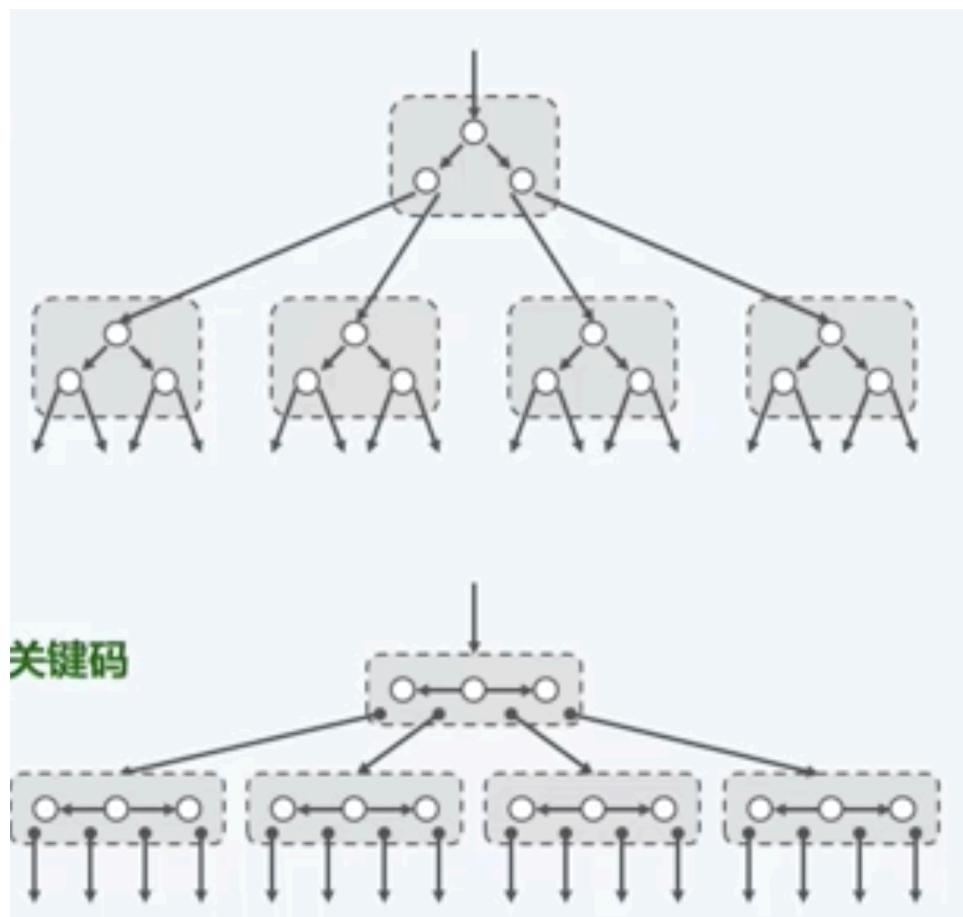


B Tree

多路搜索树

结构



- 等价于BST
- 合并代数任意
 - 2代合并: 3个字段, 4条分支
 - 3代合并: 7个字段, 8条分支
 - m代合并: $2^m - 1$ 个字段, 2^m 条分支

优势

既然等价于BST，为什么我们还需要B树？

- 多级存储系统中使用B树可以大大降低查找数据时的I/O次数
- 外存对于批量访问具有良好的支持，即可以一次读写一个节点中的所有字段，B树可以充分利用该机制
- 如果在n=1GB的数据中查找
 - AVL tree: $\log_2 2^{30} = 30$ 次，每次IO一个字段
 - B tree(假设每个节点存储256个字段)：
 $\log_{256} 2^{30} = 3.75 \leq 4$ 次，每次IO256个字段

特点

- m阶B树即使m路平衡搜索树
 - 每个节点最多有m个孩子/分支
 - 每个节点最多有m-1个字段
 - 每个内部节点（树根可例外）最少有 $\text{math.ceil}(m/2)$ 个孩子

Bayer and McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys.

