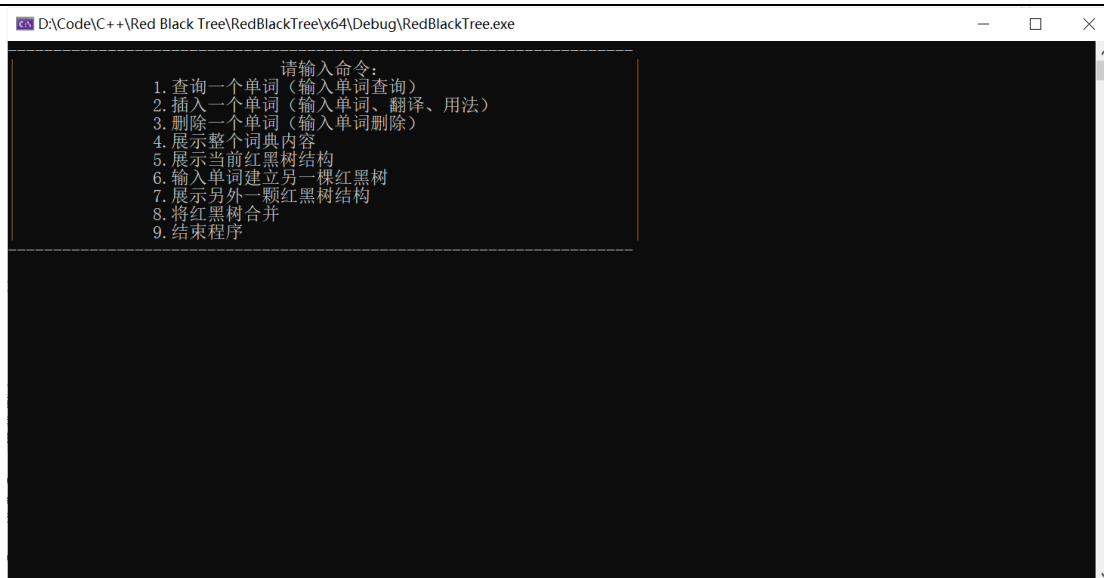


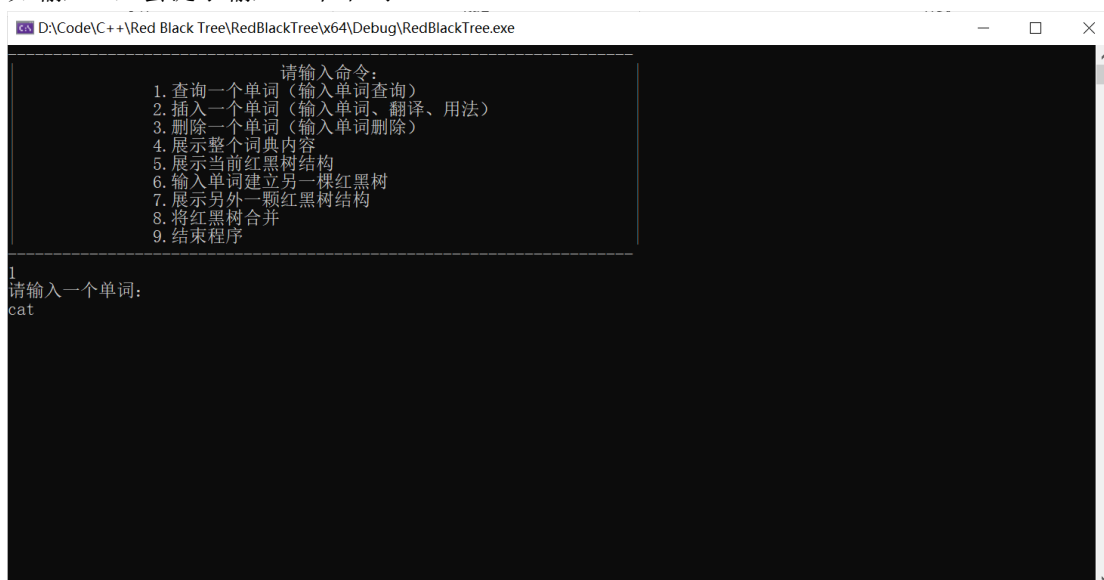
山东大学计算机科学与技术学院

数据结构与算法课程设计报告

学号：202000130109	姓名：李正华	班级：计机 20.1
上机学时：10	日期：2022. 3. 3-3. 17、5. 13-5. 20	
课程设计题目：红黑树的实现与分析		
软件环境：Visual Studio 2022		
语言环境：C++ 11，使用了 C++的扩展库 easyx. h		
报告内容：		
1. 需求描述		
1.1 问题描述		
红黑树（Red Black Tree） 是一种自平衡二叉搜索树，是在进行插入和删除操作时通过特定操作保持二叉搜索树的平衡，从而获得较高的搜索性能。		
1.2 基本要求		
（1）设计并实现红黑树（Red-Black Tree）的 ADT，该 ADT 包括 Tree 的组织存储以及其上的基本操作：包括初始化，查找，插入和删除等。并分析基本操作的时间复杂性。		
（2）实现 Red-Black Tree ADT 的基本操作演示（鼓励应用图形界面）。		
（3）采用红黑树，编写一个小型的英汉词典索引，并实现简单的检索功能，词典可以手工建立，也可以网上寻找。		
*（4）实现红黑树的合并操作，并应用于字典的合并。		
1.3 输入说明		
首先输入单词个数，单词、翻译和用法，以输入的单词为基础建立红黑树结构的词典		
<div><div>D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe</div><div><div>请输入数据建立词典： 格式为:单词、翻译、用法</div><div>请输入单词个数： 5 请输入第1个单词及其翻译和用法： apple 苹果 An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh. 请输入第2个单词及其翻译和用法： bird 小鸟 A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly. 请输入第3个单词及其翻译和用法： dog 狗 The British are renowned as a nation of dog lovers. 请输入第4个单词及其翻译和用法： cat 猫 Cats are lions, tigers, and other wild animals in the same family. 请输入第5个单词及其翻译和用法： egg 鸡蛋 ...a baby bird hatching from its egg. _</div></div></div>		
然后会出现命令菜单，根据命令菜单输入命令		



如输入 1，会提示输入一个单词



如输入 2，会提示输入一个单词以及其翻译和用法

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

2
请输入一个单词、翻译、用法：
general 将军 The General's visit to Sarajevo is part of preparations for the deployment of extra troops. _
```

如输入 3，会提示输入一个单词，会将这个单词从词典中删除

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

3
请输入一个单词：
cat _
```

如输入 6，会提示输入单词数，以及对应数量的单词、翻译、用法，以此新建一个词典

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入数据建立词典：
格式为:单词、翻译、用法

请输入单词个数：
5
请输入第1个单词及其翻译和用法：
figure 数字 It will not be long before the inflation figure starts to fall.
请输入第2个单词及其翻译和用法：
general 将军 The General's visit to Sarajevo is part of preparations for the deployment of extra troops.
请输入第3个单词及其翻译和用法：
hit 击打 Find the exact grip that allows you to hit the ball hard.
请输入第4个单词及其翻译和用法：
issue 问题 Agents will raise the issue of prize-money for next year's world championships.
请输入第5个单词及其翻译和用法：
journey 旅行 As per our rule, our bus can not stop at her journey, if you have anything important, please get off the
bus at stop.
```

1.4 输出说明

当输入对应命令和参数的时候，回有对应的输出。

比如当输入命令 1 和单词 apple，会查询到 apple 的翻译和用法并且输出

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令：
1. 查询一个单词 （输入单词查询）
2. 插入一个单词 （输入单词、翻译、用法）
3. 删除一个单词 （输入单词删除）
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

1
请输入一个单词：
apple

单词: apple
翻译: 苹果
用法: An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.

请按任意键继续. . .
```

当输入命令 2，以及单词 kind 的翻译和用法时，会提示插入成功

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

2
请输入一个单词、翻译、用法：
kind 种类 The party needs a different kind of leadership.
插入成功！

请按任意键继续. . .
```

如输入命令 3，以及对应单词，删除成功后会提示

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

3
请输入一个单词：
apple
删除成功！

请按任意键继续. . .
```

如输入命令 4，会按照单词的字典序输出字典中所有单及其翻译和用法

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

- 请输入命令：
1. 查询一个单词（输入单词查询）
 2. 插入一个单词（输入单词、翻译、用法）
 3. 删除一个单词（输入单词删除）
 4. 展示整个词典内容
 5. 展示当前红黑树结构
 6. 输入单词建立另一棵红黑树
 7. 展示另外一颗红黑树结构
 8. 将红黑树合并
 9. 结束程序

4

单词: bird
翻译: 小鸟
用法: A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.

单词: dog
翻译: 狗
用法: The British are renowned as a nation of dog lovers.

单词: egg
翻译: 鸡蛋
用法: ...a baby bird hatching from its egg.

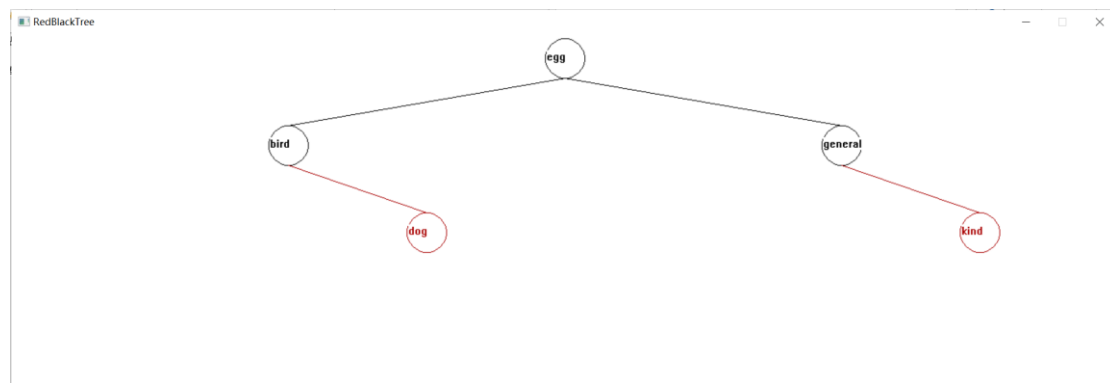
单词: general
翻译: 将军
用法: The General's visit to Sarajevo is part of preparations for the deployment of extra troops.

单词: kind
翻译: 种类
用法: The party needs a different kind of leadership.

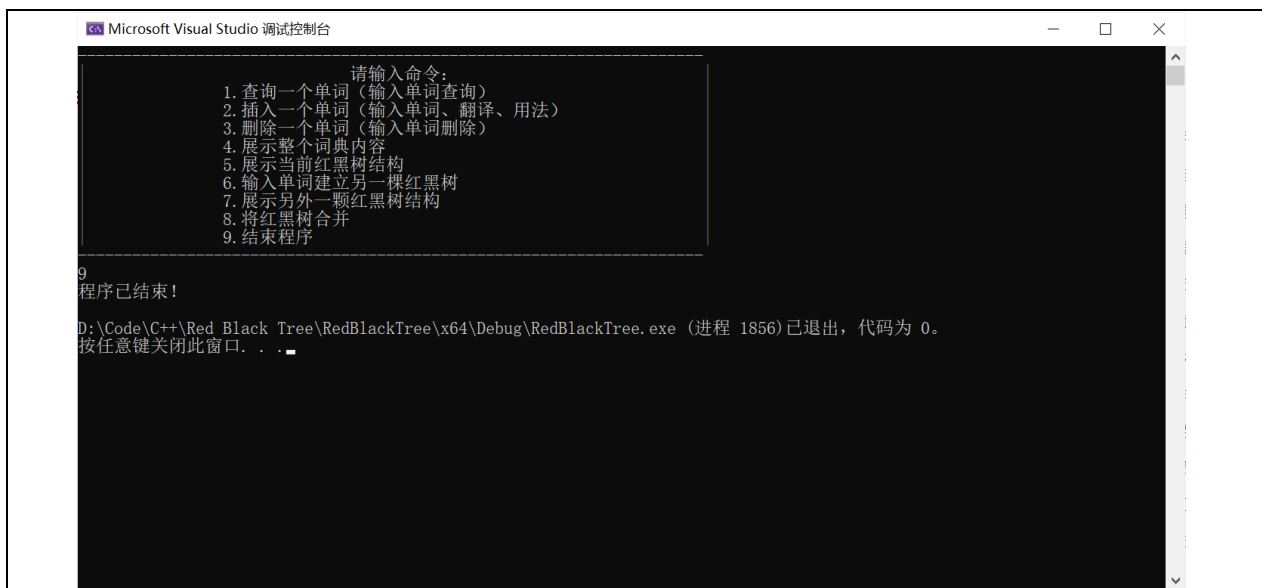
打印成功！

请按任意键继续. . .

如输入命令 5，会输出一个图形化的界面，展示当前词典的红黑树结构



如输入命令 9，则会将程序结束并且输出提示语句



其他命令会有不同的输出，仅展示部分。

2. 分析与设计

2.1 问题分析

首先，需要实现红黑树（Red-Black Tree）的 ADT，实现包括初始化，查找，插入和删除等基本操作函数，定义为红黑树的实现模块。

其次，需要实现 Red-Black Tree ADT 的基本操作演示，利用图形化界面画出整颗红黑树的结构，标明树结构、节点颜色、节点键值等信息，对基本操作进行可视化的展示。定义为基本操作操作演示模块。

然后，需要利用红黑树编写一个小型的英汉词典索引，并且实现简单的检索功能。其中单词包括英文单词、中文翻译、用法例句，以英文单词为主键，形成一个结构体插入到红黑树中。定义为英汉词典实现模块。

最后，需要实现红黑树的合并，将两颗红黑树合并为同一棵红黑树，并且应用到字典合并当中，创建两个词典并且将其合并。定义为红黑树合并模块

综上：程序主要分为四个模块的实现，对于每个模块分为多个小型函数模块实现，模块结构如下：

（1） 红黑树的实现模块

- ① 节点查找
- ② 节点插入
- ③ 节点删除
- ④ 插入调整
- ⑤ 删除调整
- ⑥ 以节点为中心左旋
- ⑦ 以节点为中心右旋
- ⑧ 将节点位置替换
- ⑨ 寻找子树中最小节点
- ⑩ 销毁整颗红黑树

（2） 基本操作演示模块

（3） 英汉词典实现模块

- ① 词典建立
- ② 词典展示
- ③ 词典基本功能（查询索引、插入、删除）

(4) 红黑树合并模块

2.2 主程序设计

程序进入之后会展示命令菜单，首先需要输入单词建立一棵基于红黑树建立的小型英汉词典索引，需要输入单词个数和每个单词的英文、中文翻译以及用法例句。

然后根据命令进行操作，命令涉及问题中的四个实现模块。命令如下：

- 1 查询一个单词。输入单词从红黑树建立的词典中查询一个单词并且输出其单词、翻译、用法。
- 2 插入一个单词。输入单词、翻译、用法向基于红黑树建立的词典中插入一个单词。
- 3 删除一个单词。输入单词从红黑树建立的词典中删除一个单词。
- 4 展示整个词典结构。按照字典序输出红黑树建立的字典里的所有单词及其翻译和用法。
- 5 展示红黑树的结构。利用图形化界面展示当前红黑树的实际结构（包含节点、父子关系、单词等），利用插入或删除前后展示图形化界面来进行红黑树基本操作的演示
- 6 输入建立另一棵红黑树。输入单词建立另一棵红黑树，为合并做准备。
- 7 展示另外一棵红黑树的结构。利用图形化界面展示另一棵红黑树的实际结构（包含节点、父子关系、单词等），利用插入或删除前后展示图形化界面来进行红黑树基本操作的演示
- 8 红黑树合并。将当前红黑树和另外一棵红黑树进行合并，合并到当前红黑树中。在合并前后图形化界面展示红黑树的实际结构来进行红黑树合并操作的演示
- 9 结束程序。

主程序代码如下：

```
int main() {
    RedBlackTree t,t1;//两颗红黑树
    cout << "-----\n"
           "|               请输入数据建立词典:               |\n"
           "|               格式为:单词、翻译、用法               |\n"
           "-----\n";

    int n;//单词个数
    cout << "请输入单词个数:\n";
    cin >> n;
    for (int i = 1; i <= n; i++) { //输入单词、翻译、用法，并且插入红黑树
        cout << "请输入第" << i << "个单词及其翻译和用法:\n";
        string key, translation;
        cin >> key >> translation;
        string read;
        getline(cin, read, '\n');
        stringstream ss(read);
        vector<string> usage;
        while (ss >> read) {
            usage.push_back(read);
        }
        bool f = t.insert(word{ key, translation, usage },0);
    }
```



```

        if (f == false)
            cout << "插入失败，存在重复值！";
    }
    while (1) {
        system("cls");//清空，放出菜单
        cout << "-----\n"
            |
            |          请输入命令：          |
            |          1.查询一个单词（输入单词查询）          |
            |          2.插入一个单词（输入单词、翻译、用法）          |
            |          3.删除一个单词（输入单词删除）          |
            |          4.展示整个词典内容          |
            |          5.展示当前红黑树结构          |
            |          6.输入单词建立另一棵红黑树          |
            |          7.展示另外一颗红黑树结构          |
            |          8.将红黑树合并          |
            |          9.结束程序          |
            |          "-----\n";
        string ord;//命令
        cin >> ord;
        if (ord == "1") { //查找单词
            string word1;
            cout << "请输入一个单词：\n";
            cin >> word1;
            Node* now = t.find(word1);
            if (now != NULL) { //查找成功，输出
                cout
                <<
                "\n-----\n" << t.find(word1)->value <<
                "-----\n";
            }
            else //查找失败
                cout << "抱歉，未查询到该单词！\n";
        }
        else if (ord == "2") { //插入单词
            string key, translation;
            cout << "请输入一个单词、翻译、用法：\n";
            cin >> key >> translation;
            string read;
            getline(cin, read, '\n');
            stringstream ss(read);
            vector<string> usage;
            while (ss >> read) {
                usage.push_back(read);
            }
            bool flag = t.insert(word{ key, translation, usage }, 0);

```

```

        if (flag)//插入成功
            cout << "插入成功！ \n";
        else//插入失败
            cout << "插入失败， 存在重复单词！ \n";
    }
    else if (ord == "3") {//删除单词
        string word1;
        cout << "请输入一个单词： \n";
        cin >> word1;
        bool flag = t.erase(word1);
        if (flag)//删除成功
            cout << "删除成功！ \n";
        else//删除失败
            cout << "抱歉， 删除失败！ \n";
    }
    else if (ord == "4") {//打印词典结构
        t.print(t.getRoot());
        cout << "打印成功！ \n";
    }
    else if (ord == "5") {//展示红黑树结构
        t.show();
    }
    else if (ord == "6") {//形成另一颗红黑树
        system("cls");
        cout << "-----\n"
              "|                请输入数据建立词典:                |\n"
              "|                格式为:单词、翻译、用法                |\n"
              "-----\n";
        cout << "请输入单词个数:\n";
        cin >> n;
        for (int i = 1; i <= n; i++) {
            cout << "请输入第" << i << "个单词及其翻译和用法: \n";
            string key, translation;
            cin >> key >> translation;
            string read;
            getline(cin, read, '\n');
            stringstream ss(read);
            vector<string> usage;
            while (ss >> read) {
                usage.push_back(read);
            }
            bool f = t1.insert(word{ key, translation, usage }, 0);
            if (f == false)
                cout << "插入失败， 存在重复值！ ";
        }
    }
}

```

```

    }
}
else if (ord == "7") { //展示另外一棵红黑树
    t1.show();
}
else if (ord == "8") { //将另外一颗红黑树合并到当前红黑树
    t.merge(t1);
    cout << "合并成功! \n";
} else if (ord == "9") { //结束程序
    cout << "程序已结束! \n";
    break;
}
else { //错误命令
    cout << "请输入正确的命令! \n";
}
cout << "\n\n";
system("pause");//暂停
}
}

```

2.3 设计思路

(1) 红黑树的实现模块实现思路:

① 节点查找——RedBlackTree::find() 函数:

从红黑树的根开始遍历, 判断当前节点是否为查询的目标, 如果是, 则将当前节点的指针返回; 如果目标值小于当前节点, 则遍历节点变为当前节点的左孩子; 如果目标值大于当前节点, 则遍历节点变为当前节点的右孩子。如果一直查询到外部节点还没有找到, 则返回 NULL 表示没有查询到

② 节点插入——RedBlackTree::insert() 函数:

从红黑树的根开始遍历, 如果插入值小于当前节点, 则遍历节点变为当前节点的左孩子; 如果插入值大于当前节点, 则遍历节点变为当前节点的右孩子, 一直遍历到外部节点, 在外部节点的位置插入新节点, 并且对树结构进行维护。然后调用 insertBalance () 函数进行插入平衡, 维护红黑树的五条性质进行自平衡。

