Chương 3 – HÀNG ĐỢI

3.1. Định nghĩa hàng

Trong các ứng dụng máy tính, chúng ta định nghĩa CTDL hàng là một danh sách trong đó việc thêm một phần tử vào được thực hiện ở một đầu của danh sách (cuối hàng), và việc lấy dữ liệu khỏi danh sách thực hiện ở đầu còn lại (đầu hàng). Chúng ta có thể hình dung CTDL hàng cũng giống như một hàng người lần lượt chờ mua vé, ai đến trước được phục vụ trước. Hàng còn được gọi là danh sách FIFO (*First In First Out*)

Hình 3.1- Hàng đợi

Các ứng dụng có sử dụng hàng còn phổ biến hơn các ứng dụng có sử dụng ngăn xếp, vì khi máy tính thực hiện các nhiệm vụ, cũng giống như các công việc trong cuộc sống, mỗi công việc đều cần phải đợi đến lượt của mình. Trong một hệ thống máy tính có thể có nhiều hàng đợi các công việc đang chờ đến lượt được in, được truy xuất đĩa hoặc được sử dụng CPU. Trong một chương trình đơn giản có thể có nhiều công việc được lưu vào hàng đợi, hoặc một công việc có thể khởi tạo một số công việc khác mà chúng cũng cần được lưu vào hàng để chờ đến lượt thực hiện.

Phần tử đầu hàng sẽ được phục vụ trước, thường phần tử này được gọi là **`front`**, hay **`head`** của hàng. Tương tự, phần tử cuối hàng, cũng là phần tử vừa được thêm vào hàng, được gọi là **`rear`** hay **`tail`** của hàng.

Định nghĩa: Một hàng các phần tử kiểu T là một chuỗi nối tiếp các phần tử của T, kèm các tác vụ sau:
1. Tạo mới một đối tượng hàng rỗng.
2. Thêm một phần tử mới vào hàng, giả sử hàng chưa đầy (phần tử dữ liệu mới luôn được thêm vào cuối hàng).
3. Loại một phần tử ra khỏi hàng, giả sử hàng chưa rỗng (phần tử bị loại là phần tử tại đầu hàng, thường là phần tử vừa được xử lý xong).
4. Xem phần tử tại đầu hàng (phần tử sắp được xử lý).

3.2. Đặc tả hàng

Để hoàn tất định nghĩa của cấu trúc dữ liệu trừu tượng hàng, chúng ta đặc tả mọi tác vụ mà hàng thực hiện. Các đặc tả này cũng tương tự như các đặc tả cho ngăn xếp, chúng ta đưa ra tên, kiểu trả về, danh sách thông số, *precondition*, *postcondition* và *uses* cho mỗi phương thức. **`Entry`** biểu diễn một kiểu tổng quát cho phần tử chứa trong hàng.

```
template <class Entry>
Queue<Entry>::Queue();
```
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item);
```
post: nếu hàng còn chỗ, `item` được thêm vào tại `rear`, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `overflow`, hàng không đổi.

```
template <class Entry>
ErrorCode Queue<Entry>::serve();
```
post: nếu hàng không rỗng, phần tử tại `front` được lấy đi, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`, hàng không đổi.

```
template <class Entry>
ErrorCode Queue<Entry>::retrieve(const Entry &item) const;
```
post: nếu hàng không rỗng, phần tử tại `front` được chép vào `item`, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`; cả hai trường hợp hàng đều không đổi.

```
template <class Entry>
bool Queue<Entry>::empty() const;
```
post: hàm trả về `true` nếu hàng rỗng; ngược lại, hàm trả về `false`.

Từ **`append`** (thêm vào hàng) và **`serve`** (đã được phục vụ) được dùng cho các tác vụ cơ bản trên hàng để chỉ ra một cách rõ ràng công việc thực hiện đối với hàng,

và để tránh nhầm lẫn với những từ mà chúng ta sẽ dùng với các cấu trúc dữ liệu khác.

Chúng ta có lớp Queue như sau:

```
template <class Entry>
class Queue {
public:
   Queue();
   bool empty() const;
   ErrorCode append(const Entry &item);
   ErrorCode serve();
   ErrorCode retrieve(Entry &item) const;
};
```

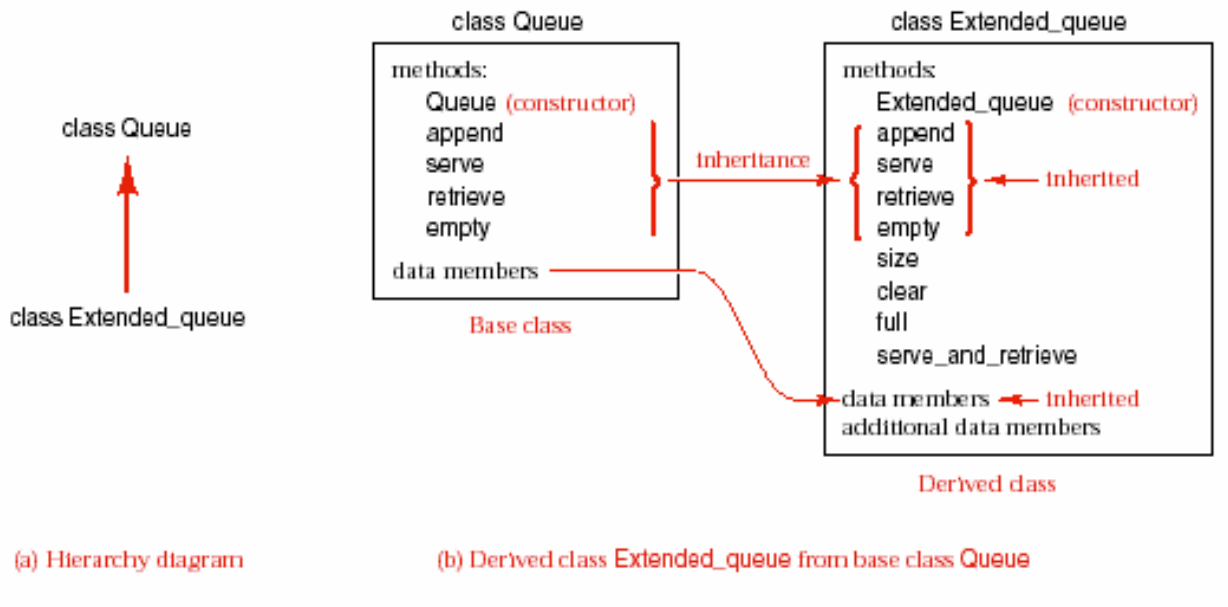
Ngoài các tác vụ cơ bản như **append**, **serve**, **retrieve**, và **empty** đôi khi chúng ta cần thêm một số tác vụ khác. Chẳng hạn như tác vụ **full** để kiểm tra xem hàng đã đầy hay chưa.

Có ba tác vụ rất tiện lợi đối với hàng: **clear** để dọn dẹp các phần tử trong một hàng có sẵn và làm cho hàng rỗng, **size** cho biết số phần tử hiện có trong hàng, cuối cùng là **serve_and_retrieve** gom hai tác vụ **serve** và **retrieve** làm một vì người sử dụng thường gọi hai tác vụ này một lúc.

Chúng ta có thể bổ sung các tác vụ trên vào lớp hàng đã có ở trên. Tuy nhiên, chúng ta có thể tạo lớp mới có thể sử dụng lại các phương thức và cách hiện thực của các lớp đã có. Trong trường hợp này chúng ta xây dựng lớp **Extended_Queue** để bổ sung các phương thức thêm vào các phương thức cơ bản của lớp Queue. Lớp Extended_Queue được gọi là lớp dẫn xuất từ lớp Queue.

Khái niệm dẫn xuất cung cấp một cách định nghĩa các lớp mới đơn giản bằng cách bổ sung thêm các phương thức vào một lớp có sẵn. Khả năng của lớp dẫn xuất sử dụng lại các thành phần của lớp cơ sở được gọi là sự thừa kế. Sự thừa kế (*inheritance*) là một trong các đặc tính cơ bản của lập trình hướng đối tượng.

Chúng ta minh họa mối quan hệ giữa lớp Queue và lớp dẫn xuất Extended_Queue bởi sơ đồ thừa kế (hình 3.2a). Mũi tên chỉ từ lớp dẫn xuất đến lớp cơ sở mà nó thừa kế. Hình 3.2b minh họa sự thừa kế các phương thức và các phương thức bổ sung.



Hình 3.2- Sự thừa kế và lớp dẫn xuất

Chúng ta có lớp `Extended_Queue`:

```
template <class Entry>
class Extended_Queue: public Queue {
public:
   bool full() const;
   int size() const;
   void clear();
   ErrorCode serve_and_retrieve(Entry &item);
};
```

Từ khóa **public** trong khai báo thừa kế có nghĩa là khả năng người sử dụng nhìn thấy đối với các thành phần mà lớp dẫn xuất có được qua sự thừa kế sẽ giống hệt như khả năng người sử dụng nhìn thấy chúng ở lớp cơ sở.

Đặc tả của các phương thức bổ sung:

```
template <class Entry>
bool Extended_Queue<Entry>::full() const;
```
`post`: trả về `true` nếu hàng đầy, ngược lại, trả về `false`. Hàng không đổi.

```
template <class Entry>
void Extended_Queue<Entry>::clear();
```
post: mọi phần tử trong hàng được loại khỏi hàng, hàng trở nên rỗng.

```
template <class Entry>
int Extended_Queue<Entry>::size() const;
```
post: trả về số phần tử hiện có của hàng. Hàng không đổi.

```
template <class Entry>
ErrorCode Extended_Queue<Entry>::serve_and_retrieve(const Entry &item);
```
post: nếu hàng không rỗng, phần tử tại `front` được chép vào `item` đồng thời được loại khỏi hàng, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`, hàng không đổi.

Mối quan hệ giữa lớp `Extended_Queue` và lớp `Queue` thường được gọi là mối quan hệ ***is-a*** vì mỗi đối tượng thuộc lớp `Extended_Queue` cũng là một đối tượng thuộc lớp `Queue` mà có thêm một số đặc tính khác, đó là các phương thức `serve_and_retrieve, full, size` và `clear.`

3.3. Các phương án hiện thực hàng

3.3.1. Các phương án hiện thực hàng liên tục

3.3.1.1. Mô hình vật lý

Tương tự như chúng ta đã làm với ngăn xếp, chúng ta có thể tạo một hàng trong bộ nhớ máy tính bằng một dãy (kiểu dữ liệu `array`) để chứa các phần tử của hàng. Tuy nhiên, ở đây chúng ta cần phải nắm giữ được cả **`front`** và **`rear.`** Một cách đơn giản là chúng ta giữ `front` luôn là vị trí đầu của dãy. Lúc đó, để thêm mới một phần tử vào hàng, chúng ta tăng biến đếm biểu diễn `rear` y hệt như chúng ta thêm phần tử vào ngăn xếp. Để lấy một phần tử ra khỏi hàng, chúng ta phải trả một giá đắt cho việc di chuyển tất cả các phần tử hiện có trong hàng tới một bước để lấp đầy chỗ trống tại `front.` Mặc dù cách hiện thực này rất giống với hình ảnh hàng người sắp hàng đợi để được phục vụ, nhưng nó là một lựa chọn rất dở trong máy tính.

3.3.1.2. Hiện thực tuyến tính

Để việc xử lý hàng có hiệu quả, chúng ta dùng hai chỉ số để nắm giữ `front` và `rear` mà không di chuyển các phần tử. Muốn thêm một phần tử vào hàng, đơn giản chúng ta chỉ cần tăng `rear` lên một và thêm phần tử vào vị trí này. Khi lấy một phần tử ra khỏi hàng chúng ta lấy phần tử tại vị trí `front` và tăng `front` lên một. Tuy nhiên phương pháp này có một nhược điểm lớn, đó là `front` và `rear` luôn luôn tăng chứ không giảm. Ngay cả khi trong hàng không bao giờ có quá hai phần tử, hàng vẫn đòi hỏi một vùng nhớ không có giới hạn nếu như các tác vụ được gọi liên tục như sau:

```
append, append, serve, append, serve, append, serve, append, serve, append, ...
```

Vấn đề ở đây là khi các phần tử trong hàng dịch chuyển tới trong dãy thì các vị trí đầu của dãy sẽ không bao giờ được sử dụng đến. Chúng ta có thể hình dung

hàng lúc đó trông như một con rắn luôn trườn mình tới. Con rắn có lúc dài ra, có lúc ngắn lại, nhưng nếu cứ trườn tới mãi theo một hướng thì cũng phải đến lúc nó gặp điểm dừng của bộ nhớ.

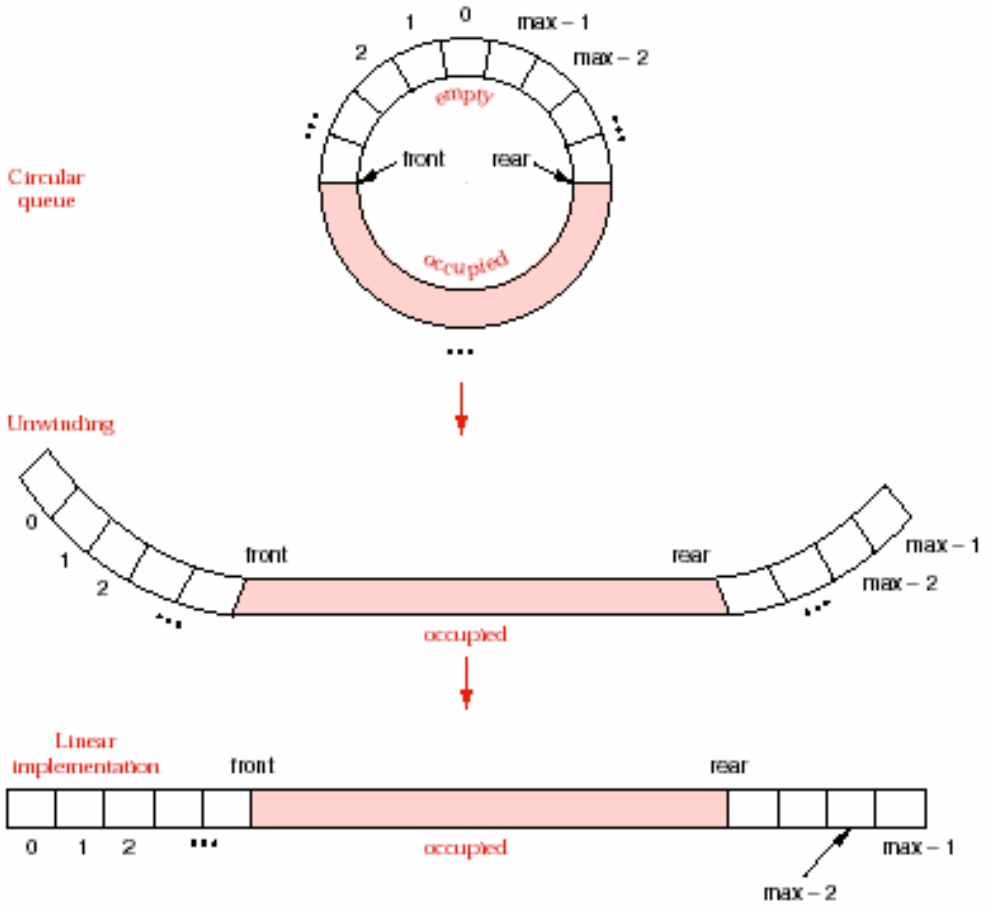
Tuy nhiên, cũng cần chú ý rằng trong các ứng dụng mà có lúc hàng trở nên rỗng (khi một loạt các yêu cầu đang đợi đã được giải quyết hết tại một thời điểm nào đó), thì tại thời điểm này hàng có thể được sắp xếp lại, `front` và `rear` được gán trở lại về đầu dãy. Trường hợp này cho thấy việc sử dụng một sơ đồ đơn giản gồm hai chỉ số và một bộ nhớ tuyến tính như vừa nêu là một cách hiện thực có hiệu quả cao.

3.3.1.3. Dãy vòng

Về ý niệm, chúng ta có thể khắc phục tính thiếu hiệu quả trong việc sử dụng bộ nhớ bằng cách hình dung dãy có dạng vòng thay vì tuyến tính. Khi phần tử được thêm vào hay lấy ra khỏi hàng, điểm đầu của hàng sẽ đuổi theo điểm cuối của hàng vòng theo dãy, và như vậy con rắn vẫn có thể trườn tới vô hạn nhưng vẫn bị nhốt trong một vòng có giới hạn. Tại các thời điểm khác nhau, hàng sẽ chiếm những phần khác nhau trong dãy vòng, nhưng chúng ta sẽ không bao giờ phải lo về sự vượt giới hạn bộ nhớ trừ khi dãy thật sự không còn phần tử trống, trường hợp này được xem như hàng đầy, `ErrorCode` sẽ nhận trị `overflow`.

Hiện thực của dãy vòng

Vấn đề tiếp theo của chúng ta là dùng một dãy tuyến tính để mô phỏng một dãy vòng. Các vị trí trong vòng tròn được đánh số từ 0 đến max-1, trong đó max là tổng số phần tử trong dãy vòng. Để hiện thực dãy vòng, chúng ta cũng sử dụng các phần tử được đánh số tương tự dãy tuyến tính. Sự thay đổi các chỉ số chỉ đơn giản là phép lấy phần dư trong số học: khi một chỉ số tăng vượt qua giới hạn `max`-1, nó được bắt đầu trở lại với trị 0. Điều này tương tự việc cộng thêm giờ trong đồng hồ mặt tròn, các giờ được đánh số từ 1 đến 12, nếu chúng ta cộng thêm 4 giờ vào 10 giờ chúng ta sẽ có 2 giờ.



Hình 3.3- Hàng trong dãy vòng

Dãy vòng trong C++

Trong C++, chúng ta có thể tăng chỉ số *i* trong một dãy vòng như sau:

```
i = ((i+1) == max) ? 0: (i+1);
```

hoặc

```
if ((i+1) == max)  i = 0; else i = i+1;
```

hoặc

```
i = (i+1) % max;
```

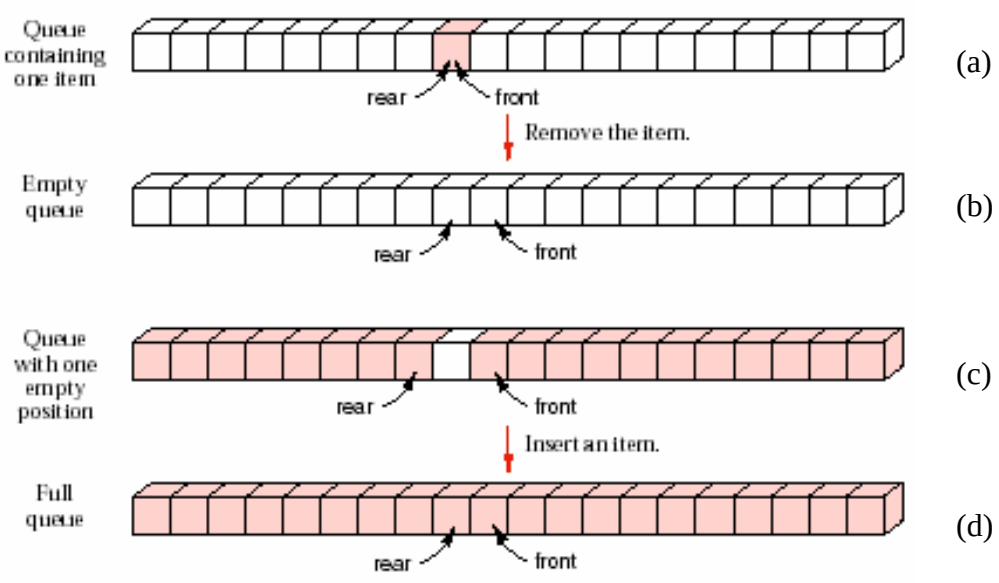
Các điều kiện biên

Trước khi viết những giải thuật thêm hoặc loại phần tử ra khỏi hàng, chúng ta hãy xem xét đến các điều kiện biên (*boundary conditions*), đó là các dấu hiệu cho biết hàng còn rỗng hay đã đầy.

Nếu trong hàng chỉ có một phần tử thì cả `front` và `rear` đều chỉ đến phần tử này (hình 3.4 a). Khi phần tử này được loại khỏi hàng, `front` sẽ tăng lên 1. Do đó hàng là rỗng khi `rear` chỉ vị trí ngay trước `front` (hình 3.4 b).

Do `rear` di chuyển về phía trước mỗi khi thêm phần tử mới, nên khi hàng sắp đầy và bằng cách di chuyển vòng thì `rear` cũng sẽ gần gặp `front` trở lại (hình 3.3 c). Lúc này khi phần tử cuối cùng được thêm vào làm cho hàng đầy thì `rear` cũng chỉ vị trí ngay trước `front` (hình 3.4 d).

Chúng ta gặp một khó khăn mới: vị trí tương đối của `front` và `rear` giống hệt nhau trong cả hai trường hợp hàng đầy và hàng rỗng.



Hình 3.4- Hình ảnh minh họa hàng rỗng và hàng đầy

Các cách giải quyết có thể

Có ít nhất 3 cách giải quyết cho vấn đề nêu trên. Cách thứ nhất là dành lại một vị trí trống khi hàng đầy, `rear` sẽ cách `front` một vị trí giữa. Cách thứ hai là sử dụng thêm một biến, chẳng hạn một biến cờ kiểu `bool` sẽ có trị `true` khi `rear` nhích đến sát `front` trong trường hợp hàng đầy (chúng ta có thể tùy ý chọn trường hợp hàng đầy hay rỗng), hay một biến đếm để đếm số phần tử hiện có trong hàng. Cách thứ ba là cho một hoặc cả hai chỉ số `front` và `rear` mang một trị đặc biệt nào đó để chỉ ra hàng rỗng, ví dụ như `rear` sẽ là -1 khi hàng rỗng.

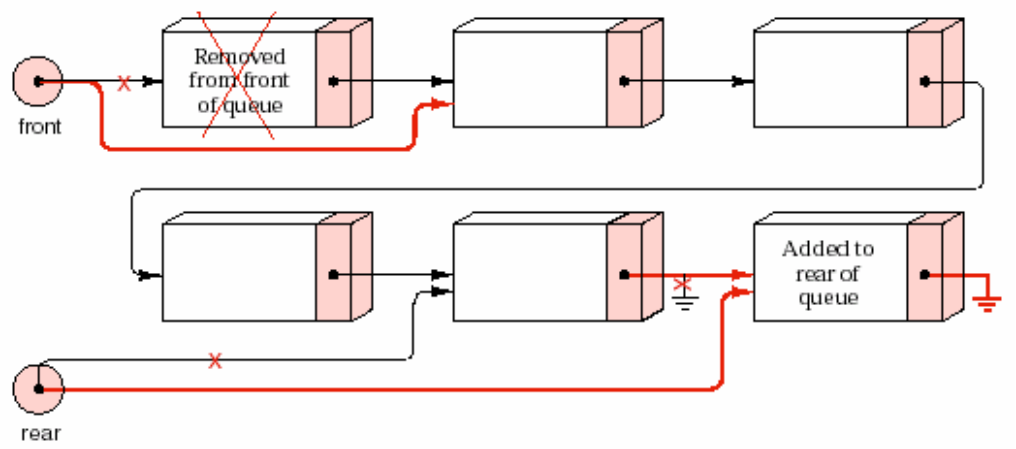
