昵称：ljsspace
园龄：1年
粉丝：9
关注：0
+加关注

< 2011年6月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

McCreight算法 后缀树(1)

随笔分类

趣味杂题(2)
数据结构和算法(35)
图形图像(2)

随笔档案

2011年10月 (2)
2011年9月 (2)
2011年8月 (13)
2011年7月 (11)
2011年6月 (40)

最新评论XML

1. Re:AA树 - 红黑树的变种
LZ 写得很好 学习下!
--LittleBirds
2. Re:First time play with Java CV/OpenCV to detect faces
there is an exception running this projectException in thread "main" java.lang.UnsatisfiedLinkError: C:\Users\lmc\AppData\L

伸展树(splay tree)自顶向下的算法

伸展树(splay tree)是一种能自我调整的二叉搜索树(BST)。虽然某一次的访问操作所花费的时间比较长，但是平摊（amortized）之后的访问操作（例如旋转）时间能达到O(logn)的复杂度。对于某一个被访问的节点，在接下来的一段时间内再次频繁访问它（90%的情况下是这样的，即符合90-10规则，类似于CPU内或磁盘的cache设计原理）的应用模式来说，伸展树是一种很理想的数据结构。这是因为最近被访问的节点一直位于根 节点的附近，从而再次被访问时的搜索路径长度比较小。这点与平衡的二叉树(比如AVL树和红黑树)不一样。另外一点与其他平衡二叉树的区别是，伸展树不需 要存储任何像AVL树中平衡因子(balance factor)那样的平衡信息，可以节省空间的开销。

但是伸展树也有自己的缺点，比如不像其他平衡二叉树那样即使最坏情况下也能达到O(logn)访问时间，它的最坏情况下只有O(n)，跟单向链表一样。另外，伸展树的查找操作会修改树的结构，这与普通意义上的查找为只读操作习惯不太一样。

伸展树的实现有两种方式，一是自底向上(bottom-up)，另外一种是自顶向下(top-down)。考虑到实现的难易程度，自顶向下的实现方式比较 简单，因为自底向上需要保存已经被访问的节点，而自顶向下可以在搜索的过程中同时完成splay操作。两者得出的树结构可能不太一样，但是他们的平摊时间 复杂度都是O(logn)。两种实现的基本操作就是splay，splay将最后被访问到的节点提升为根节点。splay具体操作过程是，在一般情况下每 次考虑两级节点（目标节点的父节点和祖父节点），按照目标节点和父节点的各自所处的位置是它们各自的父节点的左孩子还是右孩子，可以分为：zig或 zag，zig-zig或zag-zag，zig-zag或zag-zig六种，其中后一种是前一种的对称形式。zig表示某节点(可以为目标节点或父节 点)是它的父节点的左孩子；zag表示某节点(可以为目标节点或父节点)是它的父节点的右孩子。

在自顶向下(top-down)的实现中，需要将输入的树拆成三颗树，分别为左树L，中树M和右树R。其中M树维护当前还未被访问到的节点，L树中所有节 点的值都小于M树中的任何节点值，R树中所有节点的值都大于M树中的任何节点值。L树中只需要知道当前的最大节点 (leftMax)，而R树中只需要知道当前的最小节点(rightMin)。左右两棵树的根节点分别可以通过pseudoNode节点（它是 leftMax和rightMin的初始值，而且splay过程中变量pseudoNode本身未变化,只改变它的左右孩子节点）的右和左孩子节点得到， 因为leftMax中加入一个新的节点或子树时都是将新的节点作为leftMax的右孩子，而不是左孩子（注意这里的顺序），rightMin跟 leftMax相反。自顶向下的zig-zig或zag-zag需要做旋转操作，zig-zig的旋转操作叫rotateLeftChild,旋转后目标 节点的父节点和祖父节点加入R树，zag-zag的旋转操作叫rotateRightChild,旋转后目标节点的父节点和祖父节点加入L树。另外 zig-zag或zag-zig可以分别简化为zig或zag操作，这样可以将zig-zag和zig合二为一，从而只需考虑一种情况，而不需要将两种情 况单独考虑。zig操作将目标节点的父节点加入R树，zag操作将目标节点的父节点加入L树。注意L和R树中每次加入新节点都需更新变量leftMax或 rightMin。自顶向下splay操作的最后一步是重组(re-assemble)：将M树的左孩子设置为L树的根节点，将M树的右孩子设置为R树的 根节点，然后M树原来的左孩子成为leftMax的右孩子，M树原来的右孩子成为rightMin的左孩子。