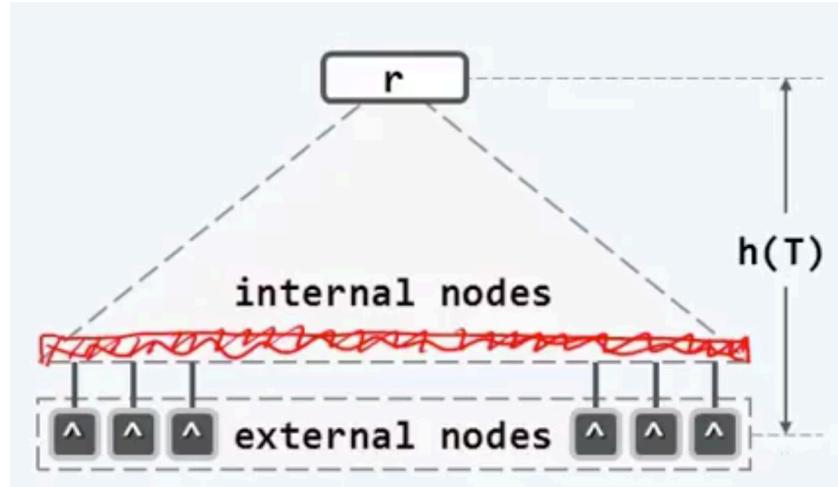
Knuth (1998) avoids the problem by defining the **order to be the maximum number of children** (which is one more than the maximum number of keys).

- 定义

According to Knuth's definition, a B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k - 1$ keys.
5. All leaves appear in the same level and carry no information.

- 所有叶节点深度全都相等
- 外部节点：假想的不存在的叶节点的孩子
 - 所有外部节点深度相等
- B tree中高度相对于外部节点而非叶节点来计算



Implementation

```

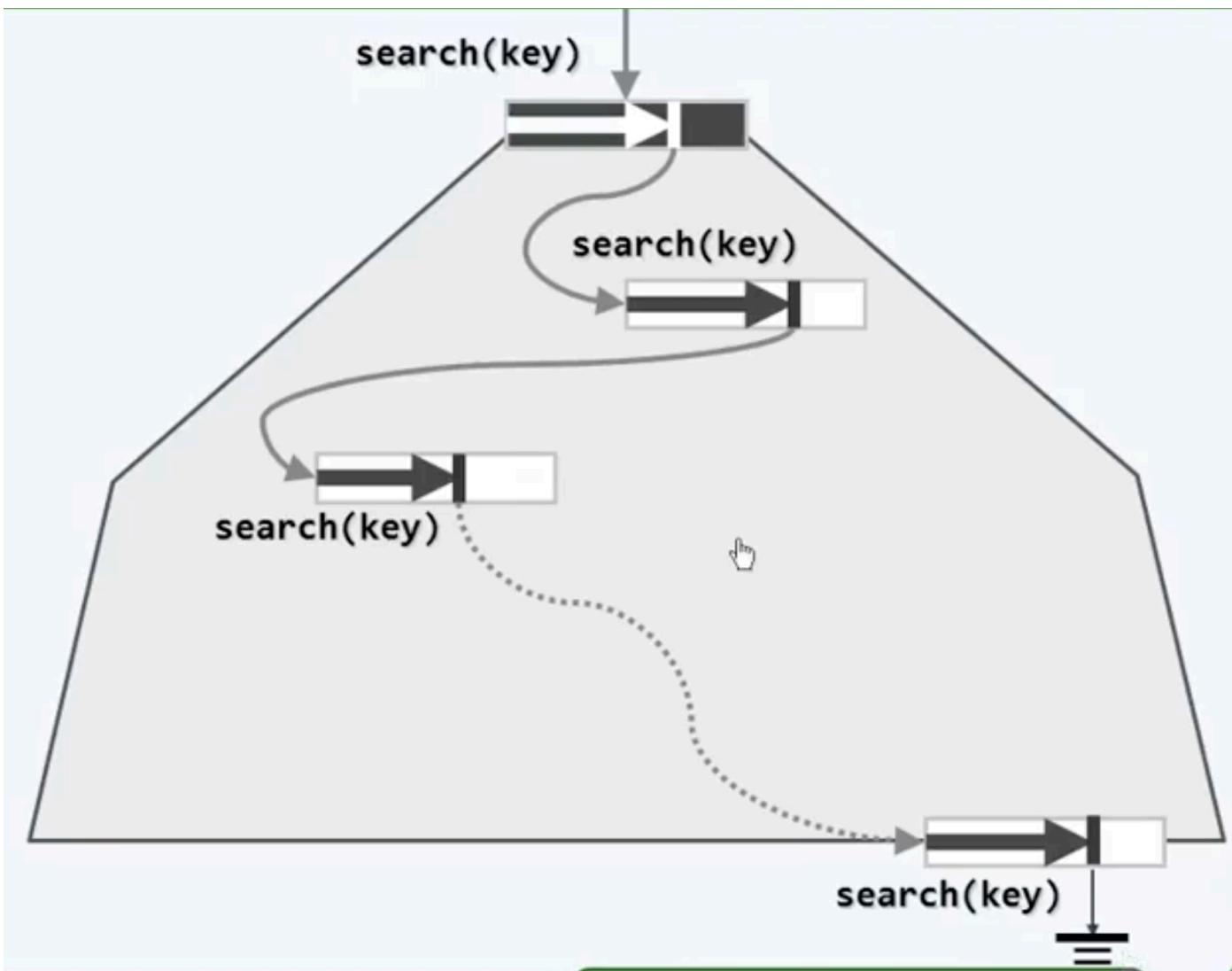
template <typename T>
class BTreeNode{
    BTreeNode& parent ;
    vector<T> keys;
    vector<BTreeNode&> children;
};

template <typename T>
class BTree{
protected:
    BTreeNode& root;
    int size;
    int order;
};

