算法与数据结构

5. 二叉树

赵登阳

中国计量大学信息工程学院

2024 年 10 月 9 日

1. 二叉树及其表示
2. 二叉树的实现
3. 遍历
4. Huffman 编码

1 二叉树及其表示
树
树的表示
二叉树

2 二叉树的实现

3 遍历

4 Huffman 编码

1 二叉树及其表示
树
树的表示
二叉树

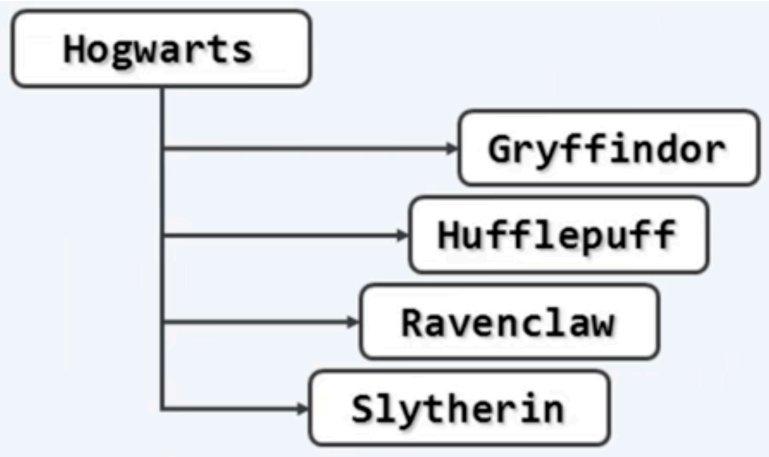
2 二叉树的实现

3 遍历

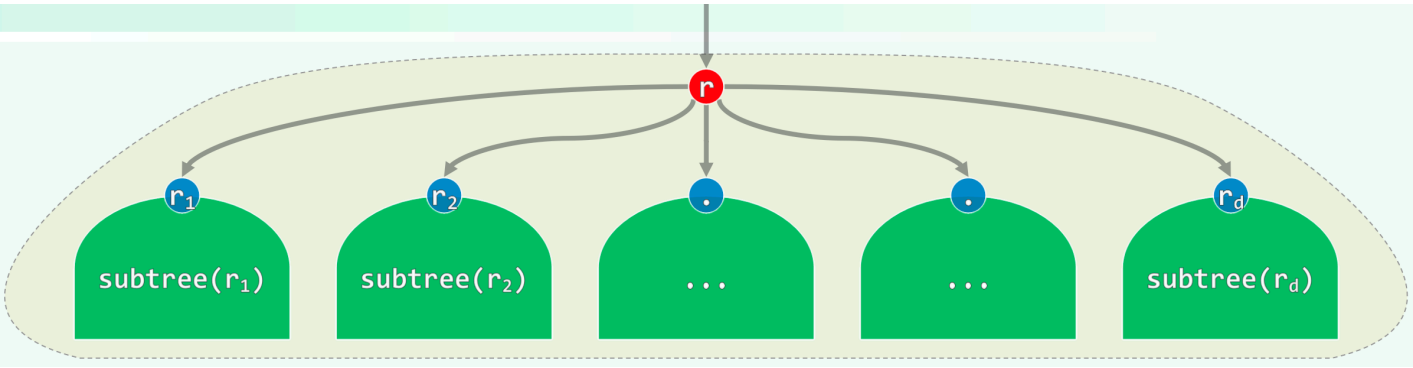
4 Huffman 编码

动机

- 【应用】层次结构的表示
  - 表达式（RPN）
  - 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

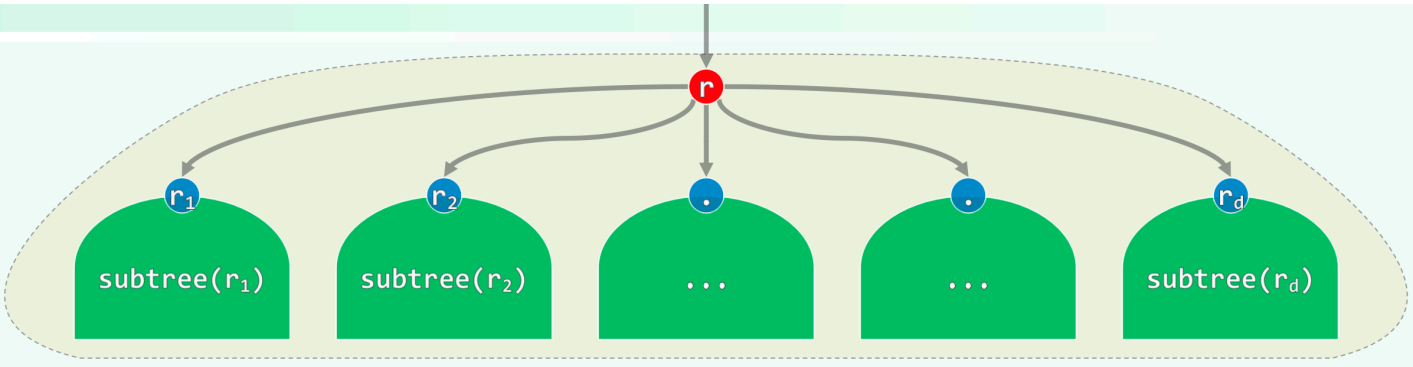


有根树



- 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 $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_d$ 为有根树，则
  $\mathcal{T} = \left((\bigcup_i V_i) \cup r, (\bigcup_i E_i) \cup \{< r, r_i > | 1 \le i \le d\} \right)$ 也是
- 相对于 $\mathcal{T}$，$\mathcal{T}_i$ 称为以 r_i 为根的子树，记作 $\mathcal{T}_i = subtree(r_i)$

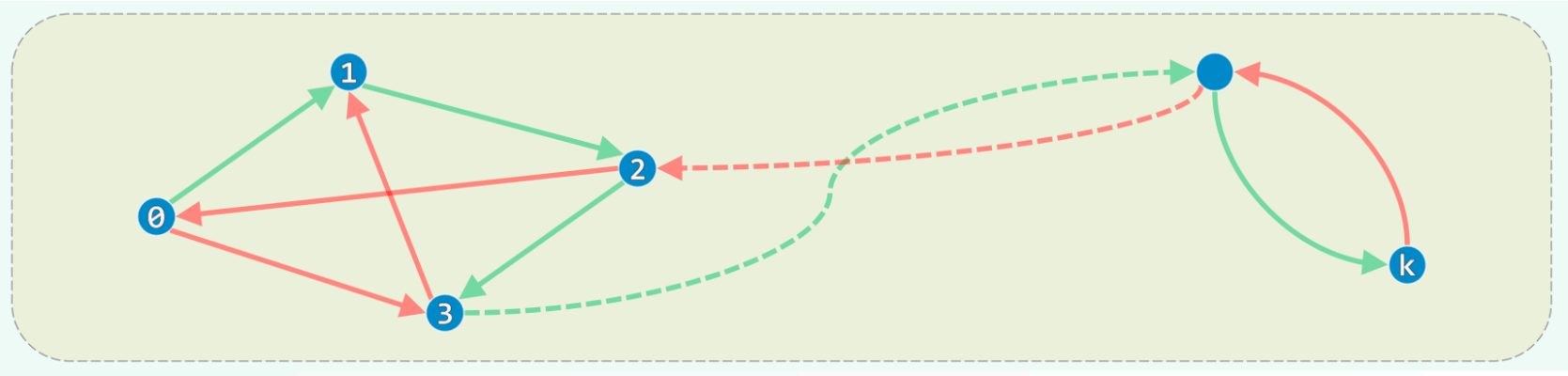
有序树



- r_i 称作 r 的孩子（child），r_i 之间互称兄弟（sibling）
  r 为其父亲（parent），$d = degree(r)$ 为 r 的（出）度
- 可归纳证明，$e = \sum_{v \in V} degree(v) = n - 1 = \Theta(n)$
  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

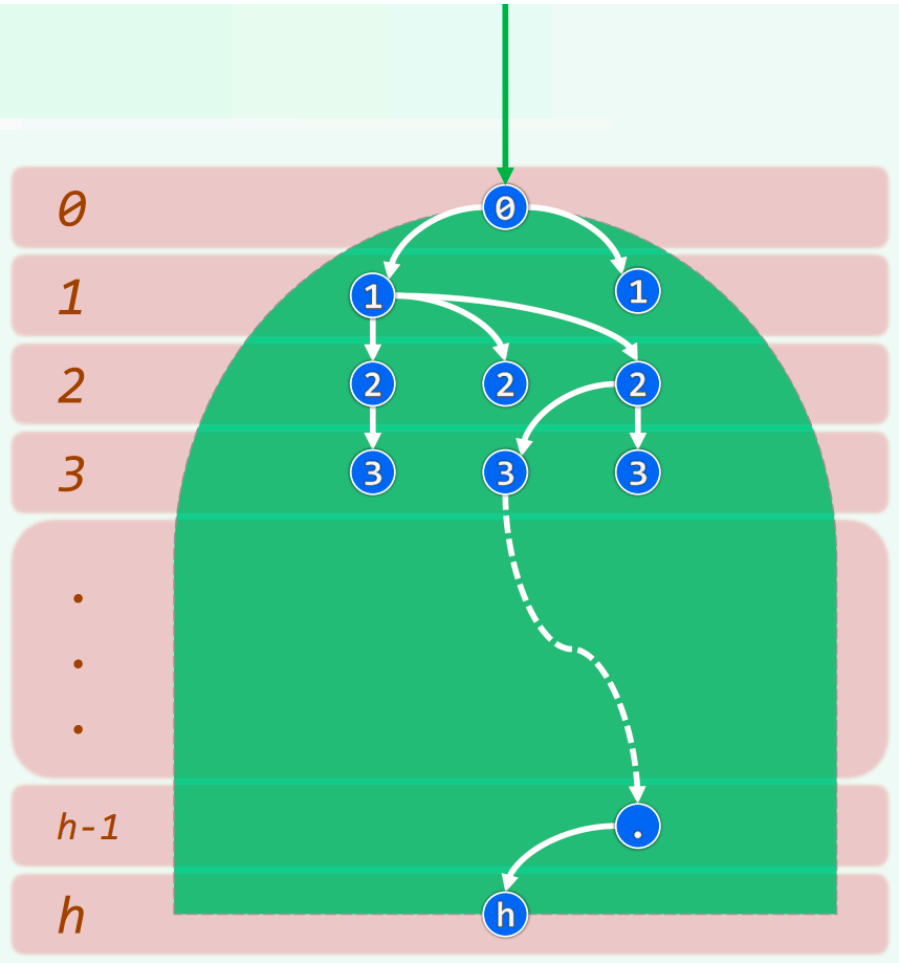
路径 + 环路

- 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）
  $\pi = \{(v_0, v_1), (v_1, v_2), (v_2, v_3) \ldots, (v_{k-1}, v_k)\}$
- 路径长度即所含边数：$|\pi| = k$
- 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）



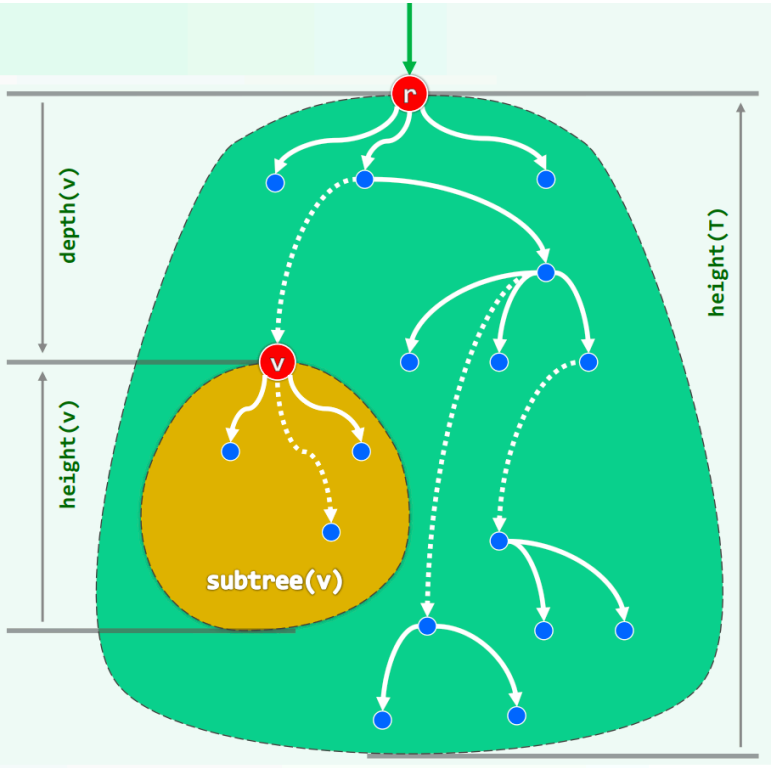
连通 + 无环

- 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
  $path(v, r) = path(v)$
- 于是以 $|path(v)|$ 为指标，可对所有节点做等价类划分...



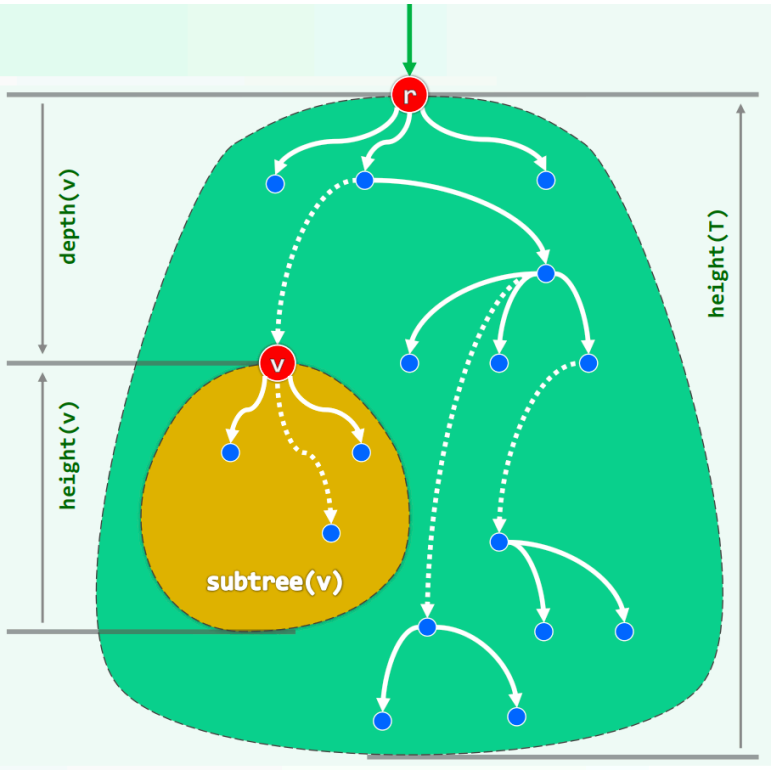
深度 + 层次

- 不致歧义时，路径、节点和子树可相互指代 path(v)~v~subtree(v)
- v 的深度：$depth(v) = |path(v)|$
- 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 $height(v) = height(subtree(v))$
- 特别地，空树的高度取作-1
- $depth(v) + height(v) \leq height(T)$



1 二叉树及其表示
树
树的表示
二叉树

2 二叉树的实现

3 遍历

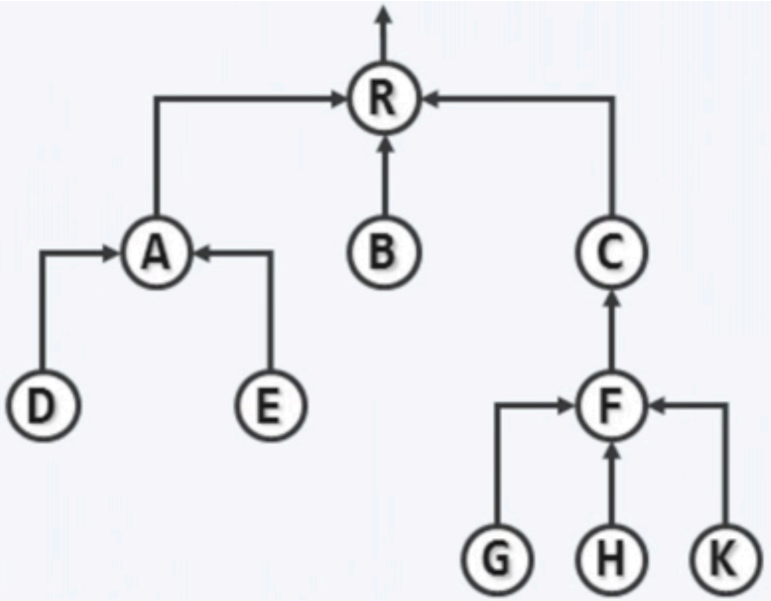
4 Huffman 编码

接口

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

父节点

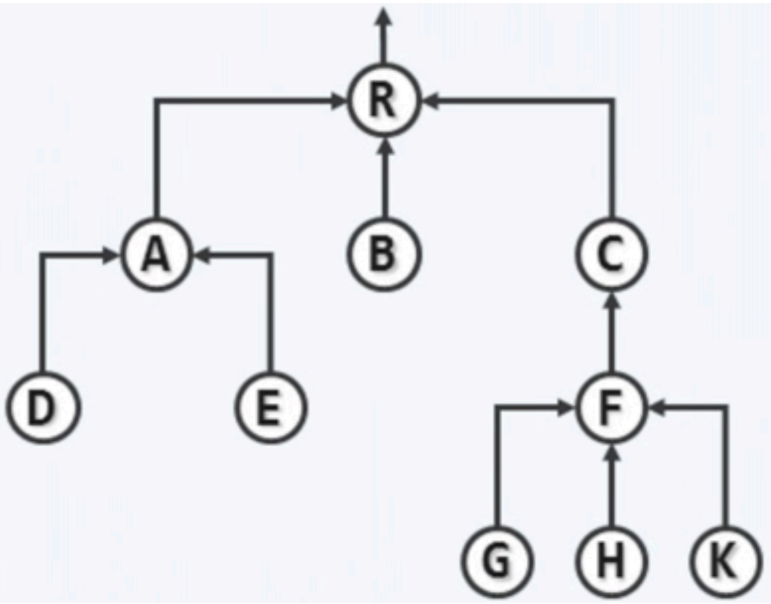
- 除根外，任一节点有且仅有一个父节点
- 节点组织为一个序列，各自记录：
  - data：本身信息
  - parent：父节点的秩或位置
- 树根 R：parent(0) = -1



