

数据结构复习整理

Base

数据结构二元组

$$DS = (D, S)$$

D: 数据元素的有限集

S: D上关系的有限集

抽象数据类型

$$ADT = (D, S, P)$$

D: 数据对象

S: D上的关系集合

P: 对D的基本操作集

算法的五个特性

有穷性、确定性、可行性、输入、输出

算法设计的要求

正确性、可读性、健壮性、效率与低存储量需求

复杂度

时间复杂度 -> 频度: 语句的重复执行次数

e.g. `for(int i=0; i<n; ++i)` 的频度是n

空间复杂度 -> 需要的辅助空间

线性表

按存储顺序分为: **顺序表** (数组)、**链表**

排序后为：有序表



插入、删除元素都涉及到元素移动 -> 元素移动次数的期望

在数组（线性表）a中插入值x：一般指 $a[i-1] = x$, i.e. 第i个元素为x

链表原地逆置

- 有头节点

```
LinkedList ReverseList(LinkedList *L) {
    LNode *p, *r;
    p = L->next;
    L->next = NULL; // 断头
    while(p!=NULL) {
        r = p->next; // 暂存
        p->next = L->next;
        L->next = p;
        p = r;
    }
    return L;
    // return p;    ==> ❌
    // 也要返回一个带有头节点的
}
```

- 无头节点

```
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr;
    ListNode *curr = head;
    ListNode *next;
    while(curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

↑抄的代码，不如直接手动变成有头链表，注意返回不要有头节点即可，链表首部为传入的head即可

 画图的时候，next为NULL时在图里画“^”表示为空

队列

队列元素结构 & 队列结构

```
typedef struct QNode{
    ElemType    data;
    struct QNode *next;
} QNode, *QueuePtr;

typedef struct{
    QueuePtr  front;
    QueuePtr  rear;
} LinkQueue;
```

队列中的元素一般会用QueuePtr数组来存，或者是链队列（元素用链表）

注意front和rear与队列判空、判满的方式。

1. 如果是数组QueuePtr[m]来存，front=-1和rear=m都表明队列已满
2. 如果是链队列来存，在不看随时可以动态扩张的情况下，front和rear分别是首尾地址


！ 假溢出：

顺序队列中存在“假溢出”现象。因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为假溢出。

用数组存的情况帮助理解：

