

- Data Struct = (D, S, P)

数据结构课程体系

$$\text{Data Struct} = (\text{D}, \text{S}, \text{P})$$

- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <ul style="list-style-type: none"> ■ 第一章 绪论 ■ 第二章 线性表 ■ 第三章 栈和队列 ■ 第四章 串 ■ 第五章 数组和广义表 ■ 第六章 树和二叉树 ■ 第七章 图 ■ 第九章 查找 ■ 第十章 排序 | } 关系 (结构S) |
| | } 操作 (算法P) |
| <ul style="list-style-type: none"> • 静态查找表 (线性、树形) • 动态查找表 (树形) • 哈希查找 (混合) | |

一些概念

查找的定义：在一堆数据中找到指定的元素

- 数据在本地时：查字、文件、字符
- 搜索问题、云计算&存储

查找效率与数据结构密切关联

- 集合、线性、树形、网状

- * 查找表：由同种类型的数据元素构成的集合
 - * 静态：不能插入、删除
 - * 动态：可以插入、删除

- * 关键字：可以标记数据元素的数据项
 - * 主关键字：唯一标识：DNA、指纹
 - * 次关键字：识别若干记录

- * 平均查找长度 (ASL)：
 - * 其中： p_{-i} 为搜索第*i*个元素的概率
 - * c_{-i} 为搜索到第*i*个元素的比较次数

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i * c_i$$

9.1 静态查找表

1. 表结构

- 顺序、链式

2. 无序表（集合）

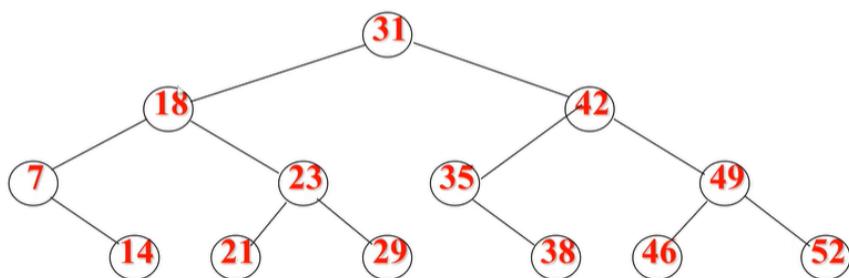
- 集合结构：松散的，无序的
- 查找方法：从一端扫描到另一端（头->尾，尾->头）
 - 查找成功、查找失败分别返回
- 技巧：将0号单元或最后一个设置为“哨兵”
 - 免去了朝朝过程中每一步都要检查是否查找完毕
 - 设计算法，使最后查看哨兵，若在哨兵中找到目标，则可判断原集合中没有
- 算法分析： $ASL_{(succ)} = (n+1)/2$

3. 有序表（线性）-折半查找

- 已知有序表，从左向右依次增大
- 与中间元素比较
 - 相等：成功
 - 小于中间元素：在左半区查找
 - 大于中间元素：在右半区查找

* low, mid, high 三个指针

- 折半查找可用二叉树来描述：判定树
- **折半查找过程可用二叉树来描述，称这个描述查找过程的二叉树为判定树。例如：(7, 14, 18, 21, 23, 29, 31, 35, 38, 42, 46, 49, 52) 的判定树为：**



$$ASL = (1*1 + 2*2 + 4*3 + 6*4) / 13 = 3.15$$

- 查找次数为结点的“层数-1”，每个概率相同，即可得出上述式子，算出ASL
- 算法分析

- **查找过程：从判定树的根节点搜索到该元素结点的过程，**

比较次数就是路径上经过的结点数，**最大比较次数**是树的

高度。对于有n个结点的判定树，树高为 $d = \lfloor \log_2 n \rfloor + 1$ 。

因此，折半查找在查找成功时，比较次数至多为 $d = \lfloor \log_2 n \rfloor + 1$ 。**平均查找长度**为：

$$\begin{aligned} ASL &= \sum_{i=1}^n P_i C_i \\ &= (1/n)[1 \times 2^0 + 2 \times 2^1 + \dots + k \times 2^{k-1}] \\ &= \frac{n+1}{n} \log_2(n+1) - 1 \quad \text{比顺序查找的ASL小} \end{aligned}$$

4. 静态树表（树形）

- 有序表中被查概率相等时，折半查找性能最优
- 被查概率不等时，折半查找不是最优
 - 对判定树加上概率作为权值，重新选根，建树，能算出新的ASL
- 问题：如何构造判定树
- 基本思路：概率大的结点与根节点的距离小，概率小距离大，类似于Huffman

