Tree Traversals

Xu Shengze

Date: 2021**-10-2**8

Chapter 1: Introduction

This question is given the partial results of the middle order, preorder and postorder traversal of a binary tree, and requires us to write a program to judge whether the unique tree can be reconstructed according to the given information. If the conclusion is affirmative, the complete result and level order of the corresponding tree will be output Traverse the sequence.

In the process of answering this question, my program is mainly divided into two parts, one is to find the root node based on the given information, and the other is to rebuild the tree, including several traversals of the tree and the corresponding output. In the specific implementation process, I mainly rebuild the tree by operating the defined global variables. The depth-first search algorithm is used to violently enumerate the values of the middle-order traversal sequence to determine the nodes of each level of the tree. In this process, we also judge the "possibility" required in the stem of the problem. After multiple recursions, pass the corresponding The operation obtains the complete middle-order, pre-order and post-order traversal results, and then traverses the reconstructed tree in sequence.

Chapter 2: Algorithm Specification

The data structure mainly used in this project is mainly an array. For the sake of simplicity, I did not use a structure. The related data structure and variables are declared as follows.

```
int n, a[max], b[max], c[max], num[max], guess[max];
//n is a given positive number, a[], b[], c[] store the input traversal sequence
in middle, pre-, and post-order respectively
int tree[max][2], root;
//root is used to store the root node, tree[][] stores left and right subtrees
int aa[max],bb[max],cc[max];
//these three arrays are used to store the subscript when a certain value appears
int in[max],pre[max],post[max];
//Complete mid-order, pre-order and post-order sequence
char s[max];
//for input string
int flag,temp;
int height[max][max]={0};
//use a two-dimensional array to store the value of a node in a certain layer of
the tree, which is convenient for output when traversing the layer sequence
int length[max]={0};
//use an array to store the number of nodes at a certain level of the tree
```

2.1 Tree reconstruction

This part is to judge whether it is possible to build a unique tree. You need to use a function `int Findroot(int *x, int inleft, int inright, int preleft, int preright, int postleft, int postright)`. The incoming parameters of the function are used to Store the variable of the root node and the subscripts of the beginning and end positions of the three arrays.

```
Input:&x,inleft,inright,preleft,preright,postleft,postright
if inleft <= inright then
    //Start traversal and inspection
    if b[preleft]!=0 && c[postright]!=0 && b[preleft] != c[postright] then
        return 0;
    end if
    for i ← inleft to inright do
        if a[i]!=0 then
            if b[preleft] && a[i] != b[preleft] then
                continue;
            end if
            if c[postright] && a[i] != c[postright] then
                continue;
            end if
        end if
        //Assign value to array
        guess[i] ← max(a[i],max(b[preleft],c[postright]));
        *x ← i;
        //Continue to test
        if aa[guess[i]] && aa[guess[i]]!=i then
            continue;
        end if
        if bb[guess[i]] && bb[guess[i]]!=preleft then
            continue;
        end if
        if cc[guess[i]] && cc[guess[i]]!=postright then
            continue;
        end if
        //Check the left and right subtrees
        if Findroot(Left Subtree)!=1 then
            continue;
        end if
        if Findroot(Right Subtree)!=1 then
            continue;
        end if
        //If the test passes, the function returns 1
        return 1;
    end for
    return 0;
else
    *x ← 0;
    return 1;
end if
```

In the above program, `x` stores the location of the target root node. Through traversal, all cases where it can be confirmed that the node is not the root node are eliminated, and then the value is stored in the array `guess[]` after the root node is determined. After verifying the existence and rationality of the root node, call this function to verify the existence and rationality of its left and right subtrees respectively. This process is done by accessing `tree[*x][0]` and `tree[*x][1]` is implemented, and before that, we need to check whether the value found is indeed the root node to check the uniqueness. If all the conditions are reasonable, the integer value `1` is returned to represent `true`, and if the possible root node is still not found after all the loops are over, the integer value `0` is returned to represent `false`.

After the program ends, we get the value of the root node, and also store the value of its child nodes through a two-dimensional array. According to this process, the values and relative positions of all nodes are obtained, which is convenient for the use of the `restore()` function later.

