高级树结构剖析

第三次讨论课

经典算法设计技术研讨

目录

CONTENTS

红黑树

1 概述

2 数据结构的定义

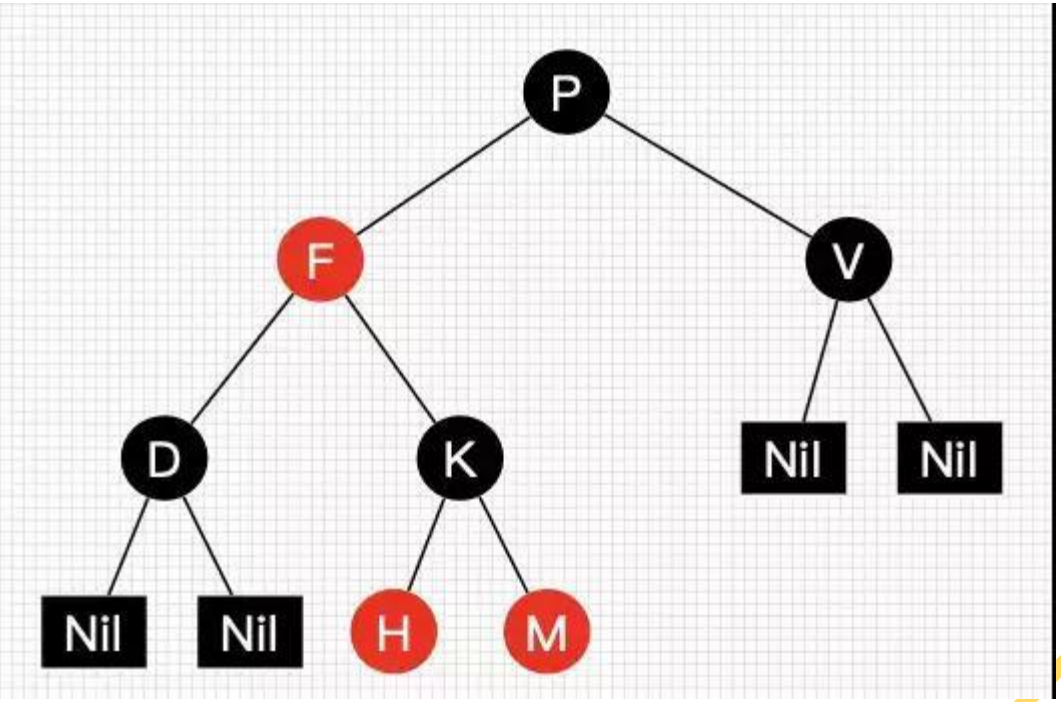
3 关键基本操作的具体实现

4 应用

红黑树 概述

红黑树是一种自平衡的二叉查找树，是一种高效的查找树。它是由 Rudolf Bayer 于1972年发明，在当时被称为对称二叉 B 树。(symmetric binary B-trees)。
后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的红黑树。

红黑树具有良好的效率，它可在 O(logN) 时间内完成查找、增加、删除等操作。因此，红黑树在业界应用很广泛，比如 Java 中的 TreeMap，JDK 1.8 中的 HashMap、C++ STL 中的 map 均是基于红黑树结构实现的。

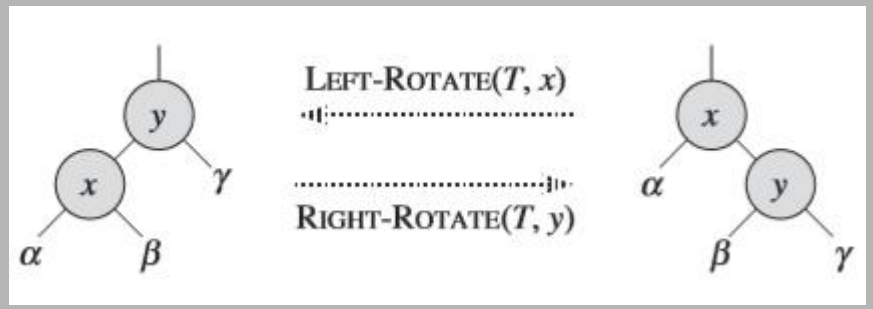


红黑树 概述

红黑树的查找 = 二叉树的查找+查找修复

红黑树的删除 = 二叉树的删除+查找修复

左旋与右旋



红黑树 概述

性质：

1.结点是红色或黑色。

2.根结点是黑色。

3.每个叶子结点都是黑色的空结点（NIL结点）。

4.每个红色结点的两个子结点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色结点)

5.从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。

推论：红黑树从根到叶子结点的最长路径不超过最短路径的2倍。

操作：

一、插入：

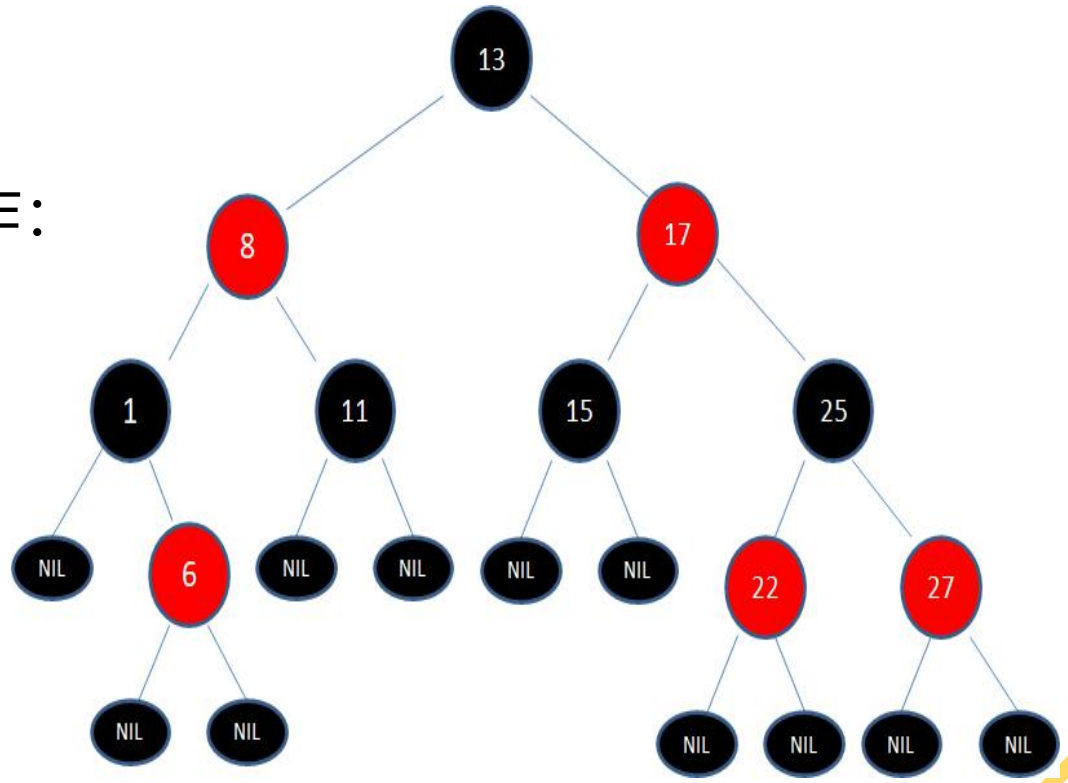
二、调整操作：

1、变色。

2、左旋转。

3、右旋转。

三、删除：



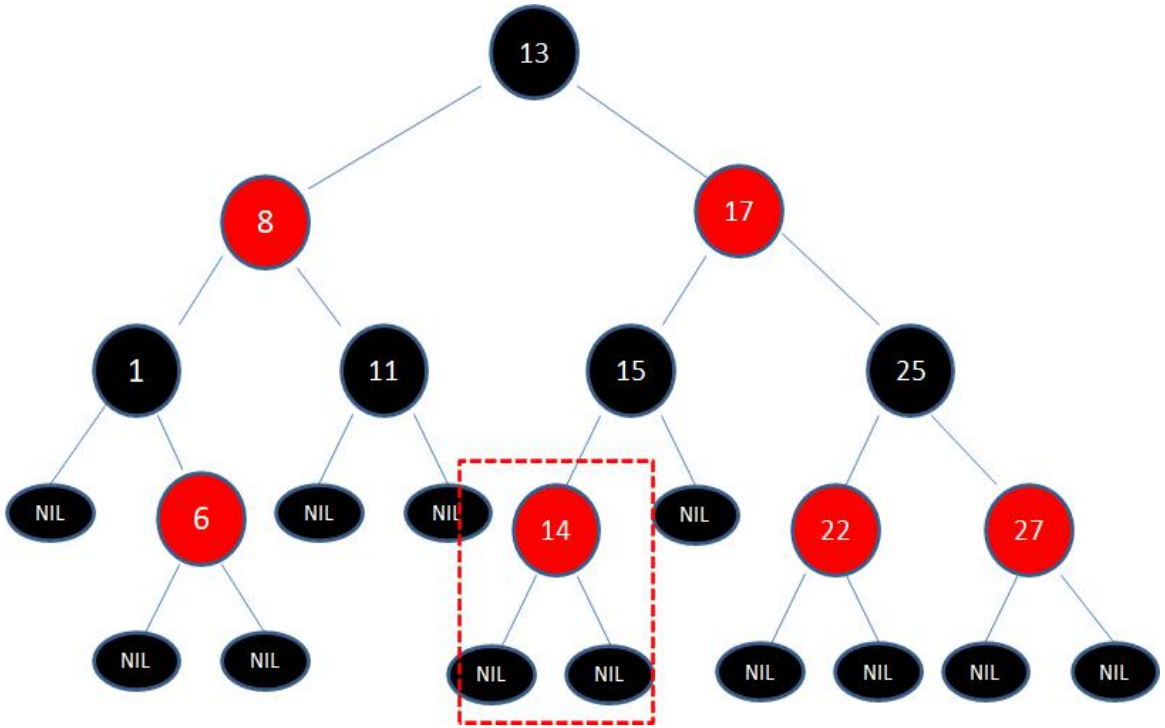
红黑树的结构实现

```
const static bool RED = 0;
const static bool BLACK = 1;
struct Node
{
    int val;
    bool color;             //颜色
    Node *left, *right, *p; //左，右孩子，父节点
    Node(const int &v,const bool &c=RED,Node *l=nullptr, Node
*r=nullptr, Node *_p = nullptr)
    :val(v),color(c),left(l),right(r),p(_p){}
};
struct RBTree
{
    Node *root;              //树根
    Node *nil;               //外部节点, color: 黑色
    RBTree()
    {
        nil = new Node(-1, BLACK, nil, nil, nil);
        root = nil;     }
};
```

