Project 2: A+B with Binary Search Trees

March 28, 2025

Contents

1 Introduction

1.1 Problem Description

The problem is related to two binary search trees and find a number A from tree1 and B from tree2 such that A+B=N.More concisely the input is two binary search trees and a number N, and I need to find a pair of numbers from the two trees respectively that add up to N and output the preorder traversal of the trees.

Supplement: We're given one line with a positive integer n1 indicating the number of nodes in tree1. Following that are n1 lines describing each node's value (k), and the index of its parent node (i). If a node is the root, since it doesn't have a parent, the index is given as -1.tree2 is given in the same way.At the end, you're given the sum N.All these numbers fall within the range specified in the question.

1.2 background–Binary Search Trees

A binary search tree (BST) is constructed based on specific criteria: each node in the tree must ensure that all the nodes in its left subtree contain values less than its own value, and all the nodes in its right subtree contain values greater than or equal to its own. Furthermore, both the left and right subtrees must adhere to the same rules of a BST.

1.3 How to solve it

In fact,the problem can be divided into three small problems:

- Construct two binary search trees.
- Perform the inorder traversal and find out the pairs.
- Perform the preorder traversal.

1.3.1 Problem 1: Construct two binary search trees

Considering the special input format(data and father's index),I can use an array to store the trees instead of the linked list.(more detailed explaination in code and Chapter 2 and 4).

1.3.2 Problem 2: Perform the inorder traversal(2.1) and find out the pairs(2.2)

Performing the inorder traversal to sort the data,then I use the two pointers method to acquire the pairs.

1.3.3 Problem 3: Perform the preorder traversal

Performing the preorder traversal in recursive way.

2 Chapter2: Algorithm Specification

2.1 BST structure

Algorithm 1: Binary Tree Node Structure

```
struct node;
int data;
int left;
int right;
```

2.2 Create Binary Search Tree

Solve problem 1

Algorithm 2: Create Binary Search Tree (Function Form)

Data: n: Number of nodes, $data$: Array of node values, fa: Array of parent indices

Result: Binary Search Tree $tree$

```
Function createBST(n, data, fa):
    tree ← allocateMemory(n);
    for i ← 0 to n − 1 do
        tree[i].data ← data[i];
        tree[i].left ← −1;
        tree[i].right ← −1;
    for i ← 0 to n − 1 do
        if fa[i] = −1 then
            continue;
        if tree[i].data < tree[fa[i]].data then
            tree[fa[i]].left ← i;
        else
            tree[fa[i]].right ← i;
    return tree;
```

2.3 Inorder traserval

Solving Problem 2.1

Algorithm 3: Inorder Traversal

Input : root: root node of the tree, n: current node index, array: output array

Output: array containing the inorder traversal of the tree

```
Function Inorder(root, n, array):
    i ← 0;
    // Initialize index for the output array
    Function InorderRecursive(root, n, array):
        if n = −1 then
            return // Base case: If the node is null, return
        end
        InorderRecursive(root, root[n].left, array);
        array[i] ← root[n].data;
        i ← i + 1;
        InorderRecursive(root, root[n].right, array);
    InorderRecursive(root, n, array);
    // Start the recursive traversal
```

2.4 Preorder traversal

Solving Problem (similar to the inorder version)

Algorithm 4: Preorder Traversal

```
Function Preorder(tree, n):
    if n = −1 then
        return;
    end
    print root[n].data;
    Preorder(tree, tree[n].left);
    Preorder(tree, tree[n].right);
end
```

2.5 Two pointers search

Solving Problem 2.2

Algorithm 5: Search Pairs with Given Sum

Input : *sum*: Target sum, *array*1: First sorted array, *array*2: Second sorted array, *n*1: Size of *array*1, *n*2: Size of *array*2

