

## 10B3 查找

#数据结构邓神

### 查找过程

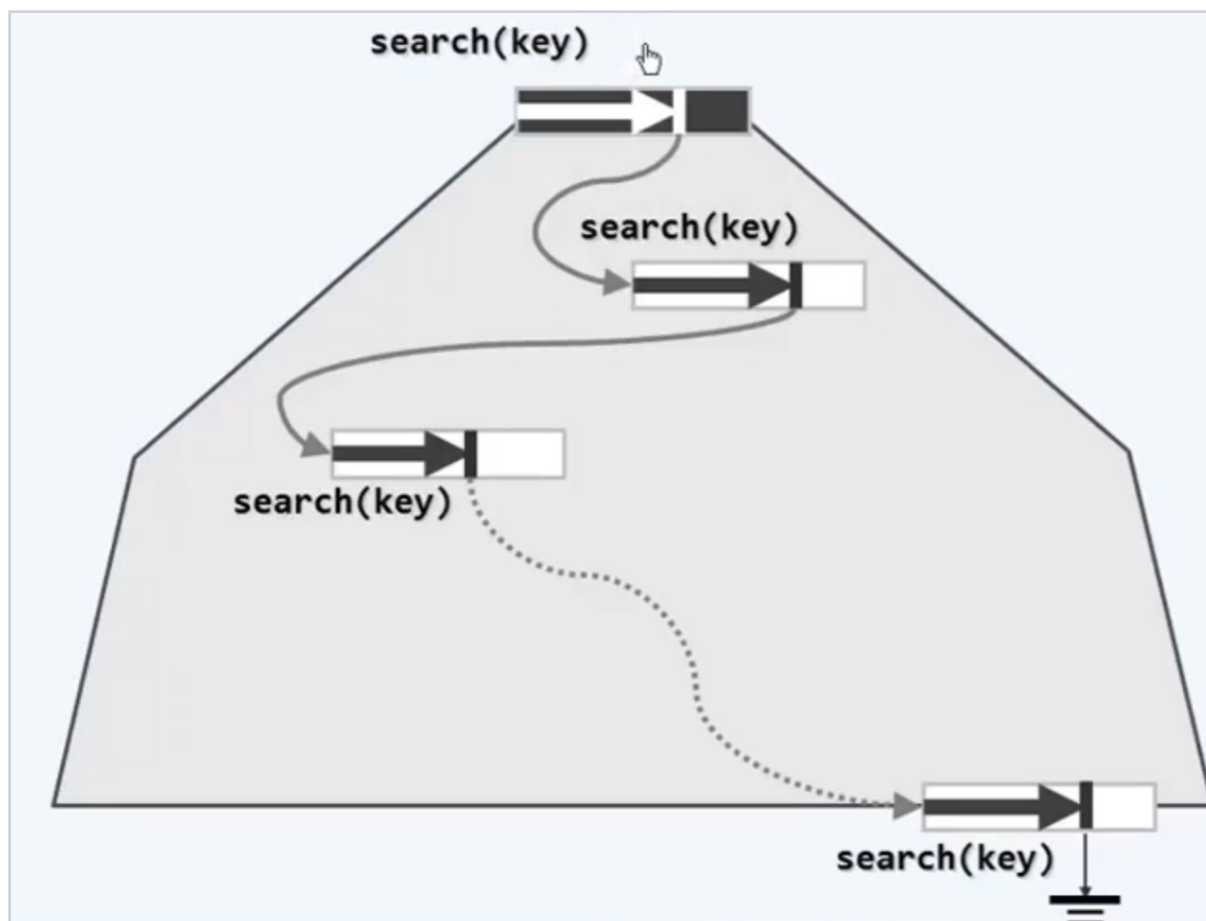
#### 查找诀窍

**只载入必需的节点  
尽可能减少I/O操作**

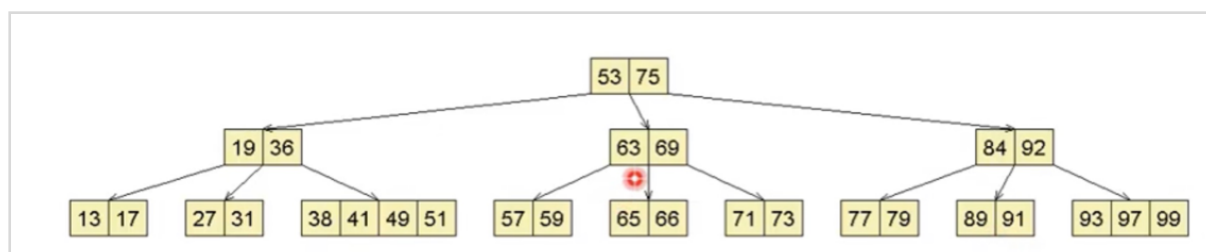
对于一棵活跃的BTree来说，根节点一定要常驻于内存

我们可以直接调用`search(key)` 采用二分查找（在 $m$ 很小的情况下可以使用顺序查找（ $<10?$ ））的算法实施

如果直接找到，就直接返回，如果没有找到就必然会找到一个中间的引用，我们可以由这个引用深入到下一层。然后在下一层进行一次类似的操作，我们不断重复该操作，最后一定会找到或者找到外部节点表示结束



