> # 数据结构汇总（下）
>
> # Edited: Rium

五、二叉堆

二叉堆是**完全二叉树**的一种应用，是实现**优先队列（Priority Queue）** 最高效的数据结构之一。

1. 二叉堆的基本概念

定义

二叉堆是一棵**完全二叉树**，满足以下堆性质（Heap Property）：

- **最大堆（Max-Heap）**：每个节点的值 ≥ 其子节点的值
- **最小堆（Min-Heap）**：每个节点的值 ≤ 其子节点的值

核心特点

1. **结构性**：一定是**完全二叉树**
2. **有序性**：满足堆性质（最大堆或最小堆）
3. **高效性**：插入O(log n)、删除O(log n)、获取最值O(1)

最大堆（大顶堆）

```
        100
       /    \
      19     36
     /  \   /  \
    17   3 25   1
   /  \
  2    7

每个父节点 ≥ 子节点
根节点是最大值
```

最小堆（小顶堆）

```
          1
        /   \
       3     6
      / \   / \
     5   9 8  10
    / \
   14 11

每个父节点 ≤ 子节点
根节点是最小值
```

2. 二叉堆的存储方式

数组存储（核心优势）

由于是完全二叉树，可以用**数组紧凑存储**，无需指针！

```
二叉堆：
         1 (0)
        /   \
      3 (1)  6 (2)
     / \     / \
    5   9   8   10
  下标：0  1  2  3  4  5  6
  数组：[1, 3, 6, 5, 9, 8, 10]
```

下标关系公式（0-based indexing）

```
// 已知节点下标 i
parent(i)    = (i - 1) / 2  // 父节点
leftChild(i) = 2*i + 1      // 左孩子
rightChild(i)= 2*i + 2      // 右孩子
```

3. 核心操作详解

上浮（Sift Up / Heapify Up）

当节点值比父节点大（最大堆）或小（最小堆）时，向上调整。

```
// 最大堆的上浮
void SiftUp(vector<int>& heap, int index) {
    while (index > 0) {
        int parent = (index - 1) / 2;
        if (heap[index] <= heap[parent]) break;  // 已满足堆性质
```

```
        swap(heap[index], heap[parent]);  // 交换
        index = parent;  // 继续向上检查
    }
}

// 最小堆的上浮（只需改变比较条件）
void SiftUpMin(vector<int>& heap, int index) {
    while (index > 0) {
        int parent = (index - 1) / 2;
        if (heap[index] >= heap[parent]) break;  // 已满足最小堆性质

        swap(heap[index], heap[parent]);
        index = parent;
    }
}
```

下沉（Sift Down / Heapify Down）

当节点值比子节点小（最大堆）或大（最小堆）时，向下调整。

```
// 最大堆的下沉
void SiftDown(vector<int>& heap, int index, int size) {
    while (true) {
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int largest = index;

        // 找到父节点和两个子节点中的最大值
        //left < size和right < size防止访问越界
        if (left < size && heap[left] > heap[largest]) {
            largest = left;
        }
        if (right < size && heap[right] > heap[largest]) {
            largest = right;
        }

        // 如果父节点已经是最大，停止
        if (largest == index) break;

        // 否则与较大的子节点交换，继续下沉
        swap(heap[index], heap[largest]);
        index = largest;
    }
}
```

插入

```cpp
// 最大堆插入函数
void HeapInsert(vector<int>& heap, int value) {
    // 1. 将新元素添加到堆的末尾
    heap.push_back(value);

    // 2. 上浮调整：从新节点开始，向上与父节点比较
    int index = heap.size() - 1;  // 新节点的索引
    SiftUp(heap, index);
}
```

删除最大值

```cpp
//最大堆删除最大值
void DeleteMax(vector<int>& heap){
    if (heap.empty()) return;
    int n = heap.size() - 1;
    swap(heap[0], heap[n]);
    heap.pop_back();
    SiftDown(heap, 0, n);
}
```

4. 建堆

方法一：自顶向下法（Insertion Method） - O(n log n)

逐个插入元素，每次插入后上浮调整。

```cpp
/**
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(1)
 */
vector<int> BuildHeapTopDown(const vector<int>& arr) {
    vector<int> heap;

    for (int value : arr) {
        heap.push_back(value);
        // 从新插入的元素开始上浮
        int index = heap.size() - 1;
        while (index > 0) {
            int parent = (index - 1) / 2;
            if (heap[index] <= heap[parent]) break;  // 最大堆
            swap(heap[index], heap[parent]);
            index = parent;
        }
    }

    return heap;
```

```
}
```

时间复杂度分析：

- 第1层：1个节点，上浮0次
- 第2层：2个节点，最多上浮1次
- 第3层：4个节点，最多上浮2次
- ...
- 第h层：2^(h-1)个节点，最多上浮(h-1)次

总操作次数：`Σ (i=0 to h-1) 2^i * i ≈ O(n log n)`

方法二：自底向上法（Heapify Method / Floyd算法） - O(n)

从最后一个非叶子节点开始，向前进行下沉调整。

```
/**
 * 方法2：自底向上建堆（Floyd建堆法）
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 */
void BuildHeapBottomUp(vector<int>& arr) {
    int n = arr.size();

    // 从最后一个非叶子节点开始，向前遍历
    // 最后一个非叶子节点下标 = n/2 - 1
    for (int i = n/2 - 1; i >= 0; i--) {
        SiftDown(arr, i, n);
    }
}
```

