第 7 章 查找习题

7.1 若简单顺序查找算法所要查找的元素的下标从 0 开始，因而不能用监视哨，故查找失败时要返回-1。试设计相应的算法。

【解】假定从左往右顺序查找，成功返回元素下标，失败返回-1，算法如下：

```
int search( elementType A[], keyType x )
{
    int i;
    for( i=0; i<n; i++ )
    {
        if( A[i].key==x )
            return i;                //查找成功，返回元素下标 i
    }
        //循环结束没有返回，说明没有好到目标元素，返回查找失败标记
    return -1;
}
```

7.2 对有序数据表（5,7,9,12,15,18,20,22,25,30,100），按二分查找方法模拟查找元素 10 和 28，并分别画出其搜索过程。

【解】画出有序序列的二分查找判定树如图 7-1，10 和 28 均查找失败，查找过程见图中虚线标记。

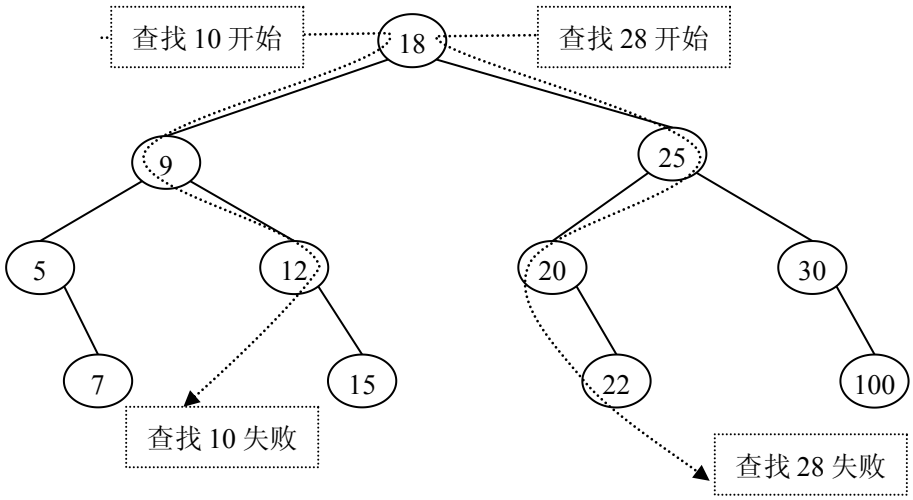


图 7-1 二分查找判定树查找 10、28 过程示意图

7.3 构造有 20 个元素的二分查找的判定树，并求解下列问题：

（1）各元素的查找长度最大是多少？

（2）查找长度为 1、2、3、4、5 的元素各有多少？具体是哪些元素？（假设下标从 0 开始）

（3）查找第 13 个元素依次要比较哪些元素？

【解】假定元素存储在 A[1...20]中，可画出其二分查找树如图 7-2 所示，图中结点数据为元素下标。

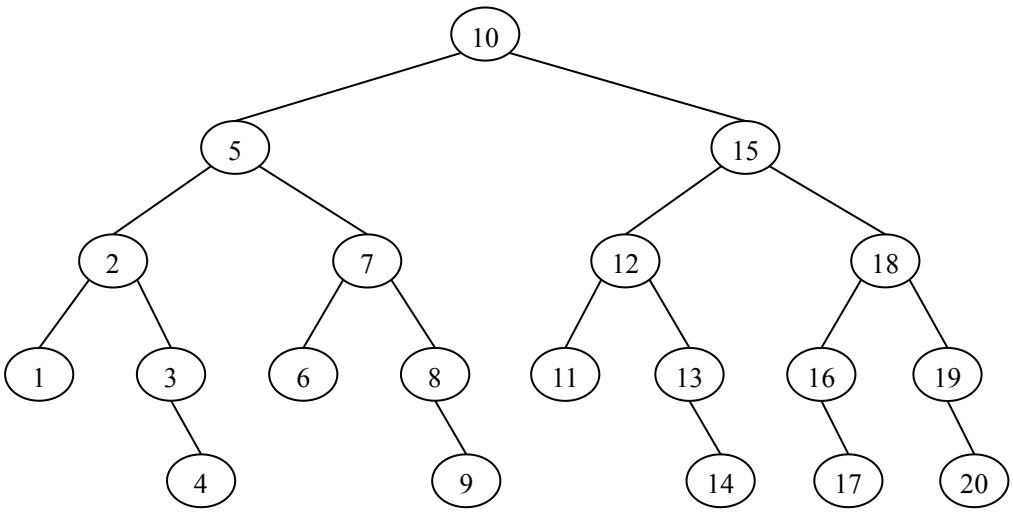


图 7-2A[1...20]序列的二分查找判定树

（1）由查找判定树可见各元素查找长度如表 7-1 所示：

表 7-1 序列 A[1...20]元素二分查找次数列表

查找长度	查找元素
1	A[10]
2	A[5]、A[15]
3	A[2]、A[7]、A[12]、A[18]
4	A[1]、A[3]、A[6]、A[8]、A[11]、A[13]、A[16]、A[19]
5	A[4]、A[9]、A[14]、A[17]、A[20]

（2）查找长度为 1、2、3、4、5 的元素见表 7-1。若下标从 0 开始，上述各元素下标减 1 即可。

（3）查找第 13 个元素依次要比较元素：A[10]->A[15]->A[12]->A[13]。

7.4 对有 n 个元素的有序表按二分查找方法查找时，最大的查找长度是多少？

【答】二分查找时元素的查找长度是元素在二分查找判定树上对应结点的层次，所以最大查找长度即为查找树的高度（深度）。而二分查找判定树除了最后

