

4

鏈結串列

在第三章中我們介紹了「陣列」這種以循序 (sequential) 對應關係在記憶體中存放資料的資料結構—邏輯上相鄰的資料，在記憶體中的實體 (physical) 配置，亦彼此相鄰。於是透過註標即可輕易地定址出每一個陣列中的元素。陣列元素與註標之間唯一的對應關係，的確使陣列元素的引用非常方便；這與陣列在記憶體中的配置為循序連續者息息相關、密不可分。倘若陣列存放在的元素經常增加或刪除，為維持此「邏輯與實體間連續配置」的關係，陣列內的元素可能須隨資料的新增（在連續配置的陣列中挪出空位）或刪除（在連續配置的陣列中將刪除的空位補齊）而頻繁地搬動其它資料 (data movement)，反而降低了陣列的使用效率。況且所用陣列的大小需要事前宣告，宣告若過大會造成記憶體浪費、反之可能不敷使用；有沒有需要多少即使用多少的可能？

「鏈結串列」(linked list) 則屬於非循序的儲存結構，邏輯上相鄰的二項資料，在記憶體中的實際配置不需要彼此相鄰¹。如果資料的引用並不強調實體配置中的連續性，或不必要賦予每一元素唯一的註標，鏈結串列可能是比較好的選擇。在資料新增或刪除頻繁的應用中，利用鏈結串列掌握所新增或刪除資料與其前後資料彼此關係，比起使用陣列，確實可以獲得較好的效率。利用鏈結串列，更有運用動態配置 (dynamic allocation) 來靈活使用記憶體空間的彈性—需要的時候提出空間的需求，使用完畢就歸還給系統。

在本章中我們將介紹鏈結串列的基本觀念與各項運用。

4.1 串列與鏈結串列

「串列」(list) 的抽象觀念是一組相同資料型態元素的有序集合。與陣列不



1 像 C, C++, PASCAL 等結構化語言，皆有位址或指標 (pointer) 的資料型態，可藉以建構鏈結串列。此時資料的邏輯關係即靠指標指定，資料是利用指標鏈結串接而產生關係，鏈結串列之名因此而生。

同處在於：它不必要存在唯一的註標與每一元素相互對應²。範例 4-1 舉了幾個串列的例子。

範例 4-1

- (a) 十二生肖所造成的串列爲：（鼠、牛、虎、.....豬）；
- (b) 小於 1000 的質數由小至大所形成的串列爲（共 168 個）：(2, 3, 5, 7, ..., 991, 997)；
- (a) apple、peach、strawberry、mango、cherry，五種水果依其英文字母順序排序後，形成的串列爲：（apple、cherry、mango、peach、strawberry）。

範例 4-1 的串列，可用陣列來儲存表示之，如圖 4-1 所示，其中 A、B 和 C 皆爲文字陣列：

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
鼠	牛	虎	兔	龍	蛇	馬	羊	猴	雞	狗	豬

(a)

B[0]	B[1]	B[2]	B[3]	...	B[166]	B[167]
2	3	5	7	...	991	997

(b)

C[0]	C[1]	C[2]	C[3]	C[4]
apple	cherry	mango	peach	strawberry

(c)

圖 4-1 以陣列的循序關係存放有序的串列資料



- 2 用陣列實作串列當然可以；本章則強調用直接以鏈結串列來實作串列。用鏈結串列也可模擬實作陣列，但是就少了註標所帶來的便利性。

4-4 資料結構與演算法

範例所用的陣列是以循序的關係，將串列中的元素逐一存放至陣列中。既然採用了陣列，任一元素自然有唯一的註標，這樣的循序擺放在資料已經過排序時，將方便搜尋演算法的運作³。但是若串列本身經常有異動的情況（範例 4-1 (a)、(b)大約不會異動，但 4-1 (c) 會因進貨銷售而常有異動），則維護排序的關係可能得付出頗大的代價，請見範例 4-2。

範例 4-2

以範例 4-1 (c) 為例，若該串列為當天水果商家出售的水果類別，且依水果英文字母順序排序，而 apple 已售完，陣列 C 將改變為如圖 4-2 (a) 所示；又若 banana 到貨，陣列 C 又將變為如圖 4-2 (b) 所示。

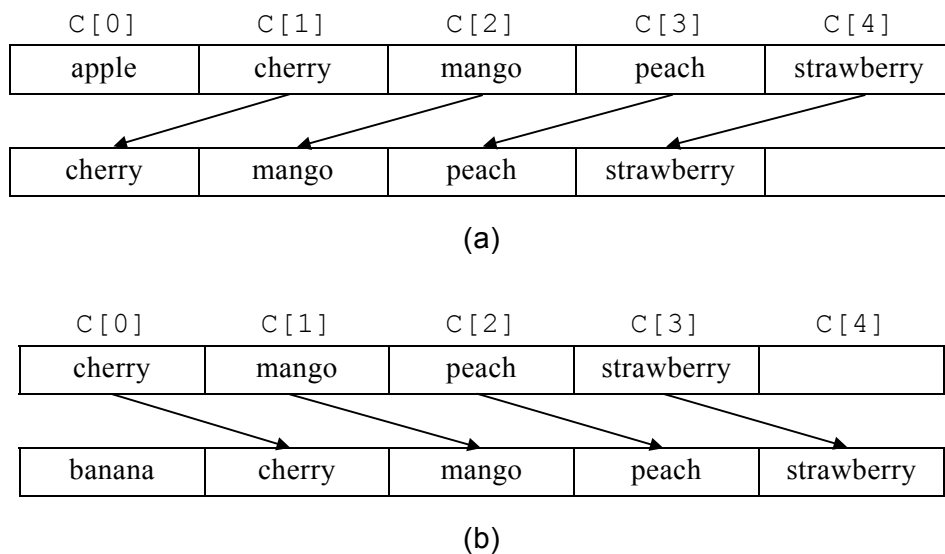


圖 4-2 以陣列存放串列資料時可能的資料搬移狀況

~~~~~

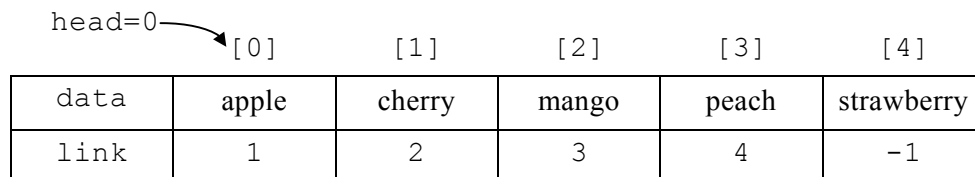
3 利用二元搜尋法(binary search)，其時間複雜度為  $O(\log n)$ ，是很有效率的。

範例 4-2 顯示欲維持資料的排序關係，資料異動造成資料搬移的程度可能相當嚴重；若資料有  $O(n)$  筆，資料搬移的次數可能在一次新增或刪除中即高達  $O(n)$ 。由此可知在資料量大時，用陣列的循序關係存放有順序關係的串列資料，雖然在搜尋時，有不錯的效率；然而在資料異動頻繁時，將非常沒有效率。

倘若仍以陣列來存放這些有順序關係的資料，但放棄註標與元素的循序對應關係—改用記住下一筆資料所在註標—形成非循序的結構，情況是否可以改善？請見範例 4-3。

### 範例 4-3

依然以範例 4-1 (c) 為例，我們使用了二個陣列 data 和 link 來存放水果的資訊。data 內存放資料，link 內則存放該資料的下一筆資料註標位置；另外再用一個額外的註標 head，記著第一項資料在 data 陣列中的位置。如圖 4-3 (a)。



|      | [0]   | [1]    | [2]   | [3]   | [4]        |
|------|-------|--------|-------|-------|------------|
| data | apple | cherry | mango | peach | strawberry |
| link | 1     | 2      | 3     | 4     | -1         |

圖 4-3 (a) 利用 link 陣列掌握 data 陣列的資料順序

刪除 apple 後的結果應如圖 4-3 (b) 所示，因刪除者為第一筆資料，遂將 head 改為其下一筆資料所在註標：

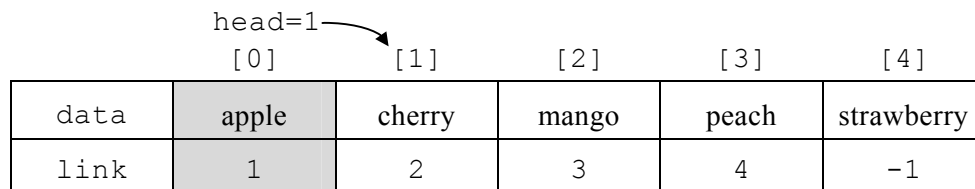
```
head = link[head];
```

即可。圖中 data[0] 和 link[0] 以灰色網底顯示表示已遭刪除<sup>4</sup>。

~~~~~

4 data[0] 和 link[0] 的內容不必刻意清除之。

4-6 資料結構與演算法



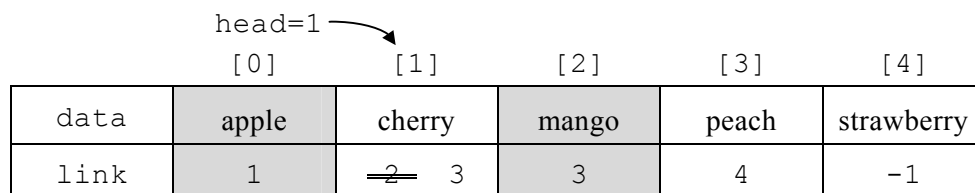
	[0]	[1]	[2]	[3]	[4]
data	apple	cherry	mango	peach	strawberry
link	1	2	3	4	-1

圖 4-3 (b) 刪除 apple

若下次資料異動為：刪除 mango，因刪除者為第 2 筆資料，遂將其前一筆資料的 link 改為

```
link[1] = link[2];
```

即可。結果應如圖 4-3 (c) 所示：



	[0]	[1]	[2]	[3]	[4]
data	apple	cherry	mango	peach	strawberry
link	1	2 3	3	4	-1

圖 4-3 (c) 刪除 mango

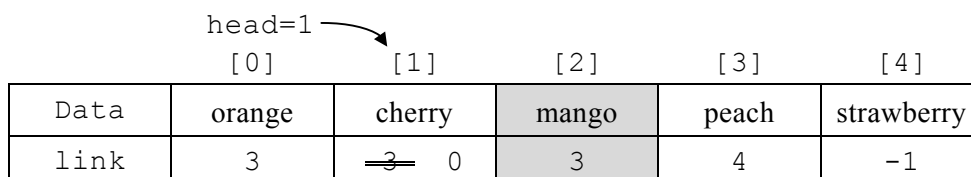
若下次資料異動為：加入 orange，因 data[0] 有空位，可放 orange；其順序在應在 cherry (data[1]) 之後，應執行：

```
data[0] = "orange";
```

```
link[0] = link[1]; // orange 之後應為 cherry 之後
```

```
link[1] = 0;      // cherry 之後即為 orange
```

(注意：此三行程式的執行順序不得更動) 結果應如圖 4-3 (d) 所示：



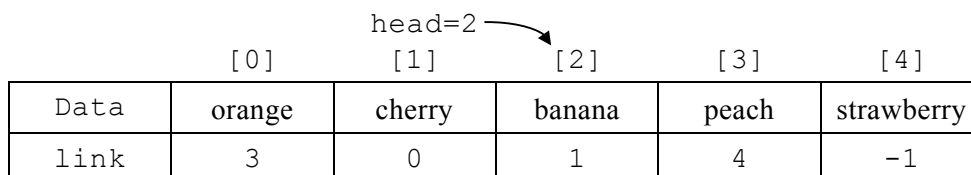
	[0]	[1]	[2]	[3]	[4]
Data	orange	cherry	mango	peach	strawberry
link	3	3 0	3	4	-1

如圖 4-3 (d) 加入 orange

若下次資料異動為：加入 banana，因下一空位為 data[2]，遂放入；其順序應在 cherry (data[1]、原來的 head) 之前，是為新的 head，應執行：

```
data[2] = "banana";
link[2] = head; // banana 之後即為 cherry
head = 2;
```

結果應如圖 4-3 (e) 所示：



	[0]	[1]	[2]	[3]	[4]
Data	orange	cherry	banana	peach	strawberry
link	3	0	1	4	-1

如圖 4-3 (e) 加入 banana

範例 4-3 的程式碼整理，留做習題請各位完成。

從範例 4-3 可得知，非循序的陣列結構，因資料異動造成的資料搬移，比起範例 4-2 使用的循序陣列結構要減輕許多（有空位即可新增、刪除者則標示其位置是空的；吾人設法記住空位—仍採用非循序的陣列結構—即可）。兩者雖然用的皆是陣列，但在範例 4-3 中，資料的順序是由 link 陣列的整數內容來決定；利用 link 來串接資料的概念，在資料異動頻繁，又想維持資料順序關係時，不失為一利器；此亦即鏈結串列的構想。

這裡的「鏈結」指的就是下一項資料所在的資訊，也就是 link 所扮演的角色——它指明了下一項資料所在的資訊⁵。在實用上鏈結可以是任何一種想串接維繫的邏輯關係。然用陣列得在宣告時即決定其大小，宣告過大會浪費空間，宣告不足又不敷使用；下一節的動態配置技巧則可克服陣列大小在宣告與實用時的迷失，鏈結串列的實施也因而有明確的措施。

4.2 動態配置之鏈結串列

鏈結串列中的每一個元素，包括至少二種類型的資訊：(1) 內含的元素，以及 (2) 下一項元素所在位置的資訊。我們先將這二項資料型態不同的資訊宣告成一個「自訂結構」⁶，此結構即成為鏈結串列元素的基本資料型態，可用以定義出符合這種結構的實體空間——謂之節點 (node)。請見下面的例子。

範例 4-4

我們可以範例 4-3 的水果英文單字為例，做下面的宣告⁷



- 5 在下一節中可看到鏈結是由指標或位址來構成，比起陣列 link 的使用更具一般性。
- 6 在 C 語言中，不同型態的多種資料，若須集成一邏輯整體，常用結構 (struct) 指令組織成一使用者自訂的資料型態 (user-defined data type)。
- 7 在 C++ 中，可用類別 (class) 定義此節點的型態：

```
class FruitType
{   char name[maxchar];
    FruitType *link;
};
```


程式 4-1 結構的宣告

```

1  #define maxchar 20
2  struct FruitType
3  {   char name[maxchar];
4      struct FruitType *link;
5  };
6  struct FruitType *head;

```

在第 2~5 行中我們宣告了新的資料型態 `struct FruitType`，它包括了一組長度為 20 的字元 `name`，和指向 `struct FruitType` 此型態的指標（位址）`*link`。它成為鏈結串列的基本型態，可用以定義符合這類資料型態的實體空間——稱為節點；其邏輯圖示如圖 4-4 所示。

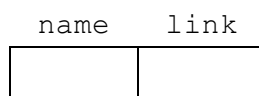


圖 4-4 以 `struct FruitType` 為資料型態的節點

而圖 4-5 則顯示範例 4-3 中 5 個水果名依字母順序排序的鏈結串列：



圖 4-5 以 `struct FruitType` 為資料型態的節點所構成的鏈結串列

請注意圖 4-5 中 `head` 指向第一個鏈結串列的節點，最後一個節點的 `link` 乃為 `NULL`⁸，圖示如上（圖 4-5 右側貌似接地的符號即表示其為空節點 `NULL`）。

接下來我們探討建立鏈結串列所須的各項動作：

~~~~~

<sup>8</sup> 在 C 中，`NULL` 為一個預設常數，值為 0，在此稱為空指標或虛無指標。

**範例 4-5**

程式 4-2 完成下列鏈結串列的建立：

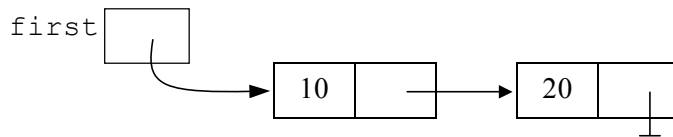


圖 4-6 包含兩個節點的鏈結串列

**程式 4-2 建立含有兩個節點的鏈結串列**

