

数据结构汇总

Edited: Rium

一、线性表

1. 链表

链表结构体

```
struct node{
    int data;
    node *next;
    node(int x) : data(x), next(nullptr){}
};

//可用于构建循环表
```

创建节点

```
//构建链表
node *build(int x){
    return new node(x);
}

//调用
node *root = build(1);
head->next = build(2);
```

查找

```
int Search(node* head, int target) {
    int i = 0;
    for (node* p = head; p != nullptr; p = p->next){
        if (p->data == target) return i;
        i++;
    }
    return -1; //没找到
}
```

插入节点

```
//指定位置插入
node* Insert(node* head, int pos, int value) {
    if (pos == 0) {
        node* newNode = new node(value);
        newNode->next = head;
        return newNode;
    } //插入头节点

    node* prev = head;
    for (int i = 0; prev != nullptr && i < pos - 1; i++) {
        prev = prev->next;
    }

    if (prev == nullptr) return head; // 位置无效

    node* newNode = new node(value);
    newNode->next = prev->next;
    prev->next = newNode;

    return head;
}

//调用
node *root = build(10);
root = insert(head, 1, 20);
```

删除节点

```
// 删除第一个值为target的节点
node* Delete(node* head, int target) {
    // 如果链表为空
    if (head == nullptr) return nullptr;

    // 如果要删除的是头节点, temp用于防治内存泄漏。
    if (head->data == target) {
        node* temp = head;
        head = head->next;
        delete temp;
        return head;
    }

    // 查找要删除的节点
    node* current = head;
    while (current->next != nullptr && current->next->data != target) {
        current = current->next;
    }
```

```

// 找到目标节点
if (current->next != nullptr) {
    node* temp = current->next;
    current->next = current->next->next;
    delete temp;
}

return head;
}

```

求长度

```

int get_length(node* head){
    int len = 0;
    while(head){
        len++;
        head = head->next;
    }
    return len;
}

```

双链表

```

struct node{
    int data;
    node *next;
    node *prior;/访问前一个节点
    node(int x) : data(x), next(nullptr), prior(nullptr){}
};

```

2. 应用

最大连续子段和

```

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

int main() {
    int n; cin >> n;
    int max = 0, sum = 0;
    for (int i = 0; i < n; i++) {
        int tmp; cin >> tmp;
        sum += tmp;
        if (sum > max) max = sum;

        //出现负数, 清零, 重新记录新的子段和
        if (sum < 0) sum = 0;
    }
}

```

```
    }
    if (max >= 0) cout << max << endl;
    else cout << 0 << endl;
    return 0;
}
```

输入样例

```
6
-2 11 -4 13 -5 -2
```

输出样例(11, -4, 13)

```
20
```

二、栈与队列

1. 栈

(1) 出栈序列

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

int main() {
    string v, k, ans, rst;
    stack<char> stk;
    cin >> v >> k;

    int j = 0;
    for(int i = 0; i < v.length(); i++){
        stk.push(v[i]);
        ans += "P";
        //先入栈
        while(!stk.empty() && stk.top() == k[j]){
            ans += "O";
            rst.push_back(stk.top())
            stk.pop();
            j++;
        }
        //一定要用while, 实现连续出栈
    }
}
```

```

if(rst == k){
    cout << "Right" << endl;
    cout << ans;
}else{
    cout << "Wrong" << endl;
    while(!stk.empty()){
        cout << stk.top();
        stk.pop();
    }
}
return 0;
}

```

输入样例

(12345是入栈顺序, 34521是判断目标)

```

12345
34521

```

输出样例

("P"是入栈, "O"是出栈)

```

Right
PPPOPOPOOO

```

(2) 列出所有可能的出栈序列

```

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

struct State {
    int idx;           // 下一个要入栈的数字
    stack<int> stk;   // 当前栈状态
    vector<int> rst;  // 当前结果序列
    int choice;        // 0:未选择, 1:入栈, 2:出栈
};

vector<vector<int>> ans;

void solveIterative(int N) {
    stack<State> state_stack;

    // 初始状态
    State init;
    init.idx = 1;
    init.choice = 0;
    state_stack.push(init);
}

```

