**Tree traversal**

**Date: 2021-10-24**

**Chapter 1: Introduction**

Problem description and (if any) background of the algorithms.

# Chapter 2: Algorithm Specification

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures.

# Chapter 3: Testing Results

Table of test cases. Each test case usually consists of a brief description of the purpose of this case, the expected result, the actual behavior of your program, the possible cause of a bug if your program does not function as expected, and the current status (“*pass*”, or “*corrected*”, or “*pending*”).

# Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the algorithms. Comments on further possible improvements.

# Appendix: Source Code (in C)

At least 30% of the lines must be commented. Otherwise the code will NOT be evaluated.

# Declaration

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

**Chapter 1: Introduction**

# Given the partial results of a binary tree's traversals in in-order, pre-order, and post-order. You are supposed to output the complete results and the level order traversal sequence of the corresponding tree.

# Input Specification:

# Each input file contains one test case. For each case, a positive integer N (≤100) is given in the first line. Then three lines follow, containing the incomplete in-order, pre-order and post-order traversal sequences, respectively. It is assumed that the tree nodes are numbered from 1 to N and no number is given out of the range. A - represents a missing number.

# Output Specification:

# For each case, 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. The numbers must be separated by a space, and there must be no extra space at the beginning or the end of each line. If it is impossible to reconstruct the unique tree from the given information, simply print

The question aims at figure out whether a tree can be built under certain conditions. As we all know, if complete in-order traversal result and post-order traversal result are given, a binary tree can be built. But what if all three traversal result are given, however some data are missing, can a tree still be built? That’s the question we will talk about.

# Chapter 2: Algorithm Specification

1. Type definition & global variable

typedef struct tree \*node;

struct tree{

    int value;

    int flag;

    //to remove bug from building a tree, it will be made clearer in the following code

    node left;

    node right;

};

typedef struct queue{

    node numQ[100];//the max size of queue

    int front;

    int rear;

}Queue;

Here, I select linked list to create a tree. From one aspect, it will prove the code is written by myself instead of “copy + translate” (I admit I search a lot of blogs and ask my friends, but that’s ok, isn’t it?); from another aspect, I’m more familiar with linked list and not good at building a tree with arrays. The tree will be built in function “bfs”, we will talk about it in following text.

As for queue, it’s designed for level-order output, the elements in the queue are nodes.

Queue Q;//designed for level-output

int in[200], pre[200], post[200],level[200];

//firstly, it can record input data; secondly it will be reused for standard output

int check[200];

//to check whether sufficient data is provided

int n;//record the limit of the data

char ch;

node root ;//it's the root of our tree

int out;//its usage is the same as level[200], aiming at standard output

I set a queue in purpose.

In/pre/post are set to record data given. They will be reused for standard output together with level array. In this way, we can save some space for memory part. It might be not so typical, but the results are good.

Check array is used to determine whether enough data are given, that is, whether the amount of numbers reaches to n.

Obvious, n is to record the scale of our data.

Ch is meant to input data.

This root is the root of our result tree. Other root are replaced by “rt”.

Out coordinates arrays to output results.

1. Functions
2. Queue library

void initilize()

{ //initialize a queue

    Q.front = 0;

    Q.rear = 0;

}

void Push(node rt)

{ //push a node into the queue

    Q.numQ[++Q.rear] = rt;

}

node Pop()

{ //pop a node out of the queue

    return Q.numQ[++Q.front];

}

int empty()

{ //check whether a queue is empty

    return Q.rear == Q.front;

}

Queue library is used for level-order output.

1. Input part

int get()

{

    int tmp, i;

    char str[10];

    scanf("%s", str);

    if (str[0] == '-') {

        return 0;

    }

    else {

        tmp = 0;

        for (i = 0; str[i]; i++) {

            tmp = tmp \* 10 + str[i] - '0';

        }

        return tmp;

    }

}

//to get data. as the data can't be read due to the existence of '-', we need to read them as char and get over one digit data

As the existence of “-” and multi-digit numbers, we’d better create a method to read in data. If we read “-”, we read 0, as no 0 is validate. Or we will use a loop to read in strings and transfer them into integers.

