第七章 搜索结构

- 静态搜索结构
- 二叉搜索树
- AVL树
- 伸展树（略）
- 红黑树（略）
- 本章小结

7.1 静态搜索结构

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。

顺序搜索 (Sequential Search)

- 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

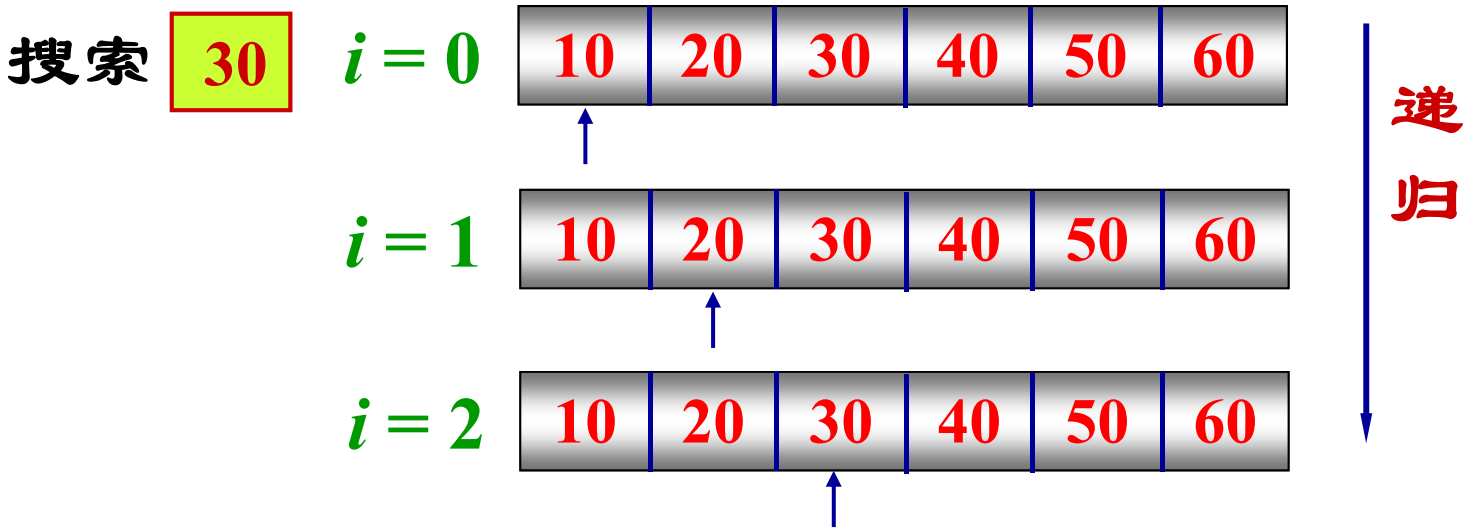
$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot (i+1)$$

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

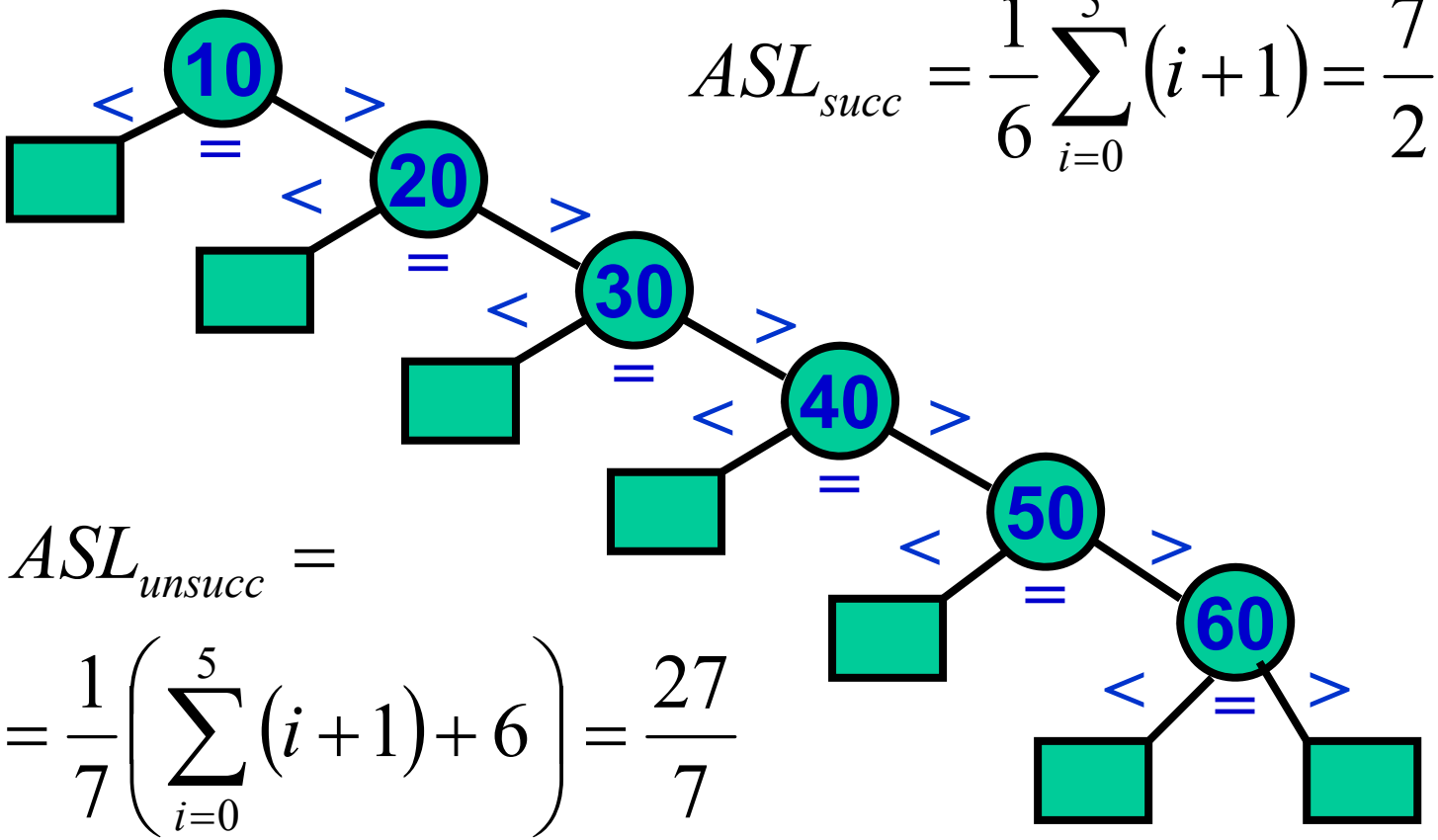
```
template <class Type> int SearchList <Type> ::
SeqSearch ( const Type &x,  int loc ) const {
//在数据表 Element[0..n-1] 中搜索其关键码
//与给定值匹配的对象，函数返回其表中位置
//参数 loc 是在表中开始搜索位置
   if ( loc >= CurrentSize ) return -1;  //搜索失败
   else if ( Element[loc].key = = x ) return loc;
   //搜索成功
   else return Search (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

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)

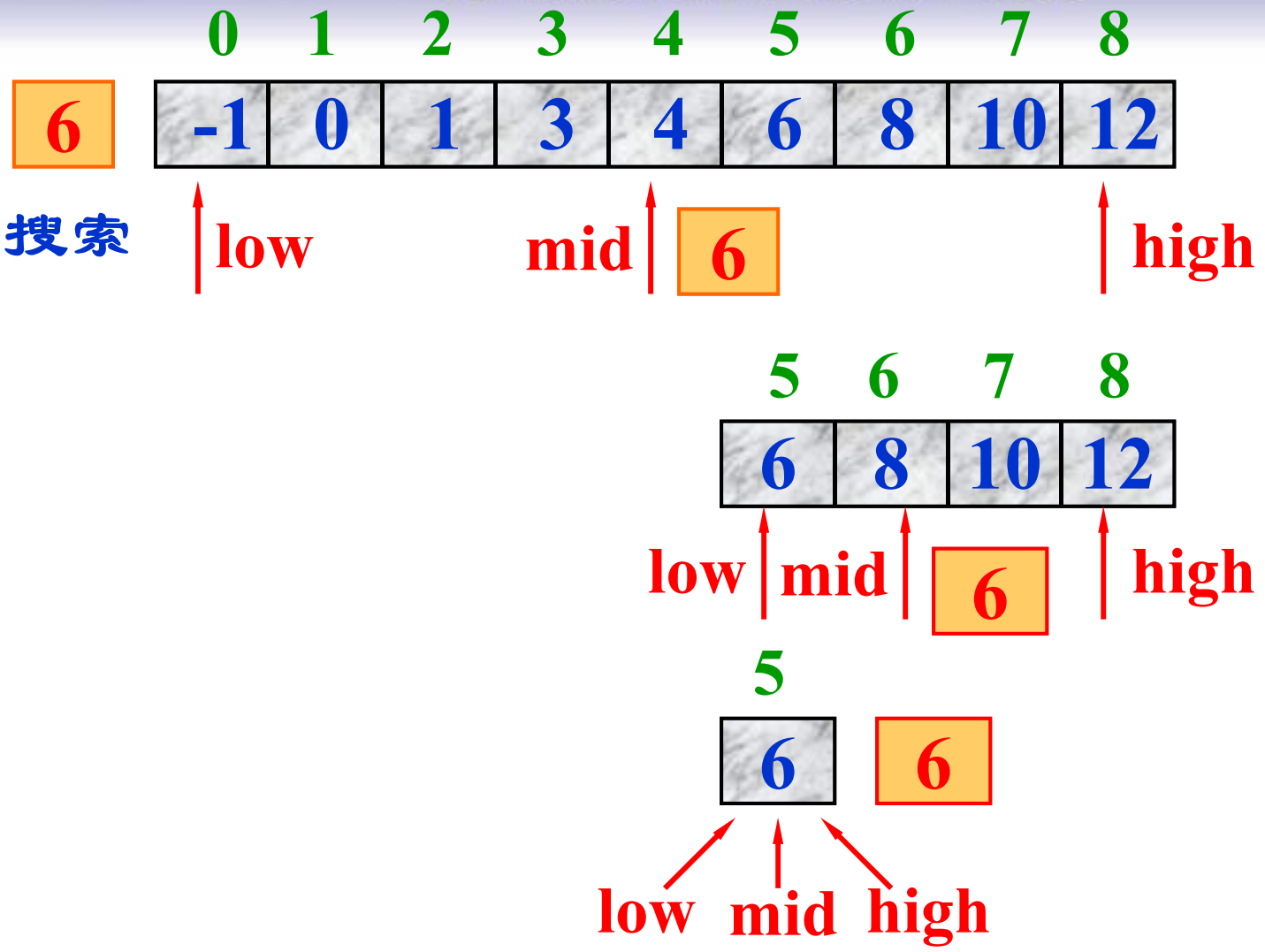


$$ASL_{succ} = \frac{1}{6}\sum_{i=0}^{5}(i+1) = \frac{7}{2}$$

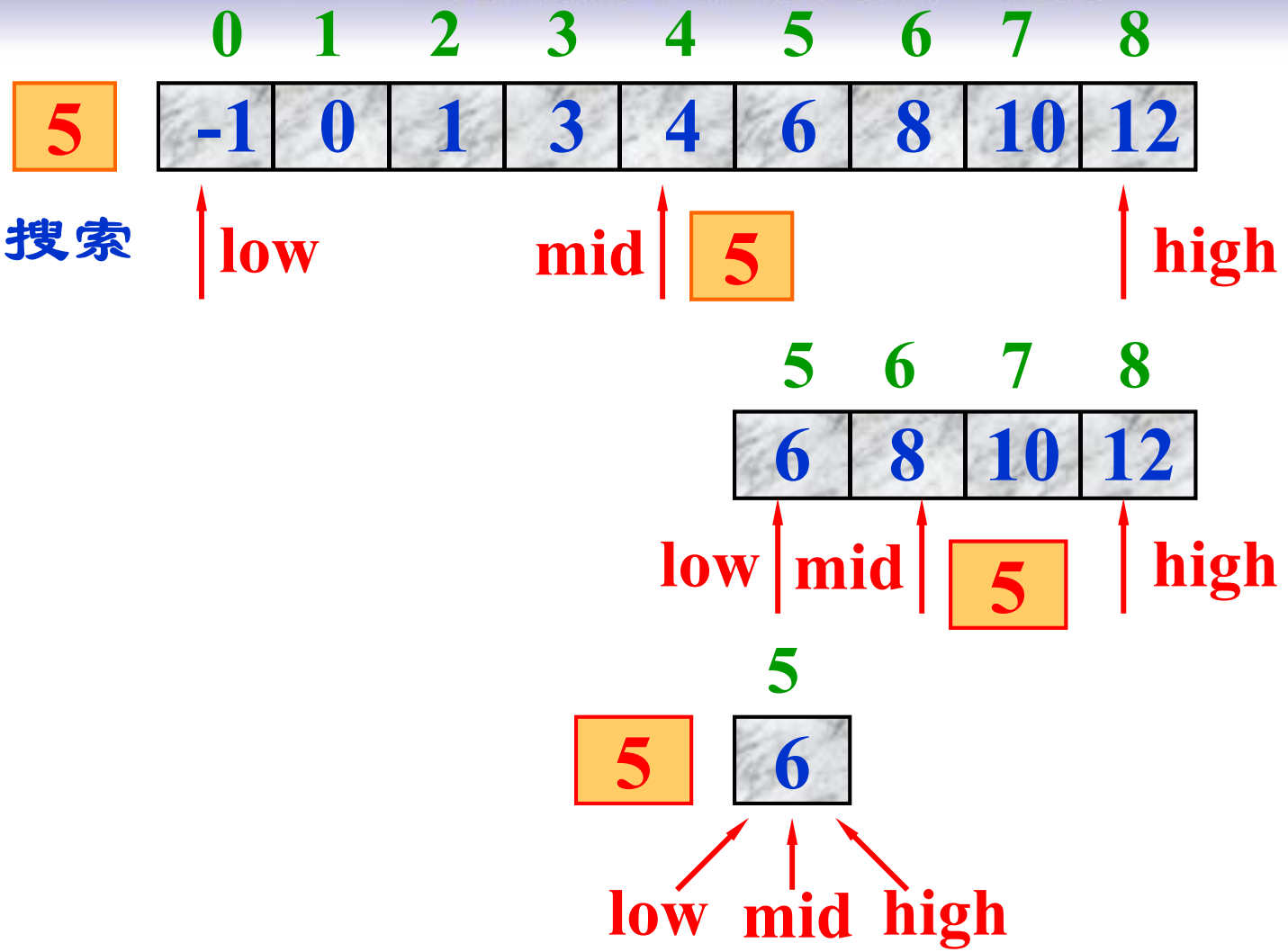
$$ASL_{unsucc} = \frac{1}{7}\left(\sum_{i=0}^{5}(i+1)+6\right) = \frac{27}{7}$$

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;
}
```