PH，路径之和

- 带权路径长度之和PH，包括非叶子结点的部分

- $PH = \sum_{i=1}^n w_i h_i$

$w_i = cp$, c是查找概率， h_i 是i结点层次

可证：PH 与 ASL 成正比

- 构造 次优判定树

次优判定树构造

- 从有序表R中取第i个记录构造根节点，使满足

$$\Delta P = \min\left\{\left|\sum_{j=i+1}^h w_j - \sum_{j=l}^{i-1} w_j\right|\right\}$$

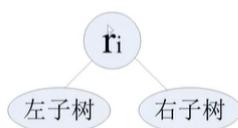
-

次优判定树构造方法

设有序表为 $R = (r_l, r_{l+1}, \dots, r_h)$

相应记录的权值为 $(w_l, w_{l+1}, \dots, w_h)$

次优判定树构造方法是：首先从有序表R中取第*i* ($l \leq i \leq h$) 个记录构造根节点，使得满足



$$\Delta P = \min\left\{ \left| \sum_{j=i+1}^h w_j - \sum_{j=l}^{i-1} w_j \right| \right\}$$

(右子树权总值与左子树总权值的差最小)

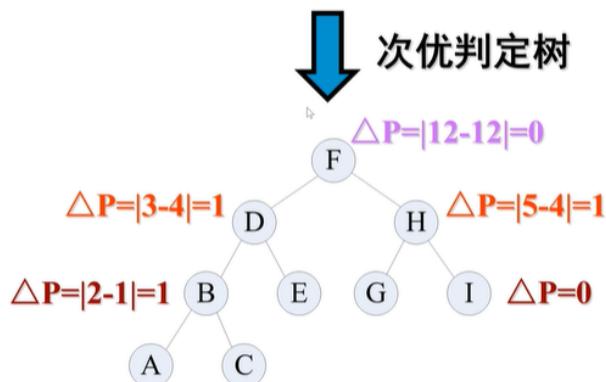
然后再对 $R_{\text{左}} = (r_l, r_{l+1}, \dots, r_{i-1})$ 和 $R_{\text{右}} = (r_{i+1}, r_{i+2}, \dots, r_h)$ 进行同样操作

算法见算法9.3，复杂度为 $O(n \log_2 n)$

例如：

关键字序列： (A, B, C, D, E, F, G, H, I)

权 值： (1, 1, 2, 5, 3, 4, 4, 3, 5)

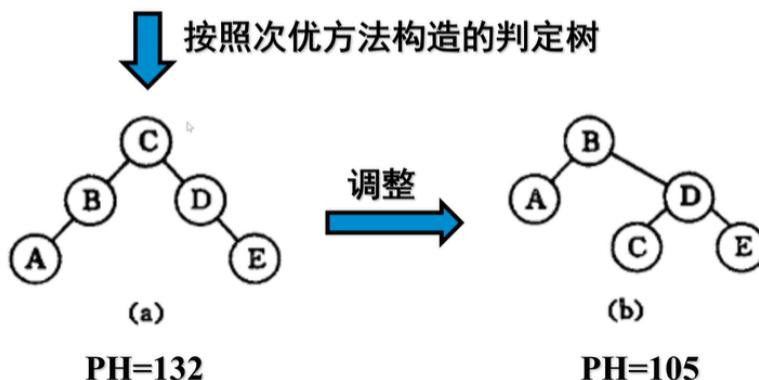


查找方法：从根节点进行搜索，大往右，小往左

- 例 9-2 需要调整

例 9-2 已知含 5 个关键字的有序表及其相应权值为

关键字	A	B	C	D	E
权值	1	30	2	29	3

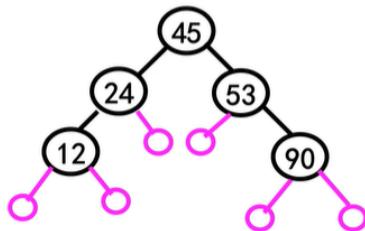


平均查找长度分成功/不成功

求平均查找长度ASL

- 成功情况：待查元素在查找表里面
- 不成功情况：待查元素不在查找表里面

例：查找表 (12, 24, 45, 53, 90)



