

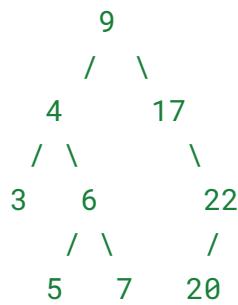
Lab 6

Problem 1

Consider an e-commerce platform where the product prices are organized in a Binary Search Tree (BST). Each node represents a product's price, and the tree is structured based on the prices. As an e-commerce system administrator, your task is to quickly find the product with the price closest to a customer's target price.

Example:

Given the following BST of product prices:



You need to implement an algorithm that finds the product with the minimum absolute price difference to the given target price K .

1. **Input:** Target price $K = 4$
Output: Closest price = 4
2. **Input:** Target price $K = 18$
Output: Closest price = 17
3. **Input:** Target price $K = 2$
Output: Closest price = 3

Task:

1. Describe an approach to solve this problem with $O(h)$ time complexity, where h is the height of the tree.
 2. Walk through the algorithm step by step for the example with the target price $K = 19$ and explain the comparisons made in the tree to find the closest price.
-

Problem 2

A financial auditing system is analyzing a company's past transactions. Each transaction amount is stored as a node in a **Binary Search Tree (BST)**, where the BST structure ensures that the left child of a node contains a smaller amount, and the right child contains a larger amount.

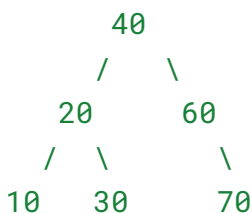
As part of a fraud detection protocol, the auditor wants to know: **Are there two transaction amounts in the BST that add up exactly to a suspicious target amount?**

Task:

Given the root of a Binary Search Tree containing **distinct positive integers** (transaction amounts) and a target sum (suspicious amount), determine whether **there exists a pair of nodes in the BST** whose sum equals the target.

1. **Explain an efficient algorithm** to solve this problem in $O(n)$ time using BST properties.
2. **Step-by-step walkthrough:** Use the BST below and target = 90. Show how the algorithm works.

Example tree:



Problem 3

Imagine an e-commerce platform that manages a large and dynamic product catalog where the prices of the products are frequently updated (products are added or removed). Initially, the platform stores these prices in a **sorted array** for efficient querying of individual product prices using **binary search** in $O(\log n)$ time. While the sorted array is efficient for searching individual prices, the platform faces performance challenges with **frequent insertions and deletions**. Specifically, maintaining the sorted order during each update operation (insertion or deletion) in a sorted array requires shifting elements, leading to $O(n)$ time complexity for each operation.

To optimize the performance of the platform's product catalog, the platform needs to **convert the sorted array into a Balanced Binary Search Tree (BST)**. This will allow the platform to perform **insertions and deletions** efficiently in $O(\log n)$ time, while maintaining fast querying capabilities for product prices.

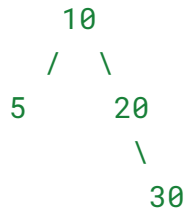
Task:

1. Explain an efficient algorithm to convert a sorted array into a Balanced Binary Search Tree in $O(n)$ time.
 2. **Step-by-step walkthrough:** Show how the algorithm works with the array [10, 20, 30, 40, 50, 60, 70]
-

Problem 4

In an e-commerce platform, product listings are managed using an **AVL tree** based on product **ID**. The tree must remain balanced after every **insertion** and **deletion** to ensure efficient searching and updating.

The initial AVL tree is shown below:

**Tasks:**

1. Given the following sequence of insertions:
 - a. **Insert Product ID = 25**
 - b. **Insert Product ID = 35**

After each insertion, analyze the AVL tree balance and identify any rotations needed. Explain the reason for the rotation (left/right, single/double)

2. Now, delete the following products:
 - a. **Delete Product ID = 35**
 - b. **Delete Product ID = 30**

After each deletion, analyze the tree's balance and identify any required rotations. Justify the rotation based on the AVL property violated.

Problem 5

Consider an e-commerce platform that stores product data in a **Binary Search Tree (BST)** based on product prices. To optimize **product recommendations** for the **highest-priced**

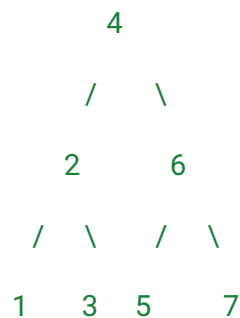
products, the platform plans to **convert the BST into a Special Max Heap**. This conversion allows for **efficient retrieval of top-priced products** by maintaining the **Max Heap property**, where the largest element is always at the root.

The platform also wishes to maintain the **BST property** within each subtree, ensuring that all values in the left subtree are smaller than those in the right subtree. This **Special Max Heap** structure balances **fast access to the highest-priced products** while preserving an **ordered structure** for efficient searching and querying within the tree.

Task:

1. Explain an efficient algorithm to convert a Binary Search Tree (BST) into a **Special Max Heap** in $O(n)$ time.
2. **Step-by-step walkthrough:** Show how the algorithm works with the below BST

Example BST:



Problem 6

In a **customer support center**, each incoming request is labeled with a **type**, represented by lowercase letters (e.g., 'b' for billing, 't' for technical). Some request types occur more frequently than others. To **reduce operator fatigue** and **optimize efficiency**, the system must be configured so that **no two requests of the same type are handled consecutively**.

Given a string `s`, where each character represents a request type, your task is to **rearrange the characters** so that **no two adjacent characters are the same**. If such an arrangement is **not possible**, return an **empty string** (`""`).

To ensure performance, your algorithm must run in $O(n \log n)$ time, where `n` is the length of the string.

Example

Input: `s = "aaabbc"`

Output: "ababac"

Tasks:

1. **Describe an algorithm** to solve this problem that runs in $O(n \log n)$ time.
2. **Demonstrate your approach** with a step-by-step explanation for the input $s =$ "aaabbc".

***Hint:** Think about how to always select the most frequent request types first, while temporarily holding back recently used types.*