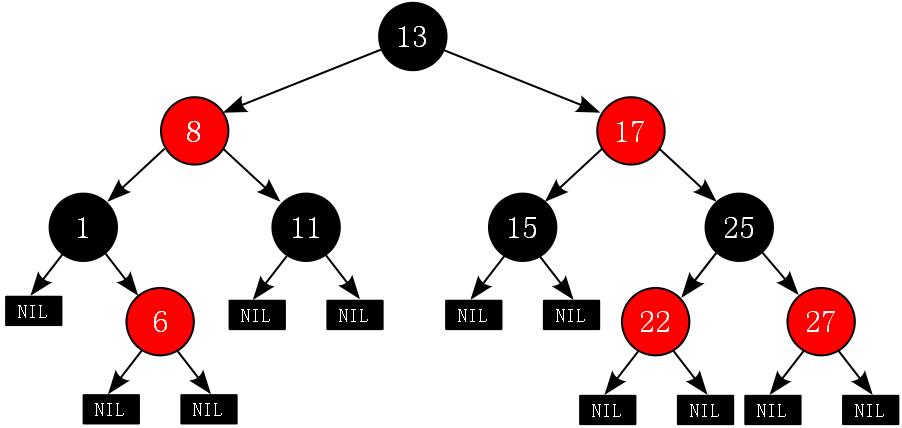
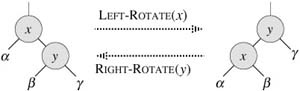
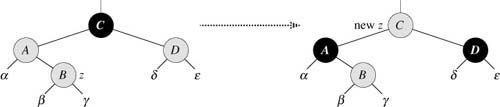
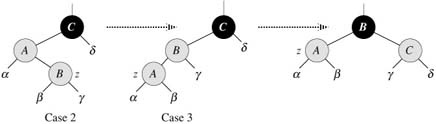
**[红黑树](http://kmplayer.javaeye.com/blog/558257)**

**文章分类:**[**C++编程**](http://www.javaeye.com/blogs/category/cpp)

**二叉排序树**   
1,二叉排序树（Binary Sort Tree）又称二叉查找树。   
2,它或者是一棵空树；或者是具有下列性质的二叉树：   
(1)若左子树不空，则左子树上所有结点的值均小于它的根结点的值；   
(2)若右子树不空，则右子树上所有结点的值均大于它的根结点的值；   
(3)左、右子树也分别为二叉排序树；   
  
**平衡二叉树**   
1,平衡二叉树，又称AVL树。   
2,它或者是一棵空树，或者是具有下列性质的二叉树：   
(1)它的左子树和右子树都是平衡二叉树;   
(2)左子树和右子树的高度之差之差的绝对值不超过1.   
 **红黑树**   
1,红黑树是一种自平衡二叉查找树.   
典型的用途是实现关联数组。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。   
2,红黑树是一种很有意思的平衡检索树。   
它的统计性能要好于平衡二叉树(AVL-树)，因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(目前包括set, multiset, map, multimap)应用了红黑树的变体(SGI STL中的红黑树有一些变化，这些修改提供了更好的性能，以及对set操作的支持)。   
3,红黑树的每个节点上的属性除了有一个key、3个指针：parent、lchild、rchild以外，还多了一个属性：color。它只能是两种颜色：红或黑。而红黑树除了具有二叉搜索树的所有性质之外，还具有以下4点性质：   
(1). 根节点是黑色的。   
(2). 空节点是黑色的（红黑树中，根节点的parent以及所有叶节点lchild、rchild都不指向NULL，而是指向一个定义好的空节点）。   
(3). 红色节点的父、左子、右子节点都是黑色。   
(4). 在任何一棵子树中，每一条从根节点向下走到空节点的路径上包含的黑色节点数量都相同。   
  
