链滴

平衡二叉树的理解

作者：suruns

原文链接：https://ld246.com/article/1578020296165

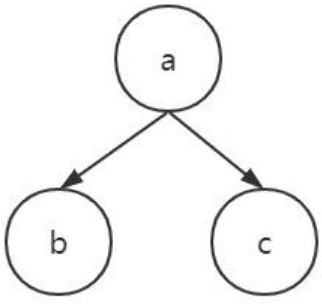
来源网站：链滴

1.定义

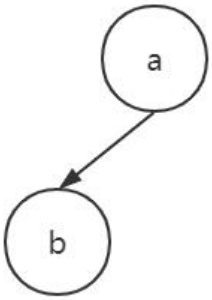
平衡二叉树是在搜索二叉树进化来的，搜索二叉树都知道，就是比根节点大的放右节点，反之放左节。但是在极端情况下，比如（1，2，3，4，5）这样排好序的情况下，搜索二叉树会退化成链表，所为了解决这个问题，产生了平衡二叉树，平衡二叉树要求左右节点的深度不能超过 1，为了达到这个件，需要旋转操作。

2.倾斜

首先，如下情况，为完全二叉树，我称之为完全平衡（完全平衡指左右高度差为 0，平衡二叉树的平指左右高度差小于 2）

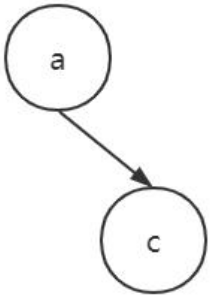


如下两种情况为不完全二叉树，都有一定的倾斜，下图为左斜一次



原文链接：平衡二叉树的理解

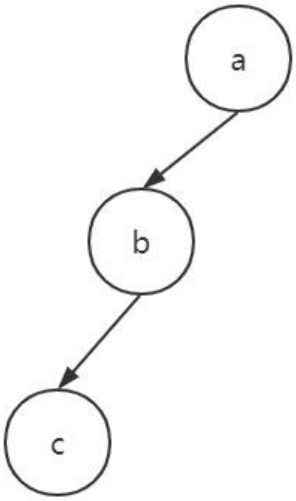
下图为右斜一次



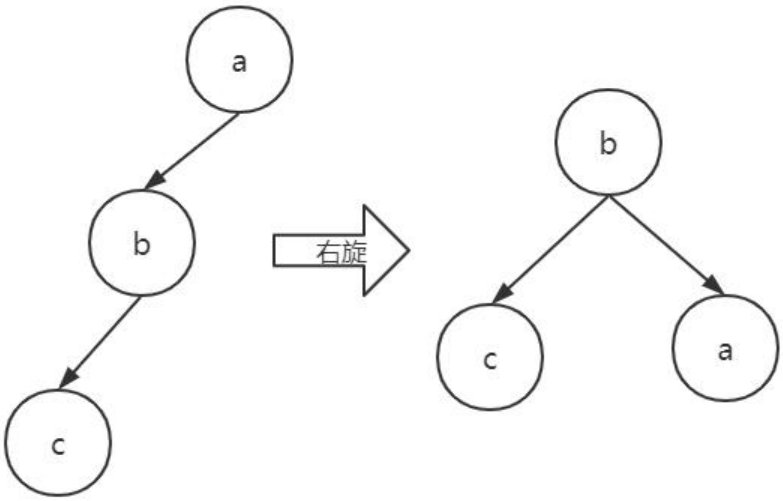
每倾斜一次，左右高度都会相差一，而平衡二叉数要求左右高度不能超过 1，所以在倾斜两次之后就要旋转操作，而两次倾斜会产生 4 种情况。

