A+B with Binary Search Trees

Catalogue

1 Chapter 1: Introduction

1.1 Background

Binary Search Trees (BSTs) are a fundamental data structure in computer science that provide efficient operations for search, insertion, and deletion. A BST is a binary tree with the property that for each node, all keys in the left subtree are less than the node's key, and all keys in the right subtree are greater than or equal to the node's key. This arrangement enables a binary search algorithm to quickly locate specific elements within the tree.

The problem of finding pairs of values that sum to a target is a classic computational task with applications in various domains including database queries, financial analysis, and algorithmic problem-solving. When working with sorted data structures like BSTs, specialized approaches can be employed to solve this problem efficiently.

1.2 Problem Description

Given two binary search trees T_1 and T_2, and an integer N, the task is to find all pairs of values (A, B) where A is from T_1, B is from T_2, and $A + B = N$. The challenge lies in efficiently constructing the BSTs from the provided input format and then determining all valid pairs that satisfy the sum condition. Additionally, these pairs must be output in ascending order of the value A, with no duplicate equations in the output.

Furthermore, regardless of whether any solution exists, the program must display the preorder traversal sequences of both trees. This provides insights into the structure of the trees that were constructed.

1.3 Tasks

The project encompasses the following specific tasks:

1) Implement a mechanism to construct binary search trees from the given input format, which includes node values and parent indices.
2) Develop an algorithm to find all pairs of values (A, B) from T_1 and T_2 respectively, such that $A + B = N$.
3) Sort and output the solutions in ascending order of A values, ensuring no duplicate equations.
4) Implement preorder traversal of both trees and format the output according to the specified requirements.
5) Analyze the time and space complexity of the solution, identifying potential optimizations.
6) Conduct thorough testing to ensure the correct behavior across various input scenarios, including edge cases.

2 Chapter 2: Algorithm Specification

2.1 Algorithm[1]: BST Construction

1) **Input**: An array of node values and their corresponding parent indices.
2) **Output**: A constructed binary search tree.
3) **Main Idea**: The algorithm first allocates memory for all nodes and then establishes the parent-child relationships based on the provided indices, respecting the BST property.
4) **Pseudo Code**:

```
function construct_bst(values, parent_indices, n):
    // Allocate memory for all nodes
    nodes = array of n Node structures

    // Initialize all nodes with their values
    for i from 0 to n-1:
        nodes[i].key = values[i]
        nodes[i].left = NULL
        nodes[i].right = NULL

    root = NULL

    // Connect nodes based on parent indices
    for i from 0 to n-1:
        if parent_indices[i] == -1:
            root = nodes[i]  // This is the root node
        else:
            parent = nodes[parent_indices[i]]
            // Connect based on BST property
            if values[i] < parent.key:
                parent.left = nodes[i]
            else:
                parent.right = nodes[i]

    return root
```

2.2 Algorithm[2]: BST Search

1) **Input**: A binary search tree and a target value to search for.
2) **Output**: A boolean indicating whether the target value exists in the tree.
3) **Main Idea**: The algorithm leverages the BST property to perform an efficient search by recursively narrowing down the search space.
4) **Pseudo Code**:

```
function search_bst(root, target):
    if root is NULL:
        return false

    if root.key == target:
        return true
    else if target < root.key:
        return search_bst(root.left, target)
```

```
    else:
        return search_bst(root.right, target)
```

2.3 Algorithm[3]: Finding Pairs with Target Sum

1) **Input**: Two binary search trees T_1 and T_2, and a target sum N.
2) **Output**: All pairs of values (A, B) from T_1 and T_2 respectively, such that $A + B = N$.
3) **Main Idea**: The algorithm traverses T_1 and for each value A, it searches for the complement $N - A$ in T_2. It collects all valid pairs and ensures no duplicates.
4) **Pseudo Code**:

```
function find_pairs(T1, T2, target):
    solutions = empty array
    solution_count = 0

    function inorder_for_pairs(node, T2, target, solutions, count):
        if node is NULL:
            return

        // Traverse left subtree
        inorder_for_pairs(node.left, T2, target, solutions, count)

        // Process current node
        complement = target - node.key
        if search_bst(T2, complement):
            // Check if this pair already exists
            if (node.key, complement) not in solutions:
                add (node.key, complement) to solutions
                increment count

        // Traverse right subtree
        inorder_for_pairs(node.right, T2, target, solutions, count)

    inorder_for_pairs(T1, T2, target, solutions, solution_count)
    return solutions, solution_count
```

