实验题第 1 题

第 1 小题

(1) 设计思路

本题要求实现查找：输入一个关键字，进行查找。

- 由于在一颗树中查找一个元素可以分解为在它的两棵子树中查找一个元素，而子树中又可以再分，所以可以采用分治法进行算法设计。设计思路如下：
  - 转化方法：把从一棵树中查找的问题转化为到它的子树中查找；
  - 迭代出口：p 迭代到 NULL 或等于 key 时终止迭代。
- 根据二叉排序树的特性，可以合理选择迭代时进入左子树还是右子树，减少空间占用，优化处理流程。

(2) 源代码

BTNode<DataType> *Search(DataType &key) const{}; **//查找接口函数**

```
template<class DataType>
BTNode<DataType>* BinarySortTree<DataType>::Search(DataType &key) const
{
    BTNode<DataType> *f;    //定义辅助指针 f
    return Search(key, f);  //进行查找
}
```

BTNode<DataType> Search(DataType &key, BTNode &f) const{}; **//内部辅助函数**

```
template<class DataType>
BTNode<DataType>* BinarySortTree<DataType>::Search(DataType &key, BTNode<DataType>*
&f) const
{
    BTNode<DataType> *p = this -> root;     //从根节点开始查找
    f = NULL;                               //初始化 f 为空指针
    while(p && p->data != key){             //p 不为空且 p 的数据域不等于给定关键字 key
        f = p;                              //把 p 的值赋给父指针 f
        if(key < p->data) p = p->lChild;    //如果 key 小于 p，往左子树查找
        else p = p->rChild;                 //如果 key 大于 p，往右子树查找
    }
    return p;                               //查找结束，返回 p
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得说明的是，在内部的 search 函数中设置了 f 指针，用来储存 p（目标节点）位置的父节点地址，以便在其他外部函数调用时（如 Insert）使用。即使最终没有找到 key 元素，f 的位置也是对应于查找元素的。

第 2 小题

(1) 设计思路

本题要求实现插入：输入一个关键字，进行插入。

- 由于 Search 函数可以获取到查找元素的上一次比较的节点地址（即父节点），可以直接利用这个节点的地址，进行一次比较，来插入元素。
- 根据以上思路，可以利用一个分支结构很容易地设计出如下程序。

(2) 源代码

bool Insert(DataType &e){}; //插入关键字

```
template<class DataType>
bool BinarySortTree<DataType>::Insert(DataType &e)
{
    BTNode<DataType> *f;                              //初始化父节点指针 f
    if(Search(e,f)) return false;                     //如果查找到已经存在，返回 false
    BTNode<DataType> *p = new BTNode<DataType>(e);    //为开辟一个 data 为 e 的节点，存入 p
    if(IsEmpty())                                     //如果是空树
        this -> root = p;                             //直接插入根节点位置
    else if (e < f->data)                             //如果 e 小于父节点 f
        f->lChild = p;                                //插入左孩子
    else                                              //如果 e 大于父节点 f
        f->rChild = p;                                //插入右孩子
    return true;                                      //返回成功 true
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，要考虑到空树的情况，此时 f 为 NULL，直接将开辟的 p 的地址存入 root 即可。

由于二叉排序树查找的程序设计属性，不会存在从中间插入的情况，search 函数会一直搜索到最底层的叶子节点为止。所以只需要考虑在叶子节点下直接插入，而不需要考虑在两个节点中间插入的情况。

第 3 小题

(1) 设计思路

本题要求实现删除：输入一个关键字，进行删除。

- 如果节点只有左子树或右子树，直接通过一个辅助节点进行删除和重接即可。
- 如果节点有左右子树，则存在两种删除方案，即用左子树的最大节点代替待删节点，或用右子树的最小节点代替待删节点。本处采用第一种方案，用左子树的最大节点替代后删除最大节点。

