

二叉树
树

e5 - A

越到你的高度上——那是我的深度！

藏在你的纯洁里——那是我的天真！

Two roads diverged in a yellow wood
And sorry I could not travel both

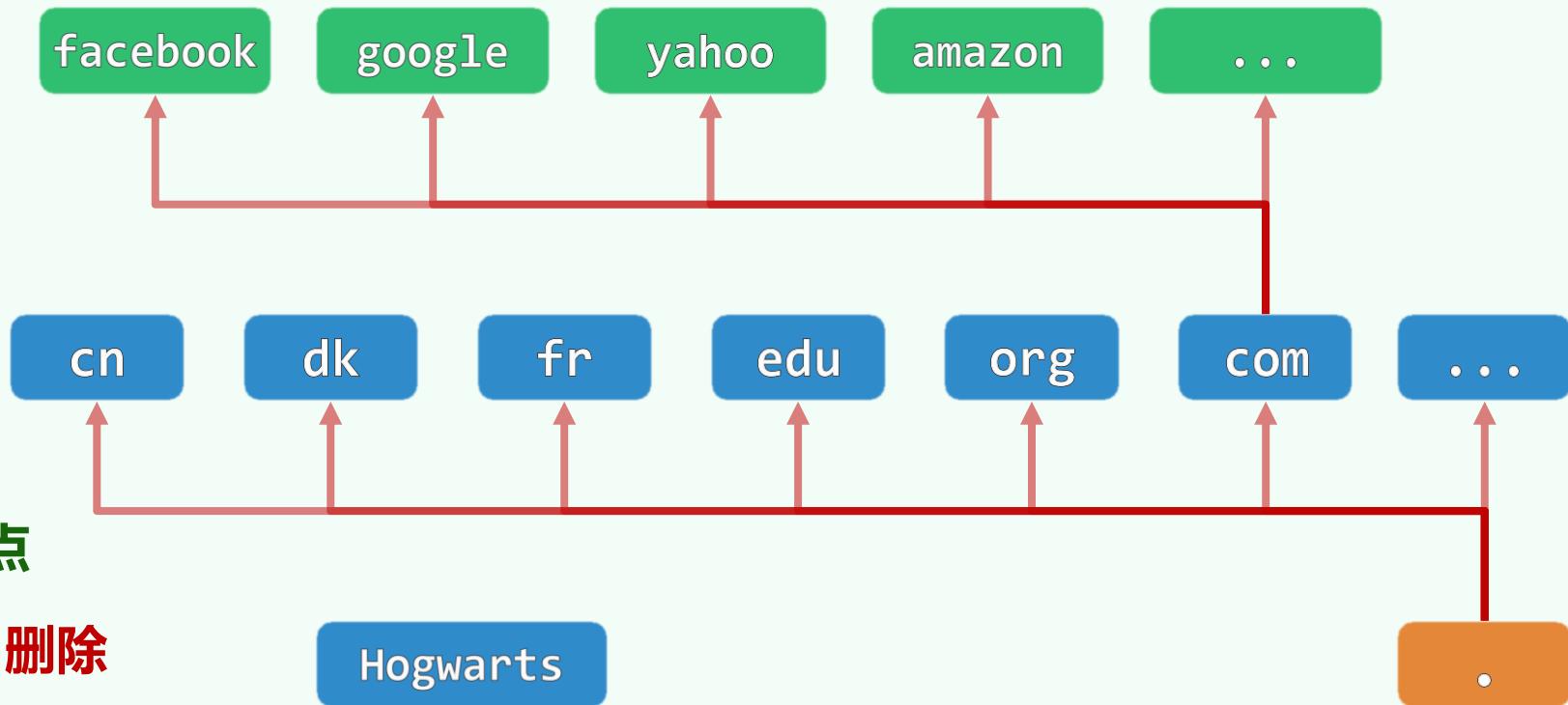
邓俊辉

deng@tsinghua.edu.cn

动机

❖ 【应用】层次结构的表示

- 表达式
- 文件系统
- URL ...



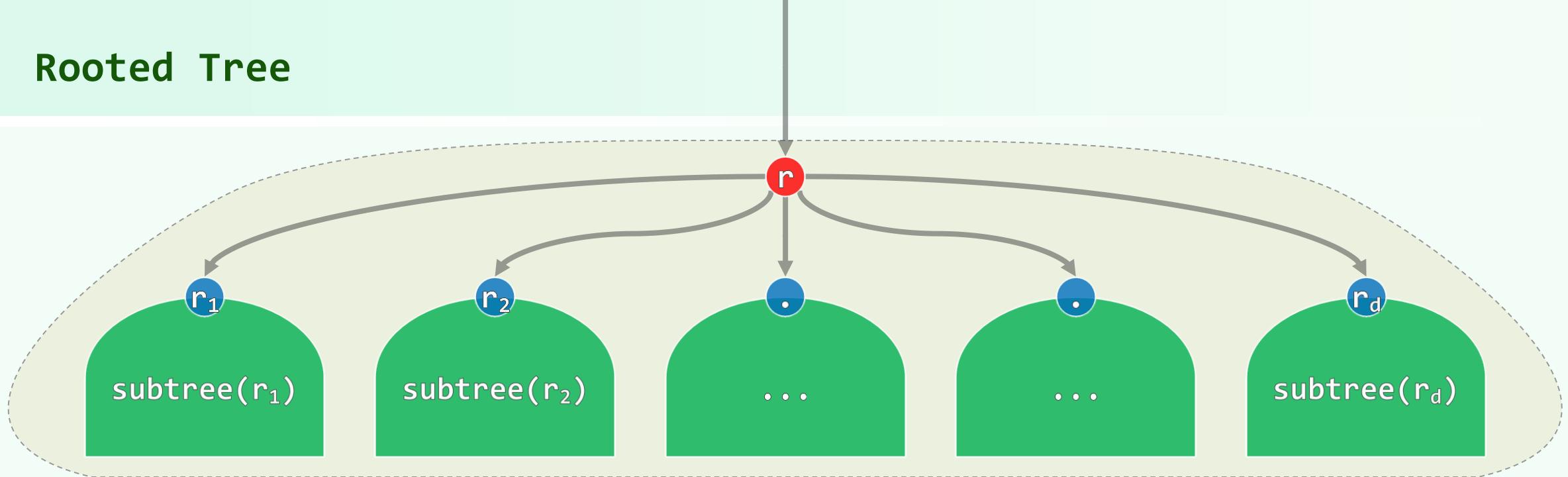
❖ 【数据结构】综合性

- 兼具Vector和List的优点
- 兼顾高效的查找、插入、删除

❖ 【半线性】

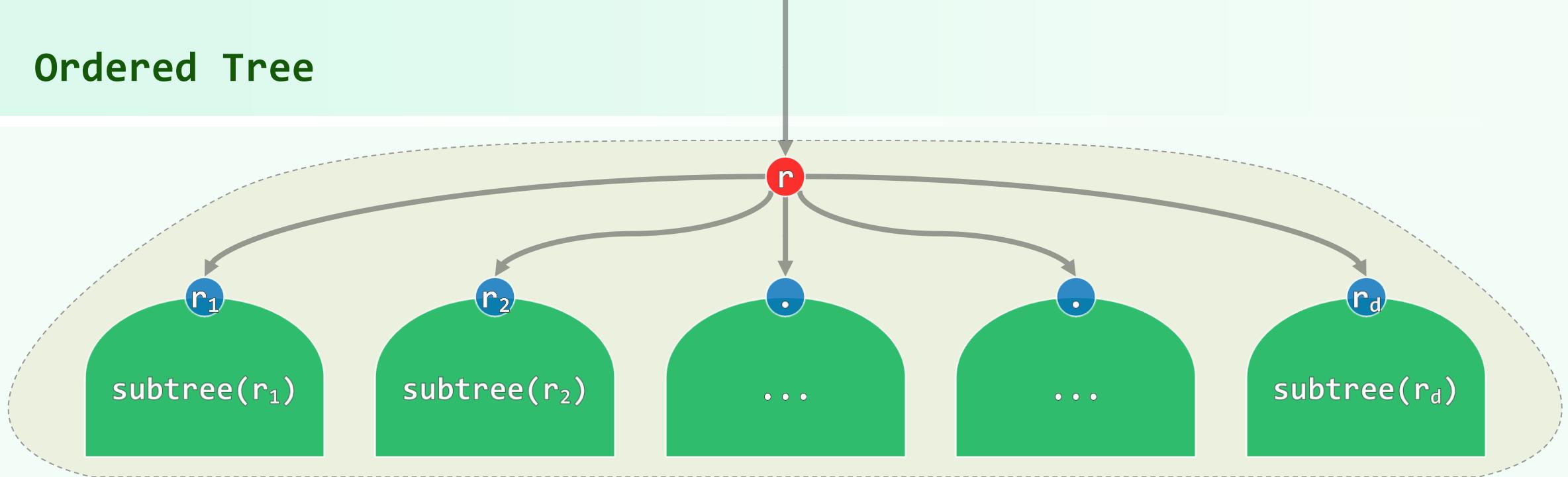
- 不再是简单的线性结构，但
- 在确定某种次序之后，具有线性特征

Rooted Tree



- ❖ 树是极小连通图、极大无环图 $\mathcal{T} = (V; E)$: 节点数 $n = |V|$, 边数 $e = |E|$
- ❖ 指定任一节点 $r \in V$ 作为根后, \mathcal{T} 即称作**有根树**
- ❖ 若 $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \dots, \mathcal{T}_d$ 为**有根树**, 则 $\mathcal{T} = ((\bigcup_i V_i) \cup \{r\}, (\bigcup_i E_i) \cup \{\langle r, r_i \rangle \mid 1 \leq i \leq d\})$ 也是
- ❖ 相对于 \mathcal{T} , \mathcal{T}_i 称作以 r_i 为根的**子树** (subtree rooted at r_i), 记作 $\mathcal{T}_i = subtree(r_i)$

Ordered Tree



- ❖ r_i 称作 r 的孩子 (child) , r_i 之间互称兄弟 (sibling)
- ❖ r 为其父亲 (parent) , $d = \text{degree}(r)$ 为 r 的 (出) 度 (degree)
- ❖ 可归纳证明: $e = \sum_{v \in V} \text{degree}(v) = n - 1 = \Theta(n)$
故在衡量相关复杂度时, 可以 n 作为参照
- ❖ 若指定 T_i 作为 T 的第 i 棵子树, r_i 作为 r 的第 i 个孩子, 则 T 称作有序树

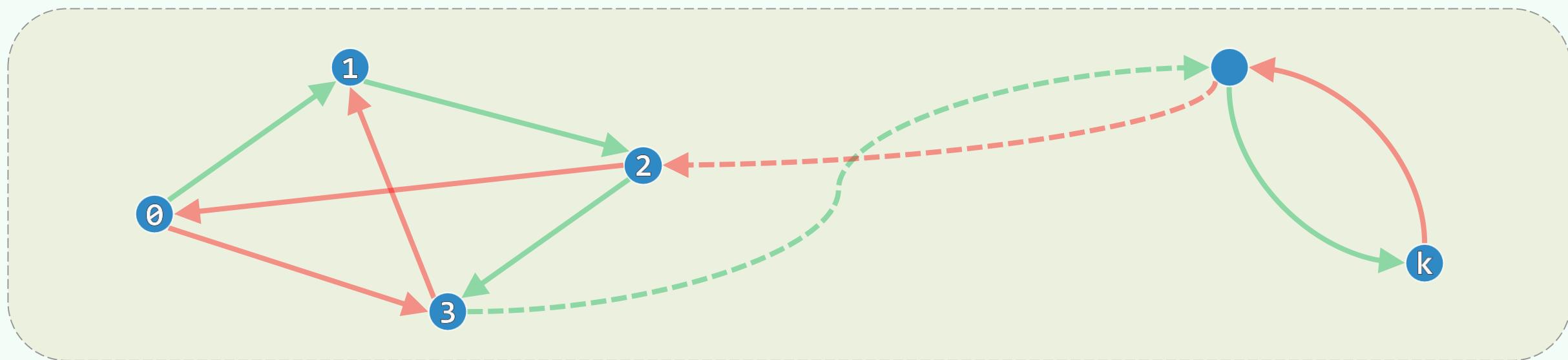
路径 + 环路

❖ V 中的 $k + 1$ 个节点，通过 V 中的 k 条边依次相联，构成一条路径/通路 (path)

$$\pi = \{ (v_0, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k) \}$$

❖ 路径长度即所含边数： $|\pi| = k$ //注意：早期文献，多以节点数为长度

❖ 环路 (cycle/loop) : $v_k = v_0$ //如果覆盖所有节点各一次，则称作周游 (tour)



连通 + 无环

❖ 连通图：节点之间均有路径 (connected)

不含环路，称作无环图 (acyclic)

❖ 树 = 无环连通图

= 极小连通图

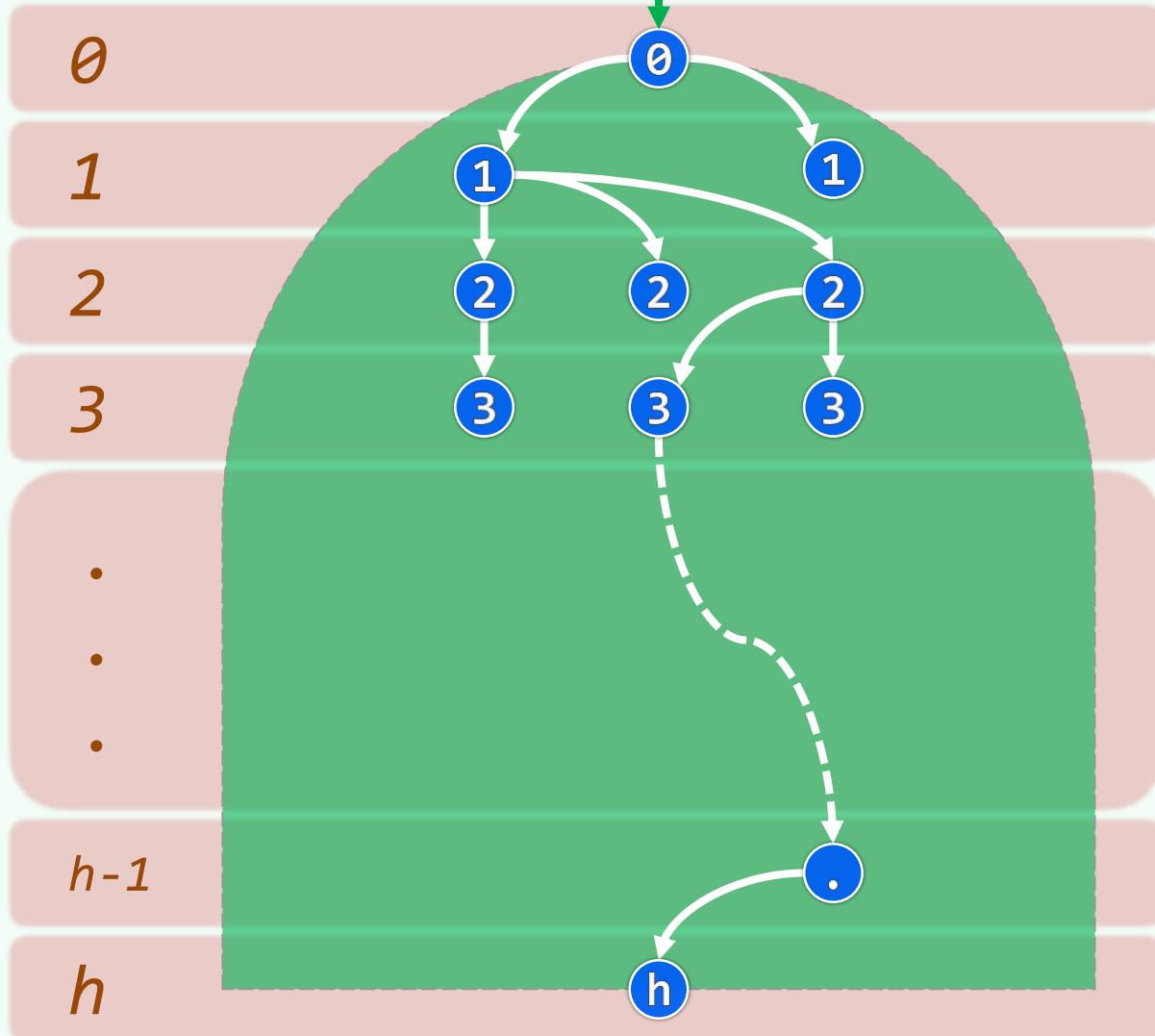
= 极大无环图

❖ 故任一节点 v 与根之间存在唯一路径

$\text{path}(v, r) = \text{path}(v)$

❖ 于是以 $|\text{path}(v)|$ 为指标

可对所有节点做等价类划分...



深度 + 层次

❖ 不致歧义时，路径、节点和子树可相互指代

- $\text{path}(v) \sim v \sim \text{subtree}(v)$

❖ v 的深度： $\text{depth}(v) = |\text{path}(v)|$

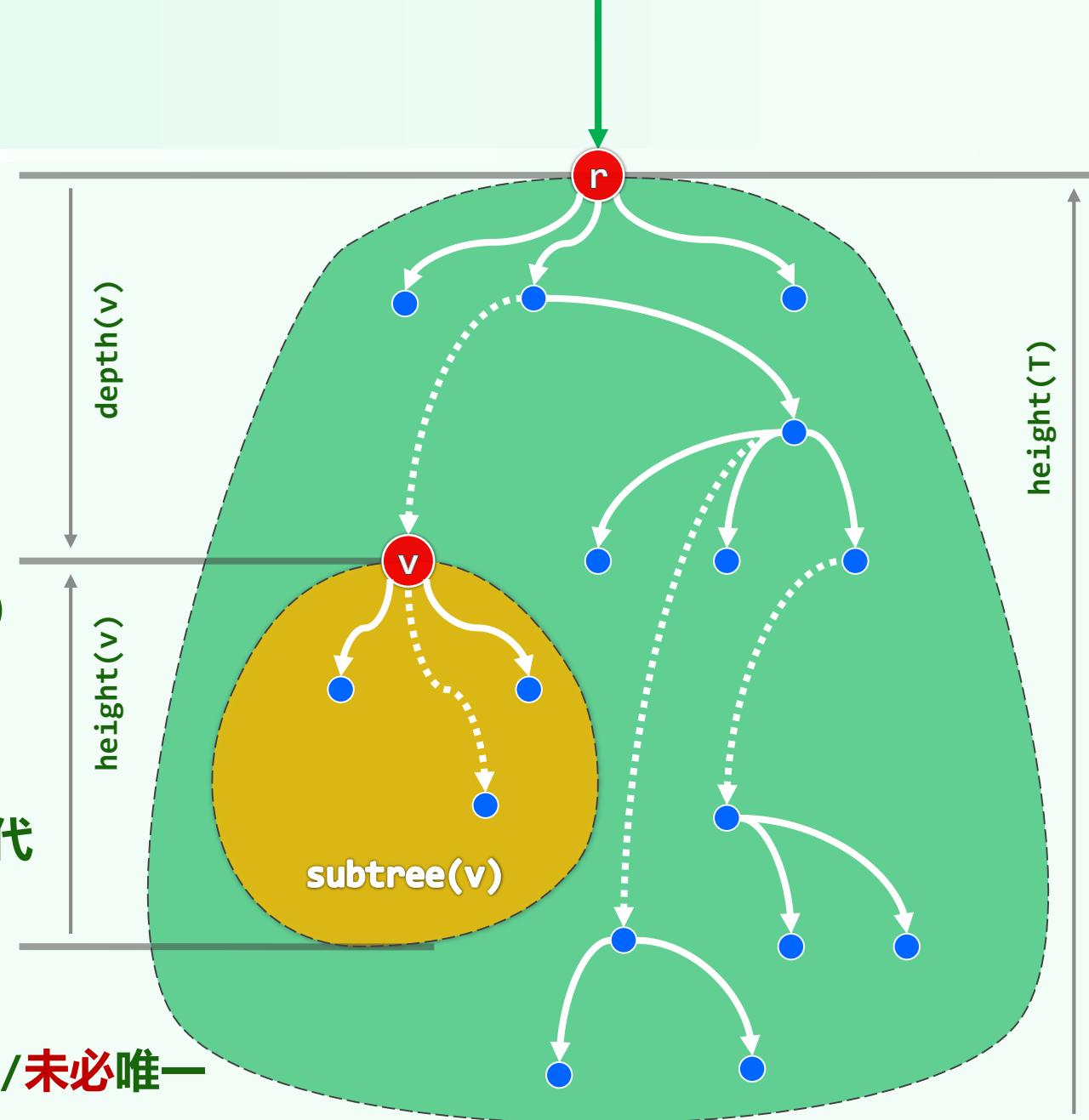
❖ $\text{path}(v)$ 上节点，均为 v 的祖先（ancestor）

v 是它们的后代（descendent）

❖ 其中除自身以外，是真（proper）祖先/后代

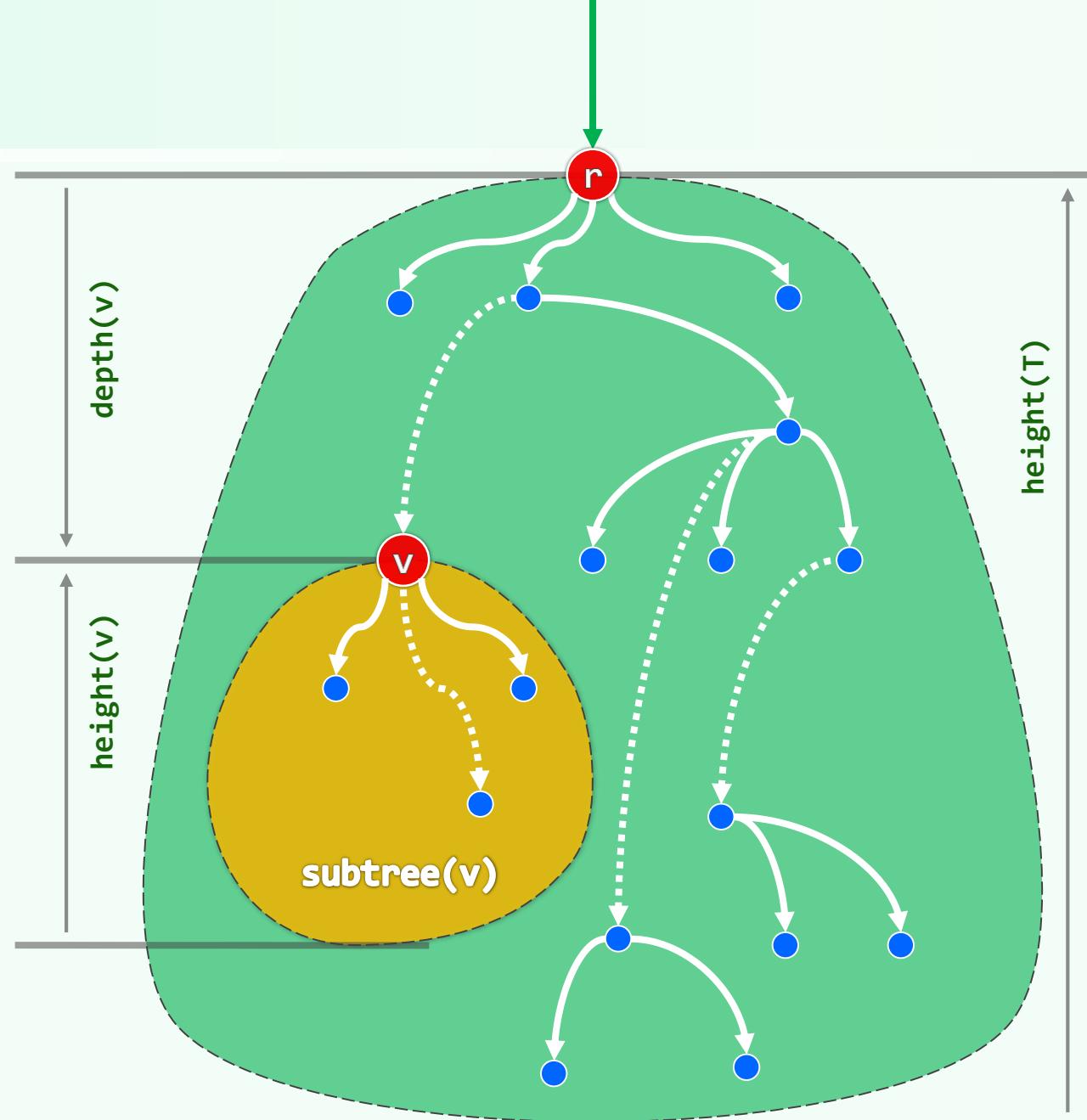
❖ 半线性：

在任一深度， v 的祖先/后代若存在，则必然/未必唯一



深度 + 层次

- ❖ 根节点是所有节点的**公共祖先**, 深度为0
- ❖ 没有后代的节点称作**叶子** (leaf)
- ❖ 所有叶子深度中的最大者称作**(子)树(根)的高度**
 - $\text{height}(v) = \text{height}(\text{subtree}(v))$
- ❖ 特别地, **空树的高度取作-1**
- ❖ $\text{depth}(v) + \text{height}(v) \leq \text{height}(T)$
何时取等号?



二叉树

树的表示

e5 - B

木晦於根，春榮華敷；人晦於身，神明內腴

然而现在他有了一个儿子，这是他的亲骨血，他所最亲爱的人，他可以好好地教养他，把他的抱负拿来在儿子的身上实现。儿子的幸福就是他自己的幸福。

邓俊辉

deng@tsinghua.edu.cn

接口

节点	功能
<code>root()</code>	根节点
<code>parent()</code>	父节点
<code>firstChild()</code>	长子
<code>nextSibling()</code>	兄弟
<code>insert(i, e)</code>	将e作为第i个孩子插入
<code>remove(i)</code>	删除第i个孩子 (及其后代)
<code>traverse()</code>	遍历

父节点

rank

0

1

2

3

4

5

6

7

8

9

	parent	data
0	2	H
1	7	E
2	9	F
3	4	B
4	4	R
5	2	K
6	7	D
7	4	A
8	2	G
9	4	C

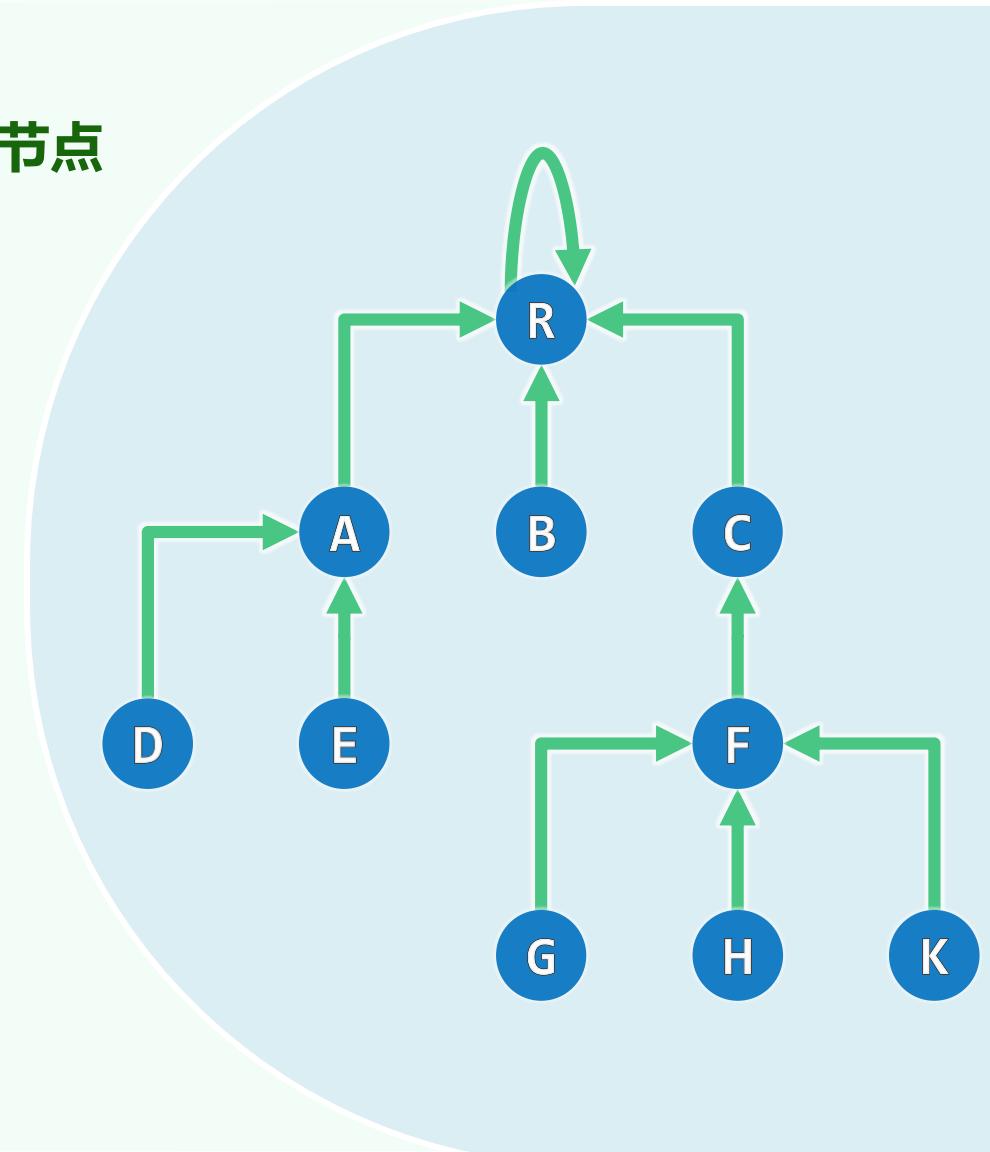
❖ 除根外，任一节点有且仅有一个父节点

❖ 节点组织为一个序列，各自记录：

data 本身信息

parent 父节点的秩或位置

❖ 树根：R ~ parent(4) = 4



孩子节点

rank

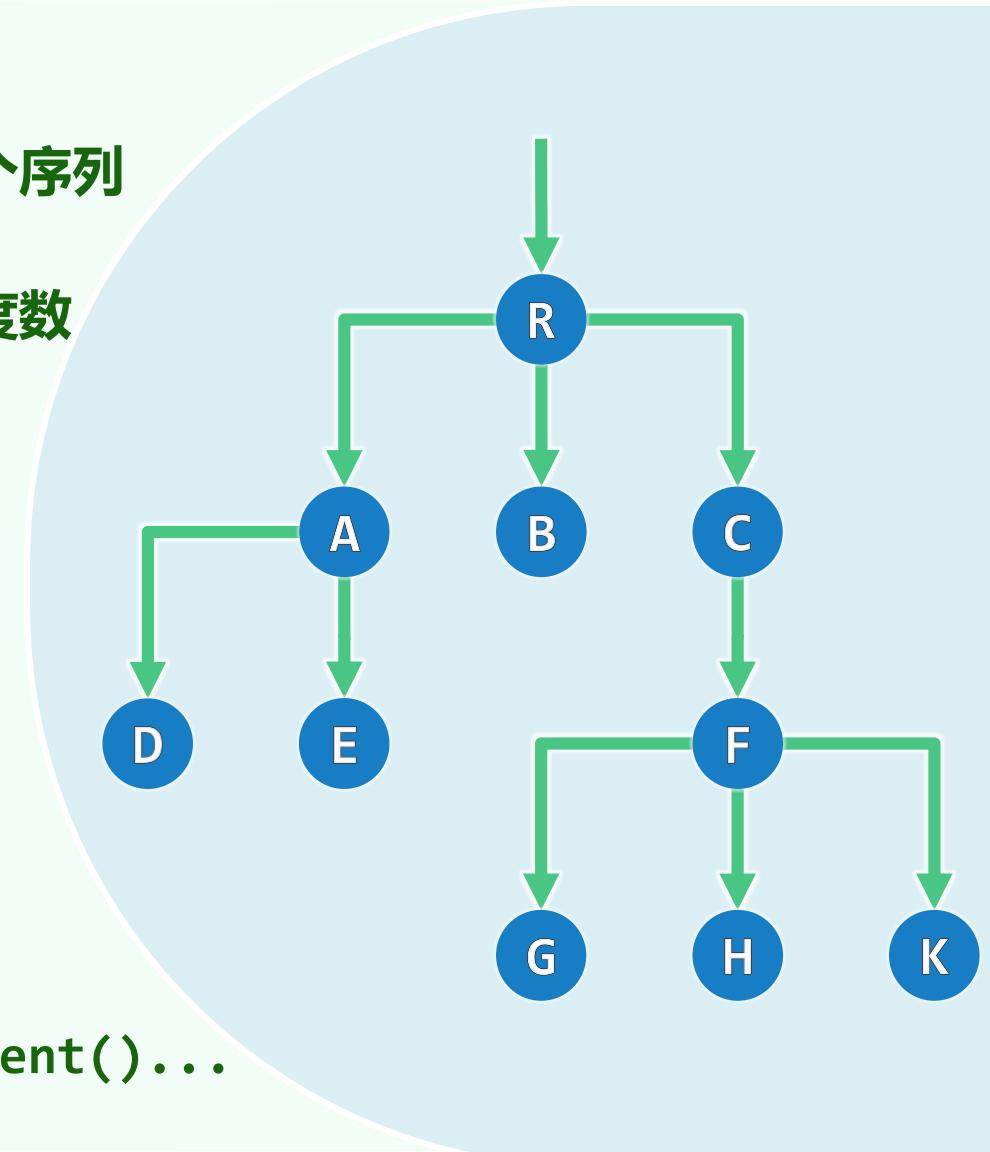
0 1 2 3 4 5 6 7 8 9

	data	children
0	H	^
1	E	^
2	F	^
3	B	0
4	R	3
5	K	6
6	D	2
7	A	
8	G	
9	C	

❖ 同一节点的所有孩子，各成一个序列

❖ 各序列的长度，即对应节点的度数

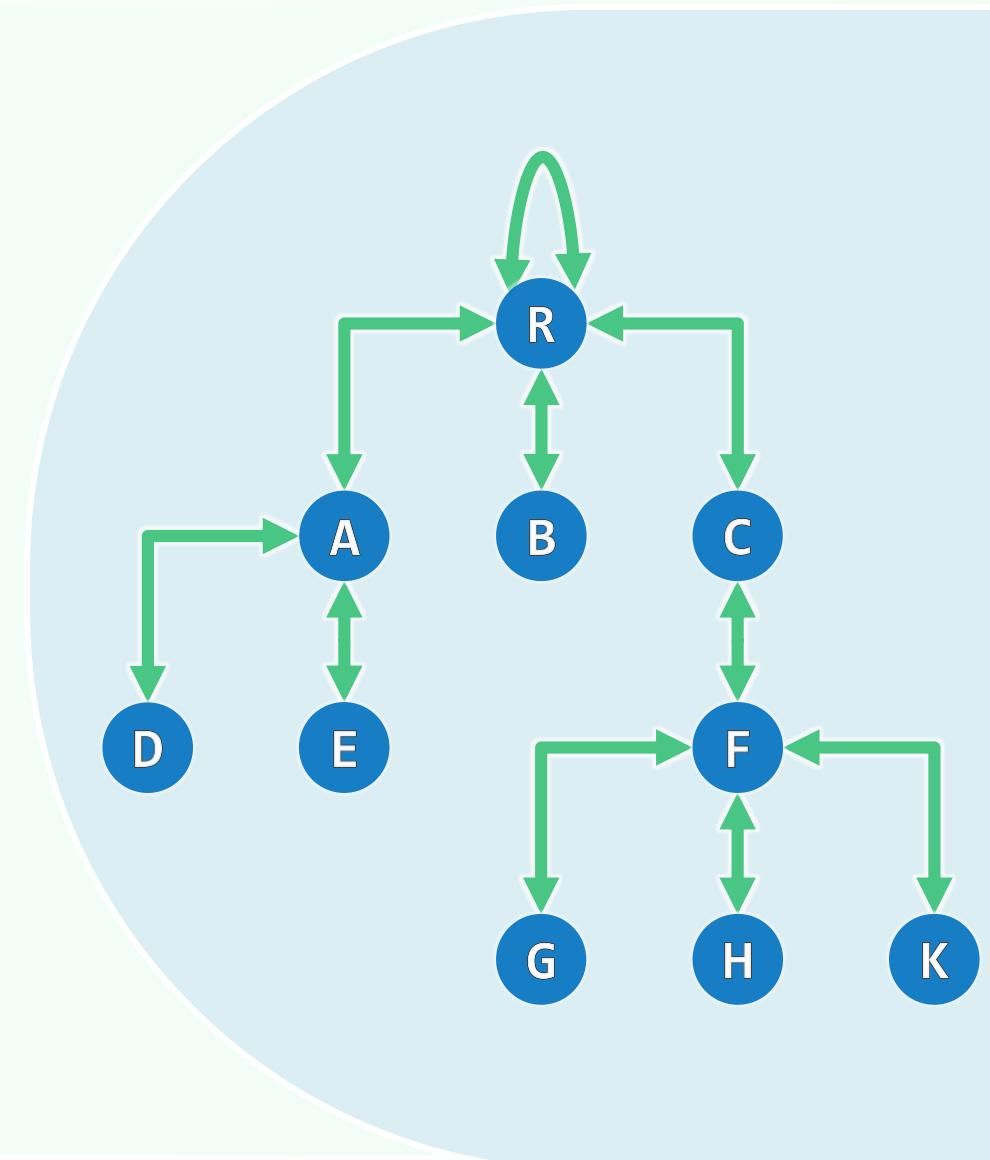
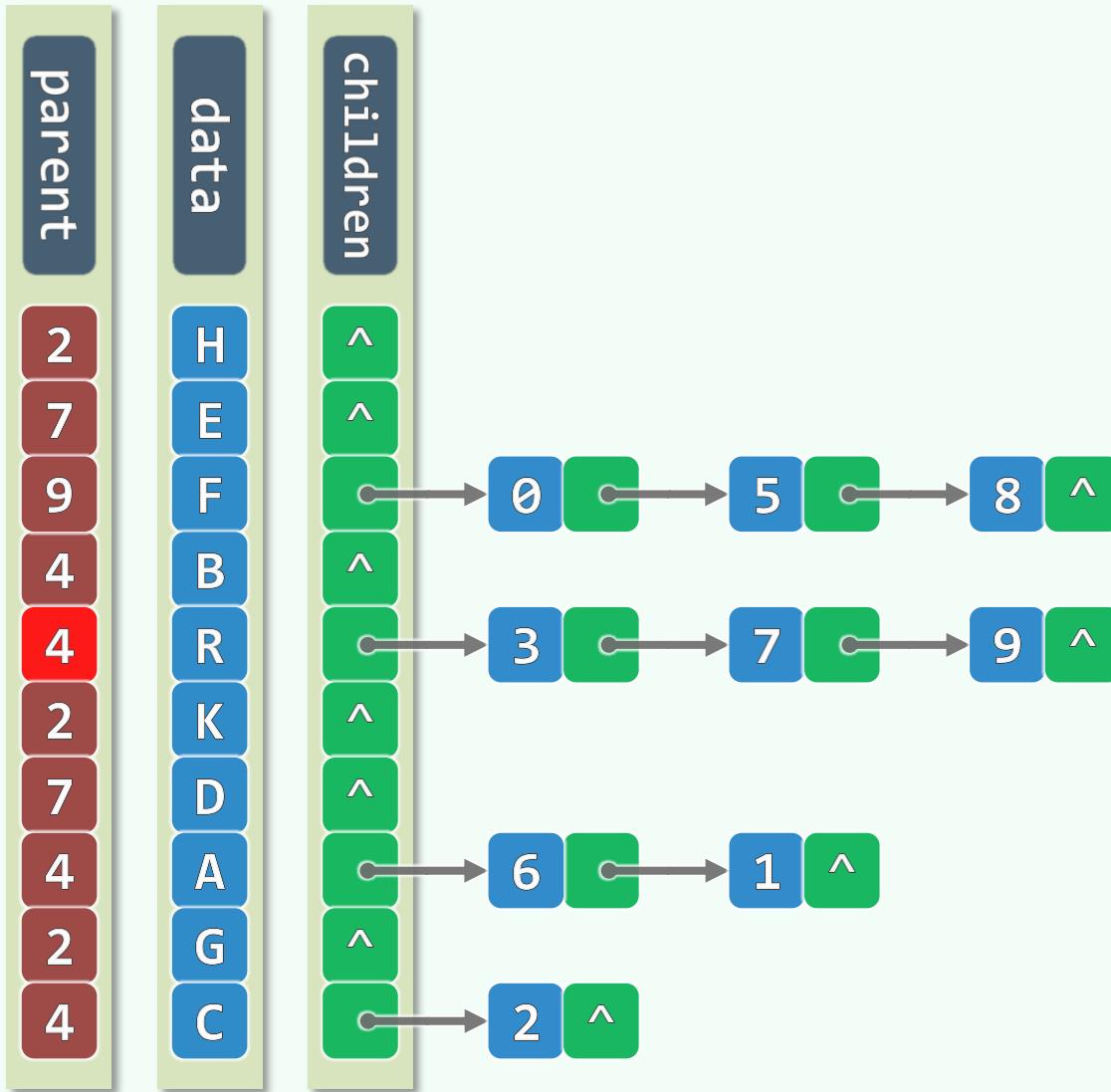
❖ 查找孩子很快，但parent(...)



