CHƯƠNG 7

HÀNG ĐỢI

Cũng như ngăn xếp, hàng đợi là CTDL tuyến tính. Hàng đợi là một danh sách các đối tượng, một đầu của danh sách được xem là đầu hàng đợi, còn đầu kia của danh sách được xem là đuôi hàng đợi. Với hàng đợi, chúng ta chỉ có thể xen một đối tượng mới vào đuôi hàng và loại đối tượng ở đầu hàng ra khỏi hàng. Trong chương này chúng ta sẽ nghiên cứu các phương pháp cài đặt hàng đợi và trình bày một số ứng dụng của hàng đợi.

1.1 KIỂU DỮ LIỆU TRỪU TƯỢNG HÀNG ĐỢI

Trong mục này chúng ta sẽ đặc tả KDLTT hàng đợi. Chúng ta có thể xem một hàng người đứng xếp hàng chờ được phục vụ (chẳng hạn, xếp hàng chờ mua vé tàu, xếp hàng chờ giao dịch ở ngân hàng, …) là một hàng đợi, bởi vì người ra khỏi hàng và được phục vụ là người đứng ở đầu hàng, còn người mới đến sẽ đứng vào đuôi hàng.

Hàng đợi là một danh sách các đối tượng dữ liệu, một trong hai đầu danh sách được xem là đầu hàng, còn đầu kia là đuôi hàng. Chẳng hạn, hàng đợi có thể là danh sách các ký tự (a, b, c, d), trong đó a đứng ở đầu hàng, còn d đứng ở đuôi hàng. Chúng ta có thể thực hiện các phép toán sau đây trên hàng đợi, trong các phép toán đó Q là một hàng đợi, còn x là một đối tượng cùng kiểu với các đối tượng trong hàng Q.

1. Empty(Q). Hàm trả về true nếu hàng Q rỗng và false nếu không.
2. Enqueue(x, Q). Thêm đối tượng x vào đuôi hàng Q.
3. Dequeue(Q). Loại đối tượng đứng ở đầu hàng Q.
4. GetHead(Q). Hàm trả về đối tượng đứng ở đầu hàng Q, còn hàng Q thì không thay đổi.

Ví dụ. Nếu Q = (a, b, c, d) và a ở đầu hàng, d ở đuôi hàng, thì khi thực hiện phép toán Enqueue(e, Q) ta nhận được Q = (a, b, c, d, e), với e đứng ở đuôi hàng, nếu sau đó thực hiện phép toán Dequeue(Q), ta sẽ có Q = (b, c, d, e) và b trở thành phần tử đứng ở đầu hàng.

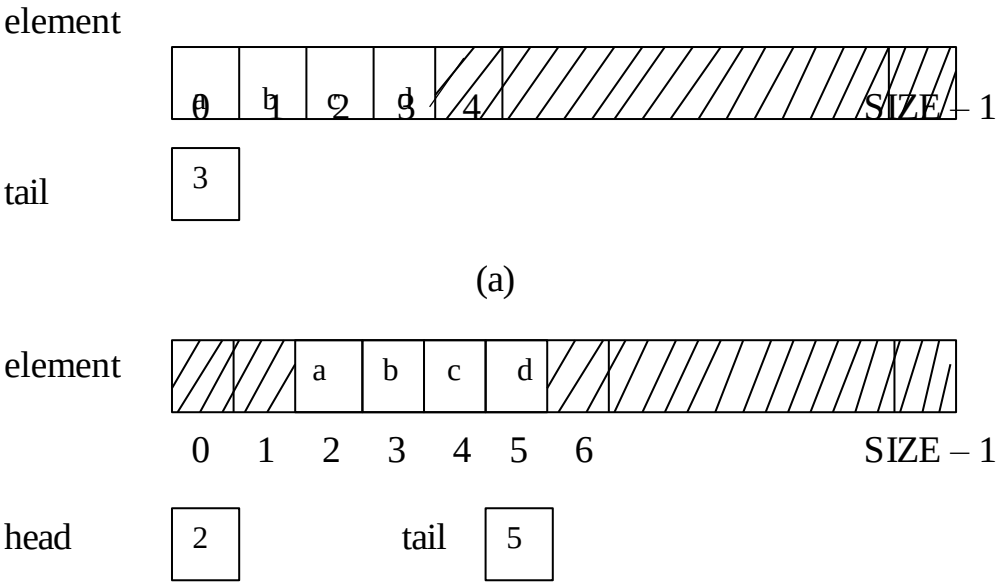
Với các phép toán Enqueue và Dequeue xác định như trên thì đối tượng vào hàng trước sẽ ra khỏi hàng trước. Vì lý do đó mà hàng đợi được gọi là cấu trúc dữ liệu FIFO (viết tắt của cụm từ First- In First- Out). Điều này đối lập với ngăn xếp, trong ngăn xếp đối tượng ra khỏi ngăn xếp là đối tượng sau cùng được đặt vào ngăn xếp.

