Chapter 2 Linerar List

outline

- **Definition of ADT**
- **Sequential List**
- **Singly Linked List**
- **Circular Linked List**
- **Doubly Linked List**
- **Applications**

2.1 Definition

- **Linear list**
- **Length of list**
- **Empty list**
- **Order**

ADT of Linear List

- ADT List {
- Data object： D＝{ ai | ai ∈ElemType, i=1,2,...,n, n≥0 }
- Relation ： R1＝{ <ai-1 ,ai >|ai-1 ,ai∈D, i=2,...,n }
- Operations ：
- InitList(&L) 。
- DestroyList(&L)
- ListEmpty(L)
- ListLength(L)
- PriorElem(L, cur_e, &pre_e)
- NextElem(L, cur_e, &next_e)
- GetElem(L, i, &e)
- LocateElem(L, e, compare( ))
- ListTraverse(L, visit( ))
- ClearList(&L)
- PutElem(& L, i, e)
- ListInsert(&L, i, e)
- ListDelete(&L, i, &e）
- } ADT List

Examples for using the basic operations

- **Set Union A＝A∪B**
- **purge（L）**
- **Merging two ordered list**

Set union

- void union(List &La, List Lb) {
- //
- La_len = ListLength(La);
- Lb_len =ListLength(Lb); // length
- for (i = 1; i <= Lb_len; i++) {
- GetElem(Lb, i, e);// e is the ith element
- if(!LocateElem(La, e, equal( ))
-   ListInsert(La, ++La_len, e);//}
- } // union

Remove all repetitive elements

- void purge(L)
- {
- int i=1;j,x,y;
- while (i<ListLength(L))
- {Getelem(L,i,x);
- j=i+1;
- while (j<listLength(L)）
- {Getelem(L,j,y);
- if (x==y) ListDelete(L,j);
- else j++;
- }
- i++;
- }
- }

Merge orderedlist

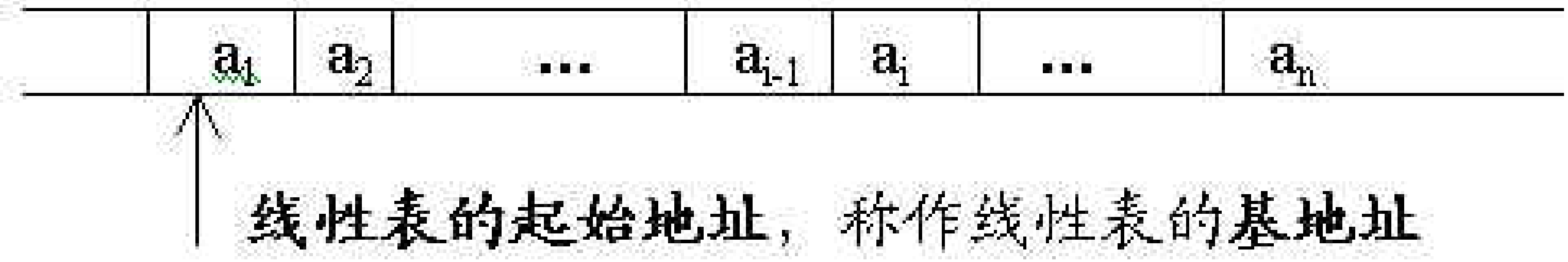
- void MergeList(List La, List Lb, List &Lc) {
- InitList(Lc);
- i = j = 1; k = 0;
- La_len = ListLength(La); Lb_len = ListLength(Lb);
- While (I <= La_len) && (J <= Lb_len) { // La and Lb's all non-empty
- GetElem(La, i, ai); GetElem(Lb, j, bj);
- if (ai <= bj) { ListInsert(Lc, ++k, ai); ++i; }
- else { ListInsert(Lc, ++k, bj); ++j; }
- }
- while (i <= La_len) {
- GetElem(La, i++, ai);
- ListInsert(Lc, ++k, ai);
- }
- while (j <= Lb_len) {
- GetElem(Lb, j++, bj);
- ListInsert(Lc, ++k, bj);
- }
- } // merge_list

Discussion on the merging

- Assuming that ，A does not include the repetition elements and B is similar to A ， how to remove the repetition element when merging A with B ？
  - if (ai < bj) { ListInsert(Lc, ++k, ai); ++i; }
  - else if (ai > bj) { ListInsert(Lc, ++k, bj); ++j; }
  - else i++;/*{ListInsert(Lc, ++k, bj); ++I;++j; }*/
  - First merging then Purge
- If A and B may possess the repetition elements How to do ？

2.2 Sequential list

The elements are stored in a consecutive storage area one by one



Notes :

- With ordered pair $<a_{i-1}, a_i>$ to express "Storage is adjacent to", $loc(a_i) = loc(a_{i-1}) + C$
- Unnecessary to store logic relationship
- First data component location can decide all data elements locations

$$LOC(a_i) = \underline{LOC(a_1)} + (i-1) \times C$$

↑ 基地址

Sequential storage map definition

- The data component is depicted in the way of the array :
- typedef struct {
- ElemType data[maxsize];
- int length; //'s current length
- } SqList;

The data component is stored in the way of the pointer :

- //----- Sequential storage organization of the dynamic allocation of linear list -----
- #define LIST_INIT_SIZE 80
- #define LISTINCREMENT 10
- typedef struct {
-   ElemType *elem; //'s dedicated space base
-   int length; //'s current length
-   int listsize; // The distributed memory capability
- } SqList;

Sequential map implementations

1. Initialization of linear list

- array：
- pointer

```
Status InitList_Sq(SqList &L) {
// Constitute a hollow linear list L 。
L.elem = (ElemType *)
      malloc(LIST_INIT_SIZE *sizeof(ElemType));
if (!L.elem) exit(OVERFLOW); //'s memory allocation is fail
L.length = 0; //'s length is 0
L.listsize = LIST_INIT_SIZE ; //'s intial stage memory capability
return OK;
} // InitList_Sq
```

2. LocateElement by content：

- A. array：
- B. pointer

```
int LocateElem_Sq(SqList L, ElemType e,Status (*compare)(ElemType, ElemType))
    {
// using  Compare ( )
// If finding ，  return the index otherwise  return 0 。
i = 1; // The initial value of I is the the 1st element
p = L.elem; // The initial value of P is the the 1st element storage site
while (i <= L.length && !(*compare)(*p++, e))  ++i;
if (i <= L.length) return i;
else return 0;
} // LocateElem_Sq
```

This algorithm time complexity is : O(ListLength(L))

Discussion：

- Unnecessary to have LocateElem (L ,i) function
- Searching frame
- Modification of searching algorithm ---Use the sentinel

3. ListInsert (&L) :

- A. the array :
- B. the pointer

Ask : When inserting the element , What does the logical organization of linear list change ?

$(a_1, \ldots, a_{i-1}, a_i, \ldots, A_n)$ revises $(A_1\ \ldots, a_{i-1}, e, a_i, \ldots, a_n)$

```
Status ListInsert_Sq(SqList &L, int pos, ElemType e) {
// fresh element E can be inserted in  sequential linear list L element
// The validate value of Pos is in the range of 1≤pos≤Listlength_Sq(L)+1
if (pos < 1 || pos > L.length+1) return ERROR;
If ( L.length >= L.listsize ) {
  newbase = (ElemType *)realloc(L.elem,(L.listsize+LISTINCREMENT)*sizeof (ElemType));
  if (!newbase) exit(OVERFLOW); //'s memory allocation is fail
  L.elem = newbase; // fresh base
  L.listsize += LISTINCREMENT; // adds the memory capability
}
```

```
q = &(L.elem[pos-1]);
for (p = &(L.elem[L.length-1]); p >= q; --p)
  *(p+1) = *p;
*q = e;
++L.length;
return OK;
} // ListInsert_Sq
```

The algorithm time complexity is : O(ListLength(L))

Discussion：

- If afterwards insertions ？
- Pay attention to the relationship between the initial value assignment with moves
- If inserting more than one (M) elements ？
  - M+L.length?L.listsize
  - Move mode

4. the ListDelete (&L) realization：

- Using array
- Using pointer

Ask ： When deleting the element ， What does the logical organization of linear list change ?

