浙江傳媒學院
COMMUNICATION UNIVERSITY OF ZHEJIANG

数据结构与算法

Data Structures and Algorithms

谢昊

xiehao@cuz.edu.cn

半线性结构
Semi-Linear Structures

大纲

1. 基本术语
2. 树
3. 二叉树
    - 几种特殊的二叉树
    - 二叉树的性质
    - 二叉树的存储结构
    - 二叉树的基本操作
    - 二叉树的遍历
    - 二叉树的简单应用：Huffman 编码
4. 小结

表 1: 线性结构的优势与不足

	顺序列表	**链式列表**
访问元素	$O(1)$	$O(n)$
增删元素	$O(n)$	$O(1)$

表 1: 线性结构的优势与不足

	顺序列表	**链式列表**
访问元素	$O(1)$	$O(n)$
增删元素	$O(n)$	$O(1)$

半线性结构：可去二者之糟粕，取二者之精华

基本术语

树 (tree) 与森林 (forest)

- **半线性 (semi-linear)** 结构一般指**树**
- 树由 n 个[1]**顶点 (vertex)**[2]与连接于其间的若干条**边 (edge)** 组成
- 空树既无结点亦无边
- 非空树应满足如下条件
  - 有且仅有 1 个特定结点为**根 (root)** 结点
  - 除根结点外的其余结点被分为 d 个[3]互不相交 的**子树 (subtree)**
  - 子树与根之间由边相连，但不形成**环 (ring)**
- 子树亦为树，满足上述性质（递归定义）
- m 棵[4]互不相交的树的集合为**森林**

[1] $n \in \mathbb{Z} \cap [0, +\infty)$
[2] 又名**结点 (node)**
[3] $d \in \mathbb{Z} \cap [0, +\infty)$
[4] $m \in \mathbb{Z} \cap [0, +\infty)$

基本术语

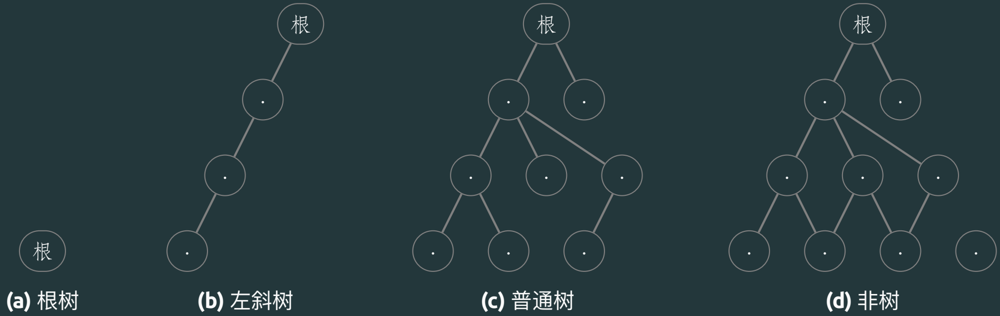


图 1: 几种树与非树

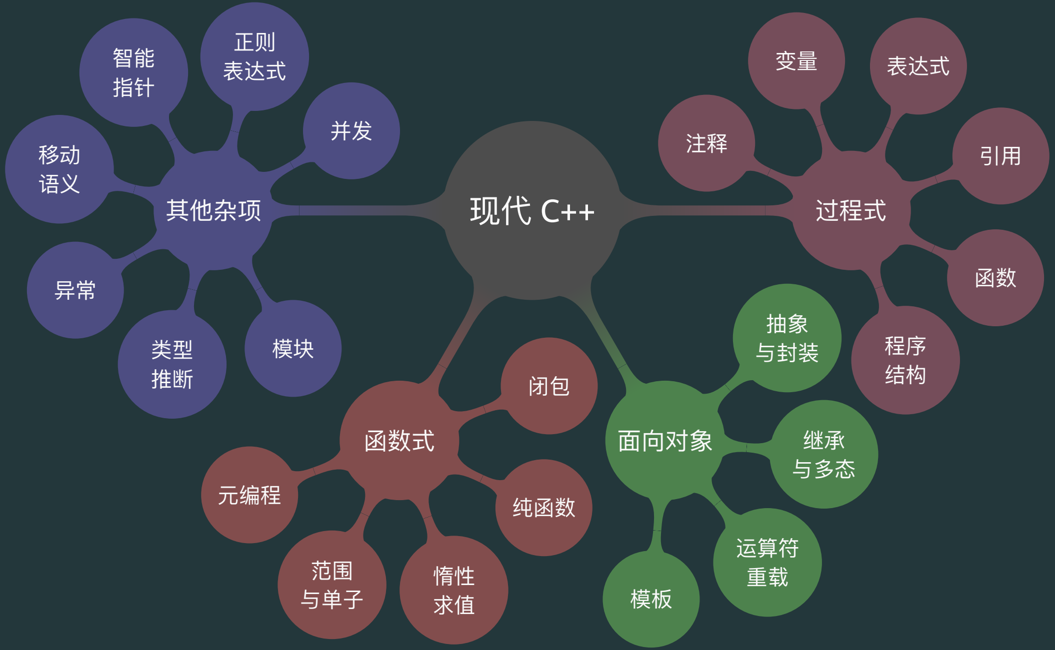

现代 C++
其他杂项
智能
指针
正则
表达式
并发
移动
语义
异常
类型
推断
模块
过程式
变量
表达式
注释
引用
函数
程序
结构
函数式
闭包
纯函数
元编程
范围
与单子
惰性
求值
面向对象
抽象
与封装
继承
与多态
运算符
重载
模板

图 2: 思维导图亦为树

树的特点

- 根结点无前驱，其余结点有且仅有 1 个直接前驱
- 所有结点均可有 n 个[5] 直接后继
- 前驱类似线性，后继则不同，故称半线性

[5] $n \in \mathbb{Z} \cap [0, +\infty)$

度 (degree)

- 结点的**度**指其子树个数
- 树的**度**指其最大结点度
- **叶 (leaf)** 结点度为 0，亦称**终端**结点
- 其余结点为**分支**结点

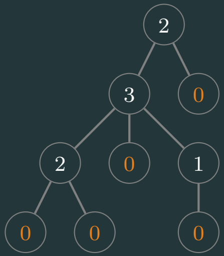


图 3: 结点的度

结点亲缘关系

- 结点的子树根为该结点的**子 (child)** 结点

$$b = child(a), \quad c = child(a)$$

- 该结点为子树根的**父 (parent)** 结点

$$a = parent(b), \quad a = parent(c)$$

- 同一结点的所有子结点互为**兄弟 (sibling)**

$$b = sibling(c), \quad c = sibling(b)$$

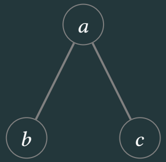


图 4: 结点间的关系[6]