Output: Pairs of elements from *array*1 and *array*2 that sum to *sum*

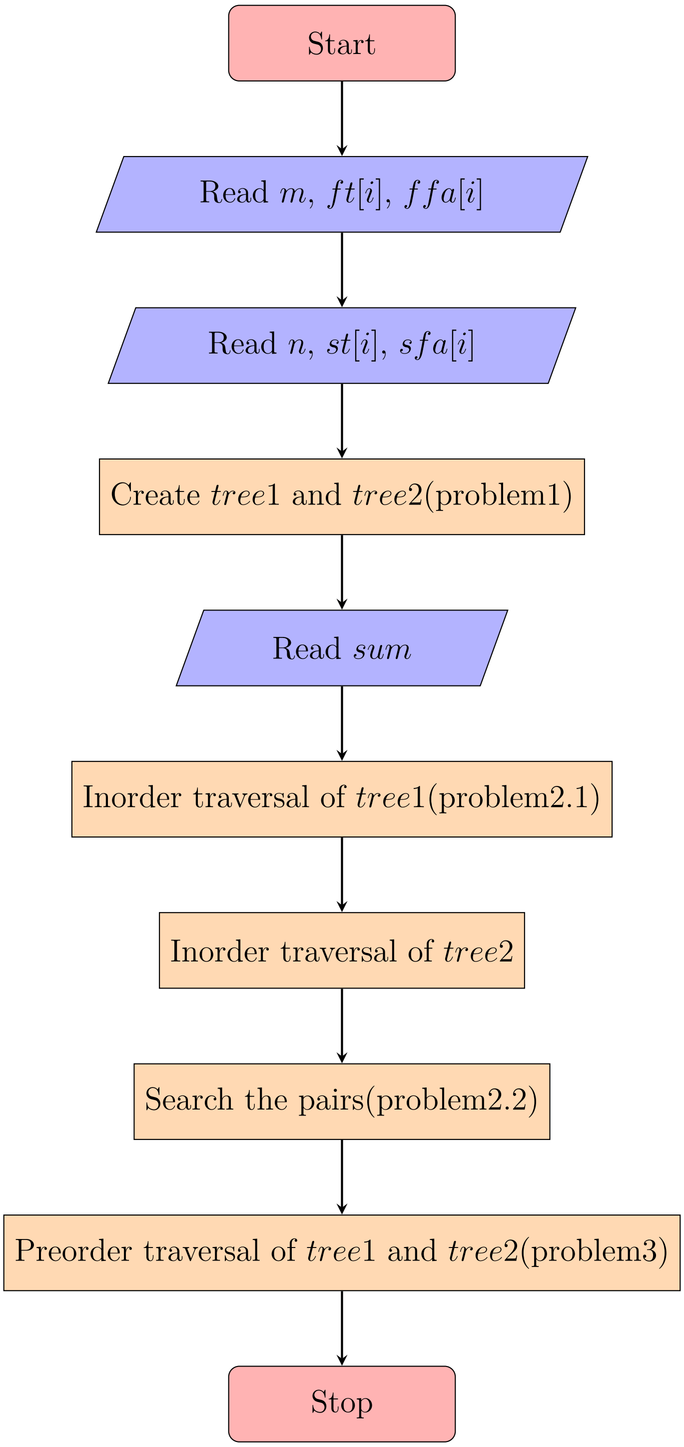
Data: *sum*, *array*1: BST1->array1, *array*2: BST2->array2, *n*1:array1.size, *n*2:array2.size

Function `SEARCH`(*sum*, *array*1, *array*2, *n*1, *n*2)**:**

```
flag ← 0;
left ← 0// Pointer for the start of array1;
right ← n2 − 1// Pointer for the end of array2;
while left ≠ n1 and right ≠ −1 do
    if array1[left] + array2[right] = sum then
        if not flag then
            print "true"// Print "true" only once;
        end
        flag ← 1// Mark that a pair is found;
        print "sum = array1[left] + array2[right]";
        left ← left + 1;
        right ← right − 1;
        if left = n1 or right = −1 then
            break;
        end
    end
    while T1[left − 1] = T1[left] do
        left ← left + 1// Skip duplicate elements in array1;
    end
    while T2[end] = T2[end + 1] do
        right ← right − 1// Skip duplicate elements in array2;
    end
    if T1[left] + T2[end] > sum then
        right ← right − 1// If sum is too large, move right pointer;
    else
        left ← left + 1// If sum is too small, move left pointer;
    end
end
if not flag then
    print "false";
    // Print "false" if no pair is found
end
```

