数据结构与算法设计

课程内容

课程内容：数据结构部分

概述	数组与广义表	图	外部排序
线性表	串	查找	
栈与队列	树	内部排序	

课程内容：算法设计部分

概述	贪心	……	计算模型
分治	回溯	……	可计算理论
动态规划	……	……	计算复杂性

Contents

本章内容

- 静态查找表
- 动态查找表
- 哈希表

查找

查找表：由同一类型的数据元素（或记录）构成的集合。

查找表的基本操作

1) 查询某个“特定的”数据元素是否在表中
2) 检索某个“特定的”数据元素的各种属性
3) 插入一个数据元素
4) 删去某个数据元素

查找

查找就是在数据集合中寻找满足某种条件的数据元素。
条件：关键字等于给定值

查找的结果通常有两种可能：
1、查找成功：即找到满足条件的数据元素。
应报告该元素在结构中的位置，或具体信息。
2、查找不成功：即查找失败。
应报告一些信息，如失败标志，位置等

查找

- 记录：由若干数据项构成的数据元素。
- 关键字：能标识一个数据元素（或记录）的数据项。
- 主关键字：能唯一地标识一个记录的关键字。
- 次关键字：用以识别若干记录的关键字。

学号	姓名	专业	年龄
20190001	王洪	睿信书院	17
20190002	李文	睿信书院	18
20190003	谢军	睿信书院	18
20190004	张辉	睿信书院	20
20190005	李文	睿信书院	19

查找

■静态查找

在指定的表中查找某一个“特定”的数据元素是否存在，检索某一个“特定”数据元素的各种属性。仅作查询和检索。

■动态查找

在查找的过程中同时插入表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素。

查找方法评价

度量查找算法的效率

查找算法的基本操作：比较

平均查找长度(Average Search Length)：为确定记录在查找表中的位置, 在查找过程中关键字的平均比较次数。

设查找成功的总概率为1： $\sum_{i=1}^{n} P_i = 1$

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

- 其中:
  - n 为表长
  - P_i 为查找表中第i个记录的概率,
  - C_i 为找到该记录时比较过的关键字的个数。

查找

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记录或数据元素，若表中存在这样的记录，则称查找成功，查找结果为该记录在查找表中的位置；否则称为查找不成功，查找结果为0或NULL。

学号	姓名	专业	年龄
20190001	王洪	睿信书院	17
20190002	李文	睿信书院	18
20190003	谢军	睿信书院	18
20190004	张辉	睿信书院	20
20190005	李文	睿信书院	19

查找

- 关键字类型定义

```
typedef float KeyType;   //实型
typedef int  KeyType;    //整型
typedef char* KeyType;  //字符串型
```

- 数据元素类型定义

```
typedef struct {
        KeyType key;   关键字域
        ……                其他域
} ElemType;
```

查找

- 对数值型关键字

```
#define  EQ(a,b)  ((a) = = (b))
#define  LT(a,b)  ((a) <  (b))
#define  LQ(a,b)  ((a) <= (b))
```

- 对字符串型关键字

```
#define  EQ(a,b)  (!strcmp((a),(b)))
#define  LT(a,b)  (strcmp((a),(b)) < 0)
#define  LQ(a,b)  (strcmp((a),(b)) <= 0)
```

静态查找表

静态查找

- 顺序表的查找：顺序查找
- 有序表的查找：折半查找
- 索引顺序表的查找：分块查找

静态查找表

查找表

用线性表表示L1=(45,61,12,3,37,24,90,53,98,78)

用顺序表表示静态查找表

用线性链表表示静态查找表

查找方法：顺序查找

顺序表查找

顺序表类型定义

```
typedef struct
{   ElemType  * elem;   // 0号单元留空
     int  length;
 } SSTable;

SSTable  ST;
ST.elem
```

0	1	2	3	4	5	6	7	8	9	10	……	m-1
	45	61	12	3	37	24	90	53	98	78		……

顺序表查找

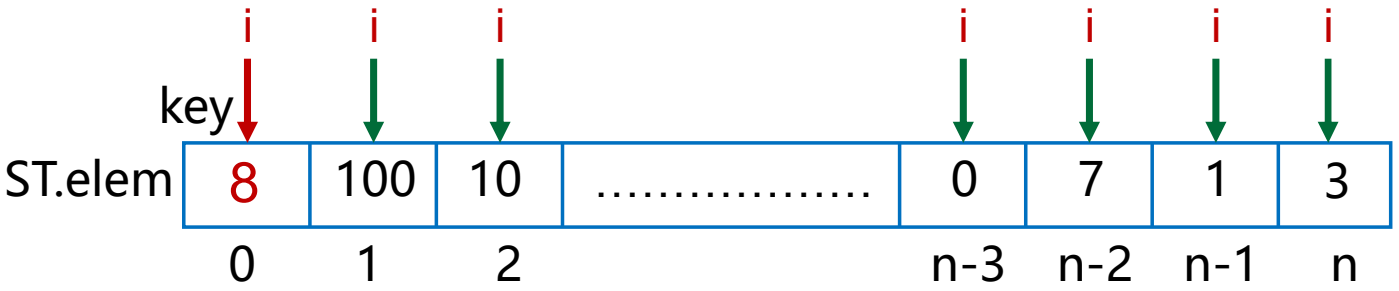
```
int Search_Seq ( SSTable ST, KeyType key )
{ //在顺序表ST中顺序查找其关键字等于key的数据元素。若找到，
  则函数值为该元素在表中的位置，否则为0。
    ST.elem[0].key = key;   //  “哨兵”
   for ( i=ST.length; !EQ(key, ST.elem[i].Key); --i );
   return i;       // 若表中不存在待查元素，i=0
} //Search_Seq
```

免去查找过程中每一步都要检测整个表是否查找完毕

0	1	2	3	4	5	6	7	8	9	10	……	m-1
key	45	61	12	3	37	24	90	53	98	78		……

顺序表查找

例1：在下表中查找 key = 8 的结点。



查找不成功，i = 0

例2：在下表中查找 key = 8 的结点。

查找成功，i = n-2

顺序查找的时间性能分析

定义： 查找算法的查找成功时的平均查找长度(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

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

其中:

n 为表长

P_i为查找表中第i个记录的概率,

C_i为找到该记录时, 曾和给定值比较过的关键字的个数。

顺序查找的时间性能分析

在等概率查找的情况下，元素成功时的查找概率为： 1/n

对顺序表而言，$C_i = i+1$(设置哨兵的情形)

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

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

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

在不等概率查找的情况下

ASL_{ss} 在 $Pn \geq Pn-1 \geq \cdots \geq P2 \geq P1$ 时取极小值

顺序查找的时间性能分析

考虑查找不成功的情况

算法的ASL= 成功时ASL + 不成功时的ASL

对于顺序表

查找不成功的比较次数均为n+1

假设查找成功和不成功的可能性相同，对每个元素的查找概率也相同则

$$P_i = \frac{1}{2n}$$

$$ASL_{ss} = \frac{1}{2n}\sum_{i=1}^{n}(i+1) + \frac{n}{2n}(n+1) = \frac{3}{4}(n+1)$$

顺序查找的特点

顺序查找表的查找算法的优缺点：

算法简单；

无排序要求；

存储结构：顺序、链式；

平均查找长度ASL_{SS}=（n+1）/2；

平均查找长度较大，不适用于表长较大的查找表。

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（头）的位置上。

L1=(45, 61, 12, 3, 37, 24, 90, 53, 98, 78)

有序表查找

查找表：用有序表表示

查找方法：顺序查找

查找过程：在有序顺序表中做顺序查找时，若查找不成功，不必检测到表尾才停止，只要发现有比它的关键码值大的即可停止查找。

```
int OrderSeqSearch1 ( OrderedList& L, DataType x ){
        // 在有序数据表L.data[0..n-1] 中顺序查找关键码值为 x 的数据元素
        for ( i = 0; i < L.n && L.data[i] < x; i++ );   // 顺序比较
        if ( i < L.n && L.data[i] == x )    // 查找成功，返回 x 所在位置
                return i;
        else
                return -1;      //查找不成功, 返回-1
}//OrderSeqSearch1
```

有序表查找

顺序查找有序表的性能分析

♦查找成功的平均查找长度: 与前面的算法相同

♦查找不成功的平均查找长度为(等概率条件下):

$$ASL_{unsucc} = \sum_{i=0}^{n} p_i c_i = \frac{1}{n+1}(1+2+\cdots+n+n) \qquad = \frac{n}{2} + \frac{n}{n+1}$$

有序表查找

查找表：用有序表表示

查找方法：折半查找（二分查找）

查找过程：先确定待查记录所在的范围（区间），然后逐步缩小范围直到找到或找不到该记录为止。

定义：

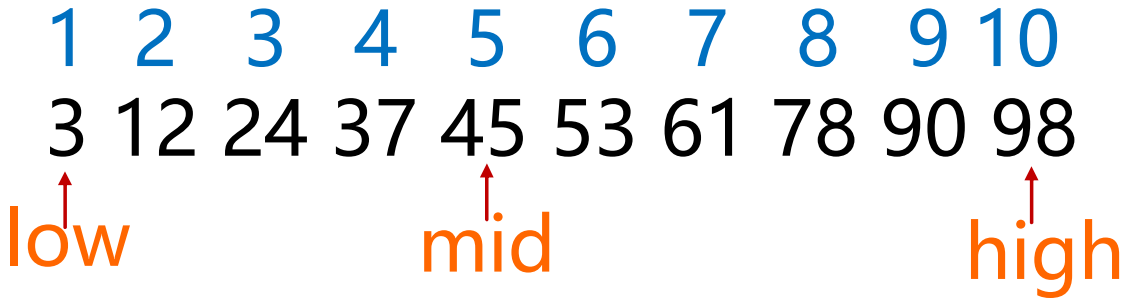
low 指示查找区间的下界

high 指示查找区间的上界

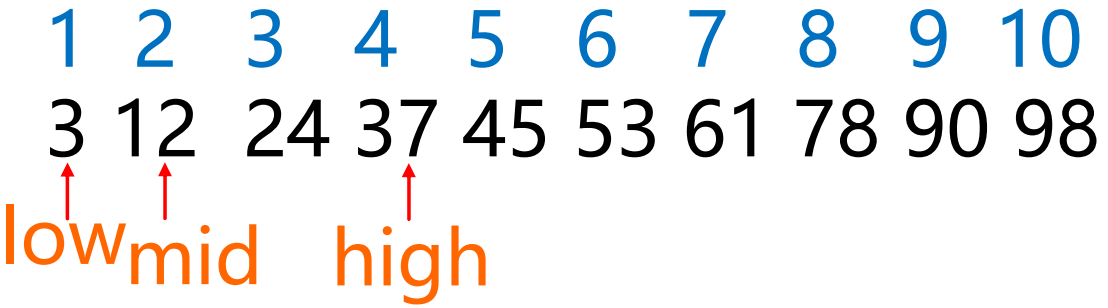
mid = ⌊(low+high)/2⌋ (向下取整)

有序表查找

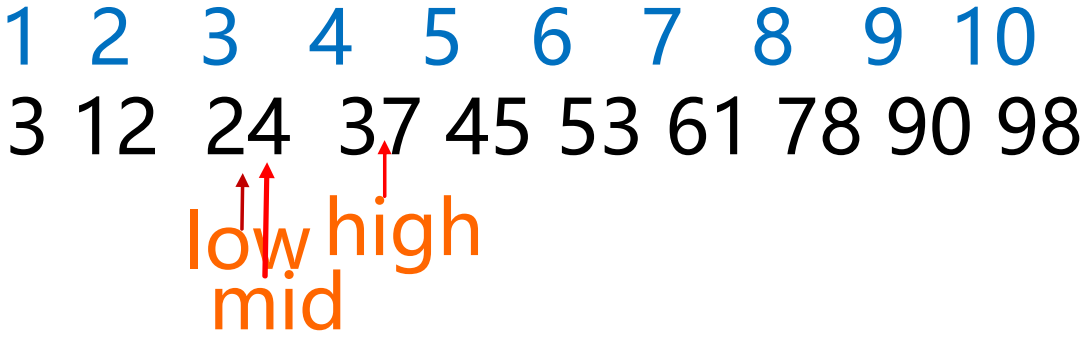
例：查找 Key=24 的记录。



24 < 45



24 > 12



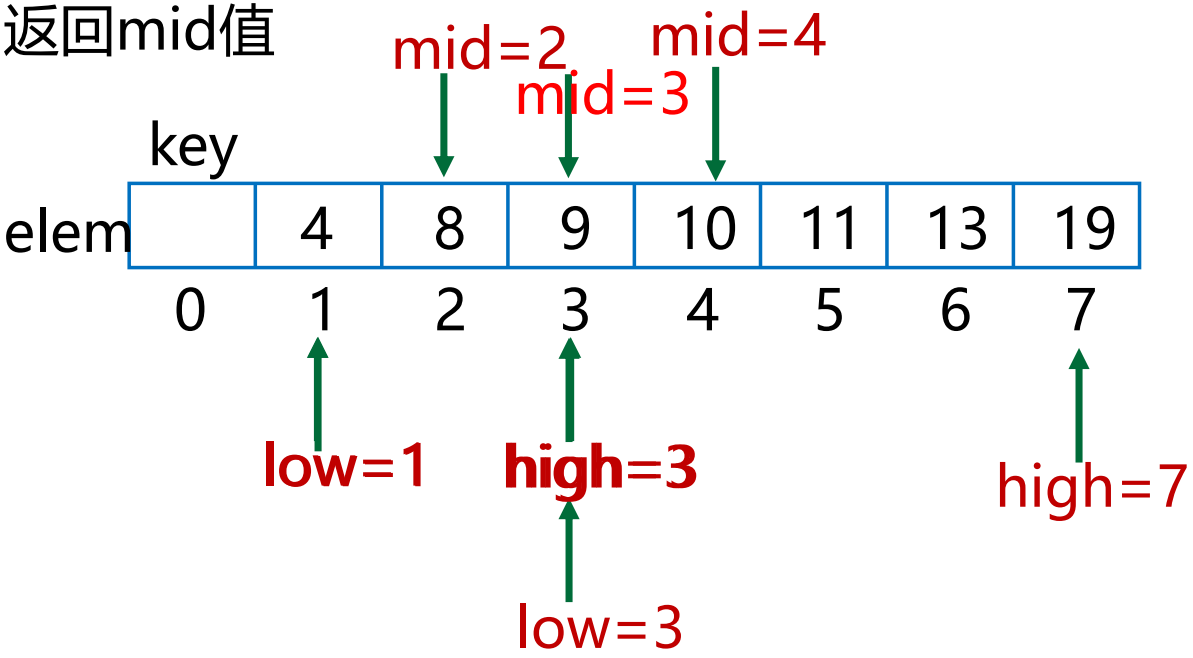
有序表查找

```
int Search_Bin ( SSTable ST, KeyType key )
{ //在有序表ST中折半查找法查找其关键字等于key的数据元素。
  若找到，则返回该元素在表中的位置，否则为0。
   low=1;   high=ST.length;
   while ( low <= high )
   {   mid = (low+high)/2;
       if EQ ( key, ST.elem[mid].key )  return mid;
       else
          if  LT ( key, ST.elem[mid].key ) high=mid-1;
          else  low = mid+1;
   }
   return 0;    //表中不存在待查元素
}//Search_Bin
```

有序表查找

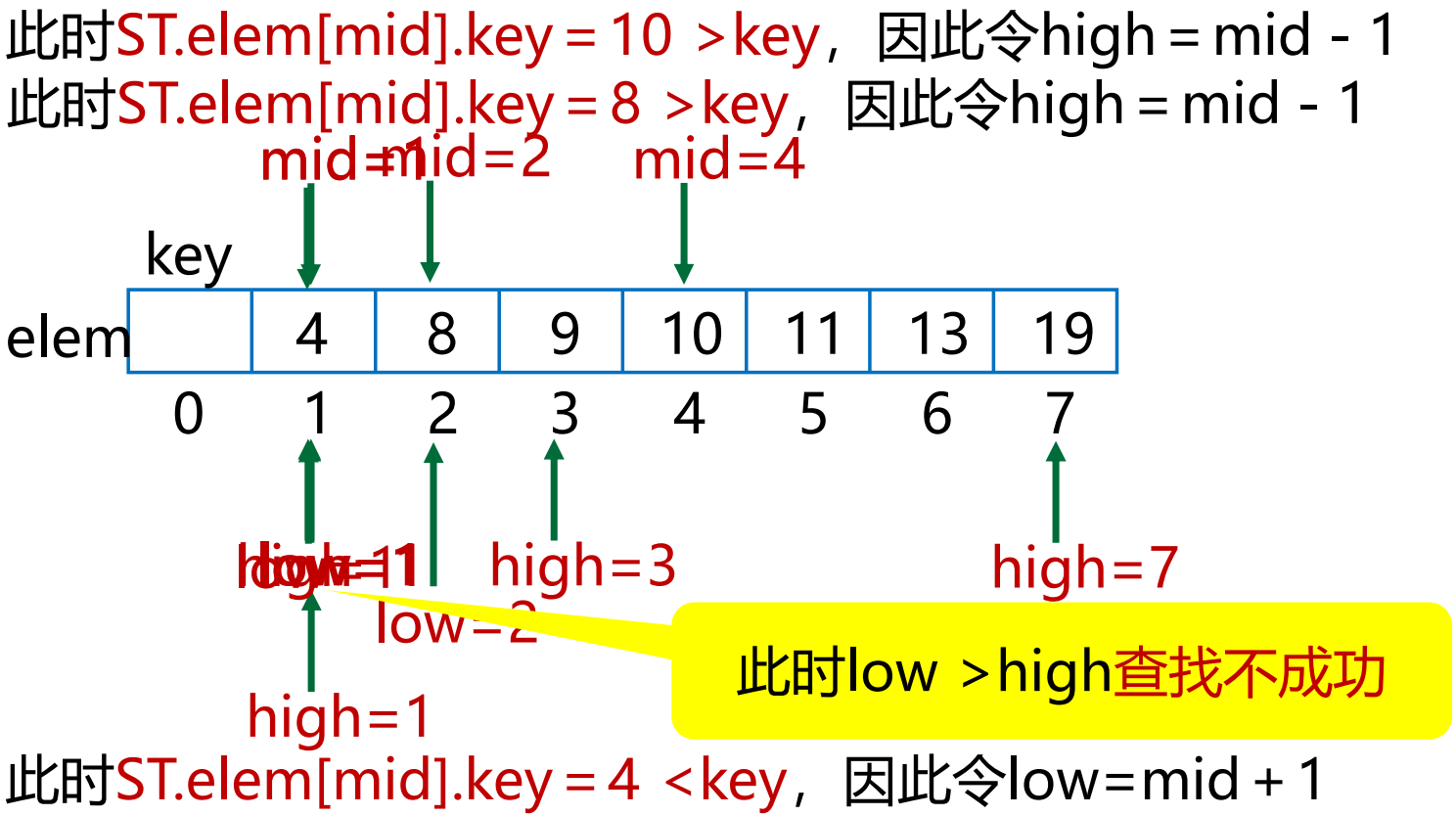
例1：在下表中查找 key = 9 的结点。

此时ST.elem[mid].key = 10 >key，因此令high = mid - 1
此时ST.elem[mid].key = 8 <key，因此令low = mid + 1
此时ST.elem[mid].key = 9 =key，查找成功，
返回mid值



8.1.2 有序表查找

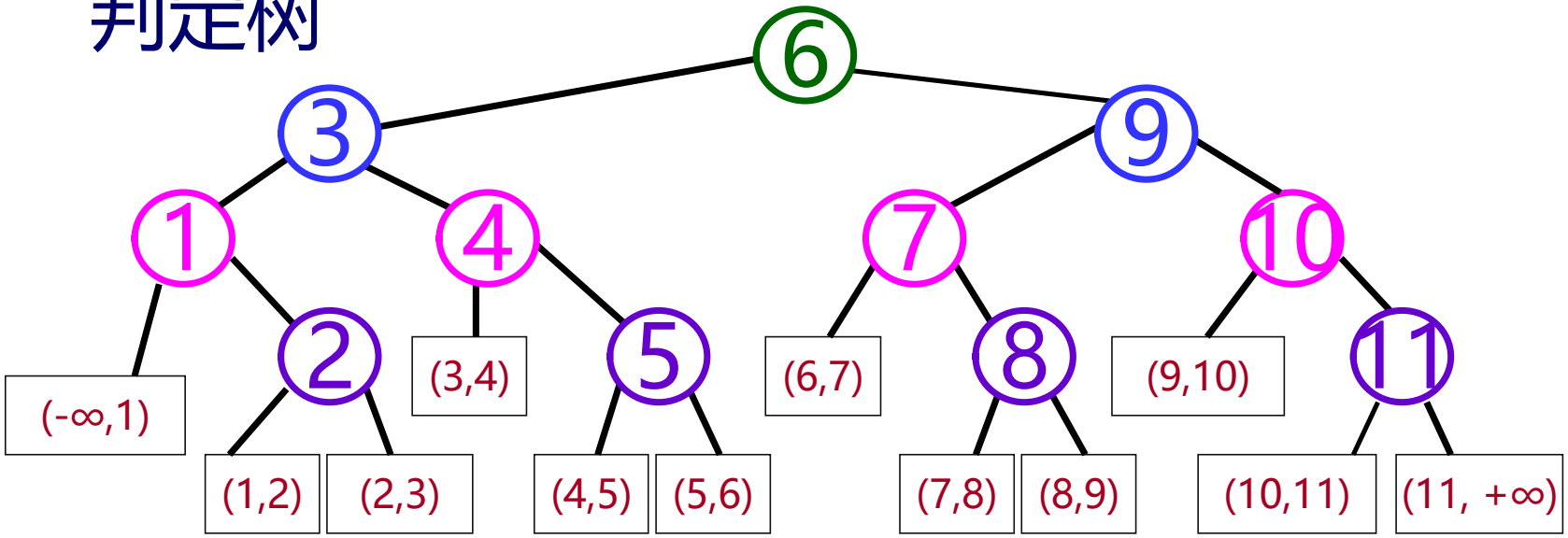
例2：在下表中查找 key = 5 的结点。



分析折半查找的平均查找长度

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

判定树

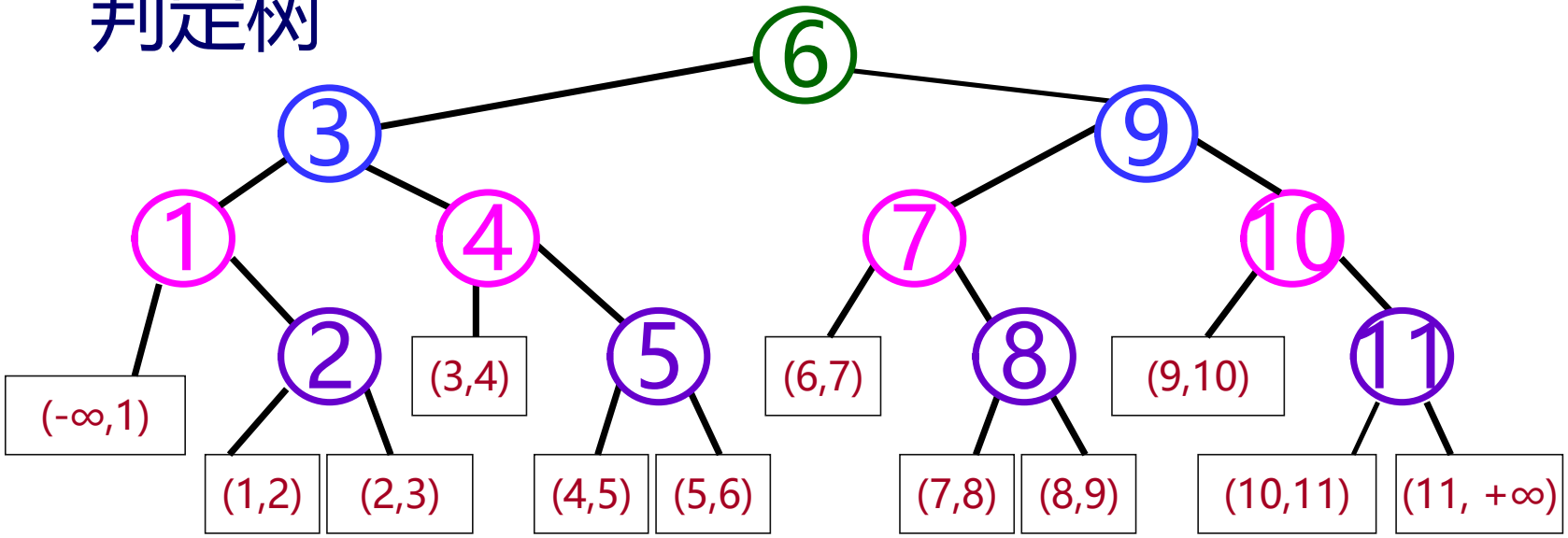


具有n个节点的判定树的深度为$\lceil \log_2(n+1) \rceil$

分析折半查找的平均查找长度

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

判定树

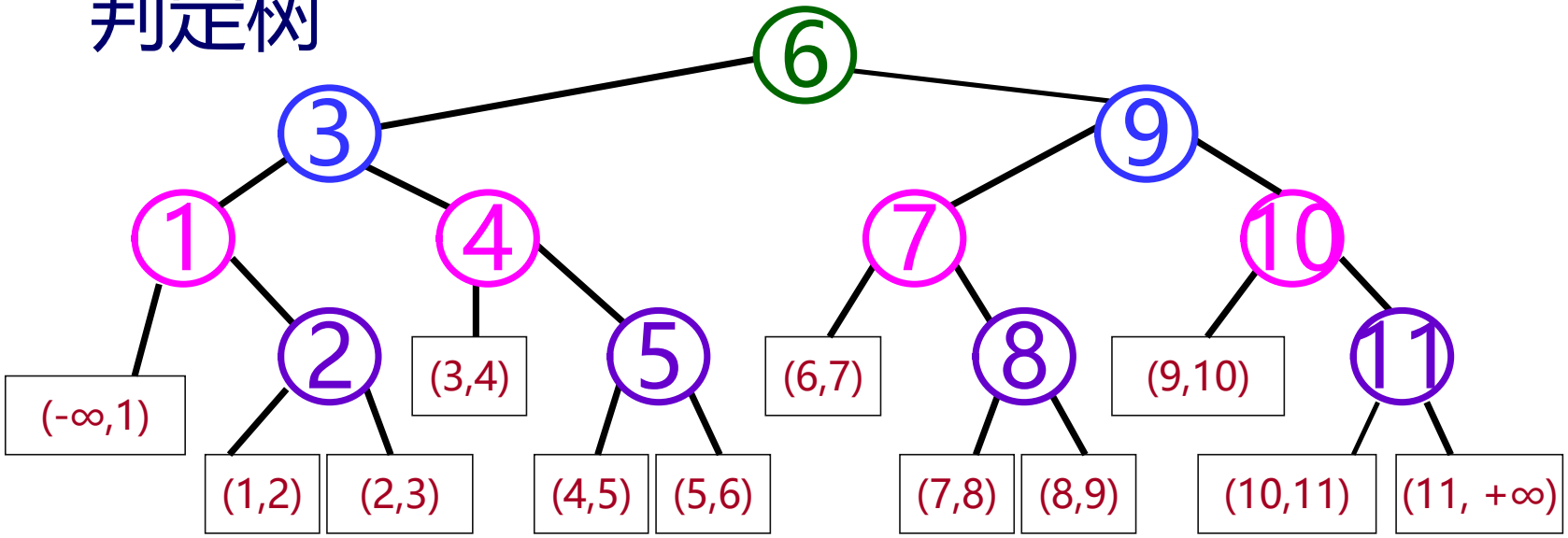


查找成功的平均查找长度 = (1*1+2*2+4*3+4*4) / 11 = 33/11 = 3

分析折半查找的平均查找长度

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

判定树

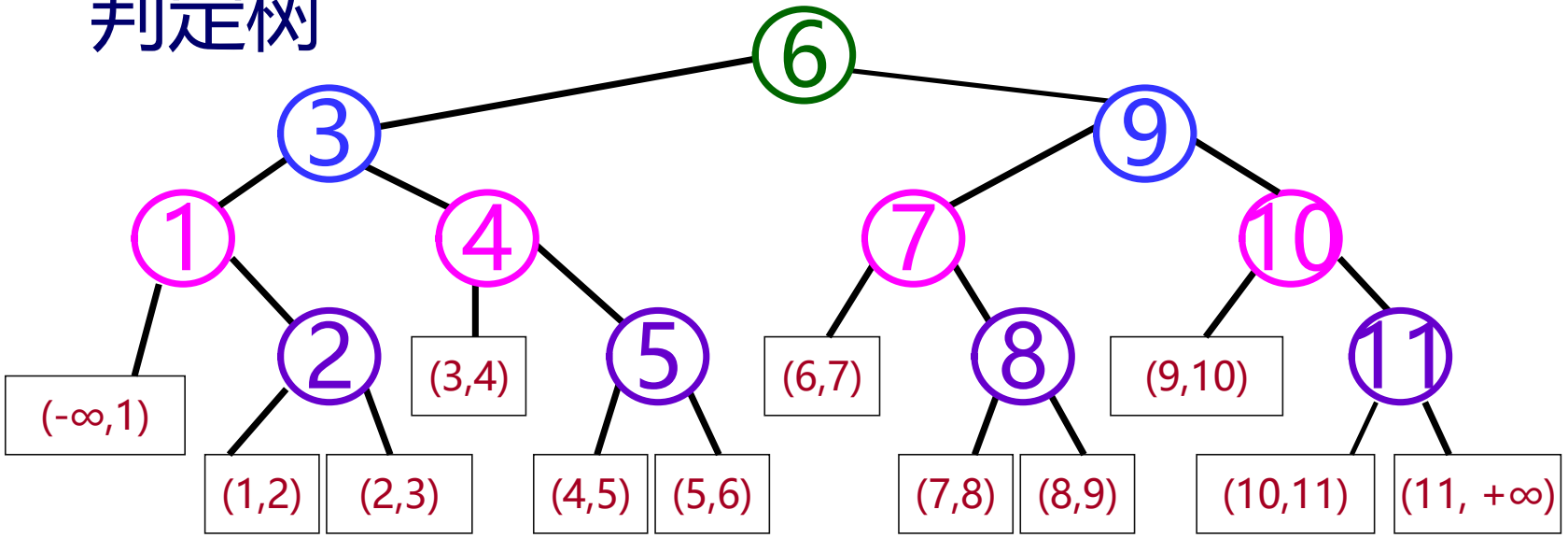


查找失败的平均查找长度 = (4*4+8*5) / 12 = 56/12

分析折半查找的平均查找长度

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

判定树



总的平均查找长度 = 成功的平均查找长度+查找失败的平均查找长度;
= 33/11 + 56/12

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:$h = \lfloor \log_2(n+1) \rfloor$。

假设判定树为满二叉树，则有序表的长度$n=2^h-1$

$$ASL_{ss} = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{j=1}^{h} j \cdot 2^{j-1} = \frac{1}{n}[(\boldsymbol{h-1})\boldsymbol{2^h+1}]$$

$$= \frac{1}{n}[(n+1)\log(n+1) - n] = \frac{n+1}{n}\log_2(n+1) - 1$$

$$ASL_{ss} \approx \log_2(n+1) - 1 \quad (n > 50)$$

有序表查找

折半查找的特点

要求元素按关键字有序。

存储结构：顺序。

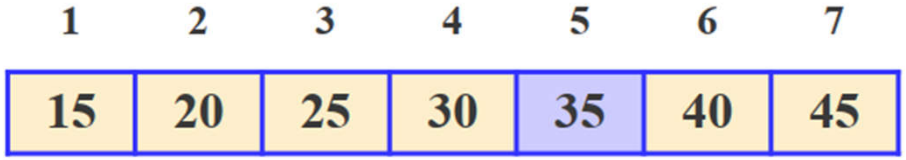
平均查找长度 $ASL_{ss}= \log_2(n+1)-1$

有序表查找-斐波那契查找

斐波那契数列

F(0)=0, F(1)=1,

F(k)=F(k-1)+F(k-2) (k≥2)



确定查找区间的原则是：若表长 n = F(k)-1， 则中间点为 F(k-1)。

左侧子表的长度为F(k-1)-1

右侧子表的长度为F(k-2) -1。

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	…
0	1	1	2	3	5	8	13	21	

有序表查找-斐波那契查找

n = 7 = F(6)-1

中间点选作F(6-1)

1	2	3	4	5	6	7
15	20	25	30	35	40	45

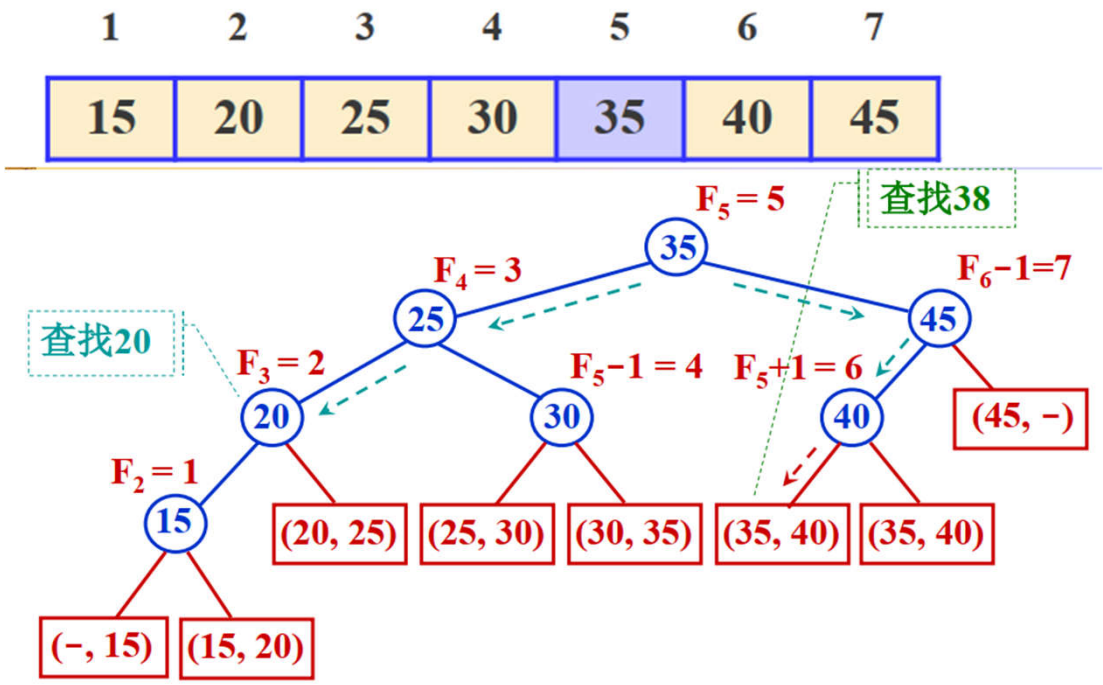
查找算法描述如下：
若查找区间存在，则
若给定值 **x = data[m]**，查找成功，返回**m**；
若给定值 **x < data[m]**，到左子区间中查找；
若给定值 **x > data[m]**，到右子区间中查找；
若查找区间不存在，则查找失败。

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	…
0	1	1	2	3	5	8	13	21	

有序表查找-斐波那契查找

n = 7 = F(6)-1

中间点选作F(6-1)



斐波那契查找的特点：
平均性能比折半查找好；
但是最坏性能比之差；

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	…
0	1	1	2	3	5	8	13	21	

有序表查找-插值查找

插值查找

分割点取序列中的位于取值范围中间的位置;

i	1	2	3	4	5	6	7	8	9	10	11
key	2	4	6	8	10	12	14	15	17	19	20

$$mid = \frac{key - \min}{\max - \min}(location(\max) - location(\min) + 1)$$

$$mid = \frac{6-2}{20-2}(11-1+1) = 2$$

特点： 适合于关键字均匀分布的情况;

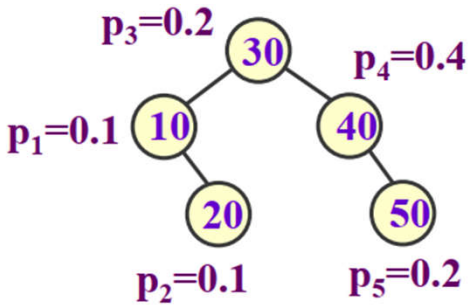
在上述情况下，平均性能比折半查找好;

有序表查找-静态搜索树

假设已知关键字的查找概率, 则最优的搜索树是：

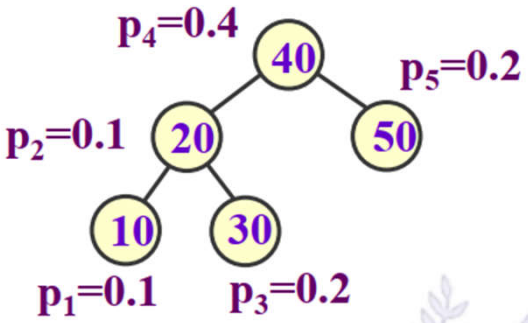
其带权内路径长度之和PH最小的二叉树。

$$PH = \sum_{i=1}^{n} w_i h_i \qquad w_i = cP_i$$



折半查找的判定树1

ASL_{succ}= 0.2*1+(0.1+0.4)*2
+(0.1+0.2)*3 = 2.1



折半查找的判定树2

ASL_{succ}= 0.4*1+(0.1+0.2)*2
+(0.1+0.2)*3 = 1.9

有序表查找-静态最优搜索树

设有序顺序表中关键码为 $k_1, k_2, \ldots, k_n$, 它们的查找概率分别为 $p_1, p_2, \ldots, p_n$, 构成查找树后，它们在树中的层次为 $l_1, l_2, \ldots, l_n$

基于该查找树的查找算法的平均查找长度为

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot l_i$$

使得 ASLsucc达到最小的静态查找树为静态最优二叉排序树。然而，求最优查找树的算法效率较低，达$O(n^3)$,

可求次优查找树，其时间代价减低为$O(n\log_2 n)$。

有序表查找-静态次优搜索树

静态次优搜索树（Nearly Optimal Search Tree）

构造方法：

已知所有记录按照其关键字大小排序

{ k_l , …, k_i , k_{i+1}, …, k_m }

1）在上述序列中取出第i个记录作为根结点, 使得下列值最小

$$\Delta P_i = \left| \sum_{j=i+1}^{m} w_j - \sum_{j=l}^{i-1} w_j \right|$$

2）按照上述步骤分别递归的构造左子树和右子树

二叉排序树和平衡二叉树

动态查找

二叉排序树：或者是一棵空树；或者是具有下列性质的二叉树：

（1）每个结点都有一个作为查找依据的关键码(key)，所有结点的关键码互不相同。

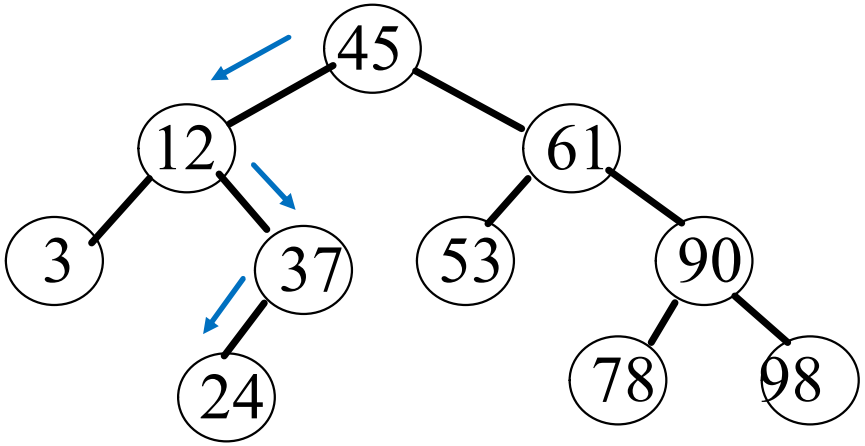
（2）若左子树不空，则左子树上所有结点的值均**小于**根结点的值；

（3）若右子树不空，则右子树上所有结点的值均**大于**根结点的值；

（4）根结点的左、右子树也分别为二叉排序树。

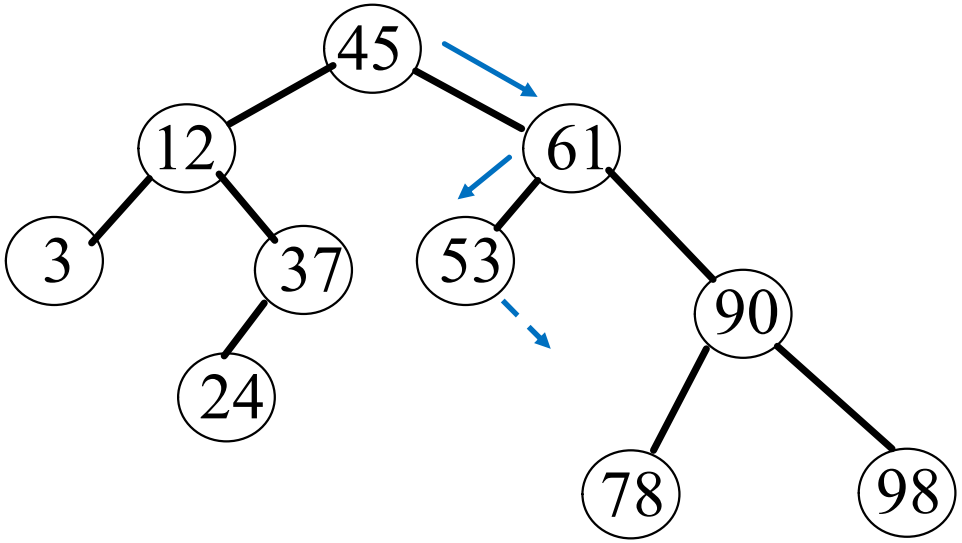
二叉排序树和平衡二叉树

例：在二叉排序树中查找关键字为24的记录。



二叉排序树和平衡二叉树

例：在二叉排序树中查找关键字为60的记录。



二叉排序树和平衡二叉树

二叉排序树的存储

```
typedef struct BiTNode {
    TElemType data;
    struct BiTNode * lchild, * rchild;
} BiTNode,*BTree;

typedef struct  {
   KeyType  key;
   …
} TElemType;
```

二叉排序树和平衡二叉树

二叉排序树的查找——非递归算法

```
BiTree SearchBST ( BiTree T, KeyType key )
{ /*二叉排序树用二叉链表存储。在根指针T所指二叉排序树中查找关键字等于key的记录，若查找成功，则返回指向该记录结点的指针，否则 返回空指针*/
    p = T;
    while ( p && ! EQ ( key, p->data.key) )
    {  if (LT(key, T->data.key) )  p=p->lchild;
       else  p=p->rchild;
    }
    return (p) ;
} //SearchBST
```

