第七章　集合与搜索

1、复习要点

集合是最基本的抽象数据类型之一。本章讨论了集合的三种存储表示：位数组表示、有序链表表示、并查集。在本章的后半部分，讨论了与集合相关的搜索方法和简单的性能分析方法，包括适用于静态搜索表的顺序搜索和折半搜索及代表动态搜索表的二叉搜索树和AVL树。可以使用扩充的二叉搜索树描述顺序搜索和折半搜索，从而推导出估算搜索效率的公式。静态搜索表在整个程序的运行期间结构不会变化，其搜索效率随着表中对象的个数 n 不断增长。动态搜索表因各个对象输入顺序不同，得到的搜索表的形态不同，典型的是二叉搜索树。在具有 n 个对象的二叉搜索树中，搜索效率最高的是高度最低的二叉搜索树。为确保二叉搜索树始终保持搜索效率最高，必须在输入新的对象时判断二叉搜索树是否“失去平衡”，并进行适当的平衡旋转，使二叉搜索树的高度降到最低。这就是 AVL 树，在 AVL 树的讨论中，四种平衡化旋转，选择参加平衡旋转的三个结点是关键，必须加以注意。

1.1 基本知识点

必须理解集合及其表示方法，包括位数组表示、有序链表表示及其相关操作的实现算法。理解并查集实现的方法。理解搜索的概念，理解静态搜索表结构，掌握静态搜索表的顺序搜索和折半搜索算法及其性能分析方法。掌握二叉搜索树的表示、搜索、插入、删除算法及其性能分析方法，掌握 AVL 树的构造、插入、删除时的调整方法及其性能分析，重点是 AVL 树的定义、平衡化旋转、AVL 树的插入和删除、AVL 树的高度。

1.2 算法设计

- ✧ 用有序链表表示集合时的求集合的并、交、差的算法；
- ✧ 并查集中的构造函数、求根及合并算法；
- ✧ 并查集中根据树的高度和根据树中结点个数进行合并的算法；
- ✧ 设置监视哨的顺序搜索算法和不设监视哨的顺序搜索算法；
- ✧ 有序顺序表的顺序搜索算法；
- ✧ 有序顺序表的折半搜索的递归算法和非递归算法；
- ✧ 二叉搜索树的搜索、插入和删除算法；
- ✧ 计算 AVL 树中指定结点高度的递归算法及利用此算法计算结点平衡因子的算法。

2、难点与重点

- ✧ 集合的概念——集合的基本运算、集合的存储表示
  - (1) 用位数组表示集合时集合基本运算的实现；
  - (2) 用有序链表表示集合时集合基本运算的实现。
- ✧ 并查集——并查集定义、并查集的三种基本运算的实现
- ✧ 基本搜索方法
  - (1) 对一般表的顺序搜索算法（包括有监视哨和无监视哨）；
  - (2) 对有序顺序表的顺序搜索方法，包括递归和非递归算法；
  - (3) 用判定树（即扩充二叉搜索树）描述有序顺序表的顺序搜索，以及平均搜索长度（成功与不成功）的计算；
  - (4) 对有序顺序表的折半搜索算法，包括递归和非递归算法；

(5) 用判定树（即扩充二叉搜索树）描述有序顺序表的折半搜索以及平均搜索长度（成功与不成功）的计算。

✧ 二叉搜索树

(1) 动态搜索树与静态搜索树的特性；
(2) 二叉搜索树的定义、二叉搜索树上的递归和非递归搜索算法；
(3) 二叉搜索树搜索时的平均搜索长度（成功与不成功）的计算；
(4) 二叉搜索树的插入与删除算法；
(5) AVL 树结点上的平衡因子、AVL 树的平衡旋转方法；
(6) 高度为 h 的 AVL 树上的最少结点个数与最多结点个数；
(7) AVL 树的搜索方法、插入与删除方法（不要求算法）。

3、重点概念分析

✧ **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。**

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。利用数学归纳法证明。

当 k=1 时，2^1-1=1，AVL 树只有一个结点，满足完全平衡的要求。

设当 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 的关键码。又因为每次都是在右下方插入，每当第 n+1 层出现新结点都会造成不平衡，需做左单旋，使其平衡化，最终使得 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。

✧ **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？**

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 $\lceil \log_2(n+1) \rceil - 1$。

4、重点习题解析

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 SubSetif(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 空否，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)
{
    if (s.IsEmpty( )= =0)
    { //集合元素不空
        if (s.Isempty( )= =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。

