第 7 章 查找技术

本章的主要内容是:

- 查找的基本概念
- 线性表的查找技术
- 树表的查找技术
- 散列表的查找技术

7.1 概述

查找的基本概念

关键码：可以标识一个记录的某个数据项。

键值：关键码的值。

主关键码：可以唯一地标识一个记录的关键码。

次关键码：不能唯一地标识一个记录的关键码。

职工号	姓名	性别	年龄	参加工作
0001	王刚	男	38	1990年4月
0002	张亮	男	25	2003年7月
0003	刘楠	女	47	1979年9月
0004	齐梅	女	25	2003年7月
0005	李爽	女	50	1972年9月

7.1 概述

查找的基本概念

查找：在具有相同类型的记录构成的集合中找出满足给定条件的记录。

查找的结果：若在查找集合中找到了与给定值相匹配的记录，则称查找成功；否则，称查找失败。

职工号	姓名	性别	年龄	参加工作
0001	王刚	男	38	1990年4月
0002	张亮	男	25	2003年7月
0003	刘楠	女	47	1979年9月
0004	齐梅	女	25	2003年7月
0005	李爽	女	50	1972年9月

7.1 概述

查找的基本概念

静态查找：不涉及插入和删除操作的查找 。

动态查找：涉及插入和删除操作的查找。

静态查找适用于：查找集合一经生成，便只对其进行查找，而不进行插入和删除操作，或经过一段时间的查找之后，集中地进行插入和删除等修改操作；

动态查找适用于：查找与插入和删除操作在同一个阶段进行，例如当查找成功时，要删除查找到的记录，当查找不成功时，要插入被查找的记录。

7.1 概述

查找的基本概念

查找结构：面向查找操作的数据结构，即查找基于的数据结构。

查找结构 → 查找方法

集合中元素之间不存在明显的组织规律，不便查找。

集合 ⇨ 线性表
树 表
散列表

7.1 概述

查找的基本概念

查找结构：面向查找操作的数据结构，即查找基于的数据结构。

本章讨论的查找结构：

线性表：适用于静态查找，主要采用顺序查找技术、折半查找技术。

树表：适用于动态查找，主要采用二叉排序树的查找技术。

散列表：静态查找和动态查找均适用，主要采用散列技术。

7.1 概述

查找算法的性能

查找算法时间性能通过关键码的比较次数来度量。

? 关键码的比较次数与哪些因素有关呢？

(1)算法；
(2)问题规模；
(3)待查关键码在查找集合中的位置；
(4)查找频率。

查找频率与算法无关，取决于具体应用。

通常假设p_i是已知的。

7.1 概述

查找算法的性能

查找算法时间性能通过关键码的比较次数来度量。

? 同一查找集合、同一查找算法，关键码的比较次数与哪些因素有关呢？

查找算法的时间复杂度是问题规模n和待查关键码在查找集合中的位置k的函数，记为$T(n，k)$。

7.1 概述

查找算法的性能

平均查找长度：将查找算法进行的关键码的比较次数的数学期望值定义为平均查找长度。计算公式为：

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

其中：n：问题规模，查找集合中的记录个数；

p_i：查找第i个记录的概率；

c_i：查找第i个记录所需的关键码的比较次数。

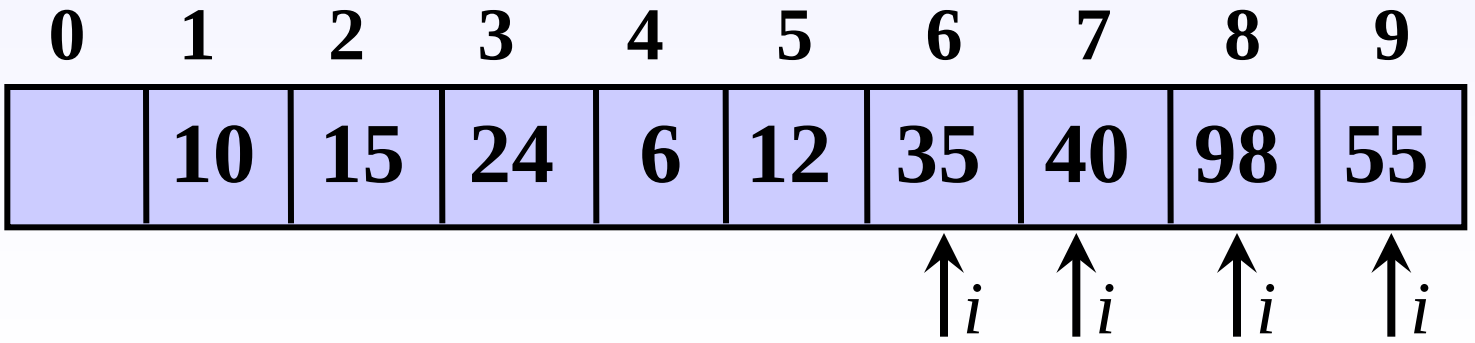
结论：c_i取决于算法；p_i与算法无关，取决于具体应用。如果p_i是已知的，则平均查找长度只是问题规模的函数。

7.2 线性表的查找技术

顺序查找（线性查找）

基本思想：从线性表的一端向另一端逐个将关键码与给定值进行比较，若相等，则查找成功，给出该记录在表中的位置；若整个表检测完仍未找到与给定值相等的关键码，则查找失败，给出失败信息。

例：查找k＝35



7.2 线性表的查找技术

顺序查找（线性查找）

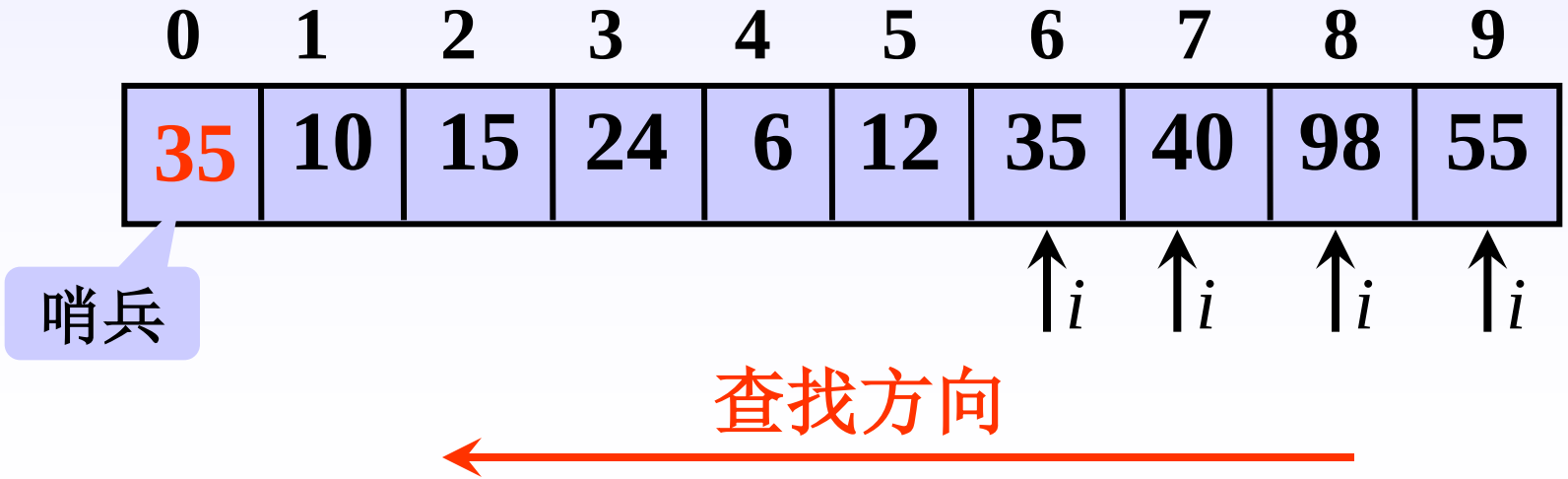
```
int SeqSearch1(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    i=n;
    while(i>0 && r[i]!=k)
        i--;
    return i;
}
```

7.2 线性表的查找技术

改进的顺序查找

基本思想：设置“哨兵”。哨兵就是待查值，将它放在查找方向的尽头处，免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度

例：查找k＝35

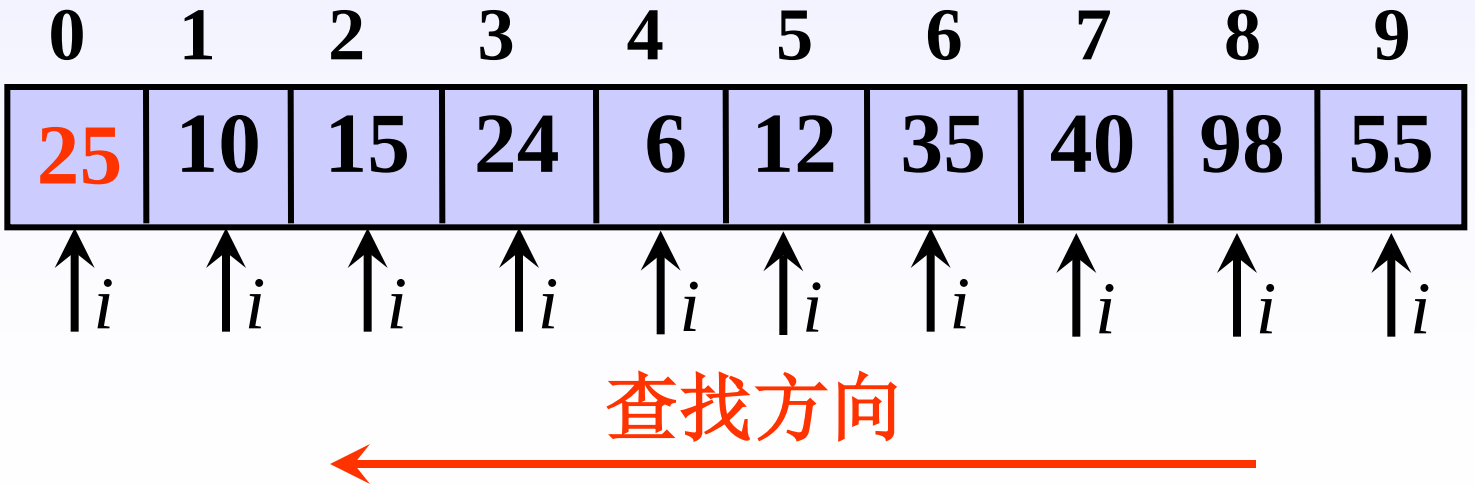


7.2 线性表的查找技术

改进的顺序查找

基本思想：设置“哨兵”。哨兵就是待查值，将它放在查找方向的尽头处，免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度

例：查找k＝25



7.2 线性表的查找技术

改进的顺序查找

```
int SeqSearch2(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    r[0]=k; i=n;
    while (r[i]!=k)
      i --;
    return i;
}
```

```
int SeqSearch1（int r[ ], int n, int k）
//数组r[1] ~ r[n]存放查找集合
{
    i=n;
    while（i>0 && r[i]!=k）
      i−−;
    return i;
}
```

