树的序列化

题目

449. 序列化和反序列化二叉搜索树
297. 二叉树的序列化与反序列化
652. 寻找重复的子树
105. 从前序与中序遍历序列构造二叉树
106. 从中序与后序遍历序列构造二叉树

449. 序列化和反序列化二叉搜索树

问题解析

1. 序列化的问题好解决,前序＆中序＆后续＆层次遍历都可以解决这个问题,但是,如何将序列化生成的字符串进行反序列化,恢复成树形结构;
2. 反序列时, 普通的做法是: 1)前序+中序; 2) 中序＆后序 进行反序列化; 3) 将树拓展为满二叉树进行存储; 空以特殊的字符进行标记;
3. 题解中,有一个有利的条件: 二叉搜索树; 二叉树的特点是: 有序！ 但是不一定平衡; 如果是平衡二叉树就可以用满二叉树的做法进行存储,因为不会浪费太多的空间; 但是,二叉搜索树的有序性可以帮助我们在前序遍历中维护中序遍历的信息; 因为二叉搜索树有序;
4. 例如:`[2,1,3,4]` 就知道, `[1]` 是根节点2的左子树,`[3,4]` 是根节点2的右子树; 但是这样同样存在一个问题: `[3,4]` 是3为根节点还是4为根节点? 答: 3为中心节点,4 为右子树; 原因: 先序遍历;
5. 这样,在不浪费空间的情况下,根据二叉搜索树的特性， 完成了二叉搜索树的序列化＆反序列化

考察点:

1. 树的遍历
2. 二叉搜索树的特性
3. 二叉树的序列化＆反序列化
    1. 前序+中序 可以反序列化二叉树
    2. 后续+中序 可以反序列化二叉树
    3. 层次遍历可以很好的完成序列化＆反序列化
4. 二叉搜索树的
    1. 二叉搜索树的中序遍历是递增数组
5. **编码优化**
6. **二分查找**: 二分查找不仅仅可以用在有序列表的查找上,还可以在整体有序的数组上进行查找； 例如， 旋转有序数组的旋转点;

可行思路:

1. 先序遍历, 不使用额外空间
2. 后序遍历, 不使用额外空间
3. 转换为满二叉树(浪费空间,不满足序列化的基本特性)
4. 层次遍历(**通用**的二叉树序列化＆反序列化)

解决思路1: 先序遍历

序列化:

1. 先序遍历,将数据append为一个字符串

```
    public String serialize(TreeNode root) {
        StringBuilder builder = new StringBuilder();
        preOrder(root, builder);
        return builder.toString();
    }

    private void preOrder(TreeNode node, StringBuilder builder){
        // 退出条件
        if(node == null){
            return;
        }
        builder.append(node.val).append(' ');
        preOrder(node.left, builder);
        preOrder(node.right, builder);
    }
```

反序列化

1. 前序序列后的数组 array, 该子树的起始指针 `start = 0`, 结束指针: `end = array.length-1`;
2. 寻找左子树的结束指针; **二分查找**位置 index: 该位置的特征为: `array[index-1] < array[start] < array[index]`; 二分查找的**算法**为: *在整体有序的数组中,寻找第一个比这个数大的元素*;
3. **分治**: 左子树为: `array[start+1:index-1]`; 右子树为: `array[index, end]`
4. 二分查找的算法: 1. 判断是否没有右叶子节点; 如果是,则提前返回; 2. 注意,可能没有左叶子节点,所以, 该位置的特征为: `array[index-1] <= array[start] < array[index]` 取值为 index作为右叶子节点的起点;

```
    public TreeNode deserialize(String data) {
        // 边界条件
        if(data == null || data.isEmpty()){
            return null;
        }
        // 将数据进行拆分
        String[] splited = data.split(" ");
        // 数据的长度
        int[] array = new int[splited.length];
        // 将数据封装进入int数组
        for (int i = 0; i < array.length; i++) {
            array[i] = Integer.parseInt(splited[i]);
        }
        // 开始递归+分治; 这里是分治的思想;
        return dfs(array, 0, array.length-1);
    }

```

```java
    private TreeNode dfs(int[] array, int start, int end){
        // 特殊情况：没有元素
        if(start > end){
            return null;
        }
        TreeNode node = new TreeNode(array[start]);
        // 查找右叶子节点的起始位置
        int rightStart = binarySearchRightBegin(array, start, end);
        // 构造左子树，递归+分治
        node.left = dfs(array, start+1, rightStart-1);
        // 构造右子树，递归+分治
        node.right = dfs(array, rightStart, end);

        return node;
    }

    // 查询右子树起始位置
    private static int binarySearchRightBegin(int[] array, int start, int end){
        // 特殊判断
        if(array[end] <= array[start]){
            return end+1;
        }

        // 开始二分查找
        int left = start+1;
        int right = end;

        while(left < right){
            int mid = (left + right) >> 1;
            if (array[mid] > array[start]){
                // 退出条件1
                if(array[mid-1] <= array[start]){
                    return mid;
                }
                right = mid-1;
            }else{
                left = mid+1;
            }
        }
        // 退出条件2
        return right;
    }
```

