红黑树

AVL：平衡二叉树

AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；  
红黑是弱平衡的，用非严格的平衡来换取增删节点时候旋转次数的降低；  
所以简单说，搜索的次数远远大于插入和删除，那么选择AVL树，如果搜索，插入删除次数几乎差不多，应该选择RB树。

**1.stl中的set底层用的什么数据结构？**

**2.红黑树的数据结构怎么定义的？**

**3.红黑树有哪些性质？**

**4.红黑树的各种操作的时间复杂度是多少？**

**5.红黑树相比于BST和AVL树有什么优点？**

**6.红黑树相对于哈希表，在选择使用的时候有什么依据？**

**7.如何扩展红黑树来获得比某个结点小的元素有多少个？**

**8.扩展数据结构有什么步骤？**

**详细解答**

**1.stl中的set底层用的什么数据结构？**

红黑树

**2.红黑树的数据结构怎么定义？**

**[cpp]** [view plain](http://blog.csdn.net/silangquan/article/details/18655795" \o "view plain) [copy](http://blog.csdn.net/silangquan/article/details/18655795" \o "copy) [IMG_256](https://code.csdn.net/snippets/166878" \o "在CODE上查看代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)[IMG_257](https://code.csdn.net/snippets/166878/fork" \o "派生到我的代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)

1. **enum** Color
2. {
3. RED = 0,
4. BLACK = 1
5. };
7. **struct** RBTreeNode
8. {
9. **struct** RBTreeNode\*left, \*right, \*parent;
10. **int**   key;
11. **int** data;
12. Color color;
13. };

**3.红黑树有哪些性质？**

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：  
1）每个结点要么是红的，要么是黑的。  
2）根结点是黑的。  
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。  
4）如果一个结点是红的，那么它的俩个儿子都是黑的。  
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

**4.红黑树的各种操作的时间复杂度是多少？**

能保证在最坏情况下，基本的动态几何操作的时间均为O（lgn）

**5.红黑树相比于BST（二叉搜索树）和AVL（平衡树）树有什么优点？**

红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。

相比于BST，因为红黑树可以能确保树的最长路径不大于两倍的最短路径的长度，所以可以看出它的查找效果是有最低保证的。在最坏的情况下也可以保证O(logN)的，这是要好于二叉查找树的。因为二叉查找树最坏情况可以让查找达到O(N)。

红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高，所以在插入和删除中所做的后期维护操作肯定会比红黑树要耗时好多，但是他们的查找效率都是O(logN)，所以红黑树应用还是高于AVL树的. 实际上插入 AVL 树和红黑树的速度取决于你所插入的数据.如果你的数据分布较好,则比较宜于采用 AVL树(例如随机产生系列数),但是如果你想处理比较杂乱的情况,则红黑树是比较快的

**6.红黑树相对于哈希表，在选择使用的时候有什么依据？**

权衡三个因素: 查找速度, 数据量, 内存使用，可扩展性。  
　　总体来说，hash查找速度会比map快，而且查找速度基本和数据量大小无关，属于常数级别;而map的查找速度是log(n)级别。并不一定常数就比log(n) 小，hash还有hash函数的耗时，明白了吧，如果你考虑效率，特别是在元素达到一定数量级时，考虑考虑hash。但若你对内存使用特别严格， 希望程序尽可能少消耗内存，那么一定要小心，hash可能会让你陷入尴尬，特别是当你的hash对象特别多时，你就更无法控制了，而且 hash的构造速度较慢。

红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。  
  
在实际的系统中，例如，需要使用动态规则的防火墙系统，使用红黑树而不是散列表被实践证明具有更好的伸缩性。Linux内核在管理vm\_area\_struct时就是采用了红黑树来维护内存块的。

红黑树通过扩展节点域可以在不改变时间复杂度的情况下得到结点的秩。

**7.如何扩展红黑树来获得比某个结点小的元素有多少个？**

这其实就是求节点元素的顺序统计量，当然任意的顺序统计量都可以需要在O(lgn)时间内确定。

在每个节点添加一个size域，表示以结点 x 为根的子树的结点树的大小，则有

size[x] = size[[left[x]] + size [right[x]] + 1;

这时候红黑树就变成了一棵顺序统计树。

利用size域可以做两件事：

1). 找到树中第i小的结点；

**[cpp]** [view plain](http://blog.csdn.net/silangquan/article/details/18655795" \o "view plain) [copy](http://blog.csdn.net/silangquan/article/details/18655795" \o "copy) [IMG_258](https://code.csdn.net/snippets/166878" \o "在CODE上查看代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)[IMG_259](https://code.csdn.net/snippets/166878/fork" \o "派生到我的代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)

1. OS-SELECT(x;,i)
2. r = size[left[x]] + 1;
3. **if** i == r
4. **return** x
5. elseif i < r
6. **return** OS-SELECT(left[x], i)
7. **else** **return** OS-SELECT(right[x],  i)

