Chapter 10 RED BLACK TREES

红黑树的性质

- 为二叉查找树
- **根元素规定为黑色**
- 两大规则（用于保证树的平衡）
    - red rule：红色的节点不能有红色的孩子（※添加红色叶子一定不违反path rule）
    - path rule：每一条路径（**从根节点到没有孩子或只有一个孩子的节点**）的黑色节点的数量都须相同

    ※规则导致的特性：从红黑树的几乎所有非叶节点都有两个子女来说，它是相当浓密的。实际上，如果某个项只有一个子女，那么这个项必定是黑色的，而它的子女一定是一个红色的树叶。

    - 应用：添加节点
        - 添加红色叶子一定不违反path rule
        - 先确保path rule不被违反 然后再改变颜色 使其不违反red rule
        - 当发现某侧树高过高时 需利用旋转操作减少树高
        - 先调整颜色 颜色无法调整再调整结构（旋转）
- 红黑树的高度
    - 和n成对数关系，n为树中项的数量。
    - 最小高度：当一个红黑树是完全的，除了最底层是红色叶子，其他所有项都是黑色，此时树高最小。

$$\log_2 n$$

    - 最大高度

红黑树的最大高度小于$2\log_2 n$

红黑树的定义

```
enum color_type = {red, black};

struct rb_tree_node
{
    color_type color_field;
    rb_tree_node* parent_link;
    rb_tree_node* left_link;
    rb_tree_node* right_link;
    Value value_field;
};
```

- **头节点header为红色**
- **新插入的节点初始颜色一定为红色**（叶子）
    - 若为根节点 插入时仍未红色 但会被改成黑色
- 特殊节点**NIL**（允许空也需被表示成一个节点）
    - 允许NULL指针作为节点的指针
    - **颜色域规定为黑色**
    - 数据域为NULL
    - 左右孩子为NULL
    - 父母指针初始为NULL
- 辅助函数

color (y) **instead of** y -> color_field

parent (header) **instead of** header -> parent_link

root( ) **instead of** header -> parent

This encapsulates allocator details. And we can test for color (y) == black **without having a special case for** y = NIL.

红黑树的insert操作

// 课堂笔记

插入操作看节点的父母和父母兄弟 主要为颜色规则

case1：插入节点的父母和父母兄弟都是红色 → 改变颜色 父母和父母兄弟都调整为黑色 父母的父母调整为红色

case2：父母（在左边）是红色 父母兄弟为空（黑色） 插入节点为右孩子 父母节点进行左旋变黑 父母的父母进行右旋变黑

case3：父母（在左边）是红色 父母兄弟为空（黑色） 插入节点为左孩子 父母节点右旋调黑色 父母的父母调红色

- 插入操作步骤
  1. 创建一个由x指向的节点
  2. 在x（指向的）节点的value_field字段中存储项v
  3. 用BinSearchTree将节点作为一个树叶插入，并将x的颜色初始化为red
  4. 必要时进行重新着色或调整结构（维护两大规则）

     ※如果x为根节点则不需要进行此步骤，因为第五步会将根节点x的颜色设置为黑色；如果color(parent(x))是黑色，则不需要进行重构，因为red rule未被

