BINARY SEARCH TREES

CSE 4303
Data Structures

Hasin Mahtab
210042174
Department of CSE
B.Sc in Software Engineering

December 30, 2023

Contents

1 Detecting Financial Anomalies

Financial experts are working with a cutting-edge stock trading system to uncover probable anomalies in the trading behaviour of specific equities. You are entrusted with creating a function that can recognise triplets of stock trades within a Binary Search Tree (BST) whose total transaction amounts add up to a particular target sum to assist them. The stock trading system maintains a BST of stock trades, where each trade is represented as a node with the following information:

- **Trade ID:** A unique identifier for each trade.
- **Stock Symbol:** The symbol of the traded stock.
- **Transaction Amount:** The monetary value of the trade.

The financial analysts suspect that certain patterns of three trades (triplets) with a specific total transaction amount might indicate irregular trading behavior.

1.1 Input

- **root:** Root of a Binary Search Tree ($1 \leq$ (nodes in the BST) $\leq 10^5$).
- **targetSum:** For which triplets need to be found ($0.0 \leq \text{targetSum} \leq 1e9$).

1.2 Output

- A list of triplets (lists) containing stock symbols whose total transaction amounts add up to the target sum.

1.3 Example

```
TradeNode* tradeTree = new TradeNode("A1", "AAPL", 100.0);
tradeTree->left = new TradeNode("A2", "GOOGL", 150.0);
tradeTree->right = new TradeNode("A3", "AMZN", 200.0);
tradeTree->left->left = new TradeNode("A4", "AAPL", 50.0);
tradeTree->left->right = new TradeNode("A5", "GOOGL", 120.0);
tradeTree->right->left = new TradeNode("A6", "AAPL", 180.0);
tradeTree->right->right = new TradeNode("A7", "MSFT", 300.0);

double targetSum = 500.0;
```

The expected **output** for this example would be:

```
Triplets with target sum 500:
[AAPL GOOGL MSFT ]
[AAPL AMZN MSFT ]
```

1.3.1 Explanation

- The first triplet [AAPL, GOOGL, MSFT] consists of trades with transaction amounts 100.0, 120.0, and 300.0, which sum up to the target sum of 500.0.
- The second triplet [AAPL, AMZN, MSFT] consists of trades with transaction amounts 50.0, 200.0, and 300.0, which also sum up to the target sum of 500.0.

2 Binary Search Tree

2.1 Introduction

A binary search tree is either an empty data structure represented by nullptr or a single node x, whose left subtree is a BST of smaller values than x and whose right subtree is a BST of larger values than x. This special property called the BST-property, which is given as follows:

- For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x.

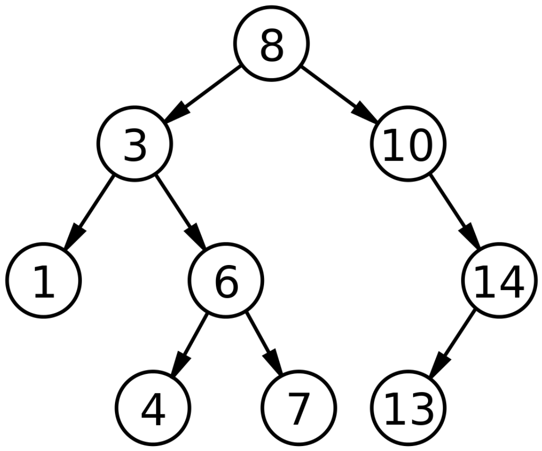

Figure 1: A simple binary search tree

In the Binary search tree, The values in the left subtree are all smaller than the root 8, whereas the values in the right subtree are all greater. Each node has the following attributes:

- p, left, and right, which are pointers to the parent, the left child, and the right child, respectively.
- key, which is key stored at the node.

2.2 Successor and predecessor

For certain operations, given a node x , finding the successor or predecessor of x is crucial. Assuming all the keys of the BST are distinct, the successor of a node x in BST is the node with the smallest key greater than x 's key. On the other hand, the predecessor of a node x in BST is the node with the largest key smaller than x 's key. Following is pseudocode for finding the successor and predecessor of a node x in BST.

2.3 Operations

Binary search trees allow binary search for fast lookup, addition, and removal of data items. Since the nodes in a BST are laid out so that each comparison skips about half of the remaining tree, the lookup performance is proportional to that of binary logarithm.

