数据结构 03

2025 年 3 月 10 日

考纲内容：

- **树和二叉树：** 树的基本概念，二叉树的概念、性质和存储结构，二叉树的遍历，线索二叉树，哈夫曼树。
- **图：** 图的基本概念与存储结构，图的遍历算法，最小生成树（Prim 和 Kruskal 算法），最短路径（Dijkstra 算法）。

1 树和二叉树

1.1 树的基本概念

1.1.1 定义

树（Tree）是 n $(n \geq 0)$ 个结点的有限集。当 $n = 0$ 时称为空树。在任意一棵非空树中，有且仅有一个特定的称为根（Root）的结点，剩余结点可分为 m $(m \geq 0)$ 个互不相交的有限集，每个集合本身又是一棵树，称为原树的子树（Subtree）。

1.1.2 术语

- **结点的度**：结点拥有的子树个数
- **树的度**：树内各结点的度的最大值
- **叶子结点**：度为 0 的结点
- **分支结点**：度不为 0 的结点
- **结点的层次**：从根开始定义，根为第 1 层，根的孩子为第 2 层
- **树的高度/深度**：树中结点的最大层次
- **森林**：m $(m \geq 0)$ 棵互不相交的树的集合

1.1.3 树的表示方法

1. **双亲表示法**：使用数组存储，每个结点中附设一个指向其双亲的指针
2. **孩子表示法**：每个结点有多个指针指向其孩子结点
3. **孩子兄弟表示法**：每个结点设两个指针，分别指向第一个孩子和下一个兄弟

1.2 二叉树的概念与性质

1.2.1 定义

二叉树（Binary Tree）是一种树形结构，每个结点最多有两个子树，分别称为左子树和右子树。二叉树是有序树，其左右子树不能互换。

1.2.2 特殊二叉树

- **满二叉树**：所有分支结点都有左子树和右子树，且所有叶子结点都在同一层
- **完全二叉树**：除最后一层外，每一层的结点数都达到最大值，且最后一层的结点从左到右连续排列
- **平衡二叉树**：任意结点的左右子树高度差不超过 1

1.2.3 二叉树的性质

1. 第 i 层的结点数最多为 2^{i-1} $(i \geq 1)$
2. 高度为 h 的二叉树最多有 $2^h - 1$ 个结点
3. 对于任意一棵二叉树，如果叶子结点的个数为 n_0，度为 2 的结点个数为 n_2，则有 $n_0 = n_2 + 1$

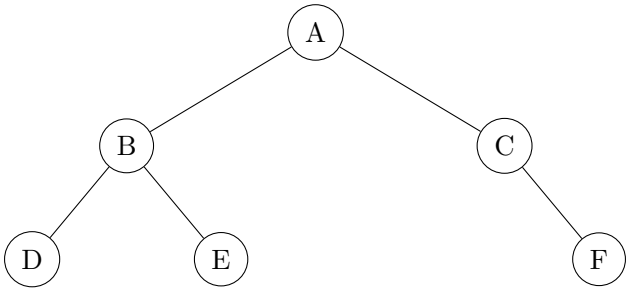


图 1: 二叉树示例

1.2.4 二叉树的存储结构

顺序存储　适用于完全二叉树，使用数组存储，按层序编号。

- 若某结点的下标为 i，则其左孩子下标为 $2i$，右孩子下标为 $2i+1$
- 若某结点的下标为 i，则其双亲结点下标为 $\lfloor i/2 \rfloor$

链式存储　使用二叉链表，每个结点包含数据域和左右指针域。

```
typedef struct BiNode {
    ElemType data;          // 数据域
    struct BiNode *lchild;  // 左孩子指针
```

```
    struct BiNode *rchild;    // 右孩子指针
} BiNode, *BiTree;
```

1.3 二叉树的遍历

1.3.1 先序遍历

先序遍历（Preorder Traversal）的访问顺序：根结点 → 左子树 → 右子树

Algorithm 1 先序遍历算法

```
1: procedure PreOrder(T)
2:     if T ≠ NULL then
3:         Visit(T)                          ▷ 访问根结点
4:         PreOrder(T.lchild)                ▷ 先序遍历左子树
5:         PreOrder(T.rchild)                ▷ 先序遍历右子树
6:     end if
7: end procedure
```