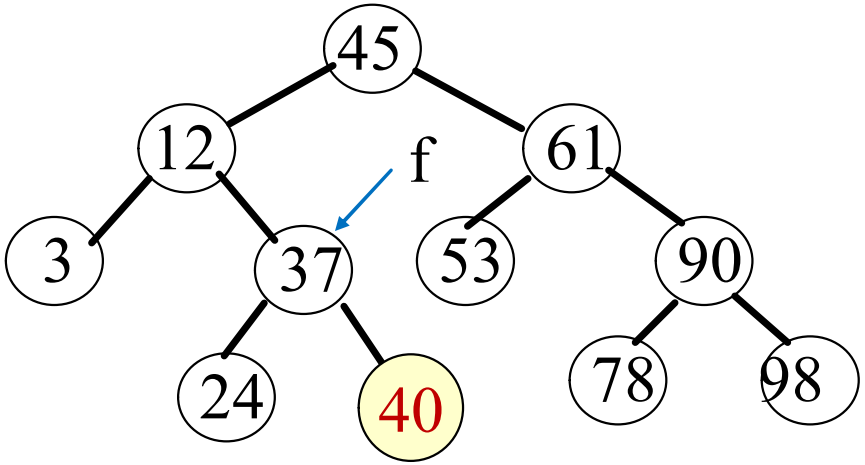
二叉排序树和平衡二叉树

二叉排序树的查找（算法9.5a）——递归算法

```
BiTree  SearchBST( BiTree T, KeyType key )
{ //将二叉链表作为二叉排序树的存储结构
  if ( !T || EQ( key,T ->data.key ) )
       return ( T );
  else
     if ( LT( key, T->data.key ) )
          return ( SearchBST( T->lchild, key ) );
     else
          return ( SearchBST( T->rchild, key ) );
} // SearchBST
```

二叉排序树和平衡二叉树

例：二叉排序树中插入结点 40。



二叉排序树和平衡二叉树

二叉排序树的特点：是一种动态树表，树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。

新插入结点的特点：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。

插入新结点的时机：当查找不成功时。

二叉排序树和平衡二叉树

插入算法：

(1) 若二叉树为空，则首先单独生成根结点。

(2) 执行查找算法，找出被插入结点的双亲结点；

(3) 判断被插入结点是其双亲结点的左孩子结点还是右孩子结点，将被插入结点作为叶子结点插入；

二叉排序树和平衡二叉树

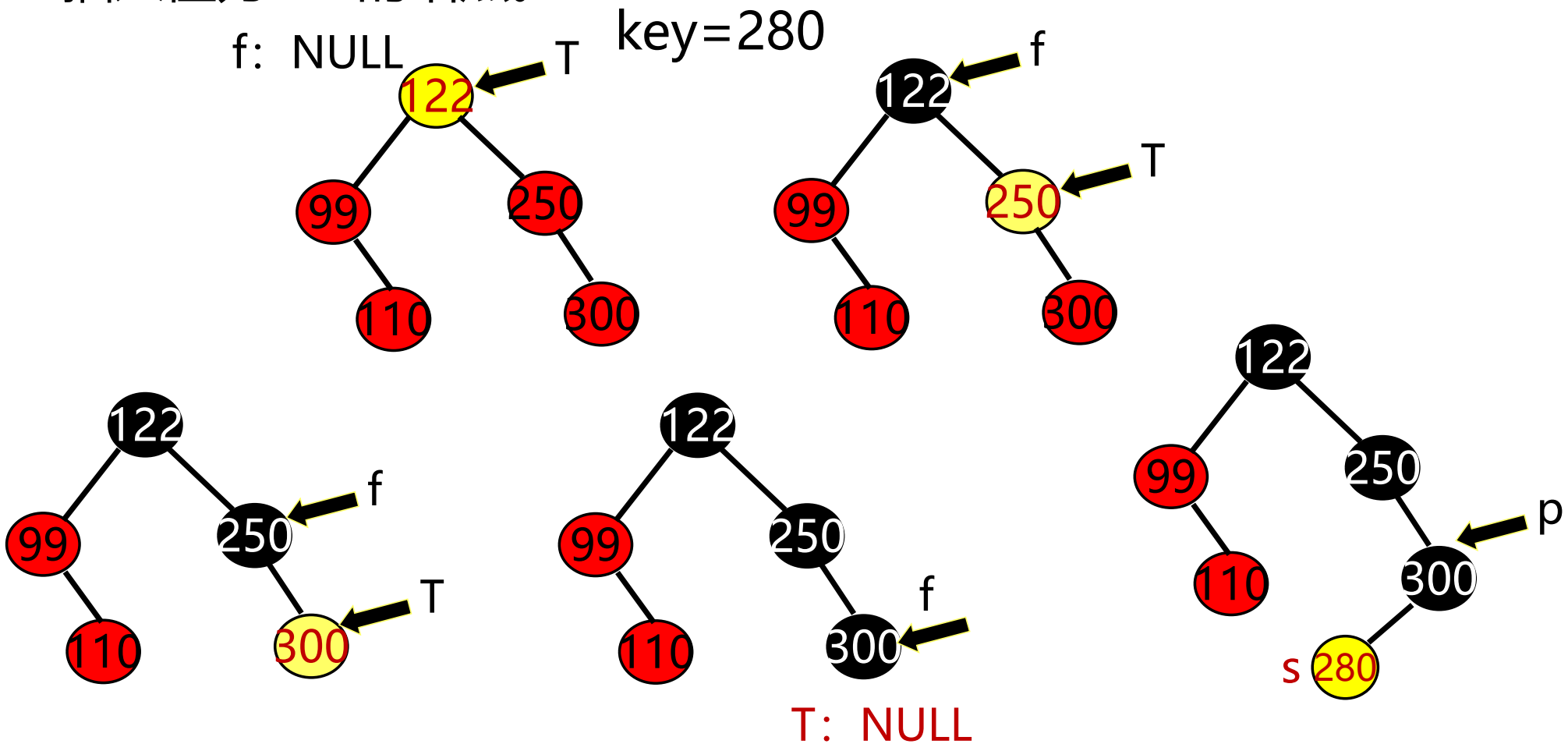
```
Status InsertBST ( BiTree  &T, TElemType e ) {
  if ( ! SearchBST( T, e.key, NULL, p ) )
  {  //查找不成功。p指向访问路径上最后一个结点
      s = (BiTree) malloc( sizeof(BiTNode) );
      s->data=e;   s->lchild=s->rchild=NULL;
      if ( !p ) T=s;         // T为空，被插结点为根结点
        else if ( LT(e.key, p->data.key) )  p->lchild = s;
          else   p->rchild = s;
      return TRUE;        // 插入成功
  }
  else return FALSE; // 树中已有e，不需要再插入
} // InsertBST
```

二叉排序树和平衡二叉树

```
Status SearchBST ( BiTree T, KeyType key, BiTree f, BiTree &p)  {
  // 在T中查找关键字key，若成功，则p指向该结点，返回TRUE；否则p指向查
  找路径的最末结点，返回FALSE。f 指向 T 的双亲，初始调用为NULL
  if ( !T )  { p=f; return FALSE; }  // 查找失败
  else if ( EQ( key, T->data.key ) ) {
      p=T;   return TRUE;              // 查找成功
  }
  else if ( LT(key, T->data.key) )          // 继续查找
          return SearchBST ( T->lchild, key, T, p );
  else   return SearchBST ( T->rchild, key, T, p );
} // SearchBST
```

二叉排序树和平衡二叉树

例1：插入值为280的结点。

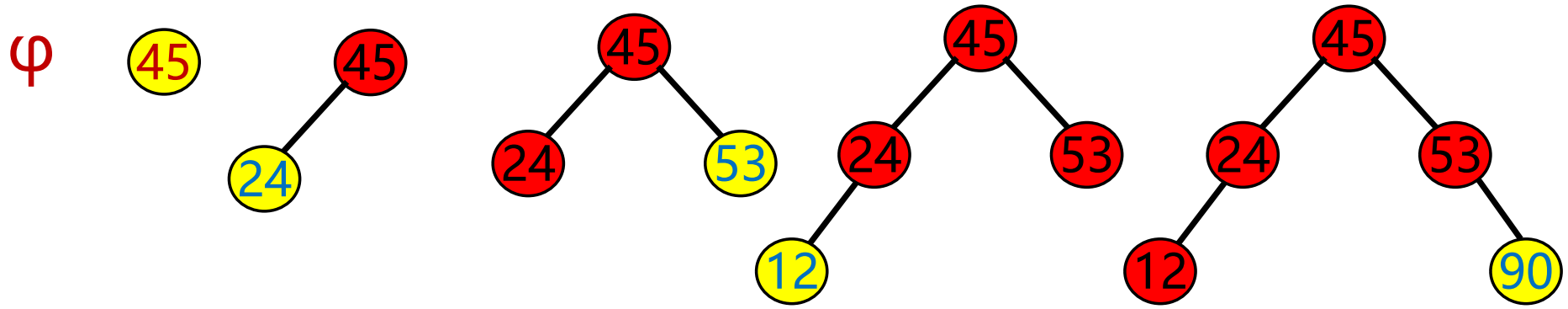


二叉排序树和平衡二叉树

例2：在一棵空树中，查找如下的关键字序列

{ 45，24，53，45，12，24，90 }

生成的二叉排序树如下：



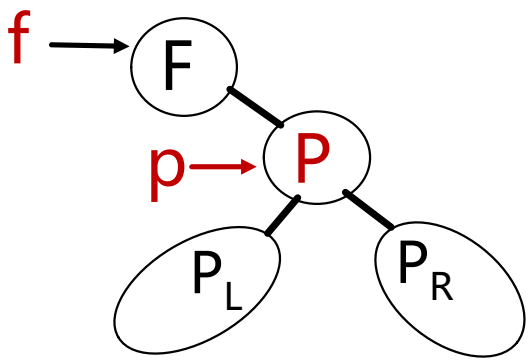
对二叉排序树进行中序遍历可以得到有序序列

二叉排序树和平衡二叉树

二叉树排序删除操作

删除原则：保持二叉排序树的特性。

设： p 指向二叉排序树中被删结点，f 指向 p的双亲结点，且 p 为 f 的右（左）孩子结点。

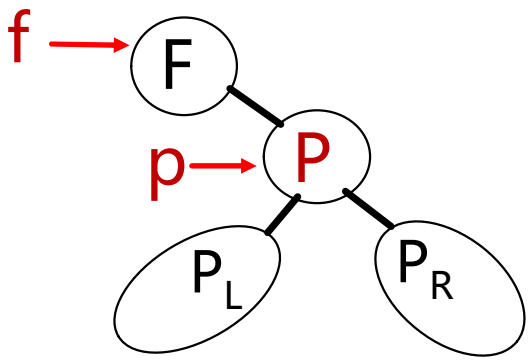


二叉排序树和平衡二叉树

二叉排序树的删除操作

要删除二叉排序树中的p结点，分3种情况：

- p为叶子结点，只需修改 p 双亲 f 的指针：
  f->lchild=NULL / f->rchild=NULL
- p只有左子树或右子树
- p左、右子树均非空



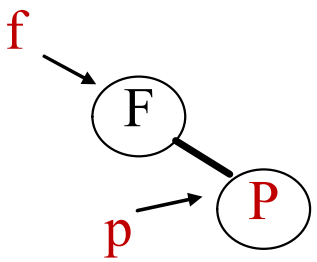
二叉排序树和平衡二叉树

二叉排序树的删除操作

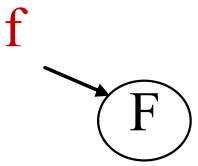
要删除二叉排序树中的p结点，分三种情况：

第1种情况：p为叶子结点，只需修改 p 双亲 f 的指针：

f->lchild=NULL / f->rchild=NULL



删除前



删除后

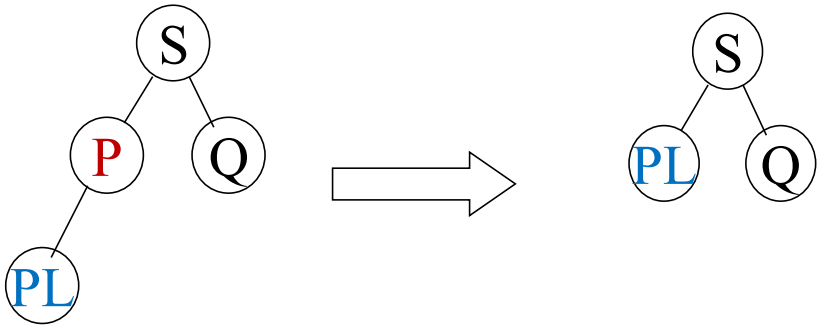
二叉排序树和平衡二叉树

二叉排序树的删除操作

第2种情况： p只有左子树或右子树

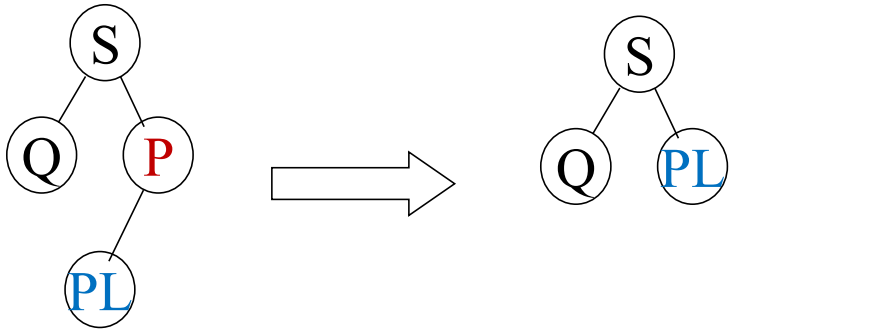
p只有左子树，用p的左孩子代替p　(1)(2)

(第2种情况之2.1)



中序遍历：PL P S Q　　中序遍历：PL S Q

(第2种情况之2.2)



中序遍历：Q S PL P　　中序遍历：Q S PL

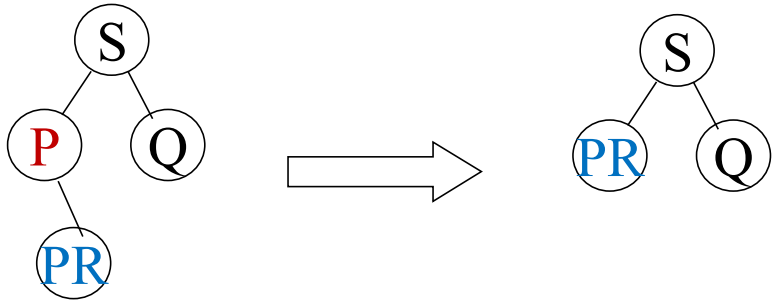
二叉排序树和平衡二叉树

二叉排序树的删除操作

第2种情况： p只有左子树或右子树

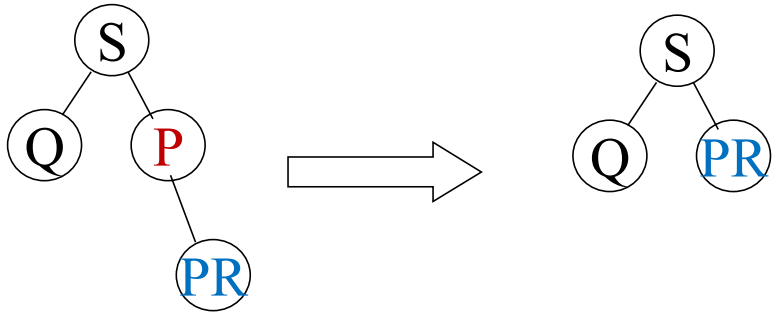
p只有右子树，用p的右孩子代替p

(第2种情况之2.3)



中序遍历：P PR S Q　　中序遍历：PR S Q

(第2种情况之2.4)



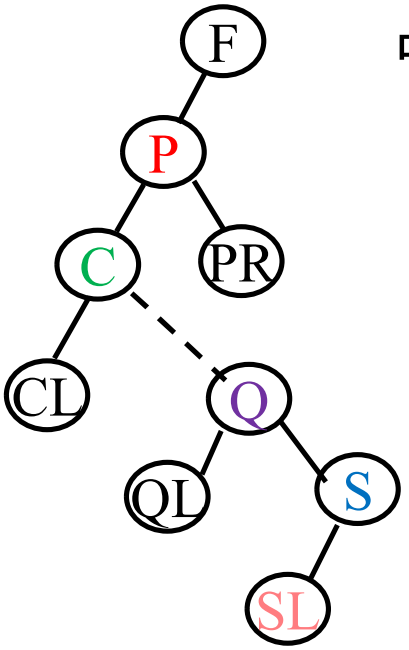
中序遍历：Q S P PR　　中序遍历：Q S PR

二叉排序树和平衡二叉树

二叉排序树的删除操作

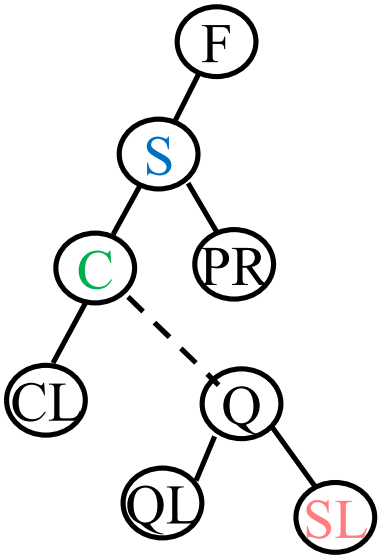
第3种情况： p左、右子树均非空

沿 p 左子树的根 C 的右子树分支找到 S，S的右子树为空，
将 S 的左子树成为 S 的双亲Q 的右子树，用 S 取代 p



中序遍历：CL C … QL Q SL S P PR F

(第3种情况之3.1)



中序遍历：CL C … QL Q SL S PR F

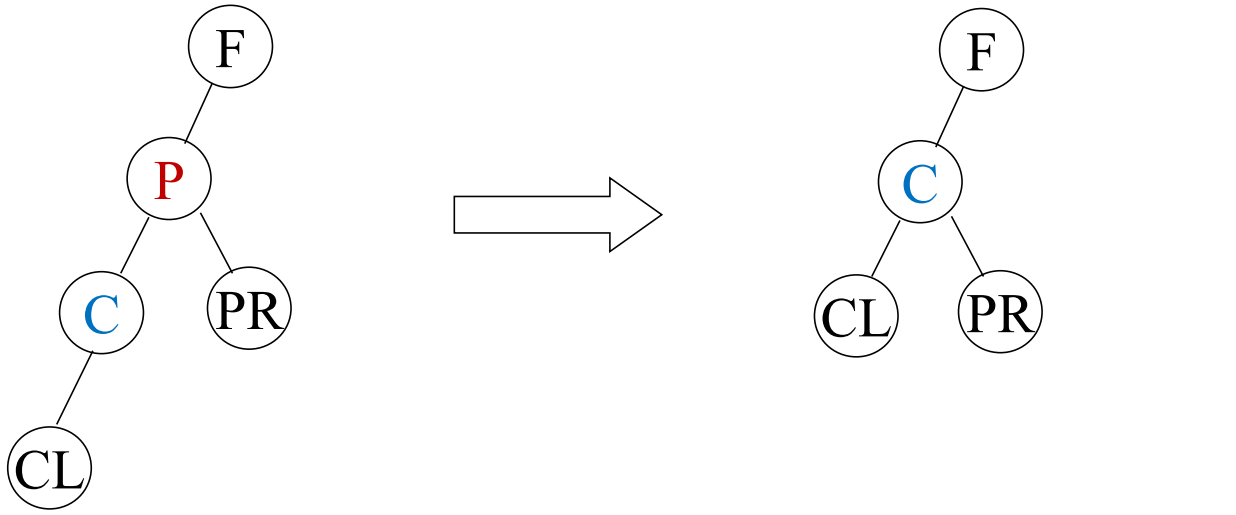
二叉排序树和平衡二叉树

二叉排序树的删除操作

p左、右子树均非空

若 C 无右子树，用 C 取代 p

(第3种情况之3.2)



中序遍历：CL C P PR F　　中序遍历：CL C PR F

二叉排序树和平衡二叉树

二叉排序树的删除操作

要删除二叉排序树中的p结点，分3种情况：

p为叶子结点，只需修改 p 双亲 f 的指针：　第1种情况

f->lchild=NULL　/　f->rchild=NULL

p只有左子树或右子树

p只有左子树，用p的左孩子代替p　第2种情况之2.1，2.2

p只有右子树，用p的右孩子代替p　第2种情况之2.3，2.4

p左、右子树均非空

沿 p 左子树的根 C 的右子树分支找到 S，S的右子树为空，将 S 的左子树成为 S 的双亲Q 的右子树，用 S 取代 p　第3种情况之3.1

若 C 无右子树，用 C 取代 p　第3种情况之3.2

二叉排序树和平衡二叉树

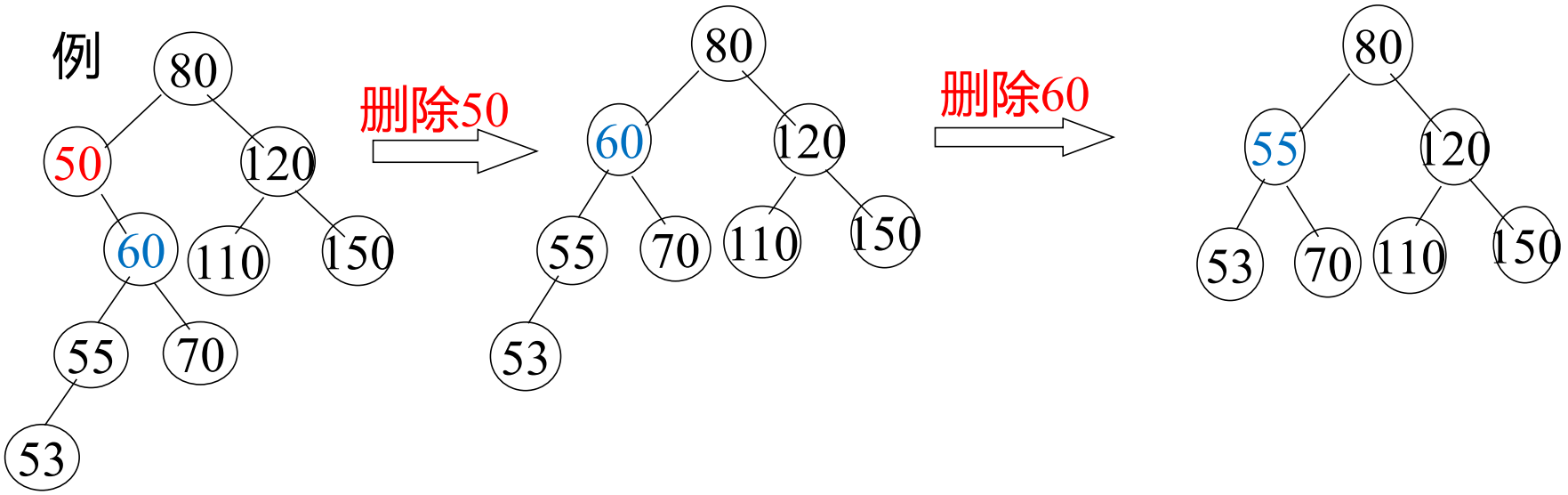
```
int DeletBST(BiTreep &rt,int key) //寻找删除节点
{   //若二叉排序树T中存在关键字等于key的数据元素时，则删除该数据元素节点，并返回TRUE，否则返回FALSE
    if(!rt) return FALSE;     //若是空树
    else    {
        if(EQ(key,rt->data)) return Delete(rt); //删除关键字等于key的数据元素
        else if( LT(key,rt->data) )  return DeletBST(rt->left,key);
        else return DeletBST(rt->right,key);
    }
}//DeleteBST
```

二叉排序树和平衡二叉树

```
void  Delete ( BiTree &p )
{ if ( ! p->rchild )    // 若右子树为空，则只需重接左子树，1，2.3，2.4
  { q = p;   p = p->lchild;   free(q); }
  else if ( ! p->lchild ) // 左子树为空，则只重接右子树，2.1，2.2
        {  q = p;   p = p->rchild;   free(q); }
        else     // 左右子树均不空
        {  q = p;   s = p->lchild;
           while ( s->rchild )   // 定位p左子树的最右结点
           { q = s; s =s->rchild;  }
           p->data = s->data;  // s覆盖原有的p
           if ( q!=p ) q->rchild = s->lchild;    //重接右子树
           else          q->lchild = s->lchild;    //重接左子树
           free(s);
        }
} //Delete  算法9.8
```

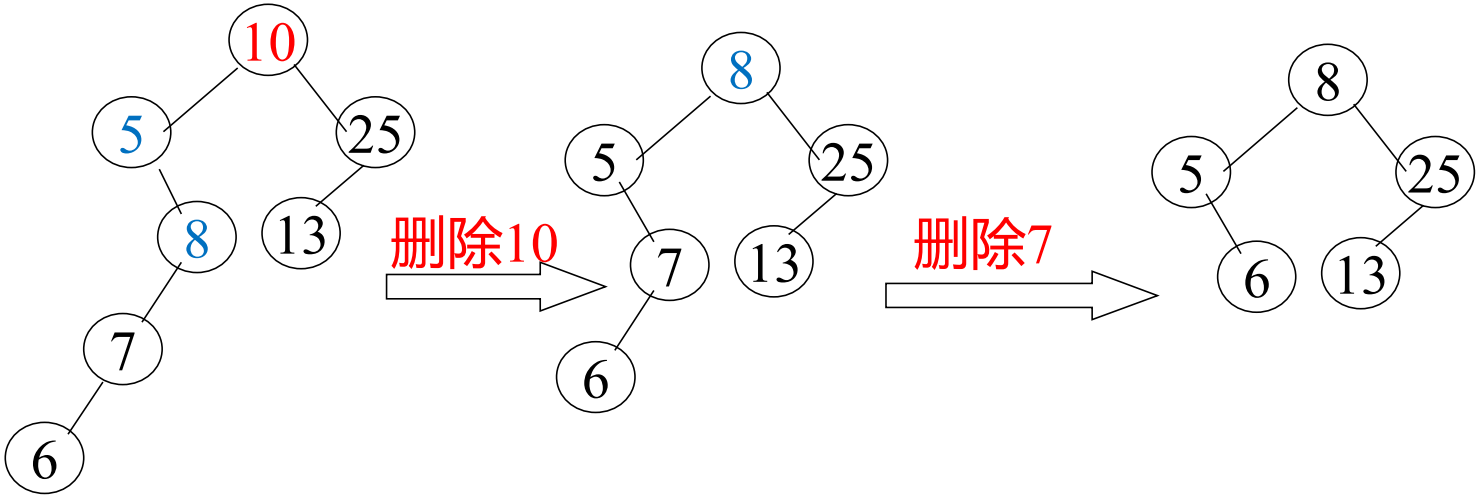
二叉排序树和平衡二叉树

二叉排序树的删除操作



二叉排序树和平衡二叉树

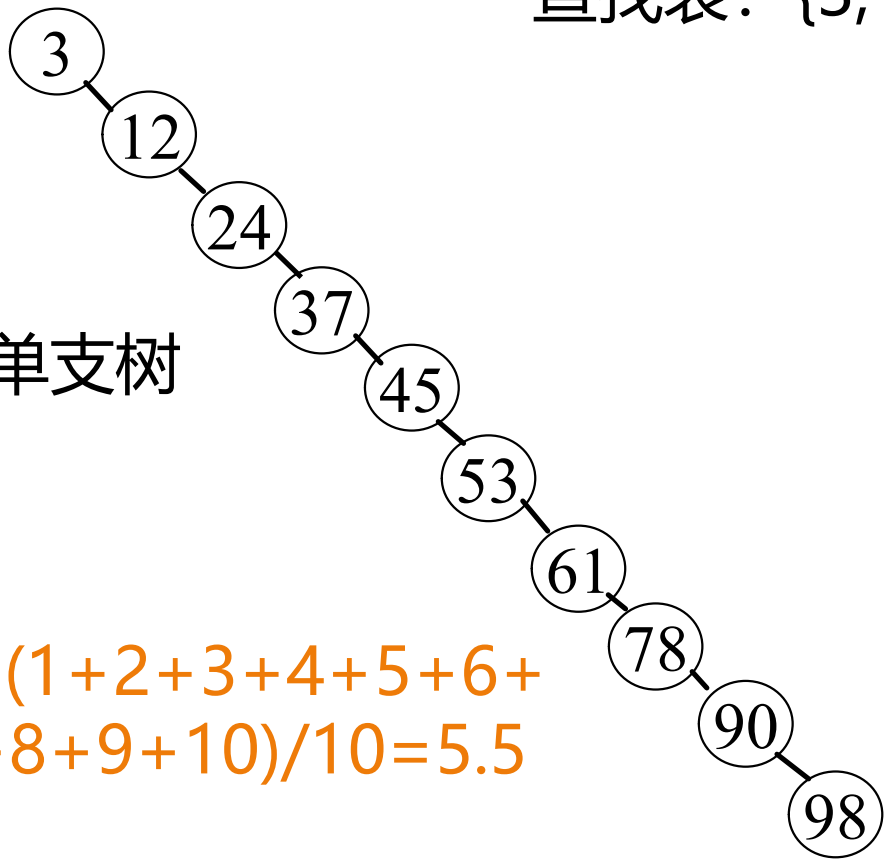
二叉排序树的删除操作



二叉排序树和平衡二叉树

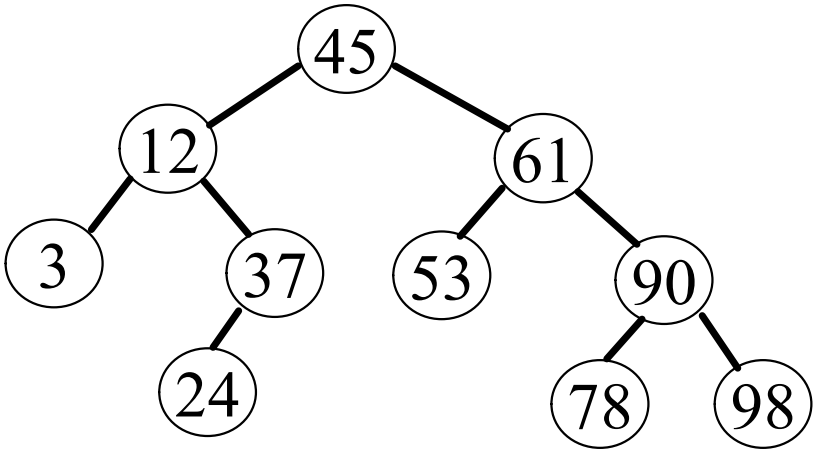
性能分析

查找表：{3,12, 24, 37, 45, 53, 61, 78, 90, 98}



单支树

ASL=(1+2+3+4+5+6+7+8+9+10)/10=5.5



ASL=(1+2×2+3×4+4×3)/10=2.9

二叉排序树和平衡二叉树

■二叉排序树的特点

含有n个结点的二叉排序树不唯一，与结点插入的顺序有直接关系。当查找失败后，在叶结点插入。

删除某个结点后，二叉排序树要重组。

没有对树的深度进行控制。

■二叉排序树的适用范围

用于组织规模较小的、内存中可以容纳的数据。对于数据量较大必须存放在外存中的数据，则无法快速处理。

二叉排序树和平衡二叉树

■平衡二叉排序树产生的原因

二叉排序树的特点之一（缺陷）：没有对树的深度进行控制。

■解决方法是平衡二叉树

结点的平衡因子：该结点左子树的高度减去右子树的高度。

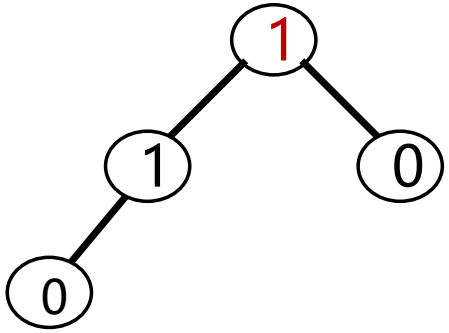
平衡二叉排序树：所有结点的平衡因子只可能是 -1、0、1 的二叉排序树，或者说每个结点的左、右子树高度差绝对值不超过1。

平衡二叉树是对二叉排序树的改进。

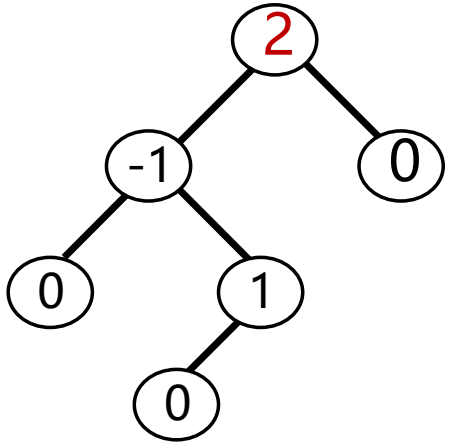
二叉排序树和平衡二叉树

平衡二叉树

所有结点的平衡因子只可能是 -1、0、1 的二叉排序树，或者说每个结点的左、右子树高度差绝对值不超过 1。



平衡二叉排序树



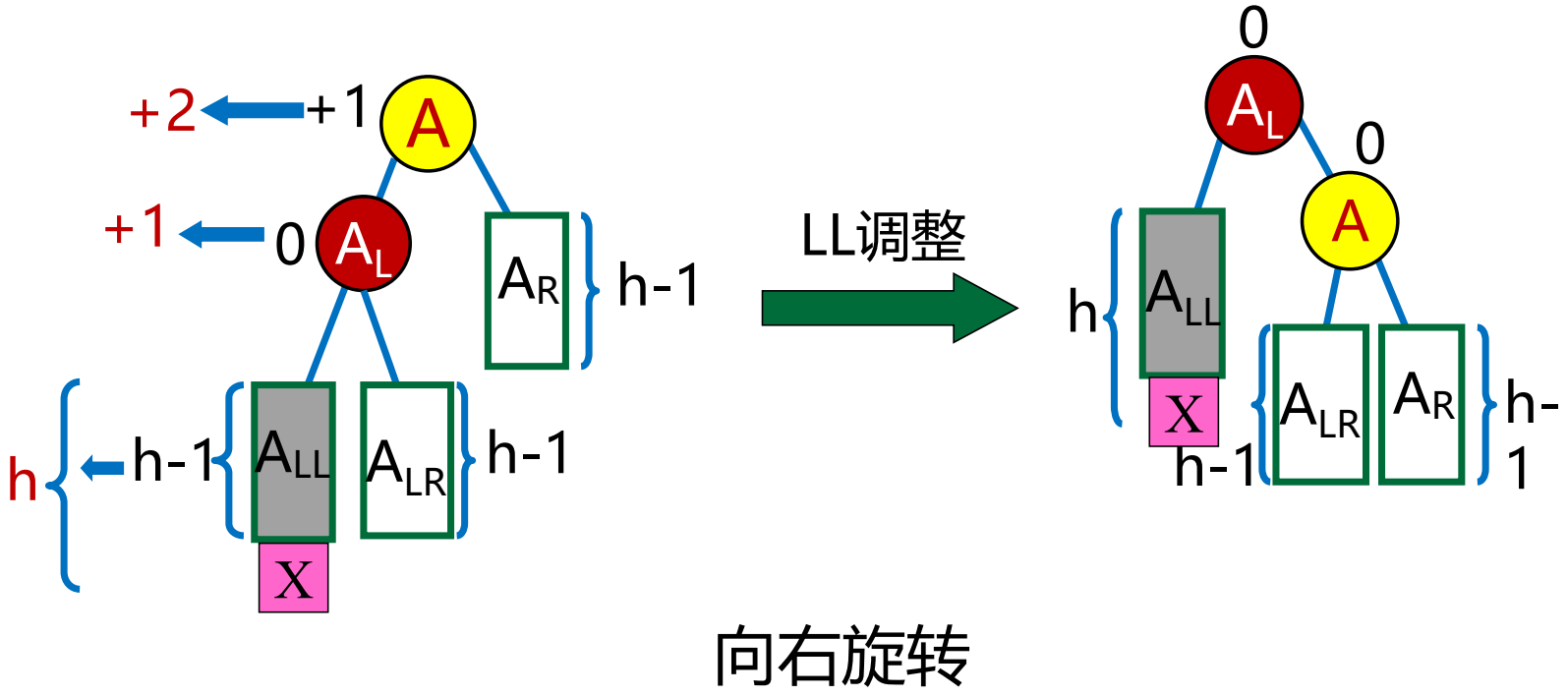
非平衡二叉排序树

二叉排序树和平衡二叉树

平衡二叉树——LL调整

新插入结点X，在 A 的 左子树的左子树 上。

A 的平衡因子由 1 增至 2，导致失衡。



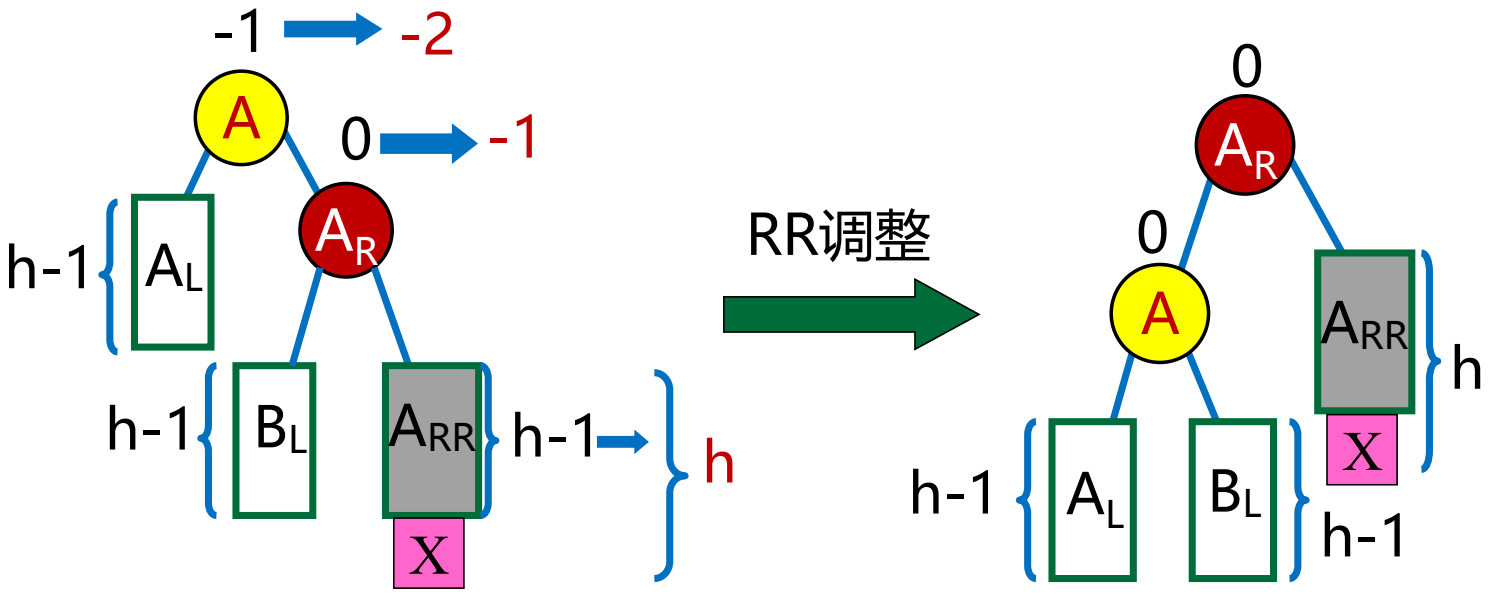
向右旋转

二叉排序树和平衡二叉树

平衡二叉树——RR调整

新插入结点X，在A的右子树的右子树上。

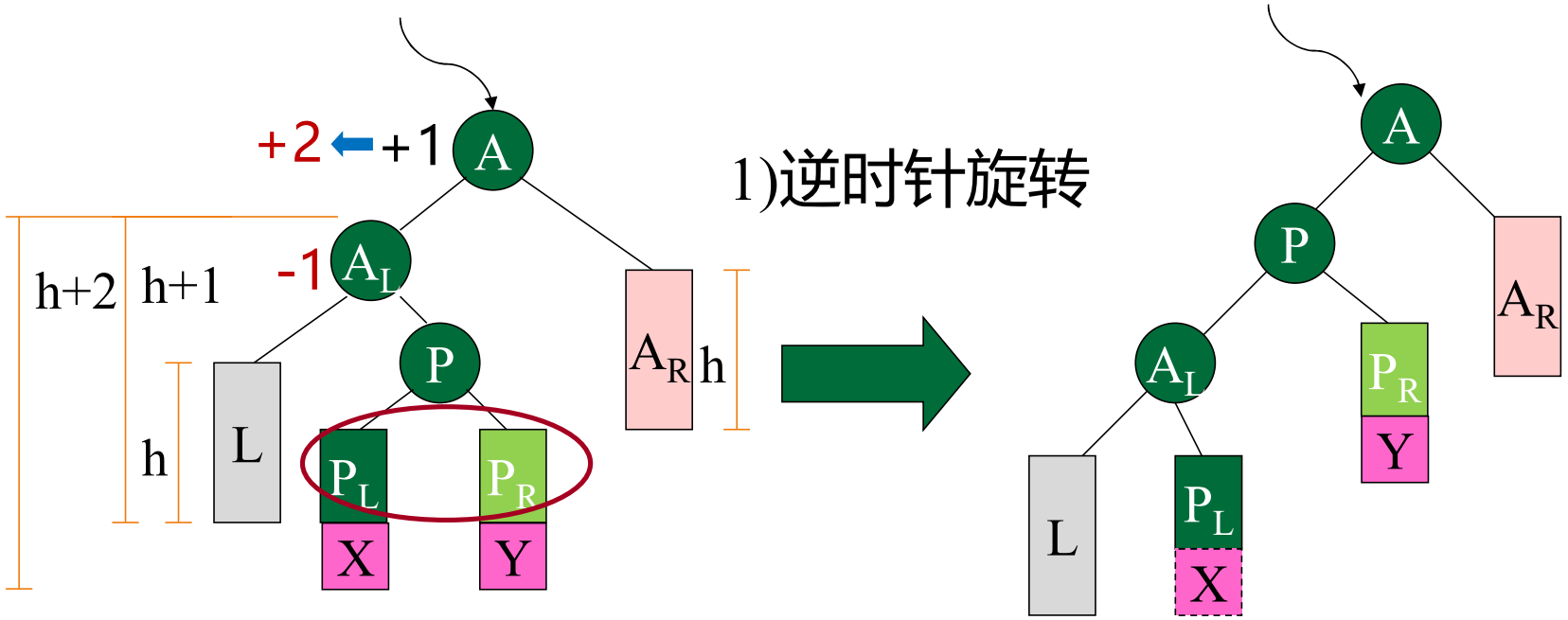
A的平衡因子由 -1 增至 -2，导致失衡。



向左旋转

二叉排序树和平衡二叉树

新插入结点X或Y，在A的左孩子(AL)的右子树(P)中；
造成以A为根的子树不平衡

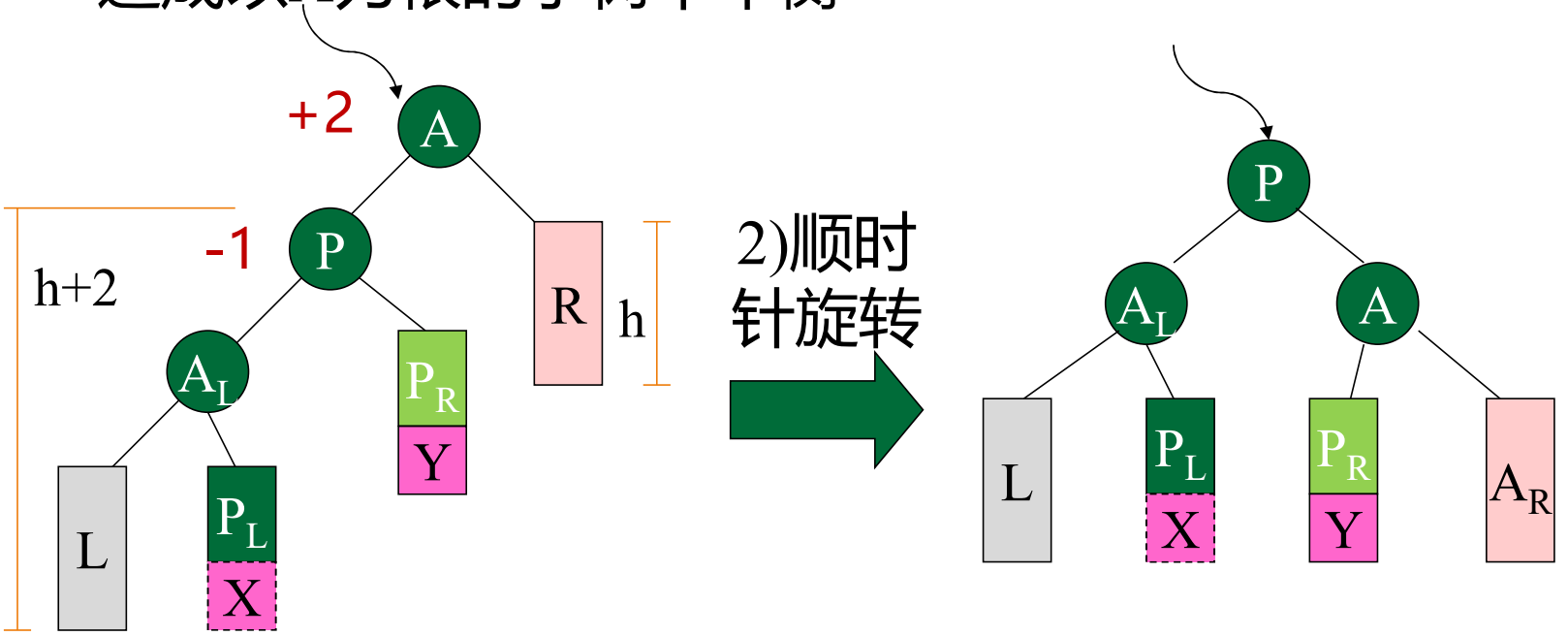


H(L) – H(R) = 2, 不平衡!

