9.3动态查找表

DS/SS

抽象数据类型定义

ADT DynamicSearchTable {

数据对象：V={具有相同性质的数据元素,各个数据元素均含有类型相同，可唯一标识数据元素的关键字}

数据关系：R: 集合关系

基本操作：P:

InitDSTable(&DT);

DestroyDSTable(&DT);

SearchDSTable(DT,key);

InsertDSTable(&DT,e);

TraverseDSTable(DT,Visit())

}ADT DynamicSearchTable

9.3动态查找表（续）

二叉排序树的查找

二叉排序树查找=查找表+二叉树+BST性质

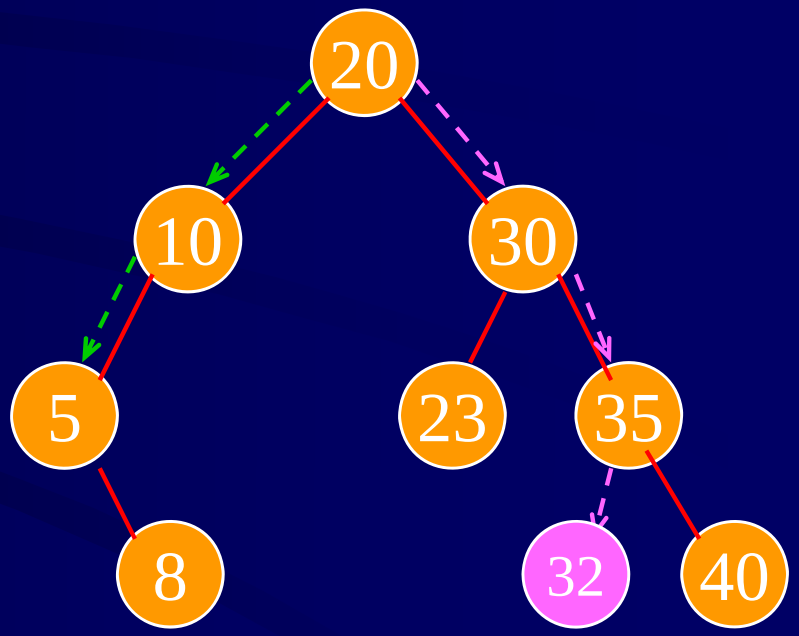
- 二叉排序树(Binary Sort Tree)：或者是一棵空树；或者具有BST性质：(1)若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；(2)若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；(3) 它的左、右子树也分别为二叉排序树(BST树)。BST树也称为二叉查找树。
- 平衡二叉树(Balanced Binary Tree或Height-Balanced Tree)：或者是一棵空树；或者具有AVL性质：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。平衡二叉树也称为AVL树。
- 平衡因子(Balance Factor):该结点的左子树的深度减去它的右子树的深度。

9.3动态查找表（续）

二叉排序树查找过程

设给定值key = 5,则查找路径为：

设给定值key = 32,则查找路径为：



BST树上的查找为动态查找，在查找失败时，要在失败的位置插入关键字为待查关键字的数据元素。

查找失败的位置在叶子处，插入位置也在叶子位置，当采用二叉链表存储BST树时，插入新元素时不需要移动其他元素，仅需修改指针即可。

9.3动态查找表（续）

二叉排序树查找过程

```
Status SearchBST(BiTree T, KeyType key,BiTree f,BiTree &p ){
//从根指针T所指二叉排序树中递归地查找某关键字等于key的数据元素。若
//查找成功，则指针p指向该数据元素结点，并返回TRUE，否则p指向查找路
//径上访问的最后一个结点并返回FALSE，f是T的双亲，初值为NULL
  if(!T)  { p =f;   return FALSE;}                          //查找不成功
  else  if  EQ(key,T->data.key)  {p =T; return TRUE; }      //查找成功
  else  if  LT(key,T->data.key)  SearchBST(T->lchild, key, T, p); //在左子树查找
  else  SearchBST(T->rchild, key, T, p);                    //在右子树中继续查找
} //SearchBST
```

