

# **Chương 3:** **Danh sách liên kết đơn** **(DSLK đơn)**

# Mục tiêu

- Tiếp cận các cấu trúc dữ liệu động :
  - Danh sách liên kết đơn,
  - Ngăn xếp - Hàng đợi,
  - Danh sách liên kết đơn vòng,
  - Danh sách liên kết kép.

## Nội dung :

- Cấu trúc dữ liệu danh sách liên kết đơn
- Các cấu trúc DSLK đơn đặc biệt :
  - Stack (Ngăn xếp)
  - Queue (Hàng đợi)
- Cấu trúc dữ liệu danh sách liên kết đơn vòng
- Cấu trúc dữ liệu danh sách liên kết kép



# Cấu trúc dữ liệu danh sách liên kết đơn

# Danh sách liên kết đơn

Tổ chức danh sách  
đơn theo cách  
cấp phát liên kết

Các thao tác  
cơ bản trên  
danh sách đơn

Tạo  
phần tử

Chèn  
phần tử

Hủy  
phần tử

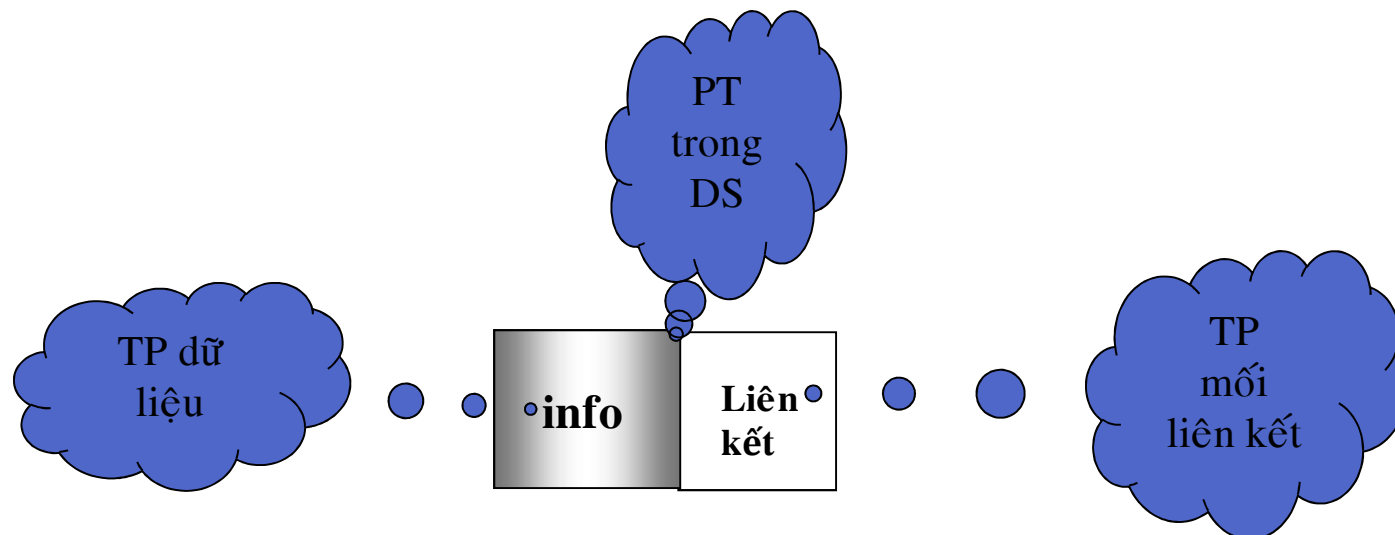
Tìm kiếm  
phần tử

Sắp xếp

...

## Tổ chức danh sách đơn theo cách cấp phát liên kết

- Mỗi phần tử (nút) của danh sách đơn là một cấu trúc chứa 2 thông tin :
  - Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử .
  - Thành phần mối liên kết: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc có giá trị NULL nếu là phần tử cuối danh sách.



■ Cài đặt: Kiểu của một phần tử (một nút) trong danh sách

- Kiểu thành phần dữ liệu của nút  
**typedef <KDL> data;**
- Kiểu phần tử ( nút) trong danh sách :

**struct tagNode**

{

**data info;** //Thành phần dữ liệu

**tagNode\* pNext;** //Thành phần mỗi liên kết (tự trỏ)

};

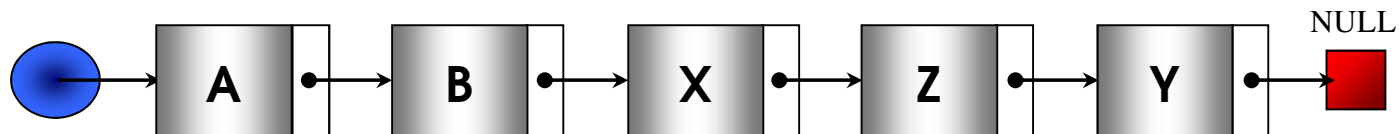
*//Đổi lại tên kiểu phần tử trong DS*

**typedef tagNode NODE;**



- Một phần tử trong danh sách đơn là một biến động (sẽ được cấp phát bộ nhớ khi cần).
- Danh sách đơn là *sự liên kết* các biến động này với nhau, thành phần mỗi liên kết của nút trước móc nối phần tử kế sau (con trỏ chứa địa chỉ của nút kế sau).

Do vậy danh sách đạt được sự linh động khi thay đổi số lượng các phần tử.



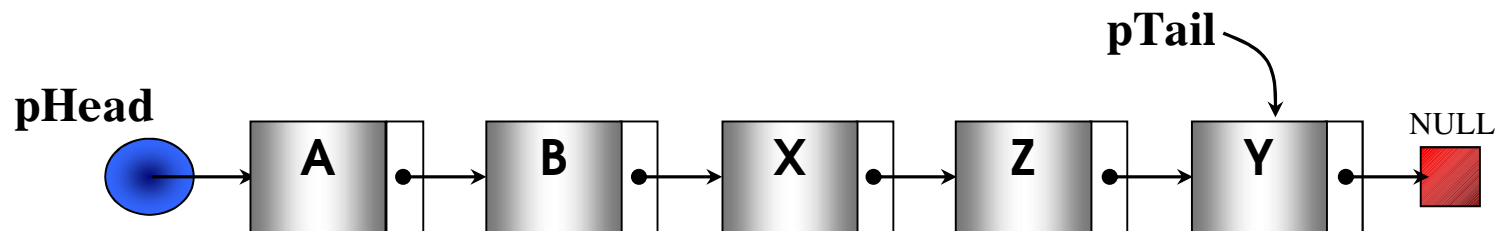


- Để quản lý một DSLK đơn chỉ cần biết địa chỉ phần tử đầu DSLK.
- Con trỏ pHead sẽ được dùng để lưu trữ địa chỉ phần tử đầu DSLK. Ta có khai báo:

**NODE\*** pHead;

- Để tiện lợi, ta sử dụng thêm một con trỏ pTail giữ địa chỉ phần tử cuối DSLK. Khai báo pTail như sau :

**NODE\*** pTail;



- Theo cách tổ chức này, ta quản lý danh sách liên kết đơn bằng 2 con trỏ : Con trỏ pHead chứa địa chỉ nút đầu, con trỏ pTail chứa địa chỉ nút cuối DSLK.

**Cài đặt:** Kiểu danh sách liên kết đơn (DSLK liên kết đơn)  
(mỗi nút trong danh sách liên kết có kiểu NODE)

- Giả sử đã có các định nghĩa:

```
typedef <KDL> data;  
struct tagNode  
{  
    data                info;  
    tagNode*           pNext;  
};
```

*// kiểu của một phần tử trong danh sách*

```
typedef tagNode NODE;
```

- **Cài đặt kiểu Danh sách liên kết: LIST**

```
struct LIST  
{  
    NODE* pHead;    //Con trỏ lưu địa chỉ phần tử đầu DSLK  
    NODE* pTail;    //Con trỏ lưu địa chỉ phần tử cuối DSLK  
};
```

- Ví dụ : Định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên:

```
//Kiểu thành phần dữ liệu data: SV  
struct SV //data  
{ char      ten[30];  
  int      maSV;  
};  
typedef SV data;  
//Kiểu phần tử trong DS: SinhvienNode  
struct SinhvienNode  
{ data      info;  
  SinhvienNode* pNext;  
};  
//Đổi lại tên kiểu phần tử trong DS: SVNNode  
typedef SinhvienNode SVNNode;
```

- 
- ❑ Cài đặt kiểu Danh sách liên kết: LIST

```
struct LIST
{
    SVNNode * pHead;
    SVNNode * pTail;
};
```



## Các thao tác cơ bản trên danh sách đơn

- Tạo một phần tử cho danh sách với thông tin  $x$ .
- Khởi tạo danh sách rỗng
- Kiểm tra danh sách rỗng
- Duyệt danh sách
- Tìm một phần tử trong danh sách đơn
- Chèn một phần tử vào danh sách đơn
- Hủy một phần tử khỏi danh sách
- Sắp xếp danh sách

- Giả sử đã có định nghĩa cấu trúc dữ liệu DSLK đơn LIST như sau :

```
typedef int data; // GD data là liệu số nguyên : int
```

```
struct tagNode
```

```
{
```

```
    data                info;
```

```
    tagNode*           pNext;
```

```
};    // kiểu của một phần tử trong danh sách
```



```
typedef tagNode NODE;
```

```
// kiểu danh sách liên kết
```

```
struct LIST
```

```
{
```

```
    NODE* pHead;
```

```
    NODE* pTail;
```

```
};
```

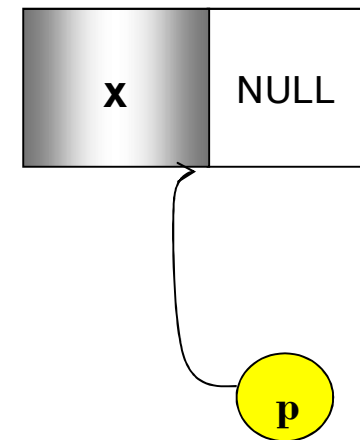
## Tạo một phần tử (nút) cho danh sách với thông tin x.

### ■ Hàm **GetNode (x)** :

- Tạo ra một phần tử (nút) của DSLK với thành phần dữ liệu là x
- Hàm trả về con trỏ lưu trữ địa chỉ của pt vừa tạo (nếu tạo thành công)

### **NODE\* GetNode(data x)**

```
{  
    NODE *p;  
    // Cấp phát vùng nhớ cho phần tử  
    p = new NODE;  
    if ( p==NULL)  
    {  
        cout<<"Loi cap phat";  
        return NULL; //exit(1);  
    }  
    p ->info = x; // Gán thông tin cho phần tử p  
    p->pNext = NULL;  
    return p;  
}
```



//Có thể viết gọn hơn như sau :

**NODE\* GetNode(data x)**

```
{  
    NODE *p;  
    p = new NODE;  
    if (p != NULL)  
    {  
        p->info = x;  
        p->pNext = NULL;  
    }  
    return p;  
}
```

■ Sử dụng hàm GetNode:

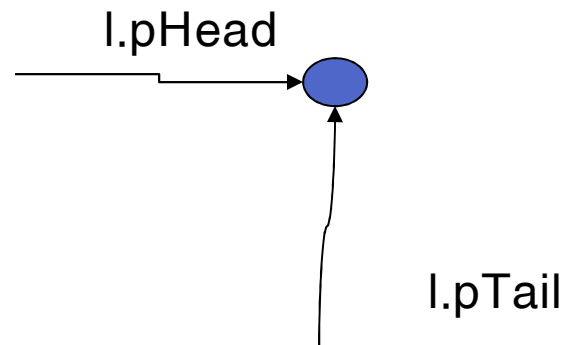
Gán giá trị của hàm (là con trỏ) cho 1 biến con trỏ kiểu NODE , Phần tử này đặt tên là new\_ele, giữ địa chỉ phần tử đã tạo :

NODE \***new\_ele** = **GetNode(x)**;



# Khởi tạo danh sách rỗng

- Input: không có/ dslk (LIST 1)
- Output: dslk rỗng



- 
- Khởi tạo danh sách rỗng l:

```
void CreateList(LIST &l)
{
    l.pHead = l.pTail = NULL;
}
```

- Kiểm tra danh sách l rỗng: **IsEmpty(l)**  
**IsEmpty(l) = 1; nếu l rỗng**  
**= 0; ngược lại**

```
int IsEmpty(LIST l)
{
    if (l.pHead == NULL) // DS rỗng
        return 1;
    return 0;
}
```

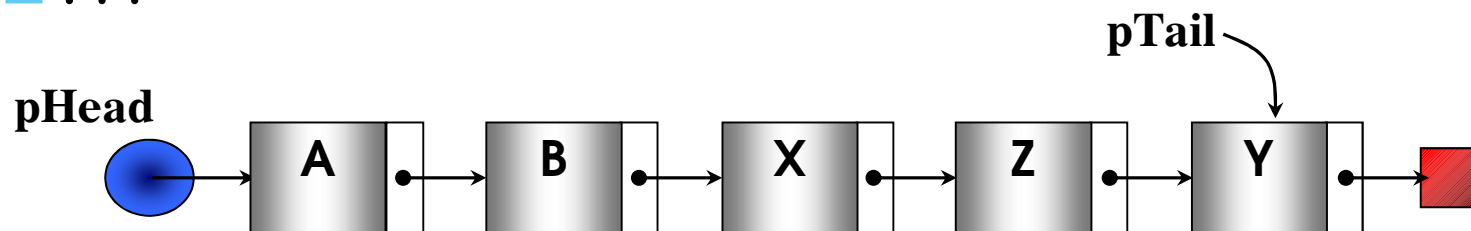
- Ghi chú : Để kiểm tra DSLK l có khác rỗng hay không, nhiều khi không cần viết hàm **IsEmpty**, mà ta chỉ kiểm tra trực tiếp bằng câu lệnh :

```
if (l.pHead != NULL) //DS khác rỗng
```

## Duyệt danh sách

- Duyệt danh sách là thao tác dịch chuyển từ nút này (thường là nút đầu) đến nút khác (thường là nút cuối) theo hướng móc nối, thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc lấy thông tin tổng hợp từ các phần tử của danh sách như:

- Tìm kiếm các phần tử thoả điều kiện,
- Xuất dữ liệu các phần tử trong DS ra màn hình
- Đếm các phần tử của danh sách,
- Hủy toàn bộ danh sách (và giải phóng bộ nhớ)
- ...



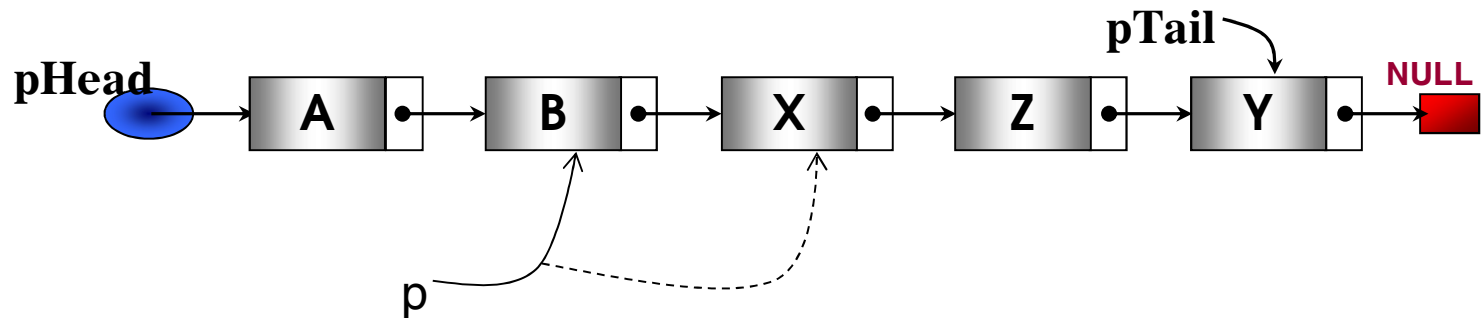
- Mô tả thuật toán duyệt :

- Bước 1:  $p = \text{pHead}$ ; // Cho  $p$  trở đến phần tử đầu danh sách
- Bước 2: Trong khi (Danh sách chưa hết) thực hiện
  - B2.1 : Xử lý phần tử  $p$ ;
  - B2.2 :  $p = p \rightarrow \text{pNext}$ ; // Cho  $p$  trở tới phần tử kế

- Cài đặt như sau:

**void ProcessList (LIST l)**

```
{  
    NODE *p;  
    p = l.pHead;  
    while (p != NULL)  
    {  
        ProcessNode(p); // xử lý cụ thể tùy ứng dụng  
        p = p->pNext;  
    }  
}
```



**p = p->pNext;**

## Xuất dữ liệu danh sách ra màn hình

(khi đó, *ProcessNode(p)* : xuất dữ liệu của p ra màn hình)

```
void XuatDS(LIST l)
{
    NODE *p;
    if(l.pHead == NULL) // if(IsEmpty(l))
    {
        cout<<"\nDS rong!\n"; return; }
    cout<<"\nDu lieu cua Danh sach:\n";
    p = l.pHead;
    while (p!= NULL)
    {
        cout<< p->info <<'\t'; // ProcessNode(p)
        p = p->pNext;
    }
}
```

```
void ProcessNode(NODE* p)
{
    cout<< p->info <<'\t';
}
```

**Tìm một phần tử  
có thành phần dữ liệu là x trong danh sách đơn.**

- Không có; trả về NULL**
- Có; trả về địa chỉ nút đầu tiên, nút cuối cùng,...**

Trường hợp 1: Trả về địa chỉ nút đầu tiên, nếu có.

■ Bước 1:

□ `p = pHead;` //Cho p trở đến phần tử đầu danh sách

■ Bước 2: Trong khi (`p != NULL`) và (`p->info != x`) thực hiện:

`p = p->pNext;` // Cho p trở tới phần tử kế

■ Bước 3:

□ Nếu `p != NULL` thì p trở tới phần tử cần tìm

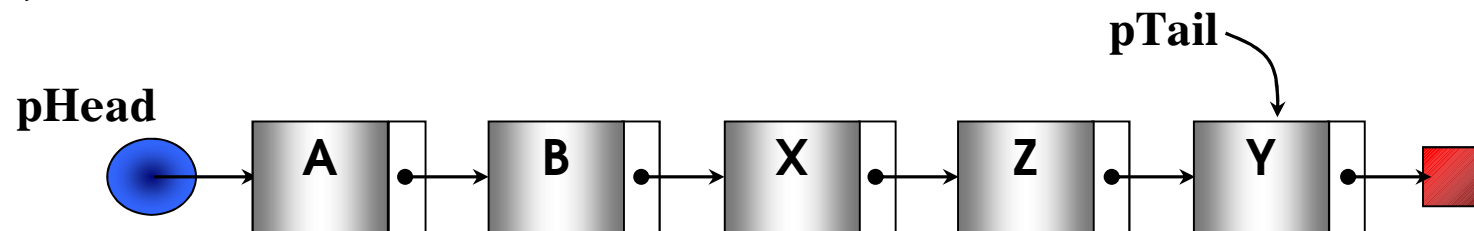
□ Ngược lại: không có phần tử cần tìm.