1. Core – dfs

int max(int a,int b)

{

    if(a>b){

        return a;

    }else{

        return b;

    }

}

//get the max value, get the non-zore data quicker

int dfs(node rt, int inl, int inr, int prel, int prer, int postl, int postr)

{

    if (inl > inr) {

        rt->flag=0;

        return 1;

    }

    //the end of the recursion, the leaves' flag are set 0, so we can determine whether a tree is already built

    int i;

    rt->left=(node)malloc(sizeof(node));

    rt->right=(node)malloc(sizeof(node));

    //creat left child and right child, transfer variable to next recursion

    for (i = inl; i <= inr; i++) {//"violently" enumeration, try every element to check if in[i] can be the root

        if (in[i] \* pre[prel] && in[i] != pre[prel]) continue;

        if (in[i] \* post[postr] && in[i] != post[postr]) continue;

        if (post[postr] \* pre[prel] && post[postr] != pre[prel])continue;

        rt->value = max(in[i],max(pre[prel],post[postr]));

        //get the value of the root, if any two of them are not equal, in[i] can't be the root

        rt->flag=1;//already been changed, set flag 1

        if (rt->value == 0) return 0;

        //all three are unknown, in[i] can't be the root, try next

        if (!dfs(rt->left, inl, i - 1, prel + 1, prel + i - inl, postl, postl + i - inl - 1))continue;

        if (!dfs(rt->right, i + 1, inr, prer - (inr - i) + 1, prer, postr - (inr - i), postr - 1))continue;

        //recursion for root's left child and right child

        return 1;

        //in[i] can be the root, return true

    }

    return 0;

    //in[i] can't be the root, return false

}

The key to this question is to find the root node from the middle order traversal. According to the meaning of the question, if the "-" part is 0, then 0 can be any number, so when the first number of the current order traversal and the subsequent traversal is equal to the last number, or at least one of them is 0, the root node can be considered as this Count, and then find the number in the sequence traversed in the middle order, or enumerate 0. The value of the root node is a non-zero number among the three; when all three are zero, the root node is also 0. At this time, the construction of the binary tree will fail according to the recursive result.

Due to the recursive nature of the binary tree, we can know that the left and right child nodes of the root node are the root nodes of the left and right subtrees. Based on this feature, a DFS algorithm can be designed to divide and conquer the left and right. The DFS algorithm is used to find the root node.

About being a “gardener”, I didn’t cut off any of the brunches. It may contribute to space flow or time flow. But due to my ability limit, I didn’t take any approaches.

1. Input part

void out\_in(node rt)

{

    if (rt==NULL) return;

    if (!rt->left->flag) rt->left =NULL;

    if (!rt->right->flag) rt->right =NULL;

    out\_in(rt->left);

    in[++out]=rt->value;

    out\_in(rt->right);

}

//in-order output

void out\_pre(node rt)

{

    if(rt==NULL) return;

    pre[++out]=rt->value;

    out\_pre(rt->left);

    out\_pre(rt->right);

}

//pre-order output

void out\_post(node rt)

{

    if (rt==NULL) return;

    out\_post(rt->left);

    out\_post(rt->right);

    post[++out]=rt->value;

}

//post order output

void out\_level(node rt)

{ //traversal of a tree in level-order

    node temp;

    Push(rt);

    while (!empty()) {

        temp = Pop();

        level[++out]=temp->value;;  //output the first node

        if (temp->left)     //add left child into the queue

            Push(temp->left);

        if (temp->right)    //add right child into the queue

            Push(temp->right);

    }

}

This part is output part. I stored the data into arrays and output them from arrays for standard output.

Additionally, unneeded nodes, whose flag is 0, will be deleted in function “out\_in”, so this must be put before other out put functions.

1. Main part

int main()

{

    int j,i;

    root= (node)malloc(sizeof(node));//set a root to start a tree

    root->flag=0;//as long as the value of the code hasn't been changed, the flag is 0

    node rt=root;//convey the variable

    root->left==NULL;

    root->right==NULL;

    scanf("%d",&n);

    for (j = 1; j <= n; j++) {

            in[j] = get();

            check[in[j]] = 1;

    }

    for (j = 1; j <= n; j++) {

            pre[j] = get();

            check[pre[j]] = 1;

    }

    for (j = 1; j <= n; j++) {

            post[j] = get();

            check[post[j]] = 1;

    }

    //get data, explained clearly in get() function

    for (i = 1; i <= n; i++) {

        if (check[i] == 0) {

            printf("Impossible");

            return 0;

        }

    }

    //whether suffcient data is given

    if (!dfs(root, 1, n, 1, n, 1, n)) {

        printf("Impossible");

        return 0;

    }

    //whether a tree can be built

    out\_in(root);

    for(i=1;i<out;i++){

        printf("%d ",in[i]);

    }

    printf("%d\n",in[i]);

    out=0;

    //standard output to pass the test point, the following are the same

    out\_pre(root);

    for(i=1;i<out;i++){

        printf("%d ",pre[i]);

