

作业七

T1

若一棵 m 叉树中，度为1的结点有 N_1 个，度为2的结点有 N_2 个，……，度为 m 的结点有 N_m 个，问该树的叶子结点有多少个？

我们约定题中的度为出度，树的结点数为 v ，边数为 e ，则有关系式：

$$v = e + 1$$

已知度 ≥ 1 的结点数 $\sum_{k=1}^m N_k$ ，则度为 0 的叶子结点数 N_0 为：

$$\begin{aligned} N_0 &= v - \sum_{k=1}^m N_k \\ &= e + 1 - \sum_{k=1}^m N_k \\ &= \sum_{k=1}^m k \cdot N_k + 1 - \sum_{k=1}^m N_k \\ &= \sum_{k=1}^m (k - 1) \cdot N_k + 1 \end{aligned}$$

T2

试找出分别满足下列条件的所有二叉树：（此题在纸上画，不用提交）

- (1) 先序序列和中序序列相同；
- (2) 中序序列和后序序列相同；
- (3) 先序序列和后序序列相同。

这道题一开始可能没什么思路，也许可以随意画几个结点凑数，但是没有一般的结论。但我们从递归的角度思考本题也许会有不一样的发现。

对于一个二叉树，我们将其定义为「根、左子树、右子树」三个部分。则先序遍历是根左右，中序遍历是左根右，后序遍历是左右根。

于是对于上述三种问题，想要某种序列和另外一种序列相同，就是对应的三部分要相同。显然的只有根一定合法，但是如果左右子树同时存在就一定不可能合法，想要合法就只能删除子树，因此可得：

- (1) 先序 = 中序 \rightarrow 根左右 = 左根右 \rightarrow 根#右 = #根右

(2) 中序 = 后序 → 左根右 = 左右根 → 左根# = 左#根

(3) 先序 = 后序 → 根左右 = 左右根 → 根## = ##根

T3

设有168个结点的完全二叉树，请问叶子结点、单分支结点、双分支结点各有多少个？

假设层数从0开始，则当前完全二叉树共有 $\lceil \log_2(N + 1) \rceil = 8$ 层，即 $[0, 7]$ 层中均有结点，其中：

- $[0, 6]$ 层是满二叉树，共 $2^{6+1} - 1 = 127$ 个结点
- 第7层有 $2^7 = 128$ 个位置。从左至右占用了 $168 - 127 = 41$ 个位置