$$ASL=\sum_{i=1}^{n} p_i\, c_i = \sum_{i=1}^{n} p_i\,(n-i+1) = (n+1)/2=O(n)$$

7.2 线性表的查找技术

顺序查找的缺点：

平均查找长度较大，特别是当待查找集合中元素较多时，查找效率较低。

顺序查找的优点：

算法简单而且使用面广。

- 对表中记录的存储没有任何要求，顺序存储和链接存储均可；
- 对表中记录的有序性也没有要求，无论记录是否按关键码有序均可。

7.2 线性表的查找技术

折半查找

使用条件：

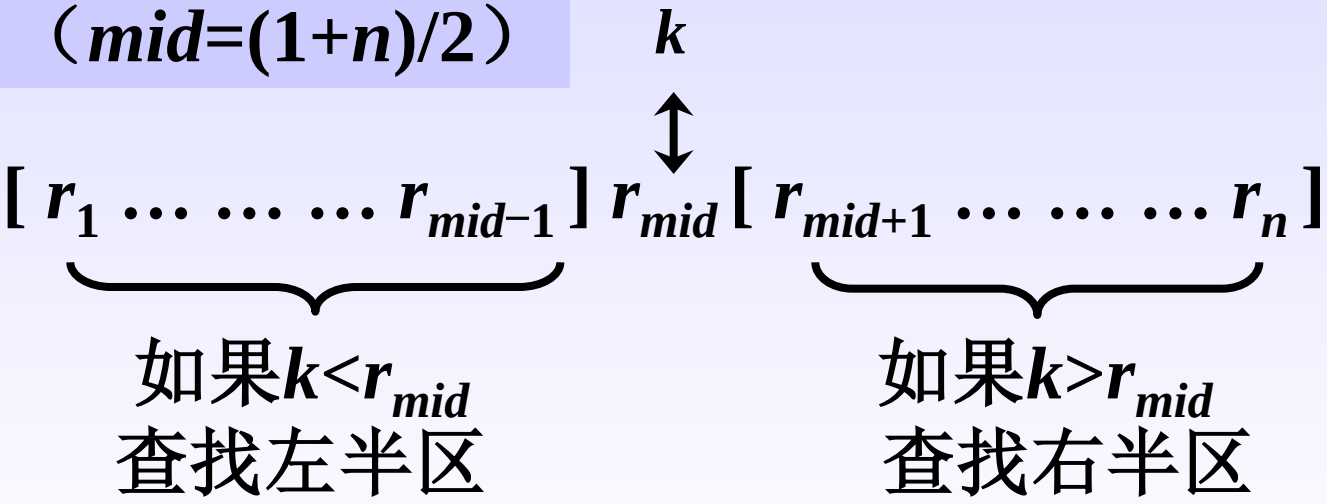
- 线性表中的记录必须按关键码有序；
- 必须采用顺序存储。

基本思想：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

7.2 线性表的查找技术

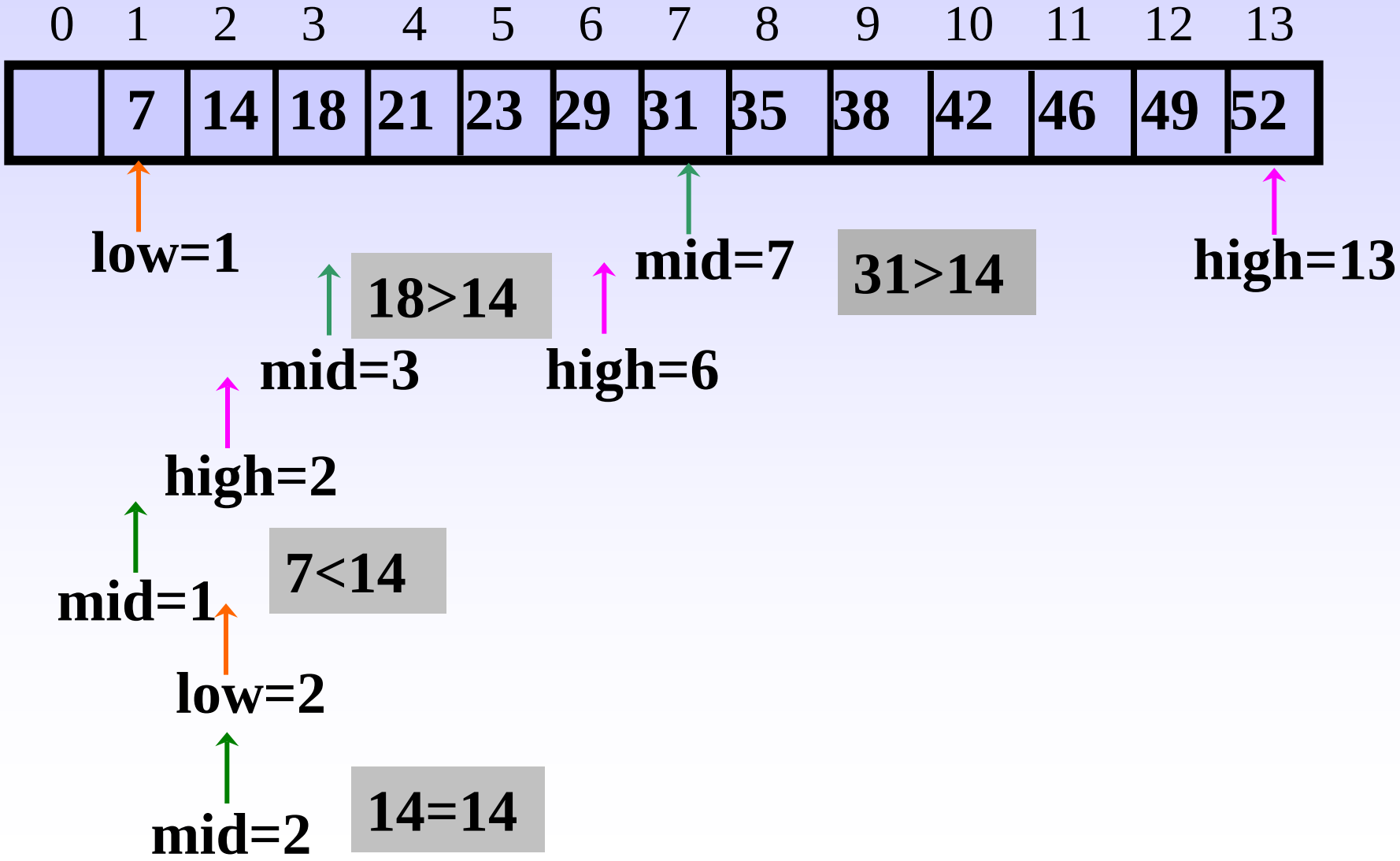
折半查找的基本思想

（$mid=(1+n)/2$）



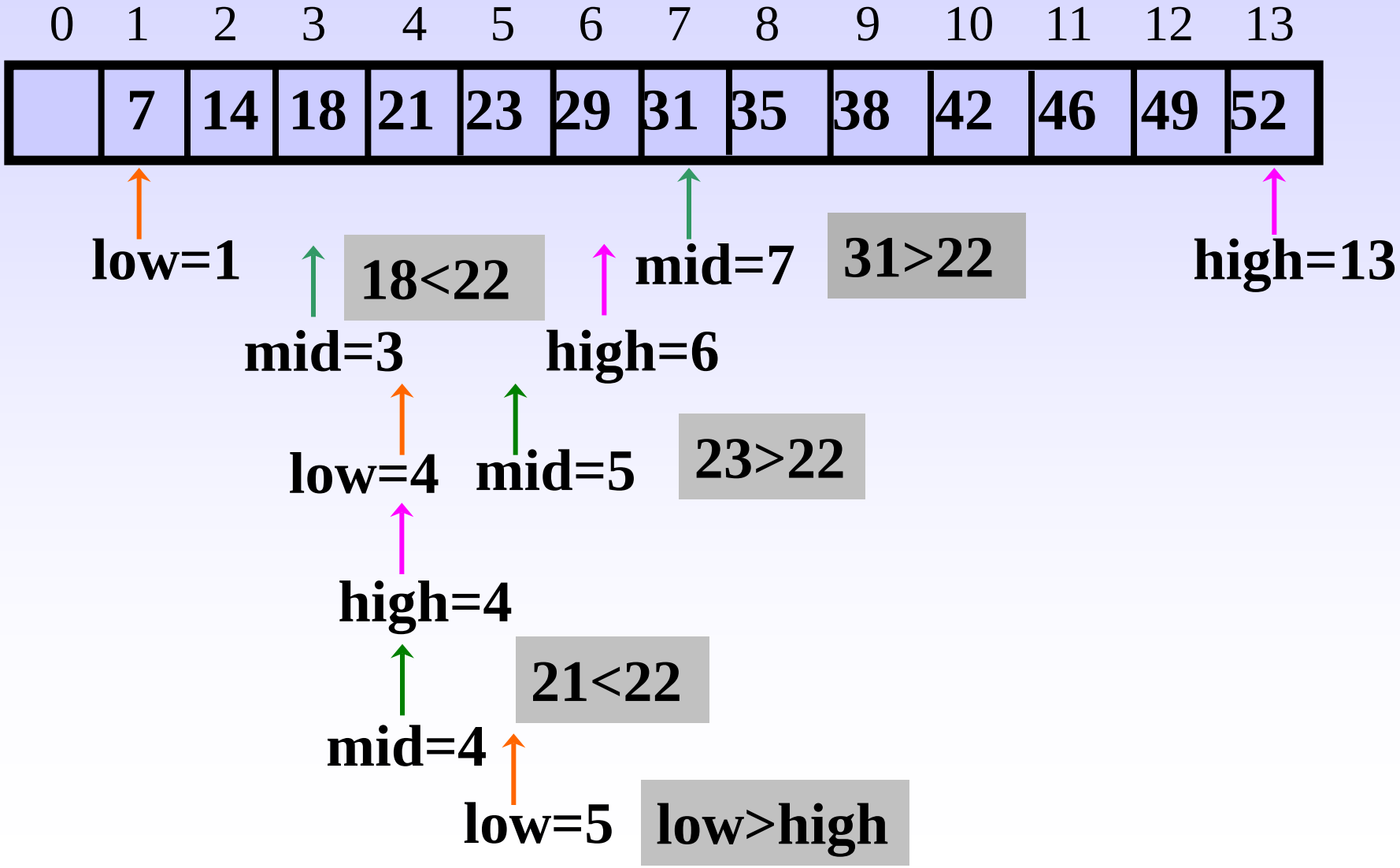
7.2 线性表的查找技术

例：查找值为14的记录的过程：



7.2 线性表的查找技术

例：查找值为22的记录的过程：



7.2 线性表的查找技术

折半查找——非递归算法

```
int BinSearch1(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    low=1; high=n;
    while (low<=high)
    {
       mid=(low+high)/2;
       if (k<r[mid])  high=mid-1;
       else if (k>r[mid])  low=mid+1;
              else return mid;
    }
    return 0;
}
```

7.2 线性表的查找技术

折半查找——递归算法

```
int BinSearch2(int r[ ], int low, int high, int k)
//数组r[1] ~ r[n]存放查找集合
{
    if (low>high) return 0;
    else {
       mid=(low+high)/2;
       if (k<r[mid])
           return BinSearch2(r, low, mid-1, k);
       else  if (k>r[mid])
                  return BinSearch2(r, mid+1, high, k);
              else return mid;
     }
 }
```

7.2 线性表的查找技术

折半查找判定树

判定树：折半查找的过程可以用二叉树来描述，树中的每个结点对应有序表中的一个记录，结点的值为该记录在表中的位置。通常称这个描述折半查找过程的二叉树为折半查找判定树，简称判定树。

7.2 线性表的查找技术

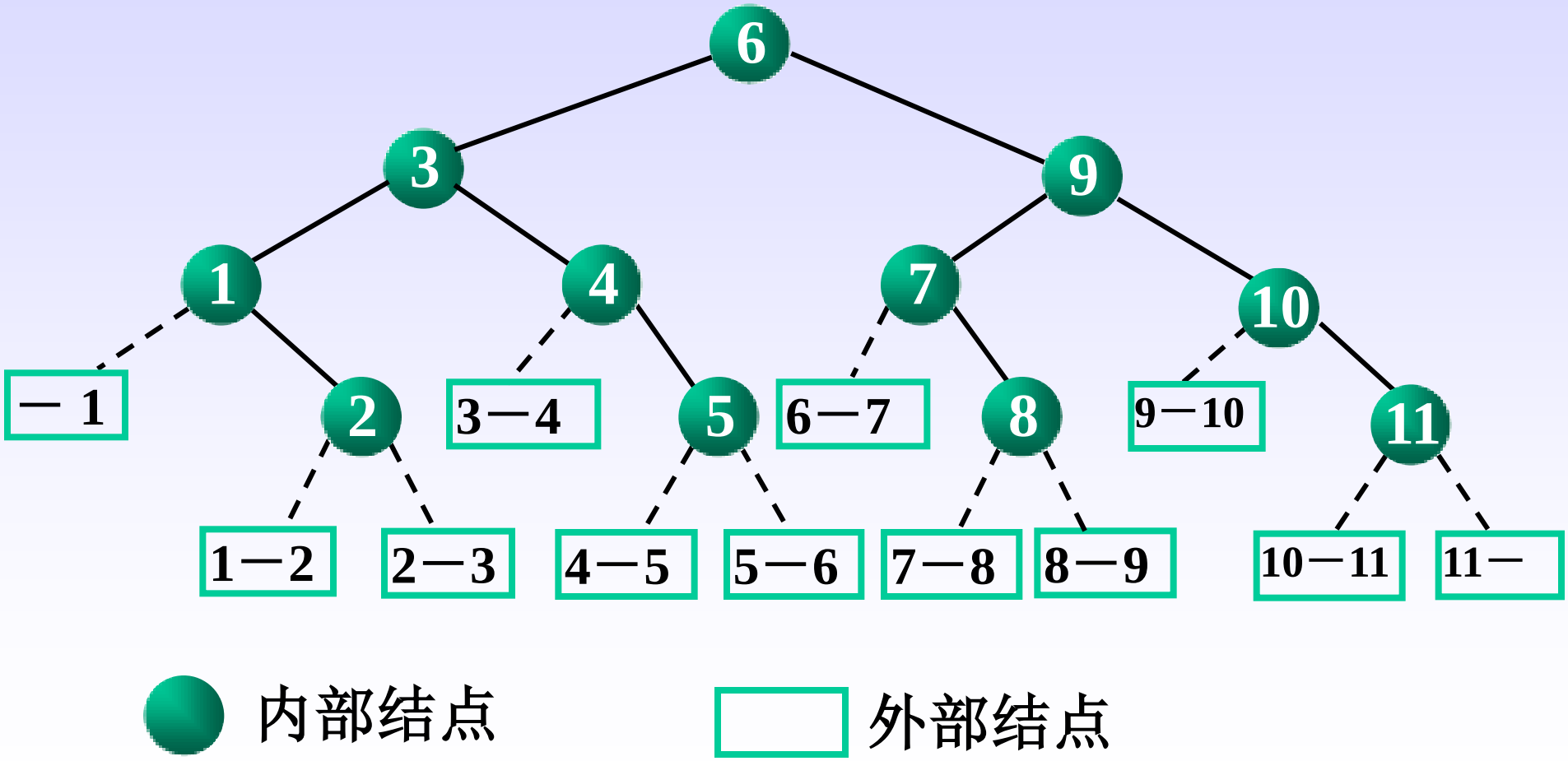
判定树的构造方法

⑴ 当n=0时，折半查找判定树为空；

⑵ 当n＞0时，折半查找判定树的根结点是有序表中序号为mid=(n+1)/2的记录，根结点的左子树是与有序表r[1] ~ r[mid−1]相对应的折半查找判定树，根结点的右子树是与r[mid+1] ~ r[n]相对应的折半查找判定树。

7.2 线性表的查找技术

判定树的构造方法



7.2 线性表的查找技术

折半查找性能分析

具有*n*个结点的折半查找判定树的深度为$\lfloor \log_2 n \rfloor + 1$。

查找成功：在表中查找任一记录的过程，即是折半查找判定树中从根结点到该记录结点的路径，和给定值的比较次数等于该记录结点在树中的层数。

查找不成功：查找失败的过程就是走了一条从根结点到外部结点的路径，和给定值进行的关键码的比较次数等于该路径上内部结点的个数。

性质5-4 具有*n*个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。

7.3 树表的查找技术

二叉排序树

二叉排序树（也称二叉查找树）：或者是一棵空的二叉树，或者是具有下列性质的二叉树：

(1) 若它的左子树不空，则左子树上所有结点的值均小于根结点的值；

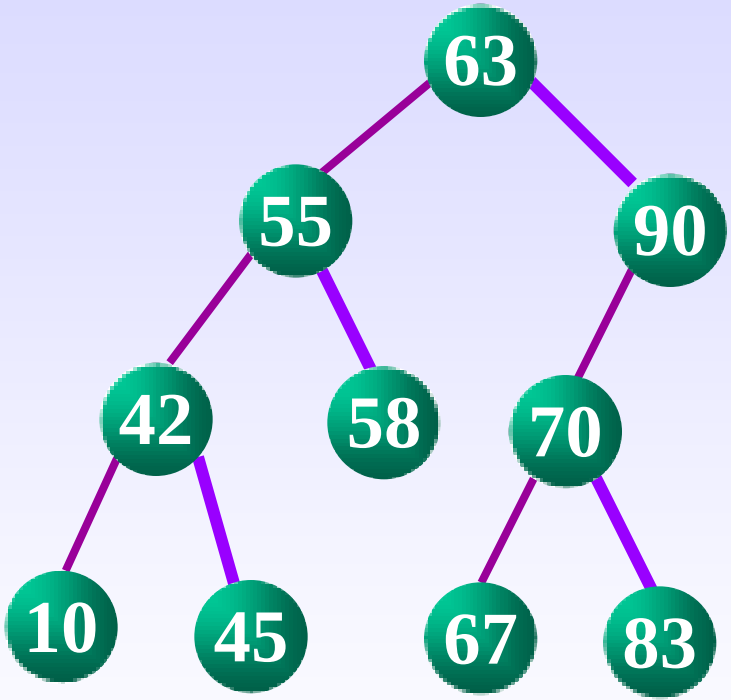
(2) 若它的右子树不空，则右子树上所有结点的值均大于根结点的值；

(3) 它的左右子树也都是二叉排序树。

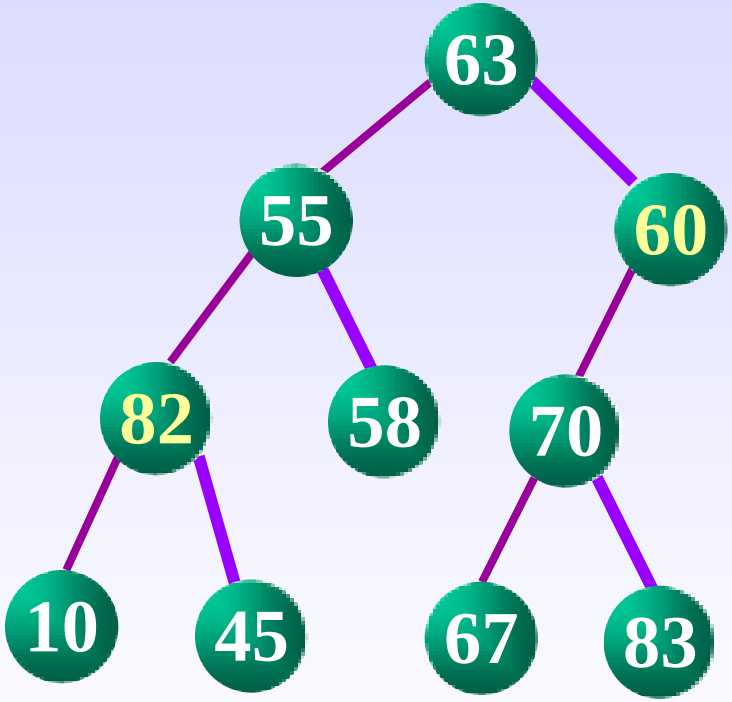
二叉排序树的定义采用的是递归方法。

7.3 树表的查找技术

二叉排序树



二叉排序树



非二叉排序树

中序遍历二叉排序树可以得到一个按关键码有序的序列

7.3 树表的查找技术

二叉排序树的存储结构

以二叉链表形式存储，类声明如下：

```
class BiSortTree
{
  public:
     BiSortTree(int a[ ], int n);
     ~ BiSortTree( );
     void InsertBST(BiNode<int> *root , BiNode<int> *s);
     void DeleteBST(BiNode<int> *p, BiNode<int> *f );
     BiNode<int> *SearchBST(BiNode<int> *root, int k);
  private:
      BiNode<int> *root;
};
```

7.3 树表的查找技术

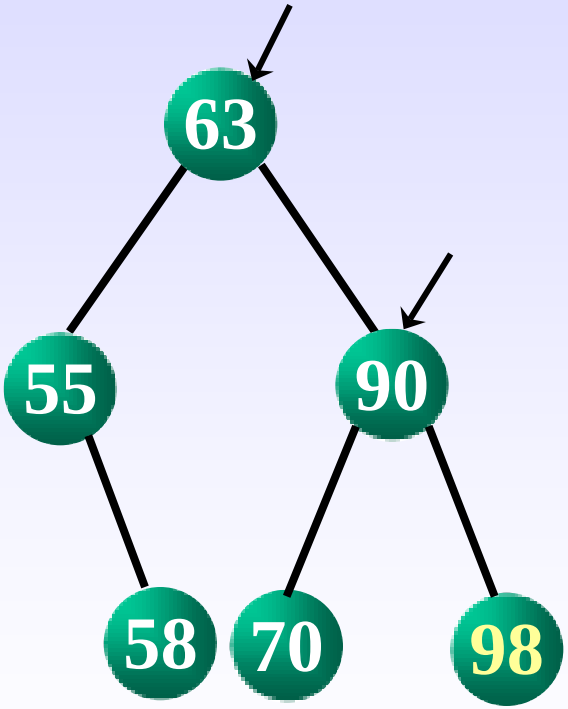
二叉排序树的插入

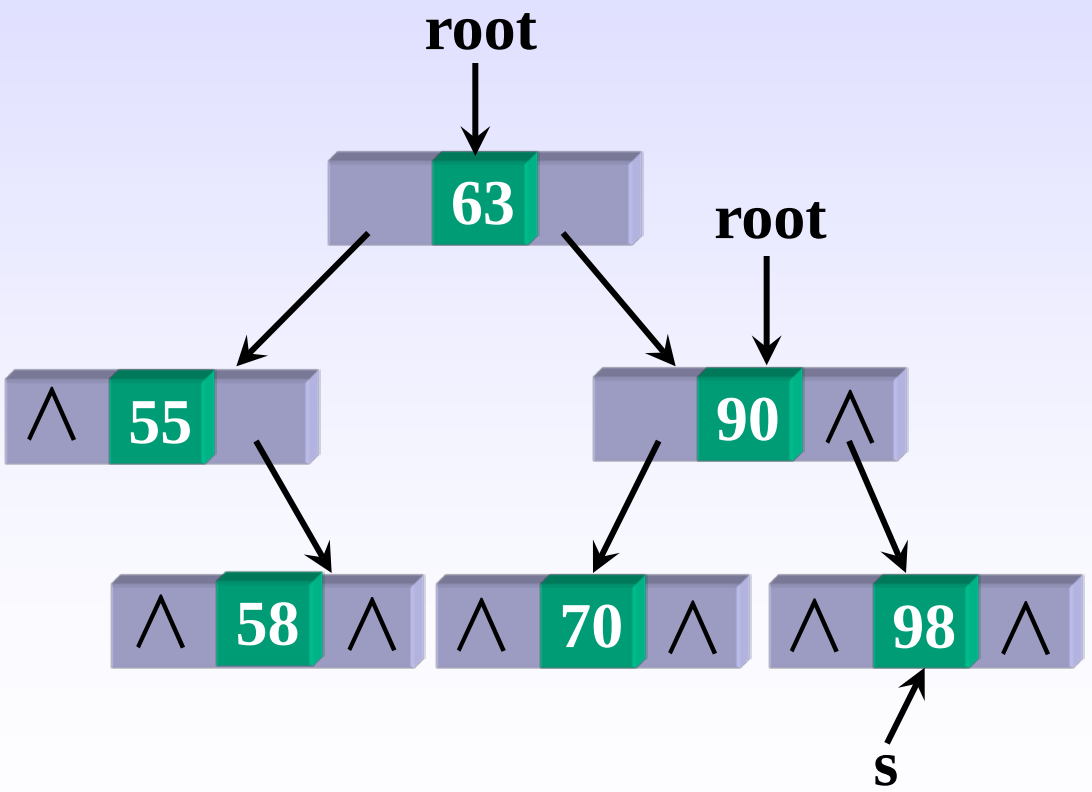
```
void InsertBST(BiNode<int> *root , BiNode<int> *s);
```

