第 7 章 集合与搜索

7-1 设 A = { 1, 2, 3 }, B = { 3, 4, 5 }，求下列结果：

(1) A + B (2) A * B (3) A - B

(4) A.Contains (1) (5) A.AddMember (1) (6) A.DelMember (1) (7) A.Min ( )

【解答】

(1) 集合的并 A + B = { 1, 2, 3, 4, 5 }

(2) 集合的交 A * B = { 3 }

(3) 集合的差 A - B = { 1, 2 }

(4) 包含 A.Contains (1) = 1，表示运算结果为"True"

(5) 增加 A.AddMember (1)，集合中仍为{ 1, 2, 3 }，因为增加的是重复元素，所以不加入

(6) 删除 A.DelMember (1)，集合中为{ 2, 3 }

(7) 求最小元素 A.Min ( )，结果为 1

7-2 试编写一个算法，打印一个有穷集合中的所有成员。要求使用集合抽象数据类型中的基本操作。如果集合中包含有子集合，各个子集合之间没有重复的元素，采用什么结构比较合适。

【解答】

集合抽象数据类型的部分内容

```
template <class Type> class Set {
//对象: 零个或多个成员的聚集。其中所有成员的类型是一致的, 但没有一个成员是相同的。
    int Contains ( const Type x );                  //判元素 x 是否集合 this 的成员
    int SubSet ( Set <Type>& right );               //判集合 this 是否集合 right 的子集
    int operator == ( Set <Type>& right );          //判集合 this 与集合 right 是否相等
    int Elemtype ( );                               //返回集合元素的类型
    Type GetData ( );                               //返回集合原子元素的值
    char GetName ( );                                   //返回集合 this 的集合名
    Set <Type>* GetSubSet ( );                      //返回集合 this 的子集合地址
    Set <Type>* GetNext ( );                        //返回集合 this 的直接后继集合元素
    int IsEmpty ( );                                //判断集合 this 是否空。空则返回 1, 否则返回 0
};

ostream& operator << ( ostream& out, Set <Type> t ) {
//友元函数, 将集合 t 输出到输出流对象 out。
    t.traverse ( out, t );    return out;
}

void traverse ( ostream& out, Set <Type> s ) {
//友元函数, 集合的遍历算法
    if ( s.IsEmpty ( ) == 0 ) {                             //集合元素不空
        if ( s.Elemtype ( ) == 0 ) out << s.GetName ( ) << '{';     //输出集合名及花括号
        else if ( s.Elemtype ( ) == 1 ) {                   //集合原子元素
                out << s.GetData ( );                       //输出原子元素的值
```

```
            if ( s.GetNext ( ) != NULL ) out << ',';
        }
    else {                                          //子集合
            traverse ( s. GetSubSet ( ) );          //输出子集合
            if ( s.GetNext ( ) != NULL ) out << ',';
    }
    traverse ( s.GetNext ( ) );                     //向同一集合下一元素搜索
  }
  else out << '}';
}
```

如果集合中包含有子集合，各个子集合之间没有重复的元素，采用广义表结构比较合适。也可以使用并查集结构。

7-3 当全集合可以映射成 1 到 N 之间的整数时，可以用位数组来表示它的任一子集合。当全集合是下列集合时，应当建立什么样的映射？用映射对照表表示。

(1) 整数 0, 1, …, 99

(2) 从 n 到 m 的所有整数，n ≤ m

(3) 整数 n, n+2, n+4, …, n+2k

(4) 字母 'a', 'b', 'c', …, 'z'

(5) 两个字母组成的字符串, 其中， 每个字母取自 'a', 'b', 'c', …, 'z'。

【解答】

(1) i → i 的映射关系，i = 0, 1, 2, …, 99

(2) i → n−i 的映射关系，i = n, n+1, n+2, …, m

0	1	2		m−n
n	n+1	n+2	…	m

(3) i → (i−n)/2 的映射关系，i = n, n+2, n+4, …, n+2k

0	1	2		k
n	n+2	n+4	…	n+2k

(4) ord (c) → ord (c) − ord ('a') 的映射关系，c = 'a', 'b', 'c', …, 'z'

0	1	2		25
'a'	'b'	'c'	…	'z'

(5) (ord (c1) − ord ('a'))*26 + ord (c2) − ord ('a')的映射关系，c1 = c2 = 'a', 'b', 'c', …, 'z'

0	1	2		675
'aa'	'ab'	'ba'	…	'zz'

7-4 试证明：集合 A 是集合 B 的子集的充分必要条件是集合 A 和集合 B 的交集是 A。

【证明】

必要条件：因为集合 A 是集合 B 的子集，有 $A \subseteq B$，此时，对于任一 $x \in A$，必有 $x \in B$，因此可以推得 $x \in A \cap B$，就是说，如果 A 是 B 的子集，一定有 $A \cap B = A$。

充分条件：如果集合 A 和集合 B 的交集 $A \cap B$ 是 A，则对于任一 $x \in A$，一定有 $x \in A \cap B$，因此可

以推得 x ∈ B，由此可得 A⊆B，即集合 A 是集合 B 的子集。

7-5 试证明：集合 A 是集合 B 的子集的充分必要条件是集合 A 和集合 B 的并集是 B。

【证明】

必要条件：因为集合 A 是集合 B 的子集，有 A⊆B，此时，对于任一 x ∈ A，必有 x ∈ B， 它一定在 A∪B 中。另一方面，对于那些 x' ∉ A, 但 x' ∈ B 的元素，它也必在 A∪B 中，因此可以得出结论：凡是属于集合 B 的元素一定在 A∪B 中，A∪B = B。

充分条件：如果存在元素 x ∈ A 且 x ∉ B，有 x ∈ A∪B，但这不符合集合 A 和集合 B 的并集 A∪B 是 B 的要求。集合的并 A∪B 是集合 B 的要求表明，对于任一 x ∈ A∪B，同时应有 x ∈ B。对于那些满足 x' ∈ A 的 x'，既然 x' ∈ A∪B，也应当 x' ∈ B，因此，在此种情况下集合 A 应是集合 B 的子集。