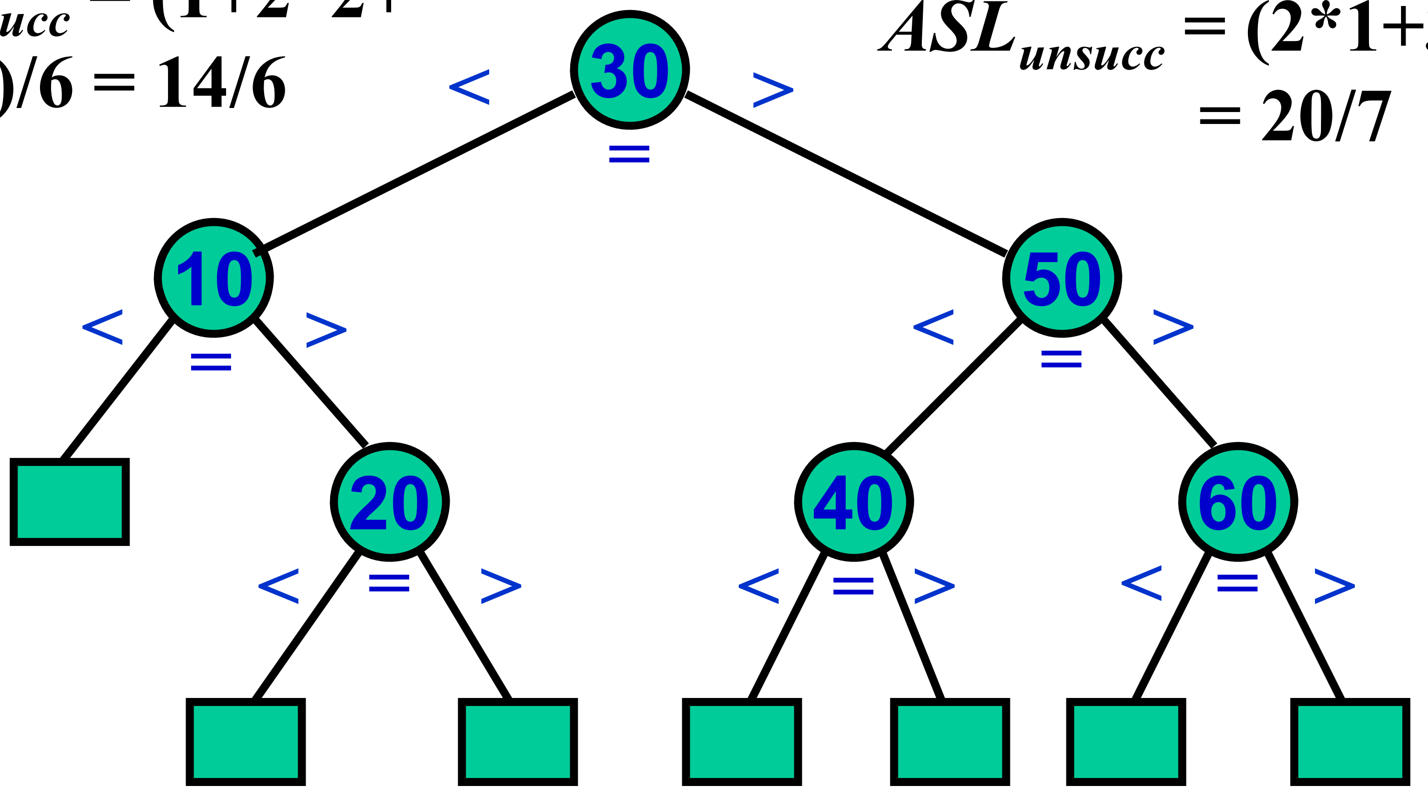
```
template <class Type> int SortedList <Type> ::
BinarySearch ( const Type &x ) const {
//折半搜索的迭代算法
   int high = CurrentSize-1,  low = 0,  mid;
   while ( low <= high ) {
      mid = ( low + high ) / 2;
      if ( Element[mid].key < x )
         low = mid + 1;  //右缩搜索区间
      else if ( Element[mid].key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

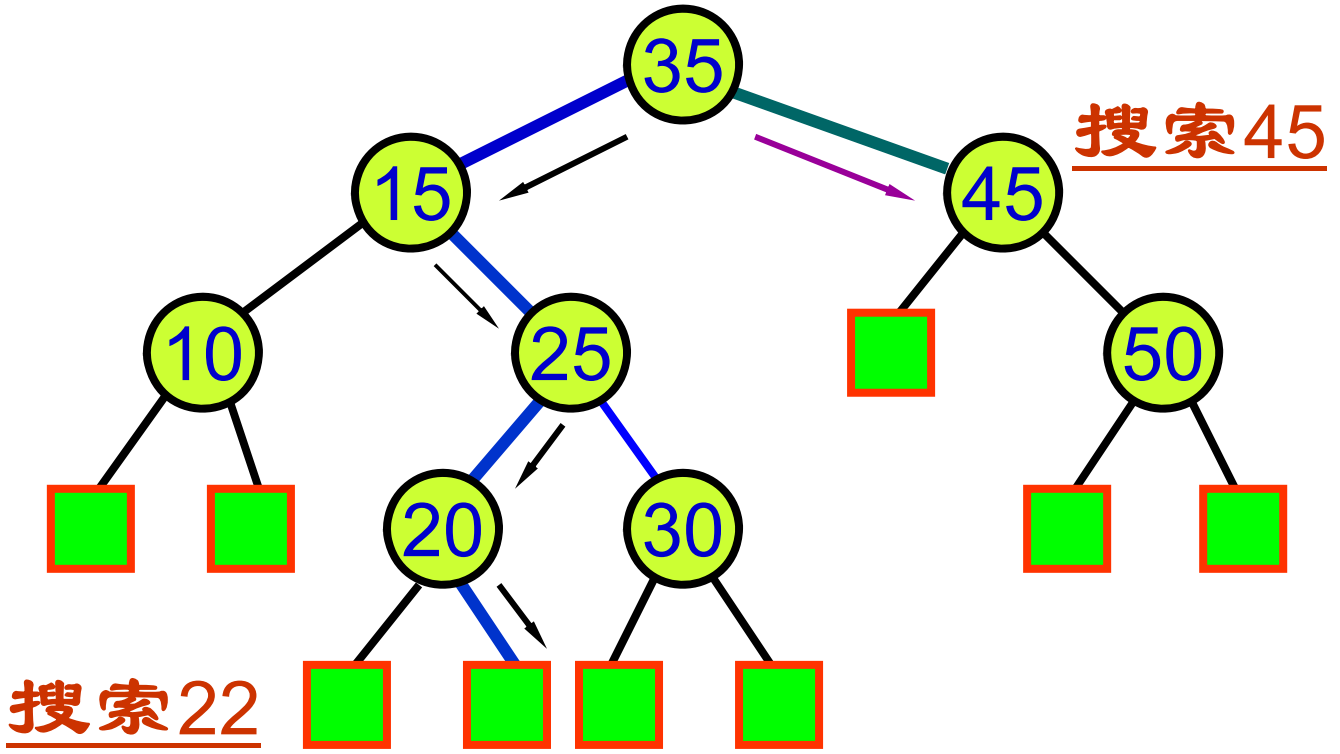
(10, 20, 30, 40, 50, 60)

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(n)**的搜索空间，其比较的中点是**tmp=F(n-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 二叉搜索树 (Binary Search Tree)

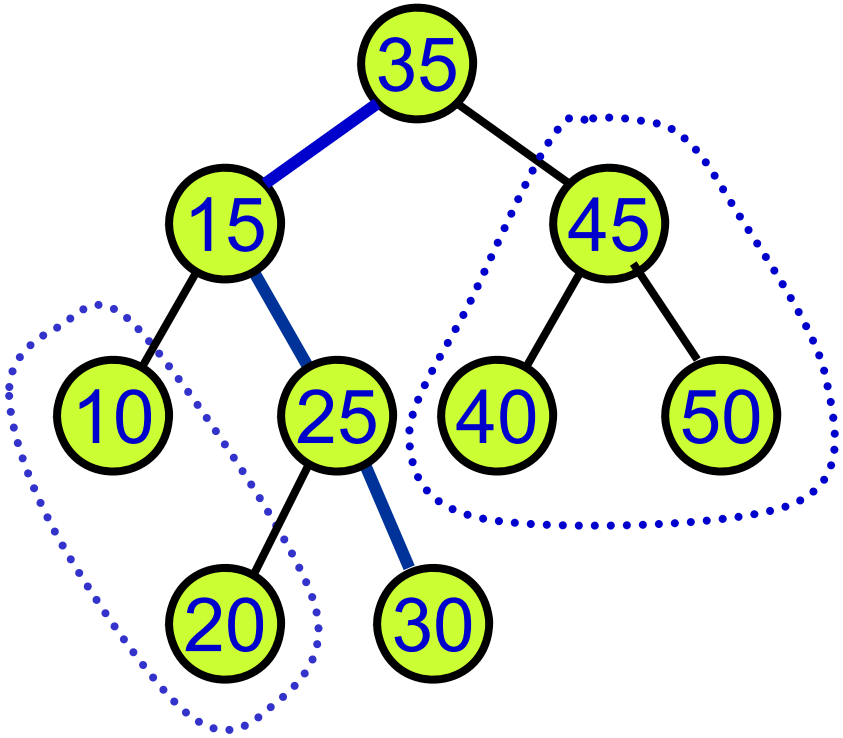
定义

二叉搜索树或者是一棵空树，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搜索树。

二叉搜索树例

- 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。



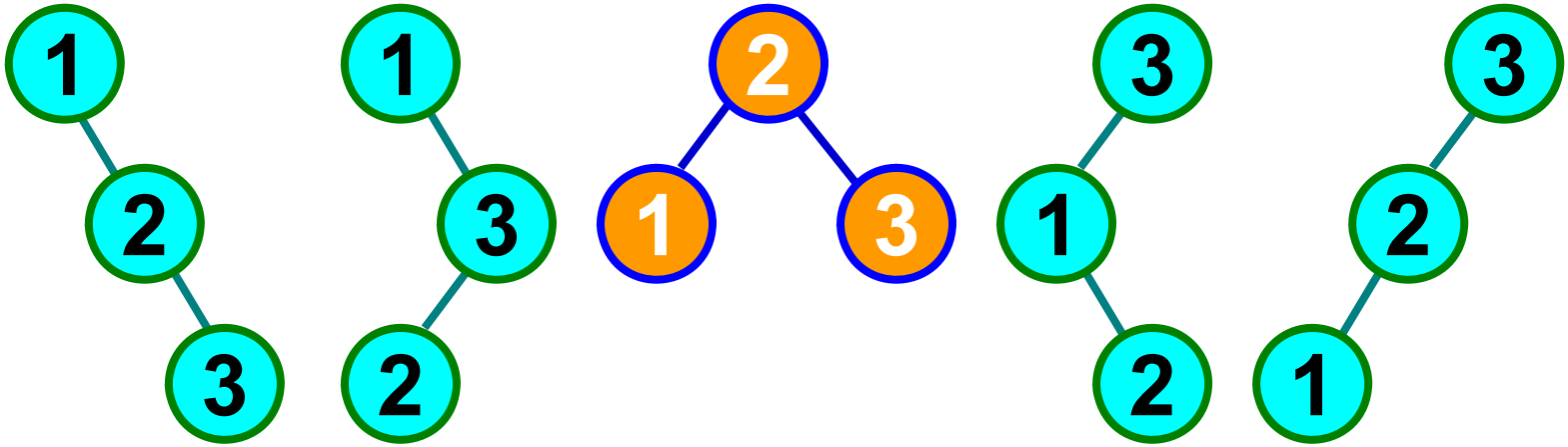
- 注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键码不一定小于路径上的结点的关键码。

n 个结点的二叉搜索树的数目

【例】3 个结点的二叉搜索树：

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

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



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

二叉搜索树的类定义

```
#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
};
```

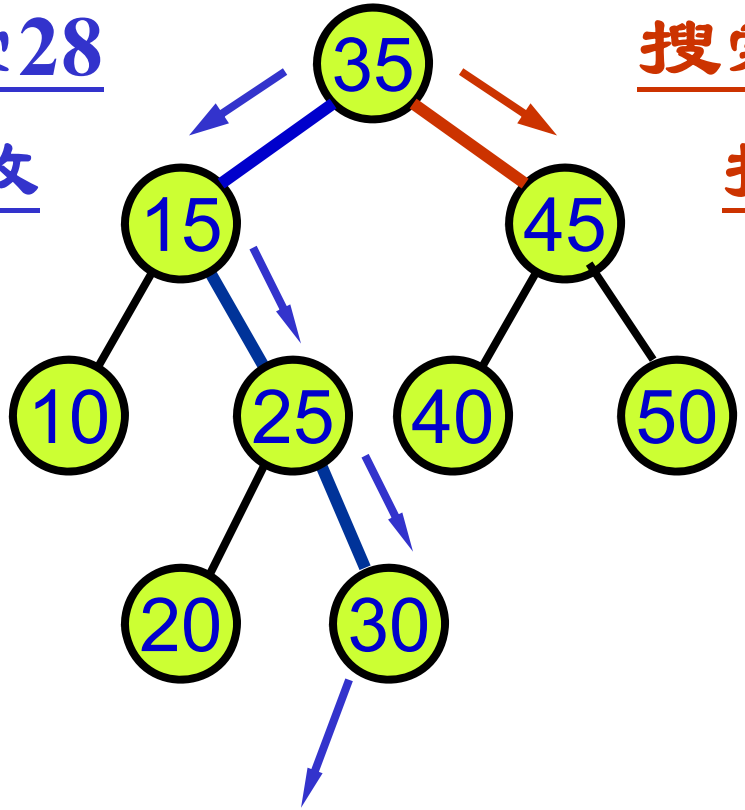
二叉搜索树上的搜索

在二叉搜索树上进行搜索，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。

- 假设想要在二叉搜索树中搜索关键码为 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



搜索45

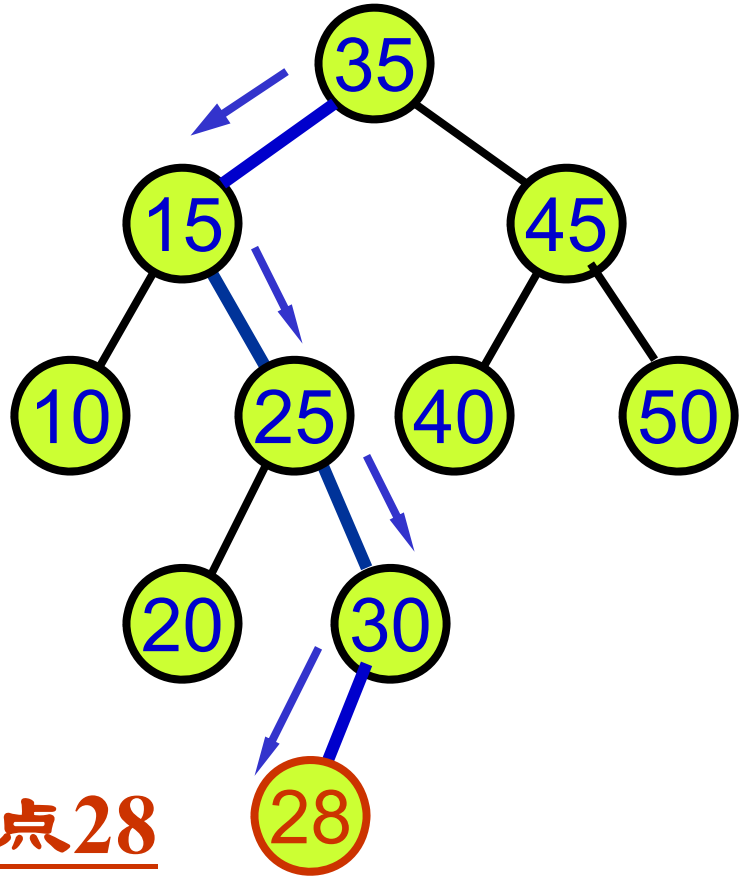
搜索成功