```
//    队列已满
front == -1 && rear == m
//    队列未满
front > -1 && rear == m    //    假溢出
front == -1 && rear < m    //    假溢出
front > -1 && rear < m
```

类似的，链队列的情况front和rear都是地址（队列内连续），若只有一个超出预设地址范围都是假溢出


 解决办法：循环队列（首尾相连，做成环状）

链表长度计算为

$$(Q.rear - Q.front + QMAXSIZE) \% QMAXSIZE$$

串

注意串的长度

 串长示例：

"12345" -> 5

"" -> 0


" " -> 1

程序中使用到的串：串变量，串常量

存储方式

定长顺序存储

将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。

 注意：使用String[0]存储串长

堆分配存储方式

仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。

块链存储方式：

是一种链式存储结构表示。

串的存储密度

$$\text{串的存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

数组和广义表

数组

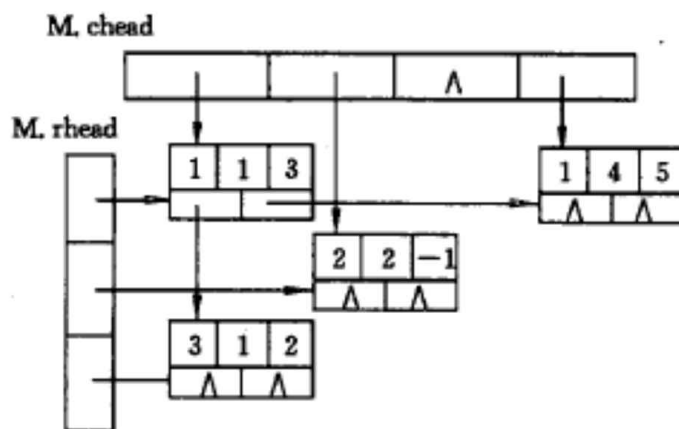
1. 高维数组元素首地址计算【列优先 or 行优先】
2. 矩阵的压缩存储
 - a. 对称阵
 - b. 三角阵
 - c. 对角阵
3. 稀疏矩阵
 - a. 用三元组 (i, j, a_{ij}) 唯一确定一个非零元素
 - b. 存储为三元组顺序表【行逻辑链接】
 - c. 稀疏矩阵
4. **稀疏矩阵：转置、乘法运算算法**

转置：先找到原矩阵三元组列表中每个j（列）在转置后的矩阵的三元组列表中的首个位置（idx）

注意：书里给的稀疏矩阵中的三元组的data[0]未用

5. 十字链表

$$M = \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$



行、列分别有头指针顺序表，一个行号或者列号都分别对应一个稀疏矩阵的三元组顺序表

稀疏度

在 $m \times n$ 的矩阵中, 有 t 个元素不为 0, 则其稀疏度 δ 计算方式为

$$\delta = \frac{t}{m \times n}$$

广义表

定义

一般记作 $LS = (\alpha_1, \alpha_2, \dots, \alpha_n)$ ，非空时有：

表头 $Head(LS) = \alpha_1$, 表尾 $Tail(LS) = (\alpha_2, \alpha_3, \dots, \alpha_n)$

💡 表头是广义表的首个元素，表尾是广义表去掉表头以后剩下表项构成的广义表【所以另外还要加括号】，按照定义写题

表长 & 表深

表5-2 广义表及其示例

广 义 表	表长n	表深h
A=()	0	0
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞

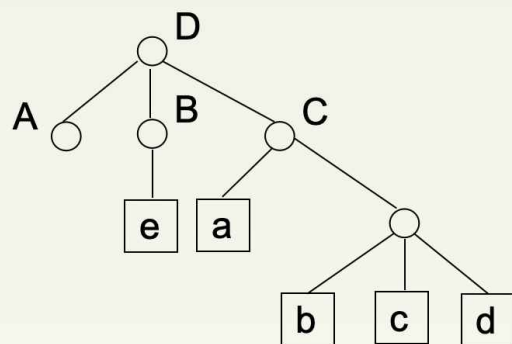


图5-12 广义表的图形表示

树 & 二叉树

注：二叉树范畴下，节点的度指的是其子树的数目

二叉树性质

在顺序存储二叉树时，节点在存储空间里的排序与层序遍历相同，空的位置要填 \emptyset

二叉树一般性质

1. 二叉树的第 i 层上之多有 2^{i-1} 个节点 ($i \geq 1$).
2. 深度为 k 的二叉树至多有 $2^k - 1$ 个节点 ($k \geq 1$).
3. \forall 二叉树 T ，记其终端节点（叶子节点，i.e. 度为0）数为 n_0 ，度为 2 度节点数为 n_2 ，则 $n_0 = n_2 + 1$
4. 具有 n 个节点的完全二叉树的深度为 $\lceil \log_2 n \rceil + 1$
5. 对二叉树进行层序编号，根节点 $i = 1$ ；另外，如果 $i \mid 2$ ，则其双亲是节点 $\lceil \frac{i}{2} \rceil$

满二叉树

深度为 k 且有 $2^k - 1$ 个节点的二叉树

完全二叉树

深度为 k 且有 n 个节点的二叉树，每个节点与满二叉树中编号为 $1, 2, \dots, n$ 的节点一一对应性质有：

1. 若完全二叉树的深度为 k ，则所有的叶子结点都出现在第 k 层或 $k - 1$ 层
2. 对于任一结点，如果其右子树的最大层次为 l ，则其左子树的最大层次为 l 或 $l + 1$
3. 完全二叉树是满二叉树的一部分，而满二叉树是完全二叉树的特例

遍历二叉树 & 搜索二叉树

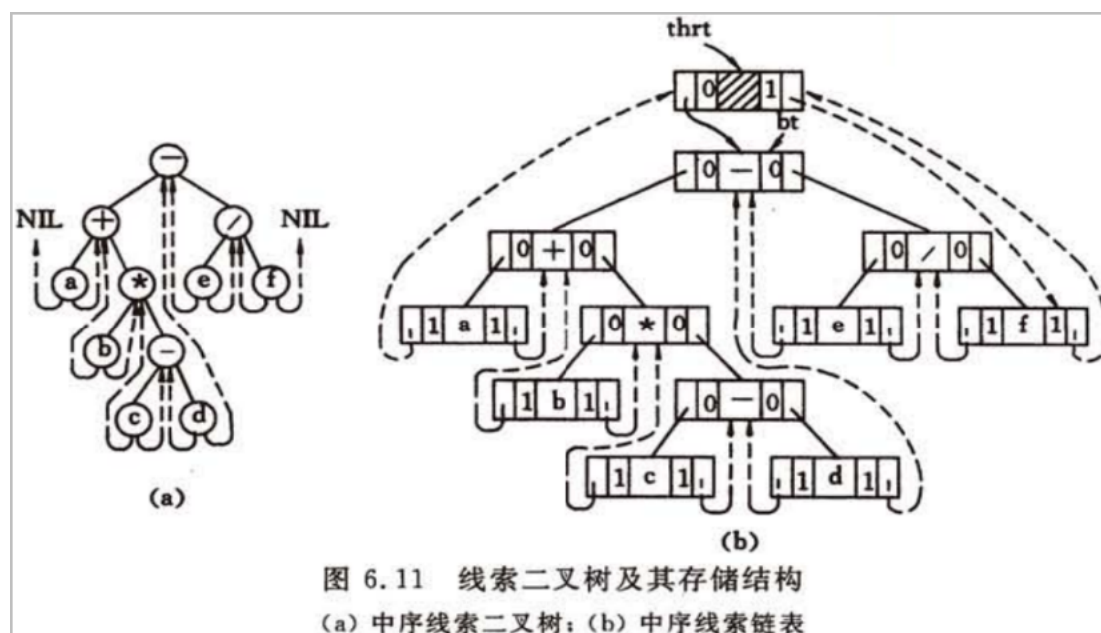
二叉树的遍历

一般约定左孩子在右孩子前面遍历，有：

1. DLR——先(根)序遍历
2. LDR——中(根)序遍历
3. LRD——后(根)序遍历
4. 层次遍历

！ 有时间的话看看非递归算法【比如考试那天早上】

线索二叉树 & 线索链表



