第6章 队 列

像堆栈一样，队列也是一种特殊的线性表。队列的插入和删除操作分别在线性表的两端进行，因此，队列是一个先进先出（first-in-first-out, FIFO）的线性表。尽管可以很容易地从线性表类LinearList（见程序3-1）和链表类Chain（见程序3-8）中派生出队列类，但在本章中并没有这样做。出于对执行效率的考虑，我们把队列设计成一个基类，分别采用了公式化描述和链表描述。

在本章的应用部分，给出了四个使用队列的应用。第一个应用是关于5.5.3节所介绍的火车车厢重排问题。在本章中对这个问题做了修改，要求缓冲铁轨按FIFO方式而不是LIFO方式工作；第二个应用是关于寻找两个给定点之间最短路径的问题，这是一个经典的问题。可以把这个应用看成是5.5.6节迷宫问题的一种变化，即寻找从迷宫入口到迷宫出口的最短路径。5.5.6节中的代码并不能保证得到一条最短的路径，它只能保证如果存在一条从入口到出口的路径，则一定能找到这样一条路径（没有限定长度）；第三个应用选自计算机视觉领域，主要用于识别图像中的图元；最后一个应用是一个工厂仿真程序。工厂内有若干台机器，每台机器能够执行一道不同的工序。每一项任务都由一系列工序组成。我们给出了一个仿真程序，它能够仿真工厂中的任务流。该程序能够确定每项任务所花费的总的等待时间以及每台机器所产生的总的等待时间，可以根据这些信息来改进工厂的设计。

为了获得较高的执行效率，本章中每个应用都采用了队列数据结构。在后续章节中还会介绍其他几种队列应用。

6.1 抽象数据类型

定义 **[队列]** 队列（quene）是一个线性表，其插入和删除操作分别在表的不同端进行。添加新元素的那一端被称为队尾(rear)，而删除元素的那一端被成为队首(front)。

一个三元素的队列如图6-1a所示，从中删除第一个元素*A*之后将得到图6-1b所示的队列。如果要向图6-1b的队列中添加一个元素*D*，必须把它放在元素*C*的后面。添加*D*以后所得到的结果如图6-1c所示。

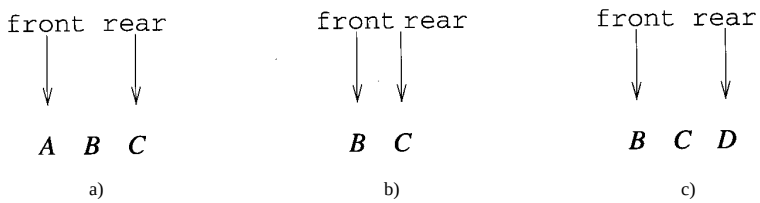


图6-1 队列举例

所以，队列是一个先进先出（FIFO）的线性表，而堆栈是一个先进后出（LIFO）的线性表。队列的抽象数据类型描述见ADT 6-1。

ADT6-1 **队列的抽象数据类型描述**

抽象数据类型 *Queue* {
实例
有序线性表，一端称为 front，另一端称为 rear；
操作
Create (): 创建一个空的队列；
IsEmpty (): 如果队列为空，则返回 true，否则返回 false；
IsFull ():如果队列满，则返回 true；否则返回 false；
First (): 返回队列的第一个元素；
Last ():返回队列的最后一个元素；
Add (*x*): 向队列中添加元素 *x*；
Delete (*x*): 删除队首元素，并送入 *x*;
}

6.2 公式化描述

假定采用公式（6-1）来描述一个队列。

$$location\ (i) = i\text{-}1 \tag{6-1}$$

这个公式在公式化描述的堆栈中工作得很好。如果使用公式（6-1）把数组queue[MaxSize]描述成一个队列，那么第一个元素为 queue[0]，第二个元素为queue[1]，…。front总是为0，rear始终是最后一个元素的位置，队列的长度为 rear+1。对于一个空队列，有 rear=－1。使用公式（6-1），图6-1中的队列可以表示成图6-2的形式。

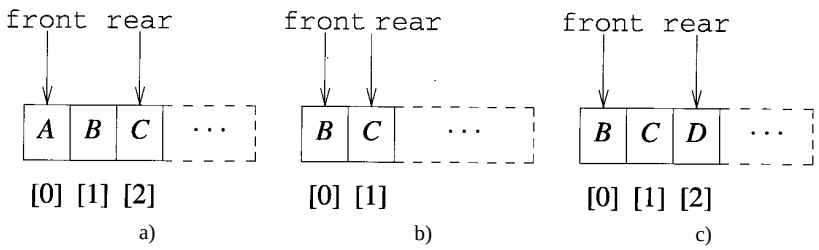


图6-2 使用公式（6-1）描述图6-1中的队列

向队列中添加一个元素时，需要把 rear增1，并把新元素放入queue[rear]。这意味着一次添加操作所需要的时间为O(1)。删除一个元素时，把位置1至位置n的元素分别左移一个位置，因此删除一个元素所花费的时间为 $\Theta(n)$，其中n为删除完成之后队列中的元素数。如此看来，公式（6-1）应用于堆栈，可使堆栈的插入和删除操作均耗时 $\Theta(1)$，而应用于队列，则使队列的删除操作所需要的时间达到 $\Theta(n)$。

如果采用公式（6-2），就可以使队列的删除操作所需要的时间减小至 $\Theta(1)$。

$$location\ (i) = location\ (1) + i\text{-}1 \tag{6-2}$$

从队列中删除一个元素时，公式（6-2）不要求把所有的元素都左移一个位置，只需简单地把*location* (1)增加1即可。图6-3给出了在使用公式（6-2）时，图6-1中各队列的相应描述。注意，在使用公式（6-2）时，front=*location* (1)，rear=*location* (最后一个元素)，一个空队列

具有性质rear<front。

如图6-3b 所示，每次删除操作将导致front右移一个位置。当rear<MaxSize-1时才可以直接在队列的尾部添加新元素。若 rear=MaxSize-1且front>0时（表明队列未满），为了能够继续向队列尾部添加元素，必须将所有元素平移到队列的左端（如图 6-4所示），以便在队列的右端留出空间。对于使用公式（ 6-1）的队列来说，这种平移操作将使最坏情况下的时间复杂性增加Θ(1)，而对于使用公式（6-2）的队列来说，最坏情况下的时间复杂性则增加了 Θ(n)。所以，使用公式（6-2）在提高删除操作执行效率的同时，却降低了添加操作的执行效率。

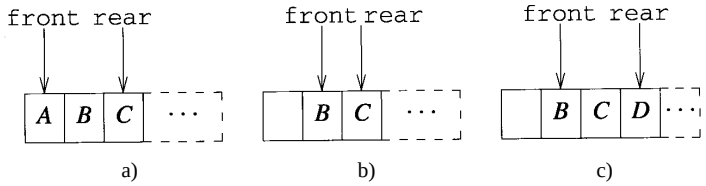


图6-3 使用公式（6-2）描述图6-1中的队列

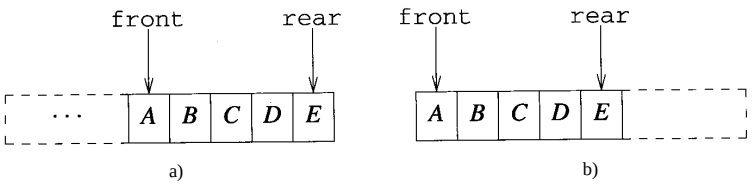


图6-4 队列的平移

a) 移位之前 b) 移位之后

若使用公式（6-3），则队列的添加和删除操作在最坏情况下的时间复杂性均变成 Θ(1)。

$$location(i) = (location(1) + i - 1) \% MaxSize \qquad (6\text{-}3)$$

这时，用来描述队列的数组被视为一个环（如图 6-5所示）。在这种情况下，对front的约定发生了变化，它指向队列首元素的下一个位置（逆时针方向），而rear的含义不变。向图 6-5a中的队列添加一个元素将得到图 6-5b 所示的队列，而从图6-5b 的队列中删除一个元素则得到图6-5c 所示的队列。

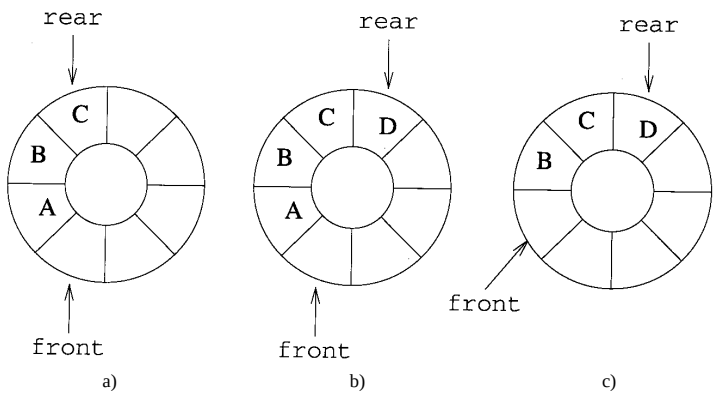


图6-5 循环队列

a) 初始状态 b) 添加 c) 删除

当且仅当 front=rear 时队列为空。初始条件front=rear=0定义了一个初始为空的队列。现在需要确定队列为满的条件。如果不断地向图6-5b 的队列添加元素，直到队列满为止，那么将看到图 6-6所示的情形。这时有front=rear，竟然与队列为空的条件完全一样！因此，我们无法区分出队列是空还是满。为了避免这个问题，可以不允许队列被填满。为此，在向队列添加一个元素之前，先判断一下本次操作是否会导致队列被填满，如果是，则报错。因此，队列的最大容量实际上是MaxSize-1。

