CHƯƠNG 5

DANH SÁCH LIÊN KẾT

KDLTT danh sách đã được xác định trong chương 4 với các phép toán xen vào, loại bỏ và tìm kiếm một đối tượng khi cho biết vị trí của nó trong danh sách và một loạt các phép toán khác cần thiết cho các xử lý đa dạng khác trên danh sách. Trong chương 4 chúng ta đã cài đặt danh sách bởi mảng. Hạn chế cơ bản của cách cài đặt này là mảng có cỡ cố định, mà danh sách thì luôn phát triển khi ta thực hiện phép toán xen vào, và do đó trong quá trình xử lý danh sách, nó có thể có độ dài vượt quá cỡ của mảng. Một cách lựa chọn khác là cài đặt danh sách bởi cấu trúc dữ liệu **danh sách liên kết** (DSLK). Các thành phần dữ liệu trong CTDL này được liên kết với nhau bởi các con trỏ; mỗi thành phần chứa con trỏ trỏ tới thành phần tiếp theo. DSLK có thể "nối dài ra" khi cần thiết, trong khi mảng chỉ lưu được một số cố định đối tượng dữ liệu.

Nội dung chính của chương này là như sau: Mục 5.1 trình bày những kiến thức cần thiết về con trỏ và cấp phát động bộ nhớ trong C + + được sử dụng thường xuyên sau này. Mục 5.2 sẽ nói về DSLK đơn và các dạng DSLK khác: DSLK vòng tròn, DSLK kép. Trong mục 5.3, chúng ta sẽ cài đặt KDLTT danh sách bởi DSLK. Sau đó, trong mục 5.4, chúng ta sẽ phân tích so sánh hai phương pháp cài đặt KDLTT danh sách: cài đặt bởi mảng và cài đặt bởi DSLK. Cuối cùng, trong mục 5.5, chúng ta sẽ thảo luận về sự cài đặt KDLTT tập động bởi DSLK.

1.1 CON TRỎ VÀ CẤP PHÁT ĐỘNG BỘ NHỚ

Cũng như nhiều ngôn ngữ lập trình khác, C + + có các con trỏ, nó cho phép ta xây dựng nên các DSLK và các CTDL phức tạp khác. **Biến con trỏ** (pointer variable), hay gọi tắt là **con trỏ** (pointer), là biến chứa **địa chỉ** của một tế bào nhớ trong bộ nhớ của máy tính. Nhờ có con trỏ, ta định vị được tế bào nhớ và do đó có thể truy cập được nội dung của nó.

Để khai báo một biến con trỏ P lưu giữ địa chỉ của tế bào nhớ chứa dữ liệu kiểu Item, chúng ta viết

```
Item *  P;
```

Chẳng hạn, khai báo

```
int *   P ;
```

nói rằng P là biến con trỏ nguyên, tức là P chỉ trỏ tới các tế bào nhớ chứa số nguyên.

Muốn khai báo P và Q là hai con trỏ nguyên, ta cần viết

```
int *   P;
```

```
int *   Q;
```

hoặc

```
int * P , * Q;
```

Cần nhớ rằng, nếu viết

```
int * P, Q;
```

thì chỉ có P là biến con trỏ nguyên, còn Q là biến nguyên.

Nếu chúng ta khai báo:

```
int * P;
int   x;
```

thì các biến này được cấp phát bộ nhớ trong thời gian dịch, sự cấp phát bộ nhớ như thế được gọi là **cấp phát tĩnh**, nó xảy ra trước khi chương trình được thực hiện. Nội dung của các tế bào nhớ đã cấp phát cho P và x chưa được xác định, xem minh hoạ trong hình 5.1a.

Bạn có thể đặt địa chỉ của x vào P bằng cách sử dụng toán tử lấy địa chỉ & như sau:

```
P = &x;
```

Khi đó chúng ta có hoàn cảnh được minh hoạ trong hình 5.16, ký hiệu *P biểu diễn tế bào nhớ mà con trỏ P trỏ tới

Đến đây, nếu bạn viết tiếp:

```
*P = 5;
```

thì biến x sẽ có giá trị 5, tức là hiệu quả của lệnh gán *P = 5 tương đương với lệnh gán x = 5, như được minh hoạ trong hình 5.1c.

Sự cấp phát bộ nhớ còn có thể xảy ra trong thời gian thực hiện chương trình, và được gọi là **cấp phát động** (dynamic allocation). Toán tử **new** trong C + + cho phép ta thực hiện sự cấp phát động bộ nhớ.

Bây giờ, nếu bạn viết

```
P = new  int;
```

thì một tế bào nhớ lưu giữ số nguyên được cấp phát cho biến động *P và con trỏ P trỏ tới tế bào nhớ đó, như được minh hoạ trong hình 5.1d. Cần lưu ý rằng biến động *P chỉ tồn tại khi nó đã được cấp phát bộ nhớ, và khi đó, nó được sử dụng như các biến khác.

Tiếp theo, bạn có thể viết

```
*P = 8;
```

khi đó số nguyên 8 được đặt trong tế bào nhớ mà con trỏ P trỏ tới, ta có hoàn cảnh như trong hình 5.1e.

Giả sử, bạn viết tiếp:

```
int  *  Q;
Q = P;
```

Khi đó, con trỏ Q sẽ trỏ tới tế bào nhớ mà con trỏ P đã trỏ tới, như được minh hoạ trong hình 5.1f.

Nếu bây giờ bạn không muốn con trỏ trỏ tới bất kỳ tế bào nhớ nào trong bộ nhớ, bạn có thể sử dụng hằng con trỏ NULL. Trong các minh hoạ, chúng ta sẽ biểu diễn hằng NULL bởi dấu chấm. Dòng lệnh:

Q = NULL

sẽ cho hiệu quả như trong hình 5.1g.

Một khi biến động *P không cần thiết nữa trong chương trình, chúng ta có thể thu hồi tế bào nhớ đã cấp phát cho nó và trả về cho hệ thống. Toán tử **delete** thực hiện công việc này.

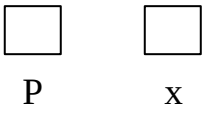
Dòng lệnh

delete P;

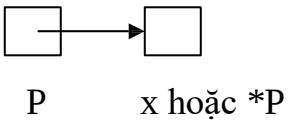
sẽ thu hồi tế bào nhớ đã cấp phát cho biến động *P bởi toán tử new, và chúng ta sẽ có hoàn cảnh được minh hoạ bởi hình 5.1h

---

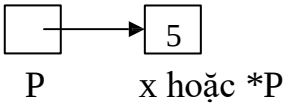
a) int * P;
int x;



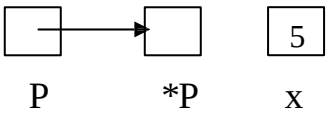
b) P = &x ;



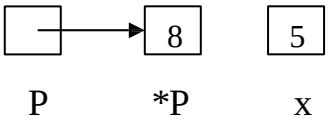
c) *P = 5;
(hoặc x = 5;)



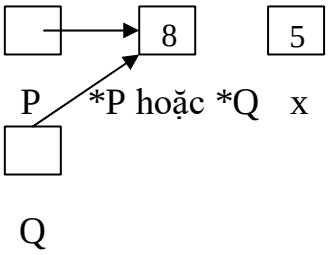
d) P = new int ;



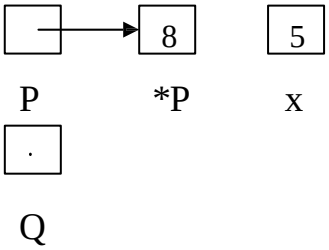
e) *P = 8 ;



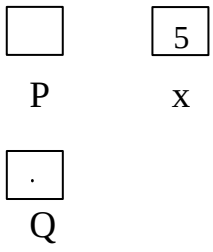
f) int * Q ;
Q = P ;



g) Q = NULL;



h) delete P ;



Hình 5.1. Các thao tác với con trỏ.

Cấp phát mảng động

Khi chúng ta khai báo một mảng, chẳng hạn

```
int  A[30];
```

thì chương trình dịch sẽ cấp phát 30 tế bào nhớ liên tiếp để lưu các số nguyên, trước khi chương trình được thực hiện. Mảng A là mảng tĩnh, vì bộ nhớ được cấp phát cho nó là cố định và tồn tại trong suốt thời gian chương trình thực hiện, đồng thời A là con trỏ trỏ tới thành phần đầu tiên A[0] của mảng.

Chúng ta có thể sử dụng toán tử **new** để cấp phát cả một mảng. Sự cấp phát này xảy ra trong thời gian thực hiện chương trình, và được gọi là sự cấp phát mảng động.

Giả sử có các khai báo sau:

```
int    size;
double *  B;
```

Sau đó, chúng ta đưa vào lệnh:

```
size  = 5;
B  =  new double[size]l
```

Khi đó, toán tử new sẽ cấp phát một mảng động B gồm 5 tế bào nhớ double, và con trỏ B trỏ tới thành phần đầu tiên của mảng này, như trong hình 5.2b.

