5.二叉树

（e2）中序遍历

邓俊辉

deng@tsinghua.edu.cn

递归

❖ template <typename T, typename VST>

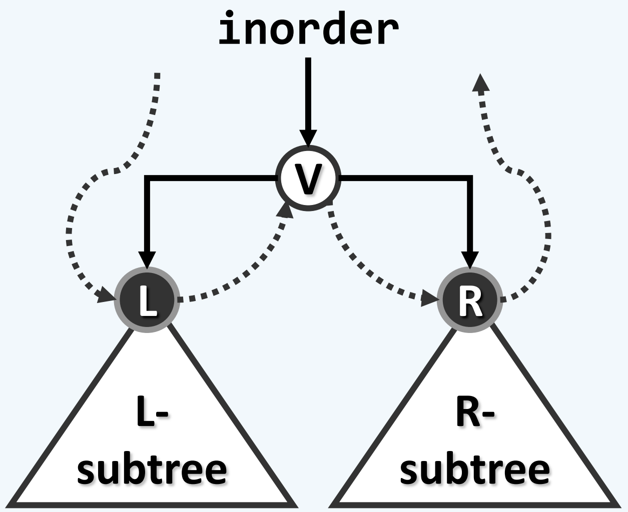
```
void traverse( BinNodePosi(T) x, VST & visit ) {
   if ( !x ) return;
   traverse( x->lChild, visit );
   visit( x->data );
   traverse( x->rChild, visit );
} //T(n) = T(a) + O(1) + T(n-a-1) = O(n)
```

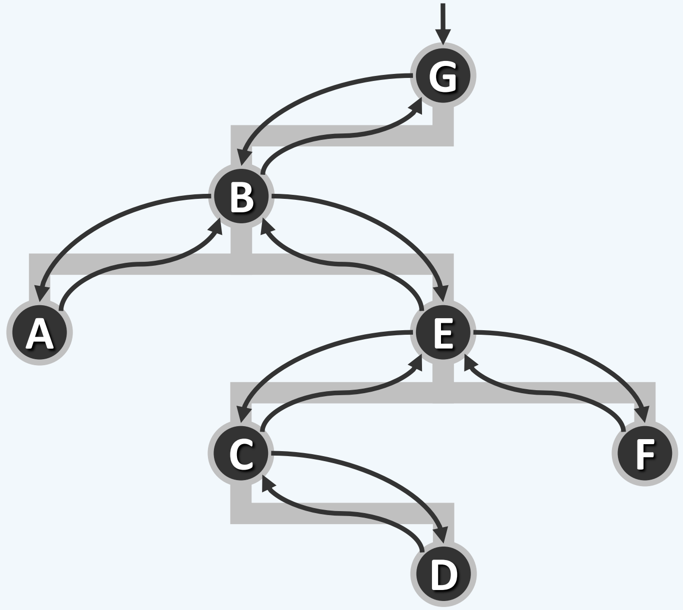
❖ 中序输出文件树结构

printBinTree()

