第8章 查找算法2

知识要点

(1) 理解查找的概念；
(2) 掌握静态查找表的方法；
(3) 熟悉并能运用散列技术解决实际问题；
(4) 熟悉线性索引以及树形索引。

薄钧戈
2021年5月19日

单选题 1分

设置

采用折半查找方法查找长度为n的线性表，当n很大时，在等概率时不成功查找的平均查找长度为（ ）。

A O(n)

B O(nlogn)

C O(log(n))

D O(n^2)

提交

8.3 散列表-几个概念

- 散列表(哈希表)是一种非关键字查找方法，可像数组检索一样实现随机查找。
  - 将关键字映射到表中的位置来访问记录的过程称为散列，又称哈希。
  - 哈希函数：一个哈希表使用一个函数H（称为哈希函数），把关键字映射到非负整数（或表的索引值，称为哈希地址）。
  - *Address=H(key)*
- ★注：按哈希表的存储范围来定义存储大小。
- 哈希函数既是建存储表的函数也是哈希查找的函数。

8.3 散列表

- 哈希表构建的关键问题：
  - 1）构造合适的哈希函数；
  - 2）设计合理的哈希冲突解决方案。
- 哈希冲突
  - 不同关键字经哈希函数映射地址相同称为哈希冲突。
    - 即：对于两个关键字分别为k_i和k_j（$i \neq j$）的记录，有$k_i \neq k_j$，但$h(k_i)=h(k_j)$。把这种现象叫做哈希冲突（同义词冲突）。
  - 一般来说，由于哈希函数把关键字的大值域映射到哈希表空间的小值域，故哈希冲突是难于避免的。

8.3.1 哈希函数的常用构建方法

- 构造哈希函数需要注意的几点要求：（好的哈希函数）
  - （1）让哈希地址在哈希表中尽可能的分布均匀，即减少冲突；
  - （2）哈希函数计算一定要简单且快速；
  - （3）哈希函数的定义域必须包括需要存储的全部关键字；
  - （4）若哈希表允许有m个地址时，其值域必须在[0..m-1]之间。

8.3.1 哈希函数的常用构建方法

- 常见的哈希函数构造方法有：
  - 直接定址法
  - 除留余数法
  - 平方取中法
  - 数字分析法
  - 折叠法
  - 随机数法

8.3.1 哈希函数的常用构建方法-直接定址法

- 哈希函数Hash(key)=a*key+b，其中a和b都为常数。
  - 直接定址法取关键字的某个线性函数值为哈希地址。
  - 此类哈希函数计算方法最简单，是一对一的映射，一般不会产生冲突。
- 直接定址法适用于关键字分布基本连续的情况，如果关键字分布不连续，空位较多，就会造成存储空间的浪费。实际中很少使用直接定址法。

8.3.1 哈希函数的常用构建方法-直接定址法

- 【例8.2】有一组记录元素的关键字集合{100，300，500，700，800，900}，选取的哈希函数为：
  - Hash(key)=key/100
  - 采用直接定址法后的存放如下所示：
  - Hash(100)=1； Hash(300)=3； Hash(500)=5；
  - Hash(700)=7； Hash(800)=8； Hash(900)=9；

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

8.3.1 哈希函数的常用构建方法-除留取余法

- 哈希函数Hash(key)=key %p，如果哈希表中允许的地址数为m，p为不大于m的最大质数。
  - 除留余数法是通过取关键字除以p的余数作为哈希地址。
  - 其中p的选择很重要，如果选得不好会产生很多冲突。
    - 比如关键字都是10的倍数，而p=10
  - 为了减少冲突，在一般情况下，p最好为素数或不包含小于20的素数因子的合数。
- 例：
  - 如有一个关键字key=11516287，哈希表大小m=100，取质数p=97，哈希函数Hash(key)=key %p，则哈希地址为Hash(11516287)=11516287 % 97=59。

8.3.1 哈希函数的常用构建方法-平方取中法

- 平方取中法首先通过计算关键字的平方值来扩大相近数之间的差别，然后按照哈希表的大小取中间的几位数作为哈希地址。
  - 由于一个乘积的中间几位数与乘数的每一位都相关，从而由此产生的哈希地址比较均匀。
  - 求“关键字的平方值”的目的是“扩大差别”和“贡献均衡”。
    - 即：关键字的各位都在平方值的中间几位有所贡献，Hash 值中应该有各位影子。
  - 这种方法在词典处理中应用十分广泛。
- 【例8.3】
  - 有一组记录元素的关键字集合{0100，0110，1010，1001，0111}，将每个关键字平方后得到集合{0010000，0012100，1020100，1002001，0012321}，如果哈希表的长度为1000，则可以取第3，4，5位作为哈希地址，分别为{100，121，201，020，123}。

8.3.1 哈希函数的常用构建方法-截断法（数字分析法）

- 设 n 个 d 位数的关键字，由 r 个不同的符号组成，此 r 个符号在关键字各位出现的频率不一定相同，可能在某些位上均匀分布，即每个符号出现的次数都接近于 n/r 次，而在另一些位上分布不均匀。则选择其中分布均匀的s位作为哈希地址。
  - 即H（key）＝“key中数字均匀分布的s位”
  - 忽略关键字中的一部分，选取剩余部分作为表的哈希值。
  - 如学号，采用学号的后三位。
  - 即寻找区分性好的段作为哈希地址。

8.3.1 哈希函数的常用构建方法-截断法（数字分析法）

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

- n=80,d=8,r=10,s=2
  - 1，2，3，8位分布不均匀，不能取
  - 可取第4、6两位组成的2位十进制数作为每个数据的哈希地址
  - 则图中列出的关键字的哈希地址分别为：45，72，84，03，28，39，51，65，13

8.3.1 哈希函数的常用构建方法-折叠法

- 关键字位数较长时，把关键字自左到右分成位数相等的几部分（最后一部分位数可以不同），每一部分的位数应与哈希表地址位数相同，取这几部分的叠加和（舍去高位的进位）作为哈希地址

- 折叠法有两种叠加方法：

  ①移位法：把各部分的最后一位对齐相加；

  ②分界法：各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果作为哈希地址。

此方法适合于：关键字的数字位数特别多

$$
\begin{array}{r}
d_r \cdots\cdots\ d_2\ \ d_1 \\
d_{2r} \cdots\cdots d_{r+2}\ \ d_{r+1} \\
+)\ d_{3r} \cdots\cdots d_{2r+2}\ d_{2r+1} \\
\hline
S_r \cdots\cdots\ S_2\ \ S_1
\end{array}
$$

(a) 移位叠加法

$$
\begin{array}{r}
d_r \cdots\cdots\ d_2\ \ d_1 \\
d_{r+1} \cdots\cdots d_{2r-1} d_{2r} \\
+)\ d_{3r} \cdots\cdots d_{2r+2}\ d_{2r+1} \\
\hline
S_r \cdots\cdots\ S_2\ \ S_1
\end{array}
$$

(b) 边界叠加法

8.3.1 哈希函数的常用构建方法-折叠法

- 如key=11516287，若存储空间限定为3位，则划分结果为每段3位。将关键字划分为3段：[1 1 5]，[1 6 2]，[8 7]，累加结果如果超出地址位数的最高位则删除，仅保留最低的3位作为哈希地址，如图所示。
  - [1 1 5]，[1 6 2]，[8 7]

移位法

```
  1  1  5
  1  6  2
+)    8  7
-----------
  3  6  4
```

分界法

```
  1  1  5
  2  6  1
+)8  7
-----------
[1] 2  4  6
```

图 8-3 折叠法

8.3.1 哈希函数的常用构建方法-伪随机数法

- 哈希函数Hash(key)=Random(key)，即取关键字的随机函数值作为关键字的哈希地址。其中Random为伪随机函数，其取值在0到m-1之间。

★注：哈希函数不是一个完全的随机数，是与关键字的值相关的伪随机数。如此才能将存储与查找相对应。

8.3.1 哈希函数的常用构建方法-考虑因素

- 在实际工作中，需要根据不同的情况来采用不同的哈希函数。通常考虑的因素有：
  - 计算哈希函数所需的时间；
  - 关键字的长度；
  - 哈希表空间的大小；
  - 关键字的分布情况；
  - 记录的查找频率。

主观题 10分

设置

8.3.1 哈希函数的常用构建方法-思考

我有一个电话号码本，怎么根据姓名建立哈希表呢？

正常使用主观题需2.0以上版本雨课堂

作答

8.3.2 冲突处理

- 冲突：
  - 是指由关键字得到的Hash地址上已有其他记录。
  - 好的哈希函数可以减少冲突，但很难避免冲突。
- 冲突处理：
  - 为出现哈希地址冲突的关键字寻找下一个哈希地址。

8.3.2 解决冲突的办法

- 冲突解决技术可以分为两类：开放哈希和封闭哈希
- 两种解决冲突的方法的区别：
  - 开放哈希法是冲突元素不占哈希表空间，即不在哈希表中存放，而是存放到哈希表之外的空间中去；
  - 封闭哈希法允许冲突元素存放到哈希表空间中，即放到哈希表的空闲区域中。

8.3.2 解决冲突的办法-开放地址法

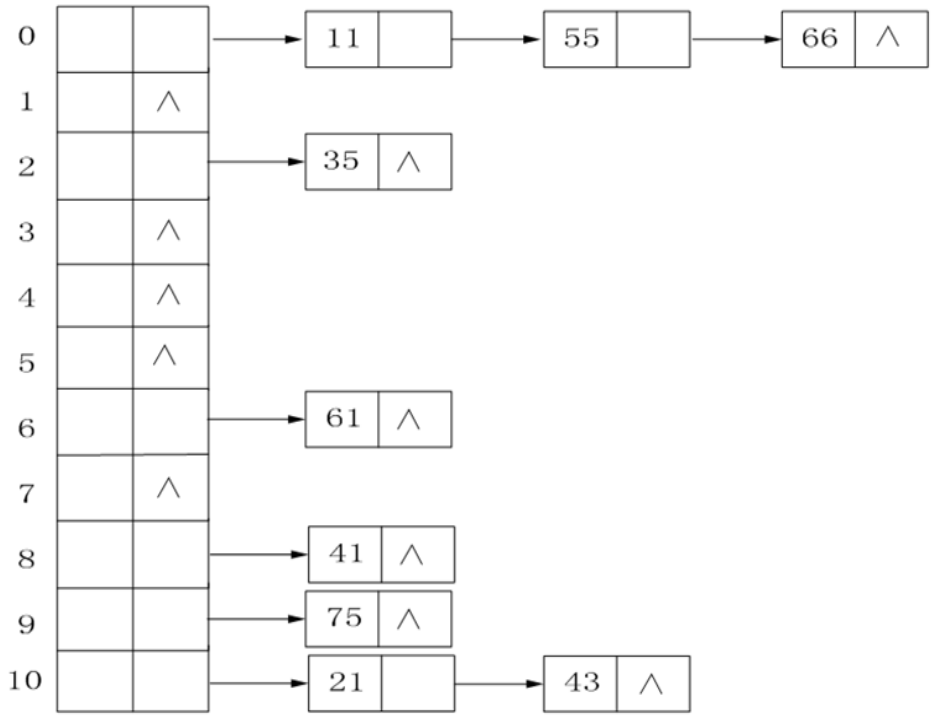
（1）链地址法（拉链法）

- 链地址法首先对关键字集合用某一个哈希函数计算它们的存放位置。
- 假设哈希表的地址空间是从0到m-1，则关键字集合中的所有关键字被划分为m个子集，具有相同地址的关键字归于同一子集。
- 将所有关键字为同一子集的记录存储在同一个线性链表中，各个链表的表头结点组成一个向量。

8.3.2 解决冲突的办法-开放地址法

（1）链地址法

- 【例8.8】将一组记录元素的关键字集合{11，21，35，41，43，55，61，66，75}，按照哈希函数Hash(x)=x%11，采用链地址法表示关键字在哈希表中的位置。
  - Hash(11)= Hash(55)= Hash(66)=0
  - Hash(35)=2
  - Hash(61)=6
  - Hash(41)=8
  - Hash(75)=9
  - Hash(21)=Hash(43)=10



（a）不占用表头结点空间的链地址法

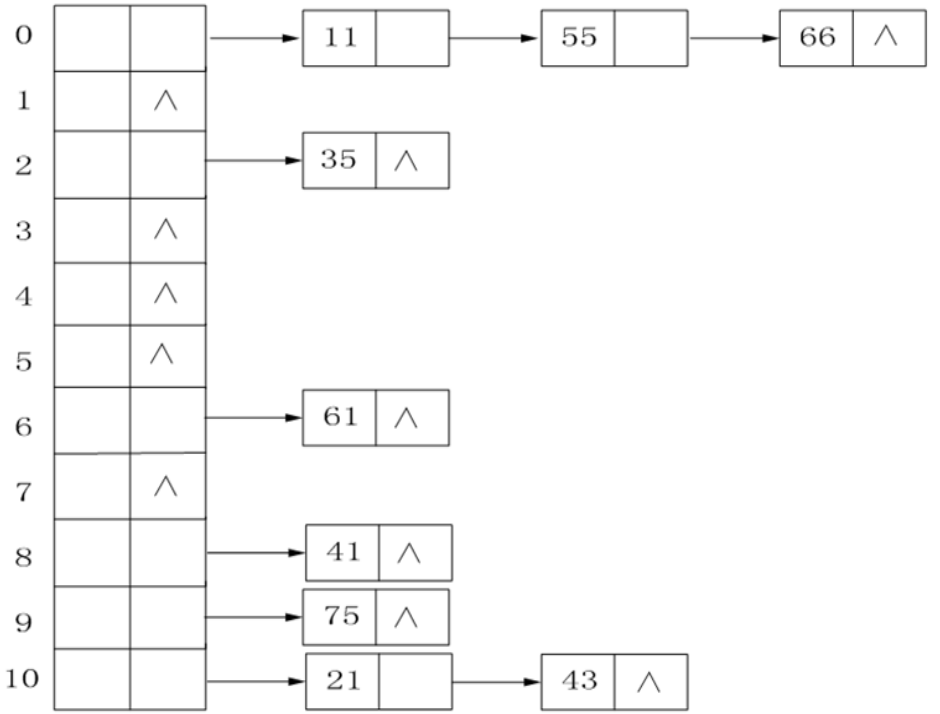
8.3.2 解决冲突的办法-开放地址法

（1）链地址法

- 平均搜索长度ASL：
  - 使用平均搜索长度ASL来衡量哈希方法的搜索性能，即用搜索成功情况下的平均比较次数来表示。
- 【例8.8】搜索成功的平均查找长度为：

$$ASL=\sum_{i=0}^{n-1}P_iC_i=\frac{1}{9}\sum_{i=1}^{9}C_i=\frac{1}{9}(1*6+2*2+3*1)=\frac{13}{9}\approx 1.44$$

思考：不成功查找的ASL计算？



（a）不占用表头结点空间的链地址法

8.3.2 解决冲突的办法-开放地址法

（2）公共溢出区法

- 通过使用另外的一维数组来存储发生冲突的元素，发生冲突的元素在溢出区内依次顺序存放。
- 假设使用的哈希函数为H(key)=key%7，需要插入的元素为8，12，6，10，16，23，36，29，存储结果如图8-5所示。

