数据结构与算法I 实验5

2019201409 于倬浩

一、实验内容

实现二叉搜索树

二、实现操作&接口

首先，对于节点数据类型，定义如下：

```
struct node{
    node *c[2], *p; //左右儿子、父节点
    int sz;         //当前节点的子树大小
    int val;        //当前节点的键值
    node() {
        c[0] = c[1] = p = null;
        sz = 1;
    }
    inline void pushup() { //修改当前节点子节点信息后，维护当前节点信息
        if(this == null) return;
        sz = c[0]->sz + c[1]->sz + 1;
    }
};
```

实现的操作如下：

```
inline node* kth(node *x, int k);
//在以x为根的树中，寻找键值排名为k的节点，返回指向该节点的指针，找不到则返回空节点。

inline node* find(node *x, int val);
//在以x为根的树中，寻找键值为val的节点，返回指向该节点的指针，找不到则返回空节点。

inline int rank(node *x, int val);
```

```
//计算以x为根的树中，键值val的排名。

inline void insert(node *&x, int val);
//在以x为根的树中，插入键值为val的节点。

inline void erase(node *&x, int val);
//在以x为根的树中，删除键值为val的节点。

inline node* predecessor(node *x);
//返回x节点的前驱

inline node* successor(node *x);
//返回x节点的后继
```

对于predecessor操作，如果当前节点存在左子树，那么就迭代找到左子树中最深的右儿子。否则，

三、性能分析

由于本次实现的二叉搜索树实际上不会保证平衡的性质，而各操作的运行时间又完全取决于树的高度，因此各个操作的时间复杂度为$\Theta(h)$。因此最坏情况下，各种操作的时间复杂度均为$\Theta(n)$。

由于支持操作有限，测试结果并不具有很好的代表性，且性能较差（与线性查找算法时间复杂度相同），因此不做对比测试，仅从源代码层面分析正确性。

核心操作：插入/删除，具体请见如下代码和注释。

```
inline void insert(node *&x, int val) { //在以x为根的子树中插入键值为val的节点。
    if(x == null) {
        //已经遍历到哨兵节点，该位置即为待插入的位置，因此直接插入
        x = new node;
        x -> val = val;
        return;
    }
    if(val <= x -> val) //待插入值在当前节点的左子树，递归进入返回后，维护子节点的父亲节点。
        insert(x->c[0], val), x->c[0]->p = x;
    else //待插入节点在当前节点的右子树。
        insert(x->c[1], val), x->c[1]->p = x;
    x->pushup(); //完成子树的处理后，以O(1)代价维护当前节点信息（如子树大小）。
    return;
}
```

```cpp
#define isrc(x) ((x)->p->c[1] == (x))
inline void erase(node *&x, int val) { //从以x为根的子树中删除值为val的节点
    node *t = find(x, val);
    //利用前述的find操作，寻找值为val的节点。
    if(t->c[0] == null && t->c[1] == null) {
        //t为叶子节点，直接删除，并维护父亲节点的子树信息。
        if(t == x) x = null;
        else {
            t->p->c[isrc(t)] = null;
            t->p->pushup();
        }
        node *cur = t->p;
        //维护当前节点到根一条链上的节点信息
        while(cur != null) cur->pushup(), cur = cur->p;
        delete t;
    }
    else if(t->c[0] == null || t->c[1] == null) { //t只有一个儿子。
        bool d = (t->c[1] == null) ^ 1; //确定非空的儿子
        if(t == x) { //分类讨论当前节点是否为根
            x = t->c[d];
            t->c[d]->p = null;
            t->c[d]->pushup();
        }
        else {
            t->p->c[isrc(t)] = t->c[d];
            t->p->pushup();
            t->c[d]->p = t->p;
        }
        node *cur = t->p;
        while(cur != null) cur->pushup(), cur = cur->p;
        delete t;
    }
    else {//t有两个儿子
        node *v = t->c[1];//找t的后继
        while(v->c[0] != null) v = v->c[0];
        v->p->c[0] = v->c[1];
        v->p->pushup();
        if(v->c[1] != null) v->c[1]->p = v->p;
        t->val = v->val;//用找到的后继替换t
        node *cur = v->p;
        while(cur != null) cur->pushup(), cur = cur->p;
        delete v;
    }
}
```

其余操作较为简单，限于篇幅，完整代码及完整注释见bst.hpp