Tree Traversals

Date: October-23th,2021

author:韩恺荣

Chapter One: Introduction

In this project, we try to build a unique tree by partial results of a binary tree's traversals in in-order, pre-order, and post-order.

As we all know, there are four ways usually used by us to describe a tree, which is in-order, pre-order and post-order. And based on the question we do in PTA, we can bulid a unique tree by in-order results and one of the order results chosen from pre-order, post-order and level-order. So in this question, we expand the result and deepen it. Can we make a unique tree by partial results?For example, assume a tree with nine nodes, and there are three sequences made from it.

In-order sequence: 3 - 2 1 7 9 - 4 6

pre-order sequence: 9 - 5 3 2 1 - 6 4

post-order sequence:3 1 - - 7 - 6 8 -

And ‘-’ represent unknown elements in the sequence. Can we make up a unique tree from three sequences? We will solve this problem in this project.

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 and no numbers is repetitive.

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 will 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 Impossible. For example,in the case i give above, we should print as follow:

```
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
```

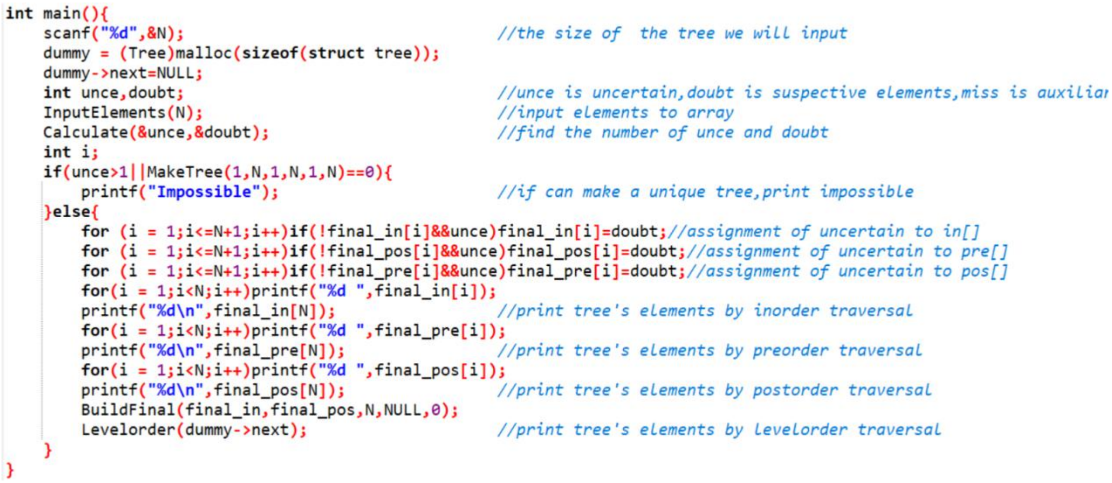
But if we input:

```
- - -
- 1 -
1 - -
```

Then we will print “Impossible” in the screen.

Chapter 2: Algorithm Specification

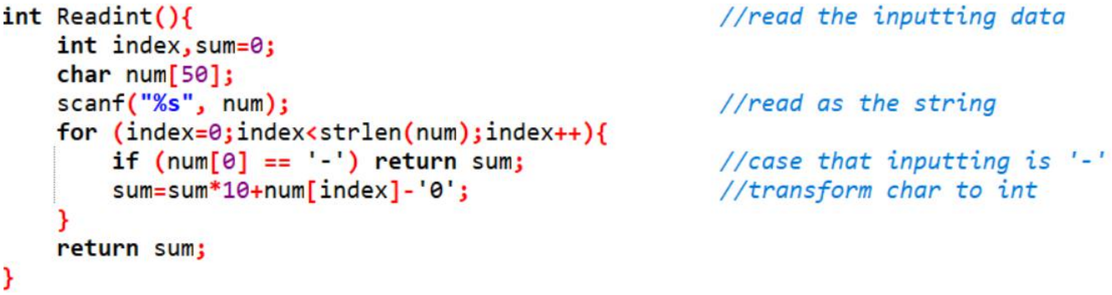
function 1: main function



```c
int main(){
    scanf("%d",&N);                             //the size of  the tree we will input
    dummy = (Tree)malloc(sizeof(struct tree));
    dummy->next=NULL;
    int unce,doubt;                             //unce is uncertain,doubt is suspective elements,miss is auxiliar
    InputElements(N);                           //input elements to array
    Calculate(&unce,&doubt);                    //find the number of unce and doubt
    int i;
    if(unce>1||MakeTree(1,N,1,N,1,N)==0){
        printf("Impossible");                   //if can make a unique tree,print impossible
    }else{
        for (i = 1;i<=N+1;i++)if(!final_in[i]&&unce)final_in[i]=doubt;//assignment of uncertain to in[]
        for (i = 1;i<=N+1;i++)if(!final_pos[i]&&unce)final_pos[i]=doubt;//assignment of uncertain to pre[]
        for (i = 1;i<=N+1;i++)if(!final_pre[i]&&unce)final_pre[i]=doubt;//assignment of uncertain to pos[]
        for(i = 1;i<N;i++)printf("%d ",final_in[i]);
        printf("%d\n",final_in[N]);             //print tree's elements by inorder traversal
        for(i = 1;i<N;i++)printf("%d ",final_pre[i]);
        printf("%d\n",final_pre[N]);            //print tree's elements by preorder traversal
        for(i = 1;i<N;i++)printf("%d ",final_pos[i]);
        printf("%d\n",final_pos[N]);            //print tree's elements by postorder traversal
        BuildFinal(final_in,final_pos,N,NULL,0);
        Levelorder(dummy->next);                //print tree's elements by Levelorder traversal
    }
}
```

By using module design ways, I break question into pieces. I deal with the input , judge and output in turn.