分析：若二叉排序树为空树，则新插入的结点为新的根结点；否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到。

7.3 树表的查找技术

例：插入值为98的结点





7.3 树表的查找技术

二叉排序树的插入算法

```
void BiSortTree::InsertBST(BiNode<int> *root, BiNode<int> *s)
{
   if (root==NULL)
      root=s;
   else  if (s->data<root->data)
            InsertBST(root->lchild, s);
         else   InsertBST(root->rchild, s);
}
```

7.3 树表的查找技术

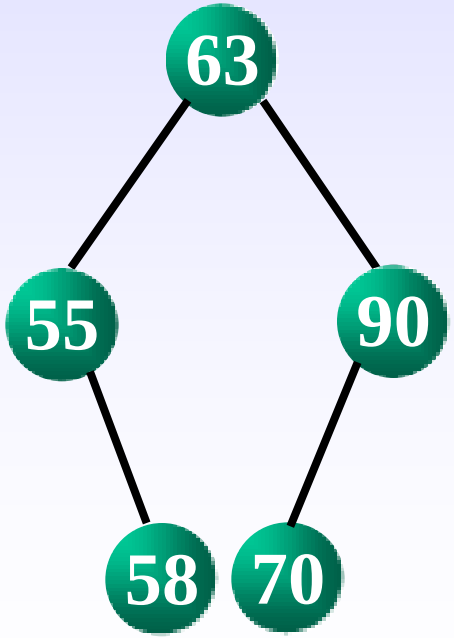
二叉排序树的构造

从空的二叉排序树开始，依次插入一个个结点 。

例：关键码集合为

{63，90，70，55，58}，

二叉排序树的构造过程为：



7.3 树表的查找技术

二叉排序树的构造算法

```
BiSortTree::BiSortTree(int r[ ], int n)
{
    for (i=0; i<n; i++)
    {
       s=new BiNode<int>;
       s->data=r[i];
       s->lchild=s->rchild=NULL;
       InsertBST(root, s);
    }
}
```

7.3 树表的查找技术

小 结：

- 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列；
- 每次插入的新结点都是二叉排序树上新的叶子结点；
- 找到插入位置后，不必移动其它结点，仅需修改某个结点的指针；
- 在左子树/右子树的查找过程与在整棵树上查找过程相同；
- 新插入的结点没有破坏原有结点之间的关系。

