第 2 章 顺序表和 vector

1. 顺序表的概念

（可以用 ppt 展示，课件里面总结）

1. 线性表的定义

线性表是 n 个具有相同特性的数据元素的有序序列。

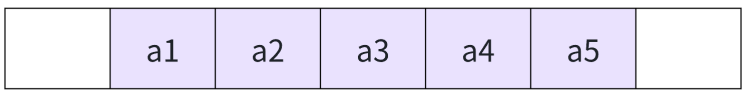
线性表在逻辑上可以想象成是连续的一条线段，线段上有很多个点，比如下图：

因此，线性表是一个比较简单和基础的数据结构。

2. 线性表的顺序存储 - 顺序表

线性表的顺序存储就是顺序表。

如果下图中的方格代表内存中的存储单元，那么存储顺序表中 $a_1 \sim a_5$ 这 5 个元素就是放在连续的位置上：



大家会发现，这不就是用一个数组把这些元素存储起来了嘛？是的，顺序表就是通过数组来实现的。

2. 顺序表的模拟实现

约定：往后实现各种数据结构的时候，如果不做特殊说明，默认里面存储的就是 int 类型的数据。

2.1 顺序表的实现方式

（可以用 ppt 展示，课件里面总结）

按照数组的申请方式，有以下两种实现方式：

- 数组采用静态分配，此时的顺序表称为静态顺序表。
- 数组采用动态分配，此时的顺序表称为动态顺序表。

静态分配就是直接向内存申请**一大块连续的区域**，然后将需要存放的数组放在这一大块连续的区域上。

动态分配就是**按需所取**。按照需要存放的数据的数量，**合理的申请大小合适的空间来存放数据**。

实现方式	优点	缺点
静态分配	1. 不需要动态管理内存，代码书写上会比较方便。 2. 没有动态管理内存中申请以及释放空间的时间开销。	1. 一旦空间占满，新来的数据就会溢出。 2. 如果为了保险而申请很大的空间，数据量小的情况下，会浪费很多空间。
动态分配	1. 自由的分配空间。数据量小，就用申请小内存；数据量大，就在原有的基础上扩容。	1. 由于需要动态管理内存，代码书写上会比较麻烦。 2. 动态内存的过程中会经常涉及扩容，而扩容需要**申请空间，转移数据，释放空间**。这些操作会有大量的时间消耗。

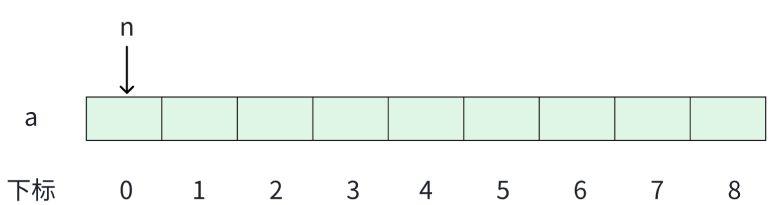
通过两者对比会发现，**并没有一种实现方式就是绝对完美的**。想要书写方便以及运行更快，就要承担空间不够或者空间浪费的情况；想要空间上合理分配，就要承担时间以及代码书写上的消耗。

在后续的学习中，会经常看到各种情况的对比。这就要求我们掌握各种数据结构的特点，从而在解决实际问题的时候，选择一个合适的数据结构。

在算法竞赛中，我们主要关心的其实是时间开销，空间上是基本够用的。因此，定义一个超大的静态数组来解决问题是完全可以接受的。因此，关于顺序表，采用的就是**静态实现**的方式。

2.2 创建

（可以用 ppt 展示，课件里面总结）



```cpp
const int N = 1e6 + 10; // 定义静态数组的最大长度

int a[N], n; // 直接创建一个大数组来实现顺序表， n 表示当前有多少个元素
```

2.3 添加一个元素

（可以用 ppt 展示，课件里面总结）

2.3.1 尾插

代码实现：

```cpp
//尾插
void push_back(int x)
{
    a[++n] = x; // 下标为 0 的位置空出来

    // 这样操作一般根据个人习惯，也可以从 0 开始计数
    // 不过有些问题从 1 计数，处理起来可以不用考虑边界情况
    // 后续学习更深的算法的时候，就会感受到
}

// 思考，这个函数有 bug 么？
// 1. 数组存满了，就不能再存了！

// 当时，我们一般不去管这个判断怎么写，因为我们在调用的时候，自己会判断合不合法，如果不合法，我们是
// 不会调用的。
```

时间复杂度：

> 直接放在后面即可，时间复杂度为 $O(1)$。

2.3.2 头插

（可以用 ppt 展示，课件里面总结）

代码实现：

```cpp
// 头插
```

```
void push_front(int x)
{
    // 要把所有的元素全部右移一位，然后再放到头部位置
    for(int i = n; i >= 1; i--) // 循环的顺序是否可以改变?
    {
        a[i + 1] = a[i];
    }
    a[1] = x; // 把 x 放在首位
    n++; // 不要忘记总个数 +1
}

// 思考，这个函数有 bug 么?
```

