第 9 章

队列(QUEUE)

本章内容

- 9.1 定义和应用
- 9.2 抽象数据类型
- 9.3 数组描述
- 9.4 链表描述
- 9.5 应用

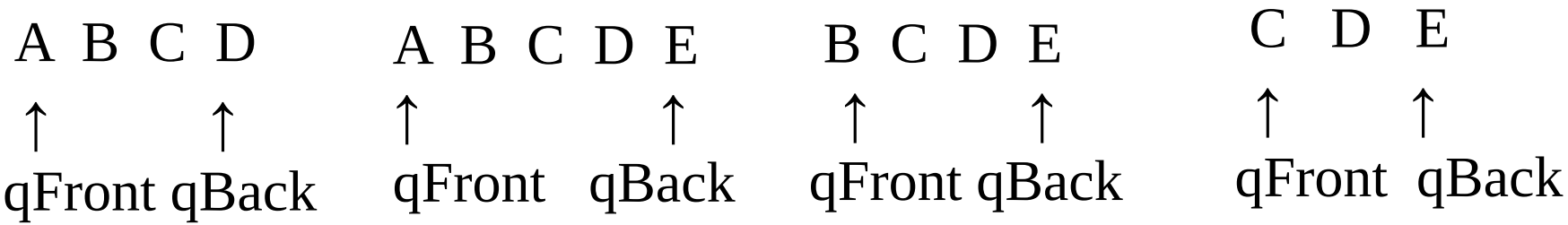
9.1 定义与应用

- 定义:
  - 队列（queue）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(queueBack).
  - 删除元素的那一端被成为队首(queueFront).

- $e_0, e_1, e_2, \cdots, e_i, \cdots, e_{n-1}$

  ↑ ↑

  queueFront queueBack



●队列是一个先进先出（ first-in-first-out, FIFO）的线性表。

9.2 抽象数据类型

抽象数据类型queue {

实例

元素的有序线性表，一端称为队首，另一端称为队尾；

操作

```
empty();  //队列为空时返回true，否则返回false ；
size();   //返回队列中元素个数
front();  //返回队列头元素；
back();   //返回队列尾元素；
pop();    //删除队列头元素
push(x);  //将元素x加入队尾
```

}

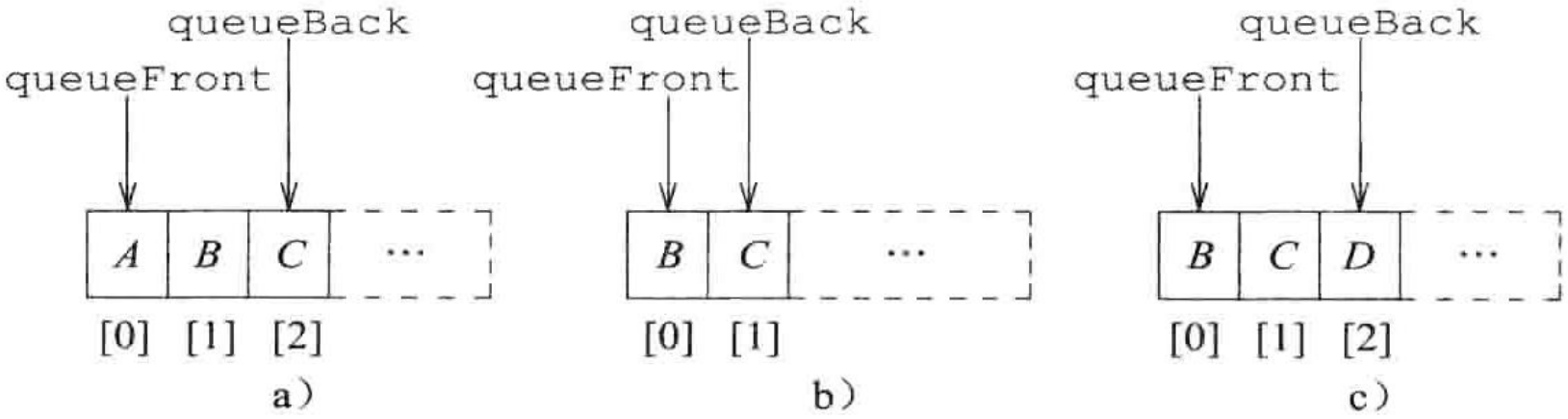
C++抽象类queue

```
template <class T>
class queue{
  public:
    virtual ~queue() {}
    virtual bool empty() const = 0;
          //队列为空时返回true，否则返回false
    virtual int size() const = 0;
          //返回队列中元素个数
    virtual T& front() = 0;
          //返回队列头元素;
    virtual T& back() = 0;
          //返回队列尾元素
     virtual void pop() = 0;
          //队列头元素
    virtual void push(const T& theElement) = 0;
          //将元素theElement加入队尾
}
```

队列的描述

- 数组描述
- 链表描述

9.3 数组描述-1



- 映射公式-1:

  $location(i) = i$

- 队首元素: queue[0];
- queueFront总是为0;
- queueBack始终是最后一个元素的位置

数组描述-1

- 队列的长度： queueBack + 1
- 空队列： queueBack=-1
-
- 队首元素出队：

  queue[0..queueBack-1]← queue[1..queueBack];

  queueBack=queueBack-1;

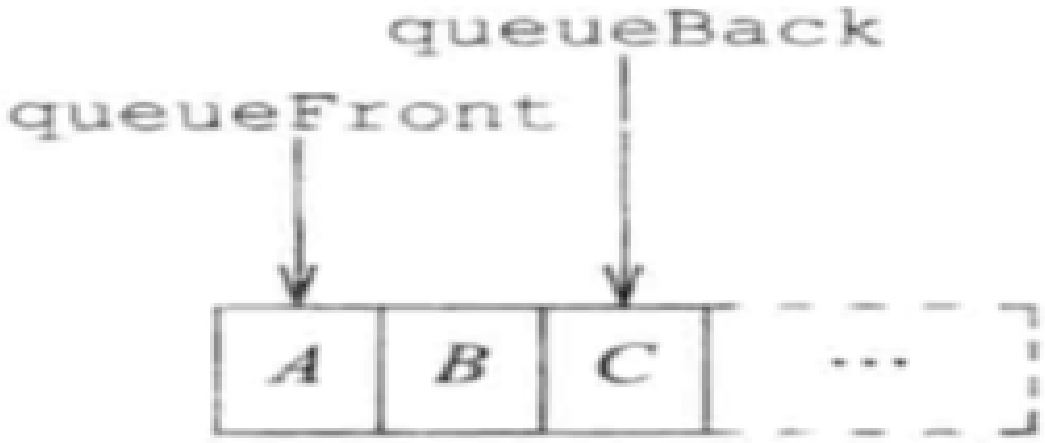
  Θ(n)

- 元素x入队： queueBack=queueBack+1;

  queue[queueBack]=x;

  O(1)

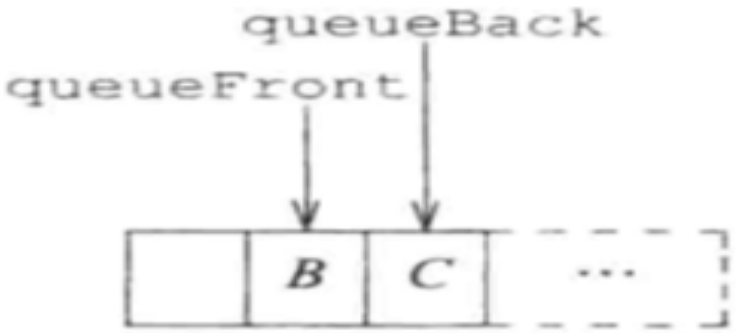
数组描述-2



- 映射公式-2:
  location(i) = location(队首元素) + i
- queueFront = location(队首元素)
- queueBack = location(队尾元素)
- 空队列：queueBack < queueFront

数组描述-2

- 删除队首元素
  - queueFront=queueFront+1;
  - Θ(1)