(a_1, …, a_{i-1}, a_i, a_{i+1}, …, A_n) the alteration is (A_1 …, a_{i-1}, a_{i+1}, …, a_n)

```
Status ListDelete_Sq(SqList &L, int pos, ElemType &e) {
// The legality of Pos is in therange of 1 ≤pos≤ListLength_Sq(L)
if ((pos < 1) || (pos > L.length)) return ERROR;
p = &(L.elem[pos-1]); // P act as the element's place to be deleted
e = *p; // the element value assigns to E
q = L.elem+L.length-1; // tail element place
for (++p; p <= q; ++p) *(p-1) = *p; // Element left shift
--L.length; //'s length reduces 1
return OK;
} // ListDelete_Sq
```

The algorithm time complexity is : O(ListLength(L))

Discussion ：

- Relationship between the moving with the initial value
- If to delete more than one elements？
  - Place Pos+m ？ L.length
  - Move mode

Assignment 1 :

- **To give an example to illustrate data structure idea and describe it in abstract data type form .**
- **Analyses the time complexity of the following algorithms。**

```
1. i=1;                 2. i=n;             3. x=y=1;
   while (s<n)             do {                 while(x++ * y++<n);
   { i++;s+=i;}              i++;
                             } while (i<n)
```

- **Design an Improve LocateElem's algorithm to look for all the elements matching the relationship。**
- **Design an algorithm to reverse an sequential list ($a_1a_2..a_n$)->($a_na_{n-1}\ldots a_1$)**

Summing Up

- Advantages ：
  - Stores a collection of items contiguously.
    - Stores no relations
    - Access randomly
- Disadvantages ：
  - Need to shift many elements in the array whenever there is an insertion or deletion.
  - Need to allocate a fix amount of memory in advance.

2.3 realization of linear list - linked list

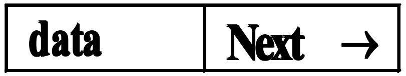
- Singly linked list
- Circular linked list
- Two-way linked list

Linked Lists vs. Sequential List

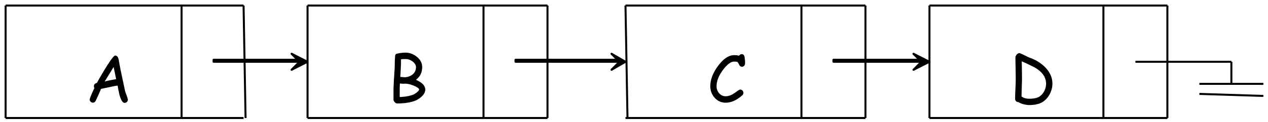
Linked Lists	Sequential List
• Stores a collection of items non-contiguously. • Allows addition or deletion of items in the middle of collection with only a constant amount of data movement. • Allow allocation and deallocation of memory dynamically.	• Stores a collection of items contiguously. • Need to shift many elements in the array whenever there is an insertion or deletion. • Need to allocate a fix amount of memory in advance.

2.3.1 Singly Linked Lists: General Idea

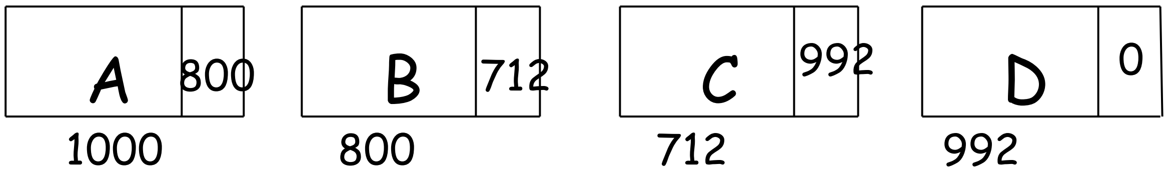
- Each item in the list is stored with an indication of where the next element is.
- Must know where first element is.
- The list will be a chain of objects of type ListNode that contain the data and a reference to the nextListNode in the list.



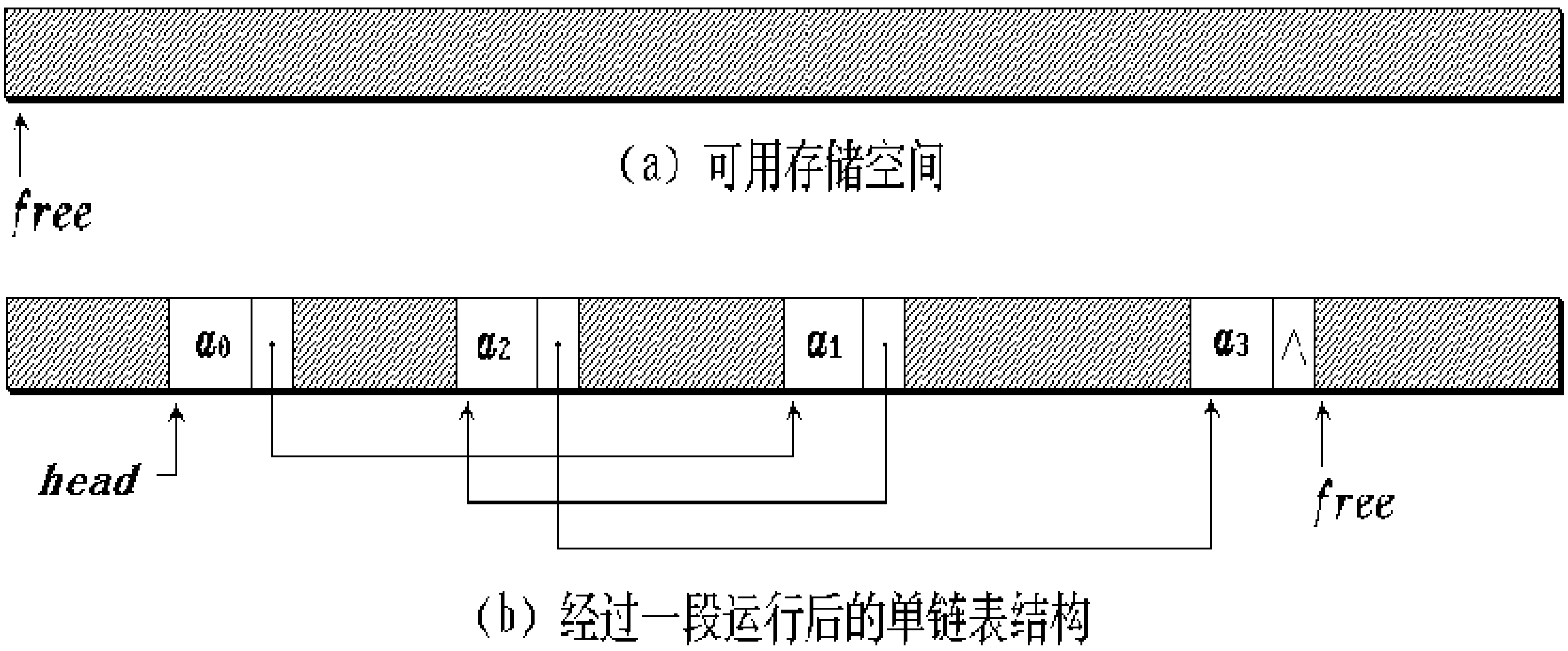
Linked List



List in memory



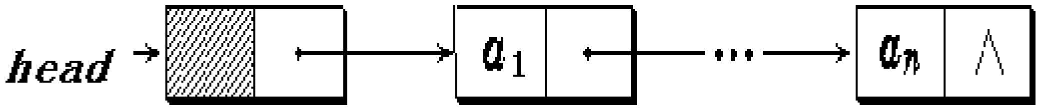
Singly linked list storage m

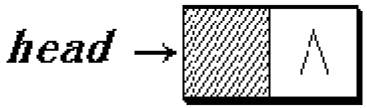


- Header pointer 、Header node 、First node

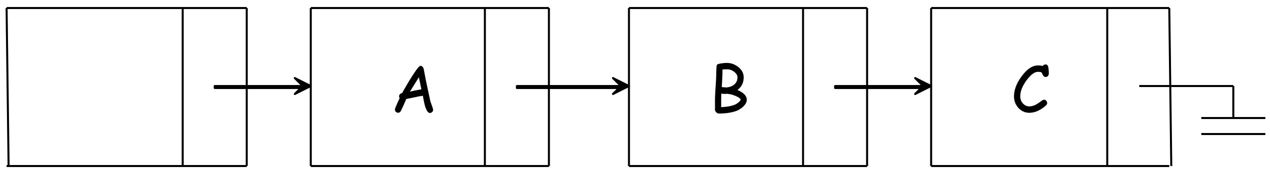
Lists: Header node

- Deletion of first item and insertion of new first item are special cases.
- Can avoid by using header node; contains no date, but serves to ensure that first "real" node in linked has a predecessor.
- Searching routines will skip header.





