



南开大学
Nankai University

数据结构实验报告 6

队列和树

姓 名： 南亚宏

学 号： 2311788

目录

一、	题目一	2
1.1	题目表述	2
1.2	代码 1 的测试用例：	3
1.3	思路：	3
1.4	代码：	3
1.5	测试用例截图	7
二、	题目二	8
2.1	题目表述	8
2.2	代码 2 的测试用例：	8
2.3	思路：	9
2.4	代码：	9
2.5	测试用例截图	11
三、	题目三	13
3.1	题目表述	13
3.2	代码 3 的测试用例：	13
3.3	思路：	13
3.4	代码：	15
3.5	测试用例截图	22

2025 年 11 月 19 日

一、 题目一

1.1 题目表述

所谓双端队列（double-ended queue, deque），就是在列表的两端都可以插入和删除数据。因此它允许的操作有 Create、IsEmpty、IsFull、Left、Right、AddLeft、AddRight、DeleteLeft、DeleteRight。使用循环数组方式实现双端队列，要求实现上述操作，并实现一个 Print 输出操作，能将队列由左至右的次序输出于一行，元素间用空格间隔。队列元素类型设为整型。

输入：input.txt，给出一个操作序列，可能是 Create、Print 之外的任何操作，需要的情况下，会给出参数。最后以关键字“End”结束，例如：

```
AddLeft 1
AddLeft 2
DeleteRight
IsFull
DeleteLeft
IsEmpty
AddRight 3
AddLeft 2
AddRight 1
End
```

输出：程序开始执行时，队列设置为空，按输入顺序执行操作，每个操作执行完后，将结果输出于一行。对于错误命令，输出“WRONG”。对 IsEmpty 和 IsFull 命令，视情况输出“Yes”或“No”。对 Left 和 Right 命令，若队列空，输出“EMPTY”，否则输出对应队列元素。对 Add 命令，若队列满，输出“FULL”，否则调用 Print，输出队列所有元素。对 Del 命令，若队列空，输出“EMPTY”，否则输出所有元素。元素间用空格间隔，最后一个元素后不能有空格。最后输出一个回车。

例如，对上例，应输出：

```
-----
1
2 1
2
No
```

Yes

3

2 3

2 3 1

1.2 代码 1 的测试用例:

默认队列容量为 3:

AddLeft 5, 输出→ 队列: [5]

AddRight 10, 输出→ 队列: [5, 10]

AddLeft 3, 输出→ 队列: [3, 5, 10]

Left→ 输出左端元素 3

Right → 输出右端元素 10

DeleteLeft, 输出 → 队列: [5, 10]

DeleteRight, 输出 → 队列: [5]

IsEmpty → 队列非空

1.3 思路:

用循环数组模拟双端队列:

维护左指针 L、右指针 R 和计数 cnt, 所有插入/删除都在模运算下循环移动指针; 输入指令驱动对应操作, 每次增删后即时 Print 队列现状, 直至读到 End 为止。

1.4 代码:

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 3;           // 默认容量为 3

struct Deque {
    int a[MAXN];
```

```

int L, R;                // L 指向真实左端, R 指向真实右端
int cnt;                 // 当前元素个数

Deque() { Create(); }

void Create() {
    L = 0;
    R = -1;              // 初始置为空
    cnt = 0;
}

bool IsEmpty() const { return cnt == 0; }
bool IsFull()  const { return cnt == MAXN; }

int Left()  const { return IsEmpty() ? -1 : a[L]; }
int Right() const { return IsEmpty() ? -1 : a[R]; }

bool AddLeft(int x) {
    if (IsFull()) return false;
    L = (L - 1 + MAXN) % MAXN;
    a[L] = x;
    ++cnt;
    if (cnt == 1) R = L; // 第一个元素
    return true;
}

bool AddRight(int x) {
    if (IsFull()) return false;
    R = (R + 1) % MAXN;
    a[R] = x;
    ++cnt;
    if (cnt == 1) L = R;
    return true;
}

bool DeleteLeft() {
    if (IsEmpty()) return false;
    L = (L + 1) % MAXN;
    --cnt;
    return true;
}

```

```

bool DeleteRight() {
    if (IsEmpty()) return false;
    R = (R - 1 + MAXN) % MAXN;
    --cnt;
    return true;
}