伸展树的基本操作及其实现：
1）查找boolean find(int x)：查找操作只需要splay最后被访问的节点。在splay操作中，如果目标值x比最后一个被查找的叶子节点小，表示未找到该值，则splay该叶子 节点作为新树的根节点，返回false；如果目标值x比最后一个被查找的叶子节点大，也表示未找到该值，则splay该叶子节点作为新树的根节点，返回 false。如果目标值x跟某个节点值匹配，直接splay该节点作为新树的根节点，返回true。

2）找最大和最小值int findMax()和int findMin()：findMax操作只需要将splay的目标值设定为整数最大值Integer.MAX_VALUE，就可以将树中的max值splay到根节点。findMin类似。

3）删除最大和最小值int deleteMax()和int deleteMin()：deleteMax调用findMax()之后，这时最大值位于根节点而且根节点没有右子树（因为所有的值都比max小），只需 删除根节点，然后将根节点的左子树设置成新的根节点。deleteMin类似。有了这两个操作和下面的insert操作，splay树可以作为优先队列 （priority queue）使用，它的平摊时间复杂度与用堆（完全二叉树）实现的优先队列相当。

4）插入新节点void insert(int x)：目的是插入完成后将x节点变成根节点。另外插入操作不允许原树中已经有相同值x的节点。首先splay(x)，将最后被访问的节点变为根节点，如果 x跟root节点的值一样，说明重复插入；如果x比root节点的值小，但是由于 splay(x)之后，x值肯定比root左子树中的任何节点值都大,因为root节点是比x大的节点中最小的，这时可以将splay后的树拆成两颗子 树，左边为根节点的左孩子作为左子树的root，右边为splay后的树的根节点作为右子树的根节点，然后将x设为root，它的左孩子为拆分出的左子 树，它的右孩子为拆分出的右子树；如果x比root节点的值大，做法类似。

5）删除已经存在的节点void remove(int x)：删除操作不允许原树中不存在具有相同值x的节点。首先splay(x)，将最后被访

问的节点变为根节点，如果x跟root节点的值不一样，说明原树 中不存在x值的节点；如果root的左子树为空，说明x是最小节点，可以直接删除root，然后将root的右孩子作为新的根节点；如果root的左子树 不为空，可以通过findMax的一个重载方法Node findMax (Node rootNode) 只找左子树的最大值（查找的过程同时splay左子树的最大值作为该子树的根节点）,然后左子树最大值节点作为新树的根节点，删除原树的根节点，将新树根 节点的右孩子设置为原树根节点的右孩子。

以下是使用自顶向下(top-down)方式实现伸展树及其基本操作。

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
/**
 * Splay tree and its operations
 * @author ljs
 * 2011-06-01
 *
 * 1. findXXX() methods are mutable operation
 * 2. All operations are not thread-safe because the tree can be changed by any operation.
 *
 */
public class SplayTree {
    static class Node{
        int val;
        Node left;
        Node right;
        public Node(int val){
            this.val = val;
        }
        public String toString(){
            return String.valueOf(val);
        }
    }
    private Node root;
    public SplayTree(Node root){
        this.root = root;
    }

