数据结构

北京邮电大学 网络空间安全学院
武 斌

上章内容

上一章（图）内容：

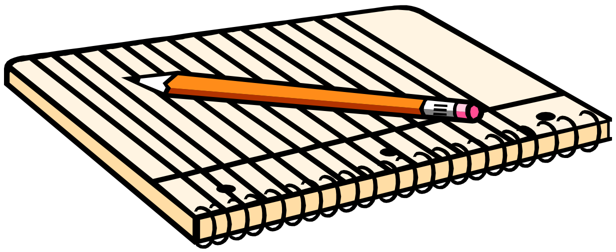
- 领会图的类型定义
- 熟悉图的各种存储结构及其构造算法，了解各种存储结构的特点及其选用原则
- 熟练掌握图的两种遍历算法
- 理解各种图的应用问题的算法

本次课程学习目标

学习完本次课程，您应该能够：

- 理解“查找表”的结构特点以及各种表示方法的适用性
- 熟练掌握以顺序表或有序表表示静态查找表时的查找方法
- 熟练掌握二叉查找树的构造和查找方法
- 熟练掌握哈希表的构造方法，深刻理解哈希表与其它结构的表的实质性的差别
- 掌握描述查找过程的判定树的构造方法，以及按定义计算各种查找方法在等概率情况下查找成功时的平均查找长度

本章课程内容（第九章 查找）



- 9.1 静态查找表
- 9.2 动态查找表
- 9.3 哈希表

- 查找表(Search Table)是由同一类型的数据元素(或记录)构成的集合。
- 对查找表经常进行的操作有：
  - (1) 查询某个“特定的”数据元素是否在表中；
  - (2) 检索某个“特定的”数据元素的各种属性；
  - (3) 在查找表中插入一个数据元素；
  - (4) 从查找表中删除某个数据元素。
- 在实际应用中的查找表通常可分两类：
  - 其中一类查找表在使用时主要作**前两种统称为“查找”的操作**，称此类查找表为**静态查找表**(**S**tatic **S**earch **T**able)。
  - 若在对查找表进行查找的过程中，**同时需要**随时**插入**当前查找表中不存在的数据元素，或者从当前的查找表中**删除**已存在的某个数据元素，则称此类查找表为**动态查找表**(**D**ynamic **S**earch **T**able)。

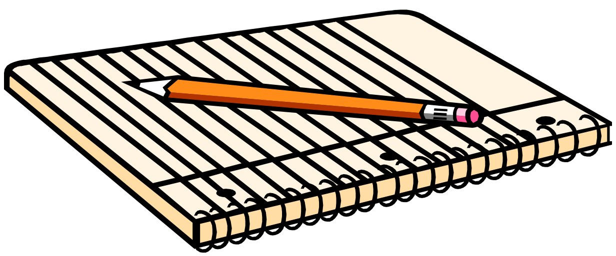
● 关键字相关概念

- 关键字(key)：是数据元素中某个数据项的值，用它可以标识（识别）一个数据元素。
- 主关键字(Primary key)：若此关键字可以唯一地标识一个元素 (对不同的元素，其主关键字均不同)。
- 次关键字(secondary key)：用以识别若干元素的关键字。
- 当数据元素只有一个数据项时，其关键字即为该数据元素的值。

- 查找相关概念：
  - 查找(searching)：根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素。
  - 若表中存在这样的一个元素，则称查找是成功的，此时查找的结果为给出整个数据元素的信息，或指示该数据元素在查找表中的位置；
  - 若表中不存在这样的元素，则称查找不成功，此时查找的结果可给出一个"**null**"元素(或空指针)。
- 举例

本章课程内容（第九章 查找）

- 9.1 静态查找表
- 9.2 动态查找表
- 9.3 哈希表



静态查找表

- 静态查找表的**抽象数据类型**定义如下：

  ➔ **ADT StaticSearchTable {**

  数据对象：D是具有相同特性的数据元素的集合。每个数据元素含有类型相同的关键字，可唯一标识数据元素。

  数据关系：D中所有数据元素同属一个集合。

  基本操作：

  CreateTable(&ST, n);

  操作结果：构造一个含 n 个数据元素的静态查找表 ST。

  DestroyTable(&ST);

  初始条件：静态查找表 ST 存在；

  操作结果：销毁查找表 ST。

静态查找表

Search(ST, key);

初始条件：静态查找表 ST 存在，key为和查找表中元素的关键字类型相同的给定值；

操作结果：若ST中存在其关键字等于key 的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。

Traverse(ST, Visit());

初始条件：静态查找表 ST 存在，Visit 是对元素操作的应用函数；

操作结果：按某种次序对ST的每个元素调用函数 visit() 一次且仅一次，一旦 visit() 失败，则操作失败。

} **ADT** StaticSearchTable

顺序表的查找

●一、顺序表的查找

以顺序表或线性链表表示静态查找表，则Search函数可用顺序查找来实现。本节中只讨论它在顺序存储结构模块中的实现。

➔ 顺序表的类型描述如下：

```
typedef struct {
        ElemType  *elem; // 数据元素存储空间基址
                  // 建表时按实际长度分配，0号单元留空
        int     length;        // 表长度
} SSTable;
```

顺序表的查找

- 顺序查找(Sequential Search)的查找过程为：
  - 从表中最后一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功，找到所查记录；
  - 反之，若直至第一个记录，其关键字和给定值比较都不等，则表明表中没有所查记录，查找不成功。
- 此查找过程可用算法9.1描述。

顺序表的查找

● 算法9.1

```
int Search_Seq (SSTable ST, KeyType key)
{
    // 在顺序表ST中顺序查找其关键字等于 key的数据元素，
    // 若找到，则函数值为该元素在表中的位置，否则为0
    ST.elem[0].key = key;                    // 设置“哨兵”
    for (i=ST.length; !EQ(ST.elem[i].key, key); --i); // 从后往前查找
    return i;                    // 找不到时，i 为0（“哨兵”的作用）
}  // Search_Seq
```

● 例如，在顺序表(21,37,88,19,92,05,64,56,80,75,13)中顺序查找64和60的过程如动画所示。演示(9-2-1)

顺序表的查找

- "监视哨"的作用：
  - "监视哨"控制住了循环变量 i 的出界，这样就免去查找过程中每一步都要检测整个表是否查找完毕，而这样的处理并不影响查找成功的情况。
  - 实践证明，当顺序查找在ST.length≥1000时，平均查找时间几乎减少一半。
- 如何评价查找算法的时间效率？
  - 由于查找算法中为确定其关键字等于给定值的数据元素的基本操作为"关键字和给定值相比"，因此通常以查找过程中关键字和给定值比较的平均次数作为比较查找算法的度量依据。

顺序表的查找

- 定义：查找过程中先后和给定值进行比较的关键字个数的期望值称作查找算法的平均查找长度(**A**verage **S**earch **L**ength)。
- 对于含有 n 个记录的查找表，查找成功时的平均查找长度为

$$ASL = \sum_{i=1}^{n} P_i C_i$$

  - P_i为查找表中第i个记录的概率，且 $\sum_{i=1}^{n} P_i = 1$；
  - C_i为找到表中其关键字与给定值相等的第i个记录时，和给定值已进行过比较的关键字个数。
  - 从顺序查找的过程可见，C_i取决于所查记录在表中的位置。如：查找表中的最后一个记录时，仅需比较一次；而查找表中第一个记录时，则需比较n次。一般情况下，C_i等于n-i+1。