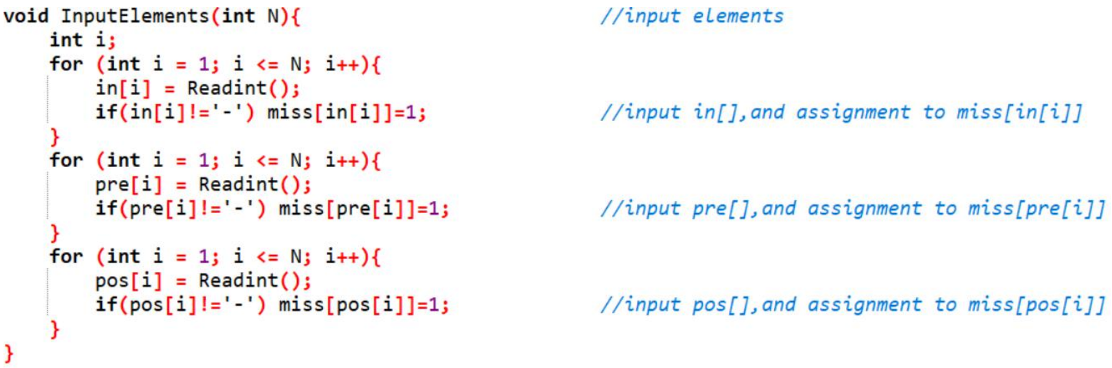
function 2: Readint



```c
int Readint(){                                  //read the inputting data
    int index,sum=0;
    char num[50];
    scanf("%s", num);                           //read as the string
    for (index=0;index<strlen(num);index++){
        if (num[0] == '-') return sum;          //case that inputting is '-'
        sum=sum*10+num[index]-'0';              //transform char to int
    }
    return sum;
}
```

This function is to read the char and transform it to int

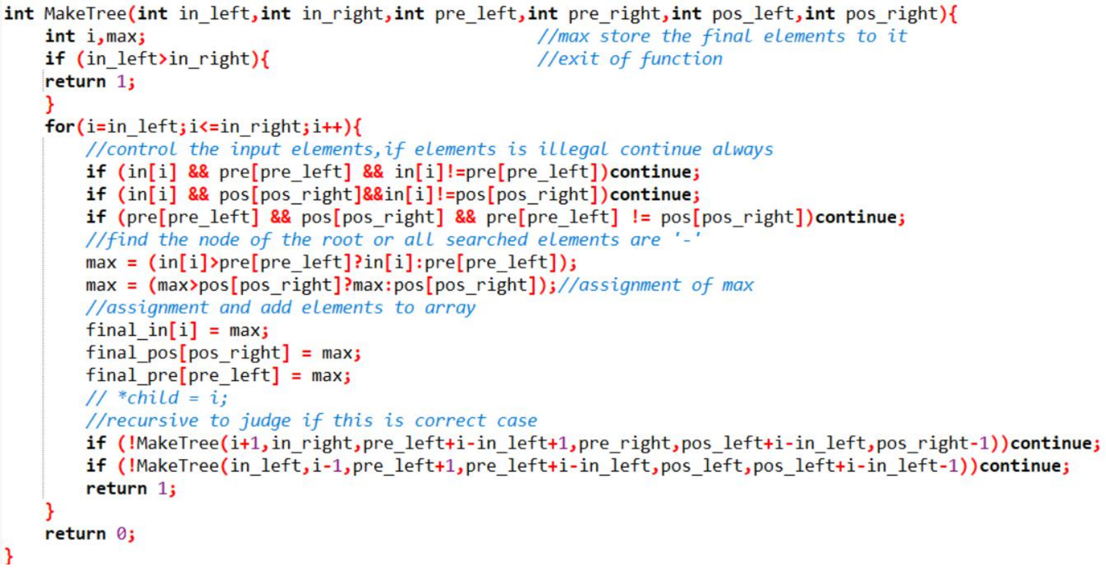
function3: InputElements



```c
void InputElements(int N){                      //input elements
    int i;
    for (int i = 1; i <= N; i++){
        in[i] = Readint();
        if(in[i]!='-') miss[in[i]]=1;           //input in[],and assignment to miss[in[i]]
    }
    for (int i = 1; i <= N; i++){
        pre[i] = Readint();
        if(pre[i]!='-') miss[pre[i]]=1;         //input pre[],and assignment to miss[pre[i]]
    }
    for (int i = 1; i <= N; i++){
        pos[i] = Readint();
        if(pos[i]!='-') miss[pos[i]]=1;         //input pos[],and assignment to miss[pos[i]]
    }
}
```

Use with function two,which is to make input as a certain sequence.