```

查找

只将必要的节点载入内存进行操作，无关节点留在外存中以减少消耗



```

template <typename T>
BTreeNode<T>* BTree<T>::search(T target){
    BTreeNode<T>* v = root;
    hot = nullptr;
    while(v != nullptr){
        int rank = -1;
        for (int i = 0 ; i < v->keys.size() ; i++){
            if(v->keys.at(i)>=target){
                rank = i;
                break;
            }
        }
    }
}

```

```

    }

    if (rank >= 0 && v->keys.at(rank) == target)
return v;

    else {
        hot = v;
        v = v->children.at(rank +1);
    }
}

return nullptr;
}

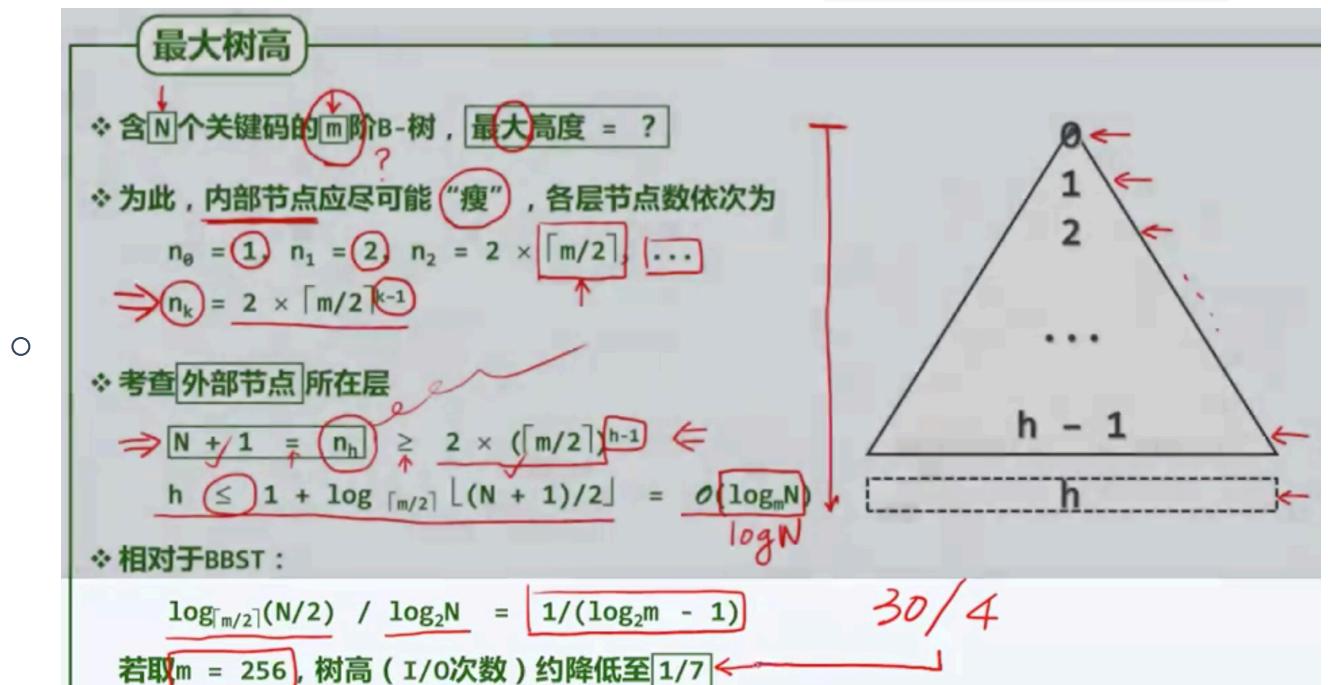
```

