CHAPTER 4

LISTS

鏈結表示法

- 如果常常需要在串列中做加入和刪除的動作，那麼使用陣列來表示串列就不合適
- 串列元素可以用任意的順序存放在記憶體內
- 使用一種清楚明白的資訊(稱為鏈結)從一個元素走到(追蹤到)下一個元素

Introduction

- Array
  successive items locate a fixed distance
- disadvantage
  - data movements during insertion and deletion
  - waste space in storing n ordered lists of varying size
- possible solution
  - linked list

指標

Pointer

基本觀念

- 指標是一個用來指示資料存在於記憶體中的位址標示器
- 在指標的運用中，我們可瞭解到資料與位址間的關係，進而對記憶體配置有很大的幫助。
- 在C語言中,若某變數所含的是一個記憶體位址，此變數稱為指標變數。

指標變數的宣告

變數資料型態*變數名稱;

Int *ptr;

- ptr為指標變數
- ptr代表一個位址

→變數ptr→

1010H

ptr為指標變數

0800H

0802H

.....

1010H

...

16 ←*ptr

記憶體

注意事項

- ptr為指標變數，ptr代表一個位址
- *ptr代表此位址內的資料
- ptr所指向此位址之變數型態為整數 (int)
- 指標變數宣告的關鍵字為"*"；指標變數的資料型態也分為整數 (int) 、浮點數 (float) 、字元 (char)
- “&”為另一重要符號，“&”為一個特殊運算子，目的為傳回運算元之位址
- scanf()中之所有引數變數一定要加上“&”符號.

動態記憶體配置-說明

- **動態記憶體配置**不同於陣列的**靜態記憶體**配置
  - **靜態記憶體**配置是在**編譯階段**就配置記憶體空間，
  - **動態記憶體配置**是等到**執行階段**，才向作業系統要求配置所需的記憶體空間，可以讓程式設計者靈活運用程式所需的記憶體空間。
- 在C語言<stdlib.h>標頭檔的標準函式庫提供兩個函數：malloc()和free()，可以配置和釋放程式所需的記憶體空間。

動態記憶體配置-malloc()

malloc()函數：配置記憶體空間

- C語言的程式碼可以呼叫malloc()函數向作業系統取得一塊可用的記憶體空間，函數的語法，如下所示：

  fp = (資料型態*) malloc(sizeof(資料型態));

- 上述語法因為函數傳回void通用型指標，所以需要加上型態迫換，將函數傳回的指標轉換成指定資料型態的指標，sizeof運算子可以計算指定資料型態的大小。例如：配置一個浮點數變數的記憶空間，如下所示：

```
fp = (float *) malloc(sizeof(float));
struct test *score;
score=(struct test *) malloc(num*sizeof(struct test));
```

動態記憶體配置-free()

free()函數：釋放配置的記憶體空間

- free()函數可以釋放malloc()函數配置的記憶體空間，例如：指標fp是一個指向malloc()函數傳回的浮點數記憶體空間的指標，呼叫free()函數釋放這塊記憶體，如下所示：

  free(fp);

- 上述程式碼的指標fp可以是float浮點數指標，也可以是malloc()函數傳回的其它資料型態指標、陣列或結構指標。

4.1.1 Pointer Can Be Dangerous

pointer

```
int i, *pi, *pf;
pi = &i;            i=10 or *pi=10
pi= malloc(size of(int));
  /* assign to pi a pointer to int */
pf=(float *) pi;
  /* casts an int pointer to a float pointer */
```

4.1.2 Using Dynamically Allocated Storage

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc (sizeof(float));
*pi =1024;
*pf =3.14;
printf(”an integer = %d, a float = %f\n”, *pi, *pf);
free(pi);
free(pf);
```

request memory

return memory

***Program4.1:Allocation and deallocation of pointers (p.138)**

鏈結串列的定義

- 鏈結串列(Linkedlist)也是常用的資料結構裡面的元素稱為節點(node)
- 有兩個節點比較特別，就是串列的頭(head)與尾(tail)
- 每個節點除了含有一些成員屬性之外，還包含了一個指向下一節點的鏈結(link)。

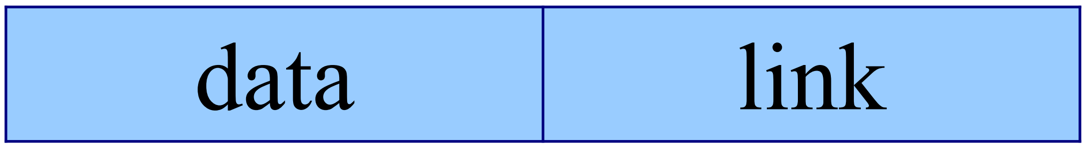
- 有次序排列之資料稱為串列(List)，如一年四季，數字0~9
- 陣列、堆疊及佇列皆屬於串列(List)的結構
- 鏈結串列(Linked List)的頭與尾決定於串列中的節點所在的位置，對於串列的處理，都是從頭節點開始，然後一一地找到其他的節點。

鏈結串列特點

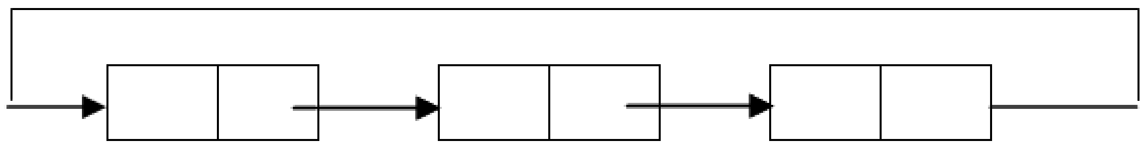
- 由節點所串成(利用鏈結)的串列稱之。
- 鏈結串列是一種有順序的串列，可以用陣列，也可以用結構。
- 鏈結串列各元素在記憶體之位置是不連續、隨機(Random)的。它是由動態記憶體分配節點(Node)串接而成。

鏈結串列的種類

- 單向鏈結：它的節點包含資料（data）及鏈結（link）兩個欄位



- 環狀鏈結＝＞一個鏈結串列之最後一個節點指向鏈結串列之最前端。



- 雙向鏈結＝＞節點至少包含資料及左右兩個結鏈。

link	data	link

陣列與鏈結串列的比較

陣列製作串列	
優點	缺點
1.易製作，宣告即可 2.亦存取資料，利用所以對應	1.刪除、插入及易動資料會造成資料移動頻繁，減少系統效率 2.宣告記憶體空間、造成不必要之浪費

鏈結製作串列	
優點	缺點
補足陣列串列之缺點	缺乏陣列串列之優點

動態配置之鍵結串列

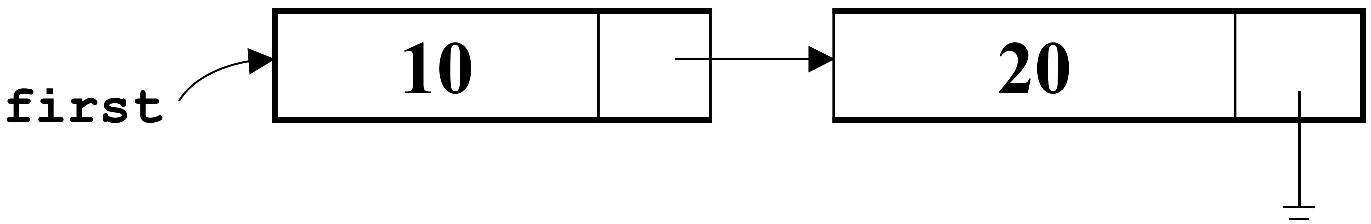
- 鍵結串列有順序的關係，包括至少二種類型的資訊：
  - 內含的元素。
  - 下一項元素所在位置的資訊。
- 鍵結是由指標或位址來構成，比起陣列link的使用更具一般性。

程式：　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;
```

- 在程式第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，可稱之為節點。

範例



程式

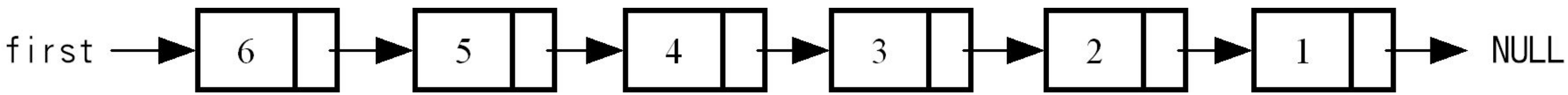
```
struct node
{   int data;
    struct node *next;
};
struct node *first;
void construct()
{   first = (node *) malloc (sizeof(node));   //
    first->data = 10;
    first->next = (node *)  malloc (sizeof(node));
    first->next->data = 20;
    first->next->next = NULL;
}
```

4-2 單向鏈結串列

- 建立和走訪單向鏈結串列
- 刪除單向鏈結串列的節點
- 加入單向鏈結串列的節點

單向鏈結串列-說明

- 單向鏈結串列是最簡單的一種鏈結串列，因為節點指標都是指向同一個方向，依序從前一個節點指向下一個節點，然後最後1個節點指向NULL，所以稱為單向鏈結串列，如下圖所示：



單向鏈結串列-標頭檔

```
/* 程式範例: Ch4-3.h */
struct Node {          /* Node節點結構 */
    int data;          /* 結構變數宣告 */
    struct Node *next; /* 指向下一個節點 */
};
typedef struct Node LNode; /* 串列節點的新型態 */
typedef LNode *List;       /* 串列的新型態 */
List first = NULL;         /* 串列的開頭指標 */
/* 抽象資料型態的操作函數宣告 */
extern void creatList(int len, int *array);
extern int isListEmpty();
extern void printList();
extern List searchNode(int d);
extern int deleteNode(List ptr);
extern void insertNode(List ptr, int d);
```

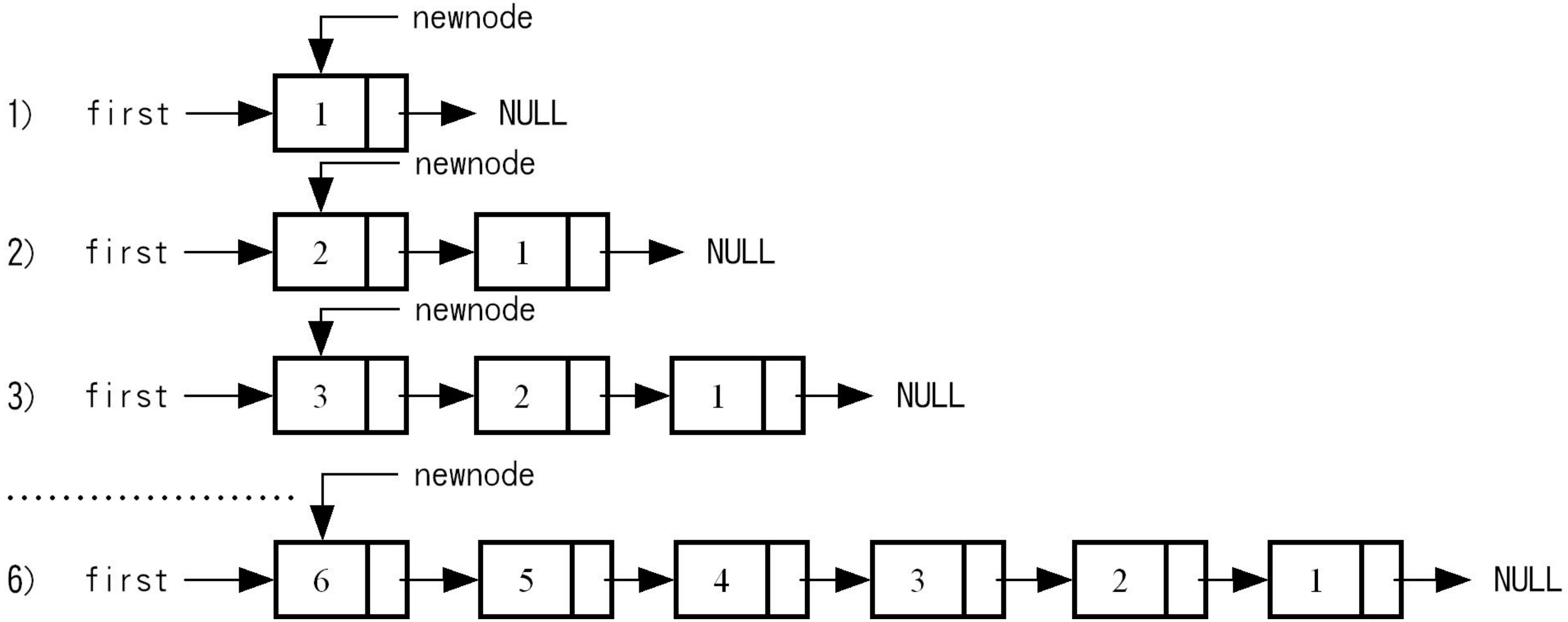
建立和走訪單向鏈結串列-建立(說明)

建立單向鏈結串列

- 在createList()函數是使用for迴圈將取得的陣列值建立成串列節點，每執行一次迴圈，就在串列開頭插入一個新節點，如下所示：

```
for ( i = 0; i < len; i++ ) {
    newnode = (List) malloc(sizeof(LNode));
    newnode->data = array[i];
    newnode->next = first;
    first = newnode;
}
```

建立和走訪單向鏈結串列-建立(圖例)



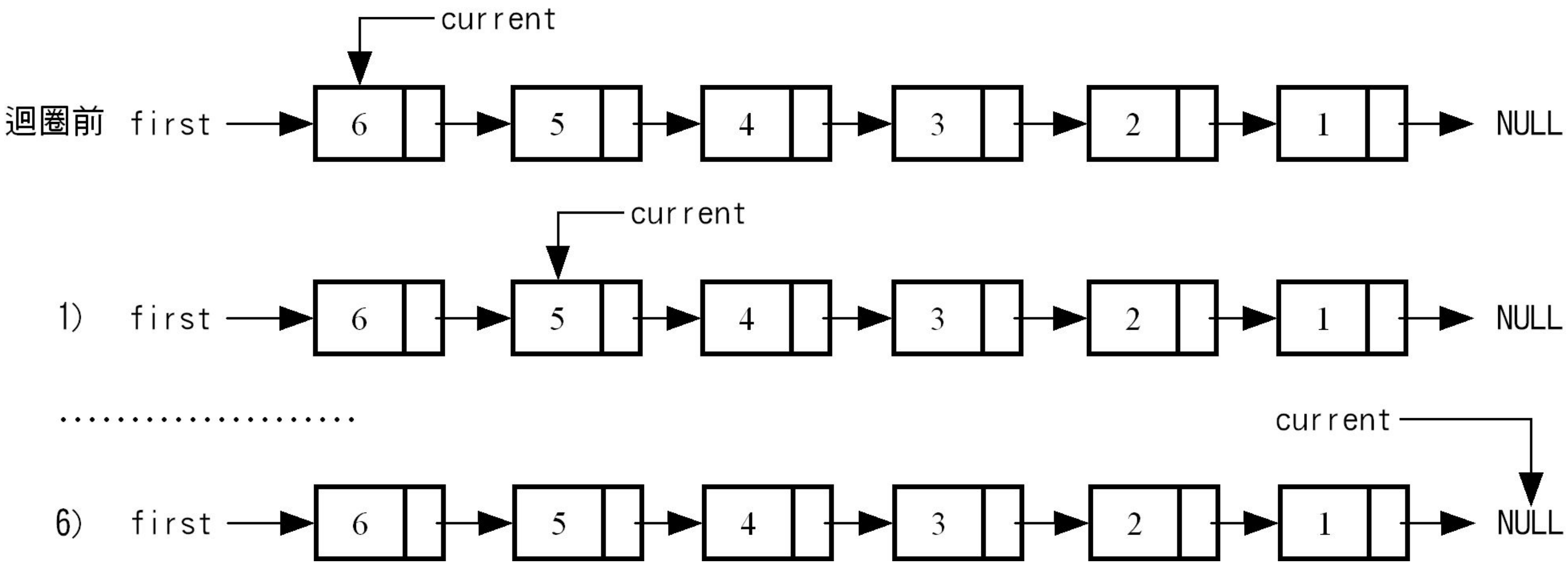
建立和走訪單向鏈結串列-走訪(說明)

