第七章 查找与索引

- 查找与索引的概念
- 基于顺序表的查找
- 二叉查找树
- **B-树和B+树（略）**
- **Trie树（略）**
- **Hash查找**
- 本章小结

7.1 查找与索引的概念

- 搜索(Search)是数据处理中最常见的一种操作。
  - 基本功能是针对给定的一个值k，从一个数据对象的集合中寻找关键码或某个属性为k的数据对象。如果找到则返回该数据对象，如果没有找到则返回失败标志。
- 搜索操作的实施方法和计算复杂性同数据的存储结构有很大的关系，总体讲数据的存储结构和操作方式可以有如下两种分类方式：
  - 静态和动态
    - 静态数据结构不考虑插入和删除操作。
    - 动态数据结构在使用过程中频繁地有数据插入和删除，数据结构需要在动态的过程中维持数据在数据结构中的数据分布模式。
  - 内存和外存
    - 内存数据是指整个数据结构完全放在内存中。
    - 基于外存的存储结构是指数据部分或全部存放在磁盘上，为此在搜索和数据组织中需要充分考虑磁盘的读写特点。

- 索引(Index)是用于支持搜索操作的数据结构及其操作，其目标是提高搜索操作的性能。目前索引被认为是提升大规模数据查询和检索操作性能的最有效的手段之一。
- 在设计一个索引结构的过程中需要从如下方面进行考虑：
  - 能够实现对大规模数据的管理能力。
  - 能够支持高效地搜索操作。
  - 能够有效地支持频繁的插入、删除和修改操作。
  - 索引结构需要考虑数据存储介质的特点。例如，在以磁盘为存储介质的索引结构应该考虑磁盘按块存取的特点。
  - 能够有效地支持数据结构本身的特点。

静态搜索表

搜索(Search)的概念

- 所谓搜索，就是在数据集合中寻找满足某种条件的数据对象。
- 搜索的结果通常有两种可能：
  - 搜索成功，即找到满足条件的数据对象。作为结果，可报告该对象在结构中的位置，还可给出该对象中的具体信息。
  - 搜索不成功，或搜索失败。作为结果，应报告一些信息，如失败标志、位置等。

- 通常称用于搜索的数据集合为搜索结构，由同一数据类型的对象（或记录）组成。
- 在每个对象中有若干属性，其中有一个属性，其值可惟一地标识这个对象，称为关键码。使用基于关键码的搜索，搜索结果应是惟一的。但在实际应用时，搜索条件是多方面的，可以使用基于属性的搜索方法，但搜索结果可能不惟一。

- 实施搜索时有两种不同的环境。
  - 静态环境——搜索结构在插入和删除等操作的前后不发生改变。

    —— 静态搜索表
  - 动态环境——为保持较高的搜索效率，搜索结构在执行插入和删除等操作的前后将自动进行调整，结构可能发生变化。

    —— 动态搜索表

静态搜索表

```
template <class Type> class DataList;
template <class Type> class Node {
  friend class DataList <Type>;
  private:
    Type key;
    other;
  public:
    Node(const Type &value) : key(value) { }
    Type GetKey( ) const { return key; }
    void SetKey(Type k) { key=k; }
};
```

```
template <class Type> class DataList {
  protected:
    Node <Type> *Element;  //数据存储数组
    int ArraySize;  //存储数组最大长度
    int CurrentSize;  //存储数组当前长度
  public:
    DataList(int sz=10) : ArraySize(sz) {
        Element=new Node <Type> [sz];
        CurrentSize=0;
    }
```

```
    virtual ~DataList( ) { delete [ ] Element; }
    int Length( ) { return CurrentSize; }
    friend ostream & operator <<
      (ostream &OutStream,
              const DataList <Type> &OutList);
    friend istream & operator >>
      (istream &InStream,
              DataList <Type> &InList);
};
```

```
template <class Type> class SearchList
                              : public DataList <Type> {
//作为派生类的静态搜索表的类定义
  public:
    SearchList(int sz=10) :
                          DataList <Type> (sz) { }
    virtual ~SearchList( ) { }
    virtual int Search(const Type &x) const;
};
```

```
template <class Type> istream & operator >>
(istream &InStream, DataList <Type> &InList)
{  //从输入流对象InStream输入数据到表InList中
   cout<<"输入数组当前大小：";
   instream>>InList.CurrentSize;
   cout<<"输入数组元素值：\n";
   for (int i=0; i<InList.CurrentSize; i++) {
      cout<<"元素"<<i<<"：";
      InStream>>InList.Element[i];
   }
   return InStream;
}
```

```
template <class Type> ostream & operator <<
(ostream &OutStream,
            const DataList <Type> &OutList) {
//将表OutList内容输出到输出流对象OutStream
    OutStream<<“数组内容：  \n”;
    for (int i=0; i<OutList.CurrentSize; i++)
       OutStream<<OutList.Element[i]<<‘ ’;
    OutStream<<endl;
    OutStream<<“数组当前大小：  ”<<
                        OutList.CurrentSize<<endl;
    return OutStream;
}
```

- 衡量一个搜索算法的时间效率的标准是：在搜索过程中关键码的平均比较次数或平均读写磁盘次数（只适合于外部搜索），也称为平均搜索长度ASL(Average Search Length)，通常它是搜索结构中对象总数n或文件结构中物理块总数n的函数。
- 在静态搜索表中，利用数组元素的下标作为数据对象的存放地址。搜索算法根据给定值x，在数组中进行搜索。直到找到x在数组中的位置或可确定在数组中找不到x为止。
- 另外，衡量一个搜索算法还要考虑算法所需要的存储量和算法的复杂性等问题。

7.2 基于顺序表的查找

- 顺序表是一种基本的数据结构类型。
  - 数据在数组中可以有两种存储方式，一种是有序的，另一种是无序的。对于无序的顺序表只能用顺序搜索的方式进行搜索，而如果数据是以有序的方式存储的，则可以用折半搜索、斐波那契搜索等方法进行搜索。
  - 同数组相比链表不需要事先申请连续的空间，同时其长度也没有上限，因此也是一种常用的线性表结构。

顺序搜索 (Sequential Search)

- 如下情况下线性表结构上的搜索需要采用顺序搜索的方法。
  - 数组中的数据是无序的；
  - 数据存储在链表中，而链表上又没有辅助的索引结构。
- 顺序搜索的计算复杂性为**O(n)**，其中**n**为线性表的长度。

- 所谓顺序搜索，又称线性搜索，主要用于在线性结构中进行搜索。
- 设若表中有**CurrentSize**个对象，则顺序搜索从表的先端开始，顺序用各对象的关键码与给定值**x**进行比较，直到找到与其值相等的对象，则搜索成功，给出该对象在表中的位置。
- 若整个表都已检测完仍未找到关键码与**x**相等的对象，则搜索失败，给出失败信息。

```
template <class Type> int SearchList <Type> ::
SeqSearch(const Type &x) const {
//顺序搜索关键码为x的对象
//第CurrentSize号位置
//作为控制搜索自动结束的“监视哨”使用
   Element[CurrentSize].key=x;
   int i=0;  //将x送CurrentSize号位置设监视哨
   while (Element[i].key!=x)  i++;
   //从前向后顺序搜索
   return i;
}
```

```
const int Size=10;
main( ) {  //顺序搜索的主过程
    SearchList <float> List1(Size);  //定义搜索表
    float Target;
    cin>>List1;  cout<<List1;  //输入List1
    int Location, len=List1.Length( );
    cout<<"搜索浮点数：";
    cin>>Target;
    if ((Location=List1.SeqSearch(Target))!=len)
        cout<<"找到位置在"<<Location<<endl;
        //搜索成功，输出找到的位置
    else cout<<"没有找到。\n";  //搜索失败
}
```

顺序搜索的平均搜索长度

设搜索第i个元素的概率为p_i，搜索到第i个元素所需比较次数为c_i，则搜索成功的平均搜索长度：

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot c_i . \qquad (\sum_{i=0}^{n-1} p_i = 1)$$

在顺序搜索并设置“监视哨”情形：

$c_i = i+1$，$i = 0, 1, \ldots, n-1$，因此

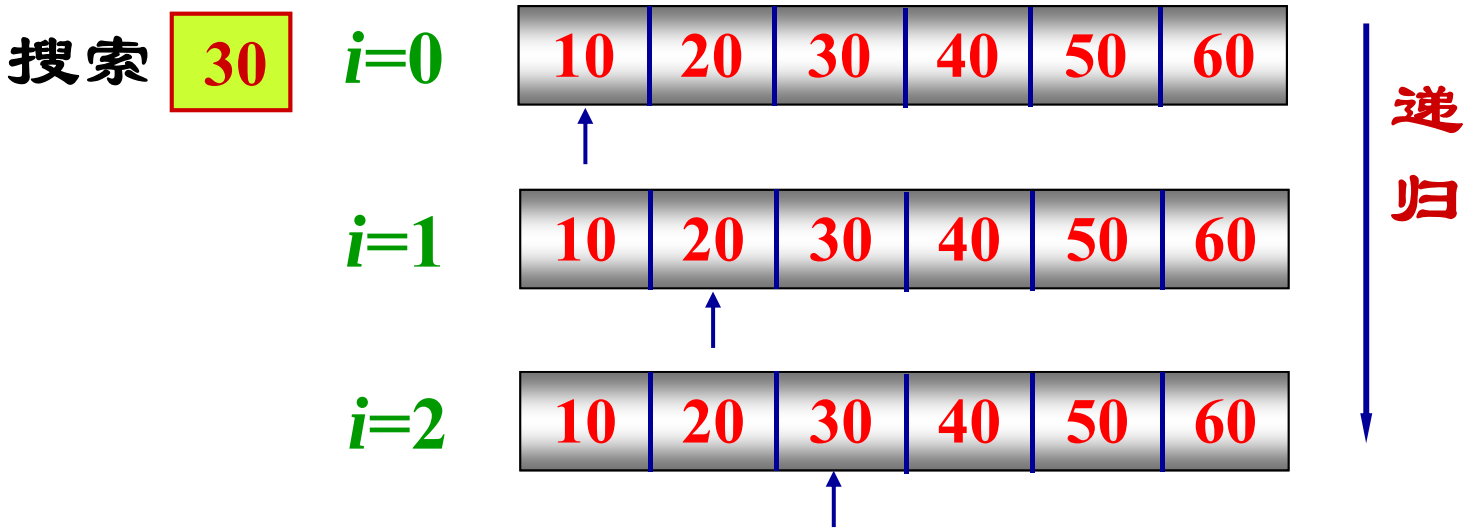
$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot (i+1)$$

在等概率情形，$p_i=1/n$, $i=1, 2, \ldots, n$。

$$ASL_{succ}=\sum_{i=0}^{n-1}\frac{1}{n}(i+1)=\frac{1}{n}\cdot\frac{n(n+1)}{2}=\frac{n+1}{2}$$

在搜索不成功情形，$ASL_{unsucc}=n+1$。

顺序搜索的递归算法



顺序搜索的递归算法

```
template <class Type> int SearchList <Type> ::
SeqSearch(const Type &x,  int loc) const {
//在数据表Element[0..n-1]中搜索其关键码
//与给定值匹配的对象，函数返回其表中位置
//参数loc是在表中开始搜索位置
   if (loc>=CurrentSize)  return -1;  //搜索失败
   else if (Element[loc].key==x)  return loc;
   //搜索成功
   else return Search(x, loc+1);  //递归搜索
}
```

有序顺序表上的顺序搜索

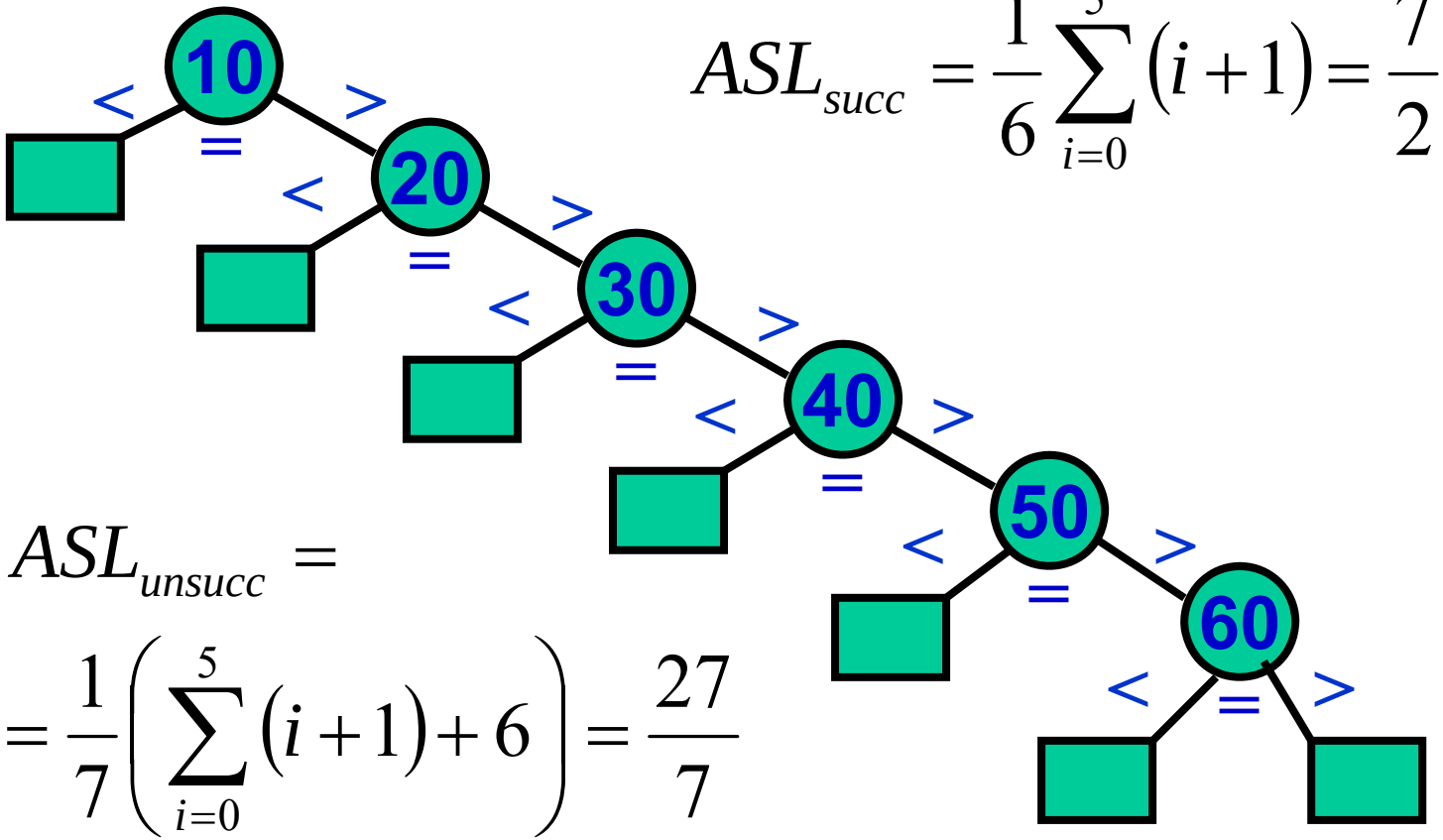
- 如果数据的排列是有序的，则存在很多高效的算法，如折半搜索、斐波那契搜索等。
  - 折半搜索
    - 循环
      - 首先，和线性表中间位置的数据进行比较；
      - 如果比中间位置的数据大，则继续数组的后半部进行搜索；
      - 如果比中间位置的数据小，则在前半部继续进行搜索。
    - 每次循环搜索的空间都是原来的1/2，直到完成搜索。这样最多$\log_2 n$（n为线性表的长度）次循环就可以完成。

基于有序顺序表的顺序搜索算法

```
template <class Type> int SortedList <Type> ::
SequentSearch(const Type &x) const {
//顺序搜索关键码为x的数据对象
    for (int i=0; i<CurrentSize; i++)
      if (Element[i].key==x)  return i;  //成功
      else if (Element[i].key>x) break;
    return -1;
    //顺序搜索失败，返回失败信息
}
```

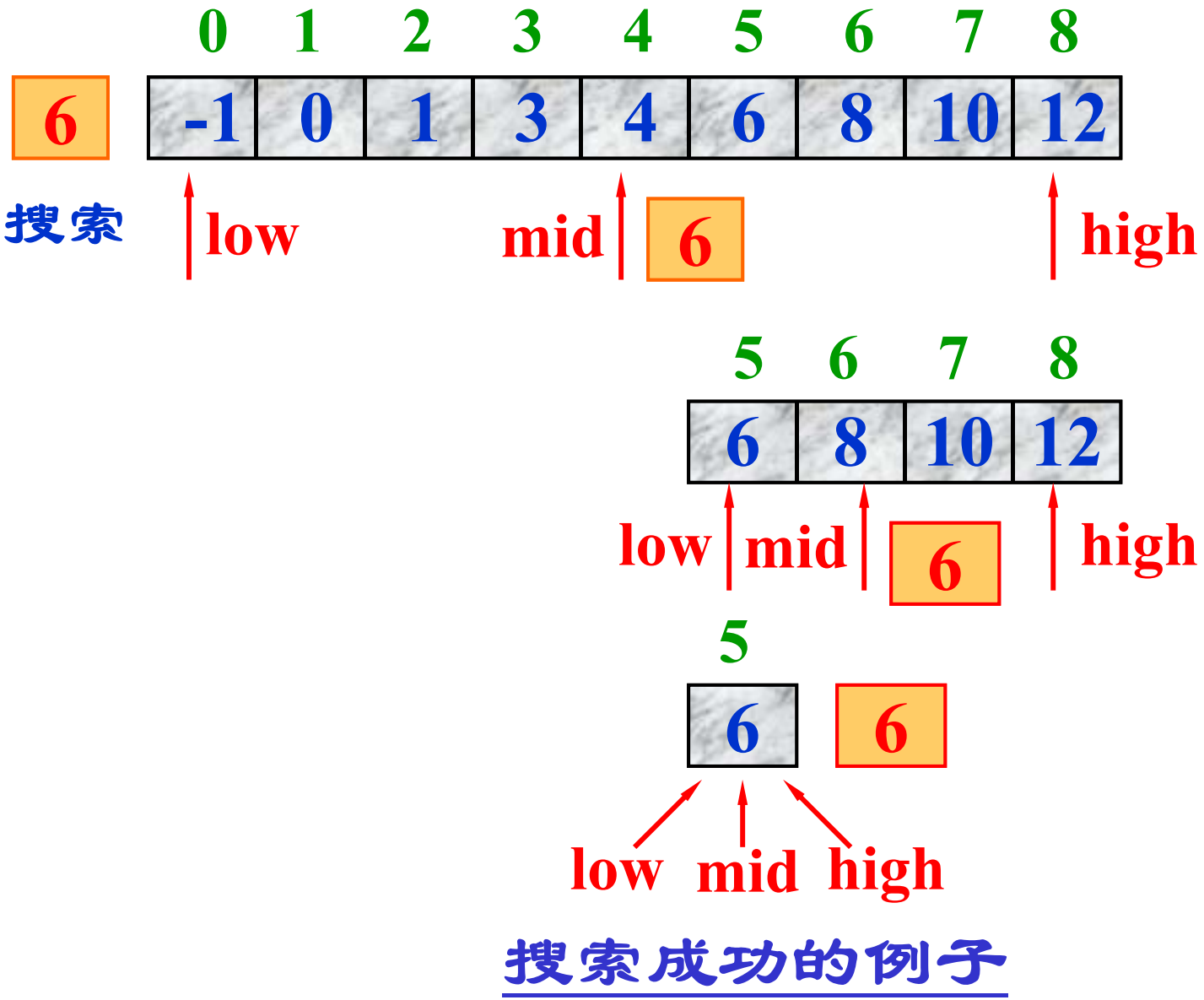
- 有序顺序表的顺序搜索

(10, 20, 30, 40, 50, 60)

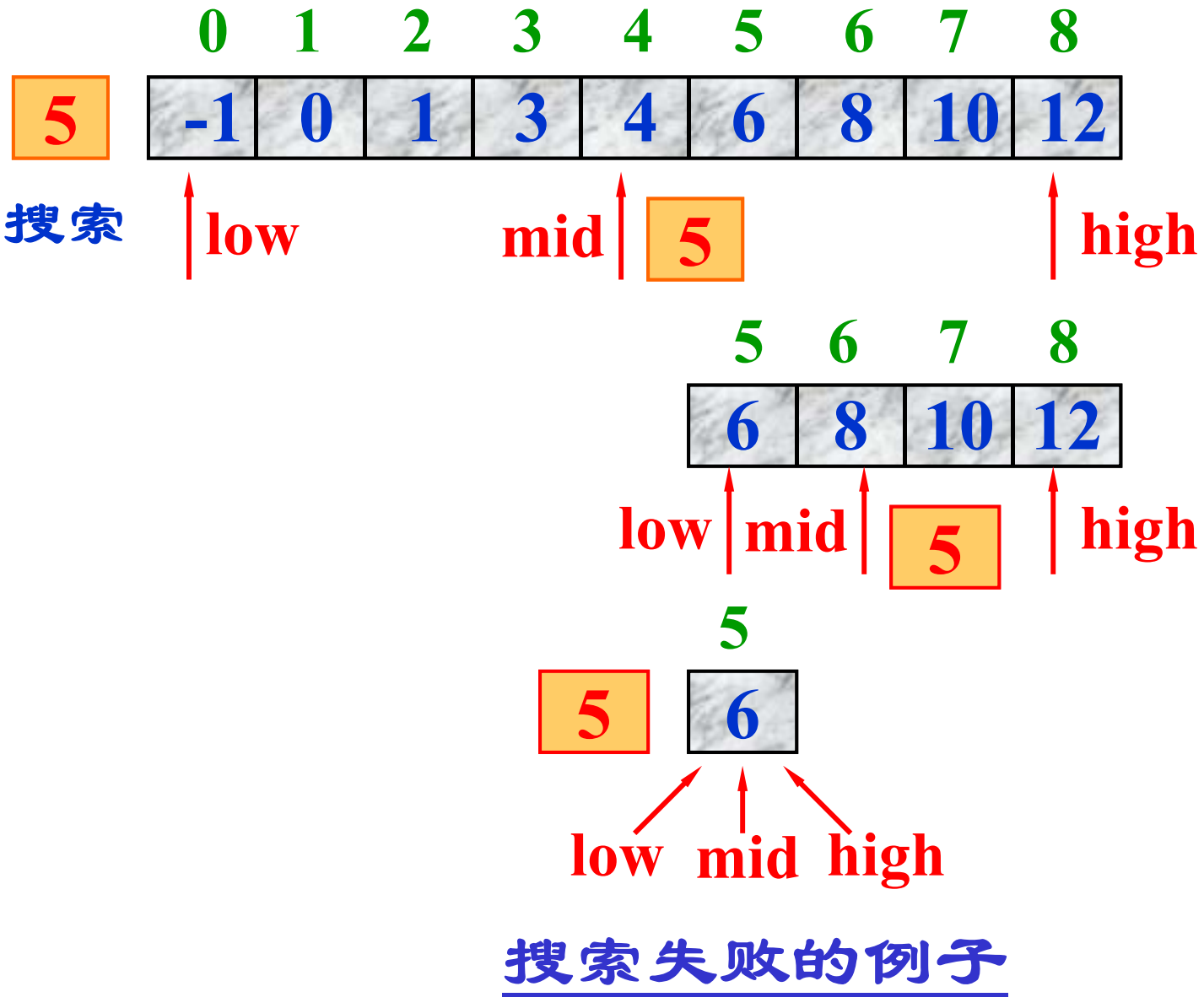


基于有序顺序表的折半搜索

- 设**n**个对象存放在一个有序顺序表中，并按其关键码从小到大排好了序。
- 折半搜索时，先求位于搜索区间正中的对象的下标**mid**，用其关键码与给定值**x**比较：
  - **Element[mid].key==x**，搜索成功；
  - **Element[mid].key>x**，把搜索区间缩小到表的前半部分，继续折半搜索；
  - **Element[mid].key<x**，把搜索区间缩小到表的后半部分，继续折半搜索。
- 如果搜索区间已缩小到一个对象，仍未找到想要搜索的对象，则搜索失败。


0 1 2 3 4 5 6 7 8
6
-1 0 1 3 4 6 8 10 12
搜索
low
mid
6
high
5 6 7 8
6 8 10 12
low
mid
6
high
5
6
6
low mid high

搜索成功的例子


0 1 2 3 4 5 6 7 8
5
-1 0 1 3 4 6 8 10 12
搜索
low
mid
5
high
5 6 7 8
6 8 10 12
low
mid
5
high
5
5
6
low mid high

搜索失败的例子

```
template <class Type> class SortedList :
                                    public SearchList <Type>
{ //有序表的类定义，继承了数据表
  public:
    SortedList(int sz=10) :
                    SearchList <Type> (sz) { }
    ~SortedList( ) { }
    int BinarySearch(const Type &x) const;
};
```

```
template <class Type> int SortedList <Type> ::
BinarySearch(const Type &x, const int low,
                        const int high) const
{  //折半搜索的递归算法
   int mid=-1;
   if (low<=high) {
     mid=(low+high)/2;
     if (Element[mid].key<x)
        mid=BinarySearch(x, mid+1, high);
     else if (Element[mid].key>x)
        mid=BinarySearch(x, low, mid-1);  }
   return mid;
}
```

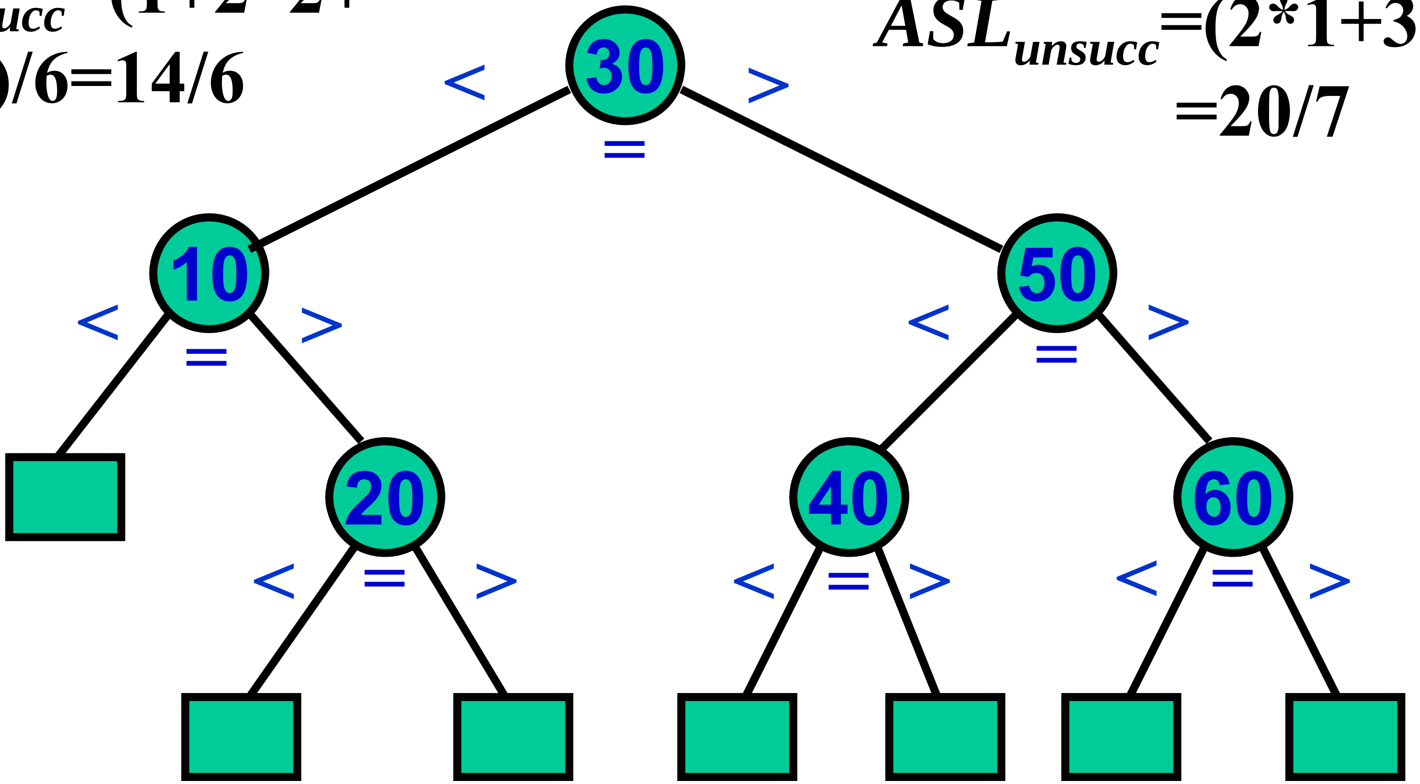
```
template <class Type> int SortedList <Type> ::
BinarySearch(const Type &x) const {
//折半搜索的迭代算法
   int high=CurrentSize-1,  low=0,  mid;
   while (low<=high) {
      mid=(low+high)/2;
      if (Element[mid].key<x)
         low=mid+1;  //右缩搜索区间
      else if (Element[mid].key>x)
               high=mid-1;  //左缩搜索区间
      else return mid;  //搜索成功  }
   return -1;  //搜索失败
}
```

有序顺序表的折半搜索的判定树

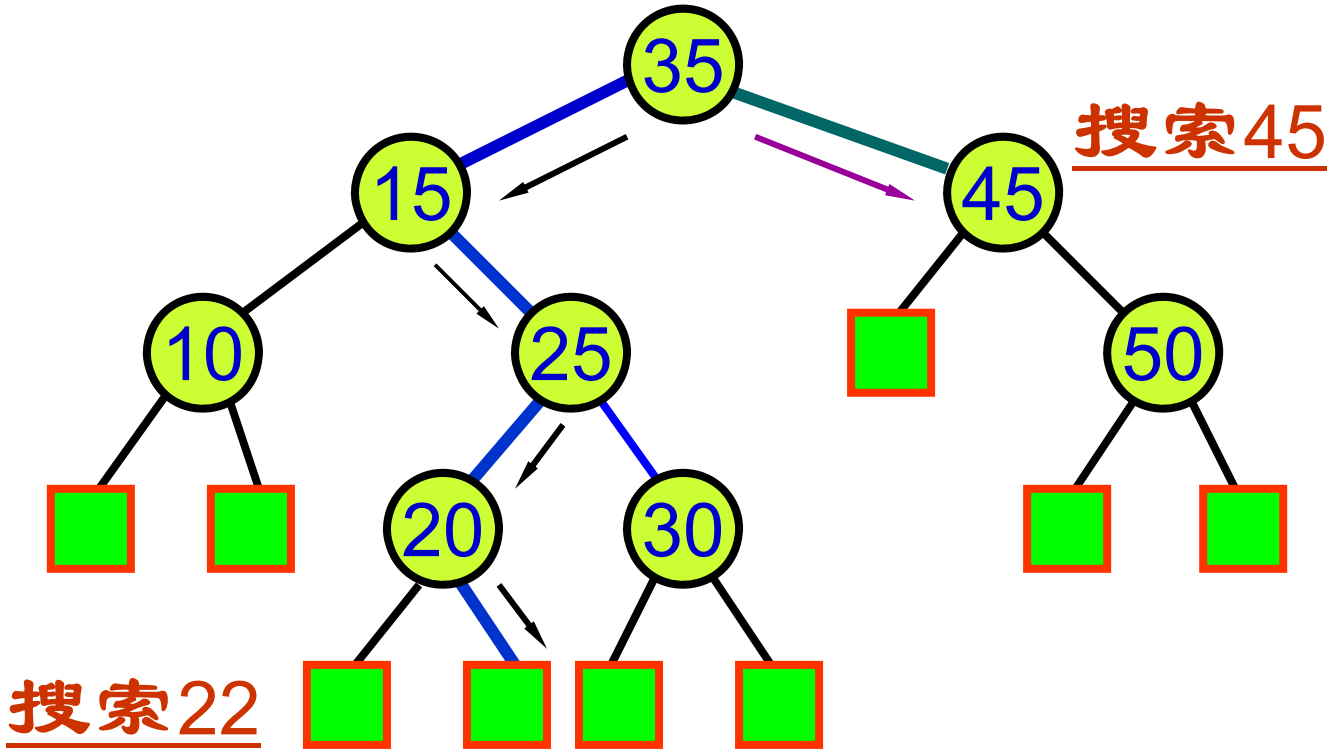
(10, 20, 30, 40, 50, 60)

$ASL_{succ}=(1+2*2+3*3)/6=14/6$

$ASL_{unsucc}=(2*1+3*6)/7=20/7$



- 搜索成功时检测指针停留在树中某个结点。
- 搜索不成功时检测指针停留在某个外结点（失败结点）。



折半搜索性能分析

- 若设$n=2^h-1$，则描述折半搜索的判定树是高度为$h-1$的满二叉树。

$$2^h=n+1,\ h=\log_2(n+1)$$

- 第0层结点有1个，搜索第0层结点要比较1次；第1层结点有2个，搜索第1层结点要比较2次；…，第 i $(0\le i<h)$ 层结点有2^i个，搜索第i层结点要比较$i+1$次，…。
- 假定每个结点的搜索概率相等，即$p_i=1/n$，则搜索成功的平均搜索长度为

