数据结构与算法

第 3 讲：队列

韩文弢

清华大学

2024–2025 学年度春季学期

本讲内容

3.1 问题引入 2
3.2 队列的概念 4
3.3 队列的实现 8
3.4 队列的扩展 28
3.5 队列的应用 36
3.6 小结 45

3.1 问题引入

3.1.1 常见的操作受限的线性表

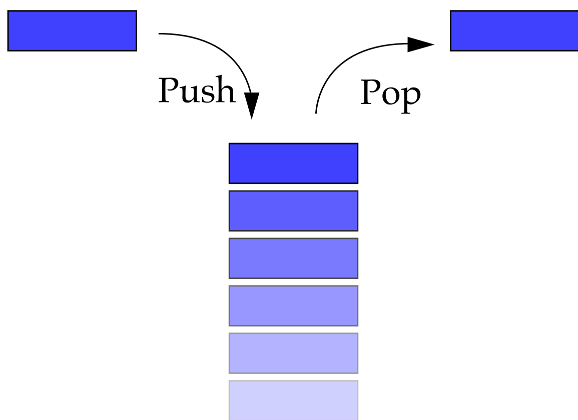


图 3.1 栈

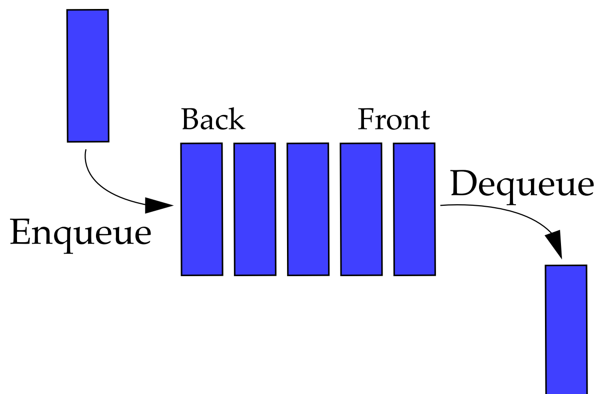


图 3.2 队列

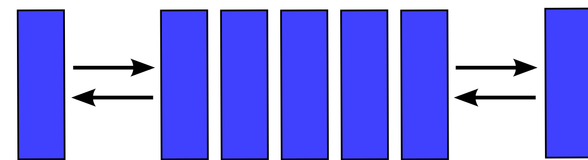


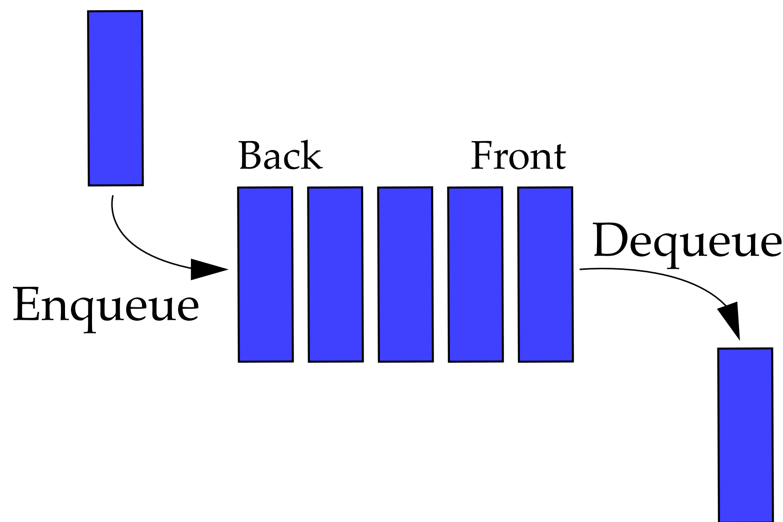
图 3.3 双端队列

3.2 队列的概念

3.2.1 队列的定义

定义 2.1.1. **队列**是一种操作受限的线性表，只能在线性表的某一端进行元素插入操作，在另一端进行元素删除操作。允许进行删除操作的一端被称为**队头**（队首），另一端被称为**队尾**（队末）。