單向鏈結串列的走訪

- 單向鏈結串列的「走訪」（Traverse）和一維陣列的走訪十分相似，其差異在於陣列是遞增索引值來走訪陣列，串列是使用指標運算來處理節點的走訪，如下所示：

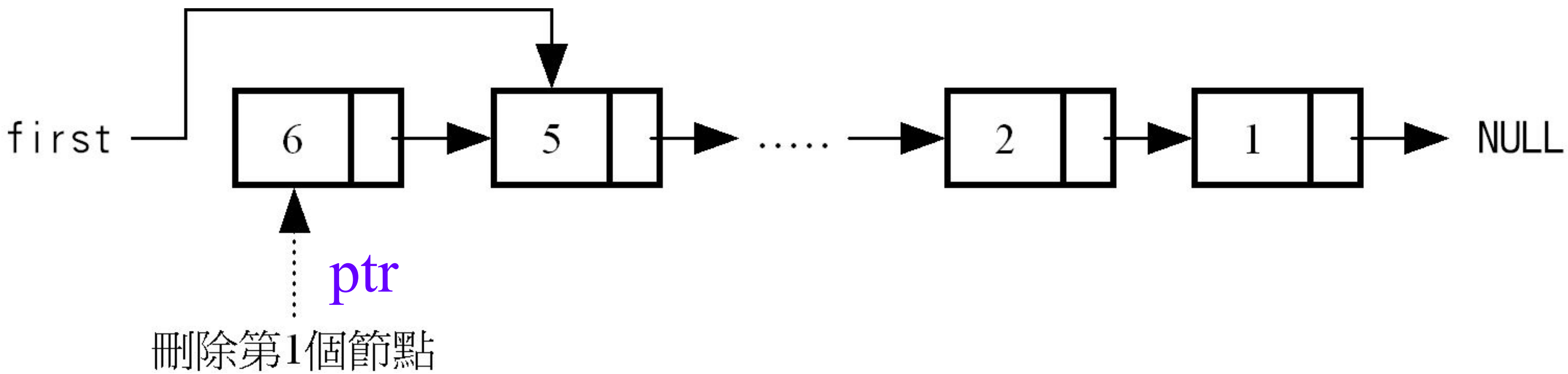
```
List current = first;
while ( current != NULL ) {
    ……………
    current = current->next;
}
```

建立和走訪單向鏈結串列-走訪(圖例)



刪除單向鏈結串列的節點-情況1

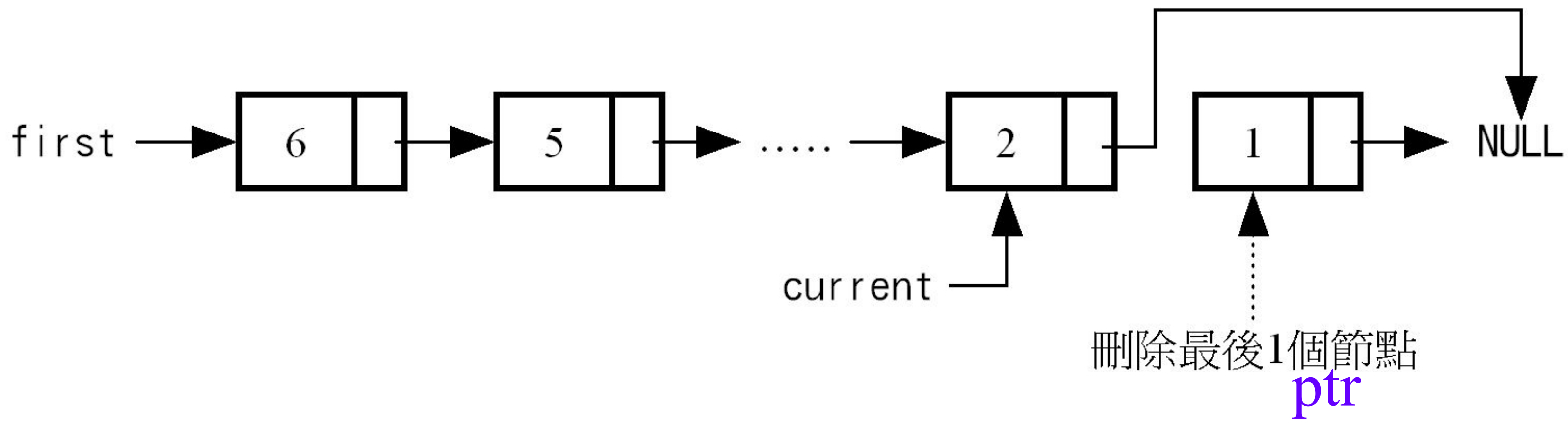
- **刪除串列的第1個節點**：只需將串列指標first指向下一個節點，如下圖所示：




```
ptr = first;
first = first->next;
free(ptr);
```

刪除單向鏈結串列的節點-情況2

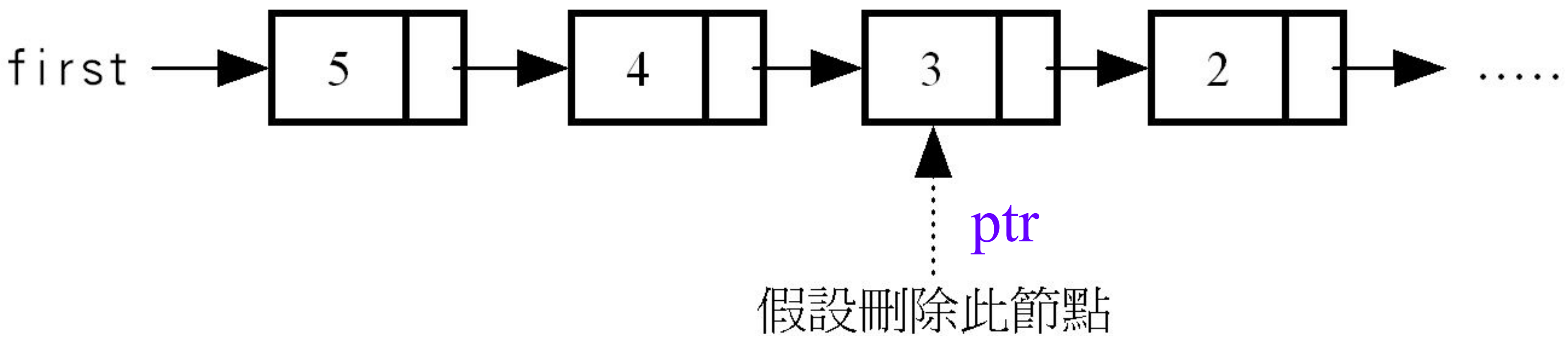
- **刪除串列的最後1個節點**：只需將最後1個節點ptr的前一個節點指標指向NULL，如下圖所示：




```
while (current->next!=ptr)
   current = current->next;
free(ptr);
current->next = NULL;
```

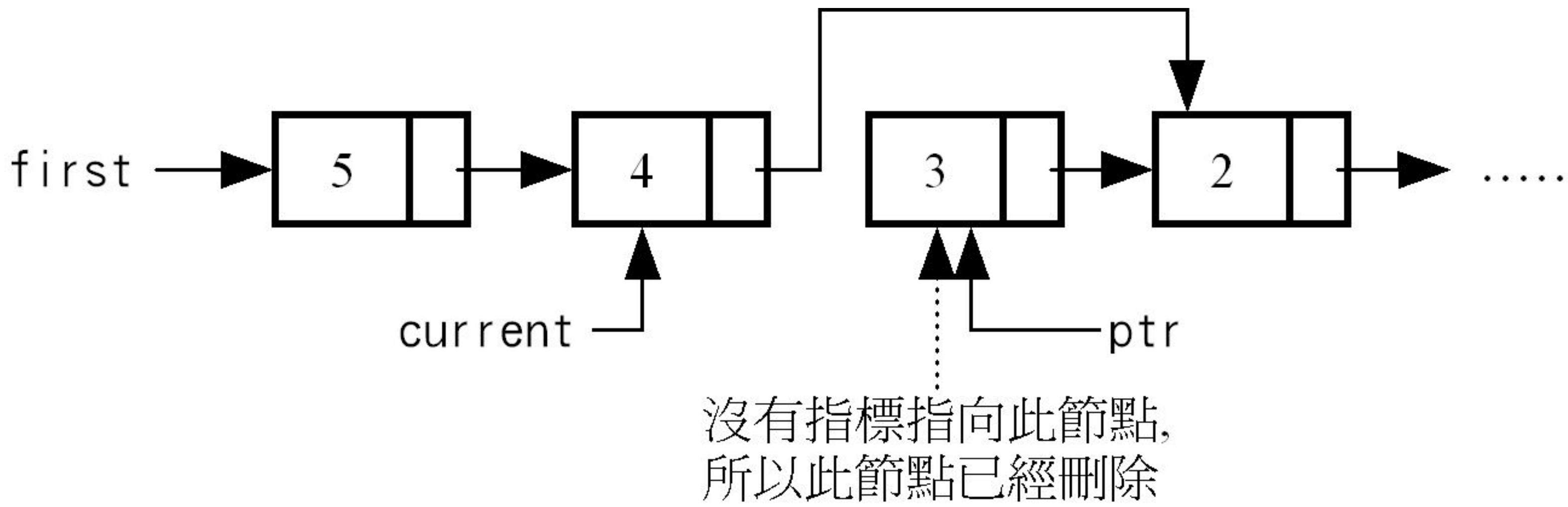
刪除單向鏈結串列的節點-情況3(1)

- **刪除串列的中間節點**：將刪除節點前一個節點的next指標，指向刪除節點下一個節點，例如：刪除節點3，如下圖所示：



刪除單向鏈結串列的節點-情況3(2)

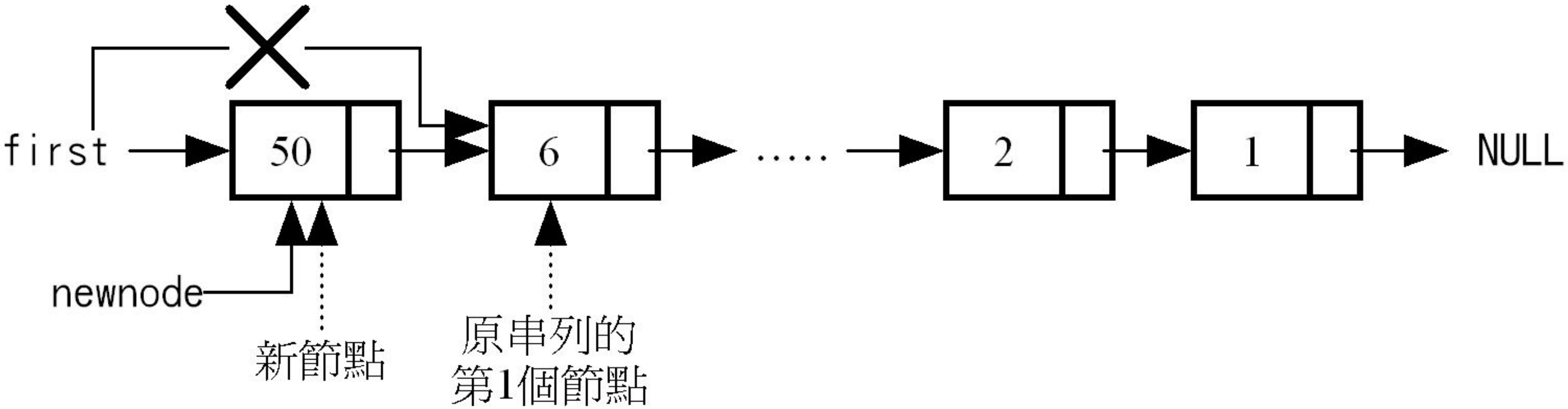
- 在執行刪除節點3操作後的串列圖形，如下所示：




```
while (current->next!=ptr)
  current = current->next;
current->next = ptr->next;
free(ptr);
```

加入單向鏈結串列的節點-情況1

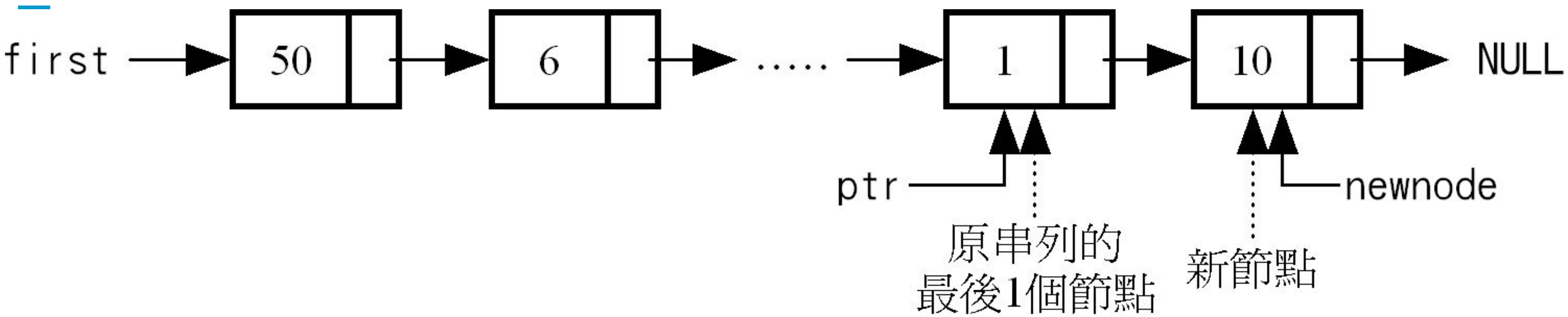
- **將節點插入串列第1個節點之前**：只需將新節點newnode的指標指向串列的第1個節點first，新節點就成為串列的第1個節點，如下圖所示：




```
newnode->next = first;
first = newnode;
```

加入單向鏈結串列的節點-情況2

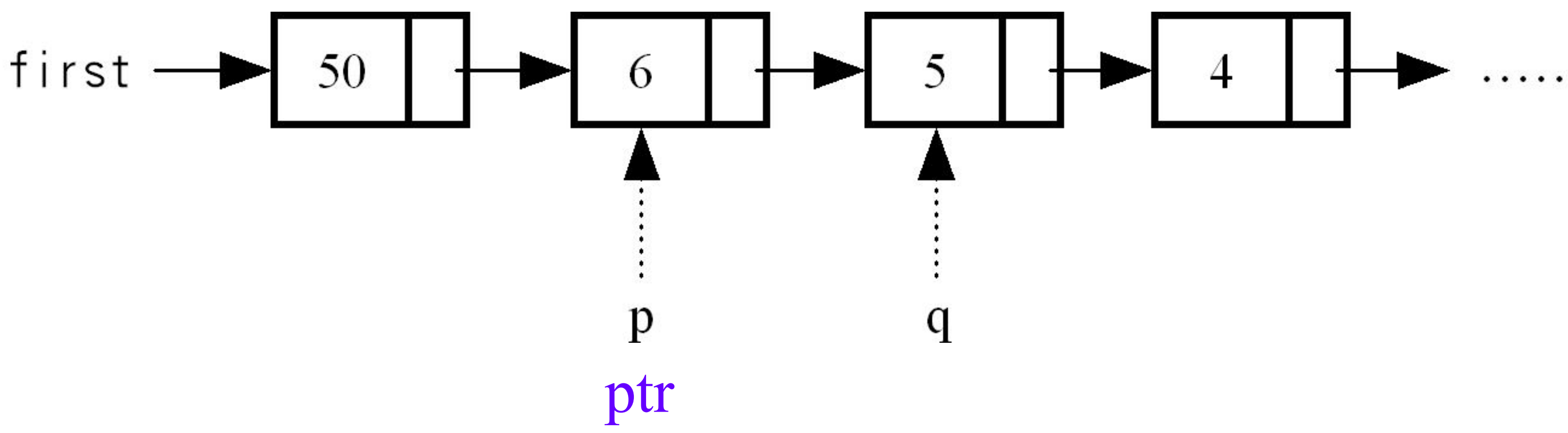
- **將節點插在串列的最後1個節點之後**：只需將原來串列最後1個節點的指標指向新節點newnode，新節點指向NULL，如下圖所示：




```
ptr->next = newnode;
newnode->next = NULL;
```

