信息与软件工程学院

程序设计与算法基础II

主讲教师：陈安龙

第8章 查找

8.1 查找的概念

8.2 线性表的查找

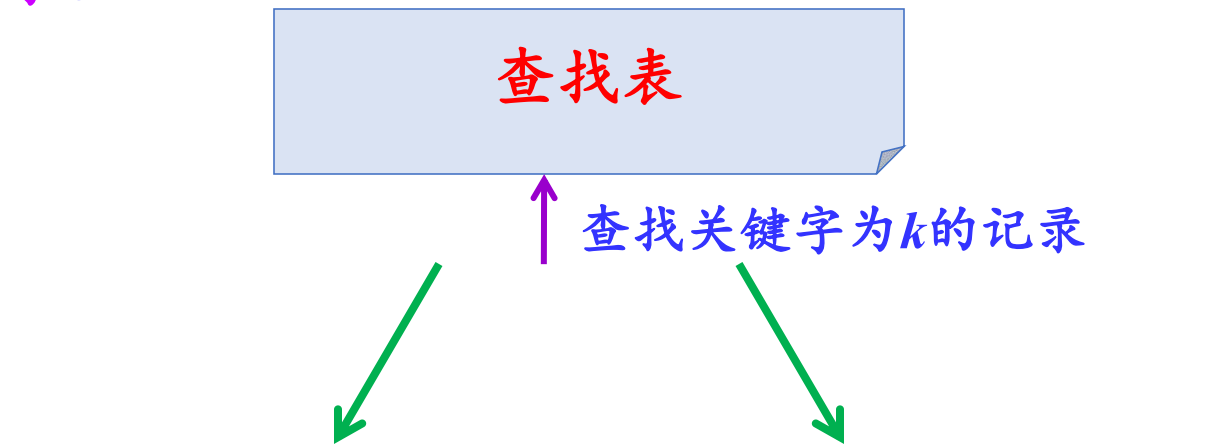
8.3 基于树的查找

8.4 哈希表的查找

8.1 查找的概念

1、查找的定义

查找表：是由一组记录组成的表或文件，而每个记录由若干个数据项组成，并假设每个记录都有一个能唯一标识该记录的关键字。



2、内查找和外查找

若整个查找过程都在内存进行，则称之为内查找；反之，若查找过程中需要访问外存，则称之为外查找。

3、查找的数据组织

采用何种存储结构？

（1）顺序表

（2）链表

（3）其他

若在查找的同时对表做修改操作（如插入和删除），则相应的表称之为动态查找表；否则称之为静态查找表。

4、影响查找的因素

采用何种查找方法?

（1）使用哪种数据结构来表示“表”，即表中记录是按何种方式组织的。

（2）表中关键字的次序。是对无序集合查找还是对有序集合查找?

5. 查找方法的性能指标

查找运算时间主要花费在关键字比较上，通常把查找过程中执行的关键字平均比较个数（也称为平均查找长度）作为衡量一个查找算法效率优劣的标准。

平均查找长度ASL（Average Search Length）定义为：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

n是查找表中记录的个数。p_i是查找第i个记录的概率，一般地，认为每个记录的查找概率相等，即$p_i=1/n$（$1 \leqslant i \leqslant n$），$c_i$是找到第$i$个记录所需进行的比较次数。

平均查找长度分为：

- 成功情况下的平均查找长度；
- 不成功情况（失败）下的平均查找长度。

查找表T：含有*n*个记录。

成功情况下（概率相等）的平均查找长度$ASL_{成功}$是指找到T中任一记录平均需要的关键字比较次数。

例如：

关键字	5	1	4	8	7	9	2	4	3
找到时的比较次数	1	2	3	4	5	6	7	8	9

$$ASL_{成功}=\frac{1+2+3+4+5+6+7+8+9}{9}=5$$

查找表T：含有n个记录。

不成功情况下的平均查找长度$ASL_{不成功}$是指查找失败（在T中未查找到）平均需要的关键字比较次数。

$\forall x \notin T$

通过关键字比较后确定不在T中

平均关键字比较次数

思考题

衡量查找算法性能的主要指标是什么？

8.2 线性表的查找

线性表查找的主要方法有：

（1）顺序查找

（2） 二分查找

（3）分块查找

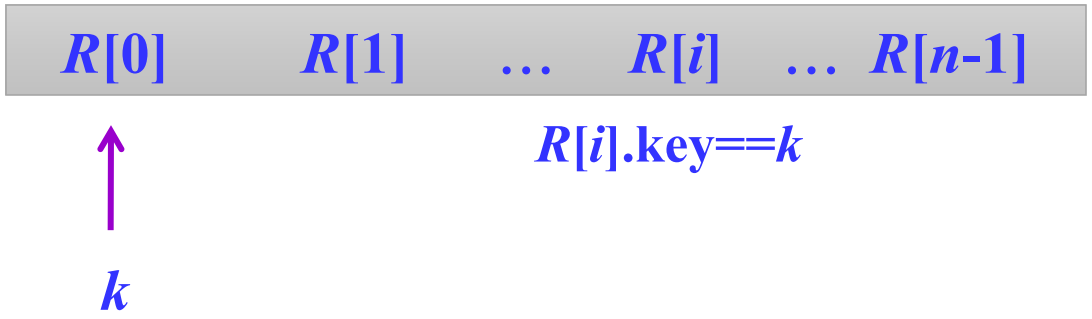
线性表有顺序和链式两种存储结构。这里介绍以顺序表作为存储结构时实现线性表的查找算法。定义被查找的顺序表类型定义如下：

```
#define MAXL <表中最多记录个数>
typedef struct
{    KeyType key;                    //KeyType为关键字的数据类型
     InfoType data;                  //其他数据项
} RecType;                           //查找顺序表元素类型
```

静态查找表

8.2.1 顺序查找

思路：从表的一端开始，顺序扫描线性表，依次将扫描到的关键字和给定值k相比较：

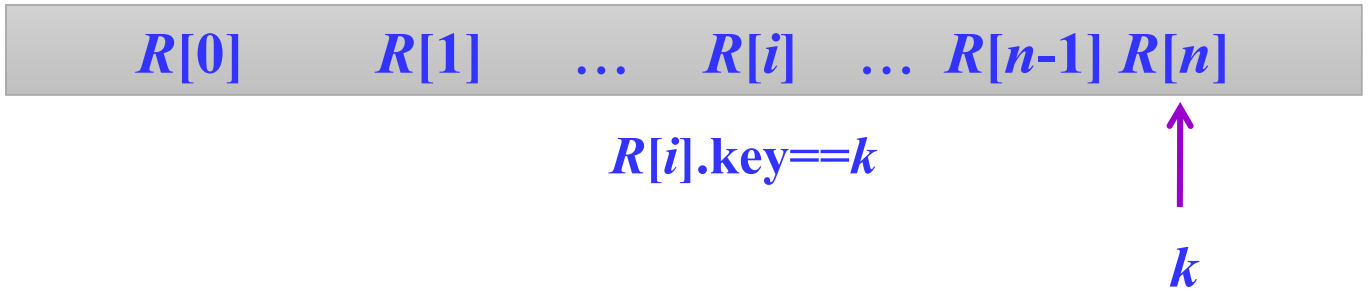


若当前扫描到的关键字与k相等，则查找成功；若扫描结束后，仍未找到关键字等于k的记录，则查找失败。

顺序查找的算法如下（在顺序表R[0..*n*−1]中查找关键字为*k*的元素，成功时返回找到的元素的逻辑序号，失败时返回0）：

```
int SeqSearch(RecType R[],int n,KeyType k)
{    int i=0;
     while (i<n && R[i].key!=k)      //从表头往后找
          i++;
     if (i>=n)                       //未找到返回0
          return 0;
     else
          return i+1;                //找到返回逻辑序号i+1
}
```

顺序查找另外一种思路：定义存放n+1元素数组R,数据存在存在R[1]至R[n],R[0]用于临时存放待查找的数据*k*；从表的R[n]一端开始，顺序扫描线性表，依次将扫描到的关键字和给定值*k*相比较：



若当前扫描到的关键字与*k*相等，则查找成功；若扫描至R[0],则说明R[1]至R[n]未找到关键字等于*k*的记录，则查找失败，返回位置0。

顺序查找的算法如下（在顺序表R[0..*n*]中查找关键字为*k*的元素，成功时返回找到的元素的逻辑序号，失败时返回0）：

```
int SeqSearch(RecType R[],int n,KeyType k)
{    int i=n;
     for (int i=n;i>=0 && R[i].key!=k；i--)    //从表最后往前找
     return i;                                 //找到返回逻辑序号i
}
```

❶ 成功情况下的平均查找长度ASL

查找到表中第i个记录$R[i-1]$时，需比较i次。因此成功时的顺序查找的平均查找长度为：

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

查找成功时的平均比较次数约为表长的一半 。

❷ 不成功情况下的平均查找长度ASL

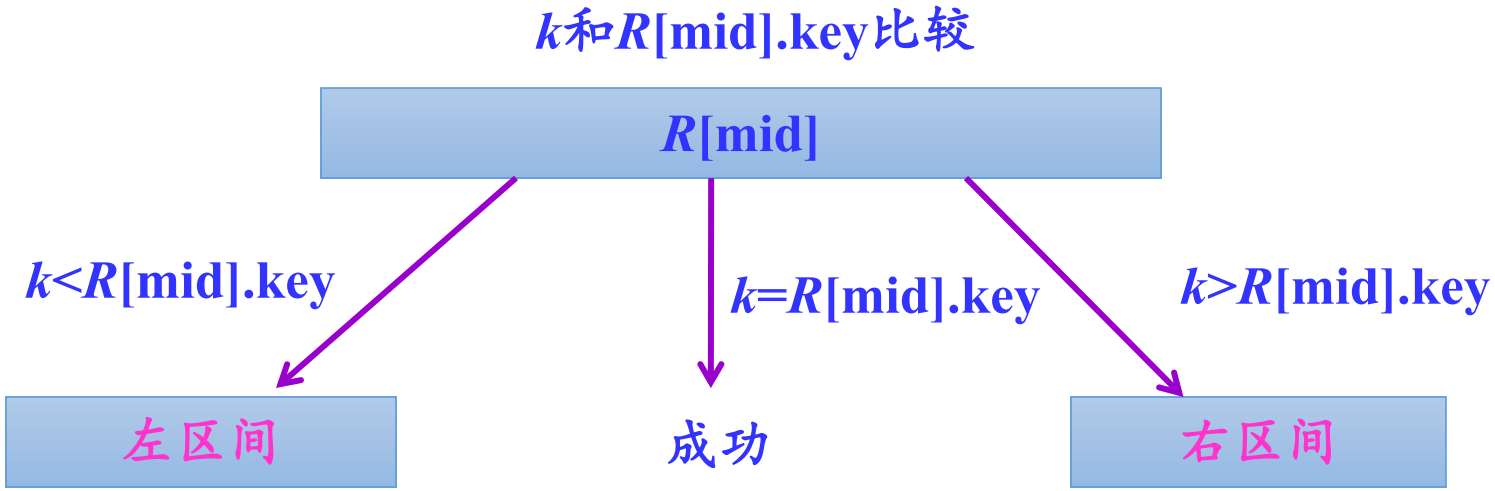
查找不成功情况时需要和表中所有元素都比较一次，所以，不成功时的平均查找长度为n。

顺序查找的时间复杂度为$O(n)$。

8.2.2 折半查找

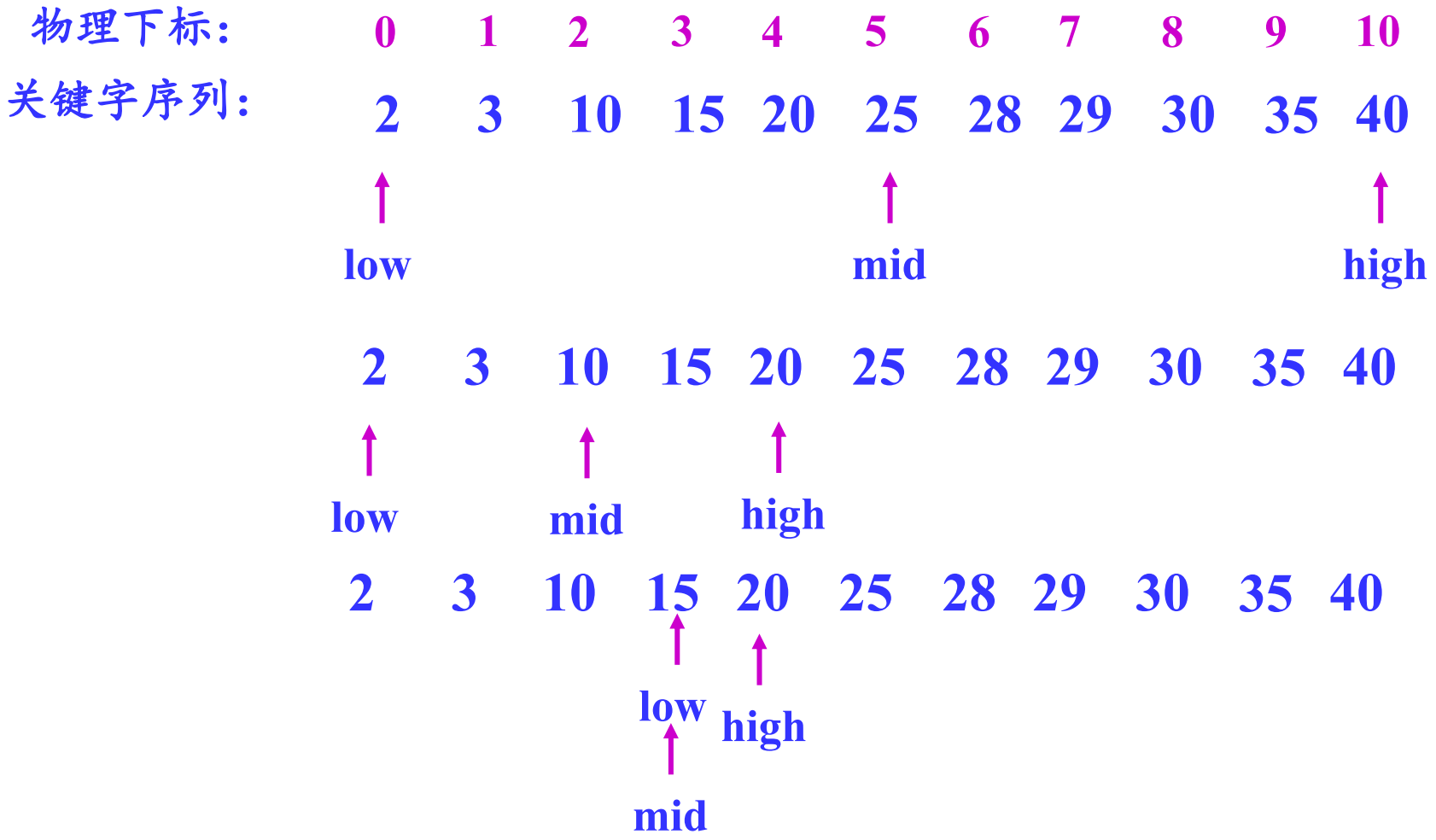
折半查找也称为二分查找，要求线性表中的记录必须已按关键字值有序（递增或递减）排列。

思路：



折半查找演示

例如，在关键字有序序列{2,3,10,15,20,25,28,29,30,35,40}中采用折半查找法查找关键字为15的元素。



查找成功，关键字为15的记录的逻辑序号为4关键字比较次数为3

其算法如下（在有序表R[0..*n*−1]中进行折半查找，成功时返回元素的逻辑序号，失败时返回0）：

```
int BinSearch(RecType R[],int n,KeyType k)
{
    int low=0,high=n-1,mid;
    while (low<=high)                  //当前区间存在元素时循环
    {   mid=(low+high)/2;
        if (R[mid].key==k)             //查找成功返回其逻辑序号mid+1
            return mid+1;
        if (k<R[mid].key)              //继续在R[low..mid-1]中查找
            high=mid-1;
        else
            low=mid+1;                 //继续在R[mid+1..high]中查找
    }
    return 0;
}
```

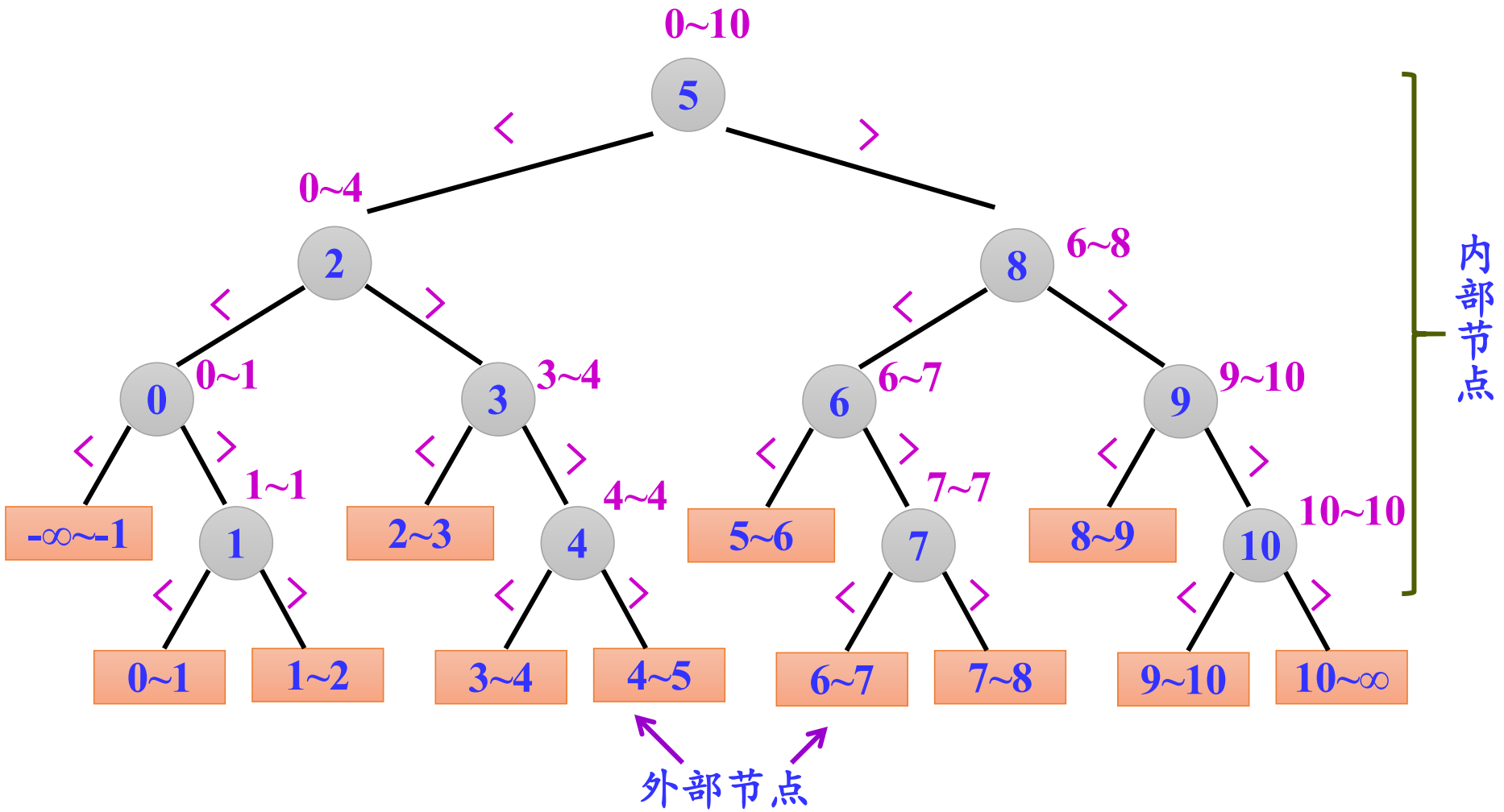
思考题

折半查找可以设计成递归算法，如何实现？

二分查找过程可用二叉树来描述：

- 把当前查找区间的中间位置上的记录作为根；
- 左子表和右子表中的记录分别作为根的左子树和右子树。

这样的二叉树称为判定树或比较树。



R[0..10]的二分查线的判定树（*n*=11）

外部节点即查找失败对应的节点，是虚拟的

*n*个关键字：内部节点为*n*个，外部节点为*n*+1个

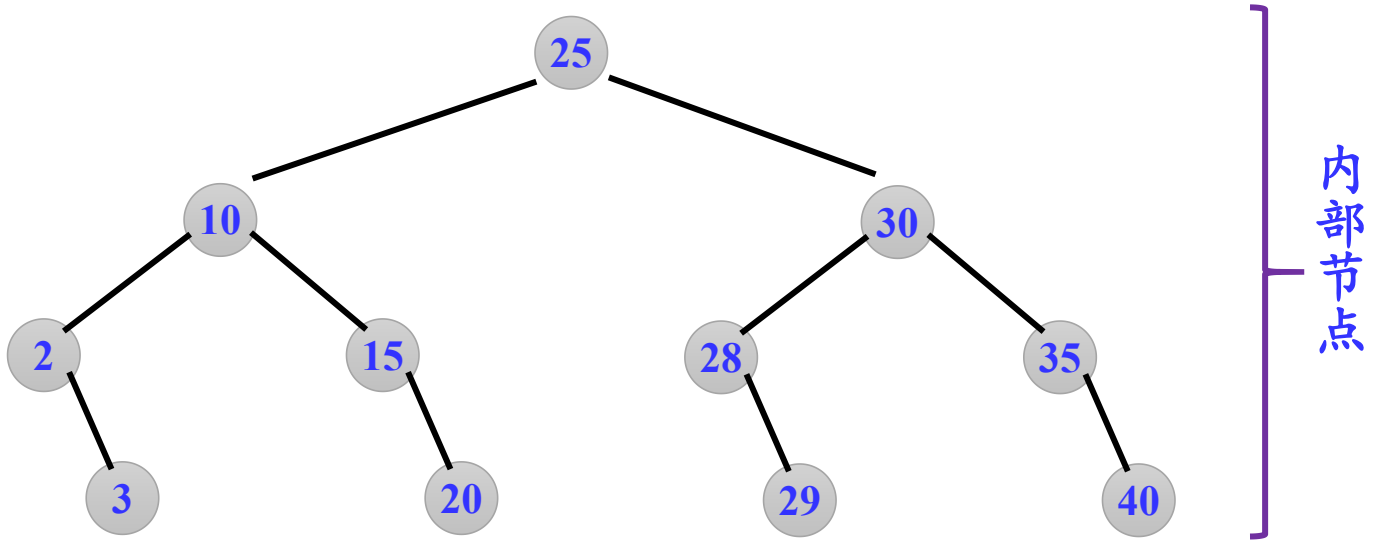
【例8-1】对于给定11个数据元素的有序表(2，3，10，15，20，25，28，29，30，35，40)，采用二分查找，试问：

（1）若查找给定值为20的元素，将依次与表中哪些元素比较？

（2）若查找给定值为26的元素，将依次与哪些元素比较？

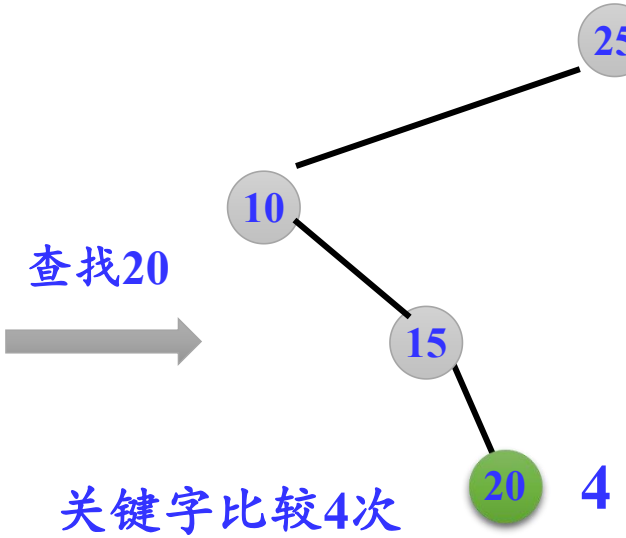
（3）假设查找表中每个元素的概率相同，求查找成功时的平均查找长度和查找不成功时的平均查找长度。