1.3.2 中序遍历

中序遍历（Inorder Traversal）的访问顺序：左子树 → 根结点 → 右子树

Algorithm 2 中序遍历算法

```
1: procedure InOrder(T)
2:     if T ≠ NULL then
3:         InOrder(T.lchild)                 ▷ 中序遍历左子树
4:         Visit(T)                          ▷ 访问根结点
5:         InOrder(T.rchild)                 ▷ 中序遍历右子树
6:     end if
7: end procedure
```

1.3.3 后序遍历

后序遍历（Postorder Traversal）的访问顺序：左子树 → 右子树 → 根结点

1.3.4 层序遍历

层序遍历（Level Order Traversal）的访问顺序：从上到下、从左到右，逐层访问树中的结点。

Algorithm 3 后序遍历算法

```
1: procedure PostOrder(T)
2:     if T ≠ NULL then
3:         PostOrder(T.lchild)                ▷ 后序遍历左子树
4:         PostOrder(T.rchild)                ▷ 后序遍历右子树
5:         Visit(T)                           ▷ 访问根结点
6:     end if
7: end procedure
```

Algorithm 4 层序遍历算法

```
1: procedure LevelOrder(T)
2:     初始化队列 Q
3:     if T ≠ NULL then
4:         将根结点 T 入队
5:         while 队列 Q 非空 do
6:             出队一个结点 p
7:             Visit(p)
8:             if p.lchild ≠ NULL then
9:                 将 p.lchild 入队
10:            end if
11:            if p.rchild ≠ NULL then
12:                将 p.rchild 入队
13:            end if
14:        end while
15:    end if
16: end procedure
```

1.4 线索二叉树

1.4.1 概念

线索二叉树（Threaded Binary Tree）是一种利用二叉树中空指针域的存储结构，通过这些空指针域存放指向前驱或后继的指针（称为线索）。

1.4.2 类型

- 前序线索二叉树：按照前序遍历建立线索
- 中序线索二叉树：按照中序遍历建立线索
- 后序线索二叉树：按照后序遍历建立线索

1.4.3 存储结构

```c
typedef struct ThreadNode {
    ElemType data;
    struct ThreadNode *lchild, *rchild;
    int ltag, rtag;  // 标志位，0表示孩子，1表示线索
} ThreadNode, *ThreadTree;
```

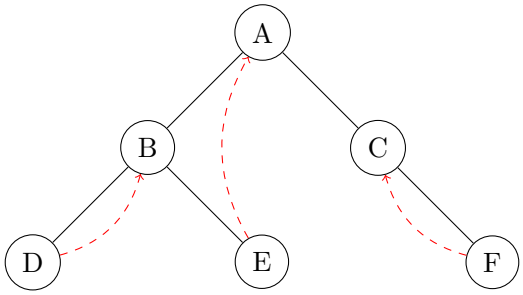


图 2: 中序线索二叉树示例（红色虚线表示线索）

1.5 哈夫曼树

1.5.1 概念

哈夫曼树（Huffman Tree），也称为最优二叉树，是一种带权路径长度最小的二叉树。其中，树的带权路径长度为从树根到每个叶子结点的路径长度乘以该结点权值之和。

1.5.2 构造算法

Example 1. 已知权值集合 $\{7, 5, 2, 4\}$，构造哈夫曼树。

带权路径长度 $WPL = 7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 7 + 10 + 6 + 12 = 35$

Algorithm 5 哈夫曼树构造算法

```
1: procedure HuffmanTree(w[1...n])                    ▷ w[1...n] 为 n 个权值
2:     初始化：将 n 个权值分别作为 n 棵仅含一个结点的二叉树
3:     for i = 1 to n − 1 do
4:         在森林中选取两棵根结点权值最小的树，分别记为 T1 和 T2
5:         构造一棵新的二叉树 T，根结点权值为 T1 和 T2 的权值之和
6:         分别以 T1 和 T2 作为新二叉树 T 的左、右子树
7:         从森林中删除 T1 和 T2，将新树 T 加入森林
8:     end for
9:     return 森林中唯一的树（即哈夫曼树）
10: end procedure
```