- 成功情况ASL:
$$ASL = (1*1+2*2+2*3)/5=11/5$$
- 不成功情况ASL:
$$ASL=(2*3+4*4)/6=11/3$$

- 备注：不成功而落到24右子：比较了2次；不成功而落到53左子：比较了2次。
- 罗列出所有不成功的情况
- 对于不成功的部分：相当于把他们视为“成功”，算其加权平均、

5.索引顺序查找（分块）

- 查找过程：分成几块，块中无序，块间有序；先确定待查所在块，再在块内查找
- 算法实现：
 - 用数组存放带查找记录
 - 建立索引表，结点含有最大关键字域和指针指向本块第一个结点的指针

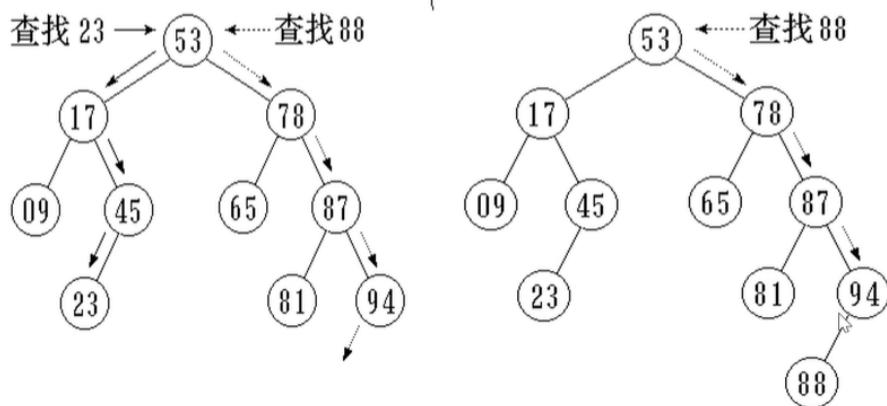
9.2 动态查找表（树形结构）

- 表结构在查找过程中动态生成
- 若有，则查找成功
- 否则，插入关键字等于key的记录
- 二叉排序树

二叉排序树

1. 左子树 所有节点 小于 根
 2. 右子树 反之
 3. 左右子树也是二叉排序树
- 在查找中动态生成

例如：



- 具体算法

```
BiTree SearchBST(BiTree T, KeyType key){  
    // 在根指针 T 所指二叉排序树中递归地查找某关键字等于 key 的数据元素，  
    // 若查找成功，则返回指向该数据元素结点的指针，否则返回空指针  
    if((!T) || EQ(key,T->data.key)) return(T);           // 查找结束  
    else if LT(key,T->data.key) return(SearchBST(T->lchild,key));  
                                              // 在左子树中继续查找  
    else return(SearchBST(T->rchild,key));           // 在右子树中继续查找  
} // SearchBST
```

- 二叉排序树的中序序列：有序序列

二叉排序树的插入和删除

插入操作

- 小于根：左边走
- 大于根：右边走
- 边“查找”，边“插入”

删除操作

- 原则：中序有序、高度不增加
- 删除叶子节点：记得释放空间
- 只有左子树/右子树：孩子顶替之
- 左右都有：在右子树中寻找中序遍历下的第一个结点，代替被删，再来处理（自己想……）

查找算法分析

- 比较次数：结点层次数（最多比较次数：树的深度： $\log_2(n) + 1$ ）
- 平均查找长度ASL：
 - 单支：n, $ASL = (n+1)/2$, 退化为顺序查找
 - 于折半查找判定树相同： $(\log_2(n)+1)$, $ASL = \log_2(n+1) - 1$, 折半查找
- 平均性能：每个查找概率相同，则ASL与 $\log n$ 同数量级
- 平衡二叉树

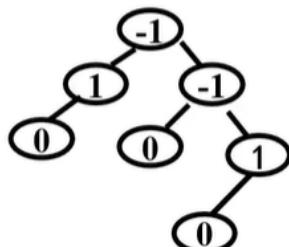
平衡二叉树

- 左右树是平衡二叉树
- 平衡因子Balance：该结点左子树与右子树的高度差：左子树高度 - 右子树高度，存在正负

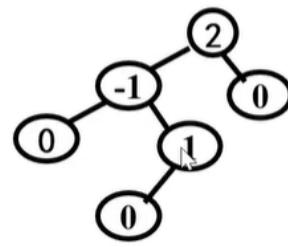
为了方便起见，给每个结点附加一个数字，给出该结点左子树与右子树的高度差。这个数字称为结点的**平衡因子balance**。这样，可以得到AVL树的其它性质：