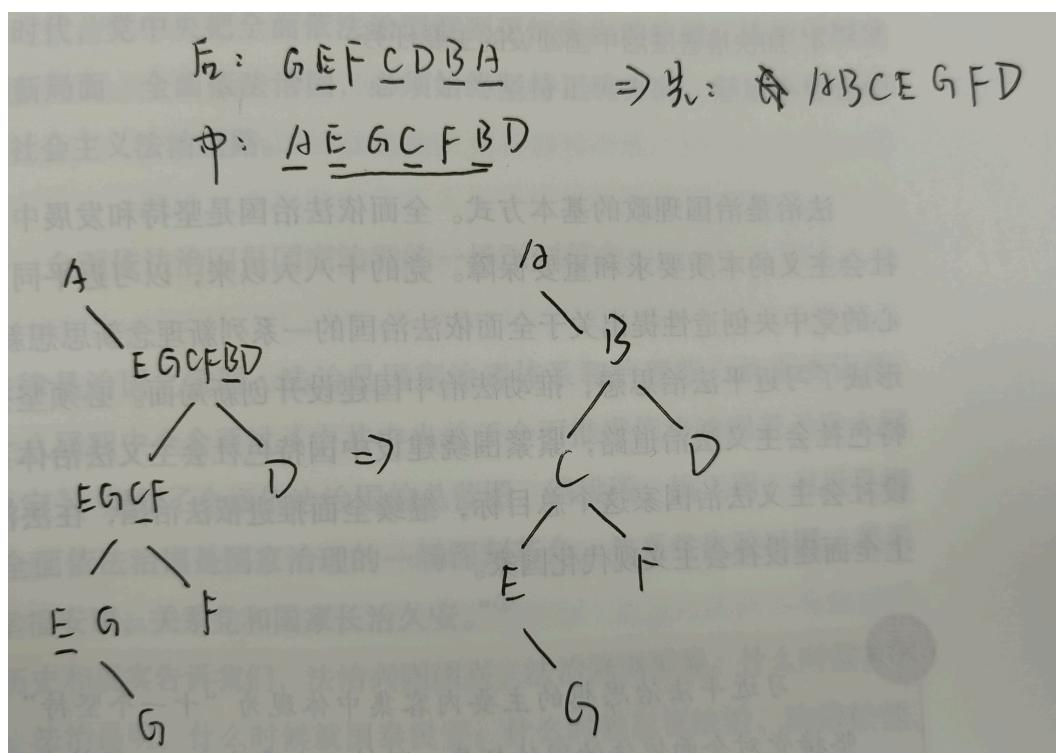
于是可以计算出：

- 双分支结点为 $127 - 64 + \lfloor \frac{41}{2} \rfloor = 83$ 个
- 单分支结点为 $41 \% 2 = 1$ 个
- 叶子结点为 $41 + 64 - \lceil \frac{41}{2} \rceil = 84$ 个

T4

已知某二叉树的后序序列是GEFCDBA，中序序列是AEGCFBD，请画出该二叉树的二叉链表结构图，并写出先序遍历序列。

每次利用后序找到「根」将中序分解为左子树、根、右子树，然后递归分解左右子树即可。



T5

试编写算法求二叉树中双分支节点的个数。

遍历每一个结点判断左右结点是否存在即可，至于如何遍历，可以 dfs 也可以 bfs，其中 dfs 也可以写成递归的形式，这里以 bfs 为例。时间复杂度 $O(n)$

```
1 template<class T>
2 int BinaryTree<T>::countDoubleBranchNode() {
3     int cnt = 0;
4     queue<BinaryTreeNode<T>*> q;
5     q.push(root);
6     while (q.size()) {
7         auto now = q.front();
8         q.pop();
9         cnt += now->lchild != nullptr && now->rchild != nullptr;
10        if (now->lchild) {
11            q.push(now->lchild);
12        }
13        if (now->rchild) {
14            q.push(now->rchild);
15        }
16    }
17    return cnt;
18 }
```

T6

试编写算法求二叉树中各个结点的平衡因子（左右子树高度之差）

回溯法解决即可。时间复杂度 $O(n)$

```

1 template<class T>
2 int BinaryTree<T>::height(BinaryTreeNode<T>* now,
3                             vector<pair<T, int>>& res) {
4     if (!now) {
5         return 0;
6     }
7     int lh = height(now->lchild, res);
8     int rh = height(now->rchild, res);
9     res.push_back({now->data, abs(lh - rh)});
10    return max(lh, rh) + 1;
11 }
```

T7

一棵二叉树以二叉链表来表示，求其指定的某一层 $k(k > 1)$ 上的叶子结点的个数。

遍历时增加参数为当前层数后哈希即可。时间复杂度 $O(n)$

```

1 template<class T>
2 void BinaryTree<T>::dfs(BinaryTreeNode<T>* now, int depth,
3                           unordered_map<int, int>& f) {
4     if (!now) {
5         return;
6     }
7     f[depth]++;
8     dfs(now->lchild, depth + 1, f);
9     dfs(now->rchild, depth + 1, f);
10 }
```

T8

试编写算法输出一棵二叉树中根结点到各个叶子结点的路径。

遍历时记录路径，遇到叶子结点保存完整路径即可，需要在回溯时弹出遍历过的结点。时间复杂度 $O(n)$

```

1 template<class T>
2 void BinaryTree<T>::dfs(BinaryTreeNode<T>* now, vector<T>& path,
3                           vector<vector<T>>& res) {
4     if (!now) {
```

```

5         return;
6     }
7     path.push_back(now->data);
8     if (!now->lchild && !now->rchild) {
9         res.push_back(path);
10        path.pop_back();
11        return;
12    }
13    dfs(now->lchild, path, res);
14    dfs(now->rchild, path, res);
15    path.pop_back();
16 }
```