2.3.1 Traversal

We define "traversal" as visiting all nodes in a graph. The ordering of the three items to visit: the current node, the left subtree, and the right subtree, can be used to specify traversal methods. Then there are three options.

- Inorder : Left Subtree $\rightarrow$ Parent Node $\rightarrow$ Right Subtree
- Preorder : Parent Node $\rightarrow$ Left Subtree $\rightarrow$ Right Subtree
- Postorder : Left Subtree $\rightarrow$ Right Subtree $\rightarrow$ Parent Node

2.3.2 Insertion

By retaining the binary search tree's property, a new key is always put at the leaf. We begin our search for a key at the root and go until we reach a leaf node. When a leaf node is discovered, the new node is inserted as a child of that node.

When looking for a place to insert a new key, traverse the tree from root to leaf, comparing keys stored in the tree's nodes and selecting whether to continue searching in the left or right subtrees based on the comparison. In other words, we evaluate the root and recursively insert the new node to the left subtree if its key is less than the root's key or to the right subtree if its key is larger than or equal to the root's key.

Following is the Insert function implementation of the above approach in C++

```
Node* insert(Node* root, int key)
{
    // if the root is null, create a new node and return it
    if (root == nullptr) {
        return new Node(key);
    }

    // if the given key is less than the root node, recur for the left
         subtree
    if (key < root->data) {
        root->left = insert(root->left, key);
    }
    // if the given key is more than the root node, recur for the
        right subtree
    else {
        root->right = insert(root->right, key);
    }

    return root;
}
```

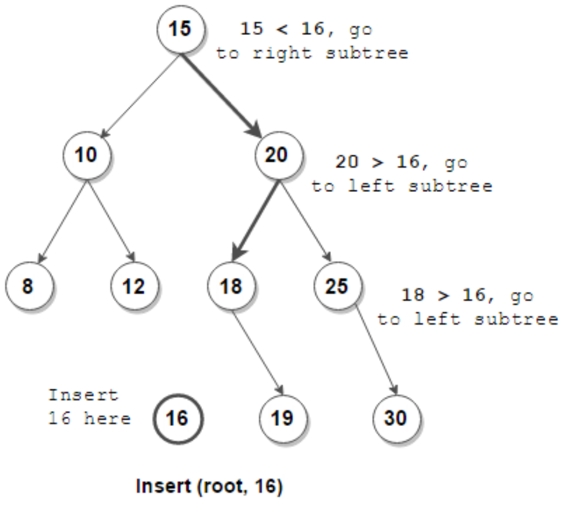
15 < 16, go to right subtree
20 > 16, go to left subtree
18 > 16, go to left subtree
Insert 16 here
Insert (root, 16)

Figure 2: Inserting 16 in BST tree

2.3.3 Searching

Searching for a given key in a binary search tree can be done recursively or iteratively.

The search begins with an examination of the root node. The key being searched for does not exist in the tree if the tree is null. Otherwise, the search is successful and the node is returned if the key matches that of the root. If the key is smaller than the root, the search advances to the left subtree. Similarly, if the key is greater than the root, the search moves on to the appropriate subtree. This process is done until either the key or the remaining subtree is located.

Following is the Search function implementation of the above approach in C++

```cpp
void search(Node* root, int key, Node* parent)
{
    // if the key is not present in the key
    if (root == nullptr)
    {
        cout << "Key not found";
        return;
    }

    // if the key is found
    if (root->data == key)
    {
        if (parent == nullptr) {
            cout << "The node with key " << key << " is root node";
        }
        else if (key < parent->data)
        {
            cout << "The given key is the left node of the node with key "
                << parent->data;
        }
        else  {
            cout << "The given key is the right node of the node with key "
                << parent->data;
        }

        return;
    }
```

