数据结构与算法

第十章 查找

裴文杰

计算机科学与技术学院 教授

本章重点与难点

重点：顺序查找、二分查找、二叉排序树查找以及散列表构造及散列方法实现。

难点：二叉排序树的删除算法和平衡二叉树的构造算法。

第十章 查找

10.1 静态查找表

10.2 动态查找表

10.3 哈希表

预备知识

查找表

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：

- 查询某个“特定的”数据元素是否在查找表中；
- 检索（知道存在）某个“特定的”数据元素的各种属性；
- 在查找表中插入一个数据元素；
- 从查找表中删去某个数据元素。

预备知识

查找的分类

✓根据查找方法取决于记录的键值还是记录的存储位置？

- 基于关键字比较的查找
  - ◆ 顺序查找、折半查找、分块查找、BST&AVL、B-树 和B+树
- 基于关键字存储位置的查找
  - ◆ 散列法

✓根据被查找的数据集合存储位置

- 内查找：整个查找过程都在内存进行；
- 外查找：若查找过程中需要访问外存，如B树和B+树

预备知识

查找的分类

✓根据查找方法是否改变数据集合？

- 静态查找：
  - ◆ 查找+提取数据元素属性信息
  - ◆ 被查找的数据集合经查找之后并不改变，就是说，既不插入新的记录，也不删除原有记录。
- 动态查找：
  - ◆ 查找+插入或删除元素
  - ◆ 被查找的数据集合经查找之后可能改变，就是说，可 以插入新的记录，也可以删除原有记录。

预备知识

有关概念

(1) 主关键字

可以识别唯一的一个记录的数据项（字段）。
例如学号

(2) 次关键字

可以识别若干的一个记录的数据项（字段）。
例如姓名

(3) 查找

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或（记录）。

预备知识

有关概念

(4) 查找成功

查找表中存在满足条件的记录。

(5) 查找不成功

查找表中不存在满足条件的记录。

本章主要讨论主关键字

预备知识

- 如何进行查找

查找的方法取决于查找表的结构。

由于查找表中的数据元素之间不存在明显的组织规律，因此不便于查找。

为了提高查找的效率， 需要在查找表中的元素之间人为地附加某种确定的关系，换句话说，用另外一种结构来表示查找表。

第十章 查找

10.1 静态查找表

10.2 动态查找表

10.3 哈希表

10.1 静态查找表

10.1.1 顺序表的查找

10.1.2 有序表的查找

*10.1.3 静态树表的查找 //看书自学

10.1.4 索引顺序表的查找

10.1 静态查找表

□ 静态查找表的抽象数据类型

ADT StaticSearchTable {

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：见下页

} ADT StaticSearchTable

10.1 静态查找表

基本操作

Create(&ST, n); //建立静态查找表

Destroy(&ST); //销毁静态查找表

Search(ST, key); //按关键字key查找

Traverse(ST, Visit()); //遍历查找表

10.1 静态查找表

10.1.1 顺序表的查找

10.1.2 有序表的查找

*10.1.3 静态树表的查找　//看书自学

10.1.4 索引顺序表的查找

10.1 静态查找表

◘ 顺序查找表存储结构

假设静态查找表的顺序存储结构为

```
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;      // 表的长度
} SSTable;
```

10.1 静态查找表

- 顺序查找表
- 有序查找表
- 静态查找树表
- 索引顺序表

10.1.1 顺序查找表

以顺序表或线性链表表示静态查找表

10.1.1 顺序查找表

◘ 顺序查找表的思想

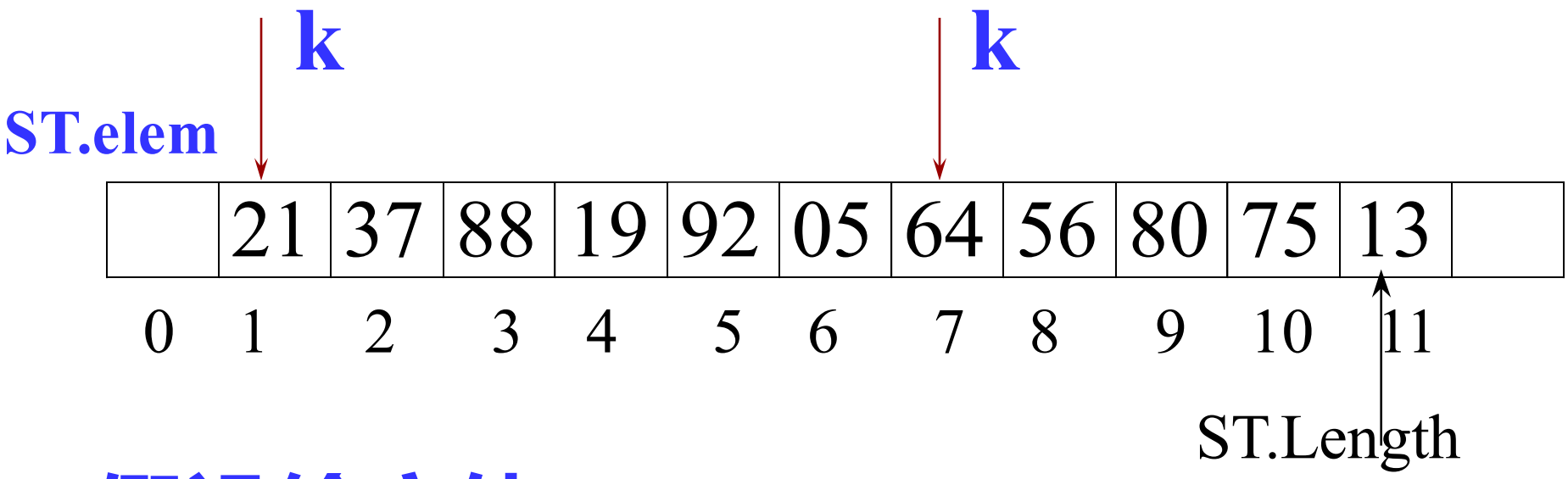
(1)从查找表的第一个元素向后（或从最后一个元素向前），比较当前位置数据元素的关键字与查找关键字；

(2)若相等，输出当前位置，查找成功，若不相等，走向下一个位置；

(3)循环执行(1)、(2)步，直到查找成功或超出范围，表示查找失败。

10.1.1 顺序查找表

回顾顺序表的查找过程---从前往后找



假设给定值 e=64,

要求 ST.elem[k] = e, 问: k = ?

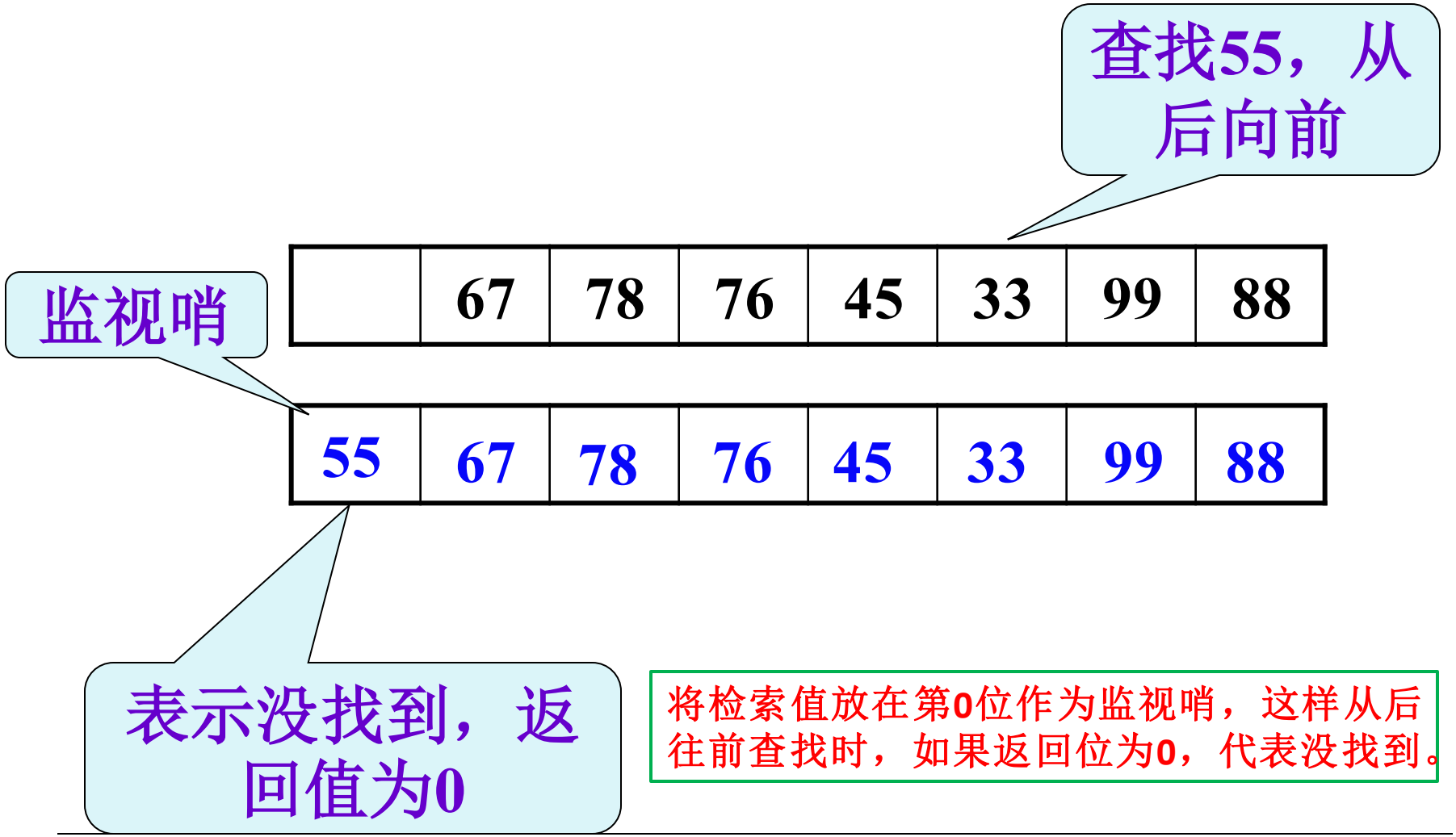
10.1.1 顺序查找表

```
int location( SqList L, ElemType& e,
      Status (*compare)(ElemType, ElemType)) {
  k = 1;  p = L.elem;
  while ( k<=L.length &&
            !(*compare)(*p++,e))) k++;
  if ( k<= L.length)  return k;
  else  return 0;
} //location
```

- 当查找成功时，条件k<=L.length比较是多余的。
- 对于静态查找一般都是查找成功的；
- 当表长>1000时，经验表明条件k<=L.length几乎
- 消耗查找时间的一半

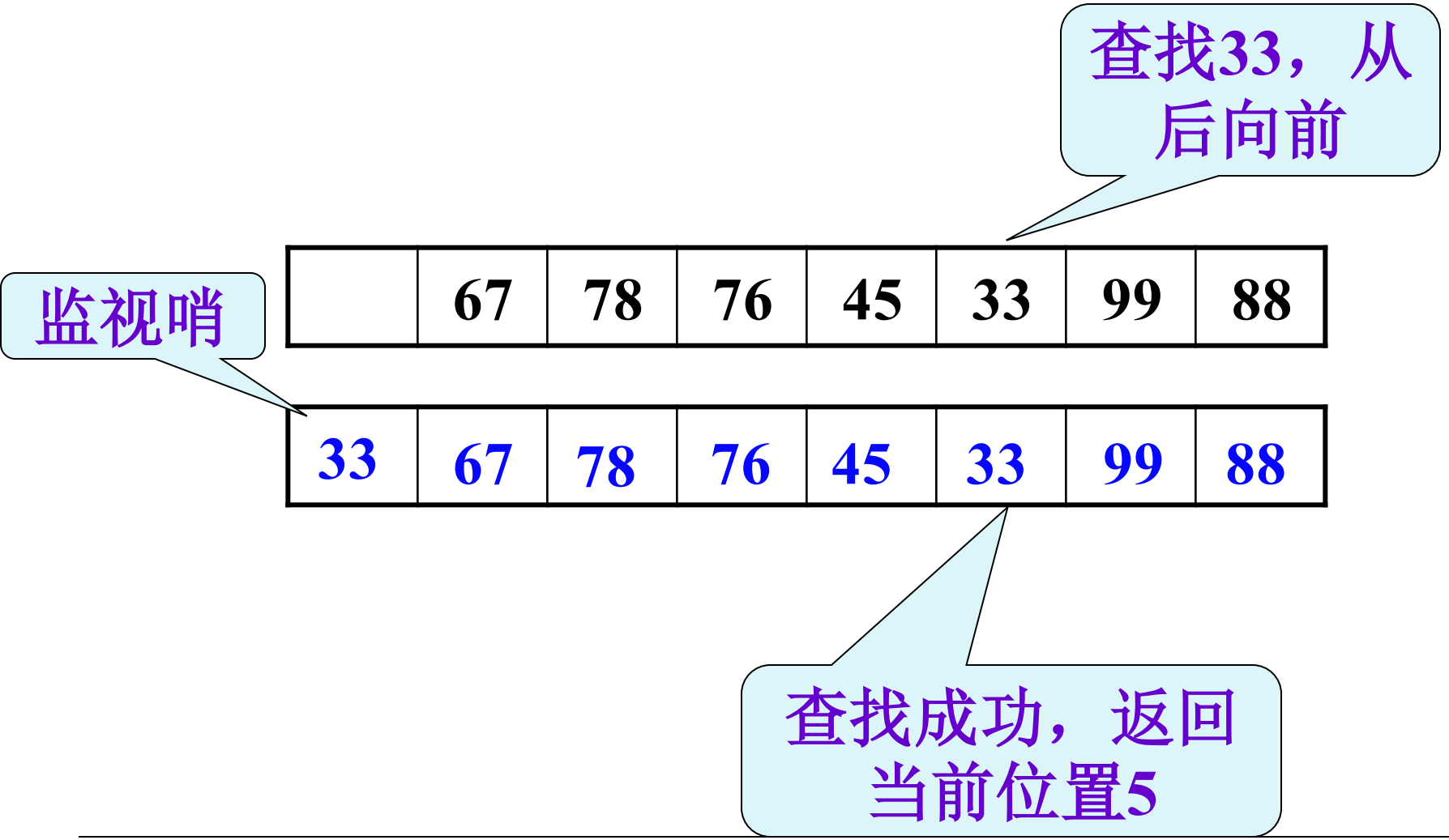
10.1.1 顺序查找表

顺序查找—从后向前



10.1.1 顺序查找表

顺序查找—从后向前



10.1.1 顺序查找表

顺序查找—从后向前

```
int Search_Seq(SSTable ST, KeyType key) {
        // 在顺序表ST中从后面开始顺序查找其关
        //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;                    // “哨兵”
  for (i=ST.length; ST.elem[i].key!=key;  --i);
                                           // 从后往前找
  return i;                                // 找不到时，i为0
} // Search_Seq
```

对于查找不用时间复杂度分析，
而是比较记录个数的平均值成为衡量算法的主要指标

10.1.1 顺序查找表

口分析顺序查找的时间性能

定义：查找算法的平均查找长度（Average Search Length）：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值

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

n 为表长

P_i 为查找表中第i个记录的概率

C_i为找到记录时，关键字的比较次数

10.1.1 顺序查找表

口分析顺序查找的时间性能

对顺序表而言，如果从后往前查：$C_i = n-i+1$

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

在等概率查找的情况下，

$$P_i = \frac{1}{n}$$

顺序表查找的平均查找长度为：

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

10.1.1 顺序查找表

顺序查找表算法分析

在不等概率查找的情况下，*ASL*取最小值，当

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

根据此性质构造表：

对于出现概率高的排在最后，便于先查找。例如最常用的书放在上面

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

例如：将全校学生的病历档案建立一个表存放在计算机中，则体弱多病同学的病历记录的查找概率必定高于健康同学的病历记录。

10.1.1 顺序查找表

- 顺序查找表算法分析

上述顺序查找表的查找算法简单，但平均查找长度较大，特别不适用于表长较大的查找表。

10.1.2 有序表的查找

□ 折半查找的前提要求

想一想：英文字典的特点及英语单词的查找过程。

英文字典是有序的，英语单词的查找用的是折半查找。

折半查找的前提要求是顺序存储并且有序。

若以有序表表示静态查找表，则查找过程可以基于“折半”进行。

10.1.2 有序表的查找

□ 折半查找表的思想

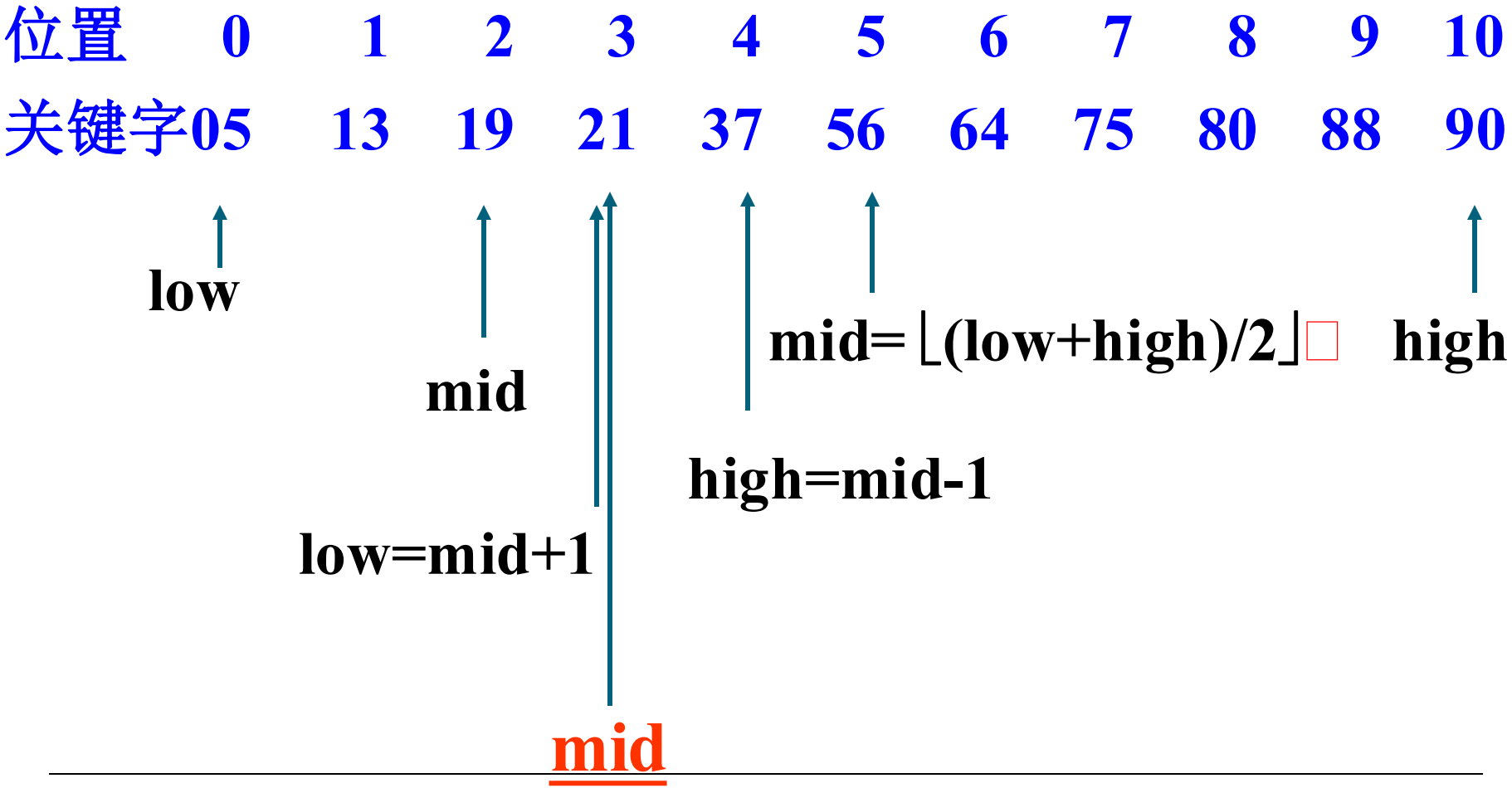
(1)、将要查找关键字与查找表的中间的元素进行比较，若相等，返回当前位值；

(2)、若查找关键字比当前位置关键字大，向前递归，否则向后递归。

10.1.2 有序表的查找

- 折半查找表示例演示

查找21



10.1.2 有序表的查找

□ 折半查找表算法与实现

```
int Search_Bin ( SSTable ST, KeyType key ) {
   low = 1;  high = ST.length;          // 置区间初值
   while (low <= high)
   {
        ……
   }
   return 0;                    // 顺序表中不存在待查元素
} // Search_Bin
```

10.1.2 有序表的查找

□ 折半查找表算法与实现

```
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;        // 继续在前半区间进行查找
    else  low = mid + 1; // 继续在后半区间进行查找
  }
```

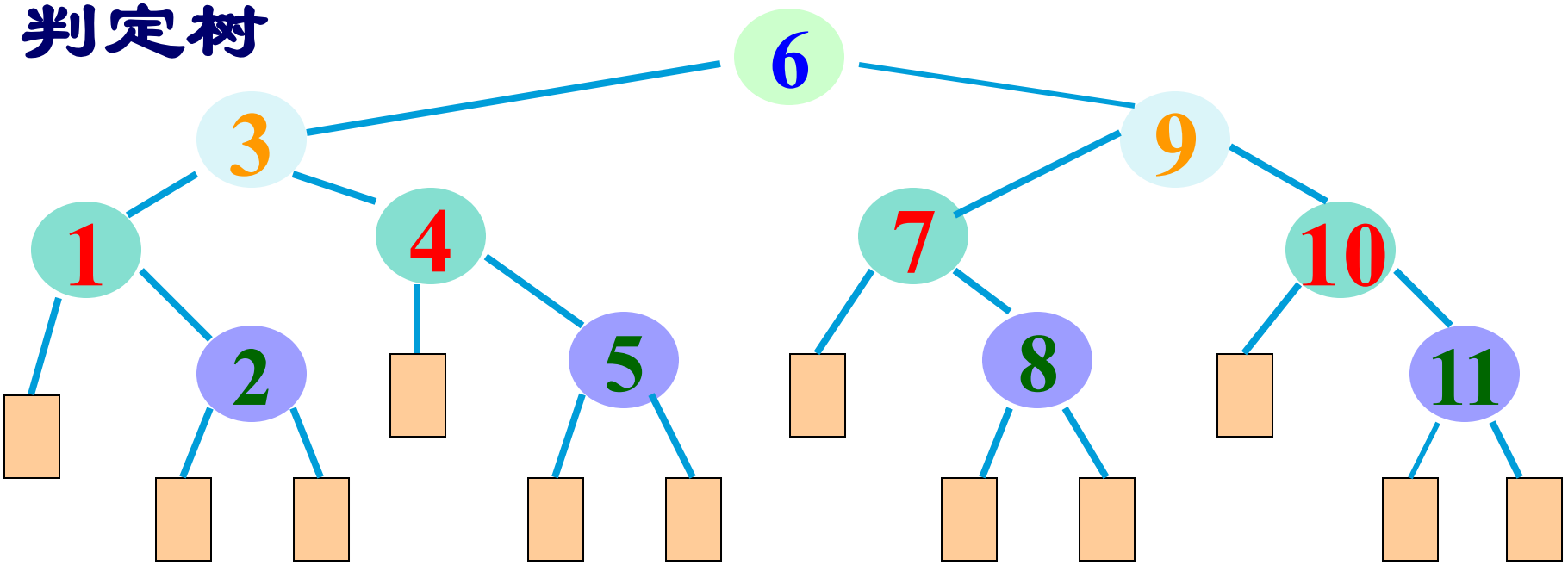
10.1.2 有序

■分析折半查找的平均查找长

折半查找判定树：
二叉搜索（排序）树（即每个结点的值均大于其左子树上所有结点的值，小于其右子树上所有结点的值），树中每个结点表示表中一个记录，结点值为该记录在表中的位置，因此：
查找一个记录的过程就是从根节点到该记录结点的路径，比较次数为该结点在树中的层次数(层高)。