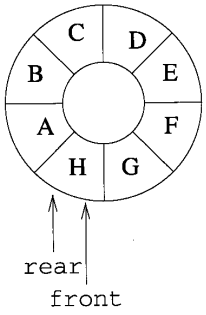


图6-6 能容纳MaxSize个元素的循环队列

可用程序6-1所示的C++类来实现抽象数据类型Queue。在实现公式化描述的堆栈时（见程序5-1），为了简化代码的设计，重用了LinearList 类（见程序3-1）的定义。然而不能通过使用同样的方法来实现Queue 类，因为Queue 的实现基于公式（6-3），而LinearList的实现基于公式（6-1）。程序6-2和程序6-3给出了Queue 成员函数的代码。注意观察Queue 的构造函数是怎样保证循环队列的容量比数组的容量少 1的。利用如下语句，可以创建一个能够容纳 12个整数的队列：

```
Queue<int>Q(12);
```

队列的成员函数与堆栈的对应函数相类似，因此这里不再具体介绍这些函数。当 T是一个内部数据类型时，队列构造函数和析构函数的复杂性均为 Θ(1)；而当T是一个用户定义的类时，构造函数和析构函数的复杂性均为O(MaxStackSize)。其他队列操作的复杂性均为 Θ(1)。

程序6-1 公式化类Queue

```
template<class T>
class Queue {
// FIFO 对象
  public:
    Queue(int MaxQueueSize = 10);
    ~Queue() {delete [] queue;}
    bool IsEmpty() const {return front == rear;}
    bool IsFull() const {return (
        ((rear + 1) % MaxSize == front) ? 1 : 0);}
    T First() const; //返回队首元素
    T Last() const; // 返回队尾元素
    Queue<T>& Add(const T& x);
    Queue<T>& Delete(T& x);
  private:
    int front;  //与第一个元素在反时针方向上相差一个位置
    int rear;   // 指向最后一个元素
    int MaxSize; // 队列数组的大小
    T *queue;   // 数组
};
```

程序6-2 Queue类的成员函数

```
template<class T>
```

```
Queue<T>::Queue(int MaxQueueSize)
{// 创建一个容量为 MaxQueueSize的空队列
   MaxSize = MaxQueueSize + 1;
   queue = new T[MaxSize];
   front = rear = 0;
}

template<class T>
T Queue<T>::First() const
{// 返回队列的第一个元素
 // 如果队列为空，则引发异常OutOfBounds
   if (IsEmpty()) throw OutOfBounds();
   return queue[(front + 1) % MaxSize];
}

template<class T>
T Queue<T>::Last() const
{// 返回队列的最后一个元素
 // 如果队列为空，则引发异常OutOfBounds
   if (IsEmpty()) throw OutOfBounds();
   return queue[rear];
}
```

程序6-3 Queue类的成员函数

```
template<class T>
Queue<T>& Queue<T>::Add(const T& x)
{// 把 x 添加到队列的尾部
 // 如果队列满，则引发异常NoMem
   if (IsFull()) throw NoMem();
   rear = (rear + 1) % MaxSize;
   queue[rear] = x;
   return *this;
}

template<class T>
Queue<T>& Queue<T>::Delete(T& x)
{// 删除第一个元素，并将其送入x
// 如果队列为空，则引发异常OutOfBounds
   if (IsEmpty()) throw OutOfBounds();
   front = (front + 1) % MaxSize;
   x = queue[front];
   return *this;
}
```

练习

1. 扩充队列的ADT，增加以下函数：

1) 确定队列中的元素数目。

2) 输入一个队列。

3) 输出一个队列。

2. 扩充队列的ADT，增加以下函数：

1) 把一个队列分解成两个队列，其中一个队列包含原队列中的第 1、3、5、… 个元素，另一个队列包含了其余元素。

2) 合并两个队列（称队列1和队列2），在新队列中，从队列1开始，两个队列的元素轮流排列，若某个队列中的元素先用完，则将另一个队列中的剩余元素依次添加在新队列的尾部。合并完成后，各元素之间的相对次序应与合并前的相对次序相同。

请扩充公式化描述的队列类的定义，增加以上两个成员函数。编写并测试代码。

3. 使用公式（6-2）设计一个相应的C++队列类，编写并测试所有代码。

4. 修改程序6-1中的Queue类，使得队列的容量与数组queue的大小相同。为此，可引入另外一个私有成员LastOp来跟踪最后一次队列操作。可以肯定，如果最后一次队列操作为 Add，则队列一定不为空；如果最后一次队列操作为 Delete，则队列一定不会满。因此，当 front=rear 时，可使用LastOp来区分一个队列是空还是满。试测试修改后的代码。

5. 双端队列（deque）是指这样一个有序线性表：可在表的任何一端进行插入和删除操作。

1) 给出双端队列的抽象数据类型描述，要求包含以下操作： *Create*，*IsEmpty*，*IsFull*，*Left*，*Right*，*AddLeft*，*AddRight*，*DeleteLeft*和*DeleteRight*。

2) 采用公式（6-3）来描述双端队列。设计一个与双端队列抽象数据类型描述相对应的C++类Deque，要求编写出所有类成员的代码。

3) 采用适当的测试数据来测试所编写的代码。

6.3 链表描述

像堆栈一样，也可以使用链表来实现一个队列。此时需要两个变量 front和rear来分别跟踪队列的两端，这时有两种可能的情形：从front开始链接到rear（如图6-7a所示）或从rear开始链接到front（如图6-7b所示）。不同的链接方向将使添加和删除操作的难易程度有所不同。图 6-8和6-9分别演示了添加元素和删除元素的过程。可以看到，两种链接方向都很适合于添加操作，而从front到rear的链接更便于删除操作的执行。因此，我们将采用从front到rear的链接模式。

可以取初值front=rear=0，并且认定当且仅当队列为空时 front=0。利用3.4.3节的扩展，可以把类LinkedQueue定义为Chain类（见程序3-8）的一个派生类，练习6即按照这种方式来实现LinkedQueue。在本节中把LinkedQueue定义为一个基类。

程序6-4给出了一个链表队列的类定义，程序6-5和程序6-6给出了相应的成员函数。Node类与自定义链表堆栈（见程序5-4）中所使用的相同。为了便于成员函数的实现，可把 LinkedQueue定义为Node的友元。可以分别采用空队列、单元素队列和多元素队列来人工跟踪LinkedQueue代码的执行。除析构函数外，链表队列所有成员函数的复杂性均为$\Theta(1)$。

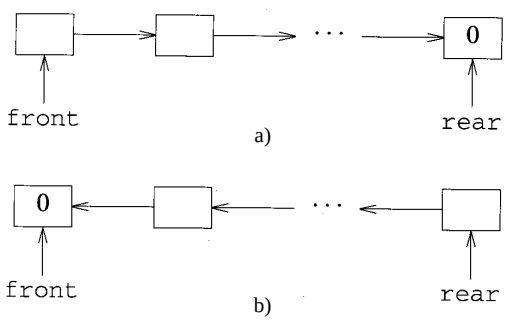
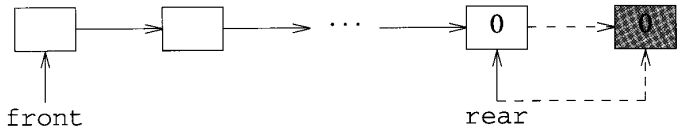
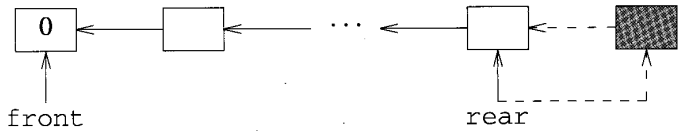


图6-7 链表队列



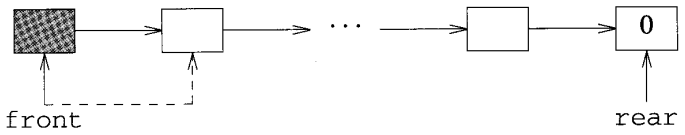
a)



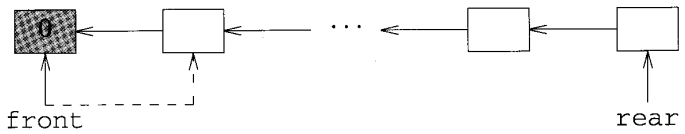
b)

图6-8 向链表队列中添加元素

a) 向图6-7a 的队列添加元素 b) 向图6-7b 的队列添加元素



a)



b)