Hàng đợi sẽ được sử dụng trong bất kỳ hoàn cảnh nào mà chúng ta cần xử lý các đối tượng theo trình tự FIFO. Cuối chương này chúng ta sẽ trình bày một ứng dụng của hàng đợi trong mô phỏng một hệ phục vụ. Nhưng

trước hết chúng ta cần nghiên cứu các phương pháp cài đặt hàng đợi. Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi mảng hoặc bởi DSLK.

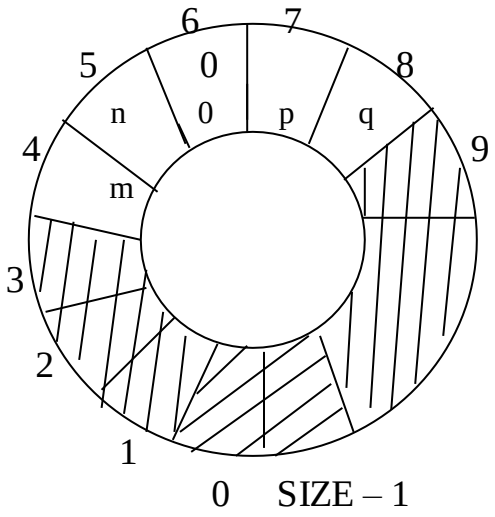
1.2 CÀI ĐẶT HÀNG ĐỢI BỞI MẢNG

Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi mảng. Song cài đặt hàng đợi bởi mảng sẽ phức tạp hơn ngăn xếp. Nhớ lại rằng, khi cài đặt danh sách (hoặc ngăn xếp) bởi mảng thì các phần tử của danh sách (hoặc ngăn xếp) sẽ được lưu trong đoạn đầu của mảng, còn đoạn sau của mảng là không gian chưa sử dụng đến. Chúng ta có thể làm như thế với hàng đợi được không? Câu trả lời là có, nhưng không hiệu quả. Giả sử chúng ta sử dụng mảng element để lưu các phần tử của hàng đợi, các phần tử của hàng đợi được lưu trong các thành phần mảng element[0], element[1], … , element[k] như trong hình 7.1a. Với cách này, phần tử ở đầu hàng luôn luôn được lưu trong thành phần mảng element[0], còn phần tử ở đuôi hàng được lưu trong element[k], và do đó ngoài mảng element ta chỉ cần một biến tail ghi lại chỉ số k. Để thêm phần tử mới vào đuôi hàng, ta chỉ cần tăng chỉ số tail lên 1 và đặt phần tử mới vào thành phần mảng element[tail]. Song nếu muốn loại phần tử ở đầu hàng, chúng ta cần chuyển lên trên một vị trí tất cả các phần tử được lưu trong element[1],…, element[tail] và giảm chỉ số tail đi 1, và như vậy tiêu tốn nhiều thời gian.



(a)

(b)



head 4 tail 8

(c)

Hình 7.1. Các phương pháp cài đặt hàng đợi bởi mảng

Để khắc phục nhược điểm của phương pháp trên, chúng ta lưu các phần tử của hàng đợi trong một đoạn con của mảng từ chỉ số đầu head tới chỉ số đuôi tail, tức là các phần tử của hàng đợi được lưu trong các thành phần mảng element[head], element[head + 1], … , element[tail], như trong hình 7.1b. Với cách cài đặt này, ngoài mảng element, chúng ta cần sử dụng hai biến: biến chỉ số đầu head và biến chỉ số đuôi tail. Để loại phần tử ở đầu hàng chúng ta chỉ cần tăng chỉ số head lên 1. Chúng ta có nhận xét răng, khi thêm phần tử mới vào đuôi hàng thì chỉ số tail tăng, khi loại phần tử ở đầu hàng thì chỉ số head tăng, tới một thời điểm nào đó hàng đợi sẽ chiếm đoạn cuối của mảng, tức là biến tail nhận giá trị SIZE – 1 (SIZE là cỡ của mảng). Lúc đó ta không thể thêm phần tử mới vào đuôi hàng, mặc dầu không gian chưa sử dụng trong mảng, đoạn đầu của mảng từ chỉ số 0 tới head – 1, có thể còn rất lớn! Để giải quyết vấn đề này, chúng ta "cuộn" mảng thành vòng tròn, như được minh hoạ trong hình 7.1c. Trong **mảng vòng tròn**, chúng ta xem thành phần tiếp theo thành phần element[SIZE - 1] là thành phần element[0]. Với mảng vòng tròn, khi thêm phần tử mới vào đuôi hàng, nếu chỉ số tail có giá trị là SIZE – 1, ta đặt tail = 0 và lưu phần tử mới trong thành phần mảng element[0]. Tương tự khi phần tử ở đầu hàng được lưu

trong element[SIZE - 1], thì để loại nó, ta chỉ cần đặt head = 0. Do đó, nếu cài đặt hàng đợi bởi mảng vòng tròn, thì phép toán thêm phần tử mới vào đuôi hàng không thực hiện được chỉ khi mảng thực sự đầy, tức là khi trong mảng không còn thành phần nào chưa sử dụng.

