# 第 10 章 索引与散列

10-1 什么是静态索引结构？什么是动态索引结构？它们各有哪些优缺点？

【解答】

静态索引结构指这种索引结构在初始创建，数据装入时就已经定型，而且在整个系统运行期间，树的结构不发生变化，只是数据在更新。动态索引结构是指在整个系统运行期间，树的结构随数据的增删及时调整，以保持最佳的搜索效率。静态索引结构的优点是结构定型，建立方法简单，存取方便；缺点是不利于更新，插入或删除时效率低。动态索引结构的优点是在插入或删除时能够自动调整索引树结构，以保持最佳的搜索效率；缺点是实现算法复杂。

10-2 设有 10000 个记录对象，通过分块划分为若干子表并建立索引，那么为了提高搜索效率，每一个子表的大小应设计为多大?

【解答】

每个子表的大小 $s = \lceil n \rceil = \lceil 10000 \rceil = 100$ 个记录对象。

10-3 如果一个磁盘页块大小为 1024 (=1K) 字节，存储的每个记录对象需要占用 16 字节，其中关键码占 4 字节，其它数据占 12 字节。所有记录均已按关键码有序地存储在磁盘文件中，每个页块的第 1 个记录用于存放线性索引。另外在内存中开辟了 256K 字节的空间可用于存放线性索引。试问：

(1) 若将线性索引常驻内存，文件中最多可以存放多少个记录？(每个索引项 8 字节，其中关键码 4 字节，地址 4 字节)

(2) 如果使用二级索引，第二级索引占用 1024 字节（有 128 个索引项），这时文件中最多可以存放多少个记录？

【解答】

(1) 因为一个磁盘页块大小为 1024 字节，每个记录对象需要占用 16 字节，则每个页块可存放 1024 / 16 = 64 个记录，除第一个记录存储线性索引外，每个页块可存储 63 个记录对象。又因为在磁盘文件中所有记录对象按关键码有序存储，所以线性索引可以是稀疏索引，每一个索引项存放一个页块的最大关键码及该页块的地址。若线性索引常驻内存，那么它最多可存放 256 * (1024 / 8 ) = 256 * 128 = 32768 个索引项，文件中可存放 32768 * 63 = 2064384 个记录对象。

(2) 由于第二级索引占用 1024 个字节，内存中还剩 255K 字节用于第一级索引。第一级索引有 255 * 128 = 32640 个索引项，作为稀疏索引，每个索引项索引一个页块，则索引文件中可存放 32640 * 63 = 2056320。

10-4 假设在数据库文件中的每一个记录是由占 2 个字节的整型数关键码和一个变长的数据字段组成。数据字段都是字符串。为了存放右面的那些记录，应如何组织线性索引？

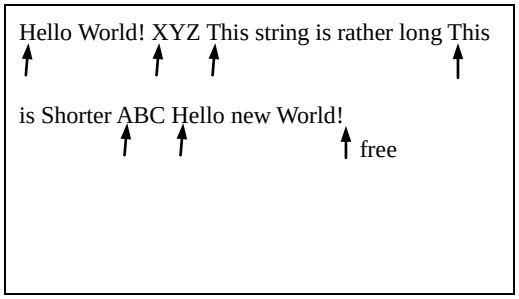
| | |
|---|---|
| 397 | Hello World! |
| 82 | XYZ |
| 1038 | This string is rather long |
| 1037 | This is Shorter |
| 42 | ABC |
| 2222 | Hello new World! |

【解答】

将所有字符串依加入的先后次序存放于一个连续的存储空间 store 中，这个空间也叫做“堆”，它是存放所有字符串的顺序文件。它有一个指针 free，指示在堆 store 中当前可存放数据的开始地址。初始时 free 置为 0，表示可从文件的 0 号位置开始存放。线性索引中每个索引项给出记录关键码，字符串在 store 中的起始地址和字符串的长度：

索引表 ID　　堆 store

| 关键码 | 串长度 | 串起始地址 |
|---|---|---|
| 42 | 3 | 56 |
| 82 | 3 | 12 |
| 397 | 12 | 0 |
| 1037 | 15 | 41 |
| 1038 | 26 | 15 |
| 2222 | 16 | 59 |




10-5 设有一个职工文件：

| 记录地址 | 职工号 | 姓　名 | 性别 | 职　业 | 年龄 | 籍贯 | 月工资(元) |
|---|---|---|---|---|---|---|---|
| 10032 | 034 | 刘激扬 | 男 | 教　师 | 29 | 山东 | 720.00 |
| 10068 | 064 | 蔡晓莉 | 女 | 教　师 | 32 | 辽宁 | 1200.00 |
| 10104 | 073 | 朱　力 | 男 | 实验员 | 26 | 广东 | 480.00 |
| 10140 | 081 | 洪　伟 | 男 | 教　师 | 36 | 北京 | 1400.00 |
| 10176 | 092 | 卢声凯 | 男 | 教　师 | 28 | 湖北 | 720.00 |
| 10212 | 123 | 林德康 | 男 | 行政秘书 | 33 | 江西 | 480.00 |
| 10248 | 140 | 熊南燕 | 女 | 教　师 | 27 | 上海 | 780.00 |
| 10284 | 175 | 吕　颖 | 女 | 实验员 | 28 | 江苏 | 480.00 |
| 10320 | 209 | 袁秋慧 | 女 | 教　师 | 24 | 广东 | 720.00 |

其中，关键码为职工号。试根据此文件，对下列查询组织主索引和倒排索引，再写出搜索结果关键码。(1) 男性职工；(2) 月工资超过 800 元的职工；(3) 月工资超过平均工资的职工；(4) 职业为实验员和行政秘书的男性职工；(5) 男性教师或者年龄超过 25 岁且职业为实验员和教师的女性职工。

【解答】

主索引

| | 职工号 | 记录地址 |
|---|---|---|
| 0 | 034 | 10032 |
| 1 | 064 | 10068 |
| 2 | 073 | 10104 |
| 3 | 081 | 10140 |
| 4 | 092 | 10176 |
| 5 | 123 | 10212 |
| 6 | 140 | 10248 |
| 7 | 175 | 10284 |
| 8 | 209 | 10320 |