LR型：先向左旋转, 再向右旋转

二叉排序树和平衡二叉树

新插入结点X或Y，在A的左孩子(AL)的右子树(P)中；
造成以A为根的子树不平衡

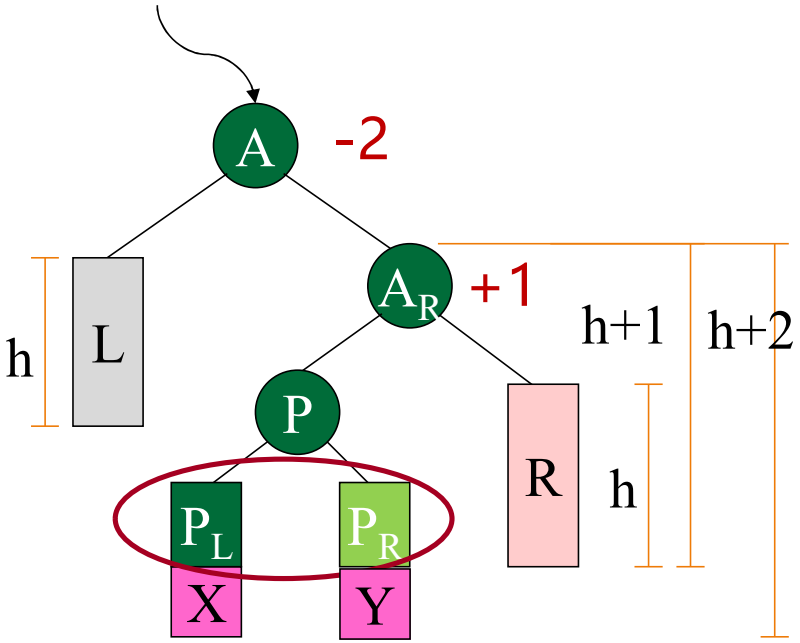


H(L) – H(R) = 2, LL型

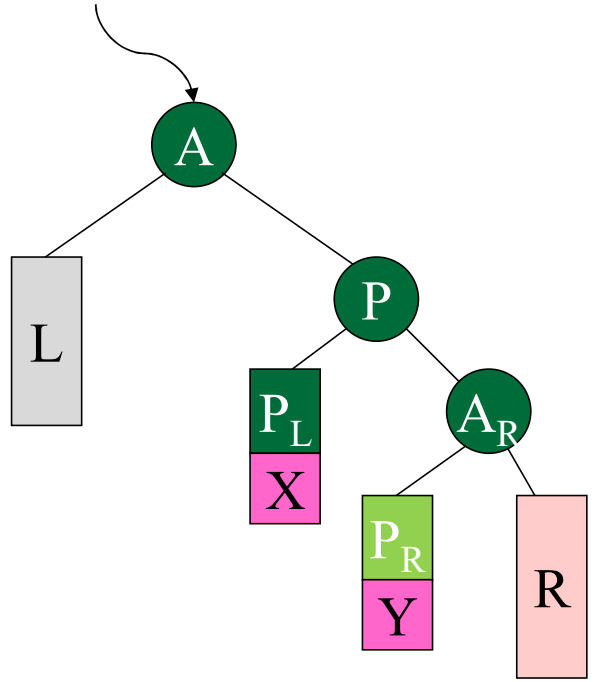
LR型：先向左旋转, 再向右旋转

二叉排序树和平衡二叉树

新插入结点X或Y，在A的右孩子(AR)的左子树(P)中；
造成以A为根的子树不平衡



1)顺时针旋转

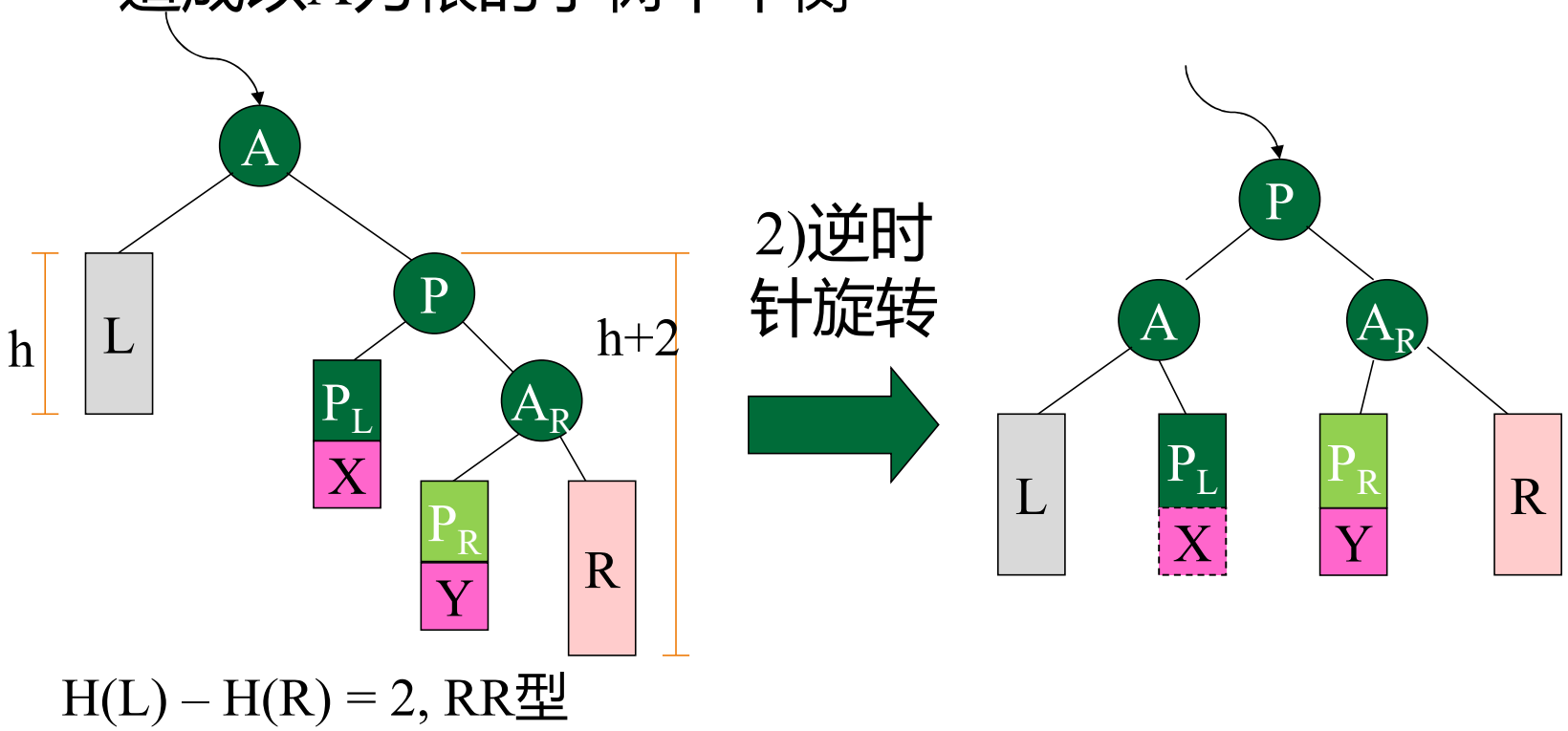


H(L) – H(R) = -2, 不平衡!

