# 红黑树插入算法实验报告

## 一、实验目的

通过实践，加深理解红黑树的性质、特点，熟悉其插入旋转等相关操作，提高编程能力。

## 二、实验内容

1. 编码实现红黑树的插入算法，使得插入后依旧保持红黑性质，即实现 RB_INSERT, RB_INSERT_FIXUP 算法。
2. 从 insert.txt 文件中按照指定格式读取数据并构建一棵红黑树，再对这棵红黑树进行层次遍历，将遍历结果按照指定格式输出到 LOT.txt 文件中。

## 三、实验步骤

### 1. 定义红黑树的数据结构：

我将红黑树的数据结构定义为一个两层结构：红黑树的节点和树。

#### (1) 红黑树节点

将红黑树节点定义为一个 struct，命名为 TNode，其内部包含五个字段：分别为 key，left 指针，right 指针，p 指针以及 color。

```
typedef bool colortype;
const colortype black = false;
const colortype red = true;

typedef pair<int,vector<int> > Input;
struct TNode{
    int key;
    TNode* left;
    TNode* right;
    TNode* p;
    colortype color;
};
```

#### (2) 红黑树

将红黑树的树结构定义为一个 class，命名为 RBTree，其私有属

性包括一个 nil 指针和一个 root 指针，分别指向红黑树的 nil 节点和根节点。另外，RBTree 中还声明了一个默认构造函数 RBTree()，四个友元函数 RB_Insert()、RB_Insert_Fixup()、left_rotate()和 right_rotate()以及 Nil()和 Root()。

```
34  class RBTree{
35      public:
36          RBTree()
37          {
38              nil = CreatNode(-1);    //nil节点的key为-1
39              nil->color = black;
40              root = nil;
41          }
42          friend RBTree& RB_Insert(RBTree& , TNode* );
43          friend RBTree& RB_Insert_Fixup(RBTree& , TNode* );
44          friend RBTree& left_rotate(RBTree&, TNode*);
45          friend RBTree& right_rotate(RBTree&, TNode*);
46          TNode* Nil();
47          TNode* Root();
48      private:
49          TNode* nil; //nil结点的key为0
50          TNode* root;
51  };
```

### (3) 构造函数 RBTree()

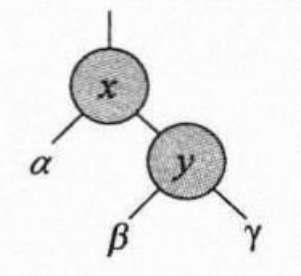
它用于对一棵红黑树进行初始化。首先它会调用 CreatNode()函数来生成一个 nil 节点，可以看到，在这里我将 nil 节点的关键字设为-1 以与其他内部结点作区分，同时将 nil 节点的左右孩子以及父亲指针均置为空，由于新生成的节点颜色均为 red，因此在构造函数中要显式地将 nil 节点的颜色置为 black，最后将 root 指针指向这个 nil 节点，此时红黑树的初始化就完成了。

```
36          RBTree()
37          {
38              nil = CreatNode(-1);    //nil节点的key为-1
39              nil->color = black;
40              root = nil;
41          }
```

```
TNode* CreatNode(int k)
{//新生成一个结点
    TNode* pointer = new TNode;
    pointer->key = k;
    pointer->left = nullptr;
    pointer->right = nullptr;
    pointer->p = nullptr;
    pointer->color = red;
    return pointer;
}
```

(4) left_rotate()和 right_rotate()

right_rotate()所做的操作与 left_rotate()完全对称，因此，这里只介绍 left_rotate()是如何设计的：




对上图中 x 节点进行左旋操作，首先令指针 y 指向 x 的右孩子，再将 y 的左孩子链到到 x 的右孩子指针上，如果 y 的左孩子不为空，还要将 y 的左孩子的父指针链到 x 上，再将 y 的父指针链到 x 的父指针上，如果 x 的父指针指向 nil 节点，说明 x 是 root 节点，还要将此红黑树的根节点设为 y(即令 root 指针指向 y)，如果 x 不根节点，那么 x 是其父节点的左孩子或右孩子，若是左孩子，将 x 的父节点的左孩子指针指向 y，否则将右孩子指针指向 y，再将 y 的左孩子指针指向 x，最后令 x 的父指针指向 y，并返回这棵旋转过后的红黑树。

```
63  RBTree& left_rotate(RBTree& T, TNode* x)
64  {
65      TNode* y = x->right;
66      x->right = y->left;
67      if(y->left != T.nil)
68          y->left->p = x;
69      y->p = x->p;
70      if(x->p==T.nil)
71          T.root = y;
72      else if(x==x->p->left)
73          x->p->left = y;
74      else
75          x->p->right = y;
76      y->left = x;
77      x->p = y;
78
79      return T;
80  }
```

