

10B2 B-树 结构

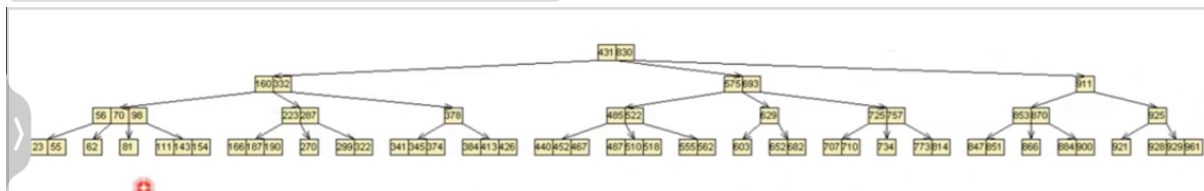
#数据结构邓神

体验



RPreplay_Final163919789...

2021年12月11日

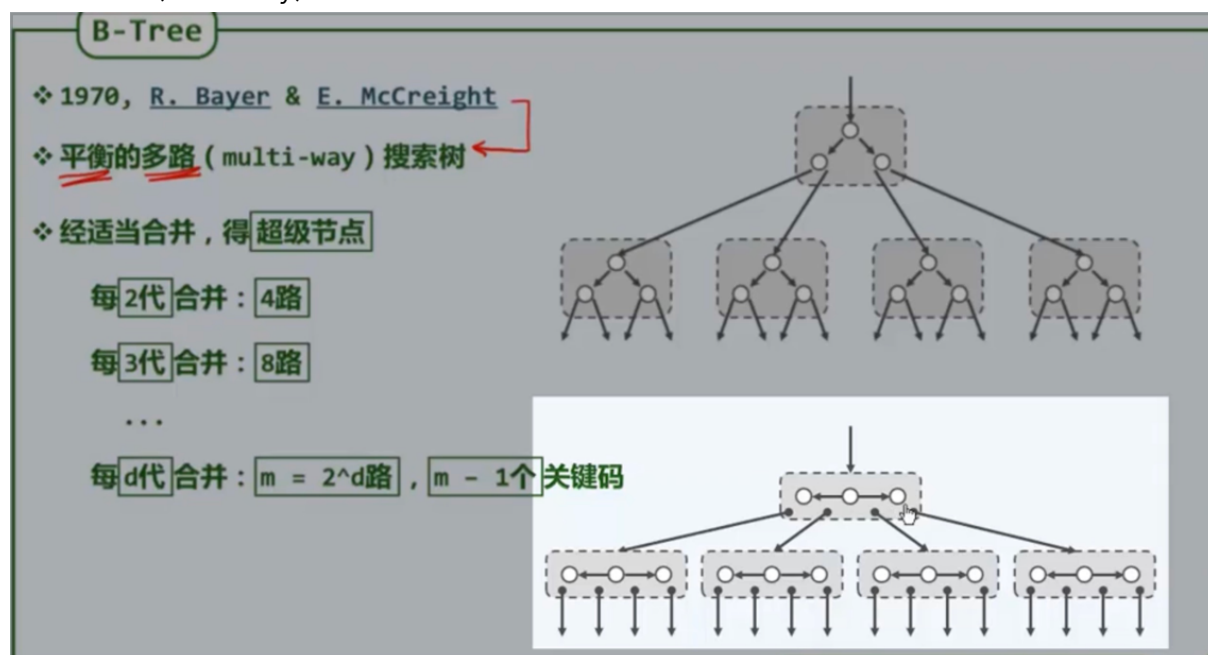


每个节点可能有多个分叉，每个底层节点的深度完全一致。

相对于常规的二叉查找树，B树会更宽更矮，为什么呢？

多路平衡

平衡的多数(multi-way)搜索树



❖ 逻辑上与BBST完全等价

——既然如此，为何还要引入B-树？

还是IO

B-Tree

❖ 多级存储系统中使用B-树，可针对外部查找，大大减少I/O次数

❖ 难道，AVL还不够？比如，若有 $n = 1G$ 个记录...

