LISTS

Introduction

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

Pointer Review (1)

Pointer Can Be Dangerous

int i, *pi;

	1000		2000
i	?	pi	?

pi = &i;

	1000		2000
i *pi	?	pi	1000

i = 10 or *pi = 10

	1000		2000
i *pi	10	pi	1000

- Set to NULL
- Explicit typecasts

Pointer Review (2)

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

1000

ptr NULL

ptr->datað(*ptr).data

ptr = malloc(sizeof(list_node));

1000 2000 *ptr

ptr 2000

data link

4.2 SINGLY LINKED LISTS

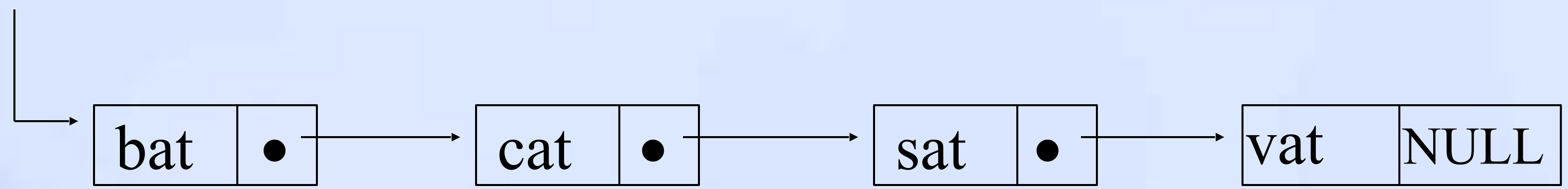


***Figure 4.1:** Usual way to draw a linked list*

Insertion

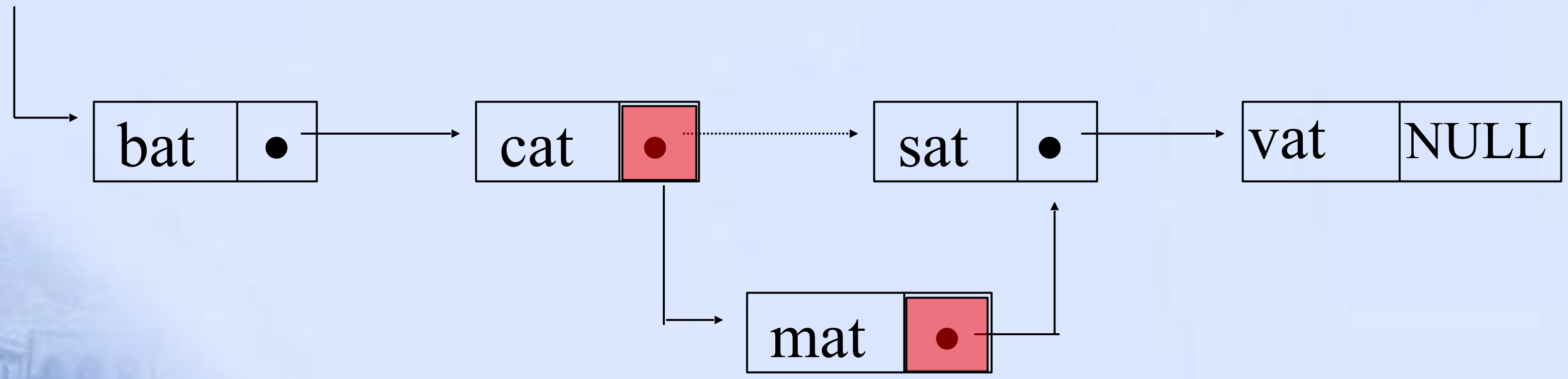


***Figure 4.2:** Insert mat after cat*

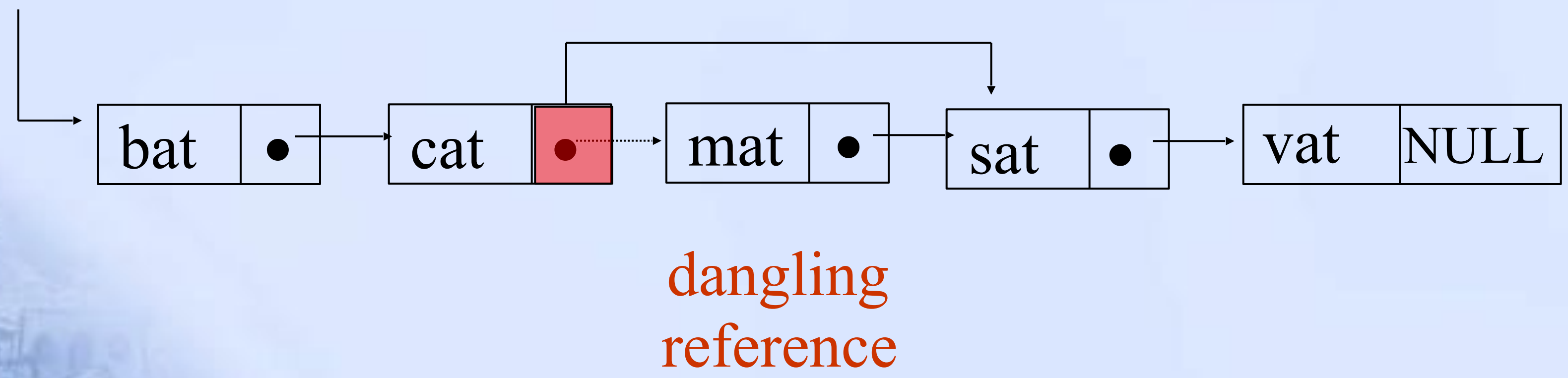


***Figure 4.3:** Delete *mat* from list*

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));
```

```
strcpy(ptr -> data, “bat”);
ptr -> link = NULL;
```

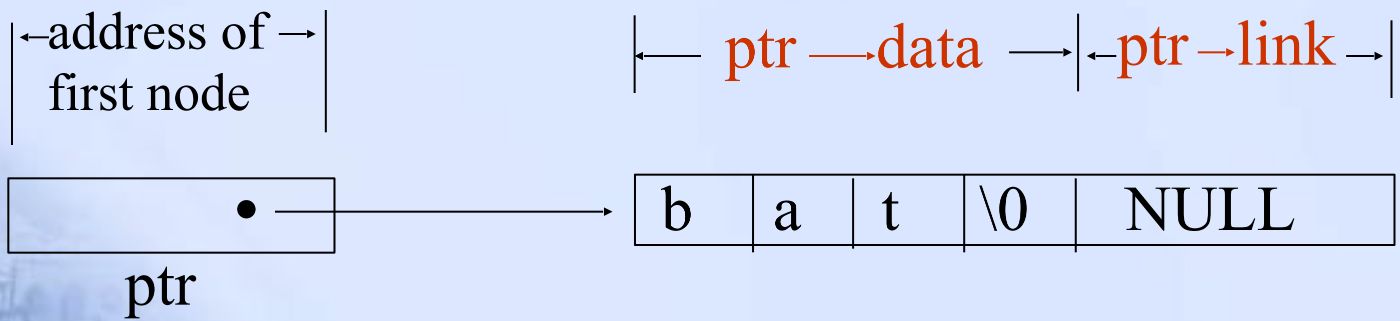
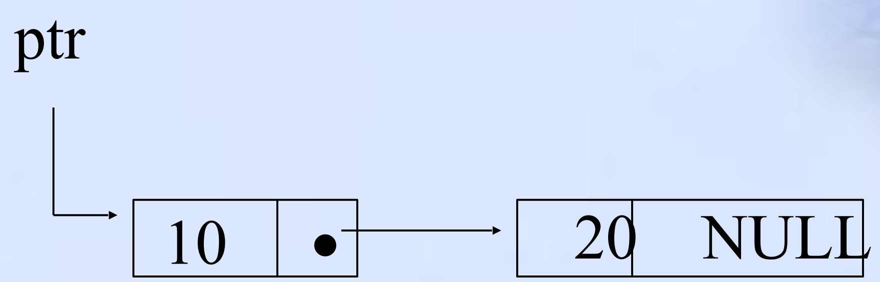


***Figure 4.4:**Referencing the fields of a node(p.142)

Example: create a two-node list

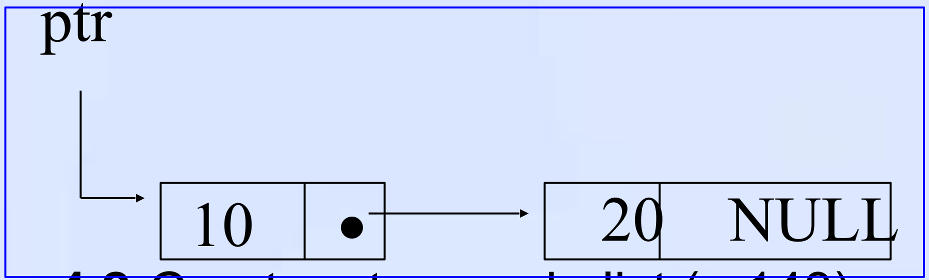



```
typedef struct list_node *list_pointer;
struct list_node {
        int data;
        list_pointer link;
        };
list_pointer ptr =NULL
```

Example 4.2: (p.142)

```
list_pointer create2( )
{
/* create a linked list with two nodes */
    list_pointer first, second;

    first = (list_pointer) malloc(sizeof(list_node)); // new list_node;
    second = ( list_pointer) malloc(sizeof(list_node));//new list_node
    second -> link = NULL;
    second -> data = 20;
    first -> data = 10;
    first ->link = second;
    return first;
}
```



***Program 4.2:**Create a tow-node list (p.143)

List Insertion:

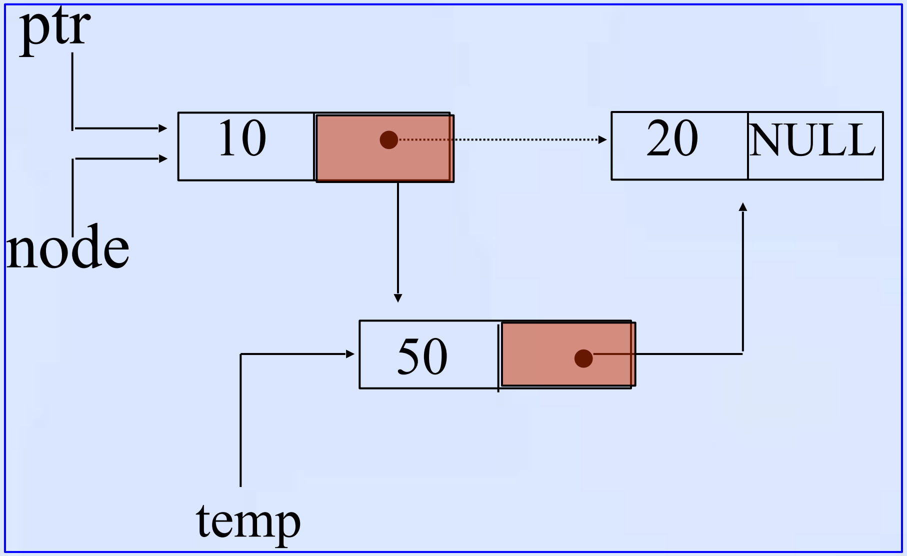
Insert a node after a specific node

