Experiment—Red-Black Insert Algorithm

PB18010496 杨乐园

Introduction

编程实现红黑树的插入算法，使得插入后红黑树依旧保持红黑性质，也即实现书中$P178$页的$RBInsert$、$RBInsertFixup$算法，于此同时，将插入后完成后的红黑树进行先序遍历(NLR)和中序遍历(LNR)，并将相应的遍历序列输出到对应的文件中。

其中，红黑树节点属性为：

$$
\begin{aligned}
TNode = \{ & \\
& color : red/color; \\
& key : int; \\
& left : TNode*; \\
& right : TNode*; \\
& parent : TNode*; \\
\} &
\end{aligned}
$$

Purpose

实验目的：熟悉并掌握红黑树的结构性质，实现红黑树的插入算法与维护算法。

其中红黑树的结构性质为：

1.每个节点必须为红色或黑色；

2.根为黑色；

3.树中的nil叶子为黑；

4.若某节点为红色，则其两个孩子节点必为黑色；

5.每个节点到其后代叶子的所有路径含有同样多的黑节点。

Design ideas

1.红黑树插入算法设计思路与步骤：

$step1$.将待插入节点z按BFS树规则插入红黑树中，其中z为叶子结点。

$step2$.将叶子节点z涂红。

$step3$.调整红黑树使插入后的树满足红黑树的结构性质。

2.维护算法的设计思路与步骤：

调整的思路：通过旋转和改变相应节点的颜色，自上而下调整（对所插入节点z进行上溯），直至使树满足红黑树结构性质。

节点z插入后违反情况：由于节点z为红色，其两个孩子节点$T.nil$为黑色，故当节点z恰为根节点时，可能违反性质2；另一方面，若其父母节点$z.parent$为红色时，则违反性质4。