Chúng ta có thể truy cập tới thành phần thứ i trong mảng động B bởi ký hiệu truyền thống B[i], hoặc *(B + i). Chẳng hạn, nếu bạn viết tiếp

```
B[3] = 3.14;
hoặc *(B + 3) = 3.14;
```

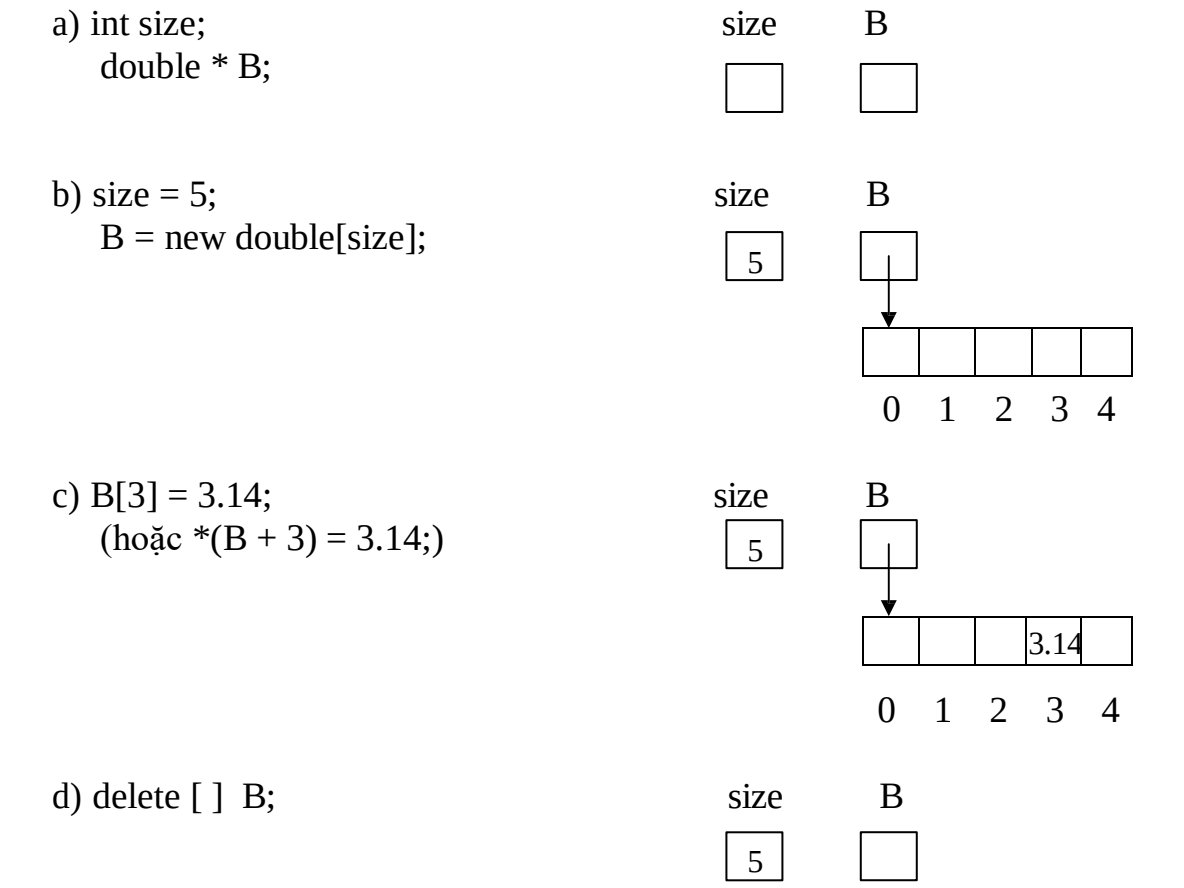
thì thành phần thứ 3 trong mảng B sẽ lưu 3.14, như được minh hoạ trong hình 5.3c. Tóm lại, các thao tác đối với mảng động là giống như đối với mảng tĩnh.

Một khi mảng động không còn được sử dụng nữa, chúng ta có thể thu hồi bộ nhớ đã cấp phát cho nó và trả về cho hệ thống để có thể sử dụng lại cho các công việc khác.

Nếu bạn viết

```
delete [ ] B;
```

thì mảng động B được thu hồi trả lại cho hệ thống và chúng ta có hoàn cảnh như trong hình 5.2d



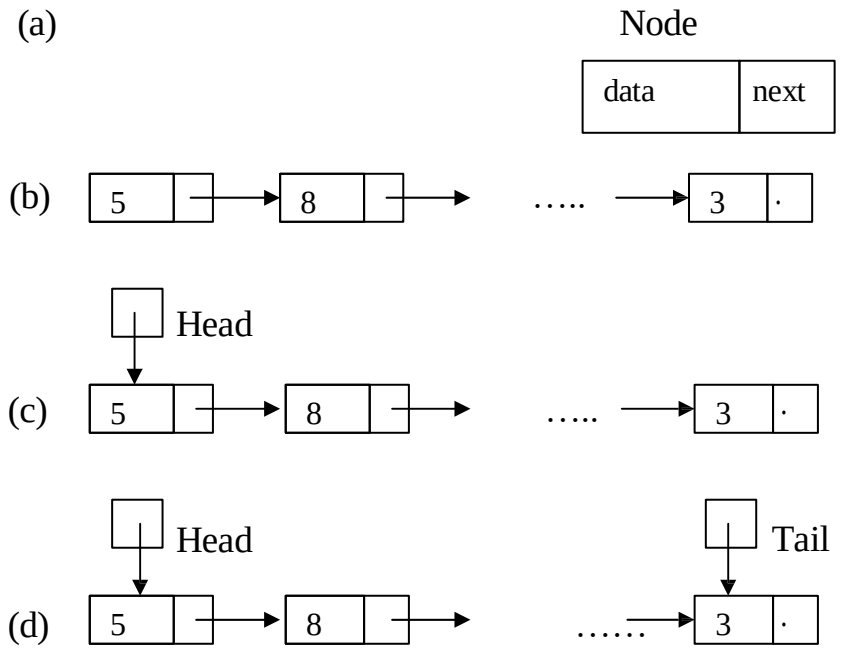
Hình 5.2. Cấp phát và thu hồi mảng động.

1.2 CẤU TRÚC DỮ LIỆU DANH SÁCH LIÊN KẾT

Trong mục này, chúng ta sẽ trình bày cấu trúc dữ liệu DSLK và các phép toán cơ bản trên DSLK.

Danh sách liên kết đơn

Danh sách liên kết đơn, gọi tắt là danh sách liên kết (DSLK) được tạo nên từ các thành phần được liên kết với nhau bởi các con trỏ. Mỗi thành phần trong DSLK chứa một dữ liệu và một con trỏ trỏ tới thành phần tiếp theo. Chúng ta sẽ mô tả mỗi thành phần của DSLK như một hộp gồm hai ngăn: một ngăn chứa dữ liệu data và một ngăn chứa con trỏ next, như trong hình 5.3a. Hình 5.3b biểu diễn một DSLK chứa dữ liệu là các số nguyên.



Hình 5.3. a) Một thành phần của DSLK.
b) DSLK chứa các số nguyên.
c) DSLK với một con trỏ ngoài Head
d) DSLK với hai con trỏ ngoài Head và Tail.

Chúng ta sẽ biểu diễn mỗi thành phần của DSLK bởi một cấu trúc trong C + +, cấu trúc này gồm hai biến thành phần: biến data có kiểu Item nào đó, và biến next là con trỏ trỏ tới cấu trúc này.

```
struct  Node
{
   Item  data ;
   Node*  next ;
} ;
```

Cần lưu ý rằng, trong thành phần cuối cùng của DSLK, giá trị của next là hằng con trỏ NULL, có nghĩa là nó không trỏ tới đâu cả và được biểu diễn bởi dấu chấm.

Để tiến hành các xử lý trên DSLK, chúng ta cần phải có khả năng truy cập tới từng thành phần của DSLK. Nếu biết được thành phần đầu tiên, "đi theo" con trỏ next, ta truy cập tới thành phần thứ hai, rồi từ thành phần thứ hai ta có thể truy cập tới thành phần thứ ba, … Do đó, khi lưu trữ một DSLK, chúng ta cần phải xác định một con trỏ trỏ tới thành phần đầu tiên trong DSLK, con trỏ này sẽ được gọi là con trỏ đầu Head. Như vậy, khi lập trình xử lý DSLK với con trỏ đầu Head, chúng ta cần đưa vào khai báo

Node* Head ;

Khi mà DSLK không chứa thành phần nào cả (ta nói DSLK rỗng), chúng ta lấy hằng con trỏ NULL làm giá trị của biến Head. Do đó, khi sử dụng DSLK với con trỏ đầu Head, để khởi tạo một DSLK rỗng, chúng ta chỉ cần đặt:

Head = NULL ;

Hình 5.3c biểu diễn DSLK với con trỏ đầu Head. Cần phân biệt rằng, con trỏ Head là con trỏ ngoài, trong khi các con trỏ next trong các thành phần là các con trỏ trong của DSLK, chúng làm nhiệm vụ "liên kết" các dữ liệu.

Trong nhiều trường hợp, để các thao tác trên DSLK được thuận lợi, ngoài con trỏ đầu Head người ta sử dụng thêm một con trỏ ngoài khác trỏ tới thành phần cuối cùng của DSLK : con trỏ đuôi Tail. Hình 5.3d biểu diễn một DSLK với hai con trỏ ngoài Head và Tail. Trong trường hợp DSLK rỗng, cả hai con trỏ Head và Tail đều có giá trị là NULL.

Sau đây chúng ta sẽ xét các phép toán cơ bản trên DSLK. Các phép toán này là cơ sở cho nhiều thuật toán trên DSLK. Chúng ta sẽ nghiên cứu các phép toán sau:

- Xen một thành phần mới vào DSLK.
- Loại một thành phần khỏi DSLK.
- Đi qua DSLK (duyệt DSLK).

1. Xen một thành phần mới vào DSLK

Giả sử chúng ta đã có một DSLK với một con trỏ ngoài Head (hình 5.3c), chúng ta cần xen một thành phần mới chứa dữ liệu là value vào sau (trước) một thành phần được trỏ tới bởi con trỏ P (ta sẽ gọi thành phần này là thành phần P).

Việc đầu tiên cần phải làm là tạo ra thành phần mới được trỏ tới bởi con trỏ Q, và đặt dữ liệu value vào thành phần này:

```
Node* Q;
Q = new Node ;
Q → data = value;
```

Các thao tác cần tiến hành để xen một thành phần mới phụ thuộc vào vị trí của thành phần P, nó ở đầu hay giữa DSLK, và chúng ta cần xen vào sau hay trước P.

Xen vào đầu DSLK. Trong trường hợp này, thành phần mới được xen vào trở thành đầu của DSLK, và do đó giá trị của con trỏ Head cần phải được thay đổi. Trước hết ta cần "móc nối" thành phần mới vào đầu DSLK, sau đó cho con trỏ Head trỏ tới thành phần mới, như được chỉ ra trong hình 5.4a. Các thao tác này được thực hiện bởi các lệnh sau:

```
Q → next = Head;
Head = Q;
```

Chúng ta có nhận xét rằng, thủ tục trên cũng đúng cho trường hợp DSLK rỗng, bởi vì khi DSLK rỗng thì giá trị của Head là NULL và do đó giá trị của con trỏ next trong thành phần đầu mới được xen vào cũng là NULL.