图6-9 从链表队列中删除元素

a) 从图6-7a 的队列中删除元素 b) 从图6-7b 的队列中删除元素

程序6-4 链表队列的类定义

```
template<class T>
class LinkedQueue {
// FIFO对象
  public:
    LinkedQueue() {front = rear = 0;} // 构造函数
    ~LinkedQueue(); // 析构函数
    bool IsEmpty() const
        {return ((front) ? false : true);}
    bool IsFull() const;
    T First() const; // 返回第一个元素
    T Last() const; // 返回最后一个元素
    LinkedQueue<T>& Add(const T& x);
    LinkedQueue<T>& Delete(T& x);
  private:
    Node<T> *front;  // 指向第一个节点
    Node<T> *rear;   //指向最后一个节点
};
```

程序6-5 链表队列的函数实现

```
template<class T>
LinkedQueue<T>:: ~LinkedQueue()
{// 队列析构函数，删除所有节点
  Node<T> *next;
  while (front) {
    next = front->link;
    delete front;
    front = next;
    }
}

template<class T>
bool LinkedQueue<T>::IsFull() const
{// 判断队列是否已满
  Node<T> *p;
  try {p = new Node<T>;
       delete p;
       return false;}
  catch (NoMem) {return true;}
}

template<class T>
T LinkedQueue<T>::First() const
{// 返回队列的第一个元素
 // 如果队列为空，则引发异常OutOfBounds
  if (IsEmpty()) throw OutOfBounds();
  return front->data;
}

template<class T>
T LinkedQueue<T>::Last() const
{// 返回队列的最后一个元素
 // 如果队列为空，则引发异常OutOfBounds
  if (IsEmpty()) throw OutOfBounds();
  return rear->data;
}
```

程序6-6 链表队列的函数实现

```
template<class T>
LinkedQueue<T>& LinkedQueue<T>::Add(const T& x)
{// 把 x 添加到队列的尾部
 // 不捕获可能由new引发的NoMem 异常

  // 为新元素创建链表节点
  Node<T> *p = new Node<T>;
  p->data = x;
  p->link = 0;
```

```
    // 在队列尾部添加新节点
    if (front) rear->link = p;  //队列不为空
    else front = p;             // 队列为空
    rear = p;

    return *this;
}

template<class T>
LinkedQueue<T>& LinkedQueue<T>::Delete(T& x)
{{// 删除第一个元素，并将其放入 x
  // 如果队列为空，则引发异常 OutOfBounds

  if (IsEmpty()) throw OutOfBounds();

    //保存第一个节点中的元素
    x = front->data;

    // 删除第一个节点
    Node<T> *p = front;
    front = front->link;
    delete p;

    return *this;
}
```

练习

6. 利用3.4.3节Chain类的扩充版本（包含函数 Append），从Chain类中派生出链表队列类LinkedQueue，并设计出相应的成员函数。

7. 采用链表队列来完成练习1。

8. 采用链表队列来完成练习 2，所不同的是，要求各操作均就地进行而不得使用新节点。在分解/合并操作完成之后，原输入队列应为空。

9. 采用链表来完成练习5。分别指出每种操作的复杂性。

10. 采用双向链表来完成练习5。分别指出每种操作的复杂性。

6.4 应用

6.4.1 火车车厢重排

下面来重新考察一下5.5.3节的火车车厢重排问题。如图6-10所示，假定缓冲铁轨位于入轨和出轨之间。由于这些缓冲铁轨均按 FIFO的方式运作，因此可将它们视为队列。与 5.5.3节一样，禁止将车厢从缓冲铁轨移动至入轨，也禁止从出轨移动车厢至缓冲铁轨。所有的车厢移动都按照图6-10中箭头所示的方向进行。

铁轨*Hk* 为可直接将车厢从入轨移动到出轨的通道。因此，可用来容留车厢的缓冲铁轨的数目为*k*-1。

假定重排9节车厢，其初始次序为5, 8, 1, 7, 4, 2, 9, 6, 3，同时令*k*=3。3号车厢不能直接移

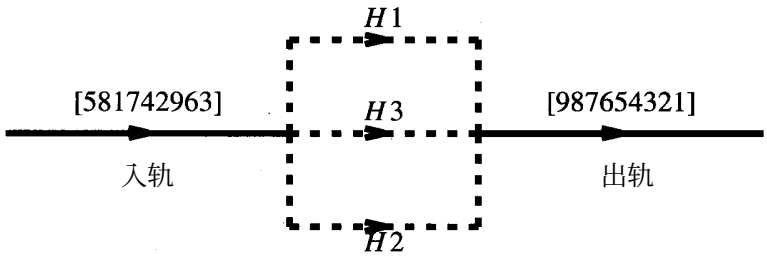


图6-10 三个缓冲铁轨示例

动到出轨，因为1号车厢和2号车厢必须排在3号车厢之前。因此，把3号车厢移动至$H1$。6号车厢可放在$H1$中3号车厢之后，因为6号车厢将在3号车厢之后输出。此后9号车厢可以继续放在$H1$中6号车厢之后，而接下来的2号车厢不可放在9号车厢之后，因为2号车厢必须在9号车厢之前输出。因此，应把2号车厢放在$H2$的首部。之后4号车厢被放在$H2$中2号车厢之后，7号车厢又被放在4号车厢之后。至此，1号车厢可通过$H3$直接移动至出轨，然后从$H2$移动2号车厢至出轨，从$H1$移动3号车厢至出轨，从$H2$移动4号车厢至出轨。由于5号车厢此时仍位于入轨之中，所以把8号车厢移动至$H2$，这样就可以把5号车厢直接从入轨移动至出轨。这之后，可依次从缓冲铁轨中输出6号、7号、8号和9号车厢。

在把一节车厢移动到缓冲铁轨中时，可以采用如下的原则来确定应该把这节车厢移动到哪一个缓冲铁轨。车厢c应移动到这样的缓冲铁轨中：该缓冲铁轨中现有各车厢的编号均小于 c；如果有多个缓冲铁轨都满足这一条件，则选择一个左端车厢编号最大的缓冲铁轨；否则选择一个空的缓冲铁轨（如果有的话）。

1. 第一种实现方法

可以采用链表队列来实现车厢重排算法，其中，用链表队列来表示 k-1个缓冲铁轨。可以按照程序5-8、程序5-9和程序5-10的模式来设计该算法。程序6-7给出了函数Output和Hold的新代码。对于程序 5-8中的函数 Railroad，应做以下修改： 1) 将k减1；2) H的类型修改为LinkedQueue<int> *；3) 把MinS改为MinQ；4) 从Hold的调用中删除最后一个参数(n)。完成车厢重排所需要的时间为O(nk)。借助于AVL树（见第11章），可以把复杂性减小至O(nlogk)。

程序6-7 使用队列来重排车厢

```
void Output(int& minH, int& minQ, LinkedQueue<int> H[], int k, int n)
{//从缓冲铁轨移动到出轨，并修改 minH 和 minQ.
  int c;  // 车厢编号

  // 从队列 minQ 中删除编号最小的车厢 minH
  H[minQ].Delete(c);
  cout << "Move car " << minH << " from holding track " << minQ << " to output" << endl;

  // 通过检查所有队列的首部，寻找新的 minH和minQ
  minH = n + 2;
  for (int i = 1; i <= k; i++)
    if (!H[i].IsEmpty() &&
          (c = H[i].First()) < minH) {
      minH = c;
```

```
        minQ = i;}
}

bool Hold(int c, int& minH, int &minQ, LinkedQueue<int> H[], int k)
{//把车厢c 移动到缓冲铁轨中
 // 如果没有可用的缓冲铁轨，则返回 false，否则返回true

  // 为车厢 c 寻找最优的缓冲铁轨
  // 初始化
  int BestTrack = 0，// 目前最优的铁轨
      BestLast = 0，  // BestTrack 中最后一节车厢
      x;              // 车厢编号

  // 扫描缓冲铁轨
  for (int i = 1; i <= k; i++)
    if (!H[i].IsEmpty()) {// 铁轨 i 不为空
      x = H[i].Last();
      if (c > x && x > BestLast) { // 铁轨 i 尾部的车厢编号较大
        BestLast = x;
        BestTrack = i;}
      }
    else // 铁轨i 为空
      if (!BestTrack) BestTrack = i;

  if (!BestTrack) return false; // 没有可用的铁轨

  // 把c 移动到最优铁轨
  H[BestTrack].Add(c);
  cout << "Move car " << c << " from input " << "to holding track " << BestTrack << endl;

  // 如果有必要，则修改 minH和minQ
  if (c < minH) {minH = c; minQ = BestTrack;}

  return true;
}
```

2. 第二种实现方法

如果只是为了简单地输出车厢重排过程中所必要的车厢移动次序，那么只需了解每个缓冲铁轨的最后一个成员是谁以及每节车厢当前位于哪个铁轨即可。如果缓冲铁轨 i为空，则令last[i]=0，否则令last[i]为铁轨中最后一节车厢的编号。如果车厢i位于入轨之中，令track[i]=0；否则，令track[i]为车厢i所在的缓冲铁轨。在起始时有last[i]=0，1≤i＜k，track[i]=0，1≤i≤n。程序6-8中并未使用队列，它所产生的输出与程序 6-7所产生的输出完全相同，二者均具有相同的渐进复杂性。