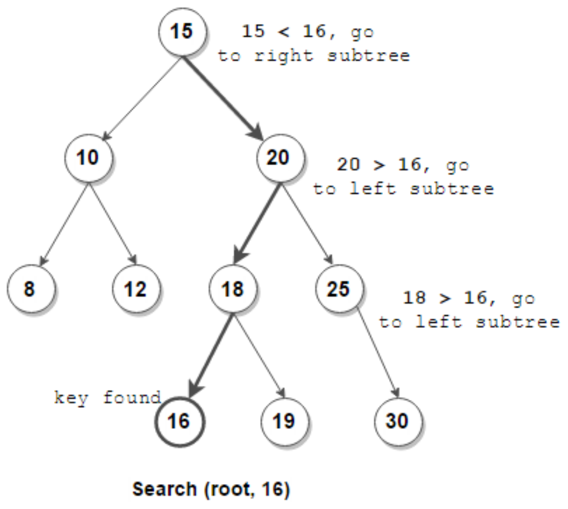

Figure 3: Searching 16 in BST tree

2.3.4 Deletion

Given a BST, the task is to delete a node in this BST, which can be broken down into 3 scenarios:

- **Deleting a node with no children:** Remove the node from the tree.

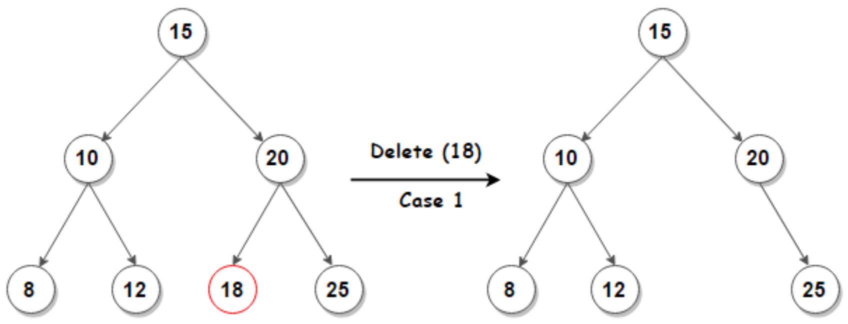

Figure 4: Searching 16 in BST tree

- **Deleting a node with two children:** Call the node to be deleted N. Do not delete N. Instead, choose either its inorder successor node or its inorder predecessor node, R. Copy the value of R to N, then recursively call delete on R until reaching one of the first two cases. If we choose the inorder successor of a node, as the right subtree is not NULL (our present case is a node with 2 children), then its inorder successor is a node with the least value in its right subtree, which will have at a maximum of 1 subtree, so deleting it would fall in one of the first 2 cases.

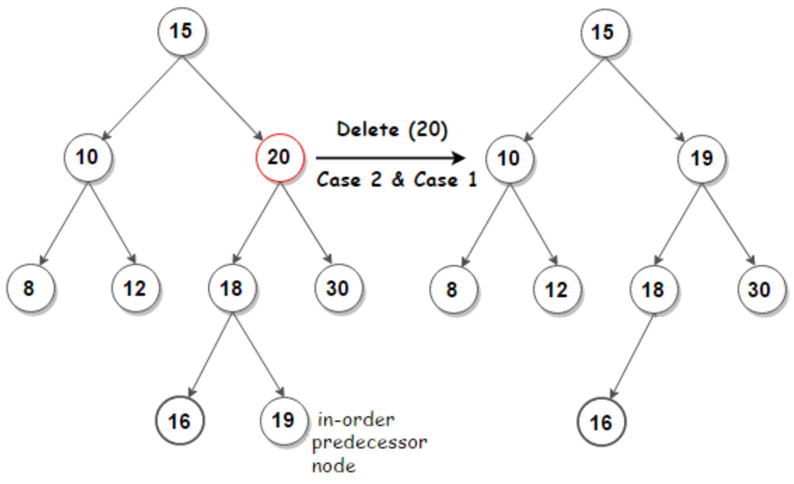

Figure 5: Searching 16 in BST tree

- **Deleting a node with one child:** Remove the node and replace it with its child.