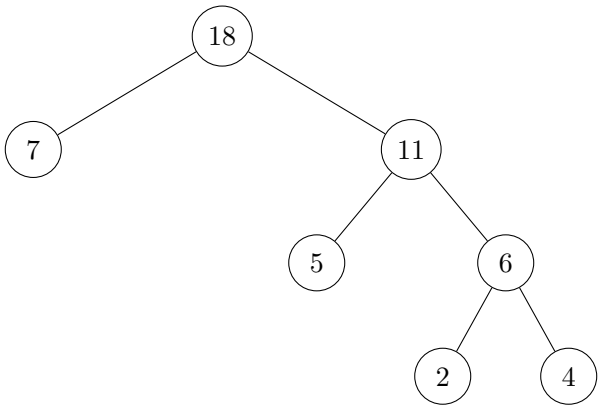


图 3: 构造的哈夫曼树

1.5.3 哈夫曼编码

哈夫曼编码是一种基于哈夫曼树的前缀编码，常用于数据压缩。其特点是：

- 频率高的字符编码短，频率低的字符编码长
- 没有任何一个字符的编码是另一个字符编码的前缀

Example 2. 对于字符集 *{A, B, C, D}* 和对应频率 *{7, 5, 2, 4}*，基于上面构造的哈夫曼树，可得编码：

- *A*：*0*（频率最高，编码最短）
- *B*：*10*
- *C*：*110*
- *D*：*111*

1.6 二叉排序树

二叉排序树（Binary Search Tree, BST）是一种特殊的二叉树，具有以下性质：

- 左子树上所有结点的值均小于根结点的值
- 右子树上所有结点的值均大于根结点的值
- 左右子树也分别是二叉排序树

1.6.1 基本操作

- **查找**：从根结点开始，若目标值小于当前结点则向左子树查找，大于则向右子树查找
- **插入**：类似查找过程，找到应插入的位置后创建新结点
- **删除**：分三种情况：删除叶结点、删除只有一个子树的结点、删除有两个子树的结点

1.7 平衡二叉树

平衡二叉树（Balanced Binary Tree）是一种特殊的二叉排序树，其任意结点的左右子树高度差不超过 1。

1.7.1 平衡因子

结点的平衡因子（Balance Factor）定义为左子树高度减去右子树高度。平衡二叉树中所有结点的平衡因子只能是-1、0 或 1。

1.7.2 AVL 树的旋转操作

不平衡情况分为四种：

- **LL 型**：在结点的左孩子的左子树上插入导致不平衡，需右旋
- **RR 型**：在结点的右孩子的右子树上插入导致不平衡，需左旋
- **LR 型**：在结点的左孩子的右子树上插入导致不平衡，需先左旋后右旋
- **RL 型**：在结点的右孩子的左子树上插入导致不平衡，需先右旋后左旋

2 图

2.1 图的基本概念

2.1.1 定义

图（Graph）是由顶点集合 V 和边集合 E 组成的二元组 G = (V, E)，其中每条边连接一对顶点。

2.1.2 基本术语

- **有向图**：边有方向，称为弧（Arc）。弧的起点称为弧尾，终点称为弧头
- **无向图**：边无方向，用无序对 (v, w) 表示
- **完全图**：任意两个顶点之间都有边相连的图
  - 有 n 个顶点的无向完全图有 $\frac{n(n-1)}{2}$ 条边
  - 有 n 个顶点的有向完全图有 $n(n-1)$ 条弧
- **权**：边或弧上的数值，表示从一个顶点到另一个顶点的距离或代价
- **子图**：原图中部分顶点及相连边构成的图

2.1.3 顶点与边的关系

- **顶点的度**：与该顶点相连的边的条数
  - 在有向图中分为入度（入边数）和出度（出边数）
  - 顶点的度 = 入度 + 出度
- **路径**：从一个顶点到另一个顶点经过的顶点序列
- **路径长度**：路径上边或弧的数目或权值之和
- **回路/环**：起点和终点相同的路径
- **简单路径**：除起点和终点可以相同外，其余顶点均不相同的路径
- **连通**：两个顶点之间存在路径