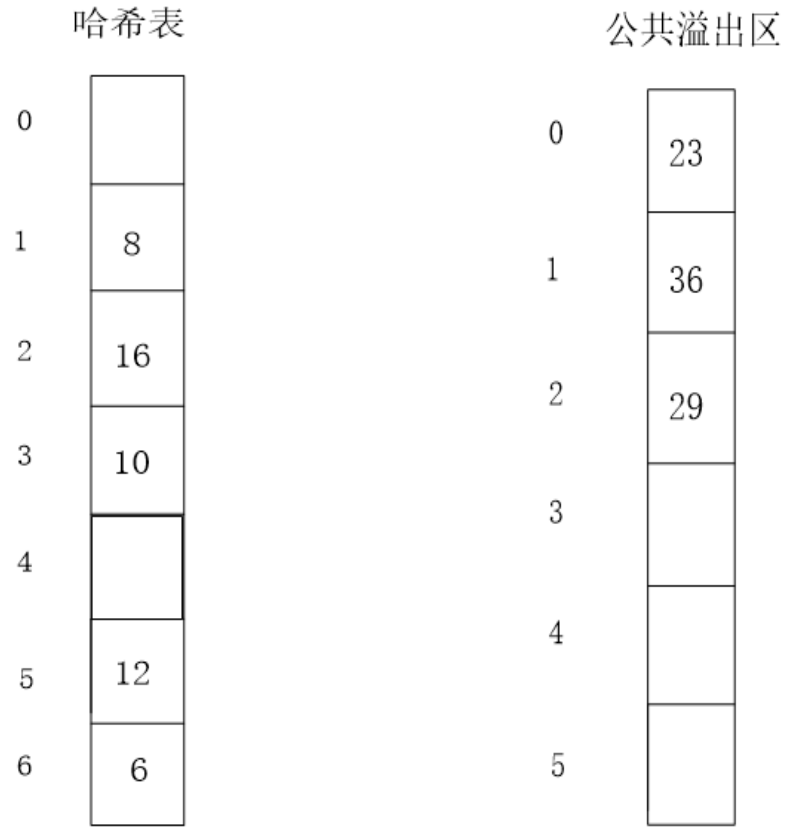


图 8-5 公共溢出区

★注：溢出区长度+1为失败的次数。

8.3.2 解决冲突的办法-封闭哈希法

（1）线性探测法（linear probing）

- 线性探测法从发生冲突的地址单元开始，依次探测下一个地址单元（把哈希表看成循环表，用表空间加1取模实现），直到遇到一个空闲单元或探测完所有地址单元时为止。
- 线性探测法用递推公式表示为：

$$\begin{cases} d_0=\text{Hash}(\text{R.key}) \\ d_i=(d_{i-1}+1)\%m \qquad (1\le i\le m-1) \end{cases}$$

8.3.2 解决冲突的办法-线性探测法

- 【例8.3】给出一组关键字集合{28，8，21，22，6，27}，已知哈希表的地址空间为[0..6]，采用哈希函数Hash(key)=key%7，线性探测法处理冲突，将关键字依次映射到哈希表中。
- 其中：
  - Hash(28)=Hash(21)=0
  - Hash(8)=Hash(22)=1
  - Hash(6)=Hash(27)=6

注：用线性探测法解决冲突时，容易产生“堆积”现象。

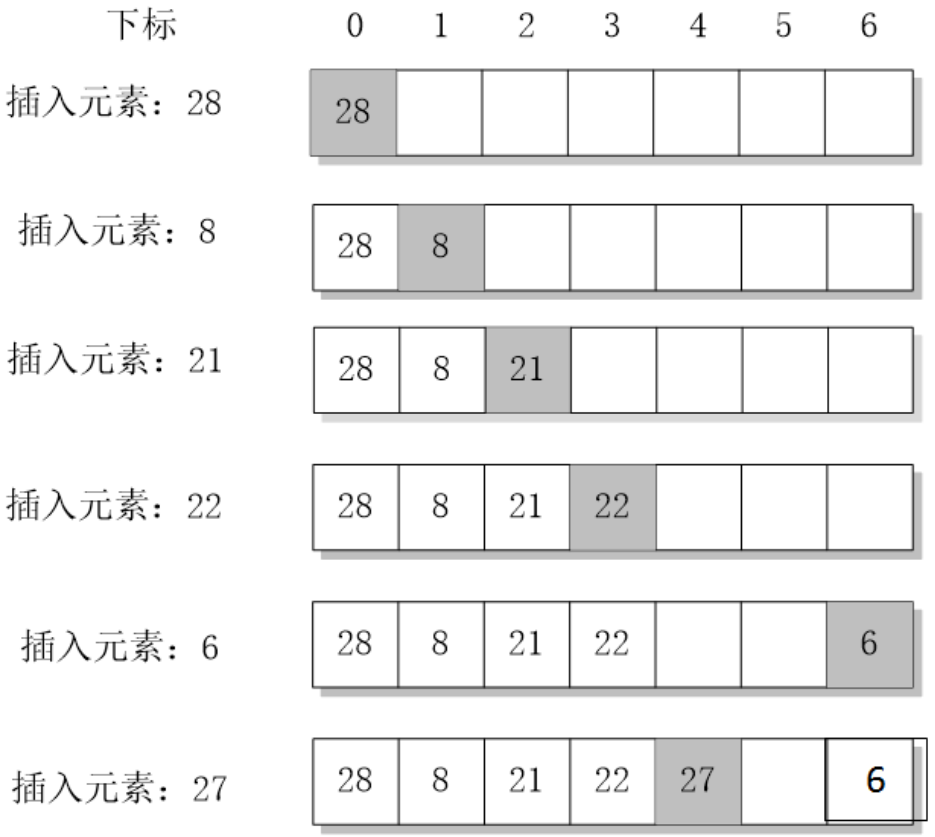


图 8-6 线性探测法及堆积问题

8.3.2 解决冲突的办法-线性探测法

- 堆积现象将造成不是同一子集的结点也处在同一个探测序列之中，产生冲突叠加效应，从而增加了探测序列的长度，即增加了查找时间。

表 8-1 哈希表构造结果及每个关键值的比较次数

下标	0	1	2	3	4	5	6
元素	28	8	21	22	27		6
比较次数	1	1	3	3	6		1

- 所对应查找成功的平均查找长度为：

$$ASL=\frac{1}{6}\sum_{i=1}^{6}C_i=\frac{1}{6}(1+1+3+3+6+1)=\frac{15}{6}=2.5$$

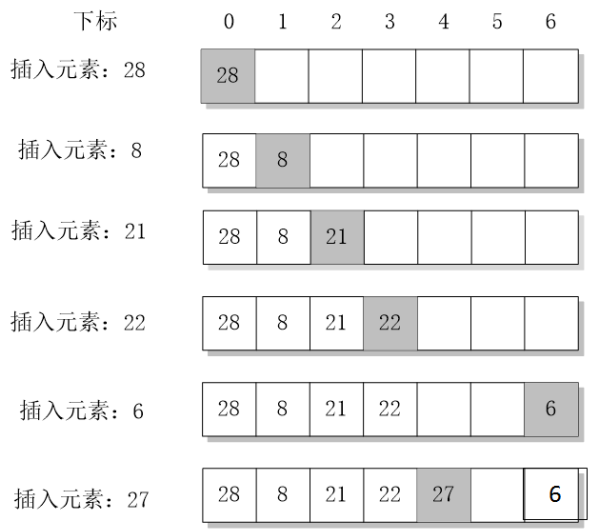


图 8-6 线性探测法及堆积问题

8.3.2 解决冲突的办法-封闭哈希法

（2）平方探测法（quadratic probing）

- 平方探测法也叫二次探测法。为了减轻线性探测法产生的局部“堆积”问题。
- 二次探测法采用如下探测方法：

  $h_0 = Hash(key)$

  $h_i = (h_0 + d_i) \% m$

- 其中：Hash(key)为哈希函数；

  m为哈希表长度。

  d_i为增量，$d_i=i^2$。增量序列为1^2，2^2，…，q^2。

  或者，d_i为增量，$d_i= \pm i^2$。增量序列为$1^2, -1^2, 2^2, -2^2, \cdots, q^2, -q^2$。

8.3.2 解决冲突的办法-平方探测法

- 【例8.4】对一组记录元素的关键字集合{28，8，21，22，6，27}，哈希函数为Hash(key)=key%7。利用二次探测法处理冲突，$d_i=i^2$，将关键字依次映射到哈希表中。

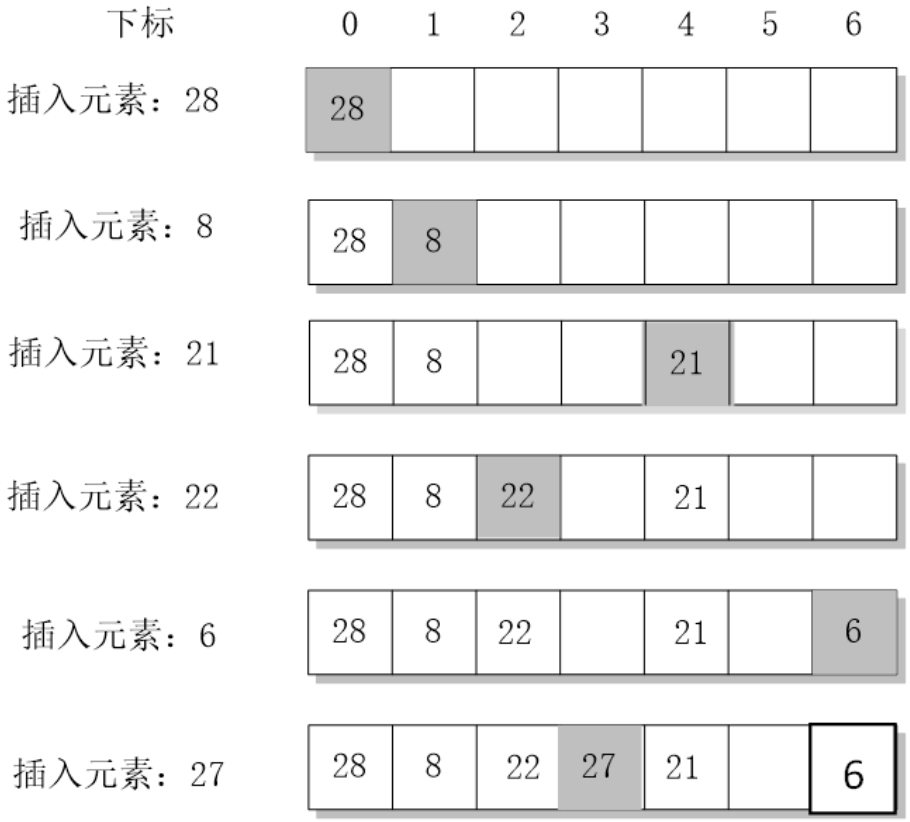


图 8-7 二次探测法

8.3.2 解决冲突的办法-平方探测法

表 8-2 哈希表构造结果及比较次数

下标	0	1	2	3	4	5	6
元素	28	8	22	27	21		6
比较次数	1	1	2	2	3		1

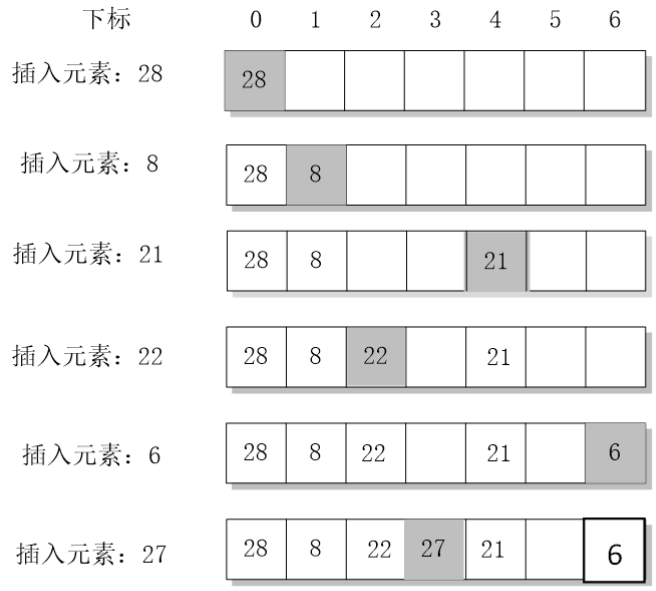


图 8-7 二次探测法

- 采用二次探测法处理冲突时的平均搜索长度为：

$$ASL=\frac{1}{6}\sum_{i=1}^{6}C_i=\frac{1}{6}(1+1+2+2+3+1)=\frac{10}{6}\approx 1.67$$

- 注意：二次探查法虽然部分解决了“堆积”的问题，但是二次探测的缺点是不能保证探测到整个哈希表的所有位置。

8.3.2 解决冲突的办法-封闭哈希法

- （3）随机探测法（也叫伪随机探测法）
  - 解决线性探测法造成的堆积问题的原则是让哈希表中的每一空位接收下一记录的概率尽可能相等。
  - 解决方法之一是采用伪随机探测法来解决冲突。此方法产生的再探测地址是从哈希表中随机选取的。
  - 随机探查方法的定义如下：
    - $h_0 = Hash(key)$
    - $h_i = (h_0 + d_i) \% m$，
    - 其中：i=1，2，…，k；
    - d_i为一组伪随机数序列(伪随机序列函数)。

8.3.3 哈希表元素删除

- 哈希表是一个动态结构的查找表。
  - 既可以通过哈希函数和冲突解决方案来实现记录元素的插入，可以进行记录元素的删除。
- 如果删除元素是具有冲突的元素，则这个元素被删除后就会破坏查找轨迹，解决方法：伪删除！
  - 伪删除是在删除一个元素时要在该位置设一个“墓碑标志”；
  - 在查找时跳过“墓碑标志”位置，即利用这个标志来查找与其冲突的元素存放位置，从而不会产生信息中断的情况。
  - 在插入时可利用“墓碑标志”位置来存放待插入的元素。

8.3.4 哈希表的实现

- 算法8.6：哈希表的类的声明以及部分函数实现

```
1.  struct Element {              //记录的定义
2.  KeyType key;              //关键字
3.  field otherdata;           //其他域
4.  };
5.  class HashList{               //用线性探查法处理冲突时哈希表类型的定义
6.  private:
7.      Element *HT;              //哈希表存储数组
8.      int  CurrentSize, MaxSize;       //当前哈希地址数及最大地址数
9.  public:
10.     HashList(int MaxSz=DefaultSize); //构造函数
11.     ~HashList(){ delete [] HT; }     //析构函数
12.     void ClearHashList();            //清空哈希表的函数
13.     bool Insert(Element item);     //哈希表HT中插入一个元素item
14.     int Search(KeyType x);      //从哈希表HT中查找元素，返回该元素的下标位置
15.     bool Delete(KeyType x);      //从哈希表HT中删除元素
16.     void Create(int num);          //建立哈希表HT
17. };
```

8.3.4 哈希表的实现

```
1.  HashList :: HashList(int MaxSz){    //构造函数
2.      HT=new Element[MaxSz];
3.      MaxSize=MaxSz;
4.      CurrentSize=0;
5.  //将哈希表HT中每一单元的关键字Key域都设为空标志
6.      for(int i=0; i<MaxSize; i++)
7.          HT[i].key=NullTag;
8.  }
9.
10. void HashList::ClearHashList(){    //清空哈希表的函数
11.      //将哈希表HT中每一单元的关键字key域都设置为空标志
12. for(int i=0; i<MaxSize; i++)
13.      HT[i].key=NullTag;
14. }
```

8.3.4 哈希表的实现

```
1.  bool HashList::Insert(Element item){//向哈希表HT中插入元素item
2.      int d=H(item.key);      //用哈希函数计算哈希地址
3.      int temp=d;             //用temp变量暂存哈希地址d
4.   while(HT[d].key != NullTag && HT[d].key != DeleteTag){
5.          //继续向后查找空元素位置或者被删除元素的位置
6.          d = (d+1) % MaxSize;    //假定采用线性探查法处理冲突
7.          if (d == temp) return false; //查找完所有位置后表示无法插入
8.      }
9.      HT[d] = item;    //将新元素插入到下标为d的位置
10.     return true;     //插入成功，返回真
11. }
```

8.3.4 哈希表的实现

算法8.7：从哈希表HT中查找元素，返回该元素的下标位置

```
1.  int HashList :: Search(KeyType x) {
2.      int d=H(x);      //计算哈希地址
3.      int temp=d;      //保存初始哈希地址到temp
4.   while(HT[d].key != NullTag) {
5.      //当哈希地址中的关键字域不为空则循环
6.          if(HT[d].key == x) return d; //查找成功返回下标d
7.      else d = (d+1) % MaxSize;
8.          if(d==temp) return -1;   //查找失败返回“-1”
9.      }
10.     return -1；   //遇到关键字为空，查找失败返回“-1”
11. }
```

