102. 二叉树的层序遍历

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new ArrayDeque<>();
        List<List<Integer>> result = new LinkedList<>();
        if(root == null)
            return result;
        queue.add(root);
        while(!queue.isEmpty()){
            int n = queue.size();
            List<Integer> list = new LinkedList<>();
            for(int i = 0; i < n; i++){
                TreeNode p = queue.remove();
                list.add(p.val);
                if(p.left != null)
                    queue.add(p.left);
                if(p.right != null)
                    queue.add(p.right);
            }
            result.add(list);
        }
        return result;
    }
}
```

二叉树的插入

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if(root == null)
```

```
            return null;
        if(root.val == key){    //删除
            if(root.left == null && root.right == null) //如果此结点为叶子结点
                return null;
            if(root.left == null)   //如果此结点没有左节点
                return root.right;
            if(root.right == null)  //如果此结点没有右结点
                return root.left;
            if(root.left != null && root.right != null){    //如果此结点左右结点均有，则必须在左子树找一个最大值或右子树中找一个最小值
                TreeNode node = root.right;
                while(node.left != null){   //找到右子树中最小的那个
                    node = node.left;
                }
                root.val = node.val;    //替代
                root.right = deleteNode(root.right, node.val); //右子树中删除那个最小值
            }

        }else if(root.val > key){   //删除点在左边
            root.left = deleteNode(root.left, key);
        }else if(root.val < key){   //删除点在右边
            root.right = deleteNode(root.right, key);
        }
        return root;
    }
}
```

```
package com.gqp;


import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;


public class UniqueItem {

/**
* 将一个数组中所有重复的元素去掉，保留第一个
*/
public static List<String> uniqueItemKeepFirst(List<String> origin){

//利用set数据的key特性去除重复保留第一个数据,值为1
Set<String> compSet = new HashSet<String>();

//定义返回唯一的去重了数组
List<String> resultList = new ArrayList<String>();

for (String item : origin) {
```

```
//添加新元素
if (!compSet.contains(item)) {
compSet.add(item);
resultList.add(item);
}

}

return resultList;

}

/**
* 将一个数组中所有重复的元素去掉，保留最后一个
*/
public static List<String> uniqueItemKeepLast(List<String> origin){

//利用set数据的key特性去除重复保留第一个数据,值为1
Set<String> compSet = new HashSet<String>();

//定义返回唯一的去重了数组
List<String> resultList = new ArrayList<String>();

for (String item : origin) {

//添加新元素
if (!compSet.contains(item)) {
compSet.add(item);
//最新的总是添加
resultList.add(item);
}else {
//存在替换这个元素
resultList.set(resultList.indexOf(item), item);
}

}

return resultList;

}

/**
* 将一个数组中所有重复的元素去掉，保留第一个
*/
public static List<User> uniqueItemKeepFirstUser(List<User> origin){

//利用set数据的key特性去除重复保留第一个数据,值为1
Set<User> compSet = new HashSet<User>();

//定义返回唯一的去重了数组
List<User> resultList = new ArrayList<User>();

for (User item : origin) {
```

```
//添加新元素
if (!compSet.contains(item)) {
compSet.add(item);
resultList.add(item);
}

}

return resultList;

}


/**
* 将一个数组中所有重复的元素去掉，保留最后一个
*/
public static List<User> uniqueItemKeepLastUser(List<User> origin){

//利用set数据的key特性去除重复保留第一个数据,值为1
Set<User> compSet = new HashSet<User>();

//定义返回唯一的去重了数组
List<User> resultList = new ArrayList<User>();

for (User item : origin) {

//添加新元素
if (!compSet.contains(item)) {
compSet.add(item);
//最新的总是添加
resultList.add(item);
}else {
//存在则替换这个元素
resultList.set(resultList.indexOf(item), item);
}
}

return resultList;

}

}
```

2.简单的一个user类

```
package com.gqp;


public class User {

private String name;
private Long genLong;
```

```
public User(String name,Long genLong){
this.name=name;
this.genLong=genLong;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Long getGenLong() {
return genLong;
}
public void setGenLong(Long genLong) {
this.genLong = genLong;
}
@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + ((name == null) ? 0 : name.hashCode());
return result;
}
@Override
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
final User other = (User) obj;
if (name == null) {
if (other.name != null)
return false;
} else if (!name.equals(other.name))
return false;
return true;
}

}
```