判定树：



解：（1）若查找给定值为20的元素，依次与表中25、10、15、20元素比较，共比较4次。

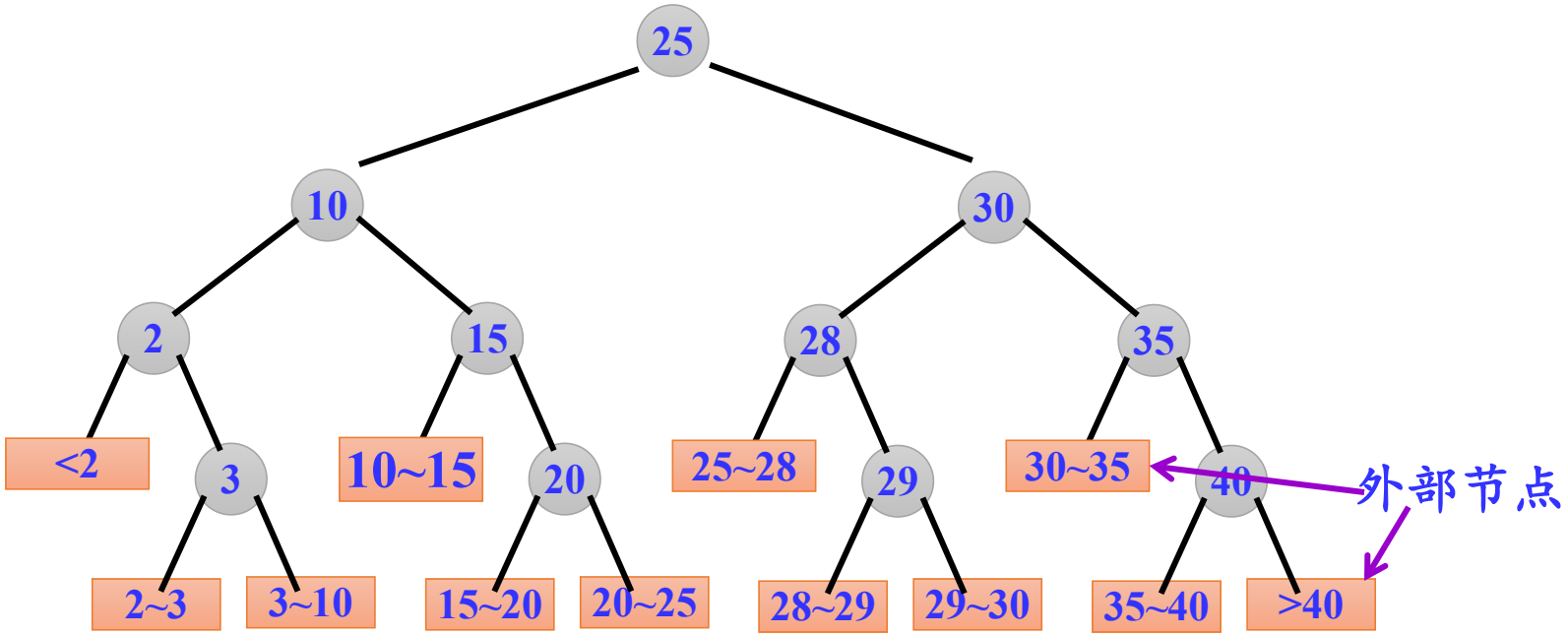
成功二分查找：恰好是走了一条从判定树的根到被查记录的路径，经历比较的关键字次数恰为该记录在树中的层数。



（3）在查找成功时，会找到图中某个内部节点，则成功时的平均查找长度：

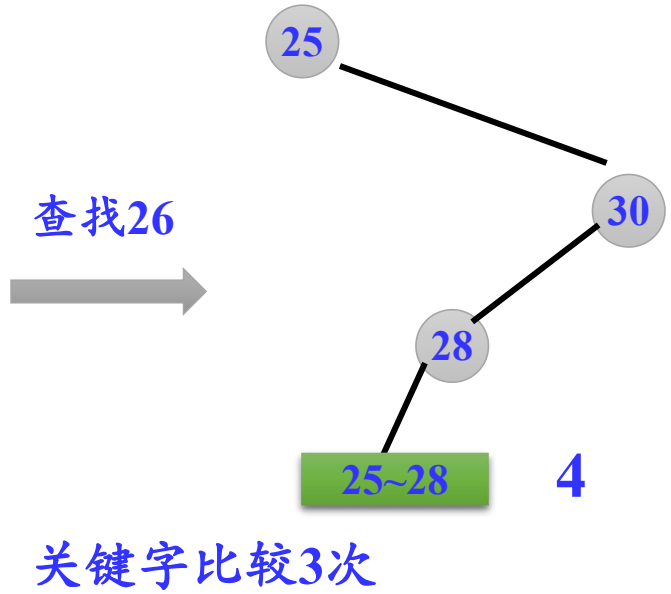
$$ASL_{成功} = \frac{1\times1+2\times2+4\times3+4\times4}{11} = 3$$

带外部节点的判定树：



（2）若查找给定值为26的元素，依次与25、30、28元素比较，共比较3次。

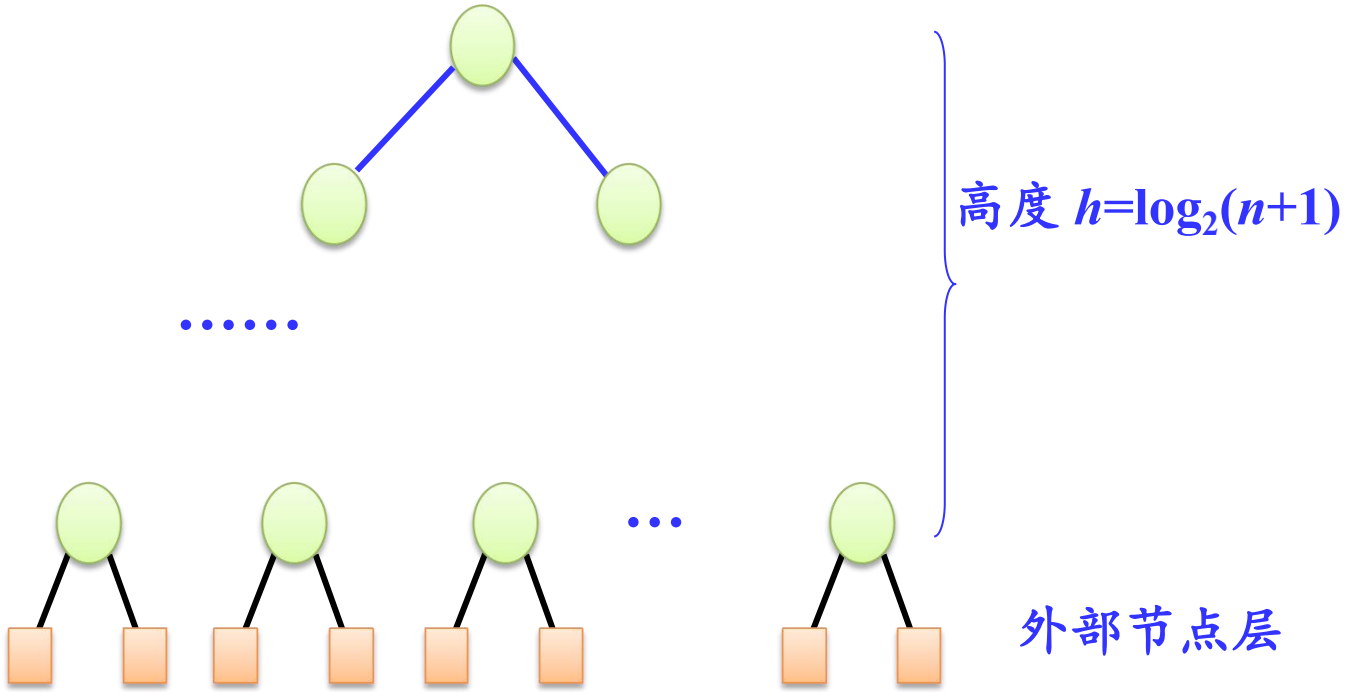
不成功二分查找：比较过程经历了一条从判定树根到某个外部节点的路径，所需的关键字比较次数是该路径上内部节点的总数。

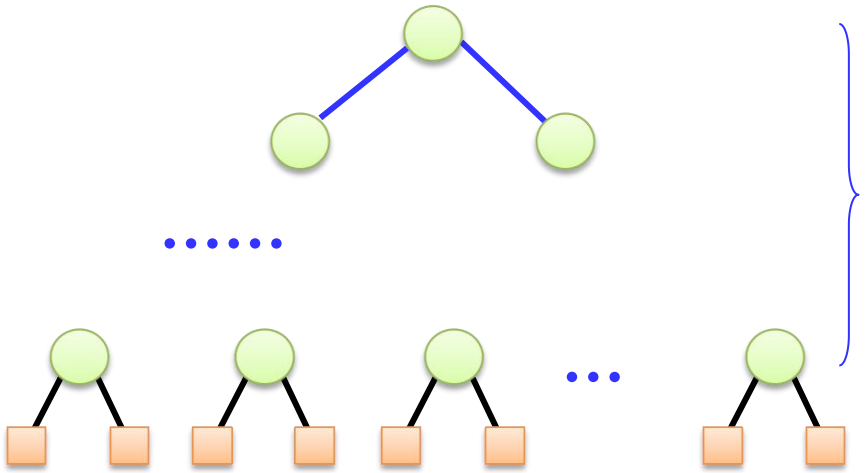


（3）在查找不成功时，会找到图中某个外部节点，则不成功时的平均查找长度：

$$ASL_{不成功}=\frac{4\times3+8\times4}{12}=3.67$$

当n比较大时，将判定树看成内部节点的总数为$n=2^h-1$、高度为$h=\log_2(n+1)$的满二叉树（高度h不计外部节点）。树中第i层上的记录个数为2^{i-1}，查找该层上的每个记录需要进行i次比较。





在等概率假设下，二分查找成功时的平均查找长度为：

$$ASL_{bn}=\sum_{i=1}^{n}p_ic_i=\frac{1}{n}\sum_{j=1}^{h}2^{j-1}\times j=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)\text{-}1$$

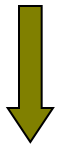
对于n个元素，二分查找，成功时最多的关键字比较次数为：$\lceil \log_2(n+1) \rceil$

不成功时关键字比较次数为：$\lceil \log_2(n+1) \rceil$。

二分查找的时间复杂度为$O(\log_2 n)$。

数据结构经典算法的启示

顺序查找算法



利用了数据的有序性

二分查找算法

8.2.3 索引存储结构和分块查找

1、索引存储结构

索引存储结构＝数据表＋索引表

索引表中的每一项称为索引项，索引项的一般形式是：（关键字，地址）

关键字唯一标识一个节点，地址作为指向该关键字对应节点的指针，也可以是相对地址。

示例

学号	姓名
1	张三
4	李四
3	王五
2	刘六

学生表

存储

数据表

存储地址	学号	姓名
0	1	张三
1	4	李四
2	3	王五
3	2	刘六

提取

学号	地址
1	0
4	1
3	2
2	3

排序

索引表

学号	地址
1	0
2	3
3	2
4	1

数据表

存储地址	学号	姓名
0	1	张三
1	4	李四
2	3	王五
3	2	刘六

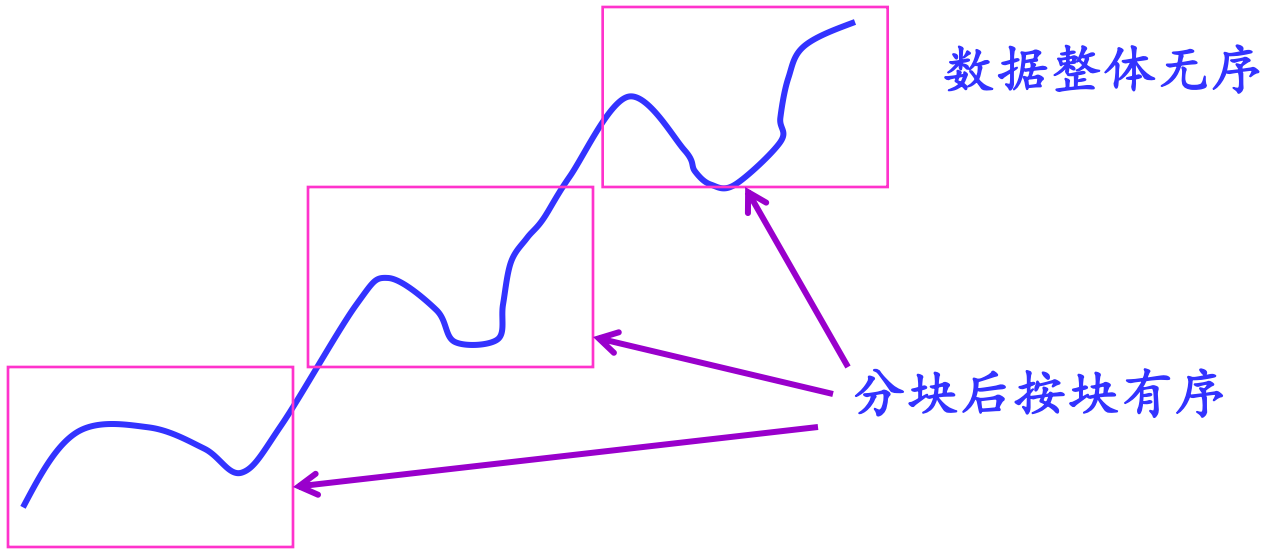
索引表

学号	地址
1	0
2	3
3	2
4	1

学生表的索引存储结构

2、分块查找

思路：

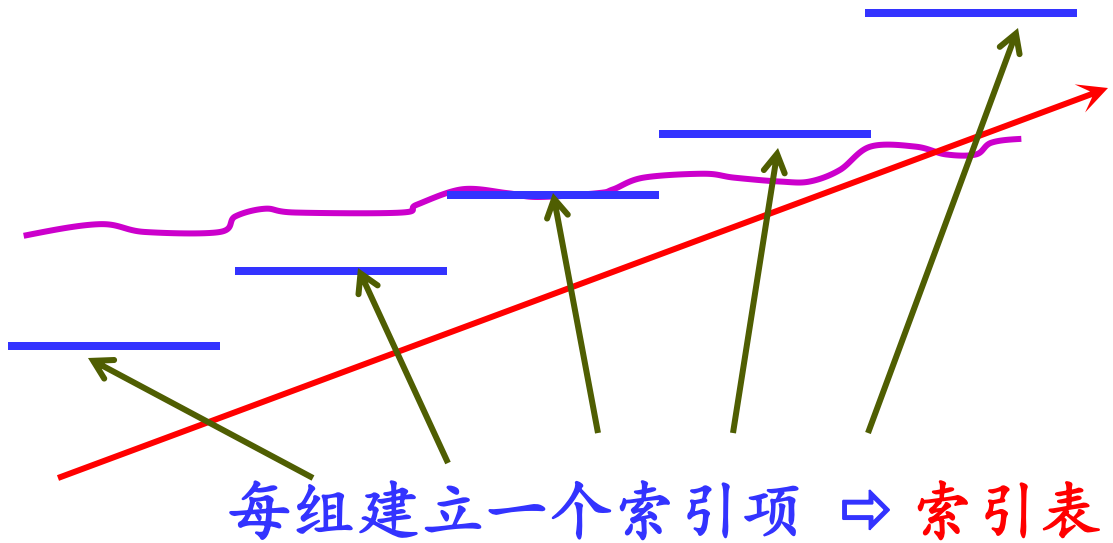


例如，设有一个线性表，其中包含25个记录，其关键字序列为：

8，14，6，9，10，22，34，18，19，31，40，38，54，66， 46，71，78，68，80，85， 100， 94，88，96，87

分块：将n=25个记录分为b=5块，每块中有s=5个记录。

数据特性：

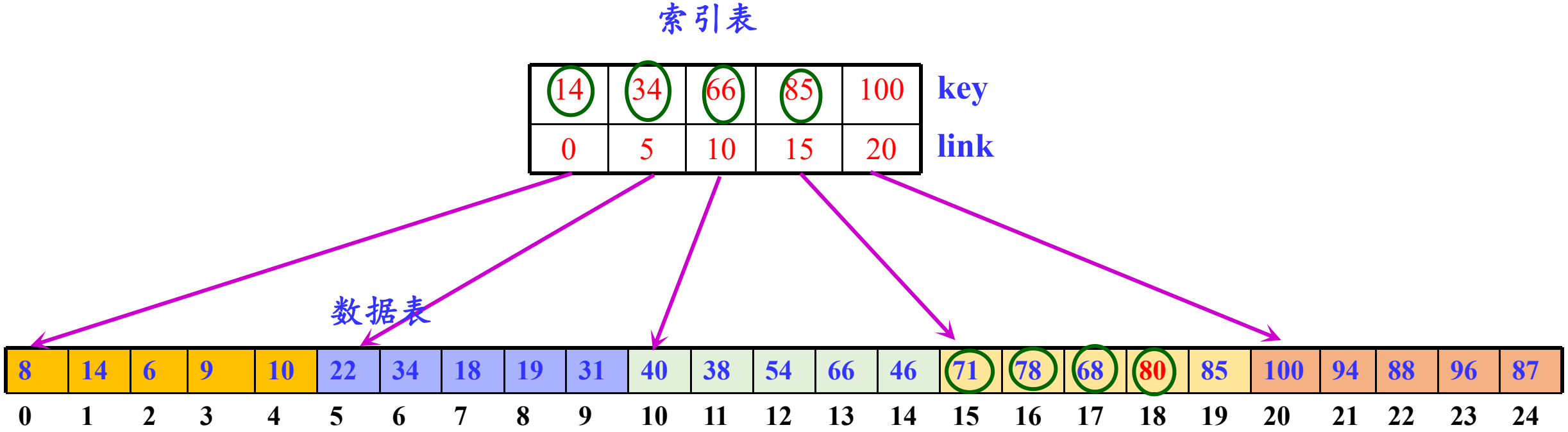


分块查找方法：

- 索引表（有序）：可以顺序查找块，也可以二分查找块。
- 数据块（无序）：只能顺序查找块中元素。

分块查找演示

查找关键字为80的记录



分块查找的索引存储结构

（1）顺序查找索引表，比较4次

（2）在对应块中查找，比较4次，共比较8次。

性能：

分块查找介于顺序查找和二分查找之间。

8.3 基于树的查找

以二叉树或树作为查找的组织形式，称为树表，它是一类动态查找表，不仅适合于数据查找，也适合于表插入和删除操作。

常见的基于树的查找算法：

- 二叉排序树
- 平衡二叉树
- B-树
- B+树

8.3.1 二叉排序树

二叉排序树（简称BST）又称二叉查找（搜索）树，其定义为：二叉排序树或者是空树，或者是满足如下性质（BST性质）的二叉树：

❶ 若它的左子树非空，则左子树上所有结点值（指关键字值）均小于根结点值；

❷ 若它的右子树非空，则右子树上所有结点值均大于根结点值；

❸ 左、右子树本身又各是一棵二叉排序树。

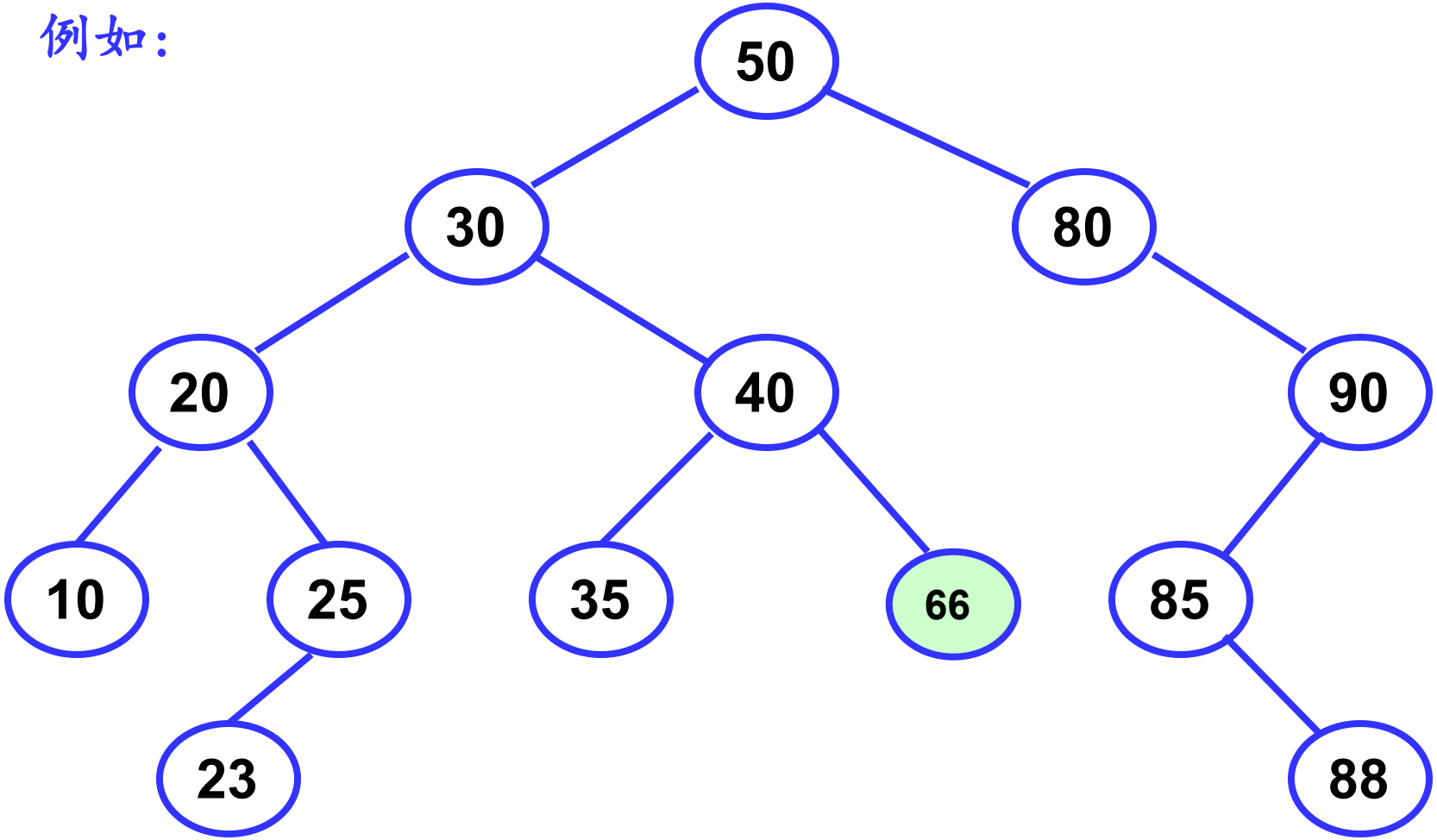
注意：二叉排序树中没有相同关键字的结点。

二叉树结构

满足BST性质：结点值约束

二叉排序树

例如：



不是二叉排序树。

试一试

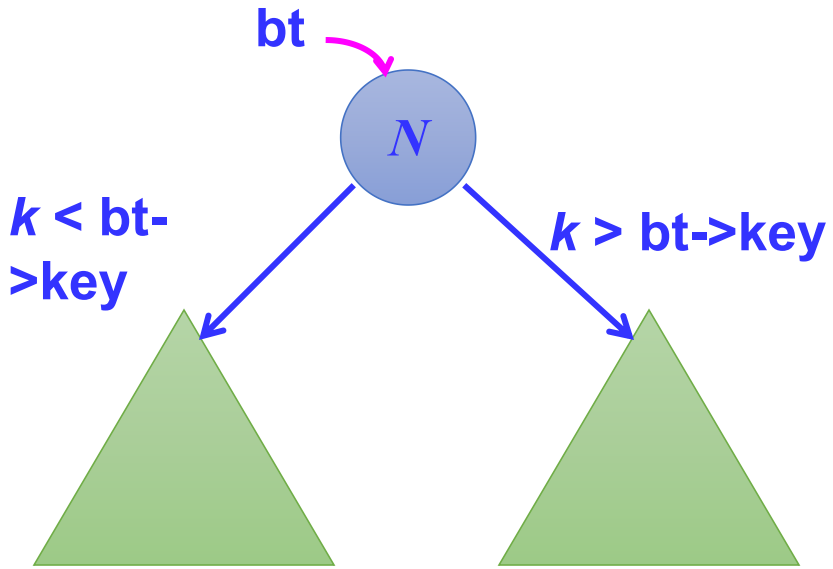
二叉排序树的中序遍历序列有什么特点？

二叉排序树的结点类型如下：

```
typedef struct node
{      KeyType key;                          //关键字项
       InfoType data;                        //其他数据域
       struct node *lchild，  *rchild;  //左右孩子指针
} BSTNode;
```