调整步骤：

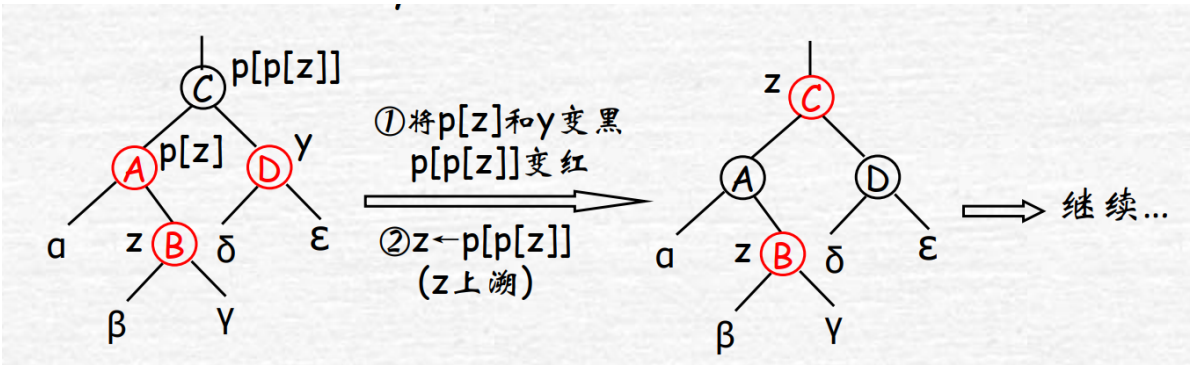
$step1$:若节点z为根节点时，只需将其涂红即可。

$step2$:若节点z非根且其父母节点$z.parent$为红色，分以下6种情况进行调整。

$case1 \sim case3$为z的双亲$z.parent$是其祖父$z.parent.parent$的左孩子。

$case1 \sim case3$为z的双亲$z.parent$是其祖父$z.parent.parent$的右孩子。

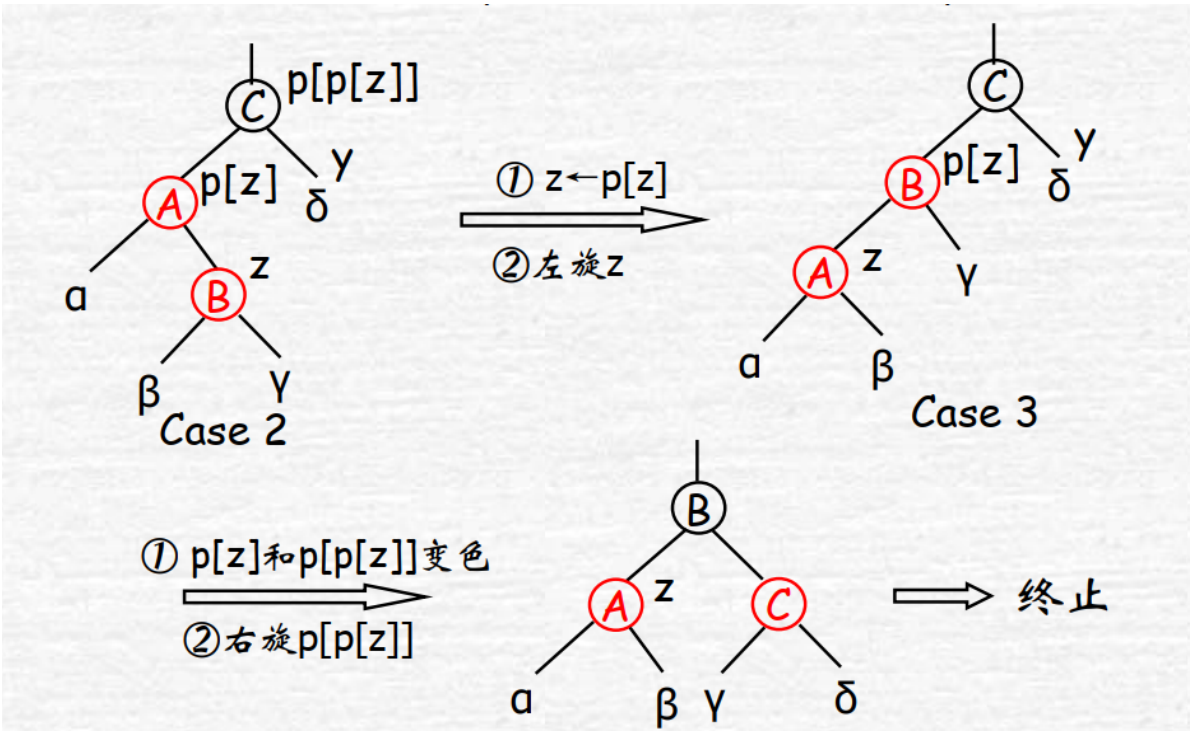
$case1$:z的叔叔y为红色。



调整步骤为：将$z.parent$和y变为黑色，将$z.parent.parent$变为红色，并对z上溯至$z.parent.parent$。可以看到调整后仍可能违反红黑树结构性质4，故需要继续上溯直至根节点检查完毕

$case2$:z的叔叔为黑色，且z是双亲$z.parent$的右孩子。对于此种情况我们可以通过对$z.parent$进行左旋，将其转化为$case3$，直接见下面$case3$即可。

$case3$:z的叔叔为黑色，且z是双亲$z.parent$的左孩子。



调整步骤为：将$z.parent$变为黑色，将$z.parent.parent$变为红色，再对$z.parent.parent$进行右旋即可。

而对于$case4 \sim case6$，完全类似上述$case1 \sim case3$情况，即只需对上述进行“左”“右”意义上的对换即可，具体可参见下述代码部分。