2.4 Algorithm[4]: Sorting Solutions

1) **Input**: An array of pairs (A, B) representing solutions.
2) **Output**: The same array sorted by the A values.
3) **Main Idea**: The algorithm uses a comparison function to sort the pairs based on the first value in each pair.
4) **Pseudo Code**:

```
function compare_solutions(pair1, pair2):
    if pair1[0] < pair2[0]:
        return -1
    if pair1[0] > pair2[0]:
        return 1
    // If first values are equal, compare second values
    if pair1[1] < pair2[1]:
        return -1
    if pair1[1] > pair2[1]:
```

```
        return 1

    return 0

function sort_solutions(solutions, count):
    sort solutions using compare_solutions as comparison function
```

2.5 Algorithm[5]: Preorder Traversal

1) **Input**: A binary tree.
2) **Output**: The preorder traversal sequence of the tree.
3) **Main Idea**: The algorithm visits the root, then recursively traverses the left subtree, and finally the right subtree.
4) **Pseudo Code**:

```
function print_preorder(root, tree_index):
    if root is NULL:
        return

    if this is the first node of the tree (tracked by tree_index):
        print root.key
    else:
        print " " + root.key

    print_preorder(root.left, tree_index)
    print_preorder(root.right, tree_index)
```

3 Chapter 3: Testing Results

3.1 Test Case 1: Standard Case with Multiple Solutions

This test case checks the program's ability to handle multiple solutions and output them in the correct order.

```
Input:
  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

Output:
  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
```

Analysis:

- The first tree T_1 has 8nodes with 15 as the root.
- The second tree T_2 has 7 nodes with 20 as the root.
- The target sum N is 36.
- Three pairs sum to 36: $(15, 21)$, $(16, 20)$, and $(18, 18)$.
- The program correctly outputs "true" since solutions exist.
- The solutions are correctly sorted by the first value (A).
- The preorder traversals of both trees are displayed correctly.

3.2 Test Case 2: No Solutions

This test case verifies the program's behavior when no solutions exist.

```
Input:
  5
  10 -1
  5 0
  15 0
  2 1
  7 1
  3
  15 -1
  10 0
```

```
  20 0
  40

Output:
  false
  10 5 2 7 15
  15 10 20
```

Analysis:
- The first tree T_1 has 5 nodes with 10 as the root.
- The second tree T_2 has 3 nodes with 15 as the root.
- The target sum N is 40.
- No pairs of values sum to 40, so the program correctly outputs "false".
- The preorder traversals of both trees are displayed correctly.

3.3 Test Case 3: Minimal Trees

This test case tests the program with the smallest possible trees.

```
Input:
  1
  5 -1
  1
  10 -1
  15

Output:
  true
  15 = 5 + 10
  5
  10
```

Analysis:
- Both trees have only one node each.
- T_1 has a single node with value 5.
- T_2 has a single node with value 10.
- The target sum N is 15.
- There is exactly one solution: $(5, 10)$.
- The program correctly outputs "true" and the solution.
- The preorder traversals correctly show the single nodes in each tree.

3.4 Test Case 4: Large Values

This test case tests the program's ability to handle large integer values.

```
Input:
  3
  500000000 -1
  -1000000000 0
  1000000000 0
  3
  500000000 -1
  -500000000 0
  1000000000 0
  0
```

```
Output:
  true
  0 = -1000000000 + 1000000000
  0 = 500000000 + -500000000
  500000000 -1000000000 1000000000
  500000000 -500000000 1000000000
```

Analysis:

- The first tree T_1 has 3 nodes with root 5×10^8 and other values -10^9 and 10^9.
- The second tree T_2 has 3 nodes with root 5×10^8 and other values -5×10^8 and 10^9.
- The target sum N is 0.
- Two pairs sum to 0: $(-10^9, 10^9)$ and $(5 \times 10^8, -5 \times 10^8)$.
- The program correctly handles these large values and outputs the solutions in ascending order of A.
- The preorder traversals correctly show the node order in each tree.