    public Node getRoot() {
        return root;
    }
    //find a node in the tree
    //return true if found; false otherwise
    public boolean find(int x) throws Exception{
        if(root == null) throw new Exception("tree is empty.");
        splay(x);
        if(root.val == x){
            return true;
        }else{
            return false;
        }
    }
```

ocal\Temp\jniopencv_core4413207510914710109.dll: Can't find dependent librari...
--dundun
3. Re:LCA问题的在线算法（很经典的一个算法）
无限仰慕楼主，太神了！！！！
请问什么地方能找到《On finding lowest common ancestors - Simplification and parallelization》这篇论文，
如果有电子版的话，能不能发到我的邮箱yz_sjr@163.com
谢谢！！
--avoid1111
4. Re:伸展树(splay tree)自顶向下的算法
不得不赞一个！！
--为爱疯狂
5. Re:拯救湖心的风筝（一道趣味平面几何问题）
@百分百好牛用这个连接试试：http://hi.csdn.net/attachment/201106/17/9952465_1308291270nO2N.gif
--ljsspace

阅读排行榜

1. LCA问题的在线算法（很经典的一个算法）(1856)
2. 配对堆(Pairing Heap)(1476)
3. 后缀树(Suffix Tree)的文本匹配算法(776)
4. 构建后缀树的Ukkonen算法及其实现(580)
5. LCA的Tarjan离线算法(568)

评论排行榜

1. LCA问题的在线算法（很经典的一个算法）(3)
2. 左倾堆（一种可高效合并的优先队列)(3)
3. 配对堆(Pairing Heap)(2)
4. 拯救湖心的风筝（一道趣味平面几何问题）(2)
5. 用非递归方法遍历二叉树（附Morris算法）(2)

推荐排行榜

1. 由1-7的随机整数函数构造1-10随机整数函数(1)
2. 用非递归方法遍历二叉树（附Morris算法）(1)

```
    private Node findMax(Node rootNode) throws Exception{
        if(rootNode == null) throw new Exception("tree or subtree is empty.");

        rootNode = splay(rootNode,Integer.MAX_VALUE);
        return rootNode; //rootNode.val=max
    }
    public int findMax() throws Exception{
        root = findMax(this.root);
        return root.val;
    }

    private Node findMin(Node rootNode) throws Exception{
        if(rootNode == null) throw new Exception("tree or subtree is empty.");


        rootNode = splay(rootNode,Integer.MIN_VALUE);
        return rootNode; //rootNode.val=min
    }

    public int findMin() throws Exception{
        root = findMin(this.root);
        return root.val;
    }
    public int deleteMax() throws Exception{
        int max = findMax();
        this.root = root.left;
        return max;
    }
    public int deleteMin() throws Exception{
        int min = findMin();
        this.root = root.right;
        return min;
    }

    public void insert(int x) throws Exception {
        if(root == null){
            //set the new node as root
            this.root = new Node(x);
        }else{
            splay(x);
            if(root.val == x){
                throw new Exception("duplicate value!");
            }else if(x<root.val){
                //split the splayed tree with right subtree including root, and set the new
node as root
                Node tmp = new Node(x);
                tmp.left = this.root.left;
                tmp.right = this.root;
                root.left = null;
                this.root = tmp;
            }else {//ie. x>root.val
                //split the splayed tree with left subtree including root, and set the new n
ode as root
                Node tmp = new Node(x);
                tmp.left = this.root;
                tmp.right = this.root.right;
                root.right = null;
                this.root = tmp;
            }
        }
    }
    public void remove(int x) throws Exception {
        if(root == null) throw new Exception("tree is empty.");

        splay(x);
        if(x != root.val){
            throw new Exception("value not found.");
        }

        if(root.left == null){
            //root(root.val==x) is the min node
            root = root.right;
        }else{
            //find the max value from left subtree, and
            //then remove root and join the right subtree with the left splayed subtree
            Node leftSubTreeRoot = this.findMax(this.root.left);
            leftSubTreeRoot.right = this.root.right;
            root = leftSubTreeRoot;
        }