数学分析：

```
对于一个高度为h的完全二叉树：

- 第1层（根）：1个节点，最多下沉h-1次

- 第2层：2个节点，最多下沉h-2次

- 第3层：4个节点，最多下沉h-3次

- ...

- 第h层（叶子）：2^(h-1)个节点，下沉0次
```

总操作次数 S：

```
S = 2^(h-2) * 1 + 2^(h-3) * 2 + ... + 2^0 * (h-1)
  = Σ (i=1 to h-1) 2^(h-i-1) * i
设 n = 2^h - 1（完美二叉树的节点数），经过数学推导：
```

设 n = 2^h - 1（完美二叉树的节点数），经过数学推导：

$$S = n - h < n$$

5. 堆排序（Heap Sort）算法

```cpp
//使用SiftDown函数堆排序

// 堆排序主函数
void HeapSort(vector<int>& arr) {
    int n = arr.size();
    if (n <= 1) return;  // 边界情况

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

    // 第一步：先建堆
    // 从最后一个非叶子节点开始，向前进行下沉操作
    // 最后一个非叶子节点的下标 = n/2 - 1
    for (int i = n/2 - 1; i >= 0; i--) {
        SiftDown(arr, i, n);
    }

    // 第二步：把最大数放在最后，然后队前面的数建堆。
    for (int i = n-1; i > 0; i--){
        swap(arr[0], arr[i]);

        // i以后的数已排好序。
        SiftDown(arr, 0, i);
    }
}
```

堆排序复杂度分析

- **时间复杂度**：O(n log n)
    - 建堆：O(n)（不是O(n log n)！）
    - 排序：O(n log n)
- **空间复杂度**：O(1)（原地排序）
- **稳定性**：不稳定排序
- **优点**：最坏情况也是O(n log n)，适合大数据量

6. 优先队列的实现

```cpp
#include <queue>  // 必须包含的头文件
using namespace std;

signed main(){
    priority_queue<int> maxHeap;  // 默认最大堆

    /*push*/
    maxHeap.push(10);  // O(log n)
    maxHeap.push(30);
    maxHeap.push(20);
    maxHeap.push(5);
    // 插入后自动调整堆，保证堆顶是最大值
    // 此时堆顶：30

    /*top*/
    int maxValue = maxHeap.top();  // O(1)
    cout << "堆顶元素：" << maxValue << endl;  // 输出：30

    /*pop*/
    maxHeap.pop();  // O(log n)
    // 删除堆顶元素（最大值），并重新调整堆
    // 删除后新的堆顶：20

    /*empty & size*/
    if (maxHeap.empty()) {
        cout << "优先队列为空" << endl;
    } else {
        cout << "优先队列非空，大小：" << maxHeap.size() << endl;
    }
    // O(1)

}
```

时间复杂度总结

操作	时间复杂度	说明
插入（push）	O(log n)	上浮调整
删除堆顶（pop）	O(log n)	下沉调整
获取堆顶（top）	O(1)	直接访问
检查是否为空 (empty)	O(1)	—
获取大小 (size)	O(1)	—
建堆（heapify）	O(n)	从后往前下沉
堆排序	O(n log n)	建堆+排序
查找任意元素	O(n)	需要遍历

7. 应用

多路归并（Huffman树）

输入：K个有序序列（数组、链表等）
输出：一个合并后的有序序列

步骤：
1. 从每个序列中取出第一个元素，放入最小堆
2. 每次从堆中取出最小元素，放入结果
3. 从该元素所在序列取出下一个元素入堆
4. 重复直到堆为空

哈夫曼树（Huffman Tree） 与 **最大堆（Max Heap）** 的对比表格：

特性	哈夫曼树（Huffman Tree）	最大堆（Max Heap）
用途	数据压缩（最优前缀编码）	实现优先队列，快速获取最大值
结构性质	不一定是完全二叉树	**一定是完全二叉树**
结点值含义	权重（频率）	优先级（值）
结点度数	**没有度为1的结点**（只有0或2）	可以有度为1的结点（最后层未满时）
值的关系	父结点的权重是两子结点权重和，不一定大于子孙	父结点的值 ≥ 子结点的值（堆性质）
唯一性	权重相同可能树形不同（可交换左右子树）	不唯一，但结构固定为完全二叉树

Dijkstra最短路径算法（优化）

```
// 使用最小堆优化，从O(V²)降到O(E log V)
void dijkstra(const Graph& graph, int start) {
    MinHeap pq;  // (distance, node)
    vector<int> dist(graph.size(), INF);

    pq.push({0, start});
    dist[start] = 0;

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;

        for (auto [v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

---

*与其他数据结构的对比

数据结构	插入	删除最大	获取最大	合并	适合场景
二叉堆	O(log n)	O(log n)	O(1)	O(n)	通用优先队列
有序数组	O(n)	O(1)	O(1)	O(n+m)	静态数据
链表	O(1)	O(n)	O(n)	O(1)	频繁插入
平衡BST	O(log n)	O(log n)	O(log n)	O(n)	需要查找
二项堆	O(1)	O(log n)	O(1)	O(1)	需要合并

六、查找

1. 顺序查找

```cpp
int SequentialSearch(vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            return i;  // 返回索引
        }
    }
    return -1;  // 未找到
}
```

情况	时间复杂度	说明
最好情况	O(1)	目标在第一个位置
最坏情况	O(n)	目标在最后一个位置或不存在
平均情况	O(n)	需要检查约 n/2 个元素

平均查找长度（ASL）

成功时的 ASL：

$$ASL_{成功} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

失败时的 ASL：

$$ASL_{失败} = n$$

查找最小和最大值

(1)朴素查找