违反。

5. 将根的颜色设置为黑色 color(root()) = black

- 步骤四的**while循环由parent(x)的兄弟节点决定**

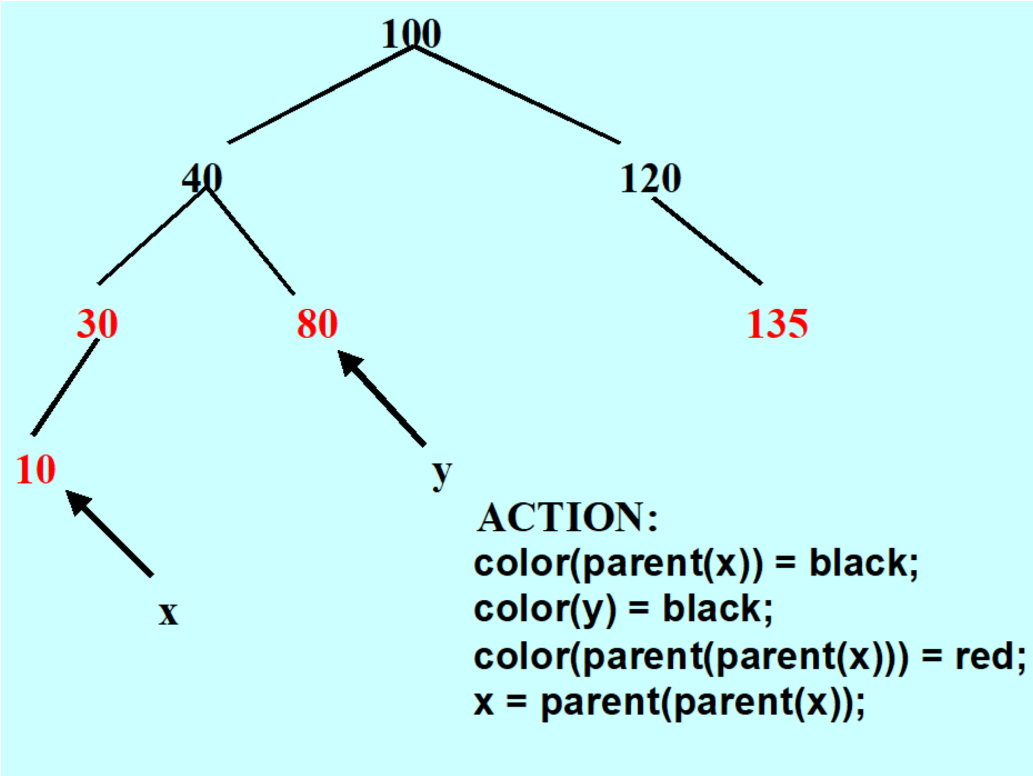
  ※循环进行条件：当x≠root()且color(parent(x))==red时，即当x==root()或color(parent(x))==black时，循环终止。

  ※基本思想：如果case1不适用，那就先进行case2，case2之后总要应用case3。

```
while (x != root() && color(parent(x)) == red)
    if (parent(x) == left(parent(parent(x)))) // parent
(x)是左孩子
    {
        y = right(parent(parent(x))); // y为parent(x)的兄
弟节点
        if (color(y) == red) // case1
        {
            color(parent(x)) = black;
            color(y) = black;
            color(parent(parent(x))) = red;
            x = parent(parent(x));
        }
        else // y为黑色
        {
            if (x == right(parent(x))) // case2
            {
                x = parent(x);
                rotate_left(x);
            }
            // case3一定进行
            color(parent(x)) = black;
            color(parent(parent(x))) = red;
            rotate_right(parent(parent(x)));
        }
}
else // parent(x)是一个右孩子
```

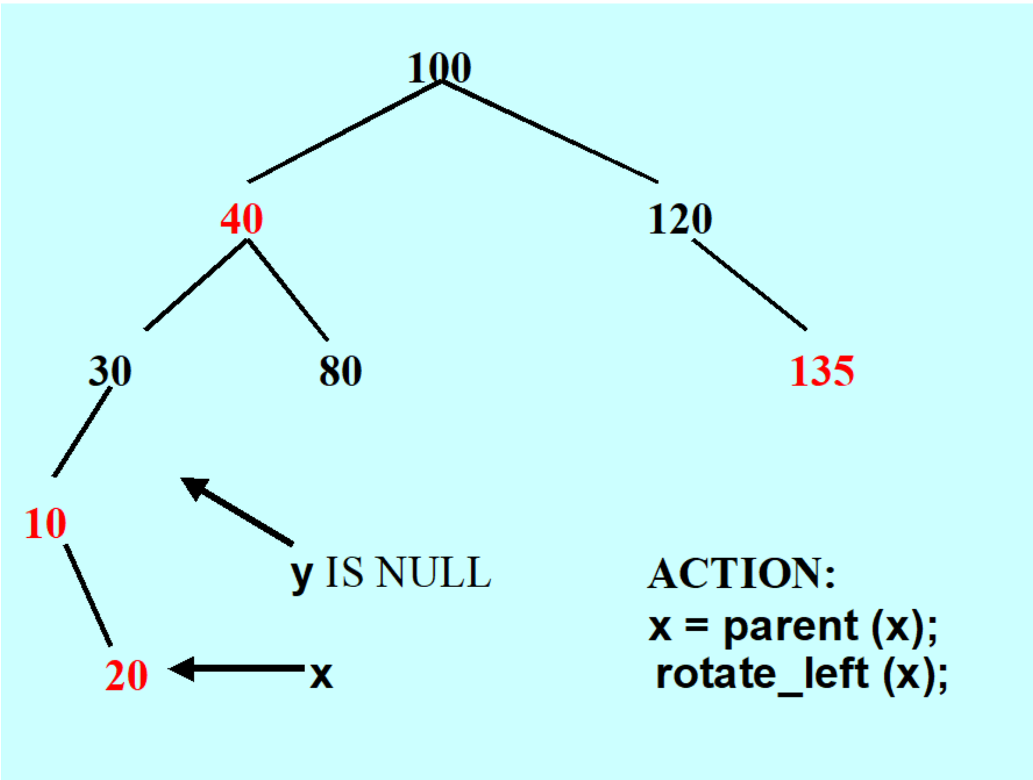
```
{
    y = left(parent(parent(x)));
    if (color(y) == red) // case1
    {
            color(parent(x)) = black;
            color(y) = black;
            color(parent(parent(x))) = red;
            x = parent(parent(x));
    }
else // y为黑色
{
            if (x == left(parent(x))) // case2
            {
                x = parent(x);
                rotate_right(x);
            }
            // case3一定进行
            color(parent(x)) = black;
            color(parent(parent(x))) = red;
            rotate_left(parent(parent(x)));
        }
}
```