4,在插入、删除节点后，就有可能破坏了红黑树的性质。所以我们要做一些操作来把整棵树修补好。下面我就来介绍一下。   
  
首先有一个预备知识，那就是节点的Left-Rotate和Right-Rotate操作。所谓Left-Rotate(x)就是把节点x向左下方向移动一格，然后让x原来的右子节点代替它的位置。而Right-Rotate当然就是把Left-Rotate左、右互反一下。如下图：   
  
  
  
一、 插入   
  
插入首先是按部就班二叉搜索树的插入步骤，把新节点z插入到某一个叶节点的位置上。   
接下来把z的颜色设成红色。为什么？还记得红黑树的性质吗，从根节点向下到空节点的每一条路径上的黑色节点数要相同。如果新插入的是黑色节点，那么它所在的路径上就多出了一个黑色的节点了。所以新插入的节点一定要设成红色。但是这样可能又有一个矛盾，如果z的父节点也是红色，怎么办，前面说过红色节点的子节点必须是黑色。因此我们要执行下面一个迭代的过程，称为Insert-Fixup，来修补这棵红黑树。   
  
在Insert-Fixup中，每一次迭代的开始，指针z一定都指向一个红色的节点。如果z->parent是黑色，那我们就大功告成了；如果z->parent是红色，显然这就违返了红黑的树性质，那么我们要想办法把z或者z->parent变成黑色，但这要建立在不破坏红黑树的其他性质的基础上。   
  
这里再引入两个指针：grandfather，指向z->parent->parent，也就是z的爷爷(显然由于 z->parent为红色，grandfather一定是黑色)；uncle，指向grandfather除了z->parent之外的另一个子节点，也就是z的父亲的兄弟，所以叫uncle。   
  
（为了说话方便，我们这里都假设z->parent是grandfather的左子节点，而uncle是grandfather的右子节点。如果遇到的实际情况不是这样，那也只要把所有操作中的左、右互反就可以了。）   
  
在每一次迭代中，我们可能遇到以下三种情况。   
Case 1. uncle也是红色。这时只要把z->parent和uncle都设成黑色，并把grandfather设成红色。这样仍然确保了每一条路径上的黑色节点数不变。然后把z指向grandfather，并开始新一轮的迭代。   
如下图：   
  
  
  
Case 2. uncle是黑色，并且z是z->parent的右子节点。这时我们只要把z指向z->parent，然后做一次Left- Rotate(z)。就可以把情况转化成Case 3。   
  
Case 3. uncle是黑色，并且z是z->parent的左子节点。到了这一步，我们就剩最后一步了。只要把z->parent设成黑色，把 grandfather设成红色，再做一次Right-Rotate(grandfather)，整棵树就修补完毕了。可以思考一下，这样一次操作之后，确实满足了所有红黑树的性质。   
Case 2和Case 3如下图：   
  
  
反复进行迭代，直到某一次迭代开始时z->parent为黑色而告终，也就是当遇到Case 3后，做完它而告终。   
  
二、删除   
  
让我们来回顾一下二叉搜索树的删除节点z的过程：如果z没有子节点，那么直接删除即可；如果z只有一个子节点，那么让这个子节点来代替z的位置，然后把z删除即可；如果z有两个子节点，那么找到z在中序遍历中的后继节点s（也就是从z->rchild开始向左下方一直走到底的那一个节点），把 s的key赋值给z的key，然后删除s。   
  
在红黑树中，删除一个节点z的方法也是首先按部就班以上的过程（当然，前面说的“如果 z 没有子节点”等类似的判别条件，在红黑树中，要加上“除了空节点之外”这个前提）。   
如果删除的节点是黑色的，那么显然它所在的路径上就少一个黑色节点，那么红黑树的性质就被破坏了。这时我们就要执行一个称为Delete-Fixup的过程，来修补这棵树。下面我就来讲解一下。   
  