7-6 设+、*、−是集合的或、与、差运算，试举一个例子，验证

A + B = (A − B) + (B − A) + A * B

【解答】

若设集合 A = { 1, 3, 4, 7, 9, 15 }，集合 B = { 2, 3, 5, 6, 7, 12, 15, 17 }。则

A + B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

又 A * B = { 3, 7, 15 }， A – B = { 1, 4, 9 }， B – A = { 2, 5, 6, 12, 17 }

则 (A – B) + (B – A) + A * B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

有 A + B = (A – B) + (B – A) + A * B。

7-7 给定一个用无序链表表示的集合，需要在其上执行 operator+( ), operator*( ), operator− ( ), Contains(x), AddMember (x), DelMember(x), Min( )，试写出它的类声明，并给出所有这些成员函数的实现。

【解答】

下面给出用无序链表表示集合时的类的声明。

```
template <class Type> class Set;                          //用以表示集合的无序链表的类的前视定义
template <class Type> class SetNode {                     //集合的结点类定义
friend class SetList<Type>;
private:
    Type data;                                            //每个成员的数据
    SetNode <Type> *link;                                 //链接指针
public:
    SetNode (const Type& item ) : data (item), link (NULL);   //构造函数
};

template <class Type> class Set {                         //集合的类定义
private:
    SetNode <Type> *first, *last;                         //无序链表的表头指针，表尾指针
public:
    SetList ( ) { first = last = new SetNode <Type>(0); }     //构造函数
    ~SetList ( ) { MakeEmpty ( );   delete first; }           //析构函数
    void MakeEmpty ( );                                   //置空集合
    int AddMember ( const Type& x );                      //把新元素 x 加入到集合之中
    int DelMember ( const Type& x );                      //把集合中成员 x 删去
```

```
    Set <Type>& operator = ( Set <Type>& right );          //复制集合 right 到 this。
    Set <Type>& operator + ( Set <Type>& right );          //求集合 this 与集合 right 的并
    Set <Type>& operator * ( Set <Type>& right );          //求集合 this 与集合 right 的交
    Set <Type>& operator − ( Set <Type>& right );          //求集合 this 与集合 right 的差
    int Contains ( const Type& x );                        //判 x 是否集合的成员
    int operator == ( Set <Type>& right );                 //判集合 this 与集合 right 相等
    Type& Min ( );                                         //返回集合中的最小元素的值
}
```

(1) **operator** + ( )

```
template <class Type> Set <Type>& Set <Type> :: operator + ( Set <Type>& right ) {
//求集合 this 与集合 right 的并，计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode <Type> *pb = right.first−>link;                //right 集合的链扫描指针
    SetNode <Type> *pa, *pc;                               //this 集合的链扫描指针和结果链的存放指针
    Set <Type> temp;
    pa = first−>link;    pc = temp.first;
    while ( pa != NULL ) {                                 //首先把集合 this 的所有元素复制到结果链
        pc−>link = new SetNode<Type> ( pa−>data ) ;
        pa = pa−>link;    pc = pc−>link;
    }
    while ( pb != NULL ) {                                 //将集合 right 中元素一个个拿出到 this 中查重
        pa = first−>link;
        while ( pa != NULL && pa−>data != pb−>data ) pa = pa−>link;
        if ( pa == NULL )                                  //在集合 this 中未出现，链入到结果链
            { pc−>link = new SetNode<Type> ( pa−>data );    pc = pc−>link; }
        pb = pb−>link;
    }
    pc−>link = NULL;    last = pc;                         //链表收尾
    return temp;
}
```

(2) **operator** * ( )

```
template <class Type> Set <Type>& Set <Type> :: operator * ( Set <Type>& right ) {
//求集合 this 与集合 right 的交，计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode<Type> *pb = right.first−>link;                 //right 集合的链扫描指针
    Set <Type> temp;
    SetNode <Type> *pc = temp. first;                      //结果链的存放指针
    while ( pb != NULL ) {                                 //将集合 right 中元素一个个拿出到 this 中查重
        SetNode<Type> *pa = first−>link;                   //this 集合的链扫描指针
        while ( pa != NULL ) {
            if ( pa−>data == pb−>data )                    //两集合公有的元素，插入到结果链
                { pc−>link = new SetNode<Type> ( pa−>data );    pc = pc−>link; }
            pa = pa−>link;
```

```
        }
        pb = pb->link;
    }
    pc->link = NULL;   last = pc;                    //置链尾指针
    return temp;
}
```

(3) **operator** − ( ),

```
template <class Type> Set <Type>& Set <Type> :: operator − ( Set <Type>& right ) {
//求集合 this 与集合 right 的差, 计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode<Type> *pa = first->link;              //this 集合的链扫描指针
    Set <Type> temp;
    SetNode<Type> *pc = temp->first;              //结果链的存放指针
    while ( pa != NULL ) {                        //将集合 this 中元素一个个拿出到 right 中查重
        SetNode <Type> *pb = right.first->link;   //right 集合的链扫描指针
        while ( pb != NULL && pa->data != pb->data )
            pb = pb->link;
        if ( pb == NULL )                         //此 this 中的元素在 right 中未找到, 插入
            { pc->link = new SetNode <Type> ( pa->data );   pc = pc->link; }
        pa = pa->link;
    }
    pc->link = NULL;   last = pc;                 //链表收尾
    return temp;
}
```

(4) Contains(x)

```
template <class Type> int Set <Type> :: Contains ( const Type& x ) {
//测试函数: 如果 x 是集合的成员, 则函数返回 1, 否则返回 0。
    SetNode<Type> * temp = first->link;                    //链的扫描指针
    while ( temp != NULL && temp->data != x ) temp = temp->link;    //循链搜索
    if ( temp != NULL ) return 1;                          //找到, 返回 1
    else return 0;                                         //未找到, 返回 0
}
```

(5) AddMember (x)

```
template <class Type> int Set <Type> :: AddMember ( const Type& x ) {
//把新元素 x 加入到集合之中。若集合中已有此元素, 则函数返回 0, 否则函数返回 1。
    SetNode<Type> * temp = first->link;                    // temp 是扫描指针
    while ( temp != NULL && temp->data != x ) temp = temp->link;    /循链扫描
    if ( temp != NULL ) return 0;                          //集合中已有此元素, 不加
    last = last->link = new SetNode (x);                   //否则, 创建数据值为 x 的新结点, 链入
    return 1;
}
```

(6) DelMember (x)

