Project 2 A+B with Binary Search Trees

March 29, 2024

Contents

1 Chapter 1

The question is about explaining how a Binary Search Tree (BST) works and what its properties are, then solving a problem related to two BSTs and an integer N, which we need to decompose N into the sum of A and B.

A BST is a binary tree that's built with the following rules. Every node's left subtree has nodes that are smaller than the node itself, and the right subtree has nodes that are larger or equal to the node. These left and right subtrees also have to follow the BST rules.

The problem at hand involves two binary search trees named T1 and T2 and a certain number N. The task is to find one number in T1 (call it A) and one in T2 (call it B) such that when you add them together (A+B), you get N.

The input for this problem includes the data for T1, T2, and N. We're given one line with a positive integer n1 indicating the number of nodes in T1. 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. T2's data is provided in the same format as T1's. And at the end, you're given the number N. All these numbers fall within the range specified in the question.

2 Chapter 2

Algorithm 1 Binary Tree Node Structure

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

Algorithm 2 Create Binary Tree

```
function CREATE(n, data, father)
    root ← allocate memory for n nodes
    for i ← 0 to n − 1 do
        root[i].data ← data[i]
        root[i].left ← −1
        root[i].right ← −1
    end for
    for i ← 0 to n − 1 do
        if father[i] = −1 then
            continue
        end if
        if root[i].data < root[father[i]].data then
            root[father[i]].left ← i
        else
            root[father[i]].right ← i
        end if
    end for
    return root
end function
```

Algorithm 3 Inorder Traversal

```
function INORDER(root, n, array)
    i ← 0
    function INORDERRECURSIVE(root, n, array)
        if n = −1 then
            return
        end if
        INORDERRECURSIVE(root, root[n].left, array)
        array[i] ← root[n].data
        i ← i + 1
        INORDERRECURSIVE(root, root[n].right, array)
    end function
    INORDERRECURSIVE(root, n, array)
end function
```

Algorithm 4 Search Pairs with Given Sum

function SEARCH(m, $T1$, $T2$, $n1$, $n2$)
  $flag \leftarrow 0$
  $start \leftarrow 0$
  $end \leftarrow n2 - 1$
  while $start \neq n1$ **and** $end \neq -1$ **do**
    if $T1[start] + T2[end] = m$ **then**
      if ¬ flag **then**
        print ”true”
      end if
      $flag \leftarrow 1$
      print ”$m = T1[start] + T2[end]$”
      $start \leftarrow start + 1$
      $end \leftarrow end - 1$
      if $start = n1$ or $end = -1$ **then**
        break
      end if
      while $T1[start - 1] = T1[start]$ **do**
        $start \leftarrow start + 1$
      end while
      while $T2[end] = T2[end + 1]$ **do**
        $end \leftarrow end - 1$
      end whilewhile(T2[end]==T2[end-1])end–;
    else if $T1[start] + T2[end] > m$ **then**
      $end \leftarrow end - 1$
    else
      $start \leftarrow start + 1$
    end if
  end while
  if ¬ flag **then**
    print ”false”
  end if
end function

