华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析　　**年级：**22 级　　**上机实践成绩：**

指导教师：金澈清　　**姓名：**郭夏辉

上机实践名称：二叉搜索树　　**学号：**10211900416　　**上机实践日期：**2023 年 4 月 20 日

上机实践编号：No.8　　**组号：**1-416

一、目的

1．熟悉算法设计的基本思想

2．掌握二叉树搜索的基本思想，并且能够分析算法性能

二、内容与设计思想

1. 编程实现二叉搜索树，能根据给定字符串（50,20,80,null,null,60,90,null,null,null,100）构建对应二叉搜索树，实现二叉搜索树的各种操作，包括先序遍历、取最小值、取最大值、搜索节点、获取前驱节点以及获取后驱节点。

2. 根据二叉搜索树的中序遍历，输出二叉搜索树的先序和后序遍历结果。

3. 选择合适的数据规模，计算二叉搜索树在不同数据量下查询操作的所需时间。

4. 思考题：对比二叉搜索树、顺序访问和散列表的性能。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

1. *写出算法的源代码：*

2. *以合适的图来表示你的实验数据*

二叉搜索树的定义和性质

一棵二叉搜索树是以二叉树的形式来组织的，可以用类似于链表的方式来组织，它的每个结点一般包含的信息包括：

```
struct Node {
    Node *left;
    Node *right;
    int val;
};
```

二叉搜索树需要满足以下的性质:1. 左子树非空时，左子树所有结点的 key，均小于根结点的 key;2. 右子树非空时，右子树所有结点的 key，均大于根结点的 key;3. 任意结点的左右子树也是一颗二叉搜索树;4. 没有结点值相等的.

二叉搜索树的三种经典的遍历方式

二叉搜索树性质允许我们通过一个简单的递归算法来按序输出二叉搜索树中的所有关键字，这种算法称为中序遍历(inorder tree walk)**算法。这样命名的原因是输出的子树根的关键字位于其左子树的关键字值和右子树的关键字值之间。(类似地，先序遍历**(preorder tree walk)**中输出的根的关键字在其左右子树的关键字值之前，而后序遍历**(postorder tree walk)**输出的根的关键字在其左右子树的关键字值之后。)调用下面的过程 INORDER-TREE-WALK(*T*.*root*)，就可以输出一棵二叉搜索树 *T* 中的所有元素。**

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x,left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

三种遍历的不同点在于打印根节点值，递归遍历左子树和右子树的顺序不同。在本次实验中，将 50,20,80,null,null,60,90,null,null,null,100 插入二叉搜索树后的三种遍历结果如下图所示:

```
50,20,80,null,null,60,90,null,null,null,100
PreOrder:50 20 80 60 90 100
InOrder:20 50 60 80 90 100
PostOrder:20 60 100 90 80 50
```

二叉搜索树的最小值

根据 BST 的性质，左子树的值均小于根，因此搜索最小值时只需要不停地沿左子树递归即可，找到最后一个不为 NULL 的结点便是二叉搜索树的最小值。

二叉搜索树的最大值

根据 BST 的性质，右子树的值均大于根，因此搜索最大值时只需要不停地沿右子树递归即可，找到最后一个不为 NULL 的结点便是二叉搜索树的最大值。

二叉搜索树的结点搜索

这其实是一个类似于二分查找的操作，因为 BST 的左子树值均小于根结点的值而右子树的值均大于根节点的值，所以为了搜索对应的结点，我们从整个 BST 的根节点开始，不断迭代，如果当前结点值为目标值，则直接返回，如果当前节点值小于目标值，则去右树中迭代寻找，大于目标值则去左树迭代寻找。

BST 的层数越少，查找时间越少，但是当数据增多时 BST 并不能保证树的高度一直是最优的，甚至在极端情况下 ：一个已经有序的数列会导致 BST 退化成一个单链表.

二叉搜索树获得前驱结点

一个二叉树中序遍历中某个节点的前一个节点叫该节点的前驱节点。这个还比较有意思，我结合的是 BST 的性质，而不是利用类似深度优先搜索的方式去搜索。若该节点存在左子树，则其前驱节点为左子树最右边的节点；若该节点不存在左子树，则利用 parent 指针向父节点找，若满足该节点是其父节点的右节点，则该父节点为当前节点的前驱节点，若不满足则更新父节点为祖父节点，当前节点更新为其父节点，直到满足条件或者父节点为空，为空表示到达根节点依旧没有找到。这个前面一种情况是比较好理解的，但是后面一种情况呢？通过画图找规律、归纳，我发现如果 x 没有左孩子，则 x 有以下两种可能:1. x 是一个"右孩子"，则 x 的前驱结点便是它的父结点;2. x 是一个"左孩子"，则查找 x 的最低的父结点，并且该父结点要具有右孩子，找到的这个"最低的父结点"就是 x 的前驱结点。(要找前驱元素,肯定是找一个有右孩子的父节点,并且这个父节点的右孩子中一定能找到当前节点.)

