**Tree Traversals**

刘思锐

**Date: 2021-10-24**

# Chap.1-Introduction

It’s easy to prove that one can reconstruct a unique binary tree according to its in-order traversal and another traversal. The question is, if parts of the traversal sequences are missing, can you still reconstruct the unique binary tree?

In this project, the number of nodes will be given in the first line. Incomplete in-order, pre-order and post-order traversal will be given in next 3 lines, respectively.

It’s your turn now to figure out whether you can reconstruct a unique tree according to them. If you can, print complete in-order, pre-order, post-order and level order traversal result in 4 lines. And if not (including 2 cases: A. Cannot reconstruct a tree with given traversal. B. There are more than one tree that correspond to the given incomplete traversal.), just print “Impossible” on screen.

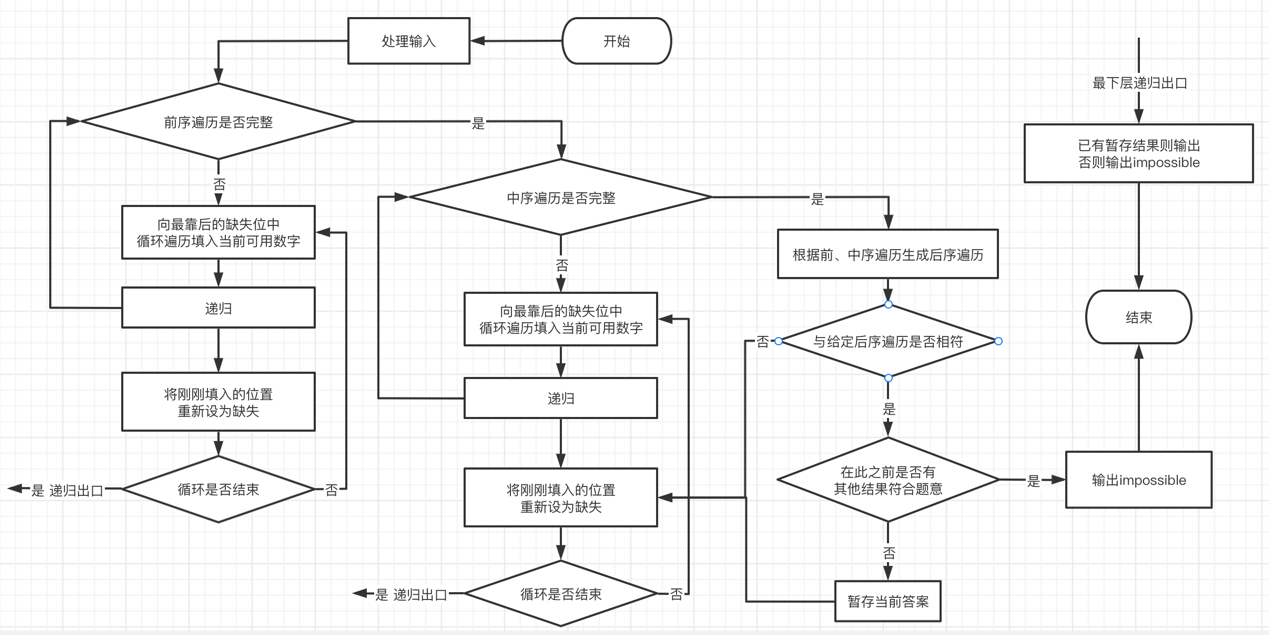
To make it a little simpler, tree nodes are numbered from 1 to N , no repetition and no number is given out of the range.

# Chap.2-Algorithm Specification

As we all know that one can reconstruct a binary tree according to its in-order and another traversal. My algorithm consists of three parts.

Part 1: Recurse to find all possible complete arrangements of in-order and pre-order traversal. To do this, the missing number and the index of missing places will be saved at first.

Part 2: Try to reconstruct a tree by the traversal completed in Part 1. If feasible, compare the post-order traversal of it with the given post-order traversal. If not, just back to Part 1 and try another sequence.

 Part 3: Assume that a tree has been successfully reconstructed and there is no conflict between this tree and the given post-order traversal. Save the tree and leave a flag which indicate that we have already own a result. Traversal in Part 1 cannot stop yet because we need to know whether the result is unique. The complete traversals will not print until tried all possible arrangements in Part 1 and 2.