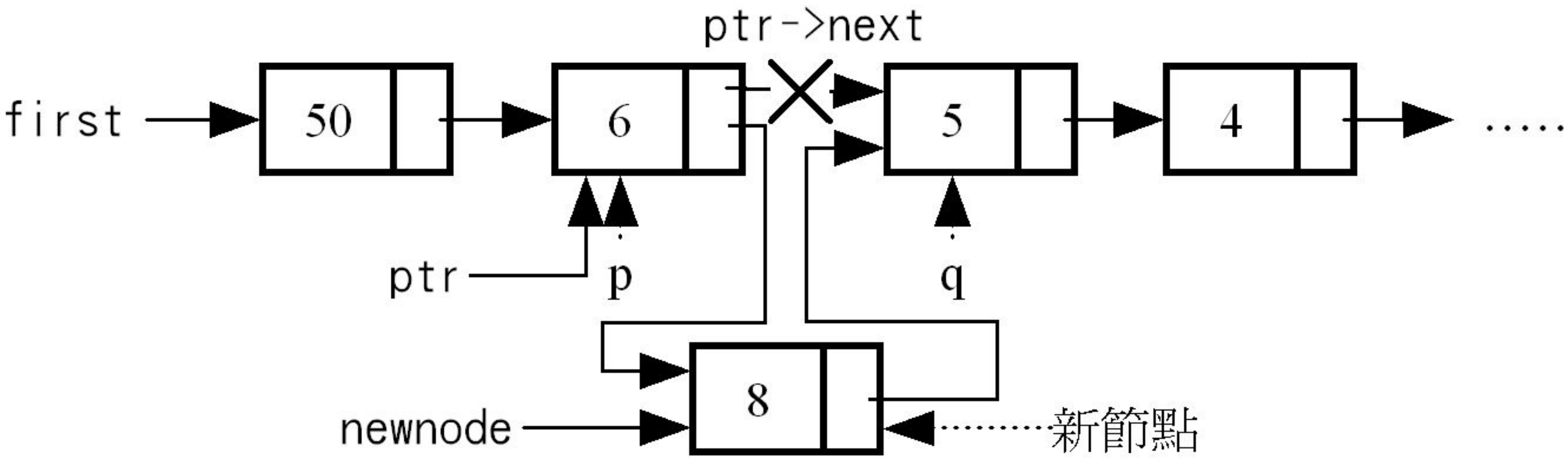
加入單向鏈結串列的節點-情況3(1)

- **將節點插在串列的中間位置**：假設節點是插在ptr和q兩個節點之間，ptr是q的前一個節點，如下圖所示：



加入單向鏈結串列的節點-情況3(2)

- 只需將新節點指標指向q (ptr->next)，然後將 ptr->next指標指向新節點 newnode，就可以插入新節點，如下圖所示：




```
newnode->next=ptr->next;
ptr->next = newnode;
```

反向串接一串列

「鍵結串列的指標反向連接」將使原來串列內資料的順序完全對調。

程式4-7反向串接一串列

```
struct  Node
{   int data;
    struct Node *link;
}
struct Node *first;
void Invert ( )
{   struct  Node *r， *s， *t ;
    r = first;
    s = NULL;
        while  (r != NULL)
        {    t = s;
             s = r;
             r = r->link;
             s->link = t;
        }
        first = s;
   }
```

r: 指向目前處理之node
→準備指向下一個node

t: 指向目前處理node(r 指向) 的前一個node

s: 將目前處理之node，改指向前一個node(t 指向)

串接兩個鏈接串列

程式：串接兩個鏈接串列

```
struct Node
{ int data;
  struct Node *link;
}
struct Node *a_first, *b_first, first;
struct Node * Concatenate
(struct Node * a_first, struct Node * b_first);
{  struct Node *p ;
   if (a_first == NULL)
   resturn b_first ;
      else
      { for(p=first; p->link!=NULL; p=p->link);
        p→link = b_first ;
        return a_first ;
      }
   }
```

4.2 SINGLY LINKED LISTS

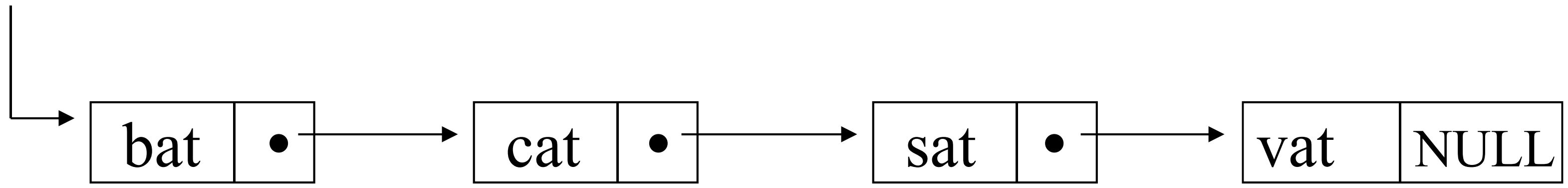


***Figure 4.2:** Usual way to draw a linked list (p.147)*

Insertion

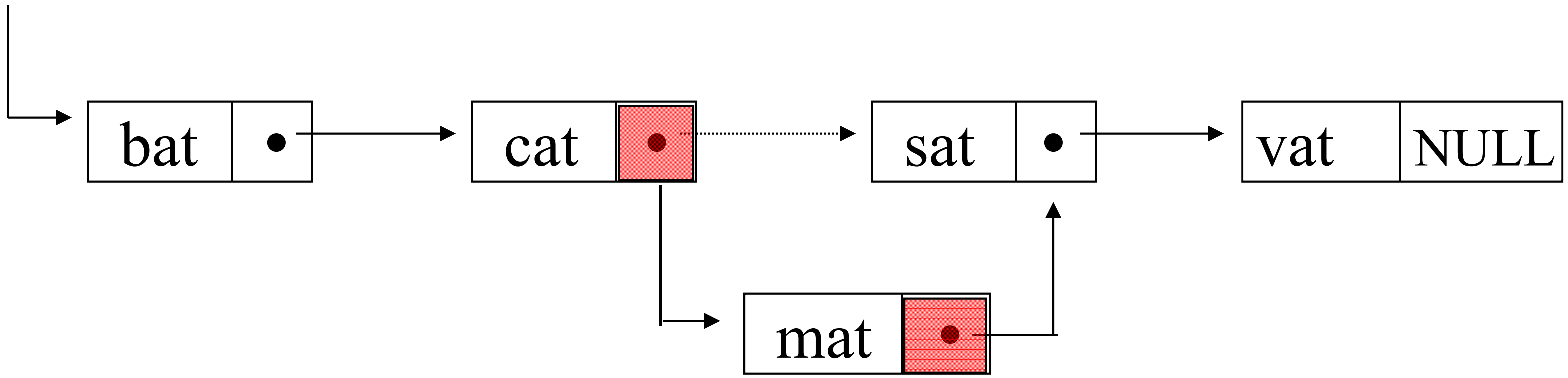


***Figure 4.3:** Insert mat after cat (p.148)*

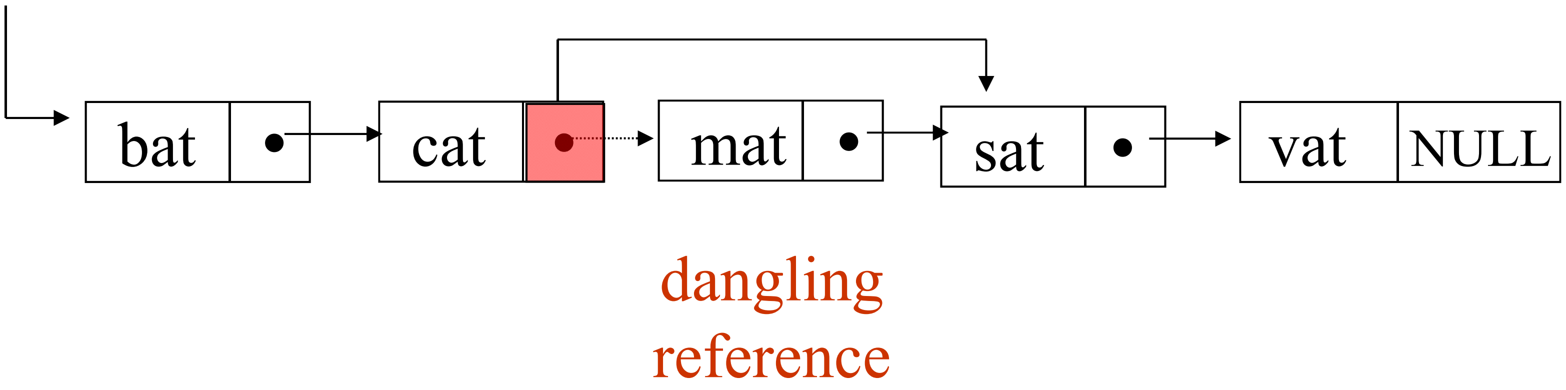


***Figure 4.4:** Delete *mat* from list (p.149)

Example 4.1: create a linked list of words

Declaration

```
typedef struct list_node, *list_pointer;
typedef struct list_node {
          char data [4];
          list_pointer link;
          };
```

Creation

```
list_pointer ptr =NULL;
```

Testing

```
#define IS_EMPTY(ptr) (!(ptr))
```

Allocation

```
ptr=(list_pointer) malloc (sizeof(list_node));
```

e -> name ⇨ (*e).name

strcpy(ptr -> data, “bat”);

ptr -> link = NULL;

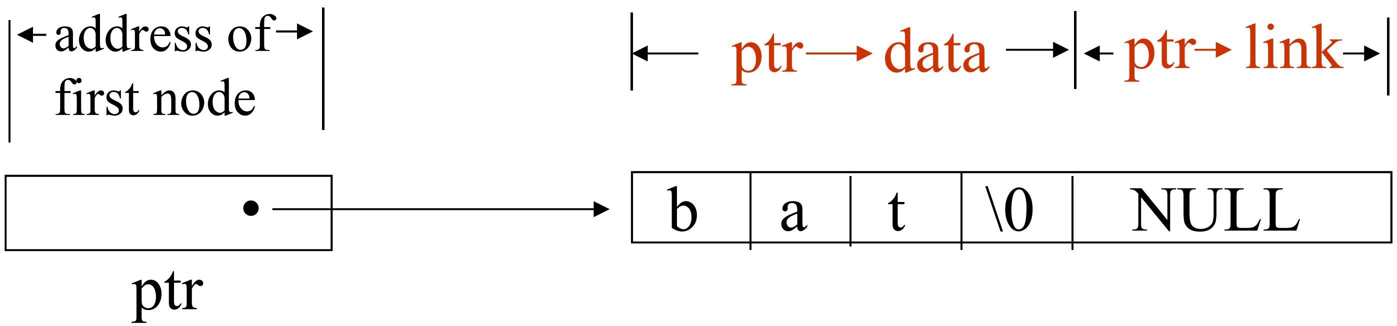
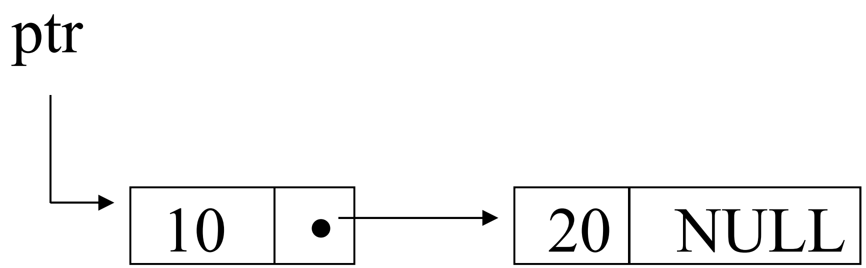


***Figure 4.5:**Referencing the fields of a node(p.151)

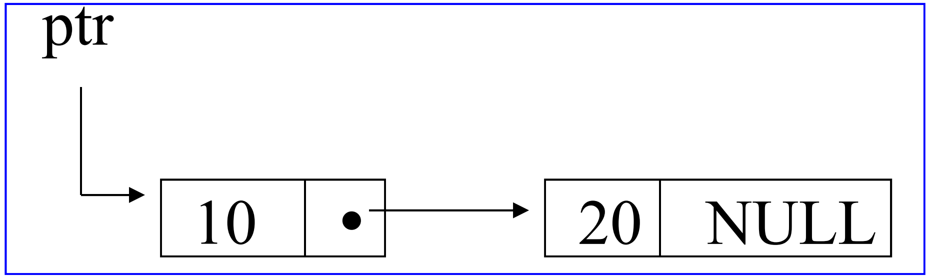
Example: create a two-node list




```
typedef struct list_node *list_pointer;
typedef struct list_node {
        int data;
        list_pointer link;
        };
list_pointer ptr =NULL
```

Example 4.1: (p.149)

```
list_pointer create2( )
{
/* create a linked list with two nodes */
    list_pointer first, second;
    first = (list_pointer) malloc(sizeof(list_node));
    second = ( list_pointer) malloc(sizeof(list_node));
    second -> link = NULL;
    second -> data = 20;
    first -> data = 10;
    first ->link = second;
    return first;
}
```



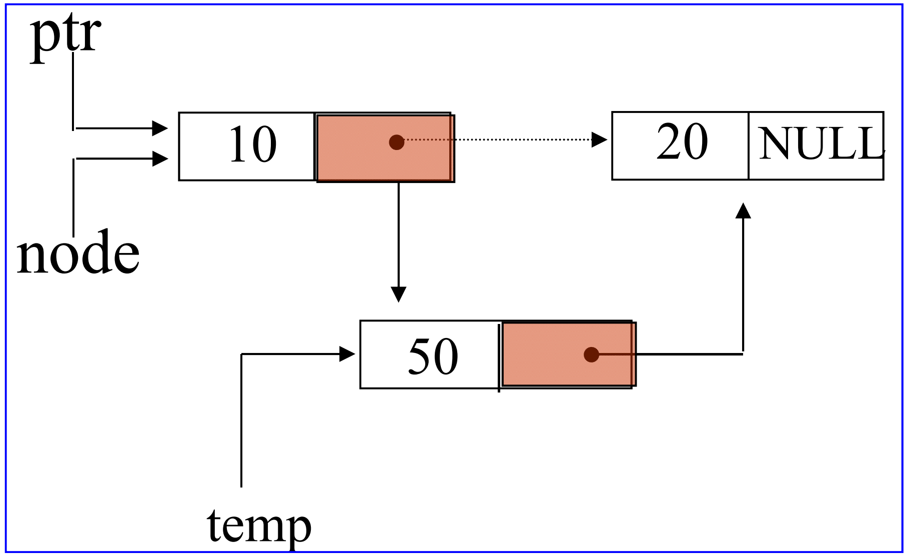
***Program 4.1:**Create a tow-node list (p.152)

List Insertion:

Insert a node after a specific node

```
void insert(list_pointer *ptr, list_pointer node)
{
/* insert a new node with data = 50 into the list ptr after node */
   list_pointer temp;
   temp = (list_pointer) malloc(sizeof(list_node));
   if (IS_FULL(temp)){
      fprintf(stderr, "The memory is full\n");
      exit (1);
   }
```

```
temp->data = 50;
if (*ptr) {  noempty list
    temp->link =node ->link;
    node->link = temp;
}
else {    empty list
   temp->link = NULL;
   *ptr =temp;
 }
}
```