❖ 挑战： 不依赖递归机制，能否实现中序遍历？

如何实现？效率如何？





难点

❖ 难度在于

尽管右子树的递归遍历是尾递归，但左子树却严格地 不是

❖ 解决方法

找到 第一个 被访问的节点 //仿照迭代的先序遍历算法

将其祖先用栈保存 //按照被访问过程的逆序

❖ 这样，原问题就被分解为

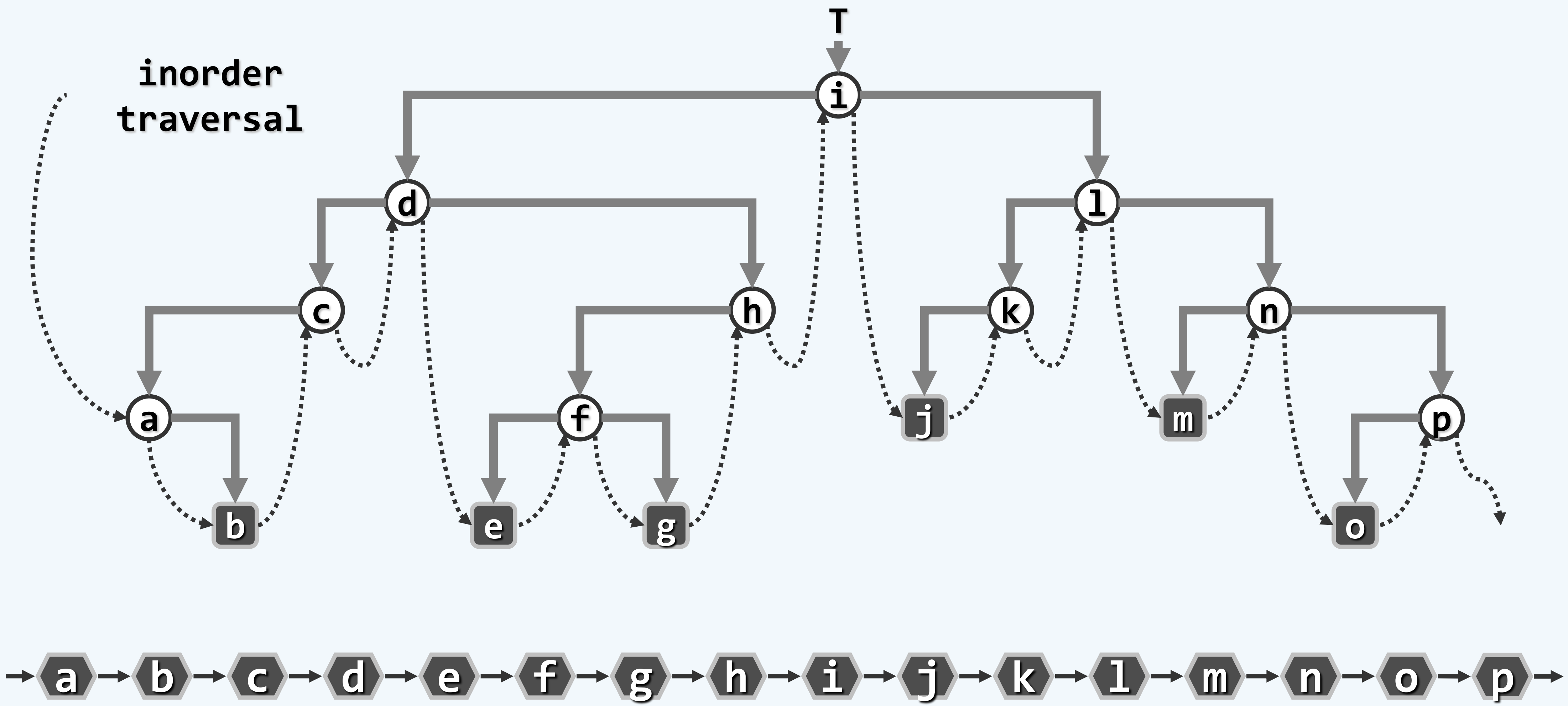
依次对若干棵 右子树 的遍历问题 //依什么“次”？

❖ 于是，首先要解决的问题就是：