```
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)){
        cout<<"The memory is full\n";
        exit (1);
    }
```

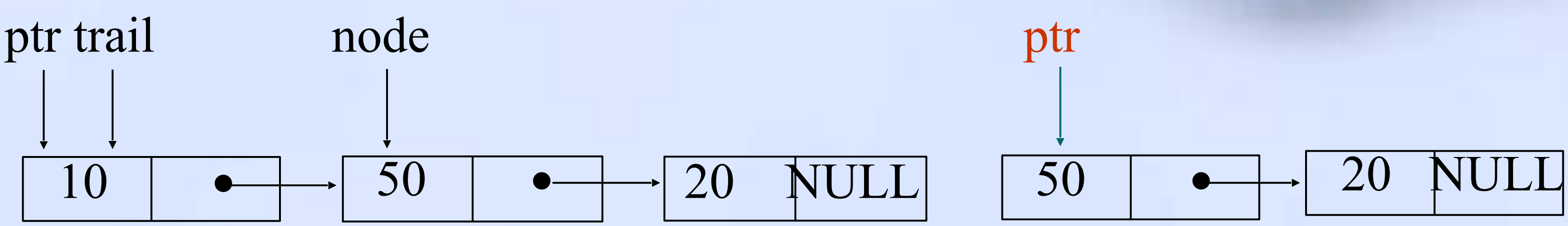
```
temp->data = 50;
 if (*ptr) {  noempty list
     temp->link =node ->link;
     node->link = temp;
}
else {    empty list
    temp->link = NULL;
    *ptr =temp;
 }
}
```



***Program 4.3:**Simple insert into front of list (p.144)

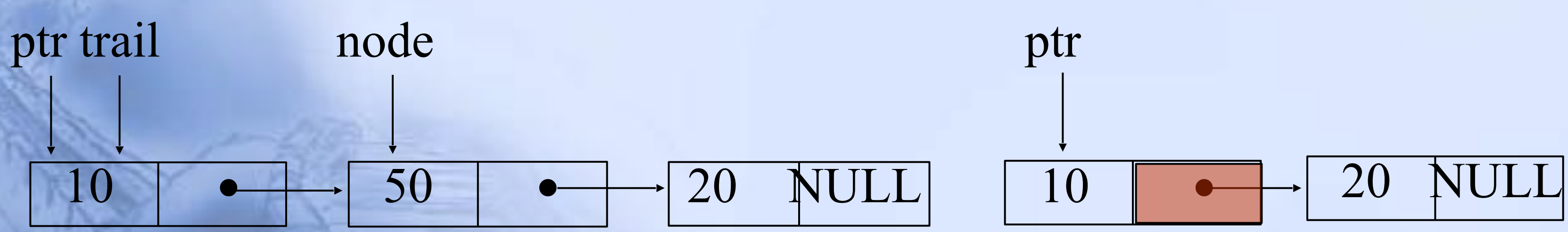
List Deletion

Delete the first node.

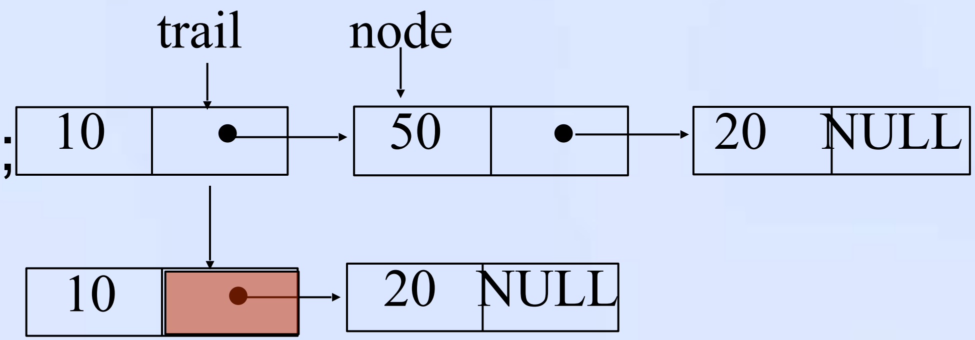


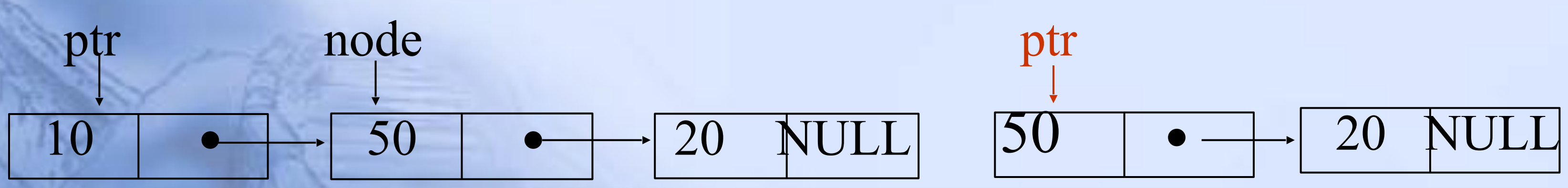