Header



Singly Linked list storage structure definition

- typedef struct LNode {
-    ElemType data; //'s data field
-   struct Lnode *next; // pointer domain
- } LNode, *LinkList;

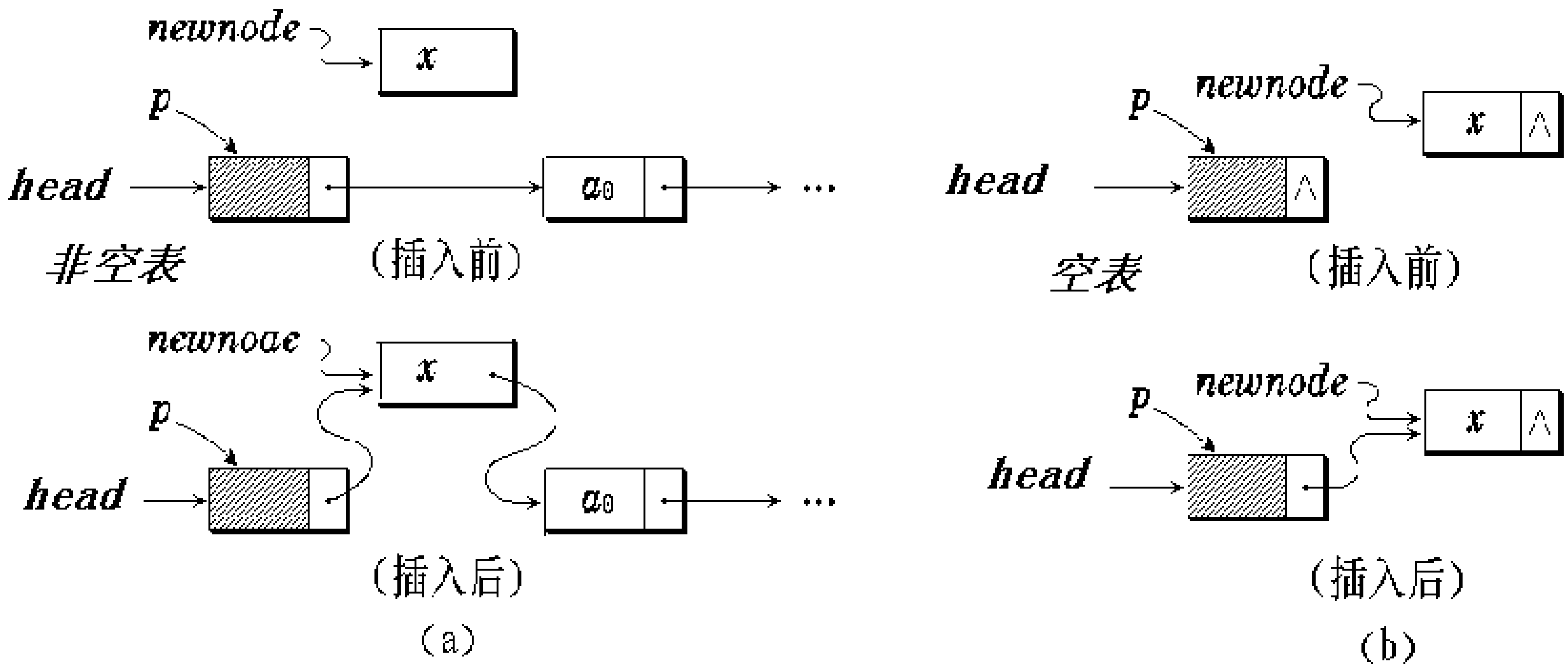
- LNode *L;// is declaration chained list L
- LinkList L;

singly linked list operation realization

1. Create a linked list

- void CreateList_L(LinkList &L, int n) {
- L = NULL; The // establishs the empty list
- for (i = n; i > 0; --i) {
- p = (LinkList) malloc (sizeof (LNode));
- scanf(&p->data); //'s input element value
- p->next = L; L = p; The //
- }
- } //

inserting before first node




```
newnode→next = p→next;
p→next = newnode;
```

There is a header node：

- void CreateList_L(LinkList &L, int n) {
- L = (LinkList) malloc (sizeof (LNode));
- L->next = NULL; // establishs a node
- for (i = n; i > 0; --i) {
-   p = (LinkList) malloc (sizeof (LNode));
-    // Generate the fresh node
-   scanf(&p->data); //'s input element value
-   p->next = L->next; L->next = p; //
- }
- } // CreateList_L
- The algorithm time complexity is : O(Listlength(L))

Insert at the rear ：

```
LinkList  create()
{
 head=NULL;
 r=NULL;
 ch=getchar();
while(ch<>'$')
 { s=malloc(sizeof(LinkList));
   s->data=ch;
   if (head==NULL) head=s; // in the empty list
   else r->next=s;
   r=s;
   ch=getchar();
  }
if (r) r->next=NULL;// is as to the non- empty list
return head;
}
```

Using a header node for rear inserting

- LinkList create() {
- head=(LinkList) malloc (sizeof (LNode));
- head->next = NULL; The // establishs a node
- r=head;
- ch=getchar();
- while(ch<>'$')
- { s=(LinkList) malloc (sizeof (LNode));
-    s->data=ch;
-    r->next=s;
-    r=s;
-    ch=getchar();
-   }
- r->next=NULL;// is as to the non- empty list
- return head;
- }

2.Searching：

A. According to the sequence searching：

- GetElem (L) :
- Status GetElem_L(L, int pos, ElemType &e) {
- // Initialization， The P points to first node， The J act as the counter
- p = L ->next; ; j = 1;//
- while (p && j<pos) {
- // With the pointer look for until P points to Pos_{th} element or p is empty
- p = p->next; ++j;
- }
- if (!p || j>pos) return ERROR;
- e = p->data; // Then gets Pos_{th} element
- return OK;
- } // GetElem_L

` Whether including a header node ?
` If not， how to implement it

- GetElem (L) :
- Manipulating essentially : traversing the list
- Status GetElem_L(L, int pos, ElemType &e) {
- p = L ； j = 1;//
- while (p && j<pos) {
- // until P points to Pos element or p is empty
- p = p->next; ++j;
- }
- if (!p||j>pos) return 0;
- e lse return j;
- } // GetElem_L

`The empty list situation ought to be considered earlier

Search on content :

```
LinkList  Find (LinkList  L, ElemType value )
{
//
      LinkList  p = L→next;      //first node
      while ( p != NULL && p→data != value )
       p = p→next;
      return p;
      // P is living , when the seeking is
succeeful
      // P is null , when the seeking is
failure or a emptying list
}
```

- Whether including a header node ?
- If not， how to implement it

3. Insertion

- insertion alternation
- First kind of situation： inserting at the front
- newnode→next = head ;
- head =newnode;

（Before ）

– **Second kind of situation： inserting in the middle**

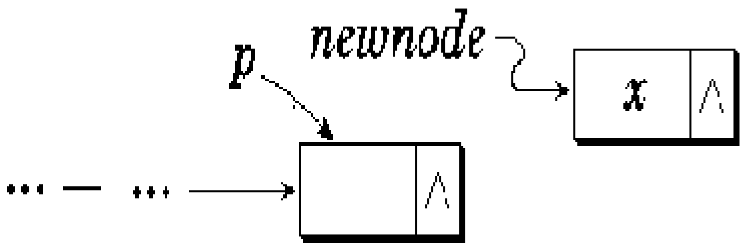
newnode→next = p→next;

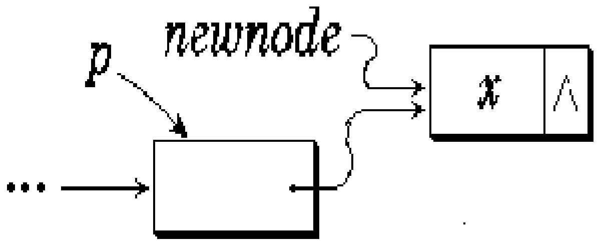
p→next = newnode;

– **Third kind of situation： inserting at the chained list end**

newnode→next = p→next;

p→next = newnode;





Insertion realization :

- Insert behind a node
- Inset before a node
- Insert P at the front
- Using the header node

example1 :

- Status ListInsert_L(L, int pos, ElemType e) {
- //The Pos element of single chained list L afterwards inserting element E
- p = L; j = 0;
- while (p && j < pos)
- { p = p->next; ++j; } // looks for Pos node
- if (!p || j > pos-1) return ERROR;
- s = (LinkList) malloc (sizeof (LNode));
- s->data = e; s->next = p->next;
- p->next = s;
- return OK;
- } // LinstInsert

单链表的插入

Insert before Pos node :

- How to find its predecessor ：
- 1 Find Pos element， Keep the predecessor Q

```
while (p && j < pos)
{ q=p;  p = p->next; ++j; }
```