顺序表的查找

- 假设n=ST.length，则顺序查找的平均查找长度为：

$$ASL = nP_1 + (n-1)P_2 + \ldots + 2P_{n-1} + P_n$$

- 假设每个记录的查找概率相等，即 $P_i = 1/n$

则在等概率情况下顺序查找的平均查找长度为

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

- 上式中的ASL_{ss}在$P_n \geq P_{n-1} \geq \ldots \geq P_2 \geq P_1$时达到极小值。
- 因此，对记录的查找概率不等的查找表若能预先得知每个记录的查找概率，则应**先对记录的查找概率进行排序**，使表中记录按查找概率由小至大重新排列，**以便提高查找效率**。

顺序表的查找

- 上述平均查找长度的前提：$\sum_{i=1}^{n} P_i = 1$

  在实际应用的大多数情况下，查找成功的可能性比不成功的可能性大得多。若考虑查找不成功的情况时，查找算法的平均查找长度应是**查找成功时的平均查找长度**与**查找不成功时的平均查找长度之和**。

- 对于顺序查找，不论给定值key为何值，查找不成功时和给定值进行比较的关键字个数均为n+1。

- 假设查找成功与不成功的可能性相同，对每个记录的查找概率也相等，则$P_i=1/(2n)$，此时顺序查找的平均查找长度为

$$ASL'_{ss} = \frac{1}{2n}\sum_{i=1}^{n}(n-i+1) + \frac{1}{2}(n+1) = \frac{3}{4}(n+1)$$

顺序表的查找

- 顺序查找优点
  - 算法简单
  - 适应面广
  - 对表的结构无任何要求
  - 无论记录是否按关键字有序均可应用
- 顺序查找缺点
  - 平均查找长度较大
  - 如果n很大时，查找效率较低

有序表的查找

- 二、有序表的查找

如果顺序表中的数据元素按关键字有序排列，即以有序表表示静态查找表时，则可进行“折半查找”。

- 折半查找(Binary Search)又称二分查找，其查找过程是，先确定待查记录所在范围（区间），然后逐步缩小范围直至找到该记录，或者当查找区间缩小到0也没有找到关键字等于给定值的记录为止。其算法描述如下页**算法9.2**所示。

有序表的查找

- 例如，在有序表(05,13,19,21,37,56,64,75,80,88,92)中进行折半查找的过程如动画所示。演示(9-2-2)
- 从例子可见，折半查找的过程为：给定值首先和处于待查区间“中间位置”的关键字进行比较，若相等，则查找成功，否则将查找区间缩小到“前半个区间” 或 “后半个区间” 之后继续进行查找。

有序表的查找

- 算法9.2

```
int Search_Bin ( SSTable ST, KeyType key )
{
    // 在有序表ST中折半查找其关键字等于key的数据元素，
    // 若找到，则函数值为该元素在表中的位置，否则为0。
    low = 1; high = ST.length;                 // 置区间初值
    while (low <= high) {
        mid = (low + high) / 2;
        if ( EQ(key , ST.elem[mid].key) )   return mid;        // 找到待查元素
        else  if ( LT(key , ST.elem[mid].key) )
            high = mid - 1;                    // 继续在前半区间内进行查找
        else
            low = mid + 1;                     // 继续在后半区间内进行查找
    } // while
    return 0;                                  // 有序表中不存在待查元素
} // Search_Bin
```

- 折半查找的性能分析

先看一个具体的情况，假设有序表的表长为11，则找到表中每个关键字时所需进行的给定值和关键字的比较次数如下表所列，假设表中各个关键字的查找概率相等，则在进行折半查找时的平均查找长度为：

i	1	2	3	4	5	6	7	8	9	10	11
C_i	3	4	2	3	4	1	3	4	2	3	4

$$ASL=\frac{1}{11}[1\times 1(次)+2\times 2(次)+4\times 3(次)+4\times 4(次)]=\frac{33}{11}=3$$

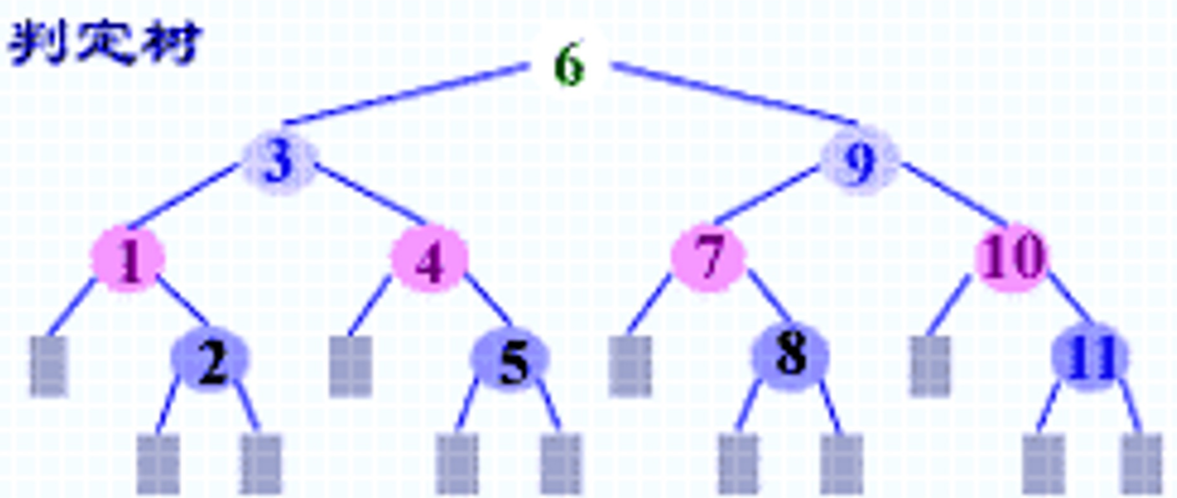
有序表的查找

- 判定树
  - 查找过程可以用二叉树描述：将只需经过n次比较即找到的关键字(序号)放在第n层，便可得到一棵二叉树
  - 描述了对有序表进行折半查找的过程。结点表示表的记录，结点的值表示位置
- 演示(9-2-3)

分析折半查找的平均查找长度

先看一个具体的情况，假设： n=11

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4



有序表的查找

- 在判定树上可以看到在折半查找的过程中，先后和给定值进行比较的关键字的位置，例如，找到表长为11的有序表中第5个关键字时，给定值先后和第6,3,4和5个关键字进行比较
- 折半查找有序表中任何一个关键字**"走了一条从根结点到该（关键字相应）结点的一条路径"**。
- 判定树中的**方形结点**表示**查找不成功**的情况，例如，当给定值的值介于有序表中第6个和第7个关键字之间时，在给定值先后和表中第6,9和7个关键字进行比较之后，查找区间缩小到0，从判定树看，落到了⑦的左子树的位置上。
- 通常称表示查找成功的**圆形结点为判定树的"内部结点"**，而称表示查找不成功的**方形结点为判定树的"外部结点"**。

有序表的查找

- 对表长为 n 的有序表进行折半查找的**判定树的深度**和含n个元素的**完全二叉树的深度相同**，也就是说，在长度为 n 的有序表中进行折半查找，所需进行给定值和关键字的比较次数至多为 $\lfloor \log_2 n \rfloor + 1$ 。