[6] a 为 b 或 c 的父结点，b 或 c 为 a 的子结点，b 与 c 互为兄弟

路径 (path)、深度 (depth) 与高度 (height)

- 若结点序列 $\{n_i\}_{i=0}^{k}$ 满足：

$$n_i = parent(n_{i+1}), \quad i = 0, 1, \cdots, k-1$$

  则称之为自 n_0 至 n_k 的一条**路径**
  - 所过边数为路径**长度 (length)**
  - 若存在自 n_a 至 n_b 的路径，则该路径唯一，且 n_a 为 n_b 的**祖先 (ancestor)**，n_b 为 n_a 的**子孙 (descendant)**
- 结点**深度**[7]为根至其的路径长度
- 结点**高度**为其最大子孙深度[8]
  - 树的高度为其根的高度

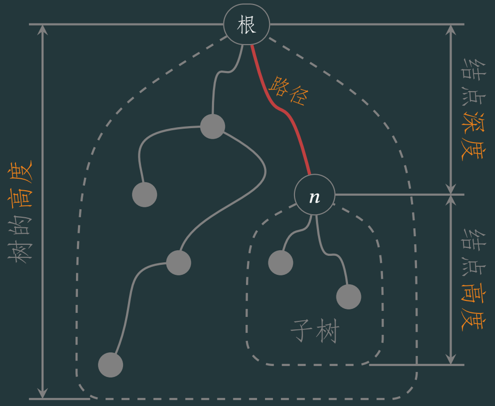


图 5: 路径、高度与深度

[7] 又称结点所在**层数 (layer)**，根在第 0 层

[8] 此处范围仅限于以其为根的子树内，一般为该子树最大叶深

与线性结构的比较

线性	半线性
首元素无前驱	根结点无父结点
尾元素无后继	叶结点无子结点
其他元素单前驱单后继	其他结点单父结点多子结点

树

树的存储结构

- 可采用顺序或链式存储结构
- 每结点须记录：数据信息、与其他结点的逻辑关系

树的存储结构

- 可采用顺序或链式存储结构
- 每结点须记录：数据信息、与其他结点的逻辑关系

树的结点关系表示方法

- 父结点表示法：只记录父结点信息
- 子结点表示法：只记录子结点信息
- 父子结点表示法：同时记录父子结点信息
- 长子兄弟表示法：同时记录第一个子结点与兄弟结点信息

父结点表示法

- 采用数组按层存储各结点
- 每结点包括数据信息与父结点序号

复杂度

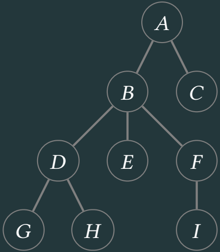
- 空间：$O(n)$
- 时间：
  - 查找父结点 $O(1)$
  - 但查找子结点$O(n)$

i	data	parent

图 6: 父结点表示法

```c
typedef struct {
    DataType data; // 数据信息
    int parent; // 父结点序号
} TreeNode;
```

树



(a) 逻辑表示

0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	3
7	H	3
8	I	5

(b) 存储表示

图 7: 父结点表示法

树

子结点表示法

- 采用数组按层存储各结点
- 每结点包括数据信息与子结点序号链表

i	data	children

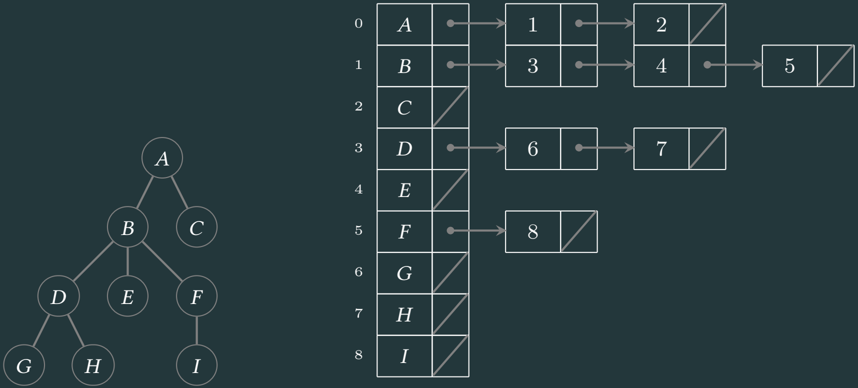
图 8: 子结点表示法

复杂度

- 空间：$O(n)$
- 时间：
  - 查找子结点 $O(d)$[9]
  - 但查找父结点$O(n)$

```c
typedef struct {
    DataType data; // 数据信息
    LinkedList children; // 子结点序号链表
} TreeNode;
```

[9]若该结点度数为 d



(a) 逻辑表示　　**(b)** 存储表示

图 9: 子结点表示法

树

父子结点表示法

- 结合上述二者

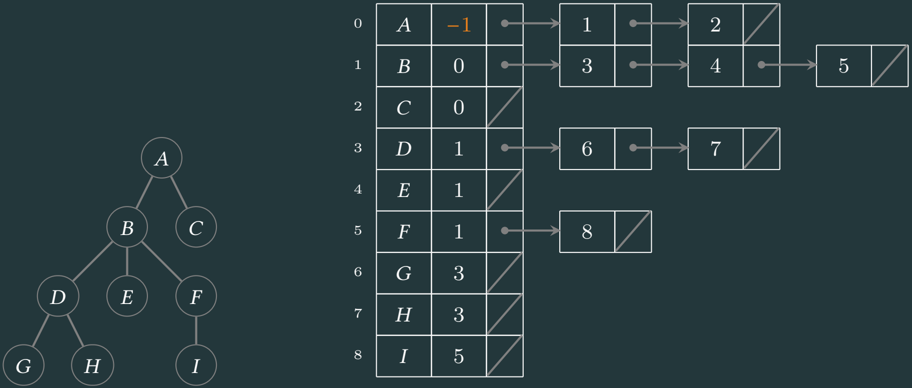
复杂度

- 空间： $O(n)$
- 时间：
  - 查找子结点 $O(d)$
  - 但查找父结点 $O(1)$

i	data	parent	children

图 10: 父子结点表示法

```c
typedef struct {
    DataType data; // 数据信息
    int parent; // 父结点序号
    LinkedList children; // 子结点序号链表
} TreeNode;
```