Do the related operation according the value of Q

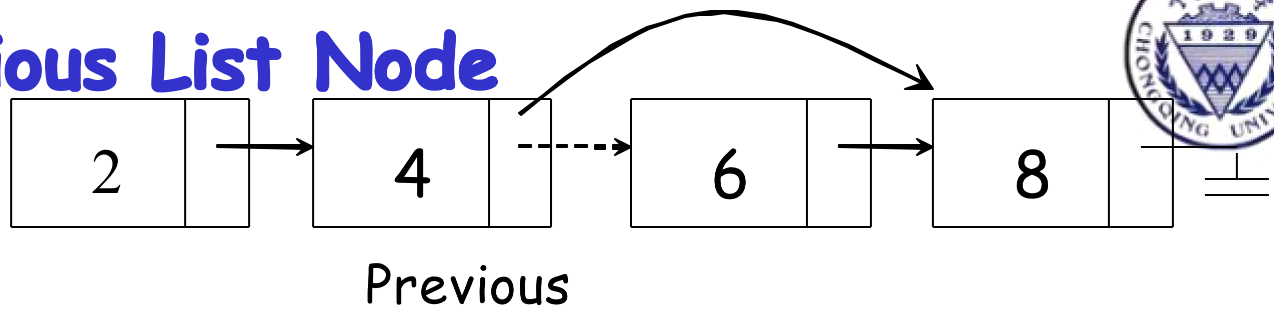
- 2 using a rear insertion ， Interchange again
- 3： Find Pos-1's element earlier
- 4： Find P earlier， and then its predecessor Q

```
q=head;//
while(q->next!=p)
  q=q->next;
```

Caution：had better use a header node 。

```
• InsertBefore  (head,p,x)
• {
•      LinkList s, q;
•      s=malloc(sizeof(LinkList));
•      s->data=x;
•      q=head;//'s head node
•      while(q->next!=p)
•        q=q->next;
•      s->next=p;
•      q->next=s;
• }
```

Find Previous List Node

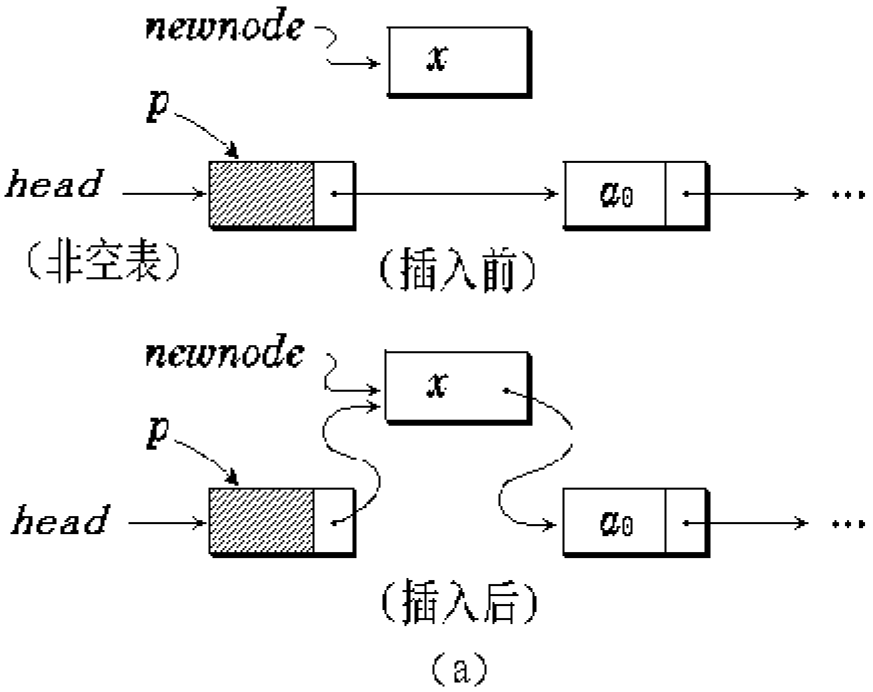


/* If X is not found, then Next field of returned value is NULL */

/* Assumes a header, why ? */

```
        Position  FindPrevious( ElementType X, List L )
        {
            Position P;

/* 1*/      P = L;
/* 2*/      while( P->Next != NULL && P->Next->data != X )
/* 3*/          P = P->Next;

/* 4*/      return P;
        }
```

Insert at the front

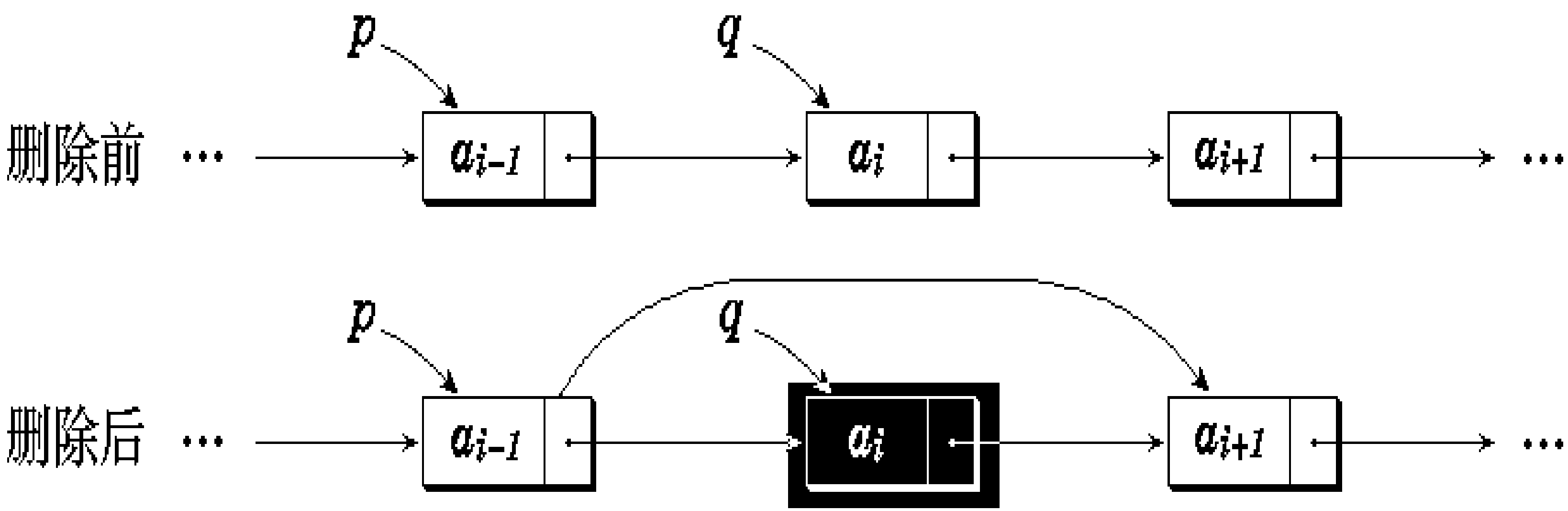


newnode→next = p→next;

p→next = newnode;

4. Delete operations

- Delete the successor of current node
- Delete current node



Delete the node at i_{th}

Delete the successor :

```
Status ListDeleteAfter(LinkList L, LinkList p) {
     if (!p->next) error("no successor!");
     r=p->next;
     p->next=r->next;
    free(r);
}
```

The algorithm time complexity is : O(1)

Delete current node :

- Find predecessor
- If current node is the first node ?

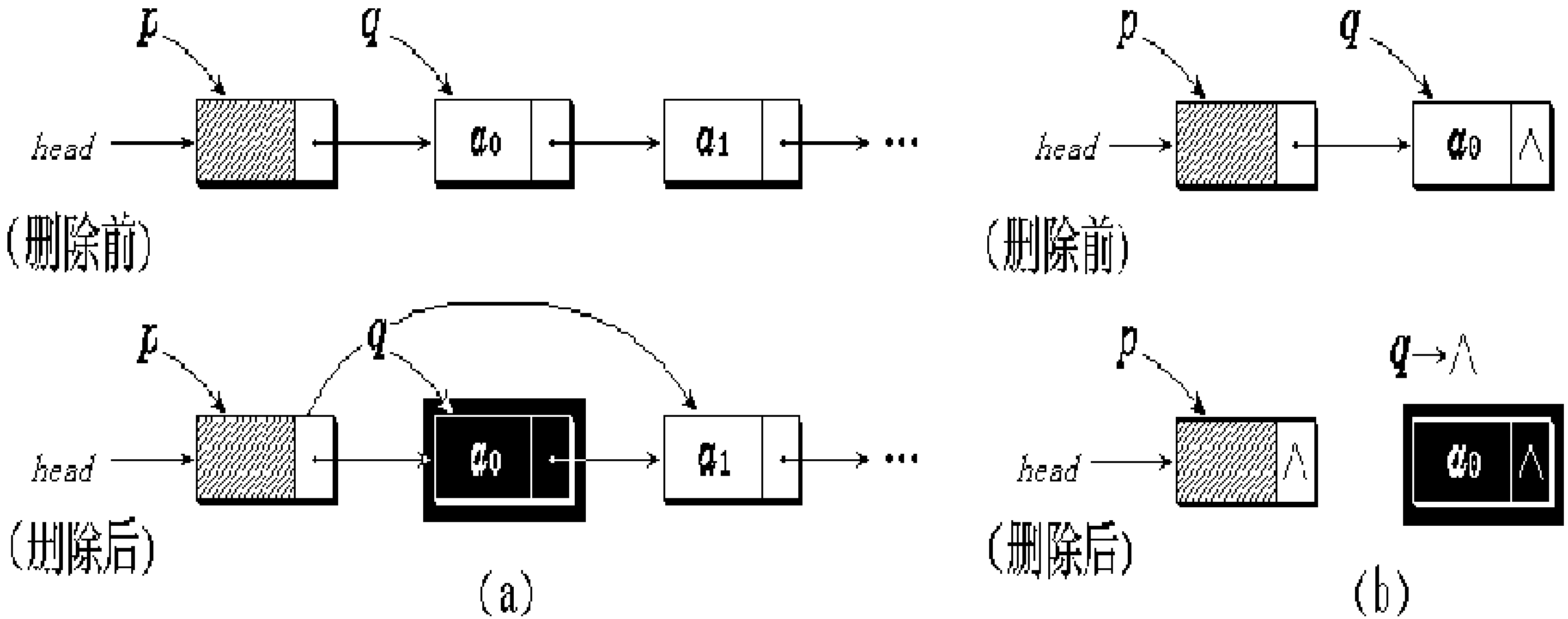
Delete current node algorithm：

```
Status ListDelete (LinkList L, ElemType e) {
    p=q=L;
    if (!P) Error(“The list is empty!”);
    if (p->data==e)
     {  L=p->next; free p; return;}
    while (p && p->data !=e)
      {   q=p; p=p->next;}
    q->next=p->next;
    free(p);
}
```