(5) RB_Insert()

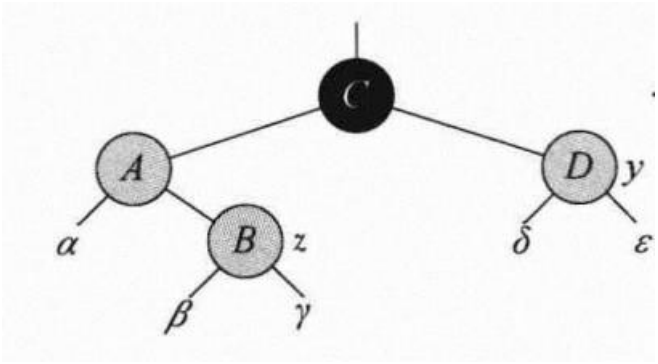
红黑树的插入操作与二叉树极其类似，只不过红黑树在插入操作之后还要进行一步 FixUp 操作，以维护它的红黑性质。在向红黑树 T 插入一节点 z 时，首先要寻找插入位置，先令指针 y 指向 nil 节点，再令 x 指向 root 节点，再执行 while 循环：令 y 指针指向 x（为叙述方便，后面指针指向的节点均以指针代替），如果待插入节点 key 小于 x 的 key，那么往左子树上找插入位置，否则往右子树上找插入位置。当退出 while 循环时，y 指针指向插入位置的父节点，故令 z 的父指针指向 y，如果此时 y(插入位置的父节点)是 nil 节点，说明插入位置是根节点的位置，故要将 root 指针指向 z，若插入位置是其父节点的左孩子，则将 y 的左孩子指向 z，否则将 y 的右孩子指向 z。接下来将插入结点的左右孩子指向 nil 节点,并将其颜色设为 red。最后执行 FixUp 操作并返回插入结点并调整之后的树 T。

```
154  RBTree& RB_Insert(RBTree& T, TNode* z)
155  {
156      TNode* y = T.nil;
157      TNode* x = T.root;
158      while(x!=T.nil)
159      {
160          y = x;
161          if(z->key < x->key)
162              x = x->left;
163          else
164              x = x->right;
165      }
166      z->p = y;
167      if(y==T.nil)
168          T.root = z;
169      else if(z->key < y->key)
170          y->left = z;
171      else
172          y->right = z;
173      z->left = T.nil;
174      z->right =T.nil;
175      z->color = red;
176
177      RB_Insert_Fixup(T, z);
178
179      return T;
180  }
```

(6)　RB_Insert_Fixup()

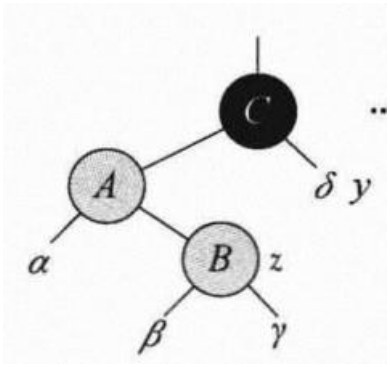
在执行 FixUp 操作时，分为 6 种情况,根据插入结点 z 的父节点是 z 的祖父节点的左孩子还是右孩子又分为前三种和后三种情况两大类，其中 case 4、case 5、case 6 与前三种情况对称，故这里只讨论前三种情况：

Case1：



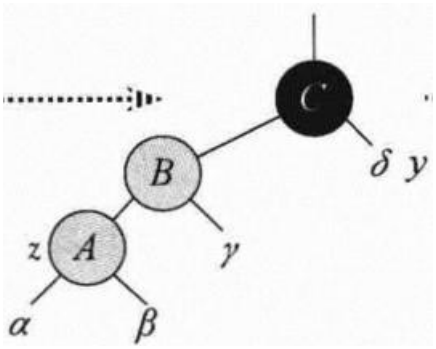

如上图所示，插入结点 z 的父亲和叔叔 y 均是红结点，将 z 的父亲和叔叔 y 的颜色均置为 black，再将 z 的祖父节点的颜色置为 red。case1 执行结束之后可能还会转入 case2 或 case3。

Case2：




如上图所示，当 z 的父节点为 red，但叔叔 y 为 black，且 z 为其父节点的右孩子时属于 case2，首先将 z 指针指向它的父节点，再对新的 z 指针指向的节点执行左旋操作，并转入对 case2 的处理。

Case3：




如上图所示，插入结点 z 和其父亲均为 red，其叔叔 y 为黑色，首先 z 的父亲变为 black，再将 z 的祖父变为 red，接着对 z 的祖父执行右旋操作。

