

# 红黑树实验报告

## 一、实验目的

通过实践，加深理解红黑树的性质、特点，熟悉其相关操作，提高编程能力。

## 二、实验内容

- 1、实现红黑树。包括插入节点等基本操作。
- 2、实现将一百万个节点的红黑树写入硬盘，并从硬盘中恢复至内存的操作。

## 三、实验步骤

### 1、红黑树主要数据结构及其说明

#### 1) 红黑树节点类：RedBlackNode

数据成员：左右子节点的指针、父节点的指针、关键字、颜色

构造函数：默认构造函数会构造一个标志节点，带 Key 型参数的构造函数会构造一个关键字为指定参数的红色节点

```
class RedBlackNode {
public:
    RedBlackNode() {
        lChild = rChild = parent = NULL;
        key = -1;
        color = FLAG;
    }
    RedBlackNode(Key d) {
        lChild = rChild = parent = NULL;
        key = d;
        color = RED;
    }
    Node lChild;
    Node rChild;
    Node parent;
    Key key;
    Color color;
    //需要重载ofstream<<和ifstream>>
};
```

#### 2) 红黑树类：RedBlackTree

数据成员：其根节点

三种主要操作：插入节点，写入文件，从文件中读取。

```
void insertNode(Node n), void saveToFile(), void loadFromFile()
```

```
class RedBlackTree {
public:
    RedBlackTree() {
        root = NULL;
    }

    void saveToFile() {
    void loadFromFile() {

    void insertNode(Node n) {
    void printTree() {
private:
    Node root;

    void placeNode(Node t, Node tPrt, Node n) {
    void insert case1(Node n) {
    void insert case2(Node n) {
    void insert case3(Node n) {
    void insert case4(Node n) {
    void insert case5(Node n) {
    Node grandparent(Node n) {
    Node uncle(Node n) {
    void rotate right(Node n) {
    void rotate left(Node n) {
};
```

## 2、插入节点的实现

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。（参考自维基百科）

- 1) 通过下列函数，可以找到一个节点的叔父和祖父节点：

```

Node grandparent(Node n) {
    return n->parent->parent;
}
Node uncle(Node n) {
    if(n->parent == grandparent(n)->lChild)
        return grandparent(n)->rChild;
    else
        return grandparent(n)->lChild;
}

```

## 2) placeNode 函数

将新节点的关键字与树根节点关键字比较，如果关键字小于等于树根节点就将新节点递归地插入左子树，否则就递归地插入右子树。当碰到某个子树树根为 NULL 时，就说明已经到达了原树的叶节点，直接安置新节点在此处即可。t 为树根节点，tPrt 为 t 的父节点，n 为要插入的节点。在调用这个函数插入新节点时，只要写成 placeNode(root, NULL, n) 即可。

```

void placeNode(Node t, Node tPrt, Node n) {
    if(!t) {
        if(!tPrt)
            root = n;
        //插入新节点
        else if (n->key <= tPrt->key)
            tPrt->lChild = n;
        else
            tPrt->rChild = n;
        n->parent = tPrt;
        return;
    }
    else if (n->key <= t->key)
        placeNode(t->lChild, t, n);
    else
        placeNode(t->rChild, t, n);
}

```

## 3) 调整树以保证红黑树的性质不被破坏。（参考自维基百科）

。

情形 1：新节点 N 位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质 2。因为它在每个路径上对黑节点数目增加一，性质 5 符合。

```

void insert_case1(Node n) {
    if (n->parent == NULL)
        n->color = BLACK;
    else{
        insert_case2(n);
    }
}

```

情形 2: 新节点的父节点 P 是黑色, 所以性质 4 没有失效 (新节点是红色的)。在这种情形下, 树仍是有效的。性质 5 也未受到威胁, 尽管新节点 N 有两个黑色叶子子节点; 但由于新节点 N 是红色, 通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点, 所以依然满足这个性质。

```

void insert_case2(Node n) {
    if (n->parent->color == BLACK)
        return;
    else {
        insert_case3(n);
    }
}