```cpp
void solve1(vector<int> &arr, int &max_idx, int &min_idx){
    if(arr.empty())return;

    for(int i = 0; i < arr.size(); i++){
        if(arr[i] > arr[max_idx]) max_idx = i;
        else if(arr[i] < arr[min_idx]) min_idx = i;
    }
}
```

- 最坏情况比较$2(n-1)$次。

（2）快速查找

```cpp
void solve2(vector<int> &a, int &max_idx, int &min_idx){
    int n = a.size();
    if(!n)return;
    max_idx = 0, min_idx = 0;
    int k = n%2;
```

```
    while(k < n-1){
        if(a[k] < a[k+1]){
            if(a[k] < a[min_idx]) min_idx = k;
            if(a[k+1] > a[max_idx]) max_idx = k+1;
        }else{
            if(a[k+1] < a[min_idx]) min_idx = k+1;
            if(a[k] > a[max_idx]) max_idx = k;
        }
        k += 2;
    }
}
```

- 比较次数是$\frac{3}{2}n$次。

埃式筛查法判断质数

```
vector<int> sieveOfEratosthenes(int n) {
    vector<bool> isPrime(n + 1, true);
    vector<int> primes;

    // 0和1不是质数
    isPrime[0] = isPrime[1] = false;

    // 埃式筛法核心
    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            // 将i的倍数标记为非质数
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    // 收集所有质数
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
        }
    }

    return primes;
}
```

- 时间复杂度：O(n log log n)
- 空间复杂度：O(n)
- 将整数a∈[1, … , n]质因数分解，可得 $a = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k}$，其中$p_1, \dots, p_k$是互异的质数（k >1）且指数n_1,…,n_k都大于0。问在埃氏筛选法中，a会被删除k 次

合数限定法

```
//理论上是O(n)复杂度
vector<int> solve(int n){
    vector<bool> isPrime(n + 1, true);
    vector<int> primes;
    isPrime[0] = isPrime[1] = false;
    queue<int> composite;    //储存合数

    //初始化composite
    for(int i = 4; i <= n; i+=2){
        isPrime[i] = false;
        composite.push(i);
    }

    for(int i = 3; i <= n; i++){
        if(!isPrime[i])continue;
        queue<int> tmp = composite;
        while(!tmp.empty()){
            int t = tmp.front();
            tmp.pop();
            while(t*i <= n){
                t *= i;
                isPrime[t] = false;
                composite.push(t);
            }
        }
    }
    //防止一个合数被删除多次

    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
        }
    }

    return primes;
}
```

欧拉筛选法

```
//O(n)复杂度
vector<int> linearSieve(int n) {
    vector<bool> isPrime(n + 1, true);
    vector<int> primes;
    isPrime[0] = isPrime[1] = false;

    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
```

```
        }

        // 用当前已找到的质数筛去合数
        for (int j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            isPrime[i * primes[j]] = false;

            // 关键：保证每个合数只被最小质因数筛一次
            if (i % primes[j] == 0) {
                break;
            }
        }
    }

    return primes;
}
```

2. 二分查找

```
int binarySearch(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;  // 防止溢出

        if (nums[mid] == target) {
            return mid;  // 找到目标，返回索引
        } else if (nums[mid] < target) {
            left = mid + 1;  // 目标在右侧
        } else {
            right = mid - 1;  // 目标在左侧
        }
    }

    return -1;  // 未找到
}
```

时间复杂度

- **最好情况**：O(1) —— 第一次就找到
- **最坏情况**：O(log n) —— 需要一直折半
- **平均情况**：O(log n)

空间复杂度

- 迭代实现：O(1) —— 只用了几个变量
- 递归实现：O(log n) —— 递归调用栈的深度

✅ 必须满足的条件：

1. **有序性**：数组必须是有序的（升序或降序）
2. **随机访问**：支持通过索引直接访问元素（数组可以，链表不行）

---

*快速幂算法

```
double fastPowRecursive(double a, int n) {
    if (n == 0) return 1;
    if (n == 1) return a;

    // 二分法：a^n = a^(n/2) × a^(n/2)
    double half = fastPowRecursive(a, n / 2);

    if (n % 2 == 0) {
        // n为偶数：a^n = (a^(n/2))^2
        return half * half;
    } else {
        // n为奇数：a^n = (a^(n/2))^2 × a
        return half * half * a;
    }
}
```

*疯牛问题

```

bool canPlaceCows(vector<int>& positions, int C, int dist) {
    int count = 1;  // 第一头牛放在第一个位置
    int lastPos = positions[0];

    for (int i = 1; i < positions.size(); i++) {
        if (positions[i] - lastPos >= dist) {
            count++;
            lastPos = positions[i];
            if (count >= C) {
                return true;
            }
        }
    }

    return false;
}

int aggressiveCows(vector<int>& positions, int C) {
    sort(positions.begin(), positions.end());
    int n = positions.size();

    int low = 1;
```

```
    int high = positions[n-1] - positions[0];
    int result = 0;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (canPlaceCows(positions, C, mid)) {
            result = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return result;
}
```

3. 索引查找

- **索引查找**是一种通过建立**索引表**来加速查找过程的方法。它将查找表分为若干块，为每块建立索引项，查找时先查索引，再在对应块中查找。