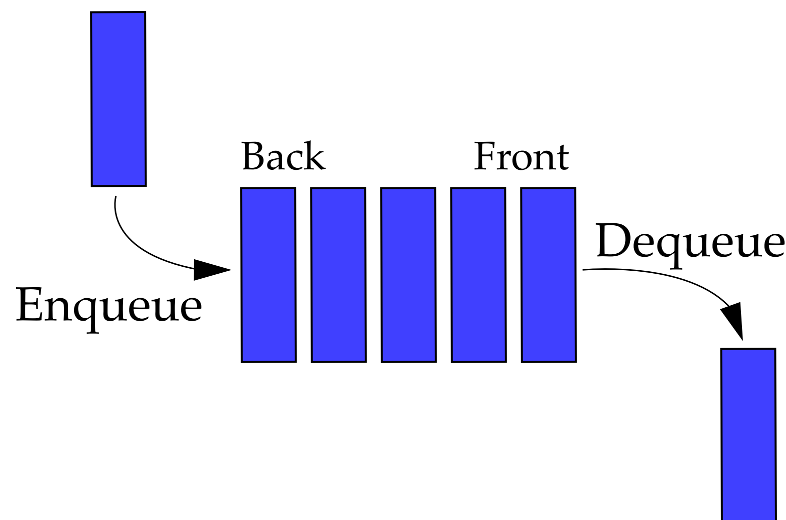
队列具有**先进先出**的特点（first in first out，FIFO），也被称为 FIFO 表。



3.2.2 队列的操作

队列的两种基本操作：

- **入队**（enqueue）：也叫进队，向队尾插入新的元素。
- **出队**（dequeue）：也叫退队，从队头删除已有元素。



3.2.3 队列的抽象数据类型

伪码 3.1 : 队列的 ADT

元素：$D = \{a_i \mid a_i \in T, i = 0, 1, 2, ..., n-1\}$，$T$ 是元素全集，n 是元素个数

关系：$R = \{(a_i, a_{i+1}) \mid a_i, a_{i+1} \in T, i = 0, 1, 2, ..., n-2\}$

基本操作：

Initialize(Q): 初始化一个空的队列 Q

Destroy(Q): 销毁队列 Q

Clear(Q): 清空队列 Q

IsEmpty(Q): 返回队列 Q 是否为空

IsFull(Q): 返回队列 Q 是否已满

Length(Q): 返回队列 Q 中的元素个数

Front(Q): 获得队列 Q 的队头元素，若队列为空则返回 $\emptyset$

Enqueue(Q, x): 将元素 x 插入队列 Q

Dequeue(Q): 从队列 Q 中删除队头元素并返回，若队列为空则返回 $\emptyset$

3.3 队列的实现

3.3.1 队列的存储方式

- **顺序队列**：队列的顺序存储实现
- **链式队列**：队列的链式存储实现

3.3.2 队列的顺序存储实现

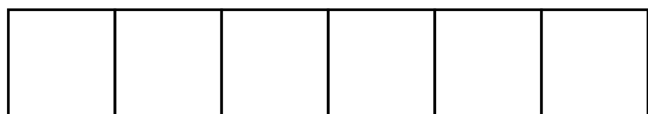
队列可以采用顺序存储方式，称为**顺序队列**。

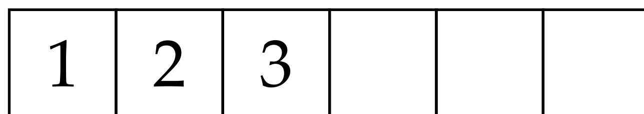
伪码 3.2 : 队列的顺序存储实现

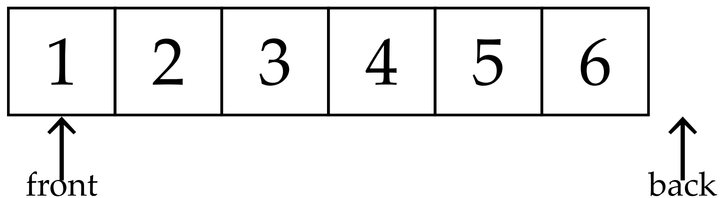
SequentialQueue:

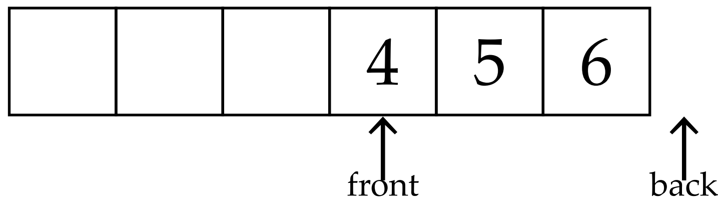
- data: 指向数据的指针，数据连续存储
- front: 下一个出队元素的位置
- back: 下一个入队元素的位置
- capacity: 队列的容量