Sau đây chúng ta sẽ nghiên cứu sự cài đặt hàng đợi bởi mảng vòng tròn. KDLTT hàng đợi sẽ được cài đặt bởi lớp Queue, đây là lớp khuôn phụ thuộc tham biến kiểu Item, trong đó Item là kiểu của các phần tử trong hàng đợi. Lớp Queue chứa các biến thành phần nào? Các phần tử của hàng đợi được lưu trong mảng vòng tròn được cấp phát động, do đó cần có biến con trỏ element trỏ tới thành phần đầu tiên của mảng đó, biến size lưu cỡ của mảng, biến chỉ số đầu head và biến chỉ số đuôi tail. Ngoài các biến trên, chúng ta thêm vào một biến length để lưu độ dài của hàng đợi (tức là số phần tử trong hàng đợi). Lớp Queue được định nghĩa như trong hình 7.2.

```
template <class Item>
class     Queue
{
  public :
      Queue (int m = 1);
      // Hàm kiến tạo hàng đợi rỗng với dung lượng là m,
      // m nguyên dương (tức là cỡ mảng động là m).
      Queue (const Queue & Q) ;
      // Hàm kiến tạo copy.
      ~ Queue( ) // Hàm huỷ.
      { delete [ ] element ; }
      Queue & operator = (const Queue & Q); // Toán tử gán.
      // Các hàm thực hiện các phép toán hàng đợi :
      bool    Empty( ) const ;
      // Kiểm tra hàng có rỗng không.
      // Postcondition: hàm trả về true nếu hàng rỗng và false nếu không rỗng
      { return length = = 0 ; }
      void    Enqueue (const Item & x)
      // Thêm phần tử mới x vào đuôi hàng.
      // Postcondition: x là phần tử ở đuôi hàng.
      Item &   Dequeue( );
      // Loại phần tử ở đầu hàng.
      // Precondition: hàng không rỗng.
      // Postcondition: phần tử ở đầu hàng bị loại khỏi hàng và hàm trả về
      // phần tử đó.
      Item &   GetHead( ) const ;
      // Precondition: hàng không rỗng.
      // Postcondition: hàm trả về phần tử ở đầu hàng, nhưng hàng vẫn
```

```
    // giữ nguyên.
  private:
    Item *   element ;
    int    size ;
    int    head ;
    int    tail ;
    int    length ;
} ;
```

Hình 7.2. Lớp hàng đợi được cài đặt bởi mảng.

Bây giờ chúng ta nghiên cứu sự cài đặt các hàm thành phần của lớp Queue.

Hàm kiến tạo hàng rỗng với sức chứa là m. Chúng ta cần cấp phát một mảng động element có cỡ là m. Vì hàng rỗng, nên giá trị của biến length là 0. Các chỉ số đầu head và chỉ số đuôi tail cần có giá trị nào? Nhớ lại rằng khi cần thêm phần tử x vào đuôi hàng, ta tăng biến tail lên 1 và đặt element[tail] = x. Để cho thao tác này làm việc trong mọi hoàn cảnh, kể cả trường hợp hàng rỗng, chúng ta đặt head = 0 và tail = -1 khi kiến tạo hàng rỗng. Do đó, hàm kiến tạo được cài đặt như sau:

```
template <class Item>
Queue<Item> :: Queue (int m)
{
    element = new Item[m] ;
    size = m ;
    head = 0 ;
    tail = -1 ;
    length = 0 ;
}
```

Hàm kiến tạo copy. Hàm này thực hiện nhiệm vụ kiến tạo ra một hàng đợi mới là bản sao của một hàng đợi đã có Q. Do đó chúng ta cần cấp phát một mảng động mới có cỡ bằng cỡ của mảng trong Q và tiến hành sao chép các dữ liệu từ đối tượng Q sang đối tượng mới.

Hàm kiến tạo copy được viết như sau:

```
template <class Item>
Queue<Item> :: Queue (const Queue<Item> & Q)
{
    element = new Item[Q.size] ;
    size = Q.size ;
    head = Q.head ;
```

```
        tail = Q.tail ;
        length = Q.length ;
        for (int i = 0 ; i < length ; i + + )
            element [(head + i) % size] = Q.element [(head + i) % size] ;
    }
```

Trong hàm trên, vòng lặp for thực hiện sao chép mảng trong Q sang mảng mới kể từ chỉ số head tới chỉ số tail, tức là sao chép length thành phần mảng kể từ chỉ số head. Nhưng cần lưu ý rằng, trong mảng vòng tròn, thành phần tiếp theo thành phần với chỉ số k, trong các trường hợp k ≠ size – 1, là thành phần với chỉ số k + 1. Song nếu k = size – 1 thì thành phần tiếp theo có chỉ số là 0. Vì vậy, trong mọi trường hợp thành phần tiếp theo thành phần với chỉ số k là thành phần với chỉ số (k + 1) % size.