<u>***Program 4.2:**Simple insert into front of list (p.153)</u>

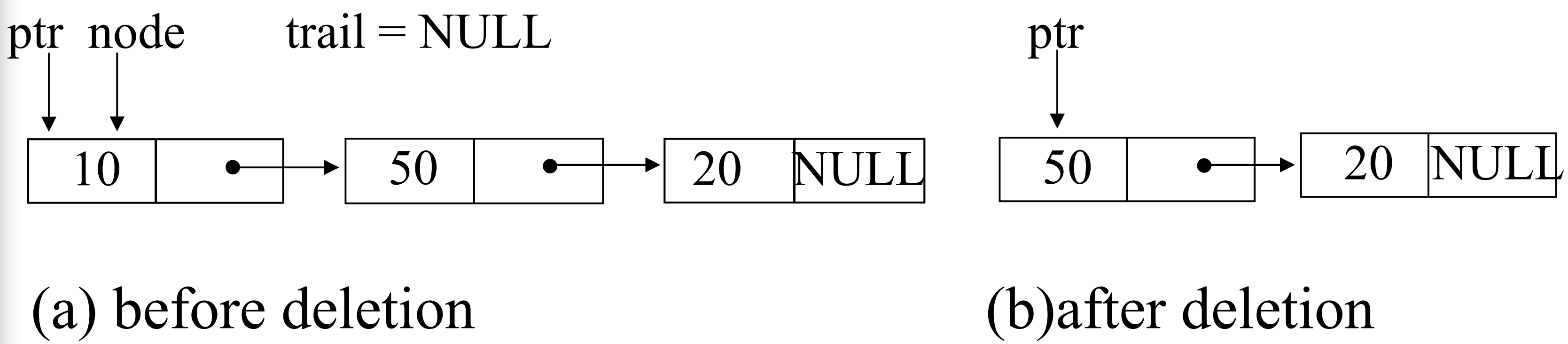
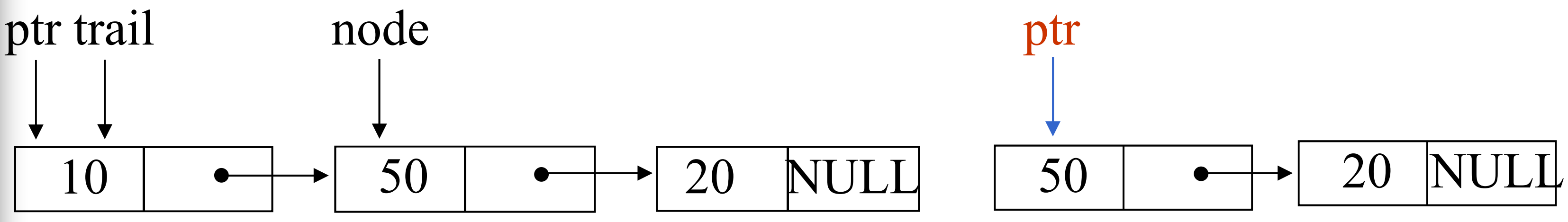


*Figure 4.8:** List after the function call *Delete*(*&ptr,NULL.ptr*);(p.154)

List Deletion

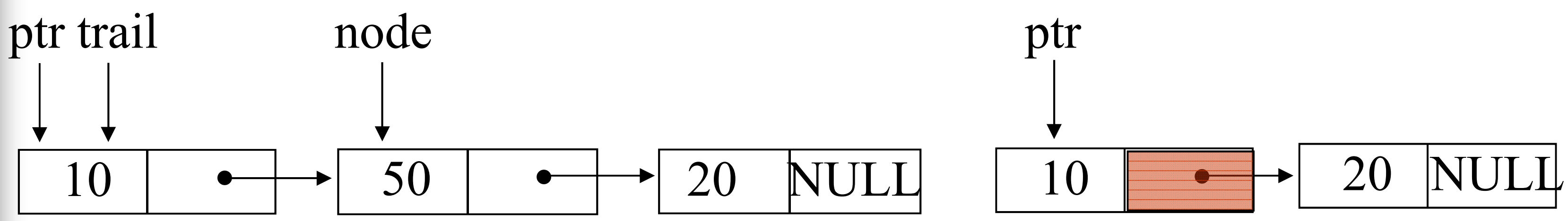
Delete the first node.



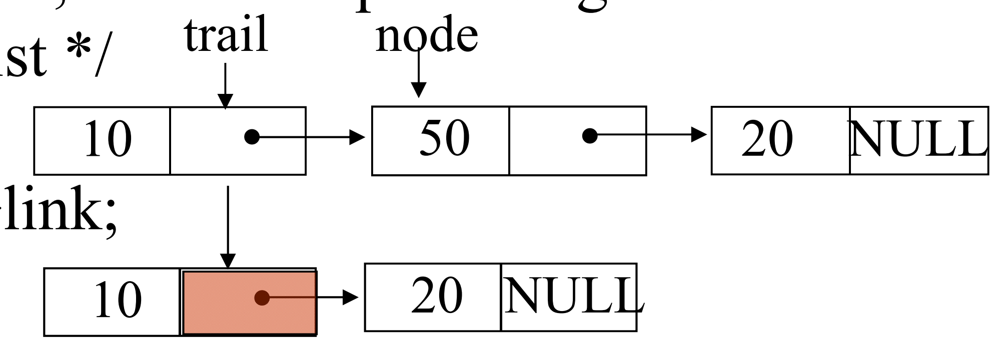
(a) before deletion

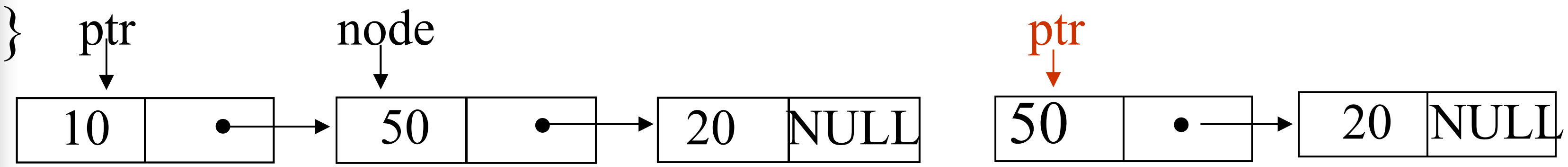
(b)after deletion

Delete the node other than the first node.



```
void delete(list_pointer *ptr, list_pointer trail,
                                        list_pointer node)
{
/* delete node from the list, trail is the preceding node
   ptr is the head of the list */
    if (trail)
       trail->link = node->link;
    else
       *ptr = (*ptr) ->link;
     free(node);
}
```





Print out a list (traverse a list)

```
void print_list(list_pointer ptr)
{
   printf("The list ocntains: ");
   for ( ; ptr; ptr = ptr->link)
     printf("%4d", ptr->data);
   printf("\n");
 }
```

***Program 4.4:** Printing a list (p.155)

作業

- Pp.154: ex6

4.3 DYNAMICALLY LINKED STACKS AND QUEUES

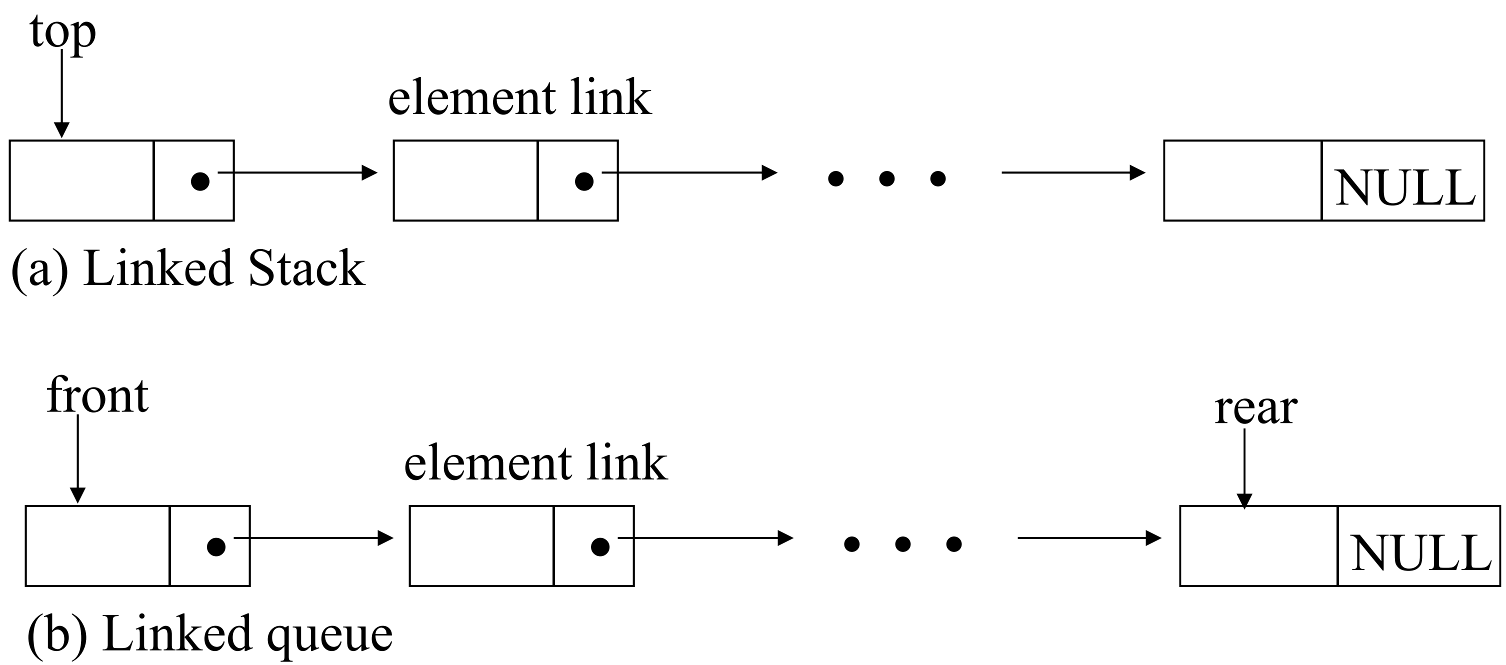


***Figure 4.11:** Linked Stack and queue (p.157)*

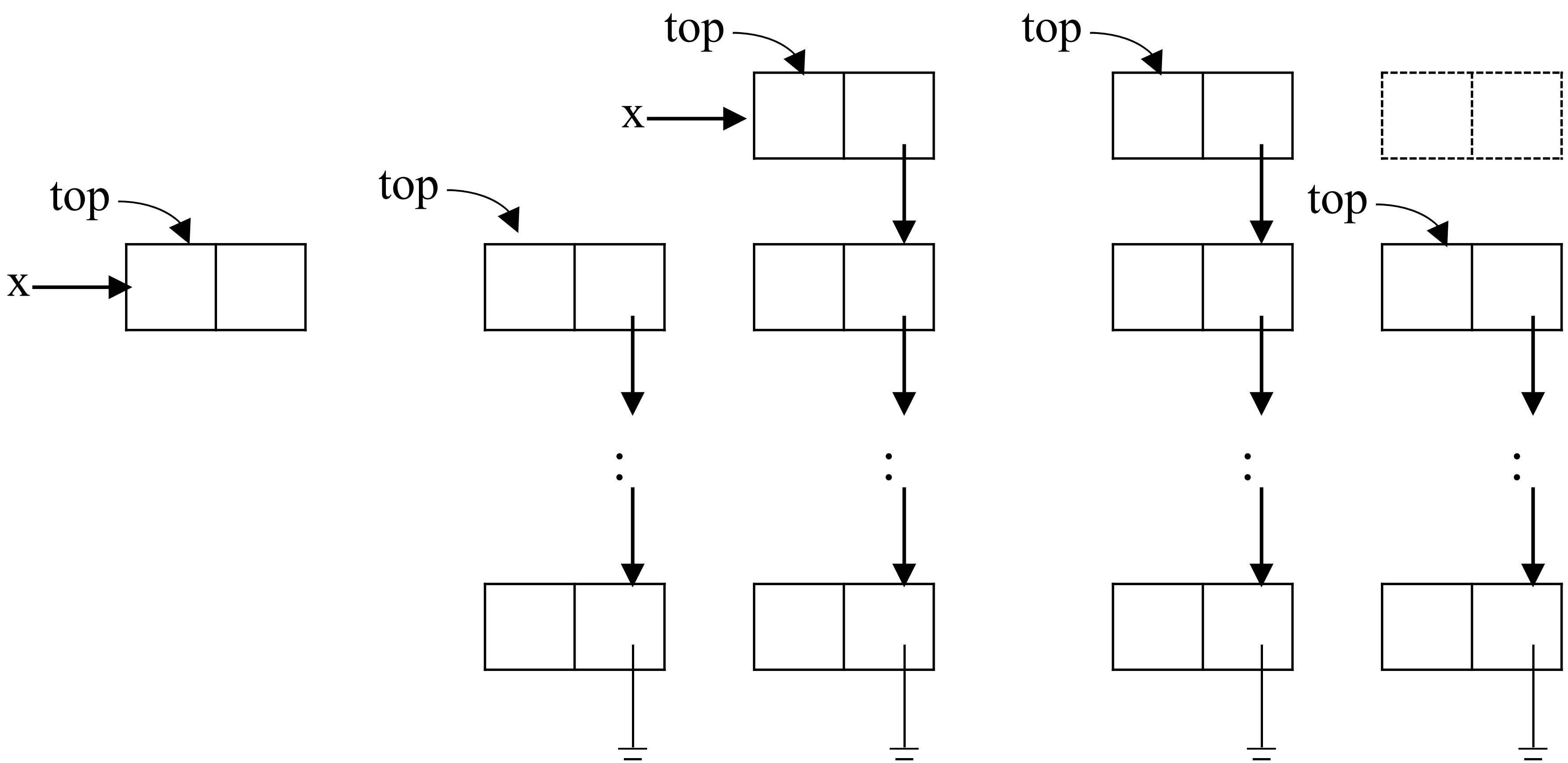
鍵結堆疊

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。改用動態配置的鍵結堆疊，即可解決使用陣列造成的缺點。

程式 鍵結堆疊

```
Struct  StackNode
{  int data;
   struct StackNode *link;
}
struct StackNode *top;
```

鍵結堆疊執行push和pop的過程



(a) push x至空堆疊　(b) push x至堆疊　(c) pop堆疊

程式鍵結堆疊(續)

```
struct Stacknode *NewNode(int element)
{  struct StackNode *p;
   p = (struct StackNode *)
        malloc (sizeof(StacdNode));
   p->data = elemeut;
        p->link = NULL;
        return p;
   }
   void PushStack(int element)
   {    struct StackNode *x ;
        x = NewNode(element);
        if (top==NULL) top = x;
        else
         {    x->link = top;
              top = x;
         }
   }
```

Represent n stacks

```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
        int key;
        /* other fields */
       } element;
typedef struct stack *stack_pointer;
typedef struct stack {
        element item;
        stack_pointer link;
        };
stack_pointer top[MAX_STACKS];
```

Push in the linked stack

```
void add(stack_pointer *top, element item)
{
  /* add an element to the top of the stack */
  stack_pointer temp =
                  (stack_pointer) malloc (sizeof (stack));
  if (IS_FULL(temp)) {
    fprintf(stderr, " The memory is full\n");
    exit(1);
    }
    temp->item = item;
    temp->link = *top;
    *top= temp;
}
```