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

父节点

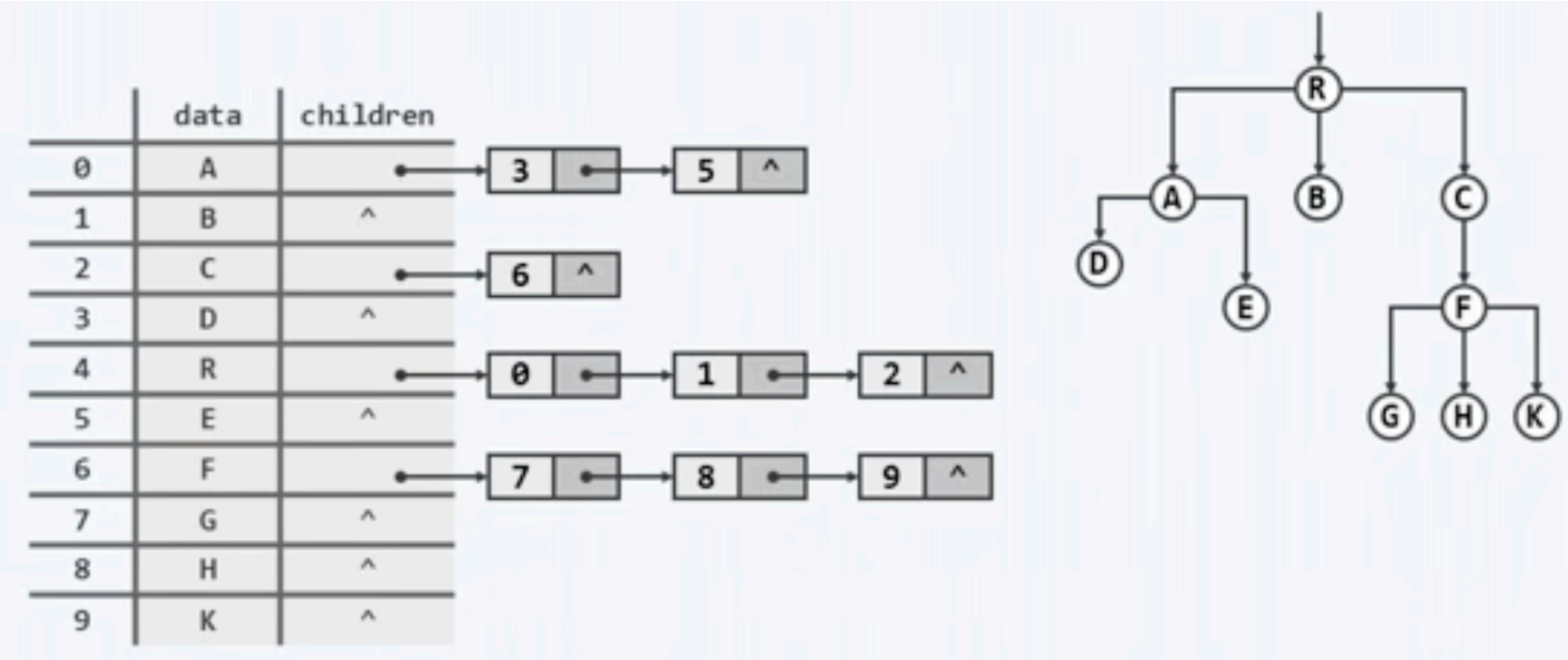
- 空间性能：$\mathcal{O}(n)$
- 时间性能：
  - parent()：$\mathcal{O}(1)$
  - root()：$\mathcal{O}(n)$ 或 $\mathcal{O}(1)$
  - firstChild()：$\mathcal{O}(n)$
  - nextSibling()：$\mathcal{O}(n)$



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

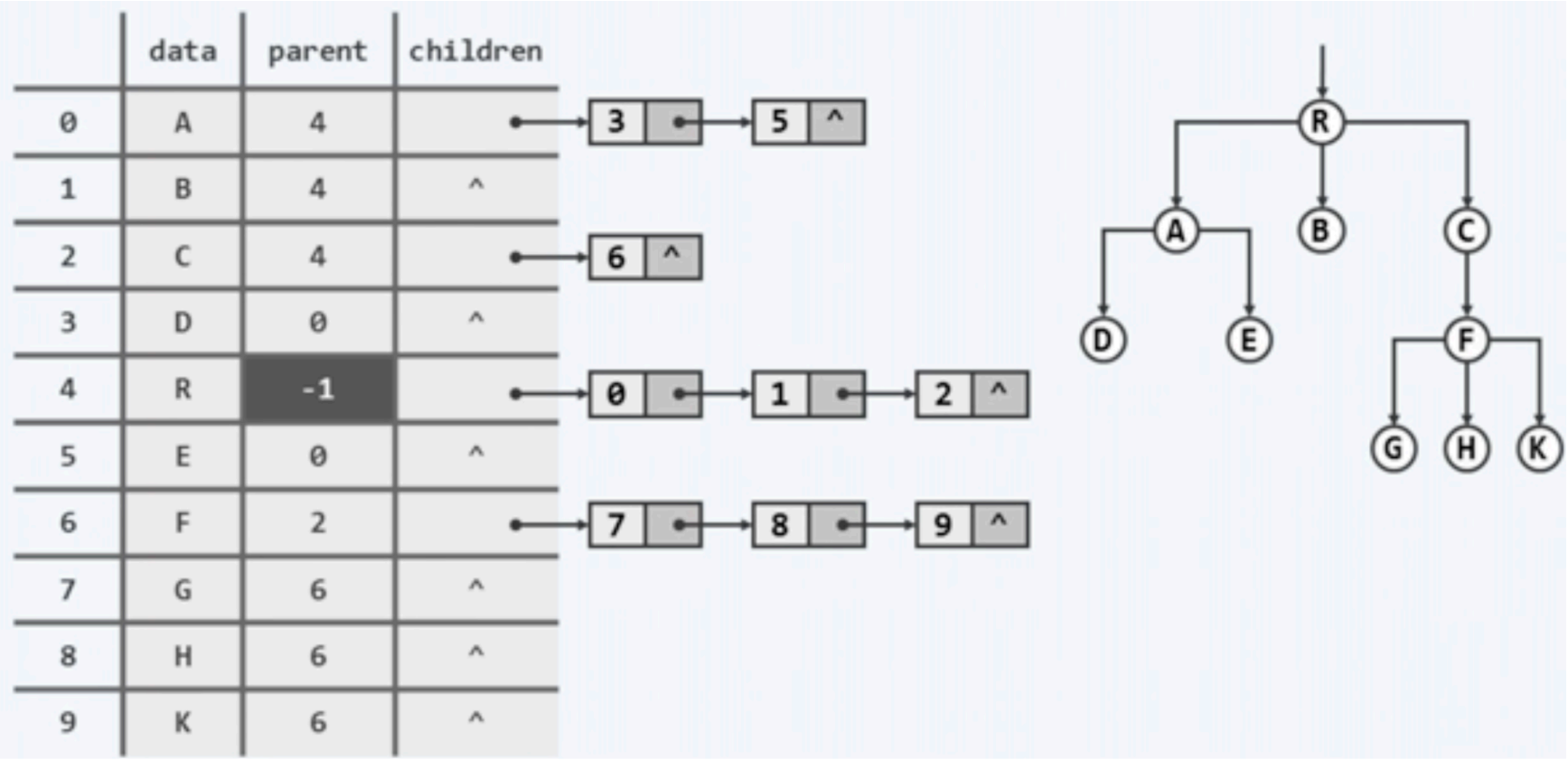
孩子节点

- 对于拥有 r 个孩子的节点，可在 $\mathcal{O}(r)$ 时间内列出所有孩子
- 缺点?



父节点 + 孩子节点

- 高效地兼顾对父节点和孩子的定位
- 插入和删除的效率?



1 二叉树及其表示
树
树的表示
二叉树

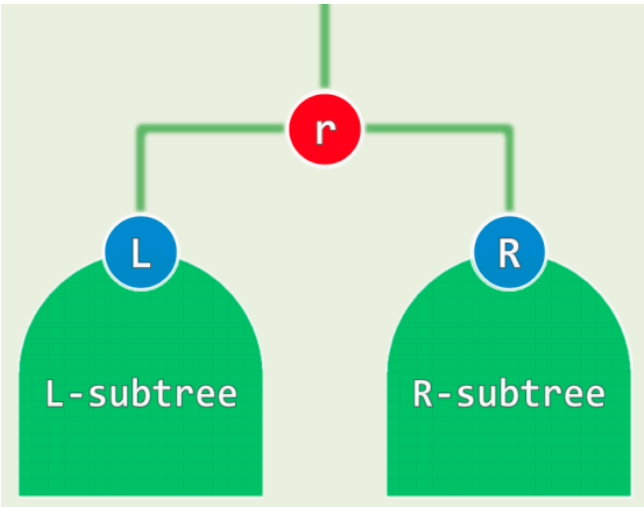
2 二叉树的实现

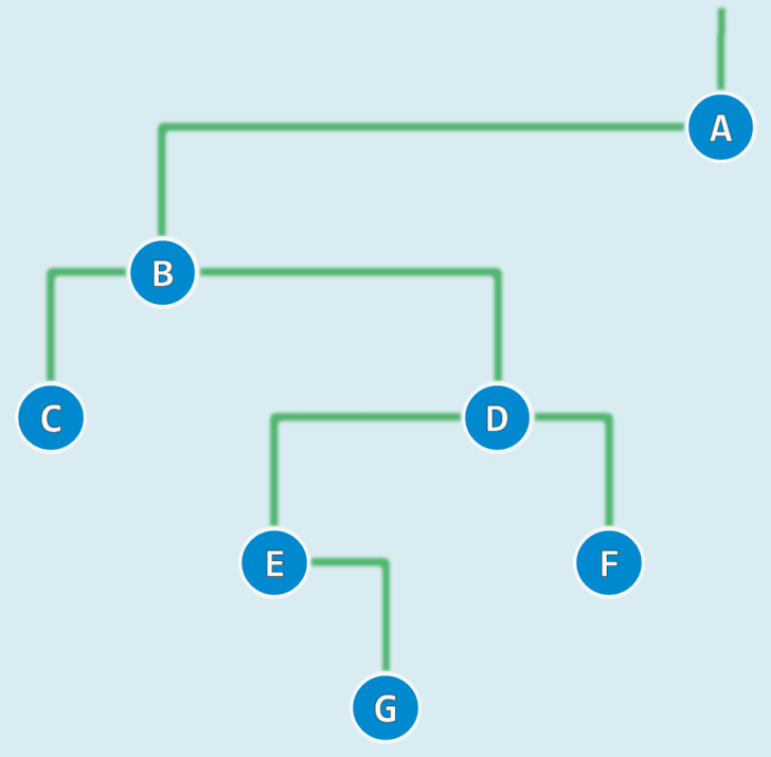
3 遍历

4 Huffman 编码

二叉树

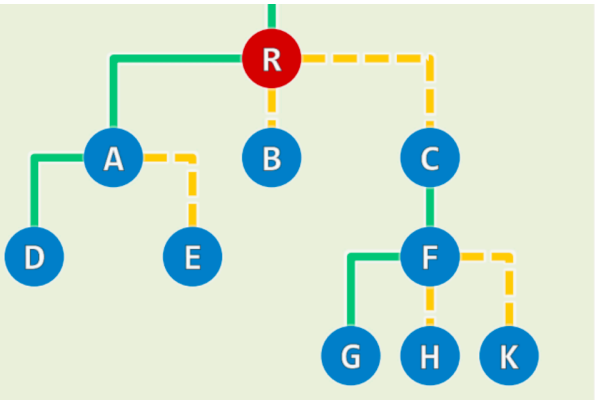
- Binary Tree：节点度数不超过 2
- 孩子（子树）可以左、右区分（隐含有序）
  - lc()：xxlSubtree()
  - rc()：rSubtree()

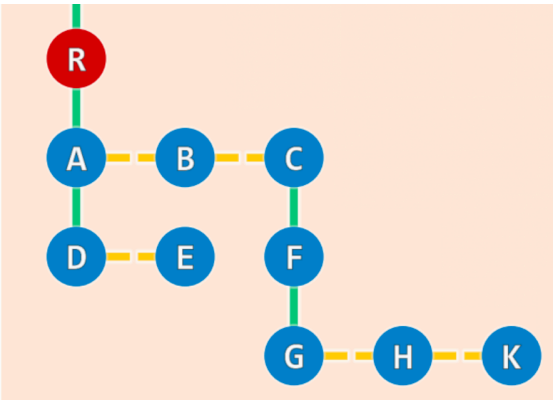


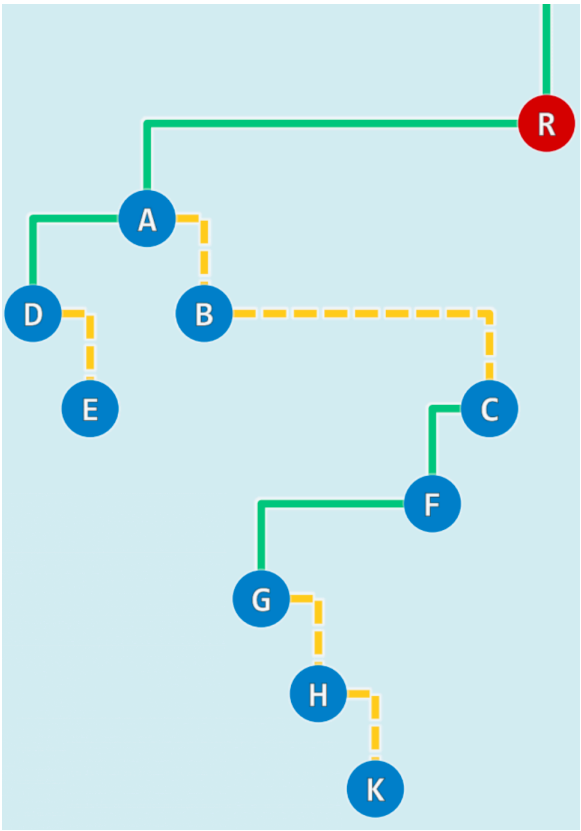


描述二叉树：长子-兄弟表示法

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





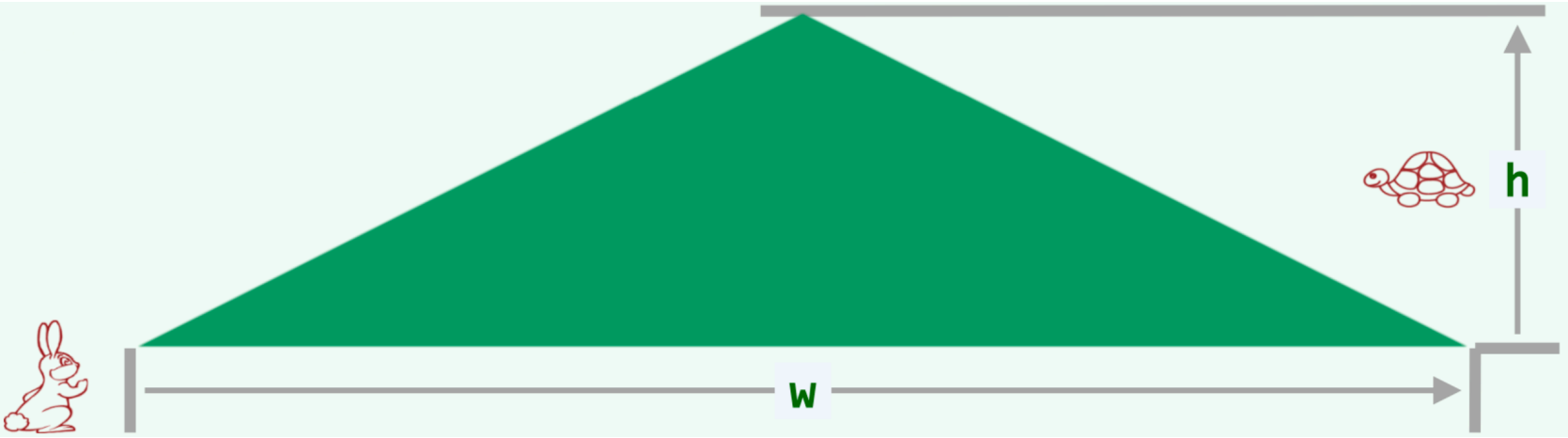


二叉树的性质

- 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
- $e = 2n_2$ 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...

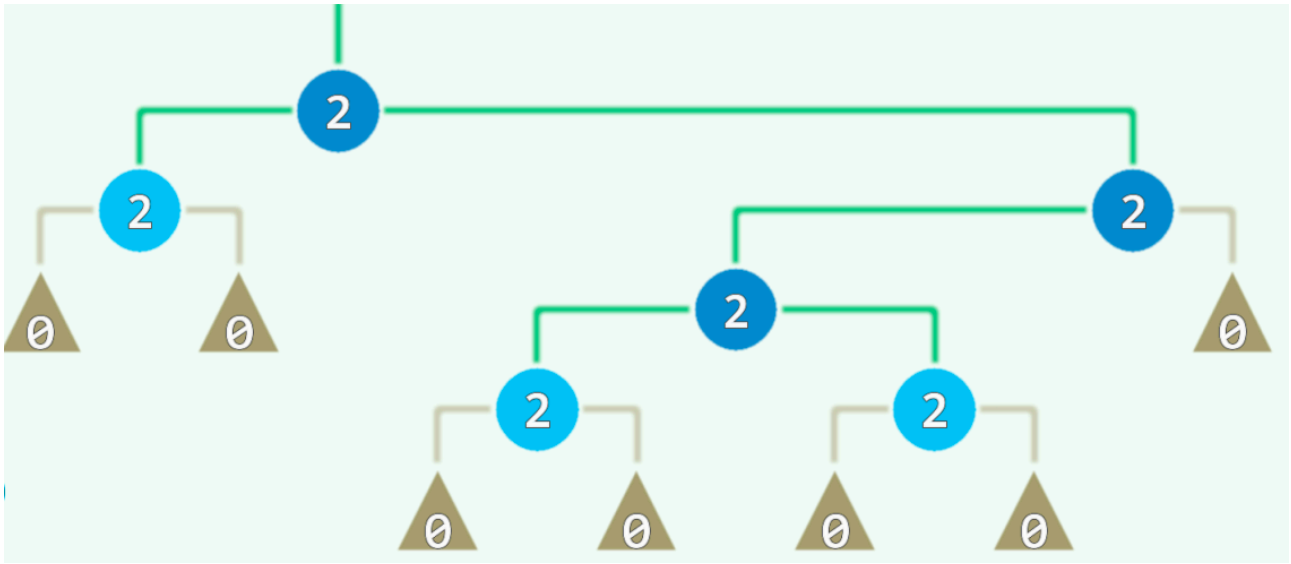
满树

- 深度为 k 的节点，至多 2^k 个
- n 个节点、高 h 的二叉树满足 $h+1 \le n \le 2^{h+1}-1$
- 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）
- 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



1 二叉树及其表示

2 二叉树的实现
二叉树节点
二叉树

3 遍历

4 Huffman 编码

1 二叉树及其表示

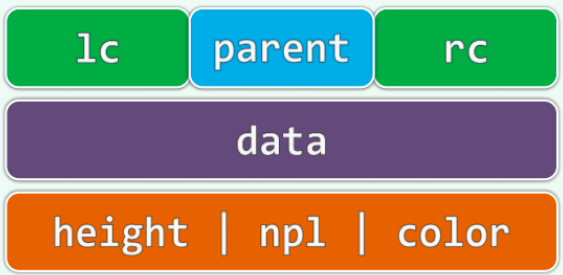
2 二叉树的实现
二叉树节点
二叉树

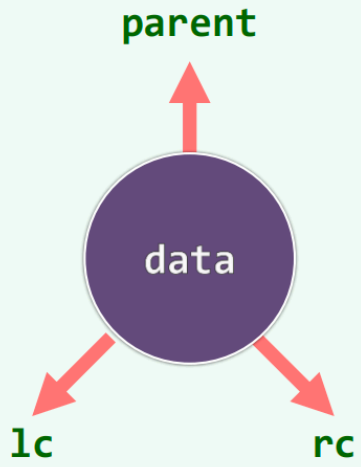
3 遍历

4 Huffman 编码

BinNode 模板类

```
//节点位置
template <typename T> using
    BinNodePosi = BinNode<T>*;
//节点高度（空树高度为 -1）
#define stature(p)
    ((p) ? (p)->height : -1)
//节点颜色
typedef enum
    {RB_RED, RB_BLACK} RBColor;
//二叉树节点模版类
template <typename T>
struct BinNode {
    //父亲、孩子
    BinNodePosi<T> parent, lc, rc;
    T data; Rank height, npl;
    //高度、npl、子树规模
    Rank size();
```





BinNode 模板类

```
    //作为左孩子插入新节点
    BinNodePosi<T> insertAsLC( T const & );
    //作为右孩子插入新节点
    BinNodePosi<T> insertAsRC( T const & );
    //（中序遍历意义下）当前节点的直接后继
    BinNodePosi<T> succ();
    // 二叉树的遍历
    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
};
```