Toán tử gán được cài đặt tương tự như hàm kiến tạo copy:

```
template <class Item>
Queue<Item> & Queue<Item> :: operator = (const Queue<Item> & Q)
{
   if ( this != Q )
     {
       delete [  ] element ;
       element = new Item[Q.size] ;
       size = Q.size ;
       head = Q.head ;
       tail = Q.tail ;
       length = Q.length ;
       for (int i = 0 ; i < length ; i + + )
           element [(head + i) % size] = Q.element[(head + i) % size] ;
     }
   return    * this ;
}
```

Hàm xen phần tử mới vào đuôi hàng được cài đặt bằng cách đặt phần tử mới vào mảng vòng tròn tại thành phần đứng ngay sau thành phần element[tail], nếu mảng chưa đầy. Nếu mảng đầy thì chúng ta cấp phát một mảng mới với cỡ lớn hơn (chẳng hạn, với cỡ gấp đôi cỡ mảng cũ), và sao chép dữ liệu từ mảng cũ sang mảng mới, rồi huỷ mảng cũ, sau đó đặt phần tử mới vào mảng mới.

```
    template <class Item>
    void    Queue<Item> :: Enqueue (const Item & x)
    {
      if (length <size) // mảng chưa đầy.
```

```
    {
      tail = (tail + 1) % size ;
      element[tail] = x ;
    }
  else  // mảng đầy
    {
      Item * array = new Item[2 * size] ;
      for (int i = 0 ; i < size ; i + +)
      array[i] = element[(head + i) % size] ;
      array[size] = x ;
      delete [  ] element ;
      element = array ;
      head = 0 ;
      tail = size ;
      size = 2 * size ;
    }
  length + + ;
}
```

Các hàm loại phần tử ở đầu hàng và truy cập phần tử ở đầu hàng được cài đặt rất đơn giản như sau:

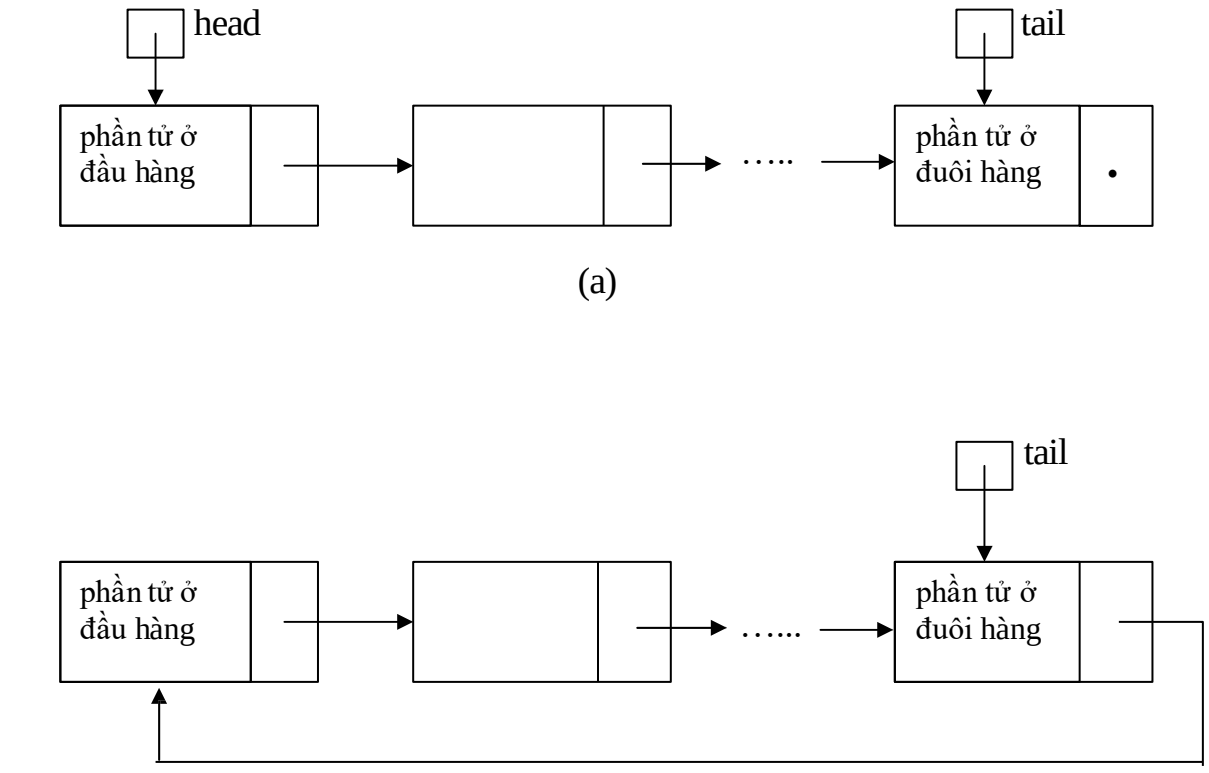
```
template <class Item>
Item &    Queue<Item> :: Dequeue(  )
{
   assert (length > 0) ; // Kiểm tra hàng không rỗng.
   Item   data = element[head] ;
   if (length = = 1) // Hàng có một phần tử.
     { head = 0 ; tail = -1 ; }
   else    head = (head + 1) % size ;
   length - - ;
   return   data ;
}

template  <class Item>
Item &   Queue<Item> :: GetHead( )
{
  assert (length > 0) ;
  return   element[head] ;
}
```