(a) before deletion (b)after deletion

Delete 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 4.5:** Printing a list (p.146)

4.3 DYNAMICALLY LINKED STACKS AND QUEUES

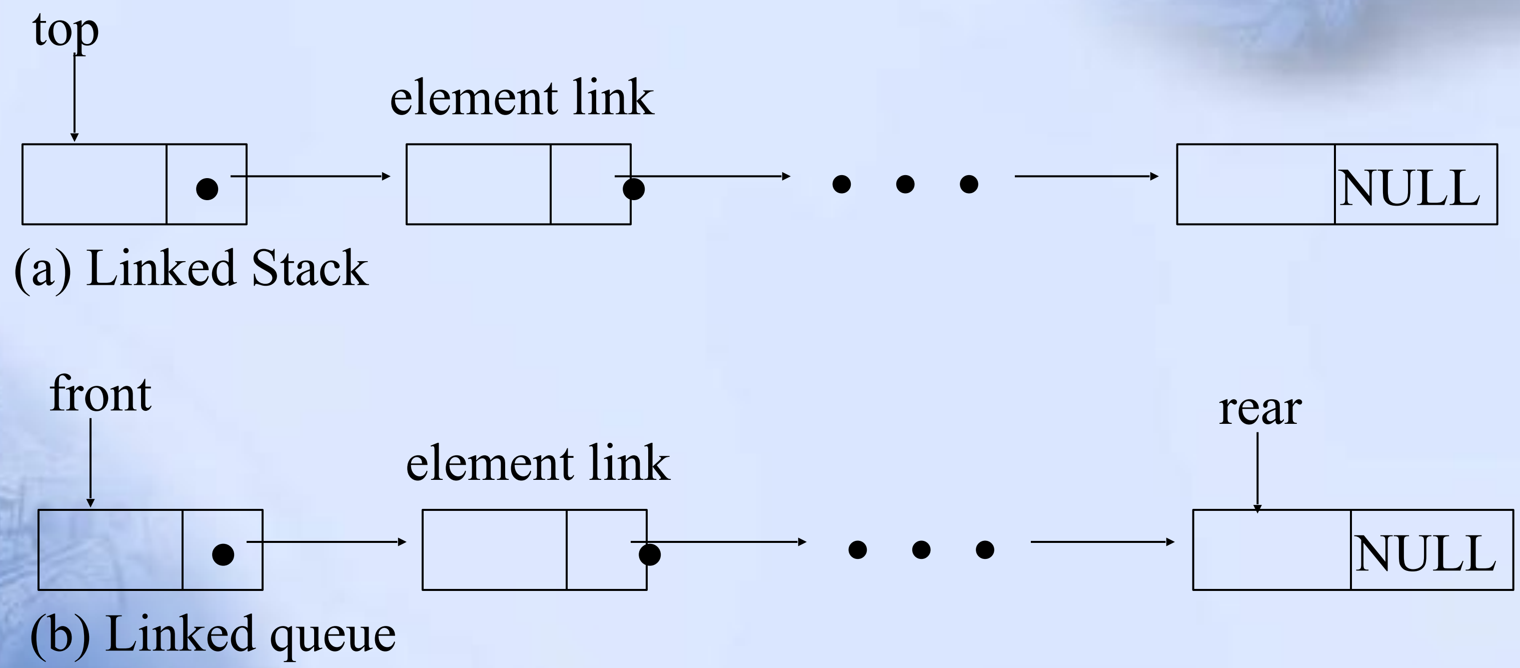


***Figure 4.10:** Linked Stack and queue (p.147)*

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)

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

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

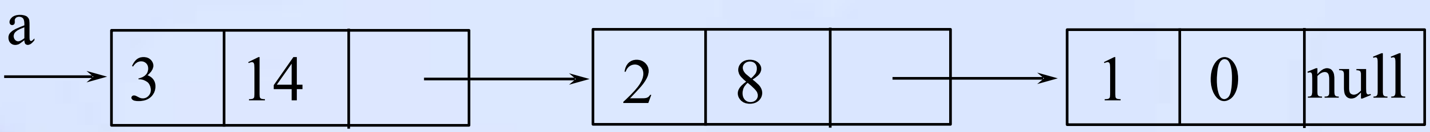
Representation

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