父节点 + 孩子节点

rank

0 1 2 3 4 5 6 7 8 9



当地平线消失
躯体保持水平
大地保持水平
但别的一切
都垂直



宝玉终是不安本分之人，竟一味的随心所欲，因此又发了癖性，又特向秦钟悄说道：
“咱们俩个人一样的年纪，况又是同窗，以后不必论叔侄，只论弟兄朋友就是了。”

二叉树

有根有序树 = 二叉树

邓俊辉

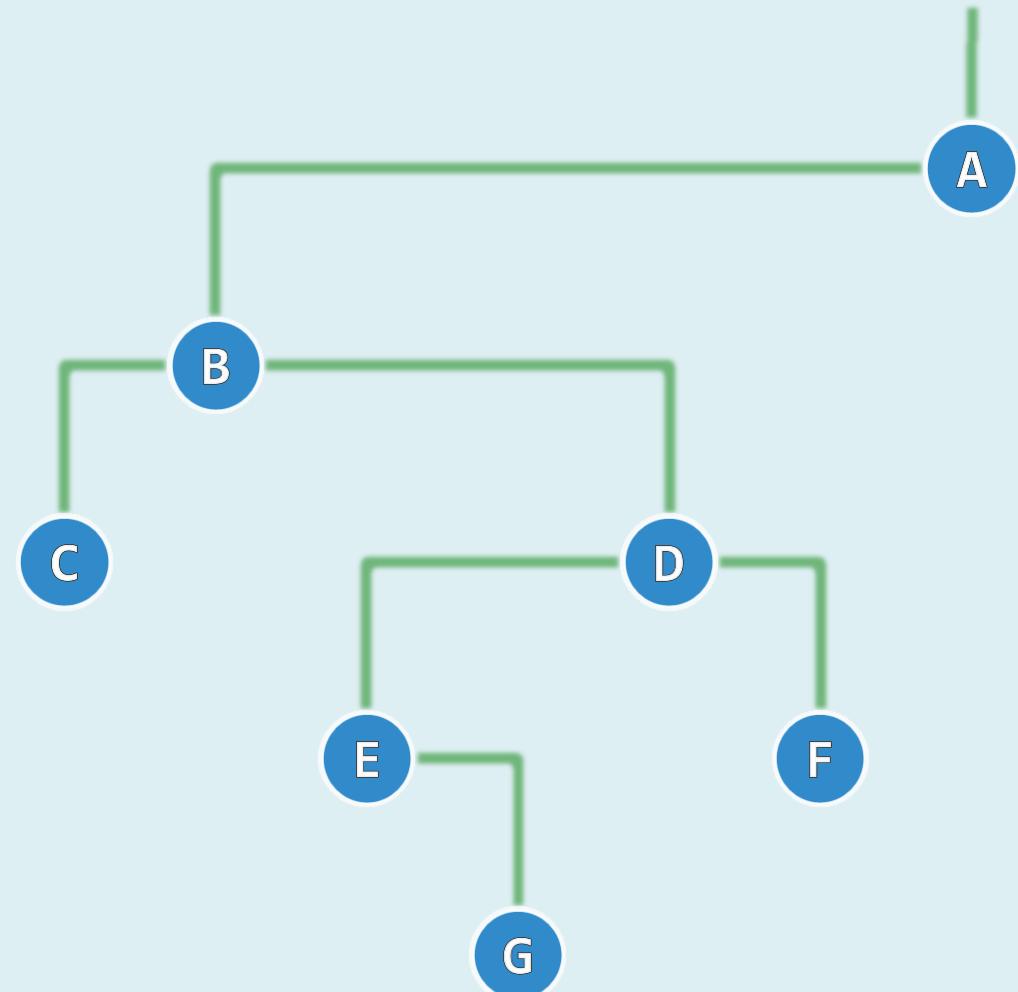
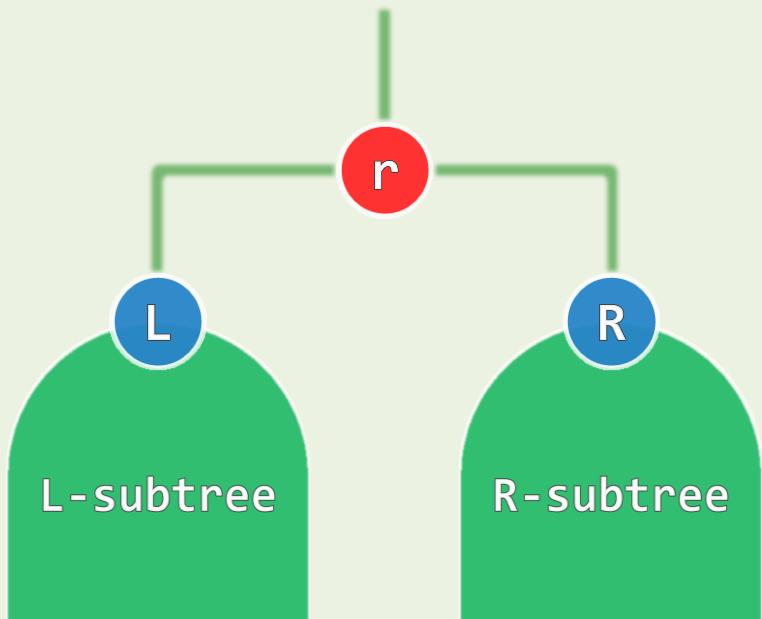
deng@tsinghua.edu.cn

二叉树

❖ Binary Tree: 节点度数不超过2

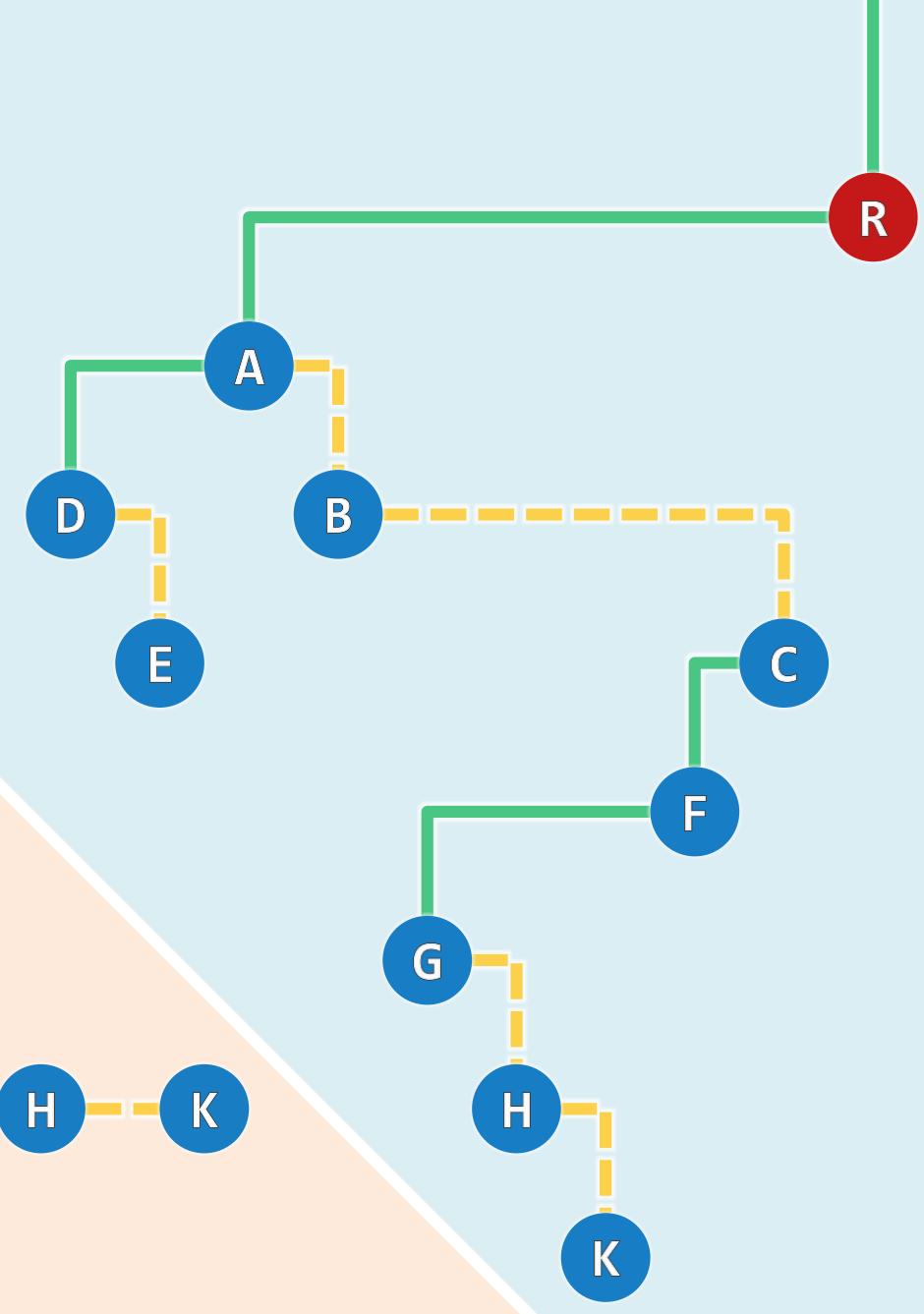
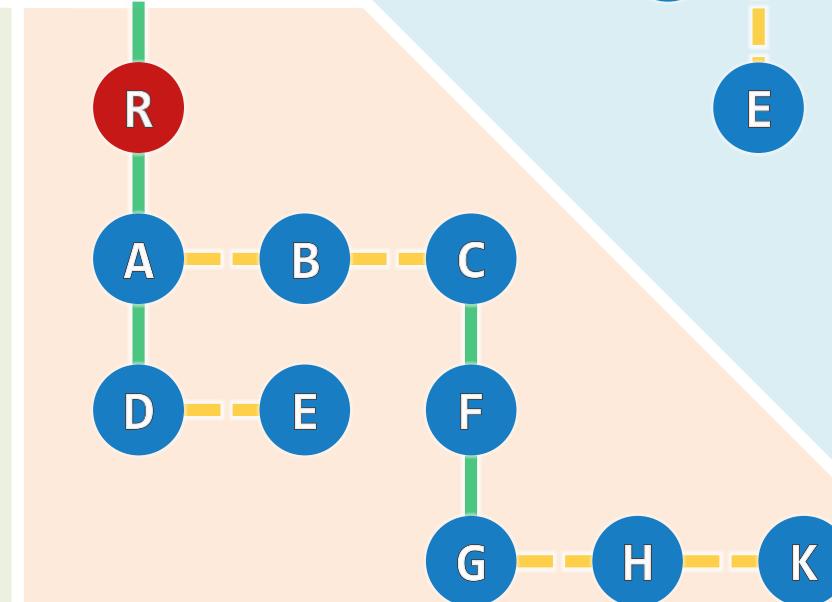
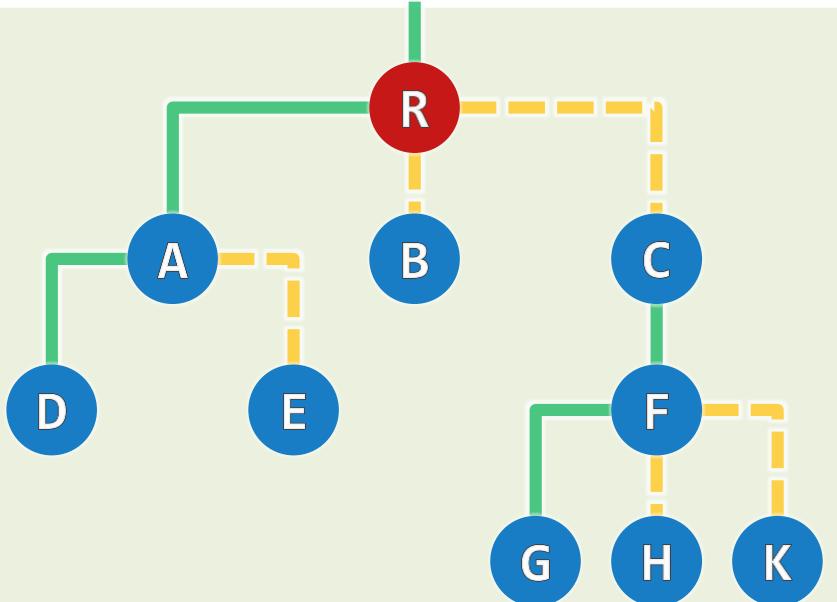
孩子 (子树) 可以左、右区分 (隐含有序)

- `lc()` ~ `lSubtree()`
- `rc()` ~ `rSubtree()`



描述多叉树：长子-兄弟表示法

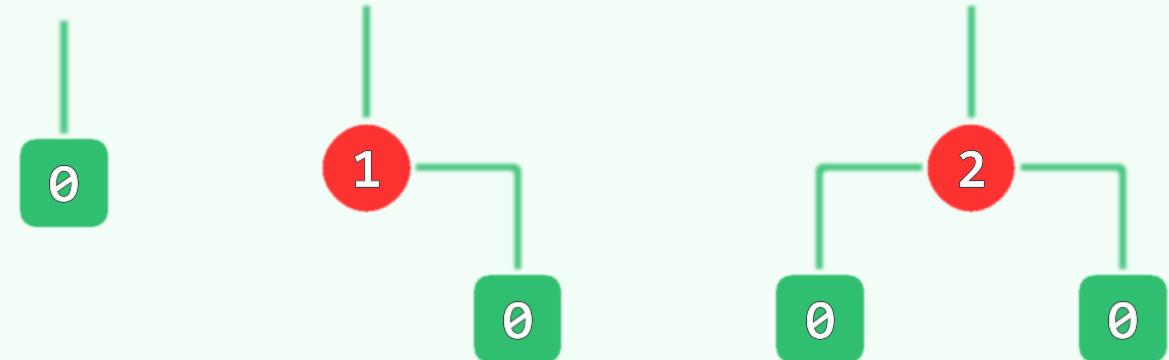
- ❖ 有根且有序的多叉树，均可转化并表示为二叉树
- ❖ 长子 ~ 左孩子 `firstChild()` ~ `lc()`
- 兄弟 ~ 右孩子 `nextSibling()` ~ `rc()`



基数：设度数为0、1和2的节点，各有 n_0 、 n_1 和 n_2 个

❖ 边数 $e = n - 1 = n_1 + 2n_2$

1/2度节点各对应于1/2条入边



❖ 叶节点数 $n_0 = n_2 + 1$

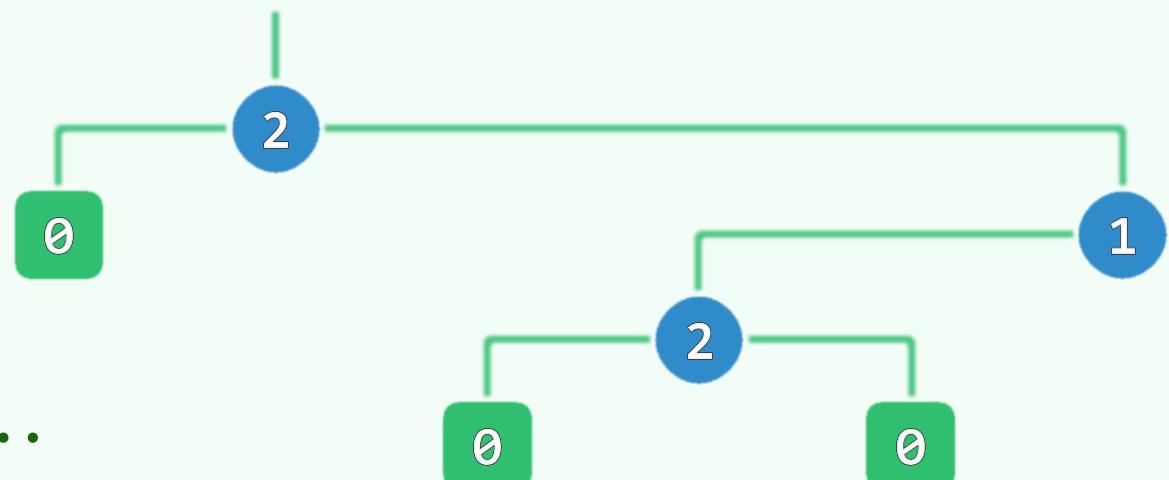
n_1 与 n_0 无关： $h = 0$ 时， $1 = 0 + 1$ ；此后， n_0 与随 n_2 同步递增

❖ 节点数 $n = n_0 + n_1 + n_2 = 1 + n_1 + 2n_2$

❖ 特别地，当 $n_1 = 0$ 时，有

$$e = 2n_2 \text{ 和 } n_0 = n_2 + 1 = (n + 1)/2$$

此时，节点度数均为偶数，不含单分支节点...

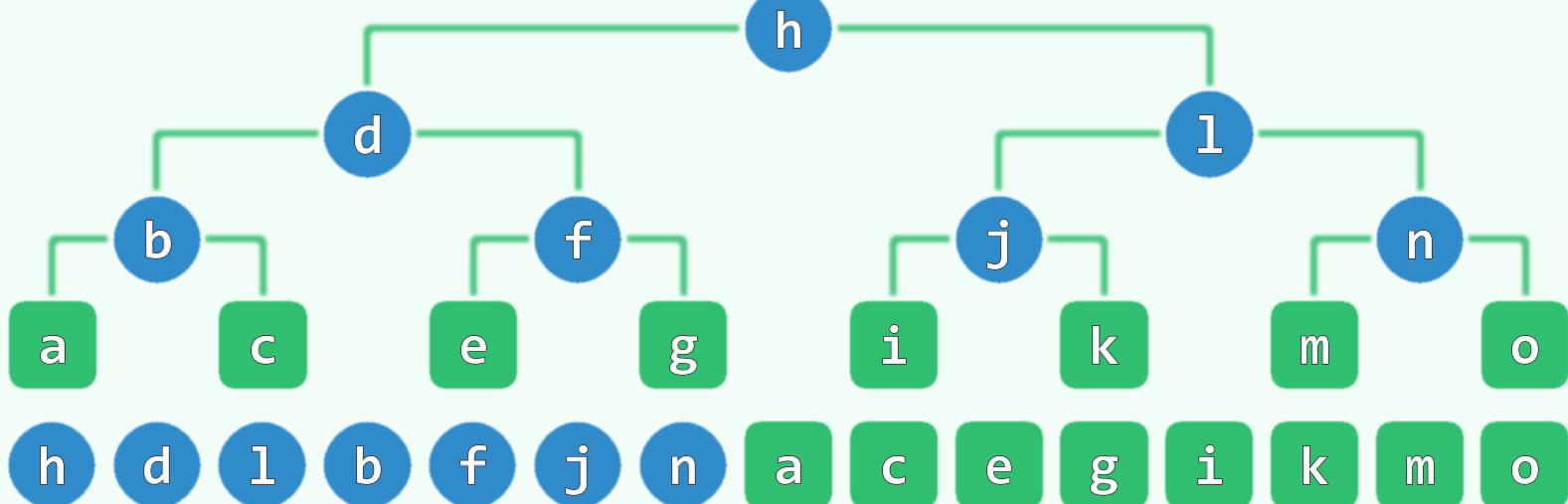


满树

❖ 深度为 k 的节点，至多 2^k 个

❖ n 个节点、高 h 的二叉树满足

$$h + 1 \leq n \leq 2^{h+1} - 1$$

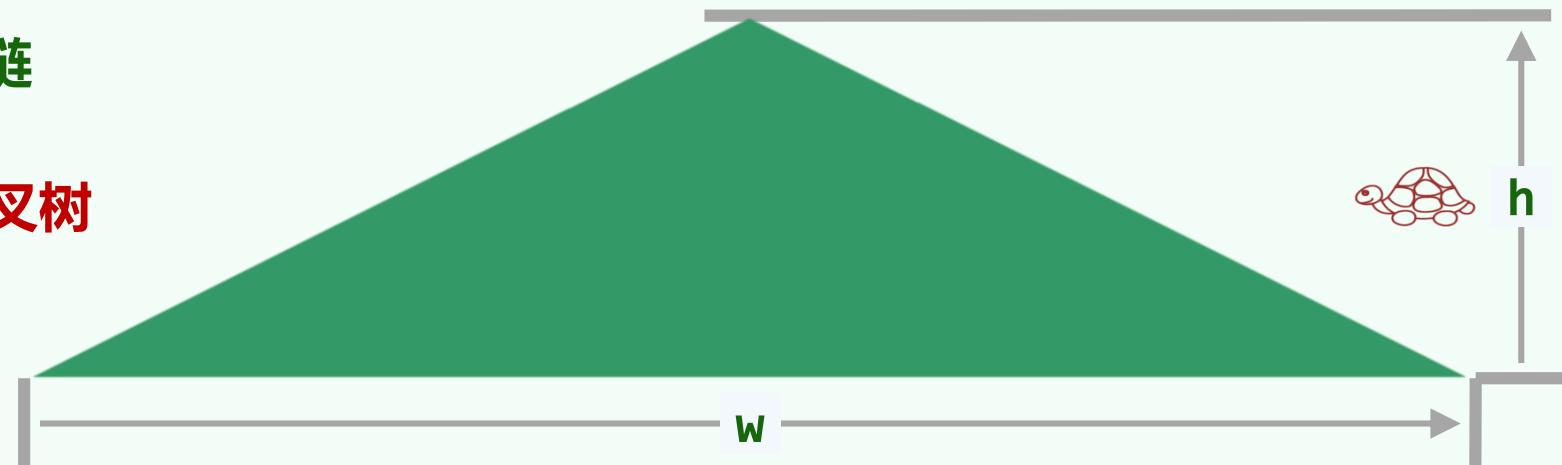


❖ 特殊情况

- $n = h + 1$: 退化为一条单链

- $n = 2^{h+1} - 1$: 即所谓满二叉树

full binary tree



真二叉树

❖ 通过引入 $n_1 + 2n_0$ 个外部节点

可使原有节点度数统一为 2

❖ 如此，即可将任一二叉树

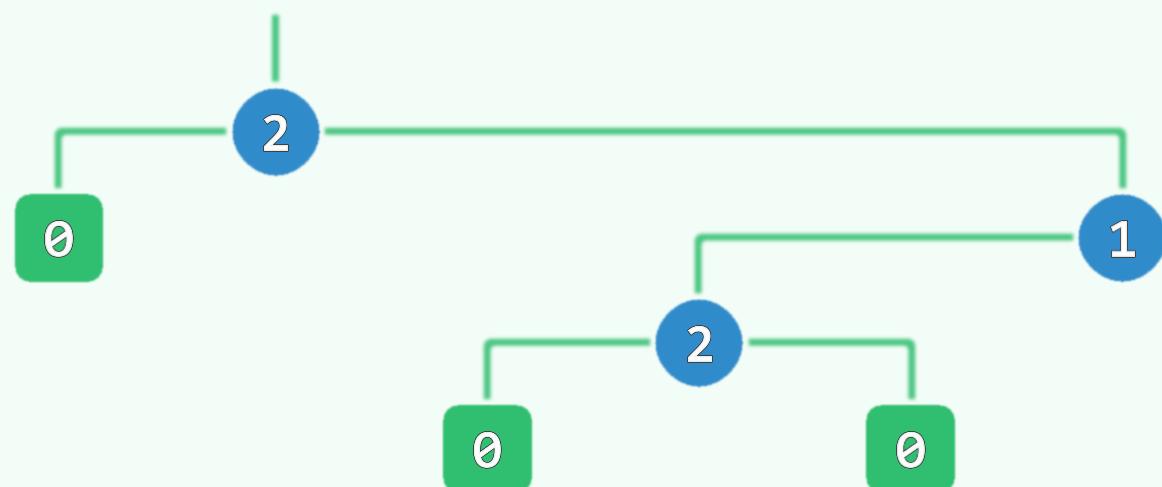
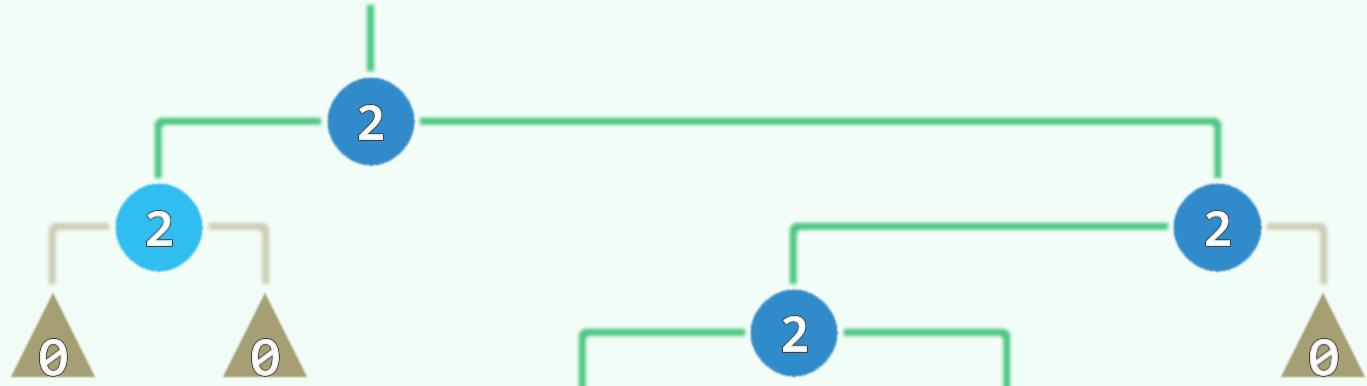
转化为真二叉树 (proper binary tree)

❖ 验证：如此转换之后，

全树自身的复杂度并未实质增加

❖ 对于红黑树之类的结构，真二叉树

可以简化描述、理解、实现和分析



e5 - D

Anyone who loves his father or mother more than me is not worthy of me; anyone who loves his son or daughter more than me is not worthy of me.

二叉树

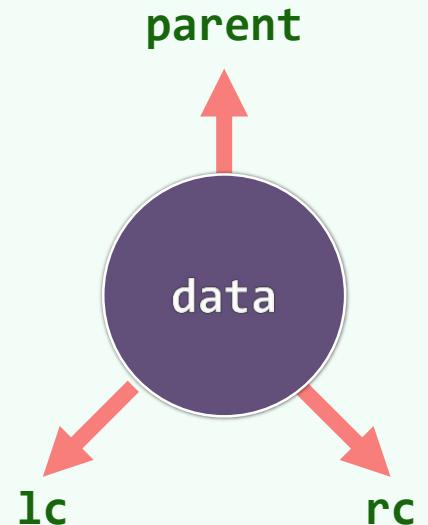
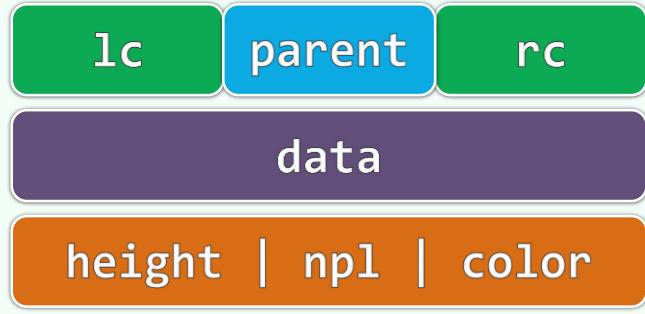
二叉树实现

邓俊辉

deng@tsinghua.edu.cn

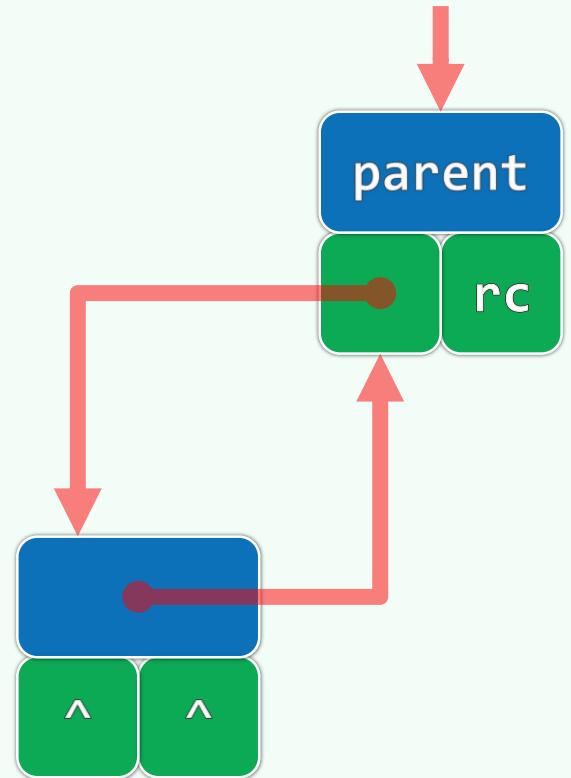
BinNode模板类

```
template <typename T> using BinNodePosi = BinNode<T>*; //节点位置  
  
template <typename T> struct BinNode {  
  
    BinNodePosi<T> parent, lc, rc; //父亲、孩子  
    T data; Rank height, npl; Rank size(); //高度、npl、子树规模  
  
    BinNodePosi<T> insertAsLC( T const & ); //作为左孩子插入新节点  
    BinNodePosi<T> insertAsRC( T const & ); //作为右孩子插入新节点  
    BinNodePosi<T> succ(); // (中序遍历意义下) 当前节点的直接后继  
  
    template <typename VST> void travLevel( VST & ); //层次遍历  
    template <typename VST> void travPre( VST & ); //先序遍历  
    template <typename VST> void travIn( VST & ); //中序遍历  
    template <typename VST> void travPost( VST & ); //后序遍历  
  
};
```



BinNode: 引入新节点

```
template <typename T>  
  
BinNodePosi<T> BinNode<T>::insertAsLC( T const & e )  
  
{ return lc = new BinNode( e, this ); }  
  
  
template <typename T>  
  
BinNodePosi<T> BinNode<T>::insertAsRC( T const & e )  
  
{ return rc = new BinNode( e, this ); }
```

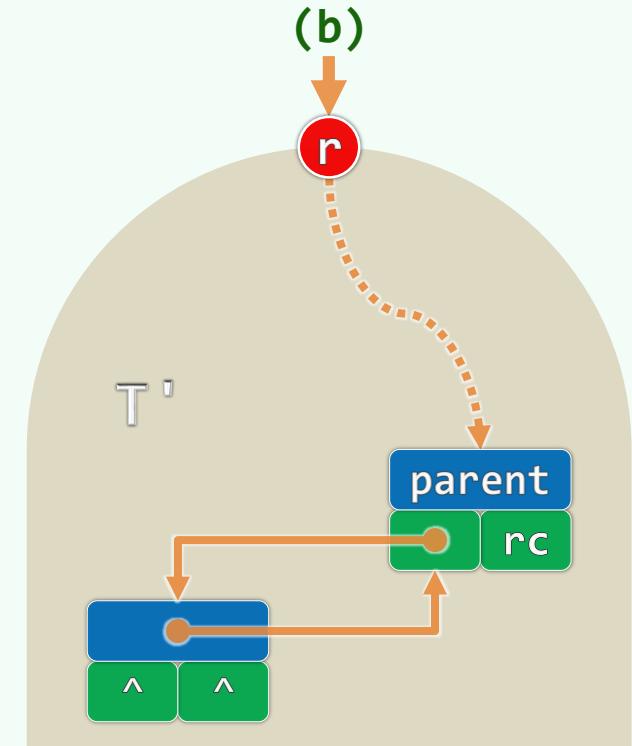
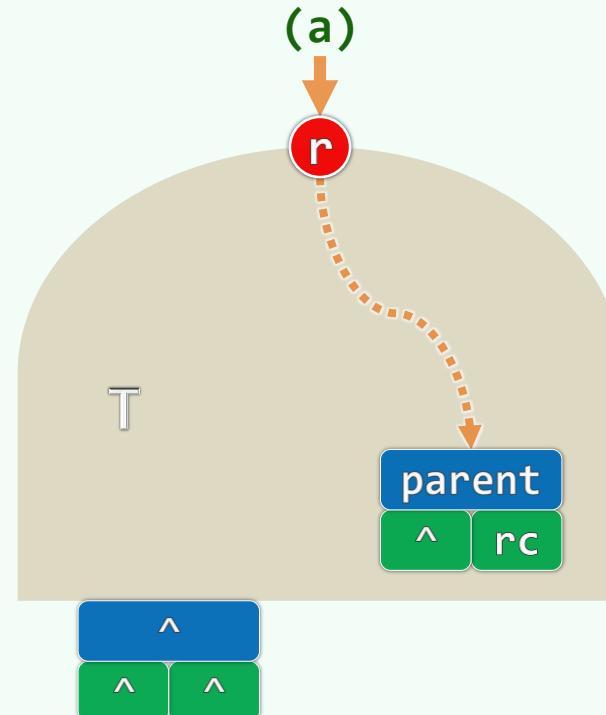


BinTree模板类

```
template <typename T> class BinTree {  
  
protected: Rank _size; //规模  
  
    BinNodePosi<T> _root; //根节点  
  
    virtual Rank updateHeight( BinNodePosi<T> x ); //更新节点x的高度  
    void updateHeightAbove( BinNodePosi<T> x ); //更新x及祖先的高度  
  
public: Rank size() const { return _size; } //规模  
    bool empty() const { return !_root; } //判空  
    BinNodePosi<T> root() const { return _root; } //树根  
    /* ... 子树接入、删除和分离接口；遍历接口 ... */  
}
```

BinTree: 引入新节点

```
BinNodePosi<T> BinTree<T>::insert( BinNodePosi<T> x, T const & e ); //作为右孩子  
  
BinNodePosi<T> BinTree<T>::insert( T const & e, BinNodePosi<T> x ) { //作为左孩子  
  
    _size++;  
  
    x->insertAsLC( e );  
  
    updateHeightAbove( x );  
  
    return x->lc;  
}
```

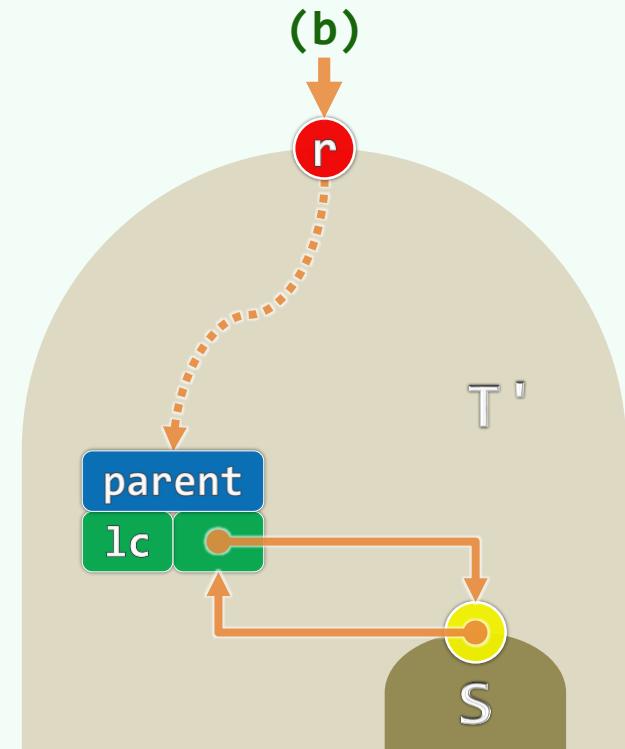
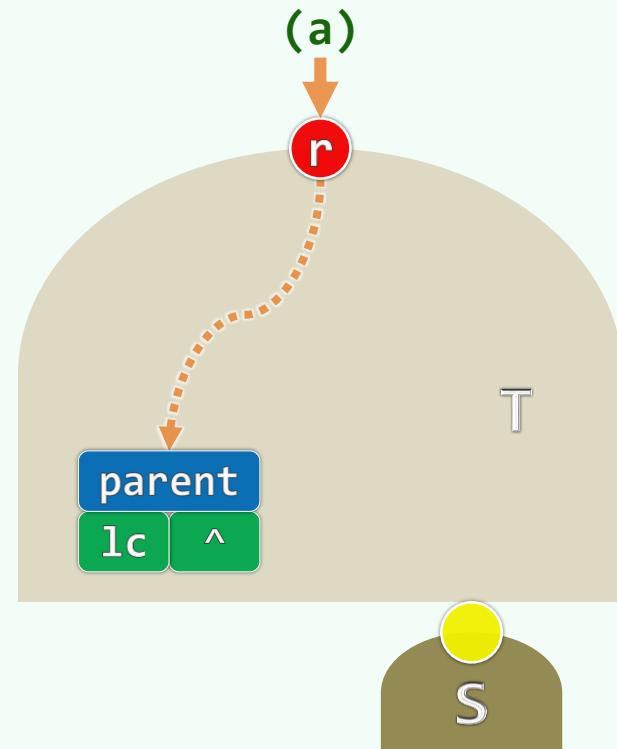


BinTree: 接入子树

```
BinNodePosi<T> BinTree<T>::attach( BinTree<T>* &S, BinNodePosi<T> x ); //接入左子树
```

```
BinNodePosi<T> BinTree<T>::attach( BinNodePosi<T> x, BinTree<T>* &S ) { //接入右子树
```

```
if ( x->rc = S->_root )  
    x->rc->parent = x;  
  
_size += S->_size;  
  
updateHeightAbove(x);  
  
S->_root = NULL; S->_size = 0;  
  
release(S); S = NULL;  
  
return x;
```



BinTree: 更新高度

```
#define stature(p) ( (int) (_(p) ? (p)->height : -1) ) //空树高度-1, 以上递推
```

```
template <typename T> //勤奋策略: 及时更新节点x高度, 具体规则因树不同而异
```

```
Rank BinTree<T>::updateHeight( BinNodePosi<T> x ) //此处采用常规二叉树规则,  $\mathcal{O}(1)$   
{ return x->height = 1 + max( stature( x->lC ), stature( x->rC ) ); }
```

```
template <typename T> //更新节点及其历代祖先的高度
```

```
void BinTree<T>::updateHeightAbove( BinNodePosi<T> x ) // $\mathcal{O}(n = \text{depth}(x))$   
{ while (x) { updateHeight(x); x = x->parent; } } //可优化
```

BinTree: 分离子树

```
template <typename T> BinTree<T>* BinTree<T>::secede( BinNodePosi<T> x ) {  
  
    FromParentTo( * x ) = NULL; updateHeightAbove( x->parent );  
  
    // 以上与BinTree<T>::remove()一致; 以下还需对分离出来的子树重新封装  
  
    BinTree<T> * S = new BinTree<T>; //创建空树  
  
    S->_root = x; x->parent = NULL; //新树以x为根  
  
    S->_size = x->size(); _size -= S->_size; //更新规模  
  
    return S; //返回封装后的子树  
}
```

二叉树

先序遍历：观察

真君曰：“昔吕洞宾居庐山而成仙，鬼谷子居云梦而得道，今或无此吉地么？”璞曰：“有，但当遍历耳。”

一桩事情的真相与奥妙，通常并不藏在最深的地方，有时就在表面。只不过，一般人视若无睹。要想成为一个好的算命先生，首先就必须学会观察...