```
template <class Type>
        BstNode <Type> * BST<Type> ::
Search ( Type x, BstNode <Type> *ptr ) const {
//二叉搜索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 {
//二叉搜索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
}
```

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

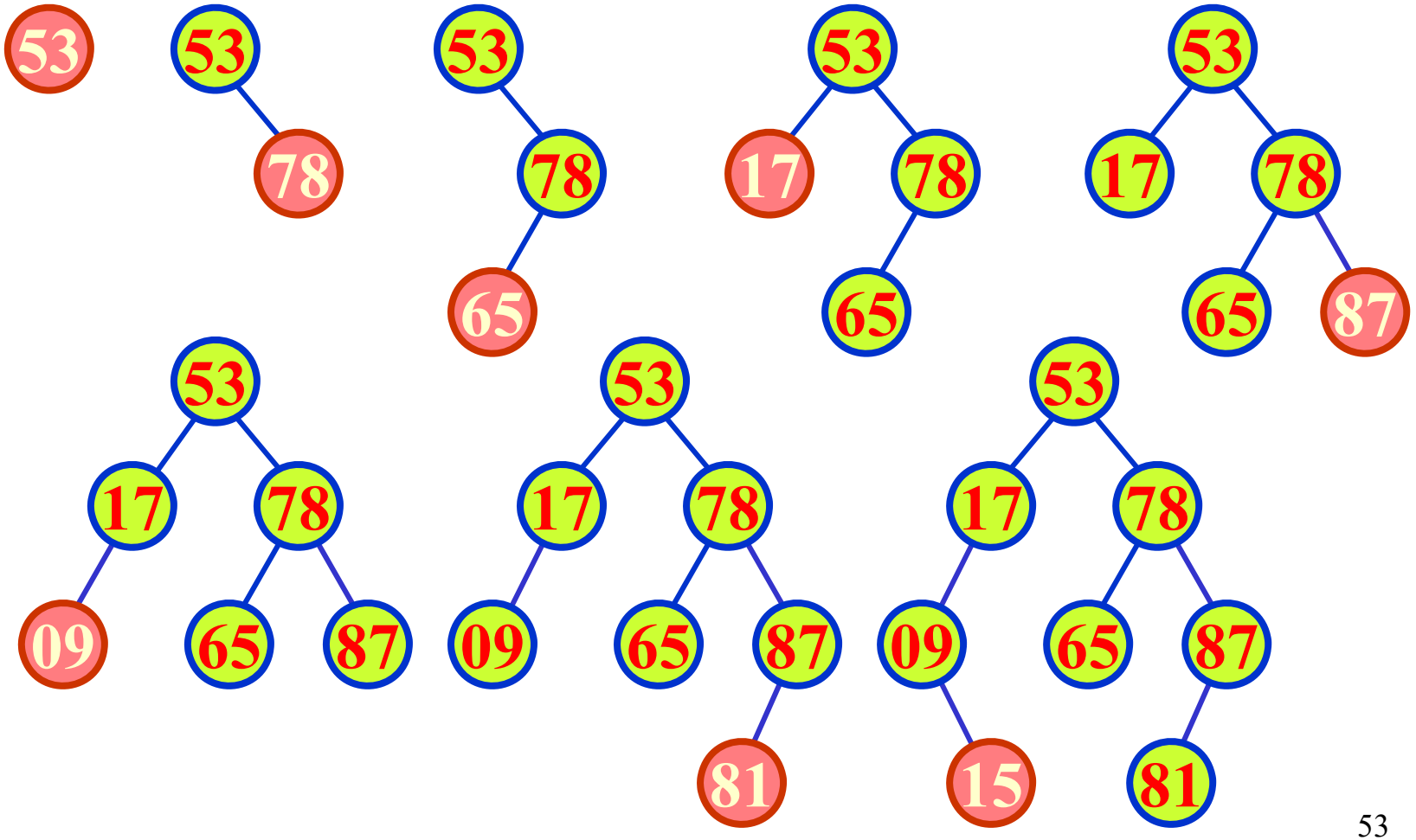
    树中原来没有关键码等于给定值的结点，把新元素加到搜索操作停止的地方。

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 }



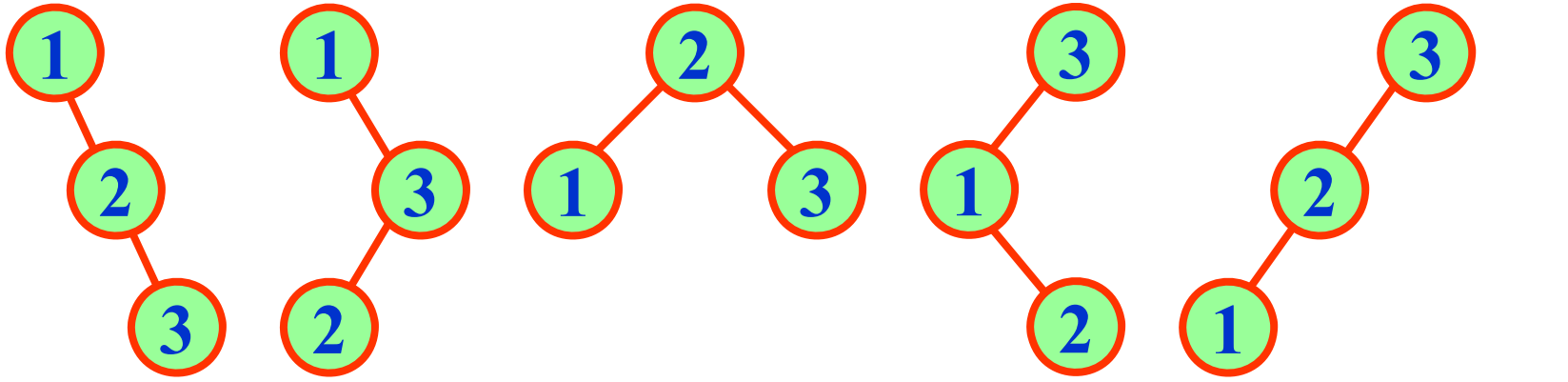
```
template <class Type> BST <Type> ::
BST ( Type value ) {
//输入数据，建立二叉搜索树。
// RefValue 是输入结束标志
//应取不可能在输入序列中出现的值
//例如输入序列的值都是正整数时，取为 0 或负数
    Type x;
    root = NULL;  RefValue = value;
    cin >> x;
    while ( x != RefValue )
        { Insert ( x, root );  cin >> x; }
}
```

同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉搜索树的形态也不同。这直接影响到二叉搜索树的搜索性能。

如果输入序列选得不好，会建立起一棵单支树，使得二叉搜索树的高度达到最大。

{2, 1, 3}

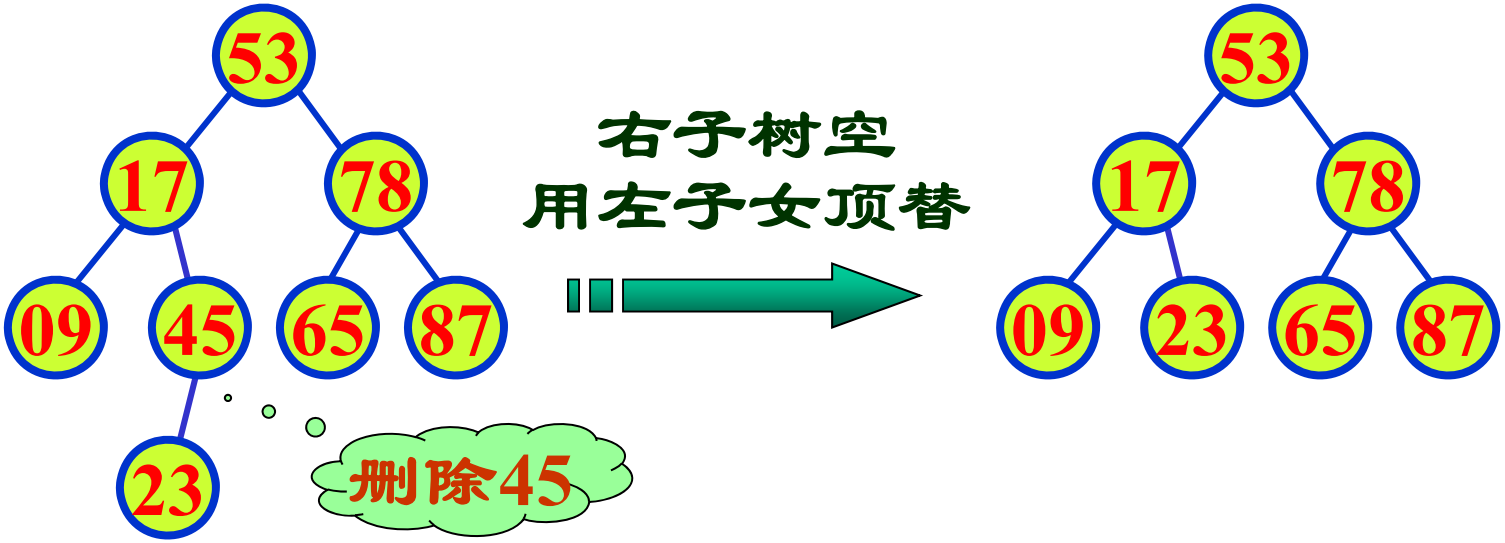
{1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}

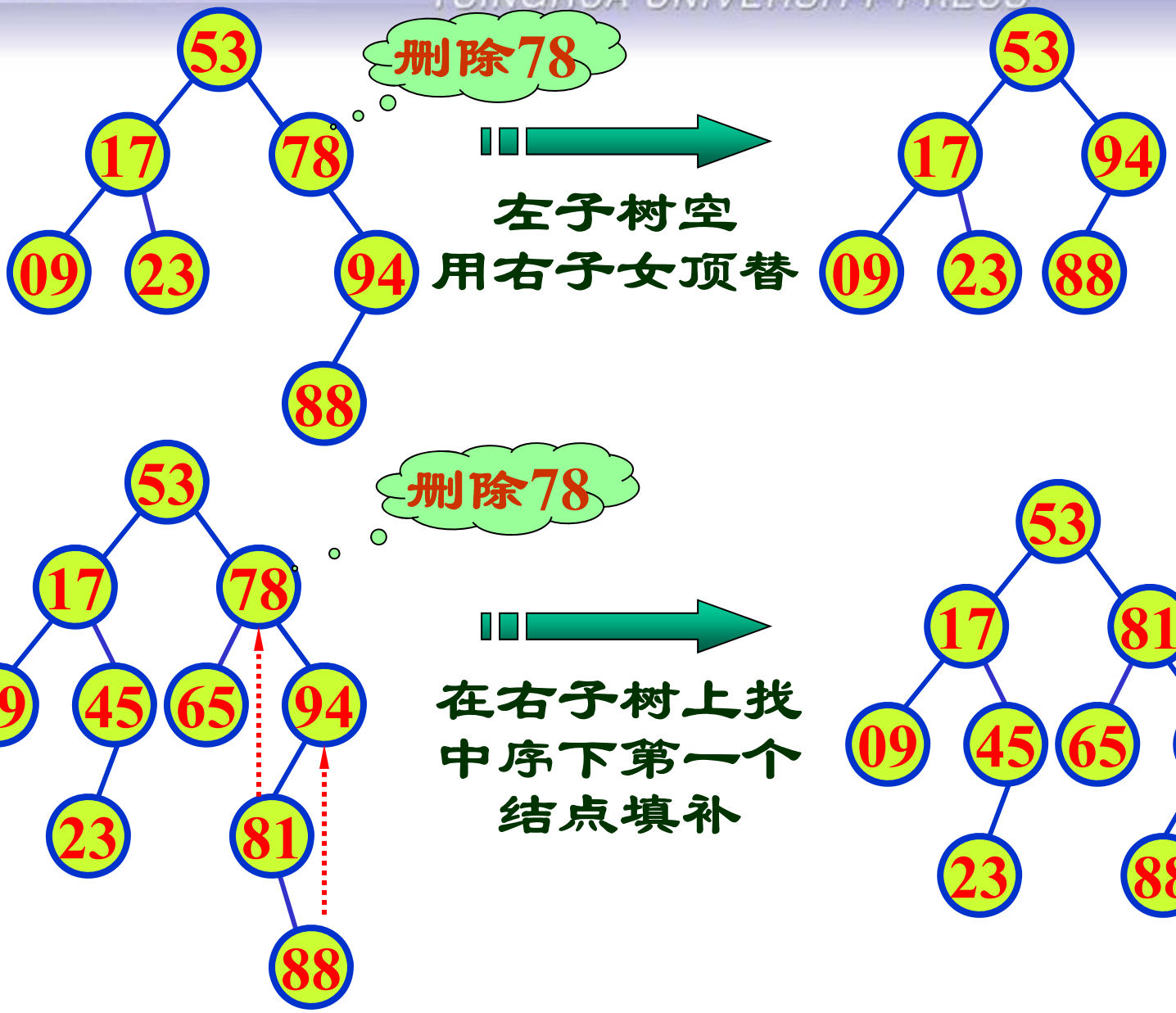


二叉搜索树的删除

- 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;
}
```

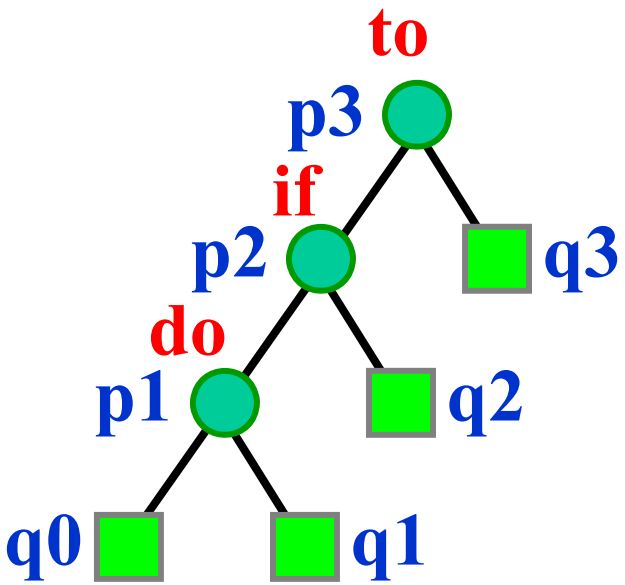
二叉搜索树性能分析

- 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

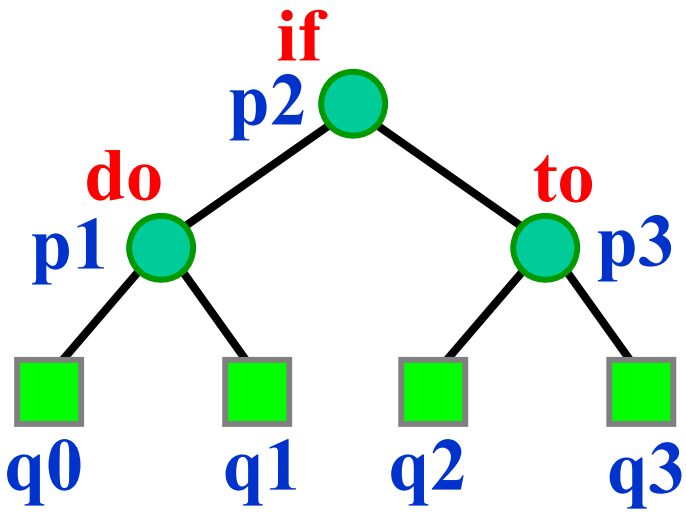
$$\frac{1}{n+1}C_{2n}^{n}$$ （棵）

- 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）。

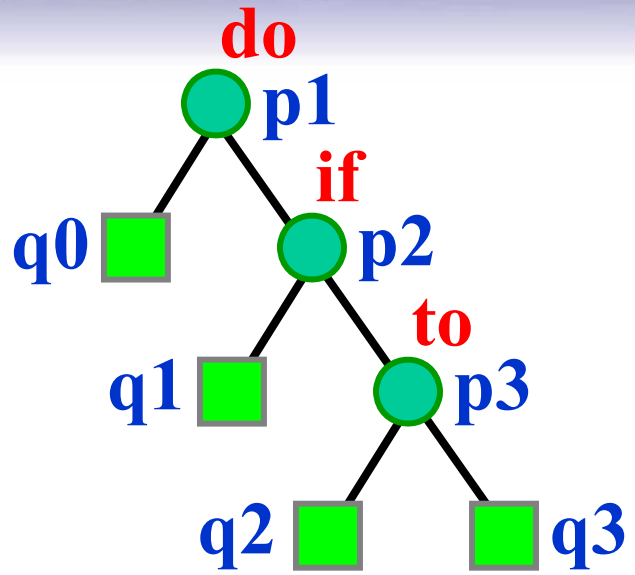
- 例，已知关键码集合 { a1, a2, a3 } = { do, if, to }，对应搜索概率 p1, p2, p3，在各搜索不成功间隔内搜索概率分别为 q0, q1, q2, q3，可能的二叉搜索树如下所示。



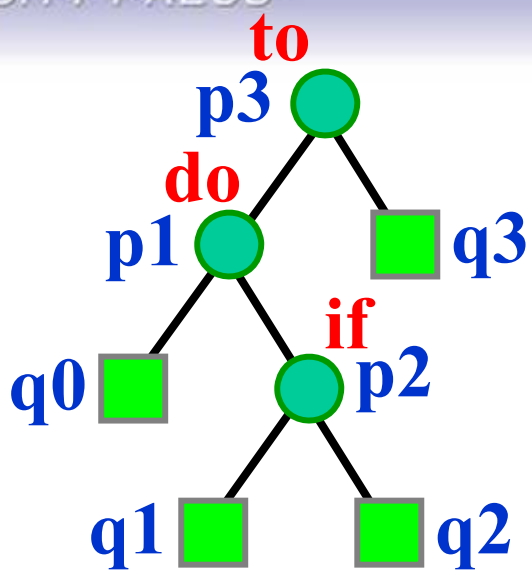
(a)



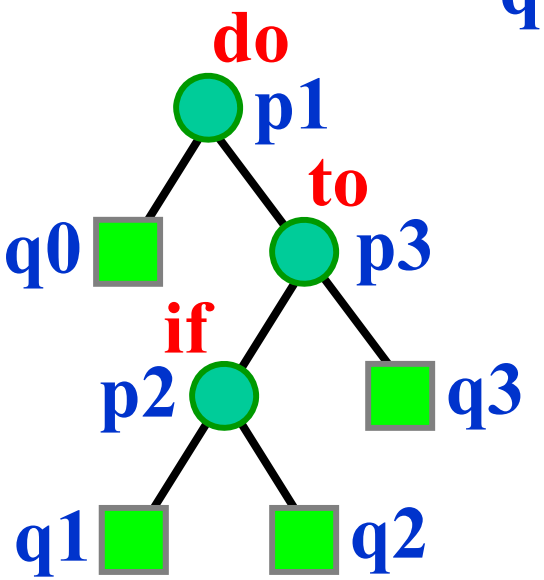
(b)


