

05-G 后序遍历

#数据结构邓神

观察

递归实现


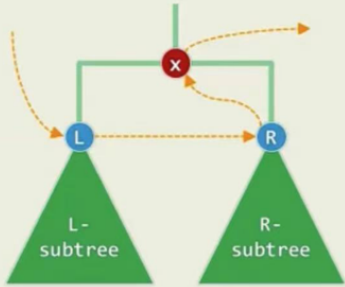
❖ 应用: `BinNode::size()` + `BinTree::updateHeight()`

❖ `template <typename T, typename VST>`

```
void traverse( BinNodePosi<T> x, VST & visit ) {  
    if ( ! x ) return;  
    traverse( x->lc, visit );  
    traverse( x->rc, visit );  
    visit( x->data );  
}
```

❖ $T(n) = O(1) + T(a) + T(n - a - 1) = O(n)$

❖ 挑战: 不依赖递归机制, 如何实现后序遍历? 效率如何?



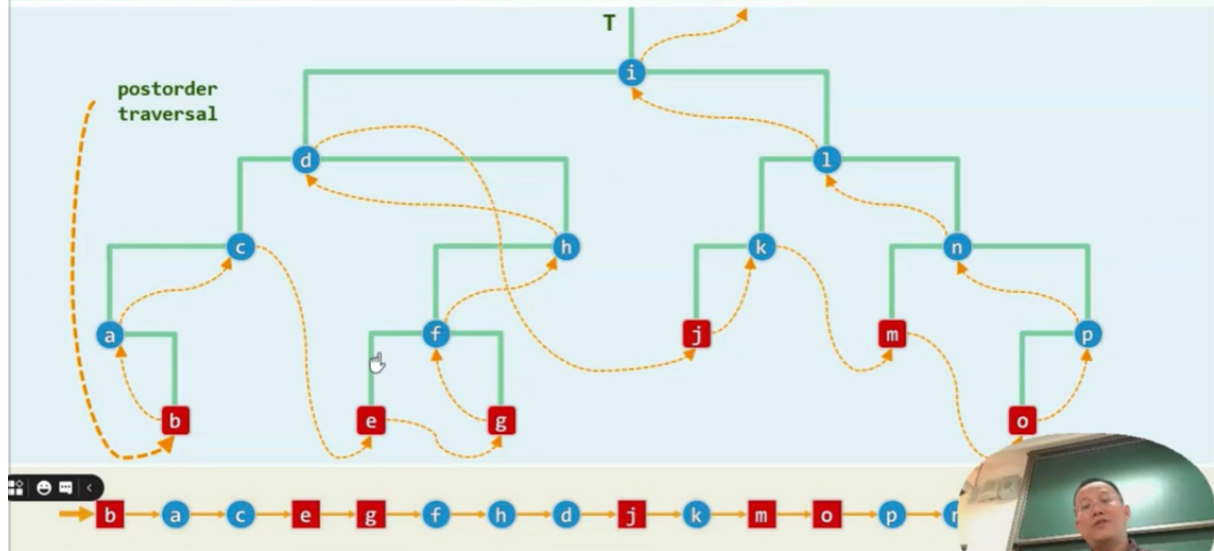
Data Structures & Algorithms, Tsinghua University

// 后序遍历 递归

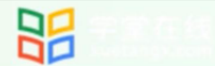
```
template <typename T, typename VST> void After_traverse(BinNodePosi(T) x, VST &  
visit){  
    if (!x){  
        return;  
    }  
    After_traverse(x->lChild, visit);  
    After_traverse(x->rChild, visit);  
    visit(x->data);  
}
```

// $T(n) = O(1) + T(a) + T(n - a - 1) = O(n)$

观察



藤缠树



❖ 从根出发下行

尽可能沿**左**分支

实不得已，才沿**右**分支

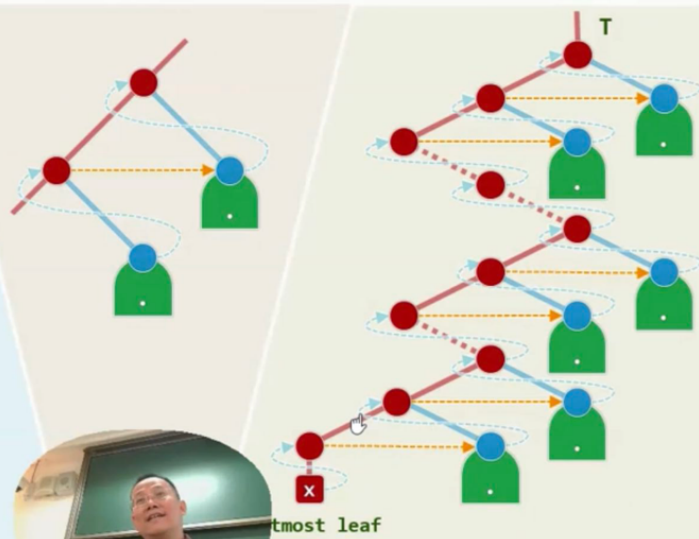
❖ 最后一个节点

必是叶子，而且

是按中序遍历次序**最靠左者**

也是递归版中visit()首次**执行处**

❖ 这片叶子，将首先接受访问...