***Program 4.6:**Add to a linked stack (p.149)

pop from the linked stack

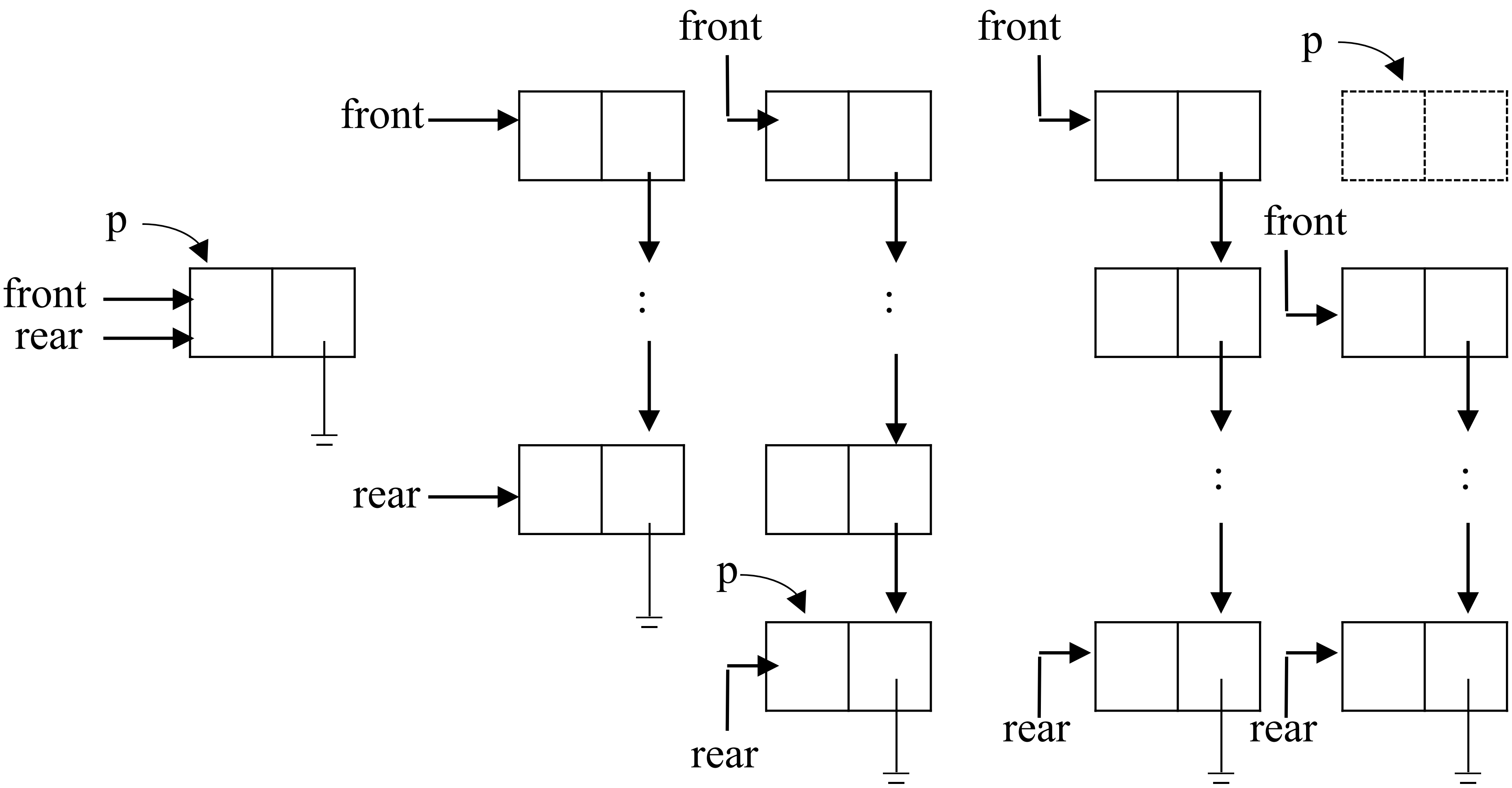
```
element delete(stack_pointer *top) {
/* delete an element from the stack */
    stack_pointer temp = *top;
    element item;
    if (IS_EMPTY(temp))  {
      fprintf(stderr,  “The stack is empty\n”);
      exit(1);
    }
    item = temp->item;
    *top = temp->link;
     free(temp);
     return item;
}
```

***Program 4.7**: Delete from a linked stack (p.149)

Represent n queues

```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queue_pointer;
typedef struct queue {
        element item;
        queue_pointer link;
        };
queue_pointer front[MAX_QUEUE], rear[MAX_QUEUES];
```

鏈結佇列執行新增和刪除的過程



(a) 新增節點至空佇列 **(b) 新增節點至佇列** **(c) 刪除佇列節點**

鏈結佇列

程式鏈結佇列

```
struct QueueNode
{   char data；
    struct QueueNode *next；
}
struct QueueNode *front, *rear；
struct QueueNode * NewQNode (char element)
{   struct QueueNode *p;
    p = (struct QueueNode *)
         malloc (sizeof(QueueNode ));
    p->data = element;
         P->next = NULL;
    }
    void AddQueue(char element)
    {    struct QueueNode *p;
         p = NewQNode(element);
         if (rear == NULL)
              front = p;
         else
```

程式結佇列(續)

```
            rear->next = p;
        rear = p;
    }
    char DeleteQueue()
    {   struct QueueNode *p;
        char element;
        if (front == NULL)
        {   QueueEmpty( );
            return ″#″;
        }
        else
        {   p = front;
            front = front->next;
            element = p->data;
            free(p);
            return element;
        }
    }
```

enqueue in the linked queue

```
void addq(queue_pointer *front, queue_pointer *rear, element item)
{  /* add an element to the rear of the queue */
  queue_pointer temp =
                  (queue_pointer) malloc(sizeof (queue));
  if (IS_FULL(temp)) {
    fprintf(stderr, " The memory is full\n");
    exit(1);
    }
    temp->item = item;
    temp->link = NULL;
    if (*front) (*rear) -> link = temp;
    else *front = temp;
    *rear = temp;   }
```

dequeue from the linked queue (similar to push)

```
element deleteq(queue_pointer *front) {
/* delete an element from the queue */
    queue_pointer temp = *front;
    element item;
    if (IS_EMPTY(*front))  {
       fprintf(stderr, “The queue is empty\n”);
       exit(1);
    }
    item = temp->item;
    *front = temp->link;
     free(temp);
     return item;
}
```

Polynomials

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

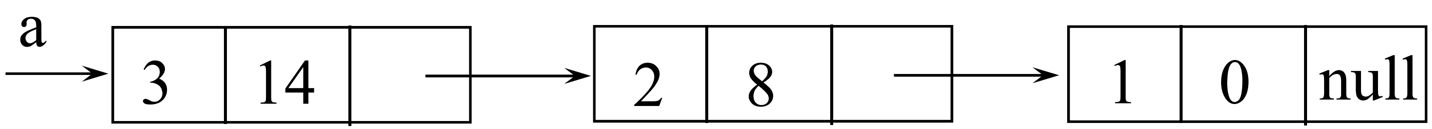
Representation

```
typedef struct poly_node *poly_pointer;
typedef struct poly_node {
    int coef;
    int expon;
    poly_pointer link;
};
poly_pointer a, b, c;
```

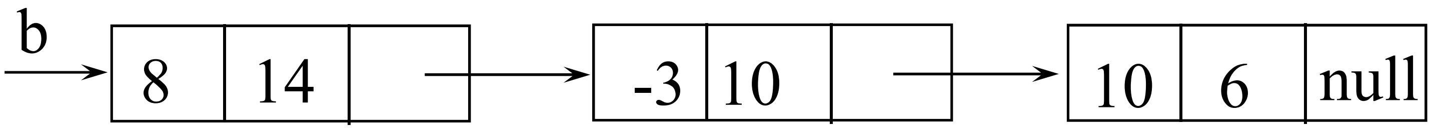
coef	expon	link

Examples

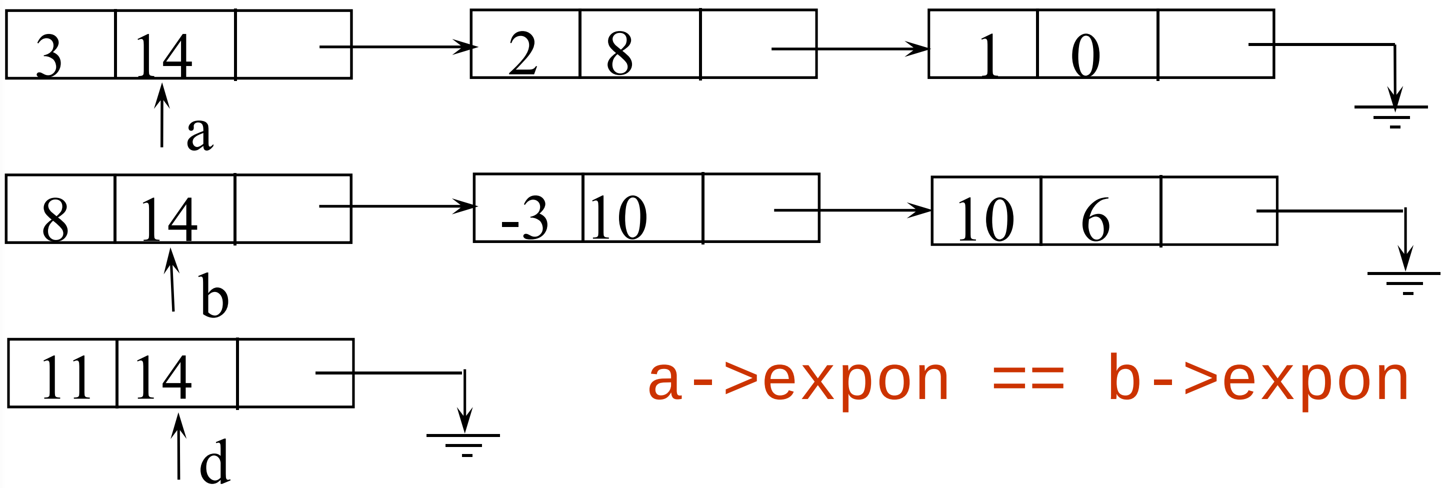
$$a = 3x^{14} + 2x^{8} + 1$$



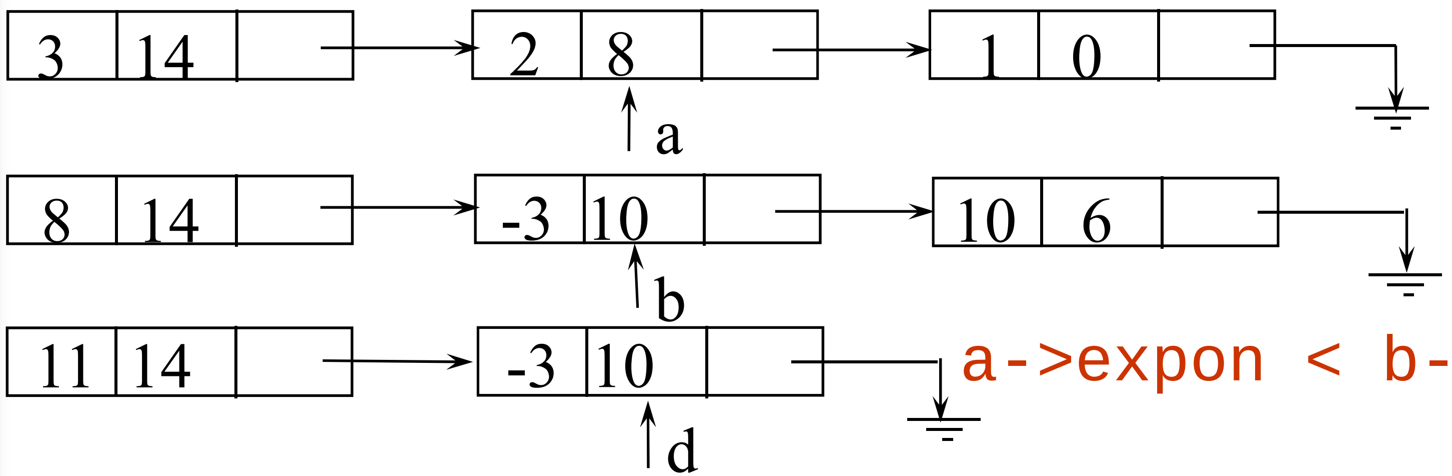
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Adding Polynomials

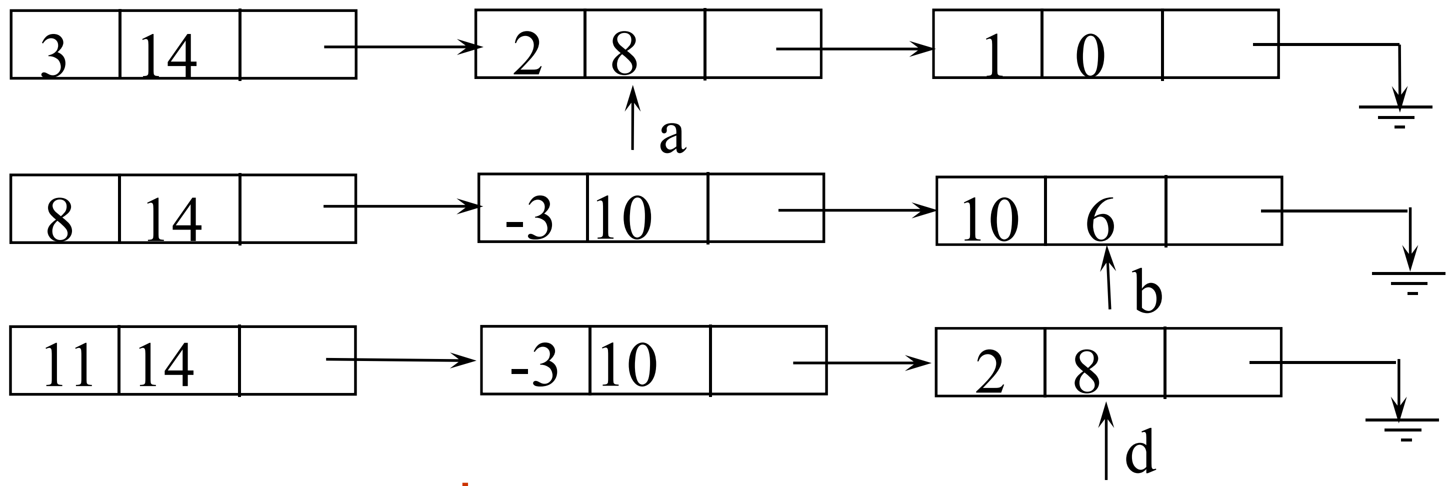


a->expon == b->expon



a->expon < b->expon

Adding Polynomials (*Continued*)



a->expon > b->expon

Alogrithm for Adding Polynomials

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
    poly_pointer front, rear, temp;
    int sum;
    rear =(poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(rear)) {
        fprintf(stderr, “The memory is full\n”);
        exit(1);
    }
    front = rear;
    while (a && b) {
        switch (COMPARE(a->expon, b->expon)) {
```

```
            case -1: /* a->expon < b->expon */
                attach(b->coef, b->expon, &rear);
                b= b->link;
                break;
            case 0: /* a->expon == b->expon */
                sum = a->coef + b->coef;
                if (sum) attach(sum,a->expon,&rear);
                a = a->link;    b = b->link;
                break;
            case 1: /* a->expon > b->expon */
                attach(a->coef, a->expon, &rear);
                a = a->link;
        }
    }
    for (; a; a = a->link)
        attach(a->coef, a->expon, &rear);
    for (; b; b=b->link)
        attach(b->coef, b->expon, &rear);
    rear->link = NULL;
    temp = front;  front = front->link;  free(temp);
    return front;
}
```

Delete extra initial node.

Analysis

(1) coefficient additions

$0 \le \text{additions} \le \min(m, n)$

where m (n) denotes the number of terms in A (B).

(2) exponent comparisons

extreme case

$e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \ldots > e_0 > f_0$

m+n-1 comparisons

(3) creation of new nodes

extreme case

m + n new nodes

summary $O(m+n)$

Attach a Term

