

## 7. 二叉搜索树

### (b) 算法及实现

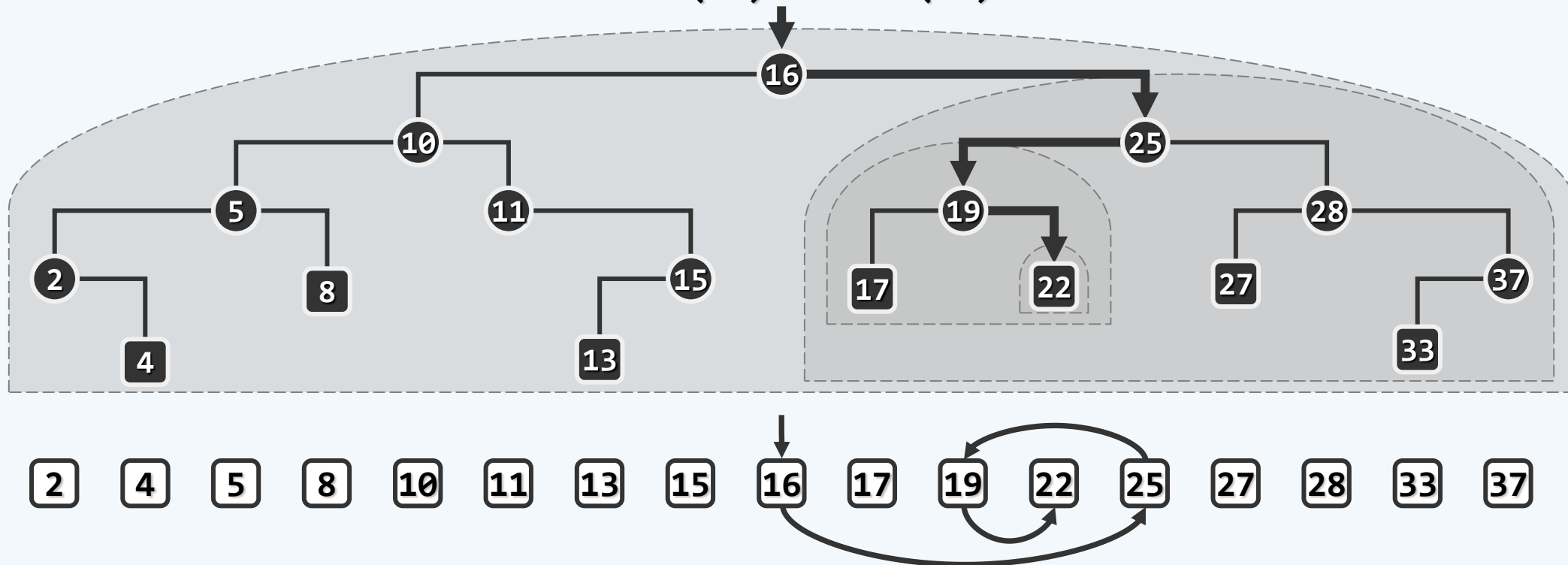
邓俊辉

[deng@tsinghua.edu.cn](mailto:deng@tsinghua.edu.cn)

## 查找：算法

- ❖ **减而治之** 从根节点出发，逐步地缩小查找范围，直到发现目标（成功），或查找范围缩小至空树（失败）

search(22) search(23)



- ❖ 对照中序遍历序列可见，整个过程可视作是在仿效有序向量的二分查找

## 查找：实现

❖ template <typename T>

```
BinNodePosi(T) & BST<T>::search(const T & e)
```

```
{ return searchIn( _root, e, _hot = NULL ); } //从根节点启动查找
```

❖ static BinNodePosi(T) & searchIn( //典型的尾递归，可改为迭代版

```
BinNodePosi(T) & v, const T & e, BinNodePosi(T) & hot) {
```

```
// 当前（子）树根、目标关键码、记忆热点
```

```
if ( !v || ( e == v->data ) ) return v; //足以确定失败、成功，或者
```

```
hot = v; //先记下当前（非空）节点，然后再...
```

