计算机保研面试_数据结构常见题

1 绪论

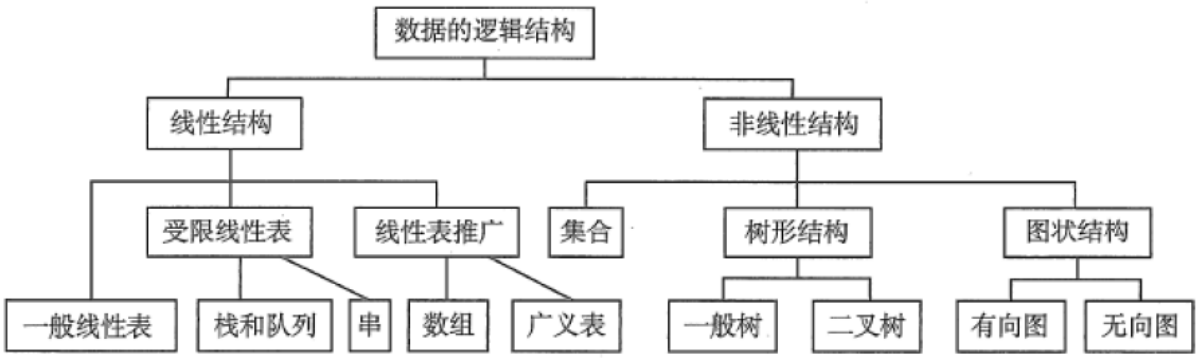
(1) 时间复杂度中的O

O: 最坏情况下的时间复杂度

渐进上界记号**O:**

$$O(g(n))=\{f(n)\mid \exists c>0 \text{和} n_0>0, \text{ s.t. } \forall n\geq n_0 \text{有} 0\leq f(n)\leq cg(n)\}$$

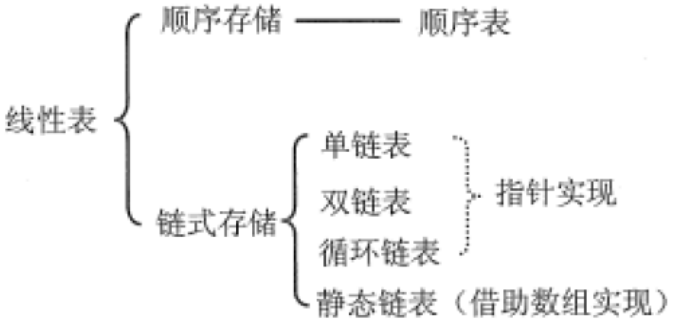
(2) 数据的逻辑结构



(3) 数据的存储结构

- **顺序存储**：把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现。
  - 优点：可以**随机存取，快；存储密度大**
  - 缺点：**删除，插入**操作**耗时**在移动元素上，只能使用相邻的一整块存储单元，因此可能产生较多的**外部碎片**
- **链式存储**：不要求逻辑上相邻的元素在物理位置上也相邻，借助指示元素存储地址的指针来表示元素之间的逻辑关系。
  - 优点：**删除，插入快，外部碎片少**，能充分利用所有存储单元
  - 缺点：**不能随机存取**
- **索引存储**：在存储元素信息的同时，还建立附加的索引表。
- **散列存储**：根据元素的关键字直接计算出该元素的存储地址，又称哈希(Hash) 存储

2 线性表

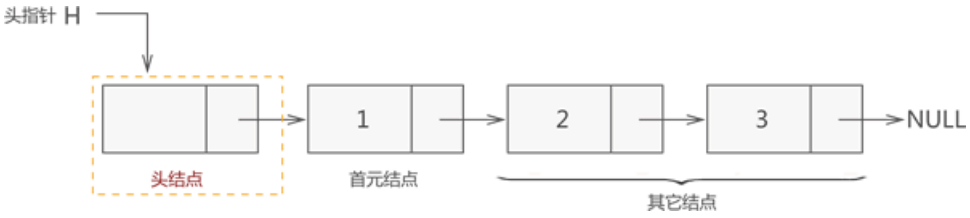


(1) 各种线性表的优缺点

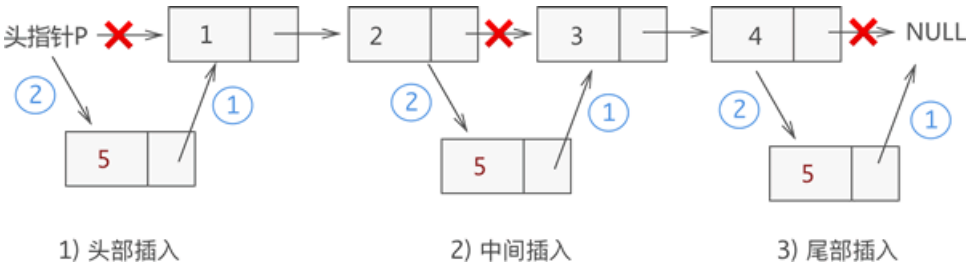
- 顺序表
  - 优点：可以**随机存取，快；存储密度大**
  - 缺点：**插入、删除效率低**；**存储空间固定**，分多了浪费，分少了又不足
- 单链表
  - 优点：**插入、删除效率高**；**空间可动态分配**
  - 缺点：**不能随机存取**，要顺序存取，**慢；存储密度不大**（有指针域）
- 静态链表：融合顺序表和单链表的优点，**既能快速访问元素，又能快速插入、删除元素**