红黑树 关键基本操作的实现

插入

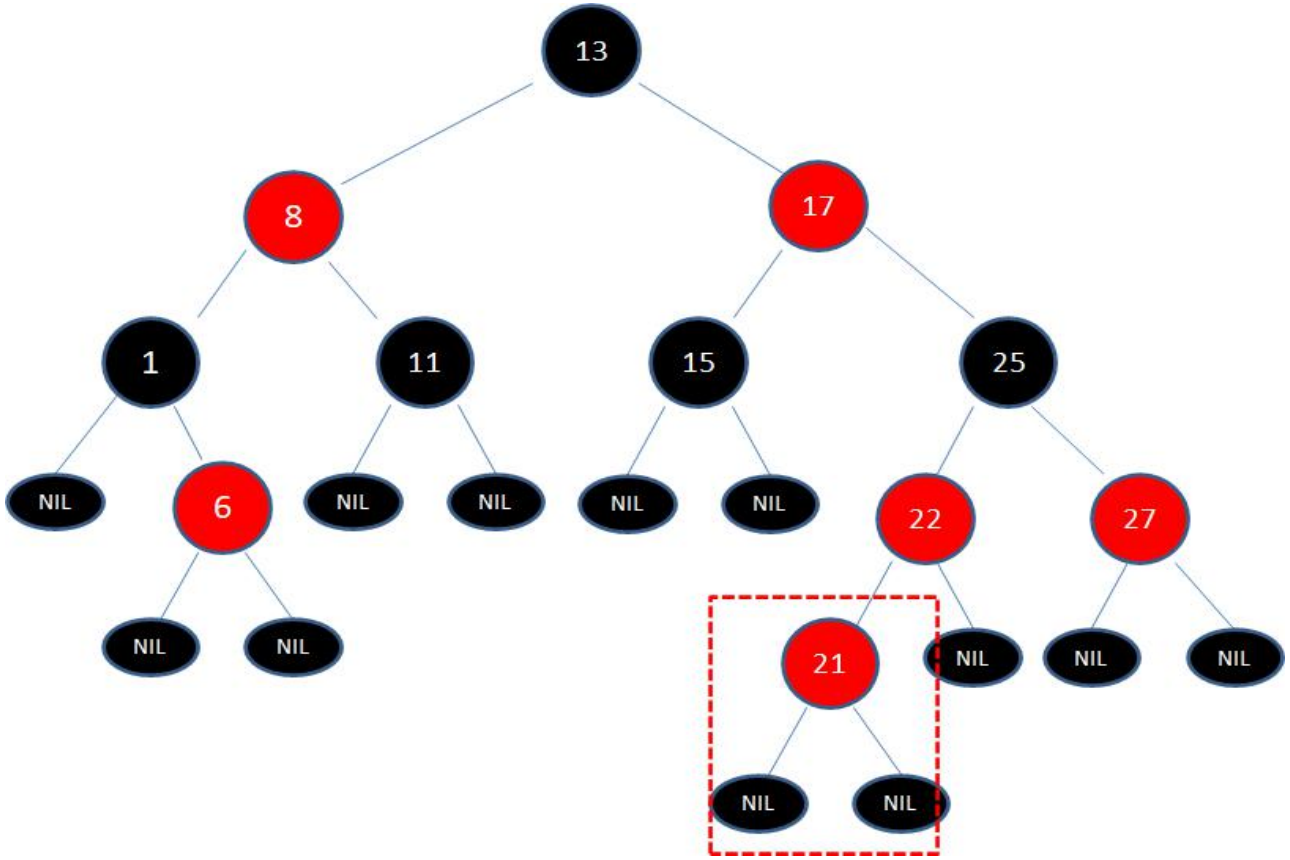
一、在原树中插入14，因不破坏规则，故无需调整。



插入

二、向原红黑树插入值为21的新结点：

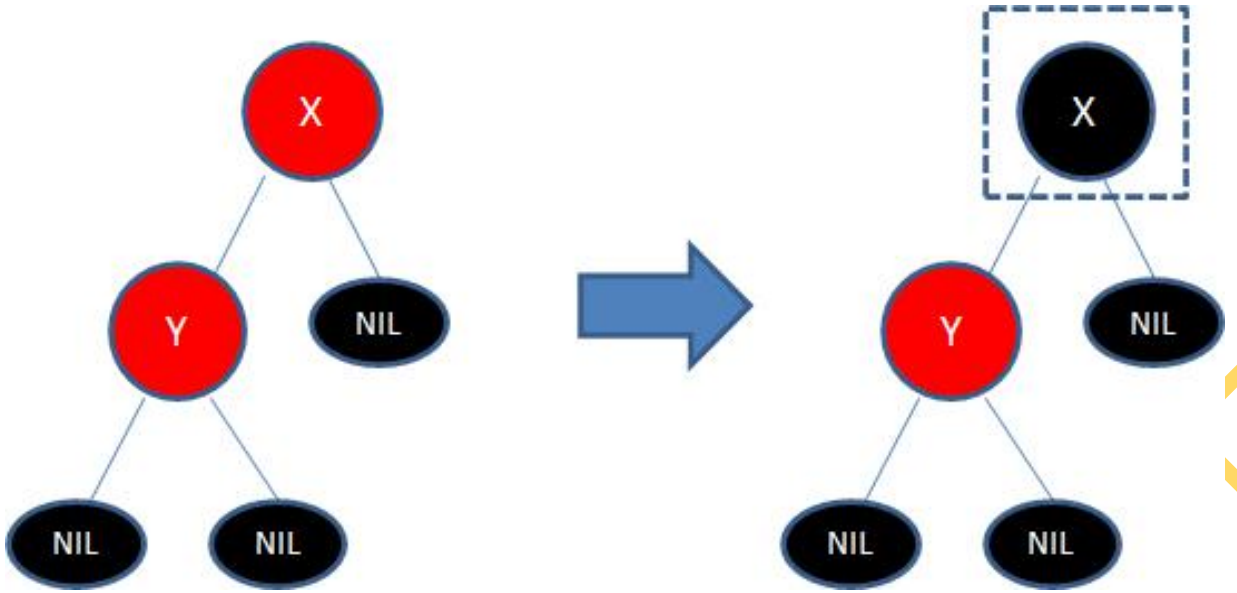
（由于父结点22是红色结点，因此这种情况打破了红黑树的规则4（每个红色结点的两个子结点都是黑色），必须进行调整，使之重新符合红黑树的规则。）



变色

三、调整操作：①变色

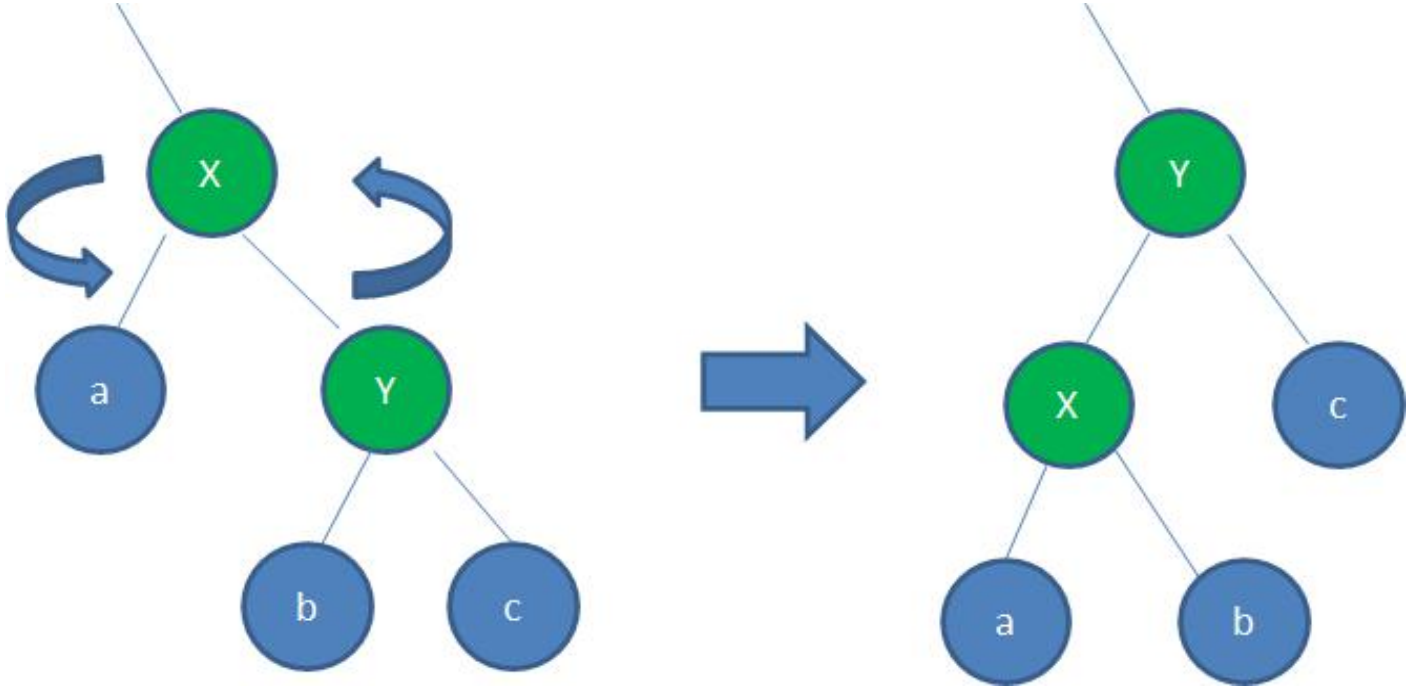
为了重新符合红黑树的规则，尝试把红色结点变为黑色，或者把黑色结点变为红色。新插入的结点Y是红色结点，它的父亲结点X也是红色的，不符合规则4，因此我们可以把结点X从红色变成黑色，并对因此二改变其他节点做出调整。



左旋转

三、调整操作：②左旋转

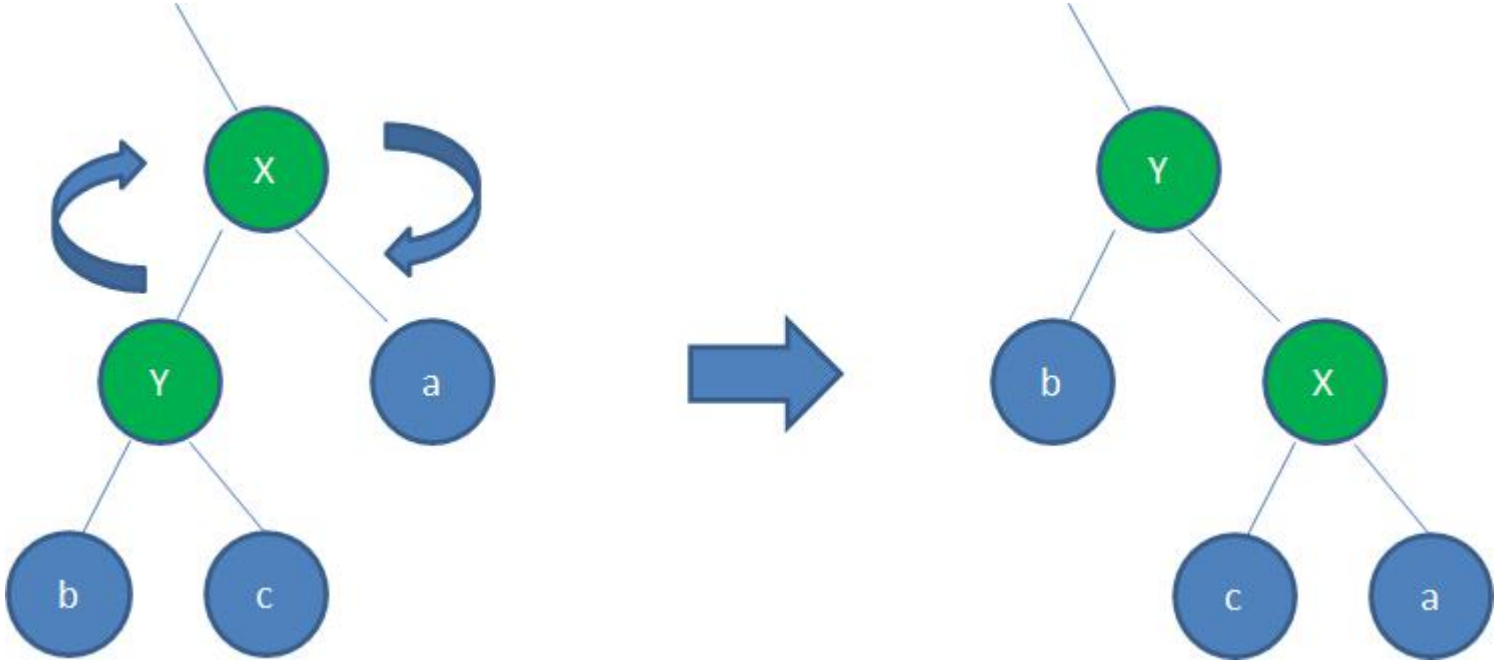
逆时针旋转红黑树的两个结点，使得父结点被自己的右孩子取代，而自己成为自己的左孩子。



右旋转

三、调整操作：②右旋转

顺时针旋转红黑树的两个结点，使得父结点被自己的左孩子取代，而自己成为自己的右孩子。



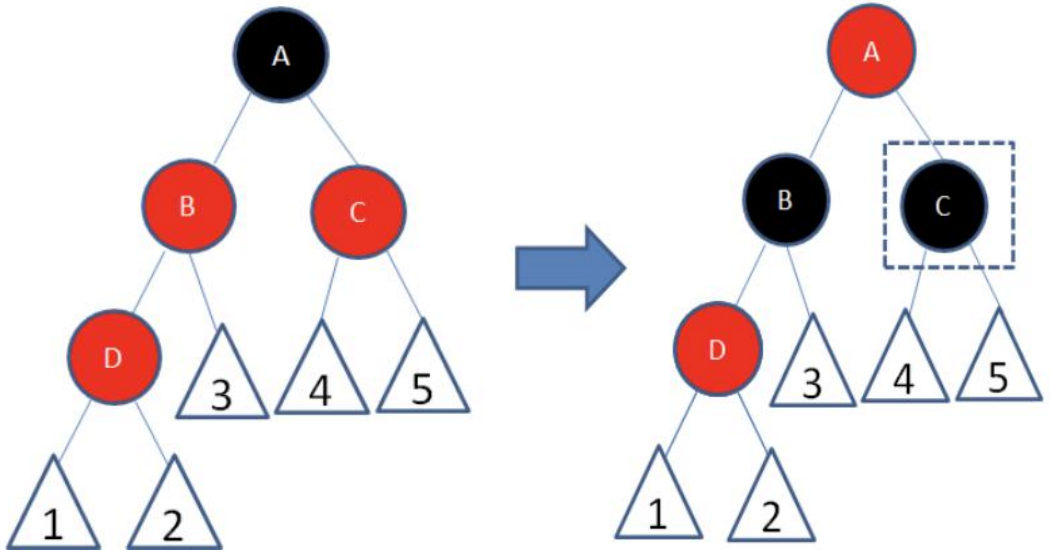
红黑树 关键基本操作的实现

插入

插入五种情况：情况①：新节点位于树根，直接让新节点变为黑色，不需要调整。

情况②：新节点的父节点为黑色，不需要调整。

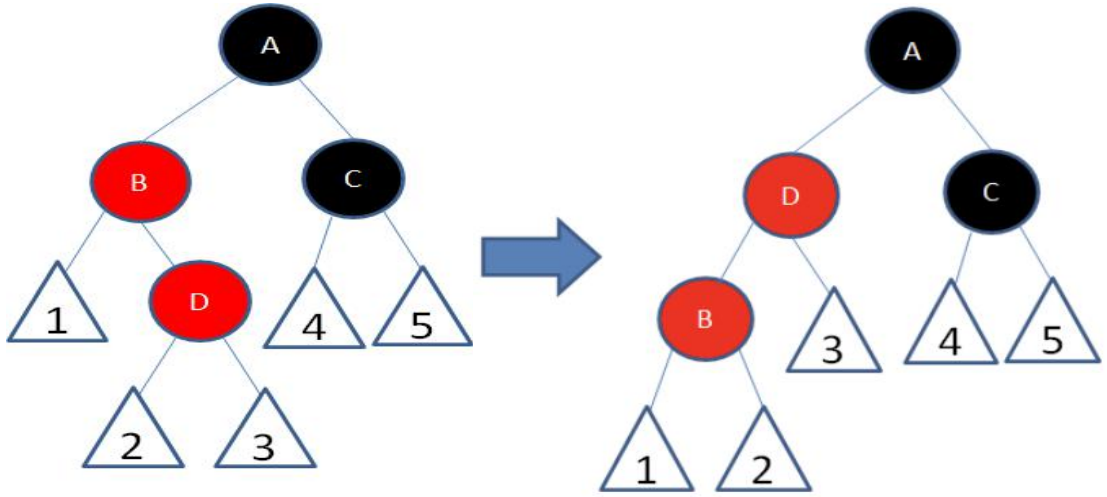
情况③：新节点的父节点与叔节点都是红色，违反规则，需要先让父节点变为黑色，祖父节点变为红色，叔节点变为黑色。