2.6 Main Structure

Below is the sketch of the main program (also main structure)



3 Chapter 3: Testing Results

3.1 basic test given in pta

below is table of basic test cases

Table 1: Basic Test Cases

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
1	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	Test whether the program can handle the sample correctly (true case)	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	5 10 -1 5 0 15 0 2 1 7 1 3 15 -1 10 0 20 0 40	false 10 5 2 7 15 15 10 20	Test whether the program can handle the sample correctly (false case)	false 10 5 2 7 15 15 10 20

3.2 special case test

below is the special case test table

Table 2: special case

Testing Number	**Input**	**Expected Output**	**Testing Purpose**	**Actual Output**
3	1 0 -1 1 0 -1 0	true 0 = 0 + 0 0 0	the smallest amount of input	true 0 = 0 + 0 0 0
4	3 1 -1 2 0 3 1 3 5 -1 4 0 3 1 6	true 6 = 1 + 5 6 = 2 + 4 6 = 3 + 3 1 2 3 5 4 3	a completely unbalanced binary tree	true 6 = 1 + 5 6 = 2 + 4 6 = 3 + 3 1 2 3 5 4 3
5	5 2 -1 2 0 2 1 2 2 2 3 6 3 -1 3 0 3 1 3 2 3 3 3 4 5	true 5 = 2 + 3 2 2 2 2 2 3 3 3 3 3 3	Test whether the program can handle the case where the data on each node is the same	true 5 = 2 + 3 2 2 2 2 2 3 3 3 3 3 3

3.3 complex test data

below are more complex test data and its result.

Table 3: complex case

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
6	15 8 -1 4 0 12 0 2 1 6 1 10 2 14 2 1 3 3 3 5 4 7 4 9 5 11 5 13 6 15 6 20 1 -1 2 0 3 1 4 2 5 3 6 4 7 5 8 6 9 7 10 8 11 9 12 10 13 11 14 12 15 13 16 14 17 15 18 16 19 17 20 18 10	true 10 = 1 + 9 10 = 2 + 8 10 = 3 + 7 10 = 4 + 6 10 = 5 + 5 10 = 6 + 4 10 = 7 + 3 10 = 8 + 2 10 = 9 + 1 8 4 2 1 3 6 5 7 12 10 9 11 14 13 15 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	Test the program's ability to handle complex situations, where a tree's nodes all have only one son or no children and the other is a full binary tree	true 10 = 1 + 9 10 = 2 + 8 10 = 3 + 7 10 = 4 + 6 10 = 5 + 5 10 = 6 + 4 10 = 7 + 3 10 = 8 + 2 10 = 9 + 1 8 4 2 1 3 6 5 7 12 10 9 11 14 13 15 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
7	look at the testinput file	in the testoutput file	To test whether the program can handle the maximum number of nodes and the max key value	look at the testoutput file

3.4 test photos

Here are part of the test photos

```
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
```

```
5
10 -1
5 0
15 0
2 1
7 1
3
15 -1
10 0
20 0
40
false
10 5 2 7 15
15 10 20
```

```
1
0 -1
1
0 -1
0
true
0 = 0 + 0
0
0
```

```
3
0 -1
1 0
2 1
3
5 -1
4 0
3 1
5
true
5 = 0 + 5
5 = 1 + 4
5 = 2 + 3
0 1 2
5 4 3
```

4 Analysis and Comments

Here I will analyze the time complexity of the program and its working principle and advantages

visit each node once means we push and pop it for a stack once.

4.1 Time Complexity

4.1.1 create BST

The time complexity of constructing a binary tree is linear, $\mathbf{O(n)}$, with n representing the total number of nodes within the tree. This is due to the necessity of traversing each node to assign both its value and its respective child nodes.

4.1.2 Inorder traserval

The time complexity of in-order traversal is also linear, $\mathbf{O(n)}$, where n represents the total number of nodes within the tree. This is because the in-order traversal algorithm requires visiting each node once, regardless of the tree's structure.