一个节点被删除之后，一定有一个它的子节点代替了它的位置（即使是叶节点被删除后，也会有一个空节点来代替它的位置。前面说过，在红黑树中，空节点是一个实际存在的节点。）。我们就设指针x指向这个代替位置的节点。   
显然，如果x是红色的，那么我们只要把它设成黑色，它所在的路径上就重新多出了一个黑色节点，那么红黑树的性质就满足了。   
然而，如果x是黑色的，那我们就要假想x上背负了2个单位的黑色。那么红黑树的性质也同样不破坏，但是我们要找到某一个红色的节点，把x上“超载”的这1 个单位的黑色丢给它，这样才算完成。Delete-Fixup做的就是这个工作。   
  
Delete-Fixup同样是一个循环迭代的过程。每一次迭代开始时，如果指针x指向一个红色节点，那么大功告成，把它设成黑色即告终。相反如果 x黑色，那么我们就会面对以下4种情况。   
  
这里引入另一个指针w，指向x的兄弟。这里我们都默认x是x->parent的左子节点，则w是x->parent的右子节点。（如果实际遇到相反的情况，只要把所有操作中的左、右互反一下就可以了。）   
  
Case 1. w是红色。这时我们根据红黑树的性质可以肯定x->parent是黑色、w->lchild是黑色。我们把x->parent与w的颜色互换，然后做一次Left-Rotate(x->parent)。做完之后x就有了一个新的兄弟：原w->lchild，前面说过它一定是黑色的。那么我们就在不破坏红黑树性质的前提下，把Case 1转换成了Case2、3、4中的一个，也就是w是黑色的情况。思考一下，这样做不会改变每条路径上黑色节点的个数，如下图：

4,实现代码:

C++代码

#include <iostream>

using namespace std;

//Key中的内容可能更复杂，比如字符串等。

struct Key

{

int value;

};

struct RBTNode

{

Key key;

int lcount;

int rcount;

RBTNode\* lchild;

RBTNode\* rchild;

RBTNode\* parent;

bool color;

};

class RBT

{

private:

const static bool RED = true;

const static bool BLACK = false;

RBTNode\* m\_null;

RBTNode\* m\_root;

void clear();

void delFixup(RBTNode\* delNode);

void insertFixup(RBTNode\* insertNode);

//比较两个Key的大小。这里可能有更复杂的比较，如字符串比较等。

inline int keyCmp(const Key& key1, const Key& key2)

{

return key1.value - key2.value;

}

//把一个节点向左下方移一格，并让他原来的右子节点代替它的位置。

inline void leftRotate(RBTNode\* node)

{

RBTNode\* right = node->rchild;

node->rchild = right->lchild;

node->rcount = right->lcount;

node->rchild->parent = node;

right->parent = node->parent;

if (right->parent == m\_null)

{

m\_root = right;

}

else if (node == node->parent->lchild)

{

node->parent->lchild = right;

}

else

{

node->parent->rchild = right;

}

right->lchild = node;

right->lcount += node->lcount + 1;

node->parent = right;

}

//把一个节点向右下方移一格，并让他原来的左子节点代替它的位置。

inline void rightRotate(RBTNode\* node)

{

RBTNode\* left = node->lchild;

node->lchild = left->rchild;

node->lcount = left->rcount;

node->lchild->parent = node;

left->parent = node->parent;

if (left->parent == m\_null) {

m\_root = left;

}

else if (node == node->parent->lchild) {

node->parent->lchild = left;

}

else {

node->parent->rchild = left;

}

left->rchild = node;

left->rcount += node->rcount + 1;

node->parent = left;

}

//找到子树中最大的一个节点

RBTNode\* treeMax(RBTNode\* root);

//找到子树中最小的一个节点

RBTNode\* treeMin(RBTNode\* root);

public:

RBT();

~RBT();

//找到从小到大排序后下标为i的节点。i从0开始。

RBTNode\* atIndex(int i);

//删除一个节点

void del(RBTNode\* node);

void init();

//插入一个节点

void insert(const Key& key);

//返回节点的总个数

int nodeCount();

//按照key查找一个节点。

RBTNode\* search(const Key& key);

//把树中节点的值放进一个数组。

void toArray(int\* array);

//一个节点在中序遍列中的下一个节点。

RBTNode\* treeNext(RBTNode\* node);