(2) 单链表

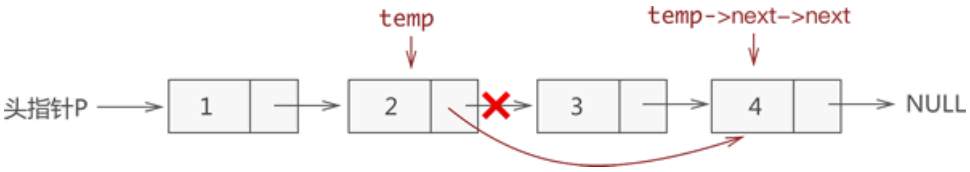
- 定义：



- 插入



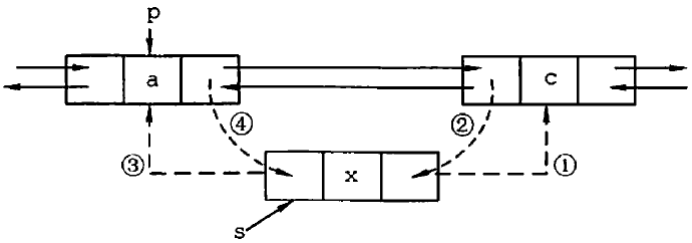
- 删除：



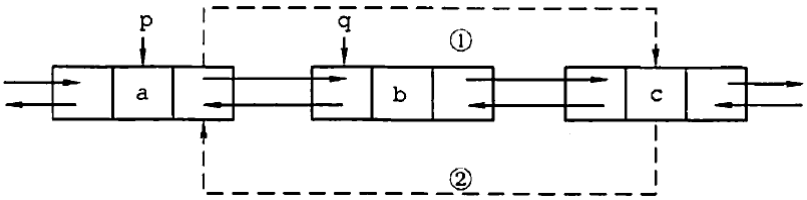
- 计算单链表长度：遍历一遍

(3) 双向链表

- 定义：
- 插入

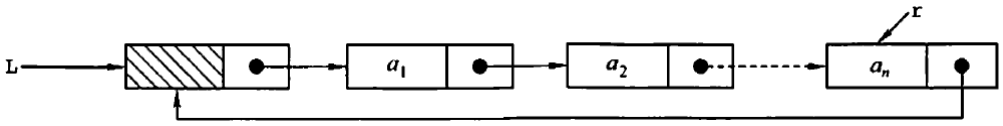


- 删除

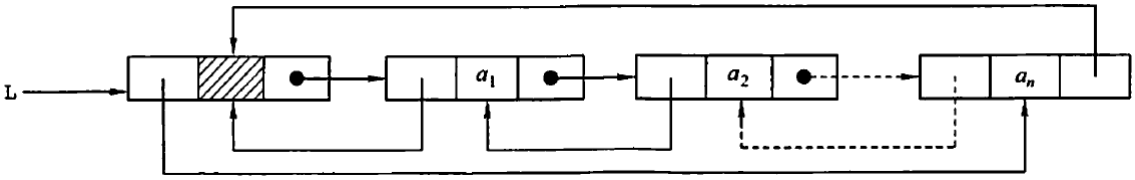


(4) 循环链表

- 单向循环链表



- 双向循环链表

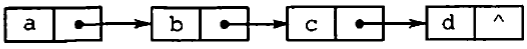


(5) 静态链表

需分配连续的内存空间，借助数组来描述链表，每个结点包含数据+游标（下一个元素的数组索引）

0		2
1	b	6
2	a	1
3	d	-1
4		
5		
6	c	3

(a)静态链表示例



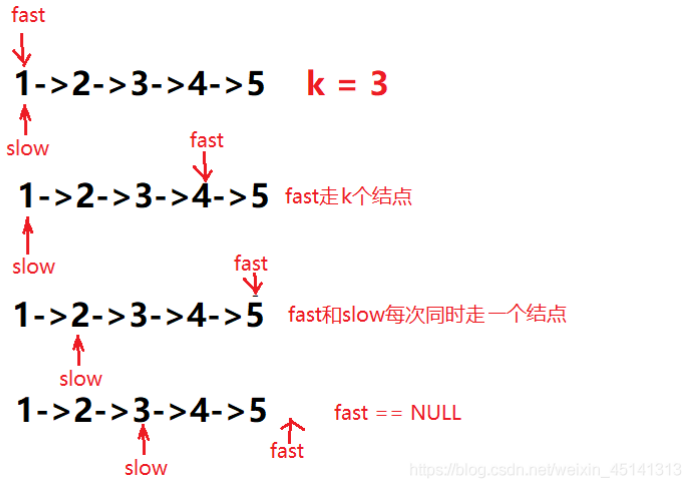
(b)静态链表对应的单链表

(6) 头指针和头结点的区别

- 头指针：**指向第一个节点存储位置的指针**，具有标识作用，头指针是链表的**必要**元素，无论链表是否为空，头指针都存在。
- 头结点：**放在第一个元素节点之前**，**便于**在第一个元素节点之前进行**插入和删除**的操作，头结点**不是**链表的**必须**元素，可有可无，头结点的数据域也可以不存储任何信息。

(7) 查找单链表中倒数第 k 个结点

1. 建立两个指针：fast、slow
2. 先让fast走k步
3. 再让fast和slow一起走，直到fast走到末尾后一位，slow指向的就是倒数第k个节点




```
listNode* get_kth_from_end(listNode *head, int k) {
    listNode *fast = head; // 先走的指针
    listNode *slow = head; // 后走的指针
    while (k--) fast = fast->next; // 先让fast走k步
    while (fast != NULL) { // 再让fast和slow一起走，直到fast走到末尾后一位
        slow = slow->next;
        fast = fast->next;
    }
    return slow;
}
```

(8) 单链表就地逆序（空间复杂度为O(1)）

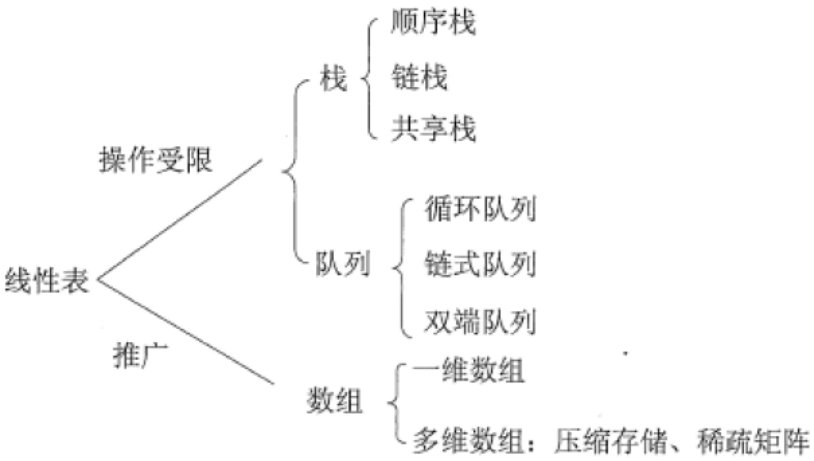
1. 建立两个指针：p、q
2. p 用于遍历链表中的每个节点
3. q 用于头插法创建新的逆序链表

```
void reverse(listNode* head) {
    listNode *p, *q;
    p = head->next;
    head->next = NULL;
    while (p) { // p用于遍历每个节点
        q = p;
        p = p->next; // 在该节点的next指针改变前更新p
        q->next = head->next; // 类似头插法
        head->next = q; // 类似头插法
    }
}
```

(9) 判断链表有没有环

快慢指针法：从头开始设置两个指针，快指针每次走2 步，慢指针每次走1 步，如果快指针先碰到尾，则无环，否则两个指针之后一定会重合，则有环。

3 栈、队列和数组

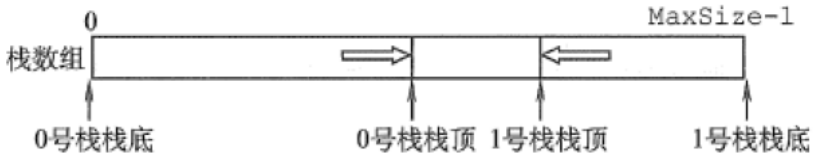


(1) 栈和队列的区别和存储结构

- 栈：
  - 定义：只允许在**表尾（栈顶）进行插入和删除**的线性表，“**先进后出**”
  - 顺序栈：数组（存放栈中元素）、栈顶指针
  - 链栈：栈顶指针
- 队列：
  - 定义：只允许在表的一端（**队尾）插入**，在另一端（**队首）删除**的线性表，“**先进先出**”
  - 顺序队列：数组（存放队列中元素）、头指针、尾指针
  - 链式队列：队首指针、队尾指针
- 两个栈模拟一个队列：队列是先进先出，栈的是先进后出。同一组数据连续执行两次先进后出之后再出栈就可以实现队列的先进先出。

(2) 什么是共享栈

利用**栈底位置相对不变**的特性，让**两个顺序栈共享一个一维数组空间**，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间的中间延伸。这样能够更**有效的利用存储空间，防止上溢。**