```

1  struct node
2  {   int data;
3      struct node *next;
4  };
5  struct node *first;
6  void construct()
7  {   first = (struct node *) malloc (sizeof(struct node)); //9
8      first->data = 10;
9      first->next = (struct node *) malloc (sizeof(struct node));
10     first->next->data = 20;
11     first->next->next = NULL;
12 }
```

在第 1~4 行中我們宣告了新的資料型態 `struct node`，它包括了一個整數 `data`，和指向 `struct node` 此型態節點的指標 `*next`（或者說成 `next` 此變數的內容是個 `struct node *`，為指向 `struct node` 這種結構的指標；`*next` 即為該 `struct node`）兩個欄位。注意：1~4 行僅是宣告，並沒有實體空間在記憶體中產生。



<sup>9</sup> 在 C++ 中可寫成 `first = new node;` 以下所用 `malloc` 者，皆適用此寫法。

第 5 行則用 `struct node *first;` 定義出變數 `first` (已有實體空間於記憶體中產生)，其為指向 `struct node` 型態的指標<sup>10</sup> (也可解讀為 `*first` 即為 `first` 所指向的 `struct node`)；目前尚未設定任何內容。

在第 7 行中用

```
first = (struct node *) malloc (sizeof(struct node));
```

其中 `malloc`<sup>11</sup> 可在記憶體中動態配置出 `struct node` 此結構的實體空間，大小為 `sizeof(struct node)`<sup>12</sup>，由 `first` 存放此節點的指標位址。第 8 行以

```
first->data = 10;
```

將 10 設定至 `first` 所指向 `struct node` 中的 `data` 欄位<sup>13</sup>；第 9 行則用

```
first->next=(structnode*) malloc (sizeof(structnode));
```

來動態配置出另一 `struct node` 的實體節點，其指標位址直接放在 `first->next` 中。執行完 7~11 行後，即可完成圖 4-6 的鏈結串列。



10 此指標變數 `first` 所占用記憶體的大小依使用電腦而定，64(32)-bit 個人電腦為 64 (32) bits 亦即 8 (4) bytes。一般編譯器會初始其值為空指標 `NULL`。

11 `malloc` 是在執行 (execution) 時才真正在記憶體中配置出該空間，與一般變數宣告在編譯 (compile) 時即配置空間，有很大的不同；前者謂之「動態配置」(dynamic allocation)，後者謂之「靜態配置」(static allocation)。`malloc` 回傳所產生空間第一個 byte 的住址，型態為 `void*`，在此例中轉型為 `(structnode*)` 指定給 `first`。

12 承 10，`sizeof(structnode)` 在 64(32)-bit 電腦中為 12 (8)；因為 `struct node` 包含兩個欄位：`int` 和 `struct node*`前者有 4 bytes，後者有 8 (4) bytes。

13 在 C 中「指標->欄位」是既定語法，亦可用「(\*指標).欄位」(即 `struct` 所定義型態中的欄位)；於是 `first->data` 與 `(*first).data` 是同義的。

下面將討論鏈結串列的一些基本運算，包括：新增資料至節點中和插入節點至鏈結串列中（4.2.1 節）、刪除鏈結串列內的節點（4.2.2 節）、和尋找鏈結串列中的資料（4.2.3 節）。

#### 4.2.1 新增資料至節點並插入該節點至串列中

在本小節中我們希望將整數資料 `element` 存入新建的節點 `x`，並且將 `x` 插入至串列中，`p` 所指向節點（簡稱節點 `p`）的後面，如圖 4-7。

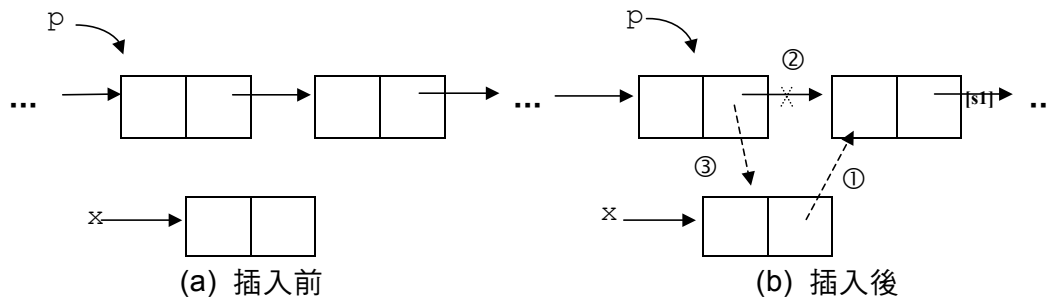


圖 4-7 將新節點插入串列中

- ① 將 `x->next` 改為 `q`（原來的 `p->next`）；
- ② 取消 `p->next`；
- ③ 將 `p->next` 改為 `x`。

上面所列的步驟，將說明新增資料進入串列的必要動作：我們沿用程式 4-2 中的節點宣告，從圖 4-7 (a) 與 (b) 可看出必要的動作包括 3 個鏈結的異動。

事實上 ② 和 ③ 可同時完成（如程式 4-3 所示）。我們還得考慮 `p` 為 `NULL` 的情形，此時 `element` 將成為串列的第一項資料，那麼新增後的鏈結串列將如圖 4-8。

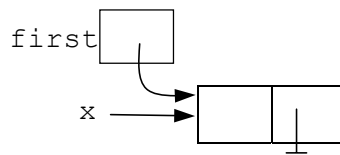


圖 4-8 若 `p` 為 `NULL`，新節點插入串列後成為第一個節點

下面的程序可完成新增節點插入串列的動作：

**程式 4-3** 新增資料 element 至節點 x 中並插入節點 x 至串列中 p 節點之後

```

1  struct node
2  {   int data;
3      struct node *next;
4  };
5  struct node *first;
6  void InsertAfter(struct node *p, int element)
7  {   struct node *x;
8      x =(struct node *) malloc (sizeof(struct node));
9      x->data = element;
10     if (p==NULL)
11     {   first = x;
12         x->next = NULL;
13         return;
14     }
15     x->next = p->next;
16     p->next = x;
17 }
```

有了程式 4-3 的定義，可呼叫

```
InsertAfter(p, element);
```

將整數資料 element 存入新建的節點 x 中，並且插入至串列中 p 所指節點的後面。

1~4 行為節點的宣告；第 5 行定義 first 為指向 struct node 的指標（或 \*first 即為 struct node）。第 6 行定義程序 InsertAfter 的傳入參數（p 為指向 struct node 的指標，element 為整數），毋庸回傳任何值（void）；7~9 行執行節點 x 的動態產生並將 element 值放入 x->data 中；10~14 行處理原串列為空的情況；15 行為步驟 ①、16 行為步驟 ②和 ③合併的程式碼。注意：步驟 ③不得在 ①之前執行，否則會導致錯誤。

### 4.2.2 刪除鏈結串列中的節點

在本節中我們希望刪除鏈結串列中， $p$  所指向節點的下一個節點，並傳回該節點的資料，如圖 4-9 所示。

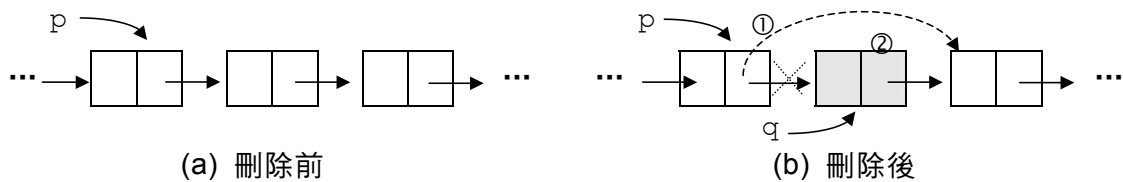


圖 4-9 刪除節點  $p$  的下一個節點

其中包括的步驟為：

- ①  $p \rightarrow \text{next}$  (令之為  $q$ ) 改成  $q \rightarrow \text{next}$ ；
- ② 將當初動態配置而得的節點  $q$  (原來的  $p \rightarrow \text{next}$ ) 還給系統<sup>14</sup>。

下面的程序可完成這個需求。

#### 程式 4-4 刪除串列中 $p$ 節點之後的節點

```

1  int DeleteAfter(struct node *p)
2  {   struct node *q;
3      if (p==NULL || p->next==NULL) return -1;
4      q = p->next;
5      p->next = q->next;
6      int item = q->data;
7      free(q);
8      return item;
9  }
```



14 動態配置的概念乃在克服陣列使用得先行宣告的限制，亦即免除空間濫用或不足的窘境；所以動態配置的空間若已不再使用，應還給系統，由系統統籌動態配置空間的掌握與管理。當然動態配置的空間也不可能無限制取用，依然受限於電腦記憶體管理系統能掌握的總量。

程式 4-4 中第 1 行定義程序 DeleteAfter 將依據所傳入的 struct node \*p 執行刪除其後節點並回傳該節點資料（或刪除未果）的任務：第 3 行先判斷指標 p 自己是空指標或後面沒有節點則回傳-1 以示刪除未果；1~4 行為節點的宣告；4~5 行為步驟 ①；第 8 行為步驟 ②—用 free(q) 指令（若用 new 生成空間者請用 delete 刪除之）把節點 q 的實體空間還給系統；第 6/8 行將節點 q 內的整數資料傳回。

### 4.2.3 尋找鏈結串列中的資料

鏈結串列不像陣列一般，擁有註標可以方便地定出各個元素所在的位置。它通常只有開始節點的指標，而每個元素僅有下一元素的指標；所以尋找資料對如此靠指標維護串列順序關係的結構，沒有像二元搜尋法般好用的演算法（第五章會用二搜尋樹解決快速搜尋的問題）。下面介紹線性搜尋的方法，在鏈結串列中尋找資料：

#### 程式 4-5 尋找鏈結串列中的資料

```

1  struct node * SearchData(int target)
2  {   struct node * p;
3      p = first;
4      while ((p!=NULL) && (p->data!=target)) p = p->next;
5      // for (p=first; (!p && p->data!=target); p=p->next);
6      return p;
7  }
```

第 1 行中傳入 SearchData 的整數 target 為欲尋找的資料，傳出的結果是 p：可能是 target 所在節點的指標（找到了）或 NULL（表示沒找到）；所以在第 1 行行首有回傳型態 struct node\* 的宣告；第 3 行及第 4 行即自 first 節點起，逐一尋找節點內的 data 是否與 target 相同，直到找到了或所有節點都看完了為止；第 9 行利用 while 判斷是否串列尚未搜尋結束（p!=NULL），而

且尚未找到 target ( $p \rightarrow data \neq target$ )，若兩者皆成立，則檢查下一節點 ( $p = p \rightarrow next$ )。注意： $p \neq NULL$  必須寫在  $p \rightarrow data \neq target$  之前！否則在  $p$  已是  $NULL$  時  $p \rightarrow data$  的先行引用會使執行因  $NULL \rightarrow data$  已無意義而中斷<sup>15</sup>；而先檢測  $p \neq NULL$ （於  $\&\&$  之前，其為  $false$  時  $\&\&$  之後不會再執行）即可避開這類引用錯誤。

第 5 行則是用 `for` 迴圈來取代 `while` 迴圈，其中  $!p$  與  $p \neq NULL$  是同義的。`for` 有迴圈控制變數初始化、繼續執行條件、控制變數的調整三項設定，通常可以使 `while` 迴圈有簡潔的程式寫法。善用 `for` 指令可以體驗陣列和串列程式寫作上巧妙的對應關係<sup>16</sup>。

這個搜尋的動作實為新增節點插入串列、刪除節點等動作的基礎。我們前面定義的程序 `InsertAfter`、`DeleteAfter`，皆須先決定在串列中欲插入或刪除節點的位置，此時 `SearchData` 即為不可缺少的程序。請注意：`InsertAfter(p, element)` 可在節點指標  $p$  之後新增存放 `element` 的節點、而 `DeleteAfter(p)` 則可刪除指標  $p$  的後一個節點！若欲搜尋 `target` 並於其之前新增（或直接刪除其所在節點），應適度修改 `Searchdata`，使之傳回找到節點的前一節點指標，方得供 `InsertAfter` 和 `DeleteAfter` 所用。請參考程式 4-6，其中  $p, q$  皆為指向 `struct node` 的指標，由第 3 行可知  $q$  持續指在  $p$  的前一個節點，在找到 `target` ( $p \rightarrow data == target$ ) 時，呼叫 `DeleteAfter(q)` 即可刪除/回傳 `target` 並回收其節點空間；若找不到 `target` 則回傳 `-1`。



- 15 常用的編譯器會稱這類錯誤為「存取錯誤」(access violation)—不知住址為何、取用不可取得的位址。撰寫指標的程式務必要小心這類錯誤。
- 16 試比較 `for (i=0; i<n; i++);` 與 `for (p=first; p; p=p->next);` 可分別順利走訪陣列與鏈結串列；前者用註標，後者用指標。C 語言語法設計上的巧思，略見一斑。



**程式 4-6 尋找並刪除鏈結串列中的資料**

```
1  int SearchDeleteData(int target)
2  {   struct node *q, *p;
3      for (q=NULL, p=first; (!p && p->data!=target);
           q=p, p=p->next);
4      if (!p) return DeleteAfter(q);
5      return -1;
6  }
```

由以上的介紹，各位不難發現在鏈結串列中，搜尋某特定資料在最差情況下，得花  $O(n)$  的時間，其中  $n$  指的是鏈結串列的節點個數。一旦插入或刪除的節點位置已知，則執行插入或刪除只需  $O(1)$  的時間。

在堆疊或佇列的應用中，少有搜尋特定資料的需求，而新增或刪除的運算對堆疊而言，皆在堆疊頂端處進行；對佇列而言，則新增在底端，刪除在頂端進行。於是利用動態配置的鏈結串列，即成為實作堆疊或佇列的最佳利器——優勢在於新增或刪除節點的時間複雜度為  $O(1)$ ，且不必事先宣告所需堆疊或佇列的大小。在 4.3 節之後，即為各位介紹利用動態配置的鏈結串列，如何實作堆疊和佇列的基本運算。

下面先看一些鏈結串列的進階運算，包括：在串列最後新增節點（4.2.4 節）、反向連接一串列（4.2.5 節）、串接兩個鏈接串列（4.2.6 節）。

#### 4.2.4 在串列最後新增節點

「在串列最後新增節點」，可能會是個經常發生的需求，試想先不管該依何欄位排序，資料庫內的資料新增，大多先加至目前所有資料的最後面。修改 4.2.3 節介紹的「程式 4-5 尋找鏈結中的資料」，可找到鏈結串列的最後一個節點，找到後再增加節點至其後，即可達成在串列最後新增節點的要求。如果這樣的需求經常發生，每次得逐一掃描串列的所有節點，以確定最後一個節點所

在位置，才能新增節點在其之後，將使新增節點的動作變得頻繁，非常沒有效率。此時我們可以多加一個指標（如 last），指向鏈結串列的最後一個節點。如圖 4-10 如示，那麼大量新增節點資料於串列最後的需求，即可有效率地解決。

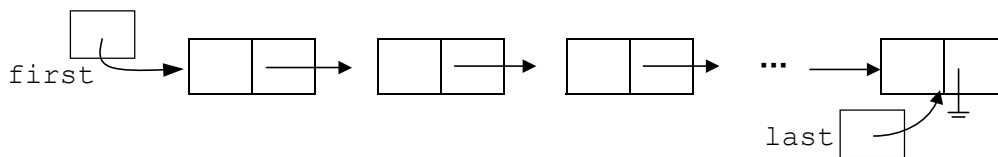


圖 4-10 多加指標 last 指向鏈結串列的最後一個節點

請看下面的程式碼：

**程式 4-7 在鏈結串列最後新增節點，last 指在串列的最後節點**