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

判定树



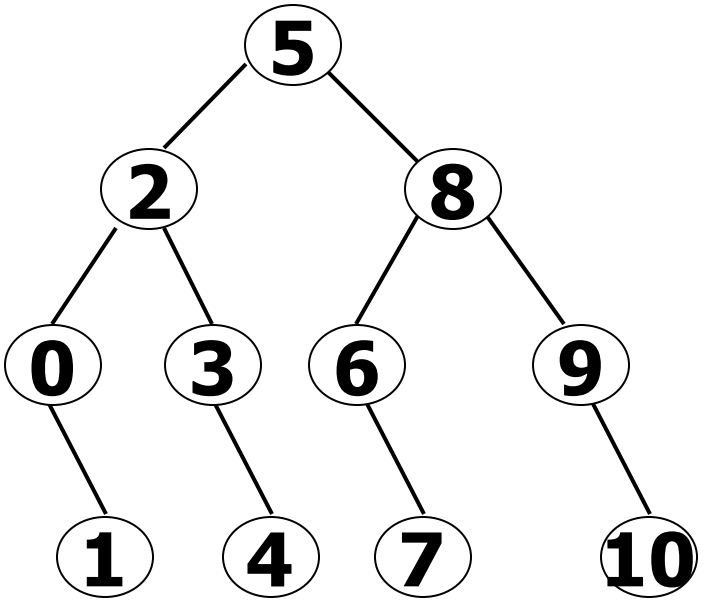
比较次数=结点所在的层次

10.1.2 有序表的查找

□ 折半查找表算法分析

0	1	2	3	4	5	6	7	8	9	10
05	13	19	21	37	56	64	75	80	88	90

用判定树描述折半查找的过程。



设$n=2^h-1$（$h=\log_2(n+1)$），则判定树为深度为h的满二叉树，那么平均查找长度为：

$$\frac{1}{n}\sum_{i=1}^{h}2^{i-1}\times i$$

2^{i-1}为第i层的结点数

$$O(\log_2 n)$$

推导过程参考数据结构书P221

10.1.2 有序表的查找

■分析折半查找的平均查找长度

一般情况下，表长为 n 的折半查找的判定树的深度和含有 n 个结点的完全二叉树的深度相同。

假设 $n=2^h-1$ 并且查找概率相等

推导过程参考数据结构书P221

则

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

j表示结点的层数

在n>50时，可得近似结果

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

10.1.4 索引顺序表的查找

顺序表和有序表的比较

	顺序表	有序表
表的特征	表中元素无序	表中元素有序
存储结构	顺序和链表结构	顺序结构
查找方法	顺序查找	折半查找
插删操作	方便	移动元素
ASL	大	小
适用范围	表长较短	表长较长 有序

分析：如何结合顺序表和有序表的特点？

10.1.4 索引顺序表的查找

索引顺序查找又称分块查找

□ 索引顺序表的前提要求

(1) 整个查找表要求顺序存储

(2) 查找表分成n块，当i>j时，第i块中的最小元素大于第j块中的最大元素；

针对每个块，建立一个索引项：包括关键字项（该块中元素的最大关键字）和指针项（该块中第一个元素的地址）

(3) 每个块的索引项构成了索引，索引按照关键字是有序的

□ 索引表的查找思想

(1) 首先确定所要查找关键字在哪一块中。

(2) 在所确定的块中用顺序查找查找关键字。

10.1.4 索引顺序表的查找

□索引顺序表的平均查找长度

索引表=索引+ 顺序表

索引顺序查找的平均查找长度

= 查找“索引”的平均查找长度

+ 查找“顺序表”的平均查找长度

一般情况下，索引是个有序表

10.1.4 索引顺序表的查找

□索引顺序表的查找过程

1）由索引确定记录所在区间；

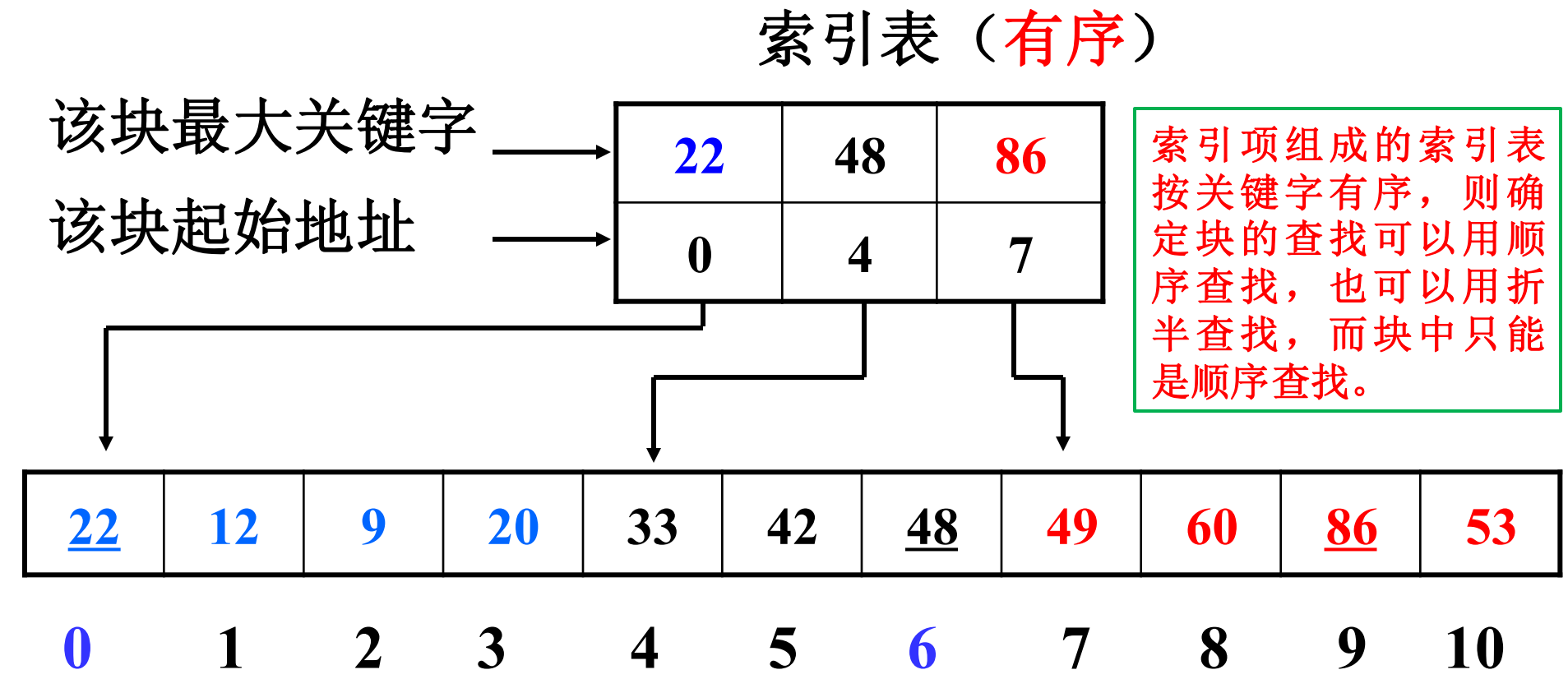
2）在顺序表的某个区间内进行查找。

索引顺序查找的过程是一个“缩小区间”的查找过程。

注意：索引可以根据查找表的特点来构造。

10.1.4 索引顺序表的查找

□ 索引顺序表的示例演示



线性表-**块内可以无序**

（可以顺序存储和链式存储）

10.1.4 索引顺序表的查找

索引表的存储结构

```
typedef struct
{keytype key;
  int addr;
}indextype;//索引表的元素类型

typedef struct
{indextype index[maxblock];
  int block; //总的块数
}INtable;
INtable IX;//索引表的类型
```

每个分块的索引项包括关键字项（该块中元素的最大关键字）和指针项（该块中第一个元素的地址）

10.1.4 索引顺序表的查找

□ 索引顺序表的算法与实现

```
int SEARCH(SSTable ST,INtable IX,
                              KeyType key )
{
  int i=0,s,e;
//s记录在查找表中的开始位置，e记录在查找表中的结束位置，即[s, e]为记录所在块的起始和终止地址
 {在索引表中查找，确定s和e的值}
 {根据s和e的值在线性表中查找}
   return -1
}
```

10.1.4 索引顺序表的查找

□ 索引顺序表的算法与实现

```
while ((key>IX.index[i].key)&&(i<=IX.block))
    i++;
if (i<=IX.block) {
    s=IX.index[i].addr;
    if (i==IX.block) e=ST.length;
    else e=IX.index[i+1].addr-1;
    while (key!=ST.elem[s].key&&(s<=e)
        s=s+1;
    if (s<=e)   return s;
}
return -1;
}
```

10.1.4 索引顺序表的查找

□ 索引顺序表的算法分析

思考题：如果索引表长度为b，每块平均长度为L，平均查找长度是多少？

答案：如果用顺序查找索引表：ASL ≈ (b+1)/2+(L+1)/2；
如果用折半查找索引表：ASL ≈ $\log_2(b+1)+(L/2)$

顺序表查找的平均查找长度为：

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

思考题：长度为n的线性表,分成多少块平均查找次数最少？（采用顺序查找）

答案：$\sqrt{n}$

$$\frac{b+1}{2}+\frac{L+1}{2}=\frac{1}{2}\left(b+\frac{n}{b}\right)+1$$

当b=$\sqrt{n}$时，取最小值$\sqrt{n}$+1

第十章 查找

10.1 静态查找表

10.2 动态查找表

10.3 哈希表

10.2 动态查找表

10.2.1 二叉排序树和平衡二叉树

10.2.2 B-树和B+树

- 动态查找：
  - ◆ 查找+插入或删除元素
  - ◆ 被查找的数据集合经查找之后可能改变，就是说，可 以插入新的记录，也可以删除原有记录。

10.2 动态查找表

□ 动态查找表的抽象数据类型

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： 见下页

}ADT DynamicSearchTable

10.2 动态查找表

基本操作

```
InitDSTable(&DT)                //初始化表
DestroyDSTable(&DT)             //销毁表
SearchDSTable(DT, key);         //按关键字查找
InsertDSTable(&DT, e);          //插入数据元素
DeleteDSTable(&T, key);         //删除数据元素
TraverseDSTable(DT, Visit()); //遍历表
```

10.2 动态查找表

10.2.1 二叉排序树和平衡二叉树

10.2.2 B-树和B+树

10.2.1 二叉排序树和平衡二叉树

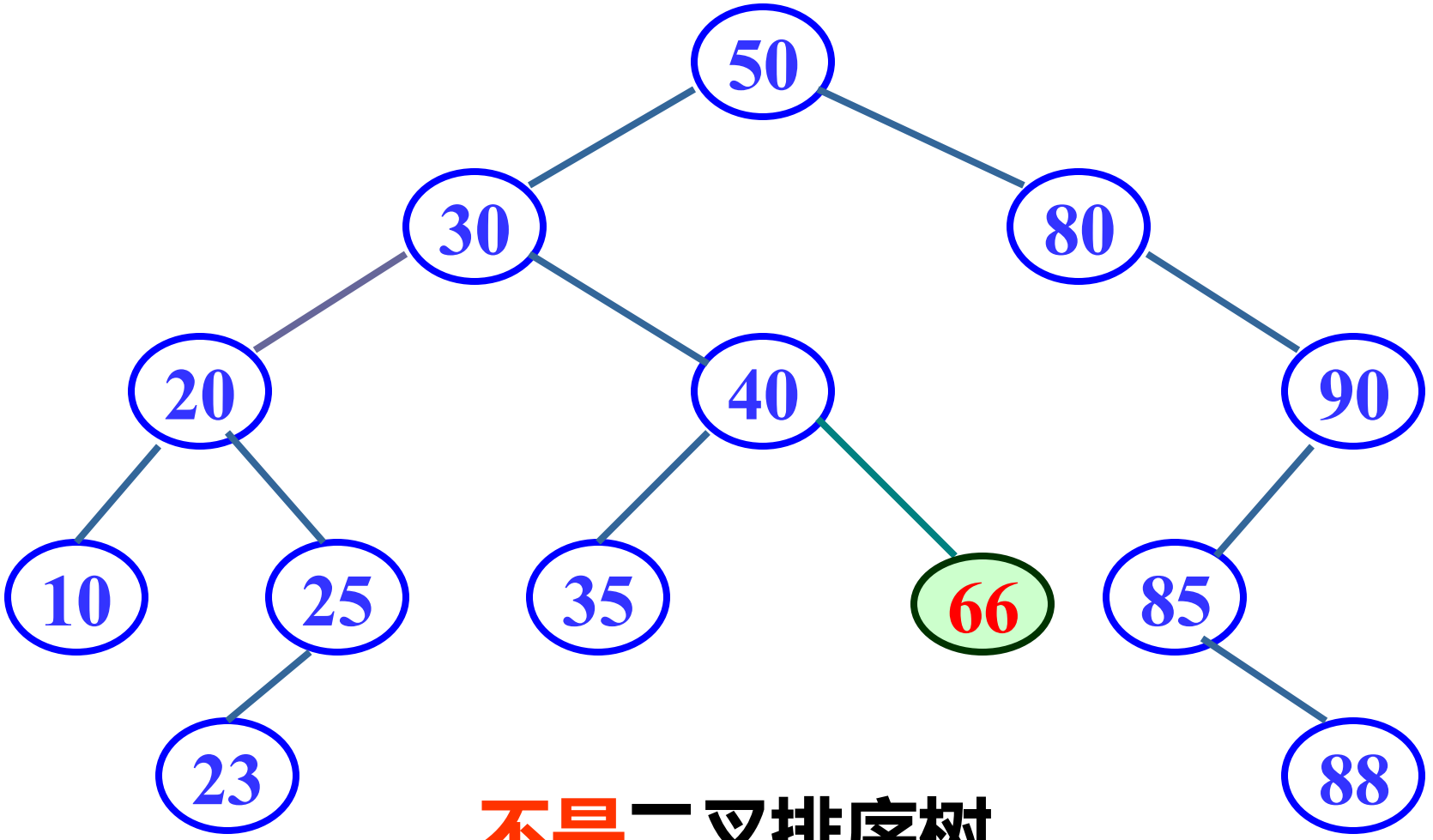
◘ 二叉排序树((Binary Sort Tree))的定义

一种特殊的二叉树，又称二元查找树。它的每个结点数据中都有一个关键值，并有如下性质：

- 对于每个结点，如果其左子树非空，则左子树的所有结点的关键值都小于该结点的关键值；
- 如果其右子树非空，则右子树的所有结点的关键值都大于该结点的关键值。
- 左右子树本身又是一棵二叉排序树

10.2.1 二叉排序树和平衡二叉树

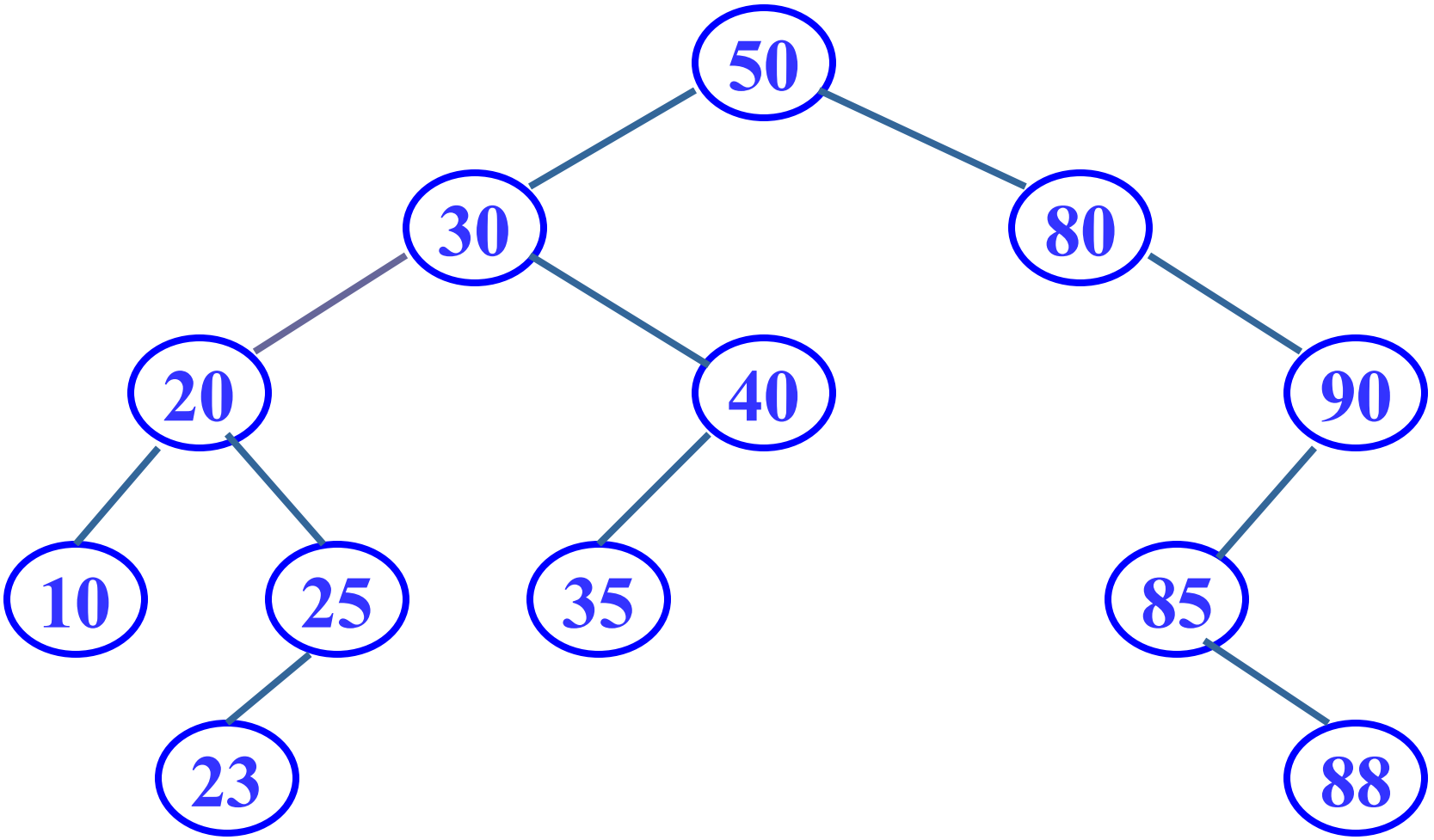
二叉排序树((Binary Sort Tree))的定义



不是二叉排序树

10.2.1 二叉排序树和平衡二叉树

- 二叉排序树((Binary Sort Tree))的定义



中序遍历二叉排序树 得到有序序列

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的存储结构

通常，取二叉链表作为二叉排序树的存储结构

```
typedef struct BiTNode { // 结点结构
    TElemType      data;
    struct BiTNode  *lchild, *rchild;
                              // 左右孩子指针
} BiTNode, *BiTree;
```

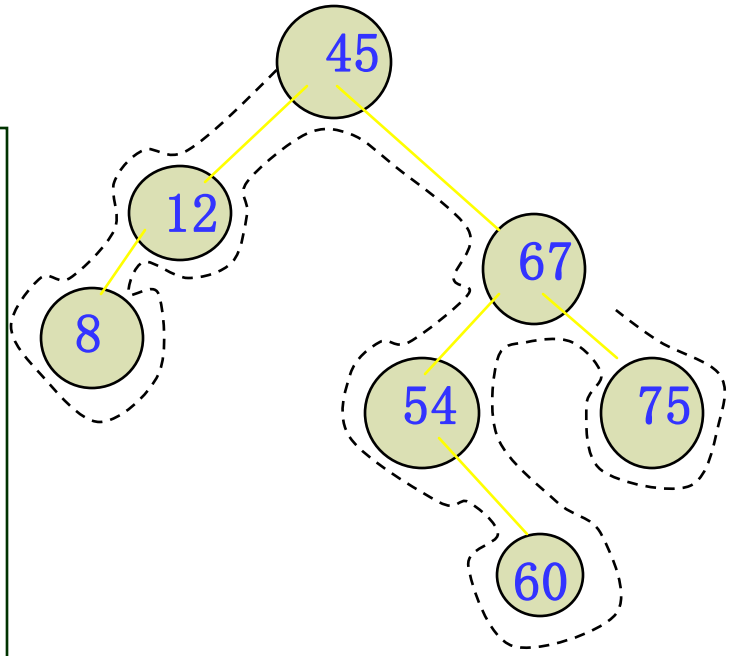
10.2.1 二叉排序树和平衡二叉树

二叉排序树的优点

◆用二叉排序树作为目录树，把一个记录的关键码和记录的地址作为二叉排序树的结点，按关键码值建成二叉排序树。

◆能像有序表那样进行高效查找；

◆能像链表那样灵活删除。



二叉排序树

10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的查找思想

(1) 当二叉排序树不空时，先将给定值和根结点的关键字比较，若相等，则查找成功；否则：

(2) 若给定值小于根结点的关键字，则在左子树上继续进行查找；

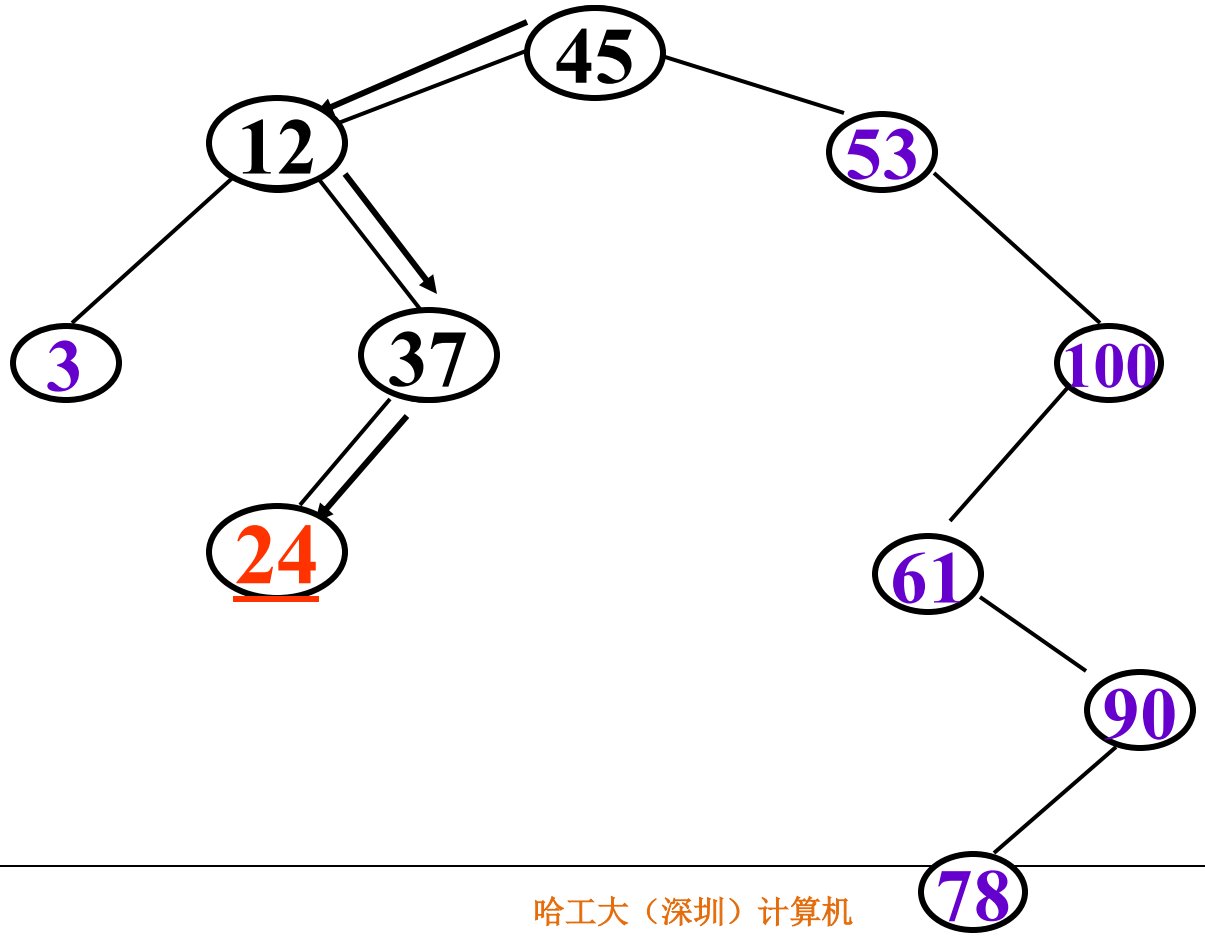
(3) 若给定值大于根结点的关键字，则在右子树上继续进行查找；

(4) 直到找到或查到空结点时为止。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找示例演示

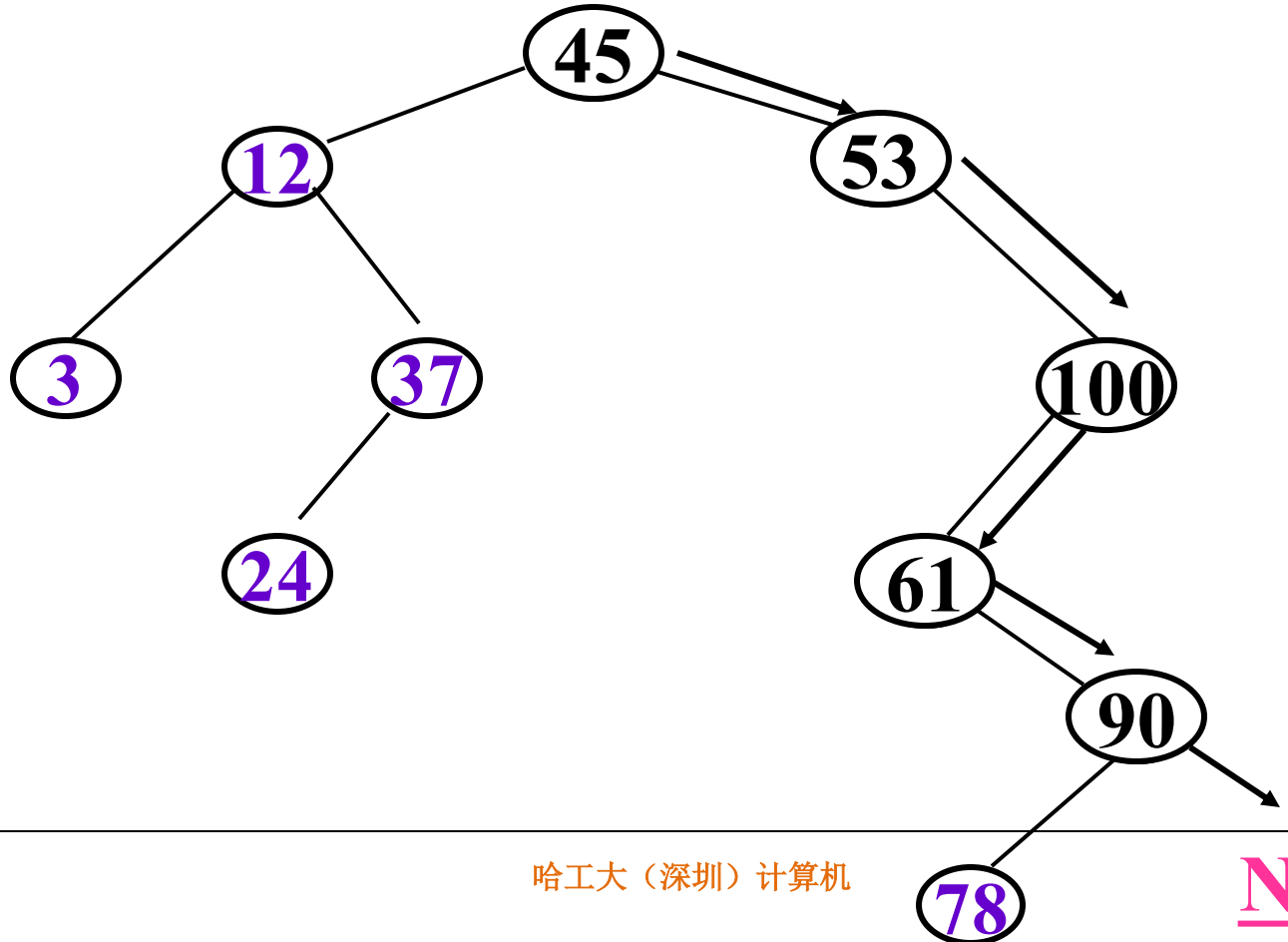
例 在图中查找关键字24



10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找示例演示

例 在图中查找关键字98



10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找过程分析

在查找过程中，生成了一条查找路径：

- **查找成功**：从根结点出发，沿着左分支或右分支逐层向下直至关键字等于给定值的结点；
- **查找不成功**：从根结点出发，沿着左分支或右分支逐层向下直至指针指向空树为止。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找算法

```
BiTree SearchBST(BiTree T,keytype k) {
    BiTree p=T;//非递归算法
    while(p!=NULL) {
        if (p->data.key==k)
            return p;
        if (k<p->data.key)
            p=p->lchild;
        else p=p->rchild;
    }
    return NULL;
}
```