时间复杂度：

> 由于需要将所有元素右移一位，时间复杂度为 $O(N)$。

2.3.3 任意位置插入

（可以用 ppt 展示，课件里面总结）

代码实现：

```
// 任意位置插入 - 在位置 p 处，插入一个 x
void insert(int p, int x)
{
    for(int i = n; i >= p; i--) // 注意顺序不要颠倒
    {
        a[i + 1] = a[i];
    }
    a[p] = x;
    n++; // 不要忘记总个数 +1
}

// 思考，这个函数有 bug 么?
// p 这个位置也要合法!
```

时间复杂度：

> 最坏情况下需要数组中所有元素右移，时间复杂度为 $O(N)$。

2.4 删除一个元素

（可以用 ppt 展示，课件里面总结）

2.4.1 尾删

代码实现：

```
void pop_back()
{
    n--;
}

// 思考，这个函数有 bug 么？
// 删除之前，要判断一下顺序表里面有没有元素
```

时间复杂度：

> 显然是 $O(1)$。

2.4.2 头删

代码实现：

```
// 头删
void pop_front()
{
    // 把所有元素向前移动一位
    for(int i = 2; i <= n; i++) // 顺序是否能颠倒？
    {
        a[i - 1] = a[i];
    }
    n--; // 总个数 -1
}

// 思考，这个函数有 bug 么？
```

时间复杂度：

> 需要所有元素整体左移，时间复杂度为 $O(N)$。

2.4.3 任意位置删除

代码实现：

```
// 任意位置删除
void erase(int p)
{
    for(int i = p + 1; i <= n; i++)
    {
        a[i - 1] = a[i];
    }
    n--; // 总个数 -1
}

// 思考，这个函数有 bug 么?
// p 的位置要是合法的位置 [1,n]
```

时间复杂度：

> 最坏情况下，所有元素都需要左移，时间复杂度为 $O(N)$。

2.5 查找元素

（可以用 ppt 展示，课件里面总结）

2.5.1 按值查找

代码实现：

```
// 查找这个数第一次出现的位置，找不到返回 0
int find(int x)
{
    for(int i = 1; i <= n; i++)
    {
        if(a[i] == x) return i;
    }
    return 0;
}
```

时间复杂度：

> 最坏情况下需要遍历整个数组，时间复杂度为 $O(N)$。

2.5.2 按位查找

代码实现：

```
// 返回 p 位置的数
int at(int p)
{
    return a[p];
}

// 思考，这个函数有 bug 么？
// p 的位置应该是合法的 [1,n]
```

时间复杂度：

> 这就是顺序表**随机存取**的特性，只要给我一个下标，就能快速访问到该元素。
>
> 时间复杂度为 $O(1)$。

2.6 修改元素

代码实现：

```
// 把 p 位置的数修改成 x
void change(int p, int x)
{
    a[p] = x;
}

// 思考，这个函数有 bug 么？
// 位置 p 要是合法的才行
```

