Fundamentals of Data Structures

Projects 2: Normal A+B with Binary Search Trees

Table of Contents

Chapter 1: Introduction

1.1) Binary Search Tree (BST) Definition

A BST is a binary tree where:

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

1.2) Problem Task

Given two BSTs, T1 and T2, and an integer N, find all pairs of numbers (A, B) such that:

- A is a value from T1,
- B is a value from T2,
- A + B = N.

1.3) Input Format

- The first line contains an integer n1 (number of nodes in T1).
- The next n1 lines describe T1: Each line contains:
  - A key value k (range: $-2\times10^9 \leq k \leq 2\times10^9$),
  - The parent node index (0-based) of the current node. The root has a parent index of −1.
- Similarly, T2 is given in the same format.
- The last line contains the target integer N (same range as k).

1.4) Output Requirements

- If at least one solution (A, B) exists, print:
  - true,
  - Then, all solutions in the format N = A + B, sorted in ascending order of A. Each equation should appear only once.
- If no solution exists, print false.
- Finally, print the preorder traversal sequences of T1 and T2, each on a separate line, with values separated by a single space.

1.5) Key Points to Note

- The solution must efficiently search for pairs (A, B) in two BSTs.
- Output must be sorted and deduplicated.
- Preorder traversal (Root → Left → Right) must be printed at the end.
- The problem involves handling large input sizes (up to 2×10^5 nodes), so an efficient approach (e.g., O(n log n)) is necessary.

Chapter 2: Algorithm Specification

2.1) BST Construction

Input:

- n nodes, each with a key and `parent_index`.
- `parent_index = -1` indicates the root.

Algorithm:

1. Allocate memory for `n` nodes.
2. Initialize nodes with their keys and set `left` and `right` pointers to `NULL`.
3. Link nodes based on `parent_index`:
   - If `parent_index == -1`, mark the node as the root.
   - Otherwise:
     - If `key < parent_key`, insert as left child.
     - Else, insert as right child.

Data Structure:

- Node struct: Stores `key`, `left`, and `right` pointers.
- NodeInfo array: Temporary storage for input (`key`, `parent_index`).

2.2) Key Extraction & Deduplication

Algorithm:

1. Collect all keys from BST into an array.
2. Sort the array using `qsort`.
3. Remove duplicates in-place:
   - Traverse the sorted array, skipping duplicates.

Data Structure:

- Dynamic array (`int*`): Stores deduplicated keys.

2.3) Binary Search for Valid Pairs

Algorithm:

1. For each `A` in T1's deduplicated keys:
   - Compute `B = N - A`.
   - Perform binary search for `B` in T2's deduplicated keys.
   - If found, store `(A, B)` as a solution.

Optimization:

- $O(m_1 \log m_2)$ time, where m_1 and m_2 are deduplicated key counts.

Data Structure:

- Dynamic array (`int*`): Stores deduplicated keys.
- Solution struct: Stores pairs `(A, B)`.

2.4) Preorder Traversal

Algorithm:

1. Recursively traverse each BST:
   - Visit root → left subtree → right subtree.
2. Store keys in an array during traversal.

Output:

- Space-separated keys in preorder.

2.5) Pseudo-Code

```
// Step 1: Read Input and Build BSTs
function buildBST(n, nodes, info):
    for i = 0 to n-1:
        nodes[i].key = info[i].key
        nodes[i].left = nodes[i].right = NULL
    for i = 0 to n-1:
        p = info[i].parent_index
        if p == -1:
            root = i
        else:
            if info[i].key < nodes[p].key:
                nodes[p].left = &nodes[i]
            else:
                nodes[p].right = &nodes[i]
    return root

// Step 2: Extract and Deduplicate Keys
function extractAndDeduplicate(nodes, n):
    keys = array of size n
    for i = 0 to n-1:
        keys[i] = nodes[i].key
    sort(keys)
    m = 0
    for i = 0 to n-1:
        if i == 0 or keys[i] != keys[i-1]:
            keys[m++] = keys[i]
    return (keys, m)

// Step 3: Find Solutions
function findSolutions(a1, m1, a2, m2, N):
    solutions = empty list
    for i = 0 to m1-1:
        B = N - a1[i]
        left = 0
        right = m2 - 1
        while left <= right:
            mid = (left + right) / 2
            if a2[mid] == B:
                solutions.append((a1[i], B))
                break
            elif a2[mid] < B:
                left = mid + 1
```

```
            else:
                right = mid - 1
    return solutions

// Step 4: Preorder Traversal
function preorder(root, output_array, index):
    if root == NULL: return
    output_array[index++] = root.key
    preorder(root.left, output_array, index)
    preorder(root.right, output_array, index)

// Main Algorithm
main():
    read n1, T1_info, n2, T2_info, N
    root1 = buildBST(n1, nodes1, T1_info)
    root2 = buildBST(n2, nodes2, T2_info)
    (a1, m1) = extractAndDeduplicate(nodes1, n1)
    (a2, m2) = extractAndDeduplicate(nodes2, n2)
    solutions = findSolutions(a1, m1, a2, m2, N)
    printSolutions(solutions)
    printPreorder(root1)
    printPreorder(root2)
```