2.2 Tree traversal

2.2.1 VLR、LDR、LRD

This part mainly uses three similar functions `void restore(int x)` to get the complete pre-order, middle-order and post-order traversal sequence. Let's take the middle-order traversal as an example to illustrate the principle of the function.

```
Input:x
if !x then
    return
end if
restorein(Left Subtree);
inoreder[temp++] ← guess[x];
restorein(Right Subtree);
```

Since the middle-order traversal is to first traverse the left subtree, then visit the root node, and finally traverse the right subtree, we call the function itself on the left subtree in the function, and then assign the value of the root node to the array. The element of the position, and then call the function on the right subtree, so that the traversal of the "left→root→right" order is realized.

In the program, `temp` is a pre-defined variable, which is re-assigned to `1` before operating on the pre-order, middle-order, and post-order traversal sequence, and the operation on the left and right subtrees is done by accessing the previously defined `tree[x][0]` and `tree[x][1]` are implemented so that when `return`, an array `inorder[]` with subscripts from `1` to `n` is obtained, The complete in-order traversal sequence is stored.

2.2.2 Level order traversal

In this part, we will rebuild the tree using the mid-order and post-order traversal sequences obtained above, use a two-dimensional array to store the nodes of each layer of the tree, and use a `void CreateList(int inleft,int inright,int postleft,int postright,int level)` function.

```
Input:inleft,inright,postleft,postright,level
temp ← inleft
if inleft>inright then
    return;
end if
if postleft>postright then
    return;
end if
height[level][length[level]] ← post[postright];
length[level]++;
while in[temp]!=post[postright] do
    temp++;
end while
len ← temp - inleft;
CreatList(Left Subtree);
CreatList(Right Subtree);
```

In this function, since the post-order traversal sequence is known, we can directly get the value and position of the root node and store it in a two-dimensional array. Then we compare the elements of the middle-order traversal sequence with the value of the root node to obtain the relative position of the root node in the middle-order sequence, and then call the function to operate on the left and right subtrees respectively, and the values of the left and right subtrees can also be changed. Stored in a two-dimensional array in turn, the specific implementation process is `CreateList(inleft,temp-1,postleft,postleft+len-1,level+1)` and `CreateList(temp+1,inright,postleft+len,postright -1,level+1)`.

In the above function, `height[level][]` accesses a certain level of the tree, and `length[level]` stores the number of nodes at that level, so we store the tree through a two-dimensional array All nodes access a unique node through two subscripts.

Next, we need to output the sequence traversed by the layer order. I used a function `void PrintBFS(int **height,int length[],int n,int level)`. The principle of the function is very simple. The output is traversed through two loops. sequence.

```
cnt ← 0
while cnt < level do
    if cnt == 0 then
        Print without spaces
    else
        for i ← 0 to length[cnt]
            do Print with spaces
        end for
    end if
    cnt ← cnt + 1;
end while
```

The incoming parameters of this function need to be processed in the main program, and then the complete layer sequence traversal output result can be obtained.

2.3 Main program

2.3.1 Input

Convert the three input strings into an array of integer type, and use the integer `0` to replace the `-` in the string.

```
for i ← 1 to n do
    scanf(s);
    a[i] ← 0;
    if strcmp(s,"-")==0 then
        a[i] ← 0;
    else
        for j=0 to s[j]!='\0' do
            a[i] ← a[i]*10+s[j]-'0';
        end for
    end if
    aa[a[i]] ← i;
    num[a[i]] ← num[a[i]] + 1;
end for
```

In this part, we also assign values to the two previously defined global array variables, among which `aa[]` stores the subscript of `a[i]`, which is used to construct the tree, and `num[]` Used to store the number of occurrences of a certain integer `a[i]`, which is convenient for counting the number of numbers that do not appear.

2.3.2 Output

The output is mainly divided into two parts. The first part is the output preorder, middle order and postorder traversal. Just simply traverse the `inorder[]`, `preorder[]` and `postorder[]` we got before. , The other part is the output of the layer sequence traversal, just call the function `PrintBFS()`, which has been explained in detail above.

2.3.3 Some processing

In the main program, we have several operations that need to be explained separately.

For example, we need to count the number of occurrences of each number in the three sequences. If the number of digits that do not appear is ≥ 2, the unique tree cannot be reconstructed at this time, because the two numbers that do not appear can be changed in order at will. The program can directly output `Impossible`.

```
for i ← 0 to n do
    if num[i] = 0 then
        number++;
    end if
end for
```

In addition, when we output the layer sequence traversal sequence, we need to get the number of levels of the tree `level`, which can be achieved by looping, adding the number of nodes in each layer each time until the total number is reached.

```
for i=0 to num=n do
    num ← num + length[i]
    level ← level + 1
end for
```

This makes it easier to get the total number of levels of the tree, and it is also convenient for us to check for problems when debugging the program.