Xen vào sau thành phần P. Giả sử DSLK không rỗng và con trỏ P trỏ tới một thành phần bất kỳ của DSLK. Để xen thành phần mới Q vào sau thành phần P, chúng ta cần "móc nối" thành phần Q vào trong "dây chuyền" đã có sẵn, như được chỉ ra trong hình 5.4b. Trước hết ta cho con trỏ next của thành phần mới Q trỏ tới thành phần đi sau P, sau đó cho con trỏ next của thành phần P trỏ tới Q:

Q → next = P → next;

P → next = Q;

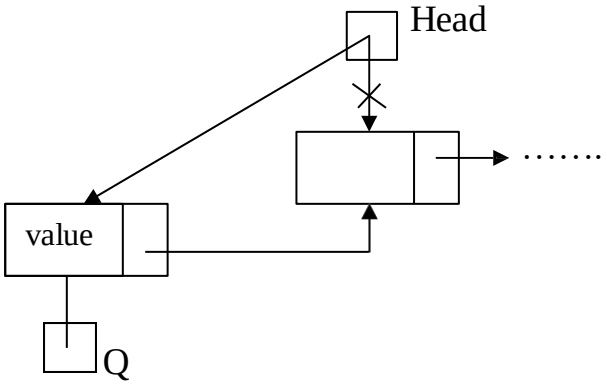
Cũng cần lưu ý rằng, thủ tục trên vẫn làm việc tốt cho trường hợp P là thành phần cuối cùng trong DSLK.

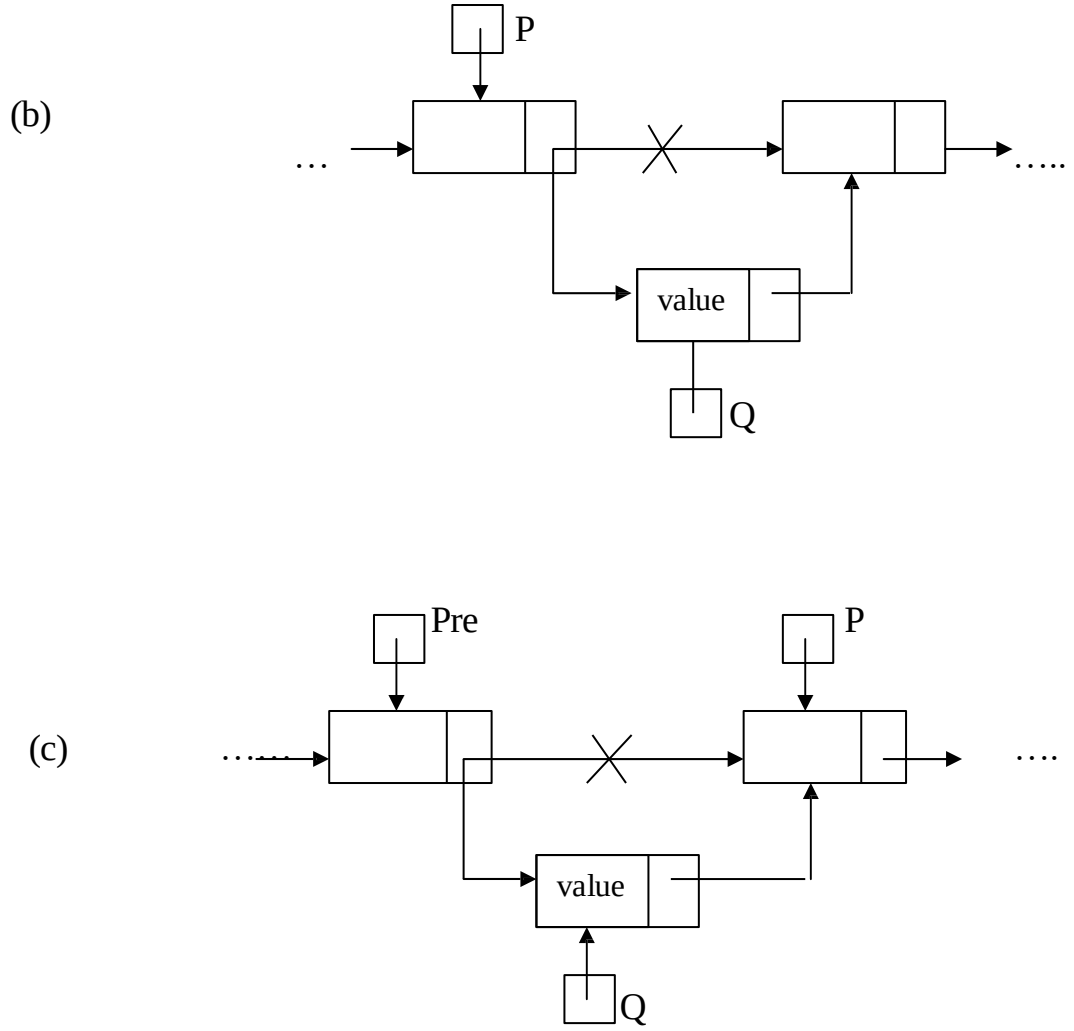
Xen vào trước thành phần P. Giả sử DSLK không rỗng, và con trỏ P trỏ tới thành phần không phải là đầu của DSLK. Trong trường hợp này, để xen thành phần mới vào trước thành phần P, chúng ta cần biết thành phần đi trước P để có thể "móc nối" thành phần mới vào trước P. Giả sử thành phần này được trỏ tới bởi con trỏ Pre. Khi đó việc xen thành phần mới vào trước thành phần P tương đương với việc xen nó vào sau thành phần Pre, xem hình 5.4c. Tức là cần thực hiện các lệnh sau:

Q → next = P; (hoặc Q → next = Pre → next)

Pre → next = Q;

(a)





Hình 5.4. a) Xen vào đầu DSLK.
b) Xen vào sau thành phần P.
c) Xen vào trước thành phần P.

2. Loại một thành phần khỏi DSLK

Chúng ta cần loại khỏi DSLK một thành phần được trỏ tới bởi con trỏ P. Cũng như phép toán xen vào, khi loại một thành phần khỏi DSLK, cần quan tâm tới nó nằm ở đâu trong DSLK. Nếu thành phần cần loại ở giữa DSLK thì giá trị của con trỏ ngoài Head không thay đổi, nhưng nếu ta loại đầu DSLK thì thành phần tiếp theo trở thành đầu của DSLK, và do đó giá trị của con trỏ Head cần thay đổi thích ứng.

Loại đầu DSLK. Đó là trường hợp P trỏ tới đầu DSLK. Để loại đầu DSLK, ta chỉ cần cho con trỏ Head trỏ tới thành phần tiếp theo (xem hình 5.5a). Với thao tác đó thành phần đầu thực sự đã bị loại khỏi DSLK, song nó

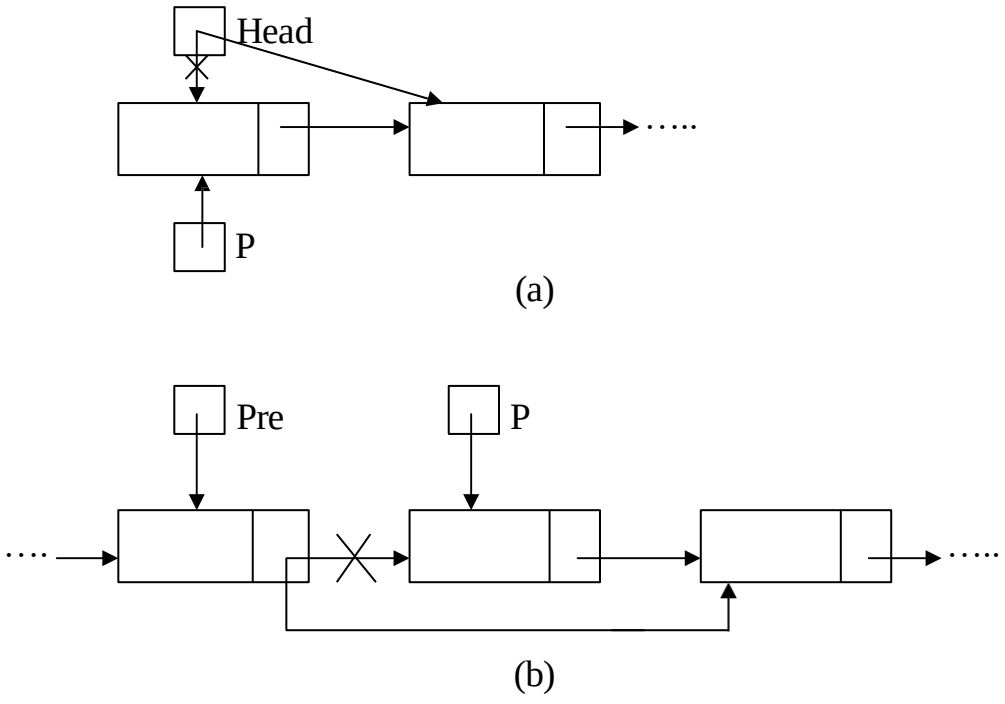
vẫn còn tồn tại trong bộ nhớ. Sẽ là lãng phí bộ nhớ, nếu để nguyên như thế, vì vậy chúng ta cần thu hồi bộ nhớ của thành phần bị loại trả về cho hệ thống. Như vậy việc loại thành phần đầu DSLK được thực hiện bởi các lệnh sau:

Head = Head → next;
delete P;

Loại thành phần không phải đầu DSLK. Trong trường hợp này để "tháo gỡ" thành phần P khỏi "dây chuyền", chúng ta cần móc nối thành phần đi trước P với thành phần đi sau P (xem hình 5.5b). Giả sử thành phần đi trước thành phần P được trỏ tới bởi con trỏ Pre. Chúng ta cần cho con trỏ next của thành phần đi trước P trỏ tới thành phần đi sau P, sau đó thu hồi bộ nhớ của thành phần bị loại. Do đó thủ tục loại thành phần P là như sau:

Pre → next = P → next;
delete P;

Thủ tục loại trên cũng đúng cho trường hợp P là thành phần cuối cùng trong DSLK.



Hình 5.5. (a) Loại đầu DSLK.
(b) Loại thành phần P

3. Đi qua DSLK (Duyệt DSLK)

Giả sử rằng, bạn đã có một DSLK, đi qua DSLK có nghĩa là truy cập tới từng thành phần của DSLK bắt đầu từ thành phần đầu tiên đến thành phần cuối cùng và tiến hành các xử lý cần thiết với mỗi thành phần của DSLK. Chúng ta thường xuyên cần đến duyệt DSLK, chẳng hạn bạn muốn biết một dữ liệu có chứa trong DSLK không, hoặc bạn muốn in ra tất cả các dữ liệu có trong DSLK.