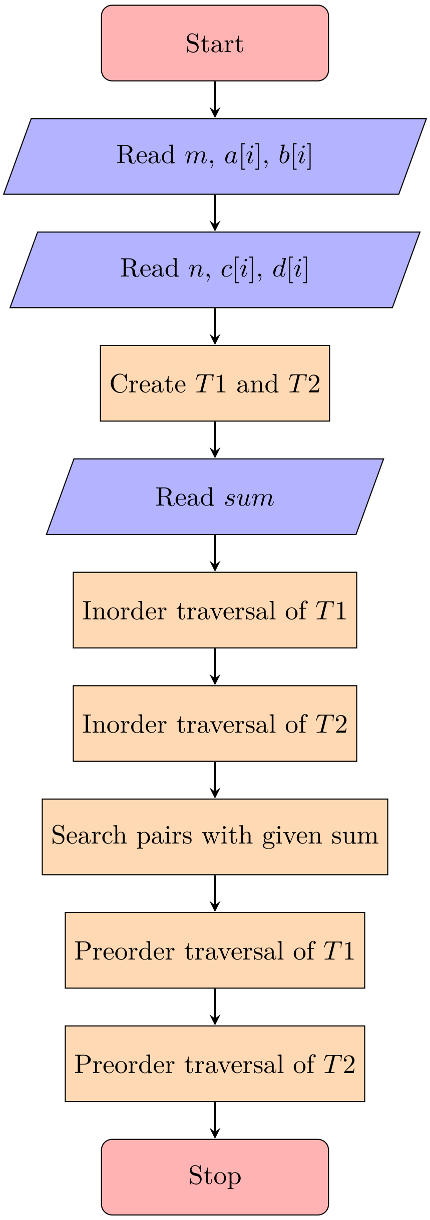
Algorithm 5 Preorder Traversal

function PREORDER($root$, n)
  if $n = -1$ **then**
    return
  end if
  print $root[n].data$
  PREORDER($root$, $root[n].left$)
  PREORDER($root$, $root[n].right$)
end function

Algorithm 6 Main Function

Input: m, n, a, b, c, d, sum
Output: None
$root1 \leftarrow 0$
$root2 \leftarrow 0$
Read m
for $i \leftarrow 0$ **to** $m - 1$ **do**
    Read $a[i]$, $b[i]$
    if $b[i] = -1$ **then** $root1 \leftarrow i$
end for
$T1 \leftarrow \textsc{Create}(m, a, b)$
Read n
for $i \leftarrow 0$ **to** $n - 1$ **do**
    Read $c[i]$, $d[i]$
    if $d[i] = -1$ **then** $root2 \leftarrow i$
end for
$T2 \leftarrow \textsc{Create}(n, c, d)$
Read sum
$\textsc{Inorder}(T1, root1, e)$
$\textsc{Inorder}(T2, root2, f)$
$\textsc{Search}(sum, e, f, m, n)$
$\textsc{Preorder}(T1, root1)$
$\textsc{Preorder}(T2, root2)$

Below is the sketch of the main program:


Start
Read m, $a[i]$, $b[i]$
Read n, $c[i]$, $d[i]$
Create $T1$ and $T2$
Read sum
Inorder traversal of $T1$
Inorder traversal of $T2$
Search pairs with given sum
Preorder traversal of $T1$
Preorder traversal of $T2$
Stop

3 Chapter 3

Table 1: 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 (i.e. the general 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 (i.e. the general false case)	false 10 5 2 7 15 15 10 20

Table 2: Test Cases(continued)

3	20 15 -1 10 0 20 0 5 1 13 1 17 2 22 2 3 3 7 3 12 4 14 4 16 5 19 5 21 6 23 6 1 7 4 8 6 9 11 10 18 11 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 18	true 18 = 1 + 17 18 = 3 + 15 18 = 5 + 13 18 = 7 + 11 18 = 10 + 8 18 = 12 + 6 18 = 13 + 5 18 = 14 + 4 18 = 15 + 3 18 = 16 + 2 15 10 5 3 1 7 4 13 12 6 14 11 20 17 16 18 19 22 21 23 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.	true 18 = 1 + 17 18 = 3 + 15 18 = 5 + 13 18 = 7 + 11 18 = 10 + 8 18 = 12 + 6 18 = 13 + 5 18 = 14 + 4 18 = 15 + 3 18 = 16 + 2 15 10 5 3 1 7 4 13 12 6 14 11 20 17 16 18 19 22 21 23 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Table 3: Test Cases (continued)