The algorithm time complexity is：O(ListLength(L))

If using a header node？

Delete first node using a header node




```
q = p→next;
p→next = q→next;
free q;
```

Delete current node （header node ）：

```
Status ListDelete_L(LinkList L, int pos, ElemType &e) {
// delete Pos_th element， returns such value by E
p = L; j = 0;
while (p->next && j < pos-1) {
// Look for Pos node， P point to its predecessor Pos-1's node
p = p->next; ++j;
}
if (!(p->next) || j > pos-1)
return ERROR; The // delete error
q = p->next; p->next = q->next; The // delete
e = q->data; free(q);
return OK;
} // ListDelete_L
```

The algorithm time complexity is : O(ListLength(L))

5.To compute the length：

```
Status ListLength_L(LinkList L) {
//traverse
p = L; j = 0;
while (p->next) {
  ++j; p = p->next;
 }
return j;
} // ListLength_L
```

The algorithm time complexity is : O(ListLength(L))

- If the initial value is a p=L->next ？
  j=0; while (p)
- If not including a header node ？
  P=L; j=0; while (p)

3 、 Application of algorithm

Example 2-1's algorithm time complexity

Control structure ： For cycles Manipulate essentially ： LocateElem(La, e, equal( ))

Acing as when achieving the abstract data form linear list with the order map :

O(ListLength(La)×ListLength(Lb))

When in order acing as when the abstract data form linear list is achieved in the link style map :

O(ListLength(La)×ListLength(Lb))

Example 2-2's algorithm time complexity

Control structure ： While's cycle is manipulated essentially ： GetElem(L, i, e)

Acing as when achieving the abstract data form linear list with the order map :

O(ListLength(L)2)

When in order acing as when the abstract data form linear list is achieved in the link style map :

(ListLength(L) 2)

example 2-3's algorithm time complexity Control structure ： Three coordinations Whiles cycle Manipulate essentially ： ListInsert(Lc, ++k, e)

Acing as when achieving the abstract data form linear list with the order map :

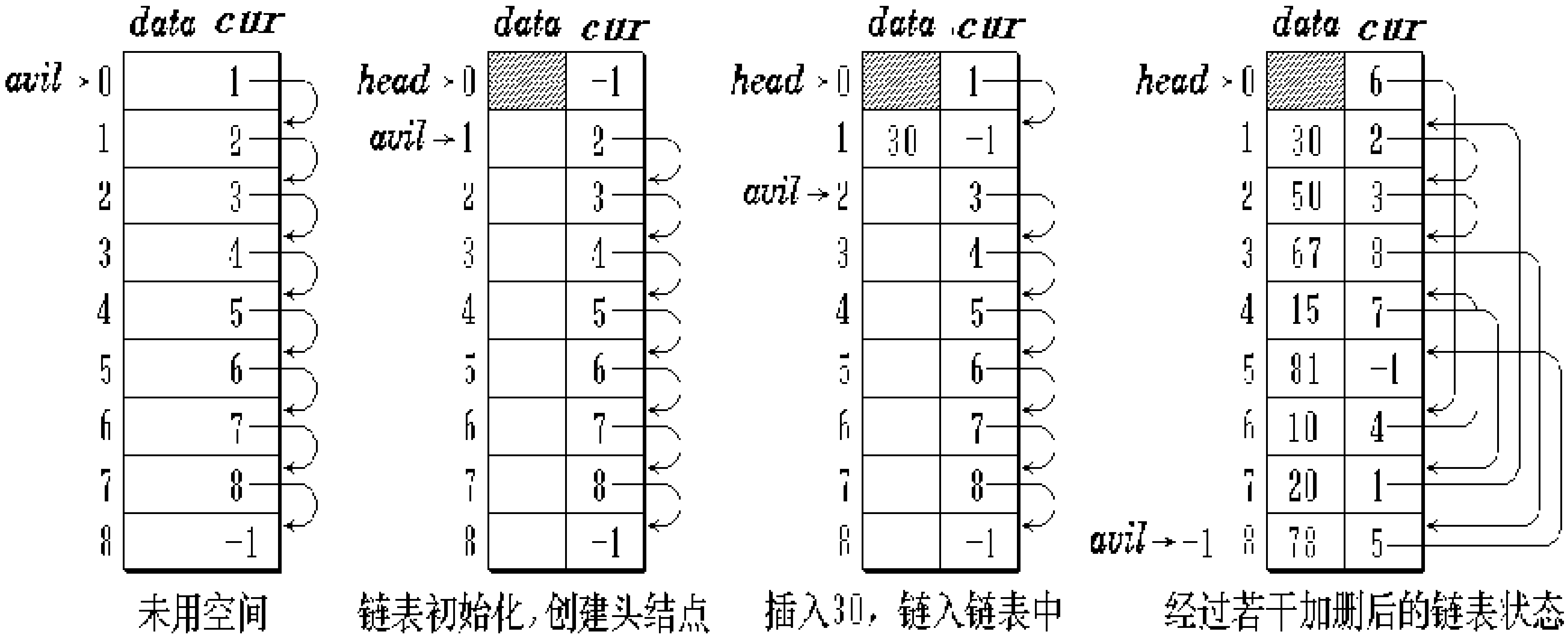
O(ListLength(La)+ListLength(Lb))

When in order acing as when the abstract data form linear list is achieved in the link style map :

O((ListLength(La)+ListLength(Lb))， Yet room complexity difference 。

2.3.2 static chained list

Use array defining , The dedicated space size is unchangeable in the calculation process



Distribute node J : j = avil; avil = A[avil].cur;//'s tail pointer page-down

free Node I : A[i].cur = avil; avil = i;//'s tail is leaved out

(Notes : Avil act as present in the form the second

Linked List Variants

A (dummy) **head node** is used so that *every node has a predecessor*
⇨ eliminates special cases for inserting and deleting.

The data part of the head node might be used to store some information about the list, e.g., the number of values in the list.

A (dummy) **trailer node** can be used so that *every node has a successor*

If data portion of element is large, two or more lists can share the same trailer node

2.3.3 Circularly Linked Lists

instead of the last node containing a NULL pointer, it contains apointer to the first node

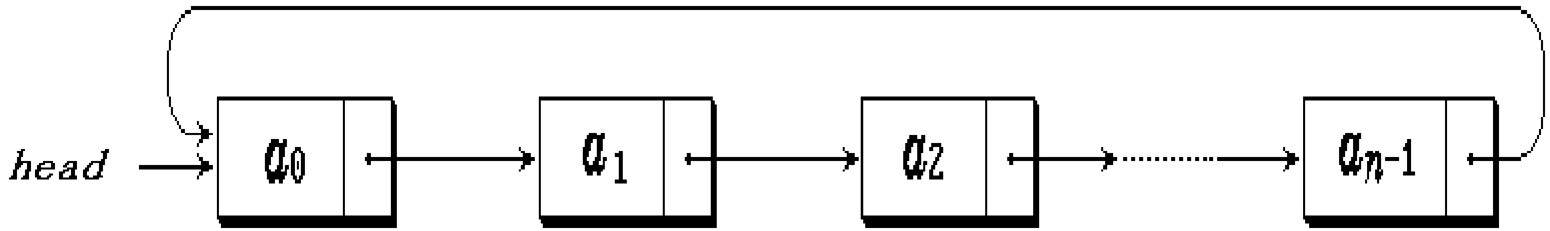
For such lists, one can use a single pointer to the last node in the list, because then one has direct access to it and "almost-direct" access to the first node.