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，$k = 0, 1, \ldots$，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。

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：

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

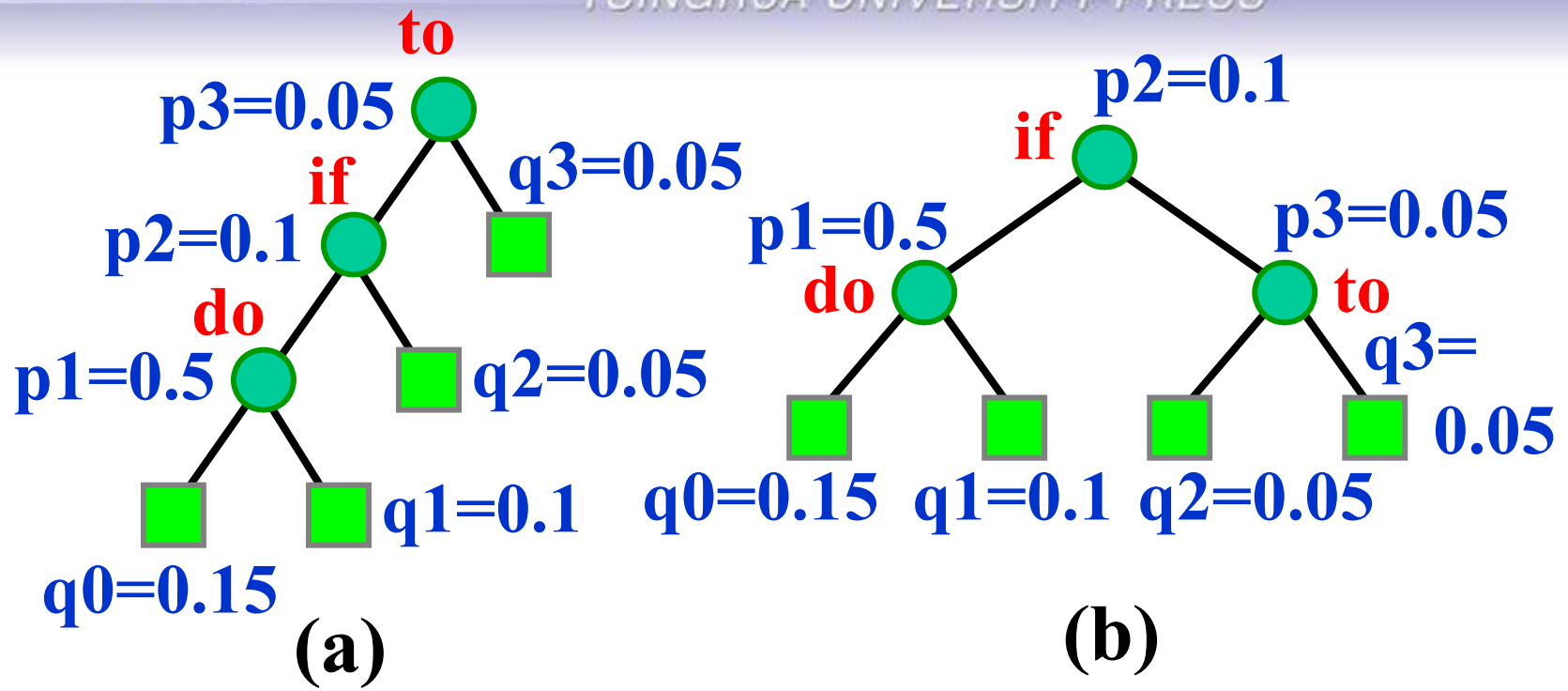
$$ASL_{unsucc} = \sum_{i=n+1}^{2n+1} \lfloor \log_2 i \rfloor.$$

(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。

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

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

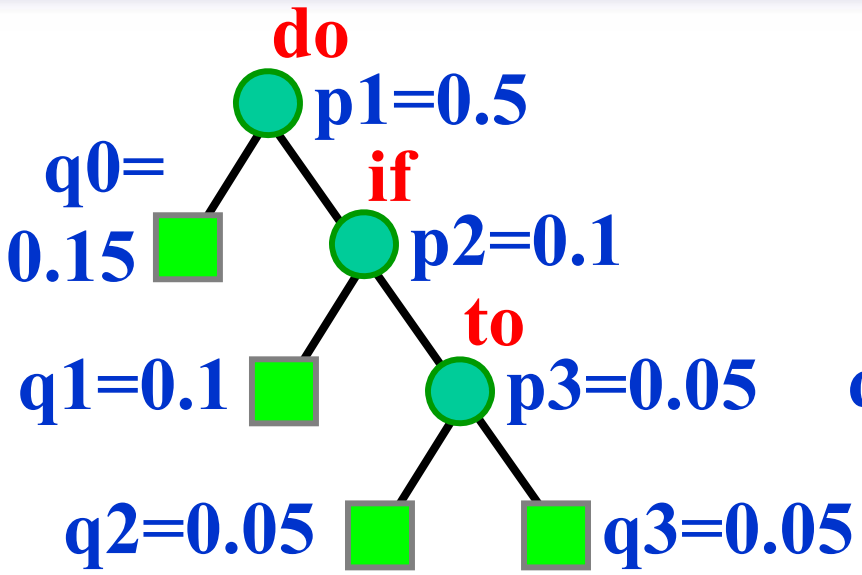


图(a): ASL_{succ} = 0.5*3+0.1*2+0.05*1 = 1.75,

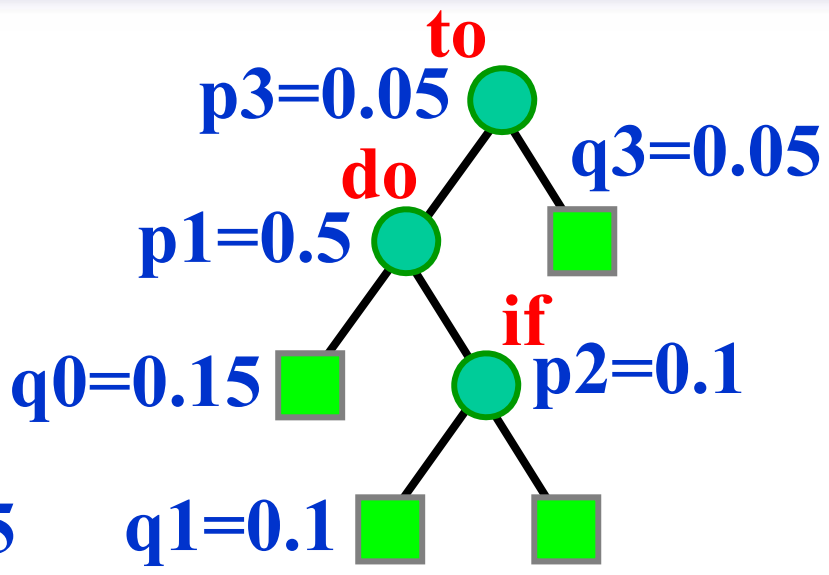
ASL_{unsucc} = 0.15*3+0.1*3+0.05*2+0.05*1 = 0.9。

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。

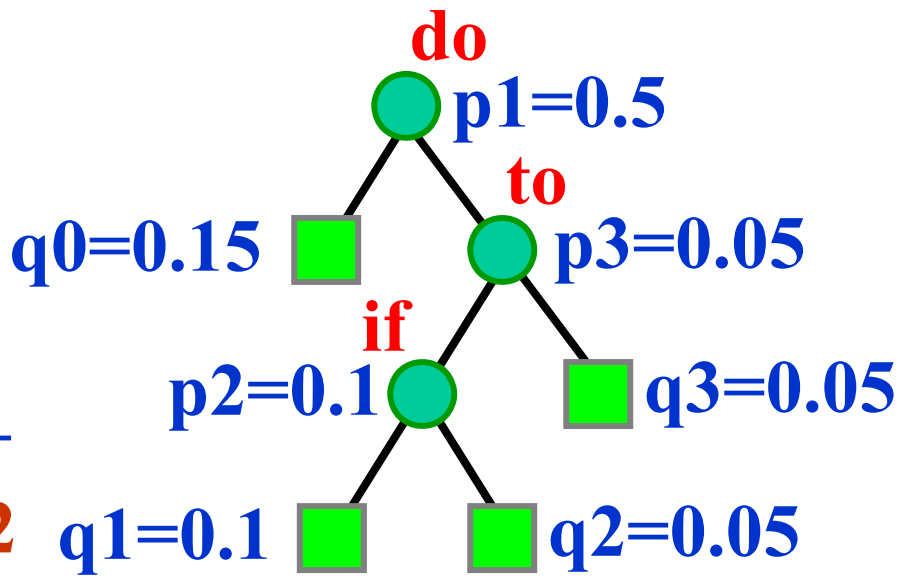
图(d): ASL_{succ} = 0.5*2+0.1*3+0.05*1 = 1.35,

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。

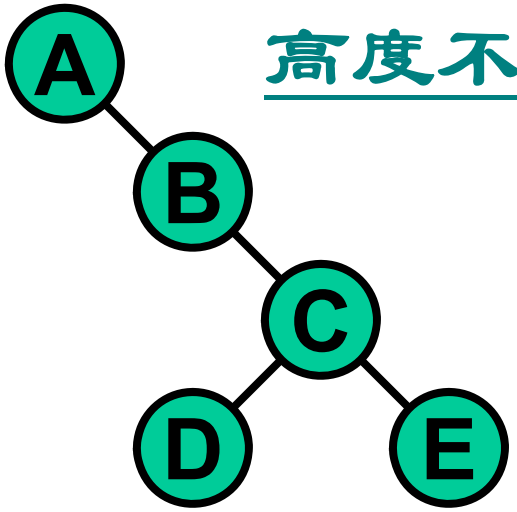
空二叉树T是一棵二叉搜索树；若T非空，首先是它的左、右两棵子树都是二叉搜索树，接着还要求左子树中的所有结点的键值都小于根结点的键值；并且右子树中的所有结点的键值都大于根结点的键值。为了能简单实现子树所有结点与根结点比较，对于左子树可用左子树中最大的键值；对于右子树可以用右子树中最小的键值。若左子树中的最大键值比根结点键值小；并且右子树中的最小键值比根结点键值大，就能得出这是一棵二叉搜索树的结论。

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。

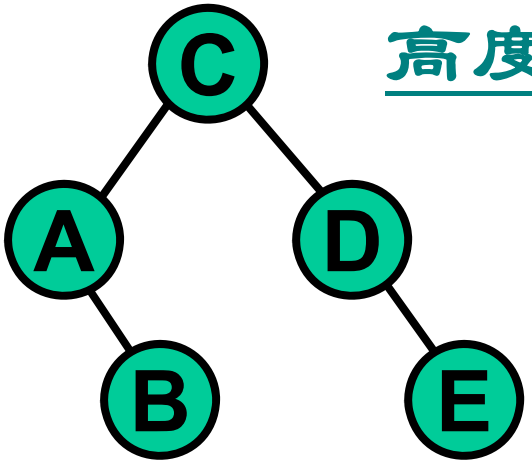
7.3 AVL树 高度平衡的二叉搜索树

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) { }
```

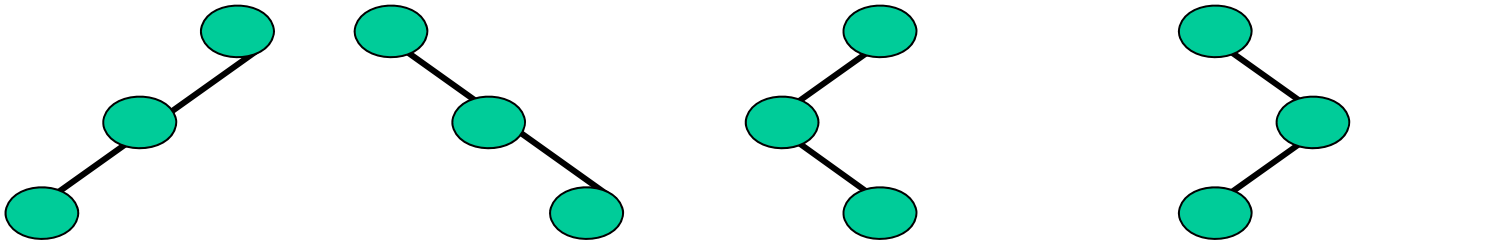
```
protected:
   Type RefValue;
   AVLNode *root;
   bool Insert ( AVLNode <Type> *&Tree, Type x );
   bool Remove ( AVLNode <Type> *&Tree, Type x );
   void RotateLeft ( AVLNode <Type> *Tree,
                     AVLNode <Type> *&NewTree );
   void RotateRight ( AVLNode <Type> *Tree,
                      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。

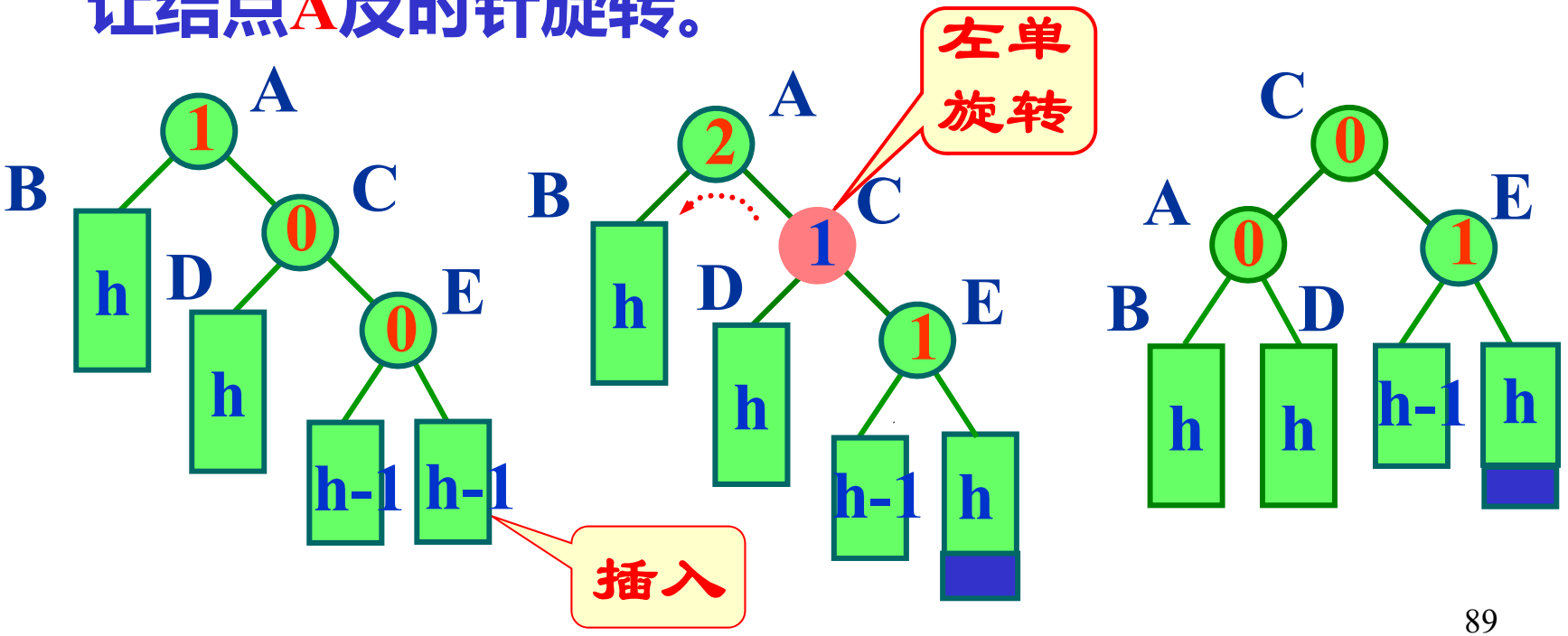
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一 个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。



右单旋转　左单旋转　左右双旋转　右左双旋转

左单旋转 (RotateLeft)