警幻冷笑道：“贵省女子固多，不过择其紧要者录之。下边二厨则又次之。余者庸常之辈，则无册可录矣。”宝玉听说，再看下首二厨上，果然写着“金陵十二钗副册”，又一个写着“金陵十二钗又副册”。



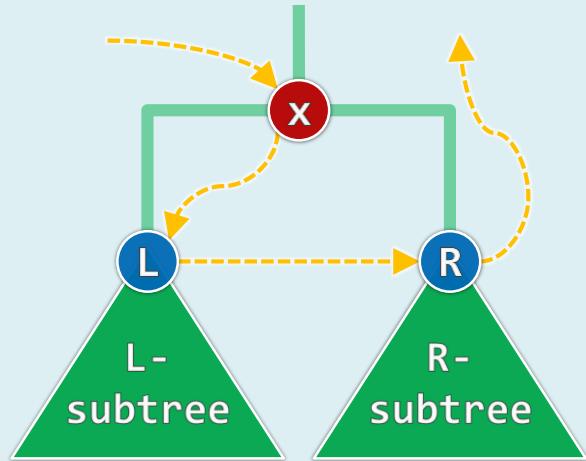
邓俊辉

deng@tsinghua.edu.cn

遍历：按照某种次序访问树中各节点，每个节点被访问恰好一次

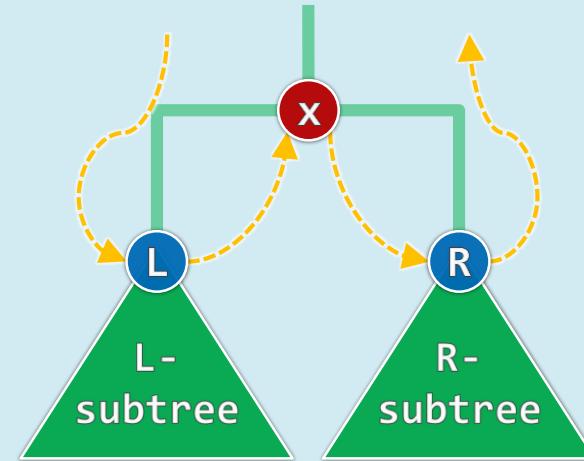
❖ $T = L \cup X \cup R$

先序
preorder



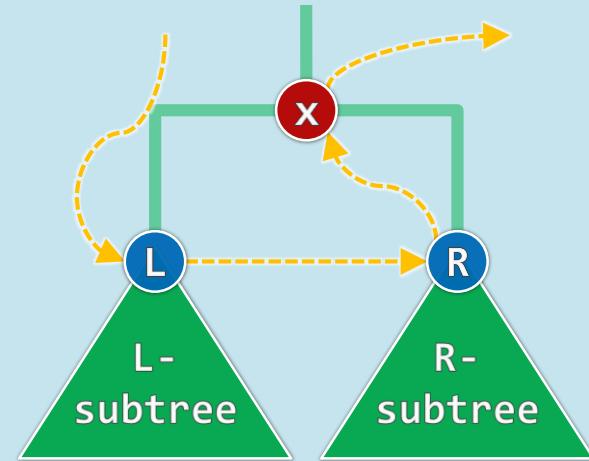
$X | L | R$

中序
inorder



$L | X | R$

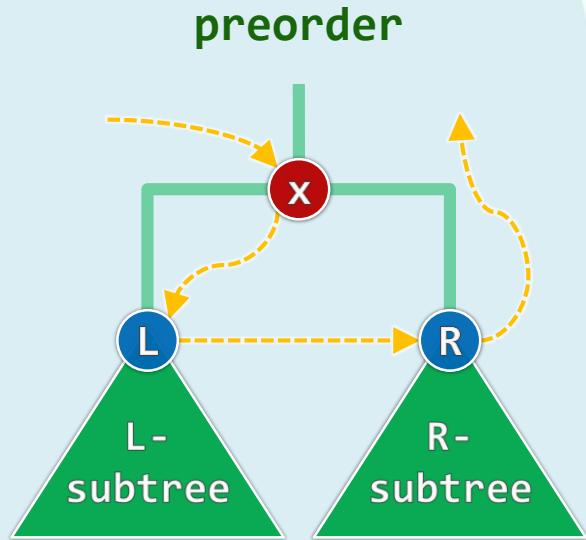
后序
postorder



$L | R | X$

❖ 遍历：结果 ~ 过程 ~ 次序 ~ 策略

先序实例：统计规模



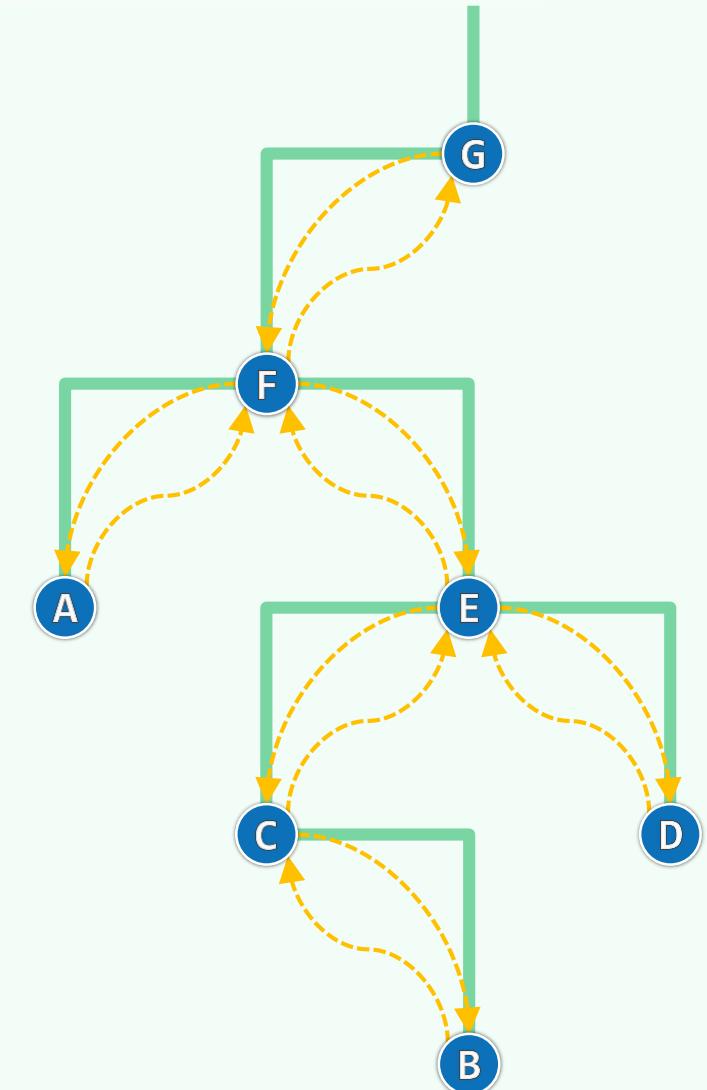
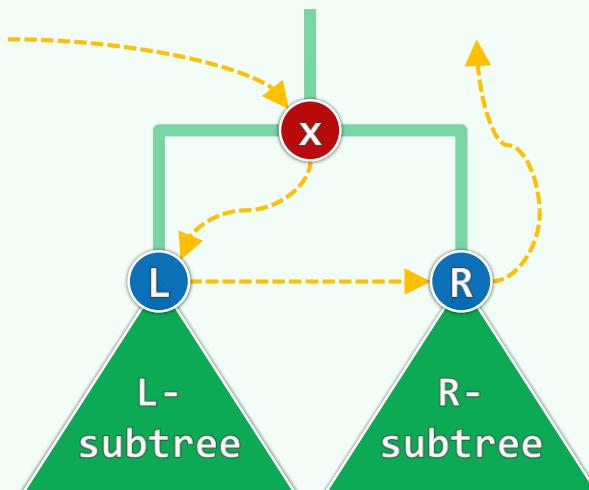
```
template <typename T>
Rank BinNode<T>::size() { //后代总数
    Rank s = 1; //计入本身
    if (lc) s += lc->size(); //递归计入左子树规模
    if (rc) s += rc->size(); //递归计入右子树规模
    return s;
} //懒惰策略,  $\mathcal{O}(n = |size|)$ 
```

递归实现

❖ 应用：先序输出文件树结构：`c:\> tree.com c:\windows`

❖ template <typename T, typename VST>

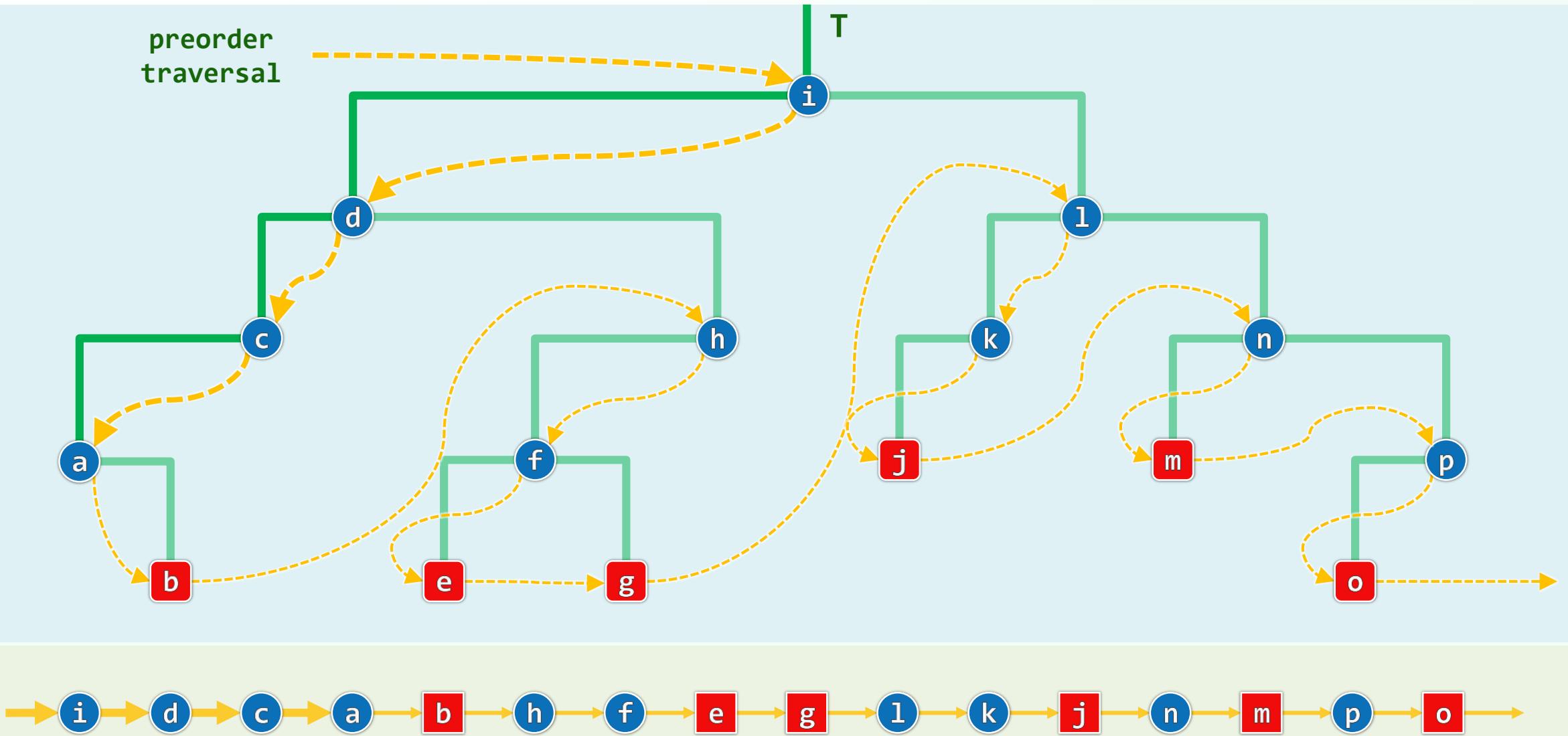
```
void traverse( BinNodePosi<T> x, VST & visit ) {  
    if ( ! x ) return;  
  
    visit( x->data );  
  
    traverse( x->l, visit );  
    traverse( x->r, visit );  
} //O(n)
```



❖ 制约：使用默认的Call Stack，允许的递归深度有限

❖ 挑战：不依赖递归机制，能否实现先序遍历？如何实现？效率如何？

观察



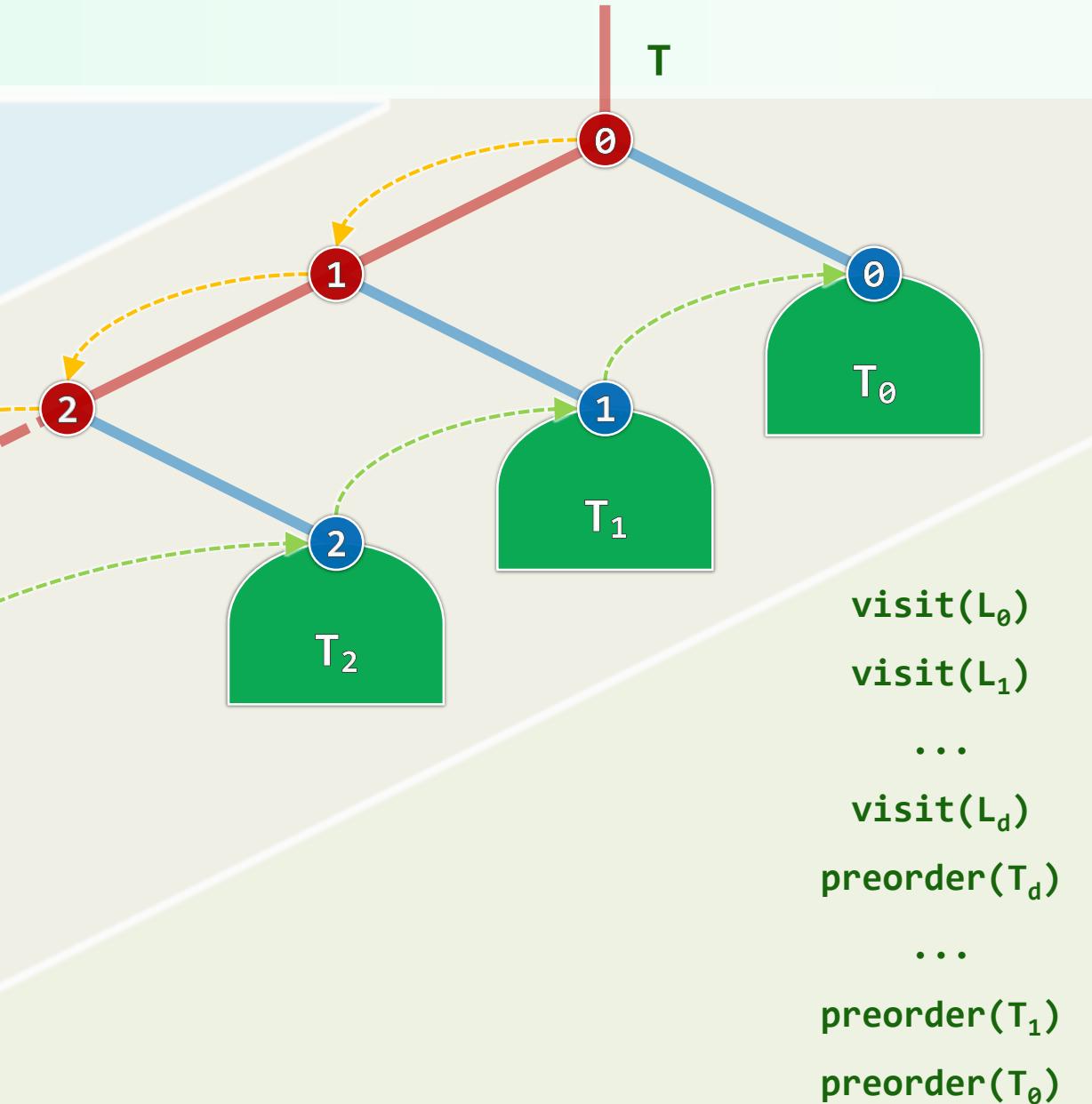
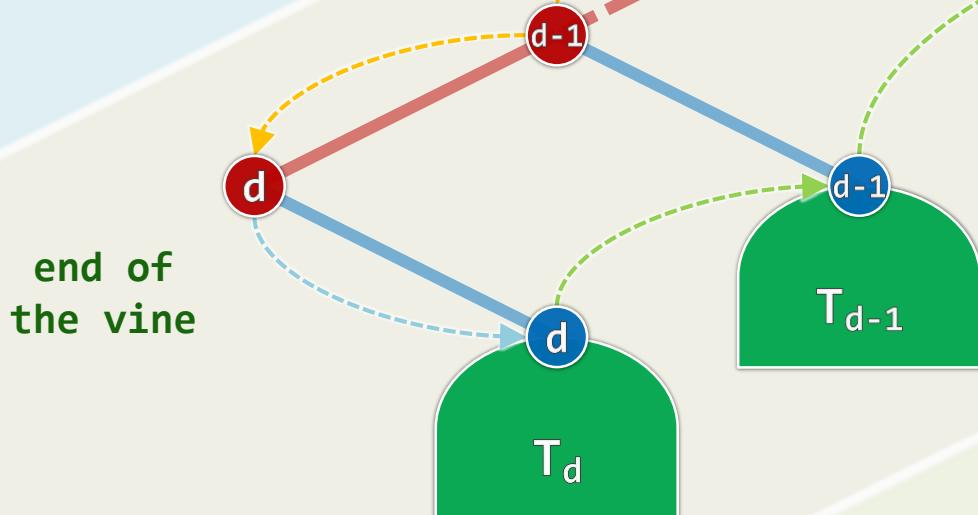
藤缠树

沿着左侧藤，整个遍历过程可分解为：

- 自上而下访问藤上节点，再
- 自下而上遍历各右子树

各右子树的遍历彼此独立

自成一个子任务



二叉树

先序遍历：迭代算法

05 - E2

邓俊辉

deng@tsinghua.edu.cn

凡是过往，皆为序章

我是如来最小之弟

序曲

```
template <typename T, typename VST> static void visitAlongVine
( BinNodePosi<T> x, VST & visit, Stack < BinNodePosi<T> > & S ) { //分摊 $O(1)$ 
    while ( x ) { //反复地
        visit( x->data ); //访问当前节点
        S.push( x->rc ); //右孩子 (右子树) 入栈 (将来逆序出栈)
        x = x->lc; //沿藤下行
    } //只有右孩子、NULL可能入栈—增加判断以剔除后者，是否值得?
}
```

全曲

```
template <typename T, typename VST>

void travPre_I2( BinNodePosi<T> x, VST & visit ) {

Stack < BinNodePosi<T> > S; //辅助栈

while ( true ) { //以右子树为单位，逐批访问节点

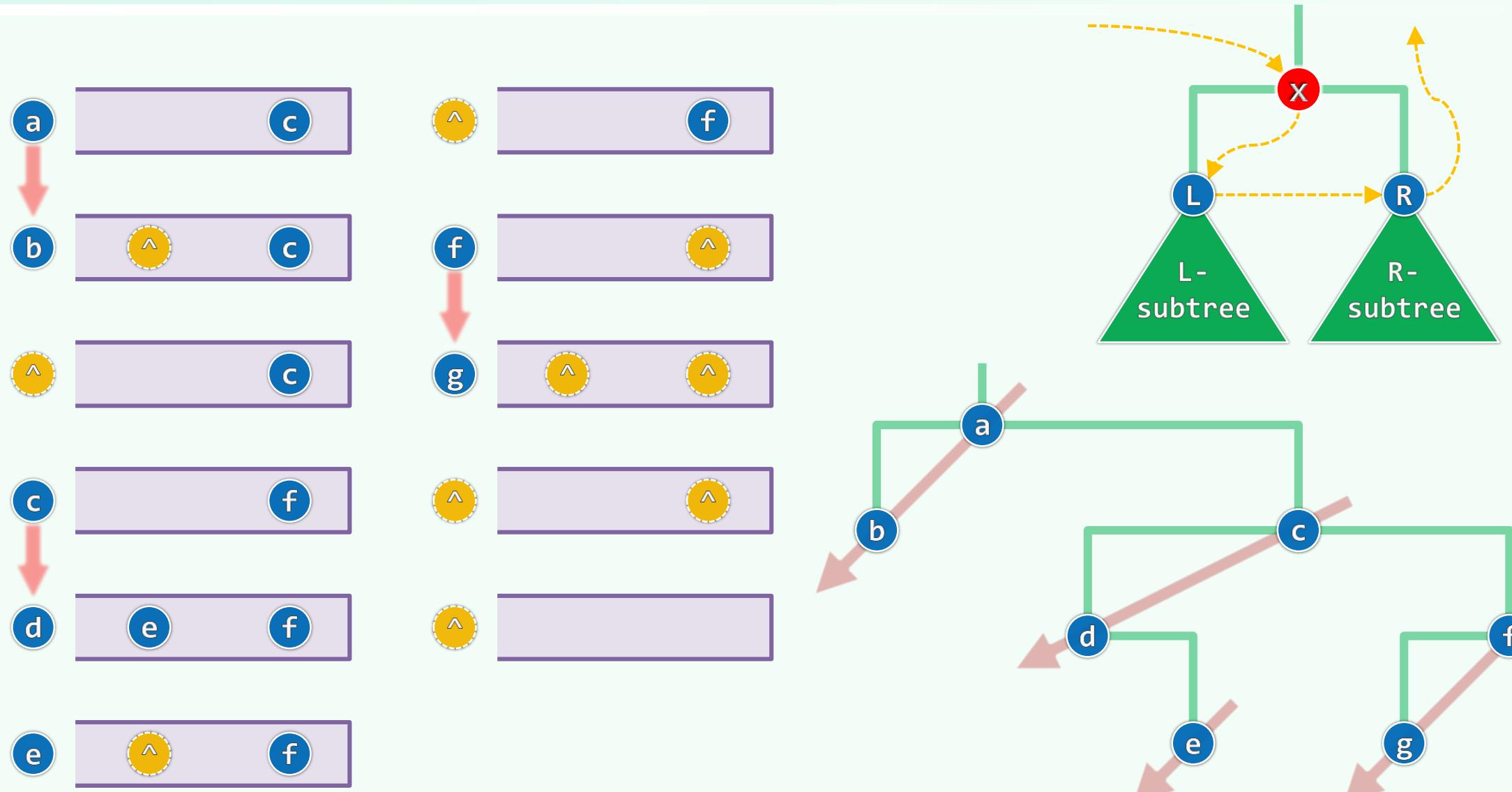
visitAlongVine( x, visit, S ); //访问子树x的藤蔓，各右子树（根）入栈缓冲

if ( S.empty() ) break; //栈空即退出

x = S.pop(); //弹出下一右子树（根）

} //#pop = #push = #visit =  $\Theta(n)$  = 分摊 $\Theta(1)$ 
```

实例



二叉树

中序遍历：观察

e5 - F1

山中只见藤缠树，世上哪见树缠藤
青藤若是不缠树，枉过一春又一春

邓俊辉

deng@tsinghua.edu.cn

递归实现

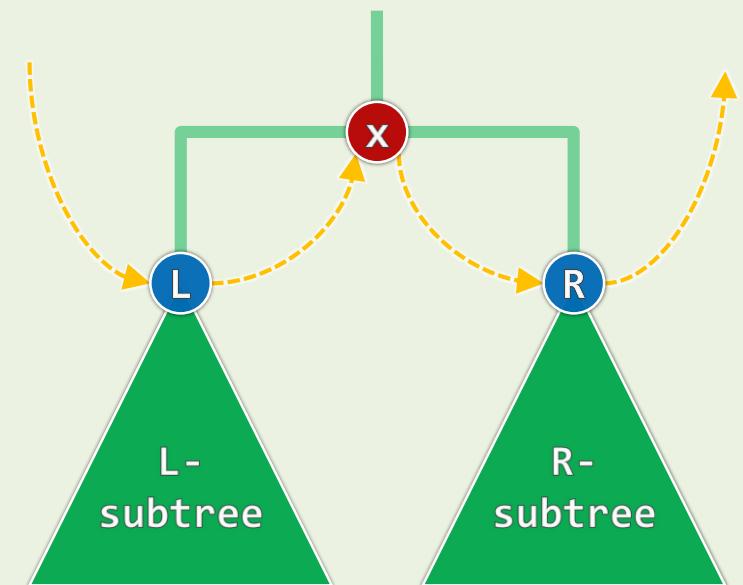
❖ 应用：中序输出文件树结构：printBinTree()

❖ template <typename T, typename VST>

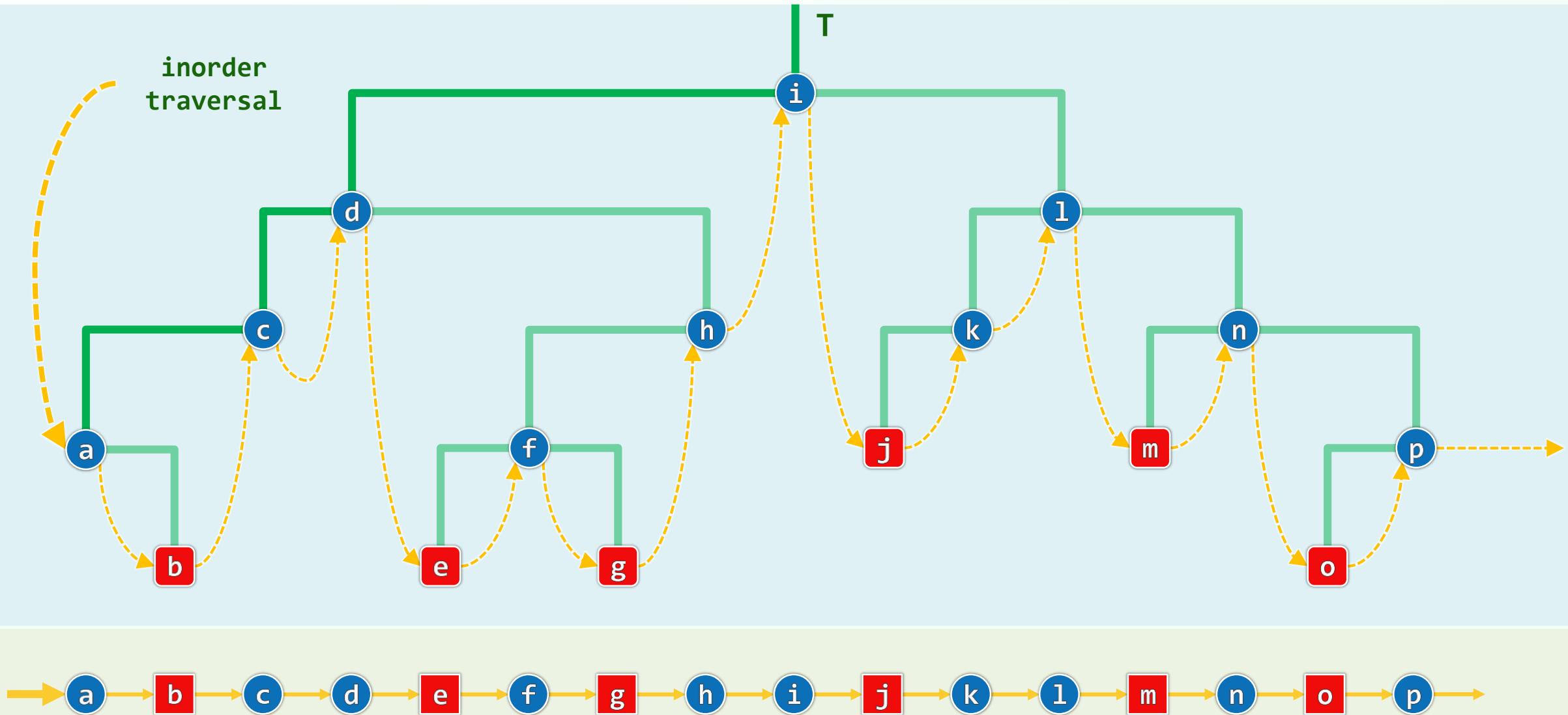
```
void traverse( BinNodePosi<T> x, VST & visit ) {  
    if ( !x ) return;  
  
    traverse( x->lc, visit );  
  
    visit( x->data );  
  
    traverse( x->rc, visit ); //tail  
}
```

❖ $T(n) = T(a) + \mathcal{O}(1) + T(n - a - 1) = \mathcal{O}(n)$

❖ 挑战：不依赖递归机制，如何实现中序遍历？效率如何？



观察



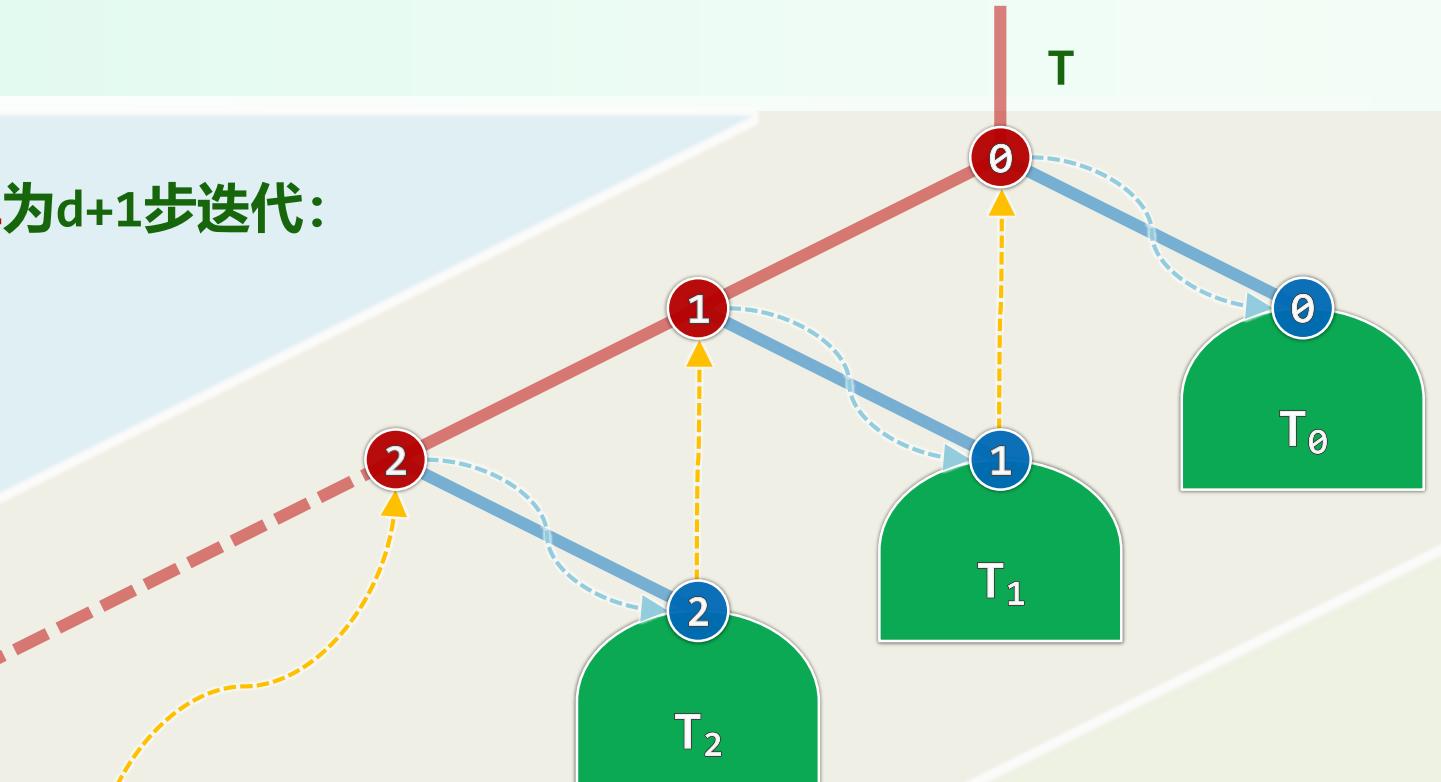
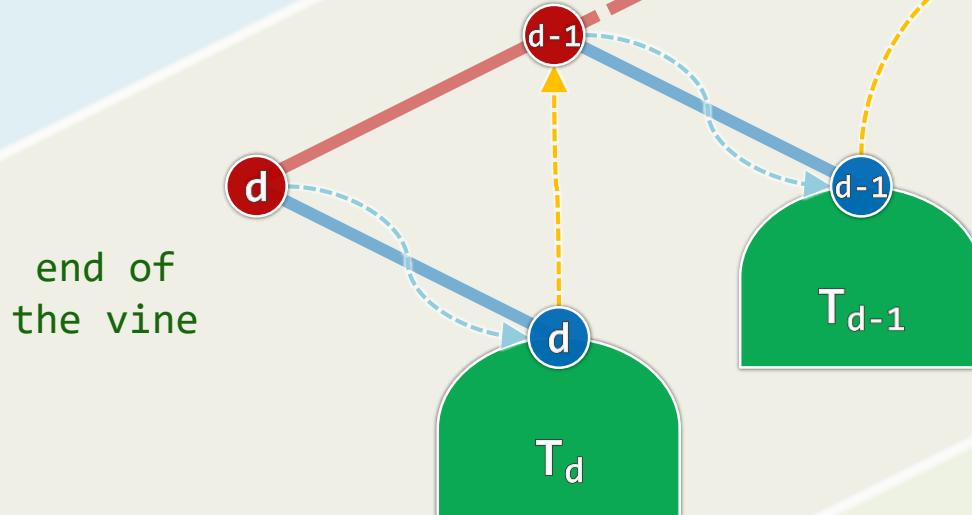
藤缠树

沿着左侧藤，遍历可自底而上分解为 $d+1$ 步迭代：

访问藤上节点，再遍历其右子树

各右子树的遍历彼此独立

自成一个子任务



$\text{visit}(L_d), \text{inorder}(T_d)$

$\text{visit}(L_{d-1}), \text{inorder}(T_{d-1})$

.....,

$\text{visit}(L_1), \text{inorder}(T_1)$

$\text{visit}(L_0), \text{inorder}(T_0)$

二叉树

中序遍历：迭代算法

e5 - F2

Although we've come to the end of the road

Still I can't let you go

邓俊辉

deng@tsinghua.edu.cn

序曲

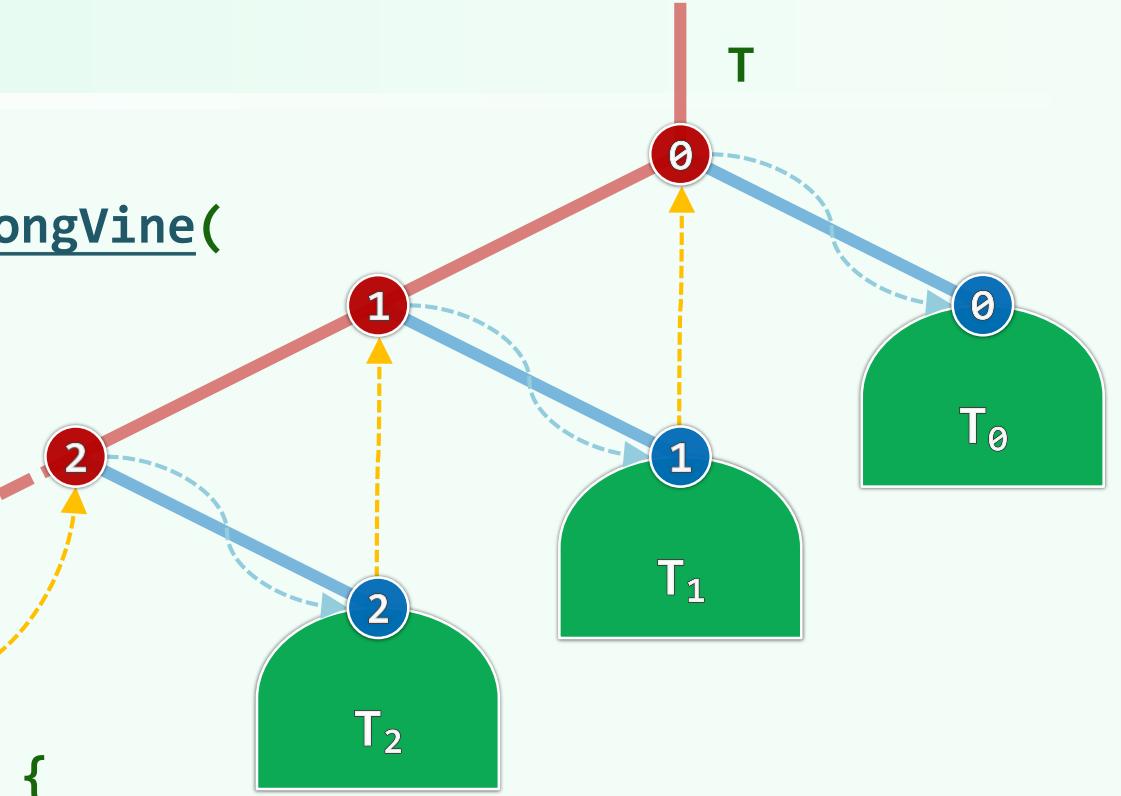
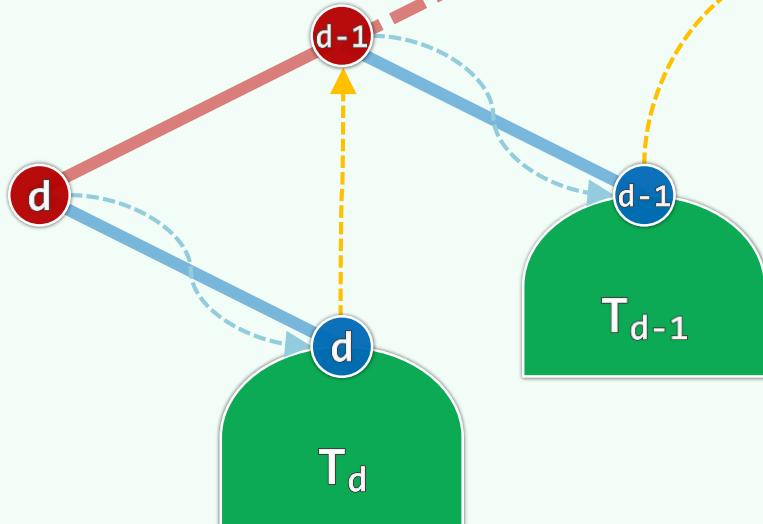
```
template <typename T> static void goAlongVine(
```

```
    BinNodePosi<T> x,
```

```
    Stack < BinNodePosi<T> > & S
```

```
)
```

end of
the vine



```
    while ( x )
```

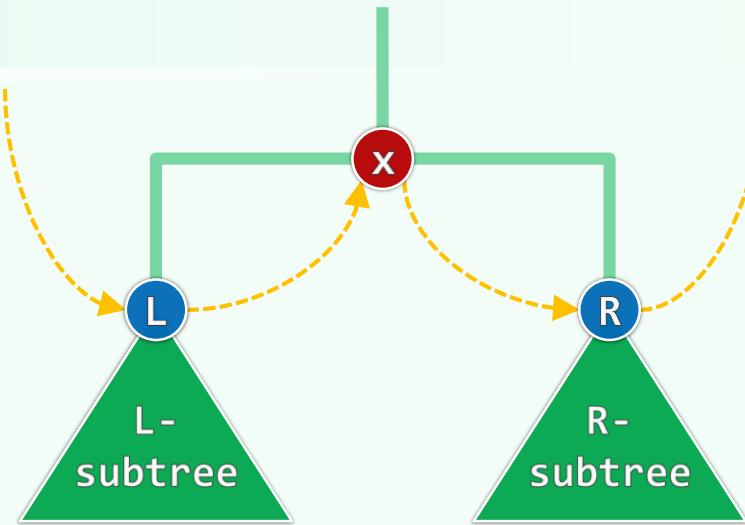
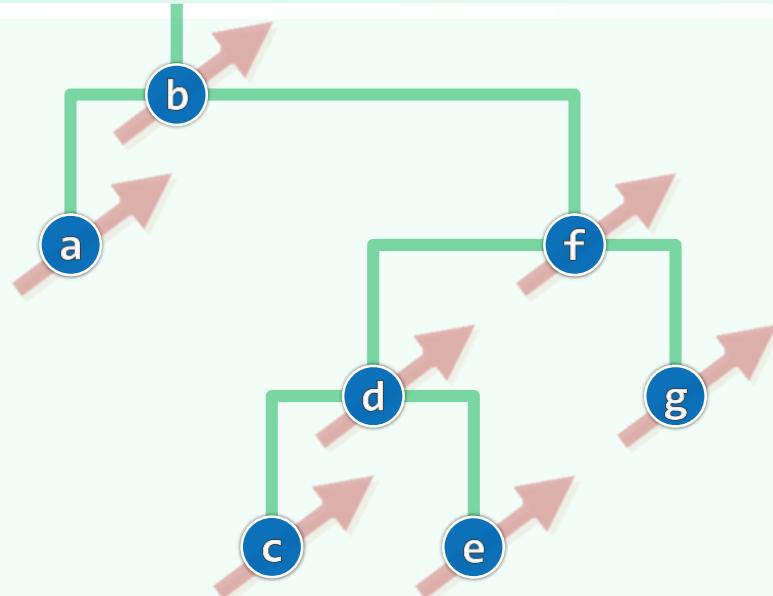
```
        { S.push( x ); x = x->lc; }
```

```
} //逐层深入，沿藤蔓各节点依次入栈
```