RL型先向右旋转, 再向左旋转

二叉排序树和平衡二叉树

新插入结点X或Y，在A的右孩子(AR)的左子树(P)中；
造成以A为根的子树不平衡



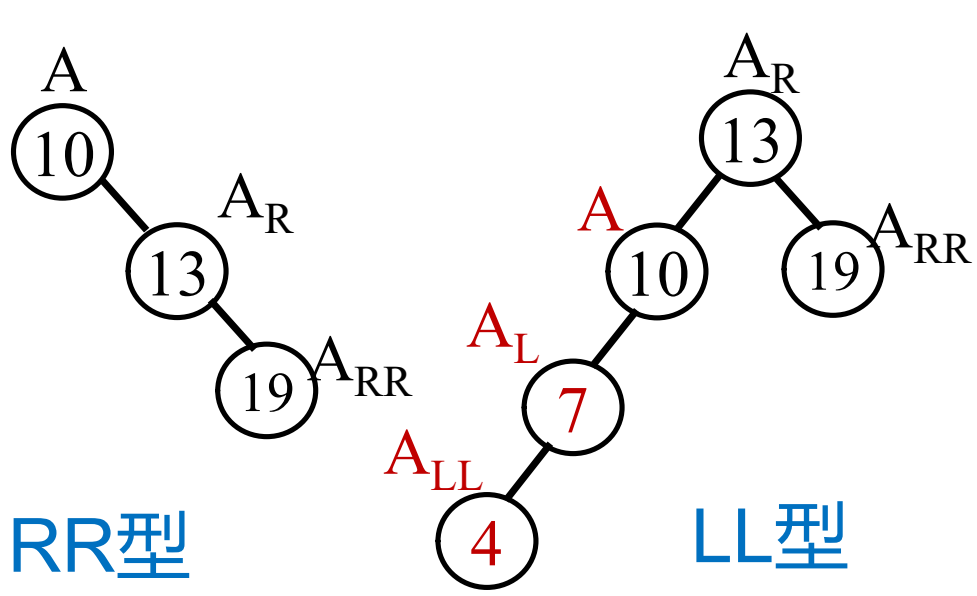
H(L) – H(R) = 2, RR型

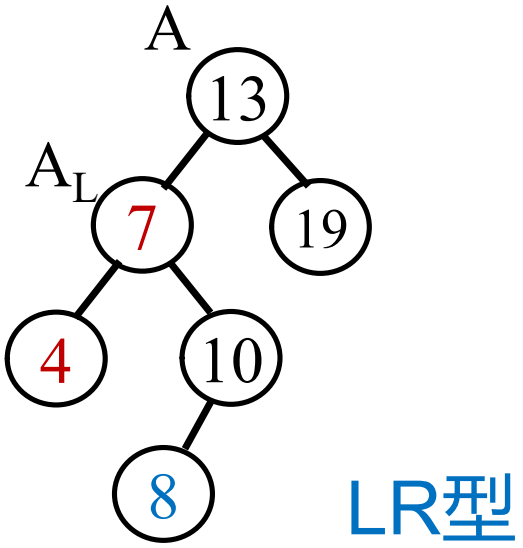
RL型先向右旋转, 再向左旋转

二叉排序树和平衡二叉树

- 例题
  按照如下顺序建立平衡二叉树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

插入10、13、19、7、4、8

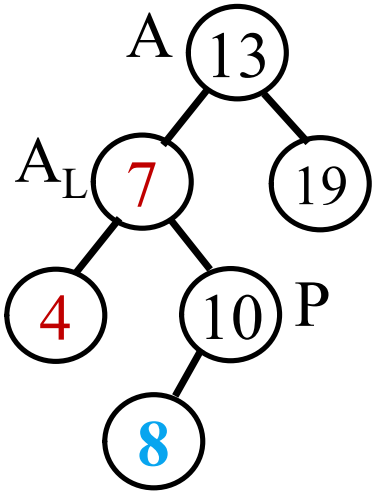




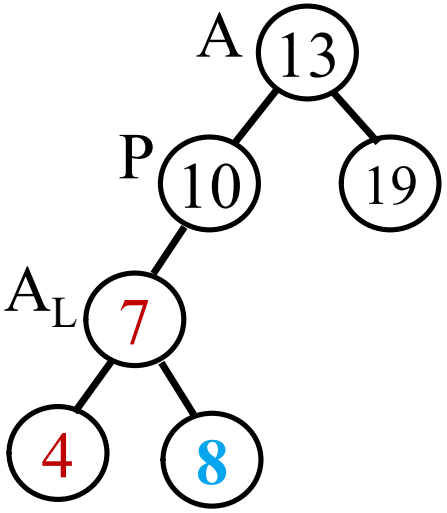
二叉排序树和平衡二叉树

- 例题
  按照如下顺序建立平衡二叉树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

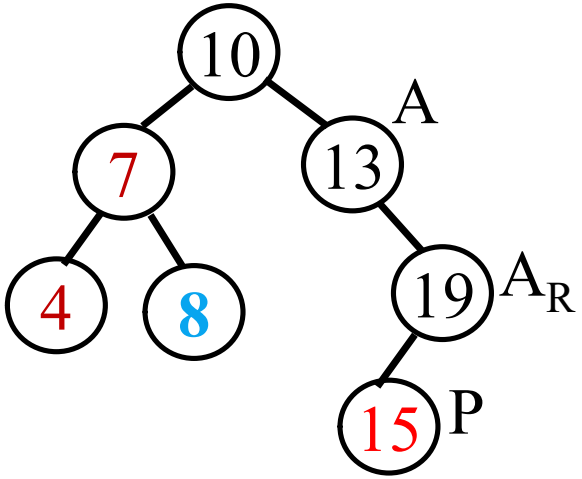
插入10、13、19 、7、4 、8 、15



LR型



先左旋

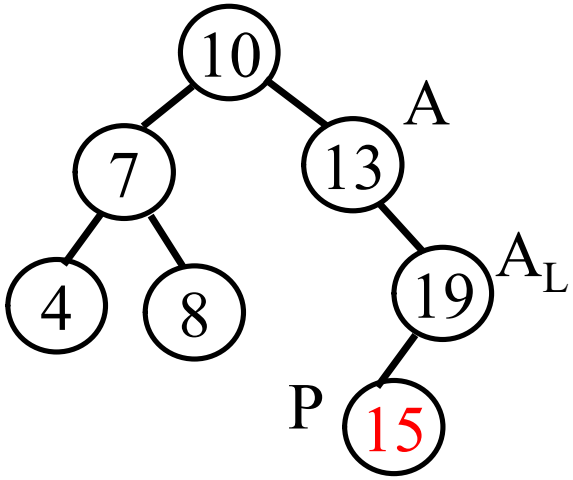


再右转　RL型

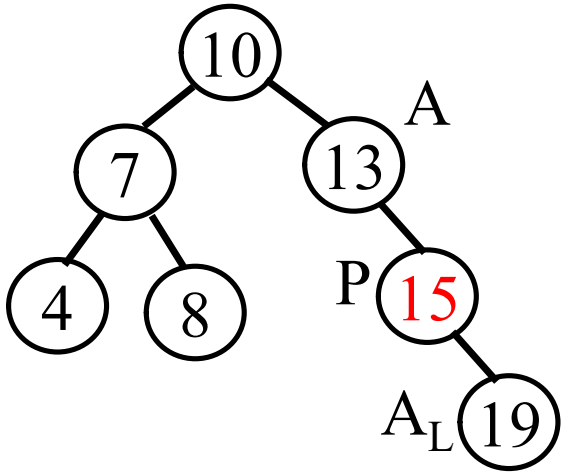
二叉排序树和平衡二叉树

例题
按照如下顺序建立平衡二叉树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

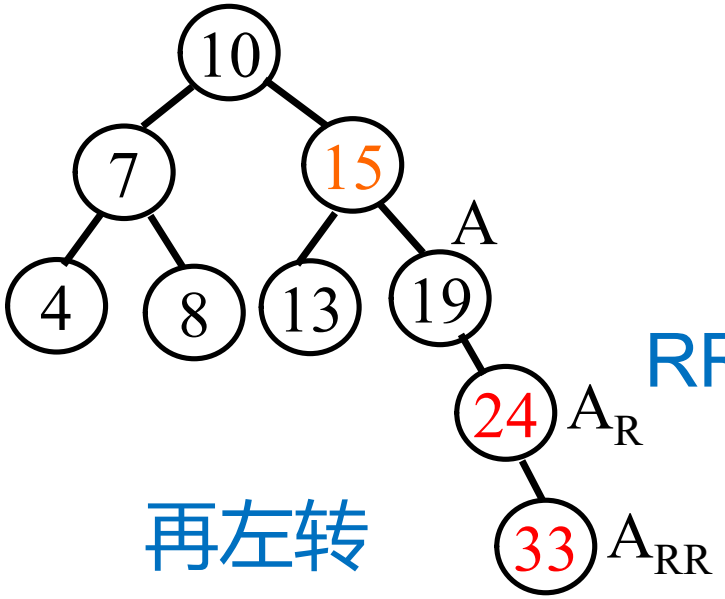
插入10、13、19 、7、4、8 、15 、24、33



RL型



先右旋



再左转

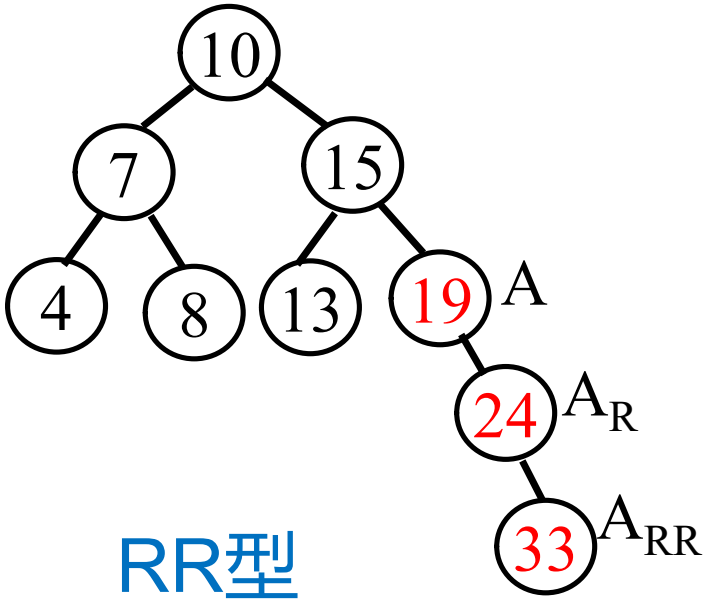
RR型

二叉排序树和平衡二叉树

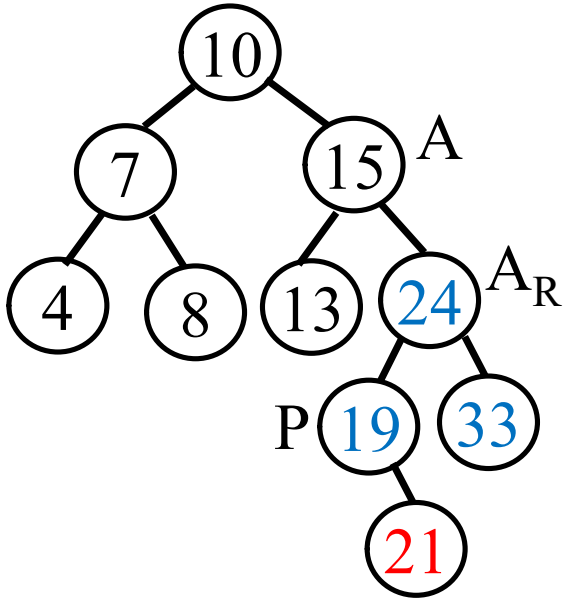
例题

按照如下顺序建立平衡二叉树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

插入10、13、19 、7、4 、8 、15 、24、33 、21



RR型

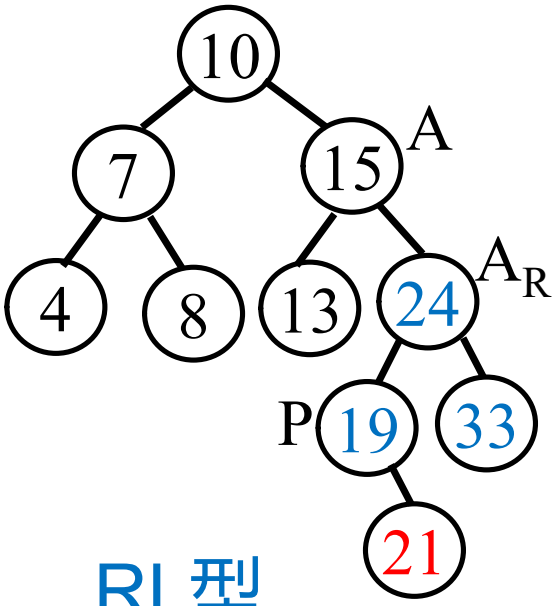


RL型

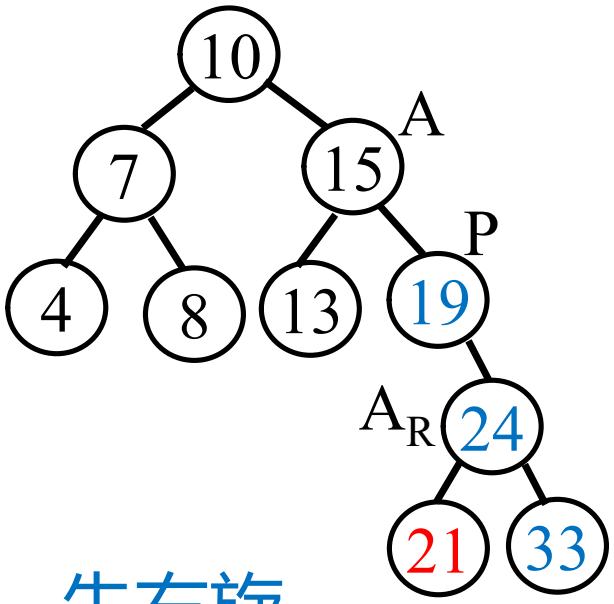
二叉排序树和平衡二叉树

例题
按照如下顺序建立平衡二叉树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

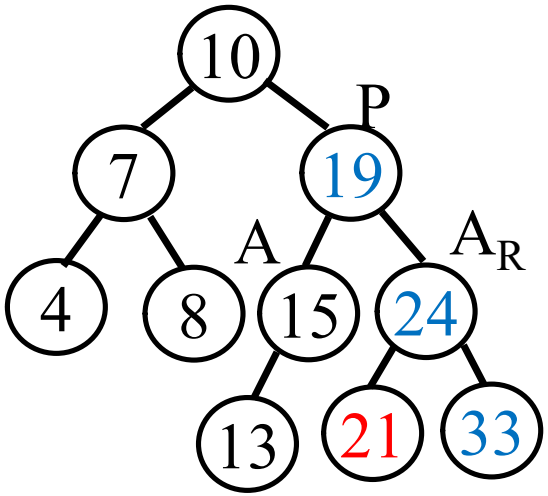
插入10、13、19 、7、4 、8 、15 、24、33 、21



RL型



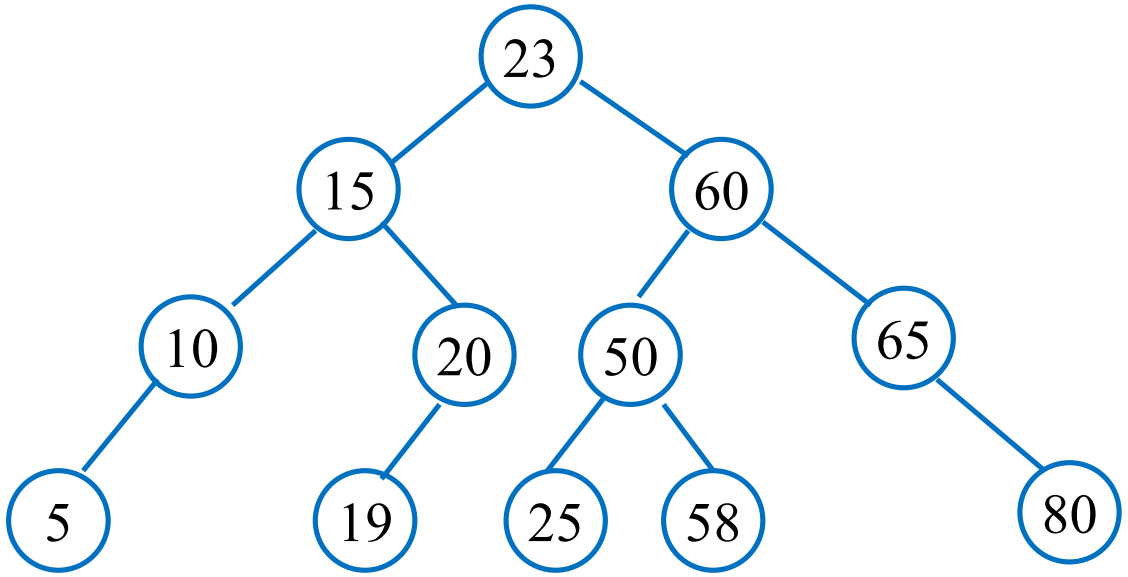
先右旋



再左转

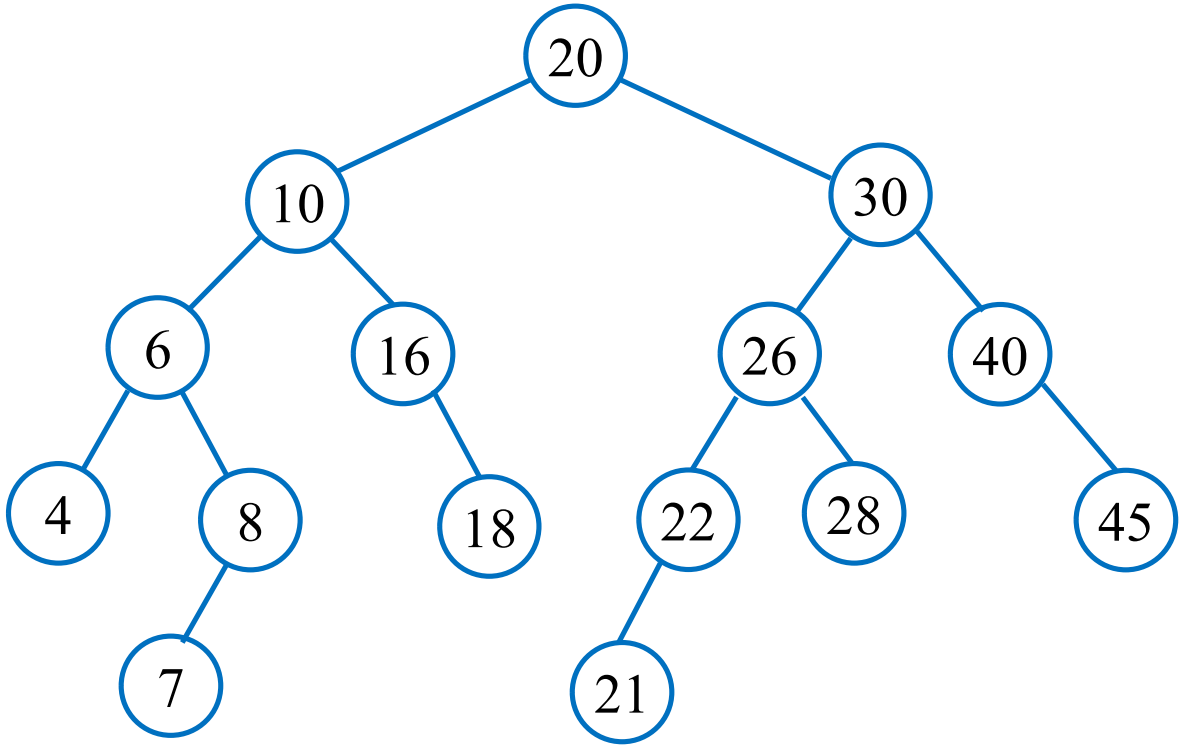
二叉排序树和平衡二叉树

按照如下顺序建立平衡二叉树：
50，23，60，15，25，65，10，20，5，80，19，58



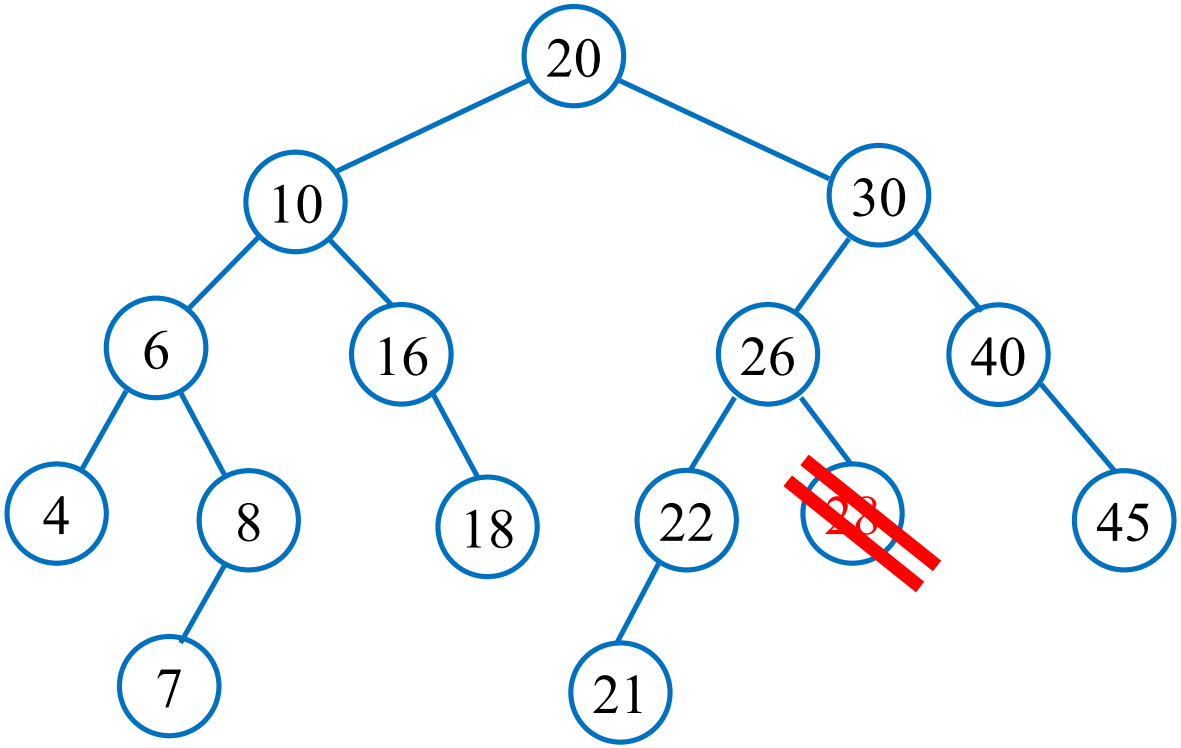
二叉排序树和平衡二叉树

按照如下顺序删除平衡二叉树中的结点：28，16，30，21，22。



二叉排序树和平衡二叉树

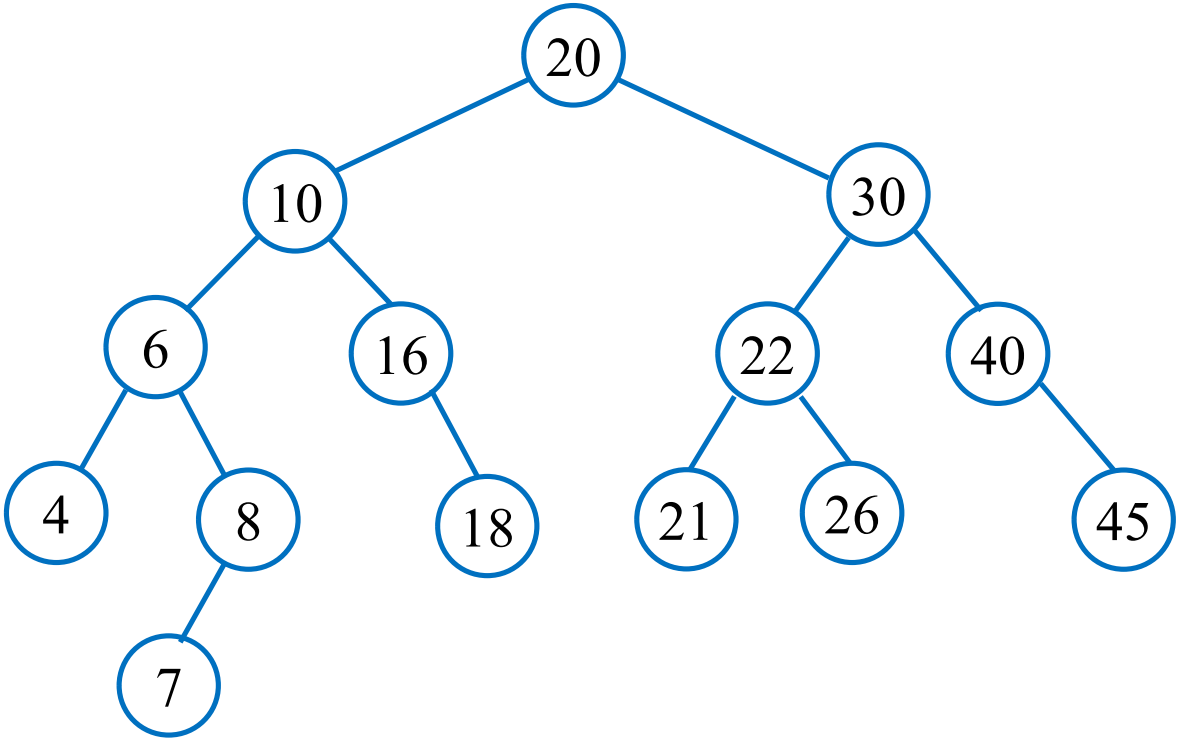
按照如下顺序删除平衡二叉树中的结点：28，16，30，21，22。



二叉排序树和平衡二叉树

删除：28，16，30，21，22。

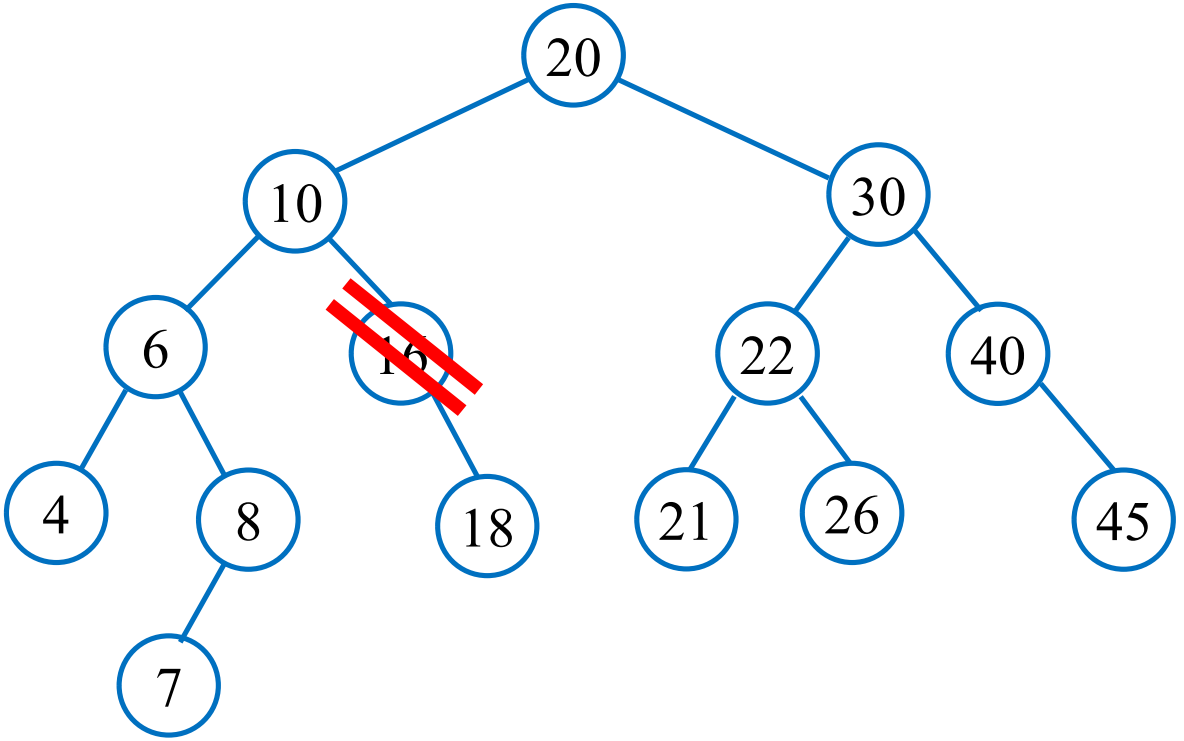
对26进行LL调整



二叉排序树和平衡二叉树

删除：28，16，30，21，22。

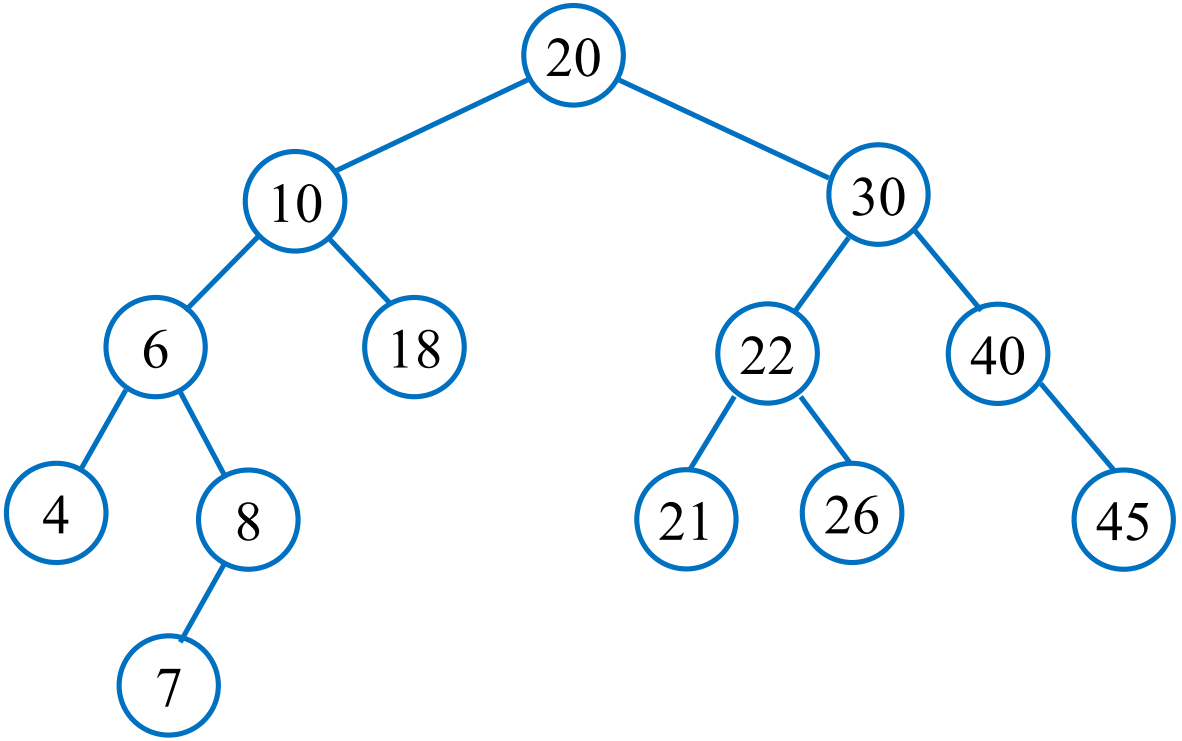
对26进行LL调整



二叉排序树和平衡二叉树

删除：28，16，30，21，22。

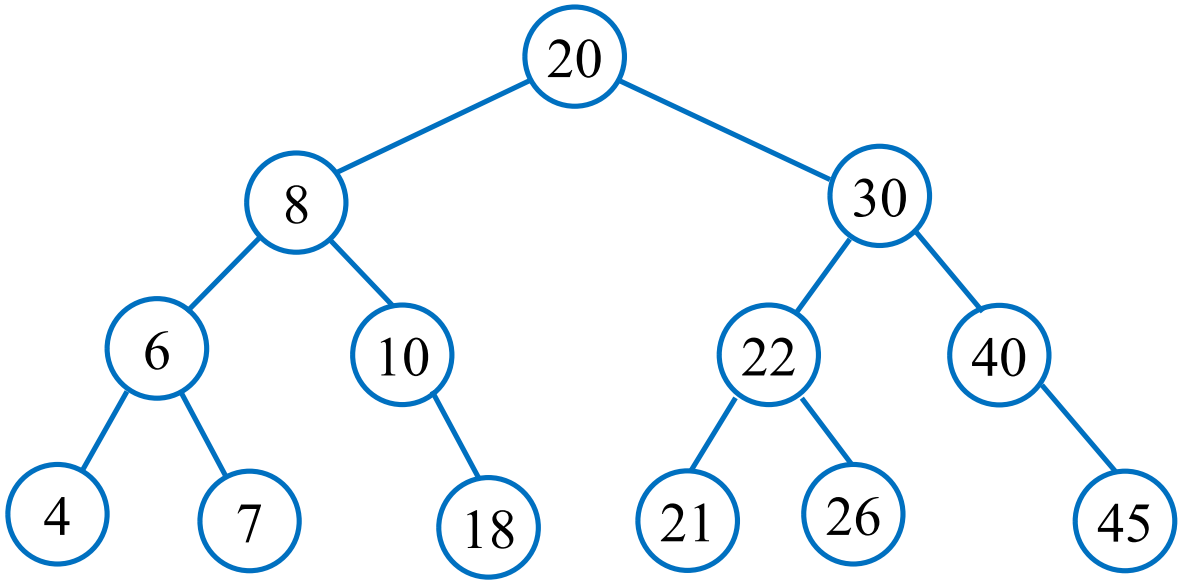
删掉16，保持二叉排序树



二叉排序树和平衡二叉树

删除：28，16，30，21，22。

对10进行LR调整

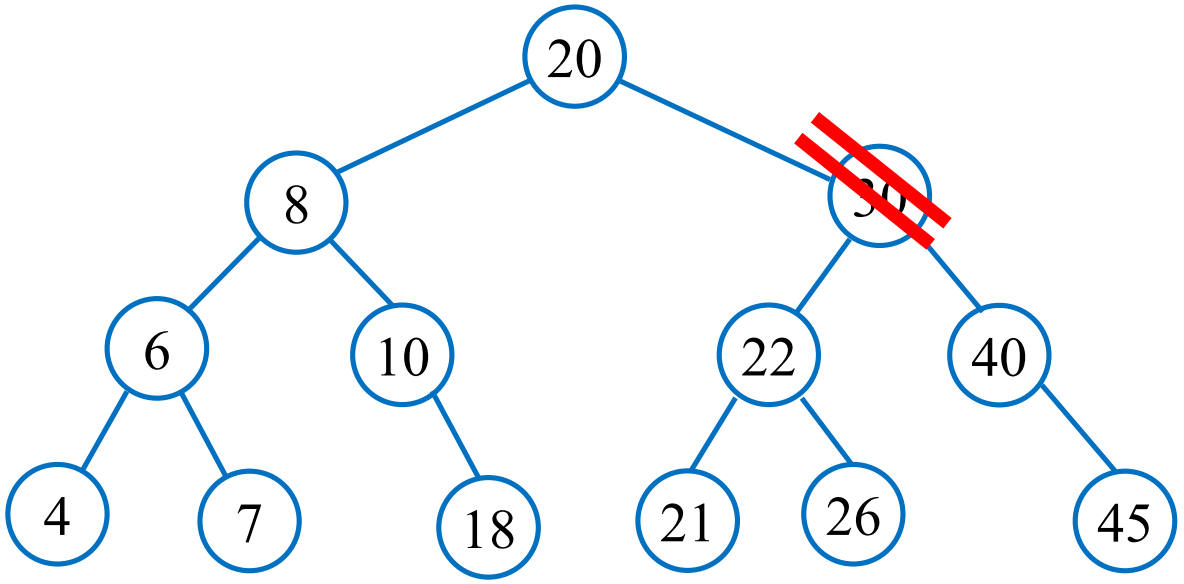


二叉排序树和平衡二叉树

删除：28，16，30，21，22。

用30的中序前驱26替换30；

并将原来的26节点删除

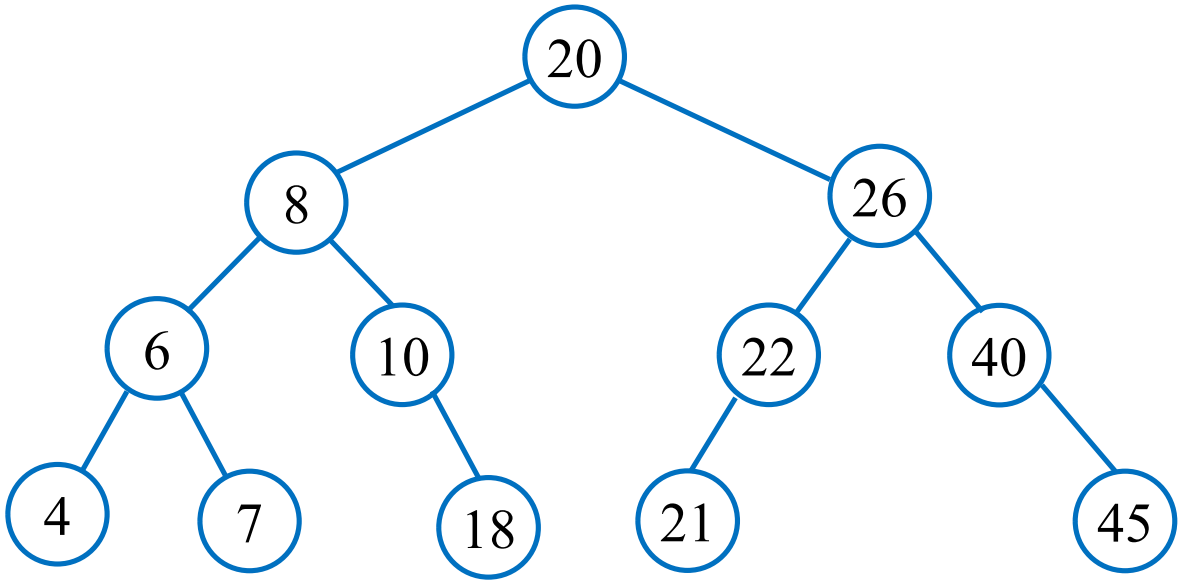


二叉排序树和平衡二叉树

删除：28，16，30，21，22。

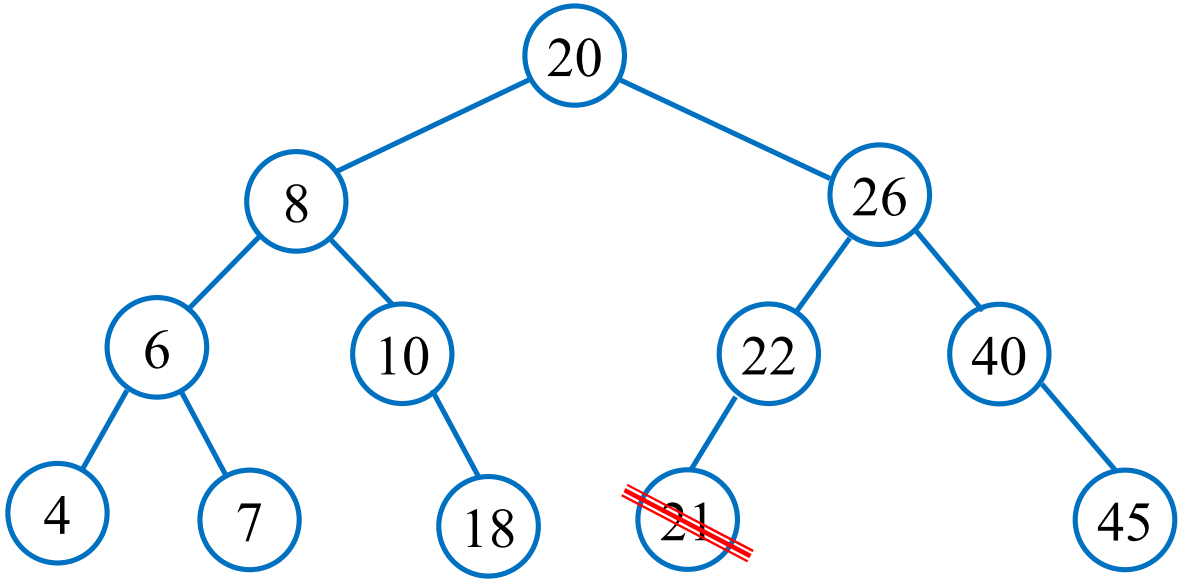
用30的中序前驱26替换30；

并将原来的26节点删除



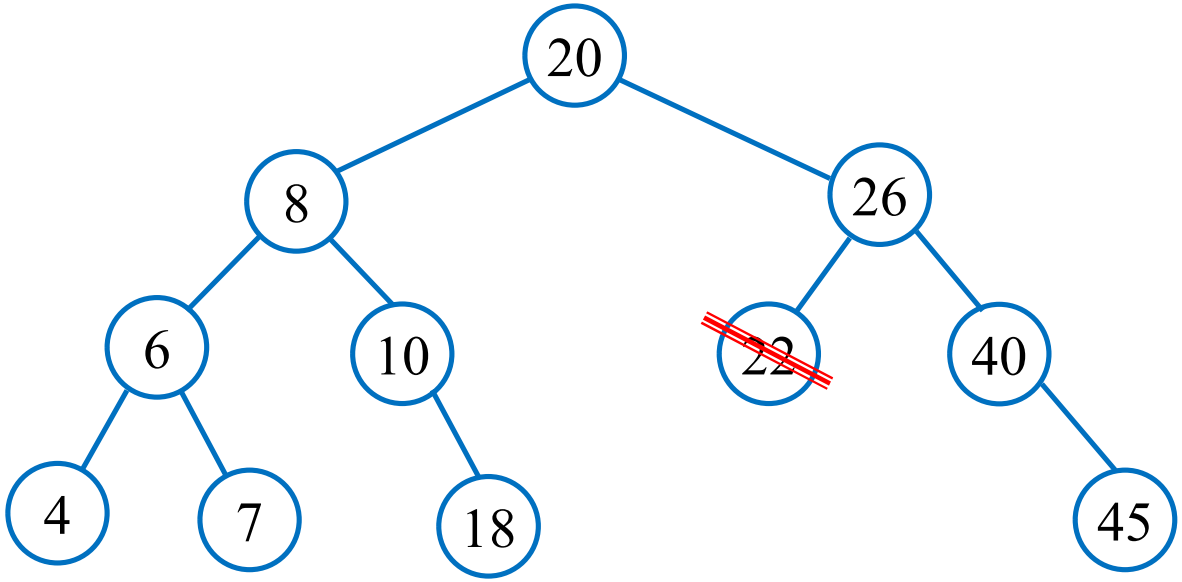
二叉排序树和平衡二叉树

删除：28，16，30，21，22。



二叉排序树和平衡二叉树

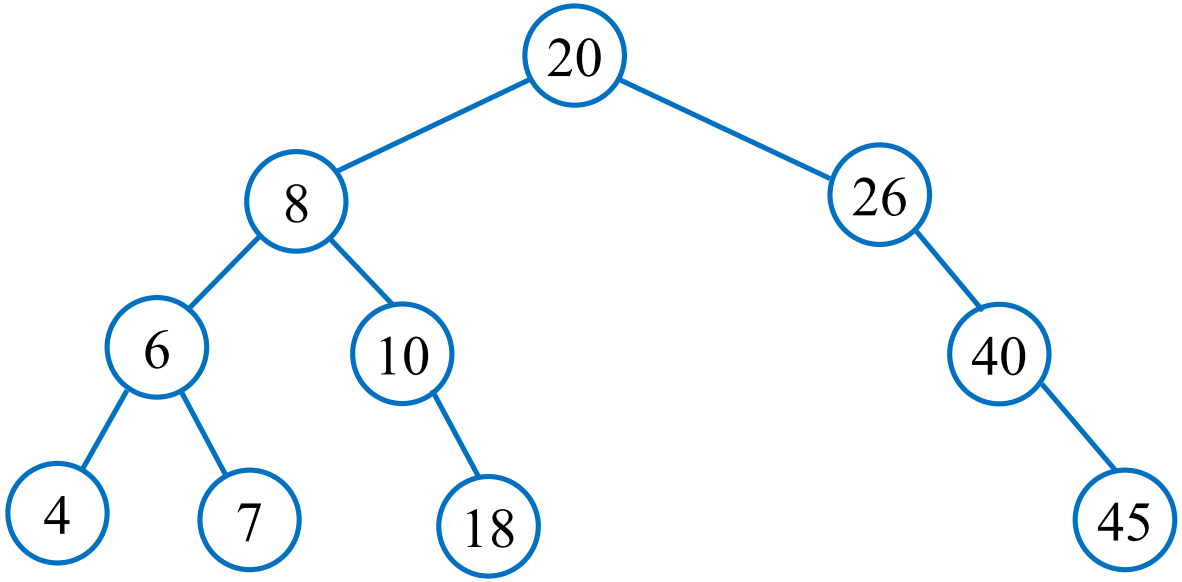
删除：28，16，30，21，22。



二叉排序树和平衡二叉树

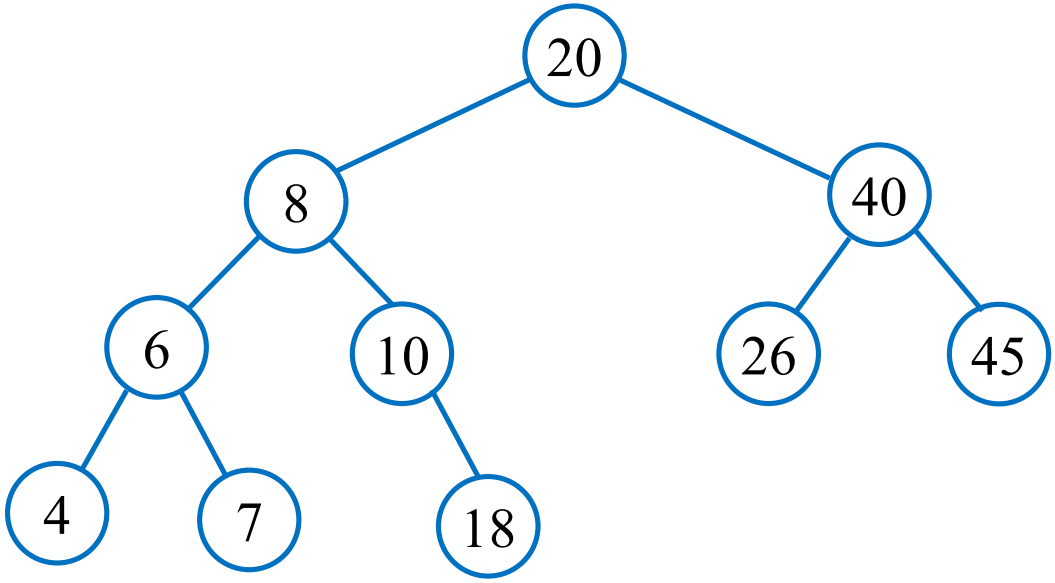
删除：28，16，30，21，22。

再对26进行RR调整



二叉排序树和平衡二叉树

按照如下顺序删除平衡二叉树中的结点：28，16，30，21，22。
最后结果如下。



二叉排序树和平衡二叉树

在平衡二叉树上查找的算法与排序二叉树相同
查找过程中的比较次数不超过树的深度
所以在平衡二叉树上查找的时间复杂度为$O(\log_2 n)$

二叉排序树和平衡二叉树

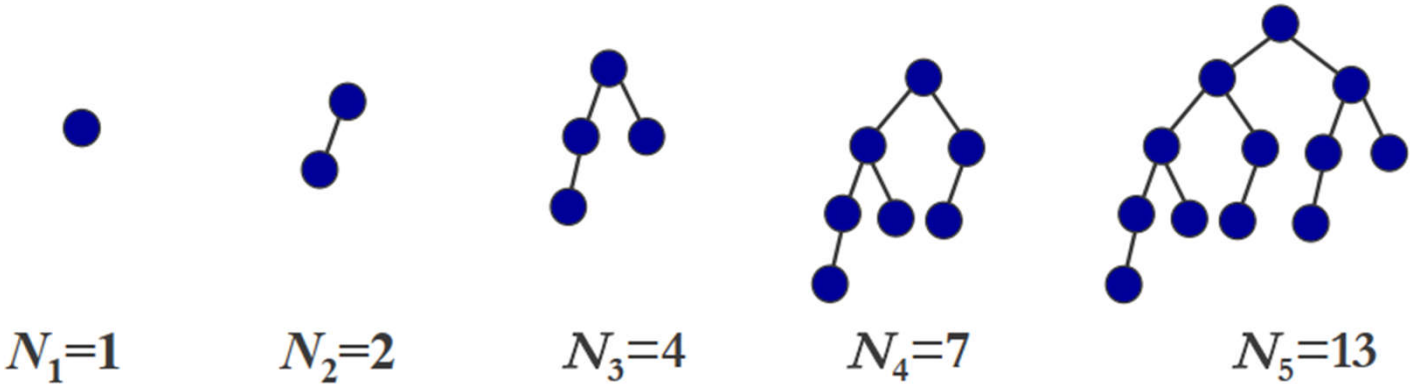
AVL树的高度
设在新结点插入前**AVL**树的高度为***h***， 结点个数为***n***， 则插入一个新结点，其时间代价是**O(*h*)**。

对于有**n**个结点的**AVL**树来说， ***h***多大?
高度为 ***h*** 的**AVL**树最多结点数为$2^h - 1$， 即满二叉树情形。

设 N_h是高度为 ***h*** 的**AVL**树的最少结点个数。

二叉排序树和平衡二叉树

设 N_h是高度为 h 的**AVL**树的最少结点个数。
例如：具有 5 层结点的AVL树至少有多少个结点?
解： 左右子树高度相差1的树是平衡的。



$N_1 = 1$， $N_2 = N_1+N_0+1 = 2$
$N_3 = N_2+N_1+1 = 4$， $N_4 = N_3+N_2+1 = 7$
$N_h = N_{h-1} + N_{h-2} +1$, $h > 1$

二叉排序树和平衡二叉树

AVL结点个数:

$N_1 = 1$，$N_2 = N_1+N_0+1 = 2$

$N_3 = N_2+N_1+1 = 4$，$N_4 = N_3+N_2+1 = 7$

$N_h = N_{h-1} + N_{h-2} +1$, $h > 1$

$F_0 = 0$，F1=1 ，$F_2 = F_1+F_0 = 1$ ，$F_3 = F_2+F_1 = 2$

$F_n = F_{n-1} + F_{n-2}$

可得：$N_h = F_{h+2} - 1$

二叉排序树和平衡二叉树

$$F_{h+2} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+2}\right) > \left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 1$$

$$N_h > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 2 \qquad \phi = \frac{1+\sqrt{5}}{2}$$

$$\phi^{h+2} < \sqrt{5}(N_h + 2) \qquad h+2 < \log_\phi\sqrt{5} + \log_\phi(N_h+2)$$

$$\log_\phi X = \log_2 X / \log_2\phi \qquad \log_2\phi = 0.694$$

$$h+2 < \log_2\sqrt{5}/\log_2\phi + \log_2(N_h+2)/\log_2\phi$$

$$h+2 < 1.6723 + 1.4404 * \log_2(N_h+2)$$

$$h_{max} = \lceil 1.44 log_2(n+1) - 0.327 \rceil$$

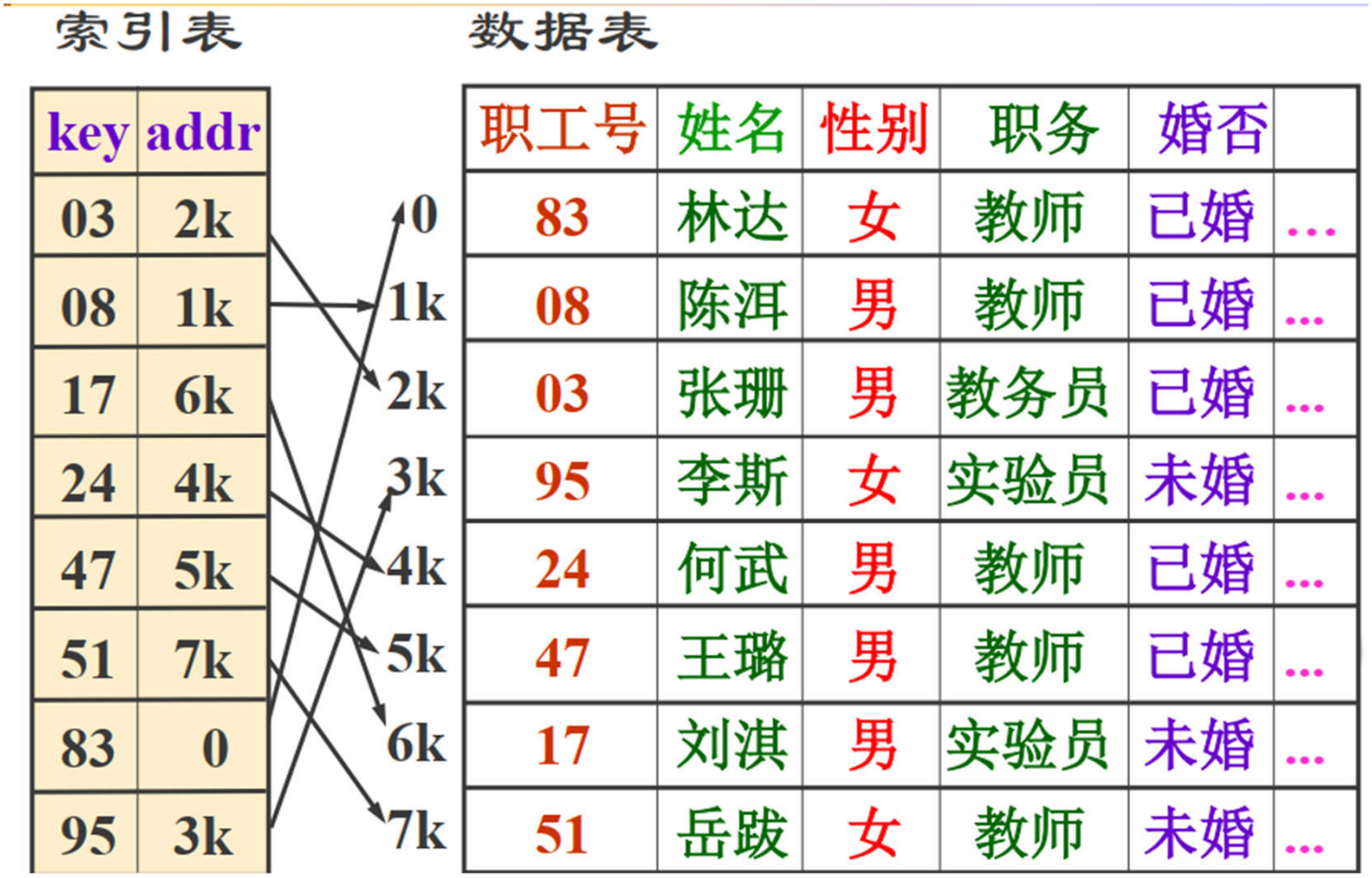
二叉排序树和平衡二叉树

若AVL树有n个结点，则：

最小高度：$\lceil \log_2(n+1) \rceil$

最大高度：$[1.44log_2(n+1) - 0.327]$

索引顺序表与分块查找


索引表
数据表
key addr
03 2k
08 1k
17 6k
24 4k
47 5k
51 7k
83 0
95 3k
0
1k
2k
3k
4k
5k
6k
7k
职工号 姓名 性别 职务 婚否
83 林达 女 教师 已婚 ...
08 陈洱 男 教师 已婚 ...
03 张珊 男 教务员 已婚 ...
95 李斯 女 实验员 未婚 ...
24 何武 男 教师 已婚 ...
47 王璐 男 教师 已婚 ...
17 刘淇 男 实验员 未婚 ...
51 岳跋 女 教师 未婚 ...

索引顺序表与分块查找

当数据元素个数很多， n 很大时，可采用索引方法来实现存储和查找。
若数据不能一次读入内存，那么无论是顺序查找或折半查找，都需要多次读取外存记录。

线性索引 (Linear Index List)
查询时需要从外存中把索引表读入内存，经过查找索引后确定了数据元素的存储地址，再经过 1 次读取元素操作就可以完成查找。只需 2 次读盘即可。
采用索引结构可以加速查找速度

索引顺序表与分块查找

索引可分为 2 种：

1. 稠密索引：一个索引项对应数据表中一个元素。

当元素在外存中按加入顺序存放而不是按关键码值有序存放时必须采用稠密索引，这时的索引结构叫做索引非顺序结构。

2. 稀疏索引：当元素在外存中有序存放时，可以把所有 n 个元素分为 b 个子表存放，一个索引项对应数据表中一组元素（子表）。第 i个索引项是第 i 个子表的索引项, i = 0, 1, ..., n-1。这种索引结构叫做索引顺序结构。

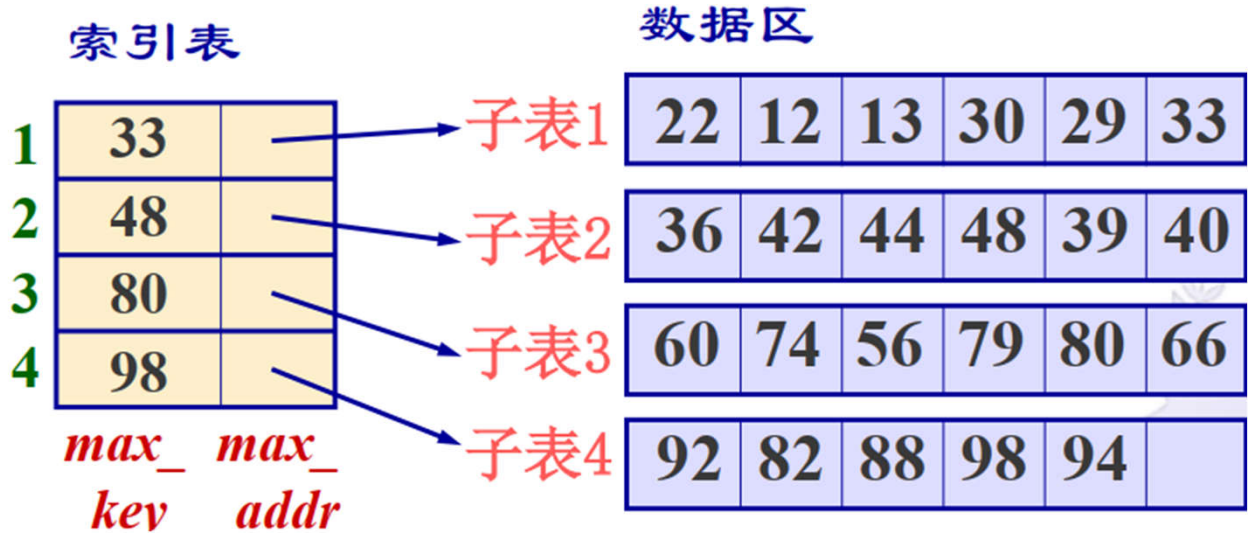
索引顺序表与分块查找

对索引顺序结构进行查找时，采用分块查找：

1. 先在索引表 ID 中查找给定值 K, 确定满足

ID[i-1].max_key < K ≤ ID[i].max_key

2. 再在第i个子表中按给定值查找要求的元素



索引顺序表与分块查找

索引顺序查找的查找成功时的平均查找长度：

$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$

ASL_{Index} 是在索引表中查找子表位置的平均查找长度.

$ASL_{SubList}$ 是在子表内查找元素位置的查找成功的平均查找长度。

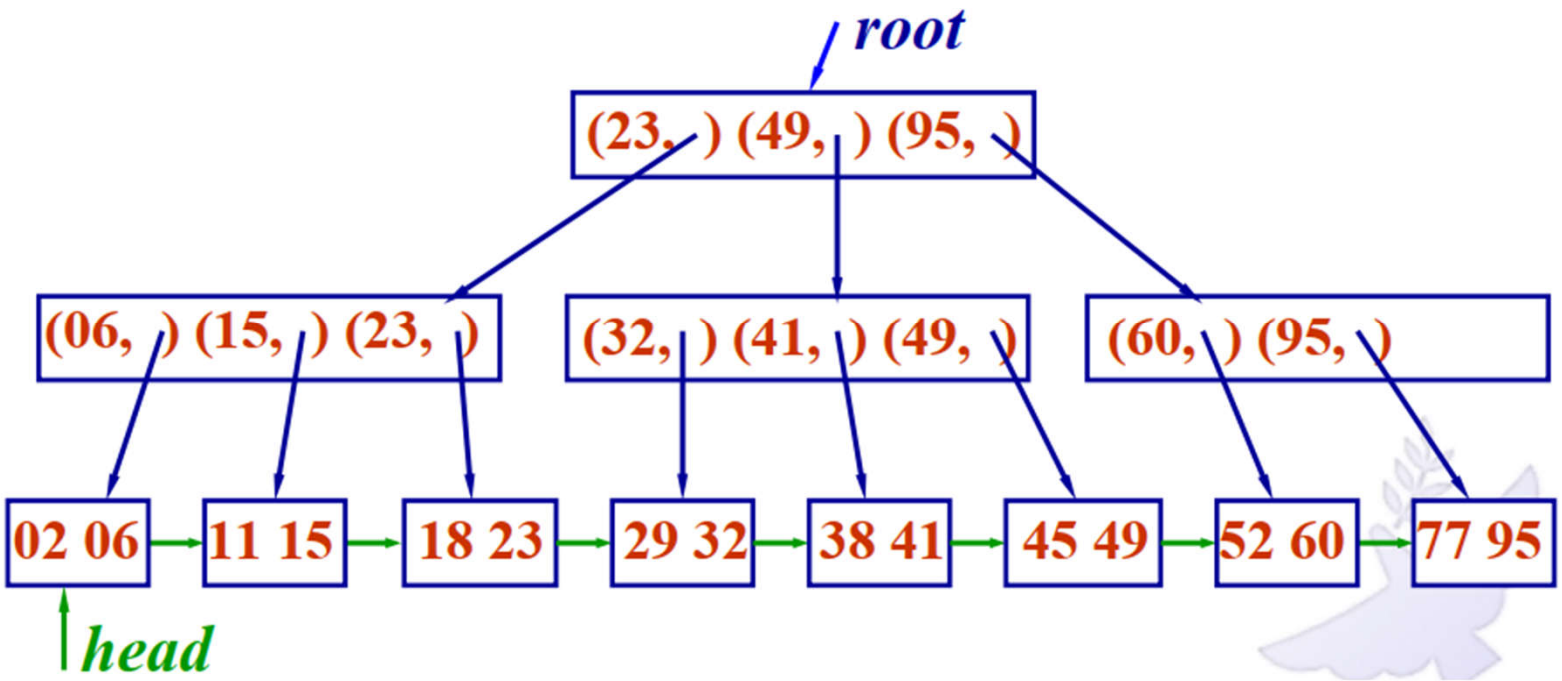
索引顺序表与分块查找

当数据元素数目特别大，索引表本身很大，在内存中放不下，需要分批多次读取外存才能把索引表查找一遍。

此时，可以建立索引的索引（二级索引）。二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。
如果二级索引在内存中也放不下，需要分为许多块多次从外存读入。可以建立二级索引的索引（三级索引）。这时，访问外存次数等于读入索引次数再加上 1 次读取元素。

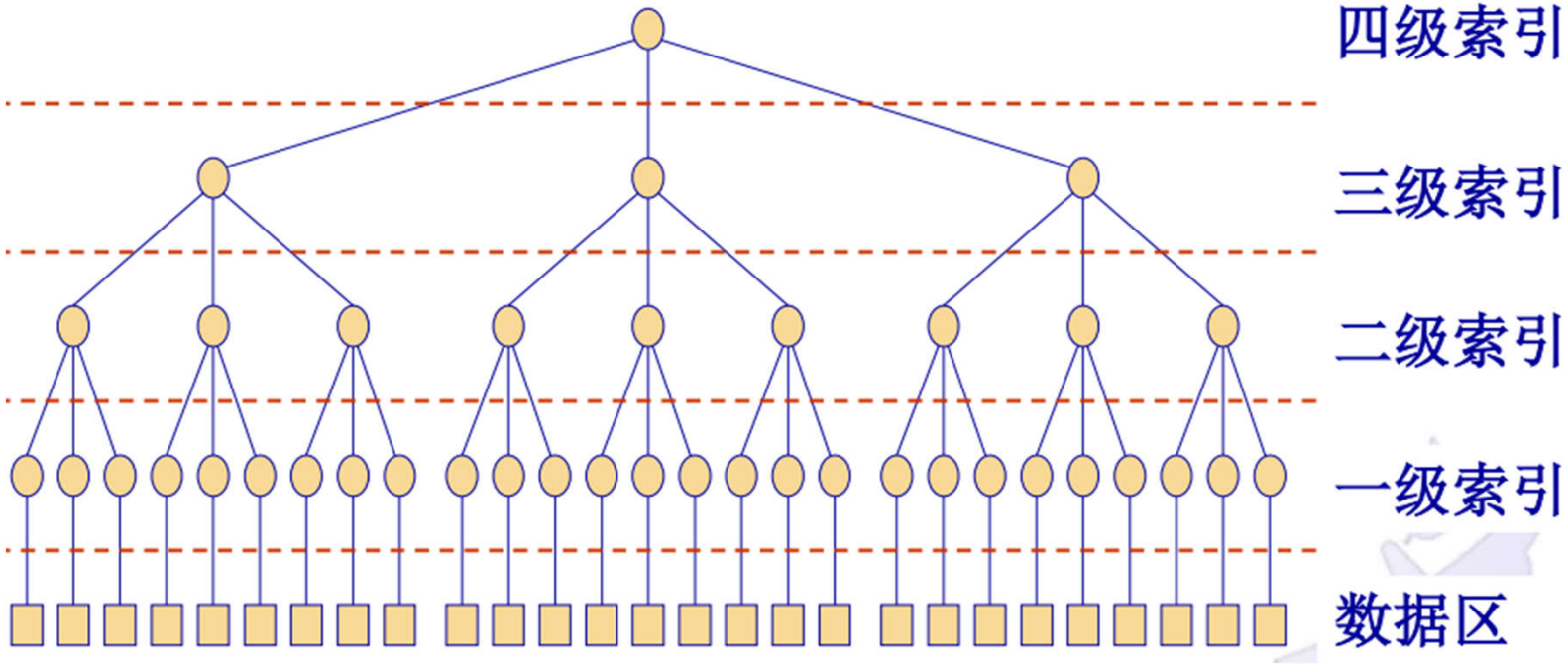
索引顺序表与分块查找

树中每一个分支结点表示一个索引块, 每个索引项给出各子树结点的最大关键码和结点地址。



索引顺序表与分块查找

- m 叉查找树：树的叶结点中各索引项给出在数据表中存放的元素的关键码和存放地址



B-树和B+树

B-树产生的原因

二叉排序树的适用范围（缺陷）：用于组织规模较小的、内存中可以容纳的数据。

若以结点作为内、外存交换的单位，则在按关键字查找一个结点时，平均要访问外存$\log_2 n$。

由于外存不仅存取速度比内存慢得多，而且每次访问首先要花费大量的时间找到数据存放得位置（称为定位）。

因此，对于数据量较大必须存放在外存中的数据，则无法快速处理。

要提高查找效率，关键是减少外存交换的次数（称为访外的次数）。

B-树和B+树

B-树： Balanced Tree

B-树在Rudolf Bayer, Edward M. McCreight(1970)的论文《Organization and Maintenance of Large Ordered Indices》中提出的。

B-树: 平衡多路查找树

B-树和B+树

B-树的基本思想：

内、外存交换采用分块技术。

外存分为若干固定大小的块，称为页块或物理块。

另外开辟一个或多个缓冲区，每个缓冲区的大小与外存的一个页块相同，每次内、外存交换以整个页块为单位，这样经过一次定位就可以交换一个页块的数据。

为使一次交换的页块中的数据得到充分的利用，在设计系统时必须仔细考虑一个页块中数据的相关性，于是产生B-树。

B-树和B+树

B-树的基本思想

采用分块技术。

将外存分为若干固定大小的块，称为页块或物理块。

另外开辟一个或多个缓冲区，每个缓冲区的大小与外存的一个页块相同，每次内、外存交换以整个页块为单位，这样经过一次定位就可以交换一个页块的数据。

为使一次交换的页块中的数据得到充分的利用，在设计系统时必须仔细考虑一个页块中数据的相关性，于是产生B-树。

B-树和B+树

B-树的基本思想：

数据结构中的指针不再指向内存，而是指向文件中的位置。

如果 x 是指向一个对象的指针

如果 x 在内存中 $key[x]$ 指向它

否则 DiskRead(x) 将对象从磁盘读入内存 (DiskWrite(x) – 将其写回磁盘)

B-树和B+树

动态查找

B-树：

定义。

查找算法。

插入算法——结点分裂，使B-树升高。

删除算法——结点合并，使B-树降低。

B+树：

定义。与B-树的差异。

查找算法，与B-树算法的差异。

B-树和B+树

B-树的定义：一种平衡的多路查找树

定义：一棵m阶的B-树，或为空，或为满足下列特征的m叉树：

(1) 树中每个结点至多有m棵子树，意味着最多m-1个关键字；

(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$\lceil m/2 \rceil$棵子树，至多含有 m 棵子树；

(4) 所有的非终端结点中包含下列信息数据

$$(n, a_0, k_1, a_1, k_2, a_2, \ldots\ldots , k_n, a_n)$$

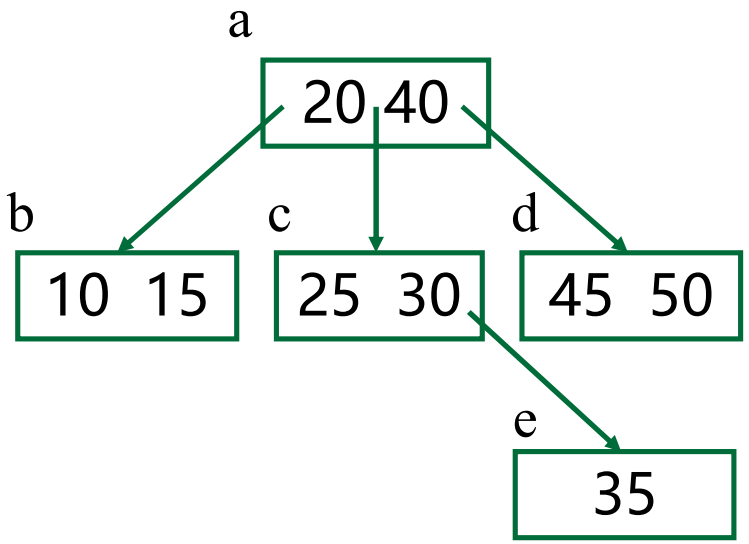
其中 k_i 为关键字，且 $k_i<k_{i+1}$；a_i 为指向子树根结点的指针，且指针 a_{i-1} 所指子树中所有结点的关键字均小于 k_i，a_n 所指子树中所有结点的关键字均大于 k_n。n 为关键字个数，有 n+1棵子树。

(5) 所有子树的叶子结点均在同一层，且不带信息。

B-树和B+树

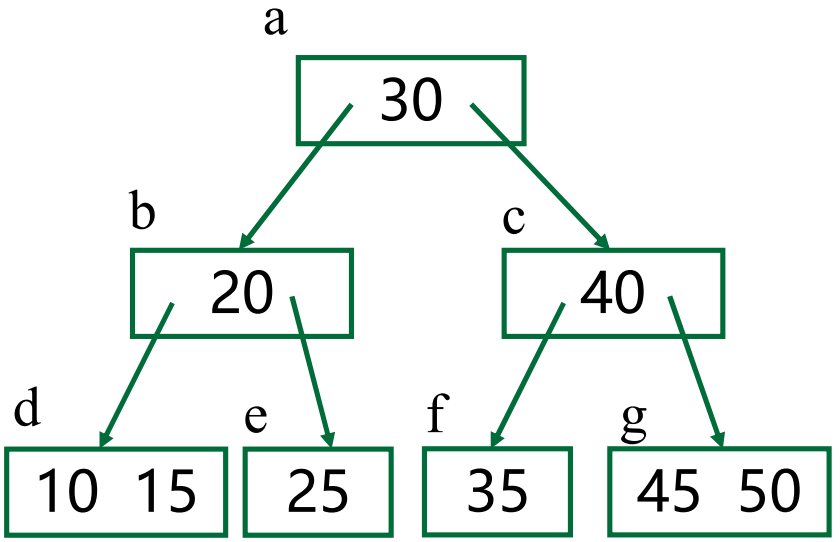
B-树的定义

B-树是一种平衡的多路查找树。



非B-树

不满足定义5



B-树

B-树和B+树

B-树的查找过程

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。

B-树和B+树

B-树的插入过程

在查找不成功之后，需进行插入。

显然，插入的位置必定在最下层的非叶结点，有下列几种情况：

1）插入后，该结点的关键字个数 n<m，则直接插入，不修改指针。插入点一定是叶子结点。

B-树和B+树

B-树的插入过程

2）插入后，该结点的关键字个数 n=m，则需进行“结点分裂”，

$(A_0,K_1,\cdots\cdots,K_{s-1},A_{s-1},K_s,A_s,K_{s+1},\cdots\cdots,K_n,A_n)$

令 $s=\lceil m/2 \rceil$，在原结点中保留：

$(A_0, K_1, \cdots\cdots, K_{s-1}, A_{s-1})$

建新结点 p

$(A_s, K_{s+1}, \cdots\cdots, K_n, A_n)$

将 (K_s, p) 插入双亲结点。

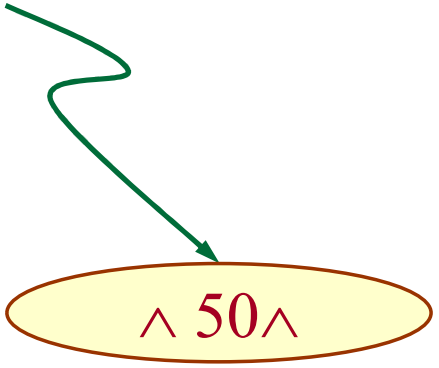
如果双亲结点插入后也满，则继续对双亲结点进行结点分裂操作。

如果双亲结点为空，则建立新的根结点。

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

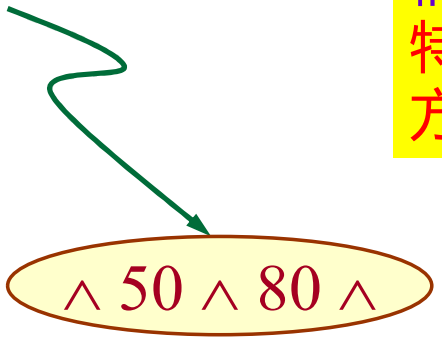


插入关键字 = 50，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。



情况1——

特点：插入后，该结点的关键字个数n<m

方法：不修改指针

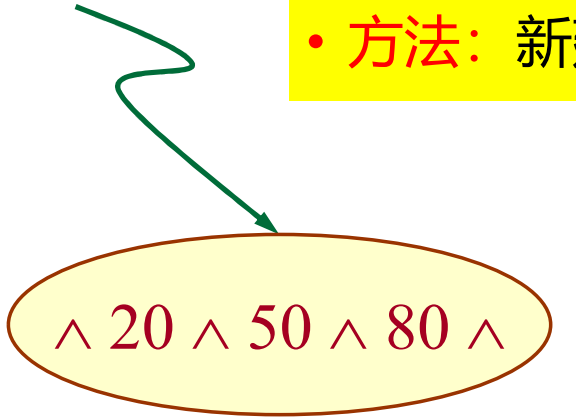
插入关键字 = 80，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