Dễ dàng thấy rằng, khi cài đặt hàng đợi bởi mảng vòng tròn thì các phép toán hàng đợi: xen phần tử mới vào đuôi hàng, loại phần tử ở đầu hàng

và truy cập phần tử ở đầu hàng chỉ đòi hỏi thời gian O(1). Chỉ trừ trường hợp mảng đầy, nếu mảng đầy thì để xen phần tử mới vào đuôi hàng chúng ta mất thời gian sao chép dữ liệu từ mảng cũ sang mảng mới với cỡ lớn hơn, do đó thời gian thực hiện phép toán thêm vào đuôi hàng trong trường hợp này là O(n), n là số phần tử trong hàng. Tuy nhiên trong các ứng dụng, nếu ta đánh giá được số tối đa các phần tử ở trong hàng và lựa chọn số đó làm dung lượng m của hàng đợi khi khởi tạo hàng đợi, thì có thể đảm bảo rằng tất cả các phép toán hàng đợi chỉ cần thời gian O(1).

1.3 CÀI ĐẶT HÀNG ĐỢI BỞI DSLK

Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi DSLK. Với ngăn xếp, chúng ta chỉ cần truy cập tới phần tử ở đỉnh ngăn xếp, nên chỉ cần một con trỏ ngoài top trỏ tới đầu DSLK (xem hình 6.3). Nhưng với hàng đợi, chúng ta cần phải truy cập tới cả phần tử ở đầu hàng và phần tử ở đuôi hàng, vì vậy chúng ta cần sử dụng hai con trỏ ngoài: con trỏ head trỏ tới thành phần đầu DSLK, tại đó lưu phần tử ở đầu hàng, và con trỏ tail trỏ tới thành phần cuối cùng của DSLK, tại đó lưu phần tử ở đuôi hàng, như trong hình 7.3a. Một cách tiếp cận khác, chúng ta có thể cài đặt hàng đợi bởi DSLK vòng tròn với một con trỏ ngoài tail như trong hình 7.3b.



Hình 7.3. (a) Cài đặt hàng đợi bởi DSLK với hai con trỏ ngoài.
(b) Cài đặt hàng đợi bởi DSLK vòng tròn.

Trong mục 5.4, chúng ta đã nghiên cứu cách cài đặt danh sách bởi DSLK, ở đó KDLTT danh sách đã được cài đặt bởi lớp LList. Các phép toán hàng đợi chỉ là các trường hợp riêng của các phép toán danh sách: xen phần tử mới vào đuôi hàng có nghĩa là xen nó vào đuôi danh sách, còn loại phần tử ở đầu hàng là loại phần tử ở vị trí đầu tiên trong danh sách. Do đó chúng ta có thể sử dụng lớp LList (xem hình 5.12) để cài đặt lớp hàng đợi Queue. Chúng ta có thể xây dựng lớp Queue bằng cách sử dụng lớp LList làm lớp cơ sở với dạng thừa kế private, hoặc cũng có thể xây dựng lớp Queue là lớp chỉ chứa một thành phần dữ liệu là đối tượng của lớp LList. Độc giả nên cài đặt lớp Queue theo các phương án trên, xem như bài tập. Các phép toán hàng đợi là rất đơn giản, nên chúng ta sẽ cài đặt lớp Queue trực tiếp, không thông qua lớp LList.

Sau đây chúng ta sẽ cài đặt hàng đợi bởi DSLK vòng tròn với con trỏ ngoài tail (hình 7.3b). KDLTT hàng đợi được cài đặt bởi lớp khuôn phụ thuộc tham biến kiểu Item (Item là kiểu của các phần tử trong hàng đợi). Lớp Queue này được định nghĩa trong hình 7.4. Lớp này chứa các hàm thành phần được khai báo hoàn toàn giống như lớp trong hình 7.2, chỉ trừ hàm kiến tạo mặc định làm nhiệm vụ khởi tạo hàng rỗng. Lớp chỉ chứa một thành phần dữ liệu là con trỏ tail.

```
template <class Item>
class    Queue
{
  public :
     Queue( ) // Hàm kiến tạo hàng đợi rỗng.
     { tail = NULL ; }
     Queue (const Queue & Q) ;
     ~ Queue( ) ;
     Queue &   operator = (const Queue & Q) ;
     bool    Empty( ) const
     {return tail = = NULL ; }
     void    Enqueue (const Item & x) ;
     Item &   Dequeue( )
     Item &   GetHead( ) const ;
  private :
     struct    Node
     {
         Item   data ;
```

```
        Node *   next ;
        Node (const Item & x)
        {data = x; next = NULL ; }
    }
    Node *   tail ;
    void    MakeEmpty( ) ; // Hàm huỷ DSLK vòng tròn tail.
} ;
```