//一个节点在中序遍列中的前一个节点。

RBTNode\* treePre(RBTNode\* node);

};

void RBT::clear()

{

RBTNode\* p = m\_root;

while (p != m\_null)

{

if (p->lchild != m\_null) {

p = p->lchild;

}

else if (p->rchild != m\_null) {

p = p->rchild;

}

else {

RBTNode\* temp = p;

p = p->parent;

if (temp == p->lchild)

{

p->lchild = m\_null; //null取代当前要被删除的节点

}

else {

p->rchild = m\_null;

}

delete temp;

}

}

}

//待看...

void RBT::delFixup(RBTNode\* delNode)

{

RBTNode\* p = delNode;

while (p != m\_root && p->color == BLACK) {

if (p == p->parent->lchild) {

RBTNode\* sibling = p->parent->rchild;

if (sibling->color == RED) {

sibling->color = BLACK;

p->parent->color = RED;

leftRotate(p->parent);

sibling = p->parent->rchild;

}

if (sibling->lchild->color == BLACK

&& sibling->rchild->color == BLACK

) {

sibling->color = RED;

p = p->parent;

}

else {

if (sibling->rchild->color == BLACK) {

sibling->lchild->color = BLACK;

sibling->color = RED;

rightRotate(sibling);

sibling = sibling->parent;

}

sibling->color = sibling->parent->color;

sibling->parent->color = BLACK;

sibling->rchild->color = BLACK;

leftRotate(sibling->parent);

p = m\_root;

}

}

else {

RBTNode\* sibling = p->parent->lchild;

if (sibling->color == RED) {

sibling->color = BLACK;

p->parent->color = RED;

rightRotate(p->parent);

sibling = p->parent->lchild;

}

if (sibling->lchild->color == BLACK

&& sibling->rchild->color == BLACK

) {

sibling->color = RED;

p = p->parent;

}

else {

if (sibling->lchild->color == BLACK) {

sibling->rchild->color = BLACK;

sibling->color = RED;

leftRotate(sibling);

sibling = sibling->parent;

}

sibling->color = sibling->parent->color;

sibling->parent->color = BLACK;

sibling->lchild->color = BLACK;

rightRotate(sibling->parent);

p = m\_root;

}

}

}

p->color = BLACK;

}

void RBT::insertFixup(RBTNode\* insertNode)

{

RBTNode\* p = insertNode;

while (p->parent->color == RED)

{

if (p->parent == p->parent->parent->lchild)

{

RBTNode\* parentRight = p->parent->parent->rchild;

if (parentRight->color == RED)

{

p->parent->color = BLACK;

parentRight->color = BLACK;

p->parent->parent->color = RED;

p = p->parent->parent;

}

else

{

if (p == p->parent->rchild)

{

p = p->parent;

leftRotate(p);

}

p->parent->color = BLACK;

p->parent->parent->color = RED;

rightRotate(p->parent->parent);

}

}

else

{

RBTNode\* parentLeft = p->parent->parent->lchild;

if (parentLeft->color == RED)

{

p->parent->color = BLACK;

parentLeft->color = BLACK;

p->parent->parent->color = RED;

p = p->parent->parent;

}

else

{

if (p == p->parent->lchild)

{

p = p->parent;

rightRotate(p);

}

p->parent->color = BLACK;

p->parent->parent->color = RED;

leftRotate(p->parent->parent);

}

}

}

m\_root->color = BLACK;

}

RBTNode\* RBT::treeMax(RBTNode\* root)

{

RBTNode\* result = root;

while (result->rchild != m\_null)

{

result = result->rchild;

}

return result;

}

//找到子树中最小的一个节点

RBTNode\* RBT::treeMin(RBTNode\* root)

{

RBTNode\* result = root;

while (result->lchild != m\_null)

{

result = result->lchild;

}

return result;

}

RBT::RBT() : m\_null(new RBTNode)

{

m\_null->color = BLACK;

m\_root = m\_null;

}

RBT::~RBT()

{

clear();

delete m\_null;

}

//找到从小到大排序后下标为i的节点。i从0开始。

RBTNode\* RBT::atIndex(int i)

{

RBTNode\* result = m\_root;

if (i > result->lcount + result->rcount)

{

result = NULL;

}

else

{

while (i != result->lcount)

{

if (i < result->lcount) //懂

{

result = result->lchild;

}

else

{

i -= result->lcount + 1;

result = result->rchild;

}

}

}

return result;

}

//删除一个节点

void RBT::del(RBTNode\* node) //?????????????