红黑树 关键基本操作的实现

插入

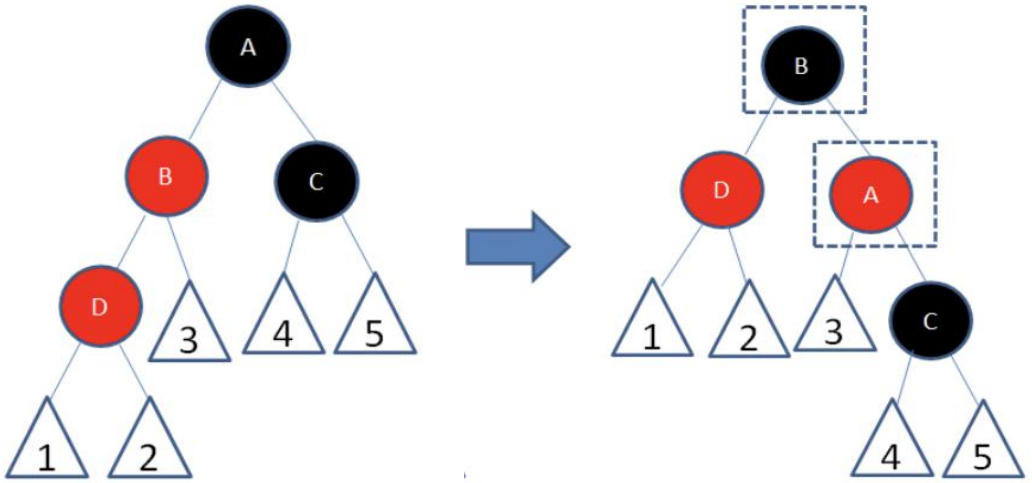
情况④：新节点为右孩子且父节点为红色，叔节点为黑色，以父节点为轴，逆时针旋转一次。使新节点变为父节点，父节点变为左孩子。并进入情况⑤。



红黑树 关键基本操作的实现

插入

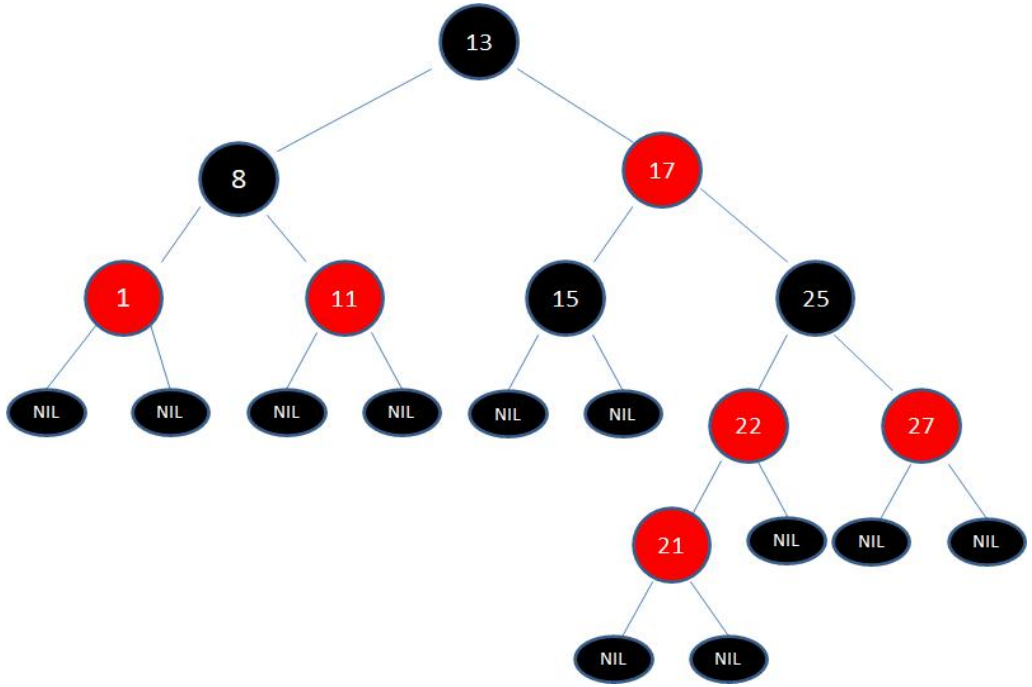
情况⑤：新节点的父节点为红，叔节点为黑，以祖父节点为轴，做一次右旋转，使祖父节点成为父节点左孩子，父节点成为祖父节点，并让父节点变为黑色，祖父节点变为红色。