- 情况3——特点：结点分裂后，没有双亲
- 方法：新建双亲节点

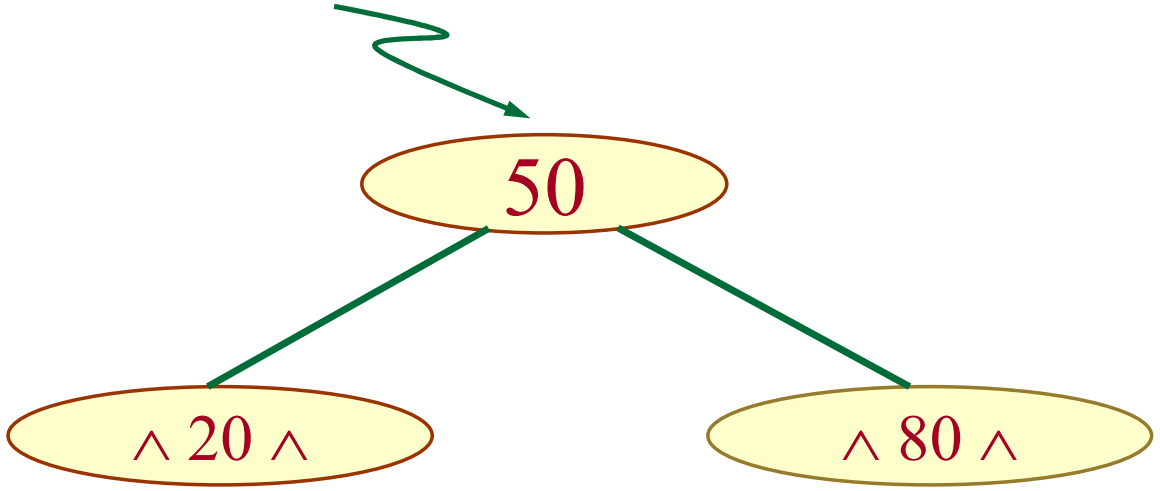


插入关键字 = 20，结点的关键字个数 n=m，
需要进行分裂

B-树和B+树

B-树的插入过程

例如：m=3的B-树。



B-树和B+树

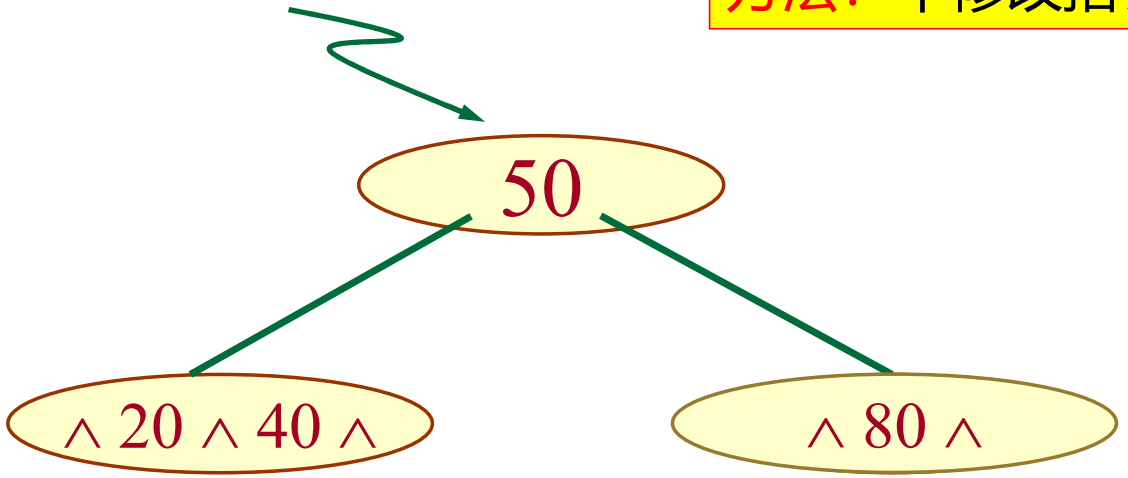
B-树的插入过程

例如：m=3的B-树。

情况1——

特点：插入后，该结点的关键字个数n<m

方法：不修改指针



插入关键字 = 40，直接插入

B-树和B+树

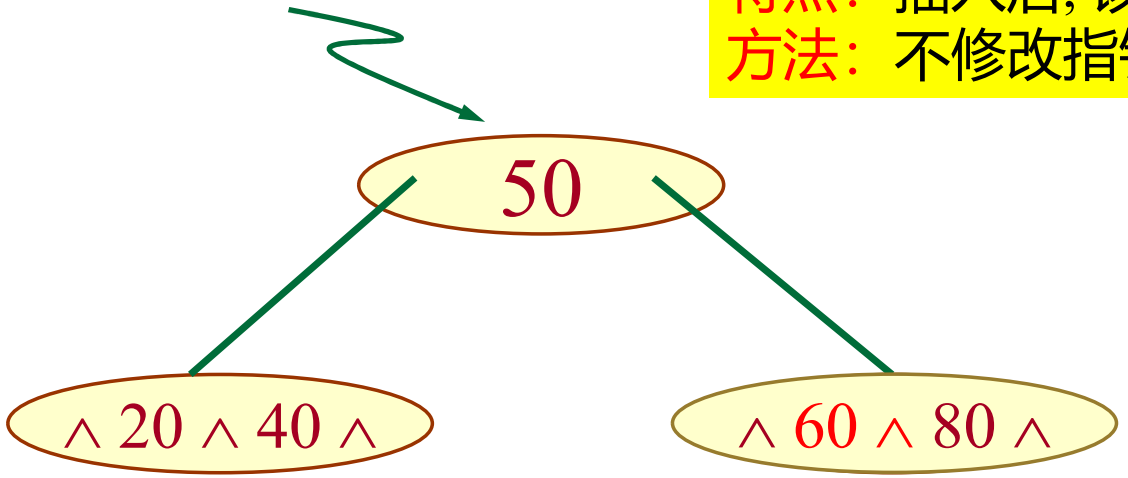
B-树的插入过程

例如：m=3的B-树。

情况1——

特点：插入后，该结点的关键字个数n<m

方法：不修改指针



插入关键字 = 60，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

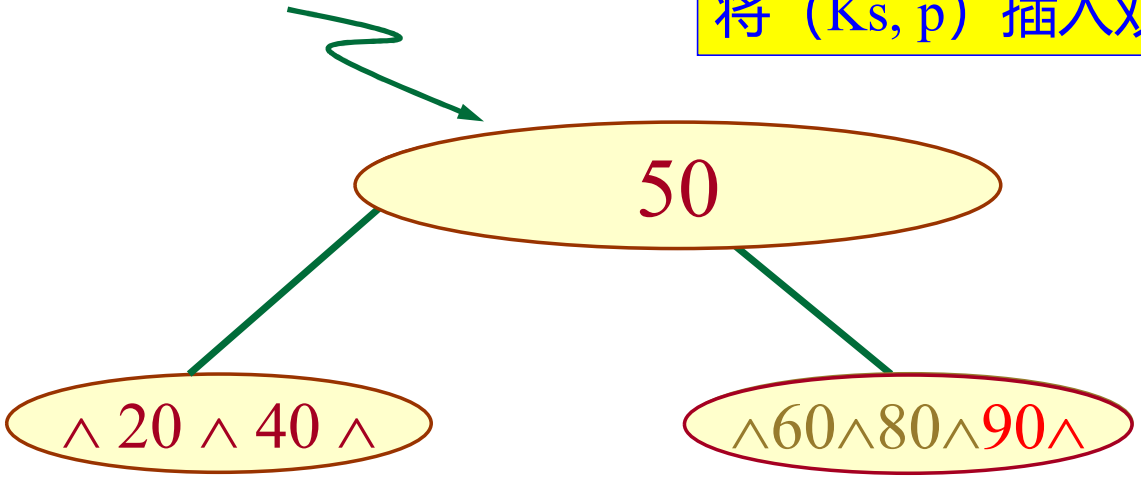
情况2——特点：插入后结点 n=m

方法：结点分裂，令 s =⌈m/2⌉,

结点中保留：（A0, K1, ……，Ks-1, As-1）；

建新结点：（As, Ks+1, ……，Kn, An）；

将（Ks, p）插入双亲结点；



插入关键字 = 60，　　90，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

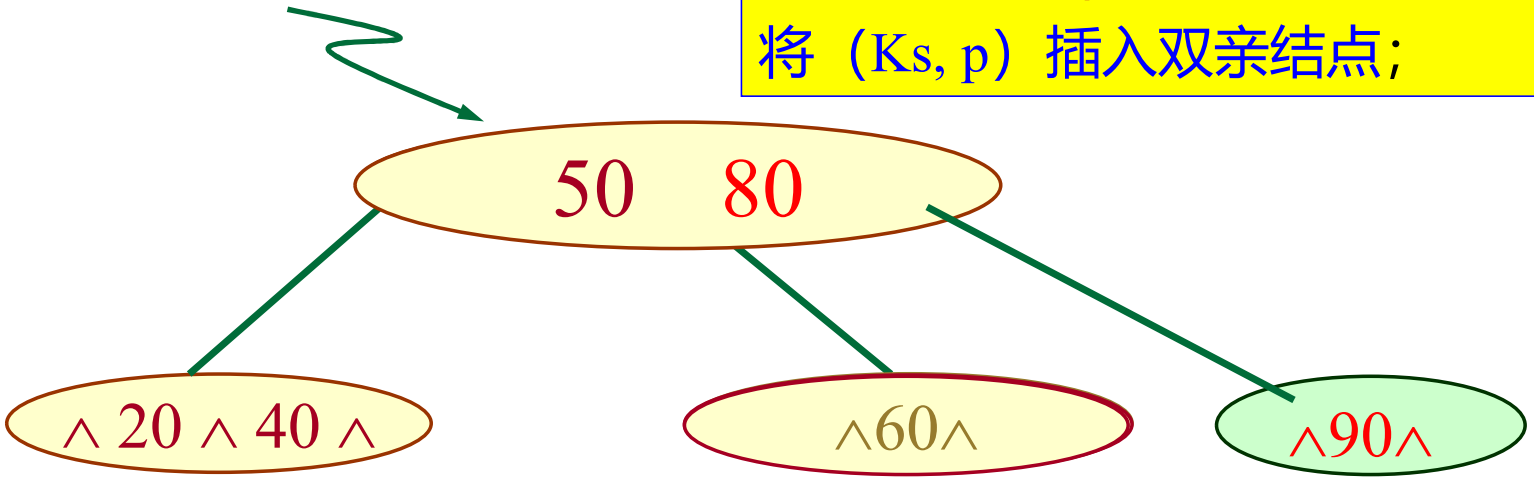
情况2——特点：插入后结点 n=m

方法：结点分裂，令 s =⌈m/2⌉,

结点中保留：（A0, K1, …… , Ks-1, As-1）；

建新结点：（As, Ks+1, …… , Kn, An）；

将（Ks, p）插入双亲结点；



插入关键字 = 60，　　90，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

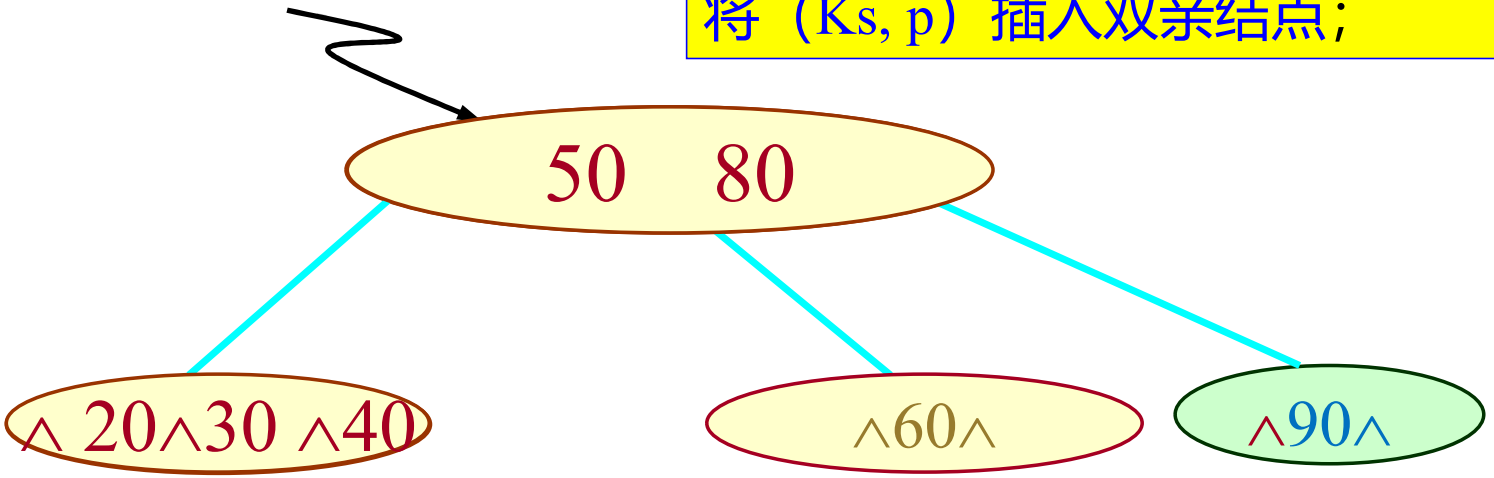
情况2——特点：插入后结点 n=m

方法：结点分裂, 令 s =⌈m/2⌉,

结点中保留：（A0, K1, ……，Ks-1, As-1）；

建新结点：（As, Ks+1, ……，Kn, An）；

将（Ks, p）插入双亲结点;



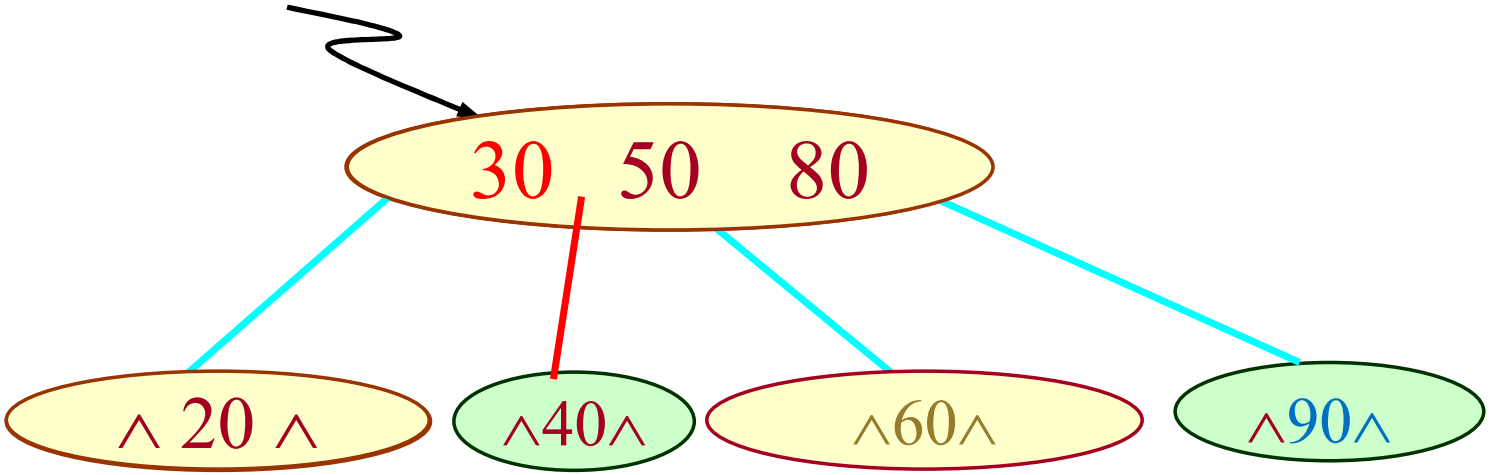
插入关键字 = 60，　90，　30，

B-树和B+树

B-树的插入过程

例如：m=3的B-树。

- 情况3——特点：结点分裂后，没有双亲
- 方法：新建双亲节点

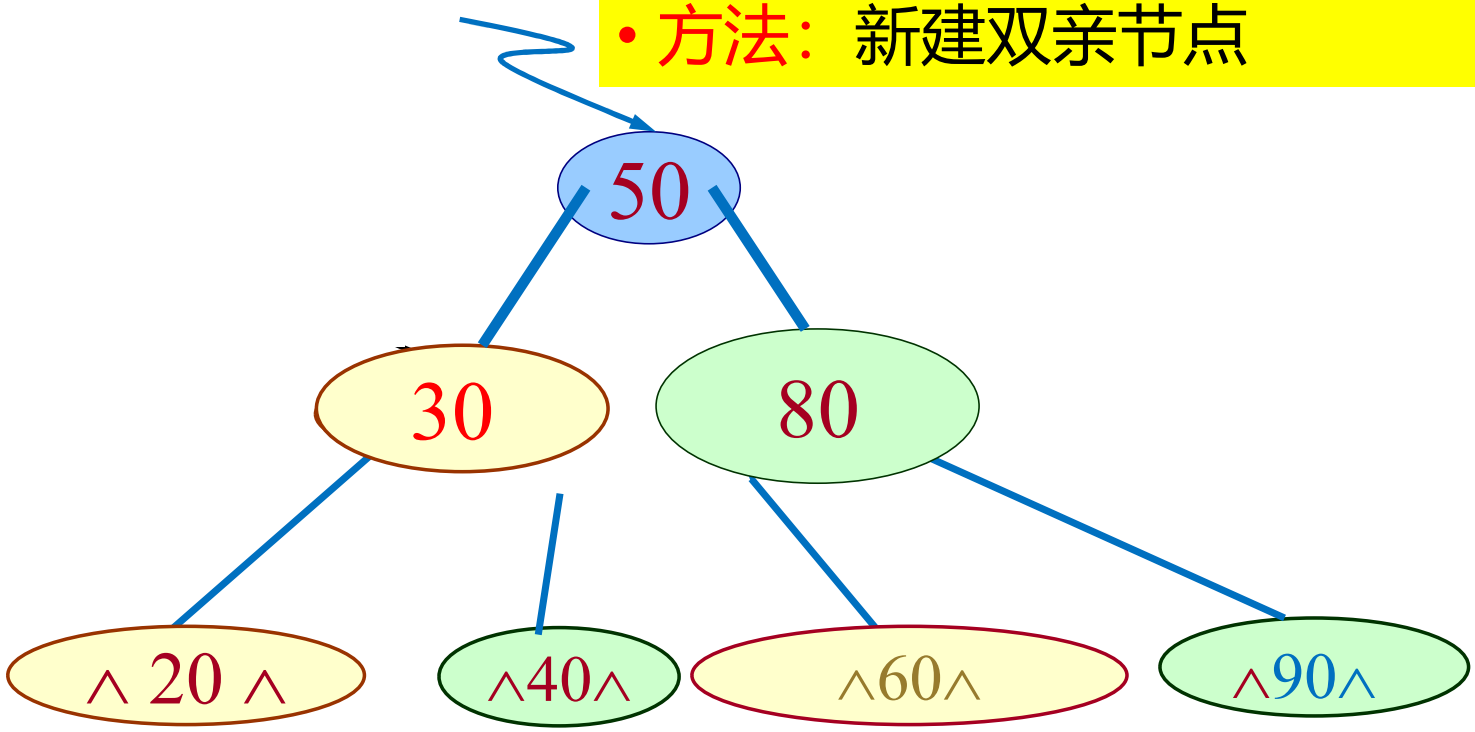


插入关键字 = 60，　90，　30，

B-树和B+树

B-树的插入过程

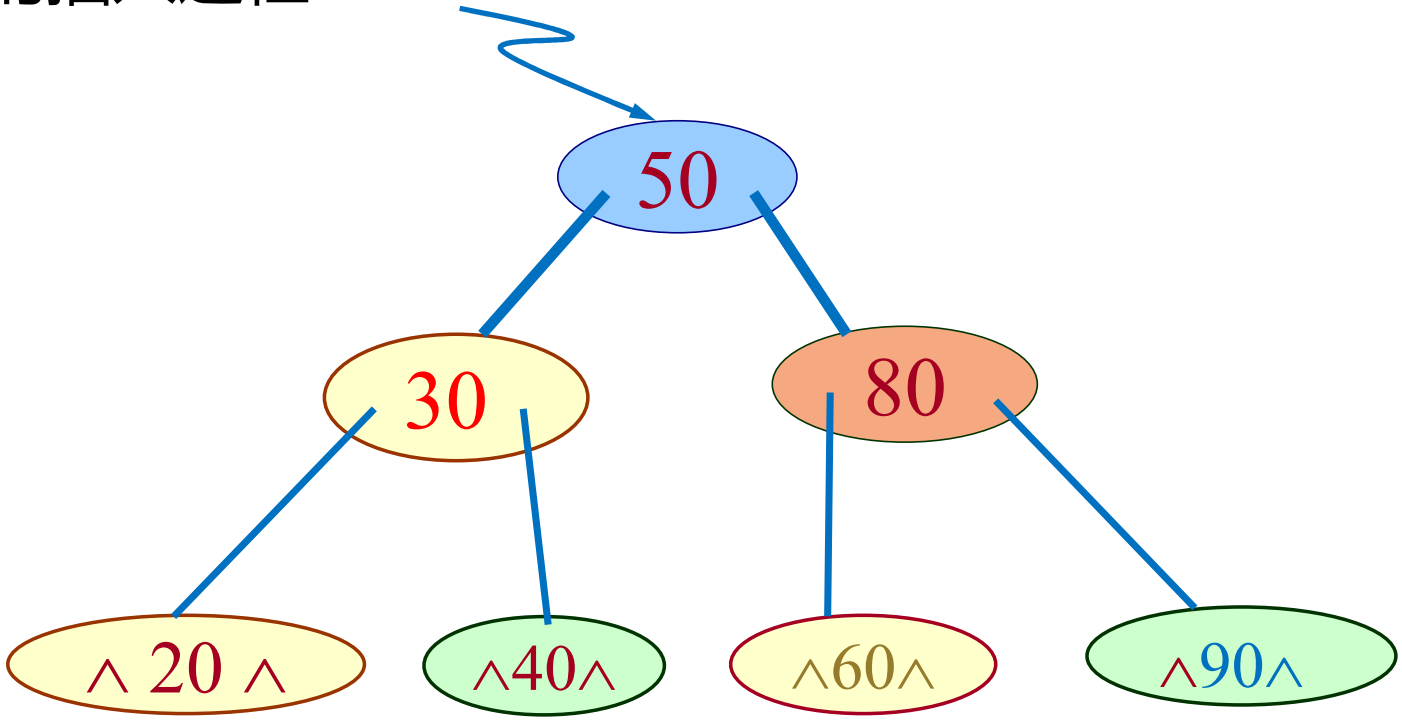
- 情况3——特点：结点分裂后，没有双亲
- 方法：新建双亲节点



插入关键字 = 60，　90，　30，

B-树和B+树

B-树的插入过程



插入关键字 = 60，　90，　30

B-树和B+树

B-树的插入过程

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

B-树是通过结点的分裂而长高的。

B-树和B+树

B-树的删除过程

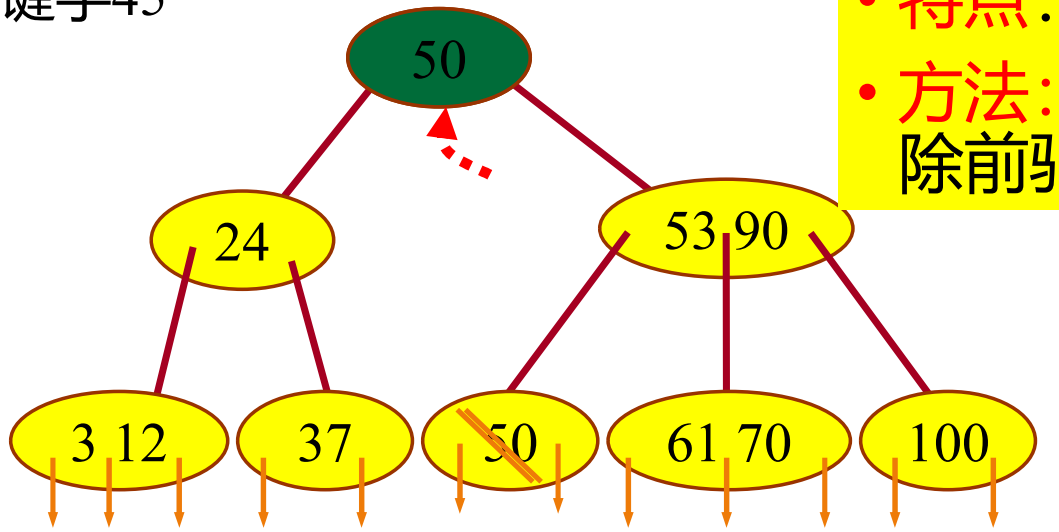
与插入操作相反，删除首先必须找到待删关键字所在结点，并且要求删除之后，结点中关键字的个数不能小于⌈m/2⌉-1，否则，要从其左（或右）兄弟结点“借调”关键字，若其左和右兄弟结点均无关键字可借（结点中只有最少量的关键字），则必须进行结点的“合并”。

B-树是通过结点的合并而降低的。

B-树和B+树

例如：m=3,⌈*m*/2⌉-1=1；至少1个关键字, 2个子结点, 最多2个关键字。

删除关键字45

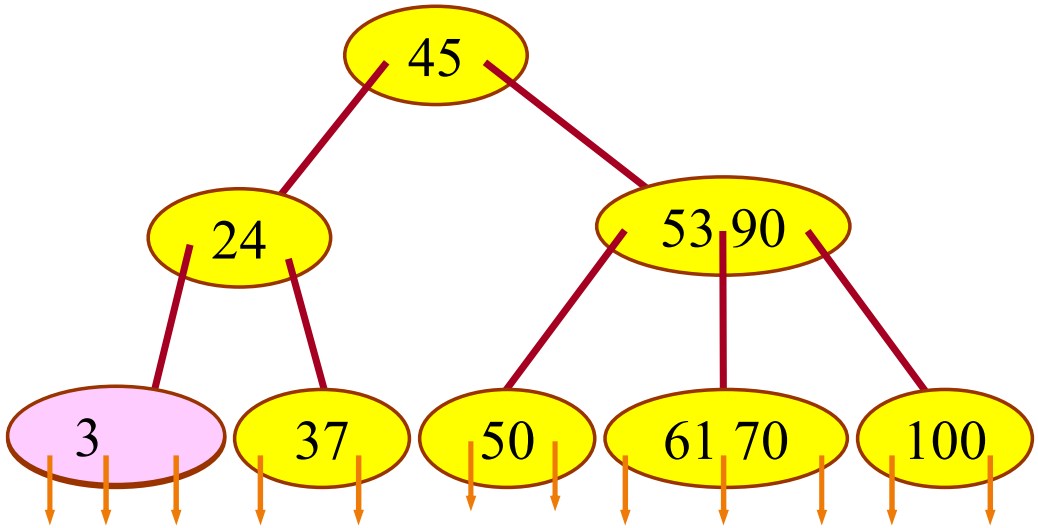


- 特点：非叶子节点
- 方法：向前驱/后继借关键字, 然后删除前驱/后继中借走的关键字

B-树和B+树

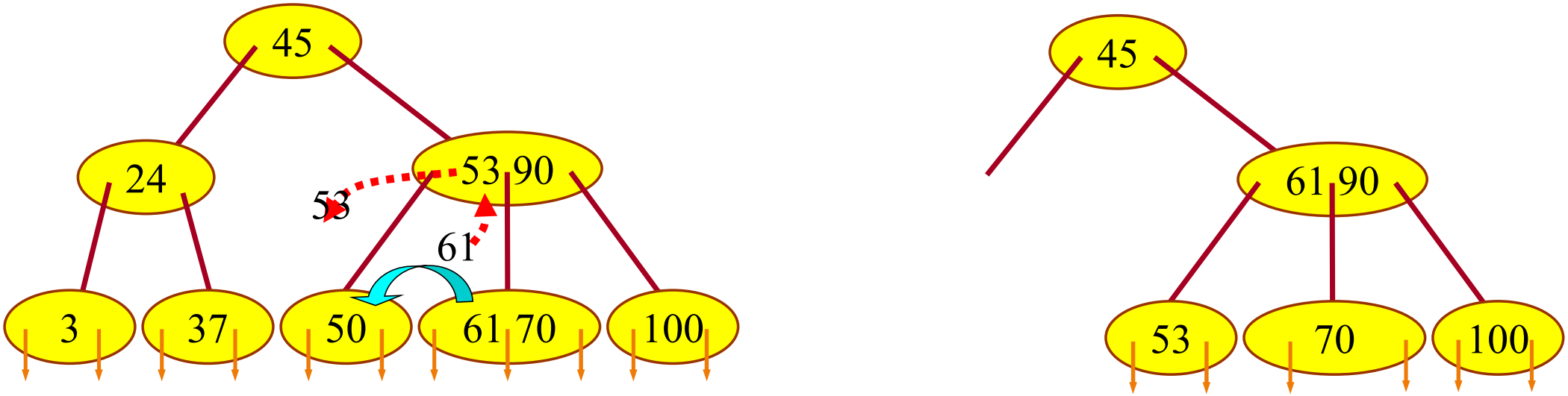
删除关键字K = 12

- 情况1——
- 特点：K所在节点Dk的关键字数目不小于⌈m/2⌉
- 方法：从Dk中删除关键字K及其对应指针



B-树和B+树

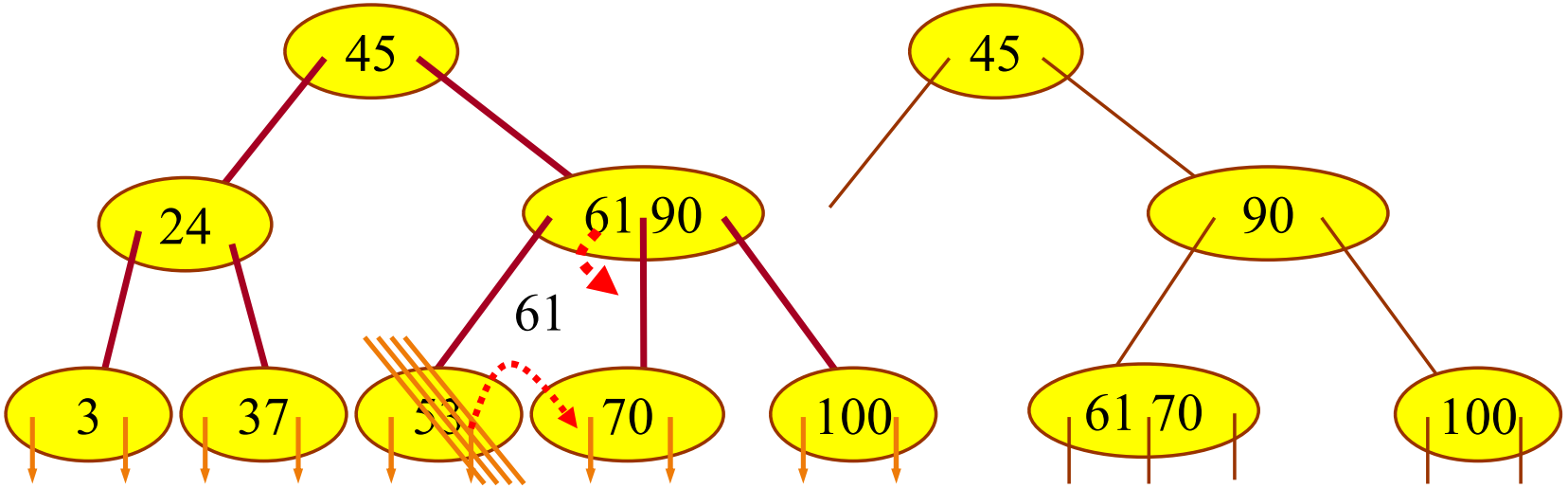
删除关键字K = 50



- 情况2——
- 特点：K所在节点Dk的关键字数目等于⌈m/2⌉ - 1, 而其相邻右兄弟Dkr（左兄弟Dkl）关键字数目大于⌈m/2⌉ - 1

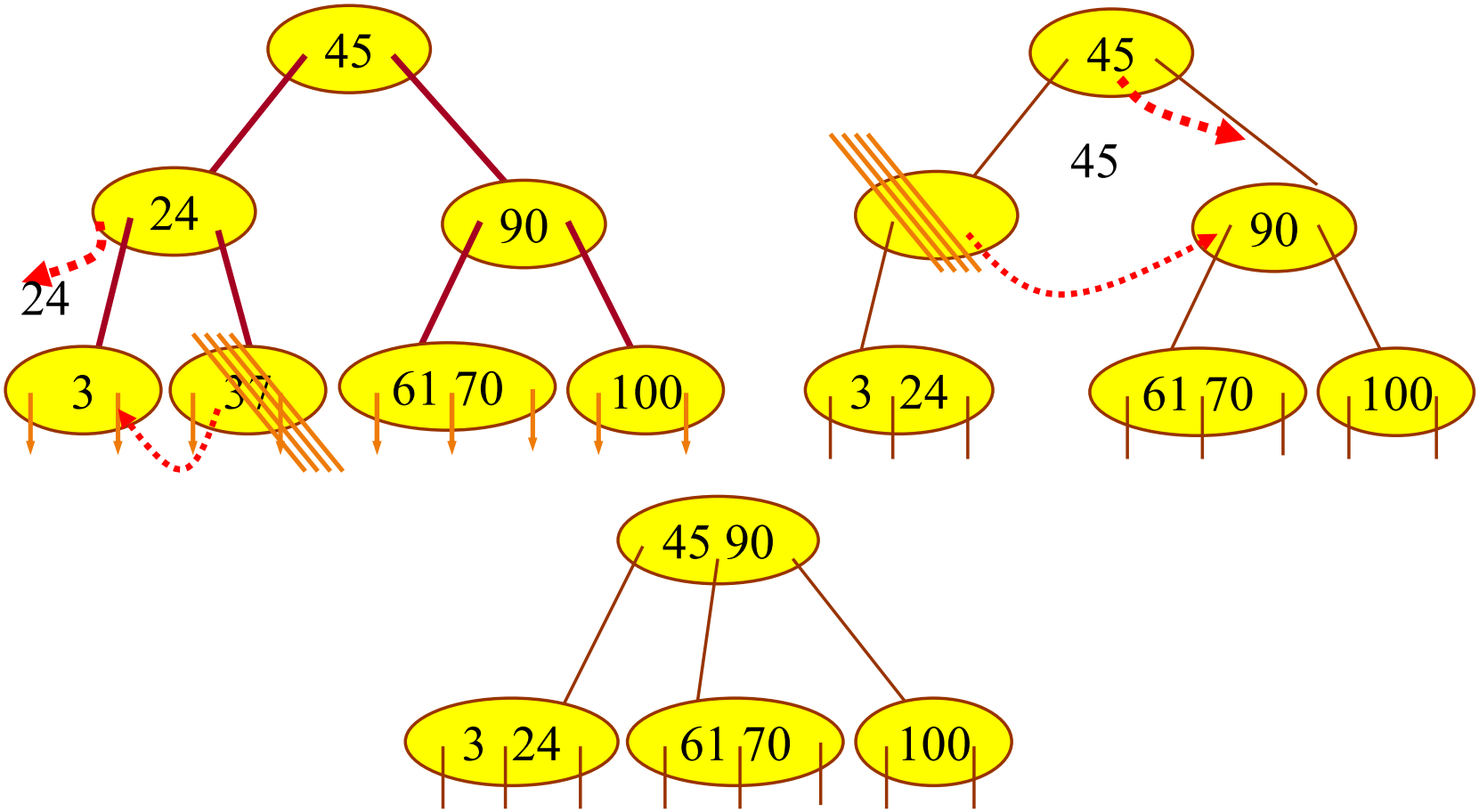
- 方法：将Dkr(Dkl)中最小（最大）的关键字上移到父节点中, 而将父节点中小于（大于）且紧邻该上移关键字的关键字下移到Dk中。

删除关键字K=53



- 情况3——特点： K所在节点Dk和其相邻右兄弟Dkr和左兄弟Dkl的关键字数目都小于⌈m/2⌉ - 1
  - 方法：删除关键字K, DK剩余的关键字及父节点中与之对应的关键字一起加入到Dkr（Dkl）中。

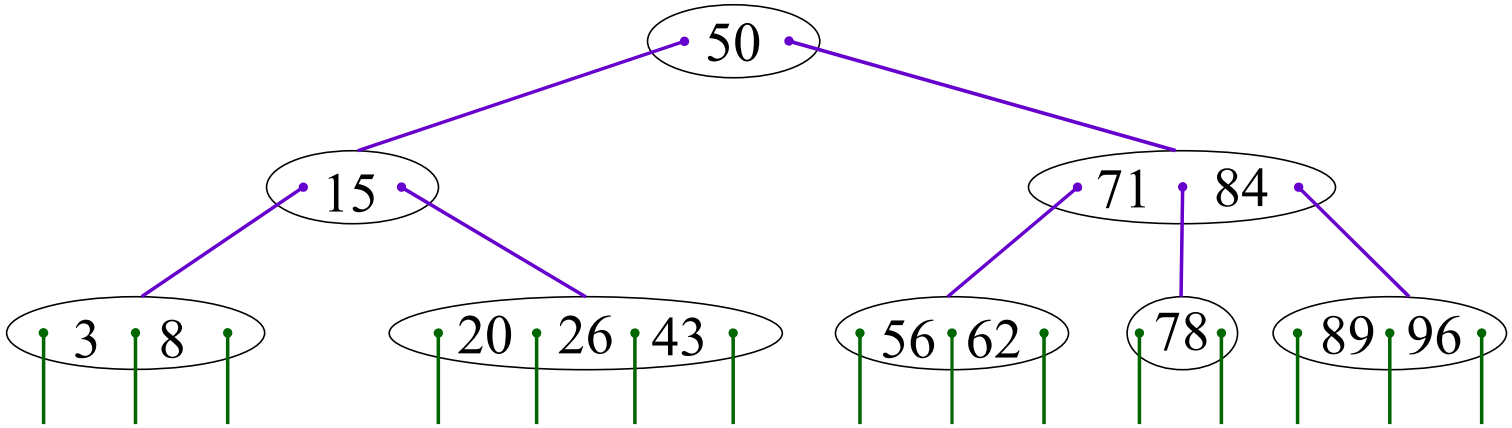
删除关键字37



- 在B-树中进行查找时, 其查找时间主要花费在搜索结点上, 即主要取决于B-树的深度H。
- 考虑最坏情况：含N个关键字的 m 阶 B-树可能达到的最大深度 H ?
- 分析:
  - 设在 m 阶B-树中，失败节点位于第 L+1层，即叶子节点。
  - 若树中关键码有 N 个, 则失败节点数为 $N+1$。这是因为失败发生在 $K_i < x < K_{i+1}$, $0 \le i \le N$，设$K_0 = -\infty$，$K_{N+1} = +\infty$。

- 假设m 阶B-树含 N 个关键字，最坏情况下查找性能
- 1）最坏情况：B-树高度/最大的情况
- 即每个节点包含最少关键字的B-树(除根和叶子外，其它结点至少有$\lceil m/2 \rceil$个孩子)：
  - 第一层：1；
  - 第二层：2，根节点至少两个孩子
  - 第三层：$2(\lceil m/2 \rceil)$ ……
  - 第L层：$2(\lceil m/2 \rceil)^{L-2}$
  - 第L+1层：$2(\lceil m/2 \rceil)^{L-1}$

- 2）第L+1层失败节点为N+1个（即第*L*层的指针数）
  - $N+1 >= 2(\lceil m/2 \rceil)^{L-1}$
- 所以：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$\log_{\lceil m/2 \rceil}((N+1)/2)+1$



- 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$\log_{\lceil m/2 \rceil}((N+1)/2)+1$
- 示例：若B-树的阶数 $m = 199$，关键码总数 $N = 1999999$，则B-树的高度 h 不超过

$$\log_{100} 1000000 + 1 = 4$$

B+树： B-树的一种变形

定义：一棵m阶的B+树是满足下列特征的m叉树：

*(1) 树中每个结点至多有m棵子树；

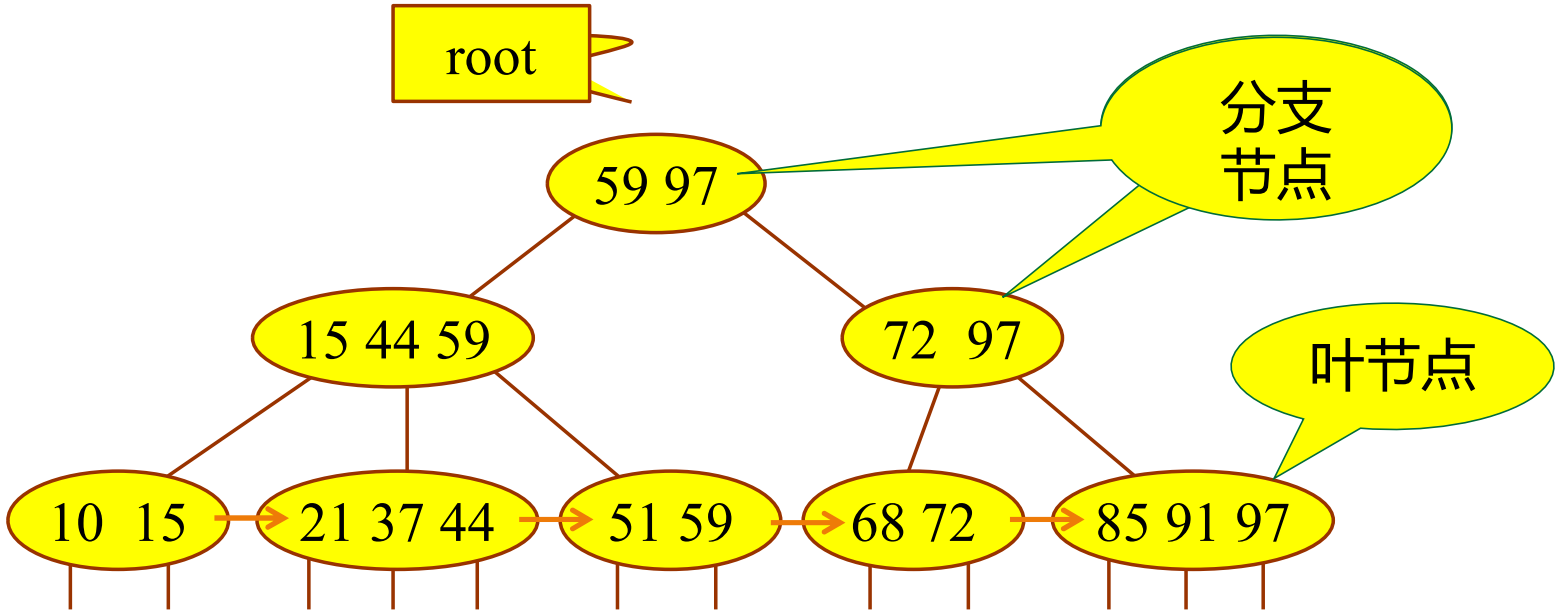
*(2) 若根结点不是叶子结点，则至少有两棵子树；

*(3) 除根之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；

与B-树的第一个区别
(B-树是n-1)