(a) 逻辑表示 (b) 存储表示

图 11: 父子结点表示法

父子结点表示法的性质

- **优势**：一定程度上兼顾了查找效率
- **不足**：插入/删除结点操作需大量修改链表，效率偏低

父子结点表示法的性质

- **优势**：一定程度上兼顾了查找效率
- **不足**：插入/删除结点操作需大量修改链表，效率偏低

基本术语

- 若同一结点的所有子结点间具备某种线性次序，则称之为**有序树 (ordered tree)**
- 有序树的任意非叶结点均有且仅有 1 个**长子 (eldest son)**

长子兄弟表示法

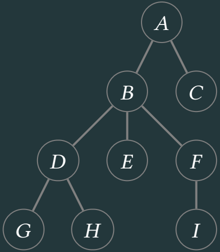
- 采用数组按层存储各结点
- 每结点包括
  - 数据信息
  - 长子结点序号
  - 首个兄弟结点序号

i	data	eldest_son	sibling

图 12: 长子兄弟表示法

```c
typedef struct {
    DataType data; // 数据信息
    int eldest_son; // 长子结点序号
    int sibling; // 兄弟结点序号
} TreeNode;
```

树



(a) 逻辑表示

0	A	1	-1
1	B	3	2
2	C	-1	-1
3	D	6	4
4	E	-1	5
5	F	8	-1
6	G	-1	7
7	H	-1	-1
8	I	-1	-1

(b) 存储表示

图 13: 长子兄弟表示法

二叉树

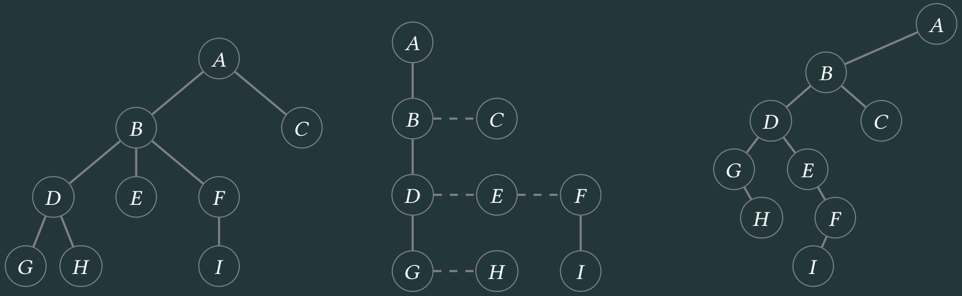
二叉树 (binary tree)

- 度不大于 2 的有序树
- 子结点可按左右区分

将树转化为二叉树

- 令长子为左子结点、首个兄弟为右子结点
- 任何树均可按此法转化为二叉树
- 因二叉树的表示与运算相对方便，故树的问题均可转化为二叉树形式进行研究

二叉树

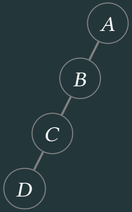


(a) 树表示　(b) 中间表示　(c) 二叉树表示

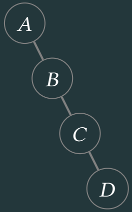
图 14: 将树转化为二叉树

左斜树[10]

- 所有非叶结点均有且仅有 1 个左子树
- 每层均有且仅有 1 个结点
- 已退化为线性结构



(a) 左斜树



(b) 右斜树

图 15: 斜树

[10]右斜树与之类似，只需将左改作右

几种特殊的二叉树

满二叉树 (full binary tree)

- 所有叶结点均在最后一层
- 所有非叶结点度均为 2

性质

- 同深度的二叉树中满二叉树结点最多
- 同深度的二叉树中满二叉树叶结点最多

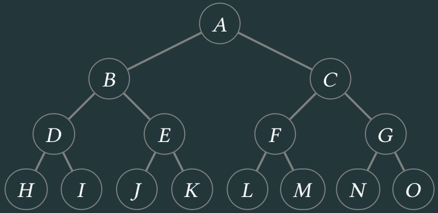
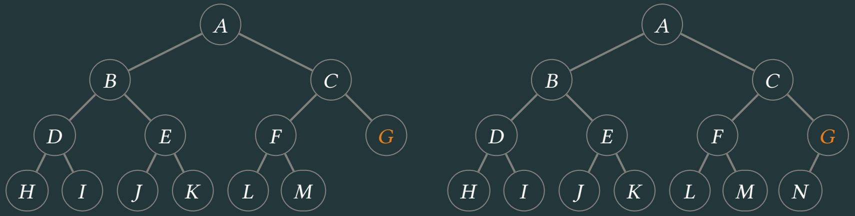


图 16: 满二叉树

几种特殊的二叉树



(a) 叶结点不在同层 **(b)** 个别非叶结点度不为 2

图 17: 非满二叉树

完全二叉树 (proper binary tree)

- 若去除最后一层结点，则为满二叉树
- 最后一层结点自左至右连续[11]排列

性质

- 同结点数的二叉树中完全二叉树最矮
- 满二叉树亦为完全二叉树的一种

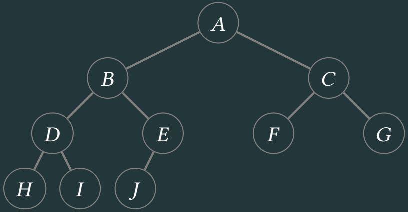


图 18: 完全二叉树

[11]指中间无空结点

性质甲

- 令 N 层二叉树第 k 层结点数为 $n_l(k)$，则有

$$1 \le n_l(k) \le 2^k,\ k \in \mathbb{Z} \cap [0, N)$$

性质甲

- 令 N 层二叉树第 k 层结点数为 $n_l(k)$，则有

$$1 \le n_l(k) \le 2^k,\ k \in \mathbb{Z} \cap [0, N)$$

证明

1. 左侧不等式显然成立
2. 右侧不等式当 $k = 0$ 时，第 0 层仅有根结点，故 $n_l(0) = 1 = 2^0$ 显然成立
3. 假设当 $k = n < N - 1$ 时成立，即 $n_l(n) \le 2^n$，因所有结点的度均不大于 2，故

$$n_l(n+1) \le 2 \cdot n_l(n) \le 2 \cdot 2^n = 2^{n+1}$$

于是当 $k = n + 1$ 时，归纳假设成立 □

性质乙

- 令 k 层二叉树的结点总数为 $n(k)$，则有

$$k \leq n(k) \leq 2^k - 1$$

性质乙

- 令 k 层二叉树的结点总数为 $n(k)$，则有

$$k \leq n(k) \leq 2^k - 1$$

证明

1. 由性质甲，$1 \leq n_l(i) \leq 2^i$，$i \in \mathbb{Z} \cap [0, k)$
2. 经累加后，有

$$k = \sum_{i=0}^{k-1} 1 \leq n(k) = \sum_{i=0}^{k-1} n_l(i) \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

$\square$

二叉树的性质

思考

- 满足 $n(k) = k$ 的二叉树一定是斜树么?
- 满足 $n(k) = 2^k - 1$ 的二叉树一定是满二叉树么?