对图中度不等于2度点，分别画出他们的前驱、后继，即为“线索”

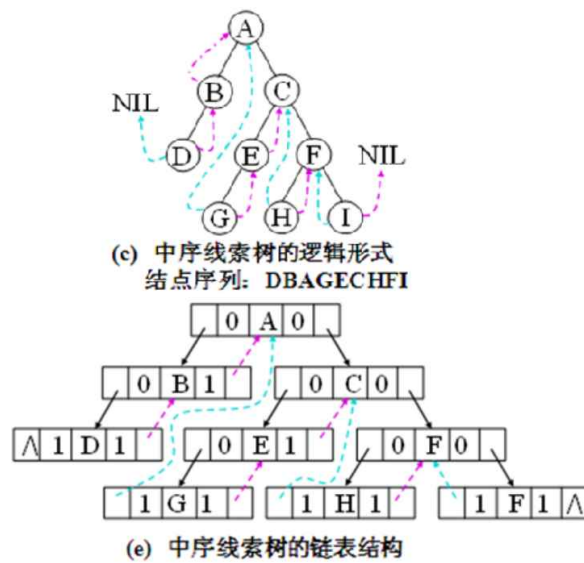
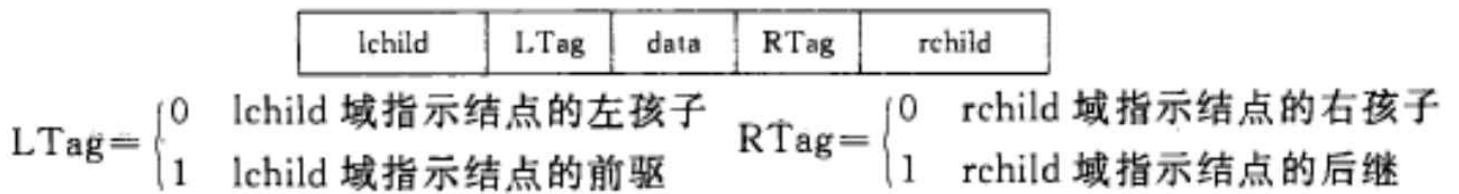


图6-11 线索二叉树及其存储结构

这里出现了一个链表结构



Tag的取值表示对应的child是指向线索（遍历信息）还是孩子（结构信息）

0 -> 孩子, 1 -> 线索（前驱或后继）

两个节点之间，前驱、后继只用画一个

树 & 森林

森林的存储

双亲表示法

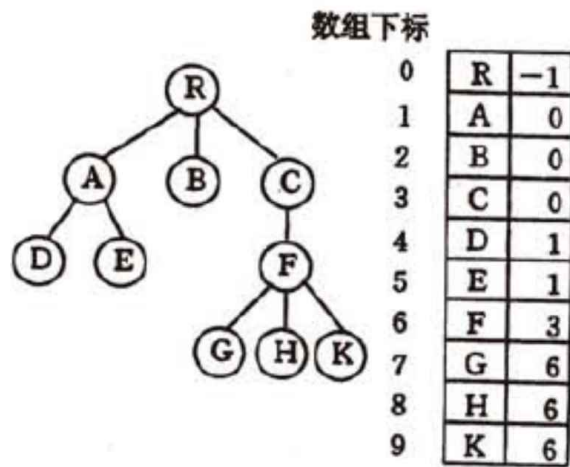
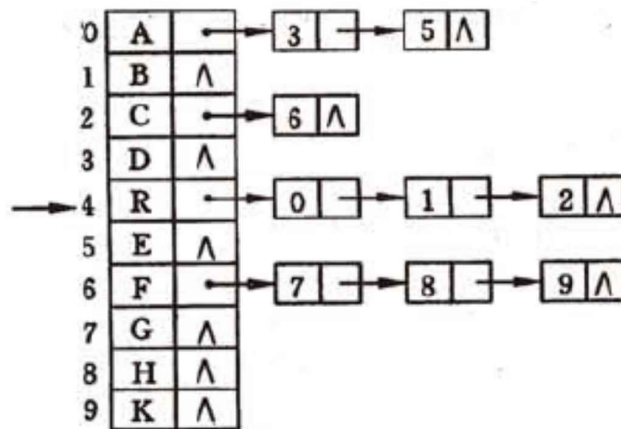


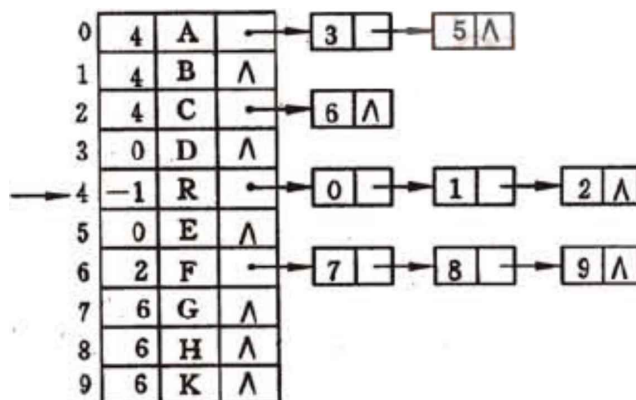
图 6.13 树的双亲表示法示例

R处对应的-1表示其为根节点（没有父节点）

孩子表示法



双亲-孩子表示



这些图好像不会在森林这边考，是在后面无向图那里会考。

森林对应的二叉树

森林中是一棵棵树，将树合并时，每一棵树的根节点分别都是前一棵树根节点的孩子。

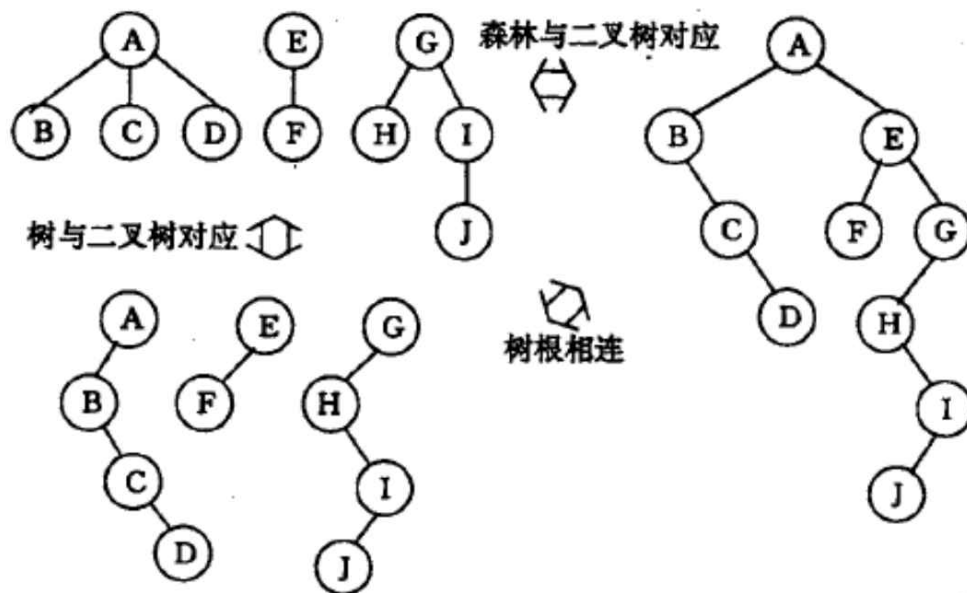


图 6.17 森林与二叉树的对应关系示例

A-E-G分别是0层节点，B-F-H分别是1层节点，...

📌 原来的一棵树上的同级节点，这些节点每个分别是自己的前一个节点的后一跳【如A-E-G，B-C-D】

原来的一棵树上亲子关系，到森林中变成左孩子关系（这里用先序对应的是左孩子）

树的遍历

1. 树的先序遍历实质上与将树转换成二叉树后对二叉树的先序遍历相同。
2. 树的后序遍历实质上与将树转换成二叉树后对二叉树的中序遍历相同。

哈夫曼编码

在哈夫曼树（最优二叉树）中，左0右1，其他没啥

图

概念整理

完全无向图： $\frac{n(n-1)}{2}$ 条无向边

完全有向图： $n(n-1)$ 条有向边

稀疏图：弧的数量 $e < n \log n$ ，否则为稠密图

权 & 网：边上带权的图通常称为网

子图：顶点集、边集均为子集

生成子图：顶点集相同，边集为子集

生成树是极小连通子图

连通图、连通分量、强连通图、强连通分量、连通子图、极大连通子图、生成树、生成森林

存储结构

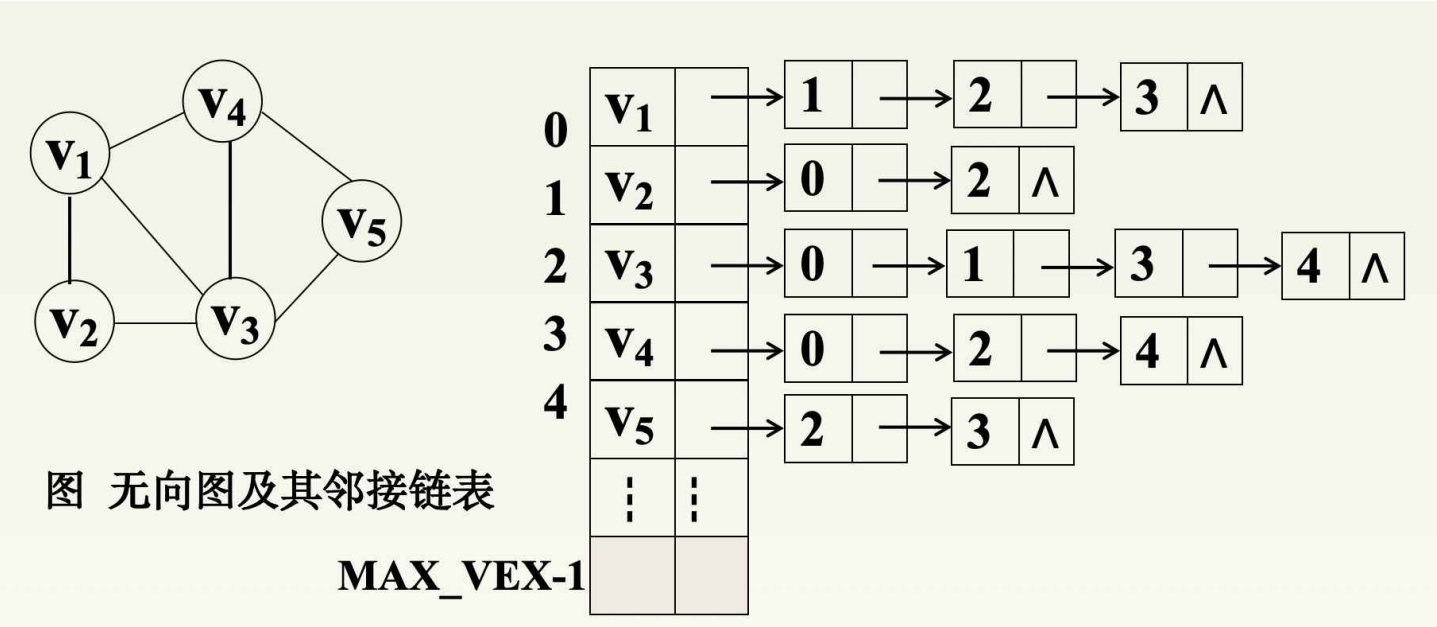
邻接矩阵

注意一般无向图、有向图、带权有向图、带权无向图的邻接矩阵区别

邻接表

注意邻接表中指向的内容都是节点在顺序表里的索引值

无向图邻接表



考试一般是这个？



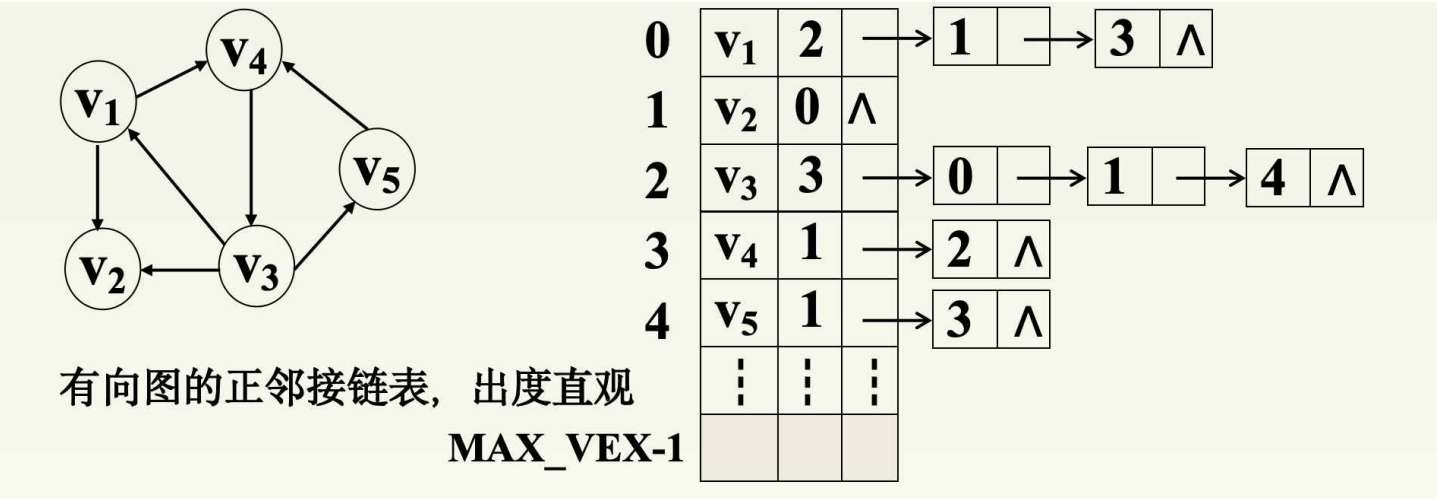
DFSTracer:

按照顺序表遍历（之间按照idx循环即可），对每一个元素做DFS，单独写一个DFS的递归方法。

DFS => 循环中经过的点（包括当前点）都标记已访问，深入时需要遍历链表（并递归链表项），对已经被标记访问过的点直接跳过即可。

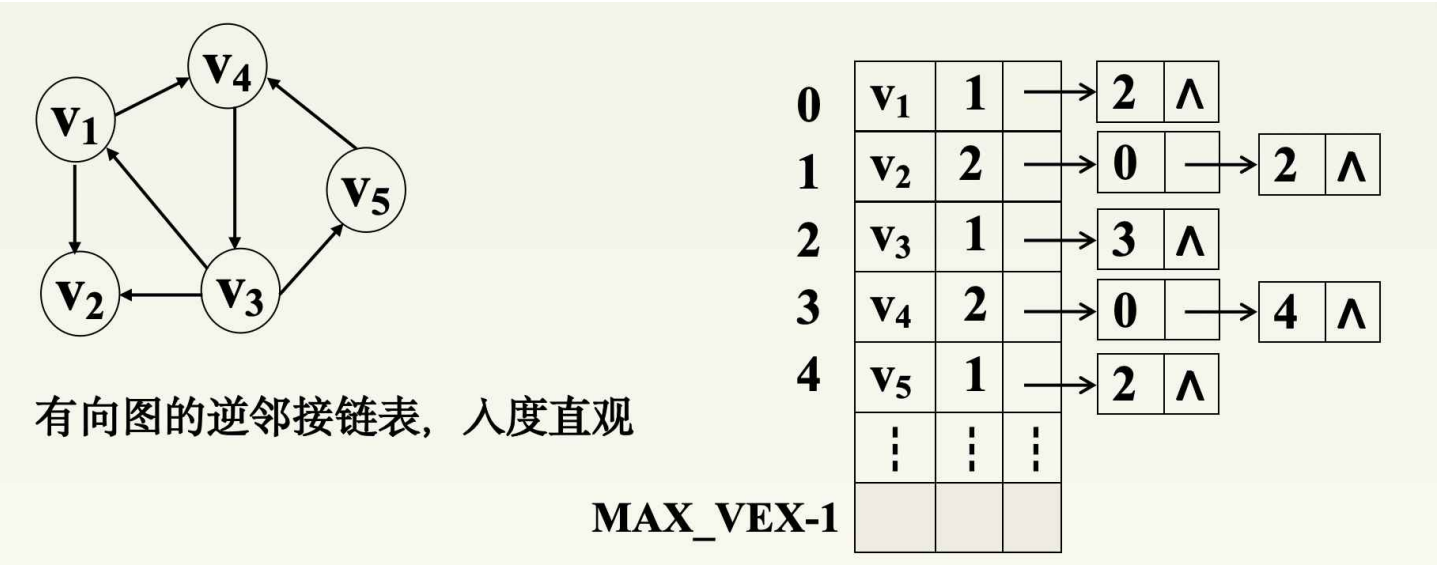
必须要用循环遍历无向邻接表的顺序表中的顺序表的原因：图不一定连通，比如是森林。

有向图的正邻接表



顺序表中的值标记了**出度**个数，方便AOV-网的计算【前面写了出度，后面链表 内就存储了所有出边】

有向图的逆邻接表



顺序表中的值标记了**入度**个数，方便AOV-网的计算

广度优先遍历（BFS）

做点题 -> 按照顺序表的顺序，访问此元素链接的一级；对该级下方不访问

画存储结构

顶点矩阵（向量+邻接矩阵）【邻接矩阵 \neq 存储结构】

连通性问题

无向图的连通分量

对于无向图，对其进行遍历时：

- a. ◆ 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
- b. ◆ 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

生成树 & 生成森林

生成森林：选择根，分配遍历顺序

生成树：DFS修改

最小生成树

在边带权的情况下才有“最小”一说

Prim算法

和Dijkstra算法类似思想，选择代价最小的边，若该边依附的顶点落在图的不同连通分量上，则选择此边【在能够连通两个连通分量的边集中选择】，否则舍去此边而选择下一条代价最小的边。

邻接矩阵时间复杂度 $O(V^2)$ ，邻接表时间复杂度 $O(E \log_2 V)$

Kruskal算法

先对边权排序，从小到大选择，若加入这条边没有形成环，则选入，否则舍弃继续看下一条边

时间复杂度 $E \log_2 E$



Prim是从一个点出发，比较和选择**已选点集**和**未选点集**之间的最小边权

Kruscal是从小到大选择边（进而选中点），依次接受不会让选中的点成环的边直至成树

拓扑排序

是一个在有向无环图上操作

概念 & 定义

偏序关系：自反的、反对称的、传递的【用 $x \leq y$ 表示 x 和 y 之间的偏序关系】

全序关系：设 R 是 X 上的偏序关系，若 $\forall x, y \in X$ 有 $xRy = yRx$ ，则 R 是 X 上的全序关系