时间复杂度：

> 这就是顺序表**随机存取**的特性，只要给我一个下标，就能快速访问到该元素。

> 时间复杂度为 $O(1)$。

2.7 清空顺序表

代码实现：

```
// 清空顺序表
void clear()
{
    n = 0;
}
```

时间复杂度：

> 要注意，我们自己实现的简单形式是 $O(1)$。
>
> 但是，严谨的方式应该是 $O(N)$。

2.8 所有测试代码

```
#include <iostream>

using namespace std;

const int N = 1e6 + 10; // 根据实际情况而定

// 创建顺序表
// int a[N]; // 用足够大的数组来模拟顺序表
// int n; // 标记顺序表里面有多少个元素

// 需要多个顺序表，才能解决问题
int a1[N], n1;
int a2[N], n2;
int a3[N], n3;


// 打印顺序表
void print()
{
    for(int i = 1; i <= n; i++)
```

```
21      {
22          cout << a[i] << " ";
23      }
24      cout << endl << endl;
25  }
26
27  // 尾插
28  void push_back(int a[], int& n, int x)
29  {
30      a[++n] = x;
31  }
32
33  void test()
34  {
35      push_back(a1, n1, 1);
36      push_back(a3, n3, 2);
37  }
38
39
40  // 头插
41  void push_front(int x)
42  {
43      // 1. 先把 [1, n] 的元素统一向后移动一位
44      for(int i = n; i >= 1; i--)
45      {
46          a[i + 1] = a[i];
47      }
48
49      // 2. 把 x 放在表头
50      a[1] = x;
51      n++; // 元素个数 +1
52  }
53
54  // 在任意位置插入
55  void insert(int p, int x)
56  {
57      // 1. 先把 [p, n] 的元素统一向后移动一位
58      for(int i = n; i >= p; i--)
59      {
60          a[i + 1] = a[i];
61      }
62
63      a[p] = x;
64      n++;
65  }
66
67  // 尾删
```

```
68 void pop_back()
69 {
70     n--;
71 }
72
73 // 头删
74 void pop_front()
75 {
76     // 1. 先把 [2, n] 区间内的所有元素，统一左移一位
77     for(int i = 2; i <= n; i++)
78     {
79         a[i - 1] = a[i];
80     }
81     n--;
82 }
83
84 // 任意位置删除
85 void erase(int p)
86 {
87     // 把 [p + 1, n] 的元素，统一左移一位
88     for(int i = p + 1; i <= n; i++)
89     {
90         a[i - 1] = a[i];
91     }
92
93     n--;
94 }
95
96 // 按值查找
97 int find(int x)
98 {
99     for(int i = 1; i <= n; i++)
100    {
101        if(a[i] == x) return i;
102    }
103
104    return 0;
105 }
106
107 // 按位查找
108 int at(int p)
109 {
110    return a[p];
111 }
112
113 // 按位修改
114 int change(int p, int x)
```

```
115 {
116     a[p] = x;
117 }
118 
119 // 清空操作
120 void clear()
121 {
122     n = 0;
123 }
124 
125 int main()
126 {
127     // 测试尾插
128     push_back(2);
129     print();
130     push_back(5);
131     print();
132     push_back(1);
133     print();
134     push_back(3);
135     print();
136 
137     // 测试头插
138     push_front(10);
139     print();
140 
141     // 测试任意位置插入
142     insert(3, 0);
143     print();
144 
145     // 测试尾删
146     // cout << "尾删: " << endl;
147     // pop_back();
148     // print();
149     // pop_back();
150     // print();
151 
152     // pop_front();
153     // pop_front();
154     // print();
155 
156     // 测试任意位置删除
157     // cout << "任意位置删除: " << endl;
158     // erase(3);
159     // print();
160     // erase(2);
161     // print();
```

```
    // erase(4);
    // print();

    for(int i = 1; i <= 10; i++)
    {
        cout << "查找" << i << ": ";
        cout << find(i) << endl;
    }

    return 0;
}
```

3. 封装静态顺序表

思考一下，如果实际情况需要特别多的顺序表来解决问题，上述的写法有什么问题么？

> 如果需要两个及以上的顺序表：
>
> - 定义数组的时候就需要定义多个 $a_1, a_2 \ldots$，还需要配套的 $n_1, n_2 \ldots$ 来描述顺序表的大小；
> - 在调用 $push\_back$ 等函数的时候，还需要将 a_1 和 n_1 作为参数传进去，不然不知道修改的是哪一个顺序表；
> - 传参的时候还需要注意传引用，因为顺序表的大小有可能改变，我们要修改 n_i 的值。

可见，如果需要多个顺序表时，上述代码虽然能很大程度上继续复用，但还是比较麻烦。那么应该如何解决这个问题呢？

利用 C++ 中的**结构体和类**把我们实现的顺序表**封装**起来，就能简化操作。

```
#include <iostream>

using namespace std;

const int N = 1e5 + 10;

// 将顺序表的创建以及增删查改封装在一个类中
class SqList
{
    int a[N];
    int n;

public:
    // 构造函数，初始化
```

```
15      SqList()
16      {
17          n = 0;
18      }
19
20      // 尾插
21      void push_back(int x)
22      {
23          a[++n] = x;
24      }
25
26      // 尾删
27      void pop_back()
28      {
29          n--;
30      }
31
32      // 打印
33      void print()
34      {
35          for(int i = 1; i <= n; i++)
36          {
37              cout << a[i] << " ";
38          }
39          cout << endl;
40      }
41  };
42
43  int main()
44  {
45      SqList s1, s2; // 创建了两个顺序表
46
47      for(int i = 1; i <= 5; i++)
48      {
49          // 直接调用 s1 和 s2 里面的 push_back
50          s1.push_back(i);
51          s2.push_back(i * 2);
52      }
53
54      s1.print();
55      s2.print();
56
57      for(int i = 1; i <= 2; i++)
58      {
59          s1.pop_back();
60          s2.pop_back();
61      }
```

```
62
63      s1.print();
64      s2.print();
65
66      return 0;
67  }
```

用类和结构体将代码进行封装，能够很大程度上减少重复的操作，使代码的复用率大大提升。当然，封装的好处不仅如此，更多的优势可以在就业课的学习中继续感受。

📌 注意：

- 为什么这里讲了封装？

  最重要的原因是想让大家知道，接下来我们要学习的 STL 为什么可以通过 "." 调用各种各样的接口。

- 为什么我们后面不做封装了？

  a. 我们做题如果用到某个数据结构，一般仅需要一个，最多两个，所以没有必要封装。因为封装之后，还要去写 xxx.xxx，比较麻烦；

  b. 如果要用到多个相同的数据结构，那么推荐使用 STL，更加方便。

4. 动态顺序表 - vector

动态顺序表就不带着实现了，因为涉及空间申请和释放的 `new` 和 `delete` 效率不高，在算法竞赛中使用会有超时的风险。而且实现一个动态顺序表代码量很大，我们不可能在竞赛中傻乎乎的实现一个动态顺序表来解决问题。