查找的复杂度

主要开支为深入树，查找时间可忽略

- 最大树高

- 每个节点的孩子数尽可能的小，为 $\text{math.ceil}(m/2)$



- 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

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

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

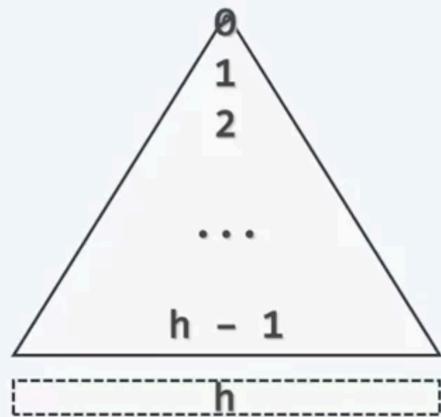
考查外部节点所在层：

$$N + 1 = n_h \leq m^h \quad O(\log_m N)$$

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

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 $1/8$ ← $1/2$



- 最大最小高度相差不多，说明B树高度关于阶数的变化非常有限
- 查找复杂度约为 $O(\log_m N)$

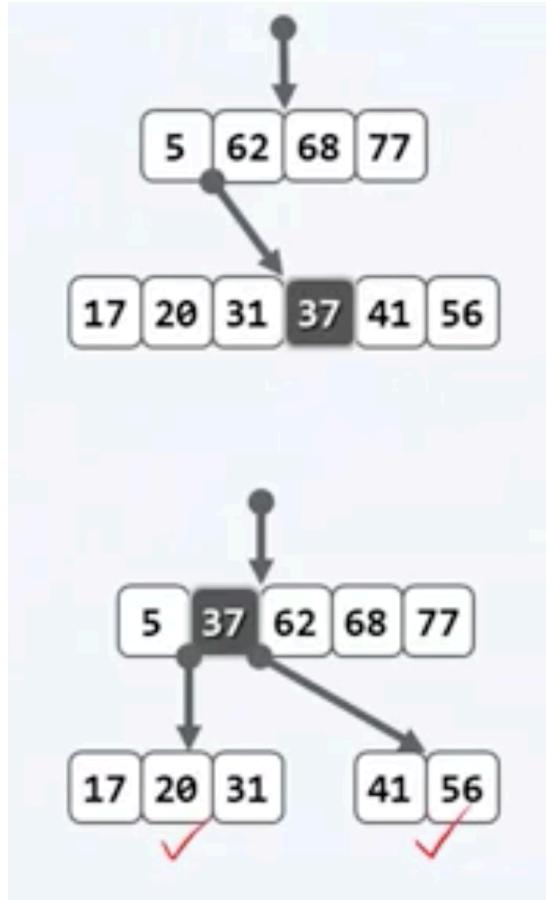
插入

- hot** 记录待插入元素所在的节点
- implementation

```
template <typename T>
bool BTTree<T>::insert(const T & element){
    if (search(element) != nullptr) return false;
    auto pos = std::find(hot->keys.begin() , hot-
>keys.end() , element);
    hot->keys.insert(pos , element);
    hot->children.insert(pos+1 , element);
    size++;
    solveOverflow(hot);
    return true;
}
```

SovleOverflow

- 如果插入后节点中所含分支的个数超过它的阶数则发生 overflow
- 将中位数位置的节点 $\text{math.floor}(n/2)$ 提升至上一层节点，左边和右边分别作为左右孩子



- 如果上层节点依然overflow则同样处理
- 根节点overflow则提升出的中位数元素作为新的根，左右分别为左右孩子
- 因为B tree规定每个节点所含分支的个数 $\lceil \text{math.ceil}(m/2) \rceil, m \rceil$ 所以分裂并不会导致孩子的分支个数underflow
- 最坏情况是从底一直overflow到根， $O(h) = O(\log_m N)$
- implementation