性质丙

- 令二叉树中度为 d 的结点数为 n_d，$(d \in \{0, 1, 2\})$，则有

$$n_0 = n_2 + 1$$

性质丙

- 令二叉树中度为 d 的结点数为 n_d，$(d \in \{0,1,2\})$，则有

$$n_0 = n_2 + 1$$

证明

1. 一方面，结点总数由各种度的结点构成，故总结点数 n 满足

$$n = \sum_{d=0}^{2} n_d = n_0 + n_1 + n_2$$

2. 另一方面，每个非根结点均由 1 个结点生成，故度为 d 的结点可生成 d 个结点，即

$$n = \sum_{d=0}^{2} d \cdot n_d + 1 = n_1 + 2n_2 + 1$$

3. 综上得证 □

思考

- 有 n 个结点的完全二叉树有多少叶结点?

思考

- 有 n 个结点的完全二叉树有多少叶结点?

提示

- 在完全二叉树中，度为 1 的结点数不多于 1
- 当 n 为偶数时，$n_1 = 1$，$n_0 = n/2$，$n_2 = n/2 - 1$
- 当 n 为奇数时，$n_1 = 0$，$n_0 = (n+1)/2$，$n_2 = (n-1)/2$
- 故 $n_0 = \lceil n/2 \rceil$, $n_2 = \lfloor (n-1)/2 \rfloor$

性质丁

- 具有 n 个结点的完全二叉树层数 $k = \lfloor \log_2 n \rfloor + 1$

性质丁

- 具有 n 个结点的完全二叉树层数 $k = \lfloor \log_2 n \rfloor + 1$

证明

1. 由完全二叉树性质与性质乙可得，k 层完全二叉树结点数 n 满足
$$2^{k-1} - 1 < n \le 2^k - 1 \text{ 或 } 2^{k-1} \le n < 2^k$$
2. 取对数
$$k - 1 \le \log_2 n < k \text{ 或 } \log_2 n < k \le \log_2 n + 1$$
3. 注意到 $k \in \mathbb{Z}$，于是得证 $\square$

二叉树的性质

完全二叉树按层序编号

- 可为完全二叉树结点按层序依次编号
- 约定根编号为 1，依次递增
- 称如此编号为 k 的结点为**结点** k

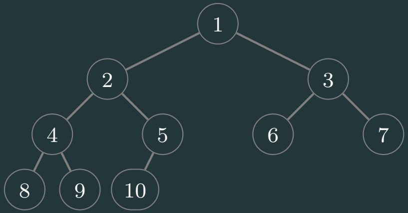


图 19: 为完全二叉树按层序编号

性质戊

- 为含有 n 个结点的完全二叉树按层序对结点编号[12]，则有
  1. 结点 k 的左右子结点序号分别为 $2k$ 与 $2k+1$, $k \in \mathbb{Z} \cap [1, \lfloor n/2 \rfloor]$
  2. 结点 k 的父结点序号为 $\lfloor k/2 \rfloor$, $k \in \mathbb{Z} \cap (1, n]$

[12]根结点为 1

性质戊

- 为含有 n 个结点的完全二叉树按层序对结点编号[12]，则有
  1. 结点 k 的左右子结点序号分别为 $2k$ 与 $2k+1$，$k \in \mathbb{Z} \cap [1, \lfloor n/2 \rfloor]$
  2. 结点 k 的父结点序号为 $\lfloor k/2 \rfloor$，$k \in \mathbb{Z} \cap (1, n]$

证明

1. 考察结论 1，当 $k = 1$ 时，显然其左右子结点序号分别为 2 与 3，成立
2. 假设当 $k = m$ 时成立，即结点 m 的左右子结点序号分别为 $2m$ 与 $2m+1$，
   - 因结点 $m+1$ 的左子结点必为结点 m 的右子结点的后继
   - 故结点 $m+1$ 的左子结点序号为 $(2m+1)+1 = 2(m+1)$
   - 且结点 $m+1$ 的右子结点序号为 $2(m+1)+1$

   则当 $k = m+1$ 时，假设成立，故结论 1 成立
3. 由结论 1 知结论 2 成立 □

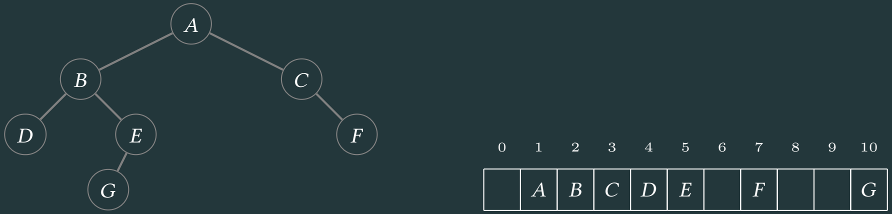
[12]根结点为 1

二叉树的存储结构

顺序存储

- 按完全二叉树层序编号方式为二叉树编号，跳过不存在结点的编号
- 以静态数组方式存储，留空不存在的结点

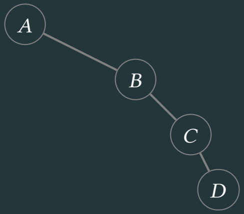
二叉树的存储结构



(a) 逻辑结构 **(b)** 物理结构

图 20: 二叉树的顺序存储示例

二叉树的存储结构



(a) 逻辑结构

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A		B				C								D

(b) 物理结构

图 21: 右斜树的顺序存储示例

顺序存储的特点

- 可利用性质戊快速访问各结点：$O(1)$
- 增删结点可能需要大幅调整存储
- 在存储含有稀疏结点的二叉树时需耗费大量存储空间
- 仅适合存储含有稠密结点的完全二叉树