Input : l, x

Output : NULL; nếu không có;

p; nếu có //p chứa địa chỉ nút đầu tiên có info = x

```
NODE *Search(LIST l, data x) //NODE *SearchFirst(LIST l, data x)
{
    NODE *p;
    p = l.pHead;
    while((p!=NULL)&&(p->info != x))
        p = p->pNext;
    return p;
}
```





Input : l, x

Output : NULL; nếu không có;

p; nếu có //p chứa địa chỉ nút cuối cùng có info = x

```
NODE *Search_End(LIST l, data x)
{
    NODE*p, *kq=NULL;
    p = l.pHead;
    while (p != NULL)
    {
        if (p->info == x)
            kq = p;
        p = p->pNext;
    }
    return kq;
}
```

## Chèn một phần tử vào danh sách đơn

- Có 3 loại thao tác chèn `new_ele` vào DSLK:
  - 1. Chèn vào đầu danh sách
  - 2. Chèn vào cuối danh sách
  - 3. Chèn vào danh sách sau một phần tử `q`
- Mỗi loại có 2 trường hợp:
  - Có nút (chứa dữ liệu) sẵn, chèn nút.
  - Chỉ có dữ liệu `x`, phải tạo nút với dữ liệu `x`, rồi chèn nút mới tạo

## Chèn một phần tử vào đầu danh sách (Nút đã có)

- Nếu Danh sách rỗng Thì //Chèn vào DS rỗng

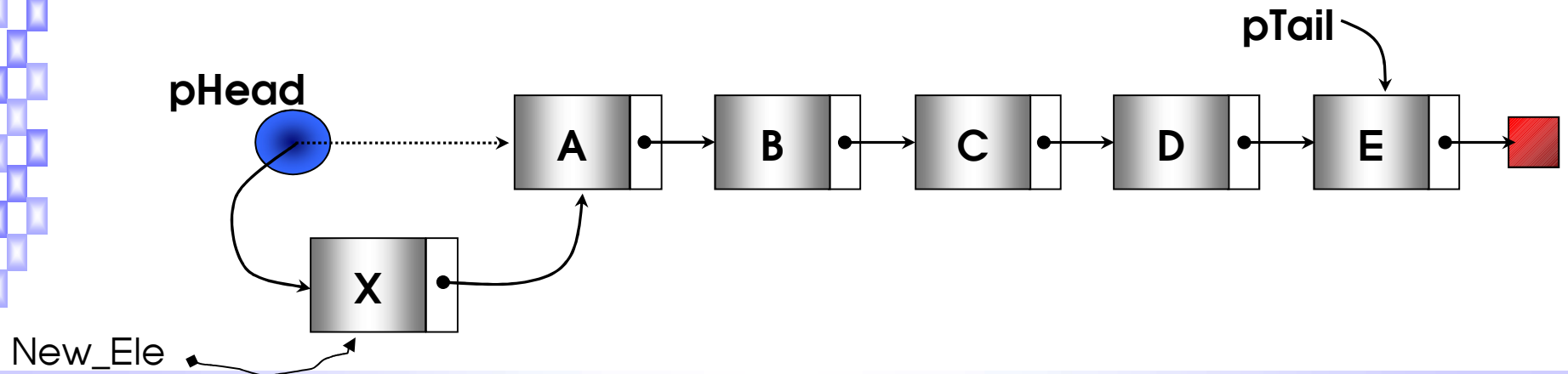
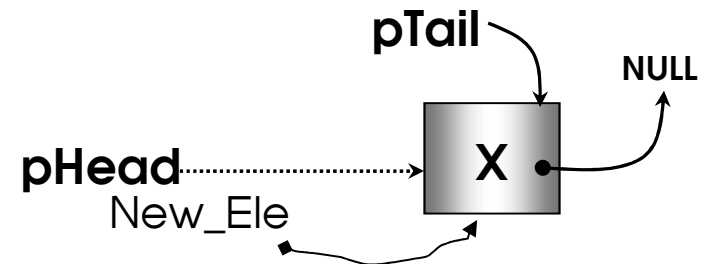
- B1.1 :  $pHead = new\_ele$ ;

- B1.2 :  $pTail = pHead$ ;

- Ngược lại

- B2.1 :  $new\_ele \rightarrow pNext = pHead$ ;

- B2.2 :  $pHead = new\_ele$  ;



**AddFirst(l,new\_ele):** Chèn một phần tử vào đầu danh sách

//Chèn nút new\_ele (đã tạo) vào đầu danh sách

```
void AddFirst(LIST &l, NODE* new_ele)
{
    if (l.pHead==NULL) //DSLK rỗng: if(IsEmpty(l))
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;
        l.pHead = new_ele;
    }
}
```

**InsertHead(l,x):** Chèn một phần tử vào đầu danh sách  
//Chức năng: tạo nút new\_ele có dữ liệu là x, sau đó chèn vào đầu DSLK

// Hàm trả về con trỏ lưu địa chỉ nút vừa tạo

//Chèn x vào đầu DS

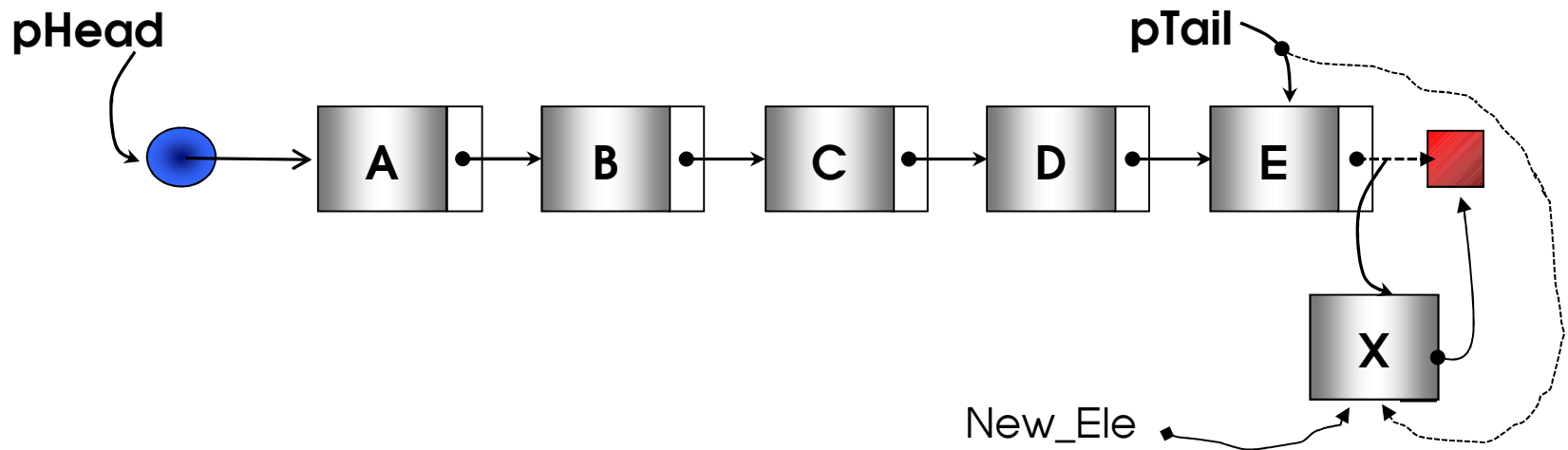
```
NODE* InsertHead(LIST &l,data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL) return NULL;
    if (l.pHead == NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;
        l.pHead = new_ele;
    }
    return new_ele;
}
```

**Hoặc có thể viết như sau:**

```
void InsertHead(LIST &l,data x)
{  NODE* new_ele = GetNode(x);
   if (new_ele == NULL)
       return; //exit(1);
   if (l.pHead == NULL) //DS rỗng
   {   l.pHead = new_ele; l.pTail = l.pHead;
       }
   else
   {   new_ele->pNext = l.pHead; //chèn vào đầu DS
       l.pHead = new_ele;
   }
}
```

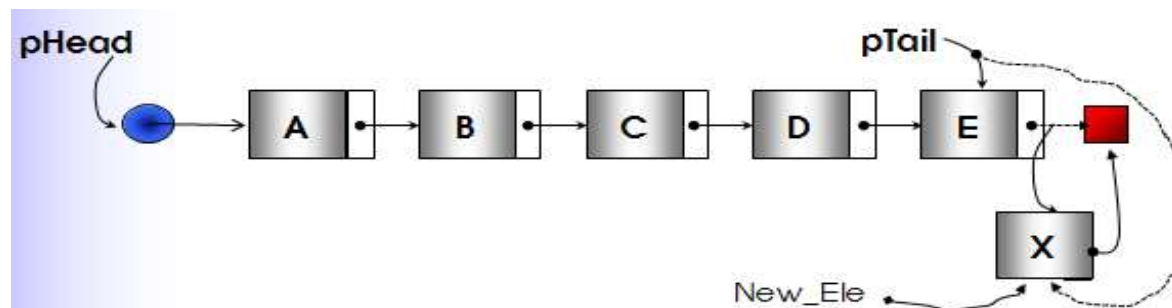
# Chèn một phần tử vào cuối danh sách

- Nếu Danh sách rỗng Thì
  - Chèn vào danh sách rỗng //như chèn đầu
- Ngược lại
  - B2.1 :  $pTail \rightarrow pNext = new\_ele$ ;
  - B2.2 :  $pTail = new\_ele$  ;



# Chèn một phần tử vào cuối danh sách

```
void AddTail(LIST &l, NODE *new_ele)
{
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;
        l.pTail = new_ele;
    }
}
```





## Chèn một phần tử vào cuối danh sách (tạo nút trước, chèn sau)

```
NODE* InsertTail(LIST &l, data x)
{
    NODE* new_ele = GetNode(x);

    if (new_ele == NULL)
        return NULL;
    if (l.pHead == NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;
        l.pTail = new_ele;
    }
    return new_ele;
}
```

**hoặc viết như sau:**

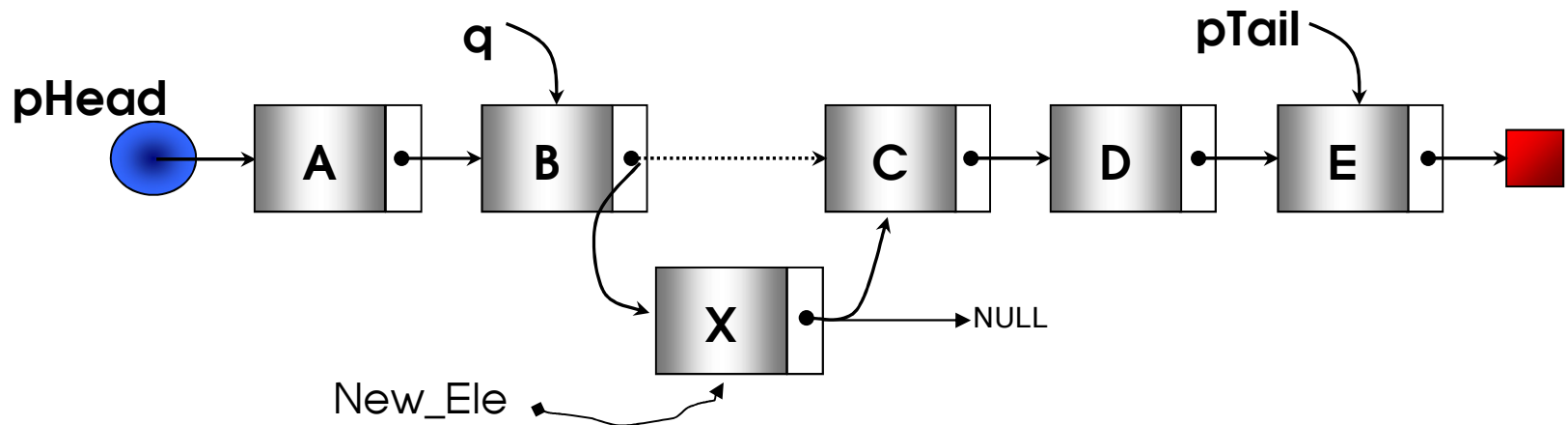
```
void InsertTail(LIST &l, data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL)
        return; //exit(1);
    if (l.pHead == NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;
        l.pTail = new_ele;
    }
}
```

# Chèn vào danh sách sau một phần tử q

■ Nếu (  $q \neq \text{NULL}$  ) thì

B1 :  $\text{new\_ele} \rightarrow \text{pNext} = q \rightarrow \text{pNext};$

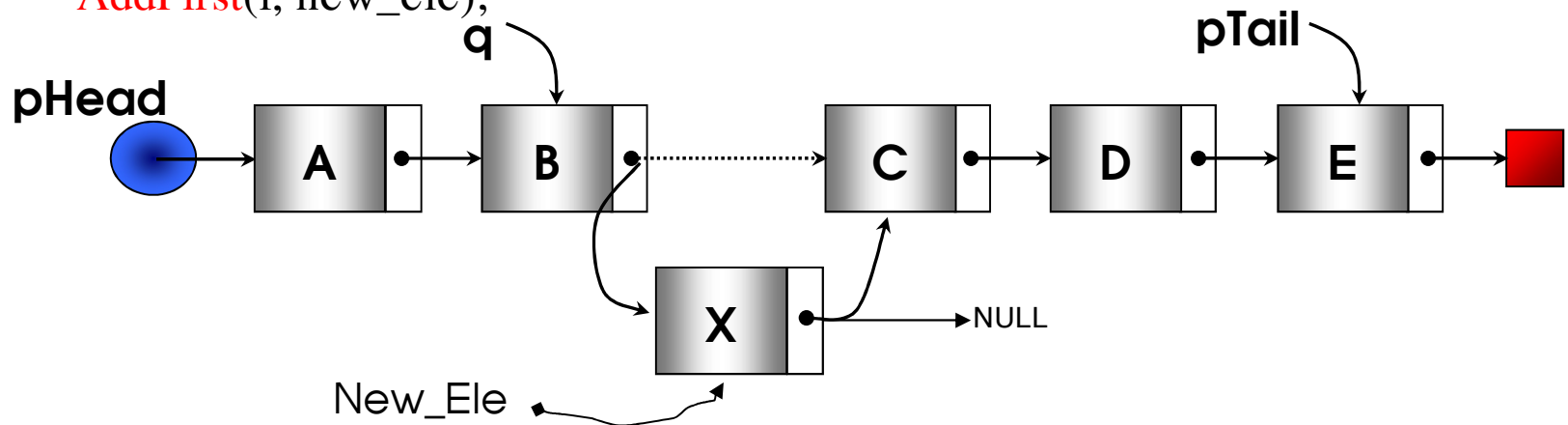
B2 :  $q \rightarrow \text{pNext} = \text{new\_ele};$



## Chèn vào danh sách sau một phần tử q

```
void AddAfter(LIST &l, NODE *q, NODE* new_ele)
```

```
{  
    if ( q!=NULL)  
    { new_ele->pNext = q->pNext;  
      q->pNext = new_ele;  
      if(q == l.pTail)//chen cuoi, cap nhat pTail  
        l.pTail = new_ele;  
    }  
    else //chen dau  
      AddFirst(l, new_ele);  
}
```



## Chèn vào danh sách sau một phần tử q

```
void InsertAfter(LIST &l, NODE *q, data x)
```

```
{
```

```
    NODE* new_ele = GetNode(x);
```

```
    if (new_ele == NULL) exit(1);
```

```
    if (q != NULL)
```

```
    { new_ele->pNext = q->pNext;
```

```
      q->pNext = new_ele;
```

```
      if(q == l.pTail)
```

```
        l.pTail = new_ele;
```

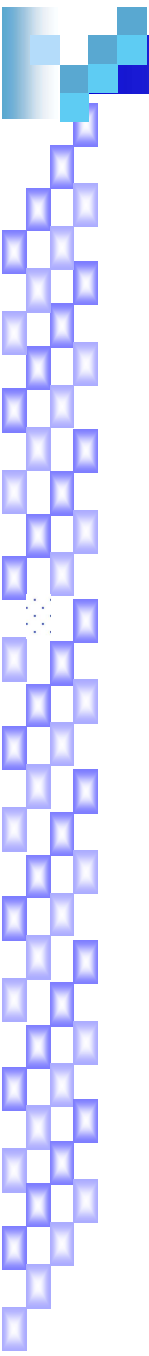
```
    }
```

```
    else
```

```
      AddFirst(l, new_ele);
```

```
}
```

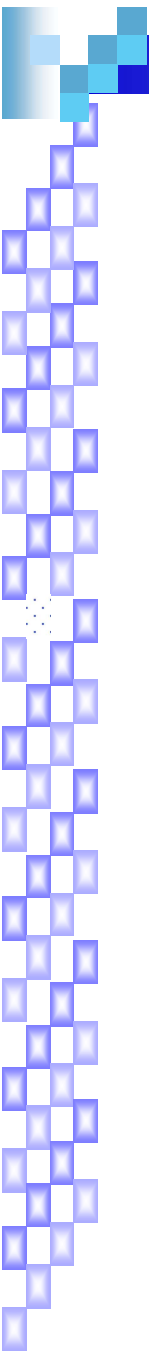
```
//Chen cuoi : InsertAfter(l, pTail, x);
```



## Nhập dữ liệu cho danh sách (sử dụng Chèn liên tiếp (đầu, cuối, ...) vào một danh sách rỗng sau khi nhập dữ liệu cho data của nút)

```
//Nhap Du lieu cho DSLKD : chèn cuối)
void NhapDS(LIST &l)
{   data x;
    ...

    CreatList(l);
    for(;;)
    {   cout<<"\nNhap x: "; cin>>x;
        if(x == Thoat)
        {   cout<<"\nDa nhap xong du lieu\n"; break; }
        InsertTail(l,x); // InsertHead(l,x);
    }
}
```