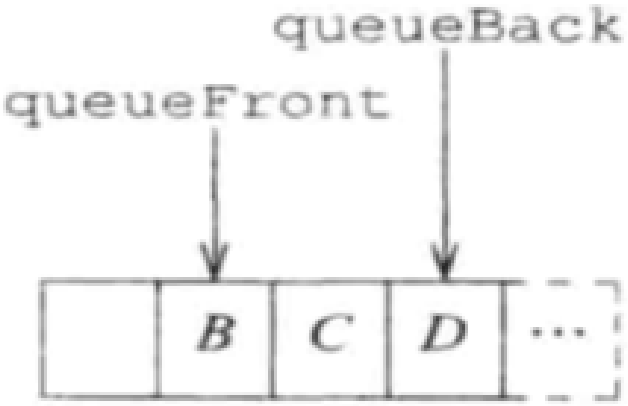


- 元素D入队

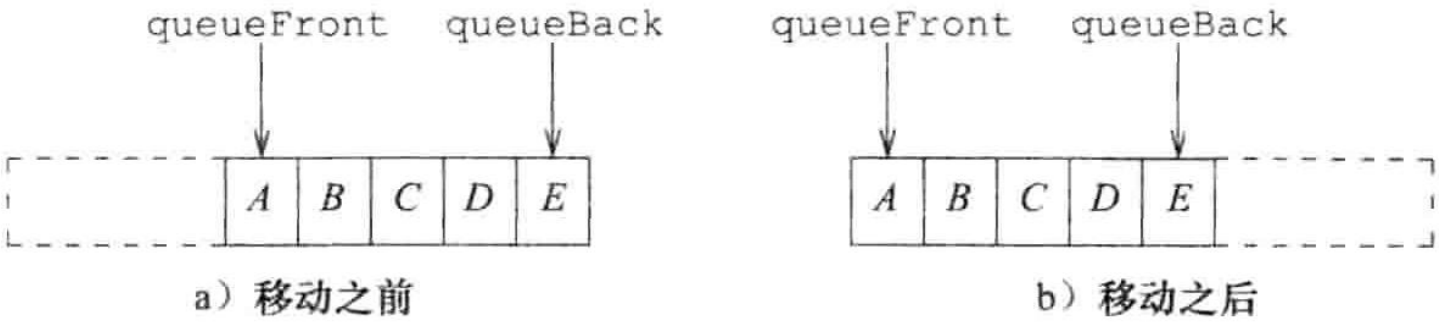
  queueBack=queueBack+1;

  queue[queueBack]=x;

  Θ(1)



数组描述-2



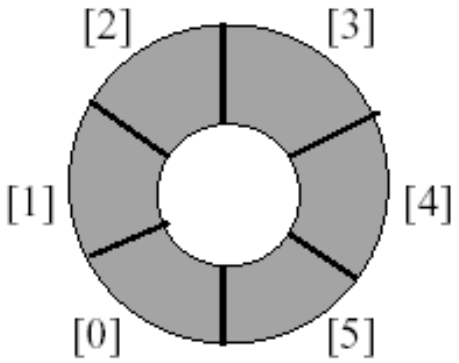
- 元素x入队:
- 当 **queueBack = arrayLength-1 且queueFront > 0** ？
- 平移队列
  - queue[0..queueBack-queueFront+1]← queue[queueFront..queueBack];
  - queueBack=queueBack+1;queue[queueBack]=x;
- 时间复杂性：$\Theta(n)$

循环队列

- 描述队列的数组

queue[]

- 描述队列的数组被视为一个环



- 公式-3：

location(i) = (location(队首元素) + i ）% arrayLength

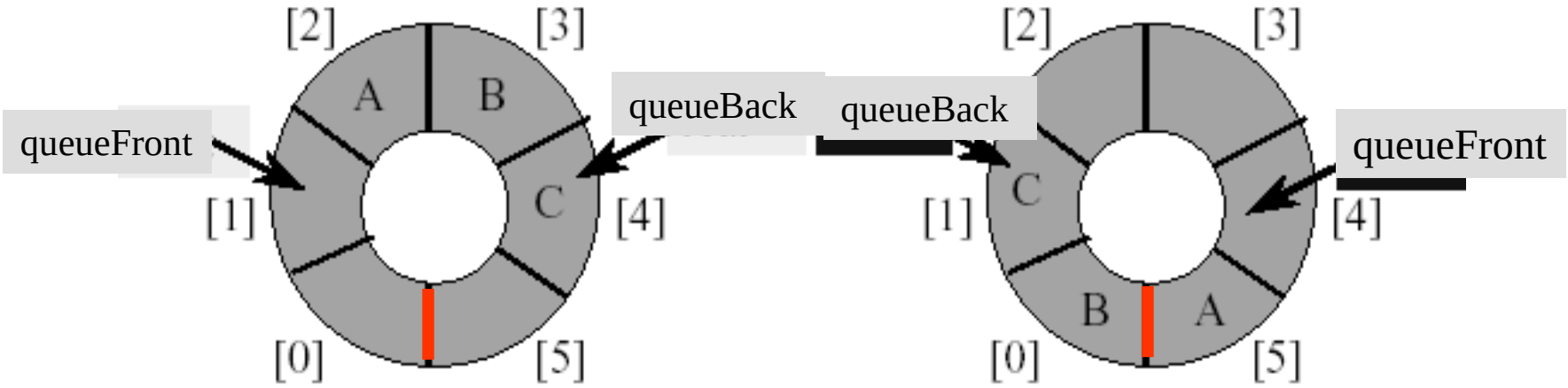
↓

location(i+1) = (location(i) + 1) % arrayLength

➔ **循环队列**

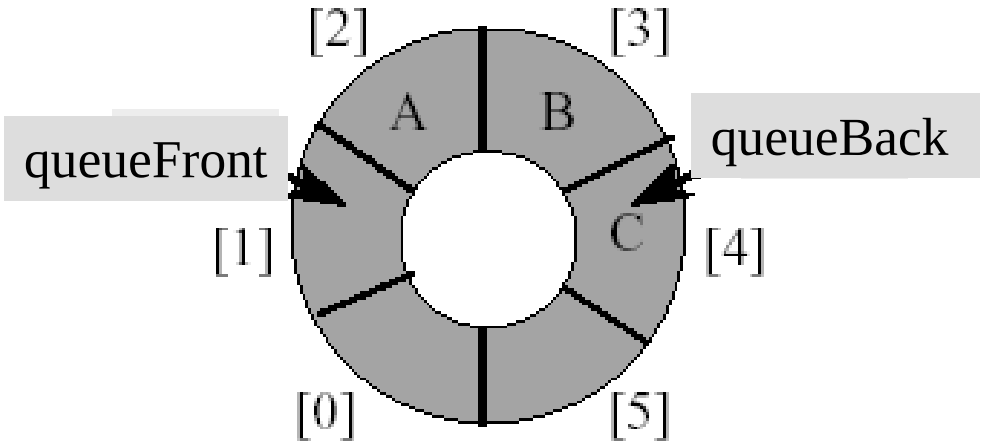
循环队列

- queueFront: 指向队列首元素的下一个位置（逆时针方向）。
- queueBack:　最后一个元素的位置



- 队列首元素的位置：
  - (queueFront+1) % arrayLength

循环队列



- 空队列: queueFront=queueBack
- 队列为满的条件: queueFront=queueBack
- 如何区分两种情况: 队列为空和队列为满?

  队列中最多元素个数=arrayLength-1
- 队列为满的条件:
  - queueFront=(queueBack+1) % arrayLength
- 队列满时,队列容量可进行加倍处理：读P210,P211程序9-3

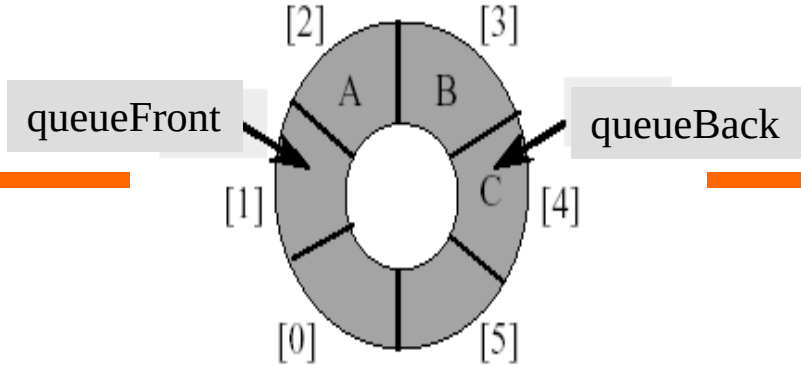
循环队列类arrayQueue

```
template<class T>
class arrayQueue ： public  queue<T>;
{public:
    arrayQueue (int initialCapacity = 10);
    ~ arrayQueue () { delete [] queue; }
    bool empty() const { return queueFront = = queueBack; }
    int size() const
      { return (arrayLength+ queueBack
                                    - queueFront) % arrayLength; }
   T& front() const; //返回队首元素
   T& back() const; // 返回队尾元素
    void pop(); //删除队首元素
   void push(const T& theElement); //元素插入到队尾
 private:
     int queueFront; //与第一个元素在反时针方向上相差一个位置
     int queueBack; // 指向最后一个元素
    int arrayLength; // 队列数组容量
    T *queue; // 元素数组
};
```