二叉树的存储结构

链式存储

- 采用二/三叉链表表示结点
- 结点除存信息包括
  - 数据信息
  - 左、右子结点地址
  - 可选的父结点地址
- 为后续处理方便，设置虚拟首结点

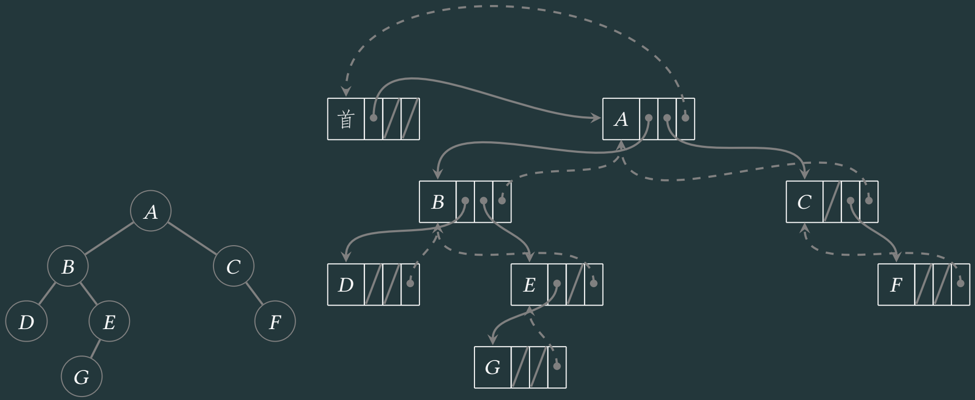
data	left	right	parent

图 22: 二叉树结点的链式存储结构

```c
typedef struct BinaryTreeNode {
    DataType data; // 数据信息
    struct BinaryTreeNode *left; // 左子结点地址
    struct BinaryTreeNode *right; // 右子结点地址
    struct BinaryTreeNode *parent; // 可选的父结点地址
} BinaryTreeNode;
```

```c
typedef struct {
    BinaryTreeNode *head; // 虚拟首结点
} BinaryTree;
```

二叉树的存储结构



(a) 逻辑结构 (b) 物理结构

图 23: 二叉树的链式存储示例

二叉树的抽象数据类型

```
ADT BinaryTree {
    数据:
        数据对象: D = {a_k | a_k ∈ 数据元素集合, k ∈ ℤ ∩ [0, n)}
        逻辑关系: 若D = ∅, 则R = ∅; 否则R = {⟨a_i, a_j⟩ | i < j; i, j ∈ ℤ ∩ [0, n); a_i, a_j ∈ D_l ∪ {a_0} 或 D_r ∪ {a_0}}[13]
    操作:
        create_binary_tree()
            构造并初始化一个空二叉树t
        insert_left_binary_tree(p, d)
            在二叉树中结点p下插入值为d的左子结点, 右子结点可简单类比, 略
        remove_left_binary_tree(p)
            在二叉树中删除结点p的左子树, 右子树可简单类比, 略
        traverse_binary_tree(t)
            以某种方式遍历二叉树t的所有结点
}
```

[13] a_0 为根结点，$\mathcal{D}_l$ 与 $\mathcal{D}_r$ 分别为其左右子树结点集合，且 $\mathcal{D}_l \cap \mathcal{D}_r = \varnothing$

二叉树的初始化

- 利用结点初始化方法
- 注意空指针处理

```c
BinaryTreeNode *create_binary_tree_node(DataType d) {
    BinaryTreeNode *n =
            malloc(sizeof(BinaryTreeNode));
    if (n) {
        n->data = d;
        n->left = n->right = n->parent = NULL;
    }
    return n;
}
```

```c
BinaryTree *create_binary_tree() {
    BinaryTree *t = malloc(sizeof(BinaryTree));
    if (t) {
        t->head = create_binary_tree_node(0);
    }
    return t;
}
```

为二叉树中的指定结点插入左子结点[14]

- 将原左子树作为新结点的左子树
- 注意更新双向链接关系的顺序
- 可将操作拆分为两部分
- 类似于双向链表的结点插入

```
BinaryTreeNode *attach_left_binary_tree(
        BinaryTreeNode *p, BinaryTreeNode *n) {
    assert(n && p);
    if (p->left) {
        n->left = p->left, p->left->parent = n;
    }
    p->left = n, n->parent = p;
    return n;
}
```

```
bool insert_left_binary_tree(
        BinaryTreeNode *p, DataType d) {
    if (!p) {
        printf("Wrong insertion place!\n");
        return false;
    }
    BinaryTreeNode *n = create_binary_tree_node(d);
    return n && attach_left_binary_tree(p, n);
}
```

[14] 右子结点可直接类比，略，下同

二叉树的基本操作

删除二叉树中的指定结点的左子树

- 注意更新双向链接关系的顺序
- 可将操作拆分为两部分
- 拆除的子树须通过遍历释放

```
BinaryTreeNode *detach_left_binary_tree(
        BinaryTreeNode *p) {
    assert(p && p->left);
    BinaryTreeNode *c = p->left;
    p->left = c->parent = NULL;
    return c;
}
```

```
bool remove_left_binary_tree(BinaryTreeNode *p) {
    if (!p) {
        printf("Wrong removal place!\n");
        return false;
    }
    BinaryTreeNode *c = detach_left_binary_tree(p);
    return cleanup_binary_tree_by_node(c);
}
```

二叉树的遍历

遍历 (traversal)

- 按某种约定顺序访问半线性结构中的所有结点
- 每个结点均被且仅被访问 1 次
- 意义：使半线性结构转化为线性结构
- 两类常见遍历方式：深度优先与广度优先
- 前者可按访问根结点的次序区分
  - **先序 (preorder) 遍历**：根结点 $\Rightarrow$ 子树序列[15]
  - **中序 (inorder) 遍历**[16]：左子树 $\Rightarrow$根结点 $\Rightarrow$ 右子树
  - **后序 (postorder) 遍历**：子树序列 $\Rightarrow$根结点
- 后者包括**层序 (level order) 遍历**

---

[15]按顺序遍历每个子树，遍历方式亦为递归相同遍历方式，其余类同
[16]仅针对二叉树

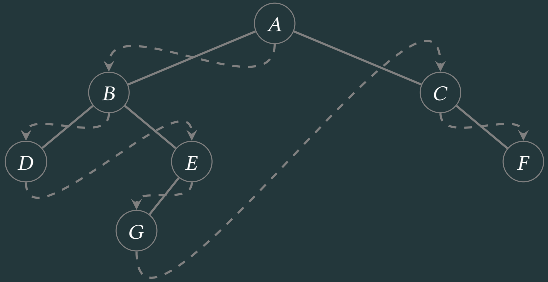


图 24: 二叉树的先序遍历示例：$A \longrightarrow B \longrightarrow D \longrightarrow E \longrightarrow G \longrightarrow C \longrightarrow F$

