Fundamentals of Data Structures

Laboratory Projects

Tree Traversals

周承扬 **3190102371**

Date: 2021-10-28

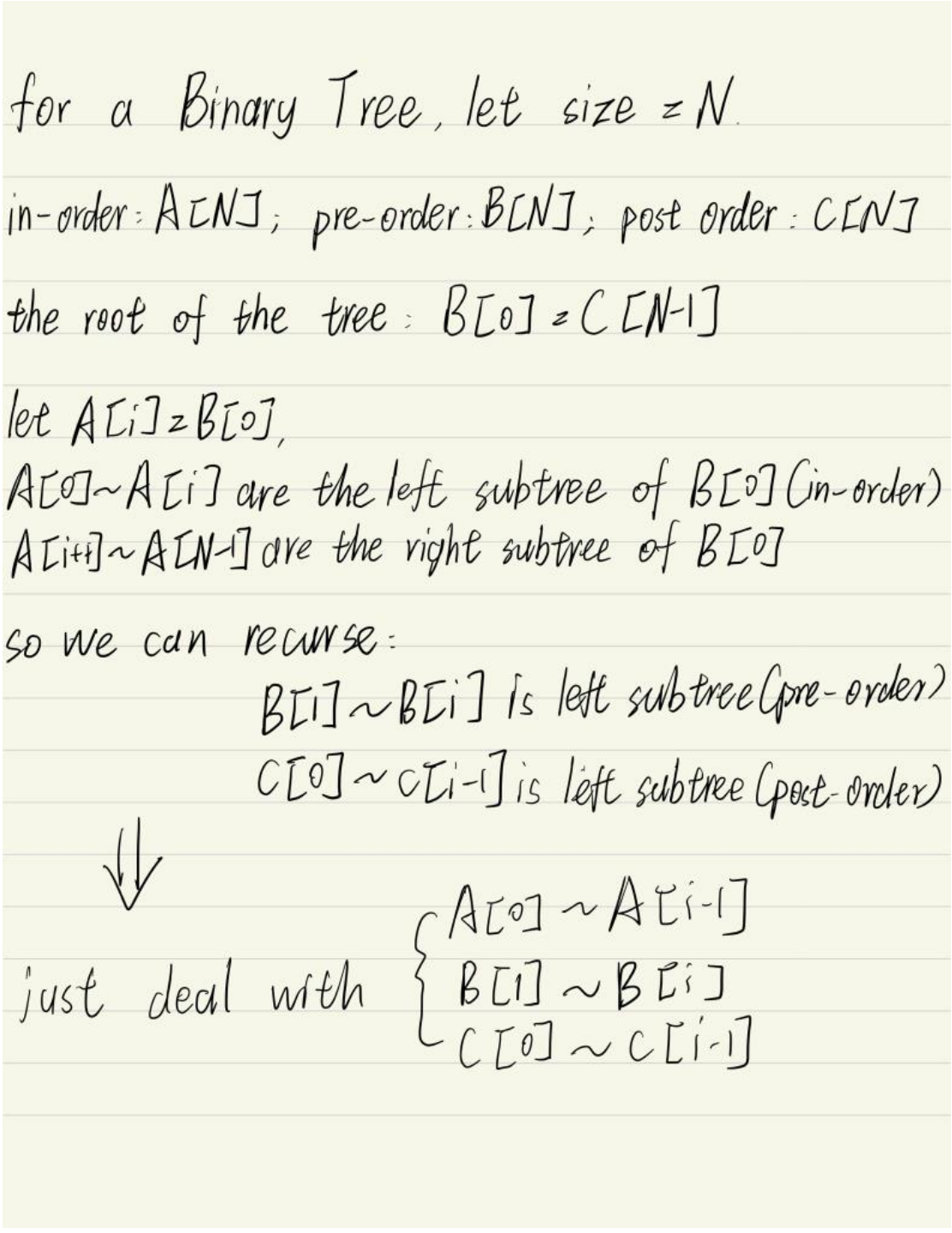
Chapter 1: Introduction

In this project, we need to solve the problem of trees with vacant elements. Initially, we will get 3 lines of a binary tree's in in-order, pre-order, and post-order with some elements lost.