在完成一次调整之后，指针 z 会在树中上移，此时如果 z 指向的新节点和其父亲均为红色，此时还要继续执行上面的步骤，直到 z 的父亲节点为 black，退出这个循环调整过程。Fixup 的最后一步操作是将根节点的颜色置为 black，之所以要这样做是因为可能出现这样两种情况：①插入的节点为树中的第一个节点，②指针 z 上溯到根节点，这两种情况中 z 的父节点均为 black，但是 root 节点为 red，因此我们加上一步将 root 节点调黑的操作。最后返回调整之后的红黑树。

```
RBTree& RB_Insert_Fixup(RBTree& T, TNode* z)
{
    while(z->p->color==red)
    {
        if(z->p==z->p->p->left)
        {
            TNode* y = z->p->p->right;
            if(y->color==red)              //case 1
            {
                z->p->color = black;  //case 1
                y->color = black;  //case 1
                z->p->p->color = red;  //case 1
                z = z->p->p;   //case 1
            }
            else
            {
                    if(z==z->p->right) //case 2
                    {
                        z = z->p;  //case 2
                        left_rotate(T, z);   //case 2
                    }
                    z->p->color = black;    //case 3
                    z->p->p->color = red;  //case 3
                    right_rotate(T, z->p->p);   //case 3
            }
        }
```

```
        else{
            TNode* y = z->p->p->left;
            if(y->color==red)              //case 4
            {
                z->p->color = black;//case 4
                y->color = black;//case 4
                z->p->p->color = red;//case 4
                z = z->p->p;//case 4
            }
            else
            {
                if(z==z->p->left) //case 5
                {
                    z = z->p; //case 5
                    right_rotate(T, z); //case 5
                }
                z->p->color = black;    //case 6
                z->p->p->color = red;  //case 6
                left_rotate(T, z->p->p);    //case 6
            }
        }

    }
    T.root->color = black;
    return T;
}
```

## (7) 输入

设计 Read()函数读取输入数据，由于输入数据分两行保存在.txt 文件中，首先调用函数 getline()按行读取数据，读取到的每行数据都是一行字符串。接下来又设计了 split()函数，用于实现对字符串进行分割,其中 split()函数的分割功能是通过调用 C 库函数 strtok()实现的。

Strtok()函数的调用格式如下所示：

```
char * strtok ( char * str, const char * delimiters );
```

可以通过对 strtok( )函数的一连串调用来将输入字符串 str 分割为一个个 token——一串由 delimiters 字符串中的某个子串分割出来的连续字符。在第一次调用 strtok()函数时，需要给 strtok()函数的第一个参数传入待分割的字符串 str，它的第一个字符会作为寻找 token 的扫描起点，当扫描出一个 token 时，返回这个 token 的开始位置。在接下来的调用中，strtok 函数的第一个参数均传入 null 指针，函数会从上个 token 的结束位置的下个位置开始扫描。当扫描到 str 字符串的结束标志 null 字符时停止分割。此时，在接下来的对 strtok 函数的调用中，返回的均是 null 指针。

实现完字符串分割以后，再通过调用 C 库函数 atoi()将每个待输

入的以字符串表示的关键字转换成对应的整数形式，此时这些数据就可以用于构建一棵红黑树了。

```
vector<vector<int> > Read(){ //读取txt文件，处理成一个vector<vector<int>>--二维整型数组
    ifstream Infile("insert.txt");
    vector<string> vec_str; //vec_str数组用于存储从data.txt文件中读取的每一行的内容--每一行的内容都是字符串
    vector<vector<string> > vec_substr; //vec_substr数组用于存储对vec数组中每一行分割过后的内容
    vector<vector<int> > vec_result;  //vec2用于存储将vec_substr中每个字符串转换为对应整数的形式
    string s; //字符串s用于存储从.txt文件中读取的每行内容
    while(getline(Infile,s)) //按行读取txt文件的内容--读取后的每行内容是字符串，存于vec_str中，
    {
        vec_str.push_back(s);
    }

    for(int i=0; i<vec_str.size(); i++) //对读取进来的每行原始字符串进行分割
    {
        string s;
        vec_substr.push_back({});
        vec_substr[vec_substr.size()-1] = split(vec_str[i], " "); //对vec数组的每一行按空格进行分割,原vec中每行字符串会被分割为多个字符串

    }

    for(int i=0; i<vec_substr.size(); i++) //将vec_substr中每个字符串转换为对应的整数
    {
        vec_result.push_back({});
        for(int j=0; j<vec_substr[i].size(); j++)
        {
            char * strs = new char[vec_substr[i][j].length() + 1] ;//vec1中每个元素都是string类型的，下面要调用的库函数atoi的接口要求其传入参数是C类型的字符串
            strcpy(strs, vec_substr[i][j].c_str());
            vec_result[vec_result.size()-1].push_back(atoi(strs)); //调用库函数atoi
        }
    }
    return vec_result;
}
```

```
vector<string> split(const string& str, const string& delim) //将输入字符串str按delim标志进行分割
{
    vector<string> vec_result;
    if("" == str) return vec_result; //如果要分割的串为空串，那么不进行分割

    //先将要切割的字符串从string类型转换为char*类型，之所以要这样做，是因为下面要调用的C库函数strtok的接口是这样要求的
    char * strs = new char[str.length() + 1] ;
    strcpy(strs, str.c_str());

    //同上
    char * d = new char[delim.length() + 1];
    strcpy(d, delim.c_str());

    char *p = strtok(strs, d); //调用C库函数strok()进行字符串分割
    while(p) {
        string s = p; //分割得到的字符串转换为string类型
        vec_result.push_back(s); //存入结果数组
        p = strtok(NULL, d); //这里之所以要这样写是因为strtok函数是多次调用的，详情见CSDN
    }

    return vec_result;
}
```