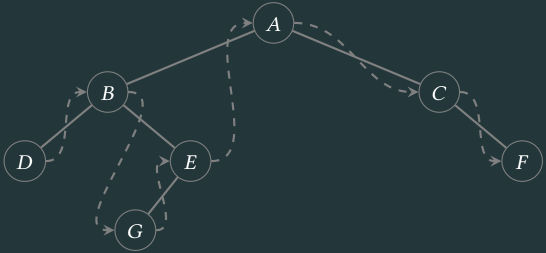


图 25: 二叉树的中序遍历示例：$D \longrightarrow B \longrightarrow G \longrightarrow E \longrightarrow A \longrightarrow C \longrightarrow F$

二叉树的遍历

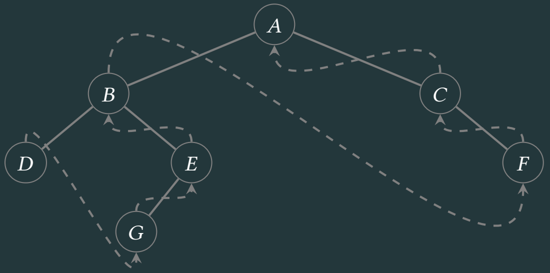


图 26: 二叉树的后序遍历示例： $D \longrightarrow G \longrightarrow E \longrightarrow B \longrightarrow F \longrightarrow C \longrightarrow A$

二叉树的遍历

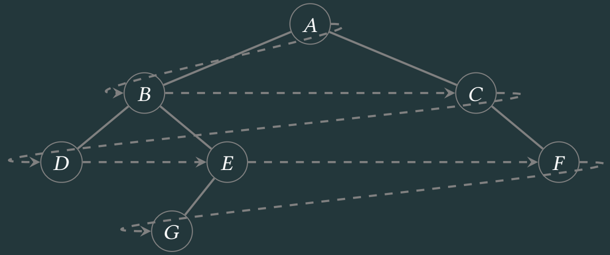


图 27: 二叉树的层序遍历示例：$A \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow E \longrightarrow F \longrightarrow G$

二叉树遍历性质甲

- 由先序遍历与中序遍历可推出后序遍历

二叉树遍历性质甲

- 由先序遍历与中序遍历可推出后序遍历

证明

1. 由先序遍历性质可找出根结点
2. 由中序遍历性质可找出左右子树
3. 对左右子树分别递归应用上述步骤直至无左右子树 □

先序：	根	左子树	右子树
中序：	左子树	根	右子树

图 28: 先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：							

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：							A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \rightarrow B \rightarrow D \rightarrow E \rightarrow G \rightarrow C \rightarrow F$
- 已知中序：$D \rightarrow B \rightarrow G \rightarrow E \rightarrow A \rightarrow C \rightarrow F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：				B			A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \rightarrow B \rightarrow D \rightarrow E \rightarrow G \rightarrow C \rightarrow F$
- 已知中序：$D \rightarrow B \rightarrow G \rightarrow E \rightarrow A \rightarrow C \rightarrow F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：	D			B			A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：	D		E	B			A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：	D	G	E	B			A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：	D	G	E	B		C	A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树的遍历

例

- 已知先序：$A \to B \to D \to E \to G \to C \to F$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求后序

先序：	A	B	D	E	G	C	F
中序：	D	B	G	E	A	C	F
后序：	D	G	E	B	F	C	A

图 29: 示例过程：先序 + 中序 $\Rightarrow$ 后序

二叉树遍历性质乙

- 由后序遍历与中序遍历可推出先序遍历

二叉树的遍历

二叉树遍历性质乙

- 由后序遍历与中序遍历可推出先序遍历

证明

1. 与性质甲类似，略

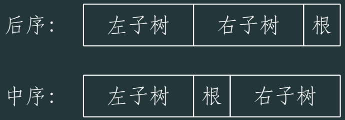


图 30: 后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：							

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A						

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A	B					

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

二叉树的遍历

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A	B	D				

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序:	D	G	E	B	F	C	A
中序:	D	B	G	E	A	C	F
先序:	A	B	D	E			

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \rightarrow G \rightarrow E \rightarrow B \rightarrow F \rightarrow C \rightarrow A$
- 已知中序：$D \rightarrow B \rightarrow G \rightarrow E \rightarrow A \rightarrow C \rightarrow F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A	B	D	E	G		

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A	B	D	E	G	C	

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

二叉树的遍历

例

- 已知后序：$D \to G \to E \to B \to F \to C \to A$
- 已知中序：$D \to B \to G \to E \to A \to C \to F$
- 求先序

后序：	D	G	E	B	F	C	A
中序：	D	B	G	E	A	C	F
先序：	A	B	D	E	G	C	F

图 31: 示例过程：后序 + 中序 $\Rightarrow$ 先序

二叉树的遍历

二叉树遍历性质丙

- 由先序遍历与后序遍历不可推出中序遍历

二叉树遍历性质丙

- 由先序遍历与后序遍历不可推出中序遍历

证明

1. 当根结点度为 1 时无法区分左右子树

先序： | 根 | 左子树 ？ 右子树 |

后序： | 左子树 ？ 右子树 | 根 |

图 32: 先序 + 后序 $\nRightarrow$ 中序

二叉树的遍历

二叉树深度优先遍历的递归实现[17]

```
void traverse_preorder(
        BinaryTreeNode *p,
        Visit v) {
    if (!p) {
        return; // 递归出口
    }
    v(p->data);
    traverse_preorder(p->left, v);
    traverse_preorder(p->right, v);
}
```

```
void traverse_inorder(
        BinaryTreeNode *p,
        Visit v) {
    if (!p) {
        return; // 递归出口
    }
    traverse_inorder(p->left, v);
    v(p->data);
    traverse_inorder(p->right, v);
}
```

```
void traverse_postorder(
        BinaryTreeNode *p,
        Visit v) {
    if (!p) {
        return; // 递归出口
    }
    traverse_postorder(p->left, v);
    traverse_postorder(p->right, v);
    v(p->data);
}
```