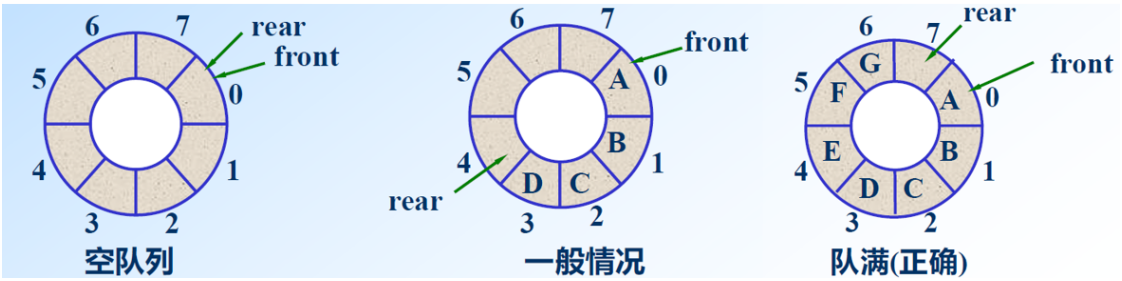
(3) 如何区分循环队列是队空还是队满

- 一般情况，队空和队满的判断条件都是Q.front == Q.rear

> front：指向第一个数
> rear：指向最后一个数的**下一个**位置，即将要入队的位置

- 怎么区分
  - 方法一：**牺牲一个单元**（即最后一个单元不存数据）来区分队空和队满

```
队空：Q.front == Q.rear
队满：(Q.rear+1) % MaxSize == Q.front
元素个数：(Q.rear-Q.front+MaxSize) % MaxSize
```



  - 方法二：队列结构体中增加一个Q.size表示元素个数

(4) 栈在括号匹配中的算法思想

- 设置一个空栈，顺序读入括号
  - 若是左括号，则进栈
  - 若是右括号
    - 若栈空，则匹配失败，右括号多余
    - 否则，弹出一个栈顶的左括号
- 读完所有括号后
  - 若栈空，则表达式中括号匹配正确
  - 否则，匹配失败，左括号多余

```
bool checkBrackets(string tokens) {
    stack<char> s;
    for (int i = 0; i < tokens.length(); i++) {
        char token = tokens[i];
        if (token == '(')
            s.push(token);
        else if (token == ')') {
            if (s.empty()) return false;
            else s.pop();
        }
    }
    return s.empty();
}
```

(5) 栈在后缀表达式求值中的算法思想

- 创建一个空栈，顺序扫描表达式的每一项
  - 若该项是**操作数**，则将其压栈
  - 若该项是**操作符**<op>，则连续从栈中pop出两个操作数Y和X，形成运算指令X<op>Y，并将计算结果重新压栈
- 当表达式的所有项都扫描完后，栈顶存放的就是最后的计算结果

(6) 队列的应用

- 分支限界算法
- 主机和外部设备（打印机）的速度不匹配问题，需要一个缓冲区（队列）
- 操作系统中，有多个进程请求占用CPU，需要排队

(7) 稀疏矩阵怎么存储

使用**三元组**存储，[i, j, value]

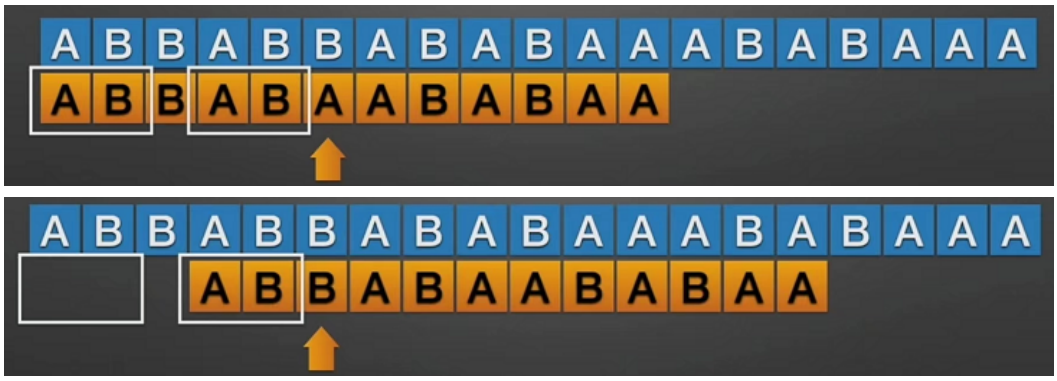
4 串

(1) kmp 算法

- kmp 是对 BF 算法（暴力匹配）的改进
  - 先计算出 next 数组：next[i] 表示子串切片 s[0:i] 的**最长公共前后缀的长度**，next 数组只和子串有关，和主串无关

> 如：ababaca
> next[0] : a 的最长公共前后缀的长度-1，即 -1（表示不存在）
> next[1] : ab 的最长公共前后缀的长度-1，即 -1（表示不存在）
> next[2] : aba 的最长公共前后缀的长度-1，即 0（表示长度为1）
> next[3] : abab 的最长公共前后缀的长度-1，即 1（表示长度为2）
> next[4] : ababa 的最长公共前后缀的长度-1，即 2（表示长度为3）
> next[5] : ababac 的最长公共前后缀的长度-1，即 -1（表示不存在）
> next[6] : ababaca 的最长公共前后缀的长度-1，即 0（表示长度为1）

  - 再基于 next 数组，让子串和主串的每个字符进行匹配，当出现匹配失败时，如果已匹配相等的序列中有某个后缀正好是子串的前缀，那么可以直接将子串滑动到与这些相等字符对齐的位置。这利用了**子串本身的最长公共前后缀信息**，使得**主串指针无须回溯。**

> 
>

- 假设主串长度为 n, 子串长度为 m，由于BF 算法中每趟匹配失败都是子串**后移一位**再从头开始比较，则 BF 算法的时间复杂度为 **O(mn)**，而 kmp 的时间复杂度仅为 **O(m+n)**

5 树

(1) 满二叉树、完全二叉树

- 满二叉树：除叶子结点外，每个结点的度都为 2 的二叉树
- 完全二叉树：除了最后一层外，其他层的每个结点的度都为 2 ，且最后一层也只是在最右侧缺少节点

(2) 如何由遍历序列构造一棵二叉树

- 先序遍历+中序遍历可以唯一确定一棵二叉树
  - 先由先序遍历的第一个结点得到二叉树的根结点
  - 再在中序遍历中找到该点，该点左边的就是其左子树，右边的就是其右子树
  - 依此递归，便能唯一确定这棵二叉树
- 后序遍历+中序遍历也可以唯一确定一棵二叉树
  - 后序遍历的最后一个结点就是二叉树的根结点
  - 其他一样
- 层序序列+中序序列也可以唯一确定一棵二叉树
  - 和先序+中序类似

- 先序遍历+后序序列，则无法唯一确定一棵二叉树

(3) 线索二叉树

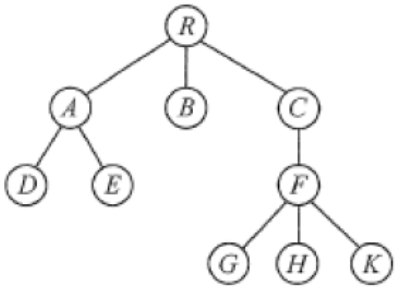
- n 个结点的二叉树有 n+1 个空指针域，将二叉树中所有为**空的左、右指针**指向某种遍历次序下该节点的**直接前驱、直接后继**结点，则得到的就是线索二叉树。
- 优点：用空指针域来存储结点之间前趋和后继关系，既不增加空间，还能加快查找结点的前驱和后继

(4) 二叉树的层次遍历

- 借助一个**队列**，先将二叉树的根结点入队
- 一个结点出队，先将左子树（如果有）入队，再将右子树（如果有）入队
- 重复上面的操作，直至队空

(5) 树的存储结构

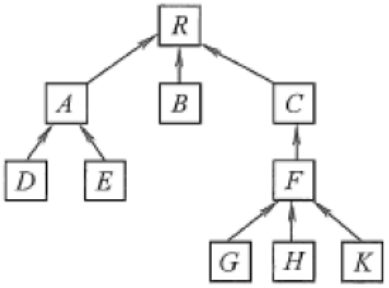
- **双亲表示法**：用一维数组存储结点，每个结点包括**结点值+双亲在数组中的索引**
  - 该方法找双亲快，找孩子慢（需遍历整个树）