7-7 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;
template <class Type> class SetNode
{
    friend class SetList <Type>;
    private:
        Type data;
        SetNode <Type> *link;
    public:
        SetNode(const Type &item) : data(item), link(NULL);
};
```

```
template <class Type> class Set
{
    private:
        SetNode <Type> *first, *last;
    public:
        SetList( ) { first=last=new SetNode <Type> (0); }
        ~SetList( ) { MakeEmpty( ); delete first; }
        void MakeEmpty( );
        int AddMember(const Type &x);
        int DelMember(const Type &x);
        Set <Type> & operator = (Set <Type> &right);
        Set <Type> operator + (Set <Type> &right);
        Set <Type> operator * (Set <Type> &right);
        Set <Type> operator - (Set <Type> &right);
        int Contains(const Type &x);
        int operator = = (Set <Type> &right);
        Type & Min( );
};
template <class 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 中元素逐个拿出到 this 集合中查重
        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; }
    }
    pc->link=NULL; last=pc;  链表收尾
    return temp;
}
template <class 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
```

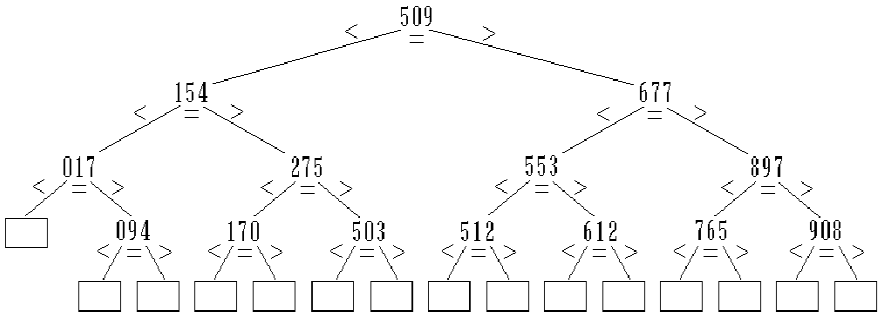
```
    Set <Type> temp;
    SetNode <Type> *pc=temp.first; //结果链的存放指针
    while (pb!=NULL)
    { //将集合 right 中元素逐个拿出到 this 中查重
        SetNode <Type> *pa=first->link;
        while (pa!=NULL)
        {
            if (pa->data= =pb->data)
            { pc->link=new SetNode <Type> (pa->data); pc=pc->link; }
            pa=pa->link;
        }
        pb=pb->link;
    }
    pc->link=NULL; last=pc;
    return temp;
}
template <class 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 中元素逐个拿出到 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;
}
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
}
template <class Type> int Set <Type> :: AddMember(const Type &x)
{ //把新元素 x 加入到集合之中，若集合中已有词元素，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;
    }
    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
    }
    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判定树，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。

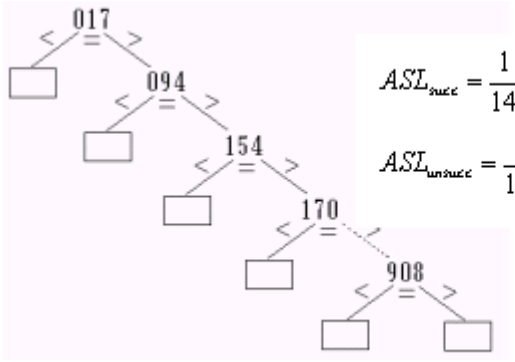
【解答】

描述对有序顺序表 017, 094, 154, 170, 275,503, 509, 512, 553, 612, 677, 765, 897, 908 进行折半查找的判定树为：



$$ASL_{succ} = \frac{1}{14}(1+2*2+3*4+4*7) = \frac{45}{14} \qquad ASL_{unsucc} = \frac{1}{15}(3+4*14) = \frac{59}{15}$$

对有序顺序表017, 094, 154, 170, 275,503, 509, 512, 553, 612, 677, 765, 897, 908进行顺序查找的判定树为：



$$ASL_{succ} = \frac{1}{14}\sum_{i=1}^{14} i = \frac{1}{14} * \frac{14*(1+14)}{2} = \frac{15}{2}$$

$$ASL_{unsucc} = \frac{1}{15}\left(\sum_{j=0}^{13}(j+1)+14\right) = \frac{1}{15}\left(\frac{14*15}{2}+14\right) = \frac{119}{15}$$

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，试就下列两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) 搜索失败时：