```

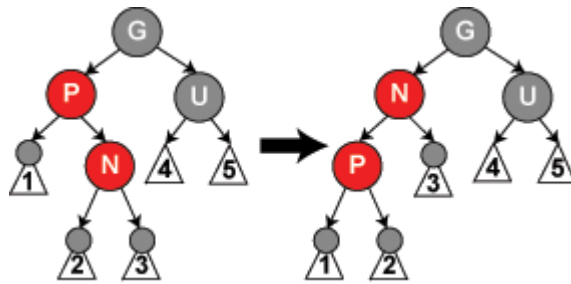
情形 3: 如果父节点 P 和叔父节点 U 二者都是红色, (此时新插入节点 N 做为 P 的左子节点或右子节点都属于情形 3, 这里右图仅显示 N 做为 P 左子的情形) 则我们可以将它们两个重绘为黑色并重绘祖父节点 G 为红色。现在我们的新节点 N 有了一个黑色的父节点 P。因为通过父节点 P 或叔父节点 U 的任何路径都必定通过祖父节点 G, 在这些路径上的黑节点数目没有改变。但是, 红色的祖父节点 G 的父节点也有可能是红色的, 这就违反了性质 4。为了解决这个问题, 我们在祖父节点 G 上递归地进行情形 1 的整个过程。

```

void insert_case3(Node n) {
    if (uncle(n) != NULL && uncle(n)->color == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(grandparent(n));
    }
    else {
        insert_case4(n);
    }
}

```

情形 4: 父节点 P 是红色而叔父节点 U 是黑色或缺少, 并且新节点 N 是其父节点 P 的右子节点而父节点 P 又是其父节点的左子节点。在这种情形下, 我们进行一次左旋转调换新节点和其父节点的角色; 接着, 我们按情形 5 处理以前的父节点 P 以解决仍然失效的性质 4。注意这个改变会导致某些路径通过它们以前不通过的新节点 N 或不通过节点 P, 但由于这两个节点都是红色的, 所以性质仍有效。

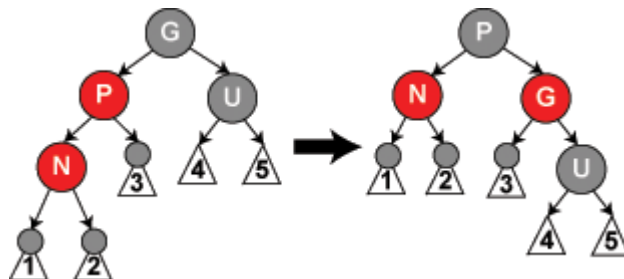


```

void insert_case4(Node n) {
    if (n == n->parent->rChild &&
        n->parent == grandparent(n)->lChild) {
        rotate_left(n->parent);
        n = n->lChild;
    } else if (n == n->parent->lChild
        && n->parent == grandparent(n)->rChild) {
        rotate_right(n->parent);
        n = n->rChild;
    }
    insert_case5(n);
}

```

情形 5: 父节点 P 是红色而叔父节点 U 是黑色或缺少, 新节点 N 是其父节点的左子节点, 而父节点 P 又是其父节点 G 的左子节点。在这种情形下, 我们进行针对祖父节点 G 的一次右旋转; 在旋转产生的树中, 以前的父节点 P 现在是新节点 N 和以前的祖父节点 G 的父节点。我们知道以前的祖父节点 G 是黑色, 否则父节点 P 就不可能是红色。我们切换以前的父节点 P 和祖父节点 G 的颜色, 结果的树满足性质 4。性质 5 也仍然保持满足, 因为通过这三个节点中任何一个的所有路径以前都通过祖父节点 G, 现在它们都通过以前的父节点 P。在各自的情形下, 这都是三个节点中唯一的黑色节点。



```

void insert_case5(Node n) {
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->lChild &&
        n->parent == grandparent(n)->lChild) {
        rotate_right(grandparent(n));
    } else {
        rotate_left(grandparent(n));
    }
}

```



#### 4) 将节点右旋或左旋的两个函数

```
void rotate_right(Node n){
    Node successor = n->lChild;
    Node midChild = successor->rChild;
    Node ancestor = n->parent;
    successor->rChild = n;
    n->lChild = midChild;
    if(midChild)
        midChild->parent = n;
    n->parent = successor;
    successor->parent = ancestor;
    if(n!=root){
        if(ancestor->rChild == n){
            ancestor->rChild = successor;
        } else {
            ancestor->lChild = successor;
        }
    } else {
        root = successor;
    }
}

void rotate_left(Node n){
    Node successor = n->rChild;
    Node midChild = successor->lChild;
    Node ancestor = n->parent;
    successor->lChild = n;
    n->rChild = midChild;
    if(midChild)
        midChild->parent = n;
    n->parent = successor;
    successor->parent = ancestor;
    if(n!=root){
        if(ancestor->lChild == n){
            ancestor->lChild = successor;
        } else {
            ancestor->rChild = successor;
        }
    } else {
        root = successor;
    }
}
```