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot C_i = \frac{1}{n}\sum_{i=0}^{n-1} C_i = \frac{1}{n}(1*1+2*2^1+$$

$$+3*2^2+\cdots+(h-1)\times 2^{h-2}+h\times 2^{h-1})$$

可以用归纳法证明

$$1\times 1+2\times 2^1+3\times 2^2+\cdots+(h-1)\times 2^{h-2}+h\times 2^{h-1}=$$

$$=(h-1)\times 2^h+1$$

这样

$$ASL_{succ} = \frac{1}{n}((h-1)\times 2^h+1) = \frac{1}{n}((n+1)\log_2(n+1)-n)$$

$$= \frac{n+1}{n}\log_2(n+1)-1 \approx \log_2(n+1)-1$$

例1

- 设有一个有序的线性表{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}，需要搜索关键字为8的数据。
  - 首先，和线性表的中间位置的数据5进行比较，发现它比5大，所以第二次就和后面一半（即6-11）的中间结点8进行比较。
    - 这样，通过2次比较就可以搜索到最终的数据。
    - 而如果采用顺序搜索的方法，则需要经过9次比较才能找到。

插值搜索

- 如果事先已经知道数据的分布情况，就可以充分利用数据的分布情况提高搜索的性能。
  - 其基本思路是根据搜索数据的取值预估被搜索数据在数据结构中的位置。
  - 与折半查找的主要差别在于中点的计算充分考虑了检索数据的取值和数据的分布。

插值搜索算法

```
template <class ElementType> int List <Element Type> ::
search(Element n)
{
        int i=0;  int low=0, high=length-1, tmp;
        low=0;
        high=m;
        while (low<=high)  {
        //先根据插值公式计算中值，假设数据是均匀分布
            if (n!=getvalue(i))  i++;
            else {
                tmp=(n-getvalue(low))/(getvalue(high)-getvalue(low))
                        *(high-low)+low;
                if (n==getvalue(tmp))  return (tmp);
                if (n>=getvalue(tmp))  low=tmp+1;
                else  high=tmp-1;
            }
        }
        return -1;
}
```

例2

- 设有一个有序的线性表{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}，需要搜索关键字为8的数据。
  - 由于8在0-11的区间中的位置是66.6%，首先和线性表中位于66.6%位置的数据进行比较。
    - 在这个位置上的数据就是8，因此通过1次比较就能找到最终的结果。

斐波那契搜索

- 插值搜索的前提是数据在线性表中分布比较均匀，但在很多情况下数据的分布是不均匀的，而且其分布也是事先未知的。
  - 为了追求更好的平均性能，斐波那契搜索是一个很好的选择，其基础是斐波那契数。斐波那契数的定义是：
    - **F(n)=n, n=0, 1; 或F(n)=F(n-1)+F(n-2)**
  - 对于长度为**F(m)**的搜索空间，其比较的中点是**tmp=F(m-1)**。如果**n<list[tmp]**，则在前**F(n-1)**的区间中进一步比较，否则在后**F(n-2)**的区间中进一步比较。

斐波那契算法

```
//程序7-5 斐波那契搜索
template <class ElementType> int List <ElementType> ::
search(ElementType n)
{
    int i, low=0, high=length-1, tmp;
    while (low<=high) {
        i=reFib(high-low);  //获取上述F(n)中的n，不足向上取整
        tmp=Fib(i-1)+low;  //取比较中点，这里规定Fib(-1)=0
        if (getvalue(tmp)==n)  return tmp;
        if (getvalue(tmp)<n)  low=tmp+1;
        else   high=tmp-1;
    }
    return -1;
}
```

- 在斐波那契搜索中搜索空间的缩减速度近乎黄金分割的比例，所以在大部分的数据分布的情况下具有比较好的性能。
- 可以用链表的方式来实现线性表，但由于链表的结点的随机访问并不方便，所以一般仍然用顺序搜索。

随堂练习

例1：试述顺序查找法、折半查找法对被查找的表中元素的要求，对长度为n的表来说，两种查找法在查找成功时的搜索长度各是多少？

两种方法对查找的要求如下：

(1) 顺序查找法：表中的元素可以任意存放，没有任何限制。

(2) 折半查找法：表中的元素必须以关键字的大小递增或递减的次序存放并且只能采用顺序存储方式存储。

两种方法的平均搜索长度分别如下：

(1) 顺序查找法：类似于顺序表查找成功时的评价方式 ASLsq=(n+1)/2。

(2) 折半查找法：利用二叉搜索树评价性能 ASLbin=((n+1)/n)*log2(n+1)-1。

7.2 二叉查找树

- 二叉查找树的结构
  - 对于线性表而言，每次插入和删除操作均需要对做大量的移动操作，性能差，二叉查找树是解决这个问题的一个很好的手段。

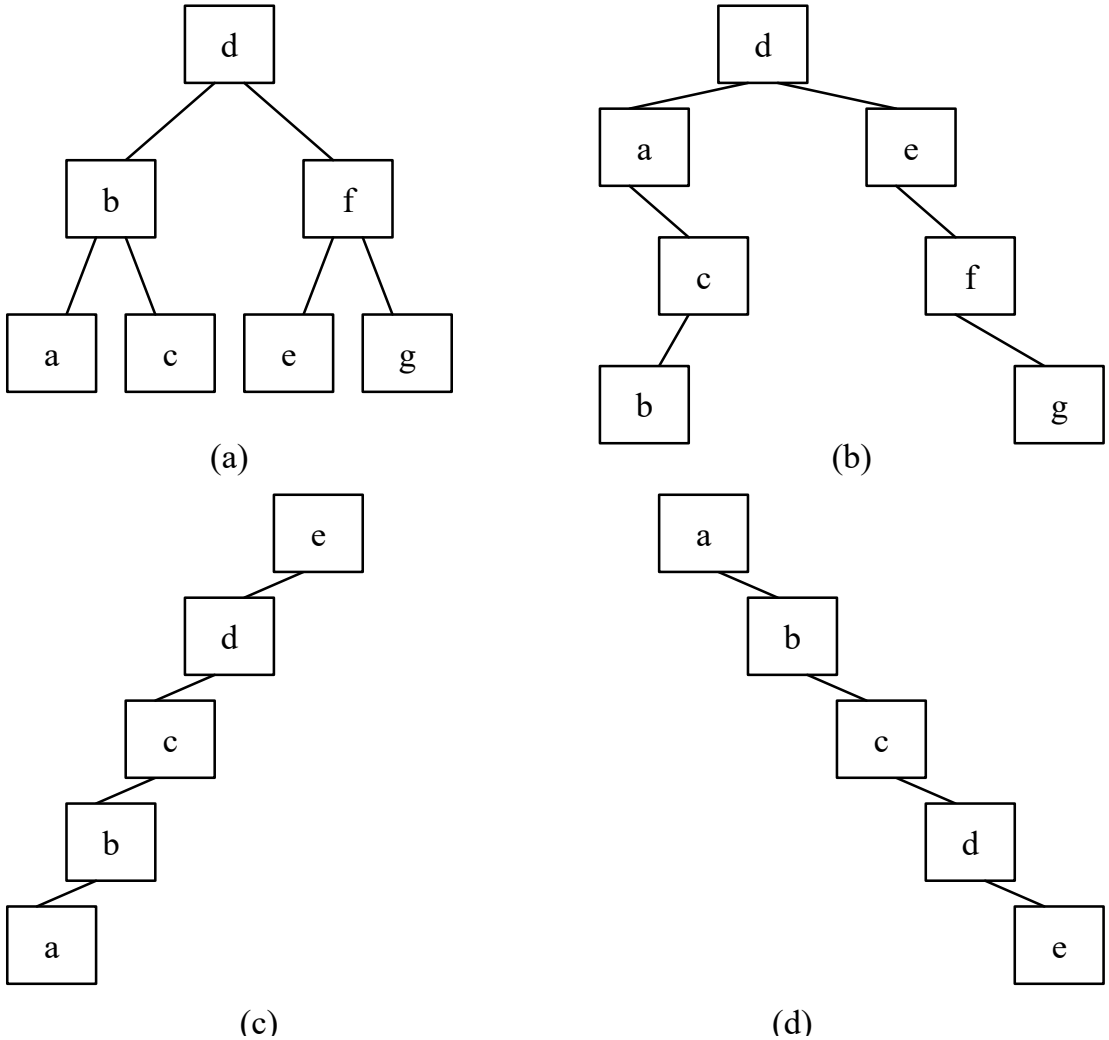
- 二叉查找树T是一棵二叉树T，它是或空，或满足下面三个条件：
- (1) 二叉查找树的每个结点包括三个属性：键值、左子结点和右子结点。
- (2) 如果树T的根结点的左子树非空，那么左子树中的所有结点的键值都小于T的根结点的键值；
- (3) 如果树T的根结点的右子树非空，那么右子树中的所有结点的键值都大于T的根结点的键值；
- (4) 树T的根结点的左右子树也都是查找树。

定义

二叉查找树或者是一棵空树，或者是具有下列性质的二叉树：

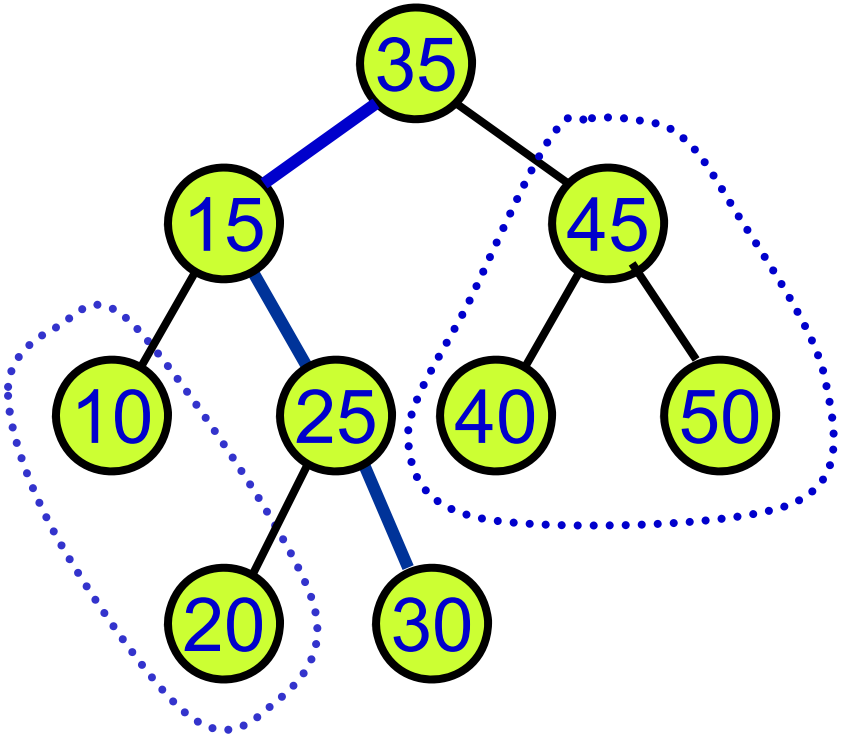
- 每个结点都有一个作为搜索依据的关键码 **(key)**，所有结点的关键码互不相同。
- 左子树（如果存在）上所有结点的关键码都小于根结点的关键码。
- 右子树（如果存在）上所有结点的关键码都大于根结点的关键码。
- 左子树和右子树也是二叉查找树。

图7.3.1 四棵二叉查找树



二叉查找树例

- 结点左子树上所有关键码小于结点关键码。
- 右子树上所有关键码大于结点关键码。



- 注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键码不一定小于路径上的结点的关键码。

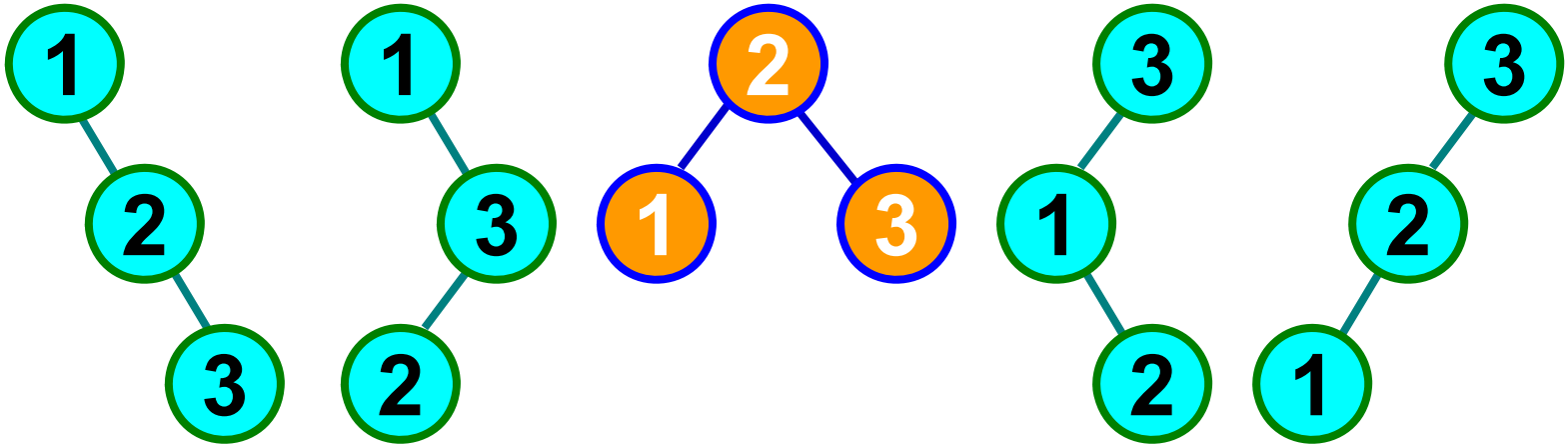
如果对一棵二叉查找树进行中序遍历，可以按从小到大的顺序，将各结点关键码排列起来，所以也称二叉查找树为二叉排序树。

*n*个结点的二叉查找树的数目

【例】3个结点的二叉查找树：

$$\frac{1}{3+1}C_{2\times 3}^{3}=\frac{1}{4}\times\frac{6\times 5\times 4}{3\times 2\times 1}=5$$

{123} {132} {213} {231} {312} {321}



二叉查找树结点的数据结构定义

```
template <class ElementType> class search_tree; //线性表类的前向引用声明
template <class ElementType> class Node { //边结点的类定义
  friend class search_tree <NameType, DistType>;
  ElementType data; //边上的权值
  Node <ElementType> *rightchild; //指向右结点的指针
  Node <ElementType> *leftchild; //指向左结点的指针
 public:
  Node( ) { } //构造函数
  Node(ElementType c) : dest(d), cost(c), link(NULL) { } //构造函数
};
template <class ElementType> class search_tree { //二叉搜索树的类定义
 private:
   Node root; //指向根结点
 public:
   search_tree(int size); //构造函数
   ~search_tree( ); //析构函数
   int treeEmpty( ) const { return length==0; } //判断二叉搜索树空否
   int length( ) { return VerticesNum; } //返回二叉搜索树中数据的数量
   Node search(ElementType element);
   //返回元素取值为element的数据项所在的位置
   ElementType getvalue(node n); //返回位置为n的元素的取值
   int insert(ElementType element);
   //向二叉搜索树中插入元素取值为element的数据项
   int delete(ElementType element);
   //二叉搜索树中删除元素取值为element的数据项
};
```

二叉查找树的类定义

```
#include <iostream.h>
template <class Type> class BST;
template <class Type> class BstNode <Type> : public BinTreeNode {
  friend class BST <Type>;
  protected:
    Type data;
    BstNode <Type> *leftChild, *rightChild;
  public:
    BstNode( ) : leftChild(NULL), rightChild(NULL) { }  //构造函数
    BstNode(const Type d, BstNode <Type> *
                    L=NULL, BstNode <Type> *R=NULL)
                      : data(d), leftChild(L), rightChild(R) { }
    void SetData(Type d) { data = d; }
    Type GetData( ) { return data; }
    ~BstNode( ) { }  //析构函数
};
```

```
template <class Type> class BST
                : public BinaryTree <Type> {
  private:
    BstNode <Type> *root;  //根指针
    Type RefValue;  //数据输入停止的标志
    void MakeEmpty(BstNode <Type> *&ptr);
    void Insert(Type x, BstNode <Type> *&ptr);
    //插入
    void Remove(Type x, BstNode <Type> *&ptr)
    //删除
    void PrintTree(BstNode <Type> *ptr) const;
    BstNode <Type> * Copy
         (const BstNode <Type> *ptr);  //复制
```

```
  BstNode <Type> * Search(Type x,
                BstNode <Type> *ptr);  //搜索
  BstNode <Type> * Min
                (BstNode <Type> *ptr) const;  //求最小
  BstNode <Type> * Max
                (BstNode <Type> *ptr) const;  //求最大
public:
  BST( ) : root(NULL) { }  //构造函数
  BST(Type value);  //构造函数
  ~BST( );  //析构函数
  const BST & operator = (const BST &Value);
```

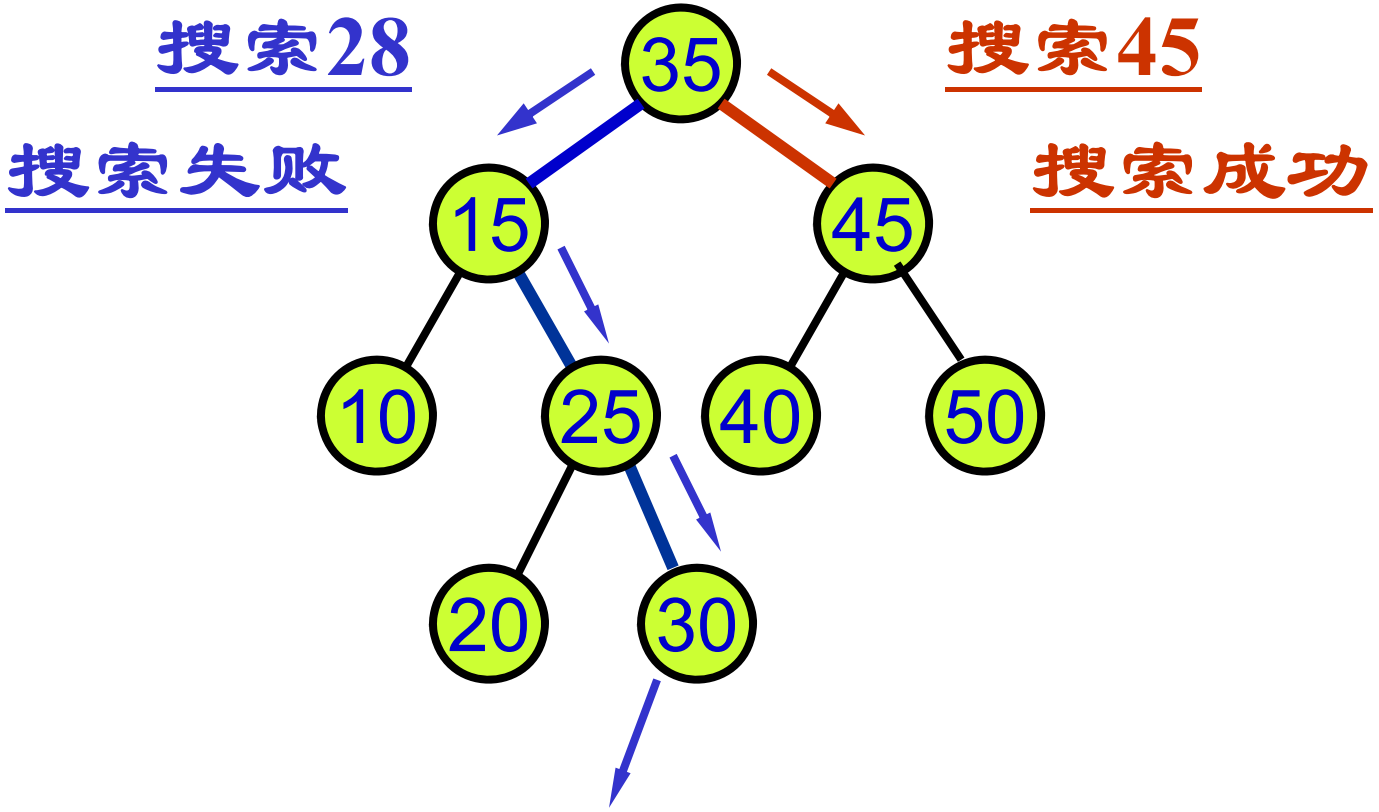
```
    void MakeEmpty( )
        { MakeEmpty(root);  root=NULL; }
    void PrintTree( ) const { PrintTree(root); }
    bool Search(Type x)  //搜索元素
        { return Search(x, root)!=NULL; }
    Type Min( ) { return Min(root)->data; }
    Type Max( ) { return Max(root)->data; }
    bool Insert(Type x) { Insert(x, root); }
    //插入新元素
    bool Remove(Type x) { Remove(x, root); }
    //删除含x的结点
};
```

7.3.2 二叉查找树上的搜索

- 二叉查找树中有一个非常好的特性，就是每个结点l左子树中点的键值均比l的键值小，右子树中结点的键值均比其大。
- 二叉查找树t中找出包含给定键值a的结点算法：
  - (1)如果t为空，那么查找失败．算法结束；
  - (2)如果t->data等于a,那么查找成功，算法结束；
  - (3)如果a<t->data,那么t->leftchild=>t，继续在其左子树中搜索；否则，t->rightchild=>t，继续在其右子树中搜索。

在二叉查找树上进行搜索，是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。它可以是一个递归的过程。

- 假设想要在二叉查找树中搜索关键码为x的元素，搜索过程从根结点开始。
- 如果根指针为NULL，则搜索不成功；否则用给定值x与根结点的关键码进行比较：
  - 如果给定值等于根结点的关键码，则搜索成功。
  - 如果给定值小于根结点的关键码，则继续递归搜索根结点的左子树；
  - 否则，递归搜索根结点的右子树。


搜索28
搜索失败
35
搜索45
搜索成功
15
45
10
25
40
50
20
30

算法7-5 二叉查找树上的搜索算法

```
template <class ElementType> Node Search_tree
<Element Type> :: search(Element n) {
    Node **p_q;
    *p_q=root;
    while ((*p_q!=NULL) && (*p_q)->data!=n)) {
        if (a<getvalue(*p_q))  *p_q=(*p_q)->leftchild;
       else *p_q=(*p_q)->rightchild;
    }
    if ((*p_q==NULL)  return -1;
    if ((*p_q)->data==n)  return p_q;
}
```

- 对于成功的查找，从n个结点的二叉查找树T中找出结点k所需的比较次数，是从根结点到结点k的树枝长度λk加1，因此可以得到：
  - 搜索的最大代价=max{1+λk|树T中的k}
  - 搜索的平均代价=Σp(k) (1+λk)
    - 其中，p(k)是结点k被搜索概率。如果所有结点被搜索的概率都相等，那么对树T中的任一结点k都有p(k)=1/n。

$$\text{平均查询代价} = \frac{1}{n}\sum_{T\text{中的k}}(1+\lambda(k))$$

- 显然，当树中结点尽量靠近树根时，平均查询代价最小；而当查找树退化成链表时，平均查询代价最大。

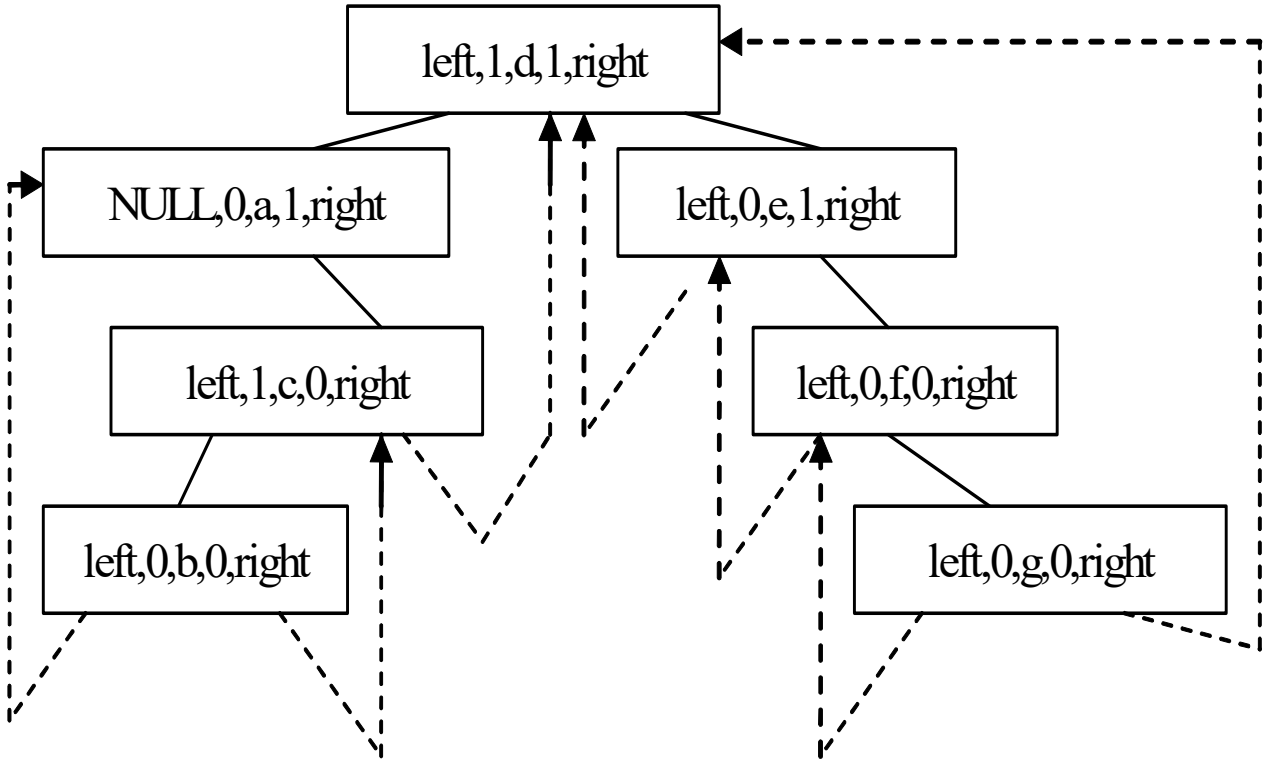
```
template <class Type>
        BstNode <Type> * BST<Type> ::
Search(Type x, BstNode <Type> *ptr) const {
//二叉查找树的递归的搜索算法
   if (ptr==NULL)  return NULL;  //搜索失败
   else if (x<ptr->data)  //在左子树搜索
           return Search(x, ptr->leftChild);
   else if (x>ptr->data)  //在右子树搜索
           return Search(x, ptr->rightChild);
   else return ptr;  //相等，搜索成功
}
```

```
template <class Type>
            BstNode <Type> * BST <Type> ::
Search(Type x, BstNode <Type> *ptr) const {
//二叉查找树的迭代的搜索算法
   if (ptr!=NULL) {
      BstNode <Type> *temp=ptr;  //从根搜索
      while (temp!=NULL) {
           if (temp->data==x)  return temp;
           if (temp->data<x)
             temp=temp->rightChild;  //右子树
           else temp=temp->leftChild;  //左子树  }
   }
   return NULL;  //搜索失败
}
```

7.3.3 基于二叉查找树的遍历

- 扩展的二叉查找树每个结点的结构在原有了左子结点、键值、右子结点三个属性以外，有增加了左子结点标签，右子结点标签两个属性。
  - 左子结点标签的含义是，如果左子结点是其树中的左子结点，则标签为**1**，否则为**0**，如果为**0**，则该指针指向比当前结点小一个的结点。
  - 右子结点标签的含义是，如果右子结点是其树中的右子结点，则标签为**1**，否则为**0**，如果为**0**，则指向比当前结点大一个的结点。
- 下图就是一个扩展的二叉查找树，其中虚线就是指向其排序上的前一个和后一个的结点。

图7.3.2 扩充的二叉查找树



- 二叉查找树构造好之后，可以对二叉查找树进行扩展，将原来指向**NULL**的指针进行修正构造扩展的二叉查找树，算法的基本想法是基于遍历的算法进行构造，在中序遍历的过程中，记住结点之间的序关系，从而对指向**NULL**的结点进行修正。

- 扩展的二叉查找树构造完成以后，就可以方便地对扩展的二叉查找树进行遍历。
- 下面介绍两个实现遍历的基本函数，**Next**和**previous**，基于这两个函数就可以实现对二叉查找树中数据集合的从小到大或从大到小的遍历。

```
template <class ElementType> Node Search_tree <Element Type> ::
Next(node) {
    Node tmp;
    if (node->rightid==0)  return node->right;
    else {
        tmp=node->right;  while (tmp->left!=NULL)  tmp=tmp->left;
        return tmp;
    }
}
```

```
template <class ElementType> Node Search_tree <Element Type> ::
previous(node) {
    Node tmp;
    if (node->leftid==0)  return node->left;
    else  {
        tmp=node->left;  while (tmp->right!= NULL)  tmp=tmp->right;
        return tmp;
    }
}
```

- 基于Next函数，我们可以实现对数据从小到大的遍历。
- **template <class ElementType>** Node Search_tree **<Element Type> ::** Travers(root)
- {
- Node tmp; Tmp=root;
- **while** (tmp->leftid!=0) tmp=tmp->left;
- **printf**(tmp->data);
- **while** (tmp!=NULL) {
- tmp=next(tmp);
- **if** (tmp!=NULL) **printf**(tmp->data);
- }
- }
- 同样也可以基于**previous**实现对数据从大到小的遍历。

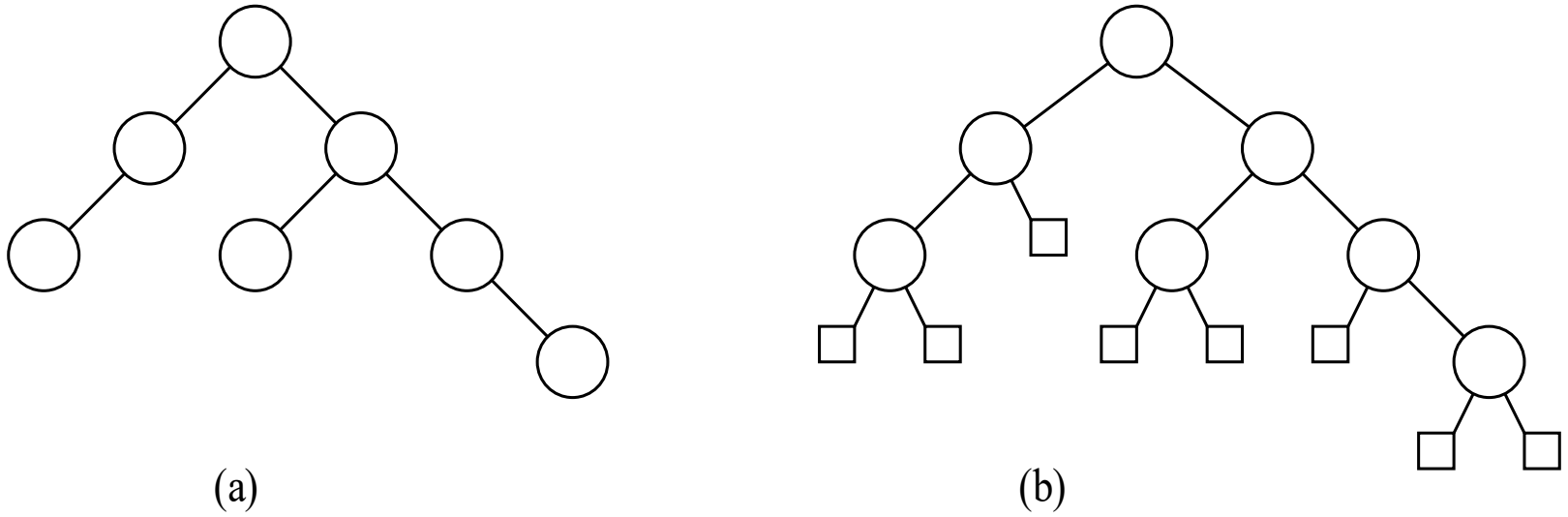
7.3.4 最优二叉搜索树（略）

- 二叉搜索树的所有结点搜索的平均代价
  - 二叉搜索树的搜索成功的平均搜索代价 **$=\Sigma p(k)(1+\lambda k)$。**
    - 其中，**$p(k)$**为结点**k**被搜索到的概率，**λk**为**k**到其根结点的路径长度。当所有的结点被搜索到的概率是相同的情况，即**$p(k)=1/n$**时，上式可改写成：
      - **$1/n\Sigma(1+\lambda k)$**
      - 这里假定每次查找都是成功的查找．所谓最优搜索树就是对于一个特定的数据集计算复杂性最小的搜索树。

外部结点

- 在给定的查找树中，在结点（包括叶子结点)的每个空指针上都挂上一个附加结点，称为外部结点，称原来树中的结点为内部结点。
  - 在画图时，用圆圈表示内部结点，而用方框表示外部结点。如图**7.3.3**。
    - 称由根到所有外部结点的路径长度总和为二叉树的外部路径长度；
    - 称由根到所有内部结点的路径长度总和为二叉树的内部路径长度。
  - 具有**n**个结点的二叉树，经过扩充后，产生**n+1**个外部结点。
    - 对不在二叉搜索树中的结点的查找总是终止于一个外部结点，外部结点代表不成功的查找，故外部结点也称为失败结点。

扩充二叉树



(a) (b)