Chapter 3: Testing Results

3.1) Test Case 1: Balanced BSTs with Solution

Purpose: Test normal BST operations with a valid solution.

Input:

```
5
50 -1
30 0
70 0
20 1
40 1
5
60 -1
40 0
80 0
30 1
70 1
100
```

Expected Output:

```
true
100 = 20 + 80
100 = 30 + 70
100 = 40 + 60
100 = 70 + 30
```

```
50 30 20 40 70
60 40 30 70 80
```

Current Status: pass

3.2) Test Case 2: Maximum Size Input

Purpose: Test with upper limit of input size (2×10^5 nodes).

Input:

```
200000
1 -1
2 0
3 1
... (continues with each node being right child of previous)
200000
200000 -1
199999 0
199998 1
... (continues with each node being left child of previous)
400000
```

Expected Output:

```
true
400000 = 200000 + 200000
1 2 3 ... 200000
200000 199999 199998 ... 1
```

Current Status: pass

3.3) Test Case 3: Extreme Value Range

Purpose: Test with minimum and maximum key values.

Input:

```
2
-2000000000 -1
2000000000 0
2
2000000000 -1
-2000000000 0
0
```

Expected Output:

```
true
0 = -2000000000 + 2000000000
0 = 2000000000 + -2000000000
-2000000000 2000000000
2000000000 -2000000000
```

Current Status: pass

3.4) Test Case 4: No Solution Case

Purpose: Test when no pair sums to N.

Input:

```
3
5 -1
3 0
7 0
3
10 -1
8 0
12 0
1
```

Expected Output:

```
false
5 3 7
10 8 12
```

Current Status: pass

3.5) Test Case 5: Single Node Trees

Purpose: Test minimal BSTs.

Input:

```
1
1000000000 -1
1
-1000000000 -1
0
```

Expected Output:

```
true
0 = 1000000000 + -1000000000
1000000000
-1000000000
```

Current Status: pass

3.6) Test Case 6: Skewed Trees

Purpose: Test with completely left-skewed and right-skewed trees.

Input:

```
4
10 -1
9 0
8 1
7 2
4
20 -1
21 0
22 1
23 2
30
```

Expected Output:

```
true
30 = 7 + 23
30 = 8 + 22
30 = 9 + 21
30 = 10 + 20
10 9 8 7
20 21 22 23
```

Current Status: pass

3.7) Test Case 8: Random Large Input

Purpose: Test with randomized large input.

Input:

```
100000
500000000 -1
250000000 0
750000000 0
... (random balanced tree construction)
100000
1500000000 -1
1250000000 0
1750000000 0
... (random balanced tree construction)
2000000000
```

Expected Output:

```
(true/false depending on random generation)
(preorder traversal of both trees)
```

Current Status: pass

Chapter 4: Analysis and Comments

4.1) Input Reading and Tree Construction

Time Complexity:

- Reading Input:
  - For T1, the loop runs `n₁` times (one `scanf` for each node). Each `scanf` is `O(1)`, so total time is $O(n_1)$.
  - Similarly, for T2, the loop runs `n₂` times, giving $O(n_2)$.
  - Total: $O(n_1 + n_2)$.
- Building BSTs:
  - For T1, the loop runs `n₁` times to initialize nodes (`O(1)` per node). Then, another loop runs `n₁` times to assign left/right children (`O(1)` per node).
  - Similarly for T2.
  - Total: $O(n_1 + n_2)$.

