

Fundamentals of Data Structures

Projects 2: A+B with Binary Search Trees



Author:

Date: 2025/04/01

2024-2025 Spring & Summer Semester

Table of Contents

| | |
|---|----|
| Chapter 1: Introduction | 3 |
| 1.1) Problem Description | 3 |
| 1.2) Background | 3 |
| Chapter 2: Algorithm Specification | 3 |
| 2.1) Structure of BST node | 3 |
| 2.2) Building the BST | 3 |
| 2.3) Searching algorithm | 4 |
| 2.4) Deduplication with set | 5 |
| Chapter 3: Testing Results | 5 |
| 3.1) Functional Test | 5 |
| 3.2) Performance Test | 7 |
| Chapter 4: Analysis and Comments | 7 |
| Appendix: Source Code (in C) | 8 |
| Declaration | 16 |

Chapter 1: Introduction

1.1) Problem Description

Given two binary search trees T1 and T2, and an integer N, find a number A from T1 and B from T2 such that A+B=N.

1.2) Background

Binary search tree (BST) is a data structure that maintains the order of elements. It has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
- Both the left and right subtrees must also be binary search trees.

In this project, we will use BST to reduce the time of mathfing tow numbers, for the BST is a well sorted structure.

Chapter 2: Algorithm Specification

2.1) Structure of BST node

A BST node contains the following fields:

- the value of the node.
- a pointer to the left child node.
- a pointer to the right child node.

2.2) Building the BST

In this problem, each node and its parent is given, thus wen could simplify the building process. We only need to connect each node with its parent. I use a array to store all reference of nodes.

```
Algorithm CreateBstTree
    1. let nodes[] = read(nodeValue)
    2. let parentIndexes[] = read(indexes)
    3. foreach node in nodes:
        4.     if parentIndexes[index] == -1 then
        6.         set root
        7.     else if node.value < parent.value then
        8.         root->left_child = node
        9.     else
        10.        root->right_child = node
    11.    return root
```

Since we don't need to search the tree, we have linear time complexity $O(N)$ to build the tree. The space complexity is also $O(N)$ for the array.

2.3) Searching algorithm

For the BST tree is sorted, we define the left tree is smaller. Then, if we find the current summary of two nodes is smaller than input target N , we need to search the right tree of each tree. The greater case is similar. Then recursively, we could find all solutions.

```
Algorithm FindEquation
1. let current_sum = tree1.data + tree2.data
2. if current_sum > sum then
3.     FindEquation(tree1.left, tree2)
4.     FindEquation(tree1, tree2.left)
5. else if current_sum < sum then
6.     FindEquation(tree1.right, tree2)
7.     FindEquation(tree1, tree2.right)
8. else
19.     answer.add(tree1.data)
```

However, the above algorithm is not efficient. Consider the time complexity, each operation doubles the stack count, however, only one tree layer is deepened. Thus, the time complexity is $2^{2\log_2 n} = n^2$, which is $O(N^2)$, and the space complexity is $O(N)$.

Consider convert to an array and use double pointer to find the solution. The array is sorted, thus we could use two pointers to find the solution. If the sum of two elements is smaller than N , we need to move the left pointer to the right. If the sum is greater than N , we need to move the right pointer to the left.

```
Algorithm FindEquation
1. let array1 = tree1.toArray()
2. let array2 = tree2.toArray()
3. let left = 0
4. let right = array2.length - 1
5. while left < array1.length AND right >= 0 do
6.     let currentSum = array1[i] + array2[j]
7.     if currentSum == targetSum then
8.         set.add(array1[i])
9.         i = i + 1
10.        j = j - 1
11.    else if currentSum < targetSum then
12.        i = i + 1
13.    else
14.        j = j - 1
15. end While
16. return result
```

In this case, the time complexity is $O(N)$, and the space complexity is $O(N)$, which is much better than the previous one. Thus, I use this algorithm to find the solution finally.