(a) 一棵树

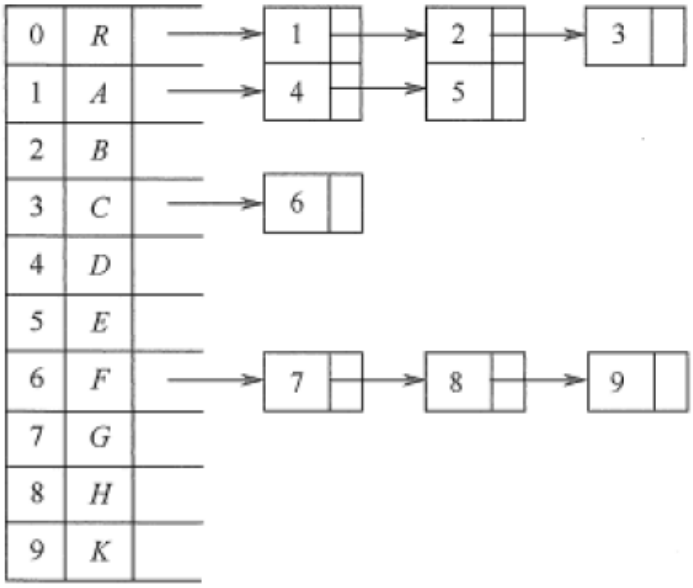
	data	parent
0	R	−1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

(b) 双亲表示



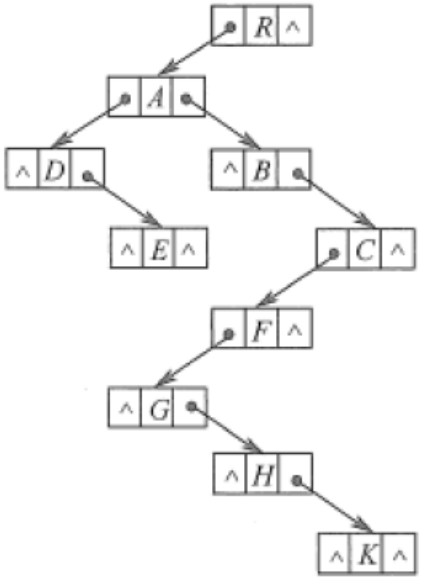
(c) 双亲指针图示

- **孩子表示法**：用数组存储每个结点，将每个结点的**孩子结点用单链表连接**起来
  - 该方法找孩子快，找双亲慢



(a) 孩子表示法

- **孩子兄弟表示法**：以**二叉链表**作为树的存储结构，每个结点包括**结点值、指向第一个孩子的指针、指向下一个兄弟的指针**
    - 优点是可以方便的转换为二叉树、易于查找孩子，缺点是找双亲麻烦（增加一个指向双亲的指针可以解决）

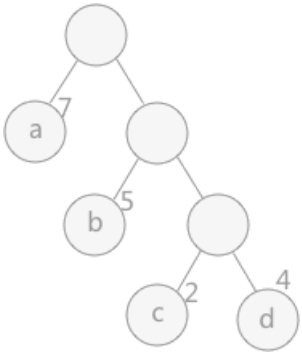


(b) 孩子兄弟表示法

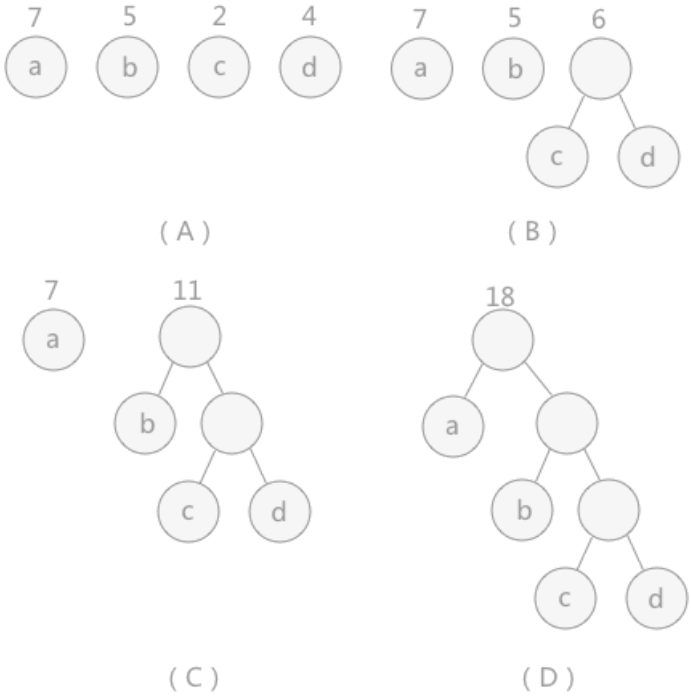
(6) 哈夫曼树

- 哈夫曼树又叫最优二叉树，在含有 n 个带权叶子结点的二叉树中，**所有叶子结点的带权路径长度之和**最小的二叉树是哈夫曼树

> WPL = 7 * 1 + 5 * 2 + 2 * 3 + 4 * 3



- 构建哈夫曼树
  1. 在 n 个权值中选出两个最小的权值，对应的两个结点组成一个新的二叉树，且新二叉树的根结点的权值为左右孩子权值的和；
  2. 在原有的 n 个权值中删除那两个最小的权值，同时将新的权值加入到 n–2 个权值的行列中，以此类推；
  3. 重复 1 和 2 ，直到所以的结点构建成了一棵二叉树为止，这棵树就是哈夫曼树。

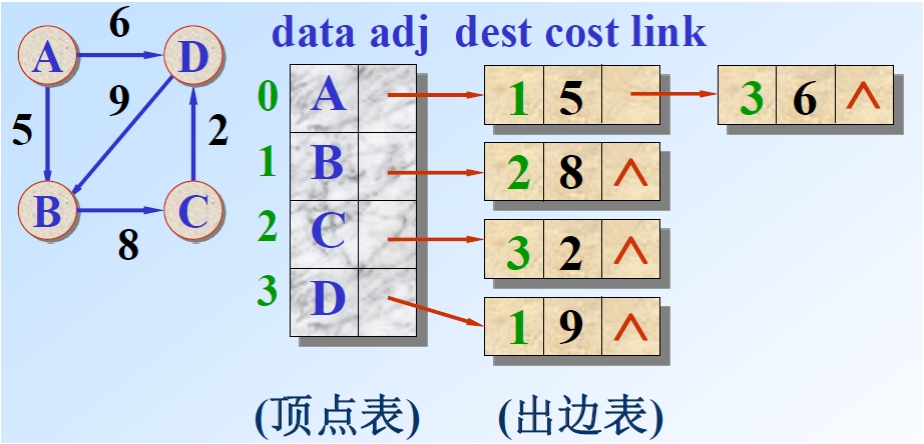


- 哈夫曼编码
  - 在通信中，对数据进行变长度的前缀编码（没有一个编码是另一个编码的前缀），**频率越高的字符，编码长度越短。**
  - 将每个字符当作叶子结点，其**权值为频度**，构造哈夫曼树，左子树编码为 0，右子树为 1
- 时间复杂度：$O(n \log n)$

6 图

(1) 图的存储结构

- **邻接矩阵法**：用一个一维数组存储图中顶点的信息，用一个二维数组存储图中边的信息（邻接矩阵），适用于稠密图
- **邻接表法**：包含两种表，顶点表和出边表
  - 用一个**顶点表**存储图中的顶点，每个**顶点表结点**包括结点值+指向出边表的指针
  - **出边表**用单链表连接着该顶点的所有出边，每个**出边表结点**包含出边指向的结点、出边的权重、next指针