- 假设表长 **$n=2^h-1$**，则折半查找的判定树即为深度为h的满二叉树，树中第 j 层的结点数为 2^{j-1}，并且找到和这些结点位置相应的关键字的比较次数为 j，由此，在**进行等概率(折半)查找时查找成功时的平均查找长度**为：

$$ASL_{bs} = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{j=1}^{h} j \bullet 2^{j-1} = \frac{n+1}{n} \log_2(n+1) - 1$$

- 对任意的n，当n较大(n>50)时，可有下列近似结果

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

有序表的查找

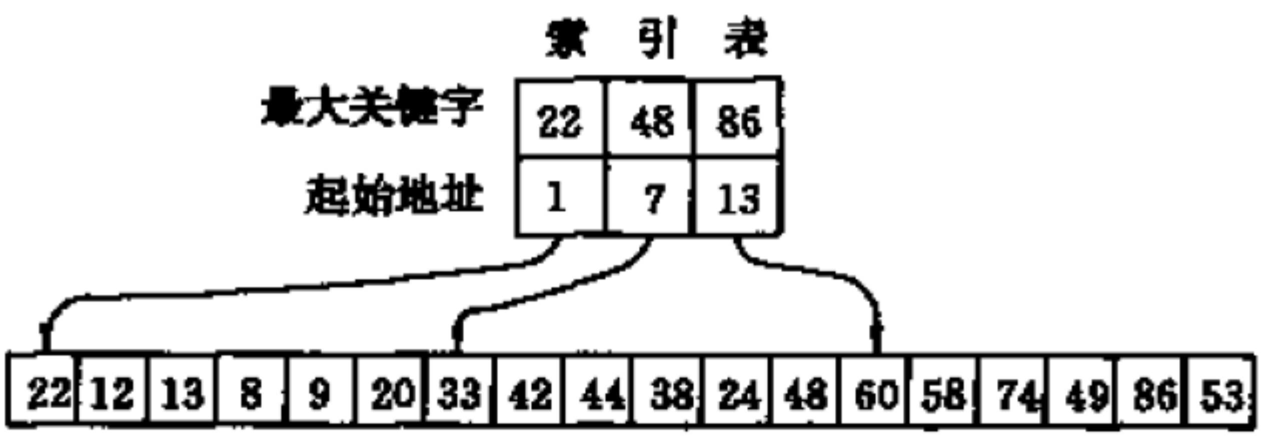
- 折半查找的优缺点
  - 优点：效率比顺序查找高
  - 缺点：只适用于顺序存储的有序表

索引顺序表

- 三、索引顺序表的查找
  - “**索引顺序查找**”又称“**分块查找**”，是顺序查找的一种改进方法
  - 建立“索引表”，对每个子表建立“**索引项**”；
  - “索引项”包括
    - **关键字项**：其值为该子表内的最大关键字
    - **指针项**：指示该子表的第一个记录在表中的位置
  - 索引表按关键字有序：表有序或分块有序（后面子表的所有关键字大于前面子表的最大关键字）
  - 由“分块有序表”和相应的“索引”构成一个“**索引顺序表**”，也是静态查找表的一种实现方法。演示 (9-2-4)

索引顺序表

- 查找过程：首先在索引表中进行折半或顺序查找，以确定待查关键字在分块有序表中所在“块”，块中记录是任意排列的，在“块”中进行顺序查找。
- 例：P225



三个子表$(R_1, R_2, \cdots, R_6)$、$(R_7, R_8, \cdots, R_{12})$、$(R_{13}, R_{14}, \cdots R_{18})$

索引顺序表

- 由此，在索引顺序表中进行**索引顺序查找的平均查找长度**为**查找索引表确定所在块的平均查找长度L_b**和**在块中查找元素的平均查找长度 L_w**之和。

- 假设顺序表的表长为n，并均匀地分成b块，设每块长度为s，即 $b=\lceil n/s \rceil$

则在**等概率查找并顺序查找索引**的情况下，若使用顺序查找，索引顺序查找的平均查找长度为

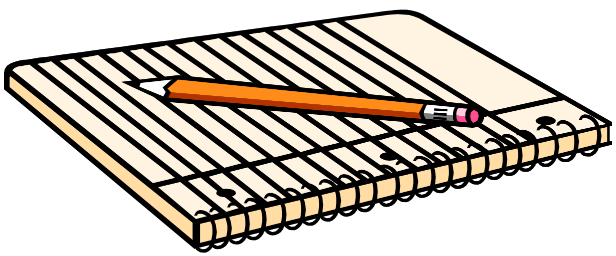
$$ASL_{bs}=L_b+L_w=\frac{1}{b}\sum_{j=1}^{b}j+\frac{1}{s}\sum_{i=1}^{s}i=\frac{b+1}{2}+\frac{s+1}{2}=\frac{1}{2}(\frac{n}{s}+s)+1$$

- 容易证明，当s取 $\sqrt{n}$时，ASL$_{bs}$取最小值 $+1\sqrt{n}$ 这个值比顺序查找有了很大改进，但远不及折半查找。

- 若用折半查找确定所在块，则分块查找的平均长度为

$$ASL'_{bs}\cong\log_2(\frac{n}{s}+1)+\frac{s}{2}$$

本章课程内容（第九章 查找）



- 9.1 静态查找表
- 9.2 动态查找表
- 9.3 哈希表

动态查找表

- 动态查找表的**特点：**
  - 表结构本身是在查找过程中**动态生成**的
  - 对于给定值key，若表中**存在**其关键字等于key的记录，则**查找成功返回**，否则**插入**关键字等于key的记录。
- 动态查找表的抽象数据类型定义如下：
  - **ADT DynamicSearchTable {**

数据对象D：D是具有相同特性的数据元素的集合。每个数据元素含有类型相同的关键字，可唯一标识数据元素。

数据关系R：数据元素同属一个集合。

基本操作P：

InitDSTable(&DT);

操作结果：构造一个空的动态查找表 DT。

动态查找表

DestroyDSTable(&DT);

初始条件：动态查找表DT存在；

操作结果：销毁动态查找表 DT。

SearchDSTable(DT, key);

初始条件：动态查找表DT存在，key为和关键字类型相同的给定值；

操作结果：若DT中存在其关键字等于key的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。

InsertDSTable(&DT, e);

初始条件：动态查找表DT存在，e 为待插入的数据元素；

操作结果：若DT中不存在其关键字等于e.key的数据元素，则插入e 到DT。

DeleteDSTable(&T, key);

初始条件：动态查找表DT存在，key为和关键字类型相同的给定值；

操作结果：若DT中存在其关键字等于key的数据元素，则删除之。

TraverseDSTable(DT, Visit());