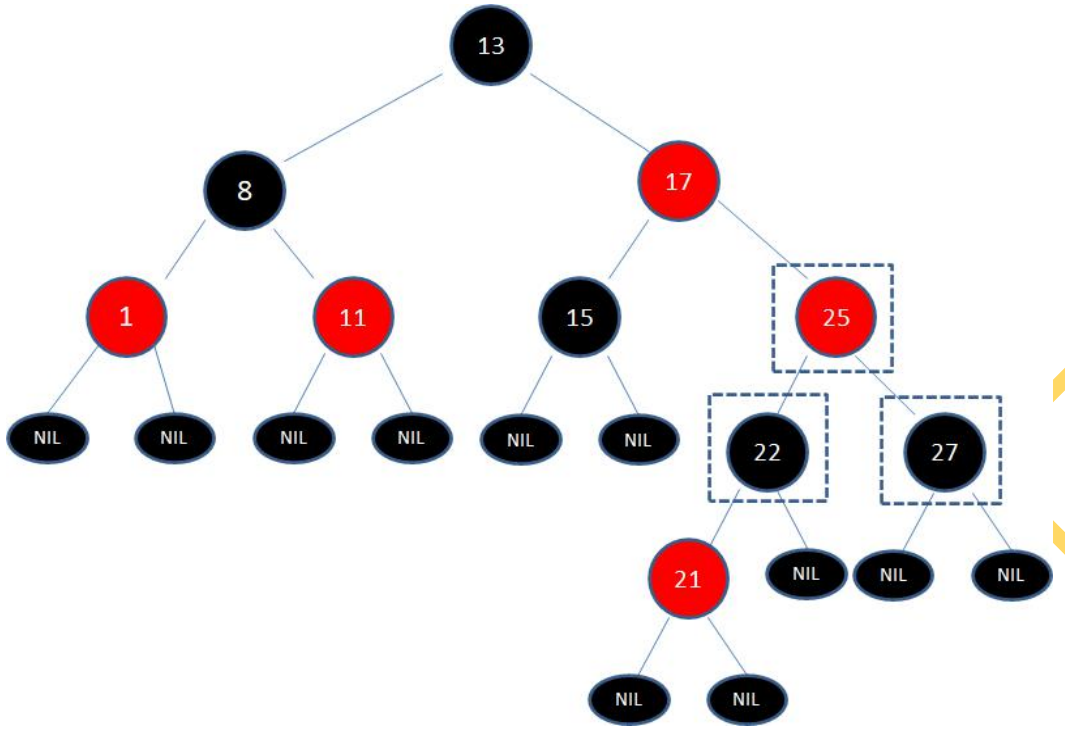


插入

四、插入实例演示：

插入21后，将其父节点22,27改为黑色，祖父节点25改为红色。

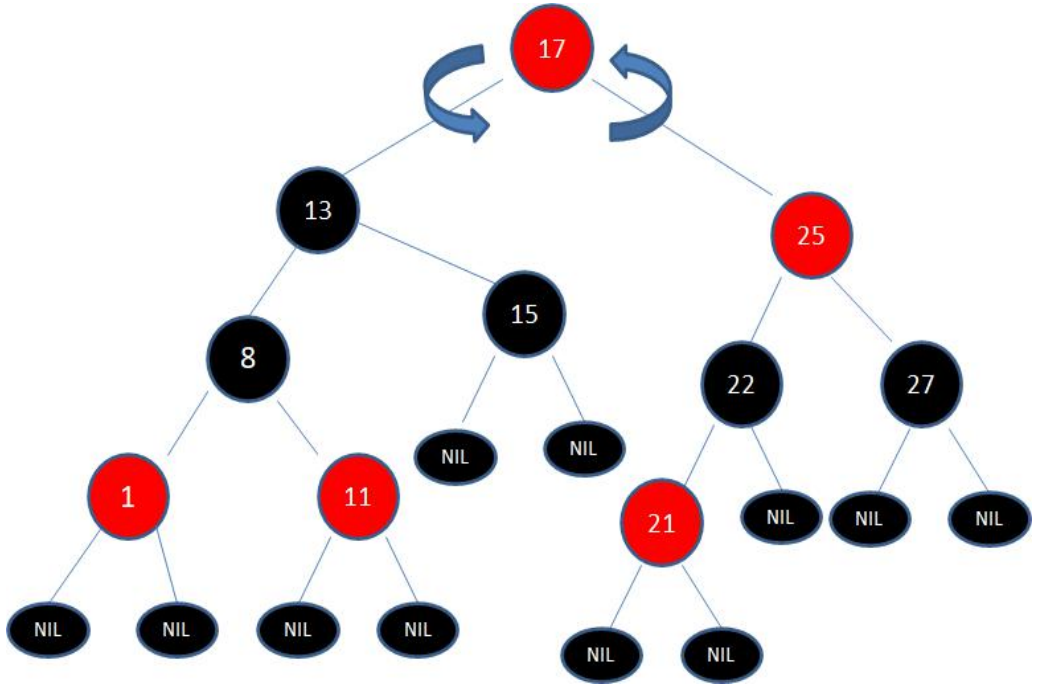


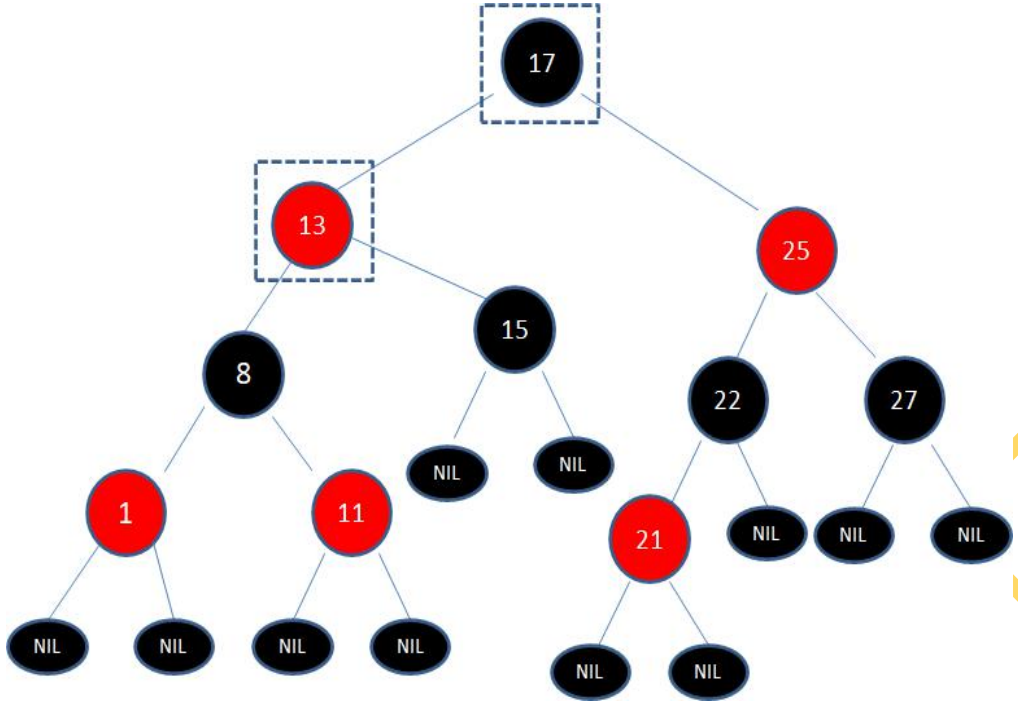


插入

四、插入实例演示：

以根结点13为轴进行左旋转，使得结点17成为了新的根结点，让结点17变为黑色，结点13变为红色。





删除

五、删除基本操作：

步骤一：如果待删除结点有两个非空的孩子结点，转化成待删除结点只有一个孩子（或没有孩子）的情况。

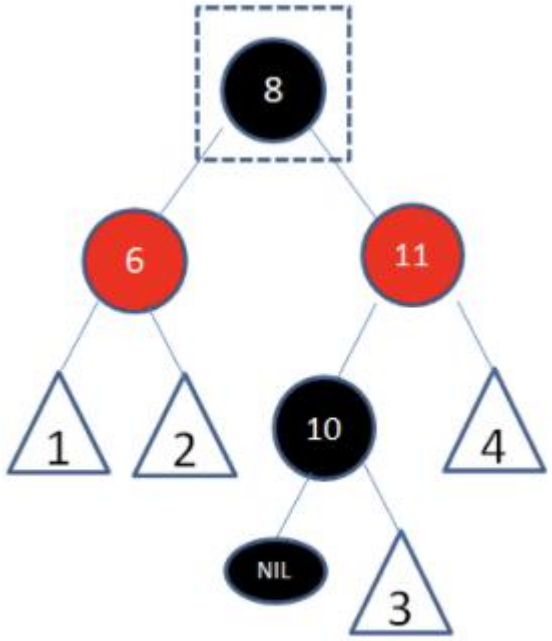
步骤二：根据待删除结点和其唯一子结点的颜色，分情况处理。

步骤三：遇到双黑结点，在子结点顶替父结点之后，分成6种子情况处理。

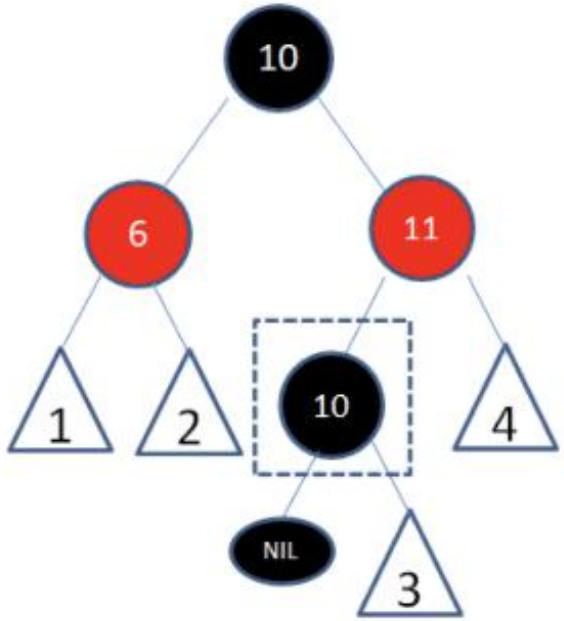
红黑树 关键基本操作的实现

删除

步骤一：如果待删除结点有两个非空的孩子结点，转化成待删除结点只有一个孩子（或没有孩子）的情况。



选择仅大于8的结点10复制到8的位置，结点颜色变成待删除结点的颜色，并按照情况进行删除。



红黑树 关键基本操作的实现

删除

步骤二：根据待删除结点和其唯一子结点的颜色，分情况处理。

情况①：自身是红色，子结点是黑色，删除此节点即可。

情况②：自身是黑色，子结点是红色，删除此节点，并将子节点变为黑色。

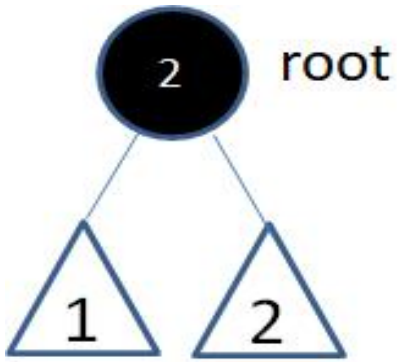
情况③：自身是黑色，子结点也是黑色，或者子结点是空叶子结点，此时先删除此节点，然后进入步骤三。

红黑树 关键基本操作的实现

删除

步骤三：遇到双黑结点，在子结点顶替父结点之后，分成6种子情况处理。

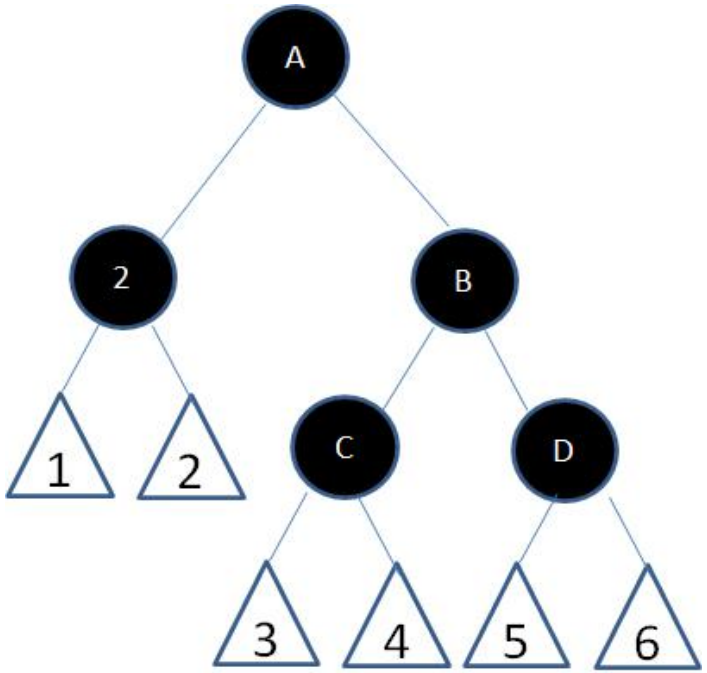
子情况a：此结点是红黑树的根结点。此时所有路径都减少了一个黑色结点，并未打破规则，不需要调整。



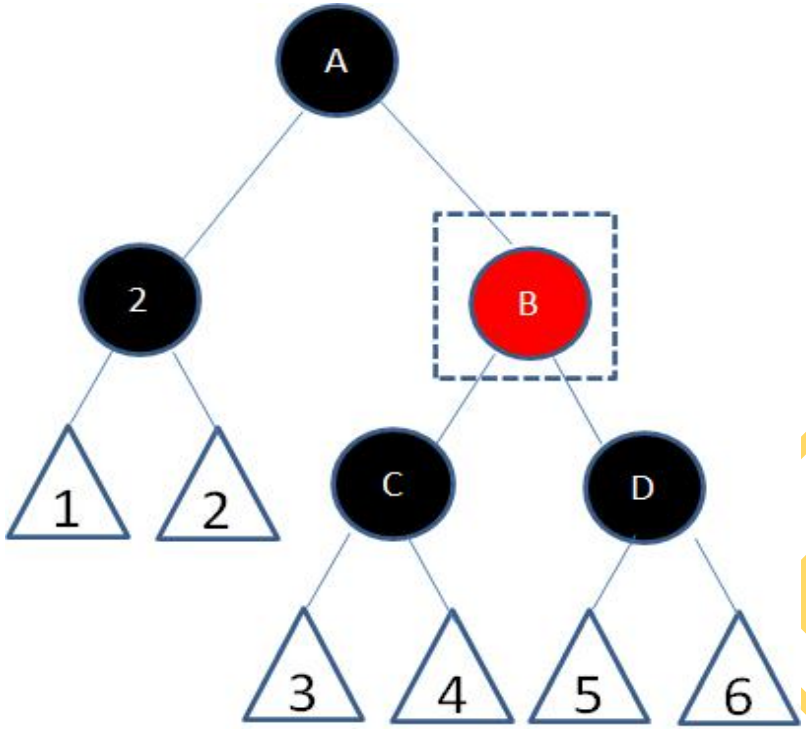
红黑树 关键基本操作的实现

删除

子情况b：此结点的父亲、兄弟、侄子结点都是黑色，将兄弟节点改为红色。



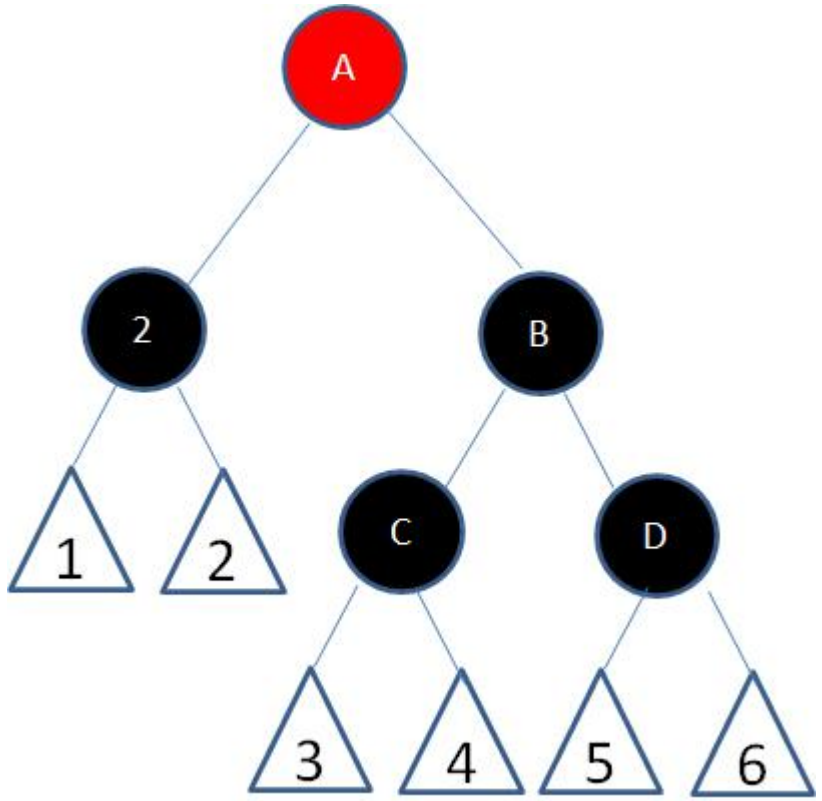
直接把结点2的兄弟结点B改为红色。



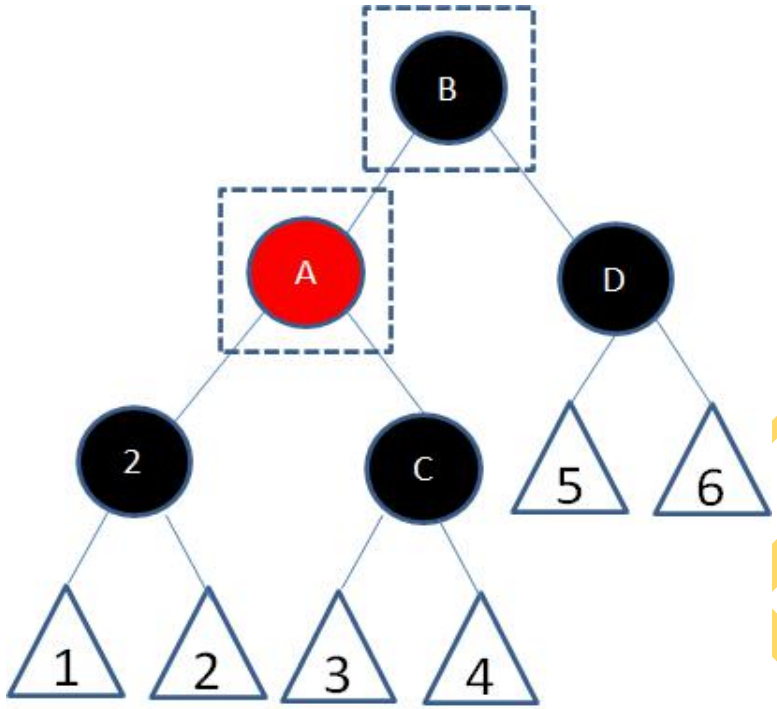
红黑树 关键基本操作的实现

删除

子情况c：此结点的父结点是红色，兄弟和侄子结点是黑色。



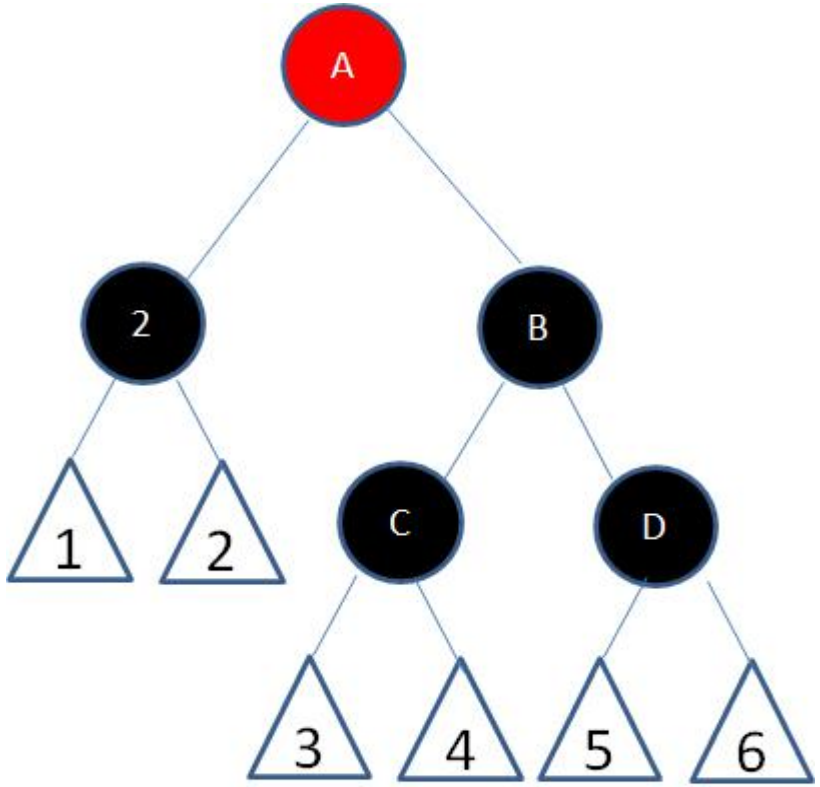
首先以结点2的父结点A为轴，进行左旋，然后结点A变成红色、结点B变成黑色，转换成子情况d、e、f中的任意一种，在子情况d、e、f当中进一步解决。



