> # 数据结构汇总（上）
>
> ## Edited: Rium

一、线性表

- **线性表的逻辑结构**：数据元素之间线性的序列关系，即数据元素之间的前驱和后继关系。
- **线性表的物理结构**：线性表在计算机中的存储方式，又称为存储结构，即从程序实现的角度将逻辑结构映射到计算机存储单元中。
- **顺序存储结构**：数据元素被顺序地存储在连续的内存空间中，前驱和后继元素在物理空间上是相邻的。
- **链式存储结构**：可以动态地申请存储数据的结点空间，并使用类似指针这样的手段将结点按顺序前后链接起来

1. 链表和线性表对比

访问操作（Access）

操作	顺序表	链表
随机访问（按索引）	**O(1)** - 直接通过地址偏移	**O(n)** - 需要从头开始遍历
按值查找	**O(n)** - 线性搜索	**O(n)** - 线性搜索

插入操作（Insertion）

操作	顺序表	链表
头部插入	**O(n)** - 需要移动所有元素	**O(1)** - 修改头指针
尾部插入	**O(1)** (如果有尾指针/已知长度)	**O(1)** (如果有尾指针)
中间插入	**O(n)** - 需要移动后续元素	**O(n)** - 需要先找到插入位置
尾部追加	**O(1)** 平摊复杂度	**O(1)** (如果有尾指针)

删除操作（Deletion）

操作	顺序表	链表
头部删除	**O(n)** - 需要移动所有元素	**O(1)** - 修改头指针
尾部删除	**O(1)**	**O(n)** (单链表) 或 **O(1)** (双向链表+尾指针)
中间删除	**O(n)** - 需要移动后续元素	**O(n)** - 需要先找到节点

空间复杂度

方面	顺序表	链表
存储密度	高 - 只存数据	低 - 需要额外存储指针
内存分配	连续内存块	分散的内存节点
扩容成本	高 - 可能需要复制全部元素	低 - 动态分配

小结

特性	顺序表	链表
内存连续性	连续	不连续
随机访问	快 (O(1))	慢 (O(n))
头部插入删除	慢 (O(n))	快 (O(1))
空间利用率	高	较低（有指针开销）
缓存友好性	好	差
适用场景	频繁随机访问	频繁插入删除

2. 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);
root->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){}
};
```

3. 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
```

龟兔赛跑算法

```
// 龟兔赛跑算法 - 找环的入口点
node* findCycleStart(node* head) {
    if (!head || !head->next) return nullptr;

    // 第一阶段：检测是否有环
```

```
    node* slow = head;  // 乌龟（每次走1步）
    node* fast = head;  // 兔子（每次走2步）

    // 检测环是否存在
    while (fast && fast->next) {
        slow = slow->next;           // 乌龟走一步
        fast = fast->next->next;     // 兔子走两步

        if (slow == fast) {
            // 有环，进入第二阶段
            break;
        }
    }

    // 如果没有环，返回nullptr
    if (!fast || !fast->next) {
        return nullptr;
    }

    // 第二阶段：找到环的入口点
    // 将一个指针放回头部，两个指针都每次走一步
    slow = head;

    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    // 相遇点就是环的入口点
    return slow;
}
```

二、栈与队列

1. 栈

栈（Stack）是一种**后进先出（LIFO）** 的线性数据结构。以下是栈的常规操作，我按重要性和使用频率分类：

操作	描述	时间复杂度
push(x)	入栈/压栈，将元素x放入栈顶	O(1)
pop()	出栈/弹栈，移除并返回栈顶元素	O(1)
top() / peek()	查看栈顶元素（不弹出）	O(1)
empty() / isEmpty()	判断栈是否为空	O(1)
size()	返回栈中元素个数	O(1)
clear()	清空栈	O(n)
search(x)	查找元素x在栈中的位置（从栈顶开始计数）	O(n)