3.3.3 顺序队列实现的问题



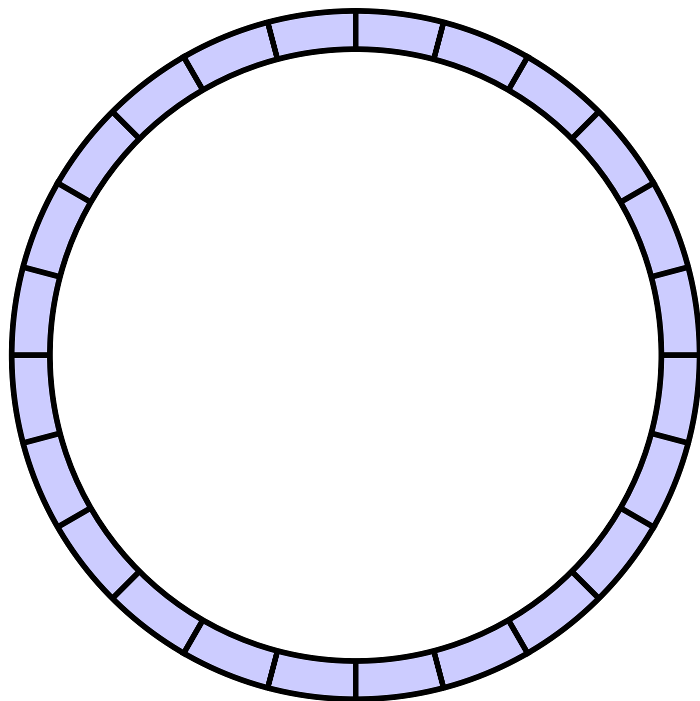






此时，虽然队列分配的空间是有空闲的，却无法再插入元素，出现了“假满”的情况。

3.3.4 圆形队列



圆形队列

伪码 3.3 : 圆形队列的实现

CircularQueue:

- data: 指向数据的指针，数据连续存储
- front: 下一个出队元素的位置
- back: 下一个入队元素的位置
- capacity: 队列的容量

3.3.5 圆形队列：初始化

伪码 3.4 : 初始化圆形队列

Initialize(Q):

1 Q.data $\leftarrow$ Allocate(队列的容量)
2 Q.front $\leftarrow 0$
3 Q.back $\leftarrow 0$
4 Q.capacity $\leftarrow$ 队列的容量

3.3.6 圆形队列：销毁

伪码 3.5 : 销毁圆形队列

Destroy(Q):

1. Free(Q.data)
2. Q.data $\leftarrow \emptyset$
3. Q.front $\leftarrow 0$
4. Q.back $\leftarrow 0$
5. Q.capacity $\leftarrow 0$

3.3.7 圆形队列：清空

伪码 3.6 : 清空圆形队列

Clear(Q):

1. $Q.\text{front} \leftarrow 0$
2. $Q.\text{back} \leftarrow 0$

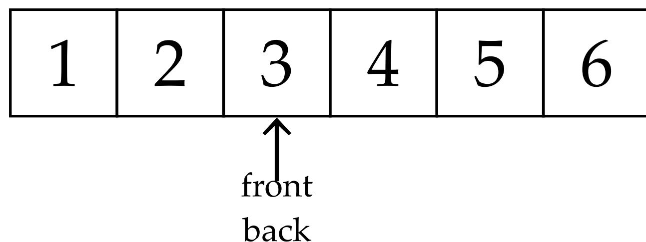
3.3.8 圆形队列：判空

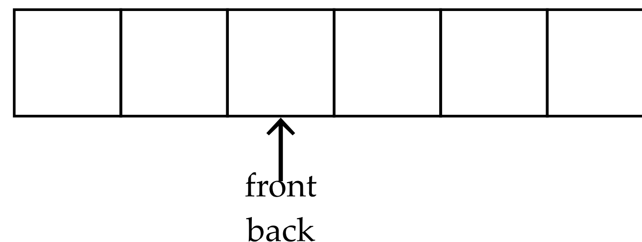
伪码 3.7 : 判断圆形队列是否为空

IsEmpty(Q):

1 **return** Q.front $=$ Q.back

3.3.9 圆形队列：判满





无法区分以上两种情况。可以少用一个元素的空间来进行区分。

伪码 3.8 : 判断圆形队列是否为满

IsFull(Q):