Each node in a circular linked list has apredecessor(and a successor), provided that the list is nonempty.

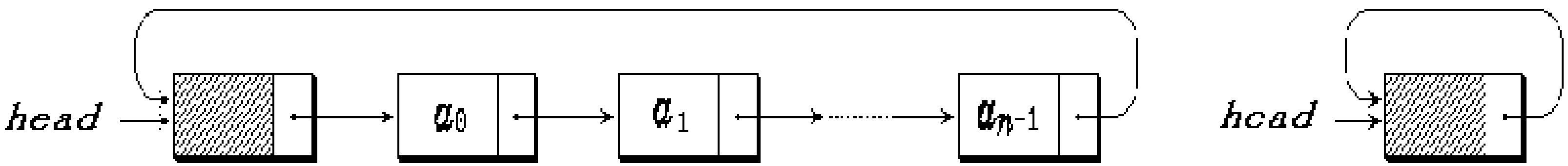
⇨ insertion and deletion do not require special consideration of the first node.

Using a trailer node
Treat each node as the first

- **Example of circular linked list**



- **Girdle form head node**



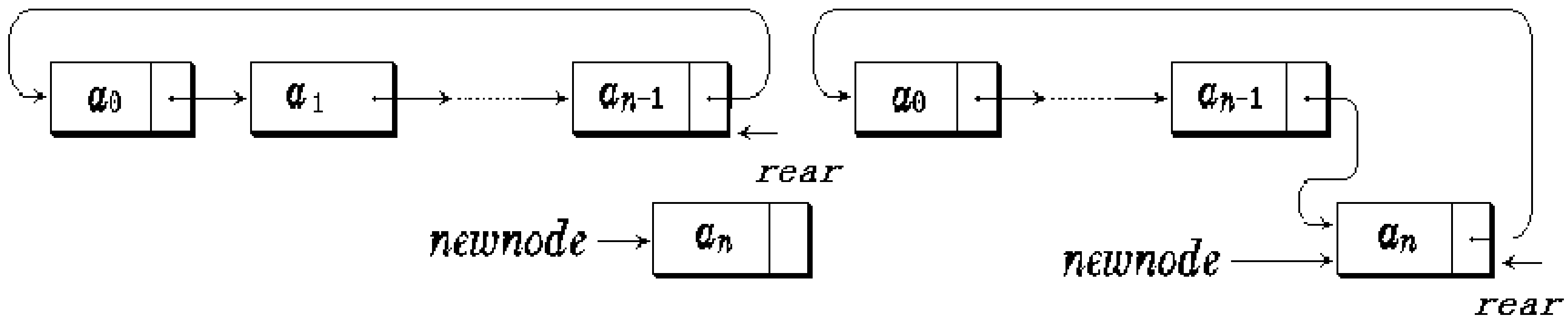
Circularly Linked Lists

Traversal must be modified: don't an infinite loop looking for end of list as signalled by a null pointer.

Like other methods, deletion must also be slightly modified.

Deleting the last node is signalled when the node deleted points to itself.

```
if (first == 0) // list is empty
    // Signal that the list is empty
else
{
    ptr = predptr->next; // hold node for deletion

    if (ptr == predptr) // one-node list
        first = 0;
    else // list with 2 or more nodes
        predptr->next = ptr->next;

    delete ptr;
}
```



Left insertion :

```
void left_insert_CL(rear,x)
 {
     p=malloc(sizeof(LNode));
    p->data=x;
    if (rear==NULL) { p->next=p; rear
=p;}
    else { p->next= rear ->next; rear -
>next=p;}
 }
```

If using an header node ?

(right insertion of circular linked list) :

```
void right_insert_CL(rear,x)
  {
     p=malloc(sizeof(LNode));
     p->data=x;
     if (rear==NULL)
       { p->next=p; rear =p;}
     else
       { p->next= rear ->next; rear
->next=p;rear=p;}
 }
```

If using an header node ?

left deletion:

```
void left_dele_CL(rear)
  {
      if ( rear!=NULL)
     {
     p=rear->next;
     if (p==rear)
rear=NULL;//only one node
     else rear->next=p->next;
     free p;
  }
}
```

Example 1： LA+LB==>LC

```
• Linklist  connect(ra ,rb)
•   Linklist  ra,  rb;
•  {  Linklist  p;
•      p=ra->next;
•      ra->next=rb->next->next;
•      free(rb->next); rb->next=p;
•       return   rb;
•      }
```

example2： The length is more than 1， there is not a herder node、 Head pointer， The P points to some nodes in the list， Attempt to delete that node's predecessor。

```
LinkList DelCirList（p）
 {   q=p;The predecessor of // searching P
     while (q->next!=p) q=q->next;
     r=q ;The predecessor of // searching Q
     while (r->next!=q) r=r->next;
     r->next=p;
     free（ q);
     return p;
}
```

If first While cycles using
(q->next->next! = P)

2.3.4 polynomial (Polynomial) application

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

Expressing the polynomial

- **Express the linear list :**

P = (p0, p1, …， pn)

- **It is also unsuitable to express the form like S (X) = 1 + $3x^{10000}$**
- **Writing factor and index number**

（（p1, e1）, (p2, e2), ---, (pm,em) ）

- **How about the defects**

The link expressing

- Every one node addd data member Nexts during the polynomial chained list being living is expressed , As the link pointer 。

data = *Term*

coef	exp	next

- Strong point is :

  The number of item of polynomial may rise dynamicly 。

  It is convenient to insert,delete the element 。

Polynomial (Polynomial) type definition

ADT Polynomial {
Data object： D＝{ ai | ai ∈TermSet, i=1,2,...,m, m≥0 }

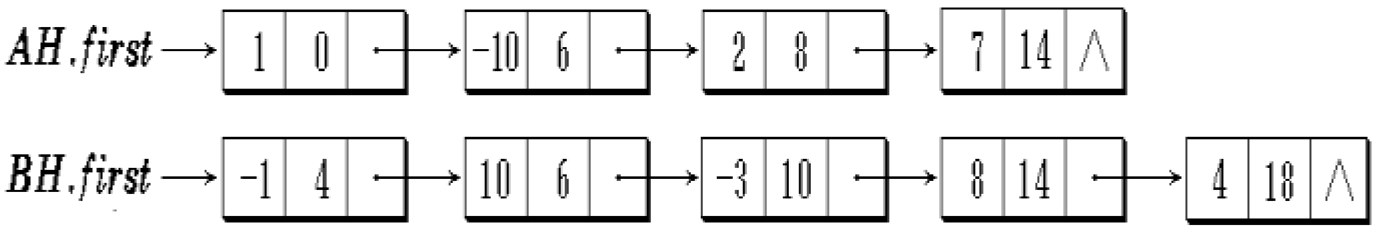
Data relationship ： R1＝{ <ai-1 ,ai >|ai-1 ,ai ∈The index number value of Ai-1 ＜ The index number value of Ai， i=2,...,n }
Basic opertions：
CreatPolyn (&P)
DestroyPolyn (&P)
PrintPolyn (&P)
AddPolyn (…)
SubtractPolyn (…)
MultiplyPolyn (…)
PolynLength (P)
} ADT Polynomial

Polynomial (Polynomial) node definition

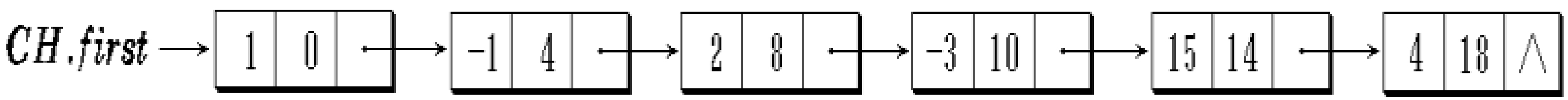
```
typedef struct  {
    int coef;
    int exp;
    PolyLink next;
} *PolyLink;
```

Polynomial adding to of chained list

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$
$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式