```
主表（数据表） + 索引表 + 索引项
struct IndexItem {
    KeyType maxKey;     // 块内最大关键字
    int startAddr;      // 块起始地址
    int blockLength;    // 块长度（可选）
};

示例
主表：[22, 12, 13, 8, 9, 20, 33, 42, 44, 38, 24, 48, 60, 58, 74]
分为3块，每块5个元素

索引表：
块1：maxKey=22, start=0, length=5
块2：maxKey=48, start=5, length=5
块3：maxKey=74, start=10, length=5

查找38：
1. 查索引：38>22, 38≤48 → 在块2
2. 在块2中顺序查找：找到38
```

七、二叉查找树

二叉查找树或者是一棵空树；或者是具有如下特性的二叉树：

- 若根结点的左子树非空，则左子树上所有结点的值均小于根结点的值；
- 若根结点的右子树非空，则右子树上所有结点的值均大于根结点的值；
- 左、右子树本身也是一棵二叉查找树。

1. 二叉查找树的构造

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

(1) 已知前序序列

```
tree* build_pre(vector<int>& preorder, int start, int end) {
     if (start > end) return nullptr;

     // 第一个元素是根节点
     tree* root = new tree(preorder[start]);

     // 如果只有一个元素，直接返回
     if (start == end) return root;

     // 找右子树起始下标
     int i = start + 1;
     while (i <= end && preorder[i] < preorder[start]){
         i++;
     }

     // 递归构建左右子树
     // 左子树：start+1 到 rightStart-1
     root->left = build(preorder, start + 1, i - 1);
     // 右子树：rightStart 到 end
     root->right = build(preorder, i, end);

     return root;
}
```

(2) 已知后序序列

```
tree* build_post(vector<int>& postorder, int start, int end) {
     if (start > end) return nullptr;

     // 第一个元素是根节点
     tree* root = new tree(preorder[end]);
```

```
    // 如果只有一个元素，直接返回
    if (start == end) return root;

    // 找右子树起始下标
    int i = start;
    while (i <= end && preorder[i] < preorder[start]){
        i++;
    }

    // 递归构建左右子树
    // 左子树：start+1 到 rightStart-1
    root->left = build(postorder, start, i - 1);
    // 右子树：rightStart 到 end
    root->right = build(postorder, i, end-1);

    return root;
}
```

(3) 已知层序序列

```
tree *build_level(vector<int> &levelorder){
    if(levelorder.empty()) return nullptr;

    int root_val = levelorder[0];
    tree *root = new tree(root_val);
    vector<int>L, R;
    for(int i = 1; i < levelorder.size(); i++){
        if(levelorder[i] < root_val) L.push_back(levelorder[i]);
        if(levelorder[i] > root_val) R.push_back(levelorder[i]);
    }
    root->left = build_level(L);
    root->right = build_level(R);

    return root;
}
```

2. 二叉查找树的操作

(1) 插入

```
tree* insert(TreeNode* root, int val) {
    if (!root) {
        return new tree(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else if (val > root->val) {
        root->right = insert(root->right, val);
```

```
    }
    // 如果val相等，根据BST定义通常不插入重复值

    return root;
}
```

(2) 删除

```
tree* findMin(tree* root) {
    while (root && root->left) {
        root = root->left;
    }
    return root;
}

tree* deleteNode(tree* root, int key) {
    if (!root) return nullptr;

    // 查找要删除的节点
    if (key < root->val) {
        // 在左子树中删除
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        // 在右子树中删除
        root->right = deleteNode(root->right, key);
    } else {
        // 找到了要删除的节点
        // 情况1：叶子节点：直接删除
        if (!root->left && !root->right) {
            delete root;
            return nullptr;
        }

        // 情况2：只有一个子节点：用子节点代替原节点
        else if (!root->left) {
            // 只有右子节点
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            // 只有左子节点
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        // 情况3：有两个子节点
        else {
            // 用右子树的最小值替换（更常用）
            TreeNode* successor = findMin(root->right);
```

```
            root->val = successor->val;  // 复制值
            root->right = deleteNode(root->right, successor->val);
            // 删除后继节点
        }
    }

    return root;
}
```

3. ASL计算

对每个节点计算深度，然后求平均。

示例：BST如下

text

```
        8
       / \
      5   10
     / \    \
    3   6    14
   / \      /
  1   4    12
```

成功查找计算：

节点	深度	比较次数（深度+1）
8	0	1
5	1	2
10	1	2
3	2	3
6	2	3
14	2	3
1	3	4
4	3	4
12	3	4

总比较次数 = 1 + 2 + 2 + 3 + 3 + 3 + 4 + 4 + 4 = 26
节点数 n = 9

$ASL_{succ} = \frac{26}{9} \approx 2.89$

*4. AVL树

- **二叉查找树**：左子树 < 根 < 右子树
- **平衡条件**：任意节点的**左右子树高度差**不超过1
- **高度平衡**：保证查找、插入、删除操作的时间复杂度为O(log n)
- **平衡因子**：对于节点X，平衡因子$BF(X)$ = 左子树高度 - 右子树高度

```
struct AVLtree {
    int key;
    int height;       // 节点高度
    AVLNode* left;
    AVLNode* right;

    AVLNode(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
};
```

AVL树失衡调节

当插入或删除节点后，可能出现平衡因子为±2的情况，需要旋转调整：

- **LL型（左左）**