8.3.4 哈希表的实现

算法8.8：从哈希表HT中删除元素

```
1.  bool HashList :: Delete(KeyType x) {
2.      int d = H(x);    //计算哈希地址
3.      int temp = d;    //保存哈希地址的初始值
4.  while(HT[d].key != NullTag) {    //不为空记录则循环
5.       if(HT[d].key == x) {
6.           HT[d].key = DeleteTag;  //设置删除标记
7.           return true;
8.           } else d = (d+1)%MaxSize;  //继续向后查找被删除的元素
9.           if(d==temp) return false;  //循环一圈后返回假，删除失败
10.     }
11.     return false;   //遇到空关键字，没有找到被删除元素，删除失败
12. }
```

8.3.5 哈希表的性能分析

- 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；否则，查找效率就低。
  - 所以，影响产生冲突多/少的因素也就是影响查找效率的因素。
- 影响产生冲突多少有以下3个因素：
  - 哈希函数是否均匀；
  - 解决冲突的方法是否合适；
  - 哈希表的装填因子。装填因子 $\alpha = n/m$，其中n为关键字个数，m为哈希表的表长。

8.3.5 哈希表的性能分析

- 实际上，哈希表的平均查找长度是装填因子α的函数，只是不同处理冲突的方法有不同的函数。用不同的方法处理冲突时哈希表的平均搜索长度如表8-4所示。

表 8-4 各种方法处理冲突时的平均搜索长度

处理冲突的方法		平均搜索长度 ASL	
		搜索成功	搜索不成功
开放哈希法	链地址法	$1+\frac{1}{\alpha}$	$\alpha+e^{-\alpha}\approx\alpha$
封闭哈希法	线性探测法	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
	二次探查法	$-\frac{1}{\alpha}\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
	伪随机探查法		

- 一般情况下，哈希表的装填因子α表明了哈希表中的装满程度。
  - α越大，说明哈希表越满，在插入新元素时发生冲突的可能性就越大。
- 希望哈希表有一定的空余量，一般装填因子小于0.8。

单选题 1分

设置

查找表32,45,18,77,5,23,44,19,7,3,哈希函数为H(key)=key %5,采用链地址法解决冲突的ASL(成功）=（　　），采用表长为11的线性探测地址法的ASL(成功）=（　　）

A 18/10，32/10

B 18/5，31/10

C 18/5，32/10

D 18/10，31/10

提交

单选题 1分

设置

以下关于哈希查找的叙述中错误的是（ ）。

A 哈希函数H(k)=k MOD p，p通常取小于等于表长的素数

B 哈希函数选得好可以减少冲突现象

C 用线性探测法解决冲突易引起堆积现象

D 用拉链法解决冲突易引起堆积现象

提交

单选题 1分

设置

以下关于哈希查找的叙述中正确的是（ ）。

A 哈希表的装填因子等于表中填入的记录数除以哈希表的长度

B 哈希表在查找成功时的平均查找长度仅仅与表长有关

C 采用拉链法解决冲突时，查找一个元素的时间是相同的

D 哈希查找中不需要任何关键字的比较

提交

单选题 1分

设置

为提高哈希（Hash）表的查找效率，可以采取的正确措施是（ ）。
Ⅰ.增大装填（载）因子
Ⅱ.设计冲突（碰撞）少的哈希函数
Ⅲ.处理冲突（碰撞）时避免产生堆积（堆积）现象

A 仅Ⅱ

B 仅Ⅰ、Ⅱ

C 仅Ⅱ、Ⅲ

D 仅Ⅰ

提交

8.4 索引

- 索引就是把一个关键字与他对应的记录相关联的过程
- 索引结构通常由数据表和索引表两部分组成
  - 数据表用于存储数据元素信息
  - 索引表用来记录关键字与记录存储地址之间的对照表。
    - 索引表中的每个元素称为索引项
    - 每个索引项至少应包含关键字和其对应的记录在存储器中的位置等信息
    - 索引项的一般形式为<关键字，指针>，
      - 其中指针指向数据表中的包含该关键字的完整记录。
- 记录的存储可以是无序的，但索引项是有序的。

8.4 索引

- 索引技术是组织大型数据库以及磁盘文件的一种重要技术。
- 索引按照结构可以分为线性索引，树形索引和多级索引。
  - 所谓线性索引就是将索引项集合组织为线性结构，也称为索引表。

8.4 线性索引-线性索引（linear index）

- 线性索引的索引文件被组织成一组简单的<关键字，指针>对的序列。
  - 在索引文件中，按照关键字的顺序进行排序，指针指向存储在磁盘上的文件记录的起始位置或者主索引中关键字的起始位置。
  - 可用顺序查找或折半查找实现记录的定位。

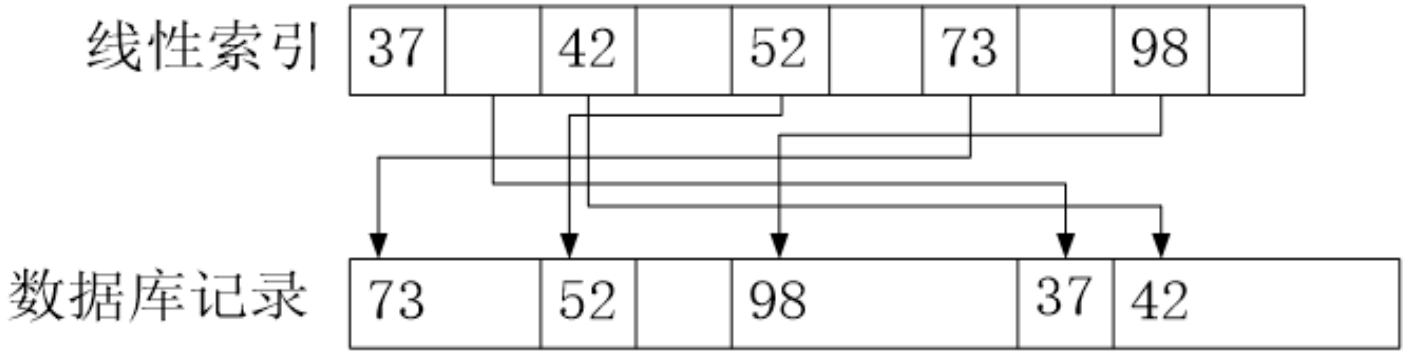


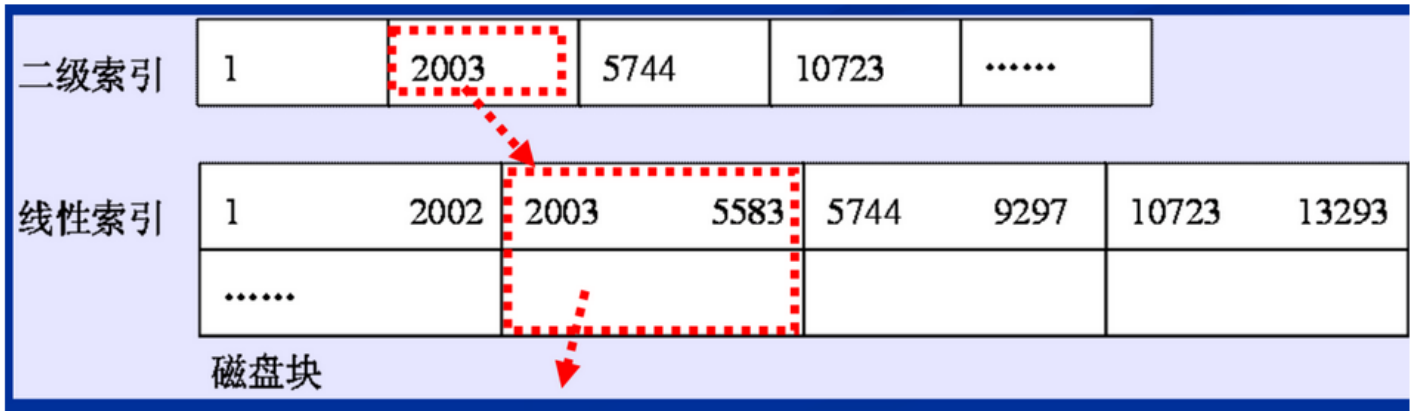
图 8-7 线性索引

8.4 线性索引-线性索引（linear index）

- 线性索引的优点：
  - 索引文件中记录是定长的，而且是按关键字有序，所以可以利用折半查找法来进行快速查找，从而提高检索效率。此外，也可减少磁盘文件的读写次数。
- 线性索引的缺点：
  - 假如存在大量的数据记录，那么可能由于索引文件太大而不能存储到内存当中。这样在检索过程中，需要多次访问磁盘，降低检索效率。
  - 注：可以通过建立二级索引来改善该问题（如下文）。

8.4 线性索引-线性索引（linear index）

- 二级线性索引文件中同样是<关键字，指针>序列，
  - 二级索引表中的每条记录都对应于一个一级索引表的磁盘块，
  - 二级索引表中的关键字值与对应的一级索引表的磁盘块中第一条记录的关键字值相同，而且指针指向相应一级索引表的磁盘块的起始位置。
- 例：
  - 检索关键码为2555的记录：二级索引找≤2555最大关键码所在记录2003；沿指针找线性索引文件的磁盘块读入内存；二分法找记录在磁盘上位置读入所需记录，完成检索



8.4 线性索引-分块索引

- 分块索引是指块间有序，即数据块之间按每个数据块中的最大（最小）关键字有序。索引表中存储块中的最大值或最小值。要求后一块的关键字大于（或小于）前一块的关键字。
- 分块索引是顺序存储的，索引表为n/s，块内搜索s。
  - n为总的记录数，s为块内元素数，则复杂度为O(n/s+s)。

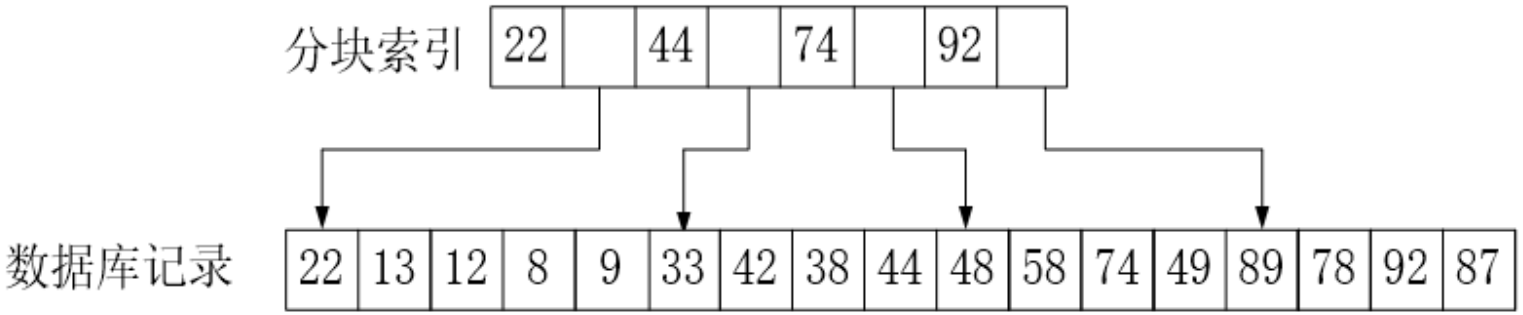


图 8-9 分块索引

单选题 1分

设置

以下适合用分块查的数据集是（ ）。

A 数据分成若干块，块内数据不必有序，但块间必须有序

B 数据分成大小相等的若干块，块内数据有序

C 数据分成若干块，块内数据必须有序，块间不必有序

D 数据分成若干块，每块（除最后一块外）中数据个数需相同

提交

单选题 1分

设置

查找表如下分组(3,7,1,8,15),(22,43,18,45),(60,58,82,77,62),(90,88,99,100)，则索引表为（　　）,查找58需要比较（　　）次

A (15,1),(45,6),(82,10),(100,15),(NULL,19)　5

B (22,1),(60,6),(88,10),(101,15)　7

C (15,1),(45,6),(82,10),(100,15)　7

D (15,1),(45,6),(82,10),(100,15)　5

提交

8.5 树形索引

- B树是目前使用最为广泛的索引方法，树结构更新之后仍能自动保持平衡，且适于按块存储。
- 为了更好地理解B树构造过程，此处先介绍一种叫2-3树的结构。实际上，2-3树是B树的特殊情况，也就是3阶B树。

8.5.1 2-3树

- 2-3树的定义：

  1) 每个结点上的子树个数可以是2-3个。每个结点包含一个或两个关键字的值；

  2) 每个内结点或有一个关键字两个子树，或有两个关键字三个子树；

  3) 所有的叶子结点在2-3树的同一层出现。即高度是一样的。

8.5.1 2-3树

- 2-3树更新代价更小。具有N个关键字的2-3树的高度比相应的二叉查找树的高度要低很多。

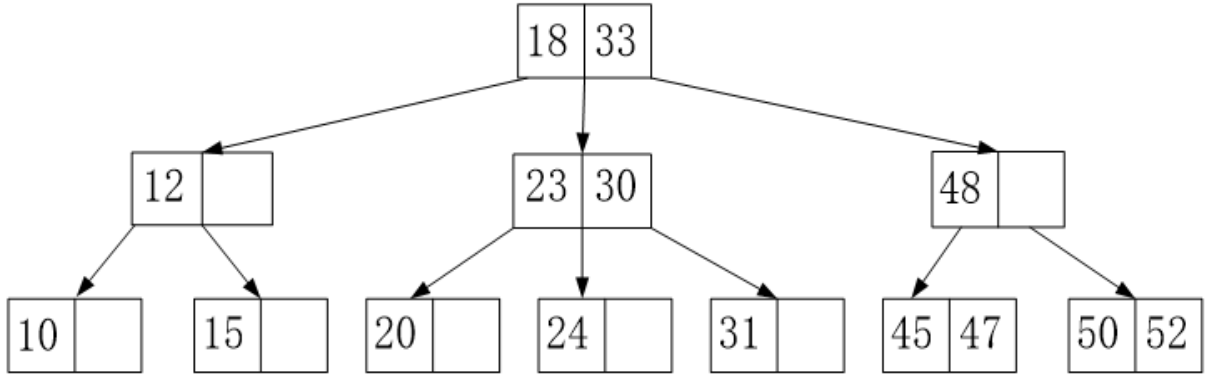


图 8-10 2-3 树

- 在2-3树中：

1）左边孩子中的结点值小于父结点中第一个关键字的值；

2）中间孩子中的结点值大于父结点中第一个关键字的值，且小于第二个关键字的值；

3）右边孩子中的结点值大于父结点中第二个关键字的值。

8.5.1 2-3树

- 2-3树的查找过程：
  - 从根结点开始，按照2-3树定义中关键字关系，逐级向下确定待查找关键字。类似于二叉查找树，要么中途比较相等找到成功，要么直到叶子层失败。
  - 在图8-10中查找15时，先比较18不等且小于，则走左子树；再跟12比较不等且大于，则走中子树；跟15比较相等查找成功。
  - 又比如查找45，在根结点分别与18和33不等且大于，则走右子树；再跟48比较不等且小于，则走左子树；再跟45比较相等成功。