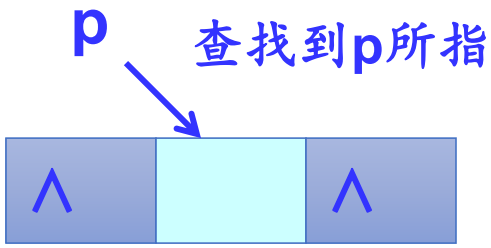
1、二叉排序树上的查找

二叉排序树可看做是一个有序表，所以在二叉排序树上进行查找，和二分查找类似，也是一个逐步缩小查找范围的过程。

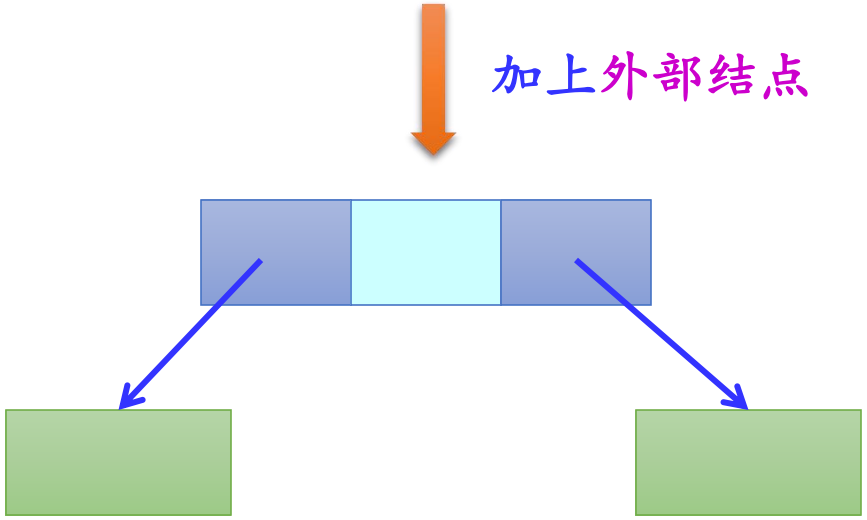


每一层只和一个结点进行关键字比较！

查找失败的情况



- 若*k*<p->data，并且p->lchild=NULL，查找失败。
- 若*k*>p->data，并且p->rchild=NULL，查找失败。



一个外部结点对应某内部结点的一个NULL指针

递归查找算法SearchBST()如下（在二叉排序树bt上查找关键字为k的记录，成功时返回该结点指针，否则返回NULL）：

```
BSTNode *SearchBST(BSTNode *bt，KeyType k)
{
    if (bt==NULL || bt->key==k)          //递归出口
        return bt;
    if (k<bt->key)
        return SearchBST(bt->lchild，k);  //在左子树中递归查找
    else
        return SearchBST(bt->rchild，k);  //在右子树中递归查找
}
```

思考题

二叉排序树查找可以设计成非递归算法，如何实现？

2、二叉排序树的插入和生成

在二叉排序树中插入一个关键字为*k*的新结点，要保证插入后仍满足**BST性质**。

插入过程：

（1）若二叉排序树T为空，则创建一个key域为*k*的结点，将它作为根结点；

（2）否则将*k*和根结点的关键字比较，若两者相等，则说明树中已有此关键字*k*，无须插入，直接返回0；

（3）若*k*<T->key，则将*k*插入根结点的左子树中。

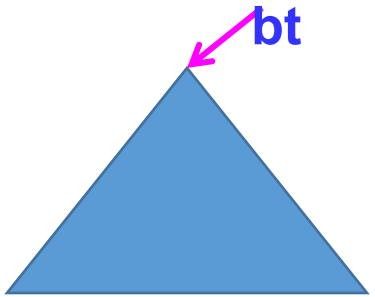
（4）否则将它插入右子树中。

对应的递归算法InsertBST()如下：

```
int InsertBST(BSTNode *&p，KeyType k)
{
    if (p==NULL)              //原树为空， 新插入的记录为根结点
    {    p=(BSTNode *)malloc(sizeof(BSTNode));
         p->key=k;p->lchild=p->rchild=NULL;
         return 1;
    }
    else if  (k==p->key)    //存在相同关键字的结点，返回0
        return 0;
    else if (k<p->key)
       return InsertBST(p->lchild，k);      //插入到左子树中
    else
       return InsertBST(p->rchild，k);      //插入到右子树中
}
```

先序遍历的思想

关键字数组$A[0..n-1]$



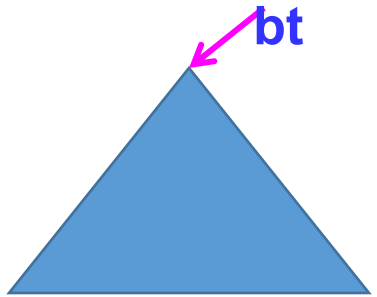

```
BSTNode *CreatBST(KeyType A[]，int n) //返回树根指针
{
    BSTNode *bt=NULL;   //初始时bt为空树
     int i=0;
     while (i<n)
     {   InsertBST(bt，A[i]); //将A[i]插入二叉排序树T中
         i++;
     }
     return bt;                  //返回建立的二叉排序树的根指针
}
```

注意：任何结点插入到二叉排序树时，都是以叶结点插入的。

对应的非递归算法NonrecursiveInsertBST()如下：

```
int NonrecursiveInsertBST(BSTNode *&p，KeyType k)
{  BSTNode *q=p;
   BSTNode *pre=q;
   while(q <> NULL)
    {       pre =q;
            if  (k==q->key)  return 0;
            else if  (k < q->key) { q=q->lchild;}
            else if  (k > q->key) { q=q->rchild;}
     }
     q=(BSTNode *)malloc(sizeof(BSTNode));
     q->key=k;q->lchild=q->rchild=NULL;
     if (p==NULL) p=q;
     else if (k < pre->key) pre->lchild=q;
     eles if (k > pre->key) pre->rchild=q;
    return 1;
}
```

关键字数组$A[0..n-1]$




```
BSTNode *NonrecursiveCreatBST(KeyType A[]，int n) //返回树根指针
{
    BSTNode *bt=NULL;   //初始时bt为空树
     int i=0;
     while (i<n)
     {   InsertBST(bt，A[i]); //将A[i]插入二叉排序树T中
         i++;
     }
     return bt;                  //返回建立的二叉排序树的根指针
}
```

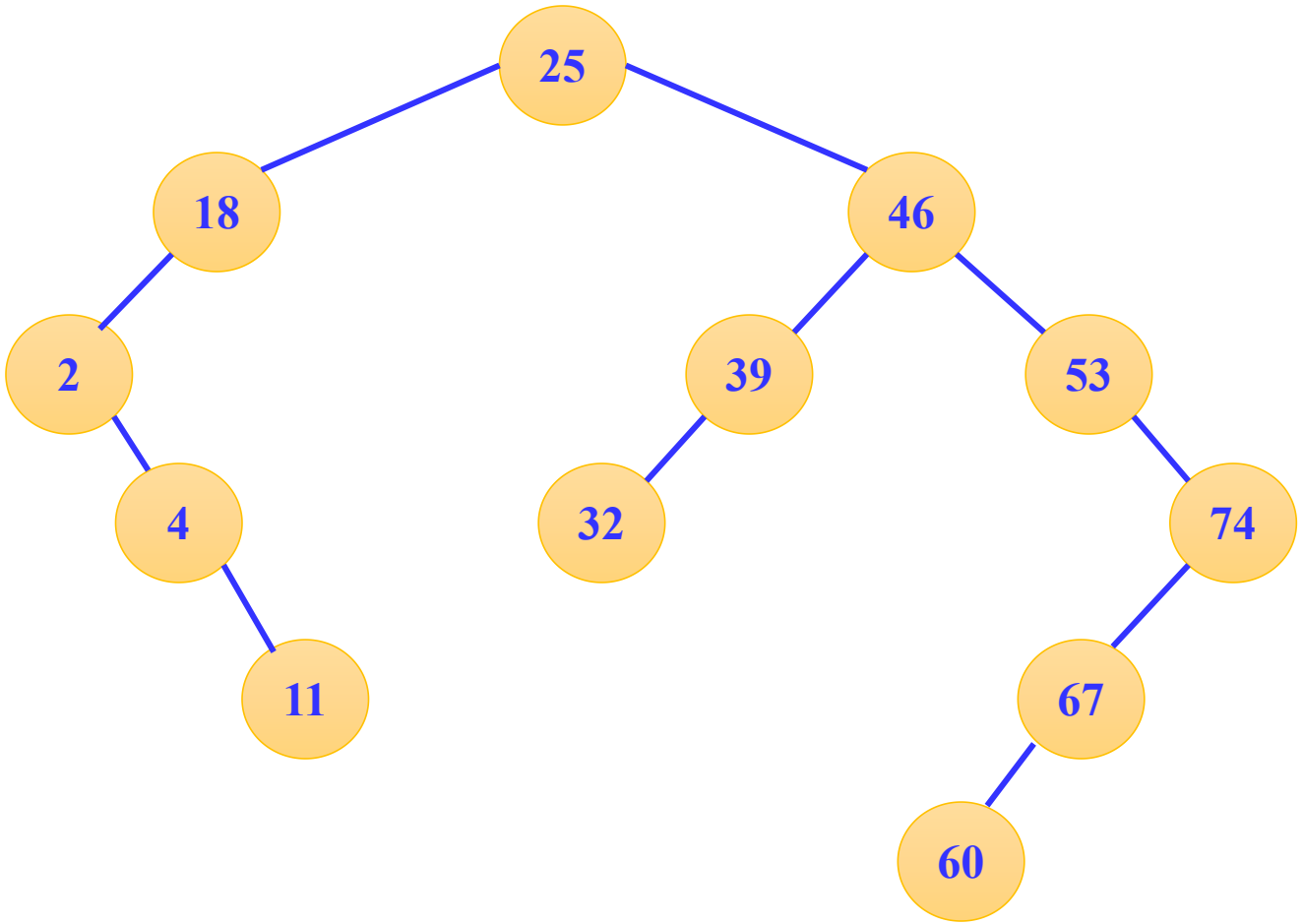
注意：任何结点插入到二叉排序树时，都是以叶结点插入的。

【例8-3】已知一组关键字为{25，18，46，2，53，39，32，4，74，67，60，11}。

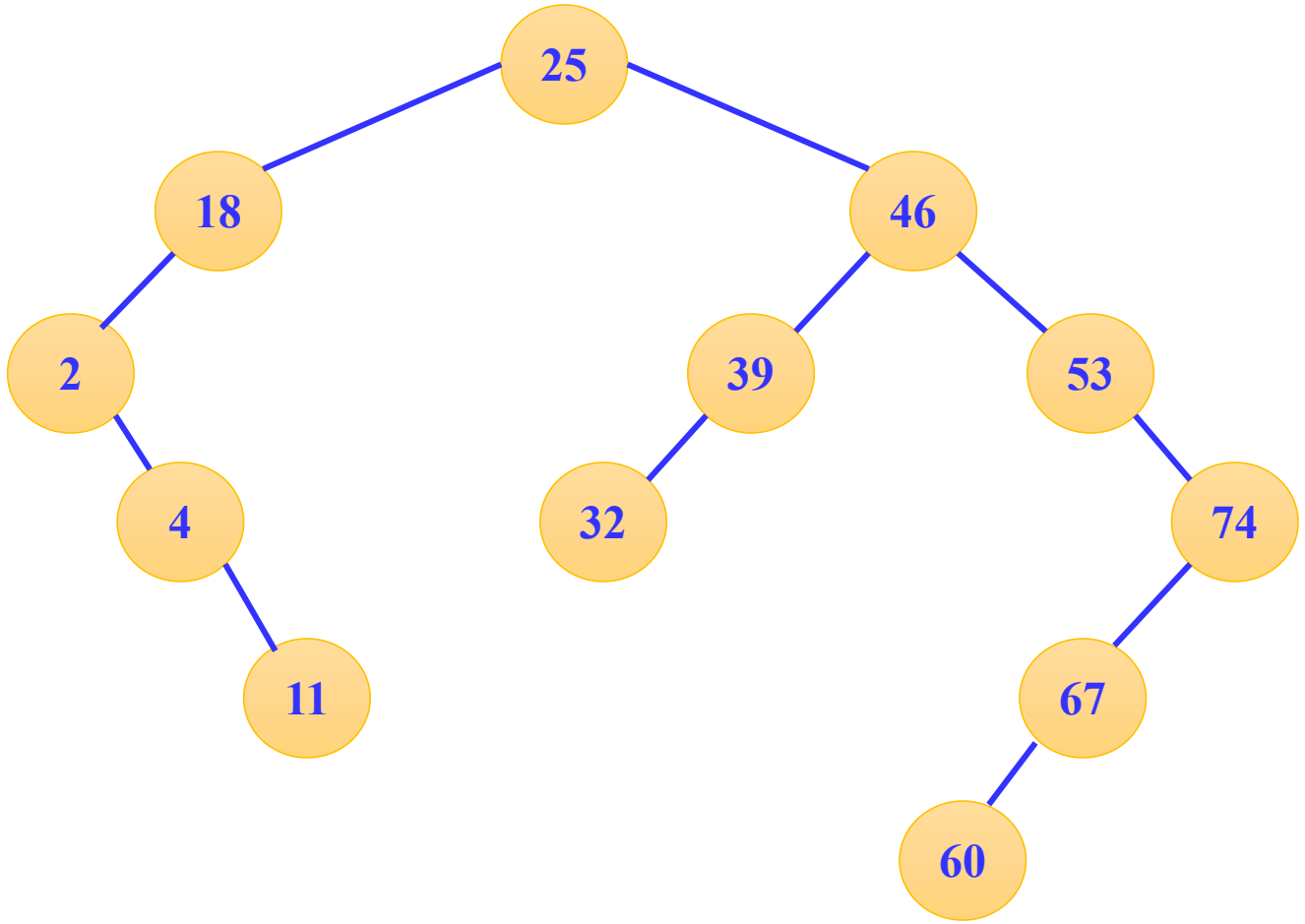
按表中的元素顺序依次插入到一棵初始为空的二叉排序树中，画出该二叉排序树。

求在等概率的情况下查找成功的平均查找长度和查找不成功的平均查找长度。

序列： 25 18 46 2 53 39 32 4 74 67 60 11

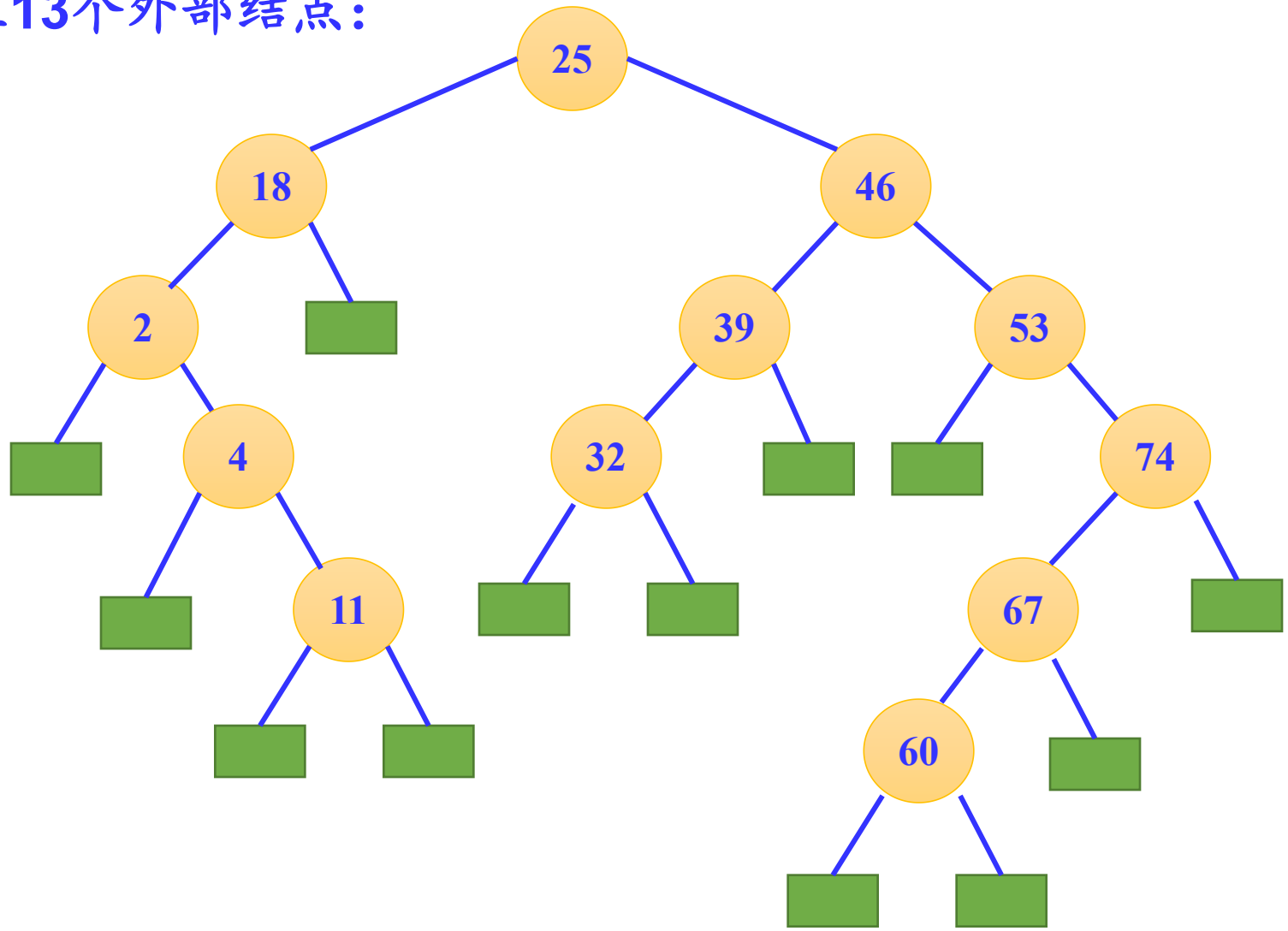


二叉排序树创建完毕



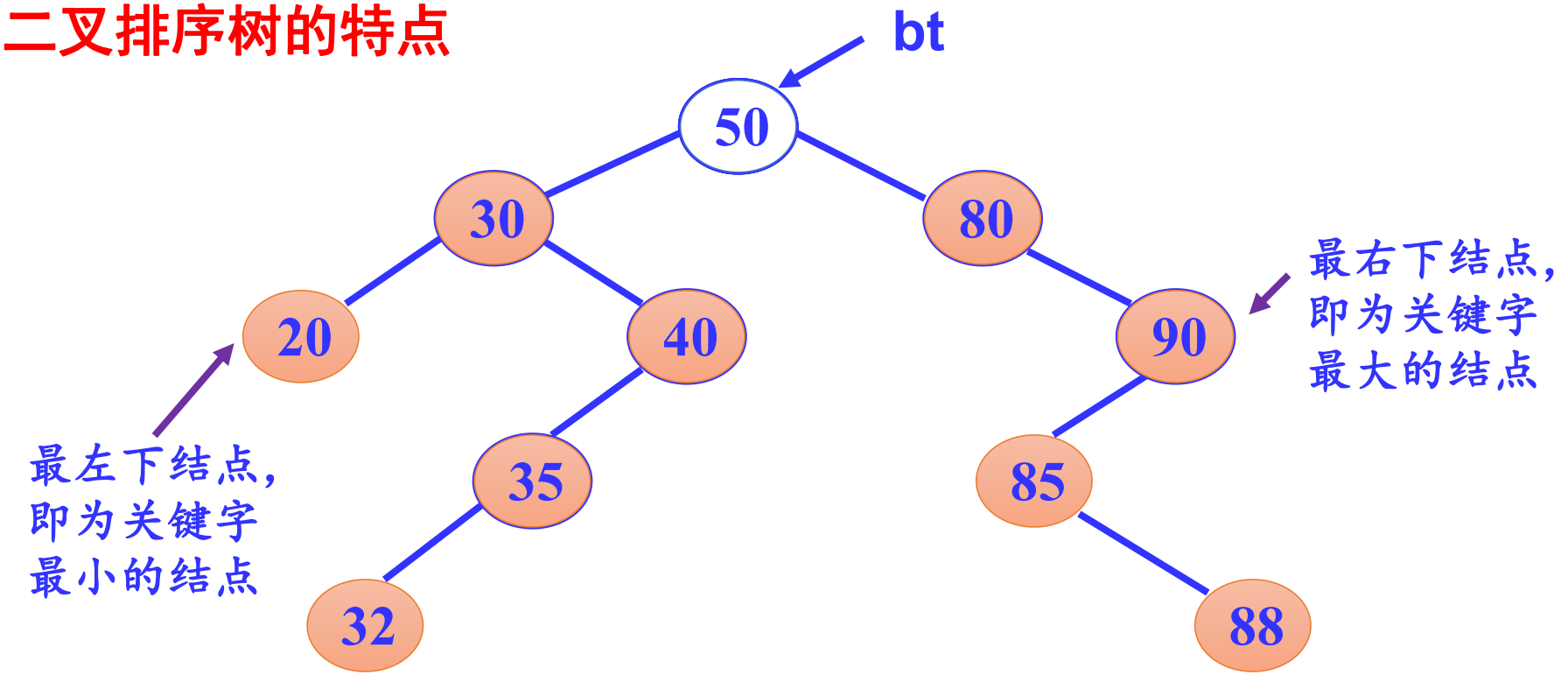
$$ASL_{成功} = \frac{1\times1+2\times2+3\times3+3\times4+2\times5+1\times6}{12} = 3.5$$