快捷方式：状态与性质的判断

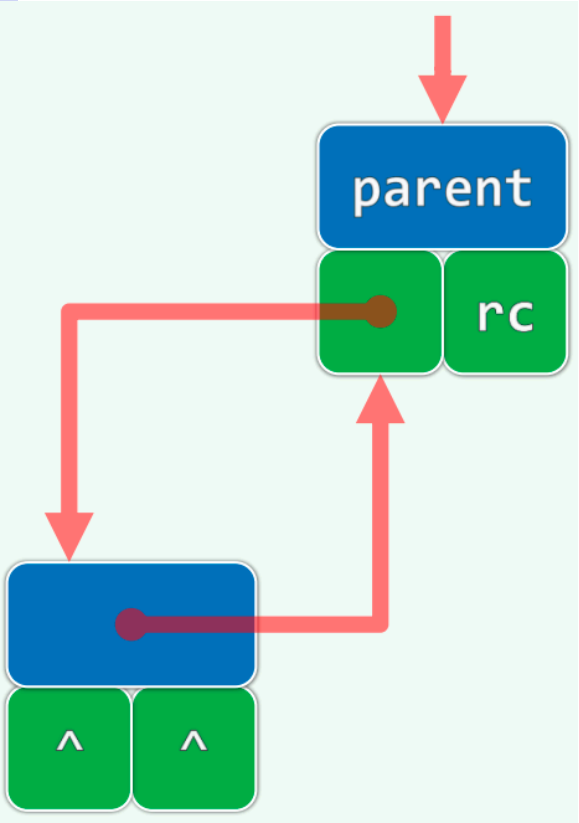
```
#define IsRoot(x) (!((x).parent))
#define IsLChild(x) \
    (!IsRoot(x) && (&(x) == (x).parent->lc))
#define IsRChild(x) \
    (!IsRoot(x) && (&(x) == (x).parent->rc))
#define HasParent(x) (!IsRoot(x))
#define HasLChild(x) ((x).lc)
#define HasRChild(x) ((x).rc)
#define HasChild(x) \
    (HasLChild(x) || HasRChild(x))
#define HasBothChild(x) \
    (HasLChild(x) && HasRChild(x))
#define IsLeaf(x) (!HasChild(x))
```

快捷方式：与 BinNode 具有特定关系的节点及指针

```
// 兄弟
#define sibling(p) (IsLChild(*(p)) ? \
    (p)->parent-rc : (p)->parent-lc)
// 叔叔
#define uncle(x) (IsLChild(*((x)->parent)) ? \
    (x)->parent->parent-rc : \
    (x)->parent->parent-lc)
// 来自父亲的引用
#define FromParentTo(x) (IsRoot(x) ? _root : \
    (IsLChild(x) ? (x).parent->lc : \
    (x).parent->rc))
```

BinNode：引入新节点

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

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



1 二叉树及其表示

2 二叉树的实现
二叉树节点
二叉树

3 遍历

4 Huffman 编码

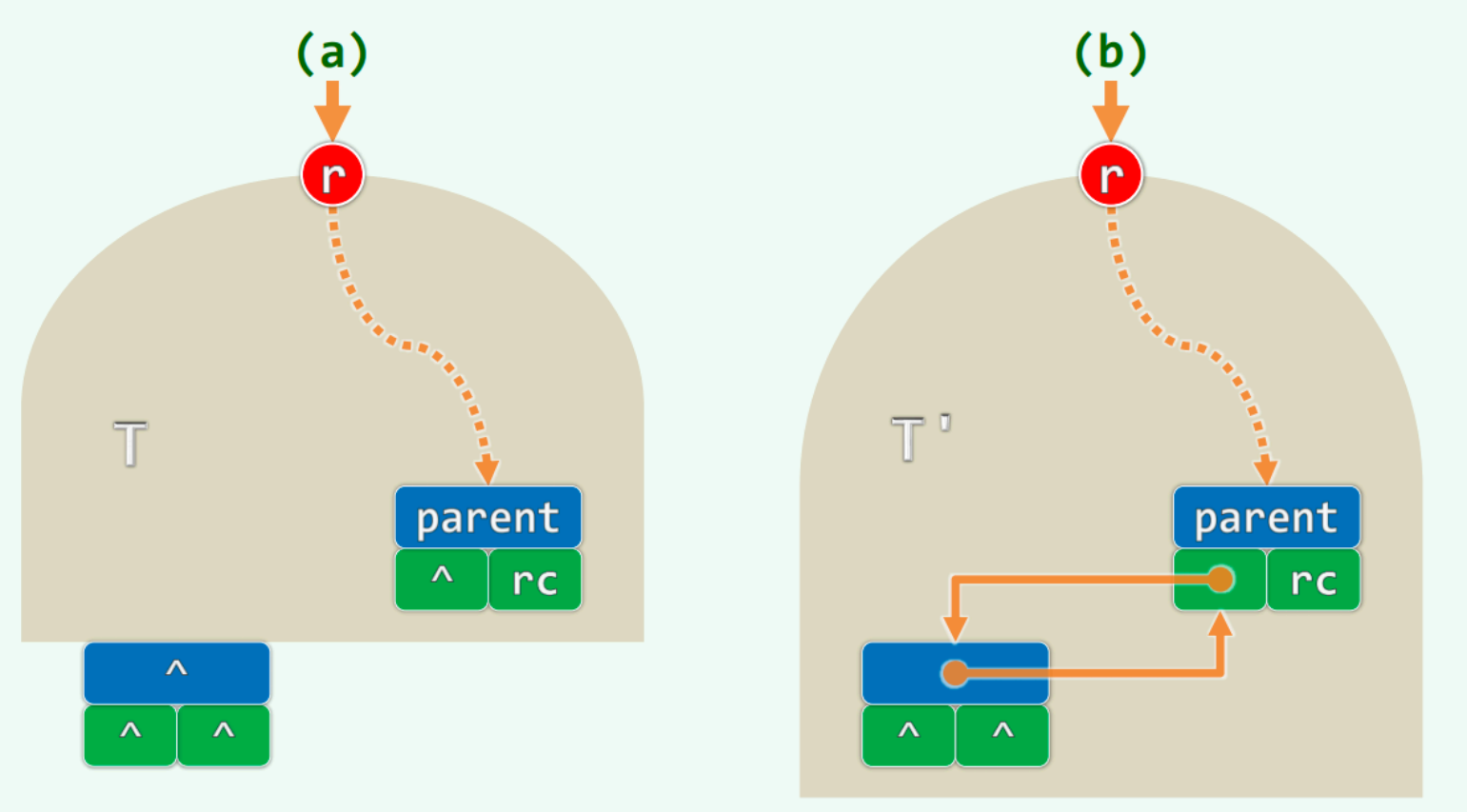
BinTree 模板类

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

BinTree：更新高度

```
//更新节点x高度，此处采用常规二叉树规则，O(1)
template <typename T> Rank BinTree<T>::
updateHeight(BinNodePos(T) x) {
    return x->height = 1 +
        max( stature(x->lc), stature(x->rc) );
}

//更新节点及其历代祖先的高度，O( n = depth(x) )
template <typename T> void BinTree<T>::
updateHeightAbove(BinNodePosi(T) x) {
    while (x) {
        updateHeight(x);
        x = x->parent;
    }
} //可优化
```

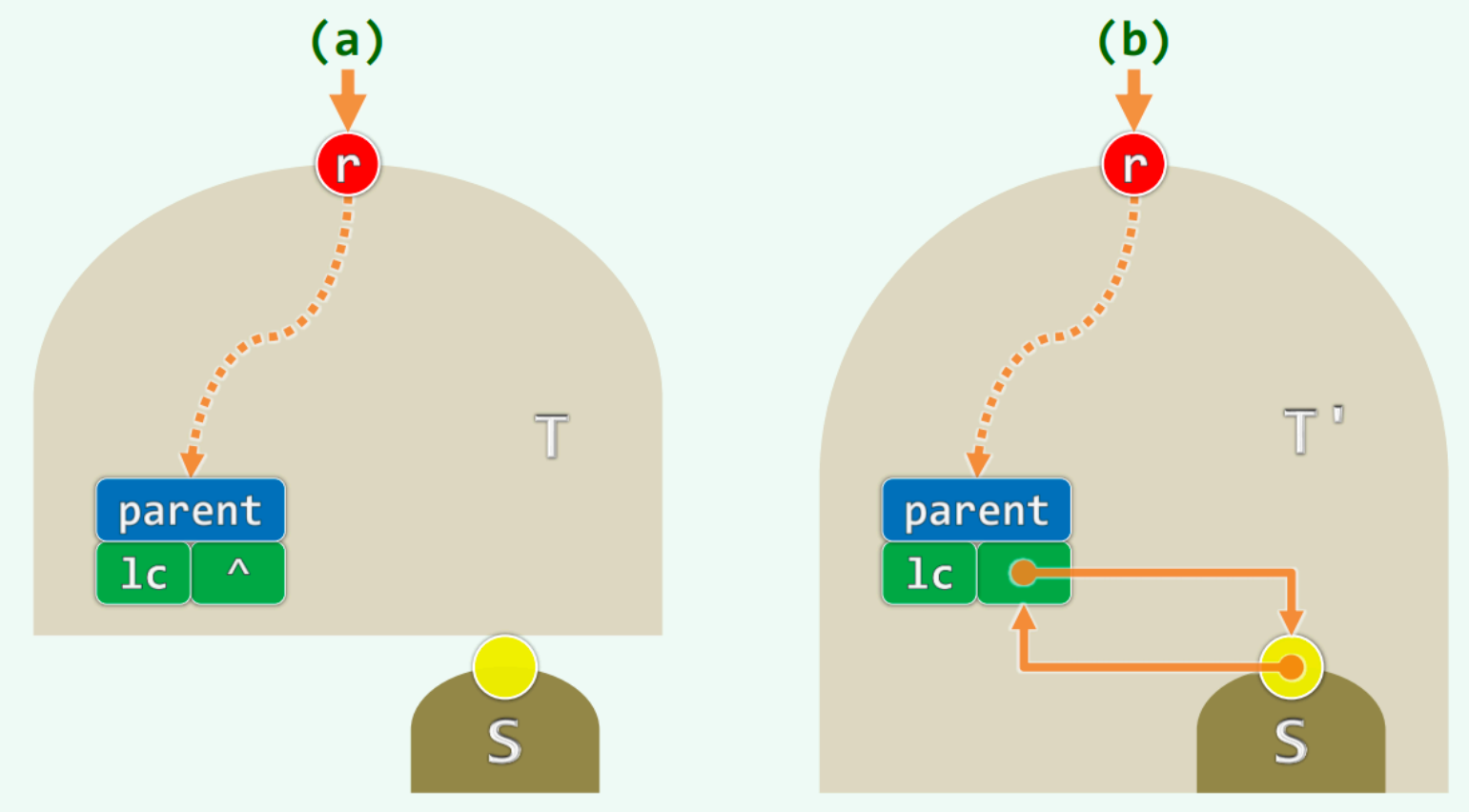
BinTree：引入新节点



BinTree：引入新节点

```
template <typename T> BinNodePosi(T) BinTree<T>::
insertAsRoot(T const & e)
{
    _size = 1;
    return _root = new BinNode<T> (e);
}

template <typename T> BinNodePosi(T) BinTree<T>::
insertAsLC( BinNodePosi(T) x, T const & e)
{
    _size++;
    x->insertAsLC( e );
    updateHeightAbove( x );
    return x->lc;
}
```

BinTree：接入子树



BinTree：接入子树

```
template <typename T> BinNodePosi(T) BinTree<T>::
attachAsRC( 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);
    // 释放原树
    S->_root = NULL;
    S->_size = 0;
    release(S); S = NULL;
    return x;
}
```

BinTree：子树删除

```
template <typename T>
int BinTree<T>::remove( BinNodePosi(T) x ) {
    FromParentTo(*x) = NULL;
    updateHeightAbove(x->parent);
    int n = removeAt(x); _size -= n; return n;
}

template <typename T> static int BinTree<T>::
removeAt( BinNodePosi(T) x ) {
    if (!x) return 0;
    int n = 1 + removeAt(x->lc) + removeAt(x-rc);
    release(x->data); release(x); return n;
}
```

BinTree：子树分离

```
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
}
```

1 二叉树及其表示

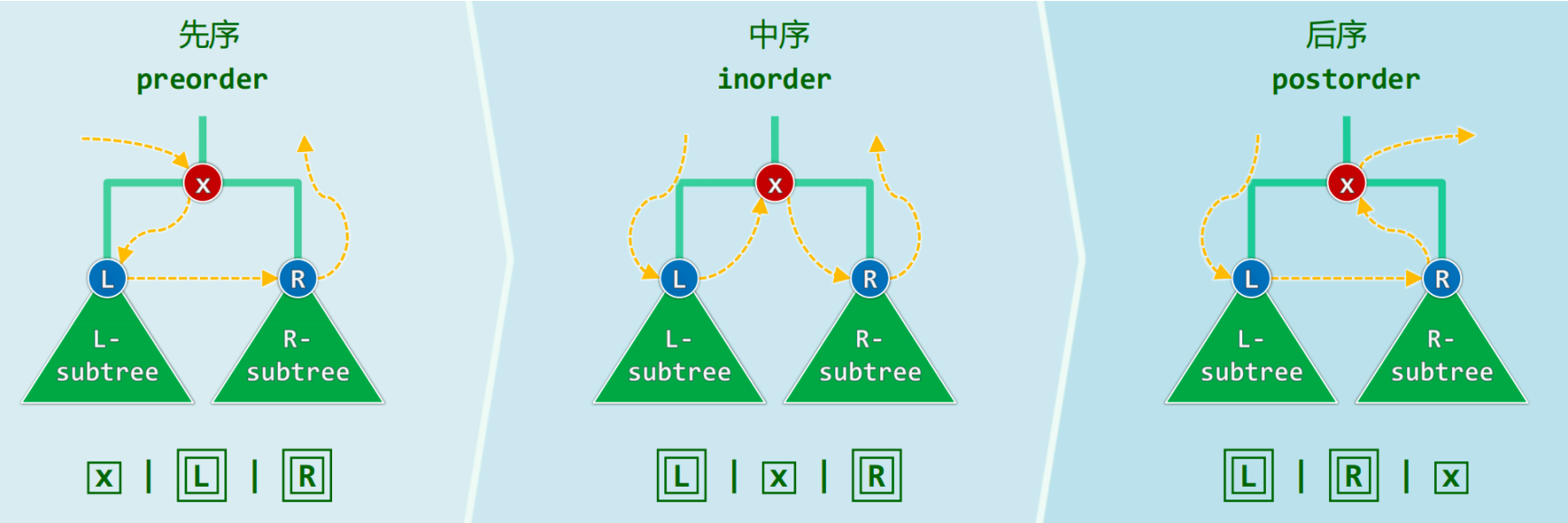
2 二叉树的实现

3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

4 Huffman 编码

遍历

- 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



1 二叉树及其表示

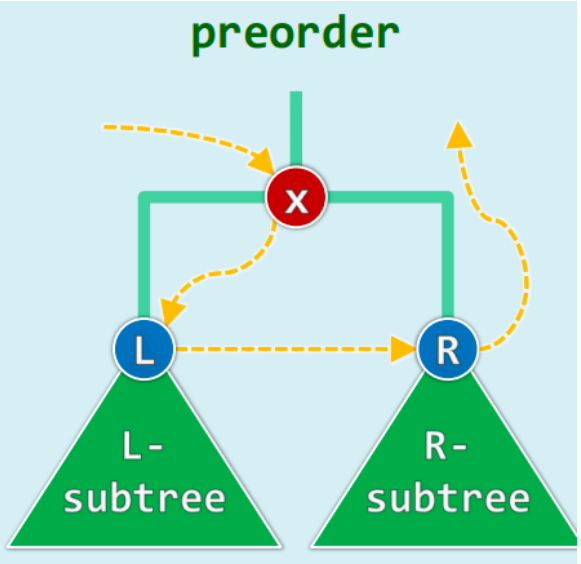
2 二叉树的实现

3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

4 Huffman 编码

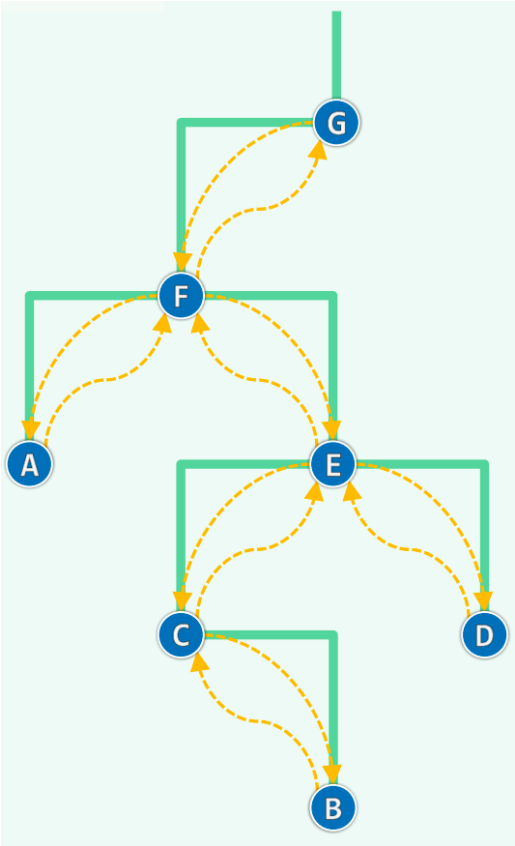
先序实例：统计规模

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



递归实现

```
template <typename T, typename VST>
void travPre_R( BinNodePosi(T) x,
     VST & visit )
{
     if ( ! x ) return;
     visit( x->data );
     travPre_R( x->lc, visit );
     travPre_R( x->rc, visit );
} //O(n)
```