9.3动态查找表（续）

二叉排序树建树过程

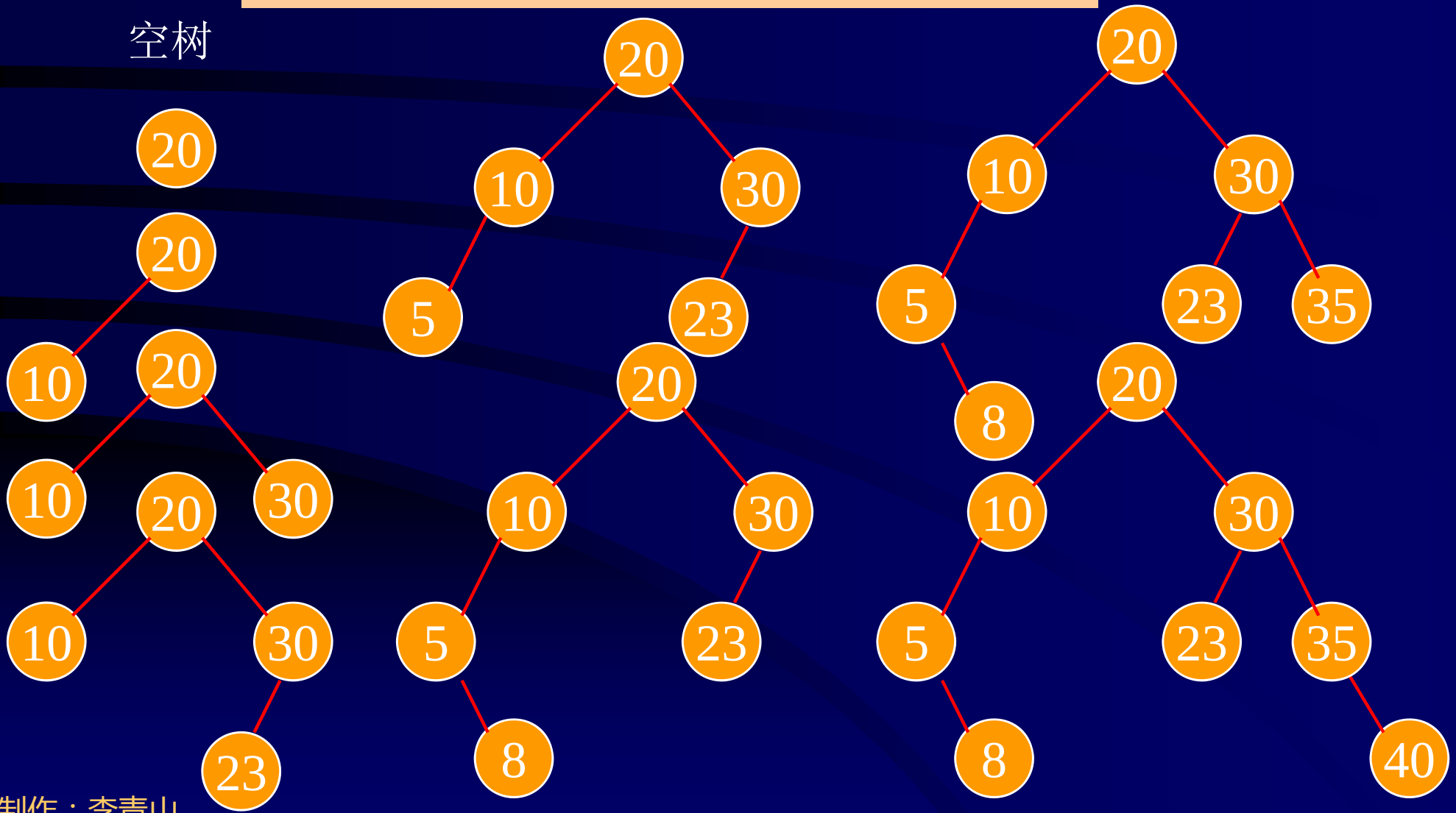
BST树上建树的过程，就是查找失败时元素不断插入的过程。

初始序列中元素的次序直接决定了BST树的结构。

9.3动态查找表（续）

初始序列：20 10 30 23 5 8 35 40

空树



9.3动态查找表（续）

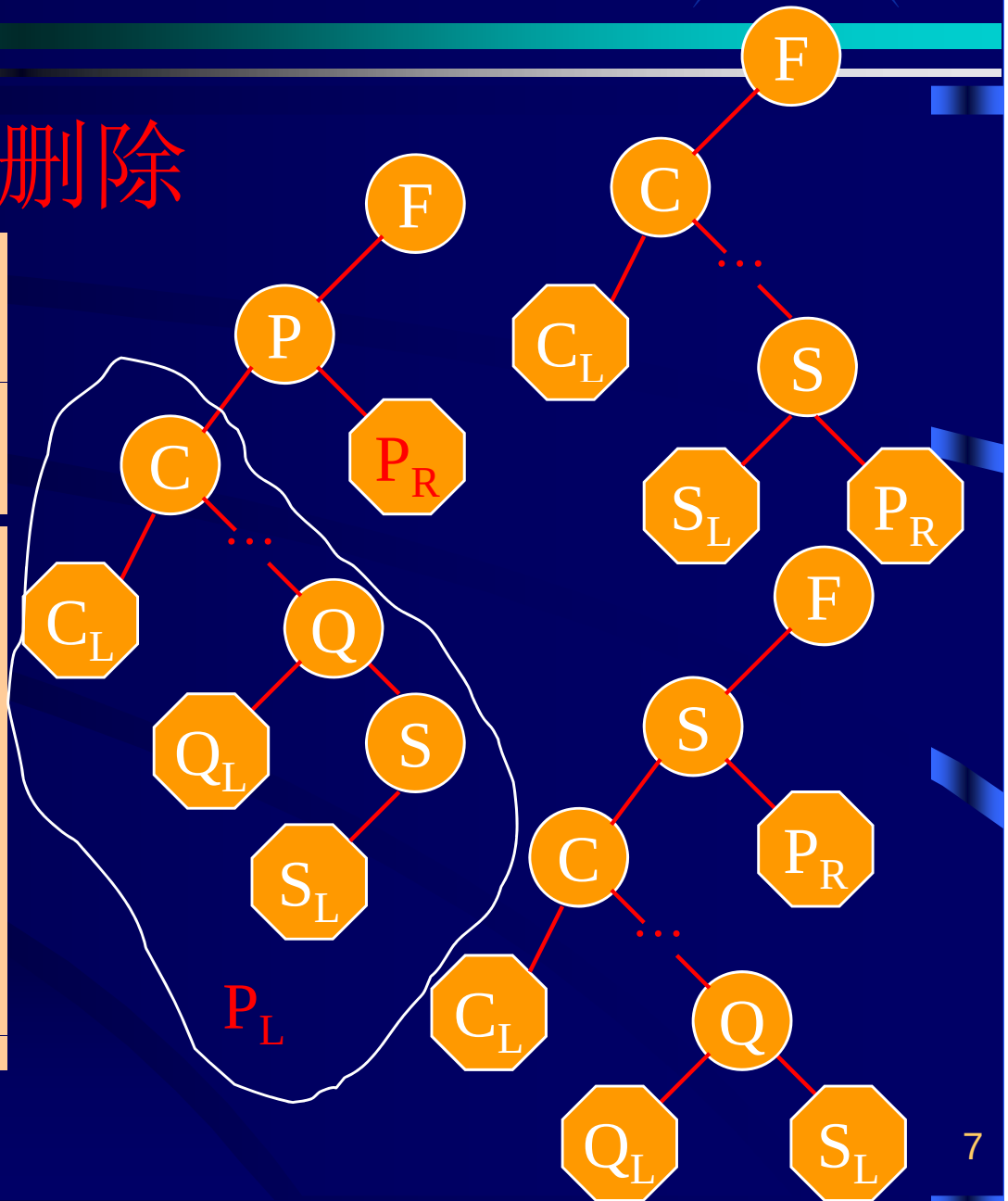
二叉排序树上结点删除

设BST树上被删除结点指针为p，其左、右子树分别表示为P_L和P_R，其双亲结点指针为f,不失一般性，设*p为*f的左孩子。则有：

1.若*p为叶子结点，直接删除;

2.若*p只有左子树P_L或者只有右子树P_R，删除*p后令P_L或P_R直接成为其双亲结点*f的左子树即可;

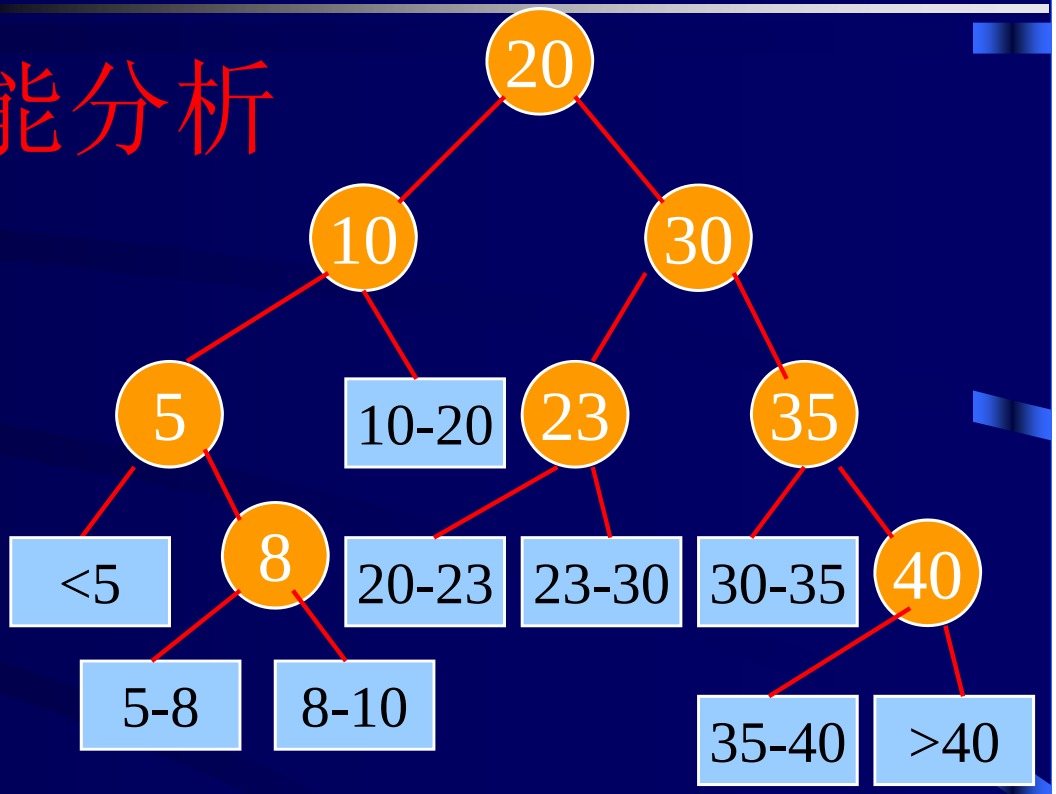
3.若*p左子树P_L和右子树P_R均不为空。将*p与它的中序序列的直接后继*s交换，然后*p满足一定没有左子树的特征，就转化为第1或2种情况。



9.3动态查找表（续）

二叉排序树查找性能分析

BST查找过程中，找到其中任何一个元素的过程就是走了一条根结点到与该元素相应的结点的路径，和给定值比较的关键字个数恰为该结点在BST树上的层次数。查找不成功的过程就是走了一条根结点到外部结点的路径，和给定值进行比较的关键字个数等于该路径上内部结点的个数。



BST树上查找成功和查找不成功的ASL与BST树的结构有关，而树的结构与初始序列构造有关。即含有n个结点的二叉排序树不唯一。效率最高的情况为BST为一棵平衡二叉树的时候。

$ASL_{bst成功}$ =(1*1+2*2+3*3+4*2)/8=11/4

$ASL_{bst不成功}$ =(1*2+3*4+4*4)/9=10/3

9.3动态查找表（续）

BST树平衡化过程

由初始序列建立BST树的过程中若因为插入一个结点而导致了当前树不平衡，则应该调整其平衡。这个过程即是BST树初始平衡化过程。在查找过程中，如查找失败，当在失败位置处插入失败元素而引起树的不平衡，也应该调整其平衡。这就是BST平衡化。