Thus we need to analyze the construction of the tree, and judge is the tree is exclusive. According to the input requirements, first we need to give the value of n, which means the number of the tree. Second we will input the three lines follow, containing the incomplete in-order, pre-order and post-order traversal sequences, respectively. And we use "-" to present the vacant value.

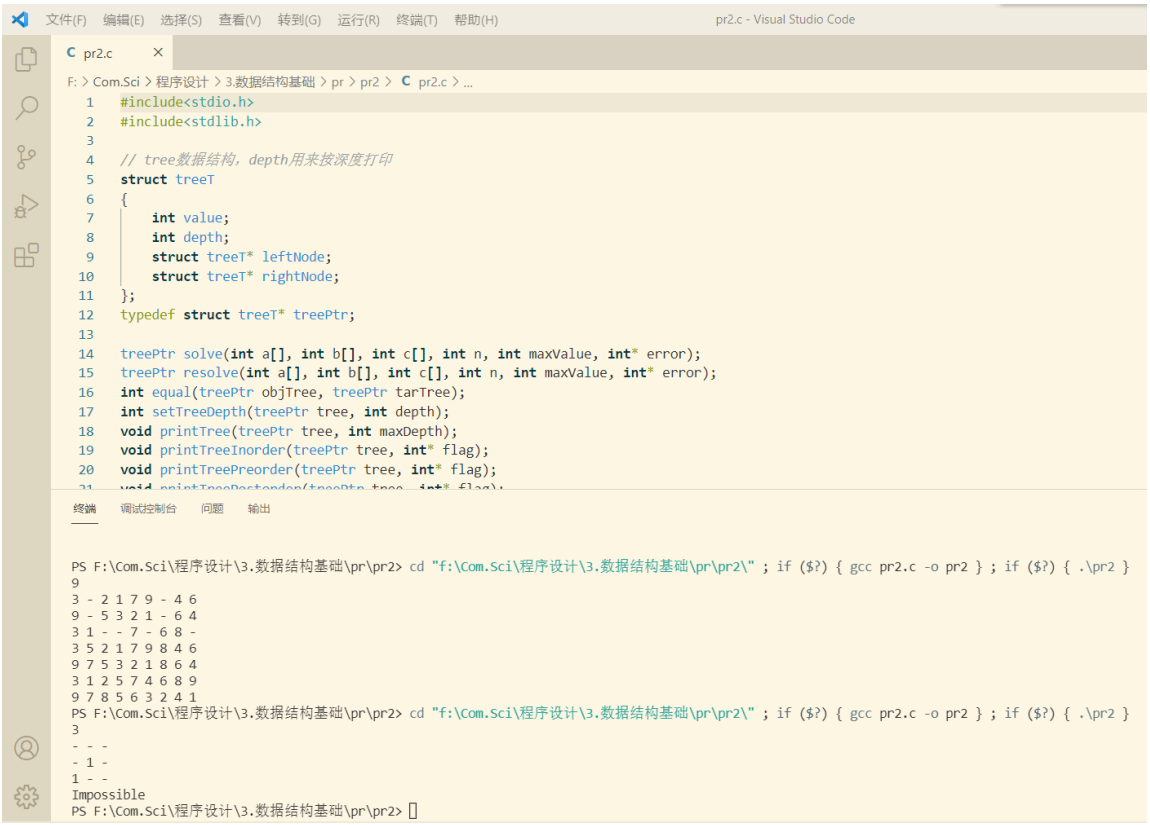
If the answer is yes, we just print in four lines the complete in-order, pre-order and post-order traversal sequences, together with the level order traversal sequence of the corresponding tree.