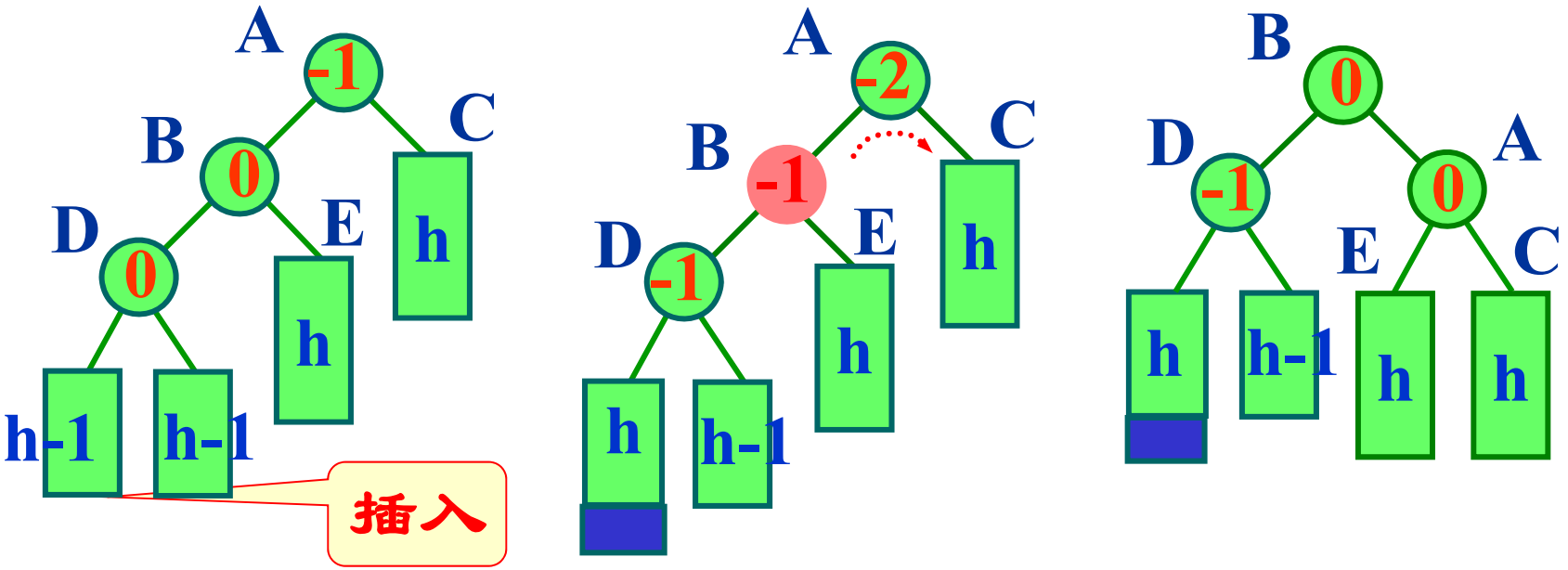
- 在结点A的右子女的右子树E中插入新结点，该子树高度增1导致结点A的平衡因子变成2，出现不平衡。为使树恢复平衡，从A沿插入路径连续取3个结点A、C和E，以结点C为旋转轴，让结点A反时针旋转。



```
template <class 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)

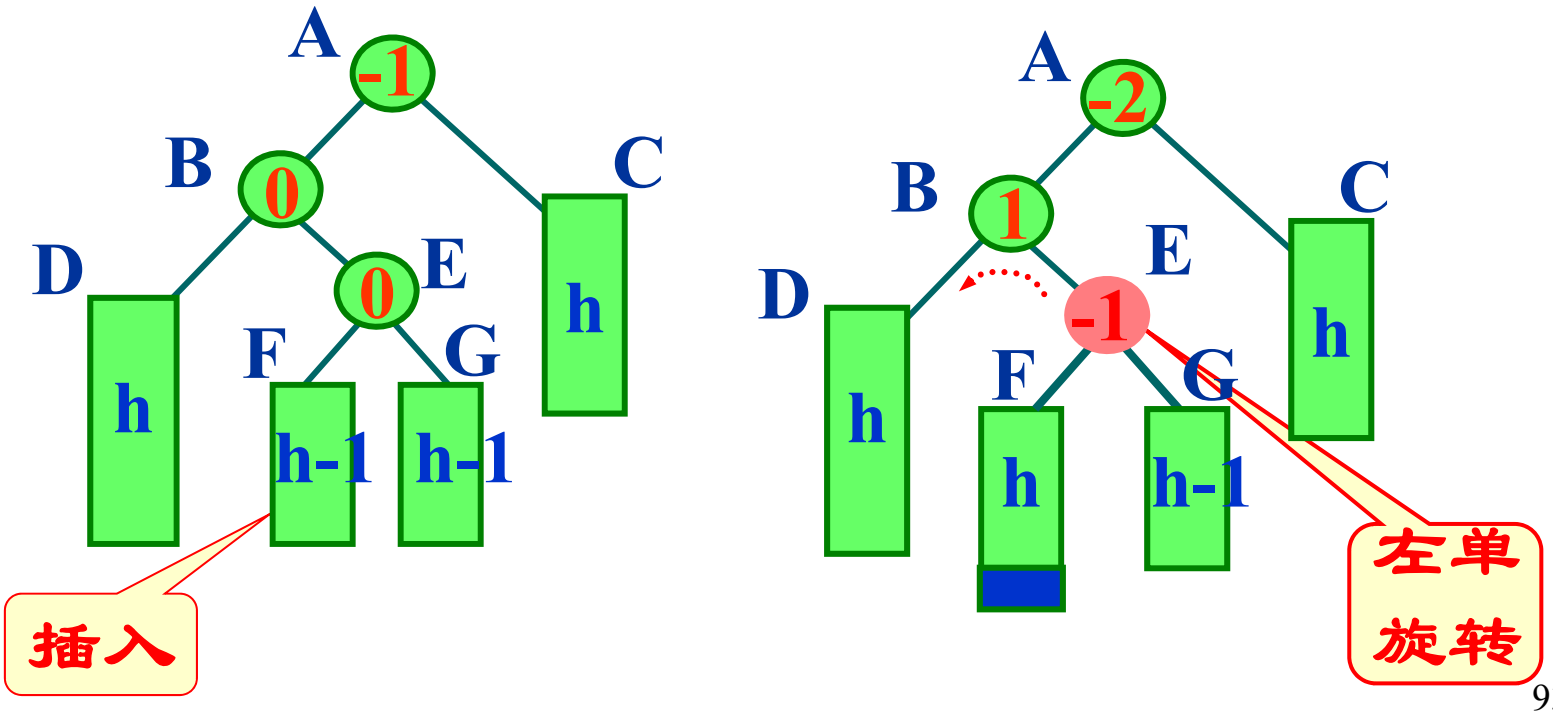
- 在结点A的左子女的左子树D上插入新结点使其高度增1导致结点A的平衡因子增到-2，造成不平衡。为使树恢复平衡，从A沿插入路径连续取3个结点A、B和D，以结点B为旋转轴，将结点A顺时针旋转。



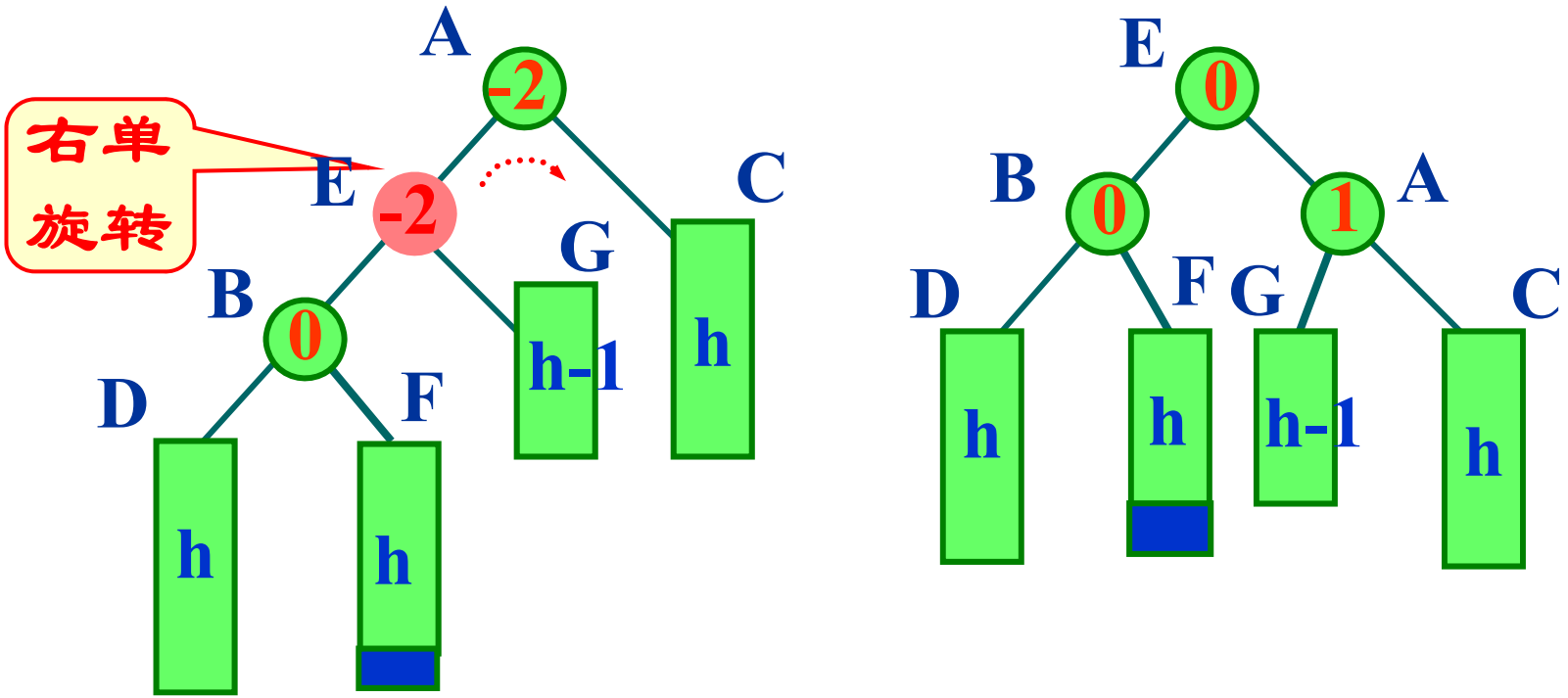
```
template <class Type> void AVLTree <Type> ::
RotateRight ( AVLNode <Type> *Tree,
                    AVLNode <Type> *&NewTree) {
//右单旋转的算法
   NewTree = Tree->left;
   Tree->left = NewTree->right;
   NewTree->right = Tree;
}
```

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。

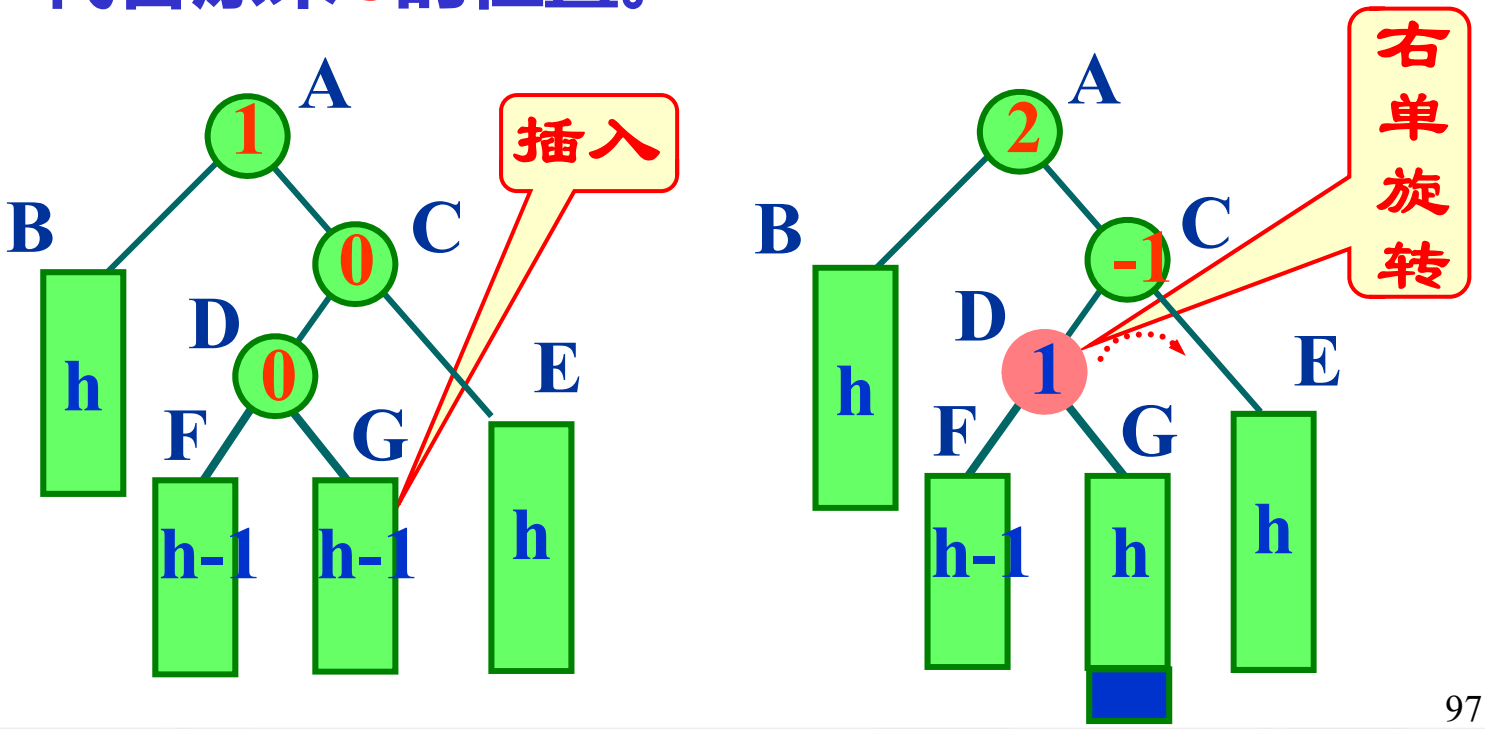


```
template <class Type> void AVLTree <Type> ::
LeftBalance ( AVLNode <Type> *&Tree, int &taller ) {
    AVLNode <Type> *leftsub = Tree->left, *rightsub;
    switch ( leftsub->balance ) {
      case -1 :
        Tree->balance = leftsub->balance = 0;
        RotateRight ( Tree, Tree );
        taller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