加上13个外部结点：



$$ASL_{不成功} = \frac{1\times2+3\times3+4\times4+3\times5+2\times6}{13} = 4.15$$

二叉排序树的特点



中序序列： 20，30，32，35，40，50，80，85，88，90

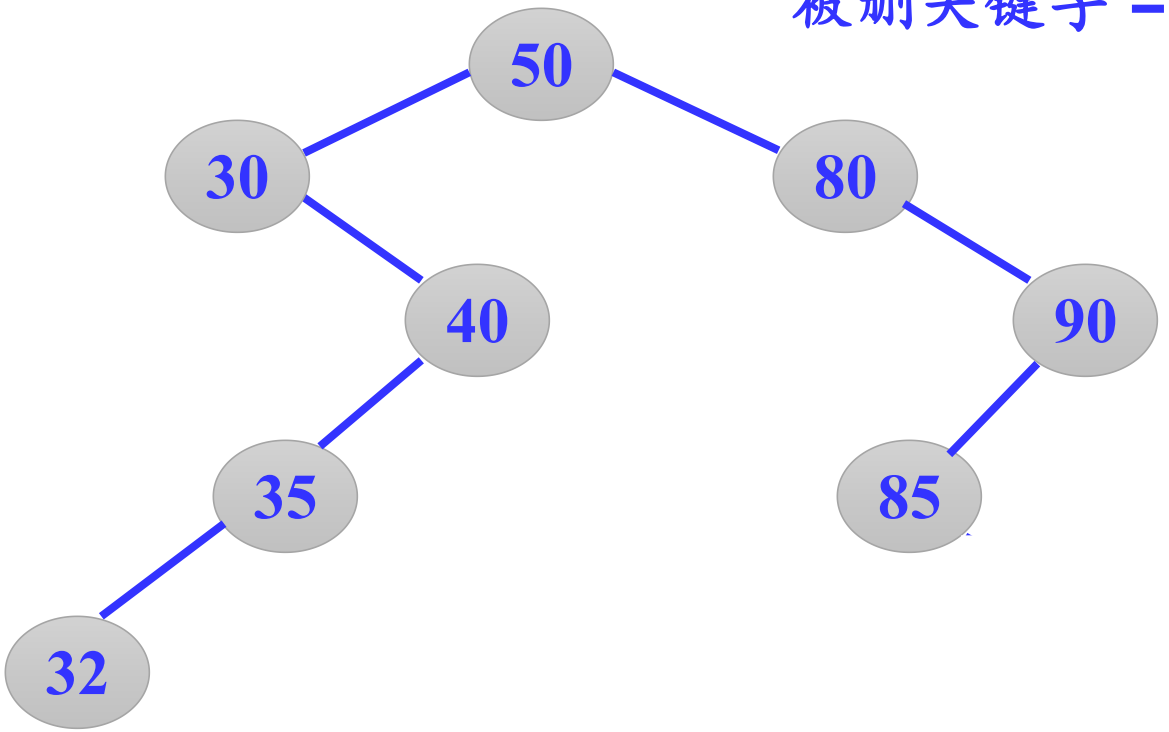
- 二叉排序树的中序序列是一个递增有序序列
- 根结点的最左下结点是关键字最小的结点
- 根结点的最右下结点是关键字最大的结点

3、 二叉排序树的结点删除

（1）被删除的结点是叶子结点：直接删去该结点。

例如：

被删关键字 = 88

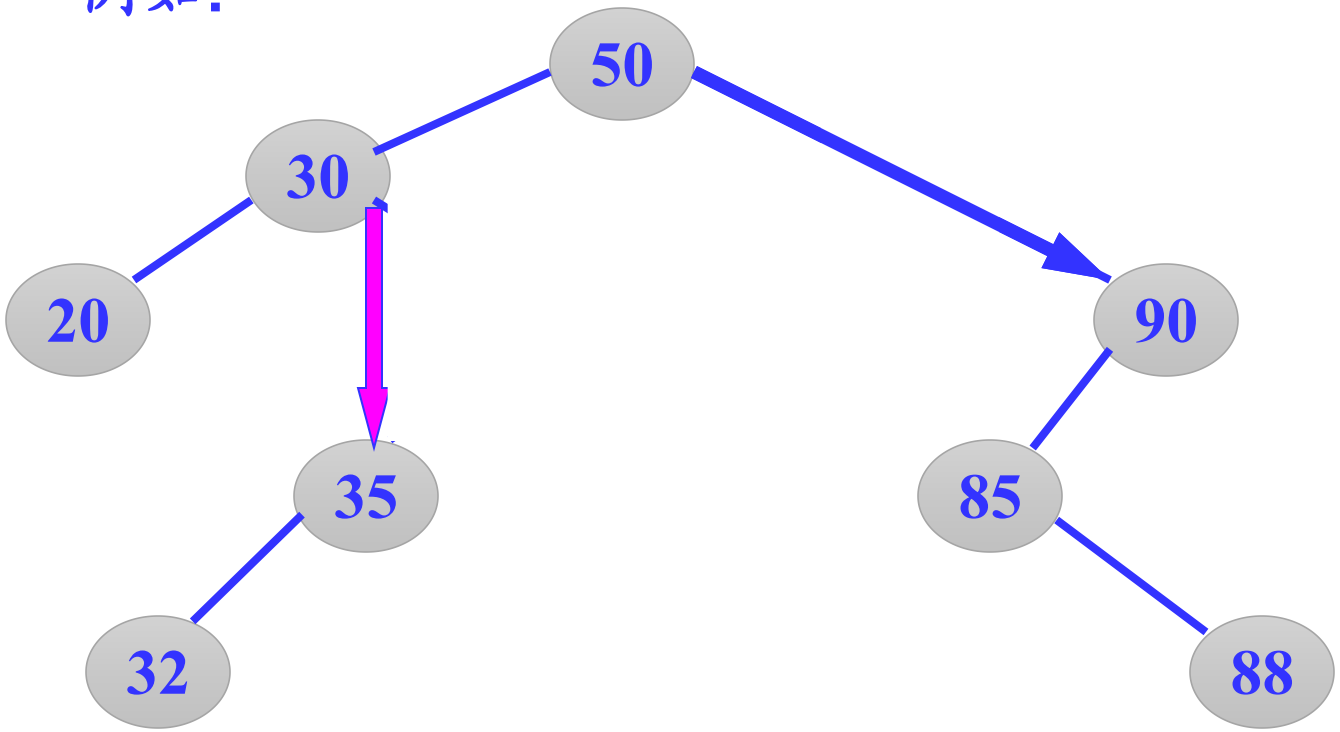


其双亲结点中相应指针域的值改为“空”

（2） 被删除的结点只有左子树或者只有右子树，用其左子树或者右子树替换它（结点替换）。

被删关键字 = 80

例如:

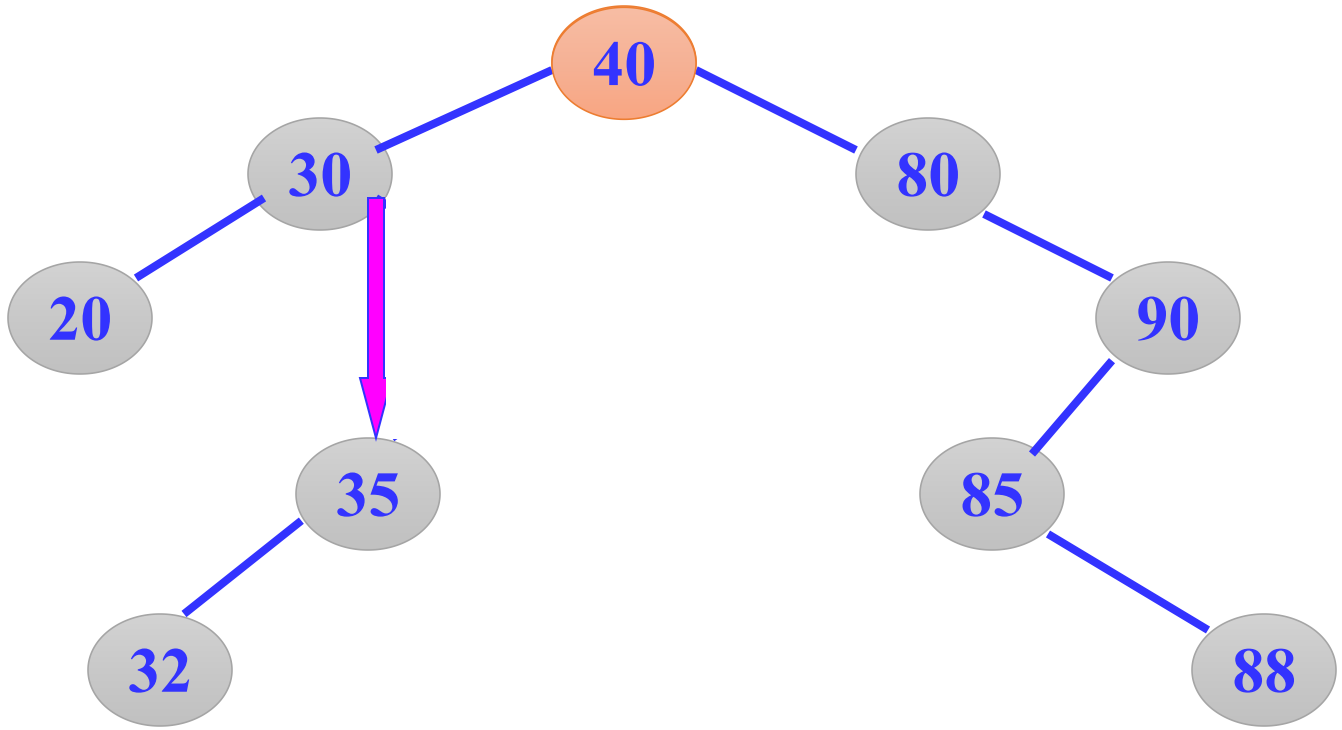


其双亲结点的相应指针域的值改为 “指向被删除结点的左子树或右子树”。

（3）被删除的结点既有左子树，也有右子树

例如：

被删关键字 = 50



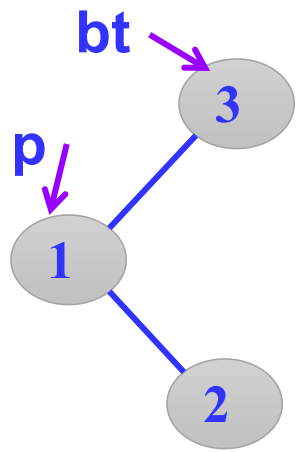
以其中序前趋值替换之（值替换），然后再删除该前趋结点。前趋是左子树中最大的结点。

也可以用其后继替换之，然后再删除该后继结点。后继是右子树中最小的结点。

算法实现：如何删除仅仅有右子树的结点*p：

❶ 查找被删结点

```
int deletek(BSTNode *&bt，KeyType k)
{    if (bt!=NULL)
     {    if (k==bt->key)
          {    deletep(bt)
               return 1;
          }
          else if (k<bt->key)
               deletek(bt->lchild，k);
          else
               deletek(bt->rchild，k);
     }
     else
        return 0;
}
```



deletek(bt，1)

1<3

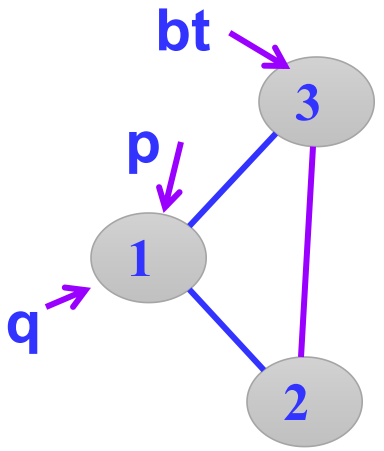
deletek(bt->lchild，1)

1=1

deletep(p)

❷ 删除结点*p

```
void deletep(BSTNode *&p)
{    BSTNode *q;
     q=p;
     p=p->rchild;
          //用其右孩子结点替换它
     free(q);
}
```



达到用*p的右孩子结点替换它的目的

deletek(bt,1)

deletek(bt->lchild,1)

bt->lchild=p

deletep(p)

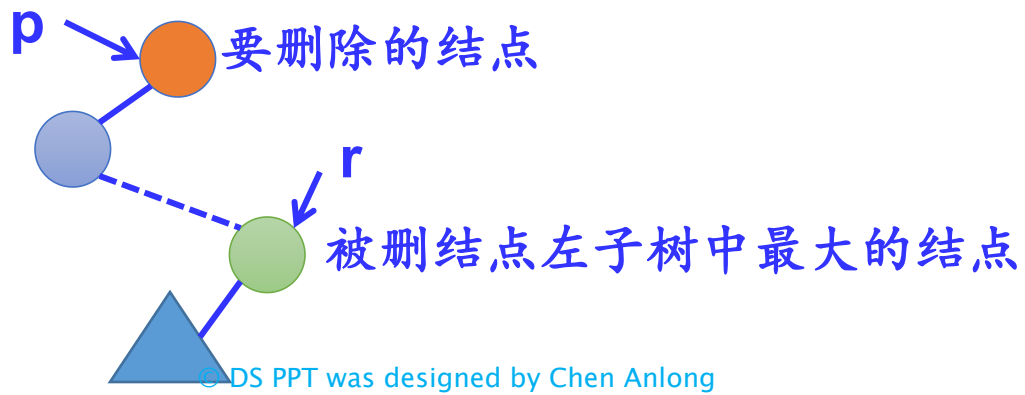
在二叉排序树bt中删除结点的算法

```
int DeleteBST(BSTNode *&bt，KeyType k)  //在bt删除关键字为k的结点
{
      if (bt==NULL) return 0;       //空树删除失败
      else
      {     if (k<bt->key) return DeleteBST(bt->lchild，k);
                  //递归在左子树中删除为k的结点
            else if (k>bt->key) return DeleteBST(bt->rchild，k);
                  //递归在右子树中删除为k的结点
           else   //bt->key=k
           {      Delete(bt);   //调用Delete(bt)函数删除*bt结点
                  return 1;
           }
     }
}
```

```
void Delete(BSTNode *&p)          //从二叉排序树中删除*p结点
{    BSTNode *q;
     if (p->rchild==NULL)         //*p结点没有右子树的情况
     {    q=p; p=p->lchild;       //用其左孩子结点替换它
          free(q);
     }
     else if (p->lchild==NULL)    //*p结点没有左子树的情况
     {    q=p; p=p->rchild;       //用其右孩子结点替换它
          free(q);
     }
     else Delete1(p，p->lchild);
          //*p结点既没有左子树又没有右子树的情况
}
```

p指向待删除的结点　　r指向其左孩子结点

```
void Delete1(BSTNode *p,   BSTNode *&r)
 //当被删*p结点有左右子树时的删除过程
 {    BSTNode *q;
      if (r->rchild!=NULL)
            Delete1(p，r->rchild); //递归找*r的最右下结点
      else                         //r指向最右下结点
      {     p->key=r->key; p->data=r->data  //值替换
            q=r; r=r->lchild;       //删除原*r结点
            free(q);                //释放原*r的空间
      }
 }
```