一层结点是满二叉树，求树的高度可沿用完全二叉树的方法为 $\lfloor \log_2 n \rfloor+1$ 。所以最大查找长度为 $\lfloor \log_2 n \rfloor+1$ 。

7.5 设计算法以构造有 n 个元素（下标范围从 1 到 n）的二分查找判定树。

【解】

【算法思想】

改造二叉树的一种遍历算法，设计访问根结点函数，根据当前数组的 low 和 high，计算 mid=(low+high)/2，申请结点 T，赋值 A[mid]，然后根据二分查找方式重新计算 low 和 high，递归创建 T 的左右子树。下面的算法改造先序遍历而成，二叉树采用二叉链表结构，初始化 low=1,high=n。

【算法描述】

```
void bstFromArr(btNode *&T,   elementType A[], int low,   int high)
{
    int mid;
    if(low<=high)
    {
        mid=(low+high)/2;          //计算子表中间元素下标
        T=new btNode;              //申请新结点并赋给数组中间元素值
        T->data=A[mid];
        T->lChild=NULL;
        T->rChild=NULL;

        bstFromArr(T->lChild, A, low, mid-1);     //递归创建 T 的左子树
        bstFromArr(T->rChild, A, mid+1, high);    //递归创建 T 的右子树

    }
}
```

用其他遍历方式创建二分查找判定树请读者自行完成。

这里的二分查找判定树采用的是二叉链表结构，也可以直接采用顺序存储方式（一维数组），但需要重新封装数组元素的成员，请读者自行完成。

7.6 判断题：若二叉树中每个结点的值均大于其左孩子的值，小于其右孩子的值，就一定是二叉排序树。

【答】不一定。这个条件不能确保左子树上所有结点的值都小于根结点，右子树上所有结点的值都大于等于根结点值，所以不一定是二叉排序树。这是大根堆的定义。举个例子如图 7-3 所示。

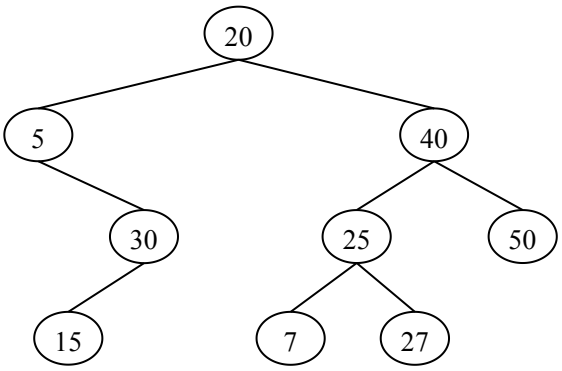


图 7-3 一棵非二叉排序树实例

7.7 设计算法，求出给定二叉排序树中值为最大的结点。

【解】

【算法思想】为中序遍历最后访问的结点，此结点右子树一定为空。如果右子树不存在即为根结点；右子树存在就在右子树上。算法从根结点开始一直寻找右分支，直到没有右子树的结点。算法返回目标结点指针。

