CSP 复赛复习 - 基础算法(4)

并查集 (Union-Find)

基本概念

并查集：用于管理元素分组的数据结构，支持两种操作：

- **合并** (Union)：将两个元素所在的集合合并
- **查找** (Find)：查询元素所在的集合

应用场景：连通性判断、最小生成树、动态连通性

时间复杂度：

- 朴素实现：$O(n)$
- 路径压缩 + 按秩合并：$O(\alpha(n))$，其中 $\alpha(n)$ 为反阿克曼函数

并查集基本实现

```
const int N = 100010;

int parent[N];  // 父节点数组

// 初始化
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }
}

// 查找（路径压缩）
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);  // 路径压缩
    }
    return parent[x];
}
```

```
// 合并
void unionSet(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX != rootY) {
        parent[rootX] = rootY;
    }
}
```

并查集应用示例

```
// 判断两个元素是否连通
bool isConnected(int x, int y) {
    return find(x) == find(y);
}

// 统计连通分量数量
int countComponents(int n) {
    int count = 0;
    for (int i = 1; i <= n; i++)
        if (parent[i] == i)
            count++;
    return count;
}
```

```
int main() {
    int n, m;
    cin >> n >> m;

    init(n);

    for (int i = 0; i < m; i++) {
        int op, x, y;
        cin >> op >> x >> y;

        if (op == 1) {
            // 合并操作
            unionSet(x, y);
        } else {
            // 查询操作
            if (isConnected(x, y)) {
                cout << "Y" << endl;
            } else {
                cout << "N" << endl;
            }
        }
    }

    return 0;
}
```

17.2 Hash 表

基本概念

哈希表：通过哈希函数将键映射到数组下标的数据结构

关键问题：

- 哈希函数设计
- 冲突解决方法
- 负载因子控制

时间复杂度：

- 平均情况：$O(1)$
- 最坏情况：$O(n)$

哈希表实现（链地址法）

```cpp
const int N = 100003;  // 质数，减少冲突

int h[N], e[N], ne[N], idx;
// 初始化
void init() {
    memset(h, -1, sizeof(h));
    idx = 0;
}
// 哈希函数
int hash_func(int x) {
    return (x % N + N) % N;  // 处理负数
}
// 插入
void insert(int x) {
    int k = hash_func(x);
    e[idx] = x,ne[idx] = h[k],h[k] = idx++;
}
// 查找
bool find(int x) {
    int k = hash_func(x);
    for (int i = h[k]; i != -1; i = ne[i])
        if (e[i] == x)
            return true;
    return false;
}
```

字符串哈希

```
typedef unsigned long long ULL;
const int N = 100010;
const int P = 131;  // 质数基数

ULL h[N], p[N];  // h[i] 前缀哈希值，p[i] P的i次方

// 初始化
void init(char str[]) {
    p[0] = 1;
    for (int i = 1; str[i]; i++) {
        p[i] = p[i-1] * P;
        h[i] = h[i-1] * P + str[i];
    }
}
```

```
// 获取子串哈希值
ULL get_hash(int l, int r) {
    return h[r] - h[l-1] * p[r-l+1];
}

// 判断两个子串是否相等
bool is_equal(int l1, int r1, int l2, int r2) {
    return get_hash(l1, r1) == get_hash(l2, r2);
}
```

18.3 DAG 与拓扑排序

基本概念

DAG (有向无环图)：没有环的有向图

拓扑排序：将 DAG 的所有顶点排成线性序列，使得对任意有向边 (u, v)，u 在 v 之前

应用场景：任务调度、依赖关系分析、编译顺序

时间复杂度：$O(V + E)$

拓扑排序实现（BFS - Kahn算法）

```
vector<int> graph[N];  // 邻接表
int indegree[N];       // 入度数组
int result[N];         // 拓扑排序结果

bool topologicalSort(int n) {
    queue<int> q;

    // 将所有入度为0的节点加入队列
    for (int i = 1; i <= n; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    int cnt = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result[cnt++] = u;

        // 遍历所有邻居
        for (int v : graph[u]) {
            indegree[v]--;
            if (indegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // 判断是否有环
    return cnt == n;
}
```

```
int main() {
    int n, m;
    cin >> n >> m;

    // 建图
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        indegree[v]++;
    }

    if (topologicalSort(n)) {
        for (int i = 0; i < n; i++) {
            cout << result[i] << " ";
        }
        cout << endl;
    } else {
        cout << "图中存在环" << endl;
    }

    return 0;
}
```