- **十字链表法**：包含两种结点，边**结点和顶点结点**
  - 顶点结点：结点值+指向以该点为边头的边的指针+以该点为边尾的边的指针
  - 边结点：边尾结点的数组索引+边头结点的数组索引+指向边头相同的下一条边的指针+指向边尾相同的下一条边的指针+权重

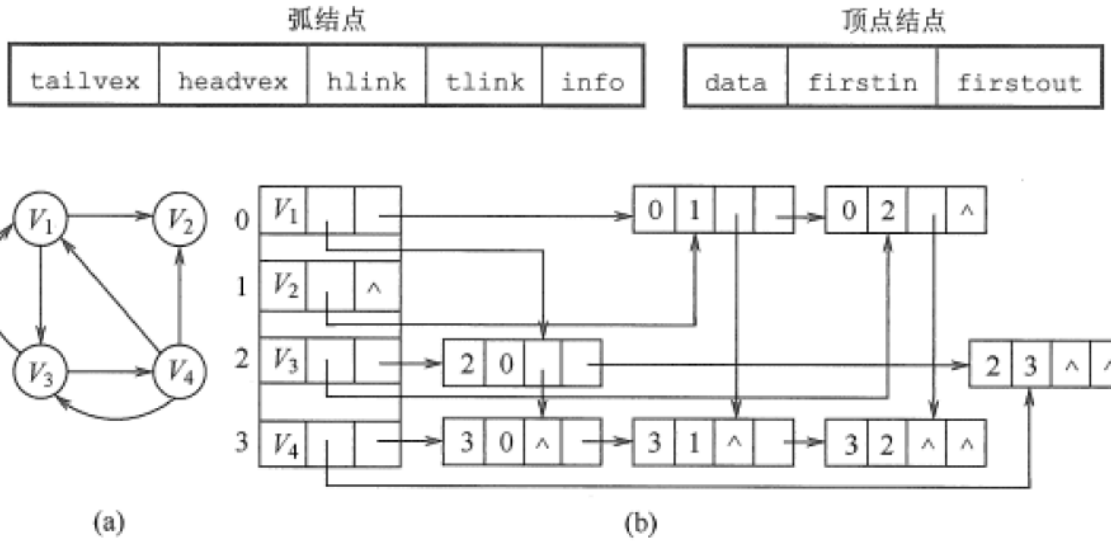


图 6.9　有向图的十字链表表示

(2) DFS 和 BFS

- DFS
  - 类似于二叉树的**先序遍历**，尽可能深的去搜索一个图
  - 先访问图中的起始顶点，再访问与该点邻接且未被访问的任一顶点 w1，再访问与 w1 邻接且未被访问的任一顶点 w2，…，重复该过程，直到不能继续访问为止。
  - 再依次回退到最近被访问的顶点，若它还有邻接顶点未被访问过，则从该点开始继续上述搜索过程，直至图中所有顶点均被访问过为止。

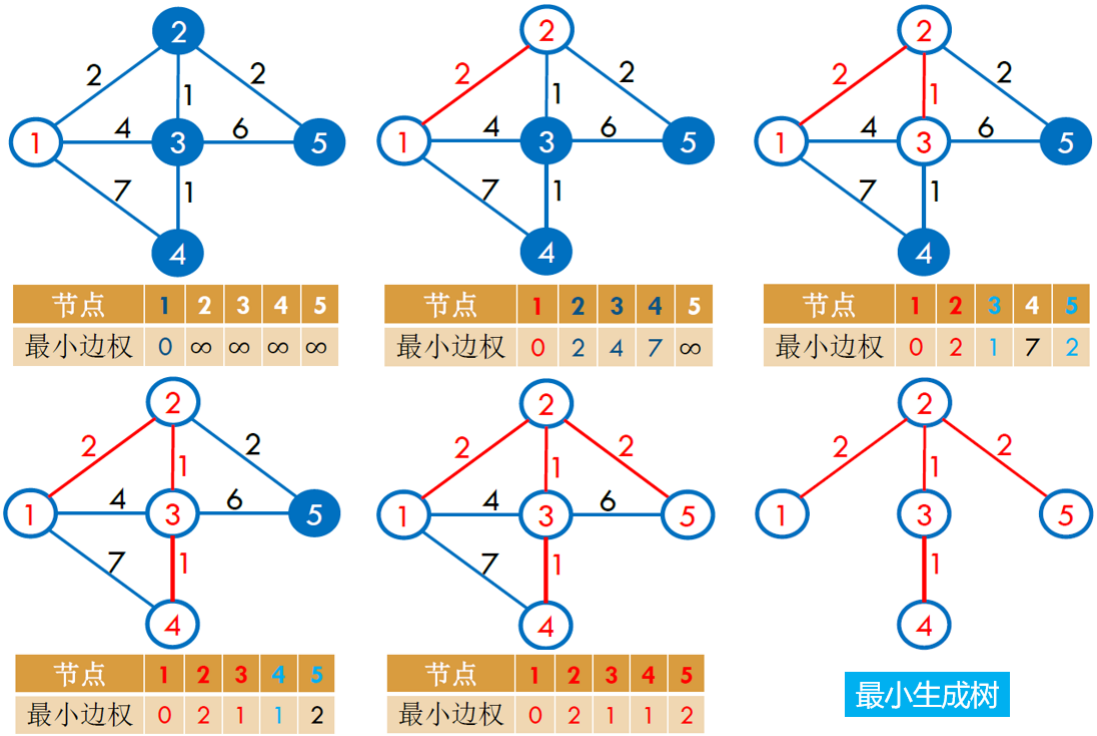
```
void dfs(参数) {
    if (终点边界) return;

    for (邻接 && 未被访问的点) {
        标记(vis[i][j]=1);
        dfs(下一个点);
        还原标记(vis[i][j]=0);
    }
}
```

- BFS
  - 类似于二叉树的**层序遍历**，以起点为中心，一层层向外扩
  - 将起始顶点放入一个 **队列**，如果队列非空，则从队列种取出一个点，并访问该点，然后将与该点邻接且未被访问的点都放入队列，重复上述过程，直到图中所有点都被访问过为止

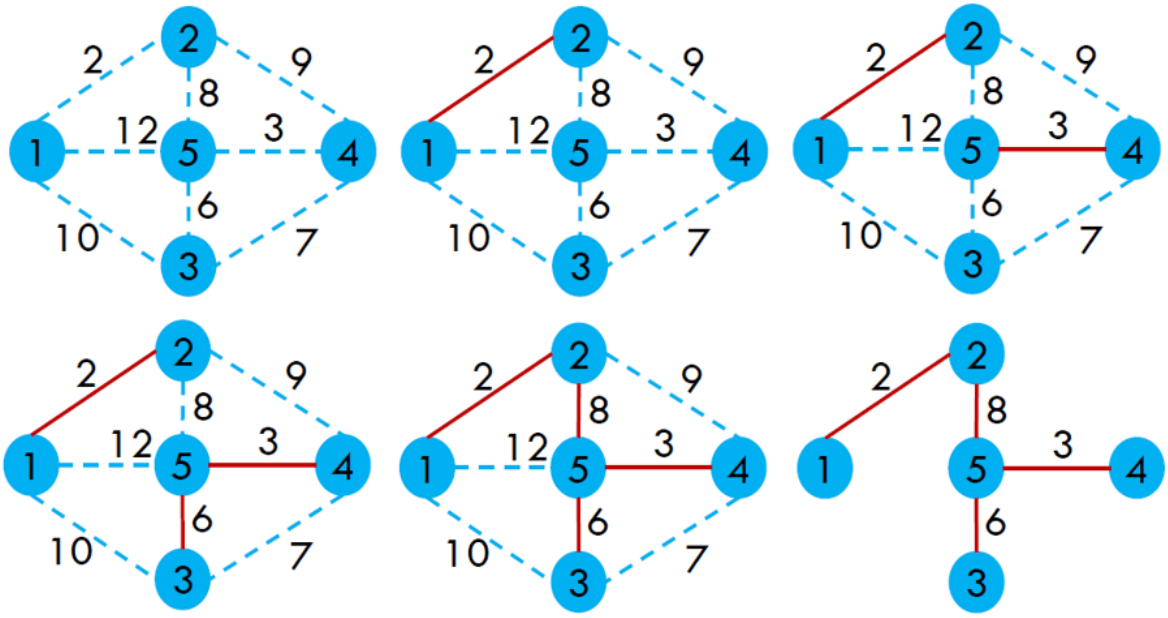
```
void dfs(参数) {
    队列 Q = {起始顶点};
    while (Q非空) {
        u = Q.pop();
        访问u;
        Q.push(所有与u邻接且未被访问的点);
        标记u为已访问;
    }
}
```