1 **return** $(Q.\text{back} + 1) \bmod Q.\text{capacity} = Q.\text{front}$

若圆形队列的容量为 n，则最多只能存储 $n-1$ 个元素。

3.3.10 圆形队列：元素个数

伪码 3.9 : 获得圆形队列的元素个数

Length(Q):

1 **return** $(Q.\text{back} - Q.\text{front}) \bmod Q.\text{capacity}$[1]

[1]注意编程语言中求余运算结果的符号，如果为负还需要调整。

3.3.11 圆形队列：队头元素

伪码 3.10 : 获得队头元素

```
Front(Q):
1  if ¬ IsEmpty(Q)
2      return Q.data[Q.front]
3  else
4      return ∅
```

3.3.12 圆形队列：入队

伪码 3.11 : 将一个元素插入到队尾

Enqueue(Q, x):

1. **if** $\mathrm{IsFull}(Q)$
2. **error** 队列已满
3. $Q.\mathrm{data}[Q.\mathrm{back}] \leftarrow x$
4. $Q.\mathrm{back} \leftarrow (Q.\mathrm{back} + 1) \bmod Q.\mathrm{capacity}$

3.3.13 圆形队列：出队

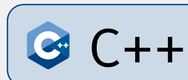
伪码 3.12 : 从队头删除一个元素

Dequeue(Q):

1 **if** $\neg$ IsEmpty(Q)
2   $x \leftarrow Q.\text{data}[Q.\text{front}]$
3   $Q.\text{front} \leftarrow (Q.\text{front} + 1) \bmod Q.\text{capacity}$
4   **return** x
5 **else**
6   **return** $\emptyset$

3.3.14 C++ 中的队列 queue

C++ 语言的标准库提供了 queue 类表示队列，注意 queue 类同样也是一种容器适配器。




```cpp
#include <queue>

template<class T, class Container = deque<T>>
class queue {
public:
  using value_type = typename Container::value_type;
  // 其他关联类型

```

3.3.14 C++ 中的队列 queue

```
9   protected:
10      Container c; // 实际容器
11
12  public:
13      queue() : queue(Container()) {}
14      explicit queue(const Container&);
15      template<class InputIter>
16      queue(InputIter first, InputIter last);
17      // 其他构造函数
18
19      bool empty() const { return c.empty(); }
```

3.3.14 C++ 中的队列 queue

```
20      size_type size() const { return c.size(); }
21      reference front() { return c.front(); }
22      const_reference front() const { return c.front(); }
23      reference back() { return c.back(); }
24      const_reference back() const { return c.back(); }
25      void push(const value_type& x) { c.push_back(x); }
26      void pop() { c.pop_front(); }
27      // 其他成员函数
28  };
```

3.3.15 queue 使用示例

```cpp
#include <cassert>
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(0); // 0 入队
    q.push(1); // q = 0 1
    q.push(2); // q = 0 1 2
    q.push(3); // q = 0 1 2 3
    assert(q.front() == 0);
```

3.3.15 queue 使用示例

```
12    assert(q.back() == 3);
13    assert(q.size() == 4);
14    q.pop(); // 0 出队
15    assert(q.size() == 3);
16    q.front() = 4; // 修改队头元素
17    // 打印并删除所有元素
18    // 注意 queue 不支持 begin()/end()，因此不能使用范围 for
      循环
19    std::cout << "q: ";
20    for (; !q.empty(); q.pop())
21      std::cout << q.front() << ' ';
```

3.3.15 queue 使用示例

```
22     std::cout << '\n';
23     // 输出: q: 4 2 3
24     assert(q.size() == 0);
25 }
```