**Nhập dữ liệu cho danh sách từ mảng 1 chiều  
(sử dụng Chèn liên tiếp (đầu, cuối, ...) vào một danh sách rỗng sau  
khi nhập dữ liệu cho data của nút)**

```
//Nhap Du lieu cho DSLKD : chèn cuối)
void NhapDS(LIST &l, data a[], int n)
{   CreatList(l);
    int i;
    for(i = 0; i < n; i++)
        InsertTail(l, a[i]); //InsertHead(l, a[i]);
}
```

**Nhập dữ liệu cho danh sách từ 1 tập tin  
( Hàng đầu là số pt của tập tin, các hàng sau là dữ  
liệu)**

```
//Nhap Du lieu cho DSLKD : chèn cuối)
void NhapDS(LIST &l, char*f)
{
    ifstream in(f);
    if(!in)
        {cout<<"\nLoi mo tap tin!"; return;}
    CreatList(l);
    int n,x;
    in>>n;
    int i;
    for(i = 0;i < n;i++)
    {
        in>>x;
        InsertTail(l,x); // InsertHead(l,x);
    }
    in.close();
}
```

Nhập dữ liệu cho danh sách từ 1 tập tin  
(Không có kích thước tập tin, chỉ có dữ liệu )

```
//Nhap Du lieu cho DSLKD : chèn cuối)
void NhapDS(LIST &l, char*f)
{
    ifstream in(f);
    if(!in)
        {cout<<"\nLoi mo tap tin!"; return;}
    CreatList(l);
    int x;
    in>>x;
    InsertTail(l,x);
    while (!in.eof())
    {
        in>>x;
        InsertTail(l,x);
    }
    in.close();
}
```



**Chuyển f sang l : thành công, trả về 1**  
**Không thành công, trả về 0**

```
int NhapDS(LIST &l, char*f)
{
    ifstream in(f);
    if(!in)
        return 0;
    CreatList(l);
    int n,x, i;
    in>>n;
    for(i = 0;i < n;i++)
    {
        in>>x;
        InsertTail(l,x); // InsertHead(l,x);
    }
    in.close();
    return 1;
}
```

```
int NhapDS(LIST &l, char*f)
{
    ifstream in(f);
    if(!in)
        return 0;
    CreatList(l);
    int x;
    in>>x;
    InsertTail(l,x);
    while (!in.eof())
    {
        in>>x;
        InsertTail(l,x);
    }
    in.close();
    return 1;
}
```

# Hủy một phần tử khỏi danh sách

- Có 3 loại thao tác thông dụng hủy một phần tử ra khỏi DSLK.
  - Hủy phần tử đầu DSLK
  - Hủy một phần tử đứng sau phần tử q
  - Hủy 1 phần tử có dữ liệu là x
- Lưu ý:
  - Cấp phát bộ nhớ : dùng hàm **new**.
  - Giải phóng bộ nhớ : dùng hàm **delete**.
  - Khi hủy, nên cô lập trước nút hủy

# Hủy phần tử đầu DSLK

- Nếu (Head  $\neq$  NULL) thì

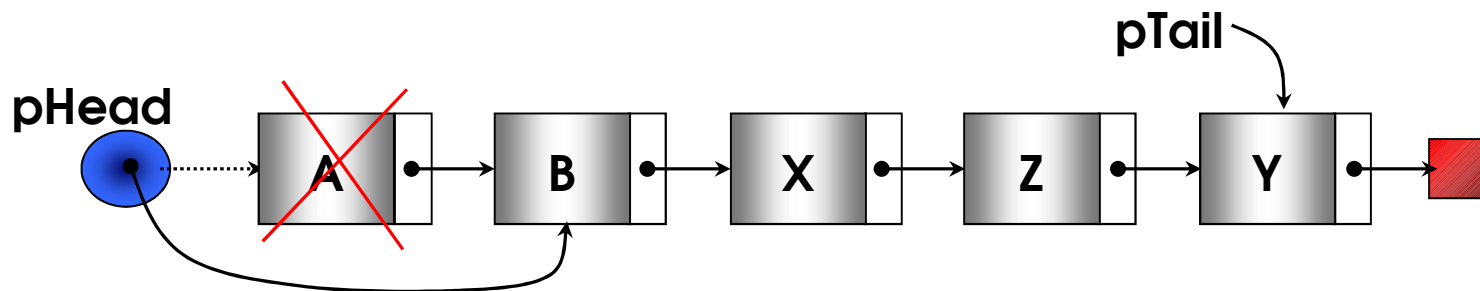
- B1:  $p = \text{Head};$  //  $p$  là phần tử cần hủy

- B2:

- B21 :  $\text{Head} = \text{Head} \rightarrow \text{pNext};$  // tách  $p$  ra khỏi DSLK

- B22 :  $\text{delete}(p);$  // hủy biến động do  $p$  trở đến

- B3: Nếu Head=NULL thì Tail = NULL; //DSLK rỗng



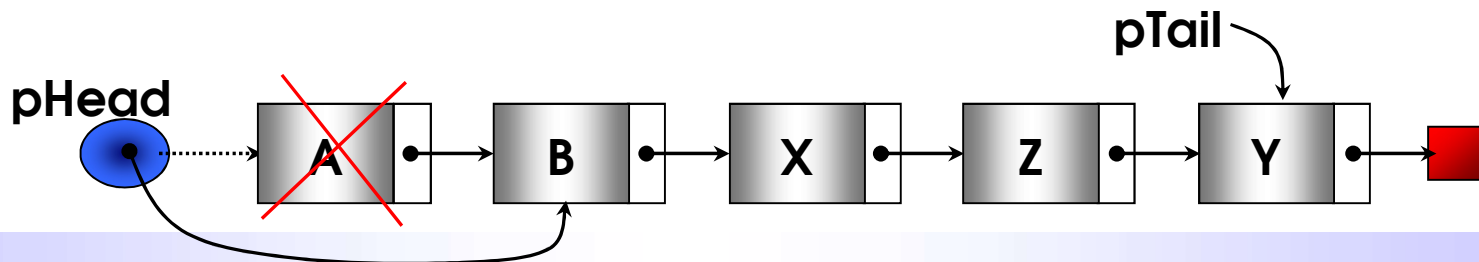
## **RemoveHead(l): Hủy phần tử đầu DSLK l**

**//Chức năng: Hủy pt đầu ra khỏi DSLK l**

**//Hàm trả về data của pt đầu đã bị Hủy**

```
data RemoveHead(LIST &l)
{
    NODE    *p;
    data     x = NULLDATA; //mot gia tri hang dac biet dinh nghia
                        //truoc, dung cho l rong

    if ( l.pHead != NULL)
    {
        p = l.pHead;
        x = p->info;
        l.pHead = l.pHead->pNext;
        delete p;
        if(l.pHead==NULL)
            l.pTail = NULL;
    }
    return x;
}
```



Nếu không cần lưu giữ lại dữ liệu đã hủy, có thể viết lại như sau:

```
void RemoveHead(LIST &l)
{
    NODE *p;
    if ( l.pHead != NULL)
    {
        p = l.pHead;
        l.pHead = l.pHead->pNext;
        delete p;
        if(l.pHead==NULL)
            l.pTail = NULL;
    }
}
```

# Hủy một phần tử đứng sau phần tử q

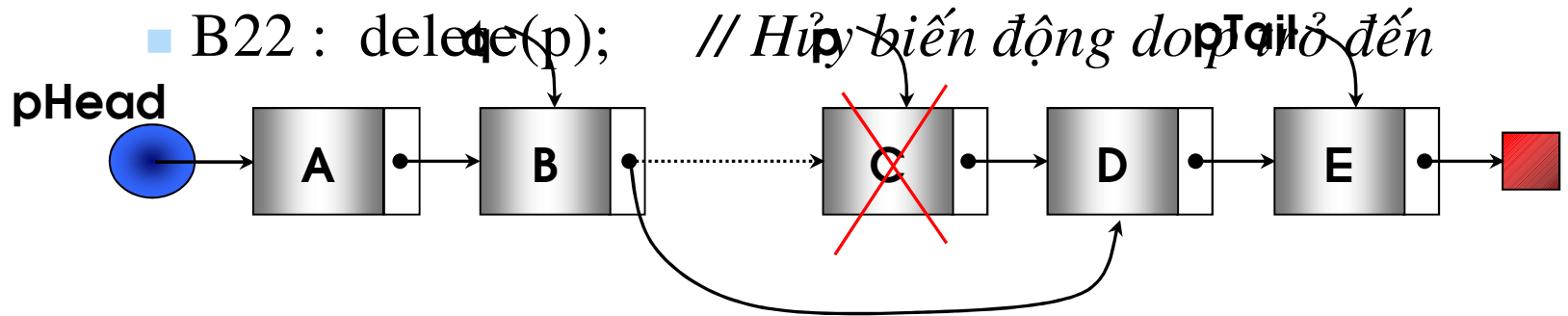
- Nếu ( $q \neq \text{NULL}$ ) thì

- B1:  $p = q \rightarrow \text{Next};$  //  $p$  là phần tử cần hủy

- B2: Nếu ( $p \neq \text{NULL}$ ) thì //  $q$  không phải là cuối  
//DSLK

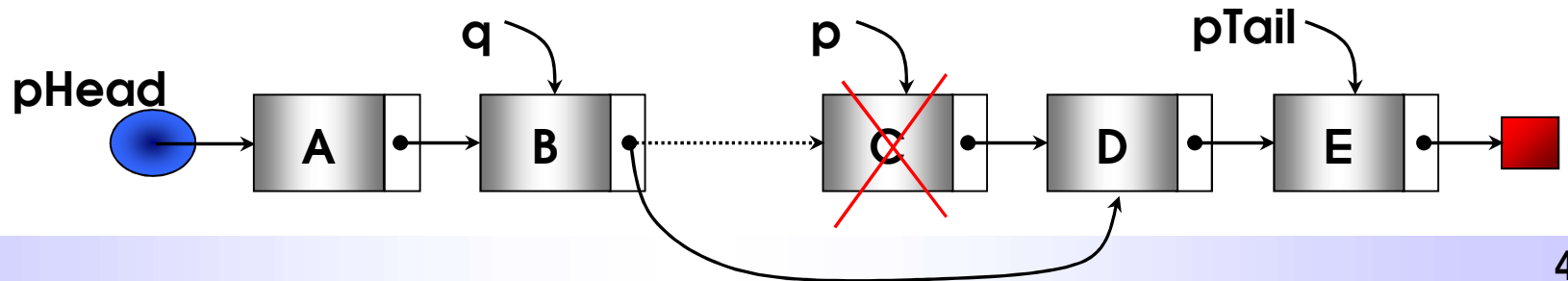
- B21 :  $q \rightarrow \text{Next} = p \rightarrow \text{Next};$  // tách  $p$  ra khỏi DSLK

- B22 :  $\text{delete}(p);$  // Hủy biến động do  $p$  chỉ đến



## Hủy một phần tử đứng sau phần tử q

```
void RemoveAfter (LIST &l, NODE *q)
{
    NODE *p;
    if ( q != NULL )
    {
        p = q ->pNext ; //nut bi huy
        if ( p != NULL ) //q không phải nút cuối
        {
            if ( p == l.pTail ) //nut bi huy la nút cuối
                l.pTail = q;
            q->pNext = p->pNext;
            delete p;
        } //else : sau Tail
    }
    else // Huy nút đầu
        RemoveHead(l);
}
```



## Hủy 1 phần tử (đầu tiên ) có dữ liệu là x (nếu có)

- Bước 1:
  - Tìm phần tử p (đầu tiên) có dữ liệu x và phần tử q đứng trước nó
- Bước 2:
  - Nếu  $(p \neq \text{NULL})$  thì // tìm thấy x
    - Hủy p ra khỏi DSLK như Hủy phần tử sau q;
  - Ngược lại
    - Báo không có pt nào có dữ liệu là x;



**RemoveNode(l, x)= 1; Neu thanh cong**  
**= 0; nguoc lai**

```
int RemoveNode(LIST &l, Data x)
{
    NODE    *p = l.pHead;
    NODE    *q = NULL; //giu lai dc cua nut truooc nut
                  //can xoa
    //Bước 1: Tìm phần tử p có data x và phần tử q đứng trước nó
    while( p != NULL)
    {   if(p->Info == x) break;
        q = p; p = p->pNext;
    }
    //Bước 2: Hủy phần tử có data x nếu tìm thấy
    .....
}
```

## Hủy phần tử (đầu tiên) có data x (*RemoveNode* 2/2)

```
int RemoveNode(LIST &l, Data x)
{ //Bước 1: Tìm phần tử p có data x và phần tử q đứng trước nó
.....
//Bước 2: Hủy phần tử có data x nếu tìm thấy
if(p == NULL) return 0; //Không tìm thấy x
if(q != NULL)
{ if(p == l.pTail) l.pTail = q;
  q->pNext = p->pNext;
}
else //p là phần tử đầu DSLK
{ l.pHead = p->pNext;
  if(l.pHead == NULL) l.pTail = NULL;
}
delete p;
return 1;
}
```

## Hủy toàn bộ danh sách

- Để Hủy toàn bộ danh sách, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan:
- Bước 1: Trong khi (Danh sách chưa hết) thực hiện
  - B11:
    - $p = \text{Head};$
    - $\text{Head} := \text{Head} \rightarrow \text{pNext};$  // Cho  $p$  trở tới phần tử kế
  - B12:
    - Hủy  $p$ ;
- Bước 2:
  - Tail = NULL; // Bảo đảm tính nhất quán khi DSLK rỗng

# Hủy toàn bộ danh sách

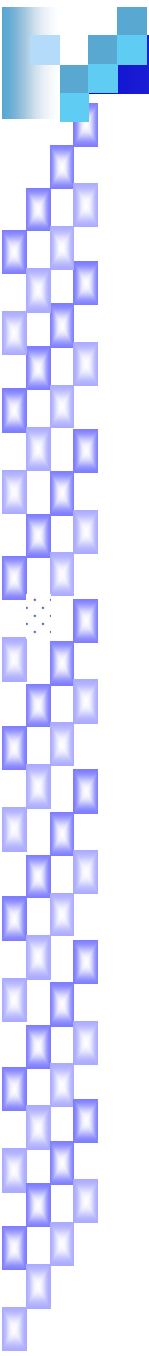
```
void RemoveList(LIST &l)
{
    NODE    *p;
    while (l.pHead != NULL)
    {   p = l.pHead;
        l.pHead = p->pNext;
        delete p;
    }
    l.pTail = NULL;
}
```



# Sắp xếp danh sách đơn

## ■ Các cách tiếp cận

- Phương án 1: Hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng Info).
- Phương án 2: Thay đổi các mối liên kết (thao tác trên vùng Next)



## **Phương án 1:**

### **Hoán vị nội dung các phần tử trong danh sách**

- Cài đặt lại trên DSLK các thuật giải sắp xếp đã biết trên mảng
- Điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên DSLK thông qua liên kết thay vì chỉ số như trên mảng.
- Thực hiện hoán vị nội dung của các phần tử khi cần thiết

```
void ListSelectionSort (LIST &l)
```

```
{  NODE    *min;  // chỉ đến phần tử có giá trị nhỏ nhất trong DSLK
```

```
  NODE    *p,*q;
```

```
  p = l.pHead;
```

```
  while(p != l.pTail)
```

```
  {  min = p;
```

```
    q = p->pNext;
```

```
    while(q != NULL)
```

```
    {
```

```
      if(q->Info < min->info )
```

```
        min = q; // ghi nhận vị trí
```

```
        // ghi nhận vị trí phần tử min hiện hành
```

```
        q = q->pNext;
```

```
    }
```

```
    // Hoán vị nội dung 2 phần tử
```

```
    Hoanvi(min->Info, p->info);
```

```
    p = p->pNext;
```

```
  }
```

```
}
```

```
void SelectionSort(int a[],int N )
{  int CSmin; // chỉ số phần tử nhỏ nhất
  //trong dãy hiện hành
  int i,j;
  for (i=0; i<N-1 ; i++)
  {  CSmin = i;
    for (j = i+1; j <N ; j++)
    {
      if (a[j ] < a[CSmin])
        CSmin = j; // ghi nhận
        //vị trí phần tử hiện nhỏ nhất
    }
    Hoanvi(a[CSmin], a[i]);
  }
}
```

```
void ListSelectionSort (LIST &l)
```

```
{
    NODE *min;
    NODE *p,*q;

    for(p = l.pHead; p != l.pTail; p = p->pNext)
    {
        min = p;
        for(q = p->pNext; q != NULL; q = q->pNext)
        {
            if(q->info < min->info )
                min = q;
        }
        // Hoán vị nội dung 2 phần tử
        Hoanvi(min->info, p->info);
    }
}
```

```
void SelectionSort(int a[],int N )
```

```
{
    int CSmin; //CS pt đầu tiên đạt min
    int i,j;

    for (i=0; i<N-1 ; i++)
    {
        CSmin = i;
        for(j = i+1; j <N ; j++)
        {
            if (a[j] < a[CSmin])
                CSmin = j;
        }
        Hoanvi(a[CSmin], a[i]);
    }
}
```



## Nhận xét

- Do thực hiện hoán vị nội dung của các phần tử nên đòi hỏi sử dụng thêm vùng nhớ trung gian  
⇒ chỉ thích hợp với các DSLK có các phần tử có thành phần Info kích thước nhỏ.
- Khi kích thước của trường Info lớn, việc hoán vị giá trị của hai phần tử sẽ chiếm chi phí đáng kể.
- Không tận dụng được các ưu điểm của DSLK!

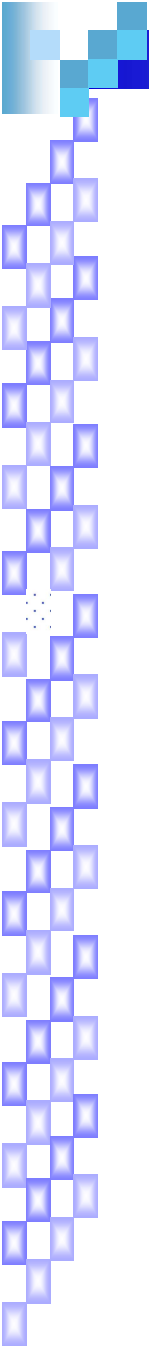
## Phương án 2: Thay đổi các mối liên kết

- Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn  $\Rightarrow$  chỉ thao tác trên các móc nối (pNext).
- Kích thước của trường pNext:
  - Không phụ thuộc vào bản chất dữ liệu lưu trong DSLK
  - Bằng kích thước 1 con trỏ (2 hoặc 4 byte trong môi trường 16 bit, 4 hoặc 8 byte trong môi trường 32 bit...)
- Thao tác trên các móc nối thường phức tạp hơn thao tác trực tiếp trên dữ liệu  
 $\Rightarrow$  Cần cân nhắc khi chọn cách tiếp cận. Nếu dữ liệu không quá lớn thì nên chọn phương án 1 hoặc một thuật giải hiệu quả nào đó.