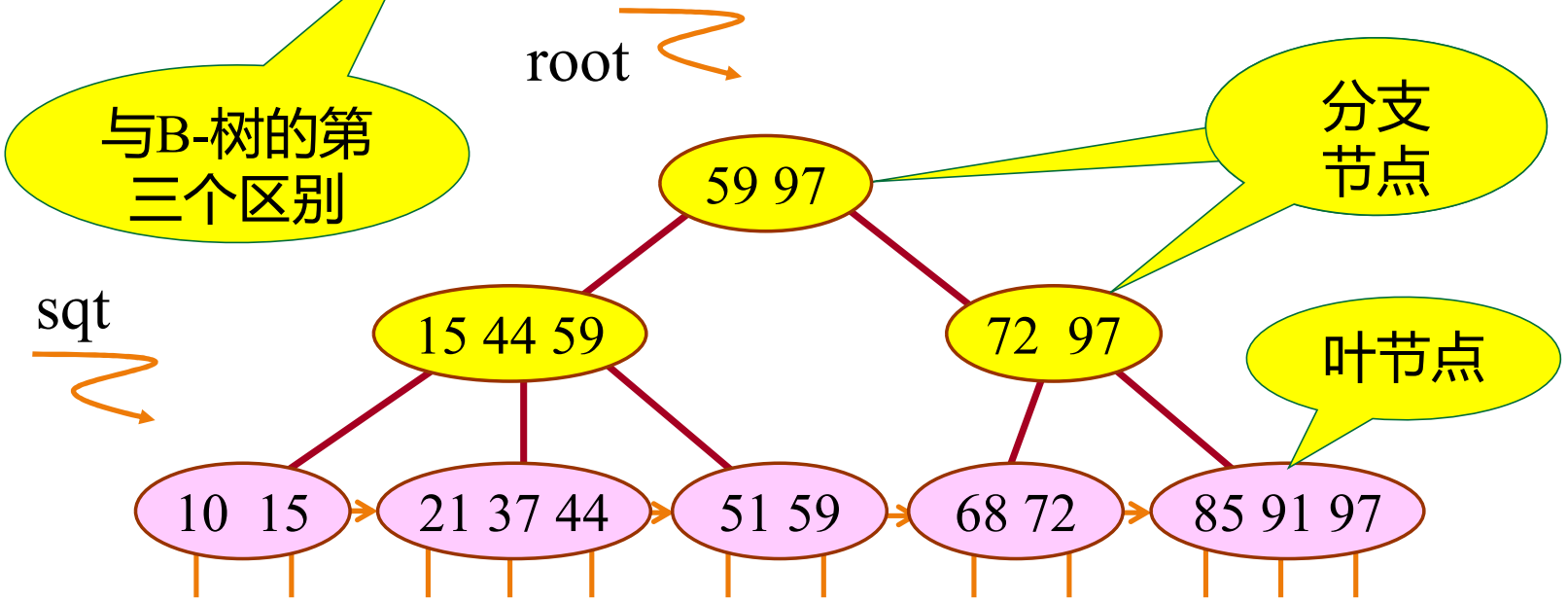
(4) 有 n 棵子树的结点有 n 个关键字, 每个结点包含：(n, k_1, a_1, k_2, a_2, ……, k_n, a_n)，其中,

- k_i 为关键字, 且 $k_i<k_{i+1}$;
- a_i 为指向子树根结点的指针, 且指针 a_i 所指子树中所有结点的关键字均小于等于k_i。



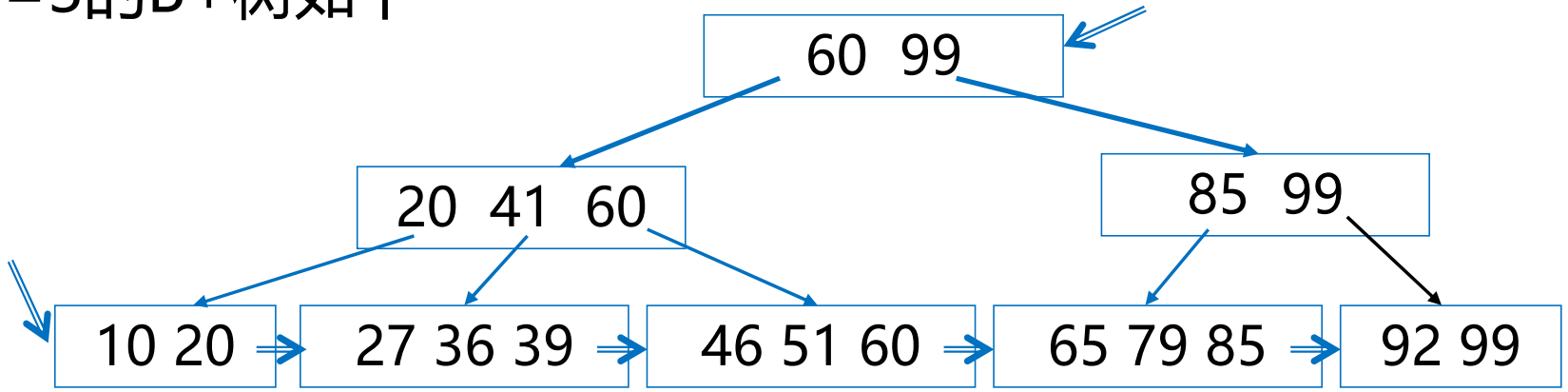
与B-树的第二个区别

- (5) 所有叶结点均在同一层。叶结点按关键字大小顺序链接。可将每个叶结点看成一个基本索引块（直接指向数据文件）。
- (6) 分支结点中仅包含它的各个结点中最大（或最小）关键字的分界值及子结点的指针。所有分支结点可看成是索引的索引。



B-树和B+树

例：m=3的B+树如下



B+树的查找

B+树有两个头指针：一是指向B+树的根结点；另一是指向关键字码最小的叶结点，所有叶结点链成线形表，则可以直接从最小关键字开始顺序检索。

与B-树的第四个区别

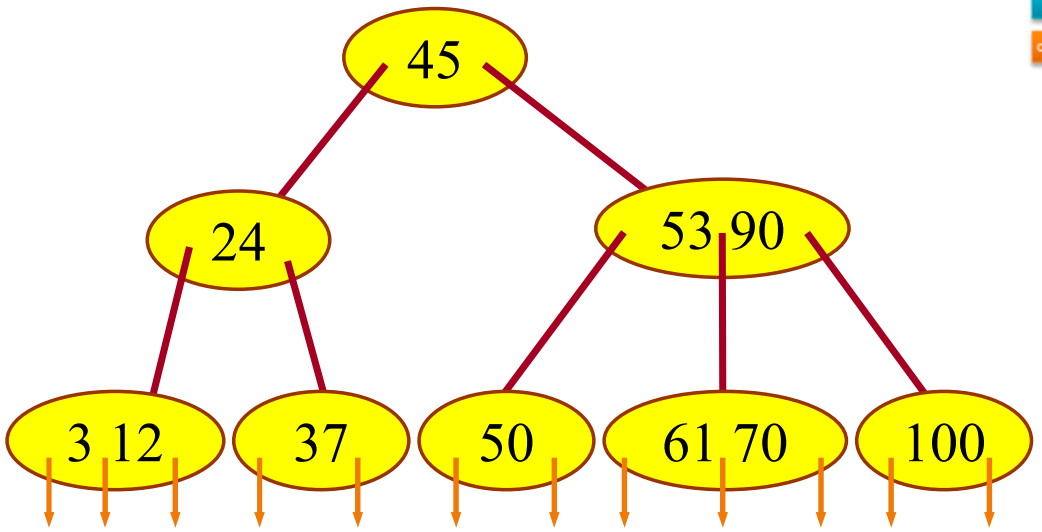
当从B+树根结点开始随机查找时，检索方法与B-树相似，但在分支结点中的关键字与检索关键字相等时，检索并不停止，要继续查找到叶结点为止。

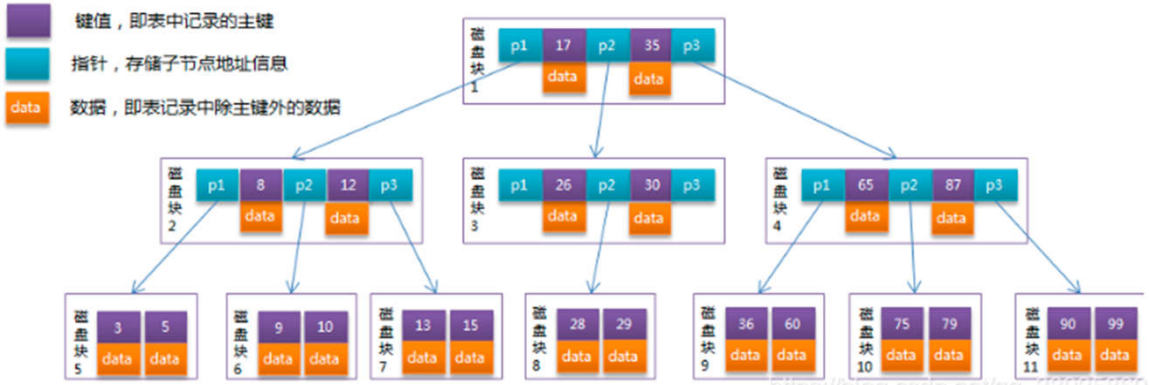
B-树和B+树

B-树在索引文件中的应用

1、B-树存储在外存中

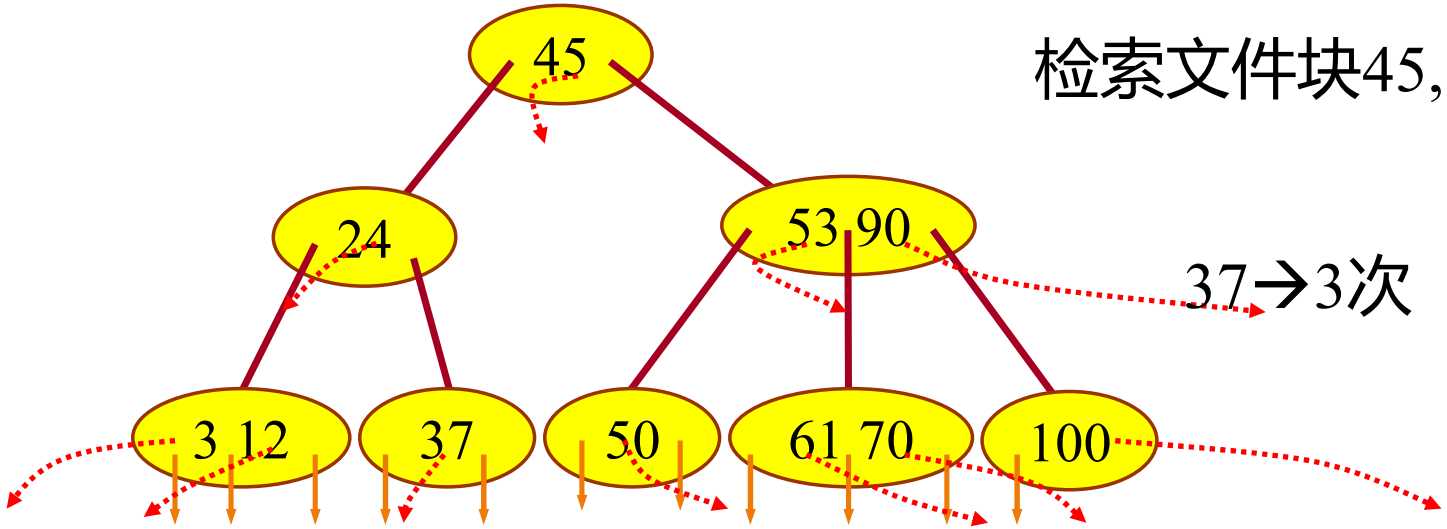
2、B-树的每个结点存放在外存的一个页块上（因此B-树的阶数一般取得较大）。





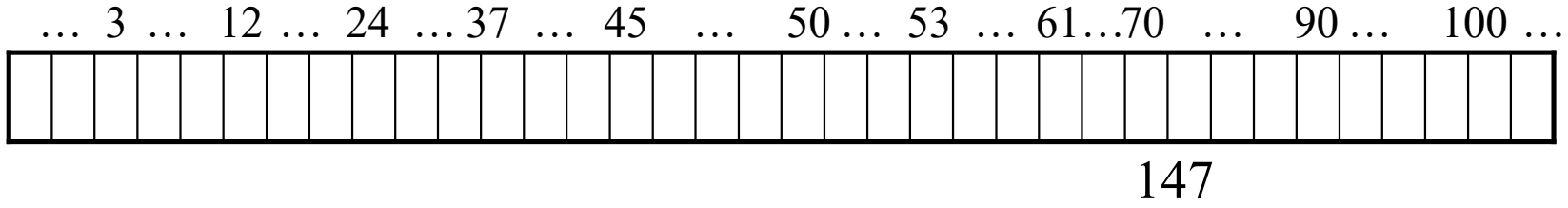
B-树和B+树

3、B-树中任何结点内的一个关键字实际上是一个索引项, 由一个关键字 k 和一个指针 q 组成二元组（k, q）。q是指向主文件页块（或主文件记录）的指针



检索文件块45, 需读取1次外存

37→3次



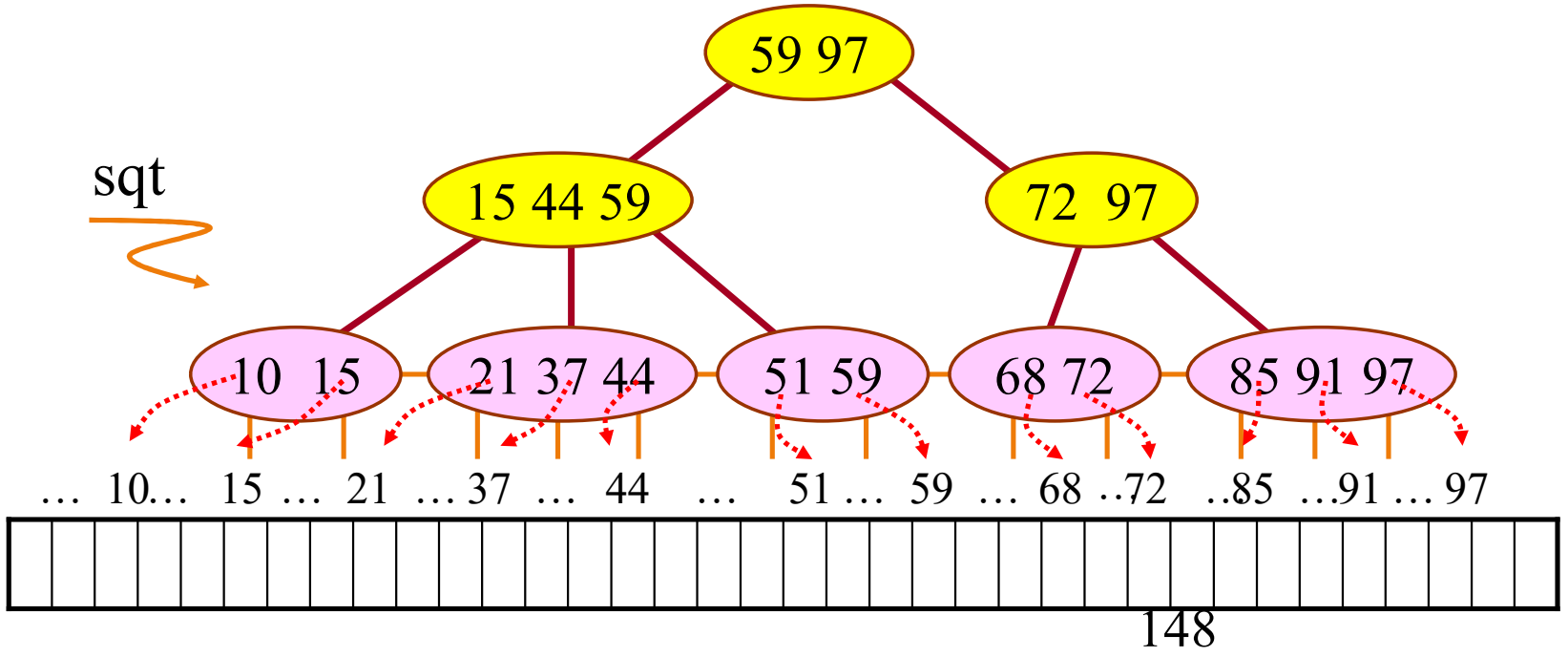
147

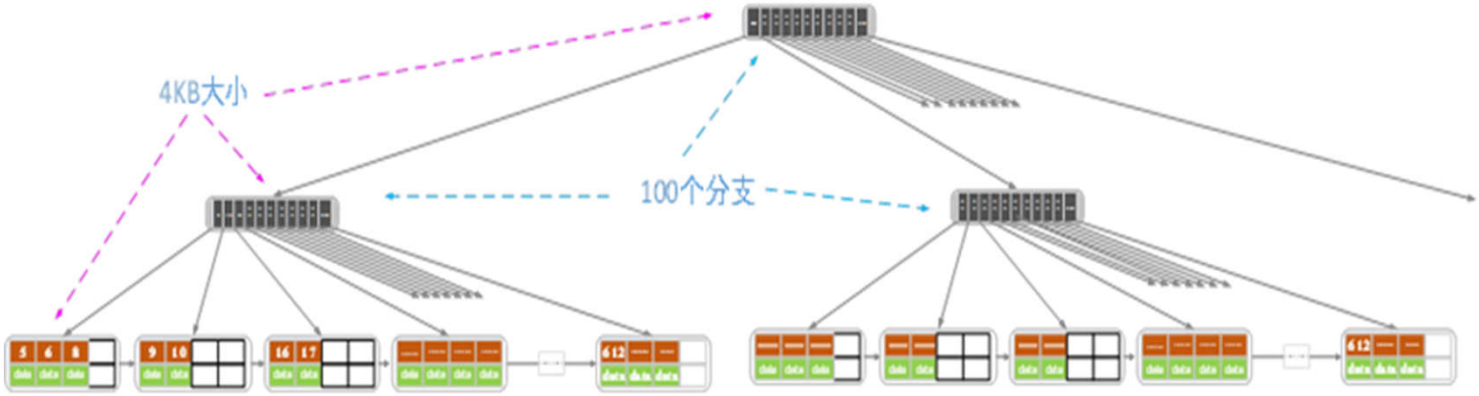
B-树和B+树

B+树在索引文件中的应用

1、B+树的每个结点存放在外存的一个页块上（因此B+树的阶数一般都比B-树大）。

2、B+树的树叶层是主文件的稀疏索引，整个B+树构成多级索引。索引项就是B+树中的一个关键字和它对应的指针所构成的二元组。





一个扇区有512字节，假设，选择4KB作为内存和磁盘之间的传输单位，那么在设计B+树的时候，不论是索引结点还是叶子结点都使用4KB作为结点的大小。再假设一个记录的大小是1KB，那么一个叶子结点可以存4个记录。而对于索引结点（大小也是4KB），由于只需要存key值和相应的指针，所以一个索引结点可以存储100~150个分支。假定索引结点的阶数是100，三级索引可存储400万个记录（1KB），只需要通过三次IO时间就从400万个记录中找到了对应的key记录

B-树和B+树

B+树比B~树更适合实际应用中操作系统的文件索引

1. 有n棵子树的非叶子结点中含有n个关键字（B-树是n-1个），这些关键字不保存数据，只用来索引，所有数据都保存在叶子节点（B-树是每个关键字都保存数据）。
2. 所有的叶子结点中包含了全部关键字的信息，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. 所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。
4. 通常在B+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。
5. 同一个数字会在不同节点中重复出现。

B-树和B+树

B+树比B~树更适合实际应用中操作系统的文件索引

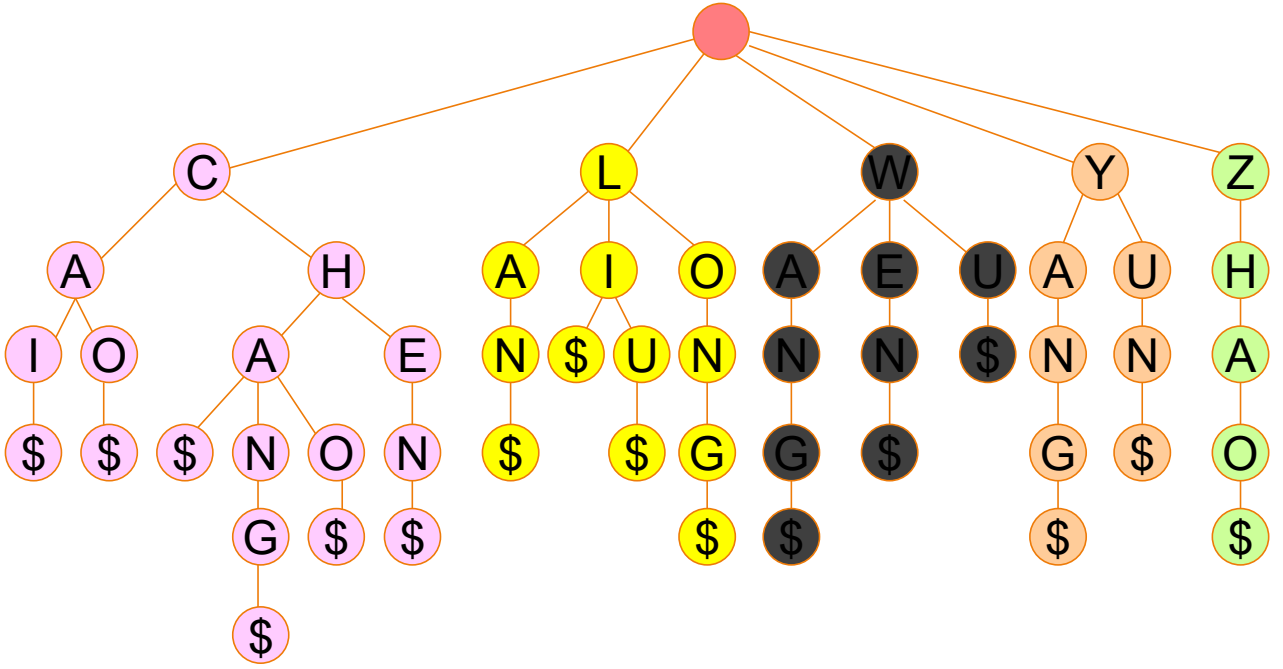
5. B+树的中间节点不保存数据，所以磁盘页能容纳更多节点元素，更“矮胖”；
6. B+树查询必须查找到叶子节点，B-树只要匹配到即可，不用管元素位置，因此B+树查找更稳定（并不慢）；
7. B+树全节点遍历更快：B+树遍历整棵树只需要遍历所有的叶子节点即可，，B-树却需要重复地中序遍历

8.2.3 键树

键树, 又称数字搜索树（Digital Search Tree）

- 是一棵度大于等于2 的树
- 树中每个节点不是包含关键字, 而含有组成关键字的符号。

例：将集合进行逐层分割, 得到下列键树{CAI, CAO, LI, LAN, CHA, CHANG, WEN, CHAO, YUN, YANG, LONG, WANG, ZHAO, LIU, WU, CHEN}



8.2.3 键树

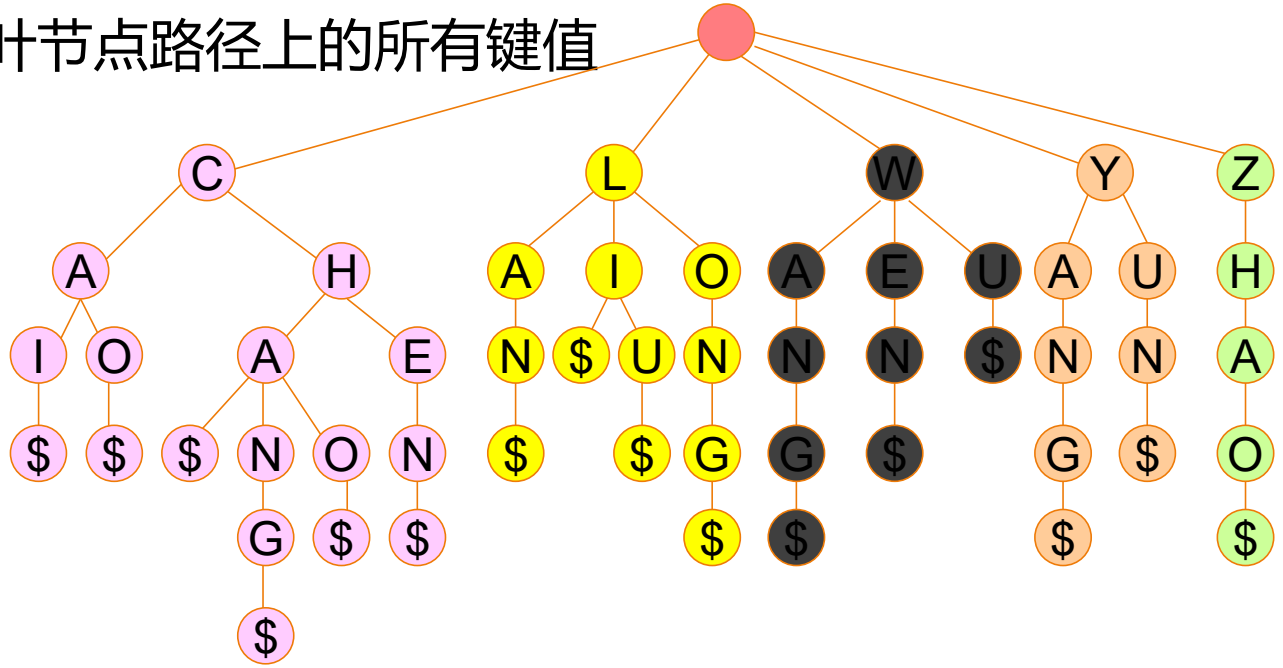
键树的存储方式

A）树的孩子兄弟链表

B）树的多重链表（Trie 树）

每个节点都含有d个指针域

每个叶节点中包含从根到该叶节点路径上的所有键值



§8.3 哈希表

- 基于关键码比较的检索
  - 顺序检索
  - 二分法、树

检索是直接面向用户的操作
当问题规模n很大时，上述检索的时间效率可能使得用户无法忍受

最理想的情况
根据关键码值，直接找到记录的存储地址
不需要把待查关键码与候选记录集合的某些记录进行逐个比较

§8.3 哈希表

散列法（哈希法）基本思想

一个确定的函数关系h

以结点的关键码 Key 为自变量

函数值 hash(Key) 作为结点的存储地址

检索时也是根据这个函数计算其存储位置

通常散列表的存储空间是一个一维数组

散列地址是数组的下标

§8.3 哈希表

- 散列法的几个重要概念
  - 负载因子 $\alpha = n/m$

    散列表的空间大小为m

    填入表中的结点数为n
  - 冲突

    某个散列函数对于不相等的关键码计算出了相同的散列地址

    在实际应用中，不产生冲突的散列函数极少存在
  - 同义词

    发生冲突的两个关键码

§8.3 哈希表

散列法（哈希法）

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

即：通过简单计算直接得到数据的地址。

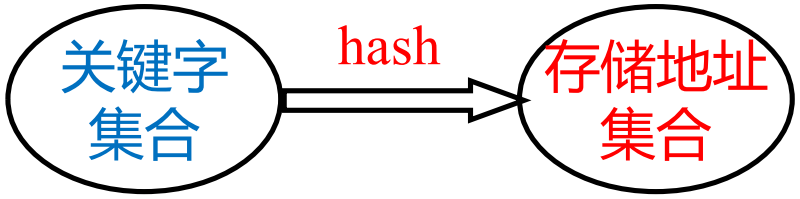
1) 哈希(Hash)函数是一个映射，即：将关键字的集合映射到某个地址集合上

哈希函数可写成：$addr(a_i)=H(k_i)$

a_i是表中的一个元素

$addr(a_i)$是a_i的存储地址

k_i是a_i的关键字。



§8.3 哈希表

- 散列法（哈希法）

2）由于哈希函数是一个压缩映射，很容易产生“冲突”，即：key1≠ key2，而 f(key1) = f(key2)。

3）很难找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

例 某地区的人口统计表

H(年度) = 年度-1948

	1	2	3		51	52
年份	1949	1950	1951		1999	2000
人数	2000	2100	2200		4400	4420

§8.3 哈希表

散列法（哈希法）

例 30个地区的各民族人口统计表

编号	地区	总人口	汉族	回族……
1	北京			
2	上海			
⋮	⋮			

以编号作关键字，
构造哈希函数：
H(key)=key
H(1)=1
H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19
H(Shenyang)=19

§8.3 哈希表

散列法（哈希法）

哈希表——应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表。

哈希查找——又叫散列查找，利用哈希函数进行查找的过程叫哈希查找。

§8.3 哈希表

散列法（哈希法）

在构造这种特殊的“查找表” 时，除了需要选择一个“好” （尽可能少产生冲突）的哈希函数之外，还需要找到一种“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。

常用的解决冲突的方法：开放地址法，链地址法，再哈希法，建立一个公共溢出区。

§9.3 哈希表

构造哈希函数的基本原则

- 运算尽可能简单
- 尽可能减少冲突
- 函数的值域必须在表长的范围内

§9.3 哈希表

构造哈希函数

1. 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。
2. 数字分析法：取关键字的若干数位组成哈希地址。
3. 平方取中法：取关键字平方后的中间几位为哈希地址。
4. 折叠法：将关键字分割成位数相同的几部分，然后取这几部分的叠加和作为哈希地址。

§9.3 哈希表

构造哈希函数

5. 除留余数法：取关键字被某个不大于哈希表长m的数p除后所得余数为哈希地址。
   哈希函数为 H(key)= key%p，p<=m
6. 随机数法：选择一个随机函数，取关键字的随机函数值为它的哈希地址。
   H(key) = random(key)

§9.3 哈希表

解决冲突的方法

开散列方法（open hashing，也称为拉链法，separate chaining）

把发生冲突的关键码存储在散列表主表之外

闭散列方法（closed hashing，也称为开地址方法，open addressing）

把发生冲突的关键码存储在表中另一个槽内

§9.3 哈希表

解决冲突的方法——开放地址法

为产生冲突的记录，求得一个地址序列：

$H_1, H_2, \ldots, H_s$　$1 \le s \le m-1$

$$H_i=(H(key) + d_i) \text{ MOD } m \qquad i=1, 2, \ldots, s$$

H(key)：哈希函数

m：　　哈希表表长

di：　　增量序列

§9.3 哈希表

- 解决冲突的方法——开放地址法

  增量 *di* 有三种取法

  1) 线性探测再散列

     di = c × i　　最简单的情况 *c=1*

  2) 二次探测再散列

     di = 1^2, -1^2, 2^2, -2^2, …

  3) 随机探测再散列

     di=伪随机数序列

     di = i × H2(key) （又称双散列函数探测）

k	19	01	23	14	55	68	11	82	36

•设定H(key) = （H(key) + d_i）MOD 11（表长=11）

1）若采用线性探测再散列处理冲突: $d_i = c \times i$

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			成功
10	9	8	7	6	5	4	3	2	1	1	不成功

成功 ASL=(1+1+2+1+3+6+2+5+1)/9 = 22/9 = 2.44

不成功 ASL=(10+9+8+7+6+5+4+3+2+1+1)/11 = 56/11 = 5.09

查找不成功时的ASL = 总查找次数 / 有效表长

k	19	01	23	14	55	68	11	82	36

•设定H(key) = (H(key) + d_i) MOD 11 (表长=11)

2）若采用平方探测再散列处理冲突： $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \ldots, \pm k^2,$

若发生冲突：

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

ASL=(1+1+2+1+4+1+4+4+1)/9 = 19/9 = 2.11

3)双散列函数探测：$di=i\times H_2(key)$

k	19	01	23	14	55	68	11	82	36

•设定H(key) = （key+d_i）MOD 11（表长=11）

$H_2(key)$ 是另设定的一个哈希函数，它的函数值应和 m 互为素数。

当 m=11时，可设：$H_2(key)$ = (3*key) %10 + 1，

当发生冲突时，可采用 $d_i=i\times H_2(key)$

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

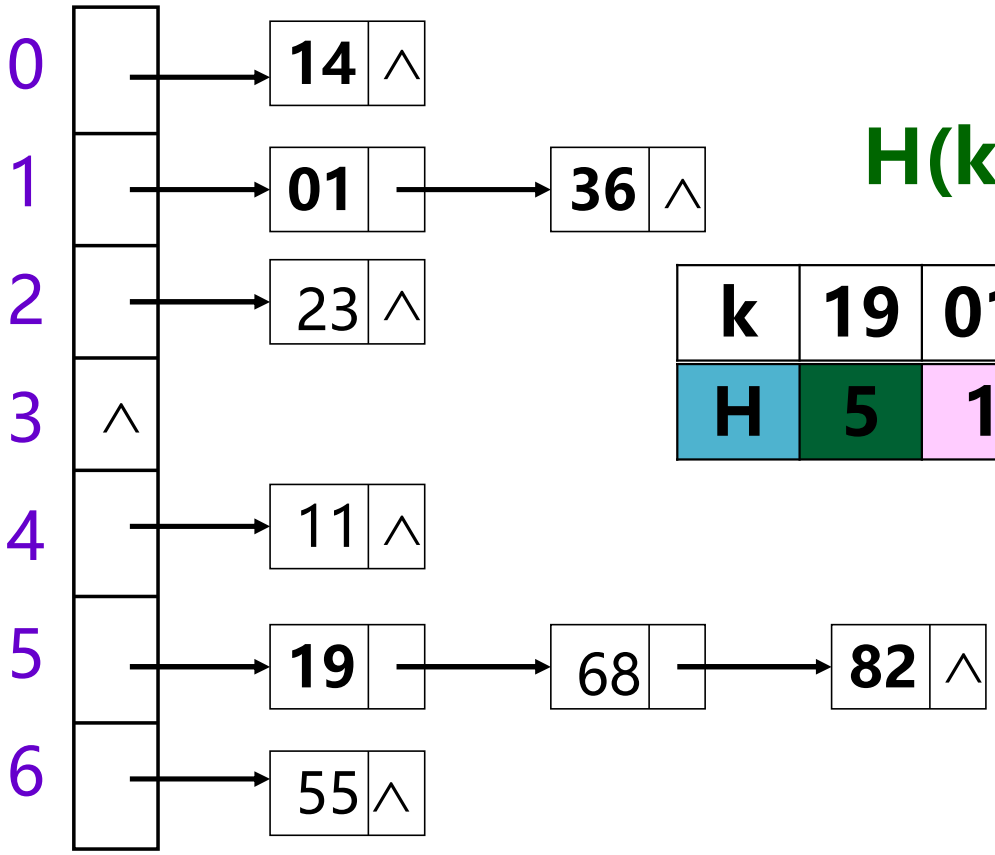
$H_2($ ASL=(2+1+1+1+2+1+2+1+3)/9 = 15/9 = 1.67 = 9

H(11) =(11+4)% 11 = 4

H(36) =(36+2*9)% 11 = 10

3) 链地址法

将所有哈希地址相同的记录都链接在同一链表中。



H(key)=key MOD 7

k	19	01	23	14	55	68	11	82	36
H	5	1	2	0	6	5	4	5	1

ASL=(6×1+2×2+3)/9=13/9

3）公共溢出区

假设哈希函数的值域为[0, m-1]，则设向量HashTable [0...m-1] 为基本表；
另设立向量OverTable[0...v]为溢出表。
一旦发生溢出，都填入溢出表

§9.3 哈希表

哈希表的查找的性能分析

与哈希函数、处理冲突方法和装载因子有关。

假设哈希函数是“均匀的”，处理冲突方法相同，则哈希表平均查找长度依赖于装载因子α 。

$$\alpha = \frac{\text{散列表中的结点数目}}{\text{基本区域能容纳的结点数目}}$$

§9.3 哈希表

查找过程和造表过程一致。

假设采用开放定址处理冲突, 则查找过程为:

对于给定值 K, 计算哈希地址 i = H(K)

若 r[i] == NULL 则查找不成功

若 r[i].key == K 则查找成功

否则 “求下一地址 Hi” , 直至

r[Hi] = NULL (查找不成功)

或 r[Hi].key = K (查找成功) 为止。

哈希表查找成功的ASL

线性探测再散列

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

随机探测、二次探测再散列和再哈希

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

链地址法

$$S_{ne} \approx 1+\frac{\alpha}{2}$$

•装填因子 α=记录数/表长

哈希表查找不成功的ASL

线性探测再散列

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

随机探测再散列

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

链地址法

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

•装填因子 α=记录数/表长

学习要点

1. 掌握顺序表和有序表的查找方法及其平均查找长度的计算方法。

2. 熟练掌握二叉排序树和平衡二叉树的构造和查找方法。

3. 理解B- 树的特点以及它们的建树和查找的过程。

4. 熟练掌握哈希表的构造方法, 深刻理解哈希表与其它结构的表的实质性的差别。

学习要点

1. 掌握顺序表和有序表的查找方法及其平均查找长度的计算方法。
2. 熟练掌握二叉排序树和平衡二叉树的构造和查找方法。
3. 理解B- 树的特点以及它们的建树和查找的过程。
4. 熟练掌握哈希表的构造方法, 深刻理解哈希表与其它结构的表的实质性的差别。

用分块查找法，对于有2000个数据项的表分成多少块最理想？每块的理想长度是多少？

在你的分块方式下平均查找长度是多少？

请写出一个在有序表中进行折半查找算法，要求返回有序表中小于或等于待查元素的最后一个元素的位置（提示：表中可能有重复的元素）。

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	5	7	7	12	12	31	31	31	31	31	40	40	50	50

已知关键字为1、2、3、4的四个节点，是回答下列问题：

（1）能构造出几种不同的二叉排序树？其中哪些是最优查找树（假设每个节点查找的概率相同）？

（2）能构造出几种不同的AVL树？

写一个算法将一棵二叉排序树分裂为两棵二叉排序树，使得其中一棵上所有节点的关键字都小于或等于x，而另一棵中树所有节点的关键字都大于x。假设分裂算法的定义如下：

int BSTree_Split(BST T, BST & T1, BST & T2, int x);

写一个算法判断给定的二叉树是否是平衡二叉树。

写一个算法判断给定的关键字序列k1, k2, …, kn是否为二叉排序树上查找过程中可能出现的关键字比较序列。

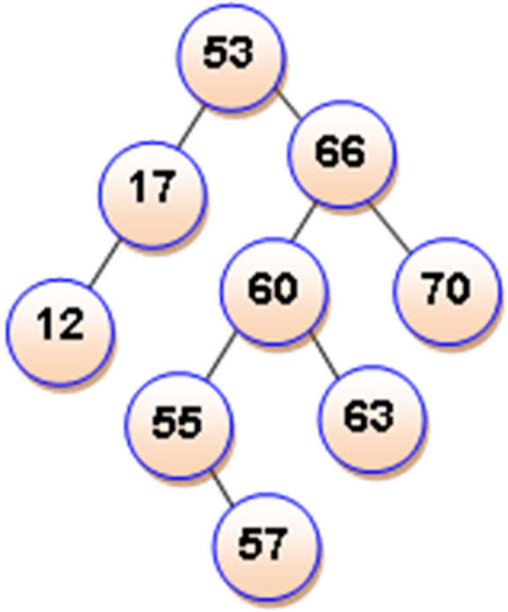
请写一个算法，将两棵二叉排序树合并为一棵二叉排序树。

在一棵二叉排序树S中，任意一条从根节点到叶子节点的路径path将S中的所有节点划分为三个集合

在path左边的节点组成集合S1，

在path上的节点组成集合S2，

在path右边的节点组成集合S3。对于任意属于S1的元素a，任意属于S2的元素b和任意属于S3的元素c，是否总满足a<=b<=c？请证明你的结论.



设有一个关键字序列{53, 17, 12, 66, 20, 70, 63, 55, 60, 57, 56}，请完成下列各个小题：

按照上述顺序，通过依次插入关键字，构造一棵平衡二叉树。请画出每插入一个关键字后树的状态.

假设依次删除关键字66、63，请画出每插入一个关键字后树的状态

请按照下列顺序依次向一棵空的3阶B-树中插入关键字，逐步建立一棵树。
20,30,50,52,60,68,70

（1）请画出该树一步步的创建过程。

（2）树建立后，依次删除50和68，请依次画出树变化后的状态

请回答下列有关B-树的问题：

（1）高度为5（除叶子层之外）的4阶B-树至少包含多少个关键字？至少有多少个节点？

（2）高度为5（除叶子层之外）的4阶B-树最多包含多少个关键字？最多有多少个节点？

（3）含9个叶子结点的3阶B-树中至少有多少个非叶子结点？含10个叶子结点的3阶B-树中至多有多少个非叶子结点？

已知一组记录的关键字为{25,40,33,47,12,66,72,87,94,22,5,58}，他们存储在散列表中，利用双散列函数解决冲突。要求向表中插入新数据的平均查找次数不超过3次。

（1）该散列表的大小应该设计为多大?