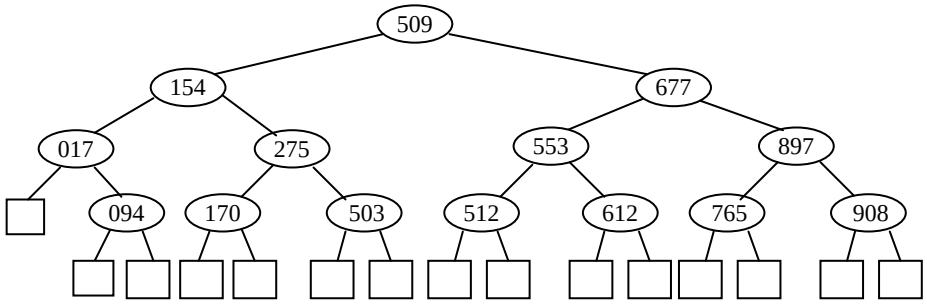
```
template <class Type> int Set <Type> :: DelMember ( const Type& x ) {
//把集合中成员 x 删去。若集合不空且元素 x 在集合中，则函数返回 1，否则返回 0。
    SetNode<Type> * p = first->link,   *q = first;
    while ( p != NULL ) {
        if ( p->data == x ) {                          //找到
            q->link = p->link;                         //重新链接
            if ( p == last ) last = q;                 //删去链尾结点时改链尾指针
            delete p;   return 1;                      //删除含 x 结点
        }
        else { q = p;   p = p->link; }                 //循链扫描
    return 0;                                          //集合中无此元素
}
```

(7) Min ( )

```
template <class Type> SetNode<Type> * Set <Type> :: Min ( ) {
//在集合中寻找值最小的成员并返回它的位置。
    SetNode<Type> * p = first->link,   *q = first->link;   //p 是检测指针, q 是记忆最小指针
    while ( p != NULL ) {
        if ( p->data < q->data ) q = p;                //找到更小的，让 q 记忆它
        p = p->link;                                   //继续检测
    }
    return q;
}
```

7-8 设有序顺序表中的元素依次为 017, 094, 154, 170, 275, 503, 509, 512, 553, 612, 677, 765, 897, 908。试画出对其进行折半搜索时的二叉搜索树，并计算搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

【解答】



$$ASL_{succ} = \frac{1}{14}\sum_{i=1}^{14} C_i = \frac{1}{14}(1+2*2+3*4+4*7) = \frac{45}{14}$$

$$ASL_{unsucc} = \frac{1}{15}\sum_{i=0}^{15} C_i' = \frac{1}{15}(3*1+4*14) = \frac{59}{15}$$

7-9 若对有 n 个元素的有序顺序表和无序顺序表进行顺序搜索，试就下列三种情况分别讨论两者在等搜索概率时的平均搜索长度是否相同?

(1) 搜索失败;

(2) 搜索成功，且表中只有一个关键码等于给定值 k 的对象;

(3) 搜索成功，且表中有若干个关键码等于给定值 k 的对象，要求一次搜索找出所有对象。

【解答】

(1) 不同。因为有序顺序表搜索到其关键码比要查找值大的对象时就停止搜索，报告失败信息，不必搜索到表尾；而无序顺序表必须搜索到表尾才能断定搜索失败。

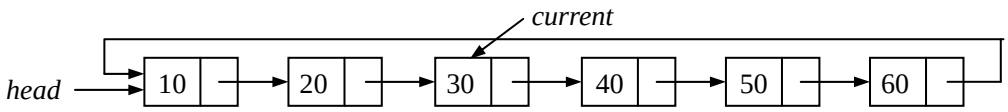
(2) 相同。搜索到表中对象的关键码等于给定值时就停止搜索，报告成功信息。

(3) 不同。有序顺序表中关键码相等的对象相继排列在一起，只要搜索到第一个就可以连续搜索到其它关键码相同的对象。而无序顺序表必须搜索全部表中对象才能确定相同关键码的对象都找了出来，所需时间就不相同了。

前两问可做定量分析。第三问推导出的公式比较复杂，不再进一步讨论。

7-10 假定用一个循环链表来实现一个有序表，并让指针 head 指向具有最小关键码的结点。指针 current 初始时等于 head，每次搜索后指向当前检索的结点，但如果搜索不成功则 current 重置为 head。试编写一个函数 search(head, current, key)实现这种搜索。当搜索成功时函数返回被检索的结点地址，若搜索不成功则函数返回空指针 0。请说明如何保持指针 current 以减少搜索时的平均搜索长度。

【解答】

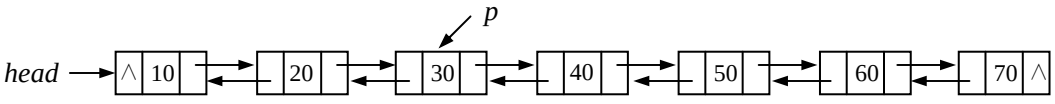


相应的搜索函数可以定义为链表及链表结点类的友元函数，直接使用链表及链表结点类的私有数据成员。

```
template<class Type>
ListNode <Type> * Search ( ListNode<Type> * head, ListNode<Type> *& current, Type key ) {
    ListNode <Type> * p, * q;
    if ( key < current ) { p = head;    q = current; }                    //确定检测范围, 用 p, q 指示
    else { p = current;    q = head; }
    while ( p != q && p->data < key ) p = p->link;                        //循链搜索其值等于 key 的结点
    if ( p->data == key ) { current = p;    return p; }                   //找到, 返回结点地址
    else { current = head;    return NULL; }                              //未找到, 返回空指针
}
```

7-11 考虑用双向链表来实现一个有序表，使得能在这个表中进行正向和反向搜索。若指针 p 总是指向最后成功搜索到的结点，搜索可以从 p 指示的结点出发沿任一方向进行。试根据这种情况编写一个函数 search(head, p, key)，检索具有关键码 key 的结点，并相应地修改 p。最后请给出搜索成功和搜索不成功时的平均搜索长度。