- **parent(x) is a left child 插入节点的父母节点为左孩子（将父母节点的兄弟节点用y表示）**
    - **case 1: color(y) is red**



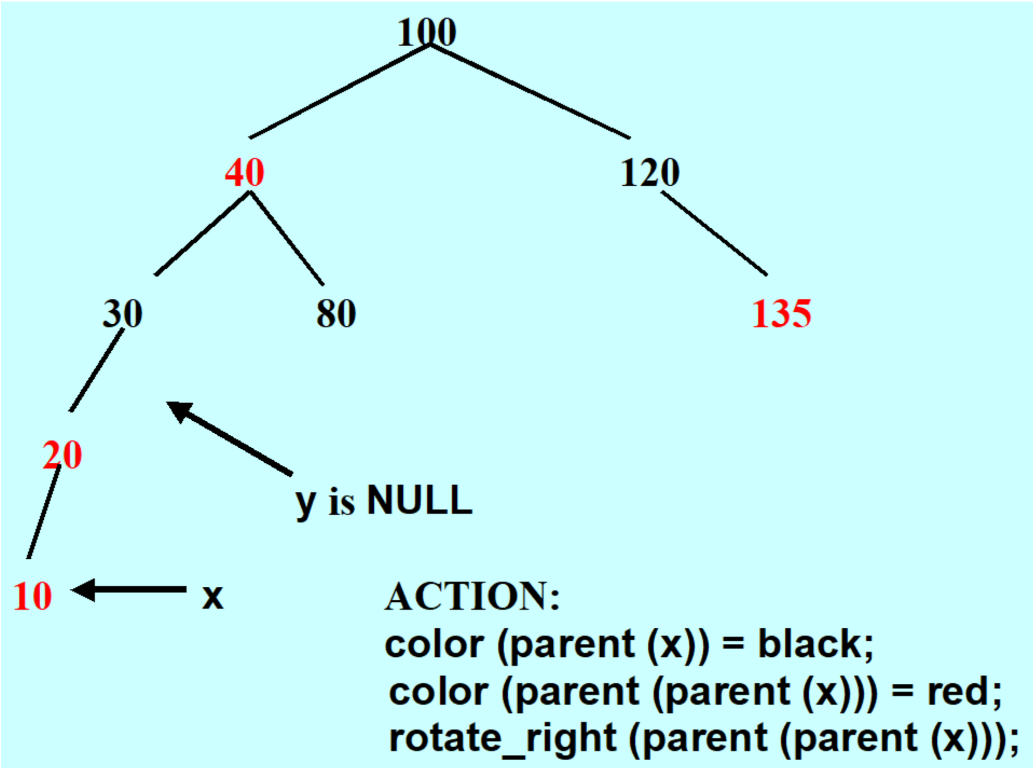
- **case 2: color(y) is black and x is a right child**