(3) 最小生成树算法

- 最小生成树：**边权之和**最小的生成树
- **Prim** 算法：贪心算法
  - 任取一个**顶点**加入集合
  - 找到与当前**集合距离最近的顶点**，将该点和对应的边加入集合
  - 重复上述过程，直至将所有顶点都加入集合



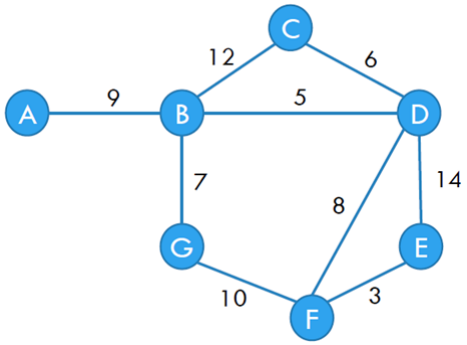
- **Kruskal** 算法：贪心算法
  - 对连通图的所有边按权值排序，取权值最小的**边**
  - 按**权重从小到大**选边，若加入该边后不构成**回路**，则取之，否则弃之
  - 重复上述过程，直至边数=顶点数-1为止



- 两个算法的区别
  - Prim 是**选点**，每次都要对边权排序；Kruskal 是**选边**，只需排序一次
  - Prim 的时间复杂度为 $O(|V|^2)$，堆优化后是 $O(|V|\log|V|)$，适合于**边稠密**的图；Kruskal 的时间复杂度为 $O(|E|\log|E|)$，适合于**边稀疏**的图

(4) 最短路径算法

- **Dijkstra** 算法：求解单源最短路径的**贪心**算法，要求**边权不为负**
  - 将源点放入一个顶点集合，计算各点通过该集合到源点的距离，不可达记为无穷
  - 选取**距离源点最近**的点，加入顶点集合，并更新各点通过该集合到源点的距离
  - 重复上述过程，直到所有点都放入集合为止
  - 时间复杂度为 $O(n^2)$，堆优化后是 $O(n\log n)$



1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞

2. S={A}, 更新各节点到源点的距离。

节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

3. S={A, B},

节点	A	B	C	D	E	F	G
离源点距离	0	9	21	14	∞	∞	16

4. S={A, B, D},

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

5. S={A, B, D, G}(没有更新)

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

6. S={A, B, D, G, C}(没有更新)

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

7. S={A, B, D, G, C, F}

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	25	22	16

8. S={A, B, D, G, C, F, E}, 算法结束。

- **Floyd** 算法：求解多源图最短路径的**动态规划**算法
  - 基本思想是对于一对顶点 i, j；看看是否存在一个顶点 k，使得 i 到 k，再到 j 的路径比已经路径更短，如果是，就更新它（松弛）。
  - 具体来说，递推产生一个 n 阶方阵序列，$c_{ij}(k)$ 表示节点 i 到节点 j 的中间节点编号不超过 k 的最短路长度，它等于节点 i 到节点 j 的中间节点编号不超过 k-1 的最短路长度，节点 i 到节点 k 的中间节点编号不超过 k-1 的最短路长度加上节点 k 到节点 j 的中间节点编号不超过 k-1 的最短路长度，这两者中的最小。

$$c_{ij}(k)=\min\{c_{ij}(k-1),c_{ik}(k-1)+c_{kj}(k-1)\}$$

  - 时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$

(5) 拓扑排序

- 拓扑排序的使用对象是**有向无环图**（DAG）
- 思想：从图中选择**入度为 0** 的顶点输出，并删除该点和所有以它为起点的边，重复该过程，直到图为空
- 应用：用一个 DAG 图表示一个工程，每个顶点表示一个任务，有向边表示做任务 B 之前要先做任务 A，用拓扑排序就可以在满足先后约束的条件下对任务排序了。

7 查找

[种树：二叉树、二叉搜索树、AVL树、红黑树、哈夫曼树、B树、树与森林 看，未来的博客-CSDN博客]

(1) 静态查找的方法有哪些

- 顺序查找：关键字和所有元素一个个对比，O(n)
- **二分**查找：要求查找表为**顺序存储**结构且**有序**，将 n 个元素分成大致相等的两部分，取 **a[n/2] 和关键字作比较**，如果相等，则找到了；如果大于或者小于，则递归地在相应的半部分查找，O(logn)
- **分块**查找：将查找表分成若干子表，保证子表间有序，而子表内无序，将各子表的最大元素构成一张索引表。查找时先用索引表找到位于哪个块，再在块内进行顺序查找。

(2) 二叉搜索树

- 二叉搜索树（二叉排序树）：左子树上所有节点的值都小于根节点的值，右子树上所有节点的值都大于根节点的值，它的左右子树也分别为二叉搜索树。
- 二叉搜索树**中序遍历**的结果是有序的。
  - 删除一个结点时，当左、右子树都不为空，则在**右子树**上找**中序**的第一个子结点填补
- 二叉搜索树方便查找，思想类似于**二分搜索**
- 时间复杂度：O(logn)（平均）、O(n)（最坏）

(3) 平衡二叉树

- 平衡二叉树（AVL树）：是一种特别的二叉排序树，它的**左右子树的高度差的绝对值不超过 1**
  - 平衡因子：左子树的高度减去右子树的高度，它的值只能为1，0，-1
- 二叉搜索树上的操作的时间复杂度取决于树高，二叉树容易退化为一条链，而平衡二叉树通过降低树的高度，提高效率
- 插入结点时，要保证平衡，则可能要进行**平衡旋转**，包括四种情况
  - 在左子树的左子树上插入节点时向右进行单向旋转；
  - 在右子树的右子树上插入节点时向左进行单向旋转；
  - 在左子树的右子树插入节点时先向左旋转再向右旋转；
  - 在右子树的左子树插入节点时先向右旋转再向左旋转。

(4) 红黑树

- 红黑树是一种**自平衡**的二叉搜索树，保证时间复杂度为 O(logn)
- 平衡二叉树为了保持平衡，需要通过频繁的旋转操作来调整树的结构，比较费时，而红黑树进一步**放宽平衡条件**，实际效率更高。
- 红黑树的有以下5个特点：
  - 结点非黑即红
  - 根结点一定是黑色
  - 叶子结点（NIL）一定是黑色
  - 不存在两个相邻的红结点
  - 从任一结点到每个叶子结点的所有路径，都包含相同数目的**黑色**节点