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的查找算法-递归算法

Status SearchBST (BiTree T, KeyType key,
BiTree f, BiTree &p)

◆查找成功， 则返回指针 p 指向该数据元素的结点，并返回TRUE;

◆查找不成功，返回指针 p 指向查找路径上访问的最后一个结点，**用于后续的插入操作**， 并返回FALSE;

◆ 指针 f 指向当前访问的结点的**双亲**，其初始调用值为NULL;

10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找算法-递归算法

```
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) )
      SearchBST (T->lchild, key, T, p );
      // 在左子树中继续查找，f(双亲)指向当前的T
else  SearchBST (T->rchild, key, T, p );
                                     // 在右子树中继续查找
```

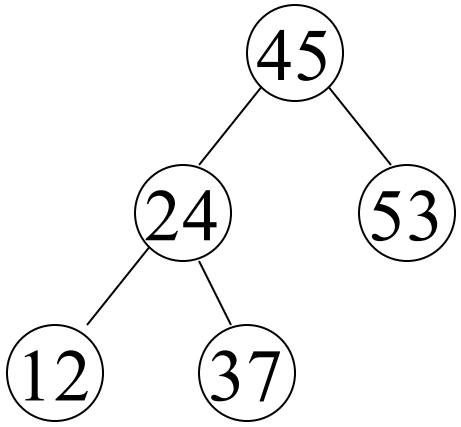
10.2.1 二叉排序树和平衡二叉树

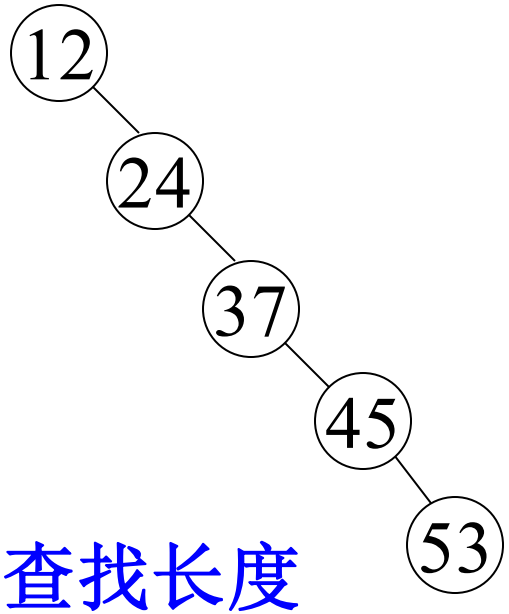
二叉排序树的查找算法分析

- 在二叉树上查找其关键字等于给定值的结点过程，恰是走了一条从根结点到该节点的路径的过程，和给定值比较的关键字个数等于路径长度加1（或结点所在层次数）。
- 因此，和折半查找类似，与给定值比较的关键字个数不超过树的深度。然而，折半查找长度为n的表的判定树是唯一的，而含有n个结点的二叉排序树却不唯一。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的查找算法分析





平均查找长度

$O(\log_2 n)$

平均查找长度

$O(n)$

$ASL = (1+2+2+3+3) / 5 = 2.2$

$ASL = (1+2+3+4+5) / 5 = 3$

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的查找算法分析

(1) 二叉排序树的平均查找长度最差情况与顺序表相同(关键字有序时),为O(n);

(2) 最好情况是二叉排序树的形态与折半查找的判定树相同,其平均查找长度和$\log_2 n$成正比;

(3) 二叉排序树的平均查找长度仍然是$O(\log_2 n)$。

思考题：若按中序对二叉查找树进行遍历,遍历序列有什么特点?

答案：按关键字升序排列。

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的插入算法思想

(1) 若二叉树为空：

则待插入结点*s作为根结点；

(2) 当二叉排序树非空时：

将待插结点关键字与根结点进行比较，若相等，则说明树中已有此结点，无须插入；

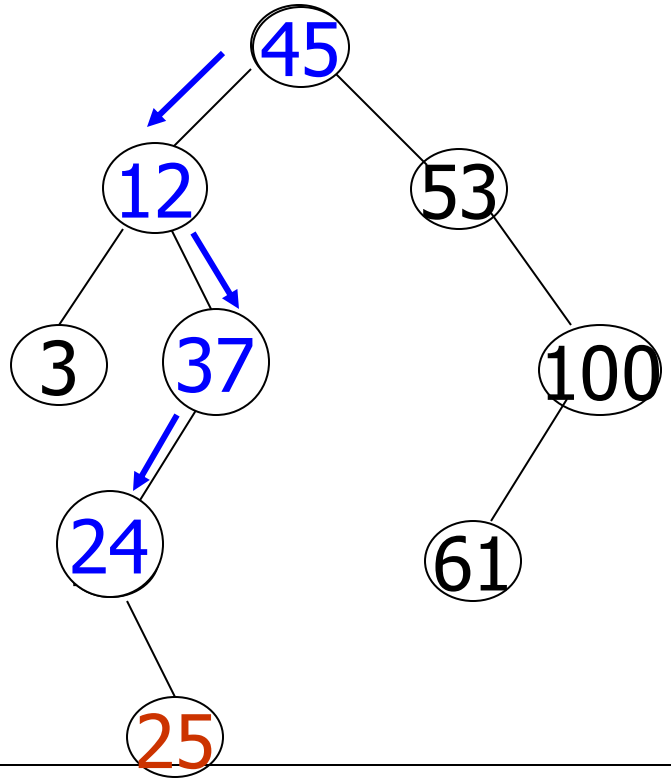
若小于根结点，插入左子树；

若大于根结点，插入右子树。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的插入过程演示

在如下二叉排序树中插入25过程演示



10.2.1 二叉排序树和平衡二叉树

二叉排序树的插入特点

- 这是一个二叉排序树的动态构造过程；
- 要插入关键值 x ，首先在二叉排序树中查找，若不存在则插入。
- 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结点的左孩子或右孩子结点。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的插入算法—递归

```
BSTNode *insert(BSTNode *root, BSTNode *p) {
    if (root= =NULL)  return(p);
    if(p->key<root->key)
        root->lch=insert(root->lch, p);
    else
        root->rch=insert(root->rch, p);
    return(root);
}
```

10.2.1 二叉排序树和平衡二叉树

二叉排序树的插入算法-非递归

```
Status InsertBST(BiTree &T, ElemType e) {
  BiTree p,s;
  if (!SearchBST(T, e.key, NULL, p)) {                    // 查找不成功
    s = (BiTree)malloc(sizeof(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为左孩子
    else
      p->rchild = s; // 插入s为右孩子
    return TRUE;
  } else return FALSE; // 树中已有关键字相同的结点，不再插入
} // Insert BST
```

查找不成功时，指针p赋值为返回指针 p 指向查找路径上访问的最后一个结点

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的插入算法分析

(1)二叉排序树的插入算法的时间复杂性与查找算法的时间复杂性相同(插入的前提是查找);

(2)最好情况是O（$\log_2 n$）;最坏情况是O(n);平均情况是O($\log_2 n$)。

思考题：如何生成二叉排序树?

答案：从空树开始循环调用插入算法。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的生成

```
bstnode*CreatBST() {
    bstnode *t,*s;
    keytype key, endflag=0;
    t=null;
    while(key!=endflag) {
        s=malloc(sizeof(bstnode));
        s->key=key;
        s->lch=s->rch=null;
        t=insertBST(t,s); //把s插入到t中
    }
    return t;
}
```

10.2.1 二叉排序树和平衡二叉树

◘ 二叉排序树的结点删除定义

删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。

可分三种情况讨论：

- 被删除的结点是叶子；
- 被删除的结点只有左子树或者只有右子树；
- 被删除的结点既有左子树，也有右子树。

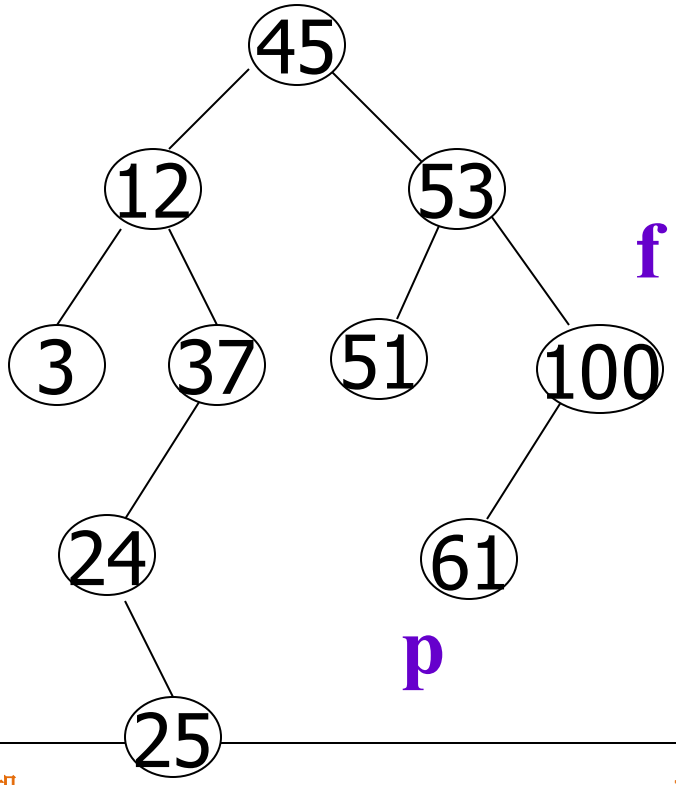
10.2.1 二叉排序树和平衡二叉树

二叉排序树的结点删除方法--p为叶结点

设被删结点为p,其双亲结点为f，且p为叶结点

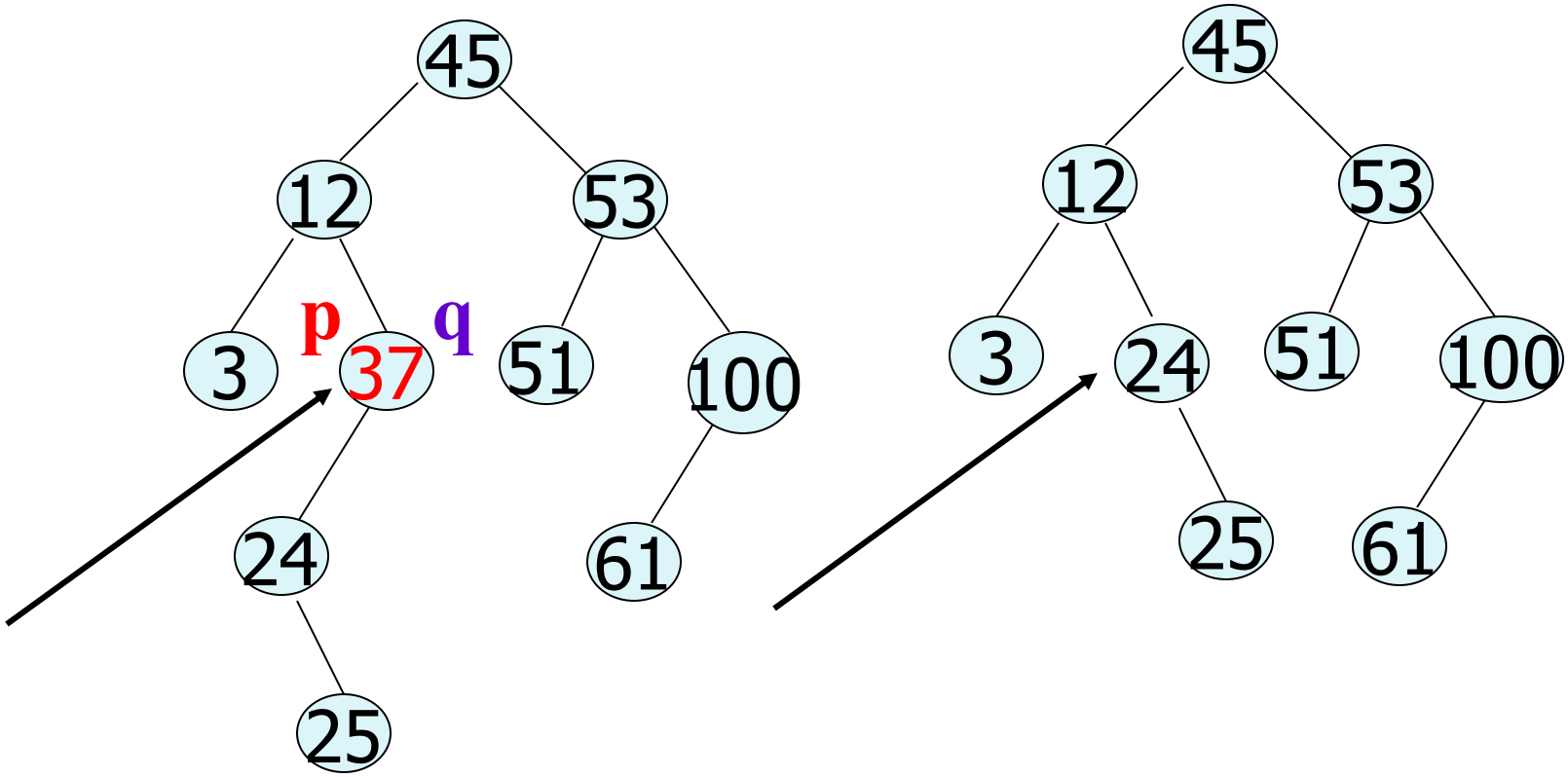
◆ 删除方法：

释放结点p，修改其双亲结点f的相应指针。



10.2.1 二叉排序树和平衡二叉树

二叉排序树的结点删除方法—右子树空

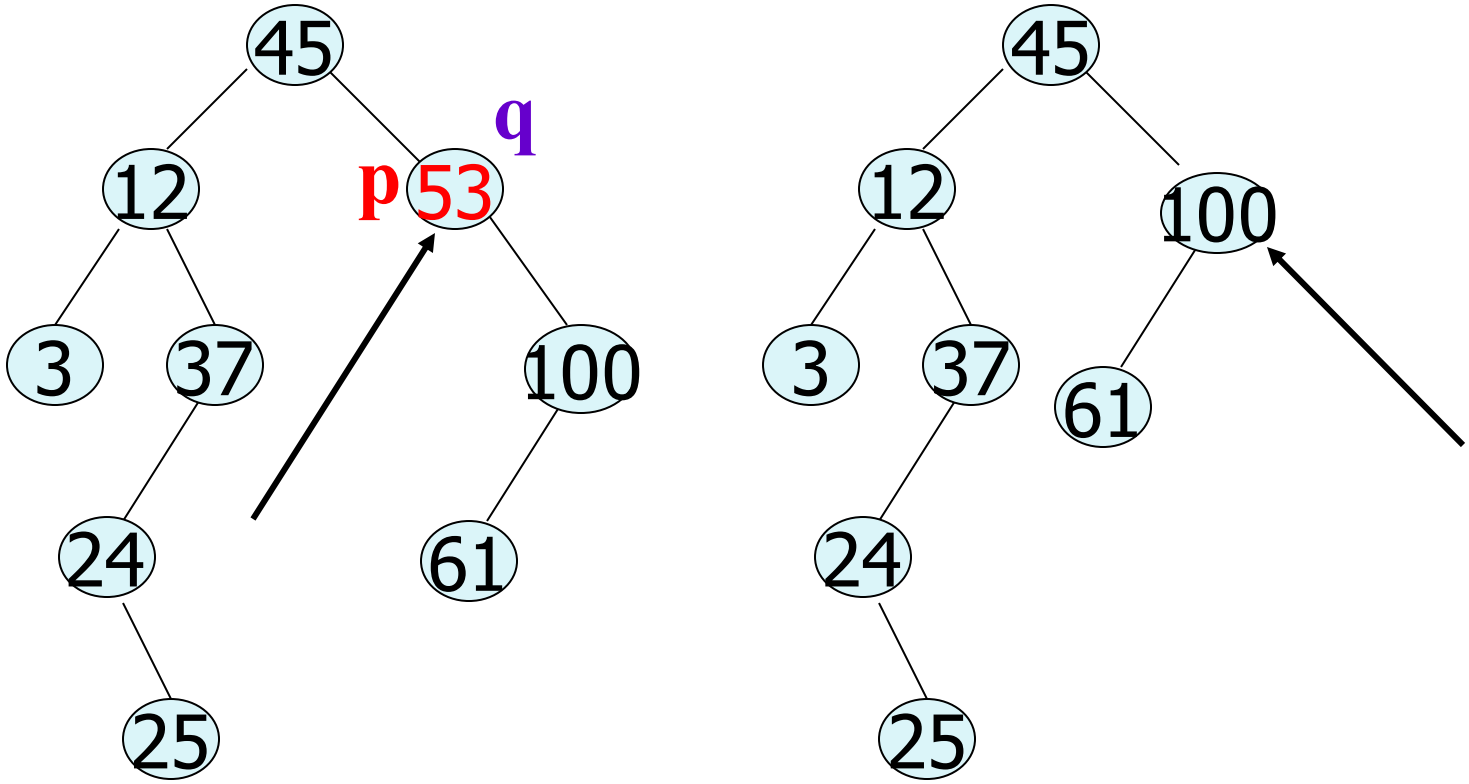


◆删除方法：p的右子树空，则用左子树顶替

◆q = p; p = p->lchild; free(q);

10.2.1 二叉排序树和平衡二叉树

二叉排序树的结点删除方法—左子树空

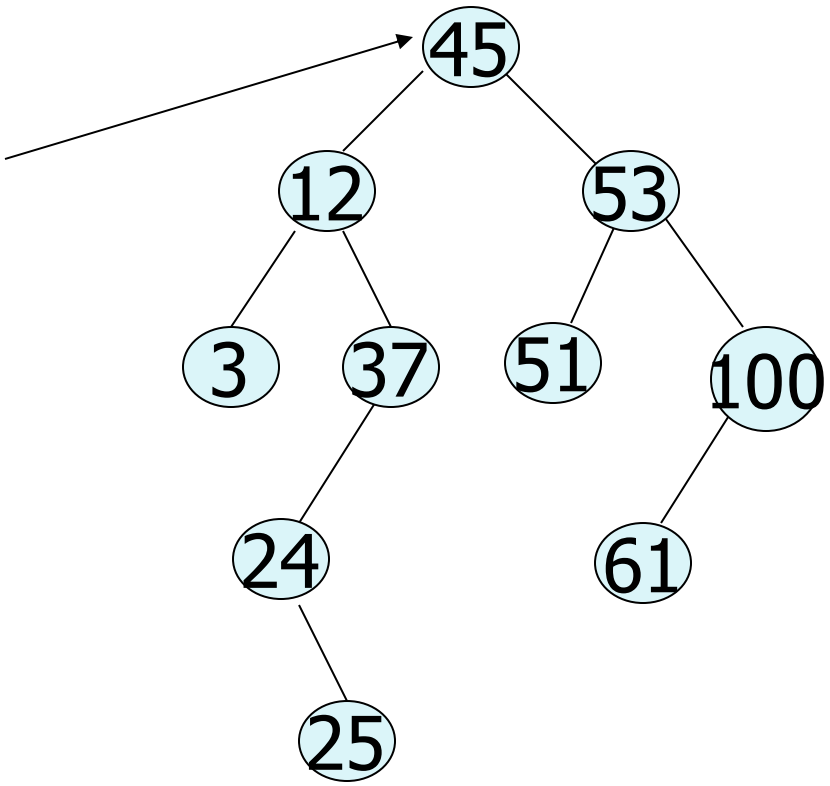


◆删除方法： p的左子树空，则用右子树顶替

◆q = p; p = p->rchild; free(q);