※空节点为黑色



- **case 3: color(y) is black and x is a left child**

※case3进行后x的父亲颜色一定会是黑色，故while循环一定终止



- parent(x) is a right child **插入节点的父母节点为左孩子（将父母节点的兄弟节点用y表示）**
  - 省略

- **插入操作的最坏和平均时间复杂度都为logarithmic in n**

红黑树的erase操作

// 课堂笔记

删除操作看节点和节点兄弟 主要为路径规则

1. 叶子或只有一个孩子（实际删除的）
2. 两个孩子 利用后记代替其位置 变为第一种情况

删除红色节点不会影响红色规则和路径规则

删除黑节点时有一个孩子 此孩子为红色节点才不会影响路径规则 被删除节点的的孩子节点颜色要变成被删除节点的颜色 让删除节点父母的孩子指针指向删除节点孩子 删除节点孩子父母指针指向删除节点父母

删除黑色节点时没有孩子 一定会违反路径规则

x is left

case1：兄弟节点是红色 则该兄弟节点一定有两个黑孩子

case2：兄弟节点是黑色 该兄弟孩子节点同为空（非实在黑色）【终结状态】- 把兄弟节点直接变为红色 x向上移至父母节点

case3：兄弟节点是黑色 该兄弟孩子节点左为红 右为空 这种结构需要调整 先做一次右旋

case4：作为旋转后 兄弟节点的孩子红节点转至右边 左为空 - 节点和兄弟节点的共同父母整个做左旋降低树高 旋上去的右孩子w需要设成和父母一样的颜色 才能替换父母的位置

```
while (x is not the root and x's color is black)
{
    if (x == left (parent (x)))
    {
        w = right (parent (x));
        if (w's color is red)
        { ... Case 1 ... }
        if (w's children are both black)
        { ... Case 2 ... }
        else
        {
            if( w's right child is black )
            { ... Case 3 ... }
            ... Case 4 ...
        } // one of w’s children is red
    } // x a left child
```

删除操作的**最坏时间复杂度为log 平均时间复杂度为常数**（因为二叉查找树的元素主要集中在下面几排）

```
while (x != root() && color(x) == black)
    if (x == left(parent(x))) // x为左孩子
    {
            link_type w = right(parent(x)) // x的兄弟节点
            if (color(w) == red) // case1
            {
                color(w) = black;
                color(parent(x)) = red;
                rotate_left(parent(x));
                w = right(parent(x));
            }
                if (color(left(w)) == black && color(right
(w))
                        == black) // case2
                {
                    color(w) =red;
                    x = parent(x);
                }
                else
                {
                        if (color(left(w)) = red) // case3
                        {
                            color(left(w)) = black;
                            color(w) = red;
                            rotate_right(w);
                            w = right(parent(x));
                        } // if-case3 end
                        // case4
                        color(w) = color(parent(x));
                        color(parent(x)) = black;
                        color(right(w)) = black;
                        rotate_left(parent(x));
                        break;
                        } // else end
```

```
    } // if-xisleft end
    else // x为右孩子
    {
        link_type w = left(parent(x));
        if (color(w) == red) // case1
        {
            color(w) = black;
            color(parent(x)) = red;
            rotate_right(parent(x));
            w = left(parent(x));
        }
        if (color(right(w)) == black && color(left(w)) == b
lack) // case2
        {
            color(w) = red;
            x = parent(x);
        }
        else
        {
            if (color(left(w)) == black) // case3
            {
                color(right(w)) = black;
                color(w) = red;
                rotate_left(w);
                w = left(parent(X));
            }
            // case4
            color(w) = color(parent(x));
            color(parent(x)) = black;
            color(left(w)) = black;
            rotate_right(parent(x));
            break;
    }
} // while循环结束
color(x) = black;
```

标准模板库的关联容器

- 关联容器是通过项之间键的比较来确定项位置的容器

只由键组成吗?	允许重复项吗?	
	是	否
是	多集合	集合
否	多映射	映射

- 四个类都基于红黑树实现
- HP的红黑树类

```
template <class Key, class Value,
            class KeyOfValue, class Compare>
class rb_tree
{
```

Key: The type of the key: the part of an item (may be the whole item) used in comparisons with other items

Value: The type of each item

KeyOfValue: A function-class type: returns the key from the value

Compare: A function-class type for comparing keys

- set类

```
// set类声明的开头
template <class Key, class Compare = less<Key>,
                    //class Allocator = allocator<Key> >
```

```
class set {
    typedef rb_tree<Key,Key,ident<Key, Key>,Compare> rep
_type;
    rep_type t; // red-black tree representing set
```

在标准C++中，set类声明的开头如下:

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key>>
class set
{
```

和往常一样，先忽略分配器的工作。在set类里，键就是整个的项，而且键是独一无二的。例如，下面是set对象的定义，其中包含了按词典顺序的字符串项:

```
set <string> names;
```

因为集合是一个关联容器，它的项是根据和其他项之间的比较进行存储的。而比较使用的是模板参数Compare对应的模板变元，这个模板变元在缺省情况下是函数类less，它在标准模板库中的<function>里的定义如下:

```
template <class T>
struct less : binary_function<T, T, bool> {
    bool operator( )(const T& x, const T& y) const { return x < y; }
};
```

因此如果函数对象comp是函数类less的一个实例，那么comp(x,y)将返回表达式x<y的数值。当然，不使用缺省情况，也可以指定除less之外的函数类——甚至可以指定用户声明的函数类。例如，可以定义一个set对象，其中包含降序存储的**double**类型的项:

```
set<double, greater <double> > salaries;
```

  - set类的Compare模板变元**缺省情况为函数类less**，则set中为**降序存储**。
    - 调用红黑树方法则为**左小右大**
  - set类包含了通常的各色容器方法：多个构造器，一个析构器，begin，end，size，empty，find，insert和erase。（大部分都是让t调用相应的rb_tree方法，因此其最坏时间复杂度都与n成对数关系

- 函数类 function class
  - 定义一个类用运算符重载函数
  - 重载的是函数调用运算符operator（）
  - 即这个类的对象完全具有函数的特征

```
class Sample // 定义一个函数类Sample
{
    public:
        double operator( ) (int i)
        {
            return 1.0 / i;
        } // operator()
}; // class Sample

Sample f; // 函数类Sample的一个对象
cout << f (39) << endl;
```

- pair类
    - 用于映射
    - pair组合从而可以整个放入红黑树的节点中 存键值对
    - 允许函数返回两个值
        - 应用（同时返回一个迭代器和一个boo值）pair<iterator, bool> insert(const value_type& x);

- multiset类
    - 每个项只有一个值，但**允许重复项存在**
        - insert方法只需要返回itr指向的新插入项位置即可，无需返回bool判断是否重复

          iterator insert (const key_type& x);
        - lower_bound方法返回某个**项第一次出现的位置**
        - upper_bound方法返回某个项最后一个可以出现的地方（不打乱多集合的前提下）

          iterator upper_bound (const T& x) const;

          如果x在multiset中，则返回的迭代器位于x最后出现的位置**后面**。
        - equal_range方法返回一对迭代器

pair<iterator, iterator> equal_range(const key_type& x);

For example, to print out each occurrence of "jade" in the multiset words:

```
typedef multiset<string>::iterator itr_type;

pair<itr_type, itr_type> range = words.equal_range ("jade");

for (itr_type itr = range.first; itr != range.second; itr++)
    cout << *itr << endl;
```

- map类
  - 类中每个值都为pair<key, T>
  - 不允许重复键
  - 开头代码

    ```
    template<class Key, class T, class Compare = less<Key
    > >
    class map
    {
    }
    ```

  - 例：key为城市名，第二部分为城市人口数，故可得定义

    map<string, int> population;

The following fragment inserts some pairs into the map and then prints out the map:

```
population.insert (pair<string, int>("Easton", 35000));
population.insert (pair<string, int> ("Bethlehem", 47000));
population.insert (pair<string, int> ("Allentown", 61000));

map<string, int>::iterator itr;
for (itr = population.begin(); itr != population.end(); itr++)
   cout << itr -> first << " " << itr -> second << endl;
```

- 若想对城市人口（第二部分）进行修改，不能通过insert，因为这样key会重复，可以通过下标操作实现

```
// Postcondition: If there is a pair with key x in this map,
//                a reference to the second component
//                in that pair has been returned.
//                Otherwise, the pair <x, T( )> has been
//                inserted in this map and a reference to
//                the second component in that pair has
//                been returned.  The worstTime(n) is
//                O(log n).
T& operator[ ] (const key_type& x);
```

pair中第二个组件的引用被返回

For example, we can modify the above-constructed population container as follows:

population [“Easton”] = 44000;
population [“Coopersburg”] = 7000;

The effect of these two assignments is to change the pair <“Easton”, 35000> TO <“Easton”, 44000> and to add the pair <“Coopersburg”, 7000> to the map population.