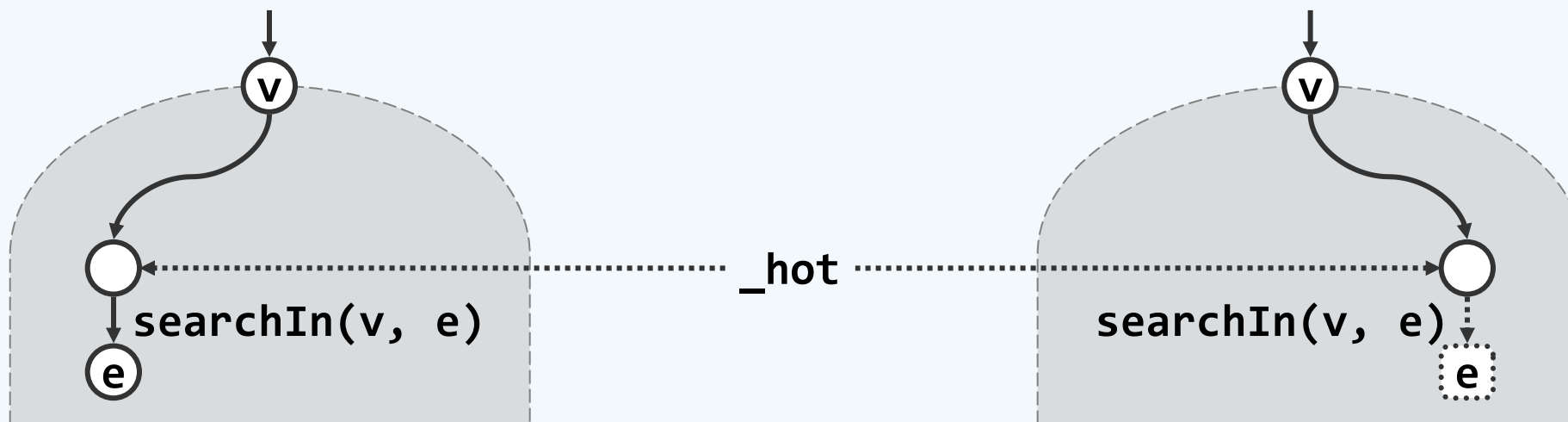
```
return searchIn( ( e < v->data ? v->lChild : v->rChild ), e, hot );
```

```
} //运行时间正比于返回节点v的深度，不超过树高 $O(h)$ 
```

## 查找：接口语义

❖ 返回的引用值：成功时，指向一个关键码为 $e$ 且**真实存在**的节点

失败时，指向最后一次试图转向的空节点**NULL**



❖ 失败时，不妨假想地将此空节点，转换为一个数值为 $e$ 的**哨兵**节点

如此，依然满足BST的充要条件；而且更重要地...

❖ 无论成功与否：返回值总是**等效地**指向**命中**节点，而 `_hot` 总是指向命中节点的**父亲**

## 插入：算法

❖ 先借助 `search(e)` 确定插入位置及方向

再将新节点作为 **叶子** 插入

❖ 若 `e` 尚不存在，则

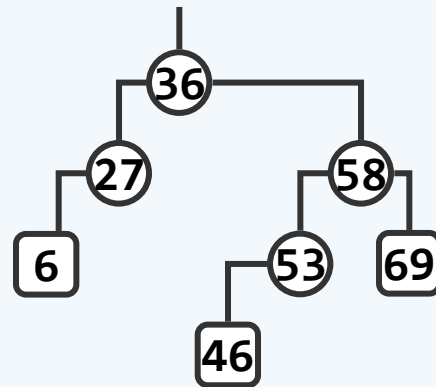
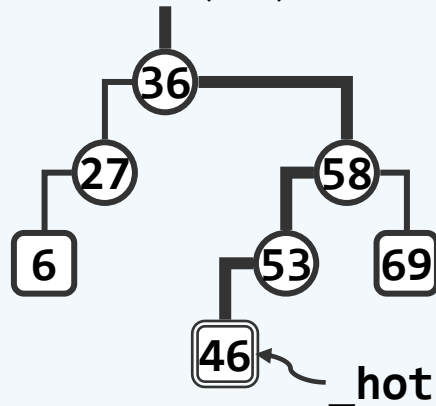
`_hot` 为新节点的 **父亲**