T9

设计一个算法，求二叉树中两个给定结点的最近公共祖先。

方法一：枚举。我们可以利用 T10 封装好的函数直接求解。即：求出根结点两个指定结点的路径，返回最右端的相同字符即可。每次询问的时间复杂度 $O(n)$

```

1 template<class T>
2 T BinaryTree<T>::getLowestCommonAncestor(T a, T b) {
3     vector<T> path_to_a = getPathFromRootToNode(a);
4     vector<T> path_to_b = getPathFromRootToNode(b);
5     T res;
6     for (int i = 0; i < path_to_a.size(); i++) {
7         if (path_to_a[i] == path_to_b[i]) {
8             res = path_to_a[i];
9         } else {
10             break;
11         }
12     }
13     return res;
14 }
```

方法二：树上倍增。每次询问的时间复杂度 $O(\log n)$ 。OJ: [P3379 【模板】最近公共祖先 \(LCA\)](#)

题意：给定 n 个结点， m 次询问，问两个结点的最近公共祖先

思路：当然，求解 LCA 还有其他众多方法，目前只掌握了树上倍增的思路，相对于上述的对路径中所有结点逐个遍历，优化策略就是二进制遍历。当然前提是提前维护好了 $fa[i][j]$ 数组，表示 i 号点向上跳 2^j 步后到达的结点。接着就是跳跃的过程了，我们首先需要将两个结点按照倍增的思路向上跳到同一个深度，接下来两个结点同时按照倍增的思路向上跳跃，为了确保求出最近的，我们需要确保在跳跃的步调一致的情况下，两者的祖先始终不相同，那么倍增结束后，两者的父结点就是最近公共祖先，即 $fa[x][k]$ 或 $fa[y][k]$

维护 fa 数组：可以发现，对于 $fa[i][j]$ ，我们可以通过递推的方式获得，即 $fa[i][j] = fa[fa[i][j-1]][j-1]$ ，当前结点向上跳跃 2^j 步可以拆分为先向上 2^{j-1} 步，在此基础之上再向上 2^{j-1} 步。于是我们可以采用宽搜的顺序维护 fa 数组。

时间复杂度：

- 维护 fa 数组时，对于树中的每一个结点，我们都要统计向上跳跃 $[0, \lg N]$ 的所有情况，故时间复杂度为 $O(n \log n)$
- 跳跃时，每一次询问中，我们都需要对两个结点进行跳跃，时间复杂度是 $O(\log n)$ ，那么 m 次询问的时间复杂度就是 $O(m \log n)$
- 总时间复杂度就是： $O((n + m) \log n)$

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int N = 5e5 + 10;
6
7 int n, Q, root;
8 vector<int> G[N];
9 int fa[N][20], dep[N];
10 queue<int> q;
11
12 void bfs() {
13     dep[root] = 1;
14     q.push(root);
15     while (q.size()) {
16         int now = q.front();
17         q.pop();
18         for (int ch : G[now]) {
19             if (dep[ch]) {
20                 continue;
21             }
22             dep[ch] = dep[now] + 1;
23             fa[ch][0] = now;
24             q.push(ch);
25         }
26     }
27 }
```

```

23         fa[ch][0] = now;
24         for (int k = 1; k <= 19; k++) {
25             fa[ch][k] = fa[fa[ch][k - 1]][k - 1];
26         }
27         q.push(ch);
28     }
29 }
31
32 int lca(int a, int b) {
33     if (dep[a] < dep[b]) {
34         swap(a, b);
35     }
36     for (int k = 19; k >= 0; k--) {
37         if (dep[fa[a][k]] >= dep[b]) {
38             a = fa[a][k];
39         }
40     }
41     if (a == b) {
42         return a;
43     }
44     for (int k = 19; k >= 0; k--) {
45         if (fa[a][k] != fa[b][k]) {
46             a = fa[a][k], b = fa[b][k];
47         }
48     }
49     return fa[a][0];
50 }
51
52 int main() {
53     cin >> n >> Q >> root;
54     for (int i = 0; i < n - 1; ++i) {
55         int a, b;
56         cin >> a >> b;
57         G[a].push_back(b);
58         G[b].push_back(a);
59     }
60     bfs();
61     while (Q--) {
62         int a, b;
63         cin >> a >> b;
64         cout << lca(a, b) << "\n";

