线性表

dbywsc

2025/7

目录

静态数组

事实上，**数组**(Array) 是一种线性的数据结构。它的特点是每个元素都是紧密相连着的，是一块连续的内存。
我们一般说的数组是**静态数组**，也就是大小不可修改，形如 $int\ a[];$ 这样的数组。
数组支持以 $O(1)$ 的时间复杂度随机访问元素，以 $O(1)$ 的时间复杂度在末尾插入元素。由于内存是连续的，所以无法直接在数组的中间或者头部插入或者一个元素。

动态数组 – I

在 C++ 中，我们通常将 *vector* 容器作为动态数组使用。
vector 既可以提前分配大小，也可以随着元素的增多动态的分配内存，下面是一些 *vector* 的常用操作：
std :: *vector* < *type* > *name*(*size*) 创建一个 type 类型、名为 name 并且大小为 size 的 vector。
std :: *vector* < *type* > *name* 创建一个 type 类型，名为 name 的 vector 。
a.size() 可以用 size() 迭代器 得到 vector 当前的容量。

```
for(int i = 0; i < a.size(); i++) std::cout << a[i] << " ";
```

可以像使用数组一样遍历和访问 vector

动态数组 – II

可以使用 $begin()$ 和 $end()$ 迭代器表示 vector 中的收尾位置。

```
sort(a.begin(), a.end());
```

上面的例子即对一个 vector a 做了从头到尾的升序排序。

$a.push_back(x)$ 以 $O(1)$ 的时间复杂度在 vector 末尾插入一个元素 x

$a.insert(it, x)$ 以 $O(n)$ 的时间复杂度在 it 处插入一个元素 x

$a.clear()$ 以 $O(n)$ 的时间复杂度清空 vector

$a.pop_back()$ 以 $O(1)$ 的时间复杂度删除数组末尾的元素

$a.erase(it)$ 以 $O(n)$ 的时间复杂度删除 it 处的元素，并且保证其余的元素仍然是紧密相连着的。

介绍

栈 (Stack) 是一种后进先出的数据结构。
栈通常能够实现以下操作：
$s.top()$ 访问栈顶元素
$s.push(x)$ 将 x 入栈，此时 x 会成为新的栈顶。
$s.pop()$ 将栈顶出栈，此时原本栈顶下的那个元素会成为新的栈顶。
$s.size()$ 返回当前栈内元素的个数。
由于栈只需要维护栈顶，所以我们无法实现在非栈顶的位置的插入和删除操作。

模拟栈 – I

B3614 【模板】栈

实现栈的方式非常多，在数据结构的课程中通常使用指针的方式实现，但是在算法竞赛中，出于速度和内存安全性的考虑，我们通常使用静态数组模拟栈，之后介绍的几个数据结构同样会采用静态数组模拟。

可以使用一个数组 $s[1..n]$ 来表示栈，用一个变量 top 表示当前的栈顶，初始时 $top = 0$，如果入栈就让 $top+ = 1$，再为 $s[top]$ 赋值，出栈就让 $top- = 1$。所以栈的元素数量等于 top，判断栈是否为空就看 top 是否为 0 。

模拟栈 – II

```
void solve(void) {
    top = 0;
    int n; std::cin >> n;
        while(n--) {
            std::string op; std::cin >> op;
            if(op == "push") {
                u64 x; std::cin >> x;
                s[++top] = x;
            } else if(op == "pop") {
                if(top > 0) {
                    top--;
                } else {
                    std::cout << "Empty" << std::endl;
                }
            } else if(op == "query") {
                if(top > 0) std::cout << s[top] << std::endl;
                else std::cout << "Anguei!" << std::endl;
            } else {
                std::cout << top << std::endl;
            }
        }
```

STL 中的栈 – I

在 STL 库中，我们可以使用 *stack* 容器。
stack 有以下迭代器：
s.size() 返回栈中的元素个数
s.empty() 返回栈是否为空
s.top() 返回栈顶元素
s.push(*x*) 将 *x* 入栈
s.pop() 将栈顶出栈

STL 中的栈 – II

所以对于这道题，我们可以这样写：

```
void solve(void) {
    std::stack<u64> s;
    int n; std::cin >> n;
    while(n--) {
        std::string op; std::cin >> op;
        if(op == "push") {
            u64 x; cin >> x; s.push(x);
        }
        else if(op == "pop") {
            if(!s.empty()) s.pop();
            else cout << "Empty" << endl;
        } else if(op == "query") {
            if(!s.empty()) cout << s.top() << endl;
            else cout << "Anguei!" << endl;
        } else cout << s.size() << endl;
    }
}
```

介绍

队列(Queue) 是一种**先进先出**的数据结构。
队列通常能实现以下操作:
$q.front()$ 访问**队首**元素
$q.push(x)$ 将 x 入队，此时 x 会成为新的**队首**。
$q.pop()$ 将**队尾**出队，此时原本队尾前的那个元素会成为新的队尾。
$q.size()$ 返回当前队内元素的个数。