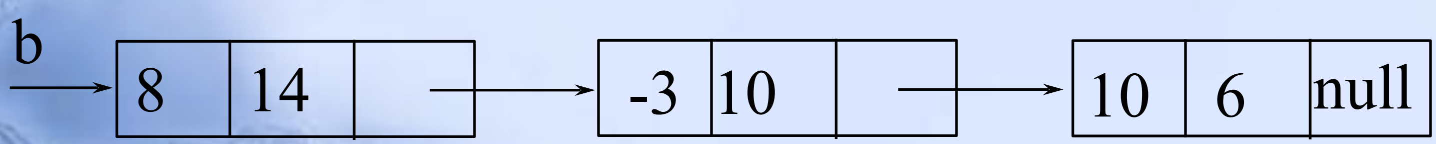
coef	expon	link

Examples

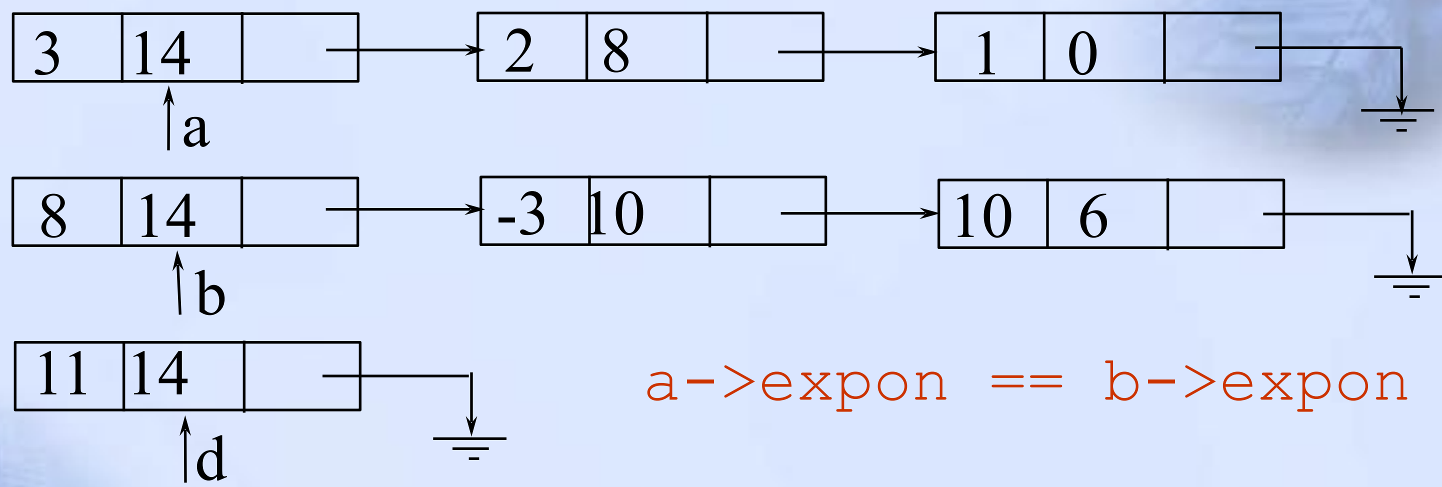
$$a = 3x^{14} + 2x^{8} + 1$$



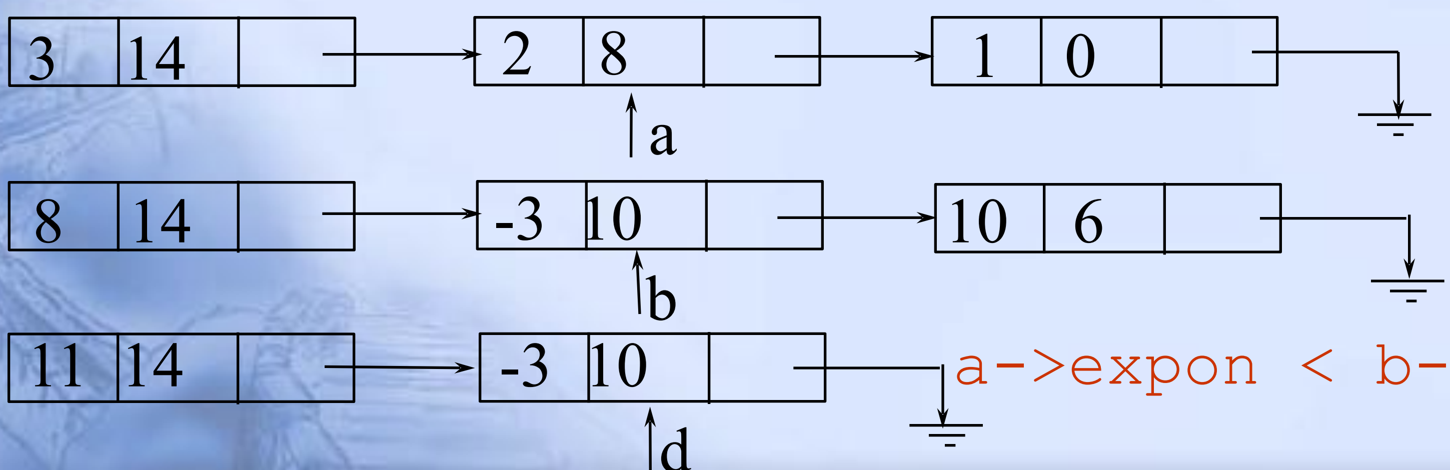
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Adding Polynomials

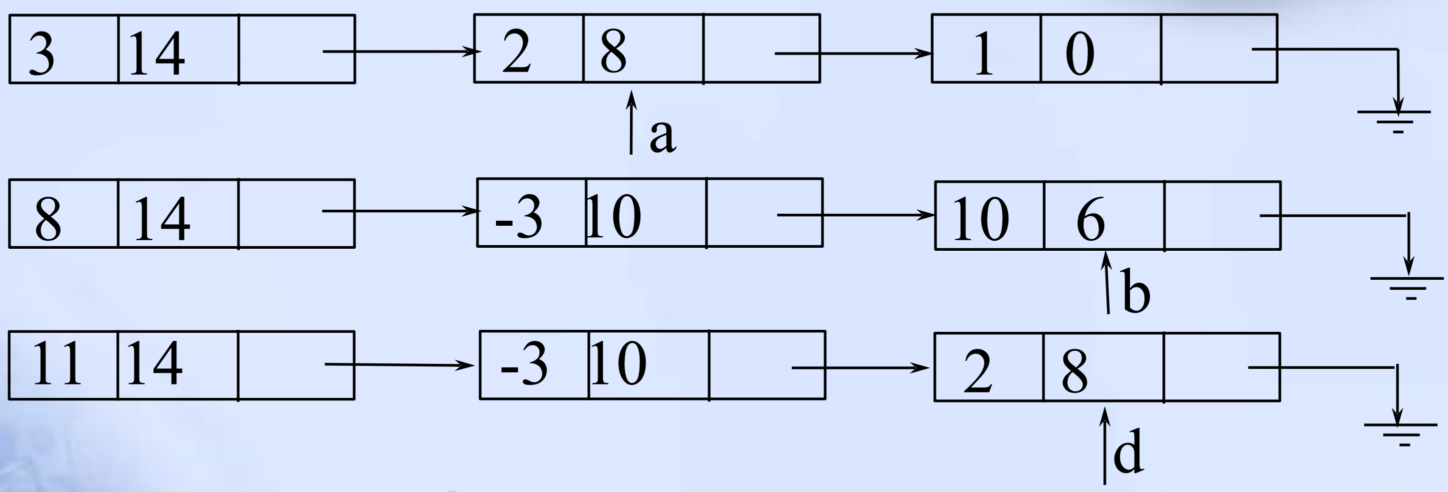


a->expon == b->expon



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;

     //create a dummy node
    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;

    //delete dummy node;
    temp = front;  front = front->link;  free(temp);
    return front;
}
```

