

## 08B3 删除

#数据结构邓神

### 框架

#### 删除：算法

```
❖ template <typename T> bool BST<T>::remove( const T & e ) {  
    BinNodePosi(T) & x = search( e ); //定位目标节点  
    if ( !x ) return false; //确认目标存在 (此时_hot为x的父亲)  
    removeAt( x, _hot ); //分两大类情况实施删除，更新全树规模  
    _size--; //更新全树规模  
    updateHeightAbove( _hot ); //更新_hot及其历代祖先的高度  
    return true;  
} //删除成功与否，由返回值指示
```

```
// 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  $O(h)$

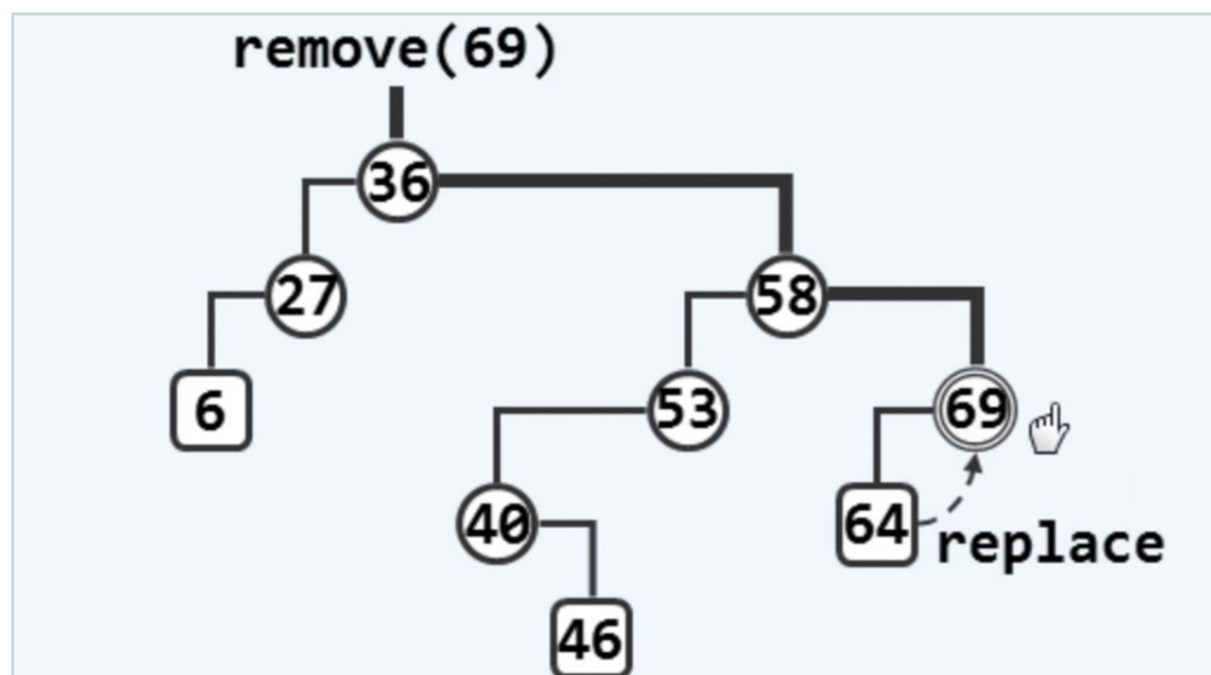
### 单分支

## 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：



实现

## 删除：情况一

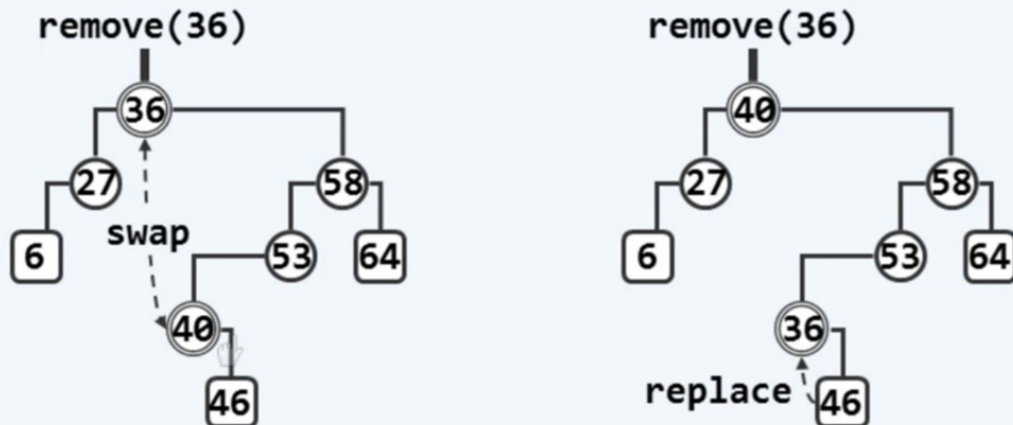
```
❖ template <typename T> static BinNodePosi(T)
removeAt( BinNodePosi(T) & x, BinNodePosi(T) & hot ) {
    BinNodePosi(T) w = x; //实际被摘除的节点，初值同x
    BinNodePosi(T) succ = NULL; //实际被删除节点的接替者
    if ( ! HasLChild( *x ) ) succ = x = x->rChild; //左子树为空
    else if ( ! HasRChild( *x ) ) succ = x = x->lChild; //右子树为空
    else { /* ...左、右子树并存的情况，略微复杂些... */ }
    hot = w->parent; //记录实际被删除节点的父亲
    if ( succ ) succ->parent = hot; //将被删除节点的接替者与hot相联
    release( w->data ); release( w ); //释放被摘除节点
    return succ; //返回接替者
} //此类情况仅需O(1)时间
```

```
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

## BinNode::succ()



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

Woooooooooooooooooooooooooooo!

```
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  $O(h)$