【解答】



```
template <class Type>
DblListNode<Type> * Search ( DblListNode<Type> * head,   DblListNode<Type> *& p, Type key ) {
//在以 head 为表头的双向有序链表中搜索具有值 key 的结点。算法可视为双向链表类和双向链表结点类的友元
//函数。若给定值 key 大于结点 p 中的数据，从 p 向右正向搜索，否则，从 p 向左反向搜索。
    DblListNode<Type> * q = p;
    if ( key < p->data ) { while ( q != NULL && q->data > key ) q = q-> lLink; }        //反向搜索
    else { while ( q != NULL && q->data < key ) q = q-> rLink; }                        //正向搜索
    if ( q != NULL && q->data == key ) { p = q;   return p; }                           //搜索成功
    else return NULL;
}
```

如果指针 p 处于第 i 个结点(i = 1, 2, …, n)，它左边有 i−1 个结点，右边有 n−i 个结点。找到左边第 i−1 号结点比较 2 次，找到第 i−2 号结点比较 3 次，…，找到第 1 号结点比较 i 次，一般地，找到左边第 k 个结点比较 i−k+1 次(k = 1, 2, …, i−1)。找到右边第 i+1 号结点比较 2 次，找到第 i+2 号结点比较 3 次，…，找到第 n 号结点比较 n−i+1 次，一般地，找到右边第 k 个结点比较 k−i+1 次(k = i+1, i+2, …, n)。因此，当指针处于第 i 个结点时的搜索成功的平均数据比较次数为

$$\left(1+\sum_{k=1}^{i-1}(i-k+1)+\sum_{k=i+1}^{n}(k-i+1)\right)\Big/n=\left(\frac{n*(n+3)}{2}+i^2-i-i*n\right)\Big/n=\frac{n+3}{2}+\frac{i^2-i-i*n}{n}$$

一般地，搜索成功的平均数据比较次数为

$$ASL_{succ}=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{n+3}{2}+\frac{i^2-i-i*n}{n}\right)=\frac{n^2+3n-1}{3n}$$

如果指针 p 处于第 i 个结点(i = 1, 2, …, n)，它左边有 i 个不成功的位置，右边有 n−i+1 个不成功的位置。

$$\left(\sum_{k=0}^{i-1}(i-k)+\sum_{k=i}^{n}(k-i+1)\right)\Big/(n+1)=\left(\frac{n*(n+3)}{2}+i^2-i-i*n+1\right)\Big/(n+1)$$

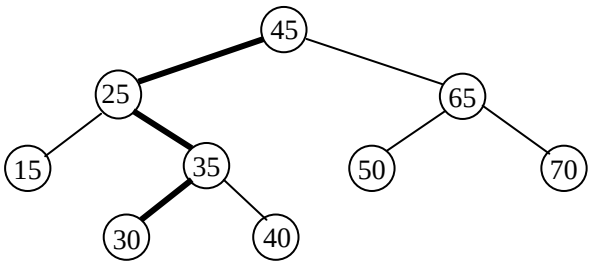
一般地，搜索不成功的平均数据比较次数为

$$ASL_{unsucc}=\frac{1}{(n+1)^2}\sum_{i=0}^{n}\left(\frac{n*(n+3)}{2}+i^2-i-i*n+1\right)=\frac{2n^2+7n+6}{n+1}$$

7−12 在一棵表示有序集 S 的二叉搜索树中，任意一条从根到叶结点的路径将 S 分为 3 部分：在该路径左边结点中的元素组成的集合 S_1；在该路径上的结点中的元素组成的集合 S_2；在该路径右边结点中的元素组成的集合 S_3。$S = S_1 \cup S_2 \cup S_3$。若对于任意的 $a \in S_1, b \in S_2, c \in S_3$， 是否总有 $a \le b \le c$？为什么？

【解答】

答案是否定的。举个反例：看下图粗线所示的路径



S1 = { 15 }， S2 = { 25, 30, 35, 45 }， S3 = { 40, 50, 65, 70 }
c = 40 ∈ S3，b = 45 ∈ S2，b ≤ c 不成立。

7-13 请给出下列操作序列运算的结果：Union(1, 2), Union(3, 4), Union(3, 5), Union(1, 7), Union(3, 6), Union(8, 9), Union(1, 8), Union(3, 10), Union(3, 11), Union(3, 12), Union(3, 13), Union(14, 15), Union(16, 17), Union(14, 16), Union(1, 3), Union(1, 14)，要求