旋转

左斜两次

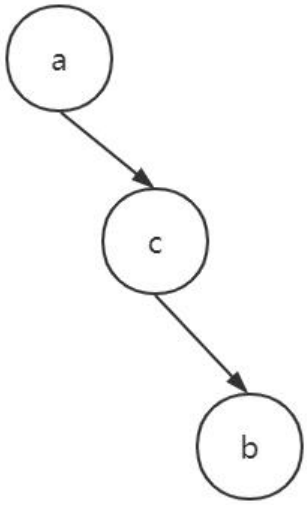


此时，只需要右旋一次即可保持平衡，

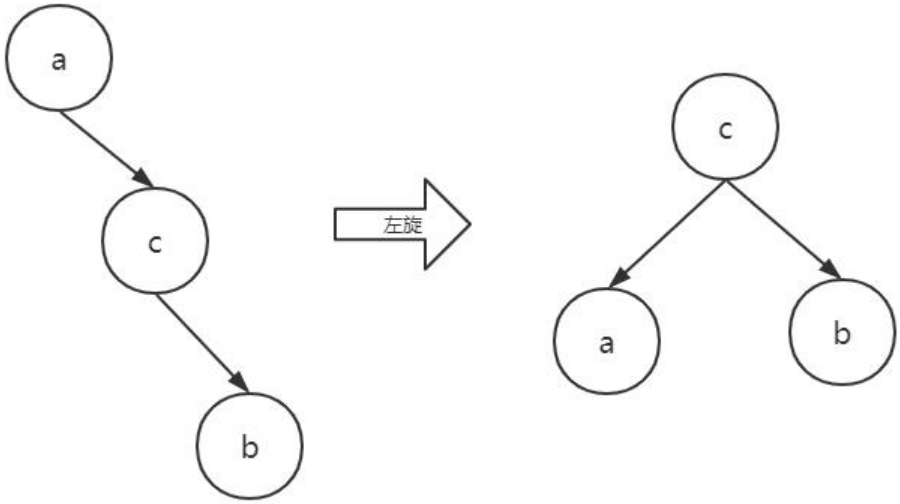


原文链接：平衡二叉树的理解

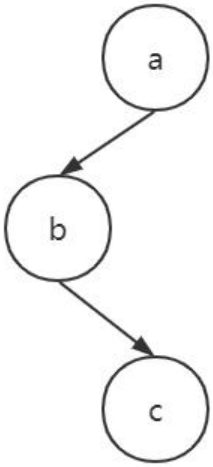
右斜两次



此时只需要左旋一次即可保持平衡

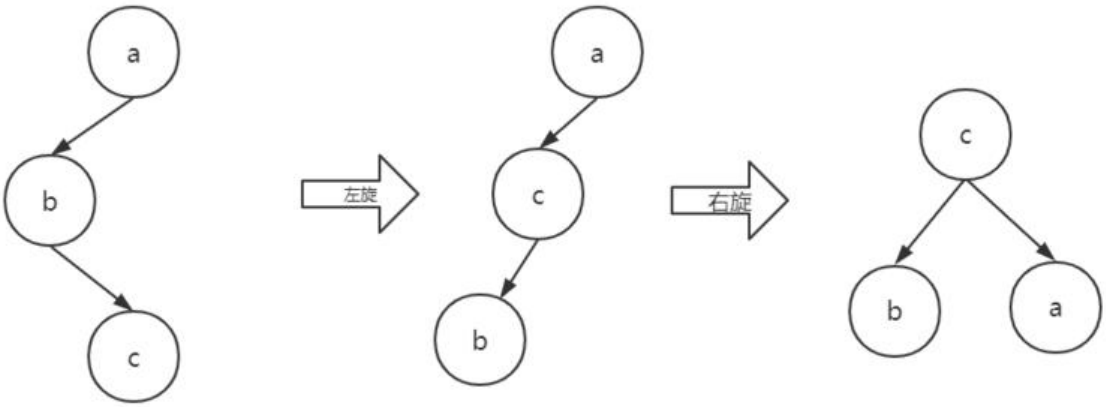


先左斜再右斜

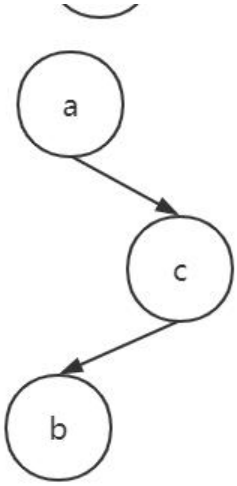


此时需要旋转两次才能平衡

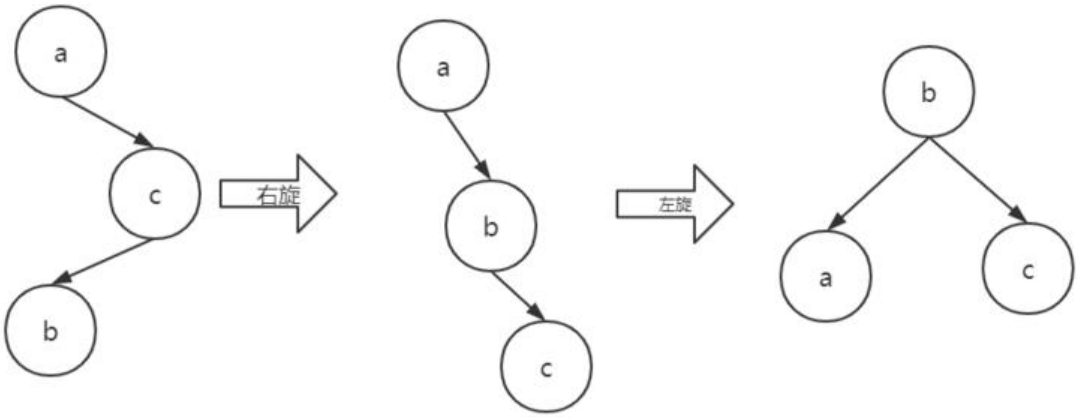
原文链接：平衡二叉树的理解



先右斜再左斜



此时需要右旋再左旋才能平衡



代码

好了，在了解了平衡二叉树的原理后就可以动手实现了，主要是用递归的思想，应该很好理解了

```
/**
* @Author: suruns
```

```
* @Date: 2019 年 12 月 26 日 11:12
* @Desc: 平衡二叉树
 */
 public class AverageTree {
 private Node root;
 public AverageTree() {
 }
 /**
 * 构造方法
 * @param values 数组
  */
  public AverageTree(Integer[] values) {
  if (Objects.isNull(values)) {
  return;
  }
  for (Integer value : values) {
  add(value);
  }
  }

 /**
 * 二叉树添加数值
 * @param value 数值
 * @return 是否成功
  */
  public boolean add(Integer value) {
  Node newNode = new Node(value);
  if (Objects.isNull(root)) {
  root = newNode;
  return true;
  } else {
  if (add(this.root, newNode)) {
  return doAverage(this.root);
  }
  }
  return false;
  }

 private boolean add(Node node, Node newNode) {
 if (newNode.value > node.value) {
 if (Objects.isNull(node.right)) {
 node.right = newNode;
 newNode.parent = node;
 return true;
 } else {
 return add(node.right, newNode);
 }
 } else {
 if (Objects.isNull(node.left)) {
 node.left = newNode;
 newNode.parent = node;
 return true;
 } else {
 return add(node.left, newNode);
```

```
}
}
}
public boolean delete(int value){
Node node = container(this.root, value);
if (Objects.isNull(node)){
return false;
}
}
private void delete(Node node){
if (Objects.isNull(node)){
throw new NullPointerException();
}
if (Objects.isNull(node.parent)){
this.root = null;