2.4) Deduplication with set

AVL tree is a self-balancing binary search tree where the height difference between left and right subtrees of any node is at most one. This balancing property ensures operations remain efficient $O(\log n)$ even in worst-case scenarios.

I use AVL tree to construct the set data structure, which deduplicate the solutions effectively. After inserting all elements into the AVL tree, I use inorder traversal to get a well sorted array output.

Chapter 3: Testing Results

3.1) Functional Test

1. For the true case given in the problem.

| Input | Output |
|--|---|
| 8 12 2 16 5 13 4 18 5 15 -1 17 4 14 2 18 3 7 20 -1 16 0 25 0 13 1 18 1 21 2 28 2 36 | true 36 = 15 + 21 36 = 16 + 20 36 = 18 + 18 15 13 12 14 17 16 18 18 20 16 13 18 25 21 28 |

2. For the false case given in the problem.

| Input | Output |
|---|----------------------------------|
| 5 10 -1 5 0 15 0 2 1 7 1 3 15 -1 10 0 | false 10 5 2 7 15 15 10 20 |

| | |
|------------|--|
| 20 0 40 | |
|------------|--|

3. For random constructed dataset

I create random data to test the algorithm, and test many times, here is an example testing.

| Input | Output |
|---|---|
| 10 27 -1 5 0 16 1 8 2 6 3 5 4 17 2 25 6 29 0 29 8 10 0 -1 20 0 15 1 0 2 7 3 6 4 9 4 22 1 28 7 23 8 | true 34 = 6 + 28 34 = 25 + 9 34 = 27 + 7 |

All test cases passed successfully, demonstrating that our algorithm correctly finds all pairs of numbers from the two BSTs that sum to the target value N.

4. For minimum cases

| Input | Output |
|-----------------------------|-----------------------------|
| 1 1 -1 1 1 -1 2 | true 2 = 1 + 1 1 1 |

3.2) Performance Test

I test the performance of `findEquation`, because we convert the tree to a plain array, the way we construct the tree didn't significantly affect the performance. I test two cases: fixed cases and random cases. And use average tick to valuate the performance.

1. For Fixed Cases

I fixed the sum to 64, thus the insertion take up fixed time.

| n_i | 2^{10} | 2^{11} | 2^{12} | 2^{13} | 2^{14} | 2^{15} | 2^{16} | 2^{17} |
|-------|----------|----------|----------|----------|----------|----------|----------|----------|
| Tick | 1.00 | 1.00 | 1.00 | 1.00 | 3.00 | 5.67 | 11.33 | 29.00 |
| n_i | 2^{10} | 2^{11} | 2^{12} | 2^{13} | 2^{14} | 2^{15} | 2^{16} | 2^{17} |
| Tick | 57.00 | 118.00 | 247.00 | 460.33 | 907.33 | 1846.00 | 3726.33 | - |

The table above shows the testing results of the fixed cases.