思路：size[left[x]]表示在对x为根的子树进行中序遍历时排在x之前的个数，递归调用的深度不会超过O(lgn);

2).确定某个结点之前有多少个结点，也就是我们要解决的问题；

**[cpp]** [view plain](http://blog.csdn.net/silangquan/article/details/18655795" \o "view plain) [copy](http://blog.csdn.net/silangquan/article/details/18655795" \o "copy) [IMG_260](https://code.csdn.net/snippets/166878" \o "在CODE上查看代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)[IMG_261](https://code.csdn.net/snippets/166878/fork" \o "派生到我的代码片" \t "http://blog.csdn.net/silangquan/article/details/_blank)

1. OS-RANK(T,x)
2. r = x.left.size + 1;
3. y = x;
4. **while** y != T.root
5. **if** y == y.p.right
6. r = r + y.p.left.size +1
7. y = y.p
8. **return** r

思路：x的秩可以视为在对树的中序遍历种，排在x之前的结点个数加上一。最坏情况下，OS-RANK运行时间与树高成正比，所以为O (lgn).

**8.扩展数据结构有什么步骤？**

1).选择基础数据结构；

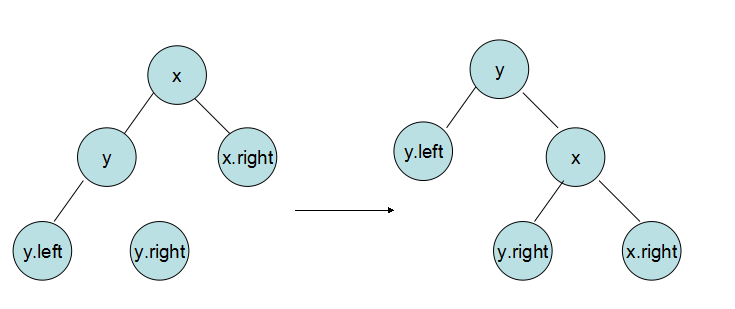
2).确定要在基础数据结构种添加哪些信息；

3).验证可用基础数据结构上的基本修改操作来维护这些新添加的信息；

4).设计新的操作。

有序统计树是以红黑树为基本数据结构，增加了一个字段size,size表示以当前结点为树根的树上的节点数目，很明显有这样的关系根节点的size等于左孩子的size加上右孩子的size+1。   
另外，如果之前有了解过有序统计量的相关知识，其中有一个操作便是查找排好顺序的列表中的某个特定索引的元素。那么有序统计树也是在红黑树的基础上加了这个操作。

下面来分析OST（有序统计树）在红黑树上变化的部分。   
1.每个结点多了一个size属性，叶子节点的size=1，Nil（哨兵结点）的size=0；   
2.当我们插入一个行的结点时我们要修改插入点路径上的所有父节点的size+1；同时因为在插入结点的时候要维护红黑树的性质，那么涉及到左旋转和右旋转的操作，就需要调整变动结点的size，删除操作也是一样，在红黑树删除维护操作时，也要进行左右旋操作，所以也要进行结点的size调整

在这里我们我们先分析右旋转操作   
  
如图所示：   
当进行右旋操作时，y取代了x的位置，那么y.size=x.size,因为原来x的size等于该树上所有节点的数目。转完之后x的size也要更新   
即：   
y.size=x.size;   
x.size=x.left.size+x.right.size+1;   
左旋操作是一样的

删除操作：   
删除一个节点，那么它的上层的所有父节点的size要减一。   
1）当删除结点只有一个孩子时，删除该节点，孩子节点替换他的位置，并不需要调整其他结点的size   
2）当删除结点有两个孩子节点时，我们先找到该节点的替换结点（它在删除结点的右子树上，可能直接是删除结点的右孩子），替换结点的size是删除结点的size-1（树上少了一个节点），其他都不变

红黑树的插入：

红黑树的插入相当于在二叉查找树的插入基础上，为了重新恢复平衡，重新着色：

插入Z后，首先将Z的左孩子和右孩子都置为叶子节点，然后将Z着色为红色。

1. 如果插入的Z为根节点，直接涂成黑色，
2. 如果插入的节点Z的父节点是黑色，那么没有破坏平衡（前面的五条性质），什么也不用做.
3. 如果当前节点Z的父节点是红色，而且祖父节点的另一个子节点（叔父节点）也是红色。那么就将当前节点的父节点和叔父节点都涂成黑色，祖父节点涂成红色，然后把祖父节点置成当前节点
4. 如果当前结点的父节点是红色且叔父节点是黑色，当前结点是父结点的左孩子，那么就将父节点变成黑色，祖父节点变成红色，以祖父节点为支点进行右旋。
5. 如果当前结点的父节点是红色且叔父节点是黑色，当前结点是父结点的右孩子，那么就将父节点作为新的当前结点，并以当前结点作为 支点进行左旋。