初始条件：动态查找表DT存在，Visit 是对结点操作的应用函数；

操作结果：按某种次序对DT的每个结点调用函数visit()一次且至多一次。一旦 visit() 失败，则操作失败。

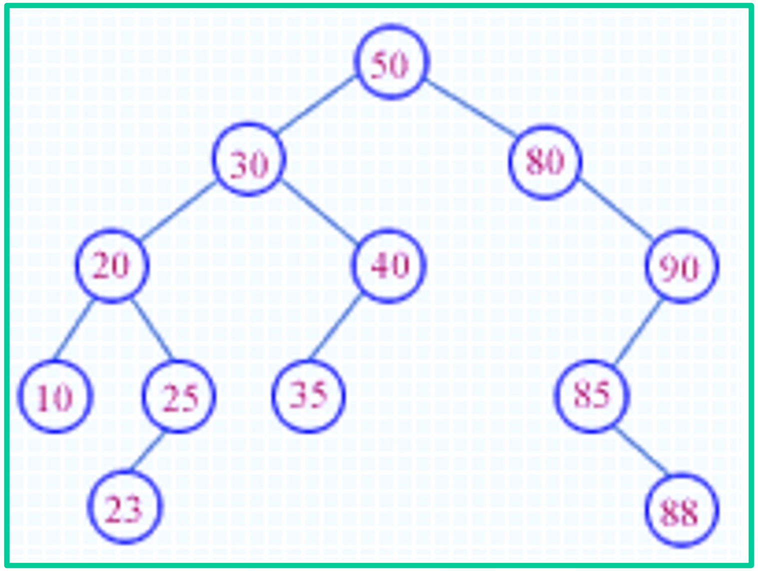
} ADT DynamicSearchTable

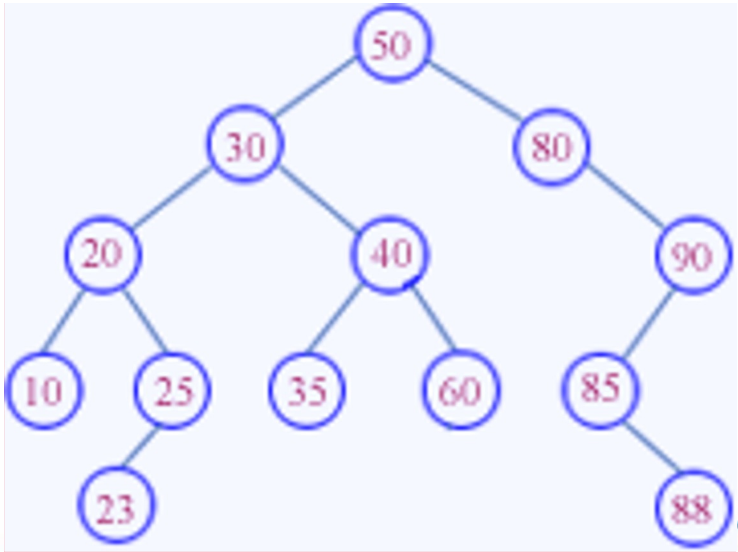
二叉排序树

一、二叉排序树

二叉排序树(Binary Sort Tree)，又称二叉查找树或者是一棵空树，或者是具有如下特性的二叉树：

- (1)若它的左子树不空，则左子树上所有结点的值均小于根结点的值；
- (2)若它的右子树不空，则右子树上所有结点的值均大于根结点的值；
- (3)它的左、右子树也都分别是二叉排序树。
  - 例如，左下图所示是一棵二叉排序树。 右图呢？如何判定？中序





二叉排序树

- 若二叉查找树为空，则查找不成功；否则
  - 1)若给定值等于根结点的关键字，则查找成功；
  - 2)若给定值小于根结点的关键字，则继续在左子树上进行查找；
  - 3)若给定值大于根结点的关键字，则继续在右子树上进行查找。
  - 演示(9-3-1)
- 从上例演示可见，在二叉查找树中进行查找的过程为：**从根结点出发，沿着左分支或右分支递归进行查询直至关键字等于给定值的结点**；或者**从根结点出发，沿着左分支或右分支递归进行查询直至子树为空树止**。前者为查找成功的情况，后者为查找不成功的情况。

二叉排序树

- 算法9.5(a)

```
BiTree SearchBST(BiTree T, KeyType key) {
    // 在根指针T所指二叉排序树中递归查找某关键字等于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

- 二叉排序树的插入和删除
  - 二叉排序树是一种动态树。其**特点**是，树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。
  - 新插入的结点一定是一个**新添加的叶子结点**，并且是查找不成功时查找路径上访问的**最后一个节点的左孩子或右孩子结点**。
  - 为此，需将前页二叉排序树的查找算法9.5(a)改写成算法9.5(b)，以便能在查找不成功时返回插入位置。

二叉排序树

● 算法9.5(b)

```
Status SearchBST (BSTree T, KeyType key, BSTree f, BSTree &p)
{   // 根指针T所指二叉查找树中递归查找关键字等于key的数据元素,若查找成功,
  // 则指针p指向该数据元素结点,并返回TRUE,否则指针p指向查找路径上访
  // 问的最后一个结点并返回FALSE,指针f指向T的双亲,其初始调用值为NULL
    if (!T) { p = f; return FALSE; }                    // 查找不成功
    else if  EQ( key, T->data.key )  { p = T; return TRUE; }      // 查找成功
    else if  LT( key, T->data.key )
           return SearchBST (T->lchild, key, T, p );          // 在左子树中继续查找
    else return SearchBST (T->rchild, key, T, p );      // 在右子树中继续查找
} // SearchBST
```

注意：若查找成功，则**p指向该关键字所在结点**；若查找不成功，则**p应该指向查找路径上最后一个结点。**

二叉排序树

● 算法9.6