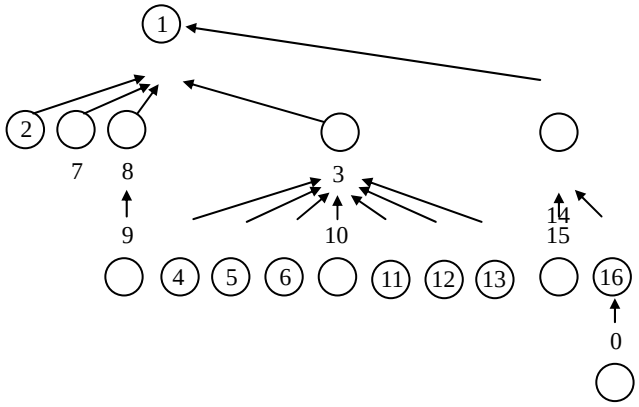
(1) 以任意方式执行 Union；

(2) 根据树的高度执行 Union；

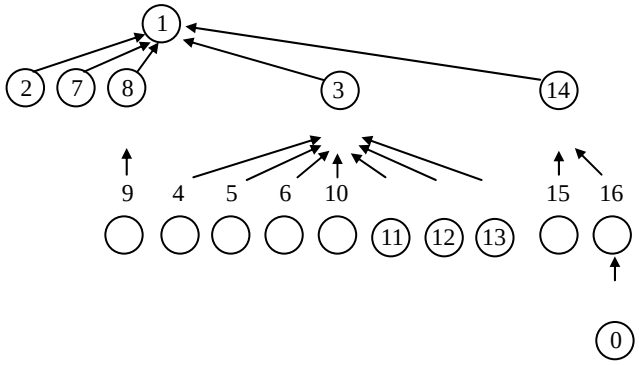
(3) 根据树中结点个数执行 Union。

【解答】

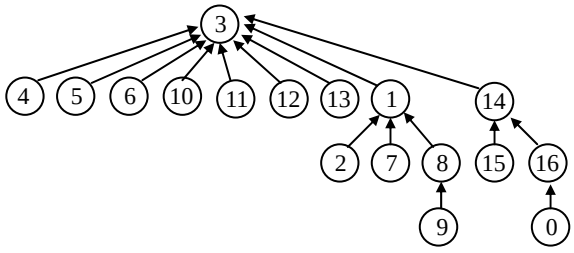
(1) 对于 union(i, j)，以 i 作为 j 的双亲



(2) 按 i 和 j 为根的树的高度实现 union(i, j)，高度大者为高度小者的双亲；



(3) 按 i 和 j 为根的树的结点个数实现 union(i, j)，结点个数大者为结点个数小者的双亲。



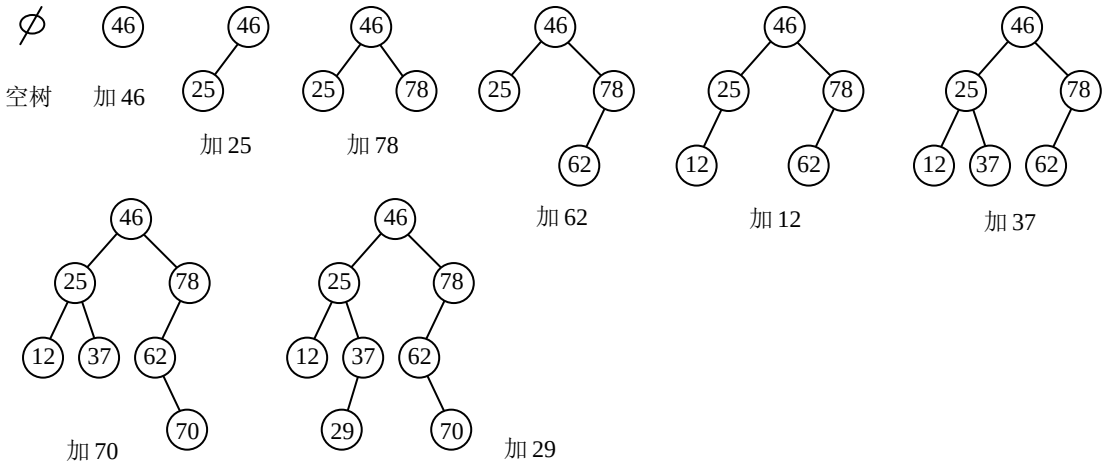
7-14 有 n 个结点的二叉搜索树具有多少种不同形态?

【解答】

$$\frac{1}{n+1}C_{2n}^{n}$$

7-15 设有一个输入数据的序列是 **{** 46, 25, 78, 62, 12, 37, 70, 29 **}**, 试画出从空树起，逐个输入各个数据而生成的二叉搜索树。

【解答】



7-16 设有一个标识符序列{else, public, return, template}，p_1=0.05, p_2=0.2, p_3=0.1, p_4=0.05, q_0=0.2, q_1=0.1, q_2=0.2, q_3=0.05, q_4=0.05，计算 W[i][j]、C[i][j]和 R[i][j]，构造最优二叉搜索树。

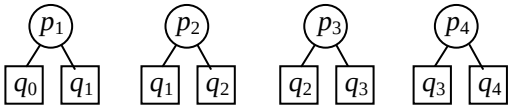
【解答】

将标识符序列简化为 { e, p, r, t }，并将各个搜索概率值化整，有

e	p	r	t
p_1 = 1	p_2 = 4	p_3 = 2	p_4 = 1

q_0 = 4	q_1 = 2	q_2 = 4	q_3 = 1	q_4 = 1

(1) 首先构造只有一个内结点的最优二叉搜索树:



三个矩阵的内容如下:

	0	1	2	3	4
0	4	7	15	18	20

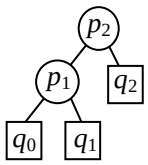
	0	1	2	3	4
0	0	7	22	32	39

	0	1	2	3	4
0	0	1	2	2	2

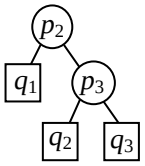
1		2	10	13	15	1		0	10	20	27	1		0	2	2	2
2			4	7	9	2			0	7	12	2			0	3	3
3				1	3	3				0	3	3				0	4
4					1	4					0	4					0
	W[i][j]						C[i][j]						R[i][j]				

(2) 构造具有两个内结点的最优二叉搜索树

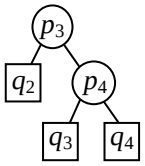
C[0][2] = 22
T[0][2] = 2



C[1][3] = 20
T[1][3] = 2

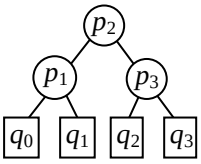


C[2][4] = 12
T[2][4] = 2

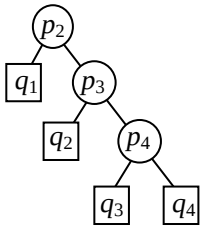


(3) 构造具有三个内结点的最优二叉搜索树

C[0][3] = 32
T[0][3] = 2

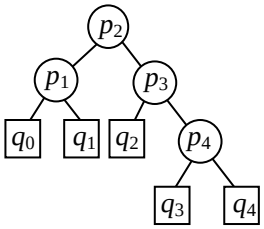


C[1][4] = 27
T[1][4] = 2



(4) 构造具有四个内结点的最优二叉搜索树

C[0][4] = 39
T[0][4] = 2



左子树 T[0][1], 其 C[0][1] = 7
右子树 T[2][4], 其 C[2][4] = 12

7-17 在二叉搜索树上删除一个有两个子女的结点时，可以采用以下三种方法：

(1) 用左子树 T_L 上具有最大关键码的结点 X 顶替，再递归地删除 X。

(2) 交替地用左子树 T_L 上具有最大关键码的结点和右子树 T_R 上具有最小关键码的结点顶替，再递归地删除适当的结点。

(3) 用左子树 T_L 上具有最大关键码的结点或者用右子树 T_R 上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

试编写程序实现这三个删除方法，并用实例说明哪一个方法最易于达到平衡化。

【解答】

① 在被删结点有两个子女时用左子树 T_L 中具最大关键码的结点顶替的算法：

```
template<class Type> BstNode<Type> * BST<Type> :: leftReplace ( BstNode<Type> * ptr ) {
    BstNode<Type> * temp = ptr->leftChild;                  //进到 ptr 的左子树
    while ( temp->rightChild != NULL ) temp = temp->rightChild;   //搜寻中序下最后一个结点
    ptr->data = temp->data;                                 //用该结点数据代替根结点数据
    return temp;
}
```