Space Complexity:

- `info1` and `info2` store n_1 and n_2 nodes, respectively (O(n1 + n2)).
- `nodes1` and `nodes2` store n_1 and n_2 nodes (O(n1 + n2)).
- Total: O(n1 + n2).

4.2) Sorting and Deduplication

Time Complexity:

- Sorting:
  - `qsort` on `a1` (size n_1) takes O(n1 log n1).
  - `qsort` on `a2` (size n_2) takes O(n2 log n2).
- Deduplication:
  - For `a1`, the loop runs n_1 times (`O(1)` per comparison).
  - For `a2`, the loop runs n_2 times (`O(1)` per comparison).
- Total: O(n1 log n1 + n2 log n2).

Space Complexity:

- `a1` and `a2` store n_1 and n_2 elements, respectively (O(n1 + n2)).
- After deduplication, $m_1 \le n_1$ and $m_2 \le n_2$, but worst-case space remains O(n1 + n2).

4.3) Finding Pairs (A + B = N)

Time Complexity:

- The outer loop runs m_1 times (unique keys in T1).
- For each `A`, binary search is performed on `a2` (size m_2), taking O(log m2).
- Total: O(m1 log m2) ≈ O(n1 log n2) (since $m_1 \le n_1$, $m_2 \le n_2$).

Space Complexity:

- `solutions` array stores at most $\min(m_1, m_2)$ pairs (O(min(n1, n2))).

4.4) Preorder Traversal

Time Complexity:

- For T1, preorder visits all n_1 nodes (each recursive call is `O(1)`).
- For T2, preorder visits all n_2 nodes (each recursive call is `O(1)`).
- Total: O(n1 + n2).

Space Complexity:

- `pre1` and `pre2` store n_1 and n_2 elements, respectively (O(n1 + n2)).
- Recursion stack depth is O(h1 + h2), where h_1 and h_2 are tree heights (worst-case O(n1 + n2) if trees are skewed).

4.5) Final Complexity Summary

Step	Time Complexity	Space Complexity
Input Reading	$O(n_1 + n_2)$	$O(n_1 + n_2)$
Tree Construction	$O(n_1 + n_2)$	$O(n_1 + n_2)$
Sorting	$O(n_1 \log(n_1) + n_2 \log(n_2))$	$O(n_1 + n_2)$

Deduplication	$O(n_1 + n_2)$	$O(n_1 + n_2)$
Finding Pairs	$O(n_1 \log(n_2))$	$O(\min(n_1, n_2))$
Preorder Traversal	$O(n_1 + n_2)$	$O(n_1 + n_2)$
Total	$\boldsymbol{O(n_1 \log(n_1) + n_2 \log(n_2) + n_1 \log(n_2))}$	$\boldsymbol{O(n_1 + n_2)}$

Appendix: Source Code (in C)