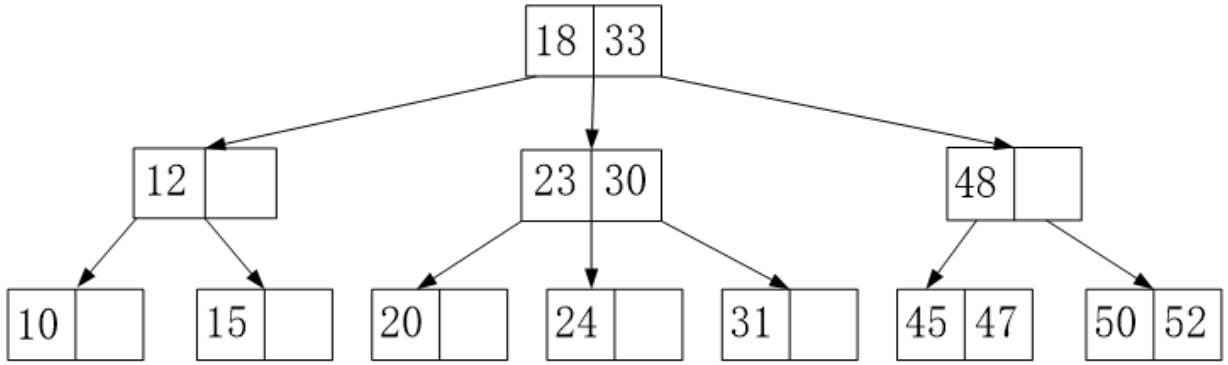


图 8-10 2-3 树

8.5.1 2-3树

- 2-3树的插入的三种情况：

（1）终端结点中有空位置，直接插入待插结点。在图8-10中插入14时，由于14的插入位置处有一个空位置，所以直接插入14。如图8-11（a）所示。

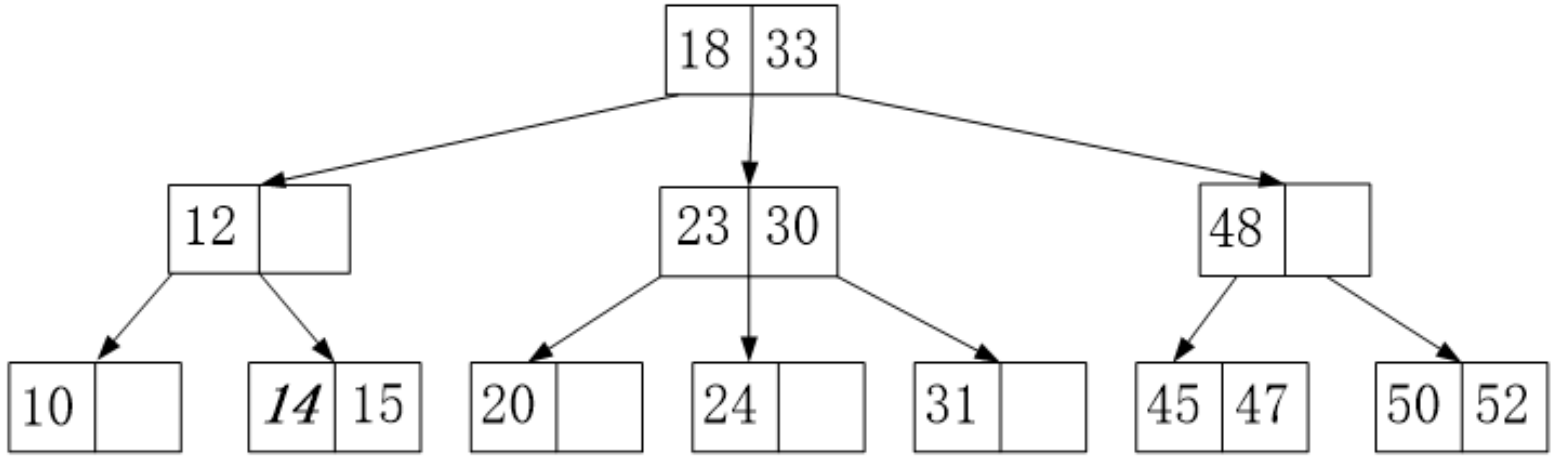


图 8-11（a）插入 14

8.5.1 2-3树

（2）叶子结点中已满，此时需要将叶子结点中的中间元素提升到父结点中，并且叶子结点分裂为两个结点。如果此时父结点中有空位置，则直接插入待插元素，否则继续分裂。在图8-11(a)中插入55。

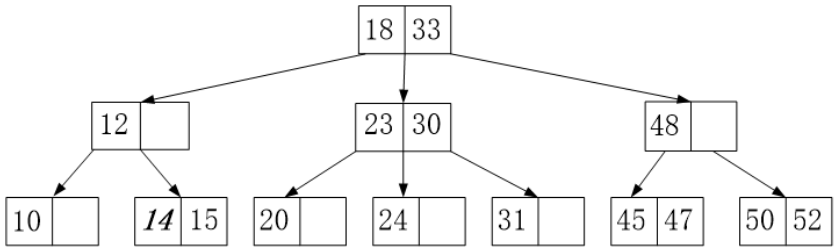


图 8-11（a）插入 14

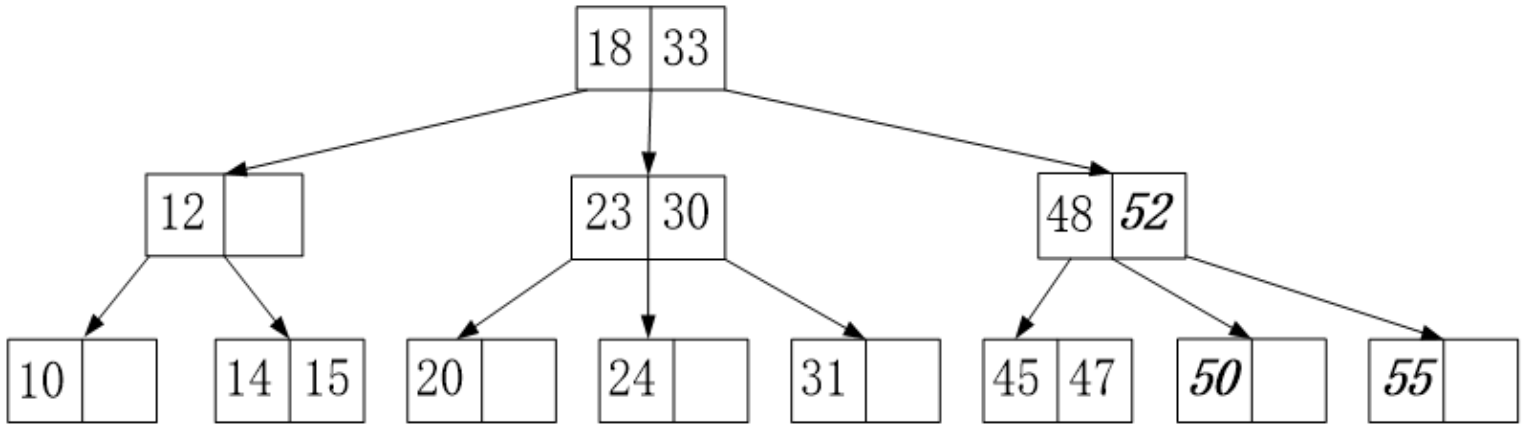


图 8-11（b）插入 55

8.5.1 2-3树

（3）**根结点分列导致树高度增加**。假如在叶子结点插入一个元素后，使得上面的结点不断地分裂，最后使得根结点也被分裂，此时2-3树的高度加1。在图8-10中插入21和19后，形成图8-11(c)。

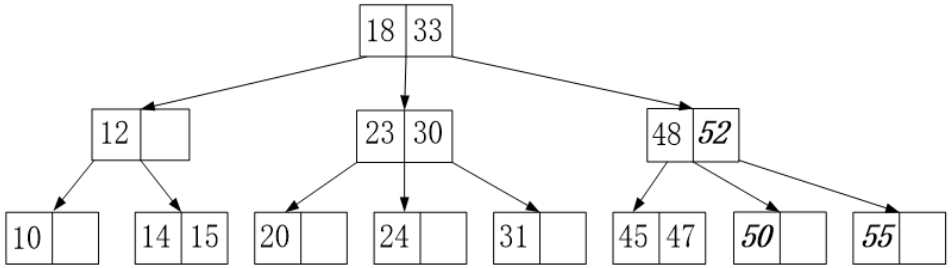


图 8-11（b）插入 55

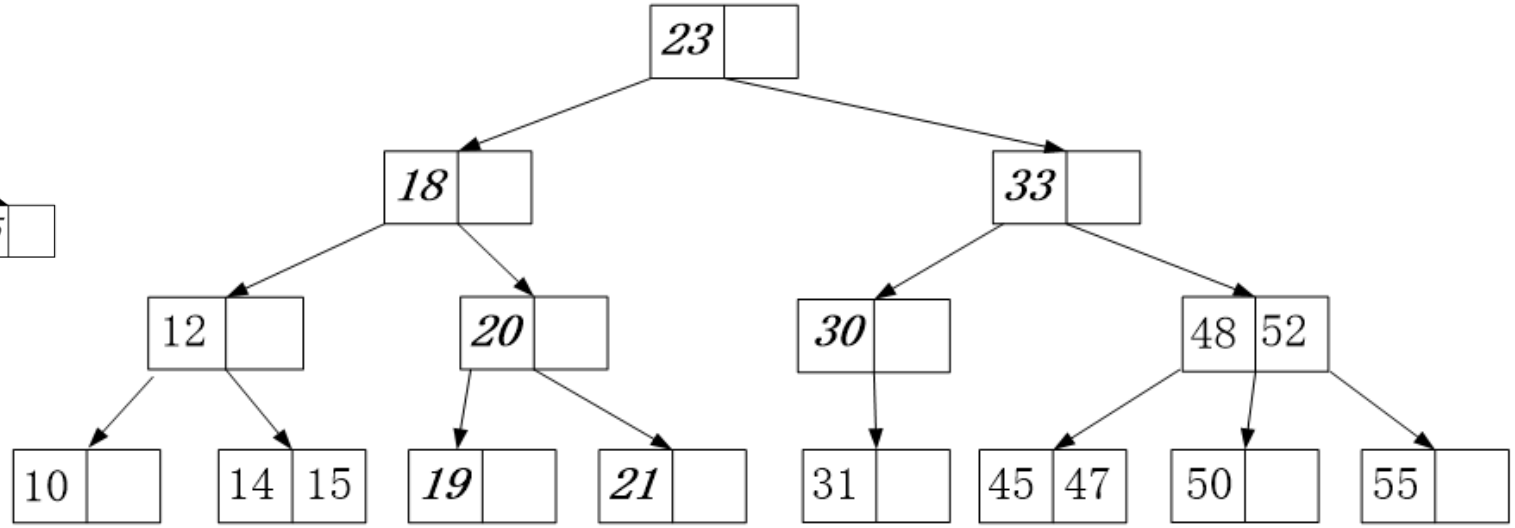


图 8-11（c）依次插入 21 和 19 后，2-3 树的最终示意图

8.5.2 B树

- 2-3树是B树的一种特殊情况。把树中结点最大的孩子数目称为B树的阶，通常记为m。
  - B树是一种平衡的多路搜索树，它在文件系统中非常有用。
- B树的定义如下：
- 一棵m阶B树，或者是空树，或者是满足下列性质B的树：

(1) 树中每个结点至多有m棵子树，即每个结点中至多含有m-1个关键字；

（两棵子树指针夹着一个关键字）

(2) 如果根结点不是叶子结点，则根结点至少含有两棵子树；

（至少一个关键字）

(3) 除根结点和叶子结点之外的所有非终端结点，每个结点至少有[m/2]棵子树，有[m/2] -1个关键字；

8.5.2 B树

（4）所有的非终结点中包含下列信息数据：

$(n,P_0,K_1,P_1,K_2,P_2,\ldots,K_n,P_n)$

其中：$K_i(i=1,\ldots,n)$为关键字，并且$K_i<K_{i+1}(i=1,\ldots,n-1)$；$P_i(i=0,\ldots,n)$为指向子树根结点的指针，且指针$P_{i-1}$所指子树中所有结点的关键字均小于$K_i(i=1,\ldots n)$，$P_n$所指子树中所有结点的关键字均大于$K_n$，$n(\lceil m/2\rceil-1\leqslant n\leqslant m-1)$为关键字的个数（或n+1为子树的个数）；

（5）所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

8.5.2 B树

- 在B树中，每个结点内的关键字从小到大有序排列。
  - 如图8-12是一棵4阶的B树，所有终端结点都在同一层上。该B树的每个结点（除根和终端结点之外）的孩子结点的个数在[4/2]和4之间，因此每个非终端结点可以包含1或2或3个关键字；根结点包含一个关键字、有两个孩子结点；在每个结点内的关键字都是按由小到大的顺序排列的。

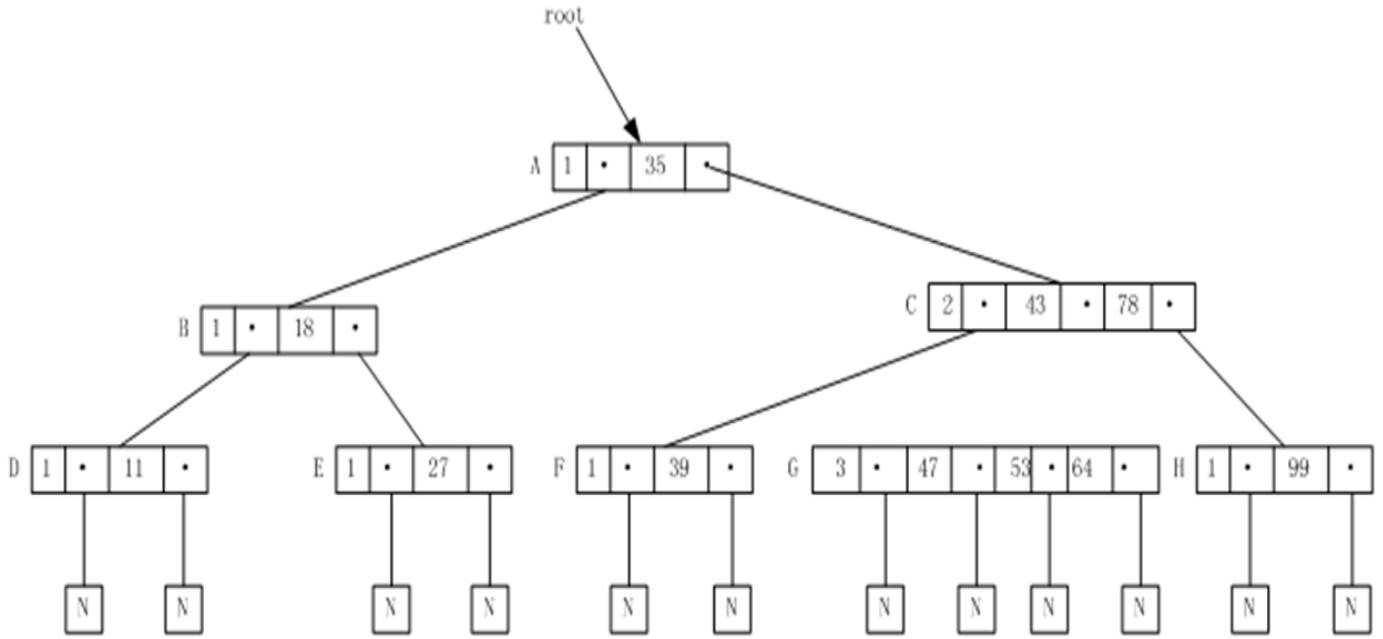


图 8-12 一棵 3 阶的 B.树

8.5.2 B树

算法8.9：B-树的抽象数据类型

```
1.  ADT  MBBTree{
2.  Data；    //Data是具有相同特性的数据元素的集合
3.  Opreation:
4.        MB_Tree();                    //构造函数
5.        int SearchMBTree(KeyType K);   //查找
6.        bool InsertMBTree(KeyType K, int addr);  //插入
7.        bool DeleteMBTree(KeyType K);      //删除
8.        int DepthMBTree();              //求深度
9.        int CountMBTree();              //统计所有结点数
10. }//ADT MBTree
```

8.5.2 B树

用链式存储结构来存储B树，其结点和类声明如下所示：

```
1.  struct MBNode{
2.      int keynum;              //关键字个数域
3.      MBNode *parent;          //指向父结点的指针域
4.      KeyType key[n+1];        //保存n个关键字的域，下标0位置未用
5.      MBNode *ptr[n+1];        //保存n+1个指向子树的指针域
6.      int recptr[n+1];         //保存每个关键字对应记录的存储位置
7.  };
```