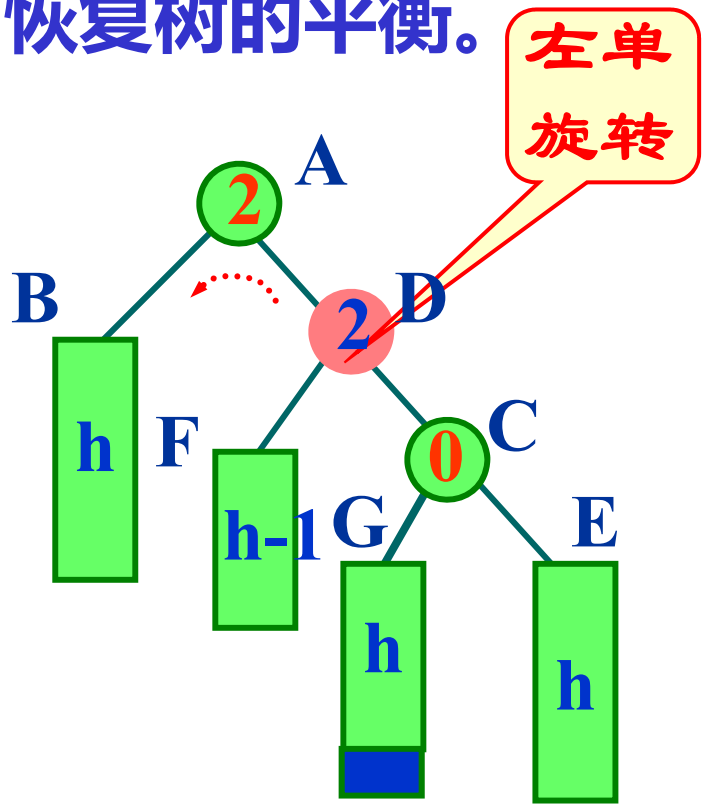
```
            case -1: Tree->balance = 1;
                     leftsub->balance = 0;  break;
            case 0 : Tree->balance = leftsub->balance = 0;
                     break;
            case 1 : Tree->balance = 0;
                      leftsub->balance = -1;
        }
        rightsub->balance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

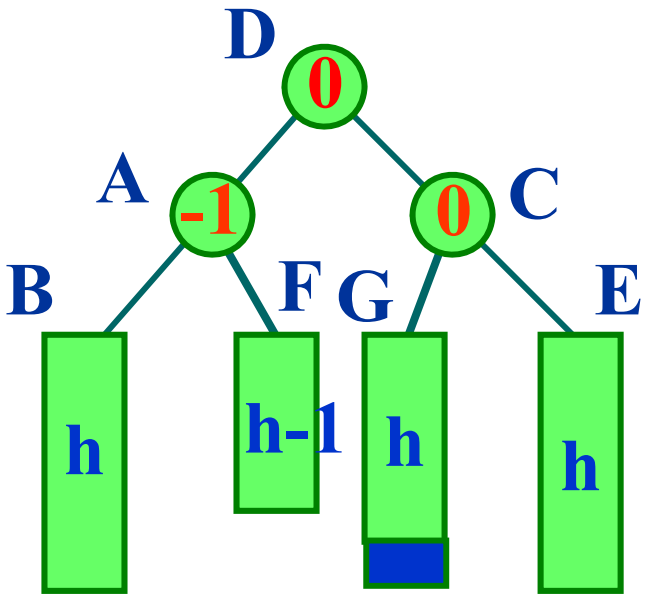
先右后左双旋转 (RotationRightLeft)

- 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。





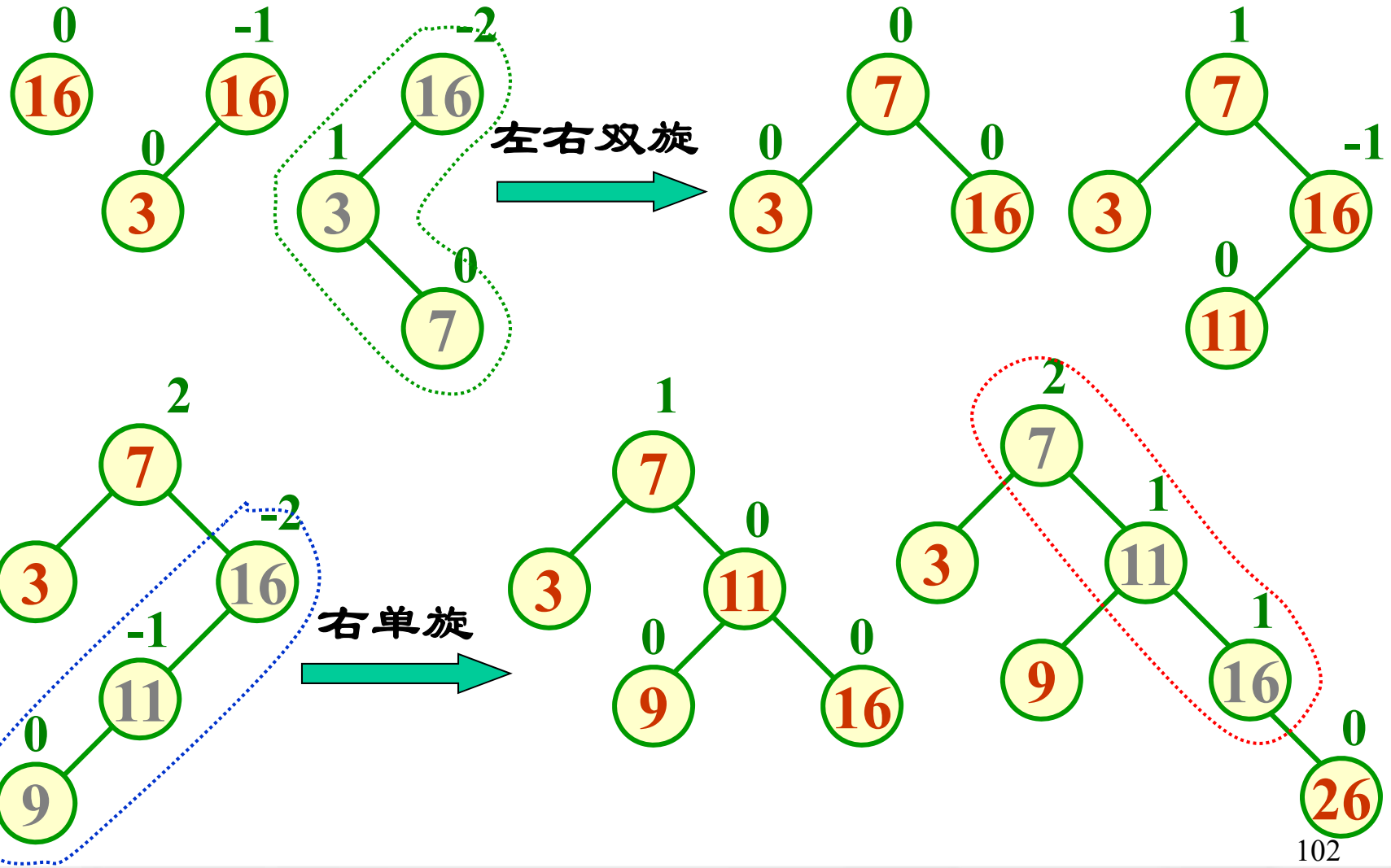
```
template <class Type> void AVLTree <Type>::
RightBalance ( AVLNode <Type> *&Tree, int &taller ) {
    AVLNode <Type> *rightsub = Tree->right, *leftsub;
    switch ( rightsub->balance ) {
      case 1 :
         Tree->balance = rightsub->balance = 0;
         RotateLeft ( Tree, Tree );
         taller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

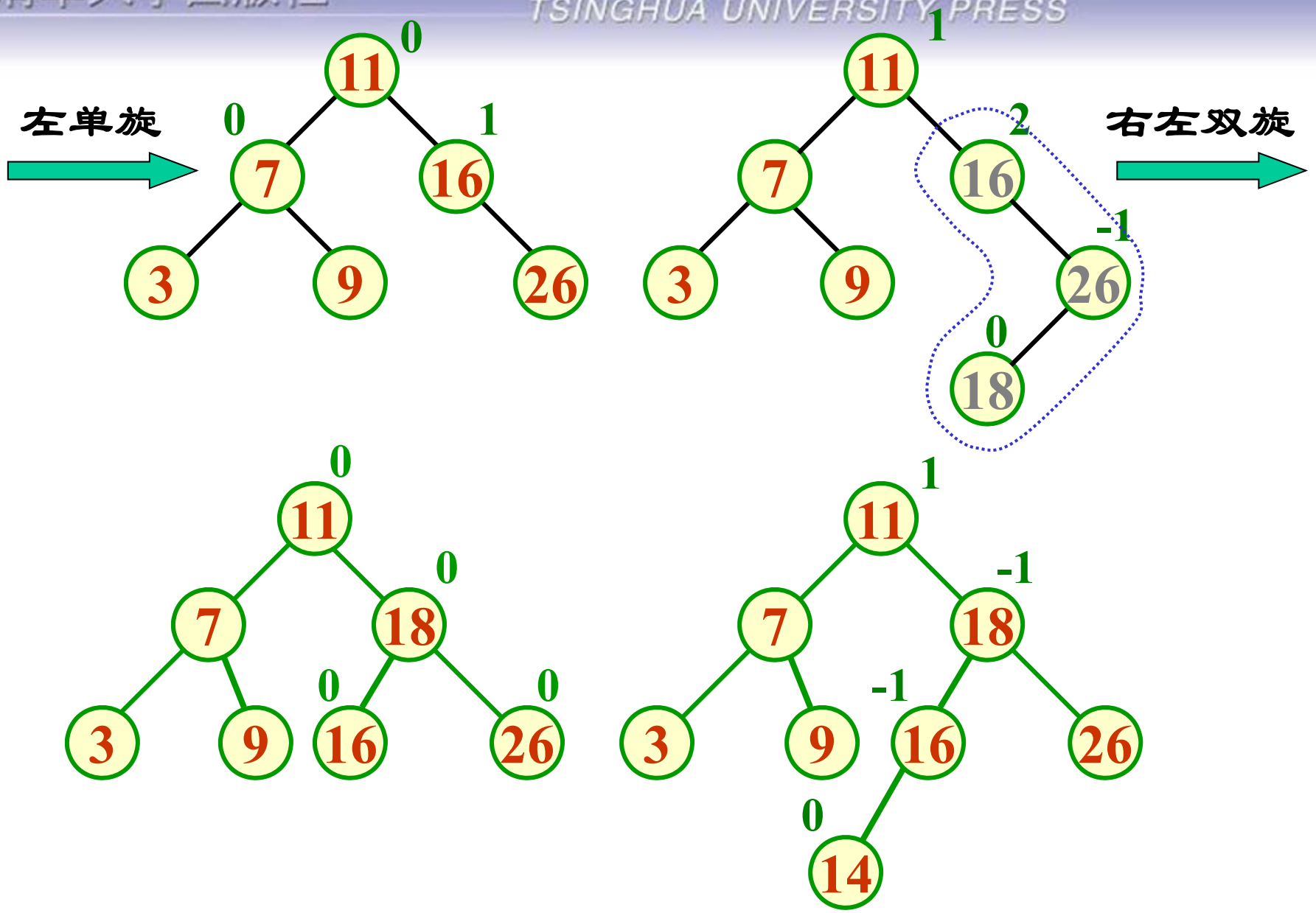
```
            case 1 : Tree->balance = -1;
                     rightsub->balance = 0;  break;
            case 0 : Tree->balance = rightsub->balance = 0;
                     break;
            case -1 : Tree->balance = 0;
                      rightsub->balance = 1;  break;
        }
        leftsub->balance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

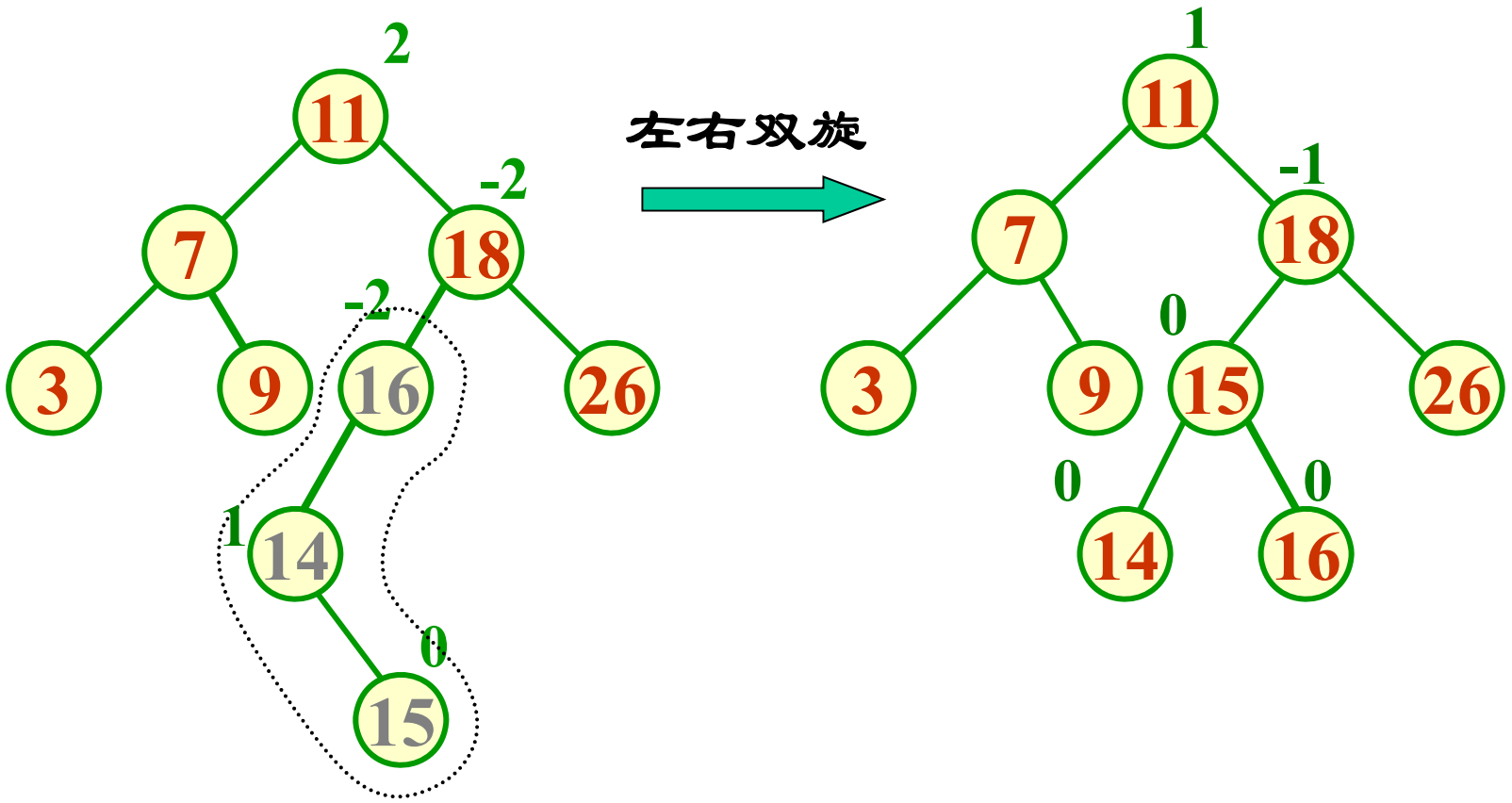
AVL树的插入

- 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



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;
}
```

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：

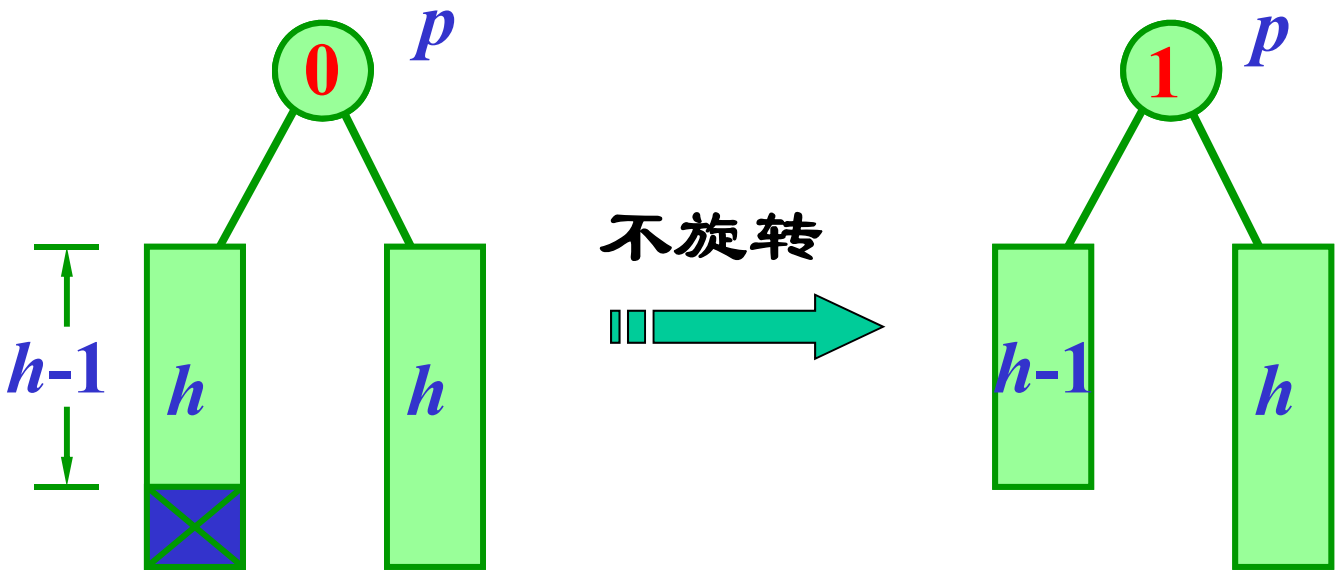
- 将结点*x*从树中删去；
- 因为结点*x*最多有一个子女，可以简单地把*x*的双亲结点中原来指向*x*的指针改指到这个子女结点；
- 如果结点*x*没有子女，*x*双亲结点的相应指针置为NULL；
- 将原来以结点*x*为根的子树的高度减1。