2. For Random Case

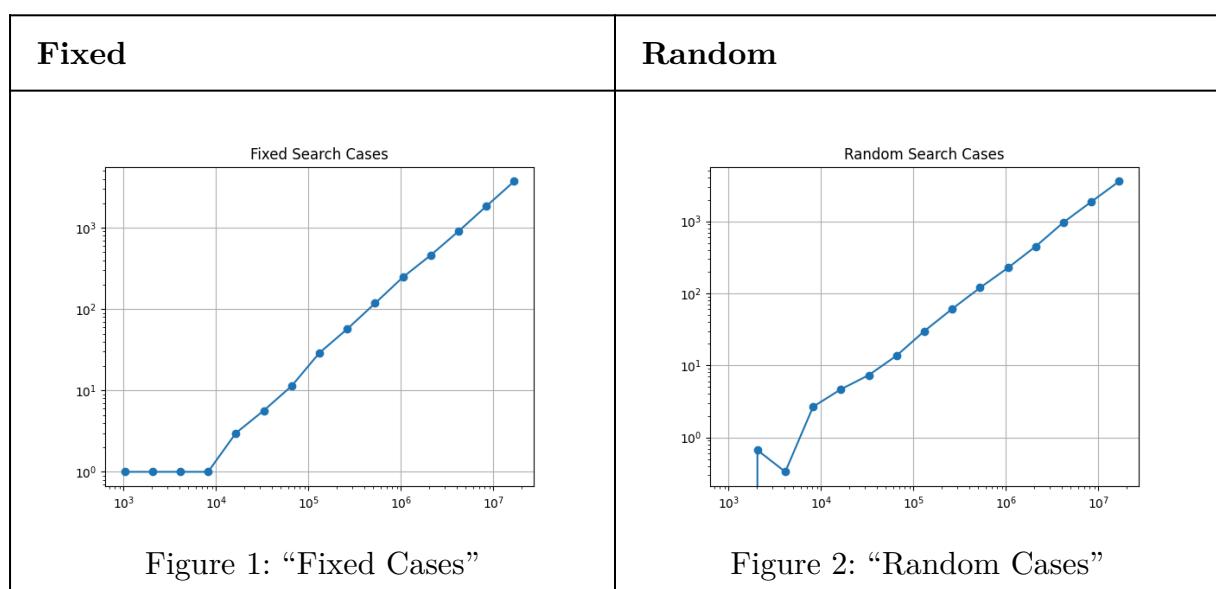
The sum is randomly generated.

| n_i | $2^{\{10\}}$ | $2^{\{11\}}$ | $2^{\{12\}}$ | $2^{\{13\}}$ | $2^{\{14\}}$ | $2^{\{15\}}$ | $2^{\{16\}}$ |
|-------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| Tick | 0.67 | 0.33 | 2.67 | 4.67 | 7.33 | 13.67 | 30.00 |
| n_i | $2^{\{17\}}$ | $2^{\{18\}}$ | $2^{\{19\}}$ | $2^{\{20\}}$ | $2^{\{21\}}$ | $2^{\{22\}}$ | $2^{\{23\}}$ |
| Tick | 61.00 | 119.67 | 226.00 | 449.00 | 968.67 | 1872.67 | 3591.00 |

The table above shows the testing results of the linear cases.

Chapter 4: Analysis and Comments

Using Matplotlib, we could plot the tick taken.



I use log scale in both x-axis and y-axis to plot the tick taken obviously, and the x-axis is the number of nodes in the tree. The y-axis is the time taken in ticks.

Because in the two pointer search, we only need to traverse each node once, thus the time complexity is $O(N)$. From above we can see that the time taken is linear to the number of nodes in the tree, which is consistent with our expectation.