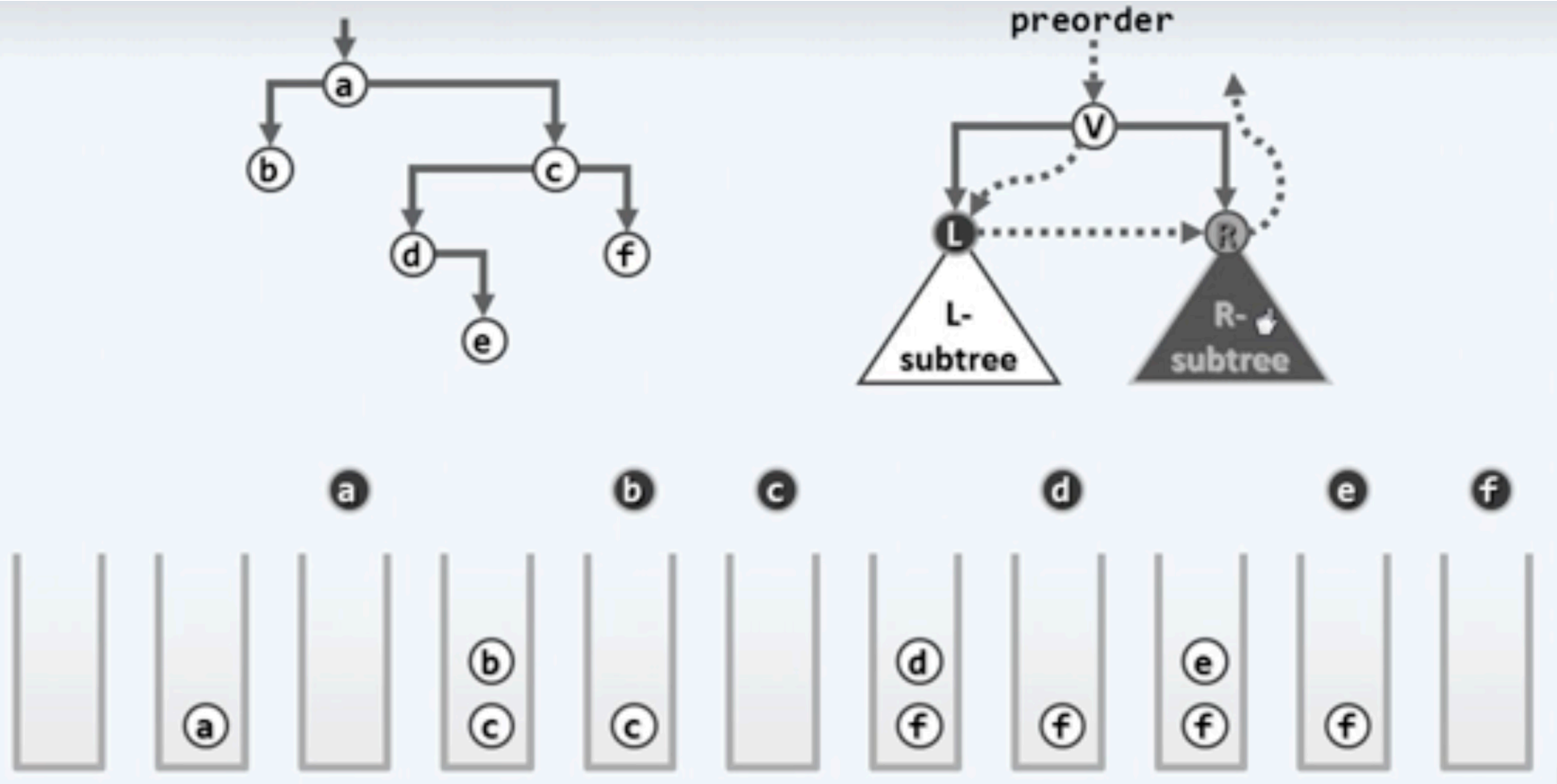


- 制约：使用默认的 Call Stack，允许的递归深度有限
- 挑战：不依赖递归机制，能否实现先序遍历？

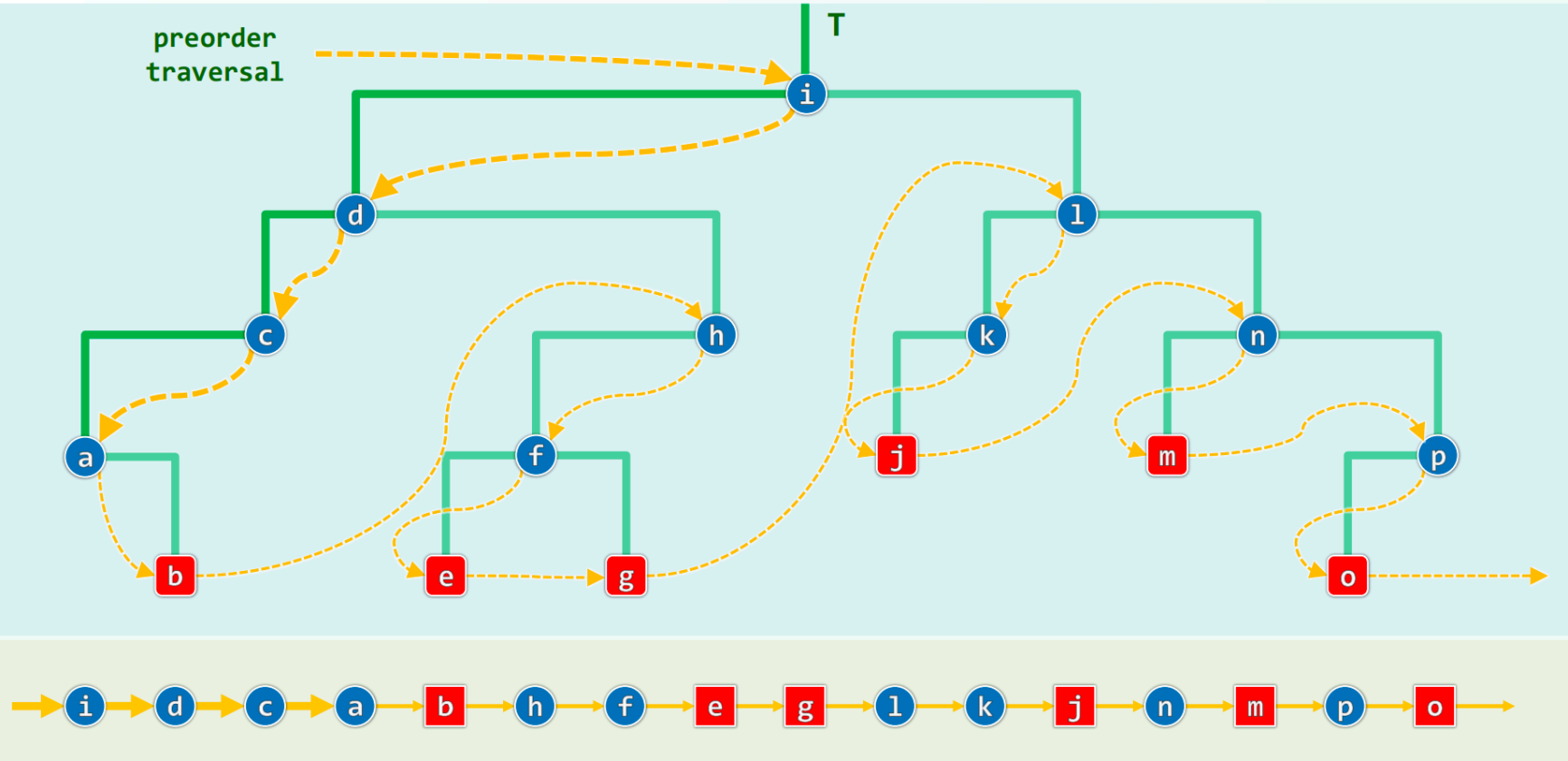
尾递归改写成迭代

```
template <typename T, typename VST>
void travPre_I1( BinNodePosi(T) x, VST & visit )
{
    Stack < BinNodePosi(T) > S; //辅助栈
    if(x) S.push(x); //根结点入栈
    while ( !S.empty() ) //在栈变空前反复迭代
    {
        x = S.pop();
        visit( x->data ); //弹出并访问当前节点
        if ( HasRChild(*x) )
            S.push( x->rc ); //右孩子先入后出
        if ( HasLChild(*x) )
            S.push( x->lc ); //左孩子后入先出
    }
}
```

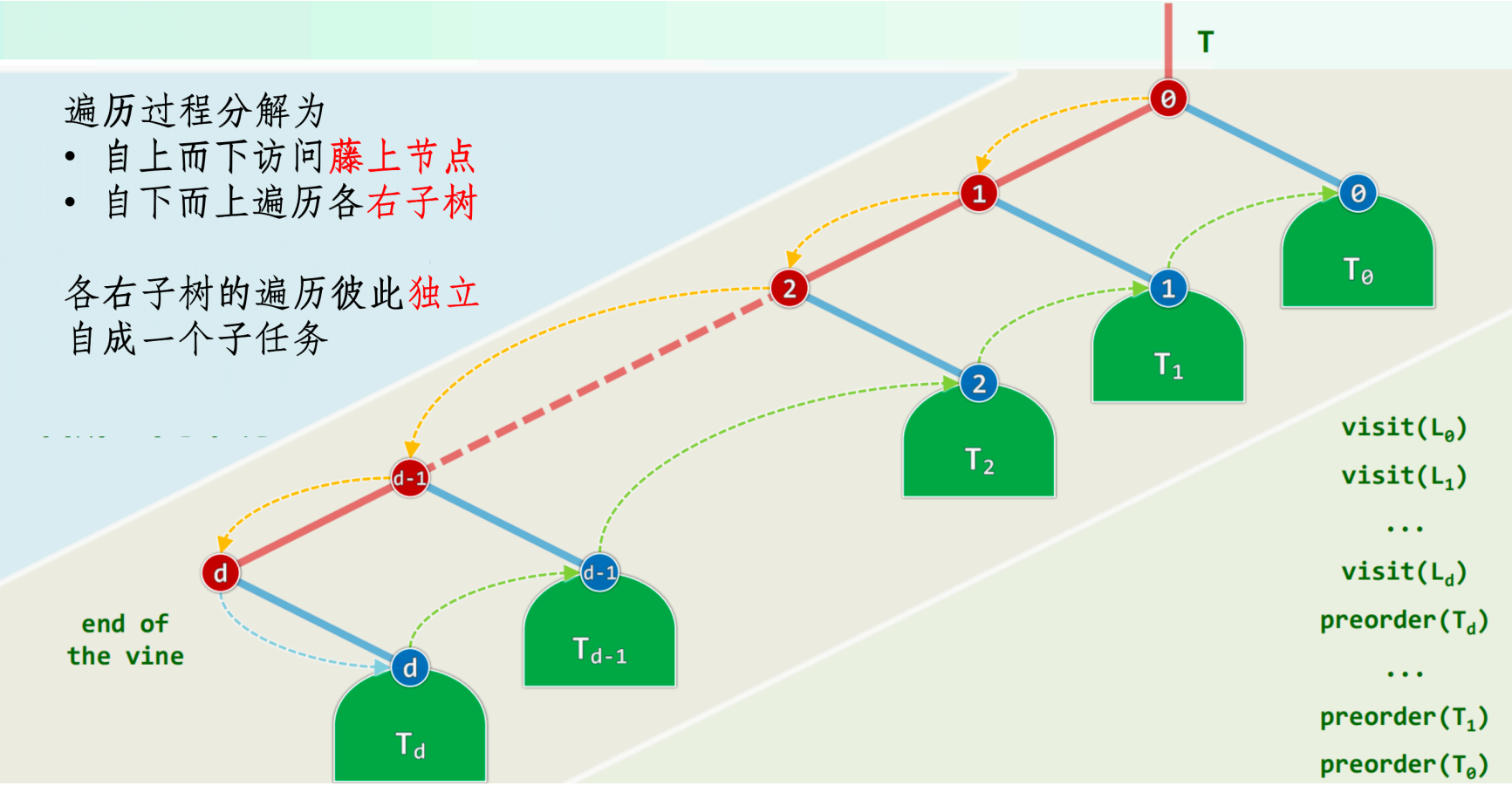
迭代 1 实例



观察


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

藤缠树


遍历过程分解为
• 自上而下访问藤上节点
• 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
T
0
1
2
d-1
d
0
1
2
d-1
d
T0
T1
T2
Td-1
Td
end of
the vine
visit(L0)
visit(L1)
...
visit(Ld)
preorder(Td)
...
preorder(T1)
preorder(T0)

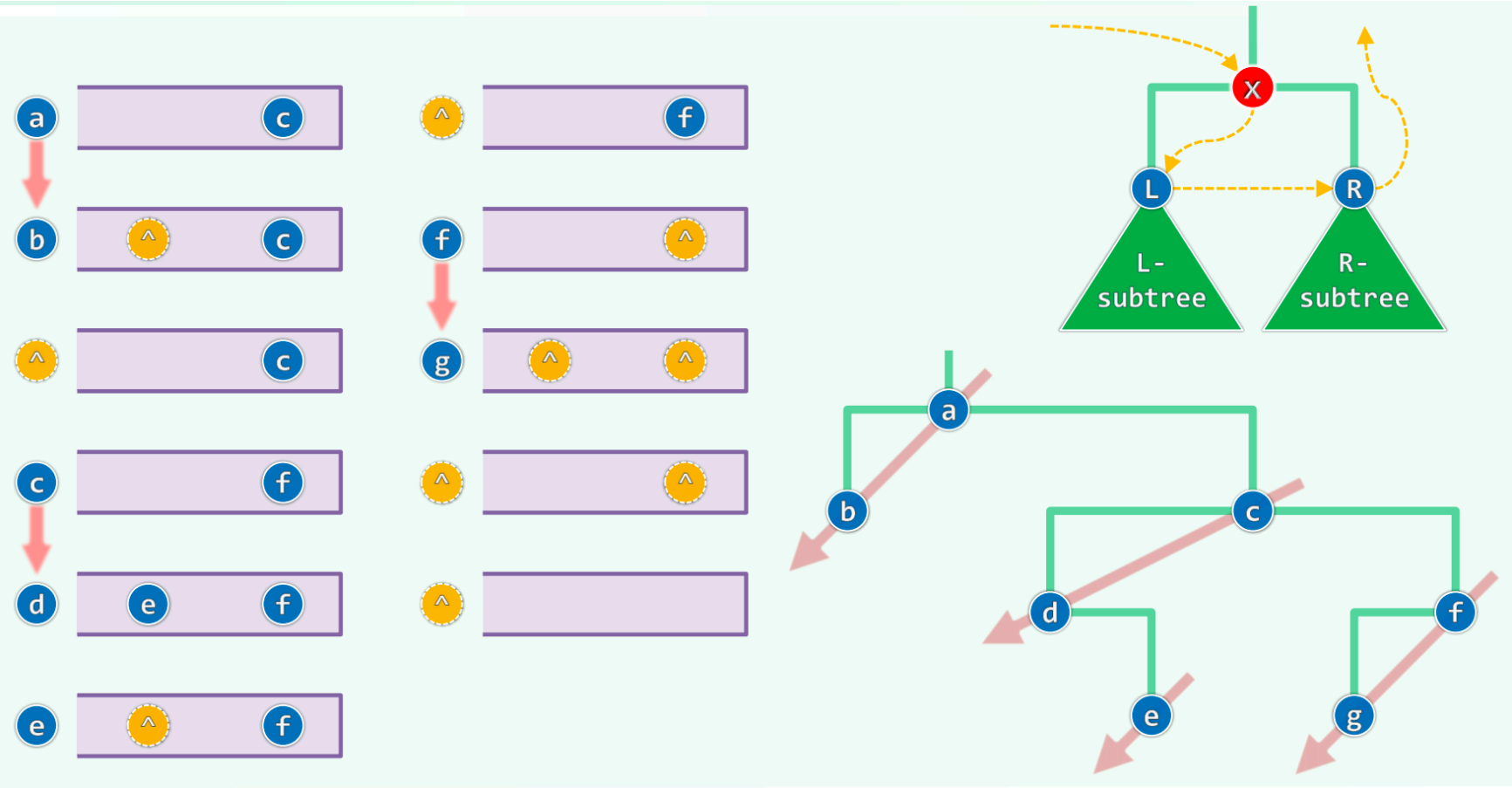
序曲

```
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
        //右孩子（右子树）入栈（将来逆序出栈）
        S.push( x->rc );
        x = x->lc; //沿藤下行
    }
}
```

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
    {
        //访问子树x的藤蔓，
        //各右子树（根）入栈缓冲
        visitAlongVine( x, visit, S );
        if ( S.empty() ) break; //栈空即退出
        x = S.pop(); //弹出下一右子树（根）
    }
}
```

迭代实例



1 二叉树及其表示

2 二叉树的实现

3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

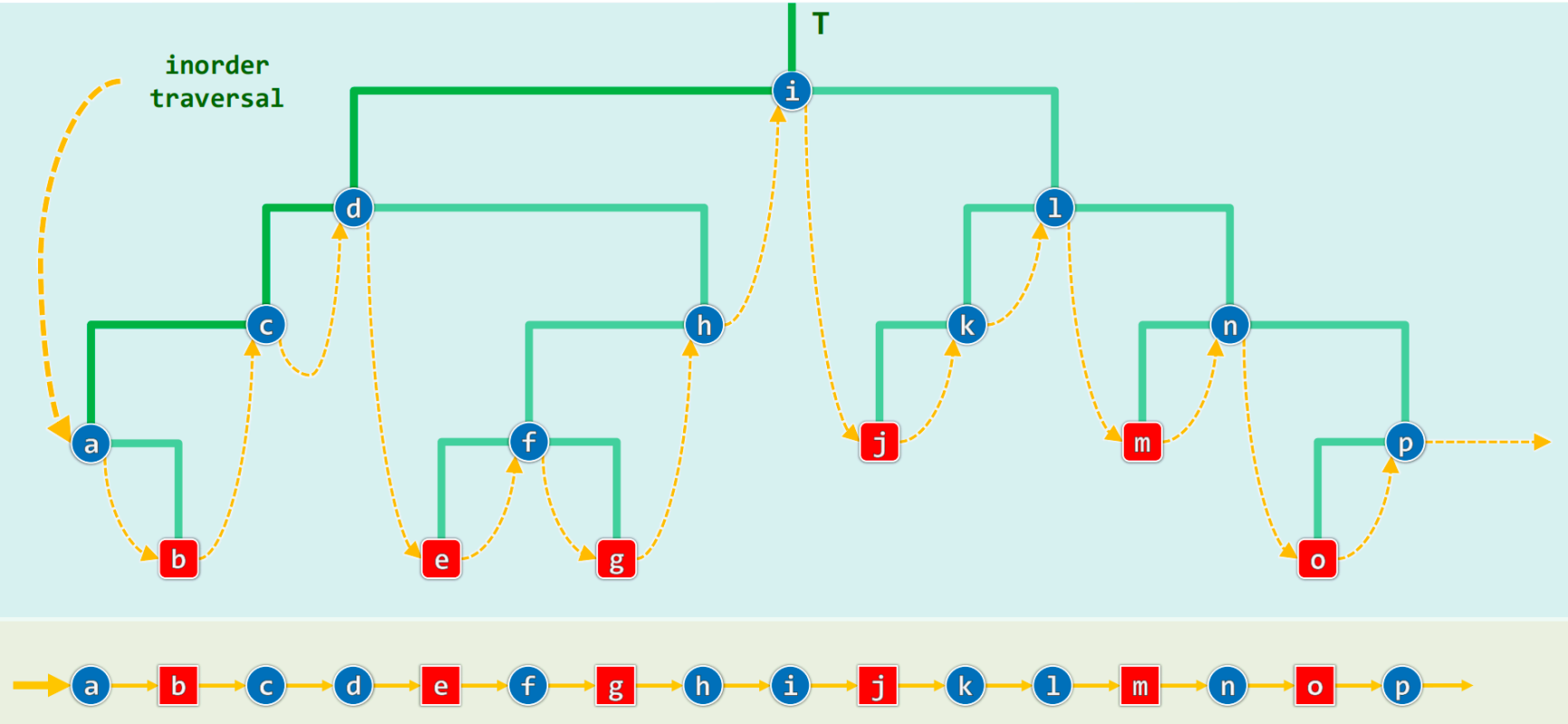
4 Huffman 编码

递归实现