Hình 7.4. Lớp hàng đợi được cài đặt bởi DSLK vòng tròn.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp Queue trong hình 7.4. Chúng ta đã đưa vào lớp Queue hàm MakeEmpty, nhiệm vụ của nó là thu hồi vùng nhớ đã cấp phát cho các thành phần của DSLK tail làm cho DSLK này trở thành rỗng. Hàm MakeEmpty là hàm thành phần private, nó được sử dụng để cài đặt hàm huỷ và toán tử gán. Hàm MakeEmpty được cài đặt như sau:

```
template <class Item>
void    Queue<Item> :: MakeEmpty( )
{
  while (tail ! = NULL)
    {
        Node*  Ptr = tail →next ;
        if  (Ptr = = tail) // DSLK chỉ có một thành phần.
                tail = NULL ;
        else
                tail → next = Ptr → next ;
        delete   Ptr ;
     }
}
```

Hàm huỷ được cài đặt bằng cách gọi hàm MakeEmpty:

```
template <class Item>
Queue <Item> :: ~ Queue( )
{
   MakeEmpty( ) ;
}
```

Hàm kiến tạo copy.

```
template <class Item>
```

```
Queue <Item> :: Queue (const Queue<Item> & Q)
{
  tail = NULL ;
  * this = Q ;
}
```

Toán tử gán.

```
template <class Item>
Queue<Item> & Queue<Item>:: operator =(const Queue <Item> & Q)
{
  if (this ! = & Q)
  {
      MakeEmpty( ) ;
      if (Q.Empty( ))
      return *this ;
      Node * Ptr = Q.tail → next ;
      do {
            Enqueue (Ptr → data) ;
            Ptr = Ptr → next ;
          }
      while (Ptr ! = Q.tail → next) ;
  }
  return    *this ;
}
```

Hàm thêm phần tử mới vào đuôi hàng:

```
template <class Item>
void    Queue<Item> :: Enqueue (const Item & x)
{
  if (Empty( ))
       {
          tail = new Node(x) ;
          tail → next = tail ;
        }
  else {
          Node *  Ptr = tail → next ;
          tail = tail →next = new Node(x) ;
          tail → next = Ptr ;
        }
}
```

Hàm loại phần tử ở đầu hàng:

```
template <class Item>
Item &   Queue<Item> :: Dequeue( )
{
   Item   headElement = GetHead( ) ;
   Node *   Ptr = tail → next ;
   if (Ptr ! = tail)
          tail → next = Ptr → next ;
   else    tail = NULL ;
   delete   Ptr ;
   return   headElement ;
}
```

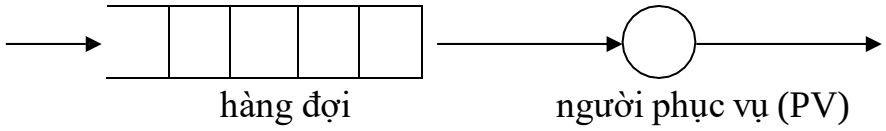
Hàm tìm phần tử ở đầu hàng:

```
template <class Item>
Item &   Queue<Item> :: GetHead( )
{
   assert (tail ! = NULL) ;
   return    tail → next → data ;
}
```