【算法描述】

```
btNode* maxNode(btNode* T)
{
    btNode* p;
    p=T;                    //p 从根结点开始
    while(p->rChild)        //循环搜索右分支
    {
        p=p->rChild;
    }
    return p;               //p 的右子树为空。
}
```

7.8 设计算法，对给定的二叉排序树，求出在等概论情况下的平均查找长度。

【解】

【算法思想】二叉排序树中结点的查找长度就是此结点的层次。算法可以改造二叉树的一种遍历，统计所有结点的层次数之和 sum，再统计出结点总数 n，则 ASL=sum/n。

//以下算法，用中序遍历实现。lev 为结点层次值，n 统计结点数，sum 累计层次数。

【算法描述】

```
void inSum( btNode* T,   int lev,   int &n,   int &sum )
{
    if(T)
    {
```

```
            inSum(T->lChild, lev+1, n, sum);     //累计左子树
            n++;                    //累计结点数
            sum=sum+lev;       //累计结点层次数
            inSum(T->rChild, lev+1, n, sum);     //累计右子树
        }
    }
```

算法参数初始化：lev=1，n=0，wum=0。初始调用：inSum(T, 1, n , sum)

7.9 对给定的二叉树，假设其中各结点的值均不相同，设计算法以判断该二叉树是否是二叉排序树。

【解】

【算法思想】

利用一种二叉树遍历算法改造而成，分情况判定：

如果空树，返回 1；

如果只有根结点，没有左右子树，返回 1；

如果只有左子树，当 T->data<T->lChild->data 返回 0；否则递归判定左子树；

如果只有右子树，当 T->data>T->rChild->data 返回 0；否则递归判定右子树；

如果左右子树都存在，当 T->data<T->lChild->data 或者 T->data>T->rChild->data 返回 0；否则递归判定左子树和右子树。

【算法描述】

```
int isBst(btNode *T)
{
    if(T==NULL)        //空树，判定为真
        return 1;
    else if(T->lChild==NULL && T->rChild==NULL)     //只有根结点,判定为真
        return 1;
    else if(T->lChild && T->rChild==NULL)              //只有左子树
    {
        if(T->data<T->lChild->data)       //违反二叉排序树条件
            return 0;
        else
            return isBst(T->lChild);     //递归判定左子树
    }
    else if(T->lChild==NULL && T->rChild)              //只有右子树
    {
        if(T->data>T->rChild->data)       //违反二叉排序树条件
            return 0;
        else
            return isBst(T->rChild);     //递归判定右子树
```

```
    }
    else                                                    //左、右子树皆存在
    {
        if(T->data<T->lChild->data || T->data>T->rChild->data)
            return 0;                           //违反二叉排序树条件
        else
            return isBst(T->lChild) && isBst(T->rChild);
    }                                                       //递归判定左子树和右子树
}
```