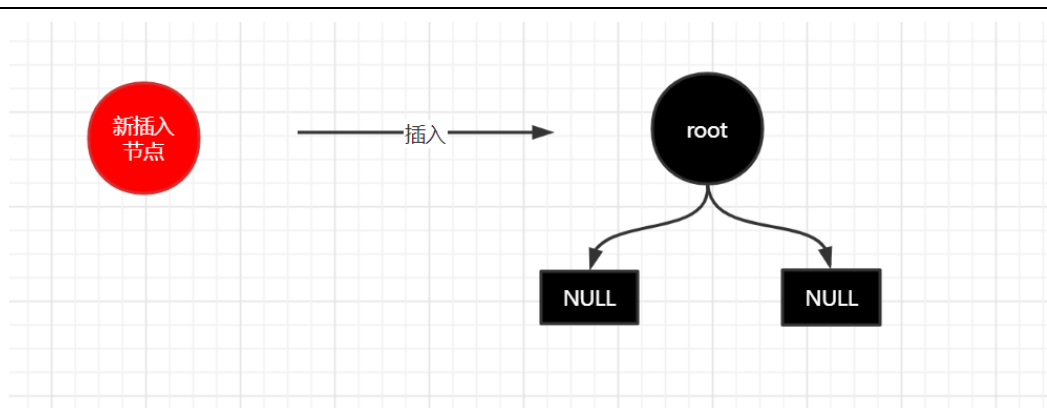
③ 节点删除——RedBlackTree::erase() 函数:

首先调用 find () 函数查询是否存在该节点, 如果不存在则不进行后续删除操作。否则, 利用 change 函数将目标节点更换删除, 并且维护前后的节点结构。然后调用 eraseBalance () 函数进行删除平衡, 维护红黑树的五条性质进行自平衡。

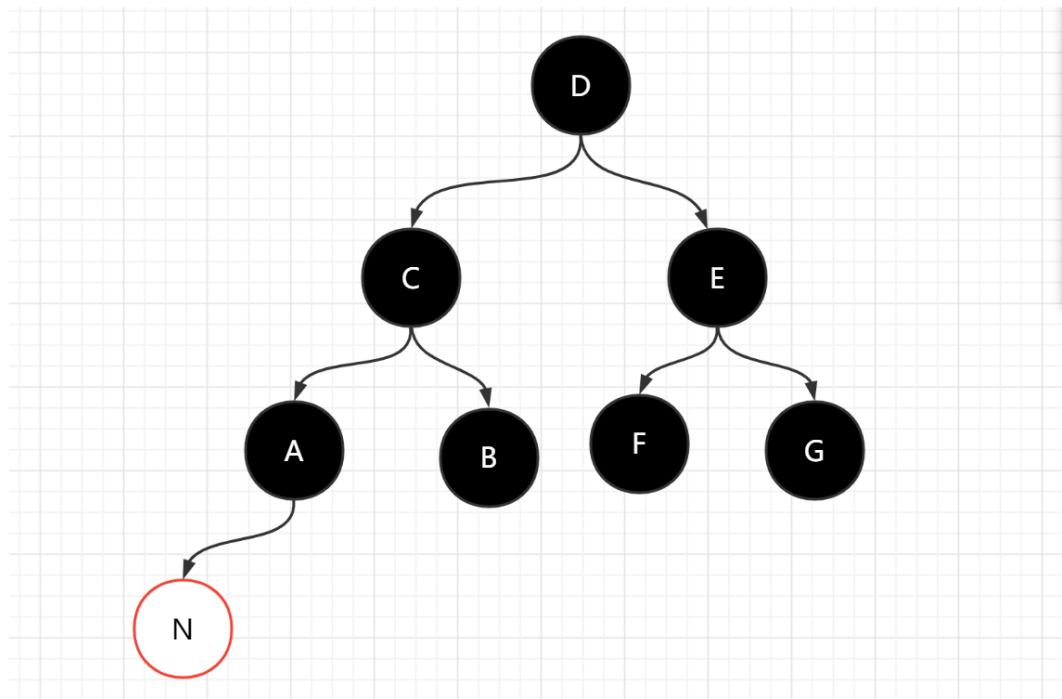
④ 插入调整——RedBlackTree::insertBalance() 函数:

进行红黑树的插入调整, 维护红黑树的自平衡, 需要分为以下五种情况进行处理:

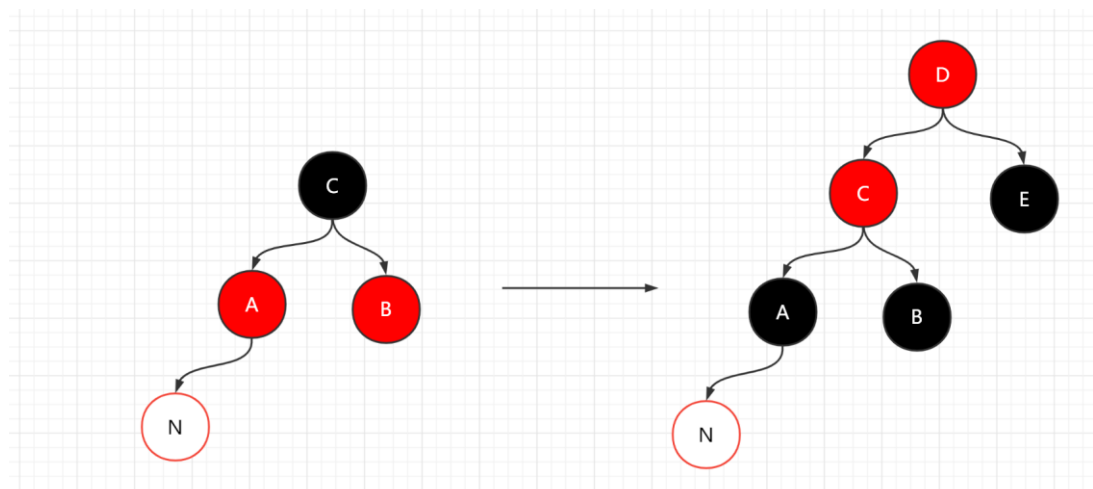
情况一: 当红黑树为空树的时候, 新插入的红色节点为根节点, 需要变为黑色



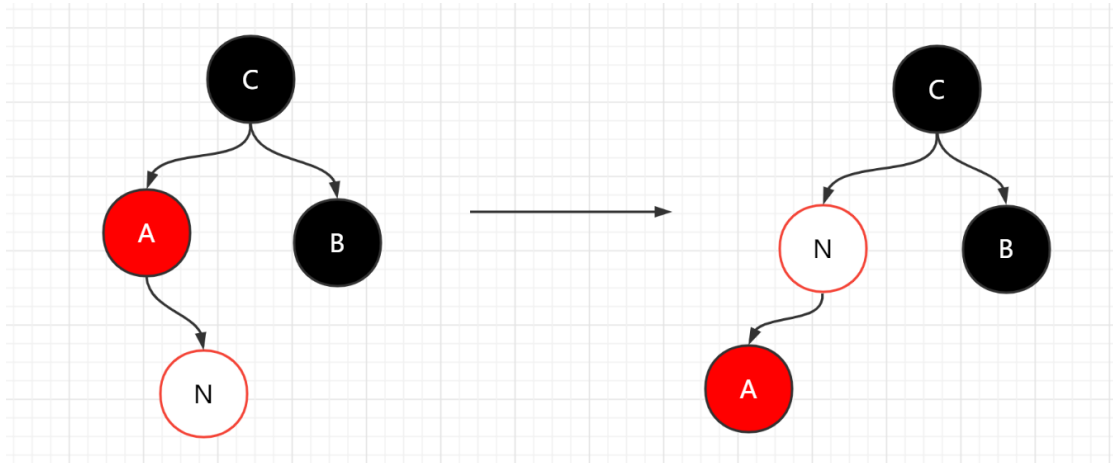
情况二：新插入的红色节点的父亲节点为黑色，不会影响红黑树的平衡



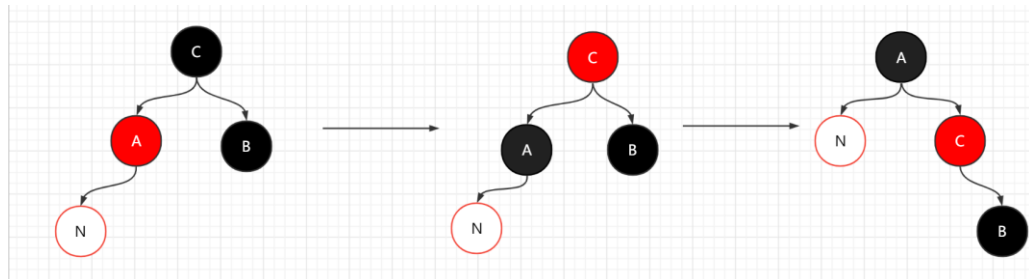
情况三：新插入的红色节点的父亲节点为红色，若叔叔节点为红色则祖父节点一定为黑色，将父节点和叔叔节点变黑，祖父节点变红，因此红色节点上移，下一步看情况处理



情况四：新插入红色节点的父亲节点为红色，叔叔节点为黑色，新节点在父亲节点右边，祖父一定为黑色。以父亲节点为中心左旋，这样新节点就在父亲节点左边了，转入情况五处理



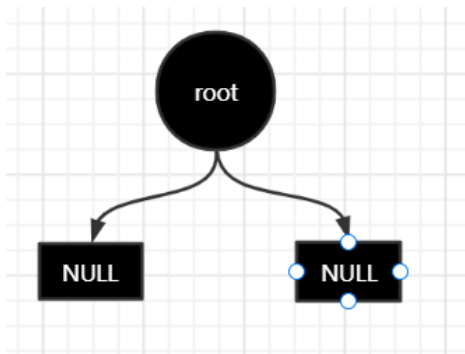
情况五：新插入的红色节点的父节点为红色，叔叔节点为黑色，新节点在父节点左边，祖父一定为黑色父亲节点变为黑色，此时多了一个黑色节点，因此祖父节点变为红色，以祖父节点为中心右旋，抵消这个黑色节点。



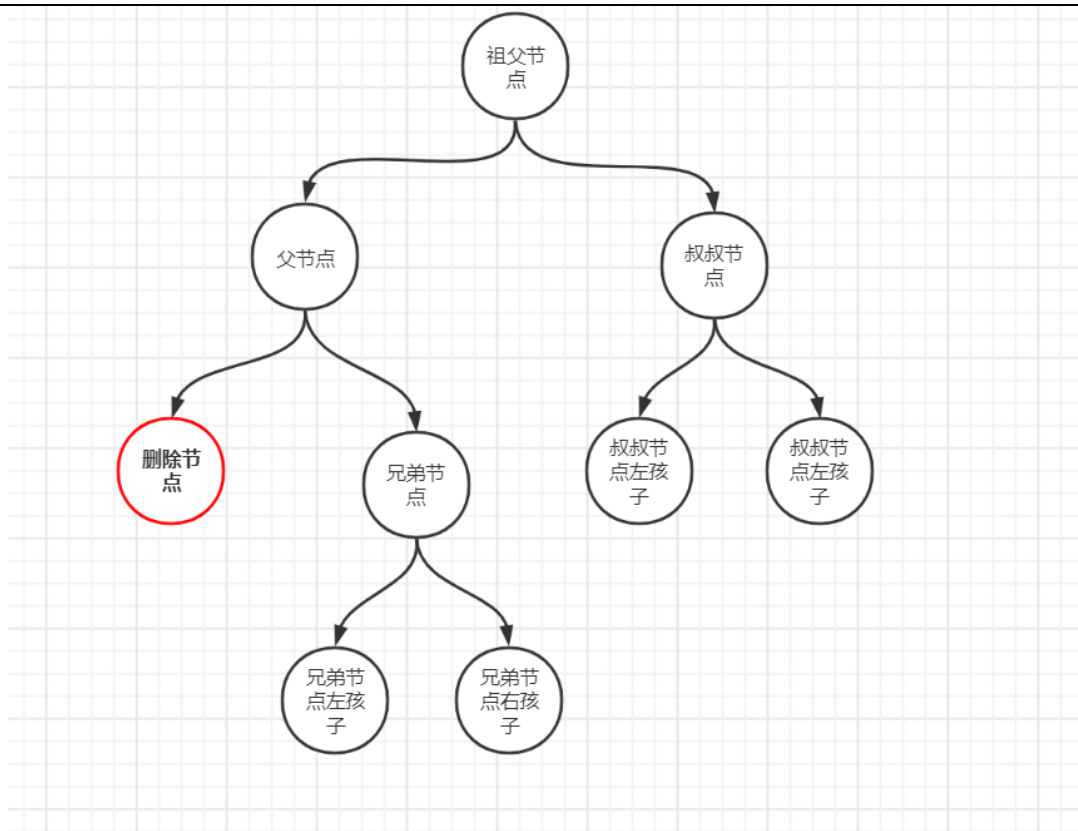
⑤ 删除调整——RedBlackTree::eraseBalance() 函数：

进行红黑树的删除调整，维护红黑树的自平衡，需要分为以下六种情况进行处理：

情况一：当被删除节点为黑并且为根节点，无需调整

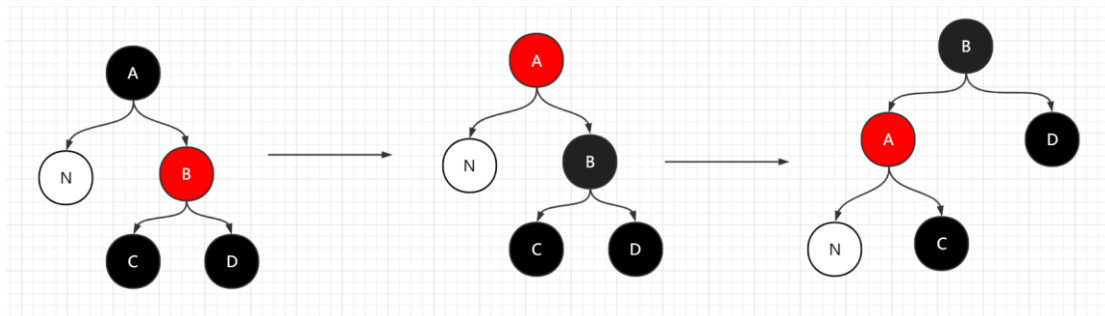


情况二：被删除节点为红色，不违反任何规则，无需调整



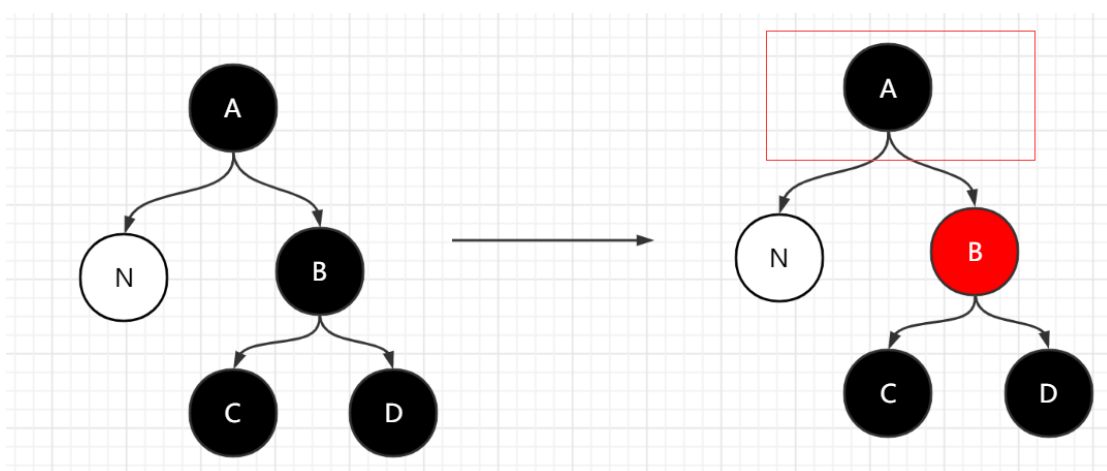
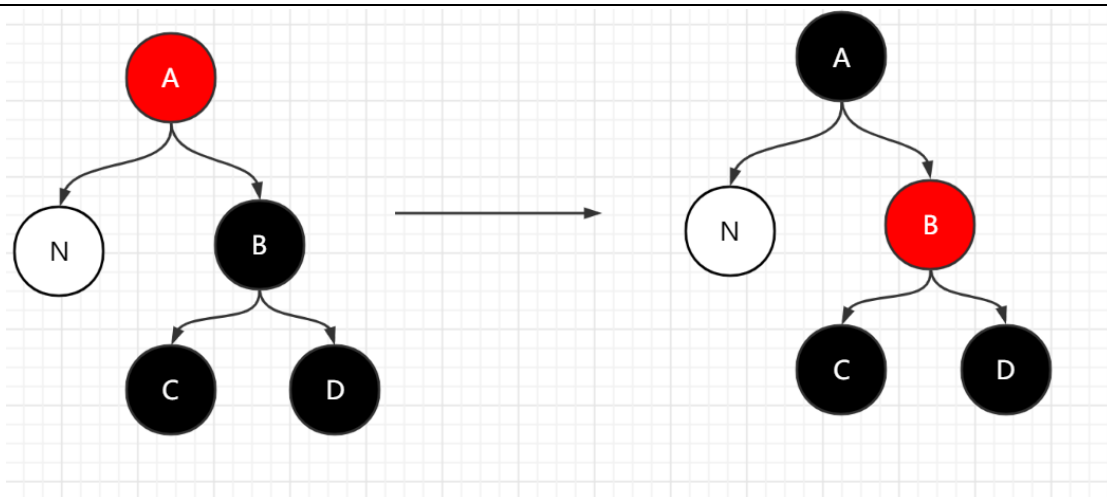
情况三：兄弟节点为红色。

这种情况不好处理，因为无法通过上移兄弟节点的颜色来调整。因此将兄弟节点变为黑色，父节点变为红色，以父节点为中心左旋，此时新的兄弟节点必为黑色，为后面调整做好准备。



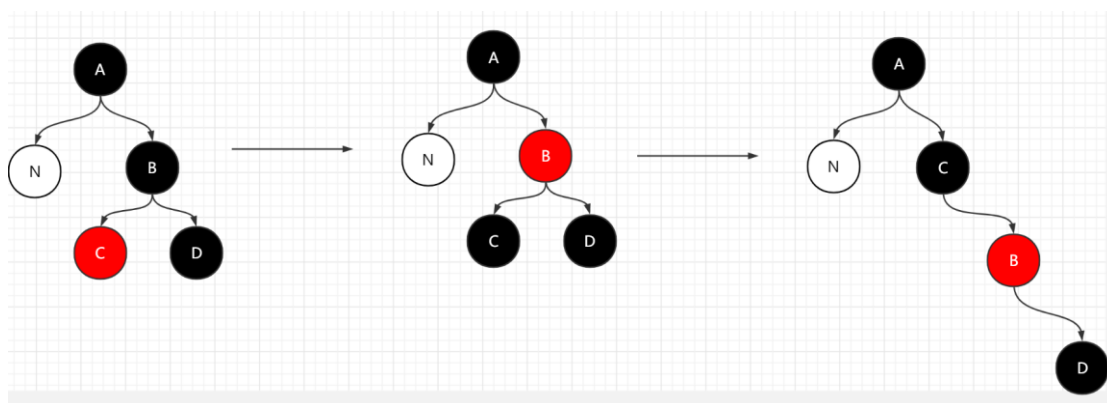
情况四：兄弟节点为黑色，并且兄弟节点两个孩子也是黑色。

看父亲节点为何颜色，如果为红色，将其变为黑色，叔叔节点变为红色，调整结束；如果为黑色，叔叔节点变为红色，新的调整节点转到父节点位置继续向上调整



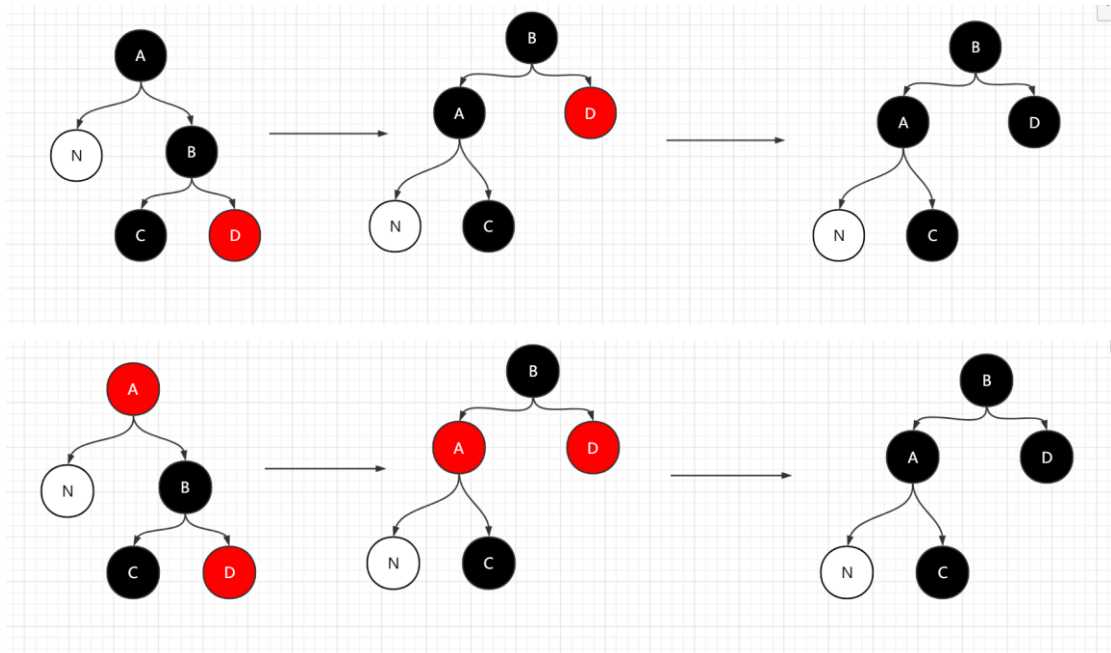
情况五：兄弟节点为黑色，并且兄弟节点右孩子为红色，左孩子任意

将兄弟节点颜色变为其父亲节点颜色，以父亲节点为中心左旋，父亲节点和兄弟节点的右孩子都变黑



情况六：兄弟节点为黑色，并且兄弟节点右孩子为红色，左孩子任意

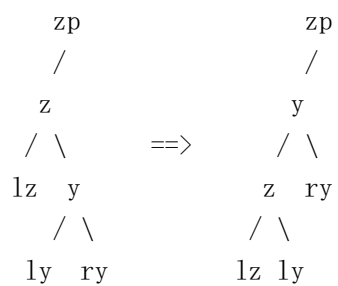
将兄弟节点颜色变为其父亲节点颜色，以父亲节点为中心左旋，父亲节点和兄弟节点的右孩子都变黑