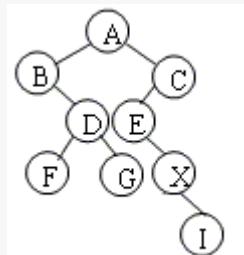
```

```

65     }
66     return 0;
67 }
```

T10

若一棵二叉树中没有数据域值相同的结点，试设计算法打印二叉树中数据域值为x的结点的所有祖先结点的数据域。如果根结点的数据域值为x或不存在数据域值为x的结点，则什么也不打印。例如对下图所示的二叉树，则打印结点序列为A、C、E。



和T8类似，只是标记点为指定的点而非叶子结点。时间复杂度 $O(n)$

```

1 template<class T>
2 void BinaryTree<T>::dfs(BinaryTreeNode<T>* now, vector<T>& path,
3                           vector<T>& res, T target) {
4     if (!now) {
5         return;
6     }
7     path.push_back(now->data);
8     if (now->data == target) {
9         res = path;
10        path.pop_back();
11        return;
12    }
13    dfs(now->lchild, path, res, target);
14    dfs(now->rchild, path, res, target);
15    path.pop_back();
16 }
```

T11

已知二叉树存于二叉链表中，试编写一个算法，判断给定二叉树是否为完全二叉树。

直接进行层序遍历，如果当前不是双分支结点，那么同一层的后面的结点都只能是叶子结点。时间复杂度 $O(n)$

```

1 template<class T>
2 bool BinaryTree<T>::isCompleteBinaryTree() {
3     queue<BinaryTreeNode<T>*> q;
4     q.push(root);
5     while (q.size()) {
6         bool appear = false; // pre appear not double branch node
7         vector<BinaryTreeNode<T*>> level;
8         while (q.size()) {
9             auto now = q.front();
10            q.pop();
11            if (!now) {
12                continue;
13            }
14            level.push_back(now);
15            if (!now->lchild || !now->rchild) {
16                appear = true;
17            }
18            if (appear && (now->lchild || now->rchild)) {
19                return false;
20            }
21        }
22        for (auto node: level) {
23            q.push(node->lchild);
24            q.push(node->rchild);
25        }
26    }
27    return true;
28 }
```

T12

已知二叉树存于二叉链表中，试编写一个算法计算二叉树的宽度，即同一层中结点数的最大值。

遍历统计即可，深搜宽搜均可。时间复杂度 $O(n)$

```

1 template<class T>
2 int BinaryTree<T>::width() {
3     int res = 0;
```

```

4     queue<BinaryTreeNode<T>*> q;
5     q.push(root);
6     while (q.size()) {
7         vector<BinaryTreeNode<T>*> level;
8         while (q.size()) {
9             auto now = q.front();
10            q.pop();
11            if (!now) {
12                continue;
13            }
14            level.push_back(now);
15        }
16        res = max(res, int(level.size()));
17        for (auto node: level) {
18            q.push(node->lchild);
19            q.push(node->rchild);
20        }
21    }
22    return res;
23 }
```

T13

已知二叉树存于二叉链表中，编写一个递归算法，利用叶结点中空的右链指针域rchild，将所有叶结点自左至右链接成一个单链表，算法返回最左叶结点的地址（链头）

我们只需要设置一个头指针 `head` 用来保存最左叶子结点的地址，再设置一个前驱指针 `pre` 用来保存前一个叶子结点的地址，在遍历二叉树的时候链接叶子结点即可。当然，由于我们此时改变了二叉树的结构，会导致原来的析构函数逻辑出错造成内存泄漏。时间复杂度 $O(n)$

```

1 template<class T>
2 void BinaryTree<T>::dfs(BinaryTreeNode<T>* now,
3                             BinaryTreeNode<T>*& pre,
4                             BinaryTreeNode<T>*& head) {
5     if (!now) {
6         return;
7     }
8     if (!now->lchild && !now->rchild) {
9         if (!head) {
10             head = pre = now;
11         } else {
```