```

1  struct Node
2  {   int data;
3      struct Node * link;
4  };
5  struct Node * first, * last;
6  void AttachDataToList(int element)
7  {   struct Node * p;
8      p = (struct Node *) malloc (sizeof (struct Node));
9      p->data = element; p->link = NULL;
10     if (first == NULL) first = last = p ;
11     else
12     {   last->link = p;
13         last = p;
14     }
15 }
```

第 1~4 行為節點的宣告 (struct Node)；第 5 行定義 first、last 為指向 struct Node 的指標。第 6 行的程序宣告 AttachDataToList 接受傳入的整數 element—其為欲置入新節點的資料，將其加入串列使成最後節

點；第 8~9 行，動態產生了一個新的節點  $p$ ，並將  $element$  放入其  $data$  欄位， $NULL$  放入其  $link$  欄位。第 10 行檢查  $first$  是否為  $NULL$ ，若是，表示原來的串列是空的； $first$  和  $last$  應設成  $p$ ，因為它正是第一個也是最後一個節點；若串列不是空的，則第 12~13 行可容易地透過  $last$  定位列串列的最後節點，直接將  $p$  串接在  $last$  之後（12 行），再將  $last$  指向  $p$ （13 行）——因為節點  $p$  成為新的最後節點。

#### 4.2.5 反向串接一串列

「將一鏈結串列的指標反向連接」是一個十分有趣的串列運算，這個運算將使原來串列內資料的順序完全對調。我們利用圖 4-11 說明我們的初步構想。

如果我們知道一個節點  $b$  之前的節點  $a$ （如圖 4-11 (a)），就可以將  $b \rightarrow link$  指向  $a$  即達反向的目的；但是原來的  $b \rightarrow link$ （圖 4-11 (b) 的灰網節點）若不記下， $b \rightarrow link = a$ ；會使其將無從考察；為使反向的操作正常無誤，我們得至少保持 3 個指標，若  $t$ 、 $s$ 、 $r$  指標分別指在如圖 4-11 (c) 的位置，則把  $s \rightarrow link$  改為  $t$ ，反向串接節點  $s$  的目的即可達成，節點  $r$  也不致於找不到，上述的動作又可重複執行。

圖 4-11 (d) 顯示當  $s$  為第一個節點時，它之前是沒有節點的，反向時其  $s \rightarrow link$  應為  $NULL$ （它本是第一節點，反向後則成最後節點），此時  $r$  應為  $NULL$ 。圖 4-11 (e) 則展示了最後一個節點執行反向時， $t$ 、 $s$ 、 $r$  應有的狀態，而反向後  $first$  應更正為  $s$ （它原是最後節點，反向後則成為第一節點）。由圖 14 (c)-(e) 可知指標  $t$ 、 $s$ 、 $r$  每個回合順勢平移（見圖 14 (f)），反向  $s \rightarrow link$ ，各節點皆反向後更新  $first$  為原最後節點，即可完成所求。請注意：圖 4-11 中僅標示  $first$  的實體空間，然  $a$ 、 $b$ 、 $t$ 、 $s$ 、 $r$  指標變數皆有其實體空間，在此省略其實體空間與箭頭符號。

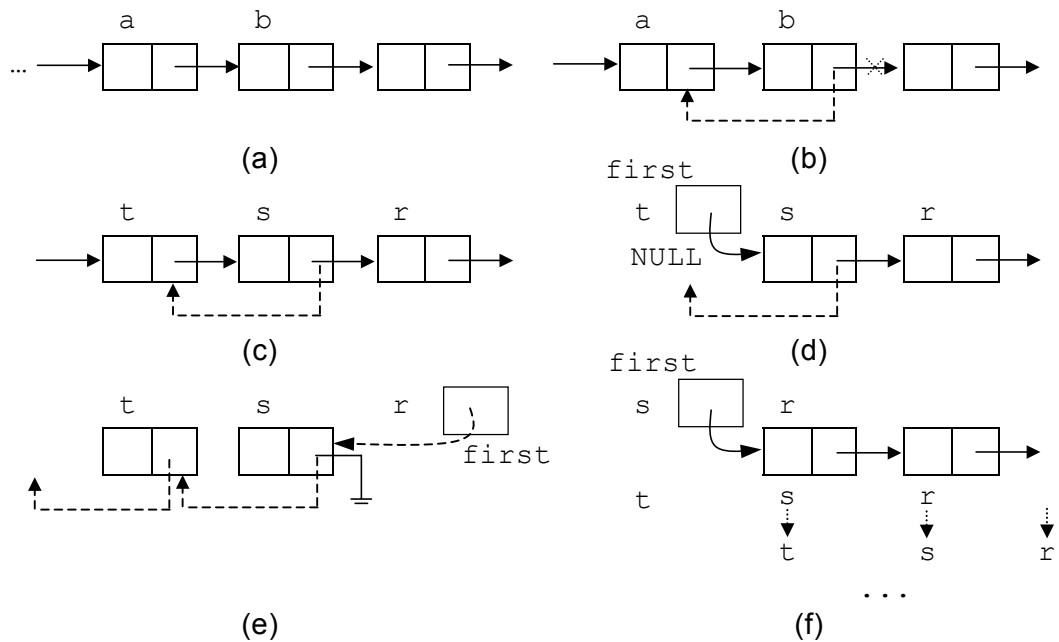


圖 4-11 反向連接串列

詳細的程式碼整理在程式 4-8。

#### 程式 4-8 反向串接一串列

```

1  struct Node
2  {   int data;
3      struct Node *link;
4  };
5  struct Node *first;
6  void Invert()
7  {   struct Node *r, *s, *t;
8      r = first;
9      s = NULL;
10     while (r != NULL)
11     {   t = s;
12         s = r;
13         r = r->link;
14         s->link = t;
15     }
16     first = s;
17 }

```

第 11~13 行即為前述順勢平移指標  $t$ 、 $s$ 、 $r$  的處理，14 行則反向節點  $s$  ( $s \rightarrow \text{link} = t$ )；第 10 行控制流程唯在最後一個節點處理完成才可結束，屆時  $r$  必為 NULL！第 8、9 行則初始化指標  $r$ 、 $s$ （見圖 14 (f)）——其在進入 while 迴圈後即順勢平移成預想的狀態（比較圖 14 (d)、(f)）。總之 8~16 行的程式碼保證了  $r$  之前的節點為  $s$ ，而  $s$  之前的節點是  $t$ ；如圖 4-11 (c) 所示。這個問題對各位掌握指標的技巧很有幫助，請多加思索。

#### 4.2.6 串接兩個鏈結串列

假設  $a\_first$  和  $b\_first$  分別是串列  $A = (a_1, a_2, \dots, a_m)$  和串列  $B = (b_1, b_2, \dots, b_n)$  第一個節點的指標，其中  $m, n \geq 0$ 。希望得到  $A$  和  $B$  的串接串列  $C = (a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n)$ ，且串列  $C$  的第一個節點指標為  $first$ ，如圖 4-12 如示；我們可用程式 4-9 得到此串接串列。

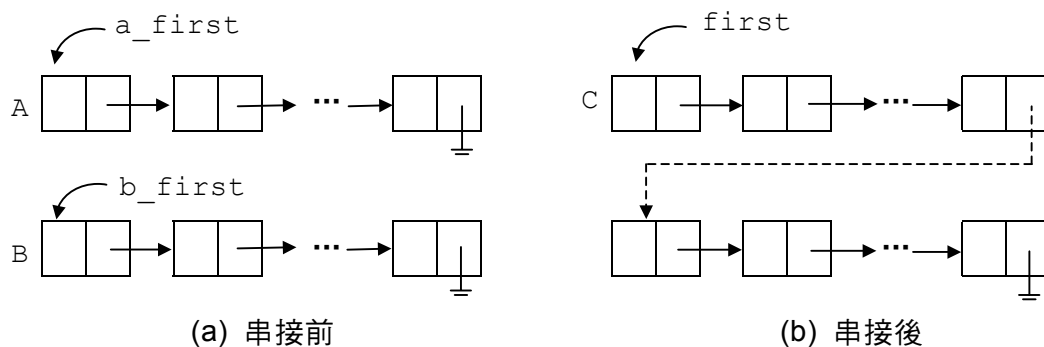


圖 4-12 串接兩個鏈結串列

程式 4-9 串接兩個鏈結串列

```

1  struct Node
2  {   int data;
3      struct Node *link;
4  };
5  struct Node *a_first, *b_first, *first;

```

```

6  struct Node *Concatenate(struct Node *a_first,
                           struct Node *b_first)
7  {   struct Node *p ;
8      if (a_first == NULL)
9          return b_first ;
10     else
11     {   for (p = a_first; p->link != NULL; p = p->link);
12         p->link = b_first ;
13         return a_first ;
14     }
15 }

```

程序 Concatenate 的傳入參數為兩個 struct Node \* 指標 a\_first 和 b\_first，傳出結果亦為 struct Node \* 指標。第 8 行檢查 a\_first 是否為 NULL，若是，表示串列 A 根本是空的，直接傳回 b\_first 即為串列 A 和 B 串接後第一個節點的指標；否則利用第 11 行的 for 迴圈，找到串列 A 的最後一個節點 p，將 p->link 指向 b\_first（第 12 行），則串列 B 即串接於 A 之後。在第 14 行處傳回 a\_first 即可。呼叫程式可以：

```
first = Concatenate(a_first, b_first);
```

得到以 first 指標（在第 5 行宣告為全域變數）為首的鏈結串列——實則將 a\_first 指標為首的串列結尾設定成 b\_first，兩者即可串接在一起（若 a\_first 為空指標，則 b\_first 指標為首的串列即為串接結果）。

### 4.3 鏈結堆疊

利用陣列來實作堆疊的確方便，但是陣列在宣告時即得定義大小，宣告太大形成空間的浪費，宣告太小又怕不敷使用。改用動態配置的鏈結堆疊，即可解決使用陣列造成的缺點。圖 4-13 顯示「鏈結堆疊」(linked stack) 的邏輯圖示：（請注意鏈結指標的方向：由頂端節點——由 top 所指向——往後指！）

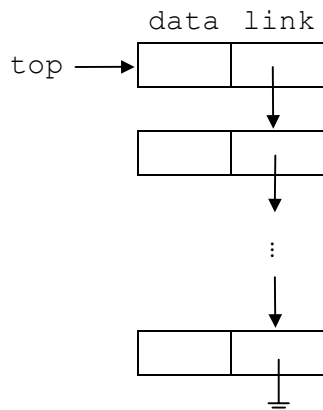


圖 4-13 鏈結堆疊的邏輯圖示

至於鏈結堆疊的宣告及 push, pop 程序的定義可見下面的程示碼：

#### 程式 4-10 鏈結堆疊

```

1  struct StackNode
2  {   int data;
3      struct StackNode *link;
4  };
5  struct StackNode *top;
6  struct StackNode *NewNode(int element)
7  {   struct StackNode *p;
8      p = (struct StackNode *) malloc (sizeof(struct StackNode));
9      if (p == null){ MemoryNotEnough();return(-1); }
10     p->data = element;
11     p->link = NULL;
12     return p;
13 }
14 void PushStack(int element)
15 {   struct StackNode *x ;
16     x = NewNode(element);
17     if (top == NULL) top = x;
18     else
19     {   x->link = top;

```

#### 4-24 資料結構與演算法

```
20         top = x;
21     }
22 }
23 int PopStack( )
24 {   struct StackNode *p;
25     if (top == NULL);
26     {   StackEmpty( ); return -1;
27     }
28     else
29     {   p = top ;
30         top = top->link;
31         data = p->data;
32         free(p);
33         return(data);
34     }
35 }
```

在上面的程式碼中第 1~5 行宣告了 struct StackNode 節點，供堆疊使用。節點中包含整數 data 欄位和 link 指標—指向型態為 struct StackNode 的節點（第 3 行；或解釋為 \*link 即為 link 所指向的 StackNode 節點）。

第 6~13 行為新增一節點的程序 NewNode，傳入的參數為欲放入新節點的整數資料 element；第 8 行動態配置了此新節點 p，型態為 struct StackNode；第 10 行把 element 置入 p->data 中，第 11 行將 p->link 先填上 NULL；第 12 行傳回此動態新增出來的節點 p。此 NewNode 程序可供任何需要新增節點存放資料的需求者呼叫使用。

第 14~22 行的程序 PushStack 定義了鏈結堆疊的 push 運算。傳入的是欲 push 入堆疊的元素 element；首先透過呼叫

```
16     x = NewNode(element);
```

動態配置一個新的 struct StackNode 節點 x（第 16 行，且 x->data 和



$x \rightarrow \text{link}$  也在此設定完成)；此  $x$  節點應成為新的堆疊頂端節點，如果堆疊頂端指標  $\text{top}$  為  $\text{NULL}$ ，則設定  $\text{top} = x$  (第 17 行) 即可，請見圖 4-14 (a)；否則  $x \rightarrow \text{link}$  應指向  $\text{top}$  (第 19 行)，且  $x$  應為新的  $\text{top}$  (第 20 行)，請見圖 4-14 (b)。第 17~21 行亦可簡潔改寫如下：

```
17  if (top) x->link=top;    // (top) 與 (top!=NULL) 同義
18  top = x;
```

或

```
17  x->link=top;    // 若 top 為 NULL 也不致出錯
18  top = x;
```

第 23~35 行的程序 `PopStack` 定義了鏈結堆疊的 `pop` 運算，我們先行檢查堆疊內是否已有元素，若沒有元素，無法執行 `pop`，應傳回 -1 (第 25~27 行)；否則在 29~34 行會將頂端元素的 `data` 欄位值傳回，並將頂端指標指向  $\text{top} \rightarrow \text{link}$  (第 30 行)，請記得將動態配置出的節點  $p$  歸還給系統 (第 32 行)。請見圖 4-14 (c)。

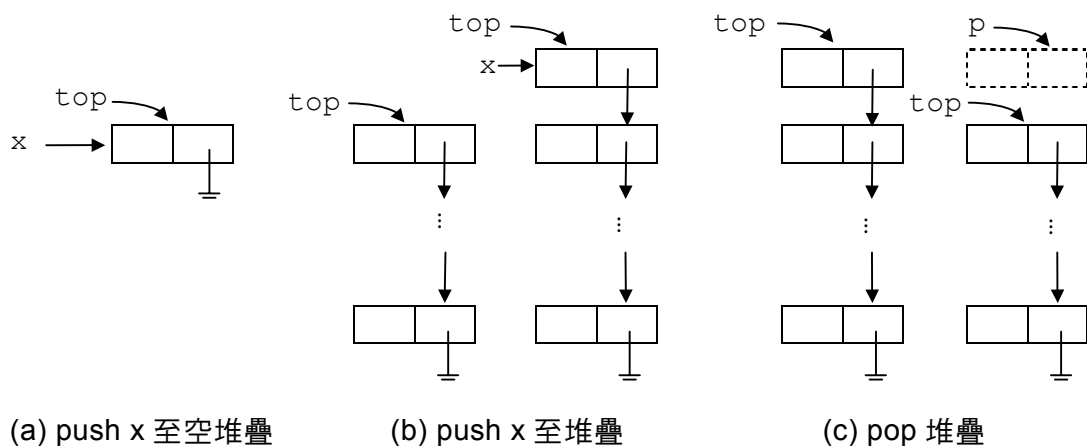


圖 4-14 鏈結堆疊執行 `push` 和 `pop` 的過程

再次提醒：堆疊節點中 `link` 指標的方向—由頂端往底端指去！請各位改由底端往頂端指去，看是否可正常運作？

## 4.4 鏈結佇列

在節 3.4 中我們曾討論過用陣列實作佇列的技巧，環狀佇列的概念也在 3.5 節中提出。然而用陣列實作佇列（或環狀佇列）依然會面對宣告太大或不足的窘境，利用動態配置實作鏈結佇列，將可克服使用陣列的缺點。圖 4-15 描繪了一個鏈結佇列：

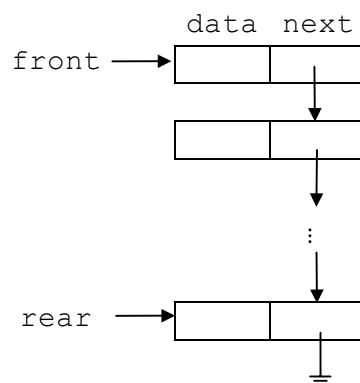


圖 4-15 鏈結佇列的邏輯圖示

節點的新增由後端（rear 指標處）加入，節點的刪除則在前端（front 指標處）進行。倘若鏈結佇列節點中的 next 指標如圖 4-16 所示，此時要自前端刪除節點資料則顯得麻煩了。

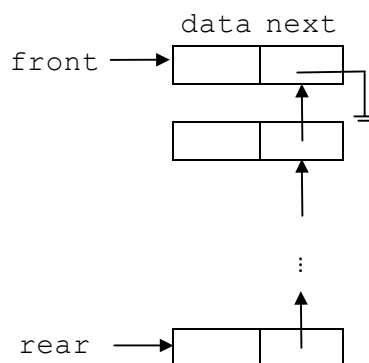


圖 4-16 不佳的鏈結佇列設計

下面是鏈結佇列必要的宣告、和進行增加、刪除佇列節點時的程序：

程式 4-11 鏈結佇列