中序遍历任一二叉树T时

首先被访问的是哪个节点？如何找到它？

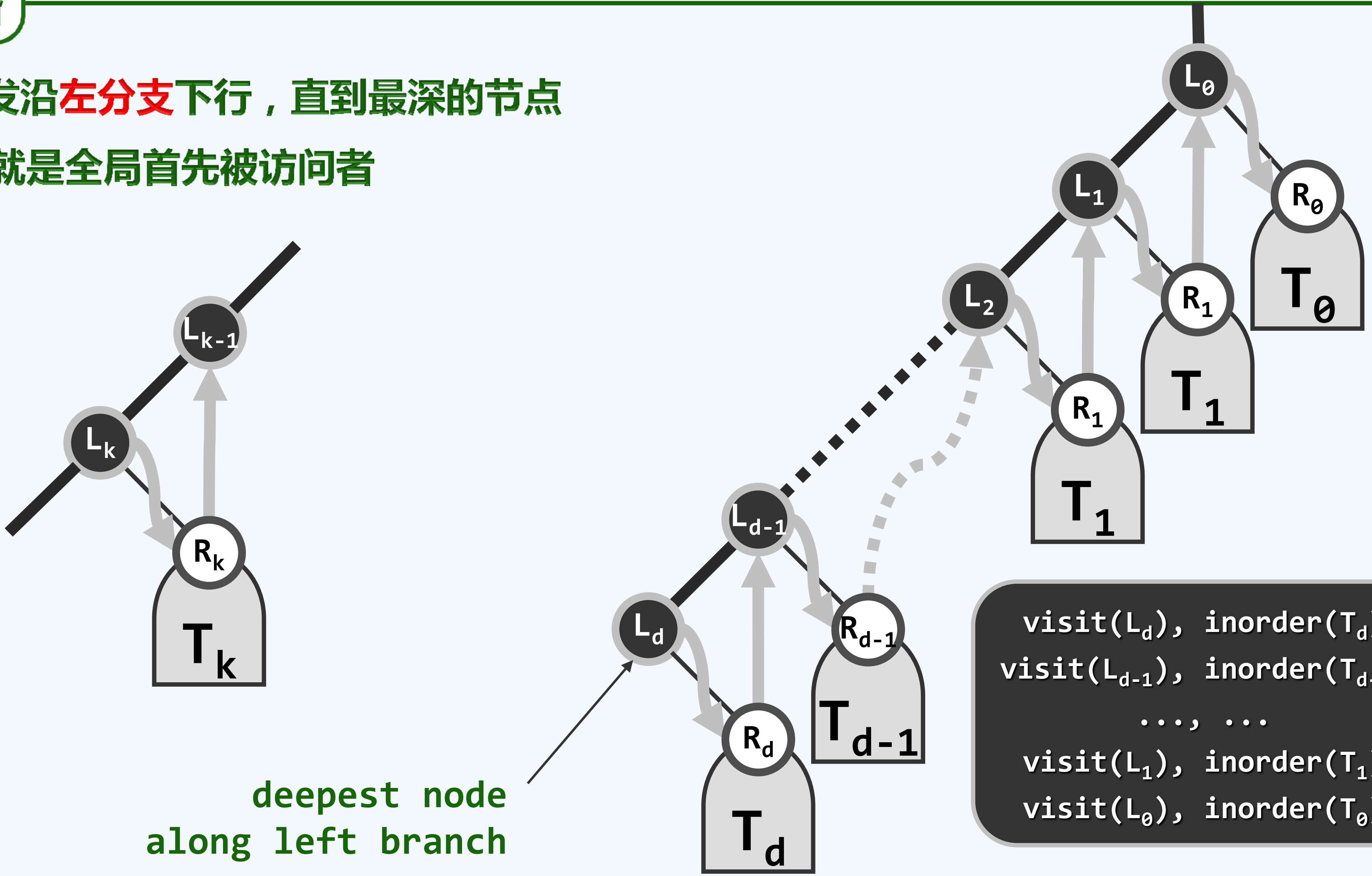
观察



思路

❖ 从根出发沿**左分支**下行，直到最深的节点

——它就是全局首先被访问者



visit(L_d), inorder(T_d)
visit(L_{d-1}), inorder(T_{d-1})
..., ...
visit(L_1), inorder(T_1)
visit(L_0), inorder(T_0)