全曲

```
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  
        goAlongVine( 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->rc; //再转向其右子树（可能为空，留意处理手法）  
    }  
}
```

实例



a

b

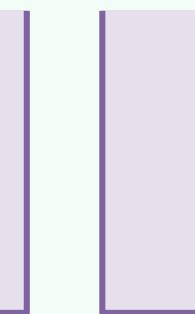
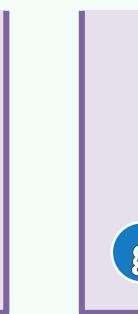
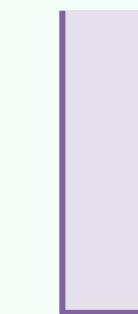
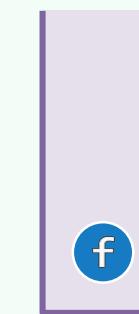
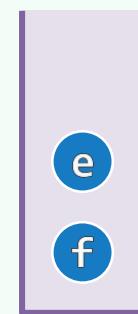
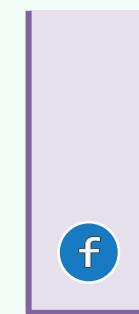
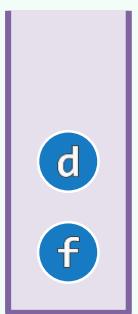
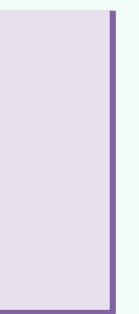
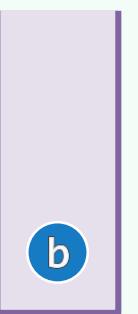
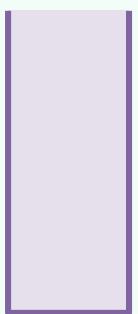
c

d

e

f

g



二叉树

中序遍历：分析

e5 - F3

邓俊辉

deng@tsinghua.edu.cn

他從哪條路來，必從哪條路回去，必不得來到這城

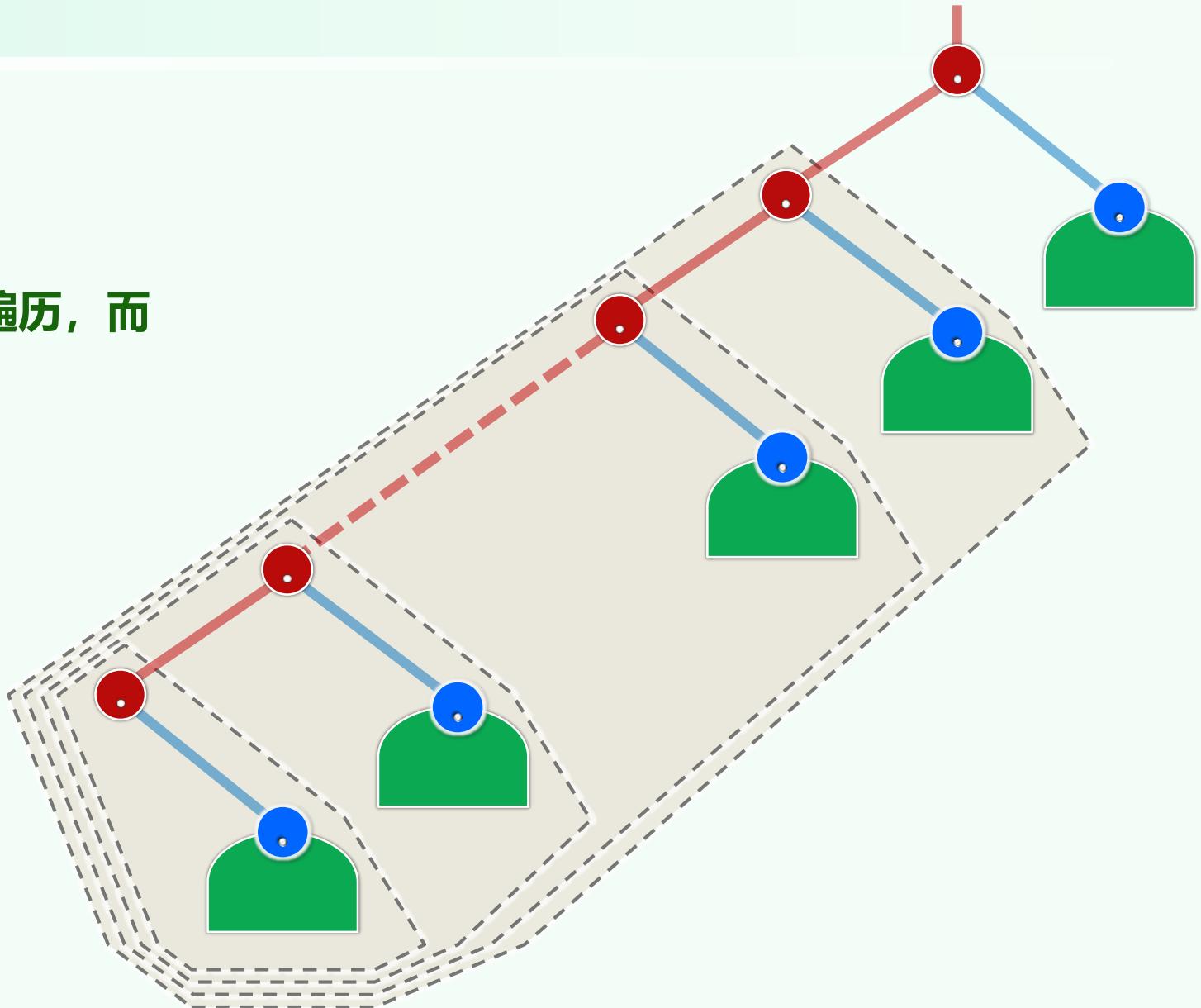
正确性：数学归纳

❖ 每个节点出栈时

- 其**左子树或不存在，或已完全遍历**，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：分摊分析

- ❖ goAlongVine()最多需调用 $\Omega(n)$ 次；单次调用最多需做 $\Omega(n)$ 次push()
- ❖ 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，更精准的分析可给出 $\mathcal{O}(n)$ 的界！
- ❖ 为此，需要纵观整个遍历过程中所有对goAlongVine()的调用...
- ❖ 实质的操作，无非是辅助栈的push()、pop()
 - 每次调用goAlongVine()后都恰有一次pop()，全程不过 $\mathcal{O}(n)$ 次
 - goAlongVine()过程中尽管push()的次数不定，但累计应与pop()一样多 //Aggregate
- ❖ 更多的实现：travIn_I2() + travIn_I3() + travIn_I4()

二叉树

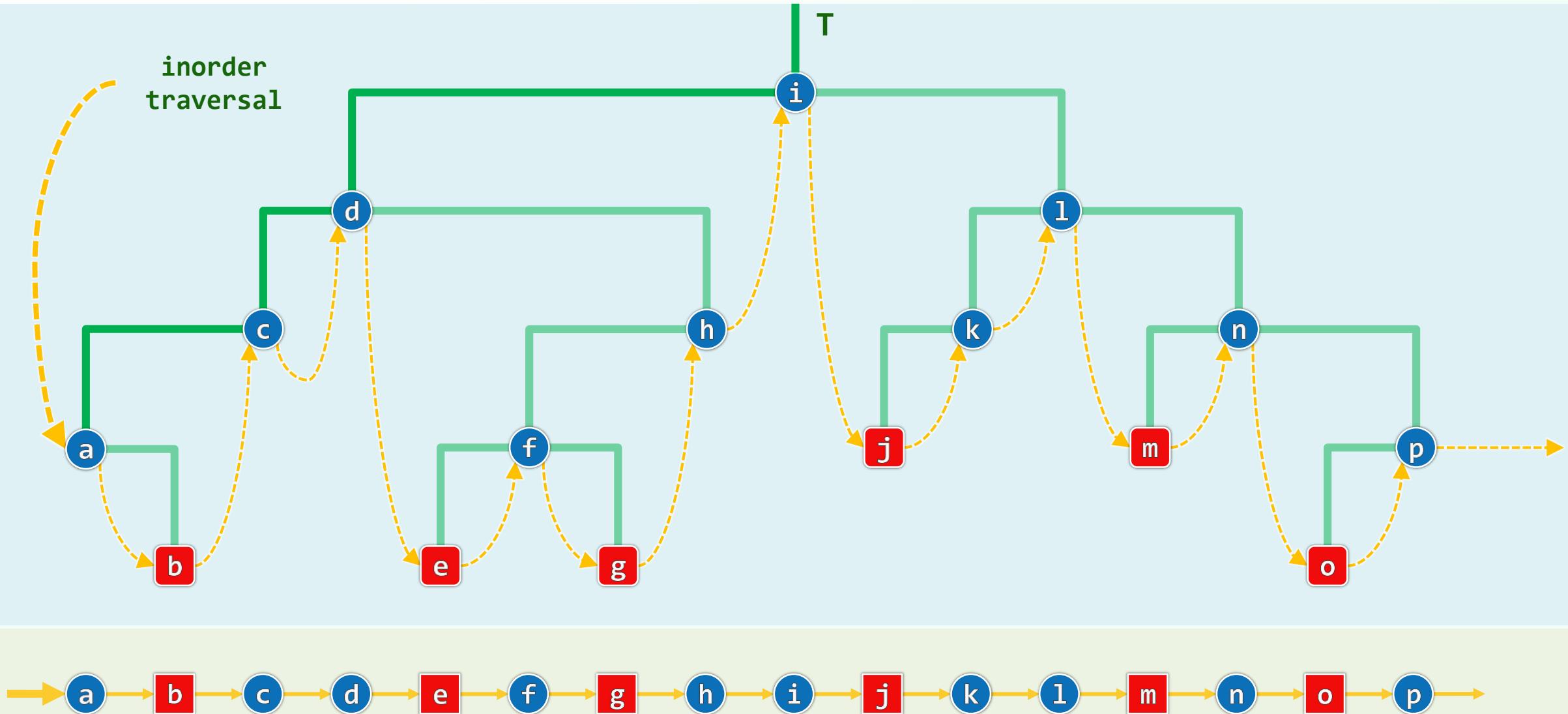
中序遍历：后继与前驱

e5 - F4

邓俊辉

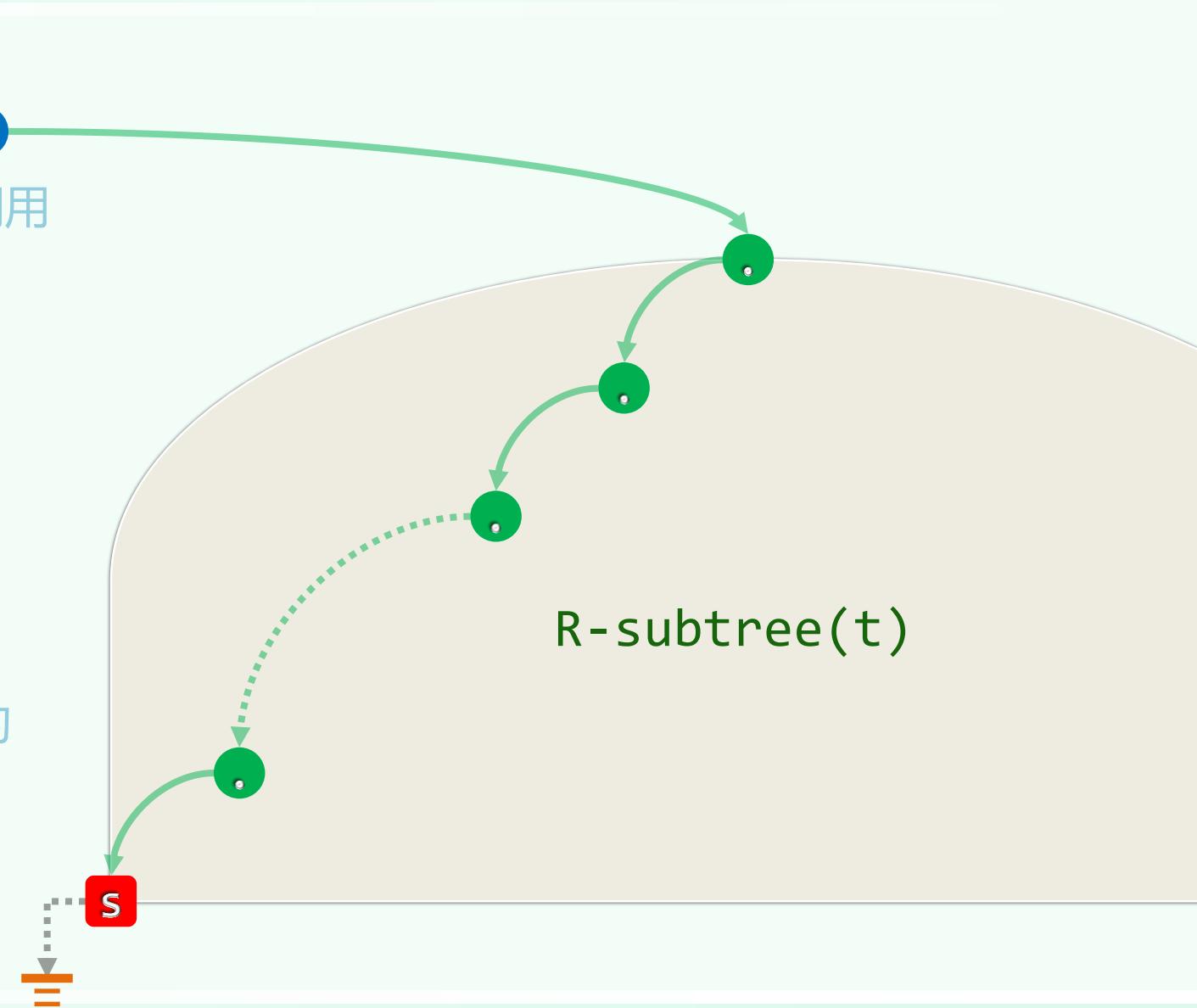
deng@tsinghua.edu.cn

```
for ( BinNodePosi<T> t = first(); t; t = t->succ() )
```

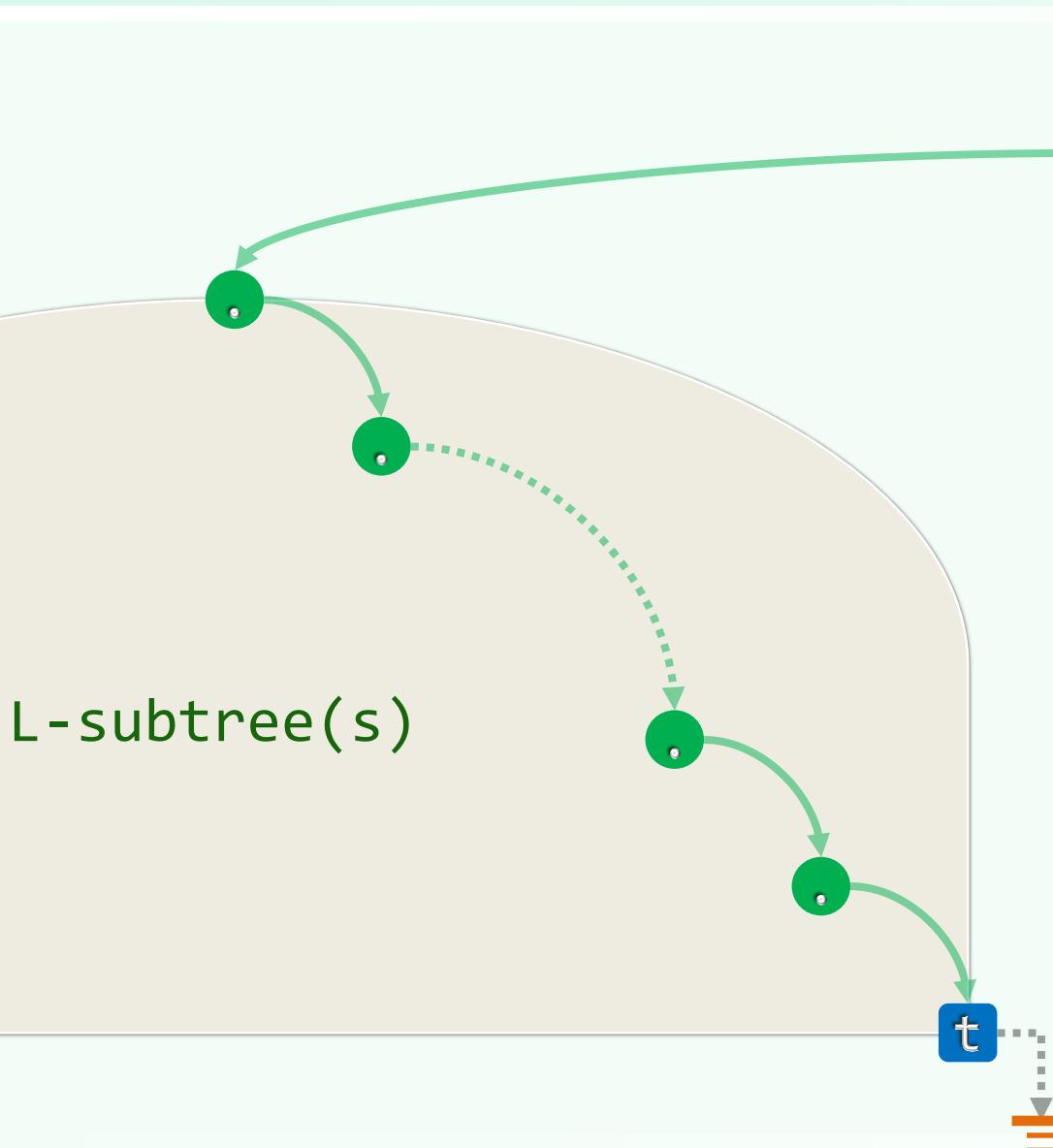


直接后继：最靠左的右后代

```
//在中序遍历意义下的直接后继  
//稍后将被BST::remove中的removeAt()调用  
template <typename T>  
BinNodePosi<T> BinNode<T>::succ() {  
  
    BinNodePosi<T> s = this;  
  
    if ( rc ) { //若有右孩子，则  
        s = rc; //直接后继必是右子树中的  
        while ( HasLChild( * s ) )  
            s = s->lC; //最小节点  
    } else /* ... */
```



直接后继：最低的左祖先



} else { //否则

//后继应是“以当前节点为直接前驱者”

while (IsRChild(* s))

s = s->parent; //不断朝左上移动

//最后再朝右上移动一步

s = s->parent; //可能是NULL

}

return s; //两种情况下，运行时间为

} //当前节点的高度与深度，不过 $\mathcal{O}(h)$

二叉树

后序遍历：观察

05 - G1

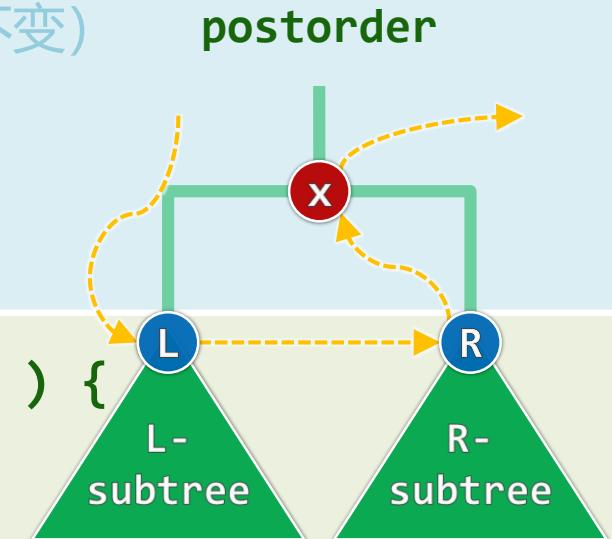
看了，因向仙姑道：“敢烦仙姑引我到那各司中游玩游玩，不知可使得？”仙姑道：“此各司中皆贮的是普天之下所有的女子过去未来的簿册，尔凡眼尘躯，未便先知的。”

邓俊辉

deng@tsinghua.edu.cn

后序实例：子树删除

```
template <typename T> Rank BinTree<T>::remove( BinNodePosi<T> x ) {  
    FromParentTo( * x ) = NULL;  
    updateHeightAbove( x->parent ); //更新祖先高度 (其余节点亦不变)  
    Rank n = removeAt(x); _size -= n; return n;  
}  
  
template <typename T> static Rank removeAt( BinNodePosi<T> x ) {  
    if ( ! x ) return 0;  
    Rank n = 1 + removeAt( x->lc ) + removeAt( x->rc );  
    release(x->data); release(x); return n;  
}
```



递归实现

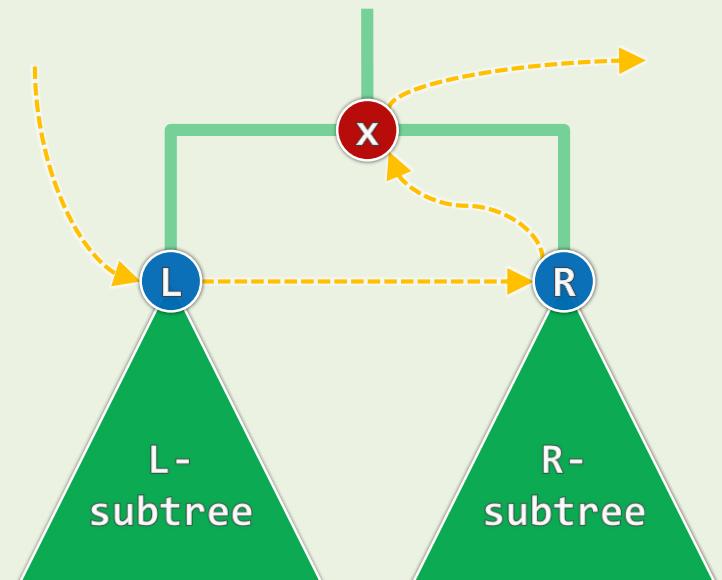
❖ 应用: BinNode::size() + BinTree::updateHeight()

❖ template <typename T, typename VST>

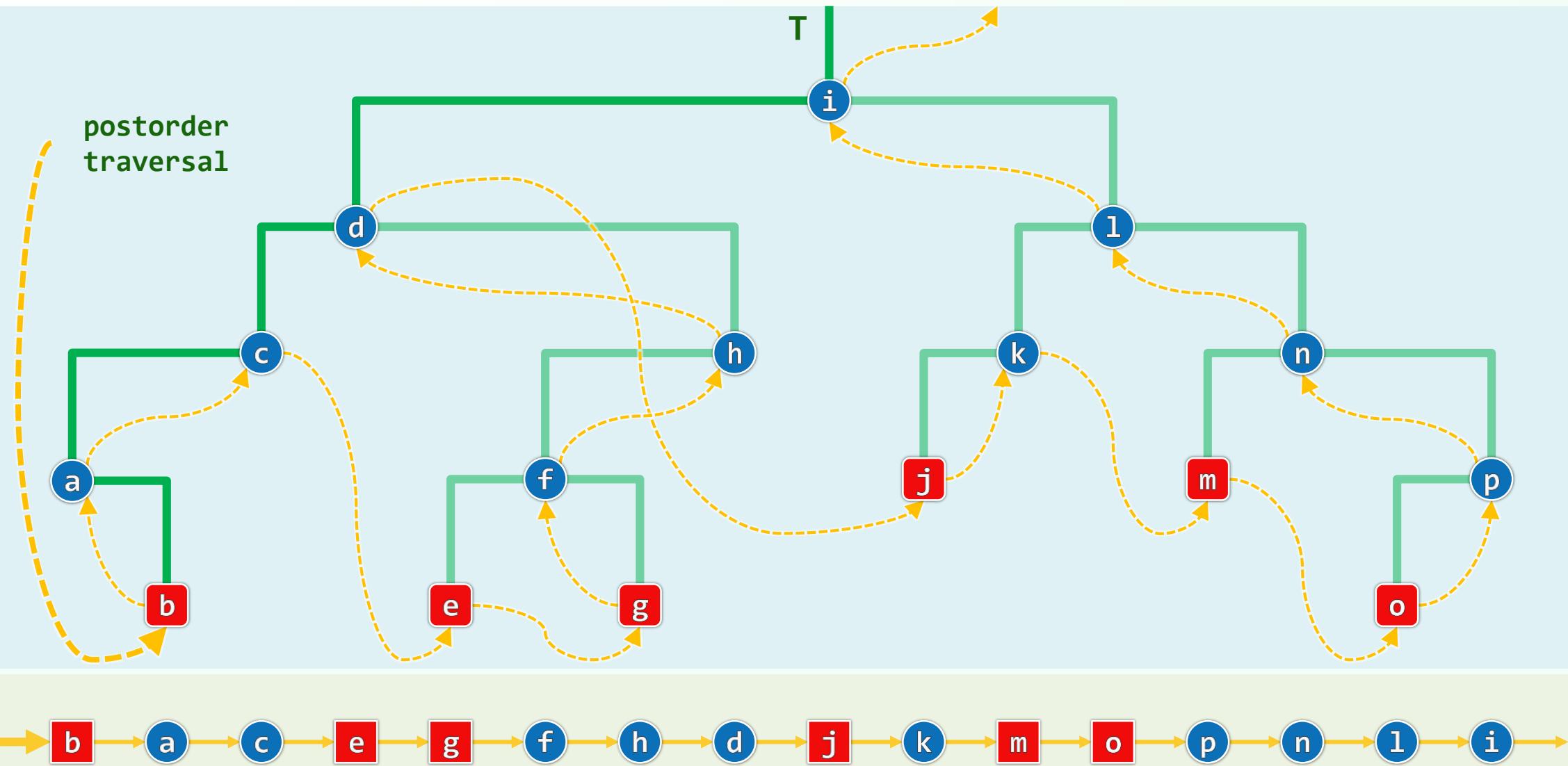
```
void traverse( BinNodePosi<T> x, VST & visit ) {  
    if ( ! x ) return;  
  
    traverse( x->lc, visit );  
  
    traverse( x->rc, visit );  
  
    visit( x->data );  
}
```

❖ $T(n) = T(a) + T(n - a - 1) + \mathcal{O}(1) = \mathcal{O}(n)$

❖ 挑战: 不依赖递归机制, 如何实现后序遍历? 效率如何?



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**

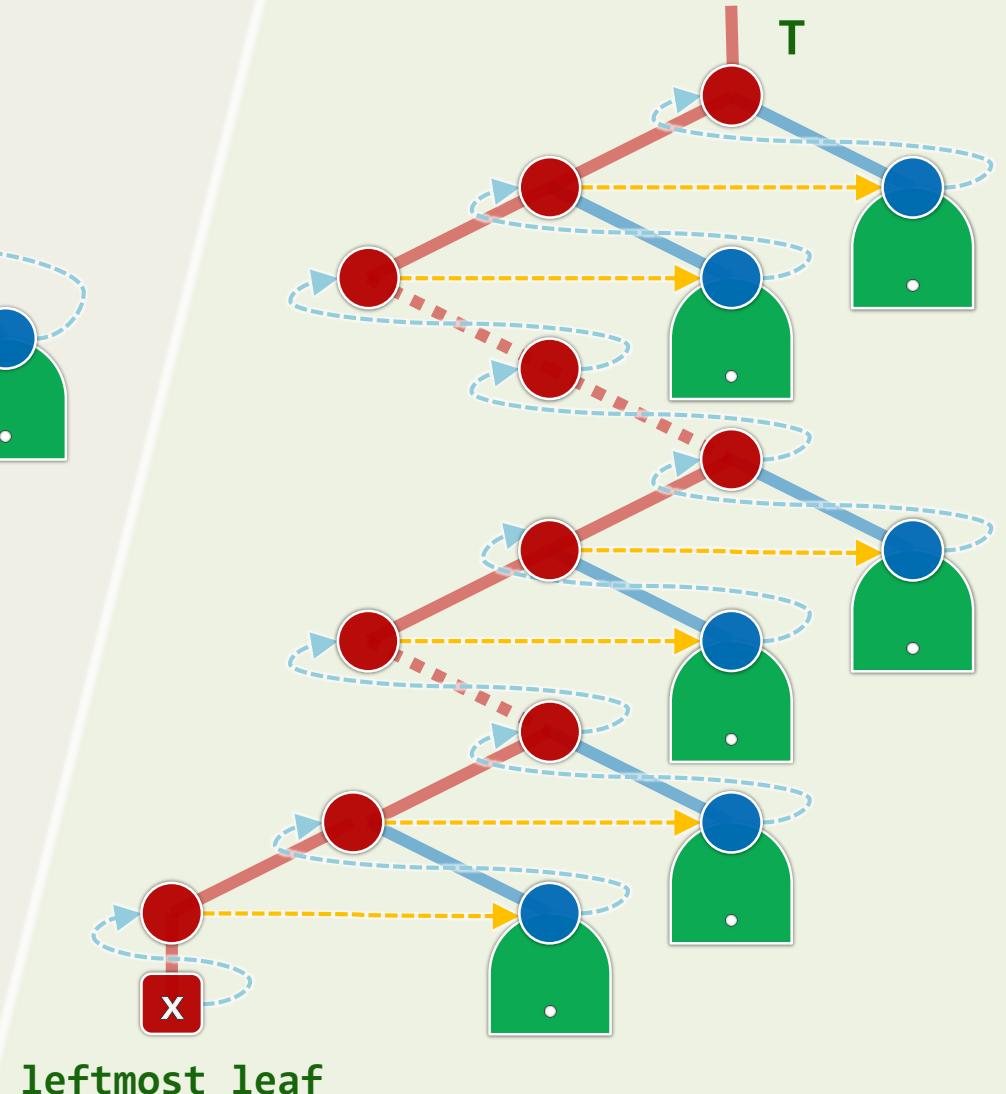
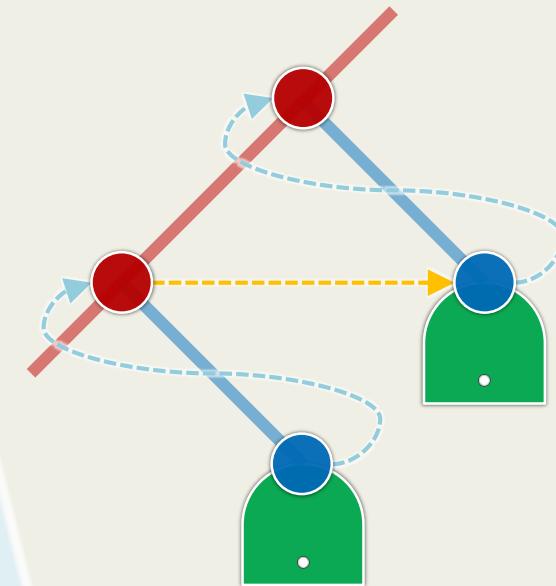
❖ 最后一个节点

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...



二叉树

后序遍历：迭代算法

05 - G2

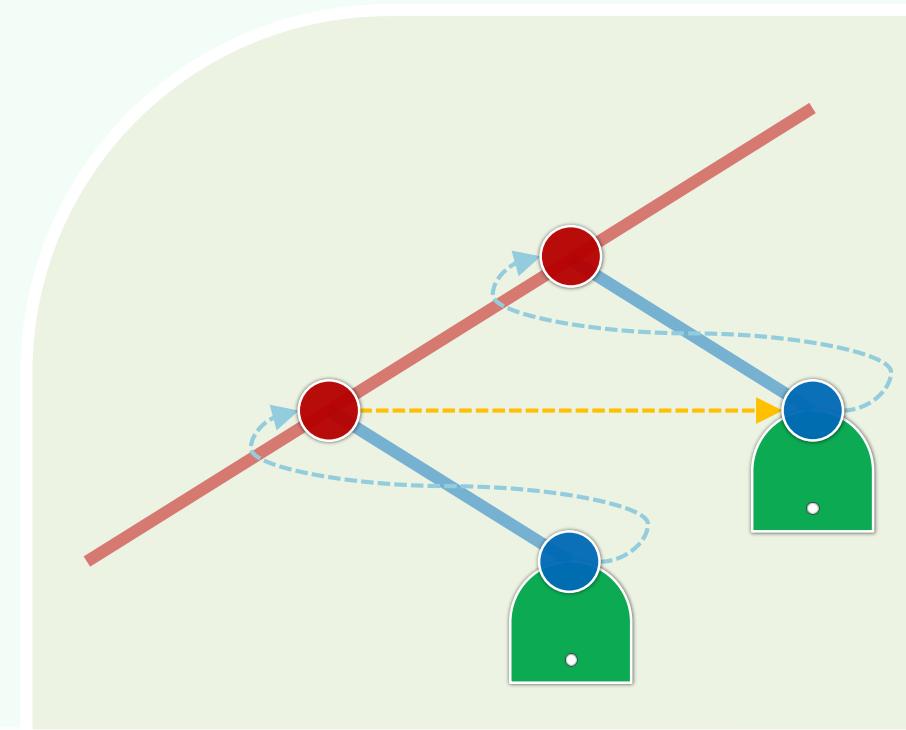
邓俊辉

deng@tsinghua.edu.cn

大宗百世不迁，小宗五世则迁

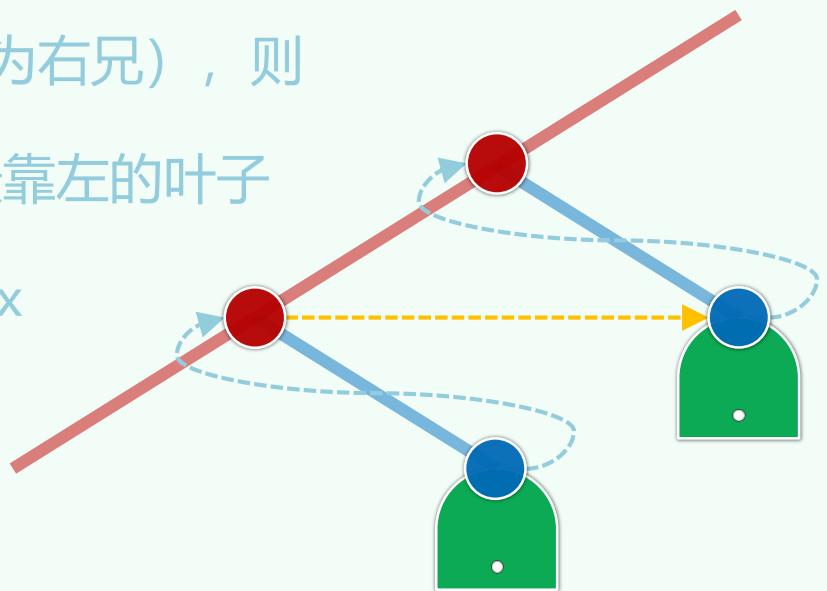
序曲

```
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  
}
```



全曲

```
template <typename T, typename V> void travPost_I( BinNodePosi<T> x, V & 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），则  
            gotoLeftmostLeaf( S ); //在其右兄子树中找到最靠左的叶子  
  
        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  
    }  
}
```



