08B3 删除

#数据结构邓神

框架


```
// remove
template <typename T> bool BST<T>::remove(const T & e) {
    BinNodePosi<T> & x = search(e);
    if (!x){
        return false;
    }
    removeAt(x,_hot);
    _size--;
    updateHeightAbove(x);
    return true;
}
```

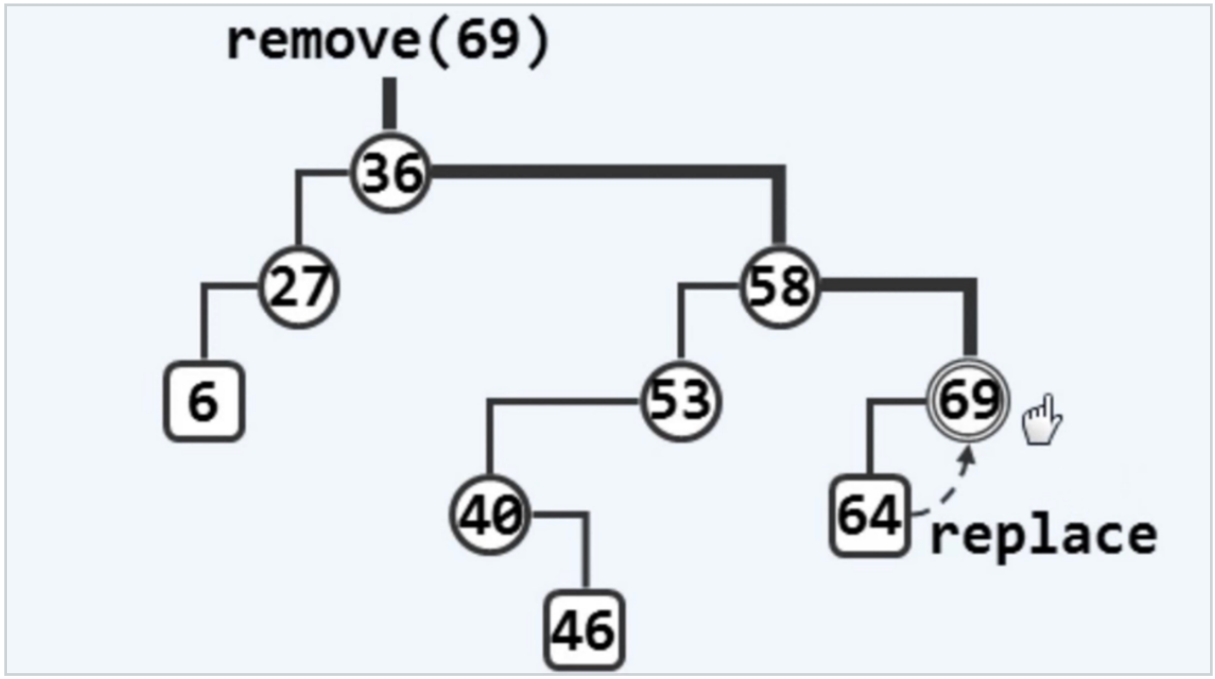
时间消耗主要集中于 removeAt updateHeightAbove search

前面可以知道 后两者时间为 O(h)