Giải pháp cho vấn đề đặt ra là, chúng ta sử dụng một con trỏ cur trỏ tới thành phần hiện thời (thành phần đang xem xét) của DSLK. Ban đầu con trỏ cur trỏ tới thành phần đầu tiên của DSLK, cur = Head, sau đó ta cho nó lần lượt trỏ tới các thành phần của DSLK, bằng cách gán cur = cur → next, cho tới khi cur chạy qua toàn bộ DSLK, tức là cur = = NULL.

Lược đồ duyệt DSLK được mô tả bởi vòng lặp sau:

Node* cur ;
for (cur = Head; cur ! = NULL; cur = cur → next)
{ các xử lý với dữ liệu trong thành phần được trỏ bởi cur}

Ví dụ. Để in ra tất cả các dữ liệu trong DSLK, bạn có thể viết

for (cur = Head; cur ! = NULL; cur = cur → next)
    cout << cur → data << endl;

Có những xử lý trên DSLK mà để các thao tác được thực hiện dễ dàng, khi duyệt DSLK người ta sử dụng hai con trỏ "chạy" trên DSLK cách nhau một bước, con trỏ cur trỏ tới thành phần hiện thời, con trỏ pre trỏ tới thành phần đứng trước thành phần hiện thời, tức là cur = = pre → next. Ban đầu cur trỏ tới đầu DSLK, cur = Head còn pre = NULL. Sau đó mỗi lần chúng ta cho hai con trỏ tiến lên một bước, tức là đặt pre = cur và cur = cur → next.

Duyệt DSLK với hai con trỏ cur và pre là rất thuận tiện cho những trường hợp mà các xử lý với thành phần hiện thời liên quan tới thành phần đứng trước nó, chẳng hạn khi bạn muốn xen một thành phần mới vào trước thành phần hiện thời, hoặc bạn muốn loại bỏ thành phần hiện thời.

Chúng ta có thể biểu diễn lược đồ duyệt DSLK sử dụng hai con trỏ bởi vòng lặp sau:

```
Node*   cur = Head ;
Node*   pre = NULL;
while (cur! = NULL)
{
  Các xử lý với thành phần được trỏ bởi cur;
  pre = cur;
  cur = cur → next;
}
```

Ví dụ. Giả sử bạn muốn loại khỏi DSLK tất cả các thành phần chứa dữ liệu là value. Để thực hiện được điều đó, chúng ta đi qua DSLK với hai

con trỏ. Khi thành phần hiện thời chứa dữ liệu value, chúng ta loại nó khỏi DSLK, nhưng trong trường hợp này chỉ cho con trỏ cur tiến lên một bước, còn con trỏ Pre đứng nguyên, và thu hồi bộ nhớ của thành phần bị loại. Chúng ta cũng cần chú ý đến thành phần cần loại là đầu hay ở giữa DSLK để có các hành động thích hợp. Khi thành phần hiện thời không chứa dữ liệu value, chúng ta cho cả hai con trỏ cur và Pre tiến lên một bước. Thuật toán loại khỏi DSLK tất cả các thành phần chứa dữ liệu value là như sau:

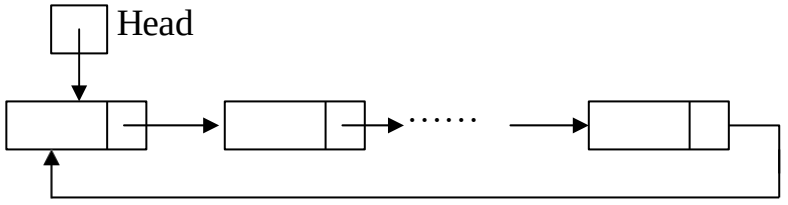
```
Node*   P;
Node*   cur = Head;
Node*   pre = NULL;
while (cur ! = NULL)
{
   if (cur → data = = value)
   if (cur = = Head)
     {
        Head = Head → next;
        P = cur;
        cur = cur → next;
        delete  P;
      }
   else {
            pre → next = cur → next;
            P = cur;
            cur = cur → next;
            delete P;
          }
   else {
            pre = cur;
            cur = cur → next;
          }

} // hết while
```

1.3 CÁC DẠNG DSLK KHÁC.

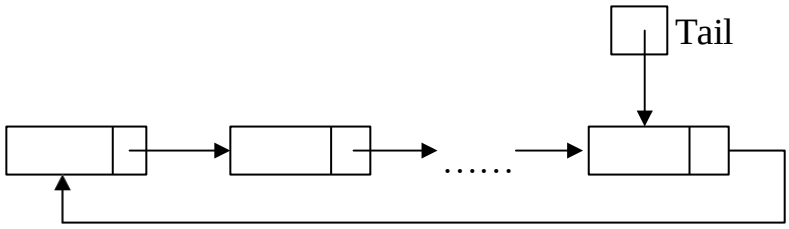
DSLK mà chúng ta đã xét trong mục 5.2 là DSLK đơn, mỗi thành phần của nó chứa một con trỏ trỏ tới thành phần đi sau, thành phần cuối cùng chứa con trỏ NULL. Trong mục này chúng ta sẽ trình bày một số dạng DSLK khác: DSLK vòng tròn, DSLK có đầu giả, DSLK kép. Và như vậy, trong một ứng dụng khi cần sử dụng DSLK, bạn sẽ có nhiều cách lựa chọn, bạn cần chọn dạng DSLK nào phù hợp với ứng dụng của mình.

1.3.1 DSLK vòng tròn

Giả sử trong chương trình bạn sử dụng một DSLK đơn (hình 5.3c) để lưu các dữ liệu. Trong chương trình ngoài các thao tác xen vào dữ liệu mới và loại bỏ các dữ liệu không cần thiết, giả sử bạn thường xuyên phải xử lý các dữ liệu theo trật tự đã lưu trong DSLK từ dữ liệu ở thành phần đầu tiên trong DSLK đến dữ liệu ở thành phần sau cùng, rồi quay lại thành phần đầu tiên và tiếp tục. Từ một thành phần, đi theo con trỏ next, chúng ta truy cập tới dữ liệu ở thành phần tiếp theo, song tới thành phần cuối cùng chúng ta phải cần đến con trỏ Head mới truy cập tới dữ liệu ở thành phần đầu. Trong hoàn cảnh này, sẽ thuận tiện hơn cho lập trình, nếu trong thành phần cuối cùng, ta cho con trỏ next trỏ tới thành phần đầu tiên trong DSLK để tạo thành một DSLK vòng tròn, như được minh hoạ trong hình 5.6a.



(a)



(b)

Hình 5.6. DSLK vòng tròn.

Trong DSLK vòng tròn, mọi thành phần đều bình đẳng, từ một thành phần bất kỳ chúng ta có thể đi qua toàn bộ danh sách. Con trỏ ngoài (có nó ta mới truy cập được DSLK) có thể trỏ tới một thành phần bất kỳ trong DSLK vòng tròn. Tuy nhiên để thuận tiện cho các xử lý, chúng ta vẫn tách biệt ra một thành phần đầu tiên và thành phần cuối cùng như trong DSLK đơn. Nếu chúng ta sử dụng con trỏ ngoài trỏ tới thành phần đầu tiên như trong hình 5.6a, thì để truy cập tới thành phần cuối cùng chúng ta không có cách nào khác là phải đi qua danh sách. Song nếu chúng ta sử dụng một con trỏ ngoài Tail trỏ tới thành phần cuối cùng như hình 5.6b, thì chúng ta có thể truy cập

tới cả thành phần cuối và thành phần đầu tiên, bởi vì con trỏ Tail → next trỏ tới thành phần đầu tiên. Do đó, sau này khi nói tới DSLK vòng tròn ta cần hiểu là DSLK vòng tròn với con trỏ ngoài Tail trỏ tới thành phần cuối cùng, như trong hình 5.6b. Khi DSLK vòng tròn rỗng, giá trị của con trỏ Tail là NULL.

Với DSLK vòng tròn, khi thực hiện các thao tác xen, loại trong các hoàn cảnh đặc biệt, bạn cần lưu ý để khỏi mắc sai sót. Chẳng hạn, từ DSLK vòng tròn rỗng, khi xen vào một thành phần mới được trỏ tới bởi con trỏ Q, bạn cần viết:

```
Tail = Q ;
Tail → next = Tail;
```

Thêm vào dòng lệnh Tail → next = Tail để tạo thành vòng tròn.

Bạn cũng cần chú ý đến phép toán duyệt DSLK. Với DSLK đơn, ta cho con trỏ cur chạy trên DSLK bắt đầu từ cur = Head, rồi thì giá trị của con trỏ cur thay đổi bởi cur = cur → next cho tới khi cur có giá trị NULL. Nhưng với DSLK vòng tròn, giá trị của con trỏ next trong các thành phần không bao giờ là NULL, do đó điều kiện kết thúc vòng lặp cần phải thay đổi, chẳng hạn như sau:

```
if (Tail ! = NULL)
  {
    Node*   first = Tail → next;
    Node*   cur = first;
    do  {
          các xử lý với dữ liệu trong thành phần được trỏ bởi cur;
          cur = cur → next;
          }
    while (cur ! = first) ;
  }
```

