

目录

Introduction	1.1
分治法	1.2
快速排序	1.2.1
中位数	1.2.2
最大子序和	1.2.3
线性时间排序	1.3
计数排序	1.3.1
二叉树	1.4
二叉查找树	1.4.1
红黑树	1.4.2

Introduction

记录我的算法分析学习历程

分治法

快速排序

问题：

实现对数组`int arr[9]={-2,1,-3,4,-1,2,1,-5,4}`的快速排序，并画出流程图

方法： 分治法

快速排序原理：

- 任找一个元素作为基准，对待排数组进行分组
- 使基准元素左边的数据都比基准元素小，右边的数据都比基准元素大。这样基准元素就放在了正确的位置上。
- 然后对基准元素左边和右边的数据分组进行相同的操作，最后完成数组的排序。

代码如下：

```

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

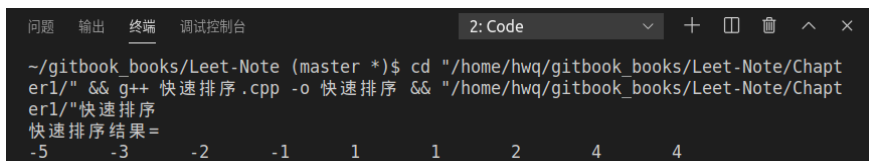
int Partition(int arr[], int low, int high){
    int pivot_key=arr[low]; //临时存储基准值
    while(low<high){
        while (low<high && arr[high]>=pivot_key) --high;
        arr[low]=arr[high];
        while (low<high && arr[low]<=pivot_key) ++low;
        arr[high]=arr[low];
    }
    arr[low]=pivot_key; //把基准值放到最后准确的位置
    return low;
}

void QuickSort(int arr[], int low, int high){
    int pivot;
    if(low<high){
        pivot=Partition(arr, low, high);
        QuickSort(arr, low, pivot-1);
        QuickSort(arr, pivot+1, high);
    }
}

int main(){
    int arr[9]={-2, 1, -3, 4, -1, 2, 1, -5, 4};
    QuickSort(arr, 0, 8);
    cout<<"快速排序结果= "<<endl;
    for (char i = 0; i < 9; i++){
        cout<<arr[i]<<'\\t';
    }
    cout<<endl;
    return 0;
}

```

运行结果：



```

~/gitbook_books/Leet-Note (master *)$ cd "/home/hwq/gitbook_books/Leet-Note/Chapter1/" && g++ 快速排序.cpp -o 快速排序 && "/home/hwq/gitbook_books/Leet-Note/Chapter1/"快速排序
快速排序结果=
-5      -3      -2      -1      1      1      2      4      4

```

快排流程图

元素 第 N 趟	-2	1	-3	4	-1	2	1	-5	4
1	-5	-3	-2	4	-1	2	1	1	4
2	-5	-3	-2	1	-1	2	1	4	4
3	-5	-3	-2	-1	1	2	1	4	4
4	-5	-3	-2	-1	1	1	2	4	4

注：褐色的数字格表示此趟快排的基准值

算法复杂度分析：

时间复杂度： $O(n\lg(n))$

空间复杂度： $O(\lg(n))$

Median of Two Sorted Arrays

问题：

There are two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively. Find the median of the two sorted arrays. The overall run time complexity should be $O(\log(m+n))$.

中位数的概念

- 将一个集合划分为两个长度相等的子集，其中一个子集中的元素总是大于另一个子集中的元素。

方法：分治法

算法分析

- 将有序数组分成两部分，可以得到如下关系式：

```
len(left_part)=len(right_part)
max(left_part)≤min(right_part)
```

left_part	right_part
A[0], A[1], ..., A[i-1]	A[i], A[i+1], ..., A[m-1]
B[0], B[1], ..., B[j-1]	B[j], B[j+1], ..., B[n-1]