有序顺序表的平均搜索长度为(1+2+ ... + n + n)/ (n+1)= n/2+ n/(n+1)；

而无序表为 n ，两者不相同。

(2) 搜索成功且关键码唯一时：

有序顺序表的平均搜索长度为(1+2+ ... + n)/n=(n+1)/2；

而无序表此时与有序表相同，也为(n+1) / 2，两者相同。

(3) 搜索成功且关键码不唯一时：

设此时一共有 m 个关键码满足条件，则有序顺序表的平均搜索长度为(n+1)/2+m-1。其中，m-1 表示在第 1 个关键码找到后，其它的 m-1 个就是接下来的 m-1 个对象。无序表此时的平均搜索长度较难计算，因为它在找第 1 个关键码时也是平均比较(n+1)/2 次，但其它的 m-1 个一般不会是接下来的 m-1 个对象。

故在(3)这种情况下，无序表的平均搜索长度一般比有序表大，两种不相同。

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，返回接地址
    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 好 2 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：

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

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：

$S_1=\{15\}$, $S_2=\{25, 30, 35, 45\}$, $S_3=\{40, 50, 65, 70\}$

$c=40 \in S_3$, $b=45 \in S_2$, $b \le c$ 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：

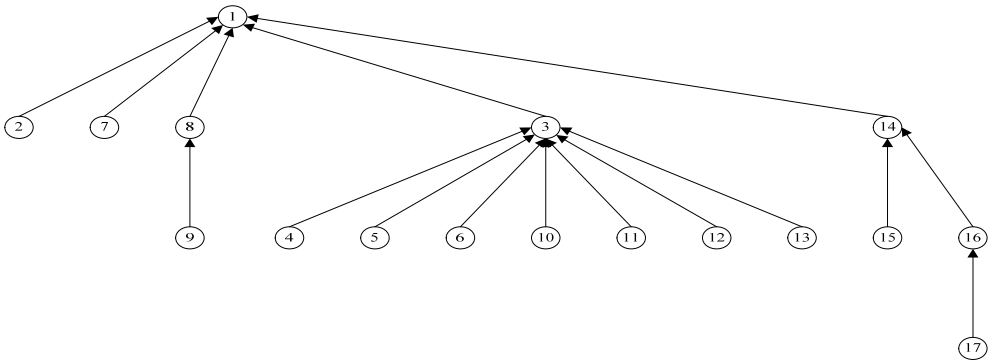
(1) 以任意方式执行 Union；

(2) 根据树的高度执行 Union；

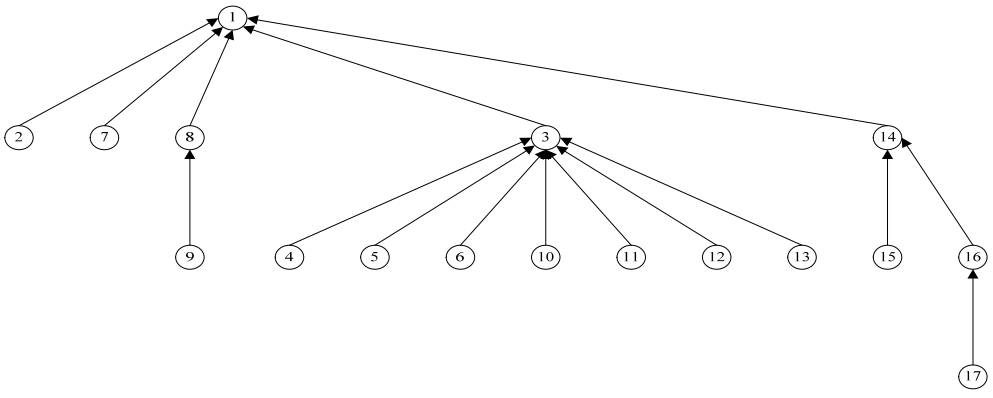
(3) 根据树中结点个数执行 Union。

【解答】

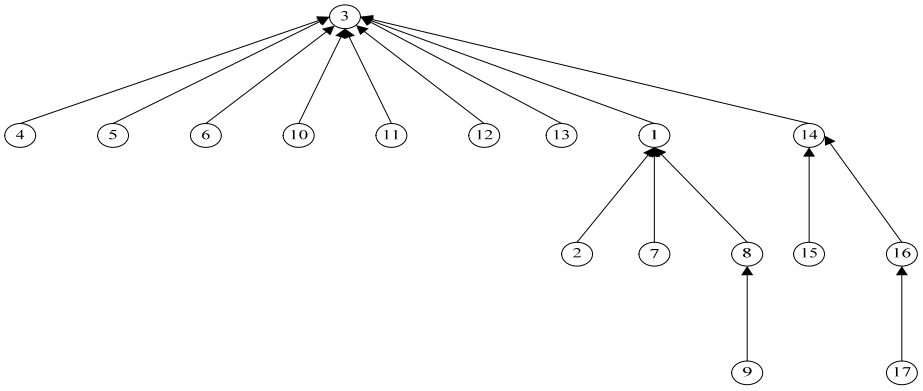
(1) 对于 Union(i, j)，以 i 作为 j 的双亲；



(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。



7-16 有 n 个结点的二叉搜索叔具有多少种不同形态？

【解答】

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

7-18 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 中具有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;
    //搜索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 中具有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;
    //搜索中序下第一个结点
    ptr->data=temp->data; //用该结点数据代替根结点数据
    return temp;
}
```

(3) 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 ptr 的左子树中删除该结点
        }
        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;
        }
    }
}
```

(4) 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，再递归地删除适当上的结点。

