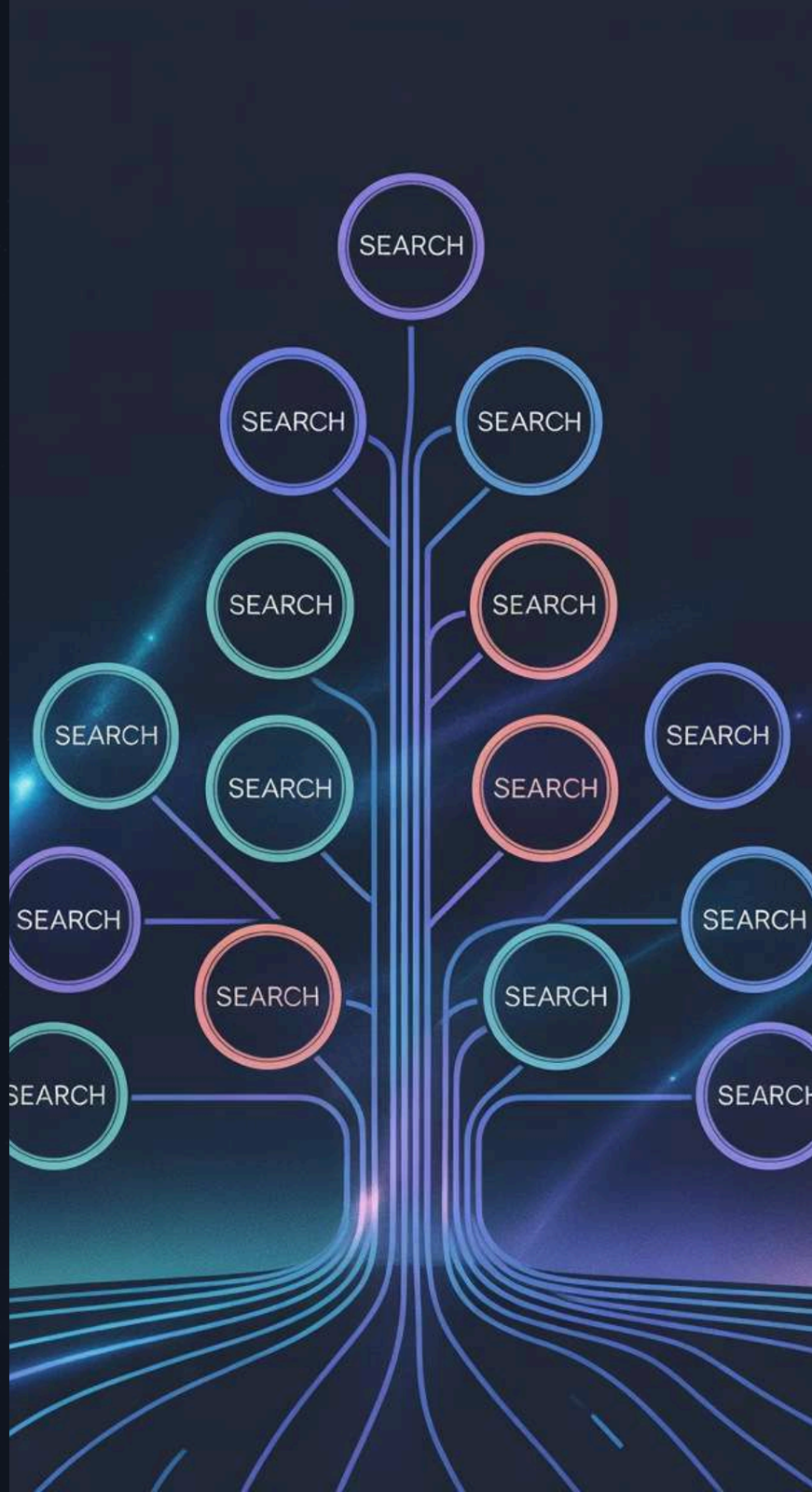
Efficiency of Operations

In the average case, BSTs provide logarithmic time for basic operations, which is significantly better than linear data structures.

4

Importance of Balancing

BST performance is critically dependent on balancing. A degenerate tree loses all advantages over a regular list.



Reinforcement Questions

Question 1

How does a BST differ from a regular binary tree?

Think about the key property of ordered elements in a BST.

Question 2

Which tree traversal method yields sorted data?

Recall the order of node visits in different types of traversals.

Question 3

When does a BST degrade into a linked list?

Consider how the order of insertion affects the tree's structure.

Discuss the answers with colleagues or your instructor. Understanding these questions is key to successfully applying trees in programming!

🎯 What's Next?

Continuing Your Learning

This is just the beginning of your journey into the world of data structures! Understanding trees opens doors to exploring more complex algorithms and structures.

1

Workshop Tasks

Implement a complete BST class with insertion, search, deletion operations, and various traversal methods.

2

Practical Exercises

Measure tree depth, find minimum and maximum elements, verify BST correctness.

3

Further Study

AVL trees, Red-Black trees, B-trees – these are the next steps in mastering data structures.

> "The best way to learn data structures is to implement them yourself!"