若集合 X 上的关系 R 是自反的、反对称的和传递的，则称 R 是集合 X 上的偏序关系。

设 R 是集合 X 上的偏序(Partial Order)，如果对每个 $x, y \in X$ 必有 xRy 或 yRx ，则称 R 是集合 X 上的全序关系。

直观地看，偏序指集合中仅有部分成员之间可比较，而全序指集合中全体成员之间均可比较。例如，图 7.25 所示的两个有向图，图中弧 $\langle x, y \rangle$ 表示 $x \leq y$ ，则 (a) 表示偏序，

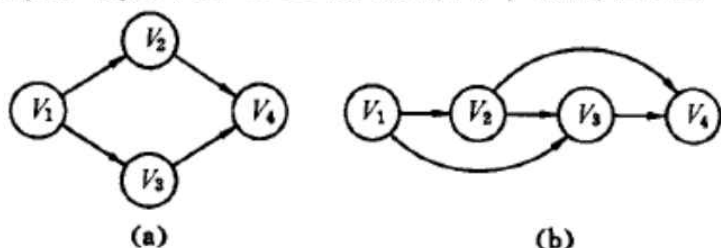


图 7.25 表示偏序和全序的有向图
(a) 表示偏序； (b) 表示全序

(b) 表示全序。若在 (a) 的有向图上人为地加一个表示 $v_2 \leq v_3$ 的弧(符号“ \leq ”表示 v_2 领先于 v_3)，则 (a) 表示的亦为全序，且这个全序称为拓扑有序(Topological Order)，而由偏序定义得到拓扑有序的操作便是拓扑排序。

AOV-网

Activity On Vertex Network, i.e. AOV-网

这个和os里资源分配图那里判断死锁的算法同样原理。

1. 在AOV-网中选择一个没有前驱的顶点且输出之；
2. 在AOV-网中删除该顶点以及从该顶点出发的所有有向弧(以该顶点为尾的弧)；
3. 重复1. 2.，直到图中全部顶点都已输出(即表示图中无环)，或图中不存在无前驱的顶点(即表示图中必有环)。

关键路径

与AOV-网相对应的是AOE-网, i.e. Activity On Edge Network, 是边表示活动的有向无环图

图中顶点表示事件(Event), 每个事件表示在其前的所有活动已经完成, 其后的活动可以开始; 弧表示活动, 弧上的权值表示相应活动所需的时间或费用。

算法目标: 优化工程花费的时间、费用等

$e(i)$: 表示活动 a_i 的最早开始时间;

$l(i)$: 在不影响进度的前提下, 表示活动 a_i 的最晚开始时间; 则 $l(i) - e(i)$ 表示活动 a_i 的时间余量, 若 $l(i) - e(i) = a_i$, 表示活动 a_i 是关键活动。

活动在 $[e(i), l(i)]$ 间开始, 总的完成时间不受影响

另外有事件最早发生时间和 $ve(j)$ 最迟发生时间 $vl(j)$ 。

先计算

$ve(0) = 0$, 向后递推;

算法:

$$ve(j) = \max_{i \leq j} \{ve(i) + dut(< i, j >)\} \quad (j = 1, 2, \dots, n-1, \text{ 对 } < j, i > \text{ 来说是出度})$$

【从 $ve(0) = 0$ 开始, 找到当前点后继最大可达 (Bellman-Ford思路)】

$vl(n-1) = ve(n-1)$, 向前反向递推

算法:

$$vl(i) = \min_{j \geq i} \{vl(j) - dut(< i, j >)\} \quad (i = n-1, 2, \dots, 0)$$

【从 $vl(n-1) = ve(n-1)$ 开始, 找到当前点的前驱的最小可达 (Bellman-Ford思路)】

 $e \sim ve \sim \text{"early"} \sim \text{Max} \sim (0, 1 \rightarrow n-1)$

$l \sim vl \sim \text{"late"} \sim \text{Min} \sim (n-1, n-2 \rightarrow 0)$

寻找关键活动

计算活动 a_i (是点 V_j 到 V_k 的有向带权边) 的工作时间范围

$$\begin{cases} e(i) = ve(j) \\ l(i) = vl(k) - dut(< j, k >) \end{cases}$$

所有 $e(i) = l(i)$ 的都是关键活动

最短路

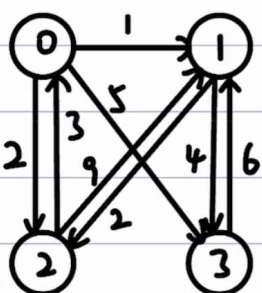
Dijkstra

时间复杂度 $O(n^2)$

Floyd

时间复杂度 $O(n^3)$

Floyd 回路 W_{ij} , $O(V^3)$



- 1° 每轮定第 k 行 k 列与主对角线.
- 2° 对应行列值相加与现有比较, 取小.

	0	1	2	3		0	1	2	3
$A^{(0)}$	0	1	2	5		0	1	2	5
	∞	0	2	4		∞	0	2	4
	3	9	0	∞		3	4	0	8
	∞	6	∞	0		∞	6	∞	0

$A^{(1)}$

算法思想

- ① 初始时令 $S=\{\}$, $A[i][j]$ 的赋初值方式是:

$$A[i][j] = \begin{cases} 0 & i=j \text{ 时} \\ W_{ij} & i \neq j \text{ 且 } \langle v_i, v_j \rangle \in E, \text{ } W_{ij} \text{ 为弧上的权值} \\ \infty & i \neq j \text{ 且 } \langle v_i, v_j \rangle \text{ 不属于 } E \end{cases}$$