```
    A (BF=2)
   /
  B (BF=1)
 /
C
```

解决方法：右旋转

```
AVLtree *RightRotate(AVLtree *x){
    if (!x || !x->left) return x;

    AVLtree *y = x->left;
    AVLtree *T2 = y->right;

    //旋转
    y->right = x;
```

```
    x->left = T2;

    // 更新高度
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    return y;
}
```

- **RR型（右右）**

```
A (BF=-2)
 \
  B (BF=-1)
   \
    C
```

解决方法：左旋转

```
AVLtree* LeftRotate(AVLtree* x) {
    if (!x || !x->right) return x;

    AVLtree* y = x->right;
    AVLtree* T2 = y->left;

    // 旋转
    y->left = x;
    x->right = T2;

    // 更新高度
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}
```

- **LR型（左右）**

```
    A (BF=2)
   /
  B (BF=-1)
   \
    C
```

解决方法：先左旋后右旋

```
AVLtree* Left_Right_Rotate(AVLtree* z) {
    z->left = leftRotate(z->left);
    return rightRotate(z);
}
```

- **RL型（右左）**

```
A (BF=-2)
 \
  B (BF=1)
 /
C
```

解决方法：先右旋后左旋

```
AVLtree* Right_Left_Rotate(AVLtree* z) {
    z->right = rightRotate(z->right);
    return leftRotate(z);
}
```

八、散列表

1. 关键术语

- **散列函数**：一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以 $H(key)$ 作为关键字为 key 的记录在表中的位置，通常称这个函数 $H(key)$ 为散列函数。
- **键（Key）**：要存储或查找的数据标识
- **值（Value）**：实际存储的数据
- **哈希值**：哈希函数的输出
- **桶（Bucket）**：数组中的一个位置
- **冲突（Collision）**：不同键映射到相同位置

2. 散列函数的设计

(1) 直接散列函数

$$H(key) = key$$

或

$$H(key) = a \times key + b$$

(2) 数字分析法

```
8 1 3 4 6 5 3 2 -> 45
8 1 3 7 2 2 4 2 -> 72
8 1 3 8 7 4 2 2 -> 84
8 1 3 0 1 3 6 7 -> 03
      ^   ^
    选取这两位
```

(3) 平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值” 的目的是“**扩大差别**”和“**贡献均衡**”。

(4) 折叠法

关键字位数较长时，可将关键字**分割成位数相等的几部分**（最后一部分位数可以不同），取这几部分的叠加和（舍去高位的进位）作为散列地址。位数由存储地址的位数确定。

(5) 除留余数法（常用）

取关键字被某个不大于散列表长度m的数p除后的余数作为散列地址，即：

$$\mathcal{H}(key) = key \bmod p (p \leq m)$$

(6) 随机函数法

$$\mathcal{H}(key) = random(key)$$

3. 冲突处理

(1) 开放地址法

为产生冲突的地址 $\mathcal{H}(\text{key})$ 求得一个地址序列：

$$H_0, H_1, H_2, \ldots, H_s, \quad 1 \leq s \leq m-1$$

其中：

$$H_0 = \mathcal{H}(\text{key})$$

$$H_i = (\mathcal{H}(\text{key}) + d_i) \bmod m, \quad i = 1, 2, \ldots, s$$

线性探测

$$d_i = c \times i$$

平方探测

$$d_i = 1^2, -1^2, 2^2, -2^2, \ldots, n^2, -n^2$$

随机探测

$$d_i = random(i), random为伪随机函数$$

(2) 再散列法

将n个不同散列函数排成一个序列, 当发生冲突时, 由$\mathcal{R}H_i$确定第i次冲突的地址H_i。即:

$$H_i = \mathcal{R}H_i(key)i = 1, 2, \ldots, n$$

其中：$\mathcal{R}H_i$ 为不同散列函数

- 这种方法不会产生“聚类”，但会增加计算时间。

(3) 链地址法

将所有散列地址相同的记录都链接在同一链表中。

(4) 公共溢出法

```
{15, 23, 47, 35, 28, 19, 10, 37, 42, 51}按序放入哈希表
主表：[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
      10  51  42  23  空  15  空  47  28  19
溢出区：[35, 37]
```

九、图

1. 图的基本概念

图（Graph）

- **定义**：由**顶点集合** V 和**边集合** E 组成的数据结构
- **记法**：G = (V, E)
- **示例**：社交网络（顶点=用户，边=好友关系）

顶点（Vertex/Node）

- 图中的基本元素，也称为**节点**
- 通常用字母表示：v_1, v_2, ..., 或 u, v, w

边（Edge）

- 连接两个顶点的关系
- **类型**：
    - **无向边**：(u, v) = (v, u)
    - **有向边**：<u, v> ≠ <v, u>（有方向）

2. 图的分类

按边的方向分类

类型	边特点	示例
无向图	边没有方向	公路网、社交网络
有向图	边有方向	网页链接、任务依赖
混合图	同时有无向和有向边	交通网（单行道+双行道）

按边的权重分类

类型	边是否有权值	示例
无权图	边无权重（或权重=1）	熟人关系
带权图	边有权重	地图距离、网络延迟

按特殊结构分类

类型	定义	特点
简单图	无自环、无平行边	最基本
多重图	允许平行边	公交线路
伪图	允许自环和平行边	状态机
完全图	每对顶点都有边	K_n（n个顶点）
二分图	顶点分两组，组内无边	任务分配

3. 基本术语

邻接与关联

- **邻接顶点**：有边直接相连的两个顶点

- **关联边**：与某个顶点相连的边

度（Degree）