解决思路2: 后续遍历

1. 后续遍历当前子树的根节点节点一定是在数组尾部;
2. 二分查找左子树的结束位置
3. 这时,知道左子树的数据范围(左区间&右区间)
4. 先处理右子树，再处理左子树

序列化

正常的后序遍历

```
public String serialize(TreeNode root) {

    if(root == null){
        return "";
    }

    StringBuilder builder = new StringBuilder();
    postOrder(root, builder);

    return builder.substring(1);
}

private void postOrder(TreeNode node, StringBuilder builder){

    if ( node == null ){
        return;
    }

    postOrder(node.left, builder);
    postOrder(node.right, builder);

    builder.append(',').append(node.val);

}
```

反序列化

1. 初始化:将字符串转为整数数组;
2. 初始化子树开始指针&数组的结束指针;
3. 范围是否无数字? **没有,则直接为空**(退出递归)，有则进行处理
4. 处理根节点；(**访问该节点**)

5. **二分查找**，找到左子树的结束位置; (收集信息,切割本问题为**子问题**)
6. 处理左子树，处理右子树; (**drill down**)

```java
public TreeNode deserialize(String data) {
    if(data == null || data.length() == 0){
        return null;
    }
    System.out.println(data);
    String[] strList = data.split(",");
    int[] intData = new int[strList.length];
    for(int i = 0; i < strList.length; i++){
        intData[i] = Integer.parseInt(strList[i]);
    }
    return builderTree(intData,0, strList.length-1);
}

private TreeNode builderTree(int[] data, int start, int end){
    if (start > end){
        return null;
    }

    TreeNode node = new TreeNode(data[end]);
    // 左子树的终点
    int leftEnd = binarySearch(data, start, end);
    node.left = builderTree(data, start, leftEnd);
    node.right = builderTree(data, leftEnd+1, end-1);

    return node;
}

// 后续遍历中,左子树的右区间在什么位置呢?
private int binarySearch(int[] data, int start, int end){
    // 无左子树的情况,注意这里是 >=
    if (data[start] >= data[end]){
       return start -1;
    }
    // 找到位置: 该位置index上, 该值data[index]  < data[end], 该值 data[index+1] >= data[end] (== 等号包含了无右子树的情况)
    int left = start;
    int right = end -1;
    while(left < right){
        int mid = (left + right) >> 1;
        if (data[mid] < data[end]){
            if(data[mid+1] >= data[end]){
                return mid;
```

```
                }
                left = mid +1;
            }else{
                right = mid-1;
            }
        }
        return left;
    }
```

解决思路4: 层次遍历

1. 如果,上一层的节点不为空时,下一层的节点一定都需要记录下该节点的左节点的值+右节点的值
2. 如果为空,则记录null； 但是null无左右节点,所以，不再拓展左右节点

序列化:

1. 用一个队列缓存某一层的非空节点(root)作为起始值
2. 按照1,2将队列中节点的左右节点的值加入 List<Integer>， 并且将其中非空节点加入下一层(上下层的交替可以有多种方式,两个队列打乒乓球, 循环内新建队列， 一个队列，每次进行循环时,用一个临时变量记录下本层的大小)
3. 当队列为空时,层次遍历完成
4. 优化1: 题目中要求字节传输尽可能少,但是，最后一层一定都是为空; 如果是满二叉树,那么就浪费了一倍的空间(极端情况)； 那么将集合中最后部分的null值全部删除
5. 优化2: 在执行第四个步骤之后,数组的大小可能为奇数,也可能为偶数; 那么，**这里需要拓展为奇数**; 原因是: 如果是**偶数**的情况下,某一个节点没有右叶子节点; 所以需要在建立右子树时,需要加入判断; 那么如果在最后append进入一个空值了之后,非空节点的左右子树一定都会存在，减少了O(n)的判断时间
6. ***Leetcode*** 实测: 优化的代码增加内存&代码执行时间,但是,最终的网络传输字节数会减少; 未优化代码:12ms, 40.6 MB 优化后的代码: 17ms, 41MB