8.5.2 B树

```
1.  class MBTree{
2.  private:
3.     MBNode *MT;
4.  public:
5.     MBTree() { MT=NULL; }
6.     int SearchMBTree(KeyType K);  //查找关键字为K的对应记录的存储位置
7.     bool InsertMBTree(KeyType K,int num);  //插入索引项（K,num,NULL)
8.     bool DeleteMBTree(KeyType K);  //删除关键字为K的记录
9.     int DepthMBTree(MBNode *T);      //求深度
10.    int DepthMBTree();
11.    int CountMBTree(MBNode *T);      //统计所有结点数
12.    int CountMBTree();
13.    void PrintMBTree(MBNode *T);
14.    void PrintMBTree();
15. };
```

8.5.2 B树-查找分析

- B树的查找过程是一个在结点内查找和顺着某一条路径向下一层查找交替进行的过程。
  - B树的查找时间复杂度与B树的阶数m和B树的高度h有关。
  - 在B树上进行查找时，查找成功所需的时间取决于关键字所在的层次，查找不成功所需的时间取决于树的高度。
- 在含有N个关键字的B树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过 $\log_{\lceil m/2 \rceil}((N+1)/2)+1$。

8.5.2 B树

算法8.10：B树的查找算法

```
1.  int MBTree::SearchMBTree(KeyType K){
2.      //从树根指针为MT的B树上查找关键字为K的对应记录的存储位置
3.  int i;
4.      MBNode *p=MT;
5.  while(p != NULL){   //从树根结点起依次向下一层查找
6.              i=1;              //用i表示待比较的关键字序号，初值为1
7.              while(K > p->key[i]) i++; //用K顺序同结点内关键字进行比较
8.       if (K == p->key[i])
9.                  return p->recptr[i]; //查找成功返回记录的存储位置
10.             else p=p->ptr[i-1];     //继续向子树查找
11.     }
12.     return -1;              //查找失败返回-1
13.  }
```

8.5.2 B树-插入分析

- B树的生成也是从空树开始，逐个插入关键字而得。但是由于m阶B树结点中的关键字个数必须≥⌈m/2⌉-1，因此，每次插入一个关键字不是在树中添加一个叶子结点，而是首先在最低层的某个非终结点中添加一个关键字，如果该结点的关键字个数不超过m-1，则插入成功，否则需要产生结点的“分裂”。

8.5.2 B树-插入分析

- 向B树中插入关键字的过程分两步完成：

  （1）定位。利用前述的B树的查找算法查找关键字的插入位置，即最底层中某个非叶子结点。（规定一定是插入在最底层的某个非叶子结点内）。

  （2）插入。判断该结点是否还有空位置。即判断该结点的关键字总数是否满足n<m-1。

  - 如果满足，则说明该结点还有空位置，直接把关键字k插入到该结点的合适位置上。
  - 若不满足，说明该结点已没有空位置，需要将结点分裂成两个。

8.5.2 B树-插入分析

- 分裂的方法：

  ① 生成一个新结点。

  ② 把原结点上的关键字和插入的关键字k排序后，从中间位置把关键字序列（不包含中间位置的关键字）分成两部分。

  ③ 左部分所含关键字放在旧结点中，右部分所含关键字放在新结点中，中间位置的关键字连同新结点的存储位置插入到双亲结点中。

- 也就是说设结点p中已经有m-1个关键字，当再插入一个关键字后结点中的状态为：$(m, p_0, k_1, p_1, k_2, p_2, \ldots, k_m, p_m)$

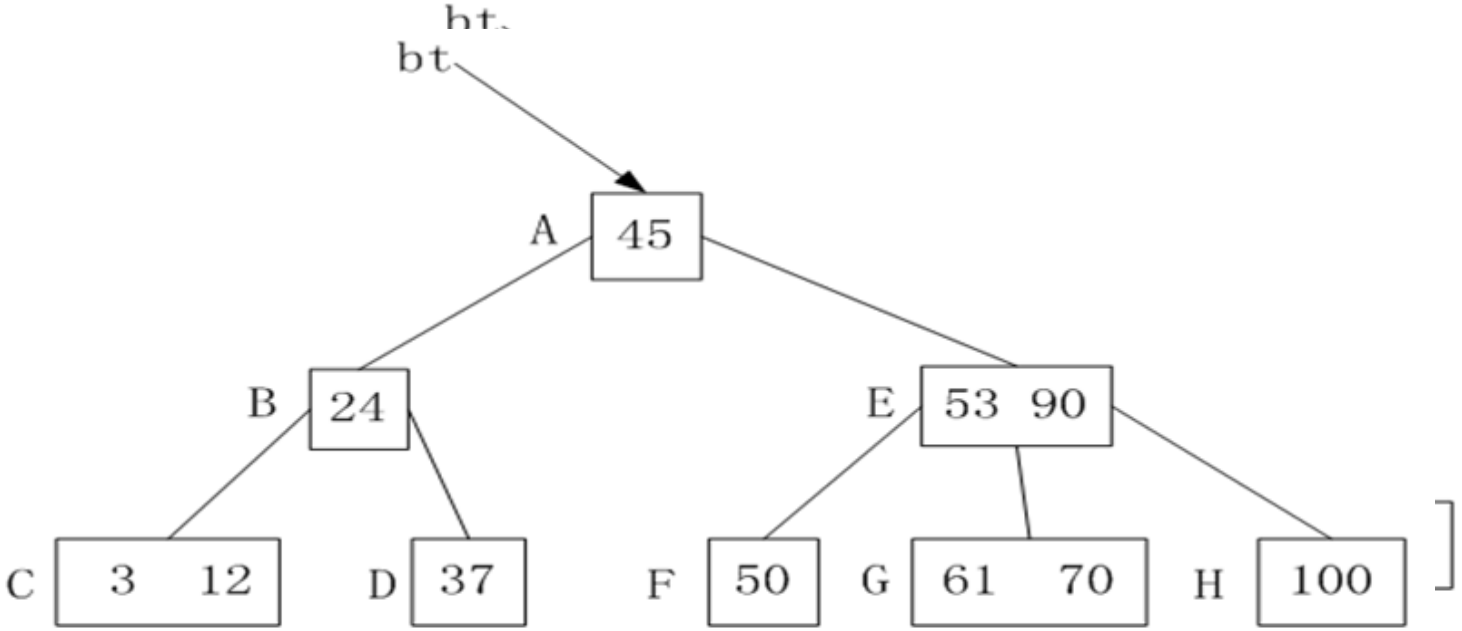
  - 其中$K_i<K_{i+1}$,$1 \leqslant i<m$。

8.5.2 B树-插入分析

- 除$K_{\lceil m/2 \rceil}$结点之外，把p结点分为左右两部分，左边的部分包含信息为$(\lceil m/2 \rceil-1, P_0, K_1, P_1, \cdots, K_{\lceil m/2 \rceil-1}, P_{\lceil m/2 \rceil-1})$，留在原p结点中；右边的部分包含信息为$(m-\lceil m/2 \rceil, P_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil+1}, P_{\lceil m/2 \rceil+1}, \cdots, K_m, P_m)$，存储到一个新建结点中，用指针q指向它；
- 位于中间的关键字$K_{\lceil m/2 \rceil}$与指向新结点的指针q生成一个新二元组$(K_{\lceil m/2 \rceil}, q)$按顺序插入到双亲结点当中。
- 如果双亲结点的关键字个数也超过m-1，则要再分裂，再往上插。直至这个过程传到根结点为止。

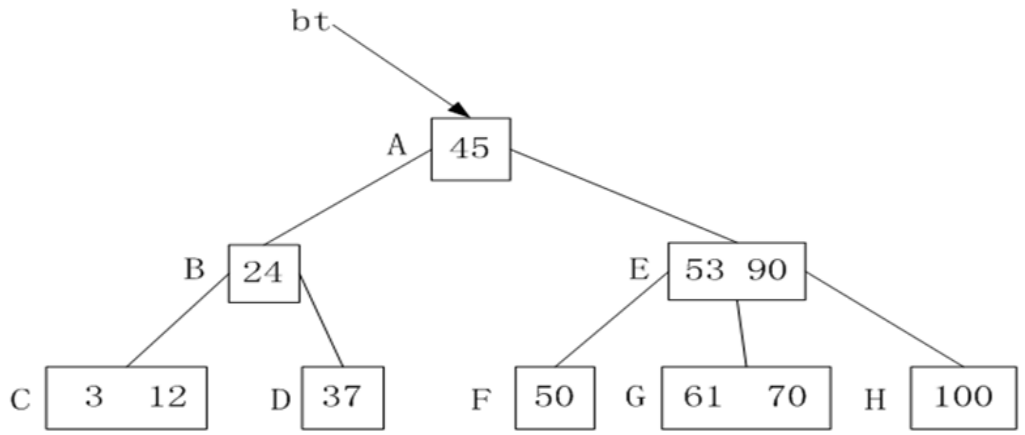
8.5.2 B树-插入分析 例

- 图8-13(a)所示为3阶的B树（图中略去叶子结点），假设需要依次插入关键字30，26，85和7。



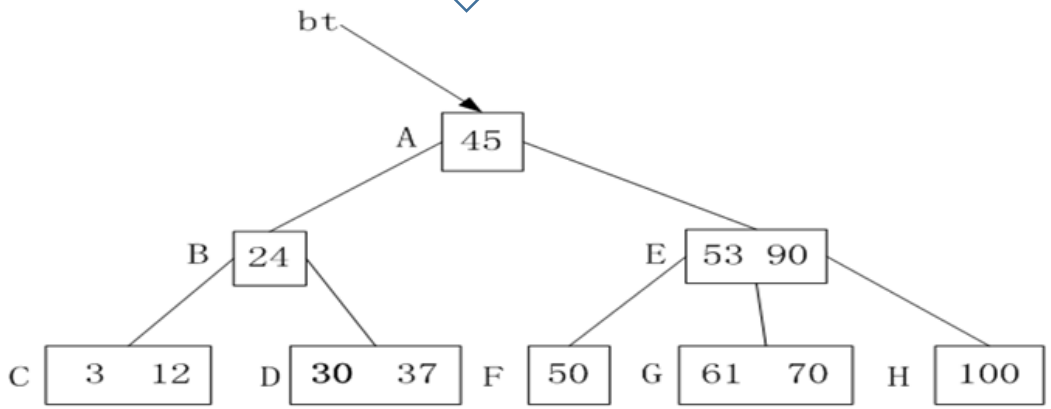
(a)

8.5.2 B树-插入分析　例



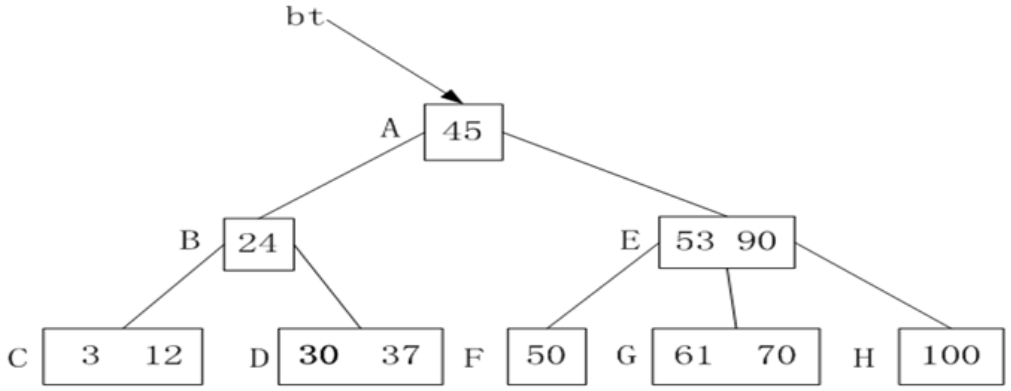
(a)

插入30之后



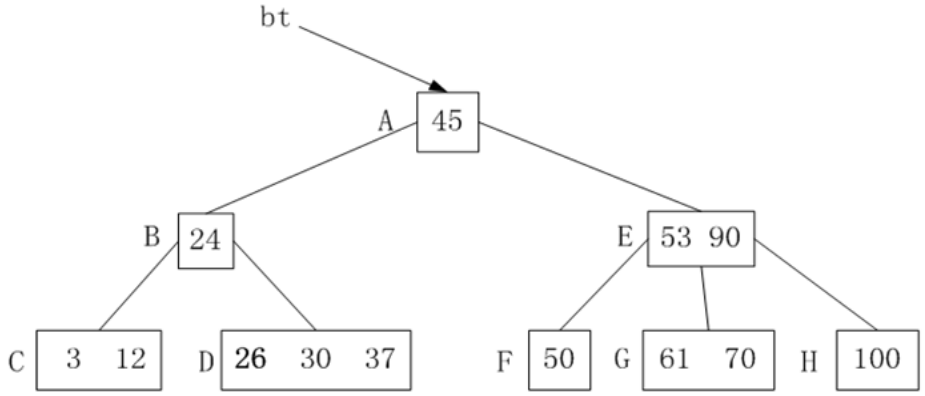
(b)

8.5.2 B树-插入分析 例

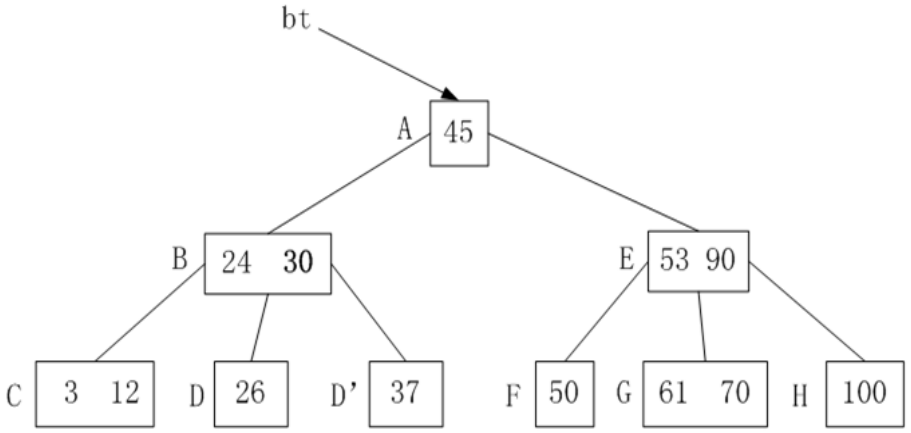


(b)

（c）-（d）插入26之后

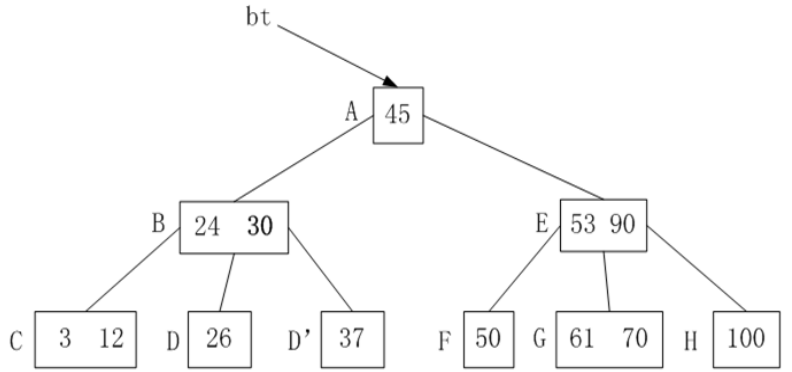


(c)



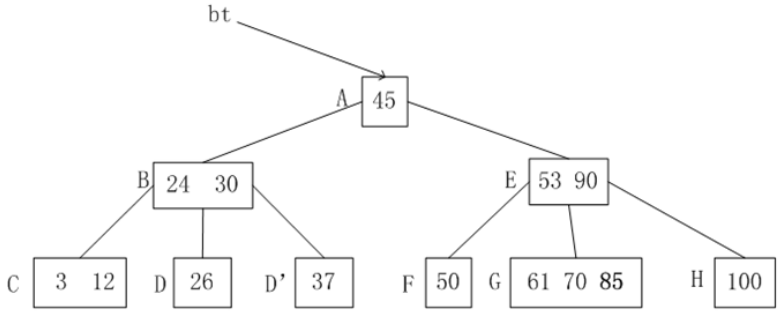
(d)