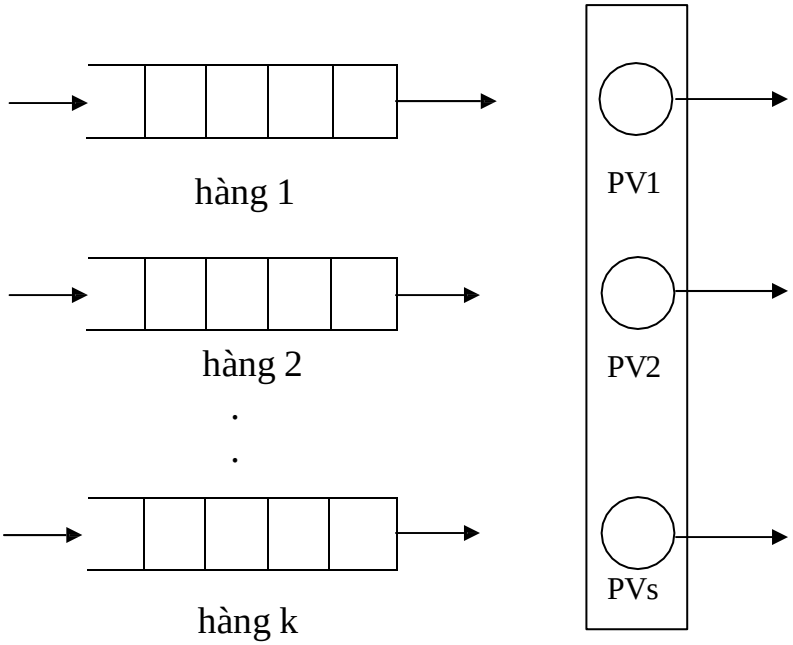
1.4 MÔ PHỎNG HỆ SẮP HÀNG

Mô phỏng (simulation) là một trong các lĩnh vực áp dụng quan trọng của máy tính. Mục đích của một chương trình mô phỏng là mô hình hoá sự hoạt động của một hệ hiện thực (hệ tồn tại trong tự nhiên hoặc hệ do con người sáng tạo ra) nhằm phân tích đánh giá hiệu năng của hệ hoặc đưa ra các tiên đoán để có thể cải tiến (đối với các hệ do con người làm ra) hoặc có thể đưa ra các biện pháp phòng ngừa, chế ngự (đối với các hệ tồn tại trong tự nhiên). Chúng ta có thể quan niệm một hệ hiện thực bao gồm các thực thể (các thành phần) phụ thuộc lẫn nhau, chúng hoạt động và tương tác với nhau để thực hiện một nhiệm vụ nào đó. Vì vậy, lập chương trình định hướng đối tượng là cách tiếp cận thích hợp nhất để xây dựng các chương trình mô phỏng. Chúng ta có thể biểu diễn mỗi thành phần trong hệ bởi một lớp đối tượng chứa các biến mô tả trạng thái của thực thể, sự tương tác giữa các thành phần của hệ được mô phỏng bởi sự truyền thông báo giữa các đối tượng. Dưới đây chúng ta sẽ xét một ví dụ: xây dựng chương trình mô phỏng một hệ sắp hàng (hệ phục vụ), tức là hệ với các hàng đợi được phục vụ theo nguyên tắc ai đến trước người đó được phục vụ trước, chẳng hạn, hệ các quầy giao dịch ở ngân hàng, hệ các cửa bán vé tàu ở nhà ga.

Trường hợp đơn giản nhất, hệ sắp hàng chỉ có một người phục vụ và một hàng đợi, như được chỉ ra trong hình 7.5a. Tổng quát hơn, một hệ sắp hàng có thể gồm k hàng đợi và s người phục vụ, như trong 7.5b.



(a)



(b)

Hình 7.5 (a) Hệ sắp hàng đơn giản
(b) Hệ sắp hàng với k hàng đợi, s người phục vụ.

Sau đây chúng ta xét trường hợp đơn giản nhất: hệ chỉ có một hàng đợi các khách hàng và một người phục vụ. Chúng ta sẽ mô phỏng sự hoạt động của hệ này trong khoảng thời gian T (tính bằng phút, chẳng hạn), kể từ thời điểm ban đầu t = 0. Trong khoảng thời gian 1 phút có thể có khách hàng đến hoặc không, chúng ta không thể biết trước được. Song giả thiết rằng, chúng ta được cho biết xác suất p: xác suất trong khoảng thời gian 1 phút có khách hàng đến. Trong trường hợp tổng quát, thời gian phục vụ dành cho mỗi khách hàng chúng ta cũng không biết trước được, và thời gian phục vụ các khách hàng khác nhau cũng khác nhau, chẳng hạn thời gian phục vụ các khách hàng ở ngân hàng. Để cho đơn giản, chúng ta giả thiết rằng thời gian

phục vụ mỗi khách hàng là như nhau và là s (phút) (chẳng hạn như một trạm rửa xe tự động, nó rửa sạch mỗi xe hết 5 phút). Như vậy, chúng ta đã biết các thông tin sau đây:

1. Xác suất trong thời gian 1 phút có khách đến là p ($0 \le p \le 1$), (chúng ta giả thiết rằng trong 1 phút chỉ có nhiều nhất một khách hàng đến)
2. Thời gian phục vụ mỗi khách hàng là s phút. Chương trình mô phỏng cần thực hiện các nhiệm vụ sau:
   - Đánh giá số khách hàng được phục vụ.
   - Đánh giá thời gian trung bình mà mỗi khách hàng phải chờ đợi.

Để xây dựng được chương trình mô phỏng hệ sắp hàng có một hàng đợi và một người phục vụ với các giả thiết đã nêu trên, chúng ta cần tạo ra hai lớp: lớp các khách hàng, và lớp người phục vụ.

Lớp các khách hàng đương nhiên là lớp hàng đợi Queue mà chúng ta đã cài đặt (lớp trong hình 7.2, hoặc 7.4). Vấn đề mà chúng ta cần giải quyết ở đây là biểu diễn các khách hàng như thế nào? Chúng ta không cần quan tâm tới các thông tin về khách hàng như tên, tuổi, giới tính, … ; với các nhiệm vụ của chương trình mô phỏng đã nêu ở trên, chúng ta chỉ cần quan tâm tới thời điểm mà khách hàng đến và vào hàng đợi. Vì vậy, chúng ta sẽ biểu diễn mỗi khách hàng bởi thời điểm t mà khách hàng vào hàng đợi, t là số nguyên dương. Và do đó, trong chương trình, chúng ta chỉ cần khai báo:

```
Queue<int>  customer ;
```