- ② 将图中一个顶点 V_k 加入到 S 中, 修改 $A[i][j]$ 的值, 修改方法是:

$$A[i][j] = \min\{A[i][j], (A[i][k] + A[k][j])\}$$

原因: 从 V_i 只经过 S 中的顶点 (V_k) 到达 V_j 的路径长度可能比原来不经过 V_k 的路径更短。

- ③ 重复②, 直到 G 的所有顶点都加入到 S 中为止。

这里的 \min 若取了后一项才把 V_k 放进对应的最短路中

https://www.bilibili.com/video/BV1tQ4y147pw/?spm_id_from=333.337.search-card.all.click&vd_source=d20c6ae79ecec1...

【五分钟速通】巨快无比秒杀Floyd算法手动模拟流程_哔哩哔哩_bilibili

极速版Floyd!, 视频播放量 21508、弹幕量 25、点赞数 693、投硬币枚数 361、收藏人数 1325、转发人数 166, 视频作者 Prism菌, 作者简介 等到人已不再奔忙 等到心也不再轻狂 我们相约老地方, 相关视频:求最短路径Floyd算法!, 【五分钟速通】Dijkstra算...

查找

平均查找长度 $ASL = \sum_{i=1}^n P_i C_i$

概率 \times 长度

静态查找

顺序查找

注意一下监视哨

有序表-折半查找

即二分查找

索引表查找

即分块查找

1. 将查找表分成几块。块间有序，即第*i*+1块的所有记录关键字均大于(或小于)第*i*块记录关键字；块内无序
2. 在查找表的基础上附加一个索引表，索引表是按关键字有序的

动态查找

二叉排序树（BST）

定义


左子树所有节点值 \leq 根节点值 \leq 右子树所有节点值

节点插入

对插入节点 x

1. 若相等，不需要插入；

2. 若 $x.key < T \rightarrow key$: 结点 x 插入到 T 的左子树中;
3. 若 $x.key > T \rightarrow key$: 结点 x 插入到 T 的右子树中

 如果只是一般二叉排序树的插入, 就按照这个算法定义来插, 正是因为按照这个定义插入导致二叉树结构失衡, 才有了平衡二叉树

节点删除

① 若 p 是叶子结点: 直接删除 p , 如图9-5(b)所示。

② 若 p 只有一棵子树(左子树或右子树): 直接用 p 的左子树(或右子树)取代 p 的位置而成为 f 的一棵子树。即原来 p 是 f 的左子树, 则 p 的子树成为 f 的左子树; 原来 p 是 f 的右子树, 则 p 的子树成为 f 的右子树, 如图9-5(c)、(e)所示。

③ 若 p 既有左子树又有右子树: 处理方法有以下两种, 可以任选其中一种。

◆ 用 p 的直接前驱结点代替 p 。即从 p 的左子树中选择值最大的结点 s 放在 p 的位置(用结点 s 的内容替换结点 p 内容), 然后删除结点 s 。 s 是 p 的左子树中的最右边的结点且没有右子树, 对 s 的删除同②, 如图9-5(e)所示。

◆ 用 p 的直接后继结点代替 p 。即从 p 的右子树中选择值最小的结点 s 放在 p 的位置(用结点 s 的内容替换结点 p 内容), 然后删除结点 s 。 s 是 p 的右子树中的最左边的结点且没有左子树, 对 s 的删除同②, 如图9-5(f)所示。

平衡二叉树 (AVL)

定义


平衡二叉树或者是空树, 或者是满足下列性质的二叉树:

1. 左子树和右子树深度之差的绝对值不大于1;
2. 左子树和右子树也都是平衡二叉树。

平衡因子(Balance Factor): 二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子

平衡因子只取-1, 0, 1

我们研究 平衡二叉排序树(Balanced Binary Sort Tree)

 最少节点数 $F(n) = F(n-1) + F(n-2) + 1$

在插入节点时, 先按照BST的规则插入, 再做平衡化旋转

AVL的平衡化旋转

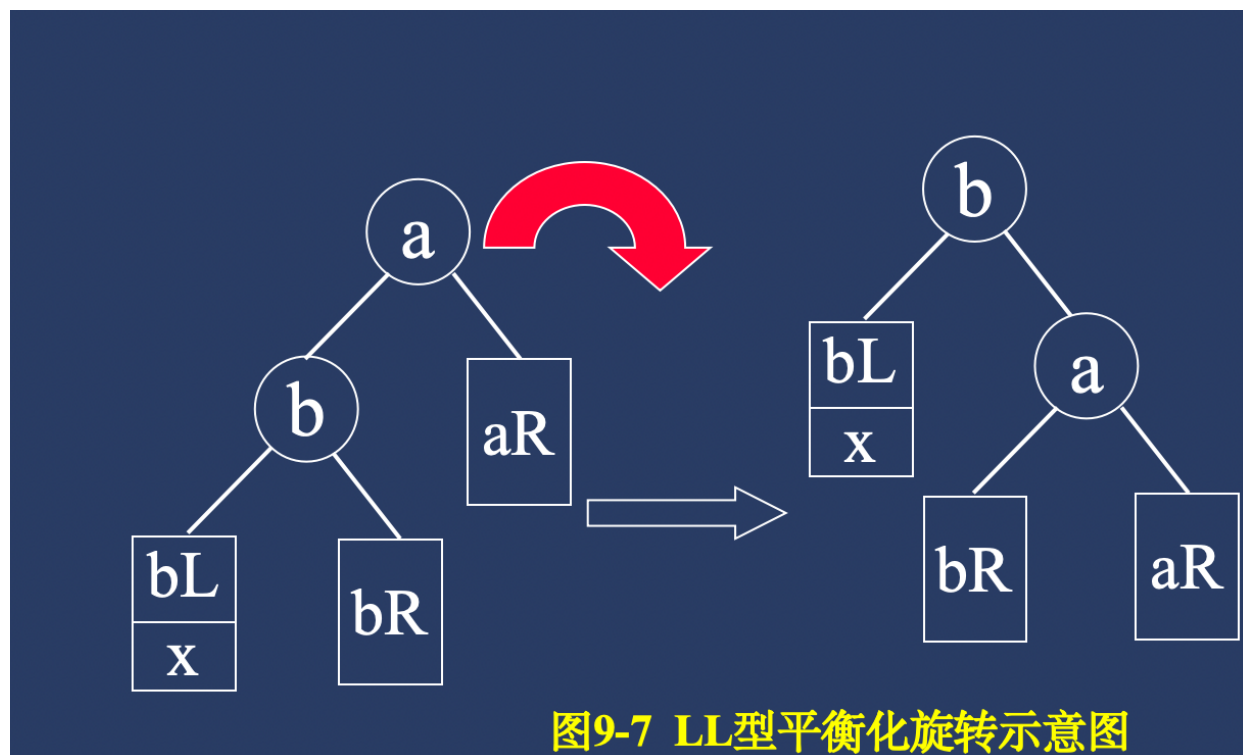
这种情况对 $BF=0$ 的子树根节点不产生影响

而对 $BF=-1, 1$ 的子树, 插入新节点后失衡则需要进行平衡化旋转

后面的示意图都可以看作是一个BF不为0的子树

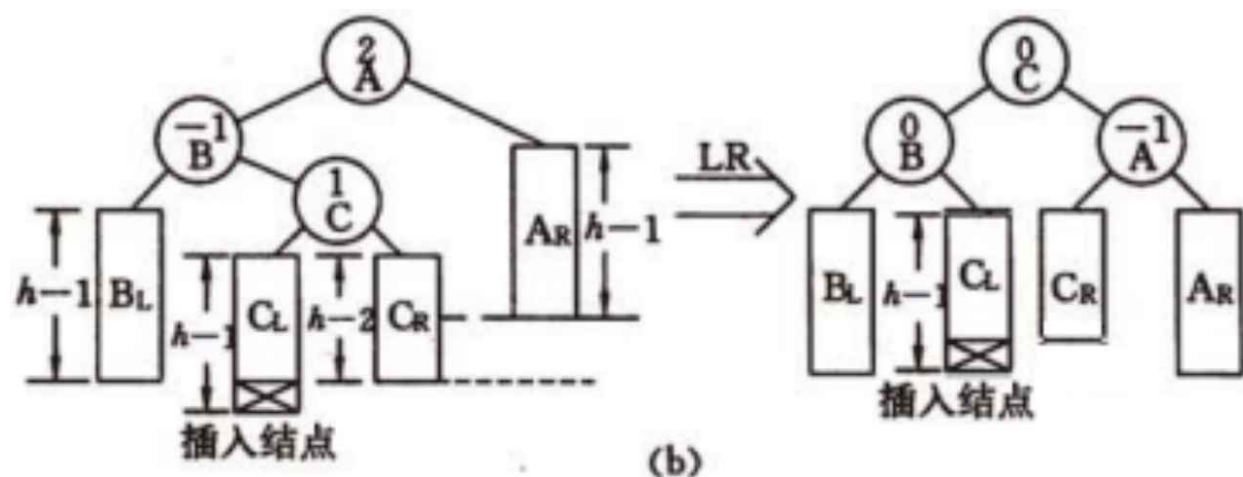
LL型

失衡原因：在结点a的左孩子的左子树上进行插入，插入使结点a失去平衡



LR型

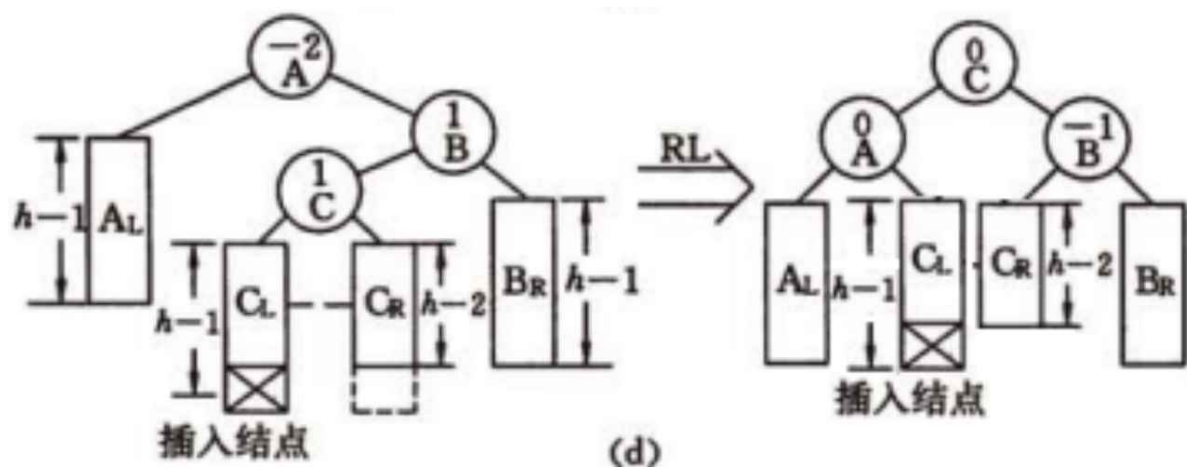
失衡原因：在结点a的左孩子的右子树上进行插入，插入使结点a失去平衡



核心思想是，插入导致BF从1变为2，故将这棵子树往上提一级以抵消BF的增加，同时保持BST的大小关系

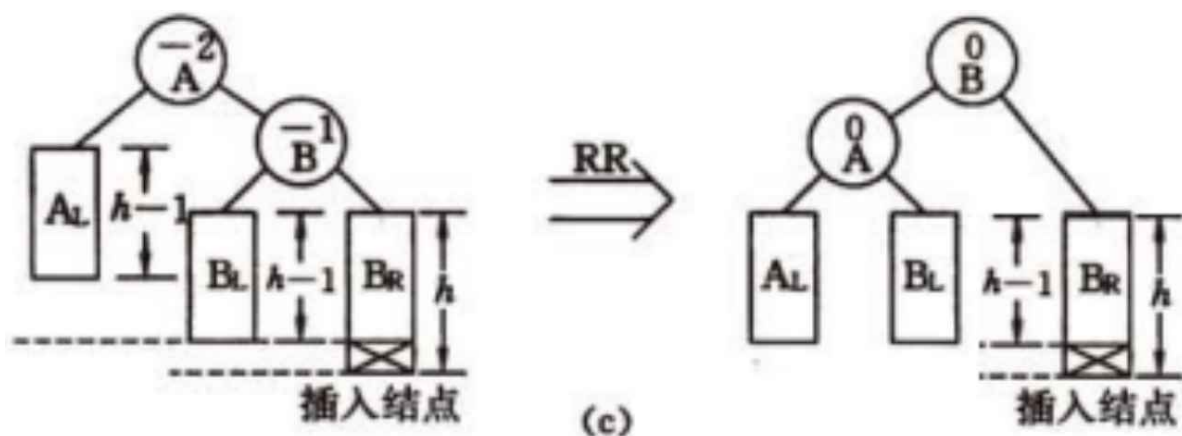
RL型

失衡原因：在结点a的右孩子的左子树上进行插入，插入使结点a失去平衡，与LR型正好对称



RR型

失衡原因：在结点a的右孩子的右子树上进行插入，插入使结点a失去平衡。要进行一次逆时针旋转，和LL型平衡化旋转正好对称