7.3 树表的查找技术

二叉排序树的删除

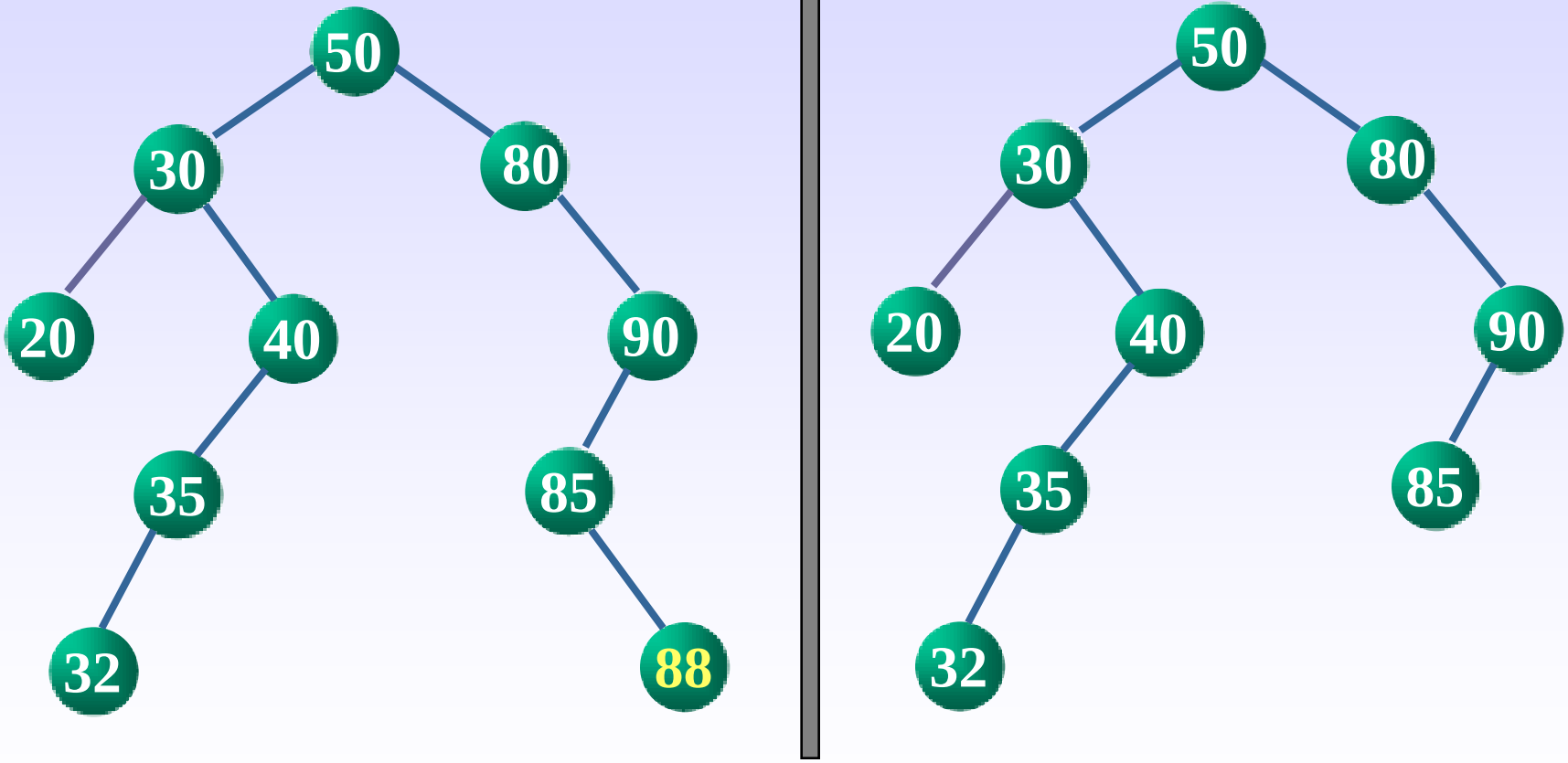
在二叉排序树上删除某个结点之后，仍然保持二叉排序树的特性。

分三种情况讨论：

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

7.3 树表的查找技术

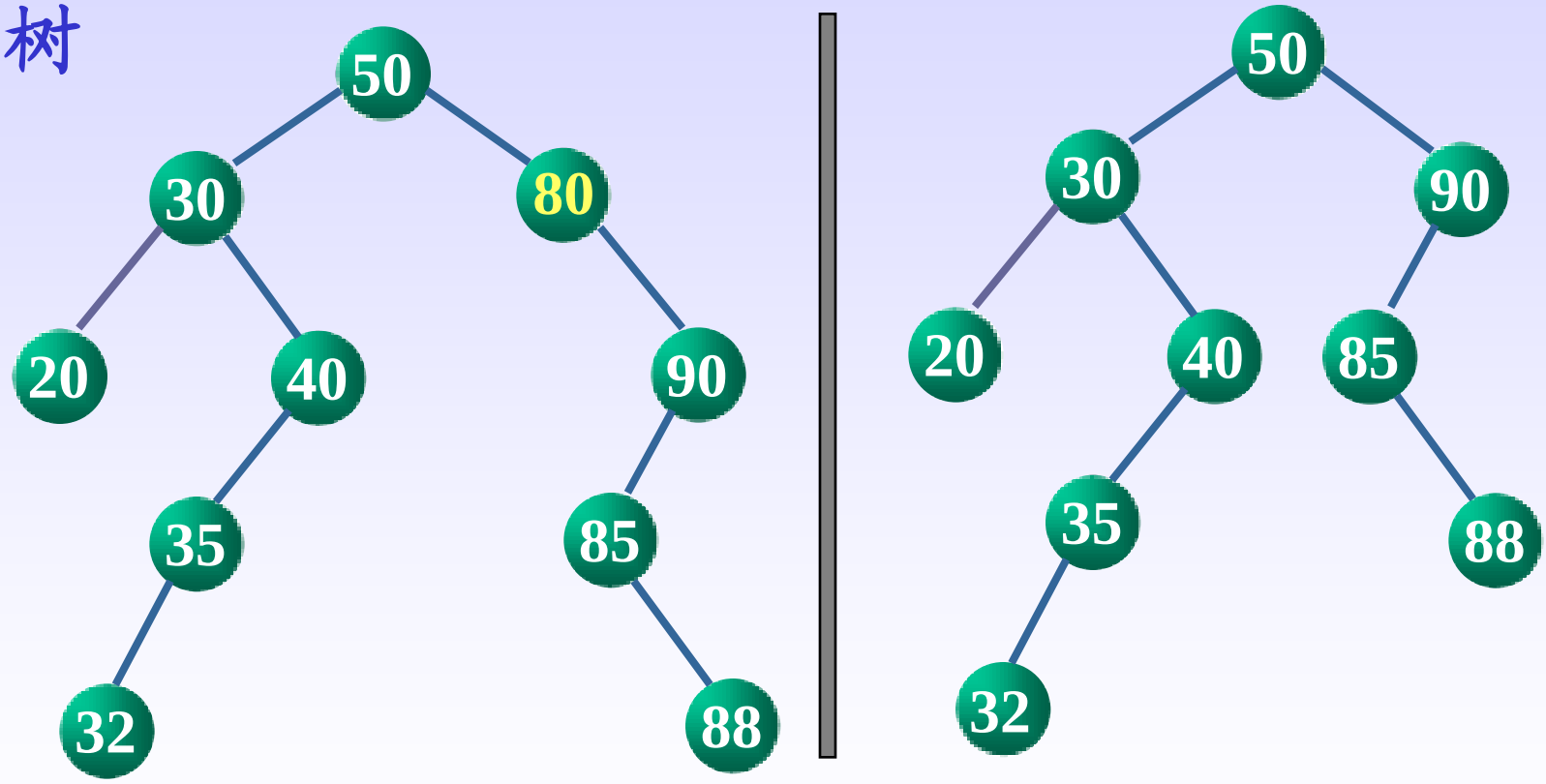
情况1——被删除的结点是叶子结点



操作： 将双亲结点中相应指针域的值改为空。

7.3 树表的查找技术

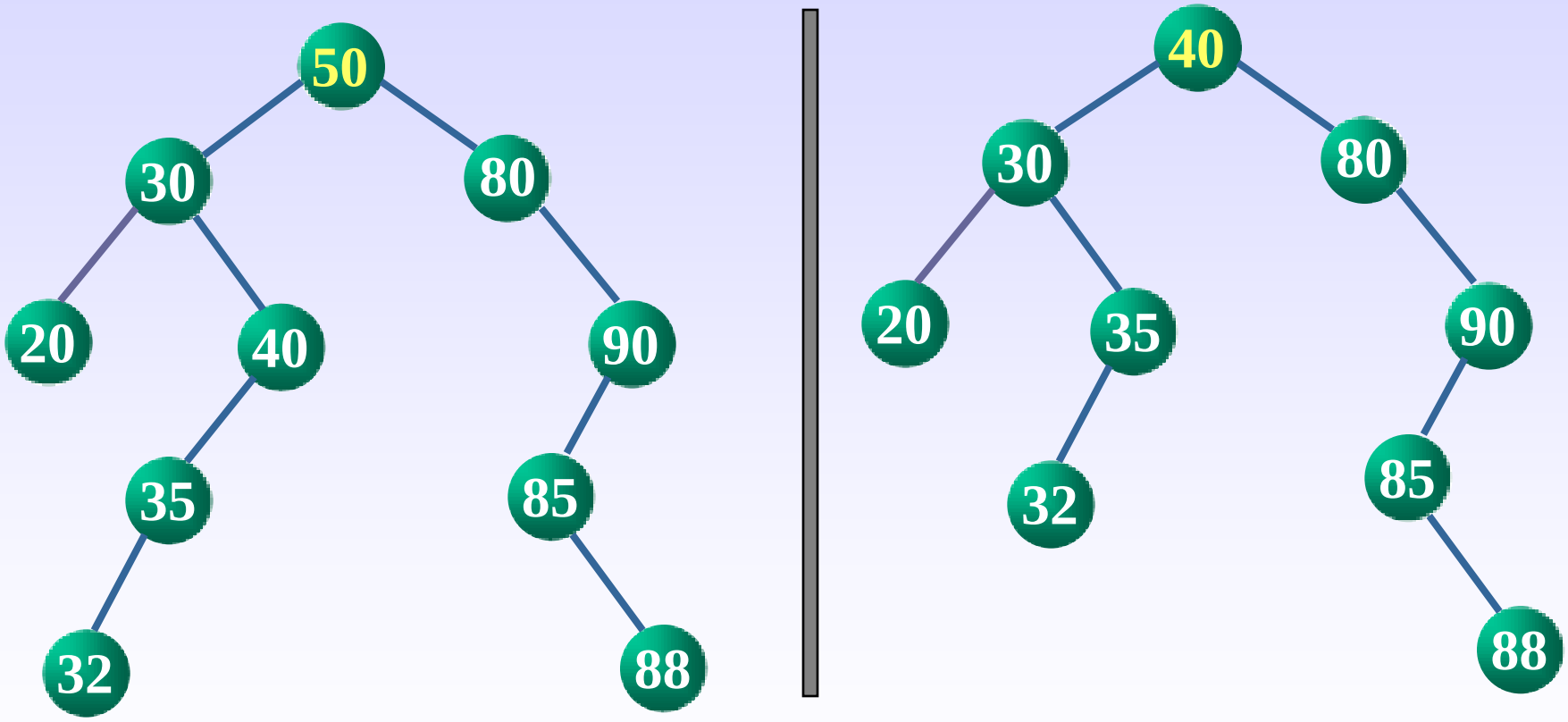
情况2——被删除的结点只有左子树或者只有右子树



操作： 将双亲结点的相应指针域的值指向被删除结点的左子树（或右子树）。

7.3 树表的查找技术

情况3——被删除的结点既有左子树也有右子树

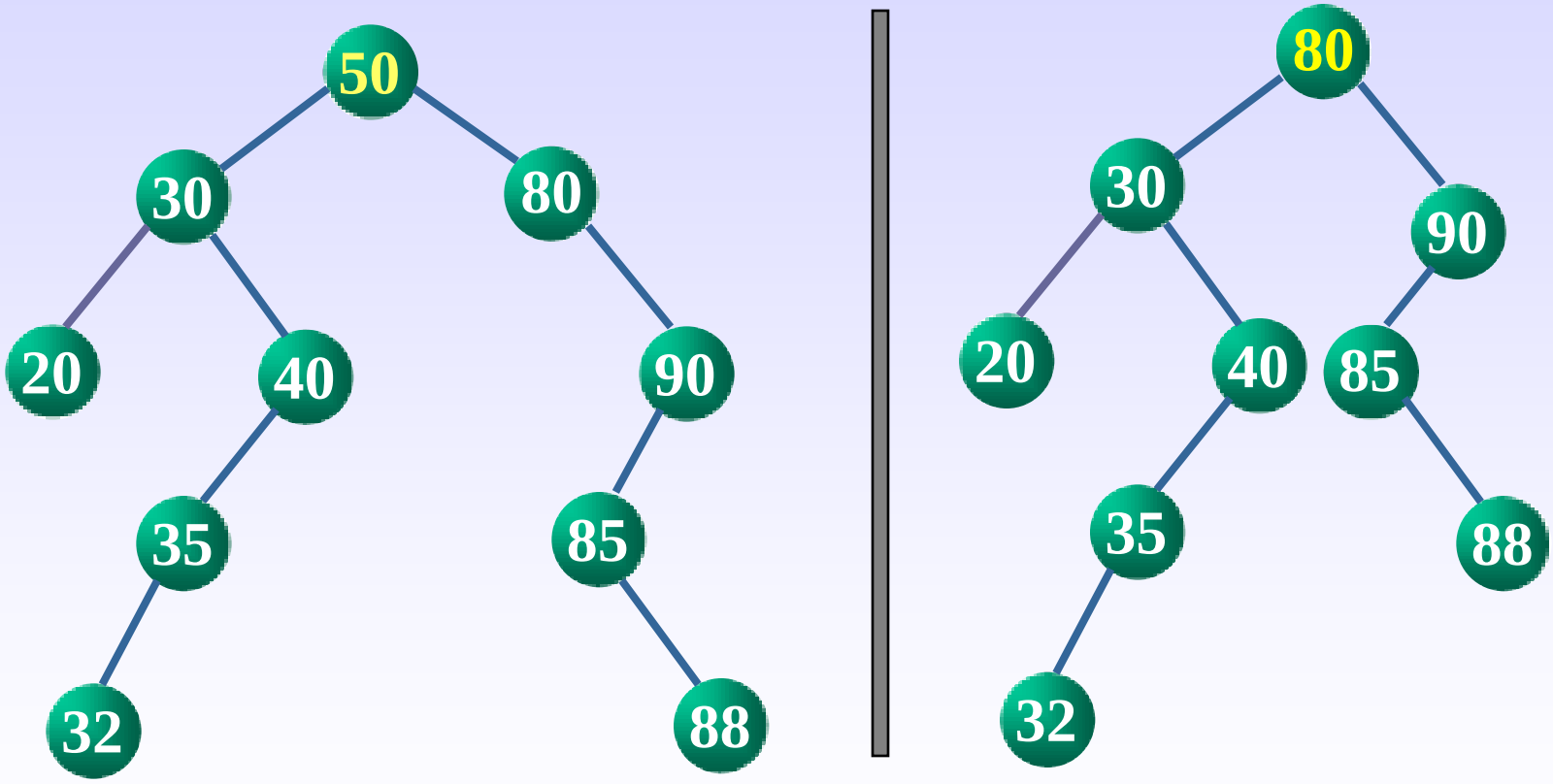


操作：以其前驱（左子树中的最大值）替代之，然后再删除该前驱结点。

无右子树

7.3 树表的查找技术

情况3——被删除的结点既有左子树也有右子树



操作：以其后继（右子树中的最小值）替代之，然后再删除该后继结点。

无左子树

7.3 树表的查找技术

二叉排序树的查找

在二叉排序树中查找给定值k的过程是：

⑴ 若root是空树，则查找失败；

⑵ 若k＝root->data，则查找成功；否则

⑶ 若k＜root->data，则在root的左子树上查找；否则

⑷ 在root的右子树上查找。

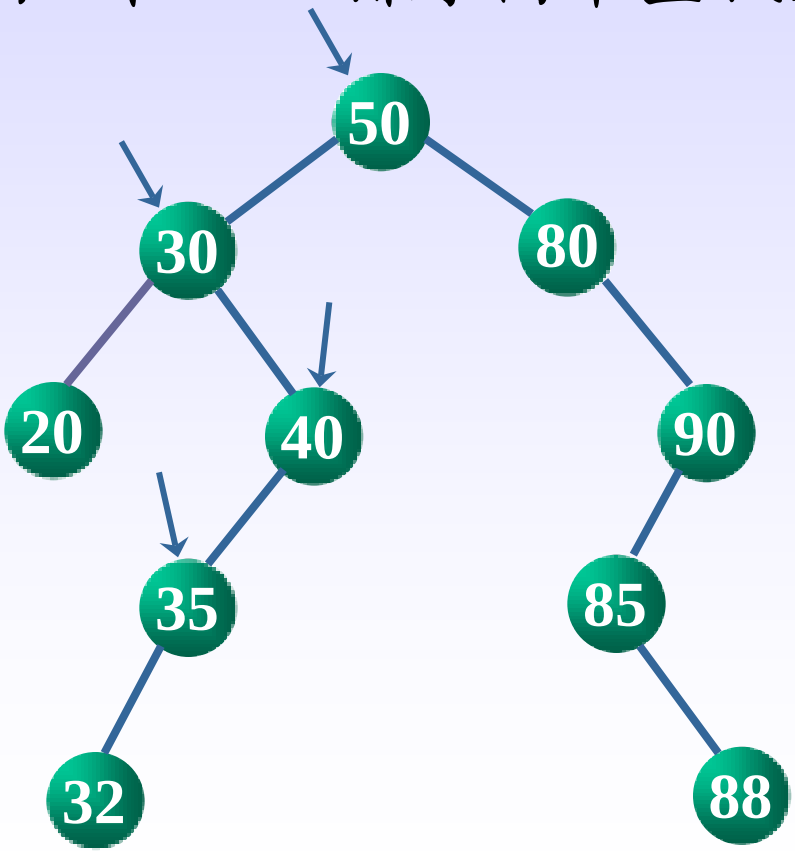
上述过程一直持续到k被找到或者待查找的子树为空，如果待查找的子树为空，则查找失败。

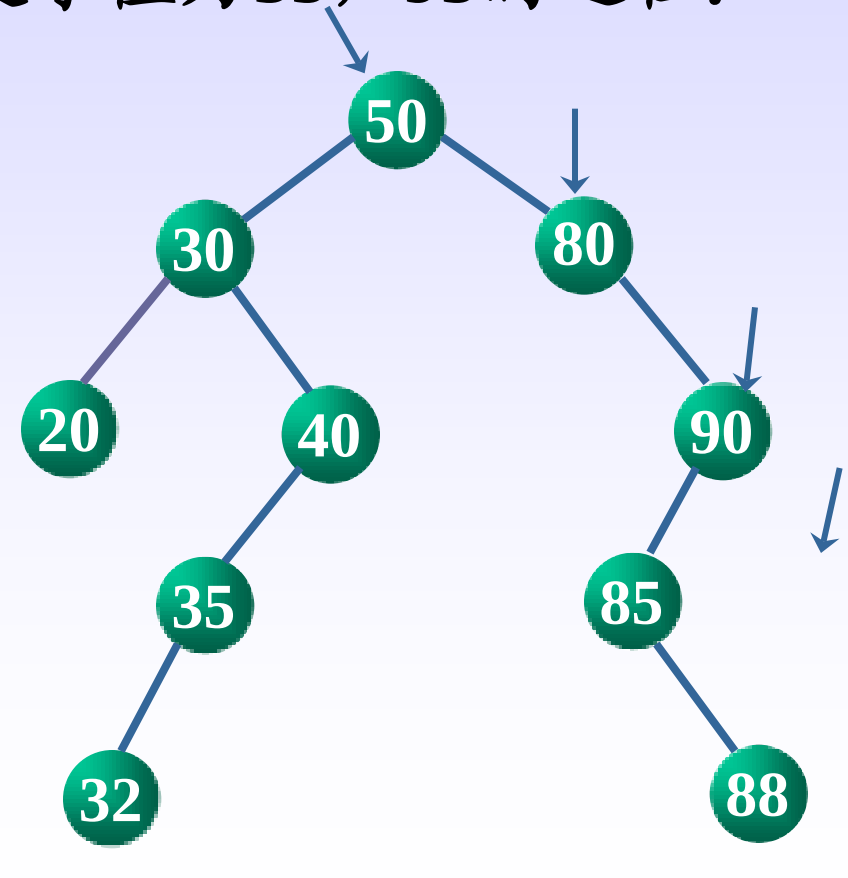
二叉排序树的查找效率在于只需查找二个子树之一。

7.3 树表的查找技术

二叉排序树的查找

例：在二叉排序树中查找关键字值为35，95的过程：





7.3 树表的查找技术

二叉排序树的查找

```
BiNode *BiSortTree::SearchBST(BiNode<int> *root, int k)
{
   if (root==NULL)
      return NULL;