Delete extra initial node.

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;
}
```

Analysis

(1) coefficient additions

$0 \leq additions \leq \min\{m, n\}$

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

(2) exponent comparisons

extreme case

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

m+n-1 comparisons

(3) creation of new nodes

extreme case

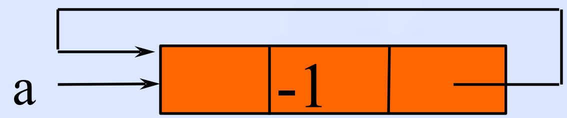
m + n new nodes

summary $O(m+n)$

Head Node

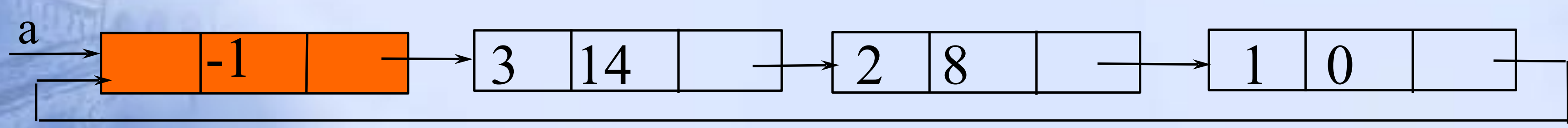
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



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

Another Padd

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
  poly_pointer starta, d, lastd;
  int sum, done = FALSE;
  starta = a;
  a = a->link;
  b = b->link;
  d = get_node();
  d->expon = -1;    lastd = d;
  do {
    switch (COMPARE(a->expon, b->expon)) {
      case -1: attach(b->coef, b->expon, &lastd);
               b = b->link;
               break;
```