2.1.4 特殊图

- **连通图**：任意两个顶点都是连通的无向图
- **连通分量**：无向图中的极大连通子图
- **强连通图**：任意两个顶点都互相可达的有向图
- **强连通分量**：有向图中的极大强连通子图
- **生成树**：包含图中全部顶点的一个极小连通子图
- **生成森林**：非连通图的各个连通分量的生成树的集合

无向图　　有向图

图 4: 无向图和有向图示例

2.2 图的存储结构

2.2.1 邻接矩阵

邻接矩阵（Adjacency Matrix）是一种使用二维数组存储图的方法，用于表示顶点之间的相邻关系。

- 无向图：$A[i][j] = A[j][i]$（矩阵对称），值为边的权值或 1（表示有边）
- 有向图：$A[i][j]$ 表示从顶点 i 到顶点 j 的弧
- 稠密图适合用邻接矩阵表示

```
#define MaxVertexNum 100  // 最大顶点数
typedef struct {
    char Vex[MaxVertexNum];            // 顶点表
    int Edge[MaxVertexNum][MaxVertexNum];  // 邻接矩阵
    int vexnum, arcnum;                // 顶点数和弧数
} MGraph;
```

2.2.2 邻接表

邻接表（Adjacency List）是一种链式存储结构，每个顶点对应一个单链表，链表中的结点表示该顶点的所有邻接顶点。

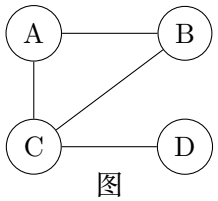
- 适合存储稀疏图
- 容易找到给定顶点的所有邻接点
- 判断两个顶点是否相邻需要搜索

```
typedef struct ArcNode {      // 边表结点
    int adjvex;               // 该弧所指向的顶点的位置
    struct ArcNode *next;     // 指向下一条弧的指针
    // InfoType info;         // 网的权值
```

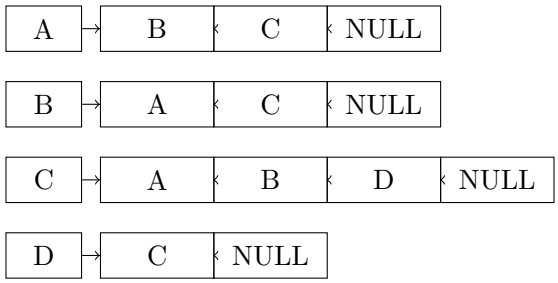
```
} ArcNode;

typedef struct VNode {          // 顶点表结点
    VertexType data;            // 顶点信息
    ArcNode *first;             // 指向第一条依附该顶点的弧
} VNode, AdjList[MaxVertexNum];

typedef struct {
    AdjList vertices;           // 邻接表
    int vexnum, arcnum;         // 顶点数和弧数
} ALGraph;
```



图



邻接表

图 5: 邻接表示例

2.3 图的遍历算法

2.3.1 广度优先搜索 (BFS)

广度优先搜索（Breadth-First Search）类似于树的层序遍历，按照距离起始顶点由近到远的顺序访问图中的顶点。

BFS 性质

- 时间复杂度：邻接矩阵为 $O(|V|^2)$，邻接表为 $O(|V|+|E|)$
- 空间复杂度：$O(|V|)$
- 能够求解无权图的单源最短路径问题

Algorithm 6 广度优先搜索算法

```
1: procedure BFS(G, start)
2:     初始化访问标记数组 visited[G.vexnum] = {FALSE}
3:     初始化队列 Q
4:     访问起始顶点 start
5:     visited[start] = TRUE
6:     将 start 入队列 Q
7:     while 队列 Q 非空 do
8:         出队一个顶点 v
9:         for 顶点 v 的每个邻接点 w do
10:            if visited[w] == FALSE then
11:                访问顶点 w
12:                visited[w] = TRUE
13:                将 w 入队列 Q
14:            end if
15:        end for
16:    end while
17: end procedure
```

2.3.2 深度优先搜索 (DFS)

深度优先搜索（Depth-First Search）类似于树的先序遍历，沿着一条路径一直走到底，直到不能再继续为止，然后回溯选择其他路径。

Algorithm 7 深度优先搜索算法

```
1: procedure DFS(G, v)
2:     访问顶点 v
3:     visited[v] = TRUE
4:     for 顶点 v 的每个邻接点 w do
5:         if visited[w] == FALSE then
6:             DFS(G, w)
7:         end if
8:     end for
9: end procedure
```