8.3.2 平衡二叉树（AVL）

1、什么是平衡二叉树

若一棵二叉树中每个结点的左、右子树的高度至多相差1，则称此二叉树为平衡二叉树（AVL）。

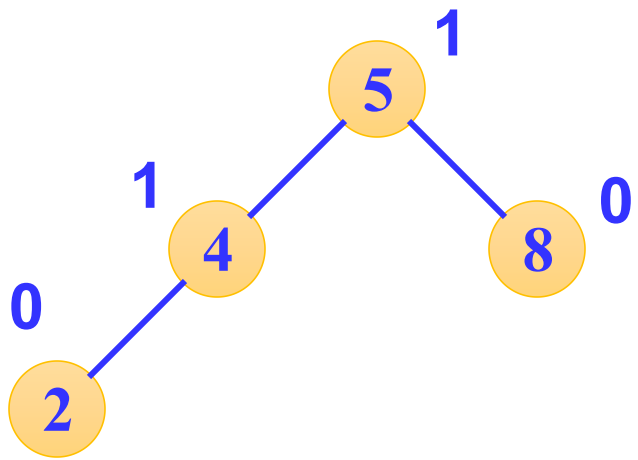
平衡因子：该结点左子树的高度减去右子树的高度。

若一棵二叉树中所有结点的平衡因子的绝对值小于或等于1，该二叉树称为平衡二叉树。

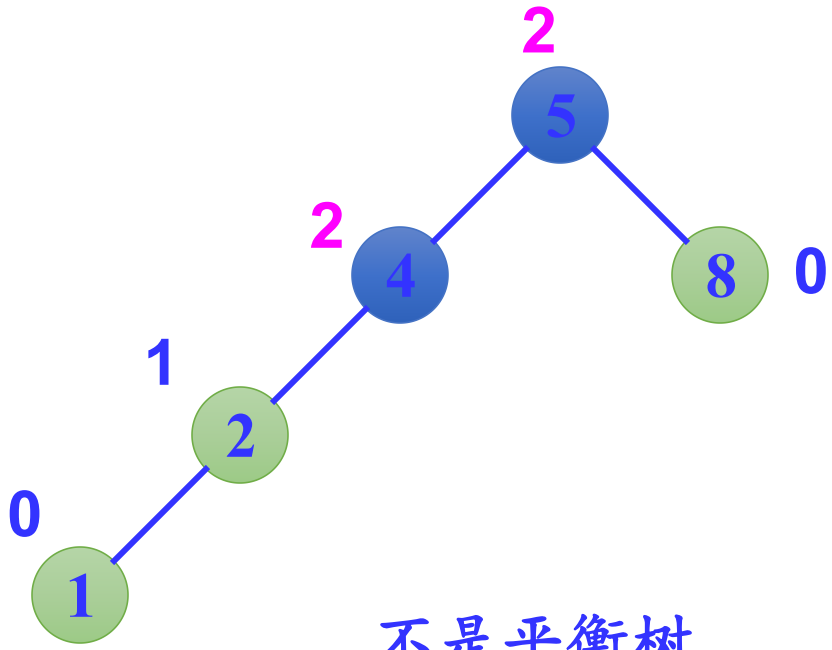
二叉排序树

所有结点的平衡因子的绝对值≤1：结构约束

平衡二叉树



是平衡树



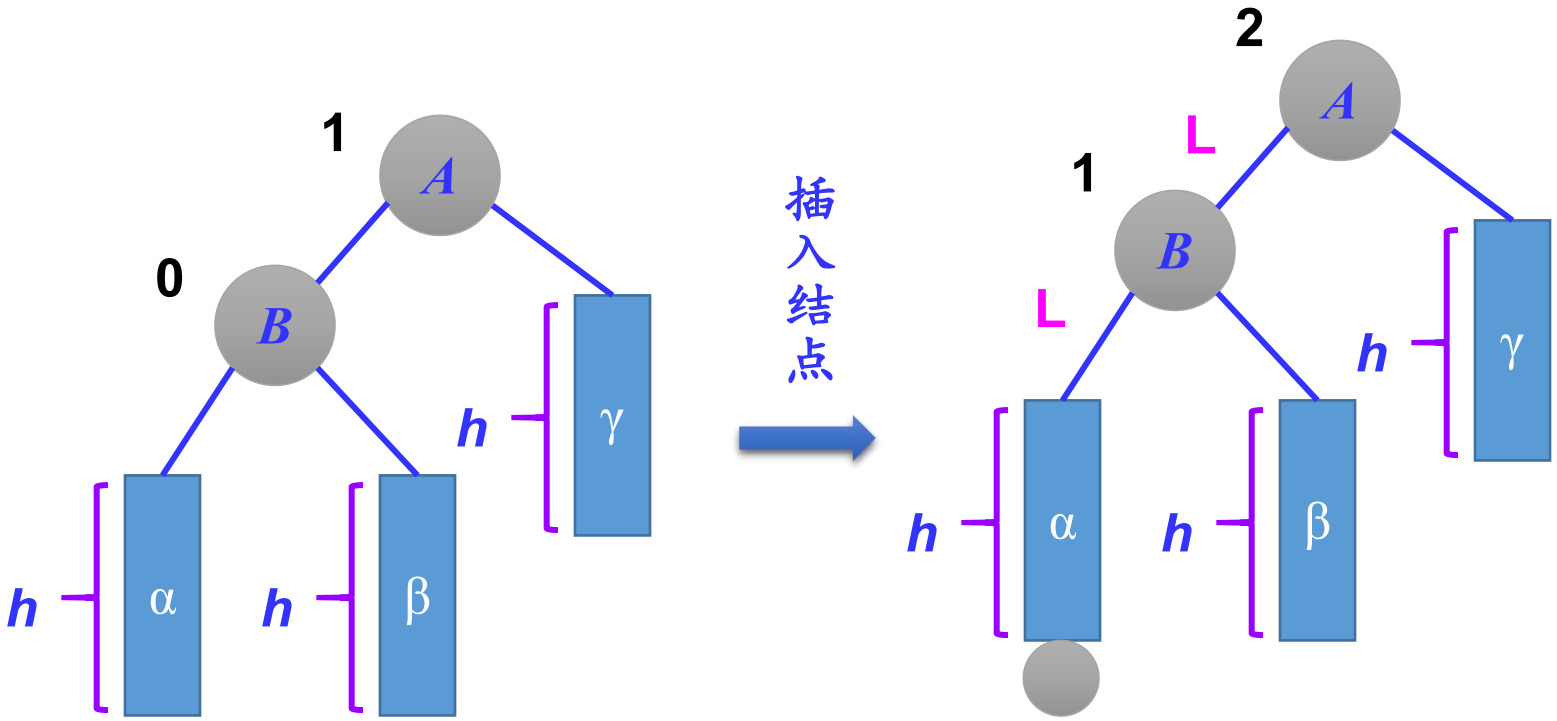
不是平衡树

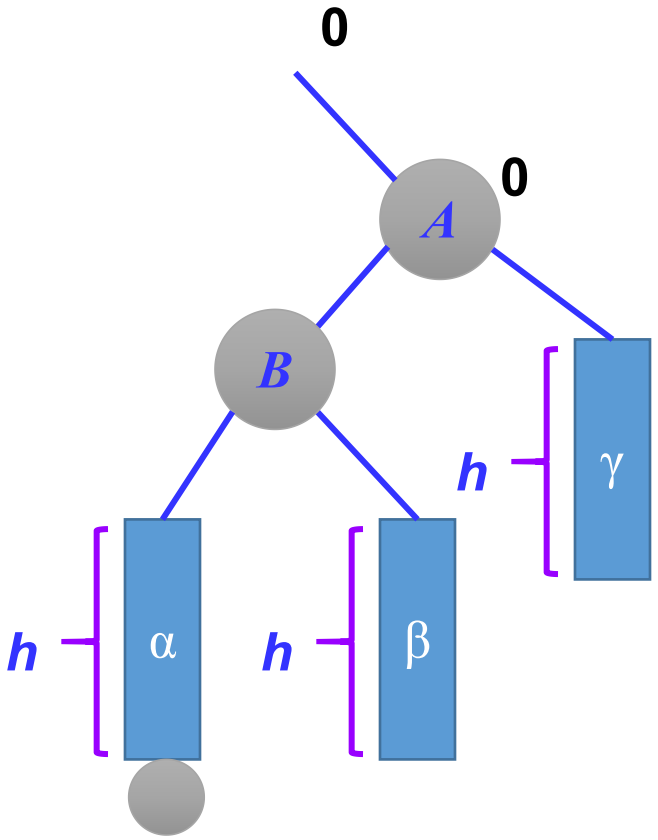
2、平衡二叉树的插入调整

平衡二叉树中插入新结点方式与二叉排序树相似，只是插入后可能破坏了平衡二叉树的平衡性，解决方法是调整。

调整操作可归纳为4种情况。

（1）LL型调整



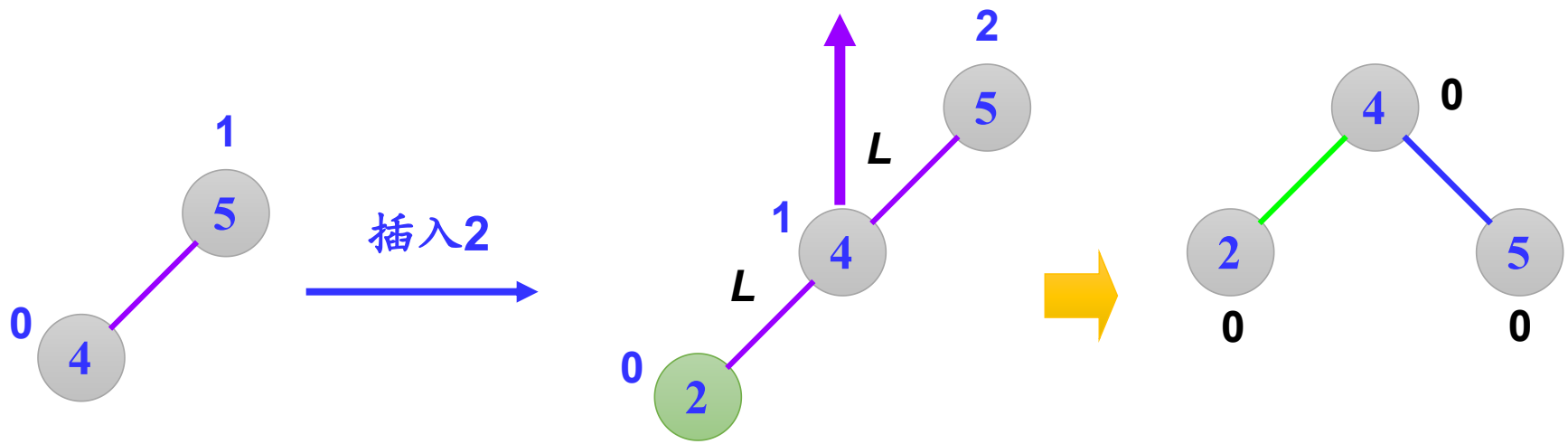


LL调整后的结果

LL型调整过程：

- B结点带左子树α一起上升
- A结点成为B的右孩子
- 原来B结点的右子树β作为A的左子树

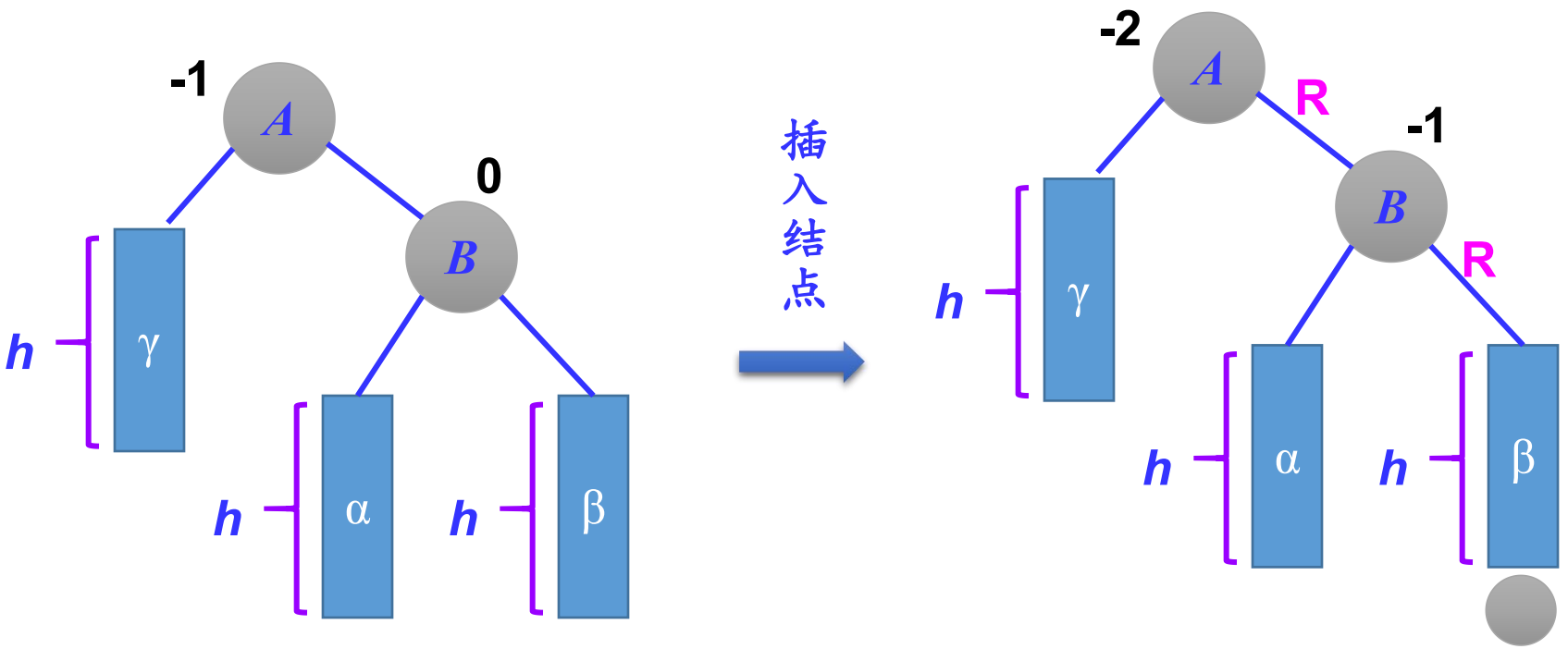
AVL树LL调整演示

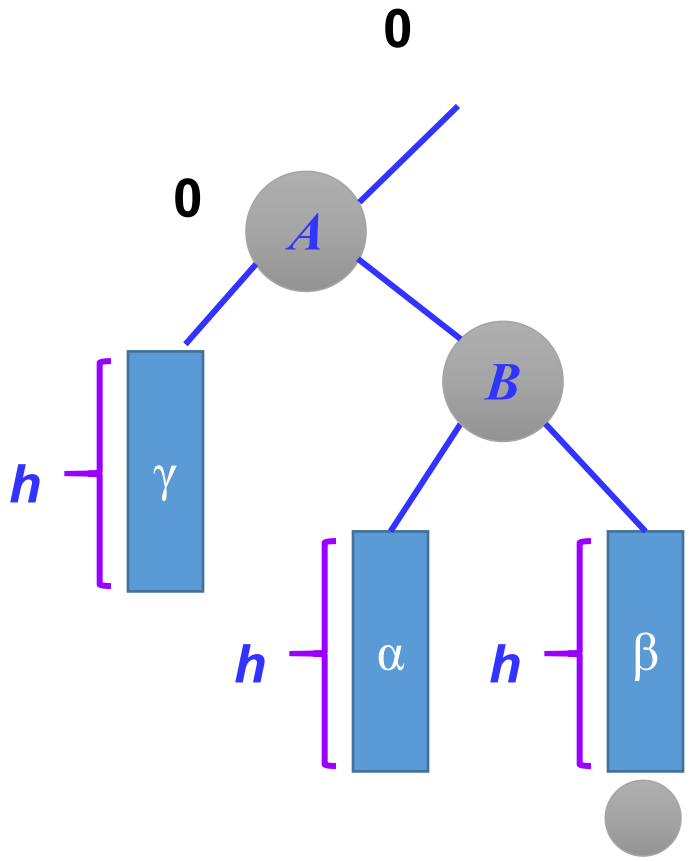


插入关键字2的结果

调整完毕

（2）RR型调整



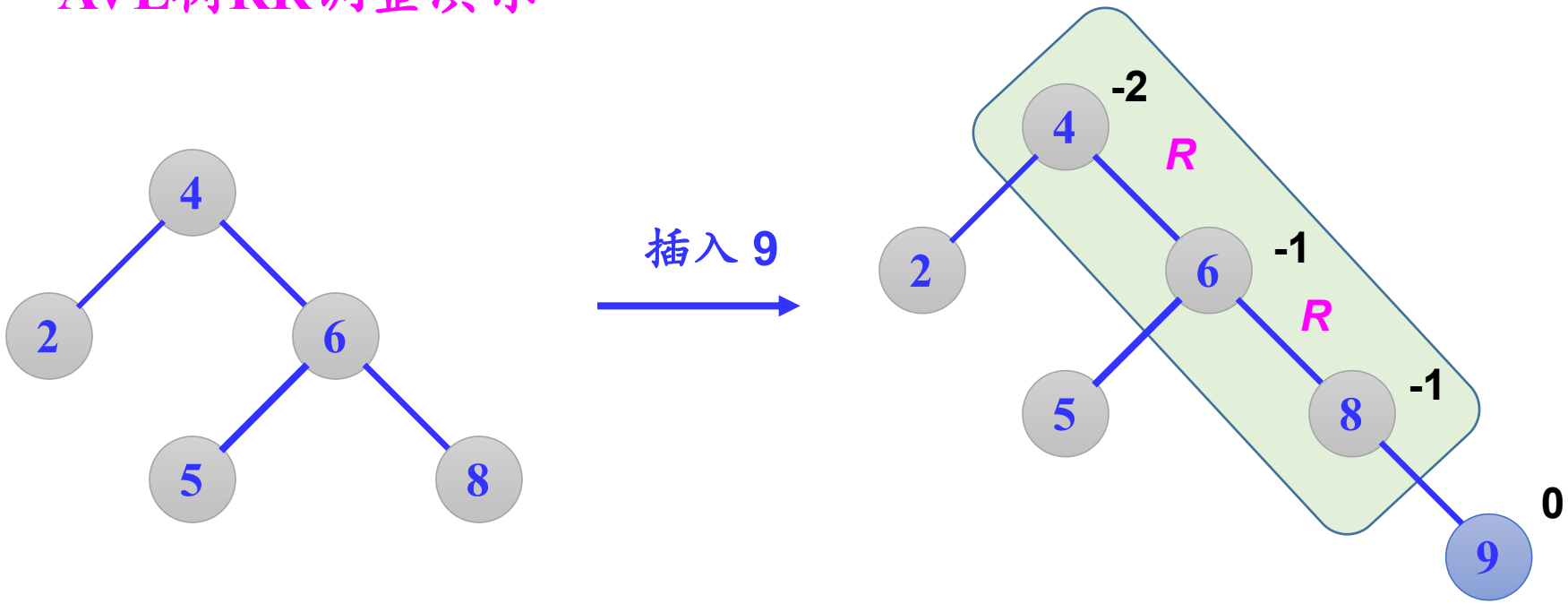


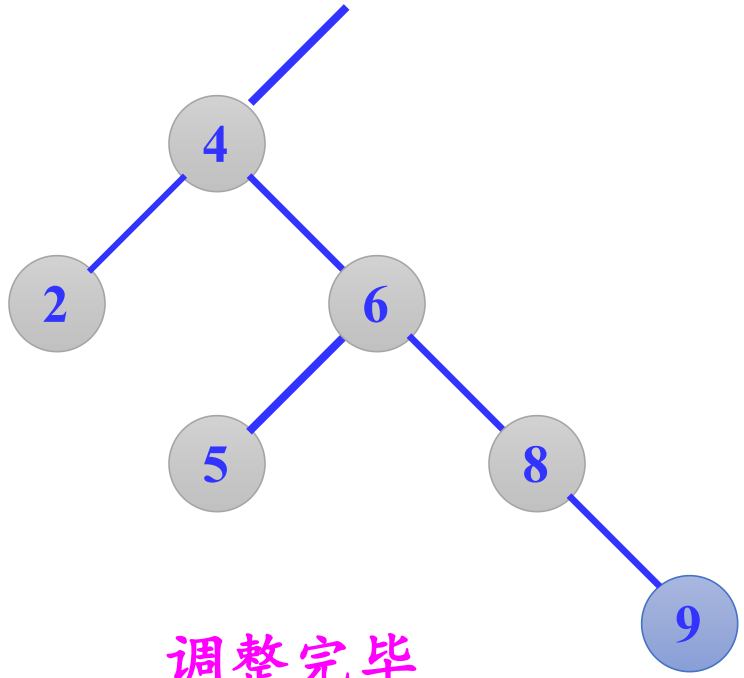
RR调整后的结果