程序6-8 不使用队列来重排车厢

```
void Output(int NowOut, int Track, int& Last)
{//将车厢NowOut 从缓冲铁轨移动到出轨，并修改 Last
```

```
  cout << "Move car " << NowOut << " from holding track " << Track << " to output" << endl;
  if (NowOut == Last) Last = 0;
}

bool Hold(int c, int last[], int track[], int k)
{//把车厢c 移动到缓冲铁轨中
 // 如果没有可用的缓冲铁轨，则返回 false，否则返回true

  // 为车厢 c 寻找最优的缓冲铁轨
  // 初始化
  int BestTrack = 0,  // 目前最优的铁轨
      BestLast = 0,   // BestTrack中最后一节车厢

  // 扫描缓冲铁轨
  for (int i = 1; i <= k; i++) // find best track
    if (last[i]) {// 铁轨 i 不为空
      if (c > last[i] && last[i] > BestLast) { // 铁轨 i 尾部的车厢编号较大
        BestLast = last[i];
        BestTrack = i;}
      }
    else // 铁轨 i 为空
      if (!BestTrack) BestTrack = i;

  if (!BestTrack) return false; //  没有可用的铁轨

  // 把c 移动到最优铁轨
  track[c] = BestTrack;
  last[BestTrack] = c;
  cout << "Move car " << c << " from input " << "to holding track " << BestTrack << endl;

  return true;
}

bool Railroad(int p[], int n, int k)
{// 用k 个缓冲铁轨进行车厢重排，车厢的初始次序为 p[1:n]
 // 如果重排成功，则返回 true，否则返回false
 // 如果空间不足，则引发异常 NoMem

  // 对数组last和track进行初始化
  int *last = new int [k + 1];
  int *track = new int [n + 1];
  for (int i = 1; i <= k; i++)
    last[i] = 0;  // 铁轨 i 为空
  for (int i = 1; i <= n; i++)
    track[i] = 0;
  k--; // 铁轨k 作为直接移动的通道

  // 对欲输出的下一节车厢的编号置初值
```

```
    int NowOut = 1;

    //按序输出车厢
    for (int i = 1; i <= n; i++)
      if (p[i] == NowOut) {// 直接输出
        cout << "Move car " << p[i] << " from input to output" << endl;
        NowOut++;

        // 从缓冲铁轨中输出
        while (NowOut <= n && track[NowOut]) { Output(NowOut, track[NowOut], last[NowOut]);
        NowOut++;
          }
        }
      else {// 把车厢 p[i] 移动到缓冲铁轨
        if (!Hold(p[i], last, track, k))
        return false;}

    return true;
  }
```

6.4.2 电路布线

在5.5.6节中，对迷宫老鼠问题的解决方案并不能保证找到一条从迷宫入口到迷宫出口的最短路径。而借助于队列，可以找到这样的路径（如果有的话）。在迷宫中寻找最短路径的问题也存在于其他许多领域。例如，在解决电路布线问题时，一种很常用的方法就是在布线区域叠上一个网格，该网格把布线区域划分成 $n \times m$ 个方格，就像迷宫一样，如图 6-11a 所示。从一个方格 a 的中心点连接到另一个方格 b 的中心点时，转弯处必须采用直角，如图 6-11b 所示。如果已经有某条线路经过一个方格，则封锁该方格。我们希望使用 a 和 b 之间的最短路径来作为布线的路径，以便减少信号的延迟。

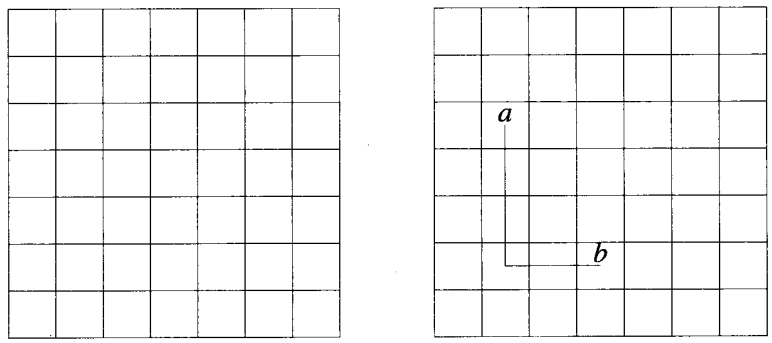
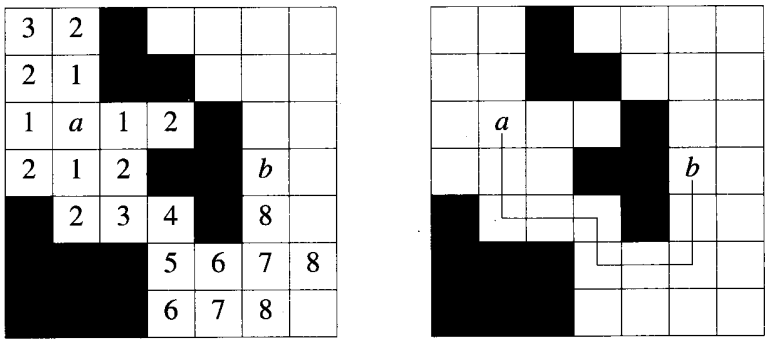


图6-11 电路布线示例

a) 7×7 网格 b) a与b之间的电线

在下面的讨论中，我们假定你对 5.5.6节所介绍的迷宫求解算法已经很熟悉。如果不熟悉，

在继续阅读之前，应该认真复习一下5.5.6节的内容。为了找到网格中位置*a*和*b*之间的最短路径，先从位置*a* 开始搜索，把*a* 可到达的相邻方格都标记为1（表示与*a* 相距为1），然后把标号为1的方格可到达的相邻方格都标记为2（表示与*a*相距为2），继续进行下去，直到到达*b*或者找不到可到达的相邻方格为止。图6-12a 演示了这种搜索过程，其中a=(3,2)，b=(4,6)。图中的阴影方格都是被封锁的方格。



a) b)

图6-12 电路布线

a) 标识间距 b) 电线路径

按照上述搜索过程，当我们到达*b*时，就可以在*b*上标出*b*与*a*之间的距离，在图6-12a 中，*b*上的标号为9。为了得到*a*与*b*之间的最短路径，从*b*开始，首先移动到一个比*b* 的编号小的相邻位置上。一定存在这样的相邻位置，因为任一个方格上的标号与它相邻方格上的标号都至少相差1。在图6-12a 中，可从*b*移动到(5,6)。接下来，从当前位置开始，继续移动到比当前标号小1的相邻位置上，重复这个过程，直至到达 *a*为止。在图6-12a 的例子中，从(5,6)移动到(6,6)，(6,4)，(5,4)，…。图6-12b 给出了所得到的路径。

现在来看看怎样实现上述策略，以设计出搜索最短路径的 C++代码。我们将从5.5.6节的迷宫解决方案中吸取很多思想。一个$m \times m$的网格被描述成一个二维数组，其中用0表示空白的位置，1表示被封锁的位置。整个网格被包围在一堵由1构成的“墙”中。数组offsets用来帮助我们从一个位置移动到其相邻位置。用一个链表队列来跟踪这样的方格：该方格本身已被编号，而它的相邻位置尚未被编号。也可以采用公式化队列，所不同的是必须估计队列的最大长度，就像在求解迷宫问题时估计堆栈的大小一样。见程序6-9。

程序6-9 寻找电路布线最短路径

```
bool FindPath(Position start, Position finish, int& PathLen, Position * &path)
{//寻找从 start到finish的路径
 // 如果成功，则返回 true，否则返回false
 // 如果空间不足，则引发异常NoMem

  if ((start.row == finish.row) &&
     (start.col == finish.col))
       {PathLen = 0; return true;} // start = finish

  // 初始化包围网格的“围墙”
```

```
for (int i = 0; i <= m+1; i++) {
   grid[0][i] = grid[m+1][i] = 1; // 底和顶
   grid[i][0] = grid[i][m+1] = 1; // 左和右
   }

// 初始化offset
Position offset[4];
offset[0].row = 0; offset[0].col = 1; // 右
offset[1].row = 1; offset[1].col = 0; // 下
offset[2].row = 0; offset[2].col = -1; // 左
offset[3].row = -1; offset[3].col = 0; // 上

int NumOfNbrs = 4; // 一个网格位置的相邻位置数
Position here, nbr;
here.row = start.row;
here.col = start.col;
grid[start.row][start.col] = 2; // 封锁

// 标记可到达的网格位置
LinkedQueue<Position> Q;
do {  // 标记相邻位置
   for (int i = 0; i < NumOfNbrs; i++) {
      nbr.row = here.row + offset[i].row;
      nbr.col = here.col + offset[i].col;
      if (grid[nbr.row][nbr.col] == 0) { // unlabeled nbr, label it
         grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
         if ((nbr.row == finish.row) && (nbr.col == finish.col)) break; // 完成
        Q.Add(nbr);} // if 结束
      } // for 结束

   //已到达finish吗?
   if ((nbr.row == finish.row) &&
       (nbr.col == finish.col)) break; // 完成

   // 未到达finish，可移动到nbr吗?
   if (Q.IsEmpty()) return false; // 没有路径
   Q.Delete(here); // 到下一位置
} while(true);

// 构造路径
PathLen = grid[finish.row][finish.col] - 2;
path = new Position [PathLen];

// 回溯至 finish
here = finish;
for (int j = PathLen-1; j >= 0; j--) {
   path[j] = here;
   // 寻找前一个位置
```

```
        for (int i = 0; i < NumOfNbrs; i++) {
          nbr.row = here.row + offset[i].row;
          nbr.col = here.col + offset[i].col;
          if (grid[nbr.row][nbr.col] == j+2) break;
          }
        here = nbr;  // 移动到前一个位置
        }

      return true;
    }
```

