

# 資料結構

# DATA STRUCTURE

Chap.04 Linked Lists

2020 Spring

# 陣列(Array)

- 陣列(array)由一連串的索引(index)和值(value)構成
- 對每個索引index，必有一個相關聯的值value



- 如果可能的話，盡量使用連續空間的記憶體
  - 減少存取時間

# Pointers (1/5)

- 考慮下列一連串依字母順序排序的三個字英文單字，以at為結尾：  
(bat, cat, sat, vat)
- If we store this list in an array
  - Add the word mat to this list
    - move sat and vat one position to the right before we insert mat.
  - Remove the word cat from the list
    - move sat and vat one position to the left
- Problems of a sequence representation (ordered list)
  - 從陣列中任意的插入/刪除非常浪費時間
  - 浪費空間

# Pointers (2/5)

## ■ using linked representation a.k.a Linked List

- 資料不用連續空間，而是可任意放在記憶體中
- 儲存下一個element的位置，以此作為正確存取資料的方式
- *Thus, associated with each list element is a node which contains both a **data component** and a **pointer** to the next item in the list. The pointers are often called **links**.*



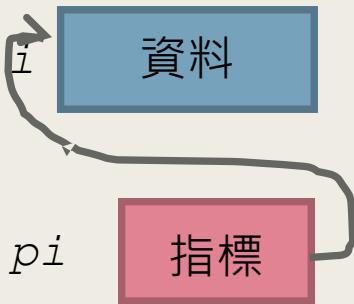
# Pointers (3/5)

- C provides extensive supports for pointers.
  - *Two most important operators used with the pointer type :*

- & the address operator

- \* the dereferencing (or indirection) operator

- Example:



- If we have the declaration:

```
int i, *pi;
```

- then *i* is an integer variable and *pi* is a pointer to an integer.

- If we say:

```
pi = &i;
```

- then *&i* returns the address of *i* and assigns it as the value of *pi*.

- To assign a value to *i* we can say:

```
i = 10; or *pi = 10;
```

# Pointers (4/5)

## ■ Pointers can be **dangerous**

- *Using pointers: high degree of flexibility and efficiency, but dangerous as well.*

- 不使用時請將所有指標指向NULL.
- Another: using explicit **type cast** when converting between pointer types.
- **Example:**

```
pi = malloc(sizeof(int)); /*assign to pi a pointer to  
int*/  
  
pf = (float *)pi; /*casts int pointer to float pointer*/
```

- In many systems, pointers have the same size as type **int**.
  - *Since **int** is the default type specifier, some programmers omit the return type when defining a function.*
  - *The return type defaults to **int** which can later be interpreted as a pointer.*

# Pointers (5/5)

- Using dynamically allocated storage(動態分配記憶體)
  - *When programming, you may not know how much space you will need, nor do you wish to allocate some vary large area that may never be required.*
    - C provides **heap**, for allocating storage at run-time.
    - You may call a function, **malloc**, and request the amount of memory you need.
    - When you no longer need an area of memory, you may free it by calling another function, **free**, and return the area of memory to the system.
  - *Example:*

```
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);
```

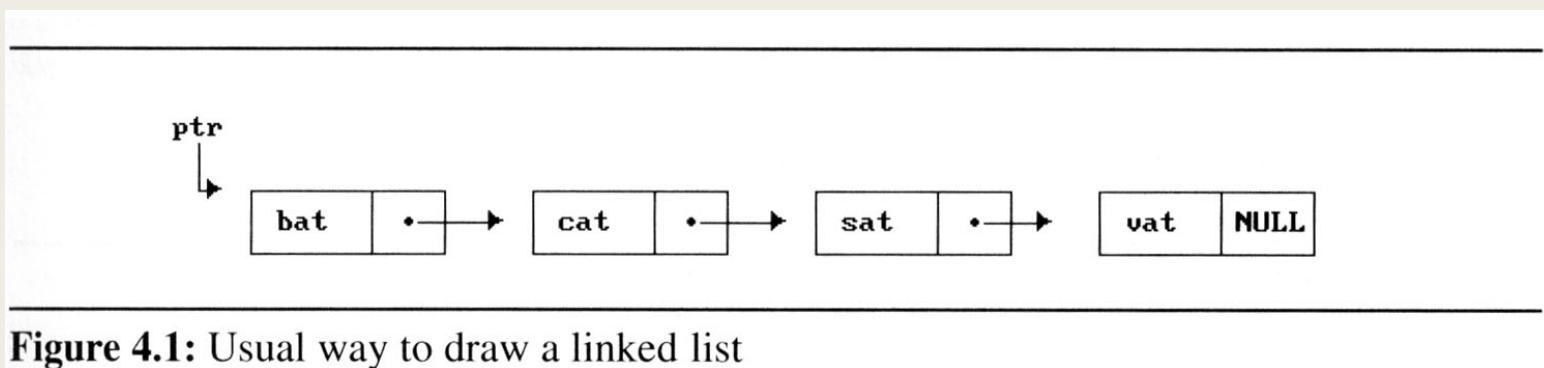
Request  
memory

Free memory

Program 4.1: Allocation and deallocation of pointers

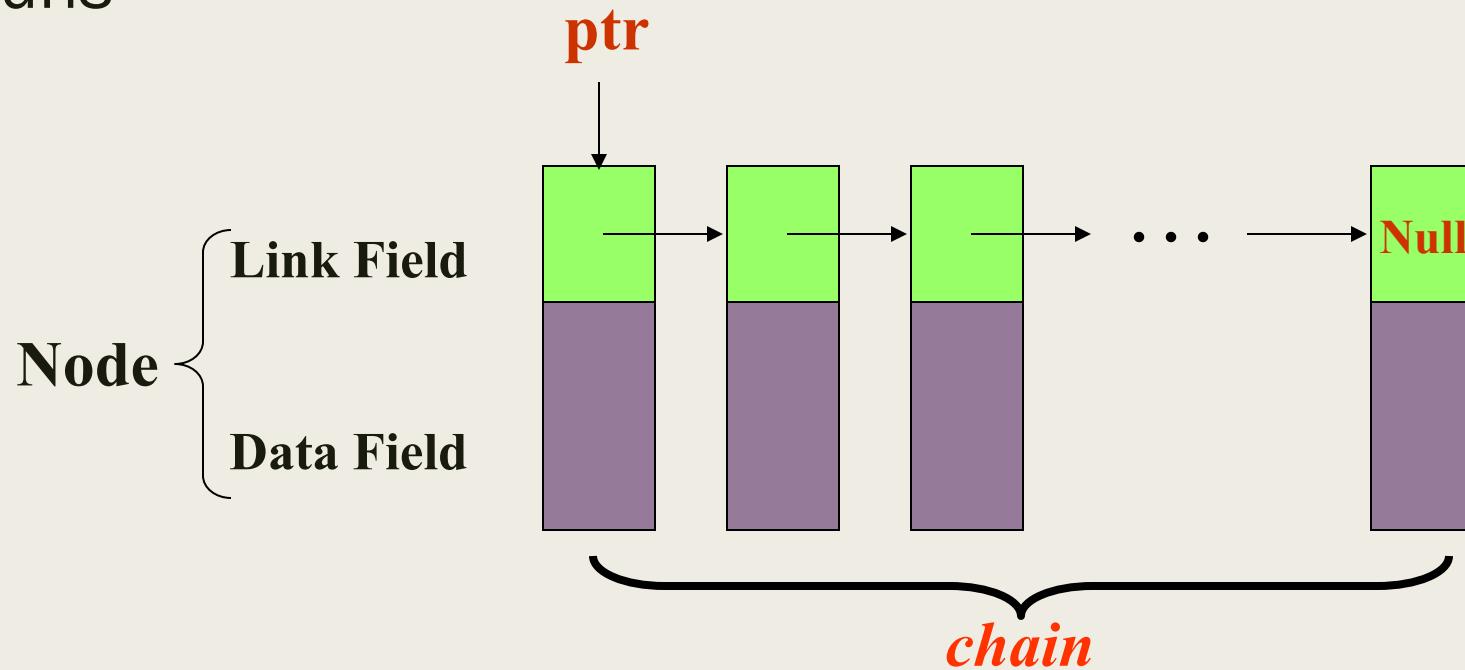
# Singly Linked Lists (1/15)

- Linked lists are drawn as an ordered sequence of nodes with links represented as arrows (Figure 4.1).
  - *The name of the pointer to the first node in the list is the name of the list. (the list of Figure 4.1 is called ptr.)*
    - Notice that we do not explicitly put in the values of pointers, but simply draw allows to indicate that they are there.



# Singly Linked Lists (2/15)

- The nodes do not reside in sequential locations
- The locations of the nodes may change on different runs



# Singly Linked Lists (3/15)

- Why it is easier to make arbitrary insertions and deletions using a linked list?
  - To insert the word *mat* between *cat* can *sat*, we must:
    - Get a node that is currently unused; let its address be *paddr*.
    - Set the data field of this node to *mat*.
    - Set *paddr*'s link field to point to the address found in the link field of the node containing *cat*.
    - Set the link field of the node containing *cat* to point to *paddr*.

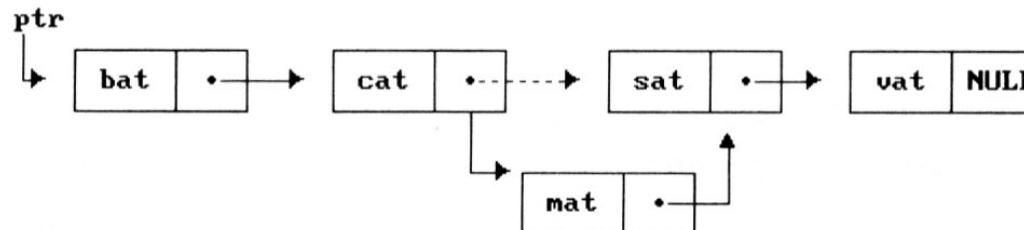
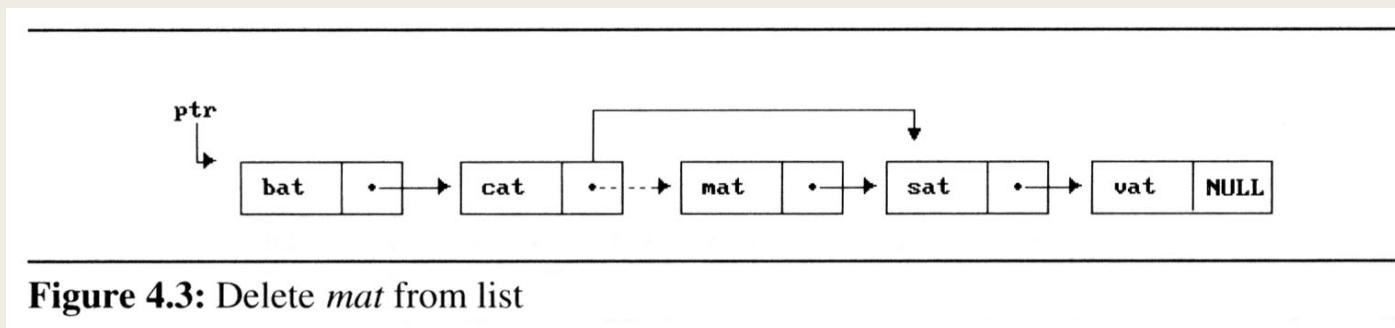


Figure 4.2: Insert *mat* after *cat*

# Singly Linked Lists (4/15)

- Delete mat from the list:
  - We only need to find the element that immediately precedes mat, which is cat, and set its link field to point to mat's link (Figure 4.3).
  - We have not moved any data, and although the link field of mat still points to sat, mat is no longer in the list.