假设由于在二叉排序树上插入结点而失去平衡的最小子树根结点的指针为a（即a是离插入结点最近，且平衡因子绝对值超过1的祖先结点），则失去平衡后进行调整有四中情况：

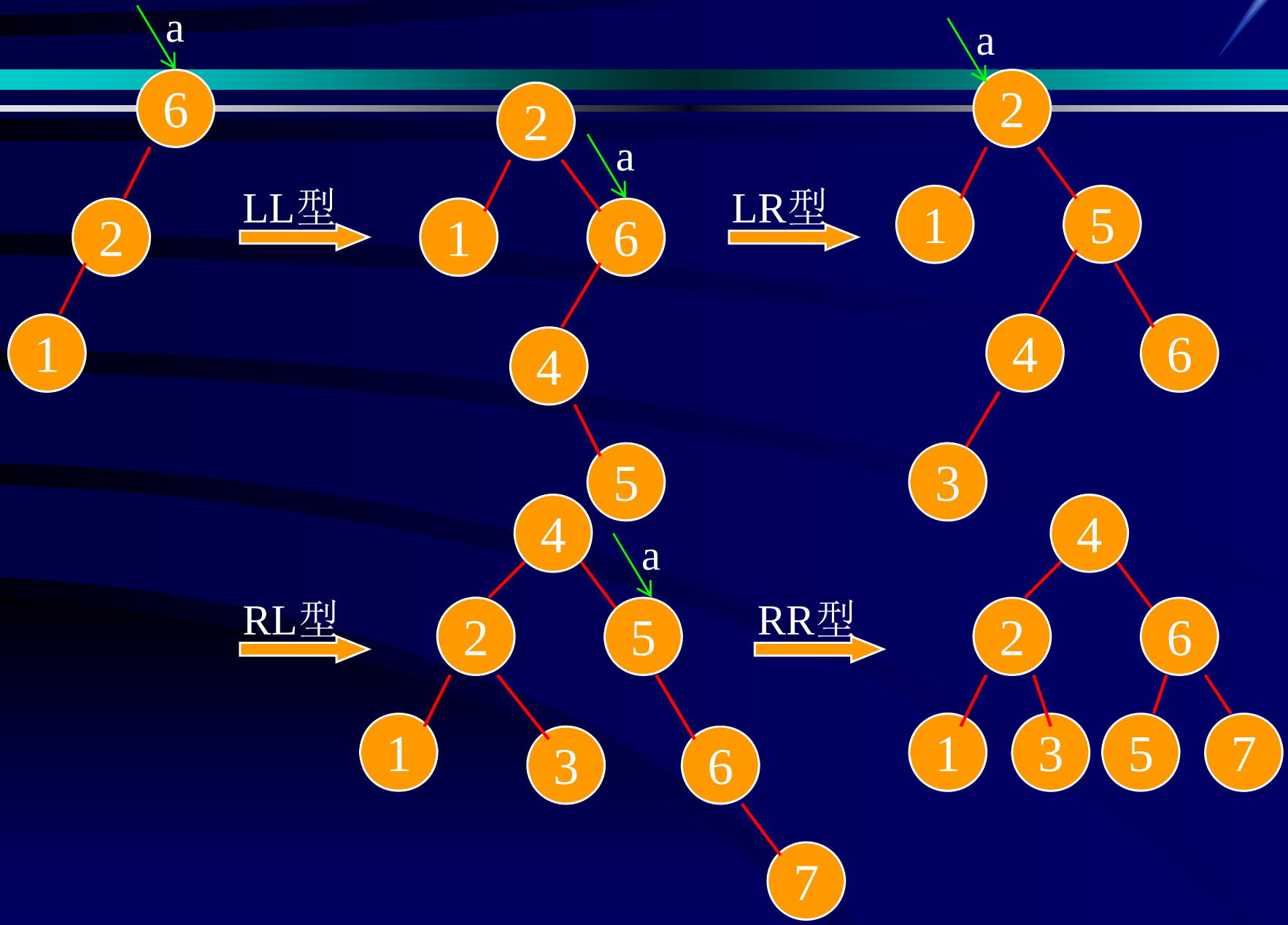
1.单向右旋转(LL)：由于在*a的左子树根结点的左子树上插入结点而导致以*a为根的子树失去平衡，则需进行向右一次顺时针旋转操作；

2.单向左旋转(RR)：由于在*a的右子树根结点的右子树上插入结点而导致以*a为根的子树失去平衡，则需进行向左一次逆时针旋转操作；

3.双向先左后右旋转(LR)：由于在*a的左子树根结点的右子树上插入结点而导致以*a为根的子树失去平衡，则需进行先左旋转后右旋转两次操作；

4.双向先右后左旋转(RL)：由于在*a的右子树根结点的左子树上插入结点而导致以*a为根的子树失去平衡，则需进行先右旋转后左旋转两次操作。

初始序列：6 2 1 4 5 3 7



9.3动态查找表（续）

平衡树查找性能分析

性能分析:

查找过程中和给定值进行比较的关键字个数不超过树的深度。

含有n个关键字的平衡树的最大深度为 (见p334参考书目[1])： $\text{Log}_{(1+\sqrt{5})/2}(\sqrt{5}*(n+1))$

平衡树上进行查找的时间复杂度 $T(n) = O(\log_2 n)$。

9.3动态查找表（续）

B-树的查找

B-树查找=查找表+多路平衡树+B-树性质

B-树(B- Tree)：或者是一棵空树；或者为满足下列特性的m叉树：

(1)树中每个结点至多有m-1棵子树；

(2)若根结点不是叶子结点，则至少有两棵子树；

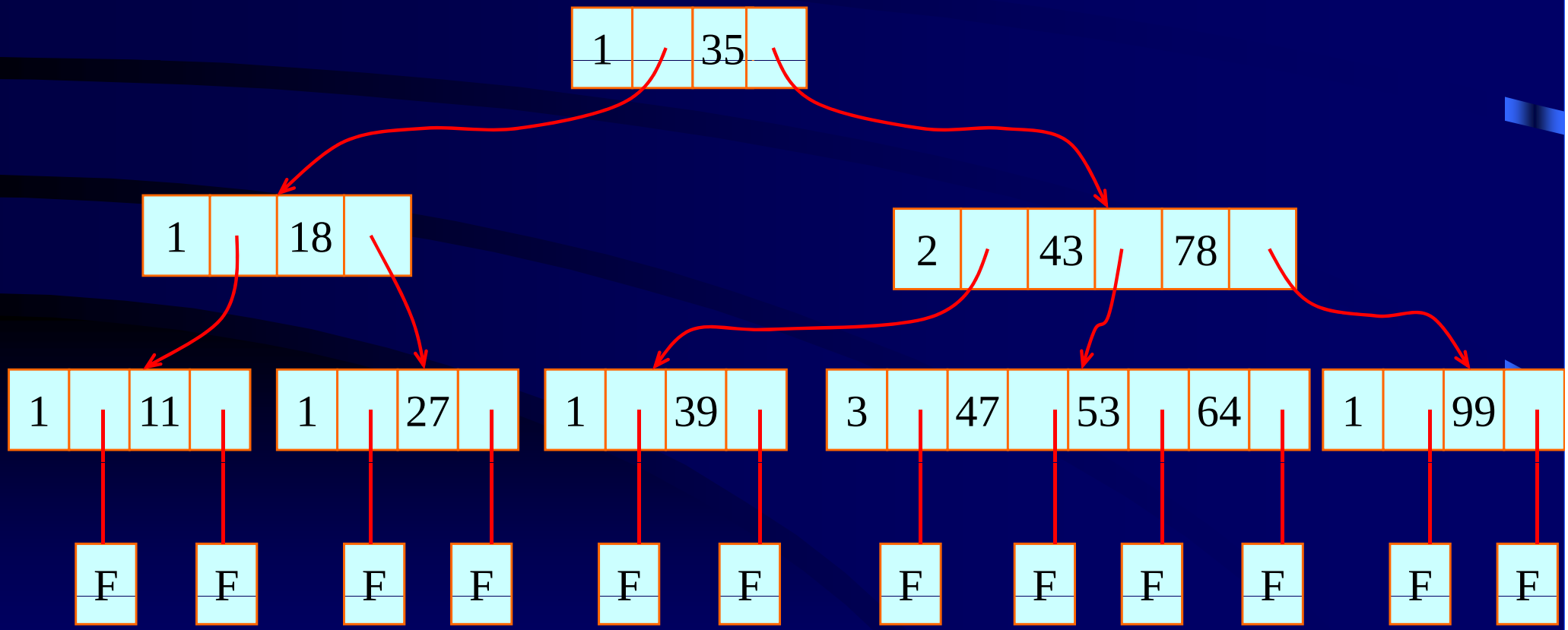
(3) 除根之外的所有非终端结点至少有[m/2]棵子树；

(4)所有非终端结点中包含下列信息数据$(n,A_0,K_1,A_1,K_2,A_2,\ldots,K_n,A_n)$，其中：$K_i(i=1,\ldots,n)$为关键字，且$K_i < K_{i+1}$ $(i=1,\ldots,n-1)$；A_i为指向子树根结点的指针，且指针A_{i-1}所指子树中所有结点的关键字均小于$K_i(i=1,\ldots,n)$，A_n所指子树中所有结点的关键字均大于K_n，$n([m/2]-1<=n<=m-1)$为关键字个数（n+1为当前结点子树个数）