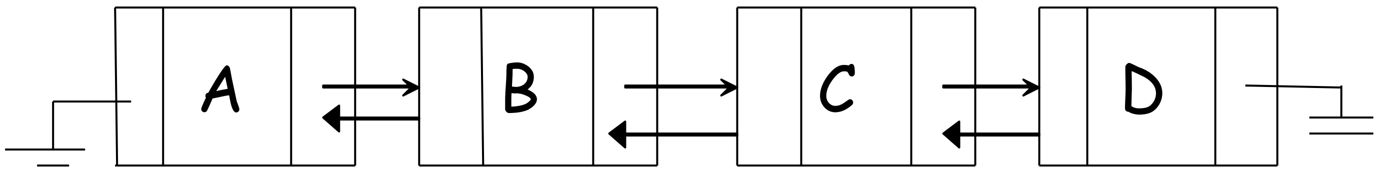
```
 Polynomial AddPolyn ( const Polynomial
        & pa, const Polynomial &pb ) {
     ha=GetHead(pa); hb=GetHead(pb);
    qa=NextPos(ha); qb=NextPos(hb);
while ( !Empty(pa) && !Empty(pb) )
 { a=GetCurElem(qa); b=GetCurElem(qb);
    switch ( *compare ( a, b ) ) {
    case '=' ://Index number is equal to

        sum = a.coef + b.coef;
        If ( Sum = 0.0 ) {// leaves out Pa's
current node
         DelFirst(ha,qa); FreeNode(qa);
        DelFirst(hb,qb); FreeNode(qb);
qb=NextPos(pb,hb);
        qa=NextPos(pa,ha);
        }
        Factor value among Else {// alteration Pa
            setCurElem(qa,sum);ha=qa;
        }
```

```
                break;
            case '<' :

           ha= qa;
qa=Nextpos(pa,qa);
                 break;
            case '>' :



DelFirst(hb,qb);InsFirst(ha,qb);
                qb=Nexpos(pb,hb);
    }//switch
  }//while
    if ( !Empty(pb) )
Append(pa,qb);
    else   FreeNode(hb);
```

Doubly Linked List

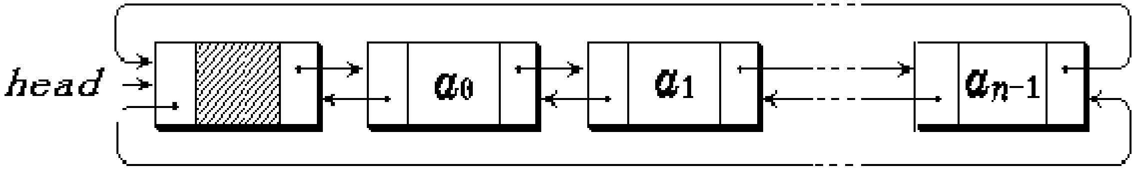
- Add an extra pointer to the previous node.
- Increase the memory used for every node.
- More pointers to adjust for insertion and deletion.
- Eliminate the use of previous node for deletion.

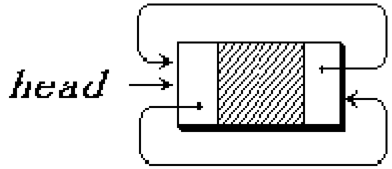
Doubly linked list



storage organization definition

- //
- typedef struct DuLNode {
-     ElemType data; //'s data field
-     struct DuLNode *prior;
-     struct DuLNode *next;
- } DuLNode, *DuLinkList;

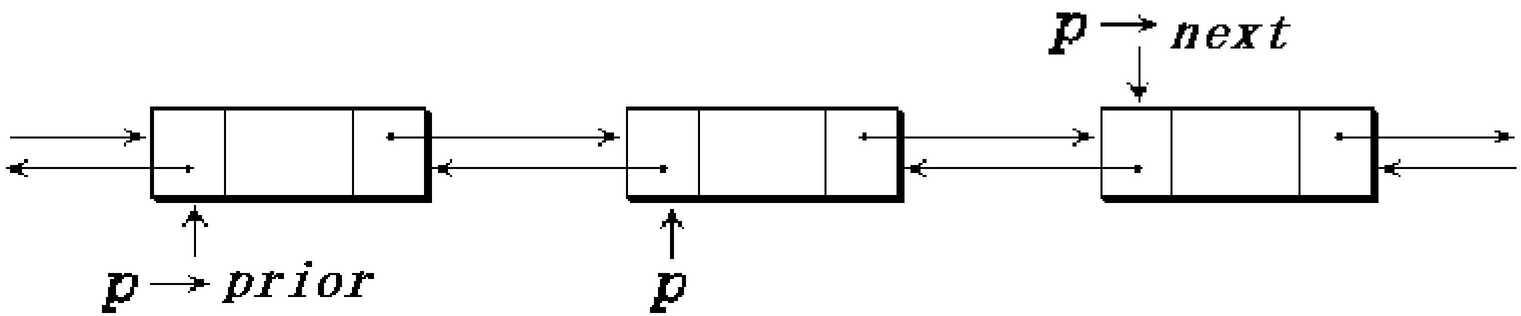




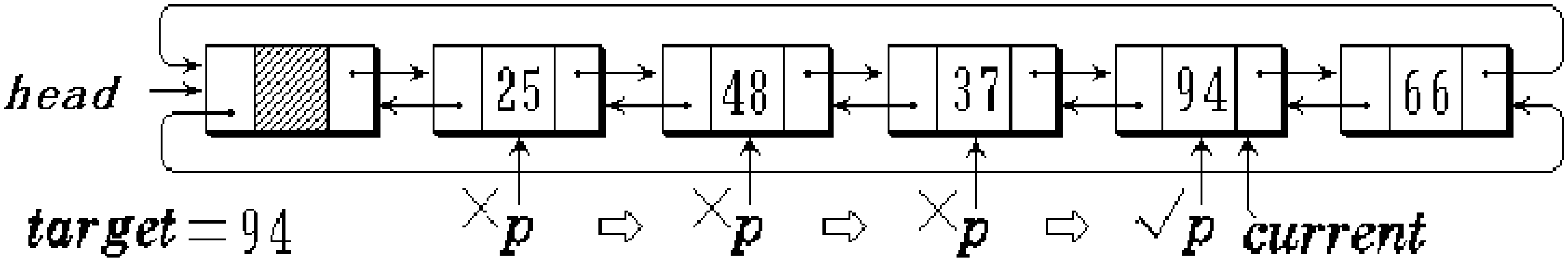
Non- empty list　　　　　　Empty list

-

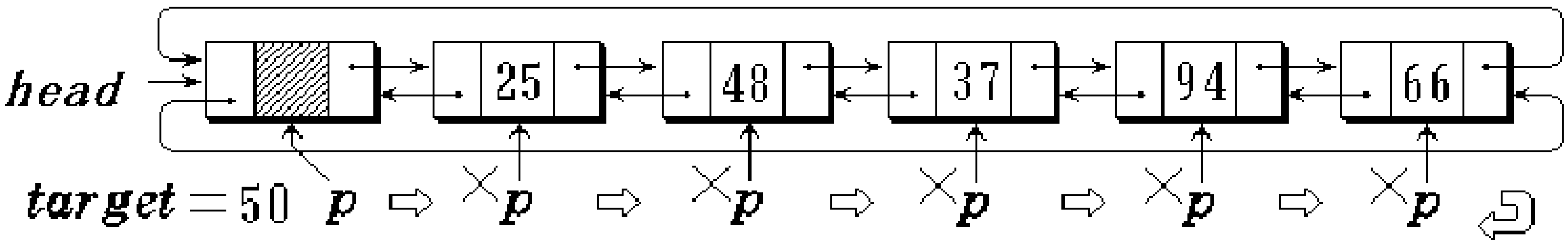
p == p→prior→next == p→next→prior



Two-way circular linked list seeking algorithm



Seeking is succeeded

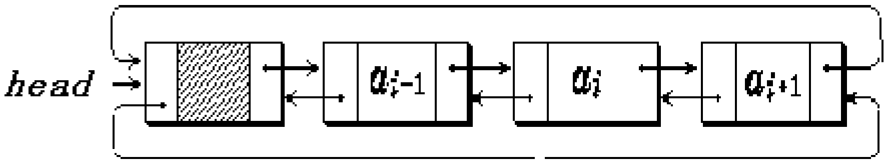


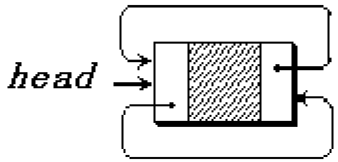
The seeking is not succeeded

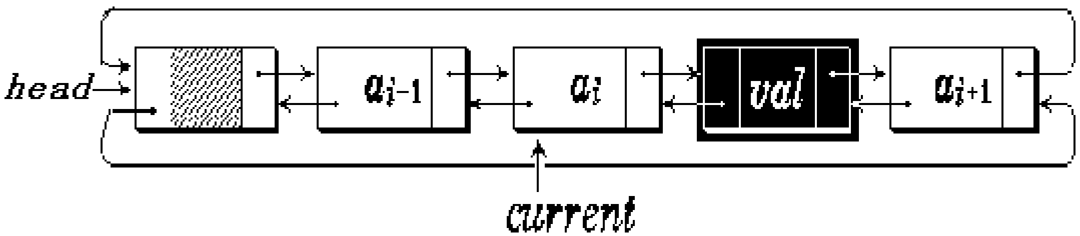
```
Int  Find (DuLinkList DL, const Type &
target ) {
//The seeking is successfully returned 1 ,
If not return 0 。
     p = DL→next;
     while ( p != DL && p→data != target )
        p = p→next;   //Abideing by the link
lookes for
     if ( p != DL ) return 1;// finds
     return 0; The // gos back up the form
head , Not find

}
```