    }
    private void rotateLeftChild(Node grandparent,Node parent){
        grandparent.left = parent.right;

        parent.right = grandparent;
        //split the parent with middle tree
        parent.left = null;
    }

    private void rotateRightChild(Node grandparent,Node parent){
        grandparent.right = parent.left;

        parent.left = grandparent;
        //split the parent with middle tree
        parent.right = null;
    }
    //x: the target value to be found for splaying
    public void splay(int x){
        this.root = splay(this.root,x);
    }
    //x: the target value to be found for splaying
    //rootNode: the root node of the tree to be splayed
    //return the new root of the splayed tree or subtree
    private Node splay(Node rootNode,int x){
        if(rootNode == null) return null;

        Node pseudoNode = new Node(Integer.MAX_VALUE);
        //left tree root (no left child)
        Node leftMax = pseudoNode;
        //right tree root (no right child)
        Node rightMin = pseudoNode;

        Node t =  rootNode;
        while(true){
            if(x == t.val){
                break;
            }else if(x<t.val){
                //Note: the variable parent is target's parent, the variable t is target's g
randparent
                Node parent = t.left;
                if(parent == null){
                    break;
                }else{
                    if(x < parent.val){
                        if(parent.left == null){
                            //zig
                            t.left = null;
                            rightMin.left = t;
                            rightMin = t;
                            t = parent;
                        }else{
                            //zig-zig
                            Node tmp = parent.left;

                            rotateLeftChild(t,parent);

                            //update right tree and its min node
                            rightMin.left = parent;
                            rightMin = parent;

                            //update the middle tree's root
                            t = tmp;
                        }
                    }else{ //ie. x >= t.left.val
                        //zig or zig-zag(simplified to zig)
                        t.left = null;
                        rightMin.left = t;
                        rightMin = t;

                        t = parent;
                    }
                }
            }else{ //ie. x>t.val
                Node parent = t.right;
                if(parent == null){
                    break;
                }else{
                    if(x > parent.val){
                        if(parent.right == null){
                            //zag
                            t.right = null;
                            leftMax.right = t;
                            leftMax = t;
                            t = parent;
                        }else{
                            //zag-zag
                            Node tmp = parent.right;

                            rotateRightChild(t,parent);

                            //update left tree and its max node
                            leftMax.right = parent;
                            leftMax = parent;

                            //update the middle tree's root
                            t = tmp;
                        }
                    }else{ //ie. x <= t.right.val
                        //zag or zag-zig (simplified to zag)
                        t.right = null;
                        leftMax.right = t;
                        leftMax = t;
                        t = parent;
                    }
                }
            }
        }
        //re-assemble (note: even if the above while is not executed, the following code works as expected.)
        leftMax.right = t.left;
        rightMin.left = t.right;

        t.left = pseudoNode.right;  //pseudoNode.right is the root of left tree
        t.right = pseudoNode.left;  //pseudoNode.left is the root of right tree

        return t;
    }

    //utility method for test purpose
    public static int getNumberOfNodes(Node root){
        int tmp = NODES;
        NODES = 0;
        return tmp;
    }
    public static int NODES=0;
    public static void recursiveInOrderTraverse(Node root){
        if(root == null)return;

        recursiveInOrderTraverse(root.left);
        System.out.format(" %d", root.val);
        NODES++;
        recursiveInOrderTraverse(root.right);
    }
    //utility method for test purpose
    //n: the nodes number of the tree
    public static void displayBinaryTree(Node root,int n){
        if(root == null) return;

        LinkedList<Node> queue = new LinkedList<Node>();

        //all nodes in each level
        List<List<Node>> nodesList = new ArrayList<List<Node>>();

        //the positions in a displayable tree for each level's nodes
        List<List<Integer>> nextPosList = new ArrayList<List<Integer>>();

        queue.add(root);
        //int level=0;
        int levelNodes = 1;

        int nextLevelNodes = 0;
        List<Node> levelNodesList = new ArrayList<Node>();
        List<Integer> nextLevelNodesPosList = new ArrayList<Integer>();

        int pos = 0;  //the position of the current node
        List<Integer> levelNodesPosList = new ArrayList<Integer>();
        levelNodesPosList.add(0); //root position
        nextPosList.add(levelNodesPosList);
        int levelNodesTotal = 1;
        while(!queue.isEmpty()) {
            Node node = queue.remove();

            if(levelNodes==0){
                nodesList.add(levelNodesList);
                nextPosList.add(nextLevelNodesPosList);
                levelNodesPosList = nextLevelNodesPosList;

                levelNodesList = new ArrayList<Node>();
                nextLevelNodesPosList = new ArrayList<Integer>();