   else if (root->data==k)
            return root;
         else if (k<root->data)
                  return SearchBST(root->lchild, k);
               else return SearchBST(root->rchild, k);
}
```

7.3 树表的查找技术

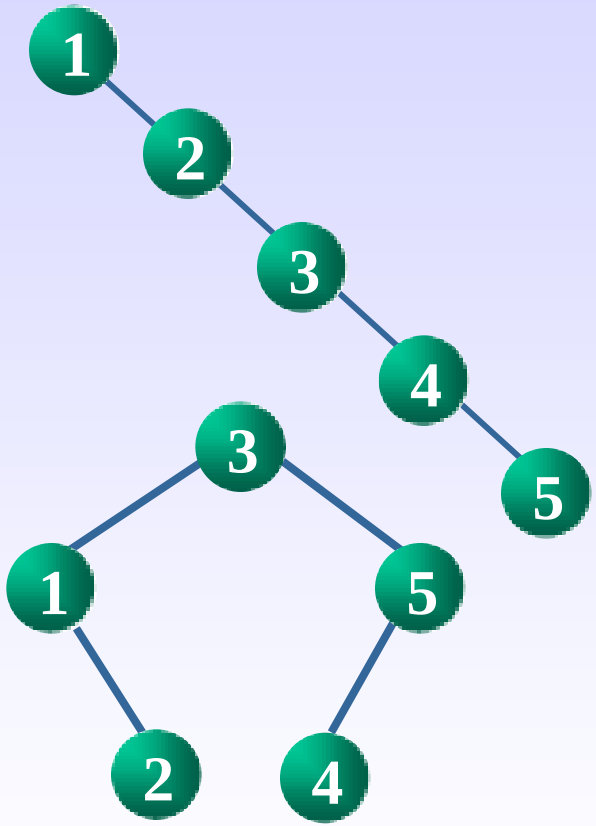
二叉排序树的查找性能分析

由序列{1, 2, 3, 4, 5}得到二叉排序树：

ASL =（1+2+3+4+5）/ 5= 3

由序列{3, 1, 2, 5, 4}得到二叉排序树：

ASL =（1+2+3+2+3）/ 5 = 2.2



二叉排序树的查找性能取决于二叉排序树的形状，在$O(\log_2 n)$和$O(n)$之间。

7.3 树表的查找技术

平衡二叉树

平衡二叉树：或者是一棵空的二叉排序树，或者是具有下列性质的二叉排序树：

(1) 根结点的左子树和右子树的深度最多相差1;

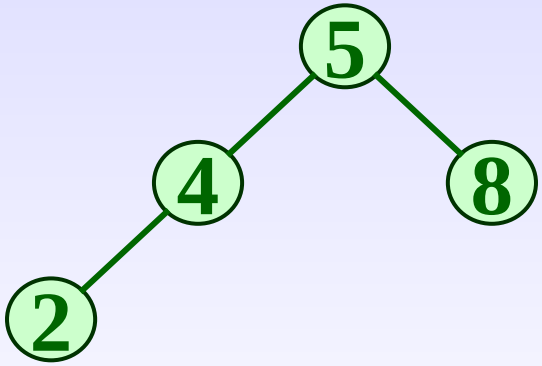
(2) 根结点的左子树和右子树也都是平衡二叉树。

平衡因子：结点的平衡因子是该结点的左子树的深度与右子树的深度之差。

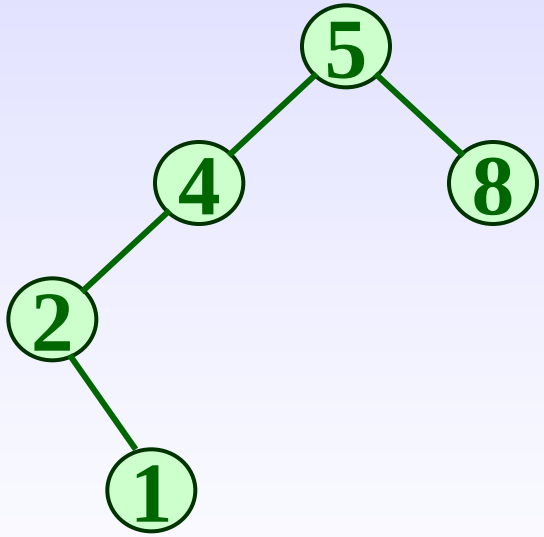
7.3 树表的查找技术

平衡二叉树

结点的平衡因子＝H_L-H_R



是平衡树



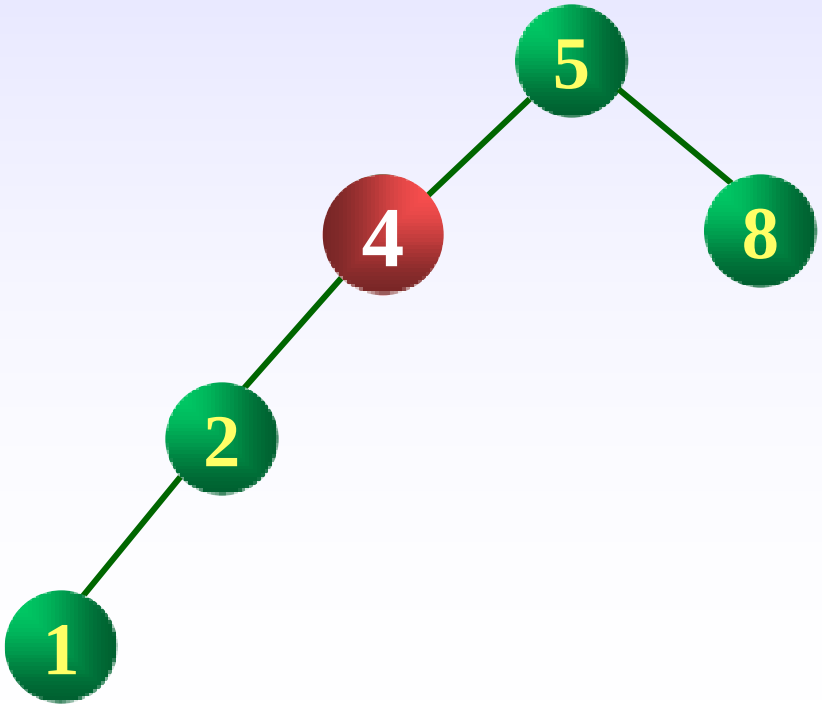
非平衡树

在平衡树中，结点的平衡因子可以是1，0，-1。

7.3 树表的查找技术

平衡二叉树

最小不平衡子树：在平衡二叉树的构造过程中，以距离插入结点最近的、且平衡因子的绝对值大于1的结点为根的子树。



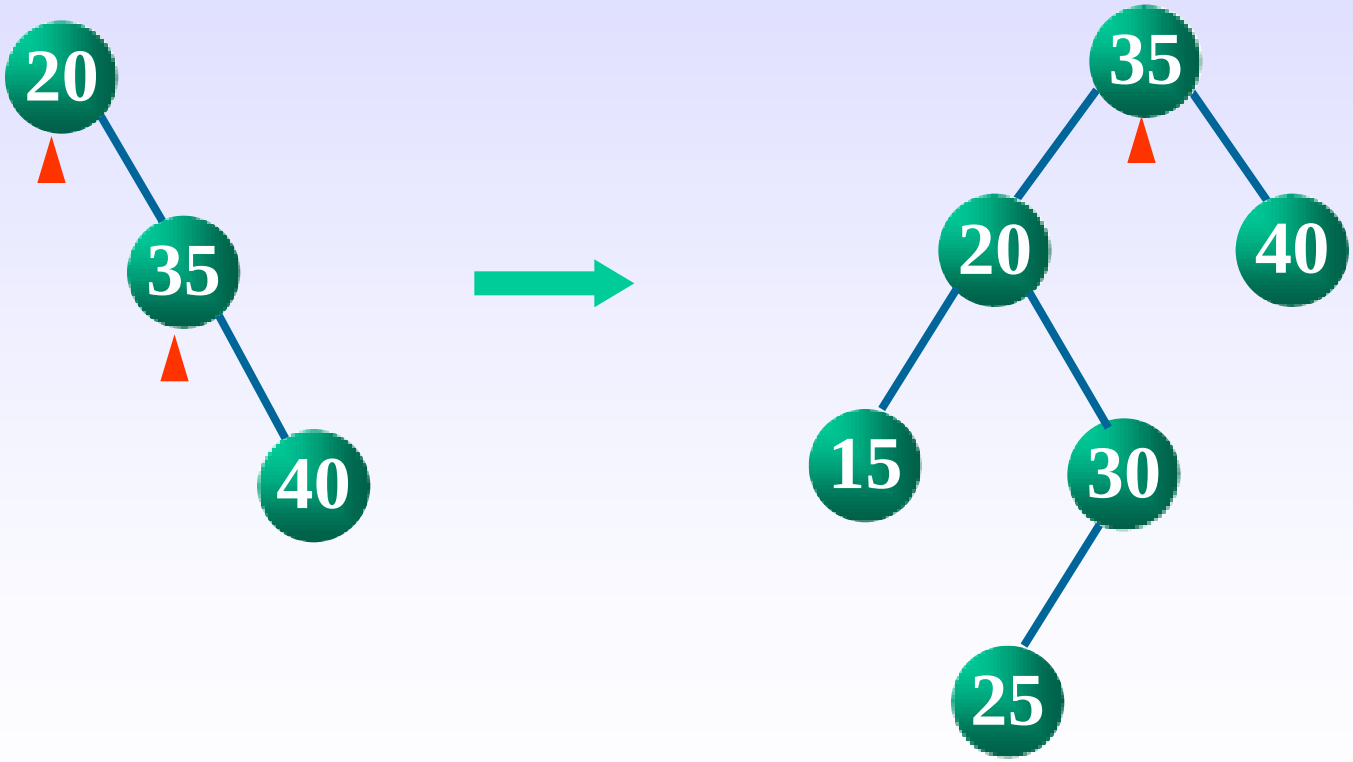
7.3 树表的查找技术

平衡二叉树

基本思想：在构造二叉排序树的过程中，每插入一个结点时，首先检查是否因插入而破坏了树的平衡性，若是，则找出最小不平衡子树，在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。