- 图7.3.3 (b)由(a)添加外部结点所得到的扩充二叉树。

内外部路径长度

- 一棵具有n个内部结点的二叉树的内部和外部路径长度之间的关系可用公式En=In+2n (7.3.1)表示。（用归纳法证明）
  - 当n=1时，由于只有一个内部结点，所以I1=0，E1=2，满足(7.3.1)式。
  - 假设所有具有n个内部结点的扩充二叉树都满足(7.3.1)式。
  - 对于具有n个内部结点的任一棵扩充二叉树，把树中一个路径长度L的外部结点换成一个内部结点，且在此结点下附加两个路径长为(L+1)的外部结点。这样原来的树就变成具有(n+1)个内部结点的扩充二叉树，且满足：
    - $I_{n+1}=I_n+L$ (7.3.2)；$E_{n+1}=E_n-L+2(L+1)$ (7.3.3)
    - 由(7.3.2)式，得$I_n=I_{n+1}-L$ (7.3.4)；由(7.3.3)式，得$E_{n+1}=E_n+L+2$ (7.3.5)
    - 由(7.3.5)、(7.3.1)和(7.3.4)式，得
    - $E_{n+1}=I_n+2n+L+2=I_{n+1}-L+2n+L+2=I_{n+1}+2n+2=I_{n+1}+2(n+1)$
- 从公式(7.3.1)，可以得出：具有n个结点的所有二叉树，最有最大（或最小）内部路径长度的二叉树，也一定具有最大（或最小）外部路径长度的二叉树。

- 当二叉树退化成线性表时，内部路径长度最大，长度为 $I_n = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

- 为得到最小内部路径长度的二叉树，内部结点应尽量靠近根。二叉树中，根结点路径长度为0，路径长度1的结点最多两个，路径长度2的结点最多4个，……具有n个内部结点的二叉树的最小内部路径长度为：

$$I_n = 1 \bullet 0 + 2 \bullet 1 + 4 \bullet 2 + 8 \bullet 3 + ... = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

- 如果查找树由内部结点序列$a_1<a_2<\ldots<a_n$构成，(n+1)个外部结点为$b_0, b_1, \ldots, b_n$。当查找值为x的结点时，如果$a_i<x<a_{i+1}$，那么在b_i上终止其查找；如果$x<a_0$，那么在b_0终止查找；如果$x>a_n$，那么在b_n终止其查找。

- 假设p_i $(1\le i\le n)$是a_i的查找概率，而q_i $(1\le j\le n)$是b_j的查找概率，那么成功查找所需的平均比较次数为

$$\sum_{i=1}^{n} p_i(1 + a_i\text{的路径长度}) = \sum_{i=1}^{n} p_i(1 + \lambda a_i)$$

- 而不成功的查找所需的平均比较次数为

$$\sum_{i=1}^{n} q_i(b_i\text{的路径长度}) = \sum_{i=1}^{n} q_i \times \lambda b_i$$

因此，最优二叉搜索树的搜索成功的平均搜索长度和搜索不成功的平均搜索长度分别为:

$$ASL_{succ} = \sum_{i=1}^{n} \left(\lfloor \log_2 i \rfloor + 1 \right).$$

$$ASL_{unsucc} = \sum_{i=n+1}^{2n+1} \lfloor \log_2 i \rfloor.$$

- 再考虑不成功的查找，所以查找的全部时间或代价为：

$$\sum_{i=1}^{n} p_i(1+\lambda a_i)+\sum_{i=1}^{n} q_i \times \lambda b_i \quad (7.3.6)$$

- 由于所有查找或者成功，或者不成功，两者必择其一。因此，应有：

$$\sum_{i=1}^{n} p_i+\sum_{i=1}^{n} q_i = 1$$

- 由结点a_1，a_2，…，a_n构成的所有可能的查找树中，使(7.3.6)式取最小值的，就是a_1，a_2，…，a_n的最佳查找树。

- 如果成功和不成功查找都具有相同的查找概率，此时的平均查找时间为：

$$AVG=\sum_{i=1}^{n}p_i(1+\lambda a_i)+\sum_{i=1}^{n}q_i\times\lambda b_i=\frac{1}{2n+1}\left[\sum_{i=1}^{n}(1+\lambda a_i)+\sum_{i=1}^{n}\lambda b_i\right]$$

$$=\frac{1}{2n+1}(I_n+n+E_n)=\frac{1}{2n+1}(I_n+n+I_n+2n)=\frac{1}{2n+1}(2I_n+3n)$$

  - 所以，只要$\mathbf{I_n}$最小，**AVG**就达到最小。
- 从上面讨论可知，满的查找树能使$\mathbf{I_n}$最小，其值为：

$$I_n=\sum_{i=1}^{n}\lfloor\log_2 i\rfloor\le n\log_2 n=O(n\log_2 n)$$

- 所以，

$$\text{平均代价}=\frac{1}{2n+1}(2I_n+3n)=O(\log_2 n)$$

- 再考虑查找概率不相等的情况，给定n个内部结点$a_1, a_2, \ldots, a_n$的查找概率$p_1, p_2, \ldots, p_n$，(n+1)个外部结点的查找概率$q_0, q_1, \ldots, q_n$。
  - 一种解决的办法是构造出所有可能的查找树，然后挑选其中代价最小的树。
    - 但是，随着n的增加，不同的查找树的个数迅速增加，因此计算复杂性非常高。
    - 如果根据最佳二叉搜索树树的性质，那么就能找到一种相当有效的构造算法。

- 首先说明一下，权w、使用频率f与使用概率p之间的关系。
  - 概率是正的小数，有时使用不方便，故引进正整数表示的权w、使用频率f 。
  - 权w和使用频率f相比，权w的含义更广泛。
    - 在本章中，它们含义相同，就是使用次数。
  - 权w、使用频率f与使用概率p有其内在的联系，它们可以进行换算。
    - 如结点$a_1, a_2, \ldots, a_n$的使用频率分别为$f_1, f_2, \ldots, f_n$，则结点a_i $(1\le i\le n)$的使用概率为$p_i=f_i/\sum f_i$；反之，如已知$a_1, a_2, \ldots, a_n$的使用概率分别为$p_1, p_2, \ldots, p_n$，又设每个概率精确到小数点t位，则$f_i=p_i*10^t$ $(1\le i\le n)$，每个f_i都是整数。
  - 权w和使用概率p之间的关系和使用频率f与使用概率p之间的关系相似。

- 用T_{ij}表示a_{i+1}，…，a_j (i<j)组成的一棵最佳查找树，约定：当0≤i≤n时；T_{ij}为空树；当i>j时，T_{ij}无定义。用c_{ij}表示T_{ij}的代价，c_{ii}=0。
  - 用r_{ij}表示T_{ij}的根；令$w_{ij}=q_i+\Sigma c_{jk}=i+1(p_k+q_k)$，表示$T_{ij}$的权。由定义，有$r_{ij}=0$，$w_{ii}=q_i$ (1≤i≤n)。
  - 如果$r_{ij}=k$ (i<k≤j)（见图7.3.4）。L为根a_k的左子树，由a_{i+1}，…，a_{k-1}组成；R是根a_k的右子树，由a_{k+1}，…，a_j组成，T_{ij}的代价c_{ij}为：

$$c_{ij} = p_k + \cos t(L) + w(L) + \cos t(R) + w(R)$$

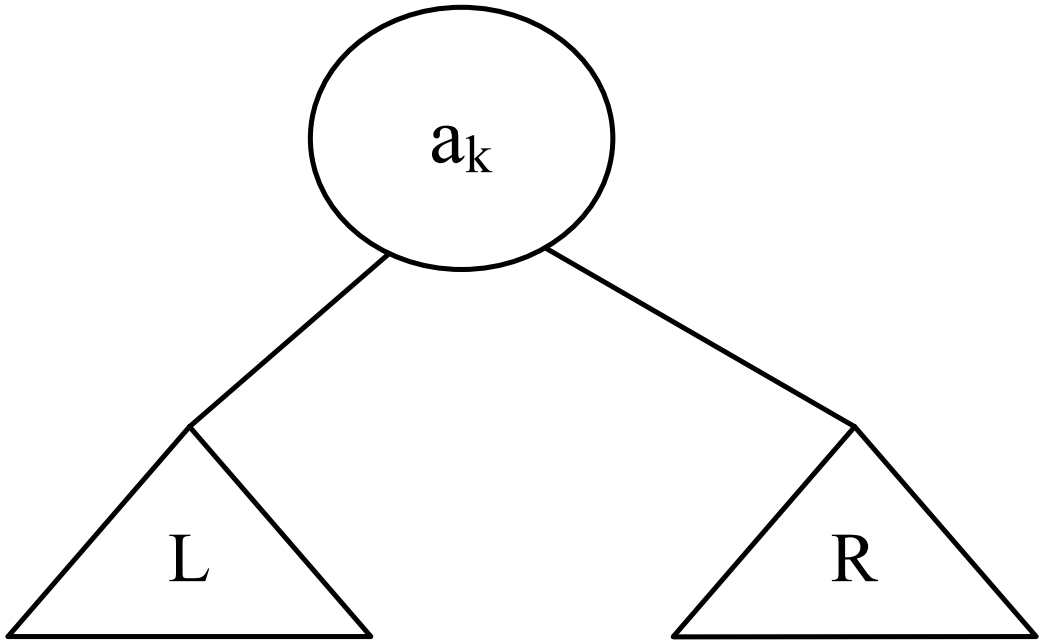
  - 其中，cost(L)和cost(R)分别是左子树L和右子树R的代价，而

$$w(L) = w(T_{ik-1}) = w_{ik-1}$$
$$w(R) = w(T_{kj}) = w_{kj}$$

- 因此，有 $c_{ij} = p_k + w_{ik-1} + w_{kj} + \cos t(L) + \cos t(R)$

- 所以有 $c_{ij} = w_{ij} + \cos t(L) + \cos t(R)$ (7.3.7)

图7.3.4 最佳搜索树的结构



- 最佳查找树具有这样的性质：一棵最佳查找树的所有子树都是最佳查找树。利用这个性质，可把(7.3.7)改写成：

$$c_{ij} = w_{ij} + c_{ik-1} + c_{kj} \quad \textbf{(7.3.8)}$$

- 由于T_{ij}是最佳查找树，从(7.3.8)可得出$r_{ij}=k$必定满足如下关系：

$$w_{ij} + c_{ik-1} + c_{kj} = \min_{i<t\le j}\{w_{ij} + c_{it-1} + c_{tj}\}$$

- 因为

$$w_{ij} = q_i + \sum_{k=i+1}^{j}(p_k + q_k) = q_i + \sum_{k=i+1}^{j-1}(p_k + q_k) + p_j + q_j$$

- 所以
-

$$w_{ij} = w_{ij-1} + p_j + q_j \quad \textbf{(7.3.10)}$$

- 关系式(7.3.8)和(7.3.10)给出了从$T_{ij}=\Phi$和$c_{ii}=0$获得Ton和con的一种方法。此方法的基本原理就是从已形成的较小的最佳查找树出发，系统地寻找越来越大的最佳查找树。

说明构造最佳查找树过程的例子

- 设n=4且$(a_1, a_2, a_3, a_4)=(a, b, c, d)$，取$(p_1, p_2, p_3, p_4)=(3, 3, 1, 1)$, $(q_0, q_1, q_2, q_3, q_4)=(2, 3, 1, 1, 1)$。
  - 为了计算方便，p和q都已乘上16。
- 开始时，取$w_{ii}=q_i$，$c_{ii}=0$和$r_{ii}=0$ $(0<=i<=4)$。
- 使用(7.3.8)和(7.3.10)式便可得到有关的子树，从而构造出所需的最佳查找树。
- 在下面的叙述中，为书写方便，把T_{ij}的根结点的左和右子树分别记为T^l_{ij}和T^r_{ij}。

- **当j-i=1时：**
  - 因为 $w_{01}=w_{00}+p_1+q_1=8$，$c_{01}=w_{01}+c_{00}+c_{11}=8$　所以 $r_{01}=1$　$T^l_{01}=T_{00}$　$T^r_{01}=T_{11}$
  - 因为 $w_{12}=w_{11}+p_2+q_2=7$，$c_{12}=w_{12}+c_{11}+c_{22}=7$　所以 $r_{12}=2$　$T^l_{12}=T_{11}$　$T^r_{12}=T_{22}$
  - 因为 $w_{23}=w_{22}+p_3+q_3=3$，$c_{23}=w_{23}+c_{22}+c_{33}=3$　所以 $r_{23}=3$　$T^l_{23}=T_{22}$　$T^r_{23}=T_{33}$
  - 因为 $w_{34}=w_{33}+p_4+q_4=3$，$c_{34}=w_{34}+c_{33}+c_{44}=3$　所以 $r_{34}=4$　$T^l_{23}=T_{22}$　$T^r_{23}=T_{33}$

- 当j-i=2时，因为 $w_{02} = w_{01} + p_2 + q_2 = 12$

  $c_{02} = w_{02} + \min\{c_{00} + c_{12},\ c_{01} + c_{22}\} = 19$

  $c_{00} + c_{12} < c_{01} + c_{22}$

- 所以 $r_{02} = 1$ $\quad T^l_{02} = T_{00}$ $\quad T^r_{02} = T_{12}$
- 因为 $w_{13} = w_{12} + p_3 + q_3 = 9$

  $c_{13} = w_{13} + \min\{c_{11} + c_{23},\ c_{12} + c_{33}\} = 12$

  $c_{11} + c_{23} < c_{12} + c_{33}$
- 所以 $r_{13} = 2$ $\quad T^l_{13} = T_{11}$ $\quad T^r_{13} = T_{23}$
- 或 $r_{24} = 4$ $\quad T^l_{24} = T_{23}$ $\quad T^r_{24} = T_{44}$

- 当j-i=3时，因为 $w_{03} = w_{02} + p_3 + q_3 = 14$

  $c_{03} = w_{03} + \min\{c_{00} + c_{13},\ c_{01} + c_{23},\ c_{02} + c_{33}\} = 25$

  $c_{01} + c_{23} < c_{00} + c_{13},\ c_{01} + c_{23} < c_{02} + c_{33}$

- 所以 $r_{03} = 2 \quad \mathrm{T}_{03}^{l} = \mathrm{T}_{01} \quad \mathrm{T}_{03}^{r} = \mathrm{T}_{23}$

- 因为 $w_{14} = w_{13} + p_4 + q_4 = 11$

  $c_{14} = w_{14} + \min\{c_{11} + c_{24},\ c_{12} + c_{34},\ c_{13} + c_{44}\} = 19$

  $c_{11} + c_{22} < c_{12} + c_{34},\ c_{11} + c_{24} < c_{13} + c_{44}$

- 所以 $r_{14} = 2 \quad \mathrm{T}_{14}^{l} = \mathrm{T}_{11} \quad \mathrm{T}_{14}^{r} = \mathrm{T}_{24}$

- 当**j-i=4**时，因为

$$w_{04} = w_{03} + p_4 + q_4 = 16$$

$$c_{04} = w_{04} + \min\{c_{00} + c_{14},\ c_{01} + c_{24},\ c_{02} + c_{34},\ c_{03} + c_{44}\} = 32$$

- $c_{01} + c_{24}$ 都小于 $c_{00} + c_{14},\ c_{02} + c_{34},\ c_{03} + c_{44}$

- 所以 $r_{04} = 2 \quad \mathrm{T}_{04}^{l} = \mathrm{T}_{01} \quad \mathrm{T}_{04}^{r} = \mathrm{T}_{24}$

图7.3.5

- 在上面的计算过程中，可以构造出图7.3.5中各棵树。

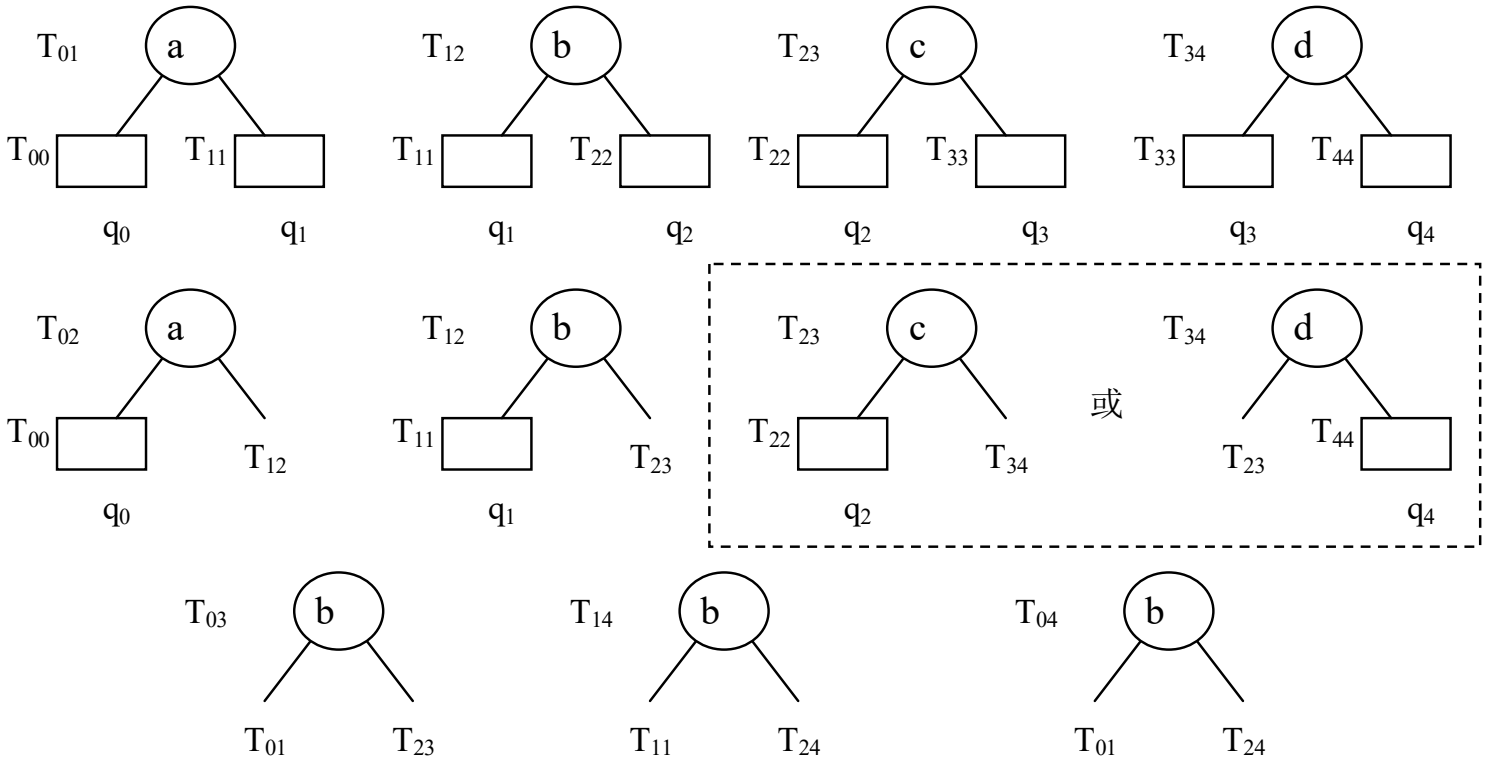
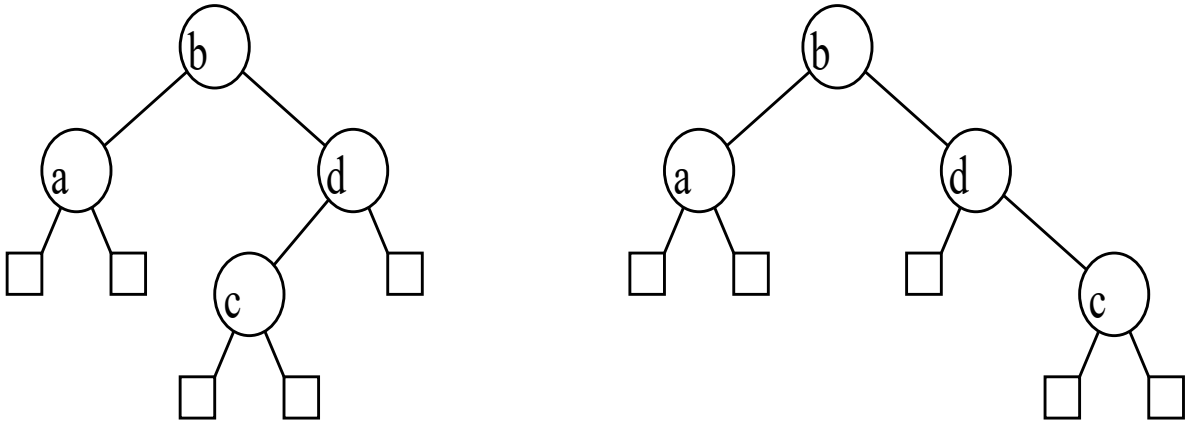


图7.3.6

- 依据根和子树之间的关系，把它们装配起来，可以得到图7.3.6中的两棵最佳查找树，它们的根结点都是a2，最小代价都为32。



计算代价c的复杂性

- 在计算c_{ij}时，按j-i=1, 2, …, n的次序进行。当j-i=1时，计算$c_{01},c_{12},\ldots,c_{(n-1)n}$；当j-i=2时，就是$c_{02}, c_{13}, \ldots, c_{(n-2)n}$；等等。
- 当j-i=m时，就需计算(n-m+1)个c_{ij}，而计算c_{ij}时又需要从m个数中找出其最小者。因此，所需的时间为
- $(n-m+1)m=nm-m^2+m$
- 而且m是从1取到n，故总的时间为$O(n^3)$。

算法基本思路

- 初始化
  - **W[i][i], C [i][i], W[i][i+1], C[i][i+1], R[i][i+1]**
- 循环（内部结点数从**2**增到**n**，步长为**1**）
  - 循环（依次对当前规模的每棵子树）
    - 循环（依次让本子树内每个内部结点为根）
      - {根据之前算好的较小规模子树的代价，推算当前子树的代价，从中寻找最小代价子树，保存，用于后续推算更大规模子树的代价}
- 最后得到最优二叉搜索树。

7.3.5 动态二叉搜索树

- 非平衡二叉搜索树
  - 前面介绍了静态二叉搜索树的基本结构，在静态二叉搜索树中可以直接通过插入和删除操作实现对数据的动态操作。

二叉搜索树的插入操作

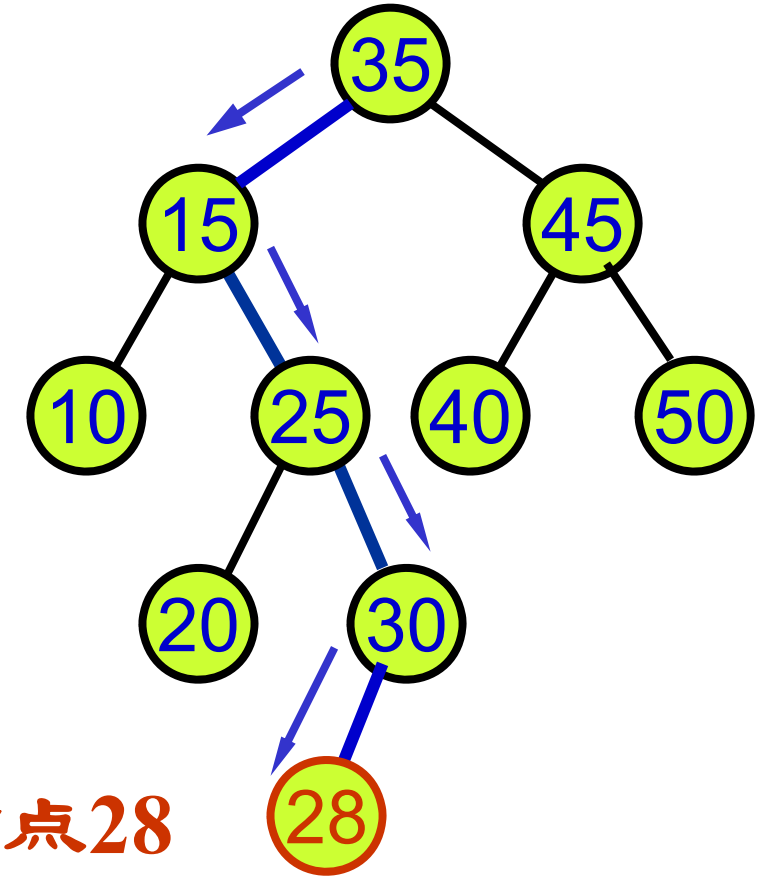
- 下面用C函数实现在给定的查找树T中插入一个键值为a的结点的操作。
  - 首先，调用search( )，如果键值为a的结点在树T中，则不插入；
  - 如果键值为a的结点不在树T中，则search函数最终返回与a相邻的结点。
    - 根据a同其相邻结点的大小比，获得适当位置
    - 插入操作插入的位置总是在叶结点。如果插入成功，那么insert( )返回0，否则，insert( )返回1。

```
template <class ElementType> int Search_tree <Element Type> ::
insert(NODE **p_t, Element a) {
    NODE *p,*q,*r;
    Search(*p_t, a, &p, &q);
    if (q!=NULL)  return 1;  //找到就直接返回
    r=(NODE *) malloc (sizeof(NODE));
    r->data=a;  r->leftchild=NULL;  r->rightchild=NULL;
    if (p==NULL)  *p_t=r;  //查找树为空
    else if (p->data>a)  p->leftchild=r;  else p->rightchild=r;
    return 0;
}
```

- 如果指针变量t指向给定的查找树T，a是插入结点的值，那么可用下面的语句“I=insert(&t, a)”实现插入。
- 在insert( )函数中，采用指针变量的指针作为参数是为了在对空的查找树进行插入时，使指针变量t能指向新的根结点。

二叉搜索树的插入

每次结点的插入，都要从根结点出发搜索插入位置，然后把新结点作为叶结点插入。



插入新结点28

- 为了向二叉搜索树中插入一个新元素，必须先检查这个元素是否在树中已经存在。
- 在插入之前，先使用搜索算法在树中检查要插入元素有还是没有。
  - 搜索成功

    树中已有这个元素，不再插入。
  - 搜索不成功

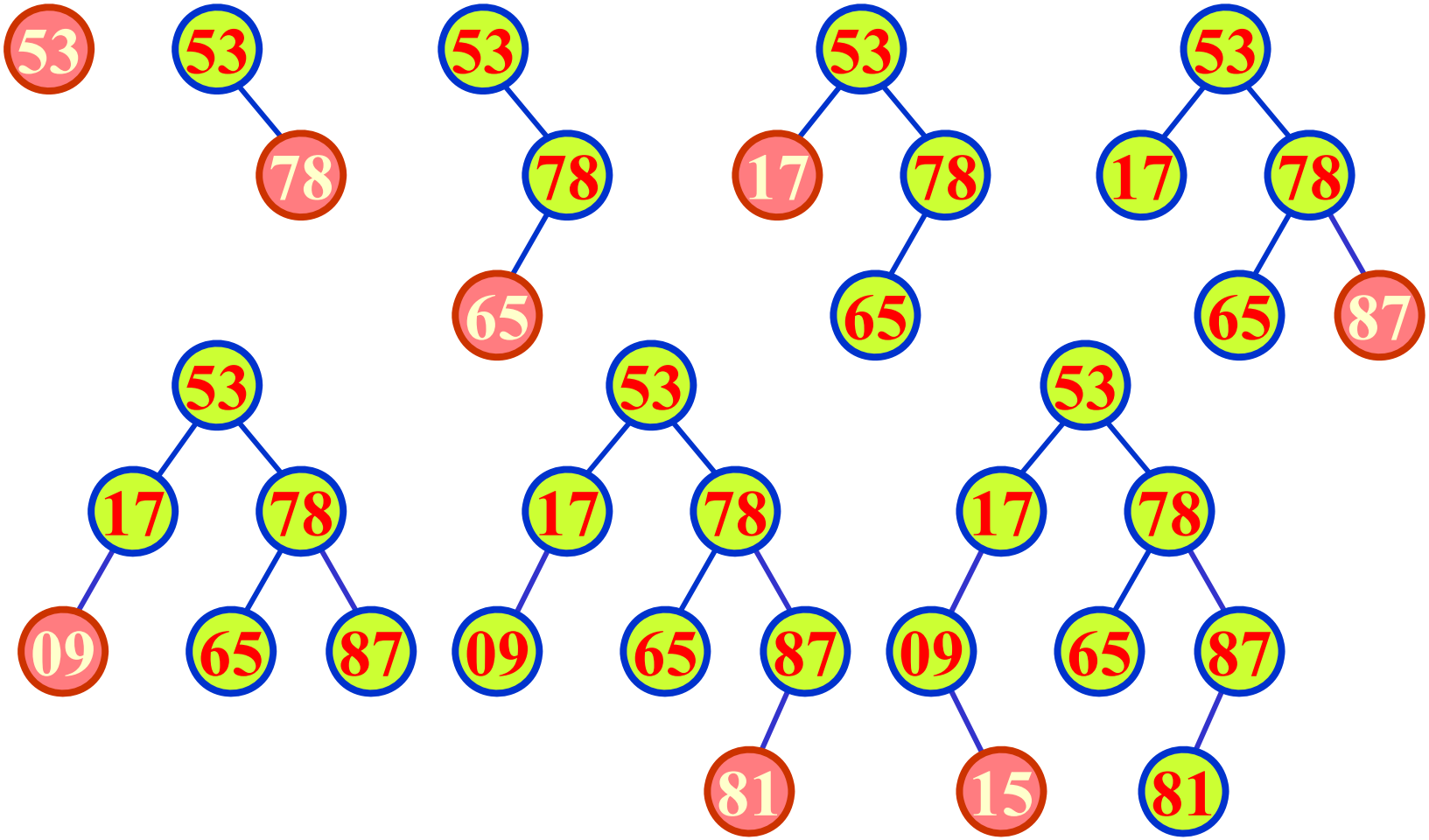
    树中原来没有关键码等于给定值的结点，把新元素加到搜索操作停止的地方。

递归的二叉搜索树插入算法

```
template <class Type> bool BST <Type> ::
Insert(Type x, BstNode <Type> *&ptr) {
    if (ptr==NULL) {  //空二叉树
       ptr=new BstNode <Type> (x);  //创建结点
       if (ptr==NULL)
         { cout<<"存储不足"<<endl;  exit(1); }
       return true;
    }
    else if (x<ptr->data)  //在左子树插入
              Insert(x, ptr->leftChild);
    else if (x>ptr->data)  //在右子树插入
              Insert(x, ptr->rightChild);
    else return false;
}
```

输入数据建立二叉搜索树的过程

{53, 78, 65, 17, 87, 09, 81, 15}



```
template <class Type> BST <Type> ::
BST(Type value) {
//输入数据，建立二叉搜索树。
//RefValue是输入结束标志
//应取不可能在输入序列中出现的值
//例如输入序列的值都是正整数时，取为0或负数
    Type x;
    root=NULL;  RefValue=value;
    cin>>x;
    while (x!=RefValue)
        { Insert(x, root);  cin>>x; }
}
```

二叉搜索树的删除操作

- 删除结点的算法思路如下：
  - 首先调用**search( )**确定被删结点在树中的位置。
  - 如果被删除结点不在树中，则删除结束。
  - 如果删除结点**n**在树中，
    - 如果**n**是根结点，那么
      - 若**n**无左子结点，则用**n**的右子树作为删除后的树
      - 否则用**n**的左子结点作为根，同时把**n**的右子树作为**n**的左子树中序最后一个结点的右子树。
    - 如果**n**不是根结点，那么
      - 若**n**无左子结点，则
        - » 如果**n**是它父结点**p**的左子结点，那么把**n**的右子树作为**p**的左子树。
        - » 如果**n**是它父结点**p**的右子结点，那么把**n**的右子树作为**p**的右子树。
      - 否则把**n**的右子树作为它父结点**p**的左子树按中序最后一个结点的右子树。同时，
        - » 如果**n**是**p**的左子结点，那么**n**的左子树作为**p**的左子树。
        - » 如果**n**是**p**的右子结点，那么**n**的左子树作为**p**的右子树。
  - 回收被删除结点的存储单元，算法结束。

```
template <class ElementType> int Search_tree <Element Type> ::
delete(Element a) {
    Node *p, *q, *r;
    Search(*p_t, a, &p, &q);
    if (q==NULL)  return 1;
    if (p==NULL)
      if (q->leftchild==NULL)  *p_t=q->rightchild;
      else {
        r=q->leftchild;  while (r->righthild!=NULL)  r=r->rightchild;
        r->rightchild=q->rightchild;  *p_t=q->leftchild;  }
   else if (q->leftchild==NULL) {
          if (q==p->leftchild)  p->leftchild=q->rightchild;
          else p->rightchild=q->rightchild;  }
   else {
          r=q->leftchild;  while (r->rightchild!=NULL)  r=r->rightchild;
          r->rightchiId=q->rightchild;
          if (q==p->leftchild)  p->leftchild=q->leftchild;
          else p->rightchild=q->1eftchild;  }
   free(q);
   return(0);
}
```

- 如果指针t指向给定的查找树T，a是所要删除的结点的键值，那么可用“I=delete(&t, a);”实现删除。
- 在delete()函数中'采用指针的指针作为参数是为了在删除根结点后，使指针t能指向新的根结点。

图7-7 删除a,e,d,c,b的过程(1)