```
1  struct QueueNode
2  {   char data;
3      struct QueueNode *next;
4  };
5  struct QueueNode *front, *rear;
6  struct QueueNode *NewQNode(char element)
7  {   struct QueueNode *p;
8      p = (struct QueueNode *) malloc (sizeof(struct QueueNode));
9      p->data = element;
10     P->next = NULL;
11     return p;
12 }
13 void AddQueue(char element)
14 {   struct QueueNode *p;
15     p = NewQNode(element);
16     if (rear == NULL)
17         front = p;
18     else
19         rear->next = p;
20     rear = p;
21 }
22 char DeleteQueue()
23 {   struct QueueNode *p;
24     char element;
25     if (front == NULL)
26     {   QueueEmpty();
27         return '#';
28     }
29     else
30     {   p = front;
31         front = front->next;
32         element = p->data;
33         free(p);
34         return element;
35     }
36 }
```

程式 4-11 中第 1~5 行宣告了 struct QueueNode 節點，它包含了一個存放字元的欄位 data 及一個指向 struct QueueNode 節點的指標欄位。

第 6~12 行定義了新增節點的程序 NewQNode(element)，它將傳入的字元 char 存放在新建的 struct QueueNode 節點—由 p 指向—的 data 欄位內，並且令 p->next 為 NULL，傳回 p 供呼叫者使用。

第 13~21 行定義了新增節點至鏈結佇列的程序 AddQueue，傳入的字元 element 會被存放在新增的 struct QueueNode 節點 p 中（第 15 行），如果原本為空佇列，則條件 rear==NULL 成立（第 16 行），p 成為佇列中的前端（和後端）節點，第 17 行的 front = p 和第 20 行的 rear = p 應要執行，如圖 4-17 (a) 所示；否則新增的 p 成為新的後端節點，應執行第 19 行的 rear->next = p 和第 20 行的 rear = p，如圖 4-17 (b)。

第 22~36 行則定義了刪除鏈結佇列的程序 DeleteQueue，應刪除 front 所指向的節點，並傳回其所含的字母。第 25~28 行是空佇列的處理步驟，傳回字元 '#' 供呼叫程序辨別（第 27 行）。第 30~35 行則為非空佇列的處理步驟，front 中所存的字母以 element 傳回（34 行），front 則改為 front->next 所指向的節點（31 行），注意舊的頂端節點 p 應還回給系統（33 行）。請見圖 4-17 (c)。

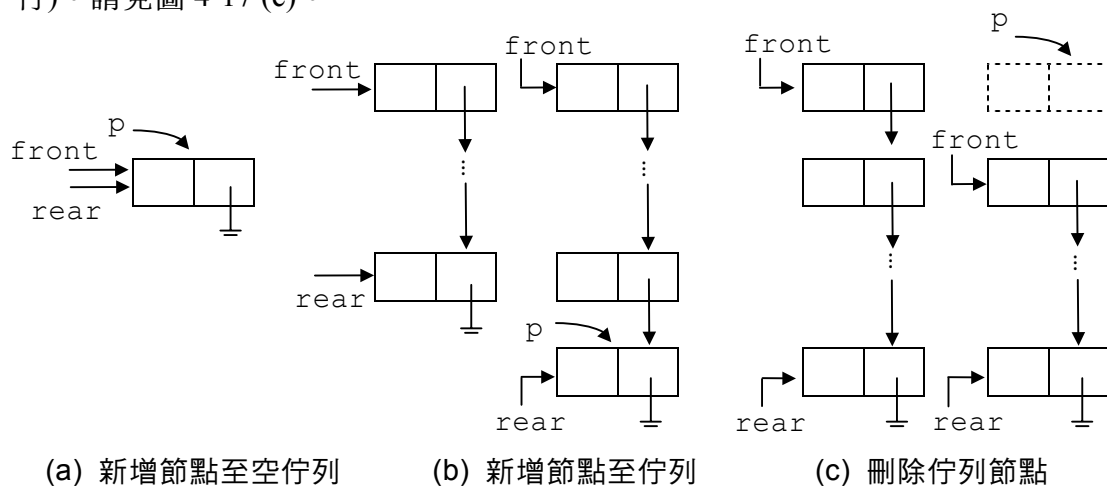


圖 4-17 鏈結佇列執行新增和刪除的過程

## 4.5 環狀串列

前面所提的鏈結串列皆為單向的鏈結串列，如圖 4-18 所示。

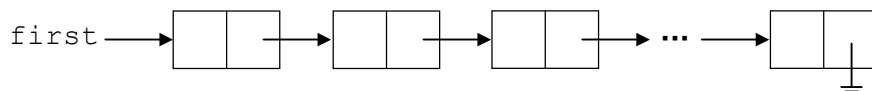


圖 4-18 單向鏈結串列

若單向鏈結串列最後一個節點的指標指向串列的第一個節點（原為空指標），即形成「環狀串列」(circular list)，如圖 4-19。

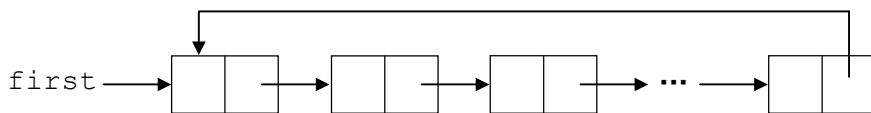


圖 4-19 環狀鏈結串列

環狀串列比起單向串列的好處在於：從串列中的任何一個節點開始皆可將此串列走訪一次。可是這種串列在遇到「在串列最前加入新節點」的需求時，倒顯得棘手。程式 4-12 即為其程式碼：

**程式 4-12** 在環狀串列中新增最前節點

```

1  struct node
2  {   int data;
3      struct node *next;
4  };
5  struct node *first;
6  void InsertFirst(struct node *x)
7  {   struct node *p;
8      if (first == NULL)
9      {   first = x;
10         first->next = first;
11     }
12     else

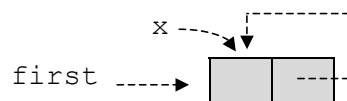
```

```

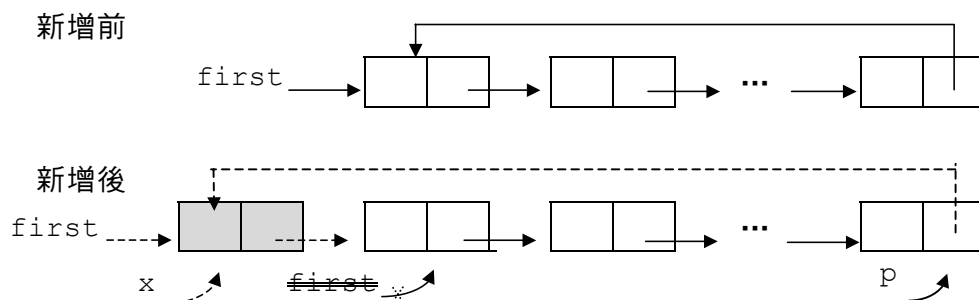
13      {  p = first;
14          while (p->next!=first) p = p->next;
15          p->next = x;
16          x->next = first;
17          first = x;
18      }
19  }

```

程式 4-12 中第 8~11 行得處理原為空環狀串列的情況，如圖 4-20 (a)；而 12~18 行實去找到串列的最後一個節點  $p$  ( $p \rightarrow \text{next}$  指標會指向  $\text{first}$ ，第 13~14 行)，然後在  $p$  之後串加  $x$ ，而  $x$  之後串聯  $\text{first}$  (15~16 行)，在 17 行中將  $\text{first}$  指在  $x$  上，請見圖 4-20 (b)。很顯然地，這個需求得走訪所有節點，時間複雜度為  $O(n)$ ，其中  $n$  為環狀串列的節點數。



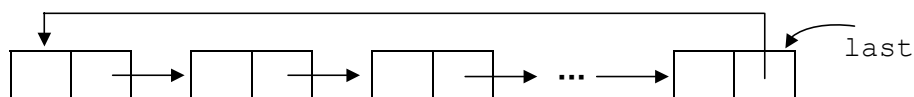
(a) 新增至空環狀串列



(b) 新增至一般環狀串列

圖 4-20 在環狀串列中新增最前節點（虛線為更動的指標）

我們可將之改為如圖 4-21 的結構來改善。

圖 4-21 指在環狀串列結尾的  $\text{last}$  指標

此時程式 4-12 中第 6~18 行的 InsertFirst 程序可改成程式 4-13：

**程式 4-13** 在環狀串列中（有 last 指標指向結尾節點）新增最前節點

```

1 void InsertFirst(struct node *x)
2 {   if (last == NULL )
3     {   last = x;
4         last->next = last ;
5     }
6     else
7     {   x->next = last->next;
8         last->next = x;
9     }
10 }
```

當原環狀串列為空時，新加入的  $x$  節點成為第一個串列節點，第 3~5 行執行必要的設定；第 6~9 行則直接在  $last$  之後串加  $x$ ，使成為串列之最節點，如圖 2-22。注意：第 7 行和第 8 行的執行順序不得顛倒！

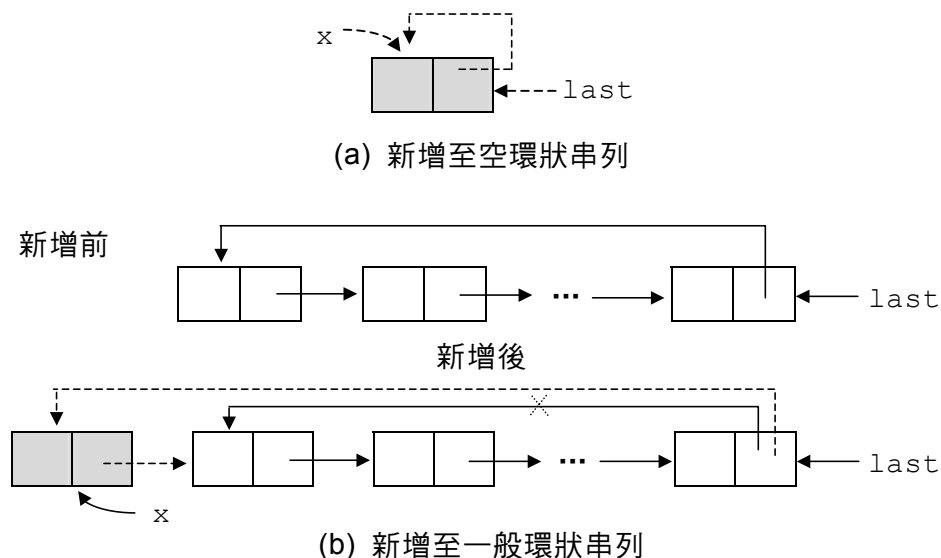


圖 4-22 在環狀串列中（有  $last$  指標指向結尾節點）新增最前節點

程式 4-12 中程序 `InsertFirst` 的時間複雜度只需  $O(1)$ 。比起程式 4-11 要改善許多。

另一種可能的解決之道，是用一個空白節點<sup>17</sup>(empty node) 當做串列的起始節點，由 `head` 指標所指向(原來的第一個節點串於其後)，如圖 4-23 所示。

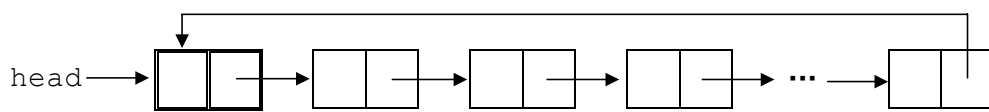


圖 4-23 以空白節點為首的環狀鏈結串列

各位應可瞭解，這種用一個空白節點當做環狀串列起始節點的設計，可使「在串列最前加入新節點」的需求變得簡單，此時「串列最前」實即 `first` 指向空白節點之後。在鏈結串列中搭配使用空節點，常可使程式碼更具一般化，省下邊界情況<sup>18</sup> (boundary condition) 的檢測。請設想：之前介紹的新增資料程序(如：`InsertAfter`、`InsertFirst`、`AttachDataToList`...等)需要針對是否為空串列做額外的條件判斷(即 `if (first==NULL)`)；倘若有起始空白節點的設計，則 `head` 一定不會是空的，前述的額外條件判斷就因毋庸置疑，豈不樂哉！

下面即是以起始空白節點為首之環狀串列(稱之為含開頭空白節點的環狀串列)，其新增最前節點的處理程序 `InsertFront`：

~~~~~

17 為免與空指標 (NULL) 混淆，特命其名為空白節點。

18 在 3.6 節老鼠走迷宮的矩陣迷宮，我們也用了圍牆的設計省下邊界情況的檢測；在本書可發現多處這樣的設計。

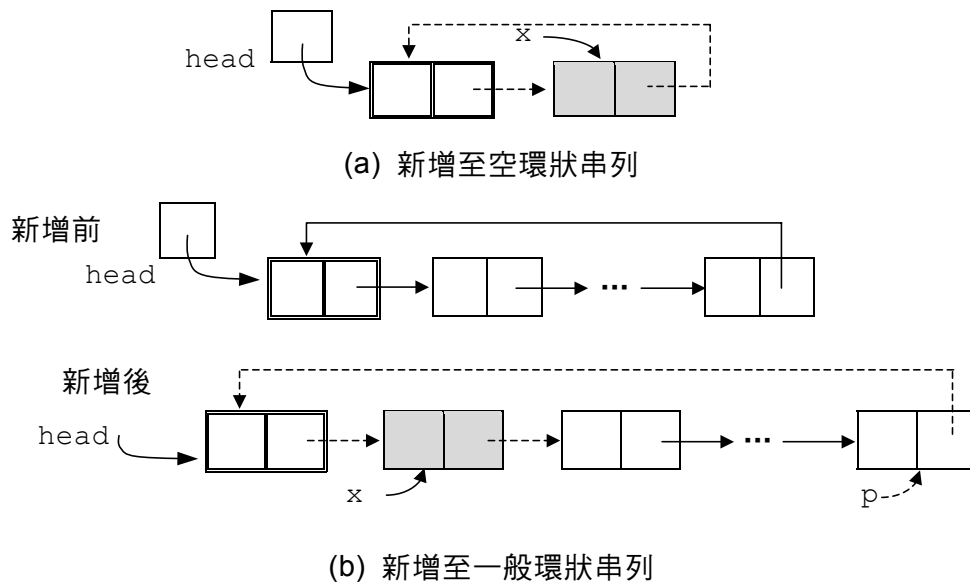
程式 4-14 在含開頭空白節點的環狀串列中新增最前節點

```

1 void InsertFront(struct node *x)
2 {   x->next = head->next;
3     head->next = x;
4 }