Algorithm

首先我们先将辅助的一些函数写好，如下分别为左旋与右旋函数：

```
//左旋，输入树T,待旋转的节点x;
void LeftRotate(RBTree* T, TNode* x)
```

```
{
    TNode* y = x->right;     //y为x的右孩子；
    x->right = y->left;
    if(y->left!=T->nil)
        y->left->parent = x;
    y->parent = x->parent;   //y的父母节点指向x的父母节点；
    if (x->parent == T->nil)
        T->root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

//右旋；
void RightRotate(RBTree* T, TNode* y)
{
    TNode* x = y->left;
    y->left = x->right;
    if (x->right != T->nil)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == T->nil)
        T->root = x;
    else if (y == y->parent->right)
        y->parent->right = x;
    else
        y->parent->left = x;
    x->right = y;
    y->parent = x;
}
```

递归形式的中序遍历与先序遍历：

```
//中序遍历，其中outfile为输出流，将结果输出到指定文件里面；
int LNR(TNode* root, ofstream& outfile)
{
    if (root->key == error)
        return 0;

    LNR(root->left, outfile);
    outfile << root->key;
    if (root->color == black)
        outfile << ",black" << endl;
    else
        outfile << ",red" << endl;
    LNR(root->right,outfile);
    return 0;
}

//先序遍历；
int NLR(TNode* root ,ofstream& outfile)
{
    if (root->key == error)
        return 0;
```

```
    outfile << root->key;
    if (root->color == black)
        outfile << ",black" << endl;
    else
        outfile << ",red" << endl;

    NLR(root->left, outfile);
    NLR(root->right, outfile);
    return 0;
}
```

红黑树插入算法：

```
void RBInsert(RBTree* T, int k)
{
    TNode* y = T->nil; //y记录当前扫描节点的双亲结点；
    TNode* x = T->root;//从根开始扫描
    TNode* z = (TNode*)malloc(sizeof(TNode));
    z->color = black;
    z->key = k;
    while (x != T->nil)
    {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;      //y是z的双亲
    if (y == T->nil)    //z插入空树，故z是根；
        T->root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;
    z->left = T->nil;  //初始化z；
    z->right = T->nil;
    z->color = red;
    RBInsertFixup(T, z);
}
```

插入算法的维护调整算法：

```
//插入的修正；
void RBInsertFixup(RBTree* T, TNode* z)
{
    while (z->parent->color == red)
        //若z为根，则z->parent=T->nil，颜色黑，不进入此循环；
        //若z->parent为黑，则无需调整，同样不进入循环；
    {
        if (z->parent == z->parent->parent->left)//case1,2,3
        {
            TNode* y = z->parent->parent->right; //y是z的叔叔；
            if (y->color == red)                 //case1
            {
```

```
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else                                    //case2 or case3，y为黑
            {
                if (z == z->parent->right)          //case2
                {
                    z = z->parent;
                    LeftRotate(T, z);
                }                                   //以下为case3
                z->parent->color = black;
                z->parent->parent->color = red;
                RightRotate(T, z->parent->parent);
            }
        }
        else                                        //case4,5,6
        {
            TNode* y = z->parent->parent->left; //y是z的叔叔；
            if (y->color == red)                    //case4
            {
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else                                    //case5 or case6，y为黑
            {
                if (z == z->parent->left)           //case6
                {
                    z = z->parent;
                    RightRotate(T, z);
                }                                   //以下为case6
                z->parent->color = black;
                z->parent->parent->color = red;
                LeftRotate(T, z->parent->parent);
            }
        }
    }
    T->root->color = black;
}
```

Results

我们对如下数据进行红黑树插入算法的正确性测试，其中第一行为插入节点的总数，第二行为依次待插入的节点关键字，类型为int型。

```
8
50 20 70 15 30 25 35 32
```

对插入后所形成的红黑树进行中序遍历(LNR)，结果如下：

```
15,black
20,red
25,black
30,black
32,red
35,black
50,red
70,black
```

对插入后所形成的红黑树进行先序遍历(NLR)，结果如下：

```
30,black
20,red
15,black
25,black
50,red
35,black
32,red
70,black
```

通过对插入所生成的红黑树进行中序与先序遍历后的结果观察我们发现，红黑树插入算法以及插入维护算法实现正确。

Code

具体完整代码，参看附件文件$RedBlackTree$。