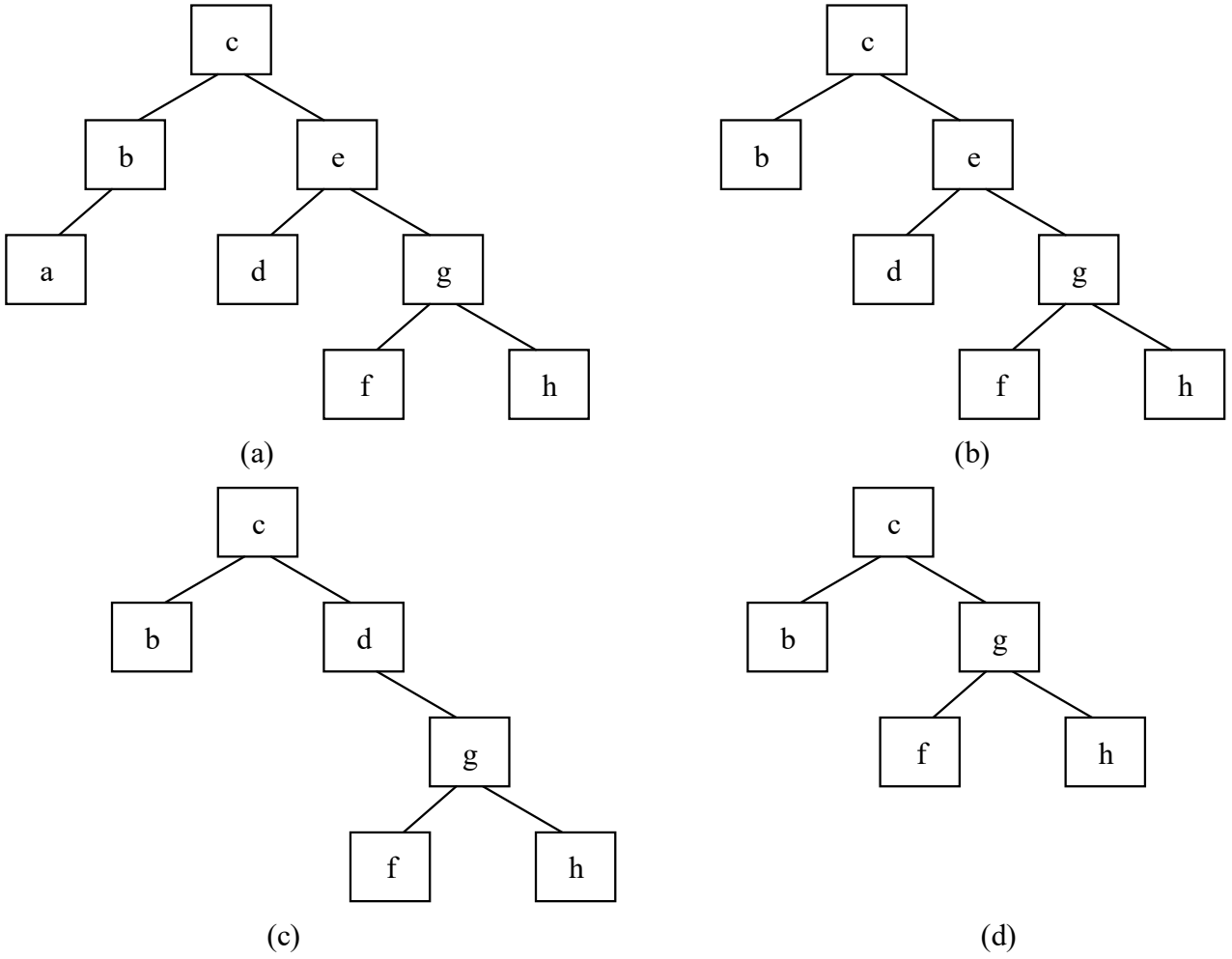
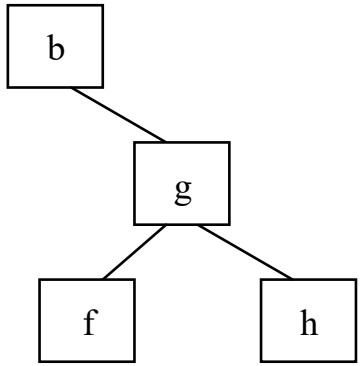
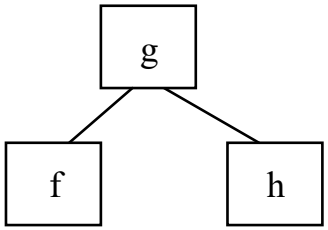


图7-7 删除a,e,d,c,b的过程(2)



(e)

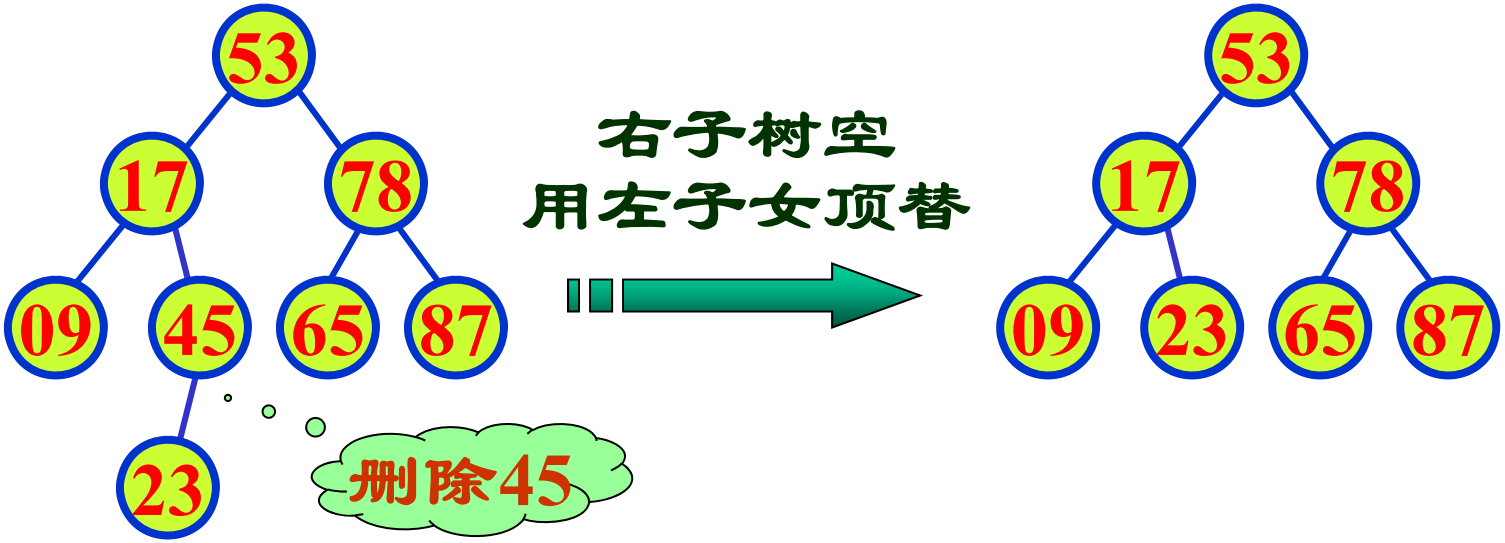


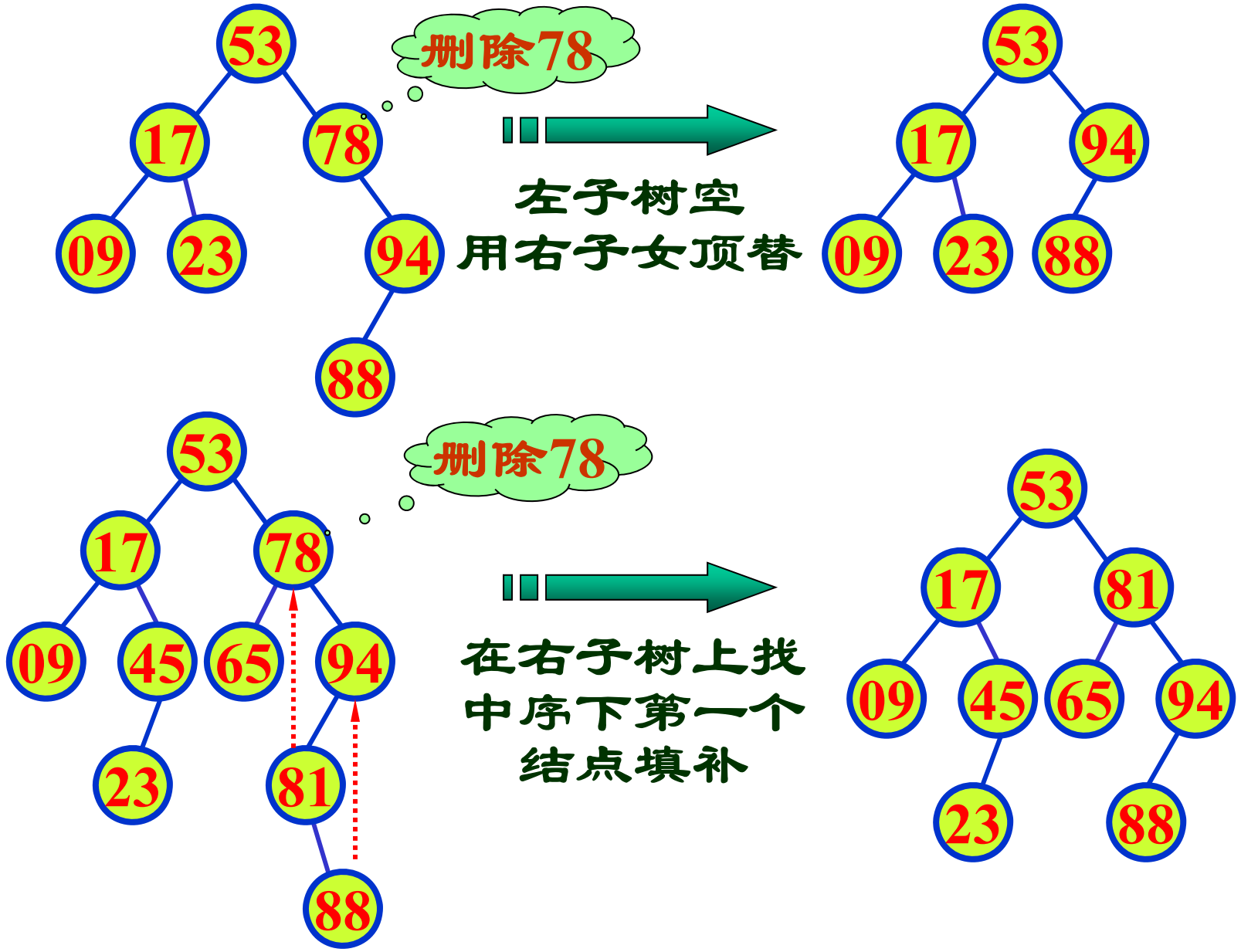
(f)

二叉搜索树的删除

- 在二叉搜索树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉搜索树的性质不会失去。
- 为保证在删除后树的搜索性能不至于降低，还需要防止重新链接后树的高度增加。
  - 删除叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。
  - 被删结点右子树为空，可以拿它的左子女结点顶替它的位置，再释放它。

- 被删结点左子树为空，可以拿它的右子女结点顶替它的位置，再释放它。
- 被删结点左、右子树都不为空，可以在它的右子树中寻找中序下的第一个结点（关键码最小），用它的值填补到被删结点中，再来处理这个结点的删除问题。




53
17
78
09
23
94
88
删除78
左子树空
用右子女顶替
53
17
94
09
23
88
53
17
78
09
45
65
94
23
81
88
删除78
在右子树上找
中序下第一个
结点填补
53
17
81
09
45
65
94
23
88

二叉搜索树的删除算法

```
template <class Type> bool BST <Type> ::
Remove(const Type &x,
            BstNode <Type> *&ptr) {
   BstNode <Type> *temp;
   if (ptr!=NULL)
     if (x<ptr->data)  //在左子树中删除
       Remove(x, ptr->leftChild);
     else if (x>ptr->data)  //在右子树中删除
            Remove(x, ptr->rightChild);
     else if (ptr->leftChild!=NULL &&
                 ptr->rightChild!=NULL) {
```

```
    temp=Min(ptr->rightChild);
    //找ptr右子树中序下第一个结点
    ptr->data=temp->data;  //填补上
    Remove(ptr->data, ptr->rightChild);
    //在ptr的右子树中删除temp结点
  }
  else {  //ptr结点只有一个或零个子女
    temp=ptr;
     if (ptr->leftChild==NULL)
        ptr=ptr->rightChild;  //只有右子女
    else if (ptr->rightChild==NULL)
        ptr=ptr->leftChild;  //只有左子女
    delete temp;  return true;
  }
  return false;
}
```

非平衡二叉搜索树的性能分析

- 在非平衡二叉搜索树中树的高度分布在**$\log_2 n$到n之间。**
  - 由于二叉搜索树的查询、插入和删除操作的性能均同树的高度相同，所以在非平衡二叉搜索树中其操作的复杂性在**$\log_2 n$到n之间。**
  - 当非平衡二叉搜索树的高度为**n**时，则系统性能较差。造成这一现象的原因在于非平衡二叉搜索树过于不平衡，为了解决这个问题，需要一个平衡性较好的数据结构。

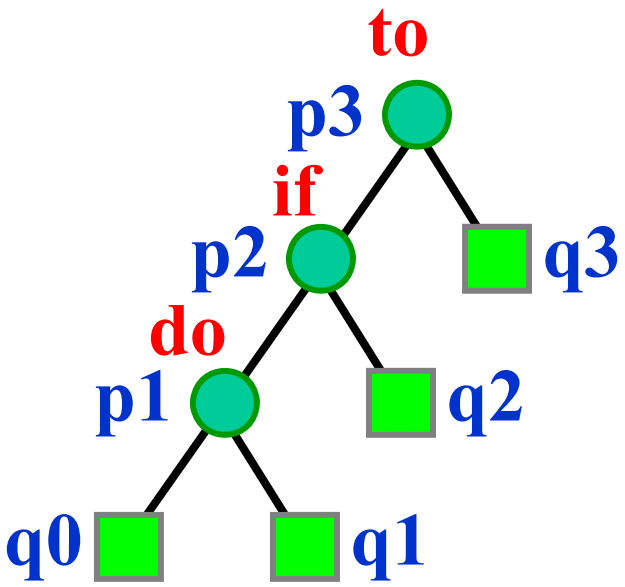
二叉搜索树性能分析

- 对于有***n***个关键码的集合，其关键码有***n!***种不同排列，可构成不同二叉搜索树有

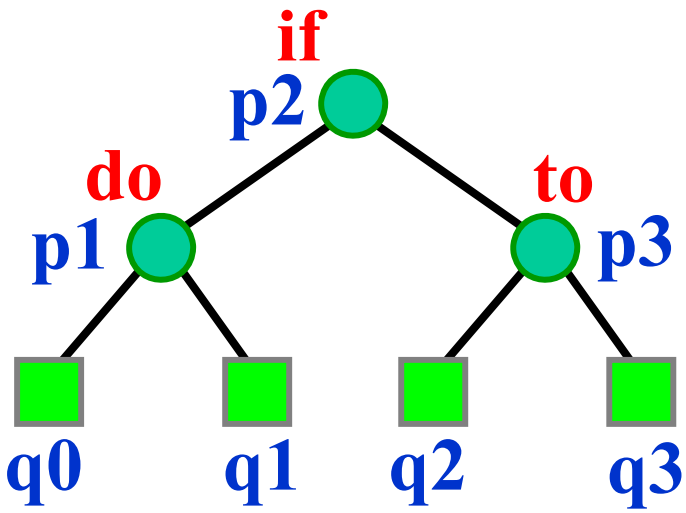
$$\frac{1}{n+1}C_{2n}^{n}$$（棵）

- 用树的搜索效率来评价这些二叉搜索树。
- 为此，在二叉搜索树中加入外结点，形成判定树（扩充二叉树）。

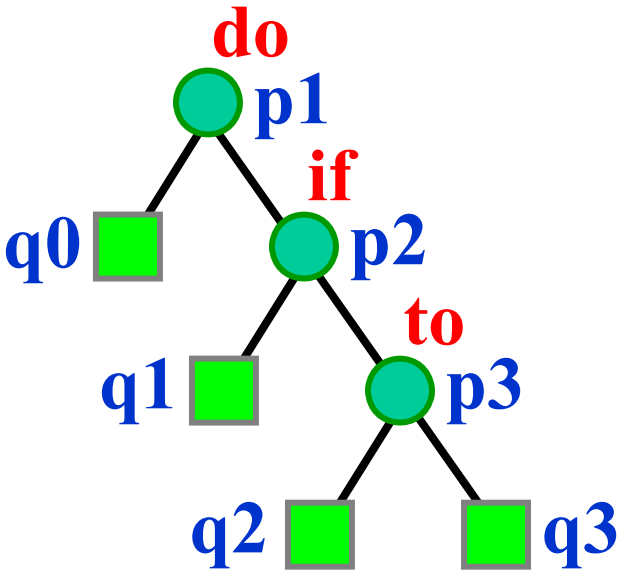
- 例，已知关键码集合{a1, a2, a3}={do, if, to}，对应搜索概率p1, p2, p3，在各搜索不成功间隔内搜索概率分别为q0, q1, q2, q3，可能的二叉搜索树如下所示。



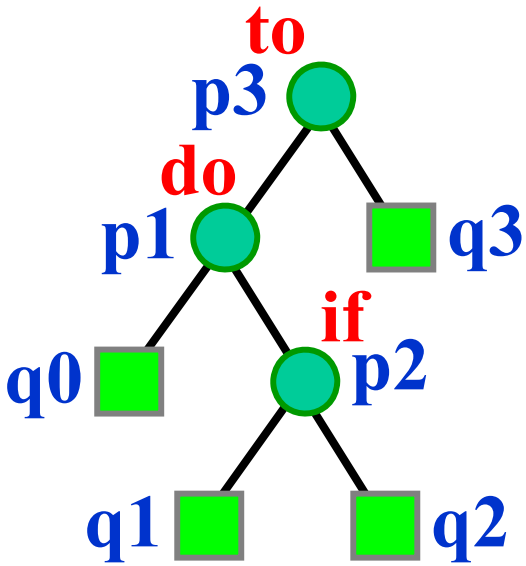
(a)



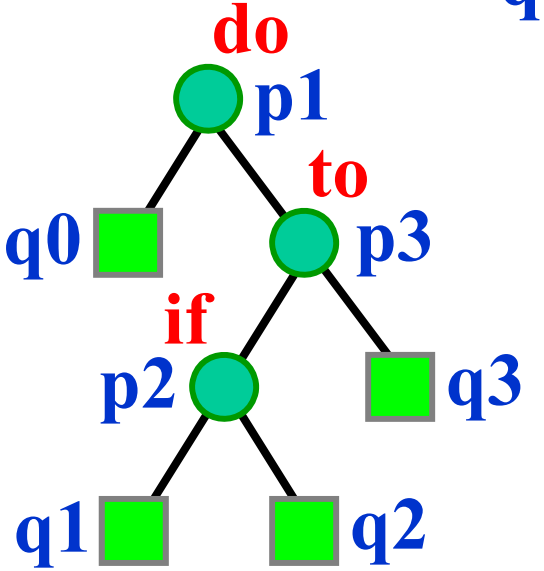
(b)


do
p1
if
q0
p2
to
q1
p3
q2
q3

(c)


to
p3
do
p1
q3
if
q0
p2
q1
q2

(d)


do
p1
to
q0
p3
if
p2
q3
q1
q2

(e)

扩充二叉搜索树

- 在扩充二叉搜索树中
  - ○表示内部结点，包含关键码集合中的某一个关键码；
  - □表示外部结点，代表各关键码间隔中的不在关键码集合中的关键码。
- 在每两个外部结点间必存在一个内部结点。
- 一棵扩充二叉搜索树的搜索成功的平均搜索长度ASL_{succ}可以定义为该树所有内部结点上的搜索概率$p[i]$与搜索该结点时所需的关键码比较次数$c[i]$ $(=l[i]+1)$乘积之和：

$$ASL_{succ} = \sum_{i=1}^{n} p[i]*(l[i]+1)$$

设各关键码的搜索概率相等$p[i]=1/n$：

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

搜索不成功的平均搜索长度ASL_{unsucc}为树中所有外部结点上搜索概率$q[j]$与到达外部结点所需关键码比较次数$c'[j](=l'[j])$乘积之和：

$$ASL_{unsucc} = \sum_{j=0}^{n} q[j] * l'[j]$$

设外部结点搜索概率相等$q[j]=1/(n+1)$：

$$ASL_{unsucc} = \frac{1}{n+1}\sum_{j=0}^{n} l'[j]$$

(1) 相等搜索概率的情形

设树中所有内、外部结点的搜索概率都相等：

$p[i]=1/3,\ 1\le i\le 3,\ \ q[j]=1/4,\ 0\le j\le 3$

图(a): $ASL_{succ}=1/3*3+1/3*2+1/3*1=6/3$,

$ASL_{unsucc}=1/4*3*2+1/4*2+1/4*1=9/4$。

图(b): $ASL_{succ}=1/3*2*2+1/3*1=5/3$,

$ASL_{unsucc}=1/4*2*4=8/4$。

图(c): $ASL_{succ}=1/3*1+1/3*2+1/3*3=6/3$,

$ASL_{unsucc}=1/4*1+1/4*2+1/4*3*2=9/4$。

图(d): $ASL_{succ}=1/3*2+1/3*3+1/3*1=6/3$,

$ASL_{unsucc}=1/4*2+1/4*3*2+1/4*1=9/4$。

图(e): $ASL_{succ}=1/3*1+1/3*3+1/3*2=6/3$,

$ASL_{unsucc}=1/4*1+1/4*3*2+1/4*2=9/4$。

∴ 图(b)的情形所得的平均搜索长度最小。

一般把平均搜索长度达到最小的扩充二叉搜索树称作最优二叉搜索树。

在相等搜索概率的情形下，所有内部、外部结点的搜索概率都相等，视它们的权值都为1。同时，第k层有2^k个结点，k=0, 1, …，则有n个内部结点的扩充二叉搜索树的内部路径长度i至少等于序列

0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, …

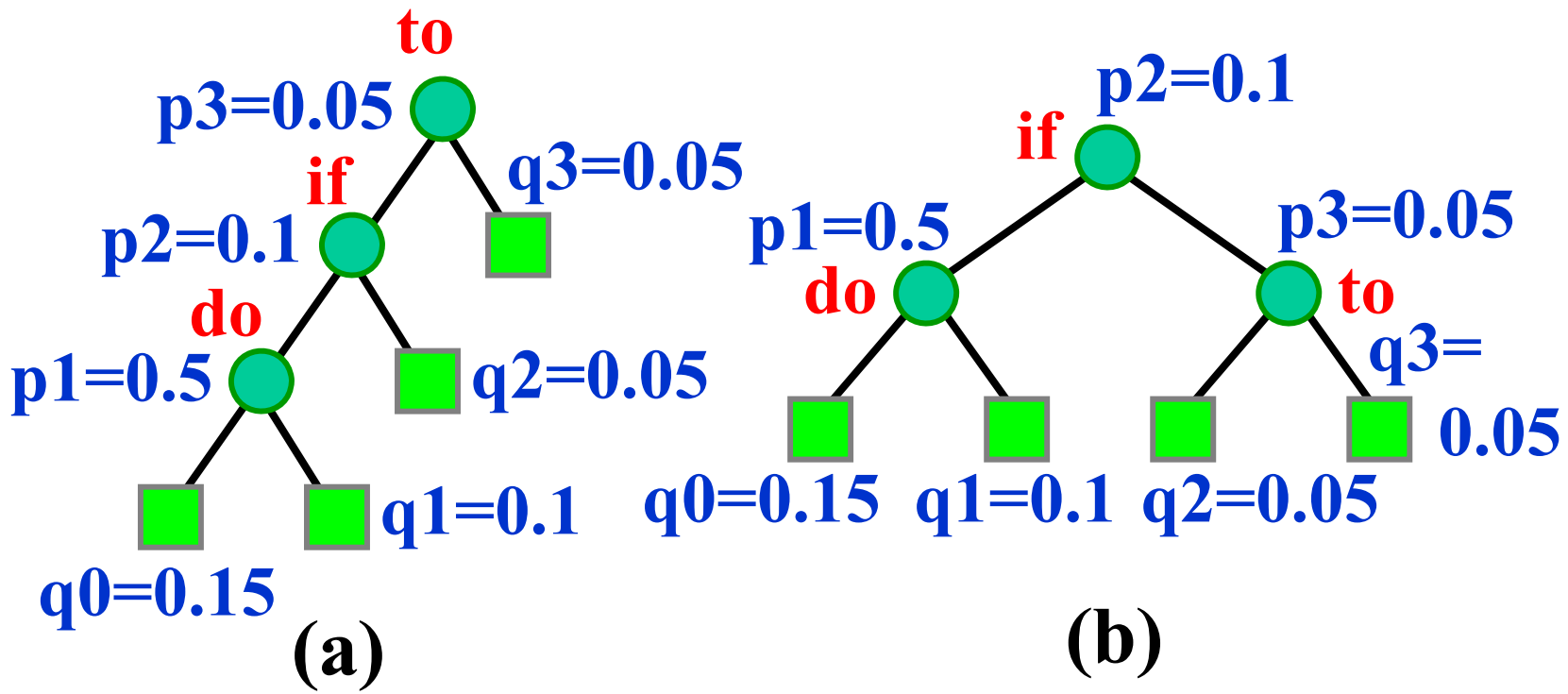
的前 n 项的和。

(2) 不相等搜索概率的情形

设二叉搜索树中所有内、外部结点的搜索概率互不相等。

$p[1]=0.5, p[2]=0.1, p[3]=0.05$

$q[0]=0.15, q[1]=0.1, q[2]=0.05, q[3]=0.05$

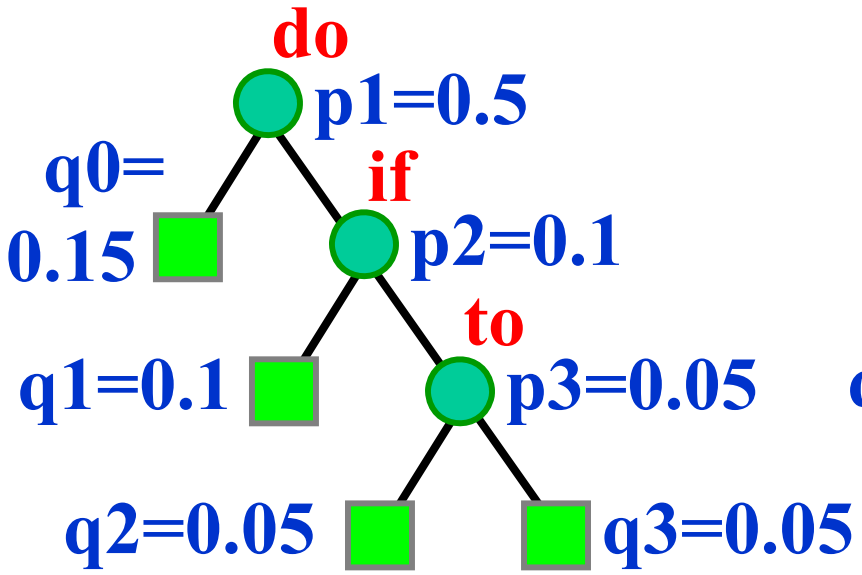


图(a): ASL_{succ}=0.5*3+0.1*2+0.05*1=1.75,

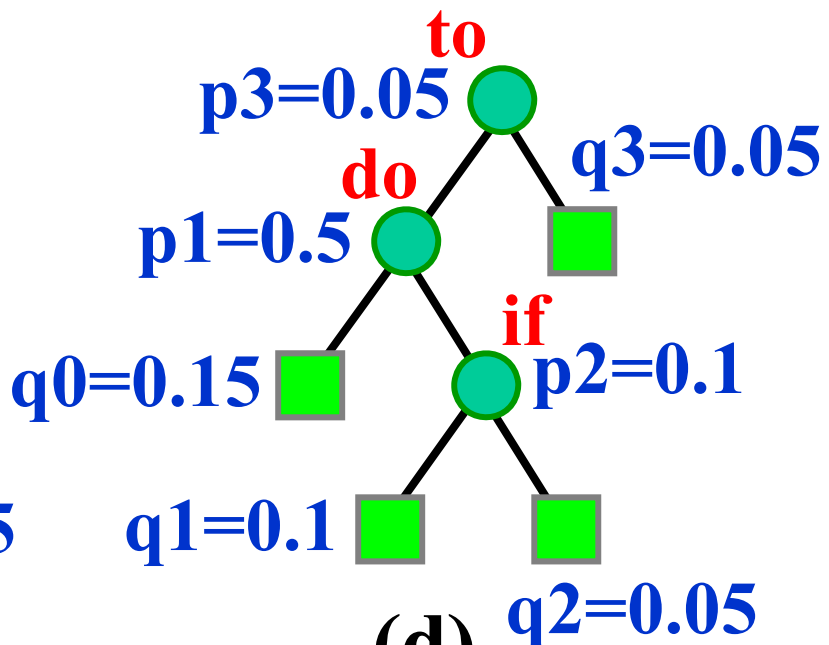
ASL_{unsucc}=0.15*3+0.1*3+0.05*2+0.05*1=0.9。

图(b): ASL_{succ}=0.5*2+0.1*1+0.05*2=1.2,

ASL_{unsucc}=(0.15+0.1+0.05+0.05)*2=0.7。



(c)



(d)

图(c): ASL_{succ}=0.5*1+0.1*2+0.05*3=0.85,

ASL_{unsucc}=0.15*1+0.1*2+0.05*3+0.05*3=0.75。

图(d): ASL_{succ}=0.5*2+0.1*3+0.05*1=1.35,

ASL_{unsucc}=0.15*2+0.1*3+0.05*3+0.05*1=0.8。

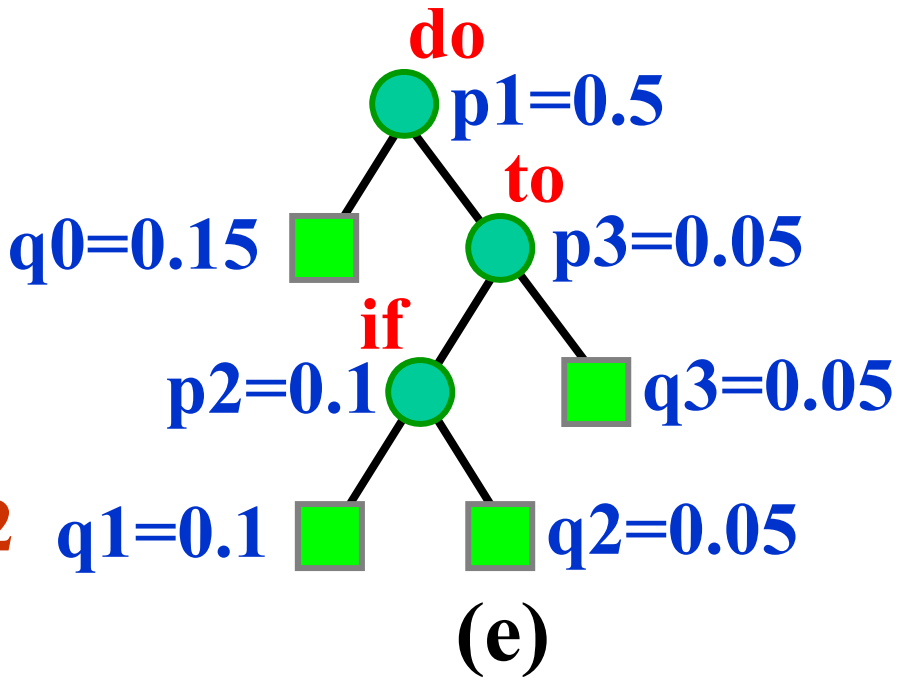
图(e):

ASL_{succ}=0.5*1+0.1*3+0.05*2

=0.9

ASL_{unsucc}=0.15*1+0.1*3+0.05*3+0.05*2

=0.7



(e)

由此可知，图(c)和图(e)的情形下树的平均搜索长度达到最小。因此，图(c)和图(e)的情形是最优二叉搜索树。

与二叉搜索树相关的中序游标类

二叉搜索树中序游标类的类定义

```
template <class Type> class InorderIterator {
  private:
    BST <Type> &ref;  //二叉搜索树对象
    Stack < BstNode <Type> *> itrStack;  //迭代工作栈
  public:
    InorderIterator(BST<Type> &Tree) : ref(Tree)
       { Init( ); }  //构造函数
    int Init( );  //迭代栈初始化
    int operator ! ( );  //判迭代栈空否
    Type operator ( ) ( );  //取栈顶元素关键码
    int operator ++ ( );  //按前序序列进栈，遍历
};
```

中序游标类的部分操作

```
template <class Type>
int InorderIterator <Type> :: Init( ) {
   itrStack.MakeEmpty( );  //迭代栈置空
   if (ref.root!=NULL)  //树非空，根进栈
      itrStack.Push(ref.root);
      return !itrStack.IsEmpty( );  //栈空返回0
}
```

```
template <class Type> BST<Type> ::
BST(const BST<Type> &T) : root(NULL) {
    InorderIterator <Type> itr( );
    for (itr.init( ); !itr; itr++)
        Insert(itr( ));
}

template <class Type>
int InorderIterator <Type> :: operator ! ( ) {
    return !itrStack.IsEmpty( );  //栈空返回0
}
```

```
template <class Type>
int InorderIterator <Type> :: operator ++ ( ) {
    BstNode <Type> *current=itrStack.GetTop( );
    BstNode <Type> *next=current->leftChild;
    if (next!=NULL)  //栈顶元素左子女进栈
       { itrStack.Push(next);  return 1; }
    while (!itrStack.IsEmpty( )) {  //栈非空时
        current=itrStack.Pop( );
        next=current->rightChild;
        if (next!=NULL)  //右子女非空，进栈
          { itrStack.Push(next);  return 1; }
    }
    return 0;
}
```

```
template <class Type>
Type InorderIterator <Type> :: operator ( ) ( ) {
    BstNode <Type> *current=itrStack.GetTop( );
    return current->data;  //返回栈顶元素值
}

template <class Type> BST <Type> :: ~BST( ) {
    MakeEmpty( );  //二叉搜索树析构函数
}
```

随堂练习

例1：从具有n个结点的二叉搜索树中查找一个元素时，最坏情况下的渐进时复杂度为_____。二叉搜索树的查找长度不仅与_____有关，而且也与二叉搜索树的_____有关。