```
Status InsertBST(BiTree &T, ElemType e )
{     // 当二叉查找树T中不存在关键字等于 e.key 的数据元素时，
      // 插入 e 并返回 TRUE，否则返回 FALSE
      if (!SearchBST ( T, e.key, NULL, p ) {     // 查找不成功
         s = (BiTree)malloc(szeof(BiTNode));
         s->data = e; s->lchild = s->rchild = NULL;
          if ( !p ) T = s;                        // 插入 *s 为新的根结点
         else if  LT( e.key, p->data.key )   p->lchild = s;// 插入 *s 为 *p 的左孩子
         else p->rchild = s;                      // 插入 *s 为 *p 的右孩子
         return TRUE;
      } // if
      else return FALSE;          // 树中已有关键字相同的结点，不再插入
} // InsertBST
```

二叉排序树

- 对于动态查找表，在**查找不成功时尚需进行插入**，即当二叉查找树中不存在其关键字等于给定值的结点时，需插入一个关键字等于给定值的数据元素。
- 实际上，二叉查找树结构本身正是从空树开始逐个插入生成的。插入的原则为：若二叉查找树为**空树**，则插入的结点为**新的根结点**；否则，插入的结点必为一个**新的叶子结点，其插入位置**由查找过程确定。
- 例如，若给定值序列为 { 50,30,40,80,20,36,90,40,38 }，从空树起，逐个插入后构成的二叉查找树如动画所示。 演示(9-3-3)

二叉排序树

- **在二叉排序树上删除一个结点之后应该仍是一棵二叉树，并保持其二叉查找树的特性**。
- 在二叉排序树上如何删除一个结点，即如何修改结点的指针？**设被删节点指针*p，双亲*f，假设*p为*f的左孩子**；分三种情况讨论：
  - (1) *p为“叶子”，仅需修改其双亲结点的相应指针即可；
  - (2) *p只有左子树P_L或右子树P_R，则只需保持该结点的子树和其双亲之间原有的关系即可，因此只需要将其P_L或P_R直接链接到其双亲结点成为为其双亲的子树即可；
  - (3) P_L和P_R均不空。两种方法
    - 令*p的左子树为*f的左子树，*p的右子树成为*s的右子树（*s是*p的前驱）
    - 令*p的直接前驱（或直接后继）替代*p，然后从二叉排序树上删去它的直接前驱（或直接后继）
- 演示(9-3-4)

二叉排序树

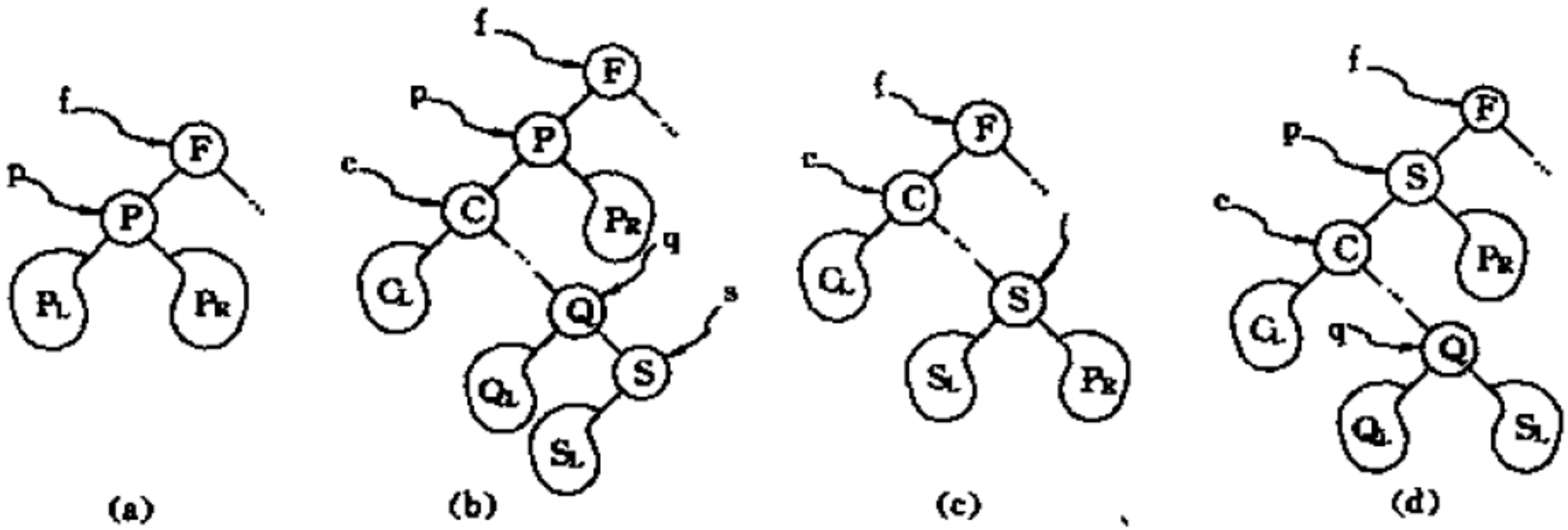


图 9.9 在二叉排序树中删除 * p

(a) 以 * f 为根的子树;(b) 删除 * p 之前;(c) 删除 * p 之后,以 P_R 作为 * s 的右子树的情形;
(d) 删除 * p 之后,以 * s 替代 * p 的情形。

二叉排序树

● 算法 9.7

```
Status DeleteBST (BiTree &T, KeyType key ) {
    // 若二叉查找树T中存在关键字等于key的数据元素时，则删除
    // 该数据元素结点 *p，并返回 TRUE；否则返回 FALSE
    if (!T) return FALSE;            // 不存在关键字等于key的数据元素
    else {
      if (EQ( key, T->data.key) )    { return Delete (T); }
                                     // 找到关键字等于key的数据元素
      else if (LT( key, T->data.key))    return DeleteBST ( T->lchild, key );
      else return DeleteBST ( T->rchild, key );
    } // else
} // DeleteBST
```

● 其中删除操作过程如算法9.8所描述。

二叉排序树

- **算法9.8**

