

第七章 集合与搜索

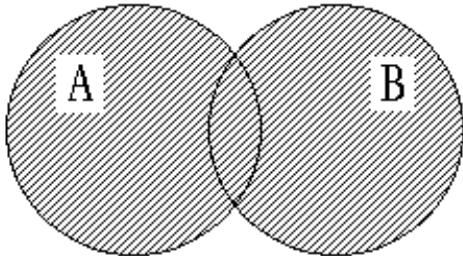
- 集合及其表示
- 等价类与并查集
- 静态搜索表
- 二叉搜索树
- 最优二叉搜索树
- AVL树
- 小结

集合及其表示

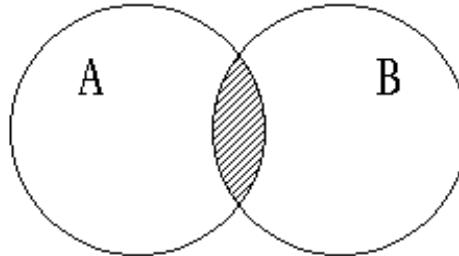
集合基本概念

- 集合是成员(对象或元素)的一个群集。集合中的成员可以是原子(单元素)，也可以是集合。
- 集合的成员必须互不相同。
- 在算法与数据结构中所遇到的集合，其单元素通常是整数、字符、字符串或指针，且同一集合中所有成员具有相同的数据类型。
- `colour = { red, orange, yellow, green, black, blue, purple, white }` `name = { “An”, “Cao”, “Liu”, “Ma”, “Peng”, “Wang”, “zhang” }`

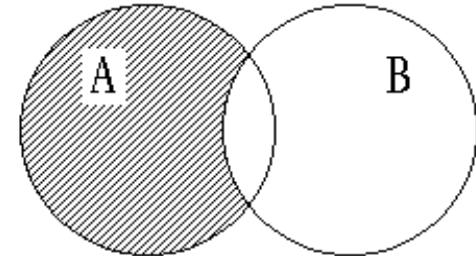
- 集合中的成员一般是无序的，没有先后次序关系。但在表示它时，常常写在一个序列里。
- 我们常设定集合中的单元素具有线性有序关系，此关系可记作“ $<$ ”，表示“优先于”。
 - 若 a, b 是集合中的两个单元素，则 $a < b, a == b, a > b$ 三者必居其一；
 - 若 a, b, c 是集合中的三个单元素，且 $a > b, b > c$ ，则 $a > c$ 。（传递性）
- 整数、字符和字符串都有一个自然的线性顺序。而指针也可以依据其在序列中安排的位置给予一个线性顺序。
- 集合操作有求集合的并、交、差、判存在等。



(a) $A \cup B$ (或 $A+B$)



(b) $A \cap B$ (或 $A \times B$)



(c) $A-B$

集合运算的文氏(Venn)图

集合(*Set*)的抽象数据类型

```
Template <class Type> class Set {  
    Set ( int MaxSetSize ) : MaxSize ( MaxSetSize );  
    void MakeEmpty ( Set &s );  
    int AddMember ( Set &s, const Type x );  
    int DelMember ( Set &s, const Type & x );
```

```
void Assign ( Set& s1, Set& s2 );
void Union ( Set& s1, Set& s2 );
void Intersection ( Set& s1, Set& s2 );
void Difference ( Set& s1, Set& s2 );
int Contains ( Set &s, const Type & x );
int Equal ( Set &s1, Set &s2 );
int SubSet ( Set &s1, Set &s2 );
}
```

用位向量实现集合抽象数据类型

- 当集合是全集合{ 0, 1, 2, ..., n }的一个子集，且 n 是不大的整数时，可用位(0, 1)向量来实现集合。
- 当全集合是由有限的可枚举的成员组成的集合时，可建立全集合成员与整数 0, 1, 2, ... 的一对对应关系，用位向量来表示该集合的子集。

集合的位向量(bit Vector)类的定义

```
#include <assert.h>
const int DefaultSize = 100;
class Set {
```

private:

int * bitVector;

int MaxSize;

public:

Set (int MaxSetSize = DefaultSize);

~Set () { delete [] bitVector; }

void MakeEmpty () {

for (int i = 0; i < MaxSize; i++)

bitVector[i] = 0;

}

int AddMember (const int x);

int DelMember (const int x);

Set& operator = (Set & right);

```
Set& operator + ( Set & right );
Set& operator * ( Set & right );
Set& operator - ( Set & right );
int Contains ( const int x );
int SubSet ( Set & right );
int operator == ( Set & right );
};
```

使用示例

```
Set s1, s2, s3, s4, s5; int index, equal; //构造集合
for ( int k = 0; k < 10; k++ )           //集合赋值
{ s1.AddMember( k ); s2.AddMember( k +7 ); }
// s1 : { 0, 1, 2, ..., 9 }, s2 : { 7, 8, 9, ..., 16 }
```

```
s3 = s1 + s2; //求s1与s2的并 { 0, 1, ..., 16 }
s4 = s1 * s2; //求s1与s2的交 { 7, 8, 9 }
s5 = s1 - s2; //求s1与s2的差 { 0, 1, 2, 3, 4, 5, 6 }
index = s1.SubSet( s4 ); //求s4在s1中首次匹配
cout << index << endl; //位置, index = 7
equal = s1 == s2; //集合s1与s2比较相等
cout << equal << endl; //equal为0, 两集合不等
```

用位向量实现集合时部分操作的实现

```
Set :: Set (int MaxSetSize) : MaxSize (MaxSetSize) {
    assert ( MaxSize > 0 );
    bitVector = new int [MaxSize];
    assert ( bitVector != 0 );
```

```
for ( int i = 0; i < MaxSize; i++ ) bitVector[i] = 0;  
}  
  
int Set :: Addmember ( const int x ) {  
    assert ( x >= 0 && x < MaxSize );  
    if ( !bitVector[x] ) { bitVector[x] = 1; return 1; }  
    return 0;  
}  
  
int Set :: DelMember ( const int x ) {  
    assert ( x >= 0 && x < MaxSize );  
    if ( bitVector[x] ) { bitVector[x] = 0; return 1; }  
    return 0;  
}
```

```
Set& Set :: operator = ( Set& right ) {  
    assert ( MaxSize == right.MaxValue );  
    for ( int i =0; i <MaxSize; i++ )  
        bitVector[i] = right.bitVector[i];  
    return *this;  
}
```

```
Set& Set :: operator + ( Set& right ) {  
    assert ( MaxSize == right.MaxValue );  
    for ( int i = 0; i < MaxSize; i++ )  
        bitVector[i] = bitVector[i] || right.bitVector[i];  
    return *this;  
}
```

```
Set& Set :: operator * ( Set & right ) {
    assert ( MaxSize == right.MaxValue );
    for ( int i = 0; i < MaxSize; i++)
        bitVector[i] = bitVector[i] && right.bitVector[i];
    return *this;
}
```

```
Set& Set :: operator - ( Set & right ) {
    assert ( MaxSize == right.MaxValue );
    for ( int i = 0; i < MaxSize; i++)
        bitVector[i] = bitVector[i] && !right.bitVector[i];
    return *this;
}
```

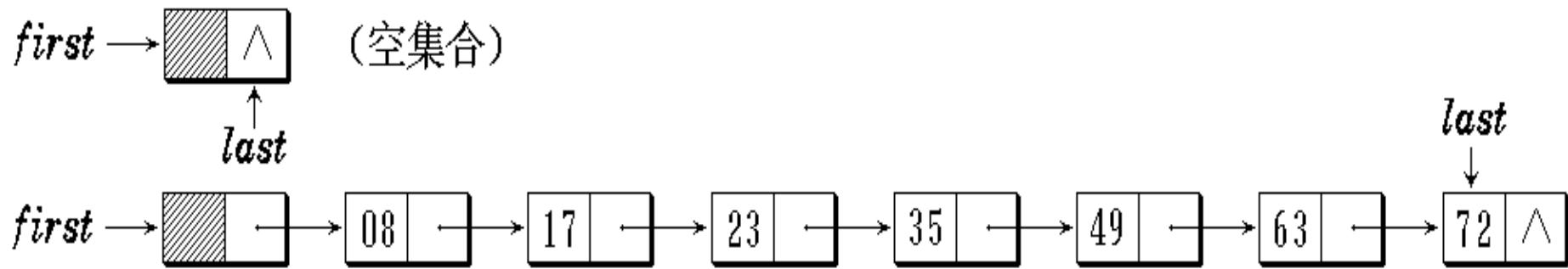
```
int Set :: Contains ( const int x ) {  
    assert ( x >= 0 && x < MaxSize );  
    return bitVector[x];  
}  
  
int Set :: operator == ( Set & right ) {  
    assert ( MaxSize == right.MaxValue );  
    for ( int i = 0; i < MaxSize; i++ )  
        if ( bitVector[i] != right.bitVector[i] ) return 0;  
    return 1;  
}  
  
int Set :: SubSet ( Set & right ) {  
    assert ( MaxSize == right.MaxValue );
```

```

for ( int i = 0; i < MaxSize; i++)
    if (bitVector[i] && ! right.bitVector[i]) return 0;
return 1;
}

```

用有序链表实现集合的抽象数据类型



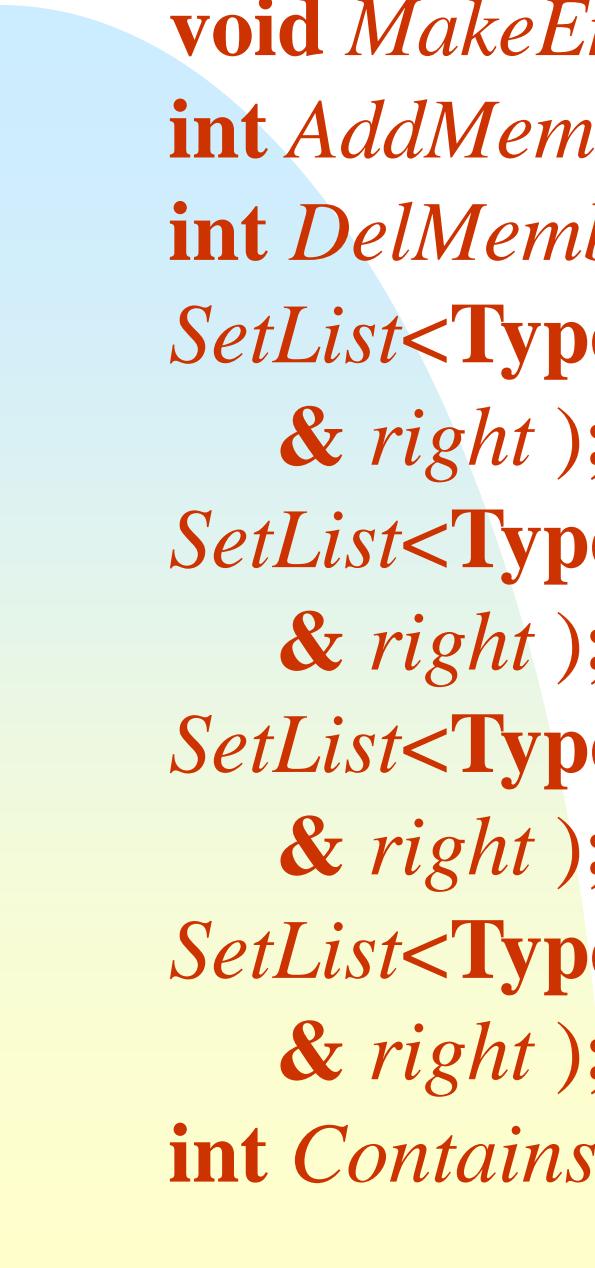
用带表头结点的有序链表表示集合

- 用有序链表来表示集合时，链表中的每个结点表示集合的一个成员。
- 各结点所表示的成员 e_0, e_1, \dots, e_n 在链表中按升序排列，即 $e_0 < e_1 < \dots < e_n$ 。
- 在一个有序链表中寻找一个集合成员时，一般不用搜索整个链表，搜索效率可以提高很多。
- 集合成员可以无限增加。因此，用有序链表可以表示无穷全集合的子集。

集合的有序链表类的定义

template <class Type> class *SetList*;

```
template <class Type> class SetNode {  
    friend class SetList<Type>;  
public:  
    SetNode (const Type & item ) :  
        data (item), link (NULL);  
private:  
    Type data;  
    SetNode<Type> *link;  
};  
  
template <class Type> class SetList {  
private:  
    SetNode<Type> *first, *last;  
public:
```



SetList ();
void *MakeEmpty* ();
int *AddMember* (**const** Type & *x*);
int *DelMember* (**const** Type & *x*);
SetList<Type> & operator = (*SetList*<Type>
 & *right*);
SetList<Type> & operator + (*SetList*<Type>
 & *right*);
SetList<Type> & operator * (*SetList*<Type>
 & *right*);
SetList<Type> & operator - (*SetList*<Type>
 & *right*);
int *Contains* (**const** Type & *x*);

```
int operator == ( SetList<Type> & right );
Type & Min ( );
Type & Max ( );
}
```

集合构造函数

```
template <class Type> void SetList<Type> ::  
SetList ( ) {  
    SetNode<Type> *first = *last =  
        new SetNode<Type>(0);  
}
```

集合的搜索、加入和删除操作

```
template <class Type>
int SetList<Type> ::  
Contains (const Type & x ) {  
    SetNode<Type> *temp = first->link;  
    while ( temp != NULL && temp->data < x )  
        temp = temp->link;  
    if (temp != NULL && temp->data == x)  
        return 1;  
    else return 0;  
}
```

```
template <class Type> int SetList<Type> ::  
AddMember ( const Type & x ) {  
    SetNode<Type> *p = first->link, *q = first;  
    while ( p != NULL && p->data < x )  
        { q = p; p = p->link; }  
    if ( p != NULL && p->data == x ) return 0;  
    SetNode<Type> *s = new SetNode (x);  
    s->link = p; q->link = s;  
    if ( p == NULL ) last = s;  
    return 1;  
}
```

```
template <class Type> int SetList<Type> ::  
DelMember ( const Type & x ) {  
    SetNode<Type> *p = first->link, *q = first;  
    while ( p != NULL && p->data < x )  
        { q = p; p = p->link; }  
    if ( p != NULL && p->data == x ) {  
        q->link = p->link;  
        if ( p == last ) last = q;  
        delete p; return 1;  
    }  
    else return 0;  
}
```

用有序链表表示集合时的几个重载函数

```
template <class Type>
SetList<Type>& SetList <Type> ::  
operator = ( SetList<Type> & right ) {  
    SetNode<Type>*pb = right.first->link;  
    SetNode<Type>*pa = first =  
        new SetNode<Type>;  
    while ( pb != NULL ) {  
        pa->link = new SetNode<Type> ( pb->data );  
        pa = pa->link; pb = pb->link;  
    }  
    pa->link = NULL; last = pa;  
    return *this;  
}
```

```
template <class Type>
SetList<Type>& SetList<Type> ::  

operator + ( SetList<Type> & right ) {  

    SetNode<Type> *pb = right.first->link;  

    SetNode<Type> *pa = first->link;  

    SetNode<Type> *pc = first;  

    while ( pa != NULL && pb != NULL ) {  

        if ( pa->data == pb->data )  

            { pc->link = pa; pa = pa->link;  

             pb = pb->link; }  

        else if ( pa->data < pb->data )  

            { pc->link = pa; pa = pa->link; }  

        else
```

```
{ pc->link = new SetNode<Type> (pb->data);
  pb = pb->link; }
pc = pc->link;
}
if ( pa != NULL ) pc->link = pa;
else {
  while ( pb != NULL ) {
    pc->link = new SetNode<Type> (pb->data);
    pc = pc->link; pb = pb->link;
  }
  pc->link = NULL; last = pc;
}
return *this;
}
```

```
template <class Type>
SetList<Type>& SetList<Type>::
operator * ( SetList<Type> & right ) {
    SetNode<Type> *pb = right.first->link;
    SetNode<Type> *pa = first->link;
    SetNode<Type> *pc = first;
    while ( pa != NULL && pb != NULL ) {
        if ( pa->data == pb->data )
            { pc = pc->link; pa = pa->link;
              pb = pb->link; }
        else if ( pa->data < pb->data )
            { pc->link = pa->link; delete pa;
              pa = pc->link; }
        else pb = pb->link;
    }
}
```

```
while ( pa != NULL ) {  
    pc→link = pa→link; delete pa;  
    pa = pc→link;  
}  
last = pc;  
return *this;  
}
```

```
template <class Type>  
SetList<Type>& SetList<Type> ::  
operator - ( SetList<Type> & right ) {  
    SetNode<Type> *pb = right.first→link;  
    SetNode<Type> *pa = first→link;
```

```
SetNode<Type> *pc = first;
while ( pa != NULL && pb != NULL ) {
    if ( pa->data == pb->data )
        { pc->link = pa->link; delete pa;
          pa = pc->link; pb = pb->link; }
    else if ( pa->data < pb->data )
        { pc = pc->link; pa = pa->link; }
    else pb = pb->link;
}
if ( pa == NULL ) last = pc;
return *this;
}
```

```
template <class Type> int SetList<Type> ::  
operator == ( SetList<Type> & right ) {  
    SetNode<Type> *pb = right.first->link;  
    SetNode<Type> *pa = first->link;  
    while ( pa != NULL && pb != NULL )  
        if ( pa->data == pb->data )  
            { pa = pa->link; pb = pb->link; }  
        else return 0;  
    if ( pa != NULL | | pb != NULL ) return 0;  
    return 1;  
}
```



等价类与并查集

等价关系与等价类(Equivalence Class)

- 在求解实际应用问题时常会遇到等价类问题。
- 从数学上看，等价类是一个对象(或成员)的集合，在此集合中所有对象应满足等价关系。
- 若用符号“ \equiv ”表示集合上的等价关系，那么对于该集合中的任意对象 x, y, z ，下列性质成立：
 - 自反性： $x \equiv x$ (即等于自身)。
 - 对称性：若 $x \equiv y$ ，则 $y \equiv x$ 。
 - 传递性：若 $x \equiv y$ 且 $y \equiv z$ ，则 $x \equiv z$ 。
- 因此，等价关系是集合上的一个自反、对称、传递的关系。

- “相等” ($=$)就是一种等价关系，它满足上述的三个特性。
- 一个集合 S 中的所有对象可以通过等价关系划分为若干个互不相交的子集 S_1, S_2, S_3, \dots ，它们的并就是 S 。这些子集即为等价类。

确定等价类的方法

分两步走。

第一步，读入并存储所有的等价对(i, j);

第二步，标记和输出所有的等价类。

```
void equivalence () {  
    初始化;  
    while 等价对未处理完  
        { 读入下一个等价对 ( $i, j$ );  
            存储这个等价对 ; }  
    输出初始化;  
    for ( 尚未输出的每个对象 )  
        输出包含这个对象的等价类 ;  
}
```

给定集合 $S = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \}$,
及如下等价对: $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4,$
 $6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

初始 $\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\},$
 $\{10\}, \{11\}$

$0 \equiv 4$ $\{0,4\}, \{1\}, \{2\}, \{3\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}, \{11\}$

$3 \equiv 1$ $\{0,4\}, \{1,3\}, \{2\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}, \{11\}$

$6 \equiv 10$ $\{0,4\}, \{1,3\}, \{2\}, \{5\}, \{6,10\}, \{7\}, \{8\}, \{9\}, \{11\}$

$8 \equiv 9$ $\{0,4\}, \{1,3\}, \{2\}, \{5\}, \{6,10\}, \{7\}, \{8,9\}, \{11\}$

$7 \equiv 4$ $\{0,4,7\}, \{1,3\}, \{2\}, \{5\}, \{6,10\}, \{8,9\}, \{11\}$

$6 \equiv 8$ $\{0,4,7\}, \{1,3\}, \{2\}, \{5\}, \{6,8,9,10\}, \{11\}$

$3 \equiv 5$ $\{0,4,7\}, \{1,3,5\}, \{2\}, \{6,8,9,10\}, \{11\}$

$2 \equiv 11$ $\{0,4,7\}, \{1,3,5\}, \{2,11\}, \{6,8,9,10\}$

$11 \equiv 0$ $\{0,2,4,7,11\}, \{1,3,5\}, \{6,8,9,10\}$

确定等价类的链表方法

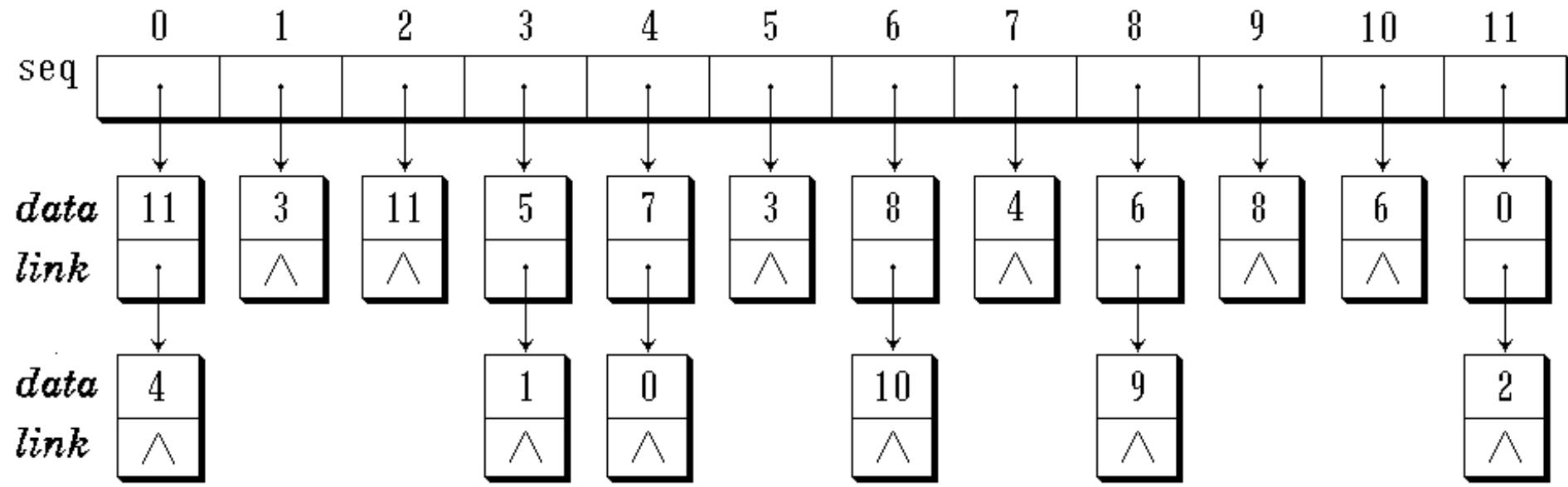
设等价对个数为 m ，对象个数为 n 。一种可选的存储表示为单链表。

可为集合的每一对象建立一个带表头结点的单链表，并建立一个一维的指针数组 $\text{seq}[n]$ 作为各单链表的表头结点向量。 $\text{seq}[i]$ 是第 i 个单链表的表头结点，第 i 个单链表中所有结点的 data 域存放在等价对中与 i 等价的对象编号。

当输入一个等价对 (i, j) 后，就将集合元素 i 链入第 j 个单链表，且将集合元素 j 链入第 i 个单链表。在输出时，设置一个布尔数组 $\text{out}[n]$ ，用 $\text{out}[i]$ 标记第 i 个单链表是否已经输出。

```
void equivalence () {  
    读入 n;  
    将 seq 初始化为 0 且将 out 初始化为 False;  
    while 等价对未处理完 {  
        读入下一个等价对( i, j );  
        将 j 链入 seq[i]链表; 将 i 链入 seq[j]链表;  
    }  
    for ( i = 0; i < n; i++ )          //检测所有对象  
        if ( out[i] == False ) {  
            out[i] = True;           //若对象i未输出  
            输出包含对象 i 的等价类;  
        }  
}
```

- 算法的输出从编号 $i = 0$ 的对象开始，对所有的对象进行检测。
- 在 $i = 0$ 时，循第0个单链表先找出形式为 $(0, j)$ 的等价对，把 0 和 j 作为同一个等价类输出。再根据等价关系的传递性，找所有形式为 (j, k) 的等价对，把 k 也纳入包含 0 的等价类中输出。如此继续，直到包含 0 的等价类完全输出为止。
- 接下来再找一个未被标记的编号，如 $i = 1$ ，该对象将属于一个新的等价类，我们再用上述方法划分、标记和输出这个等价类。
- 在算法中使用了一个栈。每次输出一个对象编号时，都要把这个编号进栈，记下以后还要检测输出的等价对象的单链表。



输入所有等价对后的seq数组及各单链表的内容

链 序号	等价 对	OUT 初态	输出	OUT 终态	栈
0		False	0	True	
0	11	False	11	True	11
0	4	False	4	True	11,4
4	7	False	7	True	11,7
4	0	True	—	True	11,7

链 序号	等价 对	OUT 初态	输出	OUT 终态	栈
7	4	True	—	True	11
11	0	True	—	True	
11	0	True	—	True	
11	2	False	2	True	2
2	11	True	—	True	