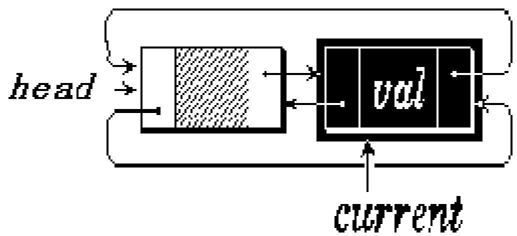
Insert

```
p→prior = current;
p→next =current→next;
p→next→prior = p;        //current->next->prior=p;
current→next = p;
```





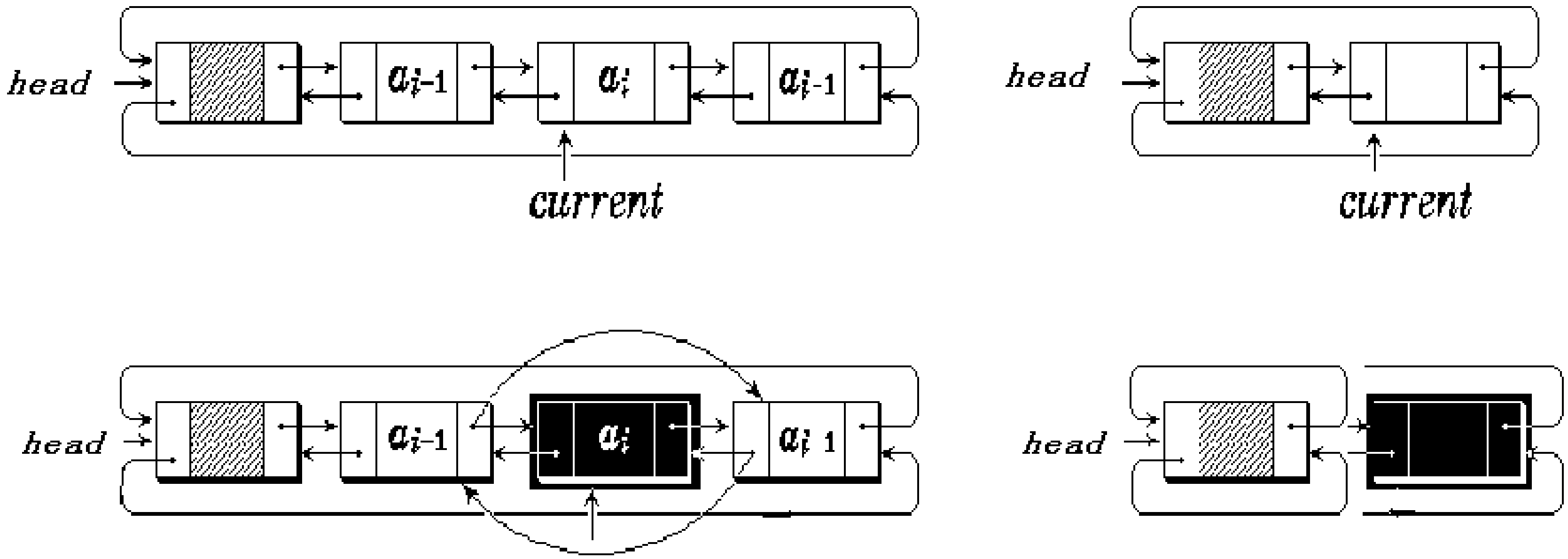




```
current=Find(DL,X);
if (current==NULL) Error;
p=(DuLinkList)malloc(sizeof(DLNode
));
p->data= Y ;
p→prior = current;
p→next =current→next;
p→next→prior = p;     //current-
>next->prior=p;
current→next = p;
}
```

Delete

current→next→prior = current→prior;   current→ prior→next = current→next;



Assignment 2

2.1 Attempt to explain an header pointer 、Head node 、First element node 。

2.2 When choose the Sequential list 、When choose the linked list .Give some examples to show this. ?

2.3 Reverse the single linked list 。

2.4 Write an algorithm to insert a X into an ordered list to maintain the ordered list 。

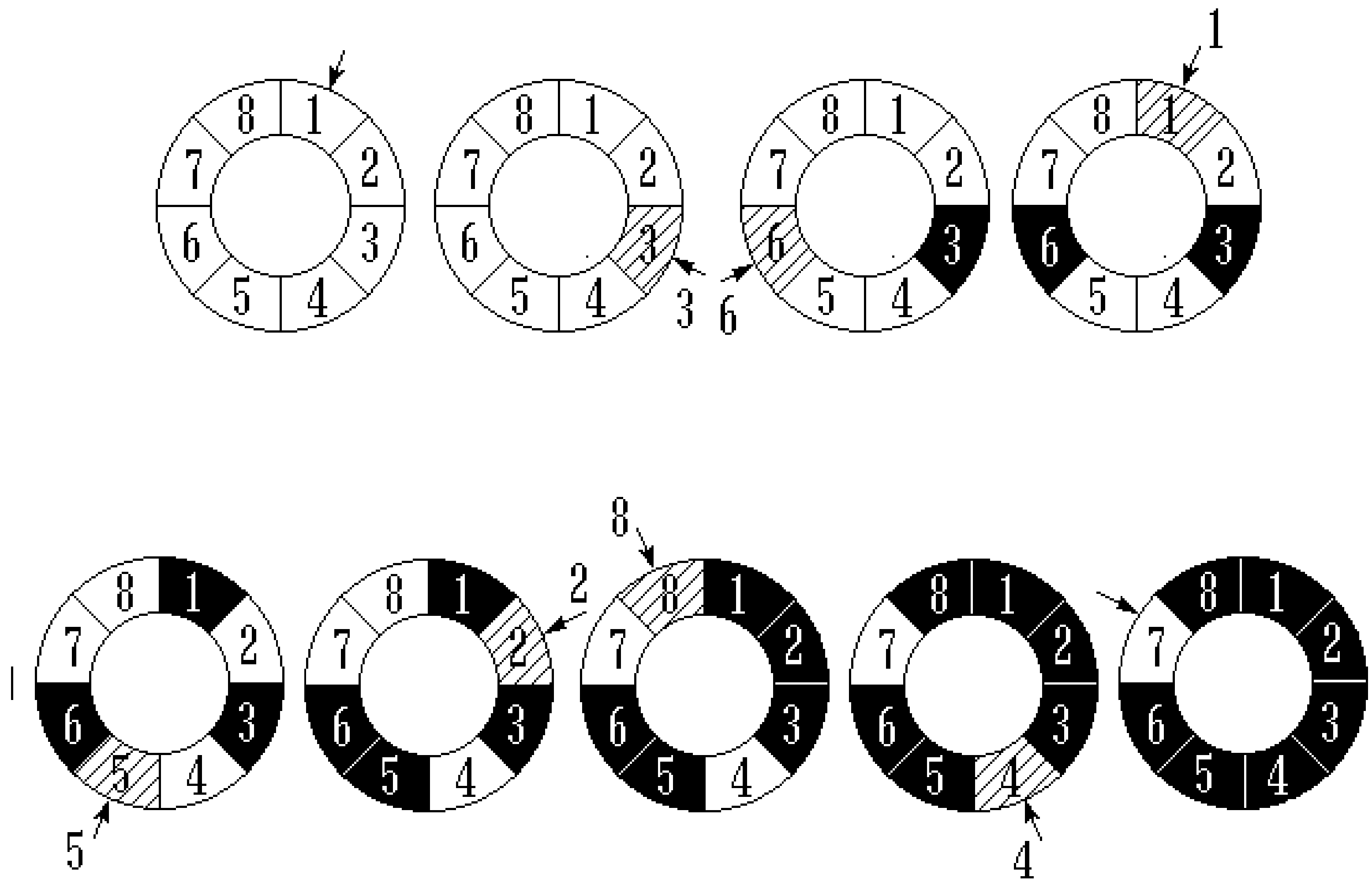
2.5 Write an algorithm to delete the element at the right side of the circular linked list 。

2.6 Write an algorithm to delete the first node that has the value X in a doubly linked list。

Experiment

2.1 implement the linked list type 。
2.2 Josephues problem

- **For example N = 3   m = 8**



Requirements ：

1 、Prepare
2 、discipline
3 、Laboratory report
Write laboratory report ，Consist of the main idea of algorithm 、Main data structure 、The algorithm achieves essentially 、Debug process 、Conclusion and what one has learned 。

-End-