Binary Search Tree

魏张鉴 刘理铖 施能

2015-10-28

Chapter 1: Introduction

A binary search tree is uniquely determined by a given ordered insertions of a sequence of positive integers. On the other hand, a given binary search tree may correspond to several different insertion sequences. Now given several insertion sequences, it is your job to determine if they can generate the same binary search tree.

Input Specification:

Input consists of several test cases. For each test case, the first line contains two positive integers N (10) and L, which are the total number of integers in an insertion sequence and the number of sequences to be tested, respectively. The second line contains N positive integers, separated by a space, which are the initially inserted numbers. Then L lines follow, each contains a sequence of N integers to be checked.

For convenience, it is guaranteed that each insertion sequence is a permutation of integers 1 to N – that is, all the N numbers are distinct and no greater than N. The input ends with N being 0. That case must NOT be processed.

Output Specification:

For each test case, output to the standard output. Print in L lines the checking results: if the sequence given in the i-th line corresponds to the same binary search tree as the initial sequence, output to the i-th line “Yes”; else output “No”.

Sample Input:

4 2

3 1 4 2

3 4 1 2

3 2 4 1

2 1

2 1

1 2

0

Sample Output:

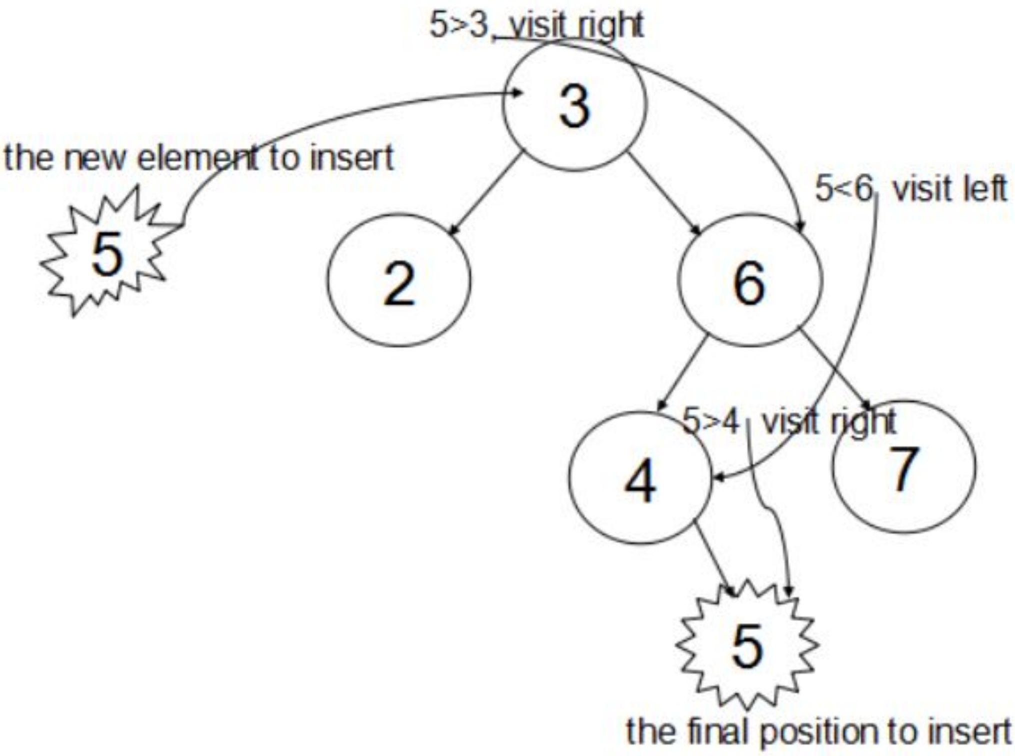
Yes

No

No

Chapter 2: Algorithm Specification

How to insert? Using recursion, firstly visit the root of the whole tree as a starting-point, if the value of the new element being inserted is less than the value of the node being visited, we will visit the left child of the node, otherwise(it is guaranteed that all the N numbers are distinct) we will visit the right child. Util the node we visit is a NULL node, we can put the new element into this spare space.