10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的结点删除方法--左右子树均不空



◆删除方法一：

•把左子树作为右子树中最小结点的左子树。

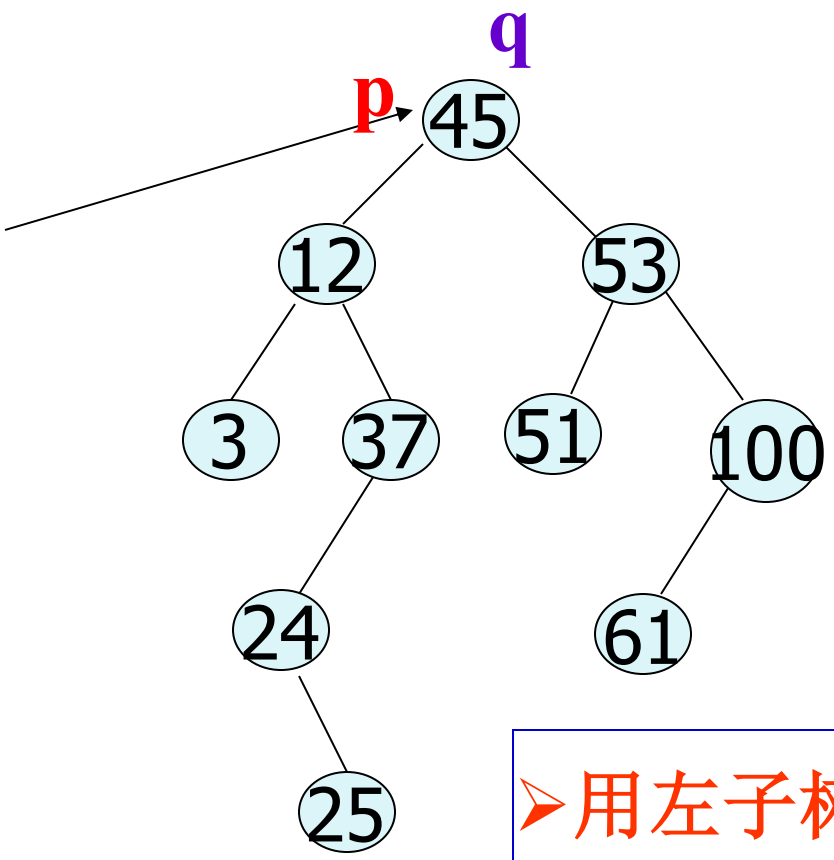
•或者把右子树作为左子树中最大结点的右子树。

缺点：

增加了树的高度。

10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的结点删除方法—左右子树均不空



◆删除方法二（改进）：

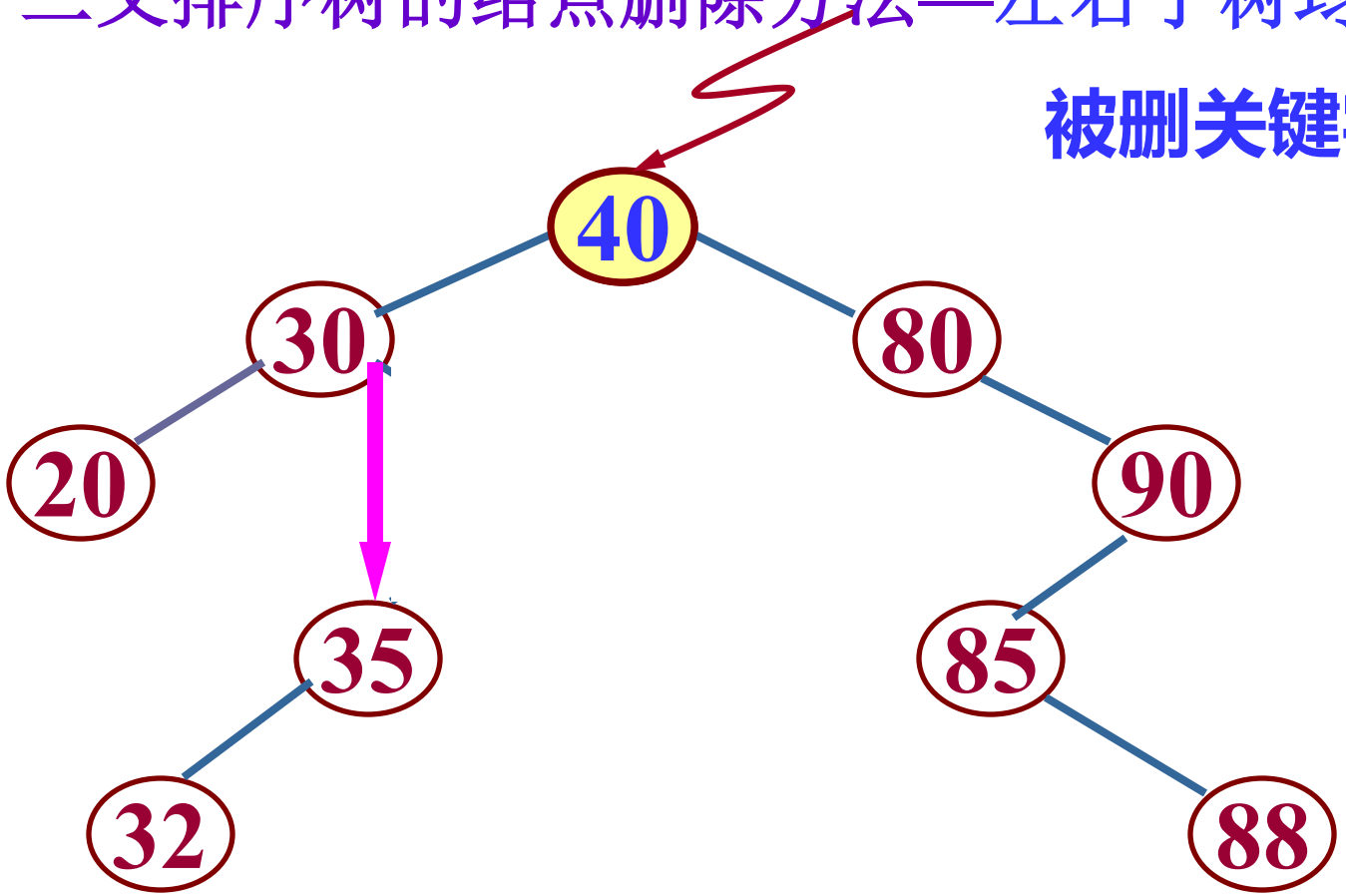
找一个结点顶替它的位置。

- 用左子树中最大的结点（前驱）替换；
- 右子树中最小的结点（后继）替换；

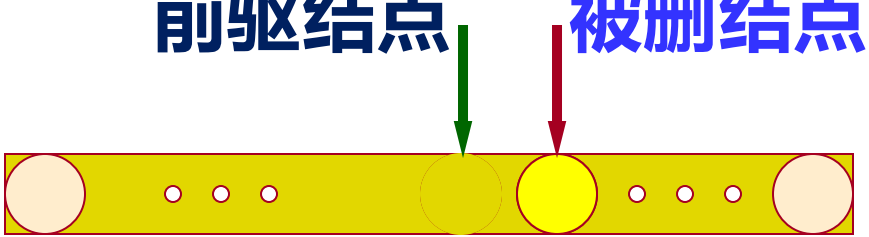
10.2.1 二叉排序树和平衡二叉树

- 二叉排序树的结点删除方法—左右子树均不空

被删关键字 = 50



前驱结点　　被删结点

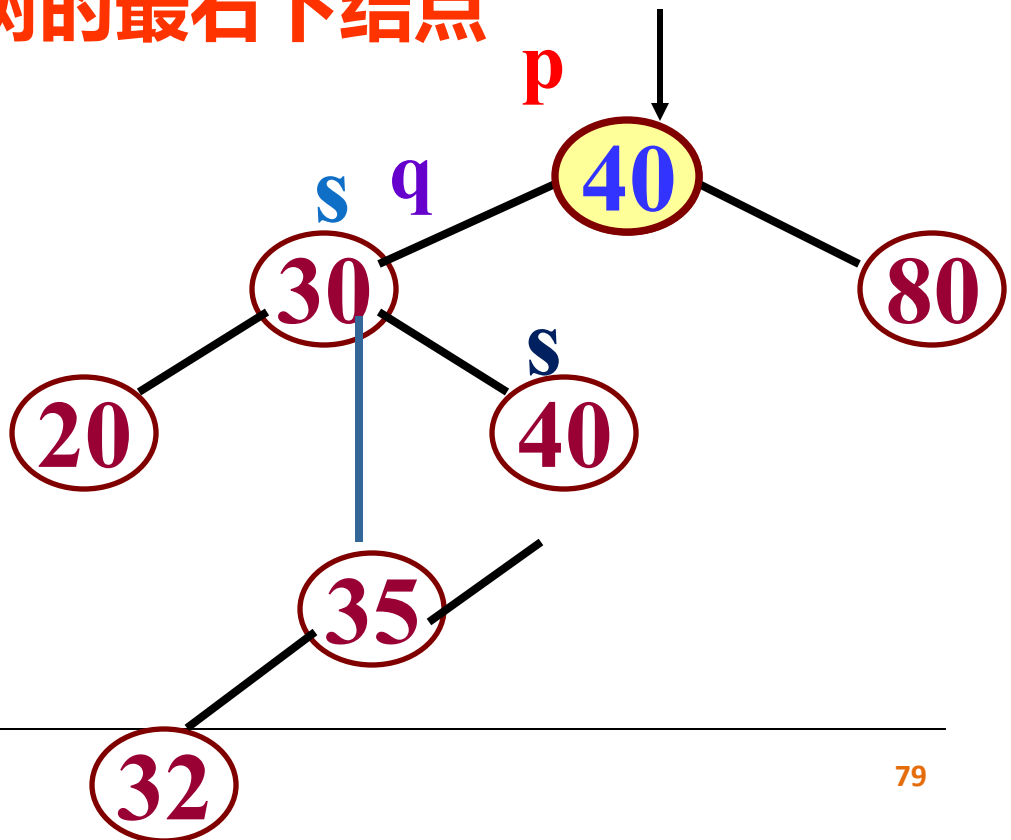


以其前驱替代之，然后再删除该前驱结点

10.2.1 二叉排序树和平衡二叉树

二叉排序树的结点删除方法—左右子树均不空

```
q = p;  s = p->lchild;//  定义s 指向被删结点的前驱
while (!  s->rchild) { q = s;  s = s->rchild; }
//s右子树不空，找右子树的最右下结点
 p->data = s->data;
q->rchild = s->lchild;
```



10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的结点删除算法-递归算法

```
Status DeleteBST(BiTree &T,  KeyType key) {
   if (!T) return FALSE;
   else {
      if (EQ(key, T->data.key))
         return Delete(T); //该函数怎么实现？看下页
      else if (LT(key, T->data.key))
         return DeleteBST(T->lchild, key);
      else
         return DeleteBST(T->rchild, key);
  }
} // DeleteBST
```

10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的结点删除算法-非递归算法

```
Status Delete(BiTree &p) {
   BiTree q, s;
   if (!p->rchild)//右子树为空把左子树挂上
      {q = p;  p = p->lchild;  free(q);}
   else if (!p->lchild) // 左子树空则右子树挂上
      {q = p;  p = p->rchild;  free(q);}
   else { //左右子树均不空
      q = p;  s = p->lchild; //找在左子树中前驱
      while (!s->rchild)//前驱在左子树的最右下
         {q = s;  s = s->rchild; } //顺着右子树往下走
         p->data = s->data;
         if (q != p) q->rchild = s->lchild;
         else q->lchild = s->lchild;
         free(s);
   }
   return TRUE;
} // Delete
```

10.2.1 二叉排序树和平衡二叉树

□ 二叉排序树的删除算法分析

(1)二叉排序树的删除算法的时间复杂性与查找算法的时间复杂性相同;

(2)最好情况是O（$\log_2 n$）;最坏情况是O(n);平均情况是O($\log_2 n$)。

10.2.1 二叉排序树和平衡二叉树

二叉排序树的小结

- 若从空树出发，经过一系列的查找插入操作之后，可生成一棵二叉排序树。
- 中序遍历二叉排序树可得到一个关键字有序的序列。
- 构造二叉排序树的过程就是一个对无序序列进行排序的过程。
- 在二叉排序树上插入一个记录或结点，不需要移动其它记录或结点。

10.2.1 二叉排序树和平衡二叉树

□ 平衡二叉树（AVL树）

由阿德尔森—维尔斯和兰迪斯(Adelson-Velskii and Landis)于1962年首先提出的，所以又称为AVL树。

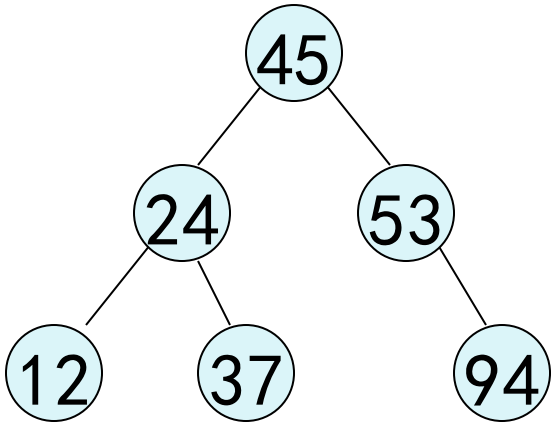
□平衡二叉树（AVL树）的定义

一棵AVL树或者是空树，或者是具有下列性质的二叉排序树：

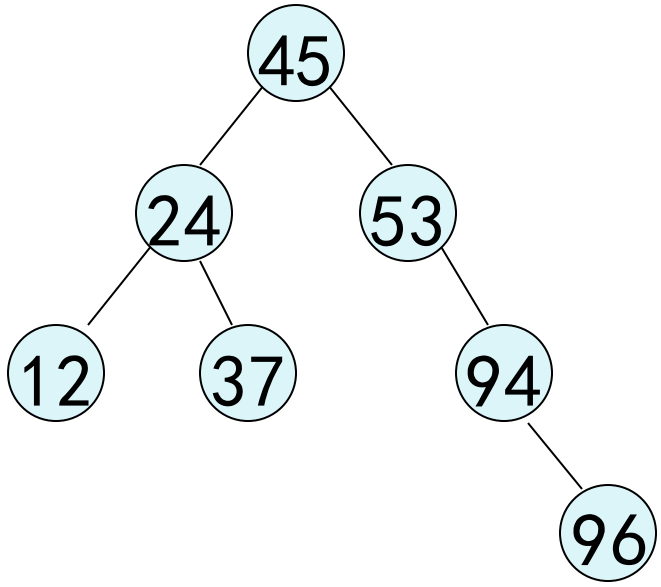
- 它的左子树和右子树都是AVL树；
- 且左子树和右子树的高度之差的绝对值不超过1。

10.2.1 二叉排序树和平衡二叉树

平衡二叉树示例



平衡二叉树



非平衡二叉树

10.2.1 二叉排序树和平衡二叉树

◘ 平衡因子的定义

平衡因子(BF--Balance Factor):
任一结点的左子树的高度减去右子树的高度所得的高度差称为该结点的平衡因子BF。

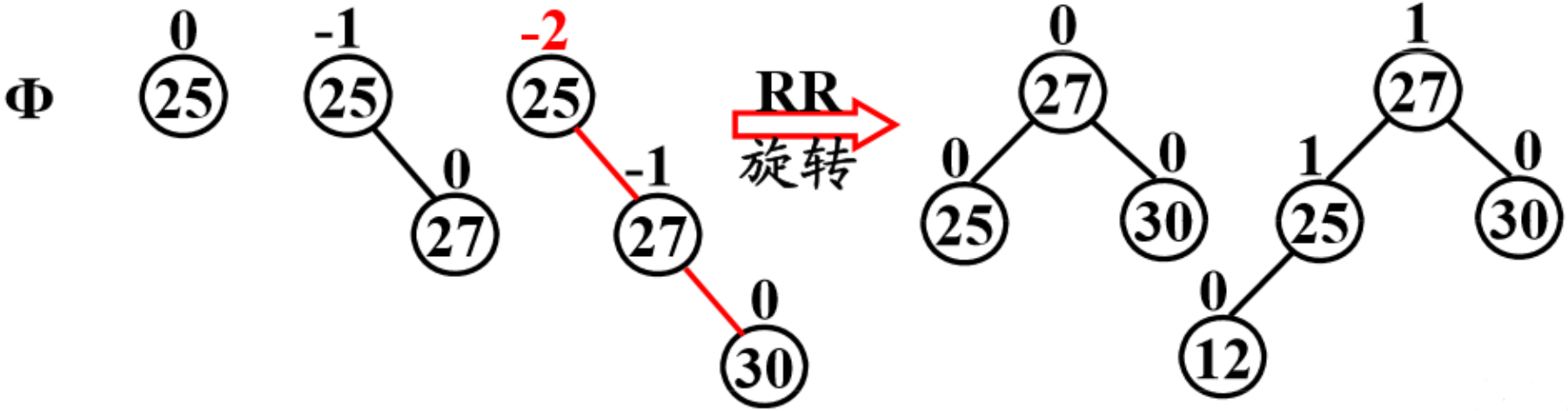
根据AVL树的定义，任一结点的平衡因子只能取-1，0 和 1。

一棵平衡二叉排序树如果有n个结点，其高度可保持在$O(\log_2 n)$，平均查找长度也可保持在$O(\log_2 n)$。

10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

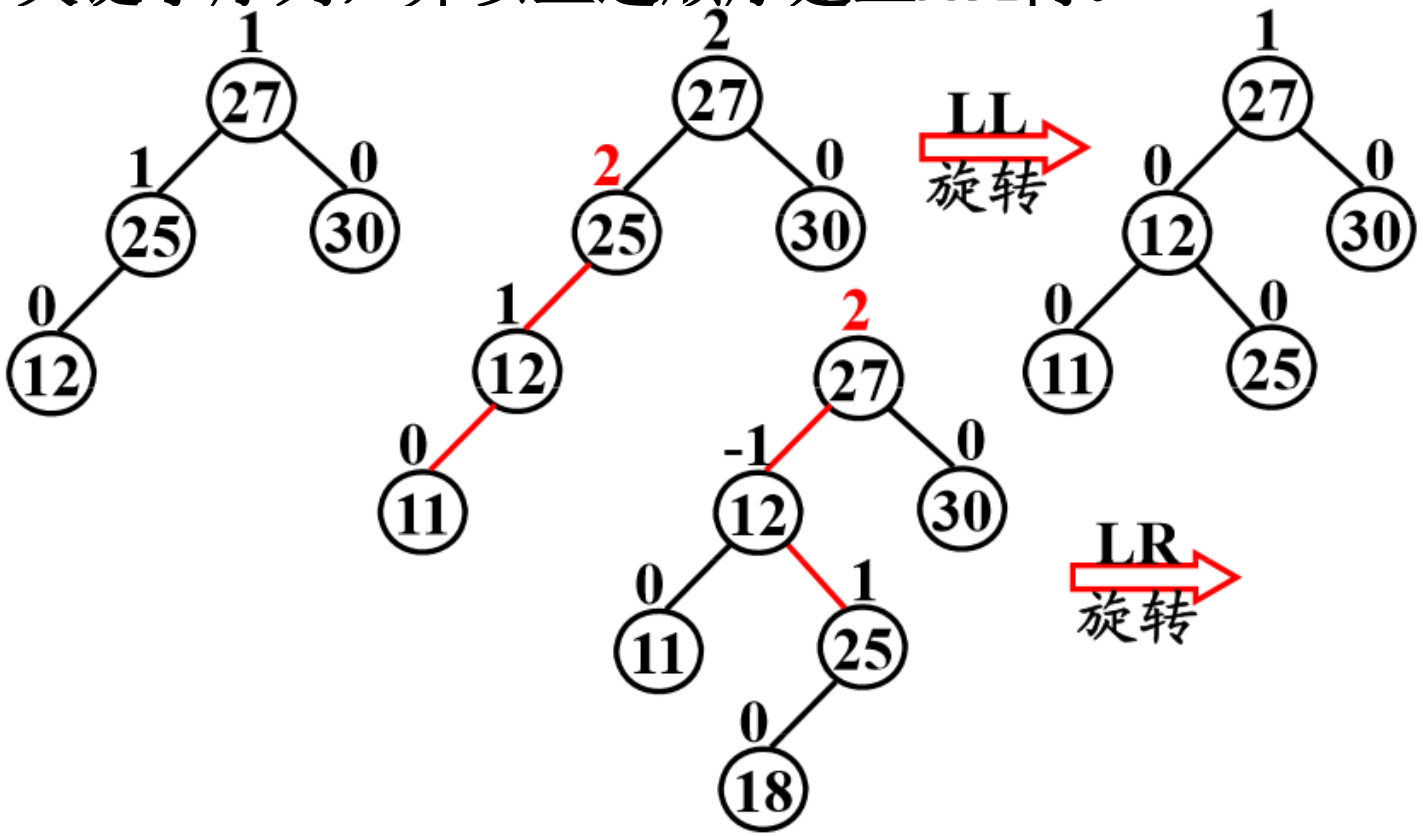
- 向AVL树插入结点可能造成不平衡，此时要调整树的结构，使之重新达到平衡。
- 我们希望任何初始序列构成的二叉排序树都是AVL树，这样它的深度和平均查找长度都是和logn是同量级的。
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

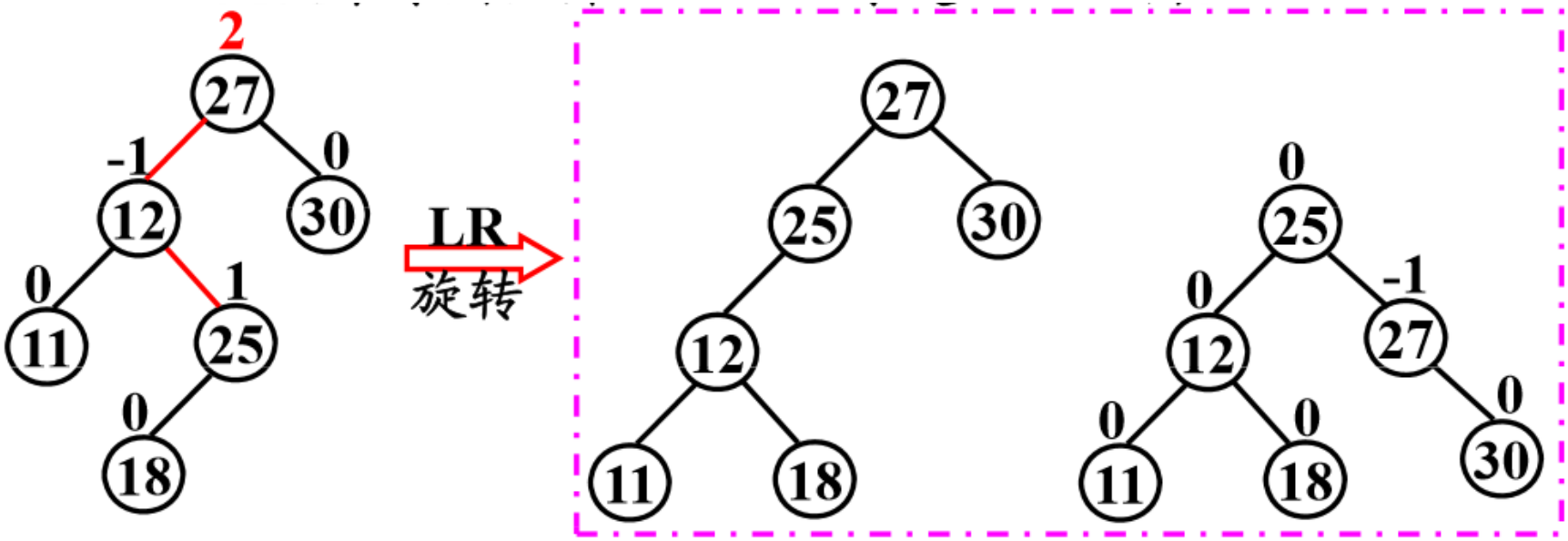
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

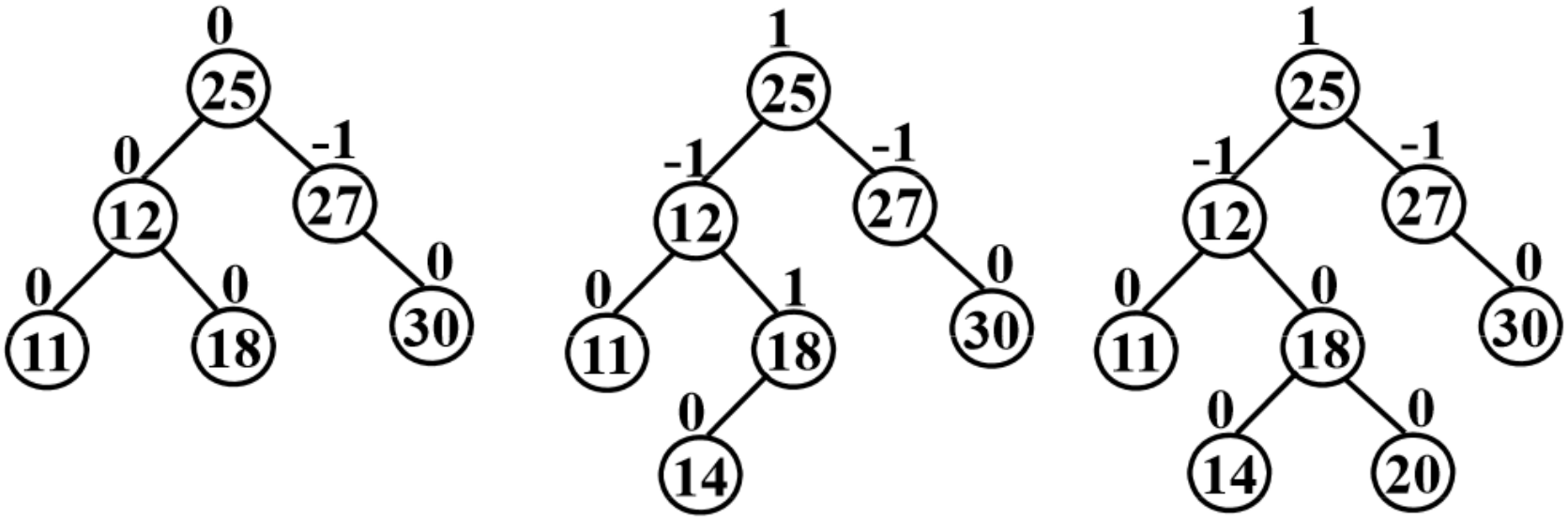
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