本题的另一种解法：用数组保存二叉树的中序遍历序，再判定数组元素（中序序列）是否递增有序。

7.10 对给定的数组数据，其不同的输入序列是否一定可以构造出不同的二叉排序树？

【答】是的，同一组数据，输入次序不同，构造出的二叉排序树形态不同。

7.11 以数据集合{1,2,3,4,5,6}的不同序列为输入构造 5 棵高度为 6 的二叉排序树。

【解】只要每个结点只有单子树，构造出的二叉排序树高度即为 6。构造的高度为 6 的二叉排序树有很多棵，图 7-4 为其中 6 棵。

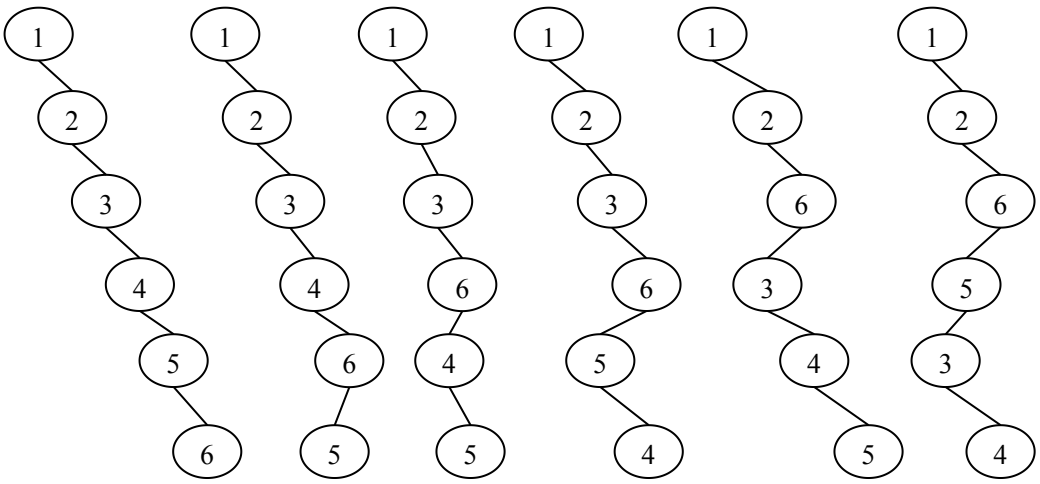


图 7-4 序列 12345 构造的高度为 6 的二叉排序树

7.12 已知一棵二叉排序树如下，其各结点的值虽然未知，但其中序序列为 1,2,3,4,5,6,7,8,9。请标注各结点的值。

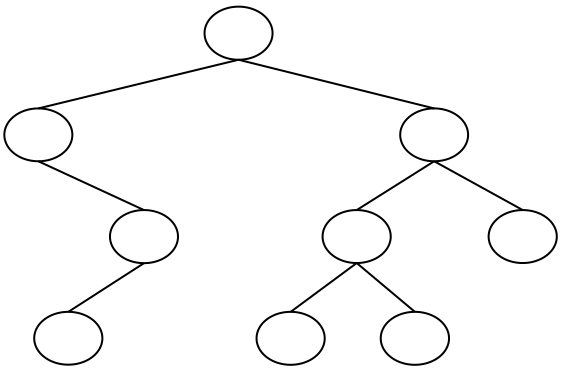


图 7-5 二叉排序树填值

【解】填法如下图。

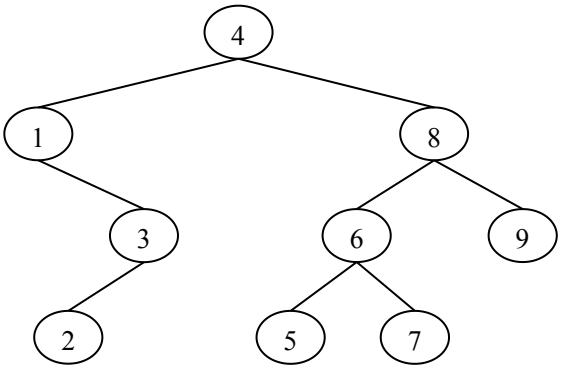


图 7-6 二叉排序树填值

7.13 已知散列表地址区间为 0～9,散列函数为 H(k)=k % 7,采用线性探测法处理冲突。将关键字序列 11,22,35,48,53,62,71,85 依次存储到散列表中，试构造出该散列表，并求出在等概论情况下的平均查找长度。