(the schematic diagram of insert algorithm)

How to judge whether 2 trees are equal? We can adopt recursive algorithm too. If left child tree, right child tree and element of one tree are all equal to those of another tree, they should be equal.

```
bool isequal(T1,T2){
        if both T1 and T2 are NULL
                return true
        if one is NULL while the other isn't
                return false
        //after that both 2 trees must be NOT NULL
        if(isequal(T1.left,T2.left)&&isequal(T1.right,T2.right)&&T1.element==T2.element)
                return true;
}
```

(pseudocode of the function isequal())

Chapter 3: Testing Results(Current Status: pass)

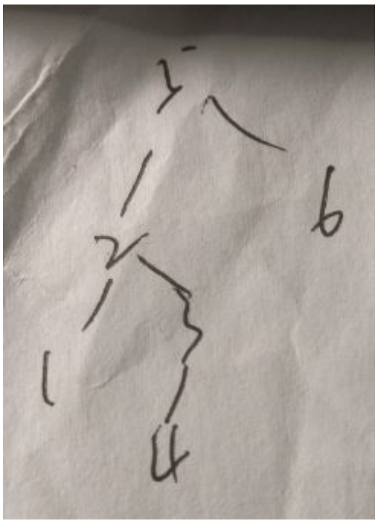
As a tester, I use a set of test cases to test the program. The data is in “p2.in” and the result is in “p2.out”.

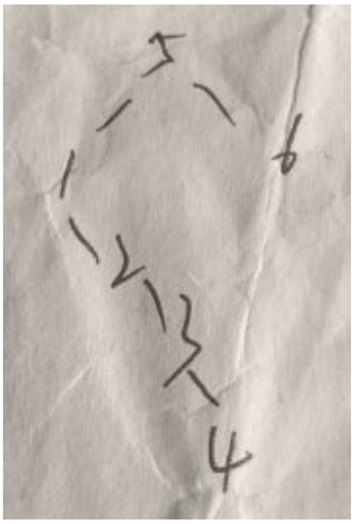
Now, we can analyse the input and the output:

The first test case :

```
6 4
5 2 1 3 4 6
5 6 2 1 3 4
5 2 3 1 4 6
5 2 1 3 6 4
5 1 2 3 4 6
```

As we can see, the first 4 lines generate the same tree, while the fifth line generates a different tree.





So the answer is

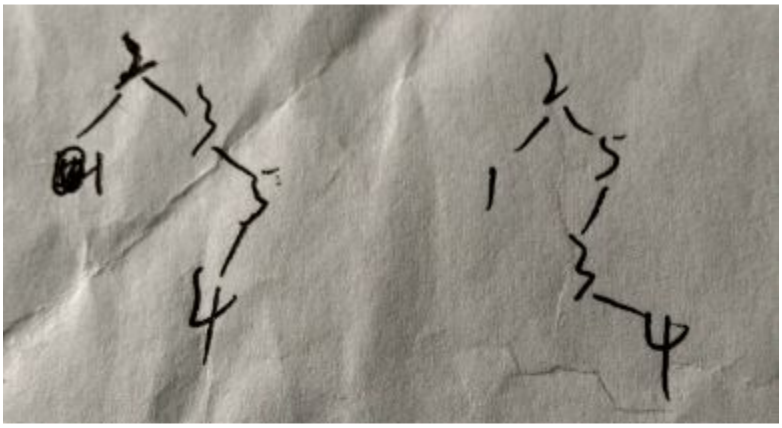
```
Yes
Yes
Yes
No
```

.

The second test case :

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

The first 2 lines generate the same tree, while the third line generates a different tree.