4	1 0 -1 1 0 -1 0	true 0 = 0 + 0 0 0	Test whether the program can correctly handle the smallest amount of input	true 0 = 0 + 0 0 0
5	5 1 -1 2 0 3 1 4 2 5 3 6 10 -1 9 0 8 1 7 2 6 3 5 4 10	true 10 = 1 + 9 10 = 2 + 8 10 = 3 + 7 10 = 4 + 6 10 = 5 + 5 1 2 3 4 5 10 9 8 7 6 5	Test whether the program can handle a completely unbalanced binary tree	true 10 = 1 + 9 10 = 2 + 8 10 = 3 + 7 10 = 4 + 6 10 = 5 + 5 1 2 3 4 5 10 9 8 7 6 5
6	5 6 -1 6 0 6 1 6 2 6 3 6 7 -1 7 0 7 1 7 2 7 3 7 4 13	true 13 = 6 + 7 6 6 6 6 6 7 7 7 7 7 7	Test whether the program can handle the corner case where the data on each node is the same	true 13 = 6 + 7 6 6 6 6 6 7 7 7 7 7 7

4 Chapter 4

Here ,we're going to analyze the complexities of time and space of our program.

4.1 Time Complexity

4.1.1 Create Binary Tree

The time complexity of creating a binary tree is $O(n)$, where n is the number of nodes in the tree. This is because we have to iterate through all the nodes to assign their values and their children.

4.1.2 Inorder Traversal

The time complexity of an inorder traversal is $O(n)$, where n is the number of nodes in the tree. This is because we have to visit all the nodes in the tree exactly once.

4.1.3 Search Pairs with Given Sum

The time complexity of searching for pairs with a given sum is $O(n1 + n2)$, where $n1$ and $n2$ are the number of nodes in the two trees. This is because we have to iterate through all the nodes in both trees to find the pairs that add up to the given sum.

4.1.4 Preorder Traversal

The time complexity of a preorder traversal is $O(n)$, where n is the number of nodes in the tree. This is because we have to visit all the nodes in the tree exactly once.

4.1.5 Main Function

The time complexity of the main function is $O(n1 + n2)$, where $n1$ and $n2$ are the number of nodes in the two trees. This is because the main function calls the other functions, which have a time complexity of $O(n1 + n2)$.

4.2 Space Complexity

4.2.1 Create Binary Tree

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 have to allocate memory for all the nodes in the tree.

4.2.2 Inorder Traversal

The space complexity of an inorder traversal is $O(n)$, where n is the number of nodes in the tree. This is because we have to store the values of all the nodes in the tree.

4.2.3 Search Pairs with Given Sum

The space complexity of searching for pairs with a given sum is $O(1)$, because we don't need any extra space to perform this operation,only to judge whether the sum is equal to the sum of two nodes.

4.2.4 Preorder Traversal

The space complexity of a preorder traversal is $O(1)$, because we don't need any extra space to perform this operation,only to printf the result we want.

4.2.5 Main Function

The space complexity of the main function is $O(n1+n2)$, where $n1$ and $n2$ are the number of nodes in the two trees. This is because we have to store the values of all the nodes in the two trees.While for array a,b,c,d,e,f,they are all of constant size.