例2：设二叉搜索树利用二叉链表作为存储结构，其每一结点数据域为整数，现给出一个整数x，请编写非递归程序，实现将data域之值小于x的结点全部删除掉。

例3：判二叉树是否是一棵二叉搜索树。

例1：从具有n个结点的二叉搜索树中查找一个元素时，最坏情况下的渐进时复杂度为 O(n) 。二叉搜索树的查找长度不仅与 所包含的结点个数 有关，而且也与二叉搜索树的 生成过程 有关。

(1) 最坏情况下二叉搜索树变为单支树，渐进时间复杂度由$O(\log_2 n)$变为$O(n)$。

例2：设二叉搜索树利用二叉链表作为存储结构，其每一结点数据域为整数，现给出一个整数x，请编写非递归程序，实现将data域之值小于x的结点全部删除掉。

在非递归中序遍历二叉搜索树过程中，若访问的结点其data值小于等于x时则删除此结点，但这种操作应仍然保持二叉搜索树中序遍历的有序特性。

难点在于，当删除某结点后，如何正确访问到所删除结点的后继结点。分析如下：若一个结点*p将要被删除，则此结点*p的左孩子必定为空，这是因为*p的左孩子其关键字一定比*p关键字小，所以必然在结点*p之前删除。这样，如果结点*p的右孩子不为空，则将p指向*p的右孩子，然后删除原*p结点，接下来再对p继续进行中序遍历。

例3：判二叉树是否是一棵二叉搜索树。采用递归算法。

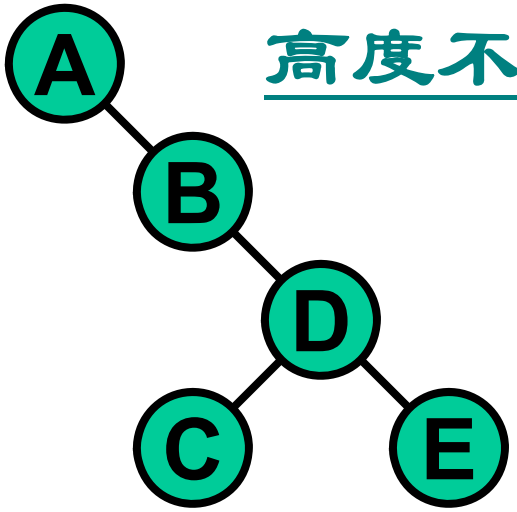
空二叉树T是一棵二叉搜索树；若T非空，首先是它的左、右两棵子树都是二叉搜索树，接着还要求左子树中的所有结点的键值都小于根结点的键值；并且右子树中的所有结点的键值都大于根结点的键值。为了能简单实现子树所有结点与根结点比较，对于左子树可用左子树中最大的键值；对于右子树可以用右子树中最小的键值。若左子树中的最大键值比根结点键值小；并且右子树中的最小键值比根结点键值大，就能得出这是一棵二叉搜索树的结论。

为此要设计的递归判定函数在判定一棵二叉树是二叉搜索树的过程中同时求得这棵二叉树的最大键值和最小键值。

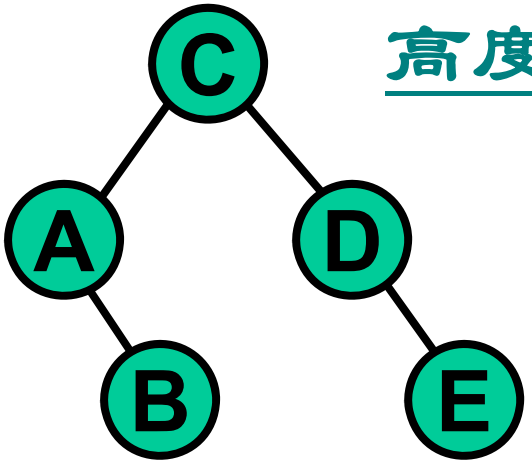
AVL树 高度平衡的二叉搜索树

AVL树的定义

一棵AVL树或者是空树，或者是具有下列性质的二叉搜索树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。



高度不平衡



高度平衡

结点的平衡因子 (Balance Factor)

- 每个结点附加一个数字，给出该结点右子树的高度减去左子树的高度所得的高度差，这个数字即为结点的平衡因子**balance**。
- **AVL**树任一结点平衡因子只能取**-1, 0, 1**。
- 如果一个结点的平衡因子的绝对值大于**1**，则这棵二叉搜索树就失去了平衡，不再是**AVL**树。
- 如果一棵二叉搜索树是高度平衡的，且有n个结点，其高度可保持在$O(\log_2 n)$，平均搜索长度也可保持在$O(\log_2 n)$。

AVL树的类定义

```
template <class Type> class AVLTree {
  public:
    struct AVLNode {  //AVL树结点
        Type data;  int balance;
        AVLNode <Type> *left, *right;
        AVLNode( ) : left(NULL), right(NULL),
                              balance(0) { }
        AVLNode(Type d,
                        AVLNode <Type> *l=NULL,
                        AVLNode <Type> *r=NULL)
          : data(d), left(l), right(r), balance(0) { }
```

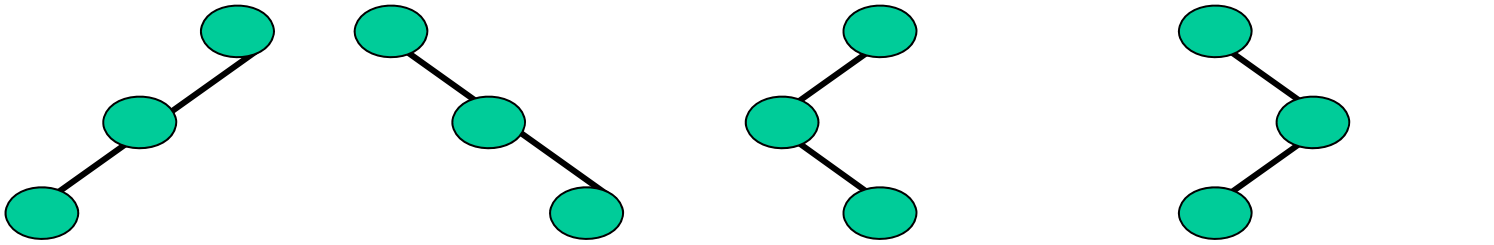
```
protected:
 Type RefValue;
 AVLNode *root;
 bool Insert(AVLNode <Type> *&Tree, Type x);
 bool Remove(AVLNode <Type> *&Tree, Type x);
 void RotateLeft(AVLNode <Type> *Tree,
                 AVLNode <Type> *&NewTree);
 void RotateRight(AVLNode <Type> *Tree,
                  AVLNode <Type> *&NewTree);
 void LeftBalance(AVLNode <Type> *&Tree);
 void RightBalance (AVLNode <Type> *&Tree);
 int Height(AVLNode <Type> *t) const;
```

```
public:
    AVLTree( ) : root(NULL) { }
    AVLNode(Type Ref) : RefValue(Ref),
                                root(NULL) { }
   bool Insert(Type x) { return Insert(root, x); }
   bool Remove(Type x) { return Remove(root, x); }
   friend istream & operator >>
          (istream &in, AVLTree <Type> &Tree);
   friend ostream & operator <<
          (ostream &out, const  AVLTree <Type> &Tree);
   int Height( ) const;
};
```

平衡化旋转

- 如果在一棵平衡的二叉搜索树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 平衡化旋转有两类：
  - 单旋转（左旋和右旋）
  - 双旋转（左平衡和右平衡）
- 每插入一个新结点时，**AVL**树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。

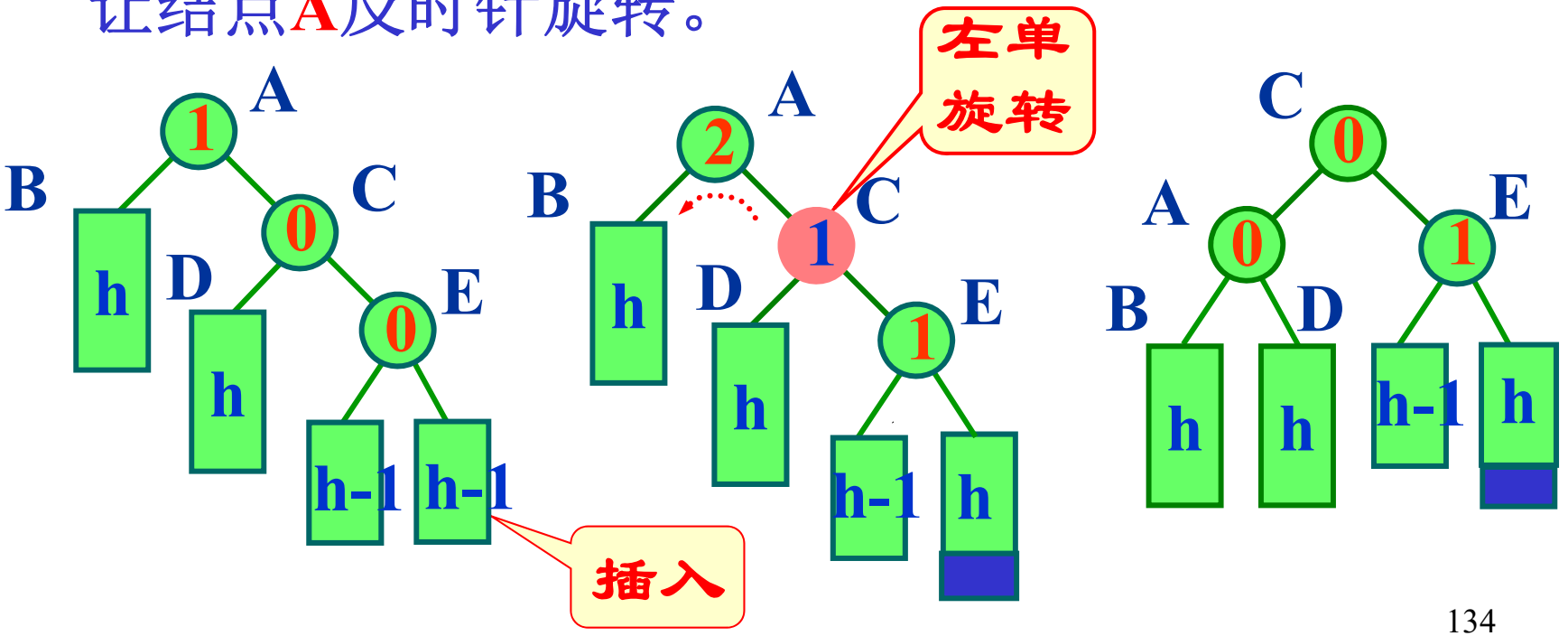
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。



右单旋转　左单旋转　左右双旋转　右左双旋转

左单旋转 (RotateLeft)

- 在结点A的右子女的右子树E中插入新结点，该子树高度增1导致结点A的平衡因子变成2，出现不平衡。为使树恢复平衡，从A沿插入路径连续取3个结点A、C和E，以结点C为旋转轴，让结点A反时针旋转。

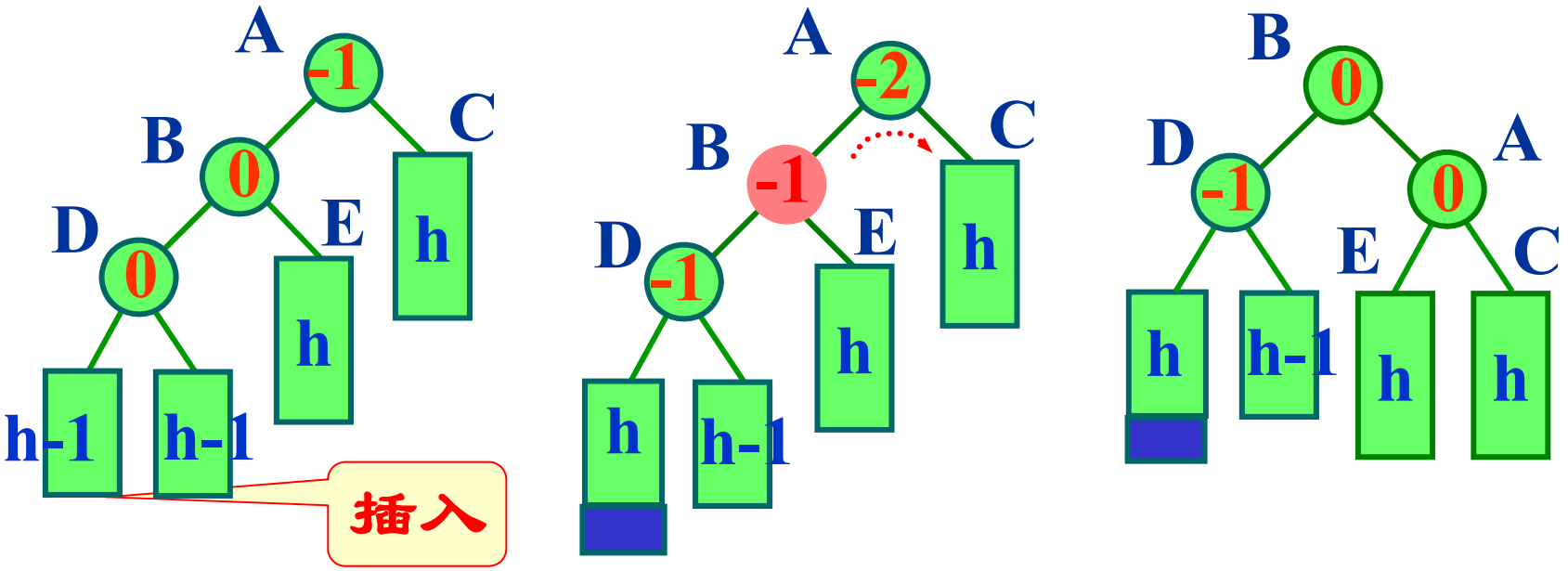


```
template <class E, class K>
void AVLTree <E, K> ::
RotateL(AVLNode <E, K> * &ptr) {
//左单旋转的算法
    AVLNode <E, K> *subL=ptr;
    ptr=subL->right;
    subL->right=ptr->left;
    ptr->left=subL;
    ptr->bf=subL->bf=0;
}
```

```
template <class Type> void AVLTree <Type> ::
RotateLeft(AVLNode <Type> *Tree,
                    AVLNode <Type> *&NewTree) {
//左单旋转的算法
    NewTree=Tree->right;
    Tree->right=NewTree->left;
    NewTree->left=Tree;
}
```

右单旋转 (RotateRight)

- 在结点A的左子女的左子树D上插入新结点使其高度增1导致结点A的平衡因子增到-2，造成不平衡。为使树恢复平衡，从A沿插入路径连续取3 个结点A、B和D，以结点B为旋转轴，将结点A顺时针旋转。

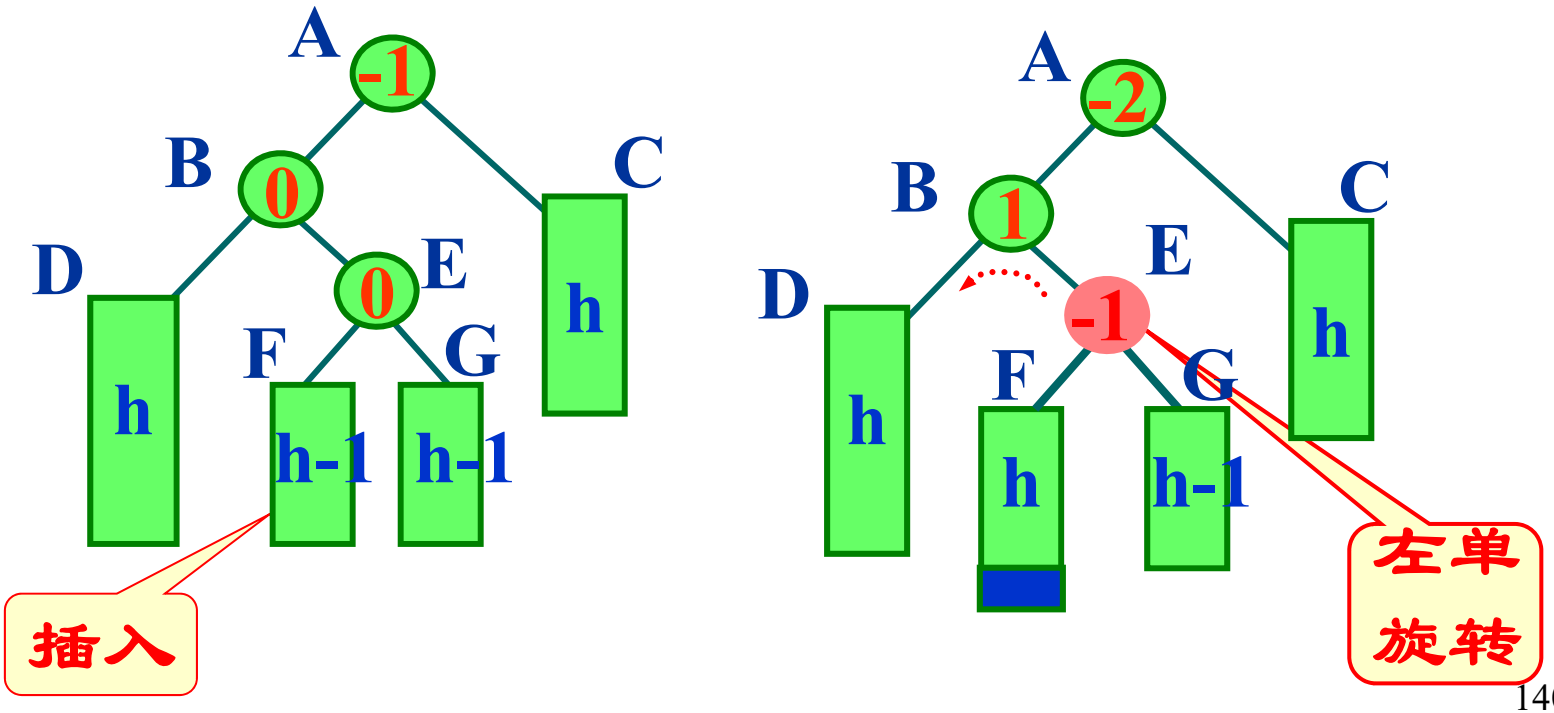


```
template <class E, class K>
void AVLTree <E, K> ::
RotateR(AVLNode <E, K> * &ptr) {
//右单旋转的算法
    AVLNode <E, K> *subR=ptr;
    ptr=subR->left;
    subR->left=ptr->right;
    ptr->right=subR;
    ptr->bf=subR->bf=0;
}
```

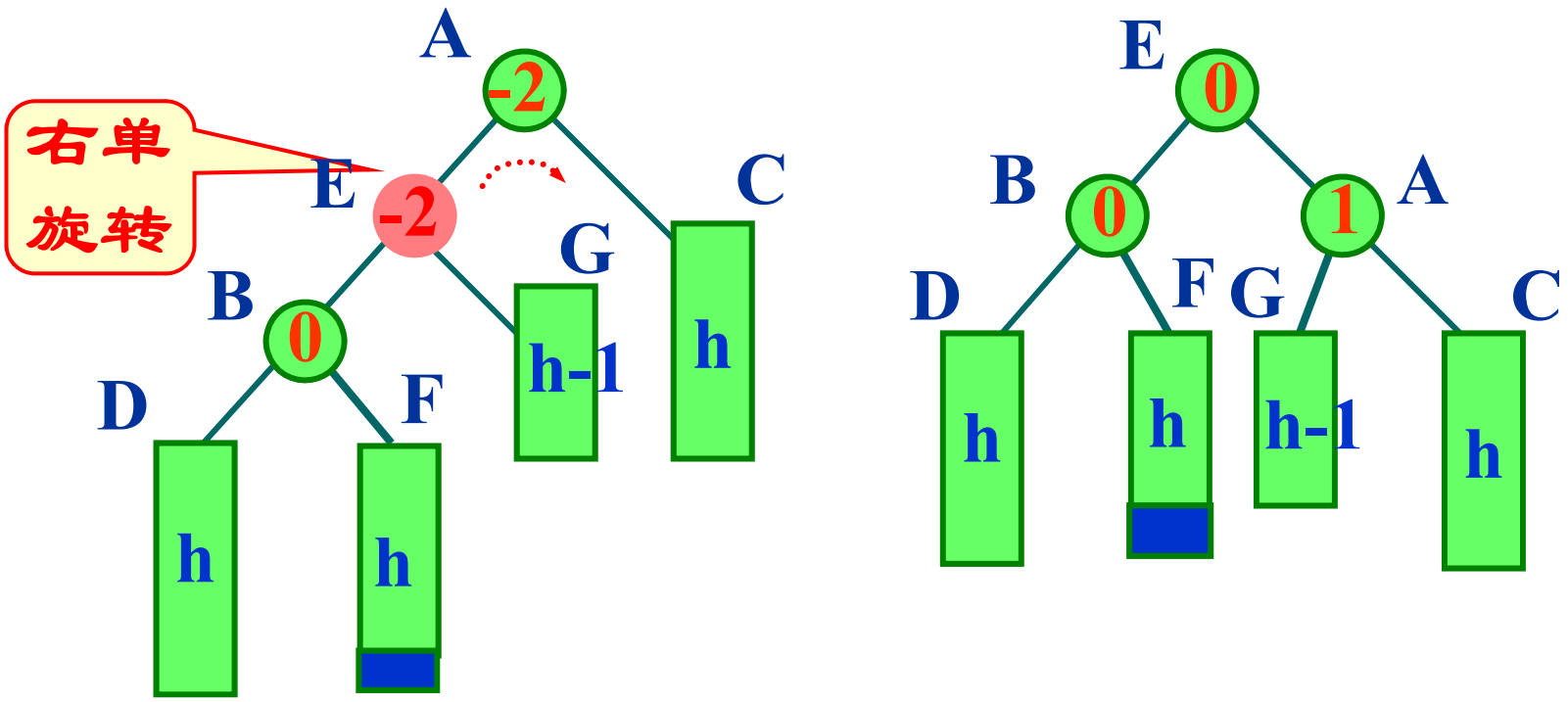
```
template <class Type> void AVLTree <Type> ::
RotateRight(AVLNode <Type> *Tree,
                    AVLNode <Type> *&NewTree) {
//右单旋转的算法
   NewTree=Tree->left;
   Tree->left=NewTree->right;
   NewTree->right=Tree;
}
```

先左后右双旋转 (RotationLeftRight)

- 在结点A的左子女的右子树中插入新结点，该子树的高度增1导致结点A的平衡因子变为-2，造成不平衡。以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置。



- 再以结点E为旋转轴，将结点A顺时针旋转，使之平衡化。



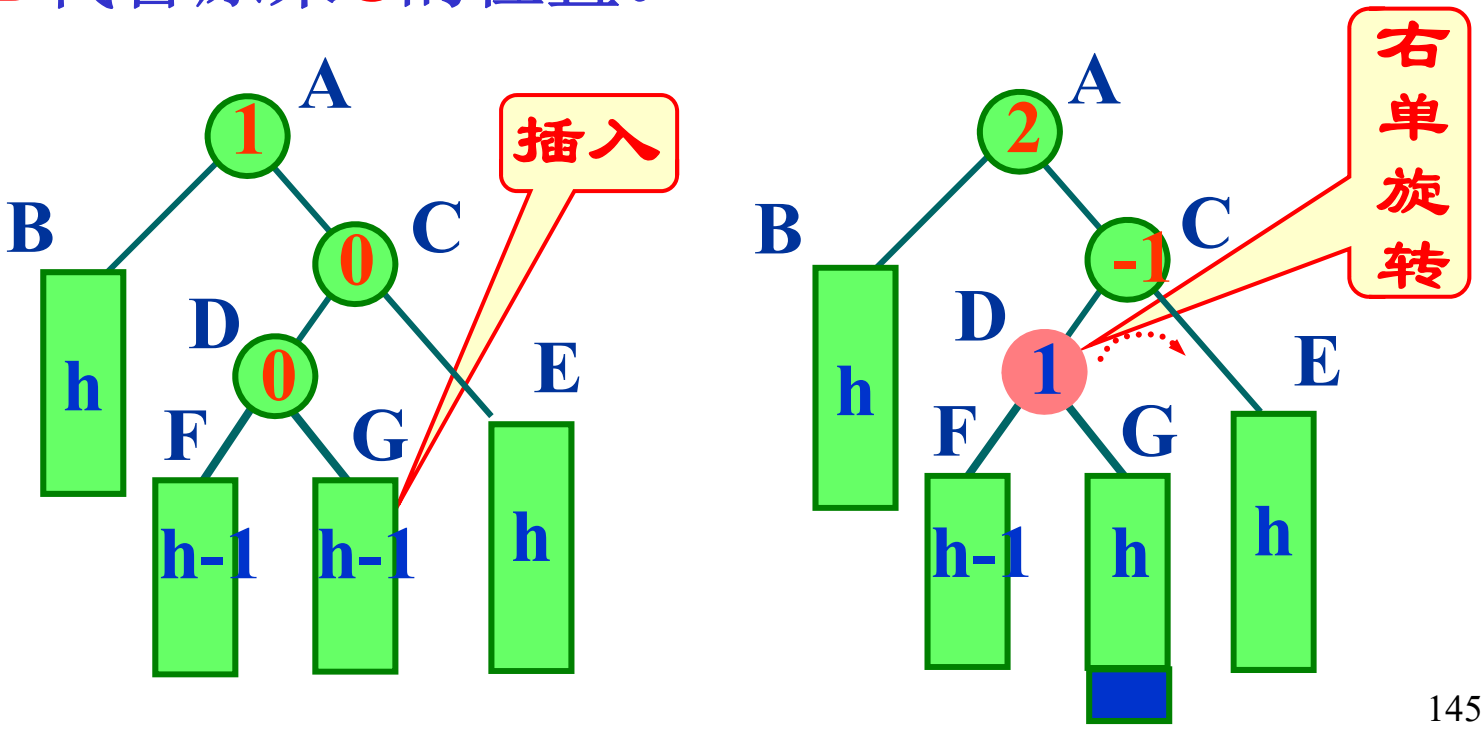
```
template <class E, class K> void AVLTree <E, K> ::
RotateLR(AVLNode <E, K> * &ptr) {
    AVLNode <E, K> *subR=ptr, *subL=subR->left;
    ptr=subL->right;
    subL->right=ptr->left;
    ptr->left=subL;
    if (ptr->bf<=0)  subL->bf=0;
    else subL->bf=-1;
    subR->left=ptr->right;
    ptr->right=subR;
    if (ptr->bf==-1)  subR->bf=1;
    else subR->bf=0;
    ptr->bf=0;
}
```

```
template <class Type> void AVLTree <Type> ::
LeftBalance(AVLNode <Type> *&Tree, int &taller) {
    AVLNode <Type> *leftsub=Tree->left, *rightsub;
    switch (leftsub->balance) {
      case -1 :
        Tree->balance=leftsub->balance=0;
        RotateRight(Tree, Tree);
        taller=0;  break;
      case 0 :
        cout<<"树已经平衡化\n";  break;
      case 1 :
        rightsub=leftsub->right;
        switch (rightsub->balance) {
```

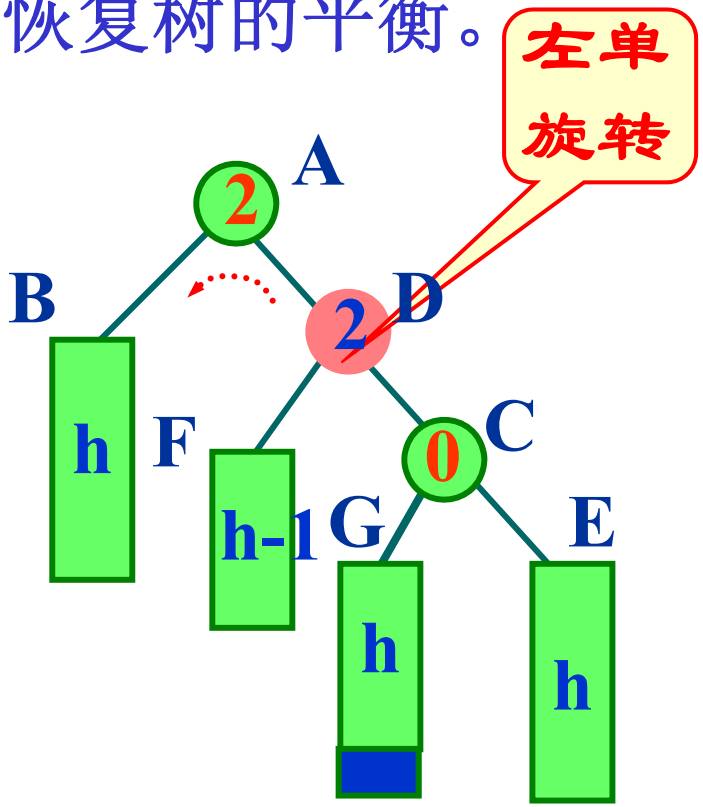
```
            case -1: Tree->balance=1;
                     leftsub->balance=0;  break;
            case 0 : Tree->balance=leftsub->balance=0;
                     break;
            case 1 : Tree->balance=0;
                      leftsub->balance=-1;
        }
        rightsub->balance=0;
        RotateLeft(leftsub, Tree->left);
        RotateRight(Tree, Tree);
        taller=0;
    }
}
```

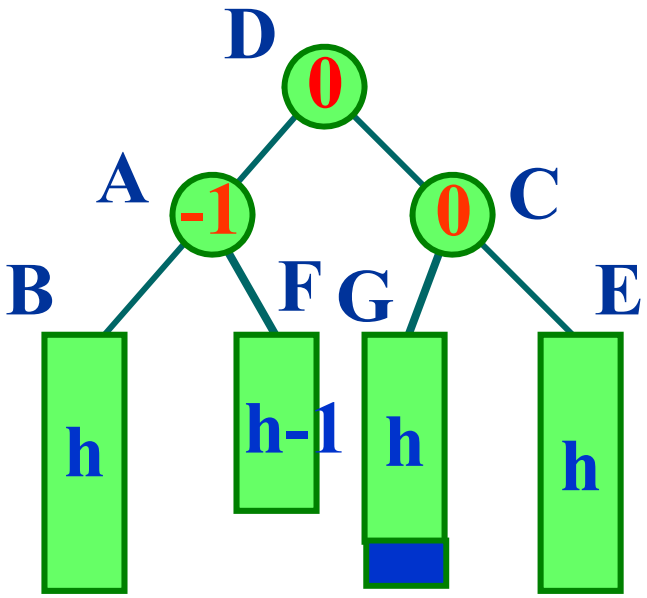
先右后左双旋转 (RotationRightLeft)

- 在结点A的右子女的左子树中插入新结点，该子树高度增1。结点A的平衡因子变为2，发生了不平衡。以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。



- 再以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。





```
template <class E, class K> void AVLTree <E, K> ::
RotateRL(AVLNode <E, K> * &ptr) {
    AVLNode <E, K> *subL=ptr, *subR=subL->right;
    ptr=subR->left;
    subR->left=ptr->right;
    ptr->right=subR;
    if (ptr->bf >=0)  subR->bf=0;
    else subR->bf=1;
    subL->right=ptr->left;
    ptr->left=subL;
    if (ptr->bf==1)  subL->bf=-1;
    else subL->bf=0;
    ptr->bf=0;
}
```

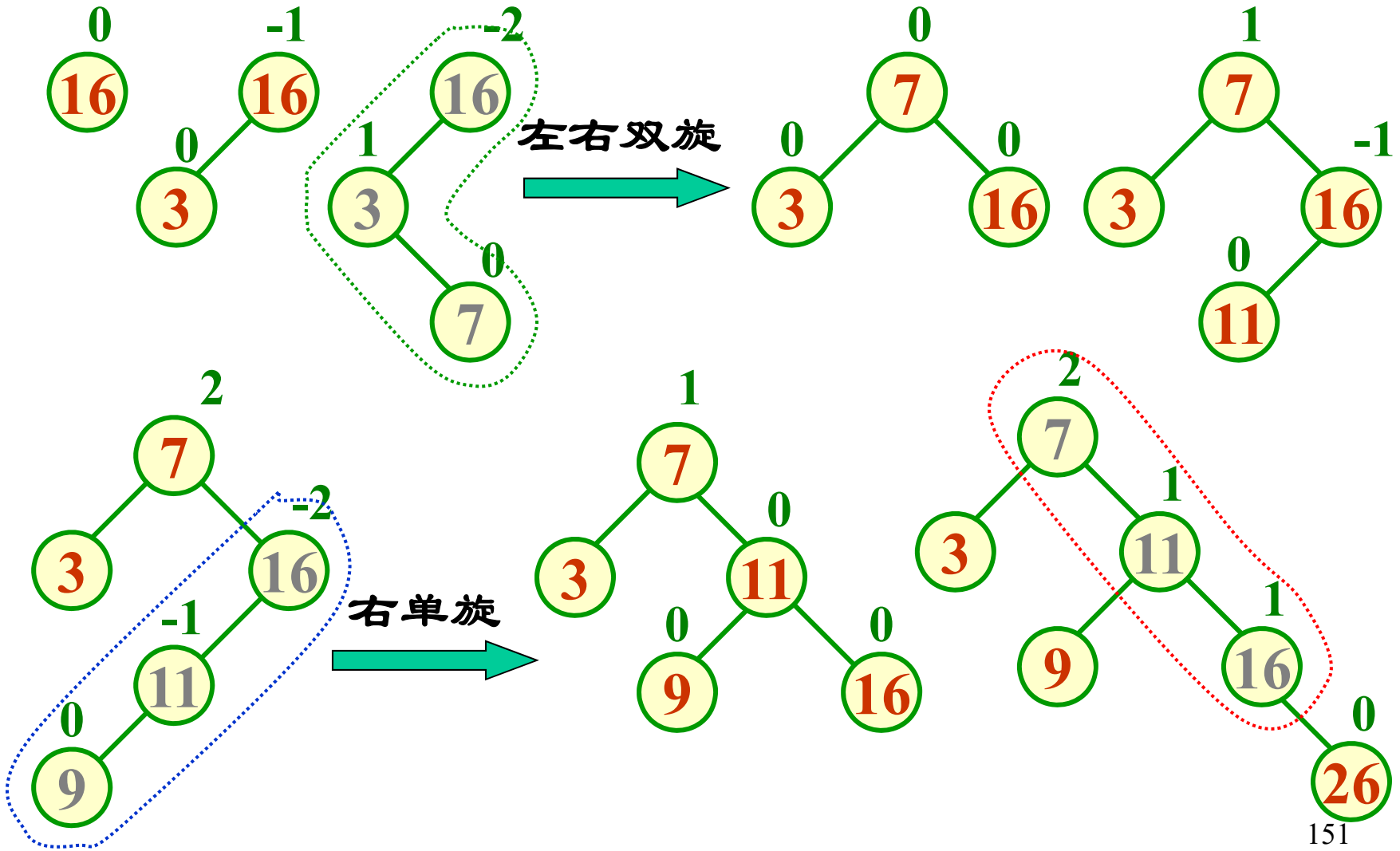
```
template <class Type> void AVLTree <Type>::
RightBalance(AVLNode <Type> *&Tree, int &taller) {
    AVLNode <Type> *rightsub=Tree->right, *leftsub;
    switch (rightsub->balance) {
      case 1 :
         Tree->balance=rightsub->balance=0;
         RotateLeft(Tree, Tree);
         taller=0;  break;
      case 0 : cout<<"树已经平衡化\n";  break;
      case -1 :
         leftsub=rightsub->left;
         switch (leftsub->balance) {
```