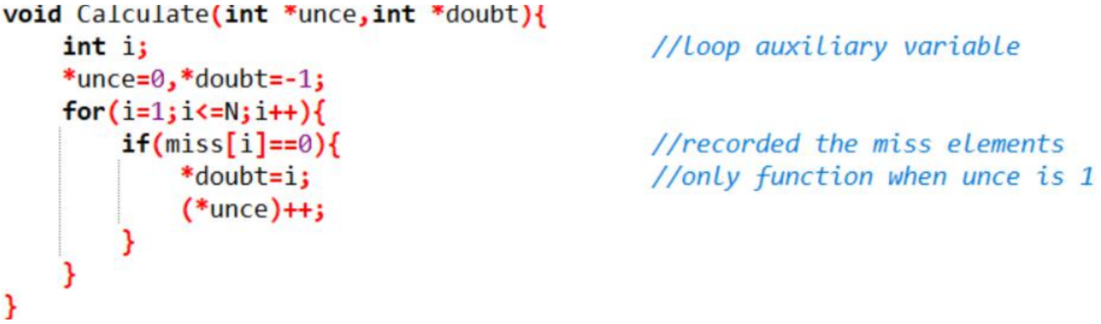
function 4:Make tree



```
int MakeTree(int in_left,int in_right,int pre_left,int pre_right,int pos_left,int pos_right){
    int i,max;                                  //max store the final elements to it
    if (in_left>in_right){                      //exit of function
    return 1;
    }
    for(i=in_left;i<=in_right;i++){
        //control the input elements,if elements is illegal continue always
        if (in[i] && pre[pre_left] && in[i]!=pre[pre_left])continue;
        if (in[i] && pos[pos_right]&&in[i]!=pos[pos_right])continue;
        if (pre[pre_left] && pos[pos_right] && pre[pre_left] != pos[pos_right])continue;
        //find the node of the root or all searched elements are '-'
        max = (in[i]>pre[pre_left]?in[i]:pre[pre_left]);
        max = (max>pos[pos_right]?max:pos[pos_right]);//assignment of max
        //assignment and add elements to array
        final_in[i] = max;
        final_pos[pos_right] = max;
        final_pre[pre_left] = max;
        // *child = i;
        //recursive to judge if this is correct case
        if (!MakeTree(i+1,in_right,pre_left+i-in_left+1,pre_right,pos_left+i-in_left,pos_right-1))continue;
        if (!MakeTree(in_left,i-1,pre_left+1,pre_left+i-in_left,pos_left,pos_left+i-in_left-1))continue;
        return 1;
    }
    return 0;
}
```

This function is to judge whether the array can reconstruct a unique tree, and make up three arrays by the way.

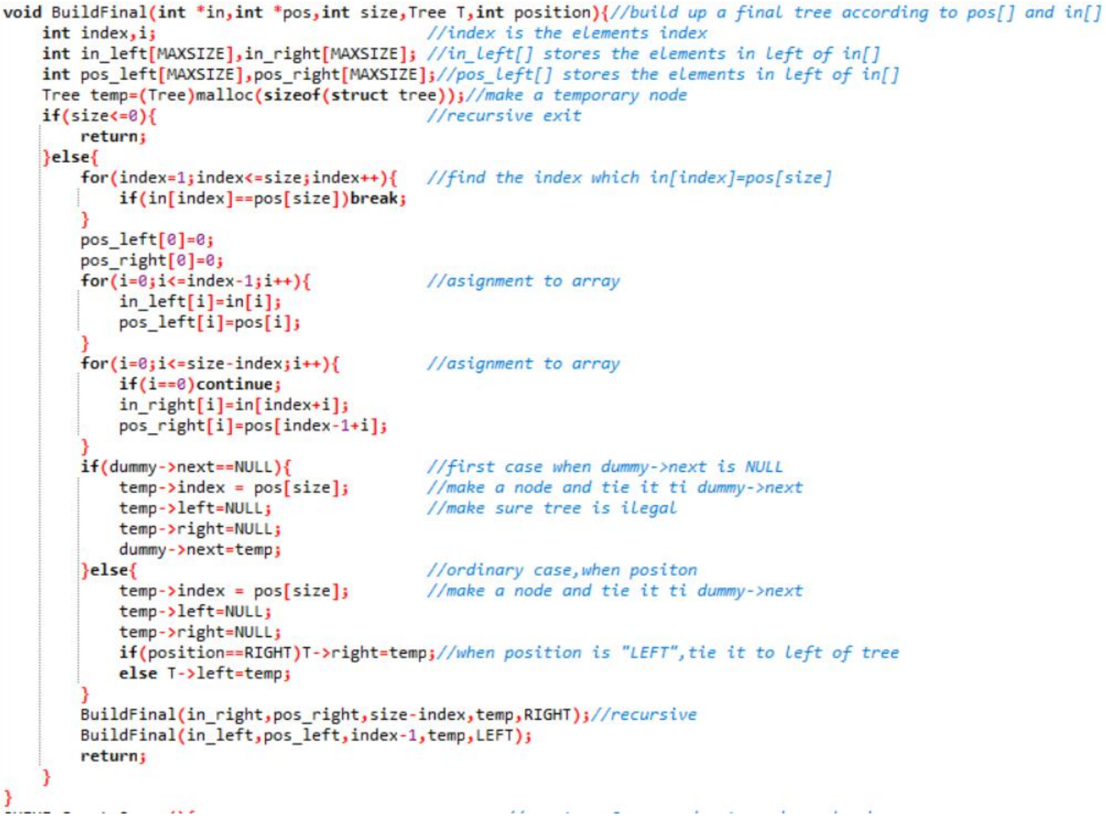
function 5: Calculate



```
void Calculate(int *unce,int *doubt){
    int i;                                  //loop auxiliary variable
    *unce=0,*doubt=-1;
    for(i=1;i<=N;i++){
        if(miss[i]==0){                     //recorded the miss elements
            *doubt=i;                       //only function when unce is 1
            (*unce)++;
        }
    }
}
```