- 所有结点，左、右子树深度只差 ≤ 1
- 左右结点高度只差小于等于1
- 对于一颗 n 个结点的 AVL树，其高度保持在 $O(\log_2(n))$ 数量级，ASL也保持在 $O(\log_2(n))$ 数量级

例如：



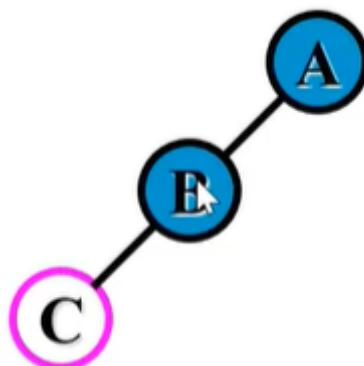
a、平衡树

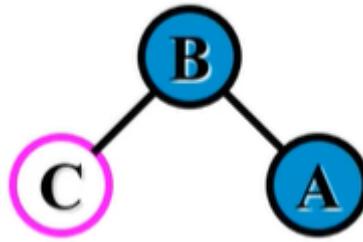


b、不平衡树

平衡旋转调整之

- LL平衡旋转
 - 若在左子树插入节点，使其平衡因子从1增加到2；则进行一次顺时针旋转，中序序列不变

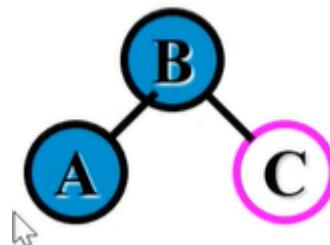
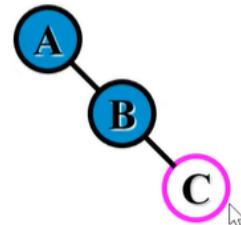




- RR平衡旋转

2) RR平衡旋转:

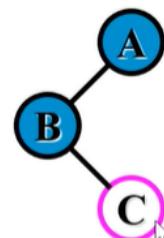
若在A的右子树的右子树上插入节点，使A的平衡因子从-1增加至-2，需要进行一次逆时针旋转



- LR平衡旋转

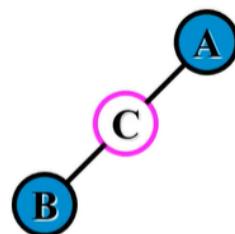
3) LR平衡旋转:

若在A的左子树的右子树上插入节点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转



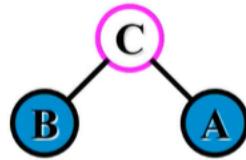
3) LR平衡旋转:

若在A的左子树的右子树上插入节点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转



3) LR平衡旋转:

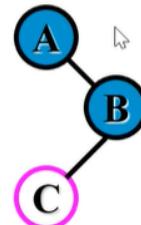
若在A的左子树的右子树上插入节点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转



- RL平衡旋转

4) RL平衡旋转:

若在A的右子树的左子树上插入节点，使A的平衡因子从-1增加至-2，需要先进行顺时针旋转，再逆时针旋转



9.3 哈希查找

哈希表的概念

集合（散列） -> 顺序 -> 树

- 建立函数关系
- 关键字 存储位置 对应关系

【例如】

11个元素的关键码分别为 18, 27, 1, 20, 22, 6, 10, 13, 41, 15, 25。选取关键码与元素位置间的函数为 $f(key)=key \bmod 11$