等价类链表的定义

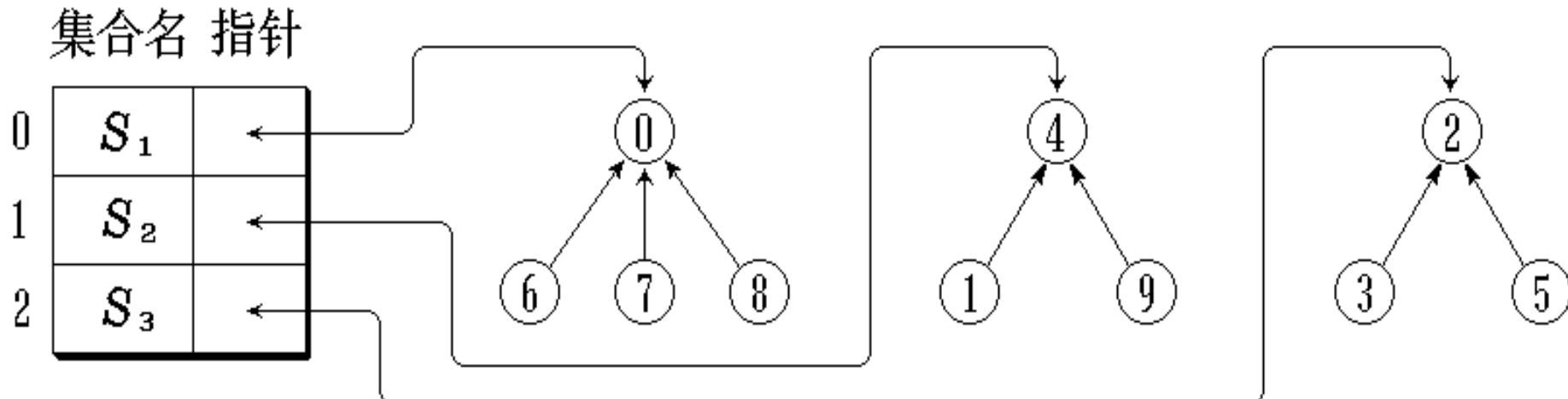
```
enum Boolean { False, True };

class ListNode {                                // 定义链表结点类
    friend void equivalence ();
    private:
        int data;                                // 结点数据
        ListNode *link;                          // 结点链指针
        ListNode ( int d ) { data = d; link = NULL; }
    };
    typedef ListNode *ListNodePtr;
```

并查集 (Union-Find Sets)

- 建立等价类的另一种解决方案是先把每一个对象看作是一个单元素集合，然后按一定顺序将属于同一等价类的元素所在的集合合并。
- 在此过程中将反复地使用一个搜索运算，确定一个元素在哪一个集合中。
- 能够完成这种功能的集合就是并查集。它支持以下三种操作：
 - *Union (Root1, Root2)* //并操作；
 - *Find (x)* //搜索操作；
 - *UFSets (s)* //构造函数。
- 一般情形，并查集主要涉及两种数据类型：集合名类型和集合元素的类型。

- 对于并查集来说，每个集合用一棵树表示。
- 集合中每个元素的元素名分别存放在树的结点中，此外，树的每一个结点还有一个指向其双亲结点的指针。
- 为此，需要有两个映射：
 - 集合元素到存放该元素名的树结点间的对应；
 - 集合名到表示该集合的树的根结点间的对应。
- 设 $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$



利用并查集来解决等价问题的步骤如下：

- 利用 ***UFSets*** 操作，建立 ***UFSets*** 型集合 **this**，集合中每一个元素初始化为 0，各自形成一个单元素子集合， $i = 1, 2, \dots, n$ 。 n 是集合中元素个数。
- 重复以下步骤，直到所有等价对读入并处理完为止。
 - 读入一个等价对 $[i][j]$ ；
 - 用 ***Find(i)***, ***Find(j)*** 搜索 i 、 j 所属子集合的名字 x 和 y ；
 - 若 $x \neq y$. 用 ***Union(x,y)*** 或 ***Union(y,x)*** 将它们合并，前者的根在 x ；后者的根在 y 。

- 为简化讨论，忽略实际的集合名，仅用表示集合的树的根来标识集合。
- 如果我们确定了元素 i 在根为 j 的树中，而且 j 有一个指向集合名字表中第 k 项的指针，则集合名即为 $name[k]$ 。
- 为此，采用树的双亲表示作为集合存储表示。集合元素的编号从0到 $n-1$ 。其中 n 是最大元素个数。在双亲表示中，第 i 个数组元素代表包含集合元素 i 的树结点。根结点的双亲为-1，表示集合中的元素个数。为了区别双亲指针信息(≥ 0)，集合元素个数信息用负数表示。

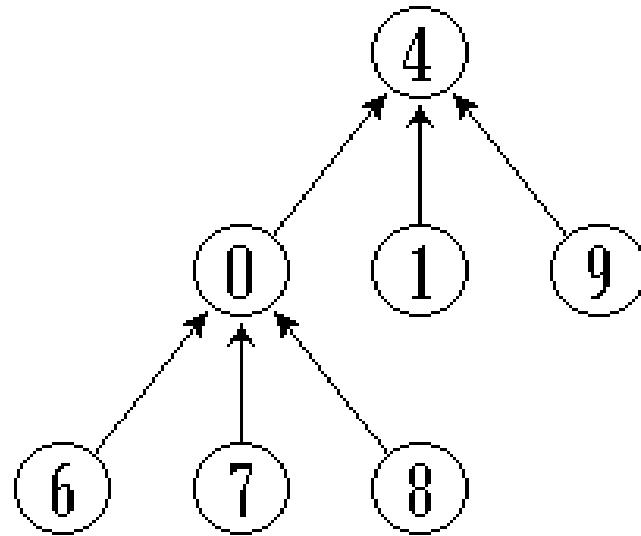
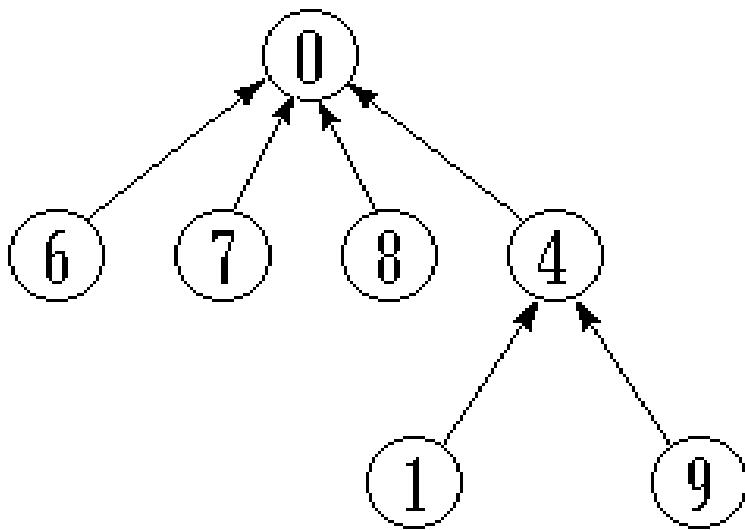
下标

0 1 2 3 4 5 6 7 8 9

parent

-1	4	-1	2	-1	2	0	0	0	4
----	---	----	---	----	---	---	---	---	---

集合 S_1 , S_2 和 S_3 的双亲表示



$S_1 + S_2$ 的可能的表示方法

```
const int DefaultSize = 10;
class UFSets {          //并查集的类定义
public:
    UFSets ( int s = DefaultSize );
    ~UFSets () { delete [ ] parent; }
    const UFSets & operator =
        ( UFSets const & Value );
    void Union ( int Root1, int Root2 );
    int Find ( int x );
    void UnionByHeight ( int Root1, int Root2 );
private:
    int *parent;
    int size;
};
```

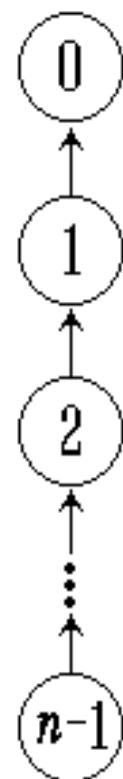
```
UFSets :: UFSets ( int s ) { //构造函数  
    size = s;  
    parent = new int [size+1];  
    for ( int i = 0; i <= size; i++ ) parent[i] = -1;  
}
```

```
unsigned int UFSets :: Find ( int x ) { //搜索操作  
    if ( parent[x] <= 0 ) return x;  
    else return Find ( parent[x] );  
}
```

```
void UFSets :: Union ( int Root1, int Root2 ) { //并  
    parent[Root2] = Root1; //Root2指向Root1  
}
```

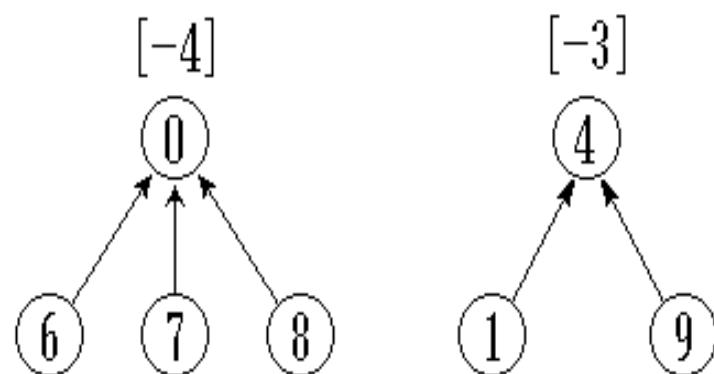
- ***Find***和***Union***操作性能不好。假设最初 n 个元素构成 n 棵树组成的森林， $parent[i] = -1$ 。做处理 $Union(n-2, n-1), \dots, Union(1, 2), Union(0, 1)$ 后，将产生如图所示的退化的树。
- 执行一次***Union***操作所需时间是 **$O(1)$** ， $n-1$ 次***Union***操作所需时间是 **$O(n)$** 。若再执行 $Find(0), Find(1), \dots, Find(n-1)$ ，若被搜索的元素为 i ，完成 $Find(i)$ 操作需要时间为 **$O(i)$** ，完成 n 次搜索需要的总时间将达到

$$O\left(\sum_{i=1}^n i\right) = O(n^2)$$

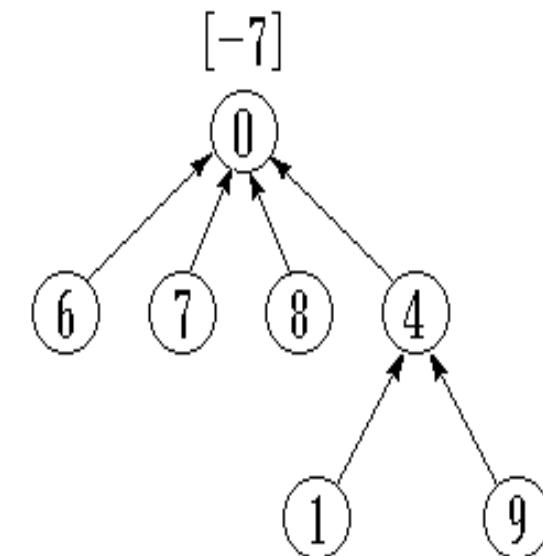


*Union*操作的加权规则

- 为避免产生退化的树，改进方法是先判断两集合中元素的个数，如果以 i 为根的树中的结点个数少于以 j 为根的树中的结点个数，即 $\text{parent}[i] > \text{parent}[j]$ ，则让 j 成为 i 的双亲，否则，让 i 成为 j 的双亲。此即 *Union* 的加权规则。



$\text{parent}[0] < \text{parent}[4]$
—————
 $\text{parent}[4] = 0$



$\text{parent}[0](== -4) < \text{parent}[4] (== -3)$

void UFSets ::

WeightedUnion (int Root1, int Root2) {

//按Union的加权规则改进的算法

int temp = parent[Root1] + parent[Root2];

if (parent[Root2] < parent[Root1]) {

parent[Root1] = Root2; //Root2中结点数多

parent[Root2] = temp; //Root1指向Root2

}

else {

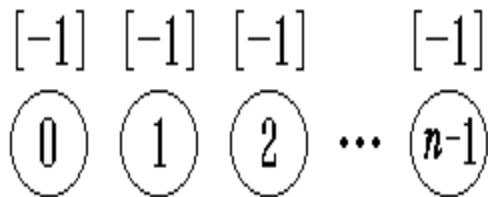
parent[Root2] = Root1; //Root1中结点数多

parent[Root1] = temp; //Root2指向Root1

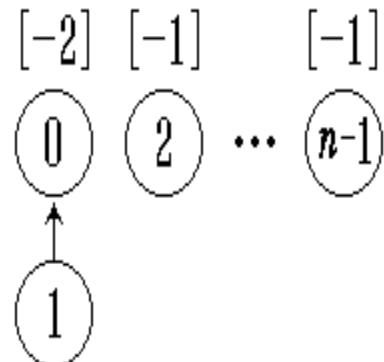
}

}

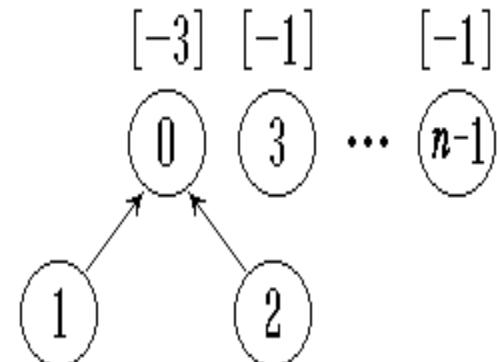
初始状态



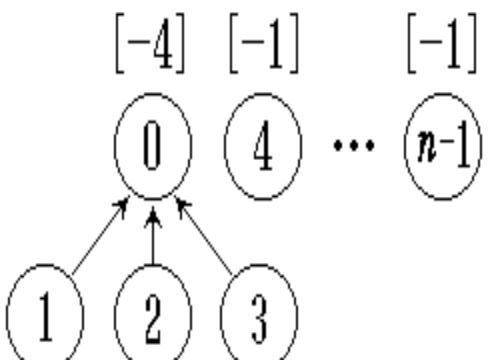
$Union(0, 1)$



$Union(0, 2)$

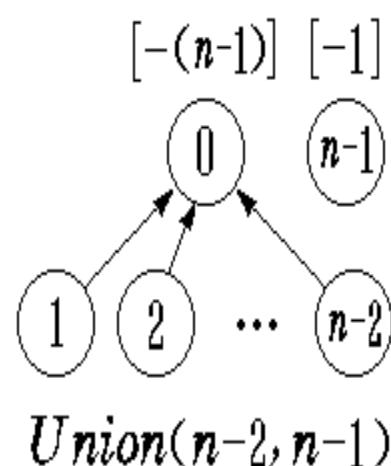


$Union(0, 3)$

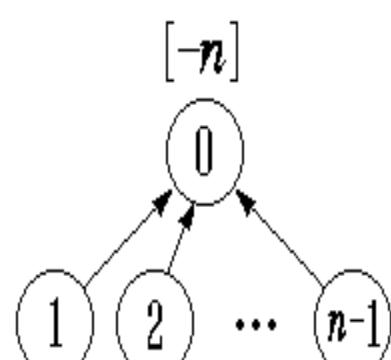


$Union(0, 4)$

.....

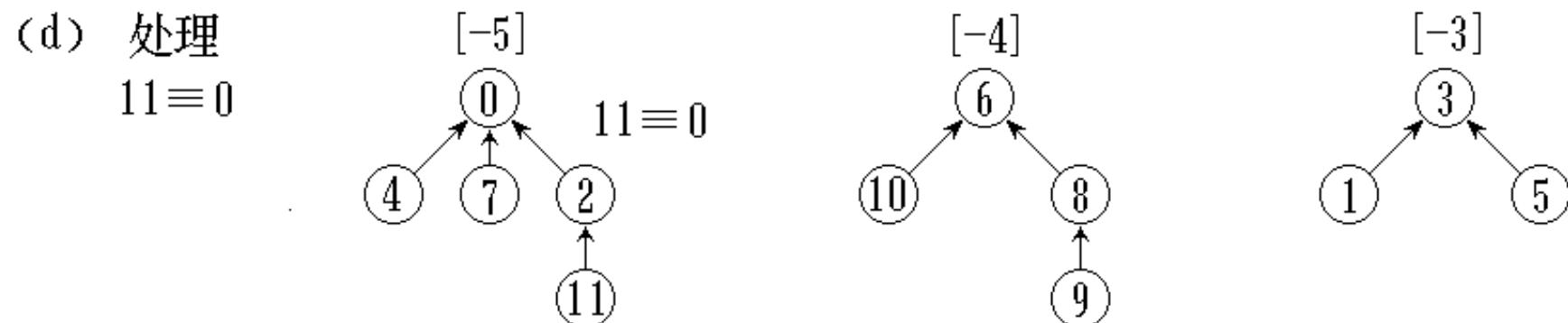
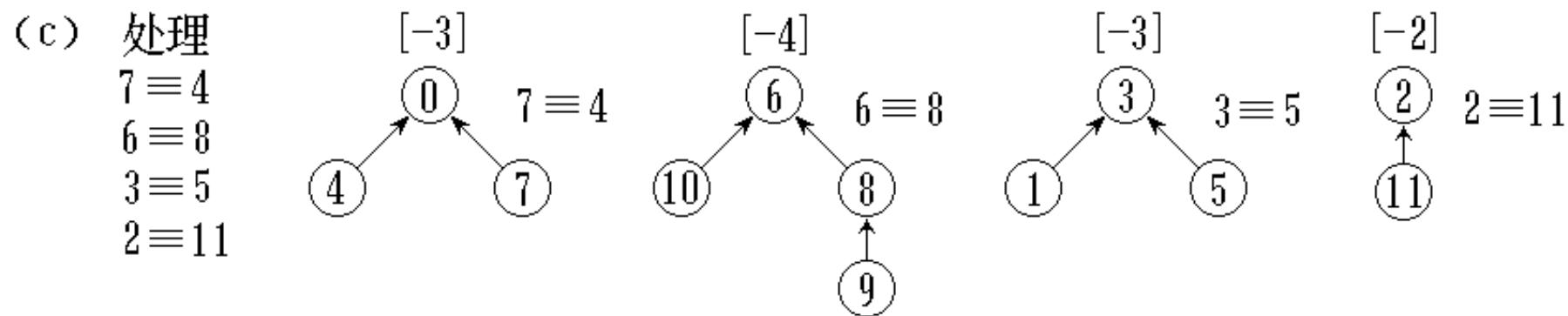
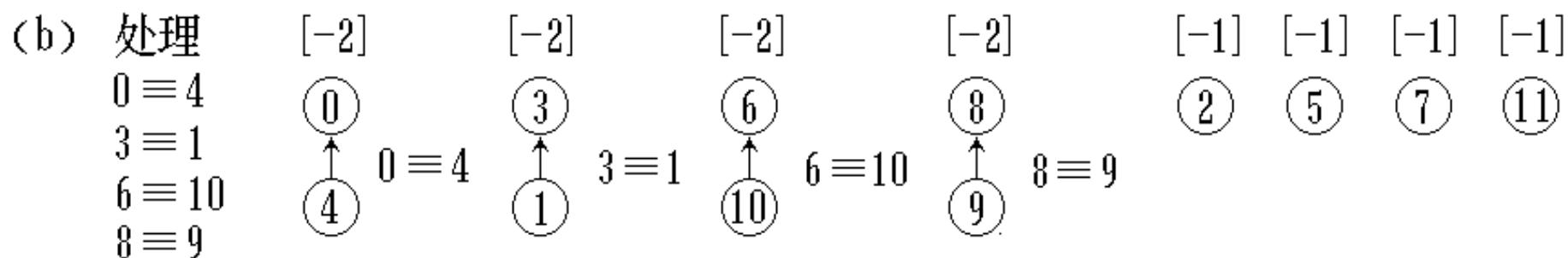
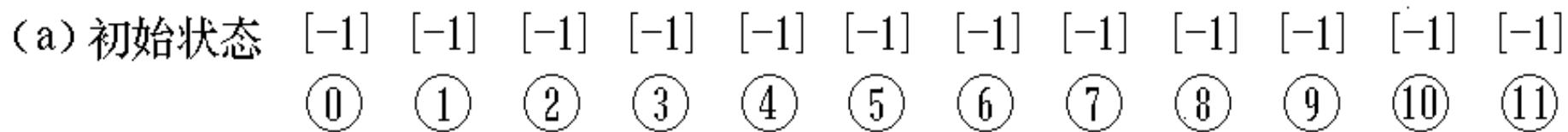


$Union(n-2, n-1)$



完成

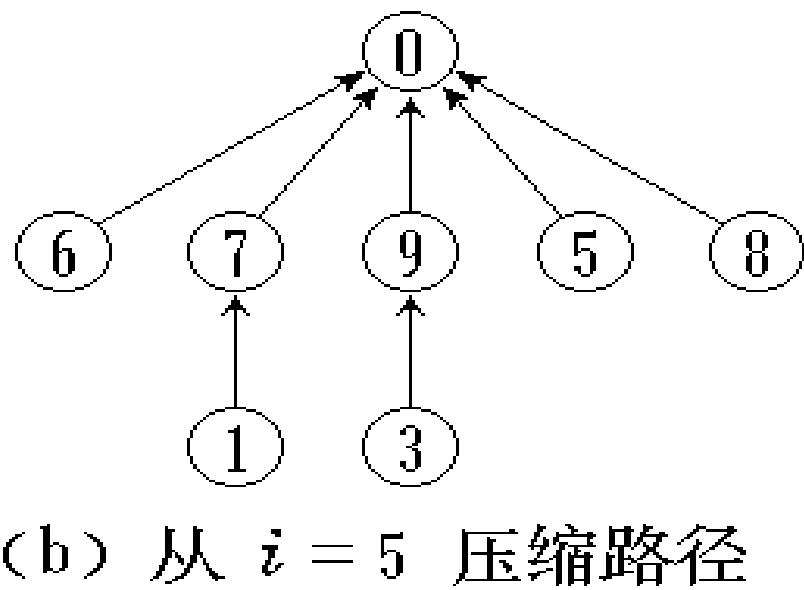
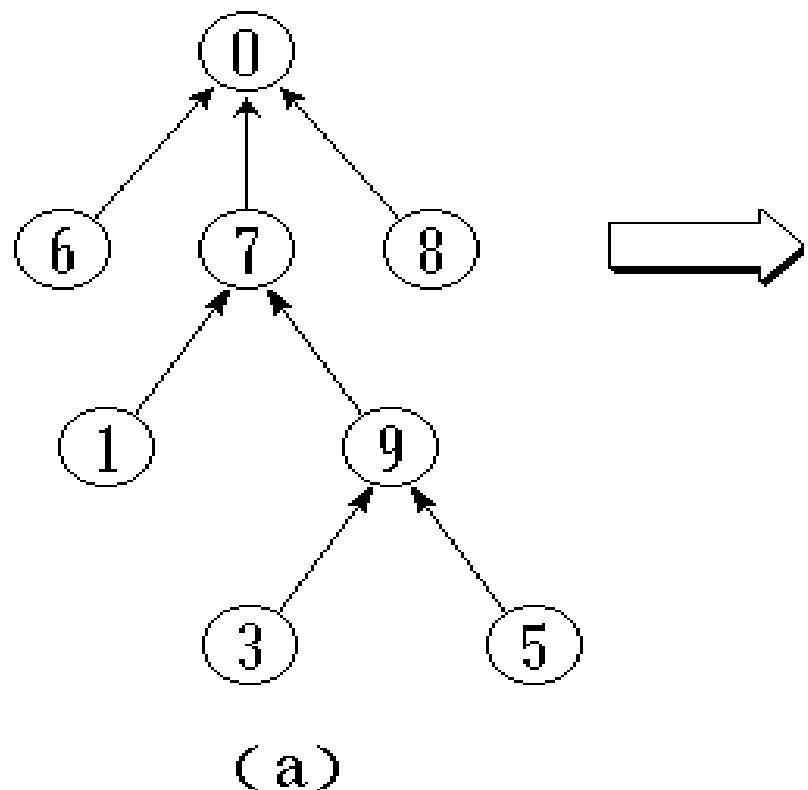
使用加权规则得到的树



使用并查集处理等价对，形成等价类的过程

*Union*操作的折叠规则

- 为进一步改进树的性能，可以使用如下折叠规则来“压缩路径”。即：如果 j 是从 i 到根的路径上的一个结点，且 $\text{parent}[j] \neq \text{root}[j]$ ，则把 $\text{parent}[j]$ 置为 $\text{root}[i]$ 。



```
int UFSets :: CollapsingFind ( int i ) {  
    //使用折叠规则的搜索算法  
    for ( int j = i; parent[j] >= 0; j = parent[j]);  
        //让 j 循双亲指针走到根  
    while ( i != j )      //换 parent[i] 到 j  
        { int temp = parent[i]; parent[i] = j; i = temp; }  
    return j;  
}
```

- 使用折叠规则完成单个搜索，所需时间大约增加一倍。但是，它能减少在最坏情况下完成一系列搜索操作所需的时间。



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。这时，作为结果，可报告该对象在结构中的位置，还可进一步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，也应报告一些信息，如失败标志、失败位置等。
- 通常称用于搜索的数据集合为搜索结构，它是由同一数据类型的对象(或记录)组成。

- 在每一个对象中有若干属性，其中应当有一个属性，其值可唯一地标识这个对象。它称为关键码。使用基于关键码的搜索，搜索结果应是唯一的。但在实际应用时，搜索条件是多方面的，这样，可以使用基于属性的搜索方法，但搜索结果可能不唯一。
- 实施搜索时有两种不同的环境。
 - ◆ 静态环境，搜索结构在执行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>;
```

```
public:
```

```
    Node ( const Type & value ) : key ( value ) { }
```

```
    Type getKey ( ) const { return key; }
```

```
    void setKey ( Type k ) { key = k; }
```

```
private:
```

```
    Type key;
```

```
    other;
```

```
};
```

```
template <class Type> class dataList {  
public:  
    dataList ( int sz = 10 ) : ArraySize (sz),  
                Element (new Node <Type> [sz]) { }  
    virtual ~dataList ( ) { delete [ ] Element; }  
    friend ostream &operator << (ostream &  
        OutStream, const dataList<Type> & OutList );  
    friend istream & operator >> ( istream &  
        InStream, dataList<Type> & InList );  
protected:  
    Type *Element;  
    int ArraySize, CurrentSize;  
};
```

```
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输入数据到数据表InList
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 ) {
//将数据表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)，这个标准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。
- 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。
- 在静态搜索表中，数据对象存放于数组中，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 在数组中的存放位置或可确定在数组中找不到 x 为止。

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> ::  
Search ( const Type & x ) const {  
//顺序搜索的迭代算法， 0号元素为监视哨  
Element[0].setKey ( x ); int i = CurrentSize;  
while ( Element[i].getKey ( ) != x ) i --;  
return i;  
}
```