⑥ 以节点为中心左旋——RedBlackTree::leftRotate() 函数:

按照下图改变节点的指针，完成以节点 z 为中心的左旋

/* 左旋

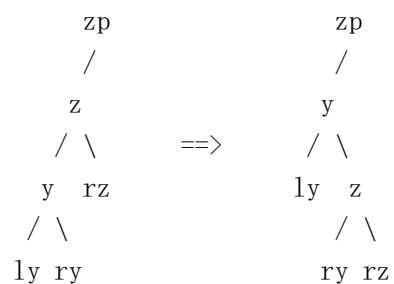


*/

⑦ 以节点为中心右旋——RedBlackTree::rightRotate() 函数:

按照下图改变节点的指针，完成以节点 z 为中心的右旋

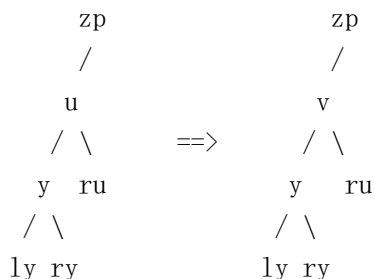
/* 左旋



*/

⑧ 将节点位置替换——RedBlackTree::change() 函数:

改变节点的指针，将 u 的节点替换为 v



⑨ 寻找子树中最小节点——RedBlackTree::minNum() 函数:

寻找子树中最小节点, 根据红黑树的搜索树的性质, 需要一直访问左孩子直到访问到外部节点之前即可

⑩ 销毁整颗红黑树——RedBlackTree::destroy() 函数:

从根开始遍历左孩子、右孩子, 访问整棵树的节点并且释放内存空间

(2) 基本操作演示模块——RedBlackTree::show() 函数:

实现一个函数, 利用 `easyx` 库, 经过合适的几何布局, 将红黑树的结构图形化演示出来, 并且调整节点的颜色和内容, 表示红黑树节点的颜色和键值。

在插入操作、删除操作、合并操作的前后进行红黑树结构的图形化演示出来, 经过前后结构的对比演示红黑树的操作。

(3) 英汉词典实现模块

英汉词典以红黑树的数据结构建立, 红黑树的节点为单词节点, 包含了英文单词、中文用法、翻译, 以及重载的运算符, 以英文单词为主键。

① 词典建立

输入单词、翻译、用法例句, 组合成结构体, 以单词为主键, 插入红黑树中建立一个小型的英汉词典索引

② 词典展示

先序遍历整颗红黑树, 将单词、翻译、用法输出, 因为先序遍历可以保证遍历的顺序是单词的字典序从小到大的顺序

③ 词典基本功能 (查询索引、插入、删除)

查询: 输入单词, 可以查找词典中是否有对应单词, 如果有, 输出查询到的翻译和用法例句

插入: 输入单词、翻译、用法例句, 向词典中插入单词。

删除: 输入单词, 将查找到的对应单词删除。

(4) 红黑树合并模块——RedBlackTree::merge() 函数

将其中一棵红黑树的所有节点全部插入到另外一棵红黑树中, 并且将第一棵红黑树销毁。

2.4 数据及数据类(型) 定义

(1) 创建一个单词结构, 重载其运算符, 实现其构造函数, 其结构如下

```

struct word { //单词
    string key; //英文单词
    string translation; //中文翻译
    vector<string> usage; //用法例句
    //三个构造函数: 有参、无参、复制构造
    word() {} ;
    word(string k, string t, vector<string> u) :key(k), translation(t), usage(u) {} ;
    word(const word& w) :key(w.key), translation(w.translation), usage(w.usage) {} ;
    //重载运算符: <, >, ==, =, <<
    bool operator < (const word& w) {
  
```

```

        return this->key < w.key;
    }

    bool operator > (const word& w) {
        return this->key > w.key;
    }

    bool operator == (const word& w) {
        return this->key == w.key;
    }

    void operator = (const word& w) {
        this->key = w.key;
    }

    friend ostream& operator << (ostream& os, const word& w) {
        os << "单词: " << w.key << "\n";
        os << "翻译: " << w.translation << "\n";
        os << "用法: "; for (auto& i : w.usage) os << i << " "; os << "\n";
        return os;
    }
};

```

(2) 创建一个红黑树节点，包含单词结构，结构如下

```

struct Node { //红黑树节点
    bool color = 0; //节点颜色，0 为红色，1 为黑色，默认为 0
    word value; //关键词
    Node* left = NULL; //左孩子
    Node* right = NULL; //右孩子
    Node* father = NULL; //父亲
    //以下为四个构造函数
    Node() {};
    Node(word v) : value(v) {};
    Node(word v, Node* f) : value(v), father(f) {};
    Node(word v, Node* f, Node* l, Node* r) : value(v), father(f), left(l), right(r) {};
};

```

(3) 创建一个红黑树类，依托红黑树节点，结构如下：

```

class RedBlackTree { //红黑树类
public:
    RedBlackTree(); //构造函数
    ~RedBlackTree(); //析构函数
    Node* find(string value); //查找
    bool insert(word value, int ord); //插入
    void insertBalance(Node* s); //插入调整
    bool erase(string value); //删除
    void eraseBalance(Node* x); //删除调整
    void print(Node* now); //打印
    void show(); //基本操作演示
    void leftRotate(Node* z); //左旋

```

```

void rightRotate(Node* z);//右旋
void change(Node* u, Node* v);//将 v 替换 u
Node* minNum(Node* x); //找到 x 开始最小关键词的节点
void destory(Node* now);//递归销毁整棵树
void merge(RedBlackTree& t);//红黑树合并
Node* getRoot();//获得根
private:
    Node* root;//根
    Node* exNode;//外部节点
};

```

2.5. 算法设计及分析

(1) 红黑树的实现模块:

- ① **节点查找——RedBlackTree::find()函数**。最多访问相当于树高的个数的节点，树高最高为 $2 \times \log n$ ，故时间复杂度为 $O(\log n)$ ，实现代码如下：

```

Node* RedBlackTree::find(string value) { //寻找一个节点
    Node* s = root;
    while (s != exNode) {
        if (s->value.key == value)
            return s;
        if (value < s->value.key) {
            s = s->left;
        }
        else {
            s = s->right;
        }
    }
    return NULL; //没找到
}

```

- ② **节点插入——RedBlackTree::insert()函数**。最多访问相当于树高的个数的节点，树高最高为 $2 \times \log n$ ，故时间复杂度为 $O(\log n)$ ，实现代码如下：

```

bool RedBlackTree::insert(word value, int ord) { //插入节点
    Node* pr = exNode; //s 的父亲
    Node* s = root; //当前节点
    while (s != exNode) {
        if (value == s->value)
            return false; //找到相同元素，返回
        pr = s;
        if (value < s->value) { //在左子树
            s = s->left;
        }
        else { //在右子树
            s = s->right;
        }
    }
}

```

```

    }
    s = new Node(value, pr, exNode, exNode); //申请新节点，两个孩子都是外部节点
    if (pr == exNode) { //空树
        root = s;
    }
    else {
        if (value < pr->value) { //在左子树
            pr->left = s;
        }
        else { //在右子树
            pr->right = s;
        }
        s->father = pr;
    }
    insertBalance(s); //插入平衡
    if (ord == 2)
        show();
    return true; //找到
}

```

③ **节点删除——RedBlackTree::erase()函数**。最多访问相当于树高的个数的节点，树高最高为 $2 \times \log n$ ，故时间复杂度为 $O(\log n)$ ，实现代码如下：

```

bool RedBlackTree::erase(string value) { //删除节点
    Node* z;
    z = find(value);
    if (z == NULL) //没找到
        return false;
    if (z != exNode) { //不是外部节点
        Node* x = exNode;
        Node* y;
        word ycolor;
        if (z->right == exNode) { //右子树为外部节点
            eraseBalance(z);
            change(z, z->left);
        }
        else {
            y = minNum(z->right);
            z->value = y->value;
            eraseBalance(y);
            change(y, y->right);
        }
    }
    else {
        cout << "删除失败，不存在该节点" << endl;
    }
}

```

```
return true;
```

```
}
```

- ④ 插入调整——RedBlackTree::insertBalance()函数。最多三次旋转，每次旋转时间复杂度为 $O(1)$ ，故时间复杂度为 $O(1)$ ，实现代码如下：

```
void RedBlackTree::insertBalance(Node* s) {
    Node* uncle;//叔叔节点，当前节点的父亲的另外一个孩子
    while (s->father->color == 0) {
        if (s->father == s->father->father->left) {//当前为父亲的左孩子
            uncle = s->father->father->right;
            if (uncle->color == 0) {//情况三
                s->father->color = 0;
                uncle->color = 0;
                s->father->father->color = 1;
                s = s->father->father;
            }
            else {
                if (s == s->father->right) {//情况四
                    s = s->father;
                    leftRotate(s);
                }
                s->father->color = 1;//情况五
                s->father->father->color = 0;
                rightRotate(s->father->father);
            }
        }
        else {//镜像另一半
            uncle = s->father->father->left;
            if (uncle->color == 0) {
                s->father->color = 1;
                uncle->color = 1;
                s->father->father->color = 0;
                s = s->father->father;
            }
            else {
                if (s == s->father->left) {
                    s = s->father;
                    rightRotate(s);
                }
                s->father->color = 1;
                s->father->father->color = 0;
                leftRotate(s->father->father);
            }
        }
    }
}
```

```
root->color = 1;
```

```
}
```

- ⑤ **删除调整——RedBlackTree::eraseBalance()函数**。最多三次旋转，每次旋转时间复杂度为 $O(1)$ ，故时间复杂度为 $O(1)$ ，实现代码如下：

```
void RedBlackTree::eraseBalance(Node* x) {
    while (x != root && x->color != 0) {
        if (x == x->father->left) { //当前节点在父亲节点左侧
            Node* brother = x->father->right;
            if (brother->color == 0) { //情况三
                x->father->color = 0;
                brother->color = 1;
                leftRotate(x->father);
                brother = x->father->right;
            }
            if (brother->left->color == 1 && brother->right->color == 1) { //情况四
                brother->color = 0;
                if (x->father->color == 0)
                    x->father->color = 1, x = root;
                else
                    x = x->father;
            }
            else if (brother->right->color == 1) { //情况五
                brother->color = 0;
                brother->left->color = 1;
                rightRotate(brother);
                brother = x->father->right;
            }
            else { //情况六
                brother->color = x->father->color;
                x->father->color = 1;
                brother->right->color = 1;
                leftRotate(x->father);
                x = root;
            }
        }
        else { //镜像另一半
            Node* brother = x->father->left;
            if (brother->color == 0) {
                x->father->color = 0;
                brother->color = 1;
                rightRotate(x->father);
                brother = x->father->left;
            }
            if (brother->left->color == 1 && brother->right->color == 1) {
```

```

        brother->color = 0;
        if (x->father->color == 0)
            x->father->color = 1, x = root;
        else
            x = x->father;
    }
    else if (brother->left->color == 1) {
        brother->color = 0;
        brother->right->color = 1;
        leftRotate(brother);
        brother = x->father->left;
    }
    else {
        brother->color = x->father->color;
        x->father->color = 1;
        brother->left->color = 1;
        rightRotate(x->father);
        x = root;
    }
}
}
}

```

⑥ 以节点为中心左旋——RedBlackTree:: leftRotate() 函数。进行一次 $O(1)$ 的旋转，时间复杂度为 $O(1)$ ，实现代码如下：

/* 左旋

```

      zp          zp
      /           /
     z           y
    /\          /\
   lz y        z ry
    /\          /\
   ly ry      lz ly

```

*/

void RedBlackTree::leftRotate(Node* z) { //以 z 节点为中心左旋，其原理图如上

Node* y = z->right;

z->right = y->left;

if (y->left != exNode) { //y 左孩子不空

y->left->father = z;

}

y->father = z->father;

if (root == z) { //z 为根节点

root = y;

}

else if (z == z->father->left) { //z 为父亲的左孩子

```

        z->father->left = y;
    }
    else { //z 为父亲的右孩子
        z->father->right = y;
    }
    y->left = z;
    z->father = y;
}

```

- ⑦ 以节点为中心右旋——RedBlackTree:: rightRotate() 函数。进行一次 $O(1)$ 的旋转，时间复杂度为 $O(1)$ ，实现代码如下：

/* 左旋

```

        zp          zp
        /           /
       z           y
      /\          /\
     y  rz       ly z
    /\          /\
   ly ry       ry rz
*/

```

void RedBlackTree::rightRotate(Node* z) { //以 z 为中心右旋，原理图如上

```

    Node* y = z->left;
    z->left = y->right;
    if (y->right != exNode) { //y 的右孩子不为空
        y->right->father = z;
    }
    y->father = z->father;
    if (z == root) { //z 为根节点
        root = y;
    }
    else if (z == z->father->left) { //z 为父亲的左孩子
        z->father->left = y;
    }
    else { //z 为父亲的右孩子
        z->father->right = y;
    }
    y->right = z;
    z->father = y;
}

```

- ⑧ 将节点位置替换——RedBlackTree:: change() 函数。时间复杂度为 $O(1)$ ，实现代码如下：

```

void RedBlackTree::change(Node* u, Node* v) { //将 u 的位置替换为 v
    if (u->father == exNode) {
        root = v;
    }
    else if (u == u->father->left) {

```



```

        u->father->left = v;
    }
    else {
        u->father->right = v;
    }
    v->father = u->father;
}

```

- ⑨ **寻找子树中最小节点——RedBlackTree::minNum()函数。**最多访问相当于树高的个数的节点，树高最高为 $2 \times \log n$ ，故时间复杂度为 $O(\log n)$ ，实现代码如下：

```

Node* RedBlackTree::minNum(Node* x) { //找到当前节点子树中值最小的节点
    if (x->left == exNode) //找到了
        return x;
    return minNum(x->left);
}

```

- ⑩ **销毁整颗红黑树——RedBlackTree::destroy()函数。**对于每个节点进行一次访问，故时间复杂度为 $O(n)$ ，实现代码如下：

```

void RedBlackTree::destroy(Node* now) { //销毁整颗红黑树，释放内存空间
    if (now == exNode)
        return;
    if (now->left != exNode) //有左孩子
        destroy(now->left);
    else
        destroy(now->right); //有右孩子
    delete now;
    now = NULL;
}

```

- (2) **基本操作演示模块——RedBlackTree::show()函数。**对于每一个节点访问一遍，展示在图形化界面上，故时间复杂度为 $O(n)$ 。实现代码如下：

```

void RedBlackTree::show() { //画出整个红黑树，进行操作演示
    initgraph(1400, 800); //初始化画布
    setbkcolor(WHITE); //画布颜色为白色
    setlinecolor(BLACK); //线条颜色为黑色
    setorigin(700, 30); //初始化原点位置
    settextrcolor(BLACK); //字体颜色为黑色
    cleardevice(); //清空画布
    struct cir { //节点结构体
        Node* node; //红黑树节点
        int x, y; //坐标
        string s; //单词
        bool color; //1 为黑色
        int cnt = 1; //第几行
    };
    queue<cir> q;
    cir newNode;
}

```

```

newNode.x = 0, newNode.y = 0, newNode.color = 1, newNode.s = root->value.key, newNode.node = root; //根节点
q.push(newNode);
while (q.size()) { //层次遍历画出整颗红黑树
    cir now = q.front();
    q.pop();
    if (now.color == 1) //红色节点
        setlinecolor(BLACK), settextcolor(BLACK);
    else //黑色节点
        setlinecolor(RED), settextcolor(RED);
    circle(now.x, now.y, 25); //画出一个○作为节点
    wchar_t s[100];
    for (int i = 0; i < now.s.length(); i++)
        s[i] = now.s[i];
    s[now.s.length()] = '\0';
    outtextxy(now.x - 23, now.y - 10, s); //在节点内填入单词
    if (now.node->left != exNode) { //左孩子
        //给新节点属性赋值
        newNode.x = now.x - 1400 / pow(2, now.cnt + 1), newNode.y = now.y + 110, newNode.s =
now.node->left->value.key, newNode.node = now.node->left, newNode.color = now.node->left->color, newNode.cnt =
now.cnt + 1;

        if (newNode.color == 1)
            setlinecolor(BLACK);
        else
            setlinecolor(RED);
        line(now.x, now.y + 25, newNode.x, newNode.y - 25);
        q.push(newNode);
    }
    if (now.node->right != exNode) { //右孩子
        //给新节点属性赋值
        newNode.x = now.x + 1400 / pow(2, now.cnt + 1), newNode.y = now.y + 110, newNode.s =
now.node->right->value.key, newNode.node = now.node->right, newNode.color = now.node->right->color, newNode.cnt =
now.cnt + 1;

        if (newNode.color == 1)
            setlinecolor(BLACK);
        else
            setlinecolor(RED);
        line(now.x, now.y + 25, newNode.x, newNode.y - 25);
        q.push(newNode);
    }
}
}
/*    EndBatchDraw();*/
_getwch(); //暂停，按任意键继续
closegraph(); //关闭画布
}

```

(3) 英汉词典实现模块

① 词典建立, 输入单词、翻译、用法, 包装成单词结构插入到红黑树中。时间复杂度为 $O(n \log n)$, 实现代码如下:

```
int n;//单词个数
cout << "请输入单词个数:\n";
cin >> n;
for (int i = 1; i <= n; i++) { //输入单词、翻译、用法, 并且插入红黑树
    cout << "请输入第" << i << "个单词及其翻译和用法:\n";
    string key, translation;
    cin >> key >> translation;
    string read;
    getline(cin, read, '\n');
    stringstream ss(read);
    vector<string> usage;
    while (ss >> read) {
        usage.push_back(read);
    }
    bool f = t.insert(word{ key, translation, usage }, 0);
    if (f == false)
        cout << "插入失败, 存在重复值! ";
}
```

② 词典展示, 先序遍历打印出整个红黑树的节点, 将单词、翻译、用法输出。时间复杂度为 $O(n)$, 实现代码如下:

```
void RedBlackTree::print(Node* now) { //先序遍历打印出整个红黑树的节点单词
    if (now->left != exNode)
        print(now->left);
    cout << " \n-----\n" <<
now->value << "-----\n";
    if (now->right != exNode)
        print(now->right);
    //cout << endl;
}
```