- 上溢（Overflow）：栈中已有maxsize个元素时，再做进栈运算时产生的现象
- 下溢（Underflow）：对空栈进行出栈运算时所产生的现象

(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)
```

- 卡特兰数：给定长度为 N 的入栈序列（假设为 1, 2, ..., N），合法的出栈序列数量就是第 N 个卡特兰数。

$$F(n) = \frac{2n!}{(n+1)!n!} = \frac{C_{2n}^{n}}{n+1}$$

(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

队列（Queue）是一种**先进先出（FIFO）** 的线性数据结构。以下是队列的常规操作：

操作	描述	时间复杂度
enqueue(x)	入队/插入，将元素x添加到队尾	O(1)
dequeue()	出队/删除，移除并返回队首元素	O(1)
front() / peek()	查看队首元素（不删除）	O(1)
isEmpty()	判断队列是否为空	O(1)
rear() / back()	查看队尾元素	O(1)
size()	返回队列中元素个数	O(1) 或 O(n)
clear()	清空队列	O(n)

- 上溢：当队列满时，入队操作所出现的现象
- 下溢：当队列空时，出队操作所出现的现象
- 假溢出：顺序队列可能出现的一种现象：当队尾指针达到最大值（rear = mSize）时，再做入队运算产生溢出，但此时队列前端可能尚有空闲位置

实指 & 虚指

为了防止满/空判断条件相同，往往要浪费一个空间。

实指

- front指向队首元素
- rear指向最后一个元素
- 空队列判断：(rear + 1) % capacity == front
- 满队列判断：(rear + 2) % capacity == front
- size = (capacity + rear - front + 1) mod capacity

虚指

- front指向队首元素
- rear指向最后一个元素
- 空队列判断：rear== front
- 满队列判断：(rear + 1) % capacity == front
- size = (capacity + rear - front) mod capacity

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;
}
```

约瑟夫环问题

```
vector<int> josephusQueue(int n, int k, int m) {
    //n为总人数，k表示从第k个人开始报数，m表示报到m的人淘汰。

    vector<int> result;  // 存储出列顺序
    queue<int> q;

    // 1. 初始化队列：1, 2, 3, ..., n
```

```
    for (int i = 1; i <= n; i++) {
        q.push(i);
    }

    // 2. 调整起始位置到第k个人
    // 将前k-1个人移动到队尾
    for (int i = 1; i < k; i++) {
        q.push(q.front());
        q.pop();
    }

    // 3. 开始报数淘汰
    while (!q.empty()) {
        // 报数1到m-1：出队再入队
        for (int i = 1; i < m; i++) {
            q.push(q.front());
            q.pop();
        }

        // 报数m：淘汰（出队不返回）
        result.push_back(q.front());
        q.pop();
    }

    return result;
    //result是淘汰顺序。
}
```

*单调栈/单调队列

栈/队列中的元素保持**单调递增**或**单调递减**的顺序。

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;
    }
    /*可以在这里输出一遍排序结果*/
}

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）
```

```cpp
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;

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

    // 将基准放到正确位置
    swap(arr[i], arr[high]);
    return i;  // 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

```cpp
#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++];
        }
    }

    // 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