arrayQueue构造函数

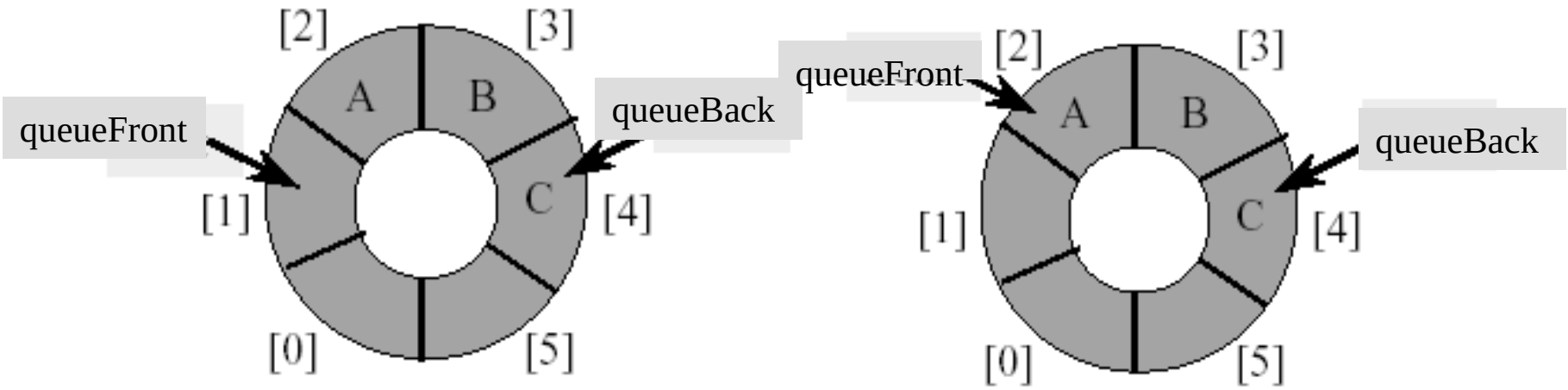
```
template<class T>
arrayQueue <T>:: arrayQueue(int initialCapacity = 10);
{ // 构造函数
   if (initialCapacity < 1)  …..//输出错误信息，抛出异常
   arrayLength = initialCapacity;
    queue = new T[arrayLength];
   queueFront = queueBack = 0;
}
```

方法 ‘front’ ‘back’



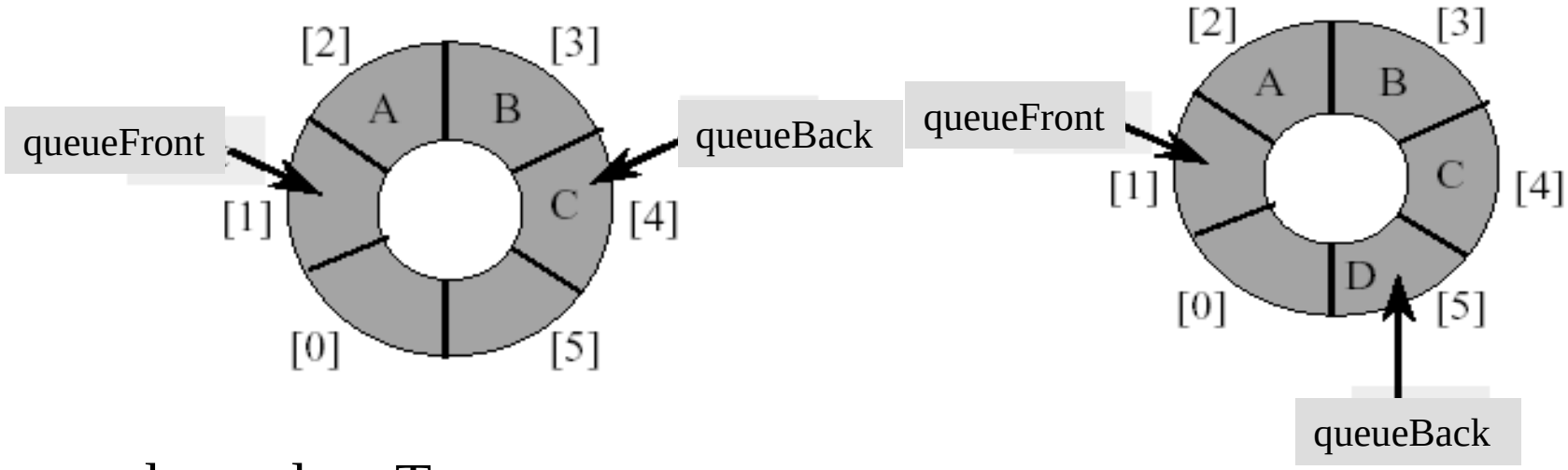

```
template<class T>
 T& arrayQueue <T>::front() const
{//返回队首元素
   if (queueFront = = queueBack) throw  QueueEmpty();
    return queue[(queueFront + 1) % arrayLength];
 }

template<class T>
  T& arrayQueue <T>::back() const
  {// 返回队尾元素
    if (queueFront = = queueBack) throw  QueueEmpty();
      return queue[queueBack];
  }
```

方法 ‘pop’




```
template<class T>
void  arrayQueue <T>::pop()
{// 删除队首元素
  if (queueFront = = queueBack) throw  QueueEmpty();
    queueFront = (queueFront + 1) % arrayLength;
    queue[queueFront].~T();
}
```

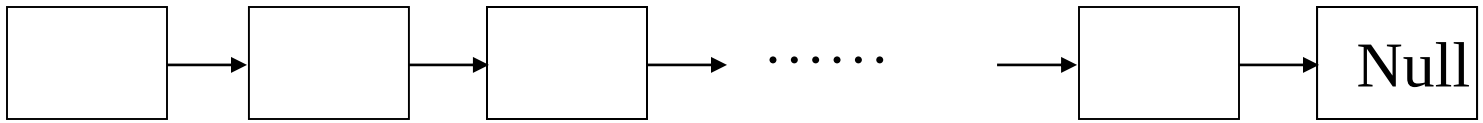
方法 ‘push’




```
template<class T>
void  arrayQueue <T>::push(const T& theElement)
 {// 把元素theElement添加到队列的尾部
    //如果队列空间满，则加倍数组长度
    if (queueBack+1) % arrayLength == queueFront)
        …….//加倍数组长度,程序9-3
    queueBack = (queueBack + 1) % arrayLength;
    queue[queueBack] = theElement;
}
```