// 按题意从左到右输出，元素间空格，行末无空格
void Print() const {
    if (IsEmpty()) {
        cout << '\n';
        return;
    }
    for (int i = 0, p = L; i < cnt; ++i, p = (p + 1) %
MAXN) {
        if (i) cout << ' ';
        cout << a[p];
    }
    cout << '\n';
}

};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    ifstream fin("input.txt");
    if (!fin) {
        cerr << "无法打开 input.txt\n";
        return 0;
    }

    Deque dq;
    string op;
    while (fin >> op) {
        if (op == "End") break;

        if (op == "Create") {
            dq.Create();
            continue;           // Create 无输出
        }
    }

```

```

if (op == "IsEmpty") {
    cout << (dq.IsEmpty() ? "Yes\n" : "No\n");
}
else if (op == "IsFull") {
    cout << (dq.IsFull() ? "Yes\n" : "No\n");
}
else if (op == "Left") {
    if (dq.IsEmpty()) cout << "EMPTY\n";
    else cout << dq.Left() << '\n';
}
else if (op == "Right") {
    if (dq.IsEmpty()) cout << "EMPTY\n";
    else cout << dq.Right() << '\n';
}
else if (op == "AddLeft") {
    int x; fin >> x;
    if (dq.IsFull()) cout << "FULL\n";
    else {
        dq.AddLeft(x);
        dq.Print();
    }
}
else if (op == "AddRight") {
    int x; fin >> x;
    if (dq.IsFull()) cout << "FULL\n";
    else {
        dq.AddRight(x);
        dq.Print();
    }
}
else if (op == "DeleteLeft") {
    if (dq.IsEmpty()) cout << "EMPTY\n";
    else {
        dq.DeleteLeft();
        dq.Print();
    }
}
else if (op == "DeleteRight") {
    if (dq.IsEmpty()) cout << "EMPTY\n";
    else {
        dq.DeleteRight();
        dq.Print();
    }
}

```

```

    }
}
else if (op == "Print") {
    dq.Print();
}
else {                                     // 非法指令
    cout << "WRONG\n";
}
}
return 0;
}

```

1.5 测试用例截图

```

nwo.1 / > input.txt
1  AddLeft 5
2  AddRight 10
3  AddLeft 3
4  Left
5  Right
6  DeleteLeft
7  DeleteRight
8  IsEmpty
9  End

```



二、 题目二

2.1 题目表述

在 input.txt 输入一个中缀表达式，构造表达式树，以文本方式输出树结构。

输入：例如，输入 $a+b+c*(d+e)$

输出：以缩进表示二叉树的层次，左——根、右——叶、上——右子树、下——左子树

```

-----
              e
            +
          d
        *
      c
+
    b
+
  a
-----
  
```

提示：以什么样的顺序对树进行遍历可以容易地输出为这种形式？标准顺序显然是不行的。不同层次对应不同缩进如何实现？可以为递归函数设定一个参数表示层次（缩进量），递归调用时加以改变即可。另外，如何将 input.txt 转换为二叉树结构，除了链接实现的二叉树结构外，可能还需要一些辅助结构。

2.2 代码 2 的测试用例：

1.a 输出：a

2. $a*(b+c)$ 输出构造成由字符和空格表示的树结构如：

```

      c
    +
    b
*
a
  
```

3. $(a+b)*(c-d)/e$ 输出：略

4. $a+(b-(c+d)*e)$ 输出：略

2.3 思路:

用“调度场算法”把中缀转后缀，用后缀表达式构造表达式二叉树（每个内部节点是运算符，叶节点是操作数）。按“右子树在上，左子树在下”的约定，用递归方式输出缩进文本树。

2.4 代码:

```
#include <bits/stdc++.h>
using namespace std;

// 二叉树节点
struct Node {
    char op;           // 运算符或操作数
    Node *left = nullptr;
    Node *right = nullptr;
    Node(char c): op(c) {}
};

// 中缀转后缀
int prec(char c) {
    if (c == '+' || c == '-') return 1;
    if (c == '*' || c == '/') return 2;
    return 0;
}

vector<char> infixToPostfix(const string& s) {
    vector<char> out;
    stack<char> st;
    for (char c : s) {
        if (isspace(c)) continue;
        if (isalpha(c)) { out.push_back(c); continue; }
        if (c == '(') { st.push(c); continue; }
        if (c == ')') {
            while (!st.empty() && st.top() != '(') {
                out.push_back(st.top()); st.pop();
            }
            if (!st.empty()) st.pop();
            continue;
        }
        // 运算符
        while (!st.empty() && prec(st.top()) >= prec(c)) {
```