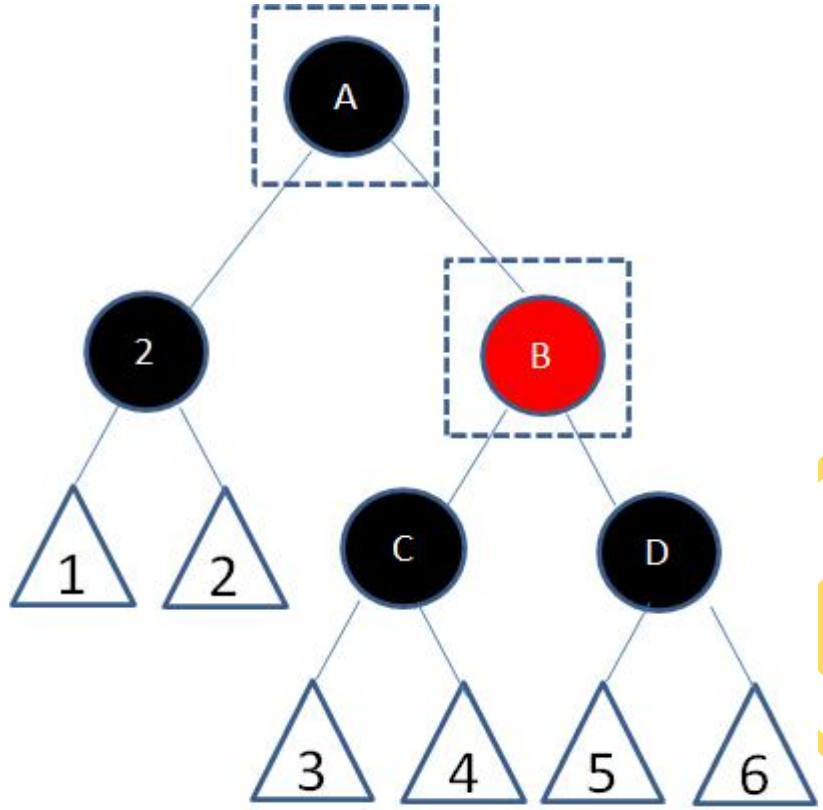
红黑树 关键基本操作的实现

删除

子情况d：此结点的父亲、兄弟、侄子结点都是黑色，将兄弟节点改为红色。



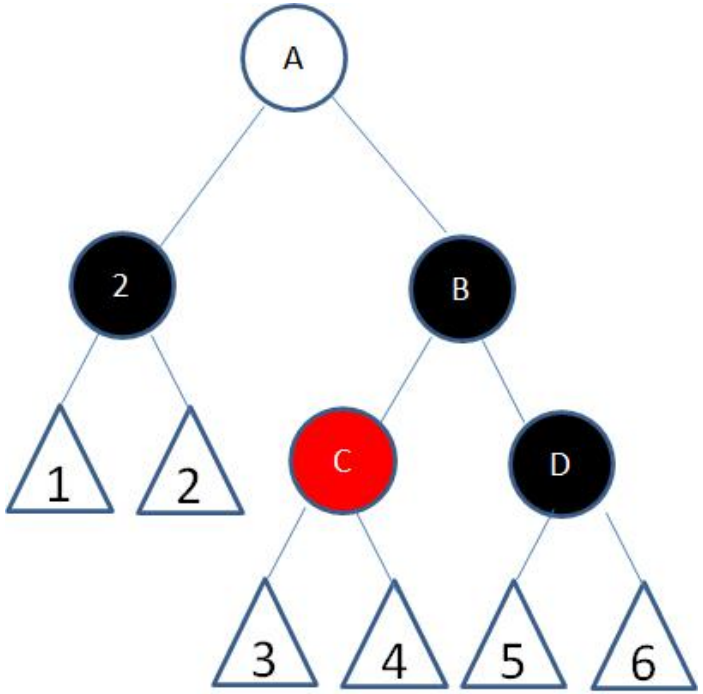
直接让结点2的父结点A变成黑色，兄弟结点B变成红色：



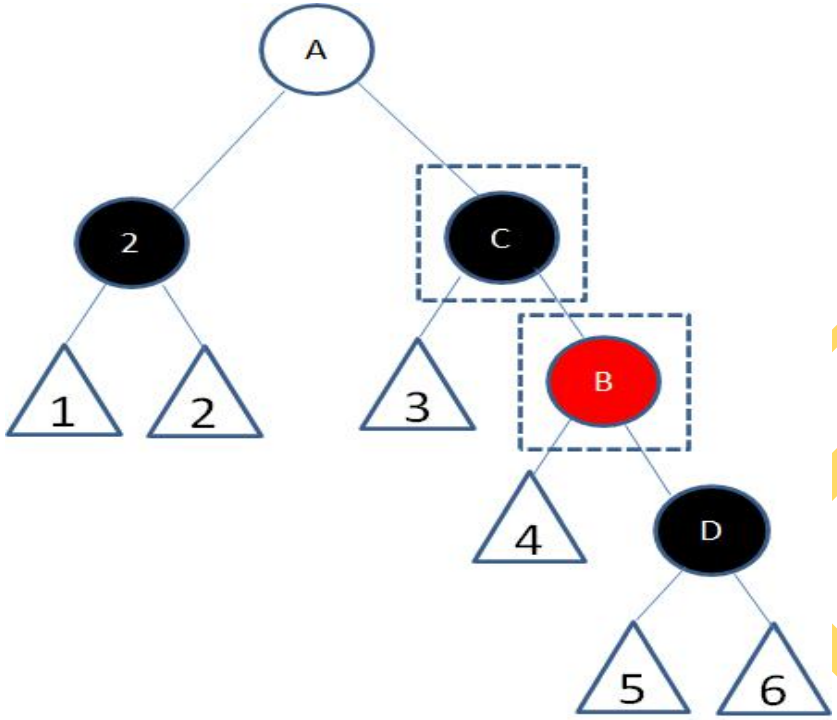
红黑树 关键基本操作的实现

删除

子情况e：此结点的父结点随意，兄弟结点是黑色右孩子，左侄子结点是红色，右侄子结点是黑色。



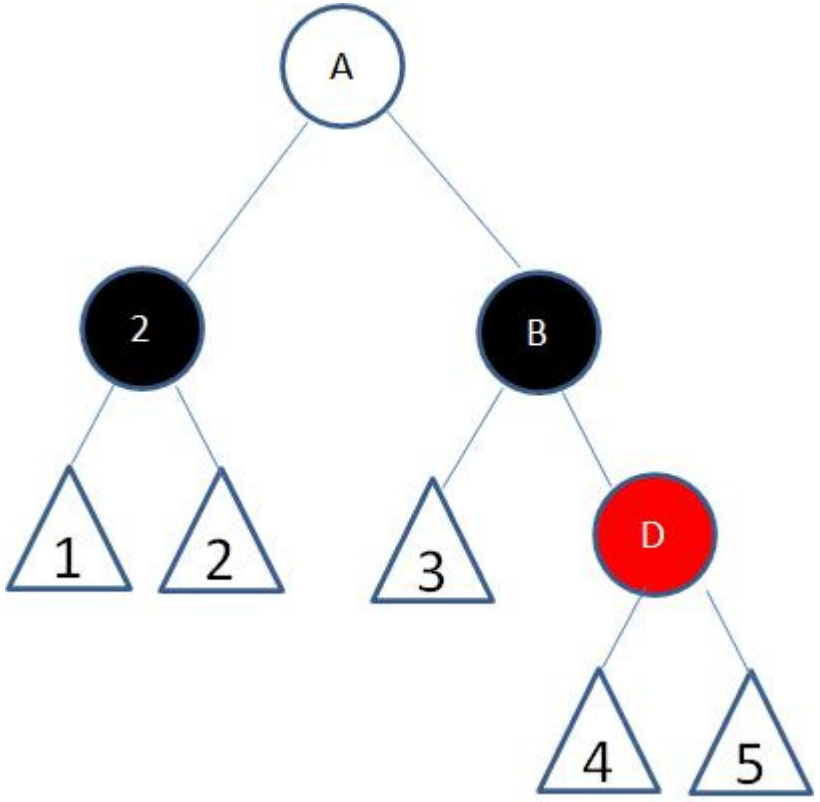
结点2的父结点随意，兄弟结点B是黑色右孩子，左侄子结点是红色，右侄子结点是黑色。
首先以结点2的兄弟结点B为轴进行右旋，
接下来结点B变为红色，结点C变为黑色。



