第六章 二叉树的应用

习 题 六

一、填空题

1. 从二叉搜索树中查找一个元素时，其时间复杂度大致为 C 。

A $O(n)$　　B $O(1)$　　C $O(\log_2 n)$　　D $O(n^2)$

2. 向二叉搜索树中插入一个元素时，其时间复杂度大致为 B 。

A $O(1)$　　B $O(\log_2 n)$　　C $O(n)$　　D $O(n\log_2 n)$

3. 根据 n 个元素建立一棵二叉搜索树时，其时间复杂度大致为 D 。

A $O(n)$　　B $O(\log_2 n)$　　C $O(n^2)$　　D $O(n\log_2 n)$

4. 从堆中删除一个元素的时间复杂度为 C 。

A $O(1)$　　B $O(n)$　　C $O(\log_2 n)$　　D $O(n\log_2 n)$

5. 向堆中插入一个元素的时间复杂度为 A 。

A $O(\log_2 n)$　　B $O(n)$　　C $O(1)$　　D $O(n\log_2 n)$

6. 权值分别为 3，8，6，2，5 的叶子结点生成一棵哈夫曼树，它的带权路径长度为 D 。

A 24　　B 48　　C 72　　D 51

二、填空题

1. 在一棵二叉搜索树中，每个分支结点的左子树上所有结点的值一定 小于 该结点的值，右子树上所有结点的值一定 大于 该结点的值。

2. 对一棵二叉搜索树进行中序遍历时，得到结点序列是一个 有序序列 。

3. 从一棵二叉搜索树中查找一个元素时，若元素的值等于根结点的值，则表明 查找成功 ，若元素的值小于根结点的值，则继续向 左子树 查找，若元素的值大于根结点的值，则继续向 右子树 查找。

4. 在一个堆的顺序存储中，若一个元素的下标为 i，则它的左孩子元素的下标为 2i+1 ，右孩子元素的下标为 2i+2 。

5. 在一个小根堆中，堆顶结点的值是所有结点中的 最小值 ，在一个大根堆中，堆顶结点的值是所有结点中的 最大值 。

6. 当向一个小根堆插入一个具有最小值的元素时，该元素需要逐层 向上 调整，直到被调整到 堆顶 位置为止。

7. 当从一个小根堆中删除一个元素时，需要把_堆尾_元素填补到_堆顶_位置，然后再按条件把它逐层_向下_调整。

8. 在哈夫曼编码中，若编码长度只允许小于等于 4，则除了已对两个字符编码为 0 和 10 外，还可以最多对_4_个字符编码。

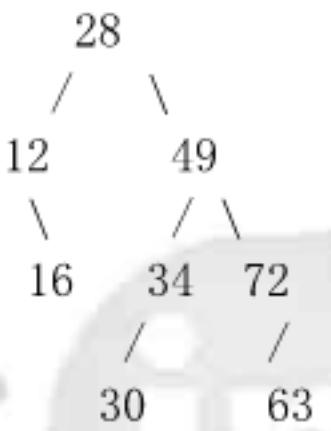
三、普通题

1. 已知一组元素(46, 25, 78, 62, 12, 37, 70, 29)，画出按元素排列顺序输入生成的一棵二叉树。

解：略

2. 已知一棵二叉排序树如图 6-11 所示，若从中依次删除 72, 12, 49, 28 结点，试分别画出每删除一个结点后得到的二叉排序树。

解：略



3. 设在一棵二叉排序树的每个结点中，含有关键字 key 域和统计相同关键字结点个数的 count 域，当向该树插入一个元素时，若树中已存在与该元素的关键字相同的结点，则就使该结点的 count 域增 1，否则就由该元素生成一个新结点而插入到树中，并使其 count 域置为 1，试按照这种插入要求编写一个算法。

解：

```
void Insert(BTreeNode*&BST,ElemType& item)
    //向二叉排序树中插入一个不重复元素 item，若树中存在该元素，则将一配结点值域中的
    //count 域的值加 1 即可
    {
    //从二叉排序树中查找关键字为 item.key 的结点，若查找成功则指针 t 指向该点结点，否则
    //指针 s 指向待插入新结点的双亲结点
    BTreeNode *t=BST,*S=NULL;
    while(t!=NULL){
        s=t;
        if(item.key==t->data.key)
        break;
        else if(item.key<t->data.key)
        t=t->left;
        else
        t=t->right;
```

```
            }
            //元素已存在，则将其值域中的 count 域的值增 1，否则建立新结点
并插入到合适位置
            if(t!=NULL)
             t->data.count++;
             else{
             BTreeNode* p=new BTreeNode;
             p->data=item;
             p->data.count=1;
             p->left=p->right=NULL;
             if(s==NULL)
                BST=p;
             else{
                  if(item.key<s->data.key)
                  s->left=p;
                  else
                  s->right=p;
                 }
        }
     }
```