```

在程式 4-14 中只須將 $x \rightarrow \text{next}$ 設為 $\text{head} \rightarrow \text{next}$ (第 2 行)，並把 $\text{head} \rightarrow \text{next}$ 改為 x 即大功告成；不論是新增至空環狀串列 (如圖 4-24 (a)) 或一般非空環狀串列 (如圖 4-24 (b))，處理步驟皆相同。時間複雜度亦為 $O(1)$ ，而程式碼則精簡許多。

**圖 4-24** 新增最前節點一於以開頭空白節點為首之環狀串列中

倘若「新增資料節點使成含開頭空白節點環狀串列的最後節點」是經常發生的運算，可加入 `last` 指標指向最後節點 (初始狀態 `head` 與 `last` 皆指向開頭空白節點)，而此運算的處理程序 `InsertLast` 請見程式 4-15：

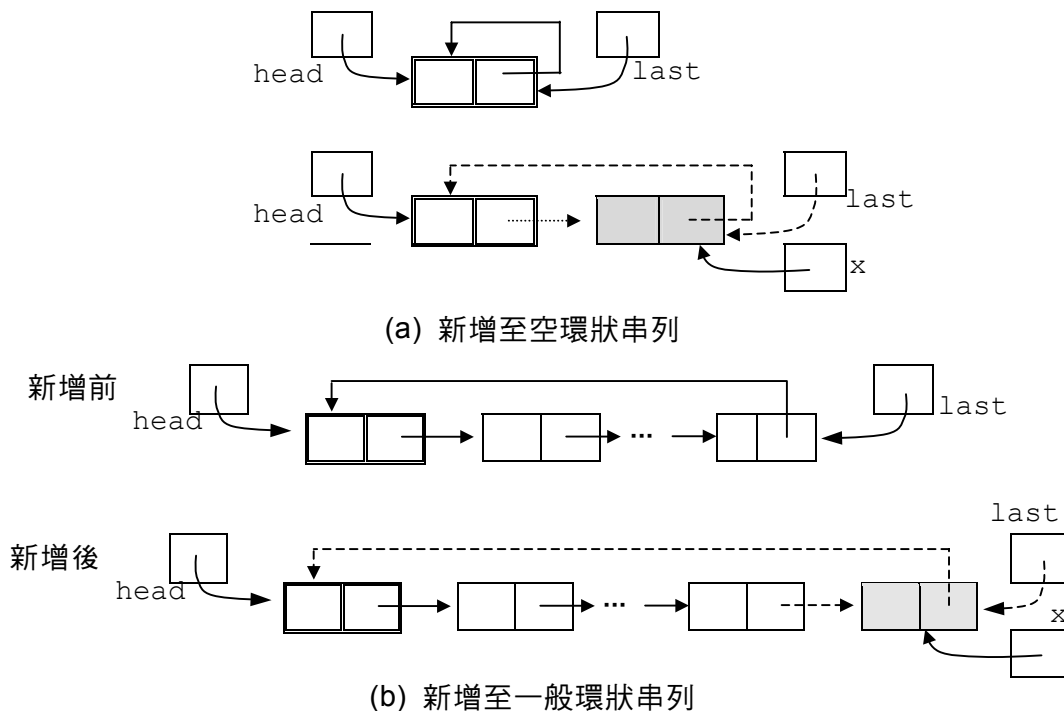
程式 4-15 在含開頭空白節點的環狀串列中新增最後節點

```

1 void InsertLast(struct node *x)
2 {   x->next = last->next;
3     last->next = x;
4     last = x;
5 }

```

與程式 4-14 比較，程式 4-15 多了第 4 行更新 `last` 指標的動作；時間複雜度亦為 $O(1)$ 。不論是新增至空環狀串列（如圖 4-25 (a)）或一般非空環狀串列（如圖 4-24 (b)），處理步驟皆相同，然而程式碼則精簡許多。

**圖 4-25** 新增最後節點一於以開頭空白節點為首之環狀串列中

我們應要熟稔含開頭空白節點環狀串列的各種運算，程式碼的精簡程度和可讀性都會有相當程度的提昇。下面是「搜尋資料」和「搜尋並刪除之」的處理程序：

程式 4-16 在含開頭空白節點環狀串列中搜尋資料

```

1  struct node * SearchDataHeader(int target)
2  {   struct node *p;
3      for (p=header->next; (p->data!=target && p!=header);
           p=p->next) ;
4      return p;
5  }
```

第 3 行中 for 迴圈控制指標 p 的初始值是 header->next，因為開頭空白節點的下一個才是串列的第一筆資料；迴圈執行條件是：「節點資料非所欲尋和尚未尋訪所有節點」(p->data!=target && p!=header) 後者「尚未尋訪所有節點」由 p!=header 來確認（當 p 自 first->next 起，經歷 p=p->next，至 p==header 則遍訪所有節點）；迴圈控制指標的調整是 p=p->next 以保證各個節點資料皆有機會（在未找到 target 時）被查訪。注意：(p->data!=target && p!=header) 兩條件的順序可以更動，因為 p->data 不可能因為形成 NULL->data 而執行中斷。若尋獲，會傳回尋得的節點指標 p；否則，亦回傳 p（其為 NULL 也）。

程式 4-16-1 在含開頭空白節點環狀串列中搜尋資料並刪除之

```

1  int SearchDeleteHeader(int target)
2  {   struct node *q, *p;
3      for (q=first, p= header->next; (p->data!=target
           && p!=header); q=p, p=p->next);
4      if (p==header) return -1;
5      q->next = p->next;
6      free(p);
7      return target;
8  }
```

第 3 行中 for 迴圈控制指標 p 的初始值是 first->next，順帶設定 q=p；迴圈執行條件是：「節點資料非所欲尋和尚未尋訪所有節點」

($p \rightarrow \text{data} \neq \text{target} \ \&\& \ p \neq \text{header}$)；迴圈控制指標的調整是 $q=p$ 之後， $p=p \rightarrow \text{next}$ 以保證各個節點資料皆有機會（在未找到 target 時）被 p 查訪，而且 q 會是 p 的前一節點指標。若尋訪所有節點後仍未尋獲，會傳回 -1；若尋獲 target 且刪除並回收空間（第 6 行），則回傳 target 。

4.6 多項式與串列

一般而言多項式可表示如下：

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$$

其中 a_ix^i ($0 \leq i \leq n-1$) 稱為 x 的 i 次方項 (term)、 i 為 x^i 的次方或幂 (power)、 a_i 則為其係數 (coefficient)；在此僅考慮幂次皆為正整數的多項式。上式 x 的幂次由大到小陳列，稱之為降幂多項式，若係數 $a_i = 0$ ，則省略該項。

用陣列或鏈結串列皆可表示，並進行多項式間的運算。我們在 4.6.1 節討論陣列與串列分別表示並處理多項式相加運算的細節；在 4.6.2 節利用串列多項式相加的基礎完成多項式相乘。

4.6.1 串列多項式相加

請先看下面的例子：假設 $A(x)$ 與 $B(x)$ 為 x 的多項式，以降幂排列如下

$$A(x) = 6x^5 + 5x^3 - 4x^2 + 8$$

$$B(x) = x^4 - 3x^3 + 4x^2 + 3x - 3$$

欲求 $C(x) = A(x) + B(x)$ ，可得

$$C(x) = 6x^5 + x^4 + 2x^3 + 3x + 5$$

由於各項係數與其幕次，是多項式的關鍵成員，可考慮用陣列存放之！我們可用 A、B 和 C 三個陣列分別儲存多項式 $A(x)$ 、 $B(x)$ 和 $C(x)$ 如圖 4-26——其中 $A[i]$ 存放的是 a_i ($A(x)$ 中 x^i 的係數)， $B[i]$ 和 $C[i]$ 亦然。

	5	4	3	2	1	0
A	6	0	5	-4	0	8
B	0	1	-3	4	3	-3
C	6	1	2	0	3	5

圖 4-26 以陣列表示多項式

陣列 C 的產生也很容易，利用

```
for (i=0; i<n; i++) C[i] = A[i]+B[i];
```

即可求出 $C(x)$ 的各項係數；其中 n 為 $A(x)$ 和 $B(x)$ 中幕次最高者。但是請設想：若 $A(x) = x^{1024} - 1$ 且 $B(x) = x^{2048} + 32$ ，諸如此類幕次很大、但項次很少的多項式，用陣列儲存會非常浪費空間！

我們嘗試運用鏈結串列——需要記住的關鍵資訊應是多項式中各項的幕次與係數，於是構思記憶各項的節點結構如下：

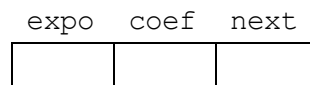


圖 4-27 多項式中各項的節點結構

程式 4-17 第 1~5 行即為此節點結構的宣告指令 (struct PolyNode)。

利用圖 4-27 的節點記住多項式中的各項，搭配上節提到的開頭空白節點（甚至加入指向最後節點的指標），上例多項式 $A(x)$ 、 $B(x)$ 和 $C(x)$ 可以用環狀串列結構如圖 4-28。利用指向開頭空白和最後節點的指標（圖 4-28 中的 head 和 last）可以很方便地新增各項的資料於串列最後（請參考程式 4-15）。

習題中有請讀者輸入各項（未必依降冪的順序）資料，建立降冪多項式串列的練習。

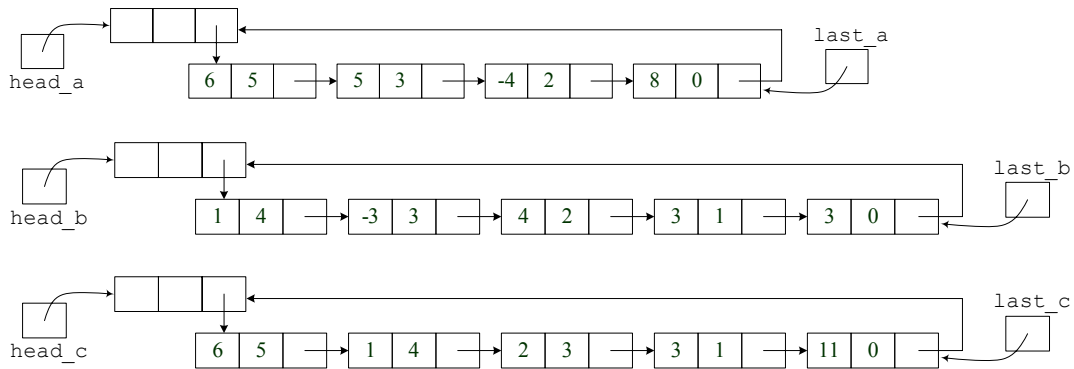


圖 4-28 以包含開頭空白節點的環狀單向串列表示多項式

至於如圖 4-35 中 $C(x) = A(x) + B(x)$ 該如何計算？請看下面的剖析：設定指標 p 和 q 分別指向多項式串列 A 和 B 中的某一項（顯然 $p = \text{head_a} \rightarrow \text{next}$ 和 $q = \text{head_b} \rightarrow \text{next}$ 是最初的設定），依據 $p \rightarrow \text{expo}$ 和 $q \rightarrow \text{expo}$ 的大小，有下列計算的規則：

- (1) 若 $p \rightarrow \text{expo}$ 大於 $q \rightarrow \text{expo}$ ，應將節點 p 複製，加入多項式串列 C 之後， p 往下一節點走訪；
- (2) 若 $p \rightarrow \text{expo}$ 小於 $q \rightarrow \text{expo}$ ，應將節點 q 複製，加入串列 C 之後， q 往下一節點走訪；
- (3) 若 $p \rightarrow \text{expo}$ 等於 $q \rightarrow \text{expo}$ ，應求算 $p \rightarrow \text{coef} + q \rightarrow \text{coef}$ ，若其不為 0，則應新建節點 x ，分別存入 $p \rightarrow \text{expo}$ 和 $p \rightarrow \text{coef} + q \rightarrow \text{coef}$ 於 expo 和 coef 欄位中，將 x 加入串列 C 之後；反之，若相加後係數為 0，則不必新增任何節點（捨去此項）。不論係數相加結果是否為 0， p 和 q 皆往下一節點走訪。

上述規則可反覆執行，俟 p 已走訪所有節點指在 head_a 或 q 已走訪所有節點指在 head_b ，再將剩下的項（可能在 A 、也可能在 B 、或恰好沒有剩餘）複製至串列 C 之後即可。程式與相關細節請見程式 4-17。

程式 4-17 多項式相加—含開頭空白節點的環狀串列

```

1  struct PolyNode
2  {   int coef;
3      int expo;
4      struct PolyNode * next;
5  };
6  struct PolyNode head_a, head_b, head_c;
7  struct PolyNode last_a, last_b, last_c;
8  struct node * NewTerm(int coefficient, int exponent)
9  {   struct PolyNode * r = new struct PolyNode;
10     r->coef = coefficient;
11     r->expo = exponent;
12     r->next = r;        // circular node
13     return r;
14 }
15 struct PolyNode * CopyTerm(struct PolyNode *p)
16 {   struct PolyNode *q = new struct PolyNode;
17     *q = *p;
18     return q;
19 }
20 void InsertLast(struct PolyNode *x, struct PolyNode *last)
21 {   x->next = last->next;
22     last->next = x;
23     last = x;
24 }
25 void Initilization()
26 {   head_a = NewTerm(0, -1);
27     last_a = head_a;
28     head_b = NewTerm(0, -1);
29     last_b = head_b;
30 }
31 struct PolyNode * PolyAdd(struct PolyNode * head_a,
                           struct PolyNode * head_b)
32 {   struct node *p, *q, *r = NULL;
33     int coef;
34     head_c = NewTerm(0, -1);
35     last_c = head_c;

```

```

36     p = head_a->next, q = head_b->next,
37     while (p!=head_a && p!=head_b)
38     {   if (p->expo > q->expo)
39         {   r = CopyTerm(p); p = p->next;   }
40         else if (p->expo < q->expo)
41         {   r = CopyTerm(q); q = q->next;   }
42         else
43         {   if ((coef=p->coef+q->coef)!=0)
44             {   r = CopyTerm(p); r->coef = coef;   }
45             p = p->next; q = q->next;
46         }
47         if (r)
48             {   InsertLast(r, last_c); r = NULL;   }
49     }
50     while (p!=head_a)
51     {   r = CopyTerm(p); p = p->next;   }
52     while (p!=head_a)
53     {   r = CopyTerm(q); q = q->next;   }
54     return head_c;
54 }

```

主程式可在多項式串列 A 和 B 皆輸入完成¹⁹ 後，呼叫

```
head_c = PolyAdd(head_a, head_b);
```

取得多項式串列 C (=A+B) 的結果。

程式 4-17 中第 1~7 行宣告了必要的節點結構 PolyNode 和指向串列開頭空白節點的 head 和 last (皆為全域變數)——共三組分別供多項式串列 A、B

~~~~~

19 若輸入多項式 A 的各項已是降冪型式，可用