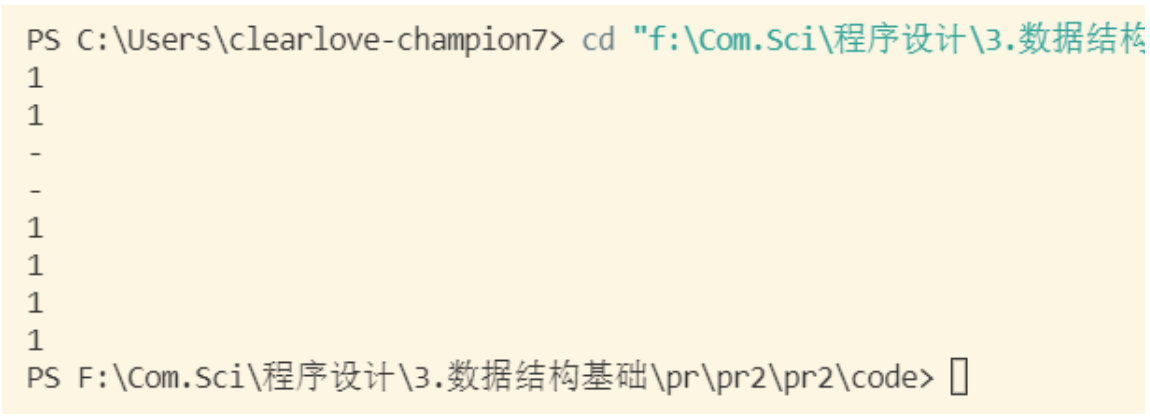
Chapter 2: Algorithm Specification

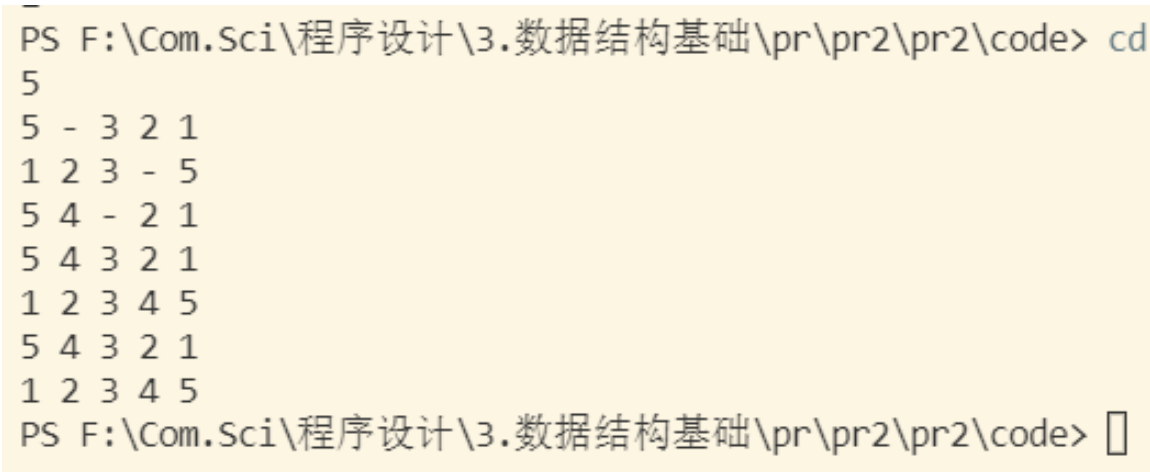


Chapter 3: Testing Results

The test results are shown below:


文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
pr2.c - Visual Studio Code
pr2.c
F: > Com.Sci > 程序设计 > 3.数据结构基础 > pr > pr2 > pr2.c > ...
#include<stdio.h>
#include<stdlib.h>
// tree数据结构，depth用来按深度打印
struct treeT
{
int value;
int depth;
struct treeT* leftNode;
struct treeT* rightNode;
};
typedef struct treeT* treePtr;
treePtr solve(int a[], int b[], int c[], int n, int maxValue, int* error);
treePtr resolve(int a[], int b[], int c[], int n, int maxValue, int* error);
int equal(treePtr objTree, treePtr tarTree);
int setTreeDepth(treePtr tree, int depth);
void printTree(treePtr tree, int maxDepth);
void printTreeInorder(treePtr tree, int* flag);
void printTreePreorder(treePtr tree, int* flag);
终端 调试控制台 问题 输出
PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2> cd "f:\Com.Sci\程序设计\3.数据结构基础\pr\pr2\" ; if ($?) { gcc pr2.c -o pr2 } ; if ($?) { .\pr2 }
9
3 - 2 1 7 9 - 4 6
9 - 5 3 2 1 - 6 4
3 1 - - 7 - 6 8 -
3 5 2 1 7 9 8 4 6
9 7 5 3 2 1 8 6 4
3 1 2 5 7 4 6 8 9
9 7 8 5 6 3 2 4 1
PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2> cd "f:\Com.Sci\程序设计\3.数据结构基础\pr\pr2\" ; if ($?) { gcc pr2.c -o pr2 } ; if ($?) { .\pr2 }
3
- - -
- 1 -
1 - -
Impossible
PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2>


PS C:\Users\clearlove-champion7> cd "f:\Com.Sci\程序设计\3.数据结构
1
1
-
-
1
1
1
1
PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2\pr2\code>


PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2\pr2\code> cd
5
5 - 3 2 1
1 2 3 - 5
5 4 - 2 1
5 4 3 2 1
1 2 3 4 5
5 4 3 2 1
1 2 3 4 5
PS F:\Com.Sci\程序设计\3.数据结构基础\pr\pr2\pr2\code>

Chapter 4: Analysis and Comments

The time complexity is $O(N^2 \log N)$ and space complexity is O(N).

According to code, the main complexity is in the part of solve and resolve, and this 2 parts are similar logically. So we just need to analyze the solve. We can see that the solve part is combined with 2 n's circulation with one iteration. Thus the time complexity is N * N * logN.

The code is complex so that the complexity is relatively high, which needs to be optimized.

Appendix: Source Code (in C)

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


// tree 数据结构，depth 用来按深度打印
struct treeT
{
    int value;
    int depth;
    struct treeT* leftNode;
    struct treeT* rightNode;
};
typedef struct treeT* treePtr;

treePtr solve(int a[], int b[], int c[], int n, int x[], int maxValue, int*
error);
treePtr resolve(int a[], int b[], int c[], int n, int x[], int maxValue,
int* error);
int equal(treePtr objTree, treePtr tarTree);
int setTreeDepth(treePtr tree, int depth);
void printTree(treePtr tree, int maxDepth);
void printTreeInorder(treePtr tree, int* flag);
void printTreePreorder(treePtr tree, int* flag);
void printTreePostorder(treePtr tree, int* flag);
void printTreeByDepth(treePtr tree, int depth);
int check(treePtr tree);
int checkTree(treePtr tree, int flag[]);

int main(){
    // n 是树的 size
    int n, i, ch, error = 0, maxDepth;
```

```
    int a[100], b[100], c[100], x[100] = {0};
    treePtr tree, retree;
    // 接收输入，分别存到a, b, c三个数组中
    scanf("%d", &n);
    for(i = 0; i < n; i++) {
        getchar();
        ch = getchar();
        if(ch == '-') {
            a[i] = 0;
        } else {
            a[i] = ch - '0';
        }
    }
    for(i = 0; i < n; i++) {
        getchar();
        ch = getchar();
        if(ch == '-') {
            b[i] = 0;
        } else {
            b[i] = ch - '0';
            x[b[i] - 1] = 1;
        }
    }
    for(i = 0; i < n; i++) {
        getchar();
        ch = getchar();
        if(ch == '-') {
            c[i] = 0;
        } else {
            c[i] = ch - '0';
            x[c[i] - 1] = 1;
        }
    }
    // 按正向生成树
    tree = solve(a, b, c, n, x, n, &error);
    if(error == -1) {
        printf("Impossible");
        return 0;
    }
    // 按反向生成树
    retree = resolve(a, b, c, n, x, n, &error);
    if(error == -1) {
        printf("Impossible");
        return 0;
```