```
# 无向图：顶点v的度 = 与v相连的边数
deg(v) = |{e ∈ E : v ∈ e}|

# 有向图：
- 入度(in-degree)：指向v的边数
- 出度(out-degree)：从v指出的边数
- 总度 = 入度 + 出度
```

握手定理：无向图中，所有顶点的度之和 = 2 × 边数

$$\sum_{v \in V} \deg(v) = 2|E|$$

路径（Path）

- **定义**：顶点序列 $v_0, v_1, \ldots, v_k$，其中 $(v_i, v_{i+1}) \in E$
- **长度**：路径中边的数量（或权重和）
- ***简单路径**：顶点不重复的路径
- ***回路/环**：起点=终点的路径

连通性

术语	定义	示例
连通图	任意两顶点间都有路径	完整公路网
连通分量	极大连通子图	孤岛上的公路
强连通	有向图中双向可达	互相链接的网页
弱连通	忽略方向后连通	单行道网络

4. 特殊图结构

环（Cycle）

- 长度≥3的闭合路径
- **偶环**：长度为偶数的环
- **奇环**：长度为奇数的环

自环

- 自环是指图中一条边的两个端点是同一个顶点的边。

完全图 K_n

- n个顶点，每对顶点间都有边
- 边数：|E| = n(n-1)/2
- K_5（五角形加所有对角线）

5. 图的表示方法

（1） 邻接矩阵

```
// n个顶点的图 → n×n矩阵
// A[i][j] = 1 表示顶点i到j有边（权重）
int adjMatrix[n][n];

// 无向图：对称矩阵
// 带权图：存储权重而不是1
```

优缺点：

- ✅ 快速判断两顶点是否相邻：O(1)
- ❌ 空间复杂度：$O(n^2)$，稀疏图浪费空间

（2）邻接表

```
// 每个顶点维护一个邻居列表
vector<vector<int>> adjList(n);

// 无向图：双向添加
adjList[u].push_back(v);
adjList[v].push_back(u);

// 带权图：存储pair(邻居, 权重)
vector<vector<pair<int, int>>> weightedAdjList(n);
```

优缺点：

- ✅ 空间效率高：$O(|V| + |E|)$
- ❌ 判断(u,v)是否相邻：$O(deg(u))$

（3）关联矩阵

- 顶点×边的矩阵
- 每列只有两个1（无向图）或一个1一个-1（有向图）

- 较少使用，多用于理论分析

6. 遍历算法

算法	特点	数据结构
深度优先搜索	一条路走到黑再回溯	栈
广度优先搜索	层层扩展，找最短路径	队列

- 以邻接表为例

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

signed main(){
    int n;  cin >> n;
    vector<vector<int>> adj(n+1); // 邻接表
    vector<bool> vis1(n+1, false);
    vector<bool> vis2(n+1, false);

    int t; cin >> t;
    while(t--){
        int x, y;
        cin >> x >> y;
        //无向图处理
        adj[x].push_back(y);
        adj[y].push_back(x);
    }

    dfs(1, adj, vis1);
    //dfs_satck(1, adj, vis1);

    bfs_all(1,adj, vis2);
    //bfs(1, adj, vis2);
    return 0;
}
```

(1) 深度优先搜索

```
//递归实现遍历
void dfs(int node, vector<vector<int>>& adj, vector<bool>& vis) {
    // 标记当前节点已访问
    vis[node] = true;
    cout << node << " ";  // 输出访问的节点

    // 遍历当前节点的所有邻居
    for (int neighbor : adj[node]) {
```

```
        // 如果邻居未被访问，则递归访问
        if (!vis[neighbor]) {
            dfs(neighbor, adj, vis);
        }
    }
}

//栈实现遍历
void dfs_stack(int start, vector<vector<int>>& adj, vector<bool>& vis) {
    stack<int> stk;
    stk.push(start);

    while (!stk.empty()) {
        int node = stk.top();
        stk.pop();

        if (vis[node]) continue;  // 已经访问过
        vis[node] = true;
        cout << node << " ";

        // 反向遍历邻居以保持与递归相同的顺序
        for (int i = adj[node].size() - 1; i >= 0; i--) {
            int neighbor = adj[node][i];
            if (!vis[neighbor]) {
                stk.push(neighbor);
            }
        }
    }
}
```

(2) 广度优先搜索

```
//只遍历一个连通分支
void bfs(int start, vector<vector<int>>& adj, vector<bool>& visited) {
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";  // 处理当前节点

        // 遍历所有邻居
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
```

```
        }
    }
}

//遍历所有连通分支，DFS也是类似处理
void bfs_all() {
    vector<bool> visited(n + 1, false);

    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            cout << "分量 [ ";
            bfs(i, visited);
            cout << "]" << endl;
        }
    }
}
```

7. 拓扑序列&拓扑排序

（1）拓扑序列（Topological Order/Sequence）

- **定义**：有向无环图（DAG）中顶点的一个**线性排序**
- **特性**：对于图中的每一条有向边 `u → v`，在序列中 `u` 都出现在 `v` 之前
- **意义**：表示顶点间的**依赖关系**或**先后顺序**

（2）拓扑排序（Topological Sorting）

- **定义**：找到有向无环图的一个拓扑序列的过程
- **结果**：一个顶点的线性序列
- **应用**：任务调度、课程安排、依赖解析

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

//Kahn算法-BFS思路：
vector<int> topoBFS(int n, vector<vector<int>> &adj){
    vector<int> indegree(n + 1, 0);  // 入度数组
    vector<int> result;