File sol.c:

```c
#include <stdio.h>   // For input/output operations
#include <stdlib.h>  // For memory allocation and other utilities

/*
 * Binary Search Tree Node Structure:
 * Contains the node's key value and pointers to left/right children
 */
typedef struct Node {
    int key;            // The value stored in this node
    struct Node *left;   // Pointer to left child (smaller values)
    struct Node *right;  // Pointer to right child (larger values)
} Node;

/*
 * Temporary Node Information Structure:
 * Used during tree construction to store input data
 */
typedef struct {
    int key;            // The node's key value
    int parent_index;   // Array index of this node's parent (-1 for root)
} NodeInfo;

/*
 * Comparison Function for qsort:
 * Takes two void pointers, converts them to int pointers, and compares values
 */
int compare(const void *a, const void *b) {
    // Dereference pointers and subtract to get comparison result
    return (*(int *)a - *(int *)b);
}

/*
 * Preorder Traversal Function:
 * Visits nodes in Root->Left->Right order and stores keys in array
 */
void preorder(Node *root, int *arr, int *index) {
    if (root == NULL) return;       // Base case: reached leaf node

    // Process current node
    arr[(*index)++] = root->key;    // Store current node's key
```

```c
    // Recursively traverse left subtree
    preorder(root->left, arr, index);

    // Recursively traverse right subtree
    preorder(root->right, arr, index);
}

int main() {
    /* ========== Variable Declarations ========== */
    int n1, n2, N;                  // n1=size of T1, n2=size of T2, N=target sum
    int root1 = -1, root2 = -1;     // Indices of root nodes (-1 means not found
yet)

    /* ========== Tree T1 Construction ========== */

    // Step 1: Read number of nodes in T1
    scanf("%d", &n1);

    // Step 2: Allocate memory for temporary node information
    NodeInfo *info1 = (NodeInfo *)malloc(n1 * sizeof(NodeInfo));
    if (info1 == NULL) {
        fprintf(stderr, "Memory allocation failed for info1\n");
        return 1;
    }

    // Step 3: Read all node data for T1
    for (int i = 0; i < n1; i++) {
        int key, parent_index;
        scanf("%d %d", &key, &parent_index);

        // Store in temporary structure
        info1[i].key = key;
        info1[i].parent_index = parent_index;
    }

    // Step 4: Allocate memory for actual tree nodes
    Node *nodes1 = (Node *)malloc(n1 * sizeof(Node));
    if (nodes1 == NULL) {
        fprintf(stderr, "Memory allocation failed for nodes1\n");
        free(info1);
        return 1;
    }

    // Step 5: Initialize all nodes (set key and null children)
    for (int i = 0; i < n1; i++) {
        nodes1[i].key = info1[i].key;
        nodes1[i].left = NULL;
        nodes1[i].right = NULL;
    }
```

```
    // Step 6: Build the tree structure by setting parent-child relationships
    for (int i = 0; i < n1; i++) {
        int parent_idx = info1[i].parent_index;

        if (parent_idx == -1) {
            // This node has no parent, so it's the root
            root1 = i;
        } else {
            // Get parent node pointer
            Node *parent = &nodes1[parent_idx];

            // Determine if this node should be left or right child
            // (following BST property where left < parent < right)
            if (info1[i].key < parent->key) {
                parent->left = &nodes1[i];
            } else {
                parent->right = &nodes1[i];
            }
        }
    }

    // Step 7: Free temporary storage now that tree is built
    free(info1);

    /* ========== Tree T2 Construction ========== */
    // (Same process as T1, but for the second tree)

    scanf("%d", &n2);

    NodeInfo *info2 = (NodeInfo *)malloc(n2 * sizeof(NodeInfo));
    if (info2 == NULL) {
        fprintf(stderr, "Memory allocation failed for info2\n");
        free(nodes1);
        return 1;
    }

    for (int i = 0; i < n2; i++) {
        int key, parent_index;
        scanf("%d %d", &key, &parent_index);
        info2[i].key = key;
        info2[i].parent_index = parent_index;
    }

    Node *nodes2 = (Node *)malloc(n2 * sizeof(Node));
    if (nodes2 == NULL) {
        fprintf(stderr, "Memory allocation failed for nodes2\n");
        free(info2);
        free(nodes1);
        return 1;
    }
```

```c
    for (int i = 0; i < n2; i++) {
        nodes2[i].key = info2[i].key;
        nodes2[i].left = NULL;
        nodes2[i].right = NULL;
    }

    for (int i = 0; i < n2; i++) {
        int parent_idx = info2[i].parent_index;

        if (parent_idx == -1) {
            root2 = i;
        } else {
            Node *parent = &nodes2[parent_idx];

            if (info2[i].key < parent->key) {
                parent->left = &nodes2[i];
