1 B-Tree 的定义

一颗 B_Tree 是一个具有下列性质的有根数，假设根结点是 T.root:

1. 每个结点的性质
   (a) $x.n$ 当前结点中关键字的个数
   (b) $x.n$ 个关键子本身 $x.key_1$, $x.key_2$, $\cdots$, $x.key_{x.n}$ 非降序排列，使得 $x.key_1 \leqslant x.key_2 \leqslant \cdots \leqslant x.key_{x.n}$
   (c) $x.leaf$ 是一个 $bool$ 至。如果 x 是叶子结点，则为 $TRUE$，否则为 $FALSE$
2. 每个内部结点有 $x.n+1$ 个指向孩子的指针 $x.c_1$, $x.c_2$, $\cdots$, $x.c_n$。叶子结点没有孩子，它们的孩子指针无定义。
3. 关键字 $x.key_i$ 对存储在各子树的关键字加以分割：如果 k_i 是任意一个存储在以 $x.c_i$ 为根的子树中的关键字，那么

$$k_1 \leqslant x.key_1 \leqslant k_2 \leqslant x.key_2 \leqslant \cdots \leqslant x.key_{x.n} \leqslant k_{x.n+1}$$

4. 每个叶子结点具有相同的深度，即树高 h
5. 每个结点的关键字个数有上界和下界。用最小度数的固定整数 $t \geqslant 2$ 表示这些上下界:
   (a) 除根结点之外，每个结点至少有 $t-1$ 个关键字和 t 个孩子。非空树的情况下，根结点至少有一个关键字。
   (b) 每个结点最多有 $2t-1$ 个关键字和 $2t$ 个孩子。当一个结点有 $2t$ 个孩子时，该结点是满的。

2 B-Tree 的高度

如果 $n \geqslant 1$，对于任意一颗包含 n 个关键字、高度是 h、最小度数 $t \geqslant 2$ 的 B_Tree，都有:

$$h \leqslant \log_t \frac{n+1}{2}$$

3 B-Tree 查找关键字

B 树寻找关键字的过程类似于二叉搜索树, 区别在于二叉搜索树在每个结点只需要确定两个方向, 而 B 树需要确定多个方向. 查找过程采用递归的方式, 这里使用尾递归. 易于理解, 编译优化后效率较高.B 树搜索过程需要进行 $O(\log_t n)$ 次磁盘的读取, 总的时间复杂度为 $O(t \log_t n)$

```
B-TREE-SESRCH(x, k)
i = 1
while  i ⩽ x.n and k > x.key_i
       i = i + 1
if  x ⩽ x.n and k == x.key_i
       return(x, i)
elseif x.leaf
       return NIL
else
       DISKREAD(x, c_i)
       return B-TREE-SESRCH(x.c_i, k)
```

4 B-Tree 的插入

往 B 树插入元素的操作永远在 B 树的叶子结点上进行. 一个叶子根据 B 树的定义, 除根结点外, 每个结点的元素个数介于 $t-1$ 和 $2t-1$ 之间, 所以插入元素的时候要考虑维护 B 树的性质.

插入过程分为三种情况讨论:

1. 初始化情况, 直接在根结点插入
2. 直接插入叶子结点后, 满足 B 树的性质
3. 插入叶子结点不满足 B 树的性质, 此时需要对叶子结点进行分裂

情况 1 比较简单, 可以直接进行. 情况 2 和 3 都是递归的过程. 在向下搜索的过程中, 为了避免回溯的发生, 不能等到找到需要插入的结点才进行分裂, 而是在递归向下的过程中, 分裂沿途的每一个满结点. 采取这样的操作后, 如果是情况 2, 可以直接插入; 如果是情况 3, 那么可以保证分裂叶子结点是, 父结点不是满结点, 不必回溯. 分裂结点是使 B 树增高的唯一途径, 而且一次只能增加一个高度.

