#ADS02 BTree

Red–Black Trees and B Tree

[TOC]

1. Red–Black Trees

Red–Black Trees are one of many search tree schemes that are balanced in order to guarantee that basic dynamic–set operations take `O(lg N)` time in the worst case.

红黑树可以保证在最坏情况下基本动态集合操作的时间复杂度为 `O(lgn)`

By constraining the node colors on any simple path from the root to a leaf, red–black trees **ensure that no such path is more than twice as long as any other**.

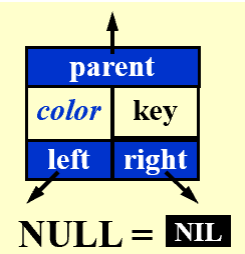
红黑树是一种二叉搜索树，通过在节点上增加一个存储位来表示节点的颜色。通过对任何一条从根到叶子的简单路径上各个结点颜色进行约束，红黑树保证没有一条路径会比其他路径长出2倍，因而是近乎平衡的。

Each node of the tree now contains the attributes **color, key, left, right, and p**. If a **child** or the **parent** of a node does not exist, the corresponding pointer attribute of the node contains the value **NIL**. External node NIL is defined as leaf. **A leaf must be black**. Internal node.

如果一个节点没有子节点或父节点，相应的指针指向 NIL， 可以把这些NIL视为二叉搜索树的叶节点（外部节点）的指针，而带关键字的结点视为内部结点。

1.1 Definition

Target : Balanced binary search tree.



【Definition】: A **red–black** tree is a binary search tree that satisfies the following red–black properties:

1. Every node is either **red** or **black**.
2. The root is **black**.
3. Every leaf (NIL) is **black**.
4. If a node is red, then both its children are **black**.
5. For each node, all simple paths from the node to descendant leaves contain the same number of **black nodes**.
   对每一个结点，从该结点到其所有后代叶节点的简单路径上，均包含相同数目的黑色结点。

可以使用一个哨兵**T.nil** 来代表所有的 **NIL**： 所有的叶结点和根节点的父结点。

【**Definition**】 The **black–height** of any node x, denoted by **bh(x)**, is the number of black nodes on any simple path from x `(x not included)` down to a leaf. **bh(Tree) = bh(root)**.

从某个结点x出发（不包含该结点），到达一个叶节点的任意一条简单路径上的黑色结点个数称为该点的黑高。

【**Lemma**】 A red–black tree with **N internal nodes** has height at most **2 ln(N +1)**.

一个具有N个内部节点的红黑树的高度至多为 **2ln(N +1)**。

【**Proof**】

- 以任一结点x为根的子树至少包含 $2^{bh(x)} - 1$ 个内部结点。
  数学归纳
  对于归纳步骤，考虑一个高度为正且有两个子节点的内部结点x。每个子结点有黑高 bh(x) 或 bh(x) –1 . 可以归纳出每个子节点至少有 $2^{bh(x)-1} - 1$
- 根据性质4，从根节点到叶结点（不包含根节点）的任何一条简单路径上都至少有一半的结点为黑色，因此根的黑高至少为 h/2.

1.2 Rotate

To restore the red–black properties, we must change the colors of some of the nodes in the tree and also change the pointer structure

We change the pointer structure through **rotation**.

C 复制代码

```
Left-rotate(T,x):
y = x.right //set y
x.right = y.left
if y.left != T.nil
    t.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
else if x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

LEFT – ROTATE 和 RIGHT–ROTATE 都在O(1) 内完成，其中只有指针的变化。

1.3 Insertion

We can insert a node into an n–node red–black tree in **O(lg n)** time.

To do so, we:

- **TREE – INSERT**： to insert node *z* into the tree and color z red
- **RB–INSERT–FIXUP** : to guarantee that the red–black properties are preserved

```
RB-INSERT(T,z)
y = T.nil
x = T.root
while x != T.nil
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.p = y  //parent
if y == T.nil
    T.root = z
else if z.key < y.key
    y.left = z
else
    y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T,z)
```

```
RB-INSERT-FIXUP(T,z)
while z.p.color == RED   //插入的结点与其父节点都是红色
    if z.p == z.p.p.left  //如果z父亲是其父节点的左儿子
        y = z.p.p.right
        if y.color == RED   //case 1: 叔结点为红色
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
            //case 2 ： 叔结点为黑色或空，z是右儿子
                z = z.p
                LEFT-ROTATE(T,z)
        else
            //case 3 ： 叔结点为黑色或空，z是左儿子
            z.p.clolr = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T,z.p.p)
    else(same as then clause with right and left exchanged)
T.root.color = BLACK
```

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right;
        if y.color == RED
            y.color = BLACK
            z.p.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else z.p.right == z
            z = z.p
            LEFT-ROTATE(T,z)
        else
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T,z.p.p)
    else
T.root.color = BLACK
```

1.4 Delete

Deletion of a node takes time **O(lg n)**.

