CSP-J中，关于树的算法考点

一、树的基础定义与核心性质（必考基础）

1. 核心概念

- 树的定义：无环的连通无向图，n个顶点恰好有n-1条边（CSP-J常用来判断一棵树是否合法）；
- 关键术语：根节点、父节点、子节点、叶子节点（度为1的节点，根节点除外）、深度（根节点深度为1/0，需贴合题目要求）、高度（叶子节点高度为1/0）、子树；
- 常见树结构：普通树、二叉树（重点）、完全二叉树（高频场景，适配数组存储）。

2. 核心性质（CSP-J解题关键）

- 连通性：任意两个顶点之间有且仅有一条路径，无环；
- 边数与顶点数关系：n个顶点必有n-1条边，增减一条边都会破坏树的性质（增边成环，减边不连通）；
- 二叉树特殊性质：非空二叉树中，叶子节点数 = 度为2的节点数 + 1（常考判断题、计算题）；
- 完全二叉树性质：若用数组存储（根节点下标为1），则节点i的左孩子为2i，右孩子为2i+1，父节点为i/2（向下取整）。

3. 考情说明

不直接考定义默写，多结合后续遍历、路径查询等算法考查，或作为题干条件（如“给定一棵树，求xxx”），需熟练判断树的合法性。

二、树的存储方式（适配CSP-J的核心实现）

1. 邻接表（首选，适配普通树、二叉树，兼容图的存储逻辑）

树是特殊的图，可复用图的邻接表存储，需注意避免回溯（记录父节点，不访问父节点），适合顶点数较多的场景（n≤1e4）。

2. 二叉树专属存储

- 数组存储：适配完全二叉树，空间紧凑，访问孩子/父节点高效，CSP-J高频使用（如堆排序基础、二叉树遍历真题）；
- 结构体+指针：适配任意二叉树，代码灵活，但CSP-J中使用较少（需注意内存分配，入门级考点不侧重）。

3. 补充说明

CSP-J中树的存储优先掌握邻接表和完全二叉树的数组存储，足以应对所有树论题。

三、树的核心算法（必考模块，模板固定）

1. 二叉树遍历（高频必考）

核心考点：前序、中序、后序（递归/迭代版）、层序遍历，常考“根据两种遍历序列求二叉树”“输出遍历结果”。

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 二叉树结构体（指针版，适配任意二叉树）
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 1. 前序遍历（根→左→右）- 递归版（简洁，CSP-J首选）
void preOrder(TreeNode* root, vector<int>& res) {
    if (!root) return;
    res.push_back(root->val); // 访问根
    preOrder(root->left, res); // 遍历左子树
    preOrder(root->right, res); // 遍历右子树
}

// 2. 中序遍历（左→根→右）- 递归版
void inOrder(TreeNode* root, vector<int>& res) {
    if (!root) return;
    inOrder(root->left, res);
    res.push_back(root->val);
    inOrder(root->right, res);
}

// 3. 后序遍历（左→右→根）- 递归版
void postOrder(TreeNode* root, vector<int>& res) {
    if (!root) return;
    postOrder(root->left, res);
    postOrder(root->right, res);
```

```
    res.push_back(root->val);
}

// 4. 层序遍历（广度优先，按层输出）- 队列实现
void levelOrder(TreeNode* root, vector<int>& res) {
    if (!root) return;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* cur = q.front();
        q.pop();
        res.push_back(cur->val);
        if (cur->left) q.push(cur->left);
        if (cur->right) q.push(cur->right);
    }
}

// 补充：完全二叉树（数组存储）的遍历（根节点下标1）
vector<int> arrLevelOrder(vector<int>& tree) {
    vector<int> res;
    int n = tree.size() - 1; // 数组下标从1开始，tree[0]无用
    for (int i = 1; i <= n; i++) {
        res.push_back(tree[i]);
    }
    return res;
}

int main() {
    // 示例：构建简单二叉树
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);

    vector<int> pre, in, post, level;
    preOrder(root, pre);
    inOrder(root, in);
    postOrder(root, post);
    levelOrder(root, level);

    // 输出结果
    cout << "前序遍历：";
    for (int x : pre) cout << x << " ";
    cout << endl << "中序遍历：";
    for (int x : in) cout << x << " ";
    cout << endl << "后序遍历：";
    for (int x : post) cout << x << " ";
```

```
    cout << endl << "层序遍历: ";
    for (int x : level) cout << x << " ";
    return 0;
}

```

2. 树的遍历拓展（连通块、深度/高度计算）

树是无环连通图，可复用图的DFS/BFS遍历，核心考查“计算节点深度/高度”“统计子树大小”，是后续算法的基础。