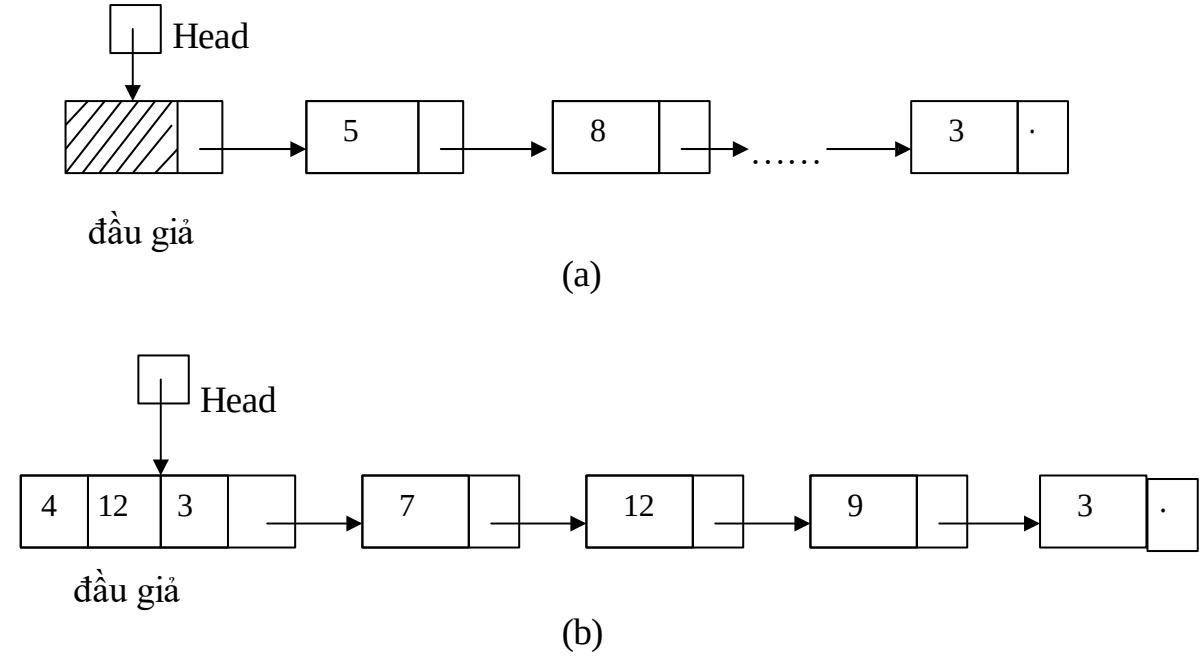
1.3.2 DSLK có đầu giả

Trong DSLK đơn, khi thực hiện phép toán xen, loại bạn cần phải xét riêng trường hợp xen thành phần mới vào đầu DSLK và loại thành phần ở đầu DSLK. Để tránh phải phân biệt các trường hợp đặc biệt này, người ta đưa vào DSLK một thành phần được gọi là đầu giả. Chẳng hạn, DSLK đơn trong hình 5.3c, nếu đưa vào đầu giả chúng ta có DSLK như trong hình 5.7a. Cần chú ý rằng, trong DSLK có đầu giả, các thành phần thực sự của danh sách bắt đầu từ thành phần thứ hai, tức là được trỏ bởi Head → next. DSLK có đầu giả không bao giờ rỗng, vì ít nhất nó cũng chứa đầu giả.

Khi đi qua DSLK có đầu giả sử dụng hai con trỏ: con trỏ trước Pre và con trỏ hiện thời cur, thì ban đầu Pre = Head và cur = Head → next. Để loại

thành phần được trỏ bởi cur ta không cần lưu ý thành phần đó có là đầu DSLK không, trong mọi hoàn cảnh, ta chỉ cần đặt:

Pre → next = cur → next;



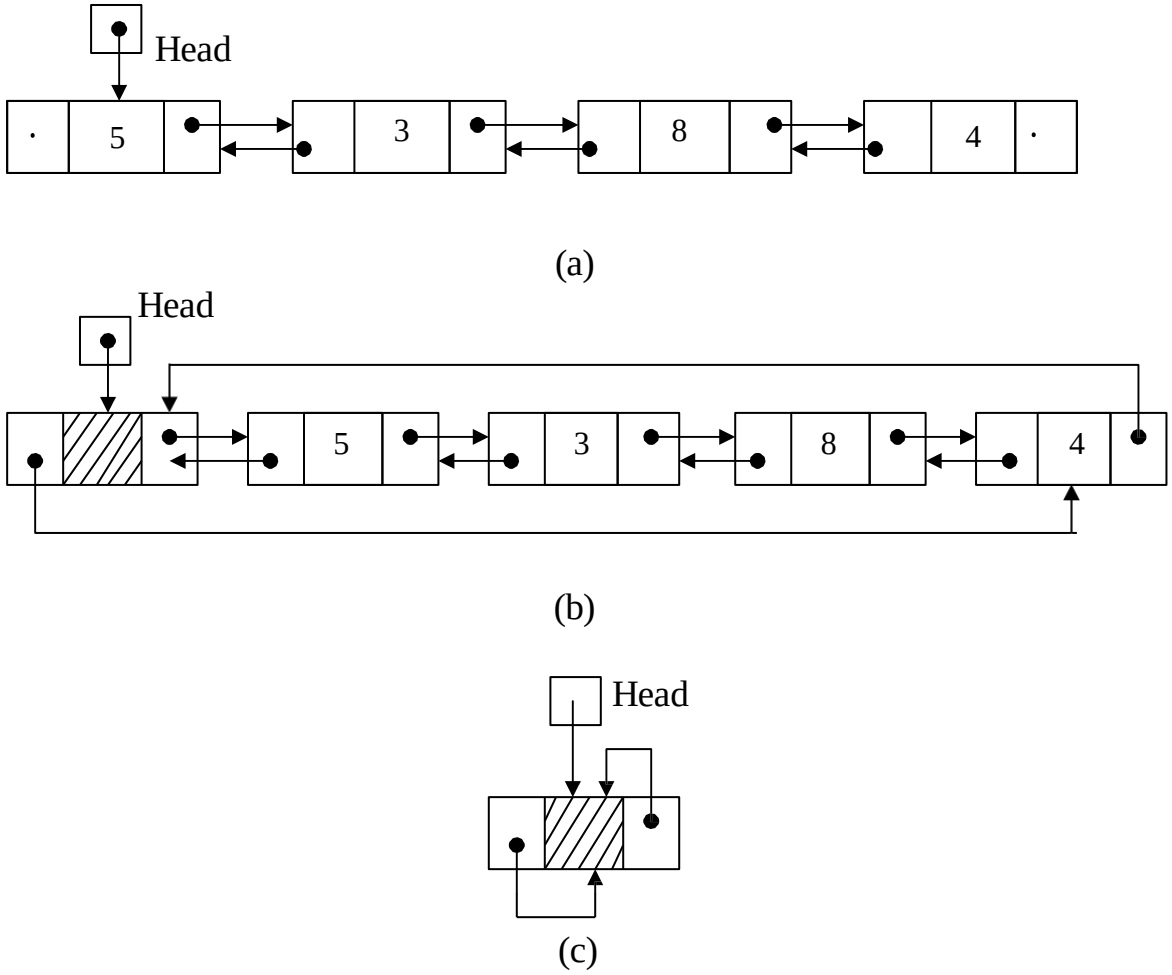
Hình 5.7. (a) DSLK có đầu giả
(b) DSLK với đầu giả lưu thông tin

Đôi khi, người ta sử dụng đầu giả để lưu một số thông tin về danh sách, chẳng hạn như độ dài, giá trị lớn nhất, giá trị nhỏ nhất trong danh sách, như trong hình 5.7b. Nhưng khi đó, đầu giả có thể có kiểu khác với các thành phần thực sự của DSLK. Và do đó các phép toán xen, loại vẫn cần các thao tác riêng cho các trường hợp xen một thành phần mới vào đầu DSLK và loại thành phần đầu của DSLK.

1.3.3 DSLK kép

Trong DSLK đơn, giả sử bạn cần loại một thành phần được định vị bởi con trỏ P, bạn cần phải biết thành phần đứng trước, nếu không biết bạn không có cách nào khác là phải đi từ đầu DSLK. Một hoàn cảnh khác, các xử lý với dữ liệu trong một thành phần lại liên quan tới các dữ liệu trong thành phần đi sau và cả thành phần đi trước. Trong các hoàn cảnh như thế, để thuận lợi người ta thêm vào mỗi thành phần của DSLK một con trỏ mới: con trỏ trước precede, nó trỏ tới thành phần đứng trước. Và khi đó ta có một DSLK kép, như trong hình 5.8a. Mỗi thành phần của DSLK kép chứa một dữ liệu và hai con trỏ: con trỏ next trỏ tới thành phần đi sau, con trỏ precede

trỏ tới thành phần đi trước. Giá trị của con trỏ precede ở thành phần đầu tiên và giá trị của con trỏ next ở thành phần sau cùng là hằng NULL.



Hình 5.8. (a) DSLK kép.
(b) DSLK kép vòng tròn có đầu giả.
(c) DSLK kép rỗng vòng tròn với đầu giả.

Để thuận tiện cho việc thực hiện các phép toán xen, loại trên DSLK kép, người ta thường sử dụng **DSLK kép vòng tròn có đầu giả** như được minh hoạ trong hình 5.8b. Cần lưu ý rằng một DSLK kép rỗng vòng tròn với đầu giả sẽ có dạng như trong hình 5.8c. Bạn có thể khởi tạo ra nó bởi các dòng lệnh:

Head = new Node;
Head → next = Head;
Head → precede = Head;

Với DSLK kép vòng tròn có đầu giả, bạn có thể thực hiện các phép toán xen, loại mà không cần quan tâm tới vị trí đặc biệt. Xen, loại ở vị trí đầu tiên hay cuối cùng cũng giống như ở vị trí bất kỳ khác.

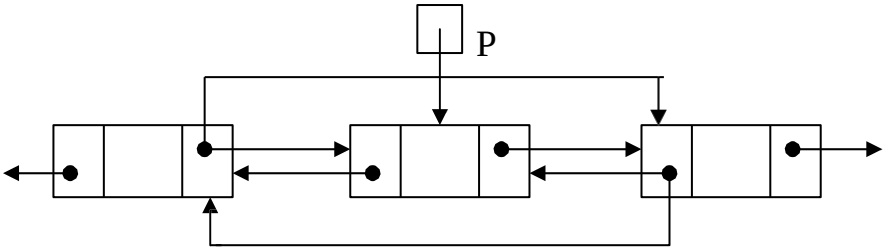
Giả sử bạn cần loại thành phần được trỏ bởi con trỏ P, bạn chỉ cần tiến hành các thao tác được chỉ ra trong hình 5.9a, tức là:

1. Cho con trỏ next của thành phần đi trước P trỏ tới thành phần đi sau P.
2. Cho con trỏ precede của thành phần đi sau P trỏ tới thành phần đi trước P.