月工资 倒排索引

| 月工资 | 长度 | 指针 |
|---|---|---|
| 480. | 3 | 073 |
| | | 123 |
| | | 175 |
| 720. | 3 | 034 |
| | | 092 |
| | | 209 |
| 780. | 1 | 140 |
| 1200. | 1 | 064 |
| 1400. | 1 | 081 |

职务 倒排索引

| 职务 | 长度 | 指针 |
|---|---|---|
| 教师 | 6 | 034 |
| | | 064 |
| | | 081 |
| | | 092 |
| | | 140 |
| | | 209 |
| 实验员 | 2 | 073 |
| | | 175 |
| 行政秘书 | 1 | 123 |

性别 倒排索引

| 性别 | 长度 | 指针 |
|---|---|---|
| 男 | 5 | 034 |
| | | 073 |
| | | 081 |
| | | 092 |
| | | 123 |
| 女 | 4 | 064 |

年龄 倒排索引

| 年龄 | 长度 | 指针 |
|---|---|---|
| 24 | 1 | 209 |
| 26 | 1 | 073 |
| 27 | 1 | 140 |
| 28 | 2 | 092 |
| | | 175 |
| 29 | 1 | 034 |

| | | | | |
|---|---|---|---|---|
| | 140 | 32 | 1 | 064 |
| | 175 | 33 | 1 | 123 |
| | 209 | 36 | 1 | 081 |

搜索结果：

(1) 男性职工 (搜索性别倒排索引)：**{**034, 073, 081, 092, 123**}**

(2) 月工资超过 800 元的职工 (搜索月工资倒排索引)：**{**064, 081**}**

(3) 月工资超过平均工资的职工(搜索月工资倒排索引) **{**月平均工资 776 元**}**：

**{**064, 081, 140**}**

(4) 职业为实验员和行政秘书的男性职工(搜索职务和性别倒排索引)：

**{**073, 123, 175**} && {**034, 073, 081, 092, 123**}** = **{**073, 123**}**

(5) 男性教师 (搜索性别与职务倒排索引)：

**{**034, 073, 081, 092, 123**} && {** 034, 064, 081, 092, 140, 209**}** = **{**034, 081, 092**}**

年龄超过 25 岁且职业为实验员和教师的女性职工 (搜索性别、职务和年龄倒排索引)：

**{**064, 140, 175, 209**} && {**034, 064, 073, 081, 092, 140, 175, 209**} && {**034, 064, 073, 081,092, 123, 140, 175**}** = **{**064, 140, 175**}**

10−6 倒排索引中的记录地址可以是记录的实际存放地址，也可以是记录的关键码。试比较这两种方式的优缺点。

【解答】

在倒排索引中的记录地址用记录的实际存放地址，搜索的速度快；但以后在文件中插入或删除记录对象时需要移动文件中的记录对象，从而改变记录的实际存放地址，这将对所有的索引产生影响：修改所有倒排索引的指针，不但工作量大而且容易引入新的错误或遗漏，使得系统不易维护。

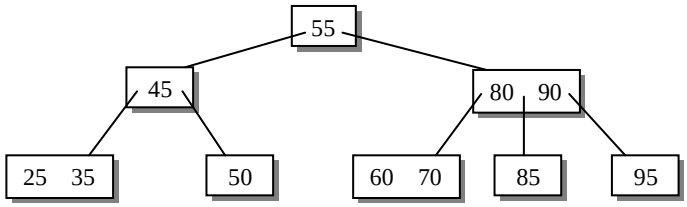
记录地址采用记录的关键码，缺点是寻找实际记录对象需要再经过主索引，降低了搜索速度；但以后在文件中插入或删除记录对象时，如果移动文件中的记录对象，导致许多记录对象的实际存放地址发生变化，只需改变主索引中的相应记录地址，其他倒排索引中的指针一律不变，使得系统容易维护，且不易产生新的错误和遗漏。

10−7 m = 2 的平衡 m 路搜索树是 AVL 树，m = 3 的平衡 m 路搜索树是 2−3 树。它们的叶结点必须在同一层吗？m 阶 B 树是平衡 m 路搜索树，反过来，平衡 m 路搜索树一定是 B 树吗？为什么？

【解答】

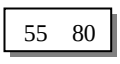
m = 3 的平衡 m 路搜索树的叶结点不一定在同一层，而 m 阶 B_树的叶结点必须在同一层，所以 m 阶 B_树是平衡 m 路搜索树，反过来，平衡 m 路搜索树不一定是 B_树。

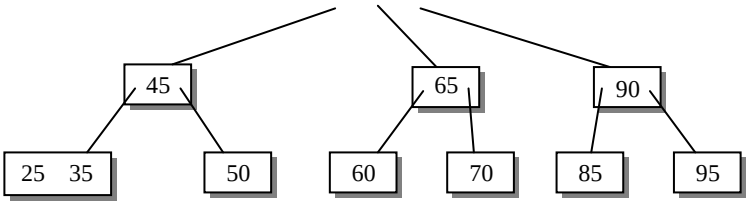
10−8 下图是一个 3 阶 B 树。试分别画出在插入 65、15、40、30 之后 B 树的变化。