```
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 1e4 + 5;
vector<int> adj[MAXN]; // 邻接表存储树
int depth[MAXN]; // 存储每个节点的深度（根节点深度设为1）
int height[MAXN]; // 存储每个节点的高度
int size_[MAXN]; // 存储每个节点的子树大小（包含自身）

// DFS计算深度、高度、子树大小（root为当前节点，fa为父节点）
void dfs(int root, int fa) {
    depth[root] = depth[fa] + 1; // 深度 = 父节点深度 + 1
    size_[root] = 1; // 初始子树大小为自身
    int max_h = 0;
    for (int v : adj[root]) {
        if (v == fa) continue; // 跳过父节点，避免回溯
        dfs(v, root);
        size_[root] += size_[v]; // 累加子树大小
        max_h = max(max_h, height[v] + 1); // 高度 = 子节点高度最大值 + 1
    }
    height[root] = max_h;
}

int main() {
    int n; // 节点数
    cin >> n;
    // 读入n-1条边（树的边数=节点数-1）
    for (int i = 0; i < n-1; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
```

```
    int root = 1; // 假设根节点为1
    depth[0] = 0; // 父节点（虚拟）深度为0
    dfs(root, 0);

    // 输出结果
    cout << "节点1的深度：" << depth[1] << endl;
    cout << "节点1的高度：" << height[1] << endl;
    cout << "节点1的子树大小：" << size_[1] << endl;
    return 0;
}

```

3. 树的直径（高频考点，入门级）

核心定义：树中任意两个节点之间的最长路径，CSP-J考法固定，用两次DFS/BFS即可求解，步骤简单、模板固定。

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int MAXN = 1e4 + 5;
vector<pair<int, int>> adj[MAXN]; // 邻接表：{邻接节点，边权}（默认无权则边权为1）
int dist[MAXN]; // 存储距离

// BFS求解最长距离（无权树，有权树用Dijkstra），返回最远节点编号
int bfs(int start, int n) {
    fill(dist, dist + n + 1, -1);
    queue<int> q;
    q.push(start);
    dist[start] = 0;
    int max_dist = 0, far_node = start;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto& p : adj[u]) {
            int v = p.first, w = p.second;
            if (dist[v] == -1) {
                dist[v] = dist[u] + w;
                if (dist[v] > max_dist) {
                    max_dist = dist[v];
                    far_node = v;
                }
```

```
                    q.push(v);
                }
            }
        }
        return far_node;
    }

    // 两次BFS求树的直径
    int treeDiameter(int n) {
        int u = bfs(1, n); // 第一次BFS，找离任意节点（如1）最远的节点u
        int v = bfs(u, n); // 第二次BFS，找离u最远的节点v
        return dist[v]; // u到v的距离就是树的直径
    }

    int main() {
        int n;
        cin >> n;
        for (int i = 0; i < n-1; i++) {
            int u, v, w = 1; // 无权树，边权设为1；有权树可输入w
            cin >> u >> v;
            adj[u].push_back({v, w});
            adj[v].push_back({u, w});
        }
        cout << "树的直径：" << treeDiameter(n) << endl;
        return 0;
    }

```

4. 并查集应用（树的连通性、最小生成树入门）

CSP-J中，树的连通性判断、最小生成树（Prim/Kruskal，入门级）常结合并查集考查，复用图论中并查集模板，核心是“树无环、连通”的性质。

备注：最小生成树J组偶考，侧重Kruskal算法（结合并查集），模板可复用图论中并查集代码，无需额外拓展。

四、CSP-J树的算法避坑与备考要点

1. 易错点

- 遍历回溯问题：邻接表存储树时，必须跳过父节点，否则会陷入死循环；
- 深度/高度定义：题目可能约定根节点深度为0或1，需先看题干说明，避免计算错误；
- 二叉树序列题：根据两种遍历求二叉树时，需注意“中序遍历确定左右子树范围”，前序/后序确定根节点。

2. 备考优先级

- 必掌握：二叉树遍历（递归版）、树的DFS/BFS（深度/高度计算）、树的直径；
- 选掌握：并查集结合树的连通性、最小生成树（Kruskal入门）；
- 不考内容：线段树、树状数组（入门级不考）、平衡二叉树、树上倍增（S组考点）。

3. 刷题建议

每掌握一个模板，刷3-5道J组真题/模拟题，侧重“遍历输出”“深度/高度计算”“树的直径”三类基础题，这类题模板固定、易得分。