5 Source code

```
#include<stdio.h>
#include<stdlib.h>

static int iffirst = 1; // Flag to check if it is the first
   node

typedef struct node* binarytree; // Define a binary tree as a
   pointer to a struct node

int a[200000]; // Array to store data for the first tree
int b[200000]; // Array to store father nodes for the first
   tree
int c[200000]; // Array to store data for the second tree
int d[200000]; // Array to store father nodes for the second
   tree
int e[200000]; // Array to store the inorder traversal of the
   first tree
```

```
int f[200000]; // Array to store the inorder traversal of the second tree

struct node{
    int data;
    int left;   // Index of the left child node
    int right;  // Index of the right child node
};

// Function to create a binary tree
binarytree create(int n, int *data, int *father){
    binarytree root = (binarytree)malloc(sizeof(struct node)*n); // Allocate memory for the binary tree
    for(int i=0; i<n; i++){
        root[i].data = data[i];
        root[i].left = -1;     // Initialize left child index as -1
        root[i].right = -1;    // Initialize right child index as -1
    }
    for(int i=0; i<n; i++){
        if(father[i] == -1) continue; // Skip if the node has no father
        if(root[i].data < root[father[i]].data) root[father[i]].left = i;  // Assign the current node as the left child of its father
        else root[father[i]].right = i; // Assign the current node as the right child of its father
    }
    return root;
}

// Function to perform inorder traversal of a binary tree and store the result in an array
void inorder(binarytree root, int n, int *array){
    static int i = 0; // Static variable to keep track of the index in the array
    if(n == -1) return; // Base case: if the current node is NULL, return
    inorder(root, root[n].left, array); // Recursively traverse the left subtree
    array[i++] = root[n].data; // Store the data of the current node in the array
    inorder(root, root[n].right, array); // Recursively traverse the right subtree
}


void inorder1(binarytree root, int n, int *array){
    static int j = 0;
    if(n == -1) return;
    inorder1(root, root[n].left, array);
```

```
        array[j++] = root[n].data;
        inorder1(root, root[n].right, array);
    }

    // Function to search for pairs of nodes in two binary trees that sum up to a given value
    void search(int m, int *T1, int *T2, int n1, int n2){
        int flag = 0; // Flag to check if any pairs are found
        int start = 0; // Start index of the first array
        int end = n2 - 1; // End index of the second array
        while(start != n1 && end != -1){
            if(T1[start] + T2[end] == m){ // If the sum of the current pair is equal to the given value
                if(!flag) printf("true\n"); // Print "true" if it is the first pair found
                flag = 1; // Set the flag to indicate that at least one pair is found
                printf("%d = %d + %d\n", m, T1[start], T2[end]); // Print the pair
                start++;
                end--;
                if(start == n1 || end == -1) break; // If the start index of the first array reaches the end or the end index of the second array reaches the start, break the loop (to avoid infinite loop
                while(T1[start-1]==T1[start])start++; // Move to the next distinct element in the first array
                while(T2[end]==T2[end+1])end--; // Move to the previous distinct element in the second array
            }
            else if(T1[start] + T2[end] > m) end--; // If the sum is greater than the given value, move to the previous element in the second array
            else start++; // If the sum is less than the given value, move to the next element in the first array
        }
        if(!flag) printf("false\n"); // If no pairs are found, print "false"
    }

    // Function to perform preorder traversal of a binary tree and print the nodes
    void preorder(binarytree root, int n){
        if(n == -1) return; // Base case: if the current node is NULL, return
        if(iffirst)
            iffirst = 0;
        else
            printf(" ");
        printf("%d", root[n].data); // Print the data of the current node
```

```c
        preorder(root, root[n].left); // Recursively traverse the left subtree
        preorder(root, root[n].right); // Recursively traverse the right subtree
    }

    int main(){
        int m, n, root1 = 0, root2 = 0, sum;
        scanf("%d", &m); // Read the number of nodes in the first tree
        for(int i=0; i<m; i++){
            scanf("%d %d", &a[i], &b[i]); // Read the data and father nodes for each node in the first tree
            if(b[i] == -1) root1 = i; // Find the root of the first tree
        }
        binarytree T1 = create(m, a, b); // Create the first binary tree
        scanf("%d", &n);
        for(int i=0; i<n; i++){
            scanf("%d %d", &c[i], &d[i]);
            if(d[i] == -1) root2 = i;
        }
        binarytree T2 = create(n, c, d); // Create the second binary tree
        scanf("%d", &sum); // Read the target sum
        inorder(T1, root1, e); // Perform inorder traversal of the first tree and store the result in array e
        inorder1(T2, root2, f); // Perform inorder traversal of the second tree and store the result in array f
        search(sum, e, f, m, n); // Search for pairs of nodes in the two trees that sum up to the target sum
        preorder(T1, root1);
        printf("\n");
        iffirst = 1;
        preorder(T2, root2);
        return 0;
    }
```

Listing 1: Source C Code

6 Declaration

I hereby declare that all the work done in this project titled "Project 2:A+B with Binary Search Trees" is of my independent effort.