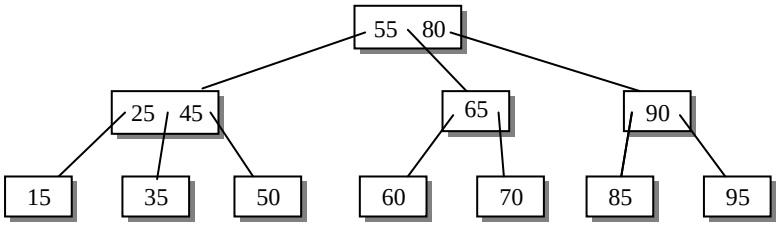

【解答】

插入 65 后：

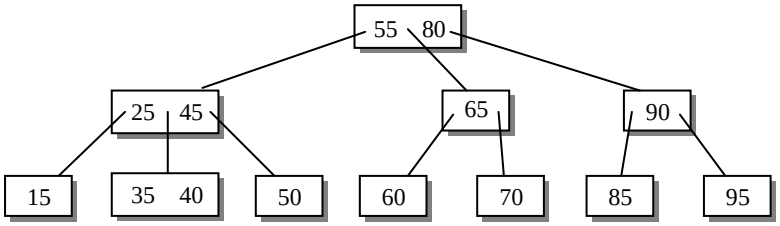




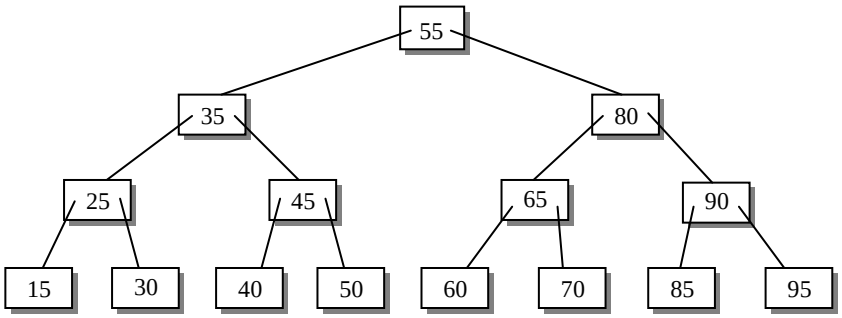

插入 15 后：



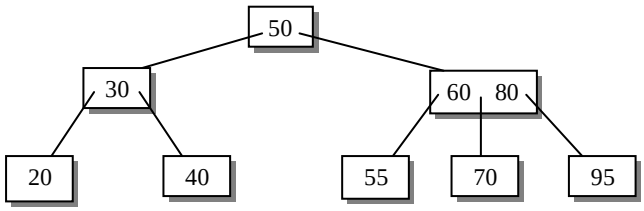

插入 40 后：




插入 30 后：

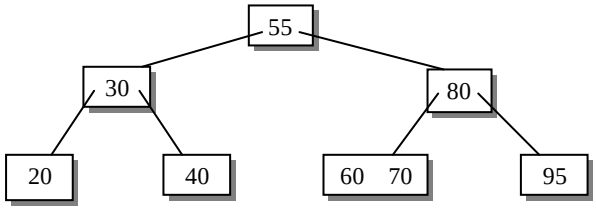



10-9 下图是一个 3 阶 B 树。试分别画出在删除 50、40 之后 B 树的变化。

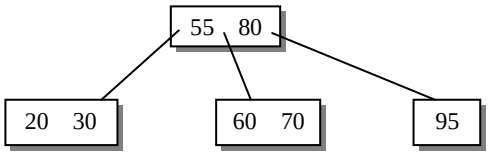



【解答】

删除 50 后：




删除 40 后：




10−10 对于一棵有 1999999 个关键码的 199 阶 B 树，试估计其最大层数(不包括失败结点)及最小层数(不包括失败结点)。

【解答】

设 B 树的阶数 m = 199，则$\lceil m/2 \rceil = 100$。若不包括失败结点层，则其最大层数为

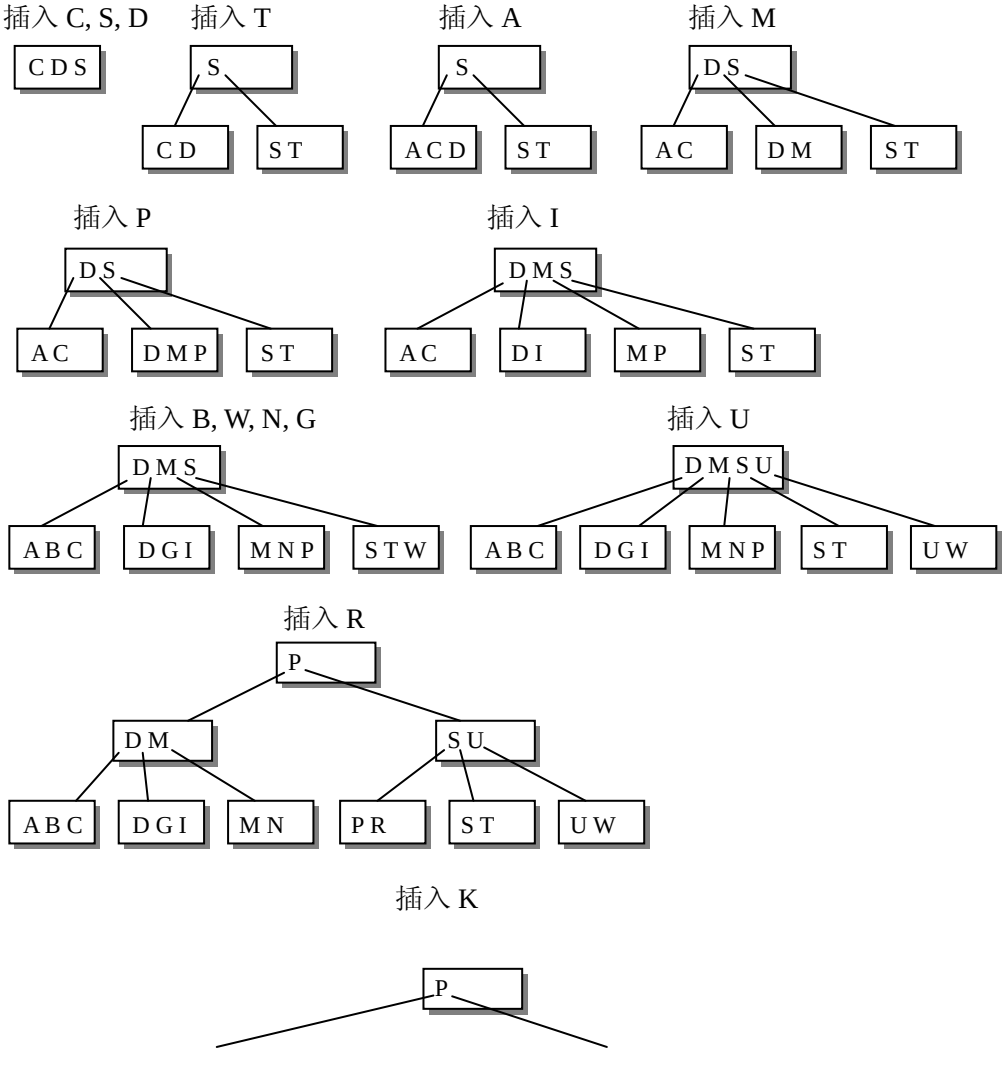
$$\lfloor \log_{\lceil m/2 \rceil}((N+1)/2) \rfloor + 1 = \lfloor \log_{100} 1000000 \rfloor + 1 = 4$$