(5) B 树和 B+ 树

- B 树是所有结点的**平衡因子均等于 0** 的多路平衡查找树
  - B 树的**一个结点可以存储多个值**，这些值是按照递增排序的，**一个结点有多个子树**，所有叶子结点都位于同一层
  - 优点：**数据库**查询中，通过 B 树来存储数据，可以减少树高，即可以**减少磁盘 I/O** ，提高查询效率（结点内查找是在内存进行的，比磁盘快得多）
  - 缺点：**不利于区间查找**，如要找 0~100 的索引值，那么 B 树需要多次从根结点开始逐个查找

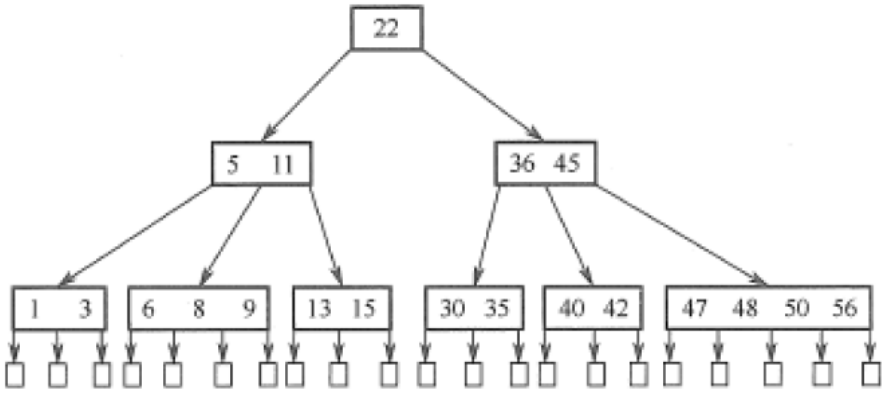
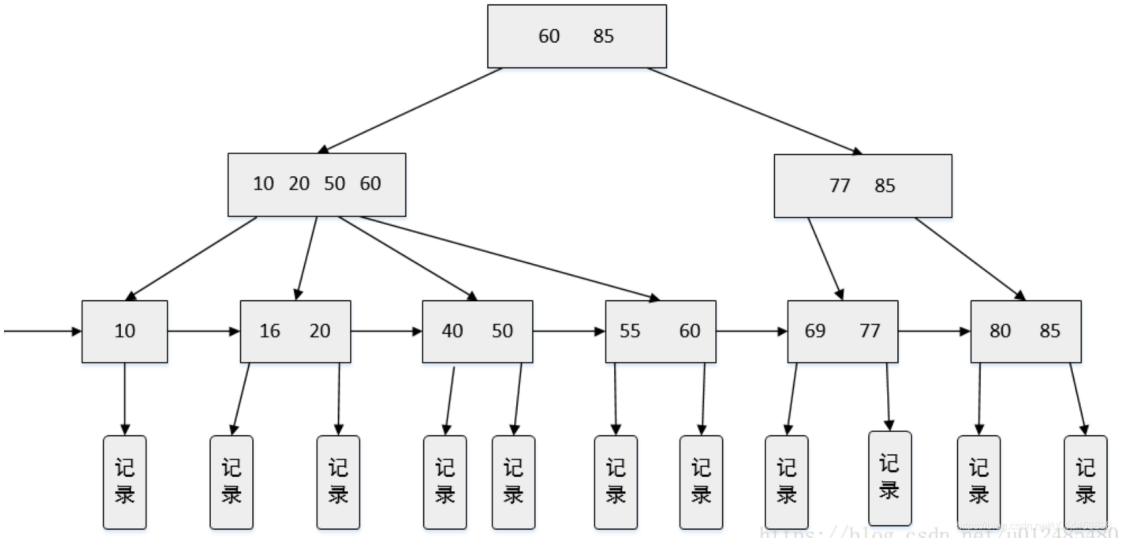


图 7.4　一棵 5 阶 B 树的实例

- B+ 树是对 B 树的改进，MySQL 种用的就是 B+ 树
  - 它的**叶子结点**直接有指针，构成了一个**链表**，这个链表是**有序**的，即可以直接通过遍历链表进行**区间查找**
  - **非叶子结点的元素在叶子结点上都冗余**了，非叶子结点仅仅是作为快速查找的**索引**



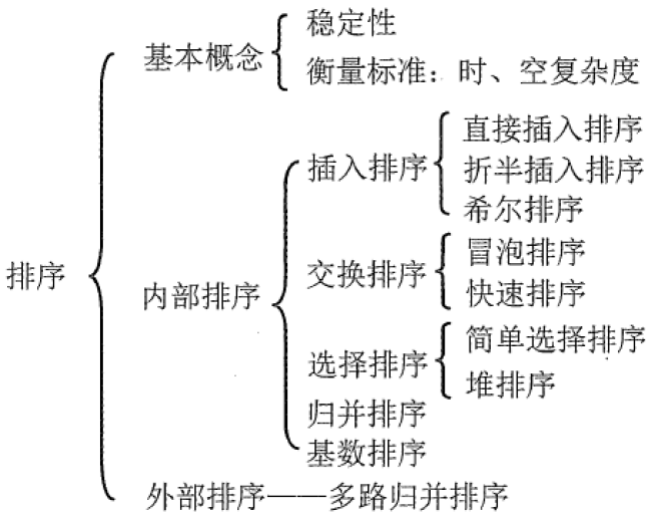
(6) 哈希表

- 哈希表（散列表）：通过**哈希函数**将查找表中的关键字**映射**成一个地址来加快查找速度
- 哈希函数的构造
  - 直接定址法：取关键字的某个**线性函数值**作为地址，适用于关键字分布均匀的情况
  - 除留余数法：取关键字**对某个数取余**的值作为地址
  - 数字分析法：当关键字位数很大时，取分布均匀的**任意几位**作为地址
  - 平方取中法：取关键字**的平方的中间几位**作为地址

(7) 如何解决哈希冲突

- 哈希冲突：两个不同的数经过哈希函数计算后得到了同一个地址
- **开放定址**法：遇到哈希冲突时，去寻找一个新的空闲的哈希地址
- **链接**法：将所有哈希地址相同的记录都链接在同一链表中
- **再哈希**法：构造多个哈希函数，如果发生哈希冲突时就使用另外的哈希函数计算
- **溢出表**法：将哈希表分为基本表和溢出表，将发生冲突的都存放在溢出表中