② 在被删结点有两个子女时用右子树 T_R 中具最小关键码的结点顶替的算法：

```
template<class Type> BstNode<Type> * BST<Type> :: rightReplace ( BstNode<Type> * ptr ) {
    BstNode<Type> * temp = ptr->rightChild;                 //进到 ptr 的右子树
```

```
    while ( temp->leftChild != NULL ) temp = temp->leftChild;       //搜寻中序下最后一个结点
    ptr->data = temp->data;                                         //用该结点数据代替根结点数据
    return temp;
}
```

(1) 用左子树 T_L 上具有最大关键码的结点 X 顶替，再递归地删除 X。

```
template <class Type> void BST<Type> :: Remove ( Type& x,   BstNode<Type> *& ptr ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild );                //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild );          //在右子树中执行删除
          else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                  // ptr 指示关键码为 x 的结点，它有两个子女
              temp = leftReplace ( ptr );         //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
              Remove ( ptr->data, temp );         //在 temp 为根的子树中删除该结点
            }
            else {                                // ptr 指示关键码为 x 的结点，它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;
            }
}
```

(2) 交替地用左子树 T_L 上具有最大关键码的结点和右子树 T_R 上具有最小关键码的结点顶替，再递归地删除适当的结点。

```
template <class Type> void BST<Type> :: Remove ( Type& x,   BstNode<Type> *& ptr,   int& dir ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。在参数表中有一个
//引用变量 dir, 作为调整方向的标记。若 dir = 0, 用左子树上具有最大关键码的结点顶替被删关键码; 若 dir = 1,
//用右子树上具有最小关键码的结点顶替被删关键码结点, 在调用它的程序中设定它的初始值为 0。
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild, dir );           //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild, dir );     //在右子树中执行删除
          else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                  // ptr 指示关键码为 x 的结点，它有两个子女
              if ( dir == 0 ) {
                  temp = leftReplace ( ptr );   dir = 1;        //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
              } else {
                  temp = rightReplace ( ptr );  dir = 0;        //在 ptr 的右子树中搜寻中序下第一个结点顶替 x
              }
              Remove ( ptr->data, temp, dir );                  //在 temp 为根的子树中删除该结点
            }
```

```
            else {                                          // ptr 指示关键码为 x 的结点，它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;
            }
    }
```

(3) 用左子树 T_L 上具有最大关键码的结点或者用右子树 T_R 上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

```
#include <stdlib.h>
template <class Type> void BST<Type> :: Remove ( Type& x,    BstNode<Type> *& ptr ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。在程序中用到一个
//随机数发生器 rand( ), 产生 0~32767 之间的随机数, 将它除以 16384, 得到 0~2 之间的浮点数。若其大于 1, 用左
//子树上具有最大关键码的结点顶替被删关键码; 若其小于或等于 1, 用右子树上具有最小关键码的结点顶替被删
//关键码结点, 在调用它的程序中设定它的初始值为 0。
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild );              //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild );        //在右子树中执行删除
          else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                          // ptr 指示关键码为 x 的结点，它有两个子女
              if ( (float) ( rand ( ) / 16384 ) > 1 ) {
                  temp = leftReplace ( ptr );   dir = 1;      //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
              } else {
                  temp = rightReplace ( ptr );   dir = 0;     //在 ptr 的右子树中搜寻中序下第一个结点顶替 x
              }
              Remove ( ptr->data, temp );                     //在 temp 为根的子树中删除该结点
          }
          else {                                          // ptr 指示关键码为 x 的结点，它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;
          }
}
```

7-18 (1) 设 T 是具有 n 个内结点的扩充二叉搜索树, I 是它的内路径长度, E 是它的外路径长度。试利用归纳法证明 $E = I + 2n, \quad n \geq 1$。

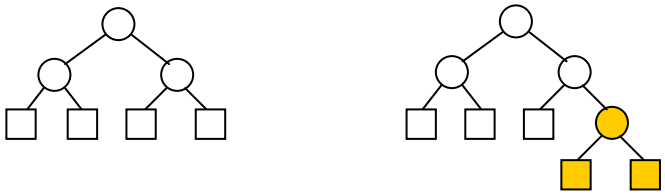
(2) 利用(1)的结果, 试说明: 成功搜索的平均搜索长度 S_n 与不成功搜索的平均搜索长度 U_n 之间的关系可用公式

$$S_n = (1 + 1/n) U_n - 1, n \geq 1$$

表示。

【解答】

(1) 用数学归纳法证明。当 n = 1 时，有 1 个内结点(I = 0)，2 个外结点(E = 2)，满足 E = I+2n。设 n = k 时结论成立，$E_k = I_k + 2k$。则当 n = k + 1 时，将增加一个层次为 l 的内结点，代替一个层次为 l 的外结点，同时在第 l+1 层增加 2 个外结点，则 $E_{k+1} = E_k - l + 2*(l+1) = E_k + l + 2$，$I_{k+1} = I_k + l$，将 $E_k = I_k + 2k$ 代入，有 $E_{k+1} = E_k + l + 2 = I_k + 2k + l + 2 = I_{k+1} + 2(k+1)$，结论得证。



(2) 因为搜索成功的平均搜索长度 S_n 与搜索不成功的平均搜索长度 U_n 分别为

$$S_n = \frac{1}{n}\sum_{i=1}^{n}(c_i + 1) = \frac{1}{n}\sum_{i=1}^{n} c_i + 1 = \frac{1}{n} I_n + 1$$

$$U_n = \frac{1}{n+1}\sum_{j=0}^{n} c_j = \frac{1}{n+1} E_k$$

其中，c_i 是各内结点所处层次，c_j 是各外结点所处层次。因此有

$(n+1)U_n = E_n = I_n + 2n = nS_n - n + 2n = nS_n + n$

$$\frac{n+1}{n} U_n = S_n + 1 \quad \Longrightarrow \quad S_n = \left(1 + \frac{1}{n}\right) U_n - 1$$