```
    }
    // 若正向树和反向树不同，则表明有多种可能;
    // 只有正向和反向树相同，则可能解唯一
    if(equal(tree, retree)) {
        maxDepth = setTreeDepth(tree, 0);
        printTree(tree, maxDepth);
    } else {
        printf("Impossible");
    }
    return 0;
}


// 解树，正向。 error 判断tree 是否正确生成，maxValue 用来保证tree 节点value
上限，
// x 数组用来标记b，c 数组中已确定的value，以便在遍历时跳过
treePtr solve(int a[], int b[], int c[], int n, int x[], int maxValue, int*
error) {
    treePtr newTree;
    //
    int root, flag = 0, i;
    if(*error == -1 || n == 0) return NULL;
    //判断 根节点是否能被直接确定
    if(b[0] != 0) {
        if(b[0] != c[n-1] && c[n-1] != 0) {
            *error = - 1;
            return NULL;
        }
        root = b[0];
        flag = 1;
    } else if(c[n-1] != 0){
        if(b[0] != c[n-1] && b[0] != 0) {
            *error = - 1;
            return NULL;
        }
        root = c[n-1];
        flag = 1;
    }
    // flag=1 表示根节点被确定
    if(flag) {
        // 在a 数组中搜索根节点
        for(i = 0; i < n; i++) {
            if(a[i] == root) {
                // 分成左子树和右子树，递归生成
                newTree = (treePtr)malloc(sizeof(struct treeT));
```

```
                newTree->value = root;
                newTree->leftNode = solve(a, b+1, c, i, x, maxValue, error);
                newTree->rightNode = solve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);
                if(check(newTree)) {
                    *error = 0;
                    return newTree;
                } else {
                    *error = -1;
                    free(newTree);
                    return NULL;
                }
            }
        }
        // 如果a数组中找不到根节点，则遍历搜索解
        for(i = 0; i < n; i++) {
            *error = 0;
            if(a[i] == 0) {
                solve(a, b+1, c, i, x, maxValue, error);
                if(*error == -1) continue;
                solve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                if(*error == -1) continue;
                newTree = (treePtr)malloc(sizeof(struct treeT));
                newTree->value = root;
                newTree->leftNode = solve(a, b+1, c, i, x, maxValue, error);
                newTree->rightNode = solve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);

                if(check(newTree)) {
                    *error = 0;
                    return newTree;
                } else {
                    *error = -1;
                    free(newTree);
                }
            }
        }
    // 遍历搜索不成功，则表明解失败
        if(i == n) {
            *error = -1;
            return NULL;
        }
    } else {
        // 若确定不了根节点，则遍历假设根节点的value
```

```
        for(root = 1; root <= maxValue; root++) {
            // 若root 已经出现过，在b，c 数组中，则跳过
            if(x[root-1] == 1) continue;
            x[root-1] == 1;
            for(i = 0; i < n; i++) {
                if(a[i] == root) {
                    *error = 0;
                    solve(a, b+1, c, i, x, maxValue, error);
                    if(*error == -1) continue;
                    solve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                    if(*error == -1) continue;
                    newTree = (treePtr)malloc(sizeof(struct treeT));
                    newTree->value = root;
                    newTree->leftNode = solve(a, b+1, c, i, x, maxValue,
error);
                    newTree->rightNode = solve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);
                    if(check(newTree)) {
                        *error = 0;
                        x[root-1] == 0;
                        return newTree;
                    } else {
                        *error = -1;
                        free(newTree);
                    }
                }
            }
            for(i = 0; i < n; i++) {
                *error = 0;
                if(a[i] == 0) {
                    solve(a, b+1, c, i, x, maxValue, error);
                    if(*error == -1) continue;
                    solve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                    if(*error == -1) continue;
                    newTree = (treePtr)malloc(sizeof(struct treeT));
                    newTree->value = root;
                    newTree->leftNode = solve(a, b+1, c, i, x, maxValue,
error);
                    newTree->rightNode = solve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);
                    if(check(newTree)) {
                        *error = 0;
                        x[root-1] == 0;
                        return newTree;
```