③ 词典基本功能(查询索引、插入、删除), 依托红黑树查找、插入、删除功能进行词典的查询索引、插入单词和删除单词, 对于每一种操作的时间复杂度都为 $O(\log n)$, 实现代码如下:

```
if (ord == "1") { //查找单词
    string word1;
    cout << "请输入一个单词: \n";
    cin >> word1;
    Node* now = t.find(word1);
    if (now != NULL) { //查找成功, 输出
        cout <<
"\n-----\n" << t.find(word1)->value <<
"-----\n";
    }
}
```

```

        else//查找失败
            cout << "抱歉，未查询到该单词！\n";
    }
    else if (ord == "2") { //插入单词
        string key, translation;
        cout << "请输入一个单词、翻译、用法：\n";
        cin >> key >> translation;
        string read;
        getline(cin, read, '\n');
        stringstream ss(read);
        vector<string> usage;
        while (ss >> read) {
            usage.push_back(read);
        }
        bool flag = t.insert(word{ key, translation, usage }, 0);
        if (flag) //插入成功
            cout << "插入成功！\n";
        else //插入失败
            cout << "插入失败，存在重复单词！\n";
    }
    else if (ord == "3") { //删除单词
        string word1;
        cout << "请输入一个单词：\n";
        cin >> word1;
        bool flag = t.erase(word1);
        if (flag) //删除成功
            cout << "删除成功！\n";
        else //删除失败
            cout << "抱歉，删除失败！\n";
    }
}

```

(4) 红黑树合并模块——RedBlackTree::merge() 函数，假设两棵红黑树节点个数分别为 n 和 m ，对于第二棵红黑树每次删除根节点，进行一次调整，每次时间复杂度为 $O(\log m)$ ，对于第一棵红黑树每次进行一次插入，每次时间复杂度为 $O(\log n)$ ，故总时间复杂度为 $O(m(\log n + \log m))$ 。实现代码如下：

```

void RedBlackTree::merge(RedBlackTree& t) //红黑树合并
{
    while (t.getRoot()->left->value.key != "" && t.getRoot()->right->value.key != "") { //将 t 的节点从根节点开始取出并且插入到当前的红黑树
        //cout << "!" << t.getRoot()->left->value.key << " " << t.getRoot()->right->value.key << endl;
        insert(t.getRoot()->value, 1);
        t.erase(t.getRoot()->value.key);
        /*      t.show();*/
    }
    if (t.getRoot()->left->value.key != "") { //把最后剩余的节点取出插入

```

```

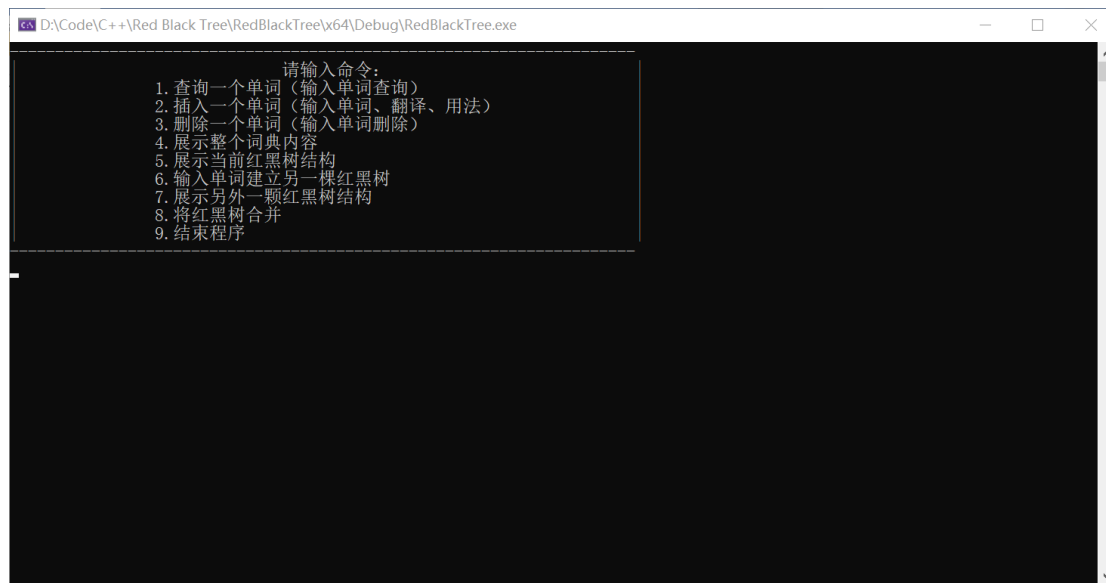
        insert(t.getRoot()->left->value, 1);
        insert(t.getRoot()->value, 1);
    }
    else if (t.getRoot()->right->value.key != "") {
        insert(t.getRoot()->right->value, 1);
        insert(t.getRoot()->value, 1);
    }
    else {
        insert(t.getRoot()->value, 1);
    }
}

```

3. 测试

基于红黑树建立一个小型的英汉词典索引（任务 3），以英文单词为主键，附属其翻译和用法。然后利用这个词典结构的插入、删除、查找（任务 1）、合并对红黑树（任务 4）进行测试，并且进行插入、删除、查找、合并操作的图形化演示（任务 2）

其中命令菜单如下



首先输入一组单词及其翻译和用法，建立初步的小型词典结构

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

    请输入数据建立词典：
    格式为:单词、翻译、用法
-----
请输入单词个数：
5
请输入第1个单词及其翻译和用法：
apple 苹果 An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.
请输入第2个单词及其翻译和用法：
bird 小鸟 A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.
请输入第3个单词及其翻译和用法：
cat 猫 Cats are lions, tigers, and other wild animals in the same family.
请输入第4个单词及其翻译和用法：
dog 狗 The British are renowned as a nation of dog lovers.
请输入第5个单词及其翻译和用法：
egg 鸡蛋 ...a baby bird hatching from its egg.
```

按照字典索引打印出整个词典

```
    请输入命令：
    1. 查询一个单词 （输入单词查询）
    2. 插入一个单词 （输入单词、翻译、用法）
    3. 删除一个单词 （输入单词删除）
    4. 展示整个词典内容
    5. 展示当前红黑树结构
    6. 输入单词建立另一棵红黑树
    7. 展示另外一颗红黑树结构
    8. 将红黑树合并
    9. 结束程序
-----
4

单词: apple
翻译: 苹果
用法: An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.
-----

单词: bird
翻译: 小鸟
用法: A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.
-----

单词: cat
翻译: 猫
用法: Cats are lions, tigers, and other wild animals in the same family.
-----

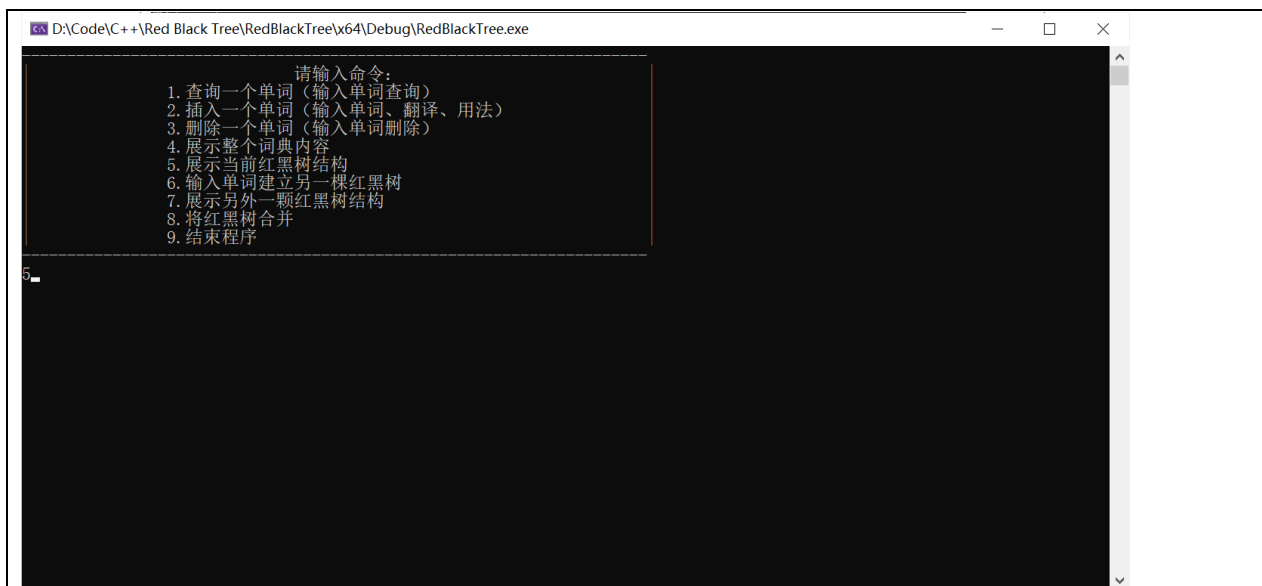
单词: dog
翻译: 狗
用法: The British are renowned as a nation of dog lovers.
-----

单词: egg
翻译: 鸡蛋
用法: ...a baby bird hatching from its egg.
-----
打印成功!

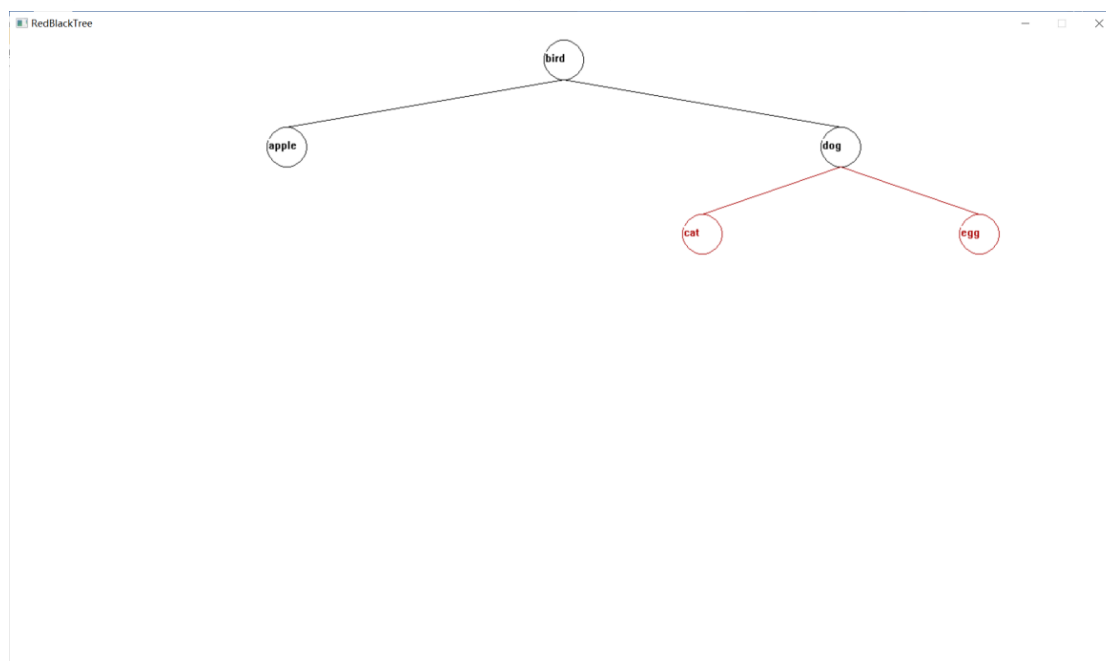
请按任意键继续. . .
```

可以看到词典已经成功地建立了，词典中包含了单词、翻译和用法。

然后展示初始化建立的红黑树结构



可以看到此时红黑树结构如下，满足红黑树的五条性质和二叉搜索树的性质，单词为主键



然后对红黑树的基本操作、合并，以及基于红黑树建立的英汉词典索引进行测试，并且用图形化界面对红黑树基本操作和合并进行演示

(1) 首先测试插入操作

插入一个单词，其英文单词、中文翻译和用法如下：

general 将军 The General's visit to Sarajevo is part of preparations for the deployment of extra troops.

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

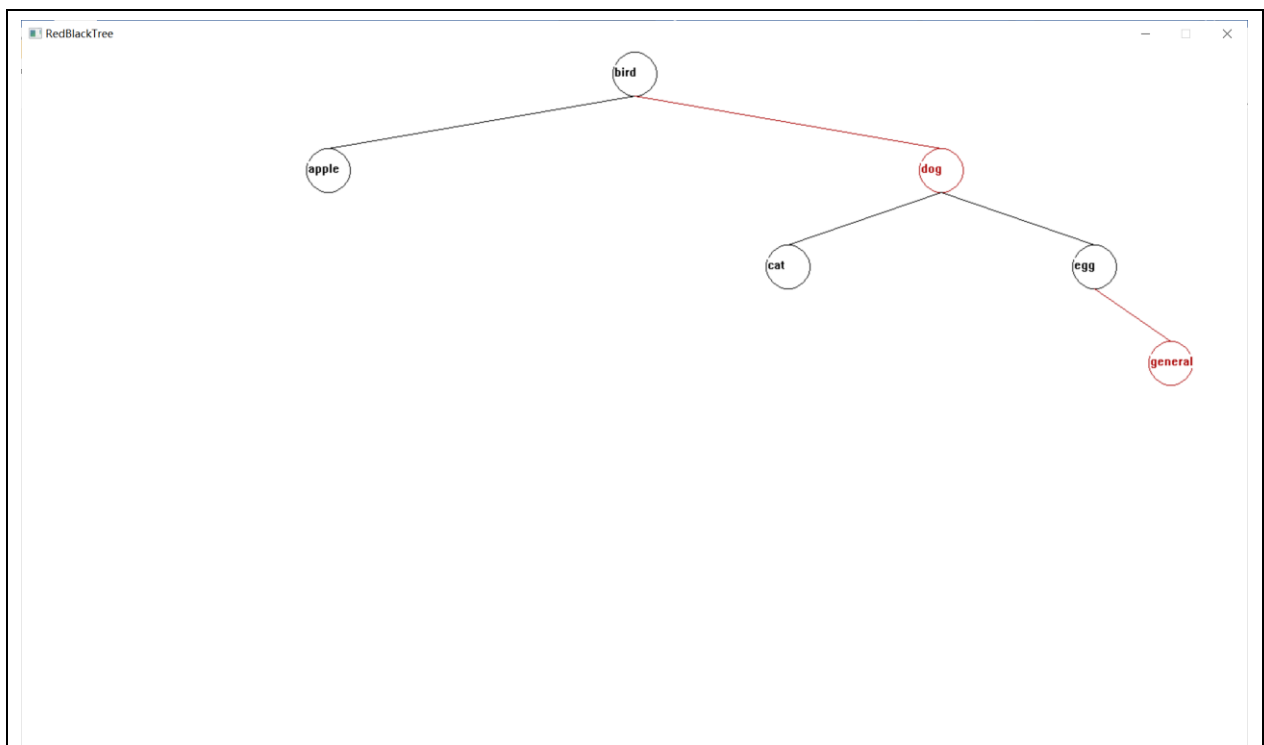
请输入命令：
1. 查询一个单词（输入单词查询）
2. 插入一个单词（输入单词、翻译、用法）
3. 删除一个单词（输入单词删除）
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

2
请输入一个单词、翻译、用法：
general 将军 The General's visit to Sarajevo is part of preparations for the deployment of extra troops.

红黑树结构变化如下：（可以看到，新插入了一个主键为 general 的单词，并且 cat、dog、egg 单词所在节点的颜色发生了变化，以维持红黑树的五条性质，保证红黑树的自平衡）

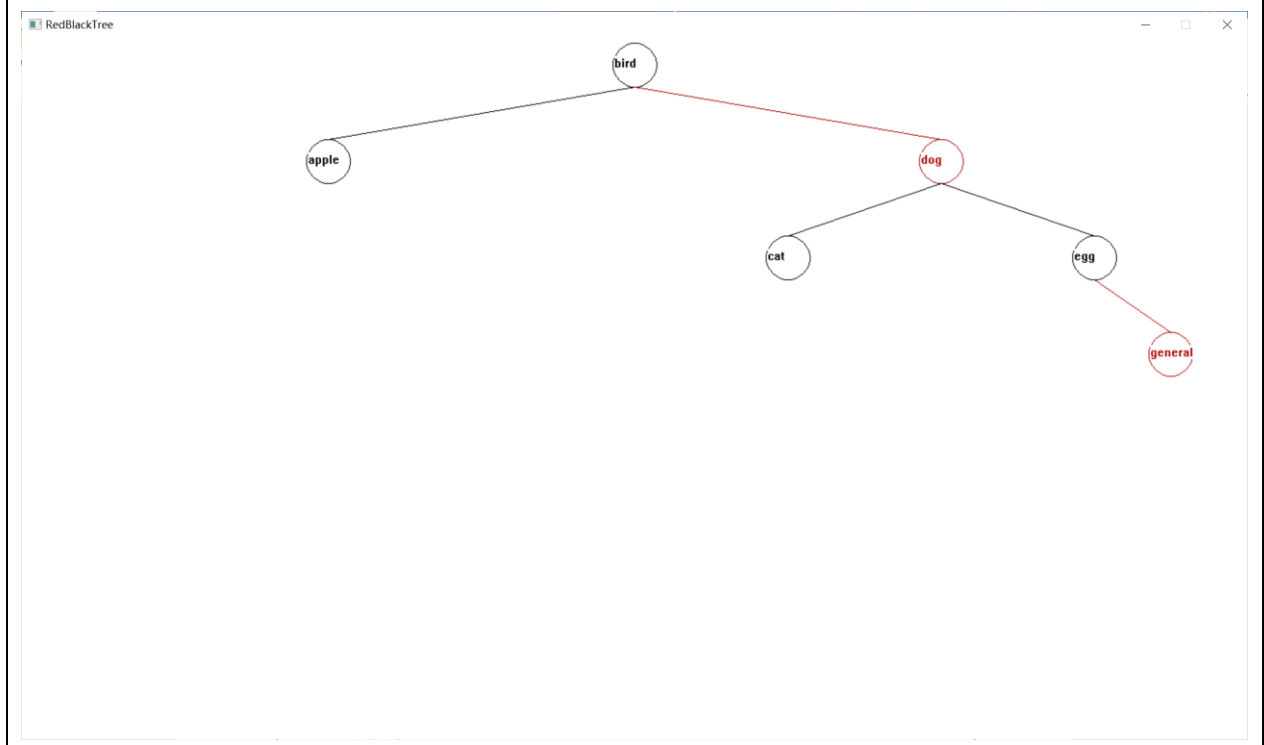
RedBlackTree

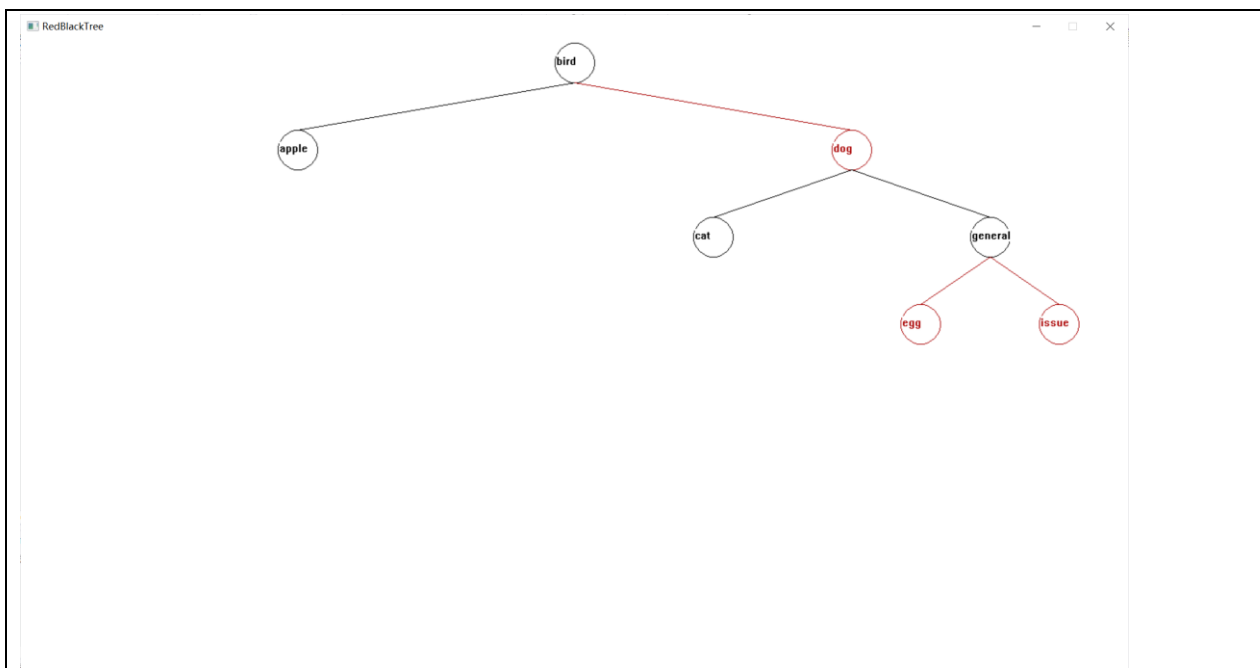
```
graph TD; bird((bird)) --- apple((apple)); bird --- dog((dog)); dog --- cat((cat)); dog --- egg((egg));
```

再插入一个单词

issue 问题 Agents will raise the issue of prize-money for next year's world championships.
红黑树结构变化如下:





可以发现，新插入了一个节点 issue 为红色，为了维持红黑树的两条性质，右下角的三个节点发生了以 egg 为中心的左旋，并且 egg 和 general 的节点颜色也发生了变化

再插入一个单词

hit 击打 Find the exact grip that allows you to hit the ball hard.

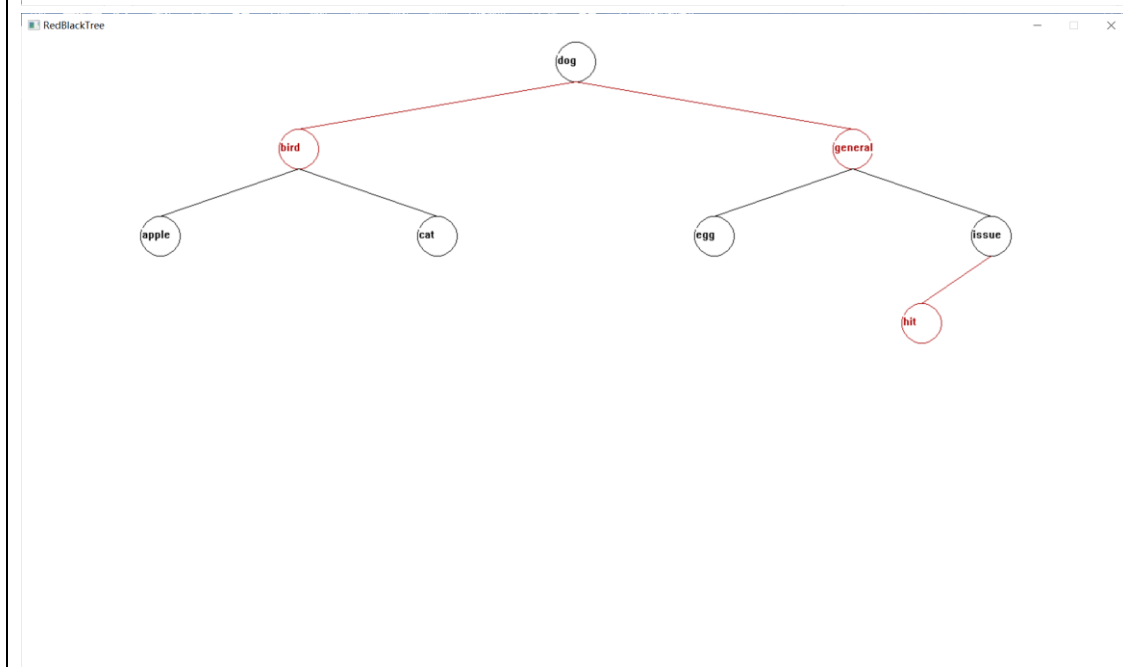
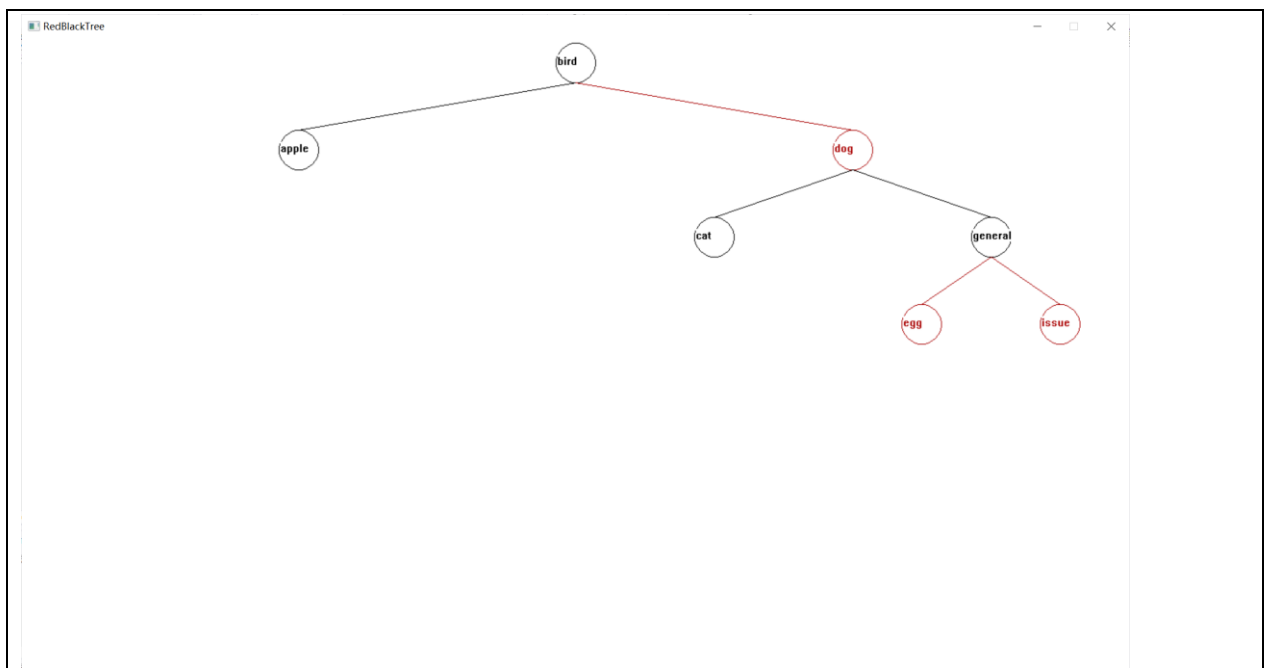
```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令:
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