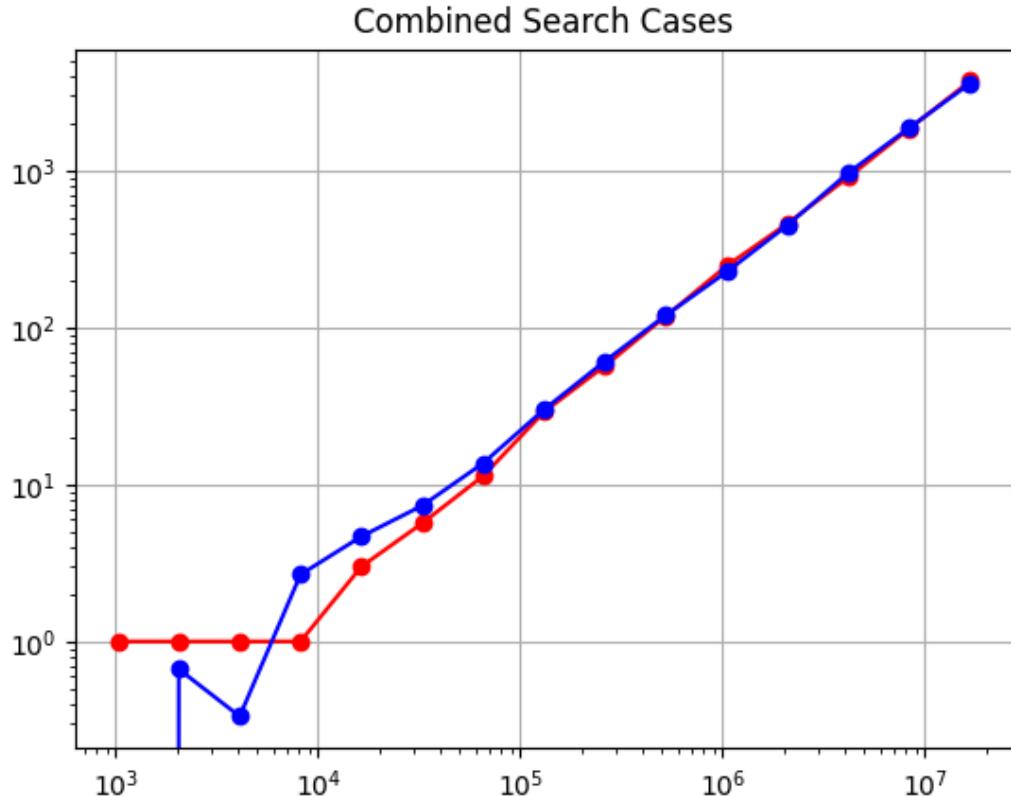


Figure 3: “Combined Cases”

Because the solution is much less than the nodes, the random sum won't affect much to the final result. Combine the two figures above, we can see that two curve are well fitted, which is also consistent with our expectation.

Finally, in this project, the space complexity is $O(N)$, because we only need to store the nodes once in the tree and array.

Appendix: Source Code (in C)

File CMAKELISTS.txt:

```
cmake_minimum_required(VERSION 3.10)
set(CMAKE_C_COMPILER "gcc")
project(A+B)
```

```

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/includes)

file(GLOB SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/src/*.c)

add_executable(main ${SOURCES} main.c)

```

File main.c:

```

#include <stdio.h>
#include <stdlib.h>
#include "./includes/bst.h"
#include "./includes/avl.h"

// Callback function to print the equation
void printResult(int sum,int data) {
    printf("%d = %d + %d\n", sum, data, sum - data);
}

int main(){
    // Create two BST trees
    BstTree tree1 = createBstTree();
    BstTree tree2 = createBstTree();

    // Get the equation sum
    int sum;
    scanf("%d",&sum);

    // The AVL tree worked as a set, to ensure that the result is unique
    AvlTree ans = NULL;

    // Find all solution satisfying the equation
    int res = findEquation(tree1, tree2, sum, &ans);
    if(!res) // No solution is found
        printf("false\n");
    else printf("true\n");

    // Traverse the AVL tree and print the equation
    inorderTraverseAvlTree(ans,sum, printResult);

    // Traverse the tree in preorder
    preorderTraverseBstTree(tree1);
    printf("\n");
    preorderTraverseBstTree(tree2);
    printf("\n");

    // Free the memory
    deleteBstTree(tree1);
    deleteBstTree(tree2);
    return 0;
}