```
template <class Type> int earchList<Type> ::  
Search ( const Type & x, int & loc ) const {  
//顺序搜索的递归算法  
if ( loc >= CurrentSize ) return -1;  
else if ( Element[loc].getKey( ) == x ) return loc;  
else return Search ( x, loc+1 );  
}
```

```
const int Size = 10;  
main () {  
//顺序搜索的主过程  
searchList<float> List1 (Size);  
    float Target;  int Location;  
    cin >> List1;  cout << List1;  
//输入/输出数据表List1  
    cout << “搜索浮点数：”;  cin >> Target;  
    if ( ( Location = List1.search (Target) ) != 0 )  
        cout << “找到下标 ” << Location << endl;  
    else  cout << “没有找到.\n”;  
}
```

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:

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

在顺序搜索情形, $c_i = i + 1$, $i = 0, 1, \dots, n-1$, 因此

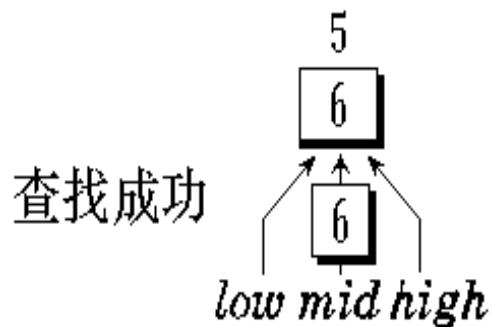
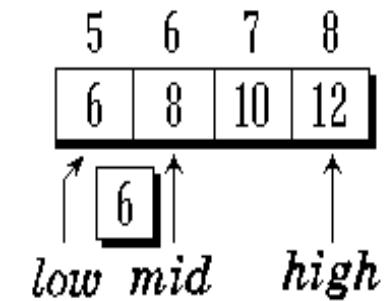
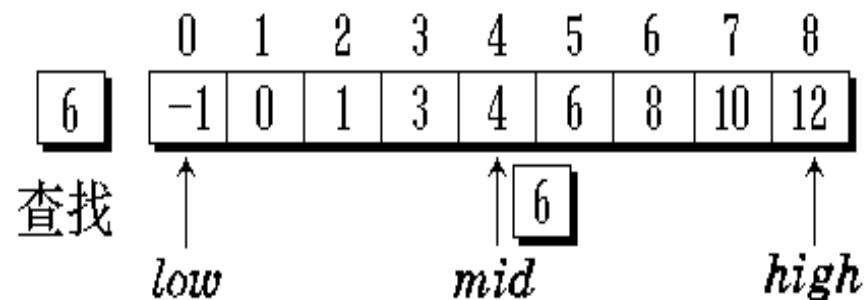
$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot (i + 1)$$

在等概率情形, $p_i = 1/n$, $i = 0, 1, \dots, n-1$ 。

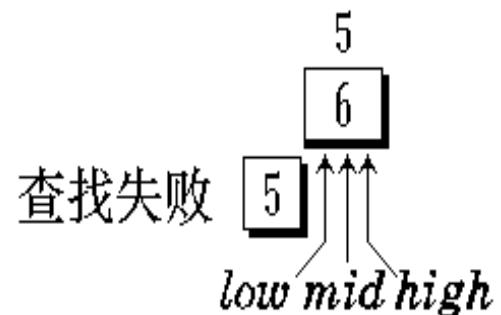
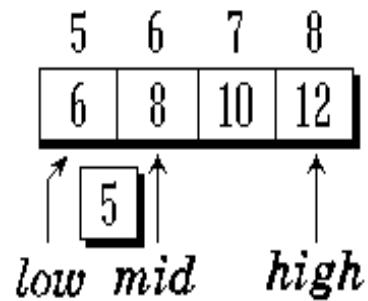
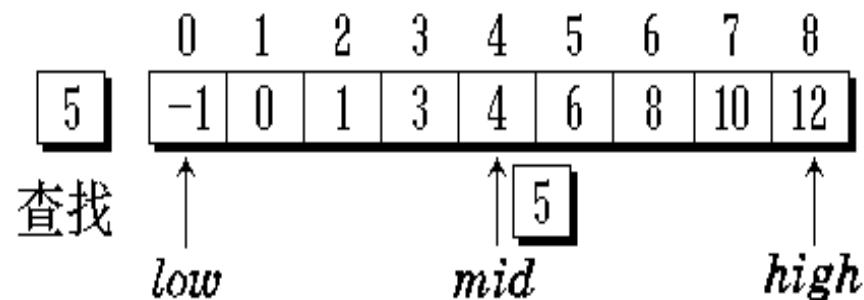
$$ASL_{succ} = \sum_{i=0}^{n-1} \frac{1}{n} (i + 1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

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。
- 采用对分搜索时，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：
 - ◆ $\text{Element}[mid].getKey() = x$ ，搜索成功；
 - ◆ $\text{Element}[mid].getKey() > x$ ，把搜索区间缩小到表的前半部分，再继续进行对分搜索；
 - ◆ $\text{Element}[mid].getKey() < x$ ，把搜索区间缩小到表的后半部分，再继续进行对分搜索。
- 每比较一次，搜索区间缩小一半。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。



搜索成功的例子



搜索失败的例子

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

```
template <class Type> int orderedList<Type> ::  
BinarySearch ( const Type & x, const int low,  
const int high ) const {
```

//折半搜索的递归算法

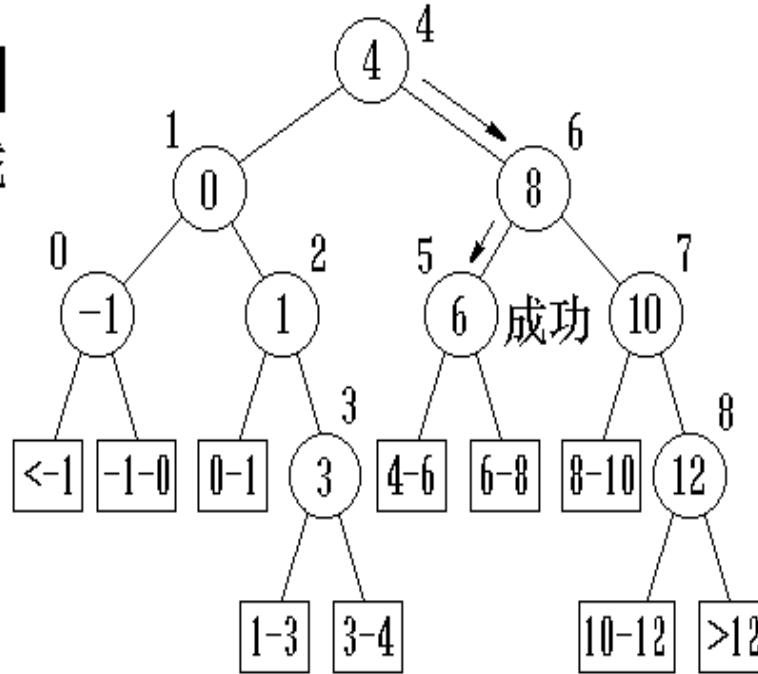
```
int mid = -1;  
if ( low <= high ) {  
    mid = ( low + high ) / 2;  
    if ( Element[mid].getKey( ) < x )  
        mid = BinarySearch ( x, mid + 1, high );  
    else if ( Element[mid].getKey( ) > x )  
        mid = BinarySearch ( x, low, mid - 1 );  
}  
return mid;
```

}

```
template <class Type> in orderedList <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].getKey ( ) < x )  
            low = mid + 1;  
        else if ( Element[mid].getKey ( ) > x )  
            high = mid - 1;  
        else return mid;  
    }  
    return -1;  
}
```

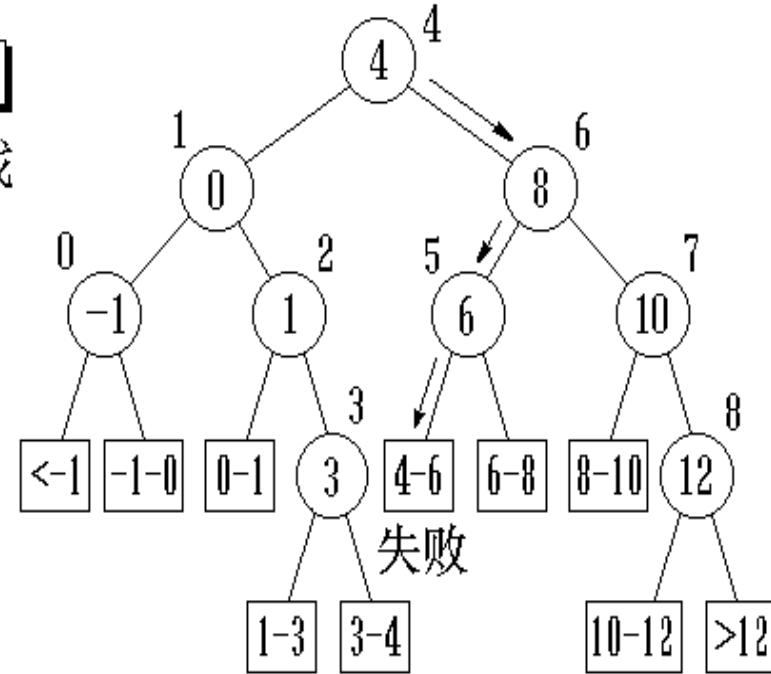
从有序表构造出的二叉搜索树(判定树)

6
查找



搜索成功的情形

5
查找



搜索不成功的情形

- 若设 $n = 2^h - 1$, 则描述对分搜索的二叉搜索树是高度为 $h-1$ 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 ($0 \leq i < h$) 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

$$+ 3 * \mathcal{S}_3 + \cdots + (n - 1) * \mathcal{S}_{n-1} + n * \mathcal{S}_1$$

$$ASL_{succ} = \sum_{i=1}^{n-1} B^i \cdot C^i = \frac{n}{1} \sum_{i=1}^{n-1} C^i = \frac{n}{1} (1 * 1 + 2 * \mathcal{S}_1 +$$

可以用归纳法证明

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

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

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

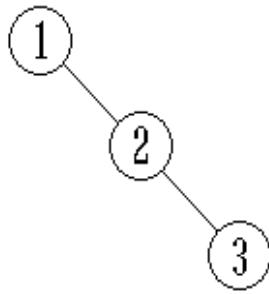


二叉搜索树 (Binary Search Tree)

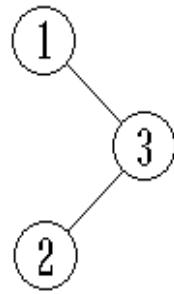
定义

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：

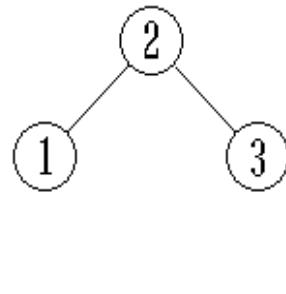
- 每个结点都有一个作为搜索依据的关键码(key)，所有结点的关键码互不相同。
- 左子树(如果存在)上所有结点的关键码都小于根结点的关键码。
- 右子树(如果存在)上所有结点的关键码都大于根结点的关键码。
- 左子树和右子树也是二叉搜索树。



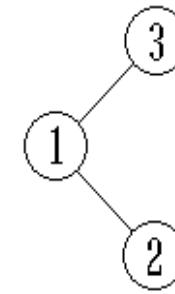
(a)



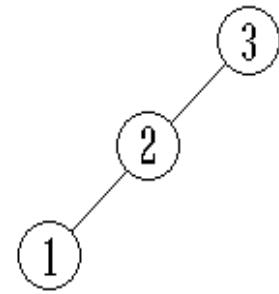
(b)



(c)



(d)



(e)

几个二叉搜索树的例子

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 { //二叉搜索树结点类  
friend class BST <Type>;  
public:  
    BstNode () :  
        leftChild (NULL), rightChild (NULL) { }  
    BstNode ( const Type d ) : data (d),  
        leftChild (NULL), rightChild (NULL) { }  
    BstNode ( const Type d = 0, BstNode *L = NULL,  
        BstNode *R = NULL) : data (d), leftChild (L),  
        rightChild (R) { }  
    ~BstNode () { }
```

protected:

Type *data*;

BstNode<Type> **leftChild*, **rightChild*;

};

template <class Type> class BST :

public : BinaryTree<Type> {

private:

BstNode<Type> **root*; //二叉搜索树的根指针

Type *RefValue*; //数据输入停止的标志

BstNode<Type> **lastfound*;

//最近搜索到的结点的指针

void MakeEmpty (BstNode<Type> *& *ptr*);

```
void Insert ( const Type & x,  
             BstNode<Type> *& ptr );      //插入  
void Remove ( const Type & x,  
              BstNode<Type> *& ptr );      //删除  
void PrintTree ( BstNode<Type> *ptr ) const;  
BstNode<Type> *Copy  
    (const BstNode<Type> *ptr );    //复制  
BstNode<Type> *Find (const Type & x,  
                     BstNode<Type> * ptr ) const;   //搜索  
BstNode<Type> *Min ( BstNode<Type> * ptr )  
    const;                          //求最小  
BstNode<Type> *Max ( BstNode<Type> * ptr )  
    const;                          //求最大
```

friend class *BSTIterator*<Type>; //中序游标类
public:

BST() : *root* (NULL) { }

BST (Type *value*) :

RefValue (*value*), *root* (NULL) { }

~BST();

const *BST* & operator = (const *BST* & *Value*);

void *MakeEmpty* ()

{ *MakeEmpty* (); *root* = NULL; }

void *PrintTree* () const { *PrintTree* (*root*); }

int *Find* (const Type & *x*)

const { return *Find* (*x*, *root*) != NULL; }

```
Type Min ( ) { return Min ( root )→data };  
Type Max ( ) { return Max ( root )→data };  
void Insert ( const Type & x )  
{ Insert ( x, root ); }  
void Remove ( const Type & x )  
{ Remove ( x, root ); }  
}
```

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。

template <class Type>

*BstNode<Type> * BST<Type> :: Find*

(const 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) //在左子树递归搜索

return Find (x, ptr->leftChild);

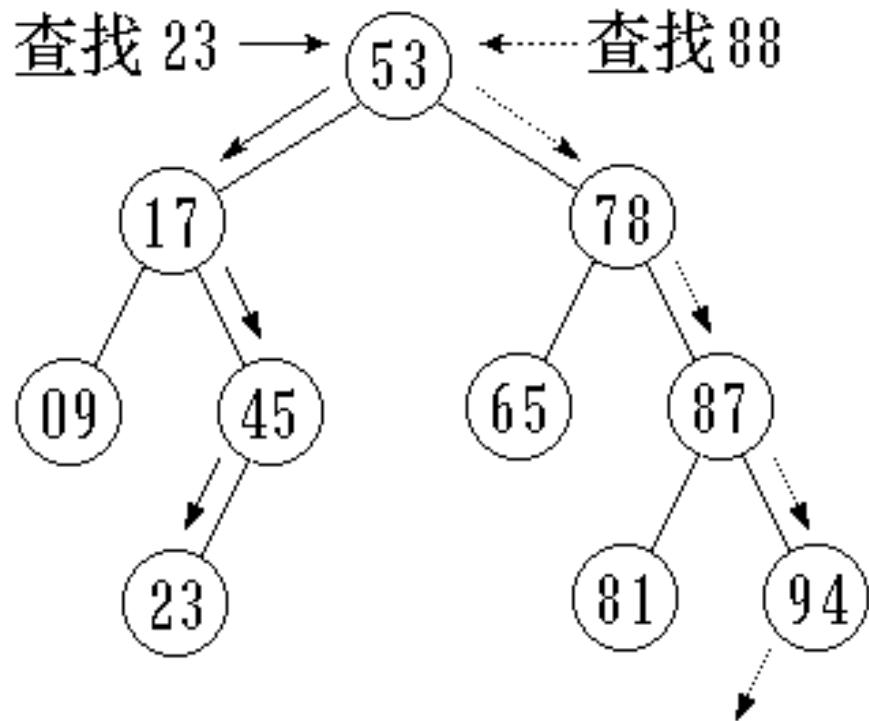
else if (x > ptr->data) //在右子树递归搜索

return Find (x, ptr->rightChild);

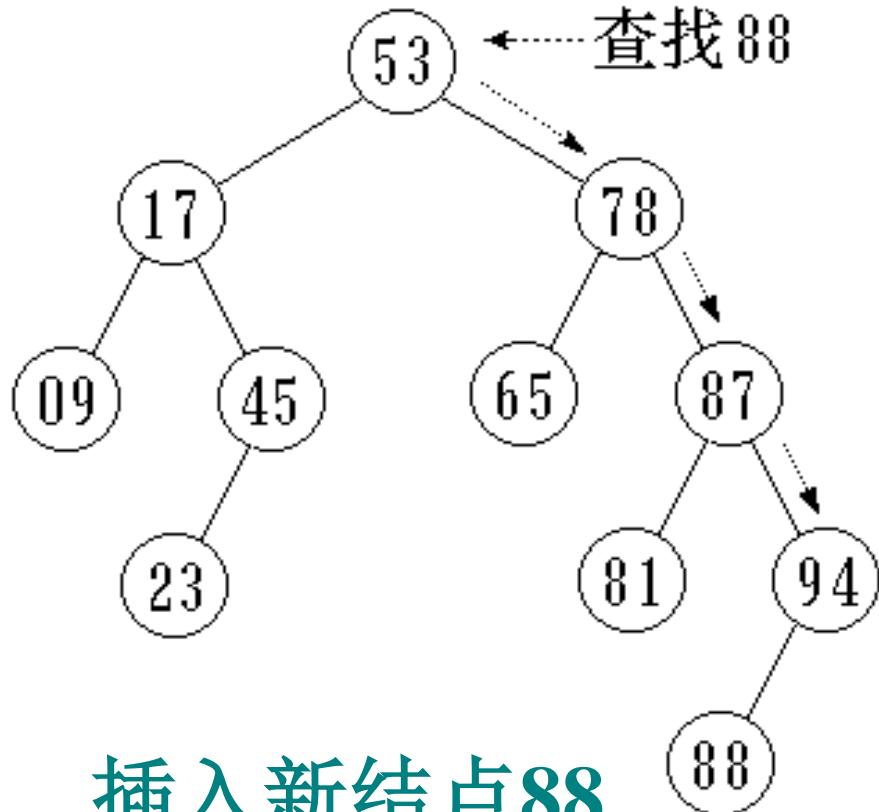
else return ptr; //相等,搜索成功

}

```
template <class Type>
BstNode <Type> * BST <Type> :: Find
    (const 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;
        while ( temp != NULL ) {
            if ( temp->data == x ) return temp; //成功
            if ( temp->data < x )
                temp = temp->rightChild; //右子树
            else temp = temp->leftChild; //左子树
        }
    }
    return NULL; //搜索失败
}
```



二叉搜索树的搜索



插入新结点88

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。

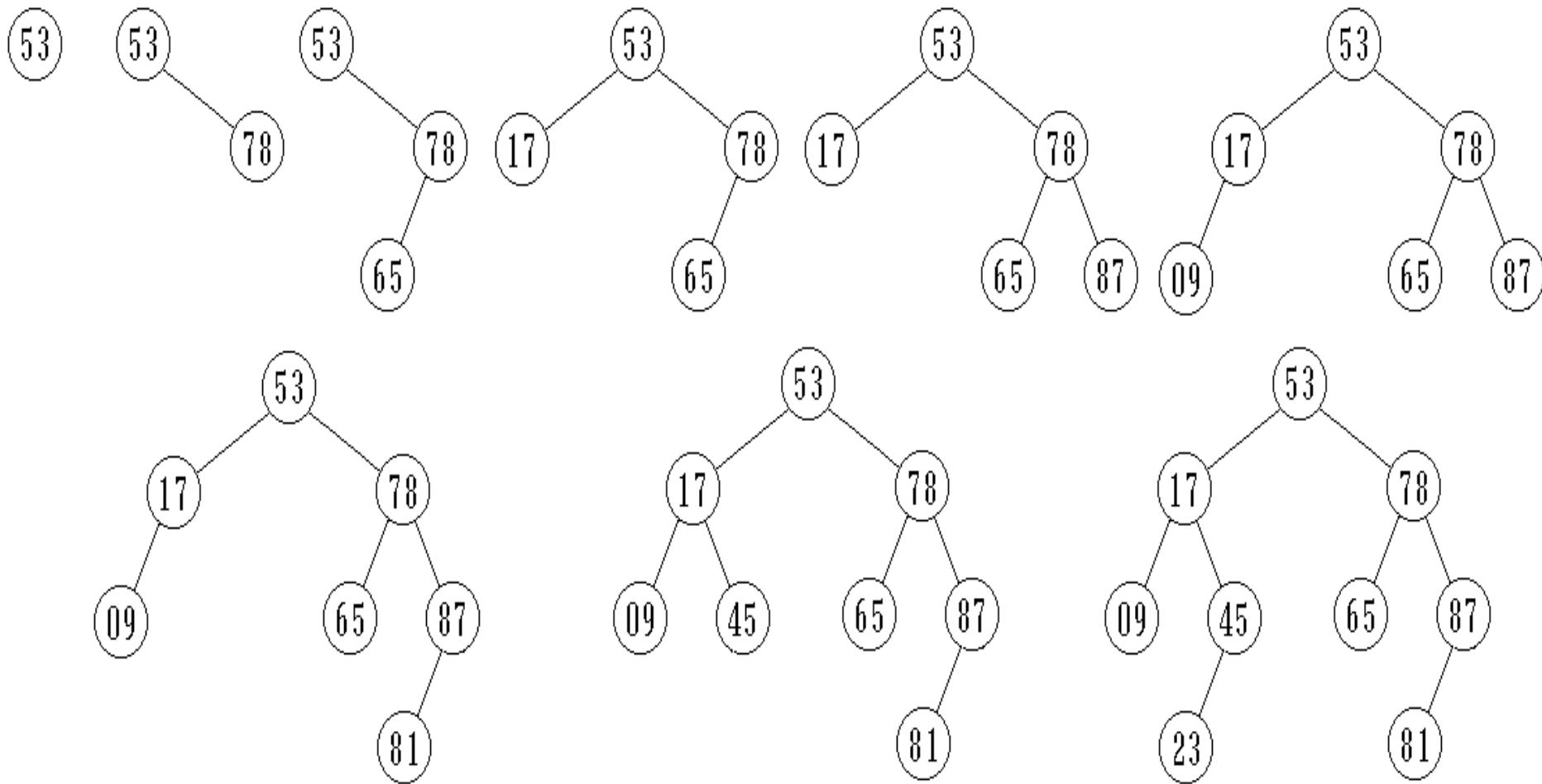
二叉搜索树的插入

- 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。

```
template <class Type> void BST<Type>::  
Insert (const Type & x, BstNode<Type> * & ptr) {  
//递归的二叉搜索树插入算法  
    if (ptr == NULL) { //空二叉树  
        ptr = new BstNode<Type> (x); //创建含 x 结点  
        if (ptr == NULL)  
            { cout << "Out of space" << endl; exit (1); }  
    }  
    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);  
}
```

输入数据，建立二叉搜索树的过程

输入数据序列 { 53, 78, 65, 17, 87, 09, 81, 45, 23 }



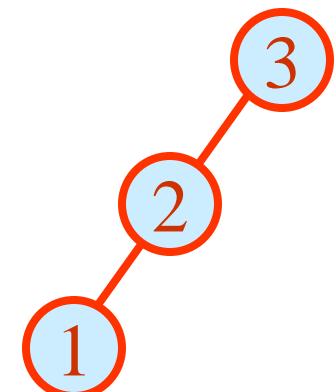
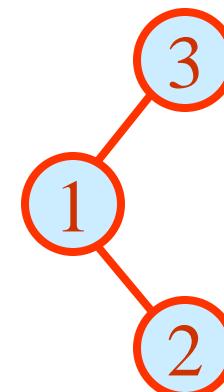
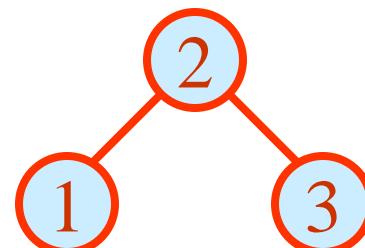
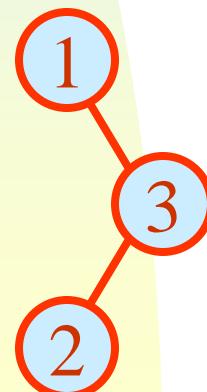
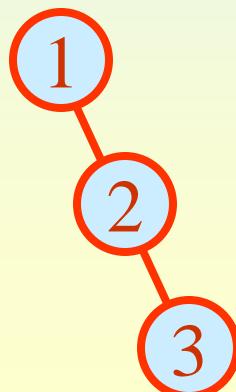
```
template <class Type>
BST<Type> :: BST( Type value ) {
//输入数据，建立二叉搜索树的算法, RefValue是
//输入结束标志
    Type &x;
    root = NULL; RefValue = value;
    cin >> x;
    while ( x.key != RefValue )
        { Insert ( x, root ); cin >> x; }
}
```

从空树开始建树，输入序列以输入一个结束标志 $value$ 结束。这个值应当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 $RefValue$ 为0或负数。

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。

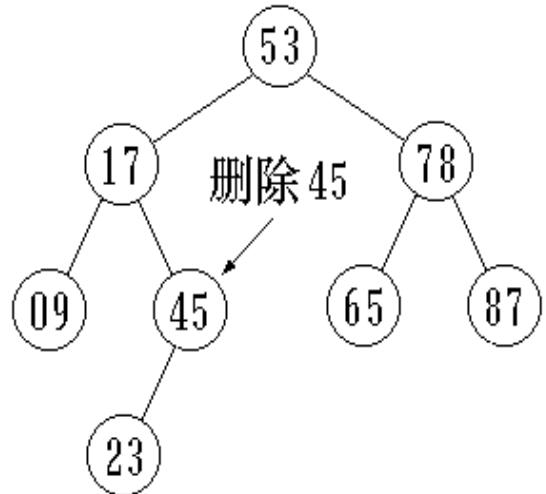
如果输入序列选得不好，会建立起一棵单支树，使得二叉搜索树的高度达到最大，这样必然会降低搜索性能。