# Singly Linked Lists (5/15)

- We need the following capabilities to make linked representations possible:
  - *Defining a node's structure, that is, the fields it contains.* We use self-referential structures, discussed in Section 2.2 to do this.
  - *Create new nodes when we need them.* (`malloc`)
  - *Remove nodes that we no longer need.* (`free`)

# Singly Linked Lists (6/15)

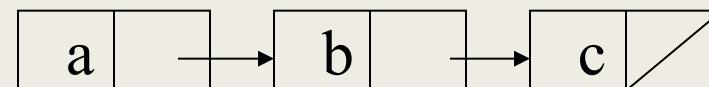
## ■ 2.2.4 Self-Referential Structures

- One or more of its components is a pointer to itself.

- ```
typedef struct list {  
    char data;  
    list *link;  
}
```

Construct a list with three nodes  
`item1.link=&item2;`  
`item2.link=&item3;`  
`malloc: obtain a node (memory)`  
`free: release memory`

- `list item1, item2, item3;`  
`item1.data='a';`  
`item2.data='b';`  
`item3.data='c';`  
`item1.link=item2.link=item3.link=NULL;`



# Singly Linked Lists (7/15)

## ■ Example 4.1 [*List of words ending in at*]:

- *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*

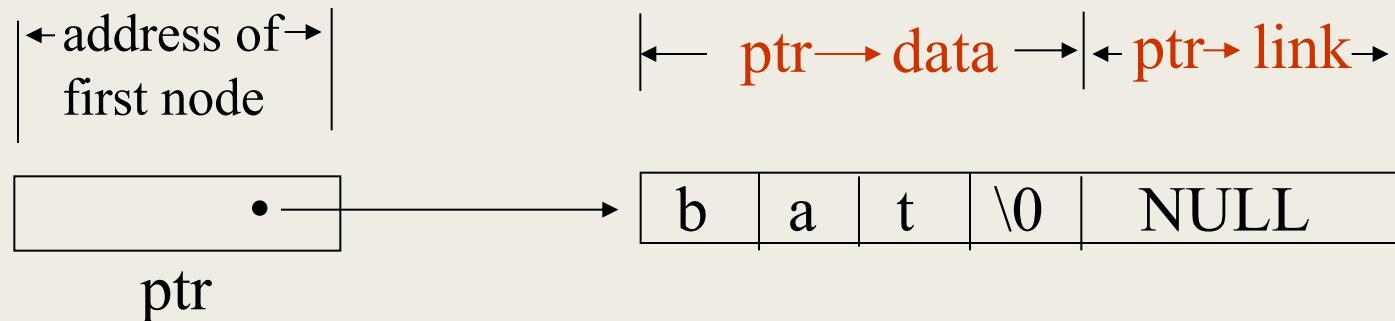
```
ptr=(list_pointer) malloc (sizeof(list_node));
```

- *Return the spaces:*

```
free(ptr);
```

# Singly Linked Lists (8/15)

```
e -> name ?? (*e).name  
strcpy(ptr -> data, "bat");  
ptr -> link = NULL;
```



**Figure 4.4:**Referencing the fields of a node(p.142)

# Singly Linked Lists (9/15)

- **Example 4.2** [*Two-node linked list*]:

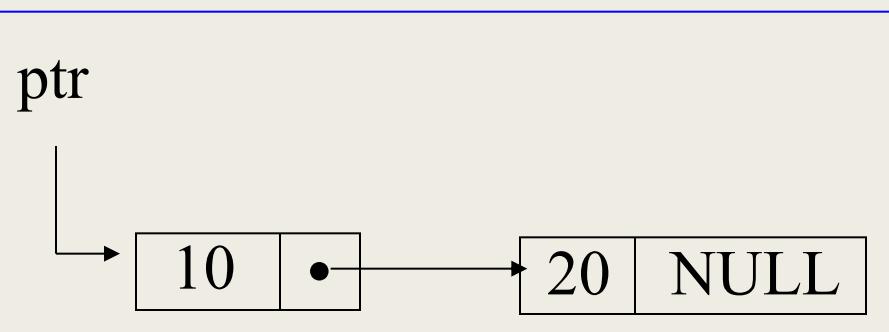
```
typedef struct list_node *list_pointer;  
typedef struct list_node {  
    int data;  
    list_pointer link;  
};  
list_pointer ptr =NULL;
```

```
#define IS_FULL(ptr) (!(ptr))
```

When returns *NULL* if there  
is no more memory.

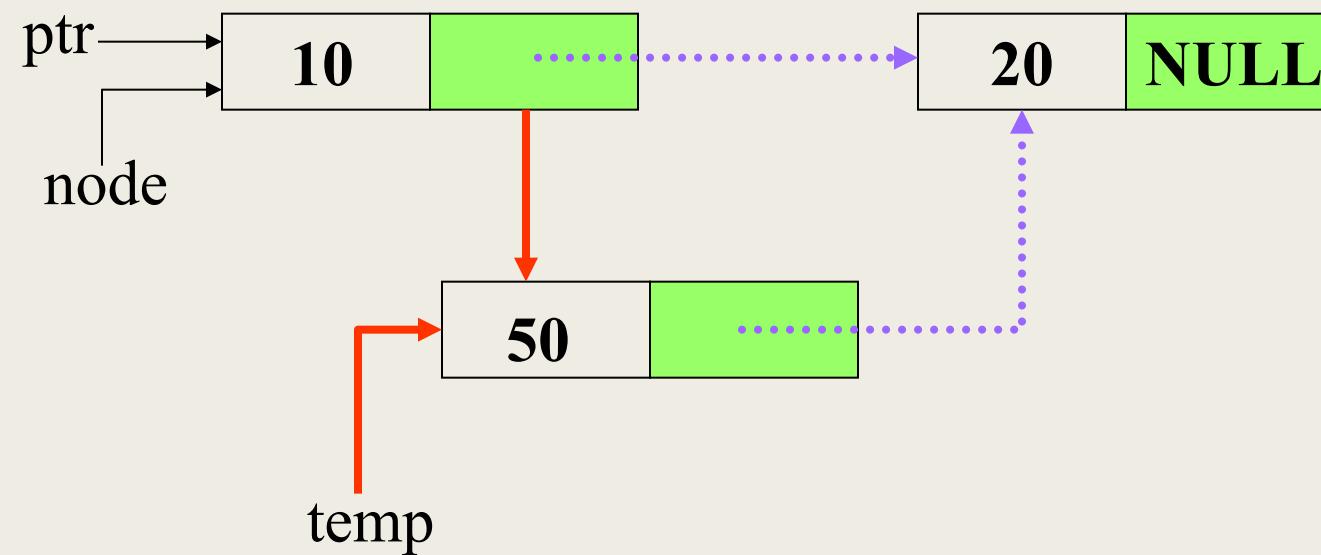
- **Program 4.2:** Create a two-node list

```
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;  
}
```



# Singly Linked Lists (10/15)

- Insertion
  - *Observation*
    - insert a new node with data = 50 into the list ptr after node

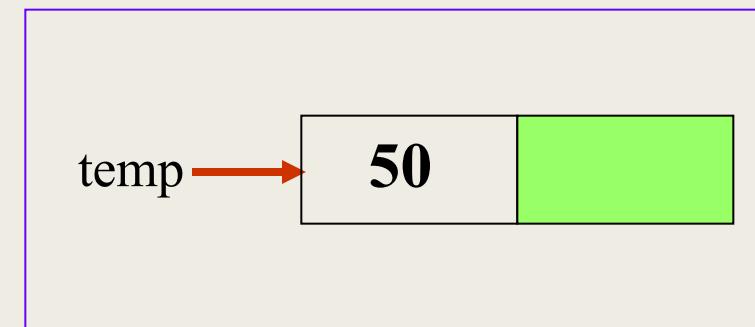


# Singly Linked Lists (11/15)

## ■ Implement Insertion:

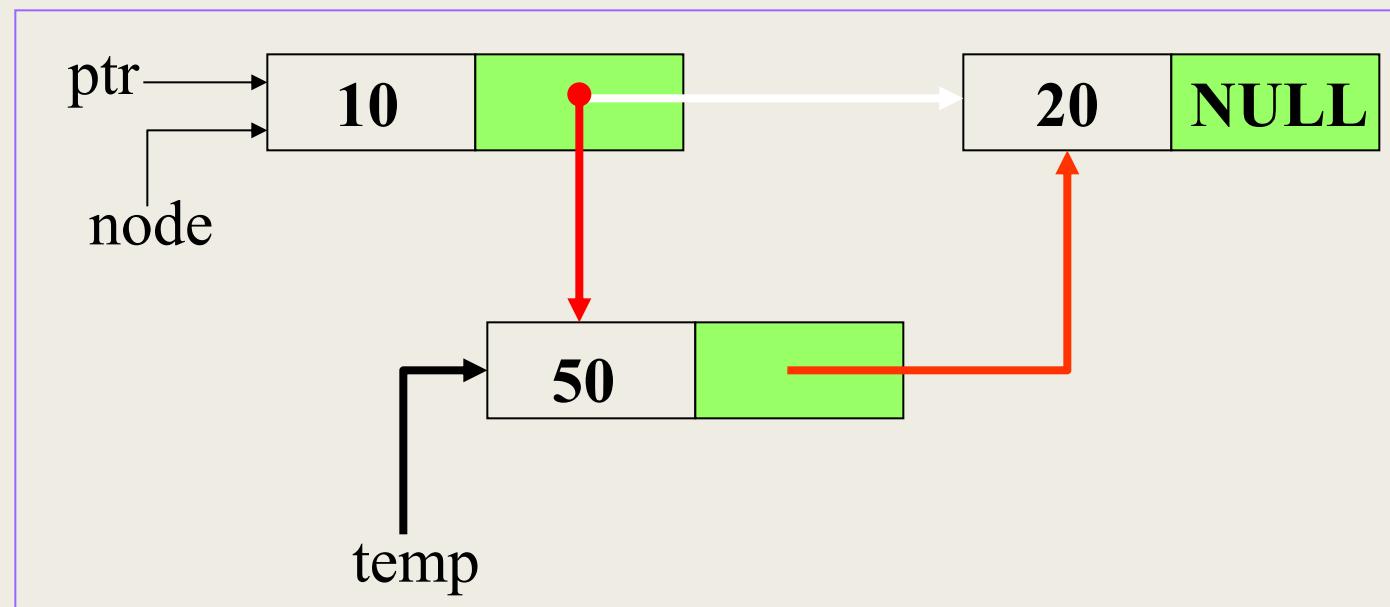
```
void insert(list_pointer *ptr, List_pointer node)
{
    /* insert a new node with data = 50 into the list ptr after
       node */

    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;
```



