dfs 二叉树的遍历

从上到下打印二叉树

```
class Solution {
public:
    // 题目要求的二叉树的 从上至下 打印（即按层打印），
    // 又称为二叉树的 广度优先搜索（BFS）
    vector<int> levelOrder(TreeNode *root) {
        vector<int> ans{};
        if(root== nullptr){
            return ans;
        }
        queue<TreeNode* > queueTree;
        queueTree.push(root);
        while (!queueTree.empty()){
            TreeNode* node = queueTree.front();
            queueTree.pop();
            ans.push_back(node->val);
            if(node->left!= nullptr){
                queueTree.push(node->left);
            }
            if(node->right!= nullptr){
                queueTree.push(node->right);
            }
        }
        return ans;
    }
};
```

从上到下打印二叉树 II

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(root== nullptr){
            return  vector<vector<int>>{};
        }
        queue<TreeNode* > queueTree;
        queueTree.push(root);
        while (!queueTree.empty()){
            vector<int> tmp;
            for (int i = queueTree.size(); i >0 ; --i) {
                TreeNode* node = queueTree.front();
                queueTree.pop();
                tmp.push_back(node->val);
                if(node->left!= nullptr){
                    queueTree.push(node->left);
                }
                if(node->right!= nullptr){
                    queueTree.push(node->right);
```

```
                }
            }
            ans.push_back(tmp);
        }
        return ans;
    }
};
```

从上到下打印二叉树 III

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode *root) {
        vector<vector<int>> ans;
        if (root == nullptr) {
            return vector<vector<int>>{};
        }
        queue<TreeNode *> queueTree;
        queueTree.push(root);
        while (!queueTree.empty()) {
            vector<int> tmp;
            for (int i = queueTree.size(); i > 0; --i) {
                TreeNode *node = queueTree.front();
                queueTree.pop();
                if(ans.size() % 2 == 0)
                {
                    tmp.push_back(node->val); // 偶数层 -> 队列头部
                }else{
                    tmp.insert(tmp.begin(),node->val);
                }
                if (node->left != nullptr) {
                    queueTree.push(node->left);
                }
                if (node->right != nullptr) {
                    queueTree.push(node->right);
                }
            }
            ans.push_back(tmp);
        }
        return ans;
    }
};
```