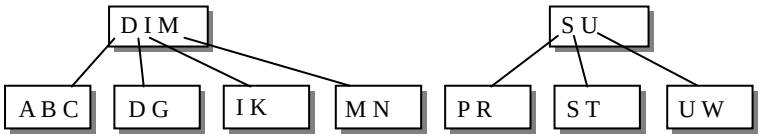
若使得每一层关键码数达到最大，可使其层数达到最小。第 0 层最多有(m−1)个关键码，第 1 层最多有$(m-1)^2$个关键码，…，第 h−1 层最多有$(m-1)^h$个关键码。层数为 h 的 B 树最多有$(m-1) + (m-1)^2 + \cdots + (m-1)^{h-1} = (m-1)\,((m-1)^h - 1)\,/\,(m-2)$个关键码。反之，若有 n 个关键码，$n \leqslant (m-1)\,((m-1)^h - 1)\,/\,(m-2)$，则 $h \geqslant \log_{(m-1)}(n(m-2)/(m-1)+1)$，所以，有 1999999 个关键码的 199 阶 B 树的最小层数为

$$\lceil \log_{(m-1)}(n*(m-2)/(m-1)+1) \rceil = \lceil \log_{198}(1999999*197 / 198 +1) \rceil = \lceil \log_{198} 1989899 \rceil$$

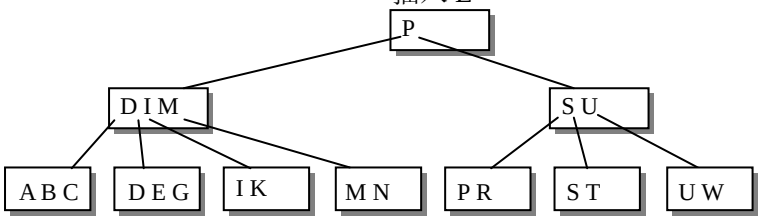
10−11 给定一组记录，其关键码为字符。记录的插入顺序为 **{** C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J **}**，给出插入这些记录后的 4 阶 B+树。假定叶结点最多可存放 3 个记录。

【解答】

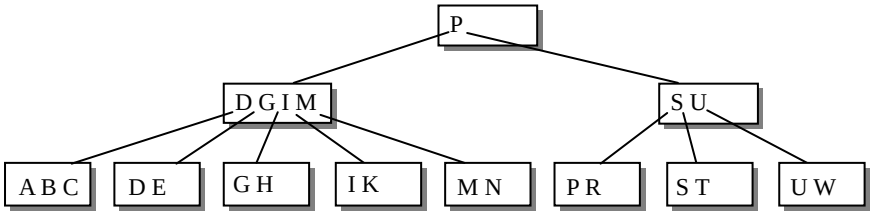




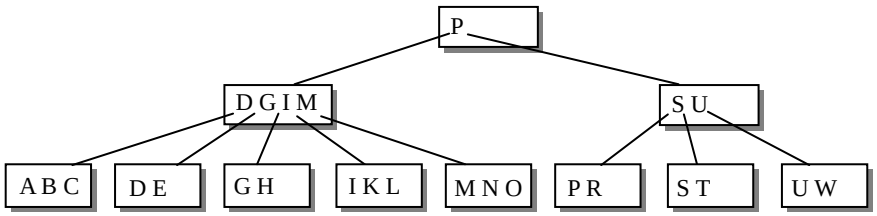

插入 E



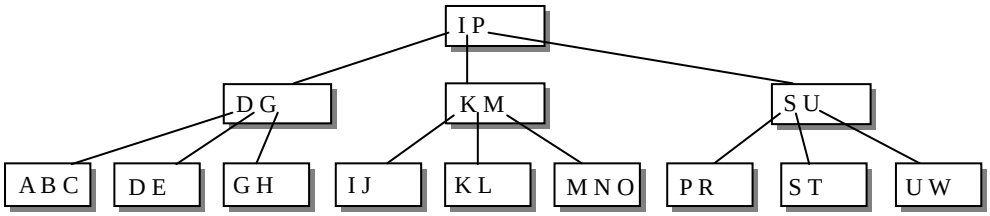

插入 H




插入 O, L




插入 J




10−12 设有一棵 B+树，其内部结点最多可存放 100 个子女，叶结点最多可存储 15 个记录。对于 1, 2, 3, 4, 5 层的 B+树，最多能存储多少记录，最少能存储多少记录。

【解答】

一层 B+树：根据 B+树定义，一层 B+树的结点只有一个，它既是根结点又是叶结点，最多可存储 m1 = 15 个记录，最少可存储 $\lceil m1/2 \rceil = 8$ 个记录。

二层 B+树：第 0 层是根结点，它最多有 m = 100 棵子树，最少有 2 个结点；第 1 层是叶结点，它最多有 m 个结点，最多可存储 m*m1 = 100*15 = 1500 个记录，最少有 2 个结点，最少可存储 $2*\lceil m1/2 \rceil = 16$ 个记录。

三层 B+树：第 2 层是叶结点。它最多有 $m^2$ 个结点，最多可存储 $m^2 * m1 = 150000$ 个记录。最少有 $2*\lceil m/2 \rceil = 100$ 个结点，最少可存储 $2*\lceil m/2 \rceil * \lceil m1/2 \rceil = 800$ 个记录。