二叉树

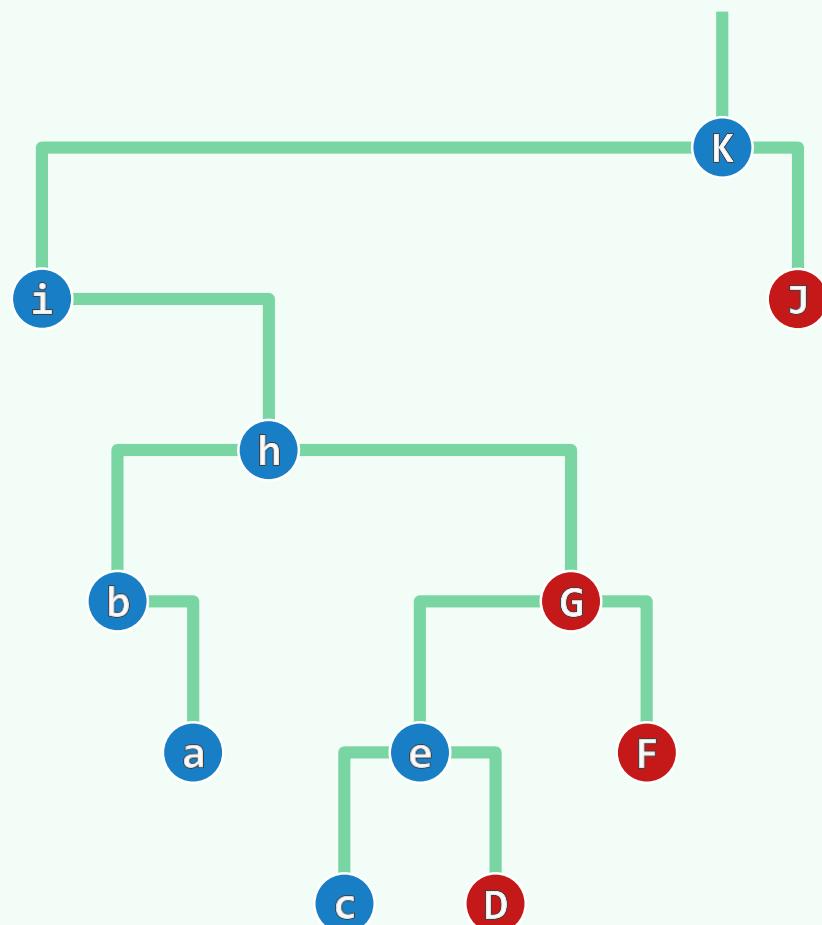
后序遍历：实例

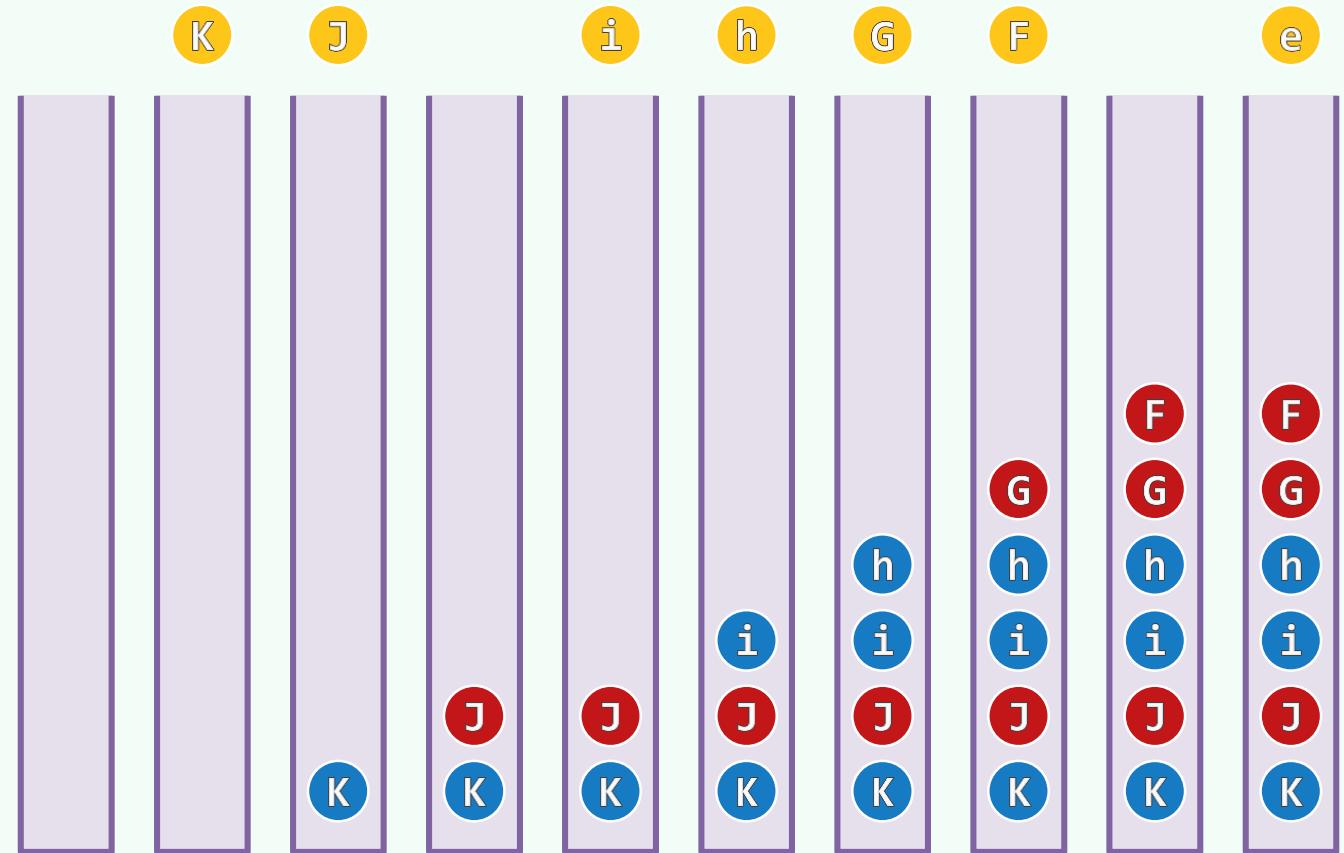
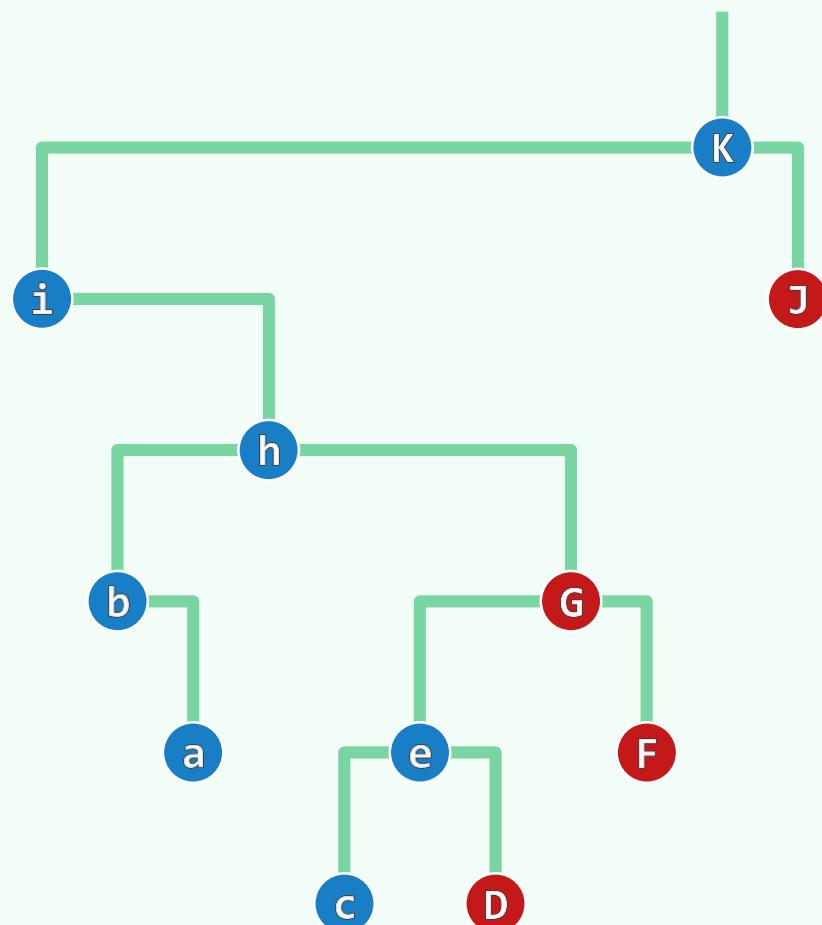
05 - G3

邓俊辉

deng@tsinghua.edu.cn

很久以来
我就渴望升起
长长的，象绿色植物
去缠绕黄昏的光线





二叉树

后序遍历：分析

05 - G4

邓俊辉

deng@tsinghua.edu.cn

這三家的人，在族譜之中尋查自己的譜系，卻尋不著，
因此算為不潔，不准供祭司的職任。

正确性：数学归纳

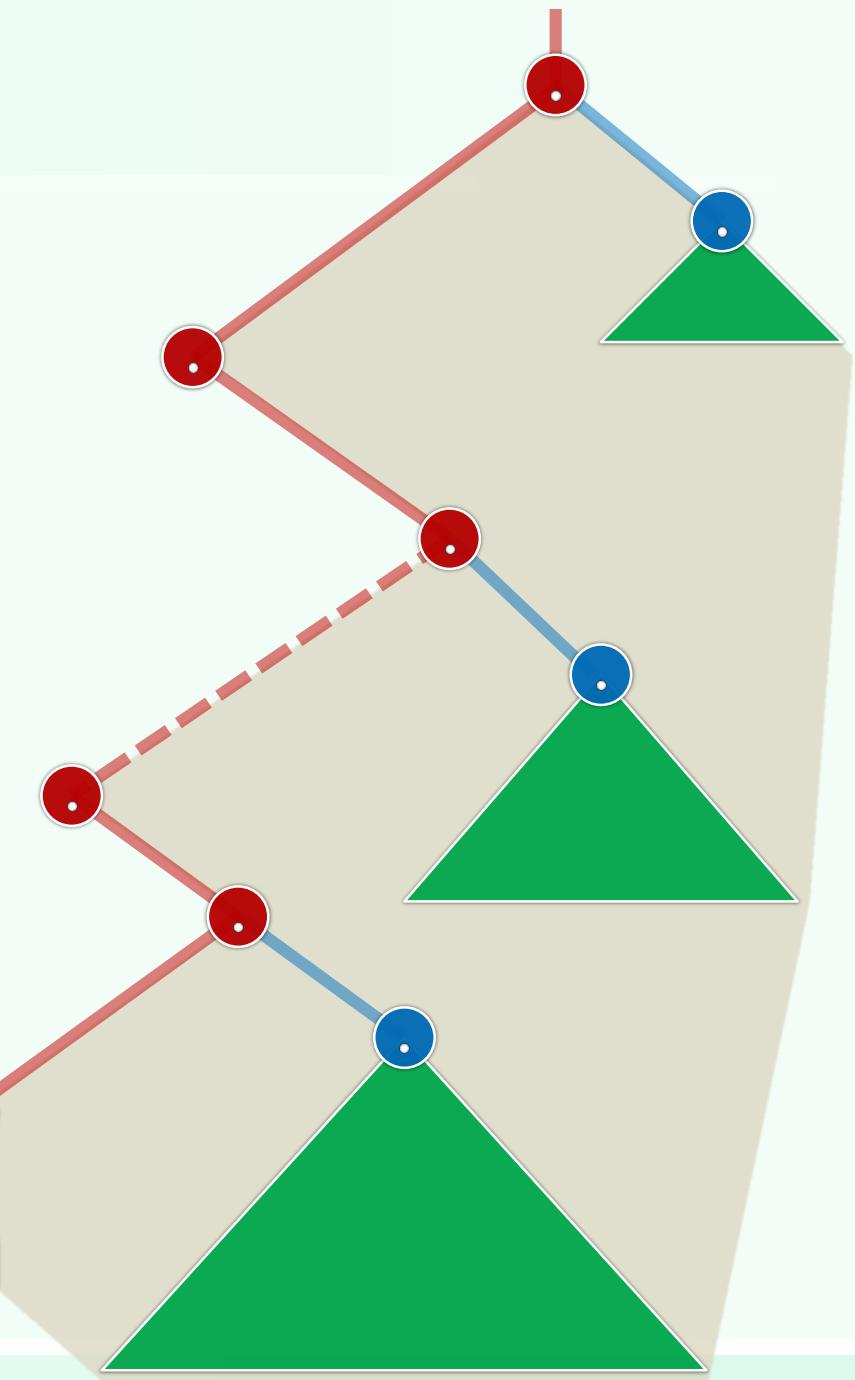
❖ 每个节点出栈后

- 以之为根的子树已经完全遍历，而且
- 其右兄弟 r 若存在，必恰在栈顶

❖ 此时正可以开始遍历子树 r

故只需从 r 出发...

leftmost leaf x



效率：分摊分析

❖ 与中序遍历类似地...

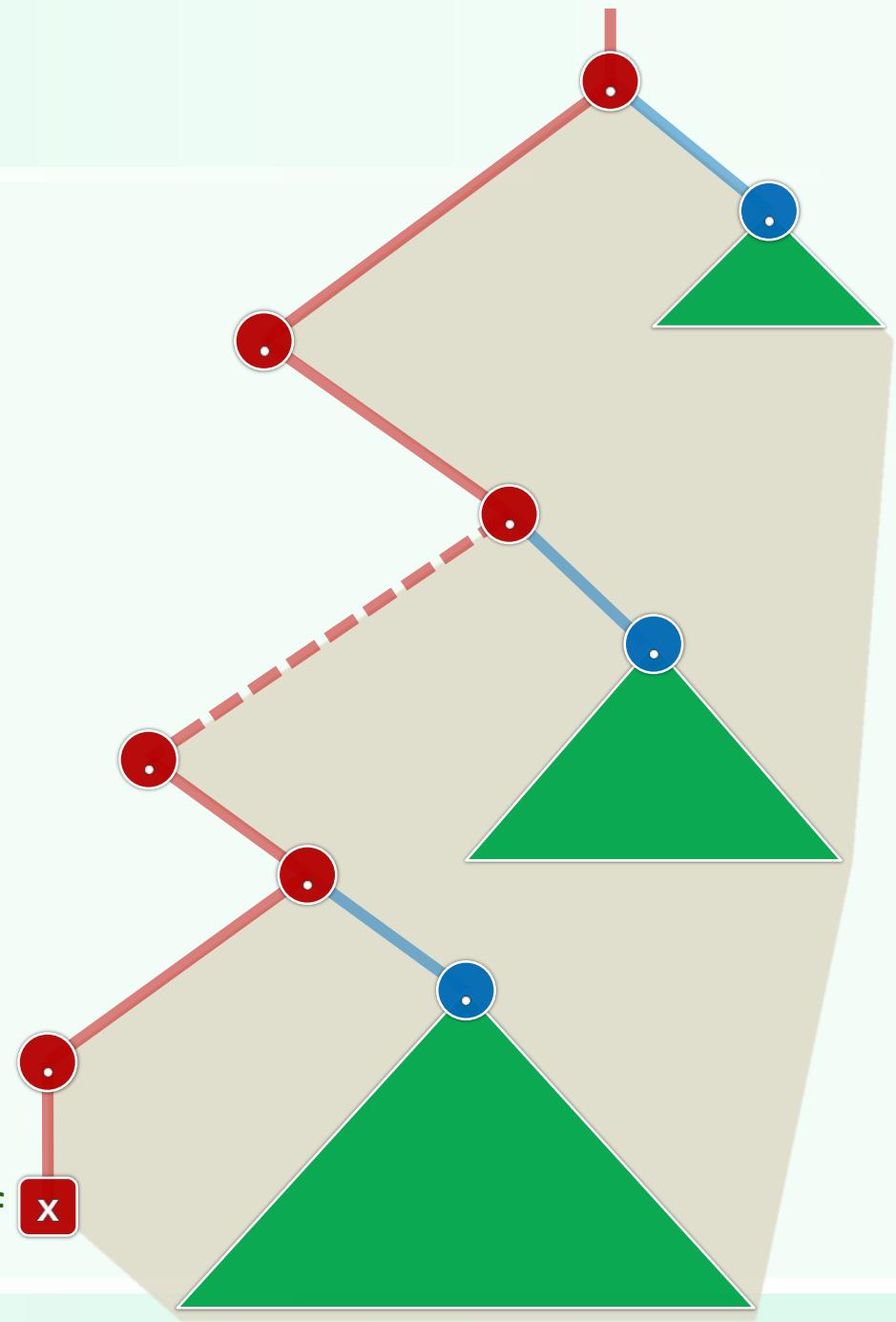
- gotoLeftmostLeaf()最多需调用 $\Omega(n)$ 次
- 单次调用最多需做 $\Omega(n)$ 次push()

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

❖ 与中序遍历一样，通过分摊分析可确定

更加精准的界是 $\mathcal{O}(n)$ ！

leftmost leaf x



二叉树

后序遍历：表达式树

e5 - G5

邓俊辉

deng@tsinghua.edu.cn

Expression + Parentheses

(0 ! + 1) ^ (2 * 3 ! + 4 - 5) + 6 * 7 - 8 * 9

(((((0 !) + 1) ^ (((2 * (3 !)) + 4) - 5)) + (6 * 7)) - (8 * 9))

() -)

() +) (*)

(^)) (*) 8 9

(+) (-) 6 7

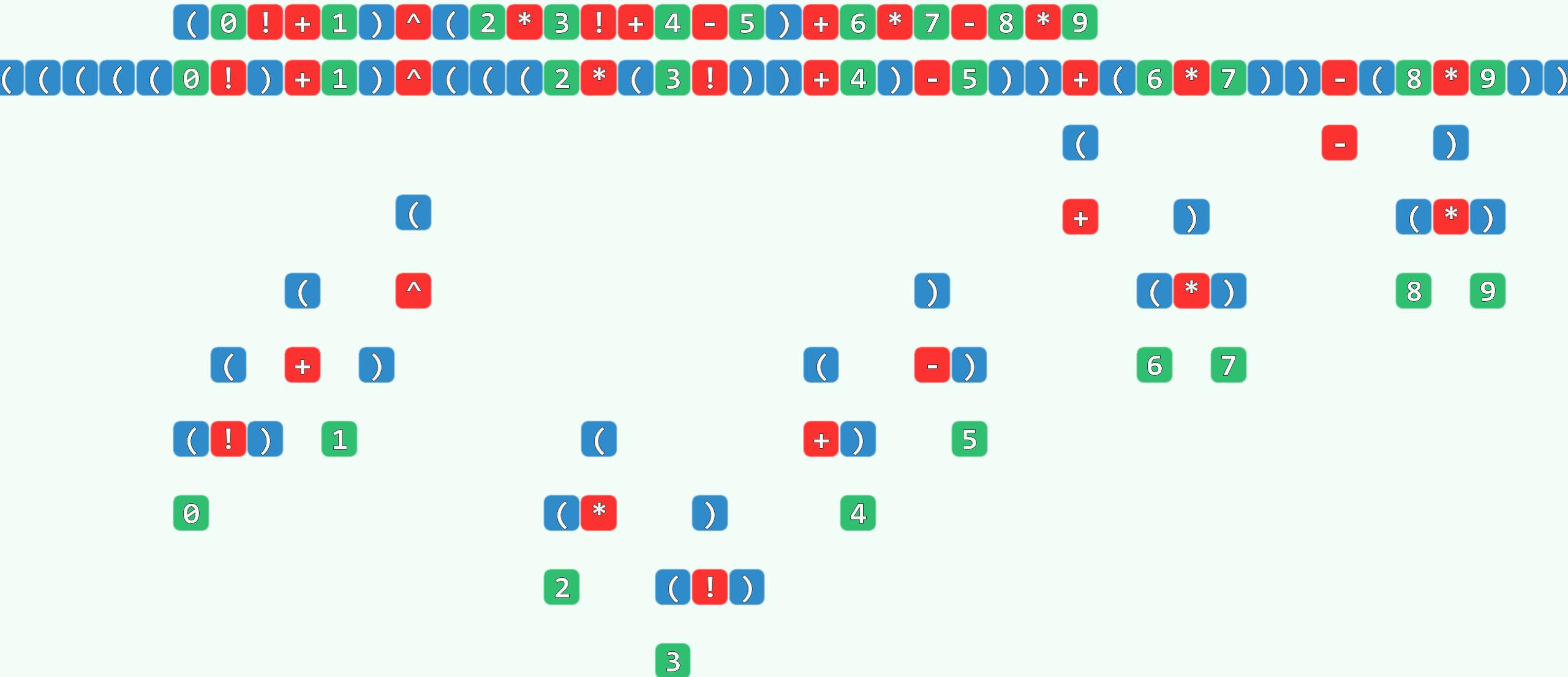
(!) 1 (+) 5

0 (*) 4

2 (!)

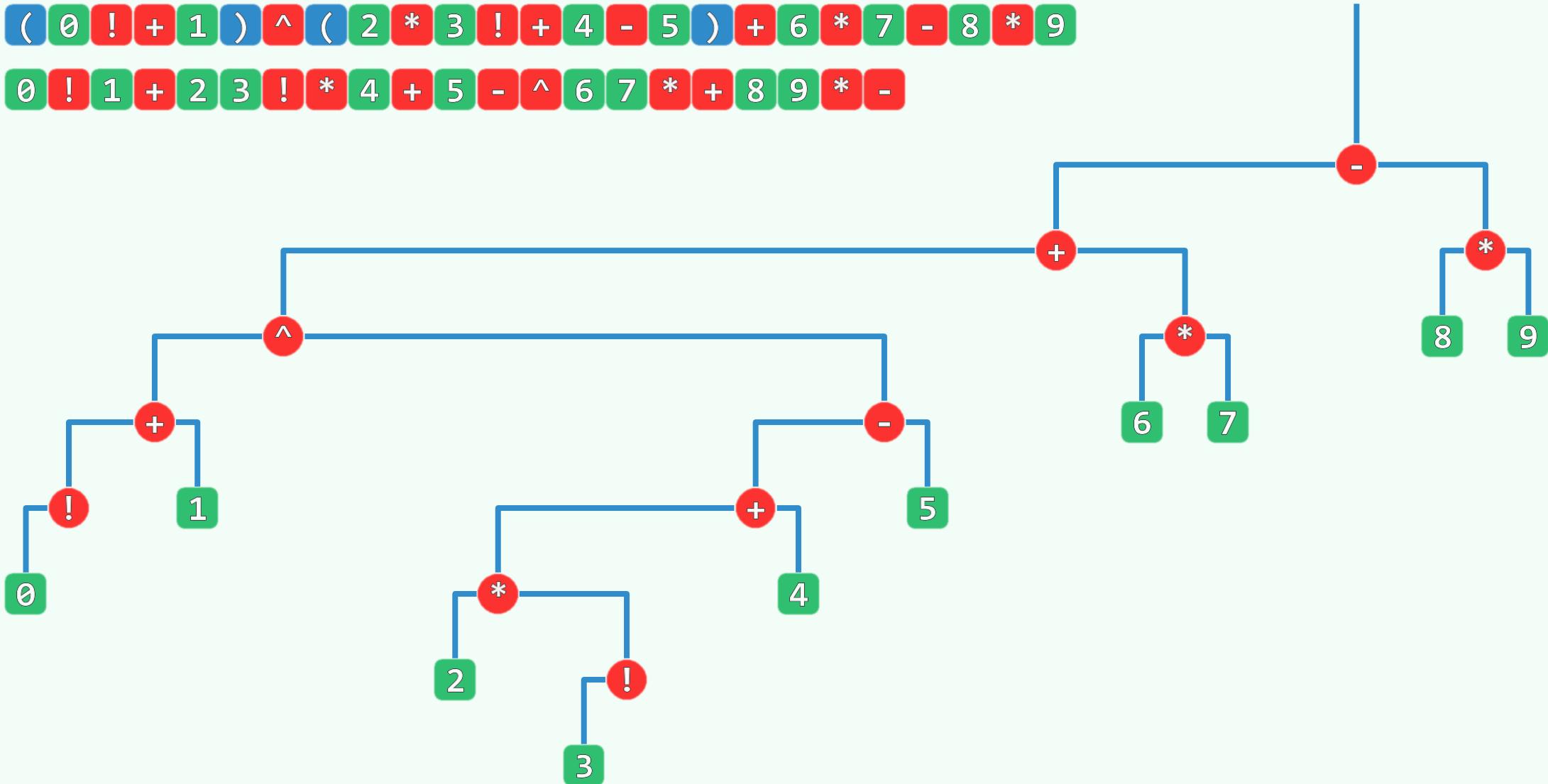
3

Expression - Parentheses



Expression Tree ~ Postorder ~ RPN

(0 ! + 1) ^ (2 * 3 ! + 4 - 5) + 6 * 7 - 8 * 9
0 ! 1 + 2 3 ! * 4 + 5 - ^ 6 7 * + 8 9 * -



二叉树

层次遍历：算法及分析

e5-H2

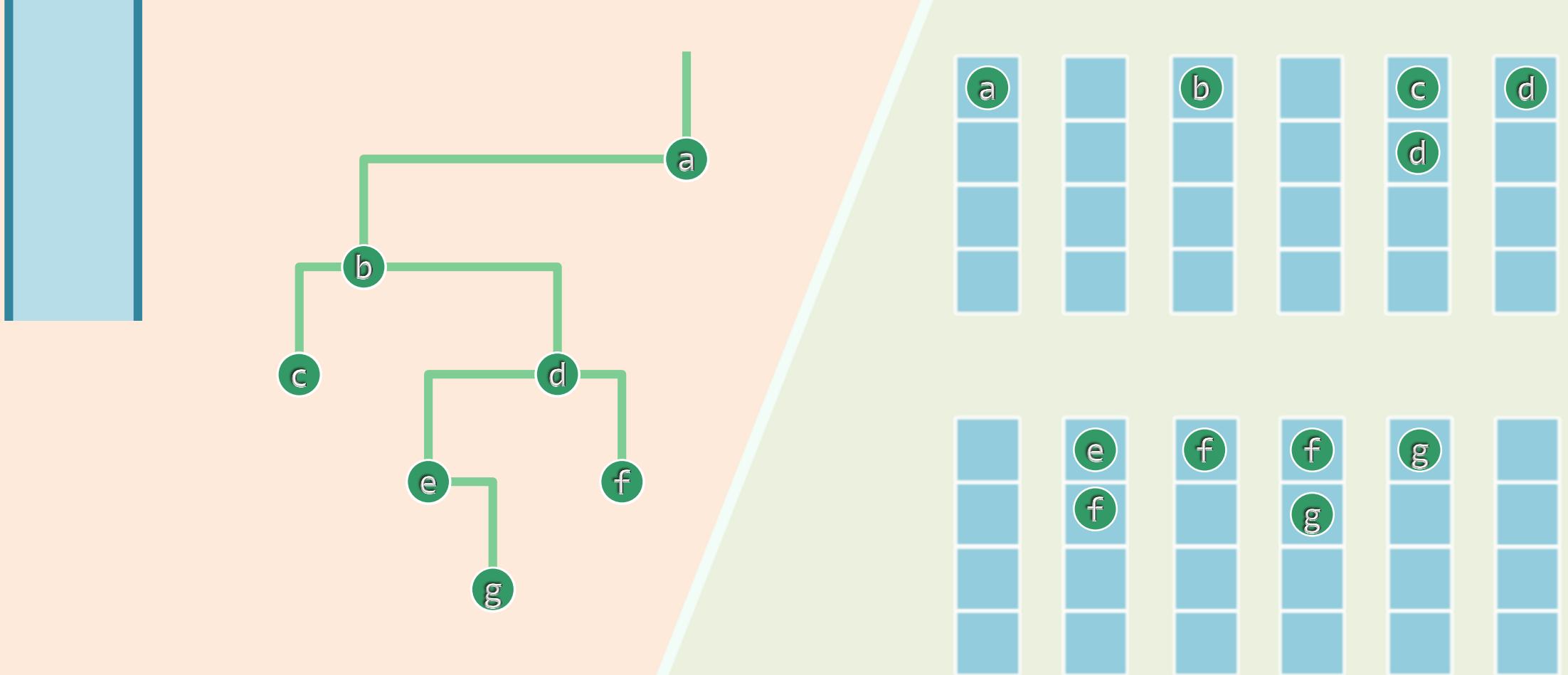
邓俊辉

deng@tsinghua.edu.cn

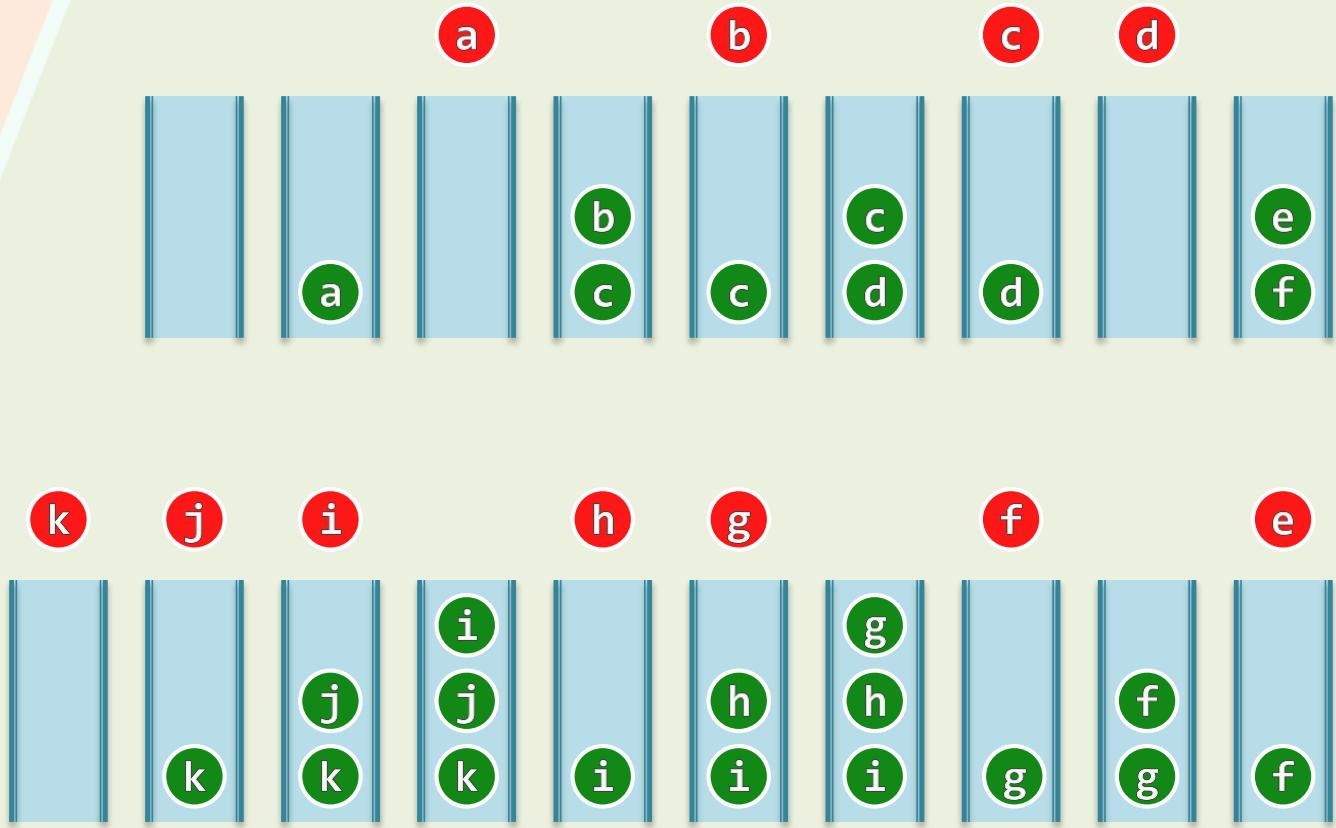
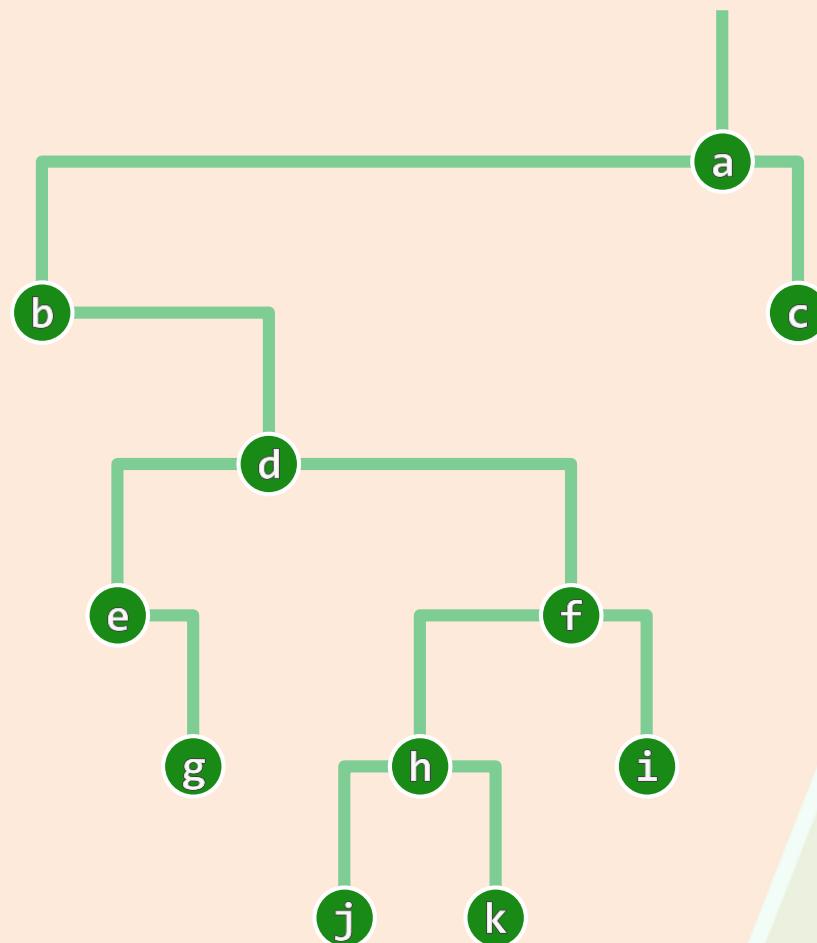
算法实现

```
template <typename T> template <typename VST>
void BinNode<T>::travLevel( VST & visit ) { //二叉树层次遍历
    Queue< BinNodePosi<T> > Q; Q.enqueue( this ); //引入辅助队列，根节点入队
    while ( ! Q.empty() ) { //在队列再次变空之前，反复迭代
        BinNodePosi<T> x = Q.dequeue(); visit( x->data ); //取出队首节点并随即访问
        if ( HasLChild( *x ) ) Q.enqueue( x->lC ); //左孩子入队
        if ( HasRChild( *x ) ) Q.enqueue( x->rC ); //右孩子入队
    }
}
```

实例



实例



二叉树

层次遍历：完全二叉树

05 - H2

我们注定是扎根于前半生的，即使后半生充满了强烈的和令人
感动的经历。

邓俊辉

deng@tsinghua.edu.cn

完全二叉树 ~ 紧凑表示 ~ 以向量实现

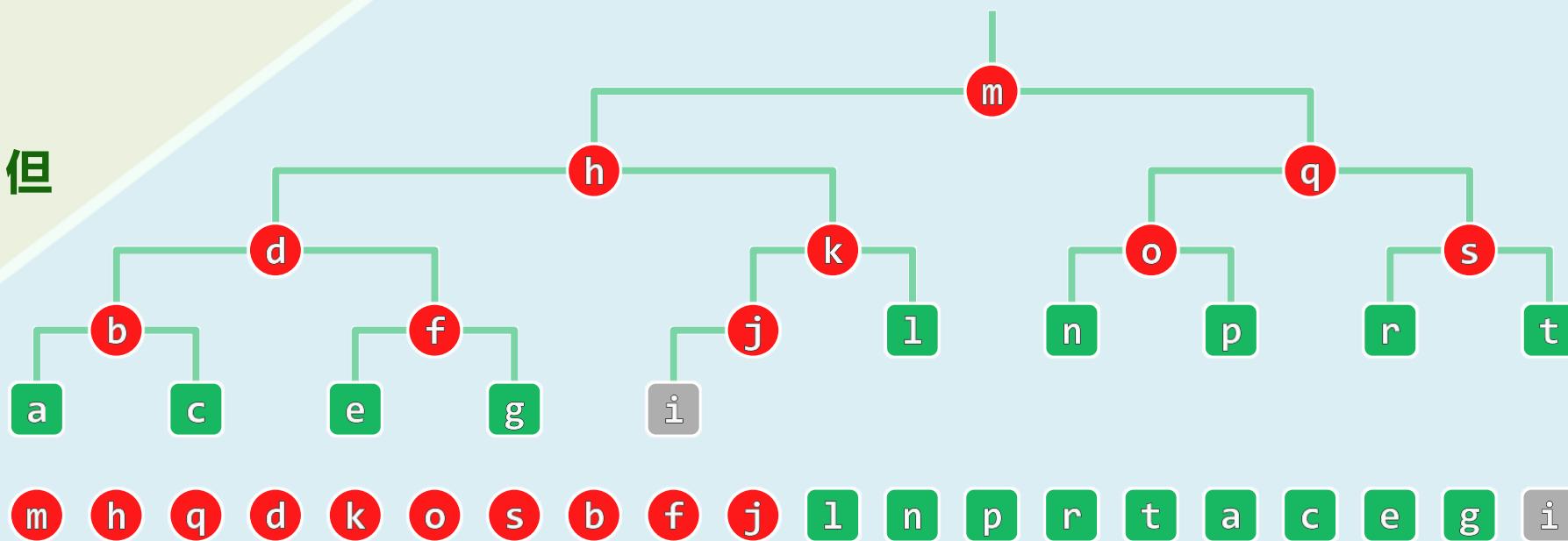
❖ 叶节点仅限于最低两层

底层叶子，均居于次底层叶子左侧（相对于LCA）

除末节点的父亲，内部节点均有双子

❖ 叶节点

- 不致少于内部节点，但
- 至多出一个



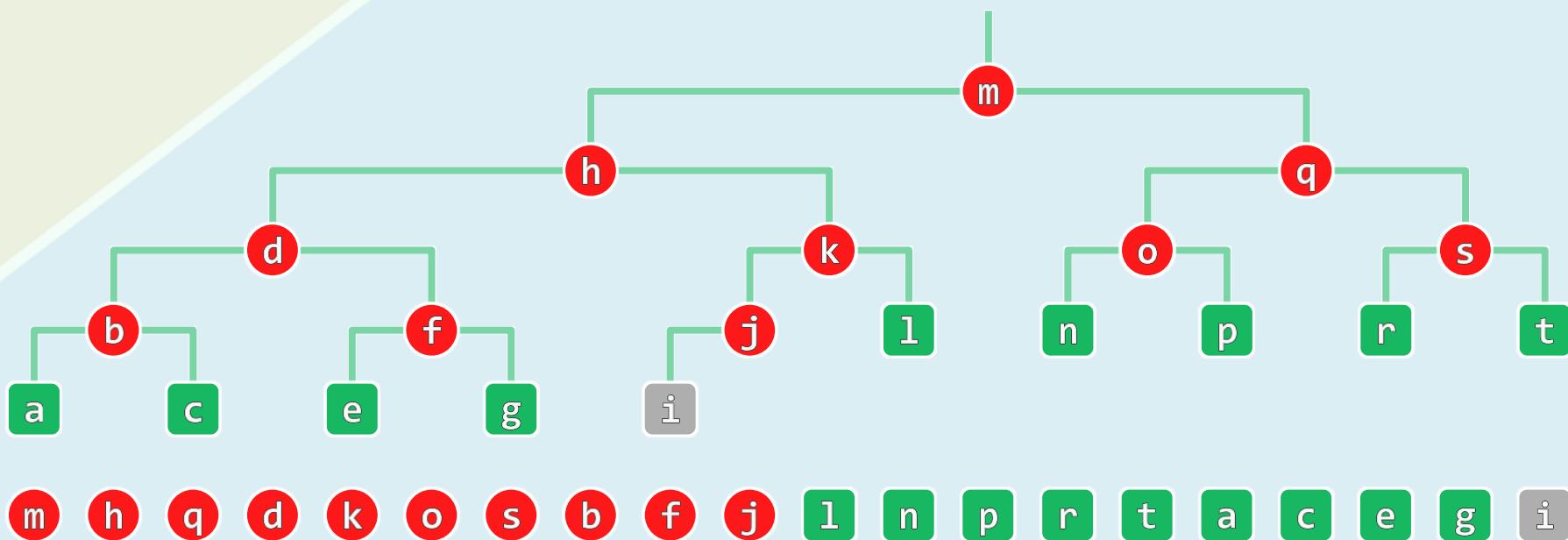
层次遍历

❖ 考察遍历过程中的 n 步迭代...