4.编写一个非递归算法求出二叉排序树中的关键字最大的元素。

解：

```
        ElemType FindMax(BTreeNode* BST)
          //从二叉排序树中返回关键字最大的元素
         {
          if(BST==NULL){
          cerr<<"不能在空树上查找最大值!"<<endl;
          exit(1);
           }
          BTreeNode* t=BST;
           while(t->right!=NULL)
           t=t->right;
           return t->data;
          }
```

5.假定一棵二叉排序树被存储在具有 ABTList 数组类型的一个对象 BST 中，试编写出以下算法。

1.初始化对象 BST。

解：

```
        void InitBTree(ABTList BST)
          {
           //将树置空
           BST[0].left=0;
```

```
        //建立空闲链接表
        BST[0].right=1;
        for(int i=1;i<BTreeMaxSize-1;i++)
        BST[i].right=i+1;
        BST[BTreeMaxSize-1].right=0;
        }
```

2.向二叉树排序树中插入一个元素。

解：

```
    void Insert(ANTList BST,int&t,const ElemType& item)
     //向数组中的二叉排序树插入一个元素 item 的递归算法，变参 t 初始指向树根结点
    {
    if(t==0)//进行插入操作
     {
      //取出一个空闲结点
      int p=BST[0].right;
      if(p==0){
      cerr<<"数组空间用完!"<<endl;
      exit(1);
    }
    //修改空闲链表的表头指针，使之指向下一个空闲结点
    BST[0].right=BST[p].right;
    //生成新结点
    BST[p].data=item;
    BST[p].left=BST[p].right=0;
    //把新结点插入到确定的位置
    t=p;
    }
    else if(item.key<BST[t].data.key)
     Insert(BST,BST[t].left,item);
      //向左子树中插入元素
    else
     Insert(BST,BST[t].right,item);
      //向右子树中插入元素
     }
    void Insert(ABTList BST,const ElemType& item)
     //向数组中的二叉排序树插入一个元素 item 的非递归算法
    {
     //为新元素寻找插入位置
     int t=BST[0].left,parent=0;
     while(t!=0){
     parent=t;
     if(item.key<BST[t].data.key)
```

```
t=BST[t].left;
else
t=BST[t].right;
}
//由 item 生成新结点并修改空闲链表的表头指针
int p=BST[0].right;
if(p==0){
cerr<<"数组空间用完!"<<endl;
exit(1);
}
 BST[0].right=BST[p].right;
 BST[p].data=item;
 BST[p].left=BST[p].right=0;
 //将新结点插入到二叉排序树中的确定位置上
 if(parent==0)
 BST[o].left=p;
 else if(item.key<BST[parent].data.key)
 BST[parent].left=p;
 else
 BST[parent].right=p;
}
```

3.根据数组 a 中的 n 个元素建立二叉排序树。

解：

```
void CreateBSTree(ABTList BST,ElemType a[],int n)
   //利用数组中的元素建立二叉排序树的算法
{
  for(int i=0;i<n;i++)
  Insert(BST,BST[0].left,a[i]);
}
```

4.中序遍历二叉排序树。

解：

```
void Inorder(ABTList BST,int t)
  //对数组中的二叉树进行中序遍历的递归算法
{
  if(t!=0){
    Inorder(BST,BST[t].left);
    cout<<BST[t].data.key<<'';
    Inorder(BST,BST[t].right);
    }
 }
void Inorder(ABTList BST)
  //对数组中的二叉树进行中序遍历的非递归算法
```

```
{
   int s[10];//定义用于存储结点指针的栈
   int top=-1;  //定义栈顶指针并赋初值使 s 栈为空
   int p=BST[0].left;//定义指针 p 并使树根指针为它的初值
   while(top=-1||p!=0)
   {//当栈非空或 p 指针非 0 时执行循环
    while(p!=0){
    top++;
    s[top]=p;
    p=BST[p].left;
    }
    if(top!=-1){
     p=s[top];
     top--;
     cout<<BST[p].data.key<<'';
     p=BST[p].right;
      }
    }
   }
```

5.写出一个完整程序调用上述算法。

解：

```
#include<iostream.h>
#include<stdlib.h>
const int BTreeMaxSize=50;
struct student{
  int key;
  int rest;
};
typedef student ElemType;
 //定义二叉排序树中元素的类型为 student 记录型
struct ABTreeNode{//定义二叉排序树的结点类型
 ElemType data;
 int left,right;
};
typedef ABTreeNode ABTList[BTreeMaxSize];
   //定义保存二叉排序树的数组类型，各算法同上，在此省略
void main()
{
  student a[8]={{36},{54},{25},{30},{43},{18},{50},{28}};
  //为简单起见在每个元素中只给出关键字
  ABTList bst;
  InitBTree(bst);//初始化数组 bst
  //利用数组 a 在数组 bst 上建立一个二叉排序树
```

```
    CreateBSTree(bst,a,8);
    cout<<"中序:"<<endl;
    Inorder(bst,bst[0].left);
    cout<<endl;
  }
```

该程序的运行结果为：

```
中序:
  18 25 28 30 36 43 50 54
```