```
template <typename T, typename VST>
void travIn_R( BinNodePosi(T) x, VST & visit )
{
    if ( ! x ) return;
    travIn_R( x->lc, visit );
    visit( x->data );
    travIn_R( x->rc, visit );
} //O(n)
```

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

观察

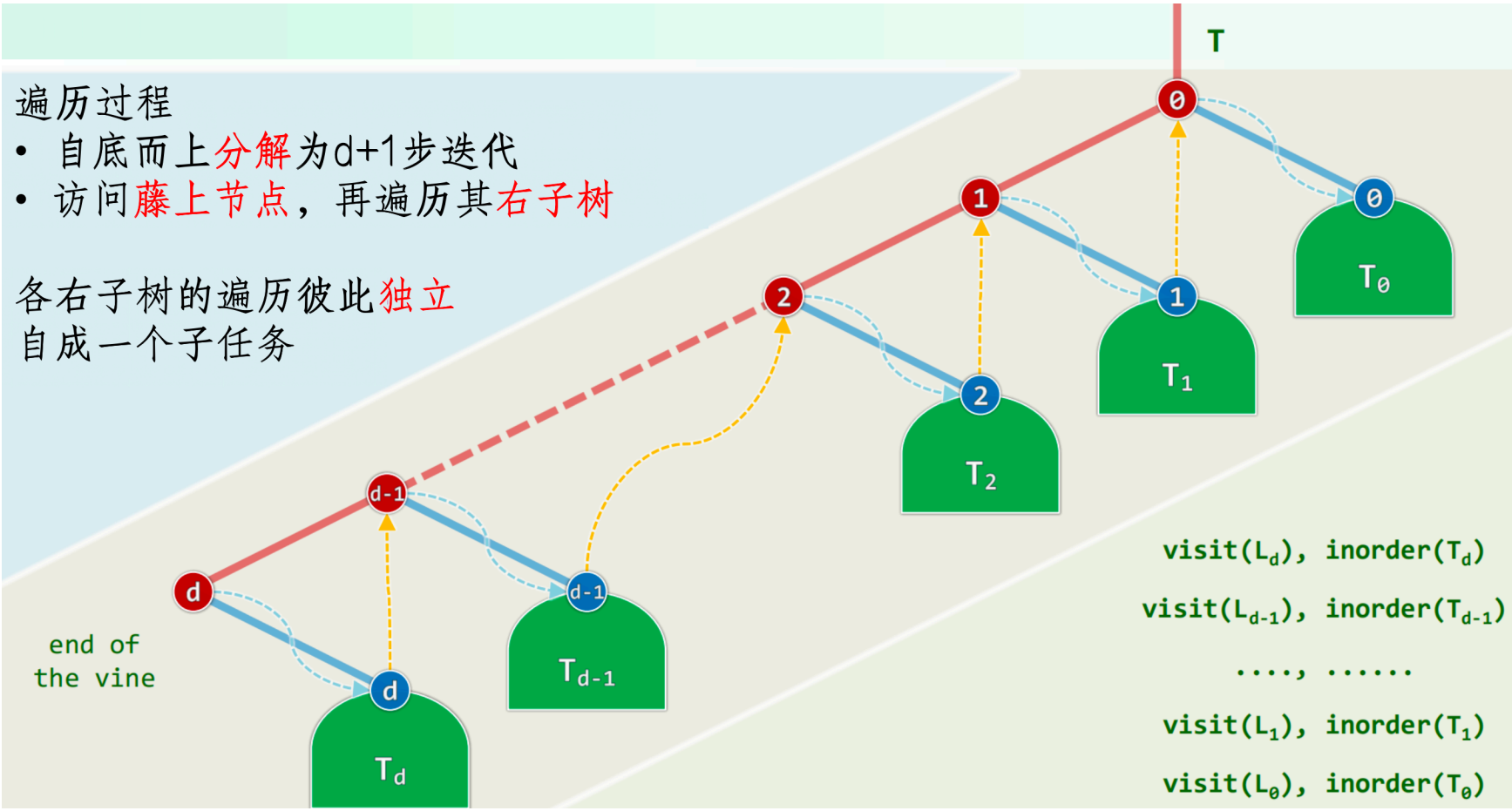


藤缠树

遍历过程

- 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



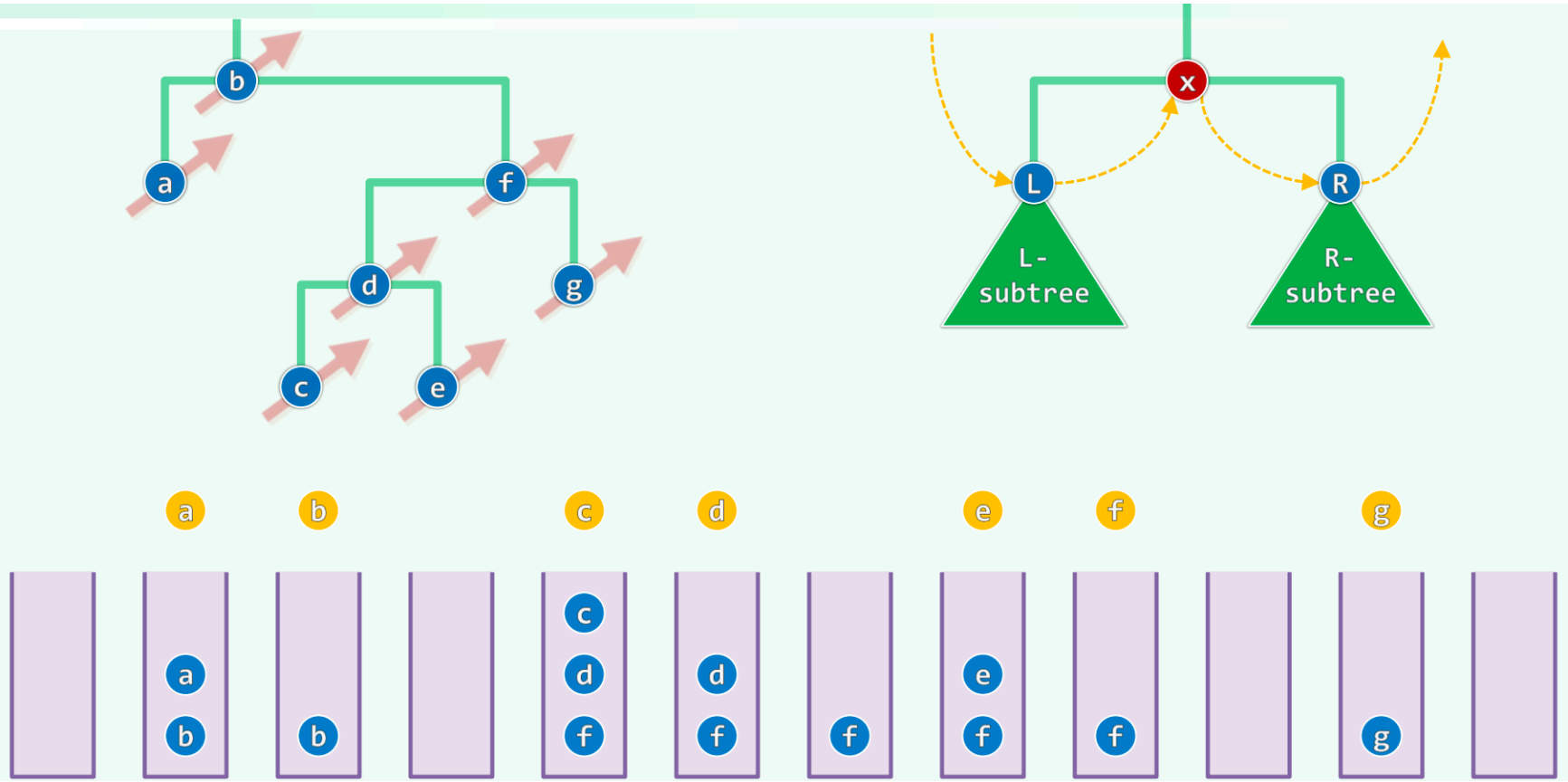
序曲

```
template <typename T, typename VST>
static void goAlongVine( BinNodePosi(T) x,
    Stack < BinNodePosi(T) > & S )
{
    while ( x )
    {
        S.push( x );
        x = x->lc;
    }
}
```

全曲

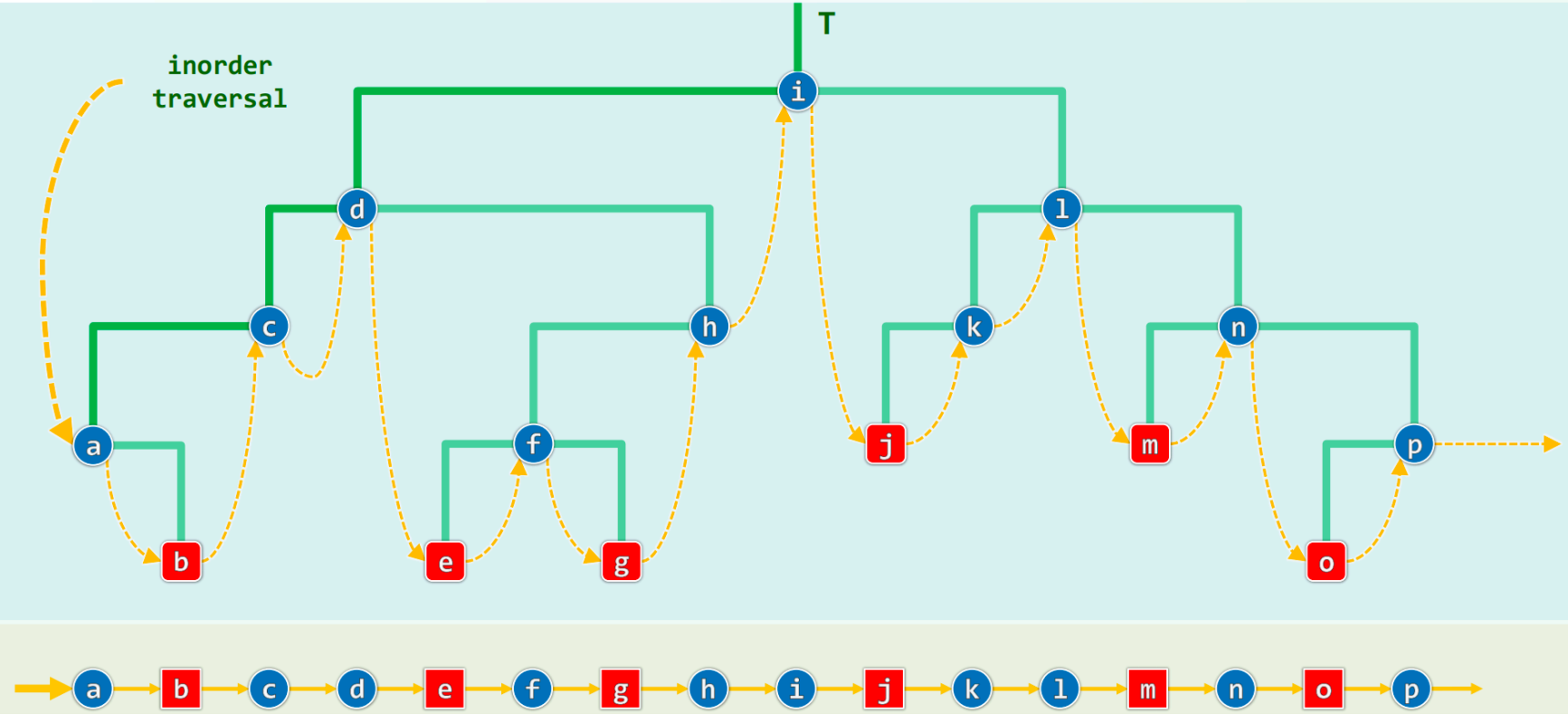
```
template <typename T, typename VST>
void travIn_I1( BinNodePosi(T) x, VST & visit )
{
    Stack < BinNodePosi(T) > S; //辅助栈
    while ( true ) //以右子树为单位，逐批访问节点
    {
        //从当前节点出发，逐批入栈
        goAlongVine( x, S );
        //直至所有节点处理完毕
        if ( S.empty() ) break;
        //x的左子树或为空，或已遍历（等效于空）
        x = S.pop();
        visit( x->data );
        //再转向其右子树
        x = x->rc;
    }
}
```

实例



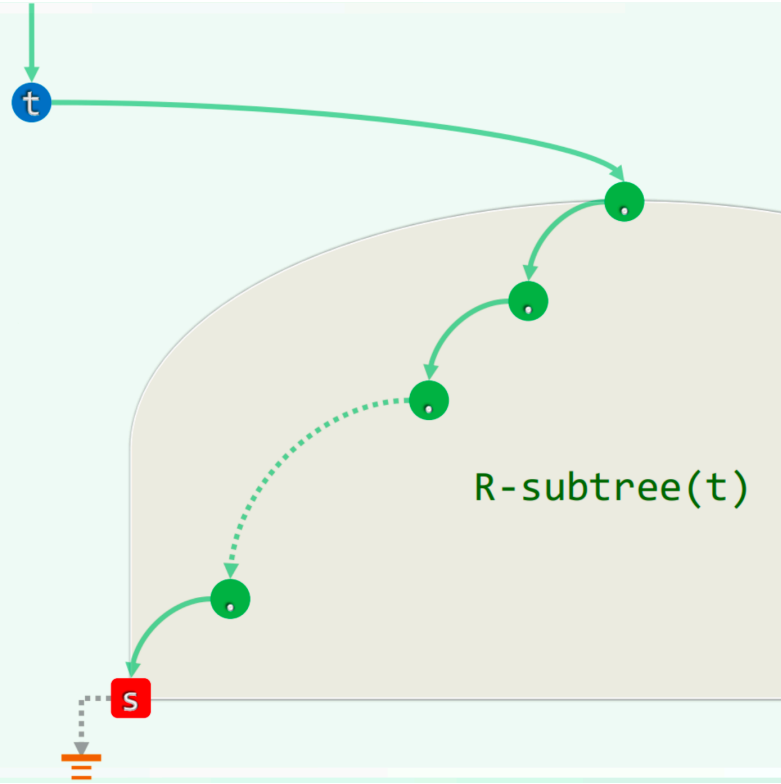
直接后继及定位

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



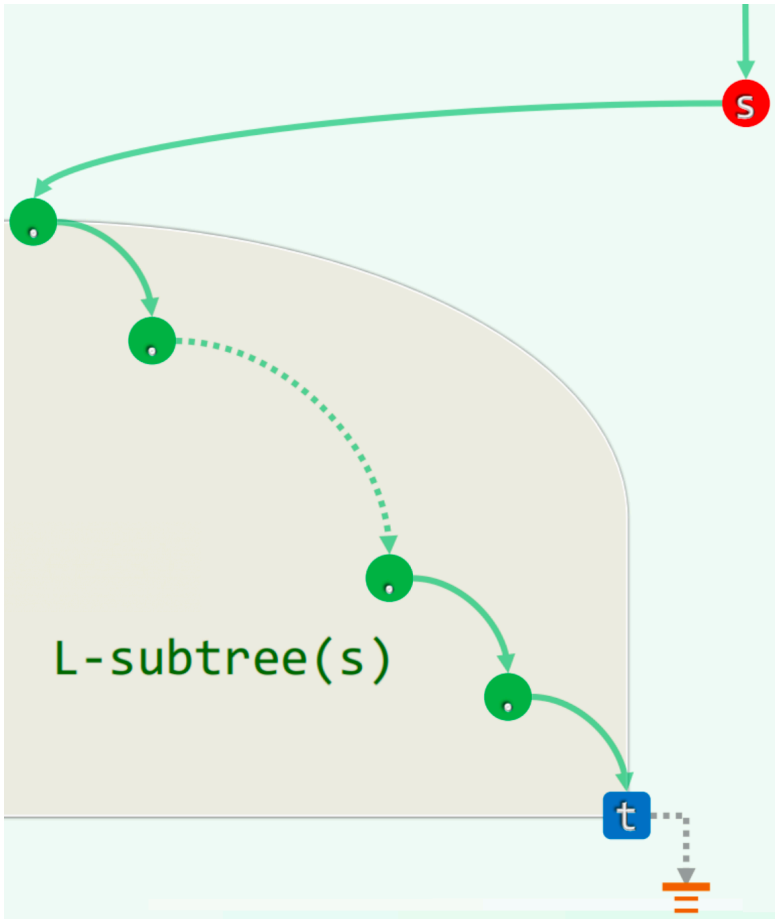
直接后继：最靠左的右后代

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



直接后继：最低的左祖先

```
    //否则后继应是
    //“将当前节点包含于
    //左子树中的最低祖先”
    else
    {
        //首先不断朝左上移动
        while (IsRChild(*s))
            s = s->parent;
        //最后朝右上移动一步
        s = s->parent;
    }
    return s;
}
```



1 二叉树及其表示

2 二叉树的实现

3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

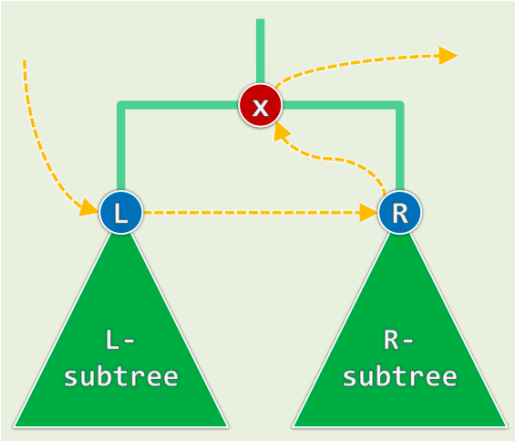
4 Huffman 编码

后序实例：子树删除

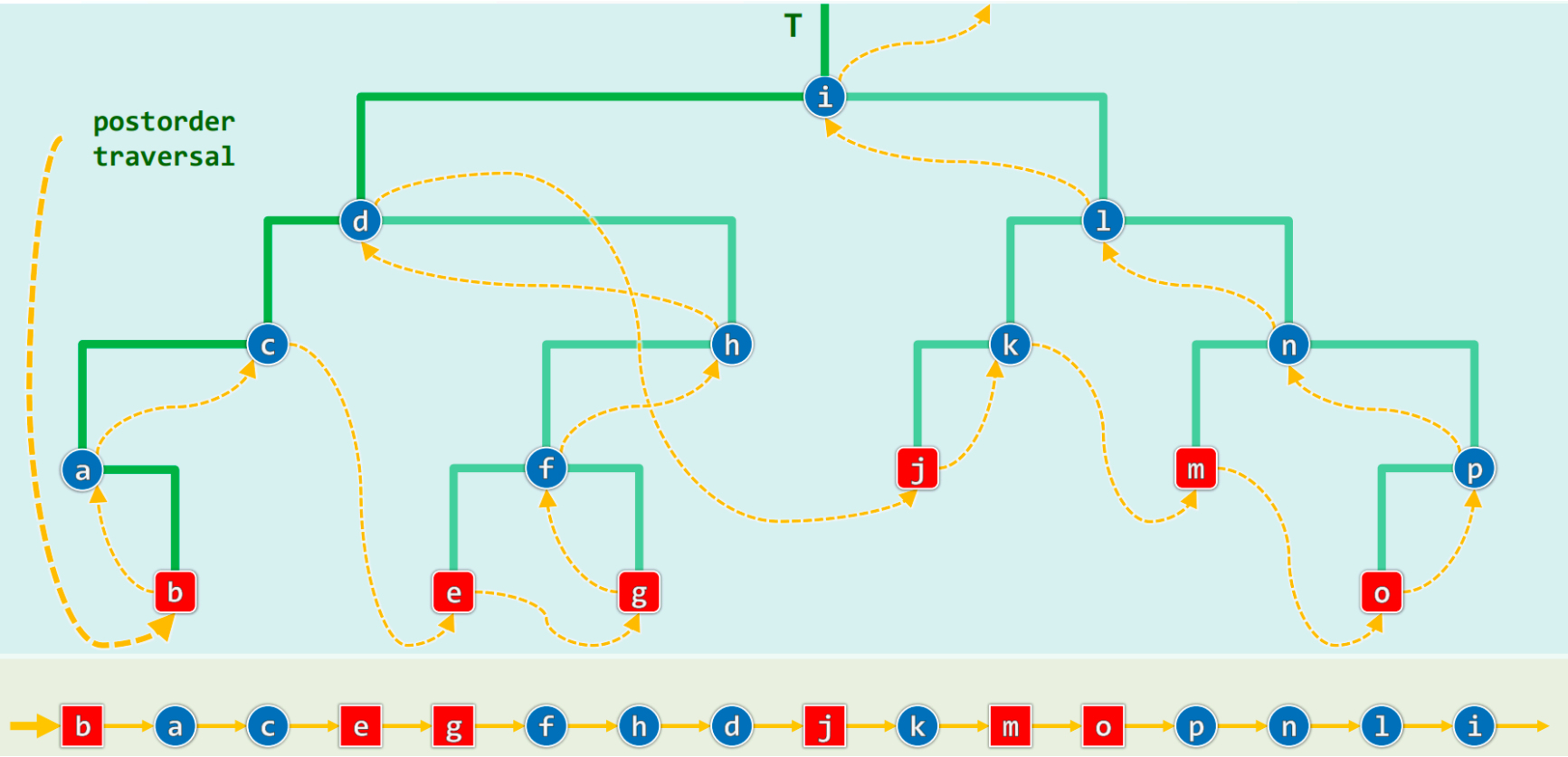
```
template <typename T>
int BinTree<T>::remove( BinNodePosi(T) x ) {
    FromParentTo(*x) = NULL;
    updateHeightAbove(x->parent);
    int n = removeAt(x); _size -= n; return n;
}

template <typename T> static int BinTree<T>::
removeAt( BinNodePosi(T) x ) {
    if (!x) return 0;
    int n = 1 + removeAt(x->lc) + removeAt(x-rc);
    release(x->data); release(x); return n;
}
```

递归实现