```

while (!state_stack.empty()) {
    State &cur = state_stack.top();

    if (cur.idx > N && cur.stk.empty()) {
        // 找到一个完整序列
        ans.push_back(cur.rst);
        state_stack.pop();
        continue;
    }

    if (cur.choice == 0) {
        // 第一次访问这个状态，尝试入栈
        cur.choice = 1;
        if (cur.idx <= N) {
            State next = cur;
            next.stk.push(next.idx);
            next.idx++;
            next.choice = 0;
            state_stack.push(next);
        }
    }
    else if (cur.choice == 1) {
        // 已经尝试过入栈，现在尝试出栈
        cur.choice = 2;
        if (!cur.stk.empty()) {
            State next = cur;
            int tmp = next.stk.top();
            next.stk.pop();
            next.rst.push_back(tmp);
            next.choice = 0;
            state_stack.push(next);
        }
    }
    else {
        // 两种选择都尝试过了，回溯
        state_stack.pop();
    }
}

int main() {
    int N;
    cin >> N;
    solveIterative(N);
    // 输出结果
    for (auto &seq : ans) {
        for (int i = 0; i < seq.size(); i++) {
            cout << seq[i];
            if (i < seq.size() - 1) cout << " ";
        }
        cout << endl;
    }
}

```

```

    }
    cout << ans.size() << endl;
    return 0;
}

```

输入样例

```
3
```

输出样例

```

3 2 1
2 3 1
2 1 3
1 3 2
1 2 3
5(总数量)

```

(3) 中缀转后缀表达式

```

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

// 获取运算符优先级
int pri(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0; // 包括 '('
}

// 判断是否为运算符
bool isOp(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

vector<char> solve(const string &str){
    stack<char> ops; // 储存运算符
    vector<char> rst;

    for(int i = 0; i < str.length(); i++){
        char c = str[i];

        // 整数
        if(isdigit(c)){
            rst.push_back(c);
        } // 默认数字为一位整数

        // 左括号
        if(c == '(') ops.push('(');
    }
}
```

```

//右括号
if(c == ')') {
    while (!ops.empty() && ops.top() != '(') {
        rst.push_back(ops.top());
        ops.pop();
    }
    // 弹出左括号
    if(!ops.empty()) ops.pop();
}

//运算符
if(isOp(c)){
    while (!ops.empty() &&
           pri(ops.top()) >= pri(c)) {
        rst.push_back(ops.top());
        ops.pop();
    }
    // 当前运算符入栈
    ops.push(c);
}
}

while (!ops.empty()) {
    rst.push_back(ops.top());
    ops.pop();
}

return rst;
}

int main() {
    string expr;
    cin >> expr;

    vector<char> result = solve(expr);
    for(char c : result) {
        cout << c << " ";
    }
    cout << endl;

    return 0;
}

```

测试样例

2+3*(7-4)+8/4

输出样例

2 3 7 4 - * + 8 4 / +

2. 队列

二叉树层序遍历

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

struct tree{
    int data;
    tree *left;
    tree *right;
    tree() : data(0), left(nullptr), right(nullptr){}
    tree(int x) : data(x), left(nullptr),right(nullptr){}
};

//层序遍历打印二叉树
void Print(tree* root) {
    if (root == nullptr) {
        return;
    }

    queue<tree*> q;
    q.push(root);

    while (!q.empty()) {
        tree* current = q.front();
        q.pop();

        //先打印节点值
        cout << current->data << " ";

        //再分别把左右子节点放入队列
        if (current->left != nullptr) {
            q.push(current->left);
        }
        if (current->right != nullptr) {
            q.push(current->right);
        }
    }
    cout << endl;
}
```

三、排序

1. 插入排序

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

// 插入排序函数
void insertionSort(vector<int>& arr) {
    int n = arr.size();

    // 从第二个元素开始 (第一个元素默认已排序)
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // 当前要插入的元素
        int j = i - 1; // 从已排序部分的最后一个元素开始比较

        // 将所有比key大的元素向右移动
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // 插入key到正确位置
        arr[j + 1] = key;
    }

    /*可以在输出一遍排序结果*/
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    insertionSort(arr);

    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}
```

排序过程 (" | "左侧已排好序)