In addition, when outputting the result of layer sequence traversal, we need to pass a two-dimensional array as a parameter in our function, which cannot be passed directly. We then define a one-dimensional array to store the address for transmission. However, it turned out later that perhaps the program can be made more concise without this function, but due to habit and out of the idea of modularizing the program, I still did such a more complicated process.

```
int *p[maxn];
for i ← 0 to level do
    p[i] ← height[i];
end for
```

Chapter 3: Testing Results

In the PAT (Top Level) Practice, the code compiles correctly and passes all test points. Here are some use cases.

First check the two test cases given in the question.

Sample Input 1:

```
9
3 - 2 1 7 9 - 4 6
9 - 5 3 2 1 - 6 4
3 1 - - 7 - 6 8 -
```

Sample Output 1:

```
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
```

Sample Input 2:

```
3
- - -
- 1 -
1 - -
```

Sample Output 2:

```
Impossible
```

This is an `Impossible` situation given in the title, because it is impossible to construct a unique tree, which is very easy to directly verify manually.

Sample Input 3:

```
1
-
-
-
```

Sample Output 3:

```
1
1
1
1
```

This is the simplest case, and the output is normal after the inspection program.

Sample Input 4:

```
100
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - -
```

Sample Output 4:

```
Impossible
```

This is a more complicated situation, but the time complexity and space complexity of this program are low, so the correct result is obtained in a short time. At the same time, because the tree corresponding to this situation is not unique, the output is `Impossible`.

Sample Input 5:

```
100
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75
74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75
74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

Sample Output 5:

```
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75
74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75
74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

This is also the case when `n` is large, but the unique tree can be rebuilt, and the program outputs normally.

The following are a few more common situations, just to test the performance and correctness of the program.

Sample Input 6:

```
10
2 - 3 7 1 5 8 10 9 -
- 4 2 3 1 - 6 8 9 -
2 7 1 3 - - 9 8 6 -
```

Sample Output 6:

```
2 4 3 7 1 5 8 10 9 6
5 4 2 3 1 7 6 8 9 10
2 7 1 3 4 10 9 8 6 5
5 4 6 2 3 8 1 9 7 10
```

Sample Input 7:

```
10
2 4 6 8 10 - - - - -
1 - 3 - 5 - 7 - 9 -
- - - - - - - - - -
```

Sample Output 7:

```
2 4 6 8 10 9 7 5 3 1
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10
```

Sample Input 8:

```
10
2 4 6 8 10 - - - - -
- 2 3 - 5 - 7 - 9 -
- - - - - - - - - -
```

Sample Output 8:

```
Impossible
```

This situation is similar to the previous one, but the incomplete items in the pre-order traversal sequence are changed. There is more than one possibility to rebuild the tree, and the uniqueness is not satisfied, so the program output is `Impossible`.

Chapter 4: Analysis and Comments

4.1 Time complexity

The most difficult part of the program is the `Findroot()` function. We mainly study the time complexity of this part. Through analysis, we can know that its recurrence can be roughly expressed as $T(n) = 2T(\frac{n}{2}) + cn$. According to the main theorem, $f(n) = cn = O(n^1)$, then $T(n) = O(n \log n)$. The time complexity of other parts is very obvious, basically $O(n)$ or $O(\log n)$.

For the average time of different situations, with the change of the value of `n` and the difference of the missing items of the sequence, the running time of the program is very different, so no specific analysis will be given.

4.2 Space complexity

In terms of space complexity, at most only one recursion is going on each time the program runs, and each time is divided into two, so the space complexity is $O(\log n)$.

4.3 Algorithm improvement

This program can deal with most situations with low time and space complexity, but there are still some areas that need to be improved, such as the following situation:

Sample Input 9:

```
5
- - - - -
1 2 3 4 5
3 2 5 4 1
```

Sample Output 9:

```
2 3 1 4 5
1 2 3 4 5
3 2 5 4 1
1 2 4 3 5
```

After inputting this example, the program outputs the result of its calculation. But in fact, when n is large and only the preorder and postorder traversal sequences are given, many different solutions can be easily obtained, which means that the program did not successfully judge the construction of the tree Uniqueness.

I tried to brutally exhaust all possible situations after traversing to get the root node position, so that I can judge whether the tree can be reconstructed, and I can also intuitively know whether the result is unique, but the problem that comes with this is the complexity of the program. Too high, it takes up a lot of space and time when the number of sequence elements is too large, which is very uneconomical.