[17]须事先定义： `typedef void (*Visit)(DataType);`

二叉树广度优先遍历的非递归实现

- 用队列对每层结点按顺序缓存
- 根结点首先入队
- 当结点出队时均将其子树按顺序入队
- 当队列空时结束

```
void traverse_binary_tree_level_order(
        BinaryTreeNode *p, Visit v) {
    LinkedQueue *buffer = create_linked_queue();
    if (!buffer) { return; }
    push_linked_queue(buffer, p);
    while (!empty_linked_queue(buffer)) {
        pop_linked_queue(buffer, (DataType *)(&p));
        v(p->data);
        if (p->left) {
            push_linked_queue(buffer, p->left);
        }
        if (p->right) {
            push_linked_queue(buffer, p->right);
        }
    }
    destroy_linked_queue(buffer);
}
```

应用：表达式树

- 考虑仅包括二元运算的表达式
- 可将运算符作为根结点，左右操作数分别为左右子结点
- 操作数为叶结点，运算符为非叶结点
- 如此可将表达式转换为二叉树
- 前/中/后缀表达式分别对应二叉树的先/中/后序遍历

例：表达式树

- 前缀表达：`+ - A / B C * D E`
- 中缀表达：`A - B / C + D * E`
- 后缀表达：`A B C / - D E * +`

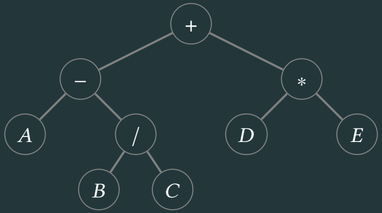


图 33: 表达式树示例

哈夫曼 (Huffman) 编码

- 需求：信息传输中的压缩编码
  - 频率高的信息采用短编码
  - 频率低的信息采用长编码
- 目标：用尽可能少的数据表示尽可能多的信息
- 应用：语音、图像、视频等流媒体数据的压缩

二叉树的简单应用：Huffman 编码

表 2: 例：两种颜色二进制编码的总数据量比较

颜色	定长	Huffman	出现概率
A	000	00	18%
B	001	11	32%
C	010	010	10%
D	011	100	14%
E	100	101	16%
F	101	0110	4%
G	110	0111	6%
总数据量[18]	$3n$	$2.6n$	

图 34: 实例图片

[18] n 为像素点个数

问题转化与建模

- 因二进制编码，故二叉树表示
- 左、右子树路径分别表示 0 与 1
- 编码值与叶结点一一对应
- 叶结点记录编码值对应权重
- 其他结点表示其子结点权重和
- 根至叶路径表示该叶对应编码

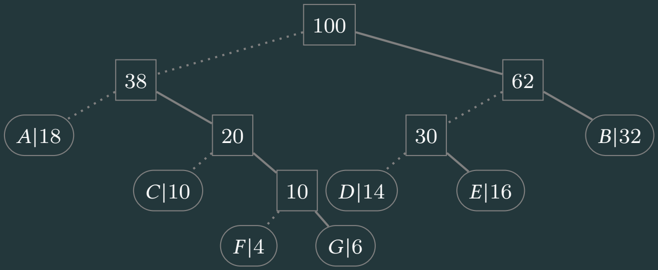


图 35: Huffman 编码树示例[19]

[19]路径中虚线表示 0，实线表示 1

Huffman 编码的目标

- 最小化所有叶结点深度的加权线性组合，即

$$\min \sum_{k=0}^{n-1} \omega_k d_k$$

其中 ω_k 与 d_k 分别为第 k 个叶结点的权重与 深度

- 满足上述要求的最优二叉树被称为相应信息的 Huffman 编码树

Huffman 编码树的构建过程

1. **初始化**：由给定的权重集合 $\{\omega_k\}_{k=0}^{n-1}$ 构建由 n 棵根树组成的森林$\mathscr{F}$
2. **合并**：在 $\mathscr{F}$ 中选取根权重最小的两棵二叉树分别作为左右子树构建新二叉树
3. **替换**：以新二叉树替换两棵旧二叉树
4. **重复**：重复步骤 2、3 直至 $\mathscr{F}$ 中只有一棵二叉树即为 Huffman 编码树