- 那么，中位数就是：

$$median = [\max(left_part) + \min(right_part)] / 2$$

代码如下：

```

int findMedianSortedArrays(int A[],int A_len, int B[],int B_len)
{
    int m=A_len,n=B_len;
    int iMin = 0, iMax = m, halfLen = (m + n + 1) / 2;
    while (iMin <= iMax) {
        int i = (iMin + iMax) / 2;
        int j = halfLen - i;
        if (i < iMax && B[j-1] > A[i]){
            iMin = i + 1; // i is too small,需要增大i, 减小j
        }
        else if (i > iMin && A[i-1] > B[j]) {
            iMax = i - 1; // i is too big,需要减小i, 增大j
        }
        else { // i is perfect, i是临界值, 0或者m
            int maxLeft = 0;
            if (i == 0) { maxLeft = B[j-1]; }
            else if (j == 0) { maxLeft = A[i-1]; }
            else { maxLeft = max(A[i-1], B[j-1]); }
            if ( (m + n) % 2 == 1 ) { return maxLeft; }

            int minRight = 0;
            if (i == m) { minRight = B[j]; }
            else if (j == n) { minRight = A[i]; }
            else { minRight = min(B[j], A[i]); }

            return (maxLeft + minRight) / 2;
        }
    }
}

```

运行结果:

数组元素为:array1[3] = {1,2,7}; array2[3] = {3,5,6};

C:\> 选择C:\Windows\system32\cmd.exe

midia_key= 4
请按任意键继续. . .

算法复杂度分析:

- 时间复杂度：查找的区间是[0,m],每次循环之后，查找区间的长度都会降为原先的一半。所以，最多执行 $\lg(m)$ 次。由于 $m \leq n$,所以时间复杂度为 $O(\lg(\min(m, n)))$ 。

最大子序和

问题：

给定一个整数数组nums，找到一个具有最大和的连续子数组(子数组最少包含一个元素)，返回其最大和。

示例：

输入: [-2,1,-3,4,-1,2,1,-5,4]

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大为 6

方法：分治法

算法分析：

1. 把数组分成左右两个子数组，最大子序和只可能出现在
 - 1.左子数组
 - 2.右子数组
 - 3.横跨左右子数组的部分或全部元素
2. 然后对左子数组或右子数组时，进一步拆分，依次循环，直至拆分的子数组中只有一个元素。
 - 拆分序列（直到只剩下一个数的数组）
 - 求左子数组最大值
 - 求右子数组最大值
 - 求横跨左右子数组的最大值
3. 合并，得出以上三个最大值的最大值
4. 当最大子数组有 n 个数字时：
 - 若 $n == 1$ ，返回此元素。
 - $left_sum$ 是左子数组的元素之和最大值
 - $right_sum$ 是右子数组的元素之和最大值
 - $cross_sum$ 是横跨左右子数组元素之和的最大值

代码如下：

```

#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int CrossSum(int nums[],int left, int right, int mid) {
    if (left == right) return nums[left];
    int leftSubSum=0;
    int leftMaxSum=nums[mid]; //横跨左右子数组，则基准元素(左
    for(int i=mid; i>=left; i--) {
        leftSubSum+=nums[i];
        leftMaxSum=leftMaxSum>=leftSubSum?leftMaxSum:leftSubSum;
    }
    int rightSubSum=0;
    int rightMaxSum=nums[mid+1]; //横跨左右子数组，则右边第一
    for(int i=mid+1; i<=right; i++) {
        rightSubSum+=nums[i];
        rightMaxSum=rightMaxSum>=rightSubSum?rightMaxSum:rightSubSum;
    }
    return leftMaxSum+rightMaxSum;
}

int fun(int nums[],int left, int right) {
    if (left == right) return nums[left];

    int mid = (left + right) / 2;
    int leftSum = fun(nums, left, mid);
    int rightSum = fun(nums, mid + 1, right);
    int crossSum = CrossSum(nums, left, right, mid);
    int temp=leftSum>rightSum?leftSum:rightSum;

    return temp>crossSum?temp:crossSum;
}

int MaxSubArray(int nums[],int length) {
    return fun(nums, 0, length-1);
}

int main() {
    int nums[9]={-2,1,-3,4,-1,2,1,-5,4};
    int maxSum=MaxSubArray(nums,9);
    cout<<"maxSum= " <<maxSum<<endl;
}

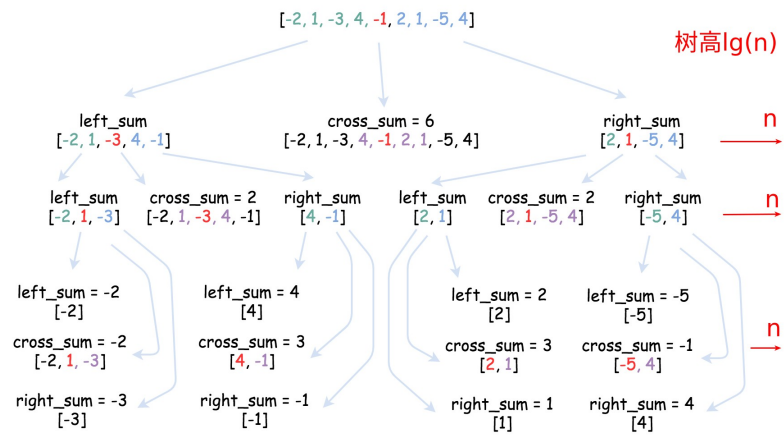
```