1. 通过这个函数对11个元素建立查找表如下：

0	1	2	3	4	5	6	7	8	9	10
22	1	13	25	15	27	6	18	41	20	10

2. 查找时，对给定值 k_x 依然通过这个函数计算出地址，再将 k_x 与该地址单元中元素的关键码比较，若相等，查找成功。

- 构造哈希函数
- 冲突：多个关键字 映射到 同样的地址了

两个问题

1. 构造好的哈希函数
 - a. 简单，提高速度
 - b. 得出的地址大致均匀分布，减少空间浪费
2. 好的解决冲突的方案

哈希函数的构造方法

常用的哈希函数构造方法有：

- 直接定址法
- 除留余数法
- 乘余取整法
- 数字分析法
- 平方取中法
- 折叠法
- 随机数法

- 直接定址

$$Hash(key) = a \times key + b$$

1、直接定址法

$$\text{Hash(key)} = a \cdot \text{key} + b \quad (a, b \text{ 为常数})$$

即取关键码的某个线性函数值为哈希地址，这类函数是一一对应函数，不会产生冲突，但要求地址集合与关键码集合大小相同，因此，对于较大的关键码集合不适用

例如：关键码集合为{100, 300, 500, 700, 800, 900}，选取哈希函数为

$$\text{Hash(key)} = \text{key}/100, \text{ 则存放如下:}$$

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

- 除留余数

$$\text{Hash(key)} = \text{key} \% p$$

- 选取合适的p：表长为m，则p<=m，且接近m或等于m。p一般选取质数

- 乘余取整法

$$\text{Hash(key)} = \lfloor B \times (A \times \text{key} \% 1) \rfloor$$

- 0 < A < 1, B 为整数

- 数字分析

数字分析法根据r种不同的符号，在各位上的分布情况，选取某几位，组合成哈希地址。所选的位应是各种符号在该位上出现的频率大致相同

【例如】有一组关键码如下：

3 4 7 0 5 2 4
3 4 9 1 4 8 7
3 4 8 2 6 9 6
3 4 8 5 2 7 0
3 4 8 6 3 0 5
3 4 9 8 0 5 8
3 4 7 9 6 7 1
3 4 7 3 9 1 9

第1、2位均是“3和4”，第3位也只有“7、8、9”，因此，这几位不能用，余下四位分布较均匀，可作为哈希地址选用。若哈希地址是两位，则可取这四位中的任意两位组合成哈希地址，也可以取其中两位与其它两位叠加求和后，取低两位作哈希地址。

①②③④⑤⑥⑦

- 平方取中

对关键码平方后，按哈希表大小，取中间的若干位作为哈希地址。例如：教材上的例子

- 折叠法

此方法将关键码自左到右分成位数相等的几部分，最后一部分位数可以短些，然后将这几部分叠加求和，并按哈希表表长，取后几位作为哈希地址。这种方法称为折叠法。通常有两种叠加方法：

1. 移位法 —— 将各部分的最后一一位对齐相加。
2. 间界叠加法 —— 从一端向另一端沿各部分分界来回折叠后，最后一位对齐相加

- 随机法

选择一个随机函数，将关键字作为这个随机函数的自变量，随机函数的函数值作为关键字的哈希地址。

$$H(key) = \text{random}(key)$$



当关键字的长度不相等时，采用此方法构造的哈希函数比较适合。

冲突处理方法

- 开放定址法
- 链地址法
- 再哈希

开放定址法

- 基本公式

$$H_i = (Hash(key) + d_i) \% m$$

- 冲突产生，则寻找下一个空的哈希地址
- 对 d_i 的取法Hash冲突的解决

对增量 d_i 的三种取法：

1) 线性探测再散列

$d_i = c \times i$ 最简单的情况 $c=1$

2) 二次探测再散列

$d_i = 1^2, -1^2, 2^2, -2^2, \dots,$

3) 随机探测再散列

d_i 是一组伪随机数列 或者

$d_i = i \times H_2(\text{key})$ (又称双散列函数探测)

注意：增量 d_i 应具有“完备性”

即：产生的 H_i 均不相同，且所产生的 $s(m-1)$ 个 H_i 值能覆盖哈希表中所有地址。则要求：

- ※ 平方探测时的表长 m 必为形如 $4j+3$ 的素数（如：7, 11, 19, 23, ... 等）；
- ※ 随机探测时的 m 和 d_i 没有公因子。

1. 线性探测法

其中， d_i 为增量 = 1, 2, 3, ..., $m - 1$

关键码集为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，哈希表表长为 11，

$\text{Hash(key)} = \text{key mod } 11$ ，用线性探测法处理冲突，建表如下：

0	1	2	3	4	5	6	7	8	9	10
11	22		47	92	16	3	7	29	8	

△ ▲ △ △

47、7、11、16、92 均是由哈希函数得到的没有冲突的哈希地址； $\text{Hash}(29)=7$ ，哈希地址有冲突，需寻找下一个空的哈希地址：由 $H_1 = (\text{Hash}(29)+1) \bmod 11 = 8$ 哈希地址 8 为空，将 29 存入。另外，22、8 同样在哈希地址上有冲突，也是由 H_1 找到空的哈希地址的；