3.3.1.4. Tổng kết các cách hiện thực cho hàng liên tục

Để tổng kết những điều đã bàn về hàng, chúng ta liệt kê dưới đây tất cả các phương pháp mà chúng ta đã thảo luận về các cách hiện thực hàng.

- Mô hình vật lý: một dãy tuyến tính có `front` luôn chỉ vị trí đầu tiên trong hàng và mọi phần tử của hàng phải di chuyển tới một bước khi phần tử tại `front` được lấy đi. Đây là phương pháp rất dở trong máy tính nói chung.
- Một dãy tuyến tính có hai chỉ số `front` và `rear` luôn luôn tăng. Đây là phương pháp tốt nếu như hàng có thể được làm rỗng.
- Một dãy vòng có hai chỉ số `front,rear` và một vị trí để trống.
- Một dãy vòng có hai chỉ số `front,rear` và một cờ cho biết hàng đầy (hoặc rỗng).
- Một dãy vòng có hai chỉ số `front,rear` và một biến đếm số phần tử hiện có trong hàng.
- Một dãy vòng có hai chỉ số `front,rear` mà hai chỉ số này sẽ mang trị đặc biệt trong trường hợp hàng rỗng.

3.3.2. Phương án hiện thực hàng liên kết

Bằng cách sử dụng bộ nhớ liên tục, việc hiệc thực hàng khó hơn việc hiện thực ngăn xếp rất nhiều do chúng ta dùng vùng nhớ tuyến tính để giả lặp tổ chức vòng và gặp khó khăn trong việc phân biệt một hàng đầy với một hàng rỗng. Tuy nhiên, hiện thực hàng liên kết lại thực sự dễ dàng như hiện thực ngăn xếp liên kết. Chúng ta chỉ cần nắm giữ hai con trỏ, `front` và `rear` để tham chiếu đến phần tử đầu và phần tử cuối của hàng. Các tác vụ thêm hay loại phần tử trên hàng được minh họa trong hình 3.5.



Hình 3.5 Các tác vụ thêm và loại phần tử trên hàng liên kết

3.4. Hiện thực hàng

3.4.1. Hiện thực hàng liên tục

Hiện thực vòng cho hàng liên tục trong C++

Phần này trình bày các phương thức của cách hiện thực hàng bằng dãy vòng có biến đếm các phần tử. Chúng ta có định nghĩa lớp `Queue` như sau:

```
const int maxQueue = 10; //  Giá trị nhỏ chỉ để kiểm tra CTDL Queue.

template <class Entry>
class Queue {
public:
   Queue();
   bool empty() const;
   ErrorCode serve();
   ErrorCode append(const Entry &item);
   ErrorCode retrieve(Entry &item) const;
protected:
   int count;
   int front, rear;
   Entry entry[maxQueue];
};
```

Các dữ liệu thành phần trong lớp `Queue` được khai báo **`protected.`** Đối với người sử dụng sẽ không có gì thay đổi, nghĩa là chúng vẫn không được người sử dụng nhìn thấy và vẫn đảm bảo sự che dấu thông tin. Mục đích ở đây là khi chúng ta xây dựng lớp `Extended_Queue` dẫn xuất từ lớp `Queue` thì lớp dẫn xuất sẽ sử dụng được các dữ liệu thành phần này. Khi các dữ liệu thành phần của lớp cơ sở được khai báo là **`private`** thì lớp dẫn xuất cũng sẽ không nhìn thấy chúng.

```
template <class Entry>
Queue<Entry>::Queue()
/*
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.
*/
```

```
{
   count = 0;
   rear = maxQueue - 1;
   front = 0;
}
```

```
template <class Entry>
bool Queue<Entry>::empty() const
/*
post: hàm trả về true nếu hàng rỗng; ngược lại, hàm trả về false.
*/
{   return count == 0;
}
```

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item)
/*
post: nếu hàng còn chỗ, item được thêm vào tại rear, ErrorCode trả về là success; ngược lại,
     ErrorCode trả về là overflow, hàng không đổi.
*/
{
   if (count >= maxQueue) return overflow;
   count++;
   rear = ((rear + 1) == maxQueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}
template <class Entry>
ErrorCode Queue<Entry>::serve()
/*
post: nếu hàng không rỗng, phần tử tại front được lấy đi, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là underflow, hàng không đổi.
*/
{
   if (count <= 0) return underflow;
   count--;
   front = ((front + 1) == maxQueue) ? 0 : (front + 1);
   return success;
}
```

```
template <class Entry>
ErrorCode Queue<Entry>::retrieve(Entry &item) const
/*
post: nếu hàng không rỗng, phần tử tại front được chép vào item, ErrorCode trả về là
      success; ngược lại, ErrorCode trả về là underflow; cả hai trường hợp hàng đều không
      đổi.
*/
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

Chúng ta dành phương thức empty lại như là bài tập. Phương thức size dưới đây rất đơn giản do lớp Extended_Queue sử dụng được thuộc tính count của lớp

Queue, nếu như count được khai báo là private thì phương thức size của lớp Extended_Queue phải sử dụng hàng loạt câu lệnh gọi các phương thức public của Queue như serve, retrieve, append mới thực hiện được. Các phương thức còn lại của Extended_Queue cũng tương tự, và chúng ta dành lại phần bài tập.

```
template <class Entry>
int Extended_Queue<Entry>::size() const
/*
post: trả về số phần tử hiện có của hàng. Hàng không đổi.
*/
{
   return count;
}
```

3.4.2. Hiện thực hàng liên kết

3.4.2.1. Các khai báo cơ bản

Với mọi hàng, chúng ta dùng kiểu Entry cho kiểu của dữ liệu lưu trong hàng. Đối với hiện thực liên kết, chúng ta khai báo các **node** tương tự như đã làm cho ngăn xếp liên kết trong chương 2. Chúng ta có đặc tả dưới đây:

```
template <class Entry>
class Queue {
public:
//  Các phương thức chuẩn của hàng
   Queue();
   bool empty() const;
   ErrorCode append(const Entry &item);
   ErrorCode serve();
   ErrorCode retrieve(Entry &item) const;
//  Các phương thức bảo đảm tính an toàn cho hàng liên kết
   ~Queue();
   Queue(const Queue<Entry> &original);
   void operator =(const Queue<Entry> &original);
protected:
   Node<Entry> *front, *rear;
};
```

Constructor thứ nhất khởi tạo một hàng rỗng:

```
template <class Entry>
Queue<Entry>::Queue()
/*
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.
*/
{
   front = rear = NULL;
}
```

Phương thức *append* thêm một phần tử vào đầu *rear* của hàng:

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item)
/*
post: nếu hàng còn chỗ, item được thêm vào tại rear, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là overflow, hàng không đổi.
*/
{
   Node *new_rear = new Node<Entry> (item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;   // trường hợp đặc biệt: thêm vào hàng
                                                   đang rỗng.
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}
```

Trường hợp hàng rỗng cần phân biệt với các trường hợp bình thường khác, do khi thêm một node vào một hàng rỗng cần phải gán cả front và rear tham chiếu đến node này, trong khi việc thêm một node vào một hàng không rỗng chỉ có rear là cần được gán lại.

Phương thức loại một phần tử ra khỏi hàng được viết như sau:

```
template <class Entry>
ErrorCode Queue::serve()
/*
post: nếu hàng không rỗng, phần tử tại front được lấy đi, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là underflow, hàng không đổi.
*/
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear= NULL;// trường hợp đặc biệt: loại phần tử cuối cùng của hàng
   delete old_front;
   return success;
}
```

Một lần nữa trường hợp hàng rỗng cần được xem xét riêng. Khi phần tử được loại khỏi hàng không phải là phần tử cuối trong hàng, chỉ có front cần được gán lại, ngược lại cả front và rear cần phải gán về NULL.

Các phương thức khác của hàng liên kết được dành lại cho phần bài tập.

Nếu so sánh với hàng liên tục, chúng ta sẽ thấy rằng hàng liên kết dễ hiểu hơn cả về mặt khái niệm cả về cách hiện thực chương trình.

3.4.3. Hàng liên kết mở rộng

Hiện thực liên kết của lớp Queue cung cấp một lớp cơ sở cho các lớp khác. Định nghĩa dưới đây dành cho lớp dẫn xuất Extended_Queue hoàn toàn tương tự như hàng liên tục.

```
template <class Entry>
class Extended_queue: public Queue {
public:
   bool full() const;
   int size() const;
   void clear();
   ErrorCode serve_and_retrieve(Entry &item);
};
```

Mặc dù lớp Extended_Queue này có hiện thực liên kết, nhưng nó không cần các phương thức như *copy contructor*, *overloaded assignment*, hoặc *destructor*. Đối với một trong các phương thức này, trình biên dịch sẽ gọi các phương thức mặc định của lớp Extended_Queue. Phương thức mặc định của một lớp dẫn xuất sẽ gọi phương thức tương ứng của lớp cơ sở. Chẳng hạn, khi một đối tượng của lớp Extended_Queue chuẩn bị ra khỏi tầm vực, *destructor* mặc định của lớp Extended_Queue sẽ gọi *destructor* của lớp Queue: mọi node cấp phát động của Extended_Queue đều được giải phóng. Do lớp Extended_Queue của chúng ta không chứa thêm thuộc tính con trỏ nào ngoài các thuộc tính con trỏ thừa kế từ lớp Queue, *destructor* mà trình biên dịch gọi đã làm được tất cả những điều mà chúng ta mong muốn.

Các phương thức được khai báo cho lớp Extended_Queue cần được viết lại cho phù hợp với cấu trúc liên kết của hàng. Chẳng hạn, phương thức *size* cần sử dụng một con trỏ tạm window để duyệt hàng (nói cách khác, con trỏ window sẽ di chuyển dọc theo hàng và lần lượt chỉ đến từng node trong hàng).

```
template <class Entry>
int Extended_queue<Entry>::size() const
/*
post: trả về số phần tử hiện có của hàng. Hàng không đổi.
*/
{
   Node *window = front;
   int count = 0;
   while (window != NULL) {
      window = window->next;
      count++;
   }
   return count;
}
```

Các phương thức khác của Extended_Queue liên kết xem như bài tập.