多路平衡查找树（B-tree）

定义

一棵 m 阶 $B-tree$ ，或者是空树，或者是满足以下性质的 m 叉树：

1. 根结点或者是叶子，或者至少有两棵子树，至多有 m 棵子树；
2. 除根结点外，所有非终端结点至少有 $\lceil \frac{m}{2} \rceil$ 棵子树，至多有 m 棵子树；
3. 所有叶子结点都在树的同一层上；

插入 & 删除

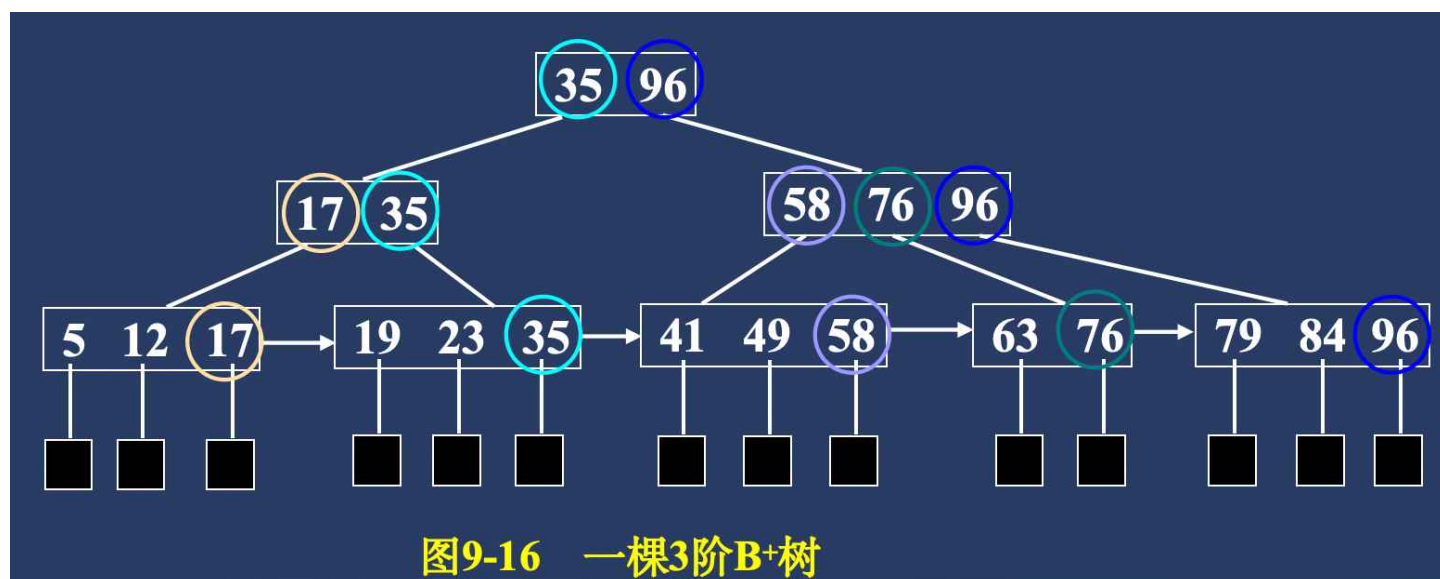
对一个节点，有 n 个关键字，则有 $n + 1$ 棵子树，且 $\lceil \frac{m}{2} \rceil \leq n \leq m - 1$ ，在进行插入操作、删除操作时一旦子树数目超出这个范围就要进行“分裂”或“吸收”。

注：关键字范围如上，因此孩子范围为 $\lceil \frac{m}{2} \rceil + 1 \leq n \leq m$ ，即子树数量的范围

B+树

在实际的文件系统中，基本上不使用B-树，而是使用B-树的一种变体，称为m阶B+树。它与B-树的主要不同是叶子结点中存储记录。在B+树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B+树与m阶B-树的主要差异是：

1. 若一个结点有n棵子树，则必含有n个关键字；
2. 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；
3. 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。



在B-树基础上，为叶子结点增加链表指针

所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引

B+树总是到叶子结点才命中；

B+树的插入仅仅在叶子结点上进行。当叶子结点中的关键字个数大于m时，“分裂”为两个结点，两个结点中所含有的关键字个数分别是 $\lceil (m + 1) / 2 \rceil$ 和 $\lfloor (m + 1) / 2 \rfloor$ ，且将这两个结点中的最大关键字提升到父结点中，用来替代原结点在父结点中所对应的关键字。提升后父结点又可能会分裂，依此类推。

哈希查找

制定哈希函数，做一个映射，这里应该是主要掌握冲突处理方法。

题目会给出使用的函数“**散列函数**”，然后指明冲突解决办法，然后计算即可。

哈希表的填满因子 α ：

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表长度}}$$

各种散列函数所构造的散列表的ASL如下：

(1) 线性探测法的平均查找长度是：

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

$$S_{\text{nl成功}} \approx \frac{1}{2} \times (1 + \frac{1}{1-\alpha})$$

$$S_{\text{nl失败}} \approx \frac{1}{2} \times (1 + \frac{1}{(1-\alpha)^2})$$

(2) 二次探测、伪随机探测、再哈希法的平均查找长度是：

$$S_{\text{nl成功}} \approx -\frac{1}{\alpha} \times \ln(1-\alpha)$$

$$S_{\text{nl失败}} \approx \frac{1}{1-\alpha}$$

(3) 用链地址法解决冲突的平均查找长度是：

$$S_{\text{nl成功}} \approx 1 + \frac{\alpha}{2}$$

$$S_{\text{nl失败}} \approx \alpha + e^{-\alpha}$$

开放定址法

基本方法：当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。

线性探测法：

将散列表 $T[0 \dots m-1]$ 看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。增量序列为： $d_i = 1, 2, \dots, m-1$ 。

设初次发生冲突的地址是 h ，则依次探测 $T[h+1]$ ， $T[h+2]$...，直到 $T[m-1]$ 时又循环到表头，再次探测 $T[0]$ ， $T[1]$...，直到 $T[h-1]$ 。

二次探测法：

增量序列选择为： $d_i = 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$ ($k \leq [\frac{m}{2}]$ ，序列长度为 $m-1$)

伪随机探测法：

增量序列使用一个伪随机函数来产生一个落在闭区间 $[1, m-1]$ 的随机序列。

再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止

链定址法

将所有关键字为同义词(散列地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针。