```

        out.push_back(st.top()); st.pop();
    }
    st.push(c);
}
while (!st.empty()) { out.push_back(st.top()); st.pop(); }
return out;
}

```

```

Node* buildTree(const vector<char>& post) {
    stack<Node*> st;
    for (char c : post) {
        if (isalpha(c)) {
            st.push(new Node(c));
        } else { // 运算符
            Node* R = st.top(); st.pop(); // 右操作数
            Node* L = st.top(); st.pop(); // 左操作数
            Node* p = new Node(c);
            p->left = L;
            p->right = R;
            st.push(p);
        }
    }
    return st.empty() ? nullptr : st.top();
}

```

```

// 缩进打印
const int IND = 4; // 每层缩进空格数
void printTree(Node* root, int depth = 0) {
    if (!root) return;
    // 右子树
    printTree(root->right, depth + 1);
    // 当前节点
    cout << string(depth * IND, ' ') << root->op << '\n';
    // 左子树
    printTree(root->left, depth + 1);
}

```

```

int main() {
    ifstream fin("input.txt");
    if (!fin) { cerr << "无法打开 input.txt\n"; return 0; }
    string line;
}

```

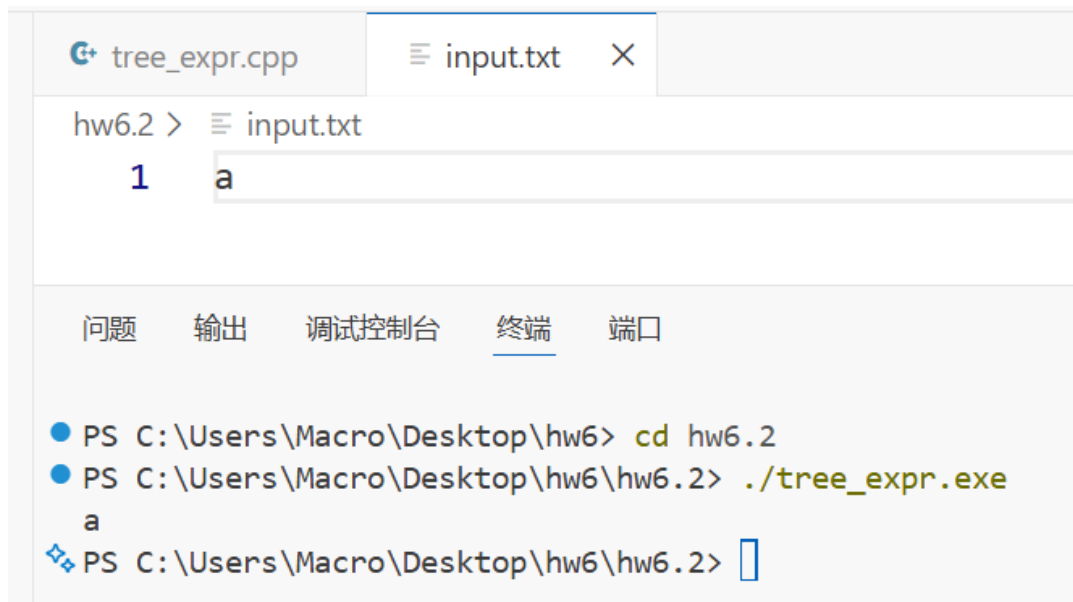
```
getline(fin, line);
fin.close();

auto post = infixToPostfix(line);
Node* root = buildTree(post);
if (!root) { cout << "空表达式\n"; return 0; }

printTree(root);

// 简单内存释放
function<void(Node*)> del = [&](Node* p){
    if (!p) return;
    del(p->left);
    del(p->right);
    delete p;
};
del(root);
return 0;
}
```

2.5 测试用例截图



tree_expr.cpp input.txt X

hw6.2 > input.txt

1 a*(b+c)

问题 输出 调试控制台 终端 端口

- PS C:\Users\Macro\Desktop\hw6> cd hw6.2
- PS C:\Users\Macro\Desktop\hw6\hw6.2> ./tree_expr.exe

```
a
  c
  +
  b
 *
a
```

- PS C:\Users\Macro\Desktop\hw6\hw6.2> ./tree_expr.exe

