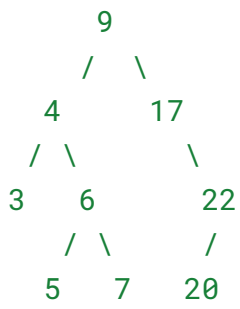
Lab 6

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.

Answer

Approach to Solve the Problem in O(h) Time Complexity:

To find the closest price, we need to traverse the tree and compare the target price `K` with the node's price at each step. At each node, we will track the closest value found so far. Since the BST structure helps us eliminate half of the tree in each comparison, this approach works in **O(h)** time, where `h` is the height of the tree.

Step-by-Step Algorithm:

1. **Start with the root node.** Initialize a variable `closest` to store the closest value to the target price found so far. Initially, set `closest` to the root's value.

2. **Traverse the tree.**
   - While the current node is not `null`, compare the absolute difference between the target price `K` and the current node's price with the absolute difference between `K` and the closest price found so far.
   - If the current node's value is closer to `K` than the `closest` value, update `closest` to the current node's value.

3. **Move to the left or right subtree.**
   - If the current node's price is greater than `K`, move to the left child, as the left subtree contains smaller values.
   - If the current node's price is smaller than `K`, move to the right child, as the right subtree contains larger values.
   - If the current node's price is equal to the target price `K`, stop the search immediately as we have found the exact match.

4. **Repeat the process** until we reach a `null` node or find a node with an exact match (i.e., the price equals `K`).

5. **Return the closest value** found after completing the traversal.

Example Walkthrough: Target Price K = 19

Step 1:

- We start at the root, which has a price of `9`.
- The absolute difference between `9` and `19` is `10`, so we initialize `closest = 9`.

Step 2:

- Since `19 > 9`, we move to the right child (value `17`).
- The absolute difference between `17` and `19` is `2`, which is smaller than the previous difference of `10`. We update `closest = 17`.

Step 3:

- Since `19 > 17`, we move to the right child (value `22`).
- The absolute difference between `22` and `19` is `3`, which is larger than the previous difference of `2`, so we do not update `closest`.

Step 4:

- Since `19 < 22`, we move to the left child (value `20`).
- The absolute difference between `20` and `19` is `1`, which is smaller than the previous difference of `2`. We update `closest = 20`.

Step 5:

- Since `19 < 20`, we move to the left child, which is `null`. The traversal stops here.

Step 6:

- The closest price found is `20`.

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.

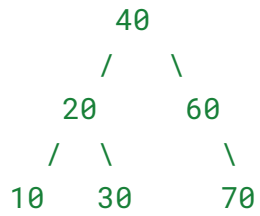
As part of a fraud detection protocol, the auditor wants to know: **Are there two transaction amounts in the BST that add up exactly to a suspicious target amount?**

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:



Answer

Algorithm Approach

1. **Inorder Traversal of the BST:**
    - Since the BST follows the property where the left subtree contains smaller values and the right subtree contains larger values, an **inorder traversal** will give us the values of the tree in **ascending order**.
    - This gives us a sorted array of transaction amounts, and we can use the **two-pointer technique** to find two values whose sum is equal to the target.
2. **Two-Pointer Technique on Sorted Array:**
    - Once we have the sorted array (from the inorder traversal), we can use two pointers: one starting at the beginning of the array and the other starting at the end.
    - The two-pointer technique works because:
        - If the sum of the two pointers is **less than the target**, we need a larger sum, so we move the left pointer to the right (increase it).
        - If the sum is **greater than the target**, we need a smaller sum, so we move the right pointer to the left (decrease it).
        - If the sum equals the target, we have found the two values that sum to the target and can return `True`.
3. **Algorithm Complexity:**
    - **Time Complexity: O(n)**, where **n** is the number of nodes in the BST. This is because:
        - We perform **inorder traversal** once, which takes **O(n)** time.
        - The **two-pointer technique** also runs in **O(n)** time on the sorted array.

Step-by-Step Walkthrough Using the Given BST and Target = 90:

Step 1: Perform Inorder Traversal

We start by performing an inorder traversal of the BST, which will give us the node values in sorted order.

- **Inorder traversal steps:**
    - Traverse left subtree of root (40), which is node 20.
    - Traverse left subtree of node 20, which is node 10.
    - Visit node 10.
    - Visit node 20.
    - Traverse right subtree of node 20, which is node 30.
    - Visit node 30.
    - Visit root node 40.
    - Traverse right subtree of root node 40, which is node 60.
    - Visit node 60.
    - Traverse right subtree of node 60, which is node 70.
    - Visit node 70.