并查集

```
//
// Created by yjs on 2022/5/27.
//

#include <bits/stdc++.h>
```

```cpp
using namespace std;


namespace Test1 {
    /* 数组中有负值
     * 未进行优化的并查集
     *  当 parents[i]==i 时说明这是一个集合的根节点
     * */
    class UnionFind {

    private:
        vector<int> parents;
        int unionCount = 0; // 合并次数


    public:
        UnionFind(int capitalSize) {
            parents.resize(capitalSize);
            // 初始化
            for (int i = 0; i < capitalSize; ++i) {
                parents[i] = i;

            }


        }

    public:

        void unionCapital(int x, int y) {
            // 将 y 合并到 x 集合中
            int findX = findCapital(x);
            int findY = findCapital(y);
            if (findX != findY) {

                parents[findY] = findX;
                unionCount++;
            }

        }

        int findCapital(int x) {
            if (parents[x] == x) return x;
            return findCapital(parents[x]);
        }

        int getUnionCount() const {
            return this->unionCount;
        }


    };
}

namespace Test2 {
```

```
/*
 * 对并操作进行优化
 * 按大小求并 将节点少的树合并到节点多的树上面去
 * 但节点数据全为正数时可以省去 size 数组 直接 在parents 存放数据的大小
 *
 * */

class UnionFind {
private:
    vector<int> parent;
    vector<int> size; //  保存树的大小
    int unionCount = 0; // 合并次数


public:
    UnionFind(int capitalSize) {
        // 初始化parents

        parent.resize(capitalSize);
        size.resize(capitalSize);
        for (int i = 0; i < capitalSize; ++i) {
            parent[i] = i;
            size[i] = 1; // 单节点时size=1
        }
    }

public:

    // 按照大小进行合并
    void unionCapital(int x, int y) {
        // 将 y 合并到 x 集合中
        int findX = findCapital(x);
        int findY = findCapital(y);

        if (findX != findY) {

            if (this->size[findY] <= this->size[findX]) {
                // 当y集合的元素个数小于x元素的个数时 将 y 集合合并到 x 集合中
                parent[findY] = findX;
                size[findX] += size[findY];

            } else {
                parent[findX] = findY;
                size[findY] += size[findX];


            }
            unionCount++;
        }
    }

    int findCapital(int x) {
        if (parent[x] == x) return x;
        return findCapital(parent[x]);
    }
```

```
        int getUnionCount() const {
            return this->unionCount;
        }


    };


}


namespace Test3 {
    /*
     * 对并操作进行优化
     * 按执 ( 高度 ) 求并 将节点少的树合并到节点多的树上面去
     * 但节点数据全为正数时可以省去 size 数组 直接 在parents 存放数据的大小
     *
     * */

    class UnionFind {
    private:
        vector<int> parent;
        vector<int> rank; //  保存树的大小
        int unionCount = 0; // 合并次数


    public:
        UnionFind(int capitalSize) {
            // 初始化parents

            parent.resize(capitalSize);
            rank.resize(capitalSize);
            for (int i = 0; i < capitalSize; ++i) {
                parent[i] = i;
                rank[i] = 1; // 单节点时size=1
            }
        }

    public:

        // 按照大小进行合并
        void unionCapital(int x, int y) {
            // 将 y 合并到 x 集合中
            int findX = findCapital(x);
            int findY = findCapital(y);

            if (findX != findY) {

                if (this->rank[findY] <= this->rank[findX]) {
                    // 当y集合的元素个数小于x元素的个数时 将 y 集合合并到 x 集合中
                    parent[findY] = findX;
                } else {
                    parent[findX] = findY;
                }
```

```
                if (rank[findX] == rank[findY] && findX != findY) {
                    rank[findX]++;
                }
                unionCount++;
            }
        }

        int findCapital(int x) {
            if (parent[x] == x) return x;
            return findCapital(parent[x]);
        }

        int getUnionCount() const {
            return this->unionCount;
        }


    };


}


namespace Test4 {
    /*
     * 对find操作进行优化
     * 压缩路径  将子树全部接到根节点上去
     *
     *
     * */

    class UnionFind {
    private:
        vector<int> parent;
        vector<int> rank; //  保存树的大小
        int unionCount = 0; // 合并次数


    public:
        UnionFind(int capitalSize) {
            // 初始化parents

            parent.resize(capitalSize);
            rank.resize(capitalSize);
            for (int i = 0; i < capitalSize; ++i) {
                parent[i] = i;
                rank[i] = 1; // 单节点时size=1
            }
        }

    public:

        // 按照大小进行合并
        void unionCapital(int x, int y) {
            // 将 y 合并到 x 集合中
```

```
            int findX = findCapital(x);
            int findY = findCapital(y);

            if (findX != findY) {

                if (this->rank[findY] <= this->rank[findX]) {
                    // 当y集合的元素个数小于x元素的个数时 将 y 集合合并到 x 集合中
                    parent[findY] = findX;
                } else {
                    parent[findX] = findY;
                }
                if (rank[findX] == rank[findY] && findX != findY) {
                    rank[findX]++;
                }
                unionCount++;
            }
        }

        int findCapital(int x) {
            if (parent[x] == x) return x;
            return parent[x] = findCapital(parent[x]);
        }


        int getUnionCount() const {
            return this->unionCount;
        }


    };


}


namespace Test5 {
    /*
     * 对find操作进行优化
     * 压缩路径  将子树全部接到根节点上去
     * */

    class UnionFind {
    private:
        vector<int> parent;
        int unionCount = 0; // 合并次数


    public:
        UnionFind(int capitalSize) {

            // 初始化parents
            // [-1 0 1 2 3 5 -2 1 2]
            parent.resize(capitalSize);
            for (int i = 0; i < capitalSize; ++i) {
                parent[i] = -1;
```

```
        }
    }

public:

    // 按照大小进行合并
    void unionDis(int x, int y) {
        // 将 y 合并到 x 集合中
        int findX = findCapital(x);
        int findY = findCapital(y);
        if (findX != findY) {
            parent[findY] = findX;
            unionCount++;
        }
    }

    void unionBySize(int x, int y) {
        int findX = findCapital(x);
        int findY = findCapital(y);
        if (findX != findY) {
            if (this->parent[findX] <= this->parent[findY]) {
                // x :  -2  y : -6
                // 当y集合的元素个数小于x元素的个数时 将 y 集合合并到 x 集合中
                parent[findX] += parent[findY];
                parent[findY] = findX;

            } else {
                parent[findY] += parent[findX];
                parent[findX] = findY;

            }
            unionCount++;
        }

    }

    void unionByRank(int x, int y) {
        int findX = findCapital(x);
        int findY = findCapital(y);
        if (findX != findY) {
            if (parent[findX] <= parent[findY]) {
                // 当y集合的元素个数小于x元素的个数时 将 y 集合合并到 x 集合中
                parent[findY] = findX;

            } else {
                parent[findX] = findY;

            }
            // 当俩颗树相等根节点不同时 高度+1
            if (parent[findX] == parent[findY]) {
                parent[findX]--;

            }
            unionCount++;
```

```
            }


        }


        int findCapital(int x) {
            if (parent[x] < 0) return x;
            return parent[x] = findCapital(parent[x]);
        }


        int getUnionCount() const {
            return this->unionCount;
        }


    };


}


int main() {
    return 0;
}
```