# Chap.3-Testing Result

Example 1 is an ordinary test case that contains no so much node and can reconstruct a tree. It also shows the format of input and output.

1. Input Example 1：
2. 9                   //number of nodes
3. 3 - 2 1 7 9 - 4 6   //in-order traversal
4. 9 - 5 3 2 1 - 6 4   //pre-order traversal
5. 3 1 - - 7 - 6 8 -   //post-order traversal
6. Output Example 1：
7. 3 5 2 1 7 9 8 4 6   //complete in-order traversal
8. 9 7 5 3 2 1 8 6 4   //complete pre-order traversal
9. 3 1 2 5 7 4 6 8 9   //complete post-order traversal
10. 9 7 8 5 6 3 2 4 1   //complete level order traversal

Example 2 is a test case that failed to reconstruct a tree for they are impossible traversal results.

1. Input Example 2:
2. 9
3. 1 - 3 5 9 8 - - 2
4. 7 6 - 2 - 9 1 - -
5. 3 5 8 1 9 - - 4 -
6. Output Example 2:
7. Impossible

Example 3 is a test case that failed to reconstruct a tree for the result is not unique.

1. Input Example 3:
2. 10
3. 1 2 3 4 5 6 7 - - -
4. 10 - - 8 - - 7 - - -
5. - - - - - - - - - -
6. Output Example 3:
7. Impossible

Example 4 proves that you cannot reconstruct a tree by only pre-order and post-order traversal.

1. Input Example 4:
2. 20
3. - - - - - - - - - - - - - - - - - - - -
4. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
5. 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
6. Output Example 4:
7. Impossible

Example 5, otherwise, proves that it’s feasible to reconstruct a tree by its in-order and post-order traversal.

1. Input Example 5:
2. 10
3. 1 2 3 4 - 6 7 8 - 10
4. - - - - - - - - - -
5. 10 - 8 7 6 5 - 3 2 -
6. Output Example 5:
7. 1 2 3 4 5 6 7 8 9 10
8. 1 2 3 4 5 6 7 8 9 10
9. 10 9 8 7 6 5 4 3 2 1
10. 1 2 3 4 5 6 7 8 9 10

Example 6 is obviously right because I don’t need post-order traversal to reconstruct my tree.

1. Input Example 6:
2. 20
3. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 -
4. - 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
5. - - - - - - - - - - - - - - - - - - - -
6. Output Example 6:
7. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
8. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
9. 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
10. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Example 7 is an extreme test case with a very small size.

1. Input Example 7:
2. 2
3. 1 -
4. - -
5. 2 1
6. Output Example 7:
7. 1 2
8. 1 2
9. 2 1
10. 1 2

Example 8 is also and extreme test case which is obviously impossible.

1. Input Example 8:
2. 100
3. - - - - (else omitted)
4. - - - - (else omitted)
5. - - - - (else omitted)
6. Output Example 8:
7. Impossible

# Chap.4-Analysis and Comment

There’re at most 2N Missing numbers in in-order and pre-order traversal altogether. It means I need at most 2N levels of recursion to complete these two sequences. The space for input and output is also of N complexity. So the space complexity is *O( N ).*

There’re at most N missing numbers in in-order traversal. Another N missing numbers in pre-order traversal as well. For each of them there’re N! cases to traverse all complete sequence. For each sensible in-order and pre-order traversal, I need a linear time complexity to reconstruct a tree and another linear time complexity to compare it with given post-order traversal. It also needs some time to handle input and output. So the time complexity of it is *O( (N!)^2 )*.

I know that it’s an algorithm of low efficiency. There’s a better way:

Treat incomplete traversals just like the complete. Pick out the first element of pre-order traversal or the last element of post-order traversal and locate it in in-order traversal. If you successfully locate it, continue to recursion. The only different is, if both first element of pre-order traversal and the last element of post-order traversal are missing or it absent in in-order traversal, iterate to try all possible numbers and positions. It seems not complicated but actually, it troublesome to code a recursion function like that.