4.1.3 Search the pairs(two pointers method)

The time complexity for finding pairs that sum to a specified value in two trees is O(n1 + n2), where n1 and n2 are the node counts of the respective trees(also the arrays' size). This is because the algorithm will traverse every node in each tree to identify pairs that collectively equal the target sum.But in this alorithm,it only need traverse each array one time.

4.1.4 Preorder traserval

The time complexity of pre-order traversal is also linear, $\mathbf{O(n)}$, where n represents the total number of nodes within the tree. This is because the pre-order traversal algorithm requires visiting each node once, regardless of the tree's structure.

4.1.5 main function

The time complexity of the main function is O(n1 + n2), with n1 and n2 being the node counts of the two trees. This is due to the main function calls other functions, each of which exhibit a combined complexity of O(n1 + n2)(O(n1)+O(n2)+O(n1+n2)+O(n1)+O(n2)+O(n1)+O

Therefore,the whole time complexity of the program is O(n1+n2)

4.2 Space Complexity

4.3 Working Principle and Advantages

4.3.1 create BST

The space complexity of creating a binary tree is O(n), where n is the number of nodes in the tree. This is because we will allocate memory for all the nodes in the tree.

4.3.2 Inorder traserval

The space complexity of an inorder traversal is O(n), where n is the number of nodes in the tree. Because we will use a dynamic array to restore the sorted nodes'data in the tree. And the space complexity of the recursive function is also O(n),because we will visit each node once and function recursion is implemented using a stack.

4.3.3 Search the pairs(two pointers method)

The space complexity for the operation of finding pairs that add up to a specific sum is O(1), as this process does not require additional memory space; it merely involves checking if the sum matches the sum of two elements.

4.3.4 Preorder traserval

The space complexity of a preorder traversal is O(n), because we will visit each node once and function recursion is implemented using a stack.

4.3.5 main function

The space complexity of the primary function is O(n1 + n2), with n1 and n2 representing the node counts in the respective trees. This is due to the necessity of storing all the node values from both trees. However, for the arrays ft, ffa, st, sfa they are all of fixed(200005), constant size.

Therefore,the whole space complexity of the program is O(n1+n2) or O(200005)

4.3.6 the range of input

The range of input is the number of nodes in tree<=200000,and the value of each node is between -2E9 and 2E9.So we will use the array with a size of 200005 and **int** to restore values.

4.4 working principle,advantages,improvement and compare

4.4.1 working principle

1.we divided the problem into three small problems,and problem1 and problem 2.1 and problem 3 have been solved in the fds class,so I will put emphasis on problem 2.2–the two pointers method.

2.The two-pointer technique for finding pairs with a given sum involves using two indices that start at opposite ends of an array. One pointer moves forward if the sum is too small, and the other moves backward if the sum is too large, until the pair with the correct sum is found or the pointers meet. The method only works in sorted arrays, and the time complexity is O(n), so it is effcient.

4.4.2 advantages

- The program is easy to comprehend and revise
- The program is efficient,the time and space complexity is only O(N)

4.4.3 improvement

- we can use the non-resursion method to implement the inorder traversal,which will save more memory
- we can use the real dynamic array as fa,sa.
- we can use the hash table to store the node values,which means we don't need the inorder traversal.

4.4.4 compare

When comparing my program with another solution– traverse the first array and use binary search to find matched the second array's nodes,the time complexity of the program will be O(nlogn),which is less efficient than my programme.

Thanks for your patience Looking forward to your valuable advice.