# Singly Linked Lists (12/15)

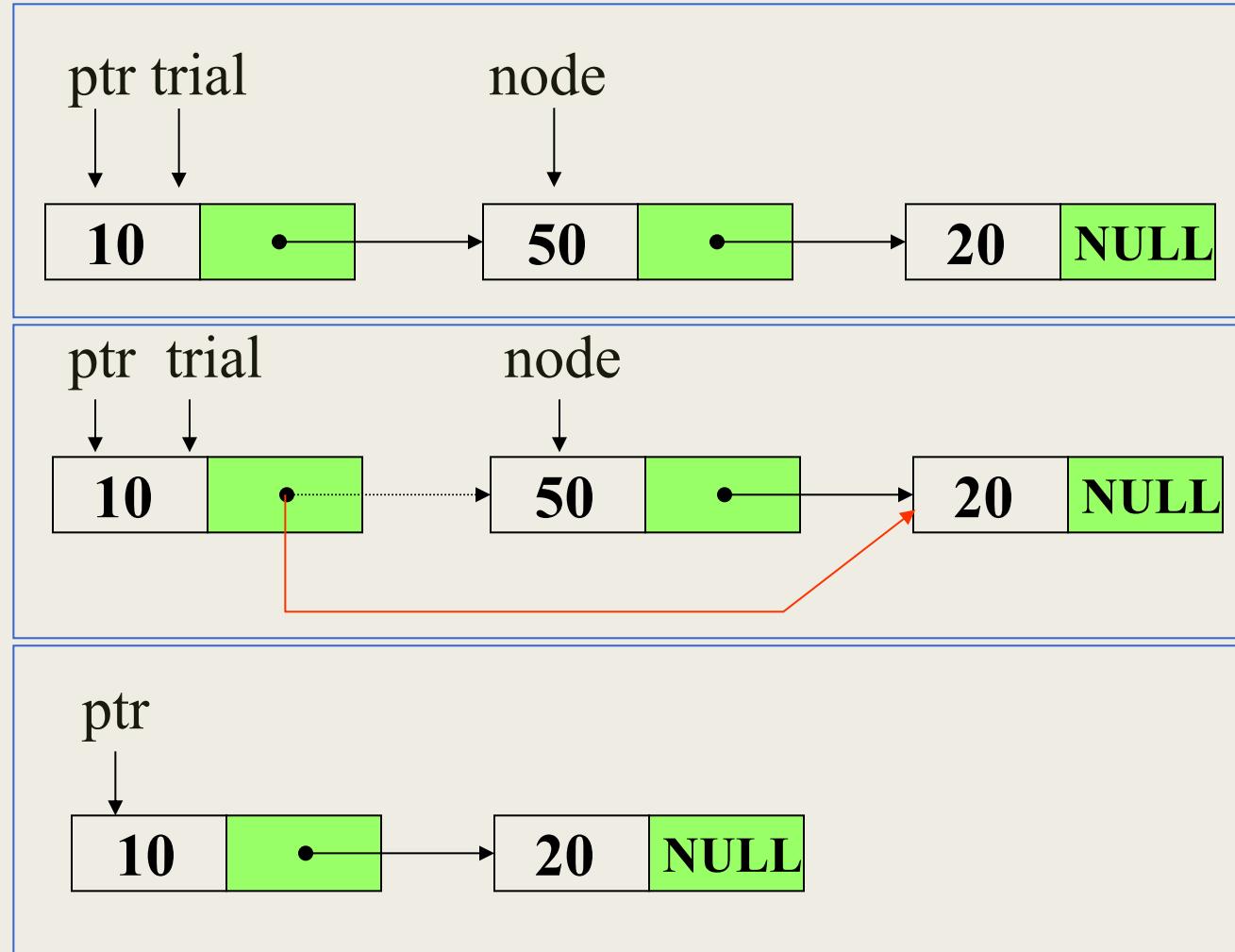
```
if(*ptr){    //nonempty list  
    temp->link = node->link;  
    node->link = temp;  
}  
  
else{        //empty list  
    temp->link = NULL;  
    *ptr = temp;  
}
```



# Singly Linked Lists (13/15)

## ■ Deletion

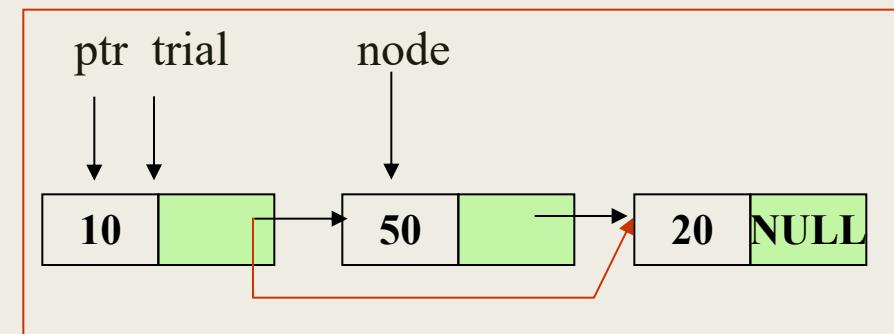
- *Observation: delete node from the list*



# Singly Linked Lists (14/15)

## ■ Implement Deletion:

```
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);
}
```



# Singly Linked Lists (15/15)

- Print out a list (traverse a list)
  - *Program 4.5: Printing a list*

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

# Dynamically Linked Stacks and Queues (1/8)

- When several stacks and queues coexisted, there was no efficient way to represent them sequentially.
  - 循序(sequential)很難管理多個stack or queue
  - Notice that direction of links for both stack and the queue facilitate easy insertion and deletion of nodes.
    - Easily add or delete a node from the top of the stack.
    - Easily add a node to the rear of the queue and add or delete a node at the front of the queue.

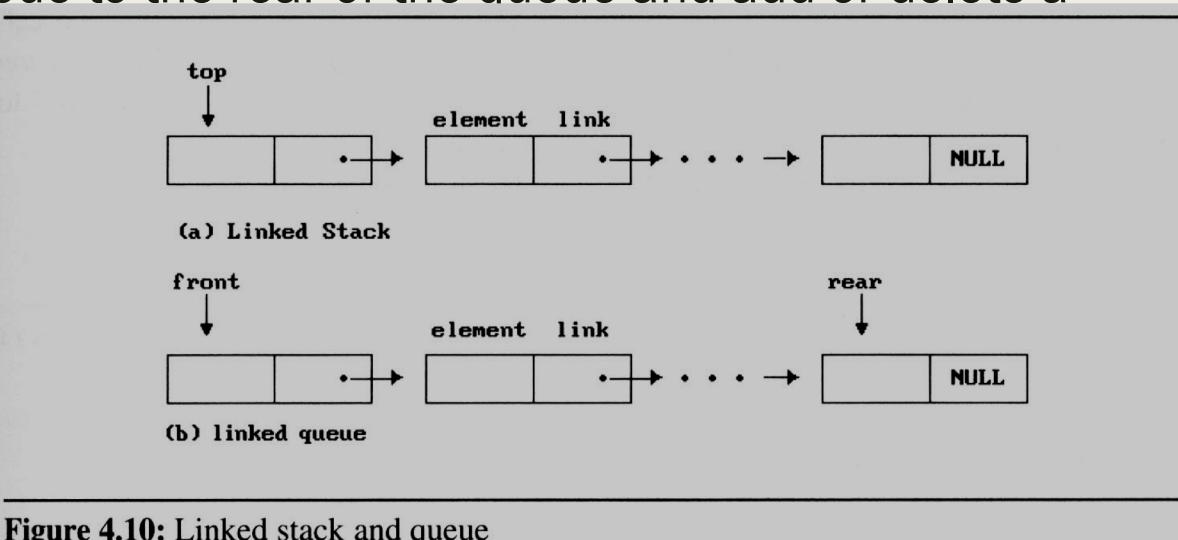
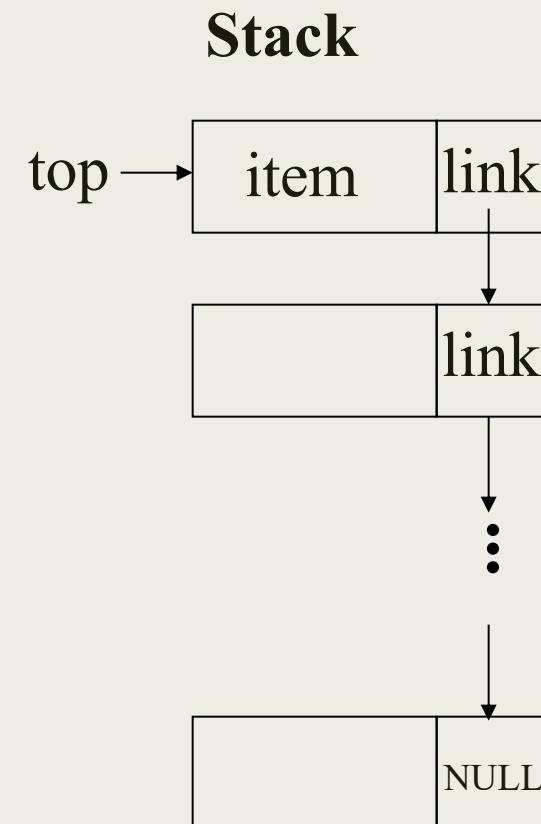


Figure 4.10: Linked stack and queue

# Dynamically Linked Stacks and Queues (2/8)

- Represent n stacks

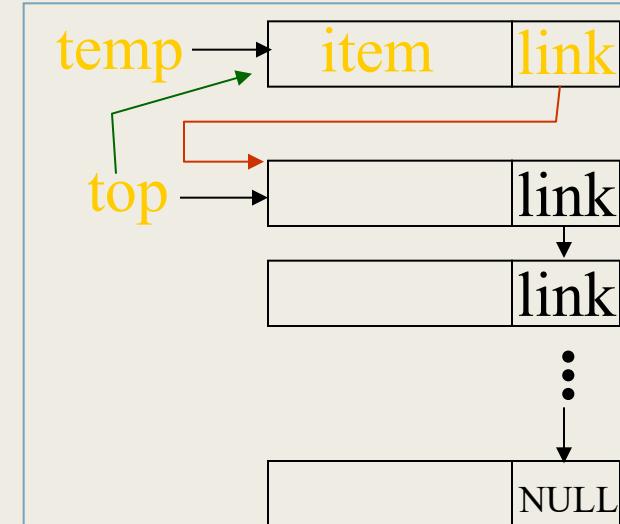
```
#define MAX_STACKS 10 /*maximum number of stacks*/
typedef struct {
    int key;
    /* other fields */
} element;
typedef struct stack *stack_pointer;
typedef struct stack {
    element item;
    stack_pointer link;
};
stack_pointer top[MAX_STACKS];
```