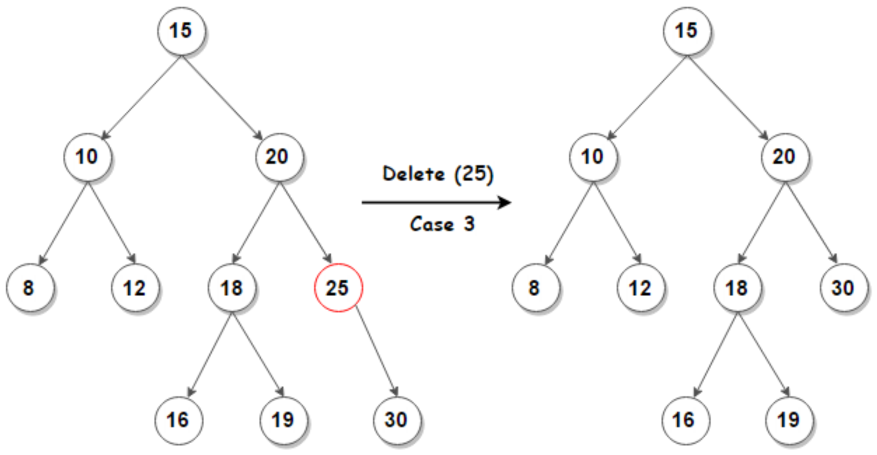

Figure 6: Searching 16 in BST tree

Nodes with children are harder to delete. As with all binary trees, a node's inorder successor is its right subtree's leftmost child, and a node's inorder predecessor is the left subtree's rightmost child. In either case, this node will have zero or one child. Delete it according to one of the two simpler cases above.

3 Problem Analysis

We are provided a Binary Search Tree (BST) representing stock trades in this case. In the BST, each trade is represented as a node, with information such as the trade ID, stock symbol, and transaction amount. The purpose is to find triplets of stock trades whose total transaction values add up to a certain amount.

3.1 Intuition behind the solution

3.1.1 BST Traversal

We need to traverse the given BST to explore all possible triplets. The traversal can be done using a recursive approach, visiting each node in a depth-first manner.

3.1.2 Triplet Detection

- While traversing the BST, we keep track of the current triplet and a set of visited stock symbols within the current triplet.
- When a node is visited, we check if its stock symbol is already in the set of visited symbols. If it is, we remove it from the set (backtracking).

- We also check if the current triplet is complete. If it is, we calculate the sum of transaction amounts and check if it equals the target sum.
- If a valid triplet is found, it is added to the result.

3.1.3 Backtracking

Backtracking is essential to explore all possible combinations of triplets and to ensure that the set of visited symbols is appropriately updated.

3.2 Solution steps

To solve the problems, let's go step by step -

1. Initialize an empty set visitedStocks
2. Initialize an empty list currentTriplet
3. Initialize an empty list result
4. Start the recursive detection process
5. return result

A simple solution for the process is as follows:

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct TradeNode
{
    string tradeId;
    string stockSymbol;
    double transactionAmount;
    TradeNode *left;
    TradeNode *right;

    TradeNode(string id, string symbol, double amount)
        : tradeId(id), stockSymbol(symbol), transactionAmount(amount),
             left(nullptr), right(nullptr) {}
};

void detectAnomaliesInStockTrades(TradeNode *root, double targetSum,
    vector<string> &currentTriplet, vector<vector<string>> &result)
{
    if (!root)
    {
        return;
    }

    // Check if the current triplet is complete
    if (currentTriplet.size() < 3)
    {
        // Add the stock symbol to the current triplet
        currentTriplet.push_back(root->stockSymbol);
    }
    else
    {
        double tripletSum = 0.0;
        for (const string &symbol : currentTriplet)
        {
            tripletSum += root->transactionAmount;
        }

        if (tripletSum == targetSum)
        {
            // Add the current triplet to the result
            result.push_back(currentTriplet);
        }
    }

    detectAnomaliesInStockTrades(root->left, targetSum, currentTriplet
        , result);
    detectAnomaliesInStockTrades(root->right, targetSum,
        currentTriplet, result);

    currentTriplet.pop_back();
}

vector<vector<string>> findTripletsInBST(TradeNode *root, double
    targetSum)
{
    vector<string> currentTriplet;
    vector<vector<string>> result;

    detectAnomaliesInStockTrades(root, targetSum, currentTriplet,
        result);
    return result;
}

int main()
{
    TradeNode *tradeTree = new TradeNode("A1", "AAPL", 100.0);
    tradeTree->left = new TradeNode("A2", "GOOGL", 150.0);
    tradeTree->right = new TradeNode("A3", "AMZN", 200.0);
    tradeTree->left->left = new TradeNode("A4", "AAPL", 50.0);
    tradeTree->left->right = new TradeNode("A5", "GOOGL", 120.0);
    tradeTree->right->left = new TradeNode("A6", "AAPL", 180.0);
    tradeTree->right->right = new TradeNode("A7", "MSFT", 300.0);