# Sắp xếp danh sách

## Thay đổi các mối liên kết

- Một trong những cách thay đổi móc nối đơn giản nhất là tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ (đồng thời hủy danh sách cũ).
- Giả sử danh sách mới lRes, ta có phương án 2 của thuật toán chọn trực tiếp như sau:
  - Bước1: Khởi tạo danh sách mới lRes là rỗng;
  - Bước2: Tìm trong danh sách cũ 1 phần tử nhỏ nhất;
  - Bước3: Tách min khỏi danh sách l;
  - Bước4: Chèn min vào cuối danh sách lRes;
  - Bước5: Lặp lại bước 2 khi chưa hết danh sách Head;



**ListSelectionSort2 (I):** Sắp xếp danh sách  
bằng cách thay đổi các mối liên kết

```
void ListSelectionSort2 (LIST &l)
```

```
{
```

```
    LIST    lRes; // DS moi
```

```
    NODE    *min; // chỉ đến phần tử có giá trị nhỏ nhất trong DSLK cần sắp
```

```
    NODE    *p,*q, *minprev; //trước min
```

```
    CreatList(lRes); // khởi tạo lRes
```

```
    while(l.pHead != NULL) //mỗi lần lặp, l giảm đi 1 nút, lRes tăng 1 nút
```

```
    {
```

```
        p = l.pHead;
```

```
        q = p->pNext;
```

```
        min = p; minprev = NULL;
```

```
        while(q != NULL)
```

```
        {
```

```
            if(q->info < min->info)
```

```
            {
```

```
                min = q; minprev = p;
```

```
            }
```

```
            p = q; q = q->pNext;
```

```
        }
```

```
        if(minprev != NULL)
```

```
        {
```

```
            minprev->pNext = min->pNext; } //Cắt nút min
```

```
        else //min là nút đầu
```

```
            l.pHead = min->pNext;
```

```
            min->pNext = NULL;
```

```
            AddTail(lRes, min);
```

```
    } l = lRes;
```

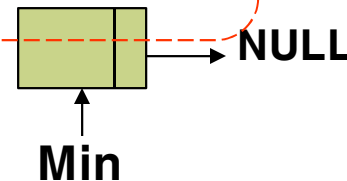
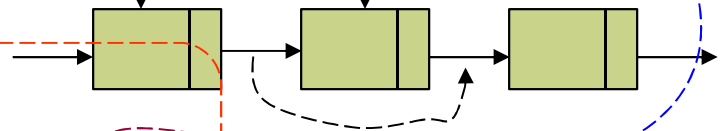
```
}
```



Ca 2 trường hợp, 1  
giảm nút min

Minprev

Min





# Các cấu trúc đặc biệt của danh sách đơn

- Stack (Ngăn xếp)
- Queue (Hàng đợi)



# Stack (Ngăn xếp)

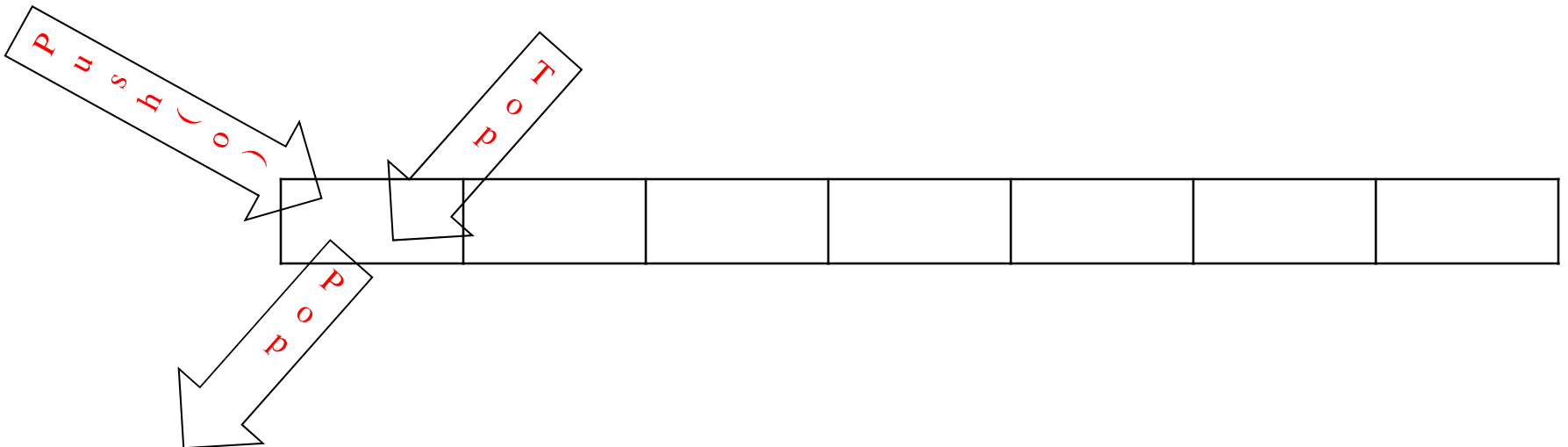
# Stack

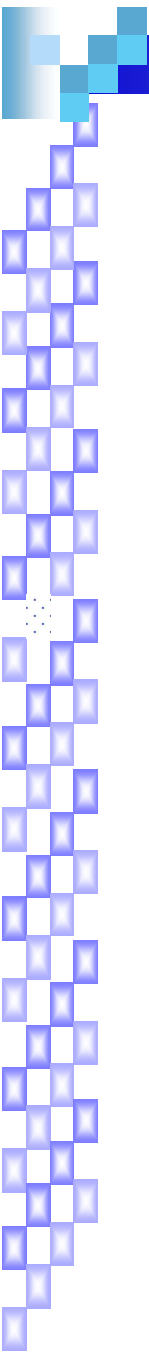
- Stack là một vật chứa (container) các đối tượng:
    - ❖ làm việc theo cơ chế LIFO (*Last In First Out, Vào sau ra trước*)
- ⇒ Việc thêm một đối tượng vào stack và lấy một đối tượng ra khỏi stack được thực hiện trên cùng một đầu.
- ❖ có nhiều ứng dụng:
    - khử đệ qui
    - tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui,
    - ứng dụng trong các bài toán tính toán biểu thức, ...



# Stack

- Stack là một CTDL trừu tượng hỗ trợ 2 thao tác chính:
  - Push(o)** : Thêm đối tượng o vào đầu stack
  - Pop()** :
    - Lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó.
    - Nếu stack rỗng thì lỗi sẽ xảy ra.



- 
- Stack cũng hỗ trợ một số thao tác khác:
    - **isEmpty()**: Kiểm tra xem stack có rỗng không.
    - **Top()**: Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

## Biểu diễn Stack:

Thường dùng:

- mảng

- danh sách liên kết đơn

Ở đây ta dùng DSLK đơn



## Biểu diễn Stack dùng danh sách liên kết đơn

- Có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn (DSLK).
- DSK có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác đều có thể thực hiện ở đầu DSK.

## Kiểu STACK cài đặt bằng danh sách liên kết đơn

- ❑ Giả sử đã có các định nghĩa:

```
typedef int data;
```

```
struct tagNode
```

```
{
```

```
    data                info;
```

```
    tagNode*          pNext;
```

```
};
```

```
// kiểu của một phần tử trong danh sách
```

```
typedef tagNode NODE;
```

- ❑ Cài đặt kiểu STACK (Danh sách liên kết) :

```
struct STACK
```

```
{
```

```
    NODE* pHead;
```

```
    NODE* pTail;
```

```
};
```

- Tạo Stack S rỗng:

```
void CreatStack(STACK &s)
{
    s.pHead=s.pTail= NULL;
}
```

- Kiểm tra stack rỗng :

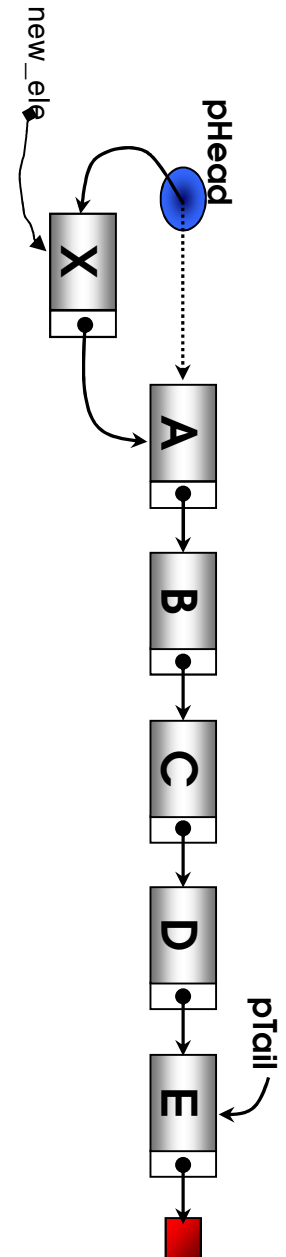
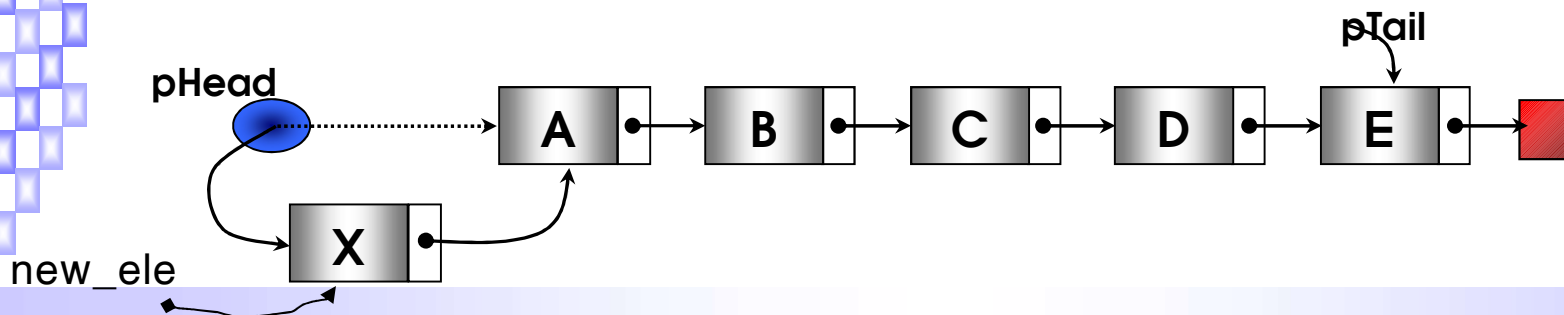
```
int IsEmpty(STACK s)
{ if (s.pHead == NULL) // stack rỗng
    return 1;
  return 0;
}
```

- 
- Thêm một phần tử x vào đầu stack S:

```
void Push(STACK &s, data x)
{
    InsertHead(s, x);
}
```

Viết trực tiếp như sau:

```
void Push(STACK &s, data x)
{
    NODE* new_ele = GetNode(x);
    if (s.pHead == NULL) //Stack rỗng
    {
        s.pHead = new_ele; s.pTail = s.pHead;
    }
    else
    {
        new_ele->pNext = s.pHead;
        s.pHead = new_ele;
    }
}
```





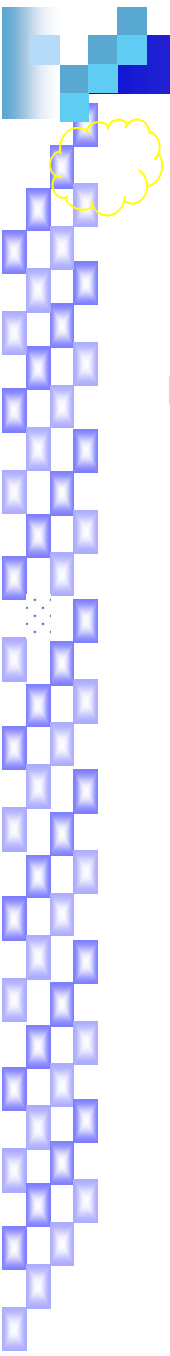
- Lấy thông tin và huỷ phần tử ở đỉnh stack S

```
data Pop(STACK &s)
{
    data x;
    if (s.pHead==NULL)
        return NULLDATA;
    x = RemoveHead(s);
    return x;
}
```

//NULLDATA là một giá trị đặc biệt nào đó được định nghĩa trước.

Viết trực tiếp như sau :

```
data Pop(STACK &s)
{ NODE *p;
  data x;
  if (s.pHead == NULL)
      return NULLDATA;
  p = s.pHead;
  x = p->info;
  s.pHead = s.pHead->pNext;
  delete p;
  if(s.pHead==NULL)
      s.pTail = NULL;
  return x;
}
```

- 
- Trả về phần tử ở đỉnh stack S
- ```
data Top(STACK s)
{
    if(s.pHead == NULL)
        return NULLDATA;
    return s.pHead->info;
}
```

## Ứng dụng

- Ngăn xếp có nhiều ứng dụng: *khử đệ quy, lưu vết* trong thuật toán *quay lui*, hay *tìm kiếm theo chiều sâu*, trong việc *chuyển đổi* giữa các dạng kí pháp khác nhau cũng như đánh giá các biểu thức chứa các toán tử không quá hai ngôi như biểu thức số học, lô-gic, ...
- Trong phần này ta xét các ứng dụng stack:
  - Chuyển đổi một biểu thức trung tố (toán tử đặt ở giữa hai toán hạng - đây là cách viết thông thường) sang dạng hậu tố (*toán tử đặt sau* các toán hạng).
  - Đánh giá các biểu thức số học (dạng hậu tố) : Sử dụng *ký pháp nghịch đảo Balan* (hay còn gọi là ký pháp hậu tố *RPN* - Reverse Polish Notation).

## Chuyển biểu thức trung tố sang hậu tố RPN:

### Thuật toán POLISH

**Input :** *sin* (biểu thức trung tố, là một chuỗi . . .)

**Output :** *sout* (biểu thức hậu tố, là một chuỗi . . .)

**Mô tả thuật toán :**

B1. Khởi tạo stack *s* rỗng (dùng để chứa các toán tử);

B2. Lặp lại các việc sau cho đến khi gặp dấu kết thúc biểu thức (NULL):

Đọc phần tử (ký tự) tiếp theo (hằng, biến, toán tử, ‘(’, ‘)’) trong biểu thức trung tố.

Nếu phần tử là:

- Toán hạng (hằng hoặc biến) : Cho ra output (sout).
- Dấu ‘(’: đẩy nó vào *s*;
- Dấu ‘)’ : lấy các toán tử trong stack ra và cho vào output cho đến khi gặp dấu mở ngoặc mở “(“.

(Dấu mở ngoặc cũng phải được đưa ra khỏi stack nhưng không cho vào Output)

- Toán tử:

\* Chùng nào đỉnh stack còn là toán tử và toán tử đó có độ ưu tiên **lớn hơn hoặc bằng** toán tử hiện tại thì lấy toán tử đó ra khỏi stack và cho vào output.

\* Đưa toán tử hiện tại vào stack s.

B3. *Khi gặp tín hiệu kết thúc biểu thức* thì lần lượt lấy các toán tử trong stack s ra và đưa vào output cho đến khi s rỗng;

+ Lưu ý về độ ưu tiên của các toán tử số học 2 ngôi :

Các toán tử 2 ngôi : '\*', '/', '%' có cùng độ ưu tiên và có độ ưu tiên cao hơn các toán tử sau (cùng độ ưu tiên) '+', '-', và qui ước rằng các toán tử trên có độ ưu tiên cao hơn toán tử ().