```
PS C:\Users\Macro\Desktop\hw6\hw6.2>
```

tree_expr.cpp input.txt X

hw6.2 > input.txt

1 (a+b)*(c-d)/e

问题 输出 调试控制台 终端 端口

- PS C:\Users\Macro\Desktop\hw6> cd hw6.2
- PS C:\Users\Macro\Desktop\hw6\hw6.2> ./tree_expr.exe

```
e
/
  d
  -
  c
 *
  b
  +
  a
```

- PS C:\Users\Macro\Desktop\hw6\hw6.2>

```
tree_expr.cpp  input.txt ×
hw6.2 > input.txt
1 a+(b-(c+d)*e)

问题 输出 调试控制台 终端 端口
PS C:\Users\Macro\Desktop\hw6> cd hw6.2
PS C:\Users\Macro\Desktop\hw6\hw6.2> ./tree_expr.exe
      e
     *
      d
     +
      c
     -
      b
     +
      a
PS C:\Users\Macro\Desktop\hw6\hw6.2>
```

三、 题目三

3.1 题目表述

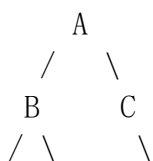
编写二叉树类的成员函数，分别实现以下功能：

- ① 统计二叉树的叶节点的数目
- ② 交换二叉树中所有节点的左右子树
- ③ 按层次顺序遍历二叉树：首先访问根节点，然后是它的两个孩子节点，然后是孙子节点，依此类推
- ④ 求二叉树的宽度，即同一层次上最多的节点数

要求以任意可行的方式输入一棵二叉树，程序依次显示上述各项处理的结果。

3.2 代码 3 的测试用例:

分别输出叶节点数目、层次遍历、各层结点数、交换左右子树后书的结构、交换后的层次遍历、交换后的叶子节点（不是数目，从左到右）、树的宽度



D E F
同样输出案例 1 的内容

```

    A
  /
 B
/
C
/
D

```

3.3 思路：

节点结构采用模板类 `template <typename T>` 设计，支持任意可打印类型（如字符、整数），提高代码复用性；每个节点包含 3 个成员：数据 `data`、左子节点指针 `left`、右子节点指针 `right`，通过构造函数初始化。

二叉树类私有成员根节点指针 `root` 控制树的访问入口，通过“公有接口 + 私有递归辅助函数”分离逻辑——公有接口对外提供调用，私有辅助函数实现递归遍历（避免暴露内部节点指针）；析构函数中通过递归 `destroyTree` 释放所有节点，避免内存泄漏。

txt 文件中按层次顺序输入节点，# 表示空节点。

若首节点为 # 或文件为空，构建空树；否则初始化根节点，用队列 `queue` 存储待处理节点（广度优先遍历思想）；遍历 `nodes` 向量，按“左子节点→右子节点”的顺序，为队列中的节点分配子节点（非 # 则创建节点并入队，# 则跳过），确保树结构与输入一致。

统计叶节点数目：递归遍历，以节点为空作为终止条件，若当前节点的左右子树均为空，则判定为叶节点并计数；否则递归累加左、右子树的叶节点数量，统计整棵树的叶节点总数（分治）。

交换所有节点的左右子树：基于深度优先递归实现，节点为空时返回；非空节点先交换自身的左右子指针，再递归处理交换后的左、右子树，确保从根节点到所有子节点的左右子树均完成交换，不遗漏。

层次顺序遍历：广度优先遍历，先将根节点入队；循环中每次取出当前层所有节点（以队列大小确定层节点数），记录节点数据并将非空左右子节点依次入队，同时收集各层节点数。

求树宽：依托层次遍历的层节点统计逻辑，遍历过程中记录每层的节点数，通过实时比较更新最大节点数，遍历结束后该最大值即为二叉树的宽度。

3.4 代码:

```

#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <fstream>
#include <string>
using namespace std;

// 二叉树节点结构（模板类，支持任意可打印类型）
template <typename T>
struct TreeNode {
    T data;
    TreeNode<T>* left;
    TreeNode<T>* right;
    TreeNode(T val) : data(val), left(nullptr), right(nullptr)
{}
};

// 二叉树类
template <typename T>
class BinaryTree {
private:
    TreeNode<T>* root; // 根节点