$\{2, 1, 3\}$
 $\{1, 2, 3\}$ $\{1, 3, 2\}$ $\{2, 3, 1\}$ $\{3, 1, 2\}$ $\{3, 2, 1\}$

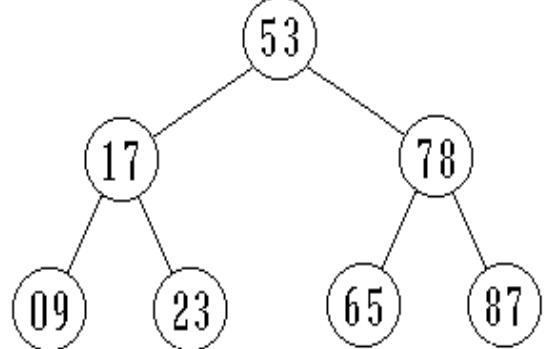


二叉搜索树的删除

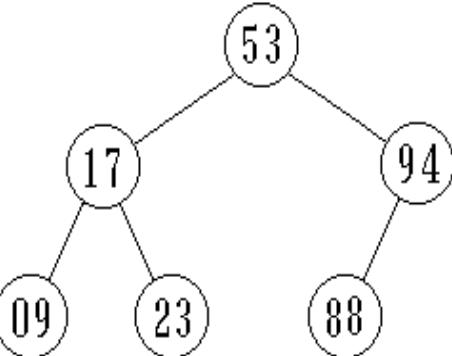
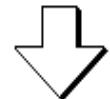
- 在二叉搜索树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉搜索树的性质不会失去。
- 为保证在执行删除后，树的搜索性能不至于降低，还需要防止重新链接后树的高度增加。
 - ◆ 删除叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。
 - ◆ 被删结点缺右子树，可以拿它的左子女结点顶替它的位置，再释放它。
 - ◆ 被删结点缺左子树，可以拿它的右子女结点顶替它的位置，再释放它。



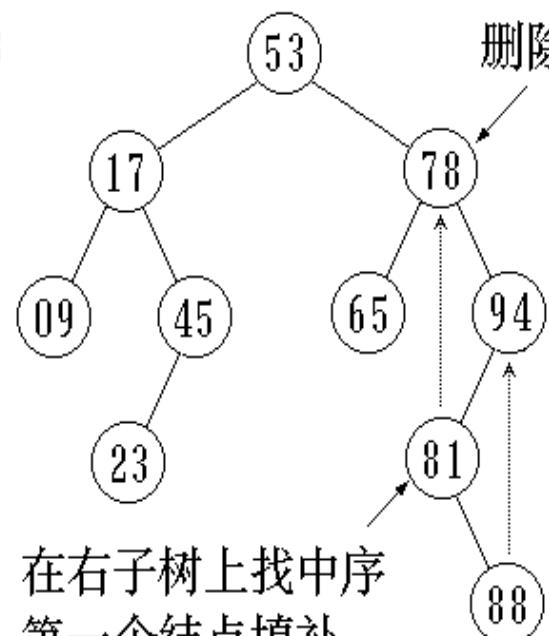
缺右子树用左子女填补



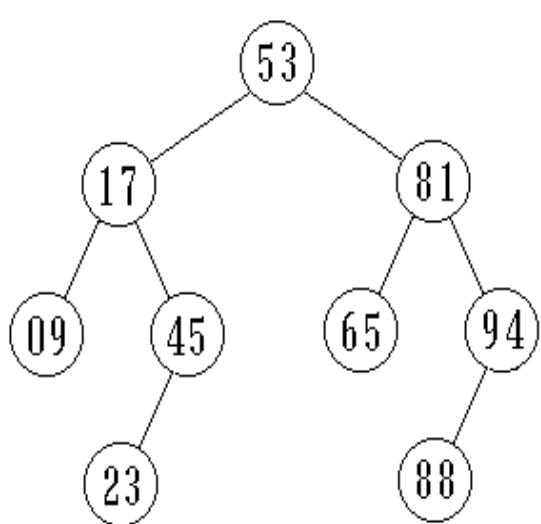
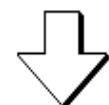
缺左子树用右子女填补



删除 78



在右子树上找中序
第一个结点填补



- ◆ 被删结点左、右子树都存在，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。

二叉搜索树的删除算法

```
template <class Type> void BST<Type> ::  
Remove (const Type &x, BstNode<Type> * &ptr) {  
    BstNode<Type> * temp;  
    if ( ptr != NULL )  
        if ( x < ptr->data ) Remove ( x, ptr->leftChild );  
        else if ( x > ptr->data )  
            Remove ( x, ptr->rightChild );
```

```
else if ( ptr→leftChild != NULL &&
          ptr→rightChild != NULL ) {
    temp = Min ( ptr→rightChild );
    ptr→data = temp→data;
    Remove ( ptr→data, temp );
}
else {
    temp = ptr;
    if ( ptr→leftChild == NULL )
        ptr = ptr→rightChild;
    else if ( ptr→rightChild == NULL )
        ptr = ptr→leftChild;
    delete temp;
}
```



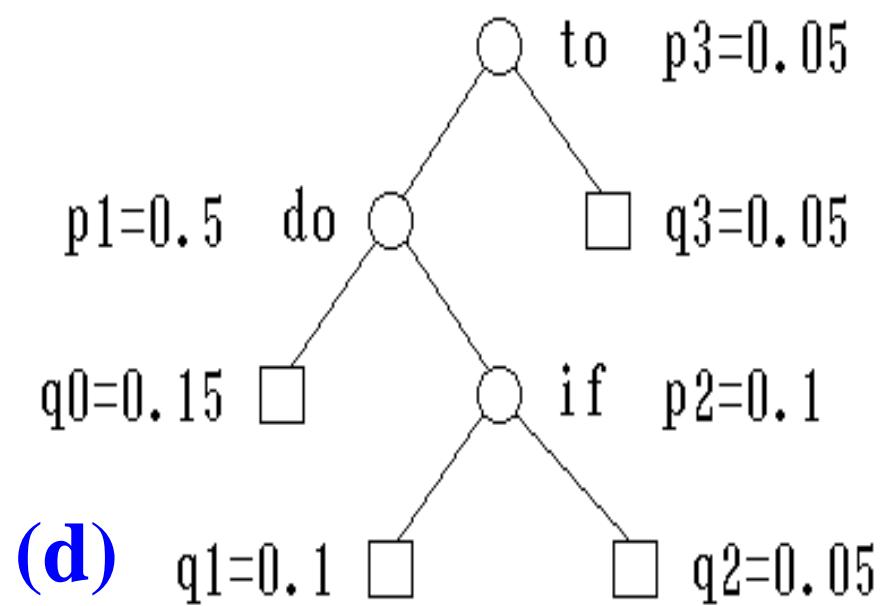
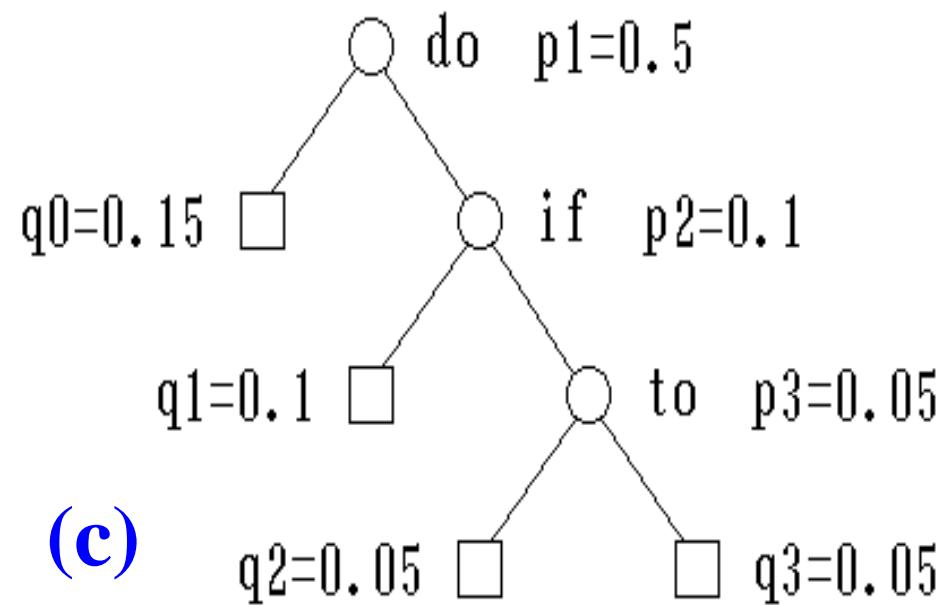
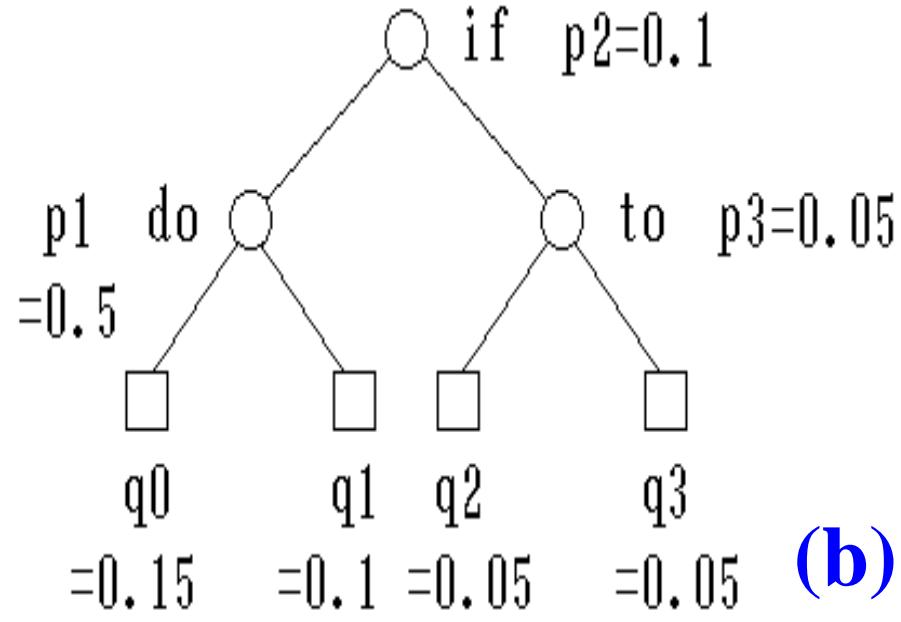
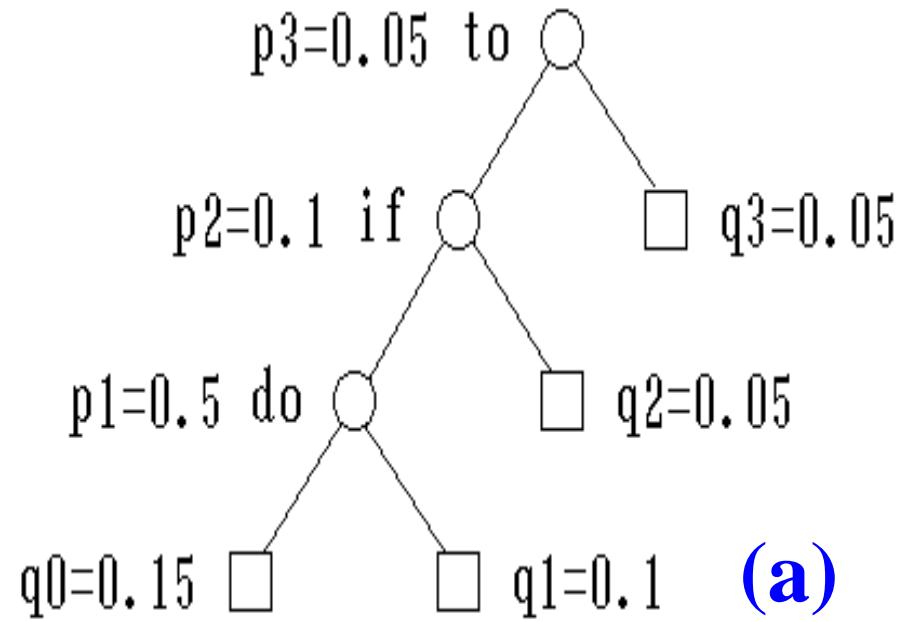
与二叉搜索树相关的中序游标类

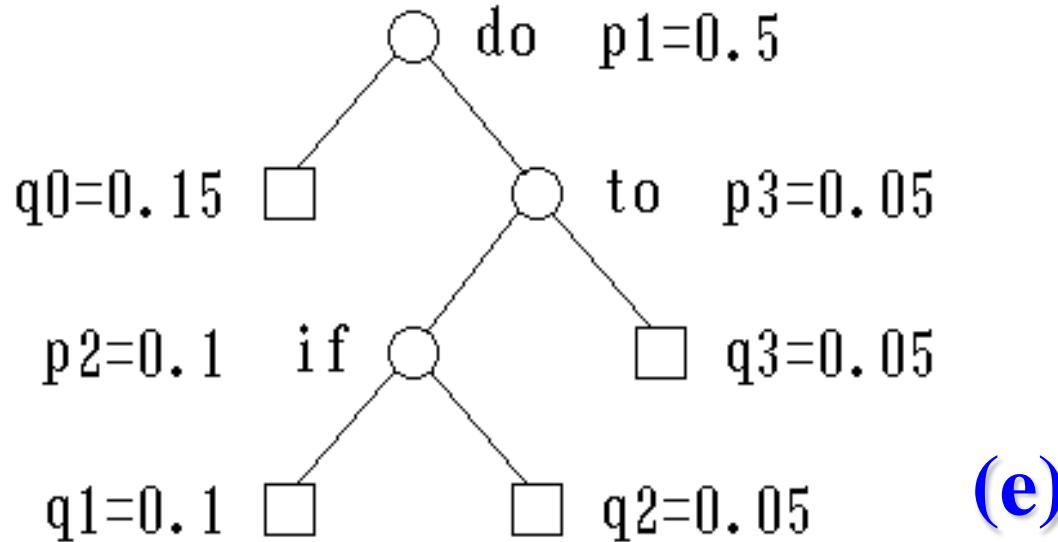
二叉搜索树中序游标类的类定义

```
template <class Type> class InorderIterator {  
public:  
    InorderIterator ( BST<Type> & Tree ) :  
        ref (Tree) { Init ( ); }  
    int Init ( );  
    int operator ! ( );  
    Type operator () ( );  
    int operator ++ ( );  
private:  
    BST<Type> & ref;      //二叉搜索树对象  
    Stack < BstNode<Type> * > itrStack; //迭代栈  
}
```

最优二叉搜索树

- 对于有 n 个关键码的集合，其关键码有 $n!$ 种不同的排列，可构成的不同二叉搜索树有
$$\frac{1}{n+1} C_{2n}^n \text{ (棵)}$$
- 如何评价这些二叉搜索树，可以用树的搜索效率来衡量。
- 例如，已知关键码集合 $\{ a1, a2, a3 \} = \{ \text{do}, \text{if}, \text{to} \}$ ，对应搜索概率为 $p1=0.5, p2=0.1, p3=0.05$ ，在各个搜索不成功的间隔内搜索概率又分别为 $q0=0.15, q1=0.1, q2=0.05, q3=0.05$ 。可能的二叉搜索树如下所示。





- 在扩充二叉搜索树中
 - ◆ ○表示内部结点，包含了关键码集合中的某一个关键码；
 - ◆ □表示外部结点，代表造成搜索失败的各关键码间隔中的不在关键码集合中的关键码。
- 在每两个外部结点之间必然存在一个内部结点。

设二叉树的内部结点有 n 个，外部结点有 $n+1$ 个。如果定义每个结点的路径长度为该结点的层次，并用 I 表示所有 n 个内部结点的路径长度之和，用 E 表示 $n+1$ 个外部结点的路径长度之和，可以用归纳法证明， $E = I + 2n$ 。

一棵扩充二叉搜索树的搜索成功的平均查找长度 ASL_{succ} 可以定义为该树所有内部结点上的权值 $p[i]$ 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：

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

扩充二叉搜索树搜索不成功的平均搜索长度

ASL_{unsucc} 为树中所有外部结点上权值 $q[j]$ 与到达外部结点所需关键码比较次数 $c'[j](=l'[j])$ 乘积之和：

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

扩充二叉搜索树总的平均搜索长度为：

$$ASL = ASL_{succ} + ASL_{unsucc}$$

所有内、外部结点上的权值关系为

$$\sum_{i=1}^n p[i] + \sum_{j=0}^n q[j] = 1$$

(1) 相等搜索概率的情形

若设树中所有内、外部结点的搜索概率都相等：

$$p[i] = q[j] = 1/7, \quad 1 \leq i \leq 3, \quad 0 \leq j \leq 3$$

图(a): $ASL_{succ} = 1/7 * 3 + 1/7 * 2 + 1/7 * 1 = 6/7,$

$$ASL_{unsucc} = 1/7 * 3 * 2 + 1/7 * 2 + 1/7 * 1 = 9/7.$$

总平均搜索长度 $ASL = 6/7 + 9/7 = 15/7.$

图(a): $ASL = 15/7$

图(d): $ASL = 15/7$

图(b): $ASL = 13/7$

图(e): $ASL = 15/7$

图(c): $ASL = 15/7$

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，因为所有内部结点、外部结点的搜索概率都相等，把它们的权值都视为1。同时，第 k 层有 2^k 个结点， $k = 0, 1, \dots$ 。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): $ASL_{succ} = 0.5*3 + 0.1*2 + 0.05*1 = 1.75,$
 $ASL_{unsucc} = 0.15*3 + 0.1*3 + 0.05*2 + 0.05*1 = 0.9.$
 $ASL = ASL_{succ} + ASL_{unsucc} = 2.65.$
图(a): $ASL = 2.65$ 图(d): $ASL = 2.15$
图(b): $ASL = 1.9$ 图(e): $ASL = 1.6$
图(c): $ASL = 0.85$
- 由此可知, 图(c)的情形下树的平均搜索长度达到最小, 因此, 图(c)的情形是最优二叉搜索树。

不相等搜索概率情形下的最优二叉搜索树可能不同于完全二叉树的形态。

考虑在不相等搜索概率情形下如何构造最优二叉搜索树。

假设关键码集合放在一个有序的顺序表中

{ $key[1], key[2], key[3], \dots key[n]$ }

设最优二叉搜索树为 $T[0][n]$, 其平均搜索长度为:

$$ASL = \sum_{i=1}^n p[i] * (l[i] + 1) + \sum_{j=0}^n q[j] * l[j] = C[0][n]$$

$l[i]$ 是内部结点 $a[i]$ 所在的层次号, $l[j]$ 是外部结点 j 所在的层次号。 $C[0][n]$ 是树的代价。

为计算方便, 将 $p[i]$ 与 $q[j]$ 化为整数。

构造最优二叉搜索树

- 要构造最优二叉搜索树，必须先构造它的左子树和右子树，它们也是最优二叉搜索树。对于任一棵子树 $T[i][j]$ ，它由

$\{ key[i+1], key[i+2], \dots, key[j] \}$

组成，其内部结点的权值为

$\{ p[i+1], p[i+2], \dots, p[j] \}$

外部结点的权值为

$\{ q[i], q[i+1], q[i+2], \dots, q[j] \}$

使用数组 $W[i][j]$ 来存放它的累计权值和：

$$W[i][j] = q[i] + p[i+1] + q[i+1] + p[i+2] + \\ + q[i+2] + \dots + p[j] + q[j]. \quad 0 \leq i \leq j \leq n$$

计算 $W[i][j]$ 可以用递推公式：

$$W[i][i] = q[i]$$

//不是二叉树，只有一个外部结点

$$\begin{aligned} W[i][i+1] &= q[i] + p[i+1] + q[i+1] \\ &= W[i][i] + p[i+1] + q[i+1] \end{aligned}$$

//有一个内部结点及两个外部结点的二叉树

$$W[i][i+2] = W[i][i+1] + p[i+2] + q[i+2]$$

//加一个内部结点和一个外部结点的二叉树

一般地，

$$W[i][j] = W[i][j-1] + p[j] + q[j]$$

//再加一个内部结点和一个外部结点

对于一棵包括关键码

{ $\underline{key[i+1]}, \dots, \underline{key[k-1]}, key[k], \underline{key[k+1]}, \dots, \underline{key[j]}$ }

的子树 $T[i][j]$, 若设它的根结点为 k , $i < k \leq j$,
它的代价为:

$$C[i][j] = p[k] + C[i][k-1] + W[i][k-1] + C[k][j] + W[k][j]$$

- $C[i][k-1]$ 是其包含关键码 { $\underline{key[i+1]}, key[i+2], \dots, \underline{key[k-1]}$ } 的左子树 $T[i][k-1]$ 的代价
- $C[i][k-1] + W[i][k-1]$ 等于把左子树每个结点的路径长度加1而计算出的代价
- $C[k][j]$ 是包含关键码 { $\underline{key[k+1]}, key[k+2], \dots, \underline{key[j]}$ } 的右子树 $T[k][j]$ 的代价
- $C[k][j] + W[k][j]$ 是把右子树每个结点的路径长度加1之后计算出的代价。

因此，公式

$$C[i][j] = p[k] + C[i][k-1] + W[i][k-1] + C[k][j] + W[k][j]$$

表明：整个树的代价等于其左子树的代价加上其右子树的代价，再加上根的权值。

因为 $W[i][j] = p[k] + W[i][k-1] + W[k][j]$ ，
故有

$$C[i][j] = W[i][j] + C[i][k-1] + C[k][j]$$

可用 $k = i+1, i+2, \dots, j$ 分别计算上式，选取使得
 $C[i][j]$ 达到最小的 k 。这样可将最优二叉搜索树
 $T[i][j]$ 构造出来。

构造最优二叉搜索树的方法就是自底向上逐步
构造的方法。

构造的步骤

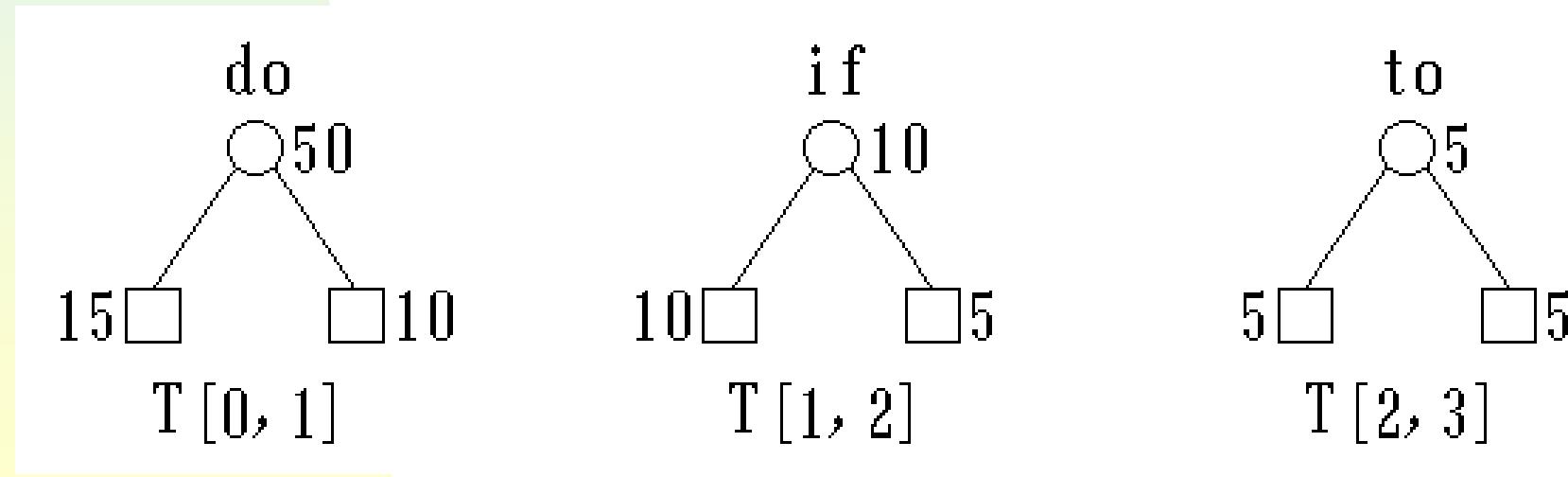
- 第一步，构造只有一个内部结点的最优搜索树： $T[0][1], T[1][2], \dots, T[n-1][n]$ 。
在 $T[i-1][i]$ ($1 \leq i \leq n$) 中只包含关键码 $key[i]$ 。其代价分别是 $C[i-1][i] = W[i-1][i]$ 。另外设立一个数组 $R[0..n][0..n]$ 存放各最优二叉搜索树的根。 $R[0][1] = 1, R[1][2] = 2, \dots, R[n-1][n] = n$ 。
- 第二步，构造有两个内部结点的最优二叉搜索树： $T[0][2], T[1][3], \dots, T[n-2][n]$ 。在 $T[i-2][i]$ 中包含两个关键码 $\{key[i-1], key[i]\}$ 。其代价取分别以 $key[i-1], key[i]$ 做根时计算出的 $C[i-2][i]$ 中的小者。

- 第三步，第四步，...，构造有 3 个内部结点，有 4 个内部结点，... 的最优二叉搜索树。
- 最后构造出包含有 n 个内部结点的最优二叉搜索树。对于这样的最优二叉搜索树，若设根为 k ，则根 k 的值存于 $R[0][n]$ 中，其代价存于 $C[0][n]$ 中，左子树的根存于 $R[0][k-1]$ 中，右子树的根存于 $R[k][n]$ 中。

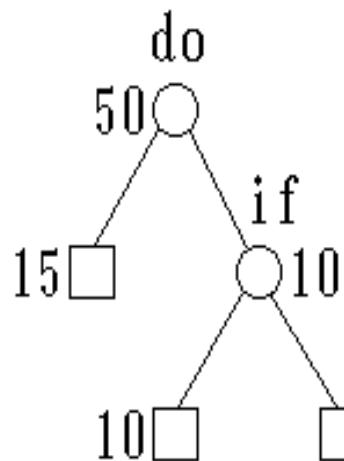
例：给出关键码集合和内、外部结点的权值序列

关键码集合	<i>key1</i>	<i>key2</i>	<i>key3</i>
实例	do	if	to
对应	内部结点 p1=50 p2=10 p3=5		
权值	外部结点 q0=15 q1=10 q2=5 q3=5		