```
void attach(float coefficient, int exponent,
            poly_pointer *ptr)
{
/* create a new node attaching to the node pointed to
   by ptr. ptr is updated to point to this new node. */
    poly_pointer temp;
    temp = (poly_pointer) malloc(sizeof(poly_node));
    if (IS_FULL(temp)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    temp->coef = coefficient;
    temp->expon = exponent;
    (*ptr)->link = temp;
    *ptr = temp;
}
```

A Suite for Polynomials

$e(x) = a(x) * b(x) + d(x)$

```
poly_pointer a, b, d, e;
...
a = read_poly();
b = read_poly();
d = read_poly();
temp = pmult(a, b);
e = padd(temp, d);
print_poly(e);
```

```
read_poly()
print_poly()
padd()
psub()
pmult()
```

temp is used to hold a partial result. By returning the nodes of temp, we may use it to hold other polynomials

Erase Polynomials

```
void earse(poly_pointer *ptr)
{
/* erase the polynomial pointed to by ptr */
    poly_pointer temp;
    while (*ptr) {
        temp = *ptr;
        *ptr = (*ptr)->link;
        free(temp);
    }
}
```

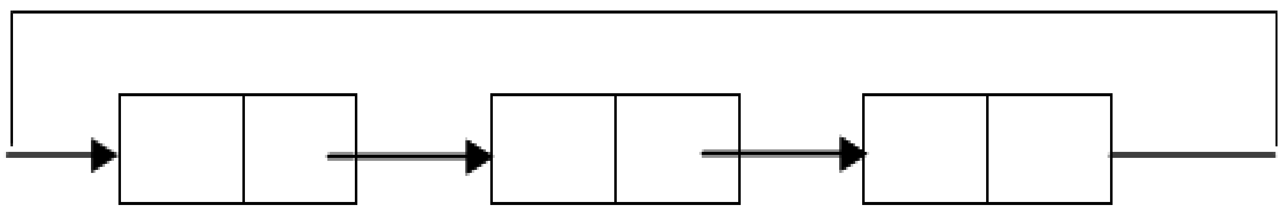
O(n)

環狀鏈結串列

(Circular Linked Lists)

定義

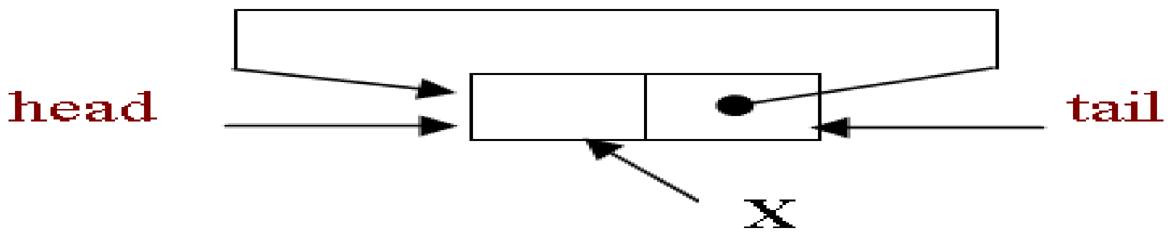
- 一個鏈結串列之最後一個節點指向鏈結串列之最前端，則形成一個環狀鏈結串列，如下圖所示：



基本運算與圖解

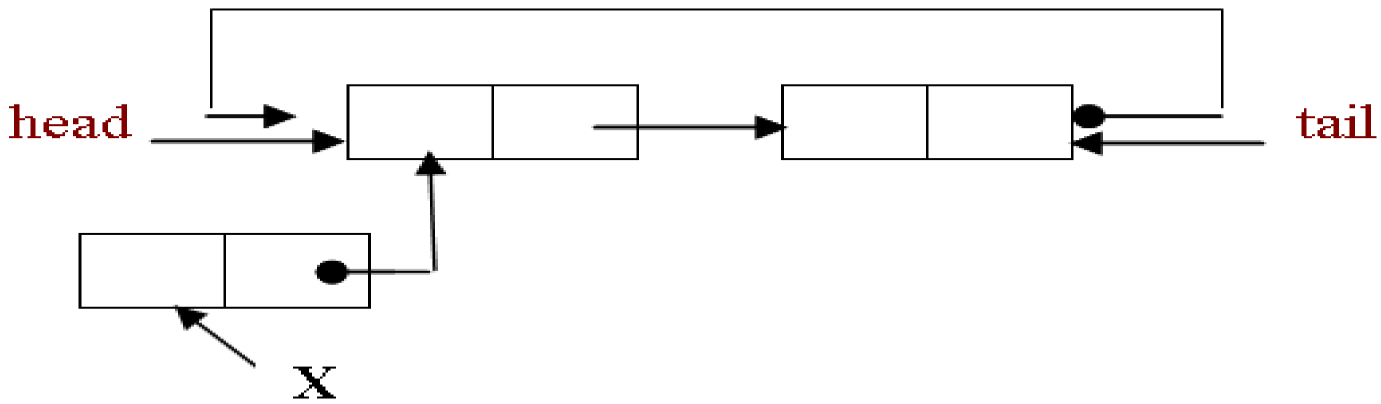
加入動作

- 加入節點於前端
  - 當head = = NULL時
  - head = x; tail = x;

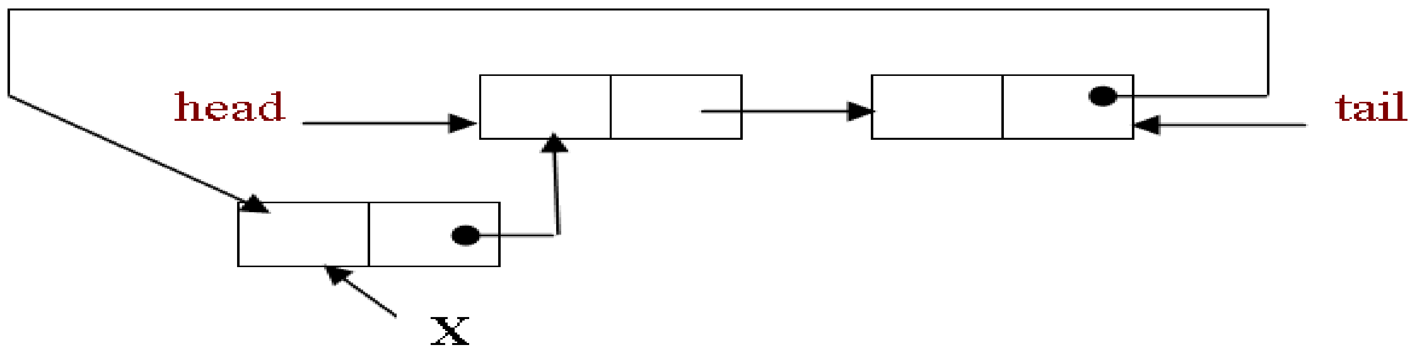


– 當head != NULL時

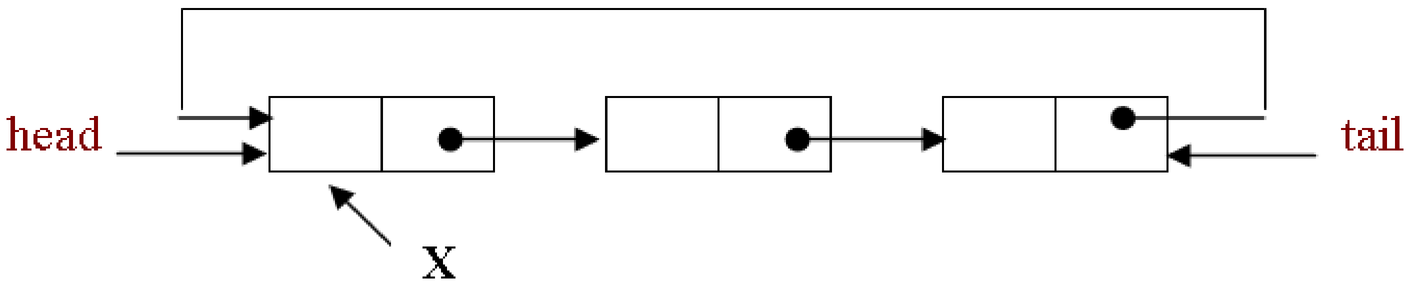
– x → next = head;



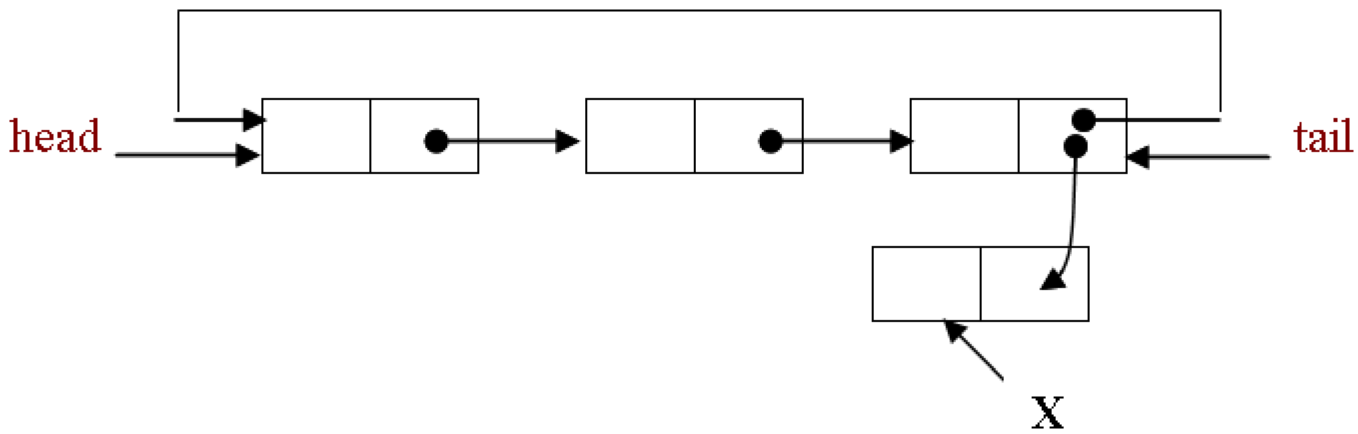
- tail → next = x;



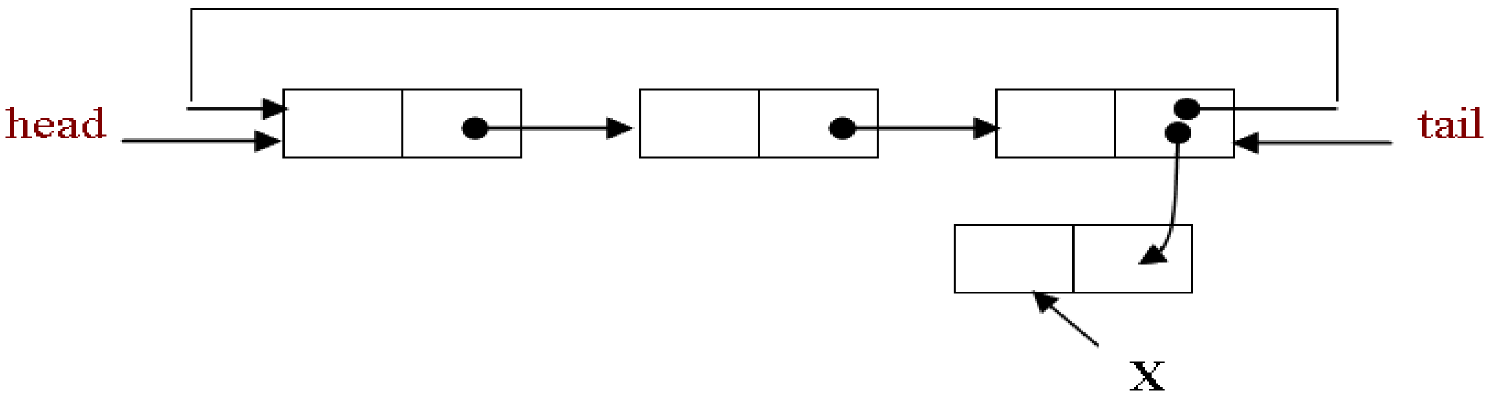
head= x;



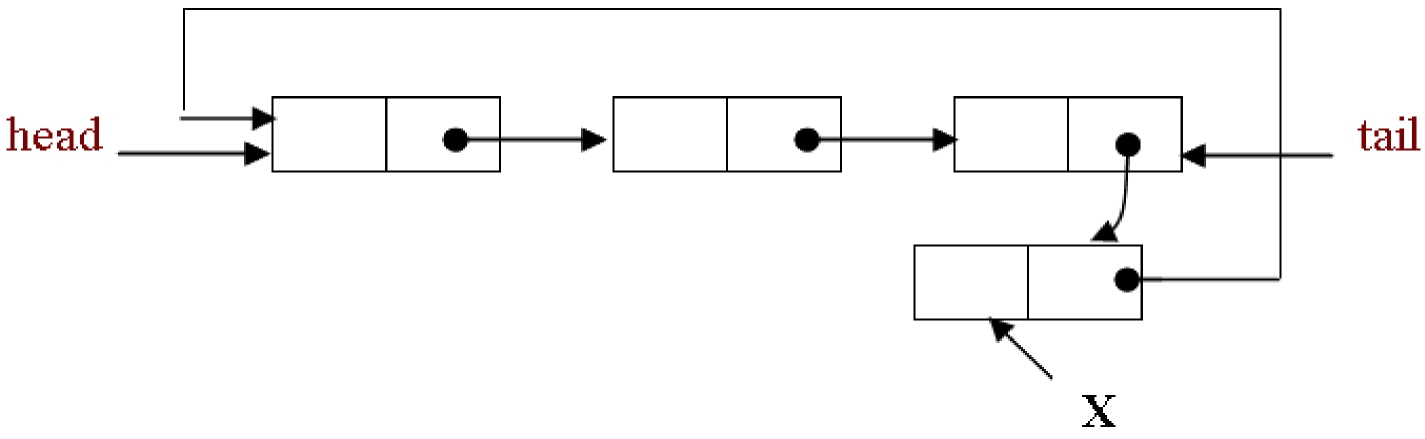
- 加入節點於尾端
  - tail → next = x;



– $x \rightarrow next = head;$



– $tail = x;$



刪除動作

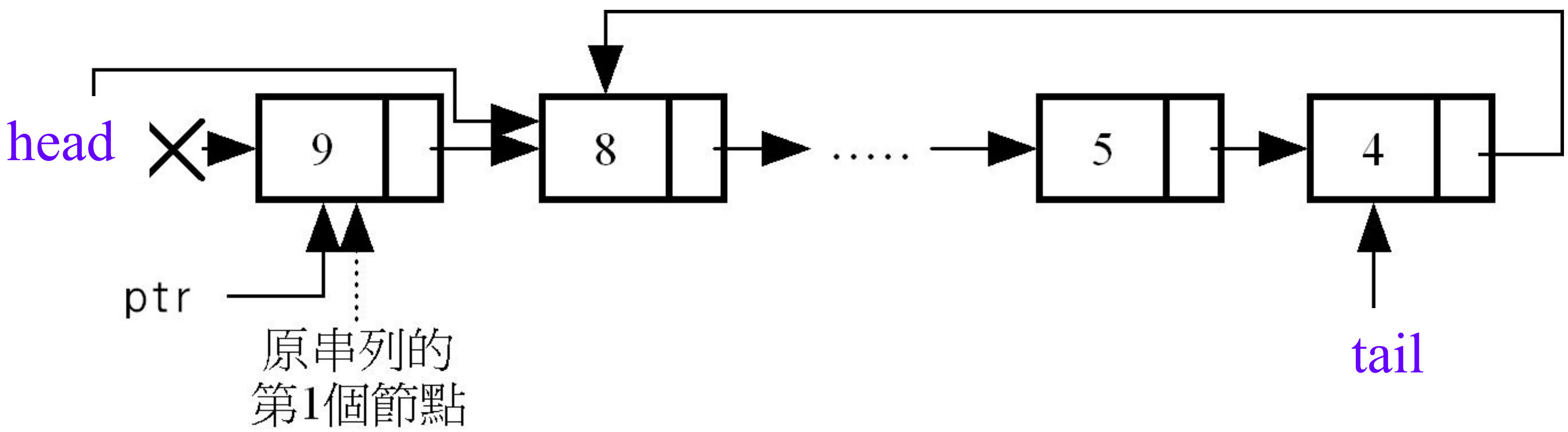
刪除節點於前端

tail → next = head → next;

ptr = head;

head =head → next ;

free(ptr);



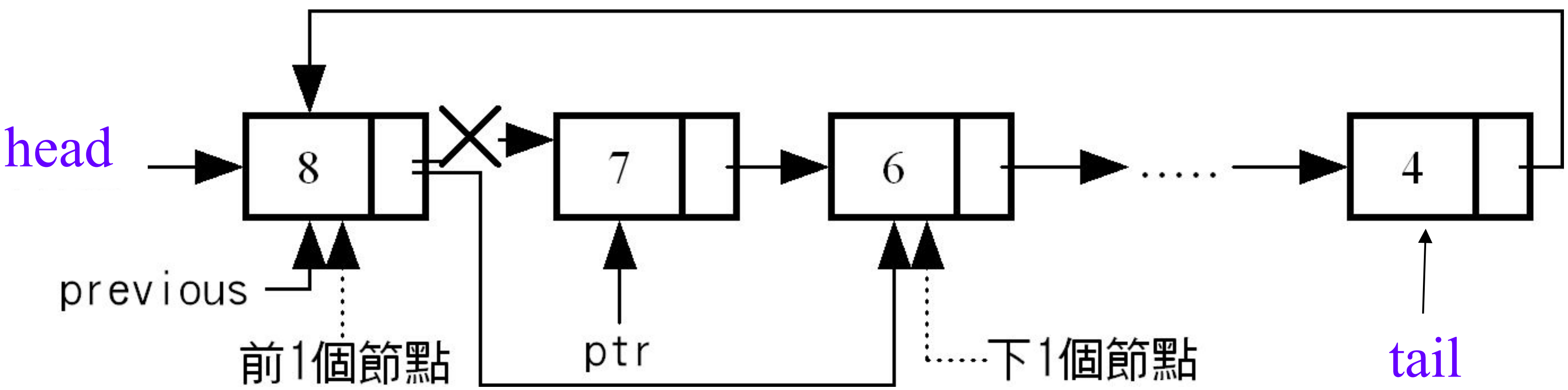
刪除動作

- 刪除環狀串列的中間節點，例如：刪除節點 ptr 分成二個步驟，如下所示：
  - Step 1：先找到節點 ptr 的前一個節點previous。