# Dynamically Linked Stacks and Queues (3/8)

## ■ Push in the linked stack

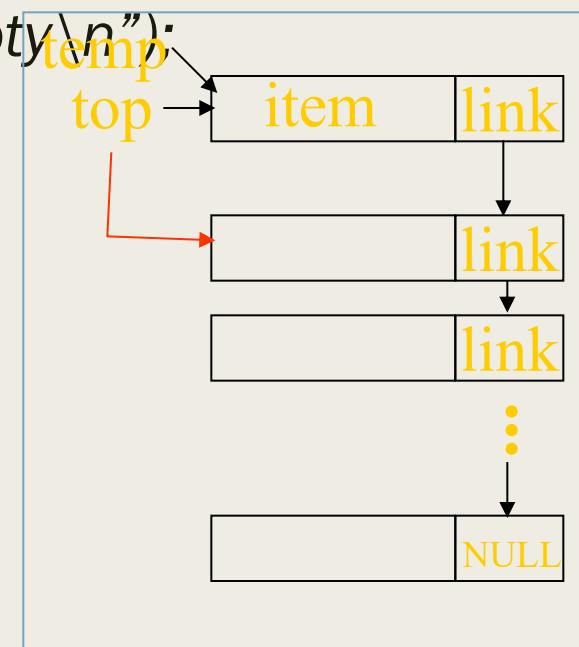
```
void add(stack_pointer *top, element item){  
    /* add an element to the top of the stack */ Push  
    stack_pointer temp = (stack_pointer) malloc (sizeof (stack));  
  
    if (IS_FULL(temp)) {  
        fprintf(stderr, "The memory is full\n");  
        exit(1);  
    }  
    temp->item = item;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



# Dynamically Linked Stacks and Queues (4/8)

## ■ Pop from the linked stack

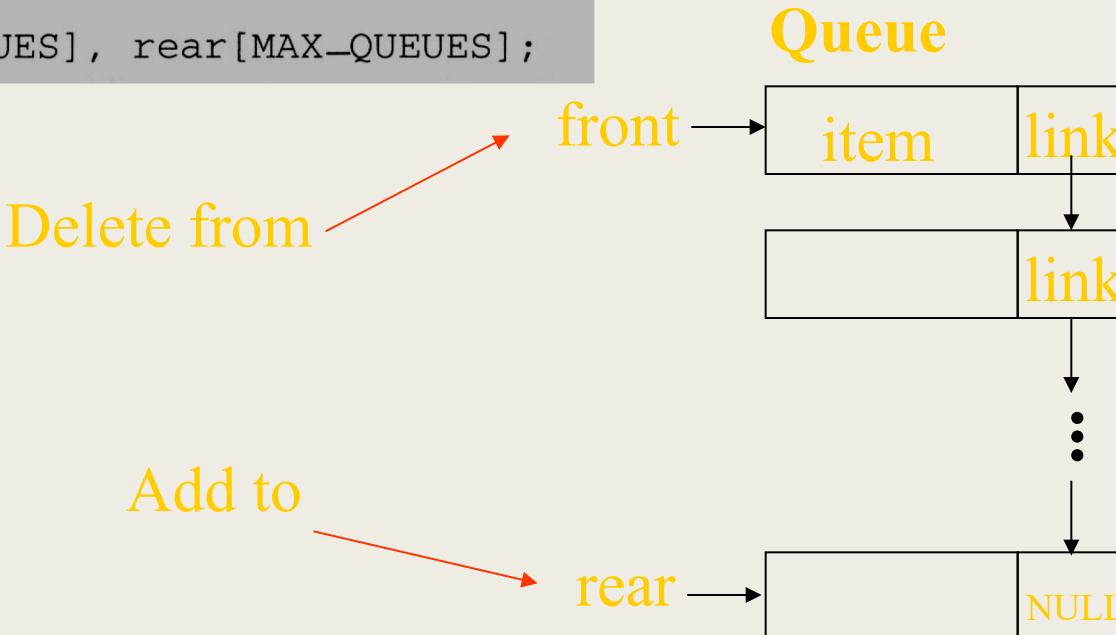
```
element delete(stack_pointer *top) {  
    /* delete an element from the stack */  Pop  
    stack_pointer temp = *top;  
    element item;  
    if (IS_EMPTY(temp)) {  
        fprintf(stderr, "The stack is empty\n");  
        exit(1);  
    }  
    item = temp->item;  
    *top = temp->link; 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



# Dynamically Linked Stacks and Queues (5/8)

- Represent n queues

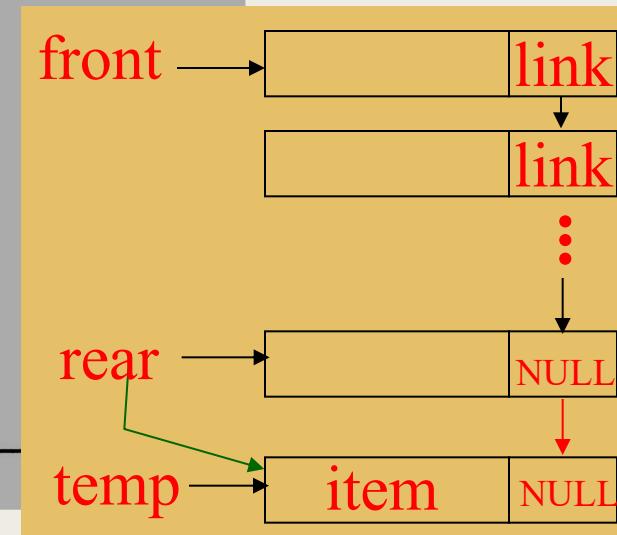
```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queue_pointer;
typedef struct queue {
    element item;
    queue_pointer link;
};
queue_pointer front[MAX_QUEUES], rear[MAX_QUEUES];
```



# Dynamically Linked Stacks and Queues (6/8)

```
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;
}
```

Program 4.8: Add to the rear of a linked queue



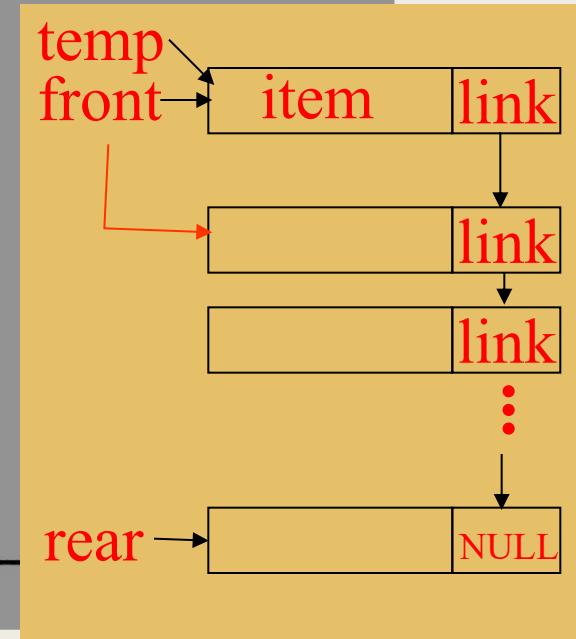
# Dynamically Linked Stacks and Queues (7/8)

- 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;
}
```

---



Program 4.9: Delete from the front of a linked queue

# Dynamically Linked Stacks and Queues (8/8)

■ The solution presented above to the  $n$ -stack,  $m$ -

queue problem is both computationally and conceptually simple.

- *We no longer need to shift stacks or queues to make space.*
- *Computation can proceed as long as there is memory available.*

# Polynomials (1/9)

## ■ Representing Polynomials As Singly Linked Lists

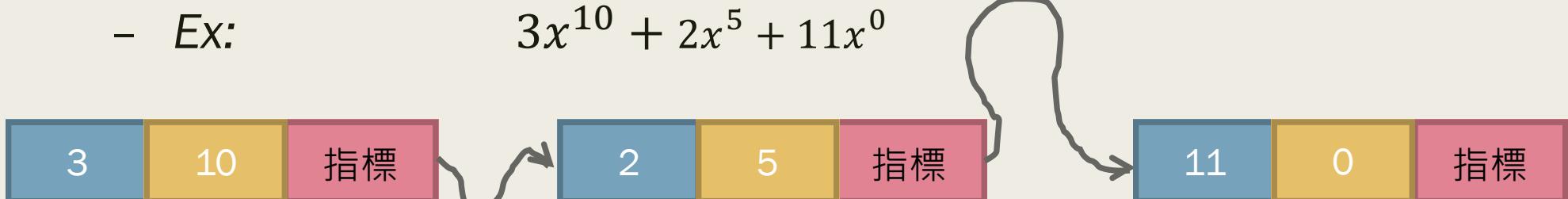
- *The manipulation of symbolic polynomials, has a classic example of list processing.*
- *In general, we want to represent the polynomial:*

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

- Where the  $a_i$  are nonzero coefficients and the  $e_i$  are nonnegative integer exponents such that

$$e_{m-1} > e_{m-2} > \dots > e_1 > e_0 \geq 0 .$$

- We will represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.
- Ex:  $3x^{10} + 2x^5 + 11x^0$



# Polynomials (2/9)

- Assuming that the coefficients are integers, the type declarations are:

```
typedef struct poly_node *poly_pointer;
```

```
typedef struct poly_node {
```

```
    int coef;
```

```
    int expon;
```

```
    poly_pointer link;
```

```
};
```

```
poly_pointer a,b,d;
```

- Draw *poly\_nodes* as:

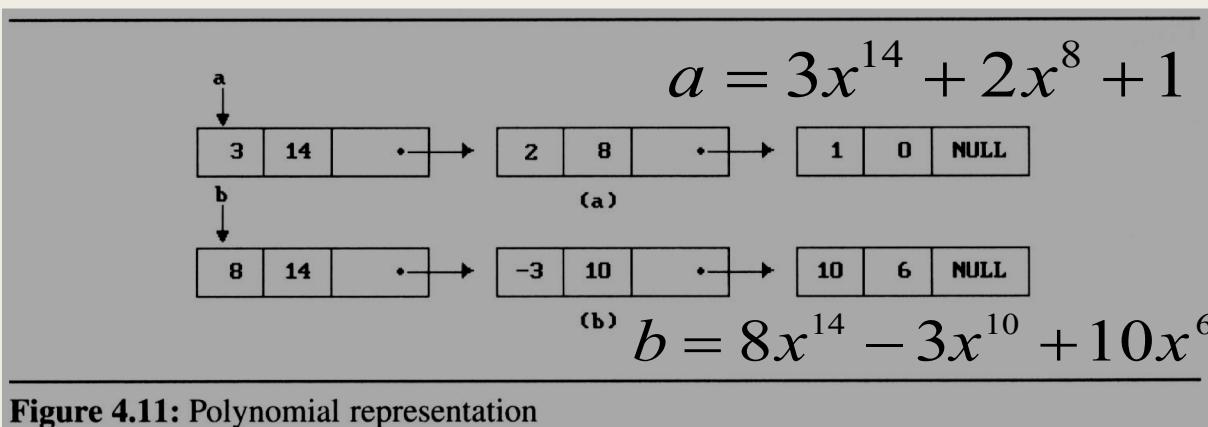
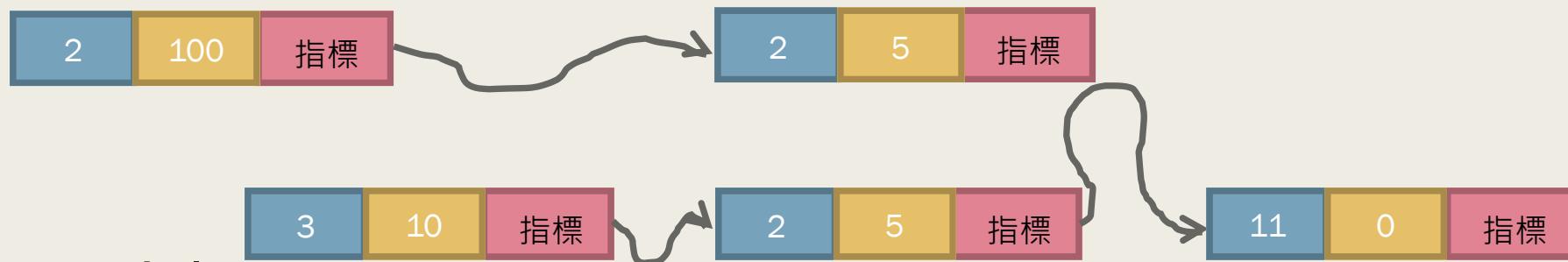


Figure 4.11: Polynomial representation

| coef | expon | link |
|------|-------|------|
|------|-------|------|





## ■ Adding Polynomials

- *To add two polynomials, we examine their terms starting at the nodes pointed to by a and b.*
  - If the exponents of the two terms are equal
    1. *add the two coefficients*
    2. *create a new term for the result.*
  - If the exponent of the current term in a is less than b
    1. *create a duplicate term of b*
    2. *attach this term to the result, called d*
    3. *advance the pointer to the next term in b.*
  - We take a similar action on a if  $a->\text{expon} > b->\text{expon}$ .
- *Figure 4.12 generating the first three term of  $d = a+b$  (next page)*

# Polynomials (4/9)

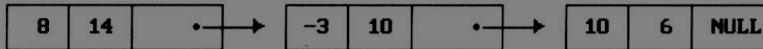


↑  
**d**

(a) **a** → **expon == b** → **expon**



↑  
**a**



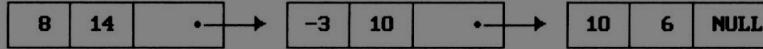
↑  
**b**

↑  
**d**

(b) **a** → **expon < b** → **expon**



↑  
**a**



↑  
**b**

↑  
**d**

(c) **a** → **expon > b** → **expon**

Figure 4.12: Generating the first three terms of  $d = a + b$

# Polynomials (5/9)

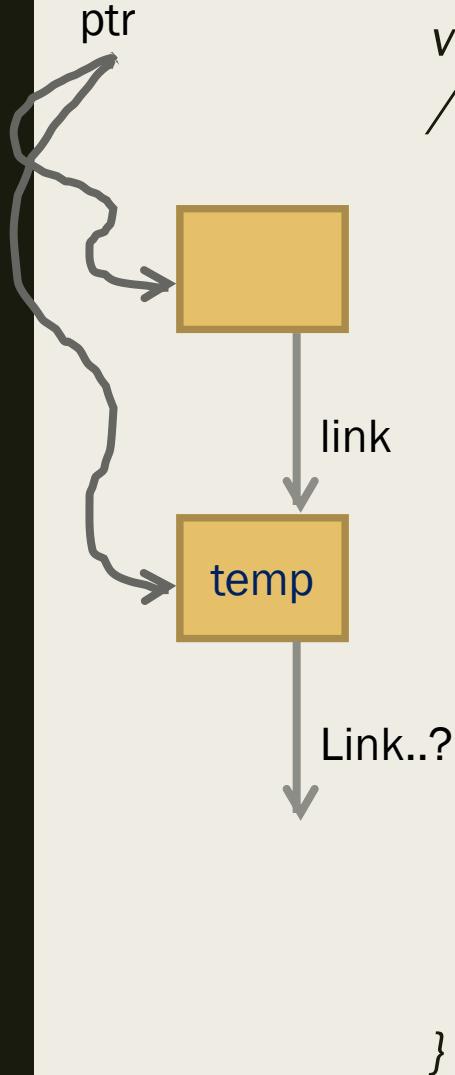
- Add two polynomials

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
    /* return a polynomial which is the sum of a and 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;
        }
    /* copy rest of list a and then list b */
    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;
    /* delete extra initial node */
    temp = front; front = front->link; free(temp);
    return front;
}
```