7-19 求最优二叉搜索树的算法的计算时间为 $O(n^3)$，下面给出一个求拟最优二叉搜索树的试探算法，可将计算时间降低到 $O(n\log_2 n)$。算法的思想是对于关键码序列{ $key_l, key_{l+1}, \cdots, key_h$ }， 轮流以 key_k 为根，k = l, l+1, …, h，求使得 | W[l−1][k−1] − W[k][h] | 达到最小的 k，用 key_k 作为由该序列构成的拟最优二叉搜索树的根。然后对以 key_k 为界的左子序列和右子序列，分别施行同样的操作，建立根 key_k 的左子树和右子树。要求：

(1) 使用 7.17 题的数据，执行这个试探算法建立拟最优二叉搜索树，该树建立的时间代价是多少？

(2) 编写一个函数，实现上述试探算法。要求该函数的时间复杂度应为 $O(n\log_2 n)$。

【解答】

(1) 各个关键码的权值为 { p_1, p_2, p_3, p_4 }，利用题 7-17 中的 W 矩阵，轮流让 k = 1, 2, 3, 4 为根，此时，下界 l = 1, 上界 h = 4。有

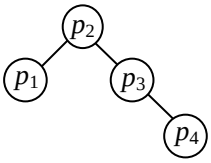
$$\min\left|W[l-1][k-1] - W[k][h]\right| = \left|W[0][1] - W[2][4]\right| = 2$$

求得 k = 2。则根结点为 2，左子树的权值为 { p_1 }，右子树的权值为 { p_3, p_4 }。

因为左子树只有一个结点，所以，权值为 p1 的关键码为左子树的根即可。 对于右子树 { p_3, p_4 }，采用上面的方法，轮流让 k = 3, 4 为根，此时，下界 l = 3，上界 h = 4。有

$$\min\left|W[l-1][k-1] - W[k][h]\right| = \left|W[2][2] - W[3][4]\right| = 1$$

求得 k = 3。于是以权值为 p_3 的关键码为根，其左子树为空，右子树为 { p_4 }。这样，得到拟最优二叉搜索树的结构如下：



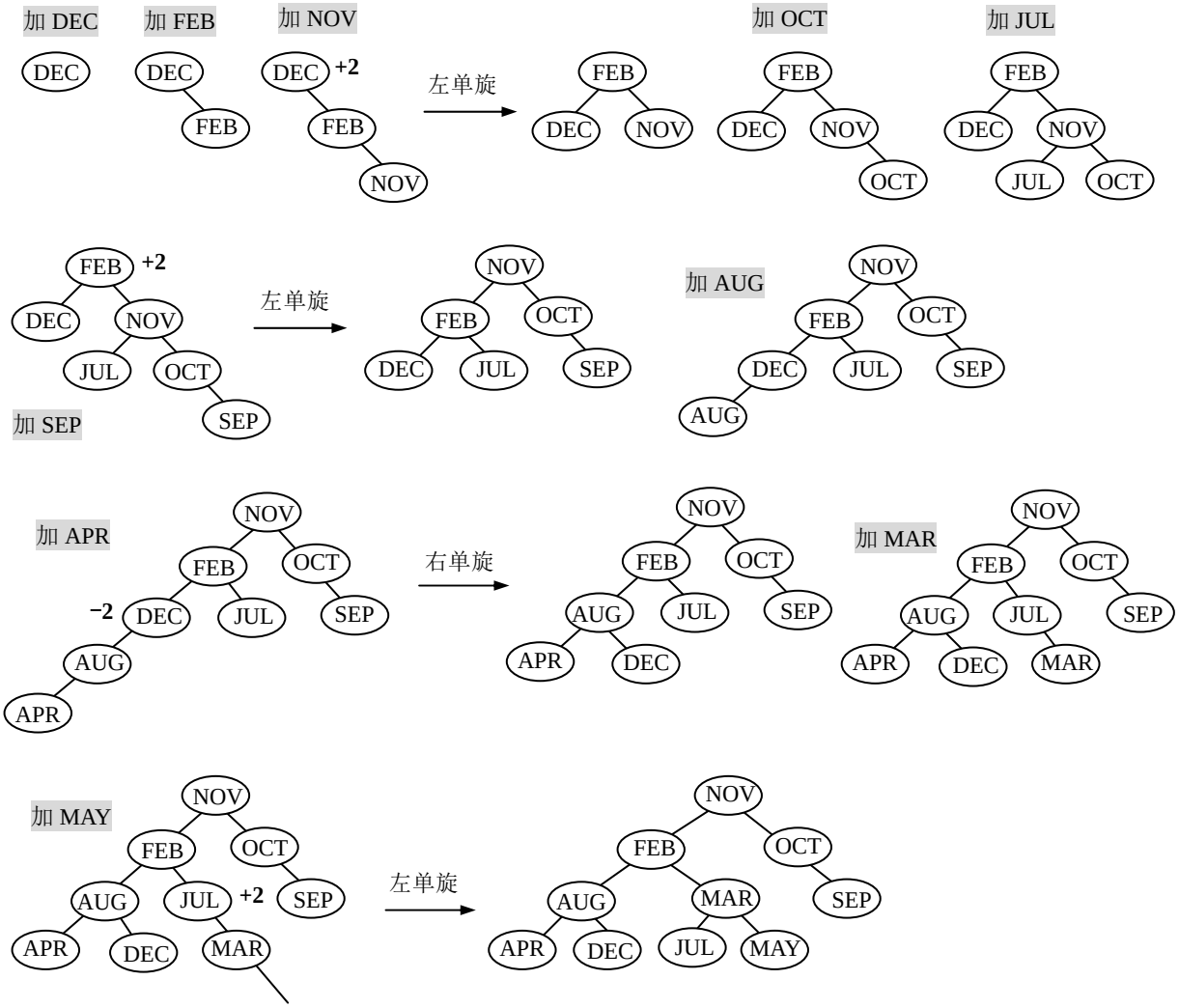
建立该拟最优二叉搜索树的时间代价为 O(4+1+2+1) = O(8)