```
previous = head;
while ( previous->next != ptr )
        previous = previous->next;
```

  - Step 2：將前節點的指標指向節點 ptr 的下節點。

```
previous->next = ptr->next;
free(ptr);
```



基本運算之演算法及程式

加入動作

```
void insert_node (struct node *ptr, struct node *head,
   struct node *tail)
{
struct node *prev; *this
if (head = = NULL)
{  /*加入資料為第一筆
ptr -> next = ptr;
head = ptr;
tail = ptr;
}
this = head;
```

```
if (ptr ->key < this ->key)
{ /*加入前端
ptr -> next = this;
head = ptr;
tail -> next = head;
}
else
{
while (this -> next != head)
{
prev = this;
this = this ->next;
```

```
if (ptr->key < this->key)
{  /*加入於特定節點*/
ptr ->next = this;
prev ->next = ptr;
break;
}
}
if (ptr->key >= tail->key)
{  /*加入尾端*/
ptr->next = head;
this->next = ptr;
tail = ptr;
}
}
}
}
```

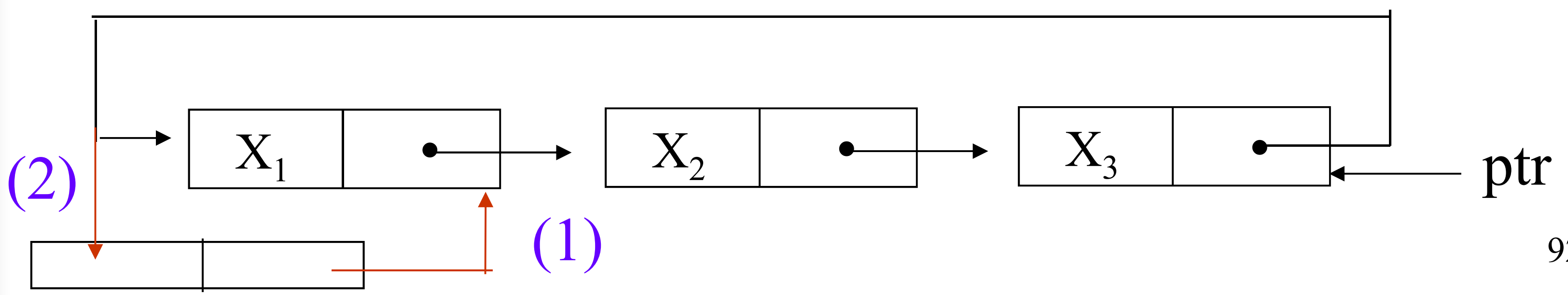
刪除動作

計算環狀鏈結串列之長度

```
int  Clength (NODE *CL )
/*此函數計算環狀鏈結串列之長度*/
{
  int num=0;
  NODE  *p;
  if (p != CL)  {
     p=CL;
     do {
num ++;
p = p->next;
} while (p != CL);
  }
  return(num);
}
```

Operations for Circular Linked Lists

```
void insert_front (list_pointer *ptr, list_pointer
node)
{
    if (IS_EMPTY(*ptr)) {
       *ptr= node;
       node->link = node;
    }
    else {
        node->link = (*ptr)->link;  (1)
        (*ptr)->link = node;        (2)
     }
}
```



Length of Linked List

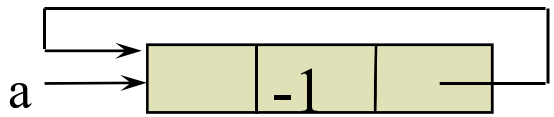
```
int length(list_pointer ptr)
{
    list_pointer temp;
    int count = 0;
    if (ptr) {
        temp = ptr;
        do {
            count++;
            temp = temp->link;
        } while (temp!=ptr);
    }
    return count;
}
```

4.4.4 Representing Polynomials As Circularly Linked Lists

Head Node

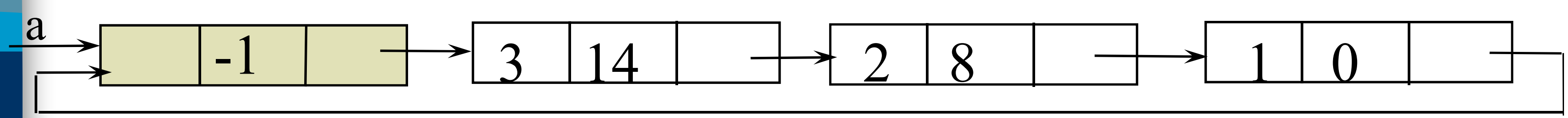
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



$$a = 3x^{14} + 2x^{8} + 1$$

Additional List Operations

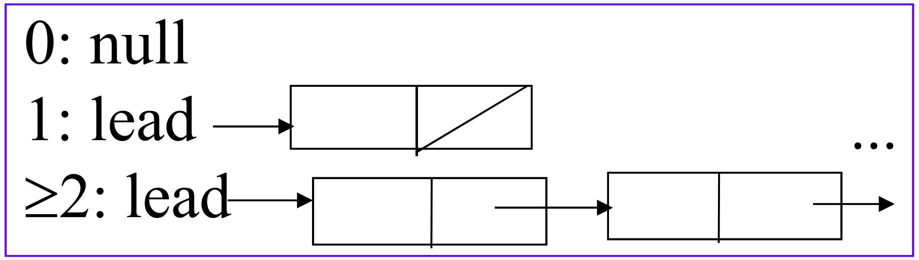
```
typedef struct list_node *list_pointer;
typedef struct list_node {
    char data;
    list_pointer link;
};
```

Invert single linked lists
Concatenate two linked lists

Invert Single Linked Lists

Use two extra pointers: middle and trail.

```
list_pointer invert(list_pointer lead)
{
    list_pointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```



Concatenate Two Lists

```
list_pointer concatenate(list_pointer
             ptr1, list_pointer ptr2)
{
  list_pointer temp;
  if (IS_EMPTY(ptr1)) return ptr2;
  else {
    if (!IS_EMPTY(ptr2)) {
      for (temp=ptr1;temp->link;temp=temp->link);
      temp->link = ptr2;
    }
    return ptr1;
  }
}
```

O(m) where m is # of elements in the first list

作業

- Pp.172: ex1

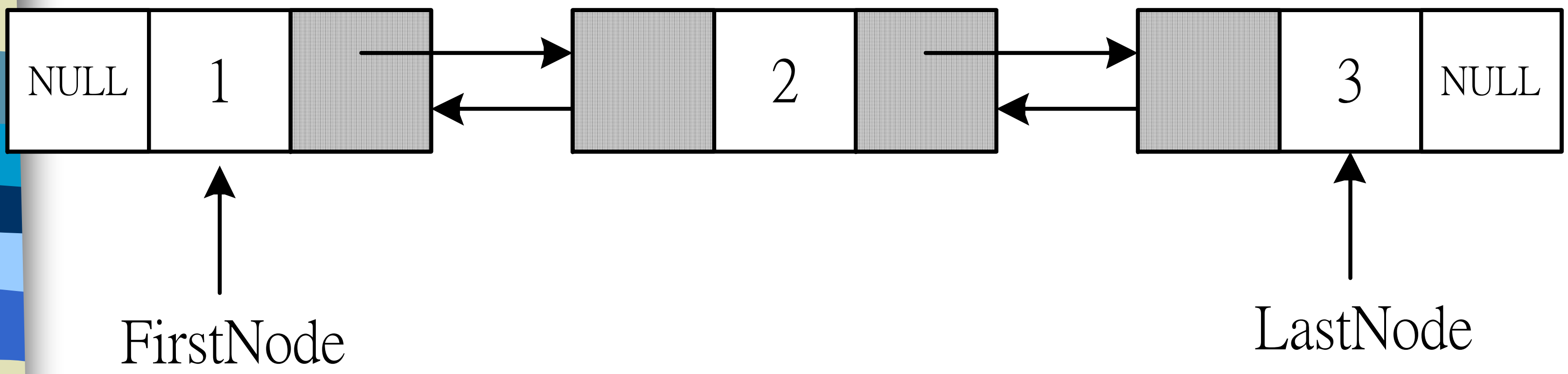
4.8 雙向鏈結串列 Doubly Linked List

Move in forward and backward direction.

Singly linked list (in one direction only)
How to get the preceding node during deletion or insertion?
Using 2 pointers

Node in doubly linked list
left link field (llink)
data field (item)
right link field (rlink)

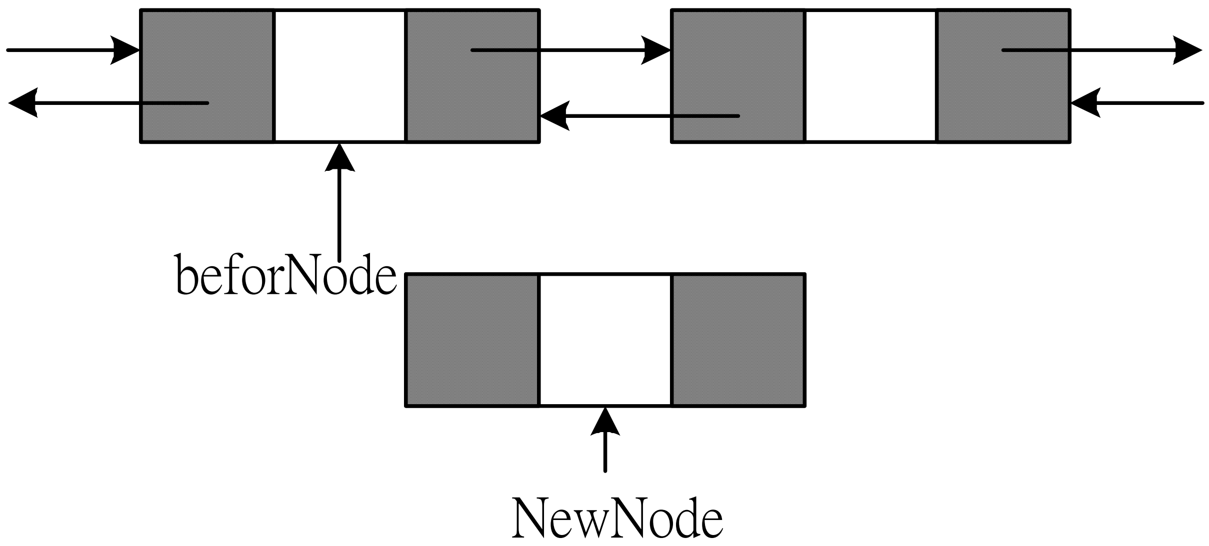
雙向鏈結串列




```
typedef struct dlist_node
{
    struct dlist_node *left;
    int              data;
    struct dlist_node *right;
}Dnode;
```

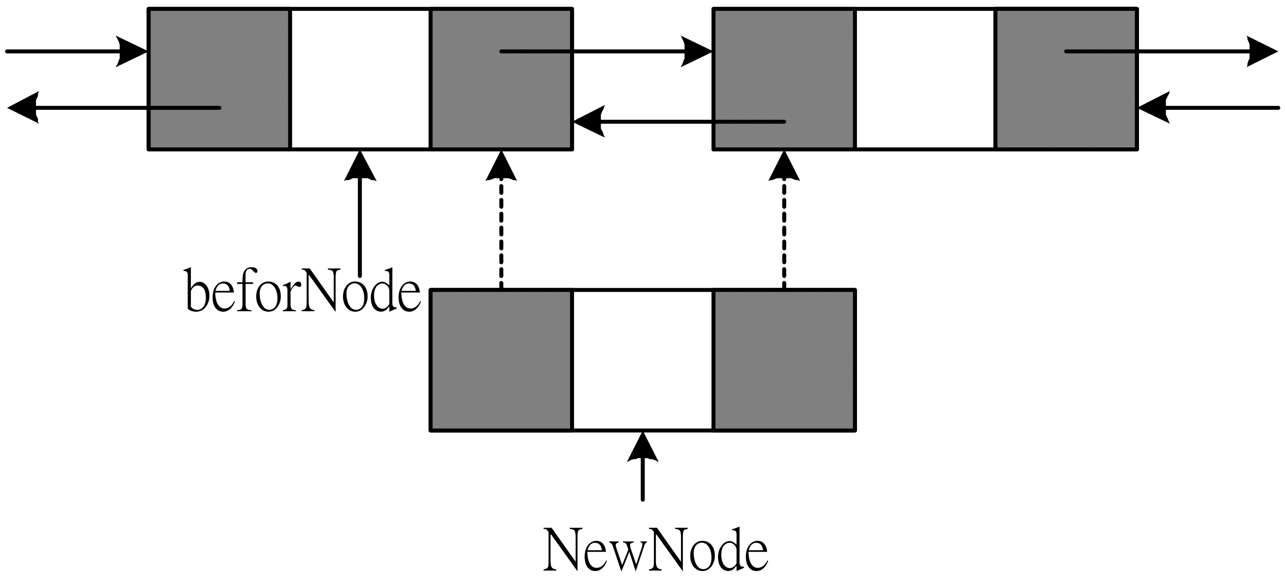
插入節點

- 要將一個新節點(由**newnode**指標所指到)，插入**beforeNode**指標節點之右



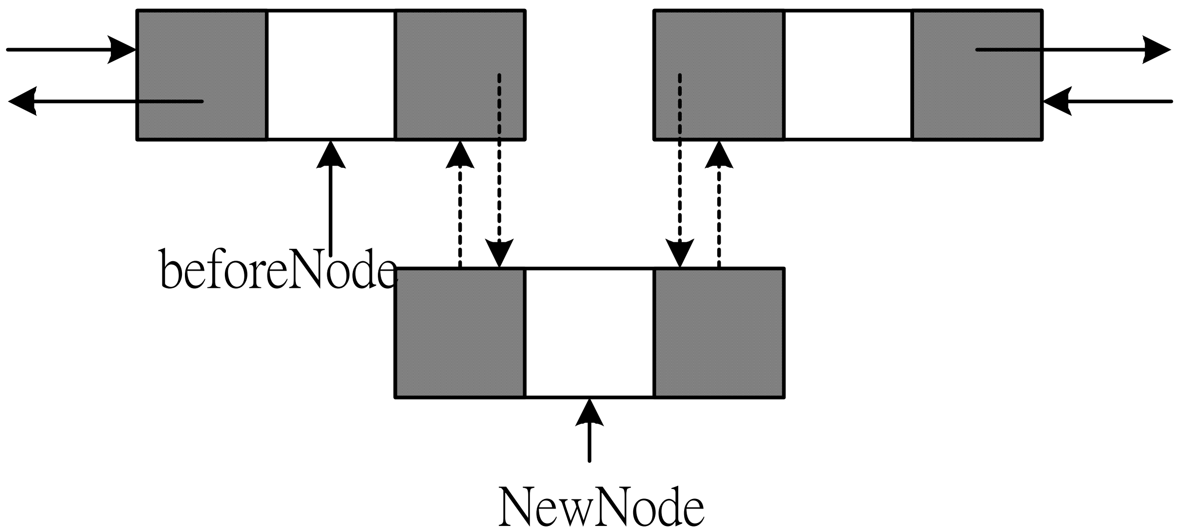
插入節點

- 處理新節點上的兩個鏈結：
  - 左鏈結：NewNode->left=beforeNode;
  - 右鏈結：NewNode->right=beforeNode->right;