四层 B+树：第 3 层是叶结点。它最多有 $m^3$ 个结点，可存储 $m^3 * m1 = 15000000$ 个记录。最少有 $2*\lceil m/2 \rceil^2 = 2 * 50^2 = 5000$ 个结点，存储 $2*\lceil m/2 \rceil^2 * \lceil m1/2 \rceil = 40000$ 个记录。

五层 B+树：第 4 层是叶结点。它最多有 $m^4$ 个结点，可存储 $m^4 * m1 = 1500000000$ 个记

录。最少有 $2*\lceil m/2\rceil^3 = 2 * 50^3 = 250000$ 个结点，存储 $2*\lceil m/2\rceil^3 * \lceil m1/2\rceil = 2000000$ 个记录。

10−13 设散列表为 HT[13]，散列函数为 H (*key*) = *key* %13。用闭散列法解决冲突，对下列关键码序列 12, 23, 45, 57, 20, 03, 78, 31, 15, 36 造表。

(1) 采用线性探查法寻找下一个空位，画出相应的散列表，并计算等概率下搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

(2) 采用双散列法寻找下一个空位，再散列函数为 RH (*key*) = (7**key*) % 10 + 1，寻找下一个空位的公式为 $H_i = (H_{i-1} + RH(key))\ \%\ 13$, $H_1 = H(key)$。画出相应的散列表，并计算等概率下搜索成功的平均搜索长度。

【解答】

使用散列函数 H(*key*) = *key* **mod** 13，有

H(12) = 12, H(23) = 10, H(45) = 6, H(57) = 5,
H(20) = 7, H(03) = 3, H(78) = 0, H(31) = 5,
H(15) = 2, H(36) = 10.

(1) 利用线性探查法造表：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 78 | | 15 | 03 | | 57 | 45 | 20 | 31 | | 23 | 36 | 12 |
| (1) | | (1) | (1) | | (1) | (1) | (1) | (4) | | (1) | (2) | (1) |

搜索成功的平均搜索长度为

$$ASL_{succ} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 4 + 1 + 2 + 1) = \frac{14}{10}$$

搜索不成功的平均搜索长度为

$$ASL_{unsucc} = \frac{1}{13}(2 + 1 + 3 + 2 + 1 + 5 + 4 + 3 + 2 + 1 + 5 + 4 + 3) = \frac{36}{13}$$

(2) 利用双散列法造表：

$H_i = (H_{i-1} + RH(key))\ \%\ 13$, $H_1 = H(key)$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 78 | | 15 | 03 | | 57 | 45 | 20 | 31 | 36 | 23 | | 12 |
| (1) | | (1) | (1) | | (1) | (1) | (1) | (3) | (5) | (1) | | (1) |

搜索成功的平均搜索长度为

$$ASL_{succ} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 3 + 5 + 1 + 1) = \frac{16}{10}$$

10−14 设有 150 个记录要存储到散列表中，要求利用线性探查法解决冲突，同时要求找到所需记录的平均比较次数不超过 2 次。试问散列表需要设计多大？ 设α是散列表的装载因子，则有

$$ASL_{succ} = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

【解答】

已知要存储的记录数为 n = 150，查找成功的平均查找长度为 $ASL_{succ} \le 2$，则有 $ASL_{succ} = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \le 2$，解得 $\alpha \le \frac{2}{3}$。又有$\alpha = \frac{n}{m} = \frac{150}{m} \le \frac{2}{3}$，则 $m \ge 225$。

10−15 若设散列表的大小为 m，利用散列函数计算出的散列地址为 h = hash(x)。

(1) 试证明：如果二次探查的顺序为$(h + q^2), (h + (q-1)^2), \cdots, (h+1), h, (h-1), \cdots, (h-q^2)$，其中， q = (m−1)/2。因此在相继被探查的两个桶之间地址相减所得的差取模(%m)的结果为

m−2, m−4, m−6, …, 5, 3, 1, 1, 3, 5, …, m−6, m−4, m−2

(2) 编写一个算法，使用课文中讨论的散列函数 h(x)和二次探查解决冲突的方法，按给定值 x 来搜索一个大小为 m 的散列表。如果 x 不在表中，则将它插入到表中。

【解答】

(1) 将探查序列分两部分讨论：

$(h + q^2), (h + (q-1)^2), \cdots, (h+1), h$ 和 $(h-1), (h-2^2), \cdots, (h-q^2)$。

对于前一部分，设其通项为 $h + ( q – d )^2$, d = 0, 1, …, q，则相邻两个桶之间地址相减所得的差取模：

$( h + (q – (d -1) )^2 – ( h + (q – d )^2 ) ) \% m = ( (q – (d -1 ) )^2 – (q – d )^2 ) \% m$

$= (2*q -2*d +1) \% m = ( m – 2*d ) \% m.$　　( 代换 q = (m−1)/2 )

代入 d = 1, 2, …, q，则可得到探查序列如下：

m−2, m−4, m−6, …, 5, 3, 1。 ( m – 2*q = m – 2* (m−1)/2 = 1 )

对于后一部分，其通项为 $h – ( q – d )^2$, d = q, q+1, …, 2q，则相邻两个桶之间地址相减所得的差取模：

$( h – ( q – d )^2 – ( h – ( q – (d+1) )^2 ) ) \% m = ( ( q – (d+1)^2 – (q – d )^2 ) \% m$

$= ( 2*d – 2*q +1) \% m = ( 2*d – m + 2) \% m$　( 代换 q = (m−1)/2 )