`v = search(e)` 为 `_hot` 对新孩子的 **引用**

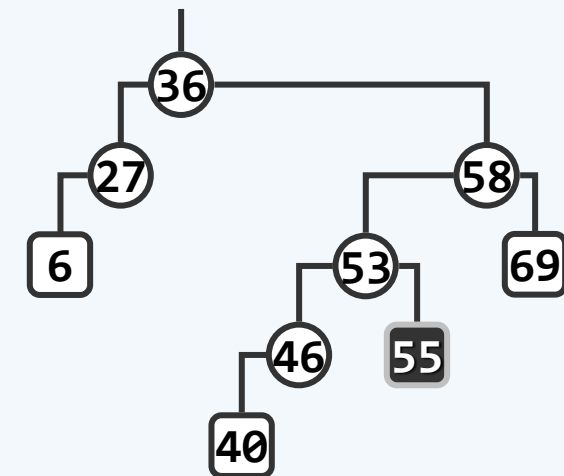
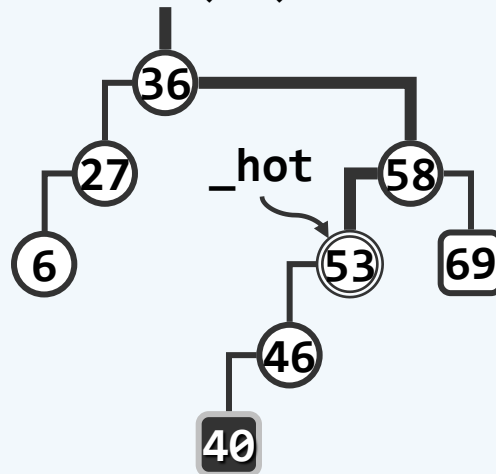
❖ 于是，只需

令 `_hot` 通过 `v` **指向** 新节点

insert(40)



insert(55)



## 插入：实现

- ❖ 

```
template <typename T> BinNodePosi(T) BST<T>::insert( const T & e ) {  
    BinNodePosi(T) & x = search( e ); //查找目标 (留意_hot的设置)  
    if ( ! x ) { //既禁止雷同元素，故仅在查找失败时才实施插入操作  
        x = new BinNode<T>( e, _hot ); //在x处创建新节点，以_hot为父亲  
        _size++; updateHeightAbove( x ); //更新全树规模，更新x及其历代祖先的高度  
    }  
    return x; //无论e是否存在于原树中，至此总有x->data == e  
} //验证：对于首个节点插入之类的边界情况，均可正确处置
```
- ❖ 时间主要消耗于search(e)和updateHeightAbove(x)  
均线性正比于返回节点x的深度，不超过树高 $O(h)$

## 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
```

❖ 时间主要消耗于search()、updateHeightAbove()

还有removeAt()中可能调用的succ()

累计 $O(h)$

## 删除：情况一

❖ 若  $*x(69)$  的某一子树为空

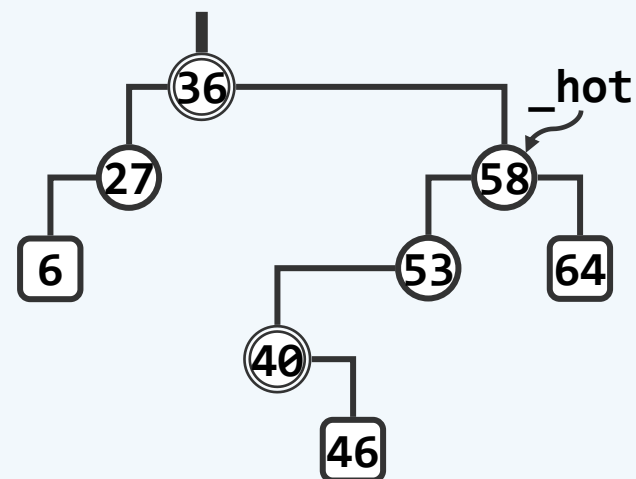
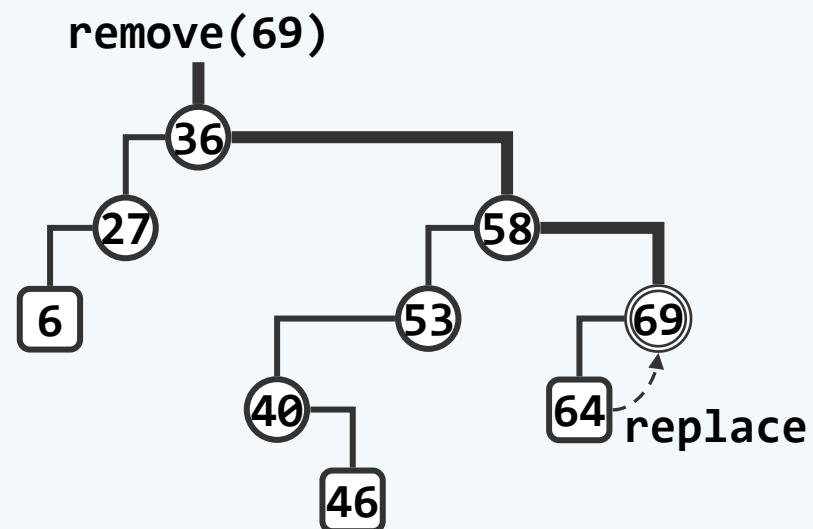
则可将其替换为另一子树 (64)

❖ 验证：

如此操作之后，二叉搜索树的

**拓扑结构** 依然完整

**顺序性** 依然满足





## 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
```