> 但是我们要明白一点，竞赛代码和工程代码是不一样的。在我们以后工作写项目的时候，还是需要动态申请空间的方式。因此，希望大家还是需要掌握动态顺序表的实现。
>
> 比特会在就业课的数据结构中，讲解动态顺序表的实现方式，这里就不再赘述了。

如果需要用动态顺序表，有更好的方式： `C++` 的 `STL` 提供了一个已经封装好的容器 - `vector` ，有的地方也叫作可变长的数组。 `vector` 的底层就是一个会自动扩容的顺序表，其中创建以及增删查改等等的逻辑已经实现好了，并且也完成了封装。

接下来就重点学习 `vector` 的使用。

4.1 创建 vector

```cpp
#include <vector> // 头文件

using namespace std;

const int N = 20;

struct node
{
    int a, b, c;
};

// 1. 创建
void init()
{
    vector<int> a1; // 创建一个空的可变长数组
    vector<int> a2(N); // 指定好了一个空间，大小为 N
    vector<int> a3(N, 10); // 创建一个大小为 N 的 vector，并且里面的所有元素都是 10
    vector<int> a4 = {1, 2, 3, 4, 5}; // 使用列表初始化，创建一个 vector

    // <> 里面可以放任意的类型，这就是模板的作用，也是模板强大的地方
    // 这样，vector 里面就可以放我们接触过的任意数据类型，甚至是 STL
    vector<string> a5; // 放字符串
    vector<node> a6; // 放一个结构体
    vector<vector<int>> a7; // 甚至可以放一个自己，当成一个二维数组来使用。并且每一维都是可变的

    vector<int> a8[N]; // 创建 N 个 vector
}
```

4.2 size / empty

1. `size`：返回实际元素的个数；
2. `empty`：返回顺序表是否为空，因此是一个 `bool` 类型的返回值。
    a. 如果为空：返回 true
    b. 否则，返回 false

时间复杂度：$O(1)$。

测试代码：

```
// 2. size
void test_size()
{
    // 创建一个一维数组
    vector<int> a1(6, 8);
    for(int i = 0; i < a1.size(); i++)
    {
        cout << a1[i] << " ";
    }
    cout << endl << endl;

    // 创建一个二维数组
    vector<vector<int>> a2(3, vector<int>(4, 5));
    for(int i = 0; i < a2.size(); i++)
    {
        // 这里的 a2[i] 相当于一个 vector<int> a(4, 5)
        for(int j = 0; j < a2[i].size(); j++)
        {
            cout << a2[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl << endl;
}
```

4.3 begin / end

1. `begin`：返回起始位置的迭代器（左闭）；
2. `end`：返回终点位置的下一个位置的迭代器（右开）；

利用迭代器可以访问整个 `vector`，存在迭代器的容器就可以使用范围 `for` 遍历。

测试代码：

```
// 3. begin/end
void test_it()
{
    vector<int> a(10, 1);

    // 迭代器的类型是 vector<int>::iterator，但是一般使用 auto 简化
    for(auto it = a.begin(); it != a.end(); it++)
```

```
    {
        cout << *it << " ";
    }
    cout << endl << endl;

    // 使用语法糖 - 范围 for 遍历
    for(auto x : a)
    {
        cout << x << " ";
    }
    cout << endl << endl;
}
```

4.4 push_back / pop_back

1. `push_back`：尾部添加一个元素
2. `pop_back`：尾部删除一个元素

当然还有 `insert` 与 `erase`。不过由于时间复杂度过高，尽量不使用。

时间复杂度：$O(1)$。

测试代码：

```
// 如果不加引用，会拷贝一份，时间开销很大
void print(vector<int>& a)
{
    for(auto x : a)
    {
        cout << x << " ";
    }
    cout << endl;
}

// 4. 添加和删除元素
void test_io()
{
    vector<int> a;
    // 尾插 1 2 3 4 5
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);
```

```
    print(a);

    // 尾删 3 次
    a.pop_back();
    a.pop_back();
    a.pop_back();
    print(a);
}
```

4.5 front / back

1. `front`：返回首元素；
2. `back`：返回尾元素；

时间复杂度：$O(1)$。

测试代码：

```
// 5. 首元素和尾元素
void test_fb()
{
    vector<int> a(5);
    for(int i = 0; i < 5; i++)
    {
        a[i] = i + 1;
    }

    cout << a.front() << " " << a.back() << endl;
}
```