代入 d = q, q+1, …, 2q−1，则可得到

2*d–m+2 = 2*q – m +2 = m – 1 – m +2 = 1,

2*d–m+2 = 2*q + 2 – m +2 = m – 1 + 2 – m +2 = 3, ……,

2*d–m+2 = 2*(2*q−1) – m +2 = 2*(m–1–1) – m + 2 = 2*m – 4 – m +2 = m – 2。〖证毕〗

(2) 编写算法

下面是使用二次探查法处理溢出时的散列表类的声明。

```
template <class Type> class HashTable {                    //散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };           //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { AllocateHt ( ); CurrentSize = 0; }  //构造函数
    ~HashTable ( ) { delete [ ] ht; }                      //析构函数
    const HashTable & operator = ( const HashTable & ht2 );  //重载函数：表赋值
    int Find ( const Type & x );                           //在散列表中搜索 x
    int IsEmpty ( ) { return !CurrentSize ? 1 : 0; }       //判散列表空否，空则返回 1
private:
    struct HashEntry {                                     //散列表的表项定义
        Type Element;                                      //表项的数据，即表项的关键码
        KindOfEntry info;                                  //三种状态: Active, Empty, Deleted
        HashEntry ( ) : info (Empty ) { }                  //表项构造函数
        HashEntry ( const Type &E, KindOfEntry i = Empty ) : Element (E), info (i) { }
    };
    enum { DefualtSize = 31; }
    HashEntry *ht;                          //散列表存储数组
    int TableSize;                          //数组长度，要求是满足 4k+3 的质数，k 是整数
    int CurrentSize;                        //已占据散列地址数目
    void AllocateHt ( ) { ht = new HashEntry[TableSize ]; }     //为散列表分配存储空间;
    int FindPos ( const Type & x );         //散列函数
};

template <class Type> const HashTable<Type> & HashTable<Type> ::
operator = ( const HashTable<Type> &ht2 ) {
//重载函数：复制一个散列表 ht2
    if ( this != &ht2 ) {
        delete [ ] ht;    TableSize = ht2.TableSize;    AllocateHt ( );   //重新分配目标散列表存储空间
```

```
        for ( int i = 0; i < TableSize; i++ ) ht[i] = ht2.ht[i];          //从源散列表向目标散列表传送
        CurrentSize = ht2.CurrentSize;                                    //传送当前表项个数
    }
    return *this;                                                         //返回目标散列表结构指针
}

template <class Type> int HashTable<Type> :: Find ( const Type& x ) {
//共有函数：  找下一散列位置的函数
    int i = 0, q = ( TableSize −1 ) / 2, h0;              // i 为探查次数
    int CurrentPos = h0 = HashPos ( x );                  //利用散列函数计算 x 的散列地址
    while ( ht[CurrentPos].info != Empty && ht[CurrentPos].Element != x ) {
                                                          /搜索是否要求表项
        if ( i <= q ) {                                   //求“下一个”桶
            CurrentPos = h0 + (q − i ) * ( q − i );
            while ( CurrentPos >= TableSize ) CurrentPos −= TableSize;        //实现取模
        }
        else {
            CurrentPos = h0 − ( i −q ) * ( i − q );
            while ( CurrentPos < 0 ) CurrentPos += TableSize;                 //实现取模
        }
        i++;
    }
    if ( ht[CurrentPos].info == Active && ht[CurrentPos].Element == x )
        return CurrentPos;                                                    //返回桶号
    else {
        ht[CurrentPos].info = Active;   ht[CurrentPos].Element = x;           //插入 x
        if ( ++CurrentSize < TableSize / 2 ) return CurrentPos;
                                          //当前已有项数加 1, 不超过表长的一半返回
        HashEntry *Oldht = ht;            //分裂空间处理：保存原来的散列表
        int OldTableSize = TableSize;
        CurrentSize = 0;
        TableSize = NextPrime ( 2 * OldTableSize );   //原表大小的 2 倍，取质数
        Allocateht ( );                               //建立新的二倍大小的空表
        for ( i = 0; i < OldTableSize; i++)           //原来的元素重新散列到新表中
            if ( Oldht[i].info == Active ) {
                Find ( Oldht[i].Element );            //递归调用
                if ( Oldht[i].Element == x ) CurrentPos = i;
            }
        delete [ ] Oldht;
        return CurrentPos;
    }
}
```

求下一个大于参数表中所给正整数 *N* 的质数的算法。

```
int NextPrime ( int N ) {                 //求下一个>N 的质数，设 N >= 5
    if ( N % 2 == 0 ) N++;                //偶数不是质数
    for ( ; !IsPrime (N); N += 2 );       //寻找质数
    return N;
}
```

```
int IsPrime ( int N ) {                    //测试 N 是否质数
    for ( int i = 3; i*i <= N; i += 2 )    //若 N 能分解为两个整数的乘积, 其中一个一定 ≤√N
        if ( N % i == 0 ) return 0;        //N 能整除 i, N 不是质数
    return 1;                              //N 是质数
}
```

10-16 编写一个算法，以字典顺序输出散列表中的所有标识符。设散列函数为 hash(x) = x 中的第一个字符，采用线性探查法来解决冲突。试估计该算法所需的时间。

【解答】

用线性探查法处理溢出时散列表的类的声明。

```
#define DefaultSize 1000
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class HashTable {                                   //散列表类定义
public:
    enum KindOfEntry { Active, Empty, Deleted };    //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { ht = new HashEntry[TableSize]; }  //构造函数
    ~HashTable ( ) { delete [ ] ht; }               //析构函数
    int Find-Ins ( const char * id );               //在散列表中搜索标识符 id
    void HashSort ( );
private:
    struct HashEntry {                              //表项定义
        Type Element;                               //表项的数据, 即表项的关键码
        KindOfEntry info;                           //三种状态: Active, Empty, Deleted
        HashEntry ( ) : info (Empty ) { }           //表项构造函数, 置空
    };
    HashEntry *ht;                                  //散列表存储数组
    int TableSize;                                  //最大桶数
    int FindPos ( string s ) const { return atoi (*s) - 32; }   //散列函数
}

int HashTable :: Find-Ins ( const char * id ) {
    int i = FindPos ( id ),  j = i;                 //i 是计算出来的散列地址
    while ( ht[j].info != Empty && strcmp ( ht[j].Element, id ) != 0 ) {     //冲突
        j = ( j + 1 ) % TableSize;                  //当做循环表处理, 找下一个空桶
        if ( j == i ) return -TableSize;            //转一圈回到开始点, 表已满, 失败
    }
    if ( ht[j].info != Active ) {                   //插入
      if ( j > i ) {
          while ( int k = j; k > i; k--)
              { ht[k].Element = ht[k-1].Element;   ht[k].info = ht[k-1].info; }
          ht[i].Element = id;   ht[i].info = Active;          //插入
       } else {
          HashEntry temp;
          temp.Element = ht[TableSize-1].Element;   temp.info = ht[TableSize-1].info;
          while ( int k = TableSize-1; k > i; k--)
              { ht[k].Element = ht[k-1].Element;   ht[k].info = ht[k-1].info; }
```

```
            ht[i].Element = id;   ht[i].info = Active;          //插入
            while ( int k = j; k > 0; k−− )
               { ht[k].Element = ht[k−1].Element;   ht[k].info = ht[k−1].info; }
            ht[0].Element = temp.Element;   ht[0].info = temp.info;
        }
        return j;
    }

    void HashTable :: HashSort ( ) {
        int n, i;   char * str;
        cin >> n >> str;
        for ( i = 0; i < n; i++ ) {
          if ( Find-Ins ( str ) == − Tablesize ) { cout << "表已满" << endl;   break; }
          cin >> str;
        }
        for ( i = 0; i < TableSize; i++ )
          if ( ht[i].info == Active ) cout << ht[i].Element << endl;
    }
```