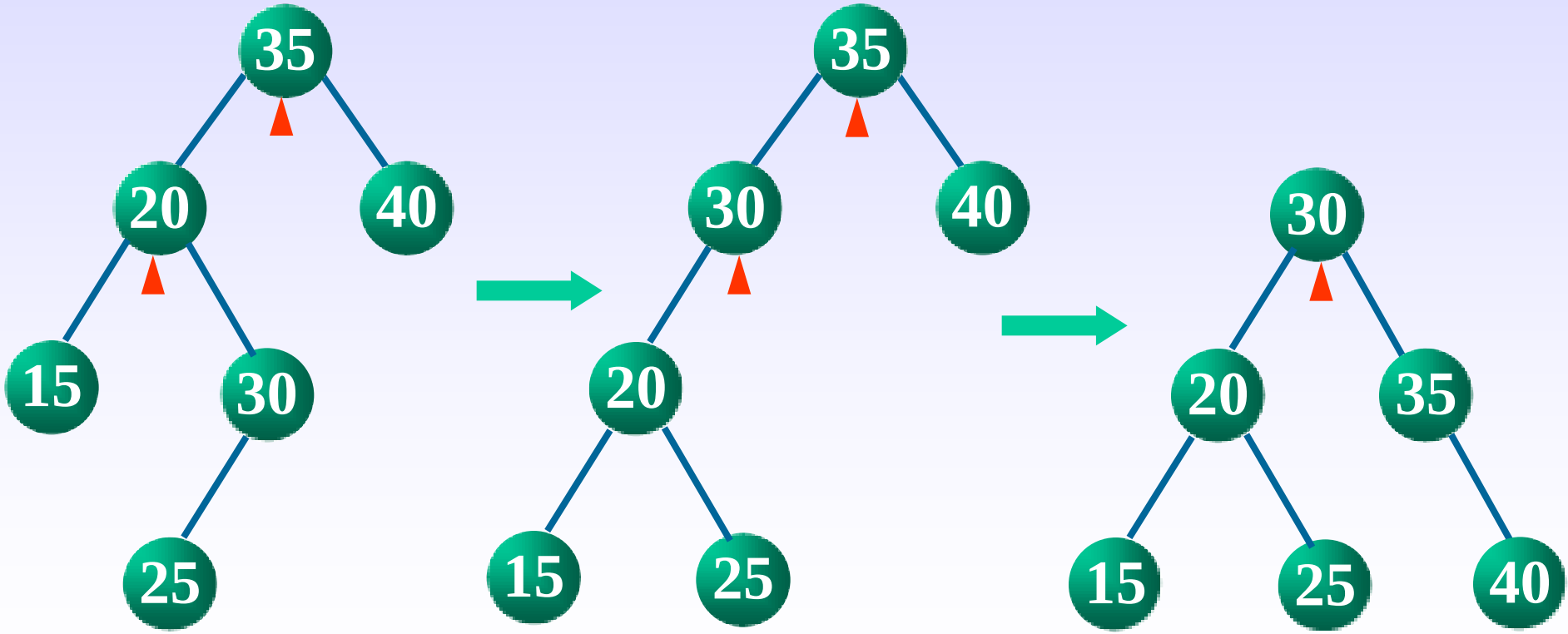
7.3 树表的查找技术

例：设序列{20，35，40，15，30，25}，构造平衡二叉树。



7.3 树表的查找技术

例：设序列{20，35，40，15，30，25}，构造平衡树。



7.3 树表的查找技术

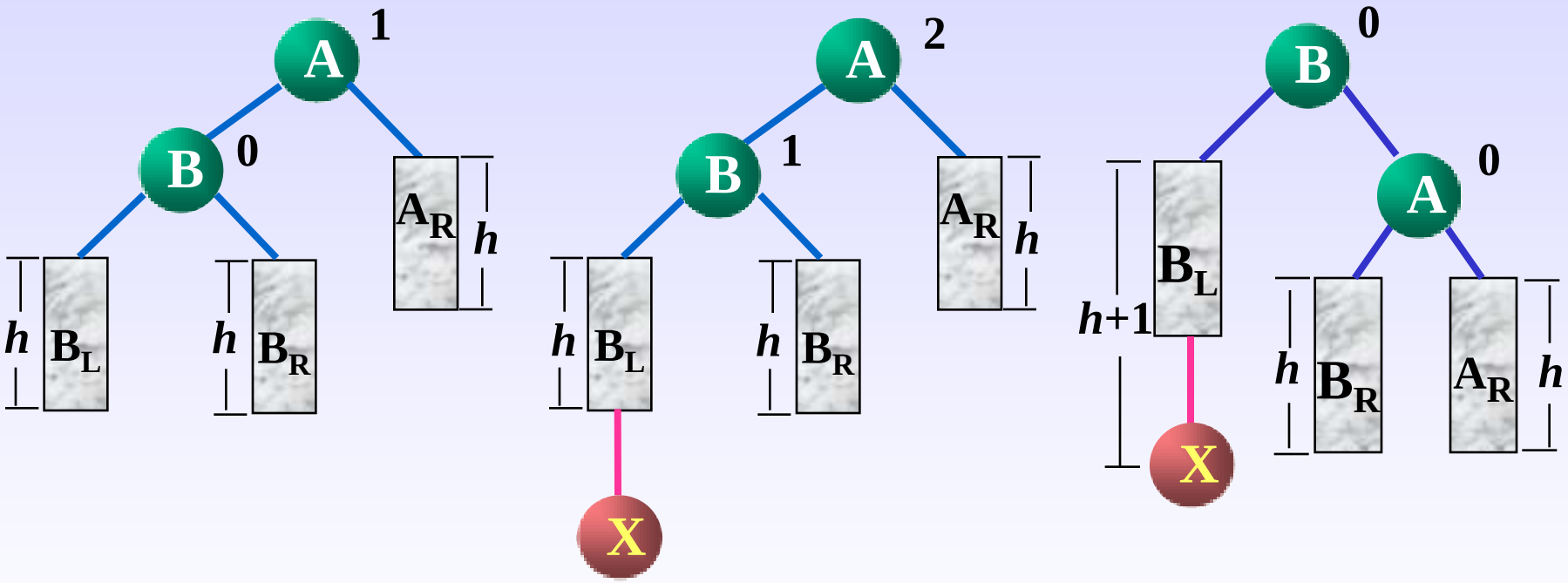
平衡二叉树

设结点A为最小不平衡子树的根结点，对该子树进行平衡调整归纳起来有以下四种情况：

1. LL型
2. RR型
3. LR型
4. RL型

7.3 树表的查找技术

平衡二叉树——LL型

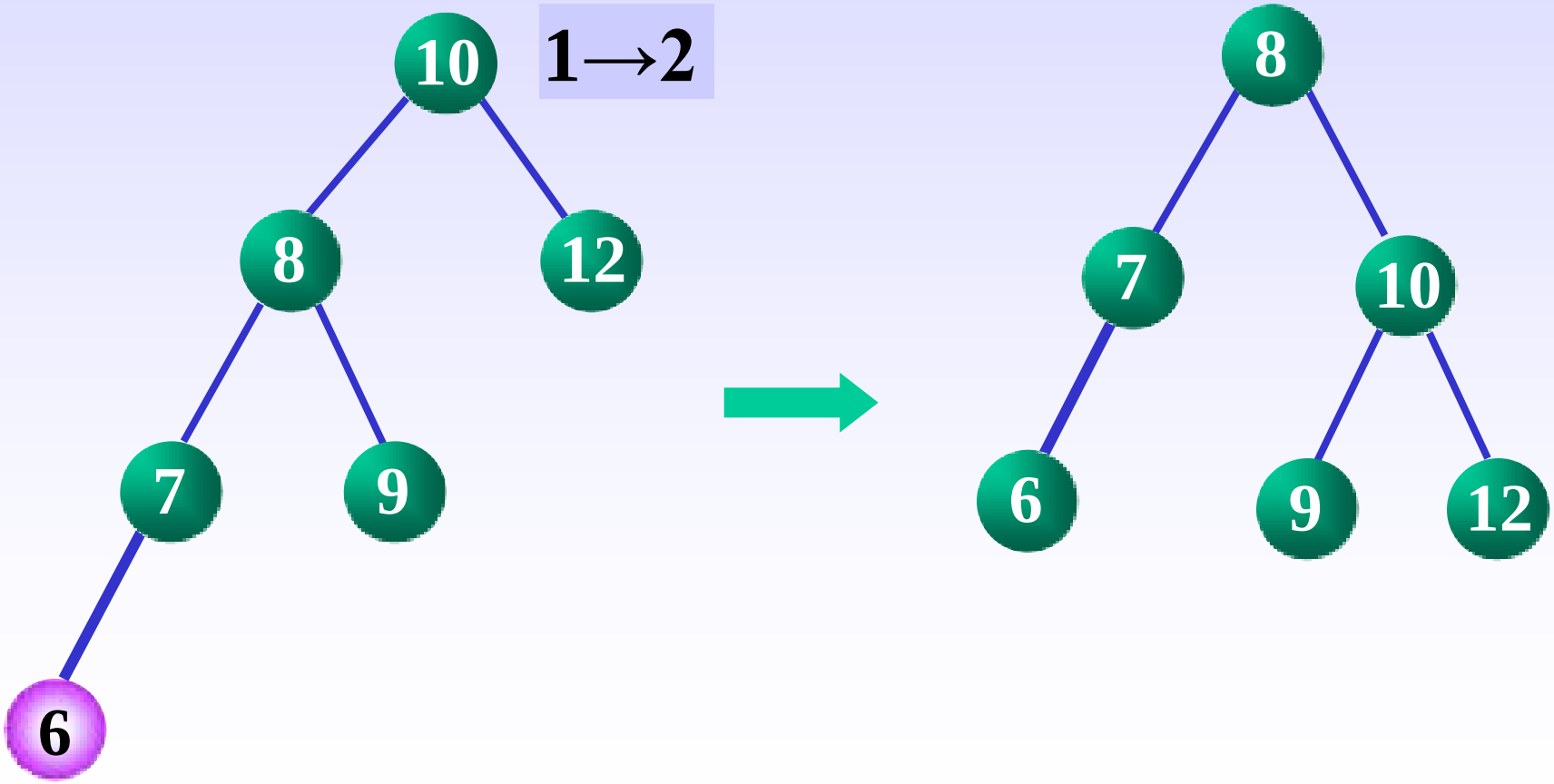


插入前　　插入后，调整前　　调整后

旋转：扁担原理；冲突：旋转优先

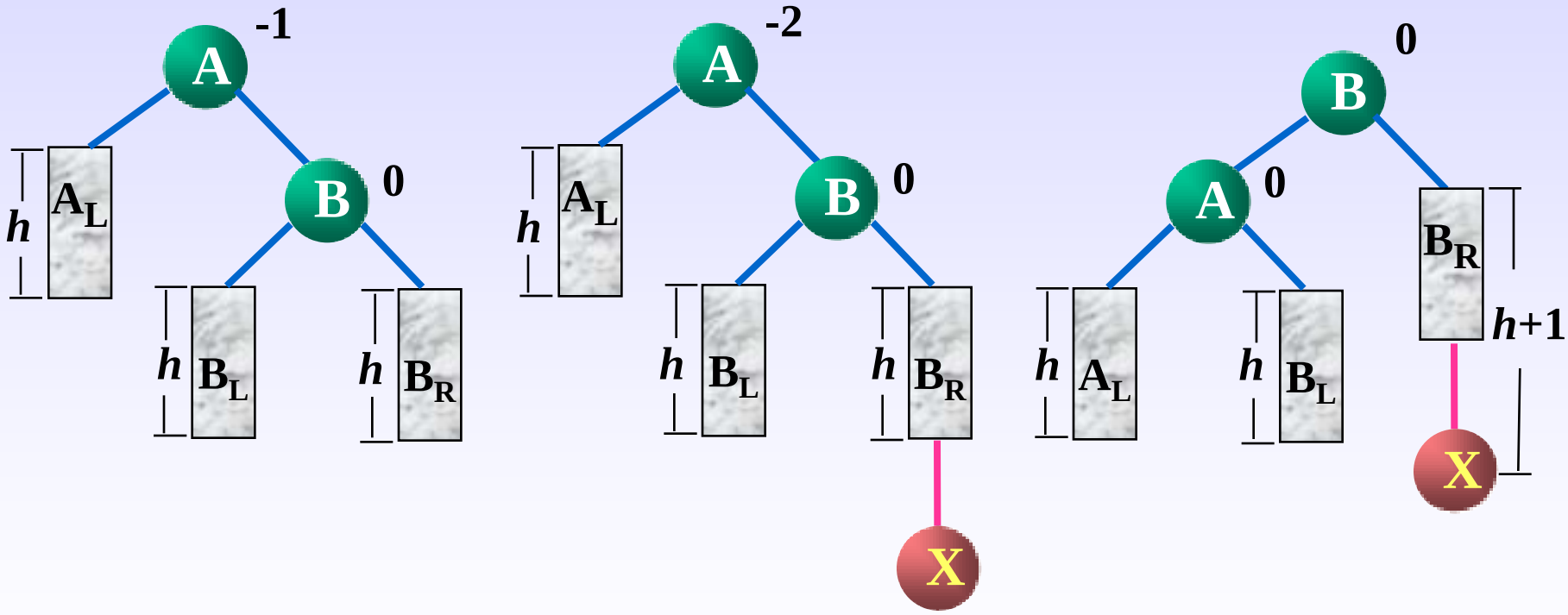
7.3 树表的查找技术

例:LL型



7.3 树表的查找技术

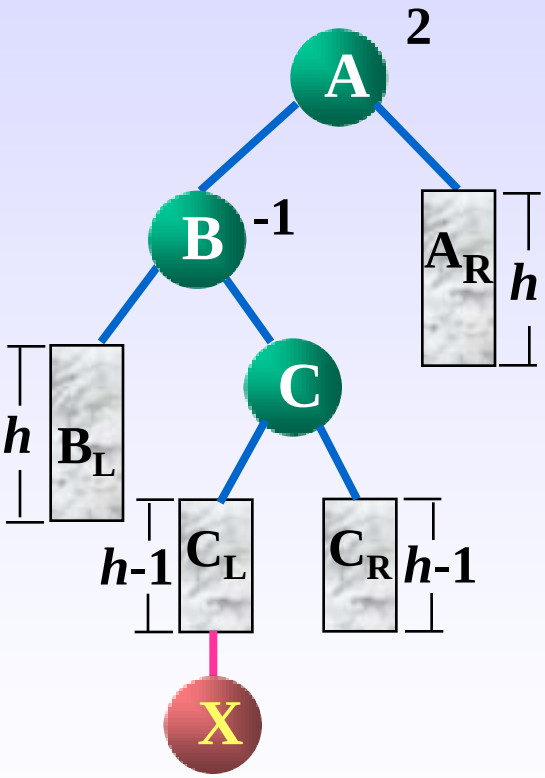
平衡二叉树——RR型



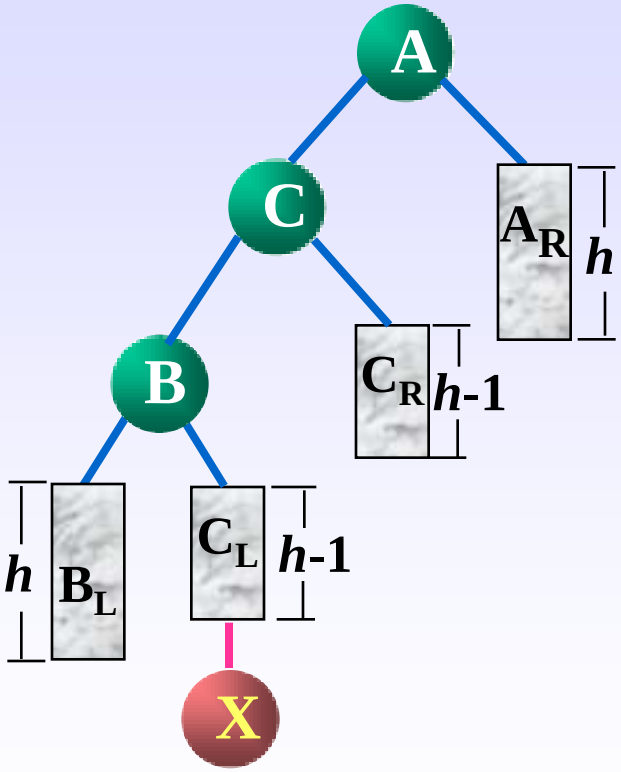
插入前　　插入后，调整前　　调整后

7.3 树表的查找技术

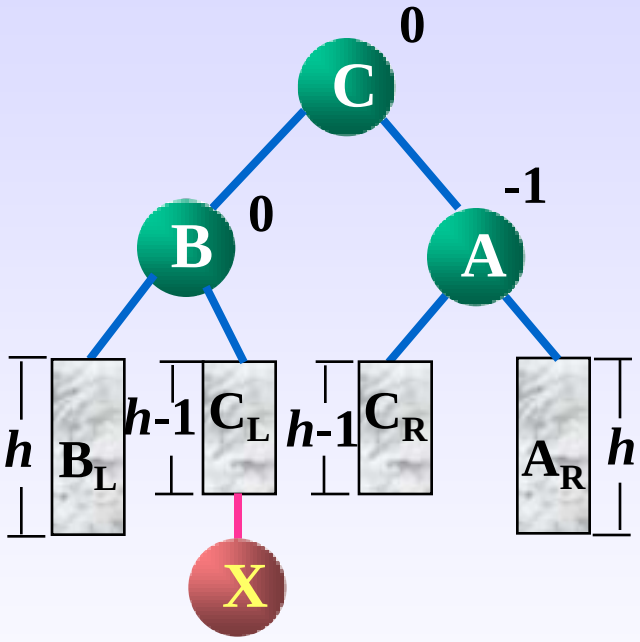
平衡二叉树——LR型



插入后，调整前



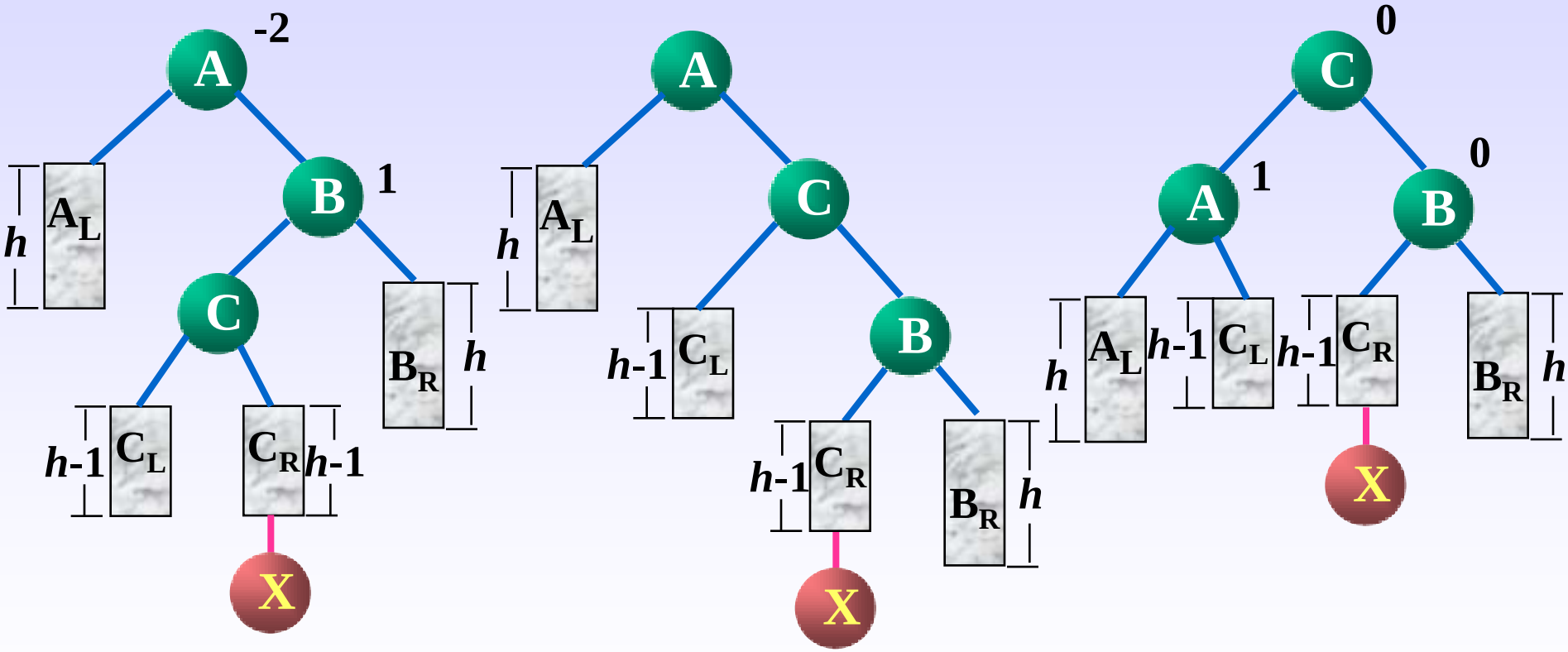
先顺时针旋转



再逆时针旋转

7.3 树表的查找技术

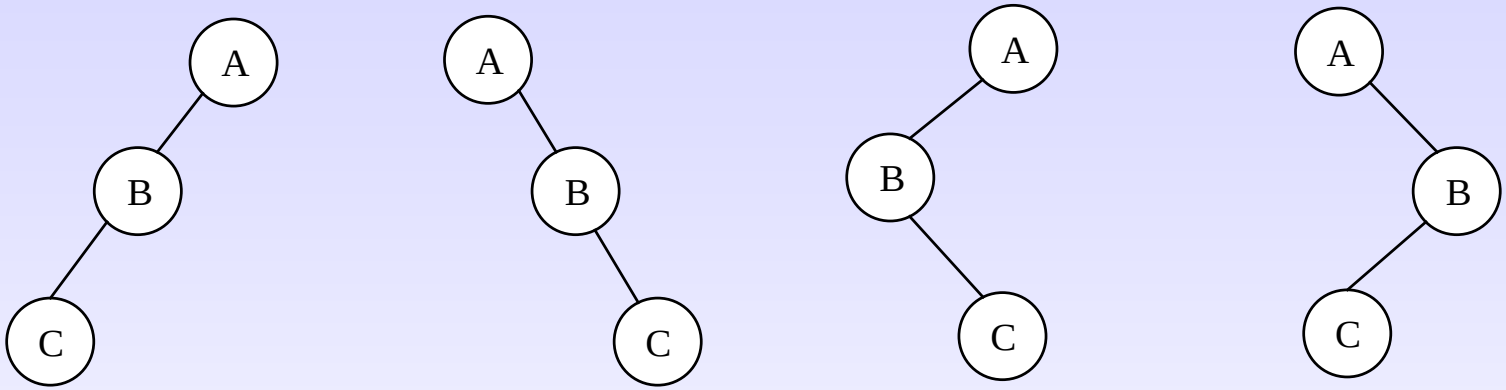
平衡二叉树——RL型



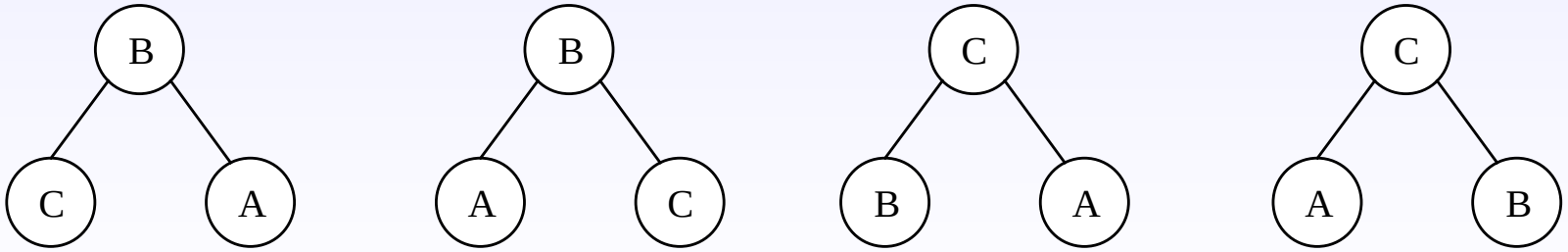
插入后，调整前　　先顺时针旋转　　再逆时针旋转

7.3 树表的查找技术

LL型 **RR型** **LR型** **RL型**



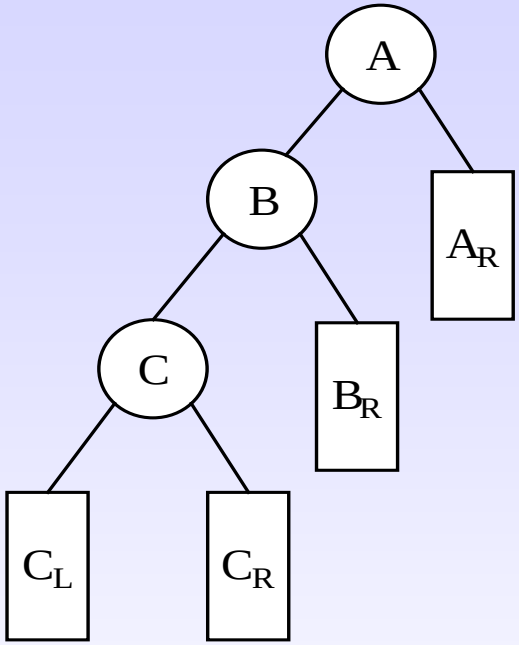
调整后：



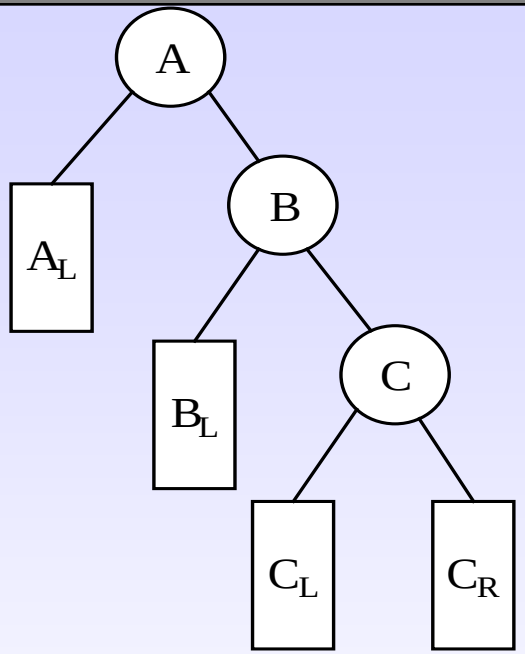
顺时针旋转　　逆时针旋转　　先逆时针旋转
再顺时针旋转　　先顺时针旋转
再逆时针旋转