    // ④ 递归统计叶节点数目（辅助函数）
    int countLeavesHelper(TreeNode<T>* node) {
        if (node == nullptr) return 0;
        // 叶节点判定：左右子树均为空
        if (node->left == nullptr && node->right == nullptr)
return 1;
        return countLeavesHelper(node->left) +
countLeavesHelper(node->right);
    }

    // ⑤ 递归交换左右子树（辅助函数）
    void swapSubtreesHelper(TreeNode<T>* node) {
        if (node == nullptr) return;
        // 交换当前节点的左右子树
        swap(node->left, node->right);
        // 递归处理左右子树
    }
};

```

```

        swapSubtreesHelper(node->left);
        swapSubtreesHelper(node->right);
    }

    // 释放二叉树内存（避免内存泄漏）
    void destroyTree(TreeNode<T>* node) {
        if (node == nullptr) return;
        destroyTree(node->left);
        destroyTree(node->right);
        delete node;
    }

public:
    // 构造函数（初始化空树）
    BinaryTree() : root(nullptr) {}

    // 析构函数（释放内存）
    ~BinaryTree() {
        destroyTree(root);
    }

    // 从文件读取二叉树（按层次顺序，'#'表示空节点）
    bool buildTreeFromFile(const string& filename) {
        ifstream infile(filename);
        if (!infile.is_open()) {
            cerr << "错误: 无法打开文件 " << filename << endl;
            return false;
        }

        vector<T> nodes;
        T val;
        // 读取文件中所有节点（空格/换行分隔），'#'表示空节点
        while (infile >> val) {
            nodes.push_back(val);
        }
        infile.close();

        // 构建二叉树
        if (nodes.empty() || nodes[0] == '#') {
            root = nullptr;
            return true;
        }
    }

```

```

// 根节点初始化
root = new TreeNode<T>(nodes[0]);
queue<TreeNode<T>*> q;
q.push(root);
int idx = 1; // 当前处理的节点索引

while (!q.empty() && idx < nodes.size()) {
    TreeNode<T>* curr = q.front();
    q.pop();

    // 处理左子节点
    if (nodes[idx] != '#') {
        curr->left = new TreeNode<T>(nodes[idx]);
        q.push(curr->left);
    }
    idx++;

    // 处理右子节点（需判断是否还有剩余节点）
    if (idx < nodes.size() && nodes[idx] != '#') {
        curr->right = new TreeNode<T>(nodes[idx]);
        q.push(curr->right);
    }
    idx++;
}

return true;
}

// ① 统计叶节点数目（对外接口）
int countLeaves() {
    return countLeavesHelper(root);
}

// ② 交换所有节点的左右子树（对外接口）
void swapAllSubtrees() {
    swapSubtreesHelper(root);
}

// ③ 层次顺序遍历（返回遍历结果和各层节点数）
pair<vector<T>, vector<int>> levelOrderTraversal() {
    vector<T> traversal;    // 层次遍历结果

```

```

vector<int> levelSizes; // 各层节点数
if (root == nullptr) return {traversal, levelSizes};

queue<TreeNode<T>*> q;
q.push(root);

while (!q.empty()) {
    int levelSize = q.size(); // 当前层节点数
    levelSizes.push_back(levelSize);

    // 遍历当前层所有节点
    for (int i = 0; i < levelSize; i++) {
        TreeNode<T>* curr = q.front();
        q.pop();
        traversal.push_back(curr->data);

        // 左子节点入队
        if (curr->left != nullptr) q.push(curr->left);
        // 右子节点入队
        if (curr->right != nullptr) q.push(curr->right);
    }
}
return {traversal, levelSizes};
}

// ④ 求二叉树的宽度（同一层次最多节点数）
int getTreeWidth() {
    if (root == nullptr) return 0;

    queue<TreeNode<T>*> q;
    q.push(root);
    int maxWidth = 0;

    while (!q.empty()) {
        int levelSize = q.size();
        maxWidth = max(maxWidth, levelSize); // 更新最大宽度

        // 下一层节点入队
        for (int i = 0; i < levelSize; i++) {
            TreeNode<T>* curr = q.front();
            q.pop();
            if (curr->left != nullptr) q.push(curr->left);

```

```

        if (curr->right != nullptr) q.push(curr->right);
    }
}
return maxWidth;
}

// 交换后获取叶节点（从左到右，层次顺序）
vector<T> getLeavesAfterSwap() {
    vector<T> leaves;
    if (root == nullptr) return leaves;

    queue<TreeNode<T>*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode<T>* curr = q.front();
        q.pop();