2
请输入一个单词、翻译、用法:
hit 击打 Find the exact grip that allows you to hit the ball hard.
插入成功!

请按任意键继续. . .
```

红黑树结构变化如下：



可以发现从插入节点 hit 向上开始，有多处发生了红黑树的插入平衡（如变色，左旋等），维持了红黑树的两条性质。

再插入一个单词

boy 男孩 He was still just a boy.

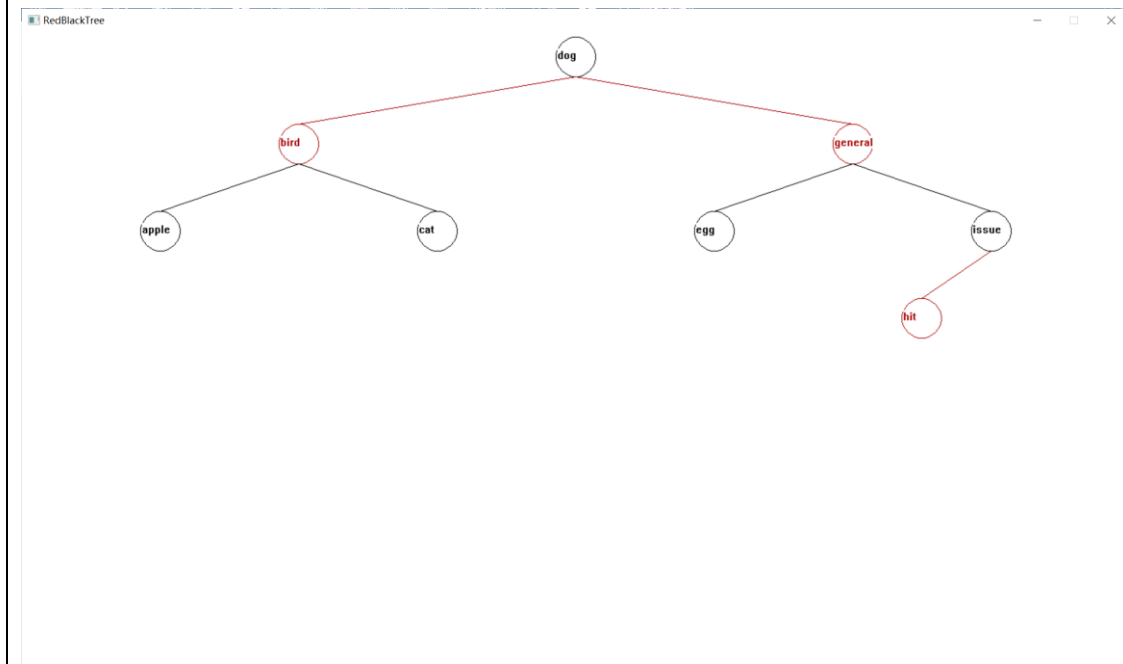
```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

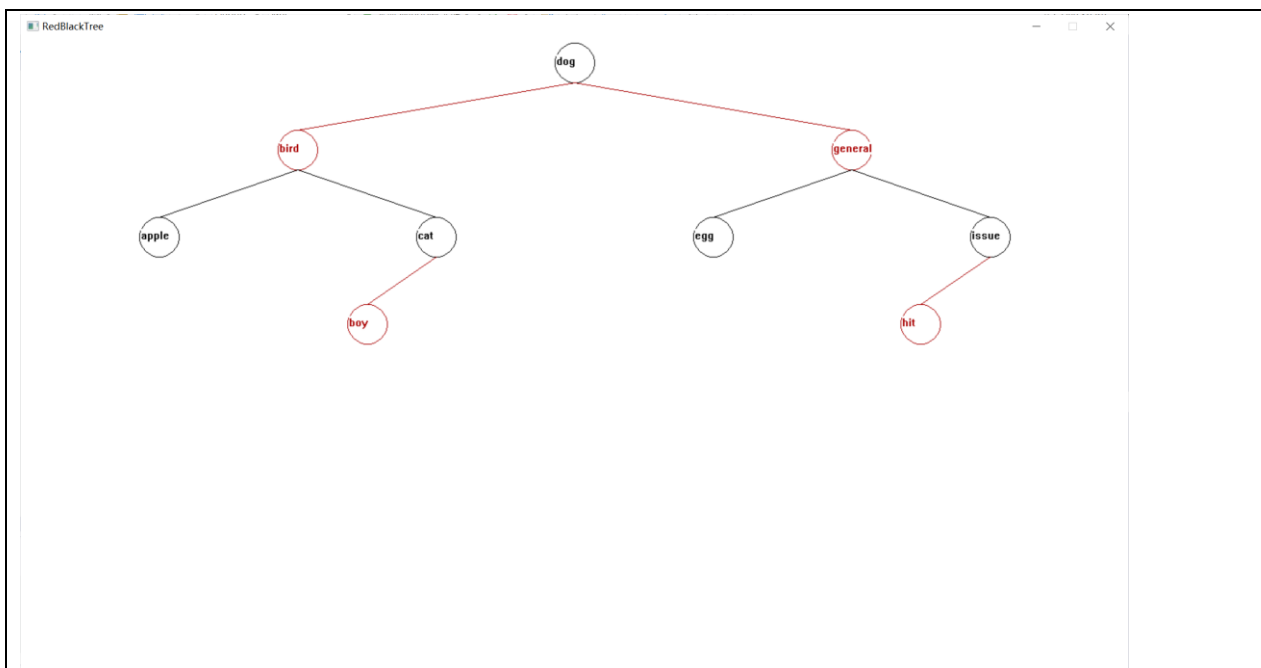
请输入命令：
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

2
请输入一个单词、翻译、用法：
boy 男孩 He was still just a boy.
插入成功！

请按任意键继续. . .
```

红黑树结构变化如下：（可以看到 boy 插到了正确的位置，因为按照字典序 $bird < boy < cat$ ，并且红黑树五条性质也是满足的）





再插入一个单词

journey 旅行 As per our rule, our bus can not stop at her jouney, if you have anything important, please get off the bus at stop.

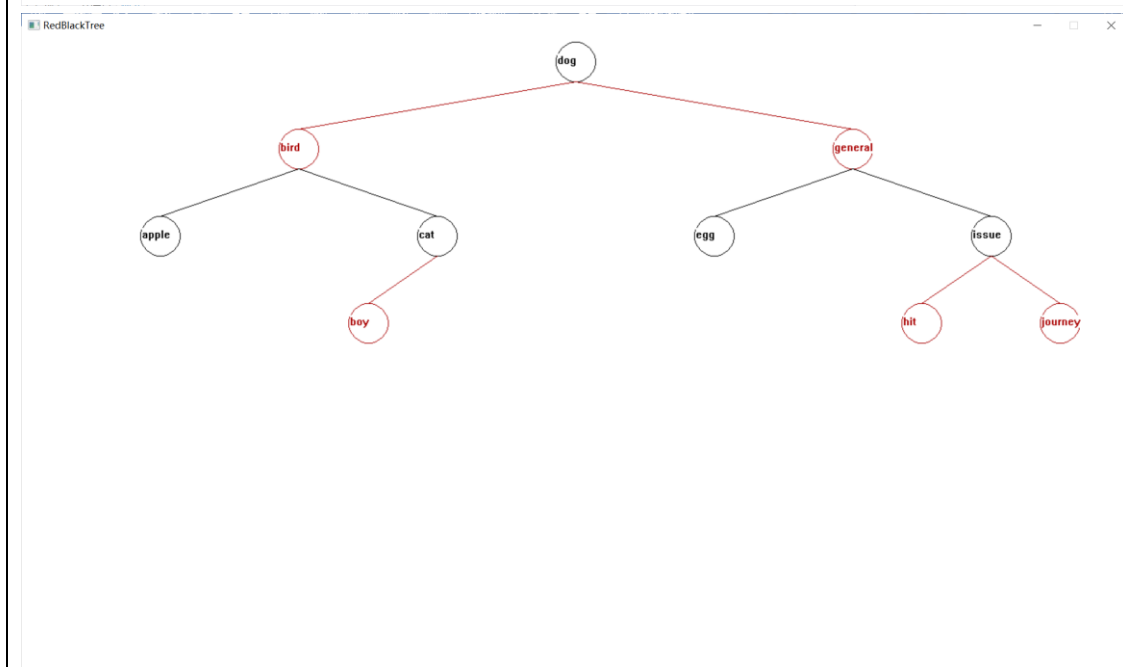
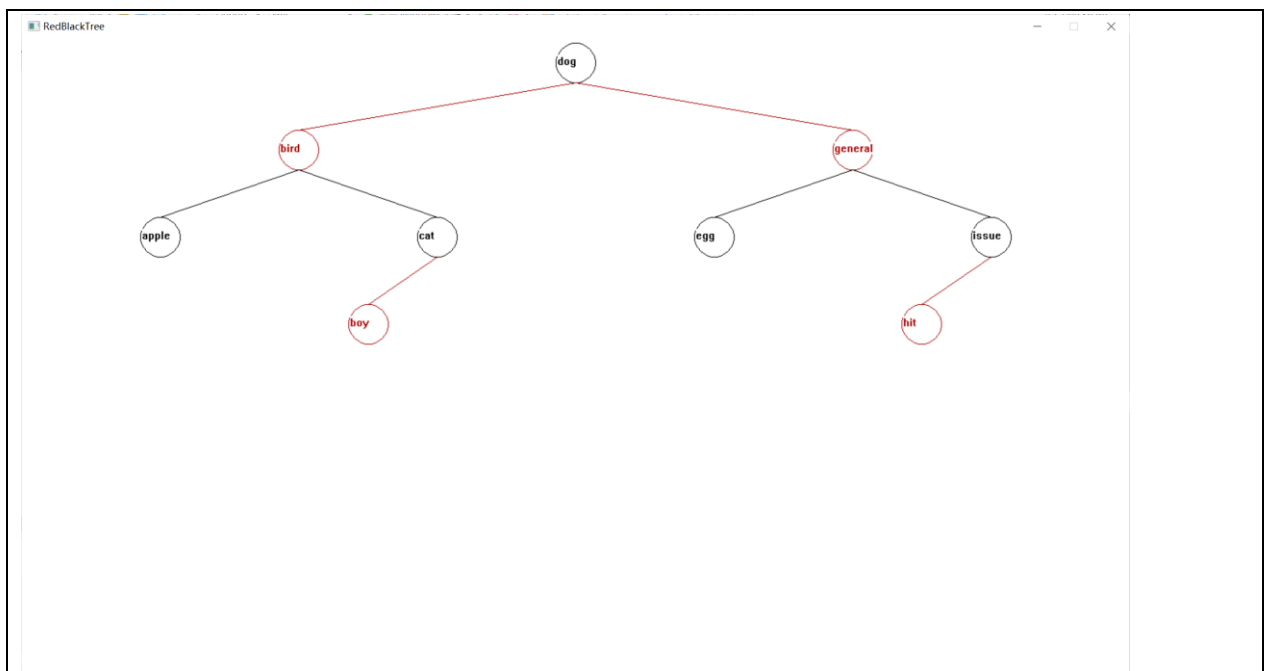
```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令:
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一颗红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

2
请输入一个单词、翻译、用法:
journey 旅行 As per our rule, our bus can not stop at her jouney, if you have anything important, please get off the bus
at stop.
插入成功!

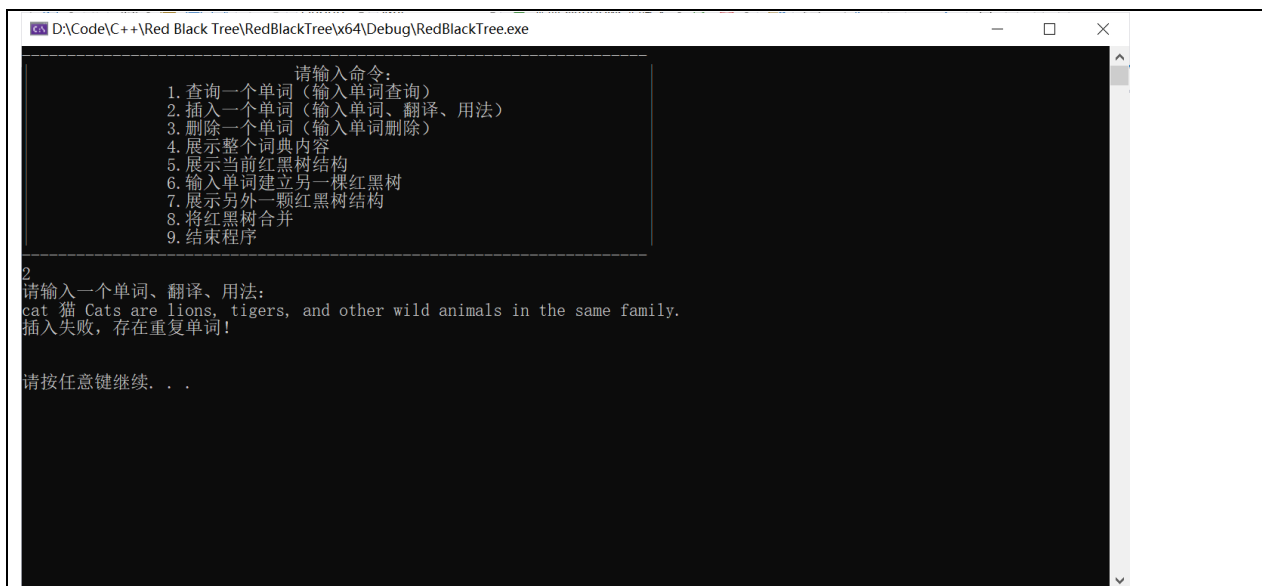
请按任意键继续. . .
```

红黑树结构变化如下:



可以看到红黑树的性质仍是被维持的。

然后我们再次插入已经插入的单词 cat



```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令：
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

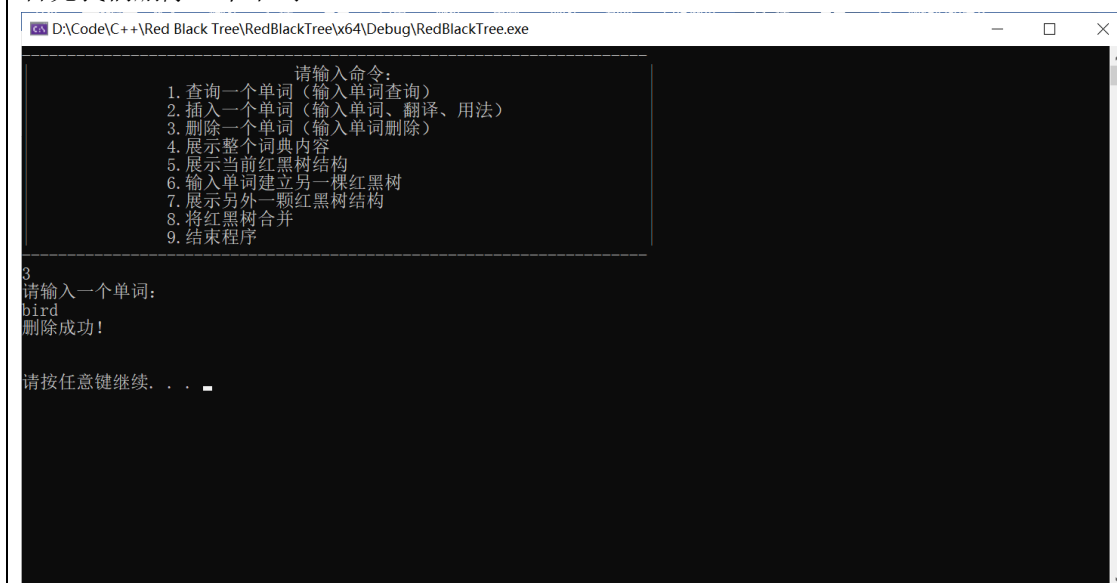
2
请输入一个单词、翻译、用法：
cat 猫 Cats are lions, tigers, and other wild animals in the same family.
插入失败，存在重复单词！