运行结果：

```
maxSum= 6
```

算法复杂度分析:

• 递归树法:

时间复杂度: $O(n \lg(n))$ 

线性时间排序方法

计数排序

问题：

实现对数组`int arr[10]={95,94,91,98,99,90,99,93,91,92}`的计数排序，并画出流程图

计数排序原理：

计数排序是由额外空间的辅助和元素本身的值决定的。计数排序过程中不存在元素之间的比较和交换操作，根据元素本身的值，将每个元素出现的次数记录到辅助空间后，通过对辅助空间内数据的计算，即可确定每一个元素最终的位置。

- 算法过程
 1. 根据待排序集合中最大元素与最小元素的差值范围，申请额外辅助空间
 2. 遍历待排序集合，将每一个元素出现的次数记录到元素值对应的辅助空间
 3. 对辅助空间内的数据进行计算，得出每一个元素的正确位置
 4. 将待排序集合的每一个元素移动到计算出的正确位置上，排序完成

代码如下：

```

#include <iostream>
#include <new>
using namespace std;

void CountSort(int *arr,int len,int max, int min)
{
    int *count=new int[max-min+1]; //计数数组
    int *Result=new int[len]; //存放排序后的结果
    int index;
    for (int i =0; i <=max-min; i++){ //初始化
        count[i]=0;
    }
    for(int i=0; i<len;i++){ //计算arr[i]元素出现的个数
        count[arr[i]-min]++;
    }
    for(int i=1; i<=max-min; i++){
        count[i]+=count[i-1];
    }

    for (int i = len-1; i >= 0; i--){
        index=count[arr[i]-min]-1;
        Result[index]=arr[i];
        count[arr[i]-min]--;
    }
    cout<<"计数排序后的结果= "<<endl;
    for(int i=0;i<len;i++){
        cout<<Result[i]<<"\t";
    }

    cout<<endl;
    delete [] count;
    delete [] Result;
}

int main()
{
    int arr[10]={95,94,91,98,99,90,99,93,91,92}; //待排数组
    CountSort(arr,10,99,90); //计数排序
}

```

运行结果：



```

~/gitbook_books/Leet-Note (master *)$ cd "/home/hwq/gitbook_books/Leet-Note/Chapter2/" && g++ 计数排序.cpp -o 计数排序 && "/home/hwq/gitbook_books/Leet-Note/Chapter2/"计数排序
计数排序后的结果=
90  91  91  92  93  94  95  98  99  99

```

计数排序流程图

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
1	arr[10]													4	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9	
	元素	95	94	91	98	99	90	99	93	91	92			元素				92							
2	count[10]													5	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9	
	个数	1	2	1	1	1	1	0	0	1	2			元素			91	92							
3	count[10]													6	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9	
	次序	1	3	4	5	6	7	7	7	8	10			元素			91	92	93						
														7	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素			91	92	93					99	
														8	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90		91	92	93					99	
														9	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90		91	92	93				99	99	
														10	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90		91	92	93			98	99	99	
														11	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90	91	91	92	93			98	99	99	
														12	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90	91	91	92	93	94		98	99	99	
														13	Result[10]										
														序号	0	1	2	3	4	5	6	7	8	9	
														元素	90	91	91	92	93	94	95	98	99	99	