Appendix: Source Code

```
#include <stdio.h>
#include <string.h>
#define max  10000
```

```c
int n, a[max], b[max], c[max], num[max], guess[max];
//n is a given positive number, a[], b[], c[] store the input traversal sequence
in middle, pre-, and post-order respectively
int tree[max][2], root;
//root is used to store the root node, tree[][] stores left and right subtrees
int aa[max],bb[max],cc[max];
//these three arrays are used to store the subscript when a certain value appears
int in[max],pre[max],post[max];
//Complete mid-order, pre-order and post-order sequence
char s[max];
//for input string

int flag,temp;
int height[max][max]={0};
//use a two-dimensional array to store the value of a node in a certain layer of
the tree, which is convenient for output when traversing the layer sequence
int length[max]={0};
//use an array to store the number of nodes at a certain level of the tree

void CreateList(int inleft,int inright,int postleft,int postright,int level){
    int temp=inleft;
    if(inleft>inright)return;//the situation at the end of the function
    if(postleft>postright)return;//the situation at the end of the function
    height[level][length[level]]=post[postright];//find the root node and store
the value
    length[level]++;//the number of nodes in this layer plus one
    while(in[temp]!=post[postright]){
        temp++;//find the position of the root node in the middle-order
traversal sequence
    }
    int len=temp-inleft;//calculate relative position
    CreateList(inleft,temp-1,postleft,postleft+len-1,level+1);//operate on the
left subtree
    CreateList(temp+1,inright,postleft+len,postright-1,level+1);//operate on the
right subtree
}

void PrintBFS(int **height,int length[],int n,int level){
    int i,cnt=0;
    while(cnt<level){//Enter the loop body, traverse each layer
        if(cnt==0){
                printf("%d",height[0][0]);//Output root node
        }
        else{
            for(i=0;i<=length[cnt]-1;i++){//Output the nodes of each layer
                printf(" %d",height[cnt][i]);
            }
        }
        cnt+=1;
    }
}

int Findroot(int *x, int inleft, int inright, int preleft, int preright, int
postleft, int postright) {
    if(inleft<=inright){//Conditions for entering operation
        int i;
        if (b[preleft] && c[postright] && b[preleft] != c[postright]) return 0;
        for (i=inleft;i<=inright;i++){
            if(a[i]){//If the root node is not found, enter the next cycle
                if(b[preleft] && a[i] != b[preleft]) continue;
                if(c[postright] && a[i] != c[postright]) continue;
            }
            /*If the root node is found, assign a value to the array defined by
ourselves*/
            if(a[i]>b[preleft]&&a[i]>c[postright]){
                guess[i]=a[i];
            }
            else{
                if(b[preleft]>c[postright]) guess[i]=b[preleft];
                else guess[i]=c[postright];
            }
            /*Verify that the value found is indeed correct*/
            if (aa[guess[i]] && aa[guess[i]] != i) continue;
            if (bb[guess[i]] && bb[guess[i]] != preleft) continue;
            if (cc[guess[i]] && cc[guess[i]] != postright) continue;
            *x = i;
            /*Check whether the left and right subtrees are reasonable*/
            if (!Findroot(&tree[*x][0], inleft, i - 1, preleft + 1, preleft + i
- inleft, postleft, postleft + i - inleft - 1)) continue;//operate on the left
subtree
            if (!Findroot(&tree[*x][1], i + 1, inright, preleft + 1 + i -
inleft, preright, postleft + i - inleft, postright - 1)) continue;//operate on
the right subtree

            return 1;
        }
        return 0;
        //If no reasonable root node is found at the end of the loop, return 0
    }
    else{
        *x = 0;
        return 1;
    }
}

void restorein(int x) {
    if (!x) return;//Conditions for ending the call
    //Call the function to operate in the order of left, root, and right
    restorein(tree[x][0]);
    in[temp++]=guess[x];
    restorein(tree[x][1]);
}

void restorepre(int x) {
    if (!x) return;
    //Call the function in the order of root, left, and right to operate
    pre[temp++]=guess[x];