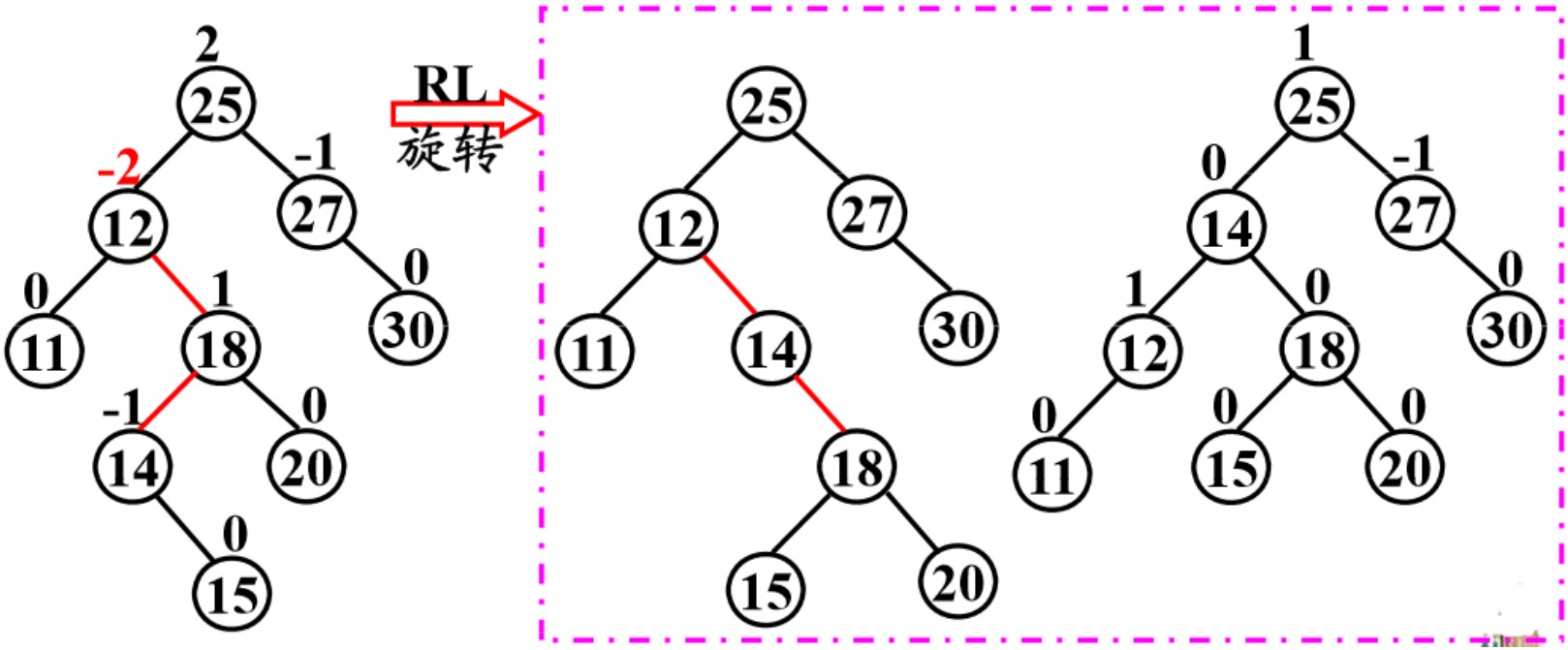
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

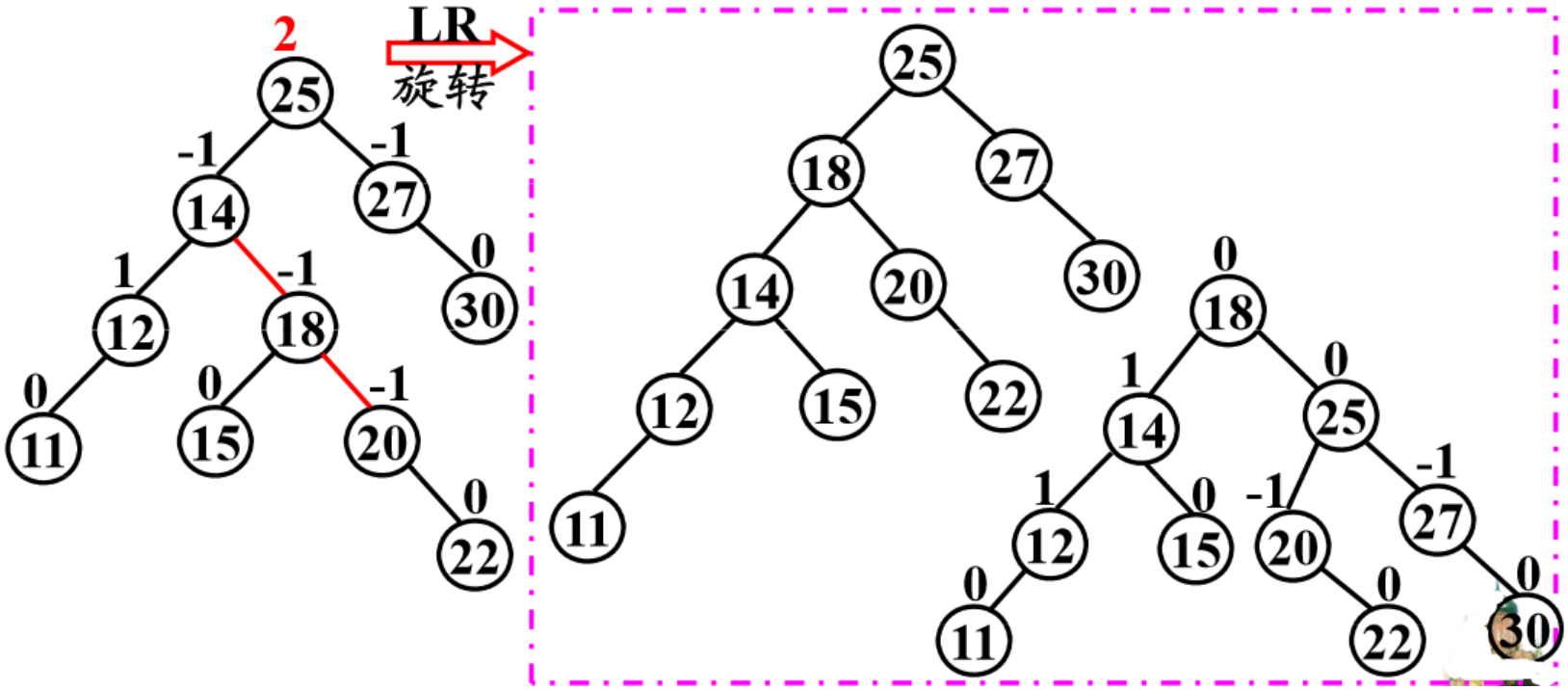
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

- 在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。
- 若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为±2的祖先结点。
- 可以用4种旋转进行平衡化处理:

①LL型：新结点Y 被插入到A 的左子树的左子树上（顺）

②RR型：新结点Y 被插入到A 的右子树的右子树上（逆）

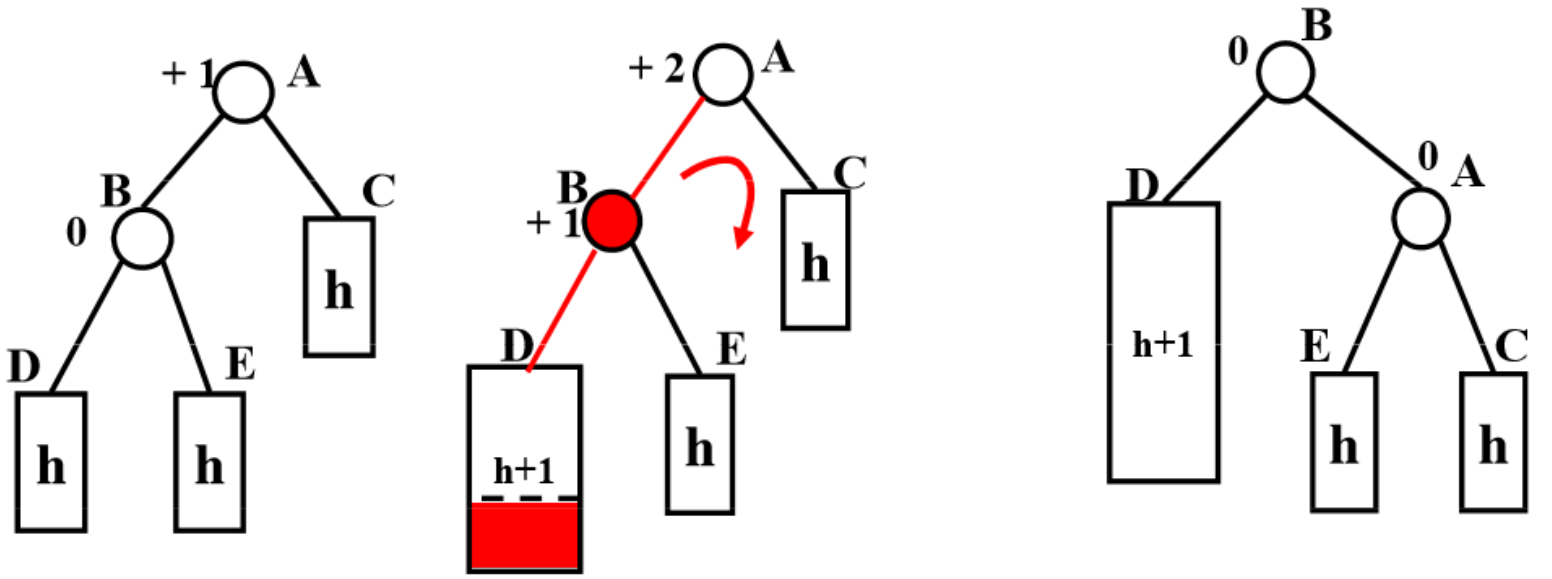
③LR型：新结点Y被插入到A的左子树的右子树上（逆、顺）

④RL型：新结点Y 被插入到A 的右子树的左子树上（顺、逆）

10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

- LL型：新结点Y被插入到A的左子树的左子树上（顺）



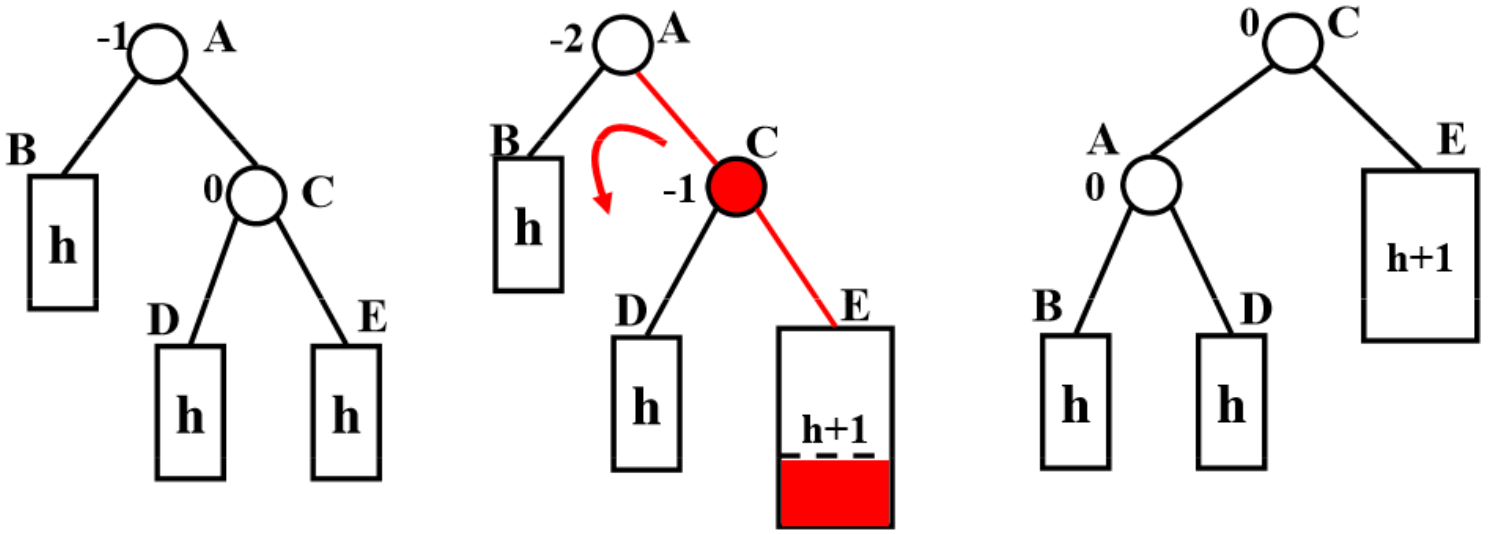
(a)AVL树　　(b) D子树中插入结点　　(c)右向旋转后的AVL树

平衡处理为：旋转的结果是保持中序遍历的次序不变。将A顺时针旋转，成为B的右子树，而原来B的右子树则变成A的左子树。

10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

- RR型：新结点Y被插入到A的右子树的右子树上（逆）



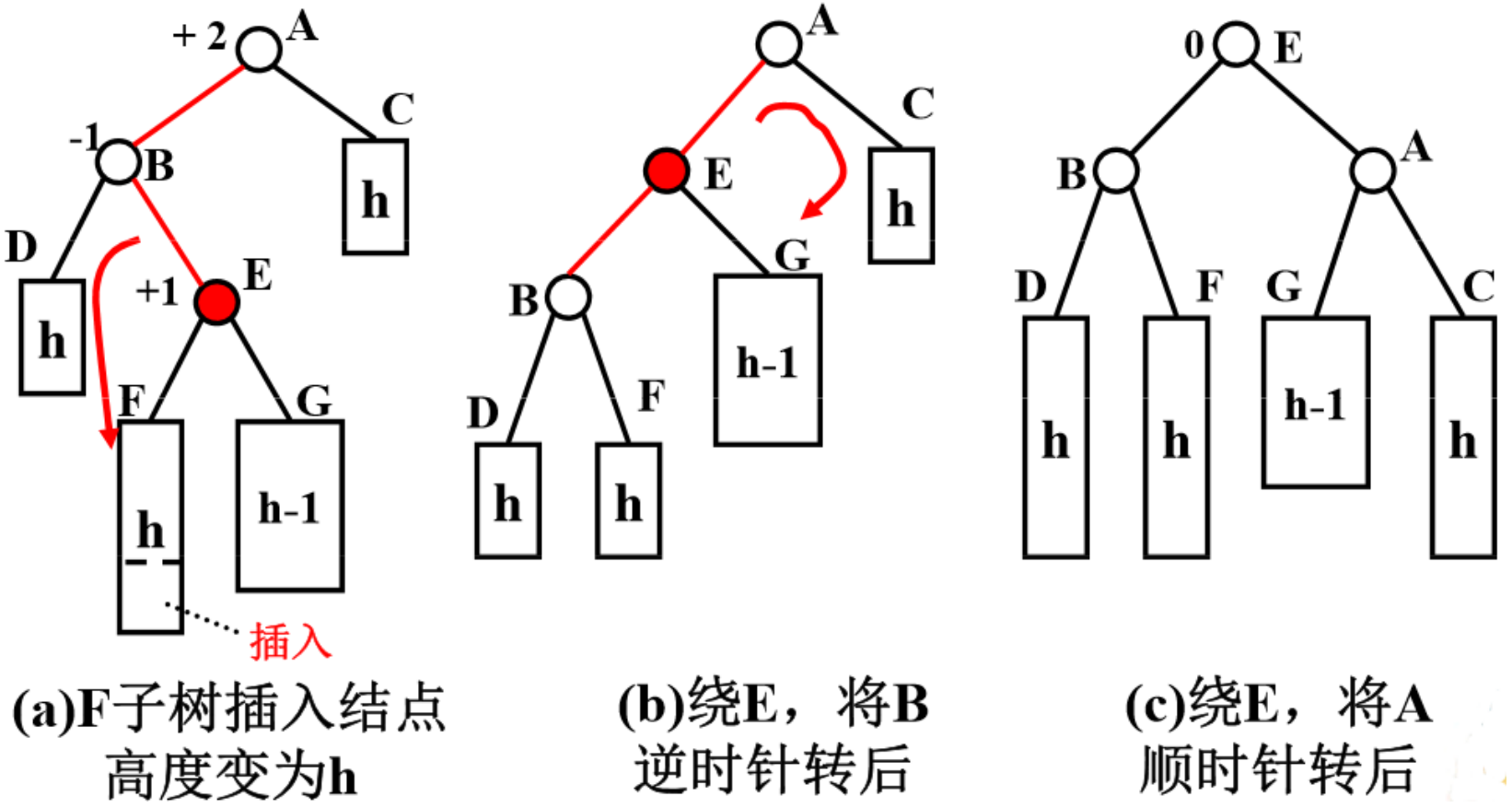
(a)AVL树　　(b) E子树中插入结点　　(c)左向旋转后的AVL树

平衡处理为：旋转的结果是保持中序遍历的次序不变。将A逆时针旋转，成为C的左子树，而原来C的左子树则变成A的右子树。

10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

- LR 型：新结点Y 被插入到A 的左子树的右子树上(逆,顺)



(a)F子树插入结点高度变为h

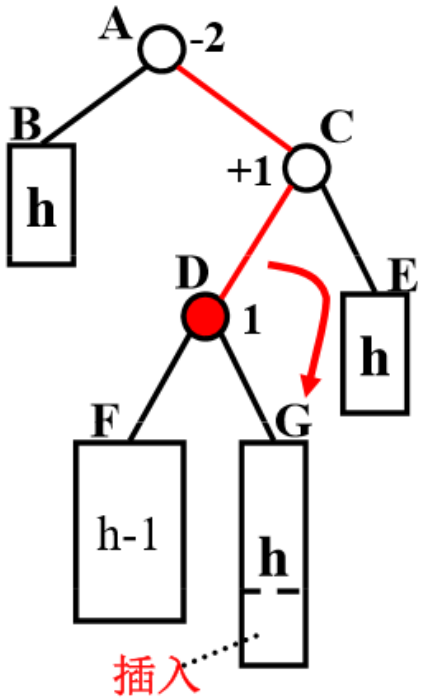
(b)绕E，将B逆时针转后

(c)绕E，将A顺时针转后

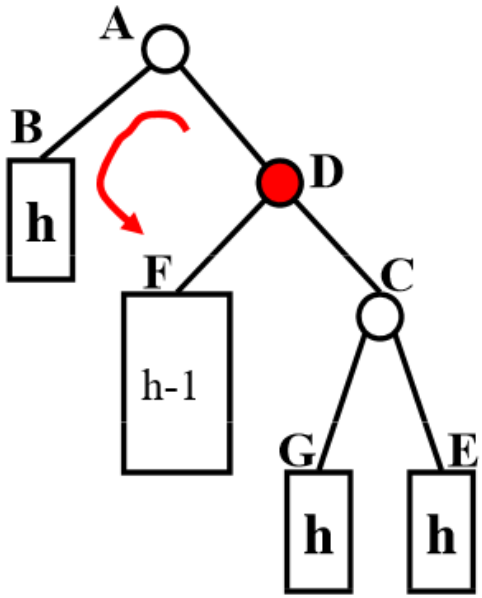
10.2.1 二叉排序树和平衡二叉树

AVL树的平衡化处理

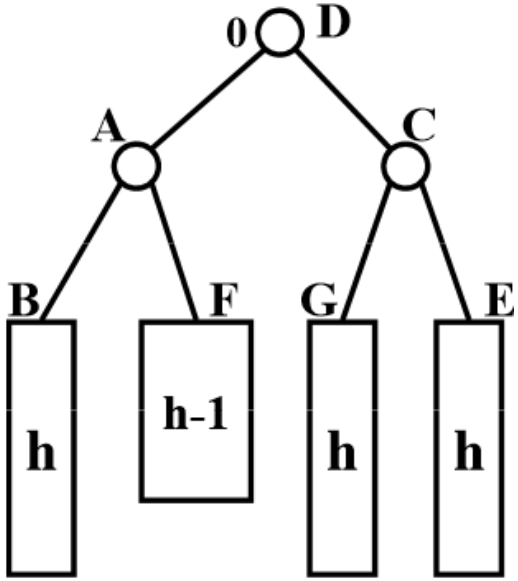
- RL 型：新结点Y 被插入到A 的右子树的左子树上(顺,逆)



(a) G子树插入结点
高度变为h



(b) 绕D，C顺时
针转之后



(c) 绕D，A逆时
针转之后

10.2.1 二叉排序树和平衡二叉树

AVL树的插入操作与建立

- 对于一组关键字的输入序列，从空开始不断地插入结点，最后构成AVL树；
- 每插入一个结点后就应判断从该结点到根的路径上有无结点发生不平衡；
- 如有不平衡问题，利用旋转方法进行树的调整，使之平衡化；
- 建AVL树过程是不断插入结点和必要时进行平衡化的过 程。

10.2.1 二叉排序树和平衡二叉树

AVL树的删除操作

- 删除操作与插入操作是对称的（镜像），但可能需要的平衡化次数多。
- 平衡化不会增加子树的高度，但可能会减少子树的高。
- 在有可能使树增高的插入操作中，一次平衡化能抵消掉树增高；
- 而在有可能使树减低的删除操作中，平衡化可能会带来祖先结点的不平衡。

10.2 动态查找表

10.2.1 二叉排序树和平衡二叉树

10.2.2 B_树和B+树

10.2.2 B-树和B+树

□ B-树的引入

当数据量太大时，不可能一次调入内存,要多次访问外存。硬盘的访问速度慢。主要矛盾变为减少访问外存次数。

用二叉树组织文件，若在查找时一次调入一个结点进内存，当文件的记录个数为100000时,要找到给定关键字的记录，平均需访问外存17次(log100000)

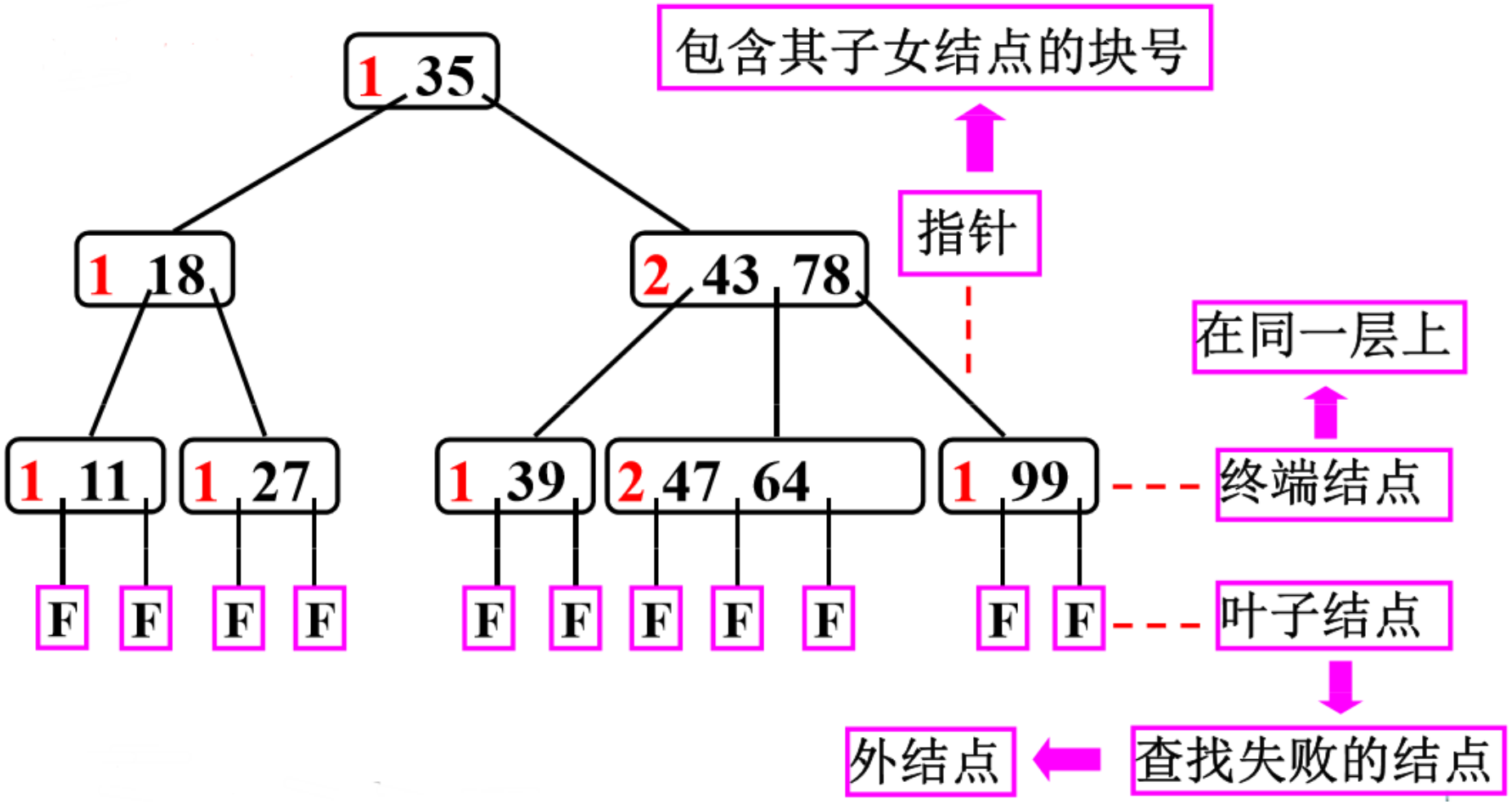
10.2.2 B-树和B+树

□ B-树的引入

- 当符号表的大小超过内存容量时，由于必须从磁盘等辅助存储设备上去读取这些查找树结构中的结点，每次只能根据需要读取一个结点，因此，AVL树性能就不是很高。
- 在AVL树在结点高度上采用相对平衡的策略，使其平均性能接近于BST的最好情况下的性能。如果保持查找树在高度上的绝对平衡，而允许查找树结点的子树个数（分支个数）在一定范围内变化，能否获得很好的查找性能呢？
- 基于这样的想法，人们设计了许多在高度上保持绝对平衡，而在宽度上保持相对平衡的查找结构 。
- 如B-树及其各种变形结构，这些查找结构不再是二叉结构，而是m-路查找树（m-way search tree），且以其子树保持等高为其基本性质，在实际中都有着广泛的应用。

10.2.2 B-树和B+树

□ B-树的示例



10.2.2 B-树和B+树

B-树的定义

一棵m阶B-树，它或者为空，或者满足以下性质的m叉树：

(1)树中每个结点最多有m个子树；

(2)若根结点不是叶子结点至少有2棵子树。(2~m)