10−17 设有 1000 个值在 1 到 10000 的整数，试设计一个利用散列方法的算法，以最少的数据比较次数和移动次数对它们进行排序。

【解答 1】

建立 TableSize = 10000 的散列表，散列函数定义为

```
int HashTable :: FindPos ( const int x ) { return x−1; }
```

相应排序算法基于散列表类

```
#define DefaultSize 10000
#define n 1000
class HashTable {                                       //散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };        //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { ht = new HashEntry[TableSize ]; }   //构造函数
    ~HashTable ( ) { delete [ ] ht; }                   //析构函数
    void HashSort ( int A[ ], int n );                  //散列法排序
private:
    struct HashEntry {                                  //散列表的表项定义
        int Element;                                    //表项的数据, 整数
        KindOfEntry info;                               //三种状态: Active, Empty, Deleted
        HashEntry ( ) : info (Empty ) { }               //表项构造函数
    };
    HashEntry *ht;                                      //散列表存储数组
    int TableSize;                                      //数组长度
    int FindPos ( int x );                              //散列函数
};

void HashTable<Type> :: HashSort ( int A[ ], int n ) {          //散列法排序
    for ( int i = 0; i < n; i++ ) {
        int position = FindPos( A[i] );
        ht[position].info = Active;   ht[position].Element = A[i];
    }
```

```
    int pos = 0;
    for ( int i = 0; i < TableSize; i++ )
      if ( ht[i].info == Active )
        { cout << ht[i].Element << endl;   A[pos] = ht[i].Element;   pos++; }
}
```

【解答 2】

利用开散列的方法进行排序。其散列表类及散列表链结点类的定义如下：

```
#define DefaultSize 3334
#define n 1000
class HashTable;                          //散列表类的前视声明

class ListNode {                          //各桶中同义词子表的链结点(表项)定义
friend class HashTable;
private:
    int key;                              //整数数据
    ListNode *link;                       //链指针
public:
    ListNode ( int x ) : key(x), link(NULL) { } //构造函数
};
typedef ListNode *ListPtr;                //链表指针

class HashTable {                         //散列表(表头指针向量)定义
public:
    HashTable( int size = DefaultSize )   //散列表的构造函数
      { TableSize = size; ht = new ListPtr[size]; }  //确定容量及创建指针数组
    void HashSort ( int A[ ]; int n )
private:
    int TableSize;                        //容量(桶的个数)
    ListPtr *ht;                          //散列表定义
    int FindPos ( int x ) { return x / 3; }
}

void HashTable<Type> :: HashSort ( int A[ ]; int n ) {
    ListPtr * p , *q;   int i, bucket, k = 0;
    for ( i = 0; i < n; i++ ) {           //对所有数据散列, 同义词子表是有序链表
      bucket = FindPos ( A[i] );          //通过一个散列函数计算桶号
      p = ht[bucket];   q = new ListNode(A[i]);
      if ( p == NULL || p->key > A[i] )   //空同义词子表或*q 的数据最小
        { q->link = ht[bucked];   ht[bucked] = q; }
      else if ( p->link == NULL || p->link->key > A[i] )
            { q->link = p->link;   p->link = q; }
          else p->link->link = q;         //同义词子表最多 3 个结点
    }
    for ( i = 0; i < TableSize; i++ ) {   //按有序次序输出
      p = ht[i];
      while ( p != NULL ) {
          cout << p->key << endl;   A[k] = p->key;   k++;
          p = p->link;
```

```
        }
      }
    }
```

10-18 设有 15000 个记录需放在散列文件中，文件中每个桶内各页块采用链接方式连结，每个页块可存放 30 个记录。若采用按桶散列，且要求搜索到一个已有记录的平均读盘时间不超过 1.5 次，则该文件应设置多少个桶？

【解答】

已知用链地址法（开散列）解决冲突，搜索成功的平均搜索长度为 $1+\alpha/2 \leqslant 1.5$，解出 $\alpha \leqslant 1$，又 $\alpha = n / m = 15000 / 30 / m = 500 / m \leqslant 1$，$m \geqslant 500$。由此可知，该文件至少应设置 500 个桶。

10-19 用可扩充散列法组织文件时，若目录深度为 d，指向某个页块的指针有 n 个，则该页块的局部深度有多大？

【解答】

设页块的局部深度为 d'，根据题意有 $n = 2^{d-d'}$，因此，$d' = d - \log_2 n$。

10-20 设一组对象的关键码为 {69, 115, 110, 255, 185, 143, 208, 96, 63, 175, 160, 99, 171, 137, 149, 229, 167, 121, 204, 52, 127, 57, 1040}。要求用散列函数将这些对象的关键码转换成二进制地址，存入用可扩充散列法组织的文件里。定义散列函数为 hash(key) = key % 64, 二进制地址取 6 位。设每个页块可容纳 4 个对象。要求按 10 .4 节介绍的方法设置目录表的初始状态，使目录表的深度为 3。然后按题中所给的顺序，将各个对象插入的可扩充散列文件中。试画出每次页块分裂或目录扩充时的状态和文件的最后状态。