```
i = 1, 排34
34 64 | 25 12 22 11 90
i = 2, 排25
25 34 64 | 12 22 11 90
i = 3, 排12
12 25 34 64 | 22 11 90
i = 4, 排22
12 22 25 34 64 | 11 90
i = 5, 排11
11 12 22 25 34 64 | 90
i = 6, 排90
11 12 22 25 34 64 90
```

2. 选择排序

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

void selectionSort(vector<int>& arr) {
    int n = arr.size();

    // 每次循环找到未排序部分的最小元素
    for (int i = 0; i < n - 1; i++) {
        // 假设当前i位置是最小值的位置
        int minIndex = i;

        // 在i+1到n-1之间找真正的最小值
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j; // 更新最小值位置
            }
        }

        // 将找到的最小值与i位置交换
        if (minIndex != i) {
            swap(arr[i], arr[minIndex]);
        }
    }
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};

    selectionSort(arr);
    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}
```

排序过程

```
epoch1: 64<->11
11 | 25 12 22 64
epoch2: 25<->12
11 12 | 25 22 64
epoch3: 25<->22
11 12 22 | 25 64
epoch4: 25<->25
11 12 22 25 | 64
```

3. 快速排序

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

// Lomuto分区方案（较容易理解）
int partition(vector<int>& arr, int low, int high) {
    // 选择最后一个元素作为基准
    int pivot = arr[high];
    int i = low - 1; // 小于基准的区域的边界

    for (int j = low; j < high; j++) {
        // 如果当前元素小于等于基准
        if (arr[j] <= pivot) {
            i++; // 扩大小于基准的区域
            swap(arr[i], arr[j]);
        }
    }

    // 将基准放到正确位置
    swap(arr[i + 1], arr[high]);
    return i + 1; // 返回基准的最终位置
}

// 快速排序主函数
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        // pi是基准的正确位置
        int pi = partition(arr, low, high);

        // 递归排序左右两部分
        quickSort(arr, low, pi - 1); // 左半部分
        quickSort(arr, pi + 1, high); // 右半部分
    }
}
```

```

// 包装函数
void quickSort(vector<int>& arr) {
    quickSort(arr, 0, arr.size() - 1);
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr);
    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}

```

排序过程

```

epoch1: 5为基准
1 | 7 8 9 10 5
1 | 5 | 8 9 10 7 (5的位置已确定)
epoch2: 左边以1为基准, 右边以7为基准
1 5 | 8 9 10 7
1 5 7 | 9 10 8 (1, 7的位置确定)
...
可以看出, 序列本身有序时, lomuto分块法反而效率降低。

```

4. 归并排序

```

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

// 合并两个有序数组
void merge(vector<int>& arr, int left, int mid, int right) {

    //拷贝划分
    vector<int> leftArr(arr.begin() + left, arr.begin() + mid + 1);
    vector<int> rightArr(arr.begin() + mid + 1, arr.begin() + right + 1);
    int n1 = leftArr.size(), n2 = rightArr.size();

    // 合并临时数组回原数组
    int i = 0;          // 左数组索引
    int j = 0;          // 右数组索引
    int k = left;       // 合并后数组索引

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }

    // 处理剩余元素
    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];
}

```

```

    }

}

// 拷贝左数组剩余元素
while (i < n1) arr[k++] = leftArr[i++];

// 拷贝右数组剩余元素
while (j < n2) arr[k++] = rightArr[j++];
}

// 归并排序主函数
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // 递归排序左右两半
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // 合并已排序的两半
        merge(arr, left, mid, right);
    }
}

// 包装函数
void mergeSort(vector<int>& arr) {
    if (arr.size() > 1) {
        mergeSort(arr, 0, arr.size() - 1);
    }
}

int main() {
    vector<int> arr = {12, 21, 13, 25, 16, 27};
    mergeSort(arr);

    for (int num : arr) cout << num << " ";
    cout << endl;
    return 0;
}

```

排序过程

划分

```

12 21 13 | 25 16 27
12 21 || 13 | 25 16 || 27
12 ||| 21 || 13 | 25 ||| 16 || 27

```

归并