- Delete a leaf node : Reset its parent link to nil
- Delete a degree 1 node： Replace the node by its single child
- Delete a degree 2 node：
    - Replace the node by the largest one in its left subtree or the smallest one in the right tree
    - Delete the replacing node from the subtree

RB-TRANSPLANT(T,u,v)

```
RB-TRANSPLANT(T,u,v) // v代替u
if u.p == T.nil  //root u是root，直接替代
    T.root = v
else if u == u.p.left  //u是左儿子
    u.p.left = v
else    //u是右儿子
    u.p.right = v
v.p = u.p
```

RB-DELETE(T,z)

```
y = z
y-original-color = y.color  //记录某点颜色
if z.left == T.nil  // z没有左儿子，z.right代替z
    x = z.right     //x记录替换的点
    RE-TRANSPLANT(T,z,z.right)
else if z.right == T.nil  //z没有右儿子，z.left代替z
    x = z.left
    RB-TRANSPLANT(T,z,z.left)
else  //有两个儿子
    y = TREE-MINIMUM(z.right) // 找到z的右子树中最小的值，它一定没有左儿子
    y-original-color = y.color  //记录颜色
    x = y.right  // x记录y的右儿子
    if y.p == z  // 如果y的父亲是z
       x.p = y
    else RB-TRANSPLANT(T,y,y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T,z,y) // y代替z
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK  //z的右子树中最小的值为如果删除的是黑色，就可能导致性质改变
    RB-DELETE-FIXUP(T,x)
```

RB-DELETE-FIXUP(T,x)

```
C | 复制代码
```

```
while x != T.root and x.color == BLACK
    if x == x.p.left //如果x是左儿子
        w = x.p.right //记w为右儿子，即x的兄弟
        if w.color == RED
            // case 1 : x的兄弟节点是红色的
            w.color = BLACK
            x.p.color = RED
            LEFT-ROTATE(T,x.p)
            w = x.p.right
        if w.left.color == BLACK AND w.right.color == BLACK
            // case 2: x的兄弟节点w是黑色的，而且w的两个子结点都是黑色的
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
            // case 3 : x的兄弟节点w是黑色的,左儿子是红色的，右儿子是黑色的
            w.left.color = BLACK
            w.color = RED
            RIGHT-ROTATE(T,w)
            w = x.p.right
        else
            // case 4 : x的兄弟节点w是黑色的,左儿子是黑色的，右儿子是红色的
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            LEFT-ROTATE(T,x.p)
            x = T.root
    else (same as then clause with right and left exchanged)
x.color = BLACK
```

2. B+ Tree

B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices. B-trees are similar to red-black trees , but they are **better at minimizing disk I/O operations**. Many database systems use B-trees, or variants of B-trees, to store information.

B–trees are similar to red–black trees in that every n–node B–tree has height O(lg n). The exact height of a B–tree can be considerably less than that of a red–black tree.
B+–tree, stores **all the satellite information in the leaves** and **stores only keys and child pointers in the internal nodes**, thus maximizing the branching factor of the internal nodes.

2.1 Definition

【**Definition**】 A **B+ tree** of order **M** is a tree with the following structural properties:
(1) The root is either a leaf or has between 2 and M children.
(2) All nonleaf nodes (except the root) have between [M/2] and M children.
(3) All leaves are at the **same depth**.
Assume each nonroot leaf also has between [M/2] and M children.

All the actual data are stored at the leaves.
Each interior node contains M pointers to the children.
And M – 1 smallest key values in the subtrees except the 1st one.

For a general B+ tree of order M

```
Btree  Insert ( ElementType X,  Btree T )
{
   Search from root to leaf for X and find the proper leaf node;
   Insert X;
   while ( this node has M+1 keys ) {
         split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
   respectively;
         if (this node is the root)
                create a new root with two children;
         check its parent;
   }
}
```

$T(M, N) = O((M/\log M) \log N)$

$\text{Depth}(M, N) = O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

$T_{Find}(M, N) = O(\log N)$

Note: The best choice of M is 3 or 4.

2.2 B-Tree Search

C | 复制代码

```c
B-Tree Search(x,k)
{
    i = 1;
    while i <= x.n and k > x.keyi
        i++;
    if(i <= x.n and k == x.keyi)
         return (x,i)
    else if x.leaf
         return NIL;
    else DISK_READ(x,ci)
         return B-Tree Search(x.ci,k)
}
```

对满的非根结点的分裂不会使B树的高度增加，导致B树高度增加的唯一方式是对根结点的分裂。