## Minh họa: (các toán hạng là ký số)

- Input:  $2*3+(6-4)$  //Biểu thức trung tố
- Output:  $2\ 3\ *\ 6\ 4\ -\ +$  //Biểu thức hậu tố RPN

|   | Sin[i] | Stack S | Sout  |
|---|--------|---------|-------|
|   |        |         | “”    |
| 1 | 2      |         | “2”   |
| 2 | *      | *       |       |
| 3 | 3      | *       | “23”  |
| 4 | +      | +       | “23*” |

|    | Sin[i] | Stack S | Sout      |
|----|--------|---------|-----------|
| 5  | (      | +(      | “23*”     |
| 6  | 6      | +(      | “23*6”    |
| 7  | -      | +(-     | “23*6”    |
| 8  | 4      | +(-     | “23*64”   |
| 9  | )      |         | “23*64-”  |
| 10 | NULL   | +       | “23*64-+” |

Vậy: Sout = “23\*64-+”

// ham chuyen tu trung to sang hau to

void TrungTo\_HauTo(char sin[MAX], char sout[MAX])

{

STACK s;//stack luu cac toan tu

char c, //Ky tu hien hanh - dang xet

x; //luu toan tu o dinh stack trong thao tac Pop(s);

int i; //duyet chuoi vao : sin

sout[0] = NULL; //Khong khoi dong se bi loi : Access violation writing location

CreatStack(s);

for (i = 0; sin[i] != NULL; i++)

{

    c = sin[i]; //ky tu dang xet

    if (LaToanHang(c) == 1) //la ky so : toan hang

        Chen\_Cuoi\_Chui(sout, c); //chen c vao cuoi sout

    else

        if (c == '(')

            Push(s, c); //day '(' vao stack s

        else

            if (c == ')')

                { x = Pop(s);

                while (x != '(') //khong dua '(' vao sout

                { Chen\_Cuoi\_Chui(sout, x);

                x = Pop(s);

                }

    }





```
else // la toan tu
```

```
{
```

```
    while (s.pHead != NULL)
```

```
    if (LaToanTu(s.pHead->info) == 1 &&
```

```
        Do_UuTien_ToanTu(s.pHead->info) >= Do_UuTien_ToanTu(c))
```

```
    {
```

```
        x = Pop(s);
```

```
        Chen_Cuoi_Chui(sout, x);
```

```
    }
```

```
    else
```

```
        break;
```

```
    //Push toan tu dang xet vao s
```

```
    Push(s, c);
```

```
}
```

```
//da het sin
```

```
while (s.pHead != NULL) //lay cac toan tu trong s chen vao cuoi sout
```

```
{
```

```
    x = Pop(s);
```

```
    Chen_Cuoi_Chui(sout, x);
```

```
}
```

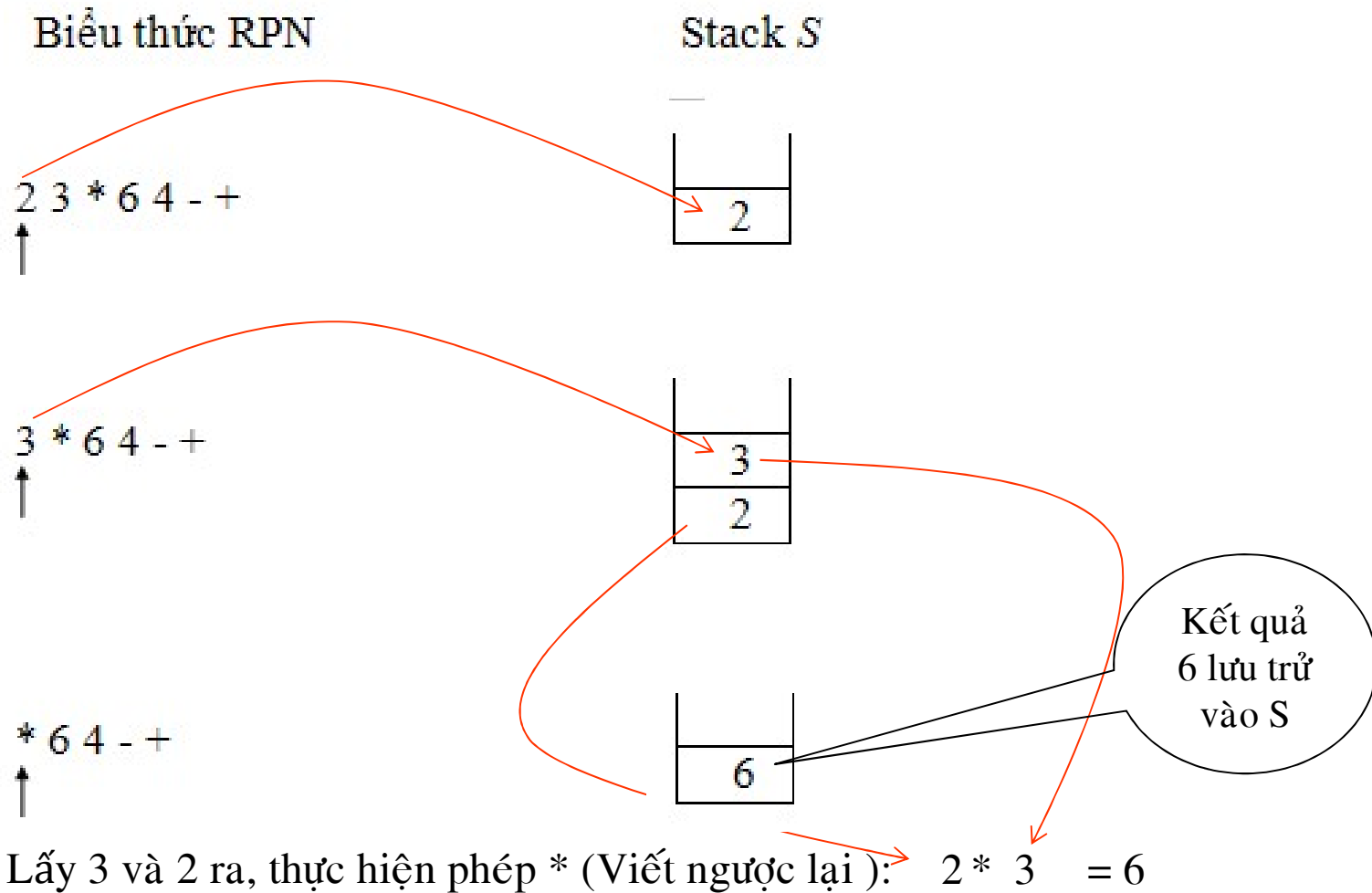
```
}
```

## Thuật toán đánh giá biểu thức hậu tố RPN

- Input: Biểu thức hậu tố
- Output: Giá trị của biểu thức trung tố tương ứng
- Mô tả thuật toán:
  1. Khởi tạo ngăn xếp  $S$  rỗng;
  2. Lặp lại các việc sau cho đến khi dấu kết thúc biểu thức được đọc:
    - Đọc phần tử (toán hạng, toán tử) tiếp theo trong biểu thức;
      - Nếu phần tử là toán hạng: đẩy nó vào  $S$ ;
      - Ngược lại: // phần tử là toán tử
        - ❖ Lấy từ đỉnh  $S$  hai toán hạng;
        - ❖ Áp dụng toán tử đó vào 2 toán hạng (theo thứ tự ngược);
        - ❖ Đẩy kết quả vừa tính trở lại  $S$ ;
  3. Khi gặp dấu kết thúc biểu thức (NULL), giá trị của biểu thức chính là giá trị ở đỉnh  $S$ ;

## Minh họa:

- Input: 2 3 \* 6 4 - + //Biểu thức hậu tố RPN
- Output: Giá trị biểu thức trung tố  $2*3+(6-4)$



6 4 - +  
↑

|   |
|---|
| 6 |
| 6 |

4 - +  
↑

|   |
|---|
| 4 |
| 6 |
| 6 |

- +  
↑

|   |
|---|
| 2 |
| 6 |

(Thực hiện phép toán -, lưu kết quả 2 vào S)

$$6 - 4 = 2$$

+  
↑

|   |
|---|
| 8 |
|---|

(Thực hiện phép toán +, lưu kết quả 8 vào S)

Kết thúc: Giá trị biểu thức = 8



```
int Tinh_BT_HauTo(char a[MAX])
```

```
{
    int i;
    char c;
    data x,y;
    STACK s;
    CreatStack(s);
    for (i = 0; a[i] != NULL; i++)
    {
        c = a[i];
        if (LaKySo(c) == 1)
        {
            x = So(c);
            Push(s, x);
        }
        else //toan tu
        {
            x = Pop(s);
            y = Pop(s);
            switch (c)
            {
```

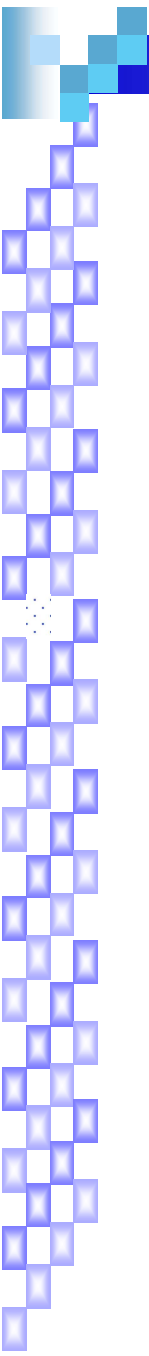
```
        case '+':
            Push(s,y + x);
            break;
        case '-':
            Push(s, y-x);
            break;
        case '*':
            Push(s, y*x);
            break;
        case '/':
            Push(s, y/x);
            break;
        case '%':
            Push(s, y % x);
            break;
        }
    }
}
return s.pHead->info;
}
```



# Queue (Hàng đợi)

# Hàng đợi ( Queue)

- Hàng đợi là một vật chứa (container) các đối tượng:
  - ❖ Làm việc theo cơ chế FIFO (*First In First Out*)
  - ❖  $\Rightarrow$  Thêm một đối tượng vào hàng đợi và lấy một đối tượng ra khỏi hàng đợi thực hiện theo cơ chế “*Vào trước ra trước*” :
    - ✓ Thêm một phần tử vào hàng đợi: luôn diễn ra ở cuối hàng đợi
    - ✓ Lấy một phần tử ra khỏi hàng đợi: luôn diễn ra ở đầu hàng đợi.
  - ❖ có nhiều ứng dụng:
    - ✓ xử lý đệ quy
    - ✓ tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui.
    - ✓ tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, ...

- 
- Hàng đợi có các thao tác chính:
    - **EnQueue**(o): Thêm đối tượng o vào cuối hàng đợi
    - **DeQueue**(): Lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.
  - Hàng đợi còn hỗ trợ các thao tác:
    - **IsEmpty**(): Kiểm tra xem hàng đợi có rỗng không.
    - **Front**(): Trả về giá trị của phần tử nằm ở đầu hàng đợi mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.





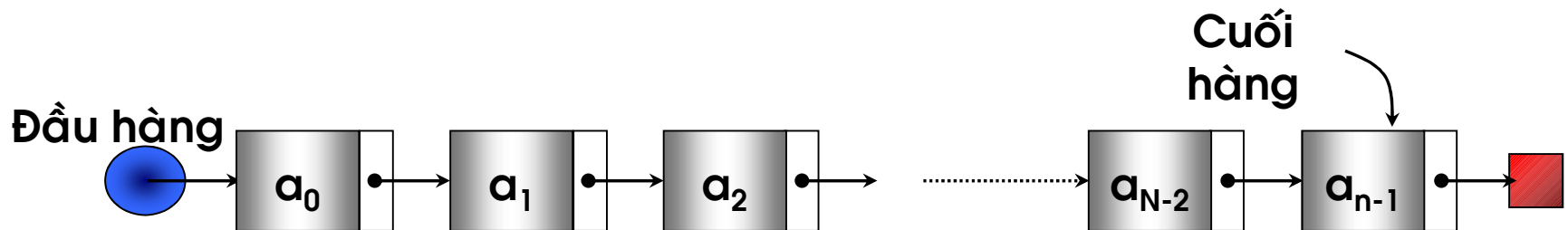
## **Biểu diễn hàng đợi (queue):**

- dùng mảng
- dùng danh sách liên kết đơn

**Biểu diễn hàng đợi  
bằng danh sách liên kết đơn**

# Biểu diễn hàng đợi dùng danh sách liên kết

- Có thể tạo một hàng đợi sử dụng một DSLK đơn.
- Phần tử đầu DSLK (head) sẽ là phần tử đầu hàng đợi, phần tử cuối DSLK (tail) sẽ là phần tử cuối hàng đợi.



## Kiểu QUEUE cài đặt bằng danh sách liên kết đơn

- ❑ Giả sử đã có các định nghĩa:

```
typedef int data;
```

```
struct tagNode
```

```
{
```

```
    data                info;
```

```
    tagNode*          pNext;
```

```
};
```

*// kiểu của một phần tử trong danh sách*

```
typedef tagNode NODE;
```

- ❑ Cài đặt kiểu QUEUE (Danh sách liên kết, STACK) :

```
struct QUEUE
```

```
{
```

```
    NODE* pHead;
```

```
    NODE* pTail;
```

```
};
```

- Tạo hàng đợi rỗng:

```
void CreatQ(QUEUE &q)
{
    q.pHead=q.pTail= NULL;
}
```

- Kiểm tra hàng đợi rỗng :

```
int IsEmpty(QUEUE q)
{
    if (q.pHead == NULL) // hàng đợi rỗng
        return 1;
    return 0;
}
```

- Ghi chú :

Câu lệnh trực tiếp kiểm tra hàng đợi khác rỗng :

```
if (q.pHead != NULL)
```

//Tạo nút có dữ liệu là x, chèn nút này vào cuối hàng đợi.

```
void EnQueue(QUEUE &q, data x) //InsertTai(q,x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL)
        return ; //exit(1);
    if (q.pHead == NULL)
    {
        q.pHead = new_ele;
        q.pTail = q.pHead;
    }
    else
    {
        q.pTail->Next = new_ele;
        q.pTail = new_ele;
    }
}
```

//Hủy nút đầu hàng đợi

data DeQueue(QUEUE &q) //RemoveHead(q)

{

    NODE \*p;

    data x;

    if (q.pHead == NULL)

        return NULLDATA;

    p = q.pHead;

    x = p->info;

    q.pHead = q.pHead->pNext;

    delete p;

    if(q.pHead==NULL)

        q.pTail = NULL;

    return x;

}

- Trả về phần tử ở đầu hàng đợi

```
data Front(QUEUE q)
{
    if (q.pHead == NULL)
        return NULLDATA;
    return
        q.pHead->info;
}
```



### Nhận xét:

- Các thao tác trên hàng đợi biểu diễn bằng danh sách liên kết làm việc với chi phí  $O(1)$ .
- Nếu không quản lý phần tử cuối DSLK, thao tác **Dequeue** sẽ có độ phức tạp  $O(n)$ .





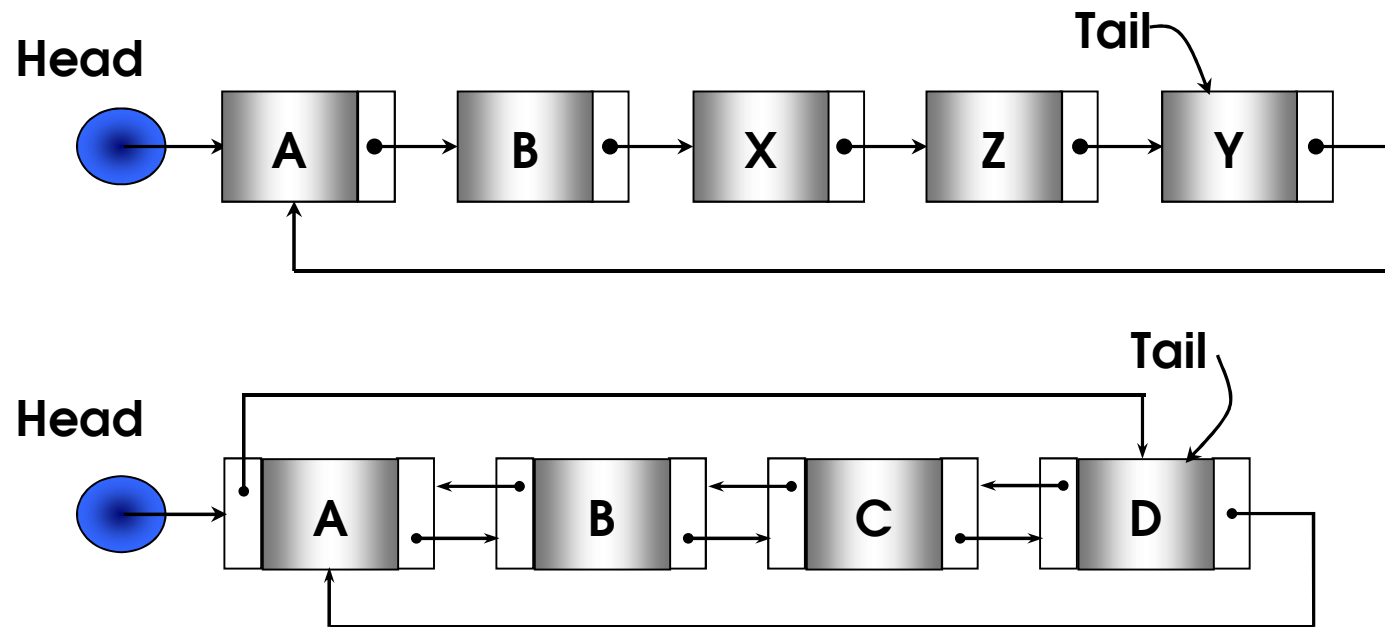
# Danh sách liên kết đơn vòng



## Danh sách liên kết đơn vòng

- Danh sách liên kết vòng (DSLK vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL, trở tới phần tử đầu danh sách.
- Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách.

- Để biểu diễn, có thể sử dụng các kỹ thuật biểu diễn như danh sách đơn (hoặc kép).



- Ta dùng danh sách đơn để biểu diễn DSLK vòng

# Danh sách liên kết đơn vòng

(Kiểu dữ liệu DSLK đơn vòng định nghĩa như DSLK đơn)

- Giả sử có các định nghĩa:

```
typedef int data;  
struct tagNode  
{  
    Data    info;  
    tagNode* pNext;  
};    // kiểu của một phần tử trong danh sách
```

```
typedef tagNode NODE;
```

```
// kiểu danh sách liên kết đơn vòng
```

```
struct LLIST  
{  
    NODE* pHead;  
    NODE* pTail;  
};
```

# Danh sách liên kết đơn vòng

- Tạo nút có thành phần dữ liệu là x:

```
NODE* GetNode(data x)
{
    NODE *p;
    p = new NODE;
    if (p != NULL)
    {
        p->info = x;
        p->pNext = NULL;
    }
    return p;
}
```

- Tạo DSLK đơn vòng rỗng:

```
void CreatLLIST(LLIST &l)
{
    l.pHead = l.pTail = NULL;
}
```

## DSLK ĐƠN vòng

- DSLK đơn vòng khác với DSLK đơn ở chỗ:  
Nút cuối, thành phần liên kết không chứa giá trị NULL mà trở đến phần tử đầu  
(Tức là: **`l.pHead == l.ptail->pNext`**)
- DSLK đơn vòng không có nút đầu rõ rệt (Nút nào cũng có thể chọn làm nút đầu), ta có thể đánh dấu một nút bất kỳ trên danh sách xem như nút đầu DSLK để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
NODE* Search(LLIST l, data x)
```

```
{
```

```
    NODE    *p;
```

```
    p = l.pHead;
```

```
    do
```

```
    {
```

```
        if ( p->info == x)
```

```
            return p;
```

```
        p = p->pNext;
```

```
    } while (p != l.pHead); // chưa đi giáp vòng
```

```
    return NULL;
```

```
}
```

# Danh sách liên kết đơn vòng

- Xuất DSLK đơn vòng ra màn hình :