```

File src/avl.c

```
#include "../includes/avl.h"
#include <stdlib.h>

// Create a new AVL tree
AvlTree createAvlTree() {
    AvlTree tree = (AvlTree)malloc(sizeof(struct AvlNode));
    if (tree == NULL) {
        return NULL;
    }
    tree->left = NULL;
    tree->right = NULL;
    tree->height = 1; // Initial height of a new node is 1
    return tree;
}

// Delete the AVL tree
void deleteAvlTree(AvlTree tree) {
    if (tree != NULL) {
        // Delete its children first
        deleteAvlTree(tree->left);
        deleteAvlTree(tree->right);
        // Then delete the root
        free(tree);
    }
}

// Calculate the height of the tree
void updateHeight(AvlTree tree) {
    if (tree != NULL) {
        // In case its children are NULL, set to 0
        int leftHeight = (tree->left != NULL) ? tree->left->height : 0;
        int rightHeight = (tree->right != NULL) ? tree->right->height : 0;
        // Set the height of the current node to the max height of its children +
1
        tree->height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
    }
}

// Rotate the tree to the left
void rotateLeft(AvlTree* tree) {
    if(tree != NULL) {
        AvlTree temp = (*tree)->right;
        (*tree)->right = temp->left;
        temp->left = *tree;
        *tree = temp;

        // Update the heights of the nodes after rotation
        updateHeight((*tree)->left);
        updateHeight(*tree);
    }
}
```

```

}

// Rotate the tree to the right
void rotateRight(AvlTree* tree) {
    if(tree != NULL){
        AvlTree temp = (*tree)->left;
        (*tree)->left = temp->right;
        temp->right = *tree;
        *tree = temp;

        // Update the heights of the nodes after rotation
        updateHeight((*tree)->right);
        updateHeight(*tree);
    }
}

int getFactor(AvlTree tree) {
    // Calculate the balance factor of the tree node
    if (tree == NULL || (tree->left == NULL && tree->right == NULL)) {
        return 0;
    }

    // Handle the case when one of the children is NULL
    if(tree->left == NULL) {
        return tree->right->height;
    } else if (tree->right == NULL) {
        return -tree->left->height;
    } else {
        // If both of its children exist, return the difference of their heights
        return tree->right->height - tree->left->height;
    }
}

// Balancing the AVL tree
void balance(AvlTree* tree) {
    // Right heavy case
    if (getFactor(*tree) > 1) {
        if (getFactor((*tree)->right) < 0) {
            // RL case
            rotateRight(&(*tree)->right);
        }
        // LL case
        rotateLeft(tree);
    }
    // Left heavy case
    } else if (getFactor(*tree) < -1) {
        if (getFactor((*tree)->left) > 0) {
            // LR case
            rotateLeft(&(*tree)->left);
        }
        // RR case
        rotateRight(tree);
    }
}

```

```

    }

}

// Insert node into AVL tree
// Didn't insert the node if it already exists in the tree
void insertAvlNode(AvlTree* tree, int data) {
    if (*tree == NULL) {
        // Create a new node if encounters leaf
        *tree = createAvlTree();
        if (*tree == NULL) {
            return;
        }
        (*tree)->data = data;
        return;
    }

    // If greater, search right subtree recursively
    if (data < (*tree)->data) {
        insertAvlNode(&(*tree)->left, data);
    } else if (data > (*tree)->data) {
        insertAvlNode(&(*tree)->right, data);
    }
    // If equal, do nothing (no duplicates allowed)
    else {
        return;
    }

    // Update the height of the current node
    updateHeight(*tree);

    // Balancing the tree after insertion
    balance(tree);
}

void inorderTraverseAvlTree(AvlTree tree, int sum, void (*callback)(int, int)) {
    // Traverse the tree in order
    if (tree != NULL) {
        inorderTraverseAvlTree(tree->left, sum, callback);
        // Callback to output the equation
        callback(sum, tree->data);
        inorderTraverseAvlTree(tree->right, sum, callback);
    }
}