【解】散列表如下图：

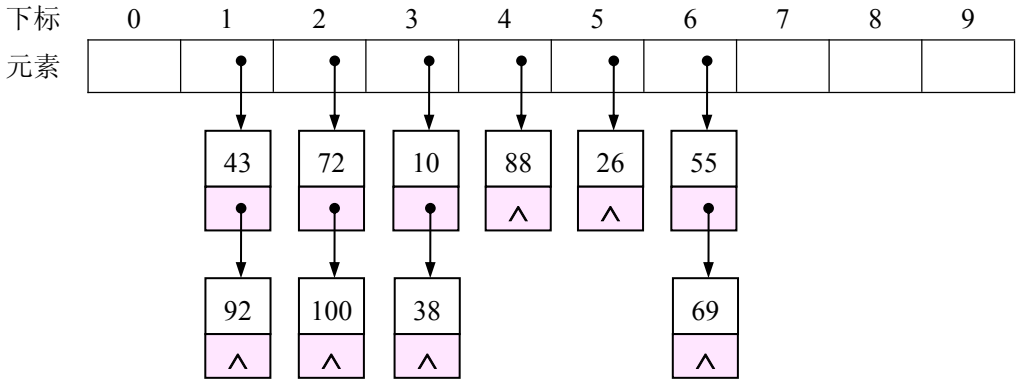
下标	0	1	2	3	4	5	6	7	8	9
元素	35	22	71	85	11	53	48	62		
次数	1	1	2	3	1	2	1	2		

平均查找长度 ASL=(1+1+2+3+1+2+1+2)/8=13/8。

7.14 设散列函数为 H(k)=k % 7，采用拉链法处理冲突，将关键字序列 10,26,38,43,55,69,72,88,100,92 依次存储到散列表中，并求出在等概论情况下的平均查找长度。

【解】拉链法构造散列表如下图。

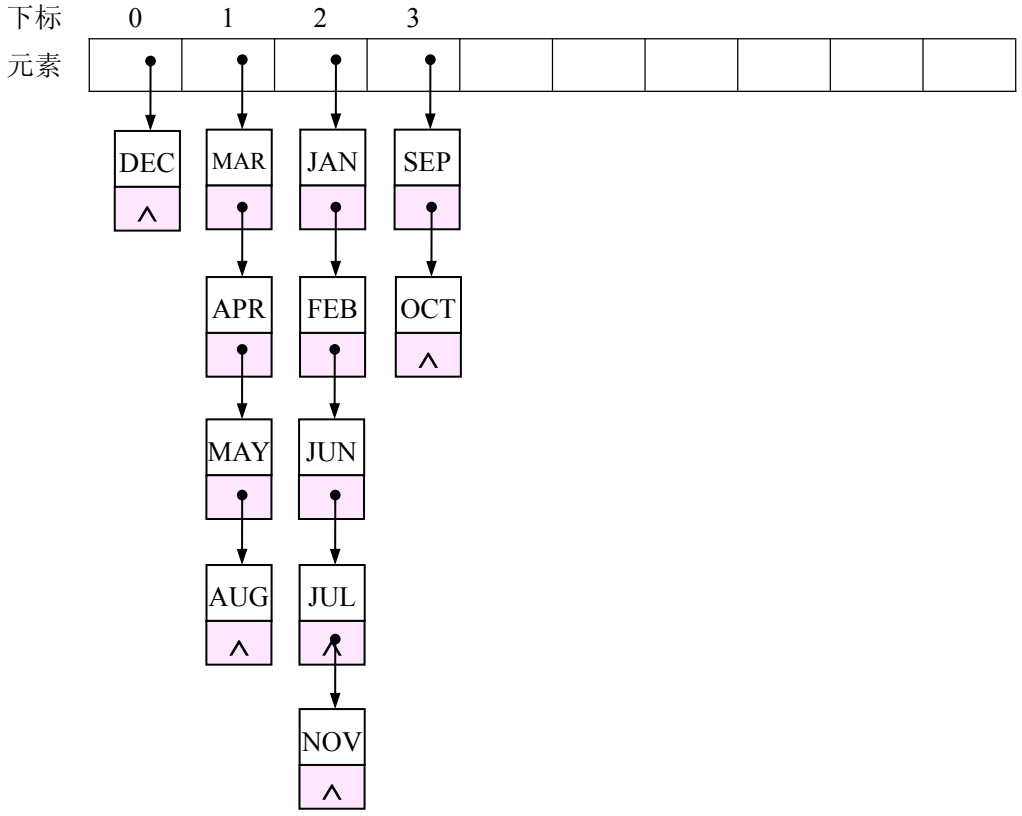
ASL=(6×1+4×2)/10=14/10。



7.16 设关键字序列为 JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC，散列函数为 H(k)=序号/4，其中序号指首字母在字母表中的序号，例如，字母 A 的序号为 1。采用拉链法处理冲突，构造出该散列表，并求出在等概论情况下的平均查找长度。