#### 5) 最终的插入函数

```
void insertNode(Node n){
    //先放置节点在合适的叶节点位置
    placeNode(root, NULL, n);
    //然后调节节点位置
    insert_case1(n);
}
```

## 2、 写入文件与读出文件的实现

写入文件时，非递归前序遍历红黑树，基本步骤是：

1) 根节点入栈 2) 从栈中取出一个节点，写入文件。 3) 将 2 中取出节点的右子节点入栈，再将其左子节点入栈。如果左子节点为空，则不作任何操作，如果右子节点为空，则将一个标志节点入栈 4) 重复 2 直到栈为空

从文件中读取时，维护一个节点栈，初始时空。栈顶节点记作 B。执行下面的步骤：

1) 如果已经到达文件末尾则终止程序，否则执行 2 2) 从文件中读取一个节点的数据，并构造该节点 A，执行 3 3) 如果 A 是标志节点，就弹出栈顶节点，执行 1。否则执行 4 4) 如果栈中无节点，将 A 作为树根，执行 6。否则执行 5 5) 如果 B 还没有子节点，A 作为 B 的左子节点，执行 6；如果 B 已有左子节点，A 作为 B 的右子节点，执

行 6：如果 B 已有两个子节点，就将 B 从栈中弹出，将栈顶节点作为新的 B（注：此处栈是不可能为空的），并重复此步。 6）将 A 入栈，执行 1

```
void saveToFile() {
    Node flagNode = new RedBlackNode();
    ofstream fout;
    fout.open("RedBlackTree.dat", ios::binary);

    Node t = root;
    stack<Node> st;
    if (t) {
        st.push(t);
        while (!st.empty()) {
            t = st.top();
            st.pop();
            Node lc = t->lChild;
            Node rc = t->rChild;
            t->lChild = t->rChild = t->parent = NULL;
            fout << *t;
            if (t->color==FLAG) continue;
            if (rc)
                st.push(rc);
            else st.push(flagNode);
            if (lc)
                st.push(lc);
        }
    }
    fout.close();
}
```

```

void loadFromFile() {
    stack<Node> st;
    ifstream fin;
    fin.open("RedBlackTree.dat", ios::binary);
    if(fin.eof()){
        cout << "Data file does not exist.\n";
        return;
    }
    Node r = new RedBlackNode();
    fin >> *r;
    root = r;
    st.push(r);

while(!fin.eof()){
    if(st.empty()){
        if(TEST)
            cout << "warning: stack deprecated.\n";
        return;
    }
    Node n = new RedBlackNode();
    fin >> *n;
    if(n->color==FLAG) {
        st.pop();
    } else {
        Node p;
        int children;
        do{

            p = st.top();
            children = 0;
            if(p->lChild) ++children;
            if(p->rChild) ++children;
            if(TEST)
                cout << "children: " << children << "\n";
            if(children!=2) {
                n->parent = p;
                break;
            }
            st.pop();
        }while(true);
        if(children==0){
            p->lChild = n;
        } else {
            p->rChild = n;
        }
        st.push(n);
    }
}
fin.close();
}

```



### 3. 红黑树的测试

printTree 函数以输出整棵树:

```
void printTree(){
    int count = 0;
    Node t = root;
    stack<Node> st;
    if (t) {
        st.push(t);
        while (!st.empty()) {
            t = st.top();
            st.pop();
            ++count;
            if(TEST)
                cout << count << " " << t->key << (t->color==RED?'r':'b') << " \t";
            if (t->rChild)
                st.push(t->rChild);
            if (t->lChild)
                st.push(t->lChild);
        }
    }
    cout << "count: " << count << endl;
}
```

驱动函数: 构造一棵红黑树, 调用 10 次树的 insertNode 操作, 在再用 saveToFile 和 loadFromFile 操作, 并在每次调用后打印树, 根据树的形态判断代码是否正常工作。