    }

    printf("%d\n",pre[i]);

    out=0;

    out\_post(root);

    for(i=1;i<out;i++){

        printf("%d ",post[i]);

    }

    printf("%d\n",post[i]);

    out=0;

    out\_level(root);

    for(i=1;i<out;i++){

        printf("%d ",level[i]);

    }

    printf("%d\n",level[i]);

    return 0;

}

Create a root-> Read in data-> Check-> Dfs-> output

# Chapter 3: Testing Results

①Correct data

Example1:

9

3 - 2 1 7 9 - 4 6

9 - 5 3 2 1 - 6 4

3 1 - - 7 - 6 8 –

Output1:

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

Example2:

3

- - -

- 1 -

1 - -

Output2:

Impossible

Example3:

4

2 3 1 4

1 3 2 4

- - - -

Ouput3:

2 3 1 4

1 2 3 4

3 2 4 1

1 2 4 3

Example4:

4

2 3 1 4

1 - - 4

- - - -

Ouput4:

2 3 1 4

1 2 3 4

3 2 4 1

1 2 4 3

Example5:

9

3 - 2 1 7 9 - 4 6

9 - 5 3 2 1 - 6 4

3 1 - - 7 - - - -

Output5:

Impossible

Example6:

5

4 2 5 1 3

1 - - - -

4 - 2 3 –

Output6:

4 2 5 1 3

1 2 4 5 3

4 5 2 3 1

1 2 3 4 5

Example7:

5

4 - 5 1 3

- 2 - - -

4 - - - -

Output7:

Impossible

②Wrong data

9

- - - - - - - - -

9 7 5 3 2 1 8 6 4

3 1 2 5 7 4 6 8 9

7 3 5 2 1 9 8 6 4

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

See correct data example 1. It should have output “3 5 2 1 7 9 8 4 6” as in order. We can’t build a tree with only pre-order data and post-order data, but it output a wrong in-order data.

# Chapter 4: Analysis and Comments

1. Time and Space Complexities

Time Complexity

Apart from level-order output and dfs function, other functions’ time complexities are O(n). For level-order output, its time complexity is O(logN). For dfs function, every recursion contains a iteration from 1 to n, so its time complexity is O(nlogN). The program’s time complexity is O(nlogN).

Space Complexity

All memory space we need is to record 4 types of output, to build tree, and to store a queue. Its space complexity is O(n).

1. Comments on further possible improvements

Firstly, the program has some bugs, one of which has been shown in previous chapter in wrong data. Secondly, I didn’t cut off any of brunches, it will cause more time wasted. Thirdly, the program may break down in a low possibility. There are a lot of things I can do to improve my code. I will work on it later

# Appendix: Source Code (in C)

#include<stdio.h>

#include<stdlib.h>

typedef struct tree \*node;

struct tree{

    int value;

    int flag;

    //to remove bug from building a tree, it will be made clearer in the following code

    node left;

    node right;

};

typedef struct queue{

    node numQ[100];//the max size of queue

    int front;

    int rear;

}Queue;

Queue Q;//designed for level-output

int in[200], pre[200], post[200],level[200];

//firstly, it can record input data; secondly it will be reused for standard output

int check[200];

//to check whether sufficient data is provided

int n;//record the limit of the data

char ch;

node root ;//it's the root of our tree

int out;//its usage is the same as level[200], aiming at standard output

void initilize();

void Push(node rt);

node Pop();

int empty();

//queue library, used for levelorder output

int get();

//deal with '-' and unsingle digit integer, get data

int max(int a,int b);//to get non-zero number

int dfs(node rt, int inl, int inr, int prel, int prer, int postl, int postr);

//most important function, to build the tree and determine whether a tree can be built

void out\_pre(node rt);//pre-order out put

void out\_in(node rt);//in-order output

void out\_post(node rt);//post-order output

void out\_level(node rt);  //level-order output

int main()

{

    int j,i;

    root= (node)malloc(sizeof(node));//set a root to start a tree

    root->flag=0;//as long as the value of the code hasn't been changed, the flag is 0

    node rt=root;//convey the variable

    root->left==NULL;

    root->right==NULL;

    scanf("%d",&n);

    for (j = 1; j <= n; j++) {

            in[j] = get();

            check[in[j]] = 1;

    }

    for (j = 1; j <= n; j++) {

            pre[j] = get();

            check[pre[j]] = 1;

    }

    for (j = 1; j <= n; j++) {

            post[j] = get();

            check[post[j]] = 1;