So the answer is

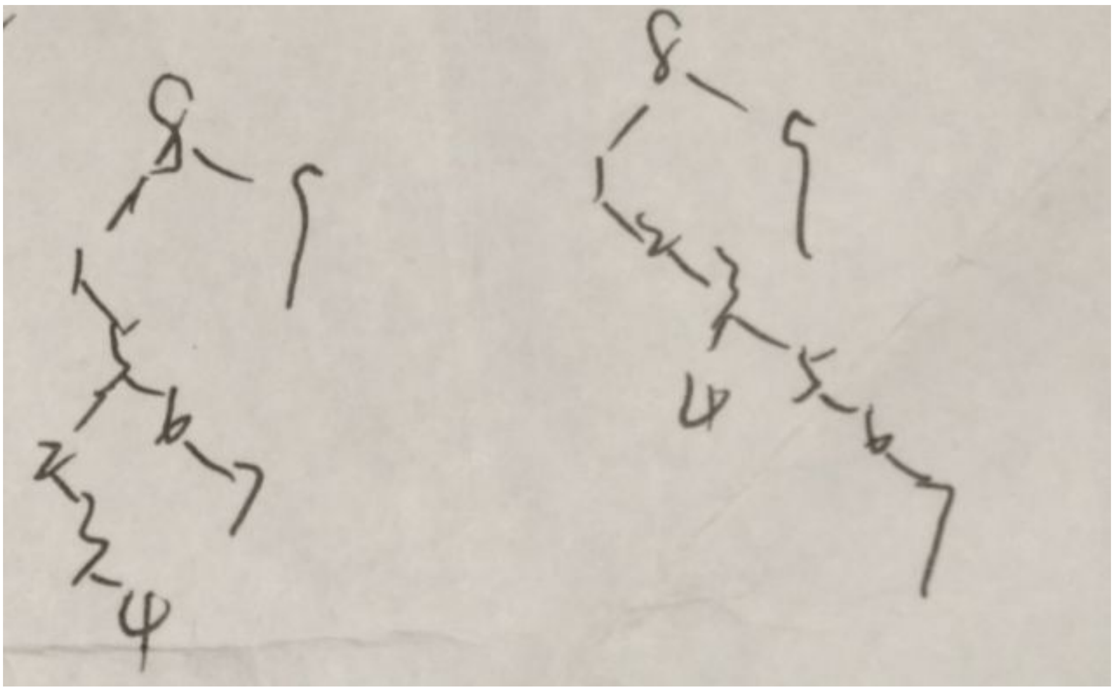
```
Yes
No
```

.

The third test case :

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

The first 4 lines generate the same tree, while the fifth line generates a different tree.



So the answer is

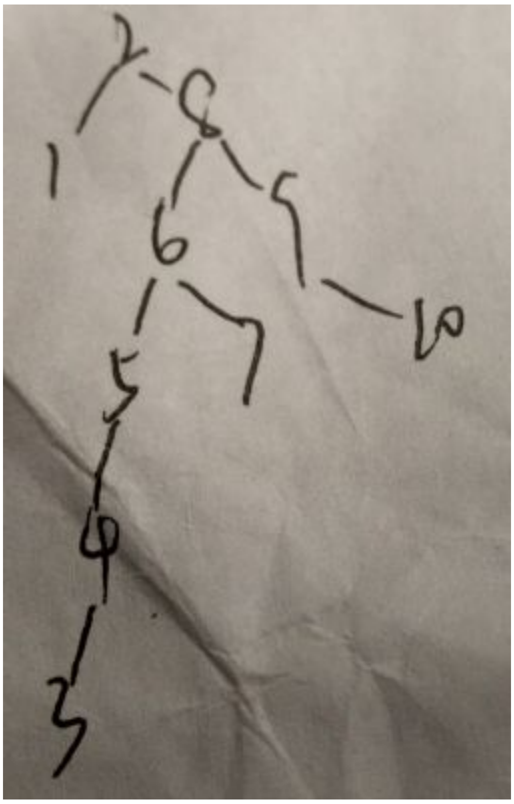
```
Yes
Yes
Yes
No
```

.

The fourth test case :

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

The 3 lines generate the same tree.



So the answer is

```
Yes
Yes
```

.

The fifth test case :

```
3 2
2 3 1
2 1 3
1 2 3
```

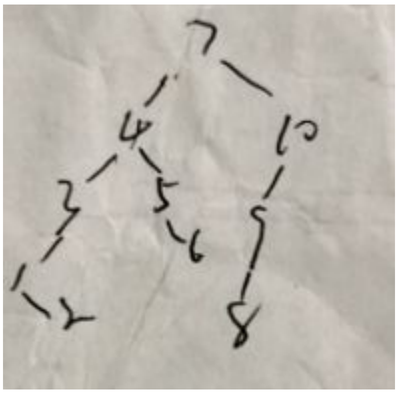
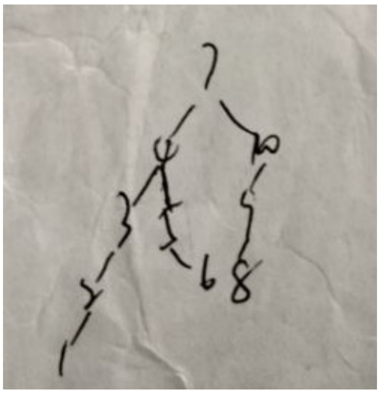
Obviously, the first 2 lines generate the same tree, while the third line generates a different tree. So the answer is

```
Yes
No
```

The sixth test case :

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

The first 3 lines generate the same tree, while the fourth line generates a different tree.