Program 4.10: Add two polynomials

# Polynomials (6/9)

- Attach a node to the end of a list



```
void attach(float coefficient, int exponent, poly_pointer *ptr){  
    /* create a new node with coef = coefficient and expon = exponent,  
       attach it 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));  
    /* create new node */  
    if (IS_FULL(temp)) {  
        fprintf(stderr, "The memory is full\n");  
        exit(1);  
    }  
    temp->coef = coefficient; /* copy item to the new node */  
    temp->expon = exponent;  
    (*ptr)->link = temp;      /* attach */  
    *ptr = temp;              /* move ptr to the end of the list */  
}
```

# Polynomials (7/9)

## ■ Analysis of padd

$$A(x)(= a_{m-1}x^{e_{m-1}} + \dots + a_0x^{e_0}) + B(x)(= b_{n-1}x^{f_{n-1}} + \dots + b_0x^{f_0})$$

1. coefficient additions

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

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

2. exponent comparisons

extreme case:

$e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \dots > e_1 > f_1 > e_0 > f_0$

$m+n-1$  comparisons

3. creation of new nodes

extreme case: maximum number of terms in  $d$  is  $m+n$

$m + n$  new nodes

summary:  $O(m+n)$

# Polynomials (8/9)

## ■ 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()
```

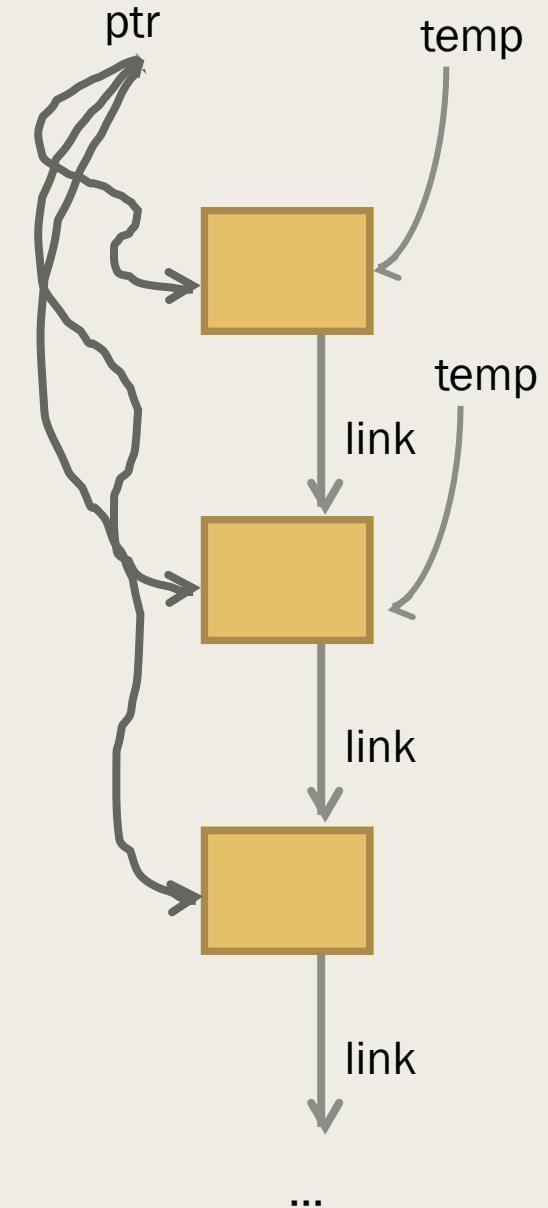
temp is used to hold a partial result.  
By returning the nodes of temp, we  
may use it to hold other polynomials

# Polynomials (9/9)

## ■ Erase Polynomials

- *erase: frees the nodes in temp*

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



# Chain (1/3)

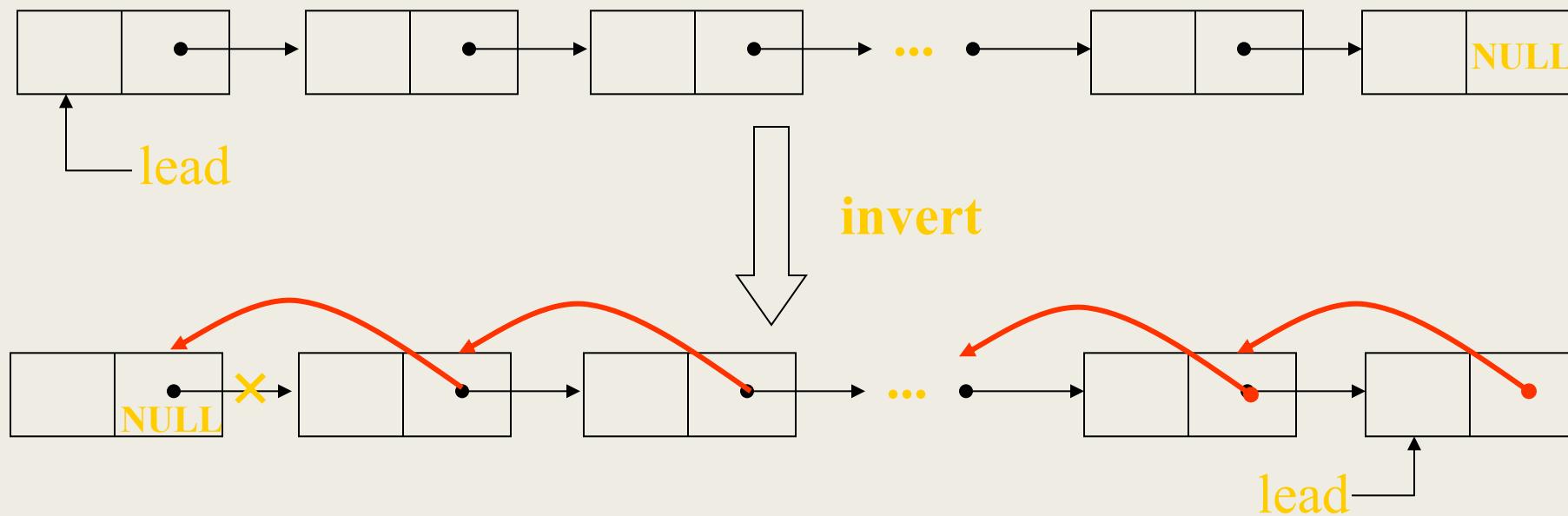
## ■ Chain:

- A singly linked list in which the last node has a null link

## ■ Operations for chains

- Inverting a chain

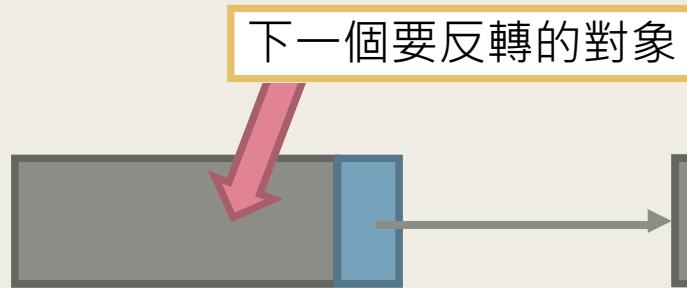
- For a list of  $length \geq 1$  nodes, the while loop is executed  $length$  times and so the computing time is linear or  $O(length)$ .



Initial :



Invert (到中間的時候)...



指到前面去了

所以我需要：

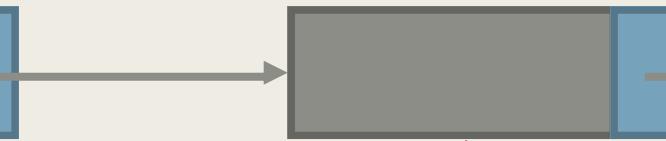
一個指向現在要處理反轉的對象的指標：Now(類似於課本的middle)

一個指向前一個處理的對象的指標：Prev(類似於課本的Trail)

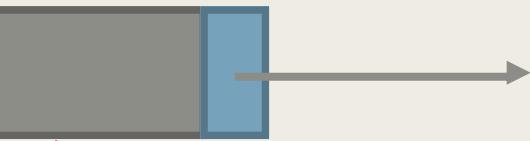
一個指向下一個要被處理的對象的指標：Next(類似於課本的Lead)



1.Now->list=Prev



3.Now=Next



4.Next=Next->List

# Chain (2/3)

```
list-pointer invert(list-pointer lead)
{
    /* invert the list pointed to by lead */
    list-pointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```