4.6 resize

- 修改 `vector` 的大小。
- 如果大于原始的大小，多出来的位置会补上默认值，一般是 `0`。
- 如果小于原始的大小，相当于把后面的元素全部删掉。

时间复杂度：$O(N)$。

测试代码：

```cpp
// 如果不加引用，会拷贝一份，时间开销很大
void print(vector<int>& a)
{
    for(auto x : a)
    {
        cout << x << " ";
    }
    cout << endl;
}

// 6. resize
void test_resize()
{
    vector<int> a(5, 1);
    a.resize(10); // 扩大
    print(a);

    a.resize(3); // 缩小
    print(a);
}
```

4.7 clear

- 清空 `vector`

底层实现的时候，会遍历整个元素，一个一个删除，因此时间复杂度：$O(N)$。

测试代码：

```cpp
// 如果不加引用，会拷贝一份，时间开销很大
void print(vector<int>& a)
{
    for(auto x : a)
    {
        cout << x << " ";
    }
    cout << endl;
}

// 7. clear
void test_clear()
{
    vector<int> a(5, 1);
    print(a);
```

```
    a.clear();
    cout << a.size() << endl;
    print(a);
}
```

💡 vector 内封装的接口其实还有很多，比如：

- insert：在指定位置插入一个元素；
- erase：删除指定位置的元素；
- ……

但是，其余的接口要么不常用；要么时间复杂度较高，比如 insert 和 erase，算法竞赛中不能频繁的调用。因此，在这里以及往后，介绍的都是常用以及高效的接口。

另外，再 https://cplusplus.com/ 里，可以查阅各种容器中的接口，以及使用方式。

4.8 所有测试代码

```
#include <iostream>
#include <vector>

using namespace std;

const int N = 10;

struct node
{
    int a, b;
    string s;
};

void print(vector<int>& a)
{
    // 利用 size 来遍历
    // for(int i = 0; i < a.size(); i++)
    // {
    //     cout << a[i] << " ";
    // }
    // cout << endl;

```

```cpp
    // 利用迭代器来遍历 - 迭代器类型 vector<int>::iterator
    // for(auto it = a.begin(); it != a.end(); it++)
    // {
    //     cout << *it << " ";
    // }
    // cout << endl;

    // 使用语法糖 - 范围 for
    for(auto x : a)
    {
        cout << x << " ";
    }
    cout << endl;
}

int main()
{
    // 1. 创建 vector
    vector<int> a1; // 创建了一个名字为 a1 的空的可变长数组，里面都是 int 类型的数据
    vector<int> a2(N); // 创建了一个大小为 10 的可变长数组，里面的值默认都是 0
    vector<int> a3(N, 2); // 创建了一个大小为 10 的可变长数组，里面的值都初始化为 2
    vector<int> a4 = {1, 2, 3, 4, 5}; // 初始化列表的创建方式

    // <> 里面可以存放任意的数据类型，这就体现了模板的作用，也体现了模板的强大之处
    // 这样，vector里面就可以存放我们见过的所有的数据类型，甚至是 STL 本身
    vector<string> a5; // 存字符串
    vector<node> a6; // 存结构体
    vector<vector<int>> a7; // 创建了一个二维的可变长数组

    vector<int> a8[N]; // 创建了一个大小为 N 的 vector 数组
    // int a[N];

    // 2. size / empty
    // print
    // print(a2);
    // print(a3);
    // print(a4);

    // if(a2.empty()) cout << "空" << endl;
    // else cout << "不空" << endl;

    // 3. begin / end
    // print
    // print(a2);
    // print(a3);
    // print(a4);

```

```
    // 4. 尾插以及尾删
    // for(int i = 0; i < 5; i++)
    // {
    //     a1.push_back(i);
    //     print(a1);
    // }

    // while(!a1.empty())
    // {
    //     print(a1);
    //     a1.pop_back();
    // }

    // 5. front / back
    // cout << a4.front() << " " << a4.back() << endl;

    // 6. resize
    vector<int> aa(5, 1);
    print(aa);

    // 扩大成 10
    aa.resize(10);
    print(aa);

    // 缩小成 3
    aa.resize(3);
    print(aa);

    // 7. clear
    cout << aa.size() << endl;
    aa.clear();
    cout << aa.size() << endl;

    return 0;
}
```

5. 算法题

5.1 询问学号

> **题目来源：** 洛谷
>
> **题目链接：** P3156 【深基15.例1】询问学号
>
> **难度系数：** ★

【题目描述】

> 有 $n(n \leq 2 \times 10^6)$ 名同学陆陆续续进入教室。我们知道每名同学的学号（在 1 到 10^9 之间），按进教室的顺序给出。上课了，老师想知道第 i 个进入教室的同学的学号是什么（最先进入教室的同学 $i=1$），询问次数不超过 10^5 次。

【输入描述】

> 第一行 2 个整数 n 和 m，表示学生个数和询问次数。
>
> 第二行 n 个整数，表示按顺序进入教室的学号。
>
> 第三行 m 个整数，表示询问第几个进入教室的同学。