    double targetSum = 500.0;

    vector<vector<string>> anomalies = findTripletsInBST(tradeTree,
        targetSum);

    cout << "Triplets with target sum " << targetSum << ":" << endl;
    for (const vector<string> &triplet : anomalies)
    {
        cout << "[";
        for (const string &symbol : triplet)
        {
            cout << symbol << " ";
        }
        cout << "]" << endl;
    }

    return 0;
}
```

3.3 Complexity Analysis

3.3.1 Time Complexity

The time complexity of the solution depends on the number of nodes in the Binary Search Tree (BST) and the number of valid triplets that satisfy the conditions.

- **Traversal Time:** The depth-first traversal of the BST contributes to the time complexity. In the worst case, we may need to visit all nodes in the BST. **O(N)**, where N is the number of nodes in the BST.
- **Triplet Detection Time:** Checking and detecting triplets involves constant time operations (e.g., updating sets, checking target sum). **O(1)** per node.
- **Backtracking:** Backtracking is performed when a stock symbol is revisited within a triplet. The set of visited symbols is updated. **O(1)** per backtracking step.

3.3.2 Space Complexity

The space complexity of the solution is influenced by the memory required for recursive calls, the storage of visited symbols, and the depth of the recursion stack.

- **Recursive Call Stack:** The space required for the recursive call stack depends on the height of the BST. In the worst case, the tree is skewed, leading to a height of N. **O(N)**, where N is the height of the BST.

- **Visited Symbols Set:** The set of visited symbols stores distinct stock symbols within a triplet. **O(1)** per node (constant number of symbols).
- **Current Triplet Storage:** The current triplet vector stores stock symbols temporarily during the recursive exploration. **O(1)** per node (constant number of symbols).
- **Auxiliary Variables:** Additional auxiliary variables and function parameters contribute to constant space usage. **O(1)**.

4 Applications of BST

Due to the BST property, BSTs allow for efficient searching by repeatedly dividing the search space in half, which makes it an important data structure in computer science and many other fields.

4.1 Efficient to use BST

- A BST can be used to sort a large dataset. By inserting the elements of the dataset into a BST and then performing an in-order traversal, the elements will be returned in sorted order.
- BSTs are used to implement Huffman coding algorithm. While binary trees are involved in the Huffman coding algorithm, they are not necessarily implemented as BSTs.
- BSTs can be used to implement symbol tables, which are used to store data such as variable and function names in a programming language.
- In the case of databases, an index is often implemented as a data structure that maps the values of one or more columns to the corresponding records in the database table. A Binary Search Tree is a type of data structure that is well-suited for this purpose.

4.2 Inefficient to use BST

- They are not well-suited for data structures that need to be accessed randomly, since the time complexity for search, insert, and delete operations is O(log n), which is good for large data sets, but not as fast as some other data structures such as arrays or hash tables.
- A BST can be imbalanced or degenerated which can increase the complexity.

- They are not guaranteed to be balanced, which means that in the worst case, the height of the tree could be O(n) and the time complexity for operations could degrade to O(n).

5 Practice Problems

Some practice problems to apply the knowledge from this article.

5.1 Binary Search Tree

Given a binary tree, your task is to choose a maximal set S of connected nodes in the given tree so that these nodes and the relations (left child, right child, or ancestor) among them form a binary search tree. Codeforces 100499E

5.2 Broken BST

A rooted binary tree where each vertex has at most two children. The tree has one vertex that doesn't have a parent, which is the root. Each vertex has an integer number written on it. Codeforces 797D

5.3 Monk and his Friends

Monk is standing at the door of his classroom. There are currently N students in the class, i'th student got Ai candies. There are still M more students to come. At every instant, a student enters the class and wishes to be seated with a student who has exactly the same number of candies. For each student, Monk shouts YES if such a student is found, NO otherwise. Monk and his Friends

References

- Binary Search Tree - Wikipedia
- C12 - BST, University of Rochester
- Binary Search Tree - GeeksforGeeks
- Operations in BST
- Practice: Binary Search Trees
- Find a triplet with the given sum in a BST