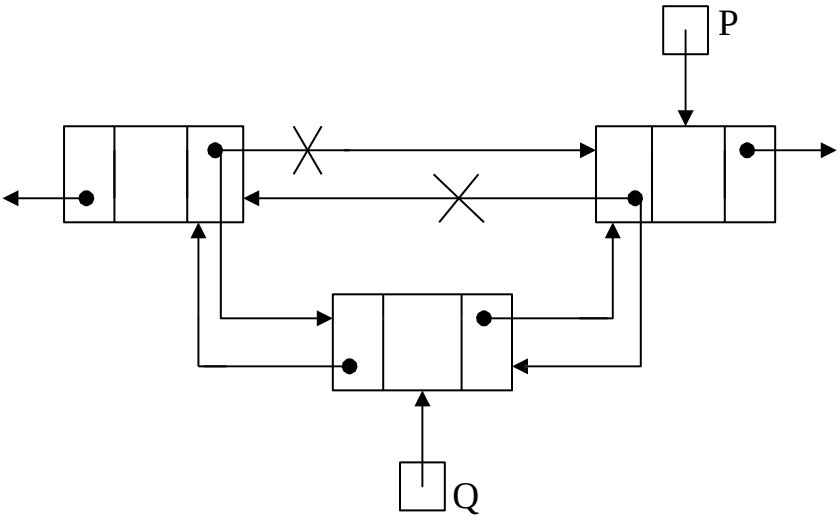
Các thao tác trên được thực hiện bởi các dòng lệnh sau:

P → precede → next = P → next;

P → next → precede = P → precede;



(a)



(b)

Hình 5.9. (a) Loại khỏi DSLK kép thành phần P.
(b) Xen thành phần Q vào trước thành phần P.

Chúng ta có thể xen vào DSLK kép một thành phần mới được trỏ bởi con trỏ Q vào trước thành phần được trỏ bởi con trỏ P bằng các thao tác được chỉ ra trong hình 5.9b

1. Đặt con trỏ next của thành phần mới trỏ tới thành phần P.
2. Đặt con trỏ precede của thành phần mới trỏ tới thành phần đi trước P.
3. Đặt con trỏ next của thành phần đi trước P trỏ tới thành phần mới.
4. Đặt con trỏ precede của thành phần P trỏ tới thành phần mới.

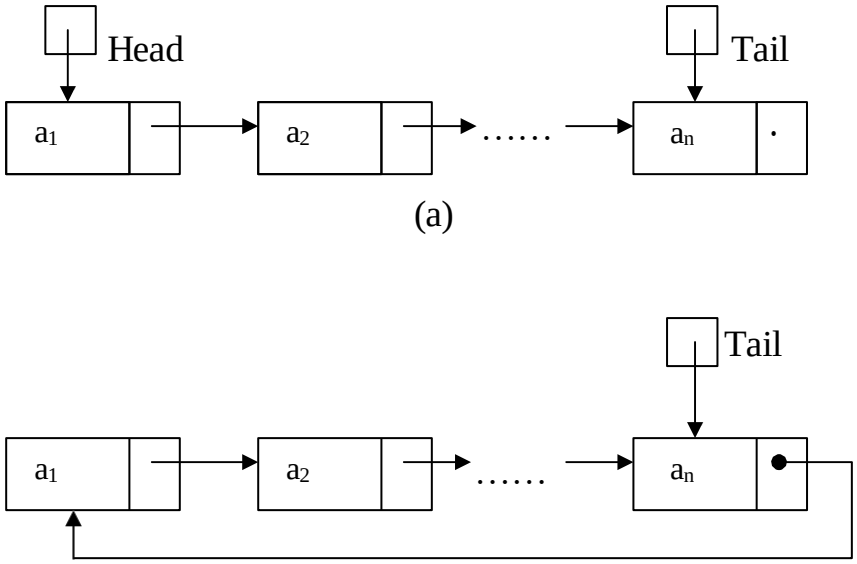
Các dòng lệnh sau thực hiện các hành động trên:

```
Q → next = P;
Q → precede = P → precede;
P → precede → next = Q;
P → precede = Q;
```

Việc xen một thành phần mới vào sau một thành phần đã định vị trong DSLK kép cũng được thực hiện tương tự.

1.4 CÀI ĐẶT DANH SÁCH BỞI DSLK

Trong chương 4, chúng ta đã nghiên cứu phương pháp cài đặt KDLTT danh sách bởi mảng, tức là một danh sách (a_1, a_2, … , a_n) sẽ được lưu ở đoạn đầu của mảng. Mục này sẽ trình bày một cách cài đặt khác: cài đặt bởi DSLK, các phần tử của danh sách sẽ được lần lượt được lưu trong các thành phần của DSLK. Để cho phép toán Append (thêm một phần tử mới vào đuôi danh sách) được thực hiện dễ dàng, chúng ta có thể sử dụng DSLK với hai con trỏ ngoài Head và Tail (hình 5.10a) hoặc DSLK vòng tròn với một con trỏ ngoài Tail (hình 5.10b). Nếu lựa chọn cách thứ hai, tức là cài đặt danh sách bởi DSLK vòng tròn, thì chỉ cần một con trỏ ngoài Tail, ta có thể truy cập tới cả đuôi và đầu (bởi Tail → next) và thuận tiện cho các xử lý mang tính "lần lượt từng thành phần và quay vòng" trên danh sách. Sau đây chúng ta sẽ cài đặt danh sách bởi DSLK với hai con trỏ ngoài Head và Tail như trong hình 5.10a. Việc cài đặt danh sách bởi DSLK vòng tròn với một con trỏ ngoài Tail (hình 5.10b) để lại cho độc giả xem như bài tập.



Hình 5.10. Cài đặt danh sách (a_1, a_2, … , a_n) bởi DSLK

Cần lưu ý rằng, nếu cài đặt DSLK chỉ với một con trỏ ngoài Head, thì khi cần xen vào đuôi danh sách chúng ta phải đi từ đầu lần lượt qua các thành phần mới đạt tới thành phần cuối cùng.

Chúng ta sẽ cài đặt KDLTT danh sách bởi lớp khuôn phụ thuộc tham biến kiểu Item, với Item là kiểu của các phần tử trong danh sách (như chúng ta đã làm trong các mục 4.2, 4.3). Danh sách sẽ được cài đặt bởi ba lớp: lớp các thành phần của DSLK (được đặt tên là lớp LNode), lớp danh sách liên kết (lớp LList) và lớp công cụ lặp (lớp LListIterator). Sau đây chúng ta sẽ lần lượt mô tả ba lớp này.

Lớp LNode sẽ chứa hai thành phần dữ liệu: biến data để lưu dữ liệu có kiểu Item và con trỏ next trỏ tới thành phần đi sau. Lớp này chỉ chứa một hàm kiến tạo để tạo ra một thành phần của DSLK chứa dữ liệu value, và giá trị của con trỏ next là NULL. Mọi thành phần của lớp này đều là private. Tuy nhiên, các lớp LList và LListIterator cần có khả năng truy cập trực tiếp đến các thành phần của lớp LNode. Do đó, chúng ta khai bào các lớp LList và LListIterator là bạn của lớp này. Định nghĩa lớp LNode được cho trong hình 5.11.

```
template <class Item>
class    LList ; // khai báo trước lớp LList.
```

```
template <class Item>
class    LListIterator ;  // khai báo trước.

template <class  Item>
class    LNode
{
     LNode (const Item & value)
     {  data = value ; next = NULL ; }
     Item  data ;
     LNode *  next ;
     friend  class   LList<Item> ;
     friend  class   LListIterator<Item> ;
} ;
```

Hình 5.11. Lớp LNode.

Lớp LList. Lớp này chứa ba thành phần dữ liệu: con trỏ Head trỏ tới đầu DSLK, con trỏ Tail trỏ tới đuôi của DSLK, biến length lưu độ dài của danh sách. Lớp LList chứa các hàm thành phần cũng giống như trong lớp Dlist (xem hình 4.3) với một vài thay đổi nhỏ. Chúng ta cũng khai báo lớp LListIterator là bạn của LList, để nó có thể truy cập trực tiếp tới các thành phần dữ liệu của lớp LList. Định nghĩa là LList được cho trong hình 5.12.

```
template <class Item>
class    LListIterator ;

template <class Item>
class     LList
{
  friend  class  LListIterator <Item> ;
  public :
       LList( )  // khởi tạo danh sách rỗng
       { Head = NULL; Tail = NULL ; length = 0 ; }
       LList (const  LList & L) ; // Hàm kiến tạo copy
       ~ LList( ) ; // Hàm huỷ
       LList & operator = (const  LList & L); // Toán tử gán.
       bool  Empty( ) const
       { return  length = = 0; }
       int  Length( ) const
       { return  length ; }
       void  Insert(const Item & x, int i);
       // xen phần tử x vào vị trí thứ i trong danh sách.
```

```
        void    Append(const Item & x);
        // xen phần tử x vào đuôi danh sách.
        void    Delete(int i);
        // loại phần tử ở vị trí thứ i trong danh sách.
        Item &    Element(int i);
        // trả về phần tử ở vị trí thứ i trong danh sách.
    private :
        LNode <Item> *  Head ;
        LNode <Item> *  Tail ;
        int    length ;
} ;
```

Hình 5.12. Định nghĩa lớp LList.

Bây giờ chúng ta xét sự cài đặt các hàm thành phần của lớp LList. Lớp LList chứa các thành phần dữ liệu được cấp phát động, vì vậy trong lớp này, ngoài hàm kiến tạo mặc định khởi tạo ra danh sách rỗng, được cài đặt inline, chúng ta cần phải đưa vào hàm kiến tạo copy, hàm huỷ và toán tử gán. Sau đây chúng ta xét lần lượt các hàm này.

