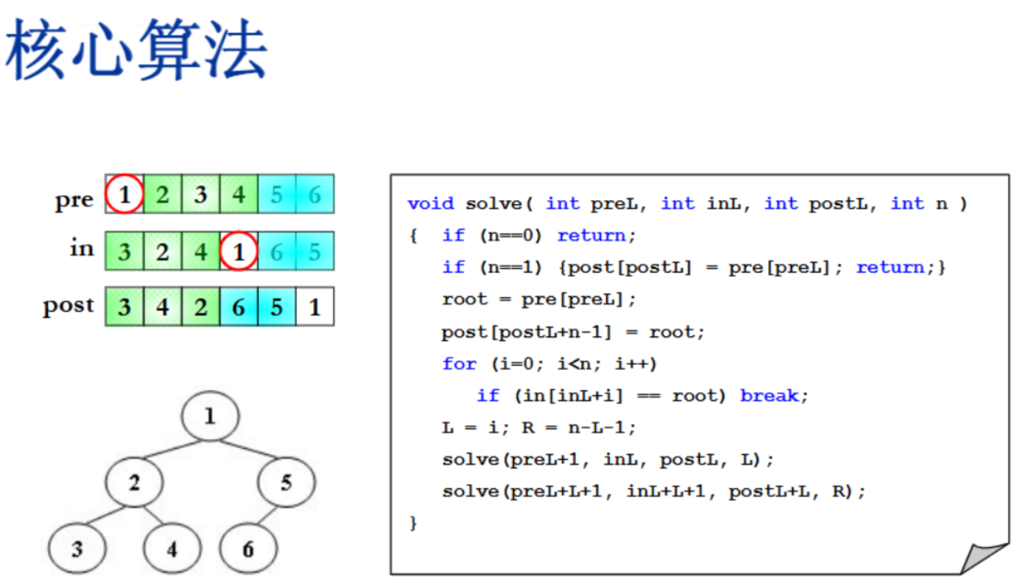
20 第七讲 图（中）

笔记本:	浙江大学《数据结构》		
创建时间:	2025/4/20 16:06	**更新时间:**	2025/4/26 18:54
作者:	panhengye@163.com		
URL:	https://www.doubao.com/chat/3508869117410818		

如果已知一棵二叉树的先序和中序遍历，怎么求后续遍历的访问顺序?

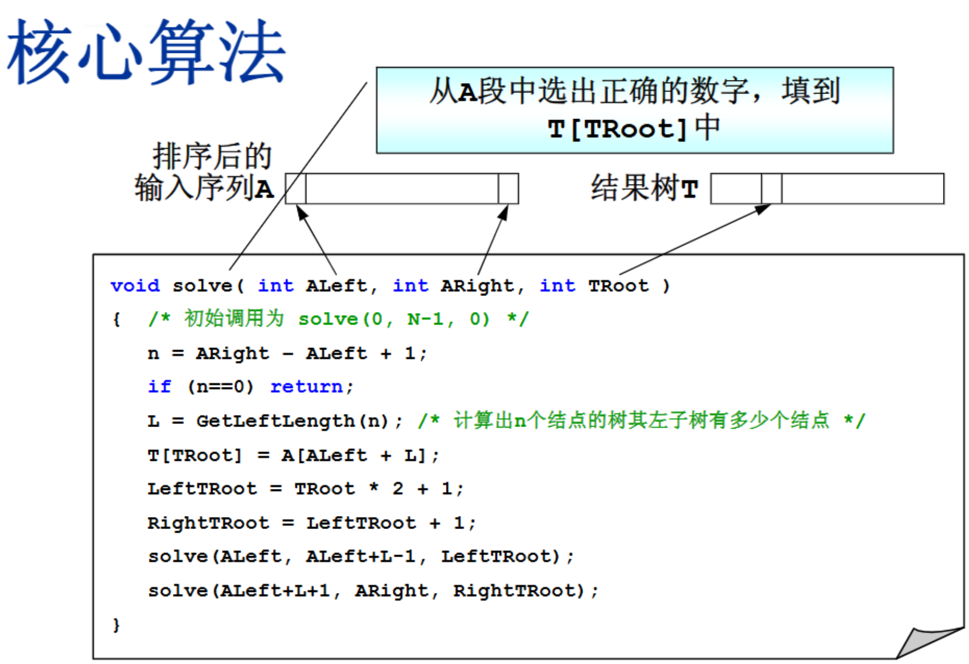


思路讲解:

- 初始调用
  - 假设最初调用 solve(0, 0, 0, 6) ，这里 preL = 0 （前序遍历起始位置 ），inL = 0 （中序遍历起始位置 ），postL = 0 （后序遍历起始位置 ），n = 6 （节点个数 ）。
- 第一步：进本情况判断
  - if (n==0) return; ：因为 n = 6 ，不满足条件，继续执行。
  - if (n==1) {post[postL] = pre[preL]; return;} ：因为 n = 6 ，不满足条件，继续执行。
- 第二步：确定根结点并赋值给后续遍历数组
  - root = pre[preL]; ，此时 preL = 0 ，pre[0] = 1 ，所以 root = 1 。
  - post[postL+n - 1] = root; ，即 post[0 + 6 - 1] = post[5] = 1 。
- 第三步：在中序遍历中找到根结点位置
  - for (i = 0; i < n; i++) 循环，用于在中序遍历数组 in 中找根节点 root （值为 1 ）的位置。
  - 当 i = 3 时，in[0 + 3] == 1 ，满足 if (in[inL + i] == root) ，执行 break ，此时 L = 3 ，R = 6 - 3 - 1 = 2 。
- 第四步：递归构建左子树
  - 调用 solve(preL + 1, inL, postL, L) ，即 solve(0 + 1, 0, 0, 3) ：
  - 新的 preL = 1 ，inL = 0 ，postL = 0 ，n = 3 。
  - 重复上述步骤
    - 基本情况判断，n = 3 ，不满足 n == 0 和 n == 1 。

- 确定根节点，root = pre[1] = 2 ，post[0 + 3 - 1] = post[2] = 2 。
    - 在中序遍历找根节点位置，循环找到 in[0 + 1] == 2 ，L = 1 ，R = 3 - 1 - 1 = 1 。
    - 递归构建左子树，调用 solve(1 + 1, 0, 0, 1) ，此时 n = 1 ，post[0] = pre[2] = 3 ，返回。
    - 递归构建右子树，调用 solve(1 + 1 + 1, 0 + 1 + 1, 0 + 1, 1) ，此时 n = 1 ，post[1] = pre[3] = 4 ，返回。
- **第五步：递归构建右子树**
  - 调用 solve(preL + L + 1, inL + L + 1, postL + L, R) ，即 solve(0 + 3 + 1, 0 + 3 + 1, 0 + 3, 2) ：
    - 新的 preL = 4 ，inL = 4 ，postL = 3 ，n = 2 。
    - 基本情况判断，n = 2 ，不满足 n == 0 和 n == 1 。
    - 确定根节点，root = pre[4] = 5 ，post[3 + 2 - 1] = post[4] = 5 。
    - 在中序遍历找根节点位置，循环找到 in[4 + 0] == 5 ，L = 0 ，R = 2 - 0 - 1 = 1 。
    - 递归构建左子树，调用 solve(4 + 1, 4, 3, 0) ，n = 0 ，直接返回。
    - 递归构建右子树，调用 solve(4 + 0 + 1, 4 + 0 + 1, 3 + 0, 1) ，此时 n = 1 ，post[4] = pre[5] = 6 ，返回。

题目：给定一个非负的连续数组，返回完全二叉搜索树的层序遍历结果



在这个核心算法中有两个问题仍待解决：

- 如何对给定数组进行排序
- 如何计算出N个结点完全二叉树的左子树有多少结点
  - H（层数） = $\log_2(N+1)$ ①
  - $2^H - 1 + X = N$ ②
  - $L = 2^{H-1} - 1 + Y$ ④
    - $Y = \min(X, 2^{H-1})$ ③
  - 将①②③的计算结果带入④

Huffman code

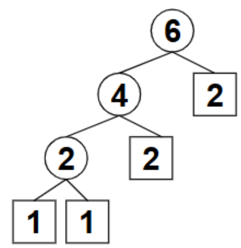
- WPL(weighted path length)最小
- 无歧义解码 - 前缀码：数据仅存放于叶子结点上

核心算法

1. 计算最优编码长度

```
MinHeap H = CreateHeap( N ); /* 创建一个空的、容量为N的最小堆 */
H = ReadData( N ); /* 将f[]读入H->Data[]中 */
HuffmanTree T = Huffman( H ); /* 建立Huffman树 */
int CodeLen = WPL( T, 0 );
```

```
int WPL( HuffmanTree T, int Depth )
{   if ( !T->Left && !T->Right )
        return (Depth*T->Weight);
    else /* 否则T一定有2个孩子 */
        return (WPL(T->Left, Depth+1)
              + WPL(T->Right, Depth+1));
}
```



注意建立了一个最小堆，这是为什么呢?