Set expon field of head node to -1.

Another Padd *(Continued)*

```
      case 0: if (starta == a) done = TRUE;
              else {
                 sum = a->coef + b->coef;
                 if (sum) attach(sum,a->expon,&lastd);
                 a = a->link;   b = b->link;
              }
              break;
      case 1: attach(a->coef,a->expon,&lastd);
              a = a->link;
    }
  } while (!done);
  lastd->link = d;
  return d;
}
```

Link last node to first

Additional List Operations

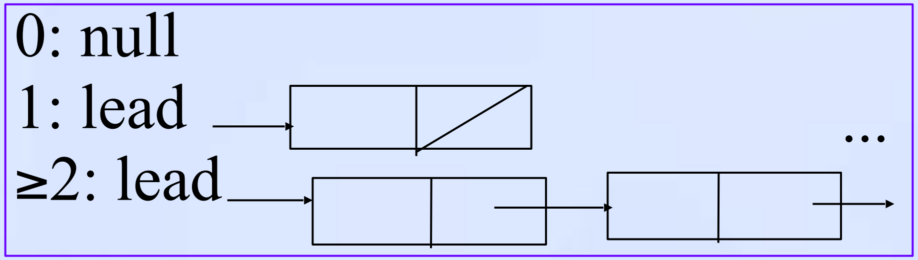
```
typedef struct list_node *list_pointer;
typedef struct list_node {
    char data;
    list_pointer link;
};
```

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;
}
```



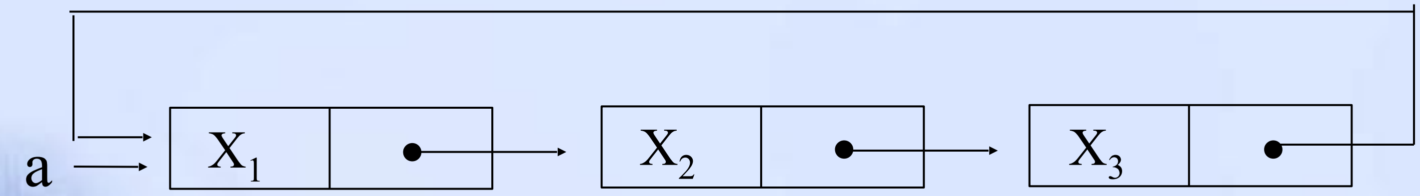
Concatenate Two Lists