未优化的代码

```
    public String serialize(TreeNode root) {
        if (root == null){
            return "";
        }
        Deque<TreeNode> deque = new LinkedList<>();
        StringBuilder builder = new StringBuilder();

        deque.addLast(root);
        while(!deque.isEmpty()){
            // 这里的层次遍历,采用的就是: 标记层的大小,而不用两个队列
            int levelSize = deque.size();
            for(int i = 0; i < levelSize; i++){
                TreeNode node = deque.removeLast();
                if(node == null) {
                    builder.append(",null");
                }else{
                    builder.append(',').append(node.val);
                    deque.addFirst(node.left);
```

```
                    deque.addFirst(node.right);
                }
            }
        }
        return builder.substring(1);
    }
```

优化后的代码

```
    public String serialize(TreeNode root) {
        if (root == null){
            return "";
        }
        LinkedList<Integer> result = new LinkedList<>();
        Deque<TreeNode> deque = new LinkedList<>();

        deque.addLast(root);
        while(!deque.isEmpty()){
           // 这里的层次遍历,采用的就是: 标记层的大小,而不用两个队列
            int levelSize = deque.size();
            for(int i = 0; i < levelSize; i++){
                TreeNode node = deque.removeLast();
                if(node == null) {
                    result.add(null);
                }else{
                    result.add(node.val);
                    deque.addFirst(node.left);
                    deque.addFirst(node.right);
                }
            }
        }
        // 优化
        while(result.peekLast() == null){
            result.removeLast();
        }
        if (result.size()%2 == 0){
            result.add(null);
        }
        // 返回字符串
        StringBuilder builder = new StringBuilder();
        for(Integer value: result){
            builder.append(',').append(value);
        }
```

```
        return builder.substring(1);
    }
```

反序列化

1. **按层** 进行处理
2. **初始化**根节点，作为初始队列; 从待处理列表中,移出根节点
3. 获取层, 将层中的节点依次移出, 从待处理列表中,移出两个元素，作为该节点的左子树/右子树;
4. 将非空的左子树,右子树 放入下一层;
5. 将**下一层作为待处理层**; 转为步骤2
6. 如果, 待**处理列表中**已经无元素,则退出循环;

```

    public TreeNode deserialize(String data) {
        if(data == null || data.length() == 0){
            return null;
        }
        String[] split = data.split(",");
        Deque<TreeNode> deque = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(split[0]));
        deque.addFirst(root);
        levelDeserialize(deque, split, 1);
        return root;
    }

    public void levelDeserialize(Deque<TreeNode> deque, String[] data, int scan){
        while(!deque.isEmpty()){
            int levelSize = deque.size();
            for(int i = 0; i < levelSize; i++){
                TreeNode node = deque.removeLast();
                if(scan == data.length){
                    return;
                }
                node.left = createNode(deque,data[scan++]);
                node.right = createNode(deque, data[scan++]);
            }
        }
    }

    private TreeNode createNode(Deque<TreeNode> deque, String rightStr) {
        if(!"null".equals(rightStr)){
            TreeNode node = new TreeNode(Integer.parseInt(rightStr));
            deque.addFirst(node);
```

```
            return node;
        }
        return null;
    }

```

297.二叉树的序列化与反序列化

知识背景:

1. 遍历的方式: 前/中/后 序遍历 + 层次遍历
2. 中序+前序/后序遍历可以反序列化二叉树
3. 前序+中序实现反序列化的根本原因是: 前序+中序之间的匹配,可以定位根节点的位置, 左子树的区间,右子树的区间
4. 中序: 左子树/根节点/右子树 前序: 根节点/左子树/右子树
5. **分治思想**: 这个大问题我不知道怎么解决, 我将这个问题拆分为若干个子问题, 如果若干个子问题解决了， 那这个大问题也解决了;

问题解析

1. 序列化和反序列化是配套实现; 序列化是遍历所产生的， 相对比较简单,现在的问题是, 如何在一个数组中实现反序列化;
2. 根据 **知识背景**,序列化的本质是: 如何找到**左子树区间,根节点的值,右子树区间**;
3. 寻找的本质找到**标记**;
4. 思考: 前序遍历是怎么做的? 是怎么思考的? 前序遍历的思想是借助了**分治思想**； 我自身并不知道怎么序列化； 那将本题拆为三部分: 根节点怎么序列化, 左子树怎么序列化, 右子树怎么序列化; 根节点怎么序列化？ 有两种情况, 若为空, 无法进行拆解; 不为空， 继续拆解为左子树+右子树； 将左子树，右子树和 根节点的值，进行拼接，就是结果;
5. 思考: 层次遍历是怎么做的? 将**一层作为一个子问题**;
6. 思考: 还有其他的方式进行标记吗？ 用**特殊的符号**，来标记左子树的位置，中间节点，右子树的位置; 例如: (左子树)根节点(右子树) 这也是一种标记方法； 根节点(左子树)(右子树) 也是一种标记方法
7. 回答问题3: 如何寻找标记与如何标记是相互匹配的;
8. 先序遍历思想: 我不知道左子树/右子树的子问题怎么解决, 但是，我不设置任何限制，由 **左子树/右子树**自己解决； 那在反序列中的思想也一致: 创建根节点, 将接力棒(开始扫描的位置)传给左子树, 左子树扫描完后,告诉根节点: 我将接力棒放到哪了，我得了多少分， 你看着办吧~ 这时候，根节点将接力棒的位置传递给右子树，右子树继续; **接力棒就是扫描位置** 这里,返参就有两个: 1. 子树的指针(引用), 子树扫描到的位置;
9. 层次遍历: 层次遍历的**基本单元**是层； 在这样的情况下,需要明确的知道某一层，并且将这一层与下一层之间的关系绑定;