```
template <typename T, typename VST>
void travPost_R( BinNodePosi(T) x,
    VST & visit )
{
    if ( ! x ) return;
    travPost_R( x->lc, visit );
    travPost_R( x->rc, visit );
    visit( x->data );
} //O(n)
```

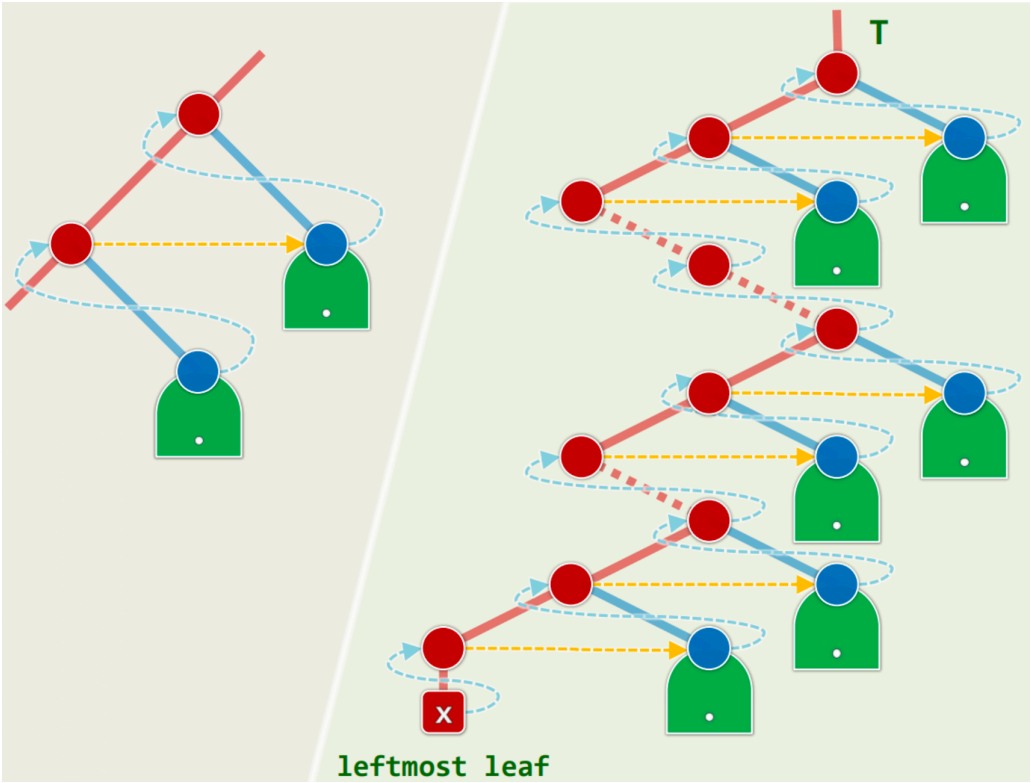


观察


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

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...



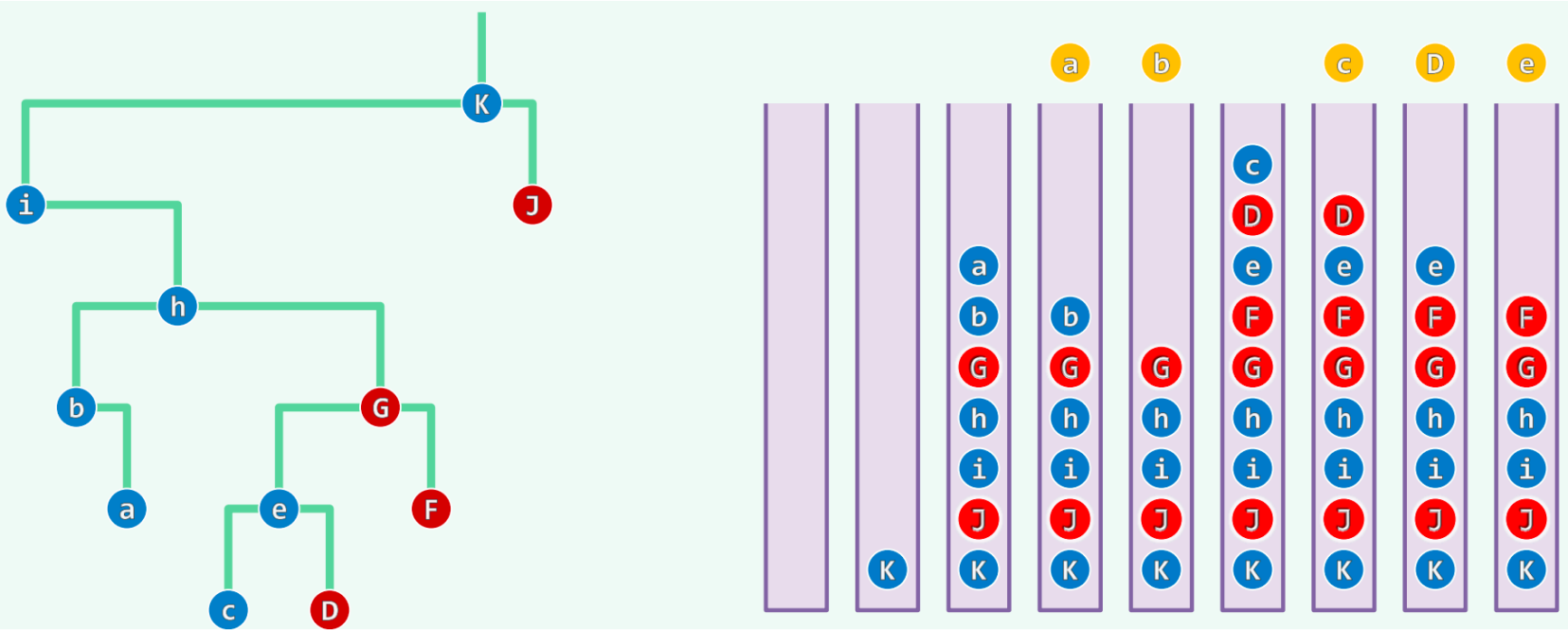
序曲

```
template <typename T>
static void goToHLVFL(Stack<BinNodePosi(T)>& S)
{
    //自顶而下反复检查栈顶节点
    while ( BinNodePosi(T) x = S.top() )
        if ( HasLChild( * x ) )
        { //尽可能向左。在此之前
            //若有右孩子，则
            if ( HasRChild( * x ) )
                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
}
```

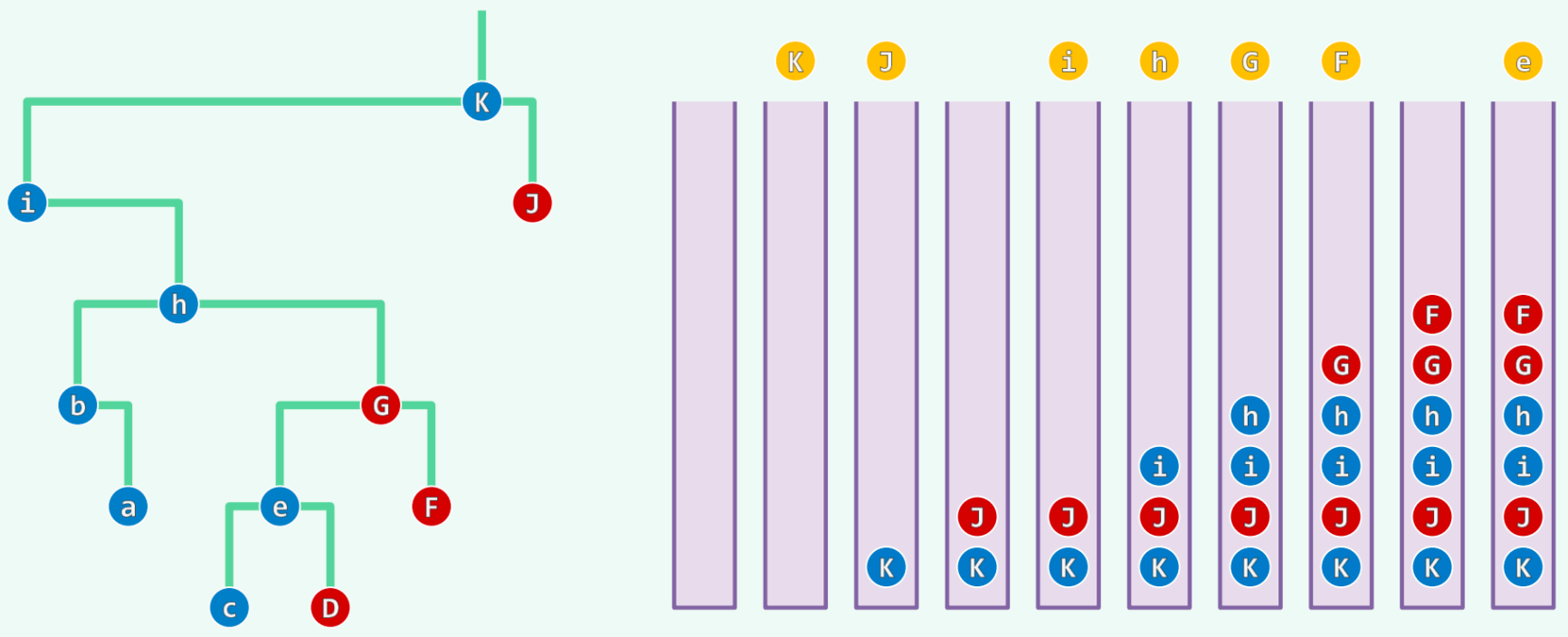
全曲