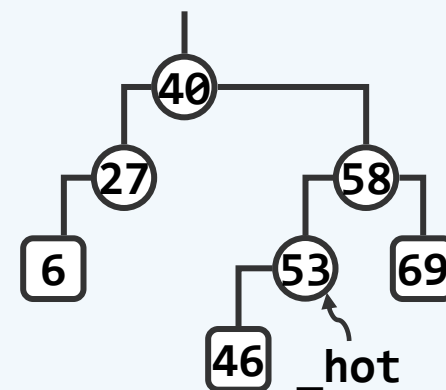
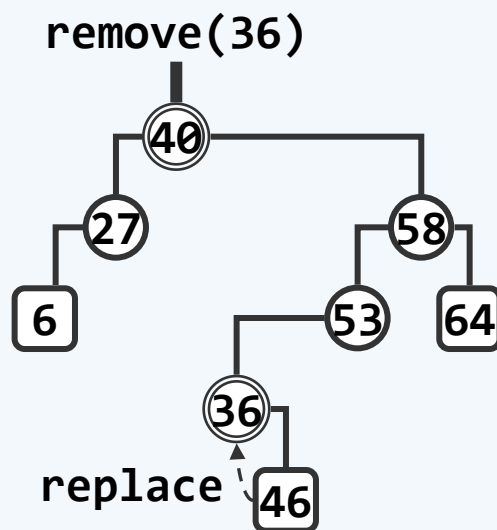
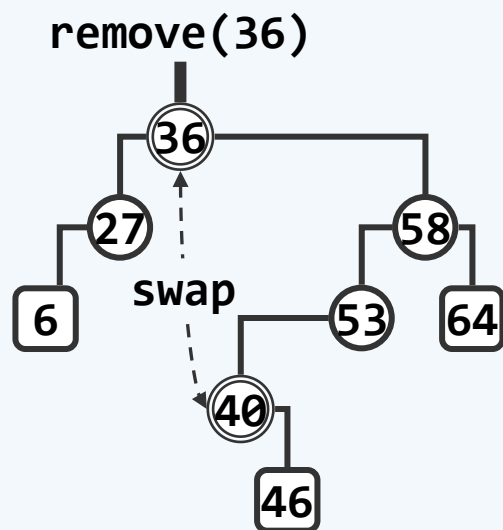
## 删除：情况二

❖ 若： $*x$  ( 36 ) 左、右孩子并存

则：调用 `BinNode::succ()` 找到  $x$  的直接后继 ( 必无左孩子 )；交换  $*x$  ( 36 ) 与  $*w$  ( 40 )

❖ 于是问题转化为删除  $w$ ，可按前一情况处理

❖ 尽管顺序性曾在中途一度不合，但最终必将重新恢复



## 删除：情况二

❖ `template <typename T> static BinNodePosi(T)`

`removeAt( BinNodePosi(T) & x, BinNodePosi(T) & hot ) {`

`/* ..... */`

`else { //若x的左、右子树并存，则`

`w = w->succ(); swap( x->data, w->data ); //令*x与其后继*w互换数据`

`BinNodePosi(T) u = w->parent; //原问题即转化为，摘除非二度的节点w`

`( u == x ? u->rChild : u->lChild ) = succ = w->rChild;`

`}`

`/* ..... */`

`}`

❖ 时间主要消耗于succ()，正比于x的高度——更精确地，search()与succ()总共不过 $O(h)$