【第三十周】字典树（Trie）与双数组字典树（Double-Array-Trie）

1、208. 实现 Trie (前缀树)

1. 插入字符串
2. 我们从字典树的根开始，插入字符串。对于当前字符对应的子节点，有两种情况：
3. 子节点存在。沿着指针移动到子节点，继续处理下一个字符。
4. 子节点不存在。创建一个新的子节点，记录在 children 数组的对应位置上，然后沿着指针移动到子节点，继续搜索下一个字符。
5. 重复以上步骤，直到处理字符串的最后一个字符，然后将当前节点标记为字符串的结尾。
6. 查找前缀
7. 我们从字典树的根开始，查找前缀。对于当前字符对应的子节点，有两种情况：
8. 子节点存在。沿着指针移动到子节点，继续搜索下一个字符。
9. 子节点不存在。说明字典树中不包含该前缀，返回空指针。
10. 重复以上步骤，直到返回空指针或搜索完前缀的最后一个字符。
11. 若搜索到了前缀的末尾，就说明字典树中存在该前缀。此外，若前缀末尾对应节点的 isEnd 为真，则说明字典树中存在该字符串。

```
  var Trie = function() {
    this.children = {};
};

Trie.prototype.insert = function(word) {
    let node = this.children;
    for (const ch of word) {
        if (!node[ch]) {
            node[ch] = {};
        }
        node = node[ch];
    }
    node.isEnd = true;
};

Trie.prototype.searchPrefix = function(prefix) {
    let node = this.children;
    for (const ch of prefix) {
        if (!node[ch]) {
            return false;
        }
        node = node[ch];
    }
    return node;
}

Trie.prototype.search = function(word) {
    const node = this.searchPrefix(word);
    return node !== undefined && node.isEnd !== undefined;
};
```

```
Trie.prototype.startsWith = function(prefix) {
    return this.searchPrefix(prefix);
};
```

2、241. 为运算表达式设计优先级

1. 采用分治
2. 遍历 字符串
3. 遇到操作符，就将左右两边的字符串，分别当作两个表达式

```
/**
 * @param {string} input
 * @return {number[]}
 */
var diffWaysToCompute = function(input) {
    var ret = [];
    for(var i = 0;i < input.length;i++){
        var c = input.charAt(i);
        if(c == '+' || c == '-' || c == '*'){
            var x = diffWaysToCompute( input.substring(0,i));
            var y = diffWaysToCompute( input.substring(i+1));
            for(var j = 0; j < x.length;j++){
                for(var n = 0; n <y.length;n++){
                    switch(c){
                        case '+':
                            ret.push(x[j]+y[n]);
                            break;
                        case '-':
                            ret.push(x[j]-y[n]);
                            break;
                        case '*':
                            ret.push(x[j]*y[n]);
                            break;
                    }

                }
            }
        }
    }
    if(ret.length == 0){
        ret.push(parseInt(input));
    }
    return ret;
};
```

3、987. 二叉树的垂序遍历

1. 从根节点开始，对整棵树进行一次遍历，在遍历的过程中使用数组 nodes 记录下每个节点的行号 row，列号 col 以及值 value。
2. 遍历完成后，对所有的节点进行排序。
3. 对 nodes 进行一次遍历,将行号 row相等的value值放到同一个数组中。

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var verticalTraversal = function (root) {
    const nodes = [];
// 1.从根节点开始，对整棵树进行一次遍历，在遍历的过程中使用数组 nodes 记录下每个节点的行号 row，列号 col 以及值 value。
    dfs(root, 0, 0, nodes);

// 2.遍历完成后，对所有的节点进行排序。
    nodes.sort((a, b) => {
        if (a[0] !== b[0]) {
            return a[0] - b[0];
        } else if (a[1] !== b[1]) {
            return a[1] - b[1];
        } else {
            return a[2] - b[2];
        }
    });


// 3.对 nodes 进行一次遍历,将行号 row相等的value值放到同一个数组中。
    const ret = [];
    let min = -Number.MAX_VALUE;

    for (const x of nodes) {
        if (min !== x[0]) {
            min = x[0];
            ret.push([]);
        }
        ret[ret.length - 1].push(x[2]);
    }

    return ret;

};

// <!-- 创建树遍历函数 -->
var  dfs = function(node, row, col, nodes) {
    if (node === null) {
        return;
    }
```

```
    nodes.push([col, row, node.val]);

    dfs(node.left, row + 1, col - 1, nodes);
    dfs(node.right, row + 1, col + 1, nodes);
}
```

4、133. 克隆图

1. 这道题其实就是让我们对图进行遍历，在遍历过程中拷贝节点。
2. 在递归函数中，对当前节点进行拷贝，再递归地对它的邻接点进行拷贝，返回拷贝后的节点
3. 由于是无向图，我们在遍历过程中需要将节点分为“已遍历”和“未遍历”两类，避免死循环。对于已经遍历过的节点，我们直接返回拷贝好的节点即可(递归出口)。因为题目提到节点值是唯一的，所以这个遍历状态的记录可以用一个简单的哈希表来实现。

```
/**
 * // Definition for a Node.
 * function Node(val, neighbors) {
 *    this.val = val === undefined ? 0 : val;
 *    this.neighbors = neighbors === undefined ? [] : neighbors;
 * };
 */

/**
 * @param {Node} node
 * @return {Node}
 */
var cloneGraph = function(node, visited = {}) {
    if (!node) return
    // 已访问过的节点，直接返回 visited 中的缓存
    if (node.val in visited) return visited[node.val]

    // 拷贝当前节点，记录在 visited 中
    const newNode = new Node(node.val)
    visited[node.val] = newNode

    // 对当前节点的邻接点进行拷贝
    const clonedNeighbors = []
    for (const neighbor of node.neighbors) {
        clonedNeighbors.push(cloneGraph(neighbor, visited))
    }
    newNode.neighbors = clonedNeighbors

    // 返回拷贝的节点
    return newNode
};
```

开课吧