实现

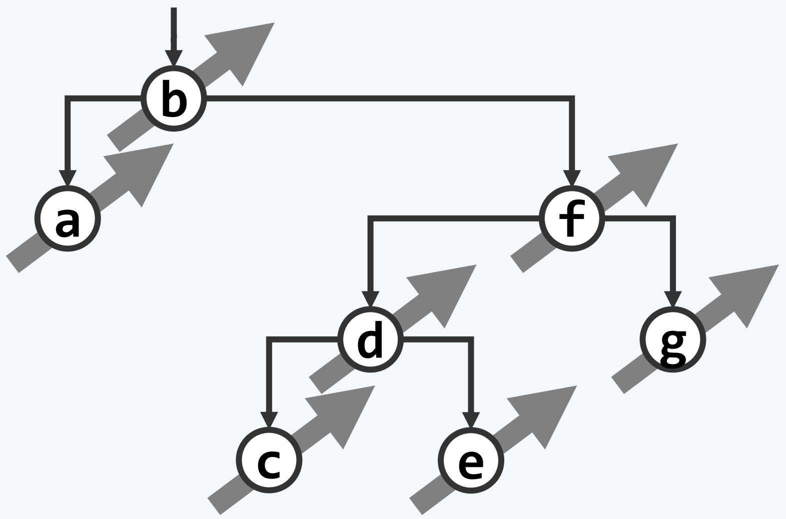
- template <typename T>

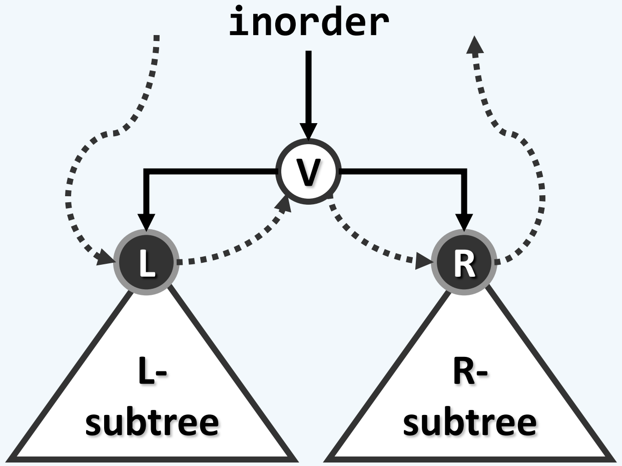
```
static void goAlongLeftBranch( BinNodePosi(T) x, Stack <BinNodePosi(T)> & S )
   { while (x) { S.push(x); x = x->lChild; } } //反复地入栈，沿左分支深入
```

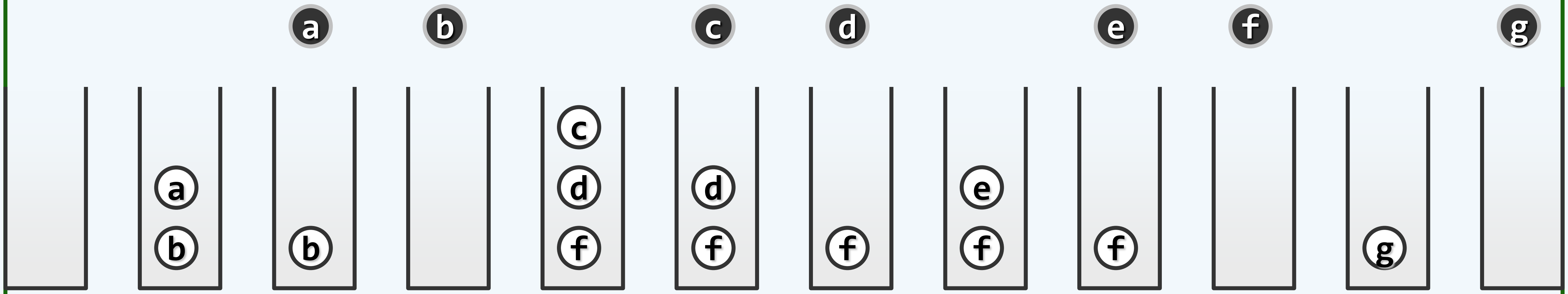
- template <typename T, typename V> void travIn_I1(BinNodePosi(T) x, V& visit) {

```
   Stack <BinNodePosi(T)> S; //辅助栈
   while (true) { //反复地
      goAlongLeftBranch( x, S ); //从当前节点出发，逐批入栈
      if ( S.empty() ) break; //直至所有节点处理完毕
      x = S.pop(); //x的左子树或为空，或已遍历（等效于空），故可以
      visit( x->data ); //立即访问之
      x = x->rChild; //再转向其右子树（可能为空，留意处理手法）
   }
}
```