每次查找需要 $\log_2(10^9) = 30$ 次I/O操作，每次只读出一个关键码，得不偿失

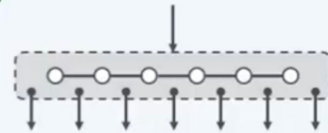
❖ B-树又能如何？

充分利用外存对批量访问的高效支持，将此特点转化为优点

每下降一层，都以超级节点为单位，读入一组关键码

❖ 具体多大一组？视磁盘的数据块大小而定， $m = \#keys / pg$

比如，目前多数数据库系统采用 $m = 200 \sim 300$



❖ 回到上例，若取 $m = 256$ ，则每次查找只需 $\log_{256}(10^9) \leq 4$ 次I/O

深度统一

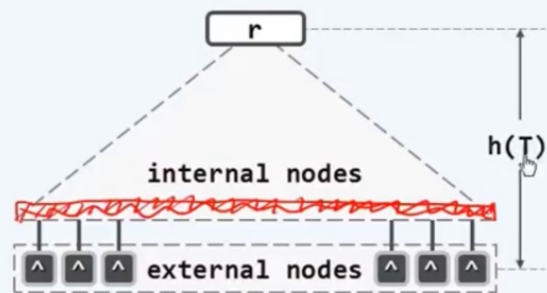
B-Tree

❖ 所谓 m 阶B-树，即 m 路平衡搜索树 ($m \geq 2$)

❖ 外部节点的深度统一相等

所有叶节点的深度统一相等

❖ 树高 $h =$ 外部节点的深度



外部节点：就是叶子节点还不存在的孩子节点

高度定义有变化

阶次含义 (m)