(5)所有叶子结点都出现在同一层次上，并且不带信息。

9.3动态查找表（续）

B-树的查找



```
#define   m   3                  //B-树的阶
typedef struct  BTNode{
    int                  keynum;          //结点中关键字个数
    struct  BTNode       *parent          //指向双亲结点
    KeyType              key[m+1];        //关键字向量，0号单元未用
    struct  BTNode       *ptr[m+1]        //子树指针向量
    struct  BTNode       *dataptr[m+1]    //元素指针向量,0号单元未用
}BTNode,   *Btree;
```

```
typedef struct {
    BTNode      *pt;      //指向找到的结点
    int         i;        //1..m在结点中的关键字序号
    int         tag       //1:查找成功;0:查找失败
}Result;                  //B-树的查找结果类型
```

9.3动态查找表（续）

```
Result SearchBTree(Btree  T, KeyType K ) {
 //在m阶B树T上查找关键字K，返回结过(pt,i,tag)。若查找成功，则特征值
 //tag=1,指针pt所指结点中第i个关键字等于K；否则特征值tag=0，等于K的关
 //键字应插入在指针pt所指结点中第i个和第i+1个关键字之间
  p = T; q = NULL; found = FALSE; i = 0;  //初始化,p指向待查结点,q指向p双亲
  while (p && !found) {
      n = p->keynum;  i = Search(p,K);  //在p->key[1..keynum]中查找,
                                        //i使得:p->key[i]<=K<=p->key[i+1]
      if (i>0 && p->key[i] = =K)  found = TRUE;   //查找待查关键字
      else {q =p;  p = p->ptr[i]; }     }//while
    if (found)  return (p,i,1);             //查找成功
    else  return (q,i,0);                   //查找不成功,返回K的插入位置信息
} //SearchBTree
```

9.3动态查找表（续）

B-树查找性能分析

性能分析:

B-树上进行查找包含两步：在B-树中找结点；在结点中找关键字。前一操作在磁盘上进行，后一操作在内存中进行。在磁盘上进行一次查找比在内存中进行一次查找耗费时间多得多，因此，在磁盘上进行查找的次数、即待查关键字所在结点在B-树上的层次数，是决定B-树查找效率的首要因素。

含 n个关键字的m阶B-树的最大深度为：

$\text{Log}_{m/2\text{取上整}}((n+1)/2) + 1$。

9.3动态查找表（续）

B-树插入

由于B-树结点中的关键字个数必须大于等于m/2取上整-1，小于等于m-1，因此，(查找失败时)每次插入一个关键字时，首先在最低层的某个非终端结点中添加一个关键字，若超过m-1，则产生结点要“分裂”。

分裂方法：

假设*p结点中已有m-1个关键字，当插入一个关键字后，结点中含有信息为：$m,A_0,(K_1,A_1),\ldots,(K_m,A_m)$,且其中$K_i<K_{i+1}$,$(1<=i<m)$,此时可将*p结点分裂为*p和*p’两个结点，其中*p结点中含有信息为：

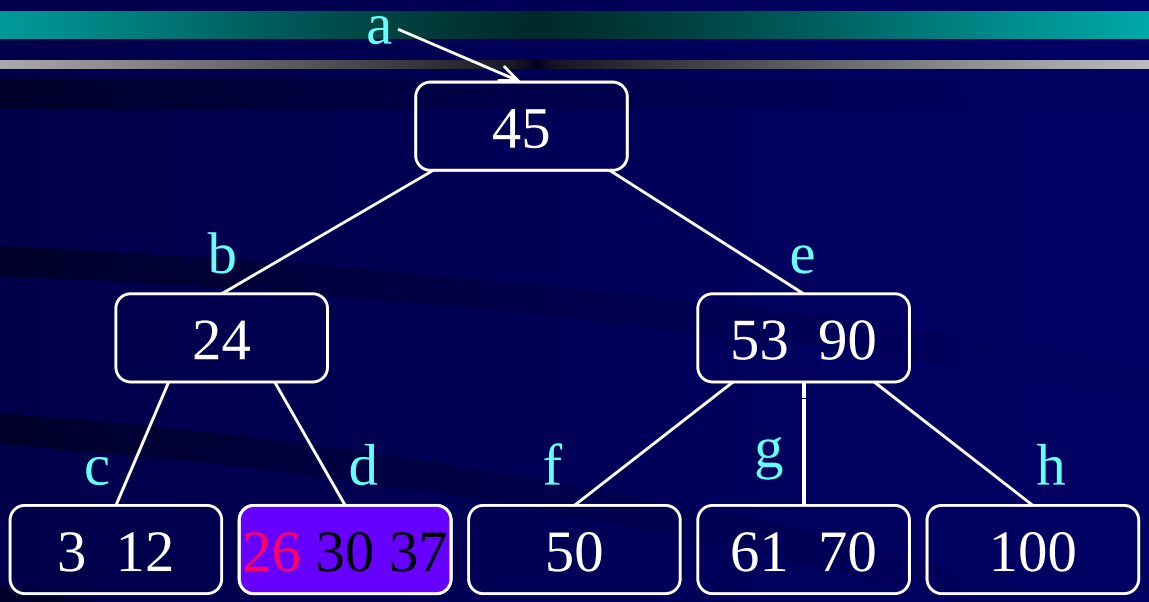
m/2取上整-1, $A_0,(K_1,A_1),\ldots,(K_{m/2取上整-1},A_{m/2取上整-1})$；*p’结点中信息为：

m-m/2取上整, $A_{m/2取上整},(K_{m/2取上整+1},A_{m/2取上整+1}),\ldots,(K_m,A_m)$;而关键字 $K_{m/2取上整}$和指针*p’一起插入到*p的双亲结点中