    restorepre(tree[x][0]);
    restorepre(tree[x][1]);
}

void restorepost(int x) {
    if (!x) return;
    //Call the function to operate in the order of left, right, and root
    restorepost(tree[x][0]);
    restorepost(tree[x][1]);
    post[temp++]=guess[x];
}

int main()
{
    scanf("%d", &n);
    int number = 0;//Numbers that did not appear
    int res;

    int i,j;

    for (i=1;i<=n;i++){
        scanf("%s",&s);
        a[i]=0;
        if(strcmp(s,"-")==0) a[i]=0;
        //If the input is -, the integer 0 is stored in the array, and the
unknown number is replaced by 0
        else {
            /*Convert the input string into an integer, whether it is a single
digit, a tens digit, a hundred digit, etc.*/
            for(j=0;s[j]!='\0';j++){
                a[i]=a[i]*10+s[j]-'0';
            }
        }
        aa[a[i]]=i;//Storage subscript
        num[a[i]]++;//The number of times a certain number has appeared
    }
    for (i=1;i<=n;i++){
        scanf("%s",&s);
        b[i]=0;
        if(strcmp(s,"-")==0) b[i]=0;
        //If the input is -, the integer 0 is stored in the array, and the
unknown number is replaced by 0
        else {
            /*Convert the input string into an integer, whether it is a single
digit, a tens digit, a hundred digit, etc.*/
            for(j=0;s[j]!='\0';j++){
                b[i]=b[i]*10+s[j]-'0';
            }
        }
        bb[b[i]]=i;//Storage subscript
        num[b[i]]++;//The number of times a certain number has appeared
    }
    for (i=1;i<=n;i++){
        scanf("%s",&s);
        c[i]=0;
        if(strcmp(s,"-")==0) c[i]=0;
        //If the input is -, the integer 0 is stored in the array, and the
unknown number is replaced by 0
        else {
            /*Convert the input string into an integer, whether it is a single
digit, a tens digit, a hundred digit, etc.*/
            for(j=0;s[j]!='\0';j++){
                c[i]=c[i]*10+s[j]-'0';
            }
        }
        cc[c[i]]=i;//Storage subscript
        num[c[i]]++;//The number of times a certain number has appeared
    }

    for (i = 1; i <= n; i++) {if (!num[i]) number++; res = i; }
    //printf("cnt=%d,res=%d\n",cnt,res);

    if (number > 1 || !Findroot(&root, 1, n, 1, n, 1, n))
printf("Impossible\n");
    else {
        //printf("root=%d\n",v[root]);
        for (i = 1; i <= n; i++) if (!guess[i]) guess[i] = res;
            //printf("v[%d]=%d\n",i,guess[i]);

        /*Output the result of in-order traversal*/
        temp=1;
        restorein(root);
        for(i=1;i<=n;i++){
            if(i==1) printf("%d",in[i]);
            else{
                printf(" %d",in[i]);
            }
        }
        printf("\n");
        /*Output the result of preorder traversal*/
        temp=1;
        restorepre(root);
        for(i=1;i<=n;i++){
            if(i==1) printf("%d",pre[i]);
            else{
                printf(" %d",pre[i]);
            }
        }
        printf("\n");
        /*Output the result of post-order traversal*/
        temp=1;
        restorepost(root);
        for(i=1;i<=n;i++){
            if(i==1) printf("%d",post[i]);
            else{
                printf(" %d",post[i]);
            }
        }
        printf("\n");
        /*Call the function to store all the values in the sequence into a two-
dimensional array*/
        CreateList(1,n,1,n,0);//
        int level=0;
        int num=0;
        for(i=0;num<n;i++){//Calculate the total number of layers
            num+=length[i];
            level+=1;
        }
        //printf("level=%d\n",level);
        int *p[max];
        for(i=0;i<level;i++){
            p[i]=height[i];//Store the address of a two-dimensional array
        }
        PrintBFS(p,length,n,level);//Output the result of layer sequence
traversal
```

```
    }
    return 0;
}



//9
//3 - 2 1 7 9 - 4 6
//9 - 5 3 2 1 - 6 4
//3 1 - - 7 - 6 8 -

//10
//2 - 3 7 1 5 8 10 9 -
//- 4 2 3 1 - 6 8 9 -
//2 7 1 3 - - 9 8 6 -

//10
//2 4 6 8 10 - - - - -
//1 - 3 - 5 - 7 - 9 -
//- - - - - - - - - -

//10
//2 4 6 8 10 - - - - -
//- 2 3 - 5 - 7 - 9 -
//- - - - - - - - - -

//9
//3 - 2 1 - 9 - 4 6
//- - 5 - 2 1 - 6 4
//3 1 - - - - 6 8 -
```

Declaration

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