Xu Han

Project about traversals.

Algorithm mechanism

The algorithm is based on the properties of inorder traversal and postorder traversal.

Inorder traversal walk the tree always in the sequence:

1. left hand subtree 2. root 3. right hand subtree

postorder traversal walk the tree always in the sequence:

1. left hand subtree 2. right hand subtree. 3.root

The main idea is that we use postorder sequence to find the root of the tree or a subtree. Root is the node whose position in postorder is the largest because it is walked last by postorder traversal. Then we divide the inorder sequence by the root. We will get two subtrees. Left-hand subtree's positions are smaller than that of the root while right-hand subtree's positions are larger in inorder sequence. We repeat this until we divide all the subtrees into leaves, which means we find all nodes and get all the relations between the nodes. At last we store all the information and also the tree is stored.

To build a tree by the two sequences, the algorithm has following steps:

1. Use function *maxposition()* to find the inorder sequence element whose position is largest in postorder sequence. It returns the position in inorder sequence, "m". It is the root of the tree or a subtree.
2. use function *position()* to find the position "i" in postorder sequence. Pass it to next call to assign LC or RC and parent.
3. Check if it is a counterexample.
4. Assign LC or RC of last root(r), parent of current root(m) by the tag of hand—"h".
5. Divide the inorder sequence by the root, by calling divideST(m+1, e,i,1…..) and divideST(b,m-1,i,0….)

6. Repeat 1~5 until finding the child which is a leaf or no children. When meet these, assign LC or RC and parent

By recursively dividing, we can assign all the information to store the tree.

Explanation

1. For step 1,

Why choose a root? Because we need to use the root to divide other elements into a left hand subtree and a right hand subtree.

Why division? By dividing, we can get all the nodes and their relations. Then store the relations. We divide the sequence until we get all the information of all the nodes.It sounds like searching all the nodes and recording the relations of all the nodes.

How to find a root? As the hint that you gave, the last element of postorder sequence is important. It is always the root of a tree or a subtree. Because postorder always walk the root at last.

So we can get the root from the elements whose position in postorder sequence is the largest. I choose an element from the inorder sequence as the root, by comparing each element's position of postorder sequence. Root is the last one walked in a tree by postorder traversal.

2. For step 2,

Why recording element's position in postorder sequence? After we get the root, we will store its relations in the future. So we need to keep the "old root", and pass its position to following calls. I use postorder sequence as the node's label array, that's the reason why I record position in postorder sequence.

3. For step 3,

Why check? We need to check if the two sequences can build the same tree. The counterexample happens when some elements of the right hand subtree walk before any elements of the left hand subtree in postorder traversal.

Counter example:

Inorder: 9 1 4 0 3

postorder: 9 3 1 0 4

Root is 4. Then divide the first sequence into “9 1” and “0 3”. “3” is the rightmost node of the tree, because we walk it last by in order traversal. “9” is the leftmost node of the tree. Because it is the node we walk first. So if we want to walk the tree by postorder traversal, we cannot walk the rightmost one just after walk the leftmost one.

In popular situation, because the root is the only node connecting the left hand subtree and right hand subtree, it is impossible to walk an element in the right hand subtree and then back to walk any element in the left hand subtree.

So I claim: *in post order traversal, elements in the right hand subtree must walk after all the elements in the left subtree.*

How to check? I check this, by finding the postorder sequence positions of all the elements in the left hand subtree and positions of all the elements in the right hand subtrees. Then I compare the two positions. *If position of a right hand subtree element is located before the position of any left hand subtree element, we need to report error and return 1 for a symbol of error.*

eg. “3” is P[1], “1” is P[2], 1<2 then report error, return.

4. For step 4,

After checking, we’ve verified the two sequences, so we need to store the relation between the current root that we just found in step 1and the old root of last call, passed by the step 2 of last call. Tag of hand is “h”. Assume h=1 is right hand, h=0 is left hand.

5. For step 5,

Divide the current tree or subtree by calling divideST(m+1,e,i,1….) which is for righthand subtree, and divideST(b,m-1,i,0….) which is for left hand subtree.

Why divide the inorder sequence? the algorithm is based on the inorder traversal’s property that elements located before the root are all the nodes of its left hand subtree, and elements located after the root are all the nodes of its right hand subtree.

We use this property to divide the tree until we find a child which is a leaf or no children, rather than a subtree containing many elements, which means we’ve got to the end in this branch.

6. For step 6,

1. child which is leaf

When we find a child which is a leaf, we cannot and don't need to divide this child. In this situation (e==b). We just store it. If it is a left hand subtree of last root r, then LC[r]= P[position(P,I[b],n)]. Parent of this child is the last root r. Similar situation for a right hand child that is a leaf.

2. No children

This is a situation we have to consider. It happens when the root is either the rightmost or leftmost node in a tree or a subtree. If it is a left hand subtree, h=0, then LC of last root r is NULL. Similar case happens when it is a right hand subtree division.