在程序6-9的代码中，假定起始位置和结束位置均未被封锁。程序首先检查 start和finish是否相同，如果相同，则路径长度为0，程序终止。否则设置一堵由封锁位置构成的“围墙”，把网格包围起来。然后对 offset数组进行初始化，并在起始位置上标记 2。（所有标号都增加了2，因为数组中采用0和1来表示空白位置和封锁位置，为了得到如图 6-12a 所示的标号，必须把程序中的每个标号减去2）。借助于队列Q并从位置start开始，首先移动到与start相距为1的网格位置，然后移动到与start相距为2的网格位置，不断进行下去，直到到达位置 finish或者无法继续移动到一个新的、空白的位置。在后一种情况下，将不存在到达位置 finish的路径，而在前一种情况下，位置finish将得到一个相应的编号，

如果到达了位置finish，则可以利用网格上的标号来重构路径。路径上的位置（ start除外）均被存储在数组path之中。

由于任意一个网格位置都至多在队列中出现 1次，所以完成网格编号过程需耗时 $O(m^2)$（对一个$m \times m$ 的网格来说）。而重构路径的过程需耗时$\Theta(PathLen)$，其中$PathLen$ 为最短路径的长度。

6.4.3 识别图元

数字化图像是一个$m \times m$ 的像素矩阵。在单色图像中，每个像素的值要么为 0，要么为1，值为0的像素表示图像的背景，而值为 1的像素则表示图元上的一个点，我们称其为图元像素。如果一个像素在另一个像素的左侧、上部、右侧或下部，则称这两个像素为相邻像素。识别图元就是对图元像素进行标记，当且仅当两个像素属于同一图元时，它们的标号相同。

考察图6-13a，其中给出了一个 7×7图像。空白方格代表背景像素，而标记为 1的方格则代表图元像素。像素(1,3)和(2,3)属于同一图元，因为它们是相邻的。像素 (2,4)与（2,3)是相邻的，它们也同样属于同一图元，因此，三个像素 (1,3)，(2,3)和(2,4)属于同一图元。由于没有其他的像素与这三个像素相邻，因此这三个像素定义了一个图元。图 6-13a 的图像中存在4个图元，分别是 {(1,3),(2,3),(2,4)}，{(3,5),(4,4),(4,5),(5,5)}，{(5,2),(6,1),(6,2),(6,3),(7,1),(7,2),(7,3)}，{(5,7),(6,7),(7,6),(7,7)}。在图6-13b 中，属于同一图元的像素被编上相同的标号。

在识别图元的程序中，采纳了许多在解决电路布线问题时所使用的策略。为了轻松地在图像中移动，在图像周围包上一圈空白像素（即 0像素）。采用数组offset来确定与一个给定像素相邻的像素。通过逐行扫描像素来识别图元。当遇到一个没有标号的图元像素时，就给它指定一个图元编号（使用数字2,3,…作为图元编号），该像素就成为一个新图元的种子。通过识别和标记与种子相邻的所有图元像素，可以确定图元中的其他像素。我们把与种子相邻的像素称为1-间距像素。接下来要识别和标记与 1-间距像素相邻的所有无标记图元像素，这些像素被称为

2-间距像素。之后继续识别和标记与2-间距像素相邻的无标记图元像素。这个过程一直持续到再也找不到新的、相邻的无标记图元像素为止。

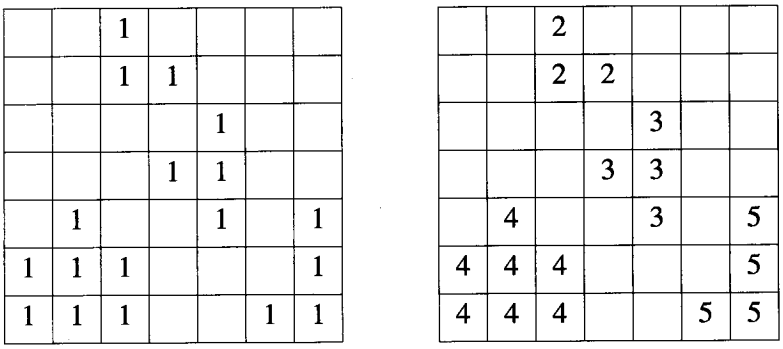


图6-13 识别图元

a) 7×7 图像 b) 标记图元

上述图元识别的过程与确定布线过程中用距离值（与起始方格的距离）来标记网格位置的过程非常类似，因此，识别图元的程序6-10与程序6-9也很类似。

程序6-10首先在图像周围包上一圈背景像素（即0像素），并对数组offset进行初始化。接下来的两个for循环通过扫描图像来寻找下一个图元的种子。种子应是一个无标记的图元像素，对种子来说，有pixel[r][c]=1。将pixel[r][c]从1变成id（图元编号），即可把图元编号设置为种子的标号。接下来借助于链表队列的帮助（也可以使用公式化队列、链表堆栈或公式化堆栈），可以识别出该图元中的其余像素。当函数Label结束时，所有的图元像素都已经获得了一个标号。

初始化“围墙”需耗时$\Theta(m)$，初始化offset需耗时$\Theta(1)$。尽管条件pixel[r][c]==1被检查了m^2次，但它为true的次数只有cnum次，其中cnum为图像中图元的总数。对于任一个图元来说，识别并标记该图元的每个像素（种子除外）所需要的时间为$\Theta(\text{cnum})$。由于任意一个像素都不会同时属于两个以上的图元，因此，识别并标记所有非种子图元像素所需要的总时间为Θ(图像中图元像素总数) = Θ(输入图像中值为1的像素数目)=$\Theta(m^2)$。因此，函数Label总的时间复杂性为$\Theta(m^2)$。

程序6-10 识别图元

```
void Label()
{// 识别图元

  // 初始化“围墙”
  for (int i = 0; i <= m+1; i++) {
    pixel[0][i] = pixel[m+1][i] = 0; // 底和顶
    pixel[i][0] = pixel[i][m+1] = 0; // 左和右
    }

  // 初始化offset
  Position offset[4];
  offset[0].row = 0; offset[0].col = 1; // 右
```

```
    offset[1].row = 1; offset[1].col = 0; // 下
    offset[2].row = 0; offset[2].col = -1; // 左
    offset[3].row = -1; offset[3].col = 0; // 上

    int NumOfNbrs = 4; // 一个像素的相邻像素个数
    LinkedQueue<Position> Q;
    int id = 1;  //图元id
    Position here, nbr;

    // 扫描所有像素
    for (int r = 1; r <= m; r++)  //图像的第 r 行
      for (int c = 1; c <= m; c++)   //图像的第c 列
        if (pixel[r][c] == 1) {// 新图元}
          pixel[r][c] = ++id; // 得到下一个 id
          here.row = r; here.col = c;

          do {// 寻找其余图元
            for (int i = 0; i < NumOfNbrs; i++) {
              // 检查当前像素的所有相邻像素
              nbr.row = here.row + offset[i].row;
              nbr.col = here.col + offset[i].col;
              if (pixel[nbr.row][nbr.col] == 1) {
                 pixel[nbr.row][nbr.col] = id; Q.Add(nbr);}} // end of if and for

            //还有未探索的像素吗？
            if (Q.IsEmpty()) break;
            Q.Delete(here); 一个图元像素
          } while(true);

        } // 结束if 和for
    }
```

6.4.4 工厂仿真

1. 问题描述

一间工厂由m台机器组成。工厂中所执行的每项任务都由若干道工序构成，一台机器用来完成一道工序，不同的机器完成不同的工序。一旦一台机器开始处理一道工序，它会连续不断地进行处理，直到该工序被完成为止。

例6-1　一个生产金属片的工厂可能对于如下每道工序都有一台相应的机器：设计、切割、钻孔、挖孔、修边、造型和焊接。每台机器每次处理一道工序。

每项任务都包含若干道工序。例如，为了在一个新房子内安装暖气管道和空调管道，首先需要花一些时间来进行设计，然后根据设计要求把整块的金属片切割成各种尺寸的金属片，在金属片上钻孔或挖孔（取决于孔的大小），把金属片塑造成管道，焊接管缝，并对粗糙的边进行修剪和打磨。

对于一项任务中的每道工序来说，都有两个属性：一是工时（即完成该道工序需要多长时

间），一是执行该工序的机器。一项任务中的各道工序必须按照一定的次序来执行。一项任务的执行是从处理第一道工序的机器开始的，当第一道工序完成后，任务转至处理第二道工序的机器，依此进行下去，直到最后一道工序完成为止。当一项任务到达一台机器时，若机器正忙，则该任务将不得不等待。事实上，很可能有多项任务同时在同一台机器旁等待。