```

File src/bst.c

```

#include "../includes/bst.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

// Get user input and create the tree
BstTree createBstTree() {
    BstTree tree = NULL;
    int n;
    scanf("%d", &n);
    // Since we know the number of nodes, we can use an array to point to all
    // nodes
    BstTree* treeNodes = (BstTree*)malloc(sizeof(BstTree*) * n);
    // As well to its parent index
    int* parentIndex = (int*)malloc(sizeof(int) * n);
    if (treeNodes == NULL) {
        return tree;
    }
    // Read inputs, allocating memory
    // I don't allocate the memory at once, that's mainly because it cause error
    // when releasing the memory, I don't know the first node's address
    for (int i = 0; i < n; i++) {
        // Allocating memory for each node
        treeNodes[i] = (BstTree)malloc(sizeof(struct BstNode));
        scanf("%d%d", &treeNodes[i]->data, &parentIndex[i]);
        treeNodes[i]->left = NULL;
        treeNodes[i]->right = NULL;
    }
    for (int i = 0; i < n; i++) {
        // If the parent index is -1, set the node as the root of the tree
        if (parentIndex[i] == -1) {
            tree = treeNodes[i];
        } else {
            // If not, set the node as a child of its parent
            // Comparing data to determine if it is a left or right child
            if (treeNodes[i]->data < treeNodes[parentIndex[i]]->data) {
                treeNodes[parentIndex[i]]->left = treeNodes[i];
            } else {
                treeNodes[parentIndex[i]]->right = treeNodes[i];
            }
        }
    }
    return tree;
}

// Delete the tree
void deleteBstTree(BstTree tree) {
    if (tree->left != NULL) deleteBstTree(tree->left);
    if (tree->right != NULL) deleteBstTree(tree->right);
    free(tree);
}

// Traverse the tree in preorder
void preorderTraverseBstTree(BstTree tree) {
    if (tree == NULL) return;
}

```

```

    printf("%d ", tree->data);
    preorderTraverseBstTree(tree->left);
    preorderTraverseBstTree(tree->right);
}

// Helper function to count nodes in a BST
int countNodes(BstTree tree) {
    if (tree == NULL) return 0;
    return 1 + countNodes(tree->left) + countNodes(tree->right);
}

// Helper function to convert BST to sorted array using inorder traversal
void bstToArray(BstTree tree, int* array, int* index) {
    if (tree == NULL) return;
    bstToArray(tree->left, array, index);
    array[(*index)++] = tree->data;
    bstToArray(tree->right, array, index);
}

// Find the equation in the two trees using array-based approach
int findEquation(BstTree tree1, BstTree tree2, int sum, AvlTree* ans) {
    // Count nodes in both trees
    int size1 = countNodes(tree1);
    int size2 = countNodes(tree2);

    if (size1 == 0 || size2 == 0) return 0;

    // Allocate arrays for both trees
    int* array1 = (int*)malloc(size1 * sizeof(int));
    int* array2 = (int*)malloc(size2 * sizeof(int));

    // Convert trees to arrays
    int index1 = 0, index2 = 0;
    bstToArray(tree1, array1, &index1);
    bstToArray(tree2, array2, &index2);

    int left = 0;
    int right = size2 - 1;
    int found = 0;

    while (left < size1 && right >= 0) {
        int current_sum = array1[left] + array2[right];

        if (current_sum == sum) {
            // Found a pair, insert into set
            insertAvlNode(ans, array1[left]);
            found = 1;
            left++;
            right--;
        } else if (current_sum < sum) {
            left++; // Need larger value from array1
        }
    }
}

```

```

        } else {
            right--; // Need smaller value from array2
        }
    }

    // Release allocated memory
    free(array1);
    free(array2);

    return found;
}

```

File includes/avl.h

```

#pragma once

// AVL node structure
struct AvlNode{
    int data;
    struct AvlNode* left,*right;
    int height; // Height of the tree
};

// Define a tree as a pointer to a tree node
typedef struct AvlNode* AvlTree;

// AVL methods, whose function is presented as it names
AvlTree createAvlTree();
void deleteAvlTree(AvlTree tree);
void insertAvlNode(AvlTree* tree,int data);
void inorderTraverseAvlTree(AvlTree tree,int sum, void (*callback)(int,int));

```

File includes/bst.h

```

#pragma once
#include "../includes/avl.h"

// BST node structure
struct BstNode{
    int data;
    struct BstNode* left,*right;
};

// Define a tree as a pointer to a tree node
typedef struct BstNode* BstTree;

// BST methods, whose function is presented as it names
BstTree createBstTree();
void deleteBstTree(BstTree tree);
int findEquation(BstTree tree1,BstTree tree2,int sum,AvlTree* ans);
void preorderTraverseBstTree(BstTree tree);

```

Declaration

I hereby declare that all the work done in this project titled “Performance Measurement” is of my independent effort.