注: Result 下的深灰色格子代表此次排好序的元素

算法复杂度分析:

时间复杂度: 因为 $k = n$, 所以为 $O(n)$

空间复杂度: 申请了额外辅助空间, 且 $k = n$, 所以为 $O(n)$

计数排序与快速排序的区别:

1. 计数排序属于线性时间排序, 用非比较的操作确定排序顺序; 而快速排序是基于元素之间的比较, 属于比较排序
2. 任意一个比较排序算法, 在最坏情况下, 都需要做 $\Omega(n \lg(n))$ 次的比较; 而计数排序的运行时间为 $\Theta(k+n)$
3. 计数排序是一种稳定的排序算法; 而快速排序不是
4. 计数排序算法适合待排序元素在一定范围内, 数值比较集中; 而快速排序没有这种要求

二叉树

二叉查找树 BST (Binary Search Tree)

二叉查找树又称二叉搜索树、二叉排序树,特点如下:

1. 左子树上所有结点值均小于根结点
2. 右子树上所有结点值均大于根结点
3. 结点的左右子树本身又是一颗二叉查找树
4. 二叉查找树中序遍历得到结果是递增排序的结点序列。

算法分析

BST的结点结构:

```
//BST结点结构
template<typename T>
class BSTNode{
public:
    T _key; //关键字
    BSTNode *_lchild; //左孩子
    BSTNode *_rchild; //右孩子
    BSTNode *_parent; //父结点

    //构造函数
    BSTNode(T key ,BSTNode *_lchild,BSTNode *_rchild,BSTNode *_parent):
        _key(key),_lchild(_lchild),_rchild(_rchild),_parent(_parent) {}
};
```

一、 判断是否为二叉查找树

根据第4条性质,可以利用中序遍历得出的结果序列为**小->大**,来判断是否为二叉查找树

```

template <typename T>
bool BSTree<T>::checkBST(BSTNode<T>* &tree) const
{
    static BSTNode<T> *prev=NULL;
    if(tree != NULL)
    {
        if(!checkBST(tree->_lchild))
            return false;
        if(prev != NULL && tree->_key < prev->_key)
            return false;
        prev = tree;
        if(!checkBST(tree->_rchild))
            return false;
    }
    return true;
}

```

可以看出，采用递归的方式，当前的结点值小于前一个结点的值，就满足性质。否则，判断失败。

二、插入操作

首先创建一个新结点，用于存储关键值。

```

template <typename T>
void BSTree<T>::insert(T key)
{
    //创建一个新的节点，使用构造函数初始化
    BSTNode<T>* z= new BSTNode<T>(key, NULL, NULL, NULL);
    if(!z) //如果创建失败则返回
        return ;
    //调用内部函数进行插入
    insert(_Root, z);
}

```

接着，判断插入值与根结点的大小关系，插入左子树还是右子树。并循环向下查找。

```

//插入操作
//内部使用函数
template<typename T>
void BSTree<T> ::insert(BSTNode<T>* &tree, BSTNode<T>* z)
{
    BSTNode<T>* parent = NULL;
    BSTNode<T>* temp = tree;

    //寻找插入点
    while(temp!=NULL)
    {
        parent= temp;
        if(z->_key > temp->_key)
            temp= temp->_rchild;
        else
            temp=temp->_lchild;
    }
    z->_parent = parent;
    if(parent==NULL) //如果树本来就是空树，则直接把z结点插入根结点
        tree = z;
    else if(z->_key > parent->_key) //如果z的值大于其双亲，则z为
        parent->_rchild = z;
    else
        parent->_lchild = z; //否则为其双亲的左孩子结点
}