```
void Output_Llist(LLIST l)
```

```
{
```

```
    NODE *p;
```

```
    if (l.pHead == NULL)
```

```
    {
```

```
        cout << "\nDS don vong rong!\n";
```

```
        return;
```

```
    }
```

```
    p = l.pHead;
```

```
    do
```

```
    {
```

```
        cout << p->info << "\t";
```

```
        p = p->pNext;
```

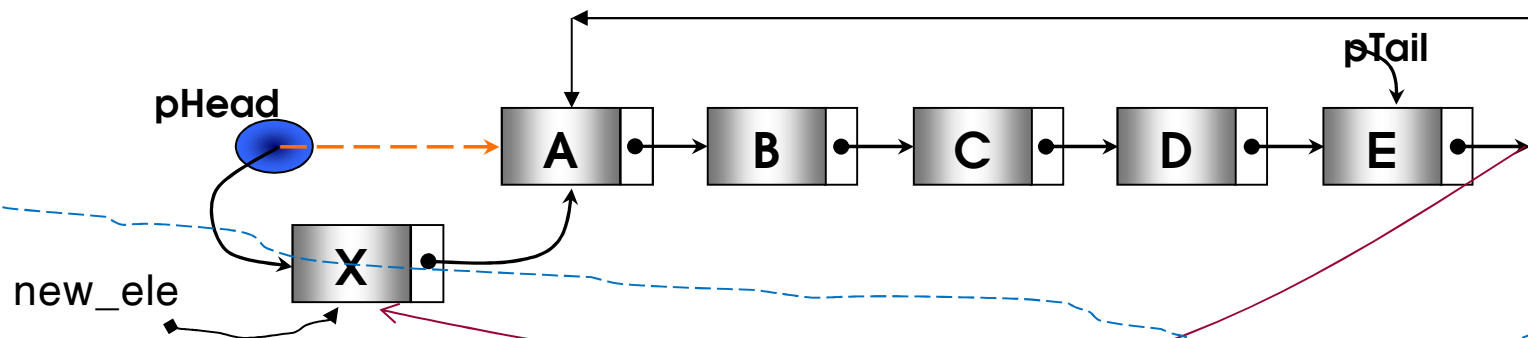
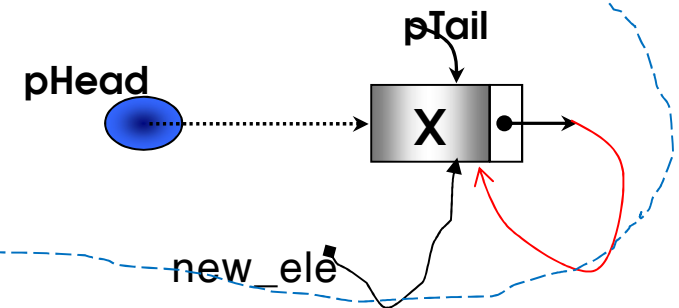
```
    } while (p != l.pHead); //Chua giap vong
```

```
}
```

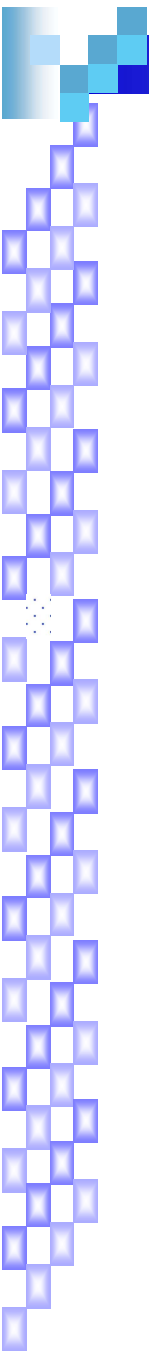
## Danh sách liên kết đơn vòng - Thêm phần tử đầu DSLK

```
void AddHead(LLIST &l, NODE *new_ele)
```

```
{  
    if(l.pHead == NULL) //DSLK vòng đơn rỗng  
    {  
        l.pHead = l.pTail = new_ele;  
        l.pTail->pNext = l.pHead;  
    }  
    else  
    {  
        new_ele->pNext = l.pHead;  
        l.pTail->pNext = new_ele;  
        l.pHead = new_ele;  
    }  
}
```







//Ham chen x vao dau l : Tao nut co info la x, next tro toi NULL, chen nut nay vao dau l

void InsertHead(LLIST &l, data x)

{

    NODE\* new\_ele = GetNode(x);

    if (new\_ele == NULL)

    {

        cout << "\nLoi cap phat bo nho! khong thuc hien duoc thao tac nay";

        return;

    }

    if (l.pHead == NULL)//Chen vao DS rong

    {

        l.pHead = new\_ele;

        l.pTail = l.pHead;

        l.pTail->pNext = l.pHead;

    }

    else

    {

        new\_ele->pNext = l.pHead;

        l.pTail->pNext = new\_ele;

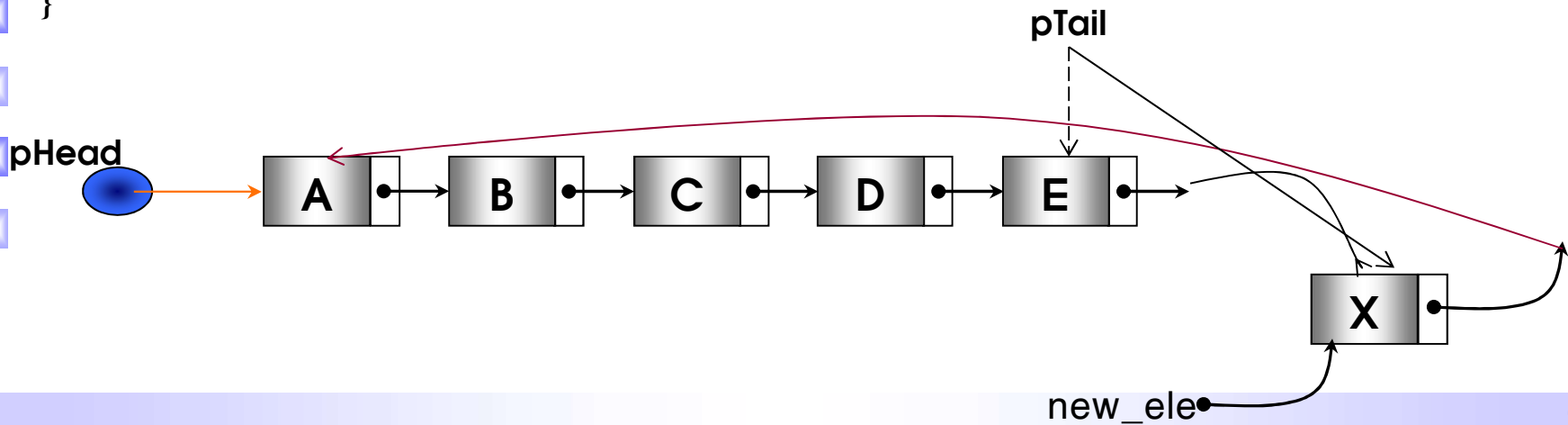
        l.pHead = new\_ele;

    }

}

## Danh sách liên kết vòng - Thêm phần tử cuối DSLK

```
void AddTail(LLIST &l, NODE *new_ele)
{
    if(l.pHead == NULL) //DSLK rỗng
    {
        l.pHead = l.pTail = new_ele;
        l.pTail->pNext = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;
        l.pTail->pNext = new_ele;
        l.pTail = new_ele;
    }
}
```





```
//Chen cuoi
```

```
void InsertTail(LLIST &l, data x)
```

```
{
```

```
    NODE* new_ele = GetNode(x);
```

```
    if (new_ele == NULL)
```

```
    {
```

```
        cout << "\nLoi cap phat bo nho! khong thuc hien duoc thao tac nay";
```

```
        return;
```

```
    }
```

```
    if (l.pHead == NULL)//Chen vao DS rong
```

```
    {
```

```
        l.pHead = new_ele;
```

```
        l.pTail = l.pHead;
```

```
        l.pTail->pNext = l.pHead;
```

```
    }
```

```
    else
```

```
    {
```

```
        new_ele->pNext = l.pHead;
```

```
        l.pTail->pNext = new_ele;
```

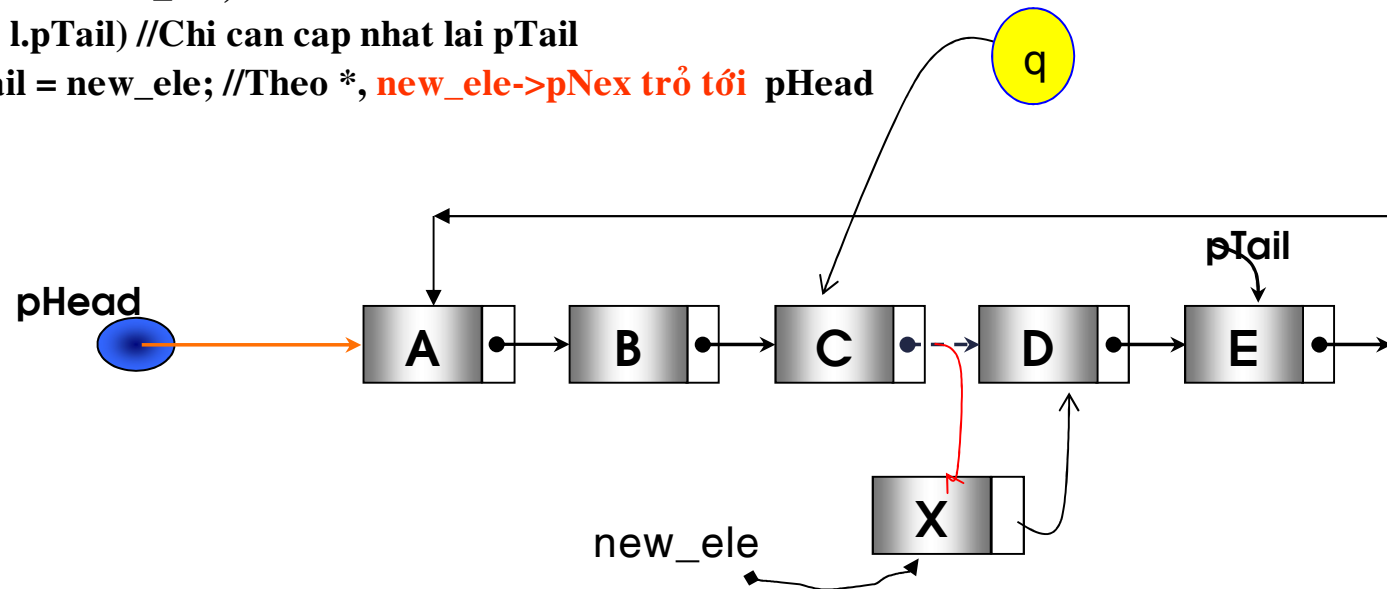
```
        l.pTail = new_ele;
```

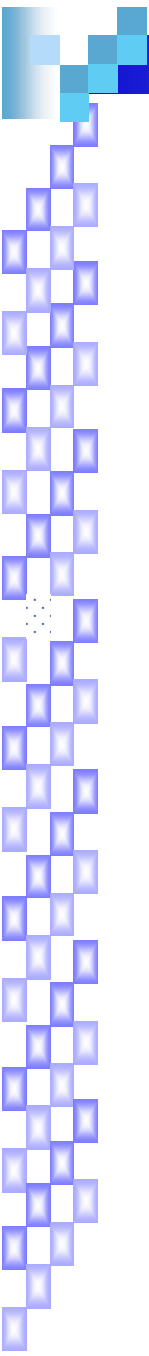
```
    }
```

```
}
```

## Danh sách liên kết vòng - Thêm phần tử sau nút q

```
void AddAfter(LLIST &l, NODE *q, NODE *new_ele)
{
    if(l.pHead == NULL) //DSLK rỗng
    {
        l.pHead = l.pTail = new_ele;
        l.pTail->pNext = l.pHead;
    }
    else
    {
        new_ele->pNext = q->pNext; // *
        q->pNext = new_ele;
        if(q == l.pTail) //Chỉ cần cập nhật lại pTail
            l.pTail = new_ele; //Theo *, new_ele->pNex trở tới pHead
    }
}
```





```
//Tập tin -> LLIST
```

```
//Định dạng tập tin : các phần tử là số nguyên, sẽ là dữ liệu tương ứng của các nút trong DS
```

```
//dùng chèn cuối
```

```
int File_LLIST(char *f, LLIST &l)
```

```
{
```

```
    ifstream in(f);          //Mở để đọc
```

```
    if (!in)
```

```
        return 0;
```

```
    CreatLLIST(l);
```

```
    data x;
```

```
    in >> x;
```

```
    InsertTail(l, x);
```

```
    while (!in.eof())
```

```
    {
```

```
        in >> x;
```

```
        InsertTail(l, x);
```

```
    }
```

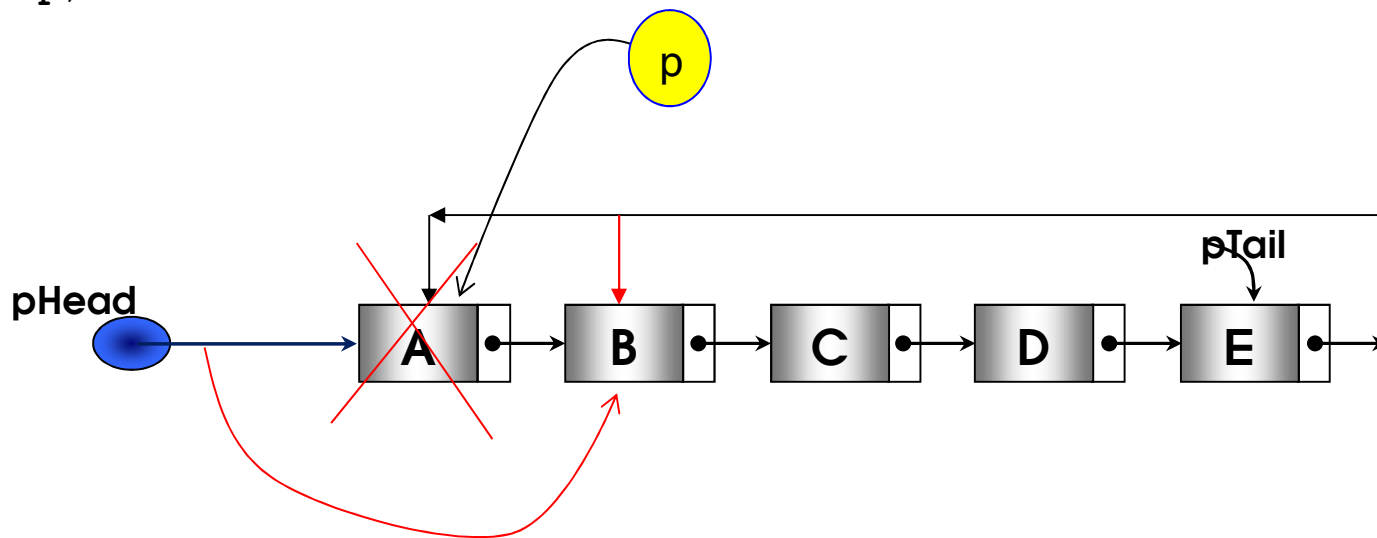
```
    in.close();
```

```
    return 1;
```

```
}
```

## Hủy phần tử đầu DSLK đơn vòng

```
void RemoveHead(LLIST &l)
{
    NODE    *p = l.pHead;
    if(p == NULL) return; //rỗng
    if (l.pHead == l.pTail) //1 nút
        l.pHead = l.pTail = NULL;
    else //nhieu nut : >1
    {
        l.pHead = p->Next; // Cap nhat lai Head, co lap p
        l.pTail->pNext = l.pHead; //Cap nhat Next của Tail, co lap p
    }
    delete p;
}
```

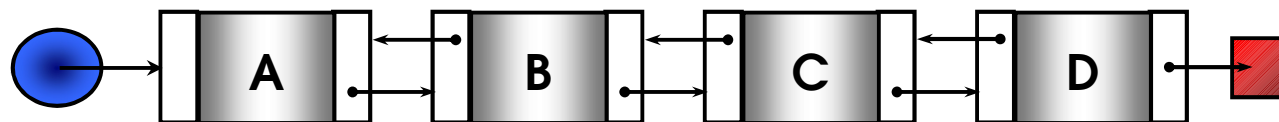
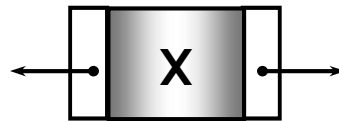




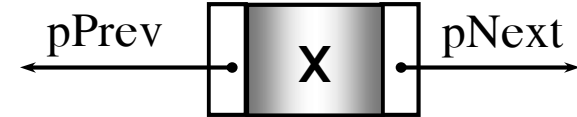
# Danh sách liên kết kép

# Danh sách liên kết kép

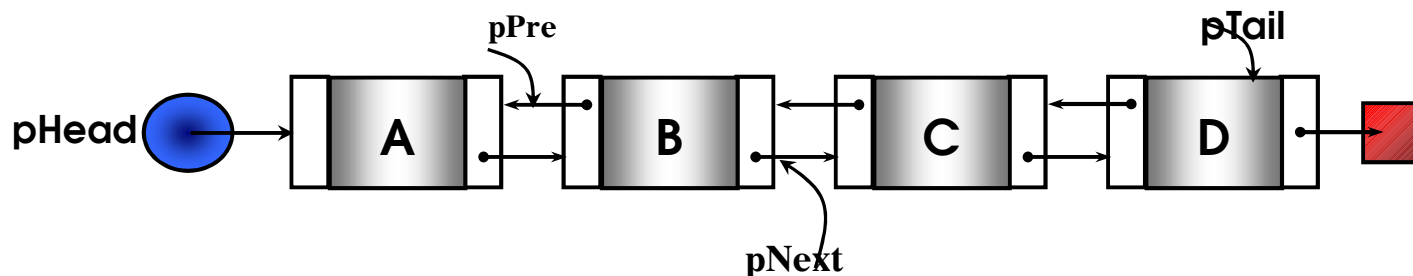
- Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.







- Tại mỗi nút trong DSLK kép có 3 thành phần, một là thành phần dữ liệu, 2 thành phần còn lại là thành phần liên kết biểu diễn bằng hai con trỏ:
  - pPrev liên kết với phần tử đứng trước . Nếu là nút đầu trong DSLK kép, thì pPrev có giá trị NULL
  - pNext liên kết với phần tử đứng sau. Nếu là nút cuối trong DSLK kép thì pNext có giá trị NULL.
- Ta quản lý DSLK kép bằng hai con trỏ:
  - pHead : chứa địa chỉ nút đầu
  - pTail : chứa địa chỉ nút cuối



## Cài đặt Danh sách liên kết kép :

```
typedef int data; // Kiểu thành phần dữ liệu của nút
struct tagDNode // kiểu các nút
{
    data      info;
    tagDNode* pPre; // trỏ đến phần tử đứng trước
    tagDNode* pNext; // trỏ đến phần tử đứng sau
}
```

//Đổi tên