- 查找时，先令 $d_i = 0$ ，若不对，则依次查找

2. 二次探测

$$H_i = (\text{Hash(key)} \pm d_i) \bmod m$$

其中：

Hash(key) 为哈希函数

m 为哈希表长度， m 要求是某个 $4k+3$ 的质数；

d_i 为增量序列 $1^2, -1^2, 2^2, -2^2, \dots, q^2$

链地址法

将有相同 Hash 地址的记录成一个单链表， m 个 hash 地址， m 个点链表，动态的结构

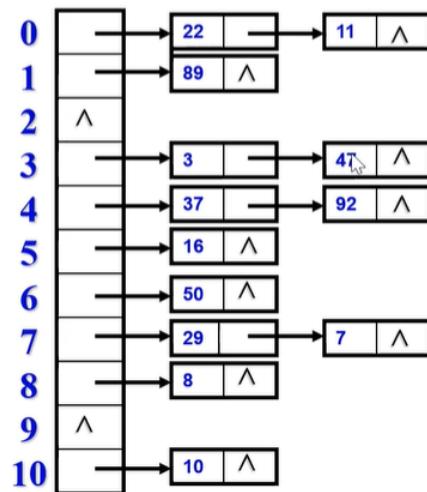
基本思想是：将具有相同哈希地址的记录链成一个单链表， m 个哈希地址，则有 m 个单链表，然后用一个数组将 m 个单链表的表头指针存储起来，形成一个动态的结构。

【例如】

设 {47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89} 的哈希函数为 $\text{Hash(key)} = \text{key mod } 11$ ，用链地址法处理冲突，建表如图所示：

$$ASL = (1*9 + 2*4) / 12$$

$$= 17/12$$



再哈希

再哈希法

方法：构造若干个哈希函数，当发生冲突时，计算下一个哈希地址，直到冲突不再发生。

即： $H_i = Rh_i(key)$ $i=1,2,\dots,k$

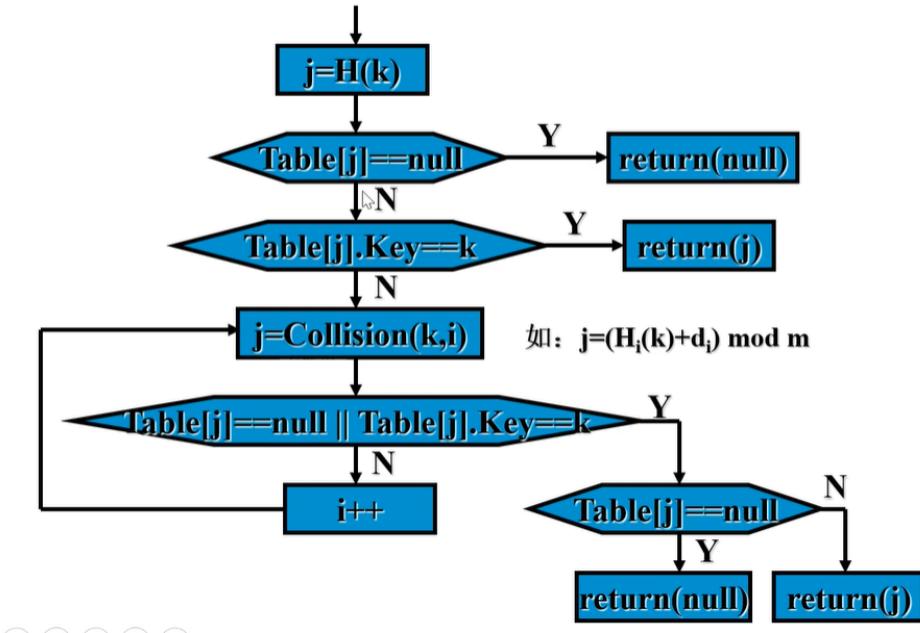
其中： Rh_i ——不同的哈希函数

特点：计算时间增加

Hash查找及分析

哈希表**查找过程**是：给定一个待查找的关键字 K ，根据哈希函数求出其哈希地址： $H(K)$ ，若表中该位置为空，则说明表中不存在 K ，查找不成功；若表中该位置有记录，则比较关键字，若相等，则查找成功，若不等，则说明有冲突发生，必须按照所设定的冲突处理方法查找下一个地址，直到关键字相等或者该位置的记录为空。

算法流程



对哈希表**查找效率**的量度，依然用平均查找长度来衡量。在查找过程中，关键字的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。而影响冲突多少的因素主要有以下三个：

1. 哈希函数是否均匀；
2. 处理冲突的方法；
3. 哈希表的**装填因子**

- 装填因子: Alpha

$$\alpha = \frac{\text{填入表中的元素个数}}{\text{哈希表的长度}}$$

α 是哈希表装满程度的标志因子。由于表长是定值， α 与“填入表中的元素个数”成正比，所以， α 越大，填入表中的元素较多，产生冲突的可能性就越大； α 越小，填入表中的元素较少，产生冲突的可能性就越小。