模拟队列 – I

B3616 【模板】队列
与模拟栈类似，我们同样是使用一个静态数组 $q[1..n]$ 来表示队列，不同的是我们需要维护两个指针 $head$ 和 $tail$ 分别表示队首和队尾。开始时 $head$ 和 $tail$ 都指向 0 ，如果入队就让 $tail+=1$，再为 $q[tail]$ 赋值；如果出队就让 $head+=1$ ，如果 $head$ 和 $tail$ 重合说明此时队列为空，队列内的元素个数应为 $head-tail$ ，队首元素应为 $q[head+1]$ 。

模拟队列 – II

```
void solve(void) {
    int Q; cin >> Q;
    while(Q--) {
        string op; cin >> op;
        if(op == "1") {
            i64 x; cin >> x;
            q[++tail] = x;
        } else if(op == "2") {
            if(head == tail) std::cout << "ERR_CANNOT_POP" << std::endl;
            else head ++;
        } else if(op == "3") {
            if(head == tail) std::cout << "ERR_CANNOT_QUERY" << std::endl;
            else std::cout << q[head+1] << std::endl;
        } else {
            std::cout << tail - head << std::endl;
        }
    }
}
```

STL 中的队列 – I

在 STL 库中，我们可以使用 *queue* 容器。
queue 有以下迭代器：
q.size() 返回队列中的元素个数
q.empty() 返回队列是否为空
q.front() 返回队首元素
q.push(*x*) 将 *x* 入队
q.pop() 将队尾出队

STL 中的队列 – II

对于这道题，我们可以这样写：

```
void solve(void) {
    std::queue<int> q;
    int Q; std::cin >> Q;
    while(Q--) {
        int a; cin >> a;
        if(a == 1) {
            int x; std::cin >> x;
            q.push(x);
        } else if(a == 2) {
            if(q.size()) q.pop();
            else std::cout << "ERR_CANNOT_POP" << std::endl;
        } else if(a == 3) {
            if(!q.empty()) std::cout << q.front() << std::endl;
            else std::cout << "ERR_CANNOT_QUERY" << std::endl;
        } else {
            std::cout << q.size() << std::endl;
        }
    }
}
```

介绍 – I

链表有非常多种形式，但是在算法竞赛主要的作用是存储图 (Graph)，因此本节我们只介绍单向链表和双向链表。

链表 (Linked_List) 是一种可以方便的在元素中间进行插入和删除操作的数据结构，链表由节点 (Node) 和连接节点的指针组成，每个节点存储本节点的元素和它指向的下一个链表的指针（双链表还会存储它的上一个节点的指针）。由于节点并不是连续的，因此无法像数组一样进行 $O(1)$ 的随机访问，只能通过 $O(n)$ 的遍历寻找节点。

介绍 – II

单链表的节点存储键值和它指向的下一个节点的指针 (next)。
单链表支持两种操作：插入和删除。
对于插入操作，假设要在 $Node_1$ 和 $Node_2$ 中间插入 $Node_3$，那么过程是这样的：创建 $Node_3$，原本 $Node_1$ 指向 $Node_2$，随着 $Node_3$ 的加入，指向关系变成了 $Node_1 \rightarrow Node_3 \rightarrow Node_2$，插入操作就结束了。
对于删除操作，假设要删除 $Node_1$ 和 $Node_2$ 之间的 $Node_3$，那么只需要更改它们的指向关系，由 $Node_1 \rightarrow Node_3 \rightarrow Node_2$ 变为 $Node_1 \rightarrow Node_2$，由于没有节点指向 $Node_3$，作为一个无法访问到的节点，我们就默认为它被删除了。

介绍 – III

双链表的节点存储键值、它指向的下一个节点的指针、指向它的上一个节点的指针，称为**前驱**(pre) 和**后继**(next)。
双链表同样支持插入和删除的操作。
对于插入操作，假设还是在 $Node_1$ 和 $Node_2$ 之间插入 $Node_3$，那么过程是这样的：创建 $Node_3$ ，之后 $Node_1$ 的后继指向 $Node_3$ ，$Node_3$ 的前驱指向 $Node_1$ 、后继指向 $Node_2$，$Node_2$ 的前驱指向 $Node_3$。
对于删除操作，假设删除 $Node_1$ 和 $Node_2$ 之间的 $Node_3$，那么将 $Node_1$ 的后继指向 $Node_2$，$Node_2$ 的前驱指向 $Node_1$ 即可。

模拟链表 – I

B3631 单向链表

数组模拟的链表，也叫静态链表。由于这道题过于简单，因此我们只需要一个 *next* 数组，用下标表示键值，用对应的元素表示指向的指针即可：

```
int next[N];
void solve(void) {
    next[1] = 0;
    int Q; std::cin >> Q;
    while(Q--) {
        int op, x, y; std::cin >> op;
        if(op == 1) {
            std::cin >> x >> y;
            next[y] = next[x];
            next[x] = y;
        } else if(op == 2) {
            std::cin >> x;
            std::cout << next[x] << std::endl;
        } else {
```