(3)除根结点以外的所有非叶结点至少有⌈m/2⌉棵子树。$\lceil m/2 \rceil$~m

(4)所有的终端结点都位于同一层，所有的叶子结点（失败结点）都位于同一层，并且叶子结点不带信息（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）。（高度绝对平衡）

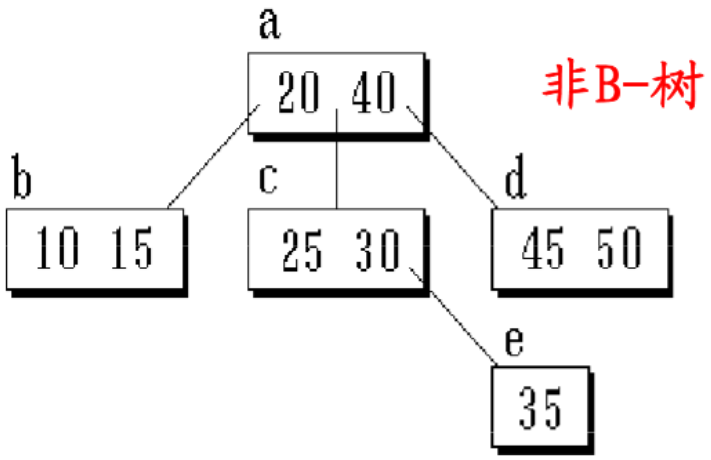
(5)所有的非叶子结点中包含下列信息数据（n，A_0，K_1，A_1，K_2，…，K_n，A_n）

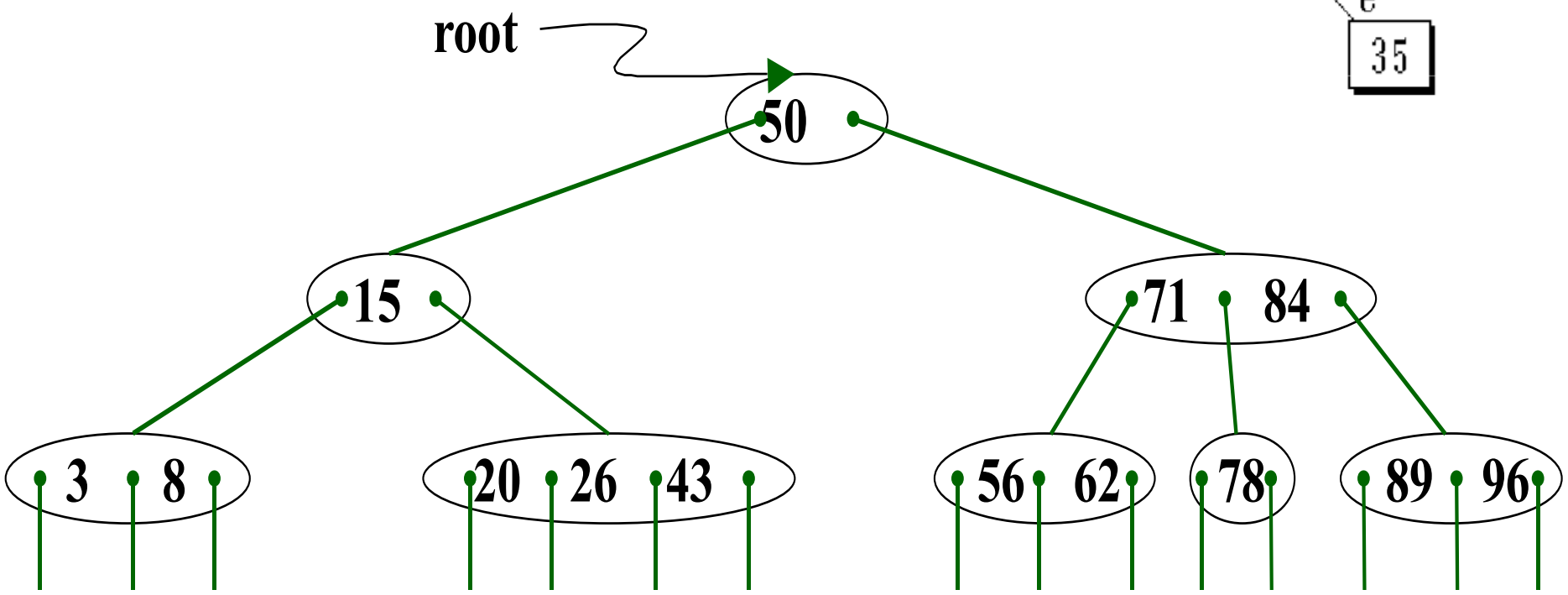
n为关键字的个数，$\lceil m/2 \rceil - 1 \le n \le m - 1$(n比子树的数量少1)，$K_i$为关键字,且$K_i<K_{i+1}$,$A_i$为指向子树根节点的指针，且$A_i$所指子树中所有结点的关键字均小于$K_{i+1}$,大于$K_i$；$A_n$所指子树中所有结点的关键码均大于$K_n$，每棵子树$A_i$都是m-路B-树，0≦i≦n。

10.2.2 B-树和B+树

□ B-树的示例

B-树是一种平衡的多路查找树：





10.2.2 B-树和B+树

□多叉树的特性

在 m 阶的B-树上，每个非终端结点可能含有：

- **n 个关键字 K_i（$1 \le i \le n$）** $\lceil m/2 \rceil - 1 \le n \le m - 1$
- **n 个指向关键字(记录)的指针 D_i（$1 \le i \le n$）**
- **$n+1$ 个指向子树的指针 A_i（$0 \le i \le n$）**

10.2.2 B-树和B+树

□ B-树-查找树的特性

- 非叶结点中的多个关键字均自小至大有序排列，即：$K_1 < K_2 < \cdots < K_n$ ；
- A_{i-1} 所指子树上所有关键字均小于K_i ；
- A_i 所指子树上所有关键字均大于K_i ；

10.2.2 B-树和B+树

□ B-树-平衡树的特性

- 树中所有叶子结点均不带信息，且在树中的同一层次上；
- 根结点或为叶子结点，或至少含有两棵子树；
- 其余所有非叶结点均至少含有$\lceil m/2 \rceil$棵子树；
- 至多含有 m 棵子树；

10.2.2 B-树和B+树

□ B-树的查找

B-树的查找类似二叉排序树的查找，所不同的是B-树每个结点上是多关键码的有序表:

(1)在到达某个结点时，先在(多关键码的)有序表中查找，若找到，则查找成功；

(2)否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。

即在B-树上的查找过程是一个顺指针查找结点和在结点中查找关键码交叉进行的过程。

10.2.2 B-树和B+树

□ B-树的查找

查找过程：

- 从根结点出发，沿指针搜索结点和在结点内进行顺序（或折半）查找两个过程交叉进行。

- 若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；

- 若查找不成功，则返回插入位置。

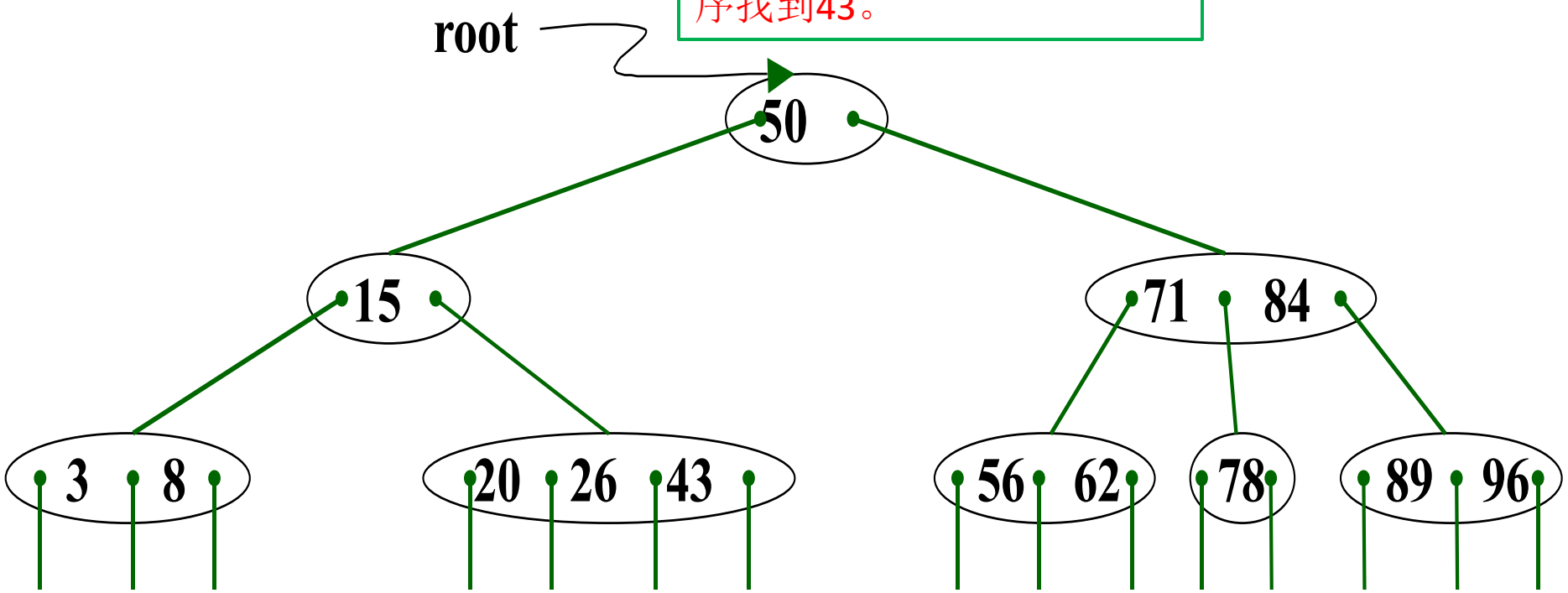
10.2.2 B-树和B+树

□ B-树的查找

例如: 4 阶B-树

比如查找43，先从根节点开始，比50小，左指针查，比15大，沿15后面的指针查找，最后在15的右子树节点中顺序找到43。

查找43



10.2.2 B-树和B+树

□ B-树-插入

- 在查找不成功之后，需进行插入。
- 由于B-树结点中的关键字个数必须≥ $\left\lceil \frac{m}{2} \right\rceil - 1$，因此每次插入一个关键字不是在树中添加一个叶子结点，而是在最下层的非叶结点中添加一个关键字，有下列几种情况：

（1）插入后，该结点的关键字个数n<m（or≤m-1），不修改指针；

（2）若双亲为空，则建新的根结点。

10.2.2 B-树和B+树

□ B-树-插入

（3）插入后，若该结点的关键字个数 n=m，则需进行“结点分裂”，令 s = ⌈m/2⌉，在原结点中保留

(s-1，A_0，K_1，…… ， K_{s-1}，A_{s-1})；

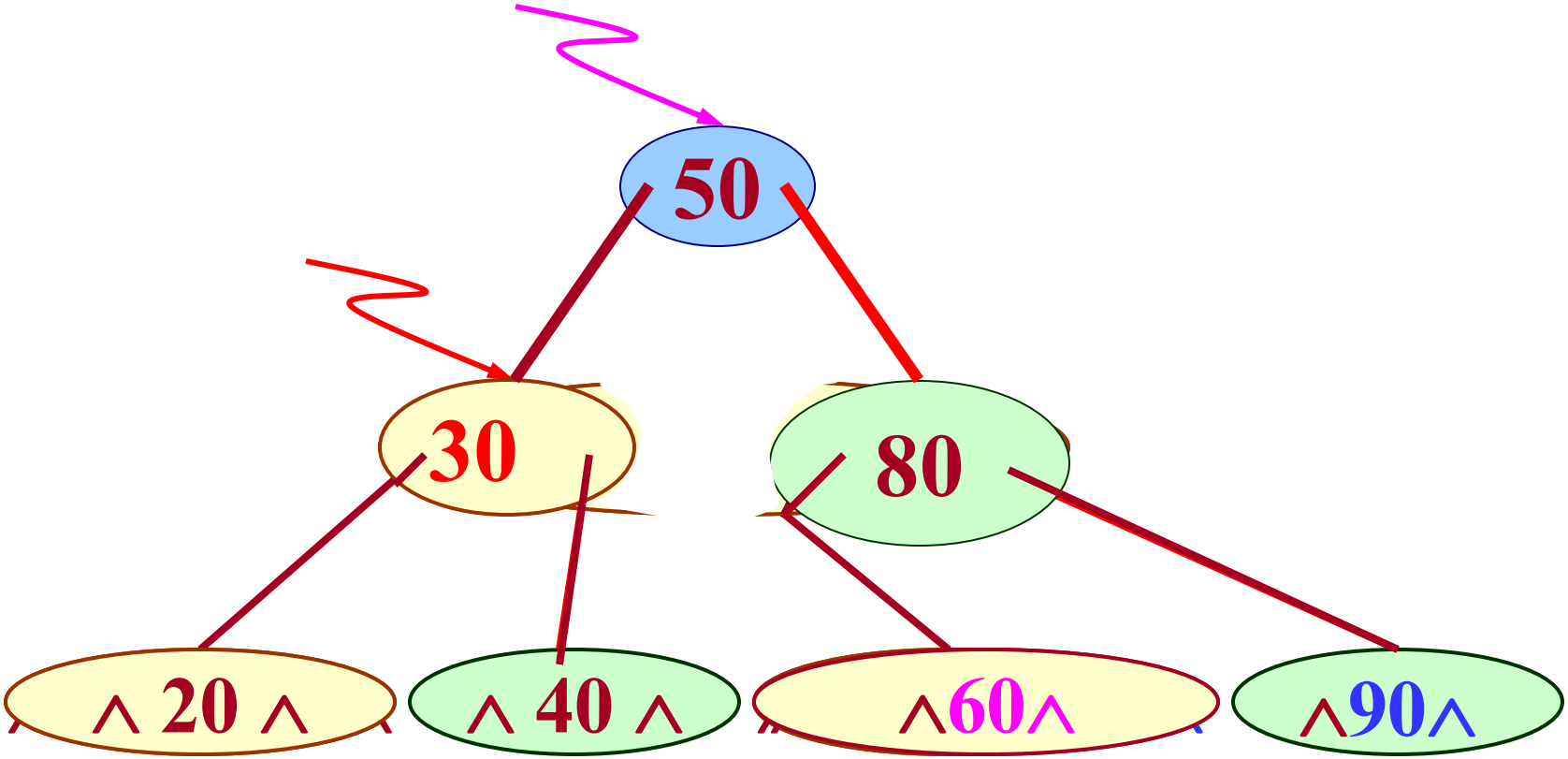
建新结点*q

(m-s, A_s，K_{s+1}，…… ，K_m，A_m)；

将(K_s，q)插入双亲结点；其中q为指向新结点的指针

10.2.2 B-树和B+树

例如:下列为 3 阶B-树



插入关键字 = 60, 90, 30,

10.2.2 B-树和B+树

□ B-树的插入操作与建立

- B-树是从空树起，逐个插入关键字而生成的。
- 在m阶B-树中，每个非失败结点的关键字个数都在[⌈m/2⌉-1, m-1]之间。
- 插入操作，首先执行查找操作以确定可以插入新关键字的终端结点p；
- 如果在关键字插入后，结点中的关键字个数超出了上界m-1，则结点需要“分裂”，否则可以直接插入。
- 实现结点“分裂”的原则是：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：

$$(m, A_0, K_1, A_1, K_2, A_2, \ldots\ldots, K_m, A_m)$$

$$\text{其中 } K_i < K_{i+1}, 1 \le i < m$$

10.2.2 B-树和B+树

□ B-树的插入操作与建立

分裂与提升

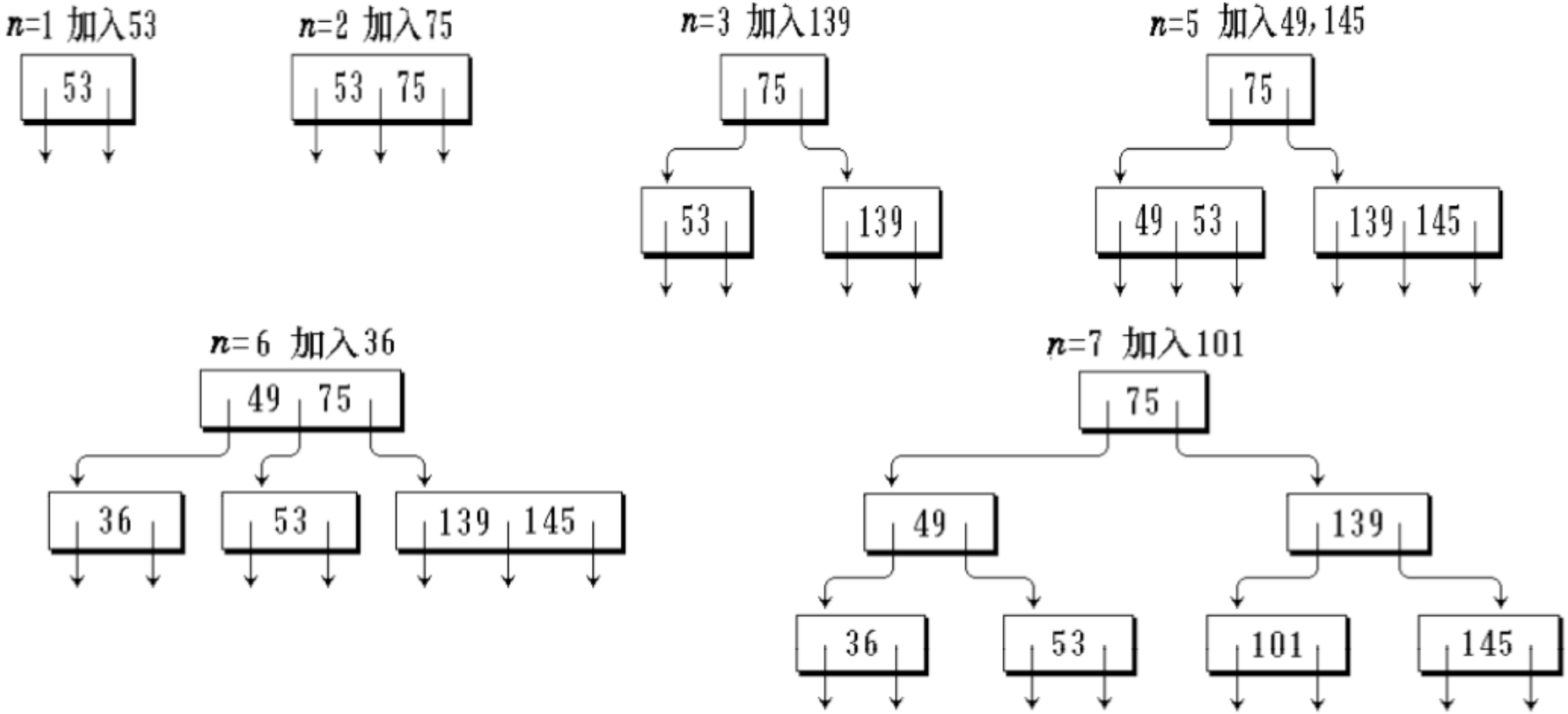
- 这时必须把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：
- 结点 p：

  $(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \ldots\ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
- 结点 q：

  $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \ldots\ldots, K_m, A_m)$
- 位于中间的关键字 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到这两个结点的双亲结点中去。
- 在插入该二元组之前，先将结点 p 和 q 写到磁盘上。
- 例：从空树开始逐个加入关键字建立3阶B-树

10.2.2 B-树和B+树

□ B-树的插入操作与建立(建立m=3阶的B树为例)

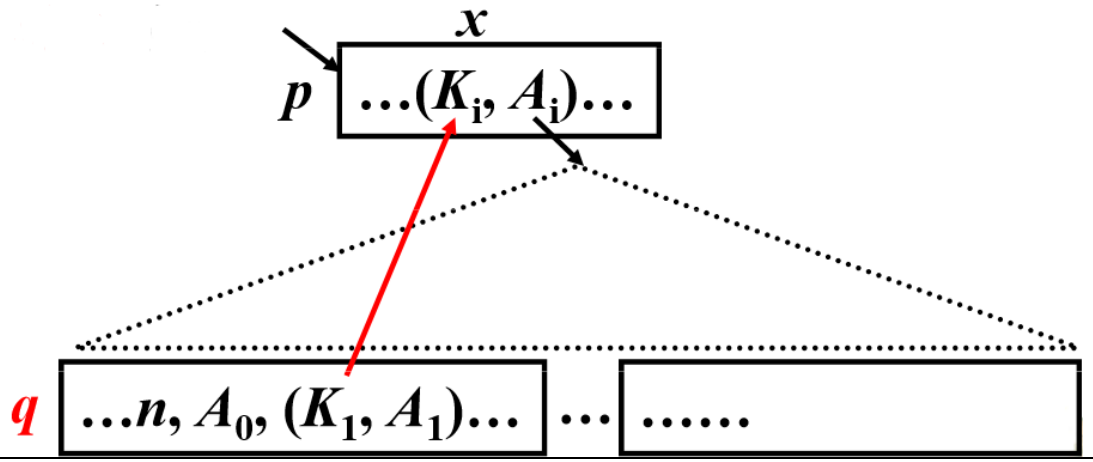


■ 在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键字所在终端结点到根的路径上的所有结点都要分裂。

10.2.2 B-树和B+树

□ B-树-删除-在B-树上删除一个关键字x时，

- 首先，要找到该关键字x所在的结点p，从p中删去这个关键字。若该结点不是终端结点，且$x=K_i$（$1 \le i \le n$），则在删去$x=K_i$之后，以该结点p中A_i所指示子树中的最小关键字（如在结点q）或者A_{i-1}所指示子树中的最大关键字来代替被删关键字K_i;
- 然后在q所在的终端结点中删除相应关键字。把删除操作转换为终端结点上的删除操作。



10.2.2 B-树和B+树

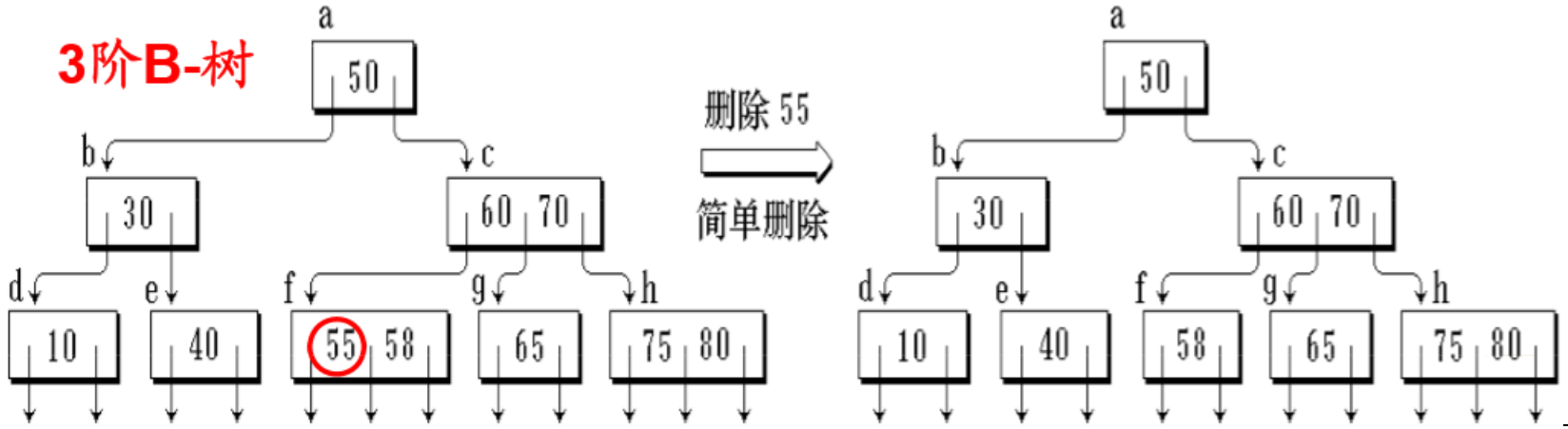
□ B-树-删除-在终端结点上的删除分4 种情况：

- 被删关键字所在终端结点p同时又是根结点，若删除后该结点中至少有一个关键字，则直接删去该关键字并将修改后的结点写回磁盘。否则，删除以后，B-树为空。

对于以下3种情况，p都不是根结点：

B-树主要用作文件索引

- 被删关键字所在终端结点p不是根结点，且删除前该结点中关键字个数n≥⌈m/2⌉，则直接删去该关键字并将修改后的结点写回磁盘，删除结束。



10.2.2 B-树和B+树

□ B-树-删除-在终端结点上的删除分4 种情况：

- 被删关键字x所在的终端结点p删除前关键字个数n = ⌈m/2⌉ -1，若此时其右兄弟(或左兄弟)q的关键字个数n≥ ⌈m/2⌉，则可按以下步骤调整结点p、右兄弟(或左兄弟)q 和其双亲结点r，以达到新的平衡。