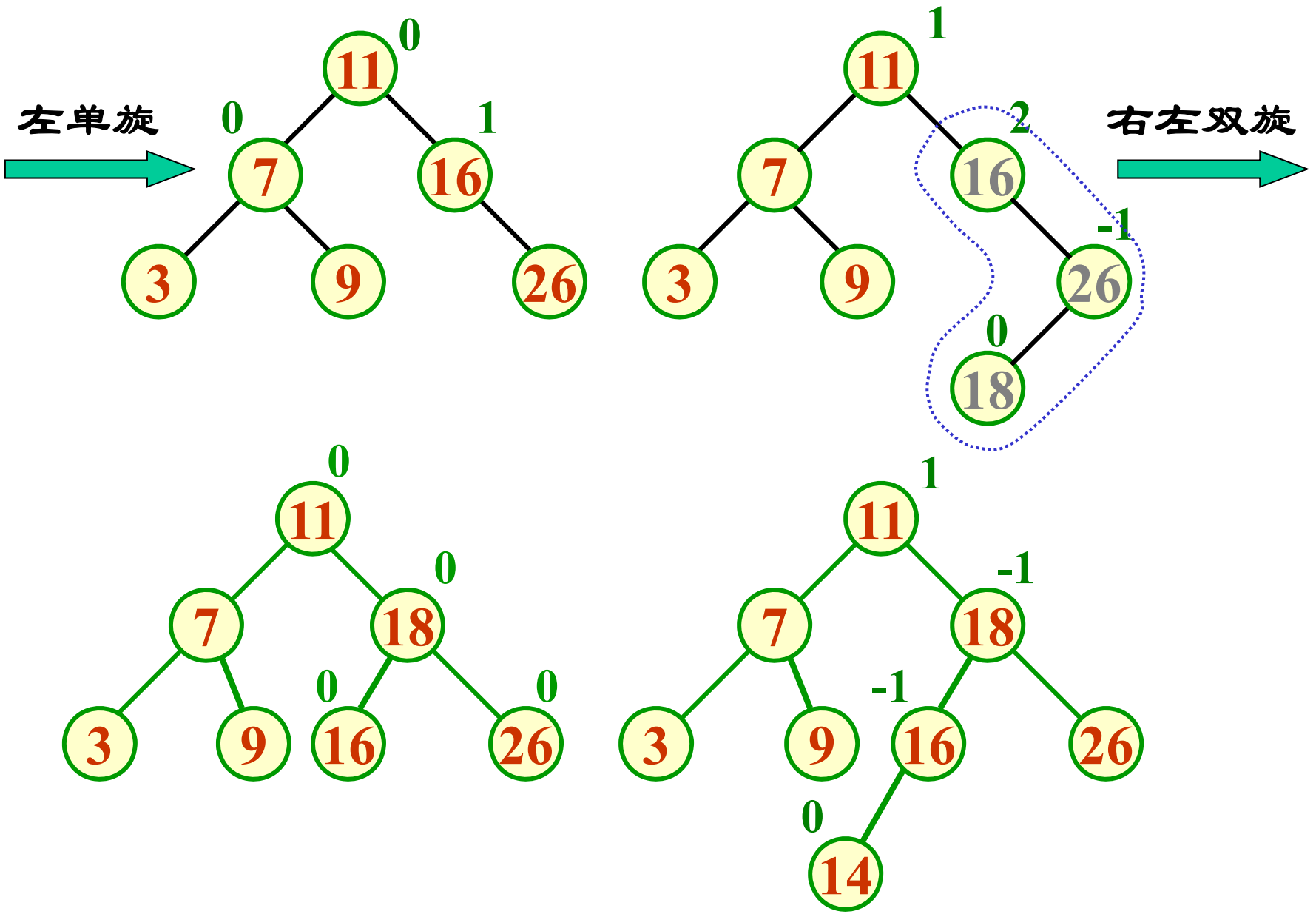
```
            case 1 : Tree->balance=-1;
                     rightsub->balance=0;  break;
            case 0 : Tree->balance=rightsub->balance=0;
                     break;
            case -1 : Tree->balance=0;
                      rightsub->balance=1;  break;
        }
        leftsub->balance=0;
        RotateRight(rightsub, Tree->left);
        RotateLeft(Tree, Tree);
        taller=0;
    }
}
```

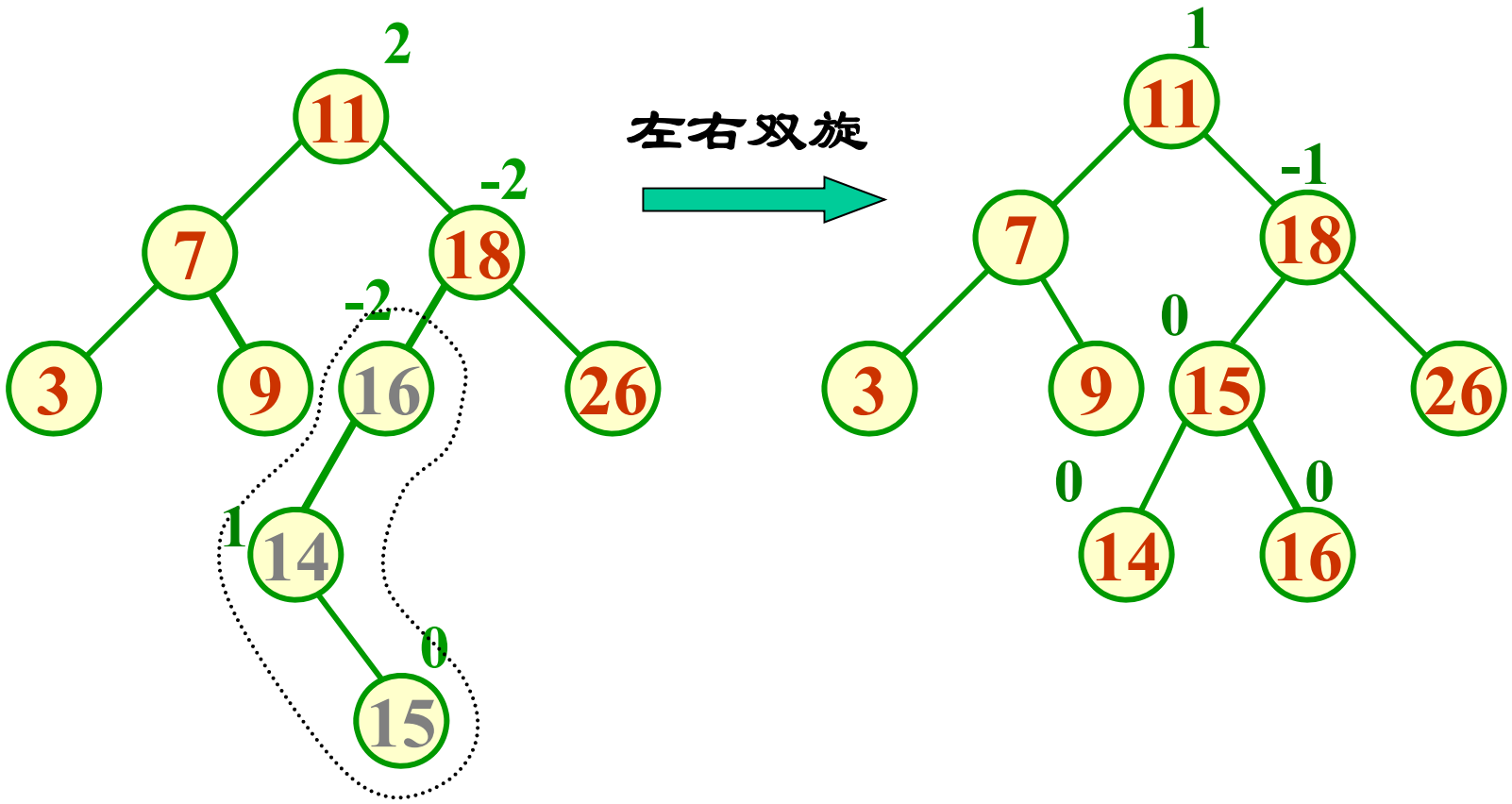
AVL树的插入

- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值|balance|>1，则出现了不平衡，需要做平衡化处理。
- 在AVL树上定义了重载操作“>>”和“<<”，以及中序遍历的算法。利用这些操作可以执行AVL树的建立和结点数据的输出。
- 算法从一棵空树开始，通过输入一系列对象关键码，逐步建立AVL树，在插入新结点时使用平衡旋转方法进行平衡化处理。

例，输入关键码序列为 {16, 3, 7, 11, 9, 26, 18, 14, 15}，插入和调整过程如下。




左单旋
右左双旋
11
7
16
3
9
26
18
14
0
1
2
-1



从空树开始的建树过程

- 下面的算法通过递归方式，将新结点作为叶结点插入并逐层修改各结点的平衡因子。
- 在发现不平衡时，立即执行相应的平衡化旋转操作，使得树中各结点重新平衡化。
- 在程序中，用变量**success**记载新结点是否存储分配成功，并用它作为函数的返回值。
- 算法从树的根结点开始，递归向下找插入位置。在找到插入位置（空指针）后，为新结点动态分配存储空间，将它作为叶结点插入，并置**success**为**1**，再将**taller**置为**1**，以表明插入成功。在退出递归沿插入路径向上返回时，做必要的调整。

```
template <class Type> bool AVLTree <Type> ::
Insert(AVLNode <Type> *&tree,
          Type x, int &taller) {  //AVL树的插入算法
    bool success;
    if (tree==NULL) {
       tree=new AVLNode (x);
       success=(tree!=NULL) ? true : false;
       if (success)  taller=1;  }
    else if (x<tree->data) {
       success=Insert(tree->left, x, taller);
       if (taller)
```

```
        switch (tree->balance) {
          case -1 : LeftBalance(tree, taller);  break;
          case 0 :  tree->balance=-1;  break;
          case 1 :  tree->balance=0;  taller=0;  }  }
    else if (x>tree->data) {
      success=Insert(tree->right, x, taller);
      if (taller)
        switch (tree->balance) {
          case -1 : tree->balance=0;  taller=0;
                    break;
          case 0 :  tree->balance=1;  break;
          case 1 :  RightBalance(tree, taller);  }  }
    return success;
}
```

程序7.20

- **while** (p!=NULL) {
- … //寻找插入位置，同时把经过的结点压栈，循环结束时要么找到结点，返回；要么p为NULL，pr为p的双亲，pr在栈顶。
- p=**new** AVLNode <E, K> (e1);
- … //创建新结点并且连接到树上，从空树变成一个结点的树的情况，则返回。否则pr为双亲，p为新的结点。
- **while** (st.IsEmpty( )==false) {
- … //出栈，栈顶元素保存在pr中，从这里开始，调整pr的平衡因子，然后判断该结束、还是继续回溯、还是做哪一种旋转。
- **if** (st.IsEmpty( )==true) ptr=pr;
- … //把从栈中弹出的pr（是原来的指针的拷贝，所以原来的指针没有改变）所指向的结点，重新连接到原来的指针上（包括一开始的连接到初始的ptr的情况）。
- 结束。

AVL树的删除

(1) 如果被删结点*x*最多只有一个子女，那么问题比较简单：

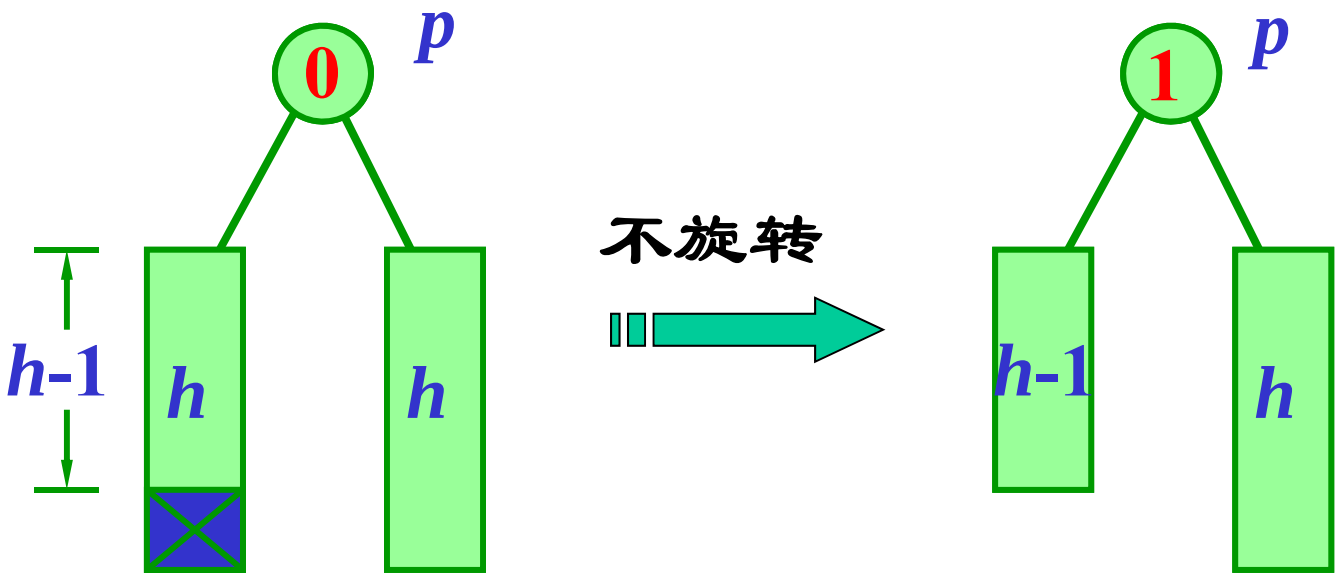
- 将结点*x*从树中删去；
- 因为结点*x*最多有一个子女，可以简单地把*x*的双亲结点中原来指向*x*的指针改指到这个子女结点；
- 如果结点*x*没有子女，*x*双亲结点的相应指针置为**NULL**；
- 将原来以结点*x*为根的子树的高度减**1**。

(2) 如果被删结点*x*有两个子女：

- 搜索*x*在中序次序下的直接前驱*y*（同样可以找直接后继）；
- 把结点*y*的内容传送给结点*x*，现在问题转移到删除结点*y*，把结点*y*当作被删结点*x*；
- 因为结点*y*最多有一个子女，我们可以简单地用**(1)**给出的方法进行删除。

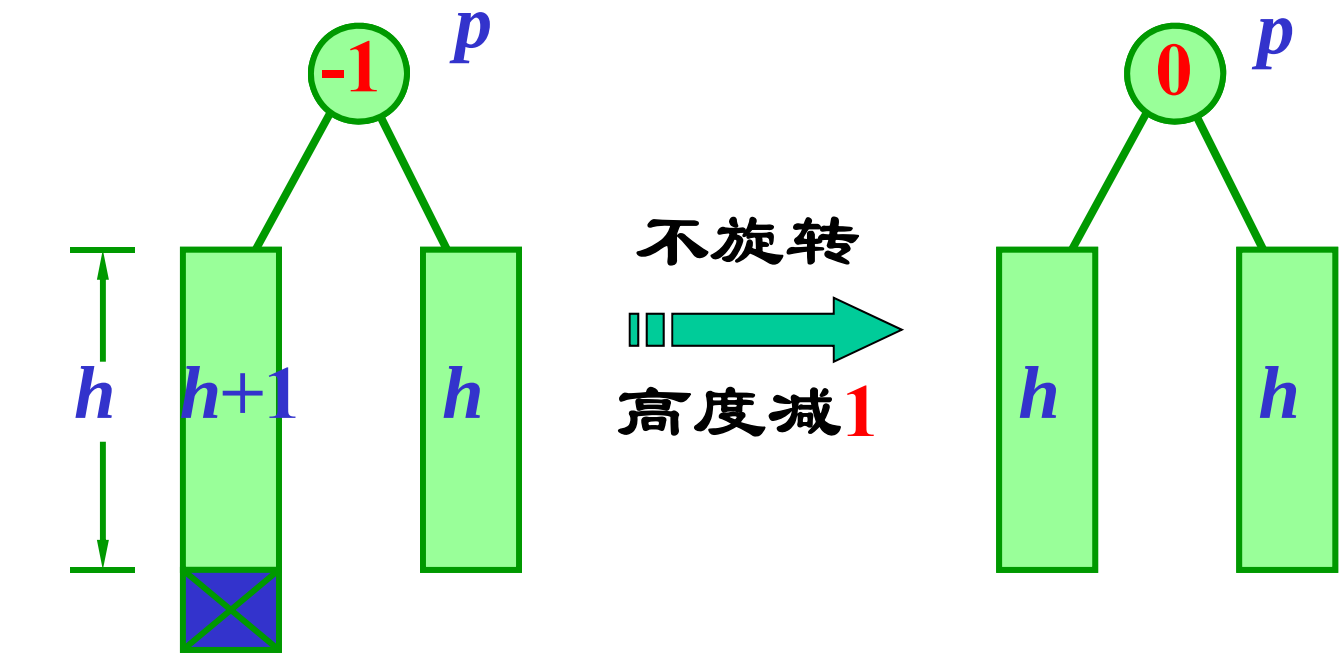
- 必须沿*x*通向根的路径反向追踪高度的变化对路径上各个结点的影响。
- 用一个布尔变量**shorter**来指明子树的高度是否被缩短。在每个结点上要做的操作取决于**shorter**的值和结点的**balance**，有时还要依赖子女的**balance**。
- 布尔变量**shorter**的值初始化为**true**，然后对于从*x*的双亲到根的路径上的各个结点*p*。在**shorter**保持为**true**时执行下面操作，如果**shorter**变成**false**，算法终止。

- **case 1：** 当前结点*p*的balance为0。如果它的左子树或右子树被缩短，则它的balance改为1或-1，同时shorter置为false。



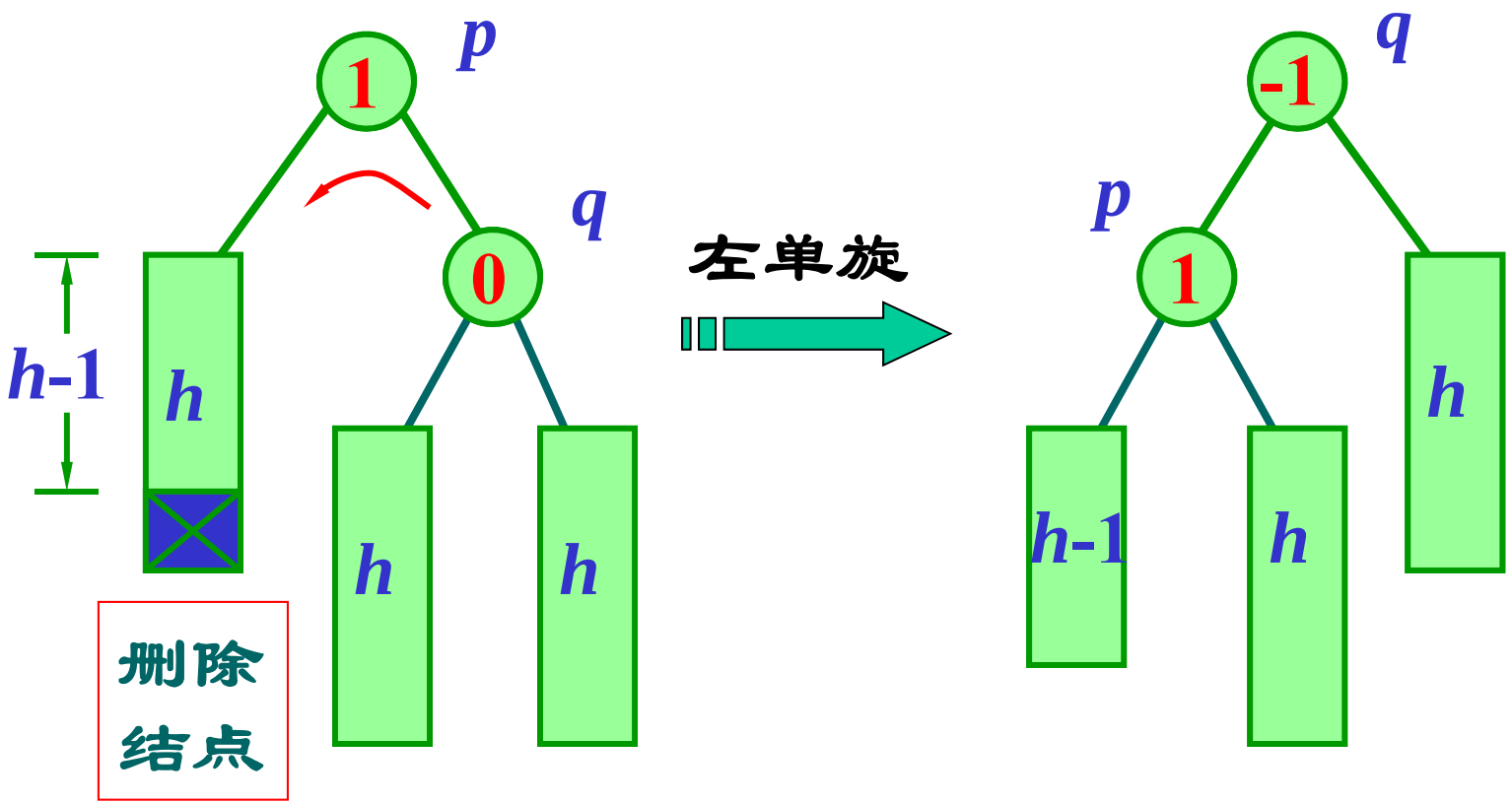
删除一个结点
高度降低一层

- **case 2：** 结点*p*的balance不为0，且较高的子树被缩短，则*p*的balance改为0，同时shorter置为true。

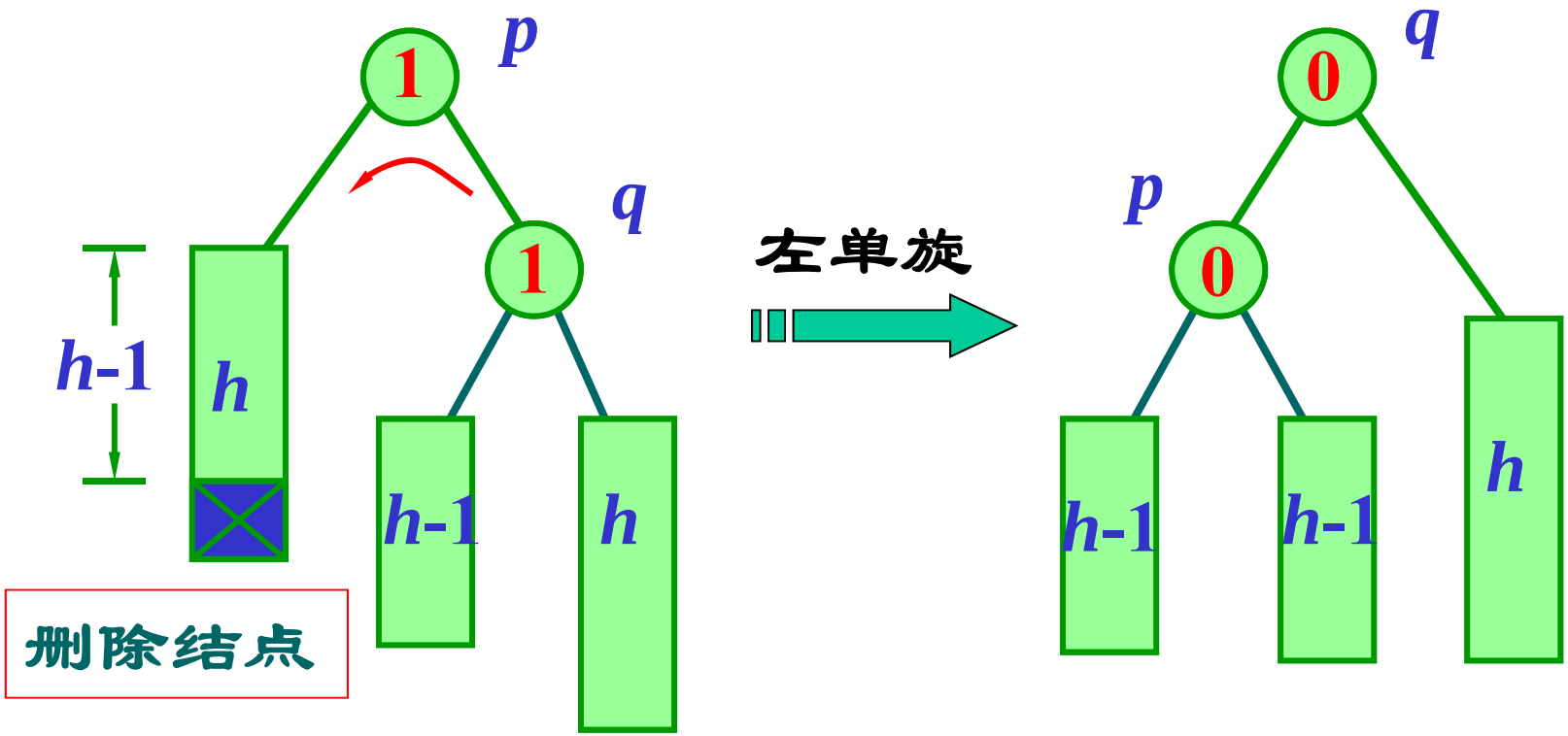


- **case 3**：结点***p***的**balance**不为**0**，且较矮的子树又被缩短，则在结点***p***发生不平衡。需要进行平衡化旋转来恢复平衡。
- 令***p***的较高的子树的根为***q***（该子树未被缩短），根据***q***的**balance**，有如下**3**种平衡化操作。
- 在**case 3a, 3b**和**3c**的情形中，旋转的方向取决于是结点***p***的哪一棵子树被缩短。

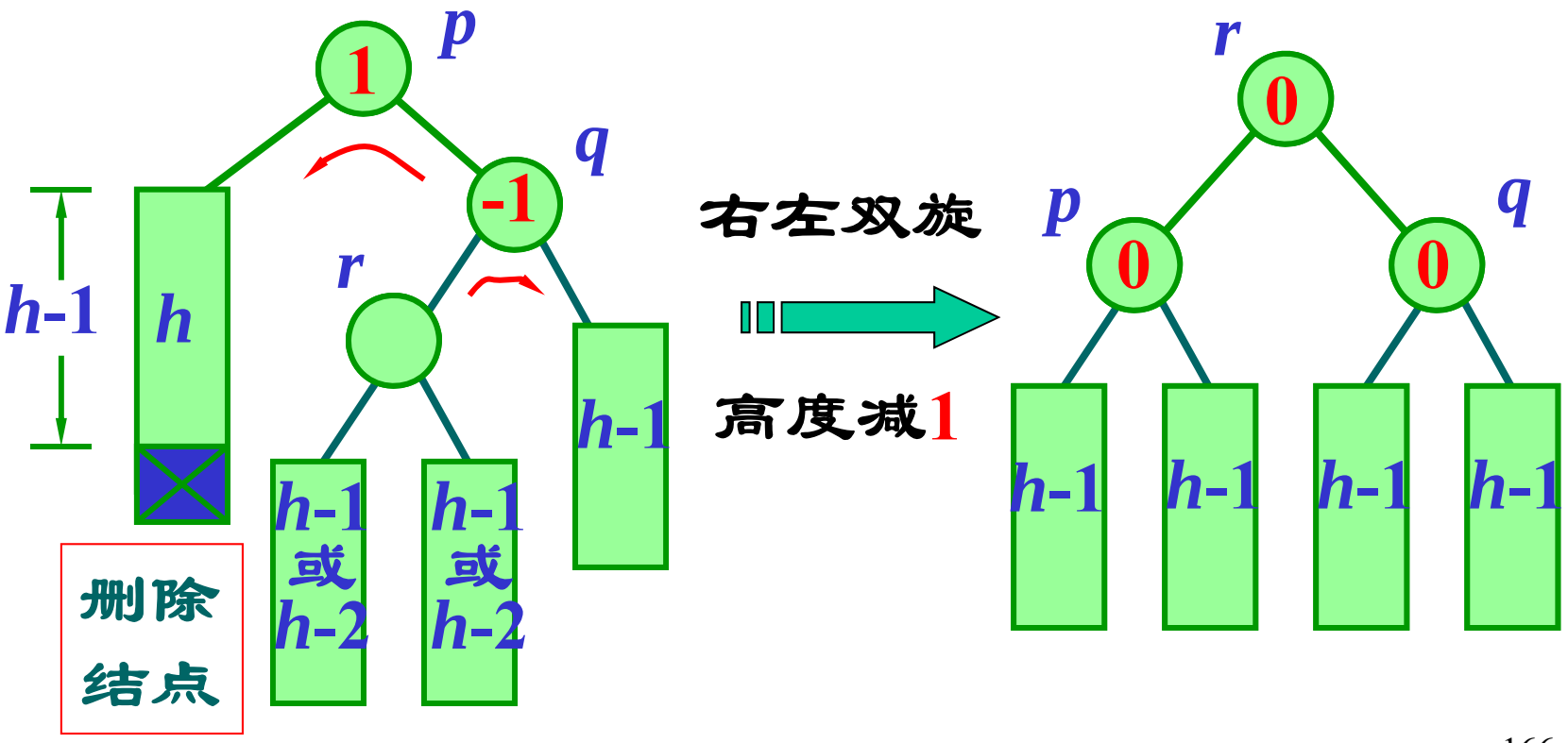
- **case 3a**：如果*q*（较高的子树）的**balance**为**0**，执行一个单旋转来恢复结点*p*的平衡，置**shorter**为**false**。



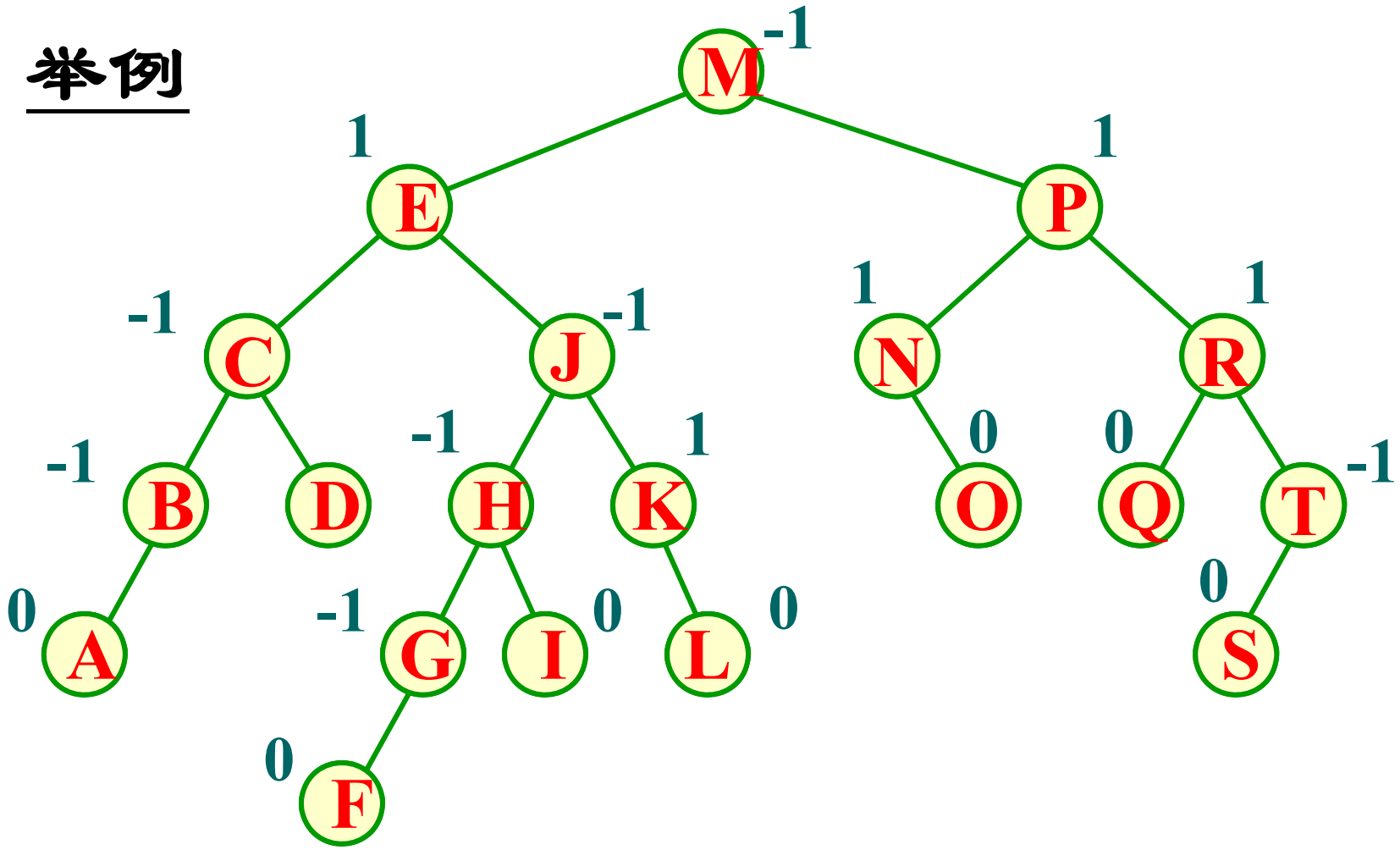
- **case 3b：如果*q*的balance与*p*的balance相同，则执行一个单旋转来恢复平衡，结点*p*和*q*的balance均改为0，同时置shorter为true。**