3.4 队列的扩展

3.4.1 双端队列

定义 4.1.1. 两端都可以进行插入和删除操作的线性表被称为**双端队列**（*double-ended queue*，*deque*）。

双端队列同样可以使用顺序表或链表来实现。

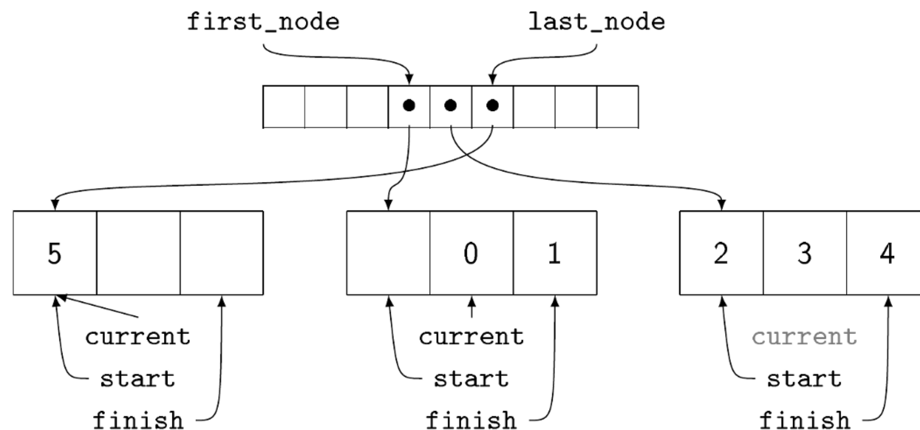
C++ 中提供 deque 容器，可视作双端队列，但是 deque 并不限制序列中间的插入和删除操作，只是效率较低。

3.4.2 C++ 中的双端队列 deque

```cpp
#include <deque>

template<class T>
class deque {
public:
    // 关联类型，同 vector
    // 构造/复制/销毁，同 vector
    // 迭代器相关函数，同 vector
    // 容量相关函数，同 vector
    // 元素操作函数，同 vector

    // 修改操作
    void push_front(const T& x);
    void push_back(const T& x);
```



以索引序列的形式实现，支持在两端快速插入和删除元素，时间复杂度：均摊 $O(1)$。

3.4.2 C++ 中的双端队列 deque

```
15    void pop_front();
16    void pop_back();
17    // 插入、删除、清空等，同 vector
18  };
```

3.4.3 优先级队列

定义 4.3.1. 按优先级进行删除元素操作的集合被称为**优先级队列**（*priority queue*）。

优先级队列每次删除的都是当前队列中优先级最高的元素（最大或最小），可用于复杂调度的各种场景。

实现：

- 素朴实现：使用普通线性表，插入和删除操作的复杂度分别为 $O(1)$ 和 $O(n)$（或者相反）
- 堆：用线性表实现的树状结构，插入和删除操作的复杂度均为 $O(\log n)$

3.4.4 C++ 中的优先级队列 priority_queue

```cpp
#include <queue>

template<class T,
         class Container = vector<T>,
         class Compare   = less<typename Container::value_type>>
class priority_queue {
public:
  // 关联类型，同 queue
  using value_compare   = Compare;

protected:
```

3.4.4 C++ 中的优先级队列 priority_queue

```
12      Container c;
13      Compare comp;
14
15  public:
16      priority_queue();
17      template<class InputIter>
18      priority_queue(InputIter first, InputIter last);
19      // 其他构造函数
20
21      bool empty() const { return c.empty(); }
22      size_type size() const { return c.size(); }
23      const_reference top() const { return c.front(); }
```

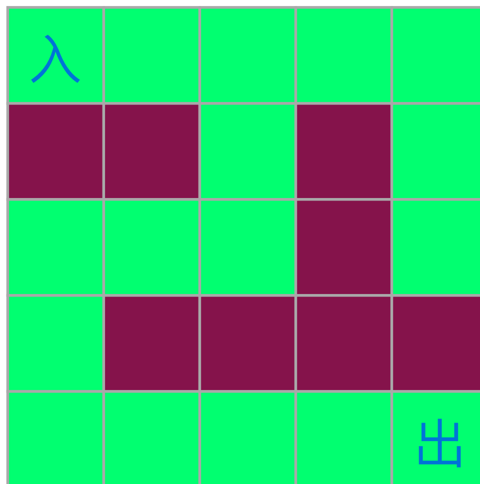
3.4.4 C++ 中的优先级队列 priority_queue

```
24      void push(const value_type& x);
25      void pop();
26      // 其他成员函数
27  };
```

3.5 队列的应用

3.5.1 走迷宫

问题. 给出一张 $n \times m$ 的方格地图，每个格子可能是空格（可通过）或障碍（不可通过）。问能否从地图左上角 $(0,0)$ 走到右下角 $(n-1, m-1)$？最少需要多少步？