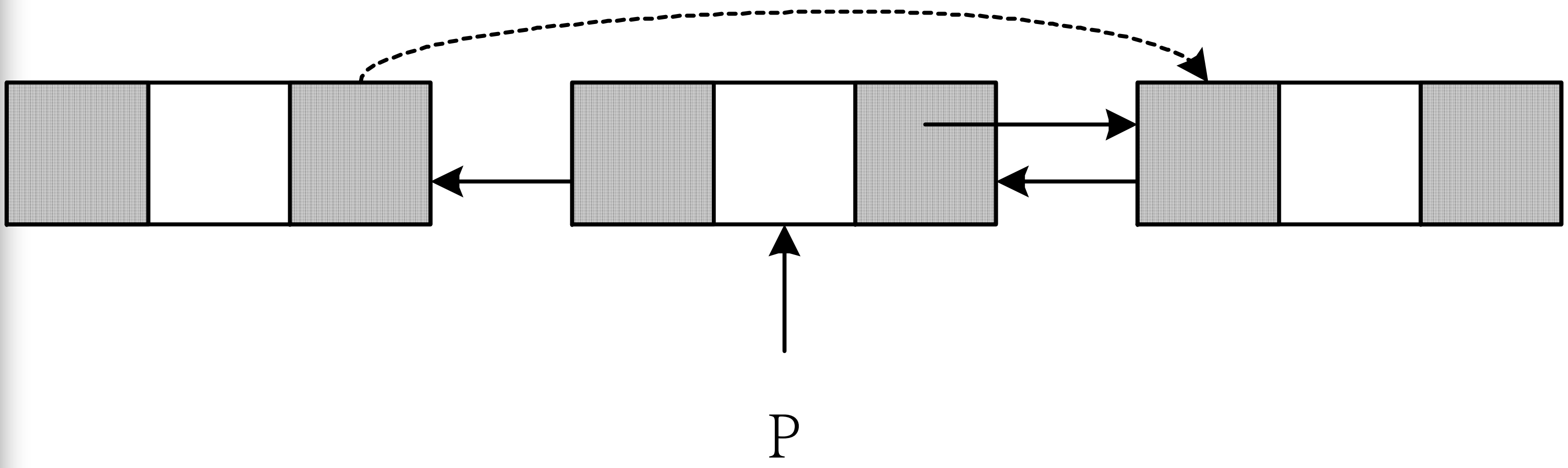
插入節點

- 改變舊節點的鏈結，讓它們指向新節點：
  - beforeNode->right->left=NewNode;
  - beforeNode->right=NewNode;



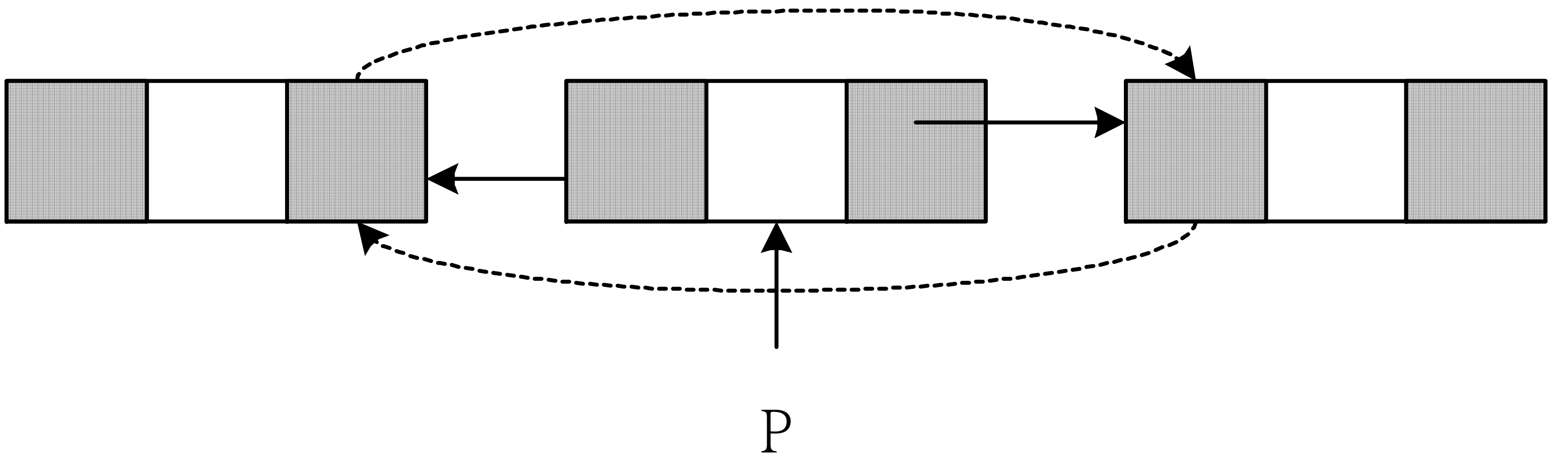
刪除節點

- 改變左邊節點的右指標使它越過舊節點：
  - p->left->right=p->right;



刪除節點

- 改變右邊節點的左指標使它越過舊節點：
  - p->right->left=p->left;



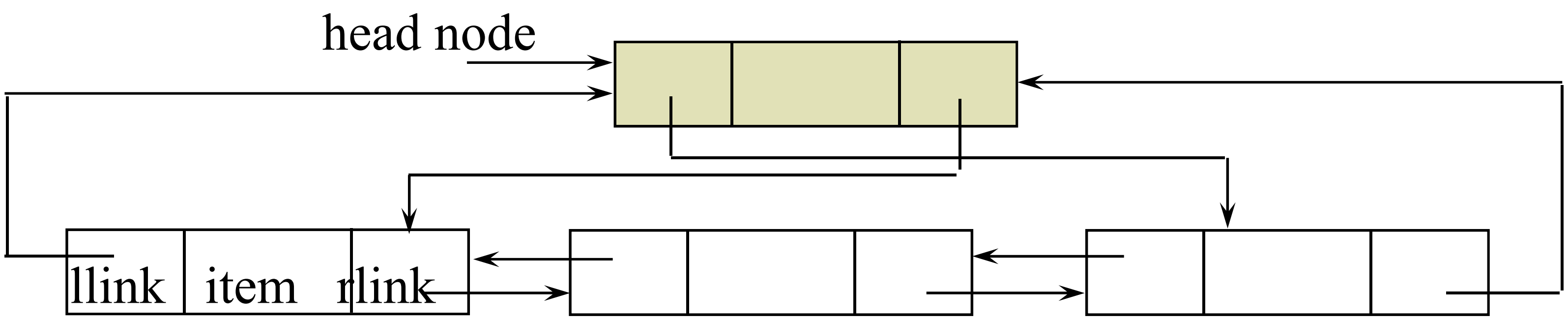
刪除節點

- 刪除舊節點
  - free(p);

Doubly Linked Lists

```
typedef struct node *node_pointer;
typedef struct node {
    node_pointer llink;
    element item;
    node_pointer rlink;
}
```

ptr
= ptr->rlink->llink
= ptr->llink->rlink



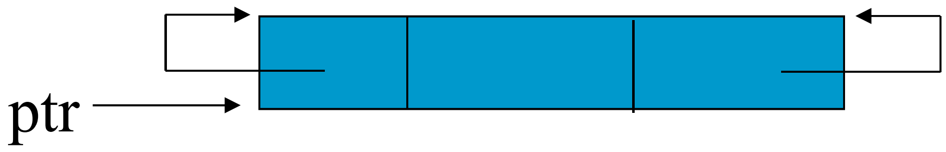


***Figure 4.22:**Empty doubly linked circular list with head node (p.188)*

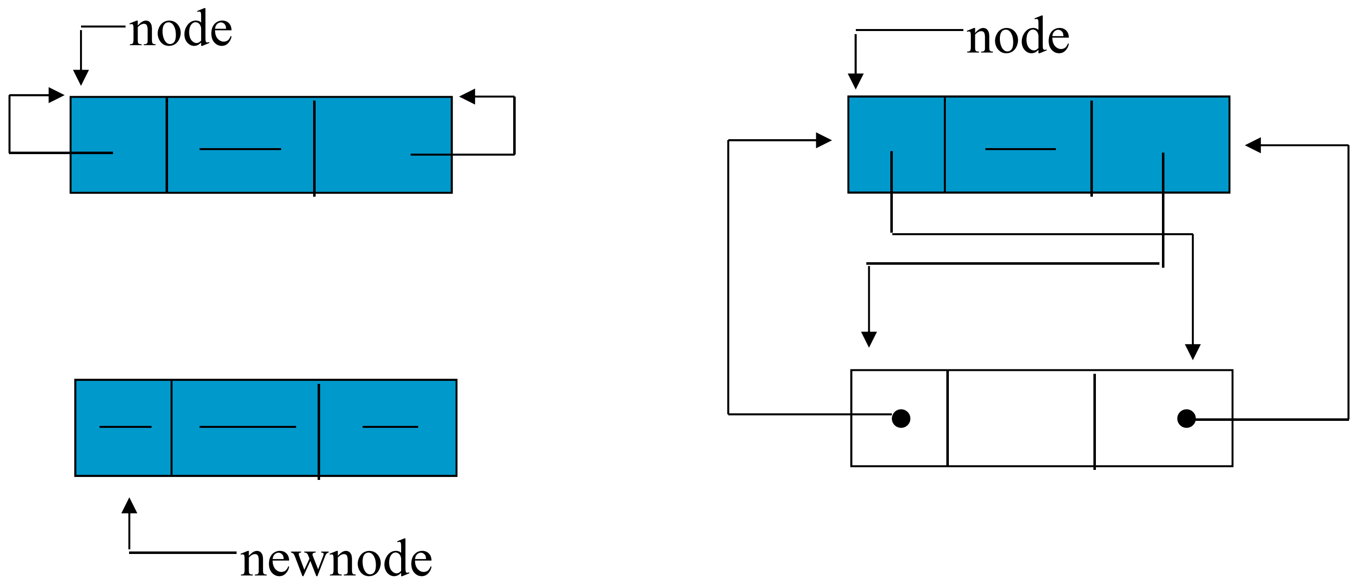
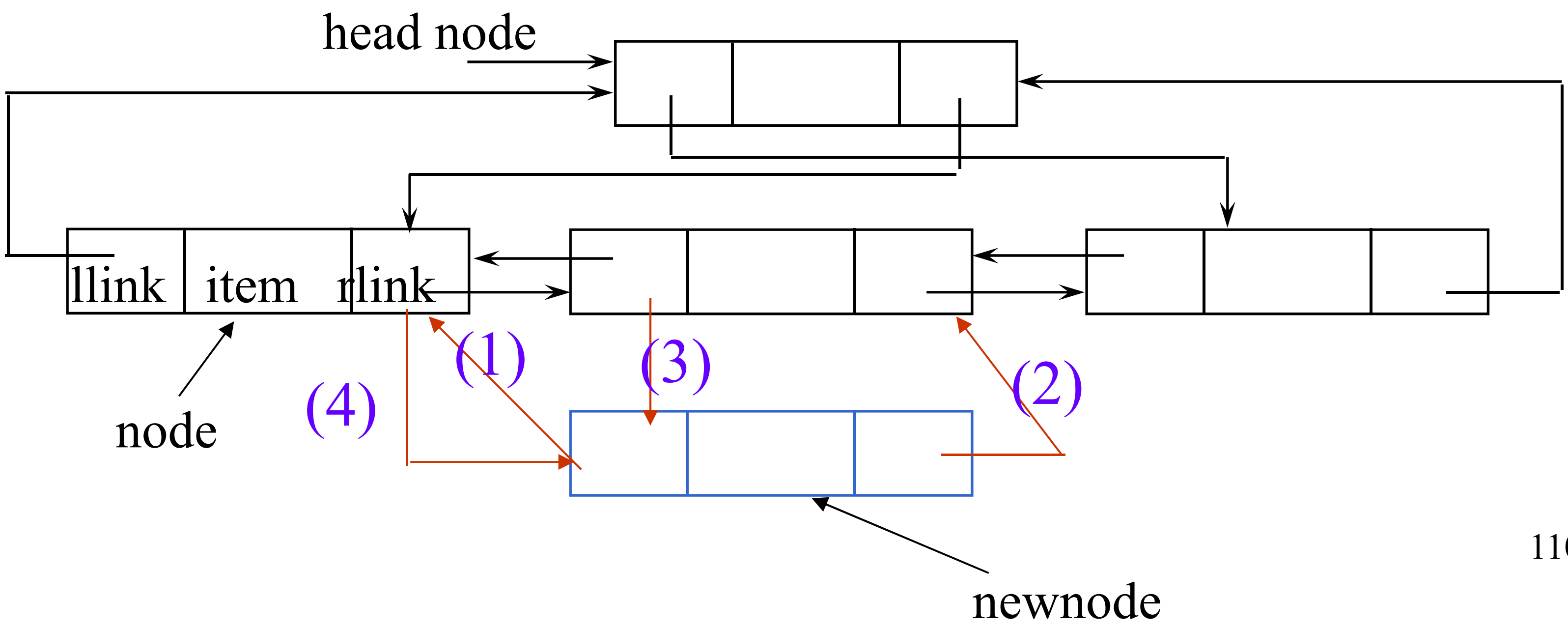


*Figure 4.25: Insertion into an empty doubly linked circular list (p.18?)

Insert

```
void dinsert(node_pointer node, node_pointer newnode)
{
    (1) newnode->llink = node;
    (2) newnode->rlink = node->rlink;
    (3) node->rlink->llink = newnode;
    (4) node->rlink = newnode;
}
```



Delete

```
void ddelete(node_pointer node, node_pointer deleted)
{
    if (node==deleted) printf("Deletion of head node
                                         not permitted.\n");
    else {
        (1) deleted->llink->rlink= deleted->rlink;
        (2) deleted->rlink->llink= deleted->llink;
            free(deleted);
    }
}
```