实例







正确性

❖ 可归纳证明：

每个节点出栈时

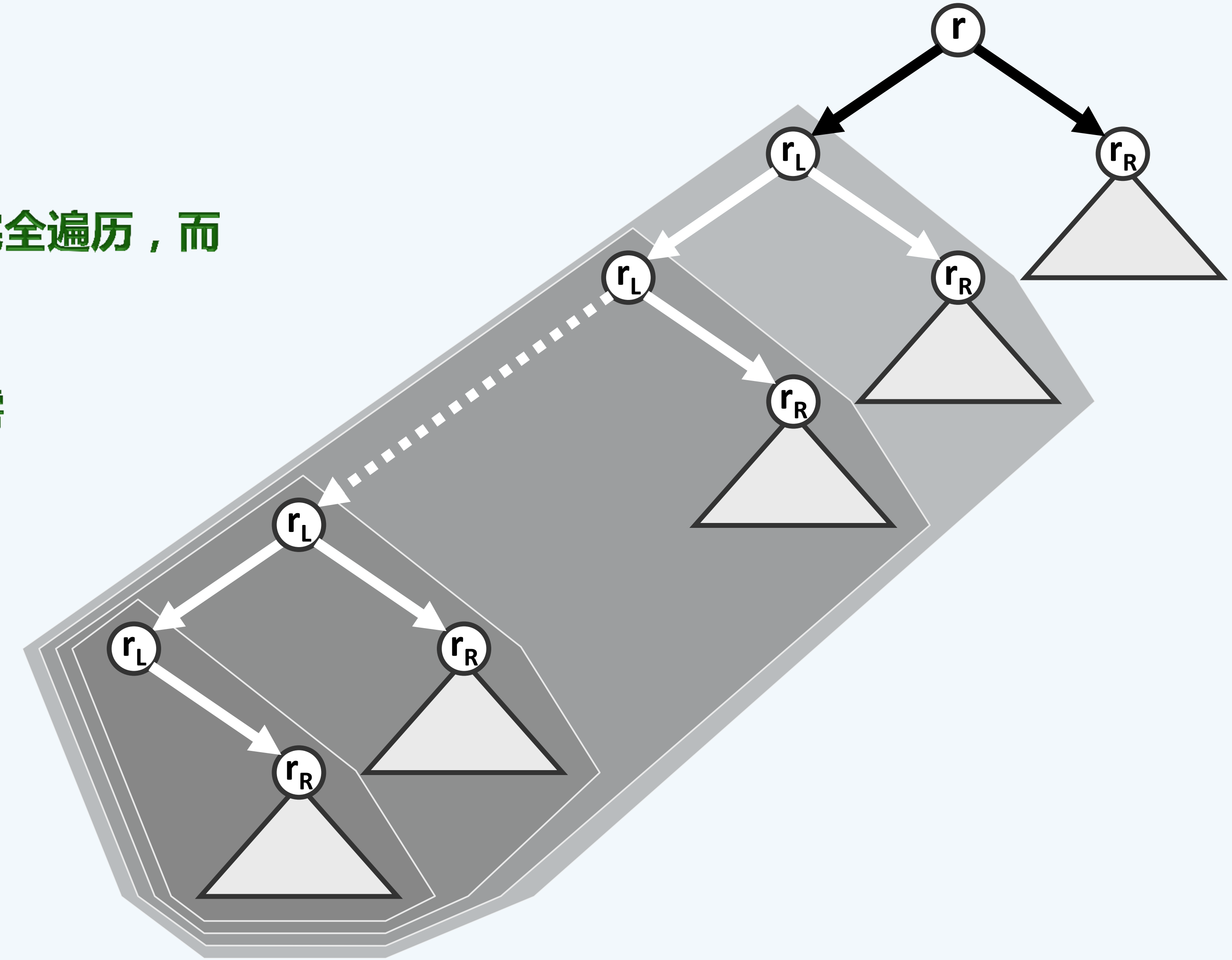
其左子树（若存在）已经完全遍历，而

右子树尚未入栈

❖ 于是，每当有节点出栈，只需

访问它，然后

从其右孩子出发. . .