9.4 链表描述

- 使用链表来描述一个队列



- 两种选择:
  1)表头为 queueFront，表尾为 queueBack
  2)表头为 queueBack，表尾为 queueFront

链表描述

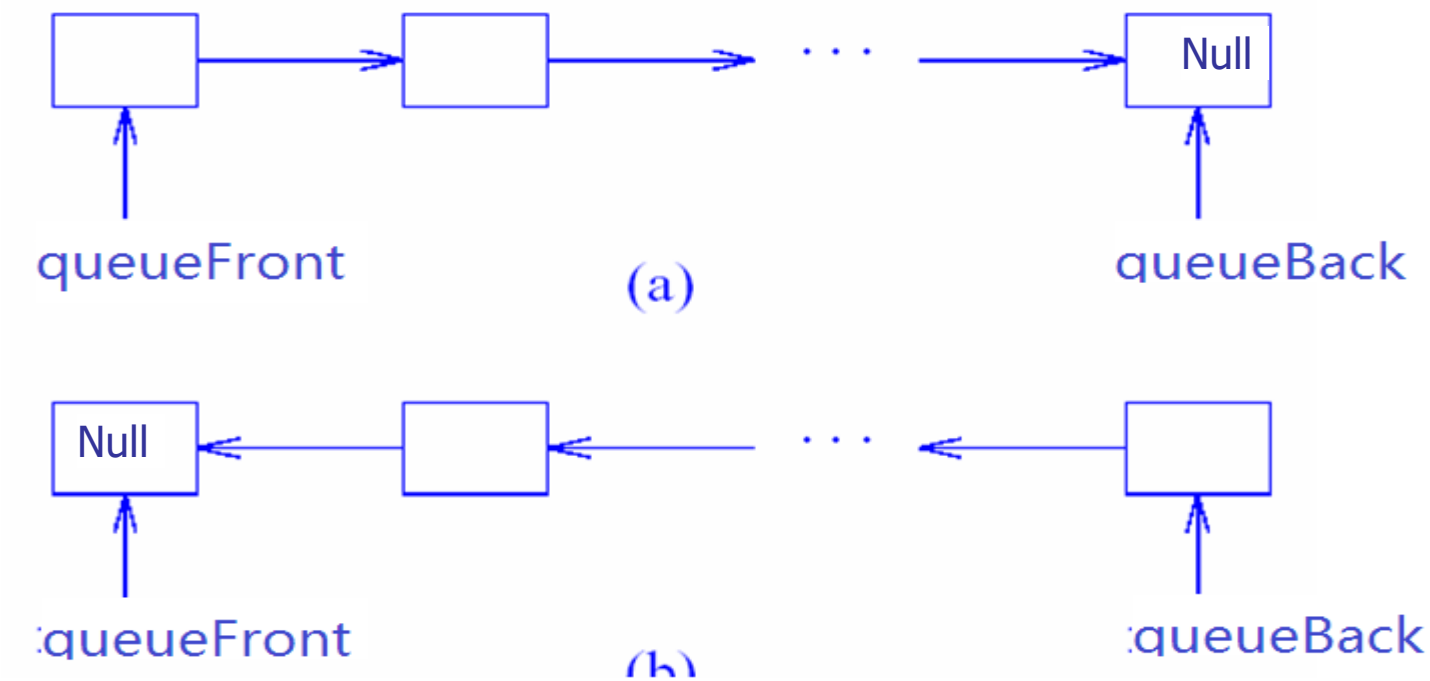


图9-8　链接队列

向链表队列中添加元素

图 9-9 (a)　向图 9-8 (a)中插入元素

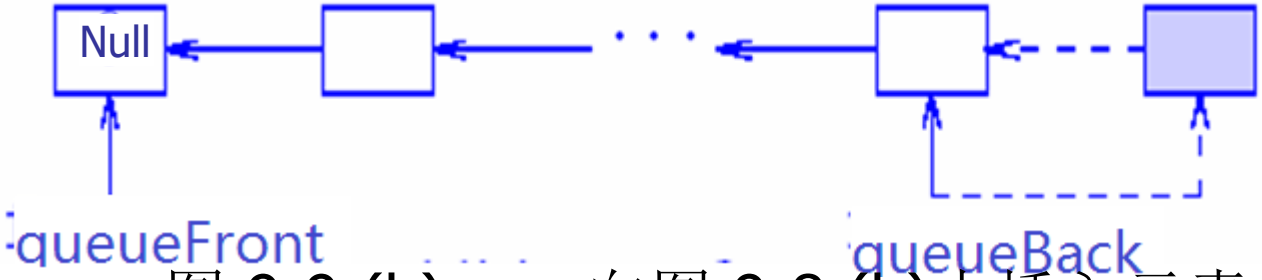


图 9-9 (b)　向图 9-8 (b)中插入元素

- 图 9-9 (a)的时间复杂性?
- 图 9-9 (b) 的时间复杂性?

从链表队列中删除元素

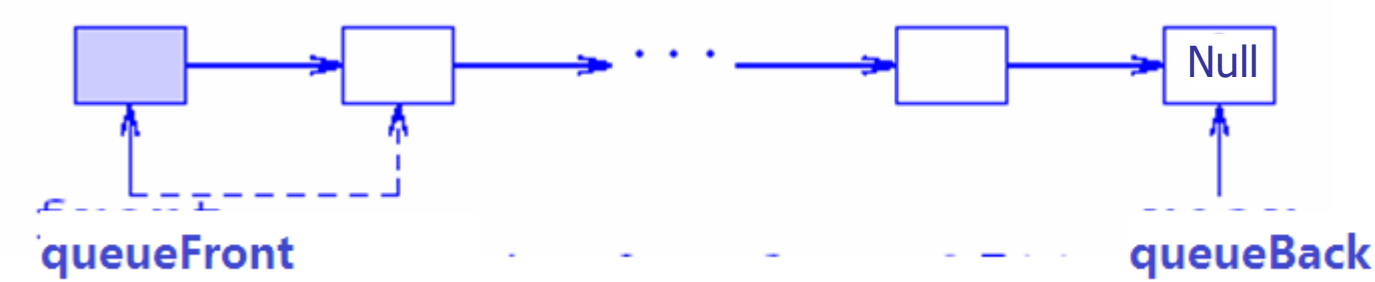


图 9-10 (a)　从图 9-8 (a)中删除元素

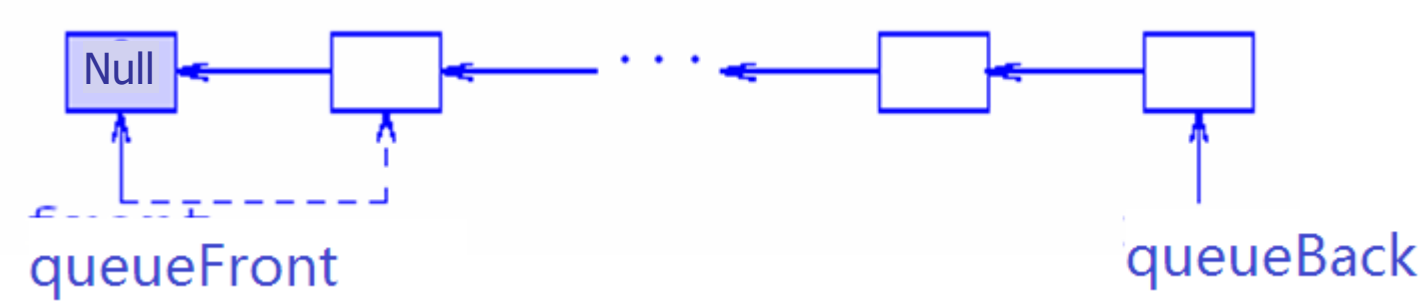
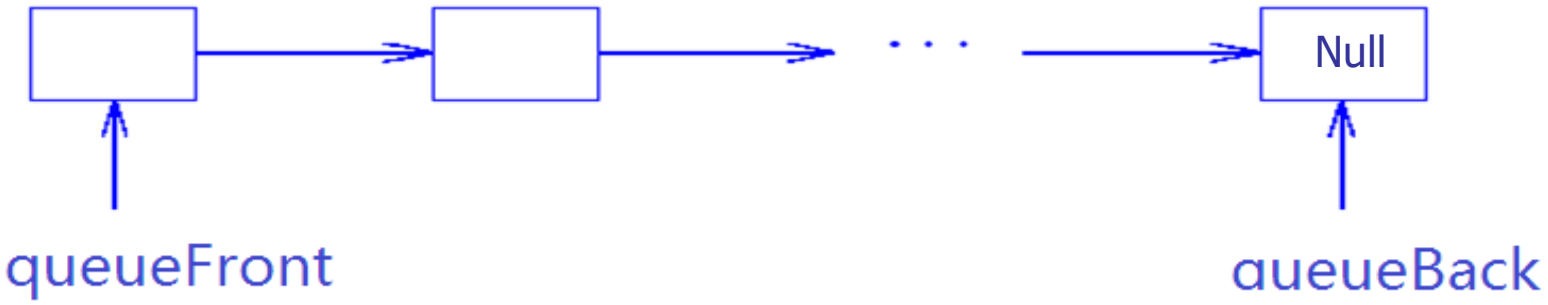


图 9-10 (b)　从图 9-8 (b)中删除元素

图 9-10 (a)的时间复杂性?
图 9-10 (b) 的时间复杂性?

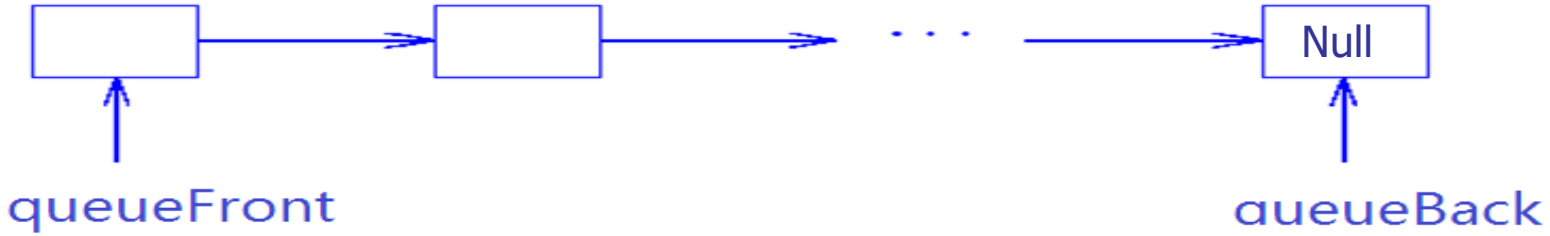
队列的链表描述



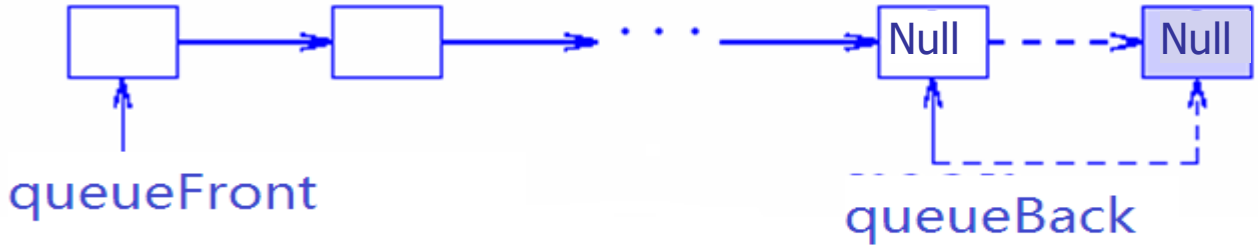
- 如何实现队列的链表描述?
  - 定义LinkedQueue为一个基类
  - 把类LinkedQueue 定义为extendedChain类（见程序6-12）的一个派生类

```
template<class T>
class LinkedQueue ： public queue <T>;
{ public:
      LinkedQueue(int initialCapacity = 10);
      ~ LinkedQueue();
      bool empty() const {return queueSize == 0; }
      int size() const { return queueSize; }
      T& front();
      T& back();
      void push(const T& theElement);
      void pop();
private:
      chainNode<T>* queueFront;  //队列首指针
      chainNode<T>* queueBack;  //队列尾指针
      int  queueSize;     //队列中元素个数
};
```