【解答】

$hash(69) = 5_{(10)} = 000101_{(2)}$　　$hash(115) = 51_{(10)} = 110011_{(2)}$

$hash(110) = 46_{(10)} = 101110_{(2)}$　　$hash(255) = 63_{(10)} = 111111_{(2)}$

$hash(185) = 57_{(10)} = 111001_{(2)}$　　$hash(143) = 15_{(10)} = 001111_{(2)}$

$hash(208) = 16_{(10)} = 010000_{(2)}$　　$hash(96) = 32_{(10)} = 100000_{(2)}$

$hash(63) = 63_{(10)} = 111111_{(2)}$　　$hash(175) = 47_{(10)} = 101111_{(2)}$

$hash(160) = 32_{(10)} = 100000_{(2)}$　　$hash(99) = 35_{(10)} = 100011_{(2)}$

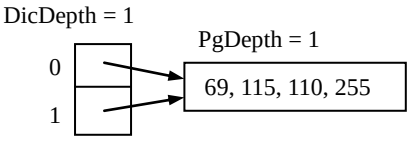
$hash(171) = 43_{(10)} = 101011_{(2)}$　　$hash(137) = 9_{(10)} = 001001_{(2)}$

$hash(149) = 21_{(10)} = 010101_{(2)}$　　$hash(229) = 37_{(10)} = 100101_{(2)}$

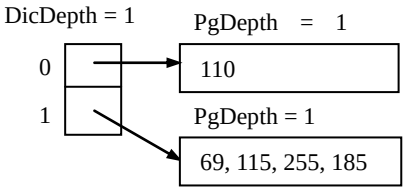
$hash(167) = 39_{(10)} = 101001_{(2)}$　　$hash(121) = 57_{(10)} = 111001_{(2)}$

$hash(204) = 12_{(10)} = 001100_{(2)}$　　$hash(52) = 52_{(10)} = 110100_{(2)}$

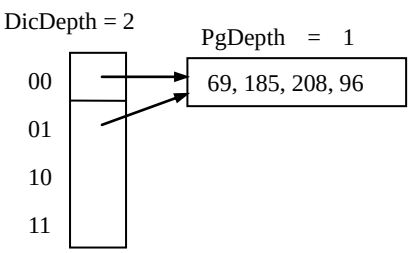
根据题意，每个页块可容纳 4 个对象，为画图清晰起见仅给出前 20 个关键码插入后的结果。目录表的深度 d = 3。

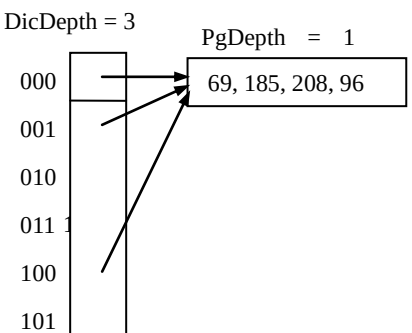


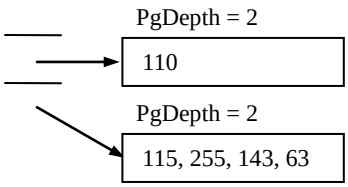

初态，插入 69, 115, 110, 255



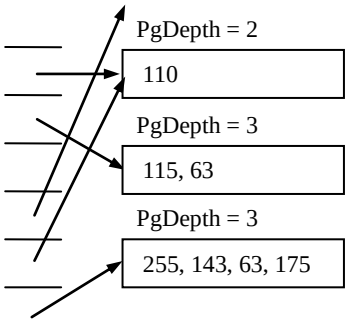

插入 185，页块分裂



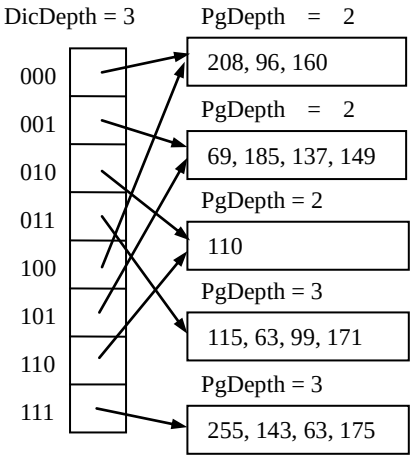





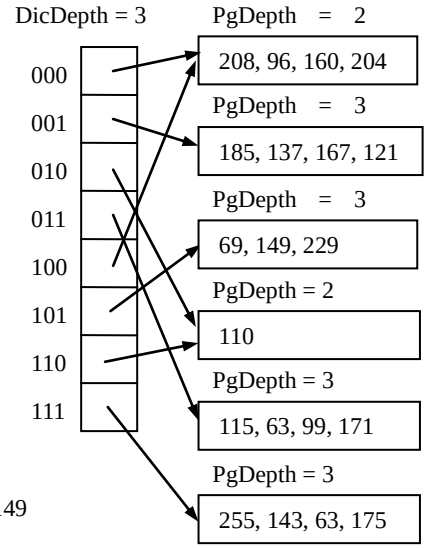

插入 143，目录分裂，插入 208, 96, 63



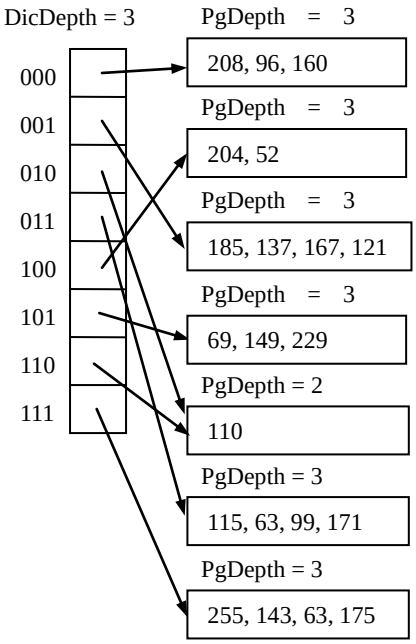

插入 175，目录分裂




插入 160，页块分裂，插入 99, 171, 137, 149




插入 229，页块分裂，插入 167, 121, 204




插入 52，页块分裂