} else {
Node p = node.parent;
if (Objects.equals(p.left, node)){
p.left = null;
} else {
p.right = null;
}
}
}
public boolean container(Integer value) {
return Objects.nonNull(container(this.root, value));
}
private Node container(Node node, Integer value) {
if (Objects.isNull(node)) {
return null;
}
if (value.equals(node.value)) {
return node;
}
if (value.compareTo(node.value) > 0) {
if (Objects.nonNull(node.right)) {
return container(node.right, value);
} else {
return null;
}
} else {
if (Objects.nonNull(node.left)) {
return container(node.left, value);
} else {
return null;
}
}
}
private boolean doAverage(Node node) {
NodeStatus status = node.isAverage();
switch (status) {
case AVARAGE:
break;
case RIGHT_LEFT:
```

```
rightLeftRotate(node);
break;
case LEFT:
rightRotate(node);
break;
case RIGHT:
leftRotate(node);
break;
case LEFT_RIGHT:
leftRightRotate(node);
break;
default:
throw new IllegalArgumentException();
}
boolean left =  true;
boolean right = true;
if (node.hasRightChild()){
right = doAverage(node.right);
}
if (node.hasLeftChild()){
left = doAverage(node.left);
}
if (!node.hasChildren()){
return true;
} else {
return left && right;
}
}
/**
*
*
*
*
*
*
*

*/
private void leftRightRotate(Node node) {
Node b = node.right;
Node d = b.left;
if (d.hasLeftChild()){
b.left = d.left;
b.left.parent = b;
}
if (d.hasRightChild()){
b.left = d.right;
b.left.parent = b;
}
d.right = b;
b.parent = d;
node.right = d;
d.parent = node;
rightRotate(node);
```

```
}
/**
*
* / \       / \        /\
* b   f  ->   d   f  ->    b  a
* \       /          \  \
* d       b            e   f
* \       \
*

*/
private void rightLeftRotate(Node node) {
Node b = node.left;
Node d = b.right;
if (d.hasLeftChild()){