例： 已知一组关键字(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)， 哈希函数为： $H(\text{key}) = \text{key} \text{ MOD } 13$ ， 用链地址法处理冲突， 如右图图9-17所示。

优点： 不易产生冲突的“聚集”； 删除记录也很简单。

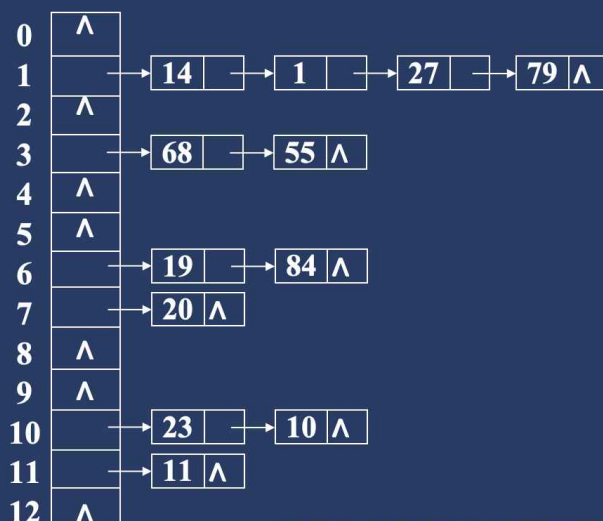


图9-17 用链地址法处理冲突的散列表

i.e. 将关于哈希函数的等价类放在一个哈希映像下，存成链表

建立公共溢出区

在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为 m ，设立基本散列表 $hashtable[m]$ ，每个分量保存一个记录；溢出表 $overtable[m]$ ，一旦某个记录的散列地址发生冲突，都填入溢出表中，e.g.

例：已知一组关键字(15, 4, 18, 7, 37, 47)，散列表长度为7，哈希函数为： $H(\text{key}) = \text{key} \bmod 7$ ，用建立公共溢出区法处理冲突。得到的基本表和溢出表如下：

Hashtable表:	散列地址	0	1	2	3	4	5	6
	关键字	7	15	37		4	47	
overtable表:	溢出地址	0	1	2	3	4	5	6
	关键字	18						

排序

区分一下内部排序和外部排序

内部排序：待排序记录存放在RAM中进行的排序

外部排序：待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中仍需对外存进行访问的排序

书中只讨论内部排序

排序动图：

https://blog.csdn.net/weixin_50886514/article/details/119045154?ops_request_misc=%257B%2522request%255Fid%25...

六大排序算法:插入排序、希尔排序、选择排序、冒泡排序、堆排序、快速排序_冒泡排序, 快速排序, 希儿排序, 堆排序 需要额外辅助空间-CSDN博客

文章浏览阅读10w+次，点赞4.7k次，收藏2.7w次。文章目录:1. 插入排序2. 希尔排序1. 插入排序步骤:1. 从第一个元素开始，该元...

插入排序

时间复杂度: $O(n^2)$

直接插入排序

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接插入排序的过程如下图10-1所示：

初始记录的关键字: [7] 4 -2 19 13 6

第一趟排序: [4 7] -2 19 13 6

第二趟排序: [-2 4 7] 19 13 6

第三趟排序: [-2 4 7 19] 13 6

第四趟排序: [-2 4 7 13 19] 6

第五趟排序: [-2 4 6 7 13 19]

图10-1 直接插入排序的过程

图10-1 直接插入排序的过程

折半插入排序

设有一组关键字30, 13, 70, 85, 39, 42, 6, 20, 采用折半插入排序方法排序的过程如图10-2所示:

i=1	(30)	13	70	85	39	42	6	20
i=2	13	(13 30)	70	85	39	42	6	20
			⋮					
i=7	6	(6 13 30 39 42 70 85)						20
i=8	20	(6 13 30 39 42 70 85)						20
		↑		↑		↑		
		low		mid		high		
i=8	20	(6 13 30 39 42 70 85)						20
		↑	↑	↑				
		low	mid	high				
i=8	20	(6 13 30 39 42 70 85)						20
			—	↑↑				
			low	mid	high			
i=8	20	(6 13 20 30 39 42 70 85)						

图10-2 折半插入排序过程

选择排序

每次从待排序列中选出一个最小值，然后放在序列的起始位置【以交换的方式】，直到全部待排数据排完即可。实际上，我们可以一趟选出两个值，一个最大值一个最小值，然后将其放在序列开头和末尾，这样可以使选择排序的效率快一倍。

希尔排序

1. 先选定一个小于N的整数gap作为第一增量，然后将所有距离为gap的元素分在同一组，并对每一组的元素进行直接插入排序。然后再取一个比第一增量小的整数作为第二增量，重复上述操作...

2. 当增量的大小减到1时，就相当于整个序列被分到一组，进行一次直接插入排序，排序完成。

时间复杂度： $O(n^2)$

△：希尔的间隔为 d 时，是指前一个开始数 d 个开始数 d 个值，两个元素中间是 $d - 1$ 个元素；亦即距离为 d 。

快速排序

以从小到大排序为例，选择key值后：

右哨兵向左走找到第一个小于key值的停住，左哨兵向右走找到第一个大于key值的停住，交换左右哨兵踩着的两个数，重复；若前面这个过程中出现哨兵相遇，则将key值更新为相遇点，左右两段分别快排递归。

最好： $O(n \log n)$ ；最坏： $O(n^2)$

不稳定！