兄弟够借，向兄弟借

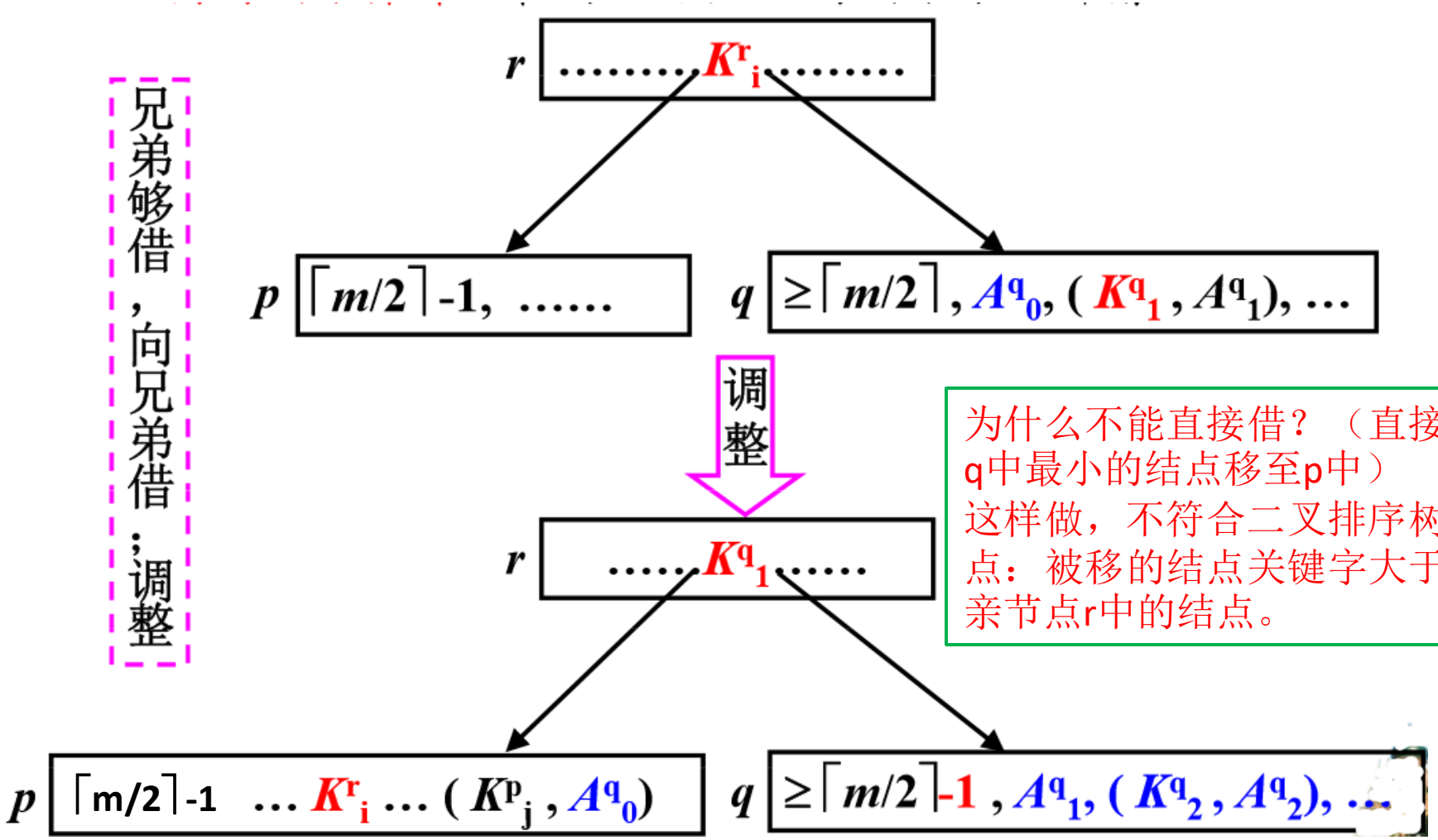
  - 将双亲结点r中大于(或小于)被删的关键字的最小(最大)关键字 $K_i(1 \le i \le n)$下移至结点p；

即紧靠被删关键字值的关键字

  - 将右兄弟 (或左兄弟) 结点中的最小 (或最大)关键字上移到双亲结点r的 K_i位置；
  - 将右兄弟 (或左兄弟) 结点中的最左 (或最右) 子树指针平移到被删关键字所在结点 p 中最后 (或最前) 子树指针位置；
  - 在右兄弟 (或左兄弟) 结点q中，将被移走的关键字和指针位置用剩余的关键字和指针填补、调整。再将结点q中的关键字个数减1。

10.2.2 B-树和B+树

□ B-树-删除-在终端结点上的删除分4 种情况：



为什么不能直接借？（直接将q中最小的结点移至p中）
这样做，不符合二叉排序树特点：被移的结点关键字大于双亲节点r中的结点。

10.2.2 B-树和B+树

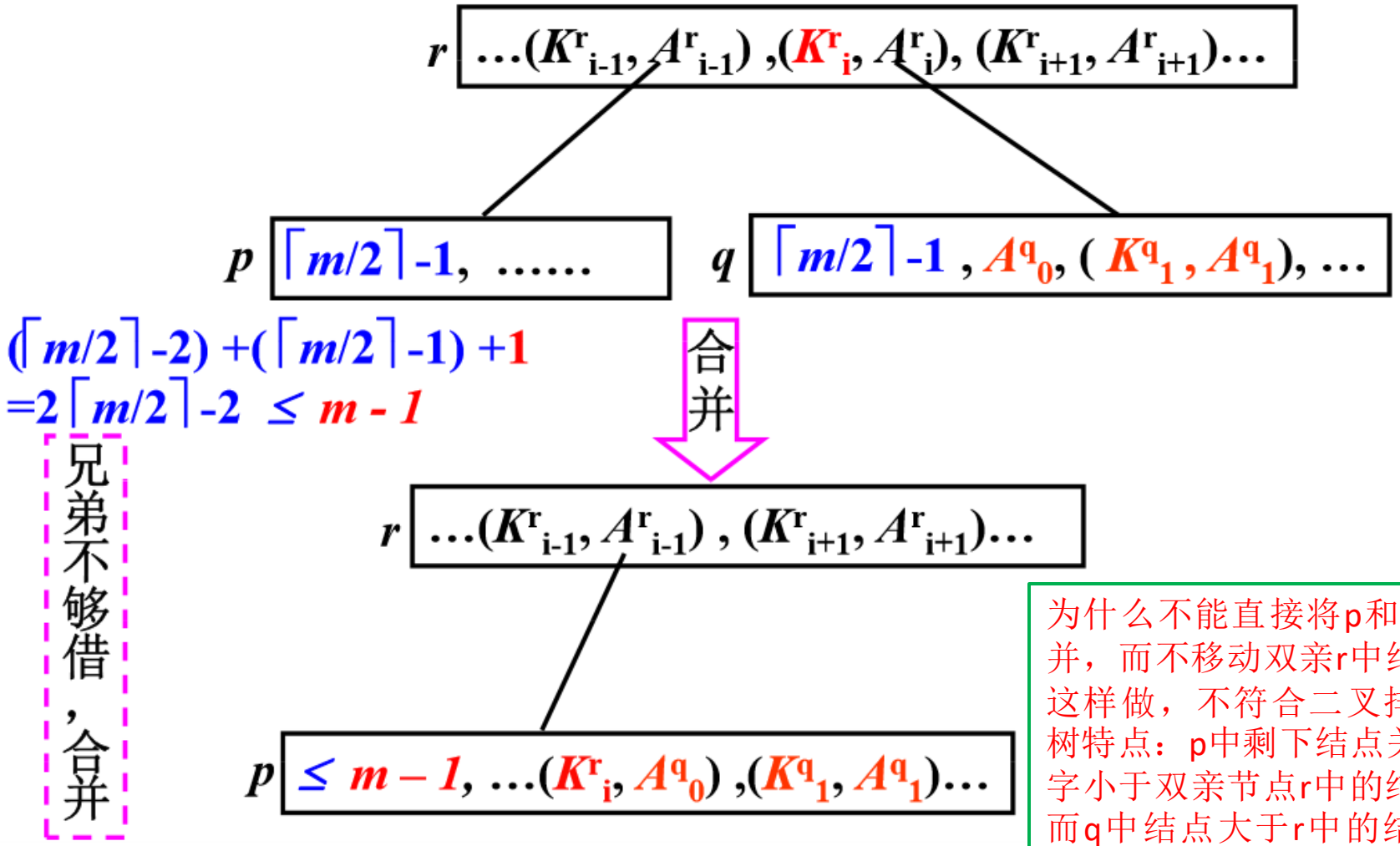
□ B-树-删除-在终端结点上的删除分4 种情况：

- 被删关键字x所在的终端结点p删除前关键字个数n =⌈m/2⌉ -1，若此时其右兄弟(或左兄弟)q的关键字个数n=⌈m/2⌉ -1，则按以下步骤合并这两个结点。兄弟不够借，合并

  - 将双亲结点 r 中相应关键字 K_i 下移到选定保留的结点中。若要合并 r 中的子树指针 A_{i-1} 与 A_i 所指的结点，且保留 A_{i-1} 所指结点，则把 r 中的关键字 K_i 下移到 A_{i-1} 所指的结点中。
  - 把 r 中子树指针 A_i 所指结点中的全部指针和关键字都拷贝到 A_{i-1} 所指结点的后面。删去 A_i 所指的结点。
  - 在结点 r 中用后面剩余的关键字和指针填补关键字 K_i 和指针 A_i。
  - 修改结点 r 和选定保留结点的关键字个数。

10.2.2 B-树和B+树

□ B-树-删除-在终端结点上的删除分4 种情况：

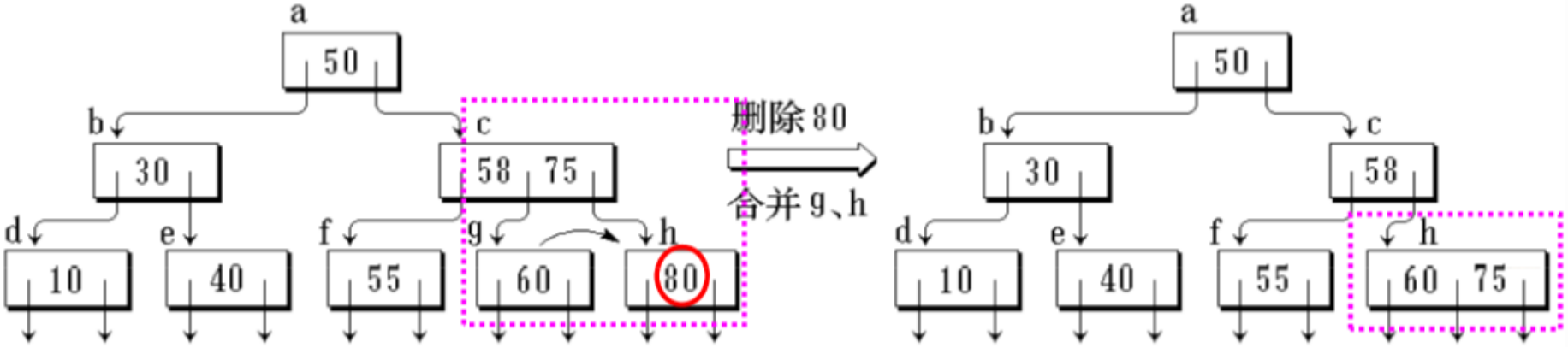


为什么不能直接将p和q合并，而不移动双亲r中结点？这样做，不符合二叉排序树特点：p中剩下结点关键字小于双亲节点r中的结点，而q中结点大于r中的结点。

10.2.2 B-树和B+树

B-树-删除-在终端结点上的删除分4 种情况：

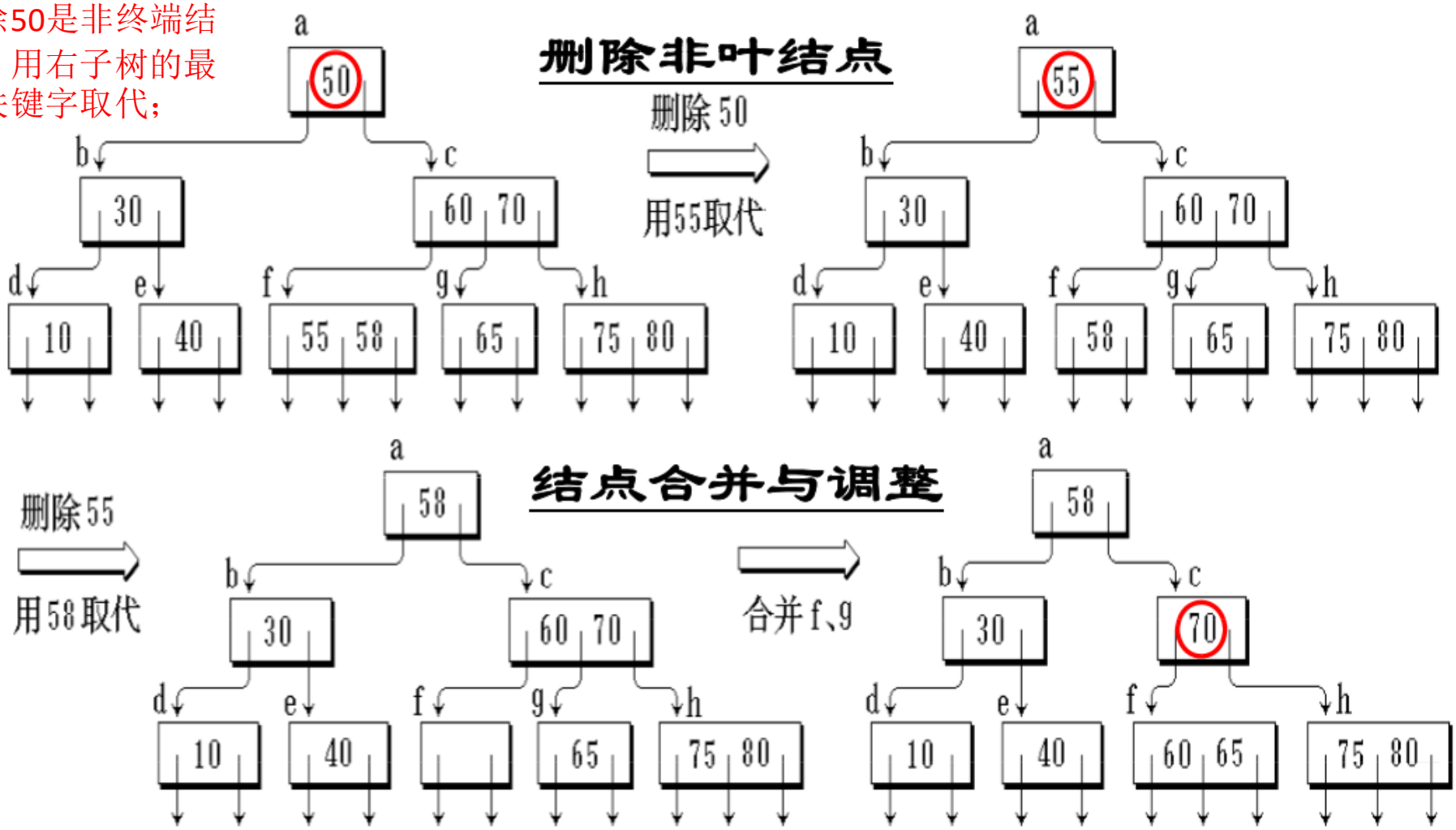
- 在合并结点的过程中，双亲结点中的关键字个数减少了。
- 若双亲结点是根结点且结点关键字个数减到 0，则该双亲结点应从树上删去，合并后保留的结点成为新的根结点；否则将双亲结点与合并后保留的结点都写回磁盘，删除处理结束。
- 若双亲结点不是根结点，且关键字个数减到⌈m/2⌉-2，则又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。



10.2.2 B-树和B+树

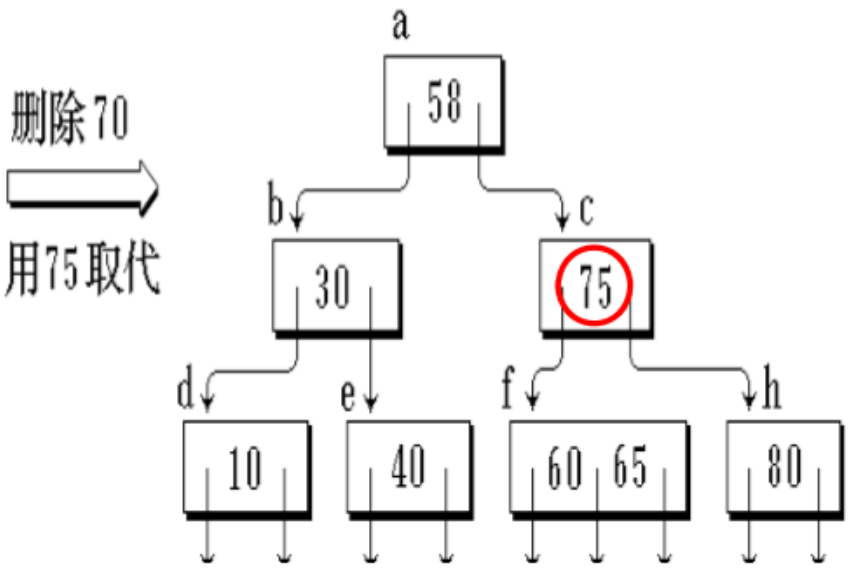
□ B-树-删除-在终端结点上的删除分4 种情况：

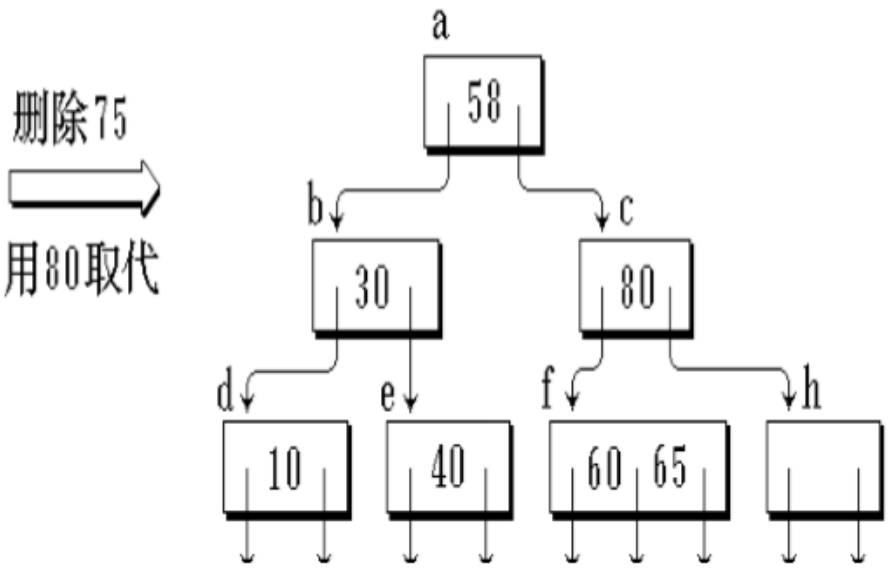
删除50是非终端结点，用右子树的最小关键字取代；

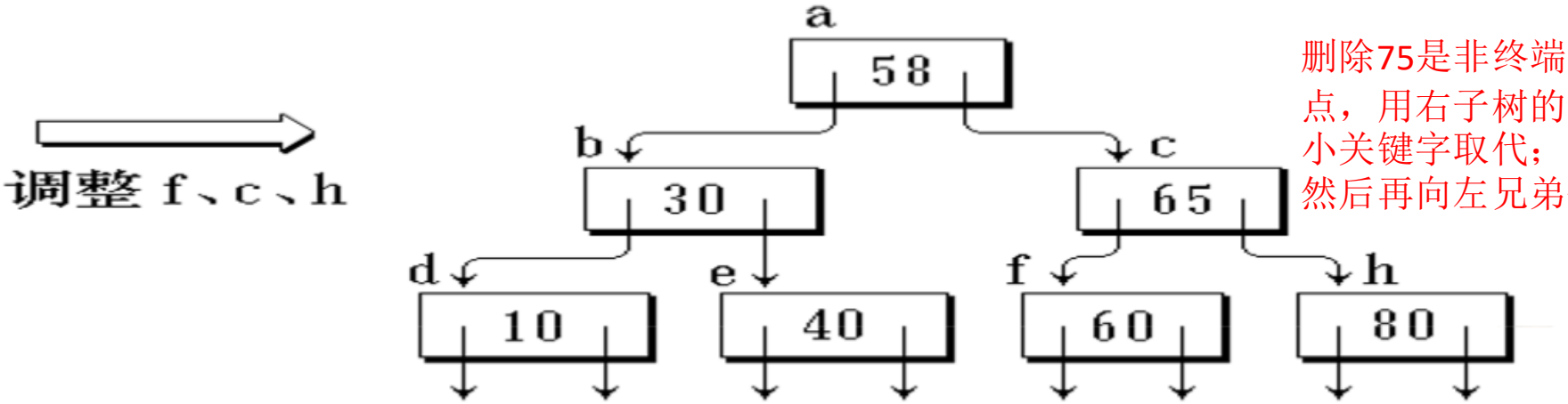


10.2.2 B-树和B+树

□ B-树-删除-在终端结点上的删除分4 种情况：







删除75是非终端结点，用右子树的最小关键字取代；然后再向左兄弟借

10.2.2 B-树和B+树

□ B-树特点分析

在B-树中进行查找时，其查找时间主要花费在搜索结点（访问外存）上，即主要取决于B-树的深度。

问：深度为H的B-树中，至少含有多少个结点？

10.2.2 B-

- 根结点至少含2棵子树；
- 非叶结点至少含有⌈m/2⌉棵子树

□ B-树的特点分析

推导每一层所含最少结点数：

第 1 层	1 个
第 2 层	2 个
第 3 层	2×⌈m/2⌉ 个
第 4 层	2×(⌈m/2⌉)^2 个
… …	
第 H+1 层	2×(⌈m/2⌉)$^{H-1}$ 个

10.2.2 B-树和B+树

□ B-树查找性能的分析

问：含 N 个关键字的 *m* 阶 B-树可能达到的最大深度 H 为多少？

10.2.2 B-树和B+树

□ B-树的特点分析

假设 m 阶 B-树的深度为 $H+1$，由于第 H+1 层为叶子结点，而当前树中含有 N 个关键字，则叶子结点必为 N+1 个，

由此可推得下列结果：

重要性质：叶子结点个数=树中关键字个数+1

$$N+1 \geq 2(\lceil m/2 \rceil)^{H-1}$$

$$H-1 \leq \log_{\lceil m/2 \rceil}((N+1)/2)$$

$$H \leq \log_{\lceil m/2 \rceil}((N+1)/2)+1$$

10.2.2 B-树和B+树

□ B-树的特点分析

结论：

在含 N 个关键字的 B-树上进行一次查找，需访问的结点个数不超过

$$\log_{\lceil m/2 \rceil}((N+1)/2)+1$$

10.2.2 B-树和B+树

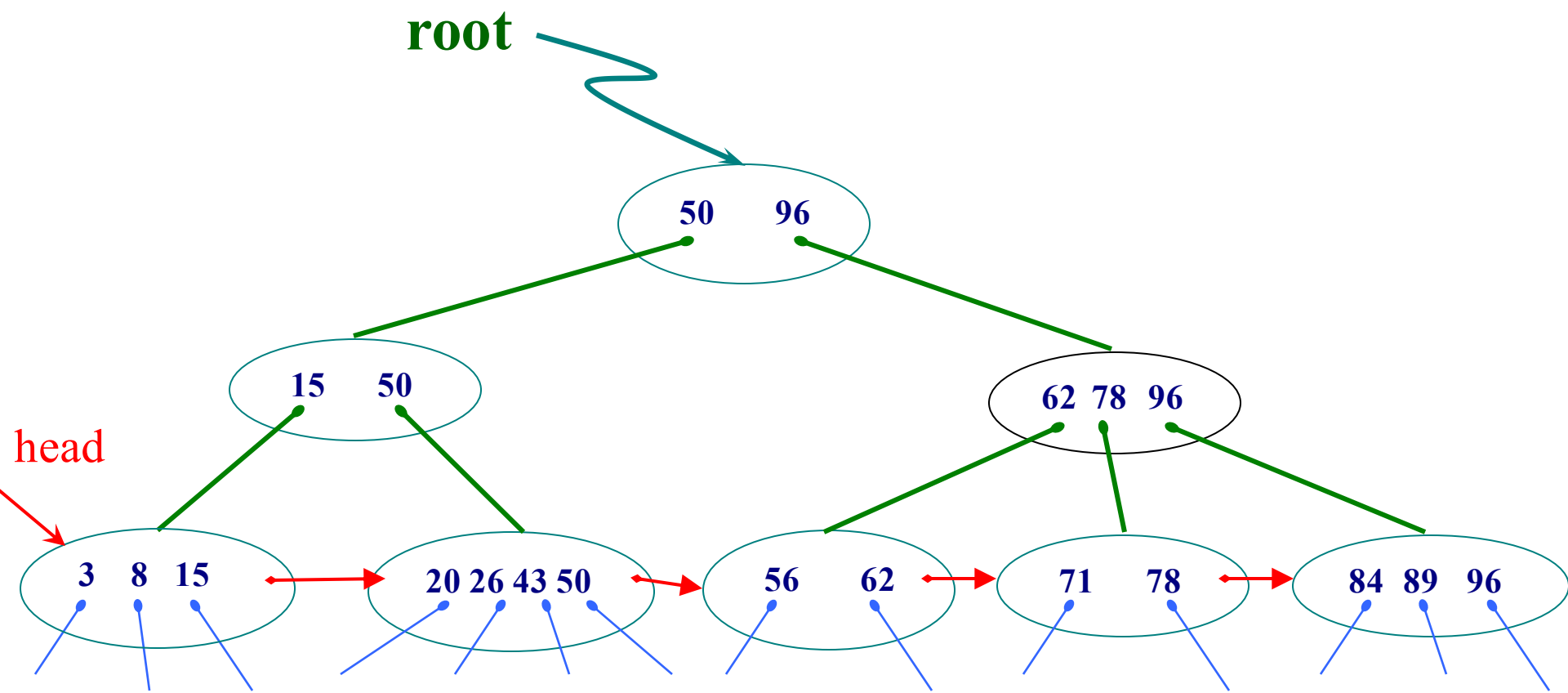
□ B+树的定义

◆B+树可以看作是B-树的一种变形;