```cpp
#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;
}
```

- 冒泡排序懒得写

---

*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
冒泡排序	✅ 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)~$O(n^2)$	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

*逆序数

- **逆序数**：在一个序列中，如果 `i < j` 但 `arr[i] > arr[j]`，则 `(i, j)` 构成一个逆序对。
- **关键性质**：
  - 完全有序数组的逆序数为 **0**
  - 完全逆序数组的逆序数为 **n(n-1)/2**

排序算法	交换/移动次数与逆序数的关系	能否直接计算逆序数	逆序数如何影响算法效率	关键特点
冒泡排序	**交换次数 = 逆序数** 每次相邻交换恰好消除一个逆序对	✅$O(n^2)$时间内精准计算	逆序数越少，效率越高 最好情况(逆序数=0): O(n) 最坏情况: $O(n^2)$	相邻交换，逆序数的物理实现
插入排序	**比较/移动次数 ≈ 逆序数** 每个元素插入时移动次数等于其左边的逆序对数	❌ 不能直接计算 但移动次数反映了逆序数	逆序数越少，效率越高 自适应算法，对部分有序数组高效	每个元素插入时消除其所有逆序
选择排序	**交换次数固定 = n-1** 与逆序数**无关** 一次交换可能消除多个逆序	❌ 完全无关	效率与逆序数**无关** 总是执行固定次数的交换	盲目交换，不考虑逆序关系
希尔排序	**比较/移动次数与逆序数相关** 通过不同间隔消除跨间隔逆序	❌ 不能直接计算	逆序数越多，需要的间隔序列越长 对中等大小逆序数有效	预处理减少逆序数，再插入排序
归并排序	**可在排序过程中计算逆序数** 交换次数 ≤ 逆序数	✅ **最佳计算方法** O(n log n)时间内精确计算	效率与逆序数**无关** 总是O(n log n)	分治算法，归并时统计逆序
快速排序	**间接相关** 分区操作减少逆序 交换次数与逆序数相关	❌ 不能直接计算	逆序数影响分区平衡性 完全有序(逆序数=0)可能最差	分区时破坏逆序关系

- 堆排序效率与逆序对数量无关。

---

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
          A (根节点)
         / \
        B   C (内部节点)
       / \   \
      D   E   F (叶子节点)
```

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
         / \
        B   C
       / \   \
      D   E   F

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 getTreeSize(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + getTreeSize(root->left) + getTreeSize(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

```
        A
       / \
      B   C
     / \   \
    D   E   F

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;
}
```

前序遍历+“#”构建

```
tree *build(string &input, int &idx){
    if(idx >= input.length())return NULL;

    char c = input[idx];
    idx++;
    if(c == '#')return NULL;

    tree *root = new tree(c);
    root->left = build(input, idx);
    root->right = build(input, idx);

    return root;
}

/*                                                          *\
str = AB##C## (“#”代表空节点)

          A
         / \
        B   C
\*                                                          */
```

5）表达式树

表达式树：

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

遍历方式	遍历结果	对应表达式类型	是否需要括号
中序	`(a + b) * (c - d)`	中缀表达式	是
前序	`* + a b - c d`	前缀表达式	否
后序	`a b + c d - *`	后缀表达式	否

表达式树：

遍历方式	遍历结果
中序	`((a + b) * c) - (d / e)`
前序	`- * + a b c / d e`
后序	`a b + c * d e / -`

6）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;
}
```

7）二叉树种类

完美二叉树 (Perfect Binary Tree)

```
深度为h，节点数 = 2^h - 1
每层都完全填满

          A            (h=3, 节点=7)
        /   \
       B     C
      / \   / \
     D   E F   G
```

完全二叉树 (Complete Binary Tree)

```
除了最后一层，其他层都完全填满
最后一层从左到右连续填充
```

```
        A           (有效)
      /   \
     B     C
    / \   /
   D   E F

        A           (无效！F应该在E的右边)
      /   \
     B     C
    / \     \
   D   E     F
```

满二叉树 (Full Binary Tree)