    // 1. 计算所有顶点的入度
    for (int u = 1; u <= n; u++) {
        for (int v : adj[u]) {
            indegree[v]++;
        }
    }
```

```
    // 2. 将入度为0的顶点加入队列
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    // 3. BFS处理
    int count = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);
        count++;

        // 减少所有邻居的入度
        for (int v : adj[u]) {
            indegree[v]--;
            if (indegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // 4. 检查是否有环
    if (count != n) {
        cout << "图中有环，无法进行拓扑排序！" << endl;
        return {};
    }

    return result;
}

//显式栈DFS思路
vector<int> topoDFS(int n, vector<vector<int>>& adj) {
    vector<int> visited(n + 1, 0);
    vector<int> result;
    stack<pair<int, int>> stk;  // pair<节点, 下一个邻居索引>

    for (int start = 1; start <= n; start++) {
        if (visited[start] != 0) continue;

        stk.push({start, 0});

        while (!stk.empty()) {
            auto& [u, idx] = stk.top();

            if (visited[u] == 0) {
                visited[u] = 1;  // 开始访问
            }
```

```
            // 如果所有邻居都处理完了
            if (idx >= adj[u].size()) {
                visited[u] = 2;  // 访问完成
                result.push_back(u);
                stk.pop();
                continue;
            }

            int v = adj[u][idx];
            idx++;

            if (visited[v] == 1) {
                // 发现环
                cout << "图中有环，无法进行拓扑排序！" << endl;
                return {};
            }

            if (visited[v] == 0) {
                stk.push({v, 0});
            }
        }
    }

    reverse(result.begin(), result.end());
    return result;
}

signed main(){
    int n;  cin >> n;
    vector<vector<int>> adj(n+1); // 邻接表
    vector<bool> vis(n+1, false);

    int t; cin >> t;
    while(t--){
        int x, y;
        cin >> x >> y;
        //有向图处理
        adj[x].push_back(y);
    }


    return 0;
}
```

- 复杂度均为$O(|V|+|E|)$

8. 最短路径

算法	适用场景	时间复杂度
Dijkstra	非负权图	O((V+E)logV)
Bellman-Ford	可有负权，检测负环	O(VE)
Floyd-Warshall	所有顶点对最短路径	$O(V^3)$

Dijkstra算法

```
const int INF = INT_MAX;

// 邻接矩阵实现 Dijkstra（未优化，O(V²)）
vector<int> dijkstraAdjMatrix(int n, vector<vector<int>>& graph, int start) {
    //n为节点数，start为起点，dist为起点到各点的最短距离

    vector<int> dist(n, INF);
    vector<bool> visited(n, false);

    // 初始化起点
    dist[start] = 0;

    // 循环 n-1 次，每次确定一个最短路径
    for (int i = 0; i < n - 1; i++) {
        // 1. 找到未访问节点中距离最小的
        int u = -1;
        int minDist = INF;

        for (int j = 0; j < n; j++) {
            if (!visited[j] && dist[j] < minDist) {
                minDist = dist[j];
                u = j;
            }
        }

        // 如果找不到，说明剩余节点不可达
        if (u == -1) break;

        // 2. 标记为已访问
        visited[u] = true;

        // 3. 更新所有邻居的距离
        for (int v = 0; v < n; v++) {
            // 如果有边且未访问
            if (graph[u][v] != INF && !visited[v]) {
                int newDist = dist[u] + graph[u][v];
                if (newDist < dist[v]) {
                    dist[v] = newDist;
                }
            }
        }
```

```
    }
    return dist;
}

// 邻接矩阵 + 优先队列优化
vector<int> dijkstraAdjMatrixOptimized(int n, vector<vector<int>>& graph, int start) {
    vector<int> dist(n, INF);
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>> pq;

    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d > dist[u]) continue;

        // 遍历所有可能的邻居
        for (int v = 0; v < n; v++) {
            if (graph[u][v] != INF) {  // 如果有边
                int newDist = dist[u] + graph[u][v];
                if (newDist < dist[v]) {
                    dist[v] = newDist;
                    pq.push({newDist, v});
                }
            }
        }
    }

    return dist;
}

// 邻接表实现 Dijkstra
vector<int> dijkstraAdjList(int n, vector<vector<pair<int, int>>>& adj, int start) {
    vector<int> dist(n, INF);      // 最短距离数组
    vector<bool> visited(n, false); // 是否已确定最短路径

    // 小顶堆：pair<距离, 顶点>
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>> pq;

    // 初始化起点
    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        // 取出当前距离最小的顶点
        auto [d, u] = pq.top();
```

```
        pq.pop();

        // 如果这个距离已经过时（有更优解），跳过
        if (d > dist[u]) continue;

        // 标记为已处理（可选的，不影响正确性）
        visited[u] = true;

        // 松弛操作：更新所有邻居的距离
        for (auto& [v, weight] : adj[u]) {
            // 如果通过 u 到 v 的距离更短
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