RR型调整过程：

- B结点带右子树β一起上升
- A结点成为B的左孩子
- 原来B结点的左子树α作为A的右子树

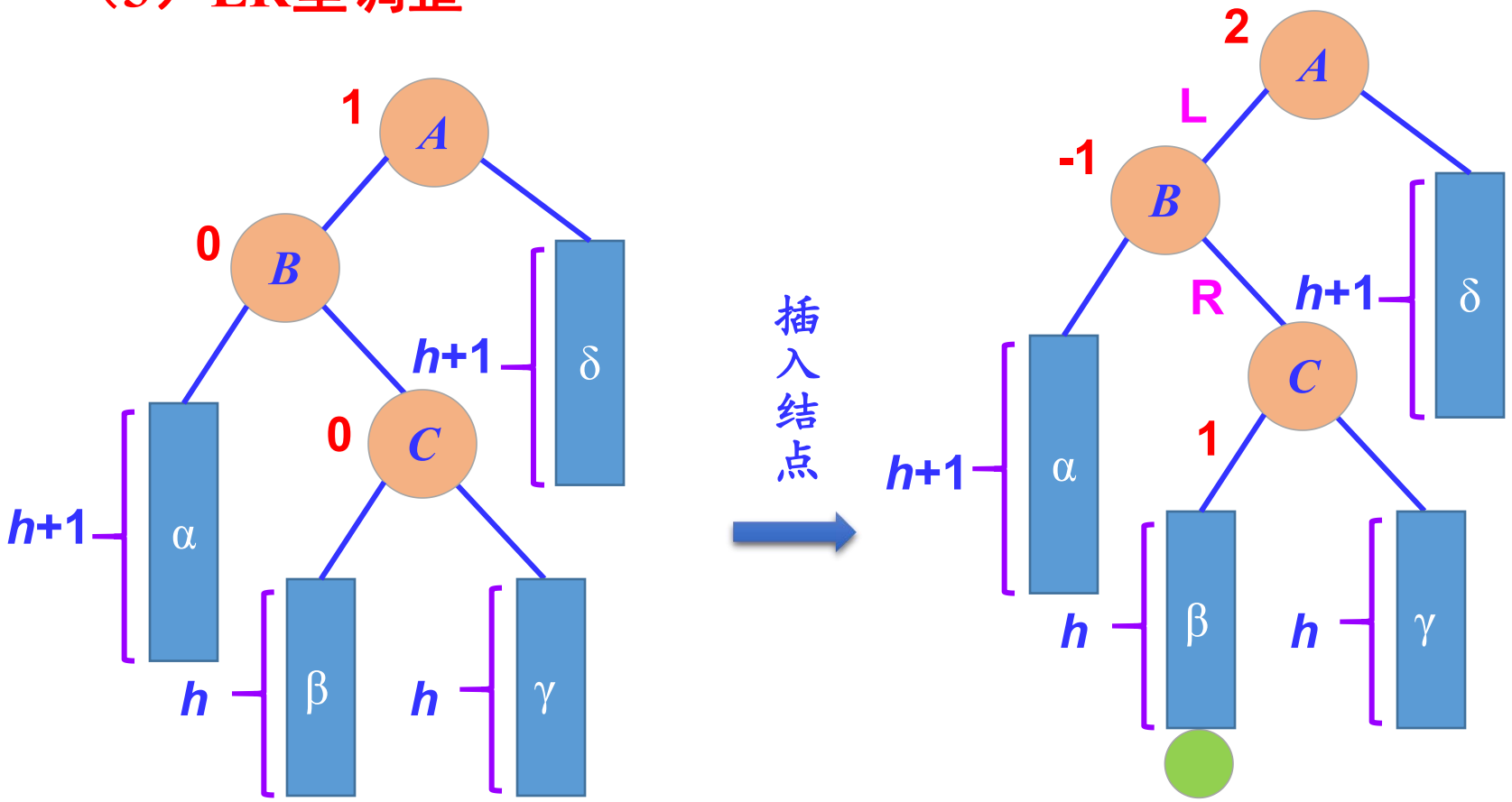
AVL树RR调整演示

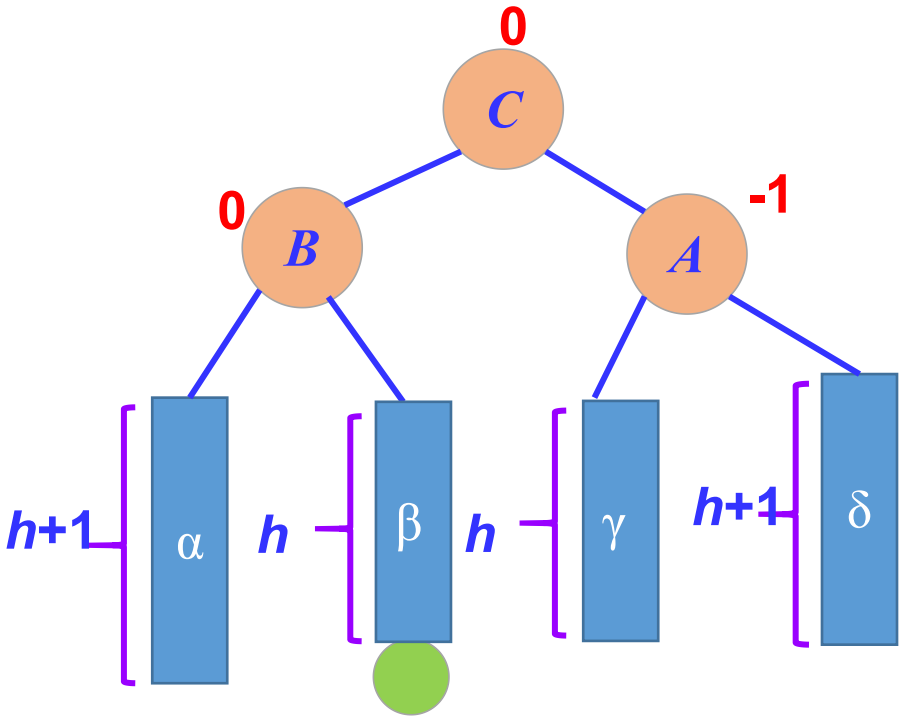




调整完毕

（3）LR型调整



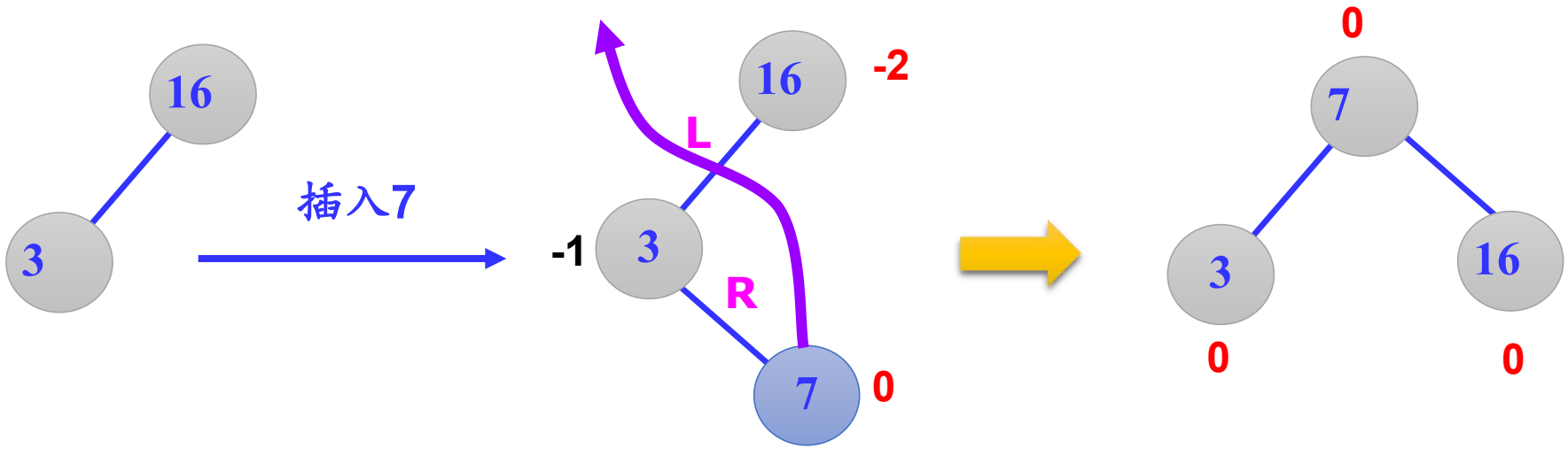


LR型调整过程：

- C结点穿过A、B结点上升
- B结点成为C的左孩子，A结点成为C的右孩子
- 原来C结点的左子树β 作为B的右子树；原来C结点的右子树γ 作为A的左子树

LR调整后的结果

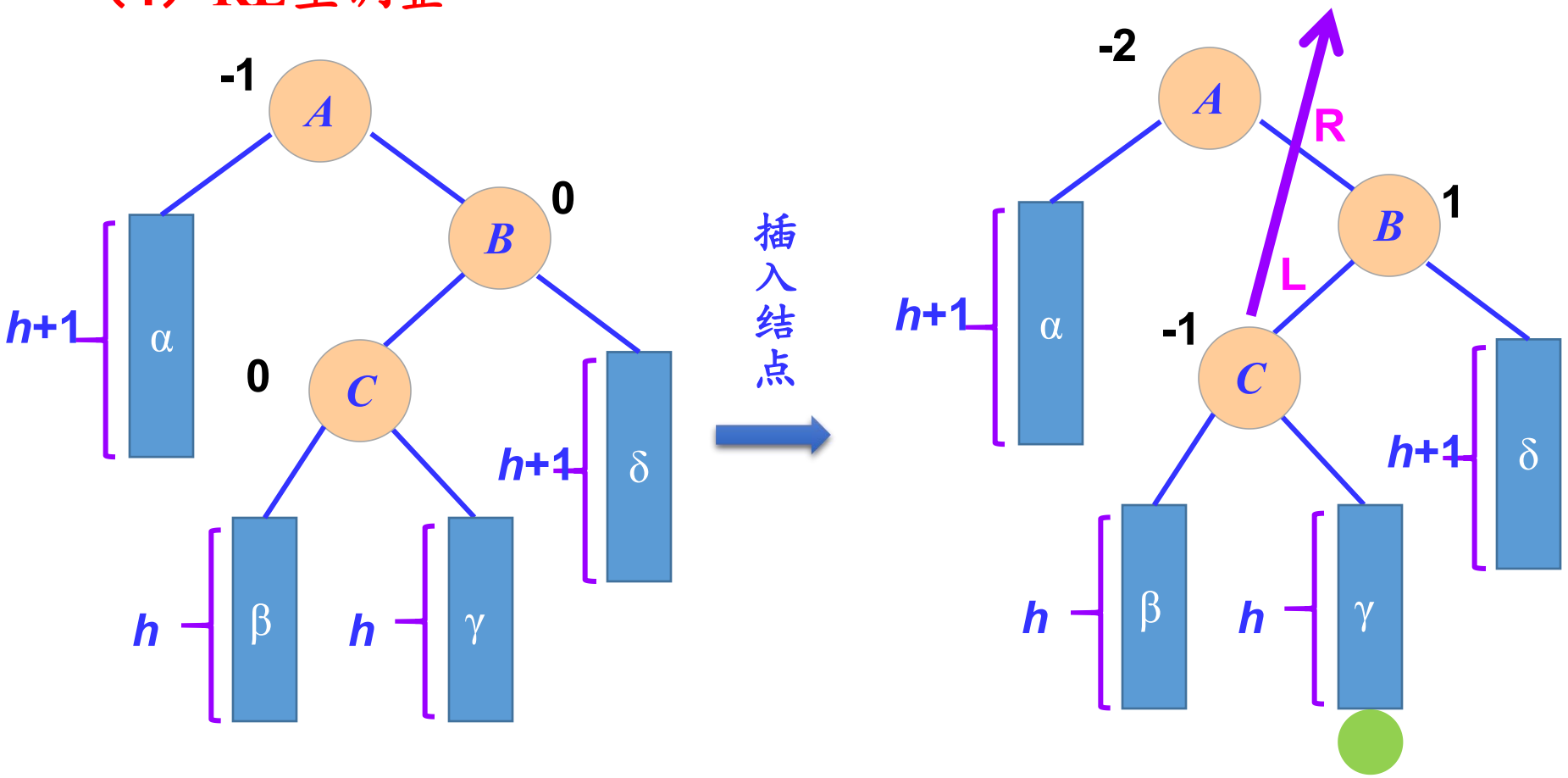
AVL树LR调整演示



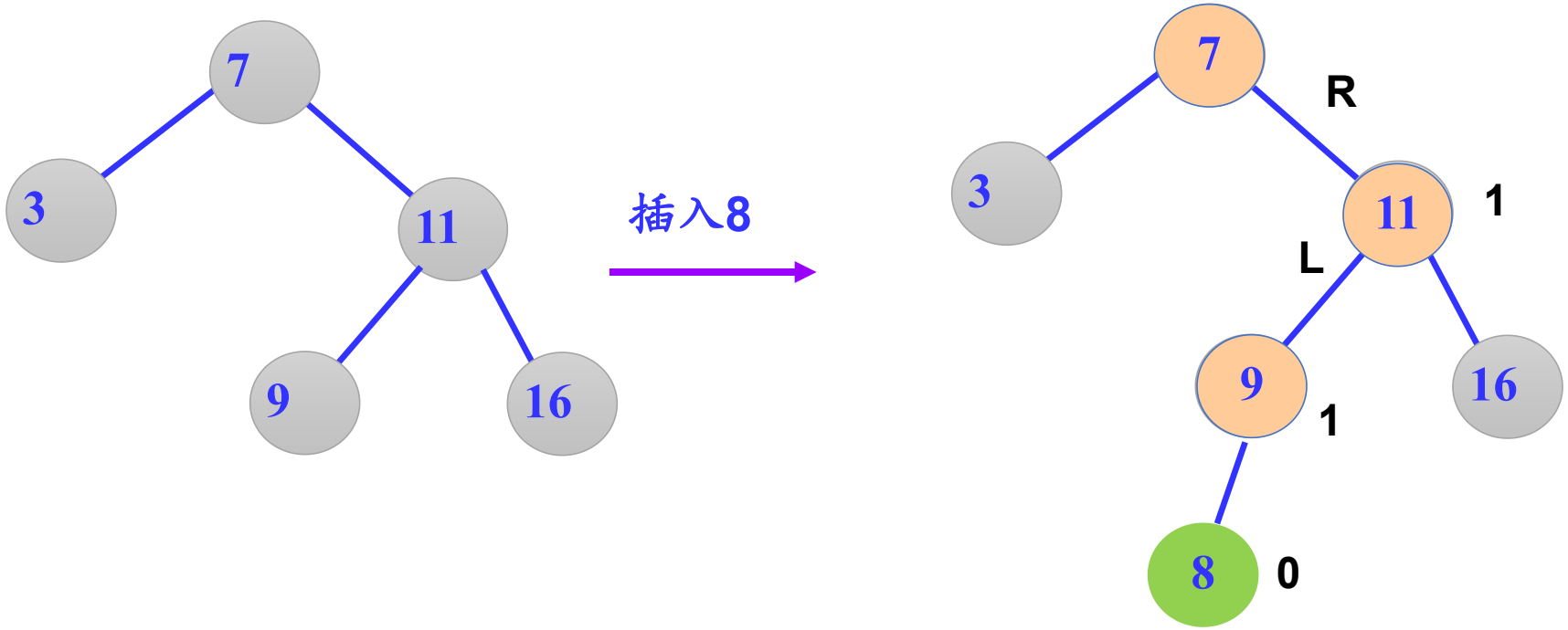
插入关键字7的结果

调整完毕

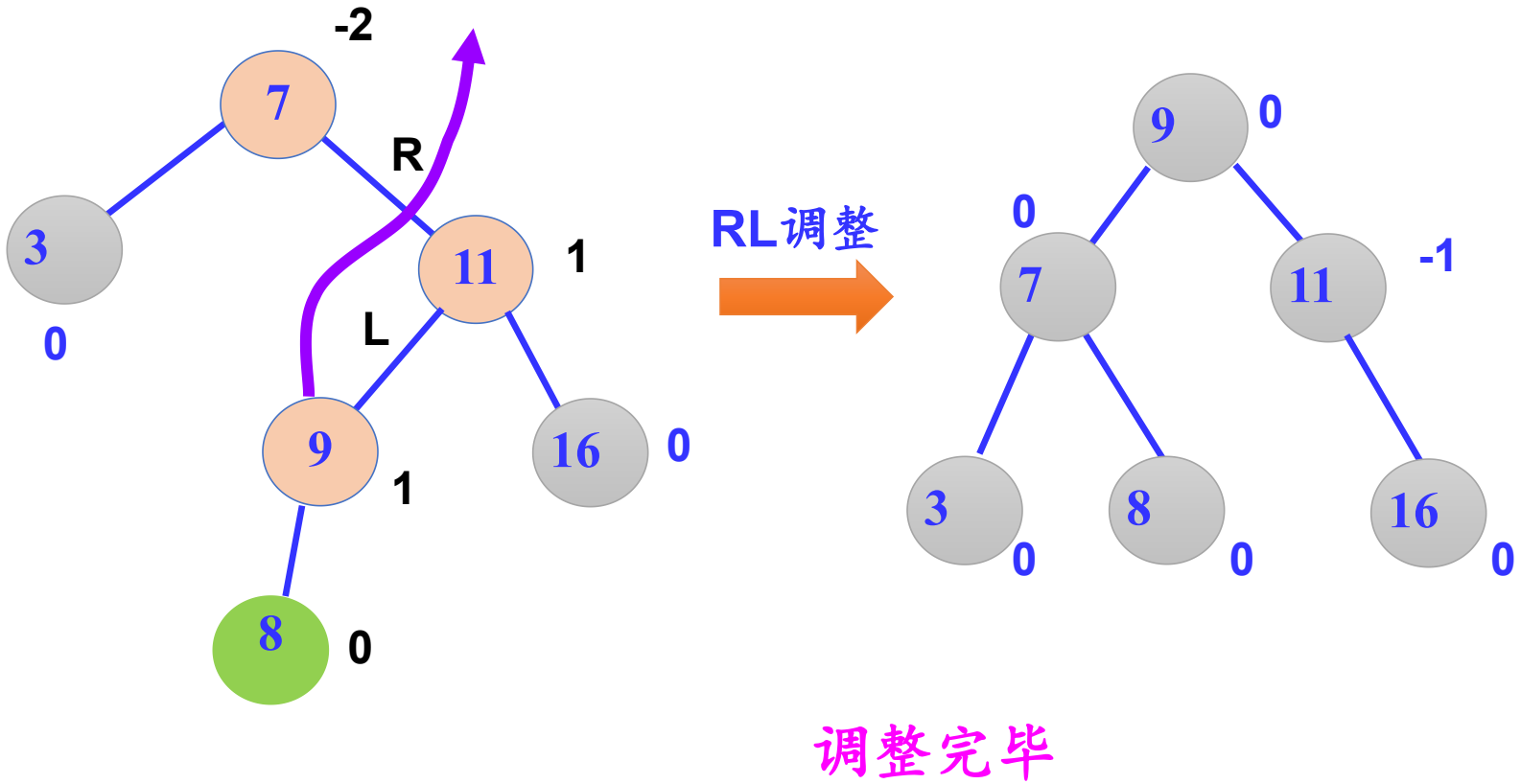
（4）RL型调整



AVL树RL调整演示

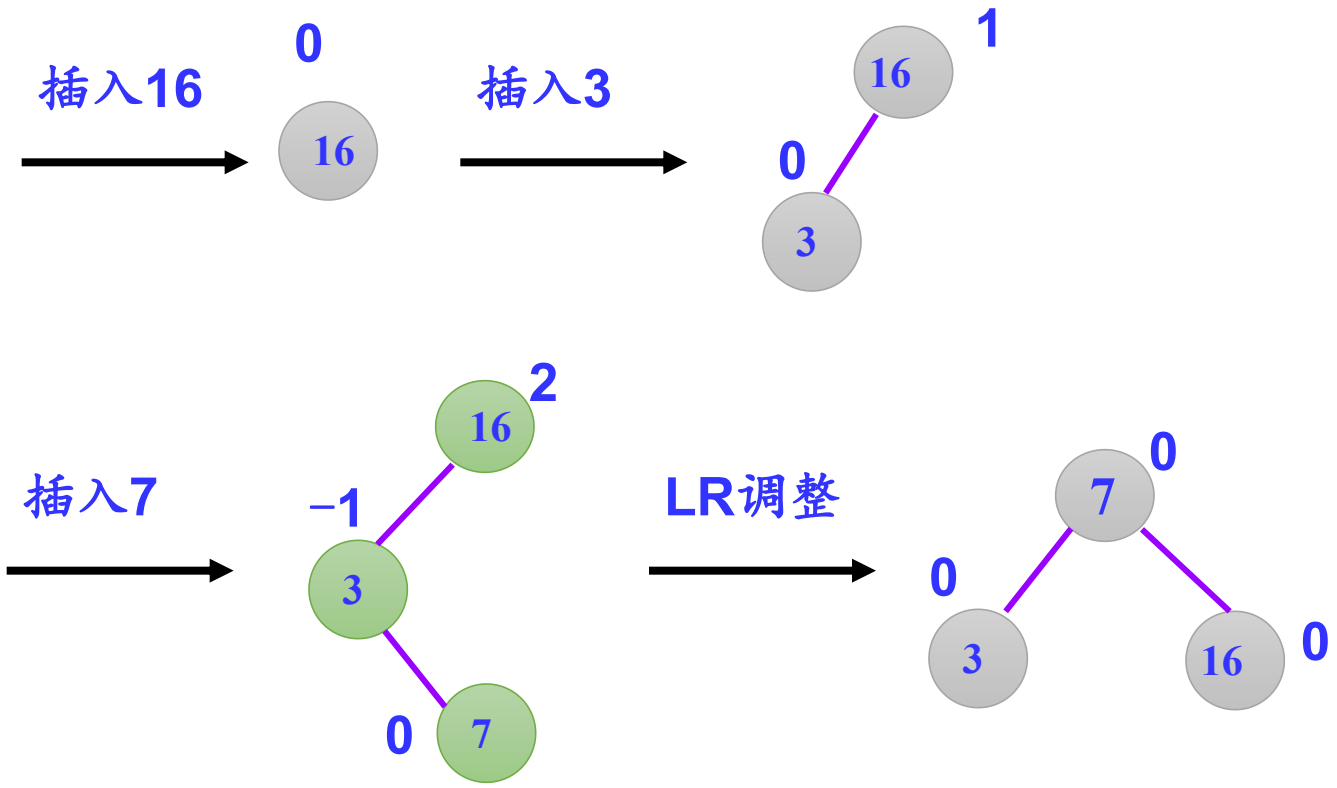


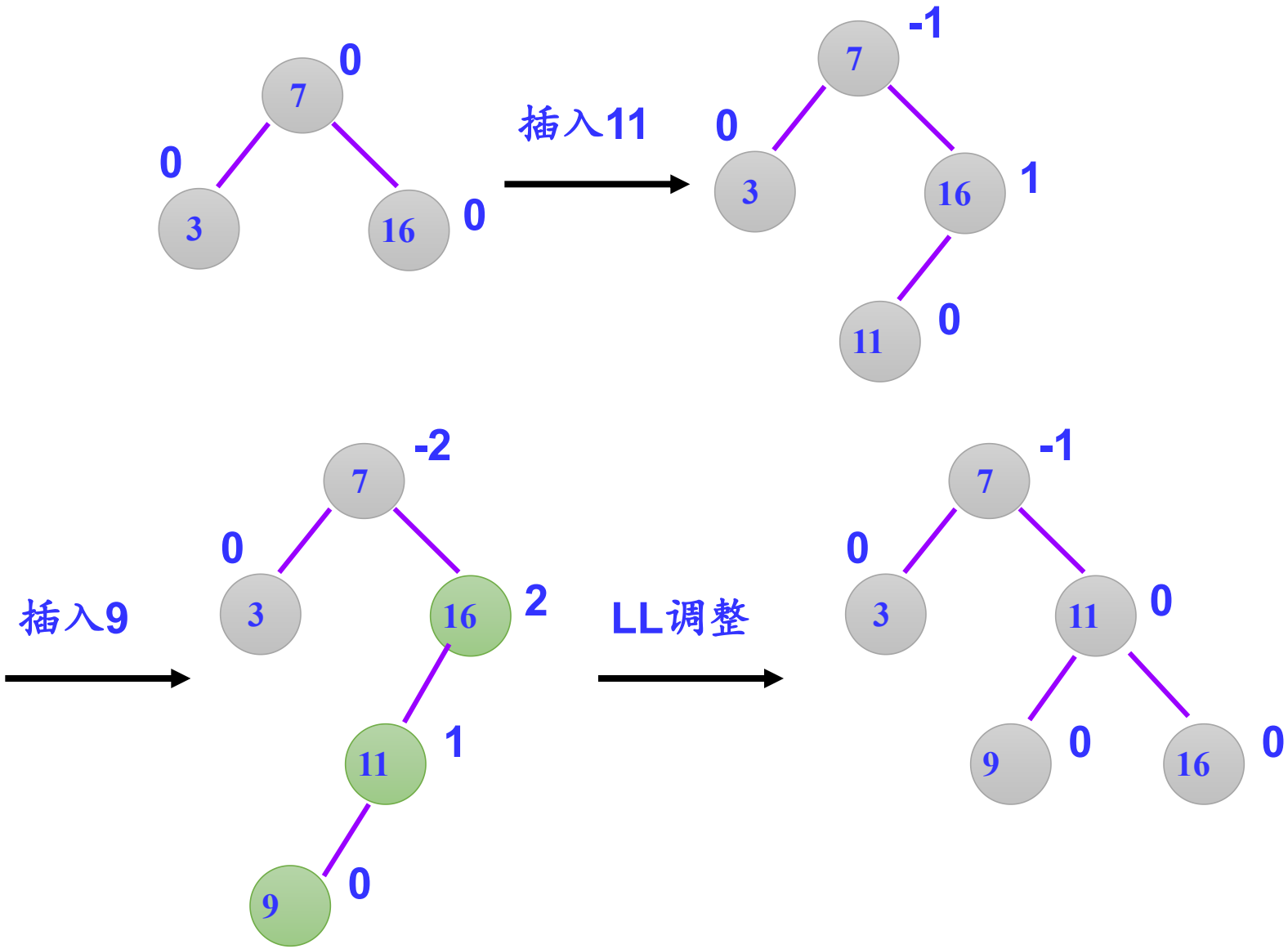
插入关键字8的结果

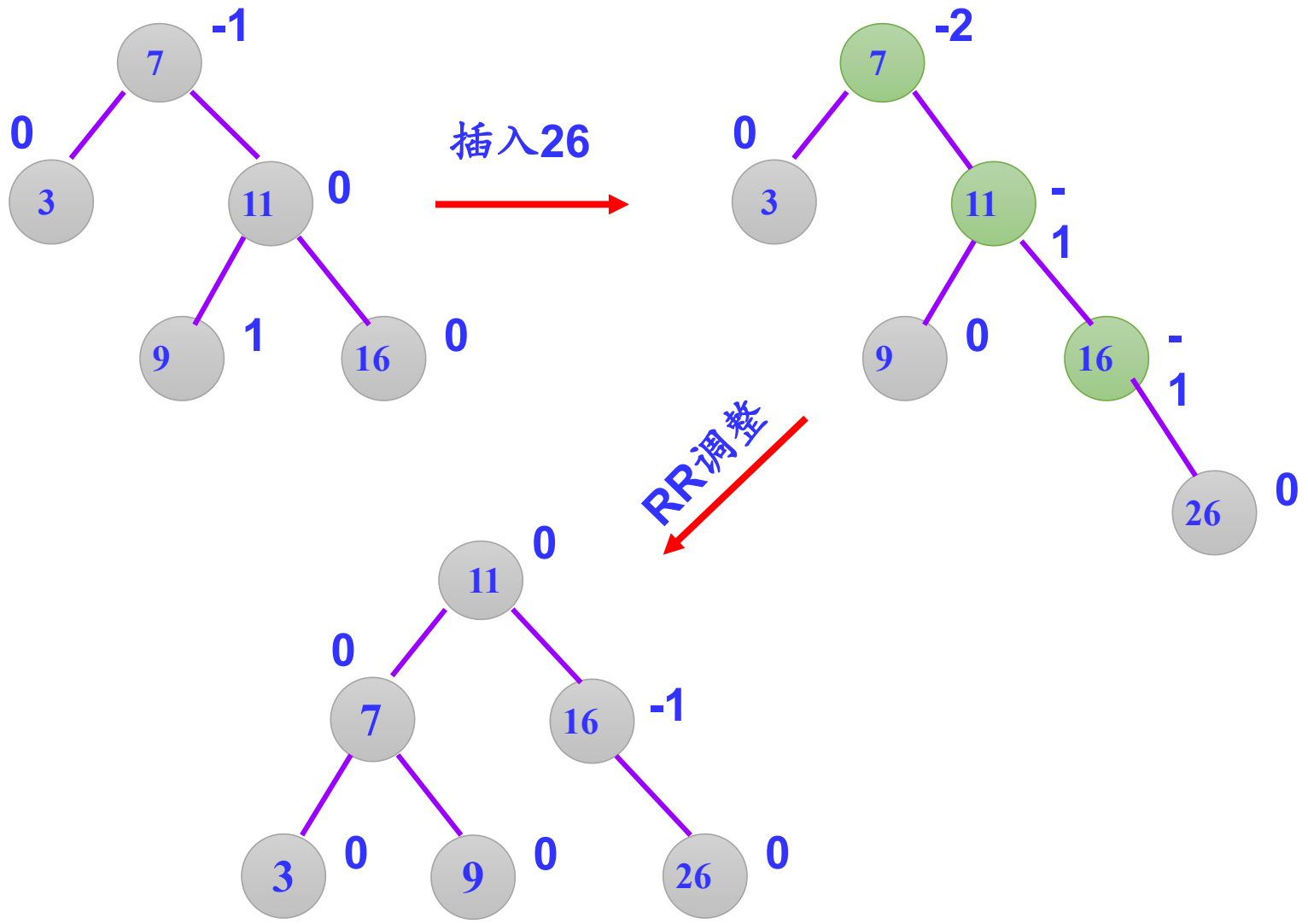


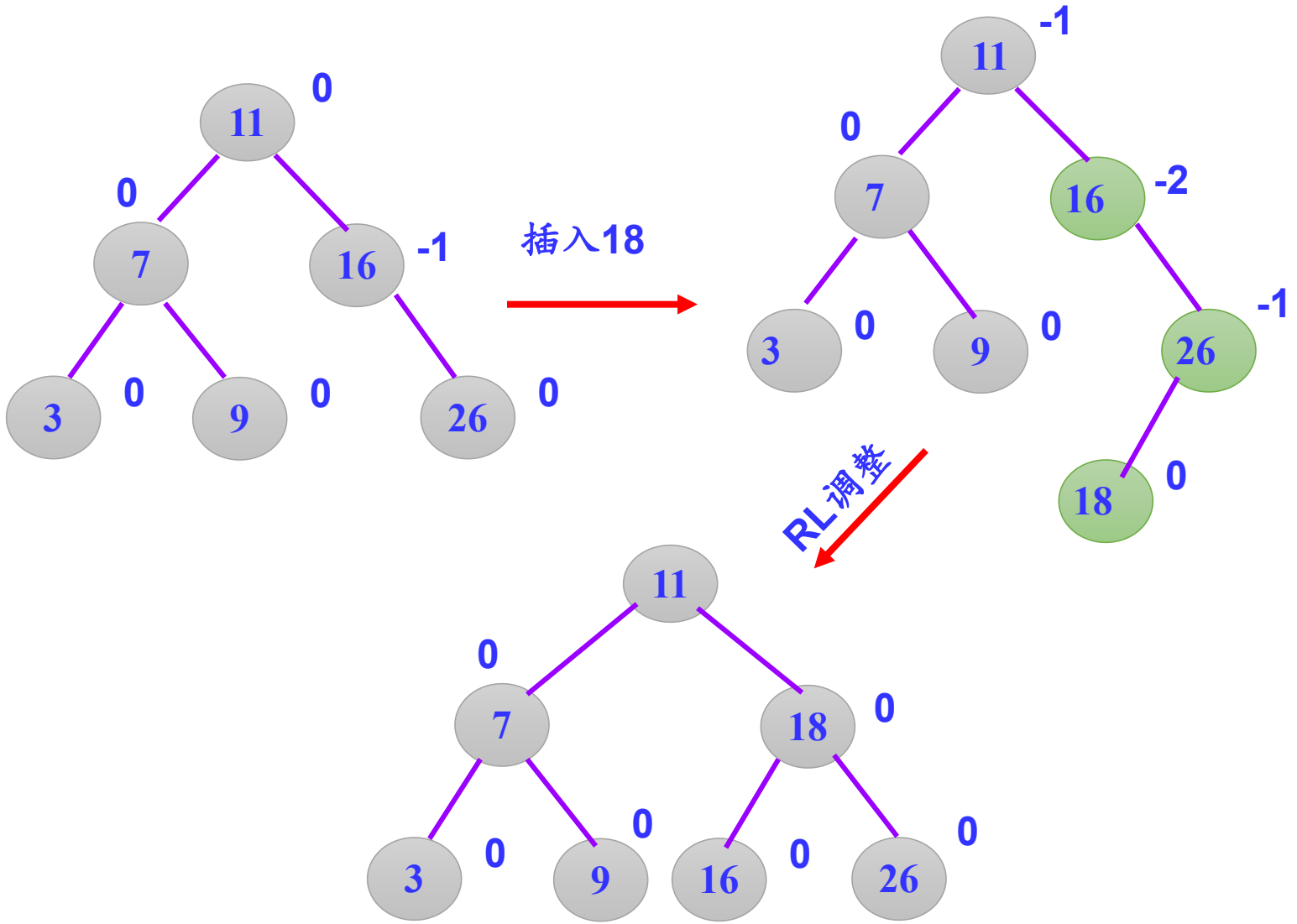
调整完毕

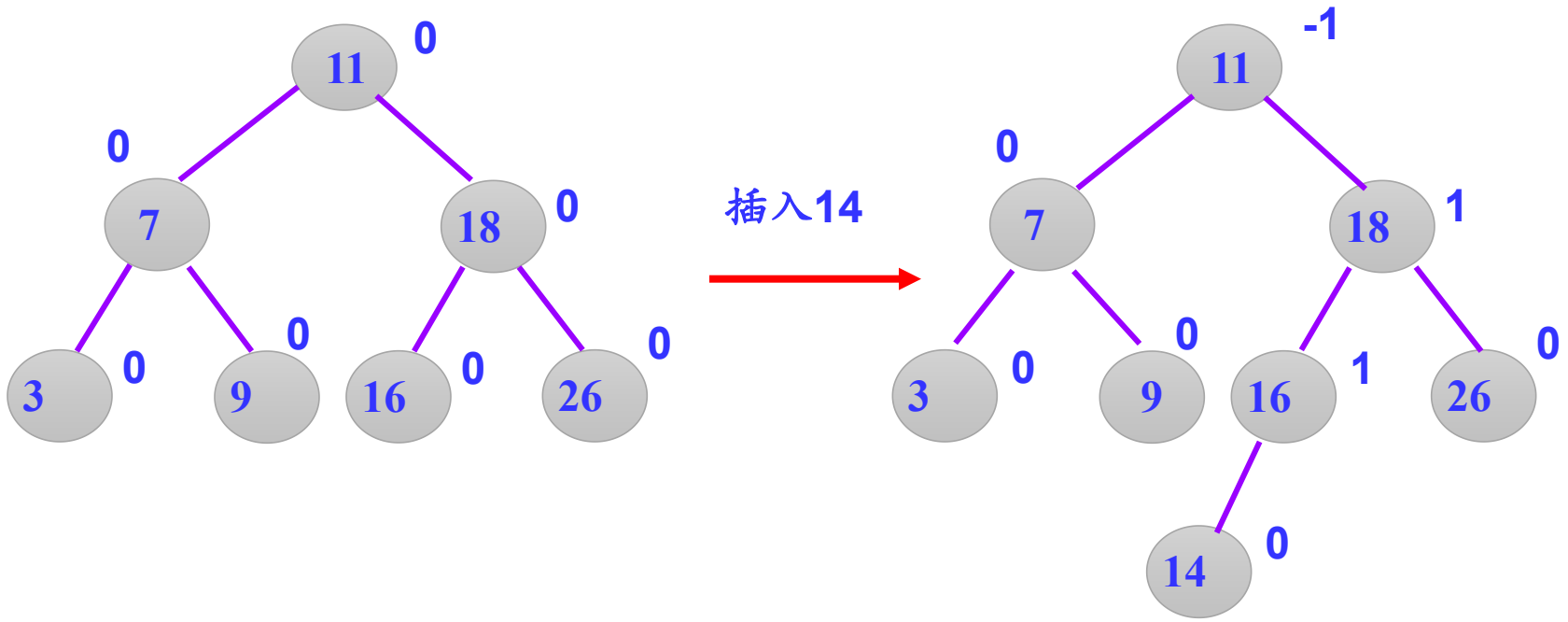
【例8-5】输入关键字序列(16，3，7，11，9，26，18，14，15)，给出构造一棵AVL树的步骤。