8 排序



(1) 快速排序

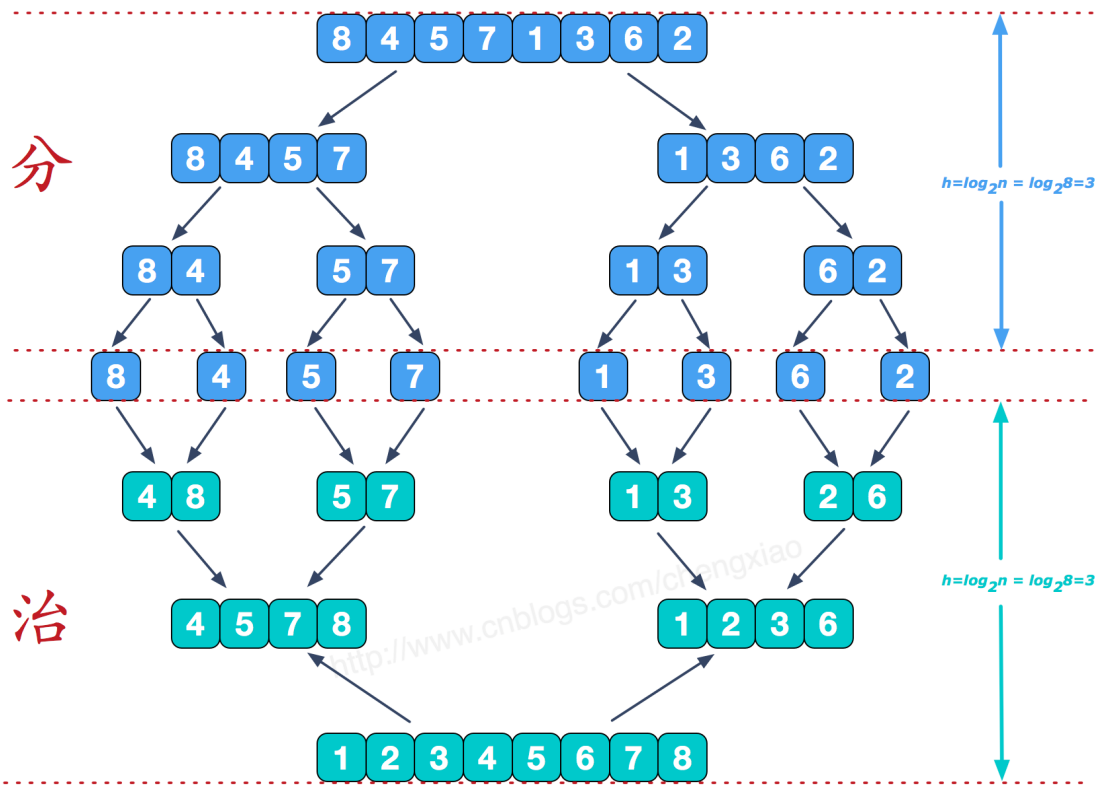
- 基本思想
  - 基于分治的思想，先选择一个支点 x，然后将数组划分由不大于 x 的元素构成的 A_L 和由大于 x 的元素构成的 A_R
  - 移动支点，使得A_L在其左边，A_R在其右边
  - 递归地对A_L和A_R进行排序，直到规模为O(1)为止
- 计算复杂度
  - 平均时间复杂度为 O(nlogn)，最坏情况为 $O(n^2)$，即支点每次都是最值
  - 平均空间复杂度为 O(logn)，最坏情况为 O(n)

```
void quick_sort(int a[], int l, int r) {
    if (l >= r) return;

    int i = l; // 左指针
    int j = r + 1; // 右指针
    int x = a[l]; // 基准点
    while (true) { // 保证x的左边都<=x, x的右边都>=x
        do i++; while (a[i] < x); // 从左边找到一个比x大的元素
        do j--; while (a[j] > x); // 从右边找到一个比x小的元素
        if (i >= j) break; // 找不到交换的元素对
        swap(a[i], a[j]);
    }

    swap(a[l], a[j]); // 放置基准点
    quick_sort(a, l, j-1);
    quick_sort(a, j+1, r);
}
```

(2) 归并排序

- 基于分治的思想，将数组划分为2部分，各个部分递归地进行排序后，再将所得的结果合并在一起，时间复杂度为 O(nlogn)

```
void merge_sort(int a[], int l, int r) {
    if (l >= r) return;

    int mid = (l+r) / 2;
    merge_sort(a, l, mid);
    merge_sort(a, mid+1, r);

    // 合并
    int i = l; // 左半部分未合并的第一个数的索引
    int j = mid + 1; // 右半部分未合并的第一个数的索引
    int k = 0; // 临时数组索引
    while (i <= mid && j <= r) // 到某个半部分合并完为止
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    while (i <= mid) tmp[k++] = a[i++]; // 合并左半部分剩下的
    while (j <= r) tmp[k++] = a[j++]; // 合并右半部分剩下的

    for (i = l, j = 0; i <= r; i++, j++)
        a[i] = tmp[j];
}
```



(3) 为什么快排比归并好

- 辅助空间：快速排序是一种**就地排序**算法，而归并排序需要使用额外的空间
- 最坏情况：快排可以通过**随机选择支点**来减少最坏情况的发生
- 缓存友好：快排的引用**局部性**比较好，对缓存比较友好

(4) 其他的内排序算法

- **冒泡**排序：通过遍历所有的**相邻的两个数**，如果他们逆序了，就交换位置，依次确定数组中的每个位置

```
void bubble_sort(int a[], int n) {
    for (int i = 0; i < n-1; i++) //确定从右数第i个数
        for (int j = 0; j < n-1-i; j++)
            if (a[j] > a[j+1])
                swap(a[j], a[j+1]);
}
```

- **选择**排序：通过遍历所有数，找到**第 k 小**的数，即数组中位置 k 的元素

```
void select_sort(int a[], int n) {
    for (int i = 0; i < n; i++) { //确定从左数第i个数
        int min = i; //当前最小值的索引
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min])
                min = j;

        if (i != min)
            swap(a[i], a[min]);
    }
}
```

- **插入**排序：就像**打扑克时边抓牌边理牌**一样，先把第1个数当成已排序序列，第2个数就插到合理位置；再把第1,2个数当成已排序序列，第3个数就插到合理位置；依次类推即可

```
void insert_sort(int a[], int n) {
    for (int i = 1; i < n; i++) { //第i个数待插入
        int key = a[i]; //待插入的数
        for (int j = i-1; j >= 0; j--) //对于第i个数左边的所有数
            if (a[j] > key) {
                a[j+1] = a[j]; //右移1位
                a[j] = key; //原位置赋为key
            }
    }
}
```

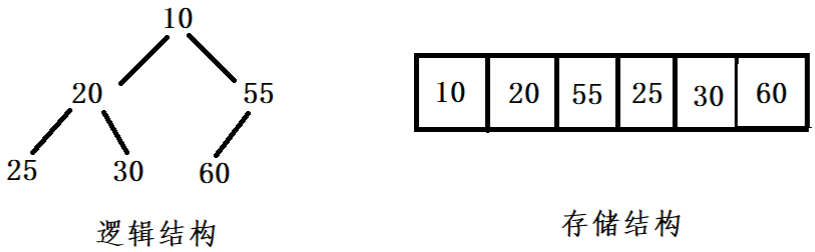
(5) 什么是排序的稳定性，哪些算法稳定

- 排序算法的稳定性：数组中值相等的元素，在排序后其**相对次序**若能保持不变，则算法是稳定的
- 稳定的算法：冒泡、插入、归并
- 不稳定的算法：选择、**快排**

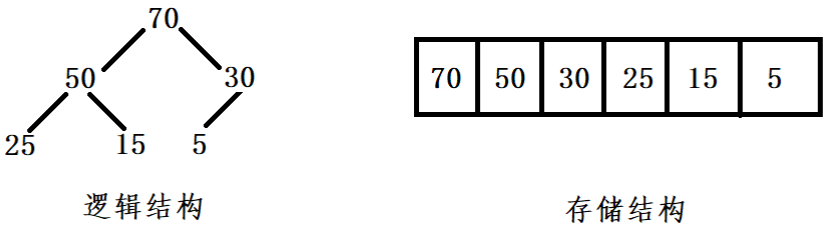
(6) 堆

[数据结构之堆Hidden.Blueee的博客-CSDN博客堆]

- 堆：把一个关键码的集合中的所有元素按**完全二叉树**的顺序存储方式存储在一个一维数组中，如果父结点的值比子结点大，则是**大根堆**；如果父结点的值比子结点小，则是**小根堆**
- 创建大根堆：
- 堆排序：通过将无序表转化为堆，可以直接找到表中最大值或者最小值，然后将其提取出来，令剩余的记录再重建一个堆，取出次大值或者次小值，如此反复执行就可以得到一个有序序列，此过程为堆排序



小 根 堆



大 根 堆

(7) 外排序

- 之前的排序都是内排序，即在内存中完成的；而对于大数据集，内存是一次性存不下的，需要在内存和磁盘上进行多次数据交换
-