依次插入关键字：30， 26， 85， 7

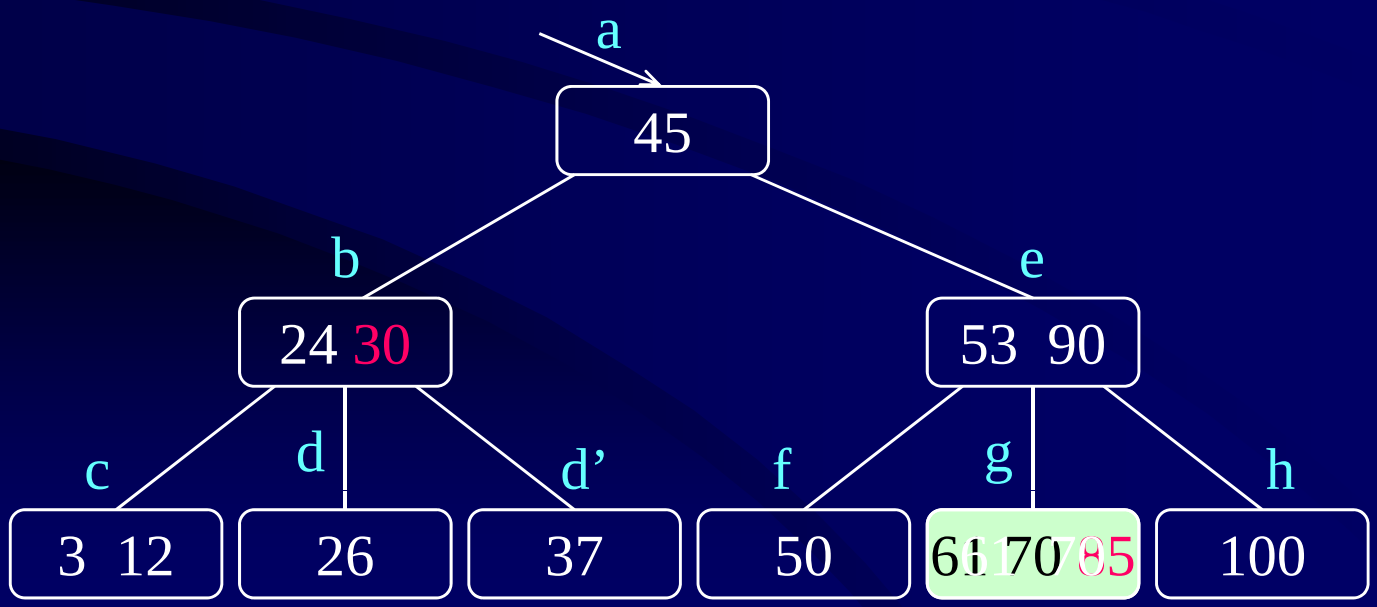
插入30

插入26

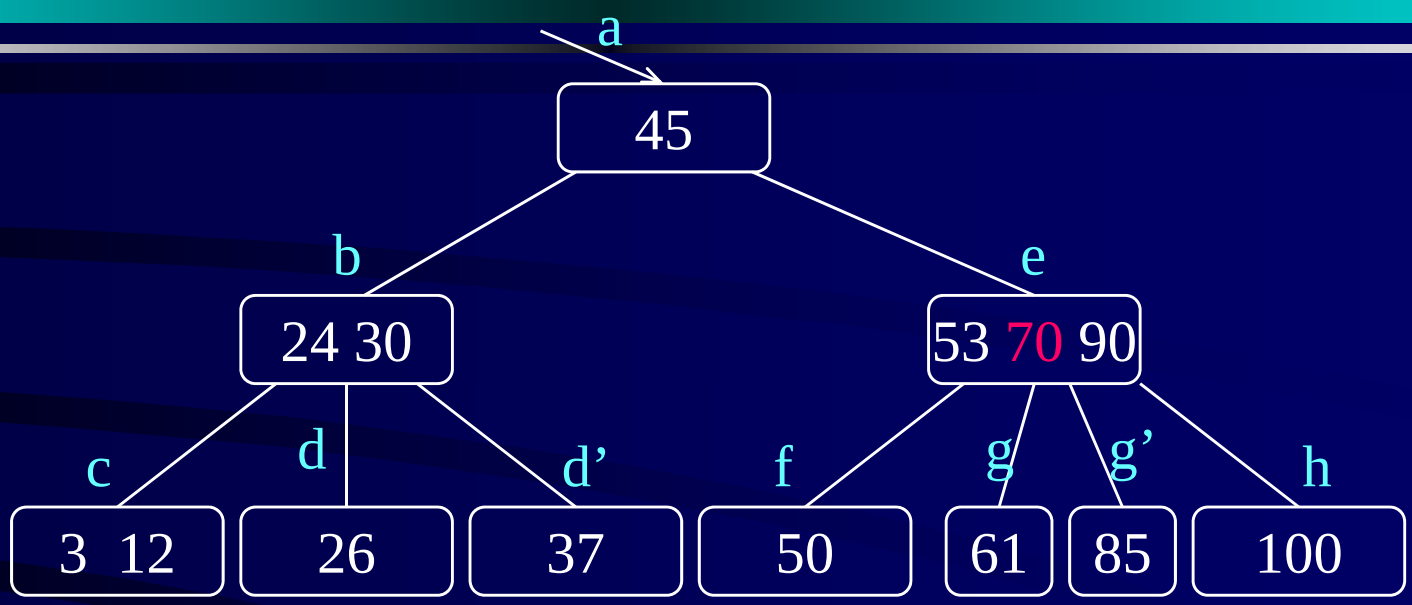


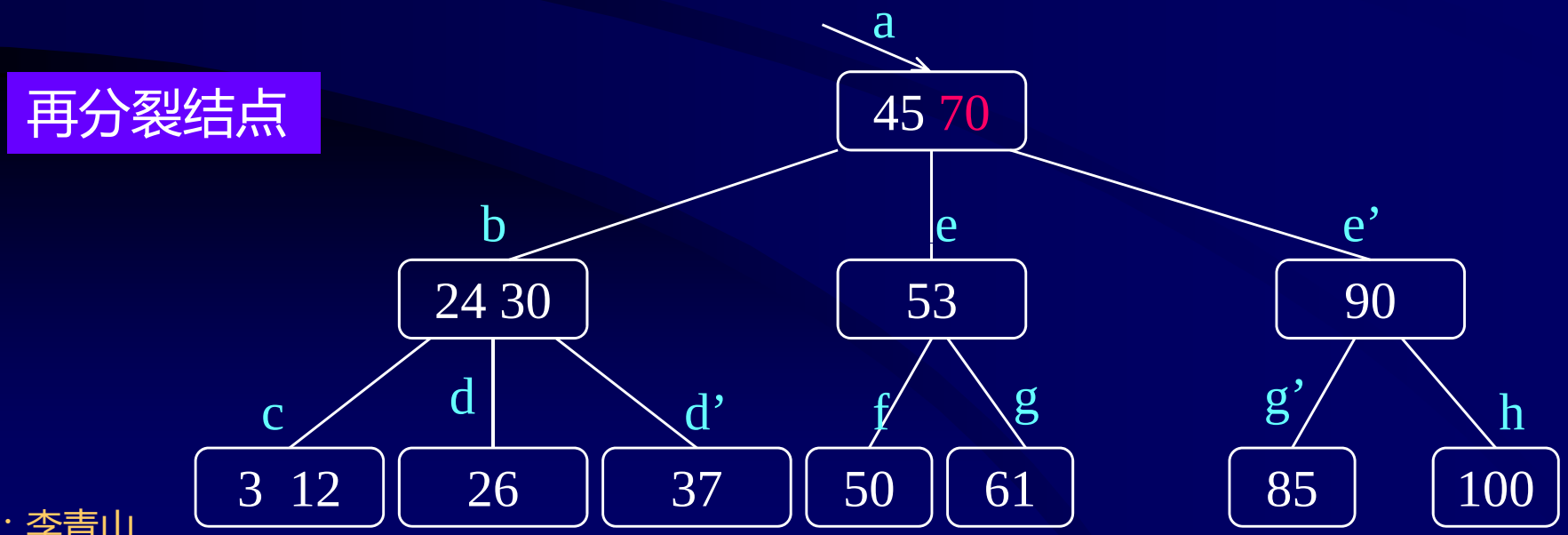
分裂结点

插入85

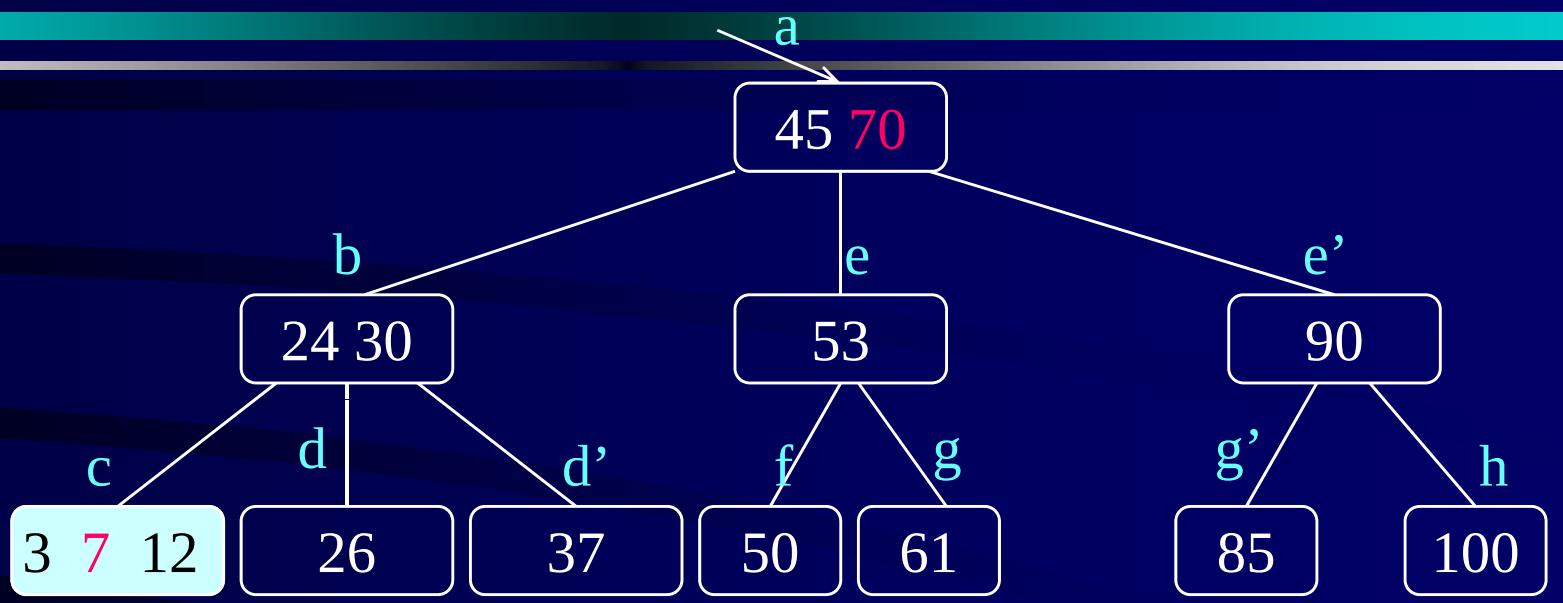


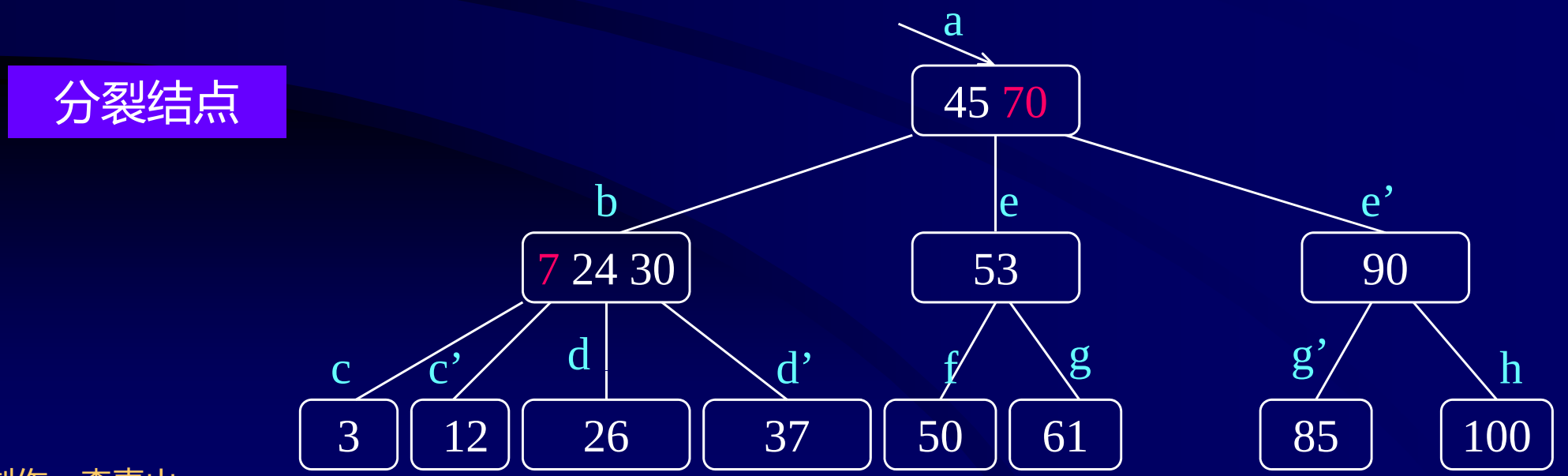
9.3动态查找表（续）


分裂结点
a
45
b
e
24 30
53 70 90
c
d