                //level++;
                levelNodes = nextLevelNodes;
                levelNodesTotal = nextLevelNodes;

                nextLevelNodes = 0;
            }
            levelNodesList.add(node);

            pos = levelNodesPosList.get(levelNodesTotal - levelNodes);
            if(node.left != null){
                queue.add(node.left);
                nextLevelNodes++;
                nextLevelNodesPosList.add(2*pos);
            }

            if(node.right != null) {
                queue.add(node.right);
                nextLevelNodes++;

                nextLevelNodesPosList.add(2*pos+1);
            }

            levelNodes--;
        }
        //save the last level's nodes list
        nodesList.add(levelNodesList);

        int maxLevel = nodesList.size()-1;  //==level

        //use both nodesList and nextPosList to set the positions for each node

        //Note: expected max columns: 2^(level+1) - 1
        int cols = 1;
        for(int i=0;i<=maxLevel;i++){
            cols <<= 1;
        }
        cols--;
        Node[][] tree = new Node[maxLevel+1][cols];

        //load the tree into an array for later display
        for(int currLevel=0;currLevel<=maxLevel;currLevel++){
            levelNodesList = nodesList.get(currLevel);
            levelNodesPosList = nextPosList.get(currLevel);
            //Note: the column for this level's j-th element: 2^(maxLevel-level)*(2*j+1) - 1
            int tmp = maxLevel-currLevel;
            int coeff = 1;
            for(int i=0;i<tmp;i++){
                coeff <<= 1;
            }
            for(int k=0;k<levelNodesList.size();k++){
                int j = levelNodesPosList.get(k);
                int col = coeff*(2*j + 1) - 1;
                tree[currLevel][col] = levelNodesList.get(k);
            }
        }

        //display the binary search tree
        System.out.format("%n");
        for(int i=0;i<=maxLevel;i++){
            for(int j=0;j<cols;j++){
                Node node = tree[i][j];
                if(node == null)
                    System.out.format("  ");
                else
                    System.out.format("%2d",node.val);
            }
            System.out.format("%n");
        }
    }

    public static void printAfterSplayed(SplayTree splayTree){
        Node root = splayTree.getRoot();
        System.out.format("%nAfter being splayed, in-order BST:%n");
        SplayTree.recursiveInOrderTraverse(root);

        System.out.format("%n%n%nAfter being splayed, the tree is:");
        SplayTree.displayBinaryTree(root,SplayTree.getNumberOfNodes(root));

    }

    public static void main(String[] args) throws Exception {
        System.out.format("%nTest case 1 - splay opeartion:%n");

        Node nn12 = new Node(12);
        Node nn5 = new Node(5);
        Node nn25 = new Node(25);
        Node nn20 = new Node(20);
        Node nn30 = new Node(30);
        Node nn15 = new Node(15);
        Node nn24 = new Node(24);
        Node nn13 = new Node(13);
        Node nn18 =  new Node(18);
        Node nn16 = new Node(16);

        nn12.left =nn5;
        nn12.right = nn25;
        nn25.left = nn20;
        nn25.right = nn30;
        nn20.left = nn15;
        nn20.right = nn24;
        nn15.left = nn13;
        nn15.right = nn18;
        nn18.left = nn16;


        Node root = nn12;
        System.out.format("%nBefore being splayed, in-order BST:%n");
        SplayTree.recursiveInOrderTraverse(root);

        SplayTree splayTree = new SplayTree(root);
        splayTree.splay(19);
        System.out.format("%n*****splay the node with value=19*****%n");
        printAfterSplayed(splayTree);