请按任意键继续. . .
```

提示插入失败，因为词典中已经存在了该单词。

(2) 然后我们测试基本操作中的删除操作

首先我们删除一个单词 bird



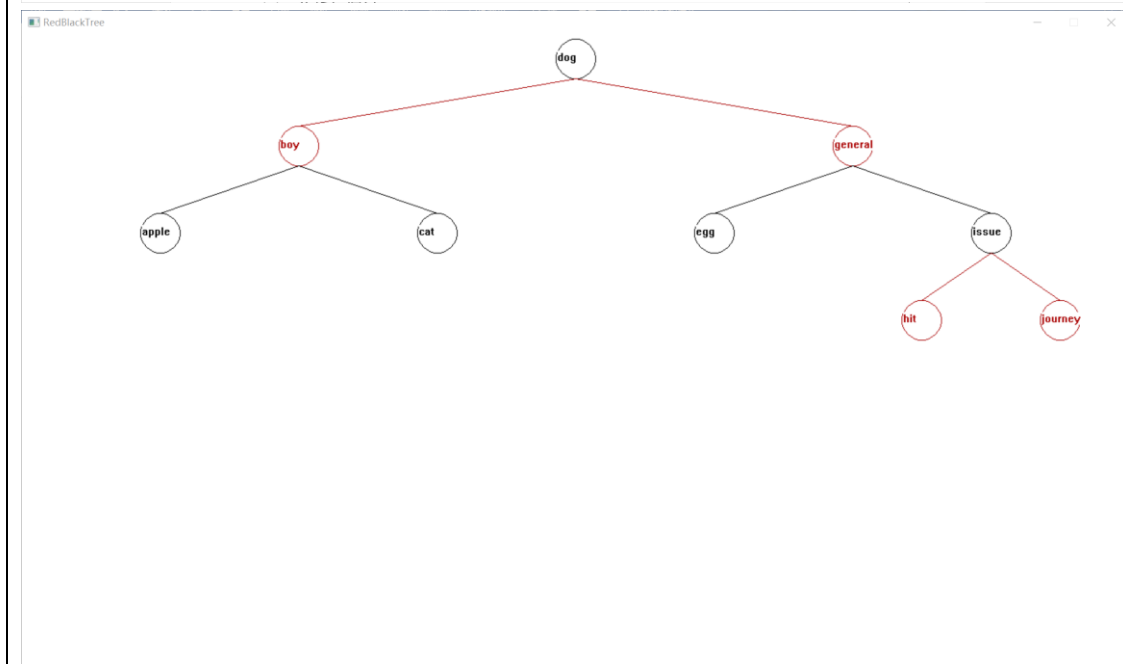
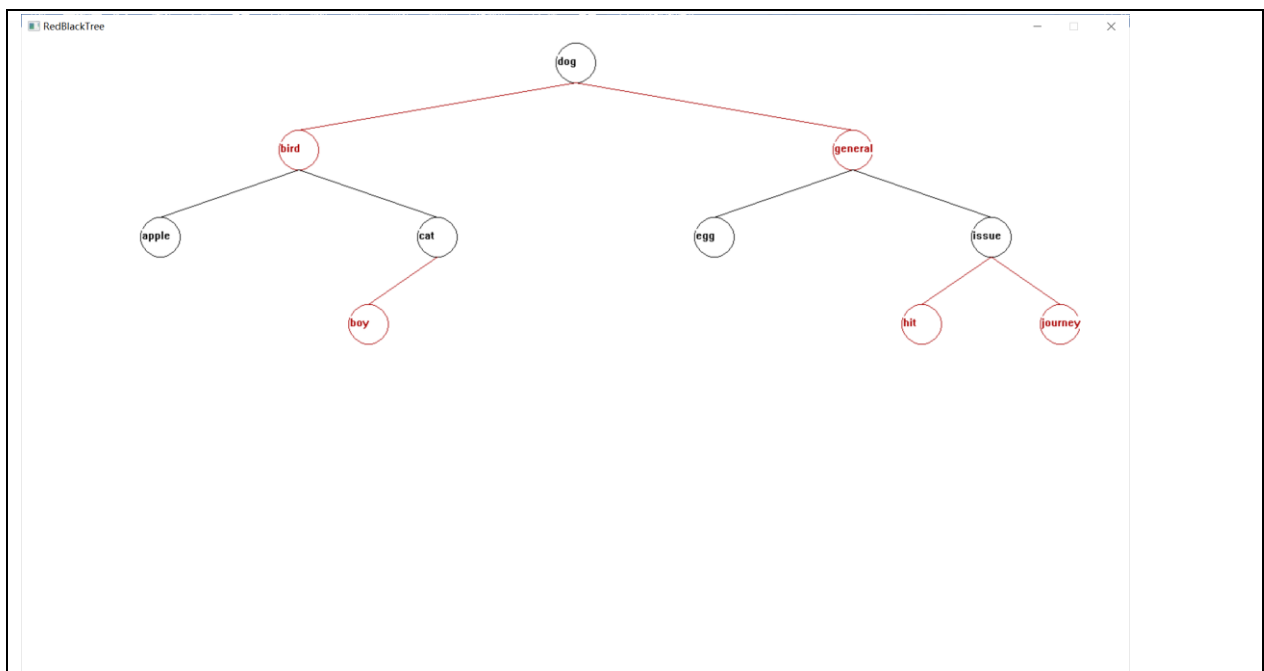
```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令：
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

3
请输入一个单词：
bird
删除成功！

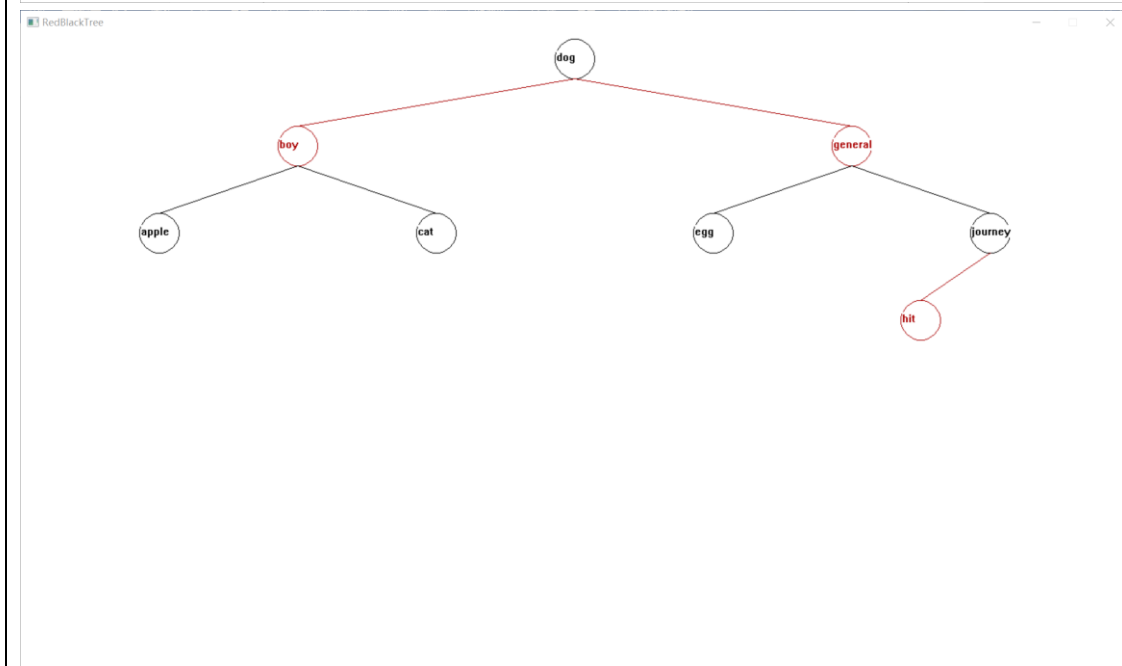
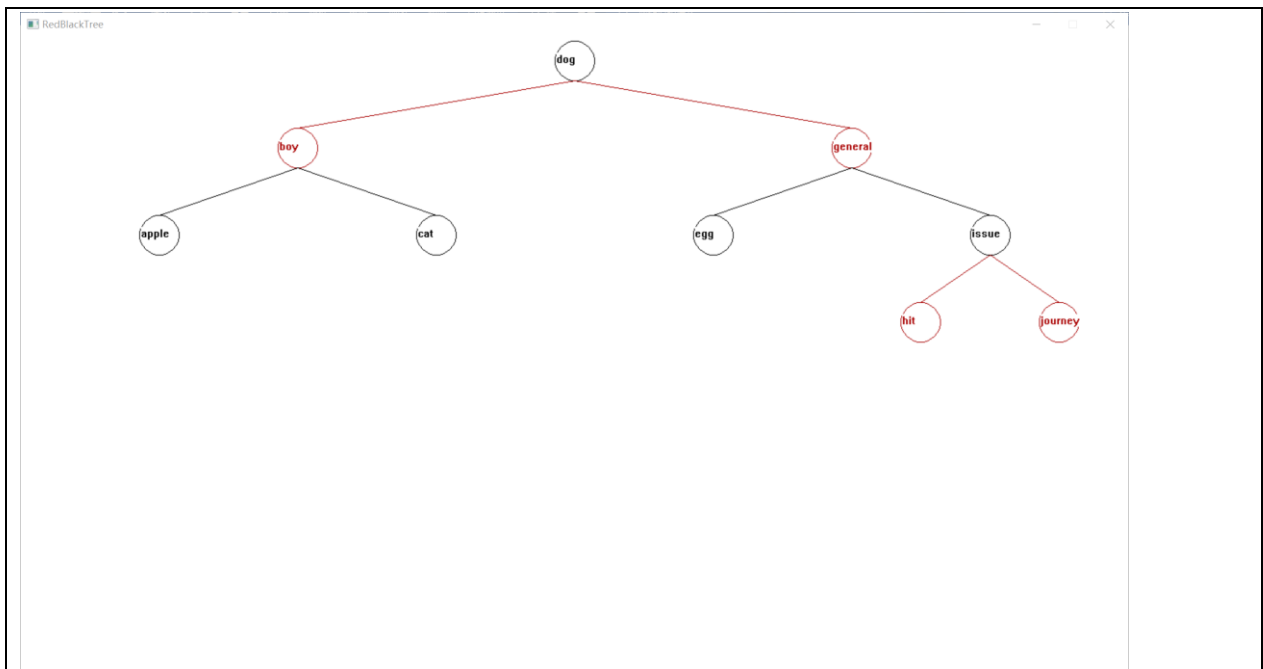
请按任意键继续. . .
```

红黑树结构变化如下：



可以看到红黑树仍满足其五条性质

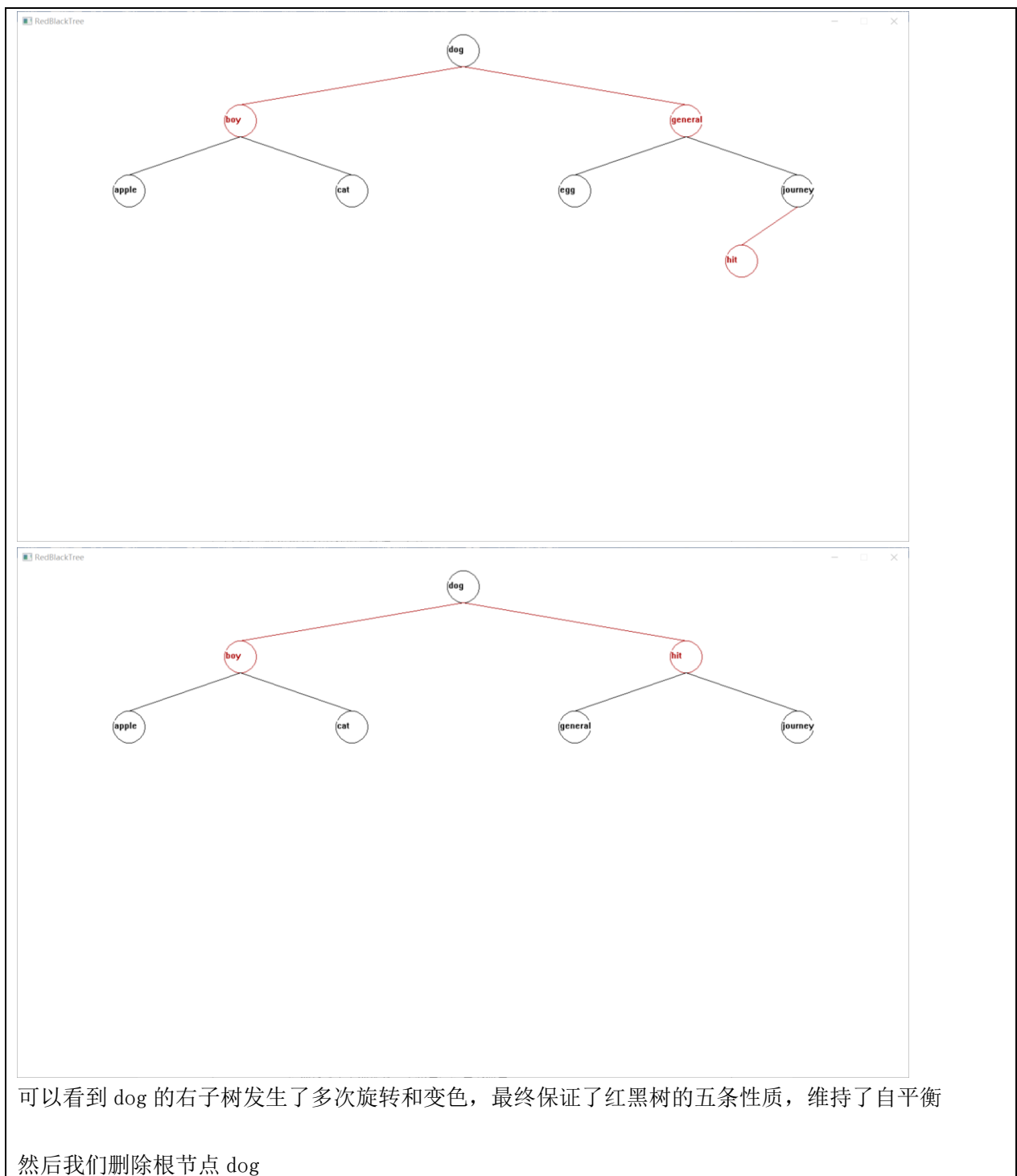
然后我们删除一个单词 issue，观察其如何保证五条性质
红黑树结构变化如下：

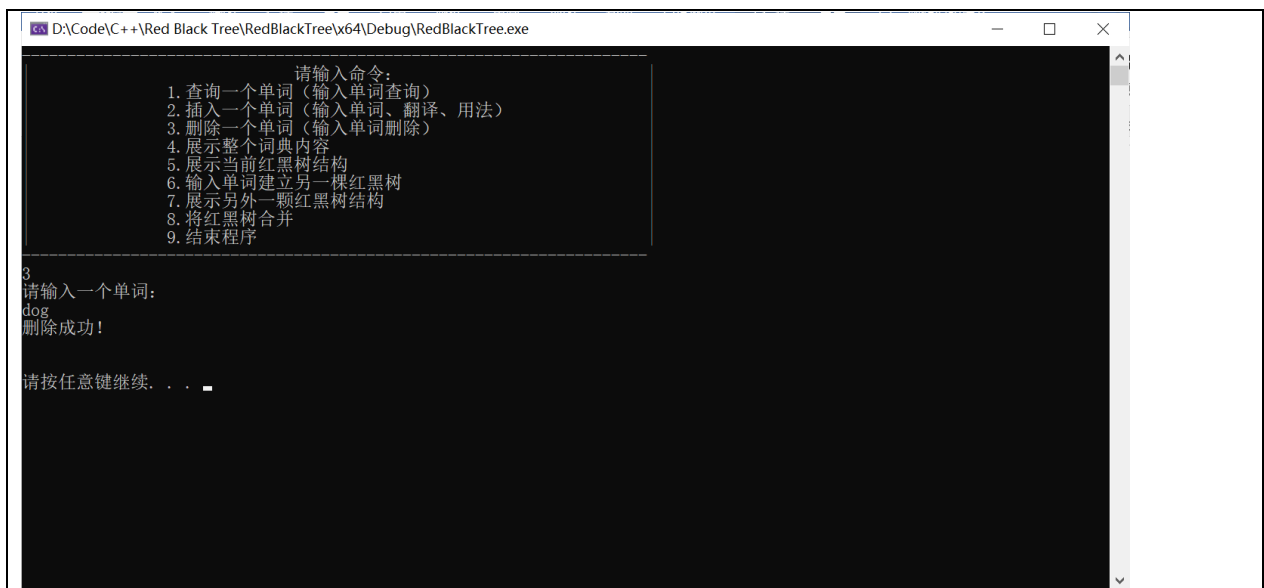


可以看到右下角的节点发生了变色和替换，保证了红黑树的两条性质

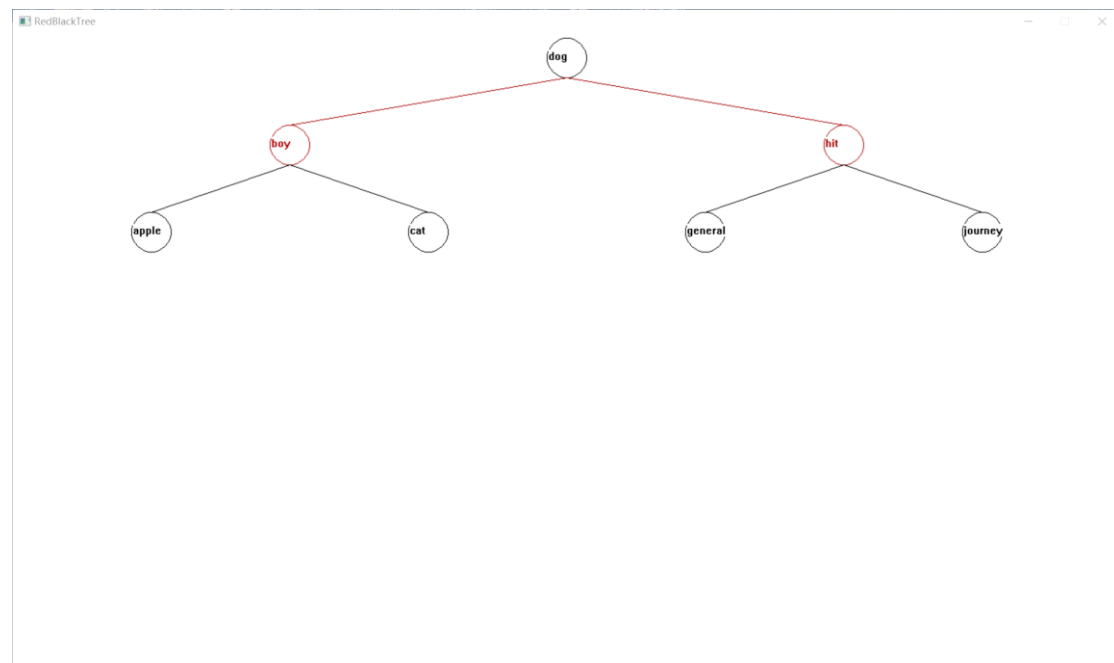
然后删除一个节点 egg

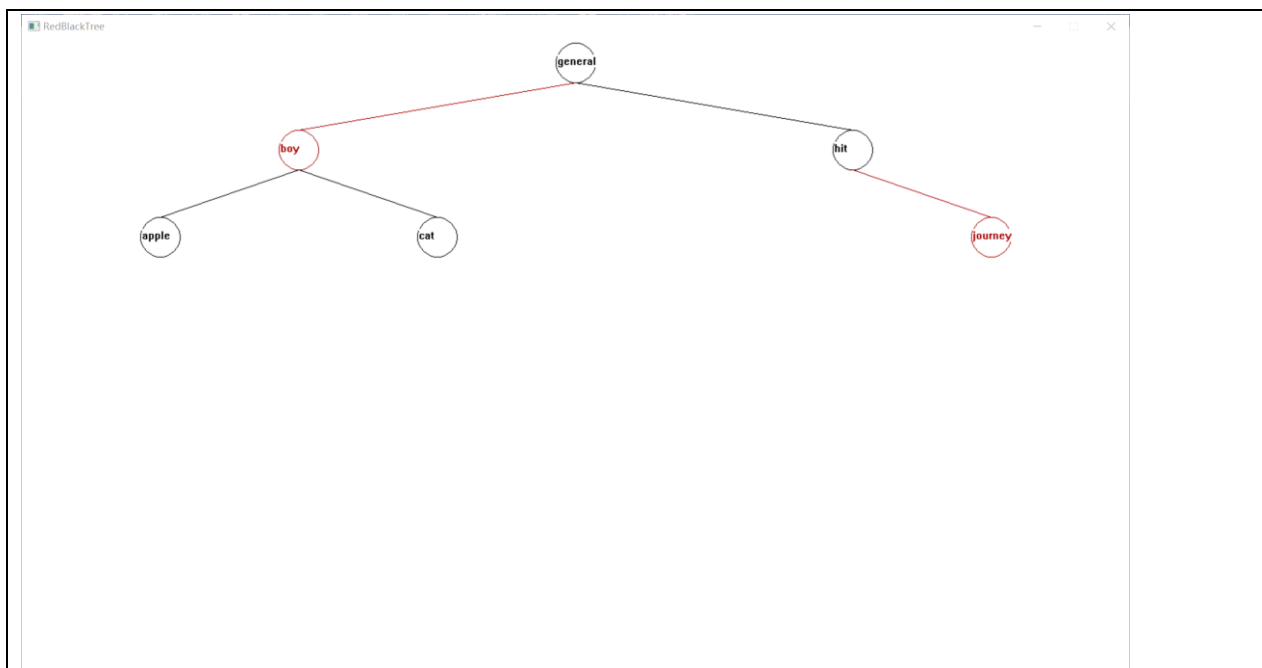
红黑树结构变化如下：





红黑树结构变化如下：





依旧维持了自平衡

(3) 然后测试基本操作中的查找操作

查找一个词典中有的单词 cat

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入命令:
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

1
请输入一个单词:
cat

-----
单词: cat
翻译: 猫
用法: Cats are lions, tigers, and other wild animals in the same family.
-----

请按任意键继续. . .
```