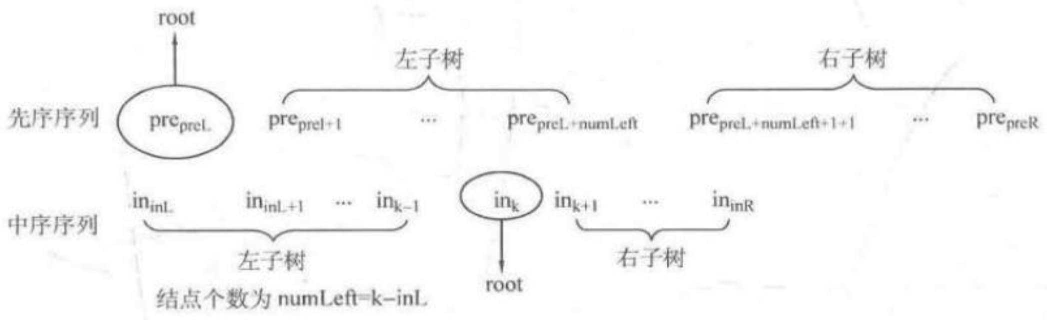
Using this function is to verify the value of “unce” and “doubt”, and unce is degree of freedom of the arrays we input.

function 6: BuildFinal

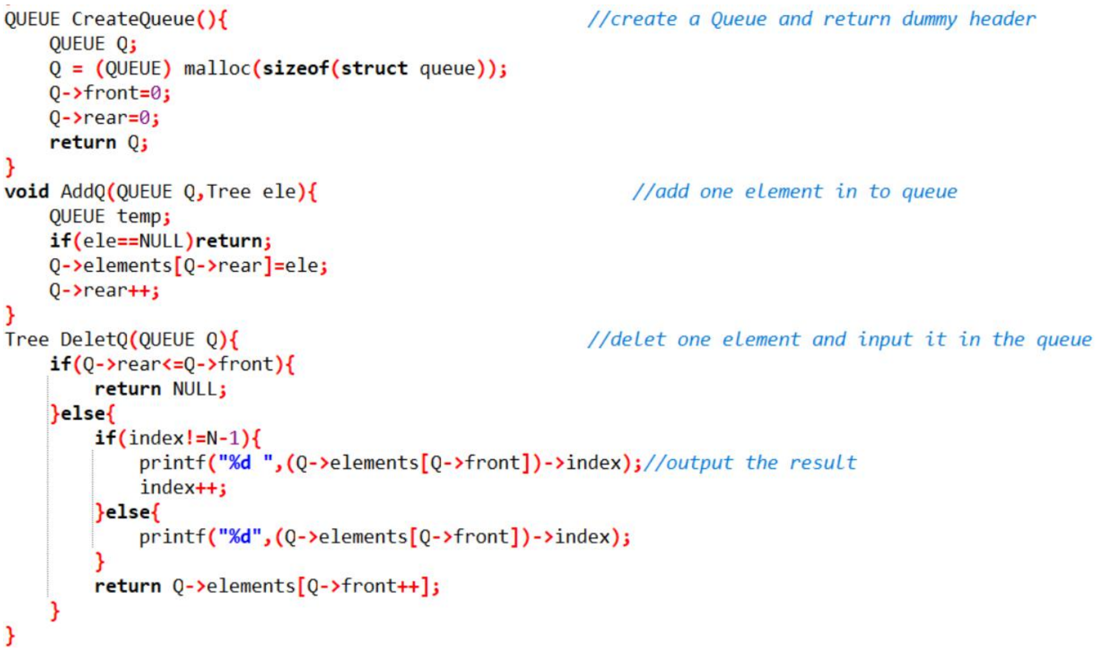


```
void BuildFinal(int *in,int *pos,int size,Tree T,int position){//build up a final tree according to pos[] and in[]
    int index,i;                            //index is the elements index
    int in_left[MAXSIZE],in_right[MAXSIZE]; //in_left[] stores the elements in left of in[]
    int pos_left[MAXSIZE],pos_right[MAXSIZE];//pos_left[] stores the elements in left of in[]
    Tree temp=(Tree)malloc(sizeof(struct tree));//make a temporary node
    if(size<=0){                            //recursive exit
        return;
    }else{
        for(index=1;index<=size;index++){   //find the index which in[index]=pos[size]
            if(in[index]==pos[size])break;
        }
        pos_left[0]=0;
        pos_right[0]=0;
        for(i=0;i<=index-1;i++){            //asignment to array
            in_left[i]=in[i];
            pos_left[i]=pos[i];
        }
        for(i=0;i<=size-index;i++){         //asignment to array
            if(i==0)continue;
            in_right[i]=in[index+i];
            pos_right[i]=pos[index-1+i];
        }
        if(dummy->next==NULL){              //first case when dummy->next is NULL
            temp->index = pos[size];        //make a node and tie it ti dummy->next
            temp->left=NULL;                //make sure tree is ilegal
            temp->right=NULL;
            dummy->next=temp;
        }else{                              //ordinary case,when positon
            temp->index = pos[size];        //make a node and tie it ti dummy->next
            temp->left=NULL;
            temp->right=NULL;
            if(position==RIGHT)T->right=temp;//when position is "LEFT",tie it to left of tree
            else T->left=temp;
        }
        BuildFinal(in_right,pos_right,size-index,temp,RIGHT);//recursive
        BuildFinal(in_left,pos_left,index-1,temp,LEFT);
        return;
    }
}
```

build the final tree according the array we generate in function “MakeTree”, and the way to build is use recursion like this:



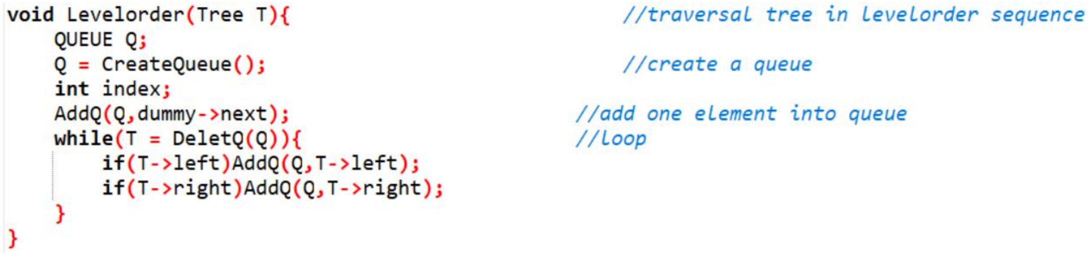
function 7/8/9:function about operations on queue



```
QUEUE CreateQueue(){                                 //create a Queue and return dummy header
    QUEUE Q;
    Q = (QUEUE) malloc(sizeof(struct queue));
    Q->front=0;
    Q->rear=0;
    return Q;
}
void AddQ(QUEUE Q,Tree ele){                              //add one element in to queue
    QUEUE temp;
    if(ele==NULL)return;
    Q->elements[Q->rear]=ele;
    Q->rear++;
}
Tree DeletQ(QUEUE Q){                                //delet one element and input it in the queue
    if(Q->rear<=Q->front){
        return NULL;
    }else{
        if(index!=N-1){
            printf("%d ",(Q->elements[Q->front])->index);//output the result
            index++;
        }else{
            printf("%d",(Q->elements[Q->front])->index);
        }
        return Q->elements[Q->front++];
    }
}
```

’ This part is use to make some operations on queue, such as Create, Add and Delete.

function 10: Levelorder



```
void Levelorder(Tree T){                              //traversal tree in levelorder sequence
    QUEUE Q;
    Q = CreateQueue();                                //create a queue
    int index;
    AddQ(Q,dummy->next);                          //add one element into queue
    while(T = DeletQ(Q)){                         //loop
        if(T->left)AddQ(Q,T->left);
        if(T->right)AddQ(Q,T->right);
    }
}
```

traversal tree in levelorder sequence.

summary: In my code, there are ten functions.And My train of thought is that we could define a tree to store all elements and it is the final tree to output.And that tree’ s dummy header is named by “dummy” .Three order we input stored by in[], pre[] and pos[] as integer variable. First, we execute the function of reading data. “in” is inorder sequence, “pre” is preorder sequence, “pos” is postorder sequence.

After a few operations on the inputting things, we have to call MakeTree function to judge whether the inputting array can reconstruct to a tree . In the function, the main ways is to use recursion to split up the array until it have only one element. And the returning value is up to the every call to the recursion function. Finally, we can get a final_in array and final_pos array, which stores the made-up elements. We can reconstruct a unique tree by two final arrays.

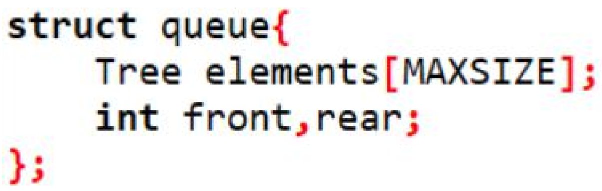
Data Structure:

(1) Tree:

it is implemented by linked-list. And every element consists of a integer type element, and two pointer pointed to left subtree and right subtree. And “next” is only used to dummy header.

(2) Queue:

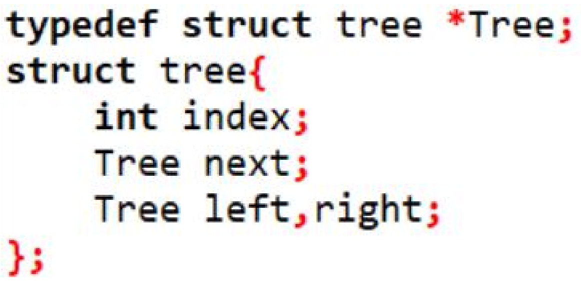
it is defined by array, and two function: AddQ and DeletQ in the code are used to add a element and delete a element in queue.



```
struct queue{
    Tree elements[MAXSIZE];
    int front,rear;
};
```

(3) linked-list:

I define a linked-list to implement Tree structure.



```
typedef struct tree *Tree;
struct tree{
    int index;
    Tree next;
    Tree left,right;
};
```

Chapter 3: Testing Results

There are four main possibilities in the project:

(1) The elements we input can not make a tree.

For example:

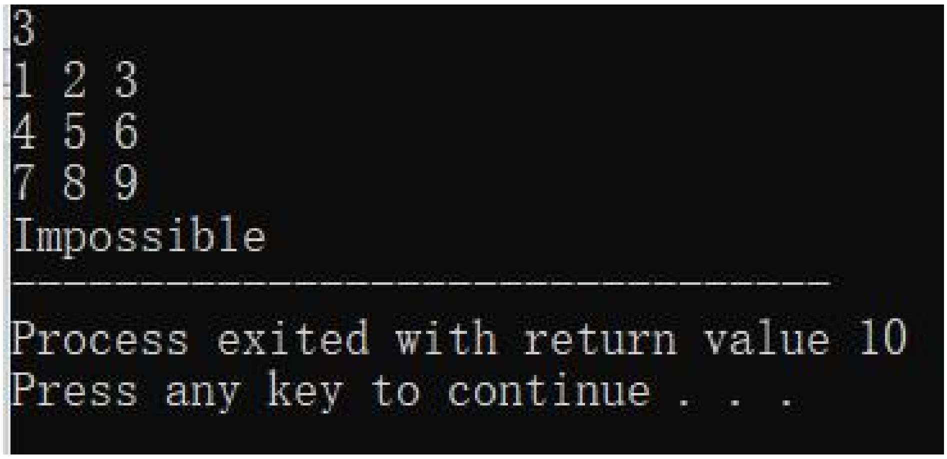
Input: 3

1 2 3

4 5 6

7 8 9

These sequence can not make a tree apparently because of the unique number of differences for input is 9,which is bigger than 3.



```
3
1 2 3
4 5 6
7 8 9
Impossible
--------------------------------
Process exited with return value 10
Press any key to continue . . .
```