```

12 21 || 13 | 16 25 || 27
12 13 21 | 16 25 27
12 13 16 21 25 27

```

5. 希尔排序

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

void ShellSort(vector<int>& arr) {
    int n = arr.size();

    // 初始增量设为数组长度的一半，逐步缩小
    for (int gap = n / 2; gap > 0; gap /= 2) {
        // 对每个子序列进行插入排序
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;

            // 对间隔为gap的元素进行插入排序
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

int main() {
    vector<int> arr = {8, 9, 1, 7, 2, 3, 5, 4, 6, 0};
    ShellSort(arr);

    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}
```

排序过程

```
{8, 9, 1, 7, 2, 3, 5, 4, 6, 0}len = 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]index

(插入排序)
(1)gap = 5
{8, 3}(index = 0, 5)->{3, 8}
{9, 5}->{5, 9}
{1, 4}->{1, 4}
{7, 6}->{6, 7}
{2, 0}->{0, 2}
放回原数组:
{3, 5, 1, 6, 0, 8, 9, 4, 7, 2}
```

```

(2)gap = 2 ((int)5/2 = 2)
{3, 1, 0, 9, 7}(index = 0, 2, 4, 6, 8)->{0, 1, 3, 7, 9}
{5, 6, 8, 4, 2}->{2, 4, 5, 6, 8}
放回原数组:
{0, 2, 1, 4, 3, 5, 7, 6, 9, 8}

(3)gap = 1 ((int)2/2 = 1)
{0, 2, 1, 4, 3, 5, 7, 6, 9, 8}->
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

```

6. 堆排序

纯手写

```

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

// 调整堆 (维护堆性质)
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;           // 初始化最大为根节点
    int left = 2 * i + 1;      // 左子节点
    int right = 2 * i + 2;     // 右子节点

    // 如果左子节点存在且大于根
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // 如果右子节点存在且大于当前最大
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // 如果最大值不是根
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // 递归调整被影响的子树
        heapify(arr, n, largest);
    }
}

// 堆排序主函数
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // 1. 构建最大堆 (从最后一个非叶子节点开始)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // 2. 一个个从堆顶取出元素

```

```

    for (int i = n - 1; i > 0; i--) {
        // 将当前堆顶（最大值）与末尾交换
        swap(arr[0], arr[i]);

        // 调整剩余元素，使其保持堆性质
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    heapSort(arr);

    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}

```

借助STL

```

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

// 方法1：直接使用STL的堆排序功能
void heapSortSTL(vector<int>& arr) {
    // 1. 将vector转换为最大堆
    make_heap(arr.begin(), arr.end());

    // 2. 使用sort_heap进行堆排序（升序）
    sort_heap(arr.begin(), arr.end());
}

// 方法2：分步实现（更容易理解）
void heapSortSTLStepByStep(vector<int>& arr) {
    // 步骤1：构建最大堆
    make_heap(arr.begin(), arr.end());

    // 步骤2：逐个取出最大元素放到末尾
    for (auto it = arr.end(); it != arr.begin(); --it) {
        // 将当前堆顶（最大值）移动到已排序部分的开始位置
        pop_heap(arr.begin(), it);
    }
}

int main() {
    vector<int> arr = {3, 7, 2, 1, 9, 5, 4, 8, 6};
    heapSortSTL(arr);

    for (int num : arr) cout << num << " ";
    cout << endl;
}

```

```
    return 0;
}
```

*算法稳定性

- 排序算法的稳定性指的是在排序过程中，相等元素的相对顺序是否保持不变。如果排序前 a 在 b 前且 $a == b$ ，排序后 a 依然在 b 前，则该算法是稳定的；否则就是不稳定的。

排序算法	稳定性	时间复杂度	空间复杂度	最佳适用场景
插入排序	✓ 稳定	$O(n^2)$	$O(1)$	小规模/基本有序数据
选择排序	✗ 不稳定	$O(n^2)$	$O(1)$	小规模/交换代价高
快速排序	✗ 不稳定	$O(n \log n)$	$O(\log n)$	通用大规模数据
归并排序	✓ 稳定	$O(n \log n)$	$O(n)$	需要稳定/外部排序
希尔排序	✗ 不稳定	$O(n \log n) \sim O(n^2)$	$O(1)$	中等规模数据
堆排序	✗ 不稳定	$O(n \log n)$	$O(1)$	大规模数据