        System.out.format("*************************************");
        System.out.format("%nTest case 2 - splaytree's operations:%n");
        /*
         13
    10          25
       12    20     35
                  29
         */


        Node n13 = new Node(13);
        Node n10 = new Node(10);
        Node n25 = new Node(25);
        Node n12 = new Node(12);
        Node n20 = new Node(20);
        Node n35 = new Node(35);
        Node n29 = new Node(29);


        n13.left = n10;
        n13.right = n25;
        n10.right = n12;
        n25.left = n20;
        n25.right = n35;
        n35.left = n29;

        root = n13;
        System.out.format("%nBefore being splayed, in-order BST:%n");
        SplayTree.recursiveInOrderTraverse(root);
        splayTree = new SplayTree(root);
        int val=25;
        boolean found = splayTree.find(val);
        System.out.format("%n*****%d is in the tree? [%s]*****%n",val,found);
        printAfterSplayed(splayTree);


        int max = splayTree.findMax();
        System.out.format("%n*****max value=%d*****%n",max);
        printAfterSplayed(splayTree);

        int min = splayTree.findMin();
        System.out.format("%n*****min value=%d*****%n",min);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        min = splayTree.deleteMin();
        System.out.format("%n*****deleted min value: %d*****%n",min);
        printAfterSplayed(splayTree);

        int newval = 24;
        splayTree.insert(newval);
        System.out.format("%n*****insert new value %d*****%n",newval);
        printAfterSplayed(splayTree);


        int removeVal = 12;
        splayTree.remove(removeVal);
        System.out.format("%n*****remove value %d*****%n",removeVal);
        printAfterSplayed(splayTree);



        System.out.format("**************************************");
        System.out.format("%nTest case 3 - priority queue:%n");

        Node m1= new Node(1);
        Node m4= new Node(4);
        Node m7= new Node(7);
        Node m9= new Node(9);
        Node m20= new Node(20);
        Node m22= new Node(22);
        Node m26= new Node(26);
        Node m29= new Node(29);
        Node m30= new Node(30);

        Node m36= new Node(36);
        m1.right = m4;
        m4.right = m7;
        m7.right = m9;
        m9.right = m20;
        m20.right = m22;
        m22.right = m26;
        m26.right = m29;
        m29.right = m30;
        m30.right = m36;

        root = m1;
        System.out.format("%nBefore being splayed, in-order BST:%n");
        SplayTree.recursiveInOrderTraverse(root);

        splayTree = new SplayTree(root);
        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        newval = 16;
        splayTree.insert(newval);
        System.out.format("%n*****insert new value %d*****%n",newval);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);

        newval = 12;
        splayTree.insert(newval);
        System.out.format("%n*****insert new value %d*****%n",newval);
        printAfterSplayed(splayTree);
        max = splayTree.deleteMax();
        System.out.format("%n*****deleted max value: %d*****%n",max);
        printAfterSplayed(splayTree);
    }
}
```

测试:

```
Test case 1 - splay opeartion:

Before being splayed, in-order BST:
 5 12 13 15 16 18 20 24 25 30
*****splay the node with value=19*****

After being splayed, in-order BST:
 5 12 13 15 16 18 20 24 25 30


After being splayed, the tree is:
              18
      12               20
   5      15              25
        13  16         24  30
***********************************
Test case 2 - splaytree operations:

Before being splayed, in-order BST:
 10 12 13 20 25 29 35
*****25 is in the tree? [true]*****

After being splayed, in-order BST:
 10 12 13 20 25 29 35


After being splayed, the tree is:
              25
      13               35
  10       20      29
    12

*****max value=35*****

After being splayed, in-order BST:
 10 12 13 20 25 29 35


After being splayed, the tree is:
                              35
              25
      13               29
  10       20
    12

*****min value=10*****

After being splayed, in-order BST:
 10 12 13 20 25 29 35


After being splayed, the tree is:
              10
                      25
                  13      35
                12  20  29
```