【解】拉链法构造散列表如下图。

ASL=(4×1+3×2+2×3+2×4+1×5)/12=29/10。



7.17 已知散列表的地址区间为 0～10，散列函数为 H(k)=k % 11，采用线性探测法处理冲突。设计算法在其中查找值为 x 的元素，若查找成功，返回其下标，否则返回－1。

【解】

```
int Search(elementType A[], keyType x)
{
    int p,h,i;
    int sn;          //保存查找次数


    if(n==0)              //n 为表中实际元素数
        return -1;     //空表，查找失败

    h=x%p;              //计算待查关键字 x 的哈希值，p=11

    if(A[h].key==x)
        return h;     //查找成功，返回元素下标
    else
    {
        i=(h+1)%MaxLen;     //从下一个元素开始线性探测查找，MaxLen=11
        while(i!=h)
        {
            if(A[i].key==x)   //探测查找成功，返回元素下标
                return i;
            i=(i+1)%MaxLen;          //否则，循环线性探测，MaxLen=11
        }
        return -1;       //元素不在表中
    }
}
```

7.18 已知散列表地址区间为 0～10，散列函数为 H(k)=k % 11，采用线性探测法处理冲突。设计算法将值为 x 的元素插入到表中。

【解】算法描述如下：

```
void Insert(elementType A[], keyType x)
{
    int h,p;
    if( n>=MaxLen) //n 为表 A[]中实际元素数，MaxLen 为表空间大小，本例 MaxLen=11
    {
        cout<<"哈希表空间满，插入元素失败。"<<endl;
        return;
```

```
        }
            //计算哈希值，本例 p=11
        h=x % p;
         while(n<MaxLen)     //线性探测插入元素
        {
            if(A[h].key==NULL)  //当前位置为空，存入元素，NULL 为空单元
标记
            {
                A[h].key=x;        //插入元素，退出探测
                n++;
                break;
            }
            h=(h+1)%MaxLen;        //线性循环探测插入位置
        }
    }
```

7.19 假设散列函数为 H(k)=k % 7，采用拉链法处理冲突。设计算法在其中查找值为 x 的元素。若查找成功，返回其所在结点的指针，否则返回 NULL。

【解】

【算法思想】

根据查找的元素值 x，利用 h=x%7，计算出哈希函数值 h，在头指针 p=A[h]的单链表中查找元素 x，成功返回 p，失败返回 NULL。

【算法描述】

```
    node* Search(node* A[], int k, elementType x )
    {          //k 为哈希函数除数值 k
        int h;
        node *p;

        if(listLen==0)
            return NULL;       //空表，查找失败

        h= x % k;                  //计算待查关键字 x 的哈希函数值

        p=A[h];                    //p 指向目标拉链首结点
        while(p)
        {
            if(p->data==x)
                return p;        //查找成功，返回目标结点指针 p
            else
                p=p->next;
        }
```

```
        return NULL;            //查找失败，元素不在表中，返回 NULL
    }
```

7.20*已知散列表地址区间为 0～10，散列函数为 H(k)=k % 11，采用线性探测法处理冲突。设计算法删除其中值为 x 的元素。

【解】

```
int DelHash(elementType A[], keyType x )
{
    int p,h,i;

    if(n==0)
        return -1;      //空表，删除失败

    h=x%p;                  //计算哈希值，p=11
    if(A[h].key==x)
    {
        A[h].key=NULL;     //标记删除
        n--;                //n 为表中实际元素数
        return   i;         //返回删除元素下标
    }
    else
    {
        i=(h+1)%MaxLen;     //从 h 下一个元素开始循环探测 x，MaxLen=11
        while(i!=h)
        {
            if(A[i].key==x)
            {
                A[i].key=NULL;   //标记删除
                n--;
                return i;           //返回删除元素下标
            }
            i=(i+1)%MaxLen;        //循环探测
        }
        return -1;                  //元素不再表中，删除失败
    }
}
```