❖ 前 $\lceil n/2 \rceil - 1$ 步迭代中，均有**右孩子入队**

前 $\lfloor n/2 \rfloor$ 步迭代中，都有**左孩子入队**

❖ 累计至少 $n - 1$ 次入队



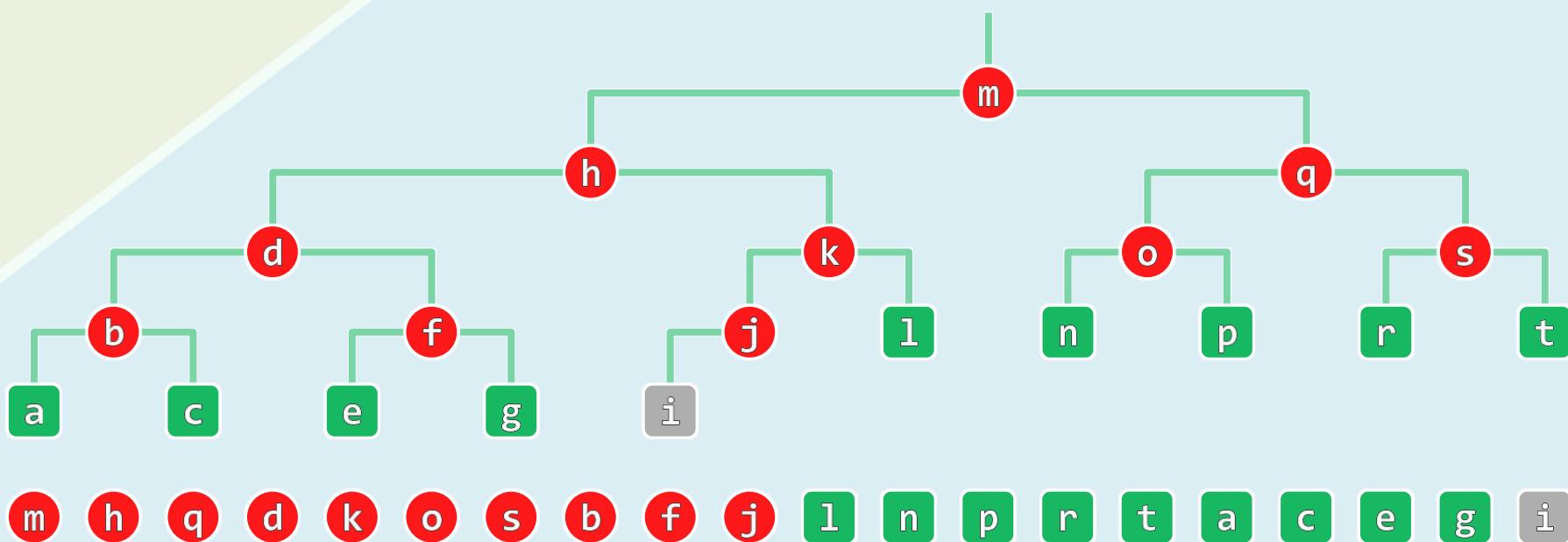
辅助队列的规模

❖ 先增后减，单峰且对称

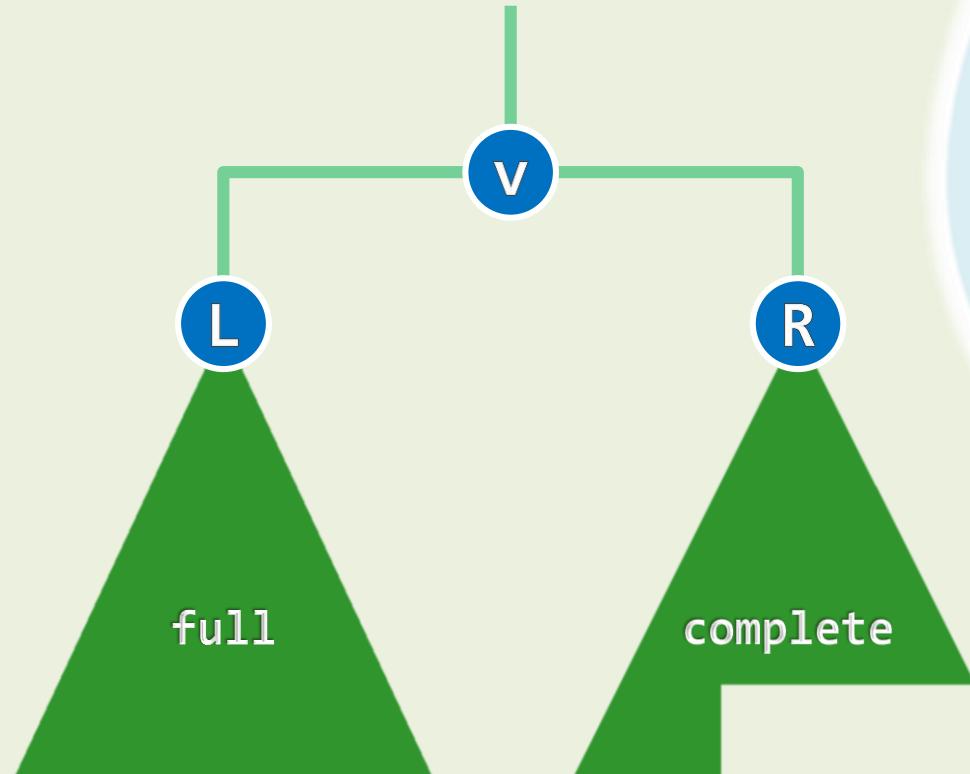
❖ 最大规模 = $\lceil n/2 \rceil$

(前 $\lceil n/2 \rceil - 1$ 次均出1入2)

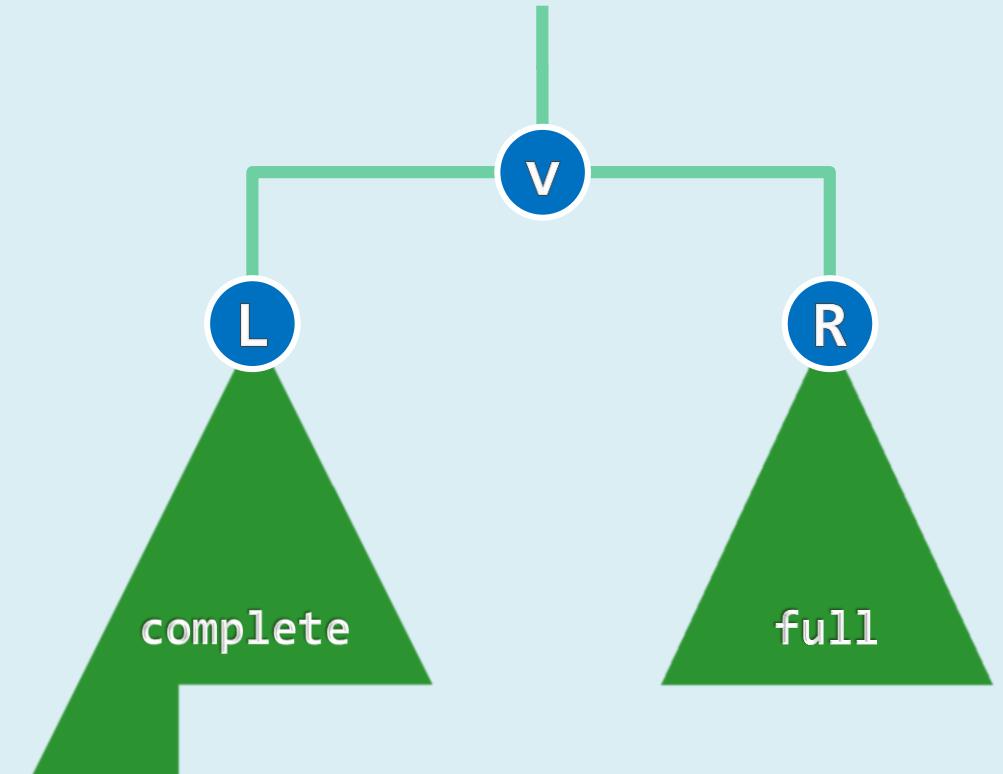
❖ 最大规模可能出现2次



完全 ~ 满



左满，右完全



左完全，右满

二叉树

重构

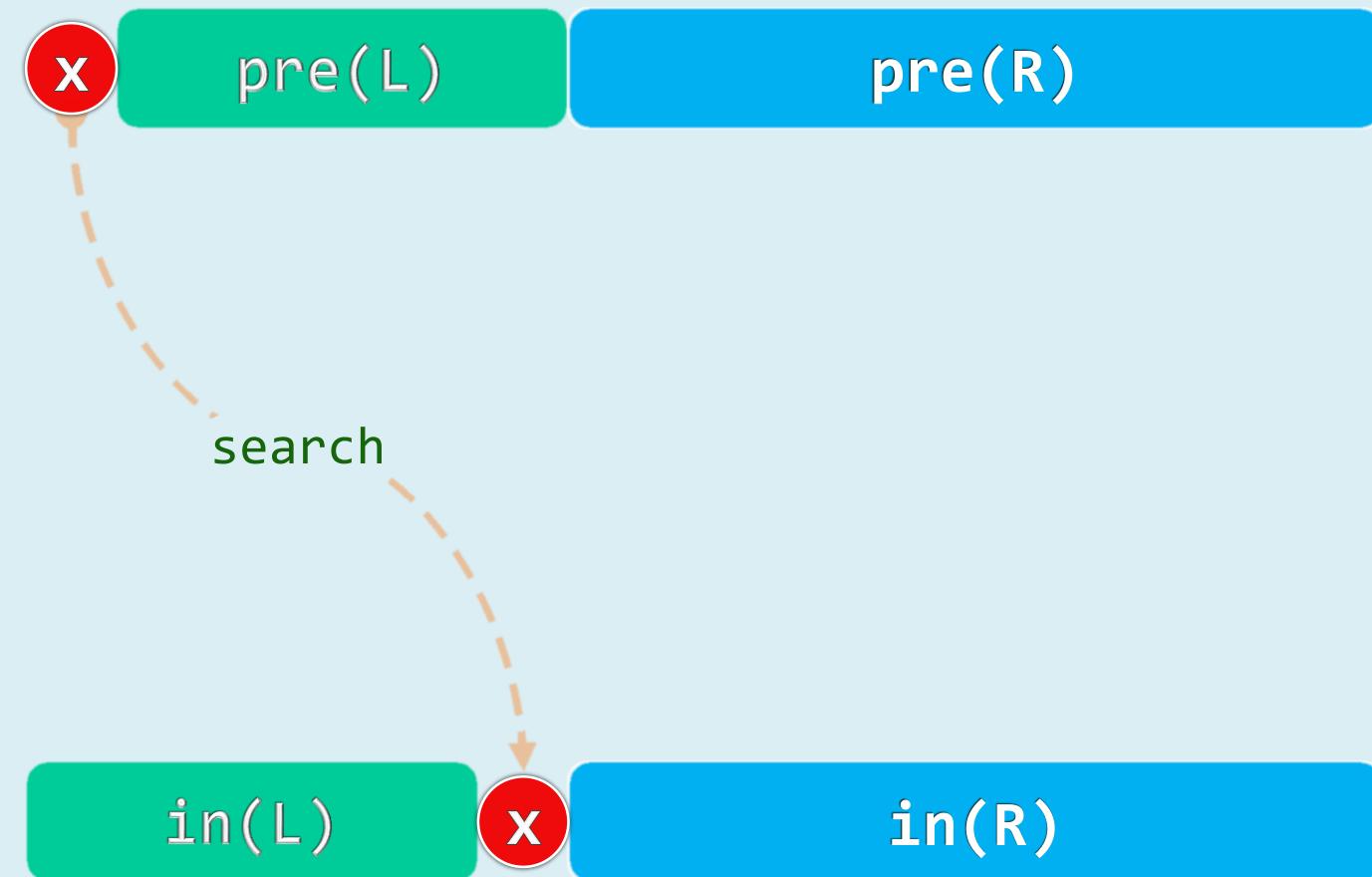
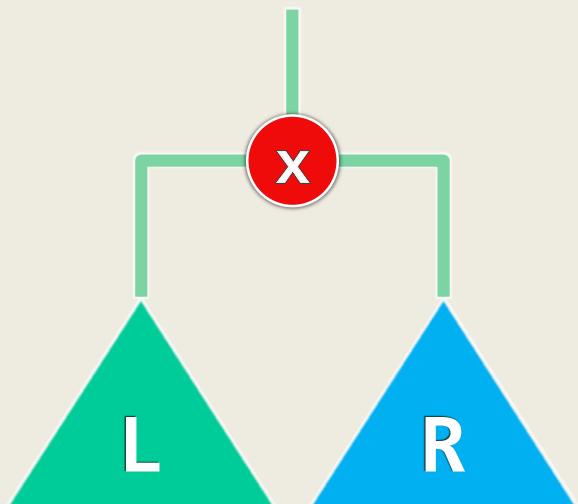


No matter where they take us,
We'll find our own way back.

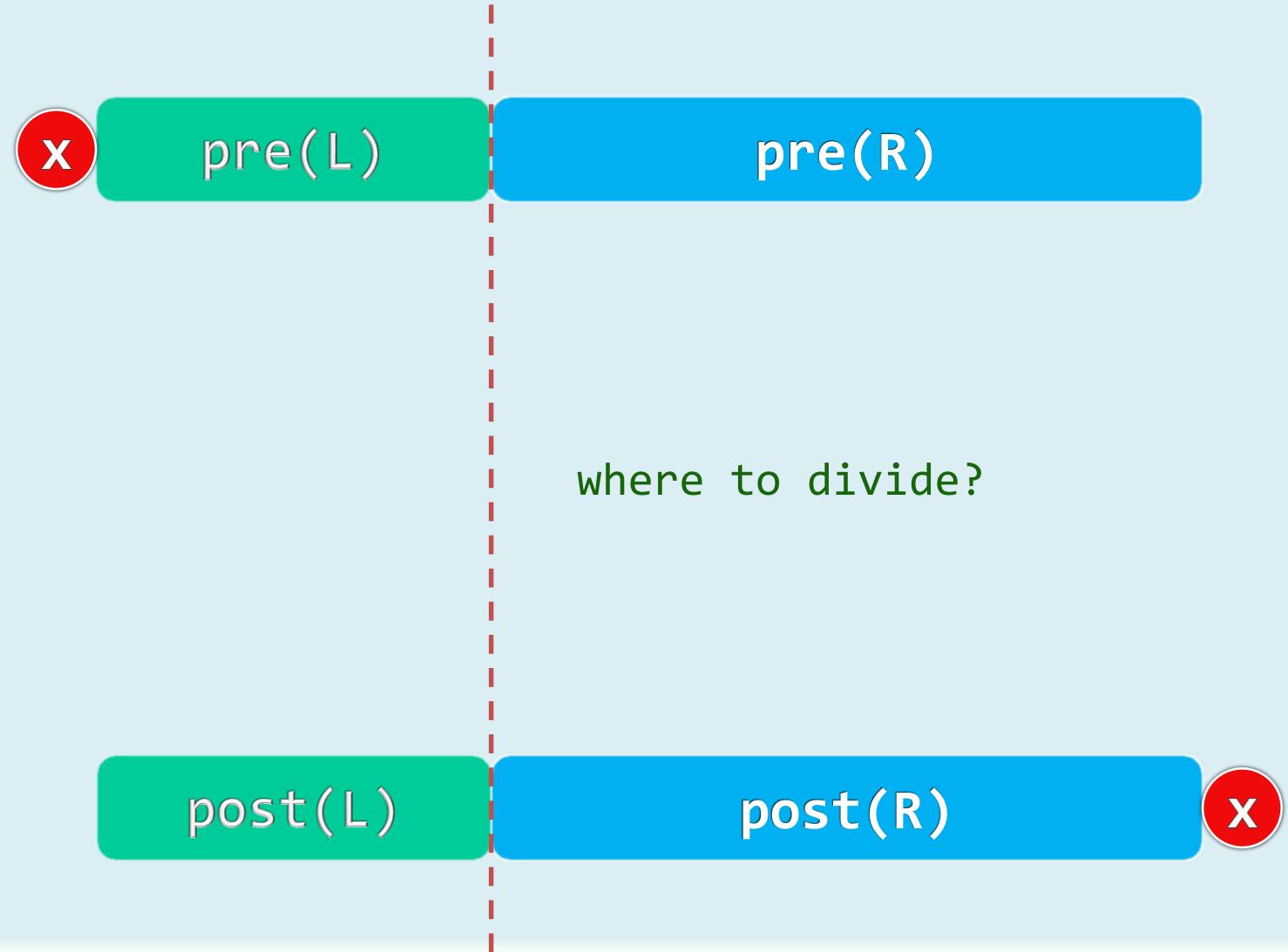
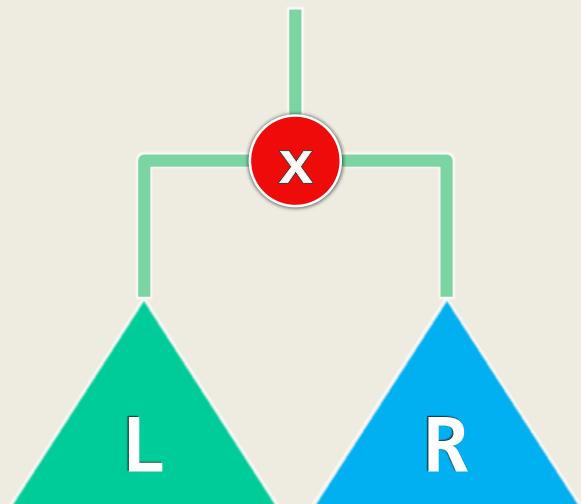
邓俊辉

deng@tsinghua.edu.cn

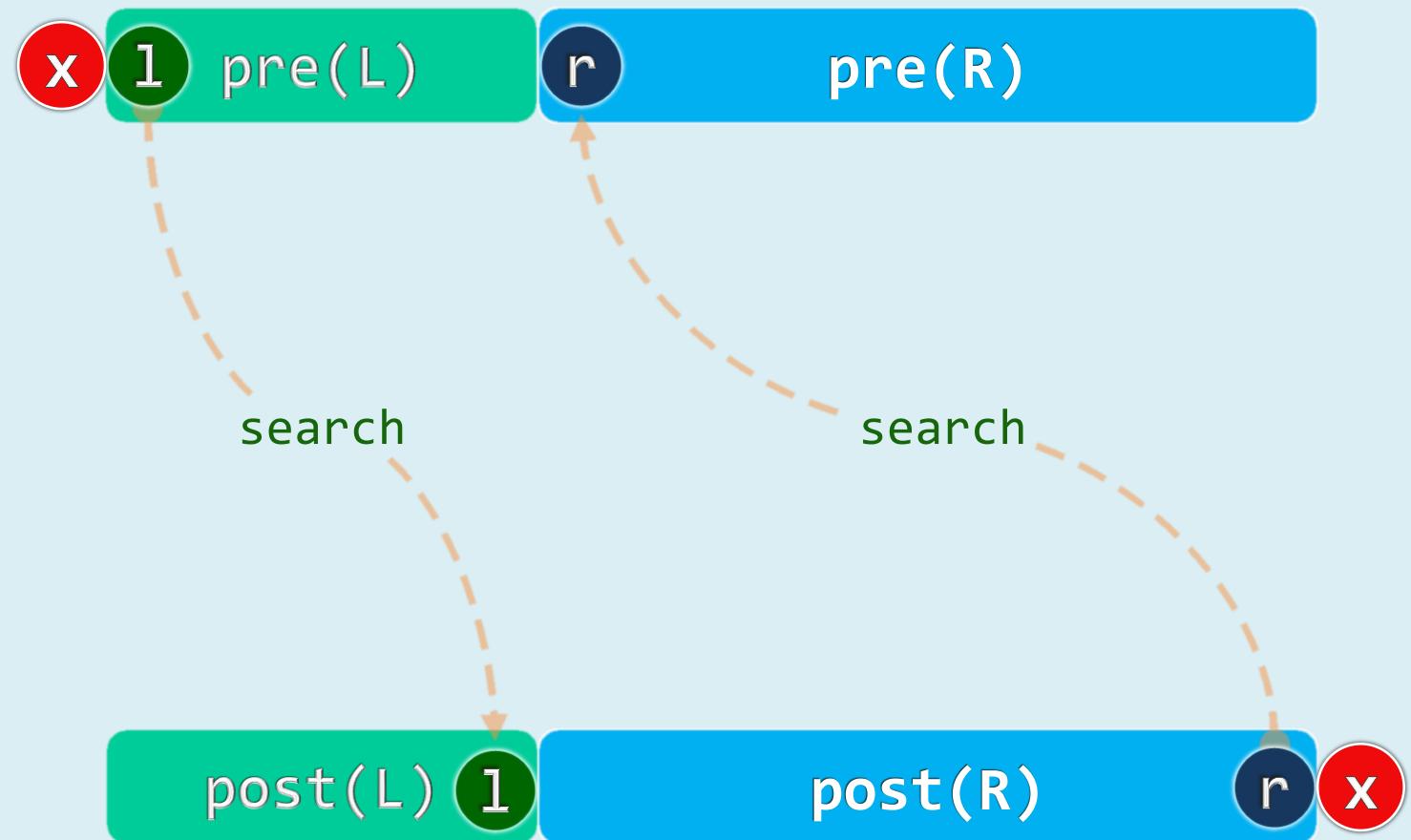
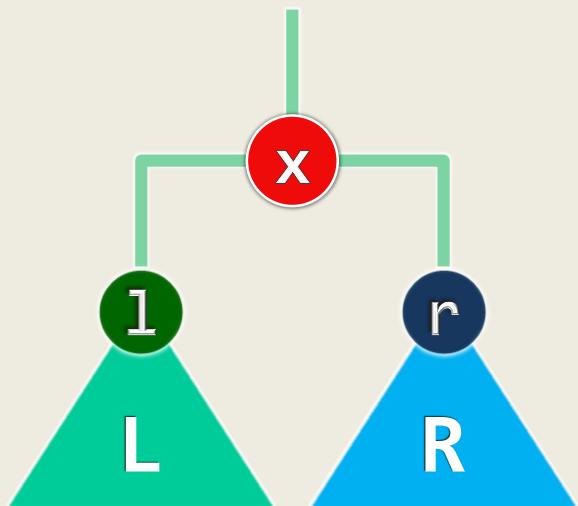
[先序 | 后序] + 中序



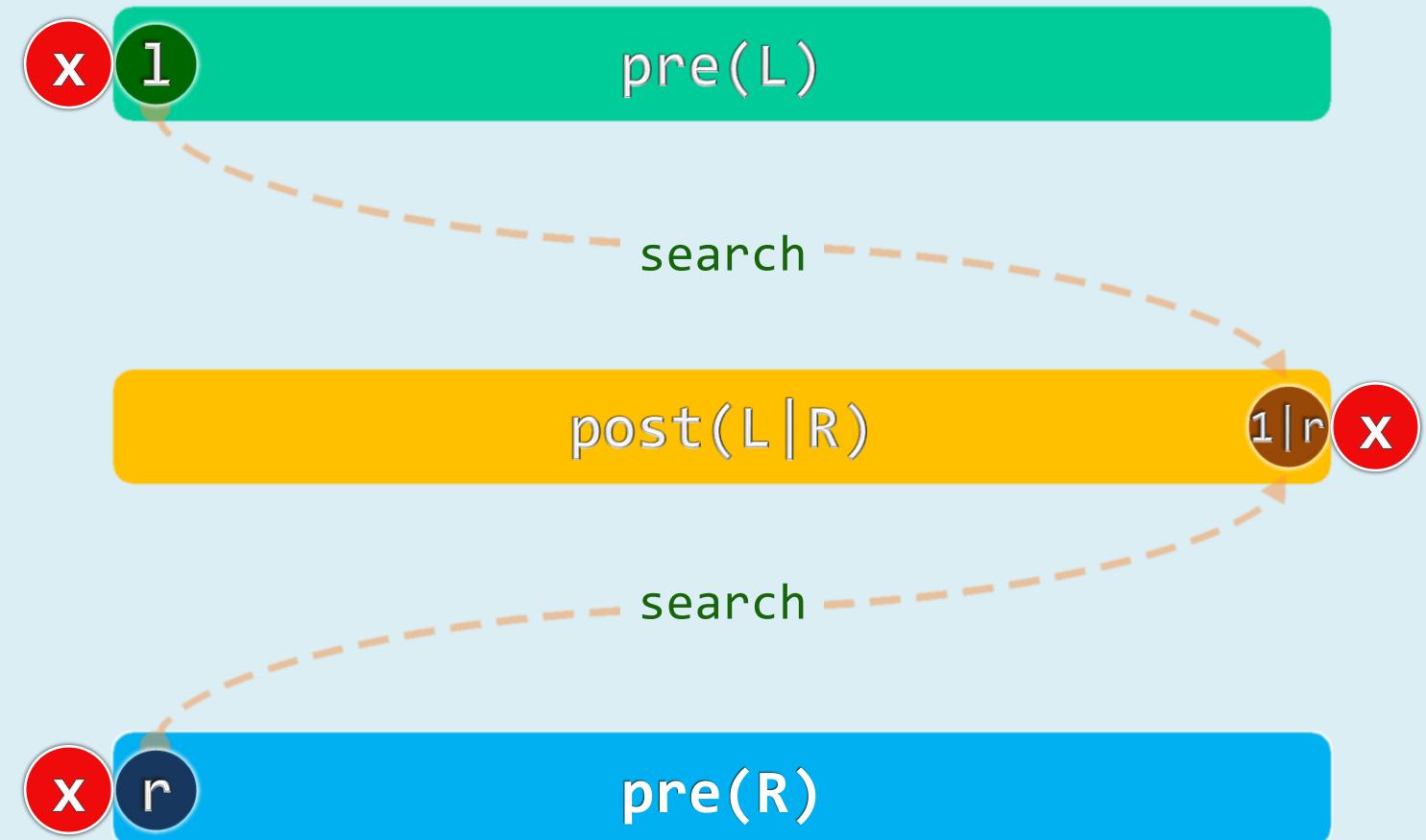
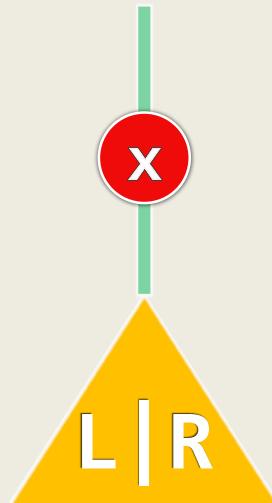
[先序 + 后序] ?



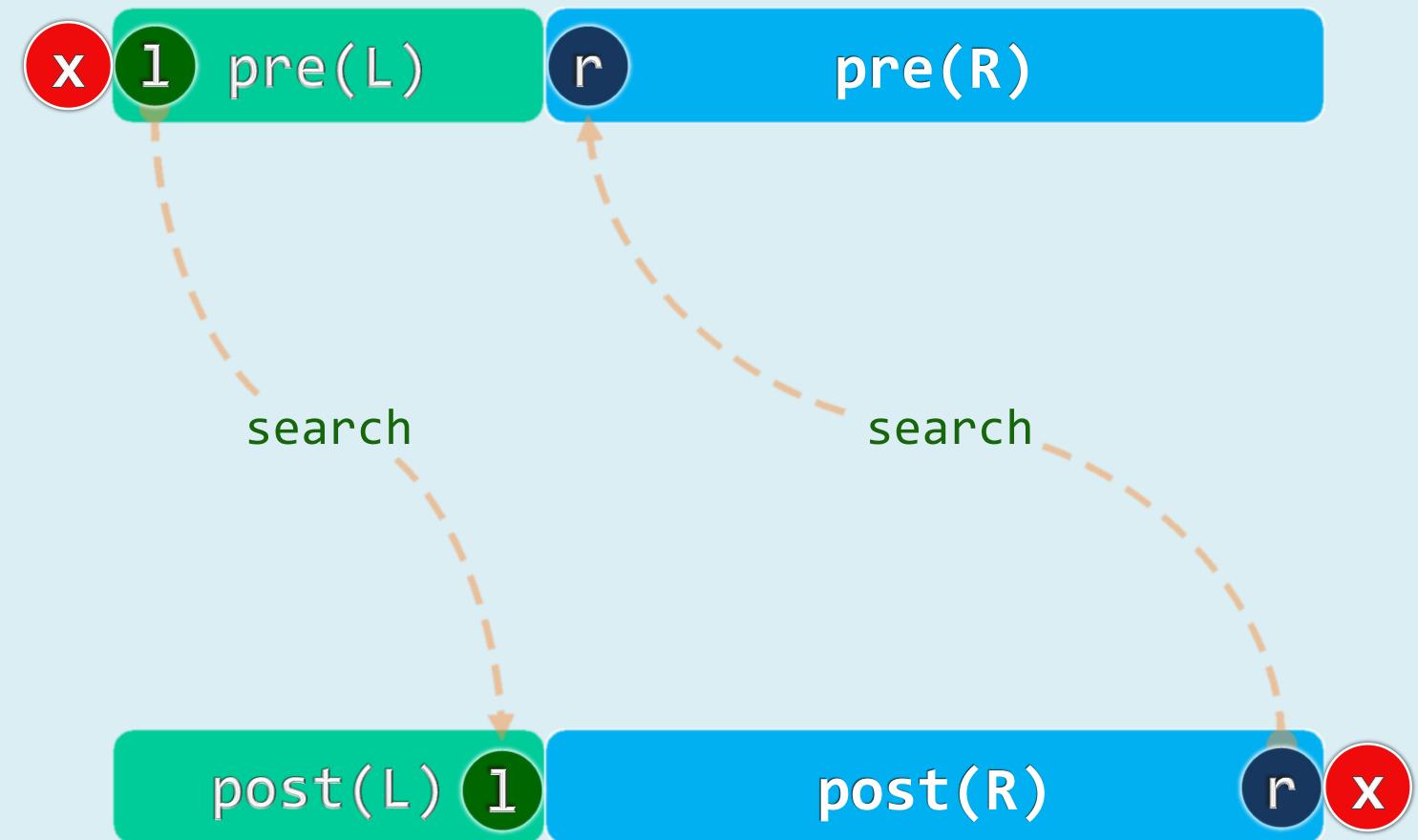
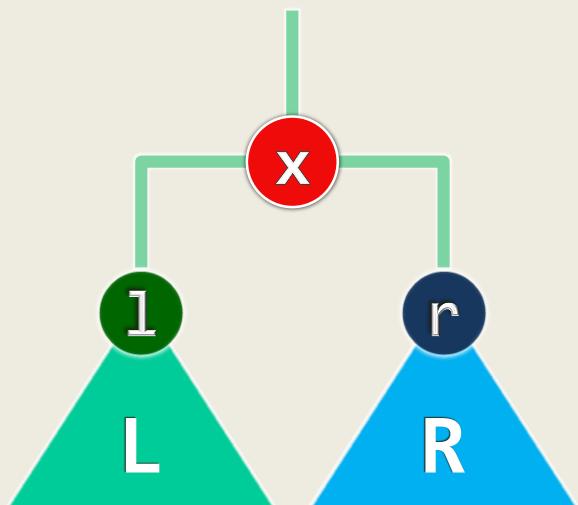
[先序 + 后序] !



[先序 + 后序] ? ?



[先序 + 后序] × 真!



增强序列

❖ 假想地认为，每个NULL也是“真实”节点，并在遍历时一并输出

每次递归返回，同时输出一个事先约定的元字符“^”

❖ 若将遍历序列表示为一个Iterator，则可将其定义为

```
Vector< BinNode<T> * >
```

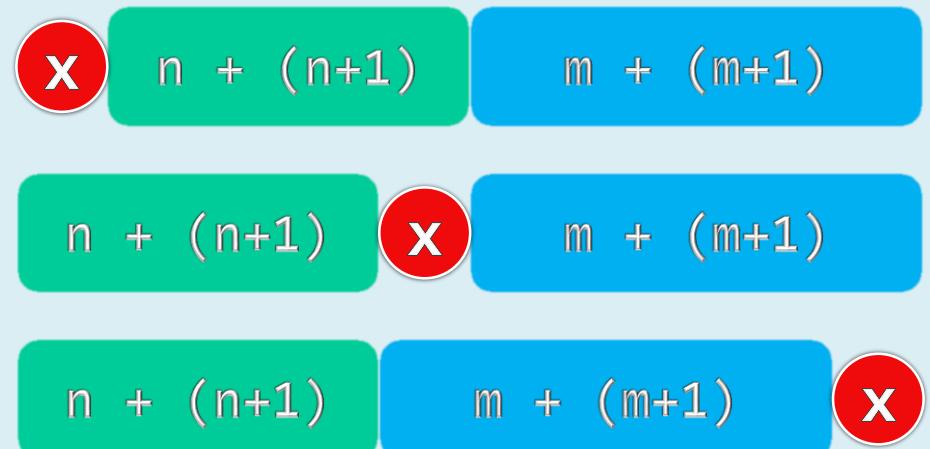
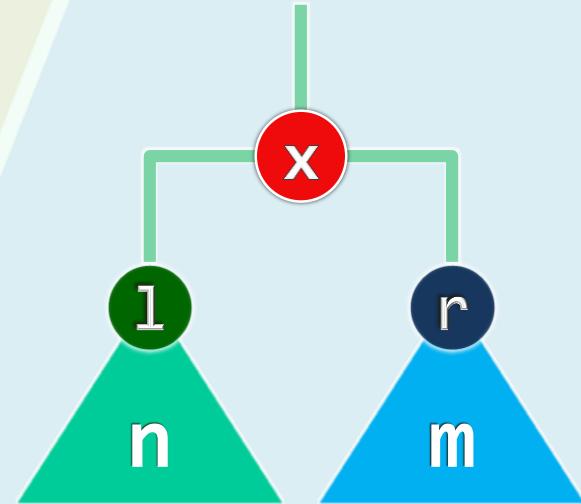
于是在增强的遍历序列中，这类“节点”可统一记作NULL

❖ 可归纳证明：在增强的先序、后序遍历序列中

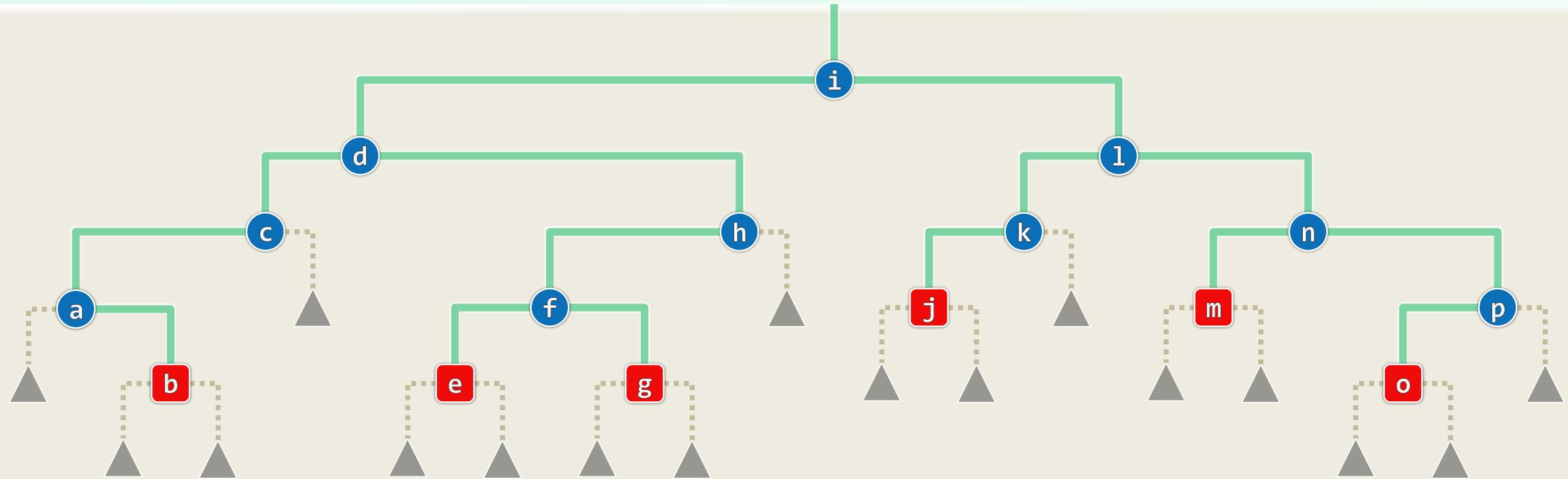
1) 任一子树依然对应于一个子序列，而且

2) 其中的NULL节点恰比非NULL节点多一个

❖ 如此，通过对增强序列分而治之，即可重构原树



增强序列：实例



preorder : i d c a ^ b ^ ^ ^ h f e ^ ^ ^ g ^ ^ ^ l k j ^ ^ ^ n m ^ ^ ^ p o ^ ^ ^

inorder : ^ a ^ b ^ c ^ d ^ e ^ f ^ g ^ h ^ i ^ j ^ k ^ l ^ m ^ n ^ o ^ p ^

postorder : ^ ^ ^ b a ^ c ^ ^ e ^ ^ g f ^ h d ^ ^ j ^ k ^ ^ m ^ ^ o ^ p n l i

二叉树

Huffman编码树：算法

05

JJ

句读之不知，惑之不解，或师焉，或不焉，小学而大遗，吾未见其明也

两年的时间，在你看来，也许就是一眨眼的功夫，对不对？可对我来说，
它实在长得没边。我用不着为两年后的事情操心。

邓俊辉

deng@tsinghua.edu.cn

编码

❖ 通讯 / 编码 / 译码

❖ 二进制编码

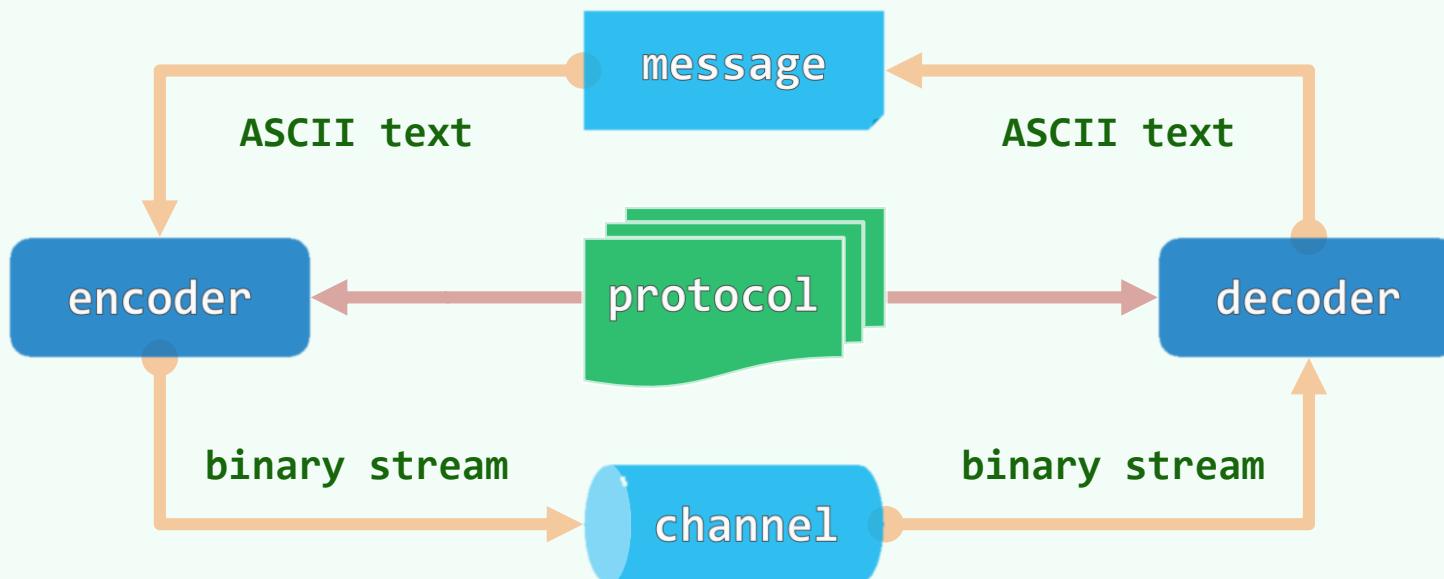
- 组成数据文件的字符来自字符集 Σ
- 字符被赋予互异的二进制串

❖ 文件的大小取决于

- 字符的数量 \times 各字符编码的长短

❖ 通讯带宽有限时

- 如何对各字符编码，使文件最小？



1 010 011 00 ~ MAIN

M	A	I	N
1	010	011	00

PFC编码

❖ 将 Σ 中的字符组织成一棵二叉树，以0/1表示左/右孩子

各字符 x 分别存放于对应的叶子 $v(x)$ 中

❖ 字符 x 的编码串 $rps(v(x)) = rps(x)$

由根到 $v(x)$ 的通路 (root path) 确定

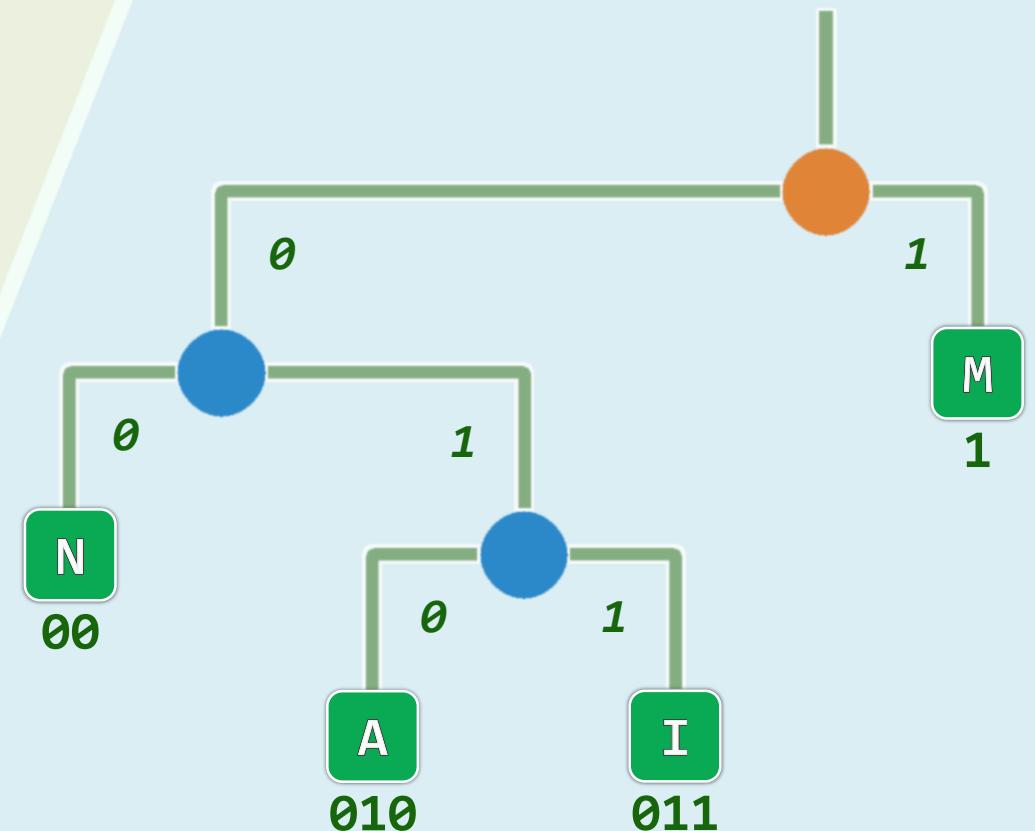
❖ 字符编码不必等长，而且...