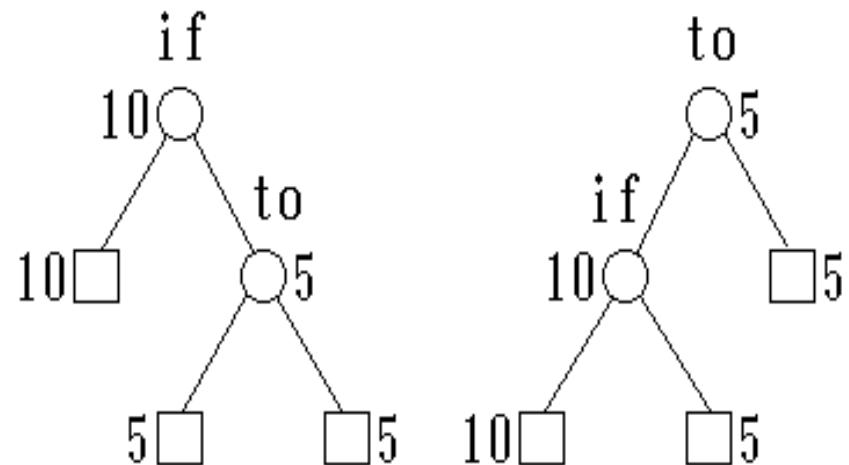
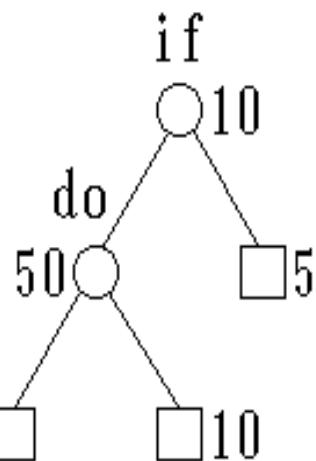
第一步



第二步



$T[0, 2]$



$T[1, 3]$

$C \quad 115$

$W \quad 90$

$R \quad 1$

左子树 $T[0,0]$

右子树 $T[1,2]$

165

90

2

$T[0,1]$

$T[2,2]$

50

35

2

$T[1,1]$

$T[2,3]$

60

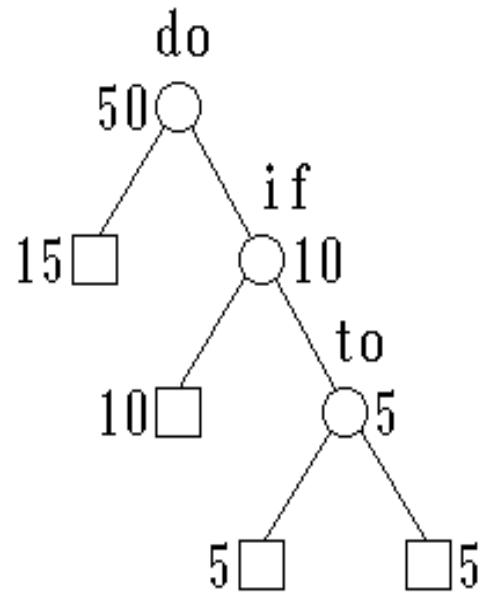
35

3

$T[1,2]$

$T[3,3]$

第三步



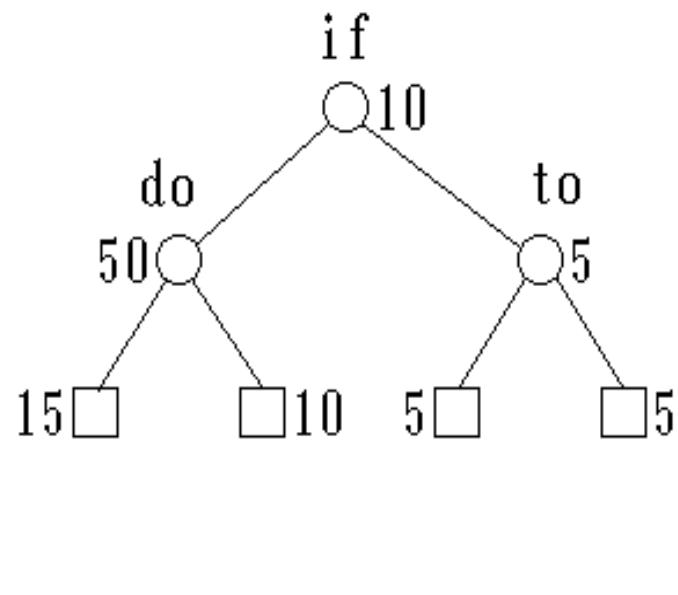
$T [0, 3]$

$C \quad 150$

$W \quad 100$

$R \quad 1$

左子树 $T[0,0]$
右子树 $T[1,3]$



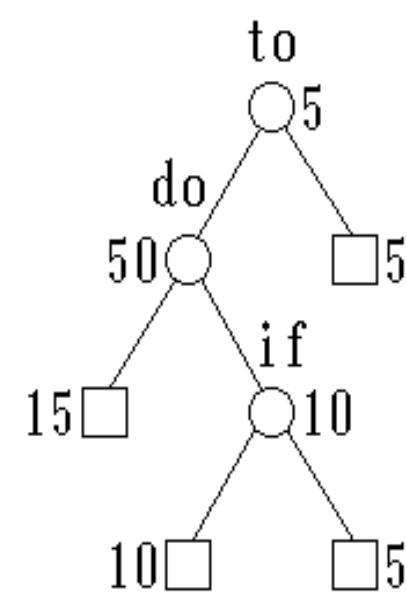
$T [0, 3]$

190

100

2

$T[0,1]$
 $T[2,3]$



$T [0, 3]$

215

100

3

$T[0,2]$
 $T[3,3]$

	0	1	2	3
0	15	75	90	100
1		10	25	35
2			5	15
3				5

$W[i][j]$

	0	1	2	3
0	0	75	115	150
1		0	25	50
2			0	15
3				0

$C[i][j]$

	0	1	2	3
0	0	1	1	1
1		0	2	2
2			0	3
3				0

$R[i][j]$

3个关键码 { do, if, to } 的
最优二叉搜索树

$$p_1=50, p_2=10, p_3=5$$

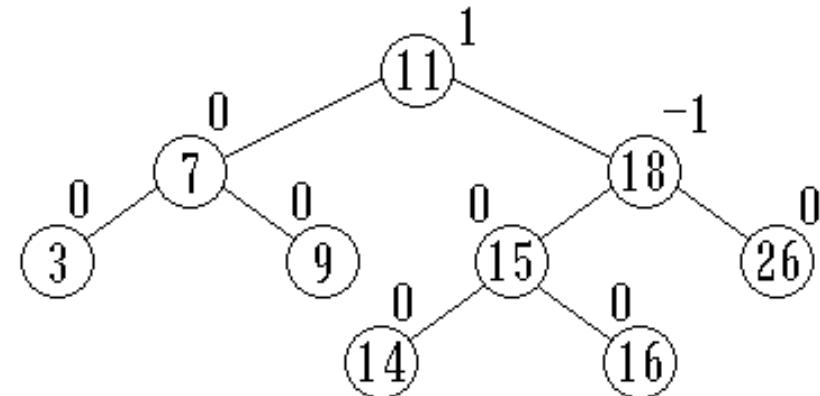
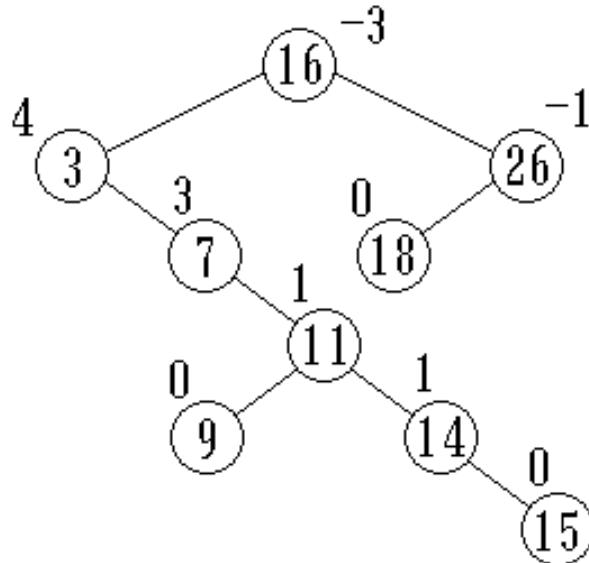
$$q_0=15, q_1=10, q_2=5, q_3=5$$



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。



高度不平衡的二叉搜索树

高度平衡的二叉搜索树

结点的平衡因子*balance* (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*。
- 根据AVL树的定义，任一结点的平衡因子只能取 -1， 0和 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，它就成为AVL树。如果它有 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 {  
        Type data;  
        AVLNode *left, *right;  
        int balance;  
        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:

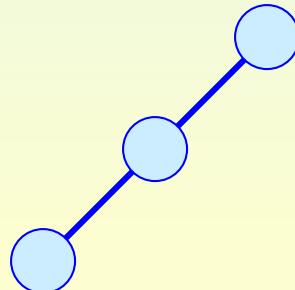
```
Type RefValue;  
AVLNode *root;  
int Insert ( AVLNode<Type>*&tree, Type x,  
             int &taller );  
void RotateLeft ( AVLNode<Type> *Tree,  
                   AVLNode<Type> *&NewTree );  
void RotateRight ( AVLNode<Type> *Tree,  
                    AVLNode<Type> *&NewTree );  
void LeftBalance ( AVLNode<Type> *&Tree,  
                     int &taller );  
void RightBalance ( AVLNode<Type> *&Tree,  
                     int &taller );
```

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

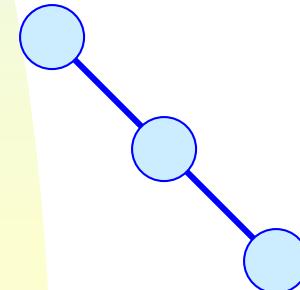
平衡化旋转

- 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(左、右子树的高度差)。
- 如果在一结点发现高度不平衡，停止回溯。

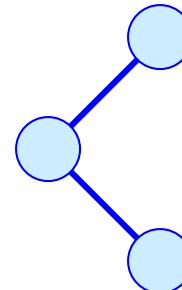
- 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。 单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。 双旋转分为先左后右和先右后左两类。



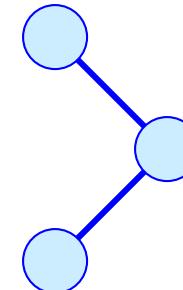
右单旋转



左单旋转

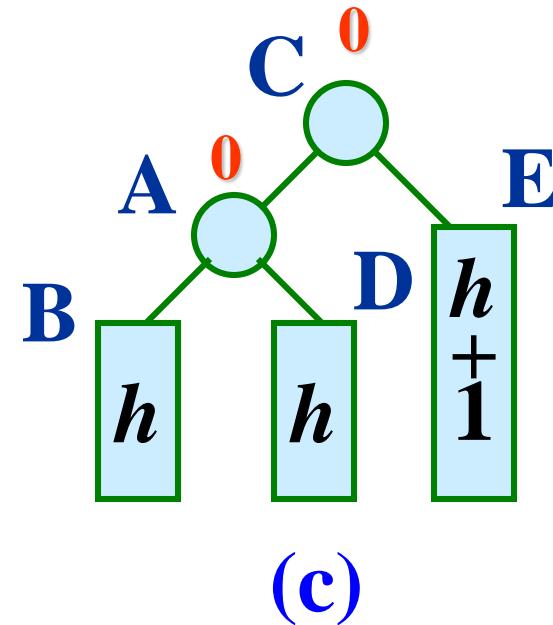
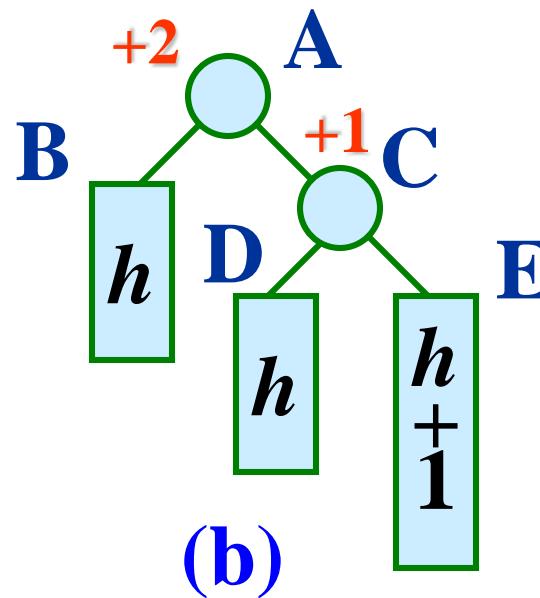
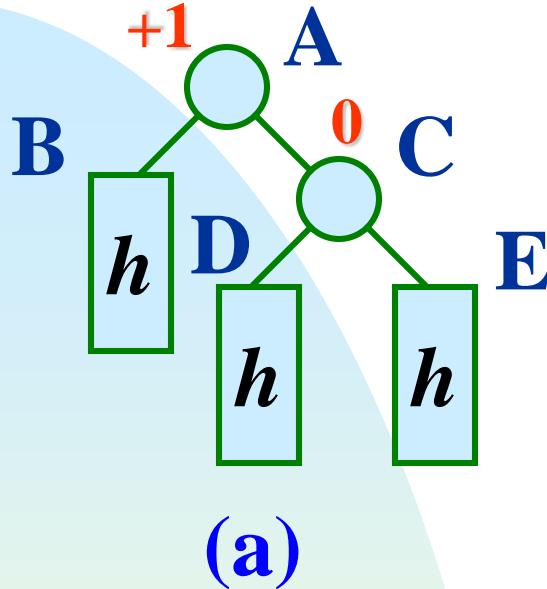


左右双旋转



右左双旋转

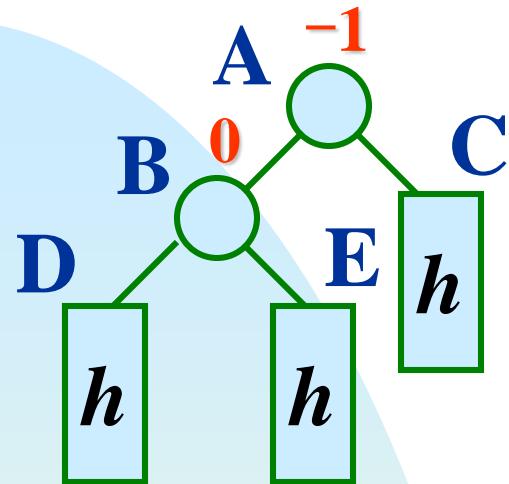
左单旋转 (RotateLeft)



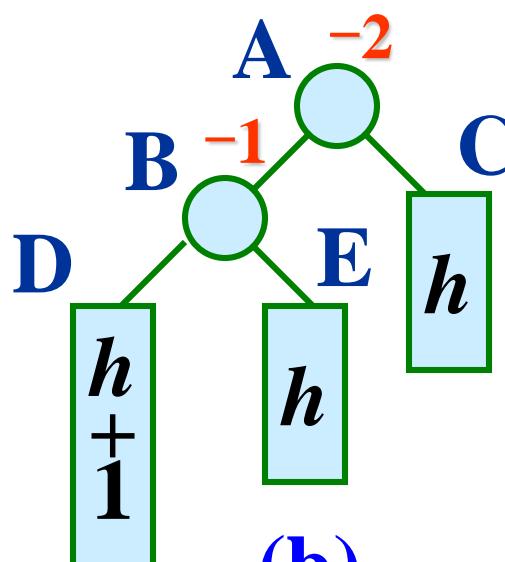
- 如果在子树E中插入一个新结点，该子树高度增1导致结点A的平衡因子变成+2，出现不平衡。
- 沿插入路径检查三个结点A、C和E。它们处于一条方向为“\”的直线上，需要做左单旋转。
- 以结点C为旋转轴，让结点A反时针旋转。

```
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;  
}
```

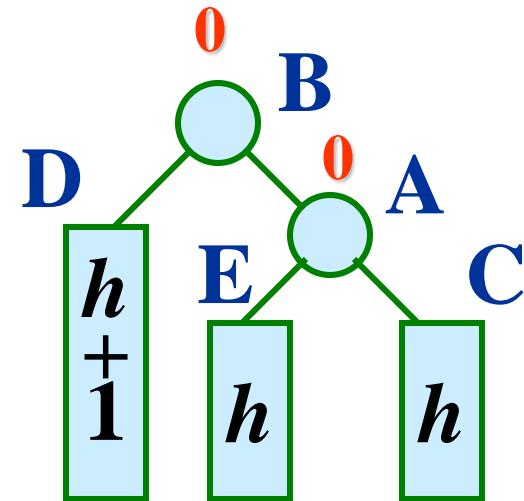
右单旋转 (RotateRight)



(a)



(b)



(c)

- 在左子树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，造成了不平衡。
- 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，它们处于一条方向为“/”的直线上，需要做右单旋转。
- 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;
```

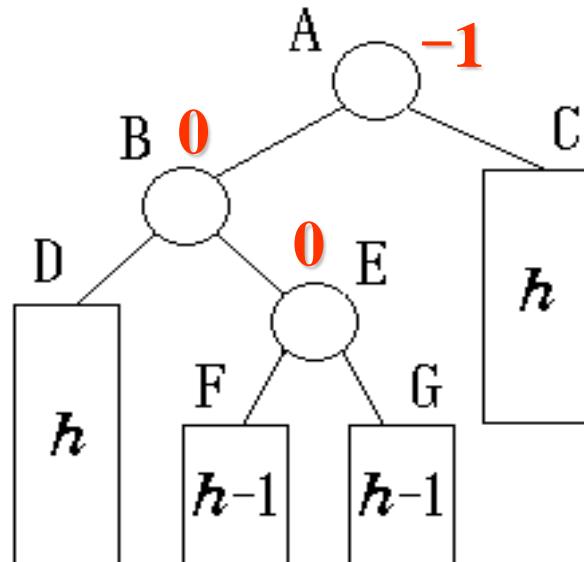
```
NewTree->right = Tree;
```

```
}
```

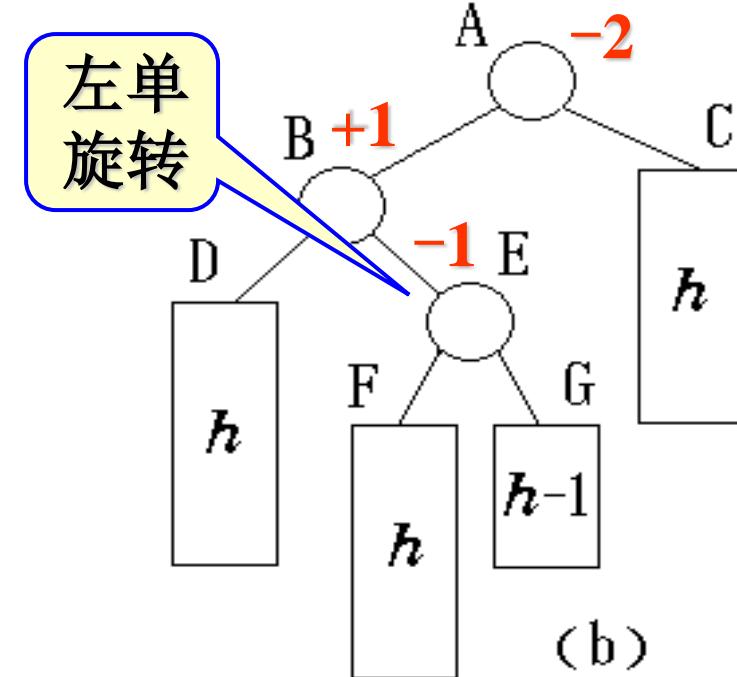
先左后右双旋转 (RotationLeftRight)

- 在子树F或G中插入新结点，该子树的高度增1。结点A的平衡因子变为 -2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、B和E，它们位于一条形如“<”的折线上，因此需要进行先左后右的双旋转。
- 首先以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置，做左单旋转。
- 再以结点E为旋转轴，将结点A顺时针旋转，做右单旋转。使之平衡化。

插入

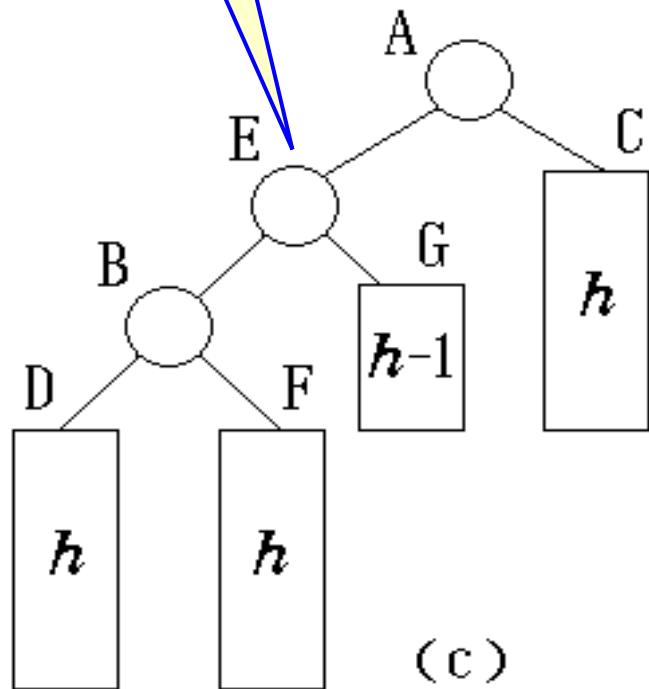


右单旋转

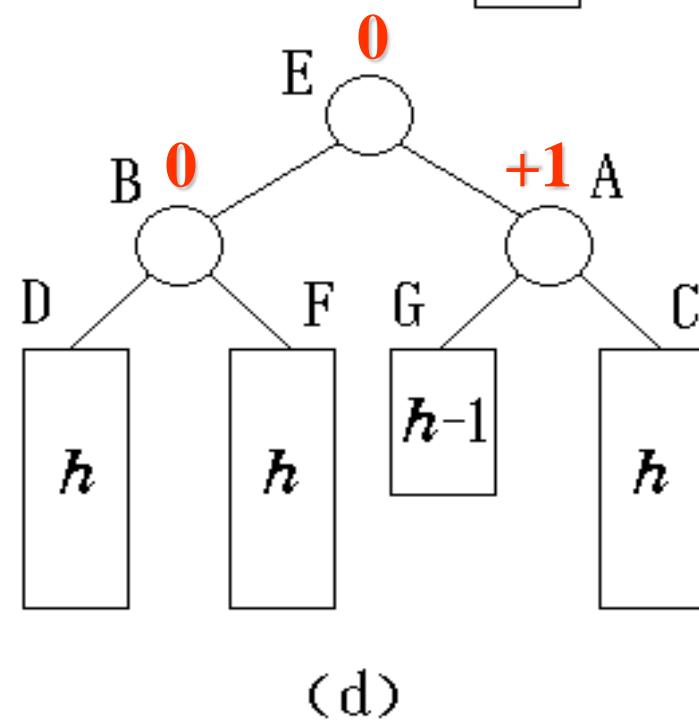


(a)

(b)



(c)



(d)

```
template <class Type> void AVLTree<Type>::  
LeftBalance ( AVLNode<Type> * &Tree,  
    int & taller ) {
```

//左平衡化的算法