```
                    } else {
                        *error = -1;
                        free(newTree);
                    }
                }
            }
            x[root-1] == 0;
        }
        if(root == maxValue+1) {
            *error = -1;
            return NULL;
        }
    }
}
// 类似solve，但遍历搜索采取反向，即从大到小，从后往前
treePtr resolve(int a[], int b[], int c[], int n, int x[], int maxValue,
int* error) {
    treePtr newTree;
    int root, flag = 0, i;
    if(*error == -1 || n == 0) return NULL;
    if(b[0] != 0) {
        if(b[0] != c[n-1] && c[n-1] != 0) {
            *error = - 1;
            return NULL;
        }
        root = b[0];
        flag = 1;
    } else if(c[n-1] != 0){
        if(b[0] != c[n-1] && b[0] != 0) {
            *error = - 1;
            return NULL;
        }
        root = c[n-1];
        flag = 1;
    }
    if(flag) {
        for(i = n-1; i >= 0; i--) {
            if(a[i] == root) {
                newTree = (treePtr)malloc(sizeof(struct treeT));
                newTree->value = root;
                newTree->leftNode = resolve(a, b+1, c, i, x, maxValue,
error);
                newTree->rightNode = resolve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);
```

```
                if(check(newTree)) {
                    *error = 0;
                    return newTree;
                } else {
                    *error = -1;
                    free(newTree);
                    return NULL;
                }
            }
        }
        for(i = n-1; i >= 0; i--) {
            *error = 0;
            if(a[i] == 0) {
                resolve(a, b+1, c, i, x, maxValue, error);
                if(*error == -1) continue;
                resolve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                if(*error == -1) continue;
                newTree = (treePtr)malloc(sizeof(struct treeT));
                newTree->value = root;
                newTree->leftNode = resolve(a, b+1, c, i, x, maxValue,
error);
                newTree->rightNode = resolve(a+i+1, b+i+1, c+i, n-i-1, x,
maxValue, error);

                if(check(newTree)) {
                    *error = 0;
                    return newTree;
                } else {
                    *error = -1;
                    free(newTree);
                }
            }
        }
        if(i == -1) {
            *error = -1;
            return NULL;
        }
    } else {
        for(root = maxValue; root > 0; root--) {
            if(x[root-1] == 1) continue;
            x[root-1] == 1;
            for(i = n-1; i >= 0; i--) {
                if(a[i] == root) {
                    *error = 0;
```

```
                    resolve(a, b+1, c, i, x, maxValue, error);
                    if(*error == -1) continue;
                    resolve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                    if(*error == -1) continue;
                    newTree = (treePtr)malloc(sizeof(struct treeT));
                    newTree->value = root;
                    newTree->leftNode = resolve(a, b+1, c, i, x, maxValue,
error);
                    newTree->rightNode = resolve(a+i+1, b+i+1, c+i, n-i-1,
x, maxValue, error);
                    if(check(newTree)) {
                        *error = 0;
                        x[root-1] == 0;
                        return newTree;
                    } else {
                        *error = -1;
                        free(newTree);
                    }
                }
            }
            for(i = n-1; i >= 0; i--) {
                *error = 0;
                if(a[i] == 0) {
                    resolve(a, b+1, c, i, x, maxValue, error);
                    if(*error == -1) continue;
                    resolve(a+i+1, b+i+1, c+i, n-i-1, x, maxValue, error);
                    if(*error == -1) continue;
                    newTree = (treePtr)malloc(sizeof(struct treeT));
                    newTree->value = root;
                    newTree->leftNode = resolve(a, b+1, c, i, x, maxValue,
error);
                    newTree->rightNode = resolve(a+i+1, b+i+1, c+i, n-i-1,
x, maxValue, error);
                    if(check(newTree)) {
                        *error = 0;
                        x[root-1] == 0;
                        return newTree;
                    } else {
                        *error = -1;
                        free(newTree);
                    }
                }
            }
            x[root-1] == 0;
```

