java【二叉树】

作者：haxLook
原文链接：https://ld246.com/article/1619774504831
来源网站：链滴

上一篇文章为大家带来了一个关于链表的结构实现，我采用是内部类实现方式，那对于链表而言的话链表缺点就是检索数据，那当然有人会提出一些优化方法，例如我们在学习集合时候了解到，二叉树红黑树等，在数据改变的情况，进行对数据旋转，达到一个树的左右平衡，以此达到一个最佳的数据索到效率。

下面的一个demo主要便于我们理解二叉树数据结构，同样我们采用也是内部类的方式，对于红黑树话，其实是在二叉树的基本在进行进一步数据优化，但是目前我还不能用代码的形式为大家展示【很杂sob 】，有兴趣的可以去关于数据结构的书籍。

不说了，上代码

代码里面有注解大部分设计思路。

```
package ITaljavaT3;
/**
 * 实现二叉树的操作
 * @author Administrator
 * @param <T> 要进行二叉树的实现
 * <? extends Comparable<? super T>>
 * 代表任何实现了comparable接口的实例，且接口的类型是comparable<T 或其父类>。
<T extends Comparable<? super T>>
代表类型是T的实例，且这个T要实现comparable 接口，接口的类型是comparable<T 或其父类>
 */
class BinaryTree< T extends Comparable<T>>{
	//节点内部类
	private class Node{
		private Comparable<T> date;//保存数据在comparable中可以进行大小的比较，还可以进向下强制性转换;
		private Node parent; //保存父节点
		private Node left;   //保存左节点
		private Node right;  //保存左节点
		public Node(Comparable<T> date) {
			this.date=date;//进行数据的初始化
		}
		/**
		 * 进行数据的保存处理
		 * @param date 需要进行保存的数据
		 */
		public void addarrays(Node newNode) {
			//将当前的数据和newNode类中的数据进行比较，如果大于0，向右，小于0向左
			if(newNode.date.compareTo((T)this.date)<=0) {
				if(this.left==null) {//现在没有左子树
					this.left=newNode;//进行数据的保存
					newNode.parent=this;//进行父节点的保存
				}else {//继续进行左子树的判断
					this.left.addarrays(newNode);
				}
			}else {
				if(this.right==null) {//右子树为的话
					this.right=newNode;//进行右子树保存
					newNode.parent=this;//进行父节点的保存
				}else {//继续济宁右子树的判断
					this.right.addarrays(newNode);
				}
```

```
    }
}
/**
 * 采用中序遍历的方式进行数据的输出
 */
public void toarraysNode() {
    //左
    if(this.left!=null) {//左子树不为空话
        //继续进行数据的获取
        this.left.toarraysNode();
    }
    //进行数据的获取并且复制(数据的获取)
    BinaryTree.this.returndate[BinaryTree.this.foot++]=this.date;
    //右
    if(this.right!=null) {//右子树不为空
        //继续进行数据的获取
        this.right.toarraysNode();
    }
}
//进行数据的查询
public boolean judgedate(Comparable<T> date) {
    if(date.compareTo((T)this.date)==0) {
        return true;
    }else if(date.compareTo((T)this.date)<0) {//左边有数据
        if(this.left!=null) {
            //递归调用
            return this.left.judgedate(date);
        }else {
            return false;
        }
    }else {
        if(this.right!=null) {
            //递归调用
            return this.right.judgedate(date);
        }else {
            return false;
        }
    }
}

//进行删除数据的获取
public Node getRemoveNode(Comparable<T> date) {
    if(date.compareTo((T)this.date)==0) {
        return this;
    }else if(date.compareTo((T)this.date)<0) {//左边有数据
        if(this.left!=null) {
            //递归调用
            return this.left.getRemoveNode(date);
        }else {
            return null;
        }
    }else {
        if(this.right!=null) {
            //递归调用
```

```
                return this.right.getRemoveNode(date);
            }else {
                return null;
            }
        }
    }

}
    //---------------以下时候BinaryTree中的操作-------------------------
    private Node root;  //二叉树的根节点
    private int count;  // 保存数据的个数
    private Object[] returndate;// 进行数据返回
    private int foot=0;//进行数据返回的角标处理
    /**
     * 进行数据的保存操作
     * @param date 要保存的数据内容
     */
    public void add(Comparable<T> date) {
        if(date==null) {
            throw new NullPointerException("date数据为空");
        }
        //所有的数据是不存在数据的节点操作的，所有需要将数据存在节点中
        Node newNode=new Node(date);
        if(this.root==null) {
            this.root=newNode;//判断节点是否为空，为空的话，将数据赋值给根节点
        }else {
            this.root.addarrays(newNode);//将数据添加的操作给Node进行数据的处理
        }
        this.count++;
    }

    //进行数据的添加以后，就得进行数据的返回操作
    public Object[] getarray() {
        if(this.count==0) {
            return null;
        }