```
Status Delete ( BiTree &p ) {
    // 从二叉查找树中删除结点 p，并重接它的左或右子树
    if (!p->rchild) {                    // 右子树空则只需重接它的左子树
        q = p; p = p->lchild; free(q);
    }
    else if (!p->lchild) {               // 只需重接它的右子树
        q = p; p = p->rchild; free(q);
    }
    else {                               // 左右子树均不空
        q = p; s = p->lchild;
        while (s->rchild) { q = s; s = s->rchild; }   // 转左，然后向右到尽头
        p->data = s->data;               // s 指向被删结点的前驱
        if (q != p ) q->rchild = s->lchild;   // 重接 *q 的右子树
        else q->lchild = s->lchild;      // 重接 *q 的左子树
        delete s;
    }
} // Delete
```

二叉排序树

- 对算法9.7中的删除操作 Delete (T) 需作三点说明：
  - (1) 叶子结点的情况可归入到“右子树空”的情况，此时因左子树也空，自然 p=NULL；
  - (2) 注意，T在函数 DeleteBST 中是一个递归调用的引用型参数，第一次调用时的参数是指向根结点的指针，当继续在子树中进行查找时，自然就是双亲结点的左指针或右指针，因此在函数 Delete(p) 中修改指针 p，实际上修改的是被删结点的双亲结点的指针域；

    演示(9-3-5)
  - (3) 在被删结点的左右子树均不空时，需删除其“前驱结点” *s，一般情况下应将 *s 的左子树接为其双亲的右子树，但当 *s 即为被删结点的左子树根时，应将 *s 的左子树接为其双亲的左子树。

    演示(9-3-6)

二叉排序树

●二叉排序树的查找分析

从二叉查找树的查找过程可见，二叉查找树的查找性能取决于它的深度，然而由于二叉查找树是在查找过程中逐个插入构成，因此它的深度取决于关键字先后插入的次序。

- 例如，含关键字1,2,3,4,5的二叉查找树，其深度可能为3或4或5，若按1,2,3,4,5的次序得到的二叉查找树的深度为5，若按3,1,2,4,5的次序得到的二叉查找树的深度则为3，如动画所示。演示(9-3-7)
- 最差：插入的关键字有序，结果为单枝树，树的深度为n，平均查找长度为(n+1)/2；
- 最好：二叉排序树的形态和折半查找的判定树相同，平均查找长度和$\log_2 n$成正比；
- 二叉排序树查找的平均性能如何呢？

二叉排序树

$$ASL_{(a)} = \frac{1}{6}[1+2+2+3+3+3] = 14/6$$

$$ASL_{(b)} = \frac{1}{6}[1+2+3+4+5+6] = 21/6$$

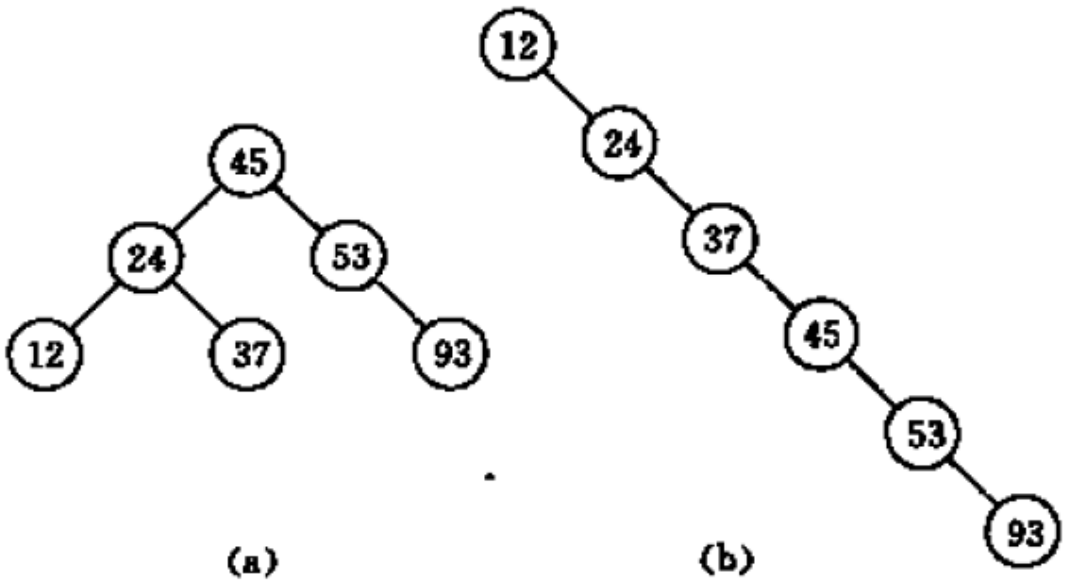


图 9.10 不同形态的二叉查找树

(a) 关键字序列为(45,24,53,12,37,93)的二叉排序树；(b) 关键字序列为(12,24,37,45,53,93)的单支树。

二叉排序树

- 假设在含有n (n≥1)个关键字的序列中，**i个关键字小于第一个关键字**，**n-i-1个关键字大于第一个关键字**，则由此构造而得的二叉排序树在n个记录的查找概率相等的情况下，其平均长度为

$$P(n,i)=\frac{1}{n}[1+i*(P(i)+1)+(n-i-1)(P(n-i-1)+1)]$$

  P(i)为含有i个节点的二叉排序树的平均查找长度

- 同时假设生成二叉排序树是一个“随机”序列(即i取0至 n-1中任一值的可能性相同)，则

$$P(n)=\frac{1}{n}\sum_{i=0}^{n-1}P(n,i)=1+\frac{1}{n^2}\sum_{i=0}^{n-1}[iP(i)+(n-i-1)P(n-i-1)]$$

  括号中两项对称，且P(0)=0，P(1)=1。

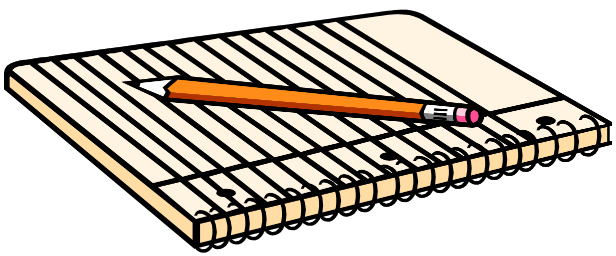
- 上式可改写为

二叉排序树

$$P(n) = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} iP(i)(n \geq 2)$$

- 这是一个递归方程，可求得其解为：$P(n) \leq 2(1+\frac{1}{n})\ln n(n \geq 2)$
- 由此可见，在随机的情况下，二叉查找树的查找性能是和 logn 等数量级的。

本章课程内容（第九章 查找）



- 9.1 静态查找表
- 9.2 动态查找表
- 9.3 哈希表

哈希表

- 上两节讨论的结构平均查找长度不仅不可能为0，并且都随n(关键字集合的大小)的增长而增长。**因为数据元素在结构中的位置都是随机的，和它的关键字无关**。在这样的结构中进行查找的主要操作就是**将给定值和表中关键字进行依次比较，其查找效率取决于比较次数**。
- 在记录的**存储位置**和它的**关键字**之间**建立一个确定的对应关系**f，使每个关键字和结构中一个惟一的存储位置相对应。因而在查找时，只要根据这个对应关系f找到给定值K的像f(K)。若结构中存在关键字和K相等的记录，则必定在f(K)的存储位置上。在此，我们称这个对应关系f为**哈希(Hash)函数**，按这个思想建立的表为**哈希表**。

哈希表

- 例如，对于关键字序列{Zhao,Qian,Sun,Li,Wu,Chen,Han,Ye,Dai}，可设关键字的哈希函数如下：

$$f(key) = \lfloor (Ord(CH1) - Ord('A') + 1)/2 \rfloor$$

  - 其中，CH1 表示关键字中第1个字母，Ord 为字符的次序函数。则以表长为14的顺序表表示的查找表如右所示。
  - 显然，此查找表的平均查找长度为 0，若给定值为 Qian，由上述关键字函数得到函数值为 8，即可从 “地址为8”的表中取得该记录。但这样的函数并不容易设计，如果同时存在关键字Zhou，则上述函数不可取，换句话说，为使平均查找长度等于0，必须找到一个能将查找表中所有关键字都 “散列” 在表中不同位置的哈希函数。

0	
1	Chen
2	Dai
3	
4	Han
5	
6	Li
7	
8	Qian
9	Sun
10	
11	Wu
12	Ye
13	Zhao

哈希表

- 若关键字不同而函数值相同，则称这两个关键字（对于该哈希函数而言）为“同义词”，并称这种现象为“冲突”。对于动态查找表很难找到不存在同义词的哈希函数，唯一弥补的办法是，在发生冲突时设法解决。
- 定义哈希表：根据设定的哈希函数 H(key)和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集(区间)上，并以关键字在地址集中的“像”作为相应记录在表中的存储位置，这种表被称为哈希表，这一映象的过程称为哈希造表或者“散列”，所得存储位置称哈希地址或散列地址。

哈希表

一、哈希函数的构造方法

若对于关键字集合中的任意一个关键字，经哈希函数映象到地址集合中任何一个地址的概率相等，则称此类哈希函数为**均匀的(Uniform)哈希函数**。换言之，就是使关键字经过哈希函数得到一个“随机的地址”，以便使一组关键字的哈希地址均匀分布在整个地址区间中，从而减少冲突。

1、直接地址法

取关键字本身或关键字的某个线性函数值作为哈希表的地址。

即 **Hash(key)=key** 或 **Hash(key)=a×key+b** （a 和 b 均为常数）

例（P253）

直接定址所得地址集的大小和关键字集的大小相同，关键字和地址一一对应，决不会产生冲突。但实际应用中能采用直接定址的情况极少。

哈希表

2、数字分析法

如果可能出现的关键字的数位相同，且取值事先知道，则可对关键字进行分析，取**其中“分布均匀”的若干位或它们的组合作为哈希表的地址**。

- 例：80个记录，关键字为8位十进制数。哈希表的表长为100_{10}，可取两位十进制数组成哈希地址。那么，取哪两位呢？
- 原则：使哈希地址尽量避免产生冲突。
- 分析：第①②位都是“8 1”，第③位只可能取3或4，第⑧位只可能取2、5或7，因此这4位都不可取。由于中间的4位可看成是近乎随机的，因此可取其中任意两位，或取其中两位与另外两位的叠加求和后舍去进位作为哈希地址。

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

哈希表

3、平方取中法

如果关键字的所有各位分布都不均匀，则**可取关键字的平方值的中间若干位作为哈希表的地址**。由于一个数的平方值的中间几位数受该数所有位影响，将使随机分布的关键字得到的哈希函数值也随机。

- 例如：下列八进制数的关键字及其哈希地址

关键字	(关键字)^2	哈希地址
0100	0 010000	010
1100	1 210000	210
1200	1 440000	440
1160	1 370400	370
2061	4 310541	310
2062	4 314704	314
2161	4 734741	734
2162	4 741304	741
2163	4 745651	745
	…	

哈希表

➔ **4、折叠法**

若关键字的位数很多，且每一位上数字分布大致均匀，则可采用**移位叠加**或**间界叠加**，即将关键字分成若干部分，然后以它们的**叠加和(舍去进位)作为哈希地址**。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是从一端向另一端沿分割界来回折叠，然后对齐相加。

- 例如，key = 110108780428895，则其移位叠加得到的哈希地址为321，间界叠加得到的哈希地址为410。(哈希表的表长为1000)

移位相加