发现成功的寻找到了单词 cat，并且输出了其翻译和用法

在查找一个单词中有的单词 apple

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

1
请输入一个单词：
apple

-----
单词： apple
翻译： 苹果
用法： An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.
-----

请按任意键继续. . .
```

发现成功的寻找到了单词 cat，并且输出了其翻译和用法

再查找一个词典中没有的单词 bird

```
D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

      请输入命令：
      1. 查询一个单词（输入单词查询）
      2. 插入一个单词（输入单词、翻译、用法）
      3. 删除一个单词（输入单词删除）
      4. 展示整个词典内容
      5. 展示当前红黑树结构
      6. 输入单词建立另一棵红黑树
      7. 展示另外一颗红黑树结构
      8. 将红黑树合并
      9. 结束程序

1
请输入一个单词：
bird
抱歉，未查询到该单词！

请按任意键继续. . .
```

发现没有找到该单词

再查找一个词典中没有的单词 kind



D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

- 请输入命令:
1. 查询一个单词 (输入单词查询)
 2. 插入一个单词 (输入单词、翻译、用法)
 3. 删除一个单词 (输入单词删除)
 4. 展示整个词典内容
 5. 展示当前红黑树结构
 6. 输入单词建立另一棵红黑树
 7. 展示另外一颗红黑树结构
 8. 将红黑树合并
 9. 结束程序

4

单词: apple
翻译: 苹果
用法: An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.

单词: boy
翻译: 小鸟
用法: A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.

单词: cat
翻译: 猫
用法: Cats are lions, tigers, and other wild animals in the same family.

单词: general
翻译: 狗
用法: The British are renowned as a nation of dog lovers.

单词: hit
翻译: 击打
用法: Find the exact grip that allows you to hit the ball hard.

单词: journey
翻译: 问题
用法: Agents will raise the issue of prize-money for next year's world championships.

打印成功!

请按任意键继续. . .

(5) 进行红黑树合并的测试，演示为两个词典的合并

首先建立一个新的词典

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

请输入数据建立词典：
格式为:单词、翻译、用法

请输入单词个数：

5

请输入第1个单词及其翻译和用法：

kind 种类 The party needs a different kind of leadership.

请输入第2个单词及其翻译和用法：

lie 躺 There was a child lying on the ground.

请输入第3个单词及其翻译和用法：

mean 意义 Do you mean me?

请输入第4个单词及其翻译和用法：

note 笔记 Stevens wrote him a note asking him to come to his apartment.

请输入第5个单词及其翻译和用法：

object 物体 He squinted his eyes as though he were studying an object on the horizon.

请按任意键继续. . .

然后进行合并

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

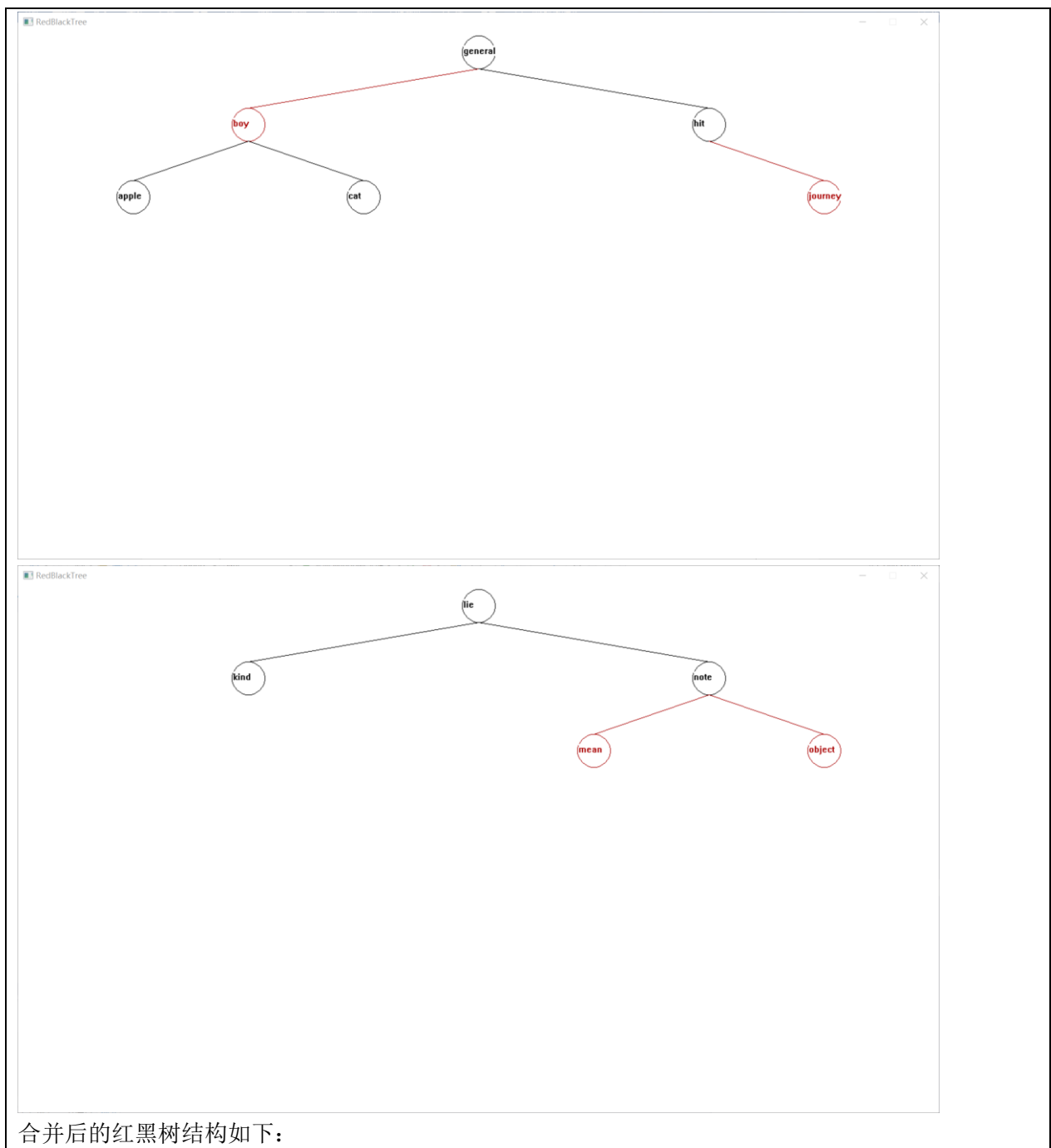
- 请输入命令：
1. 查询一个单词（输入单词查询）
 2. 插入一个单词（输入单词、翻译、用法）
 3. 删除一个单词（输入单词删除）
 4. 展示整个词典内容
 5. 展示当前红黑树结构
 6. 输入单词建立另一棵红黑树
 7. 展示另外一颗红黑树结构
 8. 将红黑树合并
 9. 结束程序

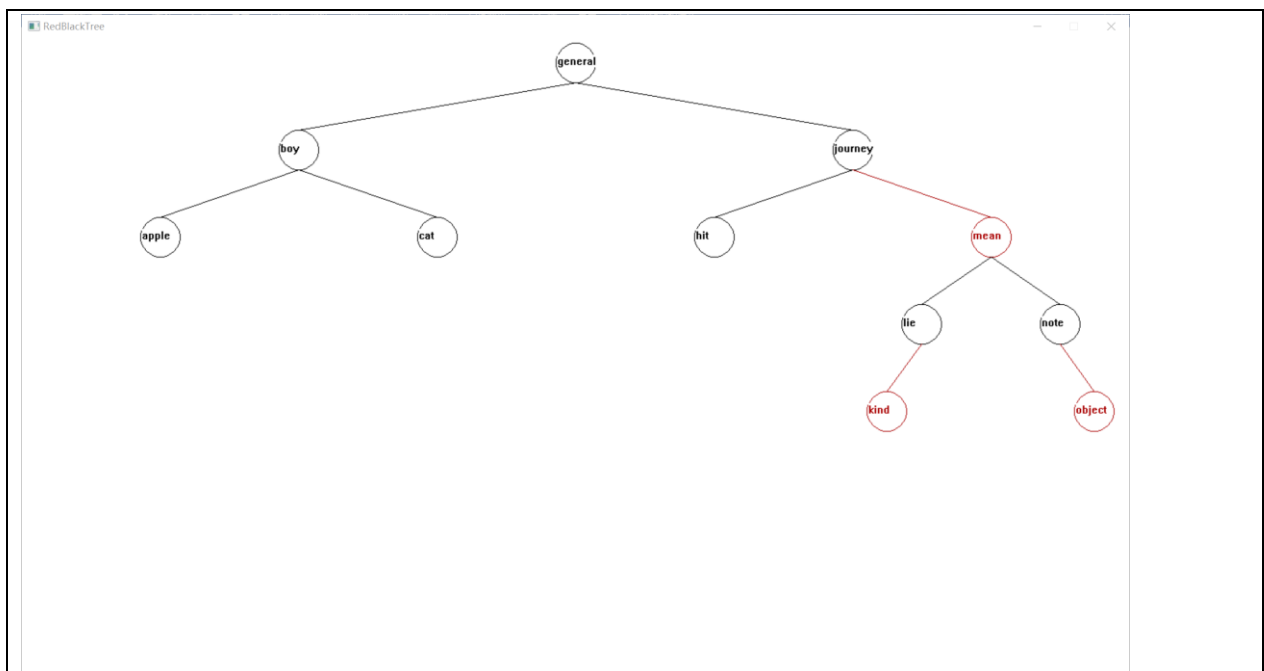
8

合并成功！

请按任意键继续. . .

合并前两个红黑树词典结构如下：





可以看到合并后的红黑树结构发生了旋转、变色等操作，满足了红黑树的两条性质，同时满足二叉搜索树的性质

输出合并后的词典

```
请输入命令:
1. 查询一个单词 (输入单词查询)
2. 插入一个单词 (输入单词、翻译、用法)
3. 删除一个单词 (输入单词删除)
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序

4

单词: apple
翻译: 苹果
用法: An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.

单词: boy
翻译: 小鸟
用法: A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.

单词: cat
翻译: 猫
用法: Cats are lions, tigers, and other wild animals in the same family.

单词: general
翻译: 狗
用法: The British are renowned as a nation of dog lovers.

单词: hit
翻译: 击打
用法: Find the exact grip that allows you to hit the ball hard.

单词: journey
翻译: 问题
用法: Agents will raise the issue of prize-money for next year's world championships.

单词: kind
翻译: 种类
用法: The party needs a different kind of leadership.

单词: lie
翻译: 躺
用法: There was a child lying on the ground.

单词: mean
翻译: 躺
用法: There was a child lying on the ground.

单词: note
翻译: 躺
用法: There was a child lying on the ground.

单词: object
翻译: 躺
用法: There was a child lying on the ground.

打印成功!
```

发现合并后的词典正确，并且其先序遍历是按照字典序递增的

(6) 然后是程序鲁棒性测试，输入命令以外的字符，会提示命令错误，并且不会影响程序正常运行

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe

```
请输入命令：
1. 查询一个单词（输入单词查询）
2. 插入一个单词（输入单词、翻译、用法）
3. 删除一个单词（输入单词删除）
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序
```

asDASDF

请输入正确的命令！

请按任意键继续. . .

（7）程序结束命令

Microsoft Visual Studio 调试控制台

```
请输入命令：
1. 查询一个单词（输入单词查询）
2. 插入一个单词（输入单词、翻译、用法）
3. 删除一个单词（输入单词删除）
4. 展示整个词典内容
5. 展示当前红黑树结构
6. 输入单词建立另一棵红黑树
7. 展示另外一颗红黑树结构
8. 将红黑树合并
9. 结束程序
```

9

程序已结束！

D:\Code\C++\Red Black Tree\RedBlackTree\x64\Debug\RedBlackTree.exe (进程 672)已退出，代码为 0。
按任意键关闭窗口. . .

测试中使用的词典数据如下：

1. apple 苹果 An apple is a round fruit with smooth red, yellow, or green skin and firm white flesh.
2. bird 小鸟 A bird is a creature with feathers and wings. Female birds lay eggs. Most birds can fly.
3. cat 猫 Cats are lions, tigers, and other wild animals in the same family.
4. dog 狗 The British are renowned as a nation of dog lovers.
5. egg 鸡蛋 ...a baby bird hatching from its egg.
6. figure 数字 It will not be long before the inflation figure starts to fall.
7. general 将军 The General's visit to Sarajevo is part of preparations for the deployment of extra troops.
8. hit 击打 Find the exact grip that allows you to hit the ball hard.
9. issue 问题 Agents will raise the issue of prize-money for next year's world championships.
10. journey 旅行 As per our rule, our bus can not stop at her jouney, if you have anything important, please get off the bus at stop.

11. kind 种类 The party needs a different kind of leadership.
12. lie 躺 There was a child lying on the ground.
13. mean 意义 Do you mean me?
14. note 笔记 Stevens wrote him a note asking him to come to his apartment.
15. object 物体 He squinted his eyes as though he were studying an object on the horizon.
16. boy 男孩 He was still just a boy.

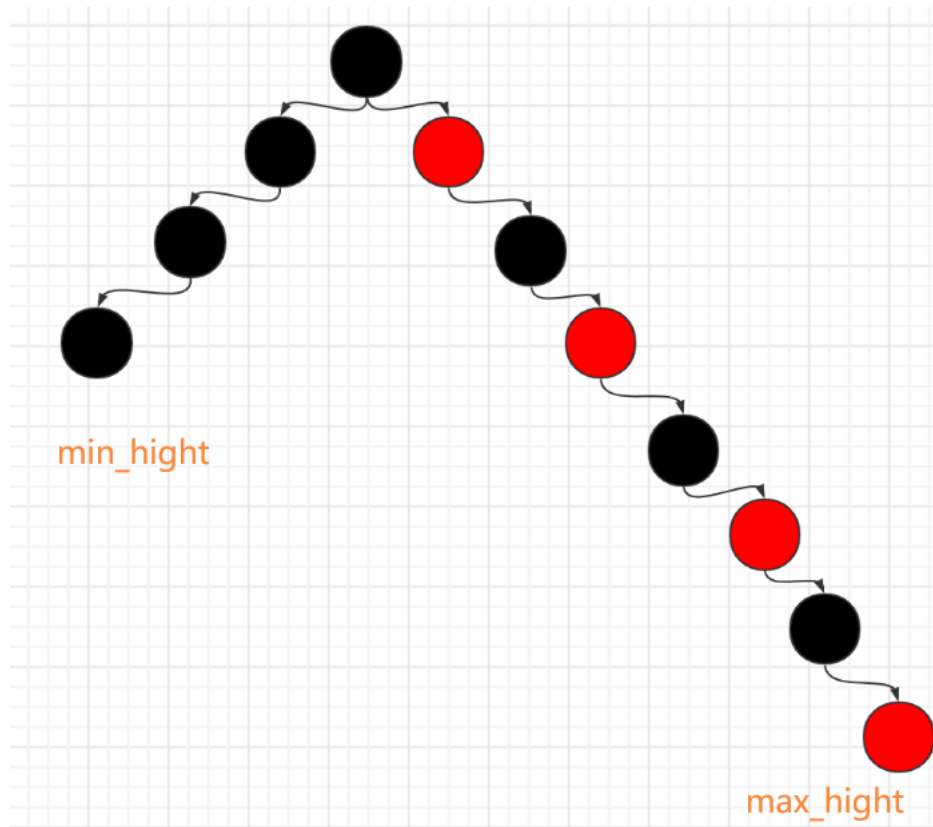
4. 分析与探讨

(1) Q: 红黑树具有怎样的“性质”?

- A: 1. 红黑树每个节点都有一个颜色，红色或黑色
2. 根节点为黑色
3. 如果一个节点为红色，那么它的两个孩子都是黑色的
4. 对于每个节点，从该节点到其后代叶节点的简单路径上，黑色节点数目相同
5. 红黑树每个带有关键词的节点为内部节点，都最下层的内部节点的两个孩子都是外部节点，外部节点为黑色，值为空

(2) Q: 为什么说红黑树是“自平衡”的二叉搜索树?

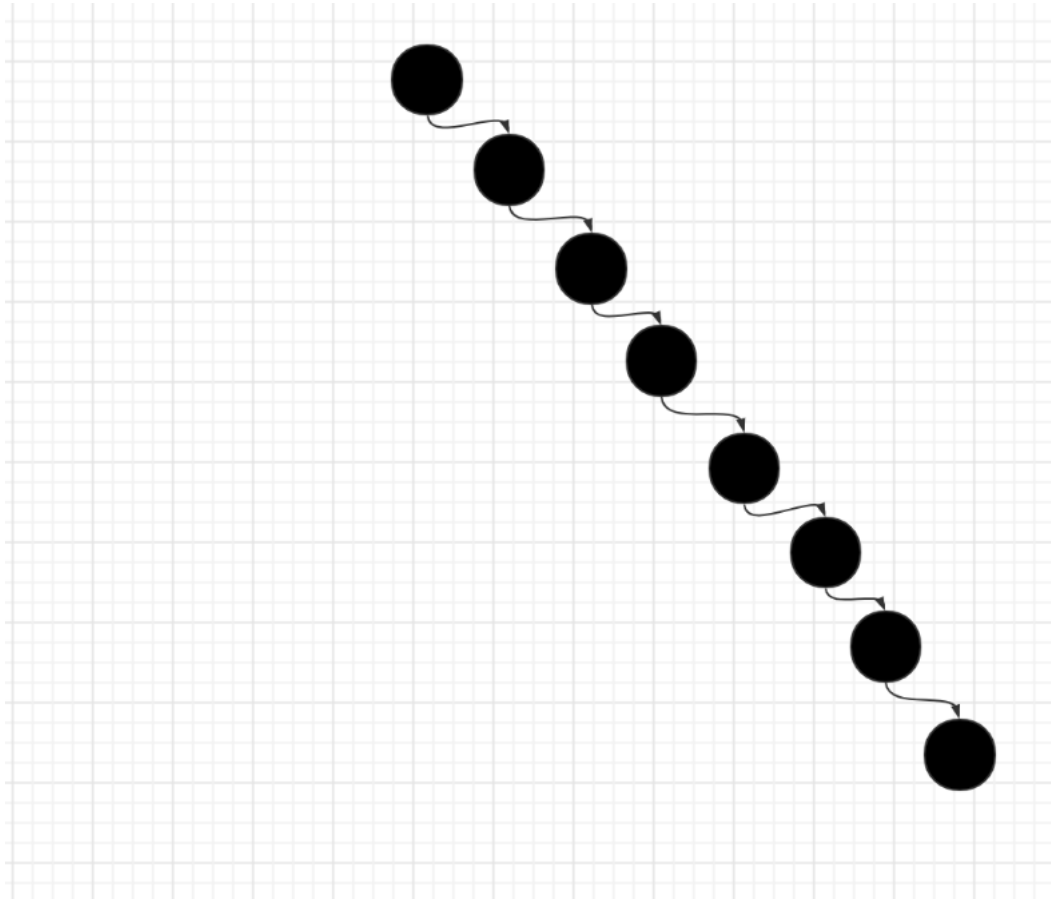
- A: 红黑树的“五条性质”保证了红黑树是“自平衡”的二叉搜索树。因为在极限情况下，由于红色节点不能够连续出现，因此最多可能会出现红黑树子树一端为红黑交替，一端全为黑色，并且两端黑色节点数目相同。此时红黑交替一端树高更大，全黑一端树高更小，因此 $\max_height \leq 2 * \min_height$ 。本身红黑树的查找、插入、删除的时间复杂度都为 $O(n)$ ，在所说的极限情况（即时间复杂度“最坏”的情况下），其复杂度也不会超过原来的两倍（即 $2 * O(n)$ ），可以忽略不计。因此红黑树只需要满足五条性质就能保证其是自平衡的。
“最坏情况”图示如下：



(3) Q: 二叉搜索树和平衡树在插入、删除、查找操作时理论时间复杂度都为 $O(\log n)$ ，什么使用平衡树而不使用二叉搜索树？

A: 因为二叉搜索树假如按照升序或者降序进行插入，即在树型结构中一直在“同一个方向”插入，那么二叉搜索树的树形结构会退化为链表结构，查找、插入、删除的时间复杂度会退化为 $O(n)$ 。而平衡树可以通过在插入和删除之后对当前不优秀的树形结构进行调整，保证比较优秀的树形结构，进而保证插入、删除、查找操作时间复杂度维持为 $O(\log n)$

“最坏情况”图示如下：



(4) Q: AVL 树和红黑树都为平衡树，试比较其不同。

A: 1. 红黑树依靠“约定”的性质并且维护性质来维持平衡，任何不平衡都可以在三次旋转之内解决；AVL 树通过很多很多次旋转维持平衡
2. 红黑树不求完全的平衡；AVL 树讲究完全平衡
3. 红黑树在最坏情况下复杂度比 AVL 树低，但是红黑树的平均性能要 AVL 树高，因为 AVL 可能需要很多 $\log n$ 的旋转

(5) Q: 红黑树有哪些实际的应用？

A: c++标准 STL 库里面 map 和 set 都是使用红黑树结构进行实现的，可以稳定的进行时间复杂度为 $O(\log n)$ 的查找、插入和删除。如果需要进行大量的插入、查找、删除中两种以上的操作，那么红黑树的平均时间复杂度无疑是最优秀的，因此也十分适合用作建立本实验中的英汉词典索引。

5. 附录：实现源代码

RedBlackTree.h //红黑树的定义程序