Hàm kiến tạo copy. Hàm này thực hiện nhiệm vụ tạo ra một DSLK mới với các con trỏ ngoài Head và Tail chứa các dữ liệu như trong DSLK đã cho với con trỏ ngoài L.Head và L.Tail. Nếu DSLK đã cho không rỗng, đầu tiên ta tạo ra DSLK mới gồm một thành phần chứa dữ liệu như trong thành phần đầu tiên của DSLK đã cho, rồi sau đó lần lượt xen vào DSLK mới các thành phần chứa dữ liệu như trong các thành phần tiếp theo của DSLK đã cho. Có thể cài đặt hàm kiến tạo copy như sau:

```
LList <Item> :: LList(const  LList<Item> & L)
{
  if (L.Empty( ))
        { Head = Tail = NULL ; length = 0 ; }
  else {
        Head = new LNode <Item> (L.Head → data);
        Tail = Head ;
        LNode <Item> * P ;
        for (P = L.Head  → next; P! = NULL ; P = P → next)
              Append (P → data) ;
        length = L.length ;
        }
}
```

Hàm huỷ. Hàm này cần thu hồi tất cả bộ nhớ đã cấp phát cho các thành phần của DSLK trả về cho hệ thống và đặt con trỏ Head và Tail là NULL. Muốn vậy, chúng ta thu hồi từng thành phần kể từ thành phần đầu tiên của DSLK. Hàm huỷ được cài đặt như sau:

```
LList <Item> :: ~ LList(  )
{
  if (Head ! = NULL)
    {
      LNode <Item> * P ;
      while (Head ! = NULL)
          {
            P = Head ;
            Head = Head → next ;
            delete  P;
          }
       Tail = NULL ;
        length = 0 ;
     }
 }
```

Toán tử gán. Nhiệm vụ của toán tử gán như sau: ta có một đối tượng (được trỏ bởi con trỏ this) chứa DSLK với các con trỏ ngoài Head và Tail và một đối tượng khác L chứa DSLK với các con trỏ ngoài L.Head và L.Tail, phép gán cần phải làm cho đối tượng *this chứa DSLK là bản sao của DSLK trong đối tượng L. Để thực hiện được điều đó, đầu tiên chúng ta cần làm cho DSLK trong đối tượng *this trở thành rỗng (với các hành động như trong hàm huỷ), sau đó copy DSLK trong đối tượng L (giống như hàm kiến tạo copy). Hàm operator = trả về đối tượng *this. Cài đặt cụ thể hàm toán tử gán để lại cho độc giả, xem như bài tập.

Sau đây chúng ta cài đặt các hàm thực hiện các phép toán trên danh sách.

Hàm Append (xen một phần tử mới x vào đuôi danh sách). Hàm này rất đơn giản, chỉ cần sử dụng thao tác xen một thành phần mới chứa dữ liệu x vào đuôi DSLK

```
void  LList <Item> :: Append (const Item & x)
{
  LNode <Item> * Q = new  LNode <Item> (x) ;
  if (Empty(  ))
        { Head = Tail = Q ; }
  else {
```

```
            Tail → next = Q ;
            Tail = Q ;
        }
  length + + ;
}
```

Các hàm xen phần tử x vào vị trí thứ i trong danh sách, loại phần tử ở vị trí thứ i khỏi danh sách và tìm phần tử ở vị trí thứ i của danh sách đều có một điểm chung là cần phải định vị được thành phần của DSLK chứa phần tử thứ i của danh sách. Chúng ta sẽ sử dụng con trỏ P chạy trên DSLK bắt đầu từ đầu, đi theo con trỏ next trong các thành phần của DSLK, đến vị trí mong muốn thì dừng lại. Một khi con trỏ P trỏ tới thành phần của DSLK chứa phần tử thứ i – 1 của danh sách, thì việc xen vào vị trí thứ i của danh sách phần tử mới x tương đương với việc xen vào DSLK một thành phần mới chứa dữ liệu x sau thành phần được trỏ bởi con trỏ P. Việc loại khỏi danh sách phần tử ở vị trí thứ i có nghĩa là loại khỏi DSLK thành phần đi sau thành phần được trỏ bởi P. Các hàm xen và loại được cài đặt như sau:

Hàm Insert

```
template  <class Item>
void   LList<Item> :: Insert(const  Item & x, int  i)
{
  assert ( i > = 1 && i < = length) ;
  LNode <Item> * Q = new LNode<Item> (x) ;
  if (i = = 1)  // xen vào đầu DSLK.
        {
          Q → next = Head ;
           Head = Q ;
        }
  else  {
           LNode <Item> * P = Head ;
           for (int k = 1 ; k < i – 1 ; k + + )
                    P = P → next ;
           Q → next = P → next ;
           P → next = Q ;
         }

  length + + ;
}
```

Hàm Delete

```
template <class Item>
void  LList <Item> :: Delete (int i)
```

```
{
   assert ( i > = 1 && i <= length) ;
   LNode<Item> * P ;
   If ( k = = 1)  // Loại đầu DSLK
        {
            P = Head ;
            Head = Head → next ;
            Delete P ;
            if (Head = = NULL)  Tail = NULL ;
        }
   else {
            P = Head ;
            for (int k = q ; k < i – 1; k + +)
                  P = P → next ;
            LNode <Item> * Q ;
            Q = P → next ;
            P →next = Q → next ;
            delete Q;
            if ( P → next = = NULL)  Tail = P ;
        }

   length - - ;
}
```

Hàm tìm phần tử ở vị trí thứ i

```
template <class Item>
Item & LList <Item> :: Element(int i)
{
  assert ( i > = 1 && i < = length) ;
  LNode <Item> * P = Head;
  for (int k = 1 ; k < i ; k + +)
       P = P → next ;
  return    P → data ;
}
```

Bây giờ chúng ta cài đặt lớp công cụ lặp LListIterator. Lớp này chứa các hàm thành phần giống như các hàm thành phần trong lớp DlistIterator (xem hình 4.4). Lớp LListIterator chứa một con trỏ hằng LlistPtr trỏ tới đối tượng của lớp LList, ngoài ra để cho các phép toán liên quan tới duyệt DSLK được thực hiện thuận tiện, chúng ta đưa vào lớp LListIterator hai biến con trỏ: con trỏ current trỏ tới thành phần hiện thời, con trỏ pre trỏ tới thành phần đứng trước thành phần hiện thời trong DSLK thuộc đối tượng mà con trỏ LListPtr trỏ tới. Định nghĩa lớp LListIterator được cho trong hình 5.12.

```
template <class  Item>
class      LListIterator
{
  public :
       LListIterator (const LList<Item> & L) // Hàm kiến tạo.
          { LListPtr = & L; current = NULL ; pre = NULL; }
       void   Start( )
          { current = LListPtr →Head; pre = NULL: }
       void   Advance( )
          { assert (current ! = NULL); pre = current; current =
                                        current → next; }
       bool    Valid( )
          {returncurrent ! = NULL; }
       Item &    Current( )
          { assert (current! = NULL); return current → data; }
       void    Add(const Item & x);
       void     Remove( );
  private :
       const  LList<Item> *    LlistPtr;
       LNode<Item> *   current;
       LNode<Item> *   pre ;
}
```

Hình 5.12. Lớp công cụ lặp LListIterator.

Các hàm thành phần của lớp LListIterator đều rất đơn giản và được cài đặt inline, trừ ra hai hàm Add và Remove. Cài đặt hai hàm này cũng chẳng có gì khó khăn cả, chúng ta chỉ cần sử dụng các phép toán xen, loại trên DSLK đã trình bày trong mục 5.2.1. Các hàm này được cài đặt như sau:

Hàm Add

```
template  <class Item>
void  LListIterator<Item> :: Add (const Item & x)
{
  assert (current ! = NULL);
  LNode <Item> * Q = new LNode<Item> (x);
  if  (current = = LListPtr → Head)
  {
      Q → next = LListPtr → Head;
      LListPtr →Head = Q;
```

```
        pre = Q;
    }
    else {
            Q →next = current;
            pre → next = Q;
            pre = Q ;
        }

    LListPtr → length + + ;
}
```

Hàm Remove

```
template <class Item>
void LListIterator<Item> :: Remove( )
{
  assert (current ! = NULL);
  if (current = = LListPtr → Head)
  {
      LListPtr → Head = current →next;
      delete current ;
      current = LListPtr →Head ;
      if (LListPtr →Head = = NULL)
                LListPtr → Tail = NULL;
  }
  else {
          pre → next = current → next;
          delete current;
          current = pre → next;
          if (current = = NULL)
                  LListPtr →Tail = pre;
       }

       LListPtr →length - - ;
}
```

1.5 SO SÁNH HAI PHƯƠNG PHÁP CÀI ĐẶT DANH SÁCH

Một KDLTT có thể cài đặt bởi các phương pháp khác nhau. Trong chương 4, chúng ta đã nghiên cứu cách cài đặt KDLTT danh sách bởi mảng (mảng tĩnh hoặc mảng động). Mục 5.4 đã trình bày cách cài đặt danh sách bởi DSLK. Trong mục này chúng ta sẽ phân tích đánh giá ưu khuyết điểm của mỗi phương pháp. Dựa vào sự phân tích này, bạn có thể đưa ra quyết định lựa chọn cách cài đặt nào cho phù hợp với ứng dụng của mình.

Cài đặt danh sách bởi mảng là cách lựa chọn tự nhiên hợp lý: các phần tử của danh sách lần lượt được lưu trong các thành phần liên tiếp của mảng, kể từ đầu mảng. Giả sử bạn cài đặt danh sách bởi mảng tĩnh có cỡ là MAX. Nếu MAX là số rất lớn, khi danh sách còn ít phần tử, thì cả một không gian nhớ rộng lớn trong mảng không sử dụng đến, và do đó sẽ lãng phí bộ nhớ. Khi danh sách phát triển, tới một lúc nào đó nó có thể có số phần tử vượt quá cỡ của mảng. Đó là hạn chế cơ bản của cách cài đặt danh sách bởi mảng tĩnh. Bây giờ giả sử bạn lưu danh sách trong mảng động. Mỗi khi mảng đầy, bạn có thể cấp phát một mảng động mới có cỡ gấp đôi mảng động cũ. Nhưng khi đó bạn lại phải mất thời gian để sao chép dữ liệu từ mảng cũ sang mảng mới. Sự lãng phí bộ nhớ vẫn xảy ra khi mà danh sách thì ngắn mà cỡ mảng thì lớn.

Trong cách cài đặt danh sách bởi DSLK, các phần tử của danh sách được lưu trong các thành phần của DSLK, các thành phần này được cấp phát động. DSLK có thể móc nối thêm các thành phần mới hoặc loại bỏ các thành phần trả về cho hệ thống mỗi khi cần thiết. Do đó cài đặt danh sách bởi DSLK sẽ tiết kiệm được bộ nhớ.