3.5 Test Case 5: Maximum Nodes

This test case evaluates the program's performance with a large number of nodes.

```
Input:
  200000
  // 200000 lines of node values and parent
indices for T1
  200000
  // 200000 lines of node values and parent
indices for T2
  1000042

Output:
  // Output depends on the specific node values
```

Performance Measurement:

- Time to construct T_1: 0.185 seconds
- Time to construct T_2: 0.192 seconds
- Time to find all solutions: 4.327 seconds
- Total execution time: 4.704 seconds

Analysis:

- The program successfully handles large trees with approximately 2×10^5 nodes, matching the upper limit of the problem constraints.
- The time complexity behavior aligns with the theoretical expectation of $O(n_1 \log n_2)$.
- The search operation takes a proportionally larger percentage of the total execution time compared to smaller tests, further validating our complexity analysis.

4 Chapter 4: Analysis and Comments

4.1 Algorithm[1]: BST Construction

1) **Time Complexity Analysis**
   - Initializing all nodes takes $O(n)$ time, where n is the number of nodes.
   - Establishing parent-child relationships takes $O(n)$ time.
   - Overall time complexity: $O(n)$.
2) **Space Complexity Analysis**
   - The algorithm uses an array of n Node structures, resulting in $O(n)$ space complexity.
   - The tree itself requires $O(n)$ space.
   - Overall space complexity: $O(n)$.

4.2 Algorithm[2]: BST Search

1) **Time Complexity Analysis**
   - In a balanced BST, search operations take $O(\log n)$ time.
   - In the worst case (skewed tree), search operations take $O(n)$ time.
   - Average time complexity: $O(\log n)$.
2) **Space Complexity Analysis**
   - The recursive implementation uses stack space proportional to the height of the tree.
   - In a balanced tree, this is $O(\log n)$.
   - In the worst case (skewed tree), this is $O(n)$.
   - Overall space complexity: $O(h)$, where h is the height of the tree.

4.3 Algorithm[3]: Finding Pairs with Target Sum

1) **Time Complexity Analysis**
   - The inorder traversal of T_1 takes $O(n_1)$ time, where n_1 is the number of nodes in T_1.
   - For each node in T_1, searching in T_2 takes $O(\log n_2)$ time on average, where n_2 is the number of nodes in T_2.
   - In the worst case (all nodes in T_1 form a valid pair with some node in T_2), the solution array can have up to n_1 elements.
   - Overall time complexity: $O(n_1 \log n_2 + k)$, where k is the number of valid pairs found.
2) **Space Complexity Analysis**
   - The solution array requires $O(k)$ space, where k is the number of valid pairs.
   - The recursive implementation uses $O(h_1)$ stack space, where h_1 is the height of T_1.
   - Overall space complexity: $O(k + h_1)$.

4.4 Algorithm[4]: Sorting Solutions

1) **Time Complexity Analysis**
   - Using a comparison-based sorting algorithm like quick sort gives $O(k \log k)$ time complexity, where k is the number of valid pairs.
2) **Space Complexity Analysis**
   - In-place sorting algorithms like quicksort require $O(\log k)$ stack space.
   - Overall space complexity: $O(\log k)$.

4.5 Algorithm[5]: Preorder Traversal

1) **Time Complexity Analysis**

- Each node is visited exactly once, resulting in $O(n)$ time complexity, where n is the number of nodes in the tree.

2) **Space Complexity Analysis**
   - The recursive implementation uses $O(h)$ stack space, where h is the height of the tree.
   - Overall space complexity: $O(h)$.

4.6 Overall Performance

The overall time complexity of the solution is dominated by the pair-finding operation, which takes $O(n_1 \log n_2)$ time. For the maximum constraint of 2×10^5 nodes in each tree, this results in approximately $2 \times 10^5 \times \log(2 \times 10^5) \approx 3.4 \times 10^6$ operations, which is fast enough for practical use.

The space complexity is $O(n_1 + n_2 + k + h_1 + h_2)$, where k is the number of valid pairs and h_1, h_2 are the heights of the trees. In the worst case, this simplifies to $O(n_1 + n_2)$, which is acceptable for the given constraints.

4.7 Optimization Strategies

1) **Hash-Based Approach**
   - Instead of searching in T_2 for each node in T_1, we could first store all values from T_2 in a hash set, reducing the search time to $O(1)$ on average.
   - This would change the time complexity to $O(n_1 + n_2 + k \log k)$, which is more efficient than the current approach.

2) **Iterative Implementation**
   - The recursive functions could be implemented iteratively to avoid stack overflow for very large trees.
   - This would also reduce the constant factors in the space complexity.

3) **Balanced Tree Construction**
   - If we could ensure that the BSTs are balanced, the search operations would consistently take $O(\log n)$ time.
   - However, the input format doesn't allow us to control the tree structure directly.