迭代算法

```
// 后序遍历 迭代算法
template <typename T,typename VST> static void
gotoLeftmostLeaf(stack<BinNodePosi(T)> & S){
    while (BinNodePosi(T) x = S.top()){
        if (HasLChild(* x)){
            if (HasRChild(* x)){
                S.push(x->rChild); // 右边的孩子要新入栈，后面出来
            }
            S.push(x->lChild);
        }
        else {

```

```

        S.push(x->rChild); // 没有左孩子才考虑右边的孩子
    }
}

// 按照前面的算法最后一定会推入一个空节点一定要弹出
S.pop();
}

template <typename T,typename VST> void travPost_I(BinNodePosi(T) x,VST &
visit){
    stack<BinNodePosi(T)> S;
    if (x) {
        S.push(x);
    }
    while(!S.empty()){
        if (S.top != x->parent){
            gotoLeftmostLeaf(S);
        }
        x = S.top();
        S.pop();
        visit(x->data);
    }
}

```

序曲

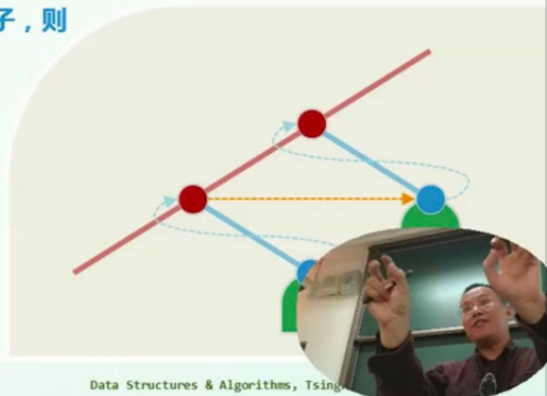


学堂在线
清华大学

```

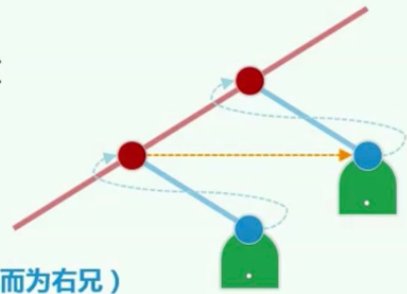
❖ template <typename T> static void gotoLeftmostLeaf( Stack <BinNodePosi<T>> & S )
    while ( BinNodePosi<T> x = S.top() ) //自顶而下反复检查栈顶节点
        if ( HasLChild( * x ) ) { //尽可能向左。在此之前
            if ( HasRChild( * x ) ) //若有右孩子，则
                S.push( x->rc ); //优先入栈
            S.push( x->lc ); //然后转向左孩子
        } else //实不得已
            S.push( x->rc ); //才转向右孩子
        S.pop(); //返回之前，弹出栈顶的空节点

```

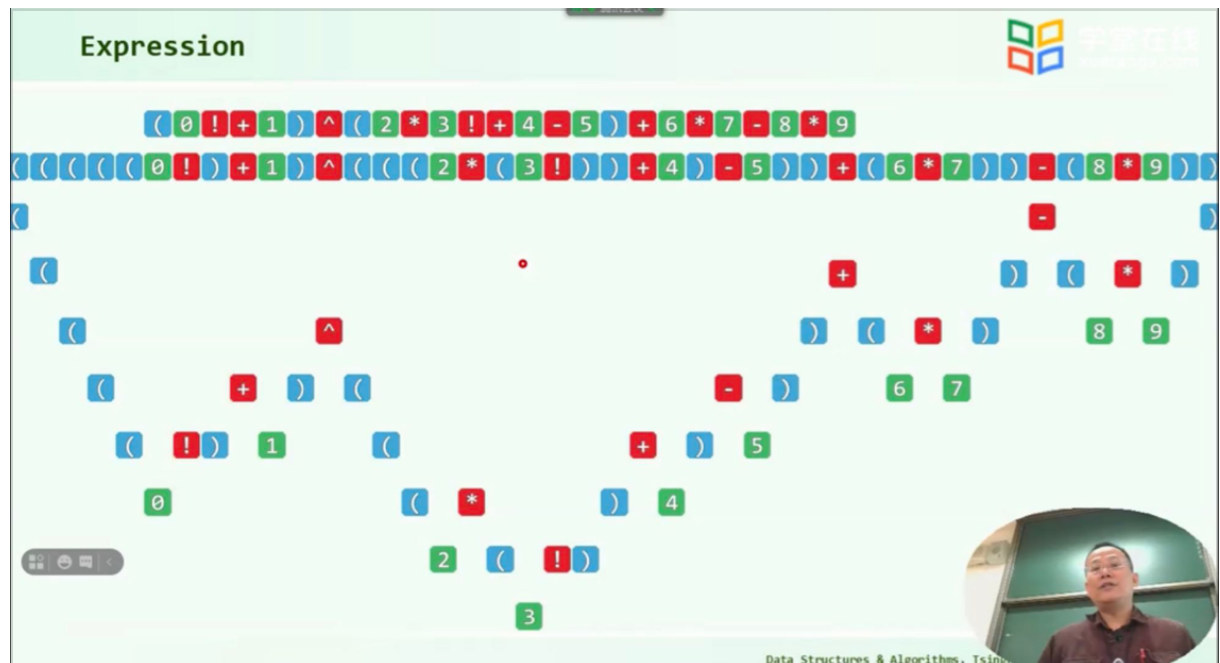


```

❖ template <typename T, typename VST>
void travPost_I( BinNodePosi<T> x, VST & visit ) {
    Stack < BinNodePosi<T> > S; //辅助栈
    if ( x ) S.push( x ); //根节点首先入栈
    while ( ! S.empty() ) { //x始终为当前节点
        if ( S.top() != x->parent ) //若栈顶非x之父 ( 而为右兄 )
            gotoLeftmostLeaf( S ); //则在其右兄子树中找到最靠左的叶子 ( 递归深入 )
        x = S.pop(); //弹出栈顶 ( 即前一节点之后继 ) 以更新x
        visit( x->data ); //并随即访问之
    }
}
  
```



表达式树 Expression Tree ~ Post-order ~ RPN



在我们层次化的过程中其实我们获得了一颗树