7. 基数排序/桶排序（非比较型排序）

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

// 获取数字num的第d位数字 (d=1表示个位, d=2表示十位...)
int getDigit(int num, int d) {
    int divisor = 1;
    for (int i = 1; i < d; i++) {
        divisor *= 10;
    }
    return (num / divisor) % 10;
}

// 获取最大数的位数
int getMaxDigits(vector<int>& arr) {
    int maxVal = *max_element(arr.begin(), arr.end());
    int digits = 0;
    while (maxVal > 0) {
        digits++;
        maxVal /= 10;
    }
}
```

```
    return max(1, digits); // 至少1位
}

// LSD基数排序
void radixSortLSD(vector<int>& arr) {
    if (arr.empty()) return;

    int maxDigits = getMaxDigits(arr);
    int n = arr.size();

    // 10个桶 (0-9)
    vector<vector<int>> buckets(10);

    // 从最低位开始排序
    for (int d = 1; d <= maxDigits; d++) {
        // 清空桶
        for (auto& bucket : buckets) {
            bucket.clear();
        }

        // 分配: 将元素放入对应的桶
        for (int num : arr) {
            int digit = getDigit(num, d);
            buckets[digit].push_back(num);
        }
    }

    // 收集: 从桶中取出元素
    int index = 0;
    for (int i = 0; i < 10; i++) {
        for (int num : buckets[i]) {
            arr[index++] = num;
        }
    }
}

int main() {
    vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66};

    cout << "排序前: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    radixSortLSD(arr);

    cout << "排序后: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}
```

过程演示

初始数组: [170, 45, 75, 90, 802, 24, 2, 66]

最大位数: 3 (802是3位数)

用一个桶中数字先后顺序由上一阶段数组顺序决定

第1轮 (按个位排序) :

桶0: 170, 90

桶2: 802, 2

桶4: 24

桶5: 45, 75

桶6: 66

收集: [170, 90, 802, 2, 24, 45, 75, 66]

第2轮 (按十位排序) :

桶0: 802, 2

桶2: 24

桶4: 45

桶6: 66

桶7: 170, 75

桶9: 90

收集: [802, 2, 24, 45, 66, 170, 75, 90]

第3轮 (按百位排序) :

桶0: 2, 24, 45, 66, 75, 90

桶1: 170

桶8: 802

收集: [2, 24, 45, 66, 75, 90, 170, 802]

最终结果: [2, 24, 45, 66, 75, 90, 170, 802]

- 基数排序是稳定算法。

四、树

1. 基本概念

1) 树的组成部分

1.1 节点 (Node)

- 根节点 (Root) : 没有父节点的节点 (树的起点)

- 内部节点 (Internal Node) : 至少有一个子节点的节点
- 叶子节点 (Leaf Node) : 没有子节点的节点 (终端节点)

示例树结构



1.2 边 (Edge)

- 连接两个节点的线段
- 有向边: 从父节点指向子节点
- 无向边: 双向连接 (在某些树中)

1.3 路径 (Path)

- 从节点A到节点B经过的边的序列
- 简单路径: 节点不重复的路径

A → B → D 是一条路径
长度 = 边的数量 = 2

2) 树的度量

2.1 深度 (Depth)

- 定义: 从根节点到该节点的路径长度 (边的数量)
- 根节点深度: 0
- 计算: 向上数到根节点的边数

```
// 计算节点深度 (递归)
int getDepth(TreeNode* node) {
    if (node == nullptr) return -1;
    if (node->parent == nullptr) return 0;
    return 1 + getDepthRecursive(node->parent);
}
```

2.2 高度 (Height)

- 定义: 从该节点到最深叶子节点的最长路径长度
- 叶子节点高度: 0
- 空树高度: -1 或 0 (约定不同)

- 树的高度 = 根节点的高度

```
// 计算节点高度（递归）
int getHeight(TreeNode* node) {
    if (node == nullptr) return -1; // 空节点高度为-1

    // 如果是叶子节点
    if (node->left == nullptr && node->right == nullptr) {
        return 0;
    }

    // 递归计算左右子树高度
    int leftHeight = getHeight(node->left);
    int rightHeight = getHeight(node->right);