```
template <typename T, typename VST>
void travPre_I2( 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
        //若栈顶非x之父（而为右兄），则
        //在其右兄子树中找到最靠左的叶子
        if ( S.top() != x->parent )
            goToHLVFL( S );
        //弹出栈顶（即前一节点之后继）以更新x
        x = S.pop();
        visit( x->data ); //并随即访问之
    }
}
```

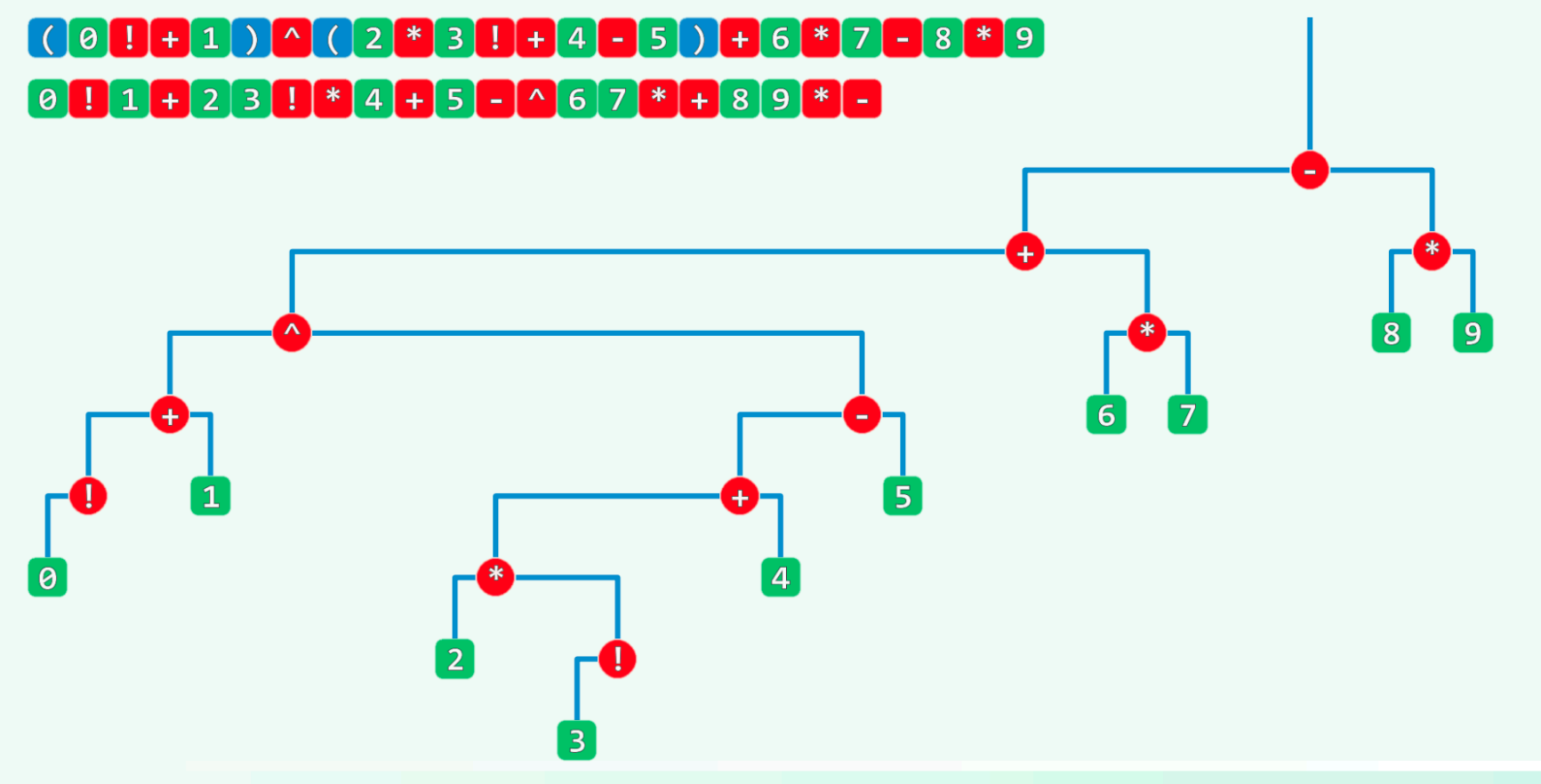
实例（1/2）


K
i
J
h
b
G
a
e
F
c
D
a
b
c
D
e
K
a b G h i J K
b G h i J K
G h i J K
c D e F G h i J K
D e F G h i J K
e F G h i J K
F G h i J K

实例（2/2）



RPN



1 二叉树及其表示

2 二叉树的实现

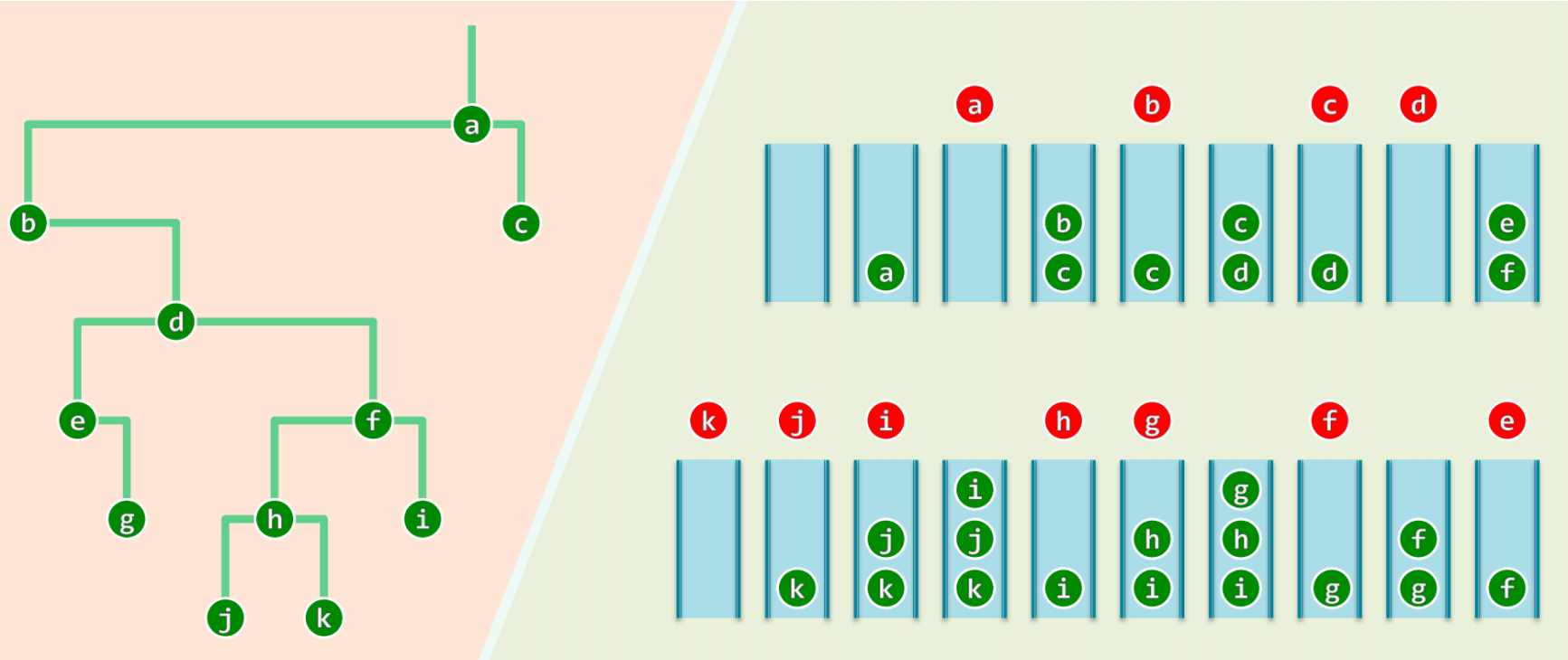
3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

4 Huffman 编码

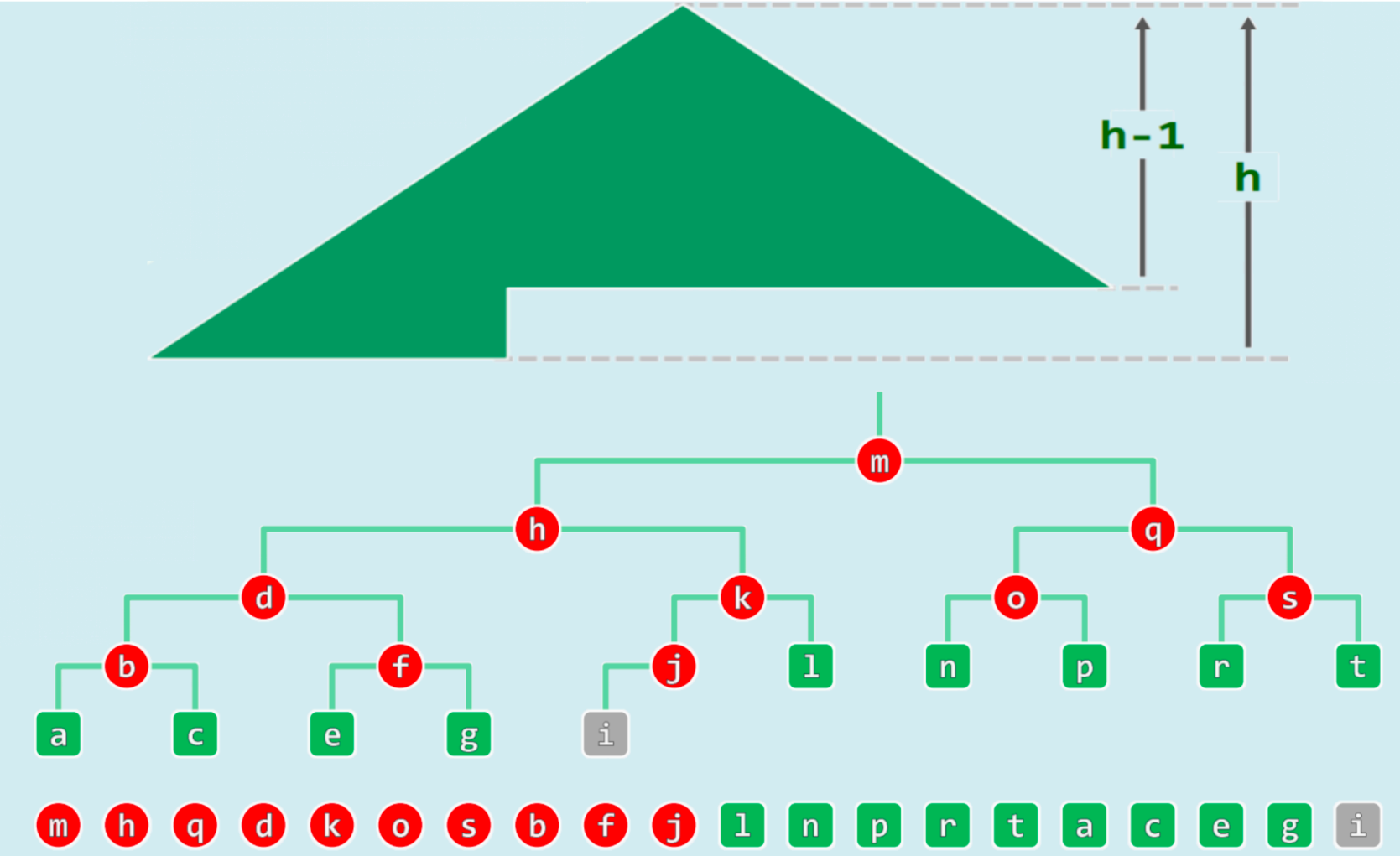
算法实现

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

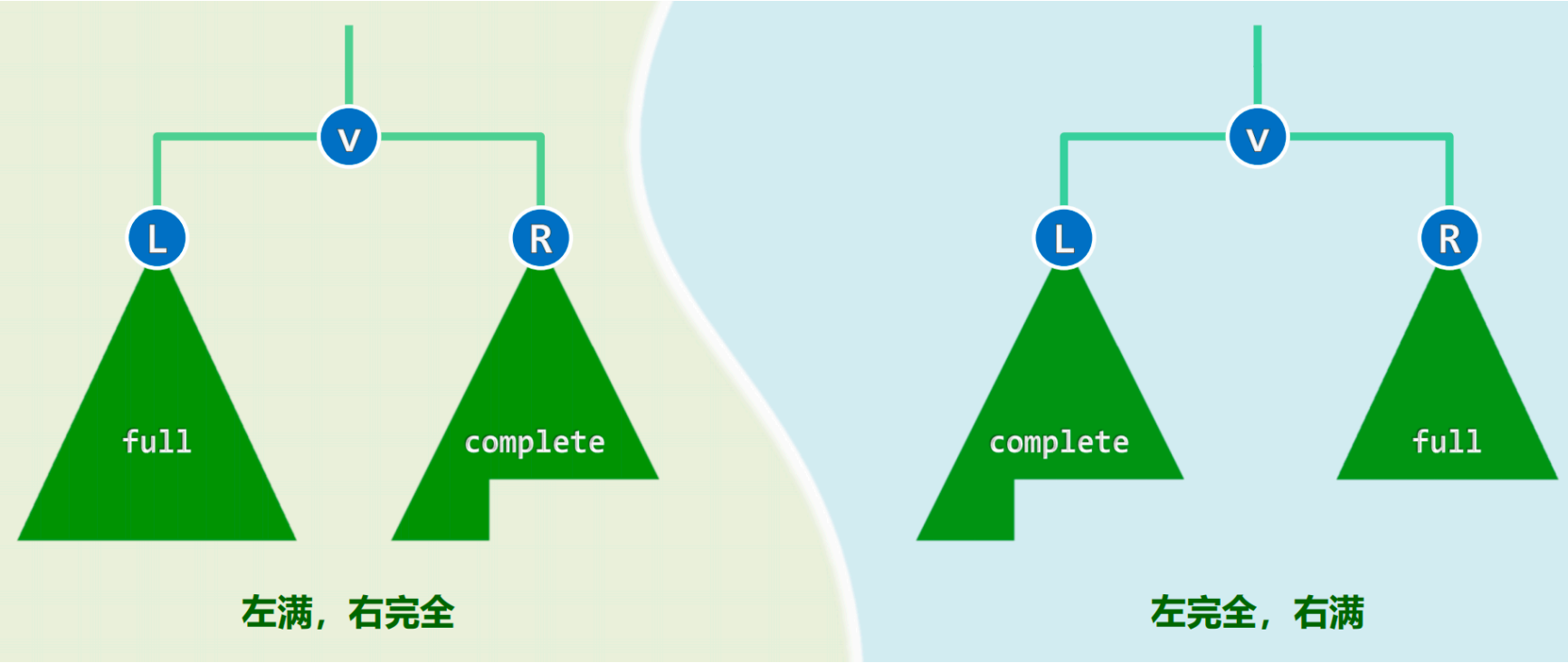
实例



完全二叉树



完全二叉树 vs 满二叉树



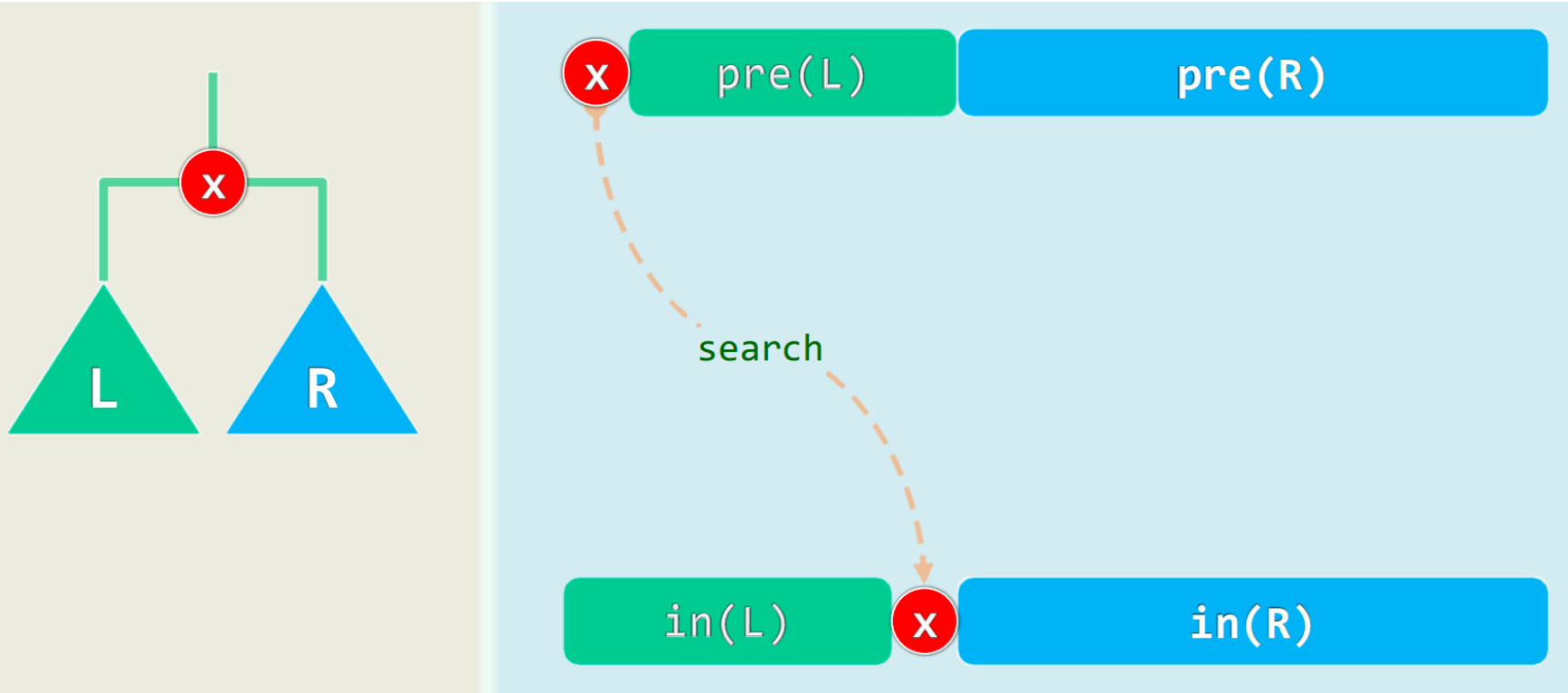
1 二叉树及其表示

2 二叉树的实现

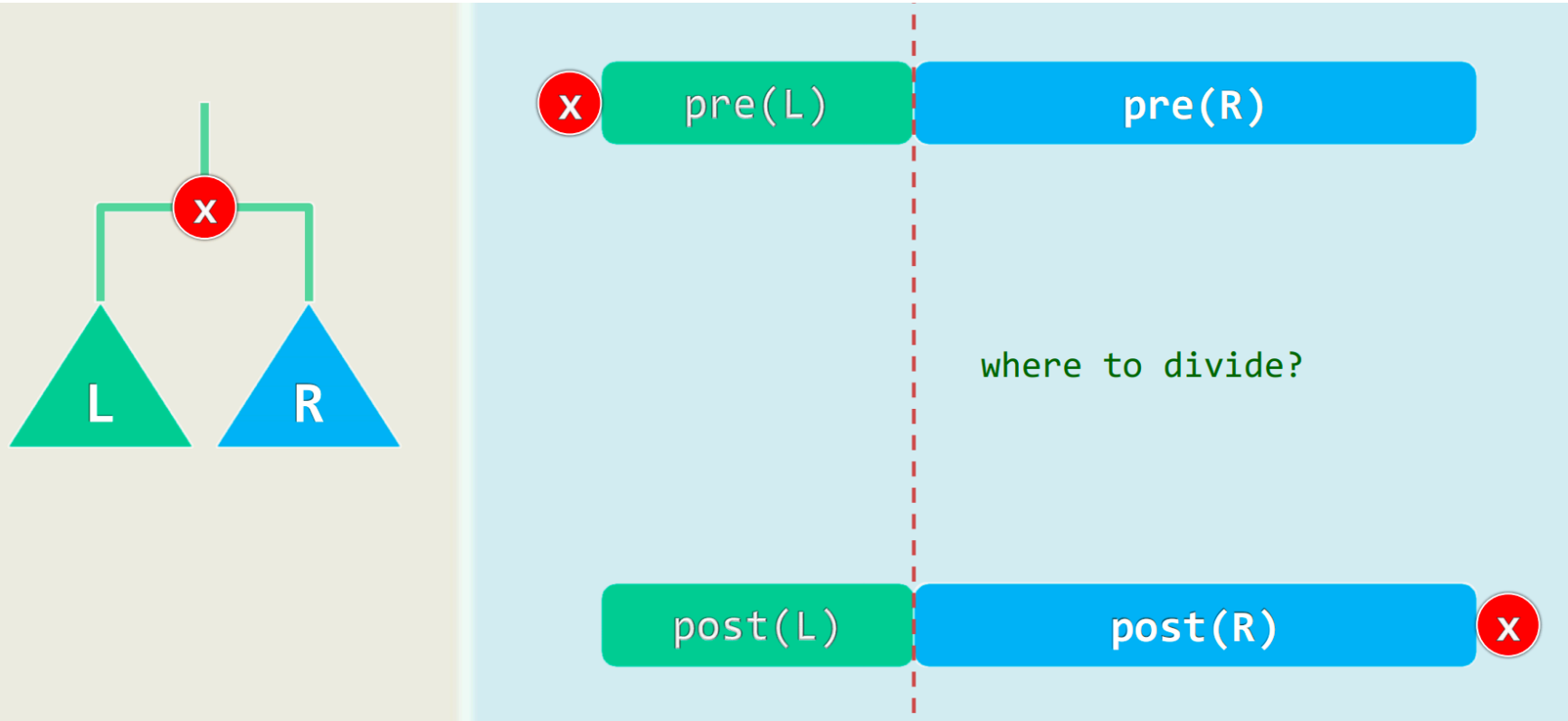
3 遍历
- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 重构

4 Huffman 编码

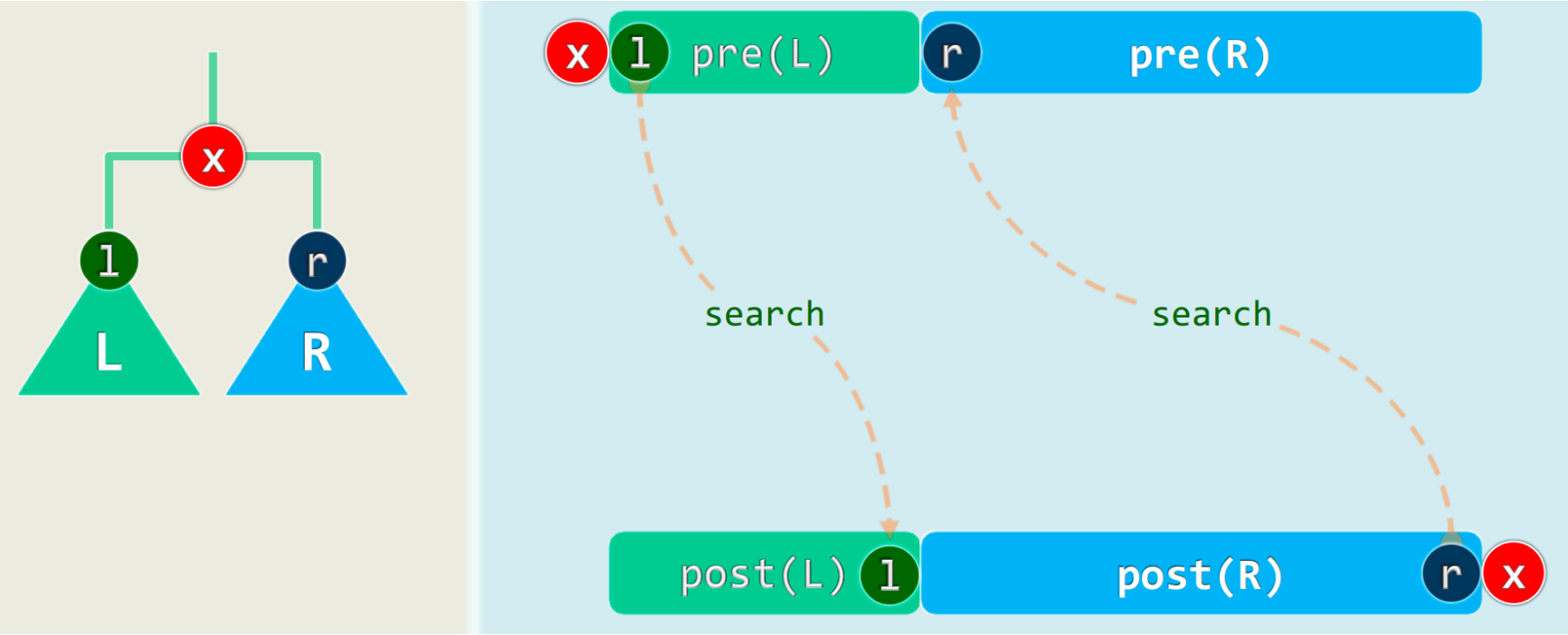
[先序 | 后序] + 中序



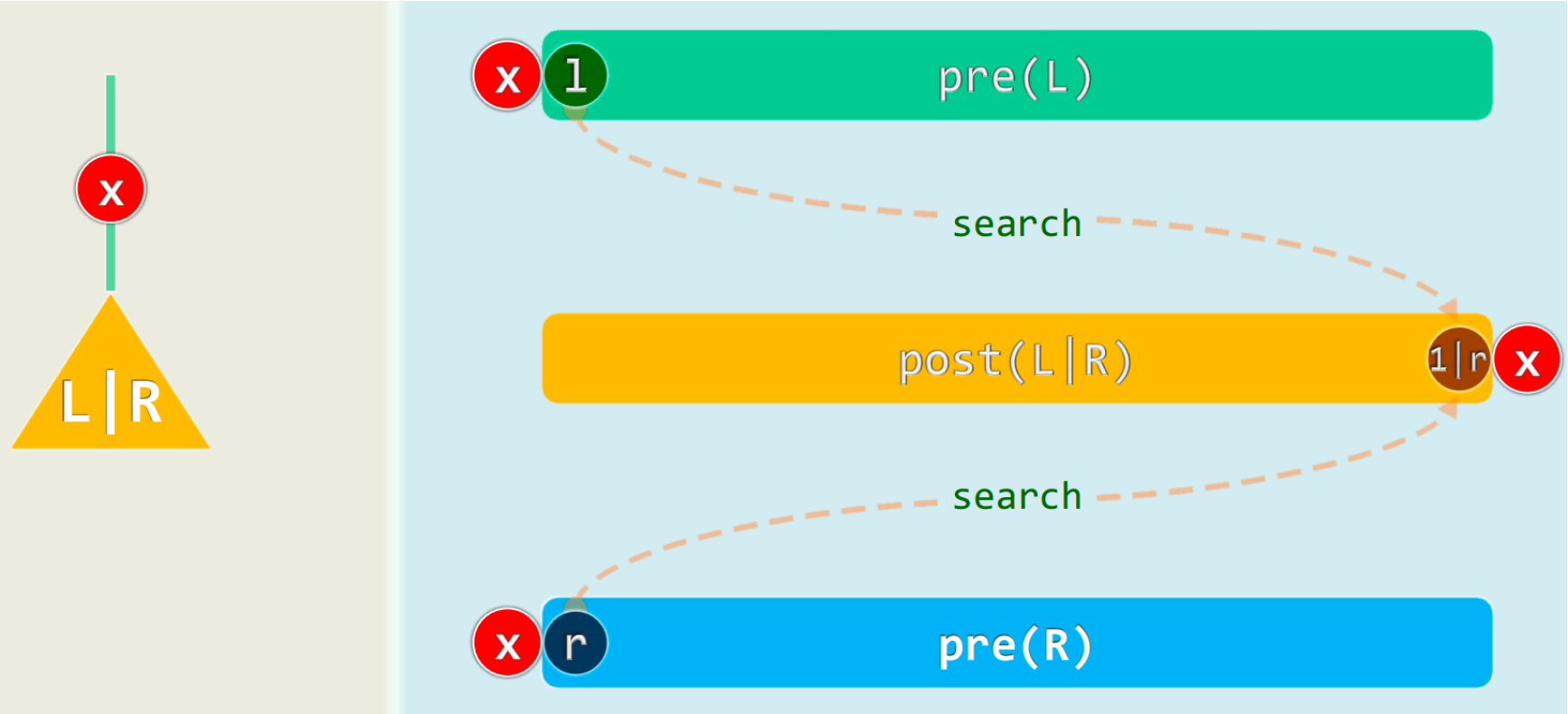
[先序 | 后序] ?



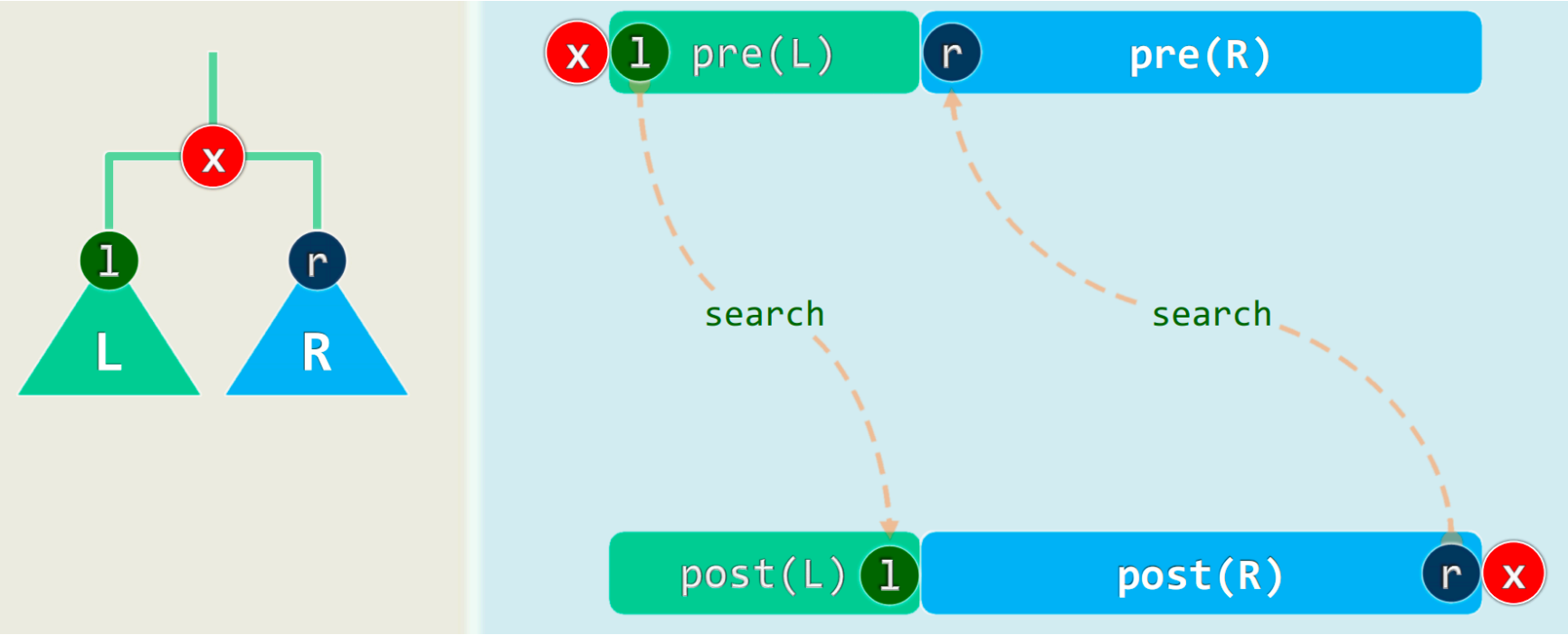
[先序 | 后序] !



[先序 | 后序] ??



[先序 | 后序] x 真!



1 二叉树及其表示

2 二叉树的实现

3 遍历

4 Huffman 编码
算法思路
算法实现

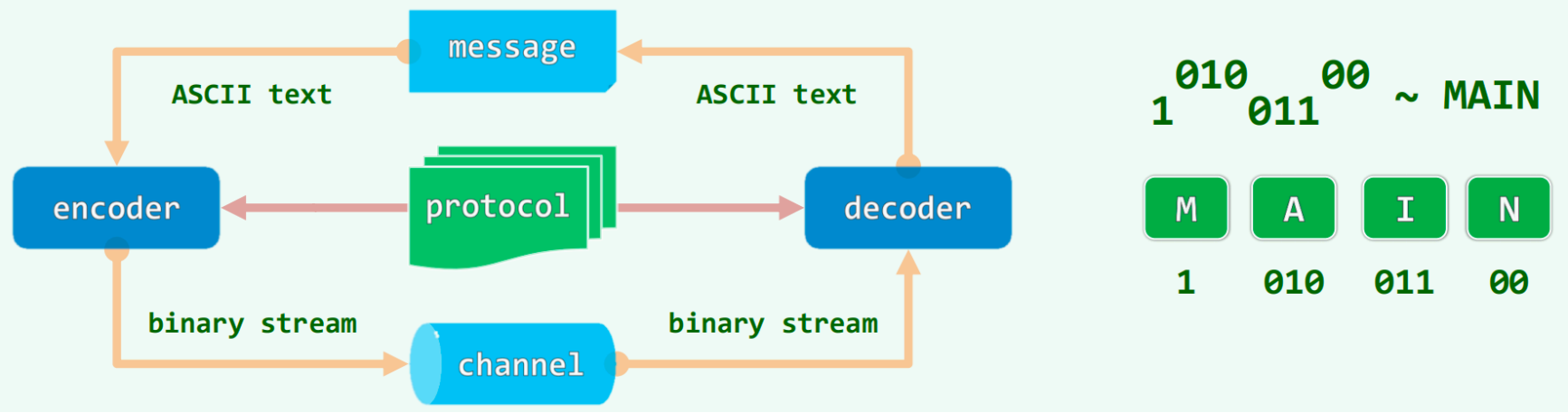
1 二叉树及其表示

2 二叉树的实现

3 遍历

4 Huffman 编码
- 算法思路
- 算法实现

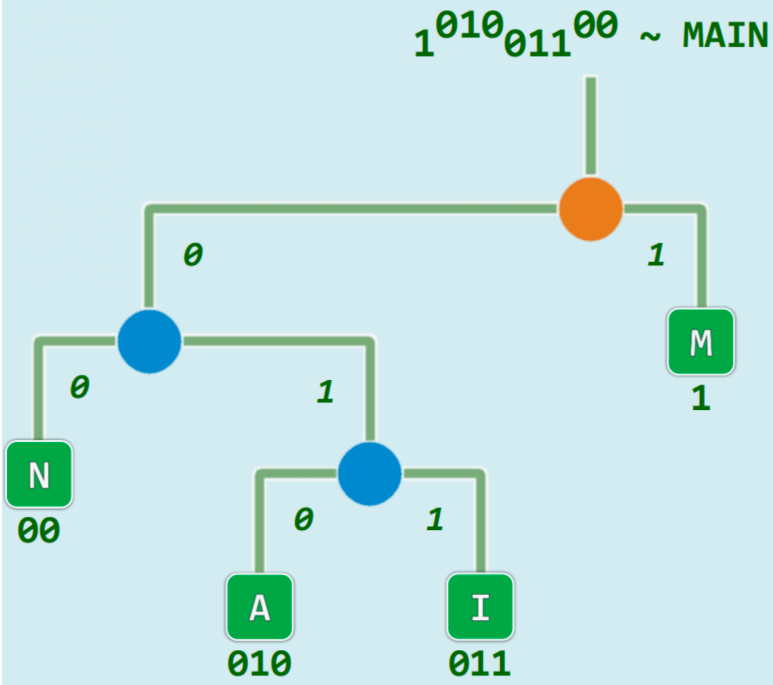
编码



- 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 × 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？

PFC 编码

- 将 $\sum$ 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
- 不同字符的编码互不为前缀，故不致歧义
- 有何缺点？

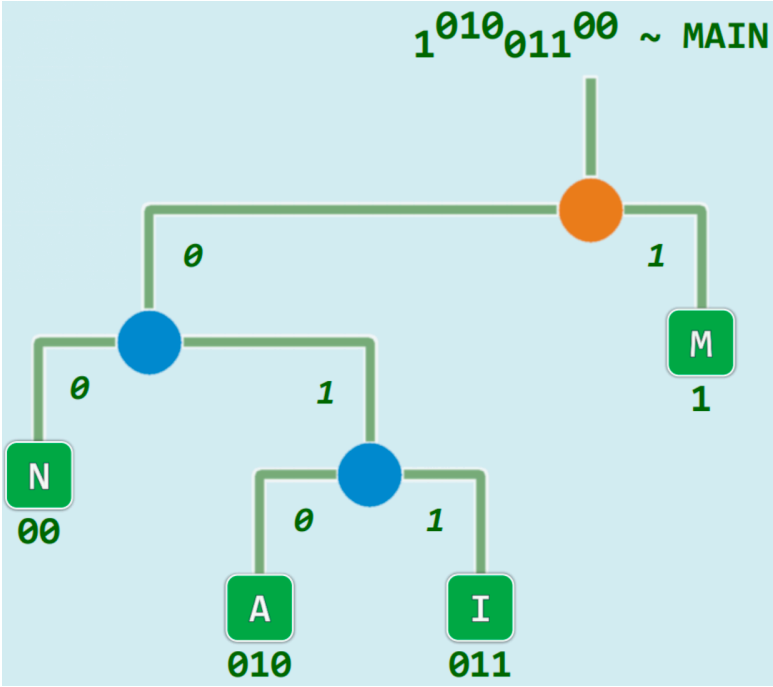


编码长度 vs. 叶节点平均深度

- 平均编码长度

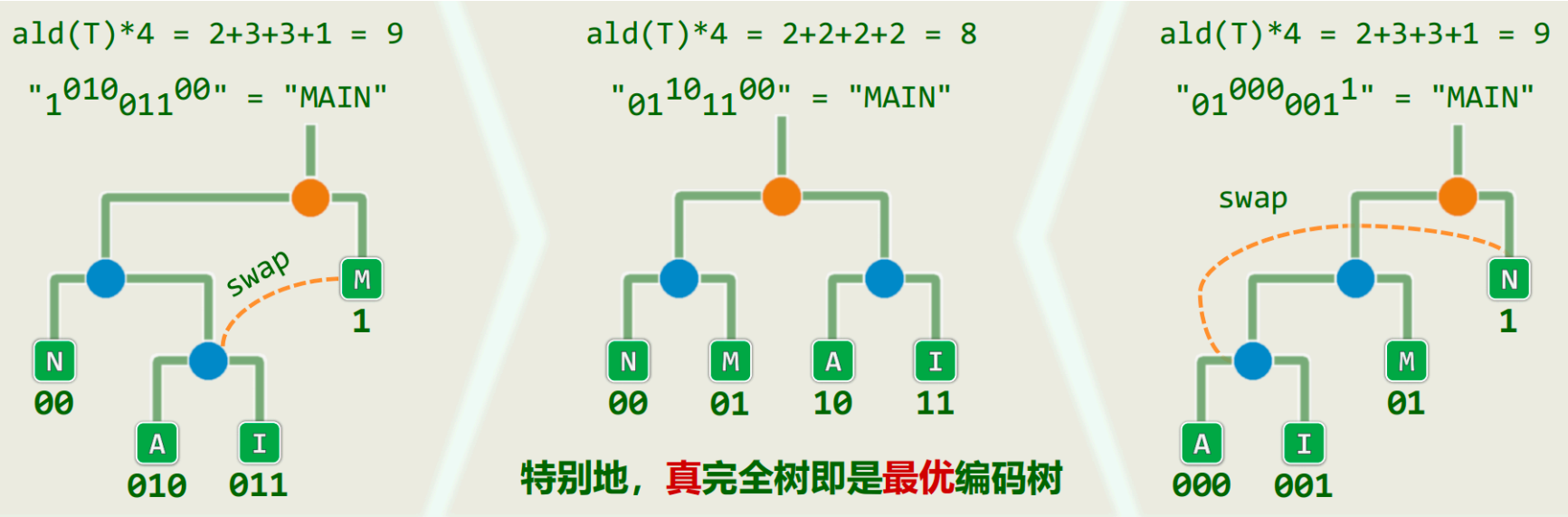
$$ald(T) = \sum_{x \in \Sigma} depth(v(x))/|\Sigma|$$

- 对于特定的字符集 Σ 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?



最优编码树

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

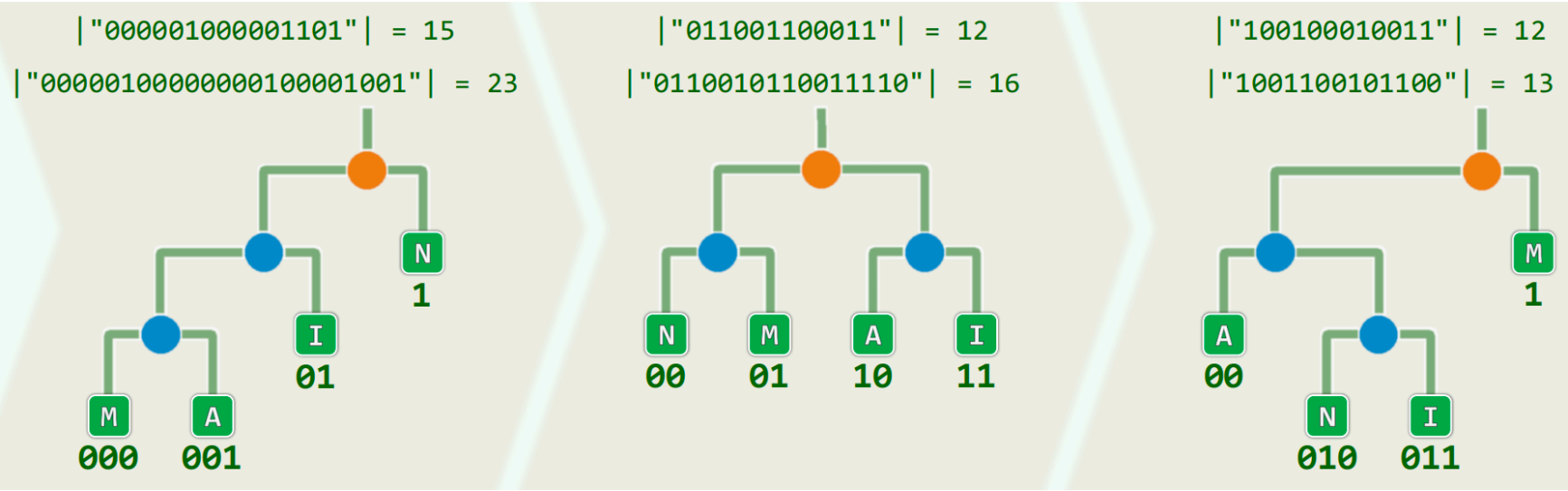


字符频率

- 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

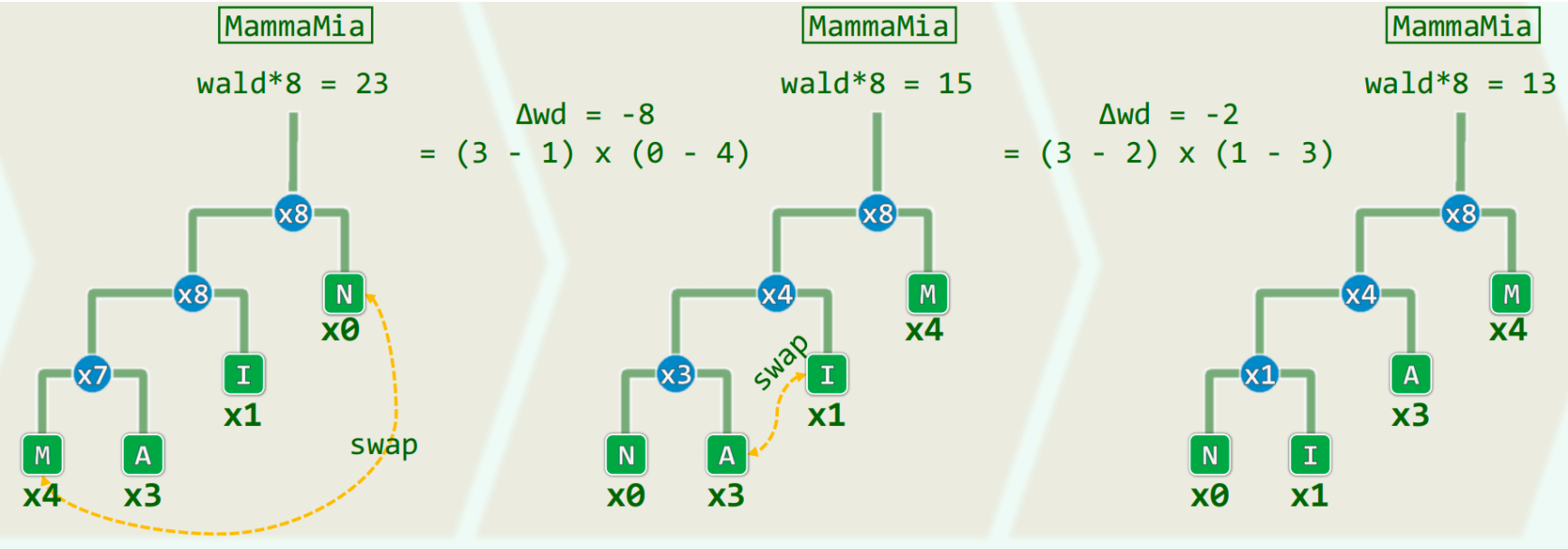
带权编码长度 vs. 叶节点平均带权深度

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



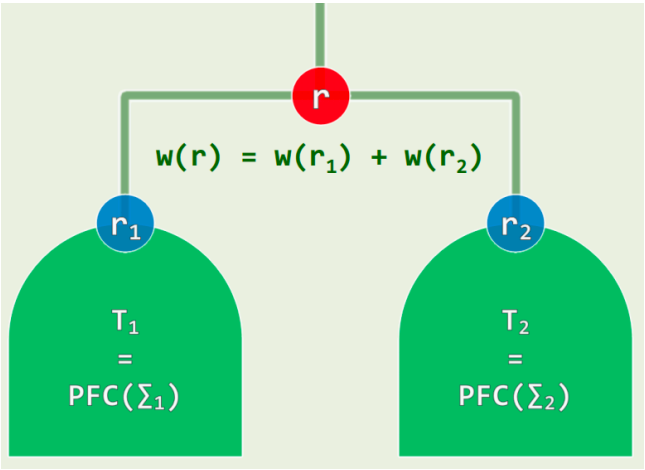
最优带权编码树

- 频率高/低的（超）字符，应尽可能放在高/低处
- 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

1. 为每个字符创建一棵单节点的树，组成森林 F
2. 按照出现频率，对所有树排序
3. while (F 中的树不止一棵)
   1. 取出频率最小的两棵树：T_1 和 T_2
   2. 将它们合并成一棵新树 T，并令：
   3. $lc(T) = T_1$ 且 $rc(T) = T_2$
   4. $w(root(T)) = w(root(T_1)) + w(root(T_2))$



https://cmps-people.ok.ubc.ca/ylucet/DS/Huffman.html

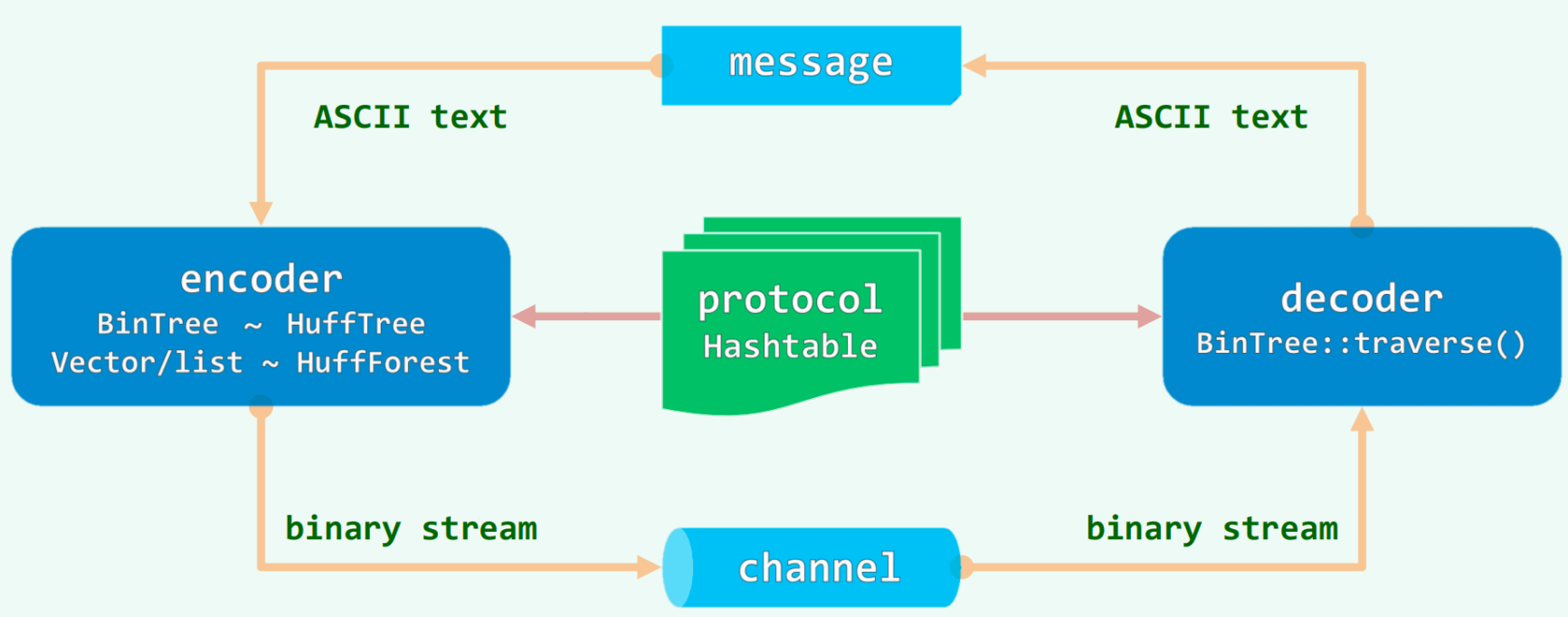
1 二叉树及其表示

2 二叉树的实现

3 遍历

4 Huffman 编码
算法思路
算法实现

数据结构与算法


message
ASCII text
ASCII text
encoder
BinTree ~ HuffTree
Vector/list ~ HuffForest
protocol
Hashtable
decoder
BinTree::traverse()
binary stream
binary stream
channel

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; } //判等器
};
```