辅助过程 B-TREE-INSERT-NOFULL, 决定了递归下降的方向和插入的位置, 参数 x 表示关键字插入的结点,k 表示关键字. 分裂过程 B-TREE-SPLIT-CHILD(x,i) 表示分裂结点 x 的第 i 个孩子.B-TREE-INSERTT(T,k) 表示插入元素 k

B-TREE-SPLIT-CHILD(x,i)

```
z = NEW_NODE()
y = x.c_i
z.leaf = x.leaf
z.n = t − 1
for z = 1 to t − 1
    z.key_j = y.key_{j+t}
if not y.leaf
for j = 1 to t
    z.c_i = y.c_{j+t}
y.n = t − 1
for j = x.n + 1 downto i + 1
    x.c_{i+1} = x.c_i
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(x)
DISK-WRITE(y)
DISK-WRITE(z)
```

B-TREE-INSERTT-NOFULL(x,k)

```
i = x.n
if x.leaf
    while i ⩾ 1 and k < x.key_i
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i ⩾ 1 and k < x.key_i
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i
            i = i + 1
    B-TREE-INSERT-NOFULL(x.c_i, k)
```

B-TREE-INSERTT(T,k)

```
r = T.root
if r.n == 2t − 1
    s = NEW-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NOFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

5 B-TREE 的删除

B 树的删除分为 3 个主要情况:

1. 关键字 k 在结点 x 中,x 是叶结点, 直接在 x 中删除 k
2. 关键字 k 在结点 x 中,x 是内部结点, 操作如下:
   (a) 结点 x 前于 k 的子结点 y 至少含有 t 个关键字, 找出 k 在以 y 为根的子树中的前驱 k', **递归地**删除 k', 并在 x 中用 k' 代替 k.
   (b) 对称的情况, 如果 y 中少于 $t-1$ 个结点, 检查结点 x 中后于 k 的子结点 z. 如果 z 有 t 个关键字, 则找出 k 在以 z 为根的子树中的后继 k'. **递归**地删除 k', 并在 x 中用 k' 代替 k;
   (c) y 和 z 都只是有 $t-1$ 个结点, 则把 k 和 z 合并进入结点 y, 此时的 x 失去了 k 和指向 z 的指针, 且 y 包含 $t-1$ 个关键字. 释放 z, **递归**地从 y 中删除 k
3. 关键字 k 不在内部结点 x 中, 则必包含于 $x.c_i$ 中. 如果 $x.c_i$ 只有 $t-1$ 个关键字, 那么执行这两步保证下降到一个至少包含 t 个关键字的结点.
   (a) 如果 $x.c_i$ 只有 $t-1$ 个关键字, 但它的兄弟至少有 t 个关键字, 则把 x 中的某个关键字下降到 $x.c_i$ 中, 讲 $x.c_i$ 的相邻左兄弟或右兄弟的一个关键字升至 x, 讲该兄弟中相应的孩子指针移到 $x.c_i$ 中, 使 $x.c_i$ 增加一个关键字.
   (b) 如果 $x.c_i$ 以及 $x.c_i$ 的所有兄弟结点都只有 $t-1$ 个关键字, 则把 $x.c_i$ 与一个兄弟合并, 即把 x 的一个关键字移到新合并的结点, 使之成为该结点的中间关键字.

几个函数的说明:

1. MERGE-NODE(x, i, y, z) 表示把内结点 x 的第 i 个孩子 y 和第 $i+1$ 个孩子 z 合并, 形成一个新结点 y
2. B-TREE-PREDECESSOR(y) 表示某结点在前驱子树根是 y 的情况下, 查找在 y 中的递归意义上的前驱结点. 使用迭代就行.

3. B-TREE-SUCCESSOR(z) 表示某结点在后继子树根是 z 的情况下, 查找在 z 中递归意义上的后继结点.

4. B-TREE-SHIFT-TO-RIGHT-CHILD(x, i, y, z) 把 x 的第 i 左孩子 y 的一个结点转移给 y 的右兄弟 z

5. B-TREE-SHIFT-TO-LEFT-CHILD(x, i, y, z) 把 x 的第 $i+1$ 右孩子 z 的一个结点转移给 z 的左兄弟 y

6. B-TREE-DELETE-NONONE(y, k) 删除内部结点 y 中的元素 k

7. DISK-WRITE(x) 把 x 的数据写入磁盘

8. FREE-NODE(x) 释放 x 占用的空间

MERGE-NODE(x,i,y,z)