```
   895
   428
   780
   108
+) 110
------
  2321
```

间界叠加

```
   895
   824
   780
   801
+) 110
------
  3410
```

哈希表

5、除留余数法

取关键字被某个不大于哈希表长m的数p除后所得余数作为哈希地址。即

Hash(key) = key MOD p，p≤m

其中，**MOD**表示“取模”运算，**p 为不大于表长的素数或不包含小于20的质因数的合数**。

- 若p为含质因子c 的合数，则将使所有含质因子c 的关键字映象到“c 的倍数”的地址上，从而增加了冲突的可能性。
- 例如，假设哈希表长为10，取p=9，则关键字序列{12,39,24,36,81,78,60} 对应的哈希地址依次为：3,3,6,0,0,6,6。显然，H(key)=key%9 对这组关键字不是一个“好”的哈希函数。

6、随机数法

当关键字不等长时，可取关键字的某个伪随机函数值作为哈希地址。

Hash(key) = random(key)

哈希表

- 对于非数值型关键字，则需先将它们转化为数字。实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是**使产生冲突的可能性降到尽可能地小**。
- 构造哈希函数考虑的因素主要有：
  - 计算哈希函数所需时间
  - 关键字的长度
  - 哈希表的大小
  - 关键字的分布情况
  - 记录的查找频率

哈希表

- 二、处理冲突的方法

  为产生冲突的地址寻找下一个哈希地址

➔ 1、开放定址法

开放定址处理冲突的办法是，**设法为发生冲突的关键字“找到”哈希表中另一个尚未被记录占用的位置**。

$$H_i = (H(key) + d_i)\mathrm{MOD}m \quad i=1,2,\ldots,k\ (k \le m-1)$$

上式的含义是，已知哈希表的表长为 m（即哈希表中可用地址为：0～m-1），若对于某个关键字 key，哈希表中地址为 Hash(key) 的位置已被占用，则为该关键字试探 “下一个”地址H_1=(Hash(key)+d_1) MOD m，若也已被占用，则试探再“下一个”地址H_2=(Hash(key)+ d_2) MOD m，…，依次类推直至找到一个地址H_k=(Hash(key)+ d_k) MOD m 未被占用为止。即H_i是为解决冲突生成的一个地址序列，其值取决于设定“增量序列d_i”。对于d_i通常可有三种设定方法：

哈希表

- 1）d_i= 1,2,3,…,m-1，称这种处理冲突的方法为“**线性探测再散列**”。

例如，假设关键字序列为{ 19,56,23,14,68,82,70,36,91 }，设哈希表的表长为11，哈希函数为 Hash(key)=key % 11。当插入关键字23(Hash(23)=1)时，出现冲突现象，取增量d_1=1，求得处理冲突后的哈希地址为1+1=2；又如，在插入关键字36(Hash(36)=3)时，因哈希表中地址为 3,4,5 和 6 的位置均已存放记录，因此取增量 d_4=4，即处理冲突后的哈希地址为(3+4) =7。

按线性探测再散列处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
	56	23	14	68	82	70	36	19	91	

哈希表

- 2）$d_i = 1^2，-1^2，2^2，-2^2，…，\pm k^2(k\le m/2)$，称这种处理冲突的方法为"**二次探测再散列**"。

例如，假设关键字序列为{ 19,56,23,14,68,82,70,36,91 }，设哈希表的表长为11，哈希函数为 Hash(key)=key % 11。对于关键字68（Hash(68)=2），因表中地址为 2,3 和 1 的位置均已填入记录，因此取增量$d_3=2^2$，即处理冲突后的哈希地址为(2+4) =6。

按平方探测再散列处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
	56	23	14	70	82	68	36	19		91

哈希表

➢ 3） d_i 为伪随机数列或者d_i =i ×H_2(key)，（H_2(key)为关键字的另一个哈希函数），称这种处理冲突的方法为“**伪随机探测再散列**”或“**双散列函数探测再散列**”。

例如，假设关键字序列为{ 19,56,23,14,68,82,70,36,91 }，设哈希表的表长为11，哈希函数为 Hash(key)=key % 11。按伪随机探测再散列构建的哈希表的增量 d_i = (3key)% 10+1 。

按伪随机探测再散列处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
23	56	68	14	70	82		91	19		36

- 上述三种散列处理构造哈希表的步骤见动画所示。演示（9-4-1）
- 注意，开放定址法中的 d_i 应具有“完备性”，即(1)增量序列中的各个d_i值均不相同；(2)由此得到的m-1个地址值必能覆盖哈希表中所有地址。

哈希表

2、再哈希法

H_i=RH_i(key) i=1,2,…,h

RH_i均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算的时间。

3、链地址法

将所有关键字为“同义词”的记录链接在一个线性链表中。此时的哈希表以“指针数组”的形式出现，数组内各个分量存储相应哈希地址的链表的头指针。

例如，假设关键字序列为{ 19，56，23，14，68，82，70，36，91 }，哈希表的表长为 7，哈希函数为 Hash(key)=key % 7，则构建的以链地址处理冲突的哈希表如动画所示。 演示（9-4-2）

4、建立一个公共溢出区

设立一个基本表HashTable[0..m-1]和一个溢出表OverTable[0..v]，所有关键字和基本表中关键字为同义词的记录，一旦冲突，都填入溢出表。

哈希表

- 开放定址的哈希表的查找和插入
  - 给定K值，根据哈希函数求得哈希地址，若表中该位置上没有记录，则查找不成功；
  - 否则，比较关键字，若关键字和给定值相等，则查找成功；
  - 若不等，则依据建表时设定的处理冲突的方法寻找“下一个”地址，直至哈希表的某个位置为“空”或者表中所填记录的关键字等于给定值为止（该地址恰为新的记录的插入位置）。

哈希表

● 开放定址哈希表的存储结构

```
int hashsize[ ] = { 997, ... };     // 哈希表容量递增表，一个合适的素数序列
typedef struct
{
    ElemType *elem; // 数据元素存储基址，动态分配数组
    int count;        // 当前表中含有的记录个数
    int sizeindex;     // hashsize[sizeindex]为当前哈希表的容量
} HashTable;
#define SUCCESS = 1
#define UNSUCCESS = 0
#define DUPLICATE = -1
```

哈希表

- 算法9.8