```

12         pre->rchild = now;
13         pre = now;
14     }
15     return;
16 }
17 dfs(now->lchild, pre, head);
18 dfs(now->rchild, pre, head);
19 }
```

实验七

T1

实验题 7.1 编写程序，实现二叉树类及若干应用算法。

要求采用二叉链表结构，实现以下功能。

- (1) 构造函数：根据带空指针标记的先序遍历序列构造对象，根据先序和中序遍历序列构造对象。
- (2) 析构函数：释放所有结点空间。
- (3) 先序遍历算法、中序遍历算法、后序遍历算法和层次遍历算法。
- (4) 统计叶子结点、单分支结点、双分支结点的个数。
- (5) 计算二叉树的高度。
- (6) 交换每个结点的左右孩子。
- (7) 输出根结点到每个叶子结点的路径。

构造函数：带空指针标记的先序遍历序列

```

1 template<class T>
2 BinaryTreeNode<T>* BinaryTree<T>::createWithPreTaged(string& pre_of_tag,
3                                         int& i) {
4     T e = pre_of_tag[i++];
5     if (e == '#') {
6         return nullptr;
7     }
8     BinaryTreeNode<T>* now = new BinaryTreeNode<T>(e);
9     now->lchild = createWithPreTaged(pre_of_tag, i);
10    now->rchild = createWithPreTaged(pre_of_tag, i);
11    return now;
12 }
```

构造函数：带空指针标记的后序遍历序列

```

1 template<class T>
2 BinaryTreeNode<T>* BinaryTree<T>::createWithPostTaged(string& post_of_tag,
3                                         int& i) {
4     T e = post_of_tag[i--];
5     if (e == '#') {
6         return nullptr;
7     }
8     BinaryTreeNode<T>* now = new BinaryTreeNode<T>(e);
9     now->rchild = createWithPostTaged(post_of_tag, i);
10    now->lchild = createWithPostTaged(post_of_tag, i);
11    return now;
12 }

```

构造函数：用先序和中序序列。构造的逻辑仍然是递归，我们根据先序序列构造根结点，并在中序序列中找到对应的结点将先序序列划分为「根、左子树、右子树」三个部分，将中序序列划分为「左子树、根、右子树」三个部分，进而递归构造左右子树。当然，我们假设树中结点的值各不相同，否则无法唯一确定左右子树。为了**简化逻辑**，构造时传递的先序序列和中序序列均为重新赋值出来的参数，这导致代码在递归时的时间开销增大，因为需要不断的拷贝构造字符串。从效率角度的最佳实践是传递原始序列的引用附加指针来划定当前子树对应序列的区间。

```

1 template<class T>
2 BinaryTreeNode<T>* BinaryTree<T>::createWithPreMid(string pre,
3                                         string mid) {
4     if (pre.size() == 0) {
5         return nullptr;
6     }
7     BinaryTreeNode<T>* now_root = new BinaryTreeNode<T>(pre[0]);
8     string mid_left, mid_right;
9     for (int i = 0; i < mid.size(); i++) {
10        if (mid[i] == pre[0]) {
11            mid_left = mid.substr(0, i);
12            mid_right = mid.substr(i + 1);
13            break;
14        }
15    }
16    int len_left = mid_left.size();
17    string pre_left = pre.substr(1, len_left);
18    string pre_right = pre.substr(len_left + 1);
19    now_root->lchild = createWithPreMid(pre_left, mid_left);
20    now_root->rchild = createWithPreMid(pre_right, mid_right);
21    return now_root;

```

22 }

构造函数：用后序和中序序列

```

1 template<class T>
2 BinaryTreeNode<T>* BinaryTree<T>::createWithMidPost(string mid,
3                                         string post) {
4     if (post.size() == 0) {
5         return nullptr;
6     }
7     BinaryTreeNode<T>* now_root = new BinaryTreeNode<T>(post.back());
8     string mid_left, mid_right;
9     for (int i = 0; i < mid.size(); i++) {
10        if (mid[i] == post.back()) {
11            mid_left = mid.substr(0, i);
12            mid_right = mid.substr(i + 1);
13            break;
14        }
15    }
16    int len_left = mid_left.size();
17    int len_right = mid_right.size();
18    string post_left = post.substr(0, len_left);
19    string post_right = post.substr(len_left, len_right);
20    now_root->lchild = createWithMidPost(mid_left, post_left);
21    now_root->rchild = createWithMidPost(mid_right, post_right);
22    return now_root;
23 }
```

析构函数：