b.right = d.left;
b.right.parent = b;
}
if (d.hasRightChild()){
b.right = d.right;
b.right.parent = b;
}
d.left = b;
b.parent = d;
node.left = d;
d.parent = node;
rightRotate(node);
}
/**
*
*
*
*
*
*
*

*/
private void leftRotate(Node node){
Node c = node.right;
Node e = c.left;
changeParent(c, node);
node.parent = c;
c.left = node;
node.right = e;
if (Objects.nonNull(e)){
e.parent = node;
}
}
/**
*
*
*
*
```

原文链接：平衡二叉树的理解

```
*
*
*

*/
private void rightRotate(Node node){
Node b = node.left;
changeParent(b, node);
node.parent = b;
Node d = b.right;
b.right = node;
node.left = d;
if (Objects.nonNull(d)){
d.parent = node;
}
}
private void setRoot(Node node) {
// root 节点
this.root = node;
}
private void changeParent(Node node, Node old) {
Node p = old.parent;
if (Objects.isNull(p)) {
setRoot(node);
} else if (p.left == old) {
p.left = node;
} else if (p.right == old) {
p.right = node;
} else {
throw new IllegalArgumentException();
}
node.parent = p;
}
@Override
public String toString() {
List <Integer>  values = new ArrayList<>();
printNode(this.root, values);
return Strings.join(values, ' ');
}
private void printNode(Node node, List <Integer>  list) {
list.add(node.value);
if (Objects.nonNull(node.left)) {
printNode(node.left, list);
}
if (Objects.nonNull(node.right)) {
printNode(node.right, list);
}
}
private class Node {
private Integer value;
private Node left;
private Node right;
private Node parent;
}
```

```
/**
* 节点状态
 */
 private enum NodeStatus {
 // 左倾斜， 需要右旋， 左旋一次
 /**
 *
 *
 *
 *
 *
 *
 *

 */
 LEFT,
 // 先左旋再右旋
 /**
 * a
 * /
 * b
 * \
 * c
 * \
 *

 */
 LEFT_RIGHT,
 /**
 *
 *
 *
 *
 *
 *
 *

 */
 RIGHT,
 /**
 *
 *
 *
 *
 *
 *
 *

 */
 RIGHT_LEFT,
 // 平衡，不需要旋转
 AVARAGE
 }
 }
```

如有错误，请及时提醒，谢谢