- 平均查找长度（统计结果）

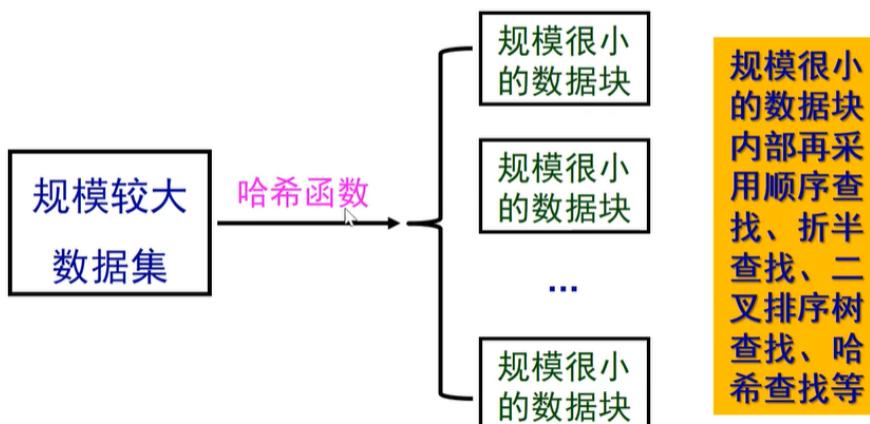
几种不同处理冲突方法的哈希查找算法的平均查找长度：

处理冲突方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl} \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$	$U_{nl} \approx \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
二次探测法	$S_{nr} \approx -\frac{1}{\alpha} \ln(1-\alpha)$	$U_{nr} \approx \frac{1}{1-\alpha}$
链地址法	$S_{nc} \approx 1 + \frac{\alpha}{2}$	$U_{nc} \approx \alpha + e^{-\alpha}$

哈希查找的平均查找长度是装填因子 α 的函数，与查找表长度几乎没关系（原因？）

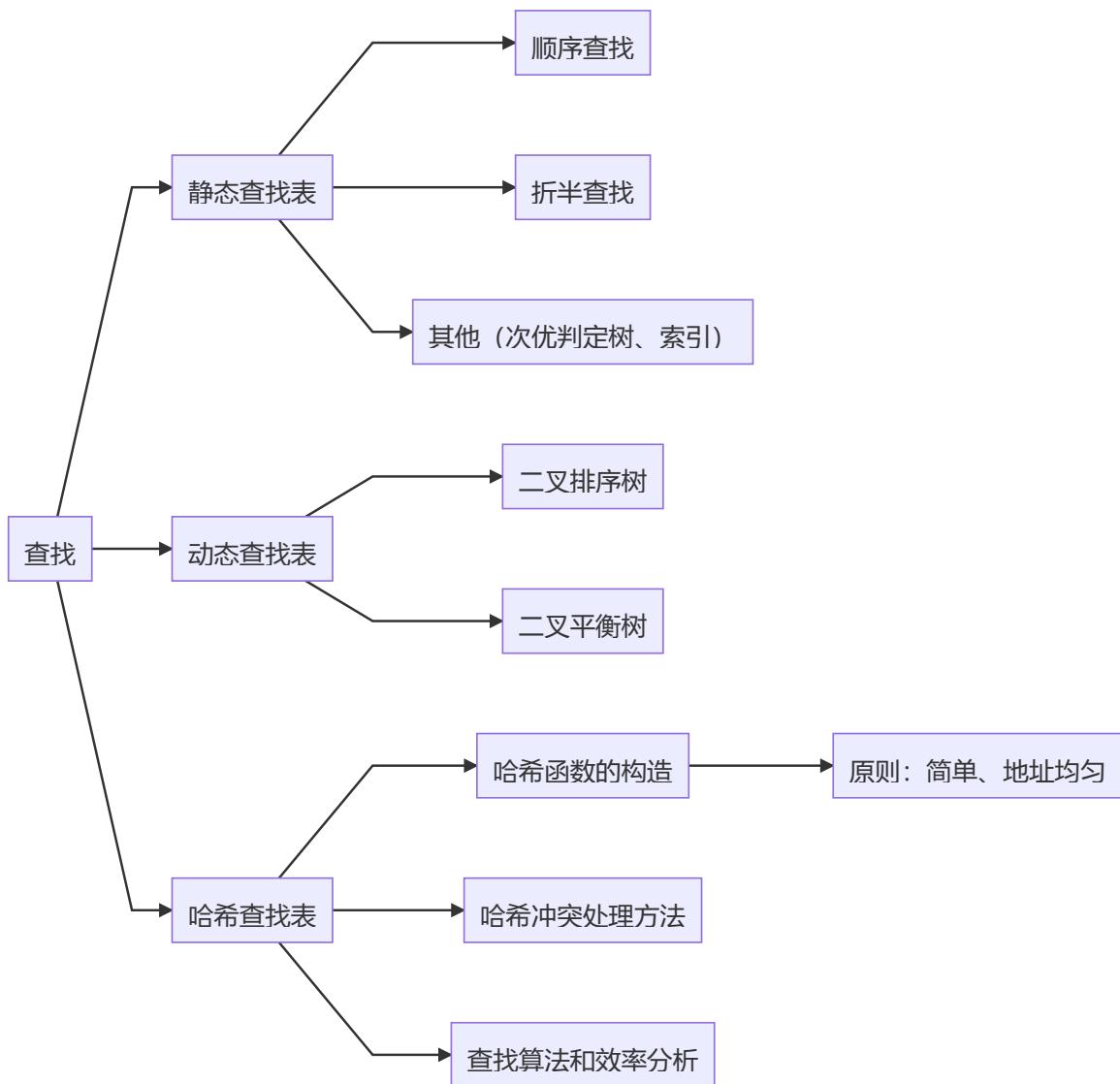
- 一般认为“开放定址法”平均查找长度高于“链接法”