5 Appendix: Source code

Listing 1: A+B with Binary Search Trees

```c
    // partial code
typedef struct node{
    int data;          // Value stored in the node
    int left;          // Index of the left child node
    int right;         // Index of the right child node
}Node;
binarytree create(int n,int *data,int *fa){
    binarytree tree=(binarytree)malloc(sizeof(struct node)*n); // Allocat
    for(int i=0; i<n; i++){
        tree[i].data = data[i];    // Assign node value
        tree[i].left = -1;         // Initialize left child index as -1 (n
        tree[i].right = -1;        // Initialize right child index as -1 (
    }
    for(int i=0;i<n;i++){
        if(fa[i]==-1) continue;    // Skip if the node has no parent (root
        else {
            if(tree[i].data<tree[fa[i]].data)
                tree[fa[i]].left=i; // Set current node as left child if
            else
                tree[fa[i]].right=i; // Otherwise, set as right child
        }
    }
    return tree; // Return the constructed tree
}
void inorder_tree1(binarytree tree,int root,int *array){
    static int size1=0; // Static variable to track the current position
    if(root==-1) return; // Base case: If the node is null, return
    inorder_tree1(tree,tree[root].left,array); // Traverse the left subtr
```

```
    array[sizet1++]=tree[root].data; // Store the current node's data in
    inorder_tree1(tree,tree[root].right,array); // Traverse the right sub
}
void preorder_tree(binarytree tree, int root){
    if(root == -1) return; // Base case: If the node is null, return
    if(flag1==0) printf(" "); // Print a space before each node except th
    if(flag1==1){
        flag1=0; // Reset the flag after printing the first node
    }
    printf("%d", tree[root].data); // Print the data of the current node
    preorder_tree(tree, tree[root].left); // Recursively traverse the lef
    preorder_tree(tree, tree[root].right); // Recursively traverse the ri
}
void match(int size1,int size2,int *array1,int *array2,int sum){
    int left=0,right=size2-1; // Initialize two pointers for the two arra
    int flag=0; // Flag to record if a valid pair is found
    while(left<=size1-1&&right>=0){ // Loop until pointers reach the boun
        int sum1=array1[left]+array2[right]; // Calculate the sum of the
        if(sum1==sum){ // If the sum matches the target
            if(flag==0){
                flag=1; // Set the flag to indicate a match is found
                printf("true\n"); // Print "true" only once
            }
            printf("%d = %d + %d\n", sum, array1[left], array2[right]); /
            while(array1[left]==array1[left+1])left++; // Skip the same v
            while(array2[right-1]==array2[right])right--; // Skip same va
            left++; // Move the left pointer forward
            right--; // Move the right pointer backward
        }
        else if(sum1<sum) left++; // If the sum is less than the target,
        else right--; // If the sum is greater than the target, move the
```

```
    }
    if(flag==0) printf("false\n"); // If no match is found, print "false"
}
int main(){
    int size1,size2,sum,root1,root2; // Variables for sizes, sum, and roo
    scanf("%d",&size1); //  the size of the first tree
    for(int i=0;i<size1;i++){
        scanf("%d %d",&ft[i],&ffa[i]); //  node data and parent index for
        if(ffa[i]==-1) root1=i; // find out the root node of the first tr
    }
    scanf("%d",&size2); //  the size of the second tree
    for(int i=0;i<size2;i++){
        scanf("%d %d",&st[i],&sfa[i]); //  node data and parent index for
        if(sfa[i]==-1) root2=i; // find out the root node of the second t
    }

    binarytree Tree1 =create(size1,ft,ffa); // Create the first binary tr
    binarytree Tree2 =create(size2,st,sfa); // Create the second binary t
    scanf("%d",&sum); //  the target sum value
    int *array1=(int*)malloc(sizeof(int)*size1); // create first array
    int *array2=(int*)malloc(sizeof(int)*size2); // create second array
    inorder_tree1(Tree1,root1,array1); // Perform inorder traversal on th
    inorder_tree2(Tree2,root2,array2); // Perform inorder traversal on th
    //printf("%d\n",array1[0]); // test(ignore it)
    match(size1,size2,array1,array2,sum); // find pairs matching the targ
    preorder_tree(Tree1,root1); // Perform preorder traversal on the firs
    printf("\n");
    flag1=1; // Reset the flag for preorder traversal
    preorder_tree(Tree2,root2); // Perform preorder traversal on the seco
    return 0; // End of program
}
```

6 Declaration

I hereby declare that all the work done in this project titled "Project 1:Performance Measurement (Search) " is of my independent effort