- **case 3c**：如果*p*与*q*的balance相反，则执行一个双旋转来恢复平衡，先围绕*q*转再围绕*p*转。新根结点的balance置为0，其它结点的balance相应处理，同时置shorter为true。

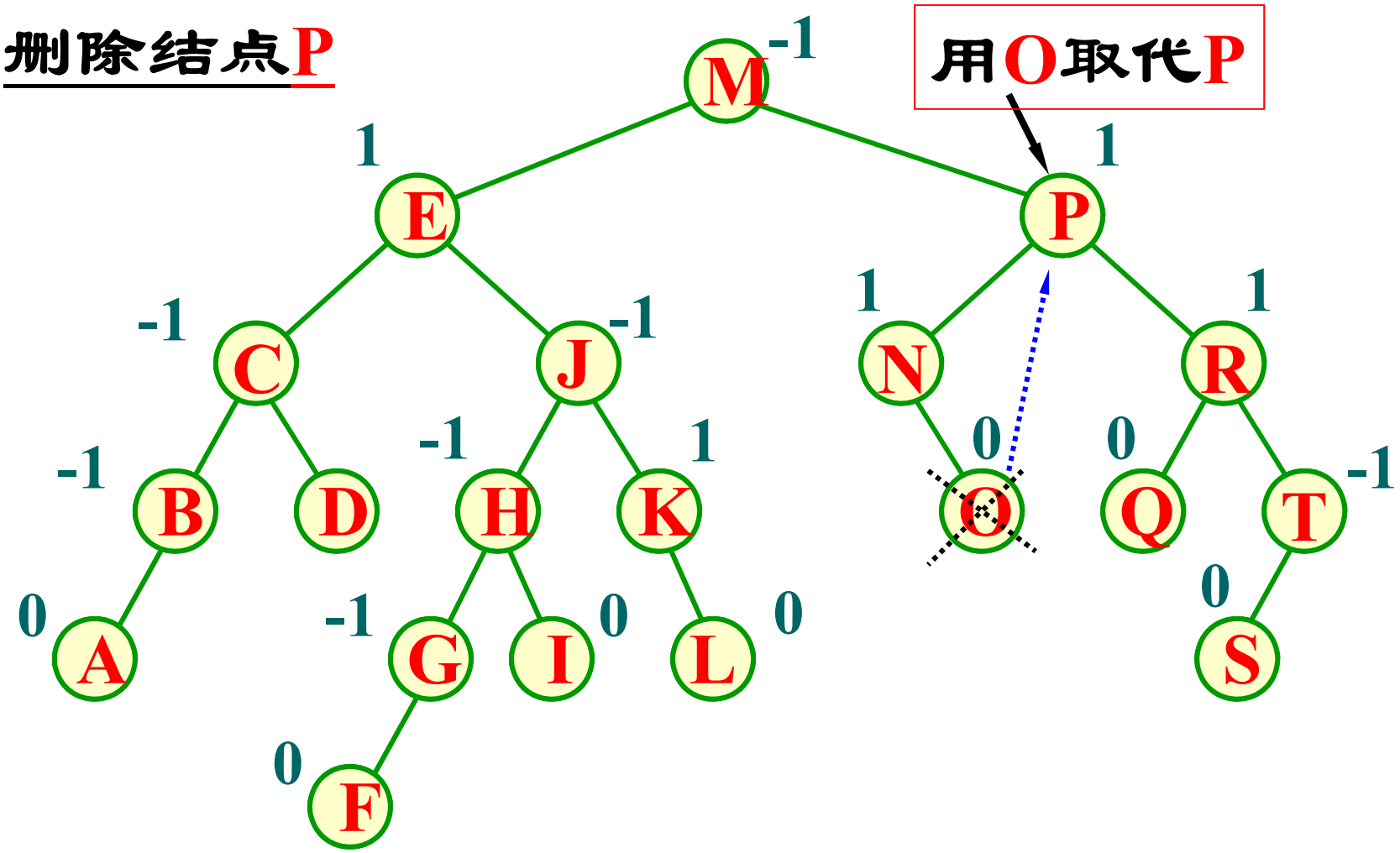


举例



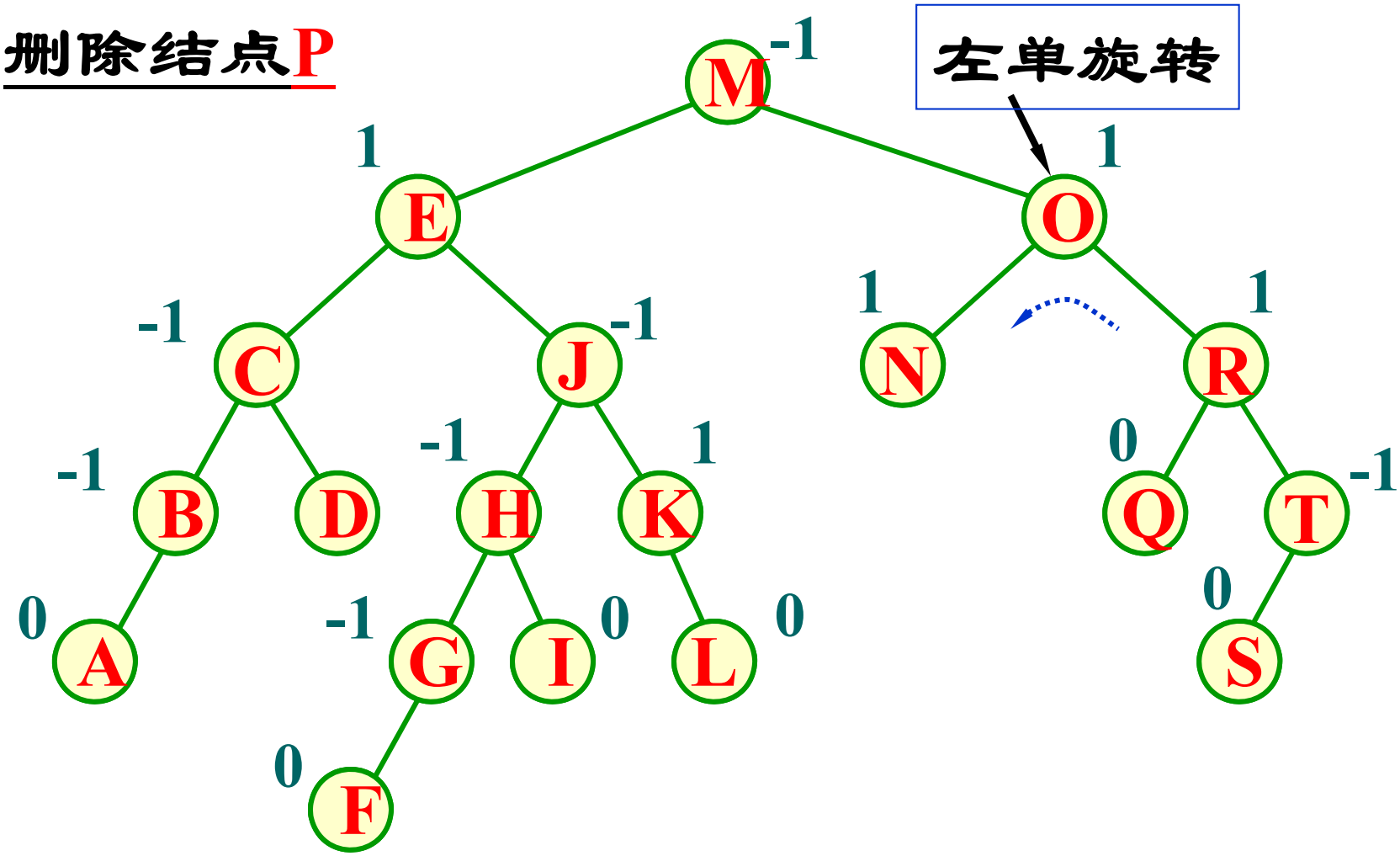
树的初始状态

删除结点P



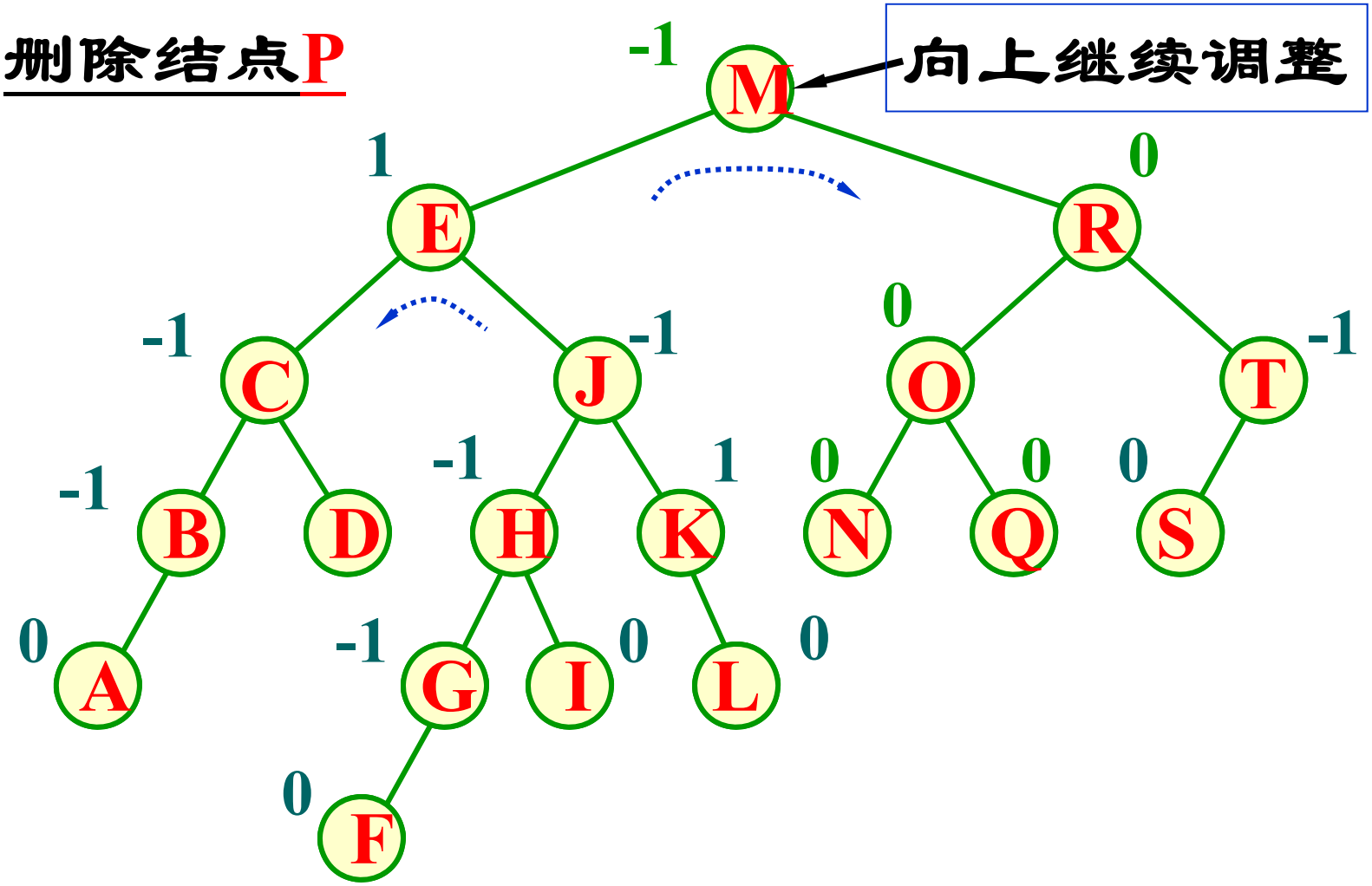
寻找结点P在中序下的直接前驱O，用O顶替P，删除O。

删除结点P



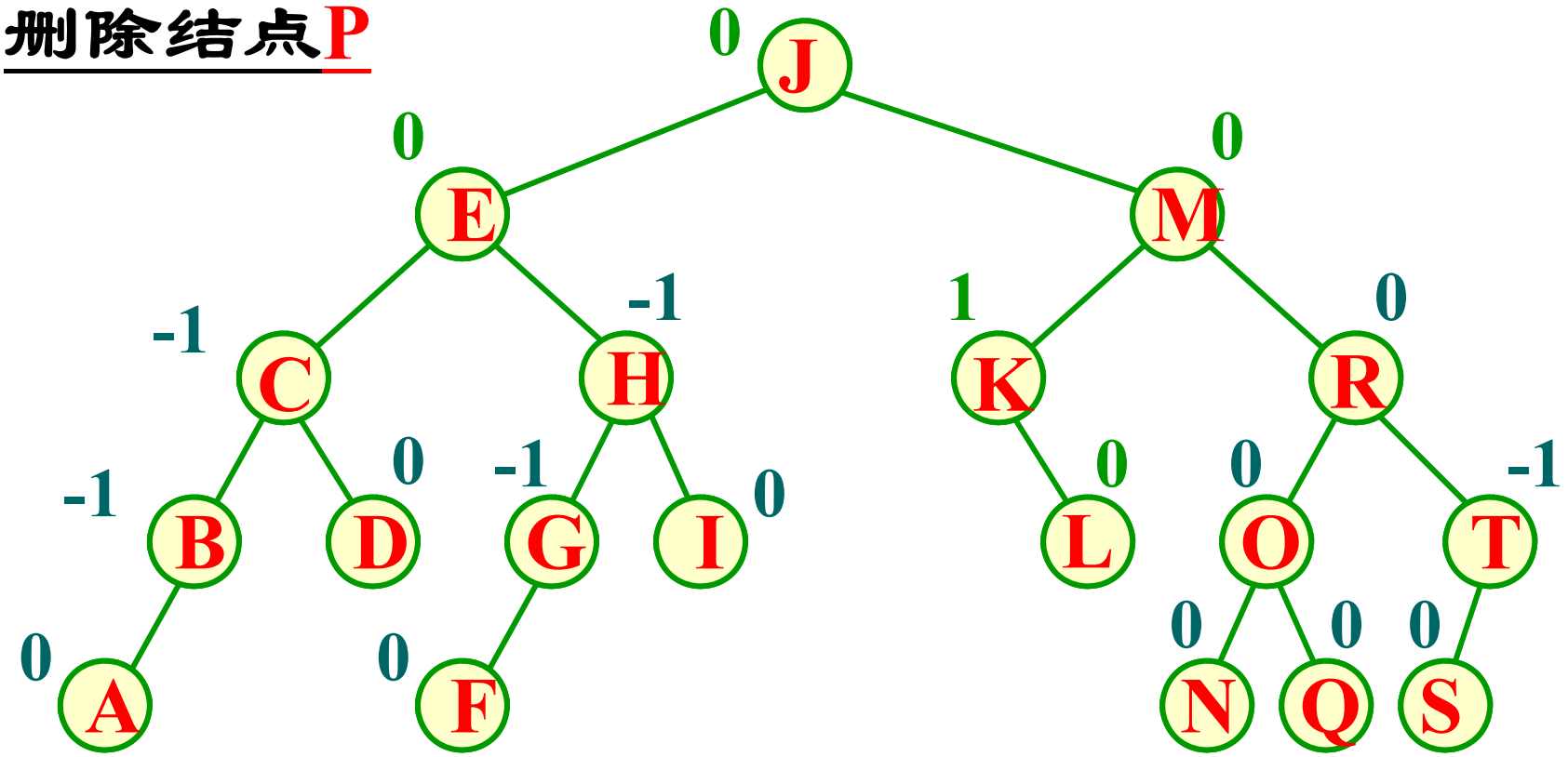
O与R的平衡因子同号，以R为旋转轴做左单旋转，M的子树高度减1。

删除结点P



M的子树高度减1，M发生不平衡，M与E的平衡因子反号，做左右双旋转。

删除结点P



程序7.22

- **while** (p!=NULL) {
- … //寻找结点，同时把经过的结点压栈，循环结束时，pr为要删除结点p的双亲，pr在栈顶。
- **if** (p==NULL) **return** false; //未找到结点，返回
- … //处理被删除结点有两个子女的情况，转变成删除只有至多1个子女的结点的情况，最后仍然是pr为要删除结点p的双亲，pr在栈顶。
- **if** (p->left!=NULL) q=p->left;
- … //让q指向p的孩子结点，特殊情况是删除根结点，则也让ptr与q指向相同的结点。
- **else** {
- … //调整删除p之后的pr和q之间的连接。
- **while** (st.IsEmpty( )==false) {
- … //调整pr的平衡因子，并且根据pr是其双亲ppr的左孩子还是右孩子决定dd的取值，以记住经过旋转修改的pr应该连接在ppr的左孩子还是右孩子。
- **if** (pr->bf ==1) || pr->bf==-1) **break;**
- … //根据pr的平衡因子和其较高的子树的平衡因子决定旋转类型，旋转，调整平衡因子。有可能要回到循环开头继续回溯。
- **if** (dd==-1) ppr->left=pr;
- … //根据dd决定经过旋转修改的pr应该连接在ppr的左孩子还是右孩子。
- 删除p,结束。

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，则插入一个新结点的时间是$O(h)$。对于AVL树来说，h多大？
- 设N_h是高度为h的AVL树的最小结点数，根的一棵子树的高度为$h-1$，另一棵子树的高度为$h-2$，这两棵子树高度平衡。

  因此有
  - $N_{-1}=0$（空树）
  - $N_0=1$（仅有根结点）
  - $N_h=N_{h-1}+N_{h-2}+1$，$h>0$

- 可以证明，对于$h \geq 0$，有$N_h = F_{h+3} - 1$成立。
- 有n个结点的AVL树的高度不超过
  $1.44 * \log_2(n+1) - 1$
- 在AVL树删除一个结点并做平衡化旋转所需时间为$O(\log_2 n)$。
- 二叉搜索树适合于组织在内存中的较小的索引（或目录），对于存放在外存中的较大的文件系统，用二叉搜索树来组织索引不太合适。
- 在文件检索系统中大量使用的是用B-树或B+树做文件索引。

7.6 Hash查找

- **Hash，音译“哈希”或意译“散列”，是另一种重要的查找方法。如果经常要从一个结点集合中找出具有给定键值的结点，那么采用Hash存储和Hash查找是一种好办法。**
- **Hash查找的前提是数据采用Hash存储。即首先通过Hash函数，将键值映射到表中某个位置来存储数据元素，然后根据键值用同样的方法直接访问。**
- **Hash函数的作用是在结点存储地址*address*与其关键字*key*之间建立一个确定的对应关系，使每个关键字与一个唯一存储地址相对应：*address*=hash(*key*)。**
  - 在搜索时，先对结点的关键字进行函数计算，把函数值当做结点的存储位置，并按此位置取结点做比较。若关键字相等，则搜索成功。
  - 在存放结点时，依相同函数计算存储位置，并按此位置存放。

例

- 含有6个结点的线性表，其中键值分别为：7、11 、12、16、20、22，使用有10个结点的数组T[0..9]作为Hash表存储，Hash函数h(*key*)=*key*%7，Hash表的存储情况如图7-27所示。如找键值为22的结点，计算22%7=1就能在T[1]中找到。

0	7
1	22
2	16
3	
4	11
5	12
6	20
7	
8	
9	

- Hash函数是一个压缩映象函数。一般关键字集合比Hash表地址集合大得多。因此有可能经过Hash函数的计算，把不同的关键码映射到同一个Hash地址上。
  - 即使把存放结点的Hash表的容量取得很大，有时也难以找到一个足够简单的一一对应函数。因此，在绝大多数情况下使用的Hash函数都不保证一一对应。
    - 在图7-27中，如果再插入一个键值为8的结点，会发现其对应的存储位置T[1]已经被键值为22的结点占用，即不同的关键字映射到同一个Hash地址上，这种情况称为冲突(Collision)。
      - 这些产生冲突的Hash地址相同的不同关键字为同义词(Synonym)。
- Hash方法包括两个主要问题
  - 选取Hash函数
  - 选取解决冲突的方法

散列 (Hashing)

- 在现实中经常遇到按给定的值进行查询的事例。为此，必须考虑在记录的存放位置和用以标识它的数据项(称为关键码)之间的对应关系，选择适当的数据结构，很方便地根据记录的关键码检索到对应记录的信息。
- 表项的存放位置及其关键码之间的对应关系可以用一个二元组表示：

  (关键码***key***，表项位置指针***addr***)
- 这个二元组构成搜索某一指定表项的索引项。
- 考虑到搜索效率，可以考虑散列表结构。

静态散列方法

- 散列方法在表项存储位置与其关键码之间建立一个确定的对应函数关系***Hash( )***，使每个关键码与结构中一个唯一存储位置相对应：

  Address＝Hash(Rec.key)

- 在搜索时，先对表项的关键码进行函数计算，把函数值当做表项的存储位置，在结构中按此位置取表项比较。若关键码相等，则搜索成功。在存放表项时，依相同函数计算存储位置，并按此位置存放。此方法称为散列方法。

- 在散列方法中使用的转换函数叫做散列函数。按此方法构造出来的表或结构就叫做散列表。
- 使用散列方法进行搜索不必进行多次关键码的比较，搜索速度比较快，可以直接到达或逼近具有此关键码的表项的实际存放地址。
- 散列函数是一个压缩映象函数。关键码集合比散列表地址集合大得多。因此有可能经过散列函数的计算，把不同的关键码映射到同一个散列地址上，这就产生了冲突。
- 示例：有一组表项，其关键码分别是

12361，07251，03309，30976

采用的散列函数是

$$hash(x)=x\%73+13420$$

其中，“%”是除法取余操作。

则有：$hash(12361)=hash(07250)=hash(03309)=hash(30976)=13444$。就是说，对不同的关键码，通过散列函数的计算，得到同一散列地址。我们称这些产生冲突的散列地址相同的不同关键码为同义词。

- 由于关键码集合比地址集合大得多，冲突很难避免。所以对于散列方法，需要讨论以下两个问题：
  - 对于给定的一个关键码集合， 选择一个计算简单且地址分布比较均匀的散列函数，避免或尽量减少冲突；
  - 拟订解决冲突的方案。

7.6.1 Hash函数

- 构造Hash函数时考虑的一般原则有：
  - Hash函数应是简单的、能在短时间内计算出结果。
  - Hash函数的定义域必须包括需要存储的全部关键码，如果Hash表允许有m个地址时，其值域必须在0 到m-1之间。
  - 关键字集合中的关键字，经Hash函数映射到地址集合中任何一个地址的概率都是相等的。
- Hash函数的种类很多，这里只介绍一些常见的Hash函数构造方法。

散列函数

构造散列函数时的几点要求：

- 散列函数应是简单的，能在较短的时间内计算出结果。
- 散列函数的定义域必须包括需要存储的全部关键码，如果散列表允许有*m*个地址时，其值域必须在0到*m*-1之间。
- 散列函数计算出来的地址应能均匀分布在整个地址空间中：若*key*是从关键码集合中随机抽取的一个关键码，散列函数应能以同等概率取0到*m*−1中的每一个值。

① 直接定址法

此类函数取关键码的某个线性函数值作为散列地址：

$Hash(key)=a*key+b$　{a, b为常数}

这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键码集合的大小相同。

示例：有一组关键码如下：{ 942148, 941269, 940527, 941630, 941805, 941558, 942047, 940001 }。散列函数为

$$Hash(key)=key-940000$$

$Hash(942148)=2148$　　$Hash(941269)=1269$

$Hash(940527)=527$　　$Hash(941630)=1630$

$Hash(941805)=1805$　　$Hash(941558)=1558$

$Hash(942047)=2047$　　$Hash(940001)=1$

可以按计算出的地址存放记录。

② 数字分析法

- 设有n个d位数，每一位可能有r种不同的符号。这r种不同符号在各位上出现的频率不一定相同。可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。
- 计算各位数字中符号分布均匀度λ_k的公式：

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

- 其中，α_i^k表示第i个符号在第k位上出现的次数，n/r表示各种符号在n个数中均匀出现的期望值。计算出的λ_k值越小，表明在该位（第k位）各种符号分布得越均匀。

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

①位，$\lambda_1 = 57.60$

②位，$\lambda_2 = 57.60$

③位，$\lambda_3 = 17.60$

④位，$\lambda_4 = 5.60$

⑤位，$\lambda_5 = 5.60$

⑥位，$\lambda_6 = 5.60$

若散列表地址范围有3位数字，取各关键码的④⑤⑥位做为记录的散列地址。

- 数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况，它完全依赖于关键码集合。如果换一个关键码集合，选择哪几位要重新决定。

③ 除留余数法

- 设散列表中允许地址数为***m***，取一个不大于***m***，但最接近于或等于***m***的质数***p***作为除数，利用以下函数把关键码转换成散列地址：

$$hash(key)=key\%p,\quad p\le m$$

- 其中，“%”是整数除法取余的运算，要求这时的质数***p***不是接近**2**的幂。

- 示例：有一个关键码*key*=962148，散列表大小*m*=25，即*HT*[25]。取质数*p*=23。散列函数*hash*(*key*)=*key*%*p*。则散列地址为

  hash(962148)=962148%23=12。

- 可以按计算出的地址存放记录。需要注意的是，使用上面的散列函数计算出来的地址范围是0到22。因此，从23到24这几个散列地址实际上在一开始是不可能用散列函数计算出来的，只可能在处理冲突时达到这些地址。

④ 平方取中法

- 此方法在词典处理中使用十分广泛。
- 它先计算构成关键码的标识符的内码的平方，然后按照散列表的大小取中间的若干位作为散列地址。
- 设标识符可以用一个计算机字长的内码表示。因为内码平方数的中间几位一般是由标识符所有字符决定，所以对不同的标识符计算出的散列地址大多不相同。
- 在平方取中法中，一般取散列地址为2的某次幂。例如，若散列地址总数取为$m=8^r$，则对内码的平方数取中间的r位。如果$r=3$，所取得的散列地址参看图的最右一列。

标识符	内码	内码的平方	散列地址
A	01	01	001
A1	0134	20420	042
A9	0144	23420	342
B	02	4	004
DMAX	04150130	215264436171 00	443
DMAX1	0415013034	526447352215 1420	352
AMAX	01150130	135423617100	236
AMAX1	0115013034	345424652215 1420	652

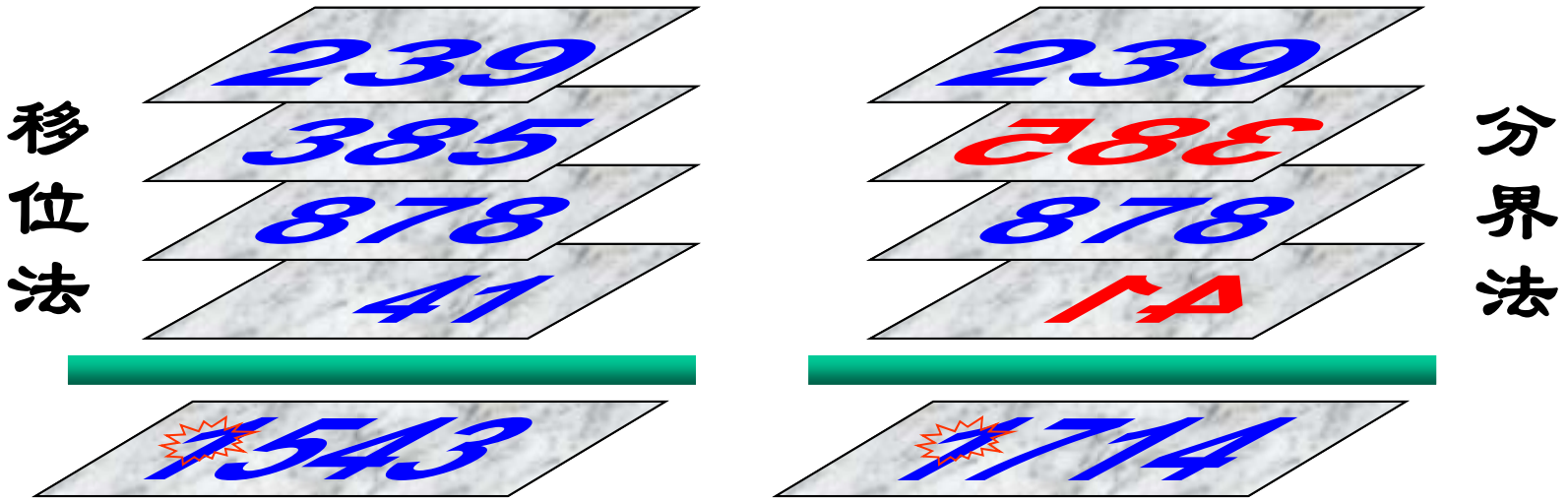
⑤ 折叠法

- 此方法把关键码自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键码的记录的散列地址。
- 有两种叠加方法：
  - 移位法 — 把各部分的最后一位对齐相加；
  - 分界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址。

- 示例：设给定的关键码为*key*=2393858784l，若存储空间限定3位，则划分结果为每段3位。上述关键码可划分为4段：

  239　385　878　41

- 把超出地址位数的最高位删去，仅保留最低的3位，做为可用的散列地址。



- 一般当关键码的位数很多，而且关键码每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。
- 以上介绍了几种常用的散列函数。在实际工作中应根据关键码的特点，选用适当的方法。有人曾用“轮盘赌”的统计分析方法对它们进行了模拟分析，结论是平方取中法最接近于“随机化”。

处理冲突的闭散列方法

因为任一种散列函数也不能避免产生冲突，因此选择好的解决冲突的方法十分重要。

为了减少冲突，对散列表加以改造。若设散列表***HT***有***m***个地址，将其改为***m***个桶。其桶号与散列地址一一对应，第***i*** **(0≤*i*<*m*)**个桶的桶号即为第***i***个散列地址。

每个桶可存放***s***个表项，这些表项的关键码互为同义词。如果对两个不同表项的关键码用散列函数计算得到同一个散列地址，就产生了冲突，它们可以放在同一个桶内的不同位置。

- 只有当桶内所有s个表项位置都放满表项后再加入表项才会产生溢出。
- 通常桶的大小s取的比较小，因此在桶内大多采用顺序搜索。
- 闭散列也叫做开地址法。在闭散列情形，所有的桶都直接放在散列表数组中。因此每个桶只有一个表项(s=1)。
- 若设散列表中各桶的编址为0到m-1，当要加入一个表项R_2时，用它的关键码$R_2.key$，通过散列函数$hash(R_2.key)$的计算，得到它的存放桶号 j。

- 但在存放时发现此桶已被另一个表项R_1占据，发生了冲突，必须处理冲突。为此，需把R_2存放到表中“下一个”空桶中。如果表未被装满，则在允许的范围内必定还有空桶。

(1) 线性探查法 (Linear Probing)

假设给出一组表项，它们的关键码为**Burke**，**Ekers**，**Broad**，**Blum**，**Attlee**，**Alton**，**Hecht**，**Ederly**。采用的散列函数是：取其第一个字母在字母表中的位置。

Hash(x)=ord(x)-ord('A')

//***ord( )***是求字符内码的函数

- 可得 *Hash*(Burke)=1　　*Hash*(Ekers)=4
  Hash(Broad)=1　　*Hash*(Blum)=1
  Hash(Attlee)=0　　*Hash*(Hecht)=7
  Hash(Alton)=0　　*Hash*(Ederly)=4
- 设散列表 *HT*[31], *m*=31。采用线性探查法处理冲突, 则散列结果如图所示。

0	1	2	3	4
Attlee	Burke	Broad	Blum	Ekers
(1)	(1)	(2)	(3)	(1)

5	6	7	8	9
Alton	Ederly	Hecht		
(6)	(3)	(1)		

- 需要搜索或加入一个表项时，使用散列函数计算桶号：

$$H_0=hash(key)$$

- 一旦发生冲突，在表中顺次向后寻找“下一个”空桶H_i的递推公式为：

$$H_i=(H_{i-1}+1)\%m，\ i=1, 2, \ldots, m-1$$

即用以下的线性探查序列在表中寻找“下一个”空桶的桶号：

$$H_0+1, H_0+2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

亦可写成如下的通项公式：

$$H_i=(H_0+i)\%m，\ i=1, 2, \ldots, m-1$$

- 当发生冲突时，探查下一个桶。当循环***m*-1**次后就会回到开始探查时的位置，说明待查关键码不在表内，而且表已满，不能再插入新关键码。
- 用平均搜索长度***ASL*(*Average Search Length*)**衡量散列方法的搜索性能。
- 根据搜索成功与否，它又有搜索成功的平均搜索长度***ASLsucc***和搜索不成功的平均搜索长度***ASLunsucc***之分。
- 搜索成功的平均搜索长度***ASLsucc***是指搜索到表中已有表项的平均探查次数。它是找到表中各个已有表项的探查次数的平均值。