```

1 template<class T>
2 void BinaryTree<T>::decrease(BinaryTreeNode<T>* now) {
3     if (!now) return;
4     decrease(now->lchild);
5     decrease(now->rchild);
6     delete now;
7 }
```

遍历：以前序遍历为例

```

1 template<class T>
2 void BinaryTree<T>::prePrint(BinaryTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     cout << now->data;
7     prePrint(now->lchild);
8     prePrint(now->rchild);
9 }
```

计算二叉树的高度:

```

1 template<class T>
2 int BinaryTree<T>::height(BinaryTreeNode<T>* now) {
3     if (!now) {
4         return 0;
5     }
6     return max(height(now->lchild), height(now->rchild)) + 1;
7 }
```

交换左右子树:

```

1 template<class T>
2 void BinaryTree<T>::dfs(BinaryTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     swap(now->lchild, now->rchild);
7     dfs(now->lchild);
8     dfs(now->rchild);
9 }
```

T2

实验题 7.2 编写程序，实现树类及若干应用算法。

要求采用孩子兄弟表示法，实现以下功能。

- (1) 构造函数：根据序偶集合构造对象。
- (2) 析构函数：释放所有结点空间。
- (3) 先根、后根遍历算法。
- (4) 计算每个结点的度。
- (5) 计算树的高度。
- (6) 输出根结点到每个叶子结点的路径。

构造函数：对于每一条边，先从根开始 find 到当前边的双亲节点 parent，然后将孩子结点插入到合适的位置。

```

1 template<class T>
2 ChildSiblingTree<T>::ChildSiblingTree(vector<pair<T, T>>& edges) {
3     if (edges.empty()) {
4         root = nullptr;
5         return;
6     }
7     root = new ChildSiblingTreeNode<T>(edges[0].first);
8     for (auto [u, v]: edges) {
9         ChildSiblingTreeNode<T>* child = new ChildSiblingTreeNode<T>(v);
10        ChildSiblingTreeNode<T>* parent = find(root, u);
11        if (!parent->first_child) {
12            parent->first_child = child;
13        } else {
14            parent = parent->first_child;
15            while (parent->next_sibling) {
16                parent = parent->next_sibling;
17            }
18            parent->next_sibling = child;
19        }
20    }
21 }
22
23 template<class T>
24 ChildSiblingTreeNode<T>*
25 ChildSiblingTree<T>::find(ChildSiblingTreeNode<T>* now, T e) {
26     if (!now) {
27         return nullptr;
28     }

```

```

29     if (now->data == e) {
30         return now;
31     }
32     ChildSiblingTreeNode<T>* l = find(now->first_child, e);
33     if (l) {
34         return l;
35     } else {
36         return find(now->next_sibling, e);
37     }
38 }
```

析构函数:

```

1 template<class T>
2 void ChildSiblingTree<T>::decrease(ChildSiblingTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     decrease(now->first_child);
7     decrease(now->next_sibling);
8     delete now;
9 }
```

先根遍历：先遍历根，再遍历所有孩子。也就是二叉树中的「根、左、右」的逻辑。

```

1 template<class T>
2 void ChildSiblingTree<T>::prePrint(ChildSiblingTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     cout << "data: " << now->data << " degree: " << now->degree << "\n";
7     prePrint(now->first_child);
8     prePrint(now->next_sibling);
9 }
```

后根遍历：先遍历所有孩子，再遍历根。也就是二叉树中的「左、根、右」的逻辑。

```

1 template<class T>
2 void ChildSiblingTree<T>::postPrint(ChildTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     postPrint(now->first_child);
7     cout << "data: " << now->data << " degree: " << now->degree << "\n";
8     postPrint(now->next_sibling);
9 }
```

计算每个结点的度：对于每一个结点都访问左孩子和其所有的右兄弟。时间复杂度 $O(n^2)$

```

1 template<class T>
2 void ChildSiblingTree<T>::countDegree(ChildTreeNode<T>* now) {
3     if (!now) {
4         return;
5     }
6     if (now->first_child) {
7         now->degree++;
8         ChildTreeNode<T>* t = now->first_child;
9         while (t->next_sibling) {
10             now->degree++;
11             t = t->next_sibling;
12         }
13     } else {
14         now->degree = 0;
15     }
16     countDegree(now->first_child);
17     countDegree(now->next_sibling);
18 }
```

计算树高：