18.4 常见的 STL 容器使用

string（字符串）

```cpp
#include <string>
string str = "Hello";

// 基本操作
str.push_back('!');        // 尾部添加字符
str.pop_back();            // 删除尾部字符
str.length();              // 字符串长度
str.size();                // 同length()
str.empty();               // 判断空
str[0];                    // 随机访问

// 子串操作
string sub = str.substr(1, 3);    // 获取子串：从位置1开始，长度3
size_t pos = str.find("ell");     // 查找子串
str.replace(1, 2, "i");           // 替换子串

// 字符串连接
string s1 = "Hello", s2 = "World";
string s3 = s1 + " " + s2;        // "Hello World"

// 遍历
for (int i = 0; i < str.length(); i++)
    cout << str[i];
for (char c : str)
    cout << c;
// 排序
sort(str.begin(), str.end());
```

vector（动态数组）

```cpp
#include <vector>
#include <algorithm>
using namespace std;

vector<int> vec;

// 基本操作
vec.push_back(1);           // 尾部插入
vec.pop_back();             // 尾部删除
vec.size();                 // 元素个数
vec.empty();                // 判断空
vec[0];                     // 随机访问

// 遍历
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}

for (auto x : vec) {
    cout << x << " ";
}

// 排序
sort(vec.begin(), vec.end());
```

queue（队列）

```cpp
#include <queue>
using namespace std;

queue<int> q;

// 基本操作
q.push(1);            // 入队
q.pop();              // 出队
q.front();            // 队首元素
q.back();             // 队尾元素
q.size();             // 元素个数
q.empty();            // 判断空

// 遍历（会破坏队列）
while (!q.empty()) {
    cout << q.front() << " ";
    q.pop();
}
```

stack（栈）

```
#include <stack>
using namespace std;

stack<int> st;

// 基本操作
st.push(1);          // 入栈
st.pop();            // 出栈
st.top();            // 栈顶元素
st.size();           // 元素个数
st.empty();          // 判断空

// 遍历（会破坏栈）
while (!st.empty()) {
    cout << st.top() << " ";
    st.pop();
}
```

set（集合）

```cpp
#include <set>
using namespace std;

set<int> s;

// 基本操作
s.insert(1);          // 插入
s.erase(1);           // 删除
s.find(1);            // 查找，返回迭代器
s.count(1);           // 统计个数（0或1）
s.size();             // 元素个数
s.empty();            // 判断空

// 遍历
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << " ";
}

for (auto x : s) {
    cout << x << " ";
}
```

unordered_set（无序集合）

```cpp
#include <unordered_set>
using namespace std;

unordered_set<int> us;

// 基本操作
us.insert(1);        // 插入
us.erase(1);         // 删除
us.find(1);          // 查找，返回迭代器
us.count(1);         // 统计个数（0或1）
us.size();           // 元素个数
us.empty();          // 判断空

// 遍历
for (auto it = us.begin(); it != us.end(); it++) {
    cout << *it << " ";
}

for (auto x : us) {
    cout << x << " ";
}

// 与set的区别：unordered_set基于哈希表，查找O(1)，但元素无序
```

map（映射）

```cpp
#include <map>
using namespace std;

map<string, int> mp;

// 基本操作
mp["apple"] = 5;     // 插入/修改
mp.erase("apple");   // 删除
mp.find("apple");    // 查找
mp.count("apple");   // 统计个数（0或1）
mp.size();           // 元素个数
mp.empty();          // 判断空

// 遍历
for (auto it = mp.begin(); it != mp.end(); it++) {
    cout << it->first << ": " << it->second << endl;
}

for (auto &[key, value] : mp) {
    cout << key << ": " << value << endl;
}
```

unordered_map（无序映射）

```cpp
#include <unordered_map>
using namespace std;

unordered_map<string, int> ump;

// 基本操作
ump["apple"] = 5;   // 插入/修改
ump.erase("apple"); // 删除
ump.find("apple");  // 查找
ump.count("apple"); // 统计个数（0或1）
ump.size();         // 元素个数
ump.empty();        // 判断空

// 遍历
for (auto it = ump.begin(); it != ump.end(); it++) {
    cout << it->first << ": " << it->second << endl;
}

for (auto &[key, value] : ump) {
    cout << key << ": " << value << endl;
}

// 与map的区别：unordered_map基于哈希表，查找O(1)，但元素无序
```

priority_queue（优先队列）

```cpp
#include <queue>
using namespace std;

// 最大堆（默认）
priority_queue<int> pq;

// 最小堆
priority_queue<int, vector<int>, greater<int>> min_pq;

// 基本操作
pq.push(1);          // 插入
pq.pop();            // 删除堆顶
pq.top();            // 堆顶元素
pq.size();           // 元素个数
pq.empty();          // 判断空

// 自定义比较函数
struct Node {
    int x, y;
    bool operator<(const Node& other) const {
        return x < other.x;  // 按x从大到小
    }
};
priority_queue<Node> custom_pq;
```

算法复杂度总结

数据结构	主要操作	时间复杂度	空间复杂度
并查集	Find/Union	$O(\alpha(n))$	$O(n)$
哈希表	插入/查找	$O(1)$	$O(n)$
拓扑排序	排序	$O(V+E)$	$O(V+E)$

STL 容器复杂度对比

容器	插入	删除	查找	有序性	底层实现
vector	$O(1)$ 尾部	$O(1)$ 尾部	$O(1)$	有序	动态数组
string	$O(1)$ 尾部	$O(1)$ 尾部	$O(1)$	有序	动态数组
queue	$O(1)$	$O(1)$	不支持	FIFO	队列
stack	$O(1)$	$O(1)$	不支持	LIFO	栈
set	$O(\log n)$	$O(\log n)$	$O(\log n)$	有序	红黑树
map	$O(\log n)$	$O(\log n)$	$O(\log n)$	有序	红黑树
unordered_set	$O(1)$	$O(1)$	$O(1)$	无序	哈希表
unordered_map	$O(1)$	$O(1)$	$O(1)$	无序	哈希表