【输出描述】

> 输出 m 个整数表示答案，用换行隔开。

【示例一】

> 输入：
>
> 10 3
>
> 1 9 2 60 8 17 11 4 5 14
>
> 1 5 9
>
> 输出：
>
> 1
>
> 8
>
> 5

【解法】

直接用 `vector` 或者数组模拟即可。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 2e6 + 10;

int n, m;
vector<int> a(N);

int main()
```

```cpp
{
    cin >> n >> m;

    for(int i = 1; i <= n; i++) cin >> a[i];

    while(m--)
    {
        int x; cin >> x;
        cout << a[x] << endl;
    }

    return 0;
}
```

5.2 寄包柜

> **题目来源：** 洛谷
>
> **题目链接：** P3613 【深基15.例2】寄包柜
>
> **难度系数：** ★

【题目描述】

> 超市里有 $n(1 \le n \le 10^5)$ 个寄包柜。每个寄包柜格子数量不一，第 i 个寄包柜有 $a_i(1 \le a_i \le 10^5)$ 个格子，不过我们并不知道各个 a_i 的值。对于每个寄包柜，格子编号从 1 开始，一直到 a_i。
>
> 现在有 $q(1 \le q \le 10^5)$ 次操作：
>
> - `1 i j k`：在第 i 个柜子的第 j 个格子存入物品 $k(0 \le k \le 10^9)$。当 $k = 0$ 时说明清空该格子。
> - `2 i j`：查询第 i 个柜子的第 j 个格子中的物品是什么，保证查询的柜子有存过东西。
>
> 已知超市里共计不会超过 10^7 个寄包格子，a_i 是确定然而未知的，但是保证一定不小于该柜子存物品请求的格子编号的最大值。当然也有可能某些寄包柜中一个格子都没有。

【输入描述】

> 第一行 2 个整数 n 和 q，寄包柜个数和询问次数。
>
> 接下来 q 个行，每行有若干个整数，表示一次操作。

【输出描述】

> 对于查询操作时，输出答案，以换行隔开。

【示例一】

> 输入：
>
> 5 4
>
> 1 3 10000 118014
>
> 1 1 1 1
>
> 2 3 10000
>
> 2 1 1
>
> 输出：
>
> 118014
>
> 1

【解法】

如果用二维数组来模拟，需要开 $10^5 \times 10^5$ 大小的数组，空间会超。但是格子的总数量是 10^7 ，用数组模拟是完全够用的。因此可以用动态扩容的数组，创建 10^5 个 $vector$ 来模拟。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 1e5 + 10;

int n, q;
vector<int> a[N]; // 创建 N 个柜子

int main()
{
    cin >> n >> q;

    while(q--)
    {
        int op, i, j, k;
        cin >> op >> i >> j;

        if(op == 1) // 存
        {
            cin >> k;
            if(a[i].size() <= j)
            {
```

```
25                  // 扩容
26                  a[i].resize(j + 1);
27              }
28
29              a[i][j] = k;
30          }
31          else // 查询
32          {
33              cout << a[i][j] << endl;
34          }
35      }
36
37
38      return 0;
39  }
```

5.3 移动零

> **题目来源：** 力扣
>
> **题目链接：** 283. 移动零
>
> **难度系数：** ★

【题目描述】

> 给定一个数组 $nums$，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
>
> 请注意，必须在不复制数组的情况下原地对数组进行操作。

【示例一】

> 输入：
>
> nums = [0,1,0,3,12]
>
> 输出：
>
> [1,3,12,0,0]

【解法】

在本题中，我们可以用一个 cur 指针来扫描整个数组，另一个 $dest$ 指针用来记录非零数序列的最后一个位置。根据 cur 在扫描的过程中，遇到的不同情况，分类处理，实现数组的划分。

在 cur 遍历期间，使 $[0, dest]$ 的元素全部都是非零元素，$[dest + 1, cur - 1]$ 的元素全是零。

【参考代码】

```cpp
class Solution
{
public:
    void moveZeroes(vector<int>& nums)
    {
        for(int i = 0, cur = -1; i < nums.size(); i++)
        {
            if(nums[i]) // 非0元素
            {
                swap(nums[++cur], nums[i]);
            }
        }
    }
};
```

5.4 颜色分类

> **题目来源：** 力扣
>
> **题目链接：** 75. 颜色分类
>
> **难度系数：** ★

【题目描述】

> 给定一个包含红色、白色和蓝色、共 n 个元素的数组 $nums$ ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
>
> 我们使用整数 0 、 1 和 2 分别表示红色、白色和蓝色。
>
> 必须在不使用库的 $sort$ 函数的情况下解决这个问题。

【示例一】

> 输入：
>
> nums = [2,0,2,1,1,0]
>
> 输出：
>
> [0,0,1,1,2,2]

【解法】

类比数组分两块的算法思想，这里是将数组分成三块，那么我们可以再添加一个指针，实现数组分三块。

设数组大小为 n ，定义三个指针 $left, cur, right$:

- $left$ ：用来标记 0 序列的末尾，因此初始化为 -1 ；

- cur：用来扫描数组，初始化为 0；
- $right$：用来标记 2 序列的起始位置，因此初始化为 n。

在 cur 往后扫描的过程中，保证：

- $[0, left]$ 内的元素都是 0；
- $[left + 1, cur - 1]$ 内的元素都是 1；
- $[cur, right - 1]$ 内的元素是待定元素；
- $[right, n]$ 内的元素都是 2。

【参考代码】

```
class Solution
{
public:
    void sortColors(vector<int>& nums)
    {
        int n = nums.size();
        int left = -1, right = n, i = 0;
        while(i < right)
        {
            if(nums[i] == 0) swap(nums[++left], nums[i++]);
            else if(nums[i] == 1) i++;
            else swap(nums[--right], nums[i]);
        }
    }
};
```

5.5 合并两个有序数组

> **题目来源：** 力扣
>
> **题目链接：** 合并两个有序数组
>
> **难度系数：** ★

【题目描述】

> 给你两个按非递减顺序排列的整数数组 $nums1$ 和 $nums2$，另有两个整数 m 和 n，分别表示 $nums1$ 和 $nums2$ 中的元素数目。
>
> 请你合并 $nums2$ 到 $nums1$ 中，使合并后的数组同样按非递减顺序排列。

> 注意：最终，合并后数组不应由函数返回，而是存储在数组 $nums1$ 中。为了应对这种情况，$nums1$ 的初始长度为 $m+n$，其中前 m 个元素表示应合并的元素，后 n 个元素为 0 ，应忽略。$nums2$ 的长度为 n 。

【示例一】

> 输入：
>
> nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
>
> 输出：
>
> [1,2,2,3,5,6]

【解法】

解法一：利用辅助数组（需要学会，归并排序的核心步骤）

可以创建一个辅助数组，然后用两个指针分别指向两个数组。每次拿出一个较小的元素放在辅助数组中，直到把所有元素全部放在辅助数组中。最后把辅助数组的结果覆盖到 $nums1$ 中。

解法二：原地修改（本题的最优解）

与解法一的核心思想是一样的。

由于第一个数组的空间本来就是 $n+m$ 个，所以我们可以直接把最终结果放在 $nums1$ 中。为了不覆盖未遍历到的元素，定义两个指针指向两个数组的末尾，从后往前扫描。每次拿出较大的元素也是从后往前放在 $nums1$ 的后面，直到把所有元素全部放在 $nums1$ 中。

通过这道题想告诉大家，在我们的算法竞赛中，只要你的空间不超，想用多少辅助数组就用多少辅助数组，怎么方便怎么来。但是在面试中，还是需要注意挖掘最优解。

【参考代码】

```
class Solution
{
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
    {
        // 解法一：利用辅助数组
        vector<int> tmp(m + n);

        int cur = 0, cur1 = 0, cur2 = 0;

        while(cur1 < m && cur2 < n)
        {
```

```
            if(nums1[cur1] <= nums2[cur2]) tmp[cur++] = nums1[cur1++];
            else tmp[cur++] = nums2[cur2++];
        }

        while(cur1 < m) tmp[cur++] = nums1[cur1++];
        while(cur2 < n) tmp[cur++] = nums2[cur2++];

        for(int i = 0; i < n + m; i++) nums1[i] = tmp[i];
    }
};


class Solution
{
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
    {
        // 解法二：原地合并
        int cur1 = m - 1, cur2 = n - 1, cur = m + n - 1;

        while(cur1 >= 0 && cur2 >= 0)
        {
            if(nums1[cur1] >= nums2[cur2]) nums1[cur--] = nums1[cur1--];
            else nums1[cur--] = nums2[cur2--];
        }

        while(cur2 >= 0) nums1[cur--] = nums2[cur2--];
    }
};
```

5.6 The Blocks Problem

> **题目来源：** 洛谷
>
> **题目链接：** The Blocks Problem
>
> **难度系数：** ★★

【题目描述】

> 初始时从左到右有 n 个木块，编号为 $0 \ldots n-1$,要求实现下列四种操作：
>
> - `move a onto b`：把 a 和 b 上方的木块归位，然后把 a 放到 b 上面。
> - `move a over b`：把 a 上方的木块归位，然后把 a 放在 b 所在木块堆的最上方。
> - `pile a onto b`：把 b 上方的木块归位，然后把 a 及以上的木块坨到 b 上面。

- `pile a over b`：把 a 及以上的木块坨到 b 的上面。
- 一组数据的结束标志为 `quit`，如果有非法指令（如 a 与 b 在同一堆），无需处理。

输出:所有操作输入完毕后，从左到右，从下到上输出每个位置的木块编号。

【输入描述】

第一行输入一个整数 $n(0 < n < 25)$ 表示木块的数量。

接下来每一行都表示一个操作。

最后一行是结束标志。

【输出描述】

一共 n 行，其中第 i 行输出 i :，后面输出第 i 格从下往上的木块编号，用空格隔开。

【示例一】

输入：

```
10
move 9 onto 1
move 8 over 1
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

输出：