# Appendix-Source Code

#include <iostream>

#include <vector>

#include <string>

using namespace std;

#define MISSING (-1)

//only used to print level order traversal

struct tree {

int element;

struct tree\* left;

struct tree\* right;

};

int N, traversalCount, postOrderCount;

vector<int> inOrder, preOrder, postOrder, tempPostOrder;

//to store the result

vector<int> inResult, preResult, postResult;

//numbers that not being used in each traversal, for node is numbered from 1 to N

vector<int> inMissingNumber, preMissingNumber, postMissingNumber;

//index that '-' is at in each traversal

vector<int> inMissingIndex, preMissingIndex, postMissingIndex;

//read a non-negative number from buffer separated by any character other than '0'~'9'

//return #define MISSING when encounter '-' with no figure in front of it

//exit(1) if no more figure 'till EOF

int ReadNumber()

{

char temp, flag=0, num=0;

while(true){

temp=getchar();//get a new char

if(temp>='0' && temp<='9'){//encounter a figure

num\*=10;

num+=temp-'0';

flag=1;

}else if(!flag && temp=='-'){//encounter a '-' with no figure before

return MISSING;

}else if(flag){//encounter a separator

return num;

}else if(!flag && temp==EOF){//no more figure

printf("NO MORE FIGURE!");

exit(1);

}

}

}

//load given traversal into corresponding vector

void Load()

{

for(int i=0; i<N; i++) inOrder.push\_back(ReadNumber());

for(int i=0; i<N; i++) preOrder.push\_back(ReadNumber());

for(int i=0; i<N; i++) postOrder.push\_back(ReadNumber());

}

//find out which number is absent in three traversal vector

//store result into vector<int> xxxMissing

void GetMissingNumber()

{

int inFlag, preFlag, postFlag;

for(int i=1; i<=N; i++){

inFlag=0; preFlag=0; postFlag=0;

for(int j=0; j<N; j++){//traverse from 1 to N, for node is numbered from 1 to N

if(inOrder[j]==i && !inFlag) inFlag=1;

if(preOrder[j]==i && !preFlag) preFlag=1;

if(postOrder[j]==i && !postFlag) postFlag=1;

}

//if not present, push i

if(!inFlag) inMissingNumber.push\_back(i);

if(!preFlag) preMissingNumber.push\_back(i);

if(!postFlag) postMissingNumber.push\_back(i);

}

}

//store the index of element MISSING of each traversal into vector<int> xxxMissingIndex

void GetMissingIndex()

{

for(int i=0; i<N; i++){

if(inOrder[i]==MISSING) inMissingIndex.push\_back(i);

if(preOrder[i]==MISSING) preMissingIndex.push\_back(i);

if(postOrder[i]==MISSING) postMissingIndex.push\_back(i);

}

}

//check if tempPostOrder correspond with postOrder. if so, traversalCount++ and the traversal will be stored in vector<int> xxxResult

//once traversalCount>1, print "Impossible" and exit(1) because it means the result is not unique

void CheckPostOrder()

{

for(int i=0; i<N; i++){//check if correspond

if(postOrder[i]!=MISSING && postOrder[i]!=tempPostOrder[i]) return;

}

if(traversalCount){//not unique, Impossible

printf("Impossible");

exit(1);

}else{//available result, save it

traversalCount++;

inResult.assign(inOrder.begin(), inOrder.end());

preResult.assign(preOrder.begin(), preOrder.end());

postResult.assign(tempPostOrder.begin(), tempPostOrder.end());

}

}

//try to restore the post-order traversal according to in-order and pre-order traversal if it's possible

//temporary result will be stored in tempPostOrder

int GeneratePostOrder(int in\_begin, int pre\_begin, int post\_begin, int length)

{

int rootIndex, i, flag=1, leftCnt, rightCnt;

for(i=0; i<length; i++){//preOrder[pre\_begin] must be the root element

if(inOrder[i+in\_begin]==preOrder[pre\_begin]){

rootIndex=i+in\_begin;

break;

}

}

if(i>=length){ //cannot find root element in inOrder traversal, cannot be a tree

return 0;

}else{ //root element is available in inOrder traversal

postOrderCount++;

tempPostOrder[post\_begin+length-1]=preOrder[pre\_begin];//save the result

//try restore left sub-tree

leftCnt=rootIndex-in\_begin;//node number of left sub-tree

if( leftCnt > 0) flag=GeneratePostOrder(in\_begin, pre\_begin+1, post\_begin, leftCnt);