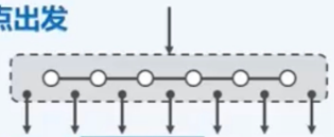
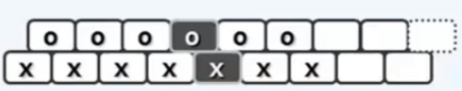
操作实例



算法

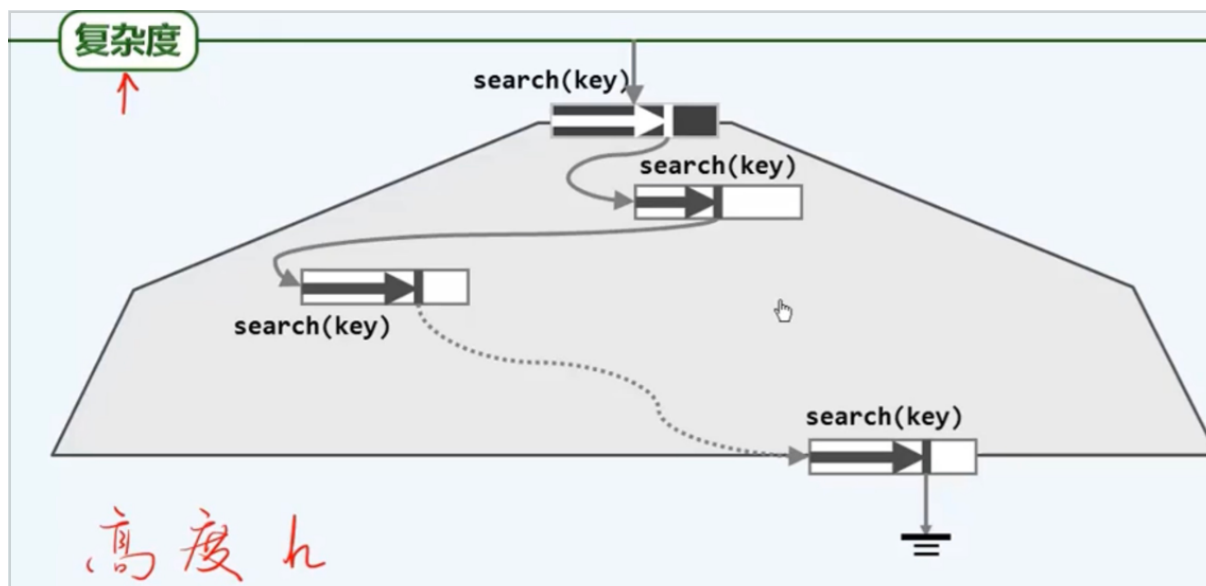
## 实现

```
❖ template <typename T> BTreeNodePosi(T) BTree<T>::search( const T & e ) {
    BTreeNodePosi(T) v = _root; _hot = NULL; //从根节点出发
    while ( v ) { //逐层查找
        Rank r = v->key.search(e); //在当前节点对应的向量中顺序查找
        if ( 0 <= r && e == v->key[ r ] ) return v; //若成功，则返回；否则...
        _hot = v; v = v->child[ r + 1 ]; //沿引用转至对应的下层子树，并载入其根 I/O
    } //若因!v而退出，则意味着抵达外部节点
    return NULL; //失败
}
```

```
// search
template <typename T> BTreeNodePosi(T) BTree<T>::search(const T & e){
    BTreeNodePosi(T) v = _root;
    _hot = nullptr;
    while (v) {
        Rank r = v->key.search(e);
        if (0 <= r && e == v->key[r]){
            return v;
        }
        _hot = v;
        v = v->child[r+1];
    }
    return nullptr;
}
```