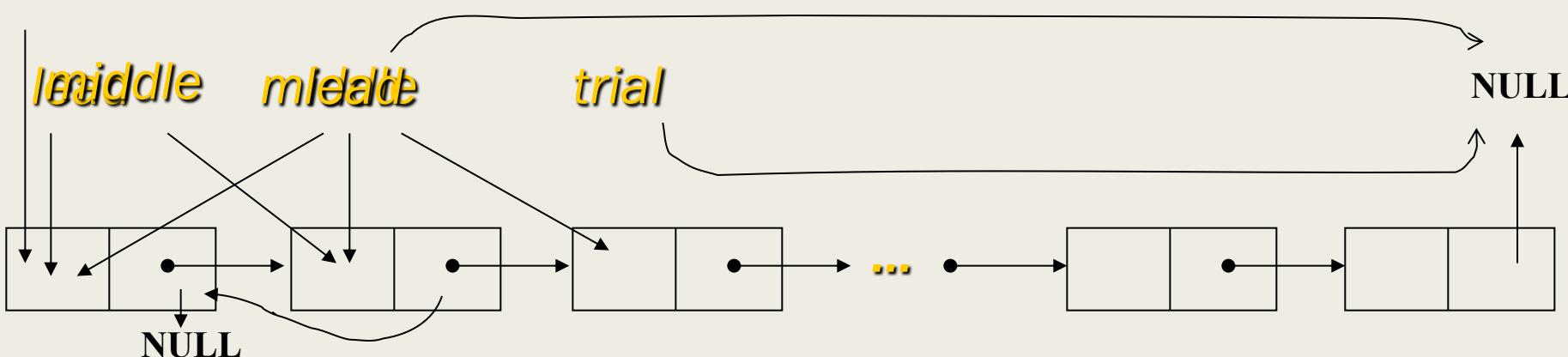
Two extra pointers

Trial : 指向目前的尾巴(後一個)

middle : 中間要被反轉的

*trial*

Program 4.17: Inverting a singly linked list



# Chain (3/3)

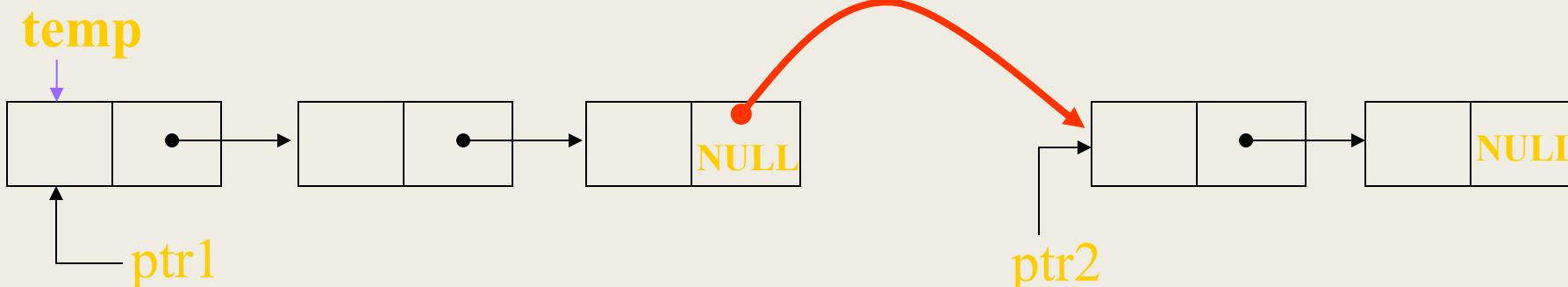
## ■ Concatenates two chains

- Concatenates two *chains*, *ptr1* and *ptr2*.
- Assign the list *ptr1* followed by the list *ptr2*.

O(length of list *ptr1*)

```
list-pointer concatenate(list-pointer ptr1,
                        list-pointer ptr2)
{
    /* produce a new list that contains the list ptr1 followed
       by the list ptr2. The list pointed to by ptr1 is changed
       permanently */
    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;
    }
}
```

**Program 4.18:** Concatenating singly linked lists



# Circularly Linked Lists (1/10)

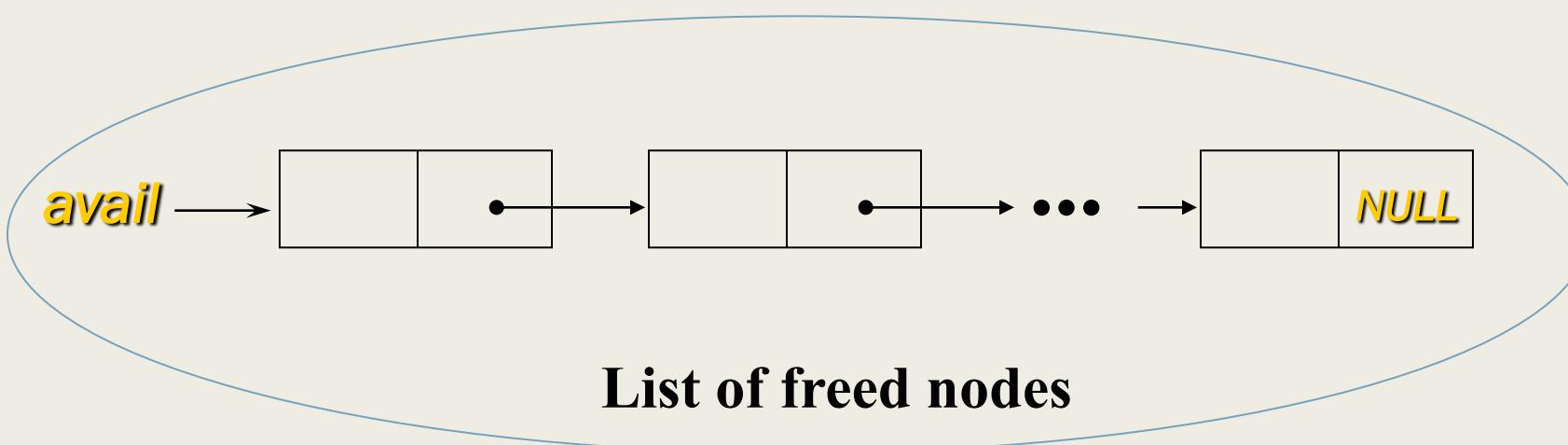
- Circular Linked list
  - *The link field of the last node points to the first node in the list.*
- Example
  - *Represent a polynomial  $ptr = 3x^{14} + 2x^8 + 1$  as a circularly linked list.*



# Circularly Linked Lists (2/10)

## ■ Maintain an Available List

- Free/Malloc 浪費時間，很多時候我們釋放了node但可能之後又會需要用到新增node
- 我們改寫一個有效率的消除，建立一個可以使用(available)的ordered node list(=chain)
- Instead of using **malloc** and **free**, we now use `get_node` (program 4.13) and `ret_node` (program 4.14).



# Circularly Linked Lists (3/10)

## ■ Maintain an Available List (cont'd)

- When we need a new node, we examine this list.
- If the list is not empty, then we may use one of its nodes.
- Only when the list is empty we do need to use **malloc** to create a new node.

---

```
poly_pointer get_node(void)
/* provide a node for use */
{
    poly_pointer node;
    if (avail) {
        node = avail;
        avail = avail->link;
    }
    else {
        node = (poly_pointer) malloc(sizeof(poly_node));
        if (IS_FULL(node)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
    }
    return node;
}
```

---

Program 4.13: *get-node* function

# Circularly Linked Lists (4/10)

- Maintain an Available List (cont'd)
- Insert **ptr** to the front of this list
  - Let **avail** be a variable of type *poly\_pointer* that points to the first node in our list of freed nodes.
  - Henceforth, we call this list the available space list or **avail** list.
  - Initially, we set **avail** to **NULL**

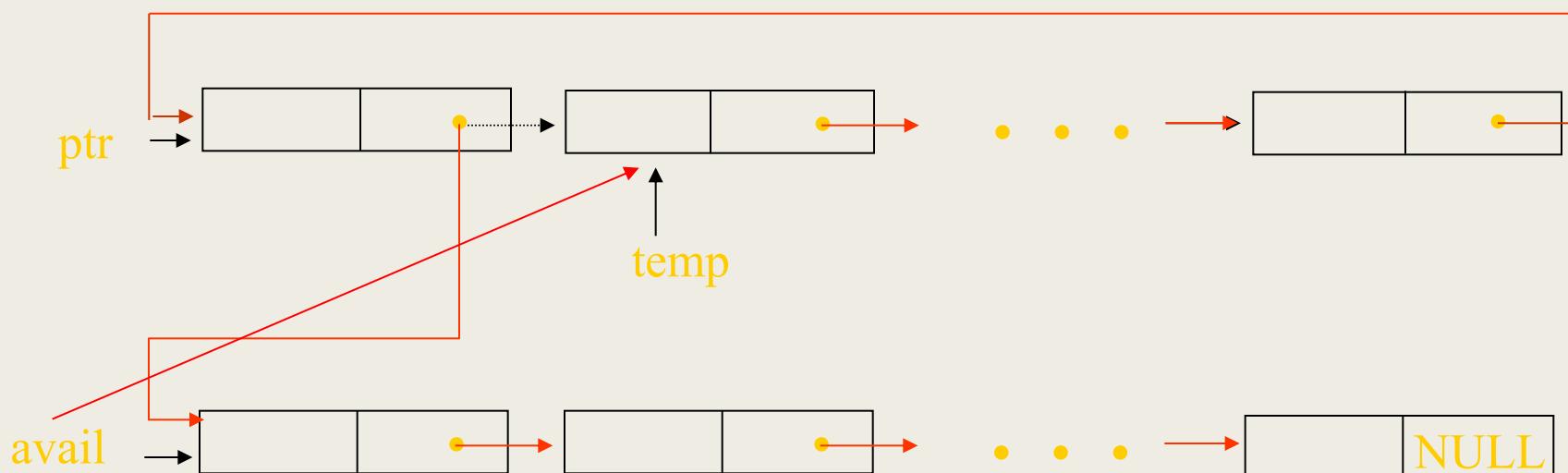
```
void ret-node(poly-pointer ptr)
{
    /* return a node to the available list */
    ptr->link = avail;
    avail = ptr;
}
```

Program 4.14: *ret-node* function

# Circularly Linked Lists (5/10)

## ■ Maintain an Available List

- Erase a circular list in a fixed amount (constant) of time  $O(1)$  independent of the number of nodes in the list using `cerase`