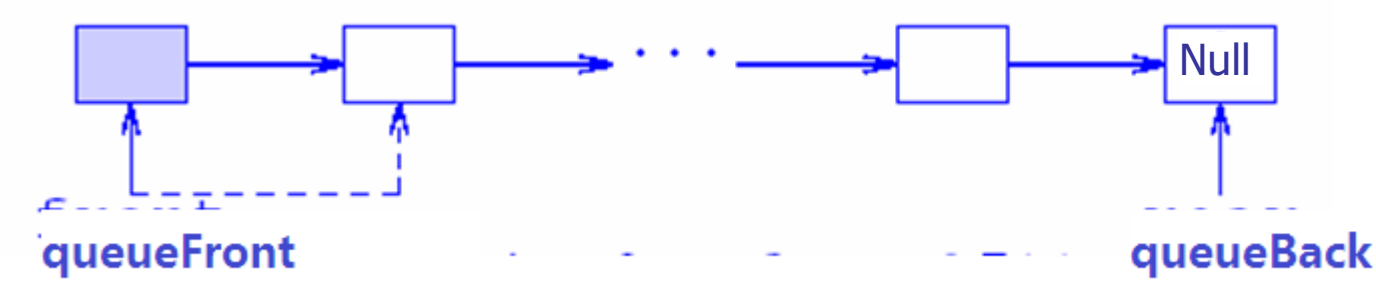
LinkedQueue类




```
template<class T>
T& LinkedQueue<T>::front()
{  // 返回队列首元素
 if (queueSize == 0) throw  QueueEmpty();
  return queueFront ->element;
}

template<class T>
T& LinkedQueue<T>::back()
{// 返回队列尾元素
   if (queueSize == 0) throw  QueueEmpty();
       return queueBack ->element;
}
```

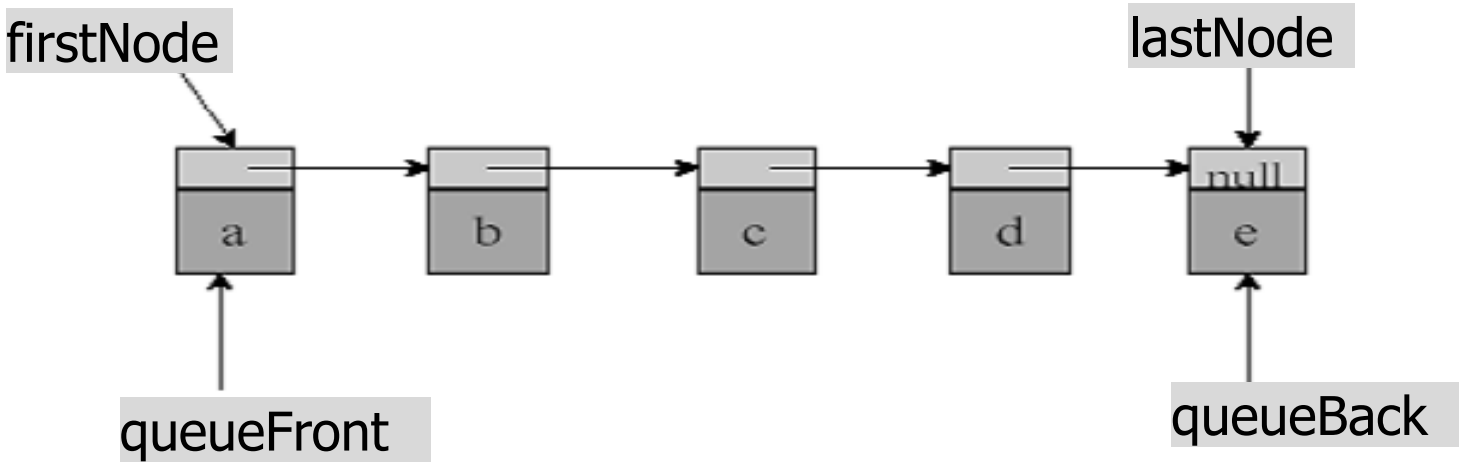



```
template<class T>
Void LinkedQueue<T>::push(const T& theElement)
{// 把元素theElement加入 到队尾

 // 为新元素申请节点
   chainNode<T> * newNode =
                          new chainNode<T>(theElement,NULL);
// 在队列尾部添加新节点
   if  (queueSize != 0)
            queueBack->next = newNode; //队列不为空
  else   queueFront = newNode;                  // 队列为空
  queueBack = newNode;
  queueSize++;
}
```




```
template<class T>
LinkedQueue<T>& LinkedQueue<T>::pop()
{// 删除队首元素
  if (queueFront == NULL) throw  QueueEmpty();

     chainNode<T> * nextNode = queueFront->next;
     delete queueFront;     // 删除第一个节点
     queueFront = nextNode;
     queueSize--;
}
```

从extendedChain 派生

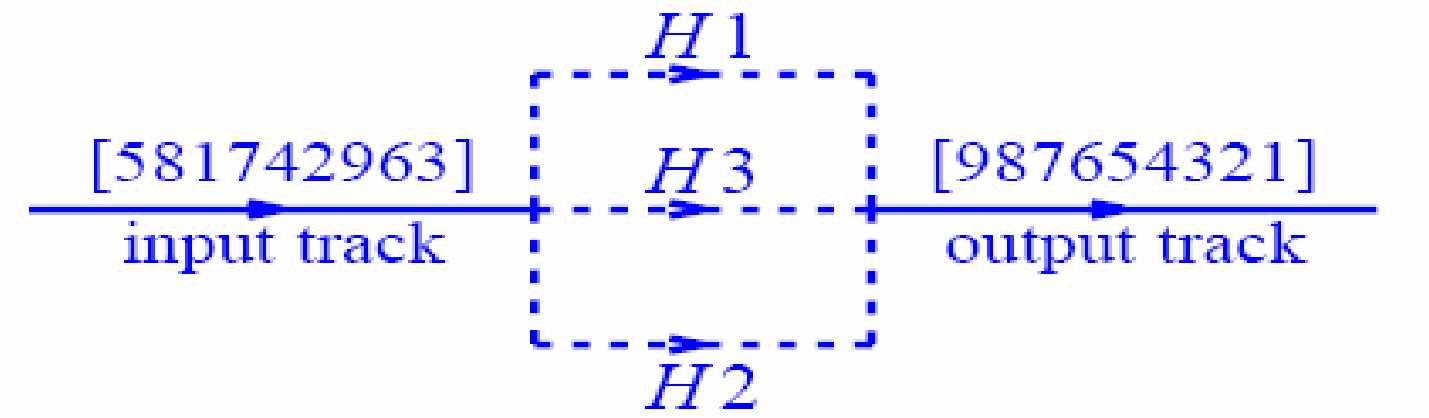


- linkedQueue.empty() → extendedChain.empty()
- linkedQueue.front() → extendedChain.get(0)
- linkedQueue.back() → extendedChain.get(size()-1)
- linkedQueue.push(x) → extendedChain.push_back(x)
- linkedQueue.pop() → extendedChain.erase(0)

9.5 应用

- **9.5.1 列车车厢重排**
- 9.5.2 电路布线
- 9.5.3 图元识别
- 9.5.4 工厂仿真

9.5.1 列车车厢重排



- 缓冲铁轨位于入轨和出轨之间
- 禁止：
  - 将车厢从缓冲铁轨移动至入轨
  - 从出轨移动车厢至缓冲铁轨
- 铁轨Hk 为可直接将车厢从入轨移动到出轨的通道

车厢移动到缓冲铁轨的原则

- 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）。

列车车厢重排思想

int NowOut=1； // NowOut:下一次要输出的车厢号

for （int i=1;i<=n;i++）//从前至后依次检查的所有车厢

{1. 车厢 p[i] 从入轨上移出

2. If (p[i] == NowOut)// NowOut:下一次要输出的车厢号

①使用缓冲铁轨Hk把p[i]放到出轨上去; NowOut++;

② while (minH(当前缓冲铁轨中编号最小的车厢)== NowOut ）

{把minH放到出轨上去;

更新 minH,minQ（minH所在的缓冲铁轨）;