(2) The elements we input can not make a unique tree:

For example:

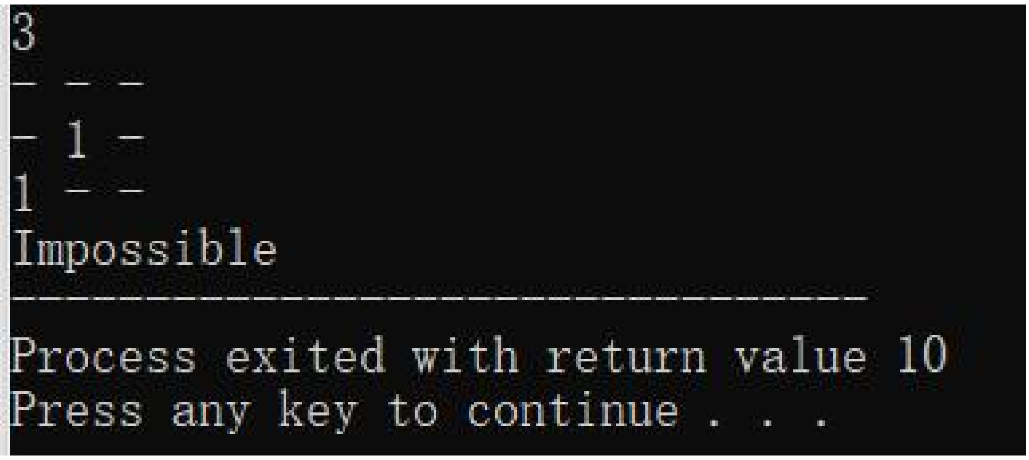
input: 3

- - -

- 1 -

1 - -

result:



```
3
- - -
- 1 -
1 - -
Impossible
--------------------------------
Process exited with return value 10
Press any key to continue . . .
```

(3) The elements we input can make a unique tree:

For example:

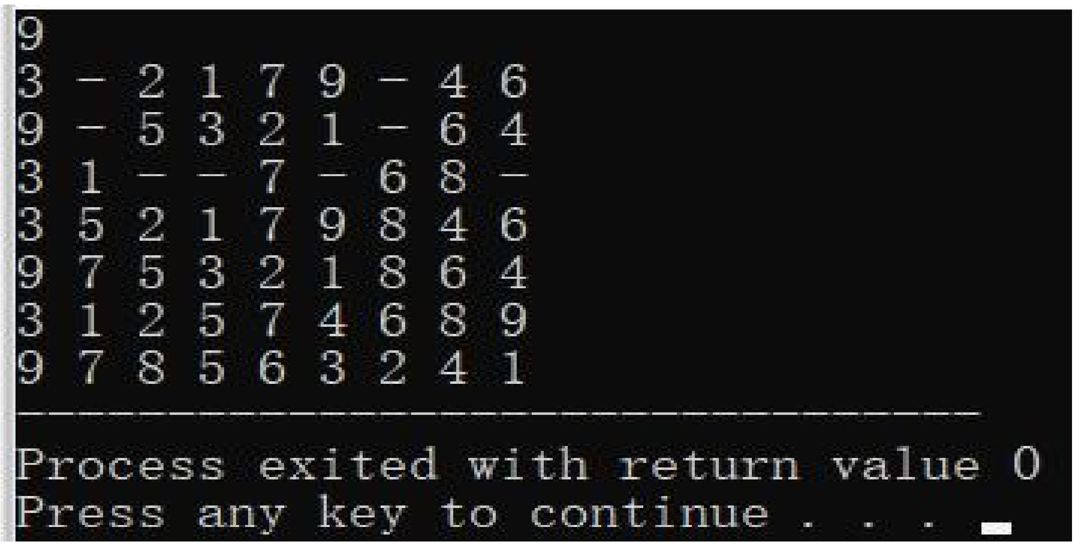
Input: 9

3 - 2 1 7 9 - 4 6

9 - 5 3 2 1 - 6 4

3 1 - - 7 - 6 8 -

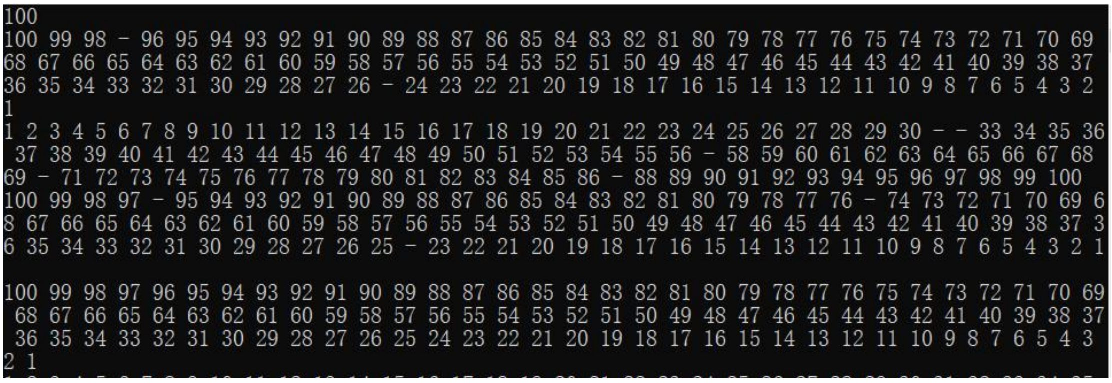
result:



```
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
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

(4) The number of elements is very large:

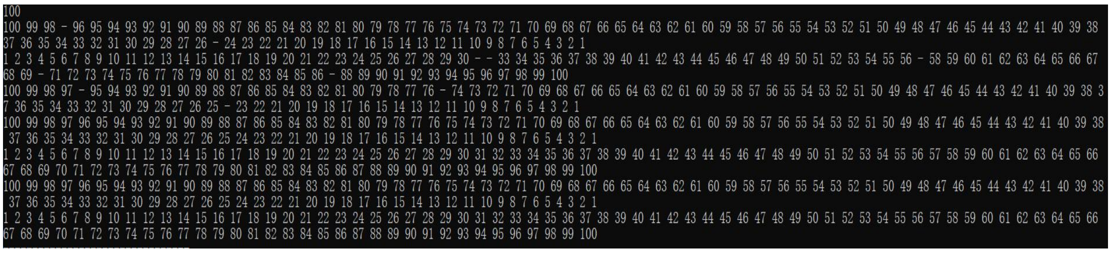
we input 100 numbers:



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

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
```