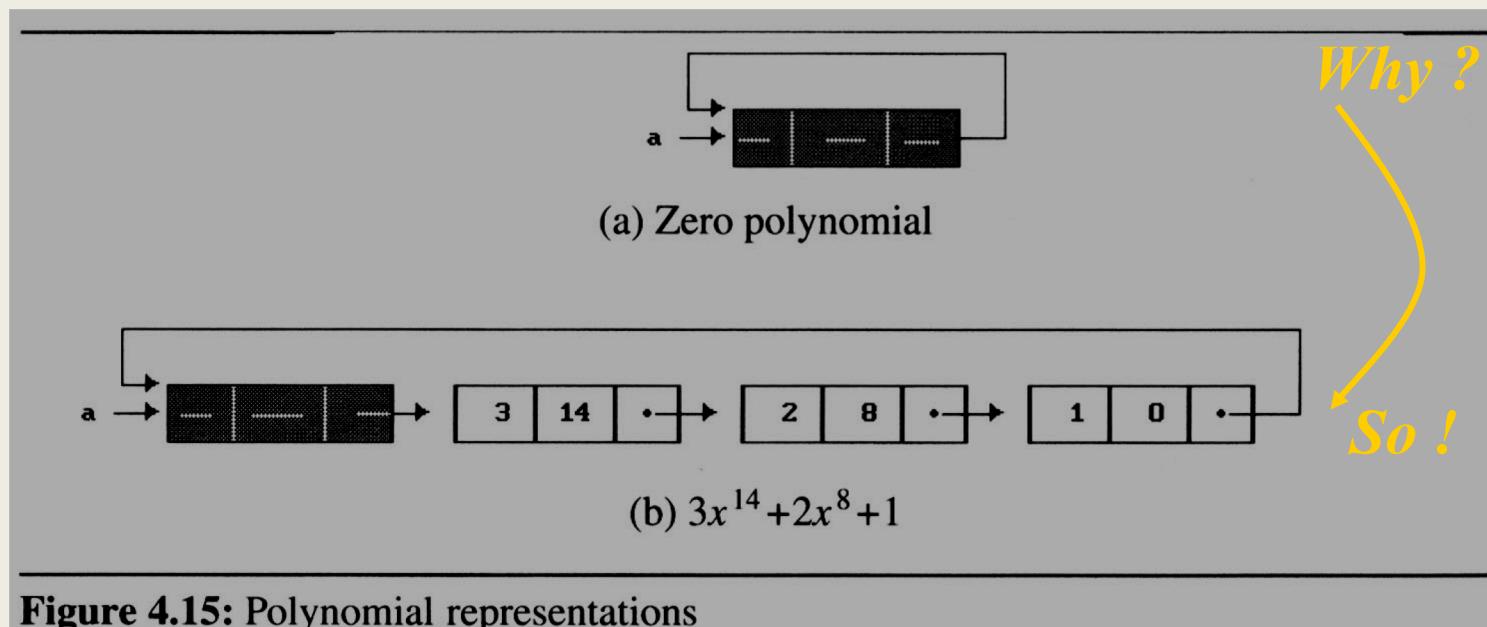
↗ 紅色link所連接而成的 chain

```
void cerase(poly_pointer *ptr)
{
    /* erase the circular list ptr */
    poly_pointer temp;
    if (*ptr) {
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```

Program 4.15: Erasing a circular list

# Circularly Linked Lists (6/10)

- We must handle the **zero polynomial** as a special case.  
To avoid it, we introduce a **head node** into each polynomial
  - *each polynomial, zero or nonzero, contains one additional node.*
  - *The expon and coef fields of this node are irrelevant.*



# Circularly Linked Lists (7/10)

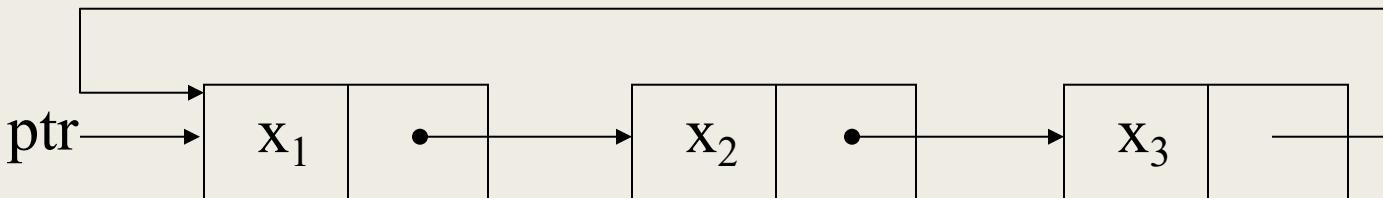
## ■ For fit the circular list with head node representation

- We may remove the test for (*\*ptr*) from cerase
- Changes the original padd to cpadd

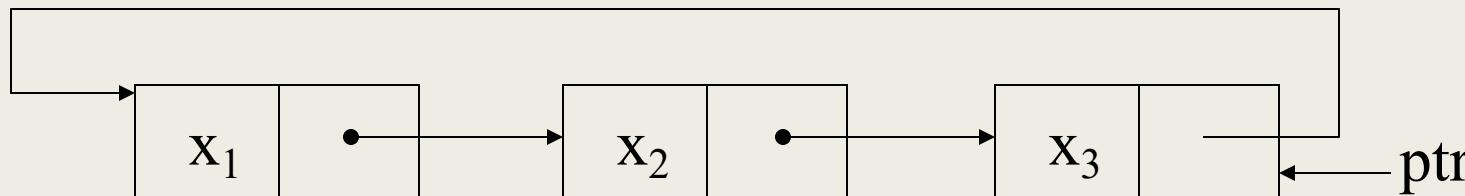
```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
    /* polynomials a and b are singly linked circular lists
     * with a head node. Return a polynomial which is the sum
     * of a and b */
    poly_pointer starta, d, lastd;
    int sum, done = FALSE;
    starta = a;           /* record start of a */
    a = a->link;         /* skip head node for a and b*/
    b = b->link;
    d = get_node();        /* get a head node for sum */
    d->expon = -1; lastd = d; /* head node */
    do {
        switch (COMPARE(a->expon, b->expon)) {
            case -1: /* a->expon < b->expon */
                attach(b->coef, b->expon, &lastd);
                b = b->link;
                break;
            case 0: /* a->expon = b->expon */
                if (starta == a) done = TRUE;
                else /*a->expon=-1, so b->exponent > -1 */
                    sum = a->coef + b->coef;
                    if (sum) attach(sum, a->expon, &lastd);
                    a = a->link; b = b->link;
                }
                break;
            case 1: /* a->expon > b->expon */
                attach(a->coef, a->expon, &lastd);
                a = a->link;
            }
        } while (!done);
    lastd->link = d; /* link to the first node */
    return d;
}
```

# Circularly Linked Lists (8/10)

- Operations for circularly linked lists
  - **Question:**
    - What happens when we want to insert a new node at the front of the circular linked list ptr?



- **Answer:**
  - move down the entire length of ptr.
- **Possible Solution:**

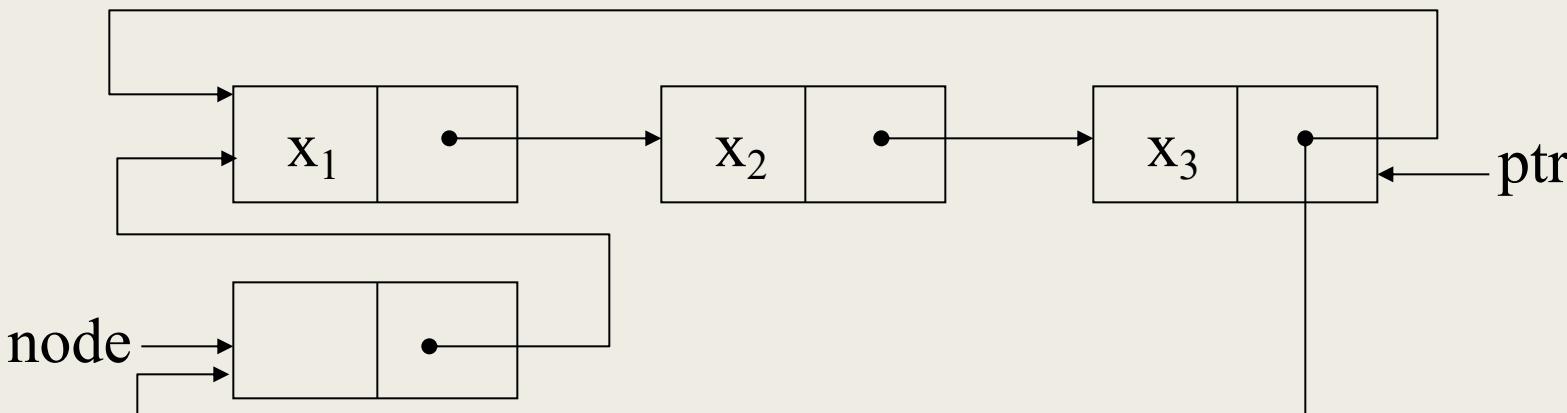


# Circularly Linked Lists (9/10)

- **Insert a new node at the front of a circular list**
- **To insert node at the rear, we only need to add the additional statement `*ptr = node` to the else clause of `insert_front`**

```
void insert_front(list_pointer *ptr, list_pointer node)
/* insert node at the front of the circular list ptr,
where ptr is the last node in the list */
{
    if (IS_EMPTY(*ptr)) {
        /* list is empty, change ptr to point to new entry */
        *ptr = node;
        node->link = node;
    }
    else {
        /* list is not empty, add new entry at front */
        node->link = (*ptr)->link;
        (*ptr)->link = node;
    }
}
```

Program 4.19: Inserting at the front of a list



# Circularly Linked Lists (10/10)

## ■ Finding the length of a circular list

---

```
int length(list_pointer ptr)
{
    /* find the length of the circular list ptr */
    list_pointer temp;
    int count = 0;
    if (ptr) {
        temp = ptr;
        do {
            count++;
            temp = temp->link;
        } while (temp != ptr);
    }
    return count;
}
```

---

**Program 4.20:** Finding the length of a circular list

# Equivalence Relations (1/6)

- Reflexive Relation

- *for any polygon  $x$ ,  $x \equiv x$  (e.g.,  $x$  is electrically equivalent to itself)*

- Symmetric Relation

- *for any two polygons  $x$  and  $y$ , if  $x \equiv y$ , then  $y \equiv x$ .*

- Transitive Relation

- *for any three polygons  $x$ ,  $y$ , and  $z$ , if  $x \equiv y$  and  $y \equiv z$ , then  $x \equiv z$ .*

- Definition:

- *A relation over a set,  $S$ , is said to be an **equivalence relation** over  $S$  iff it is **symmetric**, **reflexive**, and **transitive** over  $S$ .*

- Example:

- *“equal to” relationship is an equivalence relation*

## ■ Example: Equivalence Relations (2/6)

- if we have 12 polygons numbered 0 through 11

$0 \equiv 4$ ,  $3 \equiv 1$ ,  $6 \equiv 10$ ,  $8 \equiv 9$ ,  $7 \equiv 4$ ,  $6 \equiv 8$ ,  $3 \equiv 5$ ,  $2 \equiv 11$ ,  $11 \equiv 0$

- we can partition the twelve polygons into the following equivalence classes:

$$\{0, 2, 4, 7, 11\}; \{1, 3, 5\}; \{6, 8, 9, 10\}$$

## ■ Two phases to determine equivalence

- First phase: the equivalence pairs  $(i, j)$  are read in and stored.
- Second phase:
  - we begin at 0 and find all pairs of the form  $(0, j)$ . Continue until the entire equivalence class containing 0 has been found, marked, and printed.

## ■ Next find another object not yet output, and repeat the above process.

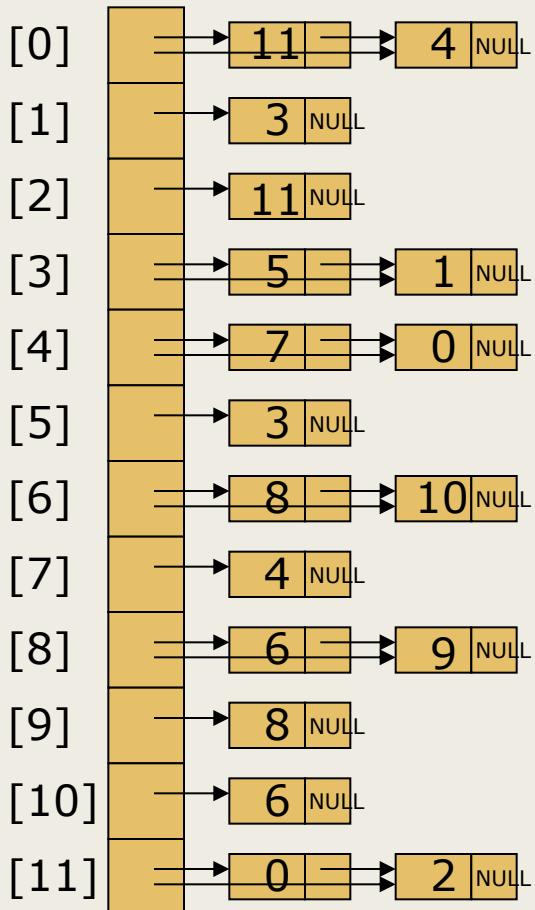
# Equivalence Relation (3/6)

- Program to find equivalence classes

```
void main(void){                                #include <stdio.h>
    short int out[MAX_SIZE];                  #define MAX_SIZE    24
    node_pointer seq[MAX_SIZE];              #define IS_FULL(ptr) (!ptr)
    node_pointer x, y, top;                  #define FALSE     0
    int i, j, n;                            #define TRUE      1
    printf("Enter the size (<=%d) ", MAX_SIZE);
    scanf("%d", &n);
    for(i=0; i<n; i++){                      typedef struct node *node_pointer;
  /*initialize seq and out */ typedef struct node {
        out[i] = TRUE;    seq[i] = NULL; int data;
    }   node_pointer link;
    /* Phase 1 */                           };
    /* Phase 2 */
}
```

# Equivalence Relations (4/6)

- Phase 1: read in and store the equivalence pairs  $\langle i, j \rangle$