```
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;
```

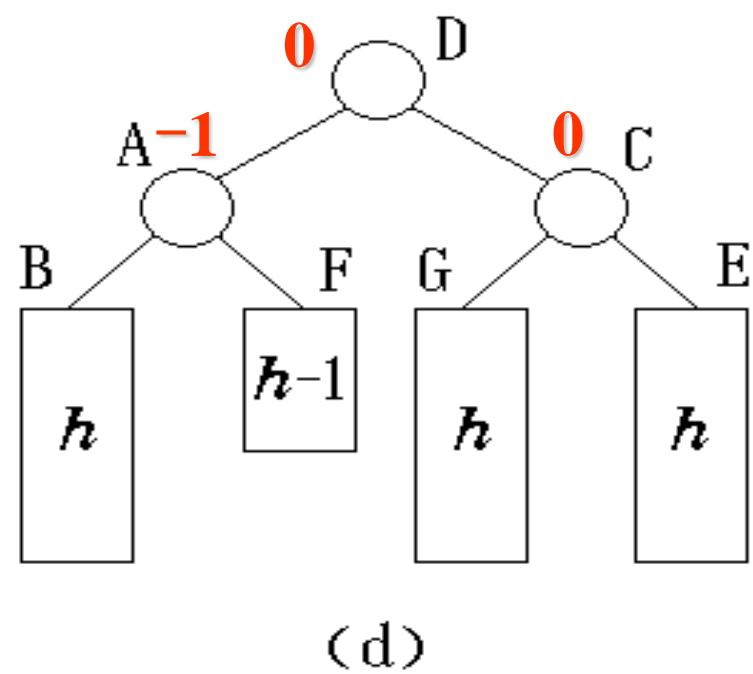
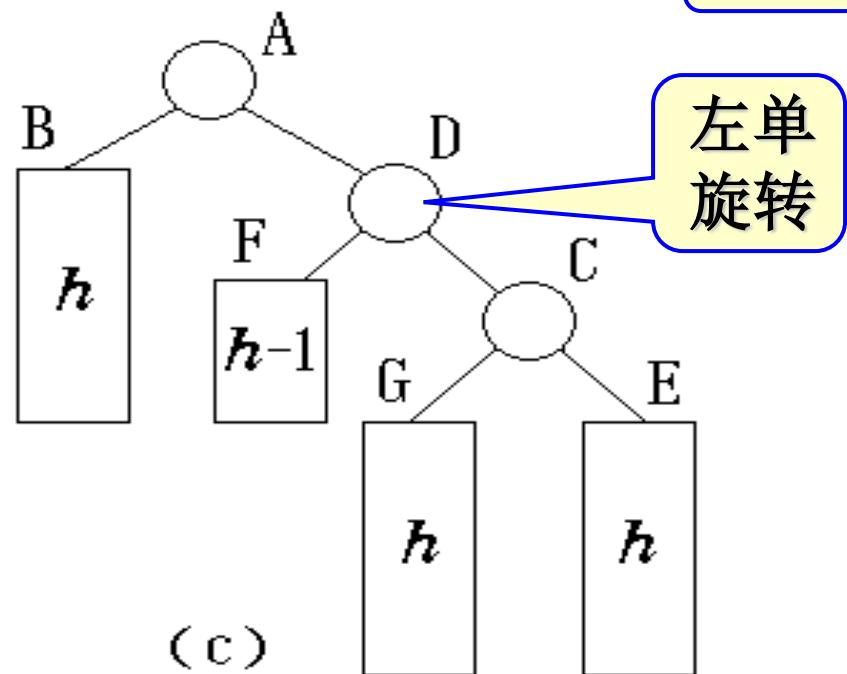
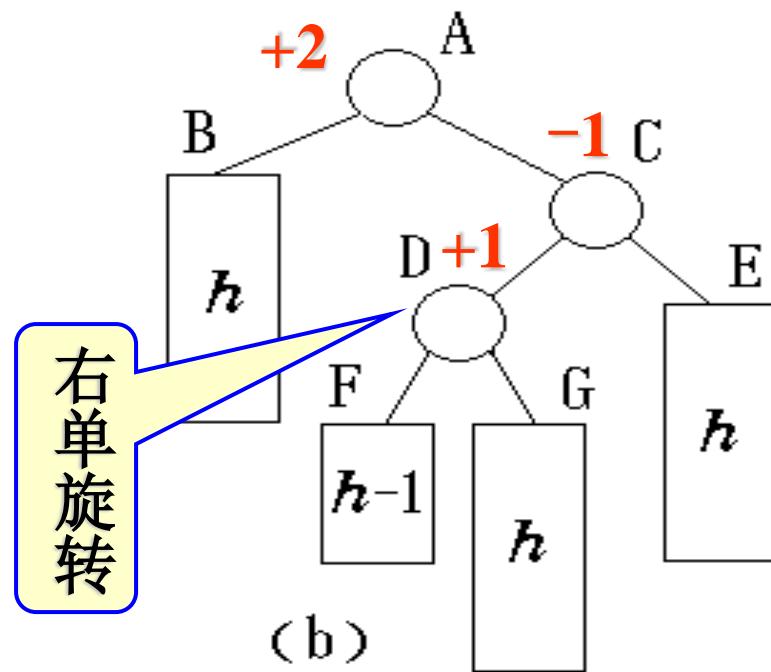
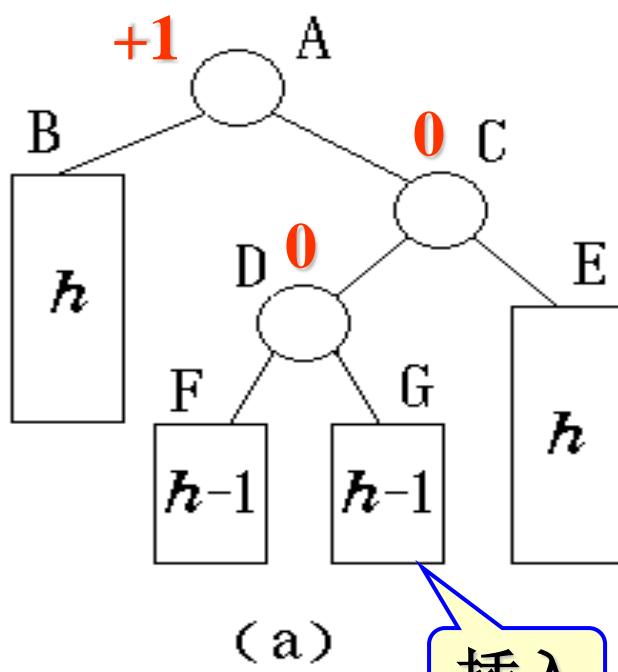
case 1 :

```
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;  
        break;  
}
```

```
rightsub→balance = 0;  
RotateLeft ( leftsub, Tree→left );  
RotateRight ( Tree, Tree );  
taller = 0;  
}  
}
```

先右后左双旋转 (RotationRightLeft)

- 右左双旋转是左右双旋转的镜像。
- 在子树F或G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。
- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“>”的折线上，需要进行先右后左的双旋转。
- 首先做右单旋转：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。
- 再做左单旋转：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 ) {
```

//右平衡化的算法

```
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;
```

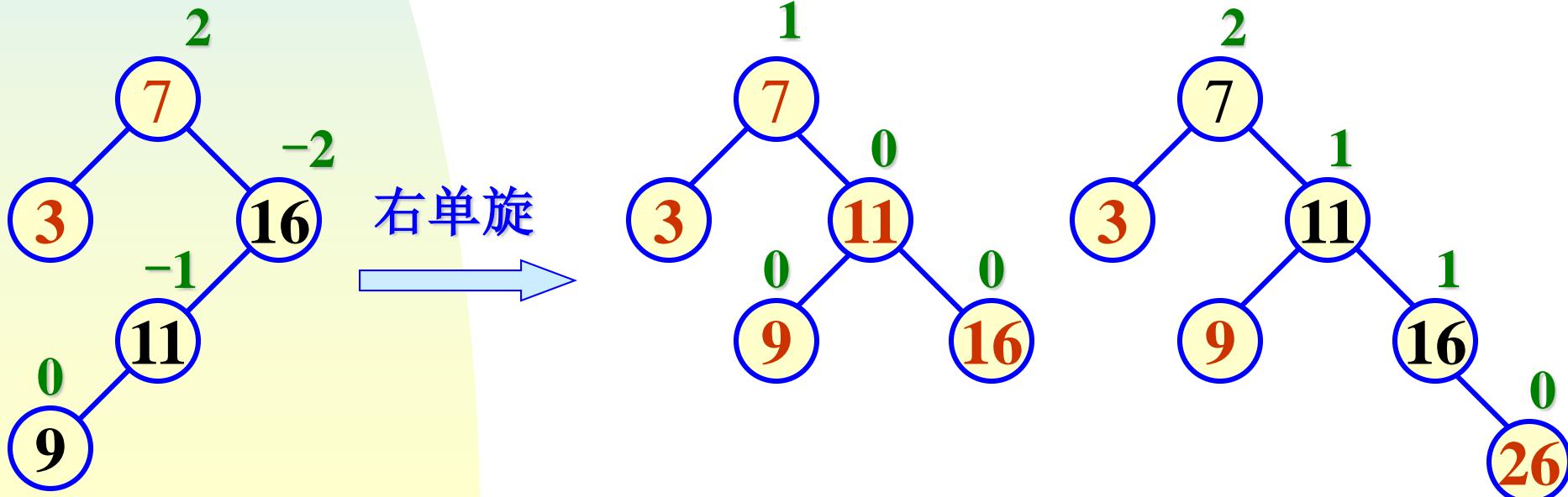
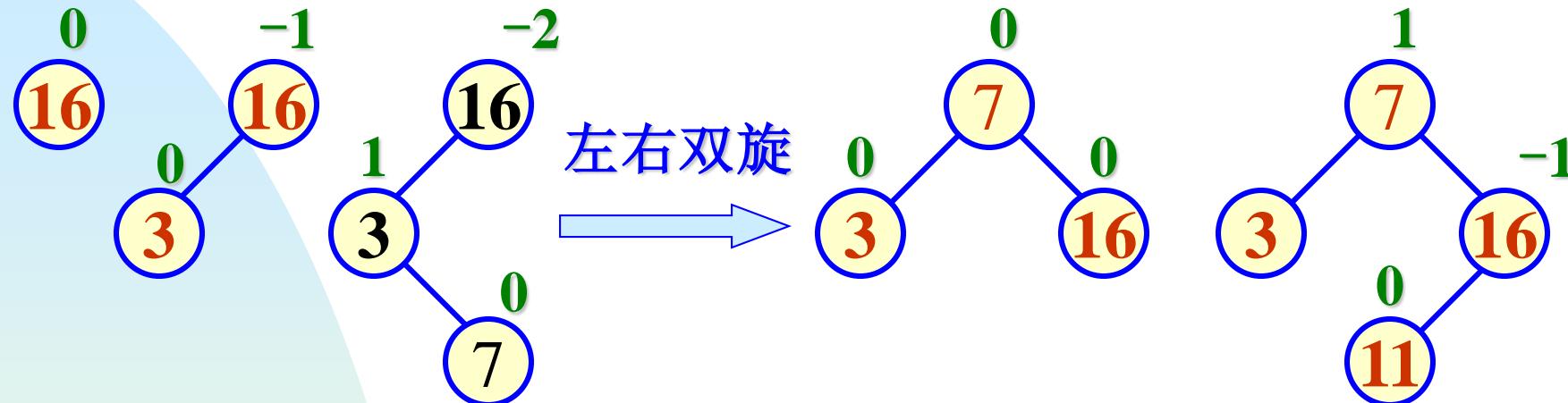
```
case -1 : leftsub = rightsub->left;
```

```
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;  
}  
}
```

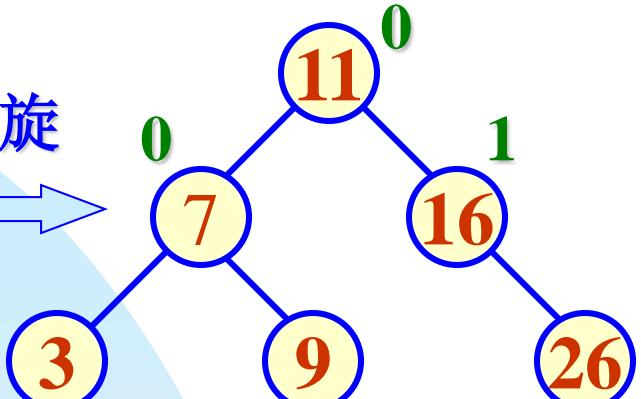
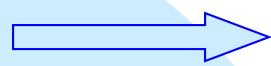
AVL树的插入

- 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的关键码，逐步建立AVL树。在插入新结点时使用了前面所给的算法进行平衡旋转。

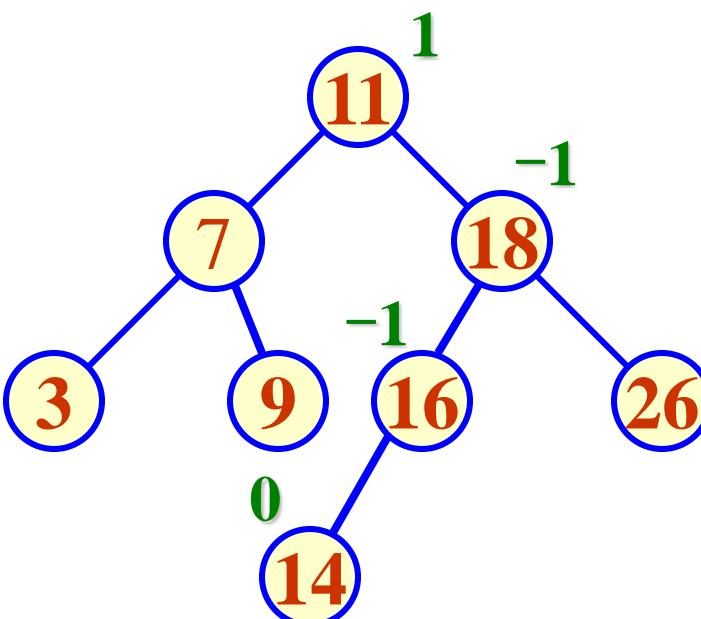
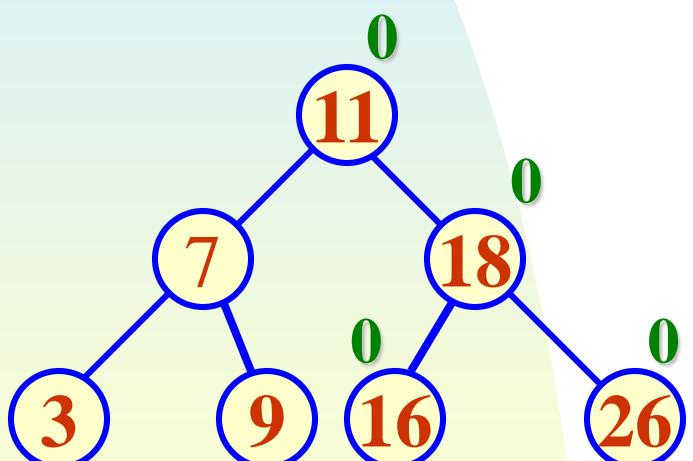
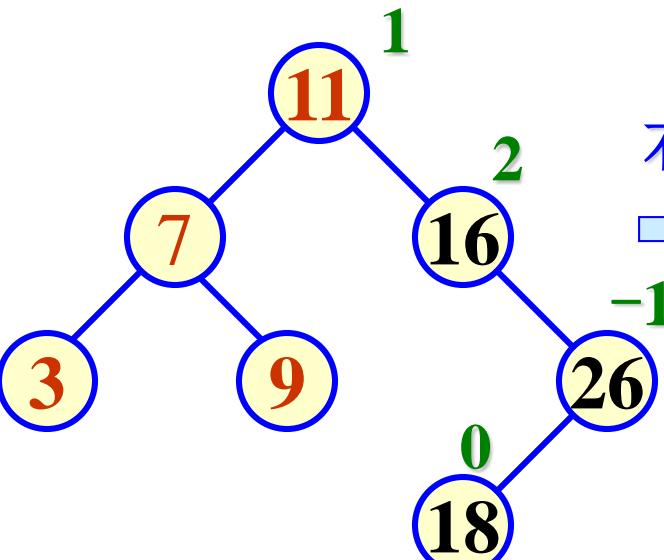
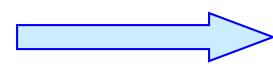
例，输入关键码序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }, 插入和调整过程如下。

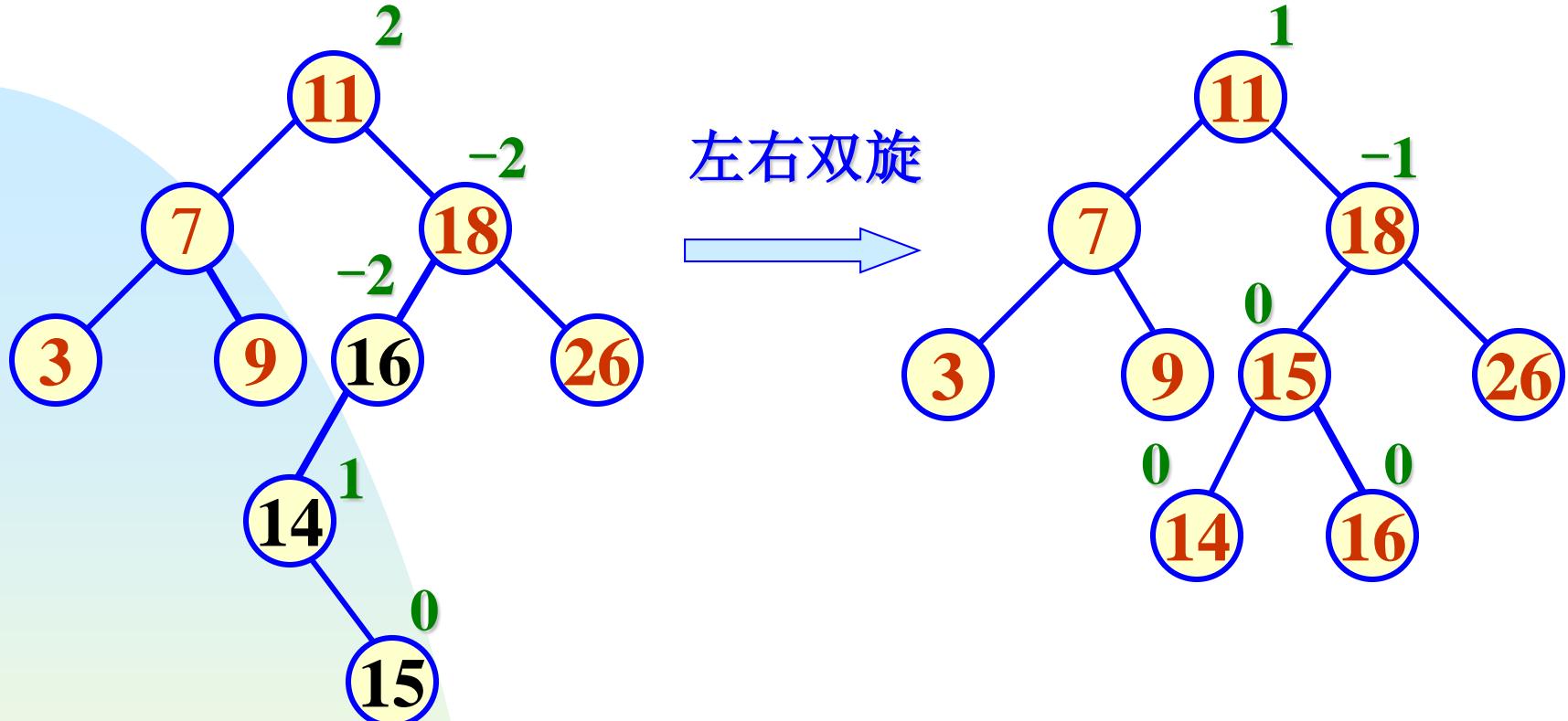


左单旋



右左双旋





从空树开始的建树过程

- 下面的算法将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> int AVLTree<Type> ::  
Insert ( AVLNode<Type>*&tree, Type x,  
        int &taller ) {
```

//AVL树的插入算法

```
int success;
```

```
if ( tree == NULL ) {
```

```
    tree = new AVLNode ( x );
```

```
    success = tree != NULL ? 1 : 0;
```

```
    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;  
        break;  
}  
}  
  
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 );  
          break;  
      }  
  }  
return success;  
}
```

AVL树的重载操作 >>、<< 和遍历算法的实现

```
template <class Type>
istream & operator >> ( istream & in,
    AVLTree<Type> & Tree ) {
    Type item;
    cout << "构造AVL树 :\n";
    cout << "输入数据 ( 以 " << Tree.RefValue
        << " 结束 ) : ";
    in >> item;
    while ( item != Tree.RefValue ) {
        Tree.Insert ( item );
        cout << "输入数据 ( 以 " << Tree.RefValue
            << " 结束 ) : ";
```

```
    in >> item;  
}  
return in;  
}  
  
template <class Type>  
ostream & operator << ( ostream & out,  
    const AVLTree<Type> & Tree ) {  
    out << “AVL树的中序遍历.\n”;  
    Tree.Traverse ( Tree.root, out );  
    out << endl;  
return out;  
}
```