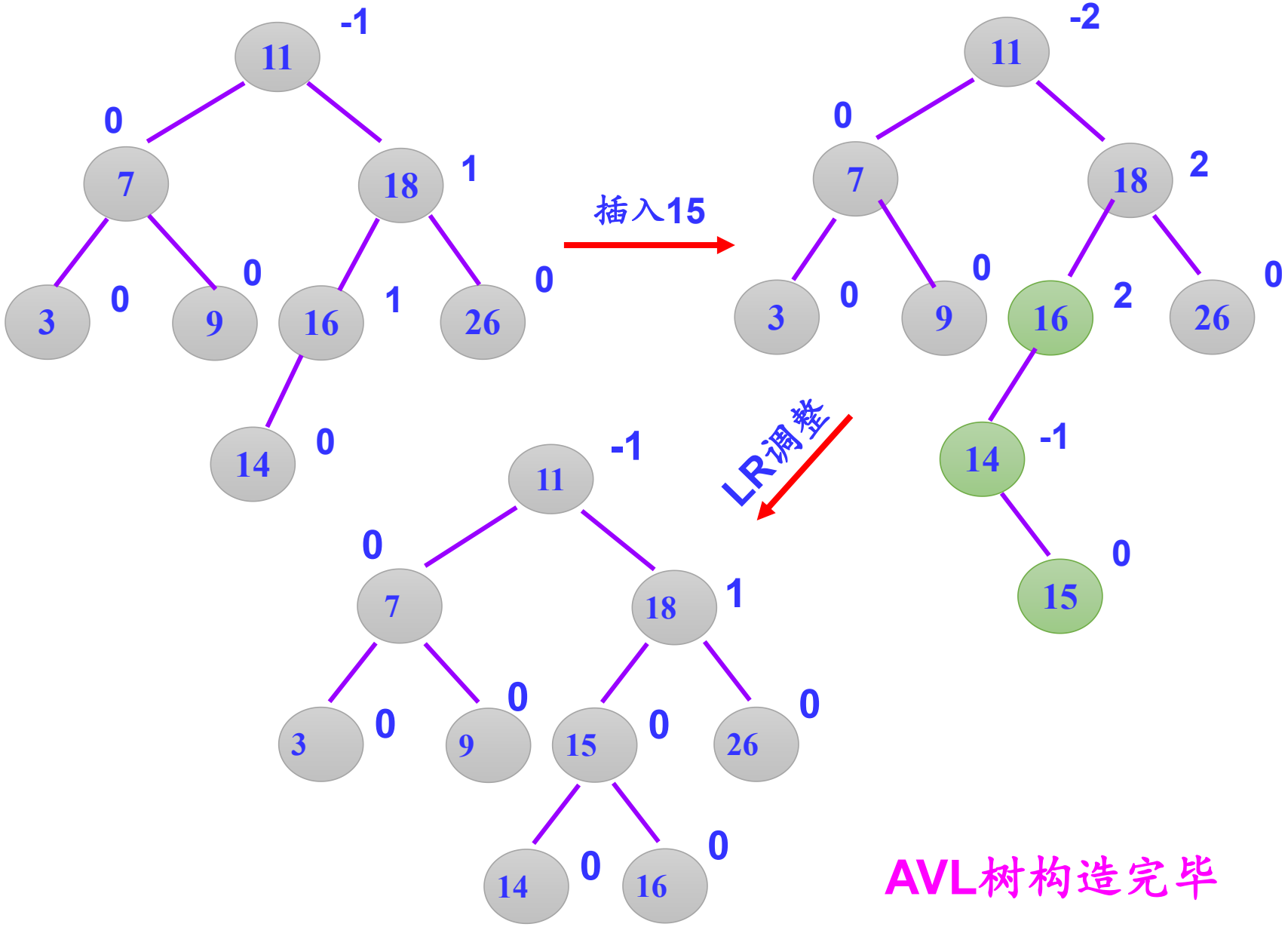












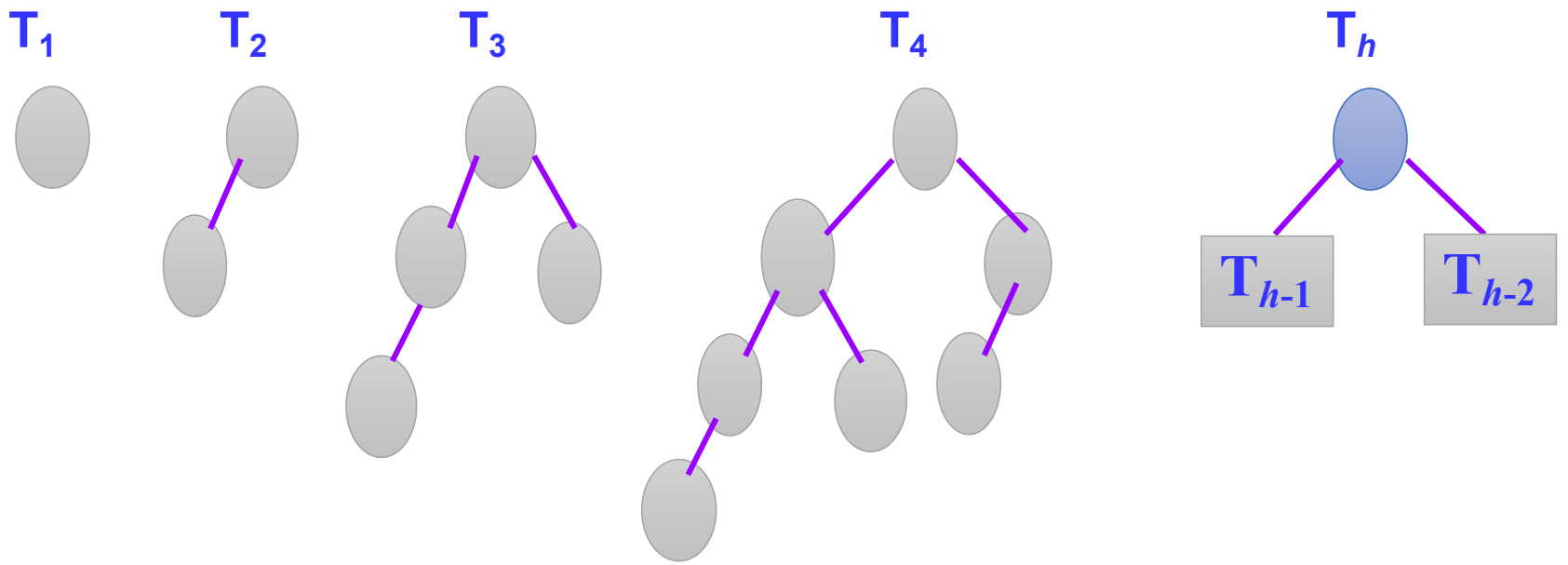
AVL树构造完毕

3、平衡二叉树的查找

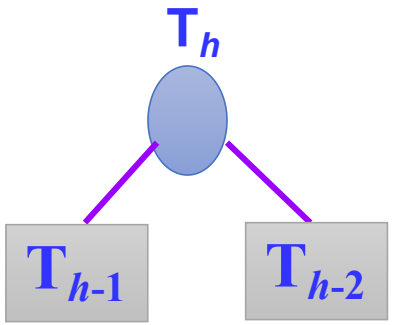
在平衡二叉树上进行查找的过程和在二叉排序树上进行查找的过程完全相同，因此在平衡二叉树上进行查找关键字的比较次数不会超过平衡二叉树的高度。

在最坏的情况下，普通二叉排序树的查找长度为$O(n)$。那么，平衡二叉树的情况又是怎样的呢？

构造一系列的平衡二叉树T_1，T_2，T_3，…，其中，T_h表示高度为h且结点数尽可能少的平衡二叉树，下图所示的T_1，T_2，T_3、T_4和T_h。



结点个数n最少的平衡二叉树



设$N(h)$为T_h的结点数，从图中可以看出有下列关系成立：

$N(1)=1$，$N(2)=2$，$N(h)=N(h-1)+N(h-2)+1$

当$h>1$时，此关系类似于定义Fibonacci数的关系：

$F(1)=1$，$F(2)=1$，$F(h)=F(h-1)+F(h-2)$

通过检查两个序列的前几项就可发现两者之间的对应关系：

$N(h)=F(h+2)-1$

由于Fibonacci数满足渐近公式：$F(h)=\frac{1}{\sqrt{5}}\phi^{h}$

其中，$\phi=\frac{1+\sqrt{5}}{2}$

故由此可得近似公式：$N(h)=\frac{1}{\sqrt{5}}\phi^{h+2}-1\approx 2^{h}-1$

即：$h\approx\log_2(N(h)+1)$

含有n个结点的平衡二叉树的平均查找长度为$O(\log_2 n)$。

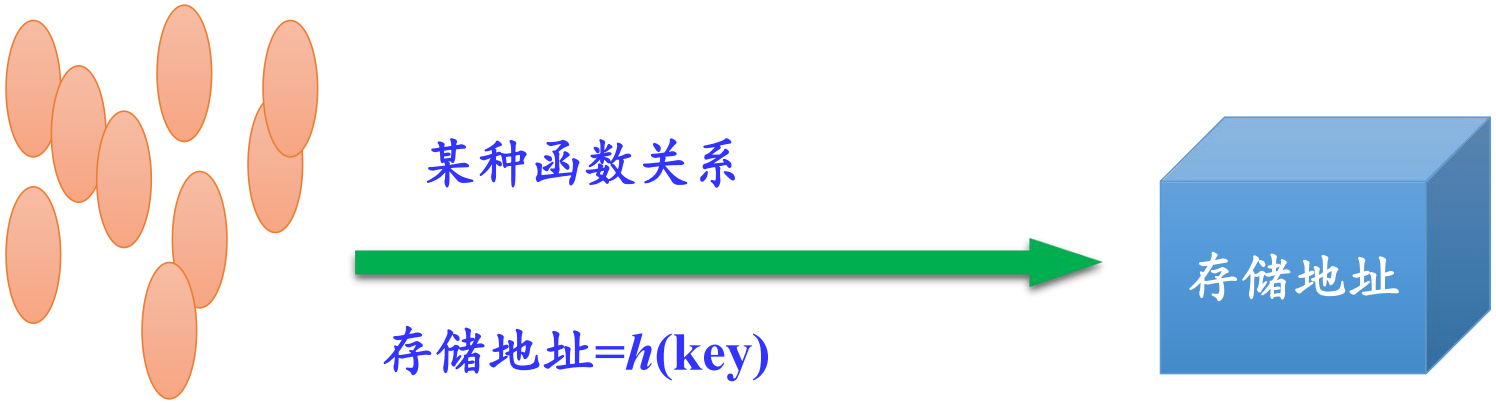
思考题

平衡二叉树和二叉排序树相比，有什么优点？

8.4 哈希表的查找

8.4.1 哈希表的基本概念

1、哈希表适合情况



注意： 哈希表是一种存储结构，它并非适合任何情况，主要适合记录的关键字与存储地址存在某种函数关系的数据。

示例

学号	姓名
201001001	张三
201001003	李四
…	
201001025	王五

记录数n=20，无序

传统存储方法:存放在一个数组中

0	1	…	19
201001001	201001003	…	201001025
张三	李四	…	王五

20个元素空间

查找学号为201001025的学生姓名：

- 从头到尾顺序查找，时间复杂度为$O(n)$。
- 若学号有序，二分查找，时间复杂度为$O(\log_2 n)$。

另一种存储结构：

学号	姓名
201001001	张三
201001003	李四
…	
201001025	王五

n=20，m=30

存放地址=学号−201001001

0	1	2	…	24	…	29
201001001	空闲	201001003	…	201001025	…	空闲
张三	空闲	李四	…	王五	…	空闲

哈希表

30个元素空间

查找学号为201001025的学生姓名：

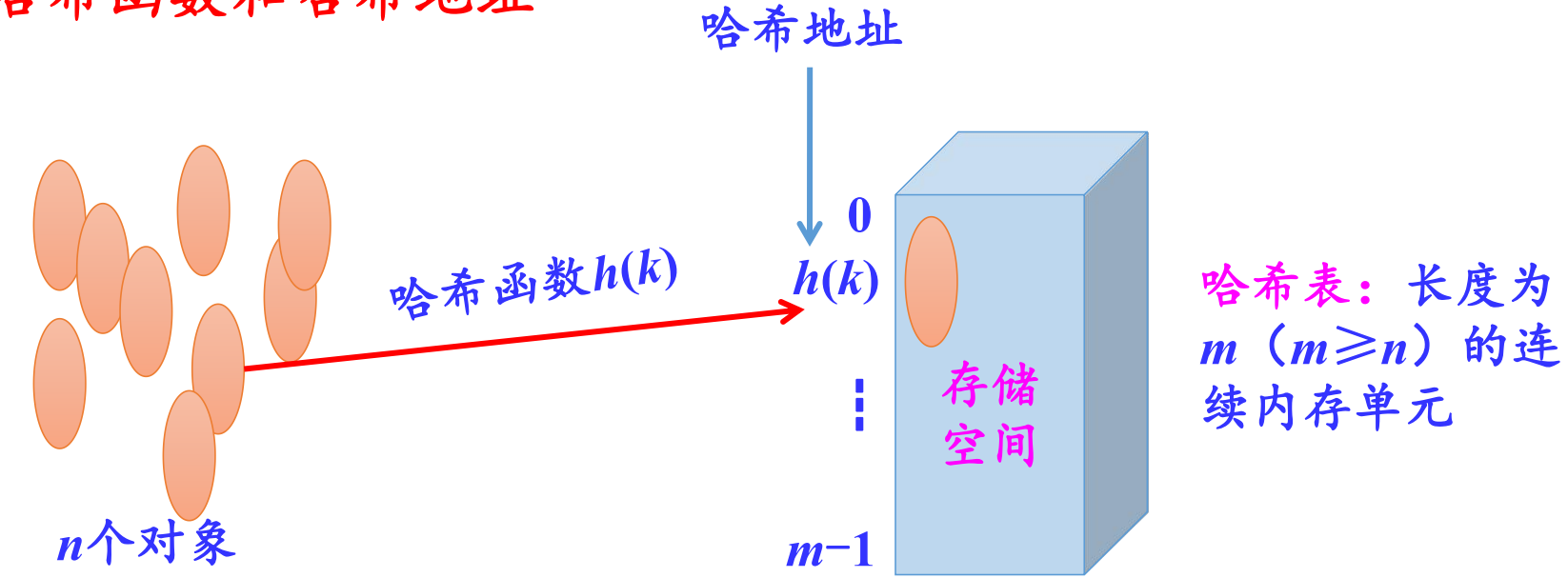
① 计算:地址d= 201001025−201001001=24

② 和24处的学号比较，相等，返回姓名“王五”

时间复杂度O(1)

2、几个概念

❶ 哈希函数和哈希地址



哈希函数：把关键字为k_i的对象存放在相应的哈希地址中

❷ 哈希冲突

对于两个关键字分别为k_i和k_j（$i \neq j$）的记录，有$k_i \neq k_j$，但$h(k_i)=h(k_j)$。把这种现象叫做**哈希冲突（同义词冲突）**。

在哈希表存储结构的存储中，哈希冲突是很难避免的！！！

3、哈希表设计

哈希表设计主要需要解决哈希冲突。实际中哈希冲突是难以避免的，主要与3个因素有关：

- ✓ 与装填因子有关。装填因子α=存储的记录个数/哈希表的大小=n/m ⇨ α越小，冲突的可能性就越小； α越大（最大可取1），冲突的可能性就越大。通常使最终的控制在0.6～0.9的范围内。
- ✓ 与所采用的哈希函数有关。好的哈希函数会减少冲突的发生；不好的哈希函数会增加冲突的发生。
- ✓ 与解决冲突方法有关。好的哈希冲突解决方法会减少冲突的发生。

所以哈希表设计的重点：

- 尽可能设计好的哈希函数
- 设计解决冲突的方法。

思考题

好的哈希函数应该具有什么特点？

8.4.2 哈希函数构造方法

1、直接定址法

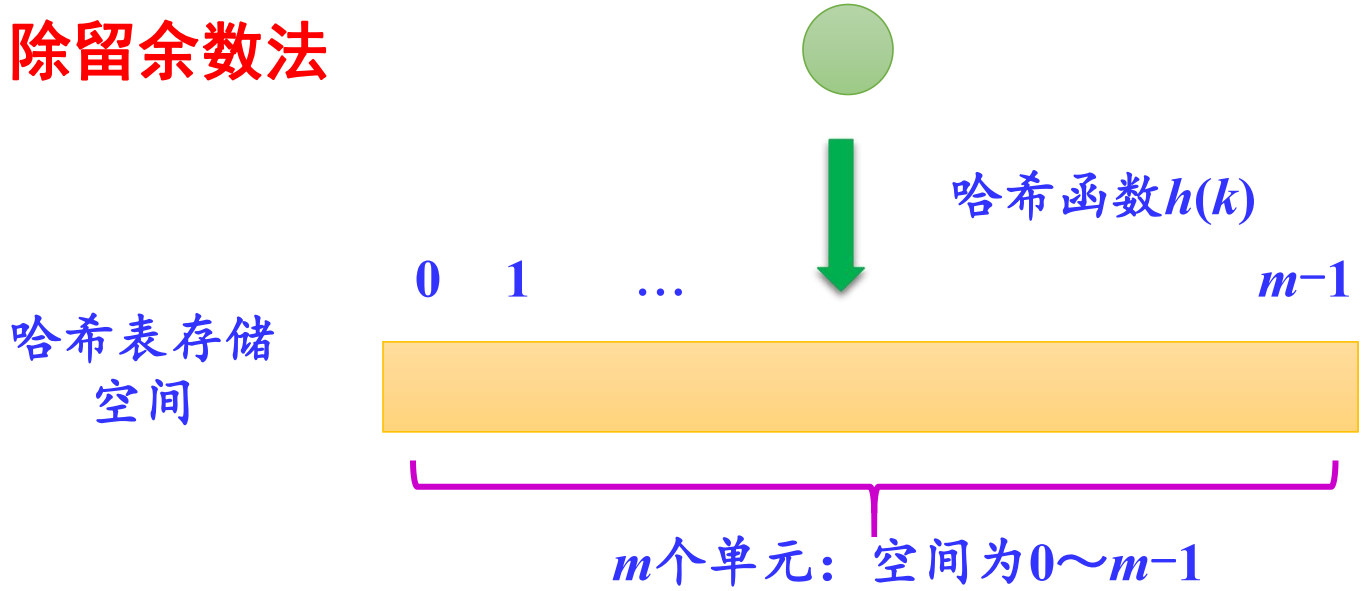
直接定址法是以关键字k本身或关键字加上某个数值常量c作为哈希地址的方法。直接定址法的哈希函数$h(k)$为：

$$h(k) = k+c$$

例如：

$$h(\text{学号}) = \text{学号}-201001001$$

2、除留余数法



除留余数法的哈希函数$h(k)$为：

$h(k)=k \bmod p$ (mod为求余运算，$p \leqslant m$)

p最好是质数（素数）。

讲解：除留余数法就是把n个记录按关键字映射的0～$m-1$的哈希空间中。而模p（素数）时出现冲突的可能性更小。

3、数字分析法

关键字

9	2	3	1	7	6	0	2
9	2	3	2	6	8	7	5
9	2	7	3	9	6	2	8
9	2	3	4	3	6	3	4
9	2	7	0	6	8	1	6
9	2	7	7	4	6	3	8
9	2	3	8	1	2	6	2
9	2	3	9	4	2	2	0

取最后两位作为哈希地址

0	2
7	5
2	8
3	4
1	6
3	8
6	2
2	0

哈希地址的集合为(2，75，28，34，16，38，62，20)。

大数值范围 —哈希函数$h(k)$→ 小数值范围

【例8-9】 假设哈希表长度m=13，采用除留余数法哈希函数建立如下关键字集合的哈希表(16，74，60，43，54，90，46，31，29，88，77)，共11个关键字。

解：n=11，m=13，设计除留余数法的哈希函数为：

$$h(k)=k \bmod p$$

p应为小于等于m的素数，设p=13。

关键字：16 74 60 43 54 90 46 31 29 88 77

h(29)=3

0	1	2	3	4	5	6	7	8	9	10	11	12
		54	16	43	31		46	60	74			90

注意：存在哈希冲突。

8.4.3 哈希冲突解决方法

1、开放定址法

开放定址法：冲突时找一个新的空闲的哈希地址。

怎么找空闲单元？

实例：晚到电影院找座位的情况就是采用开放定址法。

示例：如果你买了电影票，到电影院时已经开映了，你的位置被别人占用了，需要找一个空位置。这就是开放定址法的思路。

（1）线性探查法

线性探查法的数学递推描述公式为：

$d_0=h(k)$

$d_i=(d_{i-1}+1) \bmod m \quad (1\leqslant i\leqslant m-1)$

示例：在电影院中找被占用位置的后面空位置！模m是为了保证找到的位置在0～m-1的有效空间中。

非同义词冲突：哈希函数值不相同的两个记录争夺同一个后继哈希地址 ⇨ 堆积（或聚集）现象。

（2）平方探查法

平方探查法的数学描述公式为：

$d_0=\mathrm{h}(k)$

$d_i=(d_0\pm i^2) \bmod m \quad (1\leqslant i\leqslant m-1)$

查找的位置依次为：d_0、d_0+1、d_0-1、d_0+4、d_0-4、…

思路：在电影院中找被占用位置的前后空位置！

平方探查法是一种较好的处理冲突的方法，可以避免出现堆积现象。它的缺点是不能探查到哈希表上的所有单元，但至少能探查到一半单元。

【例8-10】 假设哈希表长度m=13，采用除留余数法哈希函数建立如下关键字集合的哈希表：

(16，74，60，43，54，90，46，31，29，88，77)。

并采用线性探查法解决冲突。

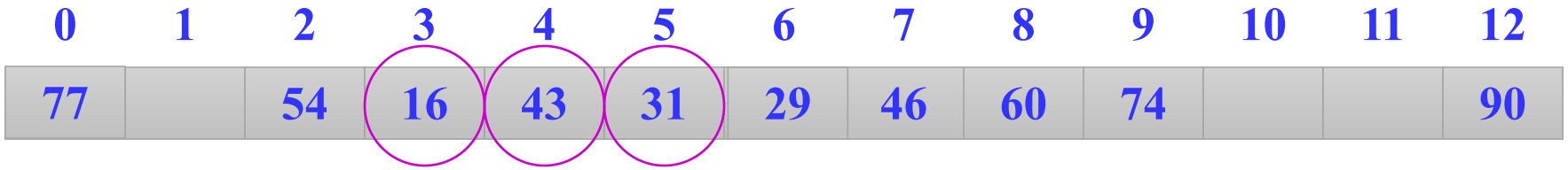
关键字：16 74 60 43 54 90 46 31 29 88 77

h(29)=3 ⇨冲突

d_0=3，d_1=(3+1) % 13=4 ⇨仍冲突

d_2=(4+1) % 13=5 ⇨仍冲突

d_3=(5+1) % 13=6 ⇨ OK



共探查4次

关键字：16 74 60 43 54 90 46 31 29 88 77

h(88)=10

0	1	2	3	4	5	6	7	8	9	10	11	12
77		54	16	43	31	29	46	60	74	88		90

共探查1次

关键字：16 74 60 43 54 90 46 31 29 88 77

h(77)=12 ⇨冲突

d_0=12，d_1=(12+1) % 13=0 ⇨ OK

0	1	2	3	4	5	6	7	8	9	10	11	12
77		54	16	43	31	29	46	60	74	88		90

共探查2次

哈希表创建完毕

最终的哈希表

哈希表ha[0..12]