```
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

4.5.2 Operations For Circularly Linked List

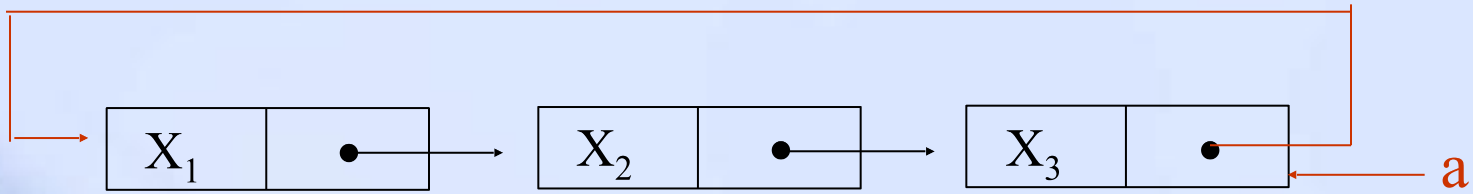
What happens when we insert a node to the front of a circular linked list?



Problem: move down the whole list.

***Figure 4.16:** Example circular list (p.165)

A possible solution:

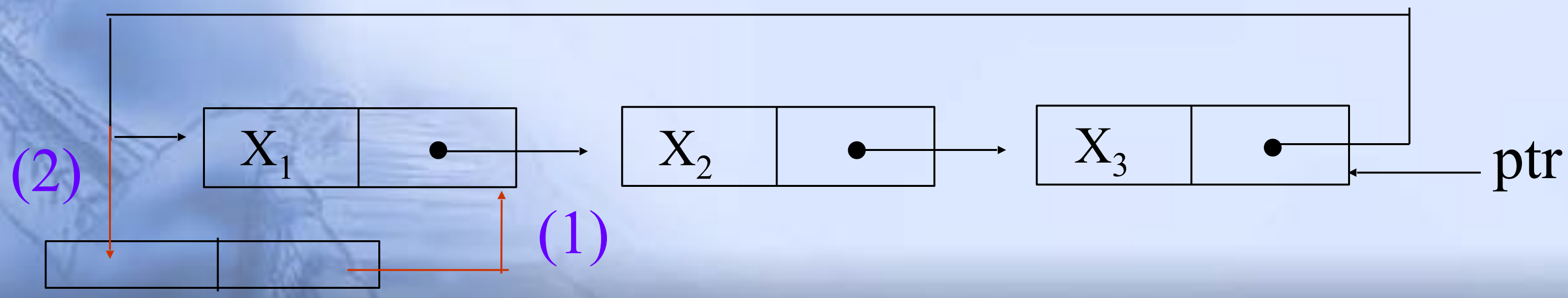


Note a pointer points to the last node.