So the answer is

```
Yes
Yes
No
```

.

The seventh test case :

```
1 1
1
1
```

Obviously, the answer is

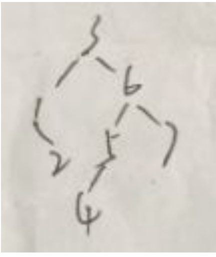
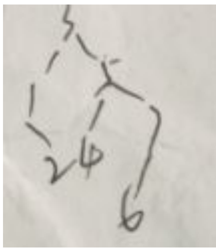
```
Yes
```

.

The eighth test case :

```
7 3
3 6 5 1 4 7 2
3 6 7 5 4 1 2
3 1 2 6 7 5 4
3 1 5 4 7 6 2
```

The first 3 lines generate the same tree, while the fourth line generates a different tree.

So the answer is

```
Yes
Yes
No
```

.

Chapter 4: Analysis and Comments

The time complexity of the function Insert is O(NlogN). In constant time we descend a level in the tree, thus operating on a tree is now roughly half as large. Indeed, one insert operation is O(d), where d is the depth of the node containing in the accessed key. We can prove that the average depth over all nodes in a is O(logN) on the assumption that all trees are equally likely.

The time complexity of the function Isequal is O(N) because all the node is supposed to be compared once in the worst case.

So the time complexity of the whole program is O(L*N*(logN+1))=O(L*NlogN) for each test case.

The space complexity of the program is O(L*N) for each test case as one node need constant space to store.

As the scale of the data is small, so I think the program performs well enough. My only comment is that we can free the space after the operation of one tree so the space complexity of the program can reduce to O(N).

Appendix: Source Code (in C)

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

typedef int ElementType;

typedef struct TreeNode *Tree;
struct TreeNode {
    ElementType Element;
    Tree  Left;
    Tree  Right;
};/*Tree is defined as this */

Tree  Insert( ElementType X, Tree T );
int isequal( Tree T1, Tree T2 );
```

```
int main()
{
    int i,j,N,L;
    ElementType x;
    Tree T0=NULL,T;

    freopen("p2.in","r",stdin);
    freopen("p2.out","w",stdout);

    while(1){
        scanf("%d",&N);
        if(!N)
            return 0;
        scanf("%d",&L);/*input N and L*/
        T0=NULL;
        for(i=1;i<=N;i++){
            scanf("%d",&x);
            T0=Insert(x,T0);/*The initially inserted numbers*/
        }
        for(i=1;i<=L;i++){
            T=NULL;
```

```
            for(j=1;j<=N;j++){
                scanf("%d",&x);
                T=Insert(x,T);
            }/* N integers to be checked.*/
                if(isequal(T,T0))
                    printf("Yes\n");/*If these two trees are equal,then print "Yes"*/
                else
                    printf("No\n");/*If these two trees are not equal,then print "No"*/
        }
    }
}

Tree   Insert( ElementType X, Tree T )
{

     if (T==NULL) /* Create and return a one-node tree */
    {
       T = malloc( sizeof( struct TreeNode ) );
       if ( T == NULL )
           return;
```

```
        else {
            T->Element = X;
            T->Left = T->Right = NULL; }
      }   /* End creating a one-node tree */
      else if ( X < T->Element ) /* If there is a tree */
          T->Left = Insert( X, T->Left );
    else if ( X > T->Element )
          T->Right = Insert( X, T->Right ); /*All the N numbers are distinct and no greater than N. */
    return   T;
}

int isequal(Tree T1, Tree T2 ) /*The function that shows two trees are equal or not*.If they're same,then return 1,else return 0.*/
{
    if(T1==NULL&&T2==NULL)/*If they're both NULL*/
        return 1;
    if(T1==NULL||T2==NULL)/*If only one of them are NULL*/
        return 0;
    if(T1->Element!=T2->Element)/*If they are both not NULL*/
        return 0;
    return isequal(T1->Left,T2->Left)*isequal(T1->Right,T2->Right);
```

}

Declaration:

We hereby declare that all the work done in this project titled " Binary Search Tree" is of our independent effort as a group.

Duty Assignments:

Programmer: 刘理铖

Tester: 施能

Report Writer:魏张鉴