- 搜索不成功的平均搜索长度*ASLunsucc*是指在表中搜索不到待查的表项，但找到插入位置的平均探查次数。它是表中所有可能散列到的位置上要插入新元素时为找到空桶的探查次数的平均值。
- 在使用线性探查法对示例进行搜索时，搜索成功的平均搜索长度为：

$$ASLsucc=\frac{1}{8}\sum_{i=1}^{8}Ci=\frac{1}{8}(1+1+2+3+1+6+3+1)=\frac{18}{8}.$$

- 搜索不成功的平均搜索长度为：

$$ASLunsucc=\frac{9+8+7+6+5+4+3+2+23}{31}=\frac{67}{31}.$$

- 下面是用线性探查法在散列表*ht*中搜索给定值*x*的算法。如果查到某一个*j*，使得

  ht[j].info==Active && ht[j].Element==x

  则搜索成功；否则搜索失败。造成失败的原因可能是表已满，或者是原来有此表项但已被删去，或者是无此表项且找到空桶。

```
class HashTable {
//用线性探查法处理冲突时散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };
    HashTable( ) : TableSize(DefaultSize)
                   { AllocateHt( );  CurrentSize=0; }
```

```
  ~HashTable( ) { delete [ ] ht; }  //析构函数
  const HashTable & operator =  //表复制
                        (const HashTable &ht2);
  int Find(const Type &x);  //搜索
  int Insert(const Type &x);  //插入
  int Remove(const Type &x);  //删除
  int IsIn(const Type &x)  //判存在
      { return (i=Find(x))>=0?1:0; }
  void MakeEmpty( );  //置空
private:
  struct HashEntry {  //散列表表项
    Type Element;  //表项关键码
    KindOfEntry info;  //三种状态
```

```
        int operator== (HashEntry &);  //判相等
        int operator != (HashEntry &);  //判不等
        HashEntry( ) : info(Empty) { }  //构造函数
        HashEntry(const Type& E, KindOfEntry
                i=Empty) : Element(E), info(i) { }
    };
    enum { DefualtSize=11; }
    HashEntry *ht;  //散列表数组
    int CurrentSize, TableSize;  //当前及最大桶数
    void AllocateHt( )  //分配空间
                { ht=new HashEntry[TableSize]; }
    int FindPos(const Type &x) const;
} //散列函数
```

```
template <class Type> int HashTable <Type>::
Find(const Type &x) {
//线性探查法的搜索算法，函数返回找到位置。
//若返回负数可能是空位，若为-TableSize则失败。
  int i= FindPos(x), j=i;  //计算散列地址
  while (ht[j].info!=Empty &&
                        ht[j].Element!=x) {
    j=(j+1)%TableSize;  //冲突，找空桶
    if (j==i)  return -TableSize;  //失败，表满
  }
  if (ht[j].info==Active)  return j;  //成功
  else -j;  //失败
}
```

- 在利用散列表进行各种处理之前，必须首先将散列表中原有的内容清掉。只需将表中所有表项的**info**域置为**Empty**即可。
- 散列表存放的表项不应有重复的关键码。在插入新表项时，如果发现表中已经有关键码相同的表项, 则不再插入。
- 在闭散列情形下不能真正删除表中已有表项。删除表项会影响其他表项的搜索。若把关键码为**Broad**的表项真正删除，把它所在位置的**info**域置为**Empty**，以后在搜索关键码为**Blum**和**Alton**的表项时就查不下去，会错误地判断表中没有关键码为**Blum**和**Alton**的表项。

- 若想删除一个表项，只能给它做一个删除标记**deleted**进行逻辑删除，不能把它真正删去。
- 逻辑删除的副作用是：在执行多次删除后，表面上看起来散列表很满，实际上有许多位置没有利用。

```
template <class Type>
void HashTabe<Type> :: MakeEmpty( ) {
//置表中所有表项为空
    for (int i=0; i<TableSize; i++)
        ht[i].info=Empty;
    CurrentSize=0;
}
```

```
template <class Type>
const HashTable <Type> & HashTable <Type> ::
operator = (const HashTable <Type> &ht2) {
//重载函数：从散列表ht2复制到当前散列表
    if (this!=&ht2) {
       delete [ ] ht;
       TableSize=ht2.TableSize;  AllocateHt( );
       for (int i=0; i<TableSize; i++)
            ht[i]=ht2.ht[i];
       CurrentSize=ht2.CurrentSize;
    }
    return *this;
}
```

```
template <class Type> int HashTable<Type>::
Insert( const Type &x) {
//将新表项x插入到当前的散列表中
    if ((int i=Find(x))>=0)  return 0;  //不插入
    else if (i!=-TableSize &&
             ht[-i].info!=Active) {  //在-i处插入x
                ht[-i].Element=x;
                ht[-i].info=Active;
                CurrentSize++;
                return 1;
            }
            else return 0;
}
```

```
template <class Type> int HashTable <Type> ::
Remove(const Type &x) {
//在当前散列表中删除表项x
    if ((int i=Find(x))>=0) {  //找到，删除
      ht[i].info=deleted;  //做删除标记
      CurrentSize--;
      return 1;
    }
    else return 0;
}
```

线性探查方法容易产生“堆积”，不同探查序列的关键码占据可用的空桶，为寻找某一关键码需要经历不同的探查序列，导致搜索时间增加。

算法分析

- 设散列表的装填因子为$\alpha=n/(s*m)$，其中n是表中已有的表项个数，s是每个桶中最多可容纳表项个数，m是表中的桶数。
- 可用α表明散列表的装满程度。α越大，表中表项数越多，表装得越满，发生冲突可能性越大。
- 通过对线性探查法的分析可知，为搜索一个关键码所需进行的探查次数的期望值P大约是$(2-\alpha)/(2-2\alpha)$。虽然平均探查次数较小，但在最坏情况下的探查次数会相当大。

(2) 二次探查法 (Quadratic Probing)

- 为改善“堆积”问题，减少为完成搜索所需的平均探查次数，可使用二次探查法。
- 通过某一个散列函数对表项的关键码x进行计算，得到桶号，它是一个非负整数。

  $H_0=hash(x)$
- 二次探查法在表中寻找“下一个”空桶的公式：

  $H_i=(H_0+i^2)\%m$,

  $H_i=(H_0-i^2)\%m$, $i=1, 2, \ldots, (m-1)/2$
- 式中的m是表的大小，它应是一个值为$4k+3$的质数，其中k是一个整数。如 3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, …。

- 探查序列如 $H_0, H_0+1, H_0-1, H_0+4, H_0-4, \ldots$。
- 在做$(H_0-i^2)\%m$的运算时，当$H_0-i^2<0$时，运算结果也是负数。实际算式可改为

  $j=(H_0-i^2)\%m$， if ($j<0$) $j+=m$;
- 示例：给出一组关键码{Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly}。

  散列函数为：*Hash* (*x*)=*ord* (*x*)-*ord* ('*A*')
- 用它计算可得

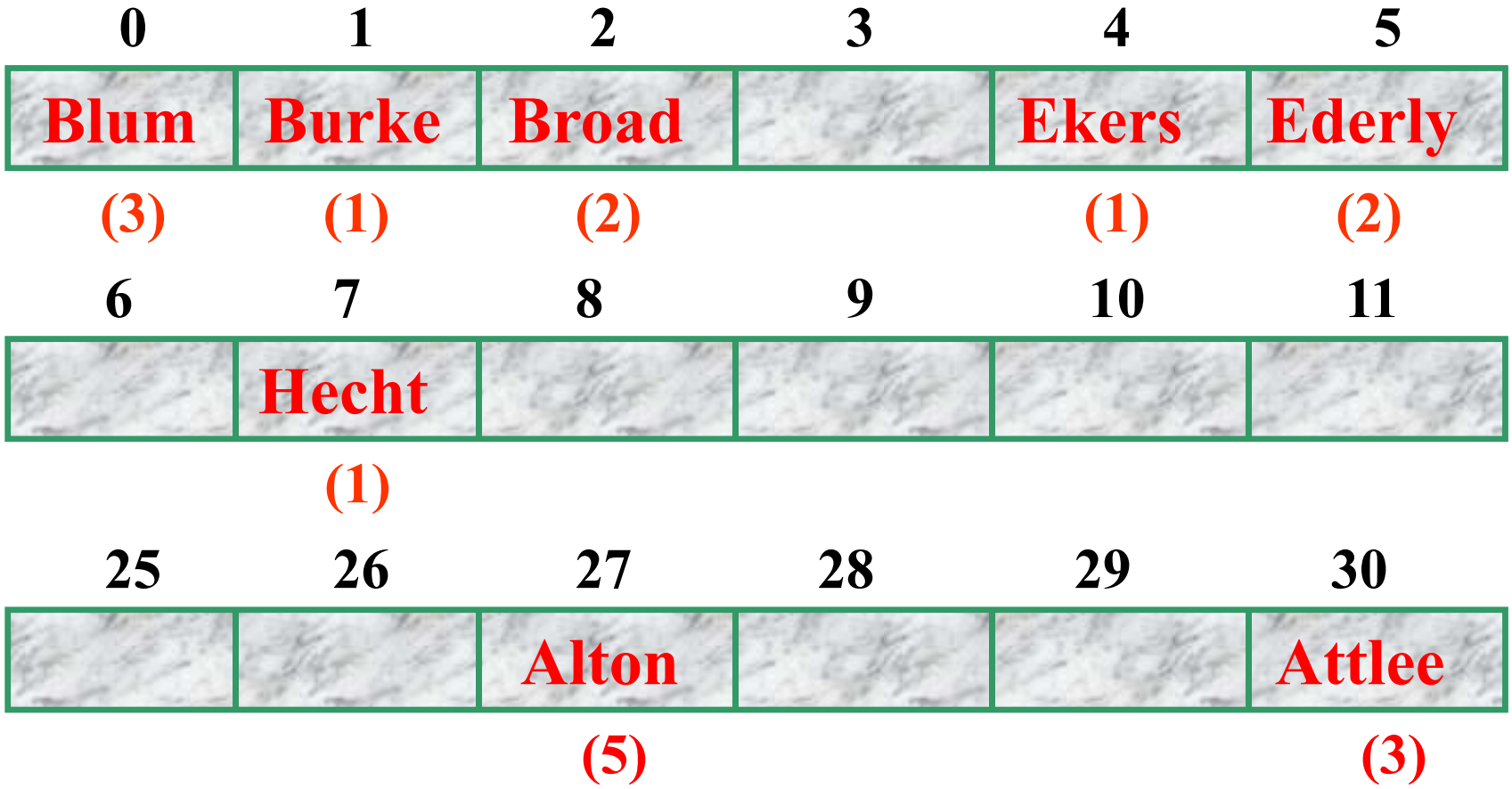
  Hash(Burke)=1　　*Hash*(Ekers)=4

  Hash(Broad)=1　　*Hash*(Blum)=1

  Hash(Attlee)=0　　*Hash*(Hecht)=7

  Hash(Alton)=0　　*Hash*(Ederly)=4

- 因为可能桶号是0~30，取满足$4k+3$的质数，表的长度为$TableSize=31$，利用二次探查法得到的散列结果如图所示。



利用二次探查法处理溢出

- 使用二次探查法处理冲突时的搜索成功的平均搜索长度为：

$$ASLsucc=\frac{1}{8}\sum_{i=1}^{8}Ci=\frac{1}{8}(3+1+2+1+2+1+5+3)=\frac{18}{8}.$$

- 搜索不成功的平均搜索长度为：

$$ASLunsucc=\frac{1}{31}(6+5+2+3+2+2+25)=\frac{45}{31}$$

设散列表桶数为***m***，待查关键码为***x***，第一次通过散列函数计算出的桶号为$H_0=hash(x)$。当发生冲突时，第***i-1***次和第***i***次计算出来的“下一个”桶号分别为：

$$H_{i-1}^{(0)} = (H_0 + (i-1)^2) \% m,$$
$$H_{i-1}^{(1)} = (H_0 - (i-1)^2) \% m.$$
$$H_i^{(0)} = (H_0 + i^2) \% m,$$
$$H_i^{(1)} = (H_0 - i^2) \% m.$$

相减，可以得到：

$$H_i^{(0)} - H_{i-1}^{(0)} = (2*i-1) \% m,$$
$$H_i^{(1)} - H_{i-1}^{(1)} = (-2*i+1) \% m.$$

从而

$$H_i^{(0)} = (H_{i-1}^{(0)} + 2*i-1) \% m,$$
$$H_i^{(1)} = (H_{i-1}^{(1)} - 2*i+1) \% m.$$

- 只要知道上一次的桶号$H_{i-1}^{(0)}$和$H_{i-1}^{(1)}$，当i增加1时可以从$H_{i-1}^{(0)}$和$H_{i-1}^{(1)}$简单地导出$H_i^{(1)}$和$H_i^{(1)}$，不需要每次计算 i 的平方。
- 在冲突处理算法**Find**中，首先求出H_0作为当前桶号**CurrentPos**，当发生冲突时求“下一个”桶号，i=1。
- 此时用一个标志***odd***控制是加 i^2 还是减 i^2 。
  - 若***odd***==0加i^2，并置***odd***=1；
  - 若***odd***==1减i^2，并置***odd***=0。
- 下次i进一后，又可由***odd***控制先加后减。

处理冲突的算法

```
template <class Type>
int HashTable<Type> :: Find(const Type &x) {
    int pos0, pos1, i=0, odd=0;
    int CurrentPos=pos0=pos1=HashPos(x);
    //初始桶号
    while (ht[CurrentPos].info!=Empty &&
                ht[CurrentPos].Element!=x) {  //冲突
        if (!odd) {  //odd==0加i²
            pos0=(pos0+2*i-1)%TableSize;
            CurrentPos=pos0;  odd=1;
        }
        else {  //odd==1减i²
```

```
            pos1=(pos1-2*i+1)%TableSize;
            CurrentPos=pos1;  odd=0;
        }
    }
    LastFindOK=ht[CurrentPos].info==Active;
    return CurrentPos;
}
```

可以证明，当表的长度**TableSize**为质数且表的装填因子α不超过**0.5**时，新的表项***x***一定能够插入，且任何一个位置不会被探查两次。只要表中至少有一半空，就不会有表满问题。

- 在搜索时可以不考虑表满的情况；但在插入时必须确保表的装填因子α不超过**0.5**。如果超出，必须将表长度扩充一倍，进行表的分裂。
- 在删除一个表项时，为确保搜索链不致中断，也只能做表项的逻辑删除，即将被删表项的标记**info**改为**Deleted**。

(3) 双散列法

- 使用双散列方法时，需要两个散列函数。
- 第一个散列函数***Hash( )***按表项的关键码***key***计算表项所在的桶号H_0=***Hash(key)***。

- 一旦冲突，利用第二个散列函数***ReHash( )***计算该表项到达“下一个”桶的移位量。它的取值与***key***的值有关，要求它的取值应是小于地址空间大小**TableSize**，且与**TableSize**互质的正整数。
- 若设表的长度为***m*=TableSize**，则在表中寻找“下一个”桶的公式为：

  $j=H_0=Hash(key)$，$p=ReHash(key)$；

  $j=(j+p)\%m$；

  p是小于***m***且与***m***互质的整数

- 利用双散列法，按一定的距离，跳跃式地寻找“下一个”桶，减少了“堆积”的机会。
- 双散列法的探查序列也可写成：

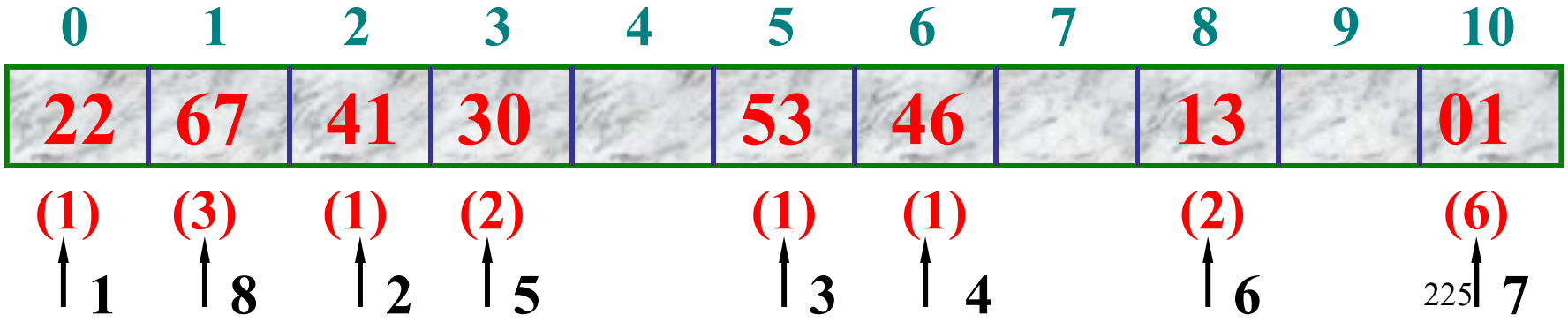
  $H_i=(H_0+i*ReHash(key))\%m,$

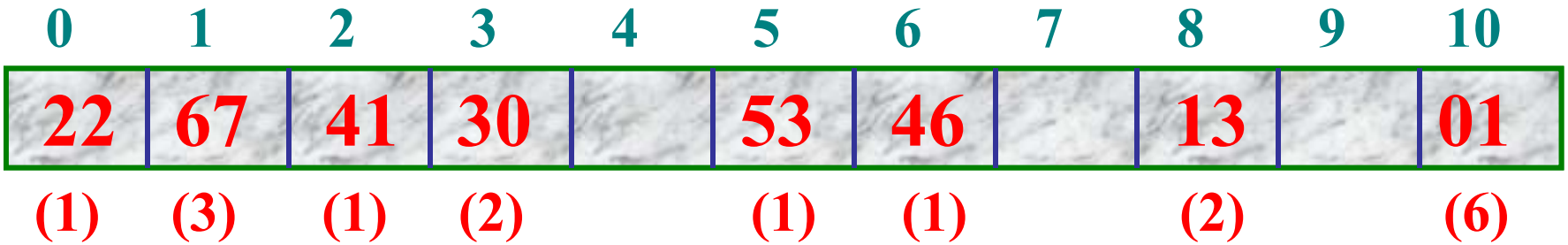
  $i=1, 2, \ldots, m-1$
- 最多经过$m-1$次探查，它会遍历表中所有位置，回到H_0位置。
- 示例：给出一组表项关键码{22, 41, 53, 46, 30, 13, 01, 67}。散列函数为：

  $Hash(x)=(3x)\%11$。
- 散列表为 $HT[0..10]$，$m=11$。因此，再散列函数为$ReHash(x)=(7x)\%10+1$。

$H_i=(H_{i-1}+(7x)\%10+1)\%11,\ i=1, 2, \ldots$

- $H_0(22)=0$ $H_0(41)=2$ $H_0(53)=5$
  $H_0(46)=6$ $H_0(30)=2$ 冲突 $H_1=(2+1)=3$
  $H_0(13)=6$ 冲突 $H_1=(6+2)=8$
  $H_0(01)=3$ 冲突 $H_1=(3+8)=0$ 冲突
  $H_2=(0+8)=8$ 冲突 $H_3=(8+8)=5$ 冲突
  $H_4=(5+8)=2$ 冲突 $H_5=(2+8)=10$
  $H_0(67)=3$ 冲突 $H_1=(3+10)=2$ 冲突
  $H_2=(2+10)=1$





- 搜索成功的平均搜索长度

$$ASLsucc = \frac{1}{8}(1+3+1+2+1+1+2+6) = \frac{17}{8}$$

- 搜索不成功的平均搜索长度
  - 每一散列位置的移位量有10种：1, 2, …, 10。
    先计算每一散列位置各种移位量情形下找到下一个空位的比较次数，求出平均值；
  - 再计算各个位置的平均比较次数的总平均值。

- ***Rehash*( )的取法很多。例如，当*m*是质数时，可定义**
  - ***ReHash*(*key*)=*key*%(*m*-2) +1**
  - ***ReHash*(*key*)=⌊*key*/*m*⌋ %(*m*-2)+1**
- **当*m*是2的方幂时，*ReHash*(*key*)可取从0到 *m*-1中的任意一个奇数。**

处理冲突的开散列方法 — 链地址法

- 开散列方法首先对关键码集合用某一个散列函数计算它们的存放位置。
- 若设散列表地址空间的所有位置是从**0**到***m*-1**，则关键码集合中的所有关键码被划分为***m***个子集，具有相同地址的关键码归于同一子集。我们称同一子集中的关键码互为同义词。每一个子集称为一个桶。
- 通常各个桶中的表项通过一个单链表链接起来，称之为同义词子表。所有桶号相同的表项都链接在同一个同义词子表中，各链表的表头结点组成一个向量。

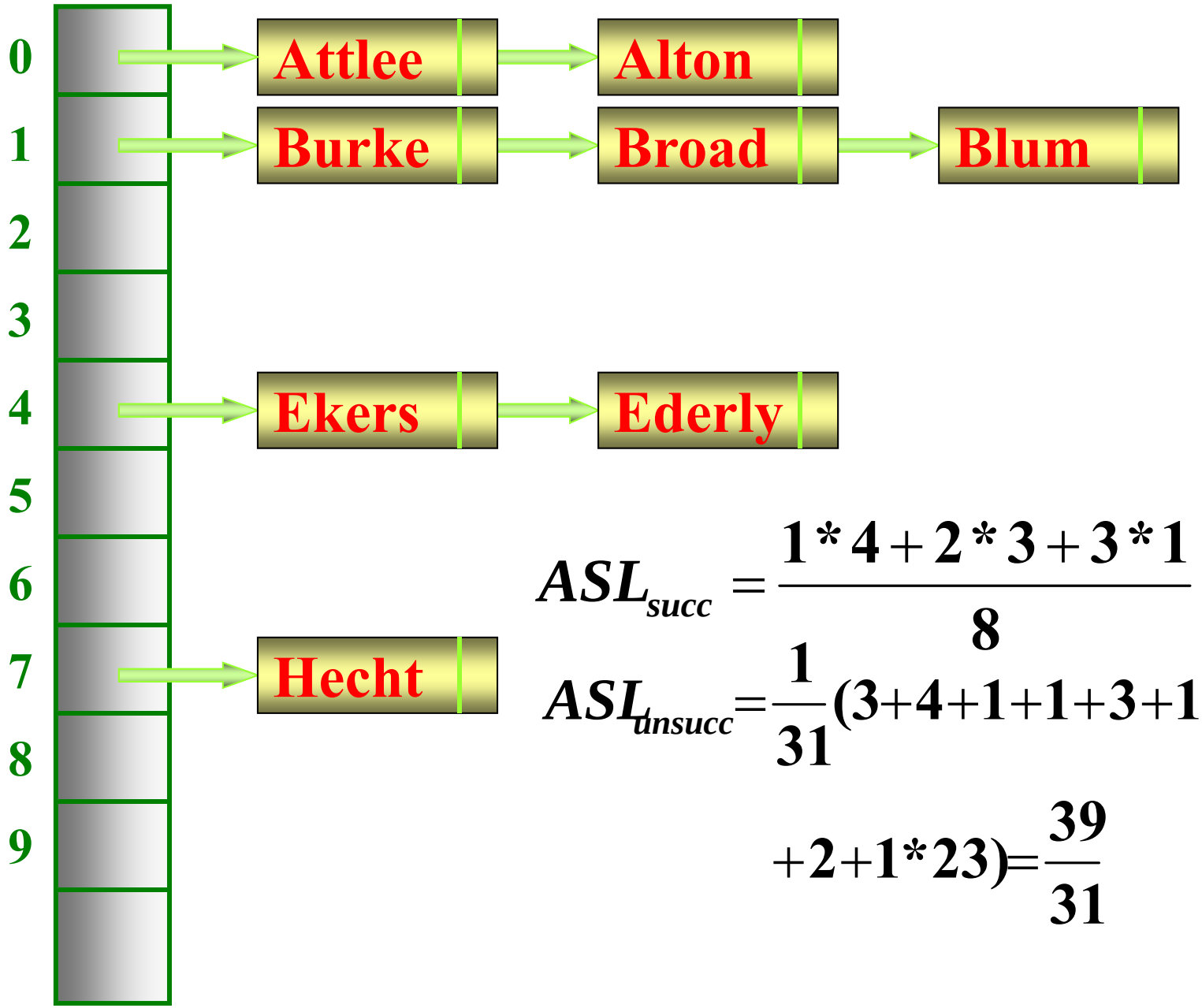
- 向量的元素个数与桶数一致。桶号为*i*的同义词子表的表头结点是向量中的第*i*个元素。
- 示例：给出一组表项关键码{Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly}。散列函数为：*Hash*(*x*)=*ord* (*x*)-*ord* (‘*A*’)。
- 用它计算可得：

  Hash(Burke)=1　　*Hash*(Ekers)=4

  Hash(Broad)=1　　*Hash*(Blum)=1

  Hash(Attlee)=0　　*Hash*(Hecht)=7

  Hash(Alton)=0　　*Hash*(Ederly)=4
- 散列表为 *HT*[0..30], *m*=31。



$$ASL_{succ} = \frac{1*4+2*3+3*1}{8} = \frac{13}{8}$$

$$ASL_{unsucc} = \frac{1}{31}(3+4+1+1+3+1+1+$$

$$+2+1*23) = \frac{39}{31}$$

- 通常，每个桶中的同义词子表都很短，设有*n*个关键码通过某一个散列函数，存放到散列表中的*m*个桶中。那么每一个桶中的同义词子表的平均长度为*n*/*m*。以搜索平均长度为*n*/*m*的同义词子表代替了搜索长度为*n*的顺序表，搜索速度快得多。

利用链地址法处理溢出时的类定义

```
template <class Type> class ListNode {  //链结点
friend HashTable;
  private:
    Type key;  //关键码
    ListNode *link;  //链指针  };
```

```
typedef ListNode <Type> *ListPtr;

class HashTable {  //散列表的类定义
  public:
    HashTable(int size=defaultsize)  //构造函数
      { buckets=size;  ht=new ListPtr[buckets]; }
  private:
    int buckets;  //桶数
    ListPtr <Type> *ht;  //散列表数组的头指针
};
```

循链搜索的算法

```
template<class Type> Type * HashTable<Type>::
Find(const Type &x) {
    int j=HashFunc(x, buckets);  //初始桶号
    ListPtr <Type> *p=ht[j];  //桶地址
    while (p!=NULL)
        if (p->key==x)  return &p->key;
        else p=p->link;
    return 0;
}
```

- 其它如插入、删除操作可参照单链表的插入、删除等算法来实现。
- 应用开散列法处理冲突，需要增设链接指针，似乎增加了存储开销。事实上，由于闭散列法必须保持大量的空闲空间以确保搜索效率，如二次探查法要求装填因子$\alpha \leq 0.5$，而表项所占空间又比指针大得多，所以使用开散列法反而比闭散列法节省存储空间。

散列表分析

- 散列表是一种直接计算记录存放地址的方法，它在关键码与存储位置之间直接建立了映象。
- 当选择的散列函数能够得到均匀的地址分布时，在搜索过程中可以不做多次探查。
- 由于很难避免冲突，增加了搜索时间。冲突的出现，与散列函数的选取（地址分布是否均匀），处理冲突的方法（是否产生堆积）有关。
- 在实际应用中使用关键码进行散列时，如在用作关键码的许多标识符具有相同的前缀或后缀，或者是相同字符的不同排列的场合，不同的散列函数往往导致散列表具有不同的搜索性能。

下图给出一些实验结果

$\alpha = n / m$	0.50		0.75		0.90		0.95	
散列函数种类	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法
平方取中	1.26	1.73	1.40	9.75	1.45	310.14	1.47	310.53
除留余数	1.19	4.52	1.31	10.20	1.38	22.42	1.41	25.79
移位折叠	1.33	21.75	1.48	65.10	1.40	710.01	1.51	118.57
分界折叠	1.39	22.97	1.57	48.70	1.55	69.63	1.51	910.56
数字分析	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
理论值	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

搜索关键码时所需对桶的平均访问次数

从图中可以看出，开散列法优于闭散列法；在散列函数中，用除留余数法作散列函数优于其它类型的散列函数，最差的是折叠法。

- 当装填因子α较高时，选择的散列函数不同，散列表的搜索性能差别很大。在一般情况下多选用除留余数法，其中的除数在实用上应选择不含有**20**以下的质因数的质数。
- 对散列表技术进行的实验评估表明，它具有很好的平均性能，优于一些传统的技术，如平衡树。但散列表在最坏情况下性能很不好。如果对一个有***n***个关键码的散列表执行一次搜索或插入操作，最坏情况下需要 **O(*n*)** 的时间。
- **Knuth**对不同的冲突处理方法进行了概率分析。

- 若设α是散列表的装填因子：

$$\alpha = \frac{\text{表中已装有记录数}}{\text{表中预设的最大记录数}} = \frac{n}{m}$$

- 用地址分布均匀的散列函数*Hash*( )计算桶号。
- S_n是搜索一个随机选择的关键码 x_i $(1 \le i \le n)$ 所需的关键码比较次数的期望值
- U_n是在长度为m的散列表中n个桶已装入表项的情况下，装入第$n+1$项所需执行的关键码比较次数期望值。
- 前者称为在$\alpha=n/m$时的搜索成功的平均搜索长度，后者称为在$\alpha=n/m$时的搜索不成功的平均搜索长度。

- 用不同的方法溢出处理冲突时散列表的平均搜索长度如图所示。

处理冲突的方法		平均搜索长度 *ASL*	
		搜索成功 *Sn*	搜索不成功 *Un*（登入新记录）
闭散列法	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	伪随机探查法 二次探查法 双散列法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
开散列法（同义词子表法）		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

- 散列表的装填因子α表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。
- 散列表的搜索性能，即平均搜索长度依赖于散列表的装填因子，不直接依赖于***n***或 ***m***。
- 不论表的长度有多大，我们总能选择一个合适的装填因子，以把平均搜索长度限制在一定范围内。

例 求散列表大小并设计散列函数

- 设有一个含200个表项的散列表，用二次探查法解决冲突，按关键码查询时找到一个新表项插入位置的平均探查次数不超过1.5，则散列表项应能够至少容纳多少个表项。再设计散列函数（设搜索不成功的平均搜索长度为$U_n=1/(1-\alpha)$，其中α为装填因子）
- 解答：设表中表项个数$n=200$，搜索不成功的平均搜索长度

$$U_n=1/(1-\alpha)\le 1.5 \Rightarrow \alpha\le 1/3$$

$$\therefore \quad n/m=200/m=\alpha\le 1/3,\ m\ge 600$$

随堂练习

例1：使用散列函数hash f(x)=x%11，把一个整数值转换成散列表下标，现要把数据1, 13, 12, 34, 38, 33, 27, 22插入到散列表中。

(1) 使用线性探查再散列法来构造散列表。

(2) 使用链地址法来构造散列表。

(3) 针对这种情况，确定其查找成功所需的平均查找次数以及查找不成功所需的平均探查次数。

(1) 首先计算出每个数据的 hash 地址如下：

f(1)=1, f(13)=2, f(12)=1, f(34)=1, f(38)=5, f(33)=0, f(27)=5, f(22)=0

使用线性探查再散列法构造的散列表：

地址	0	1	2	3	4	5	6	7	8	9	10
数据	33	1	13	12	34	38	27	22			
探查成功次数	1	1	1	3	4	1	2	8			
探查不成功次数	9	8	7	6	5	4	3	2	1	1	1

(2) hash 表长度为 m，填入表中的数据个数为 n，则装填因子 $\alpha=n/m$；因此有：$\alpha=8/11$。

在等概率情况下，

使用线性探查再散列法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=(1+1/(1-\alpha))/2=(1+1(1-8/11))/2=7/3$

使用线性探查再散列法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=(1+1/(1-\alpha)^2)/2=(1+1(1-8/11)^2)/2=65/9$

使用链地址法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=1+\alpha/2=1+8/22=15/11$

使用链地址法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=\alpha+e^{-\alpha}=8/11+e^{-8/11}$

***注意，利用装填因子所计算的平均查找长度为近似值。

也可直接计算如下：

线性探查再散列法——

$ASL_{成功}=1/8*(1+1+1+3+4+1+2+8)=21/8$

$ASL_{不成功}=1/11(9+8+7+6+5+4+3+2+1+1+1)=47/11$

链地址法查找成功时为链表中找到该结点时的比较次数，查找不成功时必须比较到链尾的空指针处，因此有：

$ASL_{成功}=1/8*(1+2+1+3+1+1+2)=13/8$

$ASL_{不成功}=1/11(3+4+2+1+1+3+1+1+1+1+1)=19/11$

本章小结

- 知识点
  - 顺序搜索与折半搜索
  - 二叉搜索树
    - 定义、插入、删除与性能分析
  - **AVL**树
    - 四种旋转方式
  - 哈希函数定义及冲突解决

- 课程习题
  - 笔做题——**7.7, 7.8, 7.13, 7.14, 7.22, 7.23**
    （以作业形式提交）
  - 上机题——**7.3, 7.5, 7.10**
  - 思考题——**7.2, 7.4, 7.15, 7.21**

Any Suggestion or Question

联系方式：

- 张玥杰

  Email: yjzhang@fudan.edu.cn