复杂度



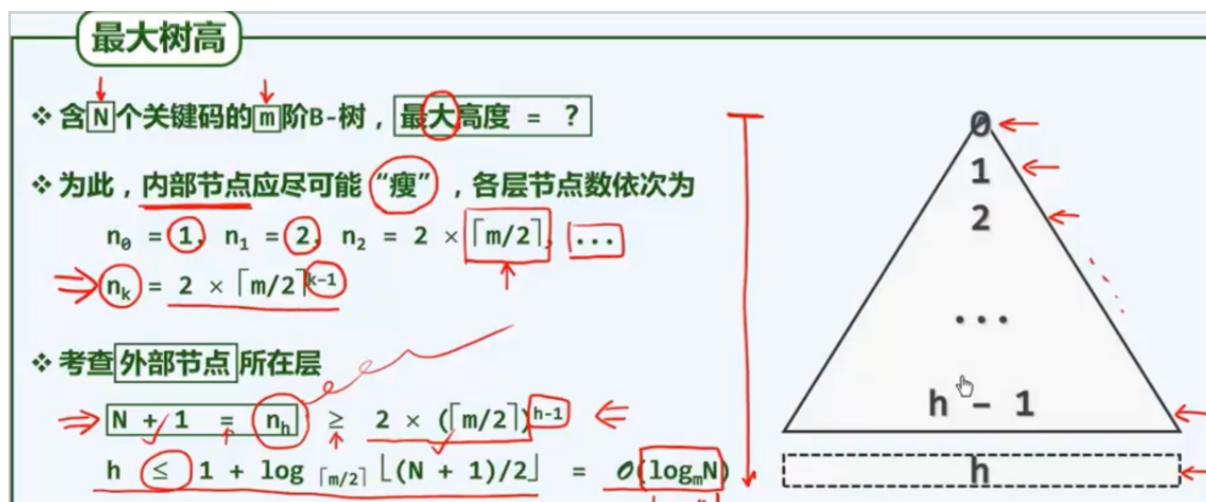
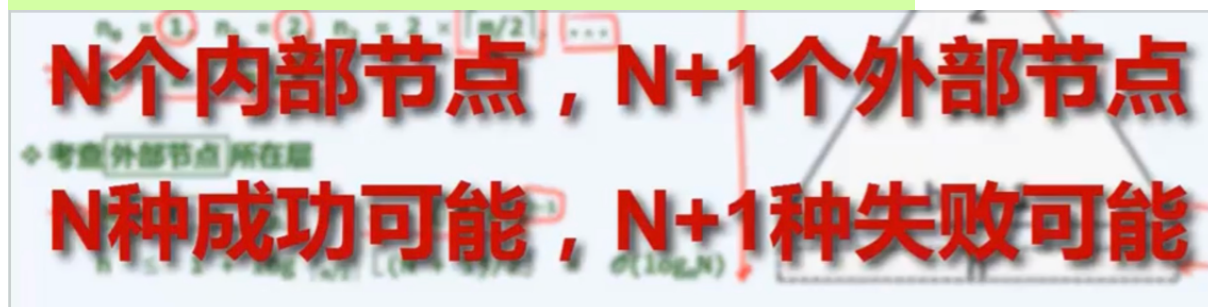
主要消耗时间都在 灰色线条（读入下层节点） 此类操作要远远的慢于访问很多很多次内存的时间消耗

而在另一方面是在每一个节点内部进行的顺序查找，为什么要使用顺序查找，因为内外存的巨大差异，这种优化的操作往往是微乎其微的，甚至往往可能是有害的。

为了与IO操作相比配，每个节点的大小应该与IO读取的大小相匹配，通常为若干个KB，每个节点内所含的关键码的数字大致为几百个，而实验结果显示对于如此规模的有序向量，相对于顺序查找而言，二分查找效率反而更低。

## 最大树高

B树而言，高度由外部节点所在的高度由外部节点所决定，所以要+1



❖ 相对于BBST :

$$\log_{\lceil m/2 \rceil}(N/2) / \log_2 N = 1/(\log_2 m - 1)$$

若取  $m = 256$  , 树高 ( I/O次数 ) 约降低至  $\boxed{1/7}$

最小树高

### 最小树高

❖ 含  $N$  个关键码的  $m$  阶B-树, 最小高度 = ?

❖ 为此, 内部节点应尽可能“胖”  $\leq m$

各层节点数依次为

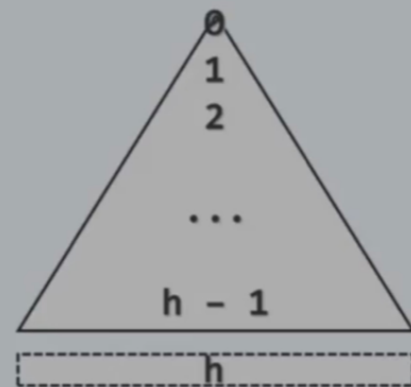
$$n_0 = 1, n_1 = m, n_2 = m^2$$

$$n_3 = m^3, \dots, n_{h-1} = m^{h-1}, n_h = m^h$$

❖ 考查外部节点所在层 :

$$N + 1 = n_h \leq m^h$$

$$h \geq \log_m(N + 1) = \Omega(\log_m N)$$



❖ 相对于BBST :  $(\log_m N - 1) / \log_2 N = \log_m 2 - \log_N 2 \approx 1 / \log_2 m$

若取  $m = 256$  , 树高 ( I/O次数 ) 约降低至  $\boxed{1/8}$

当关键码总数固定, B树高度几乎不变