```

1 template<class T>
2 int ChildSiblingTree<T>::height(ChildTreeNode<T>* now, int depth) {
3     if (!now) {
4         return depth;
5     }
6     return max(height(now->first_child, depth + 1),
7                 height(now->next_sibling, depth));
8 }
```

输出根到每个叶子结点的路径:

```

1 template<class T>
2 void ChildSiblingTree<T>::getPathFromRootToLeaf(
3     ChildSiblingTreeNode<T>* now, vector<T>& path,
4     vector<vector<T>>& res) {
5     if (!now) {
6         return;
7     }
8     path.push_back(now->data);
9     if (!now->first_child) {
10        res.push_back(path);
11    } else {
12        getPathFromRootToLeaf(now->first_child, path, res);
13    }
14    path.pop_back();
15    getPathFromRootToLeaf(now->next_sibling, path, res);
16 }
```

T3

实验题 7.3 编写程序，实现 Huffman 编码。

要求实现以下功能：

- (1) 输入字符串，统计各字符出现的频率，根据各个字符出现的频率创建 Huffman 树。
- (2) 输出该字符串中各字符的 Huffman 编码，以及该字符串的编码。
- (3) 输入相关编码串进行译码并输出。

哈夫曼树由于结点数量已知为 $2n - 1$ ，因此可以用静态数组作为存储结构，每个结点存储：左孩子、右孩子、双亲、权重、数据、相对地址，共六个域。

构造函数：

```

1 HuffmanTree::HuffmanTree(string& info) {
2     // store the frequency of each character
3     unordered_map<char, int> dict;
4     for (auto c: info) {
5         dict[c]++;
6     }
7     n = dict.size();
8     tree.resize(2 * n - 1);
9
10    // init the forest of n trees
```

```

11     priority_queue<HuffmanNode> q;
12     int idx = 0;
13     for (auto [c, freq]: dict) {
14         tree[idx] = HuffmanNode(c, freq, -1, -1, -1, idx);
15         q.push(tree[idx++]);
16     }
17
18     // create n-1 internal nodes
19     for (int i = n; i < 2 * n - 1; i++) {
20         HuffmanNode l_node = q.top();
21         q.pop();
22         HuffmanNode r_node = q.top();
23         q.pop();
24         tree[l_node.idx].parent = tree[r_node.idx].parent = i;
25         tree[i] = HuffmanNode(' ', l_node.weight + r_node.weight,
26                               -1, l_node.idx, r_node.idx, i);
27         q.push(tree[i]);
28     }
29 }
```

编码：从每一个叶子结点开始向根寻找编码。

```

1 pair<unordered_map<char, string>, string> HuffmanTree::encode(string& source) {
2     // traverse the tree to get the code of each character
3     unordered_map<char, string> dict;
4     for (int i = 0; i < n; i++) {
5         string code;
6         int now = i;
7         int pa = tree[i].parent;
8         while (pa != -1) {
9             if (tree[pa].lchild == now) {
10                 code = "0" + code;
11             } else {
12                 code = "1" + code;
13             }
14             now = pa;
15             pa = tree[pa].parent;
16         }
17         dict[tree[i].data] = code;
18     }
19
20     // encode the source
```

```

21     string res;
22     for (auto c: source) {
23         if (dict.find(c) == dict.end()) {
24             dict[c] = "|unknown character|";
25         }
26         res += dict[c];
27     }
28
29     return {dict, res};
30 }
```

译码：

```

1  string HuffmanTree::decode(string& secret) {
2      string res;
3      int root = tree.size() - 1;
4      int i = 0;
5      while (i < secret.size()) {
6          int now = root;
7          while (tree[now].lchild != -1 && tree[now].rchild != -1) {
8              if (secret[i] == '1') {
9                  now = tree[now].rchild;
10 } else if (secret[i] == '0') {
11                  now = tree[now].lchild;
12 } else {
13                 return "|invalid secret|";
14             }
15             i++;
16         }
17         res += tree[now].data;
18     }
19     return res;
20 }
```