Result of Inorder Traversal:
The sorted array obtained is:
`[10, 20, 30, 40, 60, 70]`

Step 2: Use Two-Pointer Technique to Find the Pair

Now that we have the sorted array `[10, 20, 30, 40, 60, 70]`, we can apply the two-pointer technique to find if there exists a pair whose sum is **90**.

- **Initial Pointers:**
    1. `left = 0` (points to `10`)
    2. `right = 5` (points to `70`)

- **Step-by-step Process:**
    1. **Check the sum of `left` and `right`:**

- `currentSum = 10 + 70 = 80`
- Since `80 < 90`, move the `left` pointer to the right (`left = 1`).

2. **Check the sum of `left` and `right`:**
   - `currentSum = 20 + 70 = 90`
   - **Match found!** The sum of `20` and `70` equals **90**. We return `True`.

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]

Answer

Approach:

The key idea is to use **recursion** to traverse the array and create a BST:

1. **Find the Middle Element:** The middle element of the array is selected as the root node. This guarantees that the tree will be balanced, as it divides the array into two roughly equal halves.

2. **Recursively Build the Left and Right Subtrees:**

   - The left half of the array becomes the left subtree of the root.

    - The right half of the array becomes the right subtree of the root.

3. **Base Case:** The recursion stops when the range of the array becomes invalid (i.e., when `start > end`), indicating that no more nodes need to be created.

Steps:

1. **Set the middle element** of the array as the root.
2. **Recursively apply the same operation** to the left half of the array to create the left child of the root.
3. **Recursively apply the same operation** to the right half of the array to create the right child of the root.

Step-by-Step Walkthrough Using the Array `[10, 20, 30, 40, 50, 60, 70]`:

Step 1: Initial Array:

```
[10, 20, 30, 40, 50, 60, 70]
```

- The **middle element** is `40` (index 3). This becomes the root of the tree.

Tree so far:

```
  40
```

Step 2: Create Left Subtree:

- The left half of the array is `[10, 20, 30]`.
- The **middle element** of this subarray is `20` (index 1), so `20` becomes the left child of `40`.

Tree so far:

```
  40
  /
20
```

- Now, the left half of `[10, 20, 30]` is `[10]`, and the middle element is `10`. So, `10` becomes the left child of `20`.

- The right half of `[10, 20, 30]` is `[30]`, and the middle element is `30`. So, `30` becomes the right child of `20`.

Tree so far:

```
    40
    /
   20
  /  \
10   30
```

Step 3: Create Right Subtree:

- The right half of the array is `[50, 60, 70]`.
- The **middle element** is `60` (index 5), so `60` becomes the right child of `40`.

Tree so far:

```
     40
    /  \
  20    60
  / \   /
10  30 50
```

- The left half of `[50, 60, 70]` is `[50]`, and the middle element is `50`. So, `50` becomes the left child of `60`.
- The right half of `[50, 60, 70]` is `[70]`, and the middle element is `70`. So, `70` becomes the right child for 60.

Final Tree:

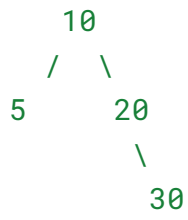
```
     40
    /  \
   20   60
  /  \  /  \
10   30 50  70
```

https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/

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)

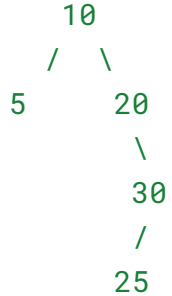
2. Now, delete the following products:
    a. **Delete Product ID = 35**
    b. **Delete Product ID = 30**

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

Answer

Insert Product ID = 25

1. Insert **25** as the left child of **30**:



2. **Balance Factor Analysis:**
    - Node **30**: Left subtree height = 1, Right subtree height = 0 → **Balance Factor = +1** (Balanced)
    - Node **20**: Left subtree height = 0, Right subtree height = 2 → **Balance Factor = -2** (**Unbalanced!**)
3. **Rotation Required:**
    - This is a **Right-Left (RL) case** because **25** was inserted into the left subtree of **30**, which is the right child of **20**.
    - **Step 1:** Perform **Right Rotation (R)** on **30**.
    - **Step 2:** Perform **Left Rotation (L)** on **20**.
4. **After RL Rotation:**

```
   10
   /  \
5     25
     /  \
   20    30
```

Now, the tree is balanced.