```
*****deleted max value: 35*****

After being splayed, in-order BST:
 10 12 13 20 25 29


After being splayed, the tree is:
              25
      10              29
          13
        12   20

*****deleted min value: 10*****

After being splayed, in-order BST:
 12 13 20 25 29


After being splayed, the tree is:
      25
  13       29
12   20

*****insert new value 24*****

After being splayed, in-order BST:
 12 13 20 24 25 29


After being splayed, the tree is:
              24
      20              25
  13                      29
12

*****remove value 12*****

After being splayed, in-order BST:
 13 20 24 25 29


After being splayed, the tree is:
              20
      13              24
                        25
                          29
**********************************
Test case 3 - priority queue:

Before being splayed, in-order BST:
 1 4 7 9 20 22 26 29 30 36
*****deleted max value: 36*****

After being splayed, in-order BST:
 1 4 7 9 20 22 26 29 30


After being splayed, the tree is:
                              4
              1                               9
                                      7               22
                                                  20       29
                                                        26   30

*****deleted max value: 30*****
```

```
After being splayed, in-order BST:
 1 4 7 9 20 22 26 29


After being splayed, the tree is:
              9
       4              29
   1       7       22
               20   26

*****deleted max value: 29*****

After being splayed, in-order BST:
 1 4 7 9 20 22 26


After being splayed, the tree is:
       9
   4       22
 1   7   20   26

*****deleted max value: 26*****

After being splayed, in-order BST:
 1 4 7 9 20 22


After being splayed, the tree is:
              22
       9
   4       20
 1   7

*****insert new value 16*****

After being splayed, in-order BST:
 1 4 7 9 16 20 22


After being splayed, the tree is:
              16
       9              20
   4                      22
 1   7

*****deleted max value: 22*****

After being splayed, in-order BST:
 1 4 7 9 16 20


After being splayed, the tree is:
                              20
              16
       9
   4
 1   7

*****deleted max value: 20*****

After being splayed, in-order BST:
 1 4 7 9 16


After being splayed, the tree is:
              16
```

```
      9
  4
1   7

*****insert new value 12*****

After being splayed, in-order BST:
 1 4 7 9 12 16


After being splayed, the tree is:
              12
      9            16
  4
1   7

*****deleted max value: 16*****

After being splayed, in-order BST:
 1 4 7 9 12


After being splayed, the tree is:
              12
      9
  4
1   7
```

参考资料:

1. "Self-adjusting Binary Search Trees", Sleator and Tarjan （1985）
2. A demonstration of top-down splaying (http://www.link.cs.cmu.edu/splay/)

分类: 数据结构和算法

« 博主前一篇: 经典算法（8）- 插入排序(Insertion Sort) 及三个基本排序算法的比较
» 博主后一篇: 经典算法（9）- 堆排序(Heapsort)

posted @ 2011-06-05 22:37 ljsspace 阅读(567) 评论(2) 编辑 收藏

评论列表

#1楼 2011-08-11 16:35 wtx

大赞～ 帅！！

支持(0) 反对(0)

#2楼 2011-10-10 10:58 为爱疯狂

不得不赞一个！！

支持(0) 反对(0)

程序员问答社区，解决您的技术难题

博客园首页　博问　新闻　闪存　程序员招聘　知识库

最新IT新闻:
· Chrome扩展将使用更少的系统资源
· IT老兵不死：柳传志隐退留下三大愿景
· Surface背后的变与不变：微软被迫变革求生存
· 解构商业模式 (一) — 搞清楚「客户」是谁
· 天宫一号与神舟九号24日实施手控交会对接
» 更多新闻...

最新知识库文章:
· 五年程序员人生的点点滴滴
· 代码质量随想录（五）：注得多不如注得巧
· 在代码重构中蜕变
· 我的大脑不能再处理面向对象了
· NoSQL 在腾讯应用实践
» 更多知识库文章...

China-Pub 低价书精选
China-Pub 计算机绝版图书按需印刷服务