```
x = NewTerm(coef, expo);
```

```
InsertLast(x, last_a);
```

將各項加入串列 A 作為最後節點；多項式 B 亦然。若非降冪型式，請在習題中練習。



和 C 使用。第 8~14 行定義了 NewTerm 程序，建立新節點 x，將傳入的係數和冪次分別存入對應欄位 coef 和 expo 中，next 則先設為指向自己（成為環狀節點，之後再依所需更改），再將指標 x 傳回。15~19 行則定義了 CopyTerm 程序，建立新節點 q，以第 16 行指令  $*q = *p$  將傳入指標 p 所指向節點 \*p 的內容設定給 \*q（節點中三個欄位一併設定；注意：next 欄位所指向顯然不正確，第 32 行會統籌更正之），再將指標 q 傳回。這裏使用了節點的複製（第 17 行  $*q = *p$ ），比起各欄位的設定（第 10~12 行）要來的簡潔易讀。20~24 行 InsertLast 可將傳入的節點 x 加至傳入指標 last 之後（與程式 4-15 相同）——其結果亦為含開頭空白節點的環狀串列。

第 25~30 行供初始化設定所用，目的在建立多項式串列 A 和 B 的開頭空白節點，分別皆為環狀串列。30~52 行定義了多項式相加的運算，傳入指向多項式串列 A 和 B 的指標 head\_a 和 head\_b 傳出指向多項式串列 C 的指標 head\_c，而  $C = A+B$ 。

InsertLas 目前的寫法沒有強調傳回的內容（void），若希望強調更改的內容可以傳回（強調更改的資訊、提高可讀性），不妨改寫 20~24 行如下：

```

20 struct PolyNode AttachLast(struct PolyNode *x,
                             struct PolyNode *last)
21 {   x->next = last->next;
22     last->next = x;
23     last = x;
24     return last;
25 }
```

讀者是否有留意程式 4-17 中 26、28 行所建的開頭空白節點係數為 0、冪次為 -1！這些設定自然合理，然而冪次為 -1 的項（假設  $p->expo$  為 -1）可以使任何項的冪次（假設  $q->expo$ ）與其比較時皆得複製後者的結果（假設  $p->expo$  為 -1，任何合理項  $q->expo$  皆大於 -1，由計算規則 (2) 即第 40 行可得  $r = \text{CopyTerm}(q)$ ； $q = q->next$ ；）；這可使程式流程的進行有更好的設計！我們可改寫程式 4-17 中 31~54 行如下：

**程式 4-18** 多項式相加—另一種寫法

```

61 struct PolyNode * PolyAdd(struct PolyNode * head_a,
                             struct PolyNode * head_b)
62 {   struct node *p, *q, *r = NULL;
63     int coef ;
64     last_c = head_c = NewTerm(0, -1);
65     for (p=head_a->next, q=head_b->next;
           (p!=head_a || q!= head_b); )
66     {   if (p->expo > q->expo)
67         {   r = CopyTerm(p); p = p->next;   }
68         else if (p->expo < q->expo)
69         {   r = CopyTerm(q); q = q->next;   }
70         else
71         {   if ((coef=p->coef+q->coef)!=0)
72             { r = CopyTerm(p); r->coef = coef;   }
73             p = p->next; q = q->next;
74         }
75         if (r)
76         {   last_c = AttachLast(r, last_c); r = NULL; }
77     }
78     return head_c;
79 }

```

其中原 36、37、50~53 行 while 迴圈與其判斷條件

```

36     p = head_a->next, q = head_b->next,
37     while (p!=head_a && p!=head_b)
...
50     while (p!=head_a)
51     {   r = CopyTerm(p); p = p->next;   }
52     while (p!=head_a)
53     {   r = CopyTerm(q); q = q->next;   }

```

修改成為第 65 行的 for 迴圈：

```
65    for (p=head_a->next, q=head_b->next;
        (p!=head_a || q!=head_b); )
```

後者利用迴圈變數設定來初始化  $p=\text{head\_a} \rightarrow \text{next}$ ,  $q=\text{head\_b} \rightarrow \text{next}$  (消化了 36 行)，條件判斷則採用  $(p \neq \text{head\_a} \ || \ q \neq \text{head\_b})$  使得  $p$  已指在  $\text{head\_a}$  時，可等待  $q$  與之比較後以規則 (2) 自然走訪剩餘節點 (並複製節點至  $C$ )，直到  $q$  也指向  $\text{head\_b}$  方才結束 (相對地，當  $q$  已指在  $\text{head\_b}$  時， $p$  也會自然走訪剩餘節點) — 原 50~53 行即可省略了。

76 行強調了 AttachLast 更動了全域變數  $\text{last\_c}$ ，提醒閱讀程式者有此變更 (原 48 行的寫法則沒有這提醒，兩者皆不影響正確性，各有風格也)。

```
76    {    last_c = AttachLast(r, last_c); r = NULL; }
```

程式 4-17 和 4-18 必須走訪多項式  $A(x)$  和  $B(x)$  的所有節點一次，所以時間複雜度皆為  $O(m+n)$  — 其中  $m$  和  $n$  是多項式  $A(x)$  和  $B(x)$  的項數。注意：令多項式  $A(x)$  和  $B(x)$  的最高冪次為  $p$  和  $q$ ，則  $p \geq m$  且  $q \geq n$ 。用陣列表示多項式時，其相加運算的時間和空間複雜度則為  $O(p+q) (\geq O(m+n))$ 。

一旦有了多項式串列相加的基礎，想完成多項式相乘的運算就不難了，我們在下一小節討論。

#### 4.6.2 串列多項式相乘

一樣地先看一個例子：令

$$A(x) = -4x^2 + 1$$

$$B(x) = 4x^3 - 9x^2 + 2x - 3$$

則  $D(x) = A(x) \times B(x) = 18x^5 + 27x^4 + 2x - 3$ 。一般的算術運算如下：

|        |          | $x^5$ | $x^4$ | $x^3$ | $x^2$ | $x^1$ | $x^0$ |
|--------|----------|-------|-------|-------|-------|-------|-------|
| $A(x)$ |          |       |       |       | 3     | 0     | 1     |
| $B(x)$ | $\times$ |       |       | 6     | 9     | 2     | -3    |
| ①      |          |       |       |       | -9    | 0     | -3    |
| ②      |          |       |       | -6    | 0     | 2     |       |
| ③      |          |       | 27    | 0     | 9     |       |       |
| ④      |          | 18    | 0     | 6     |       |       |       |
| D      |          | 18    | 27    | 0     | 0     | 2     | -3    |

圖 4-30 多項式相乘的例子

由圖 4-30 可知，最基本的運算是  $B(x)$  各項與  $A(x)$  各項相乘——其計算原則則為係數相乘、幕次相加——形成新的一項。而後可將  $B(x)$  每項乘上  $A(x)$  的結果存成不同的串列（如上式的 ①~④，其中 ① =  $-3 \times A(x)$ 、② =  $2x \times A(x)$ 、③ =  $9x^2 \times A(x)$ 、④ =  $6x^3 \times A(x)$ ），再一併把這些串列相加；但這需要  $O(mn)$  的串列節點空間——其中  $m$  和  $n$  是  $A(x)$  和  $B(x)$  的項數——空間的使用並不經濟。既然已有了上節介紹兩個多項式相加的程序 PolyAdd，時間與空間僅為  $O(m+n)$ ，可以重覆呼叫 PolyAdd，亦即：求 ①+②、再將其和加上③、爾後將該和與④相加，即可求得  $A(x) \times B(x)$ 。茲將上述討論整理成為演算法如下：

## 演算法 4-1 多項式相乘

輸入：多項式串列 A 和 B（含開頭空節點、最後節點指標），項數分別為  $m$  和  $n$   
 輸出：多項式串列  $D = A \times B$

```

1  設定串列 D 的開頭空白節點指標為 head_d，其亦為最後節點指標 last_d
2  設定串列 S 的開頭空白節點指標為 head_s，其亦為最後節點指標 last_s
   // 串列 S 用來存放 B 中某一項乘上 A 的結果
3  for (任一項  $y \in B$ ，共  $n$  項)
4  {   for (任一項  $x \in A$ ，共  $m$  項)
5      {    $z =$  以  $x \rightarrow \text{coef} * y \rightarrow \text{coef}$  為係數、 $x \rightarrow \text{expo} + y \rightarrow \text{expo}$  為幕次
           新建對應的節點；
6          將  $z$  節點新增至串列 S 之後
7      }   // 4~7 行即產生  $y * A$  (B 中一項乘上 A) 的結果，存放於串列 S
8       $D = D + S$  亦即  $\text{head\_d} = \text{PolyAdd}(\text{head\_d}, \text{head\_s})$ ;
```

```

9      清空 s，以供下回合使用；
10 } //經過 n 次累加，D 即為 A×B
11 return 串列 D (=A×B)

```

演算法 4-1 中 4~7 行產生  $y^*A$  ( $B$  中一項乘上  $A$ ) 的結果，存放於串列  $S$ ，第 8 行執行，在 3~10 行的迴圈驅使下， $D$  即可累加出各  $y^*A$  ( $y \in B$ ) 的結果——即為  $A^*B$  也。第 9 行的清空串列  $S$ ，旨在除本回合的  $y^*A$ ，以供下回合使用。

若多項式  $A(x)$  和  $B(x)$  的最高幕次分別為  $p$  和  $q$ ，則乘積  $A(x) \times B(x)$  的最高幕次為  $p+q$ ，意味著串列  $D$  最多會使用  $O(p+q)$  個節點。注意：串列  $S$  存放  $y^*A$  的結果，當其最高幕次為  $\beta$  時，會使用了  $\beta$  個節點，而且  $p \leq \beta \leq p+q$ ；此時  $D = D + S$  使用  $O(\beta + p + q) = O(p + q)$  個節點串列。總體而言，第 5~6 行執行的總次數為  $O(mn)$ ，而第 8 行執行的總次數為  $O(n(p+q))$ ，演算法 4-1 的時間複雜度為  $O(mn) + O(n(p+q)) = O(n(m+p+q))$ 。由於第 3 行與第 4 行的 for 迴圈可以對調<sup>20</sup>，屆時依相似的分析方式，可推得時間複雜度為  $O(m(n+p+q))$ 。程式 4-19 即採用後者策略編寫程式。

#### 程式 4-19 多項式相乘—含開頭空白節點的環狀串列

```

1 struct PolyNode * ClearTerm(struct node * head)
2 {   struct PolyNode * p, * q;
3     for (p = head->next; p != head; q = p, p = q->next, free(q));
4     p->next = head;           // 環狀串列只利開頭空白節點
5     return p;
6 }

```



<sup>20</sup> 演算法 4-1 目前以  $y^*A$  ( $B(x)$  的每項乘上  $A(x)$ )，然而 3 和 4 行對調（變數維持不動）用  $x^*B$  ( $A(x)$  的每項乘上  $B(x)$ ) 也可以求得  $A(x) \times B(x)$ ；後者則需  $O(m)$  回合的多項式串列相加。

```

7 struct PolyNode * PolyMult(struct PolyNode *head_a,
                             struct PolyNode *head_b)
8 {   struct PolyNode *x, *y, *z, *head_d, *head_s, *last_d, *last_s;
9     last_d = head_d = NewTerm(0, -1);
10    last_s = head_s = NewTerm(0, -1);
11    for (x=head_a->next; x!=head_a; x=x->next)
12    {   for (y=head_b->next; y!=head_b; y=y->next)
13        {   z = NewTerm(x->coef*y->coef, x->expo+y->expo);
14            last_s_ = AttachLast(z, last_s);
15        }
16        head_d = PolyAdd(head_d, head_s);
17        last_s = ClearTerm(head_s);
18    }
19    free(head_s);
20    return head_d;
21 }

```

程序 `PolyMult` 可完成以 `head_a` 和 `head_b` 所指向的多項式串列的相乘運算。主程式中只須以 `struct PolyNode *` 定義 `head_d`，即可呼叫之（開頭空白節點 `head_d` 的實體空間則在第 9 行中定義）。

```

struct PolyNode *head_d;
head_d = PolyMult(head_a, head_b);

```

本節利用多項式的運算（4.6.1 節介紹相加、4.6.2 節討論相乘），探索了指標、串列在程式寫作上的諸多技巧，讀者可以細細品味並勤加練習。

## 4.7 稀疏矩陣

顧名思義「稀疏矩陣」即為僅含少數元素值不為 0（大部份元素值為 0）

的矩陣。利用串列來儲存稀疏矩陣中非 0 的值，肯定對記憶空間的精簡有所幫助。上節提到的多項式，用到了含開頭空白節點的環狀串列，我們可視其為一維的串列；利用一維串列的結構直接存放稀疏矩陣中所有的非 0 值，是個可能的方式；而本節介紹二維串列的架構來存放這些非 0 值，對可能的運算（如：矩陣相加、相乘）大抵會比較方便。

稀疏矩陣中非 0 元素的列、行註標和值是必要的資訊，而與其同列中的下一個非 0 元素，和與其同行的下一個非 0 元素之所在也必須掌握，那些 0 的元素即可不予理會。圖 4-31 描繪了記錄這些資訊的節點結構，其中 row、col 和 value 分別存放非 0 元素的列註標、行註標和值，down 和 right 則記錄同列和同行的下一非 0 元素的節點住址。注意：雖然圖 4-31 所繪出的五個欄位並不完全相同（如此繪製僅為方便後續畫出稀疏矩陣如圖 4-32），但各欄位皆為指標——實體空間的大小是相同的！

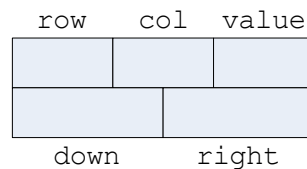


圖 4-31 適合存放稀疏矩陣元素資訊的節點結構

圖 4-32 描繪了一個稀疏矩陣與其對應串列的範例，其中包含了四類節點（都用了與圖 4-31 相同的節點結構）：

- (1) 矩陣的開頭空白節點（左上角節點）：row 欄位存放列的大小、col 欄位存放行的大小、down 欄位存放第 0 列標題節點的住址、right 欄位存放第 0 行標題節點的住址。
- (2) 列標題節點（左側節點）：row 欄位存列註標值、col 欄位存-1、value 欄位存放該列非 0 元素的個數、down 欄位存放下一列標題節點的住址、right 欄位存放該列第 1 個非 0 元素節點的住址。
- (3) 行標題節點（最上列節點）：row 欄位存-1、col 欄位存行註標值、value

欄位存放該行非 0 元素的個數、down 欄位存放該行第 1 個非 0 元素的節點住址、right 欄位存放下一行標題的節點住址 )。

- (4) 非 0 元素節點：row 欄位記載其列註標、col 欄位記載其行註標、value 欄位存其值，而其同行和同列的下一元素節點住址分別記載於 down 和 right 欄位。

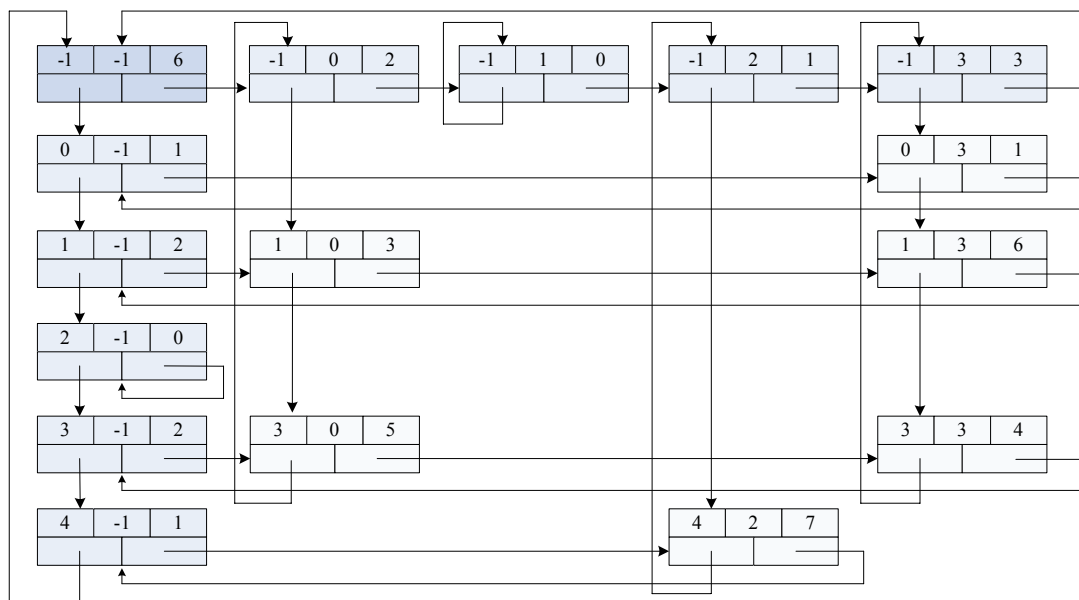


圖 4-32 稀疏矩陣的例子

在此所有的節點結構皆相同（也可設計行列標題節點和內容節點結構不同）；各列為環狀串列，其最後一個非 0 元素節點的 right 會指回該列標題節點（視其為開頭空白節點），且各行亦然（最後一個非 0 元素的 down 會指回該行標題節點），而最後一個列標題節點的 down 和最後一個行標題節點的 right，則指向矩陣的開頭空白節點。

串列、稀疏矩陣、甚至於是稀疏多維矩陣、...等結構，都有節省空間的優點，讀者宜掌握其儲存資料的精髓（將必要的資訊儲存起來、非必要者捨去之）。然而其輸入、運算處理、輸出就需要考量周全，方能搭配使用。



## 4.8 雙向鏈結串列控制

以上所介紹的鏈結串列，不論是線性或是環狀的，每個節點都只有一個鏈結指標，我們只能朝一個方向走訪其他節點；若有反向走回的需求<sup>21</sup>，單向線性或單向環狀串列將無法有效率地解決。於是有「雙向鏈結串列」(doubly linked list) 隨之孕育而生。圖 4-33 是為線性雙向鏈結串列，其第一個節點由 first 指標所指向。

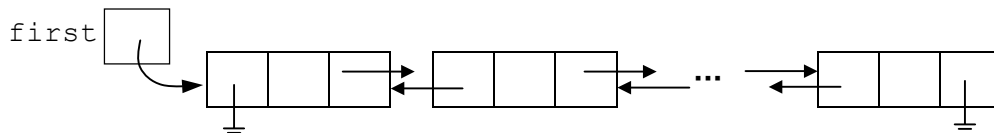


圖 4-33 線性雙向鏈結串列

這兒每個節點皆有兩個鏈結指標：『前指標』和『後指標』——分別指向該節點的前一個和下一個節點。第一個節點的『前指標』和最後一個節點的『後指標』皆為 NULL，表示它們分別正是第一個和最後一個節點。若將第一個節點的『前指標』指向最後一個節點，最後一個節點的『後指標』指向第一個節點，則形成「環狀雙向鏈結串列」，如圖 4-34 所示。

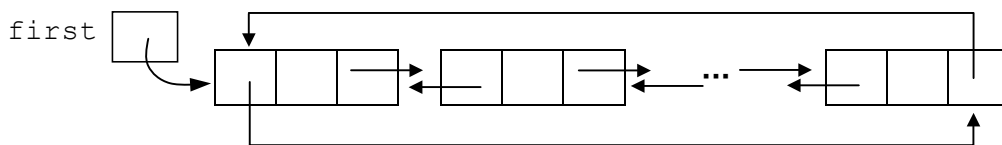


圖 4-34 環狀雙向鏈結串列

下面各小節則針對環狀雙向鏈結串列做：節點宣告和資料新增而成新節點（4.8.1 節）、搜尋資料（4.8.2 節）、插入節點（4.8.3 節）和刪除節點（4.8.4



21 例如說：串列以蛋糕價格由小至大串聯，希望找到低於 300 元的 10 種價格最便宜的蛋糕。

節)等運算的討論。

### 4.8.1 新增資料至環狀雙向鏈結節點

先討論如何宣告環狀雙向鏈結串列所需要的節點，請看程式 4-20：

**程式 4-20** 雙向鏈結串列的節點宣告