8.5.2 B树-插入分析 例

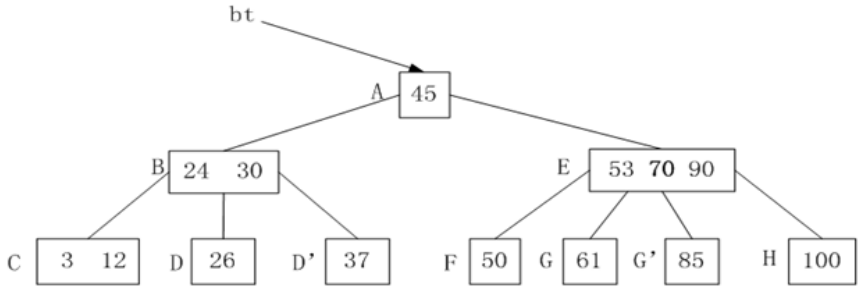


(d)

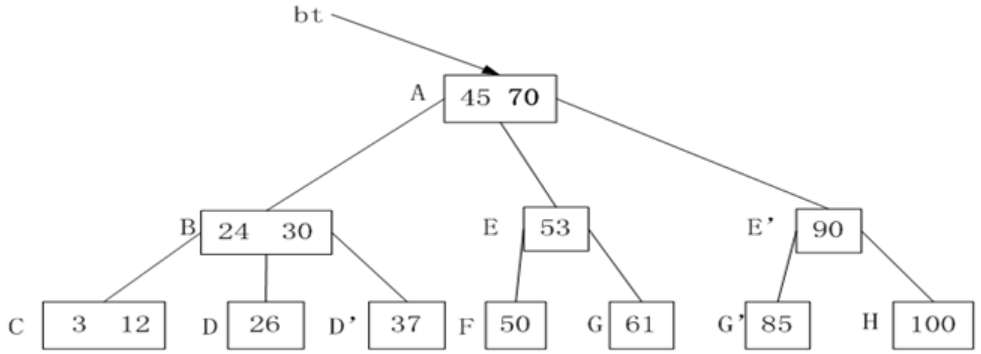
（e）-（g）插入85之后



(e)

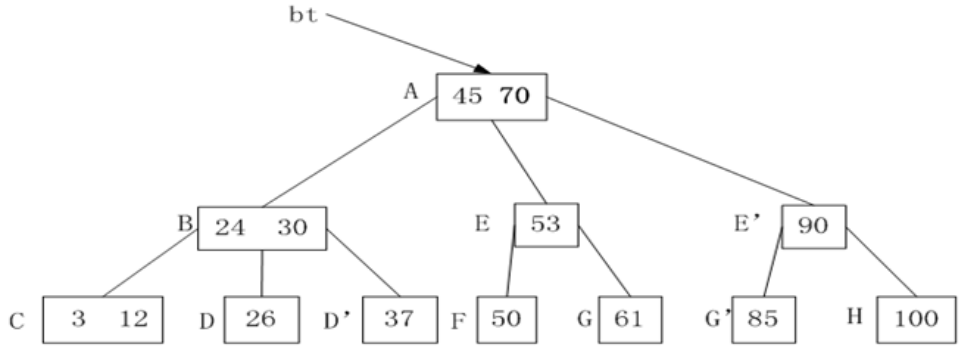


(f)



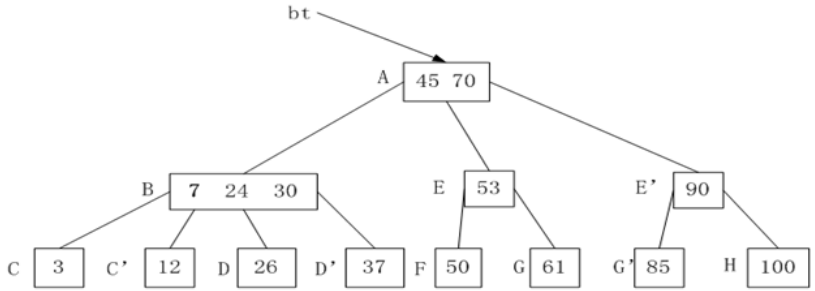
(g)

8.5.2 B树-插入分析 例

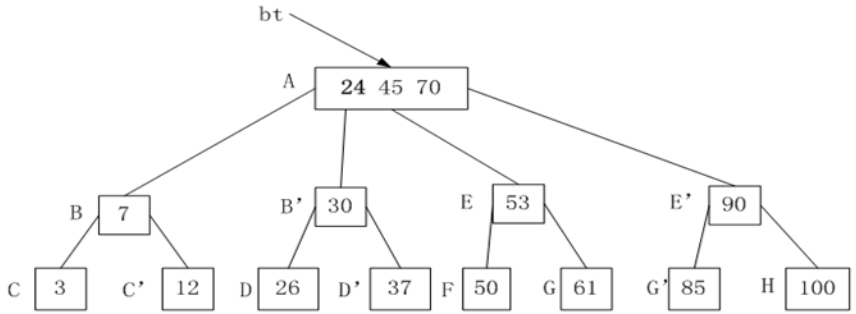


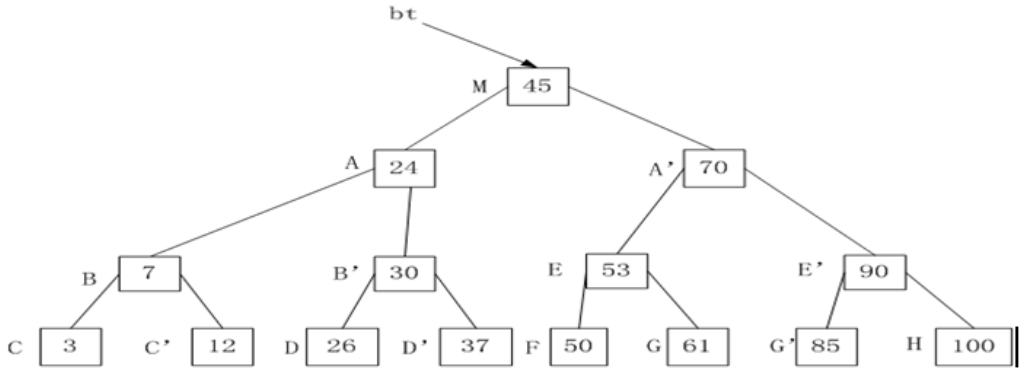
(g)

（h）-（j）插入7之后



(h)





(j)

主观题 10分

设置

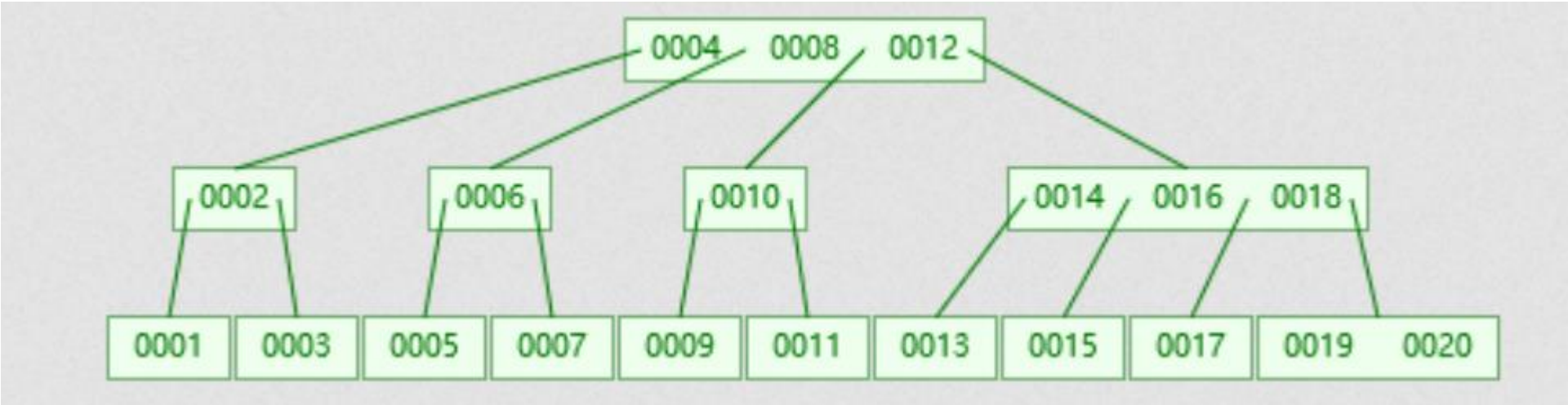
对于4阶或5阶B-树，依次插入关键字1~20，画出其相应结构。

正常使用主观题需2.0以上版本雨课堂

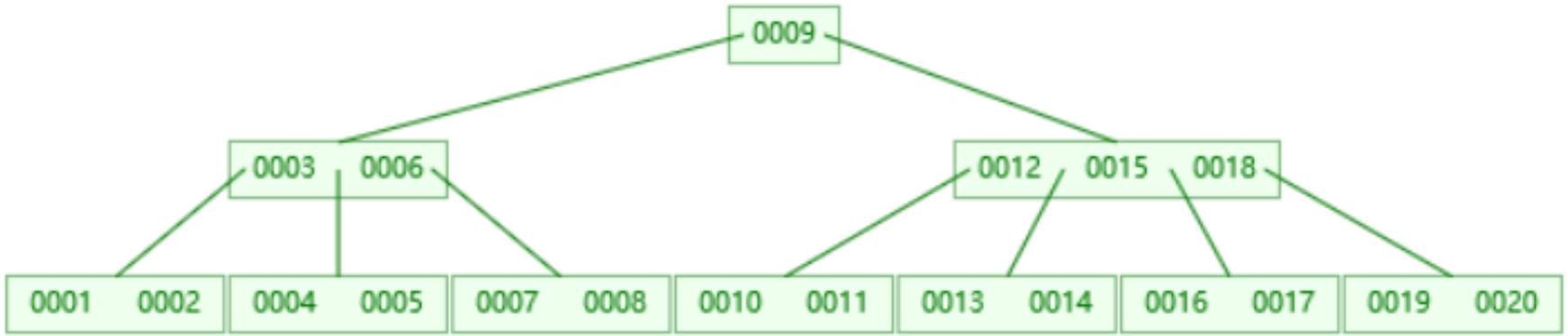
作答

8.5.2 B树-插入分析　练习

- 4阶B-树 分别插入1-20后 的结构



- 5阶B-树 分别插入1-20后 的结构



8.5.2 B树-插入 代码

算法8.11：B树的插入算法

```
1.  bool MBTree :: InsertMBTree(KeyType k, int num) {
2.  //插入索引项（k, num, NULL)---关键字、记录地址、双亲指针
3.      //当B树为空时的处理情况
4.  if(MT==NULL){
5.          MT=new MBNode;
6.          MT->keynum=1; //关键字个数
7.          MT->parent=NULL;
8.          MT->key[1]=k;
9.          MT->key[2]=MaxKey;
10.         MT->recptr[1]=num;
11.         MT->ptr[0]=MT->ptr[1]=NULL;
12.         return true;
13. }
```

8.5.2 B树-插入 代码

```
1.  //从B树上查找插入位置
2.  int i;
3.  MBNode *xp=MT, *p=NULL;    //xp和p分别指向当前结点和父结点
4.  while(xp != NULL) {
5.      i=1;
6.      while( K > xp->key[i] ) i++;
7.      if(K == xp->key[i])        //关键字已经存在，插入失败
8.          return false;
9.      else {
10.         p=xp;
11.         xp=xp->ptr[i-1];       //下移一层查找
12.         }
13.     }
```

8.5.2 B树-插入 代码

```
1.  //向非空的B树中插入索引项（K, num, ap)
2.  MBNode *ap = NUll;      //ap的初值为空
3.  while(1){
4.      int j, c;  //从最后到插入位置的所有索引项均后移一个位置
5.      for(j=p->keynum; j>=i; j--) {
6.          p->key[j+1] = p->key[j];
7.          p->recptr[j+1] = p->recptr[j];
8.          p->ptr[j+1] = p->ptr[j];
9.          }
10.         //把一个插入索引项（K,num,ap）放入p结点的i下标位置
11.     p->key[i]=k; p->recptr[i]=num; p->ptr[i]=ap;
12.         //使p结点的关键字个数增1
13.     p-> keynum ++;
```

8.5.2 B树-插入 代码

```
1.      //若插入后结点中关键字个数不超过所允许的最大值，则插入完成
2.      if(p->keynum<=m-1) {
3.          p->key[p->keynum+1] = MaxKey;
4.          return true;
5.          }
6.      //计算出m/2的向上取整值
7.      c = (m%2 ? (m+1)/2 : m/2);
8.      //建立新分裂的结点，该结点含有m-c个索引项
9.      ap=new MBNode;
10.     ap->keynum=m-c;
11.     ap->parent=p->parent;
12.     for(j=1; j<=ap->keynum; j++){      //复制关键字和记录位置
13.         ap->key[j] = p->key[j+c];
14.         ap->recptr[j] = p->recptr[j+c];
15.         }
```

8.5.2 B树-插入 代码

```
1.      for(j=0; j<=ap->keynum; j++) {          //复制指针
2.          ap->ptr[j]=p->ptr[j+c];
3.          if(ap->ptr[j] != NULL)     ap->ptr[j]->parent=ap;
4.          }
5.      ap->key[m-c+1]=MaxKey;   //最大值放入所有关键字之后
6.      //修改p结点中的关键字个数
7.      p->keynum=c-1;
8.      //建立新的待向双亲结点插入的索引项（K,num,ap)
9.      K = p->key[c];
10.     num=p->recptr[c];
11.     //在p结点的所有关键字最后放入最大关键字
12.     p->key[c]=MaxKey;
```

8.5.2 B树-插入 代码

```
1.        //建立新的树根结点
2.        if(p->parent==NULL)
3.              {
4.              MT=new MBNode;
5.              MT->keynum = 1;  MT->parent = NULL;
6.              MT->key[1] = K;   MT->key[2] = MaxKey;
7.              MT->recptr[1] = num;
8.              MT->ptr[0] = p;     MT->ptr[1] = ap;
9.              p->parent=ap->parent = MT;
10.             return true;
11.             }
12.             //求出新的索引项 (K, num, ap)在双亲结点的插入位置
13.             p=p->parent;
14.             i=1;
15.       while(K>p->key[i]) i++;
16.       }
17.  }
```

8.5.2 B树-删除分析

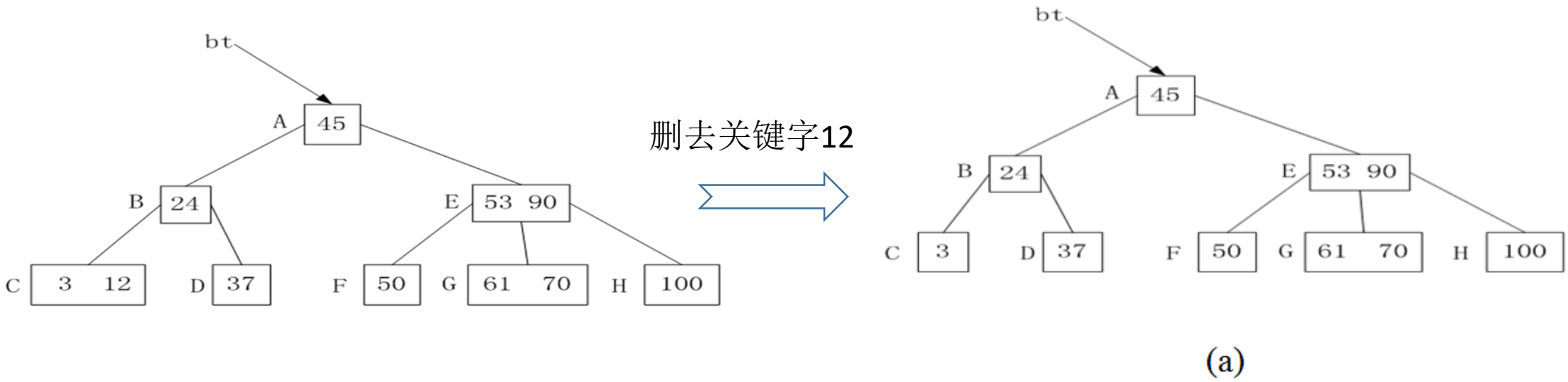
- 在B树上删除一个关键字，则首先应该利用前述的B树的查找算法找出该关键字所在的结点，并从中删除之。
  - 若该结点为最底层非叶子结点，且该结点内的关键字个数不少于⌈m/2⌉，则删除完成，否则需要进行“合并”结点的操作。
  - 假如所删关键字为非底层结点中的K_i，则可以将A_i所指子树中的最小关键字Y替代K_i，然后在相应的结点中删去Y。
    - 先转换成在终端结点上，再按照在终端结点上的情况来分别考虑对应的方法；
    - 找到相邻关键字（左子树中值最大的关键字或右子树中值最小的关键字）