(a) A **(b)** B **(c)** C **(d)** D **(e)** E **(f)** F **(g)** G

图 36: Huffman 编码树构建过程：初始化

二叉树的简单应用：Huffman 编码

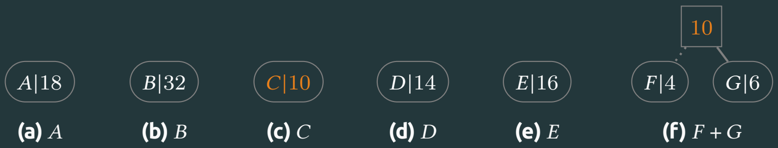


图 37: Huffman 编码树构建过程：合并与替换

二叉树的简单应用：Huffman 编码

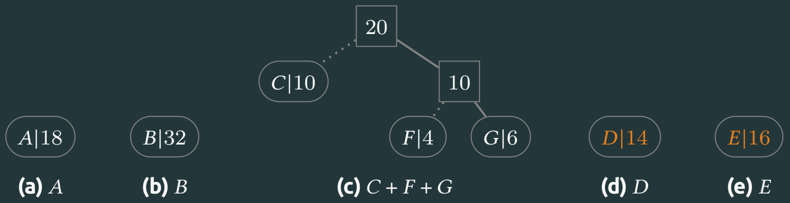


图 38: Huffman 编码树构建过程：合并与替换

二叉树的简单应用：Huffman 编码

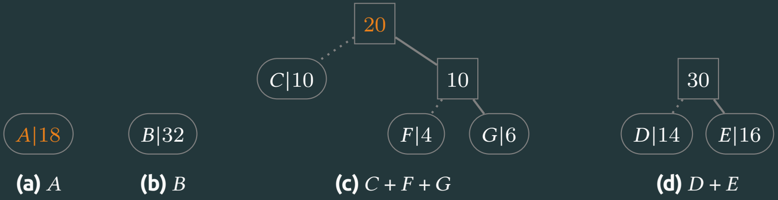


图 39: Huffman 编码树构建过程：合并与替换

二叉树的简单应用：Huffman 编码

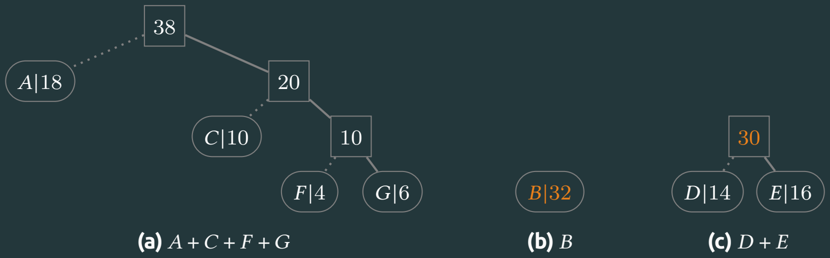


图 40: Huffman 编码树构建过程：合并与替换

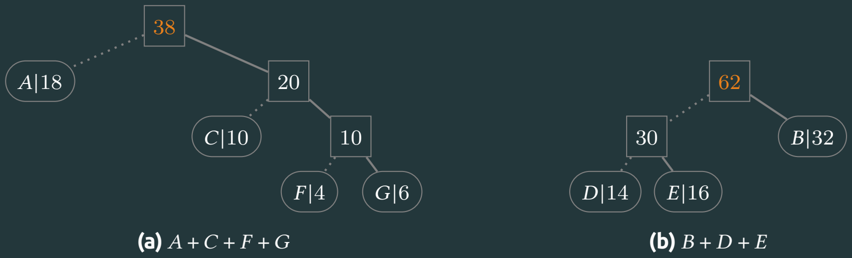
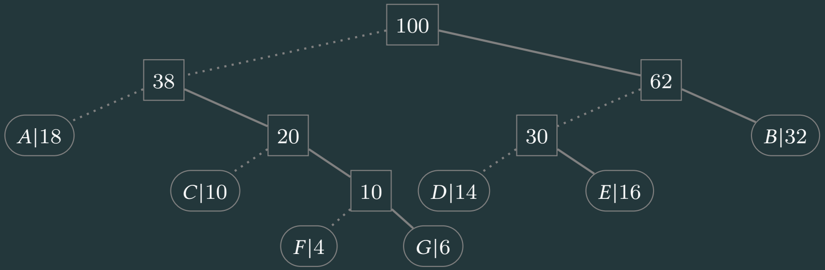


图 41: Huffman 编码树构建过程：合并与替换

二叉树的简单应用：Huffman 编码



(a) $A+B+C+D+E+F+G$

图 42: Huffman 编码树构建过程：合并与替换

Huffman 编码树的特点

- 无度为 1 的结点
  - 所有非叶结点均为两个子结点合并而成，故度为 2
- 结果不唯一
  - 可额外规定合并时左右子结点权重顺序，如左小于右

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```c
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0		18	-1	-1	-1	A
1		32	-1	-1	-1	B
2		10	-1	-1	-1	C
3		14	-1	-1	-1	D
4		16	-1	-1	-1	E
5	i_1	4	-1	-1	-1	F
6	i_2	6	-1	-1	-1	G
7		100	-1	-1	-1	
8		100	-1	-1	-1	
9		100	-1	-1	-1	
10		100	-1	-1	-1	
11		100	-1	-1	-1	
12		100	-1	-1	-1	

图 44: Huffman 编码树的构建过程

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0		18	−1	−1	−1	A
1		32	−1	−1	−1	B
2	i_1	10	−1	−1	−1	C
3		14	−1	−1	−1	D
4		16	−1	−1	−1	E
5		4	−1	−1	7	F
6		6	−1	−1	7	G
7	i_2	10	5	6	−1	
8		100	−1	−1	−1	
9		100	−1	−1	−1	
10		100	−1	−1	−1	
11		100	−1	−1	−1	
12		100	−1	−1	−1	

图 44: Huffman 编码树的构建过程

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```c
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0		18	-1	-1	-1	A
1		32	-1	-1	-1	B
2		10	-1	-1	8	C
3	i_1	14	-1	-1	-1	D
4	i_2	16	-1	-1	-1	E
5		4	-1	-1	7	F
6		6	-1	-1	7	G
7		10	5	6	8	
8		20	2	7	-1	
9		100	-1	-1	-1	
10		100	-1	-1	-1	
11		100	-1	-1	-1	
12		100	-1	-1	-1	

图 44: Huffman 编码树的构建过程

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```c
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0	i_1	18	−1	−1	−1	A
1		32	−1	−1	−1	B
2		10	−1	−1	8	C
3		14	−1	−1	9	D
4		16	−1	−1	9	E
5		4	−1	−1	7	F
6		6	−1	−1	7	G
7		10	5	6	8	
8	i_2	20	2	7	−1	
9		30	3	4	−1	
10		100	−1	−1	−1	
11		100	−1	−1	−1	
12		100	−1	−1	−1	

图 44: Huffman 编码树的构建过程

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```c
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0		18	-1	-1	10	A
1	i_2	32	-1	-1	-1	B
2		10	-1	-1	8	C
3		14	-1	-1	9	D
4		16	-1	-1	9	E
5		4	-1	-1	7	F
6		6	-1	-1	7	G
7		10	5	6	8	
8		20	2	7	10	
9	i_1	30	3	4	-1	
10		38	0	8	-1	
11		100	-1	-1	-1	
12		100	-1	-1	-1	

图 44: Huffman 编码树的构建过程

二叉树的简单应用：Huffman 编码

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0		18	−1	−1	10	A
1		32	−1	−1	11	B
2		10	−1	−1	8	C
3		14	−1	−1	9	D
4		16	−1	−1	9	E
5		4	−1	−1	7	F
6		6	−1	−1	7	G
7		10	5	6	8	
8		20	2	7	10	
9		30	3	4	11	
10	i_1	38	0	8	−1	
11	i_2	62	9	1	−1	
12		100	−1	−1	−1	

图 44: Huffman 编码树的构建过程

Huffman 编码树的顺序实现

- 结点包括权重、左右子结点与父结点信息
- 可采用顺序表或数组存储结点序列

weight	left	right	parent

图 43: Huffman 编码树的结点结构

```c
typedef struct {
    int weight; // 结点权重
    int left; // 左子结点序号
    int right; // 右子结点序号
    int parent; // 父结点序号
} HuffmanNode;
```

0	18	−1	−1	10	A
1	32	−1	−1	11	B
2	10	−1	−1	8	C
3	14	−1	−1	9	D
4	16	−1	−1	9	E
5	4	−1	−1	7	F
6	6	−1	−1	7	G
7	10	5	6	8	
8	20	2	7	10	
9	30	3	4	11	
10	38	0	8	12	
11	62	9	1	12	
12	100	10	11	−1	

图 44: Huffman 编码树的构建过程

小结

小结

- 半线性结构以树与森林为代表，前驱为线性、后继为非线性
- 树与森林的问题均可转化为二叉树，使研究更加方便、统一
- 二叉树可采用顺序与链式两种方式存储
- 二叉树的遍历可将半线性结构转化为线性结构
- Huffman 编码树可用于数据压缩

问与答