```

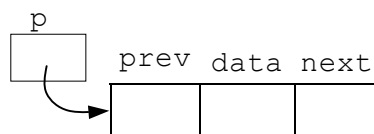
1  struct Dnode
2  {   struct Dnode *prev;
3      int data;
4      struct Dnode *next;
5  };
6  struct Dnode *first;
```

於是透過

```

struct Dnode * p;
p = (struct Dnode *) malloc(sizeof(struct Dnode));
```

的指令，可以產生一個型態為 struct Dnode 的節點，由 p 指標所指向，如圖 4-35：



**圖 4-35** 雙向鏈結節點

此後可以用 `p->prev`、`p->data` 和 `p->next` (或 `(*p).prev`、`(*p).data` 和 `(*p).next`) 分別取用此節點的三個欄位。

我們可以利用下面的程序 `NewDnode`，產生新的 struct Dnode 節點，並放置整數資料 `element` 於其中：

**程式 4-21** 產生新雙向鏈結節點存放傳入的整數

```

1  struct Dnode *NewDnode(int element)
2  {   struct Dnode *p;
3      p = (struct Dnode *) malloc(sizeof(struct Dnode));
4      p->data = element;
5      p->prev = NULL;
6      p->next = NULL;
7      return p;
8  }
```

第 1 行的宣告中，傳入的參數為欲放入新節點內的資料，傳出的是新增加節點的指標。第 3 行以動態配置的方式取得了一個新節點，大小為 struct Dnode 節點所定義的大小，指標 p 指向此新節點的位址。第 4~6 行填入新節點的必要資訊，其中兩個鏈指指標 p->prev 和 p->next 都設定為 NULL，第 7 行傳出指標 p（此新節點的位址）。任何其它程序有新增 struct Dnode 節點需求者，皆可直接呼叫之。

#### 4.8.2 搜尋環狀雙向鏈結串列中的資料

環狀雙向鏈結串列中應有一指標指向其中一個節點，做為串列的起點（既然是環狀串列，指向其中任一節點的指標，總有辦法走訪串列中的所有節點），通常選擇指向第一個節點，令之為 first，正如圖 4-34 所示。我們可以自此指標起，逐一搜尋欲尋找的資料，請看下面的程序：

**程式 4-22** 搜尋環狀雙向鏈結串列中的資料

```

1  struct Dnode *SearchDnodeList(int target)
2  {   struct Dnode *p;
3      if (first == NULL)
4          SearchEmptyList();
```

```
5     else
6     {   p = head;
7         while ((p->data!=target) && (p!=first))
8             p = p->next;
9         if (p->data==target) return p;
10    }
11    return NULL;
12 }
```

第 3~4 行檢測了是否在空串列中找資料，若是則利用程序 Search EmptyList() 做適當回應，傳回 NULL (第 11 行)；若串列中確有資料，則自 head 起逐一搜尋個節點的資料 (6~8 行)；若的確尋得，則傳回該資料所在的節點 (第 9 行)；否則傳回 NULL，以示找不到。請留意：第 7 行的條件 (p->data!=target) && (p!=first) 在尋獲時，p 會指向 target 所在的節點；搜尋不得時，p 會指向 first 所指向節點；為區分究竟有沒有找到，遂有第 9 行 (p->data==target) 的確認判斷。第 6~8 行的 while 迴圈可以用 for 迴圈精簡改寫如下：

```
for (p=head; (p->data!=target) && (p!=first); p = p->next);
```

請各位試試看可否循 prev 指標，達到一樣的搜尋效果。

### 4.8.3 插入節點至環狀雙向鏈結串列

若希望將節點 p 插入環狀雙向鏈結串列中節點 x 的後面，我們必須妥善處理鏈結間的變化，請見圖 4-36：

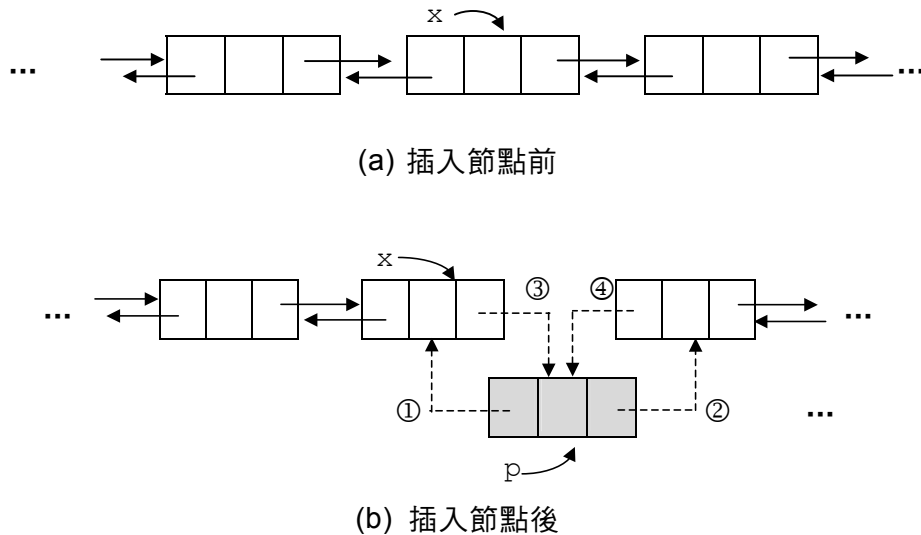


圖 4-36 插入節點至環狀雙向鏈結串列

由圖 4-36 可知有四個鏈結指標會有異動：

- ① `p->prev = x ;`
- ② `p->next = x->next ;`
- ③ `x->next->prev = p ;`
- ④ `x->next = p ;`

請注意上面的異動順序，試想若步驟 ③ 和 ④ 對調，答案就不對了<sup>22</sup>。我們整理上述理念，寫成如下的程序：

~~~~~

22 指標的指向有誤，很容易造成程式當掉或語意邏輯的錯誤，造成除錯上的困擾。一個建議：新建節點的欄位（文中的①和②）先行填入，原串列的指標（文中的③和④）爾後更新。在大多的情形中，這個建議是挺有用的。

程式 4-23 插入節點至環狀雙向鏈結串列中

```

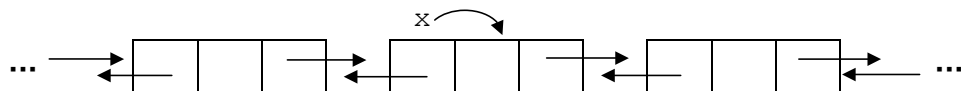
1 void InsertDList(struct Dnode *x , struct Dnode *p)
2 {   if (x==NULL)
3     {   p->prev = p;
4         p->next = p;
5         first = p;
6     }
7     else
8     {   p->prev = x;
9         p->next = x->next;
10        x->next->prev = p;
11        x->next = p;
12    }
13 }

```

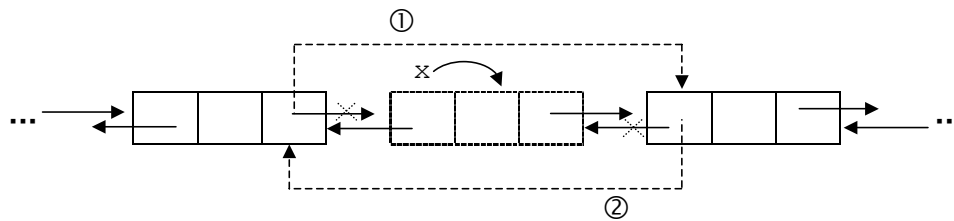
第 3~5 行執行新增至空雙向串列的情況，此時新節點 p 為雙向串列的第一個節點；第 8~12 行則依循執行上述步驟 ①~④ 的指標設定。請想想：倘若想在第一個節點之前新增資料節點該如何處理？（ $first$ 是否應改設成指向新加入的節點？）

4.8.4 刪除雙向鏈結串列的一個節點

假設希望自雙向鏈結串列中刪除 x 所指向的節點，請見圖 4-37 如下：



(a) 刪除節點前



(b) 刪除節點後

圖 4-37 自環狀雙向鏈結串列中刪除節點

由圖 4-37 可知，異動的鏈結指標有兩處：

- ① `x->prev->next = x->next ;`
- ② `x->next->prev = x->prev ;`

於是我們寫出下面的程式：

程式 4-24 刪除雙向鏈結串列中的一個節點

```

1 void DeleteDList(struct Dnode *x)
2 {   if (x==NULL)
3     DeleteEmptyList();
4     else { if ((x==first) && (first->next==first))
5         first = NULL ;
6     else
7     {   x->prev->next = x->next ;
8         x->next->prev = x->prev ;
9     }
10
11     free(x);
12 }
13 }
```

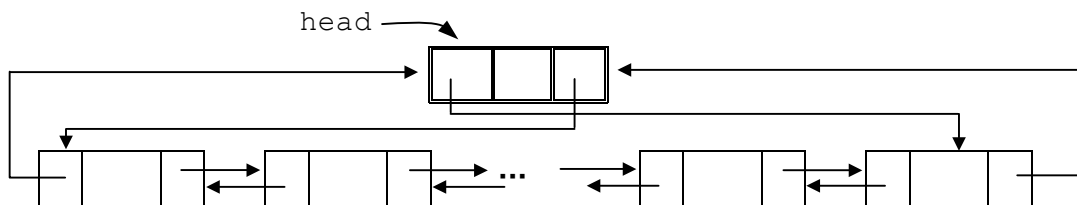
第 3 行利用程序 `DeleteEmptyList`，給予欲刪除空節點時的錯誤訊息；第 5 行所做的設定乃因欲刪除節點的串列只有一個節點（第 4 行：不但 `(x==first)` 而且 `(first->next==first)` 同時成立），遂在刪除後，將造

成空串列！第 8 和第 9 行分別對應上述的步驟①和②；第 11 行把欲刪除的節點 x 歸還給系統。

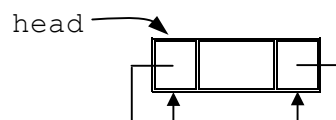
4.8.5 包含開頭空白節點的鏈結串列

不論單向或雙向鏈結串列，各位皆可發覺：串列節點的新增或刪除都要考慮是否為空串列的特例；這種空串列的檢測在每次新增、刪除動作發生時，都得執行，會降低程式的執行效率。在 4.5 節中我們曾討論在串列的開頭加上空白節點的好處；這個概念可擴展到各種鏈結串列的設計上。

於是我們加上一個不含資料的開頭空白節點，以省下檢測空串列的動作。圖 4-38 (a) 為一包含開頭空白節點（令之指標為 $head$ ）的環狀雙向鏈結串列，開頭空節點的下一個節點（ $head \rightarrow next$ ）應指向原第一個節點，開頭空節點的前一個節點（ $head \rightarrow prev$ ）應指向原最後節點；圖 4-38 (b) 則為空的環狀雙向鏈結串列，它僅有一個開頭空節點。



(a) 環狀雙向鏈結串列



(b) 環狀雙向鏈結空串列

圖 4-38 包含開頭空白節點的環狀雙向鏈結串列

如此一來，前面所提的環狀雙向鏈結串列新增節點程序 InsertDList（節 4.6.3）、環狀雙向鏈結串列刪除節點程序 DeleteDList（節 4.6.4）和環狀雙向鏈結串列搜尋程序 SearchDList（節 4.6.2）可依所加入的開頭空白節點分別改寫成 InsertHDLList、DeleteHDLList 和 SearchHDLList 如下：

程式 4-25 環狀雙向鏈結串列（含開頭空白節點）新增、刪除和搜尋節點資料

```

1 void InsertHDLList(struct Dnode *x, struct Dnode *p)
2 {   p->prev = x ;
3     p->next = x->next ;
4     x->next->prev = p ;
5     x->next = p ;
6 }
7 void DeleteHDLList(struct Dnode *x)
8 {   if (x == first)
9       DeleteEmptyList();
10    else
11    {   x->prev->next = x->next;
12        x->next->prev = x->prev;
13        free(x);
14    }
15 }
16 struct Dnode *SearchHDLList(int target)
17 {   struct Dnode *p;
18     for (p=head->next;
19          ((p->data!=target) && (p!=head));
20          p = p->next);
21     return p;
22 }
```