二叉搜索树获得后驱结点

一个二叉树中序遍历中某个节点的后一个节点叫该节点的后驱节点。若该节点存在右子树，则其后继节点为右子树最左边的节点；若该节点不存在右子树，则利用 parent 指针向父节点找，若满足该节点是其父节点的左节点，则该父节点为当前节点的后继节点，若不满足则更新父节点为祖父节点，当前节点更新为其父节点，直到满足条件或者父亲节点为空，为空表示到达根节点依旧没有找到。这个的理解类似于上面对前驱结点的理解。

```
Node *GetPrev(Node *node){
    Node *ans=nullptr;
    if(node->left!=nullptr){
        Node *r = node->left;
        while(r->right!=nullptr)r=r->right;
        ans=r;
    }
    else{
        ans=node->parent;
        while((ans!=nullptr) && (node==ans->left)){
            node=ans;
            ans=ans->parent;
        }
    }
    return ans;
}
```

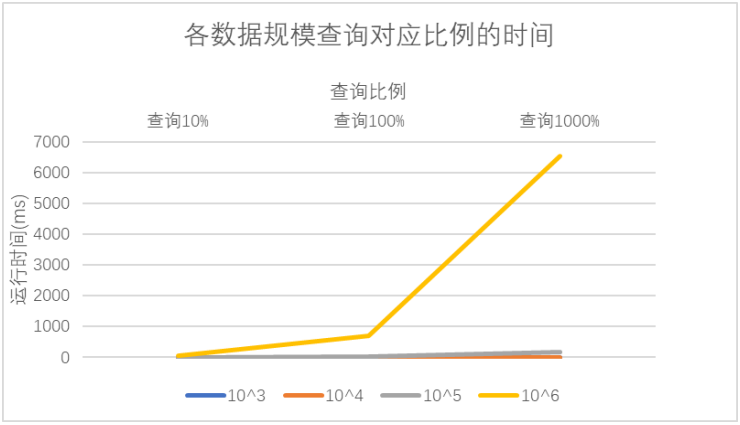
```
Node *GetSucc(Node *node){
    Node *ans=nullptr;
    if(node->right!=nullptr){
        Node *l = node->right;
        while(l->left!=nullptr)l=l->left;
        ans=l;
    }
    else{
        ans=node->parent;
        while((ans!=nullptr) && (node==ans->right)){
            node=ans;
            ans=ans->parent;
        }
    }
    return ans;
}
```

五、总结

对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。

本次实验的难度不算大，整体来说比较基础，然后我在 10^3-10^6 插入数据规模之下，每个数据规模分别查询 10%，100%，1000%的数，得到的运行时间如下图所示：

	查询 10%	查询 100%	查询 1000%
10^3	0	0	0
10^4	0	1	8
10^5	2	28	184
10^6	69	703	6521



可以看到随着数据规模的扩大，二叉搜索树的查询时间越来越大。究其本质，还在于数据规模扩大时二叉树的深度明显增大，这样在搜索结点时的迭代次数就会对应地变大，降低了查询的效率。比较引起我们重视的是普通的 BST 深度的增大是不可控制的，对于最差的情况甚至会退化为一个单链表，日后要学习的红黑树、AVL 树等平衡二叉搜索树的本质在于通过适当的左旋、右旋等操作使树的深度被控制在对数范围内，大幅提高了运行效率。

思考题：对比二叉搜索树、顺序访问和散列表的性能。

二叉搜索树:在最好情况下插入、查询和删除操作的时间复杂度都是对数级的Θ(lgn).二叉搜索树各个操作的时间复杂度取决于树的深度，如果没有红黑树这样的优化措施，不同的插入顺序可能会得到完全不同形态的树状结构——对于一个已经有序的序列的插入会退化成一个单链表。但是二叉搜索树最好的特征是有序性，我们不需要像顺序访问和散列表这种无序表一样先把数据取出来，再排序才能得到有序的序列。

顺序访问:插入的时间复杂度是Θ(1)，查询和删除的时间复杂度是线性的，即Θ(n)。对于新元素的插入，顺序访问只需要把新元素放到这个数据结构的末端就行了，常数级的复杂度效率很高。由于查询和删除操作都需要一个一个地从头往后找，直到找到对应元素，所以这两种操作的时间复杂度是Θ(n)，效率过低，应用价值比较小。

散列表:在理想情况下，插入、查询和删除操作的时间复杂度都是常数级Θ(1).在实际运用过程中可能会遇到哈希冲突这样的问题，使得散列表并不能像预期的那样总是Θ(1)级别复杂度。并且我们在实际使用散列表的过程中，要恰当地构造散列函数，使得数据尽可能均匀地分布于整个表中；与此同时，我们还要确保装载因子维持在一个理想的情况下——太大效率会显著降低；太小则空间大量浪费。

三种不同的数据结构有它们各自的特点，根据我们的需求进行选择，相信我们能最大化它们的价值！