删除

- Implementation

```
template <typename T>
```

```

bool BTTree<T>::remove(const T & element){
    BTTreeNode<T>* v = search(element);
    if (v == nullptr) return false;
    auto rank = std::find(v->keys.begin(), v-
>keys.end(), element)-v->keys.begin();
    if (v->children[0] != nullptr){ // v is not a
leaf

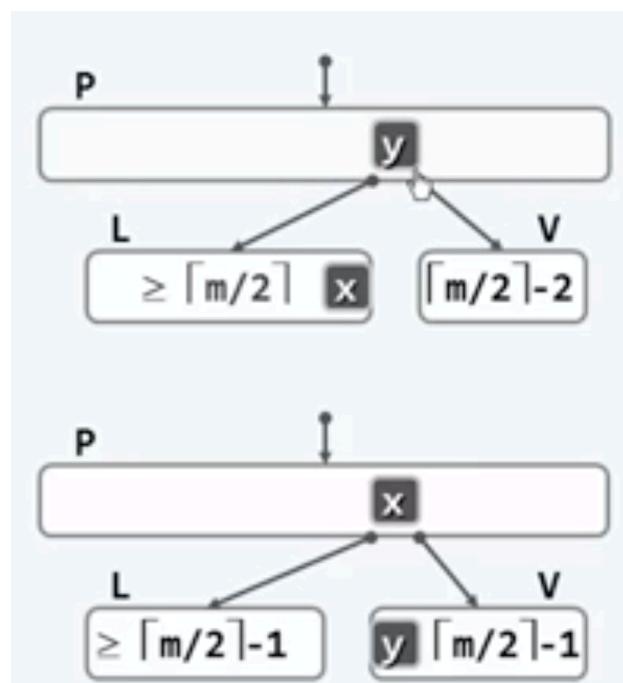
        auto u = v->children[rank+1];
        // keep going left in right child
        while (u->children[0] != nullptr) {
            u = u->children[0]; //find u's
successor
        }
        //swap
        v->keys[rank] = u->keys[0];
        v = u;
        rank = 0;
    }

    //now v must be a leaf
    v->keys.erase(v->keys.begin() + rank);
    v->children.erase(v->children.begin() + rank + 1);
    size--;
    solveUnderflow(v);
    return true;
}

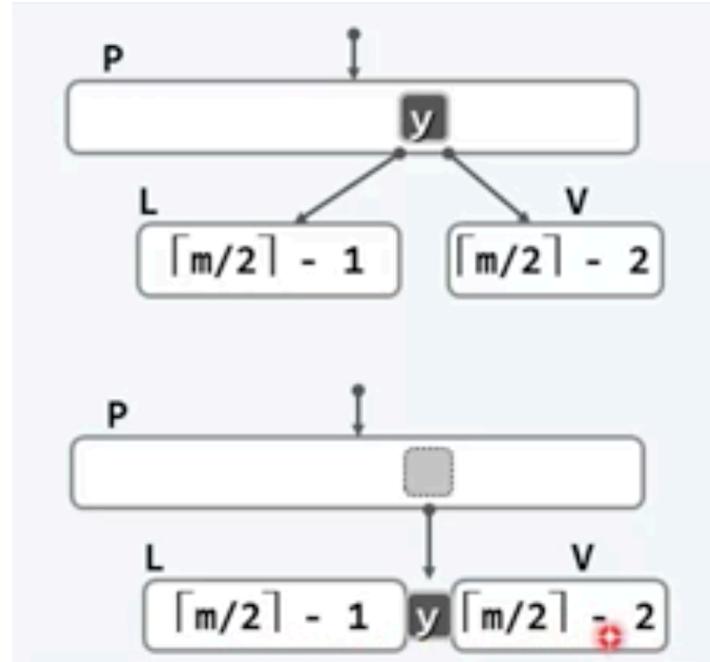
```

SolveUnderflow

- 删掉目标字段后分支个数少于 $\text{math.ceil}(m/2)$ ，发生 underflow
- 先考虑旋转：查看左右兄弟是否可以移出一个字段而不发生 underflow，可以则以父节点为中继，获得一个节点



- 如果不能旋转则合并：因为必定存在一个兄弟，则从父节点中取出一个字段与兄弟合并



如果此时父节点因为失去一个字段而导致underflow则相同
处理直至根节点

- 最坏情况不过一直向上合并至树根, $O(h) = O(\log_m N)$