    // 返回较大高度 + 1
    return max(leftHeight, rightHeight) + 1;
}

// 示例:
//           A (高度=2)
//           / \
//          B   C (高度=1)
//          /     \
//         D       E (高度=0)
```

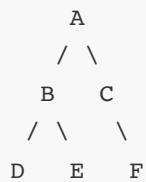
2.3 层级 (Level)

- 定义：深度 + 1
- 根节点层级：1
- 关系：Level = Depth + 1

```
int getLevel(TreeNode* node) {
    return getDepth(node) + 1;
}
```

3) 树的关系

3.1 父子关系



A是B和C的父节点 (Parent)

B和C是A的子节点 (Children)

B和C互为兄弟节点 (Siblings)

A是D、E、F的祖先 (Ancestor)

D、E、F是A的后代 (Descendant)

3.2 度 (Degree)

- 节点的度：节点拥有的子节点数量
- 树的度：树中所有节点的度的最大值

```

// 计算节点的度
int getNodeDegree(TreeNode* node) {
    if (node == nullptr) return 0;

    int degree = 0;
    if (node->left != nullptr) degree++;
    if (node->right != nullptr) degree++;
    return degree;

    // 对于多叉树：
    // return node->children.size();
}

```

4) 特殊节点统计

4.1 叶子节点 (Leaf)

```

bool isLeaf(TreeNode* node) {
    if (node == nullptr) return false;
    return node->left == nullptr && node->right == nullptr;
}

// 统计叶子节点数量
int countLeaves(TreeNode* root) {
    if (root == nullptr) return 0;

    if (isLeaf(root)) {
        return 1;
    }
}

```

```
    return countLeaves(root->left) + countLeaves(root->right);  
}
```

4.2 内部节点 (Internal Node)

```
bool isInternalNode(TreeNode* node) {  
    if (node == nullptr) return false;  
    return !isLeaf(node) && !isRoot(node);  
}  
  
// 统计内部节点数量  
int countInternalNodes(TreeNode* root) {  
    if (root == nullptr) return 0;  
    if (isLeaf(root)) return 0;  
  
    return 1 + countInternalNodes(root->left) + countInternalNodes(root->right);  
}
```

5) 树的属性

5.1 大小 (Size)

- 树中节点的总数

```
int getSize(TreeNode* root) {  
    if (root == nullptr) return 0;  
    return 1 + getSize(root->left) + getSize(root->right);  
}
```

5.2 宽度 (Width)

- 某一层的最大节点数
- 树的宽度 = 所有层宽度的最大值

6) 树的遍历相关概念

6.1 前驱 (Predecessor)

- 在中序遍历中，一个节点的前一个节点

6.2 后继 (Successor)

- 在中序遍历中，一个节点的后一个节点

```
// 在二叉搜索树中找后继  
TreeNode* findSuccessor(TreeNode* node) {
```

```

if (node == nullptr) return nullptr;

// 如果有右子树, 后继是右子树的最小节点
if (node->right != nullptr) {
    TreeNode* current = node->right;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
}

// 否则, 向上找第一个左父节点
TreeNode* parent = node->parent;
while (parent != nullptr && node == parent->right) {
    node = parent;
    parent = parent->parent;
}
return parent;
}

```

7) 重要概念总结表

概念	定义	示例/计算方法
节点	树的基本单位	包含数据和指向子节点的指针
根节点	没有父节点的节点	parent == nullptr
叶子节点	没有子节点的节点	left == nullptr && right == nullptr
内部节点	有子节点的非根节点	!isLeaf() && !isRoot()
边	连接两个节点的线段	指针/引用
深度	根节点到该节点的边数	向上数到根
高度	节点到最深叶子的边数	max(left.height, right.height) + 1
层级	深度 + 1	depth + 1
度	节点的子节点数	children.size()
路径	节点序列, 相邻节点有边连接	A→B→C
祖先	从根到该节点路径上的所有节点	递归向上找parent
后代	该节点子树中的所有节点	递归向下找children
兄弟	有相同父节点的节点	node->parent->children 中的其他节点

2. 二叉树的遍历

- 二叉树的结构体

```
struct tree{
    int data;
    tree *left;
    tree *right;
    tree(int x) : data(x), left(nullptr), right(nullptr){}
};
```

1) 前序遍历

```
void print_1(tree *root){
    if(!root) return;

    cout << root->data << " ";
    print_1(root->left);
    print_1(root->right);
    //根-左-右
}
```

- 序列第一个元素为根。 (子序列对应子树的第一个元素，是该子树的根)

2) 中序遍历