```
typedef tagDNode DNODE;
```

//Kiểu DSLK kép

```
struct DLIST
```

```
{
    DNODE* pHead; // trỏ đến phần tử đứng trước
    DNODE* pTail; // trỏ đến phần tử đứng sau
};
```

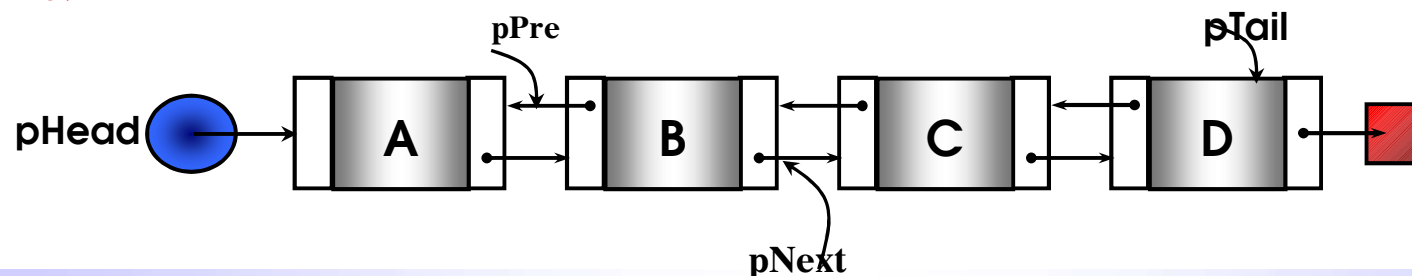
Nếu :

DLIST l;

Thì :

l.pHead -> pPre == NULL

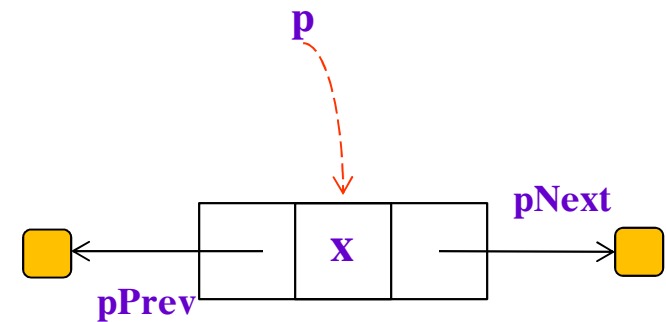
l.pTail -> pNext == NULL



# Danh sách liên kết kép

Tạo 1 phần tử cho danh sách liên kết kép:

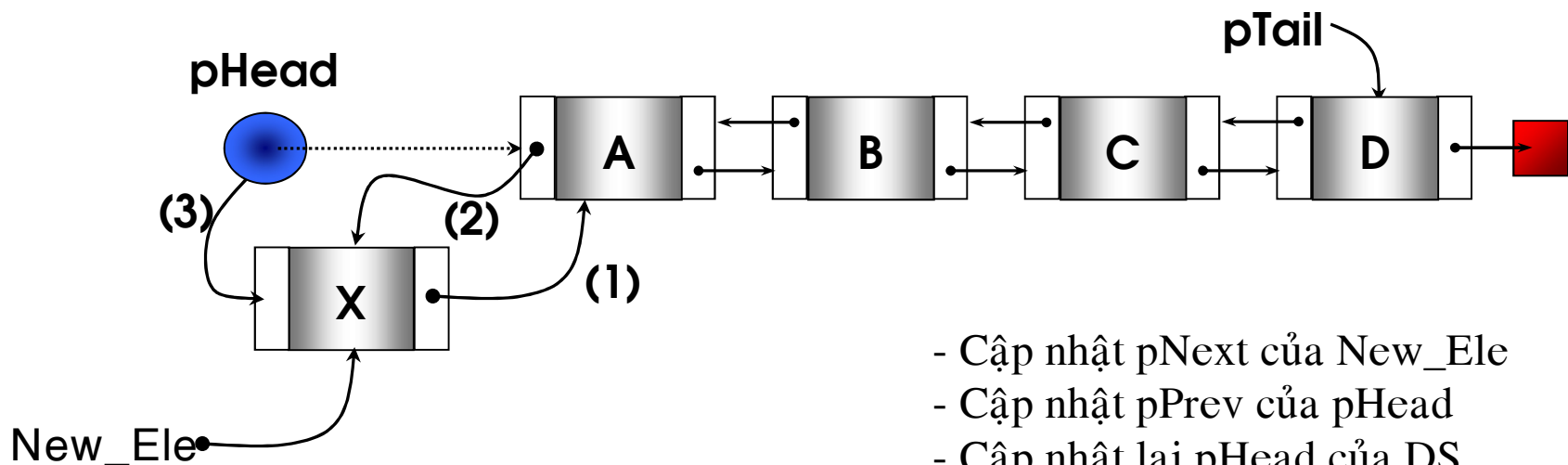
```
DNODE*   GetNode(data x)  
{   DNODE *p;  
    // Cấp phát vùng nhớ cho phần tử  
    p = new DNODE;  
    if ( p!=NULL)  
    { // Gán thông tin cho phần tử p  
        p->info = x;  
        p->pPrev = p->pNext = NULL;  
    }  
    return p;  
}
```



# Chèn một phần tử vào danh sách liên kết kép

- Có 4 loại thao tác chèn `new_ele` vào danh sách:
  - TH 1: Chèn vào đầu danh sách
  - TH 2: Chèn vào cuối danh sách
  - TH 3 : Chèn vào danh sách sau một phần tử `q`
  - TH 4 : Chèn vào danh sách trước một phần tử `q`

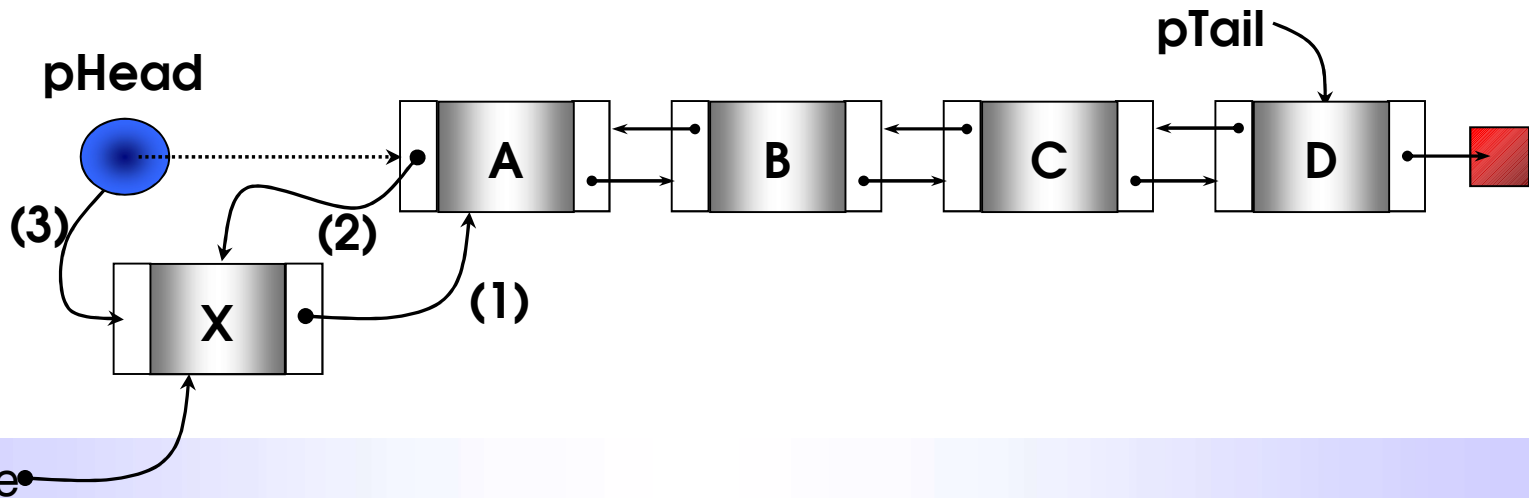
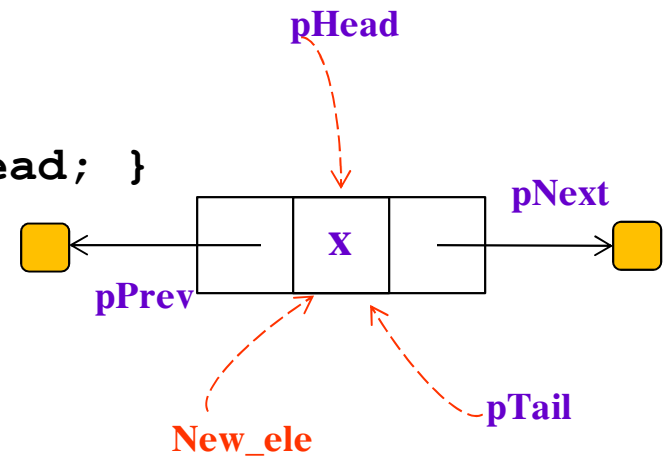
# Chèn một phần tử vào đầu danh sách liên kết kép



- Cập nhật `pNext` của `New_Ele`
- Cập nhật `pPrev` của `pHead`
- Cập nhật lại `pHead` của DS

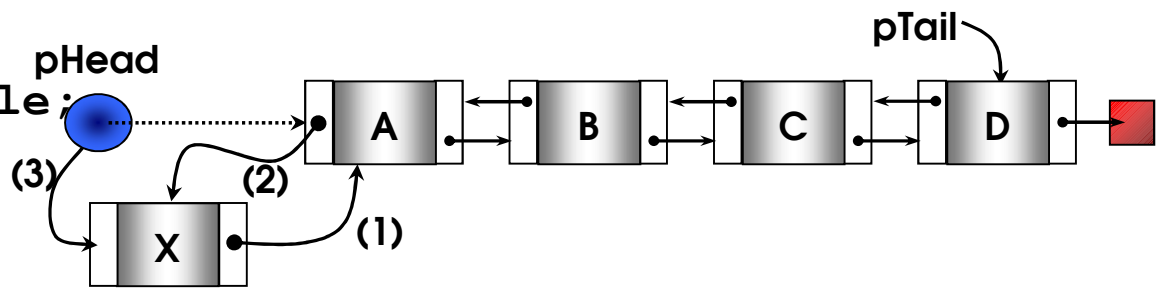
# Chèn một phần tử vào đầu danh sách liên kết kép

```
void AddFirst(DLIST &l, DNODE* new_ele)
{ if (l.pHead==NULL) //DS rỗng
  { l.pHead = new_ele; l.pTail = l.pHead; }
  else
  { new_ele->pNext = l.pHead; // (1)
    l.pHead->pPrev = new_ele; // (2)
    l.pHead = new_ele; // (3)
  }
}
```

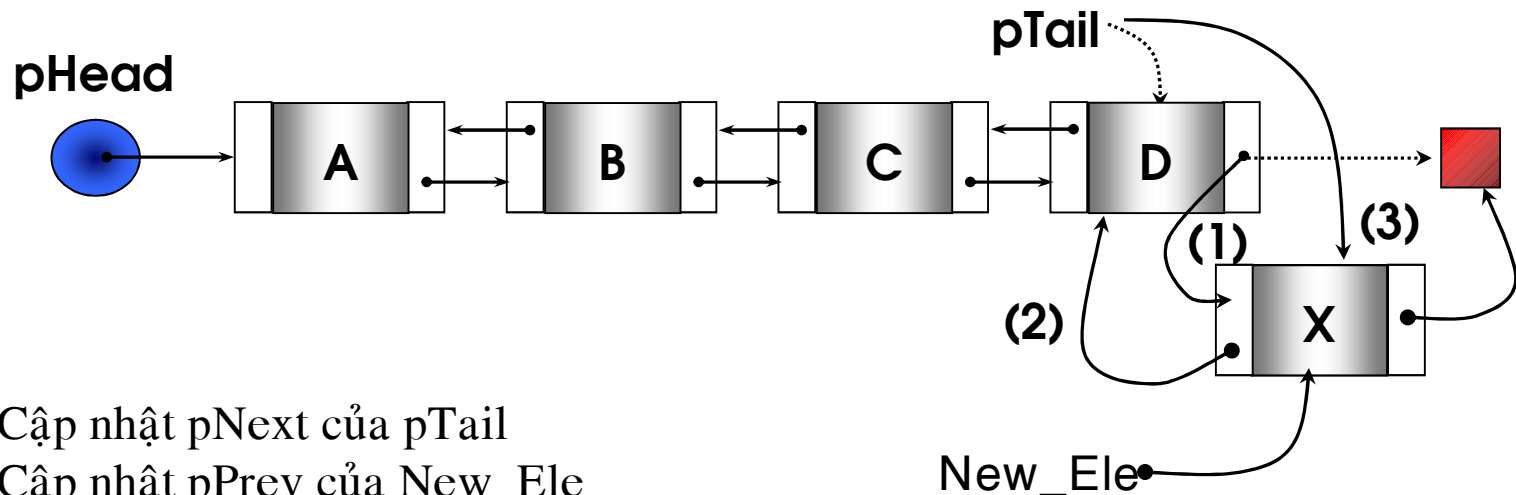


# Chèn một phần tử vào đầu danh sách liên kết kép

```
NODE* InsertHead(DLIST &l, Data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL) return NULL;
    if (l.pHead == NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead; // (1)
        l.pHead->pPrev = new_ele; // (2)
        l.pHead = new_ele;       // (3)
    }
    return new_ele;
}
```



## Chèn một phần tử vào cuối danh sách liên kết kép

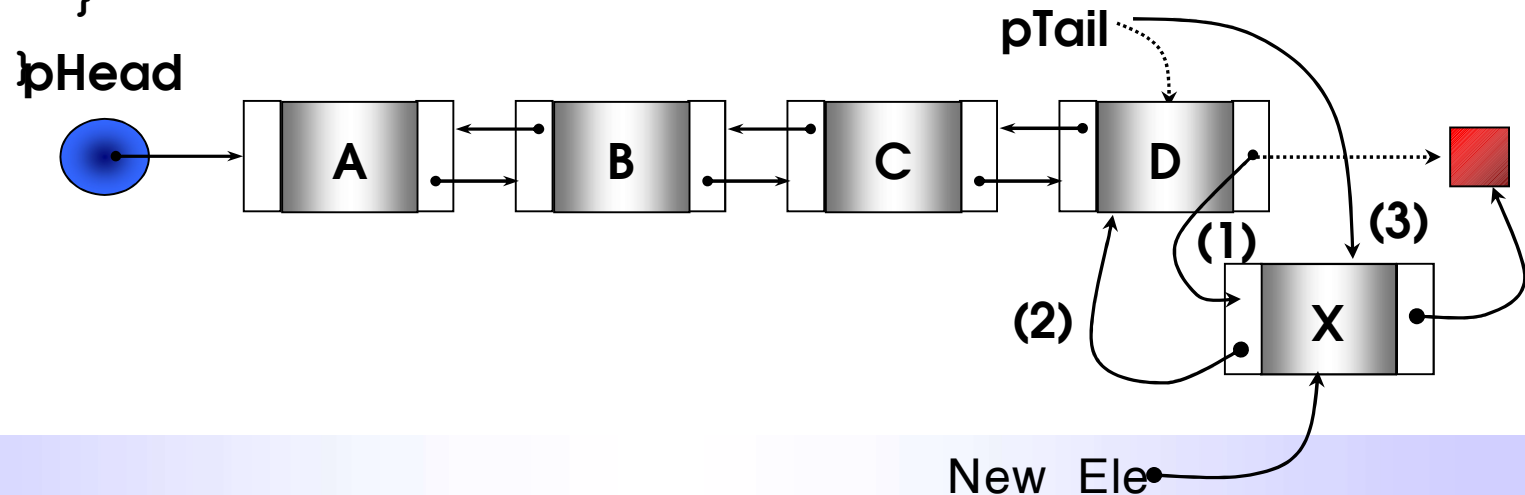


- Cập nhật pNext của pTail
- Cập nhật pPrev của New\_Ele
- Cập nhật lại pTail của DS



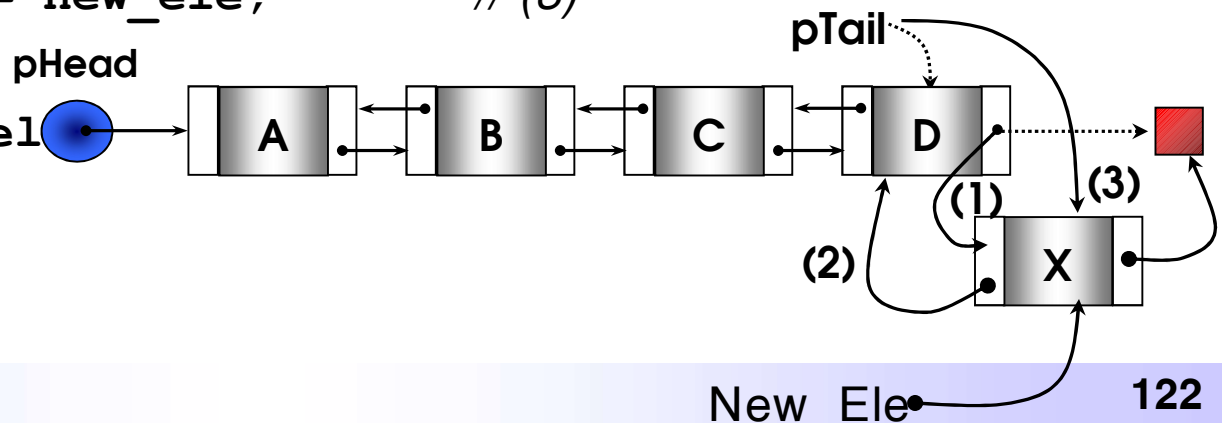
# Chèn một phần tử vào cuối danh sách liên kết kép