8.5.2 B树-删除分析

- 讨论删除最底层非叶子结点中关键字的情形。
- 有下列三种可能：

（1）被删除关键字所在结点中的关键字数目不小于[m/2]，则只需从该结点中删去该关键字K_i和相应指针A_i，树的其它部分不变。

例如，从左图B树中删去关键字12，删除后的B树如图（a）所示。

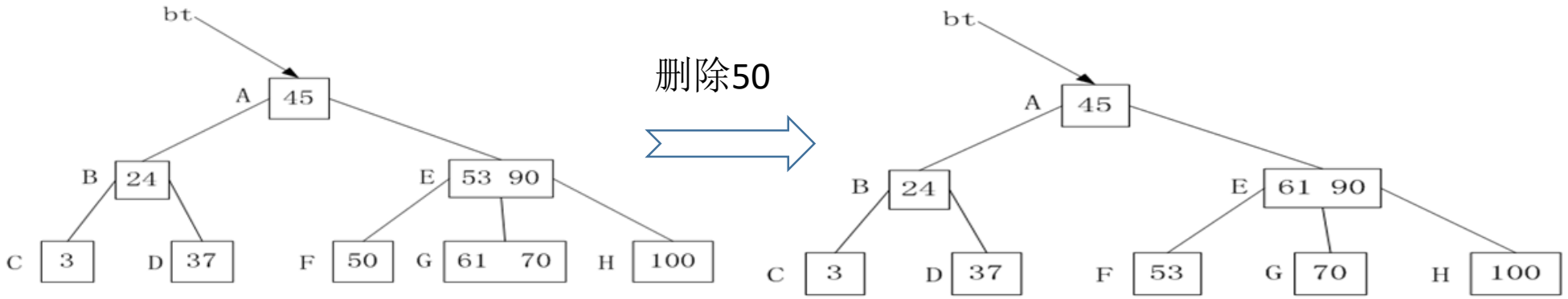


(a)

8.5.2 B树-删除分析

- （2）被删除关键字所在结点中的关键字数目等于[m/2]-1，而与该结点相邻的右兄弟（或左兄弟）结点中的关键字数目大于[m/2]-1，则需将其兄弟结点中的最小（或最大）的关键字上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字的关键字下移至被删关键字所在结点中。

  例如，在图(a)中删除50之后，得到图(b)。

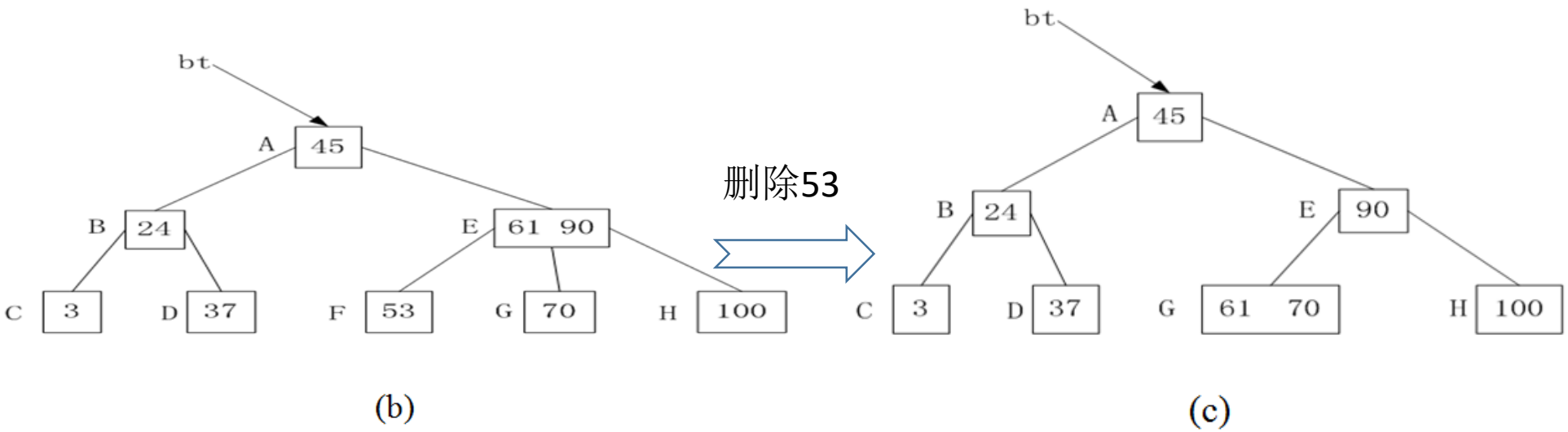


(a)　　(b)

8.5.2 B树-删除分析

- （3）被删除关键字所在结点和其相邻的兄弟结点中的关键字数目均等于[m/2]-1。假设该结点有右兄弟，且其右兄弟结点地址由双亲结点中的关键字A_i所指，则在删去关键字之后，它所在结点中剩余的关键字和指针，加上双亲结点中的关键字K_i一起，合并到A_i所指兄弟结点中（若没有右兄弟，则合并至左兄弟）。

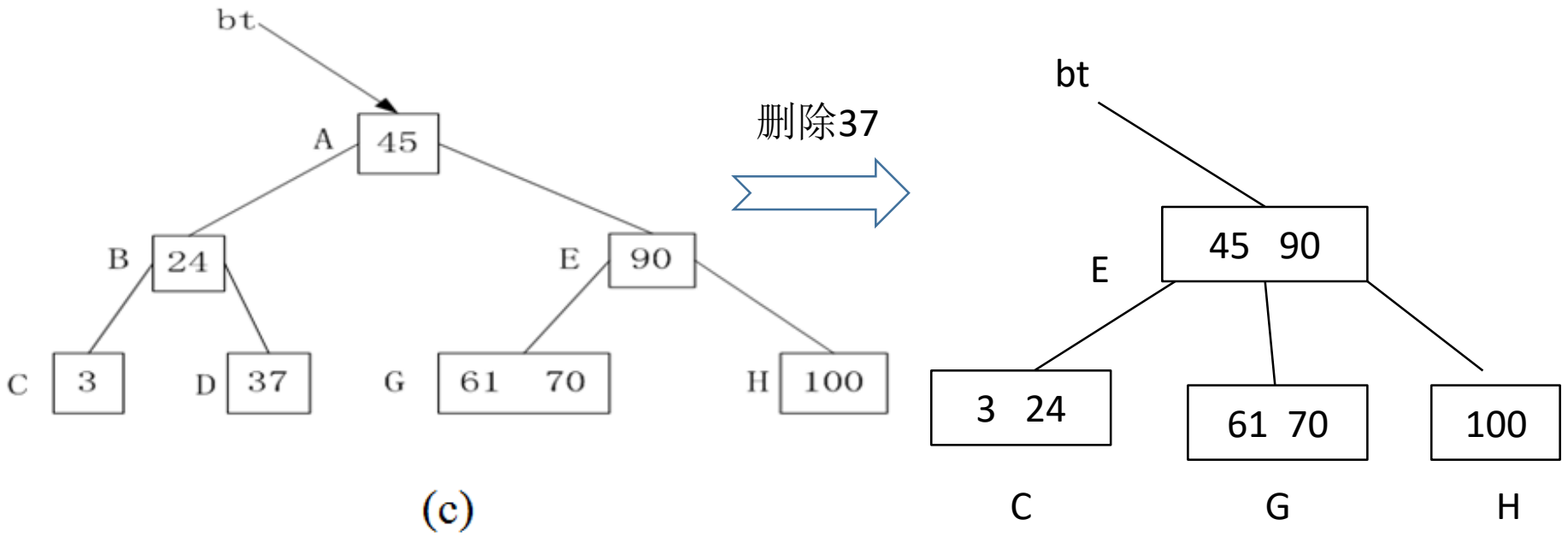
例如，图(b)B树中删去53之后，得到结果图(c)。



(b)　　(c)

8.5.2 B树-删除分析

- 在图(c)B树中删去关键字37之后，双亲B结点中剩余信息（“指针C”）应和其双亲*A结点中关键字45一起合并至右兄弟结点*E中。从而导致根被合并，树的高度降低。