3.5.2 走迷宫：解题思路

问题：

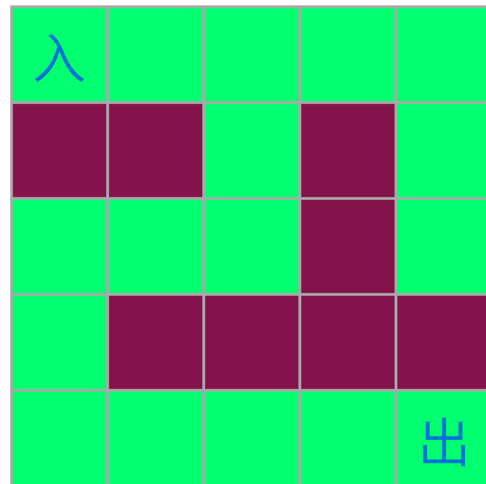
1. 是否可达
2. 最少步数

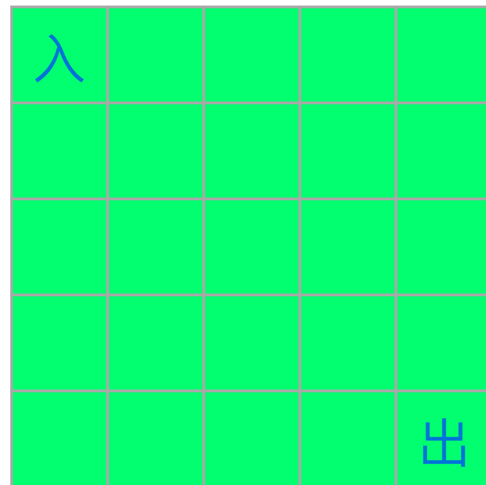
思路：枚举所有可能的路径

存在问题：组合爆炸

右下地图的路径数：$\frac{(2n)!}{(n!)^2}$

参见：https://oeis.org/A000984





3.5.3 走迷宫：改进思路

观察 1：从入口到每个格子的最少步数跟它周围相邻的格子有关

观察 2：按照最少步数的大小，格子呈现类似水波扩散的分布形态

启示：一个格子可以扩展到它的邻格，越早扩展到的格子可以越优先去扩展它的邻格

这样的方法被称为**广度优先搜索**。特点：**先进先出**，可以使用队列进行辅助

0	1	2	3	4
		3		5
6	5	4		6
7				
8	9	10	11	12

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

3.5.4 走迷宫：改进算法

伪码 3.13 : 使用广度优先搜索判断迷宫是否有解

```
FindPath(M):
1  Initialize(Q)
2  Enqueue(Q, 入口)
3  while ¬ IsEmpty(Q)
4      c ← Dequeue(Q)
5      if c = 出口
6          return true   # 找到路径
7      扩展 c 的邻格，如果没有访问过则入队
8  return false   # 没找到路径
```

3.5.5 走迷宫：题解代码

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

int main() {
  // 读入迷宫地形
  int n, m;
  std::cin >> n >> m;
  std::vector<std::vector<int>> maze(n, std::vector<int>(m));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
```

3.5.5 走迷宫：题解代码

```
13          int t;
14          std::cin >> t;        // 输入 0 表示空格，1 表示障碍
15          maze[i][j] = t - 2;  // 存储 -2 表示空格，-1 表示障碍
16        }
17      }
18      // 初始化队列，tuple 是表示元组的类
19      std::queue<std::tuple<int, int, int>> q;
20      // 加入入口，emplace 效率更高
21      q.emplace(0, 0, 0);  // 等价于 q.push(std::make_tuple(0, 0,
        0))
22      // 使用队列进行探索
23      while (!q.empty()) {
```

3.5.5 走迷宫：题解代码

```
24      // 获取当前探索位置及步数
25        auto [x, y, d] = q.front();
26        q.pop();
27        if (x < 0 || x >= n || y < 0 || y >= m ||
28            maze[x][y] != -2) {  // 出边界或不是未探索的空格
29          continue;
30        }
31        maze[x][y] = d;                      // 记录步数
32        if (x == n - 1 && y == m - 1) {  // 到达出口
33          break;
34        }
35      // 加入下一步可能探索的位置
```

3.5.5 走迷宫：题解代码

```
36        q.emplace(x + 1, y, d + 1);
37        q.emplace(x - 1, y, d + 1);
38        q.emplace(x, y + 1, d + 1);
39        q.emplace(x, y - 1, d + 1);
40    }
41    std::cout << maze[n - 1][m - 1] << std::endl;
42 }
```

3.6 小结

3.6.1 本讲小结

- 队列的概念
- 队列的实现：圆形队列
- 队列的扩展：双端队列、优先级队列
- 队列的应用：走迷宫（广度优先搜索）