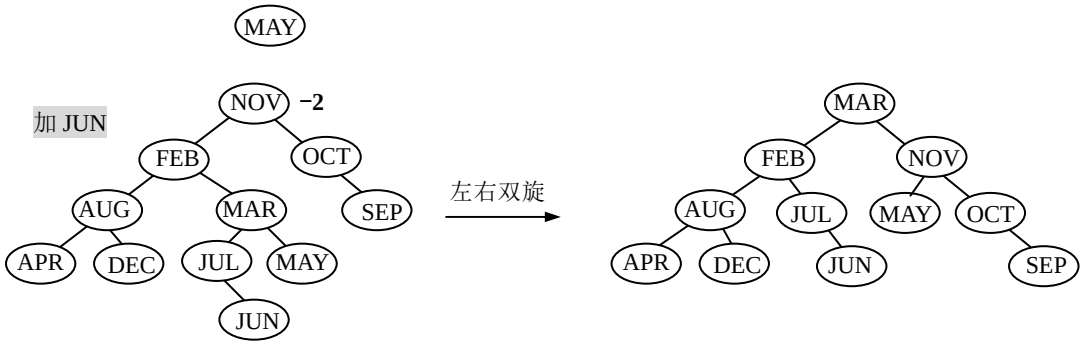
(2) 建立该拟最优二叉搜索树的算法

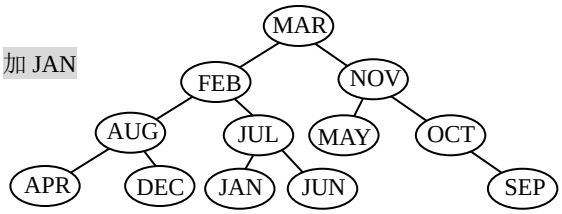
```
void nearOptiSrchTree ( int W[n+1][n+1], int n, int left, int right ) {
    if ( left > right ) { cout << "Empty Sequence! " << endl;    return; }
    if ( left == right ) { cout << left;    return; }
    int p = 0;    int k;
    for ( int j = left; j <= right; j++ )
        if ( p > abs ( W[left−1][j−1] – W[j][right] )
            { p = abs ( W[left−1][j−1] – W[j][right] );    k = j; }
    cout << k;
    if ( k == left ) nearOptiSrchTree ( W[n+1][n+1], n, k+1, right );
    else    if ( k == right ) nearOptiSrchTree ( W[n+1][n+1], n, left, k−1 );
        else { nearOptiSrchTree ( W[n+1][n+1], n, left, k−1 );
            nearOptiSrchTree ( W[n+1][n+1], n, k+1, right );
        }
}
```

7−20 将关键码 DEC, FEB, NOV, OCT, JUL, SEP, AUG, APR, MAR, MAY, JUN, JAN 依次插入到一棵初始为空的 AVL 树中，画出每插入一个关键码后的 AVL 树，并标明平衡旋转的类型。

【解答】



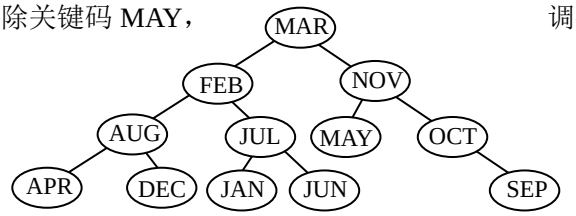




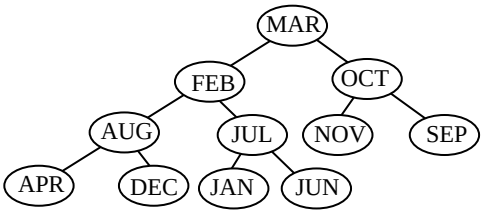
7−21 从第 7−20 题所建立的 AVL 树中删除关键码 MAY，为保持 AVL 树的特性，应如何进行删除和调整? 若接着删除关键码 FEB，又应如何删除与调整?

【解答】

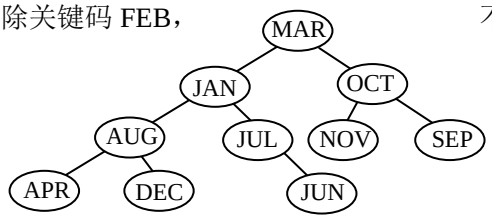
删除关键码 MAY，



调整



删除关键码 FEB，



不用调整

7−22 将关键码 1, 2, 3, …, 2^k-1 依次插入到一棵初始为空的 AVL 树中。试证明结果树是完全平衡的。

【解答】

所谓“完全平衡”是指所有叶结点处于树的同一层次上，并在该层是满的。此题可用数学归纳法证明。

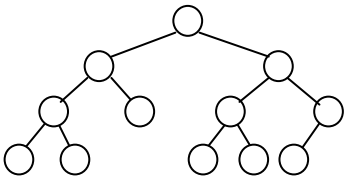
当 k = 1 时，$2^1-1 = 1$，AVL 树只有一个结点，它既是根又是叶并处在第 0 层，根据二叉树性质，应具有 $2^0 = 1$ 个结点。因此，满足完全平衡的要求。

设 k = n 时，插入关键码 1, 2, 3, …, 2^n-1 到 AVL 树中，恰好每一层(层次号码 i = 0, 1, …, n−1)有 2^i 个结点，根据二叉树性质，每一层达到最多结点个数，满足完全平衡要求。则当 k = n+1 时，插入关键码为 1, 2, 3, …, 2^n-1, 2^n, …, $2^{n+1}-1$，总共增加了从 2^n 到 $2^{n+1}-1$ 的 $2^{n+1}-1-2^n+1 = 2^n$ 个关键码，使得 AVL 树在新增的第 n 层具有 2^n 个结点，达到该层最多结点个数，因此，满足完全平衡要求。

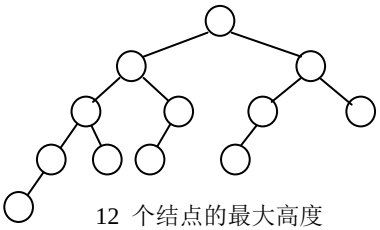
7−23 对于一个高度为 h 的 AVL 树，其最少结点数是多少？反之，对于一个有 n 个结点的 AVL 树，其最大高度是多少? 最小高度是多少?

【解答】

设高度为 h（空树的高度为 −1）的 AVL 树的最少结点数为 N_h，则 $N_h = F_{h+3} - 1$。F_h 是斐波那契数。又设 AVL 树有 n 个结点，则其最大高度不超过 $3 / 2 * \log_2(n + 1)$，最小高度为 $\lceil \log_2(n+1) \rceil - 1$。



12 个结点的最小高度



12 个结点的最大高度