3.测试用user，这样比较耗时

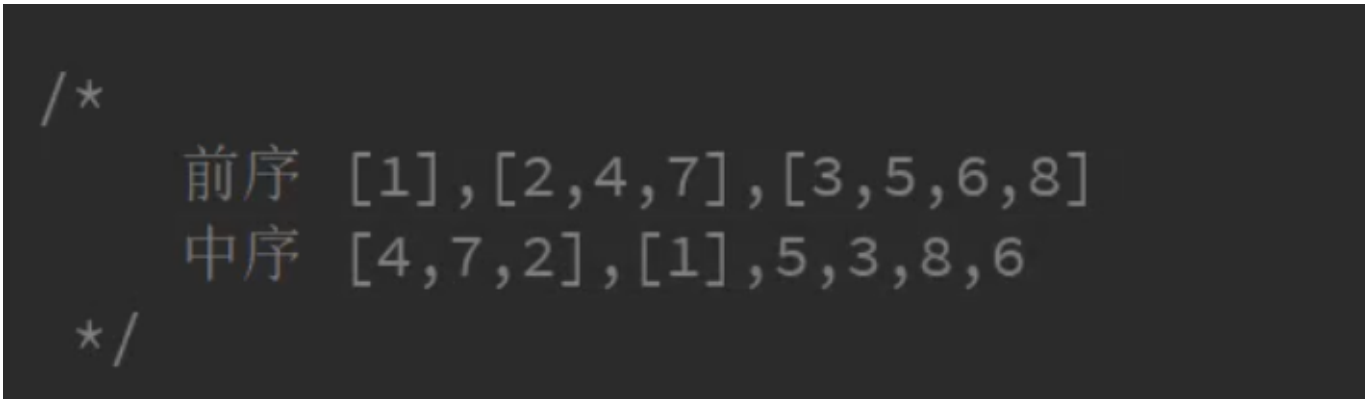
```
package com.gqp;


import java.util.ArrayList;
import java.util.List;




public class TestMain {
```

```
public static void main(String[] args) throws InterruptedException {

//
String[] millios = new String[11];
millios[0]="guo1";
millios[1]="guo2";
millios[2]="guo3";
millios[3]="guo4";
millios[4]="guor";
millios[5]="guo1";
millios[6]="guo4";
millios[7]="guo3";
millios[8]="guo2";
millios[9]="guo1";
millios[10]="nihao";

//产生100W条数据

List<User> origin = new ArrayList<User>();
long gendataStart = System.currentTimeMillis();
for (int i = 0; i < 25000000; i++) {
User u = new User(millios[i%11],System.currentTimeMillis());
origin.add(u);
}
long gendataend = System.currentTimeMillis();
System.out.println("产生100w条数据的时间是:"+(gendataend-gendataStart)+"毫秒");

//遍历
long gendataStart0 = System.currentTimeMillis();
for (int i = 0; i < origin.size(); i++) {
origin.get(i);
}
long gendataend0 = System.currentTimeMillis();
System.out.println("遍历100w条数据的时间是:"+(gendataend0-gendataStart0)+"毫秒");

//剔除重复数据,保留第一条
long gendataStart1 = System.currentTimeMillis();
List<User> fList = UniqueItem.uniqueItemKeepFirstUser(origin);
long gendataend1 = System.currentTimeMillis();
System.out.println("100w条数据去重保留第一条的时间是:"+(gendataend1-
gendataStart1)+"毫秒");

//剔除重复数据,保留第一条
long gendataStart2 = System.currentTimeMillis();
List<User> LList =UniqueItem.uniqueItemKeepLastUser(origin);
long gendataend2 = System.currentTimeMillis();
System.out.println("100w条数据去重保留最后一条的时间是:"+(gendataend2-
gendataStart2)+"毫秒");

//一下是结果的对比
System.out.println(fList.toString());
for (User user : fList) {
```

```
System.out.println(user.getName()+"-"+user.getGenLong());//结果一样，但是可以从数据
产生时的时间可以看出保留第一个

}

System.out.println(LList.toString());
for (User user : LList) {
System.out.println(user.getName()+"-"+user.getGenLong());//结果一样，但是可以从数据
产生时的时间可以看出保留最后一个
}
}


}
```

剑指 Offer 07. 重建二叉树

子烁




```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return helper(preorder, inorder, 0, preorder.length - 1, 0, inorder.length
- 1);
    }

    private TreeNode helper(int[] preorder, int[] inorder, int pre_left, int
pre_right, int in_left, int in_right){
        if(pre_left >= preorder.length || in_left >= inorder.length || pre_left >
pre_right || in_left > in_right)
            return null;
```

```
        //保存前序遍历的第一个值
        int val = preorder[pre_left];

        //在中序遍历中找到前序遍历的第一个元素，并计算长度
        int count = in_left;
        while(inorder[count] != val){
            count++;
        }
        count -= in_left;

        TreeNode node = new TreeNode(val);
        node.left  = helper(preorder, inorder, pre_left + 1, pre_left + count,
in_left, in_left + count - 1);
        node.right = helper(preorder, inorder, pre_left + count + 1, pre_right,
in_left + count + 1, pre_right);
        return node;
    }

}
}
```

剑指 Offer 54. 二叉搜索树的第k大节点

> 给定一棵二叉搜索树，请找出其中第k大的节点。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    /*
        中序遍历的倒序
    */
    int res, k;
    public int kthLargest(TreeNode root, int k) {
        this.k = k;
        drl(root);
        return res;
    }
    private void drl(TreeNode root){
        if(root == null)
            return;

        drl(root.right);

        k--;
```

```
        if(k == 0){
            res = root.val;
            return;
        }

        drl(root.left);
    }
}
```

树的子结构

子烁

> 输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

> B是A的子结构， 即 A中有出现和B相同的结构和节点值。

```
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(A == null || B == null)
            return false;
        if(A.val == B.val && isContain(A, B))
            return true;

        return isSubStructure(A.left, B) || isSubStructure(A.right, B);
    }

    private boolean isContain(TreeNode A, TreeNode B){
        if(A == null && B != null)
            return false;
        if(B == null)
            return true;
        return A.val == B.val && isContain(A.left, B.left) && isContain(A.right,
B.right);
    }
}
```

剑指 Offer 27. 二叉树的镜像

子烁

> 请完成一个函数，输入一个二叉树，该函数输出它的镜像。

```
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        mirror(root);
        return root;
    }
```

```
    private void mirror(TreeNode node){
        if(node == null)
            return;
        TreeNode temp = node.left;
        node.left = node.right;
        node.right = temp;
        mirror(node.left);
        mirror(node.right);
    }
}
```

剑指 Offer 54. 二叉搜索树的第k大节点

> 给定一棵二叉搜索树，请找出其中第k大的节点。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    int res, k;
    public int kthLargest(TreeNode root, int k) {
        this.k = k;
        drl(root);
        return res;
    }
    private void drl(TreeNode root){
        if(root == null)
            return;

        drl(root.right);    //中序遍历的倒序，所以先right

        k--;
        if(k == 0){
            res = root.val;
            return;
        }

        drl(root.left);
    }
}
```