(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。

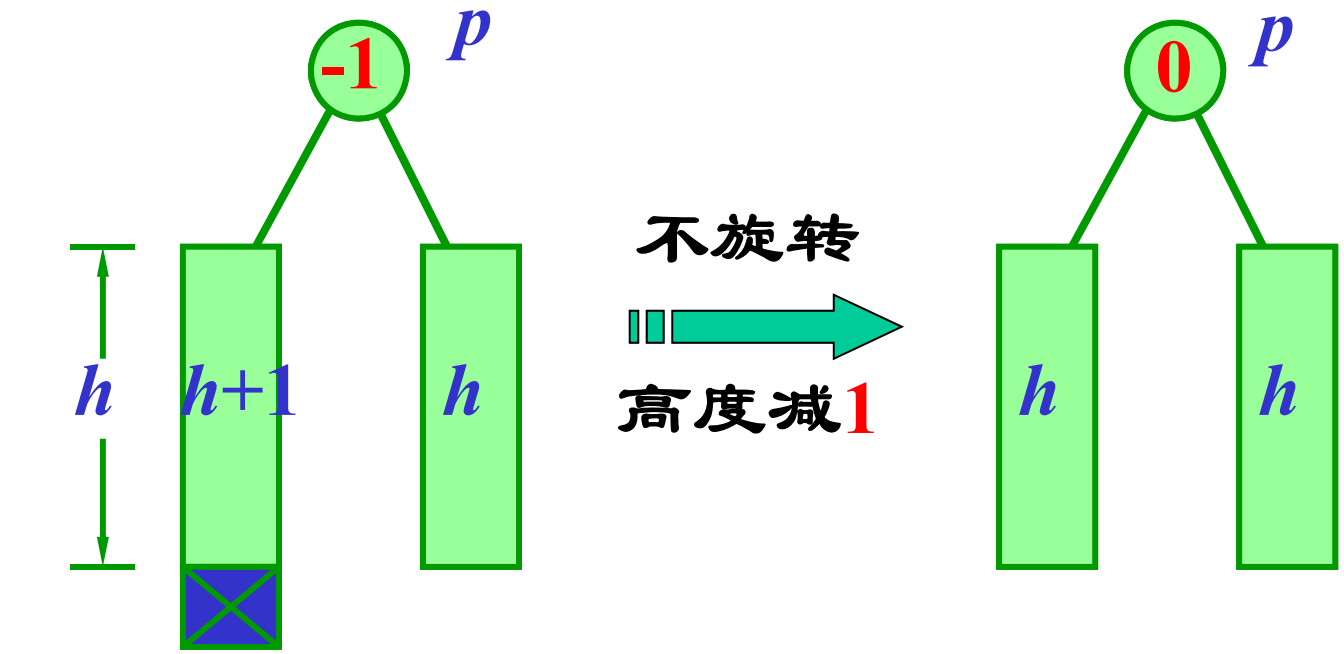
- 必须沿*x*通向根的路径反向追踪高度的变化对路径上各个结点的影响。
- 用一个布尔变量shorter 来指明子树的高度是否被缩短。在每个结点上要做的操作取决于shorter的值和结点的balance，有时还要依赖子女的balance 。
- 布尔变量shorter的值初始化为true，然后对于从*x*的双亲到根的路径上的各个结点*p*。在shorter保持为true时执行下面操作，如果shorter变成false，算法终止。

- 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

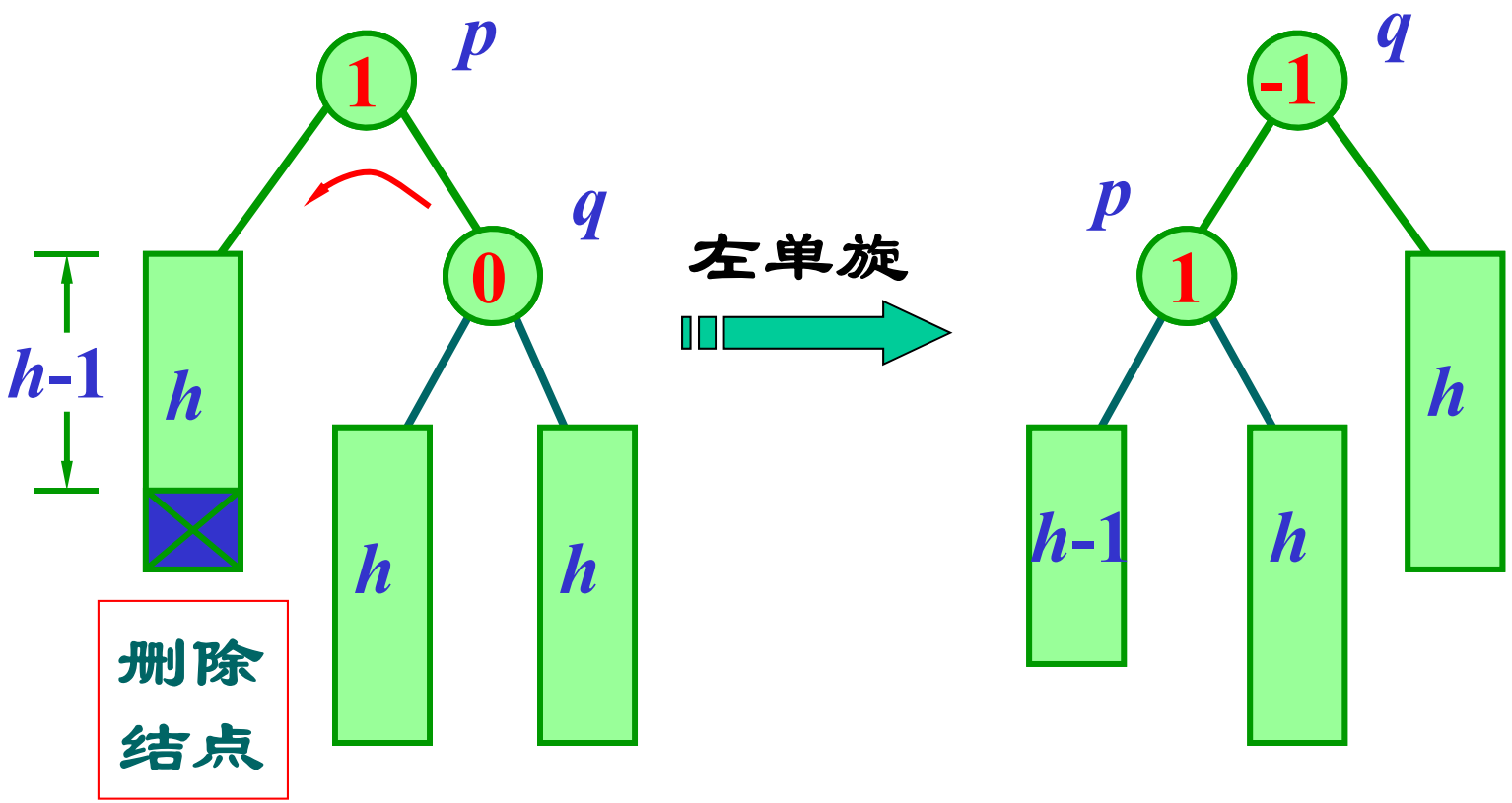
- case 2：结点*p*的balance不为0，且较高的子树被缩短，则*p*的balance改为0，同时shorter置为true。



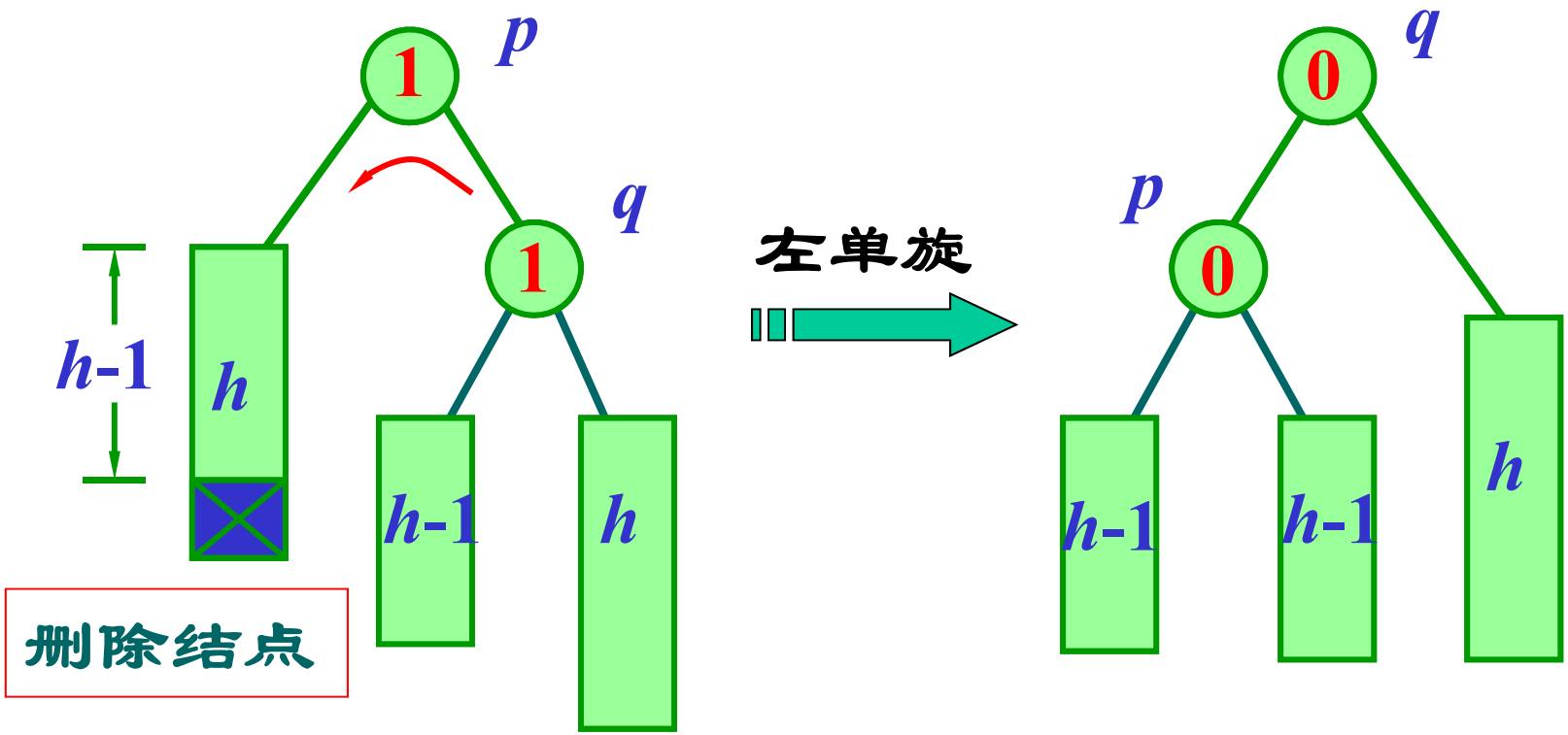
删除一个结点
高度降低一层

- 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。

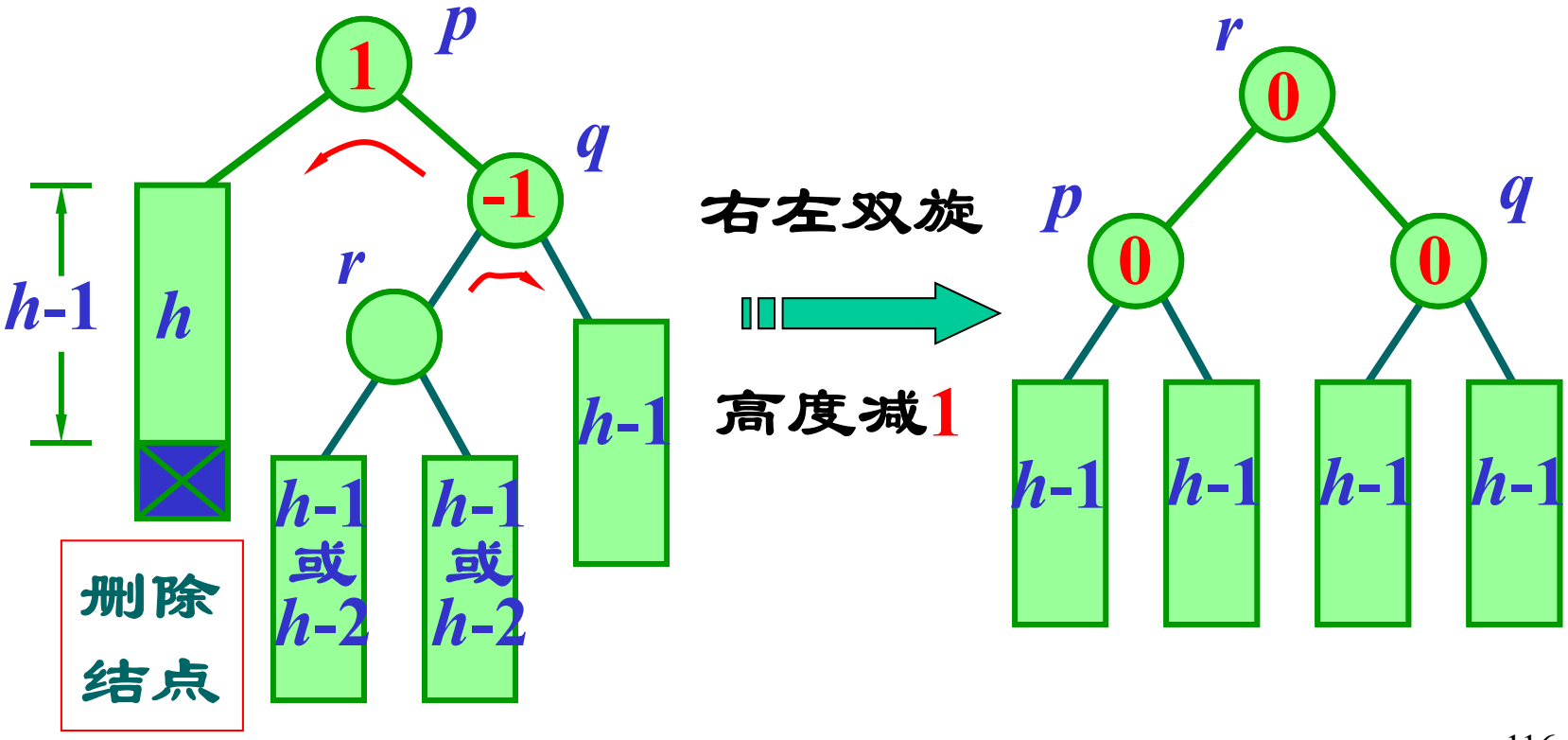
- case 3a：如果*q*（较高的子树）的balance为0，执行一个单旋转来恢复结点*p*的平衡，置shorter为false。



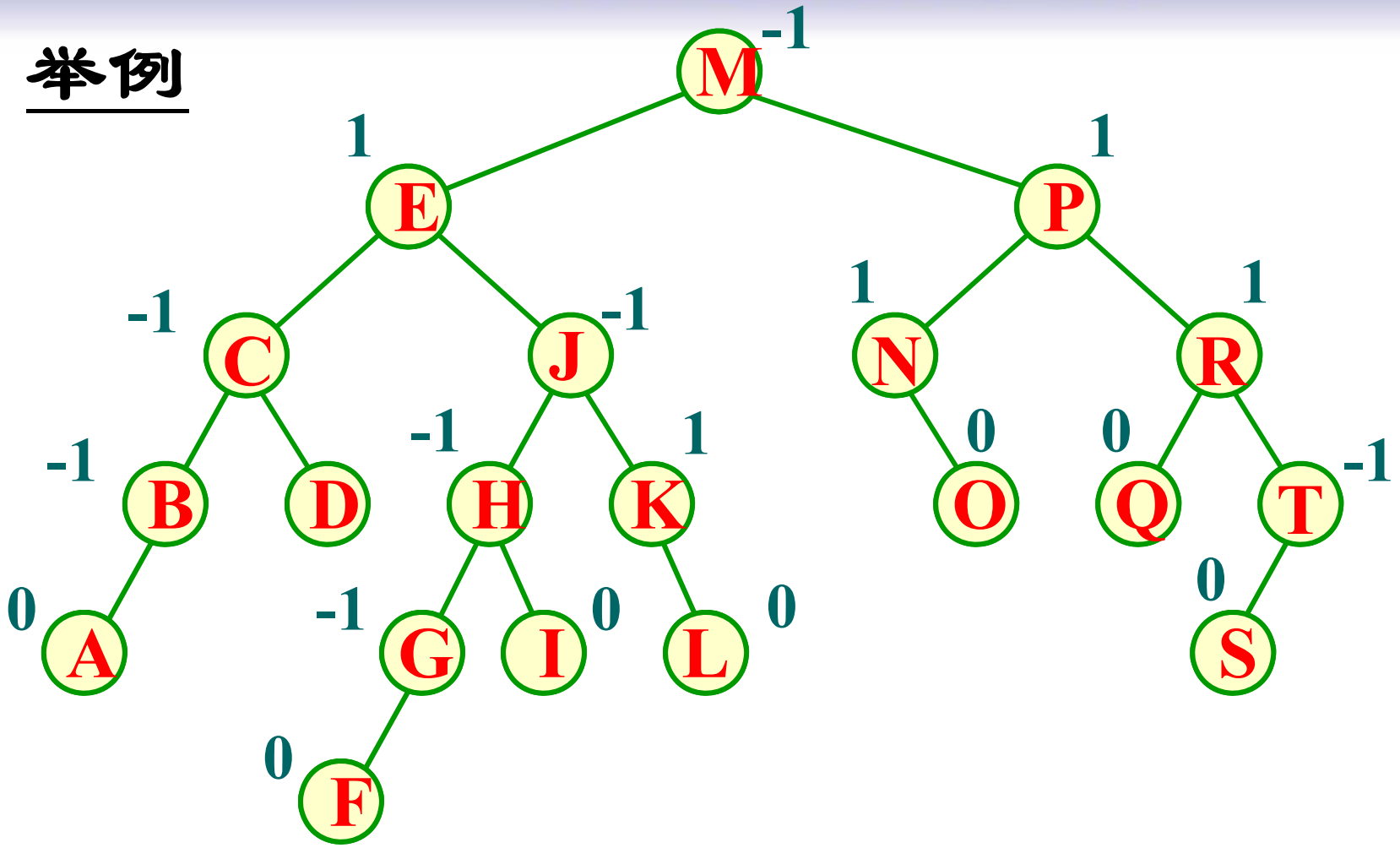
- case 3b：如果*q*的balance与*p*的balance相同，则执行一个单旋转来恢复平衡，结点*p*和*q*的balance均改为0，同时置shorter为true。