Bellman-Ford算法

```
struct Edge {
    int u, v, weight;
};

// Bellman-Ford 算法
pair<vector<int>, bool> bellmanFord(int n, vector<Edge>& edges, int start) {
    vector<int> dist(n, INF);
    dist[start] = 0;

    // 进行 n-1 轮松弛操作
    for (int i = 0; i < n - 1; i++) {
        bool relaxed = false;

        for (const Edge& e : edges) {
            if (dist[e.u] != INF && dist[e.u] + e.weight < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.weight;
                relaxed = true;
            }
        }

        // 如果一轮中没有松弛发生，提前结束
        if (!relaxed) break;
    }

    // 第 n 轮检查负权环
    bool hasNegativeCycle = false;
    for (const Edge& e : edges) {
        if (dist[e.u] != INF && dist[e.u] + e.weight < dist[e.v]) {
```

```
            hasNegativeCycle = true;
            break;
        }
    }

    return {dist, !hasNegativeCycle};  // true 表示没有负环
}
```

9. 最小生成树

算法	策略	时间复杂度
Prim	从一点开始逐步扩展	O((V+E)logV)
Kruskal	按权重排序边，避免环	O(E log E)

Prim算法

```
const int INF = INT_MAX;

// Prim 算法模板（邻接表 + 优先队列）
int prim(int n, vector<vector<pair<int, int>>>& adj) {
    vector<bool> inMST(n, false);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    //优先队列储存pair<int, int>，底层容器是vector，最小堆。
    pq.push({0, 0});  // 从顶点 0 开始

    int totalWeight = 0, edgesUsed = 0;

    while (!pq.empty() && edgesUsed < n) {
        auto [weight, u] = pq.top();
        pq.pop();

        if (inMST[u]) continue;

        inMST[u] = true;
        totalWeight += weight;
        edgesUsed++;

        for (auto& [v, w] : adj[u]) {
            if (!inMST[v]) {
                pq.push({w, v});
            }
        }
    }
```

```
    return edgesUsed == n ? totalWeight : -1;
}
```

Kruskal算法

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

struct ee{
    int x;
    int y;
    int dis;
};
bool cmp(ee &a, ee &b){
    return a.dis < b.dis;
}

vector<int> parent;

//并查集
int find(int x) {
    if (parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
}

bool unite(int x, int y) {
    int px = find(x), py = find(y);
    if (px == py) return false;
    parent[px] = py;
    return true;
}

signed main(){
    int len; cin >> len;   //节点数
    int t; cin >> t;       //边数
    vector<ee> ns;
    parent.resize(t+1);

    int ans = 0;

    for (int i = 0; i < t+1; i++) parent[i] = i;

    if(!t){
        cout << 0 <<endl;
        return 0;
    }

    while(t--){
        ee tmp;
```

```
        cin >> tmp.x >> tmp.y >> tmp.dis;
        ns.push_back(tmp);
    }

    sort(ns.begin(), ns.end(), cmp);
    // 每次选择最小的边。
    for(auto &T : ns){
        if(unite(T.x, T.y)){
            ans += T.dis;
        }
    }

    cout << ans <<endl;
    return 0;
}
```

输入样例

```
7 12
1 2 9
1 5 2
1 6 3
2 3 5
2 6 7
3 4 6
3 7 3
4 5 6
4 7 2
5 6 3
5 7 6
6 7 1
```

输出样例

```
16
```

10. 并查集

并查集是一种处理**不相交集合**的数据结构，支持：

1. **查找**（Find）：查找元素所在集合的代表元素
2. **合并**（Union）：合并两个集合
3. **查询**：判断两个元素是否在同一集合

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

```cpp
const int MAXN = 100010;  // 最大元素个数

// parent[i] 表示 i 的父节点
int parent[MAXN];

// 初始化：每个元素自成一个集合
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;  // 自己就是根
    }
}

// 查找：找到元素 x 所在集合的根
int find(int x) {
    while (parent[x] != x) {
        x = parent[x];  // 不断向上找
    }
    return x;
}

// 合并：将 x 和 y 所在集合合并
void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX != rootY) {
        parent[rootX] = rootY;  // 简单合并
    }
}

// 判断是否在同一集合
bool same_set(int x, int y) {
    return find(x) == find(y);
}
```

11. 网络流

流网络$G = (V, E)$是一个有向加权图，其有向边的方向表示流的方向，边(u,v)的非负权值c(u,v)表示边的容量capacity。

- 流网络包含两个特殊结点：源点(source) s 和汇点(sink) t
- 通常情况下，每个结点都在从s到t的某条路径上
- **网络流**：给定流网络G=(V,E)及流 f:V×V→R，网络流等于从源点流出的净流量，即

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

- **增广路径**：剩余流网络$G_f = (V, E_f)$中，从源点s到汇点t的简单路径。增广路径包含的所有边的权重的最小值，称作路径的增广容量。

- **最大流**： 流网络G中流量最大的网络流称为G的最大流。
- **割(cut)**： 流网络G=(V, E)中，一个割(S,T)是对结点集合V的划分，满足 $s \in S, t \in T$且$T = V - S$

最大流、最小割定理

设流网络$G = (V, E)$的流为 f:V×V→R，下面的描述是等价的：

(1) 存在割(S, T)，满足$|f| = c(S, T)$ （最小割）

(2) f是流网络G的最大流

(3) f对应的剩余流网络G_f中，没有从源点s到漕点t的增广路径

Ford-Fulkerson算法(最大流算法)

```
(1) 开始时，流网络G的流量设置为0

(2) 在对应当前流f的剩余流网络G_f中查找增广路径 （DFS？BFS？…）

(3) 如果不存在增广路径，则f是最大流，结束算法；否则执行(4)

(4) 沿增广路径对流网络增流，重复(2)
```