d’
f
g
g’
h
3 12
26
37
50
61
85
100


再分裂结点
a
45 70
b
e
e’
24 30
53
90
c
d
d’
f
g
g’
h
3 12
26
37
50
61
85
100

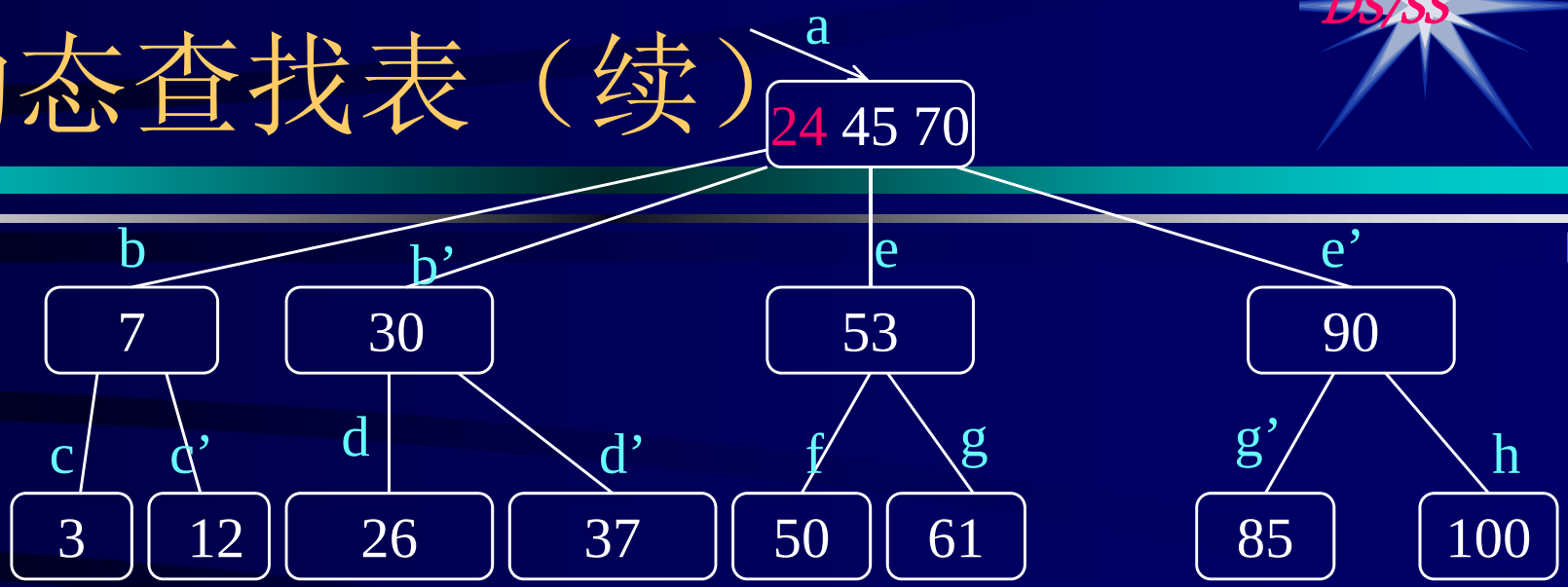
9.3动态查找表（续）


插入7
a
45 70
b
e
e’
24 30
53
90
c
d
d’
f
g
g’
h
3 7 12
26
37
50
61
85
100

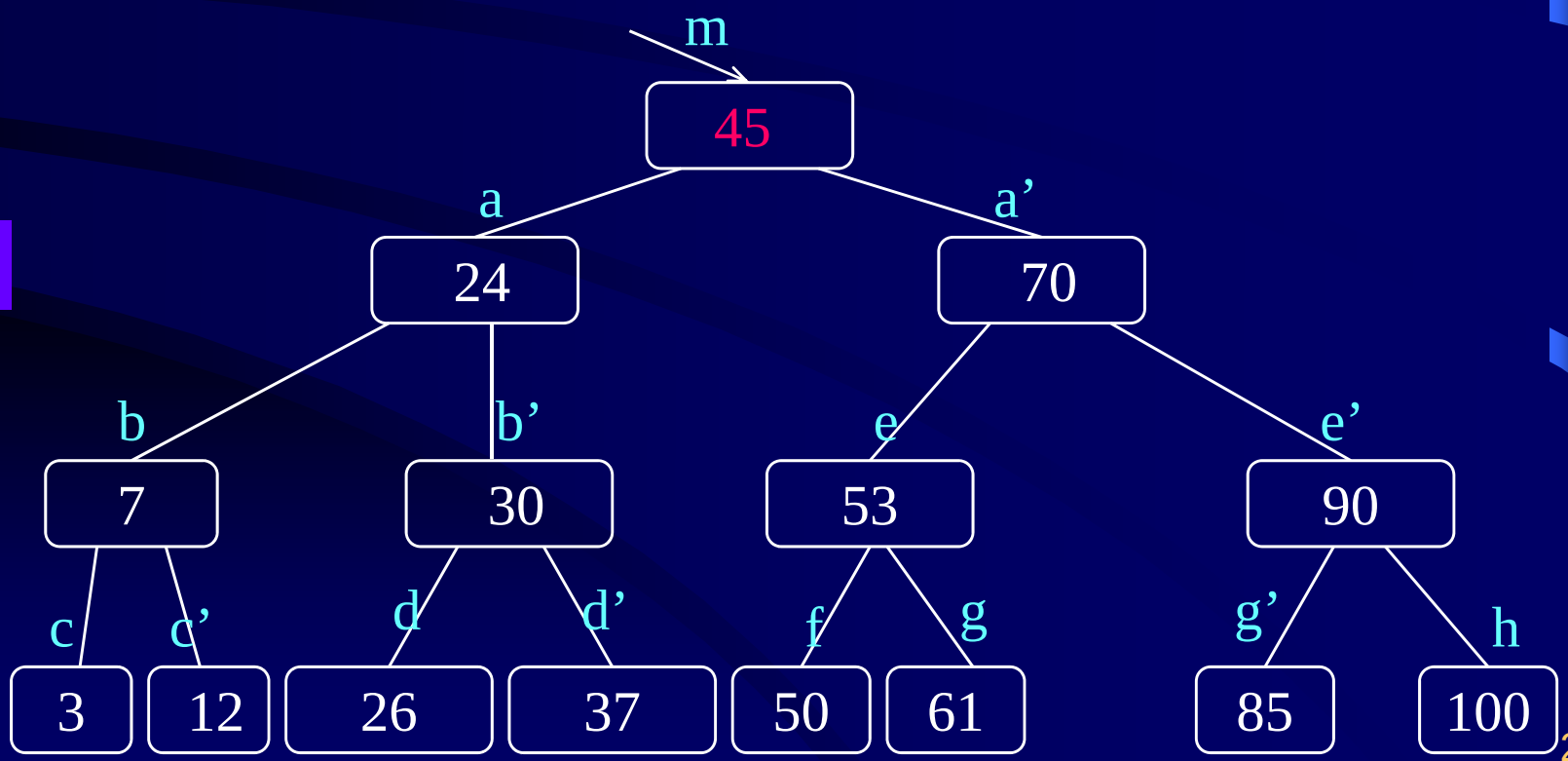

分裂结点
a
45 70
b
e
e’