```
每个节点有0个或2个子节点（不能只有1个）

        A           (有效：每个节点0或2个子节点)
      /   \
     B     C
    / \
   D   E

        A           (无效！C只有1个子节点)
      /   \
     B     C
    /     /
   D     F
```

特性	完美二叉树 (Perfect)	完全二叉树 (Complete)	满二叉树 (Full)
节点数公式	`2^h - 1`	无固定公式	无固定公式
叶子节点	全在最底层	在最后两层	可在任意层
填充规则	每层都满	最后一层从左到右连续	无填充要求
子节点数	非叶子节点都有2个子节点	无要求	0或2个子节点
空指针数	叶子节点数+1	不一定	叶子节点数+1
堆的结构	一定是堆	常用作堆	不一定是堆

数学性质对比

完美二叉树

- 高度 = h
- 节点总数 = 2^h - 1
- 叶子节点数 = 2^(h-1)

- 度为2的节点数 = 2^(h-1) - 1
- **空指针数** = 2^h (每个叶子有2个空指针)

完全二叉树（高度h）

- 节点数范围：`2^(h−1) ≤ n ≤ 2^h − 1`
- 最后一个父节点下标：`⌊n/2⌋ − 1`（0-based）
- **重要性质**：可用**数组高效存储**，无需指针

满二叉树

- 叶子节点数 = 度为2的节点数 + 1
- 空指针数 = 叶子节点数 + 1
- **性质**：没有度为1的节点

3. 哈夫曼树

示例：字符集 {A,B,C,D}，频率 {5, 1, 6, 3}

步骤1：初始化

```
叶节点：A(5)  B(1)  C(6)  D(3)
```

步骤2：每次合并最小的两个

1. 合并 B(1) 和 D(3) → 新节点 4

   ```
      4(4)
     /    \
   B(1)   D(3)
   ```

2. 合并 A(5) 和 4(4) → 新节点 9

   ```
      9(9)
     /    \
   A(5)   4(4)
         /    \
       B(1)   D(3)
   ```

3. 合并 9(9) 和 C(6) → 根节点 15

```
      15(15)
      /    \
    9(9)   C(6)
   /    \
 A(5)   4(4)
      /    \
    B(1)   D(3)
```

最终哈夫曼树：

```
        ○(15)
       /      \
     ○(9)      C(6)
    /    \
  A(5)   ○(4)
        /    \
      B(1)   D(3)
```

性质	说明
最优二叉树	WPL最小，是最优压缩
前缀编码	任意编码不是其他编码的前缀
唯一性	相同权值可能有不同形状，但WPL相同
贪心算法	每次合并最小的两个节点
满二叉树	所有内部节点都有两个子节点

哈夫曼归并

```
4个顺串：A(20), B(25), C(30), D(40)
哈夫曼合并过程：
1. 合并最小的A(20)+B(25) → AB(45)
   当前：AB(45), C(30), D(40)
2. 合并C(30)+D(40) → CD(70)
   当前：AB(45), CD(70)
3. 合并AB(45)+CD(70) → ABCD(115)

总IO次数 = (20+25) + (30+40) + (45+70) = 190
```

4. 父亲、孩子、孩子兄弟表示法

```
// 方法1：数组实现（最常用）
struct ParentTreeNode {
    int data;       // 节点数据
```

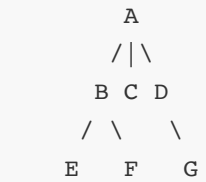
```
    int parent;     // 父节点下标，-1表示根节点
};

struct ChildrenTreeNode {
    int data;       // 节点数据
    vector<int> children;     // 孩子节点下标
};


struct Children_Sibling_TreeNode{
    int data;
    int first_kid;      //左孩子节点
    int right_sibling;     //右兄弟节点
};
 // 数组存储所有节点
vector<ParentTreeNode> tree_1;
vector<ChildrenTreeNode> tree_2;
vector<Children_Sibling_TreeNode> tree_3;

// 方法2：指针实现
struct ParentNode {
    int data;
    ParentNode* parent;  // 指向父节点
};
```

5. 树/森林的遍历

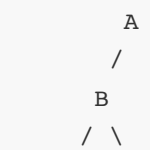
- **前序遍历**：先访问树根，然后对根的各子树从左向右依次进行前序遍历；
- **后序遍历**：遍历根的各子树，最后访问根结点；
- 森林前/后去遍历：从其中的第一个树开始，按序对每个树进行前/后序遍历
- 无**中序遍历**。
- 树的**前序遍历**与对应的二叉树的**前序遍历**结果相同；树的**后序遍历**与对应的二叉树的**中序遍历**结果相同

利用孩子兄弟表示法，把“右兄弟”变成“右孩子”，转化成二叉树

原始树：



对应二叉树：



```
    E   C
     \   \
      F   D
          /
         G
```