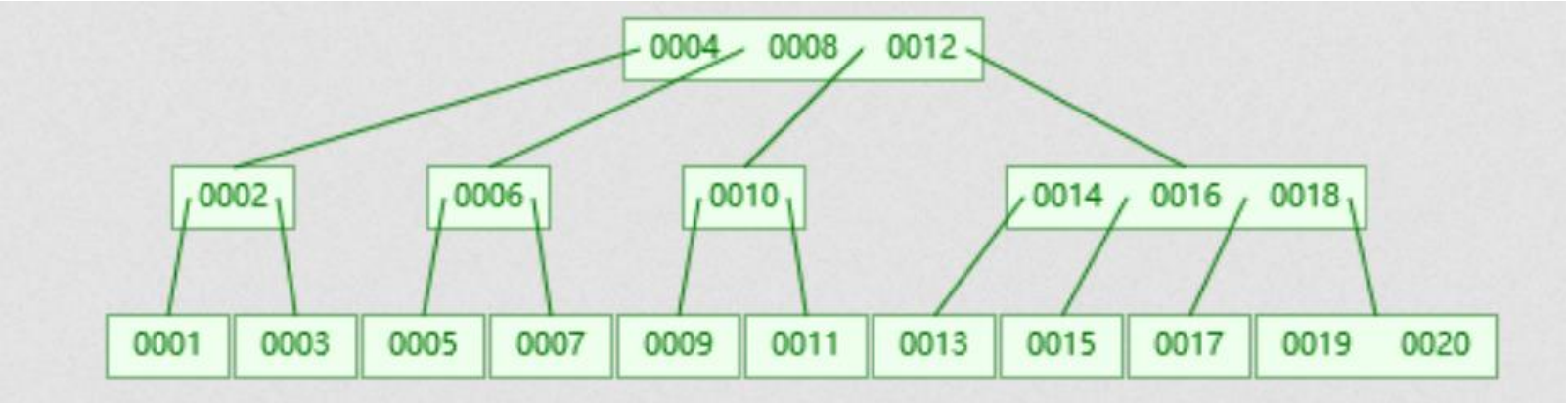


主观题 10分

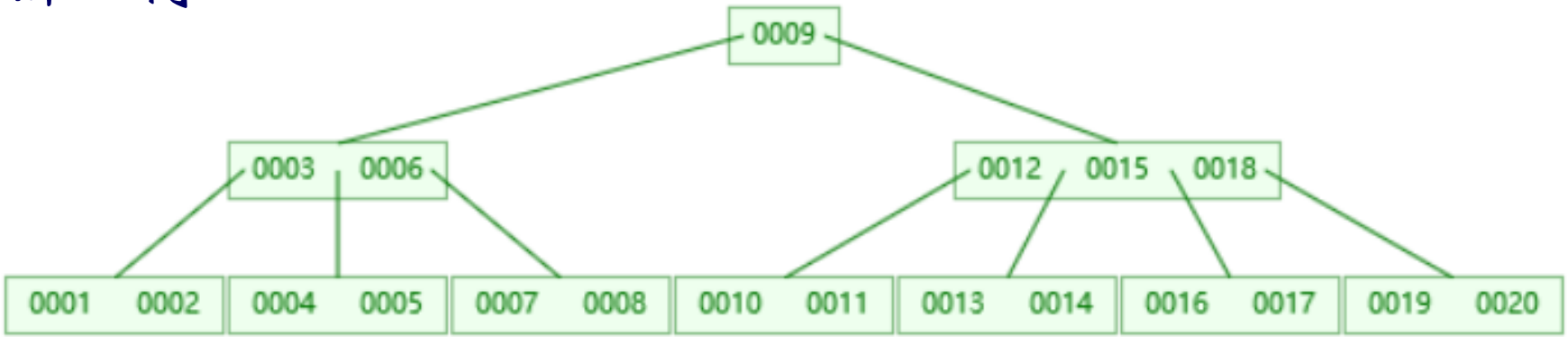
设置

对于4阶或5阶B-树，依次删除关键字1~20，画出其删除过程。

- 4阶B-树



- 5阶B-树



作答

正常使用主观题需2.0以上版本雨课堂

单选题 1分

设置

在 19 个记录中查找其中的某个记录，若要求最多只需要进行 4 次关键字比较，则可采用的查找方法是（　　）。

A 3阶B-树查找

B 折半查找

C 二叉排序树查找

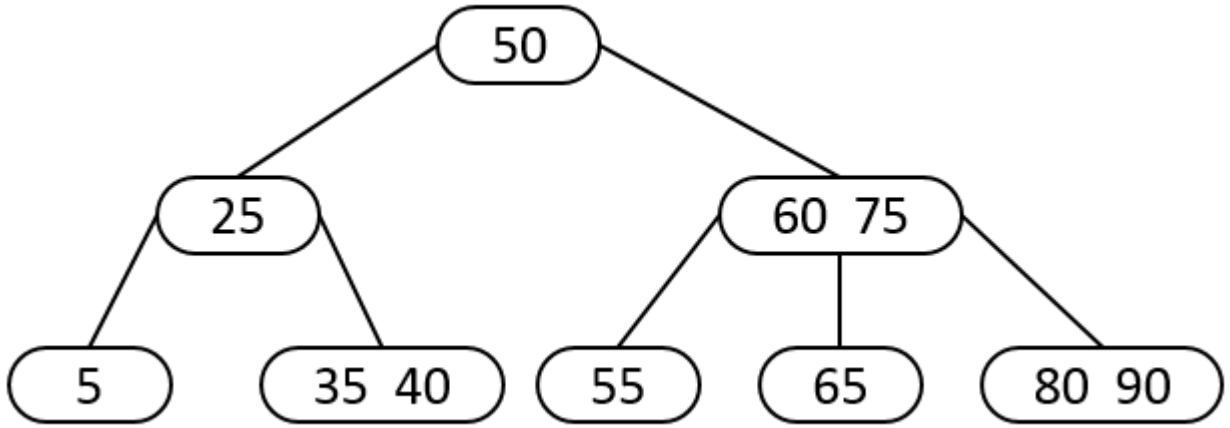
D 顺序查找

提交

单选题 1分

设置

已知一棵 3 阶 B-树如下图所示，下列关于插入关键字 85 后的树形的表述中正确的有（　　）。



A 关键字 60 和 65 都位于最底层非终端结点中。

B 最底层最右边的非终端结点包含的关键字仍为80和90。

C 关键字 80位于根结点中。

D 关键字 85 被插入到第二层最右边的结点中。

提交

8.5.3 B+树

- 一棵m阶B^+树可以定义如下：

  （1）每个结点至多有m个子结点。

  （2）每个结点（除根结点外）至少有[m/2]个子结点。

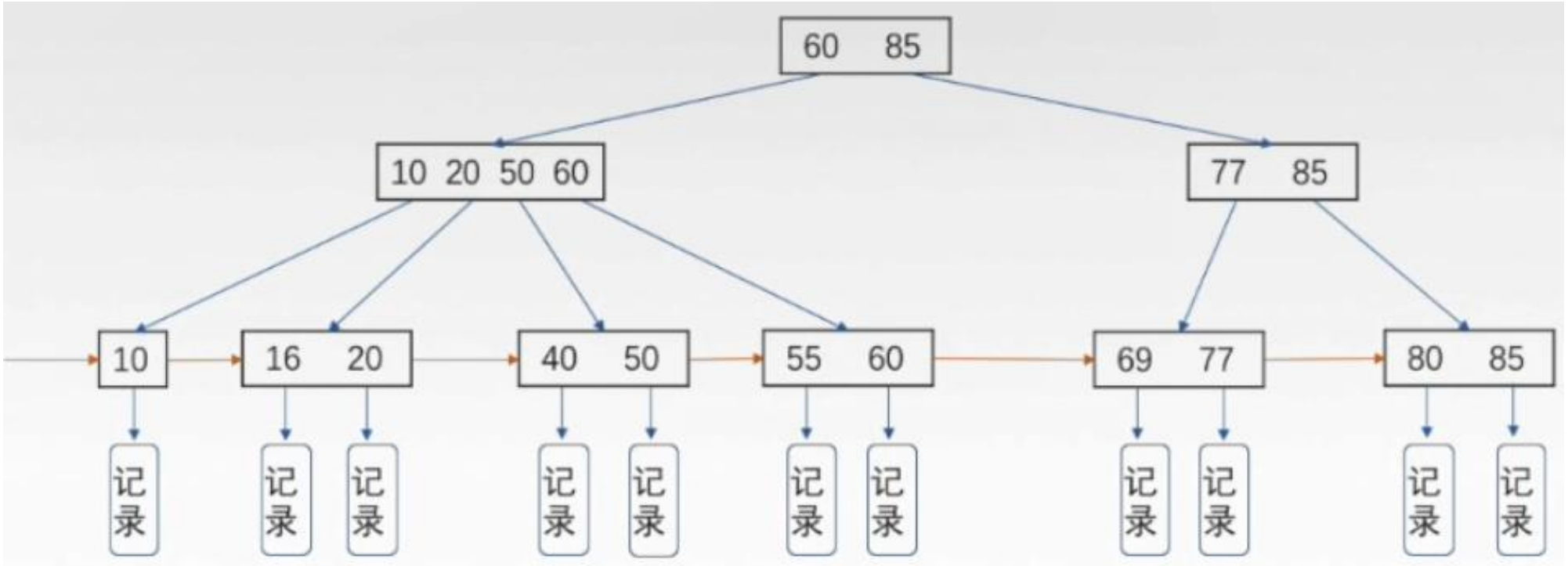
  （3）根结点至少有两个子结点（空树或独根的情况除外）。

  （4）所有的终端结点在同一层，可以有[m/2]~m个关键字，终端结点包含全部关键字以及相应的记录信息，终端结点之间可以用双链表顺序链接。

  （5）有k个子结点的结点（分支结点）必有k个关键字。

8.5.3 B+树

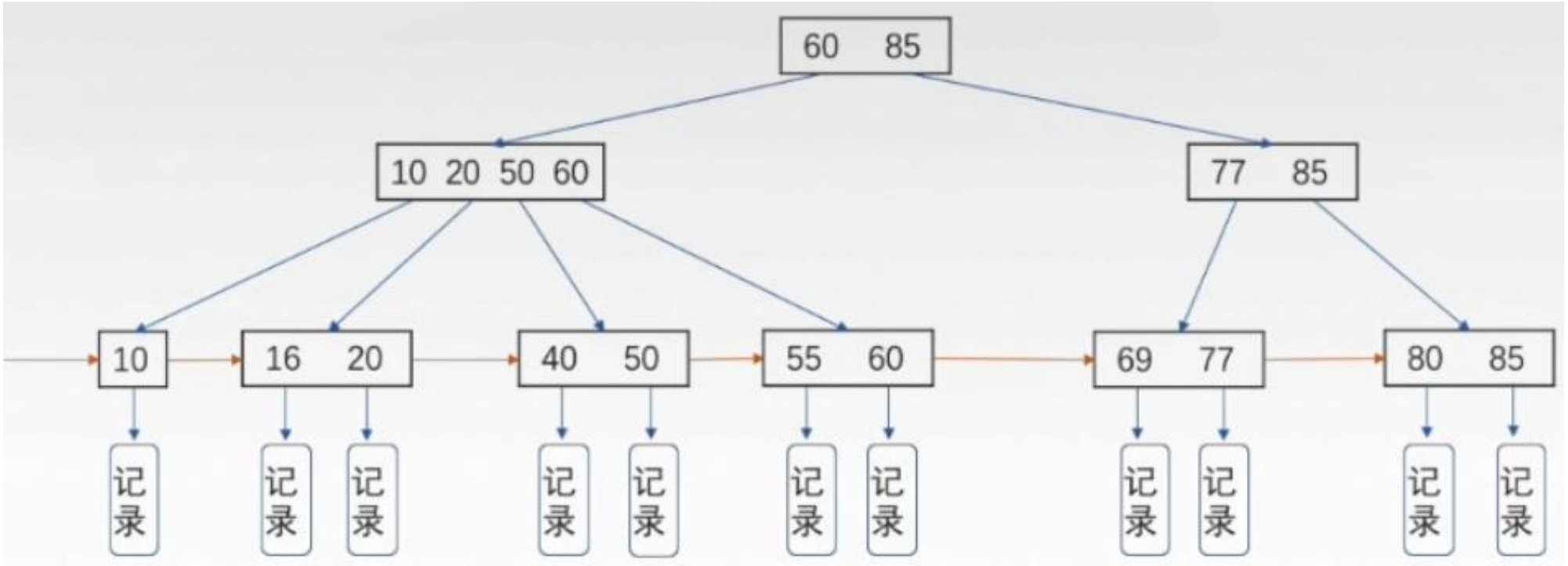
- 一棵m阶的B^+树和m阶的B树的差异在于：



（1）在B+树中，有n棵子树的结点中含有n个关键字，即每个关键字对应一颗子树；而在B树中具有n个关键字的结点含有n+1颗子树；

8.5.3 B+树

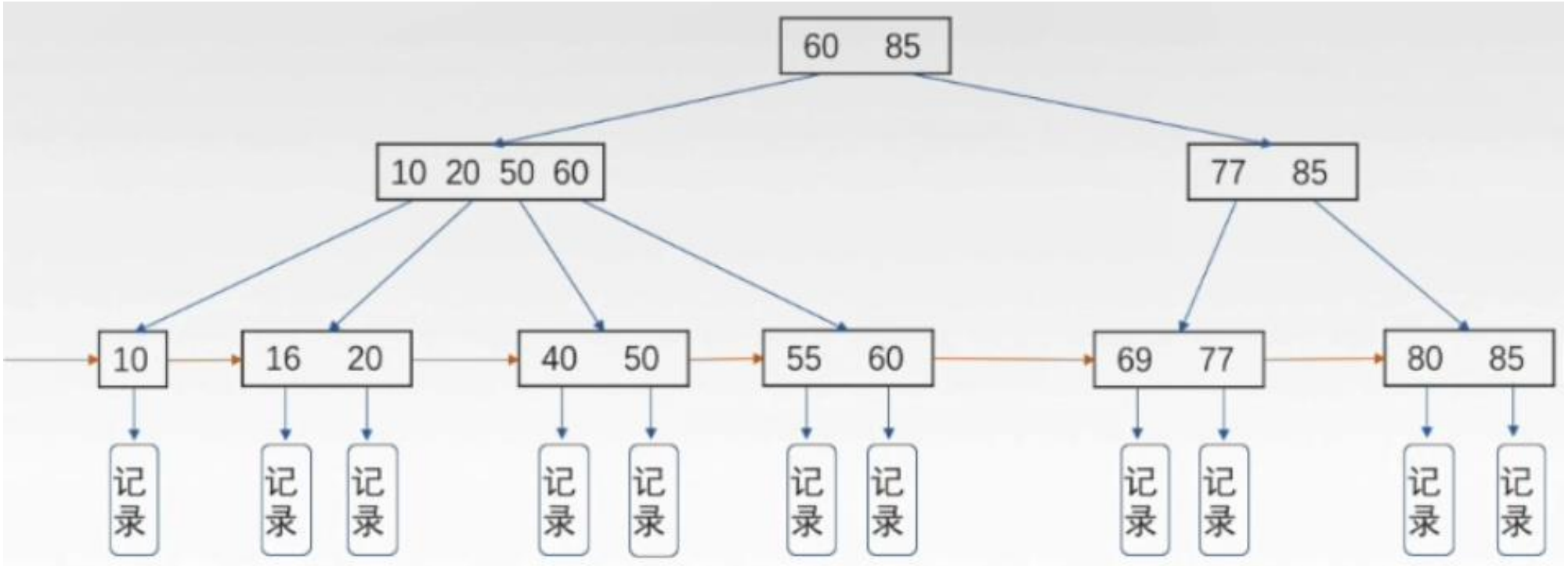
- 一棵m阶的B^+树和m阶的B树的差异在于：



（2）在B+树中，每个结点（非根内部节点）关键字个数n的范围是[m/2]~m（根节点是1~m），在B树中，每个结点（非根内部节点）关键字个数n的范围是[m/2]-1~m-1（根节点是1~m-1）。

8.5.3 B+树

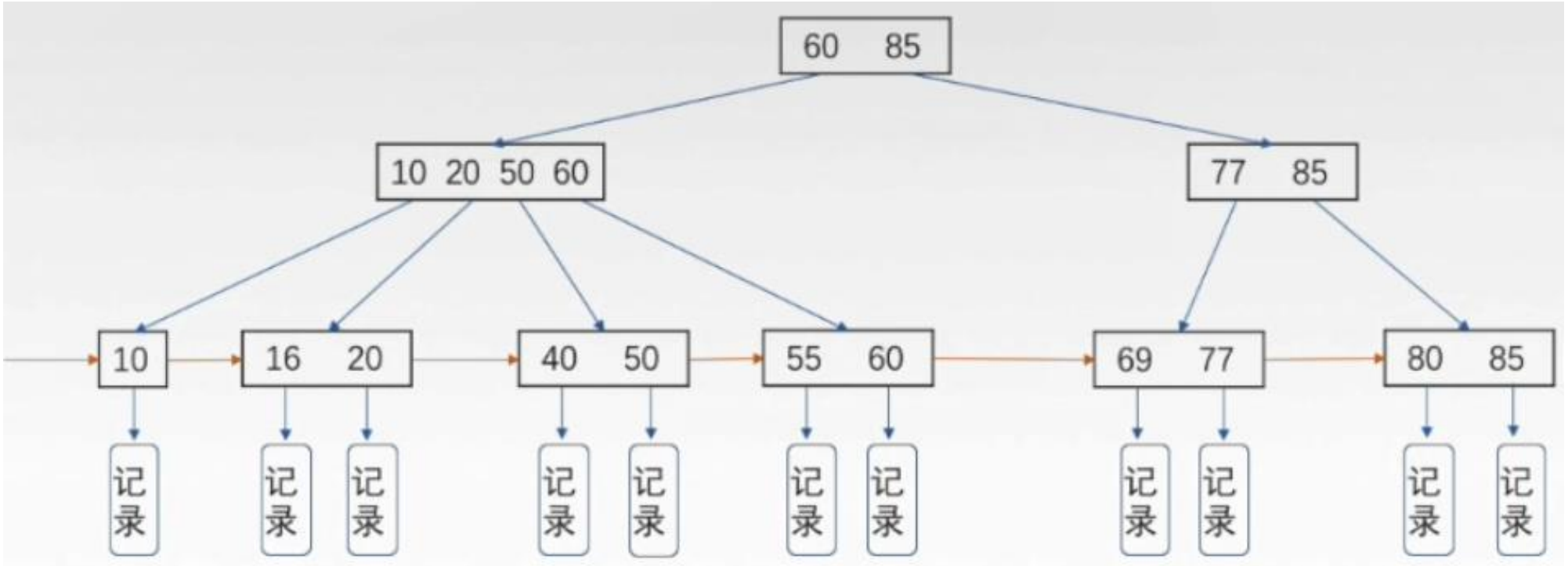
- 一棵m阶的B^+树和m阶的B树的差异在于：



（3）在B+树中，叶子结点包含信息，所有非叶结点仅起到索引作用，非叶结点中每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。而在B树中每个关键字对应一个记录的存储地址。

8.5.3 B+树

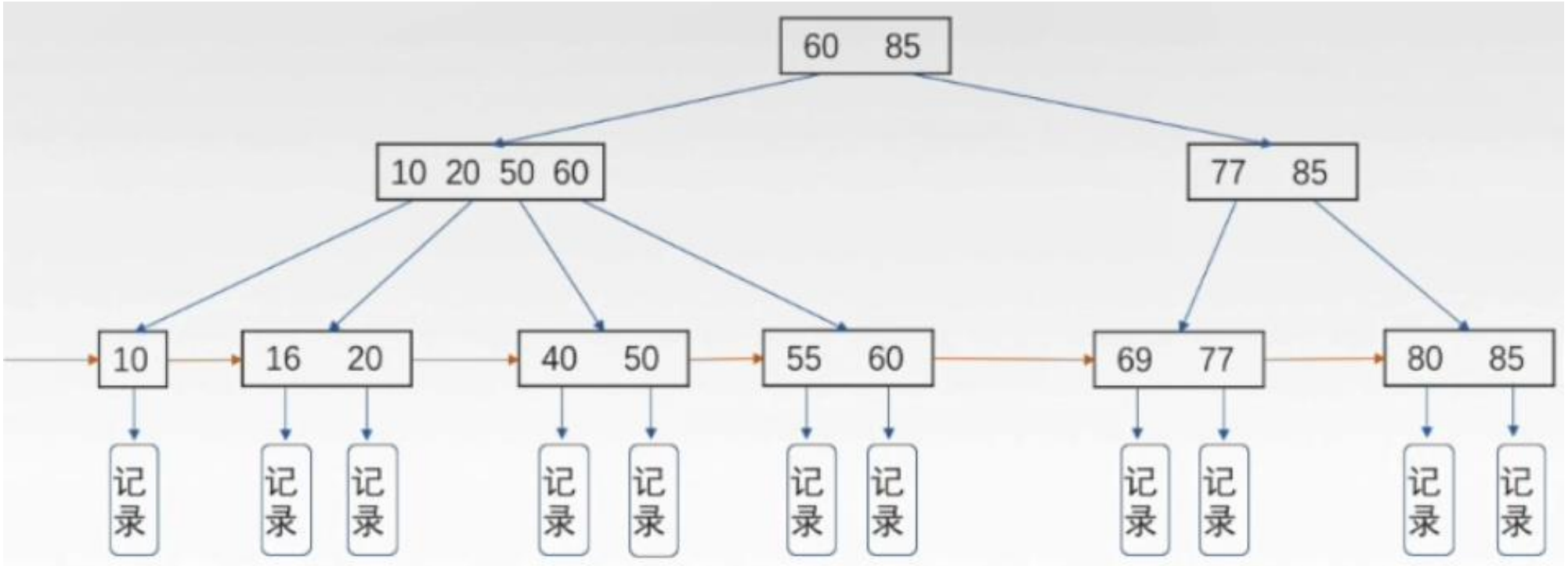
- 一棵m阶的B^+树和m阶的B树的差异在于：



（4）在B+树中，叶结点包含了全部关键字，即在非叶结点中出现的关键字也会出现在叶结点中，而且叶子结点的指针指向记录；而在B树中，叶结点包含的关键字和其他结点包含的关键字是不重复的。

8.5.3 B+树

- 一棵m阶的B^+树和m阶的B树的差异在于：



（5）在B+树中，有一个指针指向关键字最小的叶子结点，所有叶子结点链接成一个单链表。

8.5.3 B+树

- 在B^+树上进行随机查找时，不管查找成功与否，每次查找都是走了一条从根到终端结点的路径。
- B^+树的插入仅在终端结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为⌈(m+1)/2⌉和⌊(m+1)/2⌋。并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。
- B^+树的删除也仅在终端结点上进行，当终端结点中的最大关键字被删除时，其在非终结点中的值可以作为一个“分界关键字”存在。如果因删除而使得结点中的关键字个数少于⌈m/2⌉时，其和兄弟结点的合并过程也与B树类似。

本课程授课到此结束！

谢谢大家！