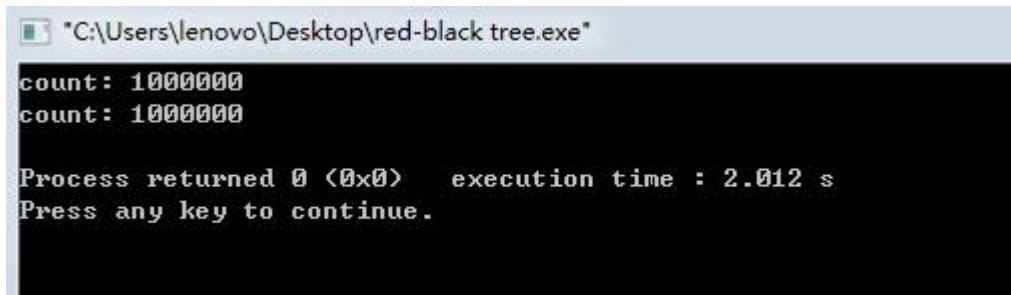
```
int main(){
    srand((unsigned)time(0));
    RedBlackTree tree;
    for(int i=0;i<1000000;++i){//树的大小为100,0000
        int k = rand()%100;
        //cout << "node " << k << " inserted\n";
        tree.insertNode(new RedBlackNode(k));
    }
    tree.printTree();
    tree.saveToFile();
    ifstream fin;
    fin.open("RedBlackTree.dat",ios::binary);
    tree.loadFromFile();
    tree.printTree();
    return 0;
}
```

## 四、 实验结果

程序构建了一个 1000,000 节点的红黑树, 写入磁盘, 并从磁

盘中读出到内存。

不打印出结果： 耗时约 2s



```
"C:\Users\lenovo\Desktop\red-black tree.exe"  
count: 1000000  
count: 1000000  
  
Process returned 0 (0x0)   execution time : 2.012 s  
Press any key to continue.
```

打印结果，因节点过点过于耗时，而且难以观察，因此用 20 个节点作为测试，以便于查看。

```
C:\Users\jenovo\Desktop\red-black tree.exe
1 56b  2 42b  3 11r  4 9b   5 11r  6 27b  7 18r  8 31r  9 50b 10 45r
11 51r 12 79b 13 68r 14 60b 15 67r 16 72b 17 71r 18 74r 19 90b 20 89r
count: 20
node written: 561
node written: 421
node written: 110
node written: 91
node written: 110
node written: -12
node written: 271
node written: 180
node written: -12
node written: 310
node written: -12
node written: 501
node written: 450
node written: -12
node written: 510
node written: -12
node written: 791
node written: 680
node written: 601
node written: 670
node written: -12
node written: 721
node written: 710
node written: -12
node written: 740
node written: -12
node written: 901
node written: 890
node written: -12
node written: -12
node read: 561
node read: 421
children: 0
node read: 110
children: 0
node read: 91
children: 0
node read: 110
children: 0
```

```

node read: 110
children: 0
node read: -12
node read: 271
children: 1
node read: 180
children: 0
node read: -12
node read: 310
children: 1
node read: -12
node read: 501
children: 2
children: 2
children: 1
node read: 450
children: 0
node read: -12
node read: 510
children: 1
node read: -12
node read: 791
children: 2
children: 2
children: 1
node read: 680
children: 0
node read: 601
children: 0
node read: 670
children: 0
node read: -12
node read: 721
children: 1
node read: 710
children: 0
node read: -12
node read: 740
children: 1
node read: -12
node read: 901

```

```

node read: -12
node read: 901
children: 2
children: 2
children: 1
node read: 890
children: 0
node read: -12
node read: -12
node read: -12
1 56b  2 42b  3 11r  4 9b   5 11r  6 27b  7 18r  8 31r  9 50b 10 45r
11 51r 12 79b 13 68r 14 60b 15 67r 16 72b 17 71r 18 74r 19 90b 20 89r
count: 20

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.

```

## 五、 实验心得

本次实验，感觉有不小难度，网上有不少例子，也使用了维基百科上现成的函数，还是遇到很多问题。在文件输入输出部分很多不太记得，重新学习后，才正确的使用。实验中，也向其

他同学寻求了帮助，解决问题，加快了工作效率，也学到了更多的知识。

实验中，没有编写代码测试红黑树与二叉搜索树的性能进行对比。但可以看出，二叉搜索树在查找方面提供了很大的方便，但是对最坏情况的查找、插入、删除、求最值的时间复杂度都是  $O(n)$ 。红黑树可以通过在插入节点时进行旋转来调整高度，以使整棵树的状态保持相对平衡，从而在最坏情况下的查找、插入、删除、求最值的时间复杂度是  $O(\log n)$ 。红黑树的查找性能优于二叉查找树的查找性能。