```

三、删除操作

```

template<typename T>
void BSTree<T>::remove(T key)
{
    BSTNode<T> *z, *node;
    if ((z = search(_Root, key)) != NULL)
        if ((node = remove(_Root, z)) != NULL)
            delete node;
}

```

四、查找操作

外部接口search函数

```

template <typename T>
BSTNode<T> * BSTree<T>::search(T key)
{
    return search(_Root, key);
}

```

内部调用search函数

```
//非递归实现
//内部使用函数
template <typename T>
BSTNode<T>* BSTree<T>::search(BSTNode<T>* &tree, T key) const
{
    BSTNode<T>* temp = tree;
    while(temp != NULL)
    {
        if(temp->_key == key)//查找成功
            return temp;
        else if(temp->_key > key)//转向左子树，继续查找
            temp = temp->_lchild;
        else
            temp = temp->_rchild;//转向右子树，继续查找
    }
    return NULL;//查找失败
}
```

五、遍历操作

1. 前序遍历：

外部preOrder接口

```
template<typename T>
void BSTree<T>::preOrder()
{
    preOrder(_Root);
}
```

内部preOrder接口

```
template<typename T>
void BSTree<T>::preOrder(BSTNode<T>*&tree) const
{
    if(tree)
    {
        cout<<tree->_key<<" ";
        preOrder(tree->_lchild);
        preOrder(tree->_rchild);
    }
}
```

2. 中序遍历：

外部inOrder接口

```
template<typename T>
void BSTree<T>::inOrder()
{
    inOrder(_Root);
}
```

内部inOrder接口

```
template <typename T>
void BSTree<T>::inOrder(BSTNode<T>*&tree) const
{
    if(tree)
    {
        inOrder(tree->_lchild);
        cout<<tree->_key<<" ";
        inOrder(tree->_rchild);
    }
}
```

3. 后序遍历:

外部postOrder接口

```
template<typename T>
void BSTree<T>::postOrder()
{
    postOrder(_Root);
}
```

内部postOrder接口

```
template <typename T>
void BSTree<T>::postOrder(BSTNode<T>*&tree) const
{
    if(tree)
    {
        postOrder(tree->_lchild);
        postOrder(tree->_rchild);
        cout<<tree->_key<<" ";
    }
}
```

六、实验结果

快速排序

如图：

```
~/gitbook_books/Leet-Note (master) $ cd ~/home/hwq/gitbook_books/Leet-Note/Chapter3/ && g++ BSTree.cpp -o BSTree && ./BSTree
请输入二叉树结点以构造二叉查找树：
2 5 3 4 1 6
打印二叉查找树
节点2有左孩子为1
节点2有右孩子为5
节点1无左孩子
节点1无右孩子
节点5有左孩子为3
节点5有右孩子为6
节点3无左孩子
节点3有右孩子为4
节点4无左孩子
节点4无右孩子
节点6无左孩子
节点6无右孩子
验证二叉查找树
原二叉查找树
请输入要查找的数：
3
3查找成功
4
4查找成功
7
7查找失败
已经将7插入二叉查找树，现在二叉查找树中序遍历为：
1 2 3 4 5 6 7
请输入要删除的结点：
4
删除后的二叉排序树：
1 2 3 5 6 7 节点2有左孩子为1
节点2有右孩子为5
节点1无左孩子
节点1无右孩子
节点5有左孩子为3
节点5有右孩子为6
节点3无左孩子
节点3有右孩子
节点6无左孩子
节点6有右孩子为7
节点7无左孩子
节点7无右孩子
```

红黑树

红黑树中每个结点包含五个域:color,key,left,right 和 p。如果某结点没有一个子结点或父结点,则该域指向 NIL。

一棵二叉树如果满足下面的红黑性质,则为一棵红黑树:

1. 每个结点或是红的,或是黑的。
2. 根结点是黑的。
3. 每个叶结点 (NIL) 是黑的。
4. 如果一个结点是红的,则它的两个儿子都是黑的。
5. 对每个结点,从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

算法分析

RBTree的结点结构:

```
class Node {
public:
    int key; //关键字
    Node * p; //父结点
    Node * left; //左子结点
    Node * right; //右子结点
    enum c {RED, BLACK};
    c color;
    Node(int k);
};
```