红黑树 关键基本操作的实现

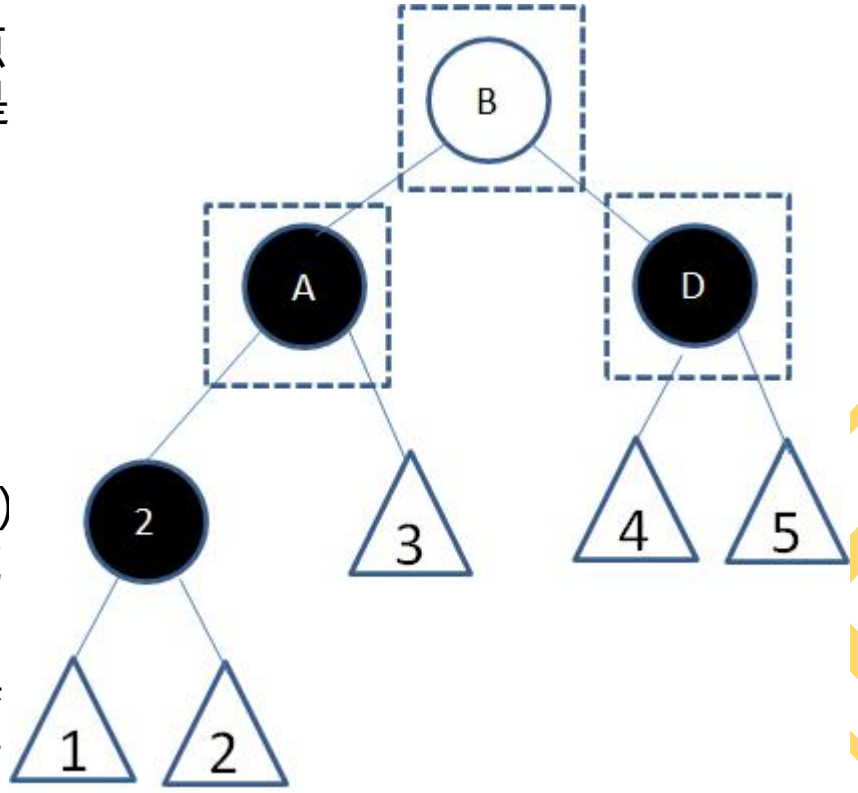
删除

子情况f：此结点的父结点随意，兄弟结点是黑色右孩子，右侄子结点是红色：

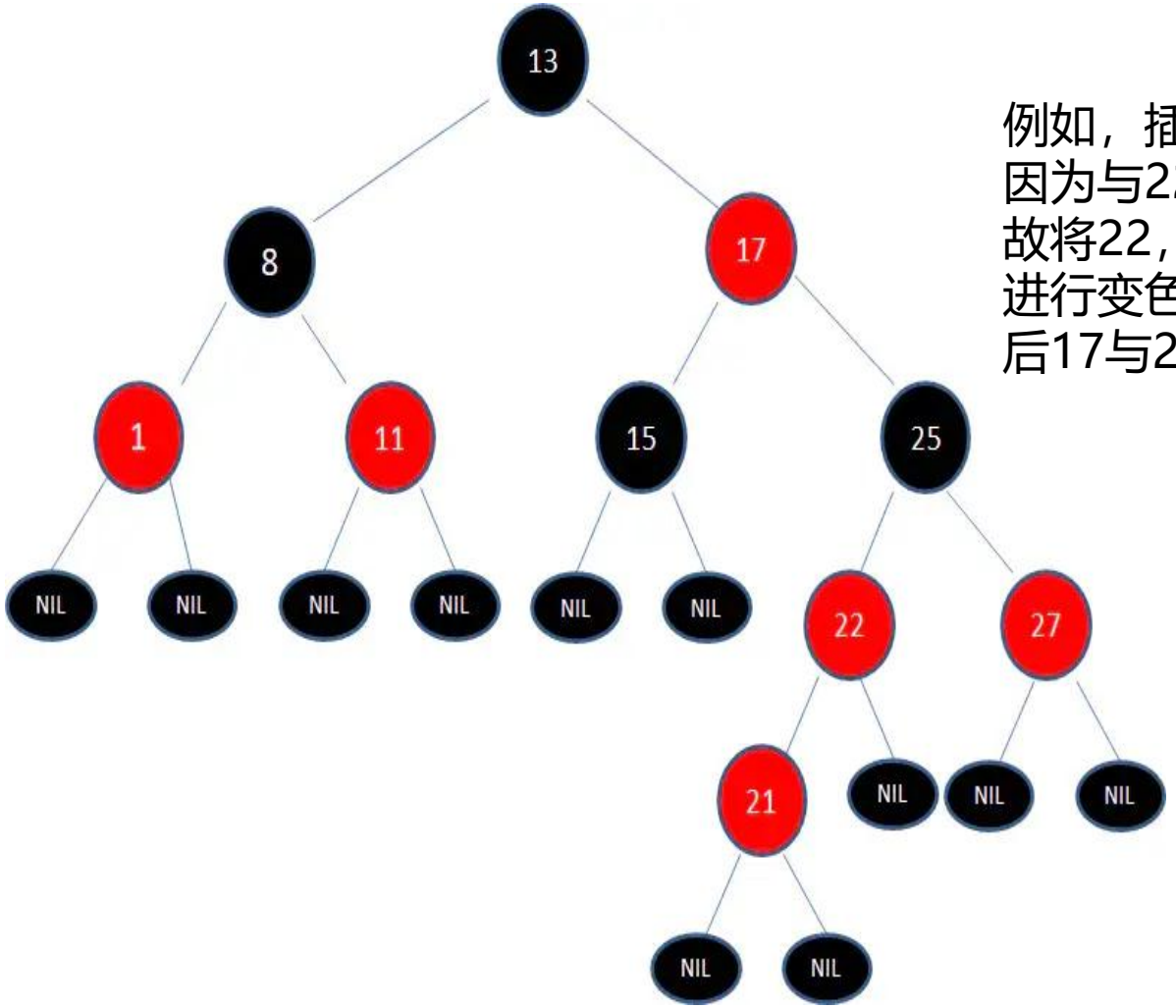


结点2的父结点随意，兄弟结点B是黑色右孩子，右侄子结点是红色：
首先以结点2的父结点A为轴左旋。
接下来让结点A和结点B的颜色交换，并且结点D变为黑色。

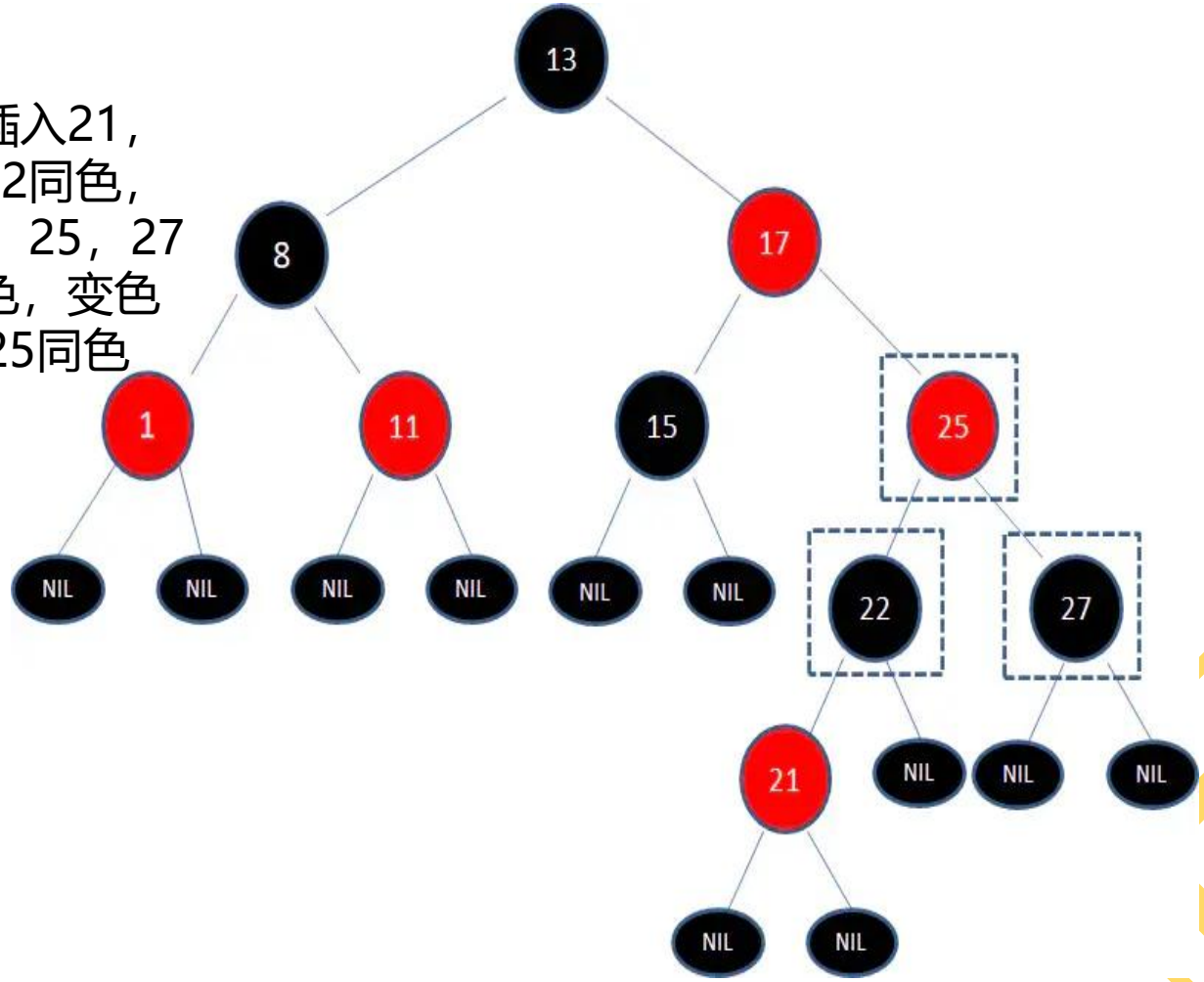
经过结点2的路径由（随意+黑）变成了（随意+黑+黑），补充了一个黑色结点；
经过结点D的路径由（随意+黑+红）变成了（随意+黑），黑色结点并没有减少。
重新符合了红黑树的规则。



六、删除实例演示：

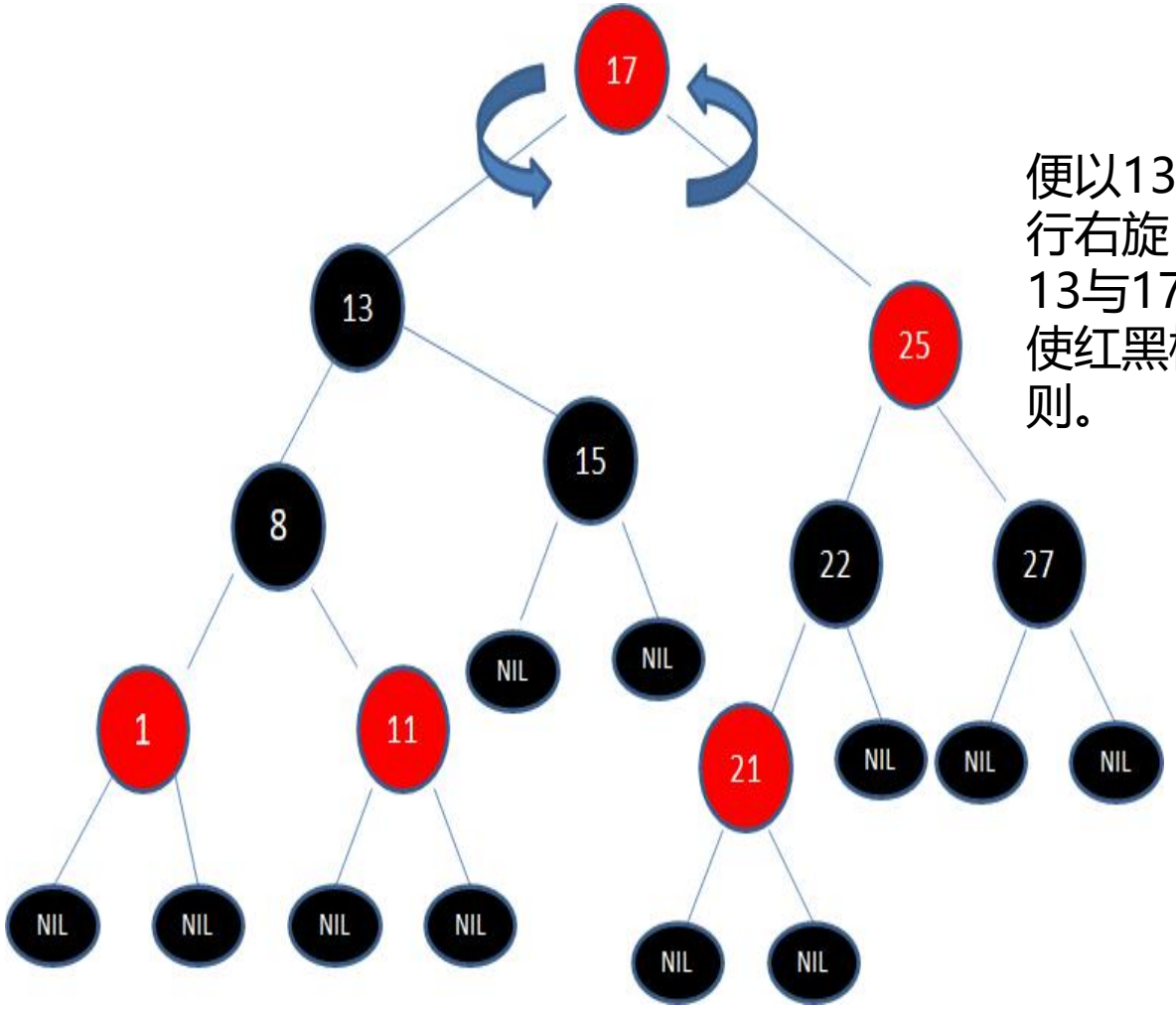


例如，插入21，
因为与22同色，
故将22，25，27
进行变色，变色
后17与25同色

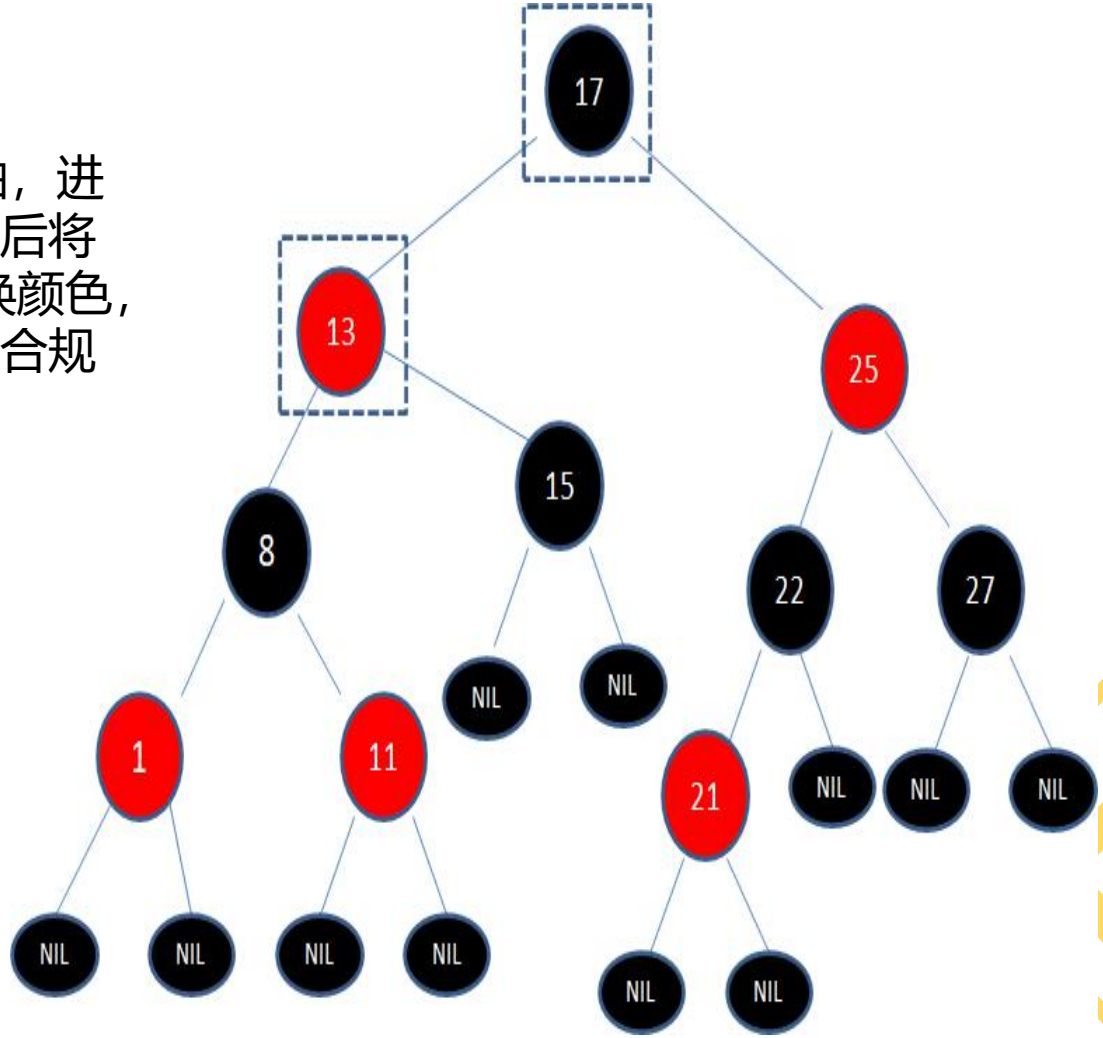


红黑树 关键基本操作的实现

六、删除实例演示：



便以13为轴，进行右旋，最后将13与17互换颜色，使红黑树符合规则。



红黑树 关键基本操作的实现

删除

七、基本代码

插入

```
void insert(const int &v)
{
    Node* z = insert_bst(nil, root, v);
    while(z->p->color==RED)
    {
        if(z->p->p->left == z->p)
        {
            if(z->p->p->right->color == RED)
            {
                z->p->color = BLACK;
                z->p->p->color = RED;
                z->p->p->right->color = BLACK;
                z = z->p->p;
            }
            else
            {
                if(z->p->right == z)
                {
                    z = z->p;
                    left_rotate(z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                right_rotate(z->p->p);
            }
        }
        else
        {
            if(z->p->p->left->color == RED)
            {
                z->p->color = BLACK;
                z->p->p->color = RED;
                z->p->p->left->color = BLACK;
                z = z->p->p;
            }
            else
            {
                if(z->p->left == z)
                {
                    z = z->p;
                    right_rotate(z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                left_rotate(z->p->p);

            }
        }
    }
    root->color = BLACK;
}
```