```
void print_2(tree *root){
    if(!root) return;

    print_2(root->left);
    cout << root->data << " ";
    print_2(root->right);
    //左-根-右
}
```

- 根节点对应元素左边序列为左子树，右边序列为右子树。

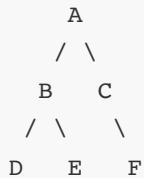
3) 后序遍历

```
void print_3(tree *root){
    if(!root) return;

    print_3(root->left);
    print_3(root->right);
    cout << root->data << " ";
    //左-右-根
}
```

- 序列最后一个元素为根。 (子序列对应子树的最后一个元素，是该子树的根)

4) 根据前、后序遍历序列其中一个和中序遍历可以构造二叉树



前序遍历: A B D E C F (根→左→右)

中序遍历: D B E A C F (左→根→右)

后序遍历: D E B F C A (左→右→根)

(1) 根据前/后序序列找到根节点 (开头/结尾) ->A

(2) 在中序序列找到根节点:

D B E | A | C F

(3) 划分子树

左子树: D B E

右子树: C F

(4) 找子树根节点, 继续划分子树的子树 (递归)

前+中序序列构造

```

tree* build_pre_in(vector<int>& preorder, int preStart, int preEnd,
                    vector<int>& inorder, int inStart, int inEnd,
                    unordered_map<int, int>& inMap) {

    //inMap: {序列中数值, 对应index}

    // 基准情况
    if (preStart > preEnd || inStart > inEnd) {
        return nullptr;
    }

    // 1. 前序第一个就是根
    tree* root = new tree(preorder[preStart]);

    // 2. 在中序里找到根的位置
    int rootIndex = inMap[root->data];

    // 3. 计算左子树节点数
    int leftSize = rootIndex - inStart;

    // 4. 递归构建左右子树
    root->left = build_pre_in(preorder, preStart + 1, preStart + leftSize,
                               inorder, inStart, rootIndex - 1, inMap);

    root->right = build_pre_in(preorder, preStart + leftSize + 1, preEnd,
                               inorder, rootIndex + 1, inEnd, inMap);
}
  
```

```
    return root;
}
```

后+中序序列构造

```
tree* build_post_in(vector<int>& postorder, int postStart, int postEnd,
                     vector<int>& inorder, int inStart, int inEnd,
                     unordered_map<int, int>& inMap) {

    // inMap: {序列中数值, 对应index}

    // 基准情况
    if (postStart > postEnd || inStart > inEnd) {
        return nullptr;
    }

    // 1. 后序最后一个就是根
    tree* root = new tree(postorder[postEnd]);

    // 2. 在中序里找到根的位置
    int rootIndex = inMap[root->data];

    // 3. 计算左子树节点数
    int leftSize = rootIndex - inStart;

    // 4. 递归构建左右子树
    root->left = build_post_in(postorder, postStart, postStart + leftSize - 1,
                                inorder, inStart, rootIndex - 1, inMap);

    root->right = build_post_in(postorder, postStart + leftSize, postEnd - 1,
                                inorder, rootIndex + 1, inEnd, inMap);

    return root;
}
```

5) 层序遍历

```
void Print4(tree* root) {
    if (root == nullptr) {
        return;
    }

    queue<tree*> q;
    //用队列储存节点
    q.push(root);

    while (!q.empty()) {
        tree* current = q.front();
        q.pop();
```

```

//先打印节点值
cout << current->data << " ";

//再分别把左右子节点放入队列
if (current->left != nullptr) {
    q.push(current->left);
}
if (current->right != nullptr) {
    q.push(current->right);
}
}

cout << endl;
}

```

层序遍历求二叉树的“宽度”

```

int getTreeWidth(tree* root) {
    if (root == nullptr) return 0;

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

    while (!q.empty()) {
        int levelSize = q.size();
        maxWidth = max(maxWidth, levelSize);

        for (int i = 0; i < levelSize; i++) {
            //只出队levelSize次，防止新加进来的节点出队
            tree* current = q.front();
            q.pop();

            if (current->left) q.push(current->left);
            if (current->right) q.push(current->right);
        }
    }

    return maxWidth;
}

```