```
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
                Remove(ptr->data, temp, dir); //在 ptr 的左子树中删除该结点
            }
```

```
            else {
                temp=rightReplace(ptr); dir=0;
                //在 ptr 的右子树中搜寻中序下第一个顶点顶替 x
                Remove(ptr->data,temp, dir); //在 ptr 的右子树中删除该结点
            }
        }
        else { //pr 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;
        }
    }
}
```

(5) 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。

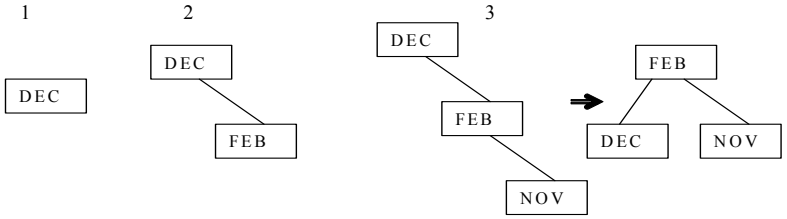
```
#include <stdlib.h>
template <class Type> void BST <Type> :: Remove(Type &x, BstNode <Type> * &ptr)
{ //私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点
  //若删除成功则新根通过 ptr 返回
  //在程序中用到一个随机数发生器 rand( )，产生 0~32767 之间的随机数
  //将它除以 16384，得到 0~2 之间的浮点数
  //若其大于 1，用左子树上具有最大关键码的结点顶替被删关键码
  //若其小于或等于 1，用右子树上具有最小关键码的结点顶替被删关键码结点
  //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);
                //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 ptr 的左子树中删除该结点
            }
            else {
                temp=rightReplace(ptr);
                //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 ptr 的右子树中删除该结点
            }
        }
```

```
            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;
            }
        }
    }
```

7-21 将关键字 DEC, FEB, NOV, OCT, JUL, SEP, AUG, APR, MAR, MAY, JUN, JAN 依次插入到初始为空的 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】

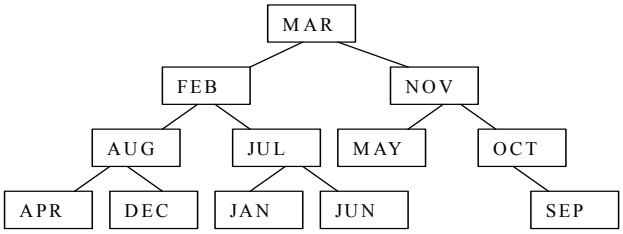
前三步如下图所示：



发生的平衡化旋转有 5 次：

(1) 插入 NOV 发生左单旋转；
(2) 插入 SEP 发生左单旋转；
(3) 插入 APR 发生右单旋转；
(4) 插入 MAY 发生左单旋转；
(5) 插入 JUN 发生先左后右旋转。

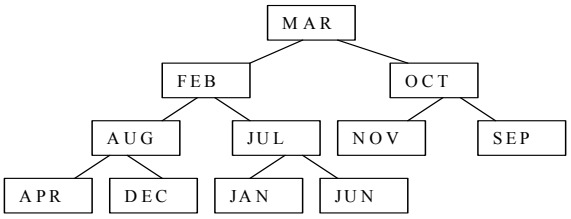
最后得到的 AVL 树如下所示：



7-22 从 7-21 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，进行调整如下：



删除关键码 FEB，不需要进行调整。