https://www.bilibili.com/video/BV1j841197rQ/?spm_id_from=333.337.search-card.all.click&vd_source=d20c6ae79ecec1e...

三分钟学会快速排序_哔哩哔哩_bilibili

三分钟学会快速排序, 视频播放量 28730、弹幕量 12、点赞数 705、投硬币枚数 214、收藏人数 464、转发人数 207, 视频作者 有个知识, 作者简介 做一个知识分享者, 相关视频: 考场上快速写出快速排序结果(自用), 数据结构 快速排序的手排过程 【一学就会...

以这里讲的挖空形式为准

堆排序

注：堆是一个完全二叉树结构

【只允许最深一层不满】

构建大根堆，大的元素向上浮动；构建小根堆，小的元素向上浮动

对输入数据 a_0, a_1, \dots, a_{n-1} 分别在 $[1, n-1], [2, n-1], \dots, [n-2, n-1], [n-1, n-1]$ 范围上建堆，每建一个堆，将堆首取出。

堆首取出以后，将完全二叉树中编号最大的节点直接拿到根的位置，上滤或下滤继续建堆。

△：建堆完毕（把这个元素调整完）后才取出新的堆根



https://www.bilibili.com/video/BV1Mr4y1b7HX/?spm_id_from=333.337.search...

堆排序动画演示_哔哩哔哩_bilibili

经典排序算法堆排序动画演示视频制作工具 <https://github.com/3b1b/manim> 制作视频的代码你可以在这个仓库里面找到 <https://github.com/jimowushuang/manim>, ...

【从堆的定义到优先队列、堆排序】 10分钟看懂必考的数据结构——堆_哔哩哔哩_bilibili

目前做的最长的视频，做了两个星期，一个人工作量确实有点大了。。。视频合成软件 pr 视频引擎 manimCE 0.15.0 配音软件 edge 浏览器 (没想到吧) 最近忙考试，祝各位不挂科。 , 视频播放量 77675、弹幕量 329、点赞数 3411、投硬币枚数 2357、收藏...

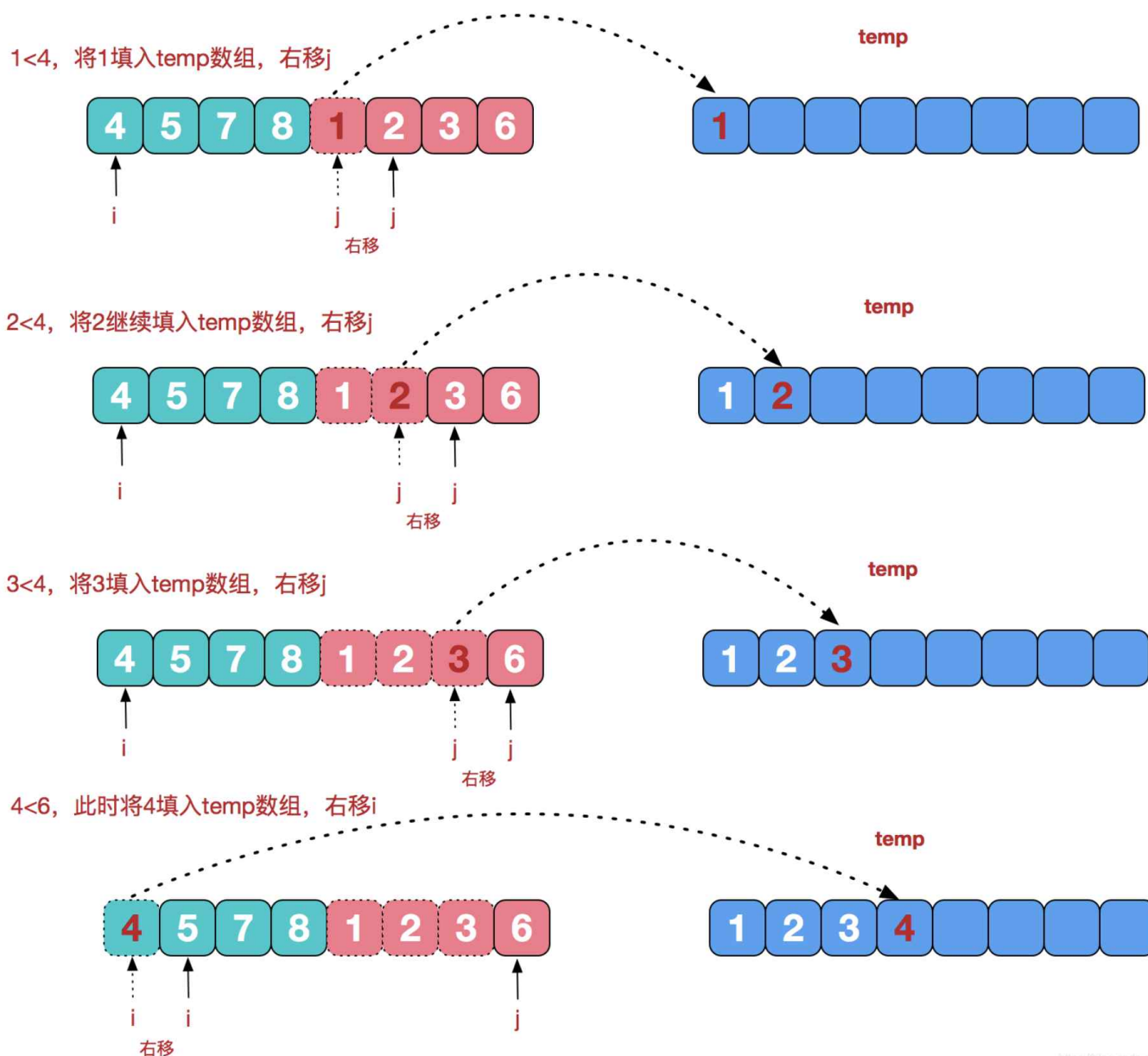
时间复杂度： $O(n \log n)$

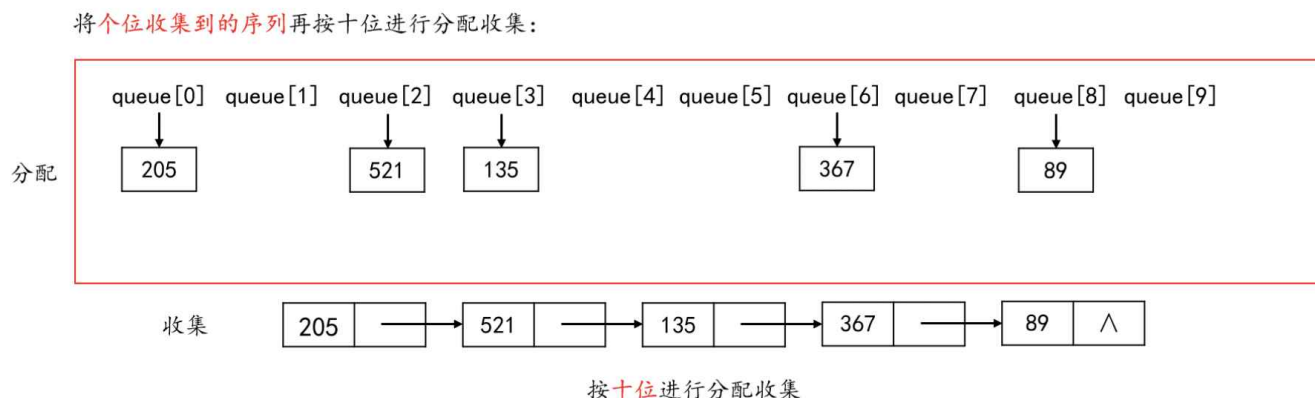
初始堆：

按照给定的输入顺序，填成一棵完全二叉树，然后做上滤和下滤即可；输入为 L ，则从编号为 $\lfloor \frac{L}{2} \rfloor$ （下去整）的节点开始向前调整。

⚠：调整一个节点，是在他的父子节点上浮动，而没有要求节点以后都不能移动。

归并排序





每个基数就是一个桶，把内容放到桶里面，以链表的形式链接桶内的元素

复杂度对比

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n + rd))$	$O(d(n + rd))$	$O(rd)$

稳定排序 & 不稳定排序

排序方式	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性	备注
快速	$O(n\log_2(n))$	$O(n^2)$	$O(n\log_2(n))$	$O(\log_2 n)$	不稳定	最坏比较次数 $\frac{n(n-1)}{2}$
归并	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n)$	稳定	
堆	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(1)$	不稳定	
冒泡	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	最坏比较次数 $\frac{n(n-1)}{2}$
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	
插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	最坏比较次数 $\frac{n(n-1)}{2}$
希尔	$O(n^{1.3-2})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	
基数	$*O(d(n+r))$	$*O(d(n+r))$	$*O(d(n+r))$	$*O(r)$	稳定	*d为位数，r为基数
计数	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定	k是整数的范围

https://blog.csdn.net/m0_54711064/article/details/126028756?ops_request_misc=%257B%2522request%255Fid%2522...

排序算法(稳定与不稳定)_稳定排序算法和不稳定排序算法-CSDN博客

文章浏览阅读990次，点赞2次，收藏12次。稳定排序与不稳定排序 稳定排序算法和不稳定排序算法

不稳定的排序算法有：快速排序、堆排序、选择排序、希尔排序

如果在一个待排序的序列中，存在2个相等的数，在排序后这2个数的相对位置保持不变，那么该**排序算法**是稳定的；否则是不稳定的。