```
template <class Type>
void AVLTree <Type> :: Traverse
( AVLNode *ptr, ostream & out ) const {
//AVL树中序遍历并输出数据
    if ( ptr != NULL ) {
        Traverse ( ptr→left, out );
        out << ptr→data << ' ';
        Traverse ( ptr→right, out );
    }
}
```

AVL树的删除

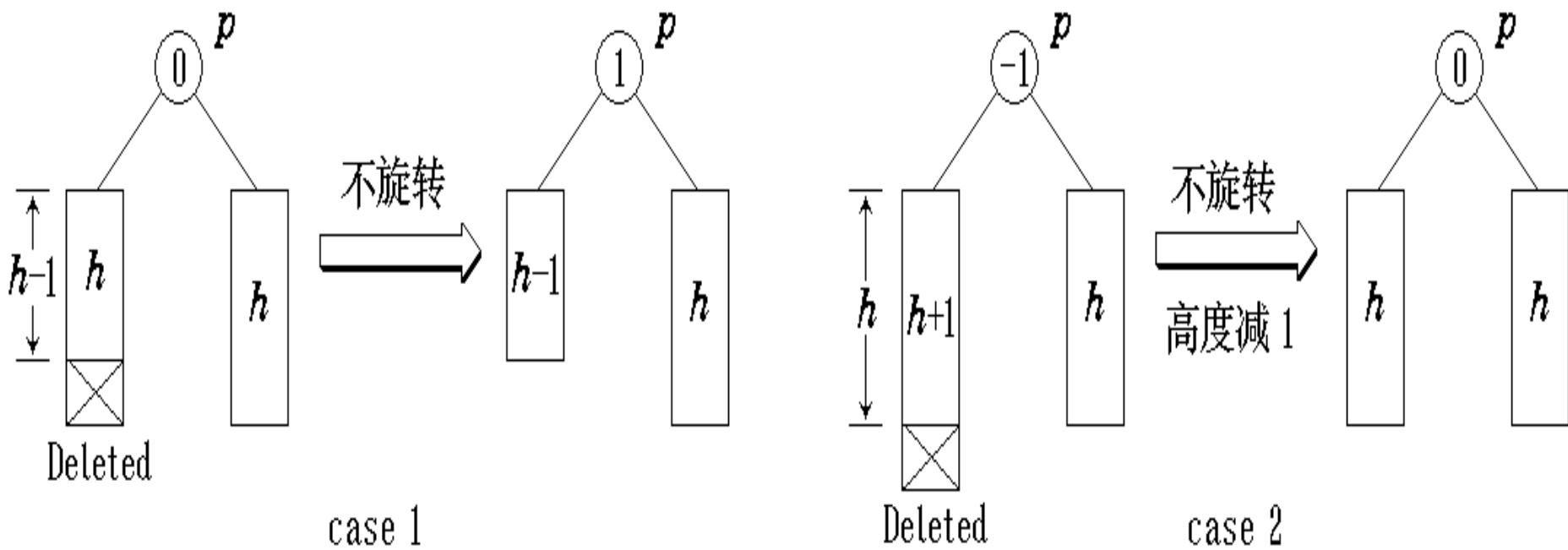
- 如果被删结点x最多只有一个子女，那么问题比较简单。如果被删结点x有两个子女，首先搜索 x 在中序次序下的直接前驱 y (同样可以找直接后继)。再把 结点y 的内容传送给结点x，现在问题转移到删除结点 y。

把结点y当作被删结点x。

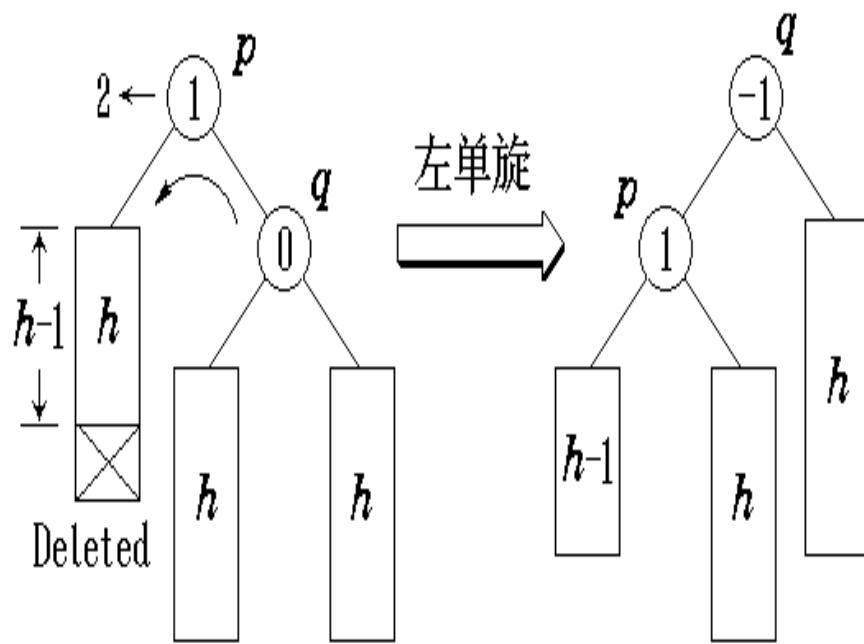
- 将结点x从树中删去。因为结点x最多有一个子女，我们可以简单地把x的双亲结点中原来指向x的指针改指到这个子女结点；如果结点x没有子女，x双亲结点的相应指针置为NULL。然后将原来以结点x为根的子树的高度减1，

- 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* 时执行下面的操作。如果 *shorter* 变成 *False*，算法终止。

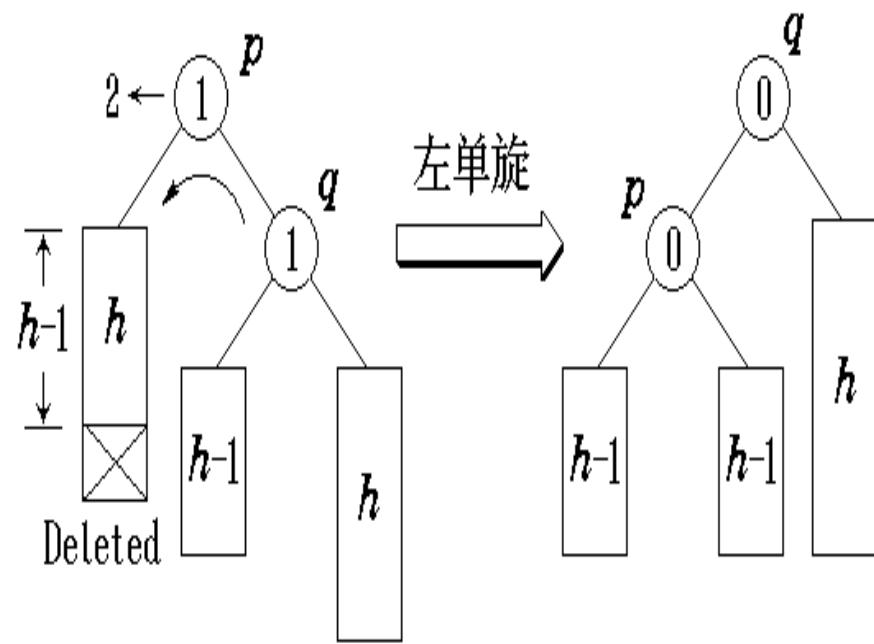
- case 1 : 当前结点 p 的 $balance$ 为 0。如果它的左子树或右子树被缩短，则它的 $balance$ 改为 1 或 -1，同时 $shorter$ 置为 *False*。
- case 2 : 结点 p 的 $balance$ 不为 0，且较高的子树被缩短，则 p 的 $balance$ 改为 0，同时 $shorter$ 置为 *True*。



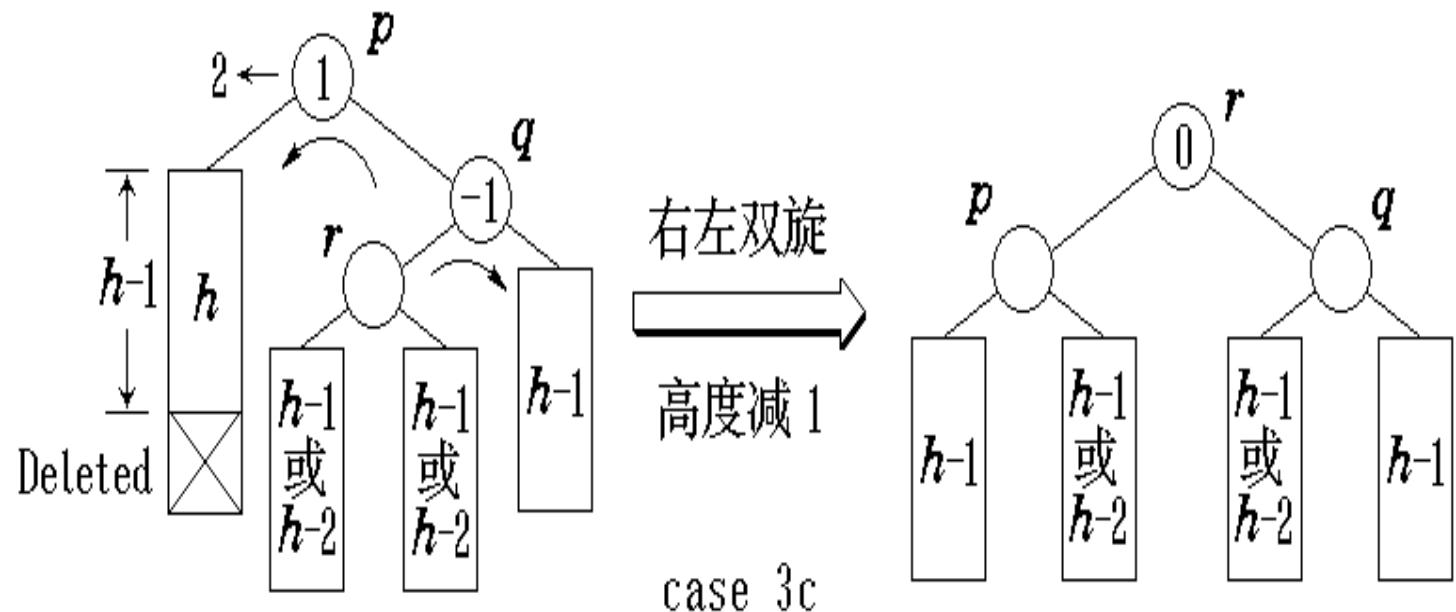
- case 3 : 结点 p 的 $balance$ 不为0，且较矮的子树又被缩短，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。令 p 的较高的子树的根为 q (该子树未被缩短)，根据 q 的 $balance$ ，有如下 3 种平衡化操作。
- case 3a : 如果 q 的 $balance$ 为0，执行一个单旋转来恢复结点 p 的平衡，置 $shorter$ 为 $False$ 。
- case 3b : 如果 q 的 $balance$ 与 p 的 $balance$ 相同，则执行一个单旋转来恢复平衡，结点 p 和 q 的 $balance$ 均改为0，同时置 $shorter$ 为 $True$ 。



case 3a



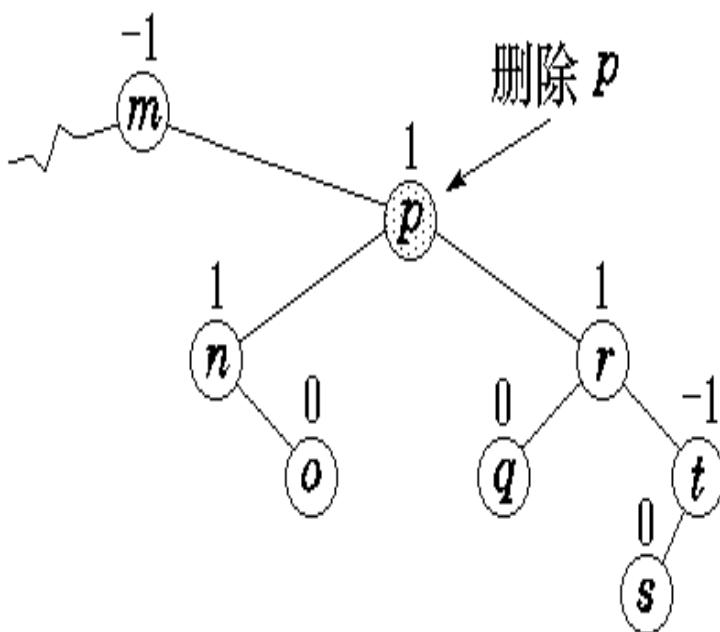
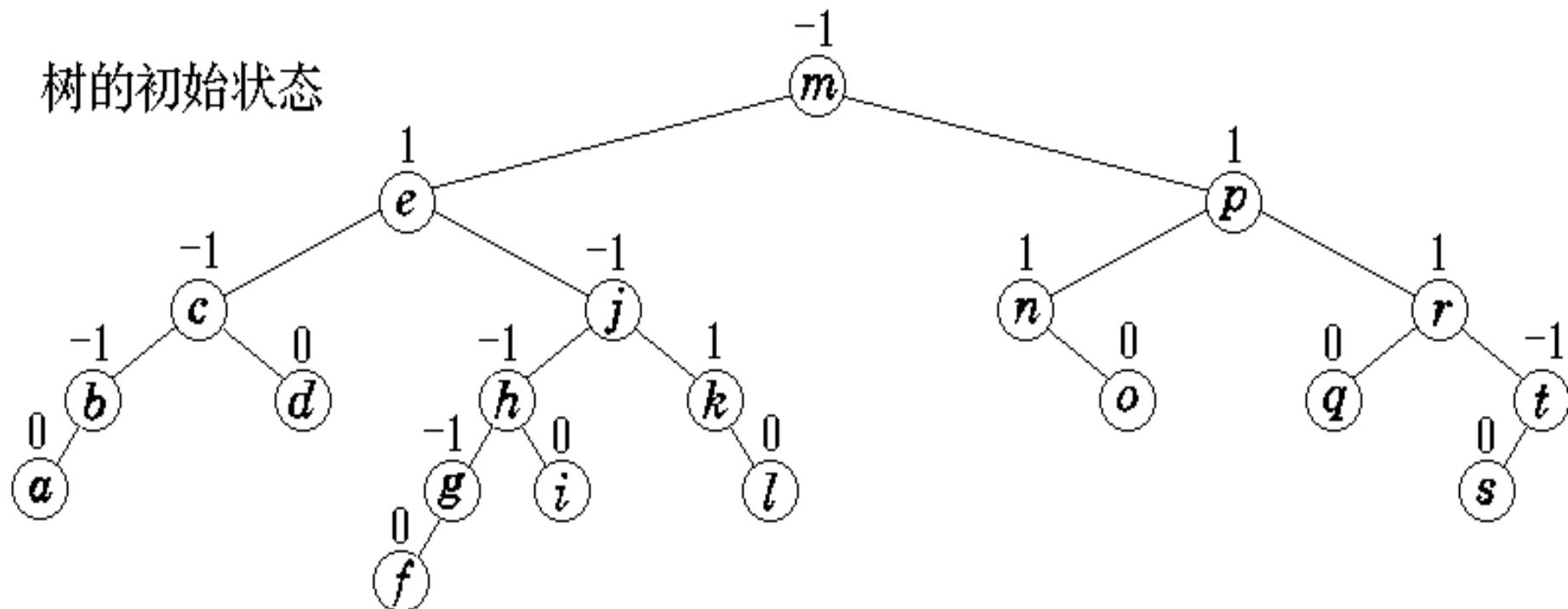
case 3b



case 3c

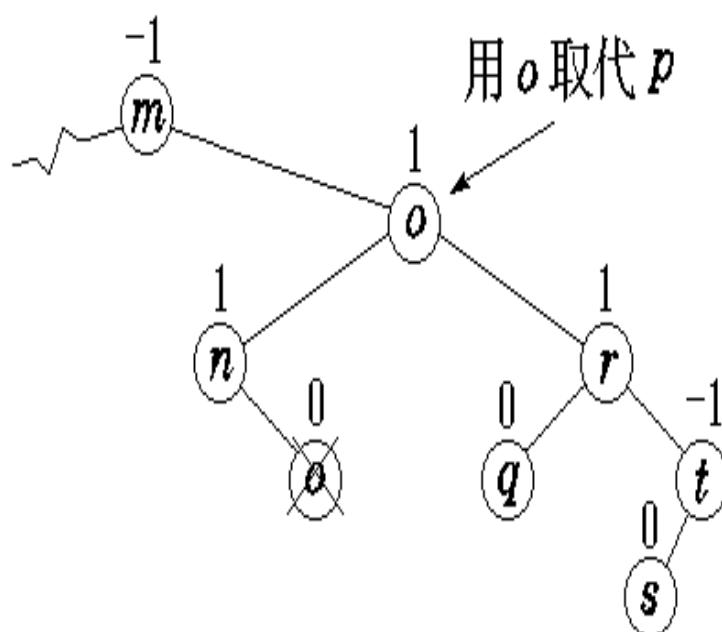
- case 3c : 如果 p 与 q 的 $balance$ 相反, 则执行一个双旋转来恢复平衡, 先围绕 q 转再围绕 p 转。新的根结点的 $balance$ 置为0, 其它结点的 $balance$ 相应处理, 同时置 $shorter$ 为 $True$ 。
- 在 case 3a, 3b 和 3c 的情形中, 旋转的方向取决于是结点 p 的哪一棵子树被缩短。

树的初始状态

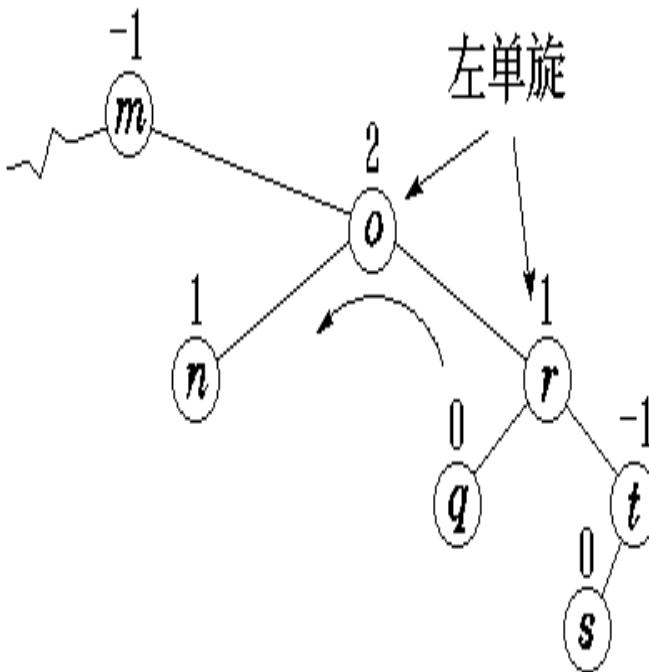


删除 p

寻找 p 的中序下的
的直接前驱 o ，
用 o 取代 p ，删
除 o ，平衡旋转

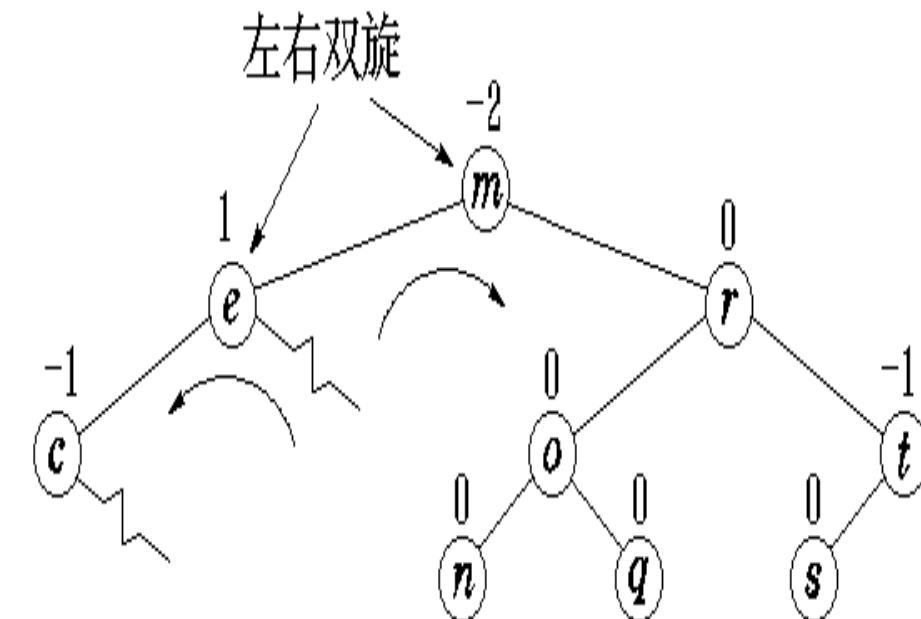


用 o 取代 p



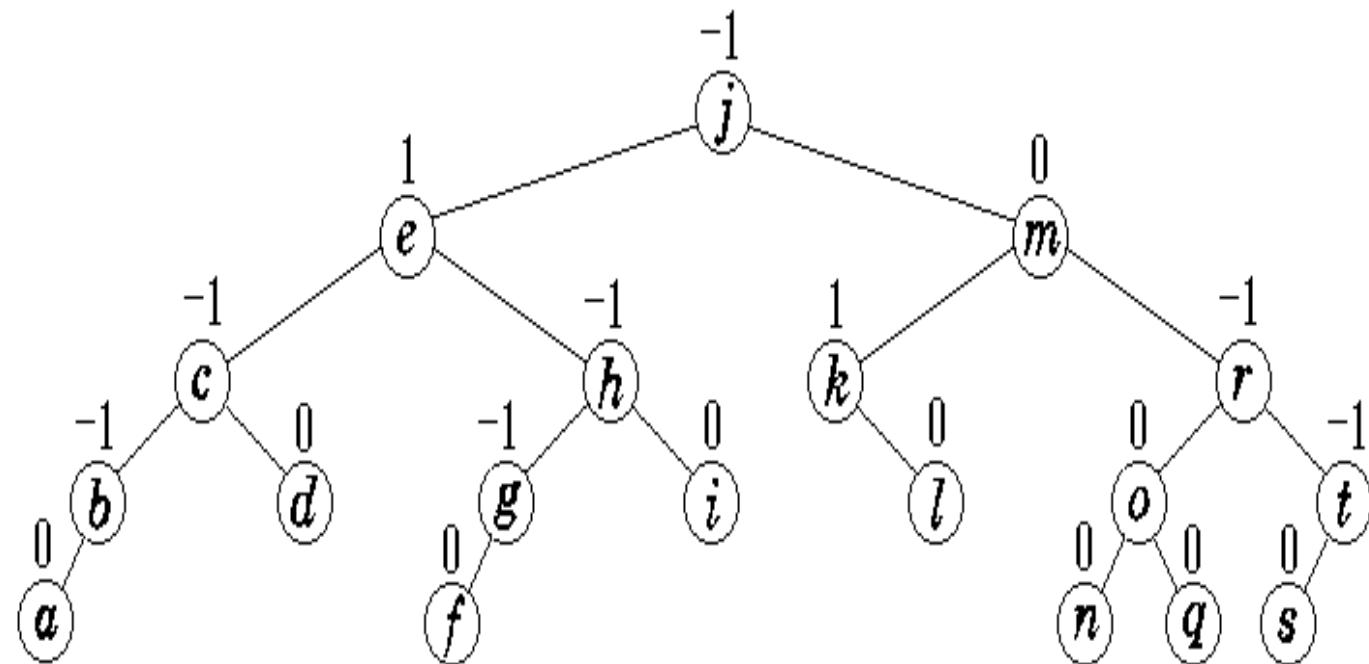
左单旋

以 r 为旋转轴
作左单旋，子
树的高度减 1
 m 发生不平衡



左右双旋

首先以 j 为旋转轴作
左单旋，再以 j 为旋
转轴作右单旋，让 e
成为 j 的左子女， m
成为 j 的右子女。树
的高度减 1



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 $h-2$, 这两棵子树也是高度平衡的。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

$$\frac{3}{2} \log_2(n + 1)$$

- 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。



小结 需要复习的知识点

- 理解 集合及其表示：
 - ◆ 集合基本概念
 - ◆ 用位向量实现集合 ADT
 - ◆ 用有序链表实现集合 ADT
 - ◆ 集合的应用
- 一般集合与位向量建立对应
- 用有序链表实现集合的操作

- 理解 等价类：
 - ◆ 等价关系与等价类
 - ◆ 确定等价类的链表方法
 - ◆ 并查集
- 实现等价类的链表算法
- 实现等价类的并查集算法
- 并查集的查找和合并操作

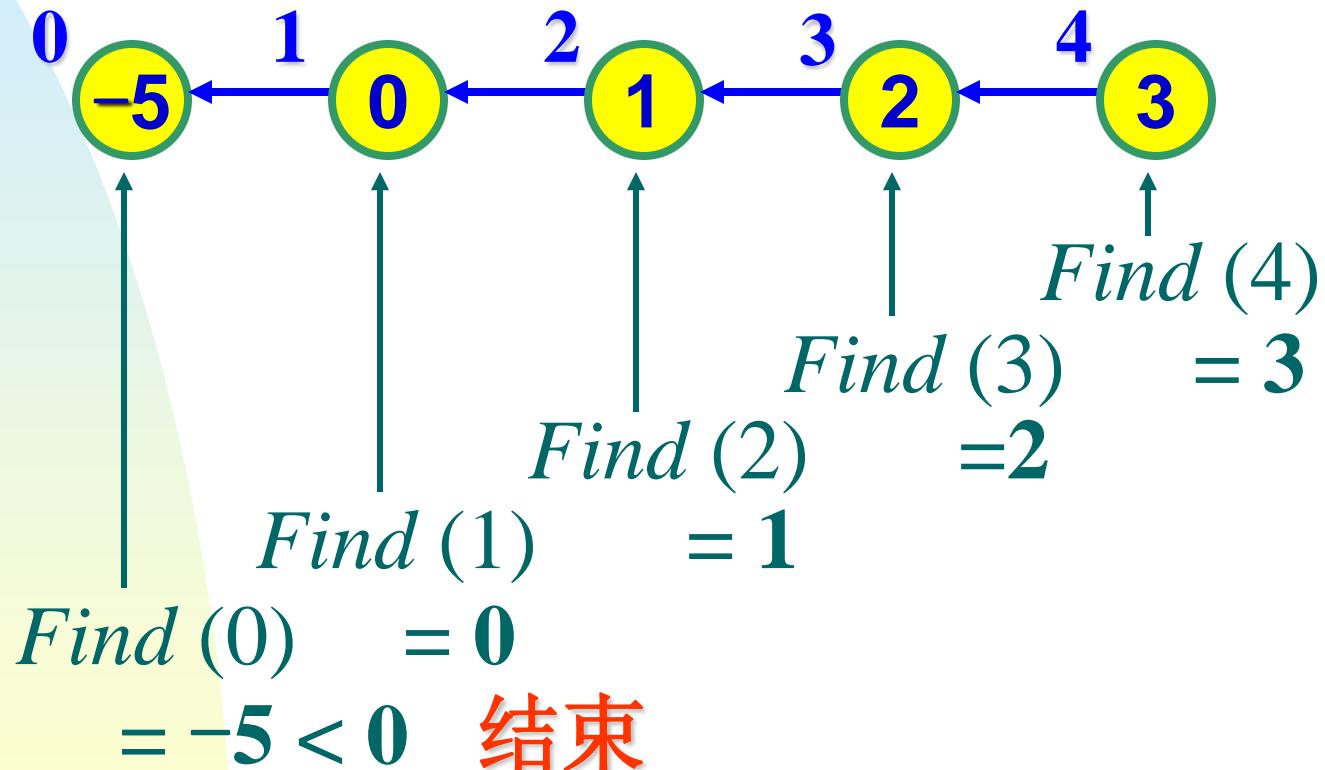
- 掌握 简单的搜索结构：
 - ◆ 搜索的概念
 - 搜索结构
 - 搜索的判定树
 - 平均搜索长度
 - ◆ 静态搜索
 - 顺序搜索算法、分析
 - 折半搜索算法、分析

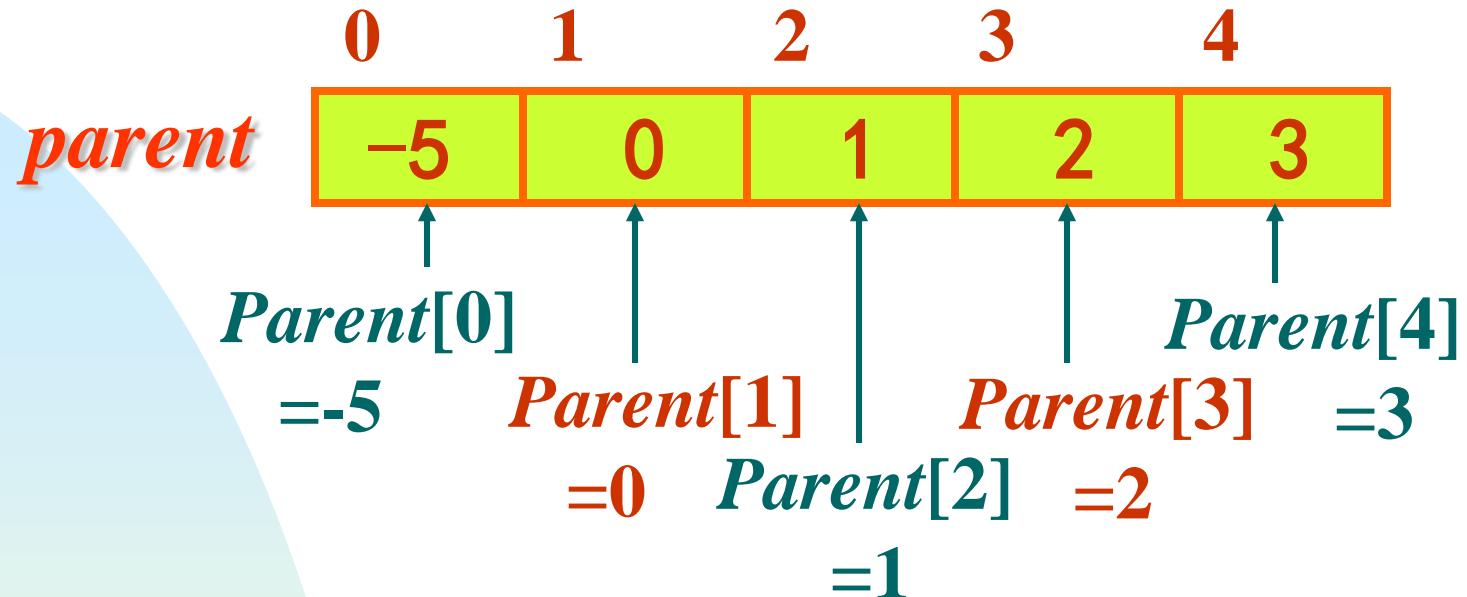
- ◆ 二叉搜索树
 - 定义
 - 搜索、 平均搜索长度
 - 插入、 删除、
- ◆ AVL树
 - 定义
 - 插入、 平衡化旋转
 - 删除、 平衡化旋转
 - 高度



并查集操作的算法

■ 查找

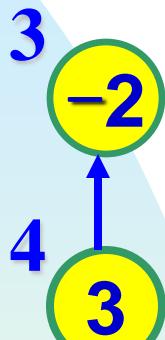
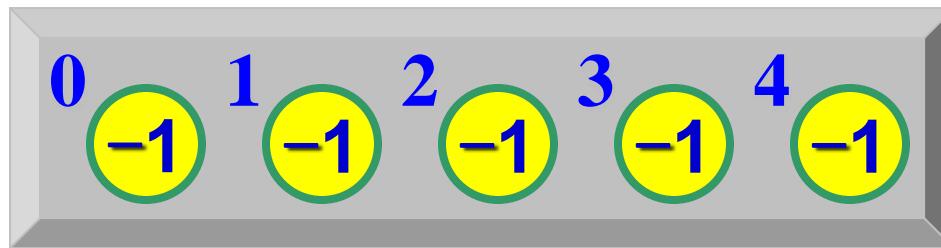




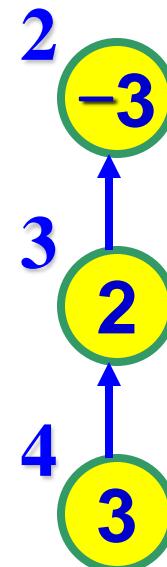
```