在工厂中每台机器都可以有如下三种状态：活动、空闲和转换。在活动状态，机器正在处理一道工序，而在空闲状态机器无事可做。在转换状态，机器刚刚完成一道工序，并在为一项新任务的执行做准备，比如机器操作员可能需要清理机器并稍作休息等。每台机器在转换状态期间所花费的时间可能各不相同。

当一台机器可以处理一项新任务时，它可能需要从各个等待者中挑选一项任务来执行。在这里，每台机器都按照FIFO的方式来处理等待者，因此每台机器旁的等待者构成了一个 FIFO队列。在其他类型的工厂中，可以为每项任务指定不同的优先权，当机器变成空闲时，从等待者中首先选择具有最高优先权的任务来执行。

一项任务最后一道工序的完成时间被称为任务完成时间。一项任务的长度等于其所有工序的执行时间之和。如果一项长度为 l 的任务在0时刻到达工厂并在 f 时刻结束，那么它在各机器队列中所花费的等待时间恰好为 $f-l$。为了让顾客满意，希望尽量减少任务在机器队列中的等待时间。如果能够知道每项任务所花费的等待时间是多少，并且知道哪些机器所导致的等待时间最多，就可以据此来改进和提高工厂的效能。

在对工厂进行仿真时，我们只是让任务在机器间流动，并没有实际执行任何工序，而是采用一个模拟时钟来进行仿真计时，每当一道工序完成或一项新任务到达工厂时，模拟时钟就推进一个单位。在完成老任务时，将产生新的任务。每当一道工序完成或一项新任务到达工厂时，我们称发生了一个事件（event）。另外，还存在一个启动事件（start event），用来启动仿真过程。下面的例子演示了在仿真期间没有新任务到达工厂时的仿真过程。

例6-2 考察一间工厂，它由 $m=3$ 台机器构成，可以处理 $n=4$ 项任务。假定所有四项任务都在0时刻出现并且在仿真期间不再有新的任务出现。仿真过程一直持续到所有任务都已完成为止。

三台机器 $M1$、$M2$和$M3$的转换状态所花费的时间分别为 2、0和1。因此，当一道工序完成时，机器$M1$在启动下一道工序之前必须等待 2个时间单元，机器$M2$可以立即启动下一道工序，机器$M3$必须等待1个时间单元。图 6-14a 分别列出了四项任务的特征。例如， 1号任务有3道工序。每道工序用形如（ machine，time）的值对来描述。 1号任务的第一道工序将在 $M1$上完成，需花费的时间为 2个时间单元，第二道工序将在 $M2$上完成，需花费的时间为 4个时间单元，第三道工序将在 $M1$上完成，需花费的时间为 1个时间单元。各项任务的长度分别为7，6，8和4。

图6-14b 列出了工厂仿真的过程。起始时刻，首先按照各任务的第一道工序把 4项任务分别放入相应的机器队列中。 1号任务和3号任务的第一道工序将在 $M1$上执行，因此这两项任务被放入$M1$的队列中。2号任务和4号任务的第一道工序将在$M3$上执行，因此这两项任务被放入$M3$的队列中。 $M2$的队列为空。在启动仿真过程之初，所有 3台机器都是空闲的。符号 I表示机器处于空闲状态。若一台机器处于空闲状态，那么该机器完成当前工序（实际上不存在）的时间没有定义，可用符号L来表示。

仿真从0时刻开始，即第一个事件——启动事件——出现在0时刻。此时，每个机器队列中的第一个任务被调度到相应的机器上执行。因此， 1号任务的第一道工序被调度到 $M1$上执行，2号任务的第一道工序被调度到$M3$上执行。这时$M1$的队列中仅包含3号任务，而$M3$的队列中仅

任务	工序数目	工序
1	3	(1,2) (2,4) (1,1)
2	2	(3,4) (1,2)
3	2	(1,4) (2,4)
4	2	(3,1) (2,3)

a)

时间	机器队列			活动的任务			完成时间		
	*M*1	*M*2	*M*3	*M*1	*M*2	*M*3	*M*1	*M*2	*M*3
Init	1,3	–	2,4	I	I	I	L	L	L
0	3	–	4	1	I	2	2	L	4
2	3	–	4	C	1	2	4	6	4
4	2	–	4	3	1	C	8	6	5
5	2	–	–	3	1	4	8	6	6
6	2,1	4	–	3	C	C	8	9	7
7	2,1	4	–	3	C	I	8	9	L
8	2,1	4,3	–	C	C	I	10	9	L
9	2,1	3	–	C	4	I	10	12	L
10	1	3	–	2	4	I	12	12	L
12	1	3	–	C	C	I	14	15	L
14	–	3	–	1	C	I	15	15	L
15	–	–	–	C	3	I	17	16	L
16	–	–	–	C	C	I	19	19	L

b)

图6-14 工厂仿真示例

a) 任务特征 b) 仿真

包含4号任务，*M*2的队列仍然为空。这样，1号任务成为*M*1上的活动任务，2号任务成为*M*3上的活动任务，*M*2仍然为空闲。*M*1的结束时间变成2（当前时刻0+工序时间2），*M*3的结束时间变成4。

下一个事件在时刻2出现，这个时刻是根据最小的机器完成时间来确定的。在时刻2，*M*1完成了它的当前活动工序，它是1号任务的工序。此后1号任务将被移动到*M*2上以执行下一道工序。由于*M*2是空闲的，因此1号任务的第二道工序将立即开始执行，这道工序将在第6个时刻完成（当前时刻2 + 工时4）。从时刻2开始，*M*1进入转换状态并将持续2个时间单元，这期间，*M*1的当前工序被设置为*C*，其完成时间为时刻4。

在时刻4，*M*1和*M*3都完成了它们自己的当前工序。由于*M*1完成的是一个“转换”工序，所以它开始执行新的任务。为此，从*M*1的队列中选择第一个任务——3号任务。3号任务第一个工序的长度为4，所以该工序的结束时间为8，8也同时成为*M*1的完成时间。*M*3上的2号任务在完成其第一道工序之后需移至*M*1上继续执行，由于*M*1正忙，所以2号任务被放入*M*1的队列，*M*3则进入转换状态，转换状态的结束时刻为5。

依此类推能够给出后续的事件序列。2号和4号任务在第12时刻完成，1号任务在第15时刻完成，3号任务在第19时刻完成。由于2号任务的长度为6，而它的完成时刻为12，所以2号任务在队列中所花费的等待时间为12-6=6个时间单元。类似地，4号任务在队列中的等待时间为12-4=8个时间单元，1号和3号任务的等待时间分别为8和11个时间单元。总的等待时间为33个时间单元。

可以确定这33个单元等待时间在3台机器上的具体分布。例如，4号任务在0时刻进入*M*3的队列，直到时刻5才开始被执行，所以该项任务在*M*3的队列中所花费的等待时间为5个时间单元。其他的任务都不需要在*M*3上等待，因此在*M*3上所花费的总的等待时间为5。仔细检查图6-14b，可以计算出在*M*1和*M*2上所花费的等待时间分别为18和10个时间单元。正如我们所预料的，各任务的等待时间之和（33）就等于在各机器上所花费的等待时间之和。

2. 高级仿真器设计

在设计仿真器时，假定所有的任务都在0时刻出现，并且在仿真过程中不再出现其他新的任务，此外还假定仿真过程将一直持续到所有任务都完成为止。

由于仿真器是一个相当复杂的程序，因此可以把它分解成若干个模块。仿真器所执行的主要任务是：输入数据，把各任务按其第一道工序放入相应队列；执行启动事件（即装入初始任务）；处理所有事件（即执行实际仿真）和输出队列等待时间。分别使用一个C++函数来实现仿真器的每项任务。每个函数都可能引发诸如NoMem和BadInput（非法的输入数据）的异常。主函数见程序6-11，其中的catch语句可用若干条catch语句来替代，每条catch语句用于引发一种异常，并输出不同的信息。练习19要求你完成这项工作。

程序6-11 工厂仿真的主函数

```
void main(void)
{//工厂仿真
  try {
    InputData();   // 获取机器和任务数据
    StartShop();   // 启动
    Simulate();    // 仿真
    OutputStats();} // 输出机器等待时间
  catch (...) {
    cout << "An exception has occurred" << endl;}
}
```

3. 类Task

在设计程序6-11中所调用的四个函数之前，必须设计所需要的数据对象，这些数据对象包括工序、任务、机器和事件表。每个工序都由两部分构成： machine（该工序将在这台机器上处理）和time（完成该工序所需要的时间）。程序6-12给出了类Task的定义。由于对机器从1至m编号，所以machine是int类型（也可使用类型unsigned）。假定所有的时间都是整数，time的类型被定义为long以便于执行长时间的仿真。类Task有两个友元：类Job和函数MoveToNextMachine。把Job定义为Task的友元是因为Job需要访问Task的私有成员。也可以避免这种授权，方法是为Task定义一些共享成员函数，用以设置和获取machine和time的值。