//try restore right sub-tree

rightCnt=in\_begin+length-rootIndex-1;//node number of right sub-tree

if(flag && rightCnt > 0) flag=GeneratePostOrder(rootIndex + 1, pre\_begin + leftCnt + 1,post\_begin + leftCnt, rightCnt);

if(postOrderCount==N) {//have traversed all node and get a possible result

CheckPostOrder();

postOrderCount=0;//don't check a same result multiple times in different levels of recursion

}

return flag;

}

}

//use recursion to find all possible pre-order traversal sequence

void FillPreOrder()

{

if(preMissingNumber.empty()){//have filled all MISSING places

postOrderCount=0;

GeneratePostOrder(0, 0, 0, N);

}else{

int tempNumber, tempIndex;

vector<int>::iterator iter;

for(iter=preMissingNumber.begin(); iter<preMissingNumber.end(); iter++){

//preparation of recursion

tempNumber=\*iter;

preMissingNumber.erase(iter);

tempIndex=\*(--preMissingIndex.end());

preMissingIndex.pop\_back();

//fill preOrder traversal sequence

preOrder[tempIndex]=tempNumber;

//recursion

FillPreOrder();

//restore variables

preMissingNumber.insert(iter, tempNumber);

preMissingIndex.push\_back(tempIndex);

}

}

}

//use recursion to find all possible in-order traversal sequence

void FillInOrder()

{

if(inMissingNumber.empty()){//have filled all MISSING placed

FillPreOrder();

}else{

int tempNumber, tempIndex;

vector<int>::iterator iter;

for(iter=inMissingNumber.begin(); iter<inMissingNumber.end(); iter++){

//preparation of recursion

tempNumber=\*iter;

inMissingNumber.erase(iter);

tempIndex=\*(--inMissingIndex.end());

inMissingIndex.pop\_back();

//fill inOrder traversal sequence

inOrder[tempIndex]=tempNumber;

//recursion

FillInOrder();

//restore

inMissingNumber.insert(iter, tempNumber);

inMissingIndex.push\_back(tempIndex);

}

}

}

//creat a binary tree according to inResult and postResult

struct tree\* CreatTree(int begin, int end)

{

static int offset=0;

//exit of recursion

if(begin > end) return 0;

//postOrder[end] is the root of current tree

//find the position of root in inorder array

int rootElement, rootPosition;

rootElement=postResult[end];

for(rootPosition= begin + offset; rootPosition <= end + offset; rootPosition++)

if(inResult[rootPosition]==rootElement) break;

//creat new tree by recursion

struct tree \*root;

root=new tree;

root->element=rootElement;

//rootPosition refer to the index of inOrder

//begin and end refer to the index of postOrder

//take care of this! What's the begin and end next time?

offset++;

root->right=CreatTree(rootPosition-offset+1, end - 1);

offset--;

root->left=CreatTree(begin, rootPosition - offset - 1);

return root;

}

//print level order traversal according to inResult and postResult

void PrintLevelOrder()

{

struct tree\* queue[N];

struct tree\* t= CreatTree(0, N-1);//creat a tree

int begin=0, end=0, tempend;

queue[end++]=t;

while(begin<end){

tempend=end;

for(int i=begin; i<tempend && i<N; i++){

//print the element in a same level

printf("%d ", queue[i]->element);

//push the child node into queue[]

//for queue[begin]~queue[end] must be in a same level, the child node must be in a same level as well

if(queue[i]->left) queue[end++]=queue[i]->left;

if(queue[i]->right) queue[end++]=queue[i]->right;

begin=tempend;

}

}

printf("\n");

}

void PrintResult()

{

//print in-order traversal

for(int i=0; i<N; i++) printf("%d ", inResult[i]);

printf("\n");

//print pre-order traversal

for(int i=0; i<N; i++) printf("%d ", preResult[i]);

printf("\n");

//print post-order traversal

for(int i=0; i<N; i++) printf("%d ", postResult[i]);

printf("\n");

//print level order traversal

PrintLevelOrder();

}

int main()

{

//preparation

scanf("%d", &N);

tempPostOrder.resize(N);

Load();

GetMissingNumber();

GetMissingIndex();

//start to traverse all possible cases

FillInOrder();

if(traversalCount){

PrintResult();

}else{

printf("Impossible\n");

}

return 0;

}

# Declaration

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