❖ 不同字符的编码互不为前缀，故不致歧义

(Prefix-Free Code)

❖ 缺点：你能发现吗？

$1010_01100 \sim \text{MAIN}$



编码长度 vs. 叶节点平均深度

❖ 平均编码长度

$$\text{ald}(T) = \sum_{x \in \Sigma} \text{depth}(v(x)) / |\Sigma|$$

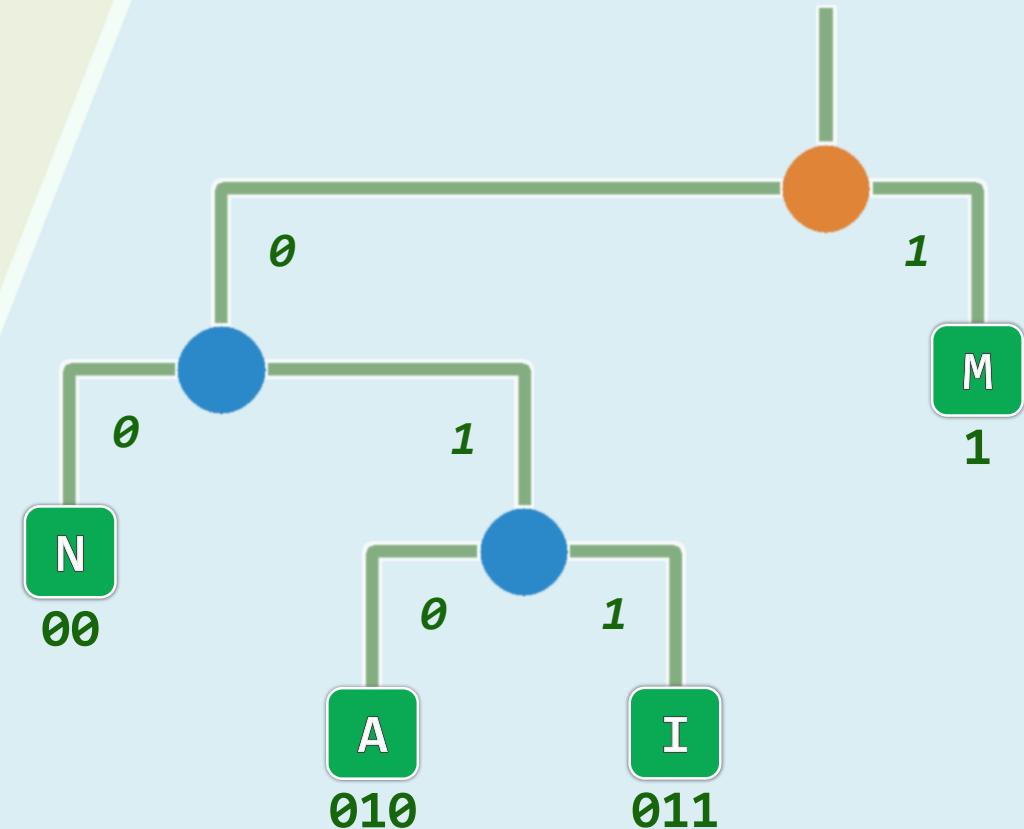
❖ 对于特定的字符集 Σ

$\text{ald}()$ 最小者即为最优编码树 T_{opt}

❖ 最优编码树必然存在，但不见得唯一

它们具有哪些特征？

$1^0 10_0 11^0 0 \sim \text{MAIN}$



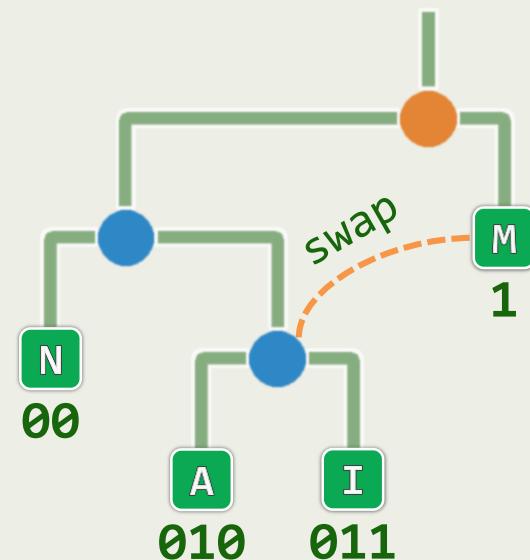
最优编码树

$$\forall v \in T_{\text{opt}}, \deg(v) = 0 \text{ only if } \text{depth}(v) \geq \text{depth}(T_{\text{opt}}) - 1$$

亦即，叶子只能出现在倒数两层以内——否则，通过节点交换即可...

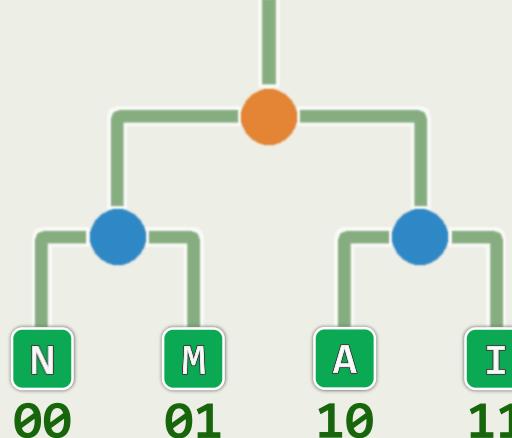
$$\text{ald}(T)*4 = 2+3+3+1 = 9$$

"1⁰100₁₁⁰⁰" = "MAIN"



$$\text{ald}(T)*4 = 2+2+2+2 = 8$$

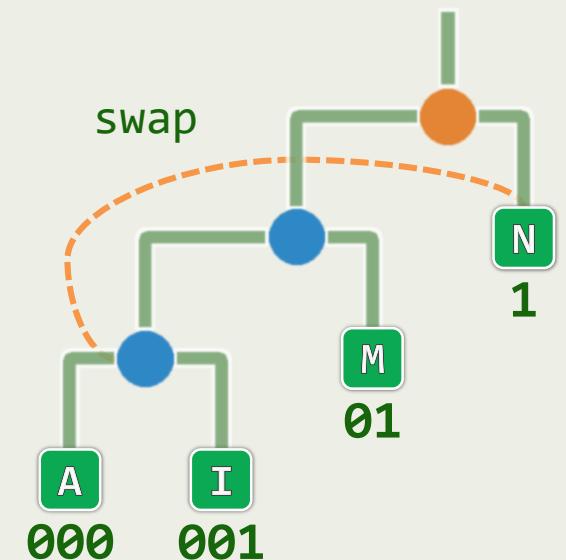
"01¹0₁₁⁰⁰" = "MAIN"



特别地，真完全树即是**最优编码树**

$$\text{ald}(T)*4 = 2+3+3+1 = 9$$

"01⁰00₀¹" = "MAIN"



字符频率：不 ~ 埠

❖ 实际上，字符的出现概率或频度不尽相同

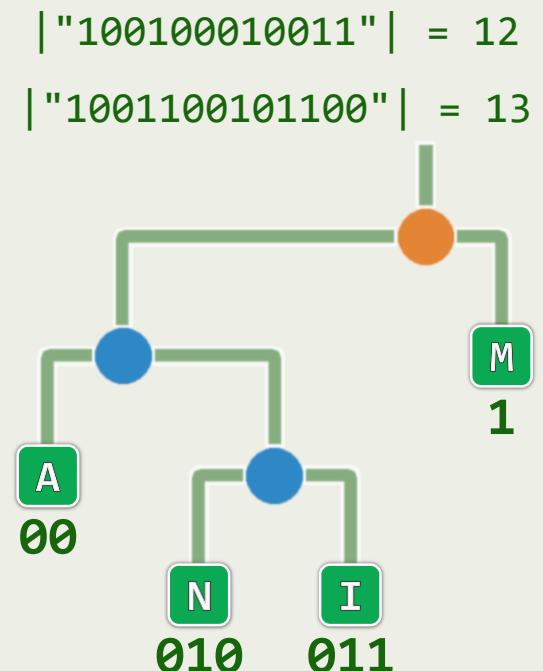
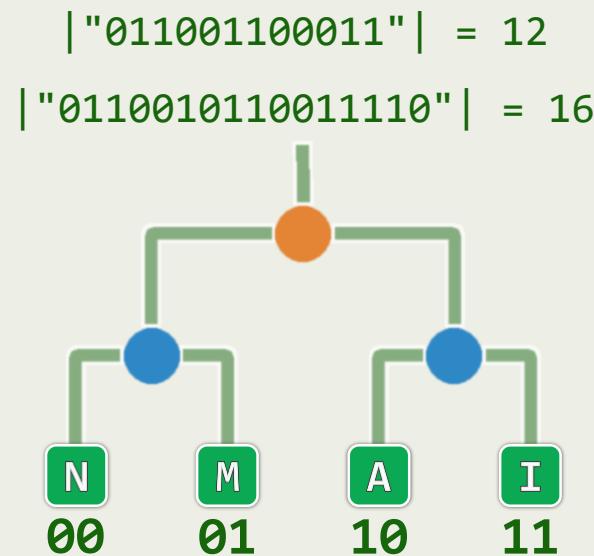
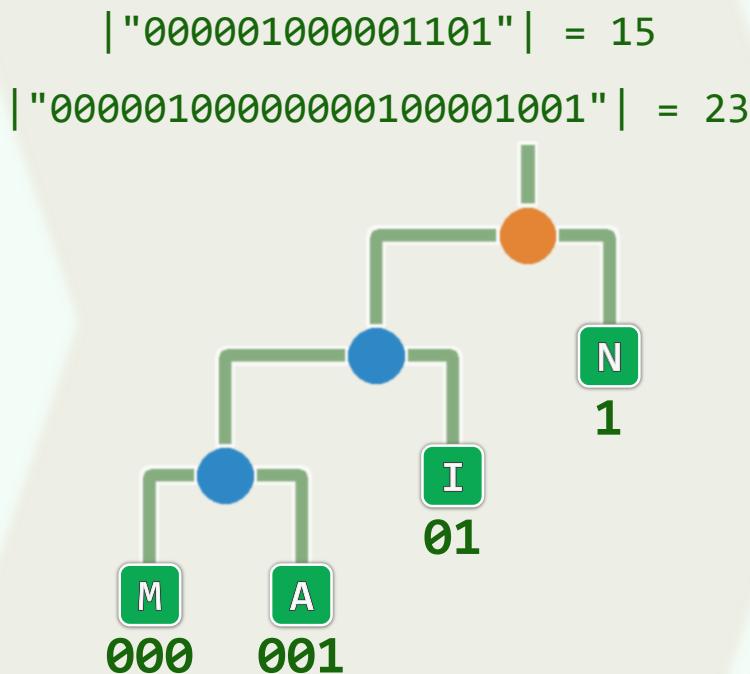
甚至，往往相差极大...



❖ 已知各字符的期望频率，如何构造最优编码树？

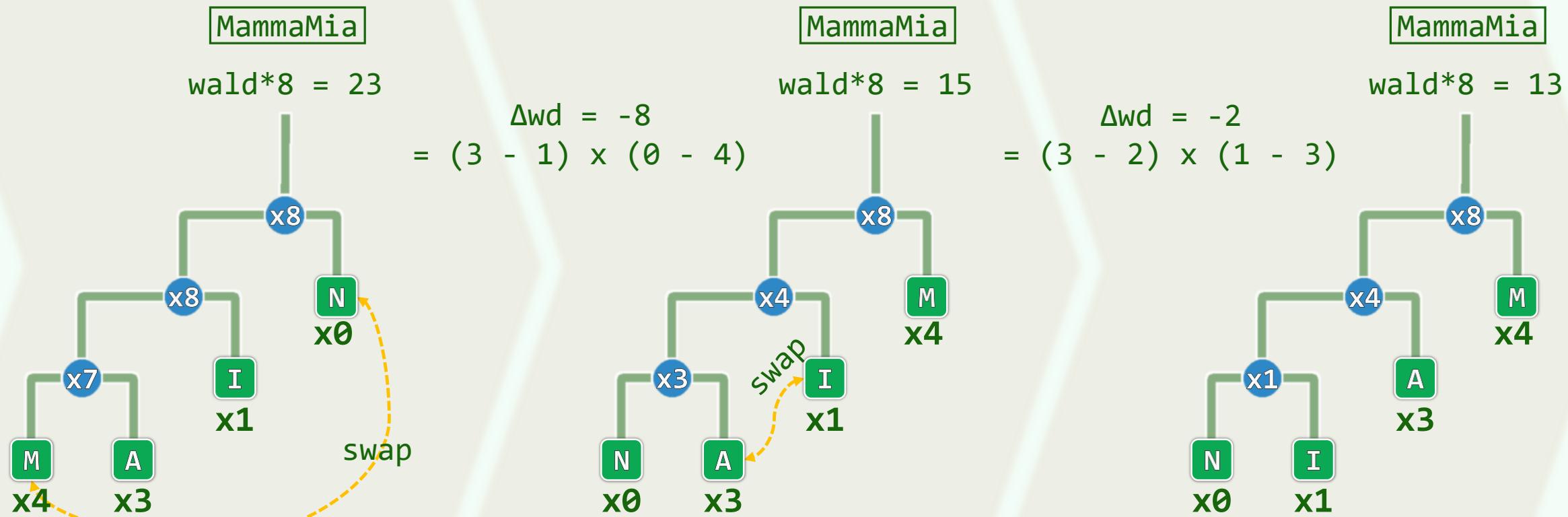
带权编码长度 vs. 叶节点平均带权深度

- ❖ 文件长度 \propto 平均带权深度 $wald(T) = \sum_x rps(x) \times w(x)$
- ❖ 此时，完全树未必就是最优编码树——比如，考查 "mamani" 和 "mammamia" ...



最优带权编码树

- 同样，频率高/低的（超）字符，应尽可能放在高/低处
- 故此，通过适当交换，同样可以缩短 $wald(T)$



Huffman的贪心策略：频率**低**的字符优先引入，其位置亦更低

为每个**字符**创建一棵单节点的树，组成**森林F**

按照出现频率，对所有树**排序**

while (F中的树不止一棵)

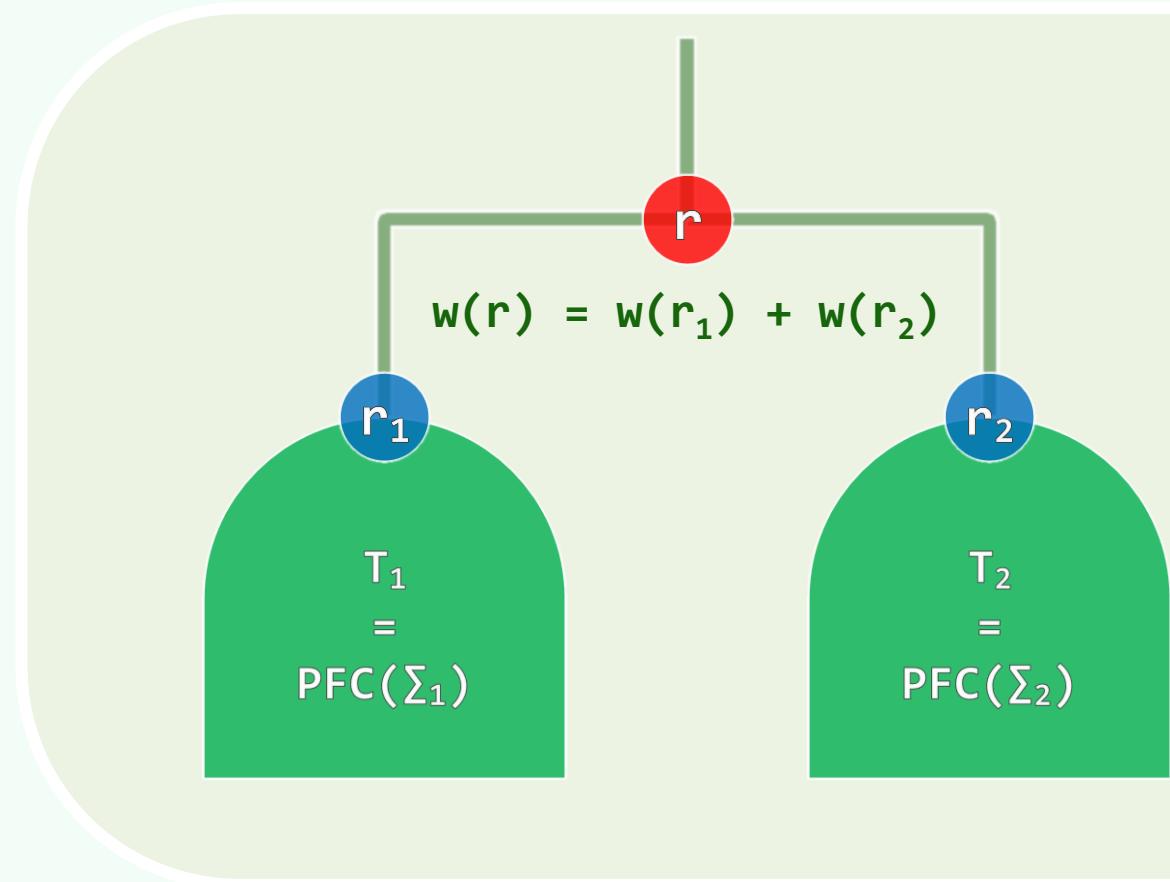
取出**频率最小**的两棵树： T_1 和 T_2

将它们**合并**成一棵新树T，并令：

$$lc(T) = T_1 \text{ 且 } rc(T) = T_2$$

$$w(\ root(T)) = w(\ root(T_1)) + w(\ root(T_2))$$

//尽管贪心策略未必总能得到最优解，但非常幸运，如上算法的确能够得到最优编码树之一



二叉树

Huffman编码树：正确性

05-J2

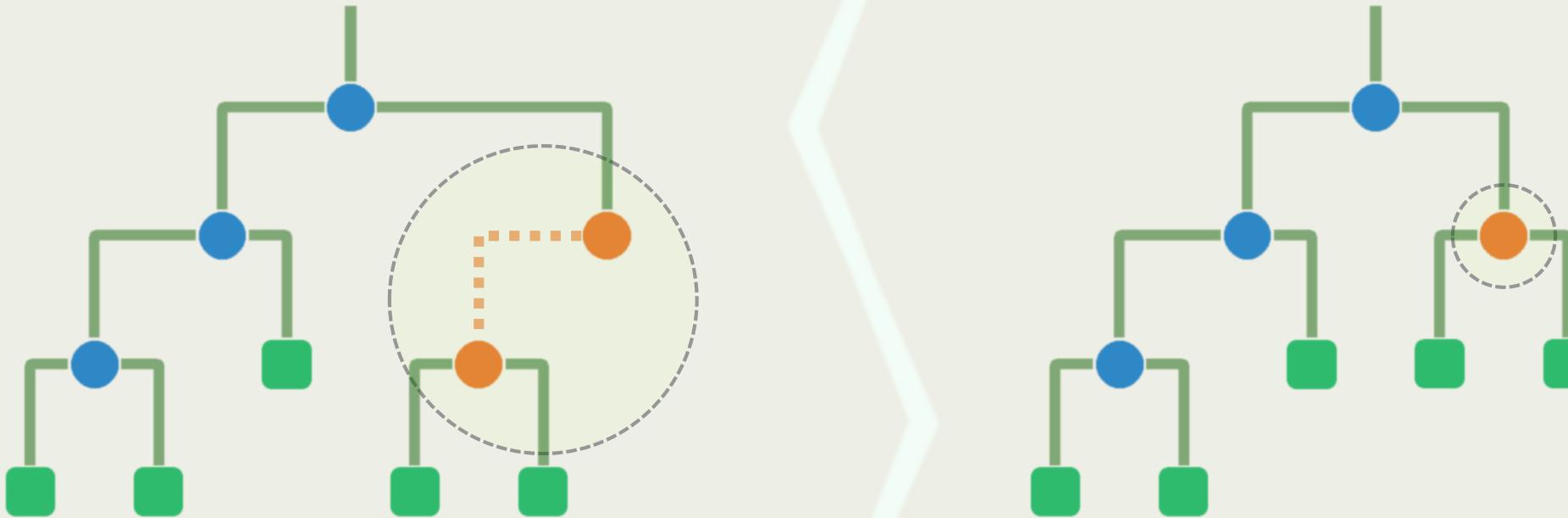
我生来就不像我所见过的任何一个人；我敢断言，我与世上的任何一个人都迥然不同；虽说我不比别人好，但至少我与他们完全两样。

邓俊辉

deng@tsinghua.edu.cn

双子性

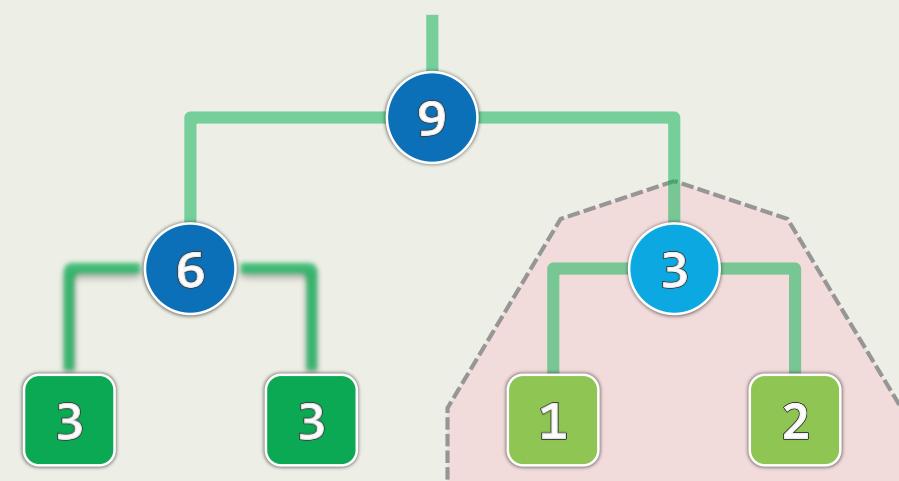
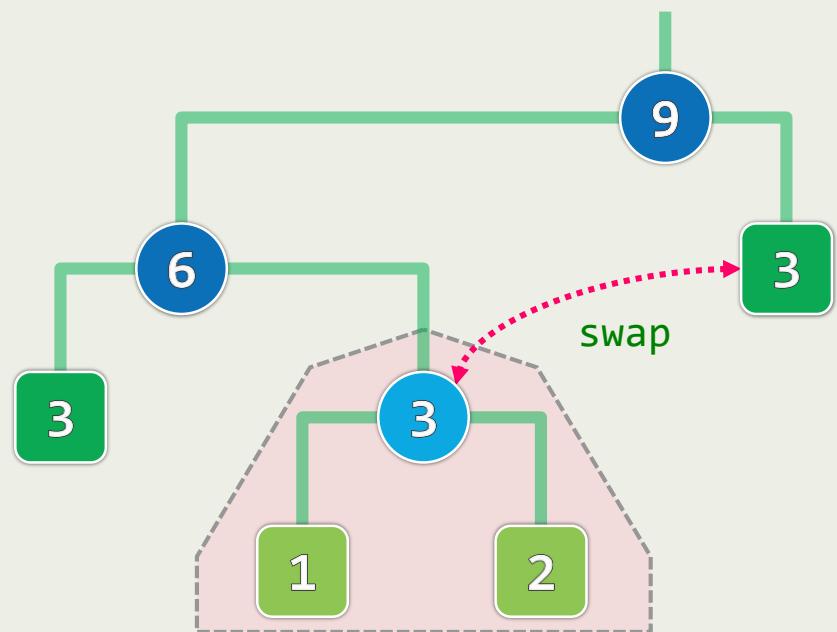
- ❖ 最优编码树有何特征?
- ❖ 首先, 每一内部节点都有两个孩子——节点度数均为偶数 (0或2), 即真二叉树
- ❖ 否则, 将1度节点替换为其唯一的孩子, 则新树的wald将更小



不唯一性

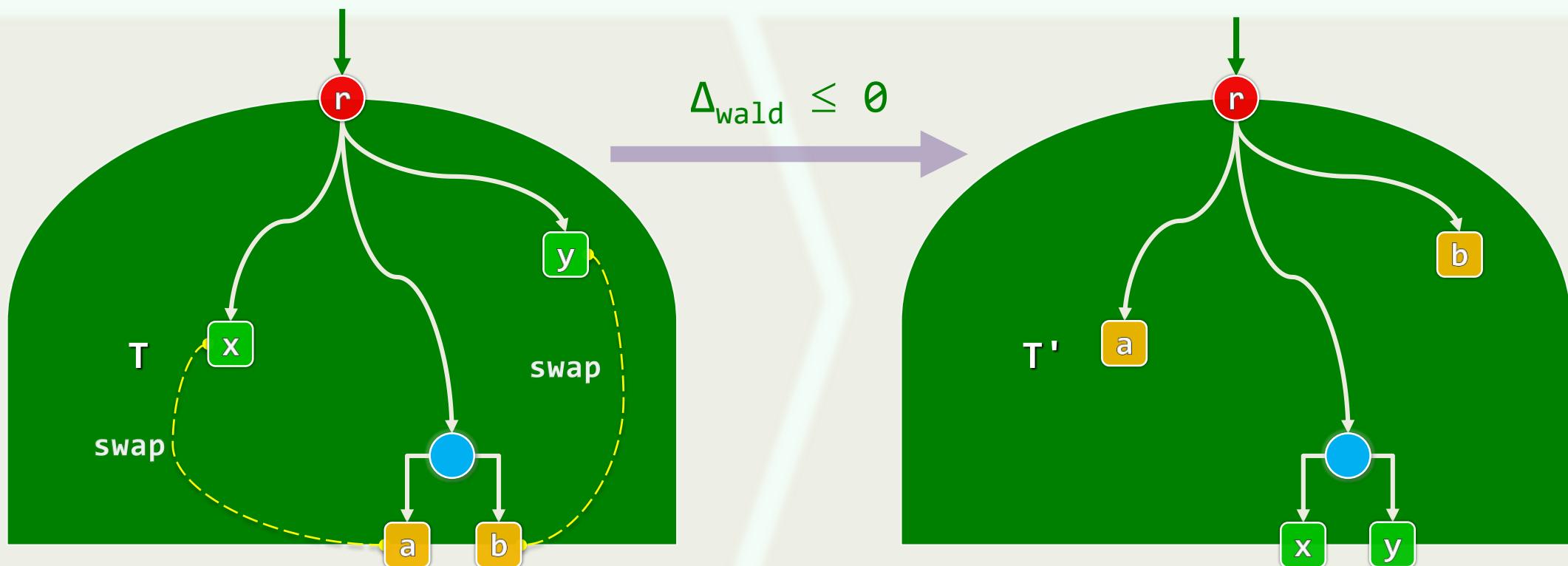
- ❖ 对任一内部节点而言
左、右子树互换之后wald不变
- ❖ 上述算法中，兄弟子树的次序系随机选取
故有可能...

- ❖ 为消除这种歧义，可以（比如）
明确要求左子树的频率更低
- ❖ 不过，倘若
它们（甚至更多节点）的频率恰好相等...



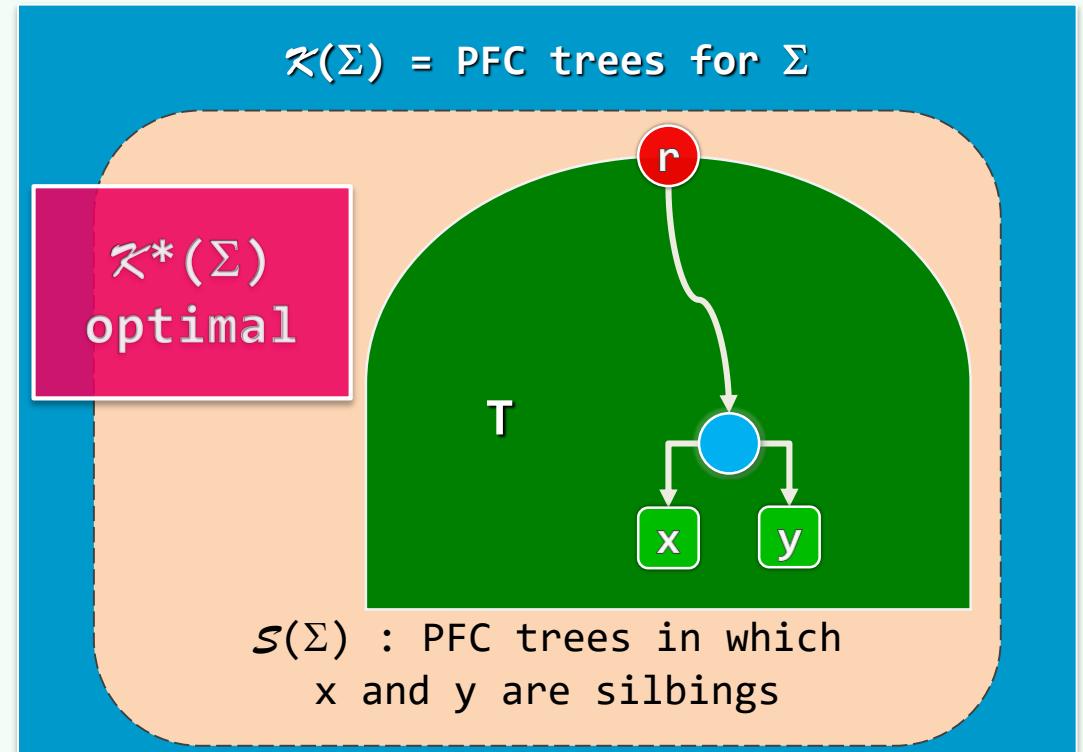
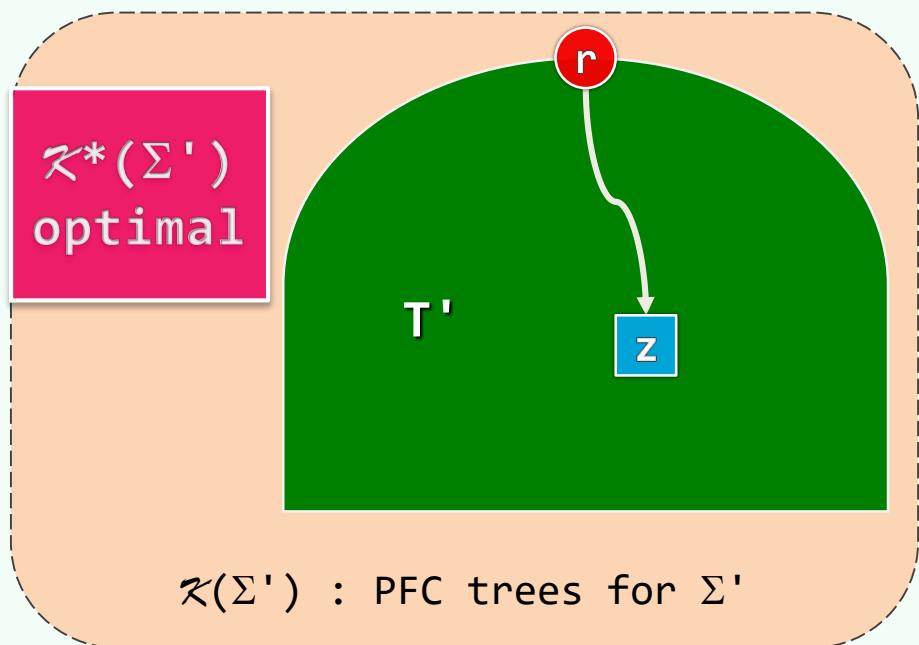
层次性

- ❖ 出现频率最低的字符 x 和 y ，必在某棵最优编码树中处于最底层，且互为兄弟
- ❖ 否则，任取一棵最优编码树，并在其最底层任取一对兄弟 a 和 b
于是， a 和 x 、 b 和 y 交换之后，wald 绝不会增加



数学归纳

- ❖ 对 $|\Sigma|$ 做归纳可证：Huffman 算法所生成的，必是一棵最优编码树！ $|\Sigma| = 2$ 时显然
- ❖ 设算法在 $|\Sigma| < n$ 时均正确。现设 $|\Sigma| = n$ ，取 Σ 中频率最低的 x 、 y （不妨就设二者互为兄弟）
- ❖ 令： $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$ ， $w(z) = w(x) + w(y)$

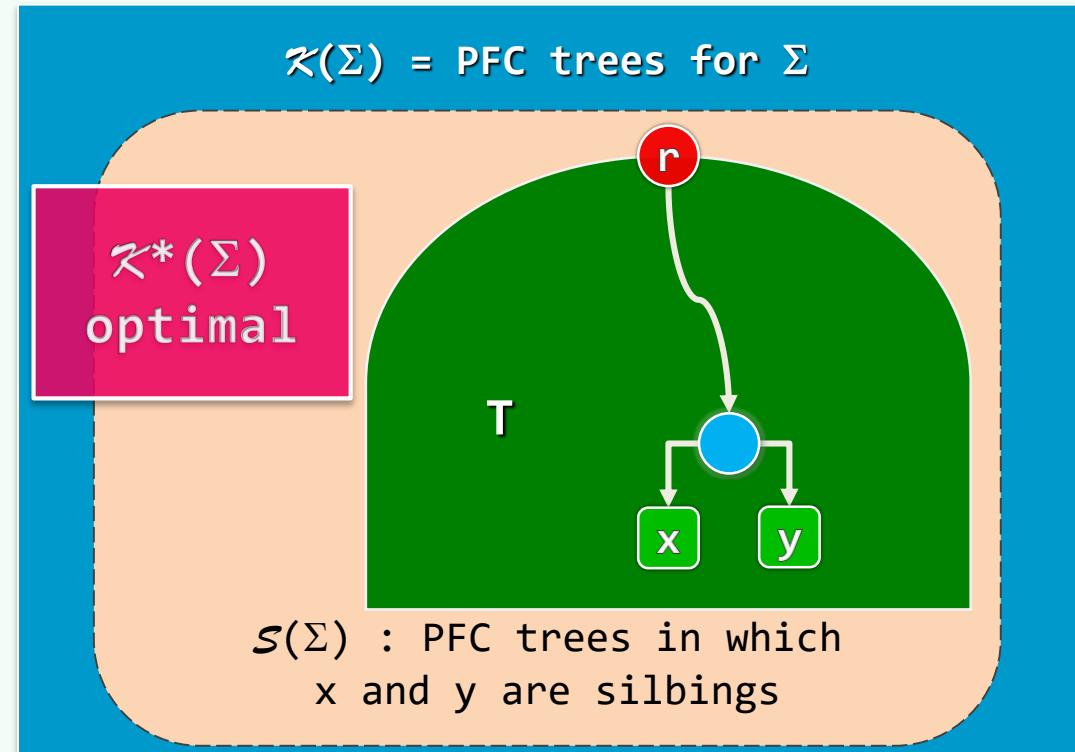
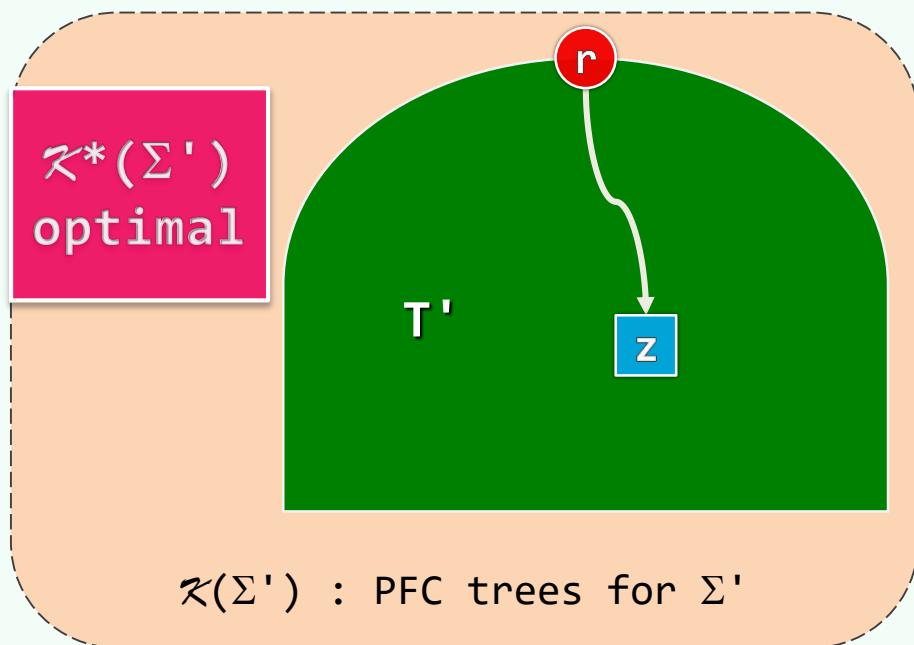


定差

❖ 对于 Σ' 的任一编码树 T' ，只要为 z 添加孩子 x 和 y ，即可得到 Σ 的一棵编码树 T ，且

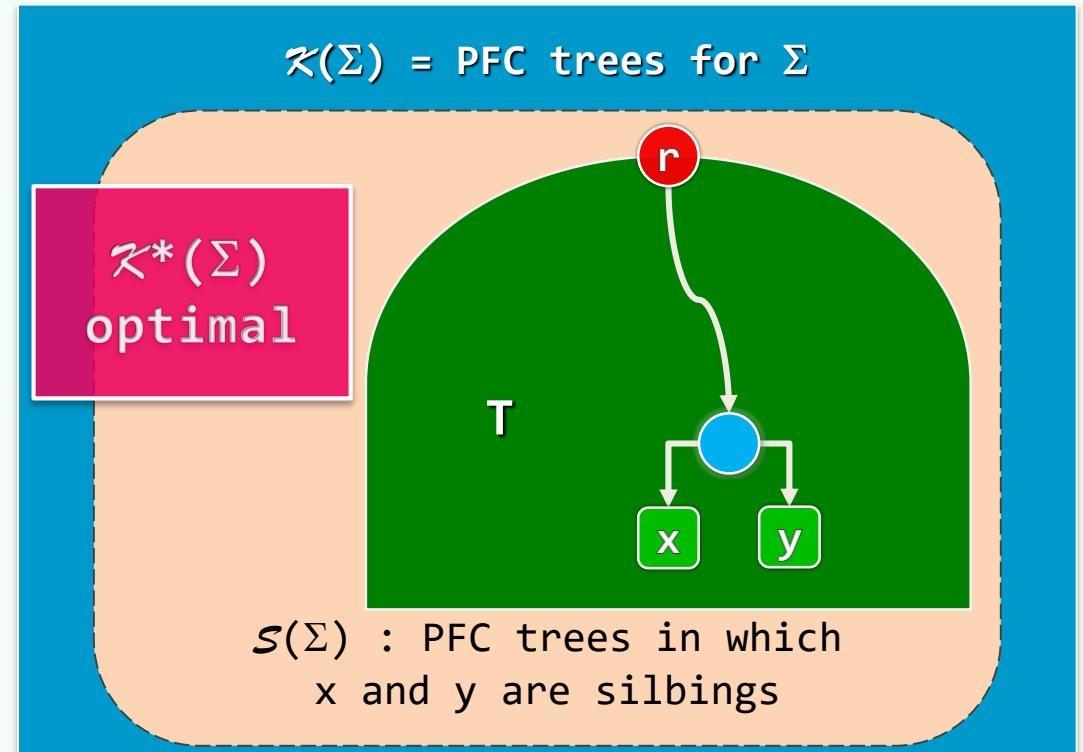
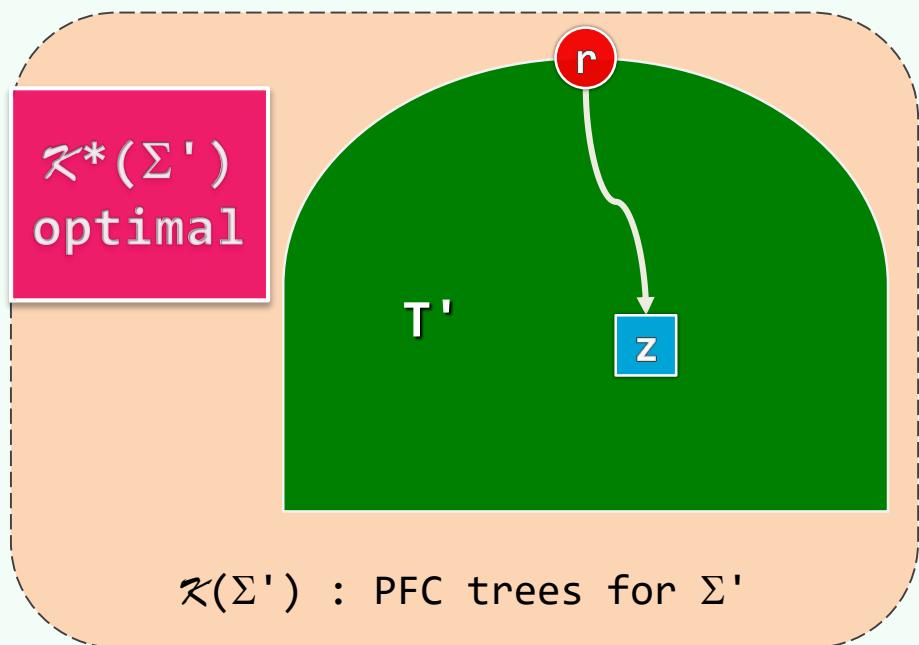
$$wd(T) - wd(T') = w(x) + w(y) = w(z)$$