4. 类Job

每项任务都有一个相关的工序表，每道工序按表中的次序执行。因此，工序表可以被描述成一个队列TaskQ。为了确定一项任务所花费的总共的等待时间，需要知道该任务的长度和完成时间。完成时间由计时来确定，而长度则为各工序的工时之和。为了确定任务的长度，我们为Job定义了一个私有数据成员Length。

程序6-12给出了类Job的定义。TaskQ被定义成一个链表队列，用来保存各道工序。由于在

输入数据时已经知道一项任务所包含的工序数目，因此可以把 TaskQ定义成一个指针，以便动态构造一个足够大的公式化队列，以容纳所期望数量的工序。对于这种受限制的仿真器来说，这个定义工作得很好，因为我们假定在仿真开始之后不再有新的任务进入工厂。若使用链表队列，则只要完成一道工序，即可释放该工序所占用的队列节点，被释放的节点可重新用来构造新任务的工序队列。若使用公式化队列，则仅当整个任务都已经完成时才释放空间，因此，仿真很可能因为空间不足而失败，即使中只剩下很少量的工序未被完成。

私有成员ArriveTime用于记录一项任务进入当前机器队列的时间，可用来确定任务在这个队列中的等待时间。任务标识符存储在ID之中，仅在输出任务的总等待时间时，才会使用该标识符。

共享成员函数AddTask用于向任务的工序队列中添加一道工序，该工序将在机器 p 上执行，需要t 个时间单元。该函数仅用于数据输入过程。当从一个机器队列中移出一项任务并将其变成活动状态时，需要使用共享成员函数 DeleteTask。该函数从工序队列（工序队列用于保存尚未被处理的工序）中删除排在队首的工序，然后返回该工序的时间并将其加入所属任务的长度中。当任务的最后一道工序完成时， Length 的值即为该任务的长度。

程序6-12 类Task和Job

```
class Task {
  friend class Job;
  friend bool MoveToNextMachine(Job*);
  private:
    long time;
    int  machine;
};

class Job {
  friend bool MoveToNextMachine(Job*);
  friend Job* ChangeState(int);
  friend void Simulate();
  friend class Machine;
  public:
    Job(long id) {ID = id;
          Length = ArriveTime = 0;}
    void AddTask(int p, long t) {
          Task x;
          x.machine = p;
          x.time = t;
          TaskQ.Add(x);}
    long DeleteTask() {  // 删除下一工序
          Task x;
          TaskQ.Delete(x);
          Length += x.time;
          return x.time;}
  private:
    LinkedQueue<Task> TaskQ;
    long Length;      // 工序时间
    long ArriveTime;  // 到达当前队列的时间
```

```
    long ID;        // 任务标识符
};
```

5. 类Machine

每台机器都包含如下三个属性：转换时间、当前任务和等待队列。由于任务是一个相当大的对象（有自己的工序队列和其他数据成员），因此把保存等待任务的队列定义成一个指针队列（每个指针指向一个任务）可以大大提高处理效率，这样每个机器队列的操作处理的是指针（很小的对象）而不是任务。

由于每项任务任意时刻只会在一台机器上，所以所有队列总的空间需求与任务的数目直接相关。不过，任务在各个机器队列中的分布会随着仿真的进行而不断变化。某一时刻出现几个很长的队列是完全可能的，而稍后这些队列可能会变短，其他队列会变长。如果采用公式化队列，则必须把每个机器队列的大小都定义成最大可能的值。（否则必须动态调整队列所在数组的大小），此时，需要存储m*(n-1) 个任务指针（m 是机器的数目，n 是任务的数目）。如果使用链表队列，则只需要存储n 个任务指针和n 个链接域。因此，我们使用链表队列。

程序6-13给出了类Machine 的定义。私有成员JobQ、ChangeTime、TotalWait、NumTasks和Active分别表示指向等待任务的指针队列、机器的转换时间、在该台机器上已花费的总的等待时间、该机器所完成的工序数目和指向当前活动任务的指针。当机器空闲或处于转换状态时，活动任务的指针为0。

共享成员函数IsEmpty 当且仅当任务队列为空时返回 true。共享成员函数AddJob向等待队列中添加一项任务。函数SetChange用于在输入期间设置ChangeTime的值。函数Status用于返回该机器迄今为止总的等待时间及完成的工序数目。

所有机器的完成时间被存储在一个事件表中。为了从一个事件转向下一个事件，必须确定最小的机器完成时间。仿真器还需要一个设置一台机器的完成时间的操作，每当一个新任务被调度到一台机器上运行时就要执行该操作。当一台机器变成空闲时，其完成时间被设置成一个很大的数。

程序6-13 类Machine

```
class Machine {
  friend Job* ChangeState(int);
  public:
    Machine() {TotalWait = NumTasks = 0;
               Active = 0;}
    bool IsEmpty() {return JobQ.IsEmpty();}
    void AddJob(Job* x) {JobQ.Add(x);}
    void SetChange(long w) {ChangeTime = w;}
    void Stats(long& tw, long& nt)
          {tw = TotalWait;
           nt = NumTasks;}
  private:
    LinkedQueue<Job*> JobQ; // 等待队列
    long ChangeTime;  //机器转换时间
    long TotalWait;   // 机器总的等待时间
    long  NumTasks;   // 所处理的工序数目
    Job *Active;      // 指向当前任务的指针
};
```

6. 类EventList

程序6-14给出了类EventList的定义，它用一个一维数组 FinishTime来处理事件，其中FinishTime[p]表示机器p的完成时间。初始化时，所有的机器都是空闲的，它们的完成时间都被设置为BigT。机器数目用变量NumMachines来记录。

函数NextEvent用于返回下一个事件对应的机器p和完成时间t。对于一个有m台机器的工厂，寻找最小的完成时间所需要的时间为 $\Theta(m)$，因此函数 NextEvent的复杂性为 $\Theta(m)$。函数SetFinishTime 用来设置一台机器的完成时间，其复杂性为 $\Theta(1)$。在第9章中将看到两个数据结构——堆和最左树，也可以用它们来描述事件。如果使用堆或最左树， NextEvent和SetFinishTime的复杂性均变成O(logm)。如果所有任务的工序总数为T，仿真器将分别调用$\Theta(T)$次NextEvent 和$\Theta(T)$次SetFinishTime。在程序6-14中，调用NextEvent和SetFinishTime所花费的时间为$\Theta(Tm)$，若使用堆或最左树，所需要的时间为O(Tlogm)。虽然堆或最左树更复杂，但当m比较大时，它们可以大大加快仿真过程。

程序6-14 类EventList

```
class EventList {
  public:
    EventList(int m, long BigT);
    ~EventList(){delete [] FinishTime;}
   void NextEvent(int& p, long& t);
     long NextEvent(int p) {return FinishTime[p];}
     void SetFinishTime(int p, long t)
          {FinishTime[p] = t;}
  private:
    long *FinishTime; // 完成时间数组
    int NumMachines;  // 机器总数
};

EventList::EventList(int m, long BigT)
{//对m台机器的完成时间进行初始化
  if (m < 1) throw BadInitializers();
  FinishTime = new long [m+1];
  NumMachines = m;
  // 所有机器均为空闲
  for (int i = 1; i <= m; i++)
    FinishTime[i] = BigT;
}

void EventList::NextEvent(int& p, long& t)
{// 返回下一个事件所对应的机器和完成时间
  // 寻找具有最小完成时间的机器
  p = 1;
  t = FinishTime[1];
  for (int i = 2; i <= NumMachines; i++)
    if (FinishTime[i] < t) {// i 较早完成
      p = i;
      t = FinishTime[i];}
}
```

7. 全局变量

程序6-11的四个函数中所使用的全局变量见程序 6-15。多数全局变量的含义从变量名中就可以看出来。Now用来记录当前的模拟时间，每次发生一个新的事件时，就会修改 Now的值。LargeTime用来表示空闲机器的完成时间。

程序6-15 仿真器中所使用的全局变量

```
//全局变量
long Now = 0;                  //当前时间
int m;                         // 机器数
long n;                        // 任务数
long LargeTime = 10000;        // 空闲机器的完成时间
EventList *EL;                 //指向事件表的指针
Machine *M;                    // 机器数组
```

8. 函数InputData

函数InputData（见程序6-16）首先输入工厂中的机器数和任务数，然后创建初始事件表*EL（其中每台机器的完成时间均被设置为LargeTime）和机器数组M。接下来输入每台机器的转换时间并依次输入每项任务。对于每项任务，首先输入该任务所包含的工序数目，然后按(machine, time)的形式输入每个工序。任务的第一道工序所对应的机器记录在变量 p 中。当一个任务的所有工序都已输入完毕时，该任务（实际上是指向该任务的指针）被放入机器 p 的队列中。