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	77		54	16	43	31	29	46	60	74	88		90
探查次数	2		1	1	1	1	4	1	1	1	1		1

哈希表的构成：

- 哈希函数： 本例为$h(k)=k \bmod 13$
- 解决冲突方法：本例为线性探查法

开放定址法哈希表查找k过程：

```
d=h(k);
while (ha[d]!=空 && ha[d]!=k)
    d=采用某种探查法求出下一地址;
if (ha[d]==空)
    return 失败标记;
else
    return ha[d];
```

成功查找的情况

查找关键字为29的记录：

$h(29)=29\%13=3$：16≠29；

$d_0=3$，$d_1=(3+1)=4$：43≠29；

$d_2=(4+1)=5$：31≠29；

$d_3=(5+1)=6$：29＝29。成功！　　需要4次关键字比较

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	77		54	16	43	31	29	46	60	74	88		90
探查次数	2		1	1	1	1	4	1	1	1	1		1

哈希表成功查找完毕

❶ 对于前面构建的哈希表：成功查找ASL计算

哈希表ha[0..12]

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	77		54	16	43	31	29	46	60	74	88		90
探查次数	2		1	1	1	1	4	1	1	1	1		1

探查次数恰好等于查找到该记录所需要的关键字比较次数！

$$ASL_{成功}=\frac{2+1+1+1+1+4+1+1+1+1+1}{11}=1.364$$

不成功查找的情况

查找关键字x=47的记录

$h(47)=47\%13=8$： ⇨ 60≠47

$d_0=8$，$d_1=(8+1)=9$： ⇨ 74≠47

$d_2=(9+1)=10$： ⇨ 88≠47

$d_3=(10+1)=11$： 此处为空，查找失败！

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	77		54	16	43	31	29	46	60	74	88		90
探查次数	2		1	1	1	1	4	1	1	1	1		1

需要4次关键字比较

哈希表失败查找完毕

❷ 对于前面构建的哈希表：不成功查找ASL计算

哈希表ha[0..12]

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	77		54	16	43	31	29	46	60	74	88		90
探查次数	2	1	10	9	8	7	6	5	4	3	2	1	3

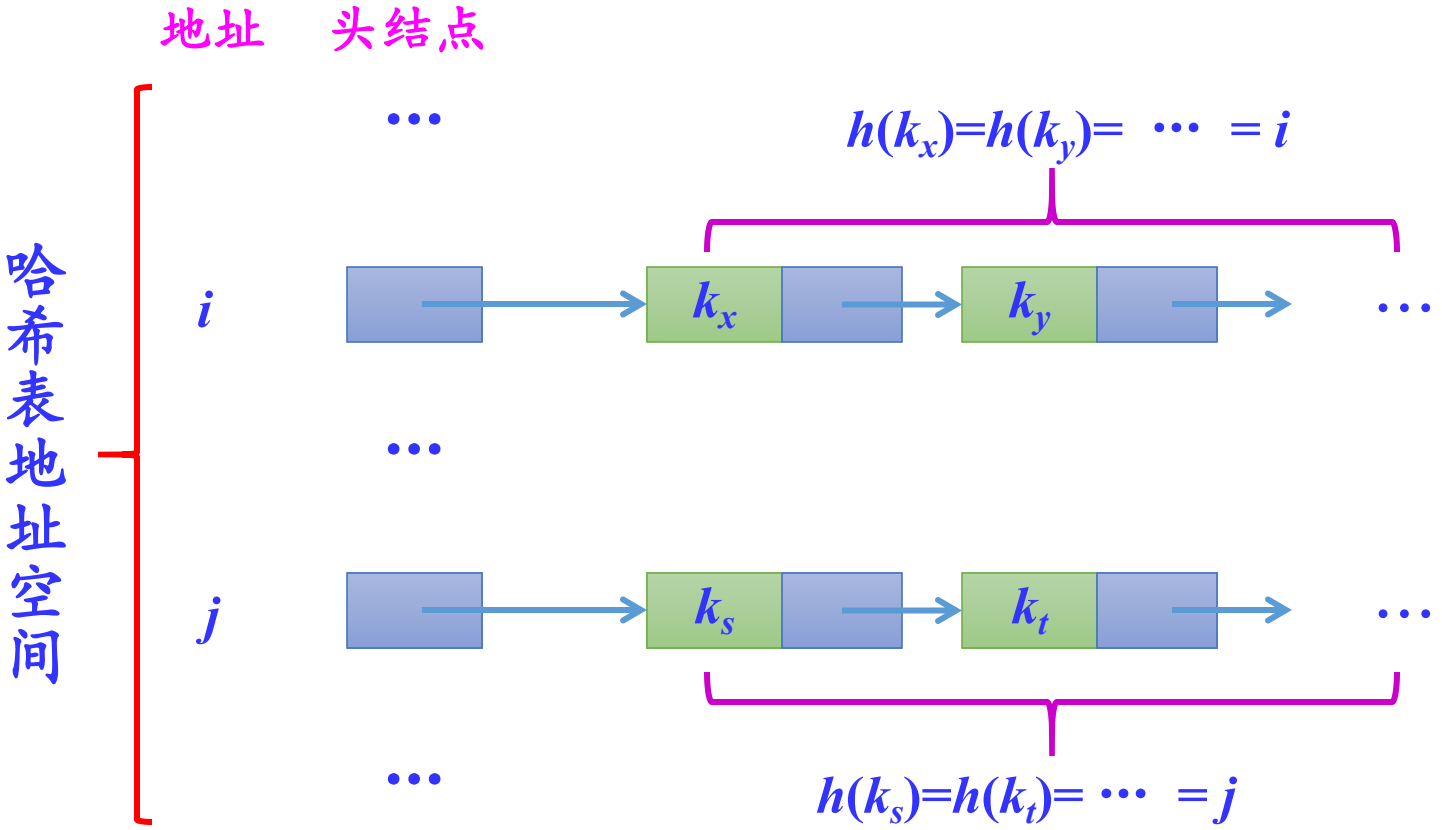
一个关键字不在有效关键字集合中，但其哈希函数值为2，采用线性探查法找到空位置，需要10次关键字比较

一个关键字不在有效关键字集合中，但其哈希函数值为0，采用线性探查法找到空位置，需要2次关键字比较

$$ASL_{不成功}=\frac{2+1+10+9+8+7+6+5+4+3+2+1+3}{13}=4.692$$

2、拉链法

拉链法是把所有的同义词用单链表链接起来的方法。

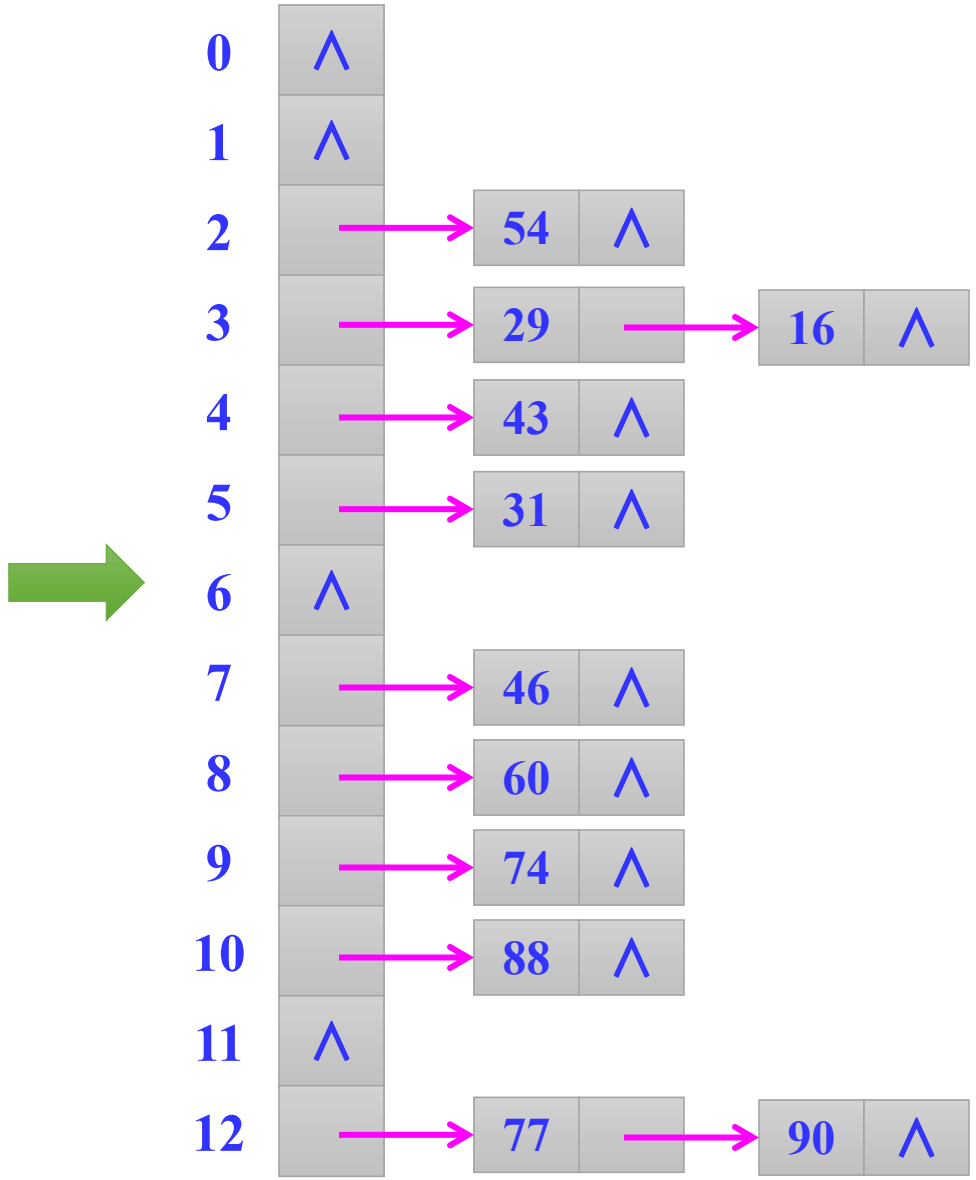


【例（补充）】 对例8-9的关键字序列，构造采用拉链法解决冲突的哈希表。

关键字集合：

(16，74，60，43，54，90，46，31，29，88，77)

采用拉链法构造的哈希表ha[0..12]



拉链法哈希表查找k过程：

```
d=h(k);
p=ha[d];
while (p!=NULL && p->key!=k)
     p=p->next;   //在ha[d]的单链表中查找
if (p==NULL)
     return 失败标记;
else
   return p所指结点;
```

成功查找的情况

查找关键字为16的记录：

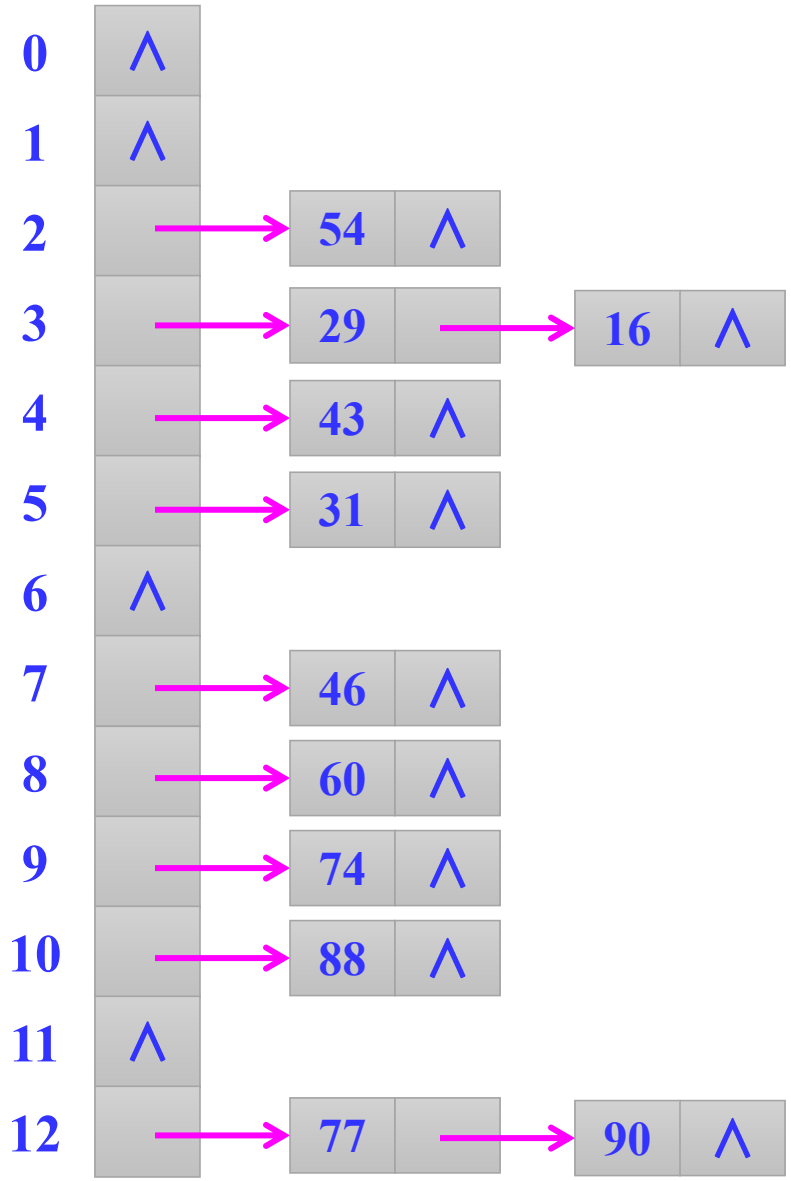
$h(16)=16\%13=3$

p指向ha[3]的第1个结点，29≠16；

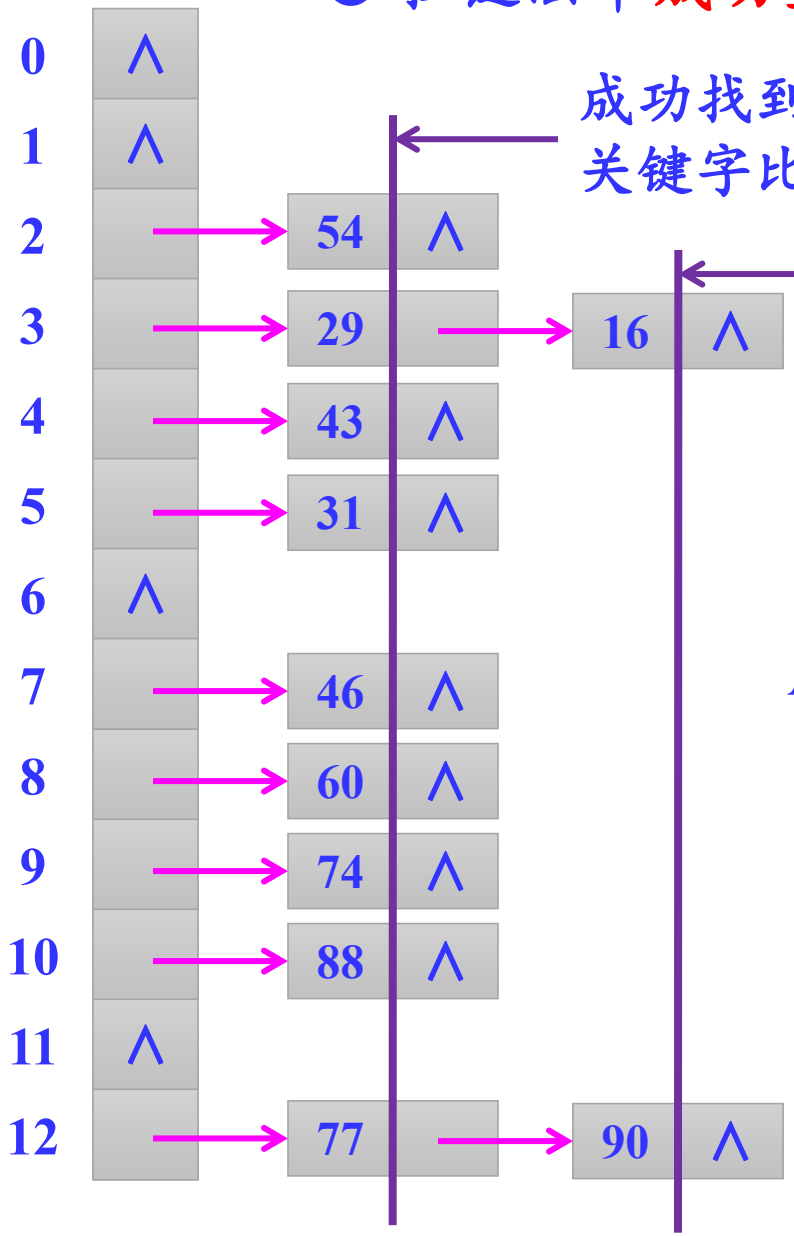
p指向ha[3]的第2个结点， 16＝16。成功！

2次关键字比较

哈希表成功查找完毕



❶ 拉链法中成功查找的ASL计算



成功找到第1层的结点，均需要1次关键字比较，共9个结点

成功找到第2层的结点，均需要2次关键字比较，共2个结点

$$ASL_{成功}=\frac{1\times 9+2\times 2}{11}=1.182$$

不成功查找的情况

查找关键字为47的记录：

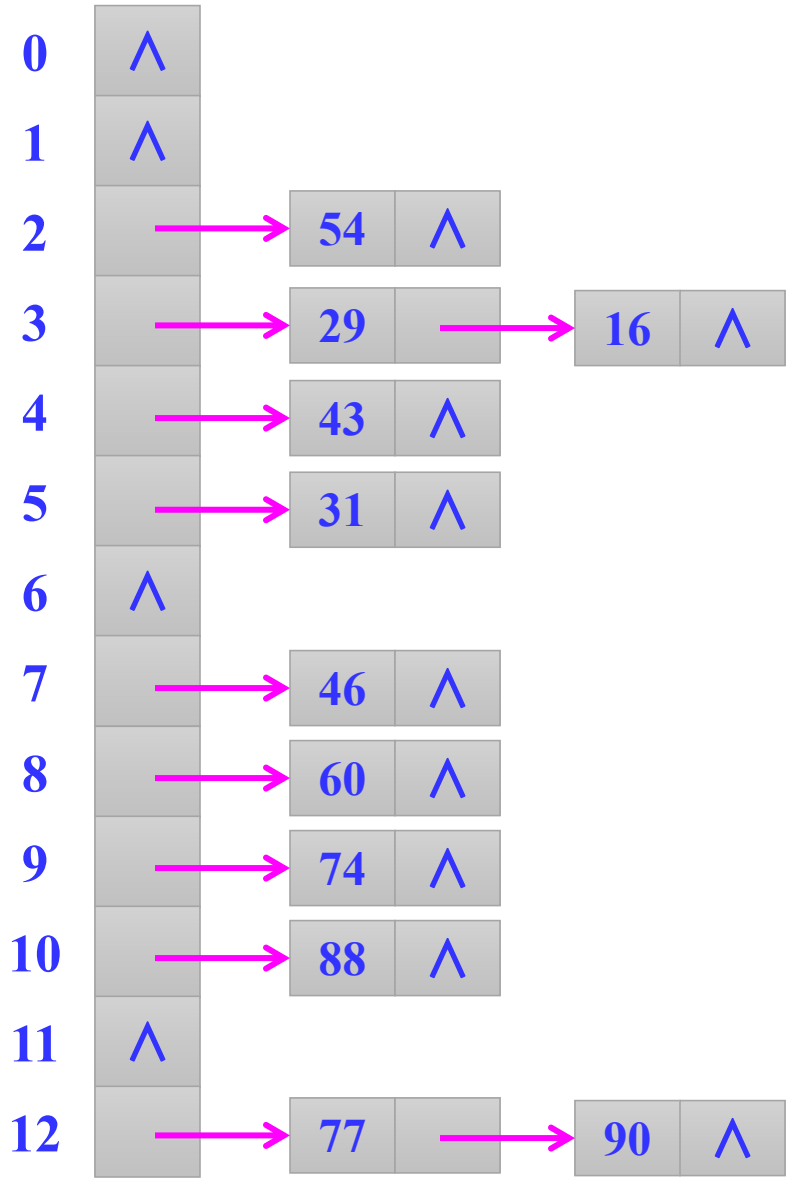
$h(47)=47\%13=8$

p指向ha[8]的第1个结点，60≠47；

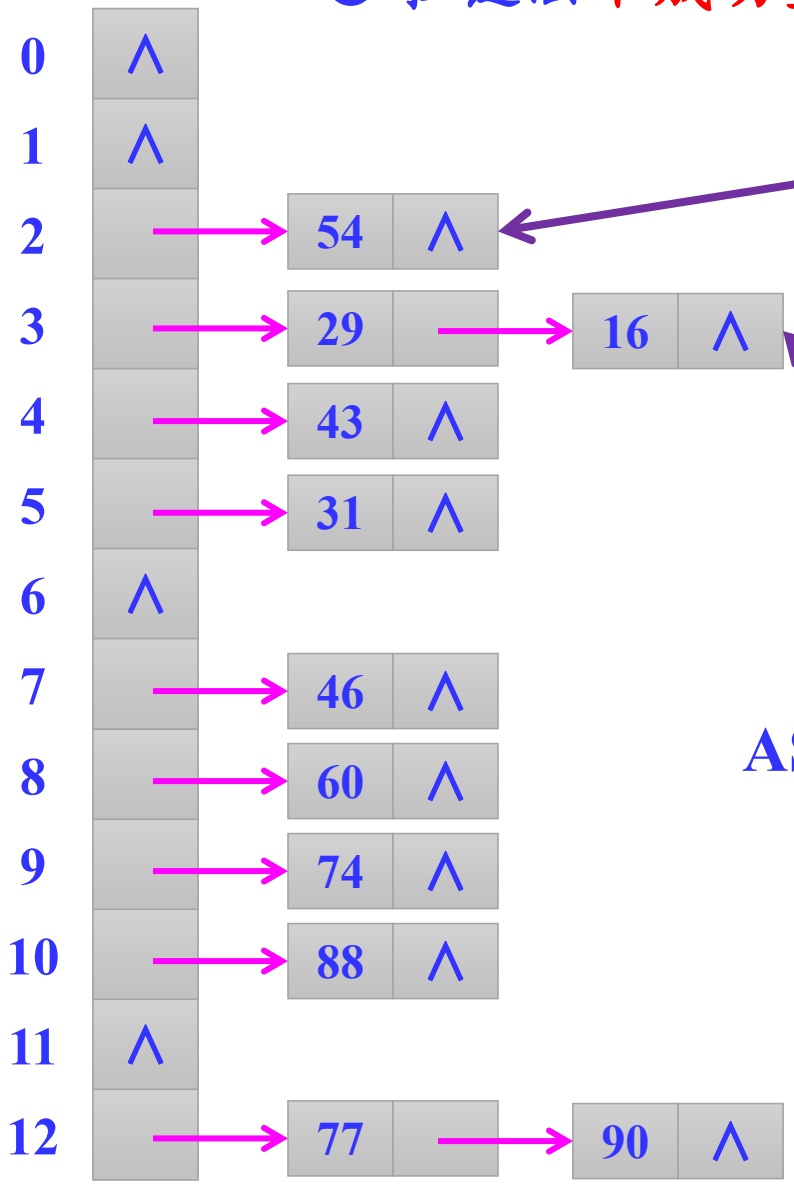
p=NULL。失败！

1次关键字比较

哈希表不成功查找完毕



❷ 拉链法不成功查找的ASL计算



有1个结点的单链表，不成功查找需要1次关键字比较，共有7个这样的单链表

有2个结点的单链表，不成功查找需要2次关键字比较，共有2个这样的单链表

$$ASL_{不成功}=\frac{1\times 7+2\times 2}{13}=0.846$$

$\alpha=n/m$

思考题

开放定址法和拉链法各有什么优缺点？

本章结束