result:



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

Chapter 4: Analysis and Comments

(1) Analyse and comments for the algorithm:

a) Time Complexity:

The Time Complexity depends on MakeTree function and buildFinal function.And worst case in the programmer is when tree we will finally get is like a linked-list, which means all node has either right subtree or left subtree.In this case, we should execute n times MakeTree function. In every call of the function, the worst case is that we should loop n times to find the element in in[], which is same as it in pos[size], thus T1(n) = 2*T1(n-1) +O(n). So the time complexity of function MakeTree is O(n^2). For the function buildFinal, the worst case happen similarly. And we have T2(n) = O(n^2). Finally, for the function levelorder, the time complexity of it is T3(n) = O(n). Therefore, the T(n) = MAX{T1,T2,T3},which is O(n^2).

b) Space Complexity:

The space complexity of the whole process is determined by MakeTree function and buildFinal function. Because it is a recursive function, so when all node has either right subtree or left subtree, it’s space complexity has worst case. it is n because every call of the function should occupy constant space in computer. So S(n) = O(n).

c)Comments:

For this algorithm, I don’t have any good ways. And time complexity along with space is big. So we should think some ways to optimize it.

Appendix: Source Code(in C)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define LEFT -1                                          //define the
position to make a linked-List
#define RIGHT -2                                      //define the
position to make a linked-List
#define MAXSIZE 300                                        //define
most range of inputting number
int N,index;                                         //index is used
to create "add" array,index is "index" of array's elements
typedef struct tree *Tree;                               //define a
tree
struct tree{                                  //implementation
of the tree
    int index;                                       //store
elements' index
    Tree next;
    Tree left,right;
};
struct queue{
    //implementation of the queue
    Tree elements[MAXSIZE];                        //elements of
the queue is a tree
    int front,rear;
};
int first;
Tree dummy;                                        //dummy
head defined only one
typedef struct queue *QUEUE;
void Levelorder(Tree T);                           //print tree's
elements by levelorder traversal
Tree DeletQ(QUEUE Q) ;                             //delet and
output one element in the queue
void AddQ(QUEUE Q,Tree ele);                       //add
elements in to the array
void BuildTree(int *in_left,int *pos_left,int size,Tree ptr,int
position);//build a tree to print elements
int MakeTree(int in_left,int in_right,int pre_left,int
pre_right,int pos_left,int pos_right);//judge and make a
final array
void Calculate(int *unce,int *doubt);              //calculate
the number
```

```
void InputElements(int N);                    //input
elements to array
int ReadInt();                                //read the
inputting data
void BuildFinal(int *in,int *pos,int size,Tree T,int position);
int
in[MAXSIZE],pos[MAXSIZE],pre[MAXSIZE],miss[MAXSIZE],
final_in[MAXSIZE],final_pos[MAXSIZE],final_pre[MAXSIZE];
//store final consequence
int main(){
    scanf("%d",&N);                                    //the
size of   the tree we will input
    dummy = (Tree)malloc(sizeof(struct tree));
    dummy->next=NULL;
    int unce,doubt;                           //unce is
uncertain,doubt is suspective elements,miss is auxiliary
array
    InputElements(N);                         //input
elements to array
    Calculate(&unce,&doubt);                  //find the
number of unce and doubt
    int i;
    if(unce>1||MakeTree(1,N,1,N,1,N)==0){
        printf("Impossible");                //if can make a unique tree,print impossible
    }else{
        for (i = 1;i<=N+1;i++)if(!final_in[i]&&unce)final_in[i]=doubt;//assignment of uncertain to in[]
        for (i = 1;i<=N+1;i++)if(!final_pos[i]&&unce)final_pos[i]=doubt;//assignment of uncertain to pre[]
        for (i = 1;i<=N+1;i++)if(!final_pre[i]&&unce)final_pre[i]=doubt;//assignment of uncertain to pos[]
        for(i = 1;i<N;i++)printf("%d ",final_in[i]);
        printf("%d\n",final_in[N]);            //print tree's elements by inorder traversal
        for(i = 1;i<N;i++)printf("%d ",final_pre[i]);
        printf("%d\n",final_pre[N]);           //print tree's elements by preorder traversal
        for(i = 1;i<N;i++)printf("%d ",final_pos[i]);
        printf("%d\n",final_pos[N]);           //print tree's elements by postorder traversal
        BuildFinal(final_in,final_pos,N,NULL,0);
        Levelorder(dummy->next);                    //print
tree's elements by levelorder traversal
    }
}
int Readint(){                                    //read the
inputting data
      int index,sum=0;
    char num[50];
      scanf("%s", num);                           //read as the
string
      for (index=0;index<strlen(num);index++){
        if (num[0] == '-') return sum;              //case that
inputting is '-'
        sum=sum*10+num[index]-'0';
    //transform char to int
    }
      return sum;
}
void InputElements(int N){                        //input
elements
    int i;
    for (int i = 1; i <= N; i++){
          in[i] = Readint();
          if(in[i]!='-') miss[in[i]]=1;               //input
in[],and assignment to miss[in[i]]
     }
     for (int i = 1; i <= N; i++){
          pre[i] = Readint();
          if(pre[i]!='-') miss[pre[i]]=1;             //input
pre[],and assignment to miss[pre[i]]
     }
     for (int i = 1; i <= N; i++){
          pos[i] = Readint();
          if(pos[i]!='-') miss[pos[i]]=1;             //input
pos[],and assignment to miss[pos[i]]
     }
}