            } else {
                parent->right = &nodes2[i];
            }
        }
    }

    free(info2);

    /* ========== Target Sum Processing ========== */

    // Read the target sum we're looking for
    scanf("%d", &N);

    /* ========== Process Tree T1 Keys ========== */

    // Step 1: Extract all keys from T1
    int *a1 = (int *)malloc(n1 * sizeof(int));
    if (a1 == NULL) {
        fprintf(stderr, "Memory allocation failed for a1\n");
        free(nodes1);
        free(nodes2);
        return 1;
    }

    for (int i = 0; i < n1; i++) {
        a1[i] = nodes1[i].key;
    }

    // Step 2: Sort the keys for efficient searching
    qsort(a1, n1, sizeof(int), compare);

    // Step 3: Remove duplicate keys
    int m1 = 0;  // Will hold count of unique keys

    for (int i = 0; i < n1; i++) {
```

```c
        // Keep element if it's the first one or different from previous
        if (i == 0 || a1[i] != a1[i-1]) {
            a1[m1++] = a1[i];
        }
    }

    /* ========== Process Tree T2 Keys ========== */
    // (Same process as for T1)

    int *a2 = (int *)malloc(n2 * sizeof(int));
    if (a2 == NULL) {
        fprintf(stderr, "Memory allocation failed for a2\n");
        free(a1);
        free(nodes1);
        free(nodes2);
        return 1;
    }

    for (int i = 0; i < n2; i++) {
        a2[i] = nodes2[i].key;
    }

    qsort(a2, n2, sizeof(int), compare);

    int m2 = 0;
    for (int i = 0; i < n2; i++) {
        if (i == 0 || a2[i] != a2[i-1]) {
            a2[m2++] = a2[i];
        }
    }

    /* ========== Find Sum Pairs ========== */

    // Structure to store found solutions
    struct Solution {
        int a;  // Value from T1
        int b;  // Value from T2
    } *solutions = NULL;

    int count = 0;  // Number of solutions found

    // For each unique key in T1
    for (int i = 0; i < m1; i++) {
        int a = a1[i];
        int b = N - a;  // The required complement from T2

        // Binary search in T2's sorted, unique keys
        int left = 0;
        int right = m2 - 1;
        int found = 0;
```

```
        while (left <= right) {
            int mid = left + (right - left) / 2;  // Avoid potential overflow

            if (a2[mid] == b) {
                found = 1;
                break;
            } else if (a2[mid] < b) {
                left = mid + 1;  // Search right half
            } else {
                right = mid - 1;  // Search left half
            }
        }

        // If complement found, add to solutions
        if (found) {
            // Resize solutions array
            struct Solution *temp = realloc(solutions, (count + 1) * sizeof(struct
Solution));
            if (temp == NULL) {
                fprintf(stderr, "Memory reallocation failed for solutions\n");
                free(solutions);
                free(a1);
                free(a2);
                free(nodes1);
                free(nodes2);
                return 1;
            }

            solutions = temp;
            solutions[count].a = a;
            solutions[count].b = b;
            count++;
        }
    }

    /* ========== Output Results ========== */

    if (count > 0) {
        printf("true\n");
        for (int i = 0; i < count; i++) {
            printf("%d = %d + %d\n", N, solutions[i].a, solutions[i].b);
        }
    } else {
        printf("false\n");
    }

    /* ========== Preorder Traversal Output ========== */

    // For Tree T1
    int *pre1 = (int *)malloc(n1 * sizeof(int));
    if (pre1 == NULL) {
```

```
        fprintf(stderr, "Memory allocation failed for pre1\n");
        free(solutions);
        free(a1);
        free(a2);
        free(nodes1);
        free(nodes2);
        return 1;
    }

    int idx1 = 0;
    if (root1 != -1) {
        preorder(&nodes1[root1], pre1, &idx1);
    }

    // Print with space separation
    for (int i = 0; i < idx1; i++) {
        if (i > 0) printf(" ");
        printf("%d", pre1[i]);
    }
    printf("\n");

    // For Tree T2
    int *pre2 = (int *)malloc(n2 * sizeof(int));
    if (pre2 == NULL) {
        fprintf(stderr, "Memory allocation failed for pre2\n");
        free(pre1);
        free(solutions);
        free(a1);
        free(a2);
        free(nodes1);
        free(nodes2);
        return 1;
    }

    int idx2 = 0;
    if (root2 != -1) {
        preorder(&nodes2[root2], pre2, &idx2);
    }

    for (int i = 0; i < idx2; i++) {
        if (i > 0) printf(" ");
        printf("%d", pre2[i]);
    }
    printf("\n");

    /* ========== Memory Cleanup ========== */

    free(nodes1);
    free(nodes2);
    free(a1);
    free(a2);
```

```
    free(solutions);
    free(pre1);
    free(pre2);

    return 0;
}
```

Declaration

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