Insert Product ID = 35

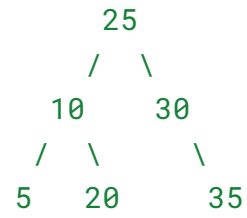
1. Insert **35** as the right child of **30**:

```
    10
   /  \
5     25
     /  \
   20    30
           \
            35
```

2. **Balance Factor Analysis:**

- Node **30**: Left subtree height = 0, Right subtree height = 1 → **Balance Factor = -1** (Balanced)
- Node **25**: Left subtree height = 1, Right subtree height = 2 → **Balance Factor = -1** (Balanced)
- **Node 10**: Left subtree height = 1, Right subtree height = 3 → **Balance Factor = -2** (**Unbalanced!**)

3. **Rotation Required:**
   - This is a **Right-Right (RR) case** because **35** was inserted into the right subtree of **25**, which is the right subtree of **10**.
   - **Perform Left Rotation (L) on 10**.
4. **After Left Rotation (L) on 10:**

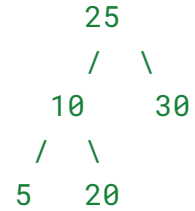


Now, the tree is balanced.

Deletion Operations:

Delete Product ID = 35

1. **Remove 35** (Leaf node):



2. **Balance Factor Analysis:**

- Node **30**: Left subtree height = 0, Right subtree height = 0 → **Balance Factor = 0** (Balanced)
- Node **25**: Left subtree height = 2, Right subtree height = 1 → **Balance Factor = +1** (Balanced)

Since the tree remains balanced, **no rotations are required**.

Delete Product ID = 30

1. **Remove 30**, leading to:

```
    25
    /
  10
 /  \
5   20
```

2. **Balance Factor Analysis:**
   - Node **25**: Left subtree height = 2, Right subtree height = 0 → **Balance Factor = +2** (**Unbalanced!**)
   - Node **10**: Left subtree height = 1, Right subtree height = 1 → **Balance Factor = 0** (Balanced)
3. **Rotation Required:**
   - This is a **Left-Left (LL) case** because **10** is in the left subtree of **25**, and the imbalance is caused by the left subtree being too tall.
   - **Perform Right Rotation (R) on 25**.
4. **After Right Rotation (R) on 25:**

```
   10
  /  \
 5    25
     /
   20
```

Now, the tree is balanced.

Problem 5

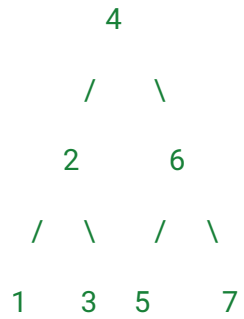
Consider an e-commerce platform that stores product data in a **Binary Search Tree (BST)** based on product prices. To optimize **product recommendations** for the **highest-priced products**, the platform plans to **convert the BST into a Special Max Heap**. This conversion allows for **efficient retrieval 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:



Answer

Algorithm Approach to Convert a BST into a Special Max Heap

The process of converting a **Binary Search Tree (BST)** into a **Special Max Heap** follows a structured approach to ensure that the resulting tree satisfies both:

1. The **Max Heap property**, where each parent node is greater than its children.
2. The **BST structure in its subtrees**, where the left child is smaller than the right child.

Step 1: Perform Inorder Traversal to Store BST Elements

- **Inorder traversal (Left → Root → Right)** is performed on the BST.

- This traversal visits nodes in ascending order and stores their values in an auxiliary array (`arr[]`).
- The array `arr[]` now contains all BST elements in sorted order.
- This ensures that when we later assign values back to the BST, we preserve the order needed for the heap.

Step 2: Perform Postorder Traversal to Reassign Values

- **Postorder traversal (Left → Right → Root)** is used to modify the BST while maintaining heap properties.
- During this traversal, each node is assigned a value from `arr[]`, starting from the smallest to the largest.
- The traversal order ensures that child nodes are processed **before** their parent, helping maintain the heap property.

Step 3: Preserve the Heap Structure

- By using postorder traversal, values are assigned such that every parent node has a greater value than its children.
- Since `arr[]` contains sorted values, assigning them sequentially during postorder traversal ensures that:
    - Parent nodes always receive larger values than their children.
    - The resulting tree maintains the **Max Heap property**.

Step 4: Maintain BST Subtree Order

- While converting the BST into a Max Heap, the tree structure itself is not altered.
- This means that in each subtree, the left child remains smaller than the right child, thus preserving the BST property within its subtrees.