单分支


删除：情况一
若*x（69）的某一子树为空
则可将其替换为另一子树（64）

例子：

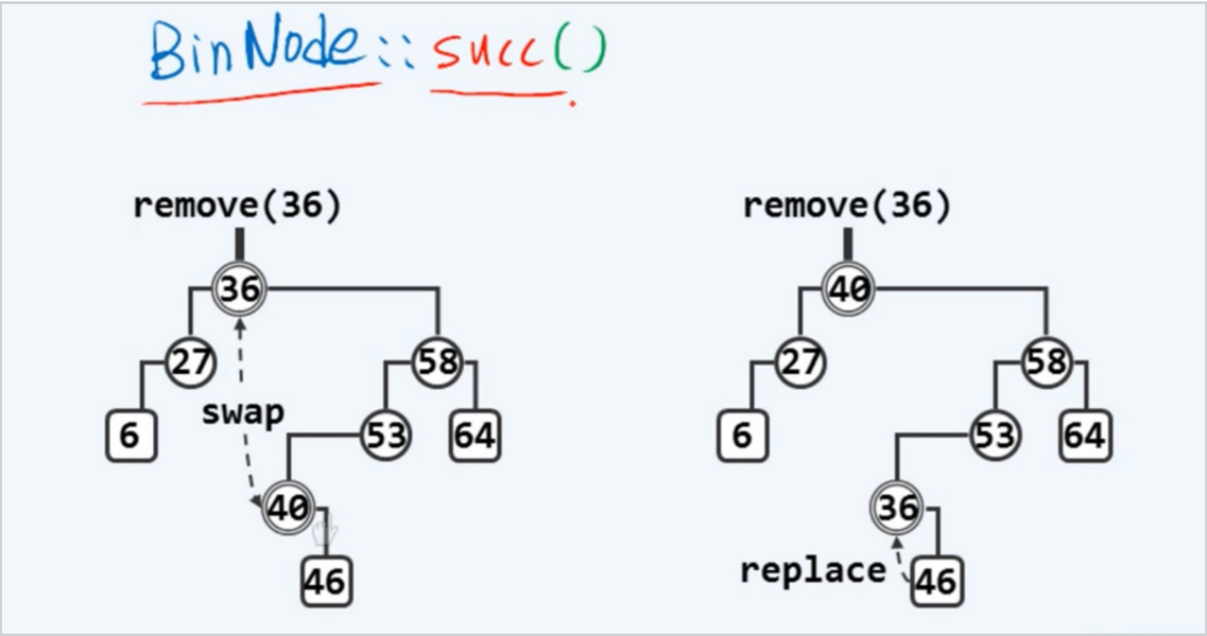

remove(69)
36
27
58
6
53
69
40
64
replace
46

实现


```
template <typename T> static BinNodePosi<T> removeAt(BinNodePosi<T> &
x,BinNodePosi<T> & hot){
    BinNodePosi<T> w = x;
    BinNodePosi<T> succ = nullptr;
    if (!HasLChild(*x)){
        succ = x = x->rChild;
    }
    else if (HasRChild(*x)){
        succ = x = x->lChild;
    }
    else {
        /*左右子树都有*/
    }
    hot = w->parent;
    if (succ) {
        succ->parent = hot;
    }
    release(w->data);
    release(w);
    return succ;
}
```

双分支

化繁为简



在树中的直接后继为在中序遍历下的直接后继

36的直接后继也就是40

也就是不小于当前节点的最小的一个节点

将他们当前节点与他的直接后继直接交换，交换后暂时的将不会成为一颗BST，我们只需要将交换后的那个要删除的节点直接删除

Wooooooooooooooooooo!

```
template <typename T> static BinNodePosi<T> removeAt(BinNodePosi<T> &
x,BinNodePosi<T> & hot){
    BinNodePosi<T> w = x;
    BinNodePosi<T> succ = nullptr;
    if (!HasLChild(*x)){
        succ = x = x->rChild;
    }
    else if (HasRChild(*x)){
        succ = x = x->lChild;
    }
    else {
        /*左右子树都有*/
        w = w->succ();
        swap(x->data,w->data);
        BinNodePosi<T> u = w->parent;
        (u == x ? u->rChild : u->lChild) = succ = w->rChild;
    }
    hot = w->parent;
```

```
    if (succ) {
        succ->parent = hot;
    }
    release(w->data);
    release(w);
    return succ;
}
```

复杂度

还是O(h) removeAt 本身就一个 succ 的复杂度也为 O(h)

所以总结一下 remove算法为 O(h)