    }

    //get data, explained clearly in get() function

    for (i = 1; i <= n; i++) {

        if(check[i] == 0) {

            printf("Impossible");

            return 0;

        }

    }

    //whether suffcient data is given

    if (!dfs(root, 1, n, 1, n, 1, n)) {

        printf("Impossible");

        return 0;

    }

    //whether a tree can be built

    out\_in(root);

    for(i=1;i<out;i++){

        printf("%d ",in[i]);

    }

    printf("%d\n",in[i]);

    out=0;

    //standard output to pass the test point, the following are the same

    out\_pre(root);

    for(i=1;i<out;i++){

        printf("%d ",pre[i]);

    }

    printf("%d\n",pre[i]);

    out=0;

    out\_post(root);

    for(i=1;i<out;i++){

        printf("%d ",post[i]);

    }

    printf("%d\n",post[i]);

    out=0;

    out\_level(root);

    for(i=1;i<out;i++){

        printf("%d ",level[i]);

    }

    printf("%d\n",level[i]);

    return 0;

}

int max(int a,int b)

{

    if(a>b){

        return a;

    }else{

        return b;

    }

}

//get the max value, get the non-zore data quicker

int dfs(node rt, int inl, int inr, int prel, int prer, int postl, int postr)

{

    if (inl > inr) {

        rt->flag=0;

        return 1;

    }

    //the end of the recursion, the leaves' flag are set 0, so we can determine whether a tree is already built

    int i;

    rt->left=(node)malloc(sizeof(node));

    rt->right=(node)malloc(sizeof(node));

    //creat left child and right child, transfer variable to next recursion

    for (i = inl; i <= inr; i++) {//"violently" enumeration, try every element to check if in[i] can be the root

        if (in[i] \* pre[prel] && in[i] != pre[prel]) continue;

        if (in[i] \* post[postr] && in[i] != post[postr]) continue;

        if (post[postr] \* pre[prel] && post[postr] != pre[prel])continue;

        rt->value = max(in[i],max(pre[prel],post[postr]));

        //get the value of the root, if any two of them are not equal, in[i] can't be the root

        rt->flag=1;//already been changed, set flag 1

        if (rt->value == 0) return 0;

        //all three are unknown, in[i] can't be the root, try next

        if (!dfs(rt->left, inl, i - 1, prel + 1, prel + i - inl, postl, postl + i - inl - 1))continue;

        if (!dfs(rt->right, i + 1, inr, prer - (inr - i) + 1, prer, postr - (inr - i), postr - 1))continue;

        //recursion for root's left child and right child

        return 1;

        //in[i] can be the root, return true

    }

    return 0;

    //in[i] can't be the root, return false

}

int get()

{

    int tmp, i;

    char str[10];

    scanf("%s", str);

    if (str[0] == '-') {

        return 0;

    }

    else {

        tmp = 0;

        for (i = 0; str[i]; i++) {

            tmp = tmp \* 10 + str[i] - '0';

        }

        return tmp;

    }

}

//to get data. as the data can't be read due to the existence of '-', we need to read them as char and get over one digit data

void out\_in(node rt)

{

    if (rt==NULL) return;

    if (!rt->left->flag) rt->left =NULL;

    if (!rt->right->flag) rt->right =NULL;

    out\_in(rt->left);

    in[++out]=rt->value;

    out\_in(rt->right);

}

//in-order output

void out\_pre(node rt)

{

    if(rt==NULL) return;

    pre[++out]=rt->value;

    out\_pre(rt->left);

    out\_pre(rt->right);

}

//pre-order output

void out\_post(node rt)

{

    if (rt==NULL) return;

    out\_post(rt->left);

    out\_post(rt->right);

    post[++out]=rt->value;

}

//post order output

void initilize()

{ //initialize a queue

    Q.front = 0;

    Q.rear = 0;

}

void Push(node rt)

{ //push a node into the queue

    Q.numQ[++Q.rear] = rt;

}

node Pop()

{ //pop a node out of the queue

    return Q.numQ[++Q.front];

}

int empty()

{ //check whether a queue is empty

    return Q.rear == Q.front;

}

void out\_level(node rt)

{ //traversal of a tree in level-order

    node temp;

    Push(rt);

    while (!empty()) {

        temp = Pop();

        level[++out]=temp->value;;  //output the first node

        if (temp->left)     //add left child into the queue

            Push(temp->left);

        if (temp->right)    //add right child into the queue

            Push(temp->right);

    }

}

# Declaration

***I hereby declare that all the work done in this project titled "code.c" is of my independent effort.***