程序6-16 输入工厂数据

```
void InputData()
{// 输入工厂数据
  cout << "Enter number of machines and jobs" << endl;
  cin >> m >> n;
  if (m < 1 || n < 1) throw BadInput();

  // 创建事件表和机器队列
  EL = new EventList(m,LargeTime);
  M = new Machine [m+1];

  // 输入机器等待时间
  cout << "Enter change-over times for machines" << endl;
  for (int j = 1; j <= m; j++) {
    long ct;  // 转换时间
    cin >> ct;
    if (ct < 0) throw BadInput();
    M[j].SetChange(ct);
    }
  // 输入n个任务
  Job *J;
  for (int i = 1; i <= n; i++) {
    cout << "Enter number of tasks for job " << i << endl;
    int tasks;  //工序数目
```

```
    int first;  // 任务的第一道工序所对应的机器
    cin >> tasks;
    if (tasks < 1) throw BadInput();
    J = new Job(i);
    cout << "Enter the tasks (machine, time)" << " in process order" << endl;
    for (int j = 1; j <= tasks; j++) {// 输入工序
      int p;    // 机器数目
      long tt;  // 工时
      cin >> p >> tt;
      if (p < 1 || p > m || tt < 1) throw BadInput();
      if (j == 1) first = p; // 任务的第一台机器
      J->AddTask(p,tt);  // 添加到工序队列
      }
    M[first].AddJob(J);   // 把任务添加到第一道工序所对应的机器
    }
}
```

9. 函数StartShop和ChangeState

为了启动仿真，需要从每个机器任务队列中取出第一个任务并放到该机器上执行。由于每台机器的初始状态为空闲状态，为了加载初始任务，需要把机器从空闲状态变成活动状态（在仿真进行过程中也如此）。函数ChangeState(i)可用来转换机器i的状态。在程序6-17中，为了启动仿真，只需对每台机器调用ChangeState即可。

程序6-17 启动仿真

```
void StartShop()
{// 加载每台机器上的第一项任务
  for (int p = 1; p <= m; p++)
    ChangeState(p);
}
```

程序6-18给出了函数ChangeState的代码。如果机器p空闲或处于转换状态，则ChangeState返回0，否则返回指向当前正在处理的任务的指针。此外，ChangeState(p) 可改变机器p 的状态。如果机器p 本来为空闲或处于转换状态，则让p 处理其等待队列中的下一项任务，如果队列为空，则将机器的状态设置为空闲。如果机器p 正处理完一项任务，则使其进入转换状态。

如果M[p].Active为0，则机器p 要么空闲，要么处于转换状态， LastJob所返回的任务指针为0。如果机器的任务队列为空，则将该机器变成空闲状态，并将其完成时间设置为LargeTime。如果任务队列不为空，则删除队列中的第一个任务并将其变成该机器的活动任务，该任务在队列中所花费的等待时间被累加到该机器的总等待时间之中，同时将该机器所处理的工序数目增加1。接下来从任务的工序队列中删除将要处理的工序，并将机器的完成时间设置为新工序完成的时间。

如果M[p].Active不为0，则表明机器p正忙，需返回指向当前任务的指针，为此将该指针存储在LastJob中。机器的状态将变成转换状态，并持续ChangeTime个时间单元。

程序6-18 修改机器状态

```
Job* ChangeState(int p)
```

```
{// 机器p上的工序已完成，调度下一个任务
  Job* LastJob;
  if (!M[p].Active) {// 空闲或转换状态
    LastJob = 0;
    // 结束等待，准备处理下一项任务
    if (M[p].JobQ.IsEmpty()) // 没有处于等待状态的任务
        EL->SetFinishTime(p,LargeTime);
    else {// 取出任务Q并处理之
      M[p].JobQ.Delete(M[p].Active);
      M[p].TotalWait +=
          Now - M[p].Active->ArriveTime;
      M[p].NumTasks++;
      long t = M[p].Active->DeleteTask();
      EL->SetFinishTime(p, Now + t);}
    }
  else {// M[p]上的工序刚刚完成
      // 进入转换状态
      LastJob = M[p].Active;
      M[p].Active = 0;
      EL->SetFinishTime(p, Now + M[p].ChangeTime);}
  return LastJob;
}
```

10. 函数Simulate和MoveToNextMachine

程序6-19中，函数Simulate对所有的事件进行循环，直到最后一项任务结束。n是尚未完成的任务数目，因此，当n=0时，while循环将结束。在while的每次循环过程中，需确定下一个事件的到达时间并将该时间存入Now。对于产生事件的机器p，需改变其状态，如果该机器刚刚处理完一道工序(J不为0)，则调度任务*J并处理该任务的下一道工序。函数MoveToNextMachine可用来对任务进行调度。如果任务*J中不再有未处理的工序，则表明该任务已完成，函数MoveToNextMachine将返回false，同时将n减1。

函数MoveToNextMachine（见程序6-20）首先检查任务*J中是否有尚未被处理的工序。如果没有，则表明该任务已完成，输出该任务的完成时间和等待时间。函数返回 false意味着任务*J不必再移动到下一台机器上去处理。

若任务*J尚未完成，则需确定该任务的下一站——机器p，同时将*J加入p的等待队列之中。如果机器p为空闲，则调用函数ChangeState来改变其状态。

程序6-19 处理所有任务

```
void Simulate()
{//处理完所有n项任务
  int p;
  long t;
  while (n) {// 至少剩下一个任务
    EL->NextEvent(p,t); // 下一个完工的机器
    Now = t; // 当前时间
    // 修改机器 p的状态
    Job *J = ChangeState(p);
```

```
    //把任务J 移至下一台机器
    // 如果 J 已完成，则将n减1
    if (J && !MoveToNextMachine(J)) n--;
    }
}
```

程序6-20 把一项任务移至下一道工序对应的机器

```
bool MoveToNextMachine(Job *J)
{// 把任务J 移至下一道工序所对应的机器
 // 如果所有工序都已处理完毕，则返回 false
  if (J->TaskQ.IsEmpty()) {// 没有未处理的工序
    cout << "Job " << J->ID << " has completed at " << Now << " Total wait was " << (Now-J->Length)
         << endl;
    return false;}
  else {// 有未处理的工序
      // 获取下一道工序所对应的机器
      int p = J->TaskQ.First().machine;
      // 放入p 的等待队列
      M[p].AddJob(J);
      J->ArriveTime = Now;
      // 如果 p 空闲，则立即执行该工序
      if (EL->NextEvent(p) == LargeTime) {
        // 机器为空闲
        ChangeState(p);}
      return true;}
}
```

11. 函数OutputStats

由于一个任务的完成时间和等待时间已由函数MoveToNextMachine输出，因此函数OutputStats只需输出完成所有任务所需时间（它也是最后一个任务的完成时间，在 MoveToNextMachine中已被输出过一次）以及每台机器的统计信息（该机器总的等待时间以及所处理的工序数目）。程序6-21给出了相应的代码。

程序6-21 输出每台机器的等待时间

```
void OutputStats()
{// 输出机器的等待时间
  cout << "Finish time = " << Now << endl;
  long TotalWait, NumTasks;
  for (int p = 1; p <= m; p++) {
    M[p].Stats(TotalWait, NumTasks);
    cout << "Machine " << p << " completed " << NumTasks << " tasks" << endl;
    cout << "The total wait time was " << TotalWait;
    cout << endl << endl;
    }
}
```

练习

11. 采用k 个缓冲铁轨，对于所有可能的车厢排列，程序 6-7都能成功地进行车厢重排吗？试证明你的结论。

12. 重写程序6-7，假定任意时刻缓冲铁轨i 中最多只能有s_i 节车厢。具有最小s_i 的铁轨作为直接通道。

13. 设计一个完整的用于寻找电路布线的C++程序。程序中应包含如下函数：Welcome函数（用来显示程序的名称和功能）；输入函数，用于输入网格的大小、封锁的和空白的网格位置、电线的端点；FindPath函数（见程序6-9）；输出函数，用于输出网格及路径。使用书中的例子测试程序的正确性。

14. 设计一个完整的用于进行图元识别的C++程序。程序中应包含如下函数：Welcome函数（用来显示程序的名称和功能）；输入函数，输入图像的大小及单色图像数据；函数 Label（见程序6-10）；输出函数，用于输出识别后的图像，不同图元中的像素用不同的颜色来表示。使用书例中的图像测试程序的正确性。

15. 采用公式化队列重写函数 Label（见程序6-10）。使用公式化队列来代替链表队列，会有哪些优点和缺点？

16. 采用堆栈来重写函数Label（见程序6-10）。使用堆栈来代替队列，会有哪些优点和缺点？

17. 能否把程序5-5中的堆栈换成队列？为什么？

18. 能否把程序5-12中的堆栈换成队列？为什么？

19. 重写程序6-11，采用多个catch语句分别输出不同的错误信息。每个 catch语句应对应一种执行期间可能出现的异常。

*20. 设计一个增强型的工厂仿真器，允许为任务指定各个相邻工序之间最少的等待时间。在完成一项任务的每一道工序（包括最后一道工序）时，仿真器必须将该任务变成等待状态。因此，当一项任务的一道工序完成时，该任务被立即放入下一个队列，此时，该任务进入等待状态。当一台机器准备启动一道新工序时，它必须跳过队列前部仍处于等待状态的任务。可以把被跳过的任务移动到队列的尾部。

*21. 设计一个增强型的工厂仿真器，允许在仿真期间有新的任务进入。仿真过程在预先指定的时刻结束，尚未完成的任务均处于未完成状态。

6.5 参考及推荐读物

6.4.2节中的电路布线算法选自N.Sherwani. *Algorithms for VLSI Physical Design Automation* 第2版. Kluwer Academic，1995。书中详细讨论了该算法，并给出了其他算法。