NowOut++;}

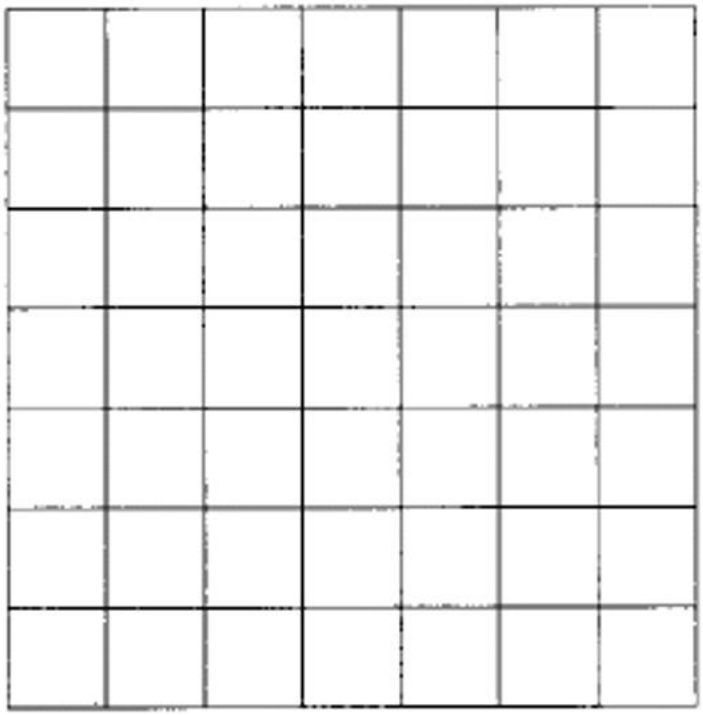
else 按照分配规则将车厢p[i]送入某个缓冲铁轨 }

- 读程序 9-6 9-7

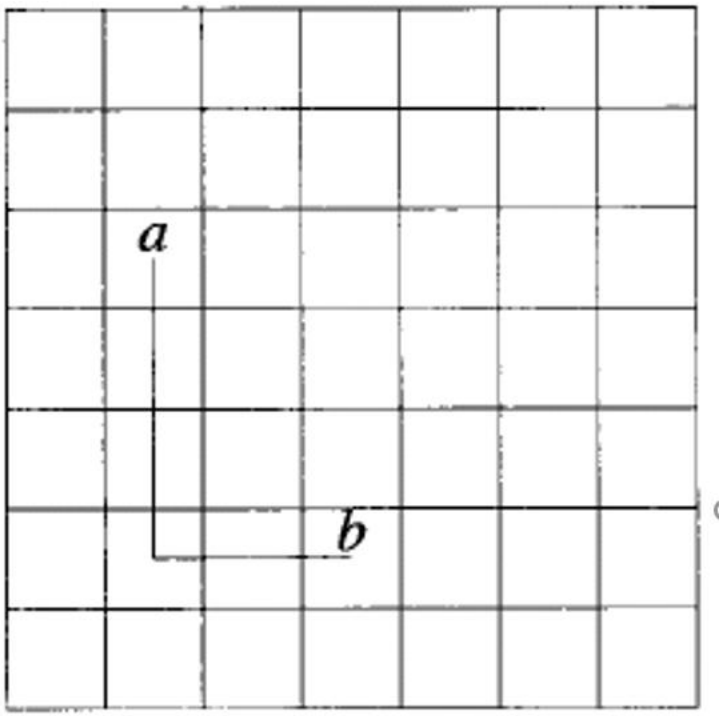
9.5.2迷宫最短路径问题扩展

- 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×m个方格，就像迷宫一样。
- 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。

电路布线



a)

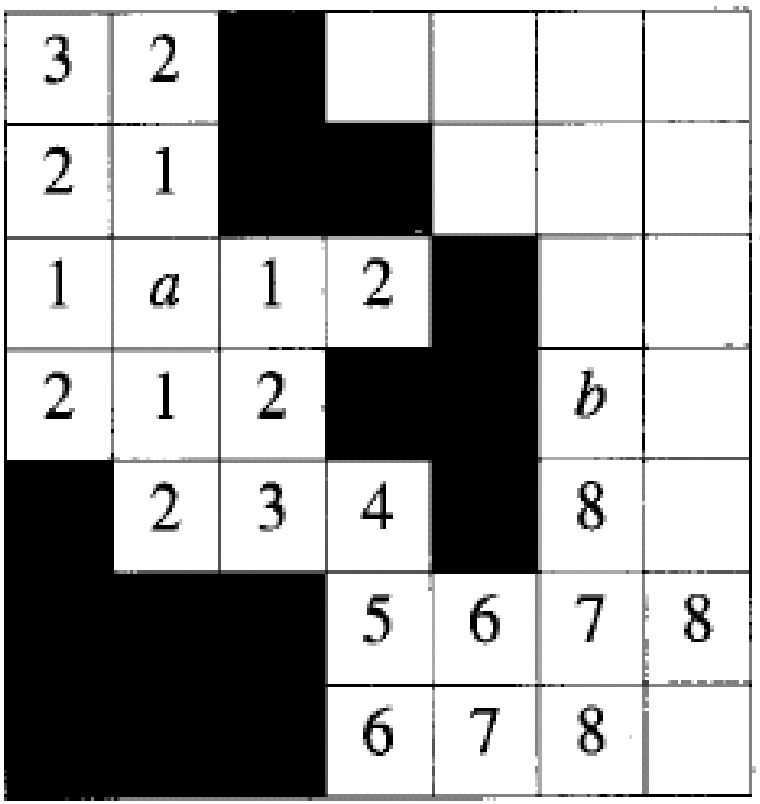


b)

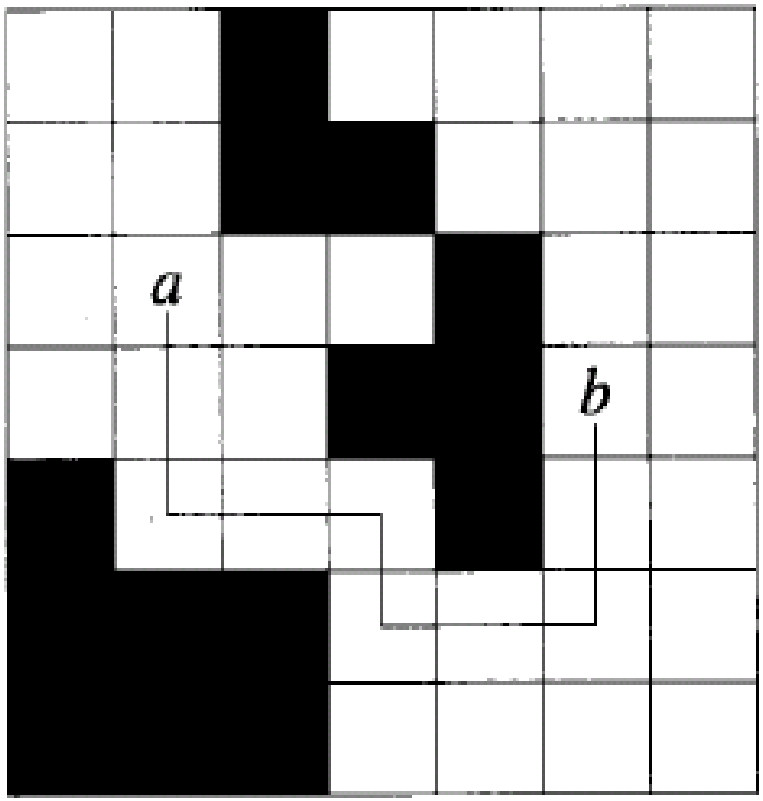
方案

- 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。

方案演示



a)



b)

a) 标识间距 b) 电线路径

输出方案

- 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。
  - 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。

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
}
```

寻找电路布线最短路径

```
// 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
```

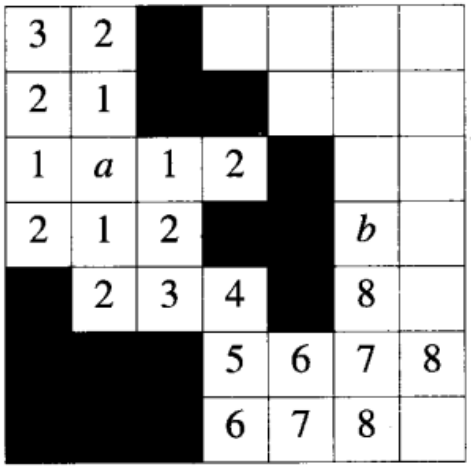
寻找电路布线最短路径

```
// 标记可到达的网格位置
ArraryQueue<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) {//新位置
        grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
        if((nbr.row==finish.row)&&(nbr.col==finish.col))
                break; // 完成
        q.push(nbr);} // if 结束
        } // for 结束
```

寻找电路布线最短路径