Bây giờ chúng ta thiết kế lớp người phục vụ ServerClass. Tại mỗi thời điểm, người phục vụ có thể đang bận phục vụ một khách hàng, hoặc không. Vì vậy để chỉ trạng thái (bận hay không bận) của người phục vụ, chúng ta đưa vào lớp một biến time-left, biến này ghi lại thời gian còn lại mà người phục vụ cần làm việc với khách hàng đang được phục vụ. Khi bắt đầu phục vụ một khách hàng biến time-left được gán giá trị là thời gian phục vụ. Người phục vụ bận có nghĩa là time-left > 0. Tại mỗi thời điểm t khi người phục vụ tiếp tục làm việc với một khách hàng, biến time-left sẽ giảm đi 1. Khi mà time-left nhận giá trị 0 có nghĩa là người phục vụ rỗi, có thể phục vụ khách hàng tiếp theo. Lớp người phục vụ được khai báo như sau:

```
class      ServerClass
{
  public:
     ServerClass (int s) // Khởi tạo người phục vụ rỗi với thời gian
     // phục vụ mỗi khách hàng là s.
          { server-time = s; time-left = 0 ; }
     bool    Busy( )  const // Người phục vụ có bận không?
          { return   time-left > 0 ; }
     void     Start( ) // Bắt đầu phục vụ một khách hàng.
```

```
        { time-left = server-time ; }
    void    Continue( ) // Tiếp tục làm việc với khách hang.
        { if (Busy( ))   time-left - - ; }
  private :
    int   server-time ; // Thời gian phục vụ mỗi khách hàng.
    int   time-left ;
}
```

Tới đây chúng ta có thể thiết kế chương trình mô phỏng. Hàm mô phỏng chứa các biến đầu vào sau:

- Thời gian phục vụ mỗi khách hàng: s
- Xác suất khách hàng đến: p
- Thời gian mô phỏng: T

Các biến đầu ra:

- Số khách hàng được phục vụ: count
- Thời gian chờ đợi trung bình của khách hàng: wait-time.

Hàm mô phỏng sẽ mô tả sự hoạt động của hệ sắp hàng bởi một vòng lặp: tại mỗi thời điểm t (t = 1, 2, … , T), nếu có khách hàng đến thì đưa nó vào hàng đợi, nếu người phục vụ bận thì cho họ tiếp tục làm việc; rồi sau đó kiểm tra xem, nếu hàng đợi không rỗng và người phục vụ rỗi thì cho khách hàng ở đầu hàng ra khỏi hàng và người phục vụ bắt đầu phục vụ khách hàng đó. Hàm mô phỏng có nội dung như sau:

```
void     Queueing_System_Simulation
(int s, double p, int T, int & count, int & wait-time)
{
   Queue <int>  customer ; // Hàng đợi các khách hàng.
   ServerClass   server(s) ; // Người phục vụ.
   void   QueueEntry( int t) ;
   // Hàm làm nhiệm vụ đưa khách hàng đến (nếu có) tại thời điểm t
   // vào hàng đợi.
   int   t ; // Biến chỉ thời điểm, t = 1, 2, … , T
   int   sum = 0 ; // Tổng thời gian chờ đợi của khách hàng.
   count = 0 ;
   for (t = 1; t < = T ; t + +)
   {
      QueueEntry(t) ; // Đưa khách hàng (nếu có) vào hàng đợi.
      if (server.Busy( ))
                server.Continue( ) ;
      else  if ( ! customer.Empty( ))
                {
                   int   t1 = customer.Dequeue( );
                   server.Start( ) ;
```

```
                    sum + = t – t1 ;
                    count + + ;
                }
    }
    wait-time = sum / count ;
}
```

Bây giờ chúng ta cần cài đặt hàm QueueEntry. Hàm này có nhiệm vụ đưa ra quyết định tại thời điểm t có khách hàng đến hay không và nếu có thì đưa khách hàng đó vào đuôi hàng đợi. Chúng ta đã biết trước xác suất để trong khoảng thời gian 1 phút có khách hàng đến là p. Vì vậy chúng ta sử dụng hàm rand( ) để sinh ra số nguyên ngẫu nhiên trong khoảng từ 0 tới RAND- MAX. (Hàm rand( ) và hằng số RAND-MAX có trong thư viện chuẩn stdlib.h.). Tại mỗi thời điểm t (t = 1, 2, …), chúng ta gọi hàm rand( ) để sinh ra một số nguyên ngẫu nhiên, nếu số nguyên này nhỏ hơn p * RAND-MAX thì chúng ta cho rằng tại thời điểm đó có khách hàng đến.

```
void QueueEntry(int t)
{
  if (rand( ) < p * RAND-MAX)
        customers.Enqueue(t) ;
}
```

BÀI TẬP.

1. Hãy cài đặt lớp Queue như là lớp dẫn xuất từ lớp cơ sở Llist với dạng thừa kế private. Làm thế nào để các hàm Empty( ), Length( ) của lớp Llist trở thành hàm thành phần public của lớp Queue ?

2. Hàng hai đầu (double-ended queue, hoặc deque) được định nghĩa là một danh sách với các phép toán xen, loại được phép thực hiện ở cả hai đầu. Hãy đặc tả KDLTT này và đưa ra các cách cài đặt thích hợp bởi mảng và bởi DSLK.

3. Cho bàn cờ tướng với một số quân nằm ở các vị trí tuỳ ý và hai vị trí A và B bất kỳ trên bàn cờ. Sử dụng hàng đợi, hãy thiết kế và cài đặt thuật toán tìm đường đi ngắn nhất (nếu có) của con mã từ vị trí A đến vị trí B.