❖ 内部节点各有

不超过 $m - 1$ 个关键码：

$K_1 < K_2 < \dots < K_n$

不超过 m 个分支：

$A_0, A_1, A_2, \dots, A_n$

❖ **内部节点**的分支数 $n + 1$ 也不能太少，具体地

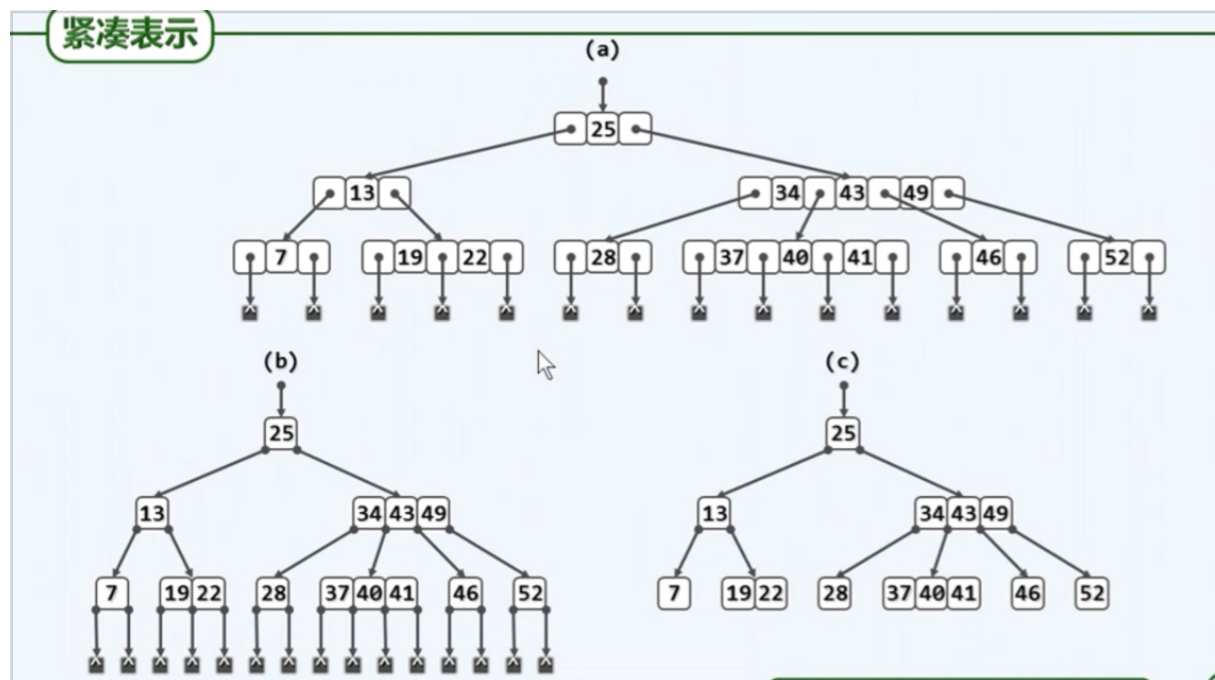
树根： $2 \leq n + 1$

其余： $\lceil m/2 \rceil \leq n + 1$

❖ 故亦称作 **$(\lceil m/2 \rceil, m)$ -树**

(2,4)树和红黑树有解不开的渊源

紧凑表示



A 是完整表示为每一个关键码分别留出一个左右分支引用的位置 (a → b)

我们可以讲所有引用简化为一个点

同时我们也可以简化掉 x (b → c)

BTreeNode

BTNode

❖ template <typename T> struct BTNode { //B-树节点

BTNodePosi(T) parent; //父

Vector<T> key; //数值向量

Vector< BTNodePosi(T) > child; //孩子向量 (其长度总比key多一)

BTNode() { parent = NULL; child.insert(0, NULL); }

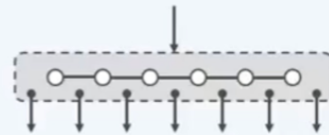
BTNode(T e, BTNodePosi(T) lc = NULL, BTNodePosi(T) rc = NULL) {

parent = NULL; //作为根节点, 而且初始时

key.insert(0, e); //仅一个关键码, 以及

child.insert(0, lc); child.insert(1, rc); //两个孩子

if (lc) lc->parent = this; if (rc) rc->parent = this;



o	o	o	o	o	o				
x	x	x	x	x	x	x	x		

```
#define BTNodePosi(T) BTNode<T>*
```

```
template <typename T> struct BTNode{
```

```
    BTNodePosi(T) parent;
```

```
    vector<T> key;
```

```
    vector<BTNodePosi(T)> child;
```

```
    BTNode(){
```

```
        parent = nullptr;
```

```
        child.insert(0,nullptr);
```

```
    }
```

```
    BTNode(T e,BTNodePosi(T) lc = nullptr,BTNodePosi(T) rc =nullptr){
```

```
        parent = nullptr;
```

```
        key.insert(0,e);
```

```
        child.insert(0,lc);
```

```
        child.insert(1,rc);
```

```
        if (lc) {
```

```
            lc->parent = this;
```

```
        }
```

```
        if (rc){
```

```
            rc->parent = this;
```

```
        }
```

```
    }
```

```
};
```

BTree

BTree

```
❖ #define BTreeNodePosi(T) BTreeNode<T>* //B-树节点位置
❖ template <typename T> class BTree { //B-树
protected:
    int _size; int _order; BTreeNodePosi(T) _root; //关键码总数、阶次、根
    BTreeNodePosi(T) _hot; //search()最后访问的非空节点位置

    void solveOverflow( BTreeNodePosi(T) ); //因插入而[上溢]后的[分裂]处理
    void solveUnderflow( BTreeNodePosi(T) ); //因删除而[下溢]后的[合并]处理
public:
    BTreeNodePosi(T) search(const T & e); //查找
    bool insert(const T & e); //插入
    bool remove(const T & e); //删除
```

```
template <typename T> class BTree {
protected:
    int _size;
    int _order;
    BTreeNodePosi(T) root;
    BTreeNodePosi(T) _hot;
    void solveOverflow(BTreeNodePosi(T));
    void solveUnderFlow(BTreeNodePosi(T));
public:
    BTreeNodePosi(T) search(const T & e);
    bool insert(const T & e);
    bool remove(const T & e);
};
```