```
/* Phase 1: Input the equivalence pairs: */
printf("Enter a pair of numbers (-1 -1 to quit): ");
scanf("%d%d",&i,&j);
while (i >= 0) {
    x = (node_pointer)malloc(sizeof(node));
    if (IS_FULL(x)) {
        fprintf(stderr,"The memory is full\n");
        exit(1);
    }
    x->data = j; x->link = seq[i]; seq[i] = x;
    x = (node_pointer)malloc(sizeof(node));
    if (IS_FULL(x)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    x->data = i; x->link = seq[j]; seq[j] = x;
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d",&i,&j);
}
```

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

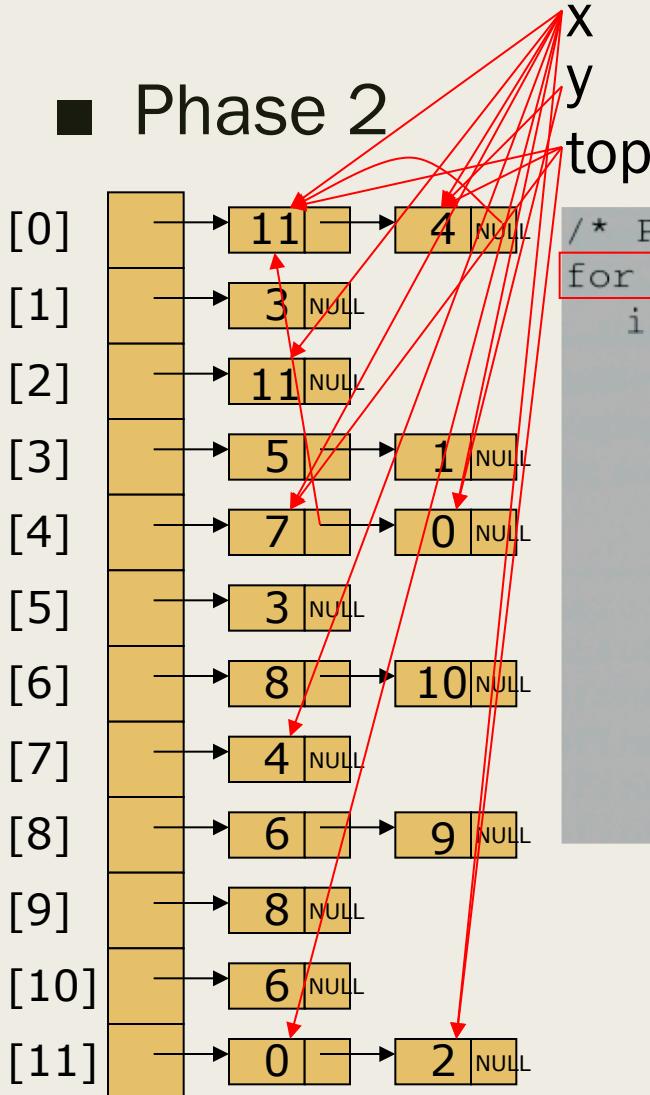
# Equivalence Relations (5/6)

## ■ Phase 2:

- begin at 0 and find all pairs of the form  $\langle 0, j \rangle$ , where 0 and  $j$  are in the same equivalence class
- by transitivity, all pairs of the form  $\langle j, k \rangle$  imply that  $k$  is in the same equivalence class as 0
- continue this way until we have found, marked, and printed the entire equivalent class containing 0

# Equivalence Relations (6/6)

## Phase 2



$i = \emptyset$   $j = 0$

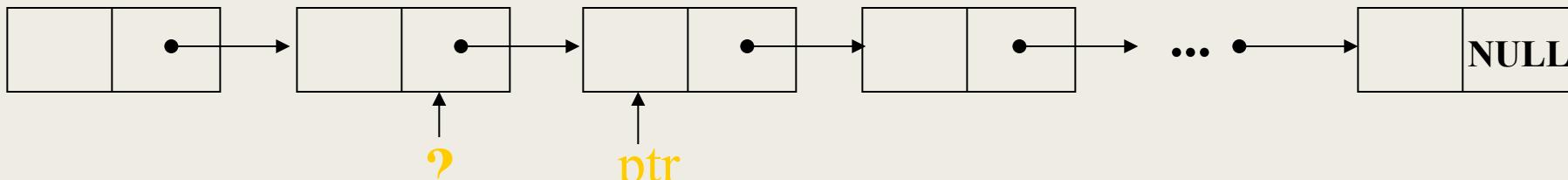
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]  
out:

```
/* Phase 2: output the equivalence classes */
for (i = 0; i < n; i++) {
    if (out[i]) {
        printf("\nNew class: %5d", i);
        out[i] = FALSE; /* set class to false */
        x = seq[i]; top = NULL; /* initialize stack */
        for (;;) { /* find rest of class */
            while (x) { /* process list */
                j = x->data;
                if (out[j]) {
                    printf("%5d", j); out[j] = FALSE;
                    y = x->link; x->link = top; top = x; x = y;
                }
                else x = x->link;
            }
            if (!top) break;
            x = seq[top->data]; top = top->link; /*unstack*/
        }
    }
}
```

New class: 0 11 4 7 2

# Doubly Linked Lists (1/4)

- Singly linked lists pose problems because we can move easily only in the direction of the links



- Doubly linked list has at least three fields
  - *left link field(llink), data field(item), right link field(rlink).*
  - *The necessary declarations:*

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

# Doubly Linked Lists (2/4)

## ■ Sample

- *doubly linked circular with head node: (Figure 4.23)*

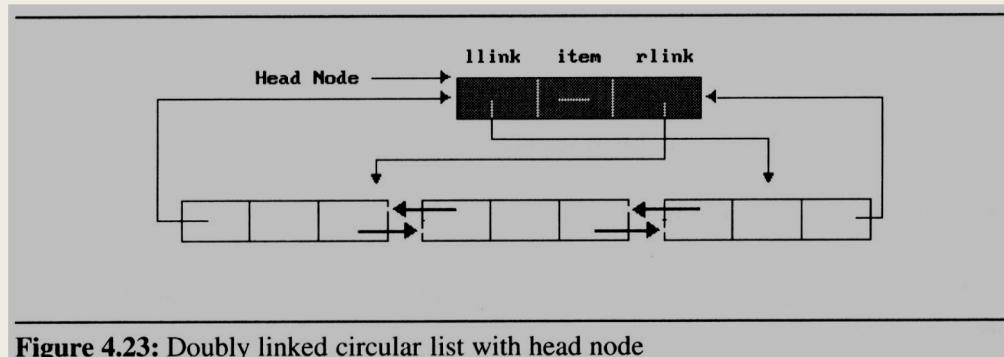


Figure 4.23: Doubly linked circular list with head node

- *empty double linked circular list with head node (Figure 4.24)*

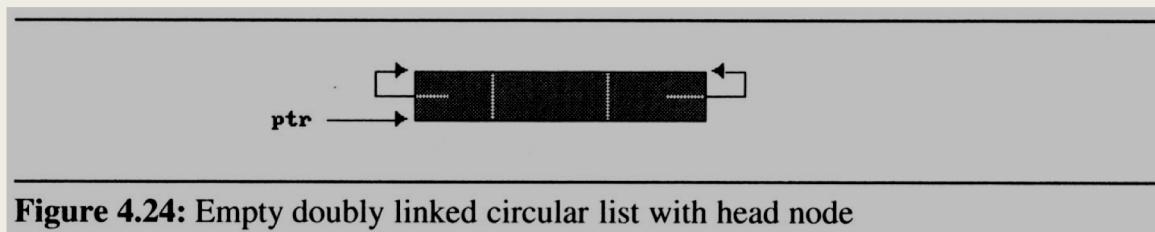


Figure 4.24: Empty doubly linked circular list with head node

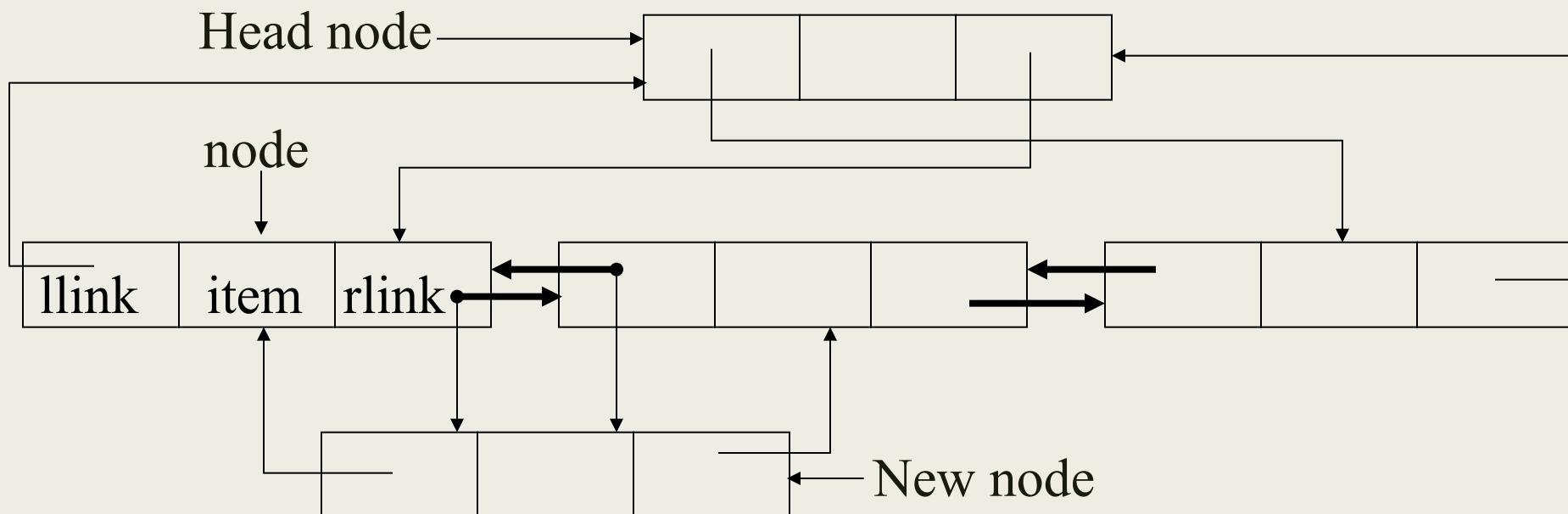
- *suppose that ptr points to any node in a doubly linked list, then:*
  - $\text{ptr} = \text{ptr} \rightarrow \text{llink} \rightarrow \text{rlink} = \text{ptr} \rightarrow \text{rlink} \rightarrow \text{llink}$

# Doubly Linked Lists (3/4)

## ■ Insert node

```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->llink = node;
    newnode->rlink = node->rlink;
    node->rlink->llink = newnode;
    node->rlink = newnode;
}
```

**Program 4.28:** Insertion into a doubly linked circular list



# Doubly Linked Lists (4/4)

## ■ Delete node

```
void ddelete(node_pointer node, node_pointer deleted)
{
    /* delete from the doubly linked list */
    if (node == deleted)
        printf("Deletion of head node not permitted.\n");
    else {
        deleted->llink->rlink = deleted->rlink;
        deleted->rlink->llink = deleted->llink;
        free(deleted);
    }
}
```

Program 4.29: Deletion from a doubly linked circular list