        // 叶节点判定
        if (curr->left == nullptr && curr->right ==
nullptr) {
            leaves.push_back(curr->data);
        } else {
            // 非叶节点，左右子节点入队（保证左到右顺序）
            if (curr->left != nullptr) q.push(curr->left);
            if (curr->right != nullptr) q.push(curr->right);
        }
    }
    return leaves;
}

// 辅助函数：打印向量（遍历结果、叶节点等）
static void printVector(const vector<T>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        if (i > 0) cout << " ";
        cout << vec[i];
    }
    cout << endl;
}

// 辅助函数：打印各层节点数
static void printLevelSizes(const vector<int>& sizes) {

```

```

        cout << "各层节点数: ";
        for (size_t i = 0; i < sizes.size(); i++) {
            if (i > 0) cout << " ";
            cout << sizes[i];
        }
        cout << endl;
    }
};

// 测试执行函数（统一处理输出格式）
template <typename T>
void runTestFromFile(const string& filename, const string&
testName) {
    cout << "===== " << testName << "
===== " << endl;
    BinaryTree<T> tree;

    // 从文件构建二叉树
    if (!tree.buildTreeFromFile(filename)) {
        cerr << "测试失败: 无法构建二叉树" << endl;
        return;
    }

    // 1. 输出叶节点数目
    cout << "1. 叶节点数目: " << tree.countLeaves() << endl;

    // 2. 层次遍历 + 各层节点数
    auto [traversal, levelSizes] = tree.levelOrderTraversal();
    cout << "2. 层次遍历: ";
    BinaryTree<T>::printVector(traversal);
    BinaryTree<T>::printLevelSizes(levelSizes);

    // 3. 树的宽度
    cout << "3. 树的宽度: " << tree.getTreeWidth() << endl;

    // 4. 交换左右子树
    tree.swapAllSubtrees();
    cout << "4. 交换左右子树后处理结果: " << endl;

    // 5. 交换后的层次遍历（即交换后的树结构）
    auto [swapTraversal, swapLevelSizes] =
tree.levelOrderTraversal();

```

```

cout << "    交换后的层次遍历（树结构）： ";
BinaryTree<T>::printVector(swapTraversal);

// 6. 交换后的叶节点（从左到右）
vector<T> swapLeaves = tree.getLeavesAfterSwap();
cout << "    交换后的叶节点（从左到右）： ";
BinaryTree<T>::printVector(swapLeaves);

// 7. 交换后的叶节点数目
cout << "    交换后的叶节点数目： " << tree.countLeaves() <<
endl;

    cout <<
    "=====
    << endl << endl;
}

int main() {
    // ----- 测试用例 1 -----
    -----
    // 树结构：
    //      A
    //     / \
    //    B  C
    //   / \  \
    //  D  E  F
    // input1.txt 内容： A B C D E # F # # # # #
    string file1 = "input1.txt";
    runTestFromFile<char>(file1, "测试用例 1");

    // ----- 测试用例 2 -----
    -----
    // 树结构：
    //    A
    //   /
    //  B
    // /
    // C
    // /
    // D
    // input2.txt 内容： A B # C # D # #
    string file2 = "input2.txt";

```

```
runTestFromFile<char>(file2, "测试用例 2");

return 0;
}
```

3.5 测试用例截图

```
● PS C:\Users\Macro\Desktop\hw6\hw6.3> ./BinaryTree.exe
===== 测试用例1 =====
1. 叶节点数目: 3
2. 层次遍历: A B C D E F
   各层节点数: 1 2 3
3. 树的宽度: 3
4. 交换左右子树后处理结果:
   交换后的层次遍历 (树结构): A C B F E D
   交换后的叶节点 (从左到右): F E D
   交换后的叶节点数目: 3
=====

===== 测试用例2 =====
1. 叶节点数目: 1
2. 层次遍历: A B C D
   各层节点数: 1 1 1 1
3. 树的宽度: 1
4. 交换左右子树后处理结果:
   交换后的层次遍历 (树结构): A B C D
   交换后的叶节点 (从左到右): D
   交换后的叶节点数目: 1
=====

❖ PS C:\Users\Macro\Desktop\hw6\hw6.3> 
```