Step-by-Step Walkthrough

Step 1: Perform Inorder Traversal

Inorder traversal of BST yields a **sorted array**: `arr = [1, 2, 3, 4, 5, 6, 7]`

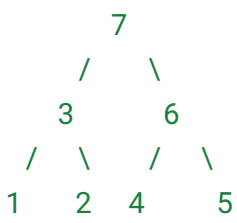
Step 2: Modify BST in Postorder Traversal

Now, we traverse the tree in **postorder (left → right → root)** and replace values using `arr[]` in order.

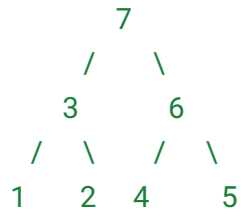
1. Visit left subtree (**1** → **3** → **2**)
2. Visit right subtree (**5** → **7** → **6**)

3. Assign values to the nodes:
   Postorder traversal order: [1, 3, 2, 5, 7, 6, 4]

   Replace nodes with values from arr[]:



Final Max Heap:



Now, the tree satisfies **Max Heap** conditions:

- Every parent is greater than its children.
- The original **BST structure** is maintained within subtrees.

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.*

Answer

Main Idea:

Always choose the character with the **highest frequency** (greedy choice), but ensure that it's **not the same** as the one added in the previous step.

Steps:

1. **Count the frequency** of each character in the string.
2. **Build a max heap** (priority queue) where each element is a pair `(-frequency, character)`.
    - We use negative frequency because Python's `heapq` is a **min-heap**, and we need **max-heap behavior**.
3. **Initialize:**
    - An empty result list `res`.
    - A variable `prev` to store the character used in the previous step (with remaining frequency) so that it can be re-added later.
4. **Loop:**
    - While the heap is not empty:
        - Pop the character with the **highest frequency** (call it `curr`).
        - Append it to the result.
        - If `prev` still has remaining frequency, push it back into the heap.
        - Update `prev` to be the current character, with frequency decreased by 1.
5. After the loop, if the length of the result is **equal to the input**, return the joined result. Otherwise, return `""` (not possible).

Step-by-Step Example:

Let's take `s = "aaabbc"`.

Step 1: Count frequencies

`a → 3`
`b → 2`
`c → 1`

Step 2: Build max heap

We store frequencies as negatives for max-heap behavior:

```
[(-3, 'a'), (-2, 'b'), (-1, 'c')]
```

Step 3: Initialize

- `res = []`
- `prev = (0, '')` → No previous character initially

Iteration 1:

- Pop `(-3, 'a')` → Most frequent character
- Append `'a'` to `res`: → `res = ['a']`
- `prev = (-2, 'a')` → Decrease frequency of `'a'` by 1
- Nothing to push back yet (prev count was 0)

Heap: `[(-2, 'b'), (-1, 'c')]`

Iteration 2:

- Pop `(-2, 'b')`
- Append `'b'` to `res`: → `res = ['a', 'b']`

- Push back `prev = (-2, 'a')` → heap becomes `[(-2, 'a'), (-1, 'c')]`
- `prev = (-1, 'b')`

Heap: `[(-2, 'a'), (-1, 'c')]`

Iteration 3:

- Pop `(-2, 'a')`
- Append `'a'` to `res`: → `res = ['a', 'b', 'a']`
- Push back `prev = (-1, 'b')` → heap becomes `[(-1, 'b'), (-1, 'c')]`
- `prev = (-1, 'a')`

Heap: `[(-1, 'b'), (-1, 'c')]`

Iteration 4:

- Pop `(-1, 'b')`
- Append `'b'` to `res`: → `res = ['a', 'b', 'a', 'b']`
- Push back `prev = (-1, 'a')` → heap becomes `[(-1, 'a'), (-1, 'c')]`
- `prev = (0, 'b')` → No more `b` left

Heap: `[(-1, 'a'), (-1, 'c')]`

Iteration 5:

- Pop `(-1, 'a')`
- Append `'a'` to `res`: → `res = ['a', 'b', 'a', 'b', 'a']`
- `prev = (-1, 'c')` → Save to reinsert in next step

Heap: `[(-1, 'c')]`

Iteration 6:

- Pop `(-1, 'c')`
- Append `'c'` to `res`: → `res = ['a', 'b', 'a', 'b', 'a', 'c']`
- `prev = (0, 'c')` → No more `c`

Heap is now empty.

Output: `"ababac"`