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
- 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

```
//基于列表的优先级队列
using HuffForest = PQ_List< HuffTree* >;
//完全二叉堆
using HuffForest = PQ_ComplHeap< HuffTree* >;
//左式堆
using HuffForest = PQ_LeftHeap< HuffTree* >;
```

查找最小超字符：遍历 List/Vector

```
//Huffman编码的整体效率，直接决定于minHChar()
HuffTree* minHChar( HuffForest* forest ) {
    //从首节点出发，遍历所有节点
    ListNodePosi<HuffTree*> m = forest->first();
    for ( ListNodePosi<HuffTree*> p = m->succ;
          forest->valid( p ); p = p->succ )
          //找到最小节点（所对应的Huffman子树）
        if( m->data->root()->data.weight >
            p->data->root()->data.weight )
            m = p;
    //从森林中取出该子树，并返回
    return forest->remove( m );
}
```

构造编码树：反复合并二叉树

```
//Huffman编码算法
HuffTree* generateTree( HuffForest * forest ) {
    //反复迭代，直至森林中仅含一棵树
    while ( 1 < forest->size() ) {
        HuffTree *T1 = minHChar( forest );
        HuffTree *T2 = minHChar( forest );
        HuffTree *S = new HuffTree();
        S->insert( HuffChar('^',
            T1->root()->data.weight +
            T2->root()->data.weight) );
        S->attach( T1, S->root() );
        S->attach( S->root(), T2 );
        //合并之后，重新插回森林
        forest->insertAsLast( S );
    } //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
}
```