        //进行数据的返回操作
        this.returndate=new Object[this.count];//进行数组的长度保存
        this.foot=0;//角标清零
        this.root.toarraysNode();//将这个获取数据的操作交给Node类进行数据的获取处理
        return this.returndate;
    }

    //进行数据的查询(判断数据是否存在)
    public boolean seacherdate(Comparable<T> date) {
        if(date==null) {
            return false;
        }
        return this.root.judgedate(date);
    }

//进行删除数据的获取，然后进行数据的删除
public void remove(Comparable<T> date) {
    if(date==null && this.root.judgedate(date)) {
```

```
            System.out.println("数据查询不到");
        return ;
    }else {
        //进行判断删除的是否为根节点
    if(date.compareTo((T)this.root.date)==0){
        //进行根节点的删除
    Node move=this.root.right;//移动的角标对象
    while(move.left!=null) {
        move=move.left;//一直进行最最左边的数据进行获取
    }
    move.left=this.root.left;//左边进行赋值
    move.right=this.root.right;//右边赋值
    move.parent.left=null;//原始上一个节点的去除
        this.root=move;
    }else {
        Node removeNode=this.root.getRemoveNode(date);//获取到需要删除的Node类
    if(removeNode.left==null && removeNode.right==null) {
        removeNode.parent.left=null;
        removeNode.parent.right=null;
        //removeNode.parent=null;
    }else if(removeNode.left==null && removeNode.right!=null) {
        //将删除Node类的下一个类中right改成删除类的上一个的左节点，然后将其删除类的right中
父节点改成删除类的上一个当前对象
        removeNode.parent.left=removeNode.right;
        removeNode.left.parent=removeNode.parent;
    }else if(removeNode.left!=null && removeNode.right==null) {
        removeNode.parent.left=removeNode.left;
        removeNode.left.parent=removeNode.parent;
    }else {
        //两者都存在的情况(这种情况是将删除数据替换成右边最左下角的那个对象Node)
        Node moveNode=removeNode.right;
        while(moveNode.left!=null) {
            moveNode=moveNode.left;
        }

        //左替换
        moveNode.left=removeNode.left;
        //右替换
        moveNode.right=removeNode.right;
        //上一个节点的左节点替换
        removeNode.parent.left=moveNode;
        moveNode.parent.left=null; //原始断开左节点
            //父类替换
        moveNode.parent=removeNode.parent;
    }

    }
        this.count--;
    }
        }

    }
    class person implements Comparable<person>{
        private String name;
```

```
    private int age;
    public person(String name,int age) {
        this.name=name;
        this.age=age;
    }
    @Override
    public int compareTo(person per) {
        // TODO Auto-generated method stub
        if(this.age>per.age) {
            return 1;
        }else if (this.age<per.age){
            return -1;
        }
        return 0;

    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return "姓名"+this.name+"年龄"+this.age+"\n";
    }
}

public class  javaDemo01{
    public static void main(String[] args) {
        //进行数据的添加
        BinaryTree<person> bt=new BinaryTree<person>();
        bt.add(new person("小强-80",80));
        bt.add(new person("小强-50",50));
        bt.add(new person("小强-30",30));
        bt.add(new person("小强-60",60));
        bt.add(new person("小强-10",10));
        bt.add(new person("小强-55",55));
        bt.add(new person("小强-70",70));
        bt.add(new person("小强-90",90));
        bt.add(new person("小强-85",85));
        bt.add(new person("小强-95",95));

        //进行数据的获取
        Object[] obj=bt.getarray();
        for(Object temp:obj) {
            System.out.println(temp);
        }
        //进行数据的删除
        //bt.remove(new person("小强10",10));
        //bt.remove(new person("小强30",30));
        //bt.remove(new person("小强80",80));
        bt.remove(new person("小强50",50));
        System.out.println("删除以后进行遍历");
        Object[] obj2=bt.getarray();
        for(Object temp:obj2) {
            System.out.println(temp);
        }
    }
```

```
}
```