（2）设计散列函数（用除留余数法），并设计出现冲突时所需的再散列函数。

（3）应用你设计的双散列函数将上述关键字存储到散列表中。

红黑树

1. 红黑树(RBtree) 是一棵满足下述性质的近似平衡的二叉查找树，它能保证任何一个节点的左右子树的高度差小于两倍，具体规则如下：

① 每个结点要么是红色, 要么是黑色。

② 根结点是黑色的。

③ 所有叶子结点都是黑色的（实际上都是Null指针）。叶子结点不包含任何关键字信息, 所有查询关键字都在非终结点上。

④ 每个红色结点的子节点必须是黑色的。换句话说：从每个叶子到根的所有路径上不能有两个连续的红色结点

⑤ 从任一结点到NULL的所有路径都包含相同数目的黑色结点。

根据规则5，新增结点必须为红，根据规则4，新增节点的父结点必须为黑。

当新结点按照二叉排序树的搜索规则到达插入位置，如果不能满足这个条件，则需要改变颜色并旋转树形。

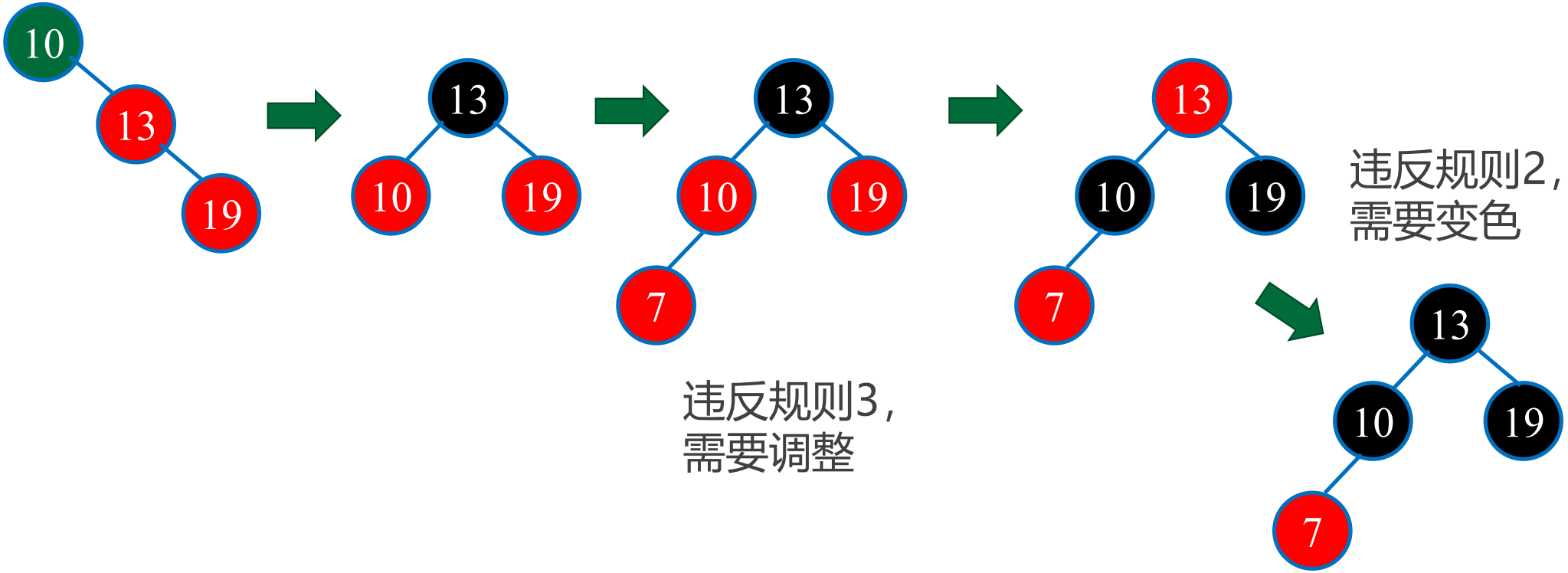
红黑树

- 例题
  按照如下顺序建立红黑树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21

如果插入的父结点为红结点，那么该父结点不可能为根结点，所以插入结点总是存在祖父结点。需要祖父节点参与旋转

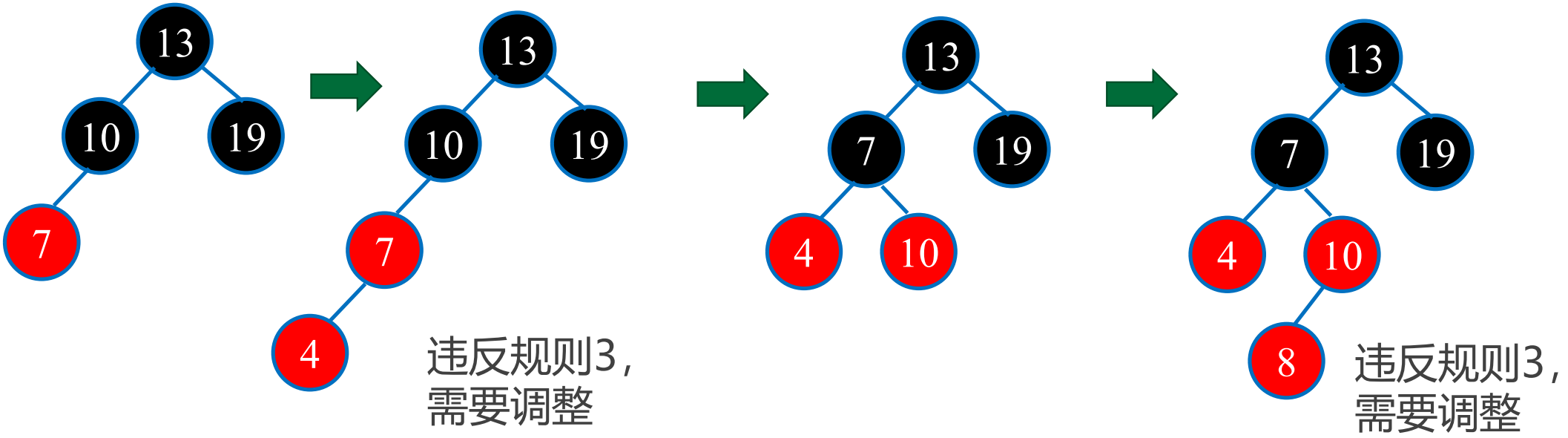
红黑树

- 例题
  按照如下顺序建立红黑树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21



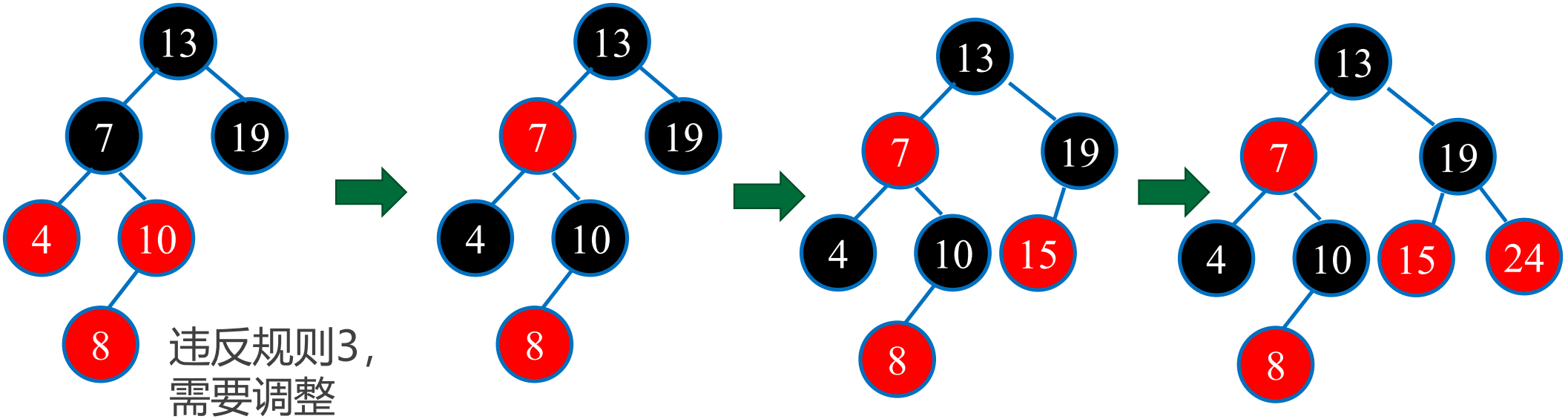
红黑树

- 例题
  按照如下顺序建立红黑树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21



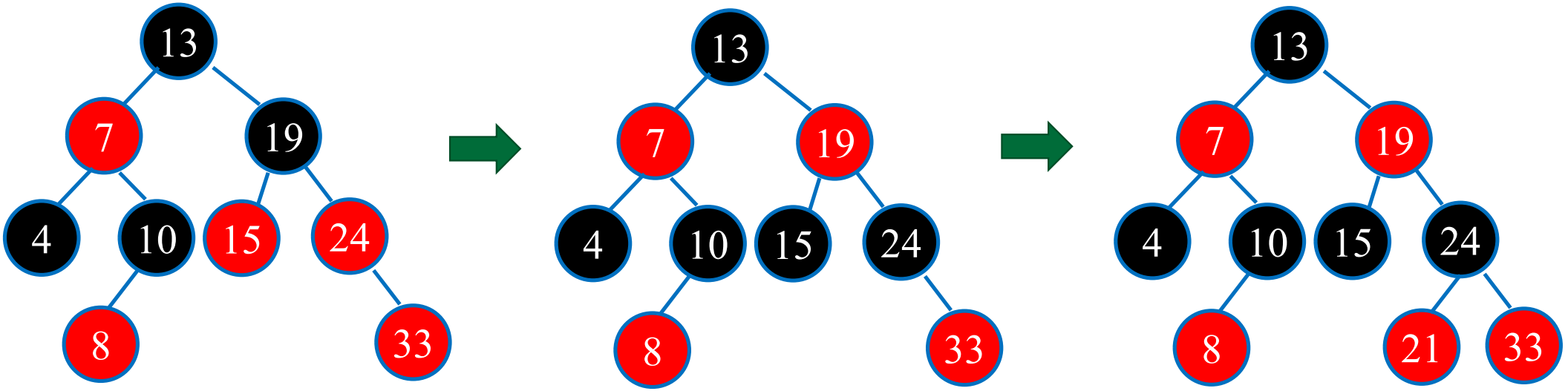
红黑树

- 例题
  按照如下顺序建立红黑树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21



红黑树

- 例题
  按照如下顺序建立红黑树：10, 13, 19, 7, 4, 8, 15, 24, 33, 21



红黑树

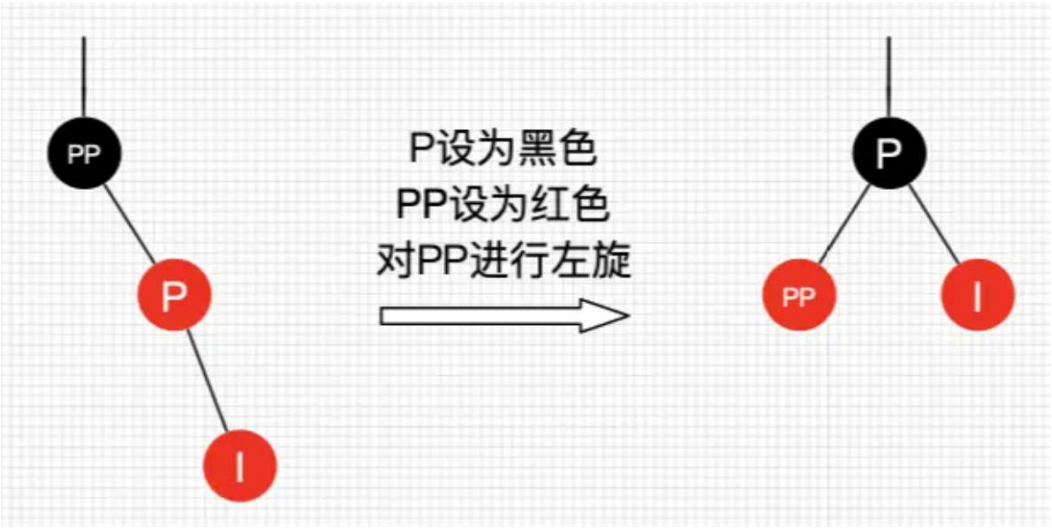
插入情景1：红黑树为空树
最简单的一种情景，直接把插入结点作为根结点就行，但注意，根据红黑树性质2：根节点是黑色。还需要把插入结点设为黑色。

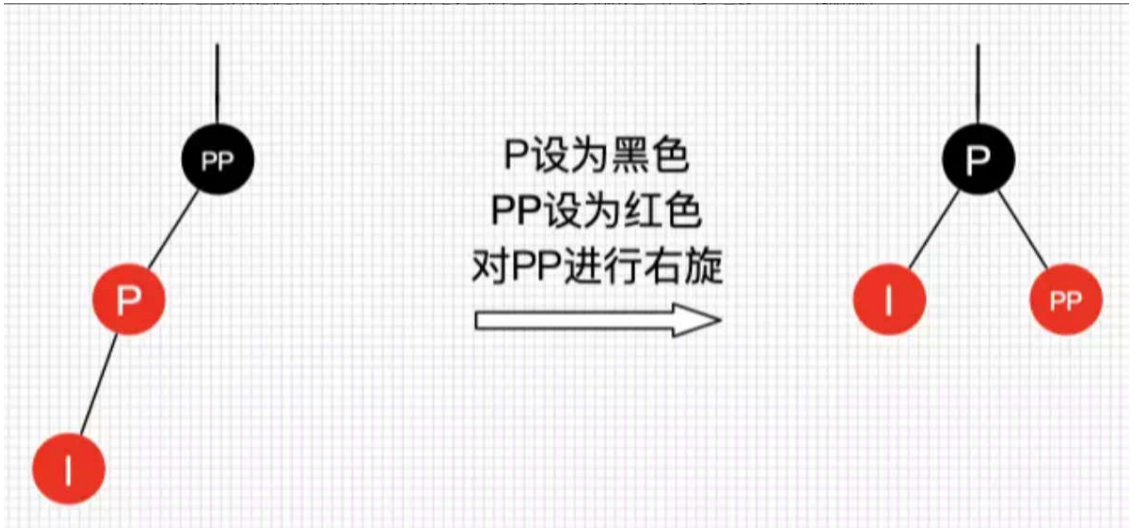
插入情景2：插入结点的父结点为黑结点
由于插入的结点是红色的，当插入结点的黑色时，并不会影响红黑树的平衡，直接插入即可，无需做自平衡。

插入情景3：插入结点的父结点为红结点
如果插入的父结点为红结点，那么该父结点不可能为根结点，所以插入结点总是存在祖父结点。这点很重要，因为后续的旋转操作肯定需要祖父结点的参与。

红黑树

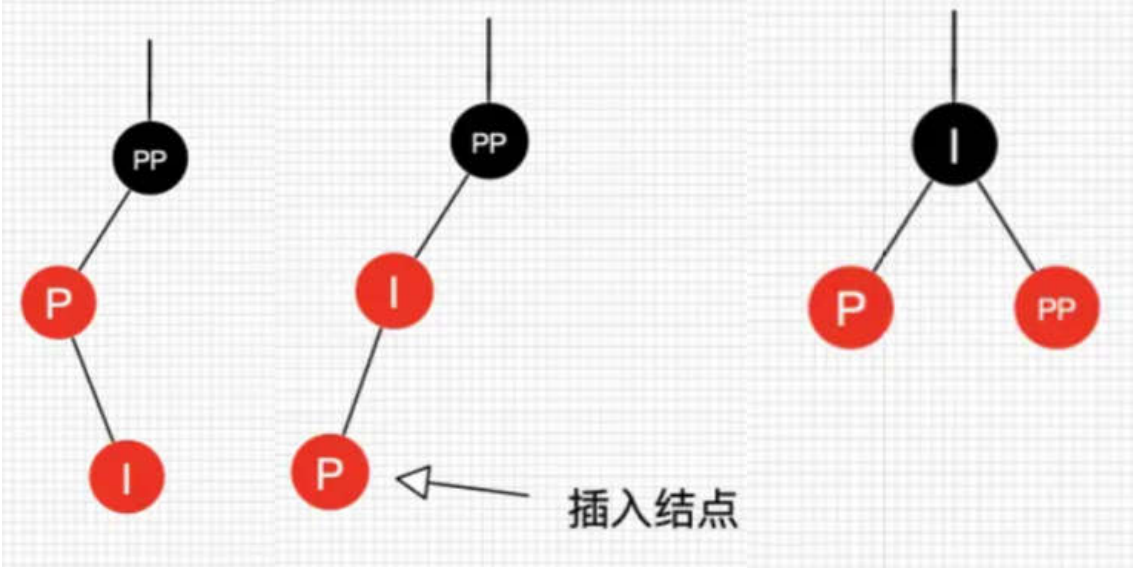
插入情景3.1：叔叔结点为黑色结点，且新增节点从外侧插入，此时应该先做一次单旋转，再改变父节点和祖父节点的颜色。

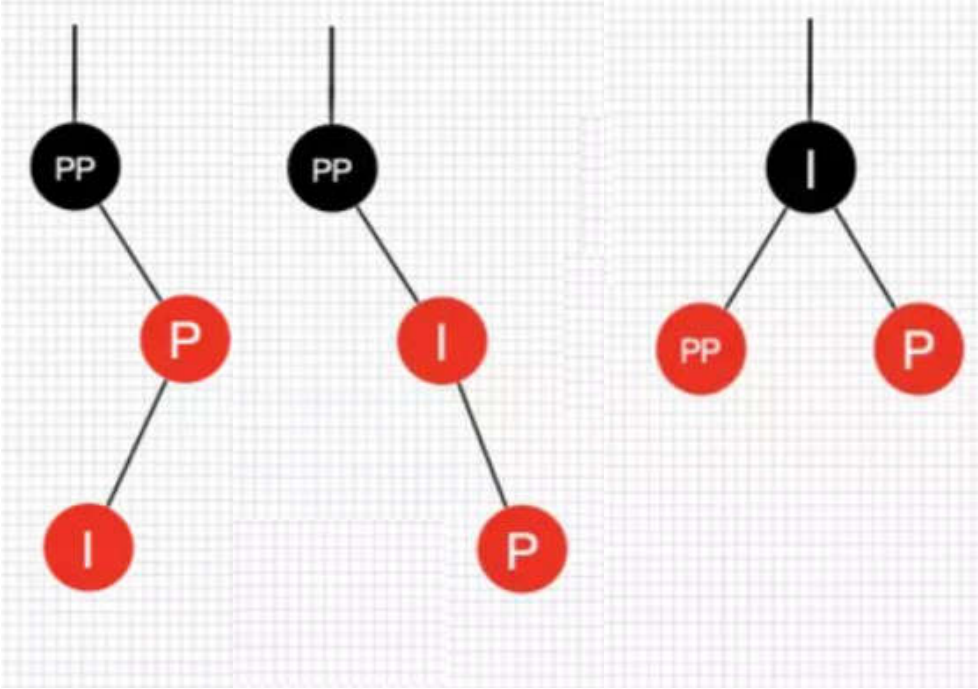




红黑树

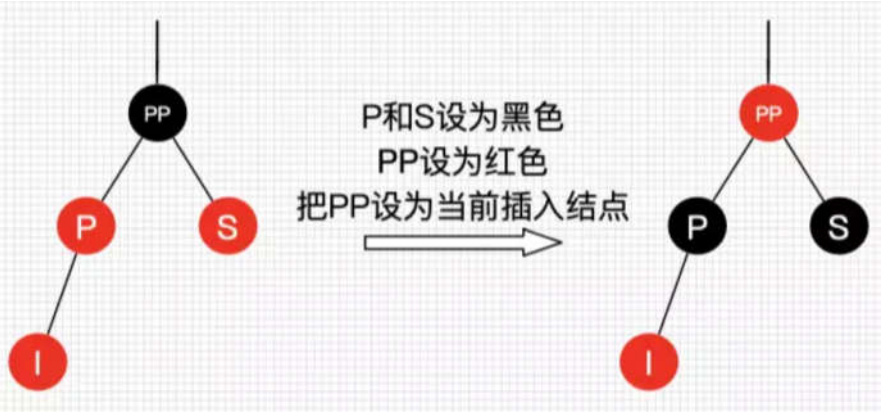
插入情景3.2：叔叔结点为黑色结点，且新增节点从内侧插入，此时应该先做一次单旋转，形成3.1的情况，继而再做一次旋转，并改变父节点和祖父节点的颜色。





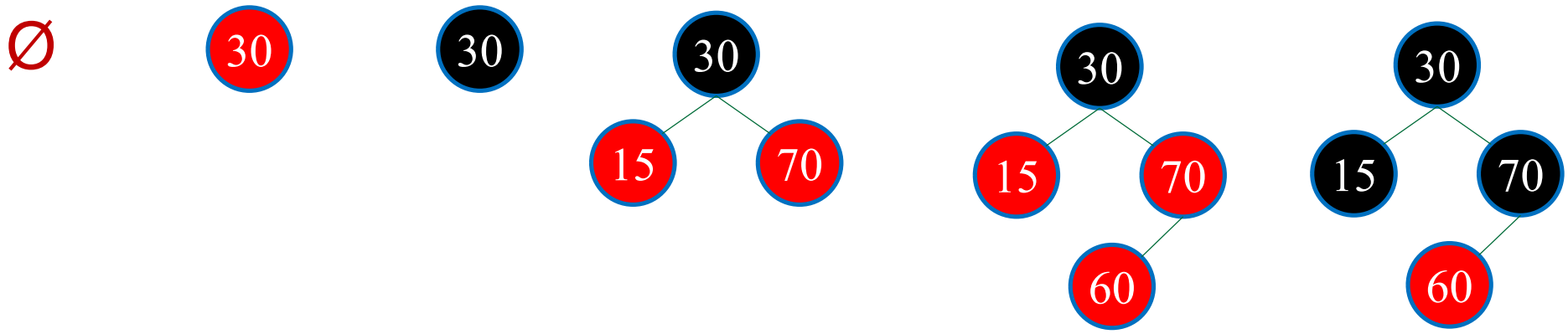
红黑树

插入情景3.3：插入结点的叔叔节点为红
处理：
•在从根到PP的路径上，将所有子节点全为红色的父节点变红，子节点变黑
•如果依旧发生颜色冲突，则按照3.1或3.2旋转



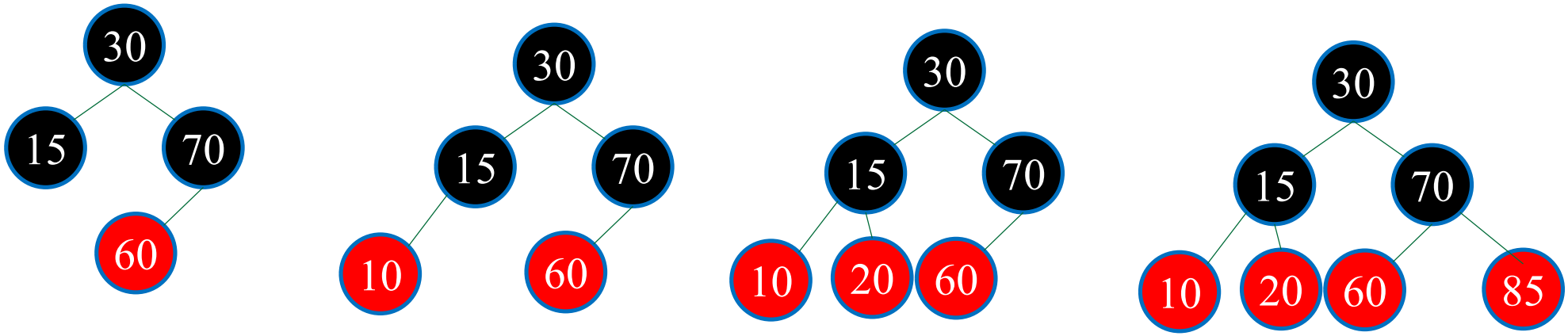
红黑树

- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90



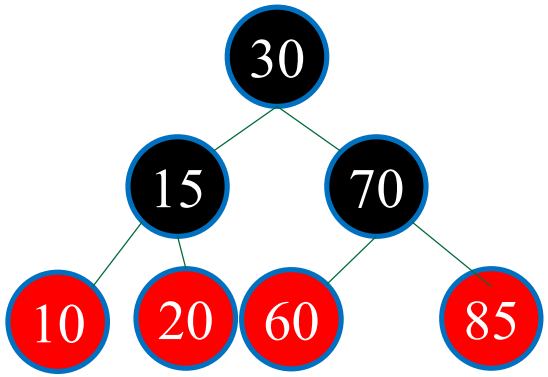
红黑树

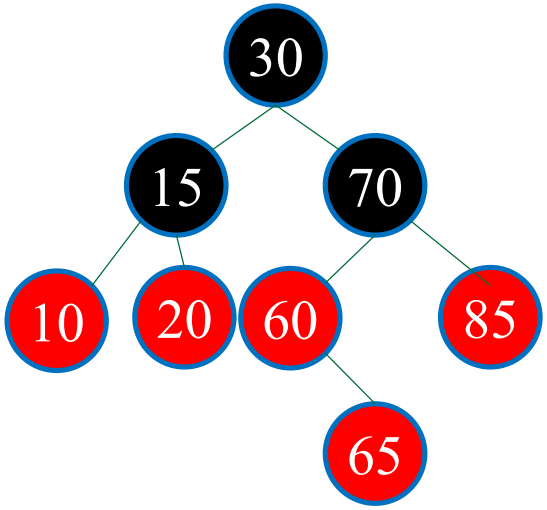
- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90

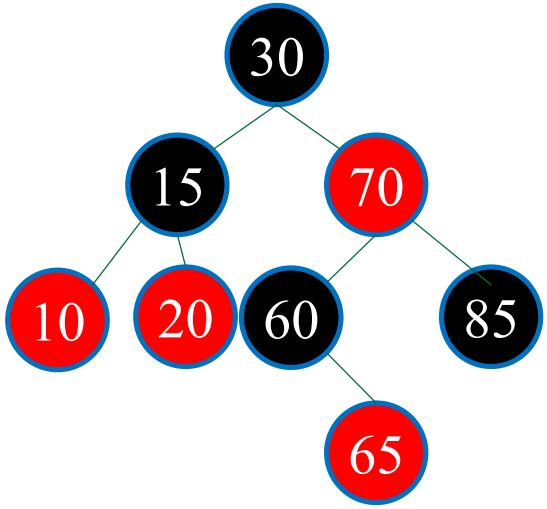


红黑树

- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90

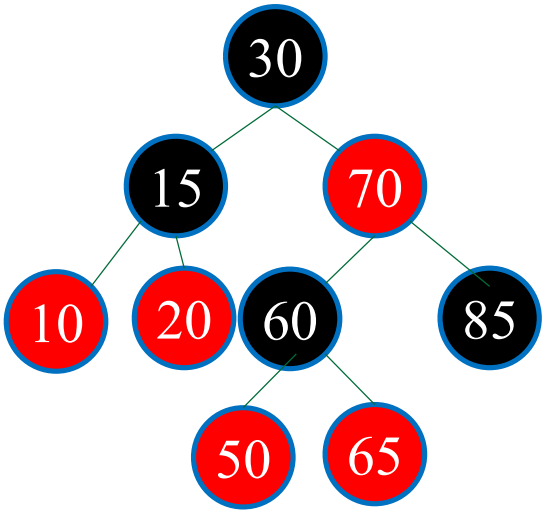


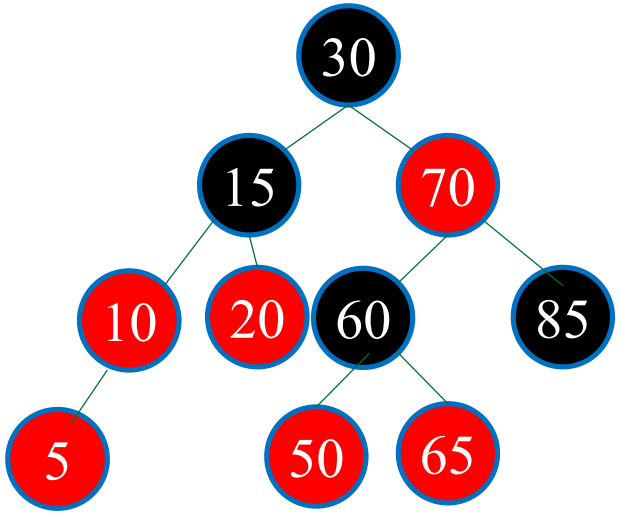


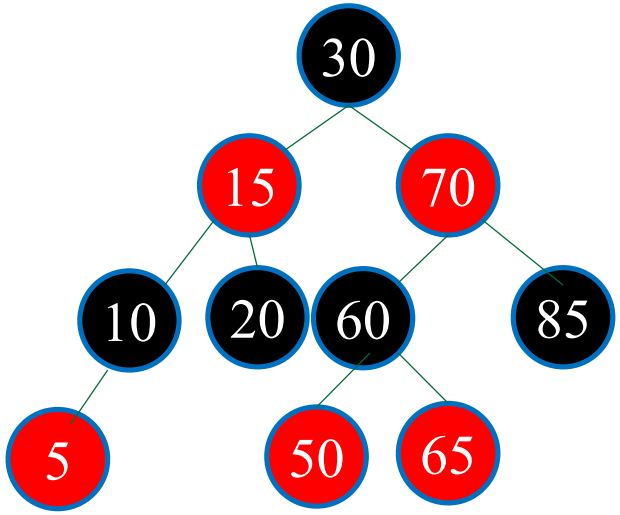


红黑树

- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90

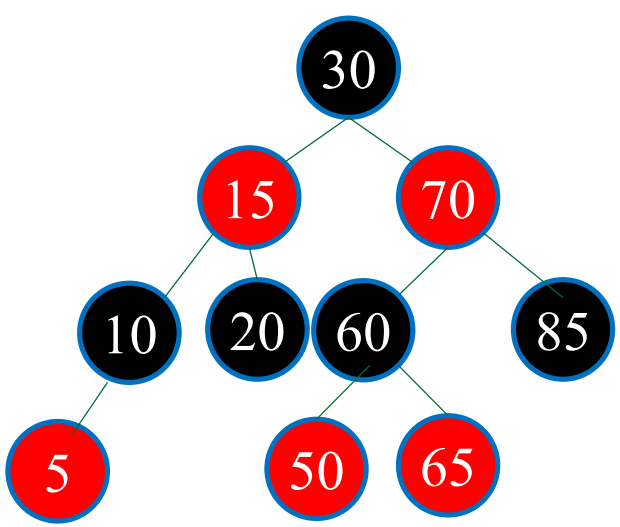


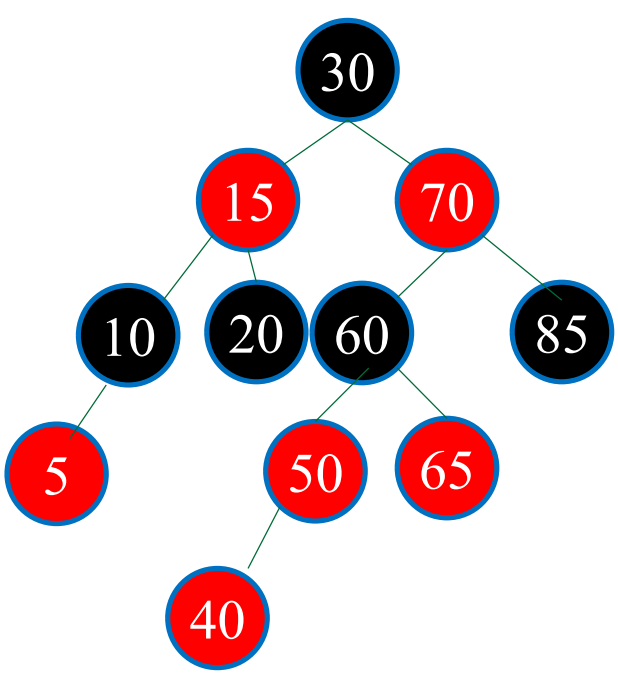


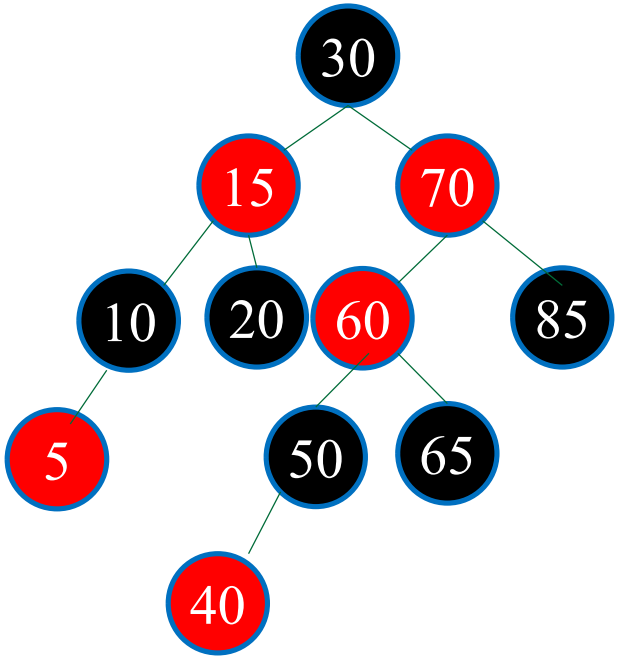


红黑树

- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90

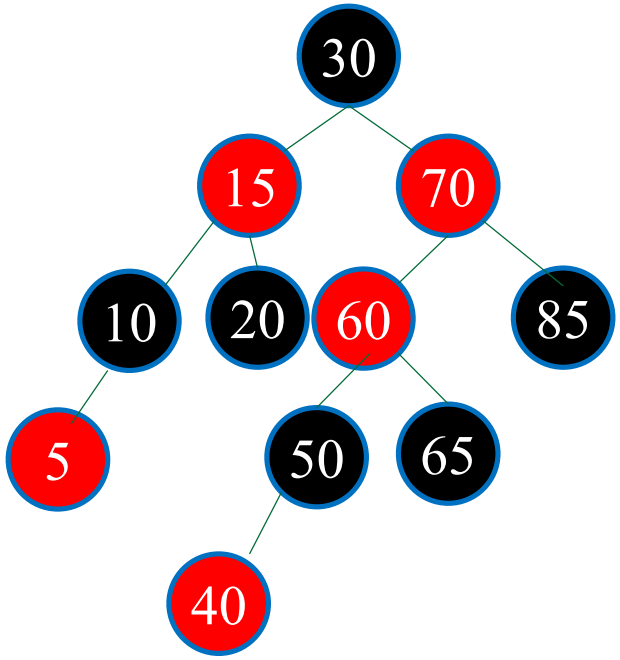


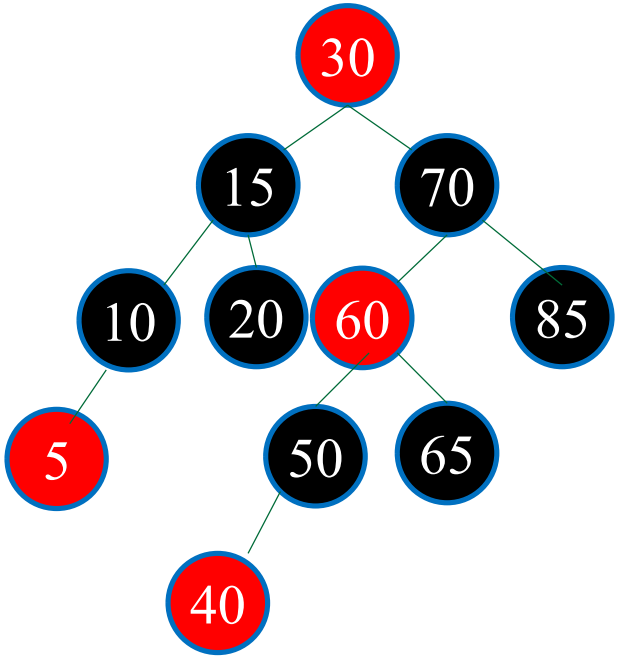


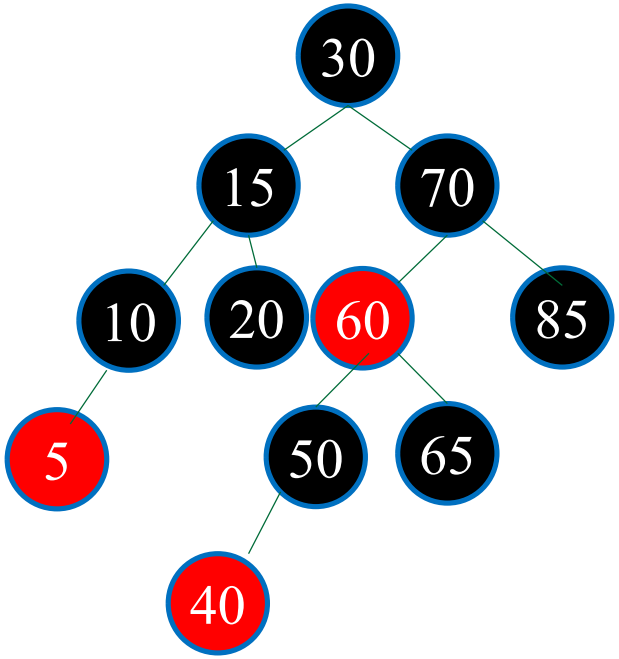


红黑树

- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90

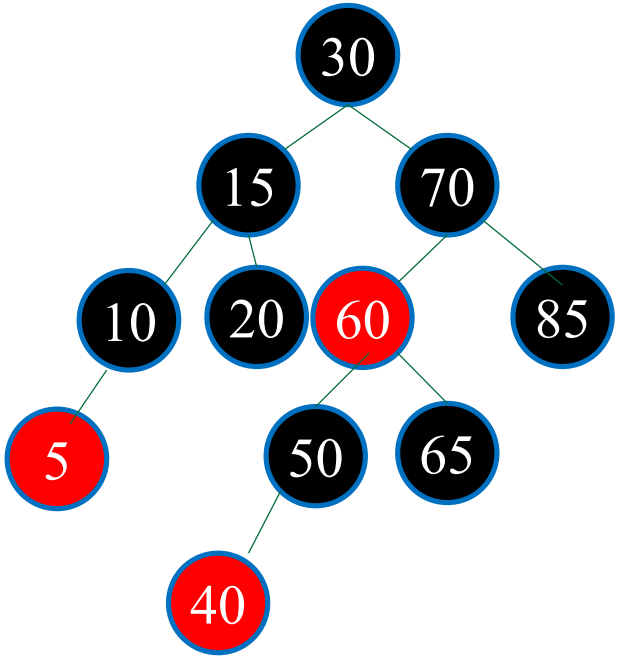


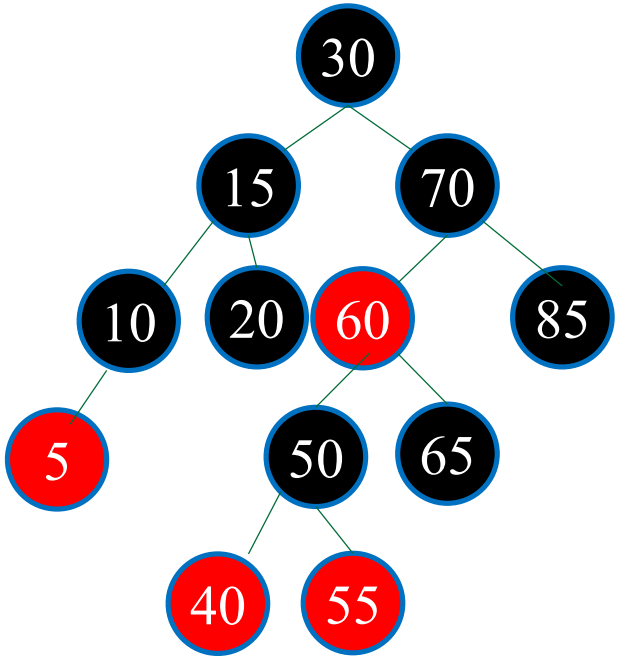


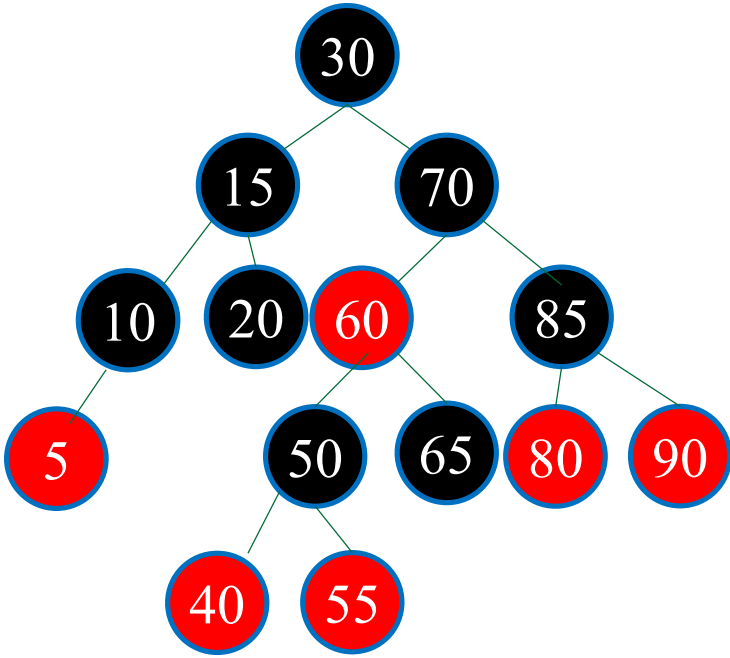


红黑树

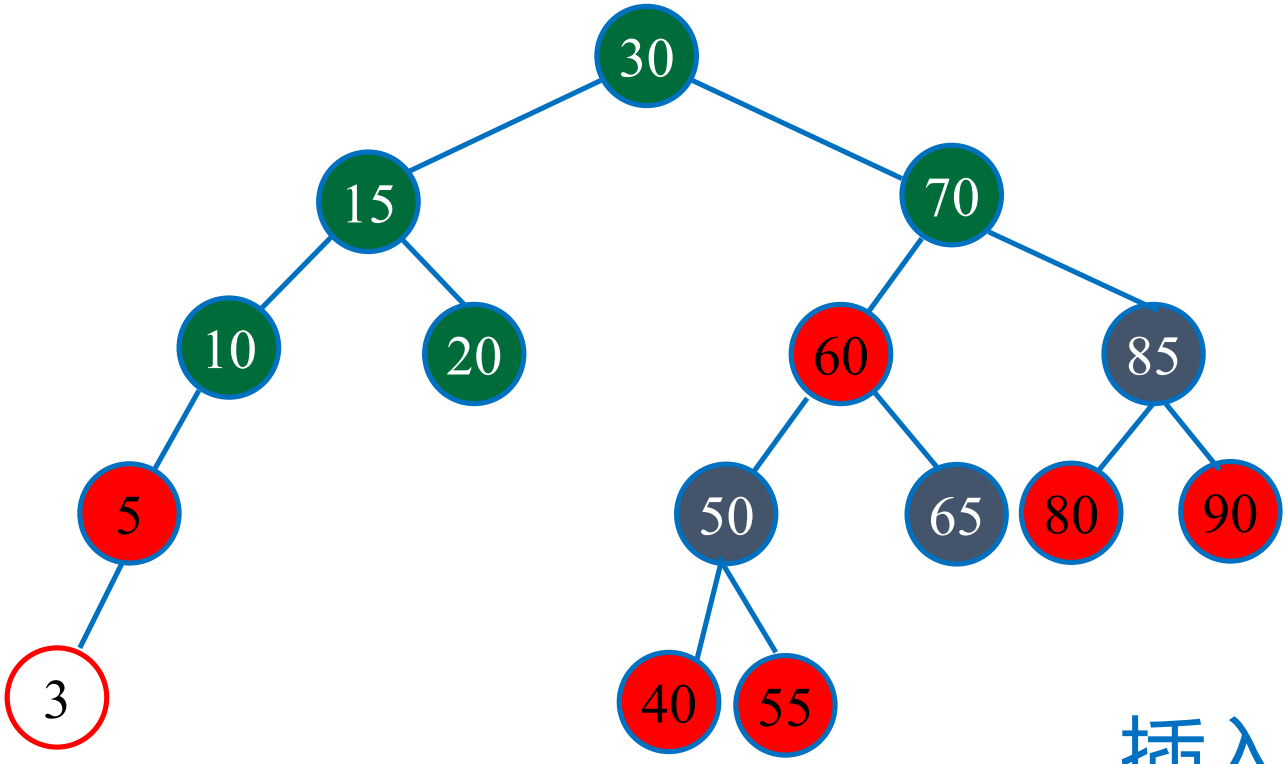
- 例题
  按照如下顺序建立红黑树：30,15,70, 60,10,20,85,65,50,5,40,55,80,90