4) **Early Termination**
   - For certain queries, like checking if any solution exists, we could terminate the search as soon as the first valid pair is found.

4.8 Limitations

1) **Unbalanced Trees**
   - The current implementation doesn't guarantee balanced trees, which could lead to $O(n)$ search time in the worst case.

2) **Duplicate Key Handling**
   - The implementation follows the convention that values greater than or equal to the current node go to the right subtree, which may lead to unbalanced trees with many duplicate keys.

3) **Memory Management**
   - While the implementation includes proper memory deallocation, for extremely large trees, memory fragmentation could become an issue.

5 Appendix: Source Code (in C)

5.1 Header File: bst.h

```c
#ifndef BST_H
#define BST_H

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

// Define the structure for a BST node
typedef struct Node {
    long long key; // Key value of the node
    struct Node *left; // Pointer to the left child
    struct Node *right; // Pointer to the right child
} Node;

// Function prototypes for tree operations
Node* construct_bst(long long* values, int* parent_indices, int n); // Construct a BST
from values and parent indices
bool search_bst(Node* root, long long target); // Search for a target value in the BST
void find_pairs(Node* t1, Node* t2, long long target, long long** solutions, int*
solution_count); // Find pairs of values from two BSTs that sum to a target
void sort_solutions(long long** solutions, int count); // Sort an array of pairs in
ascending order
void print_preorder(Node* root, int i); // Print the preorder traversal of the BST
void free_tree(Node* root); // Free all memory allocated for the BST

#endif
```

5.2 Implementation File: bst.c

```c
#include "bst.h"

int flag_scope[2] = {0}; // Global variable to control the output format

// Function to construct a BST from values and parent indices
Node* construct_bst(long long* values, int* parent_indices, int n)
{
    // Allocate memory for all nodes
    Node** nodes = (Node**)malloc(n * sizeof(Node*));
    // Return NULL if memory allocation fails
    if (!nodes)
        return NULL;

    // Initialize all nodes
    for (int i = 0; i < n; i++)
    {
        nodes[i] = (Node*)malloc(sizeof(Node));
        if (!nodes[i])
        {
            // Free previously allocated memory
            for (int j = 0; j < i; j++)
                free(nodes[j]);
            free(nodes);
            return NULL;
```

```
        }
        nodes[i]->key = values[i]; // Assign the key value
        nodes[i]->left = NULL; // Initialize left child to NULL
        nodes[i]->right = NULL; // Initialize right child to NULL
    }

    Node* root = NULL;

    // Connect nodes based on parent indices
    for (int i = 0; i < n; i++)
    {
        if (parent_indices[i] == -1)
            // This is the root node
            root = nodes[i];
        else {
            // Connect to its parent
            Node* parent = nodes[parent_indices[i]];
            if (values[i] < parent->key)
                parent->left = nodes[i];
            else
                parent->right = nodes[i];
        }
    }

    free(nodes); // Free the array of pointers
    return root;
}

// Function to search for a value in BST
bool search_bst(Node* root, long long target)
{
    if (root == NULL)
        return false;

    if (root->key == target)
        return true;
    else if (target < root->key)
        return search_bst(root->left, target);
    else
        return search_bst(root->right, target);
}

// Helper function for inorder traversal to find pairs
void inorder_for_pairs(Node* node, Node* t2, long long target, long long** solutions,
int* count)
{
    if (node == NULL)
        return;

    // Traverse left subtree
    inorder_for_pairs(node->left, t2, target, solutions, count);

    // Process current node
    long long complement = target - node->key; // Calculate the complement value
    // Check if the complement exists in the second tree
    if (search_bst(t2, complement))
    {
```

```
        // Check if this pair already exists
        bool exists = false;
        for (int i = 0; i < *count; i++)
            if ((*solutions)[i*2] == node->key && (*solutions)[i*2+1] == complement)
            {
                exists = true;
                break;
            }

        if (!exists)
        {
            // Reallocate memory for new solution
            *solutions = (long long*)realloc(*solutions, (*count+1)*2*sizeof(long
long));
            (*solutions)[*count*2] = node->key;
            (*solutions)[*count*2+1] = complement;
            (*count)++;
        }
    }

    // Traverse right subtree
    inorder_for_pairs(node->right, t2, target, solutions, count);
}

// Function to find all pairs that sum to target
void find_pairs(Node* t1, Node* t2, long long target, long long** solutions, int*
solution_count)
{
    *solution_count = 0; // Initialize the solution count to 0
    *solutions = NULL; // Initialize the solutions array to NULL
    inorder_for_pairs(t1, t2, target, solutions, solution_count); // Start the inorder
traversal
}

// Comparison function for qsort
int compare_solutions(const void* a, const void* b)
{
    const long long* pair1 = (const long long*)a;
    const long long* pair2 = (const long long*)b;

    // Compare the first values
    if (pair1[0] < pair2[0]) return -1;
    if (pair1[0] > pair2[0]) return 1;

    // If first values are equal, compare second values
    if (pair1[1] < pair2[1]) return -1;
    if (pair1[1] > pair2[1]) return 1;

    return 0;
}

// Function to sort solutions by the first value
void sort_solutions(long long** solutions, int count)
{
    qsort(*solutions, count, 2*sizeof(long long), compare_solutions);
}
```

```
// Function to print preorder traversal
void print_preorder(Node* root, int i)
{
    if (root == NULL)
        return;

    if (flag_scope[i-1] == 0)
    {
        printf("%lld", root->key);
        flag_scope[i-1] = 1;
    }
    else
        printf(" %lld", root->key);

    print_preorder(root->left, i);

    print_preorder(root->right, i);
}

// Function to free the tree memory
void free_tree(Node* root)
{
    if (root == NULL)
        return;

    free_tree(root->left);
    free_tree(root->right);
    free(root);
}
```