效率

- 是否$\mathcal{O}(n)$，取决于以下条件

  1) 每次迭代，都恰有一个节点出栈并被访问 //满足

  2) 每个节点入栈一次且仅一次 //满足

  3) 每次迭代只需$\mathcal{O}(1)$时间 //不再满足，因为...

- 单次调用goAlongLeftBranch()

  就可能需做$\Omega(n)$次入栈操作，共需$\Omega(n)$时间

- 既然如此，难道总体将需要...$\mathcal{O}(n^2)$时间？

- 事实上，这个界远远不紧...

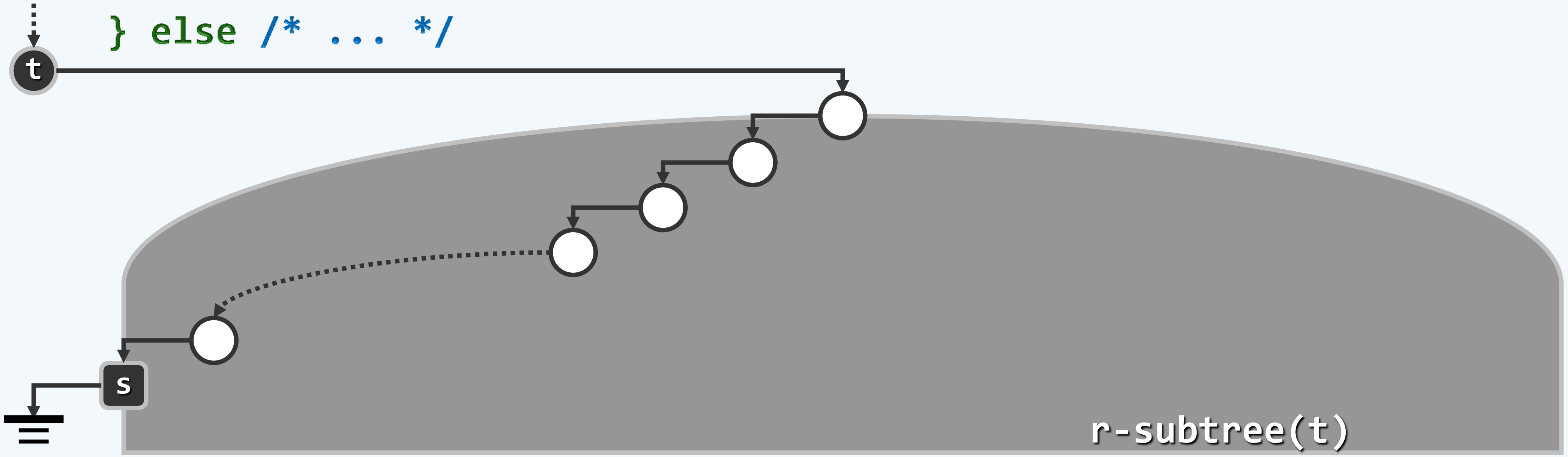
  请利用分摊原理，自行分析

- 更多的实现：travIn_I2() + travIn_I3() + travIn_I4()

直接后继

❖ template <typename T> //稍后将被BST::remove中的removeAt()调用

```
BinNodePosi(T) BinNode<T>::succ() { //在中序遍历意义下的直接后继
   BinNodePosi(T) s = this; //记录后继的临时变量
   if (rChild) { //若有右孩子，则直接后继必在右子树中，具体地就是
      s = rChild; while ( HasLChild(*s) ) s = s->lChild; //右子树中最小节点
   } else /* ... */
```



直接后继

```
❖   } else { //否则，后继应是“将当前节点包含于其左子树中的最低祖先”
        while ( IsRChild(*s) ) //根节点是左是右？
            s = s->parent; //逆向地沿右向分支，不断朝左上方移动
        s = s->parent; //最后再朝右上方移动一步，即抵达后继（若存在）
    } //两种情况的运行时间分别为当前节点的高度与深度，不过O(h)
    return s; //可能是NULL
}
```