◆在实现文件索引结构方面比B-树使用得普遍。

10.2.2 B-树和B+树

□ B+树的特点



10.2.2 B-树和B+树

□ B+树的特点

◆所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键码记录的指针；

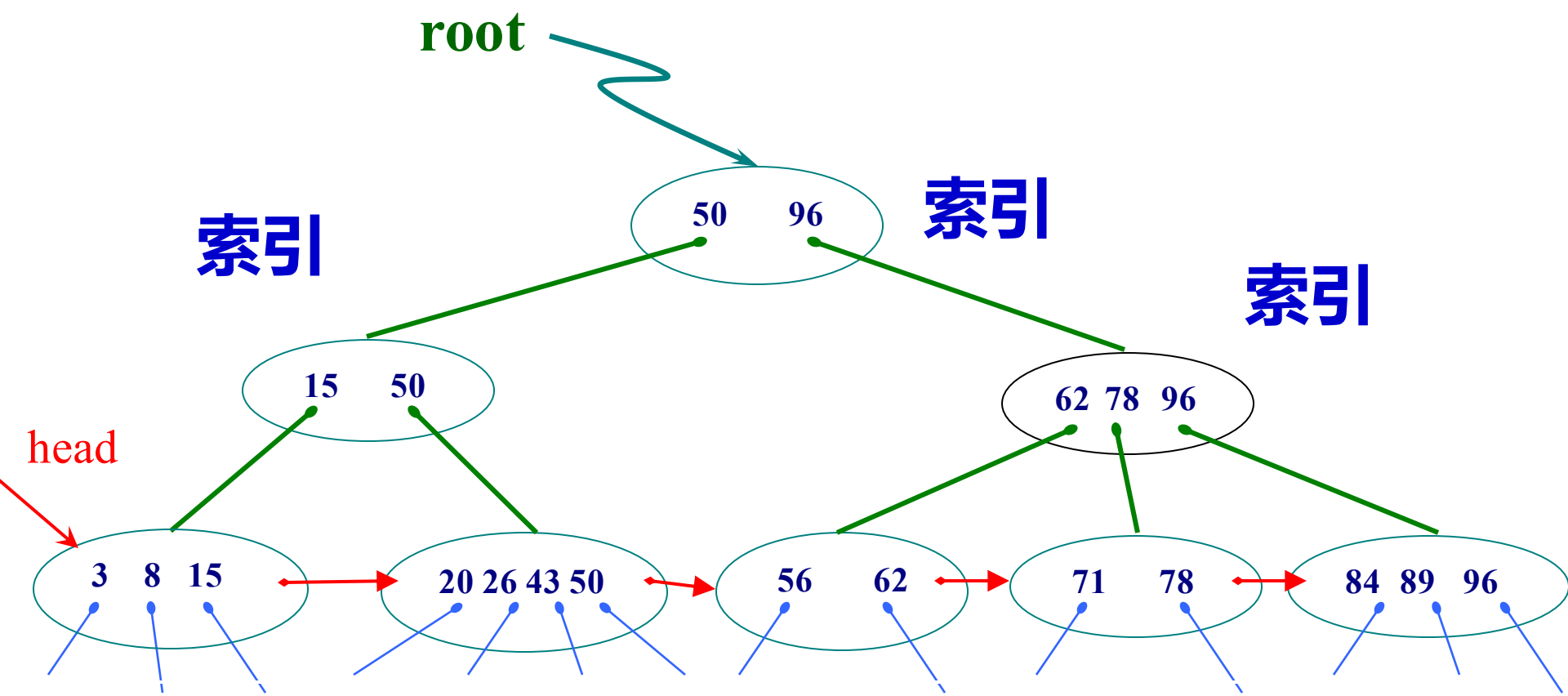
◆所有的非终端结点可以看成是索引；

◆结点中仅含有其子树根结点中最大(或最小)关键码；

◆通常在B+树上有两个头指针，一个指向根结点，另一个指向关键码最小的叶子结点。

10.2.2 B-树和B+树

□ B+树的特点



◆所有叶子结点都处在同一层次上构成一个有序链表;

◆头指针指向最小关键字的结点

10.2.2 B-树和B+树

□ B+树的查找过程

与B-树类似。

◆只是在搜索过程中，若非终端结点上的关键码等于给定值，搜索并不停止，而是继续沿右指针向下，一直查到叶结点上的这个关键码。

◆在B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。

10.2.2 B-树和B+树

□ B+树的查找特点

◆在 B^+ 树上，既可以进行缩小范围的查找，也可以进行顺序查找；

◆ 在进行缩小范围的查找时，不管成功与否，都必须查到叶子结点才能结束；

◆ 若在结点内查找时，给定值≤Ki， 则应继续在 Ai 所指子树中进行查找。

10.2.2 B-树和B+树

□ B+树的插入和删除

插入和删除的操作

类似于B-树:必要时，也需要进行结点的“分裂”或“归并”。

10.2.2 B-树和B+树

查找树

- 二叉排序树（二叉查找树）Binary Search Tree
  - 平衡二叉排序树 ---二叉平衡查找树
    Balanced Binary Tree
    - B - 树
    - B^+树

多路平衡查找树

第十章 查找

10.1 静态查找表

10.2 动态查找表

10.3 哈希表（散列表）-动态查找表

10.3 哈希表

10.3.1 什么是哈希表

10.3.2 哈希函数的构造方法

10.3.3 处理冲突的方法

10.3.4 哈希表的查找及其分析

10.3 哈希表

10.3.1 什么是哈希表

10.3.2 哈希函数的构造方法

10.3.3 处理冲突的方法

10.3.4 哈希表的查找及其分析

10.3.1 什么是哈希表

哈希表的引入

- 查找操作要完成什么任务？
  - 对于待查值k，通过比较，确定k在存储结构中的位置
- 基于关键字比较的查找的时间性能如何？
  - 其时间性能为$O(\log_2 n)$~$O(n)$。
  - 实际上用判定树可以证明，基于关键字比较的查找的平均和最坏情况下的比较次数的下界是$\log_2 n+O(1)$，即$\Omega(\log_2 n)$
  - 要想突破此下界，就不能仅依赖于基于比较来进行查找
- 能否不用比较，通过关键字的取值直接确定存储位置？
  - 在关键字值和存储位置之间建立一个确定的对应关系

10.3.1 什么是哈希表

□ 哈希表的引入

◆对于频繁使用的查找表，希望 ASL = 0。

◆办法：预先知道所查关键字在表中的位置。

要求：记录在表中位置和其关键字之间存在一种确定的关系。

10.3.1 什么是哈希表

◘ 哈希表的引入

例：全国30个地区的人口统计，每个地区为一个记录，内容如下：

编号	地区名	总人口	汉族	回族	...

$H_1(key)$：取关键字第一个字母在字母表中的序号

$H_2(key)$:求第一和最后字母在字母表中序号之和，然后取30的余数

Key	BEIJING (北京)	TIANJIN (天津)	HEBEI (河北)	SHANXI (山西)	SHANGHAI (上海)	SHANGDONG (山东)	HENAN (河南)	SICHUAN (四川)
$H_1(key)$	02	20	08	19	19	19	08	19
			Harbin	shandong	发生冲突			
$H_2(key)$	09	04	17	28	28	26	22	03

10.3.1 什么是哈希表

◘ 哈希表的有关概念

◆哈希函数：又叫散列函数，在关键字与记录在表中的存储位置之间建立一个函数关系，以 H(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 H(key) 为哈希函数。

◆哈希地址：由哈希函数得到的存储位置称为哈希地址。

◆装填因子：设散列表空间大小为n，填入表中的结点数为m,则称α=m/n为散列表的装填因子。

◆冲突（Collision）与同义词： 若H(k1)=H(k2),则称为冲突，发生冲突的两个关键字k1和k2称为同义词。

10.3.1 什么是哈希表

□ 哈希函数的选择

很难找到一个不产生冲突的哈希函数

◆ 在构造这种特殊的“查找表” 时，选择一个尽可能少产生冲突的哈希函数；

◆需要找到一种“处理冲突” 的方法。

10.3.1 什么是哈希表

□ 哈希表的定义

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“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

10.3.1 什么是哈希表

- 哈希（散列）技术仅仅是一种查找技术吗？
  - 哈希既是一种查找技术，也是一种存储技术。
- 哈希是一种完整的存储结构吗？
  - 哈希只是通过记录的关键字的值定位该记录，没有表达记 录之间的逻辑关系，所以哈希主要是面向查找的存储结构。
- 哈希技术适用于何种场合？
  - 通常用于实际出现的关键字的数目远小于关键字所有可能 取值的数量。
- 哈希技术适合于哪种类型的查找？
  - 不适用于允许多个记录有同样关键字值的情况。
  - 也不适用于范围查找，如在哈希表中，找最大或最小关键 值的记录，也不可能找到在某一范围内的记录。

10.3.1 什么是哈希表

- 哈希技术需解决的关键问题：
  - 哈希函数的构造
    - 如何设计一个简单、均匀、存储利用率高的哈希函数
  - 冲突的处理
    - 如何采取合适的处理冲突方法来解决冲突。
  - 哈希结构上的查找、插入和删除
- 哈希函数的构造
  - 哈希函数的构造的原则：
    - 计算简单：哈希函数不应该有很大的计算量，否则会降低 查找效率。
    - 分布均匀：哈希函数值即散列地址，要尽量均匀分布在地 址空间，这样才能保证存储空间的有效利用并减少冲突。

10.3.1 什么是哈希表

口构造哈希函数的方法

对数字的关键字可有下列构造方法：

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

若是非数字关键字，则需先对其进行数字化处理。

10.3 哈希表

10.3.1 什么是哈希表

10.3.2 哈希函数的构造方法

10.3.3 处理冲突的方法

10.3.4 哈希表的查找及其分析

10.3.2 哈希函数的构造方法

□ 直接定址法

取关键字或关键字的某个线性函数值为哈希地址。

即：**H(key)=key或H(key)=a*key+b**

其中a、b为常数。又称H(key)为自身函数。

优点：没有冲突；

缺点：若关键字集合很大，浪费存储空间严重。

此法仅适合于：

地址集合的大小＝＝关键字集合的大小

10.3.2 哈希函数的构造方法

□ 直接定址法

例一：Hash(*key*) = *key*

地址	01	02	03	04	05	…	25	26	…	100
年龄	1	2	3	4	5	…	25	26	…	100

例二：Hash(*key*) = *key* + (-1949) + 1

地址	01	02	03	…
年份	1949	1950	1951	…

10.3.2 哈希函数的构造方法

□数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 $(u_1, u_2, \ldots, u_s)$，分析关键字集合的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

举例：可取中间4位中的两位为散列地址

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
①	②	③	④	⑤	⑥	⑦	⑧

10.3.2 哈希函数的构造方法

□数字分析法

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。

10.3.2 哈希函数的构造方法

□平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值” 的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

10.3.2 哈希函数的构造方法

□平方取中法

取 key ^{2} 的中间的几位数作为散列地址

此方法适合于：

关键字中的每一位都有某些数字重复出现频度很高的现象。

记录	*key*	*key*^{2}	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

平方取中法

10.3.2 哈希函数的构造方法

□平方取中法

关键字	(关键字)^2	地址
0100	0010000	100
0110	0012100	121
1010	1020100	201
1001	1002001	020
0111	0012321	123

10.3.2 哈希函数的构造方法

□折叠法

将关键字分割成若干部分，然后取它们的叠加和为哈希地址。

两种叠加处理的方法：移位叠加和间界叠加。

此方法适合于：

关键字的数字位数特别多。

10.3.2 哈希函数的构造方法

□折叠法（移位和间界）

将关键字分割成位数相同的几段，最后一位可以不同。段的长度取决于散列表的地址位数，然后将各段的叠加和（舍去进位）作为散列地址

例如：图书编号: 0-442-20586-4

```
   5864
   4220
+    04
-------
  10088
```

左移位叠加：将分割后的每一部分的最低位对齐，然后相加

Hash(key) = 0088

```
   5864
   0224
+    04
-------
   6092
```

间界叠加：从一端向另一端沿分割界来回折叠，然后对齐相加

Hash(key) = 6092

10.3.2 哈希函数的构造方法

□除留余数法(质数除余法)

设桶数B，取质数 m ≤ B

Hash (key) = key % m

举例：将关键字序列(09，31，26，19，01，13，02，11，27，16，05，21)**依次填入长度为**n = 12**的表中。设映象函数为**i = INT(k/3) + 1。

表序号	1	2	3	4	5	6	7	8	9	10	11	12
按i＝INT(k/3)＋1 填入的关键字	01	05		09	13	16	19	21	26	27	31	
	02			11								

INT是将一个要取整的实数（可以为数学表达式）向下取整为最接近的整数。

10.3.2 哈希函数的构造方法

除随机数法

设定哈希函数为:

H(key) = Random(key)

其中，Random 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

10.3.2 哈希函数的构造方法

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。

10.3.2 哈希函数的构造方法

构造Hash函数应注意以下几个问题：

- **计算Hash函数所需时间**
- **关键字的长度**
- **散列表的大小**
- **关键字的分布情况**
- **记录的查找频率**

表序号	1	2	3	4	5	6	7	8	9	10	11	12
按i＝INT(k/3)＋1	01	05		09	13	16	19	21	26	27	31	
填入的关键字	02			11								

如何解决抵制冲突？

10.3.2 哈希函数的构造方法

- 处理冲突的方法

“处理冲突” 的实际含义是：

为产生冲突的地址寻找下一个哈希地址。

1. 开放定址法

2. 链地址法

10.3.3 处理冲突的方法

□ 开放定址法

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

对增量 d_i 有三种取法：

H(key) 为哈希函数，m为表长，d_i 为增量序列：

(1) $d_i = 1,2,3,\ldots,m-1$--- 线性探测再散列

(2) $d_i = 1^2,-1^2,2^2,-2^2,\ldots,\pm k^2 \ (k \le m/2)$

---二次探测再散列 或平方探测再散列

(3) d_i 为伪随机序列 ---随机探测再散列

(4)双散列函数探测-发生冲突是采用两个散列函数H1(key)和H2(key)

图示

10.3.3 处理冲突的方法

- 开放定址法

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 (表长=11)

若采用线性探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

10.3.3 处理冲突的方法

开放定址法

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 (表长=11)

若采用二次探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

$d_i = 1^2,-1^2,2^2,-2^2,\ldots,\pm$

对于68，增量为2^2

10.3.3 处理冲突的方法

- 链地址法（又称拉链法）

将所有哈希地址相同的记录
都链接在同一链表中。

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

哈希函数为 H(key)=key MOD 7

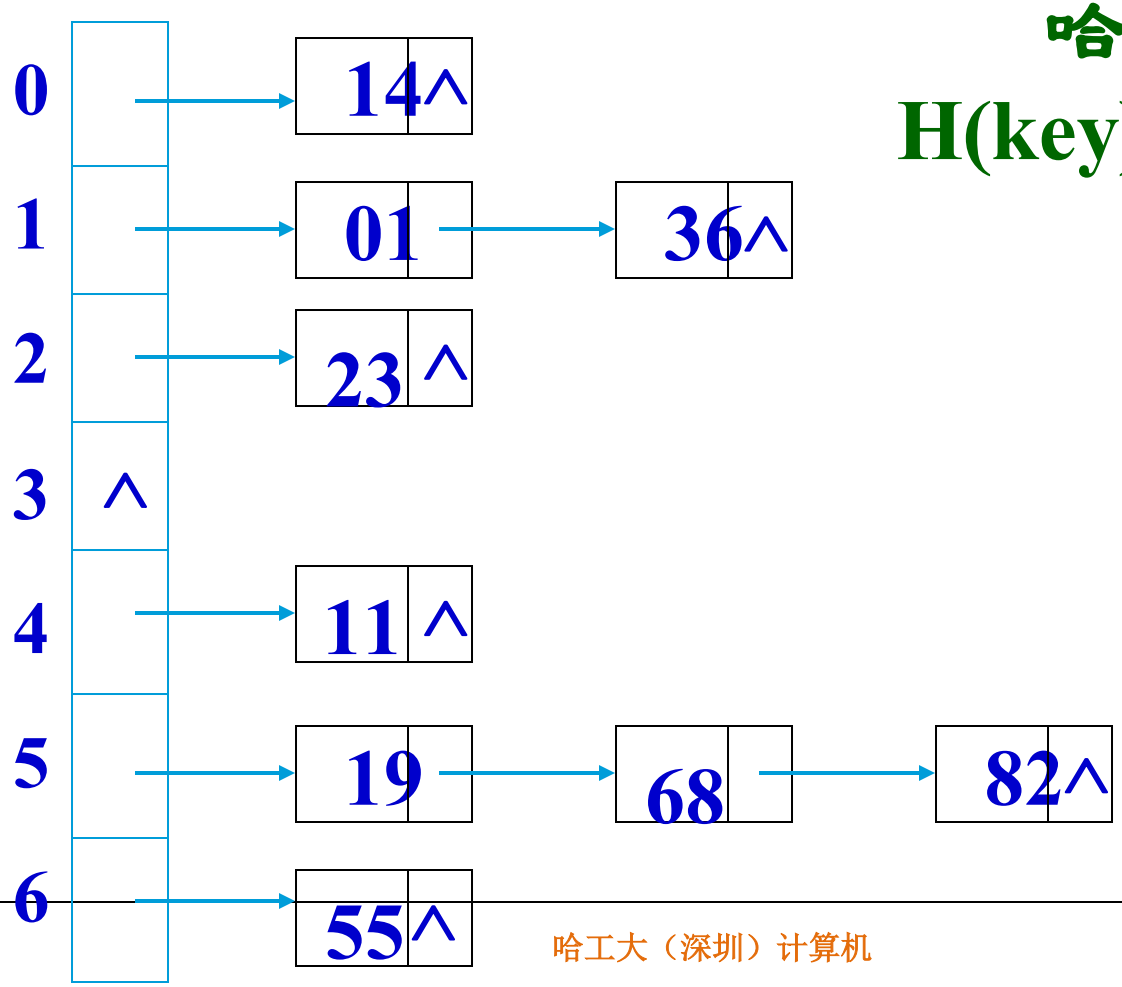
10.3.3 处理冲突的方法

链地址法（又称拉链法）

关键字集合 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }

哈希函数为

H(key)=key MOD 7

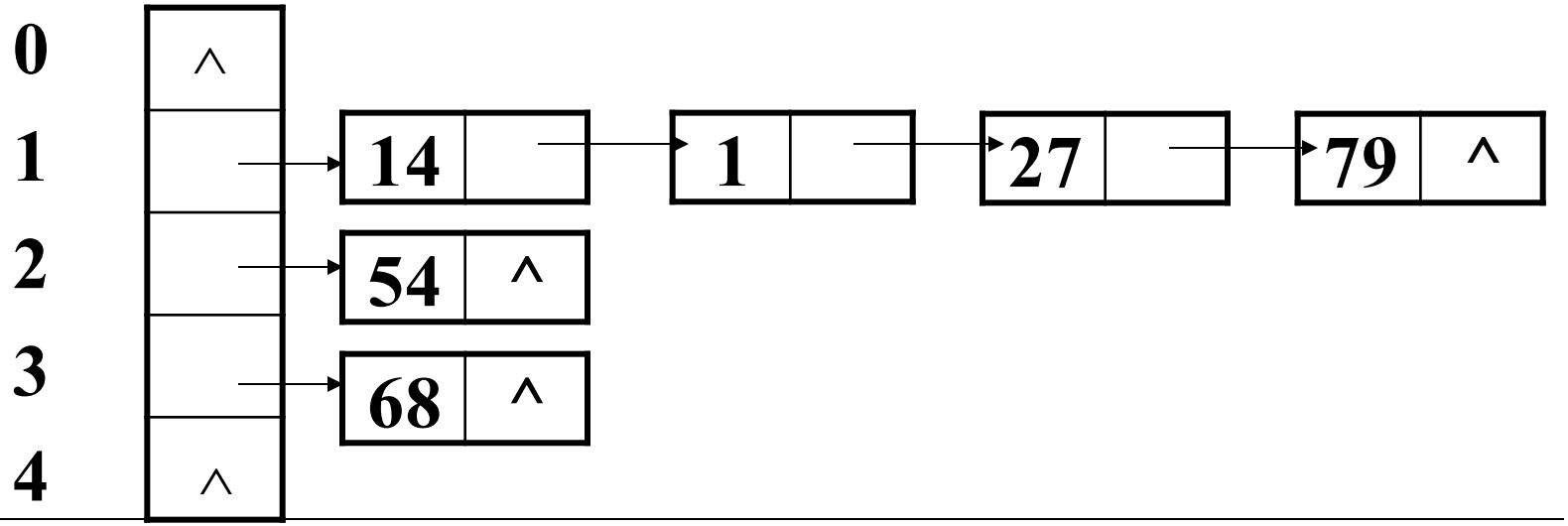


10.3.3 处理冲突的方法

拉链法的查找算法

(1) 根据关键字K找到关键字为K的结点所在单链表的首地址；

(2)在所找到的单链表上进行顺序查找，若找到返回地址值，否则返回空值。



10.3.3 处理冲突的方法

- 拉链法的插入算法：

(1) 根据关键字K找到关键字为K的结点所应该存在的单链表的首地址；

(2)在单链表中查找结点为K的关键字，若找到，则无须插入；

(3)若没找到，利用头插法或尾插法插入单链表中。

10.3.4 哈希表的查找和分析

开放定址哈希表存储结构

```
int  hashsize[] = { 997, ... };  //哈希表容量递增表，一个适合的素数序列
typedef struct {
  ElemType  *elem;    //数据元素存储基址，动态分配数组
  int  count;              // 当前数据元素个数
  int  sizeindex;
                 // hashsize[sizeindex]为当前容量
} HashTable;
#define  SUCCESS  1
#define  UNSUCCESS  0
#define  DUPLICATE  -1
```

10.3.4 哈希表的查找和分析

□ 开放定址哈希表的查找算法

```
Status SearchHash (HashTable H, KeyType K,
                                    int &p, int &c)
{  p = Hash(K);        // 求得哈希地址
   while ( H.elem[p].key != NULLKEY &&  //该地址含有记录
                 !EQ(K, H.elem[p].key)）//且关键字不等
      collision(p, ++c);      // 求得下一探查地址 p
   if (EQ(K, H.elem[p].key)) return SUCCESS;
                 // 查找成功，返回待查数据元素位置 p
   else return UNSUCCESS;   // 查找不成功
} // SearchHash
```

10.3.4 哈希表的查找和分析

◘ 哈希表的平均查找长度

决定哈希表查找的ASL的因素：

(1)选用的哈希函数；

(2)选用的处理冲突的方法；

(3)哈希表饱和的程度--装载因子

α=n/m 值的大小（n—记录数，m—表的长度）

装载因子α 越小，发生冲突的可能性就越小。

10.3.4 哈希表的查找和分析

◘ 哈希表的平均查找长度

一般情况下，可以认为选用的哈希函数是“均匀”的，也就是在讨论ASL时，认为各个位置被存数据的概率是相等的。

10.3.4 哈希表的查找和分析

查找成功的平均查找长度

(1) 线性探测再散列 $S_{nl} \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$

(2) 随机探测再散列 $S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$

(3) 链地址法 $S_{nc} \approx 1+\frac{\alpha}{2}$

哈希表的平均查找长度是 α 的函数

用哈希表构造查找表时，选择适当的装填因子 α，使得平均查找长度限定在某个范围内。

10.3.4 哈希表的查找和分析

□ 散列法与其它方法的比较

(1) 不同的散列函数构成的散列表平均查找长度是不同的。

(2) 由同一个散列函数、不同的解决冲突方法构造的散列表，其平均查找长度是不相同的。

(3)散列法是根据关键字直接求出地址的查找方法，其查找的期望时间为O(1) 。

(4) 其它查找方法有共同特征为：均是建立在比较关键字的基础上。

本章小结

✓ 熟练掌握：

(1) 熟练掌握顺序查找、二分查找、二叉排序树基本思想、算法和应用。。

(2) 熟练掌握散列表构造及应用。

(3)熟练掌握二叉排序树的生成、插入和删除的注意算法；平衡二叉树的、B-树和B+树的基本知识。