模拟链表 – II

B4324 【模板】双向链表

本题作为双向链表的模版题有些特殊，因为一开始所有的数已经练成了一个链表。因此进行操作一和操作二的时候，我们要在 $Node_y$ 的左边或者右边插入 $Node_x$ 的前提是如果 $Node_x$ 存在于链表上，我们要先把它从链表中移出来，也就是做一次删除操作。所以我们可以用结构题来存储节点，每个节点有前驱、后继、是否被删除三个属性，用下标来保存键值。

```
struct Node {
    int pre, next;
    bool deleted;
}l[N];
void solve(void) {
    int n, m; std::cin >> n >> m;
    for(int i = 1; i <= n; i++) {
        l[i].pre = i - 1;
        l[i].next = i + 1;
    }
    l[1].pre = 0, l[n].next = 0; l[0].next = 1;
```

模拟链表 – III

```
while(m--) {
    int op, x, y; std::cin >> op;
    if(op == 1) {
        std::cin >> x >> y;
        if(x == y) continue;
        if(!l[x].deleted) {
            l[l[x].pre].next = l[x].next;
            l[l[x].next].pre = l[x].pre;
            l[x].deleted = true;
        }
        int p = l[y].pre;
        l[p].next = x;l[x].pre = p;
        l[x].next = y;l[y].pre = x;
        l[x].deleted = false;
    } else if(op == 2) {
        std::cin >> x >> y;
        if(x == y) continue;
        if(!l[x].deleted) {
            l[l[x].pre].next = l[x].next;
            l[l[x].next].pre = l[x].pre;
            l[x].deleted = true;
```

模拟链表 – IV

```
            }
            int p = l[y].next;
            l[x].next = p;l[p].pre = x;
            l[y].next = x;l[x].pre = y;
            l[x].deleted = false;
        } else {
            std::cin >> x;
            if(!l[x].deleted) {
                l[l[x].pre].next = l[x].next;
                l[l[x].next].pre = l[x].pre;
                l[x].deleted = true;
            }
        }
}int cur = l[0].next;
if(cur == 0) {
    std::cout << "Empty!" << std::endl;
    return;
}
while(cur) {
    std::cout << cur << " "; cur = l[cur].next;
} std::cout << std::endl;
```

STL 中的链表 – I

在 STL 中提供了双链表容器 *list* [1]，支持以下操作：
$L.begin(), L.end()$ *list* 的首、尾迭代器。
$L.push_back(x)$ 在链表末端插入一个元素 x 。
$L.front(), L.back()$ 返回链表的首、尾迭代器。
$L.splice(x, A, y)$ 从链表 A 中将第 x 个节点转移到链表 L 的 y 号节点后。$L.erase(x)$ 将链表 L 中的 x 号节点删除。

[1]其实还提供了单链表容器 *forward_list* ，其使用方法和 list 基本一致，感兴趣的同学可以自行查阅资料。

STL 中的链表 – II

对于上面的题目，我们可以使用 *List* 解决。

```
void solve(void) {
    int n, m; std::cin >> n >> m;
    std::list<int> L;
    std::vector<bool> deleted(n + 1, false);
    std::vector<std::list<int>::iterator> pos(n + 1);
    for(int i = 1; i <= n; i++) {
        L.push_back(i);
        auto it = L.end();
        --it; pos[i] = it;
    }
    while(m--) {
        int op, x, y;
        std::cin >> op;
        if(op == 1) {
            std::cin >> x >> y;
            if(x == y) continue;
            L.splice(pos[y], L, pos[x]);
```

STL 中的链表 – III

```
            } else if(op == 2) {
                std::cin >> x >> y;
                if(x == y) continue;
                auto it = pos[y]; it++;
                L.splice(it, L, pos[x]);
            } else {
                std::cin >> x;
                if(!deleted[x]) {
                    pos[x] = L.erase(pos[x]);
                    deleted[x] = true;
                }
            }
        }
        if(L.empty()) {
            std::cout << "Empty!" << std::endl;
            return;
        }
        for(auto i : L) {
            std::cout << i << " ";
        }
    }
```

习题

线性表是数据结构的基础，之后我们学习的很多内容都需要依靠今天学到的内容才能实现，大家一定要勤加练习。

P3156 【深基 15. 例 1】询问学号
P3613 【深基 15. 例 2】寄包柜
P1449 后缀表达式
P1996 约瑟夫问题
P1160 队列安排
P1540 [NOIP 2010 提高组] 机器翻译
P2058 [NOIP 2016 普及组] 海港
P1241 括号序列
P4387 【深基 15. 习 9】验证栈序列
P2234 [HNOI2002] 营业额统计