```
        }
        if(root == 0) {
            *error = -1;
            return NULL;
        }
    }
}

// 判断两棵树是否所有的 value 和结构相同
int equal(treePtr objTree, treePtr tarTree) {
    if(objTree == NULL && tarTree == NULL) {
        return 1;
    } else if(objTree == NULL || tarTree == NULL){
        return 0;
    } else {
        if(objTree->value != tarTree->value) {
            return 0;
        }
        return equal(objTree->leftNode, tarTree->leftNode) &
equal(objTree->rightNode, tarTree->rightNode);
    }
}

// wrap 函数，用来按给定格式输出结果
void printTree(treePtr tree, int maxDepth) {
    int i, flag;
    flag = 0;
    printTreeInorder(tree, &flag);
    printf("\n");
    flag = 0;
    printTreePreorder(tree, &flag);
    printf("\n");
    flag = 0;
    printTreePostorder(tree, &flag);
    printf("\n");
    for(i = 0; i < maxDepth; i++)
        printTreeByDepth(tree, i);
}

// 设置树的每个节点的深度
int setTreeDepth(treePtr tree, int depth) {
    int a, b;
    if(tree == NULL) return depth;
    tree->depth = depth;
```

```
    a = setTreeDepth(tree->leftNode, depth+1);
    b = setTreeDepth(tree->rightNode, depth+1);
    return a > b ? a : b;
}
// 按顺序打印树
void printTreeInorder(treePtr tree, int* flag) {
    if(tree == NULL) return;
    printTreeInorder(tree->leftNode, flag);
    if(*flag) {
        printf(" ");
    } else {
        *flag = 1;
    }
    printf("%d", tree->value);
    printTreeInorder(tree->rightNode, flag);
}
// 按前序打印树
void printTreePreorder(treePtr tree, int* flag) {
    if(tree == NULL) return;
    if(*flag) {
        printf(" ");
    } else {
        *flag = 1;
    }
    printf("%d", tree->value);
    printTreePreorder(tree->leftNode, flag);
    printTreePreorder(tree->rightNode, flag);
}
// 按倒序打印树
void printTreePostorder(treePtr tree, int* flag) {
    if(tree == NULL) return;
    printTreePostorder(tree->leftNode, flag);
    printTreePostorder(tree->rightNode, flag);
    if(*flag) {
        printf(" ");
    } else {
        *flag = 1;
    }
    printf("%d", tree->value);
}
// 按树深度打印树
void printTreeByDepth(treePtr tree, int depth){
    if(tree == NULL || tree->depth > depth) {
        return;
```

```
    } if(tree->depth == depth) {
        if(depth) printf(" ");
        printf("%d", tree->value);
    } else {
        printTreeByDepth(tree->leftNode, depth);
        printTreeByDepth(tree->rightNode, depth);
    }
}
// 检查树中有无重复值节点
int check(treePtr tree) {
    int flag[100];
    memset(flag, 100, sizeof(int));
    return checkTree(tree, flag);
}
// 递归检查树中有无重复值节点
int checkTree(treePtr tree, int flag[]) {
    if(tree == NULL) return 1;
    if(flag[tree->value-1] == 1) {
        return 0;
    } else {
        flag[tree->value-1] = 1;
        return checkTree(tree->leftNode, flag) &&
checkTree(tree->rightNode, flag);
    }
}
```

Declaration

I hereby declare that all the work done in this project titled " Tree Traversals" is of my independent effort.