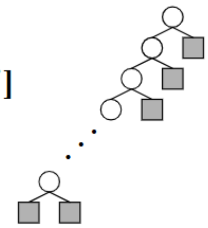
- **方便获取最小权值节点**：构建哈夫曼树的过程中，需要不断地从节点集合中选取权值最小的两个节点来合并。最小堆这种数据结构能够在\(O(log n)\)的时间复杂度内快速找到最小值，其中n是堆中元素的个数。
- **动态维护节点集合**：在哈夫曼树的构建过程中，随着节点的不断合并，节点集合是动态变化的。最小堆能够方便地支持插入和删除操作，并且在操作后能够快速调整堆的结构，以保持堆的性质。

核心算法

2. 对每位学生的提交，检查

a) 长度是否正确 $Len = \sum_{i=0}^{N-1} strlen(code[i]) \times f[i]$

注意：Code[i]的最大长度为 N-1



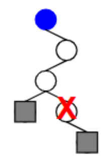
b) 建树的过程中检查是否满足前缀码要求

Code[i] = "1011"

Code[i] = "100"

Code[i] = "10~~0~~1"

Code[i] = "101"



最短路径问题

定义

- 在网络中，求两个不同顶点之间的所有路径中，边的权值之和最小的那一条路径
  - 最短路径 shortest path
  - 第一个顶点为源点 Source
  - 最后一个点为终点 Destination

问题分类

- 单源最短路径问题：从某个固定源点出发，求其所到其他顶点的最短路径
  - 无权图
  - 有权图
- 多源最短路径问题：求任意两个顶点间的最短路径

无权图的的单源最短路算法——按照递增（非递减）的顺序找出各个顶点的最短路

相当于经过顶点(vertex)最少的路

无权图的单源最短路算法

```
void BFS ( Vertex S )
{ visited[S] = true;
  Enqueue(S, Q);
  while(!IsEmpty(Q)){
    V = Dequeue(Q);
    for ( V 的每个邻接点 W )
      if ( !visited[W] ) {
        visited[W] = true;
        Enqueue(W, Q);
      }
  }
}
```

```
void Unweighted ( Vertex S )
{ Enqueue(S, Q);
  while(!IsEmpty(Q)){
    V = Dequeue(Q);
    for ( V 的每个邻接点 W )
      if ( dist[W]==-1  ) {
        dist[W] = dist[V]+1;
        path[W] = V;
        Enqueue(W, Q);
      }
  }
}
```

$T = O(|V| + |E|)$

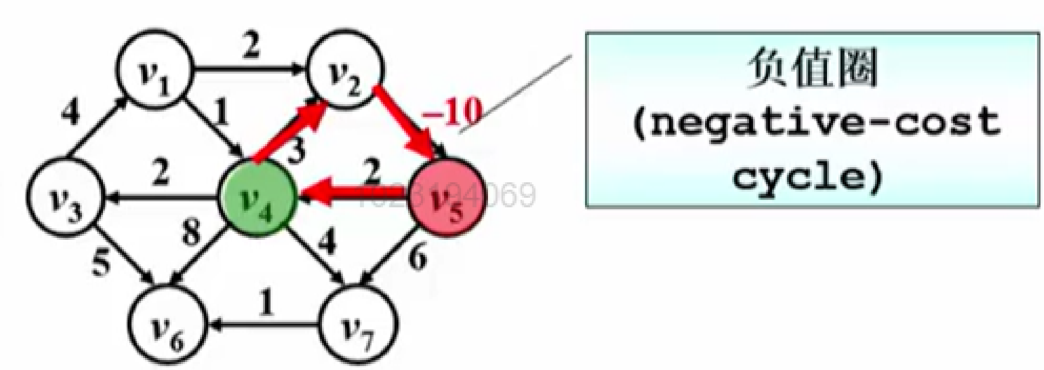
dist[W] = S到W的最短距离

dist[S] = 0

path[W] = S到W的路上经过的某顶点

有权图的单源最短路径

顶点数不再重要，更重要的是权重的和



注意：如果存在负值圈（negative-cost cycle），那么算法会失效，因此一般不讨论这种情况。

引入一个新词：Dijkstra算法，相关故事见：[艾兹格·迪科斯彻（Edsger Wybe Dijkstra）](#)