深度优先遍历-前序

关键点:

1. 子树自由拓展; 以#表示为 空值
2. 反序列化时, 注意负数的处理
3. 反序列化时， 返回参数有两个, 下一步待扫描的位置, 以及当前子树

序列化代码

```
    public String serialize(TreeNode root) {
        return serialize(root, new StringBuilder()).substring(1);
    }

```

```
    private StringBuilder serialize(TreeNode node, StringBuilder builder) {
        builder.append(' ');
        if (node != null) {
            builder.append(node.val);
            serialize(node.left, builder);
            serialize(node.right, builder);
        } else {
            builder.append('#');
        }

        return builder;
    }
```

反序列化代码

```
    // 3. 返参包含 待扫描位置+当前构造出来的子树
    public TreeNode deserialize(String data) {
        return dfs(data.toCharArray(),0).node;
    }

    private Info dfs(char[] data, int offset){
        if (data[offset] == '#') {
            // 2的含义是: '# '; #号加上空格
            return new Info(null, offset+2);
        }

        // 设置符号位
        int flag = 1;
        if(data[offset] == '-'){
            flag = -1;
            offset++;
        }

        // 取当前值
        int value = 0;
        while (data[offset] != ' ') {
            value = value * 10 + data[offset++] - '0';
        }
        // 跳过空格,指向左节点开始的位置
        ++offset;

        Info left = dfs(data, offset);
```

```
        Info right = dfs(data, left.nextOffset);

        // 构造返参
        TreeNode node =  new TreeNode(flag * value);
        node.left = left.node;
        node.right = right.node;
        return new Info(node, right.nextOffset);
    }

    private static class Info {
        TreeNode node;
        int nextOffset;

        public Info(TreeNode node, int nextOffset) {
            this.node = node;
            this.nextOffset = nextOffset;
        }
    }
```

层次遍历

```
    public String serialize(TreeNode root) {
        if (root == null){
            return "";
        }
        Deque<TreeNode> deque = new LinkedList<>();
        StringBuilder builder = new StringBuilder();
        // 初始化第一层
        deque.addLast(root);

        while(!deque.isEmpty()){
            // 用levelSize来标识层
            int levelSize = deque.size();
            // 将本层序列化; 空值序列化为 '#', 这只是一个标记
            for(int i = 0; i < levelSize; i++){
                TreeNode node = deque.removeLast();
                if(node == null) {
                    builder.append(",#");
                }else{
                    builder.append(',').append(node.val);
                    deque.addFirst(node.left);
                    deque.addFirst(node.right);
                }
```

```
            }
        }
        return builder.substring(1);
    }
```

反序列化

```
    public TreeNode deserialize(String data) {
        if(data == null || data.isEmpty()){
            return null;
        }
        String[] split = data.split(",");
        Deque<TreeNode> deque = new LinkedList<>();
        // 初始化层
        TreeNode root = createNode(deque, split[0]);
        // 开始层次反序列化
        levelDeserialize(deque, split, 1);
        return root;
    }

    public void levelDeserialize(Deque<TreeNode> deque, String[] data, int scan){

        // 反序列化
        while(!deque.isEmpty()){
            // 获取层大小
            int levelSize = deque.size();
            // 序列化这一层
            for(int i = 0; i < levelSize; i++){
                TreeNode node = deque.removeLast();
                if(scan >= data.length){
                    return;
                }
                //创建,并加入下一层
                node.left = createNode(deque,data[scan++]);
                node.right = createNode(deque, data[scan++]);
            }
        }
    }

    // 创建节点并加入队头
    private TreeNode createNode(Deque<TreeNode> deque, String rightStr) {
        if(!"#".equals(rightStr)){
```

```
36              TreeNode node = new TreeNode(Integer.parseInt(rightStr));
37              deque.addFirst(node);
38              return node;
39          }
40          return null;
41      }
```