7.3 树表的查找技术

LL型

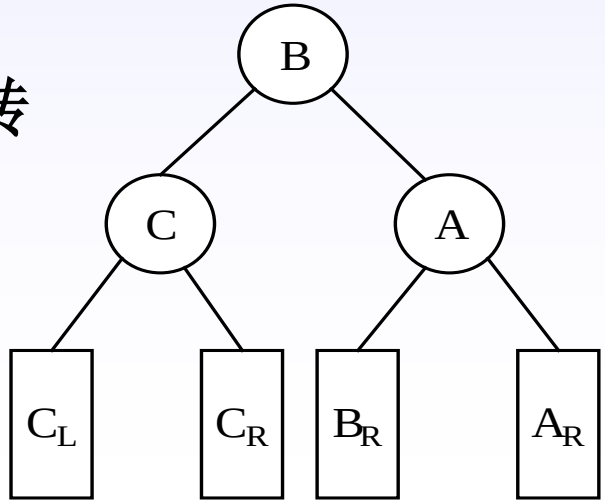


RR型

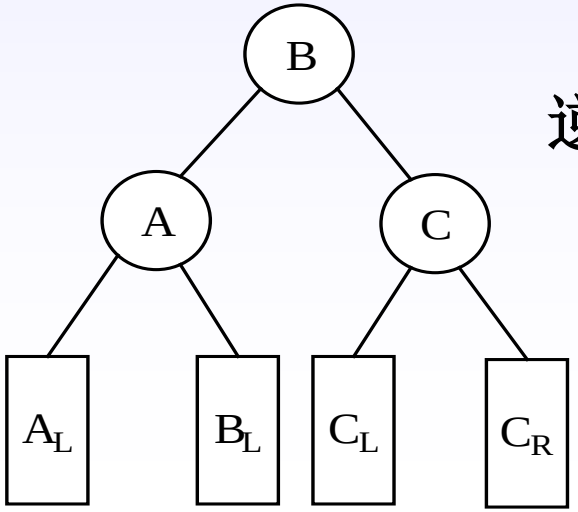


调整

顺时针旋转

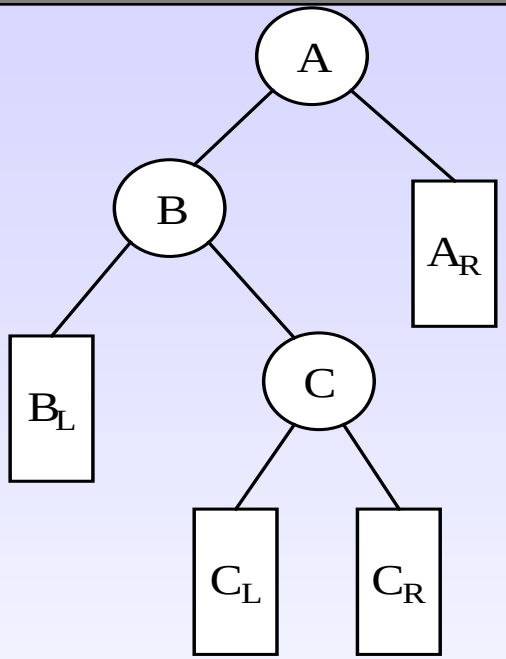


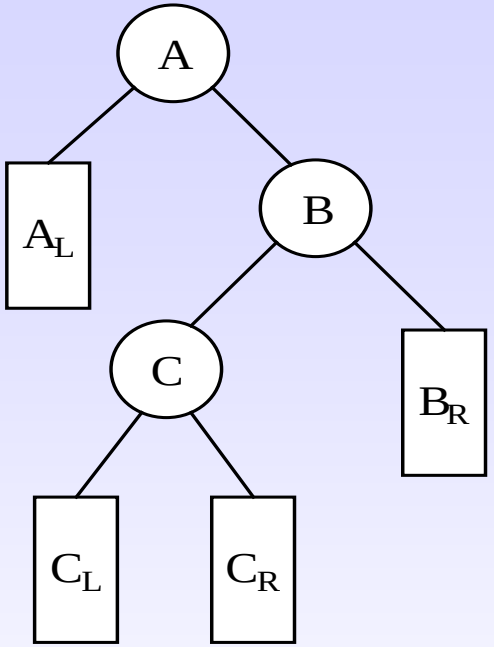
逆时针旋转



7.3 树表的查找技术

LR型

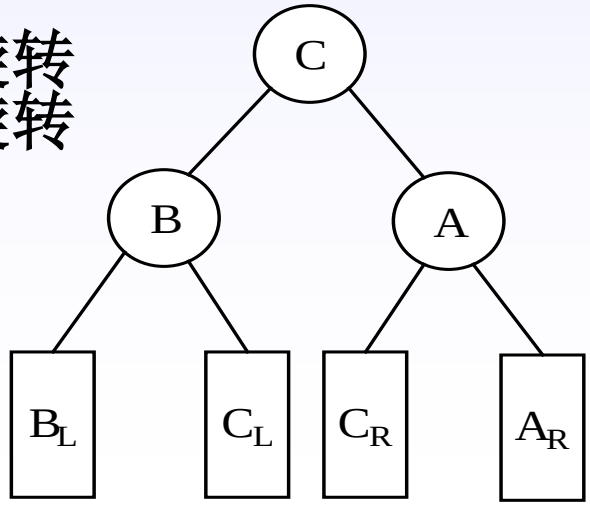




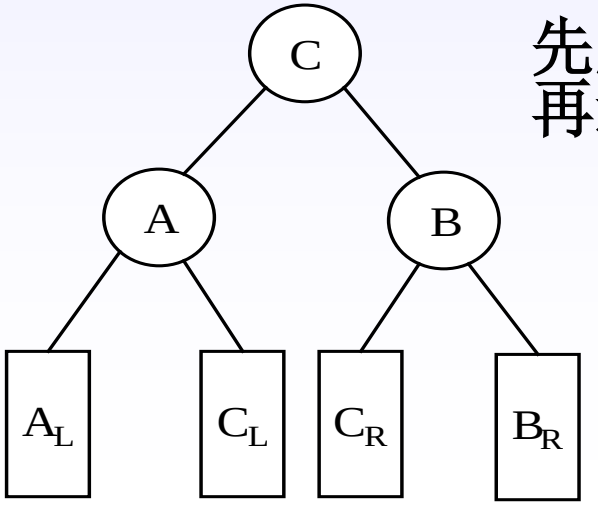
RL型

调整

先逆时针旋转
再顺时针旋转

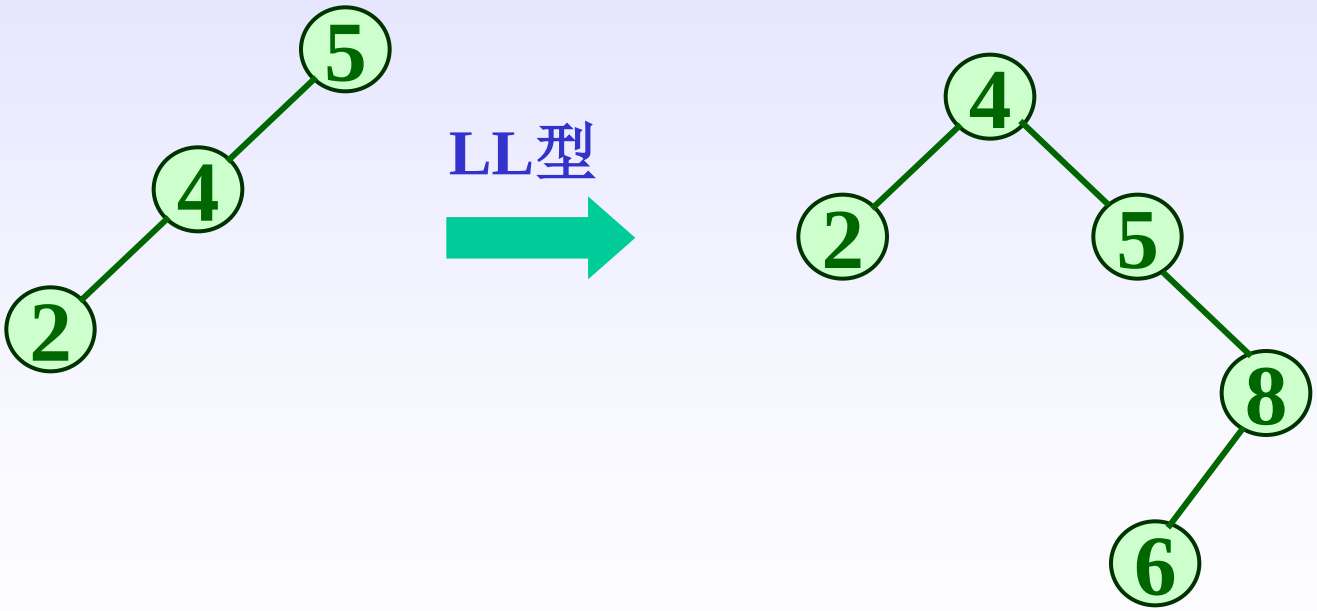


先顺时针旋转
再逆时针旋转



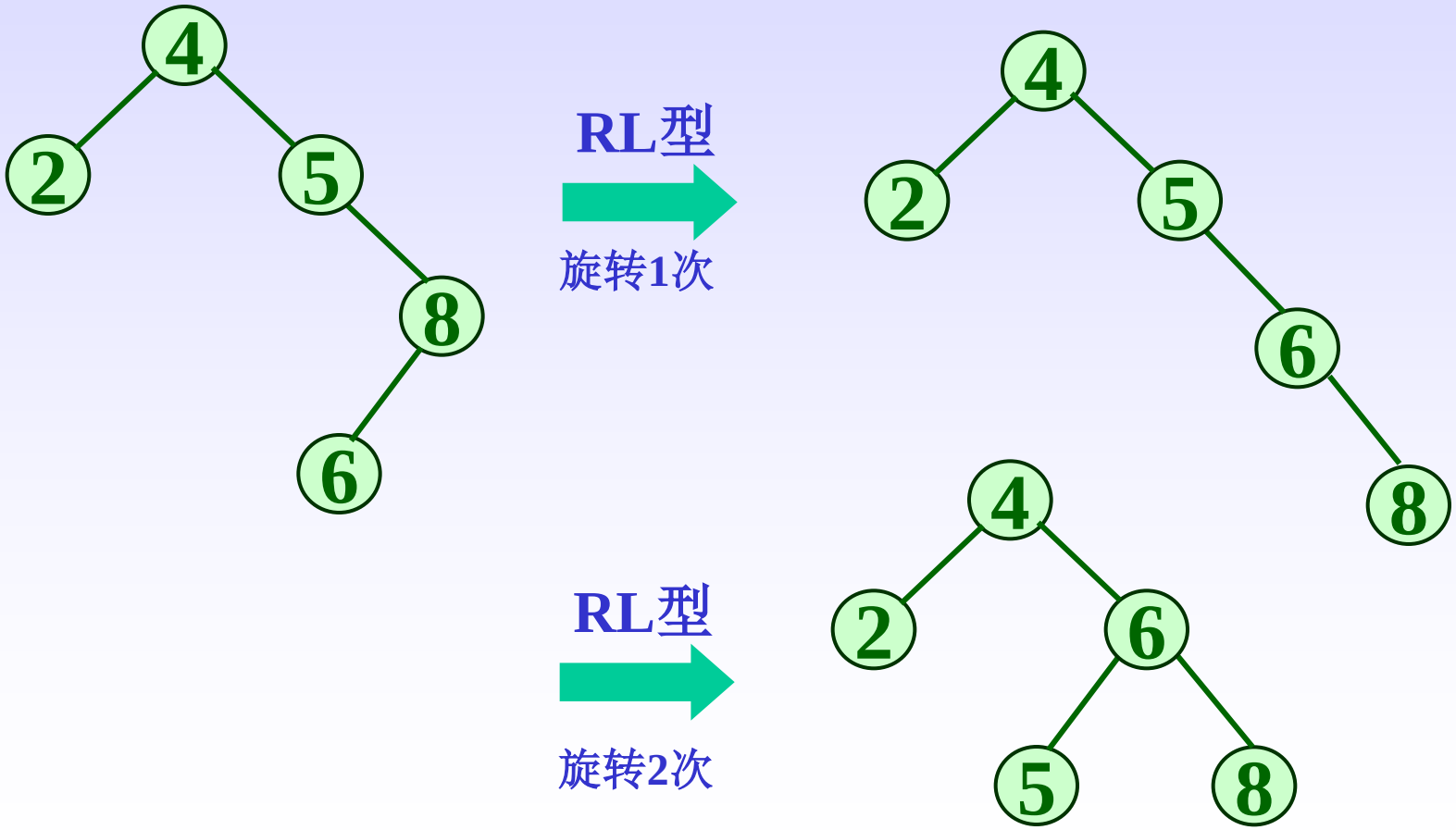
7.3 树表的查找技术

课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



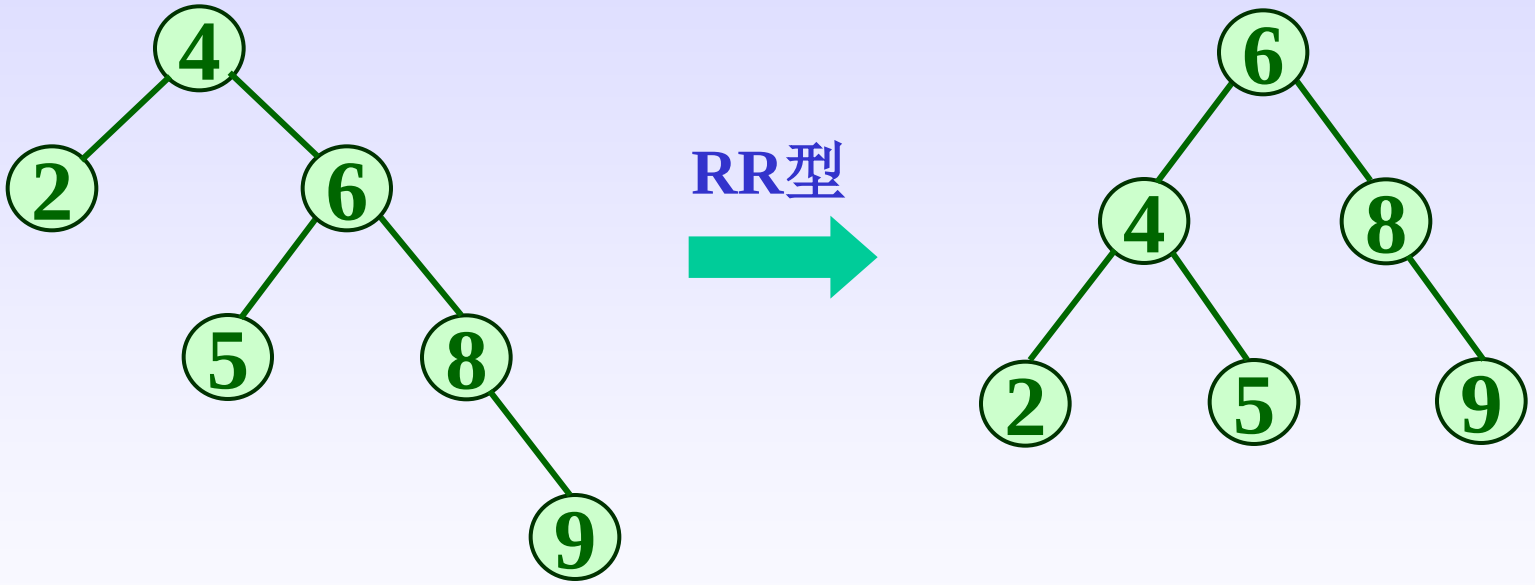
7.3 树表的查找技术

课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



7.3 树表的查找技术

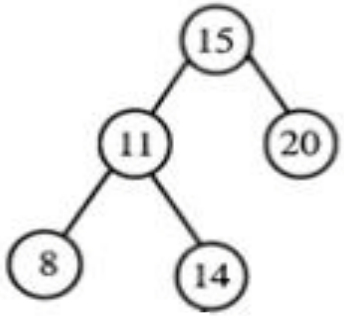
课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



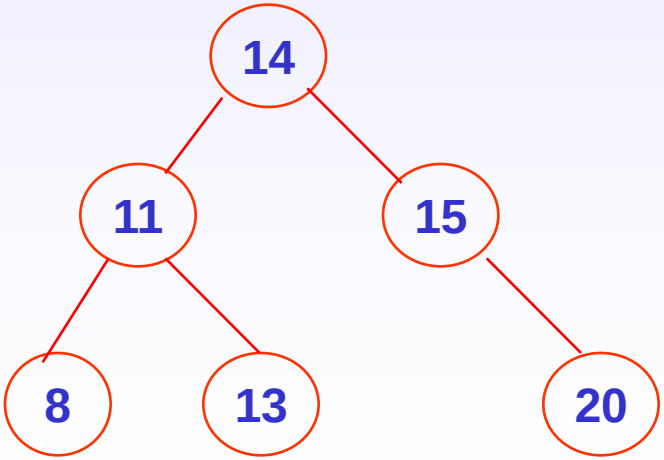
7.3 树表的查找技术

课堂练习：

所示二叉树是否为平衡二叉树？若是，说明理由；若不是，将其转换为平衡二叉树。



LR型



7.3 散列表的查找技术

? 查找操作要完成什么任务？

待查值k ⟹ 确定k在存储结构中的位置

? 我们学过哪些查找技术？这些查找技术的共性？

顺序查找、折半查找、二叉排序树查找等。
这些查找技术都是通过一系列的给定值与关键码的比较，查找效率依赖于查找过程中进行的给定值与关键码的比较次数。

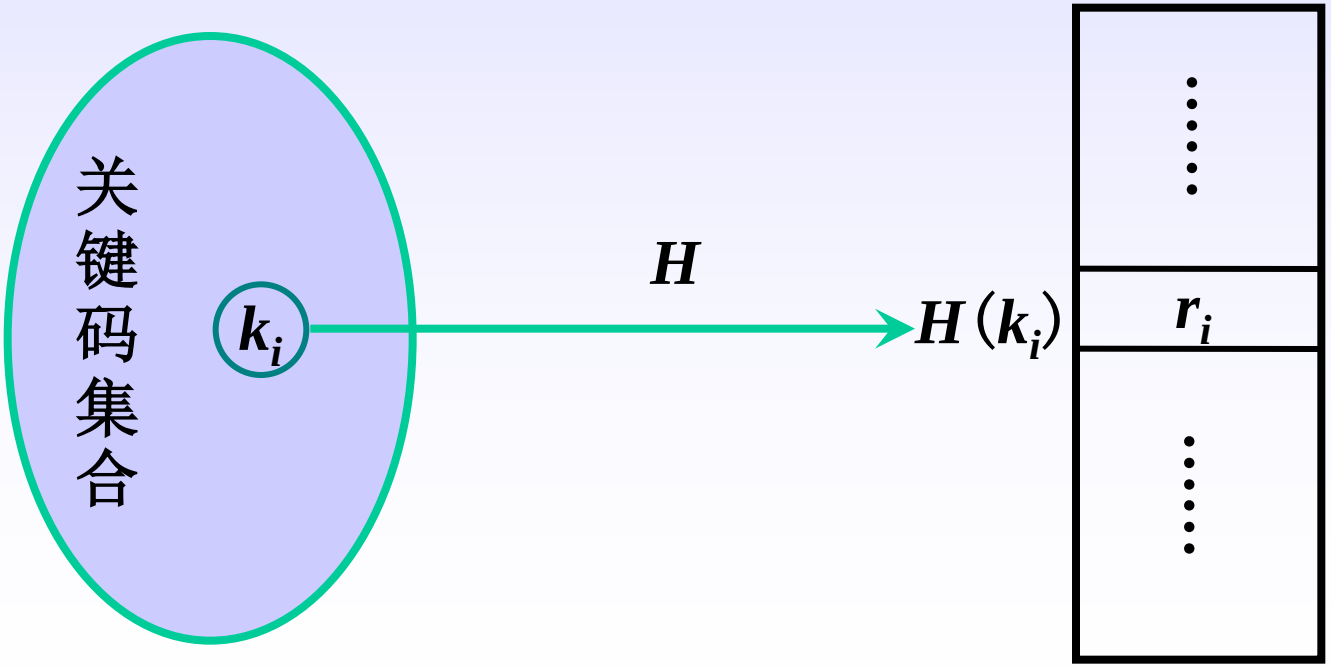
? 能否不用比较，通过关键码直接确定存储位置？

在存储位置和关键码之间建立一个确定的对应关系

7.3 散列表的查找技术

概 述

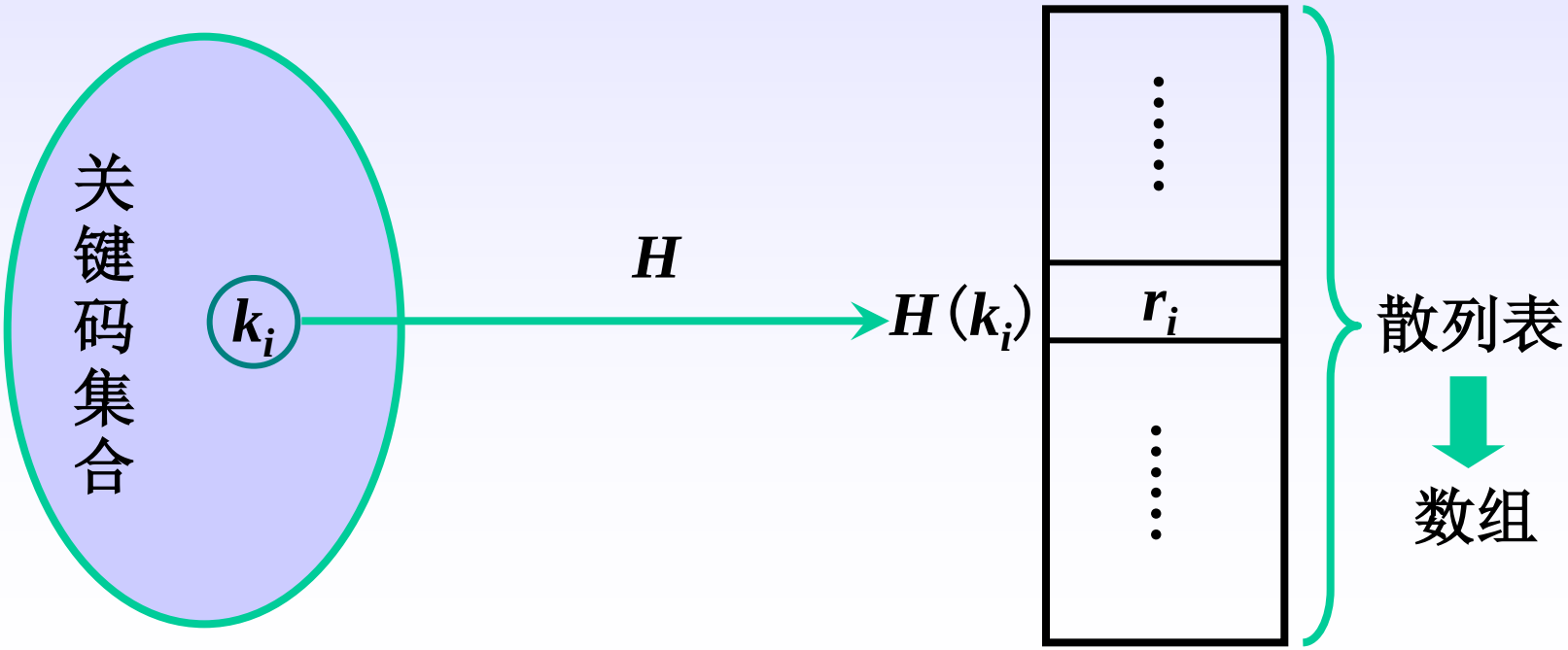
散列的基本思想：在记录的存储地址和它的关键码之间建立一个确定的对应关系。这样，不经过比较，一次读取就能得到所查元素的查找方法。



7.3 散列表的查找技术

概 述

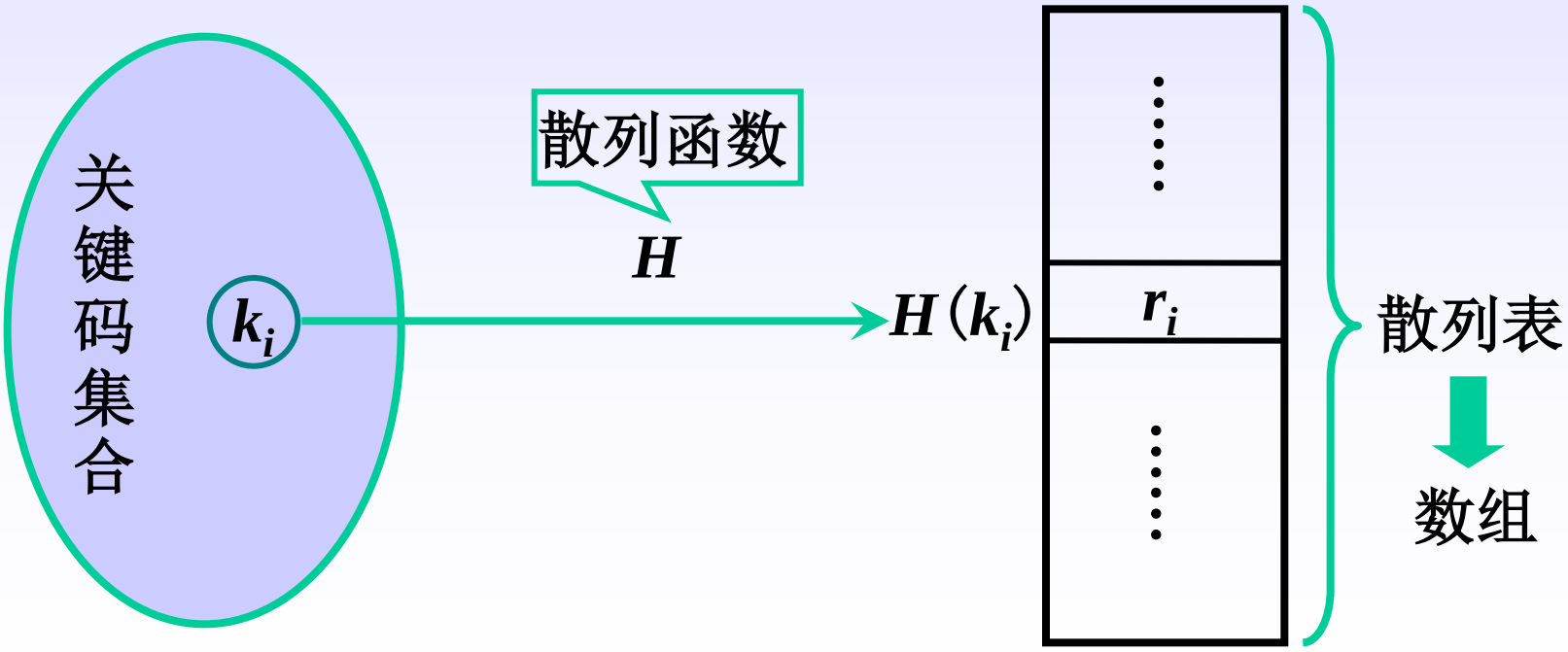
散列表：采用散列技术将记录存储在一块**连续**的存储空间中，这块连续的存储空间称为散列表。