(2) 源代码

bool Delete(DataType &key){}; //删除关键字

```
template<class DataType>
bool BinarySortTree<DataType>::Delete(DataType &key)
{
    BTNode<DataType> *p, *f;                          //初始化 p 和 f 指针
    if ((p = Search(key,f)) == NULL) return false;    //如果没有找到此节点，返回 false
    if (f == NULL) {                                  //如果 f 为空，即 p 为根节点，直接删除根
        Delete(this->root);
    }
    else if (p == f->lChild){                         //如果 p 是 f 的左孩子，删除 f 的左孩子
        Delete(f->lChild);
    }
    else{                                             //如果 p 是 f 的右孩子，删除 f 的右孩子
        Delete(f->rChild);
    }
    return true;
}
```

void Delete(BTNode<DataType>* &p){}; //内部辅助函数

```
template<class DataType>
void BinarySortTree<DataType>::Delete(BTNode<DataType>* &p)
{
    BTNode<DataType>* tmpP,* tmpF;                //初始化两个辅助指针

    //叶子节点
    if(p->lChild == NULL && p->rChild == NULL)
    {
        delete p;             //释放 p
        p = NULL;             //地址赋为空
    }

    //仅有左子树
    else if(p->rChild == NULL)
    {
        tmpP = p;             //先把 p 存入 temp
        p = tmpP->lChild;     //把 p 的左子树接入 p
        delete tmpP;          //释放 p
    }
```

```
    //仅有右子树
    else if(p->lChild == NULL)
    {
        tmpP = p;            //先把 p 存入 temp
        p = tmpP->rChild;    //把 p 的右子树接入 p
        delete tmpP;         //释放 p
    }

    //有左、右子树
    else
    {
        tmpF = p;                       //先把 p 存入 tempF
        tmpP = p -> lChild;             //到左子树中寻找最大值
        while(tmpP -> rChild){          //定位到左子树的最右节点
            tmpF = tmpP;                //tmpP 存入 tmpF
            tmpP = tmpP -> rChild;      //tmpP 右孩子存入 tmpP
        }
        p -> data = tmpP -> data;       //把左子树最右节点数据存入 p
        if (tmpF -> rChild == tmpP)     //如果 tmpP 是右孩子
            Delete(tmpF -> rChild);     //递归删除左子树中的最大值
        else
            Delete(tmpF -> lChild);     //递归删除节点的左孩子
    }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，

- 如果待删除节点的左孩子没有右子树，即左孩子本身为左子树中最大节点，那么要删除的就不是左子树中的最右节点，而是左孩子本身。
- 删除过程中用 tmpF 实时存储上一层节点的信息，在进行删除时可以方便要做的操作。
- 在删除的主调函数中，需要传入 `f -> lChild` 或 `f -> rChild`，而不能直接删除 p，因为 p 是 f 的子节点，删除 p 之后 f 的子节点指针（f->lChild 或 f->rChild）就变成了悬空指针，父节点 f 并不再指向原来的子节点了，这会破坏二叉排序树的结构，并且在进行其他操作时造成一些不可预知的 bug。

第 4 小题

(1) 设计思路

本题要求编写递归算法，从大到小输出关键字不小于 x 的数据元素。

- 根据二叉排序树中序遍历序列的特点，先递归遍历右子树，再递归遍历左子树，就可以实现从大到小输出的需求。
- 把中序遍历中的执行操作写为打印即可。

(2) 源代码

void PrintNLT(BinarySortTree<DataType>& tree, BTNode<DataType>* node, const DataType& x){}; //用递归算法从大到小输出关键字不小于 x 的数据元素

```
template <class DataType>
void PrintNLT(BinarySortTree<DataType>& tree, BTNode<DataType>* node, const DataType&
x) {
    if (node == NULL) return;          //如果当前节点为空，直接返回终止
    PrintNLT(tree, node->rChild, x);  //遍历右子树
    if (node->data >= x) {             //如果当前节点的关键字不小于 x，输出该节点的值
        cout << node->data << " ";
    }
    PrintNLT(tree, node->lChild, x);  //遍历左子树
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，题目要求输出不小于给定值 x 的数据，所以在输出元素时需要进行一次条件判断。

思考：题目给出的函数是 `PrintNLT(tree, tree.GetRoot(), data);`，此处的 tree 应该是可以省略的，通过 tree.GetRoot()就已经可以获取到树的地址信息了。