```
Status SearchHash (HashTable H, KeyType K, int &p, int &c)
{   // 在开放定址哈希表H中查找关键码为K的元素,若查找成功,以p指
  // 示待查记录在表中位置,并返回SUCCESS;否则,以p指示插入位置,并
  // 返回UNSUCCESS, c 用以计冲突次数,其初值置零,供建表插入时参考
    p = Hash(K);                              // 求得哈希地址
    while ( H.elem[p].key != NULLKEY          // 该位置中填有记录
            && !EQ( H.elem[p].key , K) )      // 并且关键字不相等
      collision(p, ++c);                      // 求得下一探查地址p
    if EQ( H.elem[p].key, K )
      return SUCCESS;                   // 查找成功，p 返回待查记录位置
    else return UNSUCCESS;
      // 查找不成功(H.elem[p].key == NULLKEY)，p返回的是插入位置
  } // SearchHash
```

哈希表

- 算法9.9

```
Status InsertHash (HashTable &H, Elemtype e)
{   // 若开放定址哈希表H中不存在记录 e 时则进行插入，并返回OK；
    // 若在查找过程中发现冲突次数过大，则需重建哈希表
    c=0;
    if (SearchHash ( H, e.key, p, c ) == SUCCESS )
       return DUPLICATE;          // 表中已有与 e 有相同关键字的记录
    else if ( c < hashsize[H.sizeindex]/2 ) {
                                  // 冲突次数c未达到上限，（阀值c可调）
       H.elem[p] = e; ++H.count;   return OK;        // 插入记录 e
    } // if
     else {RecreateHashTable(H);   return UNSUCCESS;}// 重建哈希表
} // InsertHash
```

哈希表

● 哈希表的性能分析

以哈希表表示动态查找表原**希望其平均查找长度为0**，但由于构建哈希表时**不可避免会产生冲突**，因此在哈希表上进行查找还是需要通过“比较”来确定查找是否成功，因此仍然存在平均查找长度的问题。那么**哈希表的平均查找长度是多少？**先看一下前面构建好的哈希表的例子。演示（9-4-3）

按线性探测再散列处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
	56	23	14	68	82	70	36	19	91	

按平方探测再散列处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
	56	23	14	70	82	68	36	19		91

按双散列函数探测处理冲突构造的哈希表为

0	1	2	3	4	5	6	7	8	9	10
23	56	68	14	70	82		91	19		36

哈希表

- 假设查找是等概率的，即p_i=1/9，则
  - 按线性探测再散列方法处理冲突构建的哈希表的平均查找长度为

$$ASL_1=\frac{1}{9}\sum C_i=\frac{1}{9}(1+2+1+3+1+3+5+1+7)=\frac{24}{9}$$

  - 按平方探测再散列方法处理冲突构建的哈希表的平均查找长度为

$$ASL_2=\frac{1}{9}\sum C_i=\frac{1}{9}(1+2+1+1+1+4+4+1+5)=\frac{20}{9}$$

  - 按伪随机探测再散列方法处理冲突构建的哈希表的平均查找长度为

$$ASL_3=\frac{1}{9}\sum C_i=\frac{1}{9}(2+1+1+1+1+1+2+1+3)=\frac{13}{9}$$

  - 按链地址方法处理冲突构建的哈希表的平均查找长度为

$$ASL_4=\frac{1}{9}\sum C_i=\frac{1}{9}(1\times 4+2\times 2+3\times 2+4)=\frac{18}{9}$$

哈希表

- 从例子可以得出下列结论：
  - 1)在哈希函数相同的情况下，**处理冲突的方法不同**，所得哈希表的**平均查找长度也不同**。
  - 2)**线性探测再散列处理冲突容易造成记录的"二次聚集"**，即使得本不是同义词的关键字又产生新的冲突；
  - 3)对**开放定址处理冲突的哈希表而言，表长必须≥记录数**，并且由于表中已填入的记录越多，继续插入记录发生冲突的可能性就越大，因此可以设想这样的哈希表不应该使"表长=记录数"。而链地址处理冲突的哈希表不会出现这种情况，它的平均查找长度主要取决于哈希函数本身。设想若表长仍取11，哈希函数和开放定址的一样，则链地址处理冲突的哈希表的平均查找长度为18/9。

哈希表

- 一般情况下，在哈希函数为“均匀”的前提下，哈希表的**平均查找长度仅取决于处理冲突的方法和哈希表的装填因子**。
  - ➔ 哈希表的**装填因子**定义为：

$$\alpha = \frac{\text{表中记录数}}{\text{哈希表的表长}}$$

  - ➔ 在等概率查找的情况下，可以证明：
    - ➢ **线性探测再散列**的哈希表查找成功的**平均查找长度**为

$$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

    - ➢ **随机探测再散列**、**二次探测再散列**和**再哈希**的哈希表查找成功的**平均查找长度**为

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

    - ➢ **链地址处理冲突**的哈希表查找成功的**平均查找长度**为 $S_{nc} \approx 1+\frac{\alpha}{2}$

哈希表

- 在等概率查找的情况下，同理可以证明：
  - **线性探测再散列**的哈希表查找不成功的**平均查找长度**为

$$U_{nl} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

  - **随机探测再散列**、**二次探测再散列**和**再哈希**的哈希表查找不成功的**平均查找长度**为

$$U_{nr} \approx \frac{1}{1-\alpha}$$

  - **链地址处理冲突**的哈希表查找不成功的**平均查找长度**为

$$U_{nc} \approx \alpha + e^{-\alpha}$$

哈希表

- 由此可见，由于哈希表的平均查找长度不是 n 的函数，而是α的函数，因此虽然不能做到平均查找长度为0，但可以设计一个哈希表，使它的平均查找长度控制在一个期望值之内。
- 从证明所得结论也可以看出，开放定址的哈希表的装填因子必定< 1，且不能接近于1，而链地址的哈希表的装填因子可以>1。
- 对开放定址的哈希表不能随意删除表中记录，而必须在该记录所在位置作一特殊标记，同时需修改前述查找算法添加识别已被删除记录。

本章小结

- 本章讨论查找表的各种表示方法以及查找效率的衡量标准-平均查找长度。
- 查找表即为集合结构，表中记录之间本不存在约束条件，但为了提高查找速度，在计算机中构建查找表时，应人为地在记录的关键字之间加上某些约束条件，即以其它结构表示之。
- 由于查找过程中的主要操作是关键字和给定值进行比较，因此以一次查找所需进行的比较次数的期望值作为查找方法效率的衡量标准，称之谓平均查找长度。

本章小结

- 在本章中介绍了查找表的三类存储表示方法：顺序表、树表和哈希表。这里的顺序表指的是顺序存储结构，包括有序表和索引顺序表，因此主要用于表示静态查找表，树表包括静态查找树和二叉排序树，树表和哈希表主要用于表示动态查找表。
- 二叉排序树的特点是，每经过一次比较便可将继续查找的范围缩小到某一棵子树上，但二叉排序树并不仅限于二叉树，以后还将介绍其它形式的二叉排序树。
- 所有顺序结构的表和二叉排序树的平均查找长度都是随之查找表中记录数的增加而增大，而哈希表的平均查找长度是装填因子的函数，因此有可能设计出使平均查找长度不超过某个期望值的哈希表。

本章知识点与重点

- **知识点**

  顺序表、有序表、索引顺序表、静态查找树、二叉查找树、哈希表

- **重点和难点**

  本章重点在于理解查找表的结构特点及其各种表示方法的特点和适用场合