7 24 30
53
90
c
c’
d
d’
f
g
g’
h
3
12
26
37
50
61
85
100

9.3动态查找表（续）

再分裂结点



三次分裂结点



9.3动态查找表（续）

B-树删除

B-树中元素的删除分两种情况：

*被删除元素K_i所在结点不是最下层的非终端结点。这时将K_i与A_i指针所指向子树中最小关键字Y交换，然后在相应结点中删除Y；

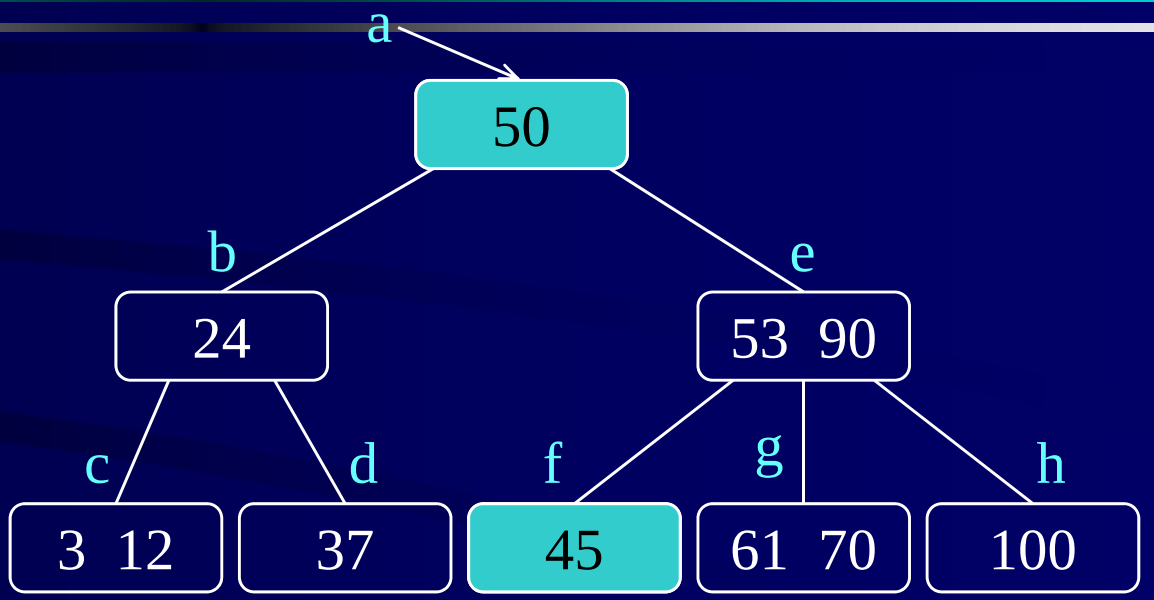
*被删除元素K_i所在结点是最下层的非终端结点。若该结点中关键字数目不少于m/2取上整，则直接删除即可；否则要进行“合并”结点的操作。