int MakeTree(int in_left,int in_right,int pre_left,int
pre_right,int pos_left,int pos_right){
    int i,max;                                      //max store the
final elements to it
    if (in_left>in_right){                          //exit of function
    return 1;
    }
      for(i=in_left;i<=in_right;i++){
        //control the input elements,if elements is illegal
continue always
        if (in[i] && pre[pre_left] &&
in[i]!=pre[pre_left])continue;
        if (in[i] &&
pos[pos_right]&&in[i]!=pos[pos_right])continue;
            if (pre[pre_left] && pos[pos_right] &&
pre[pre_left] != pos[pos_right])continue;
            //find the node of the root or all searched
elements are '-'
        max = (in[i]>pre[pre_left]?in[i]:pre[pre_left]);
        max =
(max>pos[pos_right]?max:pos[pos_right]);//assignment of
max
            //assignment and add elements to array
        final_in[i] = max;
            final_pos[pos_right] = max;
            final_pre[pre_left] = max;
        // *child = i;
```

```
           //recursive to judge if this is correct case
        if
(!MakeTree(i+1,in_right,pre_left+i-in_left+1,pre_right,pos_
left+i-in_left,pos_right-1))continue;
           if
(!MakeTree(in_left,i-1,pre_left+1,pre_left+i-in_left,pos_left,
pos_left+i-in_left-1))continue;
        return 1;
      }
      return 0;
}
void Calculate(int *unce,int *doubt){
    int i;                                  //loop auxiliary variable
    *unce=0,*doubt=-1;
    for(i=1;i<=N;i++){
        if(miss[i]==0){                     //recorded the miss
elements
            *doubt=i;                       //only function when
unce is 1
            (*unce)++;
        }
    }
```

```
}
void BuildFinal(int *in,int *pos,int size,Tree T,int
position){//build up a final tree according to pos[] and in[]
    int index,i;                        //index is the elements
index
    int in_left[MAXSIZE],in_right[MAXSIZE]; //in_left[]
stores the elements in left of in[]
    int pos_left[MAXSIZE],pos_right[MAXSIZE];//pos_left[]
stores the elements in left of in[]
    Tree temp=(Tree)malloc(sizeof(struct tree));//make a
temporary node
    if(size<=0){                        //recursive exit
        return;
    }else{
        for(index=1;index<=size;index++){//find the index
which in[index]=pos[size]
            if(in[index]==pos[size])break;
        }
        pos_left[0]=0;
        pos_right[0]=0;
        for(i=0;i<=index-1;i++){           //asignment to
array
```

```
            in_left[i]=in[i];
            pos_left[i]=pos[i];
        }
        for(i=0;i<=size-index;i++){            //asignment to
array
            if(i==0)continue;
            in_right[i]=in[index+i];
            pos_right[i]=pos[index-1+i];
        }
        if(dummy->next==NULL){                //first case when
dummy->next is NULL
            temp->index = pos[size];      //make a node and
tie it ti dummy->next
            temp->left=NULL;              //make sure tree is
ilegal
            temp->right=NULL;
            dummy->next=temp;
        }else{                            //ordinary case,when
positon
            temp->index = pos[size];      //make a node and
tie it ti dummy->next
            temp->left=NULL;
            temp->right=NULL;
            if(position==RIGHT)T->right=temp;//when
position is "LEFT",tie it to left of tree
            else T->left=temp;
        }

    BuildFinal(in_right,pos_right,size-index,temp,RIGHT);//r
ecursive
        BuildFinal(in_left,pos_left,index-1,temp,LEFT);
        return;
    }
}
QUEUE CreateQueue(){                            //create a
Queue and return dummy header
    QUEUE Q;
    Q = (QUEUE) malloc(sizeof(struct queue));
    Q->front=0;
    Q->rear=0;
    return Q;
}
void AddQ(QUEUE Q,Tree ele){                          //add
one element in to queue
```

```
    QUEUE temp;
    if(ele==NULL)return;
    Q->elements[Q->rear]=ele;
    Q->rear++;
}
Tree DeletQ(QUEUE Q){                              //delet one
element and input it in the queue
    if(Q->rear<=Q->front){
        return NULL;
    }else{
        if(index!=N-1){
            printf("%d
",(Q->elements[Q->front])->index);//output the result
            index++;
        }else{
            printf("%d",(Q->elements[Q->front])->index);
        }
        return Q->elements[Q->front++];
    }
}
void Levelorder(Tree T){                           //traversal
tree in levelorder sequence
```

```
    QUEUE Q;
    Q = CreateQueue();                              //create a
queue
    int index;
    AddQ(Q,dummy->next);                            //add one
element into queue
    while(T = DeletQ(Q)){                       //loop
        if(T->left)AddQ(Q,T->left);
        if(T->right)AddQ(Q,T->right);
    }
}
```

Declaration

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