红黑树 关键基本操作的实现

删除

七、基本代码

```
void left_rotate(Node* x)    //左旋
{
    Node *y = x->right;
    x->right = y->left;
    if(y->left != nil)
    {
        y->left->p = x;
    }
    y->p = x->p;
    if(x->p == nil)
        root = y;
    else if(x->p->left == x)
        x->p->left = y;
    else
        x->p->right = y;
    x->p = y;
    y->left = x;
}
```

```
void right_rotate(Node* x)   //右旋
{
    Node *y = x->left;
    x->left = y->right;
    if(y->right != nil)
        y->right->p = x;
    y->p = x->p;
    if(x->p == nil)
        root = y;
    else if(x->p->left == x)
        x->p->left = y;
    else
        x->p->right = y;
    x->p = y;
    y->right = x;
}
```

```
void rb_delete(Node *z) //删除
{    Node *y = z;
    bool delcol = y->color;
    Node *x = z;
    if(z->left == nil)
    {
        x = z->right;
        rb_transplant(z, z->right);
    }
    else if(z->right == nil)
    {
        x = z->left;
        rb_transplant(z, z->left);
    }
    else
    {
        y = find_min(z->right);
        delcol = y->color;
        x = y->right;
        if(y->p == z)
        {
            x->p = y;
        }
        else
        {
            rb_transplant(y, y->right);
            y->right = z->right;
            y->right->p = y;
        }
        rb_transplant(z, y);
        y->left = z->left;
        y->left->p = y;
        y->color = z->color;
    }
    if(delcol == BLACK)
        rb_delete_fixup(x);}
```

题目描述（二叉搜索树）

（未找到基于红黑树解决的题目，故找到此类似题目）

题目描述

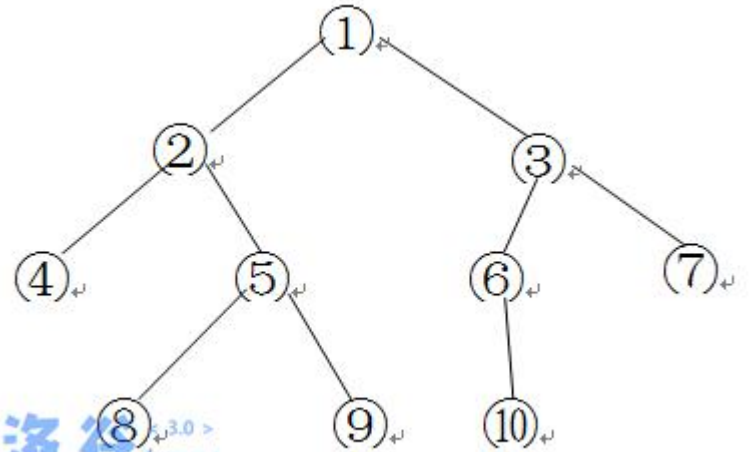
如下图所示的一棵二叉树的深度、宽度及结点间距离分别为：

深度：4 宽度：4（同一层最多结点个数）

结点间距离： ⑧→⑥为8 (3×2+2=8)

⑥→⑦为3 （1×2+1=3）

注：结点间距离的定义：由结点向根方向（上行方向）时的边数×2，与由根向叶结点方向（下行方向）时的边数之和。



输入格式

输入文件第一行为一个整数n(1≤n≤100)，表示二叉树结点个数。接下来的n-1行，表示从结点x到结点y（约定根结点为1），最后一行两个整数u、v，表示求从结点u到结点v的距离。

输出格式

三个数，每个数占一行，依次表示给定二叉树的深度、宽度及结点u到结点v间距离。

输入输出样例

输入 #1

复制

```
10
1 2
1 3
2 4
2 5
3 6
3 7
5 8
5 9
6 10
8 6
```

输出 #1

复

```
4
4
8
```

解答

```cpp
#include<bits/stdc++.h>
using namespace std;
struct node{
    int father;  //
    int left;    //左儿子
    int right;   //右儿子
    int deep;    //深度
    int data;    //记录节点走过没
}a[10001];
int sum[101];
int lca(int x,int y){       //最最重要！！！求最近公共祖先
    a[x].data=1;            //把x的节点记录已走过
    while(a[x].father!=0){ //遍历至根节点
        x=a[x].father;      //更新遍历爸爸
        a[x].data=1;        //记录已走过
    }
    while(a[y].data!=1){    //遍历至x节点已走过的节点，找到最近公共祖先
        y=a[y].father;
    }
    return y;
}
```

```cpp
int main(){
    int n,x,y,s,t,maxx=1;
    cin>>n;
    a[1].deep=1;     //根节点的深度为1
    a[1].father=0;  //根节点没有爸爸
    for(int i=1;i<n;i++){
        cin>>x>>y;
        if(a[x].left==0)     //这个节点是否有左儿子
            a[x].left=y;     //变成左儿子
        else                 //变成右儿子
            a[x].right=y;
        a[y].father=x;
        a[y].deep=a[x].deep+1;    //更新深度
        if(a[y].deep>maxx)        //求出最大深度，第一个问题完成
            maxx=a[y].deep;
    }
    cin>>s>>t;
    int f=lca(s,t),num=0,num1=0;
    while(s!=f){                  //记录s到最近公共祖先有多少条边
        s=a[s].father;
        num++;
    }
    num*=2;   //结点间距离的定义：由结点向根方向（上行方向）时的边数× 2，
    while(t!=f){                  //记录t到最近公共祖先有多少条边
        t=a[t].father;
        num1++;
    }
    for(int i=1;i<=n;i++)         //把每一个深度有多少个节点记录
        sum[a[i].deep]++;
    sort(sum+1,sum+1+100);
    cout<<maxx<<endl<<sum[100]<<endl<<num+num1; //sum[100]是最大的宽度节点个数
    return 0;}
```

应用

题目描述（相关题目）

红黑树是一类特殊的二叉搜索树，其中每个结点被染成红色或黑色。若将二叉搜索树结点中的空指针看作是指向一个空结点，则称这类空结点为二叉搜索树的前端结点。并规定所有前端结点的高度为 -1−1。

一棵红黑树是满足下面“红黑性质”的染色二叉搜索树：

1、每个结点被染成红色或黑色；

2、每个前端结点为黑色结点；

3、任一红结点的子结点均为黑结点；

在从任一结点到其子孙前端结点的所有路径上具有相同的黑结点数。

从红黑树中任一结点 xx 出发（不包括结点 xx），到达一个前端结点的任意一条路径上的黑结点个数称为结点 xx 的黑高度，记作 bh(x)bh(x) 。红黑树的黑高度定义为其根结点的黑高度。

给定正整数 NN，试设计一个算法，计算出在所有含有 NN 个结点的红黑树中，红色内结点个数的最小值和最大值。

应用

输入格式

输入一个数字 N。

输出格式

输出第一行为红色内节点的个数最小值；第二行为红色内节点的个数最大值。

输入输出样例

输入 #1 复制

```
8
```

输出 #1

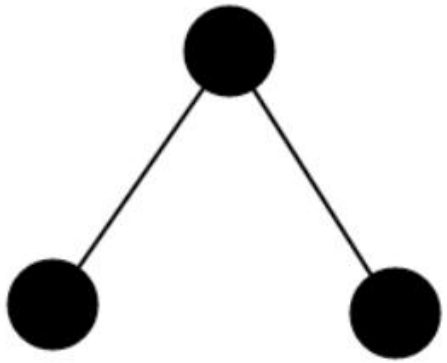
```
1
4
```

应用

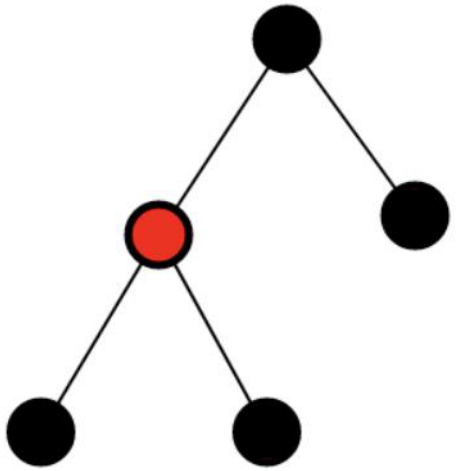
思路

先把每一个节点看成黑色的，通过红黑树性质来把一些结点变成红色的。
分为三种情况来考虑：
第一种，三个节点全是黑色，
第二种，只有

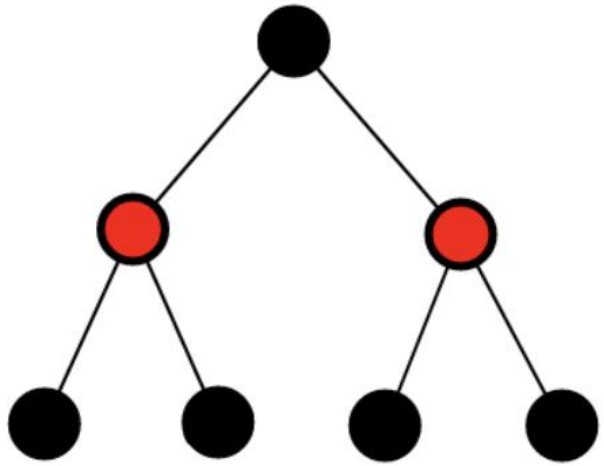
- $case1$:



- $case2$:



- $case3$:



应用

思路

```cpp
int main()
{
    int n, ans, k;
    cin>>n;
    k = n + 1;
    while (k > 1)
    {
        ans += k & 1;
        k >>= 1;
    }
    cout<<ans<<endl;

    k = n + 1;
    ans = 0;
    while (k > 1)
    {
        if (k == 2)
            ans++, k--;
        else if ((k & 3) == 1)
            ans += ((k >> 2) << 1) - 1, k >>= 2, k++;
        else if ((k & 3) == 2)
            ans += ((k >> 2) << 1), k >>= 2, k++;
        else if ((k & 3) == 3)
            ans += ((k >> 2) << 1) + 1, k >>= 2, k++;
        else
            ans += (k >> 1), k >>= 2;
    }
    cout<<ans<<endl;
    return 0;
}
```

二叉堆

1 概述

2 数据结构的定义

3 关键基本操作的具体实现

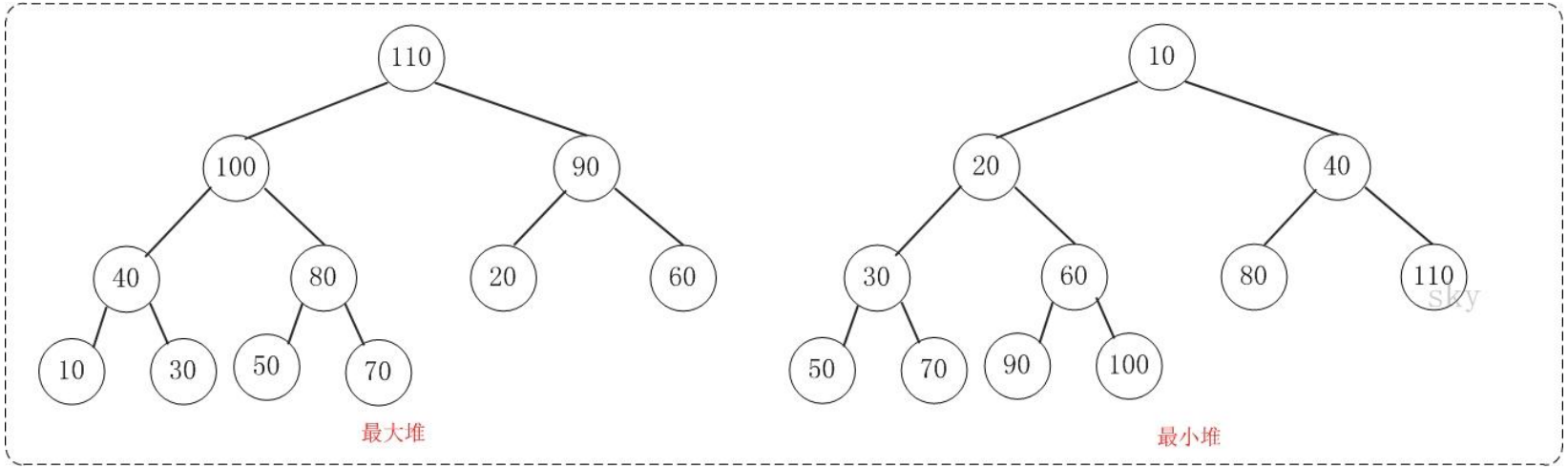
4 应用

概述

二叉堆是完全二元树或者是近似完全二元树，按照数据的排列方式可以分为两种：最大堆和最小堆。

•最大堆：父结点的键值总是大于或等于任何一个子节点的键值，也叫大根堆；

•最小堆：父结点的键值总是小于或等于任何一个子节点的键值，也叫小根堆。



二叉堆的实现一般是用简单的数组即可实现，在采用数组实现时，可以找到两个特别的规律：

左儿子：l_ son=parent*2;

右儿子：r_son=parent*2+1;

二叉堆 数据结构的定义

二叉堆是基于完全二叉树的基础上，加以一定的条件约束的一种特殊的二叉树。

根据约束条件的不同，二叉堆又可以分为两个类型：
大顶堆和小顶堆。

二叉堆是一颗二叉树，即每个节点只有一个父节点并且每个节点最多有两个子节点的树形结构。 二叉堆分为最大堆和最小堆，如果堆中的任意一个节点的值都大于等于它的子节点，称为最大堆。如果堆中的任意一个节点的值都小于等于它的子节点，称为最小堆。

二叉堆 基本操作

二叉堆初始化代码实现：

因为堆是基于完全二叉树，所以我们不需要用链式结构来表示，我们可以直接用数组存。
假设父节点的下表为parent，从父节点获取子节点：
左节点下标： 2*parent+1
右节点下标： 2*parent+2
假设子节点的下标为son(左右子节点都可以)：
父节点下标：(son-1)/2
因此得到节点之间的关系