Hash查实在实际工程中的大量使用



哈希查找：把规模大的查找转化为很多规模小的查找

本章小结



关于ASL的计算

- 顺序表查找

顺序表查找的平均查找长度为：

$$ASL_{ss} = \frac{1}{n} \sum_{i=1}^n (n - i + 1) = \frac{n+1}{2}$$

- 折半查找

则查找成功时折半查找的平均查找长度

$$ASL_{bs} = \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left[\sum_{j=1}^h j \times 2^{j-1} \right] = \frac{n+1}{n} \log_2(n+1) - 1$$

在 $n > 50$ 时，可得近似结果

$$ASL_{bs} \approx \log_2(n+1) - 1$$

- 分块查找

分块查找方法评价

$$ASL_{bs} = L_b + L_w$$

其中： L_b —— 查找索引表确定所在块的平均查找长度

L_w —— 在块中查找元素的平均查找长度

若将表长为 n 的表平均分成 b 块，每块含 s 个记录，并设表中每个记录的查找概率相等，则：

$$(1) \text{ 用顺序查找确定所在块: } ASL_{bs} = \frac{1}{b} \sum_{j=1}^b j + \frac{1}{s} \sum_{i=1}^s i \\ = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2} \left(\frac{n}{s} + s \right) + 1$$

$$(2) \text{ 用折半查找确定所在块: } ASL_{bs} \approx \log_2 \left(\frac{n}{s} + 1 \right) + \frac{s}{2}$$

- 算法比较：

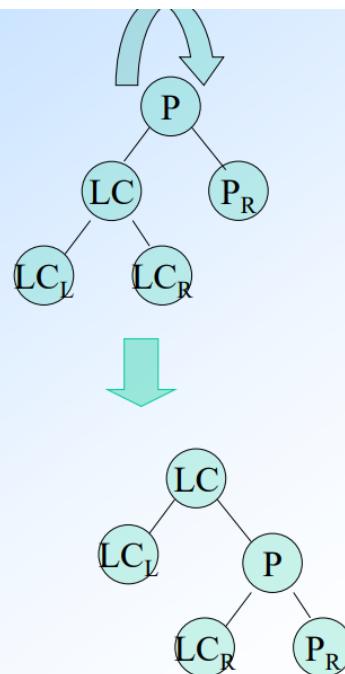
查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

二叉树的左旋与右旋

右旋：

```
Void R_Rotate(BSTree  
&p){  
    lc=p->lchild;  
    p->lchild=lc->rchild;  
    lc->rchild=p;  
    p=lc;  
}//R_Rotate
```



左旋：

```
void L_Rotate(BSTree  
&p){  
    rc=p->rchild;  
    p->rchild=rc->lchild;  
    rc->lchild=p;  
    p=rc;  
}//L_Rotate
```