```
        //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
        here=q.front(); // 到下一位置
        q.pop();
  } while(true);
```

电路布线过程-队列状态



front → 3, 3 → 4, 2 → 3, 1 → 2, 2 ← rear

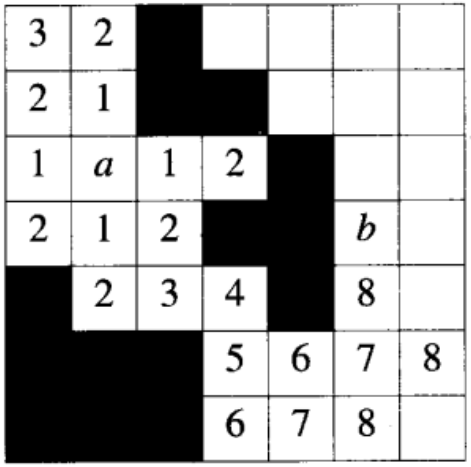
front → 4, 2 → 3, 1 → 2, 2 → 3, 4 → 4, 3 ← rear

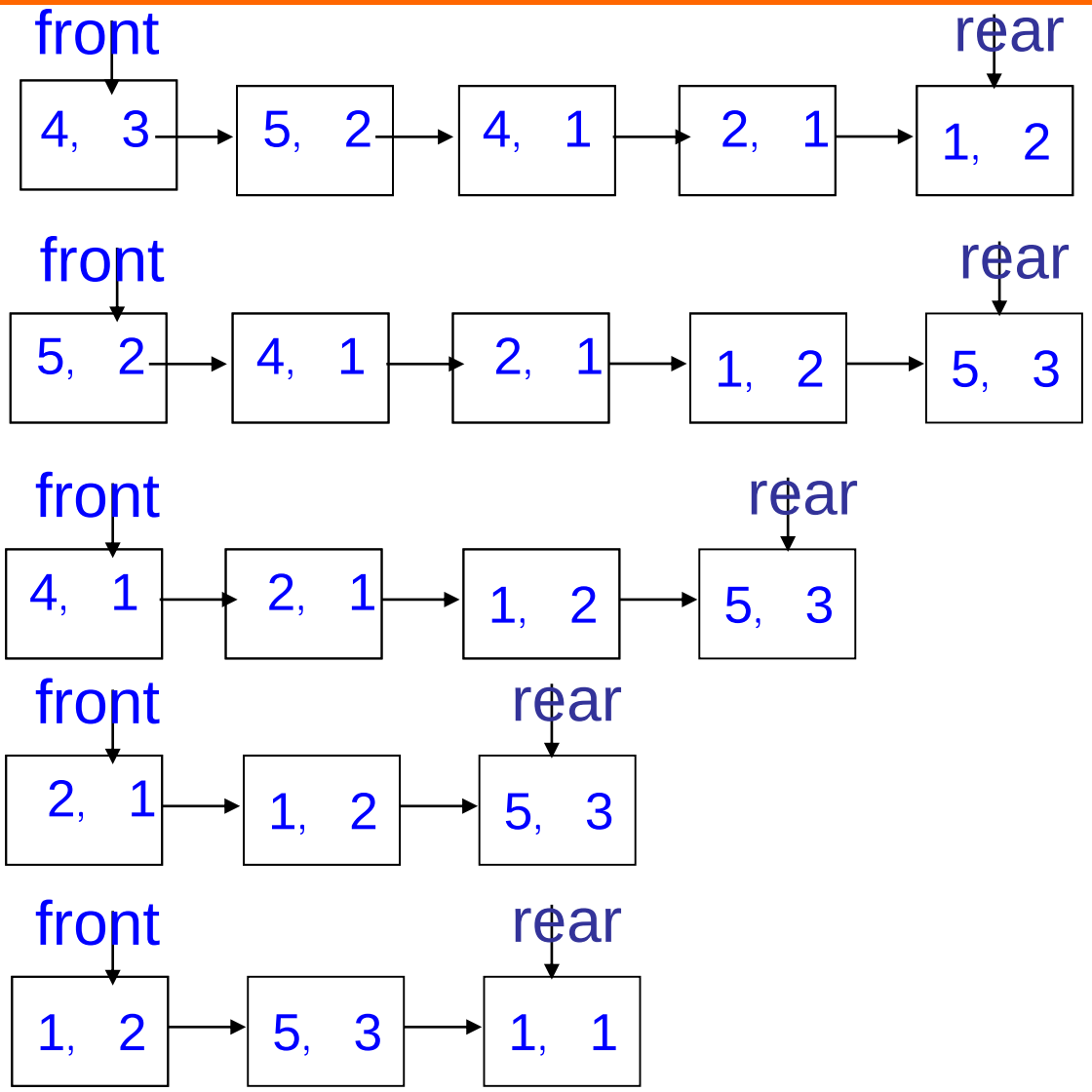
front → 3, 1 → 2, 2 → 3, 4 → 4, 3 → 5, 2 → 4, 1 ← rear

front → 2, 2 → 3, 4 → 4, 3 → 5, 2 → 4, 1 → 2, 1 ← rear

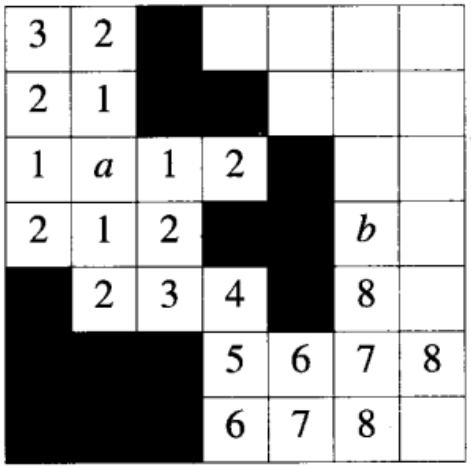
front → 3, 4 → 4, 3 → 5, 2 → 4, 1 → 2, 1 → 1, 2 ← rear

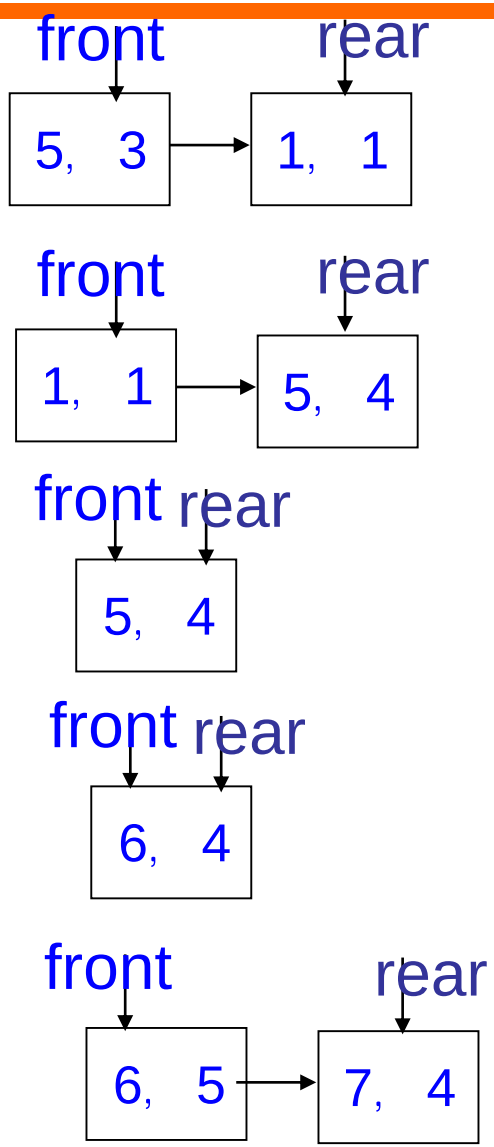
电路布线过程-队列状态


3 2
2 1
1 a 1 2
2 1 2 b
2 3 4 8
5 6 7 8
6 7 8

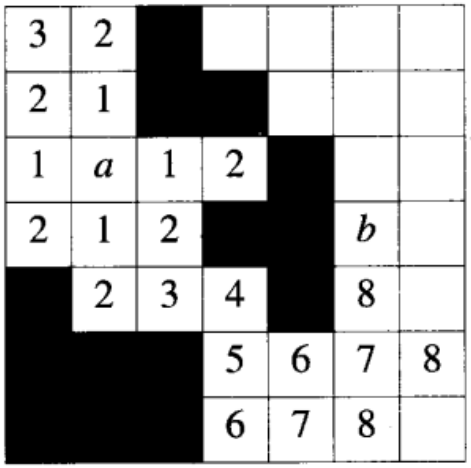

front
rear
4, 3
5, 2
4, 1
2, 1
1, 2
front
rear
5, 2
4, 1
2, 1
1, 2
5, 3
front
rear
4, 1
2, 1
1, 2
5, 3
front
rear
2, 1
1, 2
5, 3
front
rear
1, 2
5, 3
1, 1

电路布线过程-队列状态


3 2
2 1
1 a 1 2
2 1 2 b
2 3 4 8
5 6 7 8
6 7 8


front
rear
5, 3
1, 1
front
rear
1, 1
5, 4
front
rear
5, 4
front
rear
6, 4
front
rear
6, 5
7, 4

电路布线过程-队列状态



front → rear

7, 4 → 6, 6 → 7, 5

front → rear

6, 6 → 7, 5

front → rear

7, 5 → 6, 7 → 7, 6 → 5, 6

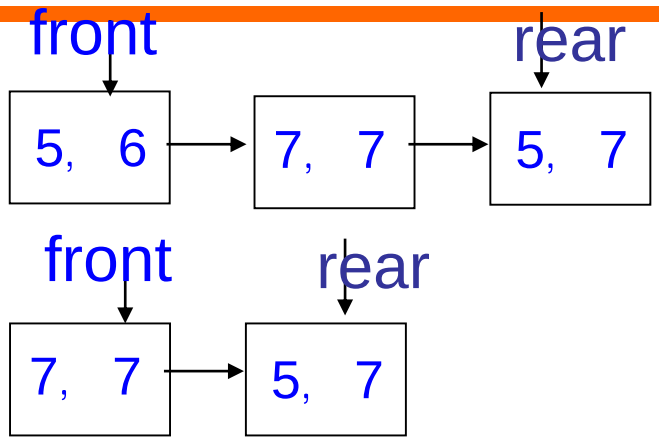
front → rear

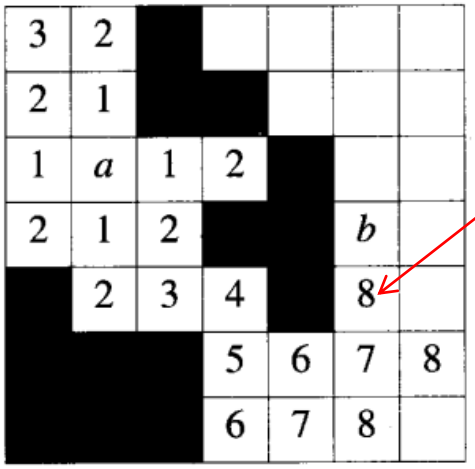
6, 7 → 7, 6 → 5, 6

front → rear

7, 6 → 5, 6 → 7, 7 → 5, 7

电路布线过程-队列状态





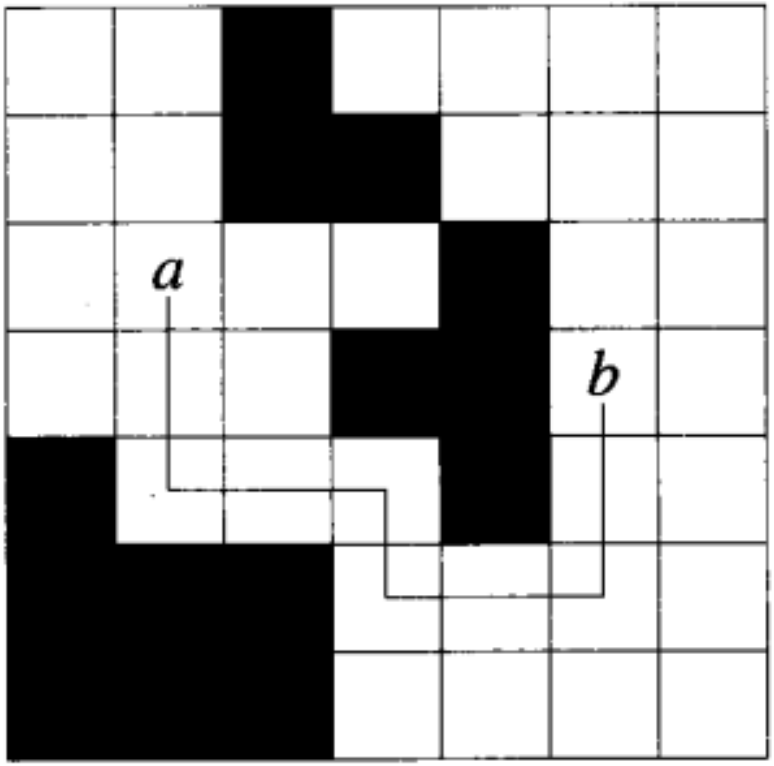