DFS 性质

- 时间复杂度：邻接矩阵为 $O(|V|^2)$，邻接表为 $O(|V|+|E|)$
- 空间复杂度：最坏情况 $O(|V|)$（递归栈深度）

2.4 最小生成树

最小生成树（Minimum Spanning Tree，MST）是一副连通加权无向图中一棵权值最小的生成树。

2.4.1 Prim 算法

Prim 算法从单一顶点开始，逐步将新顶点纳入到生成树中，每次选择代价最小的边。

Algorithm 8 Prim 算法

```
1: procedure PRIM(G, start)
2:     初始化集合 U = {start}，表示已加入生成树的顶点集合
3:     初始化集合 V − U，表示尚未加入生成树的顶点集合
4:     while U ≠ V do
5:         在所有连接 U 和 V − U 的边中，找到权值最小的边 (u, v)，其中 u ∈ U，v ∈ V − U
6:         将顶点 v 加入集合 U
7:         将边 (u, v) 加入最小生成树
8:     end while
9: end procedure
```

Prim 算法性质

- 时间复杂度：$O(|V|^2)$（邻接矩阵），可优化到 $O(|E|\log|V|)$（优先队列）
- 适合于稠密图

2.4.2 Kruskal 算法

Kruskal 算法按照边的权值从小到大依次选择边，保证不形成环。

Algorithm 9 Kruskal 算法

```
1: procedure KRUSKAL(G)
2:     初始化最小生成树 T 为空
3:     将图 G 中的边按权值从小到大排序
4:     for 图 G 中每条边 (u, v)（按权值从小到大） do
5:         if 加入边 (u, v) 不会在 T 中形成环 then
6:             将边 (u, v) 加入 T
7:         end if
8:     end for
9: end procedure
```

Kruskal 算法性质

- 时间复杂度：$O(|E|\log|E|)$

- 适合于稀疏图

图 6: 最小生成树示例

2.5 最短路径

2.5.1 Dijkstra 算法

Dijkstra 算法用于计算一个顶点到其他所有顶点的最短路径，适用于所有边权值为非负数的图。

Algorithm 10 Dijkstra 算法

```
1: procedure DIJKSTRA(G, start)
2:     初始化距离数组 dist[i] = ∞，dist[start] = 0
3:     初始化前驱数组 path[i] = −1
4:     初始化集合 S = ∅，表示已确定最短路径的顶点集合
5:     while S 不包含所有顶点 do
6:         在 V − S 中选择距离最小的顶点 u
7:         将 u 加入集合 S
8:         for 顶点 u 的每个邻接点 v do
9:             if dist[u] + weight(u, v) < dist[v] then
10:                dist[v] = dist[u] + weight(u, v)
11:                path[v] = u
12:            end if
13:        end for
14:    end while
15:    return dist, path
16: end procedure
```

Dijkstra 算法性质

- 时间复杂度：$O(|V|^2)$（邻接矩阵），可优化到 $O(|E|\log|V|)$（优先队列）
- 不适用于有负权边的图

图 7: Dijkstra 最短路径示例

2.5.2 Floyd 算法

Floyd 算法是一种动态规划算法，用于寻找加权图中所有顶点之间的最短路径。

Algorithm 11 Floyd 算法

```
procedure FLOYD(G)
    初始化距离矩阵 D[i][j] = G.Edge[i][j]
    初始化路径矩阵 Path[i][j] = j（若 i 到 j 有直接路径）
    for k = 0 to n − 1 do                                   ▷ 以 k 为中转点
        for i = 0 to n − 1 do
            for j = 0 to n − 1 do
                if D[i][k] + D[k][j] < D[i][j] then
                    D[i][j] = D[i][k] + D[k][j]
                    Path[i][j] = Path[i][k]
                end if
            end for
        end for
    end for
end procedure
```

Floyd 算法性质

- 时间复杂度：$O(|V|^3)$
- 空间复杂度：$O(|V|^2)$
- 可处理有负权边的图，但不能有负权回路