这两种情况都可以归结到删除最下层非终端结点中关键字的情形。有三种可能性。

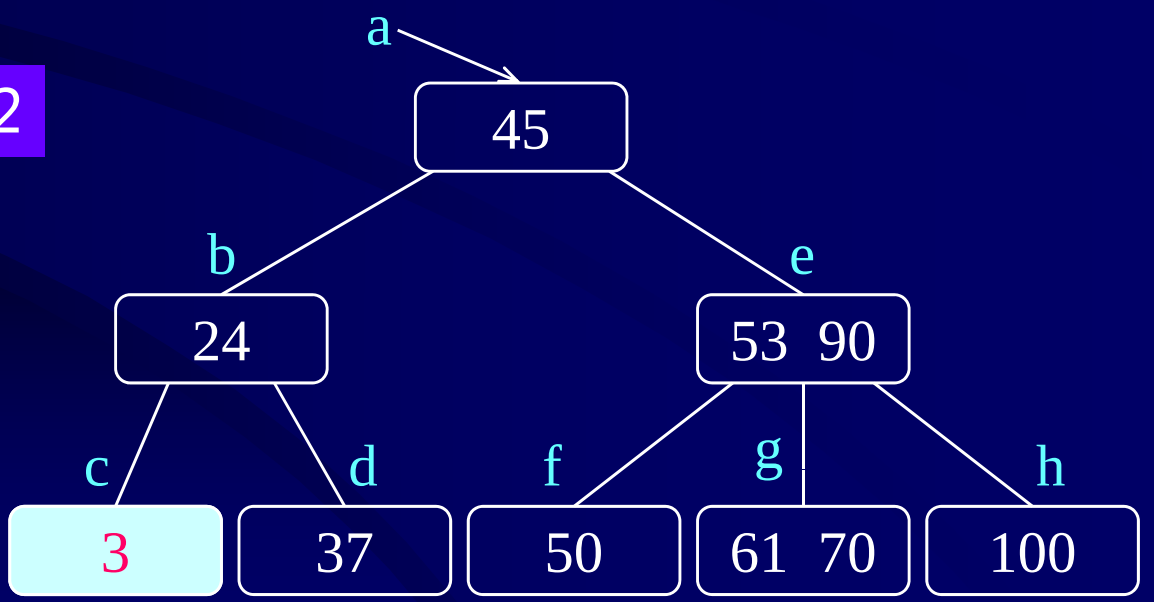
9.3动态查找表（续）

被删除元素K_i所在结点不是最下层的非终端结点。这时将K_i与A_i指针所指向子树中最小关键字Y交换，然后在相应结点中删除Y

(1)被删除关键字所在结点中关键字数目不少于m/2取上整，则只需从该结点中删去关键字K_i和相应指针A_i，树的其余部分不变



删除12

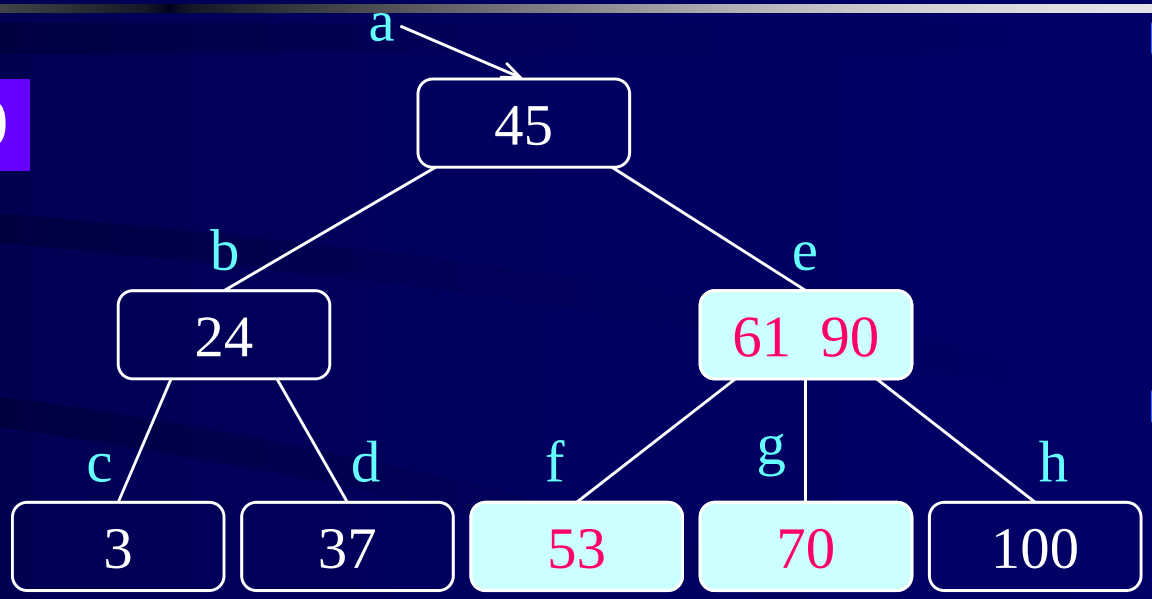


9.3动态查找表（续）

DS/SS

(2)被删除关键字所在结点中关键字数目等于m/2取上整-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取上整-1，则需将其兄弟结点中的最小(或最大)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。

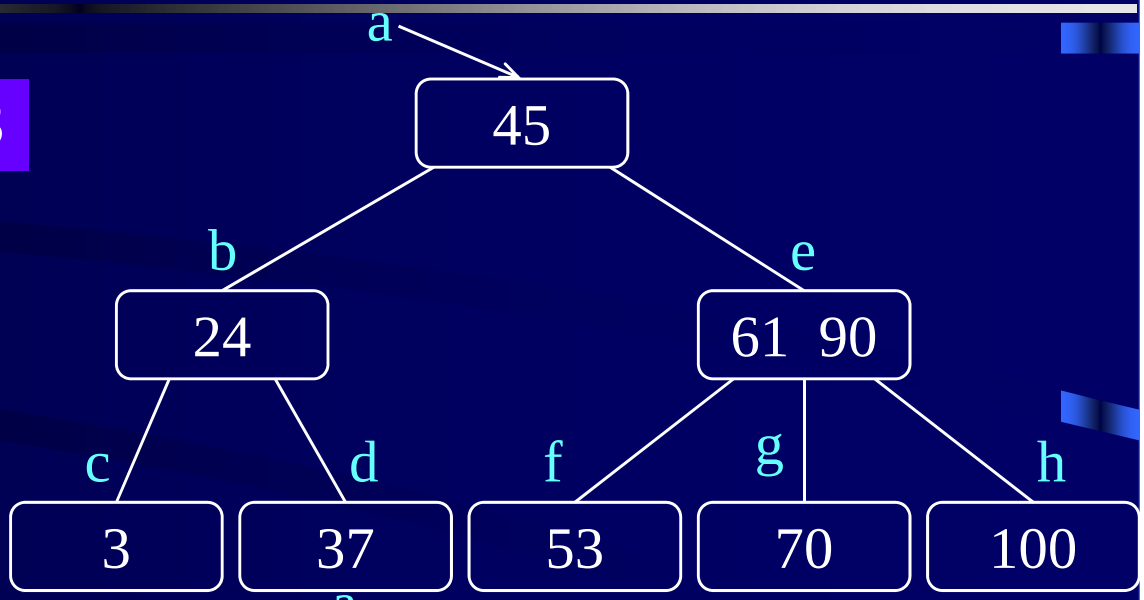
删除50



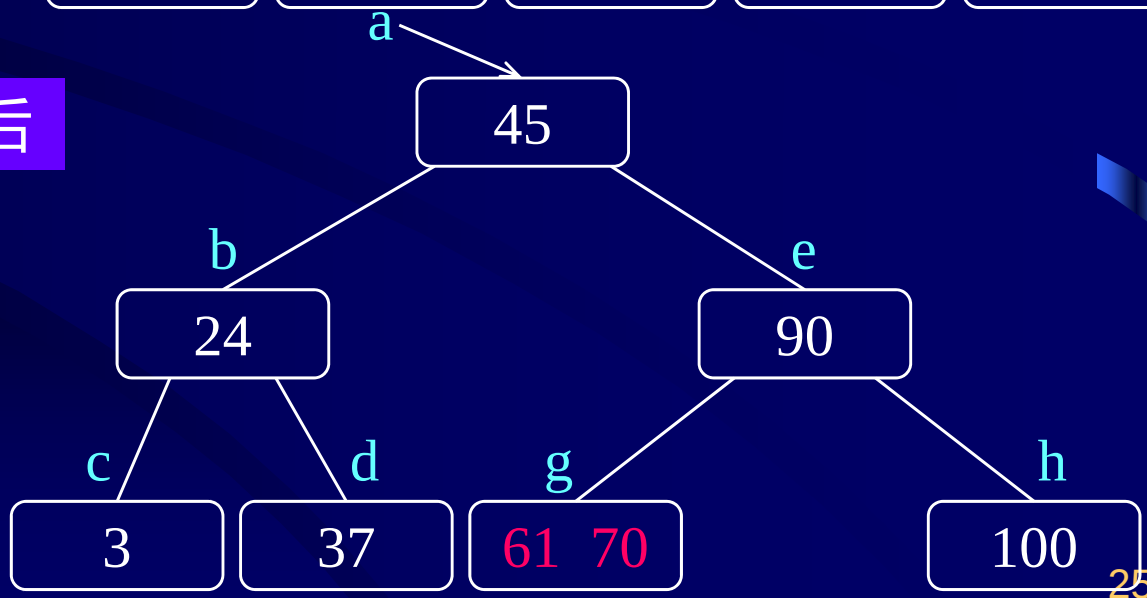
9.3动态查找表（续）

(3)被删除关键字所在结点中关键字数目和与该结点相邻的右兄弟(或左兄弟)结点中的关键字数目均等于m/2取上整-1。设该结点有右兄弟，且其右兄弟结点地址由双亲结点中的指针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，则合并到左兄弟结点中)

删除53



删除后



9.3动态查找表（续）

B+树

B+树查找=查找表+多路平衡树+B+树性质

B+树(B+ Tree)：是B-树的变型树。一棵m阶B+树与m阶B-树差异：

(1)有n棵子树的结点中含有n个关键字；

(2)所有的叶子结点中包含了全部关键字信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接；

(3) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树(根结点)中的最大(或最小)关键字

9.3动态查找表（续）

B+树

B+树上查找运算有两种：一种是从最小关键字起顺序查找，另一种是从根结点开始，进行随机查找。从根结点开始进行随机查找的过程是一个顺指针寻找索引和在叶子结点的关键字中进行查找交叉进行的过程。若非终端结点关键字和给定值相等，并不终止，而是继续向下直到叶子结点。因此，在B+树，不管查找成功与否，每次查找都走了一条从根到叶子结点的路径。