## (8)　层次遍历及输出

采用层次遍历来搜索前面建立好的红黑树，首先新建一个LOT.txt文件用于输出，在层次遍历过程中，每次遍历到一个节点首先要判断它是 nil 节点还是内部结点，如果是 nil 节点就输出“nil”，并换行继续输出，如果是内部结点，就输出节点的 key 值以及节点颜色，并换行继续输出。

```cpp
void LevelOrderTraverse(RBTree& T)
{
    ofstream outfile("LOT.txt");
    queue<TNode*> q;
    TNode* p;
    if(T.Root() != T.Nil())
        q.push(T.Root());
    while(q.size())
    {
        p = q.front();
        if(p==T.Nil())
            outfile<<"nil"<<endl;
        else if(p->color==red)
        {
            outfile<<p->key<<", "<<"red"<<endl;
        }
        else if(p->color==black)
        {
            outfile<<p->key<<", "<<"black"<<endl;
        }
        q.pop();
        if(p->left)
            q.push(p->left);
        if(p->right)
            q.push(p->right);
    }
}
```

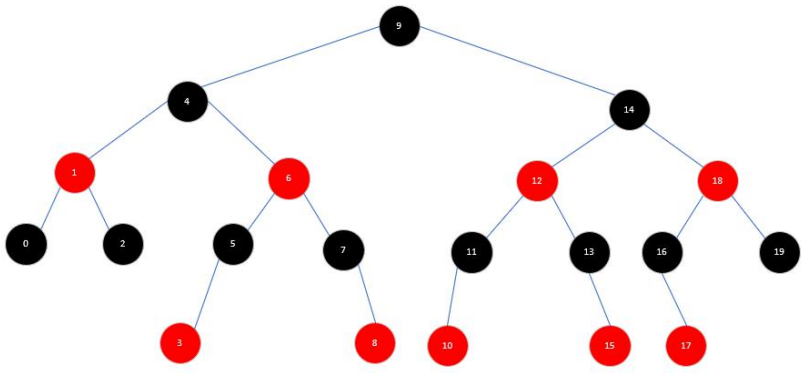
# 四、实验结果

对于以下输入：

```
20
12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17
```

建立好红黑树并采用层次遍历的输出以及该红黑树可视化形式为：

```
9, black
4, black
14, black
1, red
6, red
12, red
18, red
0, black
2, black
5, black
7, black
11, black
13, black
16, black
19, black
nil
nil
nil
3, red
nil
nil
nil
8, red
10, red
nil
nil
nil
15, red
17, red
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
```




# 五、实验心得

在编写本实验源码的过程中，无论是输入输出数据还是插入算法

的实现均比较复杂，于是我采用这样的思路进行编写源码：1.我将源码划分为两大部分：插入算法实现，以及 IO 操作，首先实现插入算法部分，接着再实现 IO 操作部分。2.在编写源码过程中，无论是实现插入操作还是 IO，总是先确定好自己为实现这一部分的功能需要定义的函数及函数的接口，包括传入参数、传出参数以及返回值的类型等，接下来再根据算法的具体过程来细化各个函数的内容。3.在实现插入算法过程中，我是用 insert()函数调用 FixUp()函数，再用 FixUp()函数调用 rotate()函数，按照 insert->FixUp->rotate 这样的顺序来定义函数接口，再按照相反地顺序来实现各个函数的内部细节。按照上述思路我在编写源码的整个过程中思路都很清晰，以后在编写代码量比较大的源码时也会按照这样的思路进行编写。