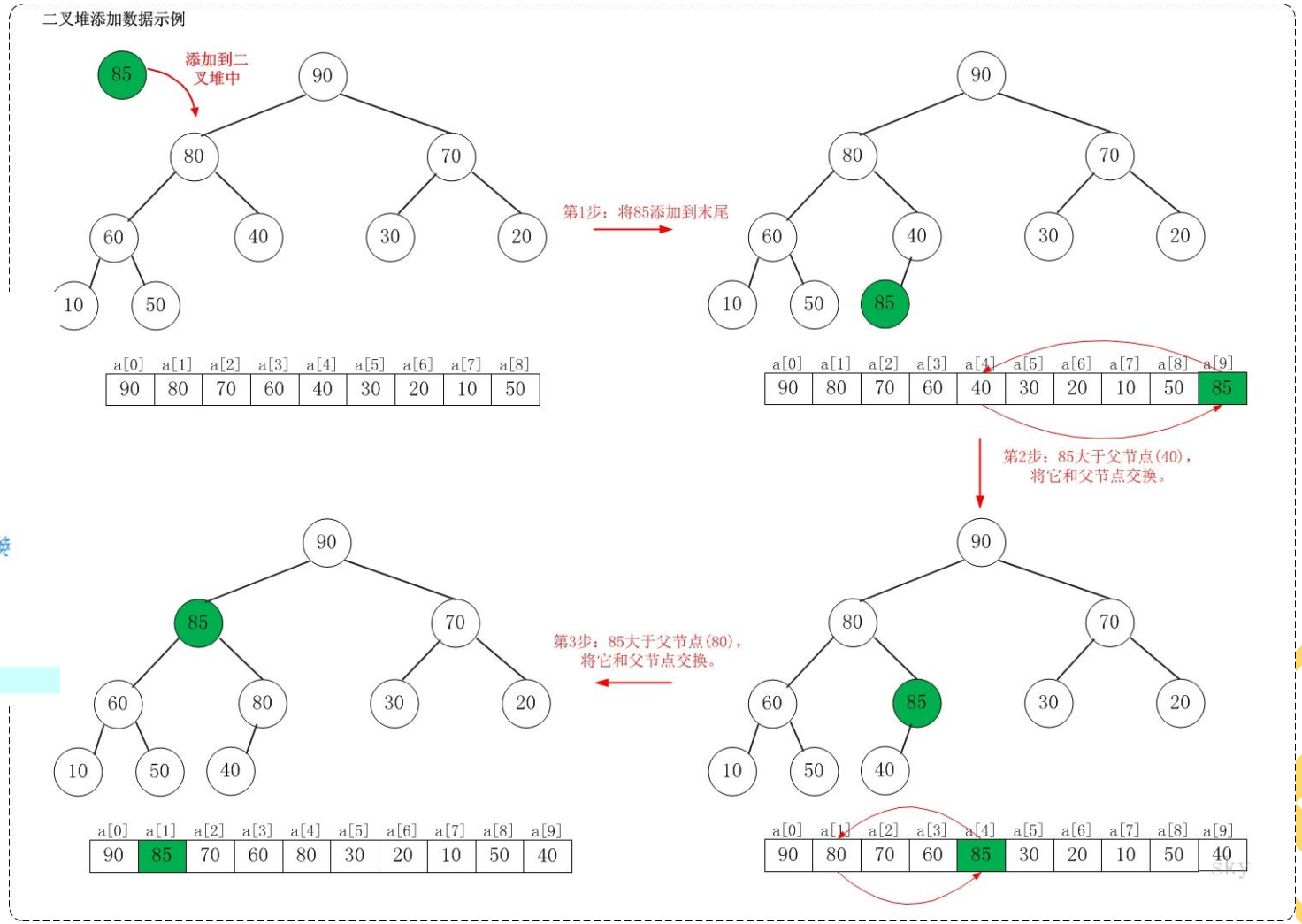
```
//初始化代码:
class BinaryHeap {
private:
    // 存放二叉堆的数组
    int *container = NULL;
    // 容量，也就是最多能存放多少个节点
    unsigned int capacity = 0;
     // 二叉堆的大小
    unsigned int size = 0;

public:
    BinaryHeap(unsigned capacity) {
        assert(capacity > 0);
        // 初始化存放二叉堆的数组
        this->container = new int[capacity];
        this->capacity = capacity;
    }
};
```

二叉堆 基本操作

插入操作：

从底依次比较交换即可

```
void swap(int &x,int &y){int t=x;x=y;y=t;}//交换函数
int heap[N];//定义一个数组来存堆
int siz;//堆的大小
void push(int x){//要插入的数
    heap[++siz]=x;
    now=siz;
    //插入到堆底
    while(now){//还没到根节点，还能交换
        ll nxt=now>>1;//找到它的父亲
        if(heap[nxt]>heap[now])swap(heap[nxt],heap[now]);//父亲比它大，那就交换
        else break;//如果比它父亲小，那就代表着插入完成了
        now=nxt;//交换
    }
    return;
}
```

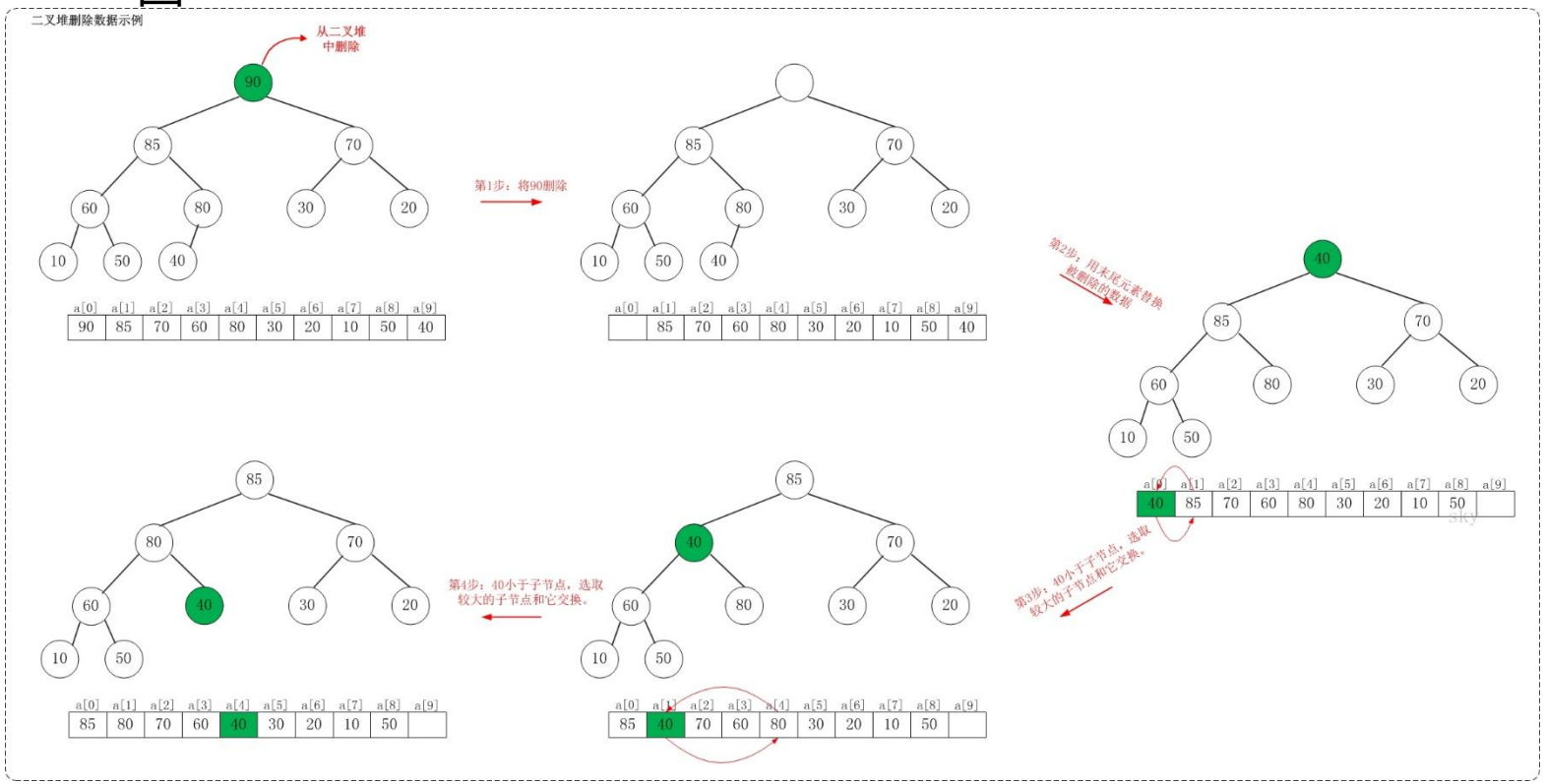


二叉堆 基本操作

删除操作：

若删除叶子结点则直接删除，若不是则如图

```
void pop(){
    swap(heap[siz],heap[1]);siz--;//交换堆顶和堆底，然后直接弹掉堆底
    int now=1;
    while((now<<1)<=siz){//对该节点进行向下交换的操作
        int nxt=now<<1;//找出当前节点的左儿子
        if(nxt+1<=siz&&heap[nxt+1]<heap[nxt])nxt++;//看看是要左儿子还是右儿子跟它换
        if(heap[nxt]<heap[now])swap(heap[now],heap[nxt]);//如果不符合堆性质就换
        else break;//否则就完成了
        now=nxt;//往下一层继续向下交换
    }
}
```



注意：在数组形式保存的二叉堆中对于i的结点而言，其父节点为（i-1）/2，用左移运算符可以实现

二叉堆 数据结构的定义

实现

二叉堆初始化代码实现：

因为堆是基于完全二叉树，所以我们不需要用链式结构来表示，我们可以直接用数组存。
假设父节点的下表为parent，从父节点获取子节点：
左节点下标： 2*parent+1
右节点下标： 2*parent+2
假设子节点的下标为son(左右子节点都可以)：
父节点下标：(son-1)/2
因此得到节点之间的关系

```
//初始化代码:
class BinaryHeap {
private:
    // 存放二叉堆的数组
    int *container = NULL;
    // 容量，也就是最多能存放多少个节点
    unsigned int capacity = 0;
     // 二叉堆的大小
    unsigned int size = 0;

public:
    BinaryHeap(unsigned capacity) {
        assert(capacity > 0);
        // 初始化存放二叉堆的数组
        this->container = new int[capacity];
        this->capacity = capacity;
    }
};
```

二叉堆 应用

题目描述

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 n−1 次合并之后， 就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 11 ，并且已知果子的种类 数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有 3 种果子，数目依次为 1 ， 2 ， 9 。可以先将 1 、 2 堆合并，新堆数目为 3 ，耗费体力为 3 。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12 ，耗费体力为 12 。所以多多总共耗费体力 3+12=15 。可以证明 15 为最小的体力耗费值。

二叉堆 应用

输入格式

共两行。

第一行是一个整数 $n(1 \leq n \leq 10000)$，表示果子的种类数。

第二行包含 n 个整数，用空格分隔，第 i 个整数 $a_i(1 \leq a_i \leq 20000)$ 是第 i 种果子的数目。

输出格式

一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 2^{31} 。

输入输出样例

输入 #1 复制

```
3
1 2 9
```

输出 #1

```
15
```

二叉堆 应用

思路

1、将n堆果子入堆建立二叉堆，

2、每次拿出最小的两堆（取堆中最小，弹堆顶，再取最小，再弹堆顶）

3、合并成为一堆，记录答案

4、将新的一堆加入堆中

5、重复上述过程直至剩下一堆果子。

二叉堆 应用

操作

```
int main()
{
  cin>>n;
  for(int i=1; i<=n; i++)
  {
    int a; cin>>a;
    insert(a); //建立二叉堆
  }
  long long ans=0;
  while(size>=2) //如果还可合并
  {
    int top1=gettop(); //取出堆顶（堆中最小值）后删除堆顶
    extract();
    int top2=gettop(); //同上
    extract();
    ans+=(top1+top2);
    insert(top1+top2); //将两数之和加入二叉堆，重复运算
  }
  cout<<ans<<endl; //输出答案
  return 0;
}
```

二叉堆 应用

操作

```
void up(int p) //二叉小根堆向上调整（子节点小于父节点就调整）
{
  while(p>1)
  {
    if(heap[p]<heap[p/2])
    {
      swap(heap[p],heap[p/2]);
      p/=2;
    }
    else break;
  }
}
```

```
void insert(int val) //二叉堆插入，新元素放在堆底，向上调整
{
  heap[++size]=val;
  up(size);
}
```

```
void down(int p) //二叉小根堆向下调整
{
  int s=p*2;
  while(s<=size)
  { //下面这句话是从左右儿子中选一个更小的做交换
    if(s<size&&heap[s+1]<heap[s]) s++;
    if(heap[s]<heap[p])
    {
      swap(heap[s],heap[p]);
      p=s; s=p*2;
    }
    else break;
  }
}
```

```
int gettop() //返回堆顶的值
{
  return heap[1];
}
```

```
void extract() //二叉堆删除堆顶
{
  heap[1]=heap[size--]; //将堆底移至堆顶，向下调整
  down(1);
}
```

参考资料

洛谷网：https://www.luogu.com.cn/problem/P5566
CSDN:https://blog.csdn.net/jiang_wang01/article/details/113715033?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216482721471678036655854 5%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%25.

https://blog.csdn.net/xiedeacc/article/details/27946831?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216482885111678218461330 2%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=164828851116782184613302&biz_id=0&utm_medium=distribute.pc_se

THANKS!