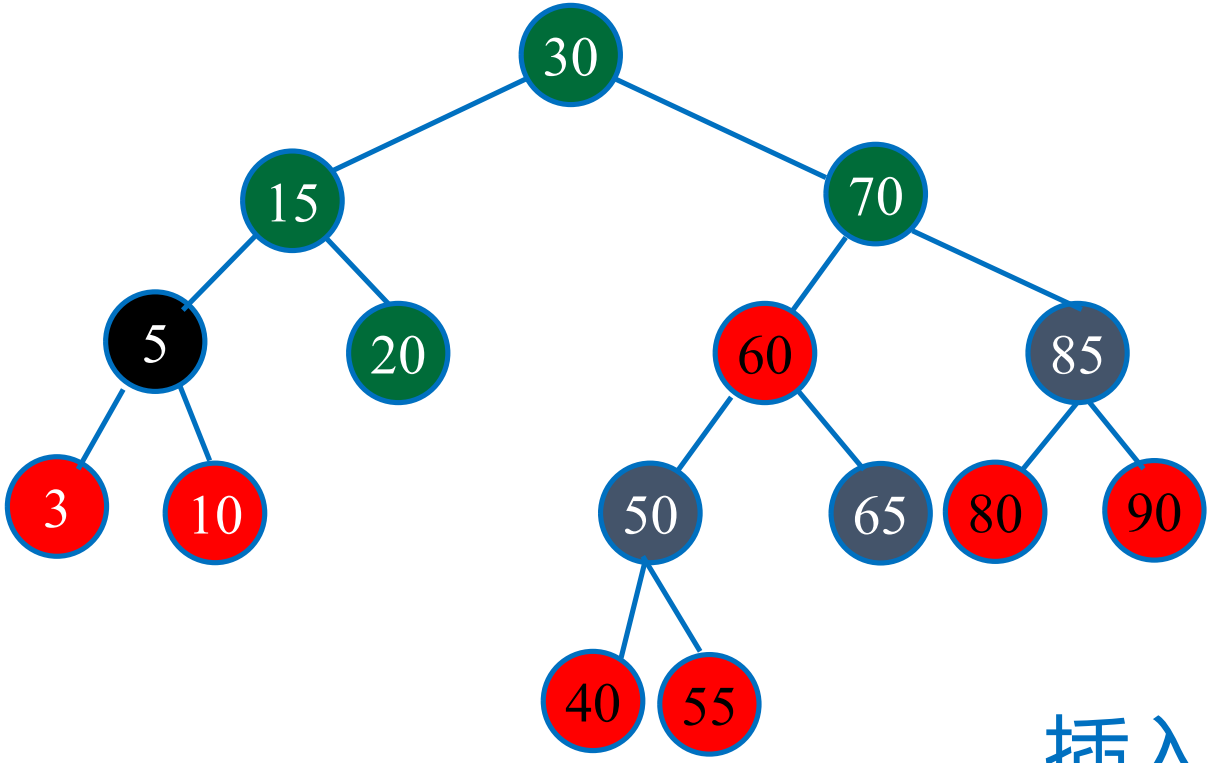


红黑树



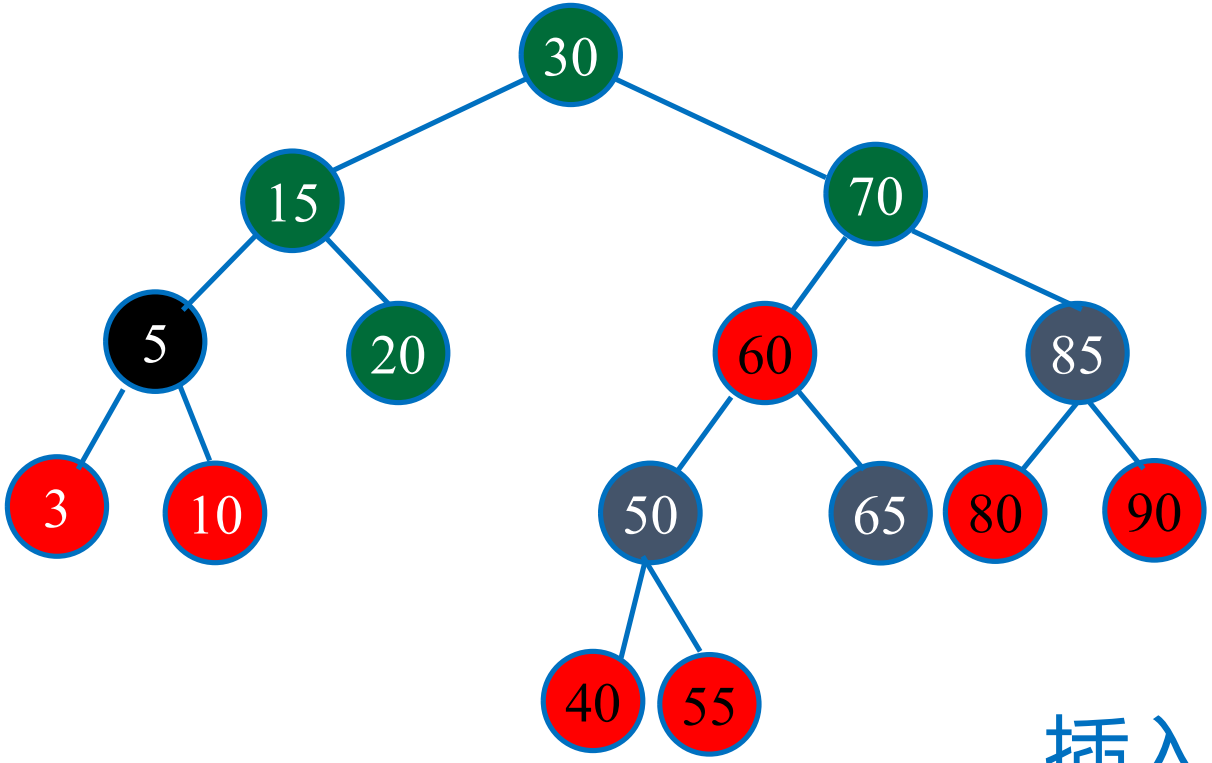
插入关键字 = 3

红黑树



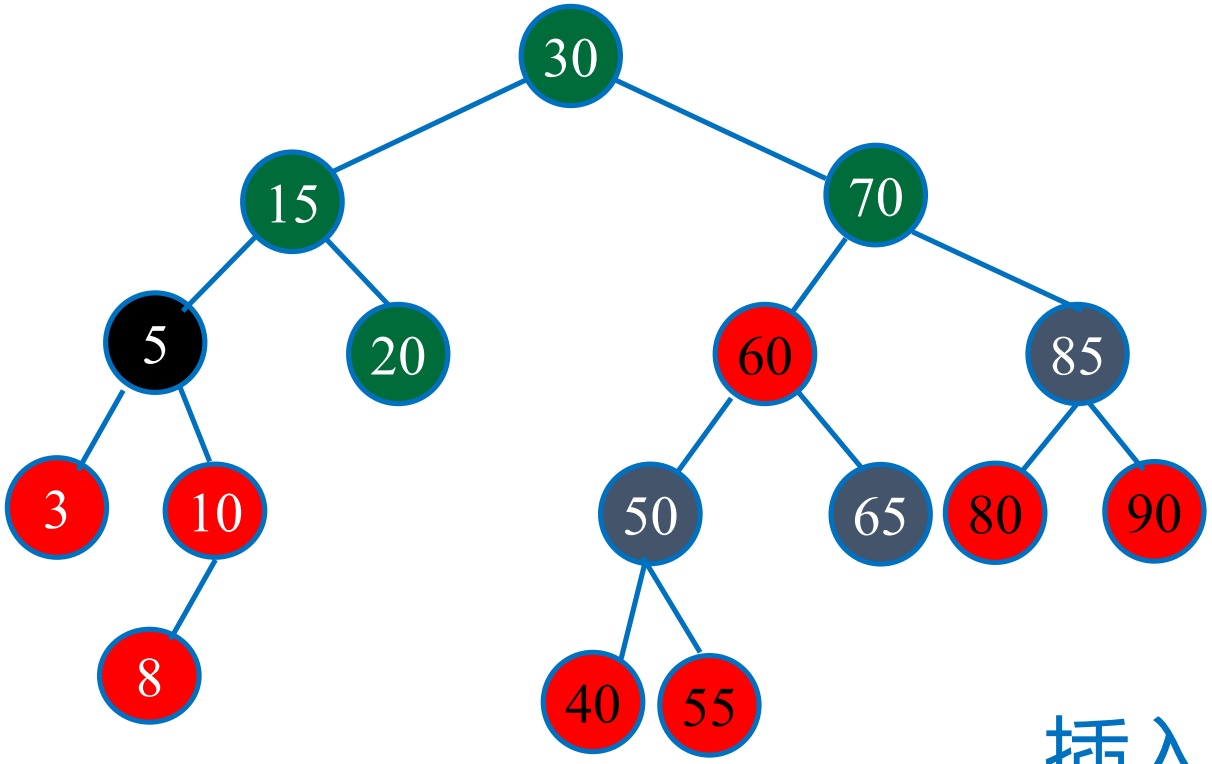
插入关键字 = 3

红黑树



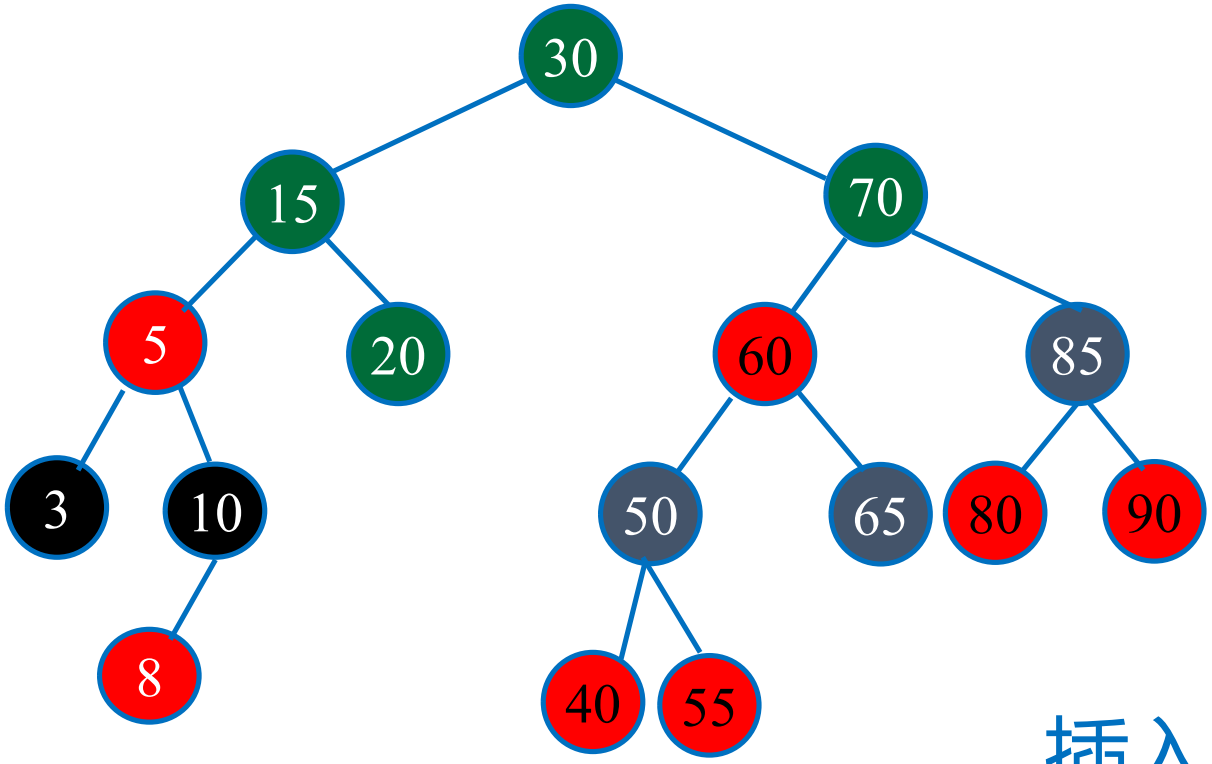
插入关键字 = 3,8

红黑树



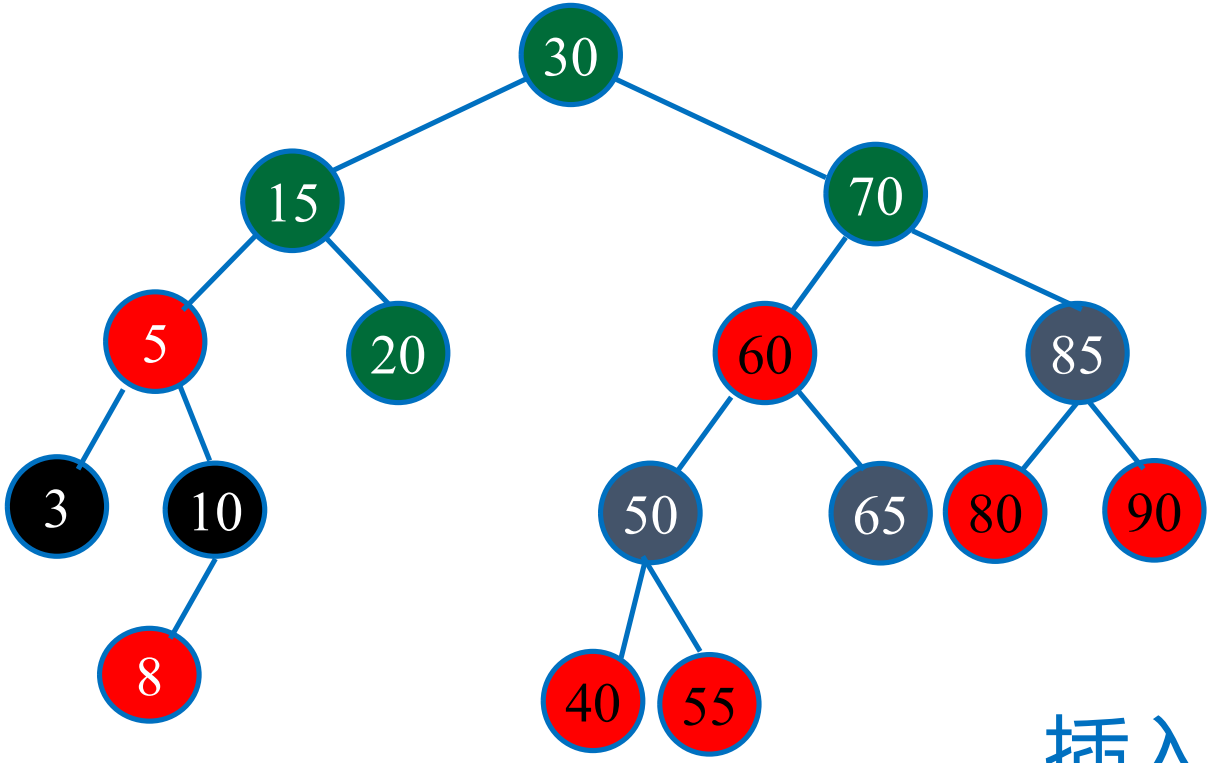
插入关键字 = 3,8

红黑树



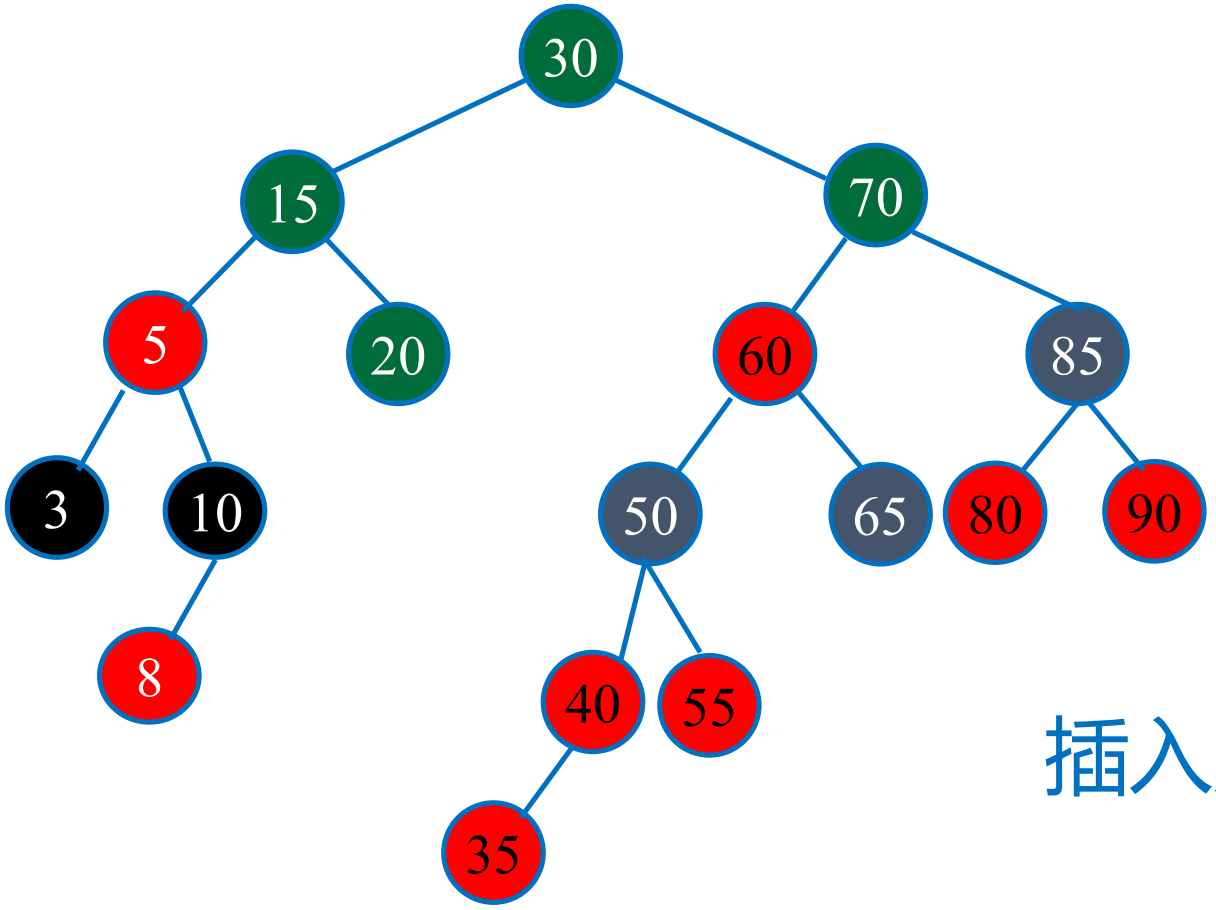
插入关键字 = 3,8

红黑树



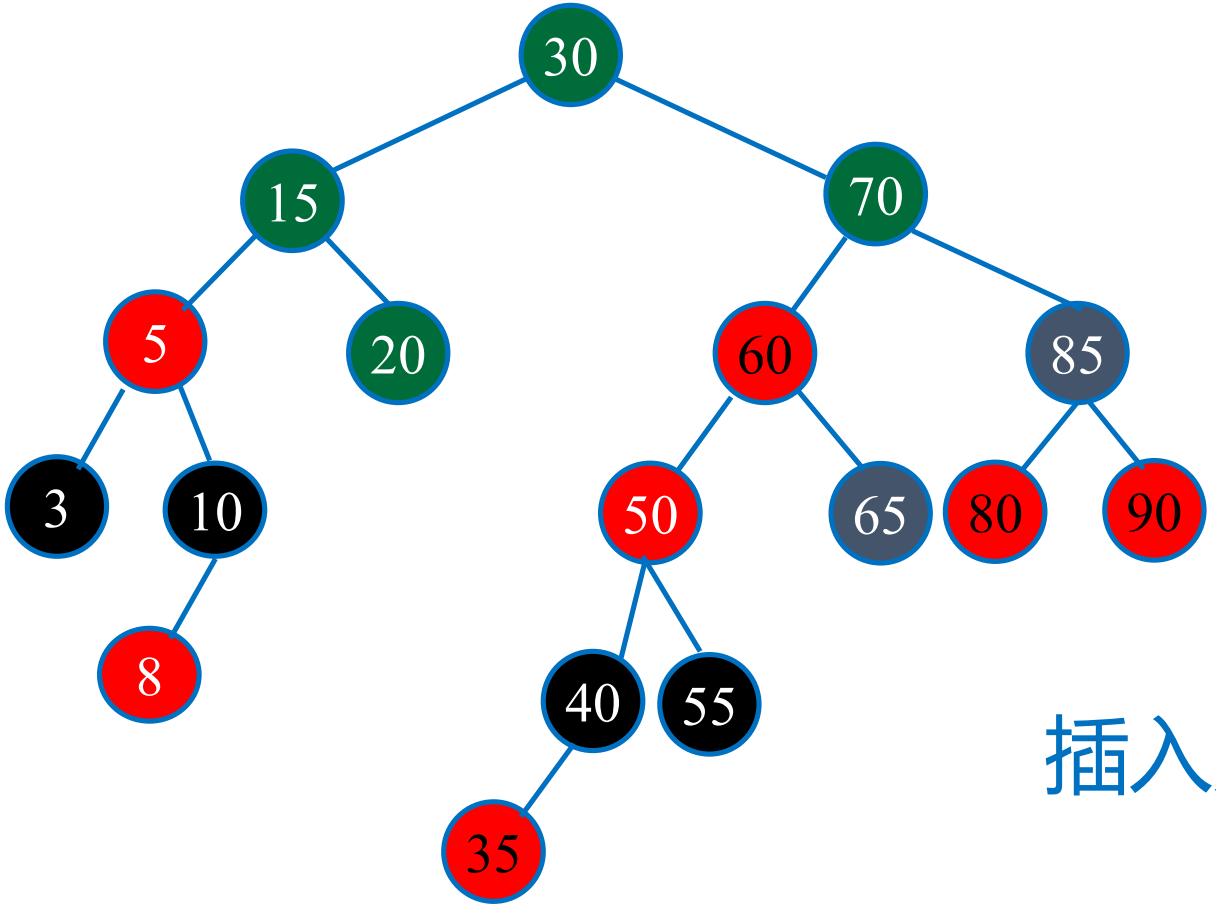
插入关键字 = 3,8,35

红黑树



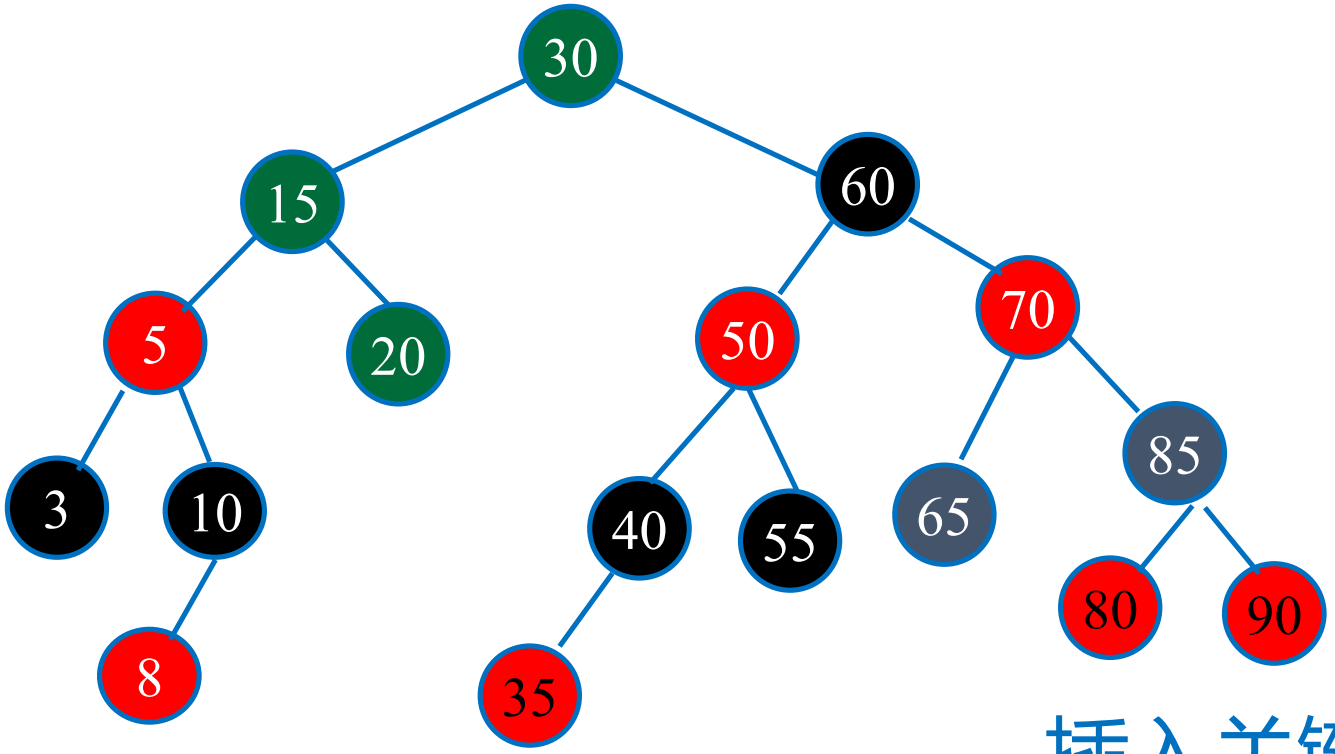
插入关键字 = 3,8,35

红黑树



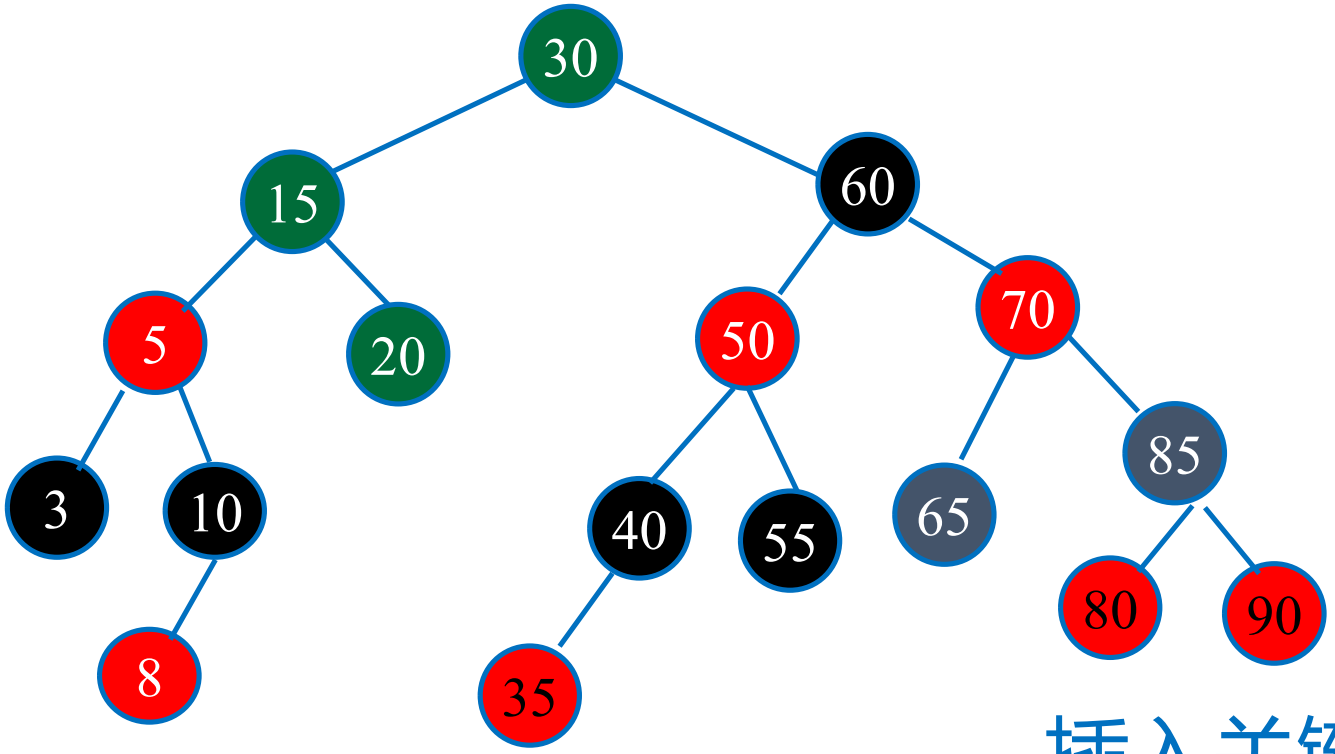
插入关键字 = 3,8,35

红黑树



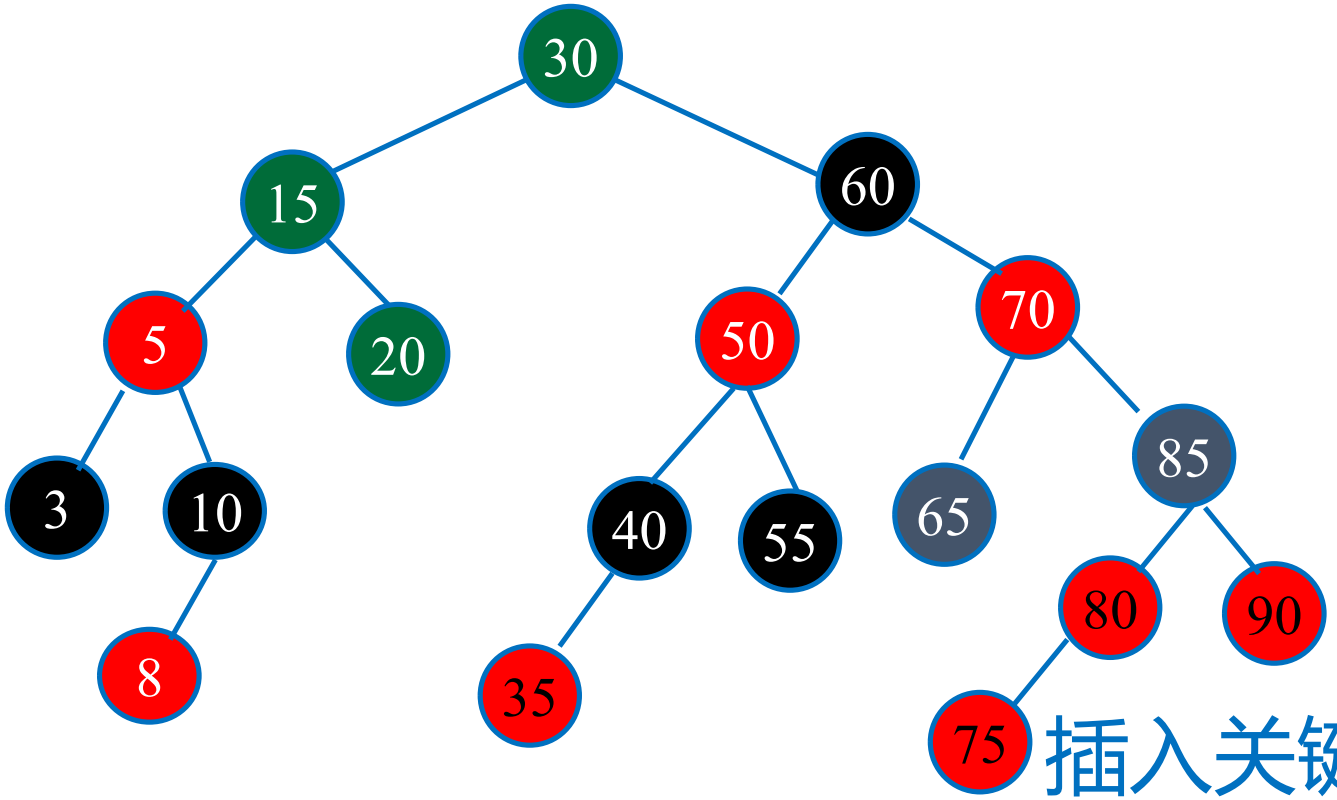
插入关键字 = 3,8,35

红黑树



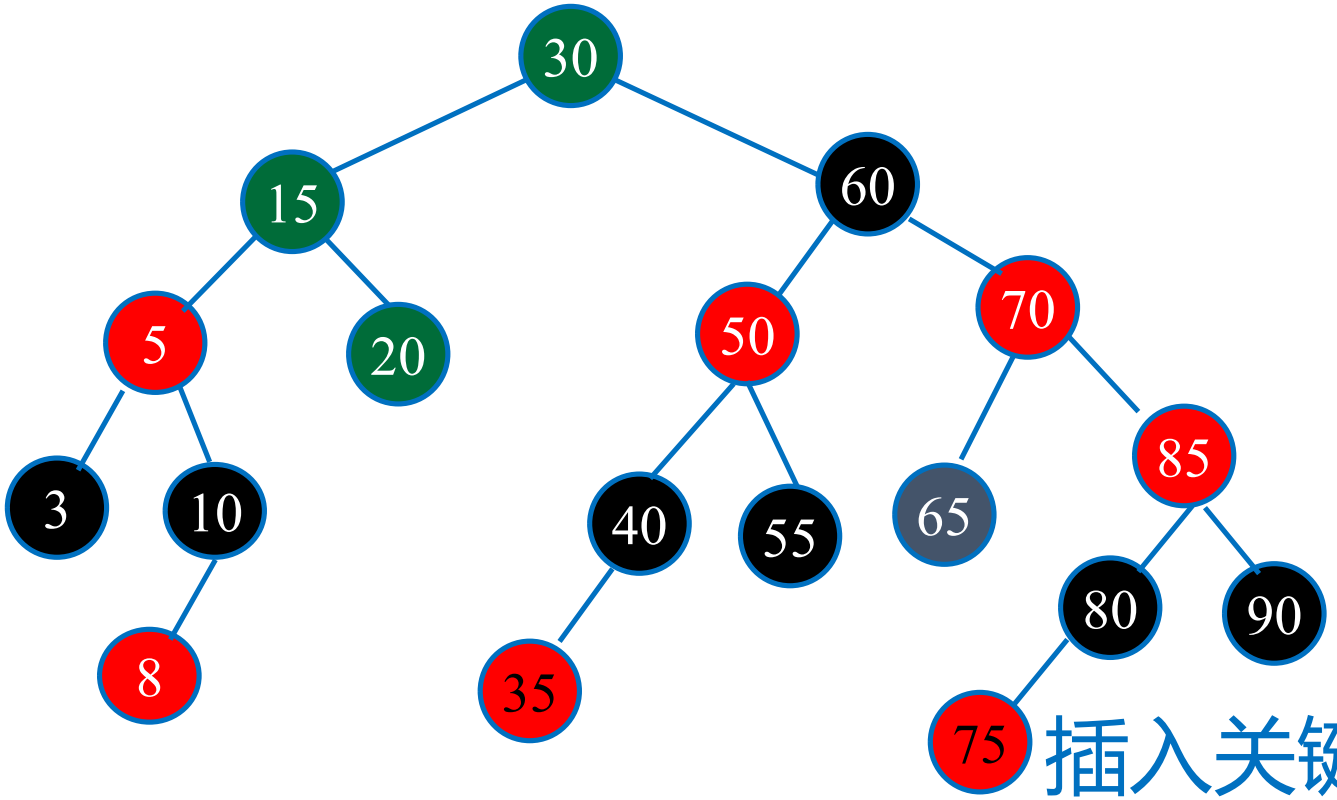
插入关键字 = 3,8,35,75

红黑树



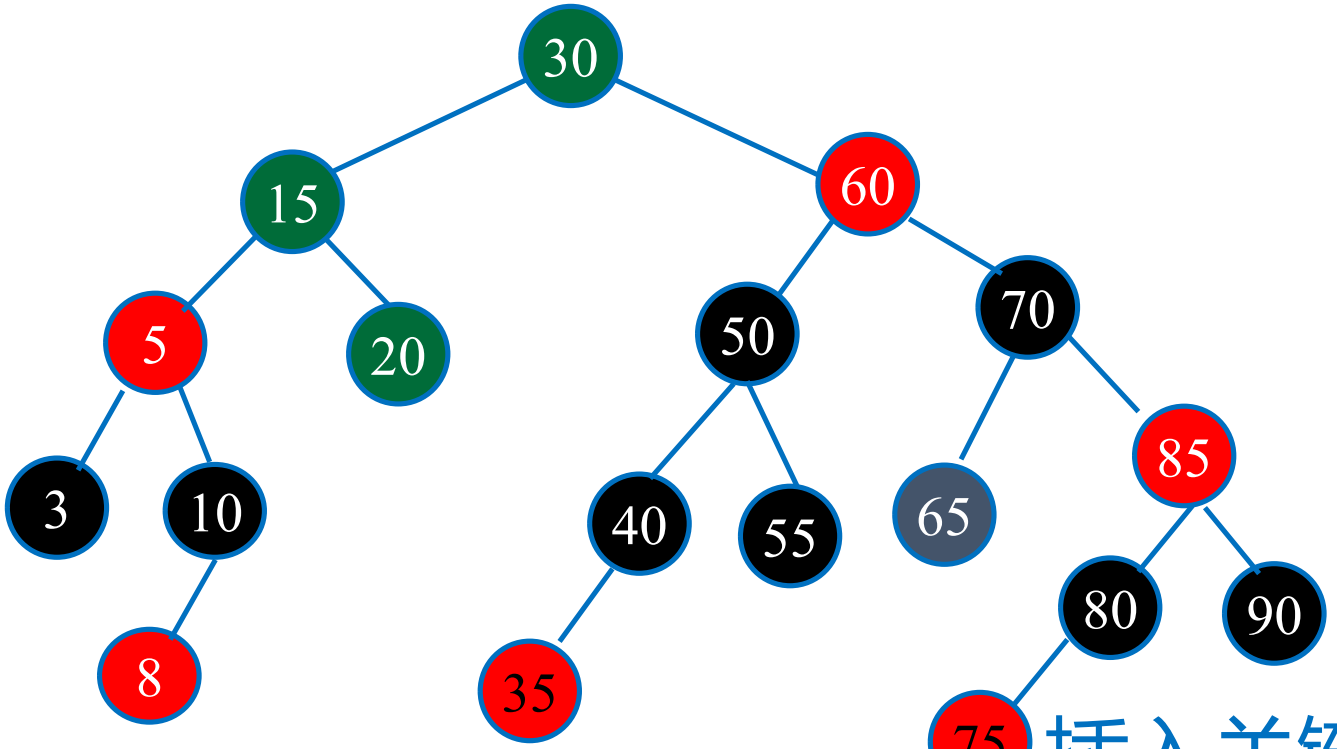
插入关键字 = 3,8,35,75

红黑树



插入关键字 = 3,8,35,75

红黑树



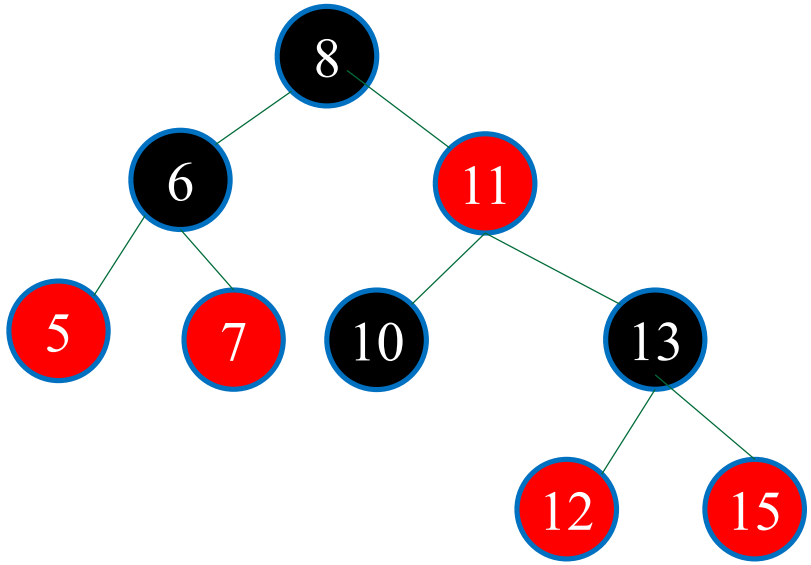
插入关键字 = 3,8,35,75

红黑树

- 例题
  按照如下顺序建立红黑树：10,7,8,15,5,6,11,13,12

红黑树

- 例题
  按照如下顺序建立红黑树：10,7,8,15,5,6,11,13,12



种类	具体实例
二叉树	二叉查找树、Van Emde Boas树、笛卡尔树、Top树、T-树
平衡二叉树	红黑树、平衡二叉查找树、AA树、伸展树、替罪羊树、Treap
B+树	B+树、B*树、UB+树、2-3树、（a,b）树，舞蹈树、H树、B^X-树
Tries	后缀树、基树、Ternary Search 树
二叉区分树	Quad树、OC树、KD-树、VP-树
非二叉树	指数树、聚合树、区间树、PQ树、SPQR树
图形树	R-树、X-树、段树
其他树	堆哈希树、手指树、度量树、覆盖树、BK树、双链树、最小期望树

扩展

B*-tree是**B⁺-tree**的变体，在**B⁺**树的基础上(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字记录的指针)，B*树中非根和非叶子结点再增加指向兄弟的指针；B*树定义了非叶子结点关键字个数至少为(2/3)*M，即块的最低使用率为2/3.
B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针。
所以，B*树分配新结点的概率比B+树要低，空间使用率更高；

In computer science, a **dancing tree** is a tree data structure similar to B+ trees. It was invented by Hans Reiser, for use by the Reiser4 file system. As opposed to self-balancing binary search trees that attempt to keep their nodes balanced at all times, dancing trees only balance their nodes when flushing data to a disk (either because of memory constraints or because a transaction has completed)