int Find ( int x ) {
  if ( parent [x] < 0 ) return x;
  else return Find ( parent [x] );
}
  
```

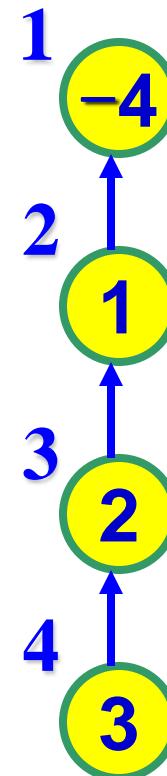
- 合并



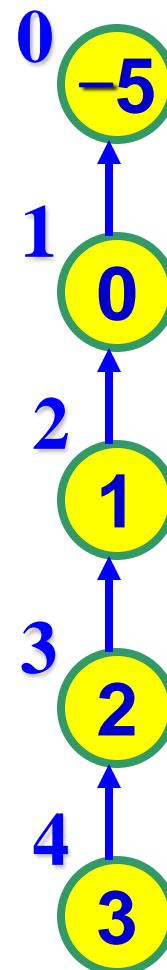
merge(3,4)



merge(2,3)

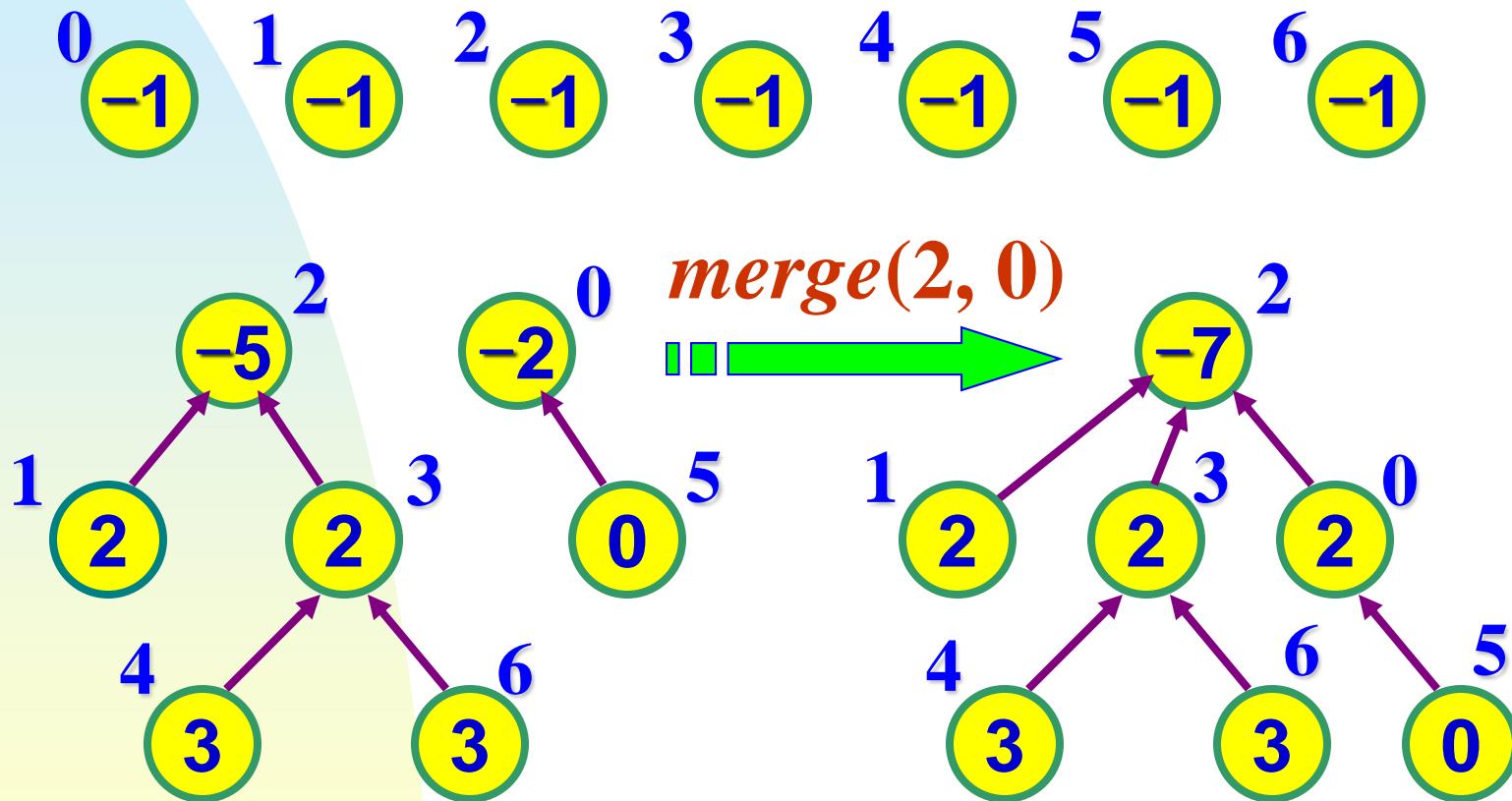


merge(1,2)

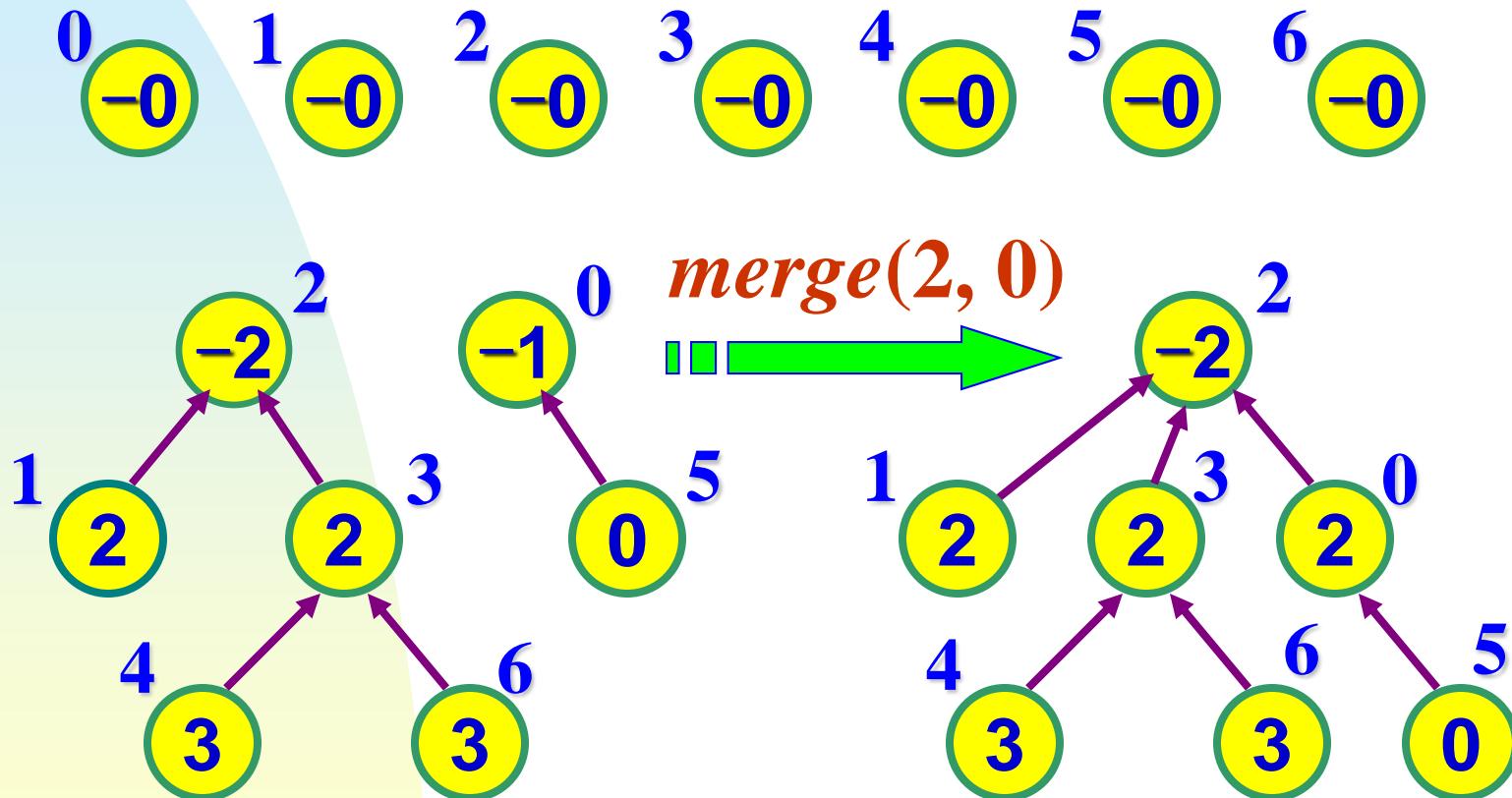


merge(0,1)

- 按树结点个数合并
 - 结点个数多的树的根结点作根



- 按树高度合并
 - 高度高的树的根结点作根





搜索的概念

- ◆ 搜索结构决定搜索的效率
- ◆ 搜索算法基于搜索结构
- ◆ 搜索效率用平均搜索长度衡量
- ◆ 平均搜索长度表明搜索算法的整体性能，避开偶然因素
- ◆ 平均搜索长度分搜索成功与搜索不成功两种情况

■ 静态搜索结构

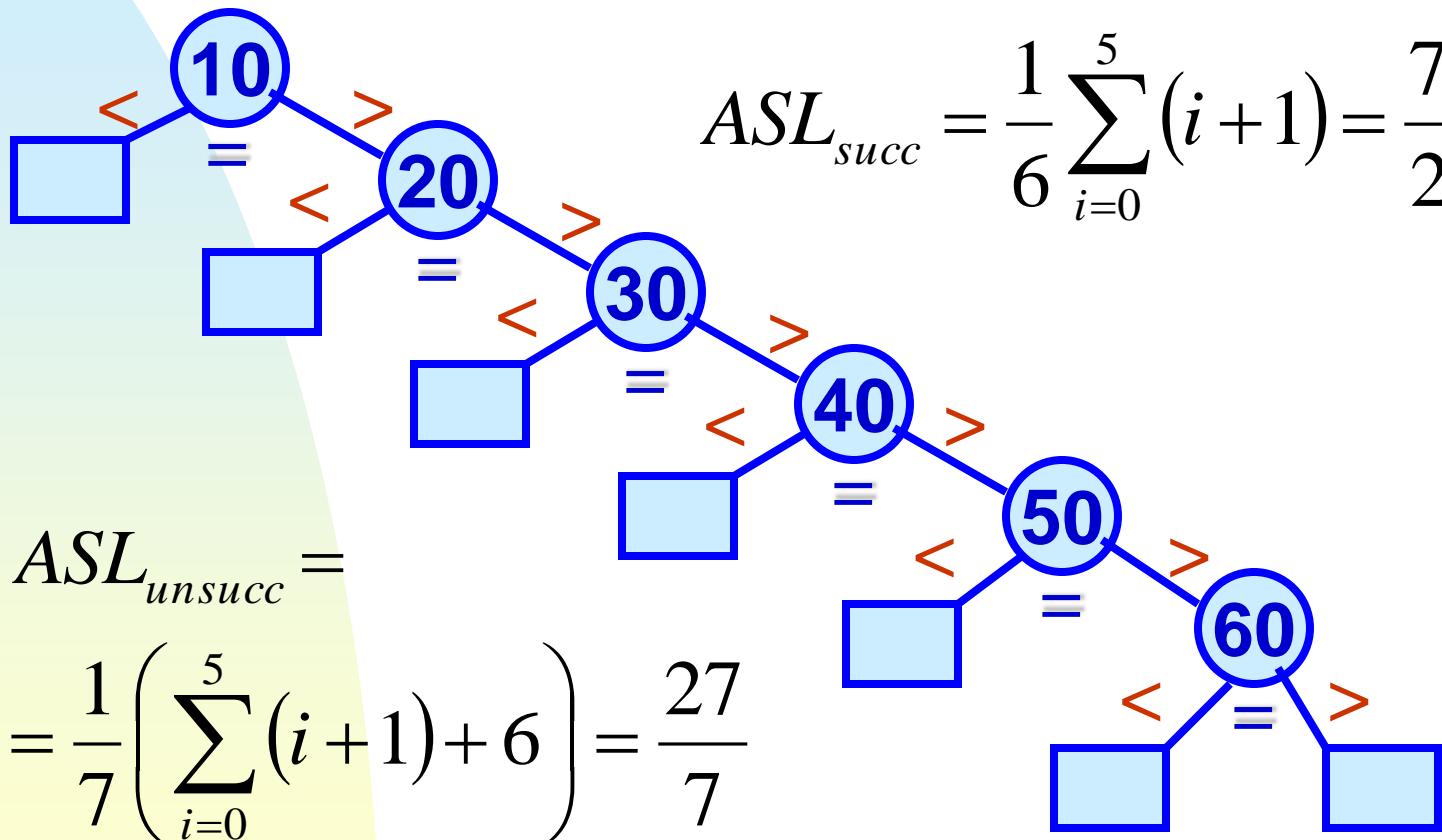
- ◆ 顺序搜索 — 顺序表、链表
- ◆ 折半搜索 — 有序顺序表

■ 动态搜索结构

- ◆ 二叉搜索树 — 无重复关键码
- ◆ AVL树 — 平衡二叉搜索树

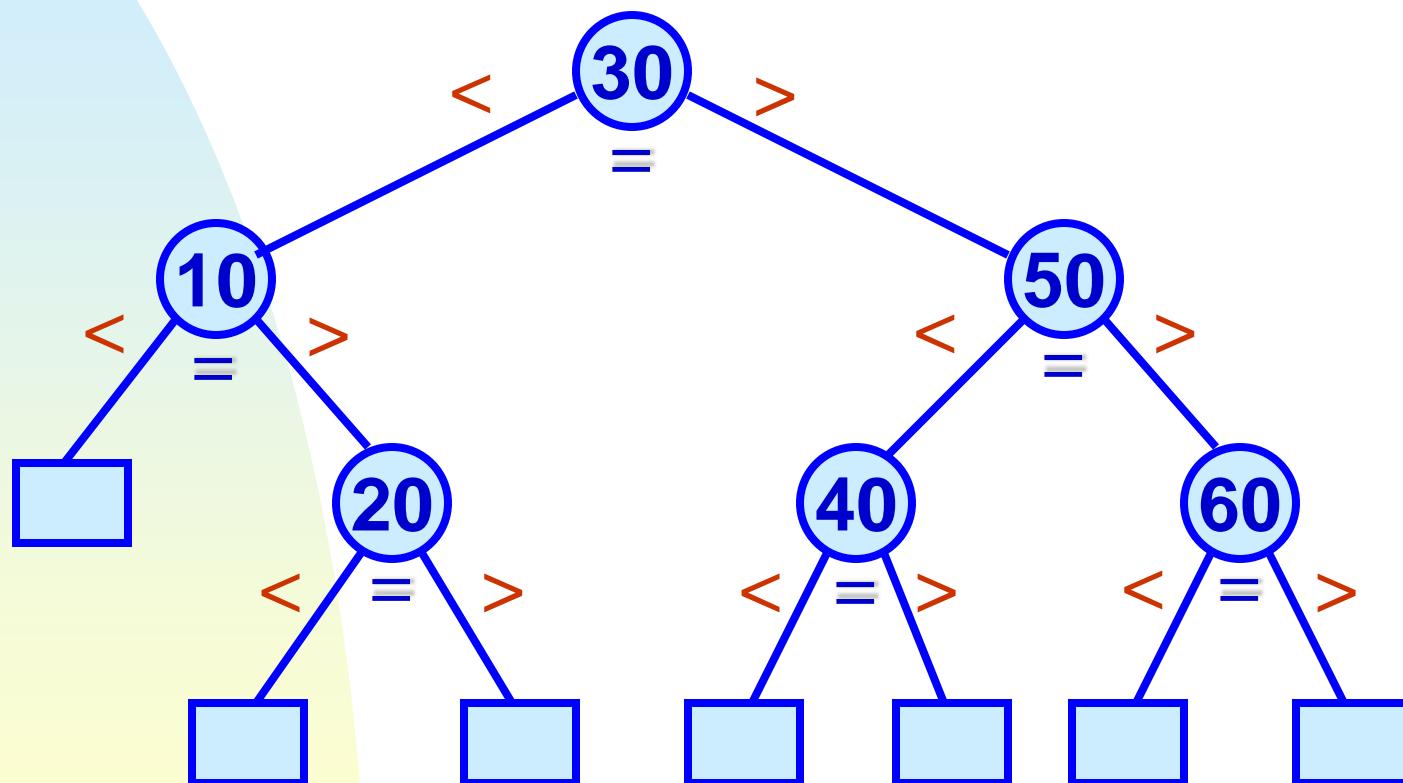
■ 有序顺序表的顺序搜索

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



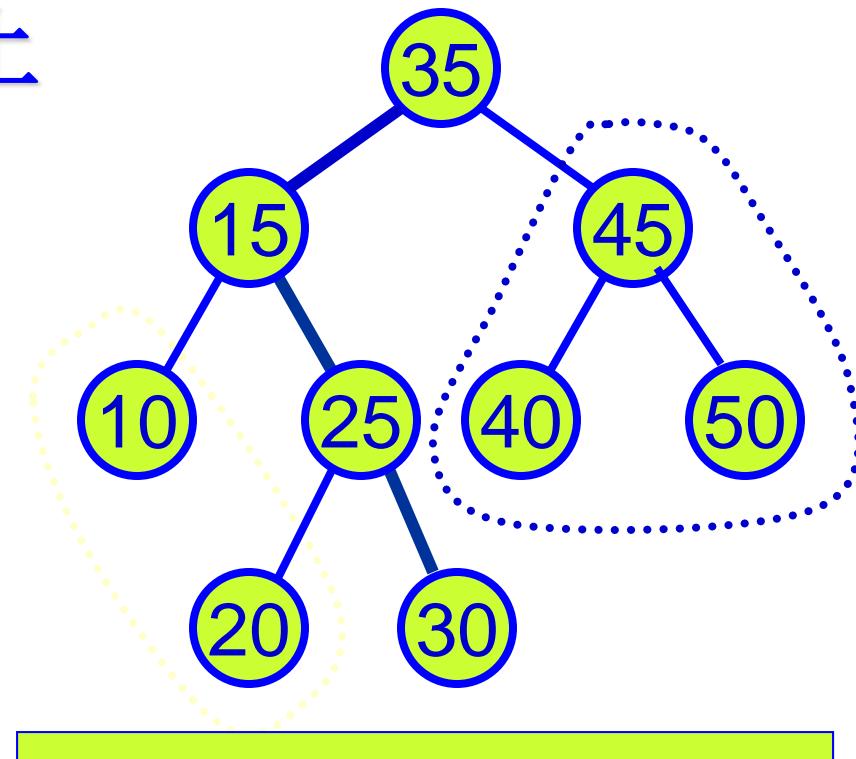
■ 有序顺序表的折半搜索

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



■ 二叉搜索树

- ◆ 二叉搜索树的子树是二叉搜索树
- ◆ 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

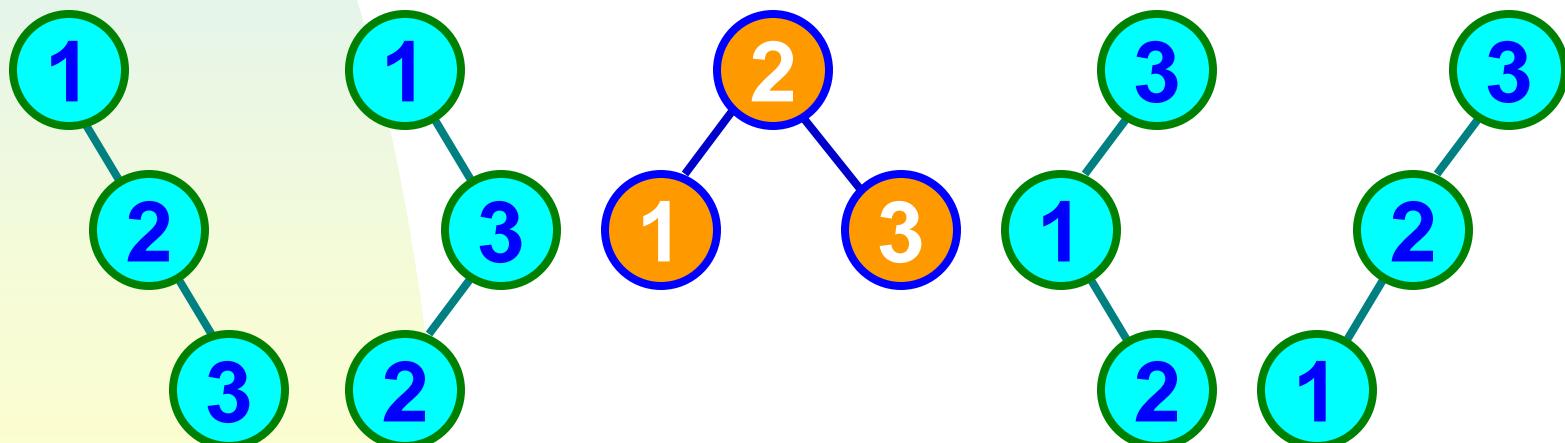


n 个结点的二叉搜索树的数目

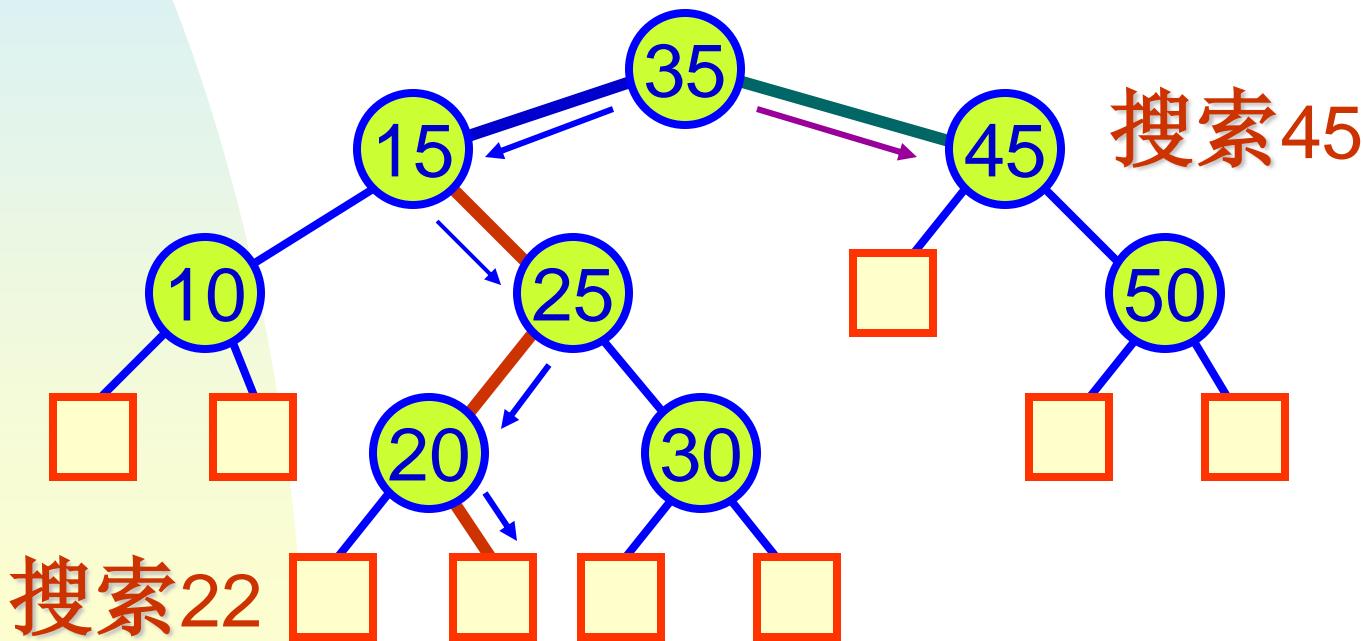
【例】3 个结点的二叉搜索树

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

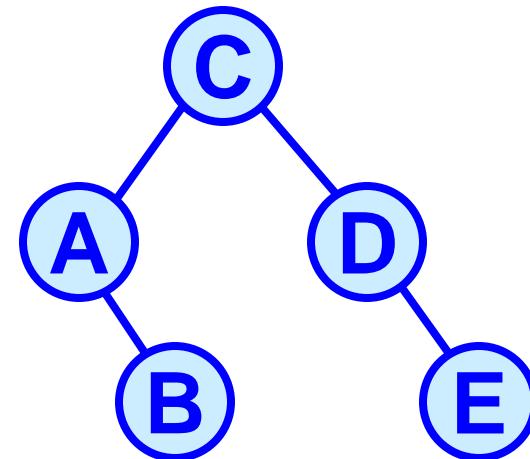
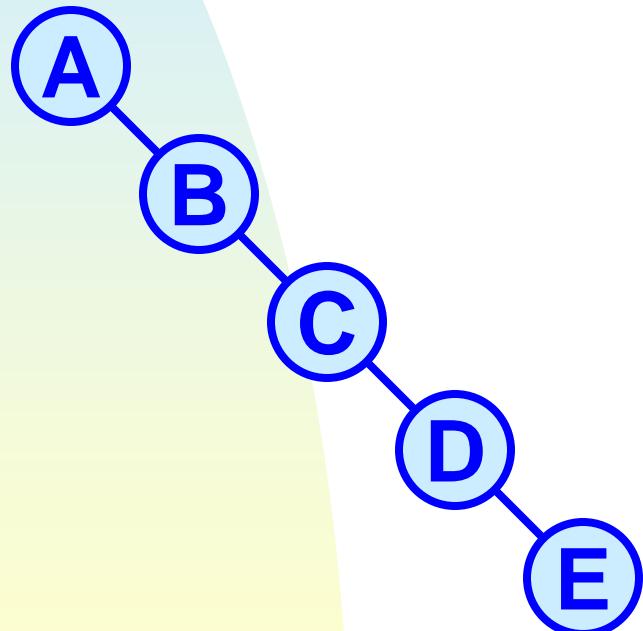
{123} {132} {213} {231} {312} {321}



- ◆ 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）。



- 二叉搜索树的高度越小， 平均搜索长度越小。
- n 个结点的二叉搜索树的高度最大为 $n-1$, 最小为 $\lfloor \log_2 n \rfloor$.



■ AVL树

- ◆ 理解：AVL树的子树也是AVL树
- ◆ 掌握：插入新结点后平衡化旋转的方法
- ◆ 掌握：删除结点后平衡化旋转的方法
- ◆ 掌握：结点高度 h 与结点数 n 的关系