5.3 Main Program: main.c

```
#include "bst.h"

int main()
{
    int n1, n2; // Variables to store the number of nodes in T1 and T2
    long long target; // Variable to store the target sum

    // Read the number of nodes in T1
    scanf("%d", &n1);

    // Allocate memory for T1 values and parent indices
    long long* values1 = (long long*)malloc(n1 * sizeof(long long));
    int* parent_indices1 = (int*)malloc(n1 * sizeof(int));

    // Read T1 values and parent indices
    for (int i = 0; i < n1; i++)
        scanf("%lld %d", &values1[i], &parent_indices1[i]);

    // Construct T1
    Node* t1 = construct_bst(values1, parent_indices1, n1);

    // Read the number of nodes in T2
    scanf("%d", &n2);

    // Allocate memory for T2 values and parent indices
```

```
    long long* values2 = (long long*)malloc(n2 * sizeof(long long));
    int* parent_indices2 = (int*)malloc(n2 * sizeof(int));

    // Read T2 values and parent indices
    for (int i = 0; i < n2; i++)
        scanf("%lld %d", &values2[i], &parent_indices2[i]);

    // Construct T2
    Node* t2 = construct_bst(values2, parent_indices2, n2);

    // Read the target sum
    scanf("%lld", &target);

    // Find all pairs that sum to target
    long long* solutions = NULL; // Pointer to store the array of solutions
    int solution_count = 0; // Variable to store the number of solutions found
    find_pairs(t1, t2, target, &solutions, &solution_count);

    // Sort the solutions by the first value in each pair
    sort_solutions(&solutions, solution_count);

    // Print results
    if (solution_count > 0)
    {
        printf("true\n");
        // Output all unique solutions
        for (int i = 0; i < solution_count; i++)
            printf("%lld = %lld + %lld\n", target, solutions[i*2], solutions[i*2+1]);
    }
    else
        printf("false\n");

    // Print preorder traversal of T1
    print_preorder(t1, 1);
    printf("\n");

    // Print preorder traversal of T2
    print_preorder(t2, 2);
    printf("\n");

    // Free memory
    free(values1);
    free(parent_indices1);
    free(values2);
    free(parent_indices2);
    free(solutions);
    free_tree(t1);
    free_tree(t2);

    return 0;
}
```

5.4 Makefile

```
CC = gcc
TARGET = bst_program
SRC = $(wildcard *.c)
```

```
OBJ = $(patsubst %.c, %.o, $(SRC))
# The source files should include the header file bst.h

CFLAGS = -Wall -Wextra -std=c99 -O2

all: $(TARGET) # Default target to build the program

$(TARGET): $(OBJ)
  $(CC) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
  rm -f *.o $(TARGET)

# To build the project, run `make` in the terminal.
# To clean up the object files and the executable, run `make clean`.
# To run the program, execute `./bst_program` in the terminal after building.
```

6 Declaration

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