***Figure 4.17:** Pointing to the last node of a circular list (p.165)*

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;
}
```

4.8 Doubly Linked List

Move in forward and backward direction.

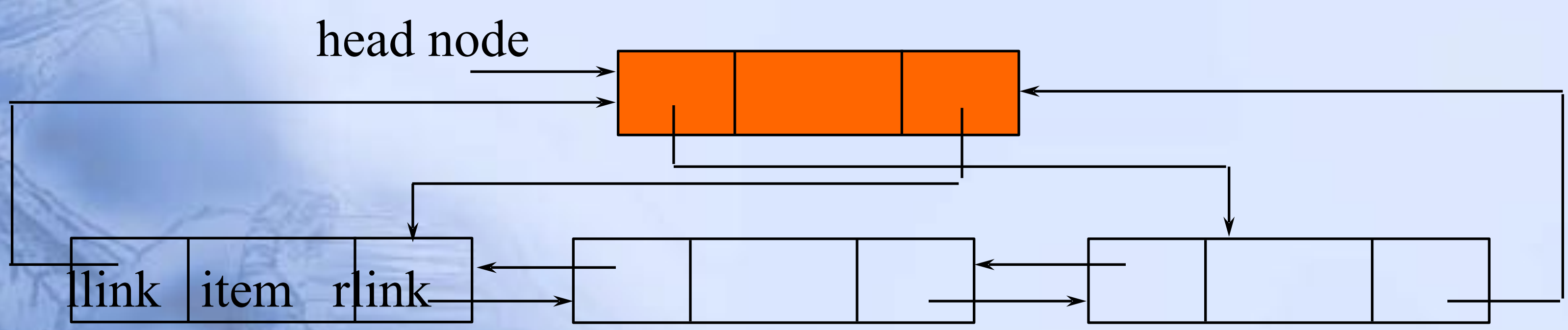
Singly linked list (*in one direction only*)
How to get the preceding node during deletion or insertion?
Using 2 pointers

Node in doubly linked list consists of:
1. *left link field* (llink)
2. *data field* (item)
3. *right link field* (rlink)

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



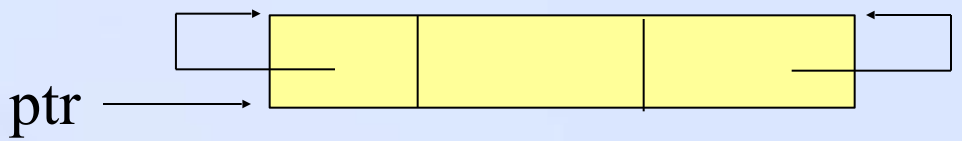


***Figure 4.24:**Empty doubly linked circular list with head node (p.180)

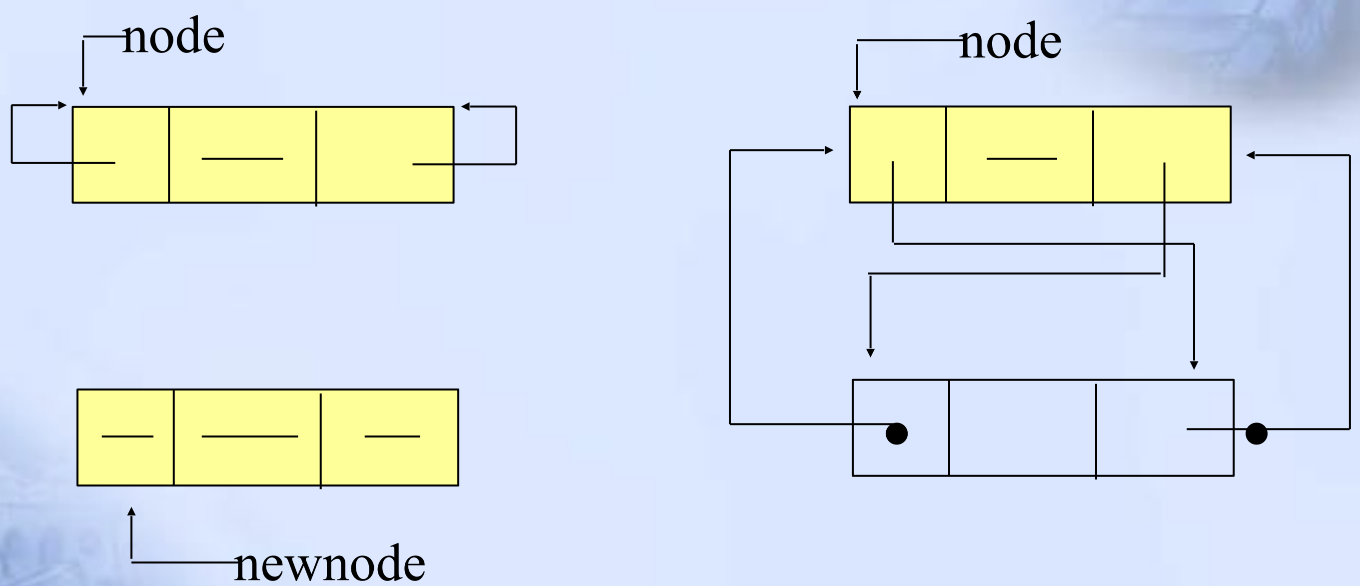
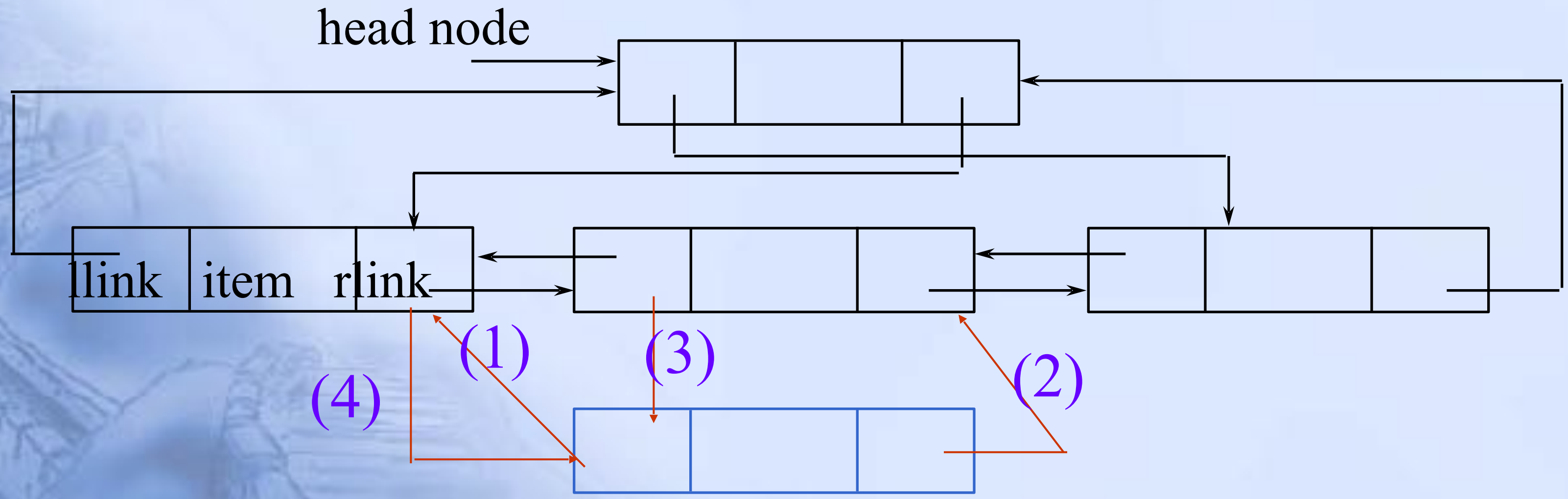


***Figure 4.25:** Insertion into an empty doubly linked circular list (p.181)*

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);
    }
}
```