```

#pragma once
#include <string>
#include<iostream>
#include<cstdio>
#include<fstream>
#include<easyx.h>
#include<graphics.h>
#include<queue>
#include <windows.h>
#include<sstream>
using namespace std;

struct word { //单词
    string key; //英文单词
    string translation; //中文翻译
    vector<string> usage; //用法例句
    //三个构造函数：有参、无参、复制构造
    word() {};
    word(string k, string t, vector<string> u) :key(k), translation(t), usage(u) {};
    word(const word& w) :key(w.key), translation(w.translation), usage(w.usage) {};
    //重载运算符：<、>、==、=、<<
    bool operator < (const word& w) {
        return this->key < w.key;
    }
    bool operator > (const word& w) {
        return this->key > w.key;
    }
    bool operator == (const word& w) {
        return this->key == w.key;
    }
    void operator = (const word& w) {
        this->key = w.key;
    }
    friend ostream& operator << (ostream& os, const word& w) {
        os << "单词： " << w.key << "\n";
        os << "翻译： " << w.translation << "\n";
        os << "用法： "; for (auto& i : w.usage) os << i << " "; os << "\n";
        return os;
    }
};

struct Node { //红黑树节点
    bool color = 0; //节点颜色，0 为红色，1 为黑色，默认为 0

```

```

word value;//关键词
Node* left = NULL; //左孩子
Node* right = NULL;//右孩子
Node* father = NULL;//父亲
//以下为四个构造函数
Node() {};
Node(word v) :value(v) {};
Node(word v, Node* f) :value(v), father(f) {};
Node(word v, Node* f, Node* l, Node* r) :value(v), father(f), left(l), right(r) {};
};

class RedBlackTree { //红黑树类
public:
    RedBlackTree();//构造函数
    ~RedBlackTree();//析构函数
    Node* find(string value);//查找
    bool insert(word value, int ord);//插入
    void insertBalance(Node* s);//插入调整
    bool erase(string value);//删除
    void eraseBalance(Node* x);//删除调整
    void print(Node* now);//打印
    void show();//基本操作演示
    void leftRotate(Node* z);//左旋
    void rightRotate(Node* z);//右旋
    void change(Node* u, Node* v);//将 v 替换 u
    Node* minNum(Node* x); //找到 x 开始最小关键词的节点
    void destory(Node* now);//递归销毁整棵树
    void merge(RedBlackTree& t);//红黑树合并
    Node* getRoot();//获得根
private:
    Node* root;//根
    Node* exNode;//外部节点
};

```

RedBlackTree.cpp //红黑树的实现程序

```

#include "RedBlackTree.h"

RedBlackTree::RedBlackTree() { //构造函数
    exNode = new Node();//申请一个外部节点
    exNode->color = 1;
    root = exNode;//根节点指向外部节点
}

```



```

RedBlackTree::~RedBlackTree() { //析构函数
    destory(root);
}

Node* RedBlackTree::find(string value) { //寻找一个节点
    Node* s = root;
    while (s != exNode) {
        if (s->value.key == value)
            return s;
        if (value < s->value.key) {
            s = s->left;
        }
        else {
            s = s->right;
        }
    }
    return NULL; //没找到
}

bool RedBlackTree::insert(word value, int ord) { //插入节点
    Node* pr = exNode; //s 的父亲
    Node* s = root; //当前节点
    while (s != exNode) {
        if (value == s->value)
            return false; //找到相同元素，返回
        pr = s;
        if (value < s->value) { //在左子树
            s = s->left;
        }
        else { //在右子树
            s = s->right;
        }
    }
    s = new Node(value, pr, exNode, exNode); //申请新节点，两个孩子都是外部节点
    if (pr == exNode) { //空树
        root = s;
    }
    else {
        if (value < pr->value) { //在左子树
            pr->left = s;
        }
        else { //在右子树
            pr->right = s;
        }
    }
}

```

```

        s->father = pr;
    }
    insertBalance(s); //插入平衡
    if (ord == 2)
        show();
    return true; //找到
}

void RedBlackTree::insertBalance(Node* s) {
    Node* uncle; //叔叔节点, 当前节点的父亲的另外一个孩子
    while (s->father->color == 0) {
        if (s->father == s->father->father->left) { //当前为父亲的左孩子
            uncle = s->father->father->right;
            if (uncle->color == 0) { //情况三
                s->father->color = 0;
                uncle->color = 0;
                s->father->father->color = 1;
                s = s->father->father;
            }
            else {
                if (s == s->father->right) { //情况四
                    s = s->father;
                    leftRotate(s);
                }
                s->father->color = 1; //情况五
                s->father->father->color = 0;
                rightRotate(s->father->father);
            }
        }
        else { //镜像另一半
            uncle = s->father->father->left;
            if (uncle->color == 0) {
                s->father->color = 1;
                uncle->color = 1;
                s->father->father->color = 0;
                s = s->father->father;
            }
            else {
                if (s == s->father->left) {
                    s = s->father;
                    rightRotate(s);
                }
                s->father->color = 1;
                s->father->father->color = 0;
            }
        }
    }
}

```

```

        leftRotate(s->father->father);
    }
}
root->color = 1;
}

bool RedBlackTree::erase(string value) { //删除节点
    Node* z;
    z = find(value);
    if (z == NULL) //没找到
        return false;
    if (z != exNode) { //不是外部节点
        Node* x = exNode;
        Node* y;
        word ycolor;
        if (z->right == exNode) { //右子树为外部节点
            eraseBalance(z);
            change(z, z->left);
        }
        else {
            y = minNum(z->right);
            z->value = y->value;
            eraseBalance(y);
            change(y, y->right);
        }
    }
    else {
        cout << "删除失败，不存在该节点" << endl;
    }
    return true;
}

void RedBlackTree::eraseBalance(Node* x) {
    while (x != root && x->color != 0) {
        if (x == x->father->left) { //当前节点在父亲节点左侧
            Node* brother = x->father->right;
            if (brother->color == 0) { //情况三
                x->father->color = 0;
                brother->color = 1;
                leftRotate(x->father);
                brother = x->father->right;
            }
            if (brother->left->color == 1 && brother->right->color == 1) { //情况四

```

```

        brother->color = 0;
        if (x->father->color == 0)
            x->father->color = 1, x = root;
        else
            x = x->father;
    }
    else if (brother->right->color == 1) { //情况五
        brother->color = 0;
        brother->left->color = 1;
        rightRotate(brother);
        brother = x->father->right;
    }
    else { //情况六
        brother->color = x->father->color;
        x->father->color = 1;
        brother->right->color = 1;
        leftRotate(x->father);
        x = root;
    }
}
else { //镜像另一半
    Node* brother = x->father->left;
    if (brother->color == 0) {
        x->father->color = 0;
        brother->color = 1;
        rightRotate(x->father);
        brother = x->father->left;
    }
    if (brother->left->color == 1 && brother->right->color == 1) {
        brother->color = 0;
        if (x->father->color == 0)
            x->father->color = 1, x = root;
        else
            x = x->father;
    }
    else if (brother->left->color == 1) {
        brother->color = 0;
        brother->right->color = 1;
        leftRotate(brother);
        brother = x->father->left;
    }
    else {
        brother->color = x->father->color;
        x->father->color = 1;
    }
}

```

```

        brother->left->color = 1;
        rightRotate(x->father);
        x = root;
    }
}
}

void RedBlackTree::change(Node* u, Node* v) { //将 u 的位置替换为 v
    if (u->father == exNode) {
        root = v;
    }
    else if (u == u->father->left) {
        u->father->left = v;
    }
    else {
        u->father->right = v;
    }
    v->father = u->father;
}

Node* RedBlackTree::minNum(Node* x) { //找到当前节点子树中值最小的节点
    if (x->left == exNode) //找到了
        return x;
    return minNum(x->left);
}

/* 左旋
    zp          zp
    /           /
   z           y
  /\          /\
 lz  y       z  ry
  /\        /\
 ly  ry     lz ly
*/

void RedBlackTree::leftRotate(Node* z) { //以 z 节点为中心左旋，其原理图如上
    Node* y = z->right;
    z->right = y->left;
    if (y->left != exNode) { //y 左孩子不空
        y->left->father = z;
    }
    y->father = z->father;
    if (root == z) { //z 为根节点

```

```

        root = y;
    }
    else if (z == z->father->left) { //z 为父亲的左孩子
        z->father->left = y;
    }
    else { //z 为父亲的右孩子
        z->father->right = y;
    }
    y->left = z;
    z->father = y;
}

/* 左旋
        zp                zp
        /                  /
       z                  y
      /\                  /\
     y  rz                ly z
    /\                    /\
   ly ry                 ry rz
*/

void RedBlackTree::rightRotate(Node* z) { //以 z 为中心右旋，原理图如上
    Node* y = z->left;
    z->left = y->right;
    if (y->right != exNode) { //y 的右孩子不为空
        y->right->father = z;
    }
    y->father = z->father;
    if (z == root) { //z 为根节点
        root = y;
    }
    else if (z == z->father->left) { //z 为父亲的左孩子
        z->father->left = y;
    }
    else { //z 为父亲的右孩子
        z->father->right = y;
    }
    y->right = z;
    z->father = y;
}

void RedBlackTree::print(Node* now) { //先序遍历打印出整个红黑树的节点单词
    if (now->left != exNode)
        print(now->left);

```

```

cout << "\n-----\n" << now->value <<
"-----\n";

    if (now->right != exNode)
        print(now->right);
    //cout << endl;
}

void RedBlackTree::show() { //画出整个红黑树，进行操作演示
    initgraph(1400, 800); //初始化画布
    setbkcolor(WHITE); //画布颜色为白色
    setlinecolor(BLACK); //线条颜色为黑色
    setorigin(700, 30); //初始化原点位置
    settextrcolor(BLACK); //字体颜色为黑色
    cleardevice(); //清空画布
    struct cir { //节点结构体
        Node* node; //红黑树节点
        int x, y; //坐标
        string s; //单词
        bool color; //1 为黑色
        int cnt = 1; //第几行
    };
    queue<cir> q;
    cir newNode;
    newNode.x = 0, newNode.y = 0, newNode.color = 1, newNode.s = root->value.key, newNode.node = root; //根节点
    q.push(newNode);
    while (q.size()) { //层次遍历画出整颗红黑树
        cir now = q.front();
        q.pop();
        if (now.color == 1) //红色节点
            setlinecolor(BLACK), settextrcolor(BLACK);
        else //黑色节点
            setlinecolor(RED), settextrcolor(RED);
        circle(now.x, now.y, 25); //画出一个○作为节点
        wchar_t s[100];
        for (int i = 0; i < now.s.length(); i++)
            s[i] = now.s[i];
        s[now.s.length()] = '\0';
        outtextxy(now.x - 23, now.y - 10, s); //在节点内填入单词
        if (now.node->left != exNode) { //左孩子
            //给新节点属性赋值
            newNode.x = now.x - 1400 / pow(2, now.cnt + 1), newNode.y = now.y + 110, newNode.s =
            now.node->left->value.key, newNode.node = now.node->left, newNode.color = now.node->left->color, newNode.cnt =
            now.cnt + 1;

            if (newNode.color == 1)

```

```

        setlinecolor(BLACK);
    else
        setlinecolor(RED);
    line(now.x, now.y + 25, newNode.x, newNode.y - 25);
    q.push(newNode);
}
if (now.node->right != exNode) { //右孩子
    //给新节点属性赋值
    newNode.x = now.x + 1400 / pow(2, now.cnt + 1), newNode.y = now.y + 110, newNode.s =
now.node->right->value.key, newNode.node = now.node->right, newNode.color = now.node->right->color, newNode.cnt =
now.cnt + 1;
    if (newNode.color == 1)
        setlinecolor(BLACK);
    else
        setlinecolor(RED);
    line(now.x, now.y + 25, newNode.x, newNode.y - 25);
    q.push(newNode);
}
}
/*      EndBatchDraw();*/
_getwch();//暂停，按任意键继续
closegraph();//关闭画布
}

void RedBlackTree::destory(Node* now) { //销毁整颗红黑树，释放内存空间
    if (now == exNode)
        return;
    if (now->left != exNode) //有左孩子
        destory(now->left);
    else
        destory(now->right); //有右孩子
    delete now;
    now = NULL;
}

void RedBlackTree::merge(RedBlackTree& t) //红黑树合并
{
    while (t.getRoot()->left->value.key != "" && t.getRoot()->right->value.key != "") { //将 t 的节点从根节点开始取出并且
插入到当前的红黑树
        //cout << "!" << t.getRoot()->left->value.key << " " << t.getRoot()->right->value.key << endl;
        insert(t.getRoot()->value, 1);
        t.erase(t.getRoot()->value.key);
        /*      t.show();*/
    }
}

```



```

    if (t.getRoot()->left->value.key != "") { //把最后剩余的节点取出插入
        insert(t.getRoot()->left->value, 1);
        insert(t.getRoot()->value, 1);
    }
    else if (t.getRoot()->right->value.key != "") {
        insert(t.getRoot()->right->value, 1);
        insert(t.getRoot()->value, 1);
    }
    else {
        insert(t.getRoot()->value, 1);
    }
}

```

Node* RedBlackTree::getRoot()//返回根节点

```

{
    return root;
}

```

Main.cpp //红黑树基本操作演示、创建小型英汉词典索引的测试主程序

```
#include "RedBlackTree.h"
```

```
int main() {
```

```
    RedBlackTree t,t1;//两颗红黑树
```

```
    cout << "-----\n"
```

```
        "|                      请输入数据建立词典:                \n"
```

```
        "|                      格式为:单词、翻译、用法            \n"
```

```
        "-----\n";
```

```
    int n;//单词个数
```

```
    cout << "请输入单词个数:\n";
```

```
    cin >> n;
```

```
    for (int i = 1; i <= n; i++) { //输入单词、翻译、用法，并且插入红黑树
```

```
        cout << "请输入第" << i << "个单词及其翻译和用法:\n";
```

```
        string key, translation;
```

```
        cin >> key >> translation;
```

```
        string read;
```

```
        getline(cin, read, '\n');
```

```
        stringstream ss(read);
```

```
        vector<string> usage;
```

```
        while (ss >> read) {
```

```
            usage.push_back(read);
```

```
        }
```

```

bool f = t.insert(word{ key, translation, usage },0);
if (f == false)
    cout << "插入失败，存在重复值！";
}
while (1) {
    system("cls");//清空，放出菜单
    cout << "-----\n"
           "|                请输入命令：                |\n"
           "|                1.查询一个单词（输入单词查询）        |\n"
           "|                2.插入一个单词（输入单词、翻译、用法）    |\n"
           "|                3.删除一个单词（输入单词删除）        |\n"
           "|                4.展示整个词典内容                    |\n"
           "|                5.展示当前红黑树结构                  |\n"
           "|                6.输入单词建立另一棵红黑树            |\n"
           "|                7.展示另外一颗红黑树结构              |\n"
           "|                8.将红黑树合并                        |\n"
           "|                9.结束程序                            |\n"
           "-----\n";

    string ord;//命令
    cin >> ord;
    if (ord == "1") { //查找单词
        string word1;
        cout << "请输入一个单词：\n";
        cin >> word1;
        Node* now = t.find(word1);
        if (now != NULL) { //查找成功，输出
            cout
            <<
            "\n-----\n"<<t.find(word1)->value<<
            "-----\n";
        }
        else//查找失败
            cout << "抱歉，未查询到该单词！\n";
    }
    else if (ord == "2") { //插入单词
        string key, translation;
        cout << "请输入一个单词、翻译、用法：\n";
        cin >> key >> translation;
        string read;
        getline(cin, read, '\n');
        stringstream ss(read);
        vector<string> usage;
        while (ss >> read) {
            usage.push_back(read);
        }
    }
}

```

```

    bool flag = t.insert(word{ key,translation,usage },0);
    if (flag)//插入成功
        cout << "插入成功! \n";
    else//插入失败
        cout << "插入失败, 存在重复单词! \n";
}

else if (ord == "3") { //删除单词
    string word1;
    cout << "请输入一个单词: \n";
    cin >> word1;
    bool flag = t.erase(word1);
    if (flag)//删除成功
        cout << "删除成功! \n";
    else//删除失败
        cout << "抱歉, 删除失败! \n";
}

else if (ord == "4") { //打印词典结构
    t.print(t.getRoot());
    cout << "打印成功! \n";
}

else if (ord == "5") { //展示红黑树结构
    t.show();
}

else if (ord == "6") { //形成另一颗红黑树
    system("cls");
    cout << "-----\n"
        << "|                请输入数据建立词典:                |\n"
        << "|                格式为:单词、翻译、用法                |\n"
        << "-----\n";

    cout << "请输入单词个数:\n";
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cout << "请输入第" << i << "个单词及其翻译和用法: \n";
        string key, translation;
        cin >> key >> translation;
        string read;
        getline(cin, read, '\n');
        stringstream ss(read);
        vector<string> usage;
        while (ss >> read) {
            usage.push_back(read);
        }
        bool f = t1.insert(word{ key, translation, usage }, 0);
        if (f == false)

```

```
        cout << "插入失败，存在重复值！";
    }
}
else if (ord == "7") { //展示另外一棵红黑树
    t1.show();
}
else if (ord == "8") { //将另外一颗红黑树合并到当前红黑树
    t.merge(t1);
    cout << "合并成功！ \n";
} else if (ord == "9") { //结束程序
    cout << "程序已结束！ \n";
    break;
}
else { //错误命令
    cout << "请输入正确的命令！ \n";
}
cout << "\n\n";
system("pause"); //暂停
}
```

```
}
```