- 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。

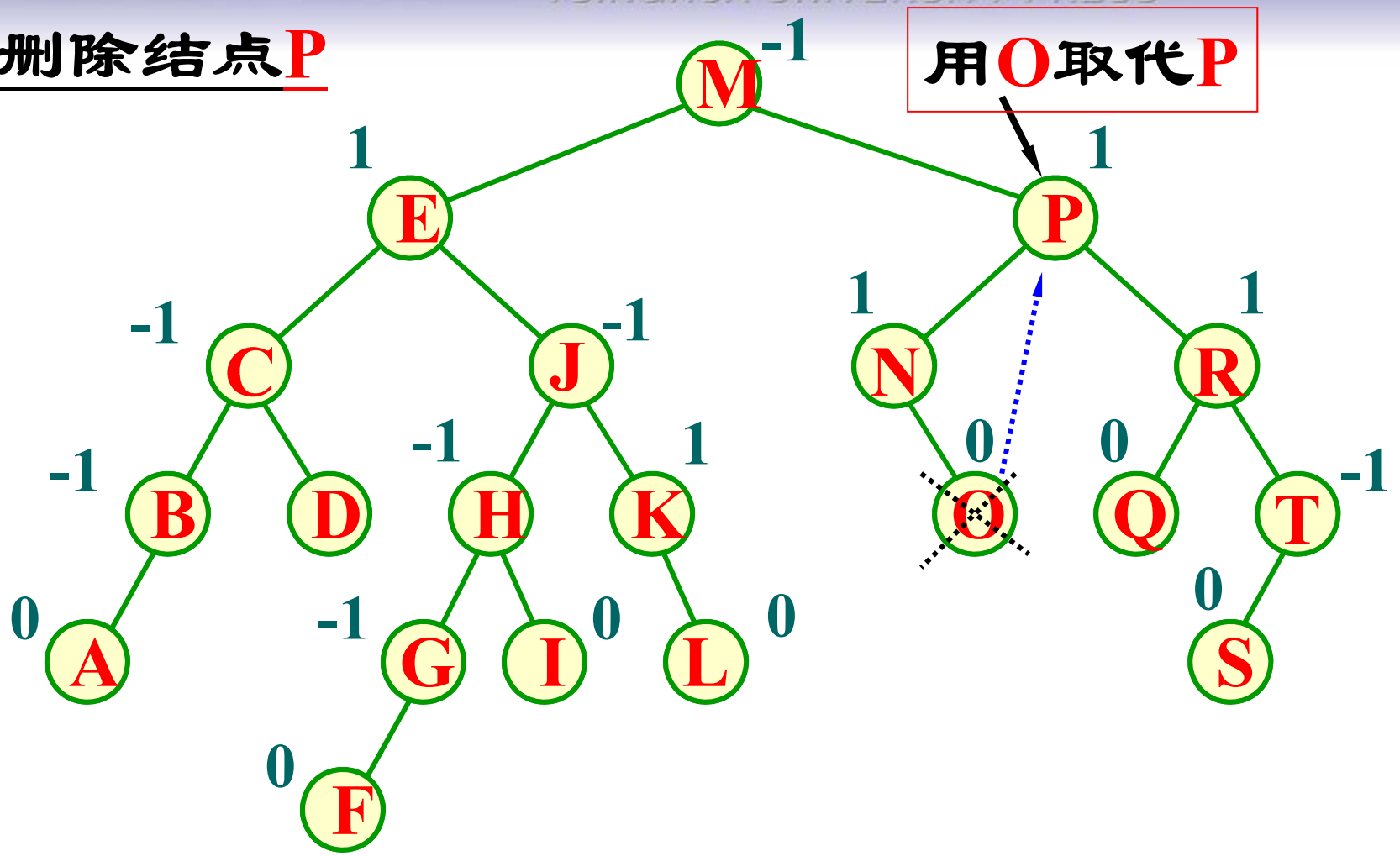


举例



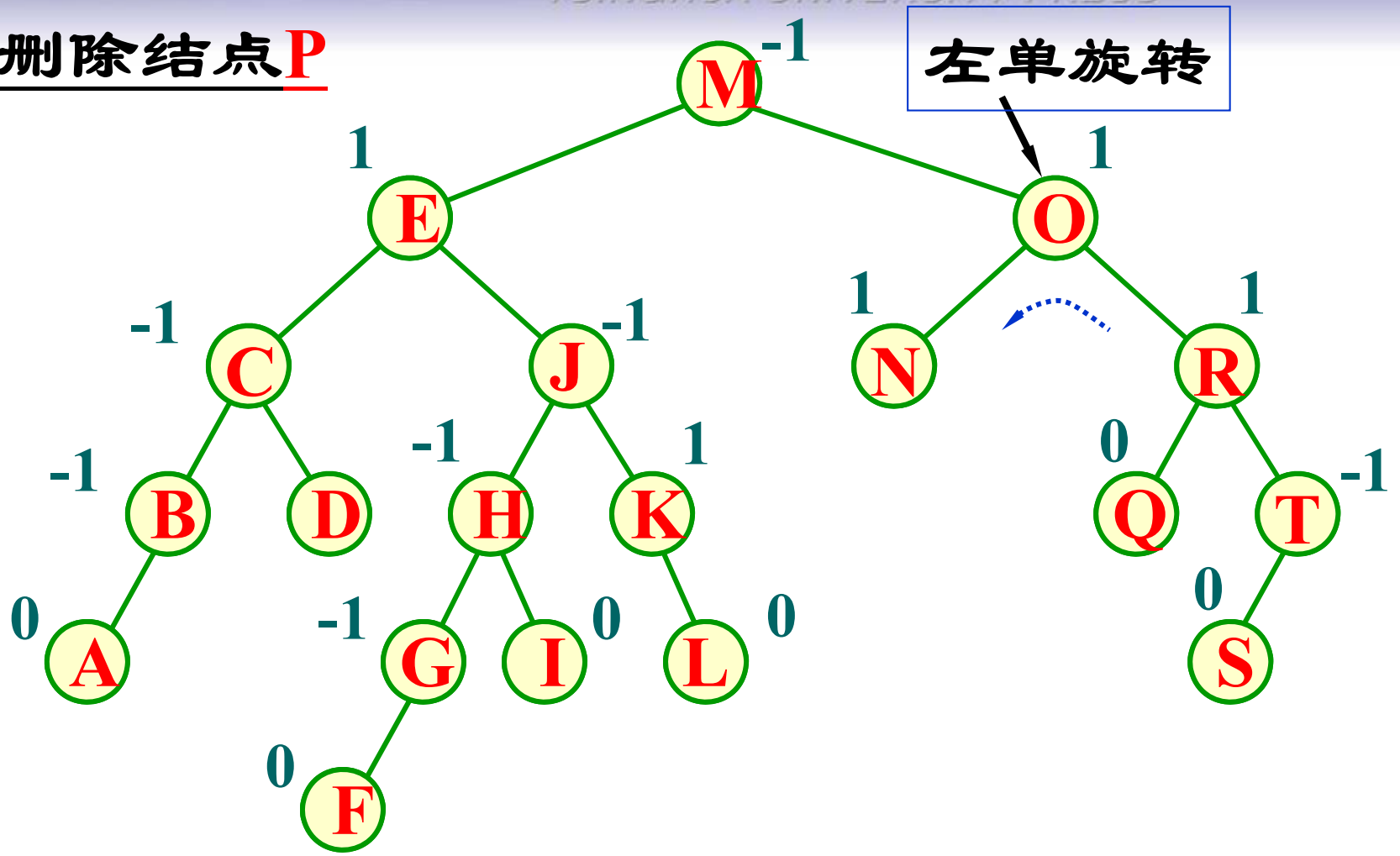
树的初始状态

删除结点P



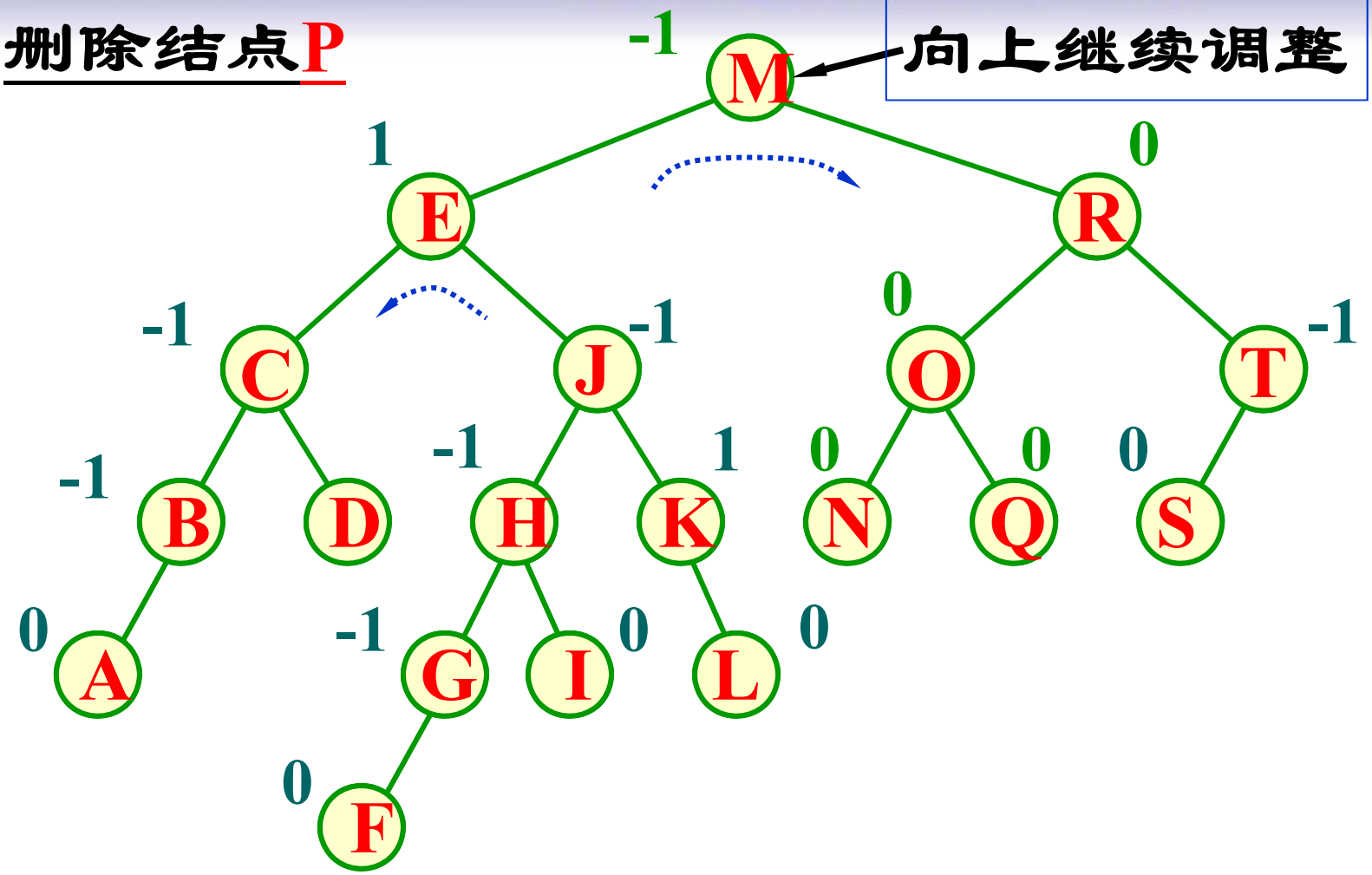
寻找结点P在中序下的直接前驱O，用O顶替P，删除O。

删除结点P



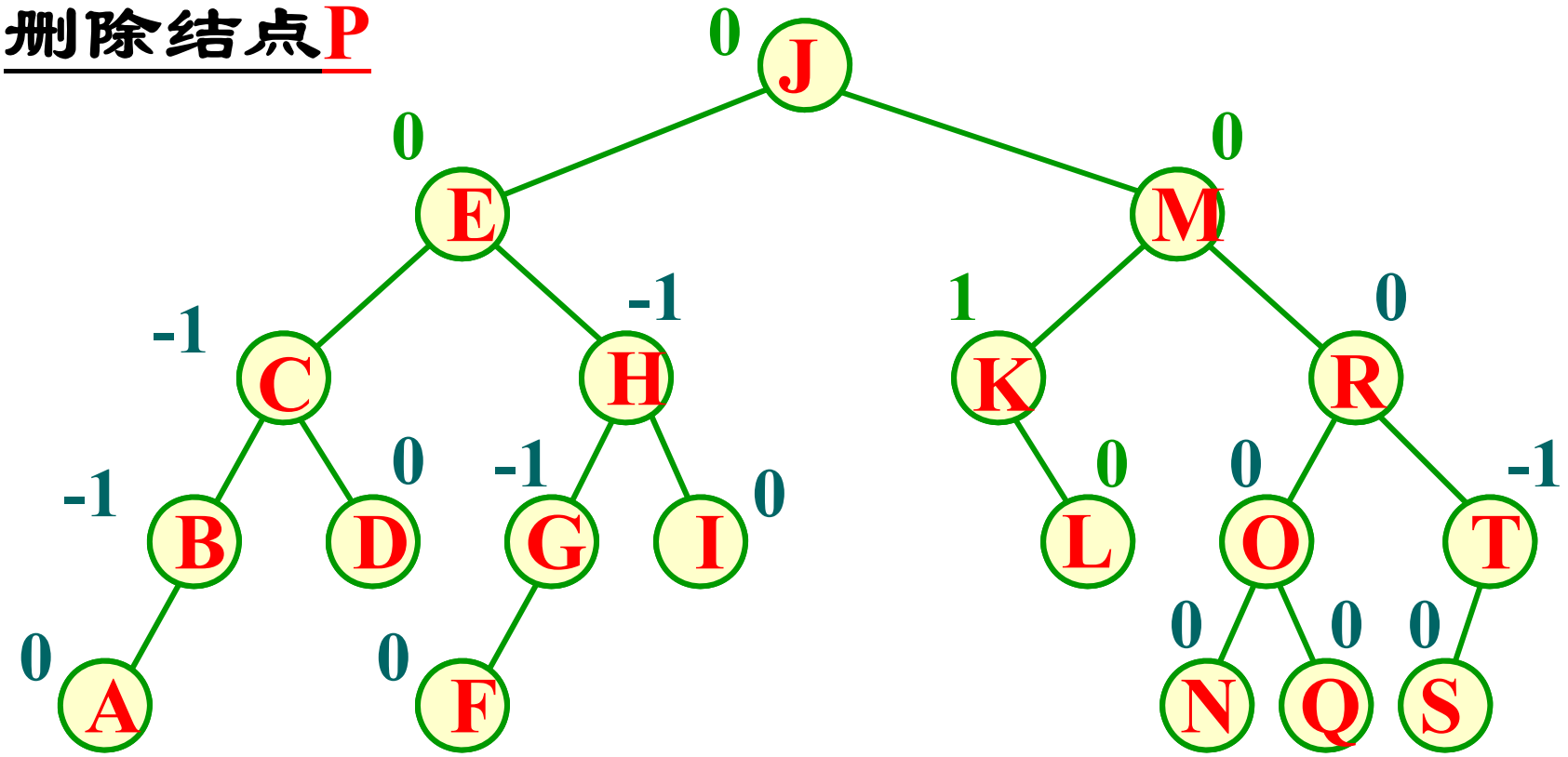
O与R的平衡因子同号，以R为旋转轴做左单旋转，M的子树高度减1。

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



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。

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

- **课程习题**

  - 笔做题——7.2, 7.3, 7.4, 7.14, 7.28
    （以作业形式提交）

  - 上机题——7.23, 7.24, 7.26

  - 思考题——7.13, 7.15, 7.16, 7.18, 7.30, 7.31