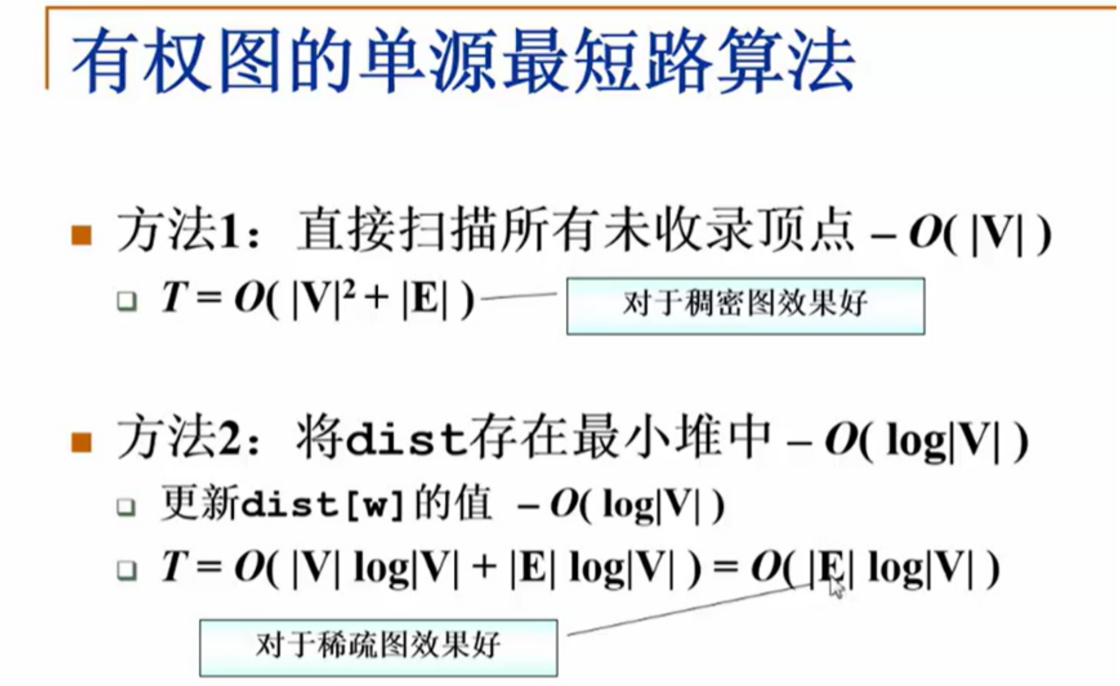
- **Dijkstra 算法**
  - 令S={源点s ＋ 已经确定了最短路径的顶点v_i}
  - 对任一未收录的顶点v，定义dist[v]为s到v的最短路径长度，但该路径仅经过S中的顶点。即路径{$s \rightarrow (v_i \in S) \rightarrow v$}的最小长度
  - 若路径是按照递增（非递减）的顺序生成的，则
    - 真正的最短路必须只经过S中的顶点（为什么？）
    - 每次从未收录的顶点中选一个dist最小的收录（贪心）
    - 增加一个v进入S，可能影响另外一个w的dist值！
      - dist[w] = min{dist[w], dist[v] + <v,w>的权重}

有权图的单源最短路算法

```
void Dijkstra( Vertex s )
{ while (1) {
    V = 未收录顶点中dist最小者;
    if ( 这样的V不存在 )
      break;
    collected[V] = true;
    for ( V 的每个邻接点 W )
      if ( collected[W] == false )
       if ( dist[V]+E<V,W> < dist[W] ) {
          dist[W] = dist[V] + E<V,W> ;
          path[W] = V;
       }
  }
} /* 不能解决有负边的情况 */
```

$T = O(\,?\,)$

那么Dijkstra算法的时间复杂度如何呢?



- 如果是稠密图：直接扫描所有未收录顶点比较好
- 如果是稀疏图：将dist存在最小堆中比较好，实现方法：[Python中内置最小堆的实现方式]

多源最短路算法

方法一：直接将单源最短路算法调用V遍 —— 对稀疏图好

方法二： Floyd算法