{

RBTNode\* toDel = node;

if (node->lchild != m\_null && node->rchild != m\_null)

{

toDel = treeNext(node); //?????????

}

RBTNode\* temp = toDel;

while (temp->parent != m\_null)

{

if (temp == temp->parent->lchild)

{

temp->parent->lcount--;

}

else

{

temp->parent->rcount--;

}

temp = temp->parent;

}

RBTNode\* replace = toDel->lchild != m\_null? toDel->lchild: toDel->rchild;

replace->parent = toDel->parent;

if (replace->parent == m\_null)

{

m\_root = replace;

}

else if (toDel == toDel->parent->lchild)

{

replace->parent->lchild = replace;

}

else

{

replace->parent->rchild = replace;

}

if (toDel != node)

{

node->key = toDel->key;

}

if (toDel->color == BLACK)

{

//修改树，以保持平衡。

delFixup(replace);

}

delete toDel;

}

void RBT::init()

{

clear();

m\_root = m\_null;

}

//插入一个节点

void RBT::insert(const Key& key) //懂

{

RBTNode\* node = new RBTNode;

node->key = key;

node->lcount = 0;

node->rcount = 0;

node->lchild = m\_null;

node->rchild = m\_null;

node->color = RED;

RBTNode\* p = m\_root;

RBTNode\* leaf = m\_null;

while (p != m\_null)

{

leaf = p;

if (keyCmp(node->key, p->key) < 0)

{

p->lcount++;

p = p->lchild;

}

else

{

p->rcount++;

p = p->rchild;

}

}

node->parent = leaf;

if (leaf == m\_null)//如果是空树。

{

m\_root = node;

}

else if (keyCmp(node->key, leaf->key) < 0)

{

leaf->lchild = node;

}

else {

leaf->rchild = node;

}

//修改树，以保持平衡。

insertFixup(node);

}

int RBT::nodeCount()

{

return m\_root != m\_null? m\_root->lcount + m\_root->rcount + 1: 0;

}

//按照key查找一个节点。

RBTNode\* RBT::search(const Key& key)

{

RBTNode\* result = m\_root;

while (result != m\_null && keyCmp(key, result->key) != 0)

{

result = keyCmp(key, result->key) < 0 ? result->lchild: result->rchild;

}

return result == m\_null? NULL: result;

}

//把树中节点的值放进一个数组。

void RBT::toArray(int\* array)

{

RBTNode\* p = treeMin(m\_root);

int i = 0;

while (p != m\_null)

{

array[i] = p->key.value;

i++;

p = treeNext(p);//按照中序遍历的顺序放入

}

}

//一个节点在中序遍列中的下一个节点。

RBTNode\* RBT::treeNext(RBTNode\* node)

{

RBTNode\* result;

if (node->rchild != m\_null)

{

result = treeMin(node->rchild);

}

else

{

result = node->parent;

RBTNode\* temp = node;

while (result != m\_null && temp == result->rchild)

{

temp = result;

result = result->parent;

}

}

return result;

}

//一个节点在中序遍列中的前一个节点。

RBTNode\* RBT::treePre(RBTNode\* node)

{

RBTNode\* result;

if (node->lchild != m\_null)

{

result = treeMax(node->lchild); //有修改

}

else

{

result = node->parent;

RBTNode\* temp = node;

while (result != m\_null && temp == result->lchild)

{

temp = result;

result = result->parent;

}

}

return result;

}

int main()

{

return 0;

}