Summary

In a word, the algorithm gets all the relationship information of all the nodes in the tree like LC, RC and parent, by dividing the inorder sequence into two subtrees until reaching every nodes in the tree. Each division implements by finding the root of the tree and then treat the elements before the root as left hand subtree, and treat the elements after the root as right hand subtree. This is because inorder traversal always walks the left hand subtree before the root, and walks the root before the right hand subtree. Deciding the root element depends on the postorder traversal's property that it always walks the root node last. To store the tree, postorder sequence is also considered the array of Label. To avoid the situation that the given inorder sequence and postorder sequence cannot build the same binary tree, a check function has been adopted, which is base on the property that in postorder traversal a right hand subtree element cannot be walked before any left hand subtree element.

Coding

1. function divideST()

```
int divideST(int b, int e, int r, int h, int *P, int *I, int *lc, int *rc, int *par, int n)
//divide the array I into subtrees, h stands for hand, r stands for the positon of root in
array P
{int m,i,p,checkb=0;
   if (b-e>0){                          // no children situation
     if (h==0) lc[r]=101;                    //use 101 as a symbol of NULL
      else rc[r]=101;
   }
      else{if(b==e) {                  //I[b] is a leaf child of P[r],P[p] is a leaf.
      p=position(P,I[b],n);
         par[p]=P[r];
```

```
            if (h==0) lc[r]=P[p];
            else {rc[r]=P[p];}
            rc[p]=101;
            lc[p]=101;
        }
    else{
        m=maxposition(b,e,P,I,n);     //find the root of the subtree : Step 1
        i=position(P,I[m],n);   //find the positon of root in array P: Step 2

        checkb=check(P,I,n,b,e,m);                //check the situation of(b)
         if(checkb==1) return 0;
        par[i]=P[r];             //set the par lc and rc of node i, which is the root of this subtree
        if (h==0) {lc[r]=P[i];}
        else{ rc[r]=P[i];}

        divideST(m+1,e,i,1,P,I,lc,rc,par,n);  //divide the right hand subtree
        divideST(b,m-1,i,0,P,I,lc,rc,par,n);  //divide the left hand subtree
        }
    }
    return 0;
}
```

2. function position()

```
int position(int *P,int val,int n)    //find a value's position in array P[]
{
    for(int j=0;j<n;j++){
        if (P[j]==val) return j;
    }
    return 0;
}
```

3. function maxposition()

```
int maxposition(int b, int e, int *P, int *I, int n)  // the root of a subtree has the maximal position number in array P
{    int max=-1,pos;
```

```
   for(int k=b;k<e+1;k++)
   {   pos=position(P,I[k],n);
         if(pos>max) max=pos;      //get the maximal position
      }
   return position(I,P[max],n);
}
```

4. function check()

```
int check(int *P,int *I,int n,int b, int e,int m){
   int checkl=0,checkr=0;          // define two parameters to compare
   for(int c=b;c<m;c++){
      checkl=position(P,I[c],n);     //get one left hand subtree element I[c]'s position in P
array
      for(int d=m+1;d<e+1;d++){
         checkr=position(P,I[d],n);   //get one right hand subtree element I[d]'s position in
P array
         if(checkr<checkl){std::cout<<"error: elements of right hand subtree cannot
traverse before elements of left hand subtree in postorder travesal"<<std::endl;
            return 1;}      // if we walk the right hand subtree element before the left one
return 1, decided by compare checkl with checker
      }
      }
   return 0;
}
```

Code test

Example:

Inorder: 9 3 1 0 4 2 7 6 8 5 I[]

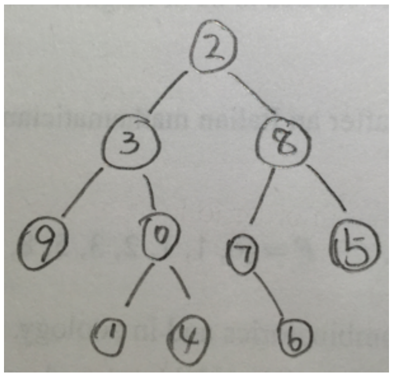
postorder: 9 1 4 0 3 6 7 5 8 2 P[]

The root of the tree is P[n-1]. Find the position of P[n-1] in array I[], refers to ir. Then first call *check*(). At last initial call the divideST(ir+1,e,n-1,1,P,I,lc,rc,par,n) and divideST(b,ir-1,n-1,0,P,I,lc,rc,par,n). Then we can start to build the tree and complete it.

The result is as follows:

```
No.0 label 9 LC 101 RC 101 parent 3
No.1 label 1 LC 101 RC 101 parent 0
No.2 label 4 LC 101 RC 101 parent 0
No.3 label 0 LC 1 RC 4 parent 3
No.4 label 3 LC 9 RC 0 parent 2
No.5 label 6 LC 101 RC 101 parent 7
No.6 label 7 LC 101 RC 6 parent 8
No.7 label 5 LC 101 RC 101 parent 8
No.8 label 8 LC 7 RC 5 parent 2
No.9 label 2 LC 3 RC 8 parent 101
```

Use this result, we can draw the tree like this.


2
3
8
9
0
7
5
1
4
6