❖ 可见，如此对应的 T 和 T' ， wd 之差与 T 的具体形态无关



从最优，到最优

- ❖ 因此，只要 T' 是 Σ' 的最优编码树，则 T 也必是 Σ 的最优编码树（之一）
- ❖ 实际上，Huffman算法的过程，与上述归纳过程完全一致
——每一步迭代都可视作，从某棵 T 转入对应的 T'



二叉树

Huffman编码树：算法实现

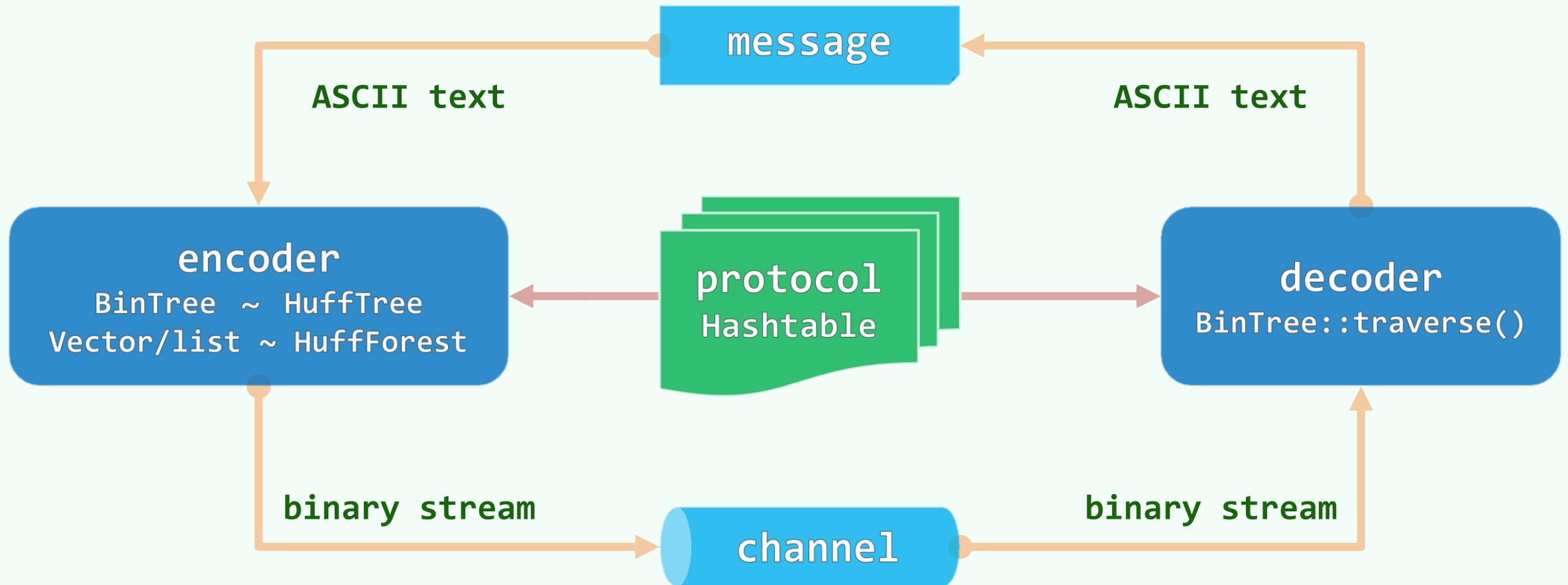
05-J3

邓俊辉

deng@tsinghua.edu.cn

树形建筑也出现了，看上去规模与地球上的差不多，
只是挂在树上的建筑叶子更为密集。

数据结构与算法



Huffman (超) 字符

```
#define N_CHAR (0x80 - 0x20) //仅以可打印字符为例
```

```
struct HuffChar { //Huffman (超) 字符
```

```
char ch; unsigned int weight; //字符、频率
```

```
HuffChar ( char c = '^', unsigned int w = 0 ) : ch ( c ), weight ( w ) {};
```

```
bool operator< ( HuffChar const& hc ) { return weight > hc.weight; } //比较器
```

```
bool operator== ( HuffChar const& hc ) { return weight == hc.weight; } //判断器
```

```
};
```

Huffman树与森林

❖ Huffman (子) 树

```
using HuffTree = BinTree< HuffChar >;
```

❖ Huffman森林

```
using HuffForest = List< HuffTree* >;
```

❖ 待日后掌握了更多数据结构之后，可改用更为高效的方式，比如：

```
using HuffForest = PQ_List< HuffTree* >; //基于列表的优先级队列
```

```
using HuffForest = PQ_ComplHeap< HuffTree* >; //完全二叉堆
```

```
using HuffForest = PQ_LeftHeap< HuffTree* >; //左式堆
```

❖ 得益于已定义的统一接口，支撑Huffman算法的这些底层数据结构可直接彼此替换

构造编码树：反复合并二叉树

```
HuffTree* generateTree( HuffForest * forest ) { //Huffman编码算法
    while ( 1 < forest->size() ) { //反复迭代，直至森林中仅含一棵树
        HuffTree *T1 = minHChar( forest ), *T2 = minHChar( forest );
        HuffTree *S = new HuffTree(); //创建新树，然后合并T1和T2
        S->insert( HuffChar('^', T1->root()->data.weight + T2->root()->data.weight) );
        S->attach( T1, S->root() ); S->attach( S->root(), T2 );
        forest->insertAsLast( S ); //合并之后，重新插回森林
    } //assert: 森林中最终唯一的那棵树，即Huffman编码树
    return forest->first()->data; //故直接返回之
}
```

查找最小超字符：遍历List/Vector

```
HuffTree* minHChar( HuffForest* forest ) { //此版本仅达到 $\mathcal{O}(n)$ , 故整体为 $\mathcal{O}(n^2)$ 
    ListNodePosi<HuffTree*> m = forest->first(); //从首节点出发, 遍历所有节点
    for ( ListNodePosi<HuffTree*> p = m->succ; forest->valid( p ); p = p->succ )
        if( m->data->root()->data.weight > p->data->root()->data.weight ) //不断更新
            m = p; //找到最小节点 (所对应的Huffman子树)
    return forest->remove( m ); //从森林中取出该子树, 并返回
} //Huffman编码的整体效率, 直接决定于minHChar()的效率
```

构造编码表：遍历二叉树

```
#include "Hashtable.h" //用HashTable (第09章) 实现

using HuffTable = Hashtable< char, char* >; //Huffman编码表

static void generateCT //通过遍历获取各字符的编码
( Bitmap* code, int length, HuffTable* table, BinNodePosi<HuffChar> v ) {
    if ( IsLeaf( * v ) ) //若是叶节点 (还有多种方法可以判断)
        { table->put( v->data.ch, code->bits2string( length ) ); return; }

    if ( HasLChild( * v ) ) //Left = 0, 深入遍历
        { code->clear( length ); generateCT( code, length + 1, table, v->lc ); }

    if ( HasRChild( * v ) ) //Right = 1
        { code->set( length ); generateCT( code, length + 1, table, v->rc ); }

} //总体 $\Theta(n)$ 
```

二叉树

Huffman编码树：便捷的改进

05 - J4

邓俊辉

deng@tsinghua.edu.cn

拿中国的情形来说，我们所依靠的不过是小米加步枪，但历史最后将证明，这小米加步枪比蒋介石的飞机加坦克还要强些。

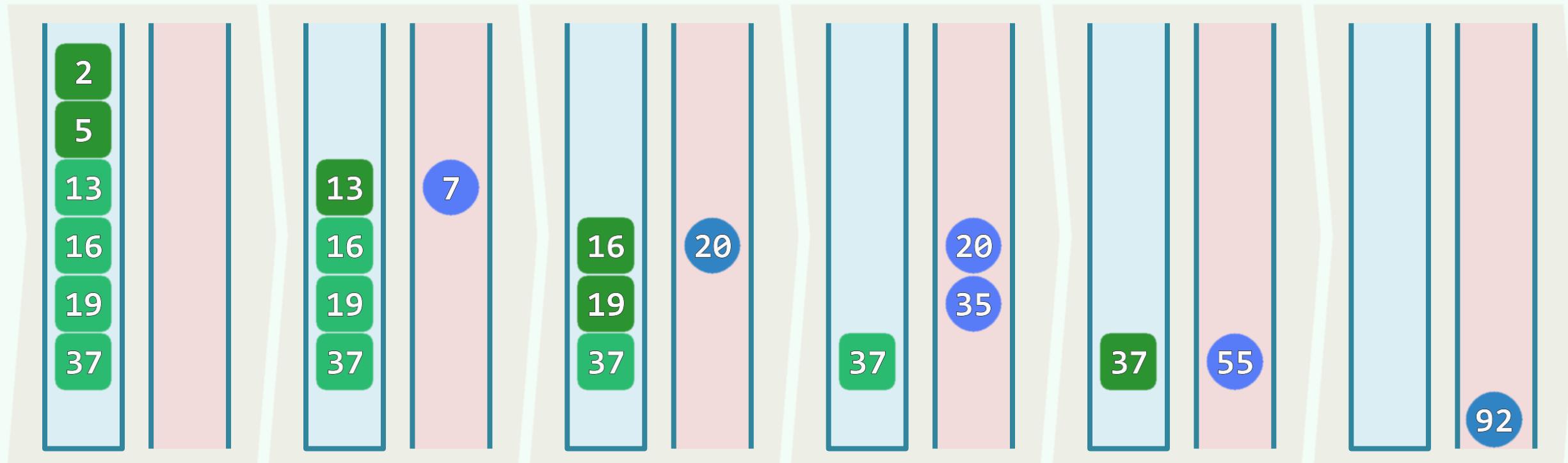
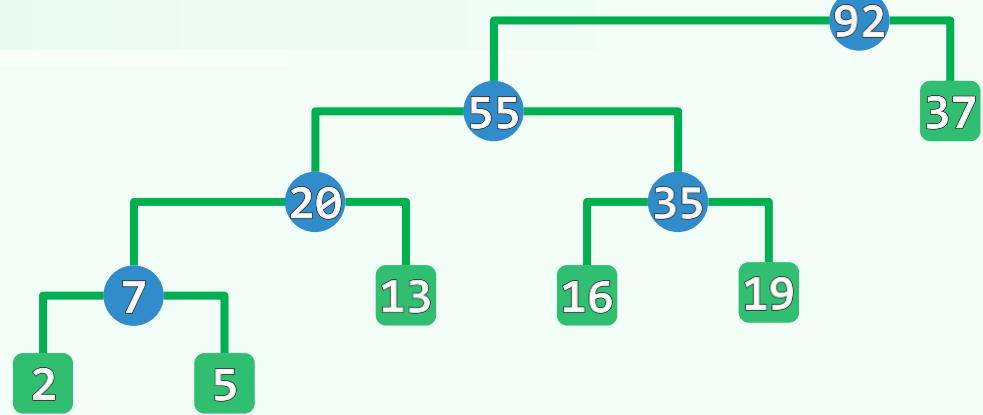
向量 + 列表 + 优先级队列

- ❖ 方案1:
 - 初始化时, 通过排序得到一个**非升序向量** $\text{ // } \mathcal{O}(n \log n)$
 - $\mathcal{O}(n^2)$ - 每次 (从**后端**) 取出频率最低的两个节点 $\text{ // } \mathcal{O}(1)$
 - 将合并得到的新树插入向量, 并保持有序 $\text{ // } \mathcal{O}(n)$
- ❖ 方案2:
 - 初始化时, 通过排序得到一个**非降序列表** $\text{ // } \mathcal{O}(n \log n)$
 - $\mathcal{O}(n^2)$ - 每次 (从**前端**) 取出频率最低的两个节点 $\text{ // } \mathcal{O}(1)$
 - 将合并得到的新树插入列表, 并保持有序 $\text{ // } \mathcal{O}(n)$
- ❖ 方案3:
 - 初始化时, 将所有树组织为一个**优先级队列** (第12章) $\text{ // } \mathcal{O}(n)$
 - $\mathcal{O}(n \log n)$ - 取出频率最低的两个节点, 合并得到的新树插入队列 $\text{ // } \mathcal{O}(\log n) + \mathcal{O}(\log n)$

预排序 x (栈 + 队列)

❖ 方案4: - 所有字符按频率非升序入栈 $\text{// } \mathcal{O}(n \log n)$

$\mathcal{O}(n \log n)$ - 维护另一 (有序) 队列... $\text{// } \mathcal{O}(n)$



二叉树

下界：代数判定树

05

要是你达到一个界线，你不能越过它，那你就会倒霉；
但是你越过了它，也许你会更倒霉。

K7

就好比假设有四样东西，我们正在寻找其中的一个。不管它在哪里，如果我们一开始就知道这样东西在哪里的话，那么剩下的就不是什么问题了。又或者我们可以首先找到其余的三样东西，那么剩下的显然就是第四个了。

邓俊辉

deng@tsinghua.edu.cn

难度与下界

- ❖ 由前述实例可见，同一问题的不同算法，复杂度可能相差悬殊
- ❖ 在可解的前提下，可否谈论问题的**难度**？如何比较不同问题的难度？
- ❖ 问题P若存在算法，则所有算法中**最低**的复杂度称为P的**难度**
- ❖ 为什么要确定问题的**难度**？给定问题P，如何确定其**难度**？
- ❖ 两个方面着手：设计复杂度更低的算法 + 证明更高的问题**难度下界**
- ❖ 一旦算法的复杂度达到**难度下界**，则说明就大O记号的意义而言，算法已经**最优**
- ❖ 例如，排序问题下界为 **$\Omega(n \log n)$** ，而且是**紧的**...

时空性能 + 稳定性

❖ 多种角度估算的时间、空间复杂度

- 最好 / best-case
- 最坏 / worst-case
- 平均 / average-case
- 分摊 / amortized

❖ 其中，对最坏情况的估计最保守、最稳妥

因此，首先应考虑**最坏情况最优**的算法

//worst-case optimal

❖ 排序所需的时间，主要取决于

- 关键码比较次数 / # {key comparison}
- 元素交换次数 / # {data swap}

❖ 就地 (in-place) :

除输入数据本身外，只需 $\mathcal{O}(1)$ 附加空间

❖ 稳定 (stability) :

关键码相同的元素，在排序后相对位置保持

最坏情况最优 + 基于比较

❖ 排序算法，最快能够有多快？

- 语境1：就最坏情况最优而言
- 语境2：就某一大类主流算法而言...

❖ 基于比较的算法 (comparison-based algorithm)

算法执行的进程，取决于一系列的数值（这里即关键码）比对结果

- 比如，`max()` 和 `bubbleSort()`
- ❖ 任何CBA在最坏情况下，都需 $\Omega(n \log n)$ 时间才能完成排序

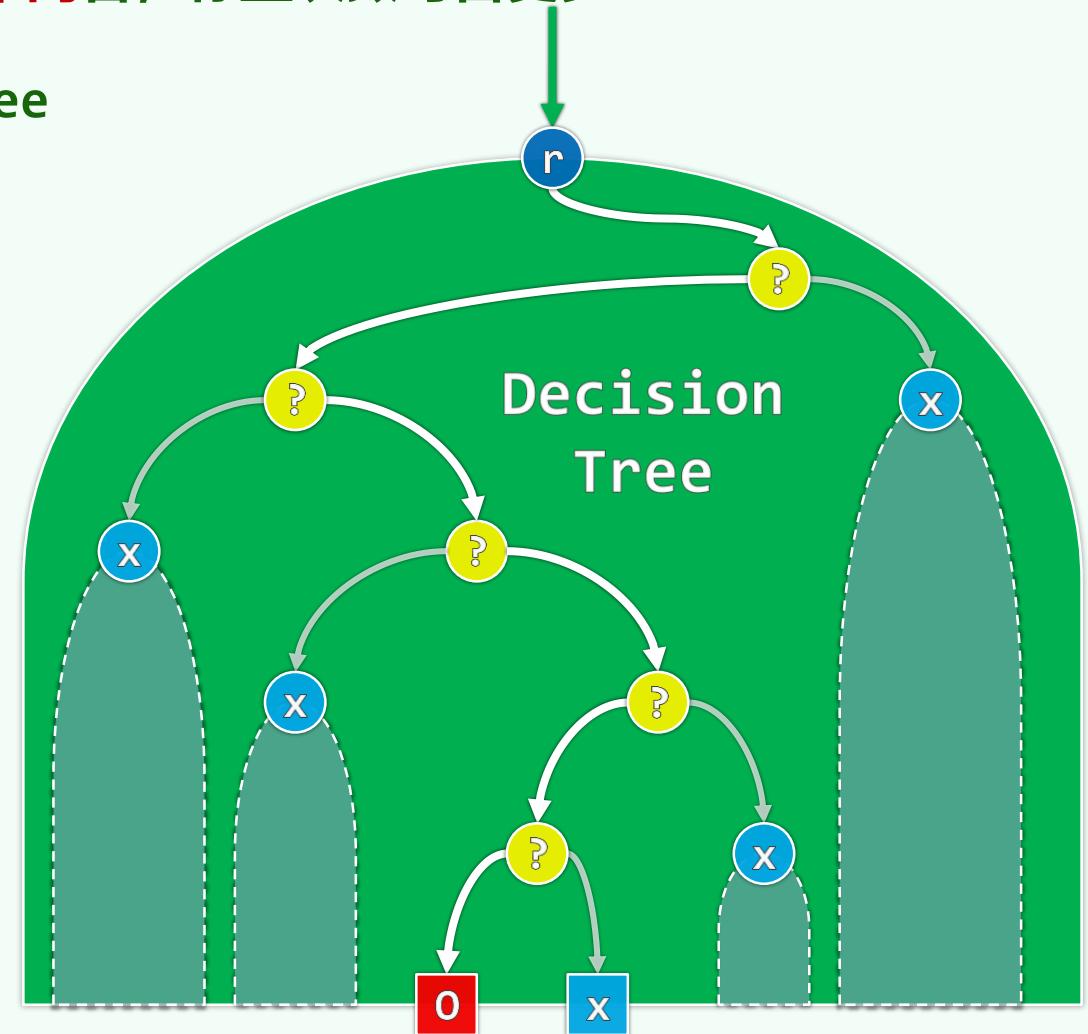
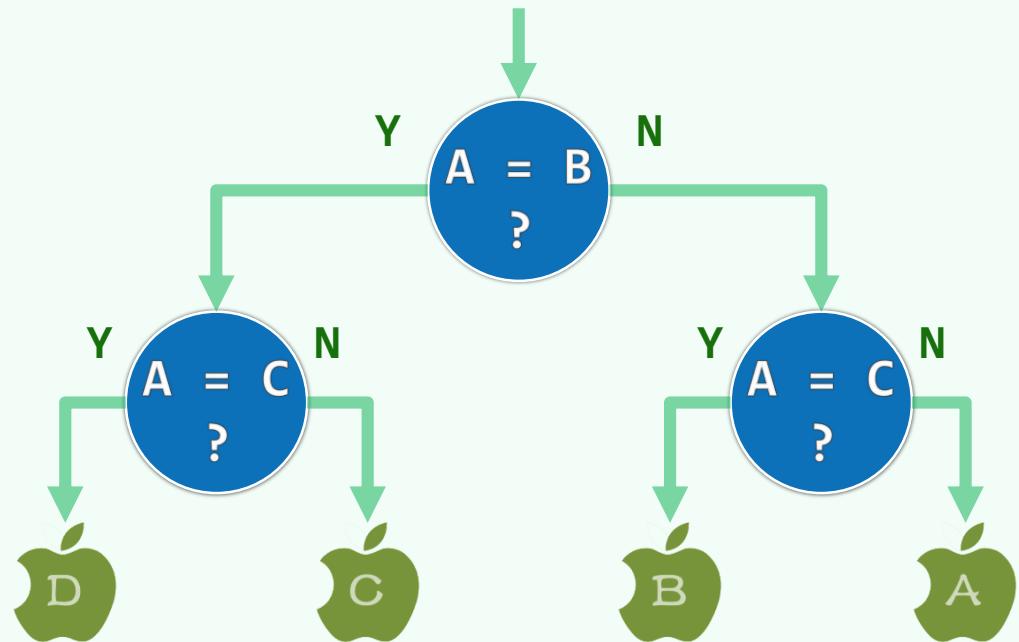
判定树

❖ 经2/4次称量，必可从4/16只苹果中找出唯一的重量不同者；称量次数可否更少？

❖ 每个CBA算法都对应于一棵Algebraic Decision Tree

每一可能的输出，都对应于至少一个叶节点

每一次运行过程，都对应于起始于根的某条路径



代数判定树

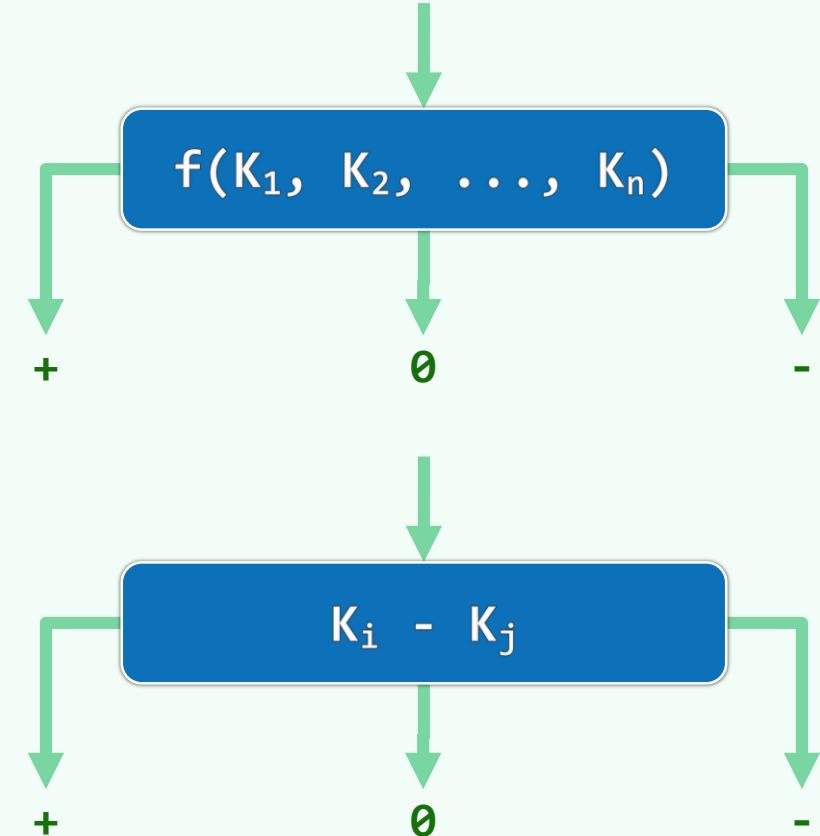
❖ Algebraic Decision Tree

- 针对“比较-判定”式算法的计算模型
- 给定输入的规模，将所有可能的输入所对应的一系列判断表示出来

❖ 代数判定：

- 使用某一常次数代数多项式
将任意一组关键码做为变量，对多项式求值
- 根据结果的符号，确定算法推进方向

❖ Comparison Tree：最简单的ADT，二元一次多项式，形如： $K_i - K_j$



下界: $\Omega(n \log n)$

❖ 比较树是三叉树 (ternary tree)

内部节点至多三个分支 (+ | 0 | -)

❖ 每一叶节点, 各对应于

- 起自根节点的一条通路
- 某一可能的运行过程
- 运行所得的输出

❖ 叶节点深度 ~ 比较次数 ~ 计算成本

❖ 树高 ~ 最坏情况时的计算成本

树高的下界 ~ 所有CBA的时间复杂度下界

❖ 对于排序算法所对应的ADT, 必有 $N \geq n!$

- ADT的每一输出 (叶子), 对应于某一置换
依此置换, 可将输入序列转换为有序序列
- 算法的输出, 须覆盖所有可能的输入

❖ 包含 N 个叶节点的排序算法ADT, 高度不低于

$$\log_3 N \geq \log_3 n!$$

$$= \log_3 e \cdot [n \ln n - n + \mathcal{O}(\ln n)] = \Omega(n \cdot \log n)$$

❖ 这一结论, 还可进一步推广到

理想平均情况、随机情况 (概率 $\geq 25\%$) ...

二叉树

下界：归约

θ₅-K₂

邓俊辉

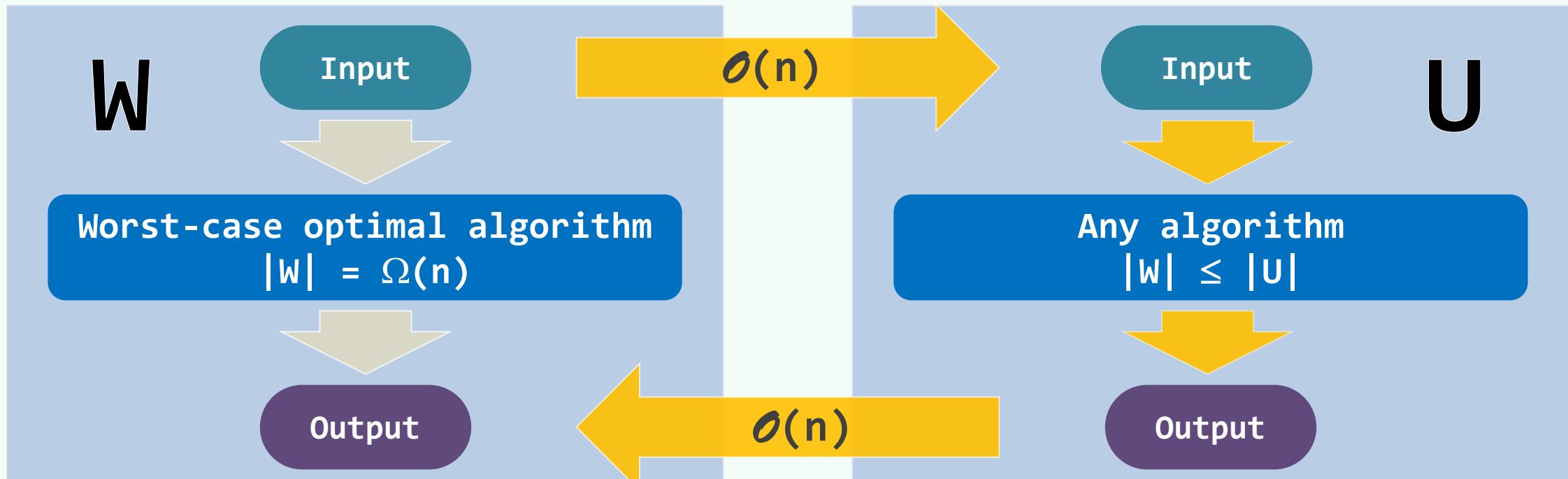
deng@tsinghua.edu.cn

言有易，言无难

不怕不识货，就怕货比货

线性归约 (Linear-Time Reduction)

❖ 除了 (代数) 判定树, 归约 (reduction) 也是确定下界的有力工具



linear-time reduction

NP-complete/Polynomial-time reduction

P-SPACE complete/ Polynomial-time many-one reduction

实例

- ❖ 【Element Uniqueness】 任意n个**实数**中，是否包含雷同？ // $\Omega(n \log n)$
EU \leq_N Closest Pair
- ❖ 【Integer Element Uniqueness】 任意n个**整数**中，是否包含雷同？ // $\Omega(n \log n)$
IEU \leq_N Segment Intersection Detection
- ❖ 【Set Disjointness】 任意一对集合A和B，是否存在**公共元素**？ // $\Omega(n \log n)$
SD \leq_N Diameter
- ❖ 【Red-Blue Matching】 平面上任给n个红色点和n个蓝色点，如何用**互不相交的线段配对**联接
Sorting \leq_N Red-Blue Matching
- ❖ Sorting \leq_N Huffman Tree \leq_N Optimal Encoding Tree

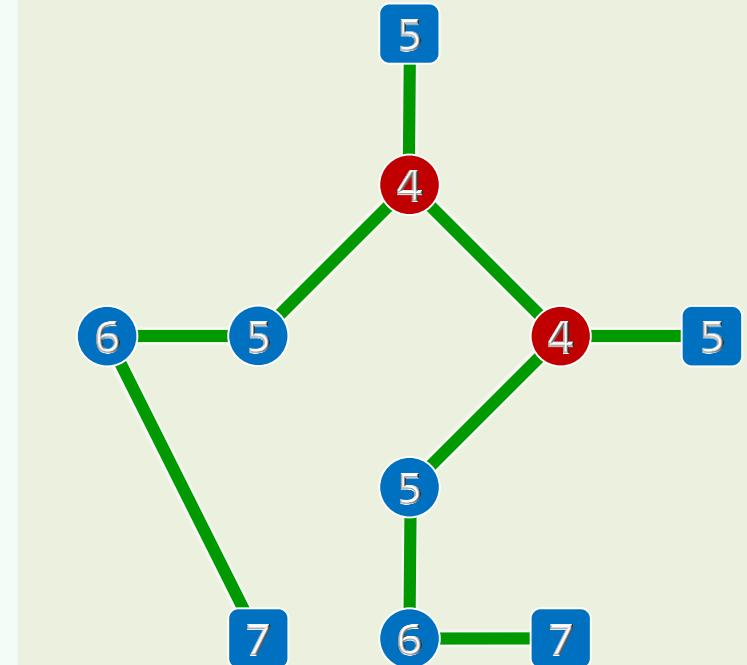
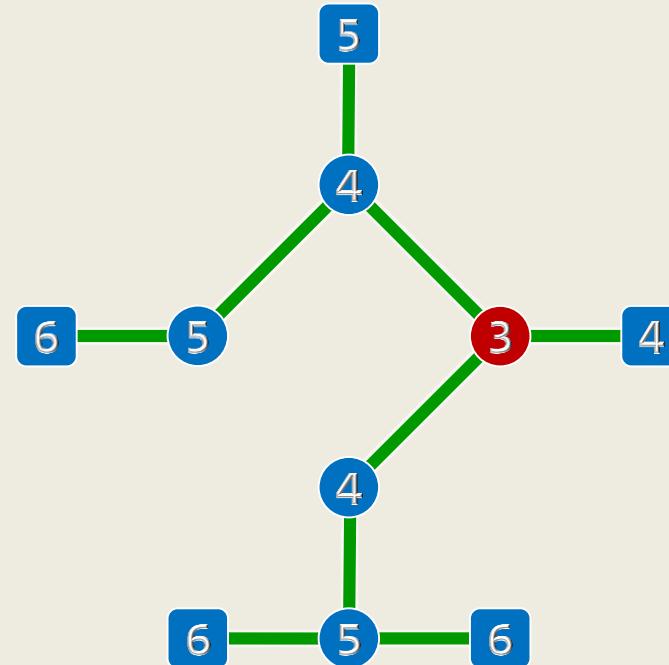
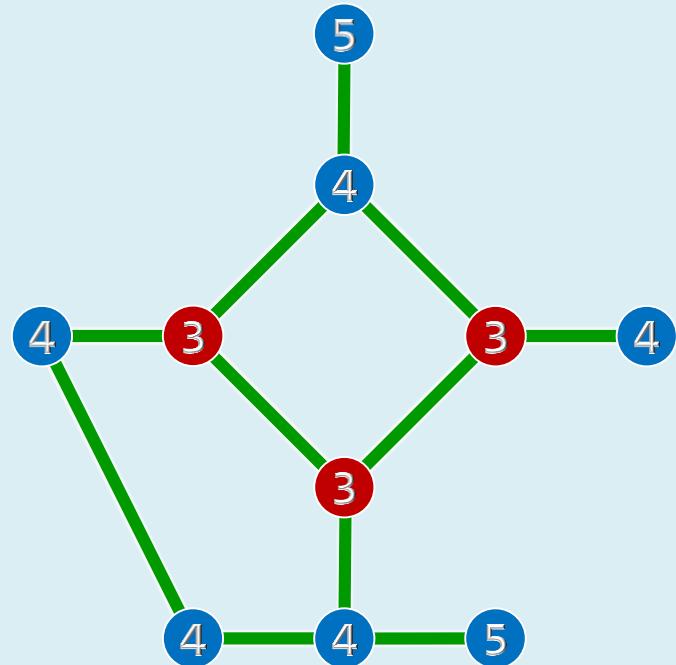
05 - XA

Binary Tree
Applications

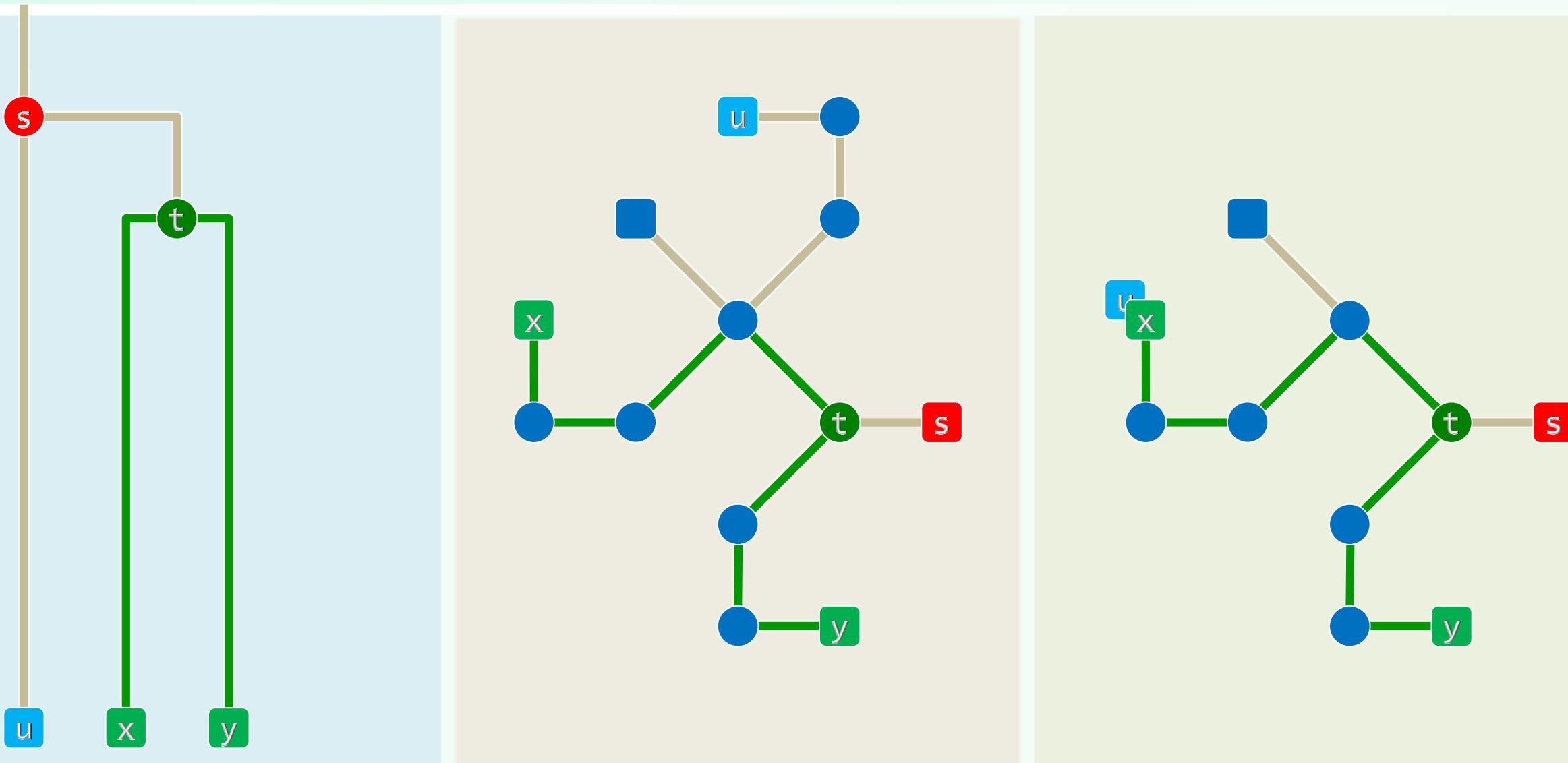
邓俊辉

deng@tsinghua.edu.cn

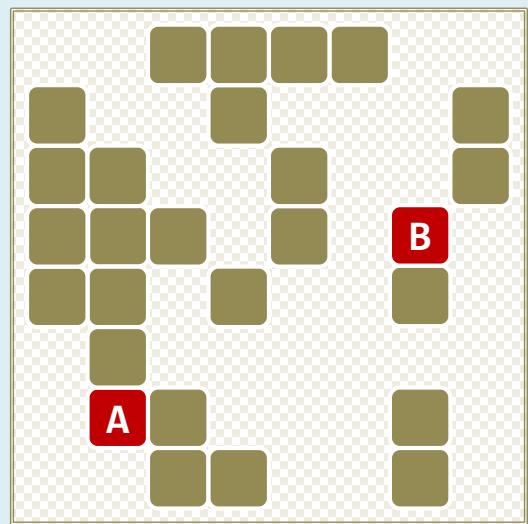
Graph/Tree: Diameter / Eccentricity / Radius / Center



Tree Diameter (by BFS x2)



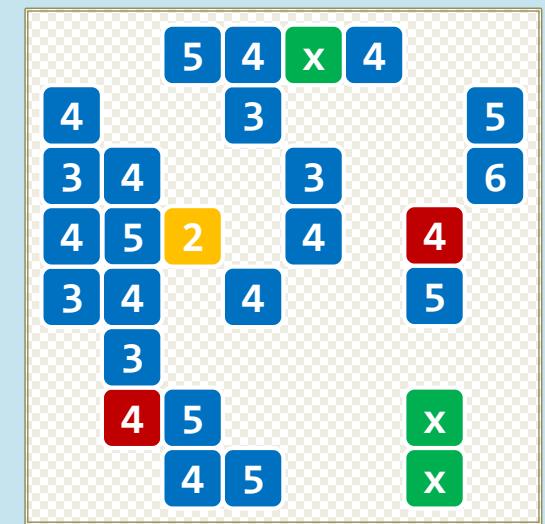
Knights of the Round Table



d_A



d_B



$\max\{d_A, d_B\}$

Traveling Knight