```
y.n = 2t - 1
for j = t + 1to 2t - 1
    y.key_j = z.key_{j-1}
y.key_t = x.key_i
if not y.leaf
    for j = t + 1to 2t - 1
        y.c_j = z.c_{j-t}
for j = i + 1 to x.n
    x.c_j = x.c_{j+1}
x.n = x,n - 1
FREE-NODE(z)
DISK-WRITE(x)
DISK-WRITE(y)
DISK-WRITE(z)
```

B-TREE-DELETE-NONODE(x,k)

```
i = 1
if x.leaf        ▷ Case 1
    while i ≤ x.n and k > x.key_i
        i = i + 1
    if k == x.key_i
        for j = i + 1 to x.n
            x.key_{j-1} = x.key_j
        x.n = x.n − 1
        DISK-WRITE(x)
    else error "key does not exist"
else while i <= x.n and k > x.key_i
        i = i + 1
    DISK-READ(x.c_{i+1})
    y = x.c_i
    if i ⩽ x.n
        DISK-READ(x.c_{i+1})
        z = x.c_{i+1}
    if k == x.key_i        ▷ Case 2
        if y.n > t − 1        ▷ Case 2a
            k′ = B-SEARCH-PREDECESSOR(Y)
            B-TREE-DELETE-NONONE(y, k′)
            x.key_i = k′
        elseif z.n > t − 1        ▷ Case 2b
            k′ = B-TREE-SUCCESSOR(z)
            B-TREE-DELETE-NONONE(z, k′)
        else B-TREE-MERGE-CHILD(x, i, y, z)        ▷ Case 2c
            B-TREE-DELETE-NONONE(y, k)
    else        ▷ Case 3
        if i > 1
            DISK-READ(x.c_{i-1})
            p = x.c_{i-1}
        if y.n == t − 1
            if i > 1 and p.n > t − 1        ▷ Case 3a
            elseif i <= x.n and z.n > t − 1        ▷ Case 3a
                B-TREE-SHIFT-TO-LEFT-CHILD(x, i, y, z)
            elseif i > 1        ▷ Case 3b
                MERGE-NODE(x, i − 1, p, y)
                y = p
            else MERGE-NODE(x, i, y, z)        ▷ Case 3b
        B-TREE-DELETE-NONONE(y, k)
```

B-TREE-PREDECESSOR(y)

```
x = y
i = x.n
while not x.leaf
    DISK-READ(x.c_{i+1})
    x = x.c_{i+1}
    i = x.n
return x.key_i
```

B-TREE-SUCCESSOR(z)

```
x = z
while not x.leaf
    DISK-READ(x.c_1)
    x = x.c_1
return x.key_1
```

B-TREE-SHIFT-TO-RIGHT-CHILD(x,i,y,z)

```
z.n = z.n + 1
j = z.n
while j > 1
    z.key_j = z.key_{j-1}
    j = j - 1
z.key_1 = x.key_i
x.key_i = y.key_{y.n}
if not z.leaf
    j = z.n
    while j > 0
        z.c_{j+1} = z.c_j
        j = j - 1
    z.c_1 = y.c_{y.n+1}
y.n = y.n - 1
DISK-WRITE(x)
DISK-WRITE(y)
DISK-WRITE(z)
```

B-TREE-SHIFT-TO-LEFT-CHILD(x,i,y,z)

```
y.n = y.n + 1
y.key_{y.n} = x.key_i
x.key_i = z.key_1
z.n = z.n - 1
j = 1
while j ⩽ z.n
    z.key_j = z.key_{j+1}
    j = j + 1
if not z.leaf
    y.c_{y.n+1} = z.c_1
    j = 1
    while j ⩽ z.n + 1
        z.c_j = z.c_{j+1}
        j = j + 1
DISK-WRITE(x)
DISK-WRITE(y)
DISK-WRITE(z)
```

6 总结

7 实验测试