一、左旋操作

```

void RBTree::LeftRotate(Node * n)
{
    Node * y=n->right;
    n->right=y->left;
    if (y->left!=NIL)
    {
        y->left->p=n;
    }
    y->p=n->p;
    if(n->p==NIL) root=y;
    else if (n==n->p->left)
    {
        n->p->left=y;
    }
    else n->p->right=y;
    y->left=n;
    n->p=y;
}

```

二、右旋操作

```

void RBTree::RightRotate(Node * n)
{
    Node * y=n->left;
    n->left=y->right;
    if (y->right!=NIL)
    {
        y->right->p=n;
    }
    y->p=n->p;
    if(n->p==NIL) root=y;
    else if (n==n->p->right)
    {
        n->p->right=y;
    }
    else n->p->left=y;
    y->right=n;
    n->p=y;
}

```

三、查询操作

根据给定的关键字值，查找出某个结点


```
Node * RBTree::Search(int k)
{
    Node * cur=root;
    while(cur->key!=k)
    {
        if(cur->key<k&&cur->right!=NIL)
            cur=cur->right;
        else if (cur->key>k&&cur->left!=NIL)
            cur=cur->left;
        else
            return NIL;
    }
    return cur;
}
```

四、插入操作

外部接口Insert函数

```

Node * RBTree::Insert(int k)
{
    Node * cur=root;
    Node * prev=root;
    while(cur!= NIL)
    {
        prev=cur;
        if(k<cur->key)
            cur=cur->left;
        else if (k>cur->key)
            cur=cur->right;
        else
        {
            std::cerr<<std::endl<<k<<" already in RBTree.\n"
            return NIL;
        }
    }
    if(k<prev->key)
    {
        prev->left=new Node(k);
        prev->left->p=prev;
        prev->left->left=prev->left->right=NIL;
        prev->left->color=Node::RED;
        InsertFixup(prev->left);
        return prev->left;
    }
    else if(k>prev->key)
    {
        prev->right=new Node(k);
        prev->right->p=prev;
        prev->right->left=prev->right->right=NIL;
        prev->right->color=Node::RED;
        InsertFixup(prev->right);
        return prev->right;
    }
}

```

内部调用Insert函数，维持红黑树性质

```

void RBTree::InsertFixup(Node * n)
{
    while(n->p->color==Node::RED)
    {
        if(n->p==n->p->p->left)
        {
            Node * uncle=n->p->p->right;
            if(uncle->color==Node::RED)
            {
                n->p->color=uncle->color=Node::BLACK;
                n->p->p->color=Node::RED;
                n=n->p->p;
            }
            else if(uncle->color==Node::BLACK&& n==n->p->right)
            {
                n=n->p;
                LeftRotate(n);
            }
            else if(uncle->color==Node::BLACK&& n==n->p->left)
            {
                n->p->color=Node::BLACK;
                n->p->p->color=Node::RED;
                RightRotate(n->p->p);
            }
        }
        else if (n->p==n->p->p->right)
        {
            Node * uncle=n->p->p->left;
            if(uncle->color==Node::RED)
            {
                n->p->color=uncle->color=Node::BLACK;
                n->p->p->color=Node::RED;
                n=n->p->p;
            }
            else if(uncle->color==Node::BLACK&& n==n->p->left)
            {
                n=n->p;
                RightRotate(n);
            }
            else if(uncle->color==Node::BLACK&& n==n->p->right)
            {
                n->p->color=Node::BLACK;
                n->p->p->color=Node::RED;
                LeftRotate(n->p->p);
            }
        }
    }
    root->color=Node::BLACK;
}