```
0: 0
1: 1 9 2 4
2:
3: 3
4:
5: 5 8 7 6
6:
7:
8:
```

> 9:

【解法】

本质是一个模拟题，可以用 $vector$ 来模拟，注意细节问题。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 30;
typedef pair<int, int> PII;

int n;
vector<int> p[N]; // 创建 n 个放木块的槽

PII find(int x)
{
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < p[i].size(); j++)
        {
            if(p[i][j] == x)
            {
                return {i, j};
            }
        }
    }
}

void clean(int x, int y)
{
    // 把 [x, y] 以上的木块归位

    for(int j = y + 1; j < p[x].size(); j++)
    {
        int t = p[x][j];
        p[t].push_back(t);
    }
    p[x].resize(y + 1);
}

void move(int x1, int y1, int x2)
```

```
39 {
40     // 把 [x1, y1] 及其以上的木块放在 x2 上面
41 
42     for(int j = y1; j < p[x1].size(); j++)
43     {
44         p[x2].push_back(p[x1][j]);
45     }
46     p[x1].resize(y1);
47 }
48 
49 int main()
50 {
51     cin >> n;
52     // 初始化
53     for(int i = 0; i < n; i++)
54     {
55         p[i].push_back(i);
56     }
57 
58     string op1, op2;
59     int a, b;
60 
61     while(cin >> op1 >> a >> op2 >> b)
62     {
63         // 查找 a 和 b 的位置
64         PII pa = find(a);
65         int x1 = pa.first, y1 = pa.second;
66         PII pb = find(b);
67         int x2 = pb.first, y2 = pb.second;
68 
69         if(x1 == x2) continue; // 处理不合法的操作
70 
71         if(op1 == "move") // 把 a 上方归位
72         {
73             clean(x1, y1);
74         }
75         if(op2 == "onto") // 把 b 上方归位
76         {
77             clean(x2, y2);
78         }
79 
80         move(x1, y1, x2);
81     }
82 
83     // 打印
84     for(int i = 0; i < n; i++)
85     {
```

```
86          cout << i << ":";
87          for(int j = 0; j < p[i].size(); j++)
88          {
89              cout << " " << p[i][j];
90          }
91          cout << endl;
92      }
93
94      return 0;
95  }
```

6. 拓展：ACM 模式 vs 核心代码模式

6.1 ACM 模式

ACM 模式一般是竞赛和笔试面试常用的模式，就是只给你一个题目描述，外加输入样例和输出样例，不会给你任何的代码。此时，选手或者应聘者需要根据题目要求，自己完成如下任务：

1. 头文件的包含
2. main 函数的设计
3. 自己定义程序所需的变量和容器（数组、链表、哈希表、字符串等等）
4. 数据的输入（根据题目叙述控制输入数据的格式）
5. 数据的处理（各种函数接口的设计）
6. 数据的输出（根据题目叙述控制返回数据的格式）

总而言之，ACM 模式相当于给你一个空白的代码框，让你自己设计程序来解决问题。

例如：牛客网上一道简单的 ACM 模式的题：牛牛学加法

题目描述：

给你两个整数，要求输出这两个整数的和

示例：

输入 1 2

输出 3

此时右边的代码框内一片空白，因此我们需要自己设计程序来解决问题。

ACM 模式代码：

```
1  #include <stdio.h> // 自己写头文件
2
```

```c
// 自己设计函数接口
int add(int a, int b)
{
    return a + b;
}

int main() // 自己写主函数
{
    int a = 0, b = 0; // 自己定义程序所需的变量或者容器（数组）

    scanf("%d %d", &a, &b); // 自己处理数据的输入

    int c = add(a, b); // 自己设计数据的处理逻辑，以及函数的接口
    //（这里为了方便演示，因此用了函数，其实我们大可不必使用函数）

    printf("%d\n", c); // 自己处理数据的打印

    return 0;
}
```

6.2 核心代码模式

核心代码模式仅仅甩给你一个函数，我们仅需完成这个函数的功能即可。在你完成这个函数之后，后台会调用你所写的函数，进行测试。

因此，这种情况下，我们只需完成核心的函数接口，无需考虑数据的输入和输出。

例如：leetcode 上一道简单的核心代码模式的题：2235. 两整数相加

题目描述：

给你两个整数 num1 和 num2，返回这两个整数的和。

示例 1：

输入：num1 = 12, num2 = 5

输出：17

核心代码模式代码：

```c
// 题目已经把接口给你设计好了，并且只给你一个函数接口
// 你只需要实现算法的主要逻辑，然后返回即可
int sum(int num1, int num2)
{
    return num1 + num2; // 只需完成算法的核心逻辑，不用考虑数据的输入和输出
}
```

因此，核心代码模式主要是锻炼算法的逻辑，无需考虑繁琐重复的输入和输出，重点在核心算法的实现上。