Một ưu điểm của cài đặt danh sách bởi mảng là ta có thể truy cập trực tiếp tới mỗi phần tử của danh sách. Nếu ta cài đặt danh sách bởi mảng A, thì phần tử thứ i trong danh sách được lưu trong thành phần A[i - 1] của mảng, do đó thời gian truy cập tới phần tử bất kỳ của danh sách là O(1). Vì vậy thời gian của phép toán tìm phần tử thứ i trong danh sách Element(i) là O(1), với i bất kỳ.

Song nếu chúng ta sử dụng DSLK để cài đặt danh sách, thì chúng ta không có cách nào truy cập trực tiếp tới thành phần của DSLK chứa phần tử thứ i của danh sách. Chúng ta phải sử dụng con trỏ P chạy trên DSLK bắt đầu từ đầu, lần lượt qua các thành phần kế tiếp để đạt tới thành phần chứa phần tử thứ i của danh sách. Do vậy, thời gian để thực hiện phép toán tìm phần tử thứ i của danh sách khi danh sách được cài đặt bởi DSLK là phụ thuộc vào i và là O(i).

Nếu danh sách được cài đặt bởi mảng A, thì để xen một phần tử mới vào vị trí thứ i trong danh sách (hoặc loại khỏi danh sách phần tử ở vị trí thứ i), chúng ta phải "đẩy" các phần tử của danh sách chứa trong các thành phần của mảng kể từ A[i] ra phía sau một vị trí (hoặc đẩy lên trước một vị trí các phần tử chứa trong các thành phần kể từ A[i + 1]). Do đó, các phép toán Insert(x,i) và Delete(i) đòi hỏi thời gian O(n – i), trong đó n là độ dài của danh sách.

Mặt khác, nếu cài đặt danh sách bởi DSLK thì để thực hiện phép toán xen, loại ở vị trí thứ i của danh sách, chúng ta lại phải mất thời gian để định vị thành phần của DSLK chứa phần tử thứ i của danh sách (bằng cách cho con trỏ P chạy từ đầu DSLK). Do đó, thời gian thực hiện các phép toán xen, loại là O(i).

Thời gian thực hiện các phép toán danh sách trong hai cách cài đặt bởi mảng và bởi DSLK được cho trong bảng sau, trong bảng này n là độ dài của danh sách.

Phép toán	Danh sách cài đặt bởi mảng	Danh sách cài đăt bởi DSLK
Insert (x, i)	O(n – i)	O(i)
Delete (i)	O(n – i)	O(i)
Append (x)	O(1)	O(1)
Element (i)	O(1)	O(i)
Add (x)	O(n)	O(1)
Remove ( )	O(n)	O(1)
Current ( )	O(1)	O(1)

1.6 CÀI ĐẶT TẬP ĐỘNG BỞI DSLK

Trong mục 4.4, chúng ta đã nghiên cứu phương pháp cài đặt tập động bởi mảng. Ở đó lớp tập động DSet đã được cài đặt bằng cách sử dụng lớp danh sách động Dlist như lớp cơ sỏ private. Đương nhiên chúng ta cũng có thể sử dụng lớp danh sách liên kết LList như lớp cơ sở private để cài đặt lớp DSet. Song cài đặt như thế thì phép toán tìm kiếm trên tập động sẽ đòi hỏi thời gian không phải là O(n) như khi sử dụng lớp DList. Hàm tìm kiếm tuần tự trong lớp DSet sử dụng lớp cơ sở DList (xem mục 4.4) chứa dòng lệnh:

for (int i = 1; i < = length( ); i + +)
if (Element (i). key = = k) return true;

Trong lớp DList, thời gian của phép toán Element(i) là O(1) với mọi i, do đó thời gian của phép toán tìm kiếm là O(n), với n là độ dài của danh sách. Tuy nhiên trong lớp LList, thời gian của Element(i) là O(i), do đó thời gian của phép toán tìm kiếm trên tập động nếu chúng ta cài đặt lớp DSet bằng cách sử dụng lớp LList làm lớp cơ sở private sẽ là $O(n^2)$.

Một cách tiếp cận khác để cài đặt tập động bởi DSLK là chúng ta biểu diễn tập động bởi một List với List là một đối tượng của lớp LList. Lớp DSet sẽ chứa một thành phần dữ liệu là List.

Cũng như trong mục 4.4, chúng ta giả thiết rằng tập động chứa các dữ liệu có kiểu Item, và Item là một cấu trúc chứa một thành phần là khoá (key) với kiểu là keyType. Lớp DSet được định nghĩa như sau:

```
template <class Item>
class    DSet
{
  public:
      void DsetInsert (const Item & x);
      void DsetDelete (keyType k) ;
```

```
        bool  Search (keyType k)  const ;
        Item & Max( )  const ;
        Item & Min( )  const ;
    private :
         LList <Item>    List;
};
```

Chú ý rằng, lớp DSet có các hàm sau đây được tự động thừa hưởng từ lớp LList:

- Hàm kiến tạo mặc định tự động (hàm kiến tạo copy tự động), nó kích hoạt hàm kiến tạo mặc định (hàm copy, tương ứng) của lớp LList để khởi tạo đối tượng List.
- Toán tử gán tự động, nó kích hoạt toán tử gán của lớp LList.
- Hàm huỷ tự động, nó kích hoạt hàm huỷ của lớp LList.

Các phép toán tập động sẽ được cài đặt bằng cách sử dụng các phép toán bộ công cụ lặp để duyệt DSLK List. Chẳng hạn, hàm tìm kiếm được cài đặt như sau:

```
template  <class Item>
bool  DSet<Item> :: Search(keyType k)
{
  LListIterator<Item>    It<List> ;   // Khởi tạo It là đối tượng của lớp
                                      // công cụ lặp, It gắn với List.
  for (It.Start( ) ; It.Valid( ); It.Advance( ))
        if (It.Current( ).key = = k)
                    return     true;
  return     false;
}
```

Hàm loại khỏi tập động dữ liệu với khoá k được cài đặt tương tự: duyệt DSLK List, khi gặp dữ liệu cần loại thì sử dụng hàm Remove( ) trong bộ công cụ lặp.

```
template <class  Item>
void    DSet<Item> :: DsetDelete (keyType    k)
{
  LListIterator<Item>     It(List);
  for  (It.Start( ); It.Valid( ); It.Advance( ))
        if (It.Current( ).key = = k)
            { It.Remove( ) ; break); }
}
```

Hàm xen một dữ liệu mới x vào tập động được thực hiện bằng cách gọi hàm Append trong lớp LList để xen x vào đuôi DSLK.

```
template  <class Item>
void  DSet<Item> :: DsetInsert(const Item & x)
{
  if (! Search (x.key))
        List.Append(x);
}
```

Các phép toán tìm phần tử có khoá lớn nhất (hàm Max), tìm phần tử có khoá nhỏ nhất (hàm Min) để lại cho độc giả , xem như bài tập.

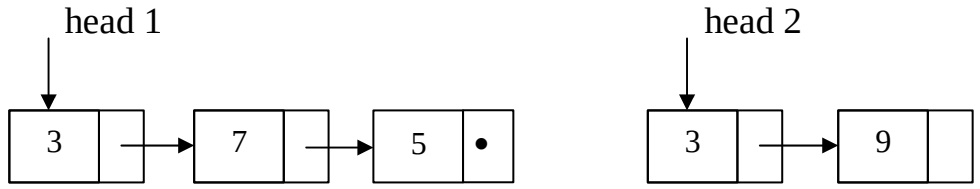
Các phép toán trong bộ công cụ lặp chỉ cần thời gian O(1), do đó với cách cài đặt lớp DSet như trên, tất cả các phép toán trong tập động chỉ đòi hỏi thời gian O(n), với n là số dữ liệu trong tập động.

Chú ý. Khi cài đặt tập động bởi DSLK chúng ta chỉ có thể tìm kiếm tuần tự. Cho dù các dữ liệu của tập động được lưu trong DSLK lần lượt theo giá trị khoá tăng dần, chúng ta cũng không thể áp dụng kỹ thuật tìm kiếm nhị phân, lý do đơn giản là trong DSLK chúng ta không thể truy cập trực tiếp tới thành phần ở giữa DSLK.

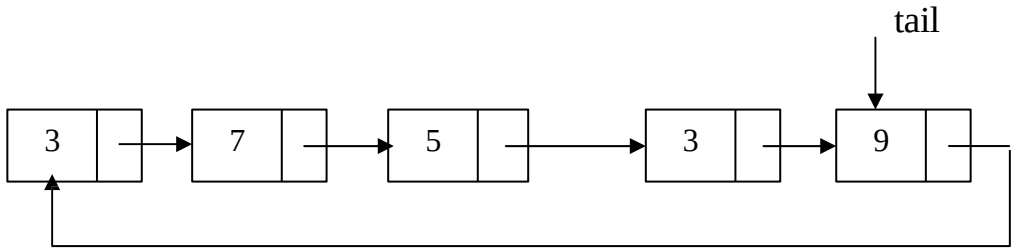
BÀI TẬP.

1. Cho DSLK đơn với con trỏ ngoài head trỏ tới đầu DSLK, và P là con trỏ trỏ tới một thành phần của DSLK đó. Hãy viết ra các mẫu hàm và cài đặt các hàm thực hiện các nhiệm vụ sau:
   a. Xen thành phần mới chứa dữ liệu d vào trước P.
   b. Loại thành phần P.
   c. In ra tất cả các dữ liệu trong DSLK.
   d. Loại khỏi DSLK tất cả các thành phần chứa dữ liệu d.

2. Cho hai DSLK, hãy viết hàm kết nối hai DSLK đó thành một DSLK vòng tròn, chẳng hạn với hai DSLK:



ta nhận được DSLK vòng tròn sau:



Chú ý rằng, một trong hoặc cả hai DSLK đã cho có thể rỗng.

3. Cho DSLK vòng tròn với con trỏ ngoài tail trỏ tới đuôi DSLK. Hãy cài đặt các hàm sau:
   a. Xen thành phần mới chứa dữ liệu d vào đuôi DSLK.
   b. Xen thành phần mới chứa dữ liệu d vào đầu DSLK.
   c. Loại thành phần ở đầu DSLK.

4. Hãy cài đặt các hàm kiến tạo copy, hàm huỷ, toán tử gán trong lớp Dlist bởi các hàm đệ quy.

5. Hãy cài đặt lớp Llist, trong đó danh sách được cài đặt bởi DSLK vòng tròn với một con trỏ ngoài tail.