```

五、删除操作

外部delete接口

```
bool RBTree::Delete(int k)
{
    Node * x;
    Node * z=Search(k);
    Node * y=z;
    Node::c y_original_color=y->color; // of toReplace
    if(z->left==NIL)
    {
        x=z->right;
        TransPlant(z, z->right);
    }
    else if(z->right==NIL)
    {
        x=z->left;
        TransPlant(z, z->left);
    }
    else
    {
        y=Minimum(z->right);
        y_original_color=y->color;
        x=y->right;
        if(y->p==z) x->p=y;
        else
        {
            TransPlant(y, y->right);
            y->right=z->right;
            y->right->p=y;
        }
        TransPlant(z, y);
        y->left=z->left;
        y->left->p=y;
        y->color=z->color;
    }

    if(y_original_color==Node::BLACK)
        DeleteFixup(x);
}
```

内部delete接口，维持红黑树性质

```

void RBTree::DeleteFixup(Node *n)
{
    Node *brother;
    while (n->color == Node::BLACK && n != root) {
        if (n == n->p->left) {
            brother = n->p->right;
            if (brother->color == Node::RED) {
                n->p->color = Node::RED;
                brother->color = Node::BLACK;
                LeftRotate(n->p);
                brother = n->p->right;
            }
            if (brother->left->color == Node::BLACK && brother->right->color == Node::RED) {
                brother->color = Node::RED;
                n = n->p;
            } else if (brother->right->color == Node::BLACK && brother->left->color == Node::RED) {
                brother->color = Node::RED;
                brother->left->color = Node::BLACK;
                RightRotate(brother);
                brother = n->p->right;
            } else {
                brother->color = n->p->color;
                n->p->color = Node::BLACK;
                brother->right->color = Node::BLACK;
                LeftRotate(n->p);
                n = root;
            }
        } else {
            brother = n->p->left;
            if (brother->color == Node::RED) {
                n->p->color = Node::RED;
                brother->color = Node::BLACK;
                LeftRotate(n->p);
                brother = n->p->left;
            }
            if (brother->right->color == Node::BLACK && brother->left->color == Node::RED) {
                brother->color = Node::RED;
                n = n->p;
            } else if (brother->left->color == Node::BLACK && brother->right->color == Node::RED) {
                brother->color = Node::RED;
                brother->right->color = Node::BLACK;
                LeftRotate(brother);
                brother = n->p->left;
            } else {
                brother->color = n->p->color;
            }
        }
    }
}

```

```

        n->p->color=Node::BLACK;
        brother->left->color=Node::BLACK;
        RightRotate(n->p);
        n=root;
    }

}

}

n->color = Node::BLACK;
}

```

六、实验结果

如图：

```

~/gitbook_books/Leet-Note (master) $ cd ~/home/hwq/gitbook_books/Leet-Note/Chapter3/RBTree/ && g++ rbtrees.cpp -o rbtrees && ./rbtrees && cat ~/home/hwq/gitbook_books/Leet-Note/Chapter3/RBTree/
e/rbtrees
num1[]={41,30,31,12,19,8};
RED 8 parent 12 left -1 right -1
BLACK 12 parent 19 left 8 right -1
RED 19 parent 30 left 12 right 31
BLACK 31 parent 19 left -1 right -1
BLACK 30 parent -1 left 19 right 41
BLACK 41 parent 30 left -1 right -1

num2[]={1,5,6,7,9,8,10,11,12,13,14,15};
BLACK 1 parent 5 left -1 right -1
BLACK 5 parent 7 left 1 right 6
BLACK 6 parent 9 left -1 right -1
BLACK 7 parent -1 left 5 right 11
RED 8 parent 9 left -1 right -1
RED 9 parent 11 left 8 right 10
BLACK 10 parent 9 left -1 right -1
BLACK 11 parent 7 left 9 right 13
BLACK 12 parent 13 left -1 right -1
RED 13 parent 11 left 12 right 14
BLACK 14 parent 13 left -1 right 15
RED 15 parent 14 left -1 right -1

delete之后:
RED 1 parent 6 left -1 right -1
BLACK 6 parent 7 left 1 right -1
BLACK 7 parent 11 left 6 right 10
RED 8 parent 10 left -1 right -1
BLACK 10 parent 7 left 8 right -1
BLACK 11 parent -1 left 7 right 13
BLACK 12 parent 13 left -1 right -1
BLACK 13 parent 11 left 12 right 15
BLACK 15 parent 13 left -1 right -1

```