7.3 散列表的查找技术

概 述

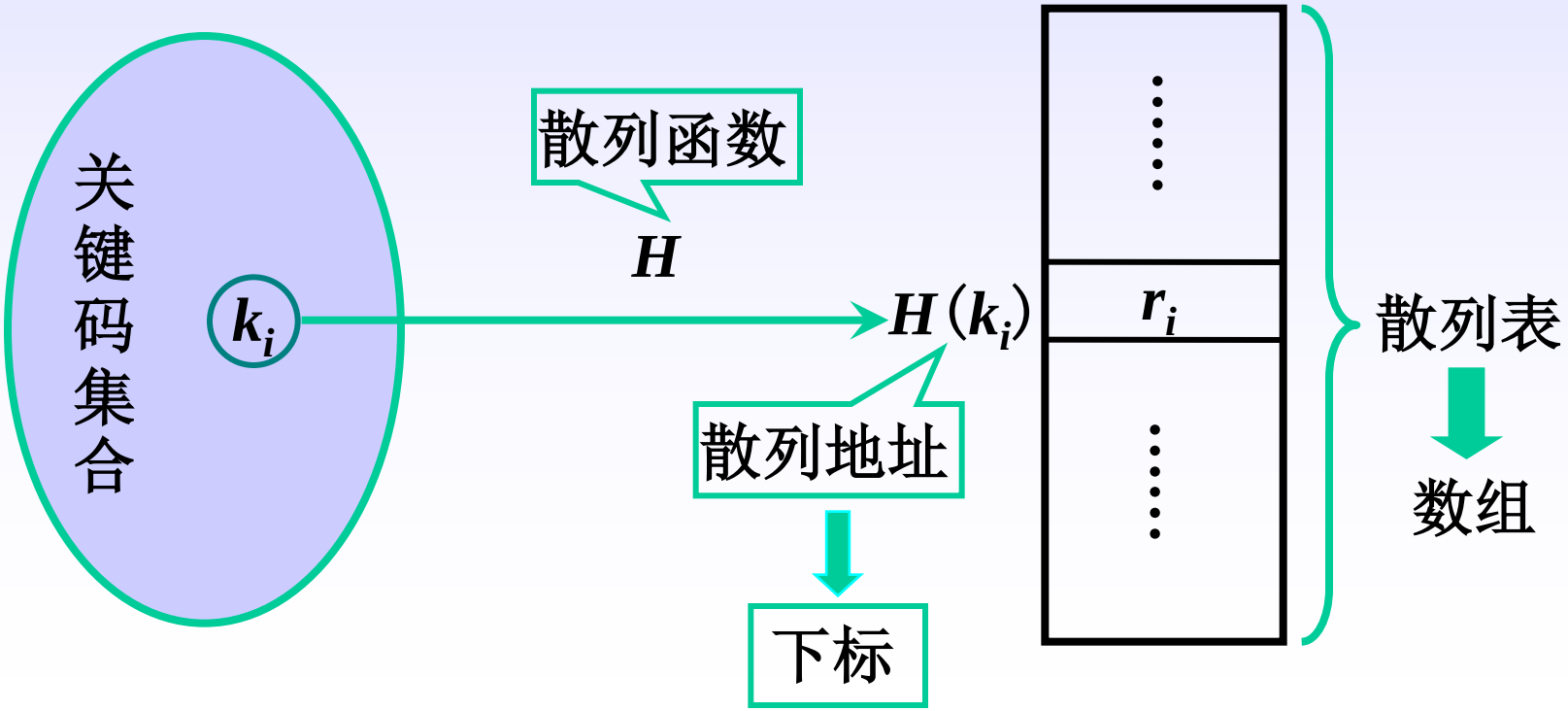
散列函数：将关键码映射为散列表中适当存储位置的函数。



7.3 散列表的查找技术

概 述

散列地址： 由散列函数所得的存储位置 。



7.3 散列表的查找技术

概 述

？散列技术仅仅是一种查找技术吗？

散列既是一种查找技术，也是一种存储技术。

？散列是一种完整的存储结构吗？

散列只是通过记录的关键码定位该记录，没有完整地表达记录之间的逻辑关系，所以，散列主要是面向查找的存储结构。

7.3 散列表的查找技术

概 述

? 散列技术适合于哪种类型的查找？

散列技术一般不适用于允许多个记录有同样关键码的情况。散列方法也不适用于范围查找，换言之，在散列表中，我们不可能找到最大或最小关键码的记录，也不可能找到在某一范围内的记录。

散列技术最适合回答的问题是：如果有的话，哪个记录的关键码等于待查值。

7.3 散列表的查找技术

概 述

散列技术的关键问题：

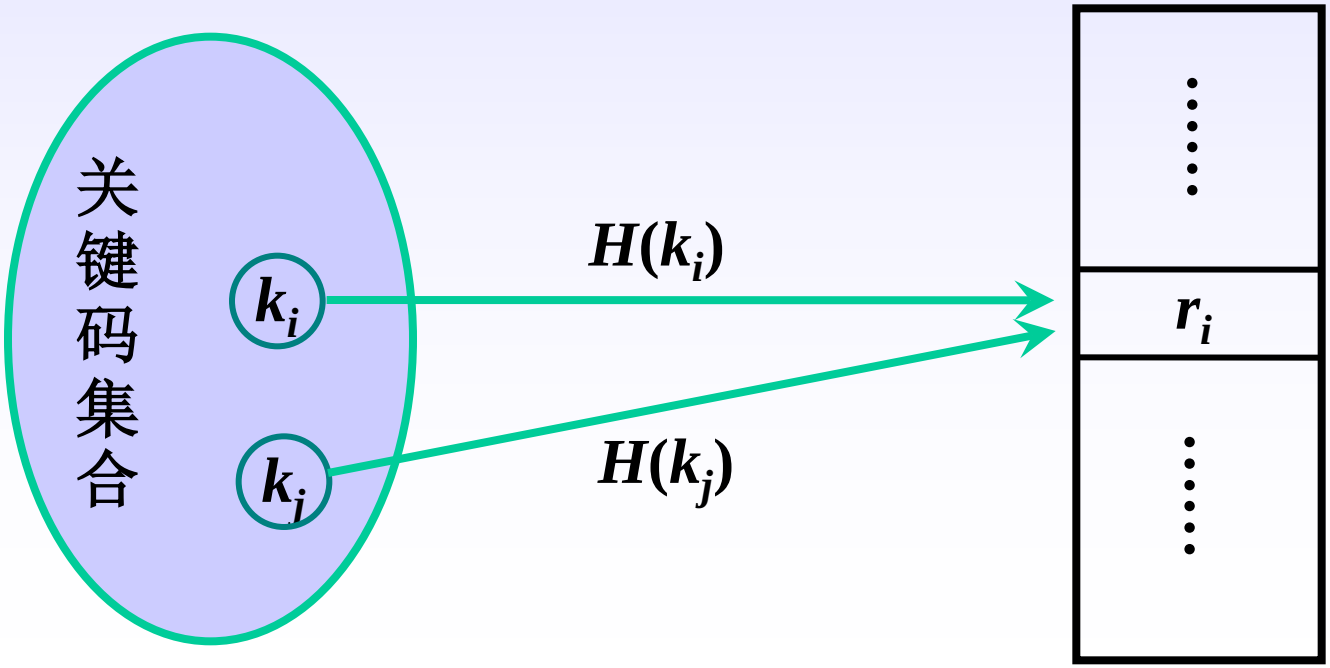
(1) **散列函数**的设计。如何设计一个简单、均匀、存储利用率高的散列函数。

(2) **冲突**的处理。如何采取合适的处理冲突方法来解决冲突。

7.3 散列表的查找技术

概 述

冲突：对于两个不同关键码$k_i \neq k_j$，有$H(k_i)=H(k_j)$，即两个不同的记录需要存放在同一个存储位置，k_i和k_j相对于H称做**同义词**。



7.3 散列表的查找技术

散列函数

设计散列函数一般应遵循以下原则：

⑴ 计算简单。散列函数不应该有很大的计算量，否则会降低查找效率。

⑵ 函数值即散列地址分布均匀。函数值要尽量均匀散布在地址空间，这样才能保证存储空间的有效利用并减少冲突。

7.3 散列表的查找技术

散列函数——直接定址法

散列函数是关键码的线性函数，即：

$$H(key) = a \times key + b \quad （a，b为常数）$$

例：关键码集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为$H(key)=key/10$，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

适用情况？

事先知道关键码，关键码集合不是很大且连续性较好。

7.3 散列表的查找技术

散列函数——除留余数法

散列函数为：

$$H(key)=key \bmod p$$

如何选取合适的 p，产生较少同义词？

例： $p=21=3\times7$

关键码	14	21	28	35	42	49	56
散列地址	14	0	7	14	0	7	14

7.3 散列表的查找技术

散列函数——除留余数法

一般情况下，选*p*为小于或等于表长（最好接近表长）的最小素数或不包含小于20质因子的合数。

适用情况？

除留余数法是一种最简单、也是最常用的构造散列函数的方法，并且不要求事先知道关键码的分布。

7.3 散列表的查找技术

散列函数——数字分析法

根据关键码在各个位上的分布情况，选取分布比较均匀的若干位组成散列地址。

例：关键码为8位十进制数，散列地址为2位十进制数

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

7.3 散列表的查找技术

散列函数——数字分析法

适用情况：

能预先估计出全部关键码的每一位上各种数字出现的频度，不同的关键码集合需要重新分析。

7.3 散列表的查找技术

散列函数——平方取中法

对关键码平方后，按散列表大小，取中间的若干位作为散列地址（平方后截取）。

例：散列地址为2位，则关键码123的散列地址为：

$(1234)^2 = 1522756$

适用情况：

事先不知道关键码的分布且关键码的位数不是很大。

7.3 散列表的查找技术

散列函数——折叠法

将关键码从左到右分割成位数相等的几部分，将这几部分叠加求和，取后几位作为散列地址。

例：设关键码为2 5 3 4 6 3 5 8 7 0 5，散列地址为三位。

```
  253
  463
  587
+  05
------
 1308
```

移位叠加

```
  253
  364
  587
+  50
------
 1254
```

间界叠加

适用情况：

关键码位数很多，事先不知道关键码的分布。

7.3 散列表的查找技术

处理冲突的方法——开放定址法

由关键码得到的散列地址一旦产生了冲突，就去寻找下一个空的散列地址，并将记录存入。

如何寻找下一个空的散列地址?

（1）线性探测法
（2）二次探测法
（3）随机探测法

用开放定址法处理冲突得到的散列表叫闭散列表。

7.3 散列表的查找技术

线性探测法

当发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。

对于键值*key*，设$H(key)=d$，闭散列表的长度为*m*，则发生冲突时，寻找下一个散列地址的公式为：

$$H_i=(H(key)+d_i)\ \%\ m \quad (d_i=1，2，\ldots，m-1)$$

7.3 散列表的查找技术

线性探测法

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为$H(key)=key \bmod 11$，用线性探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22		47	92	16	3	7	29	8
22			3	3	3		29	8	

堆积：在处理冲突的过程中出现的非同义词之间对同一个散列地址争夺的现象。

7.3 散列表的查找技术

在线性探测法构造的散列表中查找算法——伪代码

1. 计算散列地址j;
2. 若ht[j]=k，则查找成功，返回记录在散列表中的下标；
   否则
3. 若ht[j]为空或将散列表探测一遍，则查找失败，转4；
   否则，j指向下一单元，转2；
4. 若整个散列表探测一遍，则表满，抛出溢出异常；
   否则，将待查值插入；

7.3 散列表的查找技术

在线性探测法构造的散列表中查找算法——C++描述

```
int HashSearch1(int ht[ ], int m, int k)
{
    j=H(k);
    if (ht[j]==k) return j;  //没有发生冲突，比较一次查找成功
    i=(j+1) % m;
    while (ht[i]!=Empty && i!=j)
    {
       if (ht[i]==k) return i; //发生冲突，比较若干次查找成功
       i=(i+1) % m;   //向后探测一个位置
    }
    if (i==j) throw "溢出";
    else ht[i]=k;  //查找不成功时插入
}
```

7.3 散列表的查找技术

二次探测法

当发生冲突时，寻找下一个散列地址的公式为：

$$H_i=(H(key)+d_i)\ \%\ m$$

（$d_i=1^2$，-1^2，2^2，-2^2，…，q^2，$-q^2$且$q\leqslant m/2$）

7.3 散列表的查找技术

二次探测法

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为$H(key)=key \bmod 11$，用二次探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22	3	47	92	16		7	29	8
22			3	3			29	8	

7.3 散列表的查找技术

随机探测法

当发生冲突时，下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

$$H_i=(H(\text{key})+d_i)\ \%\ m$$

（d_i是一个随机数列，i=1，2，……，m−1）

计算机中产生随机数的方法通常采用线性同余法，

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1}+c) \bmod m \qquad n = 1, 2, \cdots \end{cases}$$

其中，d称为随机种子。当b、c和m的值确定后，给定一个随机种子，产生确定的随机数序列。

7.3 散列表的查找技术

处理冲突的方法——拉链法（链地址法）

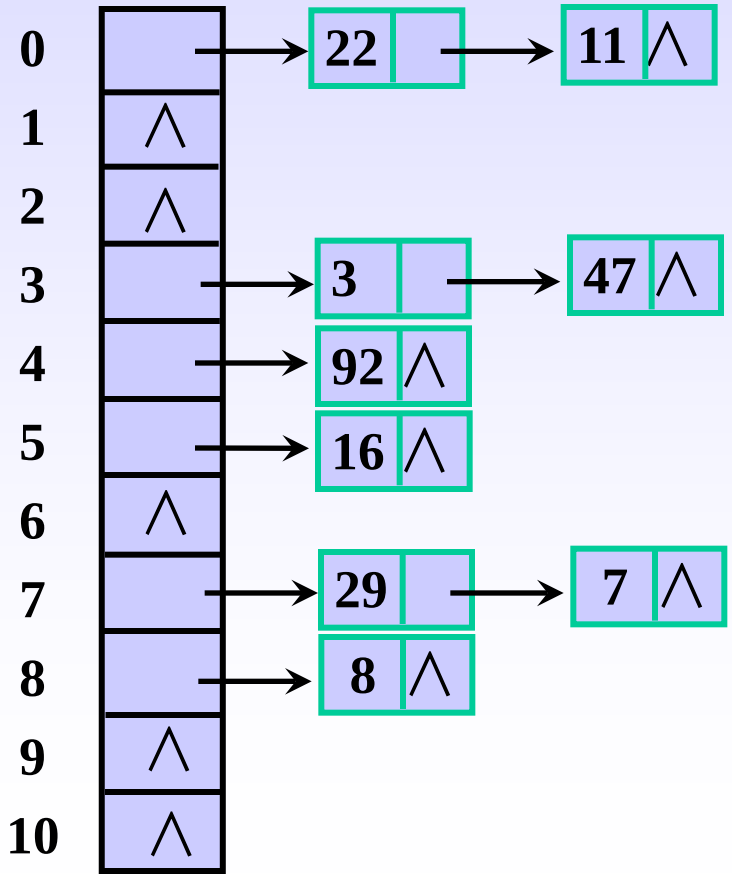
基本思想：将所有散列地址相同的记录，即所有同义词的记录存储在一个单链表中（称为同义词子表），在散列表中存储的是所有同义词子表的头指针。

用拉链法处理冲突构造的散列表叫做开散列表。

设n个记录存储在长度为m的散列表中，则同义词子表的平均长度为n / m。

7.3 散列表的查找技术

例：关键码集合 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为$H(key)=key \bmod 11$，用拉链法处理冲突，构造的开散列表为：



7.3 散列表的查找技术

在拉链法构造的散列表查找算法——伪代码

1. 计算散列地址j；
2. 在第j个同义词子表中顺序查找；
3. 若查找成功，则返回结点的地址；

   否则，将待查记录插在第j个同义词子表的表头。

7.3 散列表的查找技术

在拉链法构造的散列表查找算法——C++描述

```
Node<int> *HashSearch2(Node<int> *ht[ ], int m, int k)
{
    j=H(k);
    p=ht[j];
    while (p && p->data!=k)
        p=p->next;
    if (p->data= =k) return p;
    else {
        q=new Node<int>; q->data=k;
        q->next= ht[j];
        ht[j]=q;
    }
}
```

7.3 散列表的查找技术

处理冲突的方法——公共溢出区

基本思想：散列表包含基本表和溢出表两部分（通常溢出表和基本表的大小相同），将发生冲突的记录存储在溢出表中。查找时，对给定值通过散列函数计算散列地址，先与基本表的相应单元进行比较，若相等，则查找成功；否则，再到溢出表中进行顺序查找。

7.3 散列表的查找技术

例：关键码集合 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为$H(key)=key \bmod 11$，用公共溢出区法处理冲突，构造的散列表为：

	基本表
0	11
1	
2	
3	47
4	92
5	16
6	
7	7
8	8
9	
10	

	溢出表
0	29
1	22
2	3
3	
4	
5	
6	
7	
8	
9	
10	

7.3 散列表的查找技术

散列查找的性能分析

由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。

在查找过程中，关键码的比较次数取决于产生冲突的概率。而影响冲突产生的因素有：

（1）散列函数是否均匀

（2）处理冲突的方法

（3）散列表的装载因子

α =表中填入的记录数/表的长度

7.3 散列表的查找技术

几种不同处理冲突方法的平均查找长度

处理冲突方法 \ ASL	查找成功时	查找不成功时
线性探测法	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$	$\frac{1}{2}(1+\frac{1}{1-\alpha^2})$
二次探测法	$-\frac{1}{\alpha}\ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

7.3 散列表的查找技术

开散列表与闭散列表的比较

	堆积现象	结构开销	插入/删除	查找效率	估计容量
开散列表	不产生	有	效率高	效率高	不需要
闭散列表	产生	没有	效率低	效率低	需要