```
(nbr.row==finish.row)&&(nbr.col==finish.col)
```

寻找电路布线最短路径

1、从b开始，首先移动到一个比b的编号小的相邻位置上。可从b移动到（ 5 ， 6 ）

2、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。

（ 5 ， 6 ），
（ 6 ， 6 ），
（ 6 ， 4 ），
（ 5 ， 4 ），
…
（3 ， 2 ）。



寻找电路布线最短路径

```
// 构造路径
PathLenth = grid[finish.row][finish.col] - 2;
path = new Position [PathLenth];
here = finish; // 回溯至finish
for (int j = PathLenth-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;  }
```

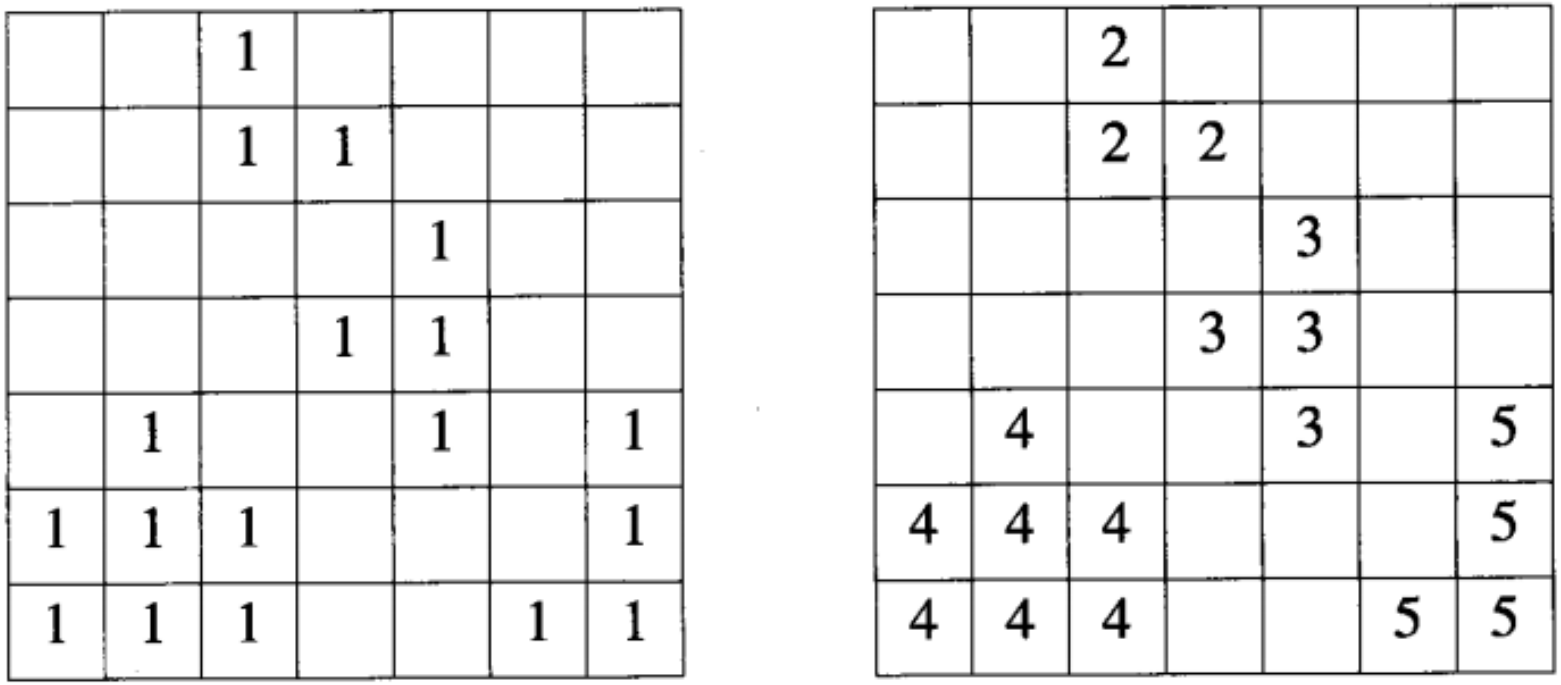
电路布线复杂度分析

- 网格编号过程需耗时 O (m^2)
- 重构路径的过程需耗时 Q（PathLenth）

9.5.3识别图元

- 数字化图像是一个m×m 的像素矩阵。
- 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。

识别图元



		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

图6-13　识别图元

a) 7×7 图像　b) 标记图元

9.5.4工厂仿真

- 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。

工序属性

- 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。

机器状态

- 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。
- 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。

任务队列

- 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。

目标

- 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。

工厂仿真实现

- 在对工厂进行仿真时，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，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。

示例

- 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

四项任务的特征

- 每道工序用形如(machine，time)的值对来描述。
- 各项任务的长度分别为7，6，8和4。

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)

仿真

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)

结果

- 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。

作业

- 假设一数列的输入顺序为1234,若采用队列结构调整数列输出顺序，设计算法求出所有可能的输出序列(合法序列)。