§4.2 红黑树

从上一节的内容可知，给定一个高度为h的二叉查找树，其上的基本动态集合操作，如：MEMBER, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, DELETE可以在时间实现。而二叉查找树的平均高度为。**这一节讲的红黑树将保证以上操作在最坏情况下，执行时间为。**

一．红黑树的性质

1. 什么是红黑树？

**一棵红黑树是一棵二叉查找树，**每个结点有一个域保存它的颜色，该颜色或者是红，或者是黑。**在红黑树中，从根到任一叶子的路径长度至多是从根到另一叶子的路径长度的两倍。**

2. 红黑树结点的数据结构

红黑树的每个结点是一个记录，包括以下域：color, key, left, right, p。其中，left和right分别指向该结点的左、右孩子。p指向该结点的父亲结点。

**在红黑树中，若某个结点没有左(右)孩子，则它的left(right)指针指向一个关键字为NIL的叶结点。NIL结点具有与普通结点相同的结构。**

3. 红黑树的性质

(1) 每一个结点或者是红，或者是黑；

(2) **每个叶子(NIL)是黑**；

(3) **如果一个结点是红，则它的两个孩子都是黑；**

**(4) 从一个结点到它任一后代叶结点的路径具有相同数目的黑结点。**

**例：**

图4.2

**设某个结点到后代叶结点路上的黑结点数(不含)称为的黑高度，记为。**

引理4.1：有n个内部结点的红黑树的高至多为。

证明：首先证明以结点为根的子树含至少个内部结点。对的高度用归纳法证明。如果的高度是0，则必然是叶子(NIL)，因而以为根的子树含至少个内部结点。归纳步，考虑一个具有正的高度的结点，并且具有两个孩子。则的每个孩子如果是红色，则其黑高度为bh(), 如果是黑色，则其黑高度为。由归纳假设，的每个孩子为根的子树至少有个内部结点，故为根的子树有至少()+()+1

个内部结点。

设h是树的高度，由性质(3)，从根结点到叶子结点的任一路径上至少一半结点(不含根)是黑色，因而树的黑高度至少为。所以

由上式变换可得，故。

4. 算法分析

由引理4.1知，在红黑树上，MEMBER, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR可在时间内实现。

INSERT, DELETE可能破坏红黑树的性质。下面我们将提供在时间内实现INSERT和DELETE的算法。

二．旋转操作

在二叉查找树的INSERT和DELETE操作中，操作结果可能与红黑树的性质相冲突，为此我们引入旋转操作，用以调整二叉查找树，使其保持红黑树的性质。

1. 旋转操作

图4.3

1. 算法：

PROCEDURE LEFT\_ROTATE(T, x);【x左旋转】

BEGIN

y := x.right;

x.right := y.left;

IF y.left NIL THEN

y.left.p := x;

y.p := x.p;

IF x.p = NIL THEN

root[T] := y

ELSE IF x = x.p.left THEN

x.p.left := y

ELSE x.p.right := y;

y.left := x;

x.p := y;

END;

RIGHT\_ROTATE与上述过程相似。

1. 算法分析

**LEFT\_ROTATE和RIGHT\_ROTATE操作只需O(1)时间。**

三．插入操作

以下给出红黑树插入元素的操作，它首先按二叉查找树的方法插入一个元素x，并给x红色，然后调整整棵红黑树使其保持红黑树的性质。

1. 算法：

PROCEDURE RB\_INSERT (T, x);

BEGIN

INSERT(T, x)

x.color := RED;

WHILE (x root[T]) AND (x.p.color = RED) DO

IF root[T] = x.p THEN

x := root[T]

ELSE IF x.p = x.p.p.left THEN

BEGIN

y := x.p.p.right;

IF y.color = RED THEN

BEGIN Case 1

x.p.color := BLACK;

y.color := BLACK;

x.p.p.color := RED;

x := x.p.p;

END

ELSE BEGIN

IF x = x.p.right THEN

BEGIN Case 2

x := x.p;

LEFT\_ROTATE(T, x);

END;

x.p.color := BLACK; Case 3

x.p.p.color := RED; Case 3

RIGHT\_ROTATE(T, x.p.p); Case 3

END

END

ELSE /\*same as THEN clause with “right” and “left”

exchanged \*/

root[T].color := BLACK;

END;

1. 算法分析

n个结点的红黑树高为。调用INSERT过程需时间，WHILE循环只有在执行Case 1时，才重复执行，指针x沿树中路径向根运动，因此WHILE 循环执行总次数为。故

RB\_INSERT需时间。

四．删除操作

红黑树的删除操作与插入操作类似，先按二叉查找树的方法删除一个元素，然后调整红黑树，使该树满足红黑树的性质。

每个叶子是一个标记为NIL的结点。为了节省存储空间，可用一个特殊的NIL结点代表所有的叶子。但在算法中必须小心地设置它的p指针域。

1. 算法：

PROCEDURE RB\_DELETE (T, z);

BEGIN

IF (z.left = nil[T]) OR (z.right = nil[T]) THEN y := z

ELSE y := SUCCESSOR(z);(z的后继)

IF y.left nil[T] THEN x := y.left

ELSE x := y.right;

x.p := y.p;

IF y.p = nil[T] THEN root[T] := x

ELSE IF y = y.p.left THEN y.p.left := x

ELSE y.p.right := x;

IF y z THEN z.key := y.key;【y是z的后继】

IF y.color = BLACK THEN

RB\_DELETE\_FIXUP(T, x);

RETURN(y);（如果y是红色的，删掉y并不影响树的结构）

END;

PROCEDURE RB\_DELETE\_FIXUP (T, x);【更新红黑树结构】

BEGIN

WHILE (x root[T]) AND (x.color = BLACK) DO

**IF** x = x.p.left THEN（x是一个左孩子）

**BEGIN**

w := x.p.right;（令w是另一边的右孩子）

**IF** w.color = RED THEN（如果w是红色的）

**BEGIN**  Case 1

w.color := BLACK;

x.p.color := RED;

LEFT\_ROTATE (T, x.p);

w := x.p.right;

END;

**IF** (w.left.color = BLACK) AND (w.right.color = BLACK) THEN BEGIN Case 2

w.color := RED;

x := x.p;

END

**ELSE** BEGIN

IF w.right.color = BLACK THEN

BEGIN Case 3

w.left.color := BLACK;

w.color := RED;

RIGHT\_ROTATE (T, w);

w := x.p.right;

END;

w.color := x.p.color; Case 4

x.p.color := BLACK; Case 4

w.right.color := BLACK; Case 4

LEFT\_ROTATE (T, x.p); Case 4

x := root[T]; Case 4

END

END

ELSE /\*same as THEN clause with “right” and “left” exchanged\*/

x.color := BLACK;

END;

1. 算法分析：

由于红黑树的高为, RB\_DELETE不含RB\_DELETE\_FIXUP调用所花的时间为。在RB\_DELETE\_FIXUP中，只有在Case 2下，WHILE循环才重复执行，指针x沿树中路径向根移动，因而WHILE循环最多重复次，而整个过程的执行时间为。故删除操作可在时间内完成。

作业6：

1.给出在空树中连续插入41, 38, 31, 12, 19, 8后得到的红黑树，并给出按以下顺序删除元素8, 12, 19, 31, 38, 41, 每删除一个元素后的红黑树。

2.设一个元素用RB\_INSERT插入到一棵红黑树中。然后立即用

RB\_DELETE从该红黑树中删除。问删除后的红黑树是否与插入前的红黑树相同？证明你的答案。