程式 4-25 中的程序 `InsertHDLList` 比起程式 4-20 中的程序 `InsertDLList` 要簡潔許多，原因即在於開頭空白節點的設計，省下了檢測是否為空串列的程式碼（程式 4-25 程序 `DeleteHDLList` 比起程式 4-24 程序 `DeleteDLList` 亦然；`SearchDLList` 更是明顯）；這樣的程式碼執行時也比較有效率。`SearchDLList` 中 18 行 `for` 迴圈控制指標 `p` 的初始值是 `head->next`；迴圈執行條件是：「節點資料非所欲尋和尚未尋訪所有節點」（`p->data!=target && p!=head`）；迴圈控制指標的調整是 `p=p->next`，以保證各個節點資料皆有機會（在未找到 `target` 時）被 `p` 查訪。若尋訪所有節點後仍未尋獲（`p==head`），即傳回 `p`（即為 `head`）；否則傳回尋獲的節點所在 `p`。這樣寫法是否比程式 4-22 的 `SearchDnodeList` 簡潔些。

4.9 C 語言的指標

在 C 語言中，指標 (pointer) 是個存放位址 (address) 的變數。指標在運用的時候，可能與位址、程序中的參數以及陣列間有密切的關係，我們就分別在以下各小節中，討論在不同情境下指標的用法。

4.9.1 指標與位址

直接看範例如下：

範例 4-6

若我們定義如下：

```
int x = 5;
int *p;
```

則表示整數變數 `x` 的內容設定為 5，而 `p` 是個位址變數，可以儲存某個整數在

記憶體中所在的位址；或解讀成 *p 是個整數。若執行

```
p = &x;
```

表示整數變數 x 在記憶體中的位址，要存在整數位址變數 p 中；或說「p 指向 x」。圖 4-39 顯示了 x 與 p 的邏輯圖示：

變數	位址	內容
x	A0A0	5
		:
p	A0D0	A0A0

圖 4-39 變數、位址與內容（可能是數值或指標）之間的關係

其中 &x 即為整數變數 x 的位址 A0A0，p = &x 則把位址 A0A0 指定給整數位址變數 p。所以現在 x 是 5，*p 也是 5。

整數 *p 可解讀成：「以 p 的內容為位址，取出該位址的整數內容，令其為 *p」。在此例中，p 的內容為 A0A0，取出位址 A0A0 的整數內容即為 5。&x 則可解讀為：「取出整數變數 x 在記憶體中的位址，令之為 &x」。在此例中，x 為 5，而 &x 是 A0A0。同理 p 為 A0A0，而 &p 為 A0D0。

於是下面的指令皆是有意義的：

```
*p = 0;
*p += 1;
++*p;
(*p)++;    //注意 (*p)++和*p++不相同，後者與*(p++)同
int *q;
q = p;
```

我們用圖 4-40 來加深各位的印象：

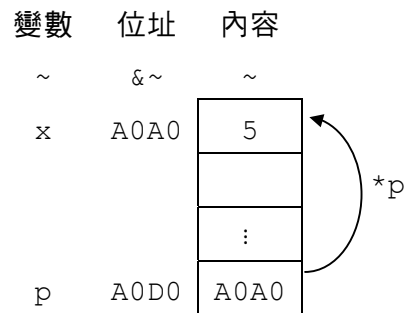


圖 4-40 利用指標（位址）取得位址內的數值

在 C 語言中用到變數名稱 ~（以上例而言，就是 x 或 p），即在取用其內容；而 &~ 則在取用其位址；*~ 則是以 ~ 內容為位址，取用該位址的內容。

下面是類似的例子：

範例 4-7

若我們定義如下：

```
char c = 'a';
char *p;
```

則表示字元變數 c 的內容設定為 'a'，而 p 是個位址變數，可以儲存某個字元在記憶體中所在的位址。若執行

```
p = &c;
```

表示字元變數 c 在記憶體中的位址，要存在字元位址變數 p 中；或說「p 指向 c」。圖 4-41 顯示了 c 與 p 的邏輯圖示：

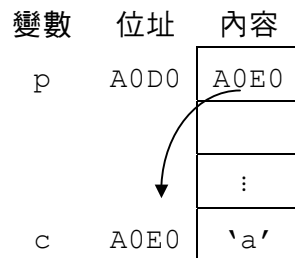


圖 4-41 利用指標（位址）取得位址內的字元

4.9.2 指標與程序參數

在本小節中，我們討論指標該如何傳入程序中，請看程式 4-26，它是個錯誤的資料對調處理程序：

程式 4-26 錯誤的資料對調

```

1 void swap(int x, int y)
2 {   int temp;
3     temp = x;
4     x = y;
5     y = temp;
6 }
```

呼叫

```
swap(a, b);
```

無法得到 a 和 b 資料對調的結果。錯誤的理由在於：參數 x 和 y 是以傳值傳入程序 swap 中，在程序內的變數 x 和 y 的確對調了，但並沒有傳出至呼叫程序中（程序內的變數 x 和 y 非呼叫程序中 a 和 b）。我們應如下般，將 a 和 b 的指標傳入程序 swap 中：

程式 4-27 資料對調

```

1 void swap(int *px, int *py)
2 {   int temp;
3     temp = *px;
4     *px = *py;
5     *py = temp;
6 }

```

呼叫

```
swap(&a, &b);
```

即可將 a 和 b 的位址 &a 和 &b 傳入程式 4-27 的程序 swap 中，分別以 px 和 py 存其位址，而 *px 和 *py 即可順利取用 a 和 b 的資料內容，再以 *px 和 *py 透過 temp 執行對調。

圖 4-42 說明了 a, b, px 和 py 間的關係。

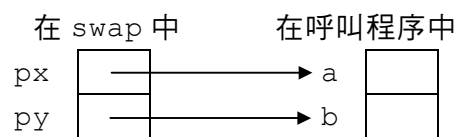


圖 4-42 在被呼叫程序中利用指標取用呼叫程序中的變數內容

4.9.3 指標與陣列

在第二章中曾介紹過陣列在記憶體中的配置，陣列中的元素與其位址的關係，可見 2-8 節。我們在範例 4-8 中舉例說明。

範例 4-8

若我們定義如下：

```
int a[]={1, 2, 3, 4, 5};
int *pa;
pa = &a[0];          // 將 a[0]的位址存在 pa 中
```

則陣列 a 中元素在記憶體中的配置，將如圖 4-43 (a) 所示（在此不另列出其位址，乃因其與不同的電腦硬體有關，在此只須掌握陣列元素的相對置即可）；而 pa 與陣列 a 的關係則如圖 4-43 (b) 所示，即 pa 存了 a[0] 的位置，*pa 會取用 a[0] 的內容（*pa 等於 1）。

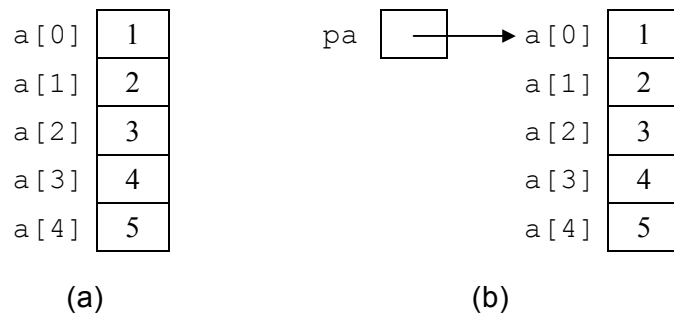


圖 4-43 陣列元素的位址可設定給指標變數

事實上陣列 a 的名稱本身與陣列的起始位址是同義的，所以 `pa=&a[0]` 亦可寫成：

```
pa = a;          //將陣列 a 的起始位址存在 pa 中
```

於是當 `i` 是整數時，`a[i]` 和 `*(a+i)` 是一樣的。在定址 `a[i]` 時，C 編譯器即在計算 `*(a+i)`（從整數陣列起始位址 `a` 算起第 `i` 個整數的位址的內容）。換句話說 `&a[i]` 和 `a+i` 也是相同的，請見圖 4-44。

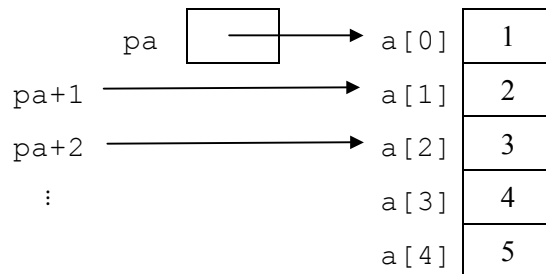


圖 4-44 以指標變數指向陣列元素

指標和陣列名稱也有不同之處：指標是個變數，所以 `pa=a` 或 `pa++` 是合理的；然而陣列名稱本身不是變數，所以 `a=pa` 和 `a++` 是不合理的（請比較：`pa=a` 是可以的，`a` 當位址而非變數、`a+i` 是位址的計算，是可以的；`a++` 相當於 `a=a+1`，`a` 當變數用，不可以！）

當陣列名稱要傳入程序中時，其實傳的即為陣列的起始位址。程式 4-28 是個計算字串長度的程序，它用字元位址指標變數來存放傳入的字元位址。

程式 4-28 計算字串長度

```

1  int strlen(char *s)
2  {   int n;
3      for (n=0; *s!='\0'; s++) n++;
4      return n;
5  }
```

第 1 行中已定義參數 `s` 為字元位址指標變數，第 3 行中的 `s++` 在計算下一元的位址，不會影響呼叫程序中的原始字元（串）。由於 C 以 `'\0'` 做為字串的結束識別字元，所以 `*s!='\0'` 可判斷是否已達字串的結尾。下面的呼叫皆是合理的：

```

strlen("dancing in the rain");
strlen(array);    // array 已宣告為 char array[100];
strlen(p);        // p 已宣告為 char *p;
```


各位應可認出

```
char s[];
```

和

```
char *s;
```

是同義的。在程式 4-28 中第 1 行遂可寫成：

```
int strlen(char s[])
```

本章習題

1. 設計一個演算法來計算一個由 *first* 指標指向開頭節點的單向鏈結串列所含的節點總數目，串列中最後一個節點的 *link* 欄為 NULL。計算此演算法的時間複雜度。
2. 用遞迴的概念，重做習題 1。
3. 設計三個演算法分分別來計算：
 - (a) 由 *first* 指標指向開頭節點的雙向鏈結串；
 - (b) 由 *first* 指標指向開頭空節點的雙向鏈結串；
 - (c) 由 *p* 指標指向其中一節點的雙向鏈結串；

所含的非空節點總數目，串列中最後一個節點的 *link* 欄為 NULL。分析此三個演算法的時間複雜度。
4. 寫出三個程序，分別刪除由 *first* 指標指向開頭節點的
 - (a) 單向鏈結串列（含或不含開頭空節點）；
 - (b) 單向環狀鏈結串列；
 - (c) 雙向環狀鏈結串列；

的所有節點。並計算此三程序的時間複雜度。
5. 令兩個鏈結串列分別為： $x = (x_1, x_2, \dots, x_n)$ 與 $y = (y_1, y_2, \dots, y_m)$ ，其穿插合併串列 z 定義成：若 $m \leq n$ 則 $z = (x_1, y_1, x_2, y_2, \dots, x_m, y_m, x_{m+1}, x_{m+2}, \dots, x_n)$ ，若 $m > n$ 則 $z = (x_1, y_1, x_2, y_2, \dots, x_n, y_n, y_{n+1}, x_{n+2}, \dots, y_m)$ ；而合併後， x 與 y 其原先每

一節點現在都存於 z 中，所以 x 與 y 將變為空串列。請設計演算法求得 x 與 y 的穿插合併串列 z ；且合併過程中不能使用其他節點。請分別考慮：

- (a) 單向鏈結串列（含或不含開頭空節點）；
- (b) 單向環狀鏈結串列；
- (c) 雙向環狀鏈結串列；

並計算演算法的時間複雜度。

6. 令兩個非遞減鏈結串列分別為： $x = (x_1, x_2, \dots, x_n)$ 與 $y = (y_1, y_2, \dots, y_m)$ ，每個鏈結串列中的節點是以資料欄值的非遞減順序排列。撰寫程序使能合併兩串列 x 與 y ，使成非遞減鏈結串列 z 。合併過程中不能使用其他節點。並計算此演算法的時間複雜度。請分別考慮：
 - (a) 單向鏈結串列（含或不含開頭空節點）；
 - (b) 單向環狀鏈結串列；
 - (c) 雙向環狀鏈結串列。
7. 令 p 為指向環形鏈結串列的一指標，如何將此串列當作一佇列使用？（亦即寫出其新增與刪除元素的程序）。
8. 多項式 (polynomial) 的數學表示為：

$$A(x) = \sum_{i=0}^n c_i x^i$$

$$= c_n x^n + c_{n-1} x^{n-1} + \dots + c_1 x^1 + c_0 \circ$$

此為降幂排列，因 x 的次數由大至小排列（若 x 的次數由小至大排列，則為升幂排列）；此多項式的幂次為 n 。請用鏈結串列來表示幂次為 n 的降幂多項式。

9. 令 A 和 B 皆為多項式，

$$A(x) = \sum_{i=0}^n a_i x^i, \quad B(x) = \sum_{i=0}^n b_i x^i,$$

多項式相加的定義為：

$$A(x) + B(x) = \sum_{i=0}^n (a_i + b_i) x^i。$$

請參考 3.6.1 節用鏈結串列表示多項式，設計出輸入多項式的程序，並與程式 4-17（多項式相加的程序）搭配，實作兩輸入多項式的相加，並輸出其結果。請分析並驗證使用演算法的時間與空間複雜度。

提示：可將節點設計成可儲存：非 0 係數 c_i 、其對應幕次 i 與下一非 0 項指標的節點。注意在多項式的 n 很大而非 0 項數很少時，鏈結串列的表示可節省空間。

10. 令 A 和 B 皆為多項式，

$$A(x) = \sum_{i=0}^n a_i x^i, \quad B(x) = \sum_{i=0}^n b_i x^i,$$

多項式相乘的定義為：

$$A(x) \times B(x) = \sum_{i=0}^n (a_i \times \sum_{j=0}^n b_j x^j)。$$

請用前題的鏈結串列表示並輸入兩多項式，並與程式 4-19（多項式相乘的程序）搭配，實作兩輸入多項式的相乘，並輸出其結果。請分析並驗證使用演算法的時間與空間複雜度。

11. 仿照 8、9 和 10 的題意，設計出多項式相減和相除的演算法。請分析所用演算法的時間與空間複雜度。

12. 請將 7~10 與第二章習題 7~10 比較，討論其陣列與鏈結串列在表示多項式時的優劣。
13. 利用環形串列重做習題 7~10。請比較兩做法的時間複雜度。
14. 設計一種串列表示法可以在串列兩頭做刪除與插入的動作，這種結構稱為雙向佇列 (deque)，寫一個自兩端做插入動作的程序。請分別考慮：
 - (a) 單向鏈結串列（含或不含開頭空節點）；
 - (b) 單向環狀鏈結串列；
 - (c) 雙向環狀鏈結串列。
15. 如節 4.7 所介紹：當矩陣的非 0 元素很多時，稱此矩陣為稀疏矩陣 (sparse matrix)。請設計儲存非 0 元素的節點結構。若所設計的結構與圖 4-31 不同，請比較並討論其優劣。
16. 請設計演算法並實作：將所輸入的資料，存成稀疏矩陣。
17. 請設計演算法，解決兩稀疏矩陣的相加。請分析演算法的複雜度。
18. 請設計演算法，解決兩稀疏矩陣的相乘。請分析演算法的複雜度。