```
void AddTail(DLIST &l, DNODE *new_ele)
{ if (l.pHead==NULL)
  { l.pHead = new_ele; l.pTail = l.pHead; }
  else
  { l.pTail->Next = new_ele; // (1)
    new_ele->pPrev = l.pTail; // (2)
    l.pTail = new_ele;       // (3)
  }
}
```

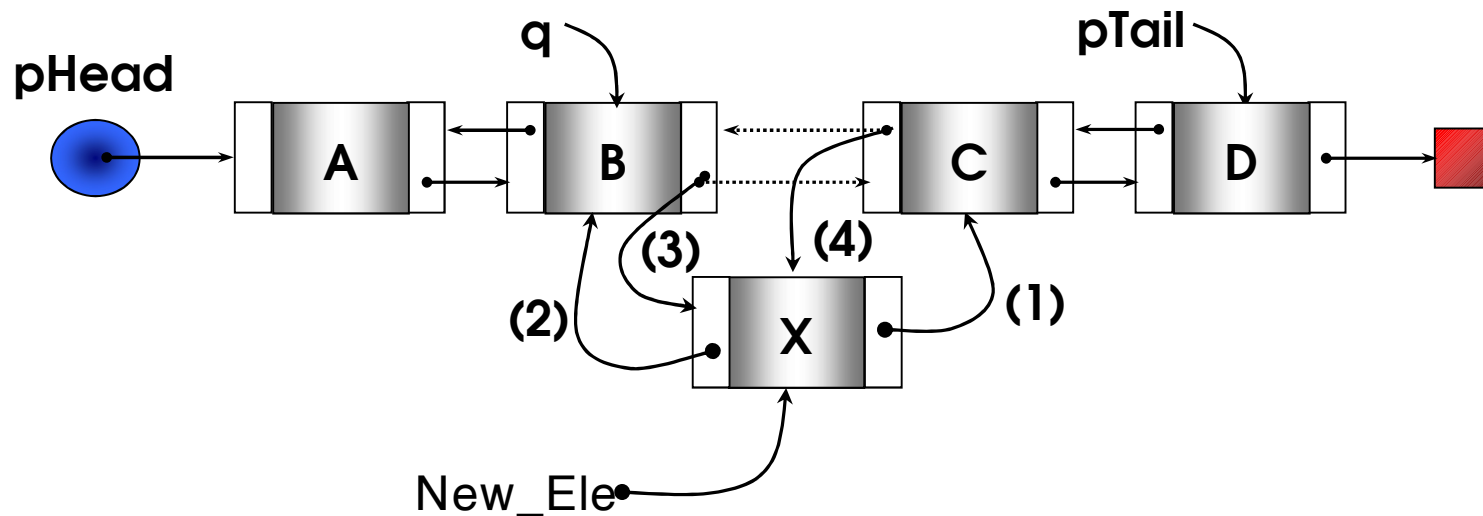


# Chèn một phần tử vào cuối danh sách liên kết kép

```
NODE* InsertTail(DLIST &l, data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL) return NULL;
    if (l.pHead == NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele; // (1)
        new_ele->pPrev = l.pTail; // (2)
        l.pTail = new_ele;      // (3)
    }
    return new_ele;
}
```

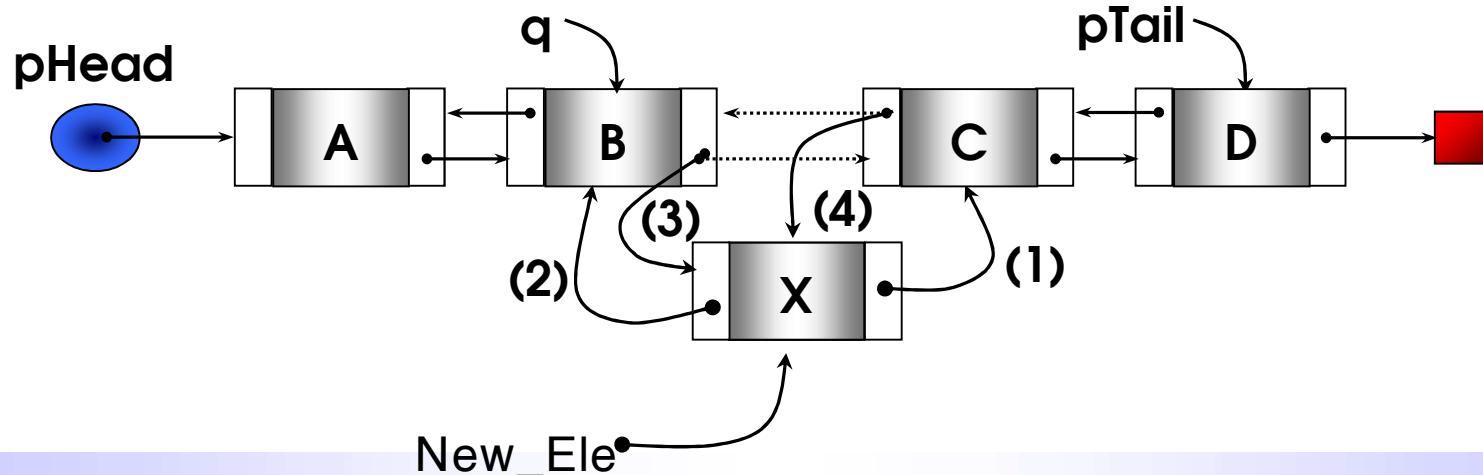


# Chèn vào DSLK kép sau một phần tử $q$



## Chèn vào DSLK kép sau một phần tử q

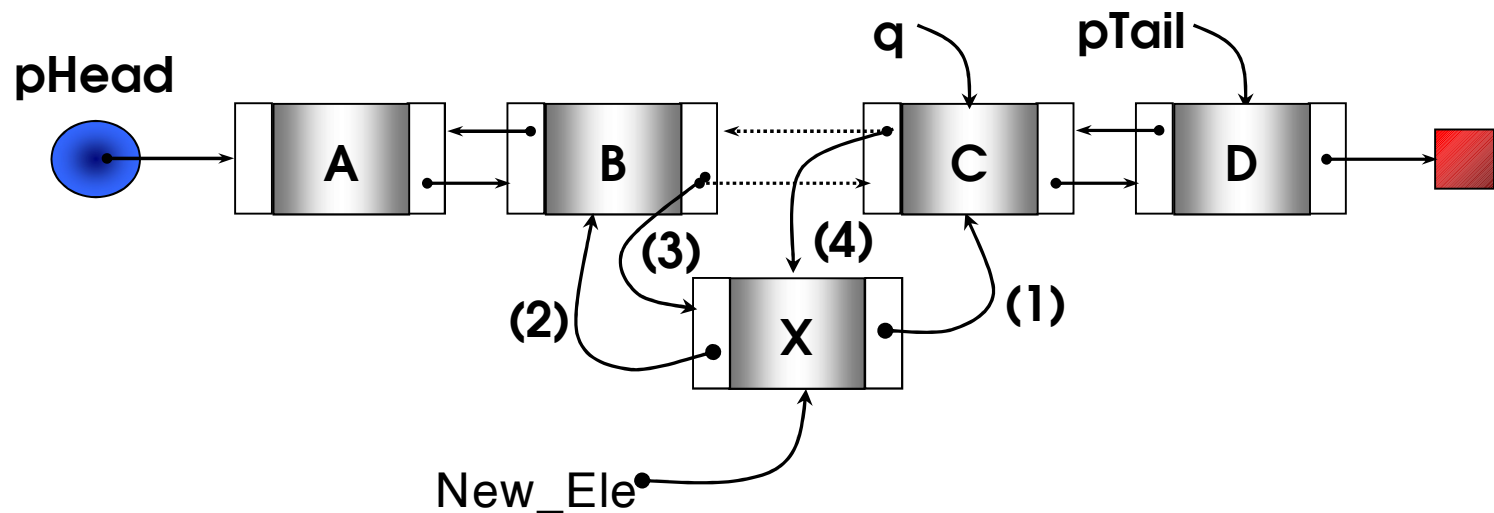
```
void AddAfter(DLIST &l, DNODE* q, DNODE* new_ele)
{
    DNODE* p = q->pNext;
    if ( q!=NULL)
    {
        new_ele->pNext = p;           //(1)
        new_ele->pPrev = q;           //(2)
        q->pNext = new_ele;           //(3)
        if(p != NULL) p->pPrev = new_ele; //(4)
        if(q == l.pTail) l.pTail = new_ele; // p == NULL
    }
    else //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}
```



# Chèn vào DSLK kép sau một phần tử q

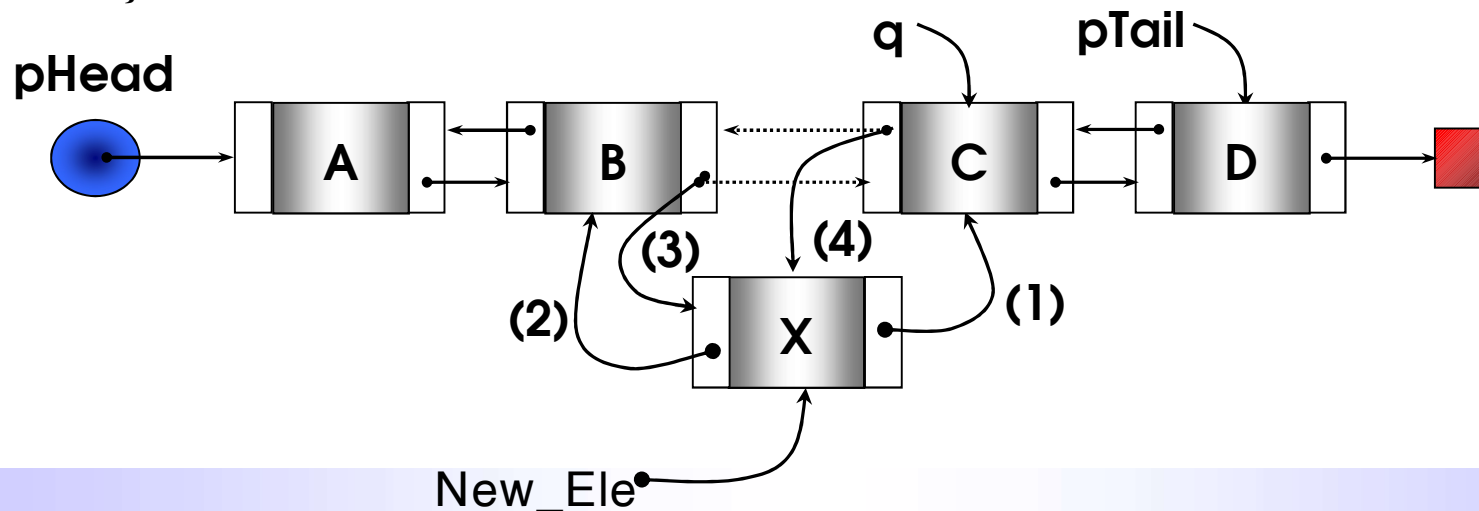
```
void InsertAfter(DLIST &l, DNODE *q, data x)
{ DNODE* p = q->pNext;
  NODE* new_ele = GetNode(x);
  if (new_ele == NULL) return NULL;
  if (q != NULL)
  { new_ele->pNext = p;           //(1)
    new_ele->pPrev = q;          //(2)
    q->pNext = new_ele;          //(3)
    if (p != NULL) p->pPrev = new_ele;  //(4)
    if (q == l.pTail) l.pTail = new_ele;
  }
  else //chèn vào đầu danh sách
    AddFirst(l, new_ele);
}
```

# Chèn vào DSLK kép trước một phần tử q



## Chèn vào DSLK kép trước một phần tử q

```
void AddBefore(DLIST &l, DNODE q, DNODE* new_ele)
{
    DNODE* p = q->pPrev;
    if ( q!=NULL)
    {
        new_ele->pNext = q;           //(1)
        new_ele->pPrev = p;           //(2)
        q->pPrev = new_ele;           //(3)
        if(p != NULL) p->pNext = new_ele; //(4)
        if(q == l.pHead) l.pHead = new_ele;
    }
    else //chèn vào đầu danh sách
        AddTail(l, new_ele);
}
```



# Chèn vào DSLK kép trước một phần tử q

```
void InsertBefore(DLIST &l, DNODE q, data x)
{ DNODE* p = q->pPrev;
  NODE* new_ele = GetNode(x);
  if (new_ele == NULL) return NULL;
  if (q != NULL)
  { new_ele->pNext = q;           //(1)
    new_ele->pPrev = p;          //(2)
    q->pPrev = new_ele;          //(3)
    if(p != NULL) p->pNext = new_ele; //(4)
    if(q == l.pHead) l.pHead = new_ele;
  }
  else //chèn vào đầu danh sách
    AddTail(l, new_ele);
}
```





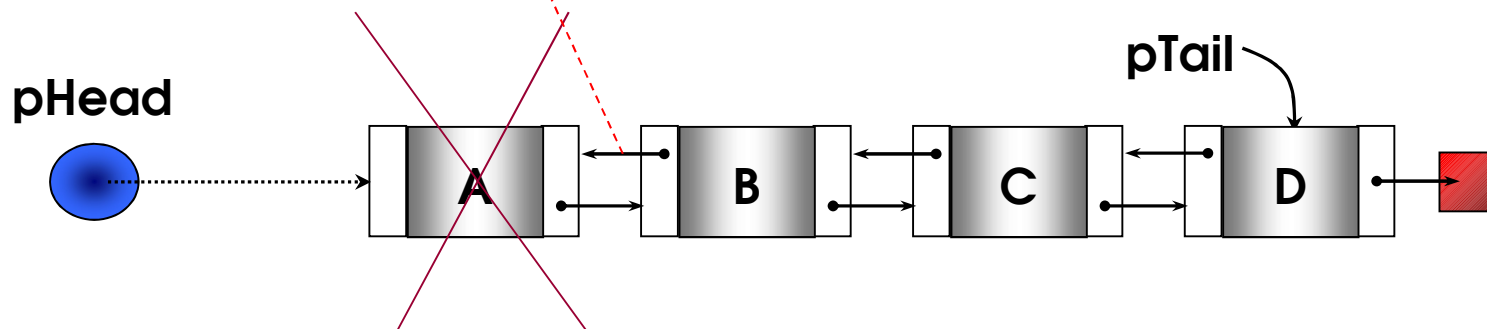
# Hủy một phần tử khỏi danh sách liên kết kép

- Có 5 loại thao tác thông dụng hủy một phần tử ra khỏi danh sách liên kết kép:
  - ☐ Hủy phần tử đầu danh sách
  - ☐ Hủy phần tử cuối danh sách
  - ☐ Hủy một phần tử đứng sau phần tử  $q$
  - ☐ Hủy một phần tử đứng trước phần tử  $q$
  - ☐ Hủy 1 phần tử có khóa  $k$

## Danh sách liên kết kép - Hủy phần tử đầu

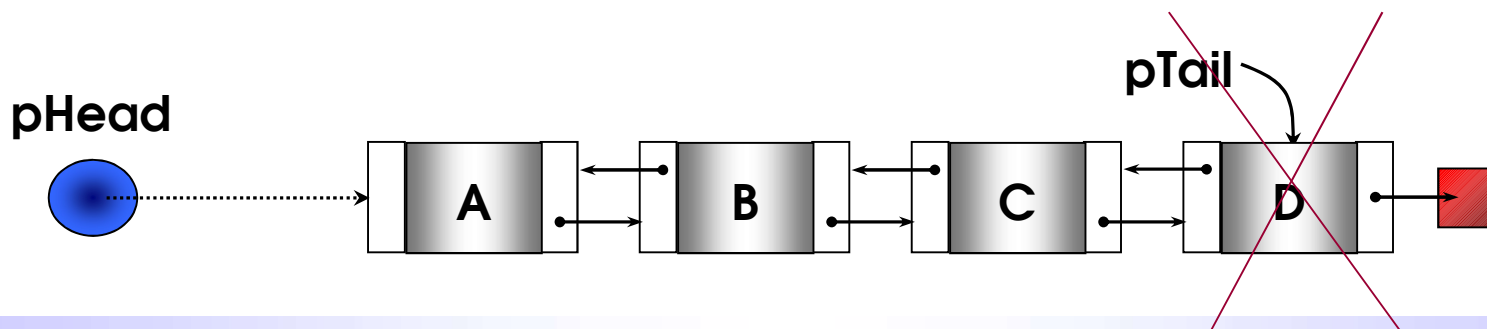
**Data RemoveHead(DLIST &l)**

```
{ DNODE    *p;  
  Data      x = NULLDATA;  
  if ( l.pHead != NULL )  
  { p = l.pHead; x = p->Info;  
    l.pHead = l.pHead->pNext;  
    delete p;  
    if(l.pHead == NULL) l.pTail = NULL;  
    else  
      l.pHead->pPrev = NULL;  
  }  
  return x;  
}
```



## Danh sách liên kết kép - Hủy phần tử cuối danh sách

```
Data RemoveTail(DLIST &l)
{
    DNODE    *p;
    Data      x = NULLDATA;
    if ( l.pTail != NULL )
    {
        p = l.pTail; x = p->Info;
        l.pTail = l.pTail->pPrev;
        l.pTail->pNext = NULL;
        delete p;
        if(l.pTail == NULL)
            l.pHead = NULL;
    }
    return x;
}
```



# Danh sách liên kết kép

## Hủy một phần tử đứng sau phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{ DNODE *p;
  if ( q != NULL)
  { p = q ->pNext ;
    if ( p != NULL)
    { q->pNext = p->pNext;
      if(p == l.pTail) l.pTail = q;
      else p->pNext->pPrev = q;
      delete p;
    }
  }
  else
    RemoveHead(l) ;
}
```

# Danh sách liên kết kép

## Hủy một phần tử đứng trước phần tử q

```
void RemoveBefore (DLIST &l, DNODE *q)
{ DNODE *p;
  if ( q != NULL)
  { p = q ->pPrev;
    if ( p != NULL)
    { q->pPrev = p->pPrev;
      if(p == l.pHead) l.pHead = q;
      else p->pPrev->pNext = q;
      delete p;
    }
  }
  else
    RemoveTail (l) ;
}
```

# Danh sách liên kết kép

## Hủy một phần tử có khóa k (1/3)

```
int RemoveNode(DLIST &l, Data k)
{ DNODE    *p = l.pHead;
  NODE      *q;
  while( p != NULL)
  {  if(p->Info == k) break;
     p = p->pNext;
  }
  .....
}
```

RemoveNode (2/3)



# Danh sách liên kết kép

## Hủy một phần tử có khóa k (2/3)

```
int RemoveNode (DLIST &l, Data k)
{
    .....
    if (p == NULL) return 0; //Không tìm thấy k
    q = p->pPrev;
    if ( q != NULL)
    { p = q ->pNext ;
      if ( p != NULL)
      { q->pNext = p->pNext;
        if (p == l.pTail) l.pTail = q;
        else p->pNext->pPrev = q;
      }
    }
    ..... RemoveNode (3/3)
}
```

# Danh sách liên kết kép

## Hủy một phần tử có khóa k (3/3)

```
int RemoveNode(DLIST &l, Data k)
{ .....
  else //p là phần tử đầu xâu
  {   l.pHead = p->pNext;
      if(l.pHead == NULL) l.pTail = NULL;
      else                l.pHead->pPrev = NULL;
  }
  delete p;
  return 1;
}
```



# Danh sách liên kết kép

DSLK kép về mặt cơ bản có tính chất giống như DSLK đơn.

Tuy nhiên nó có một số tính chất khác xDSLK đơn như sau:

- DSLK kép có mỗi liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên DSLK đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối DSLK kép, còn trên DSLK đơn thao tác này tốn chi phí  $O(n)$ .
- Bù lại, DSLK kép tốn chi phí gấp đôi so với DSLK đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.