# CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

## CHƯƠNG 3:CẤU TRÚC DỮ LIỆU ĐỘNG

# Nội dung

- 3.1. Kiểu dữ liệu con trỏ
- 3.2. Giới thiệu danh sách liên kết
- 3.3. Danh sách liên kết đơn
- 3.4. Danh sách liên kết kép
- 3.5. Danh sách liên kết vòng

# Giới thiệu đối tượng dữ liệu con trỏ

- **Cấu trúc dữ liệu tĩnh và cấu trúc dữ liệu động**
- Với kiểu dữ liệu tĩnh, đối tượng dữ liệu được định nghĩa đệ quy, và tổng kích thước vùng nhớ dành cho tất cả các biến dữ liệu tĩnh chỉ là 64Kb (1 segment bộ nhớ). Do đó, khi có nhu cầu dùng nhiều bộ nhớ hơn ta phải sử dụng các cấu trúc dữ liệu động.
- Nhằm đáp ứng nhu cầu thể hiện xác thực bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, ta cần phải tìm cách tổ chức kết hợp dữ liệu với những hình thức linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống.
- Các hình thức tổ chức dữ liệu như vậy được gọi là cấu trúc dữ liệu động. Cấu trúc dữ liệu động cơ bản nhất là danh sách liên kết.

# Giới thiệu đối tượng dữ liệu con trỏ

- **3.1.2 Kiểu con trỏ**
- **1. Biến tĩnh**
  - Biến không động (biến tĩnh) là những biến thỏa các tính chất sau:
  - - Được khai báo tường minh.
  - - Tồn tại khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này.
  - - Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc là Stack (đối với các biến nửa tĩnh, các biến cục bộ).
  - - Kích thước không thay đổi trong suốt quá trình sống.

# Giới thiệu đối tượng dữ liệu con trỏ

- **2. Kiểu dữ liệu con trỏ**
  - Khi nói đến kiểu dữ liệu T, ta thường chú ý đến hai đặc trưng quan trọng và liên hệ mật thiết với nhau:
- - Tập V các giá trị thuộc kiểu: đó là tập các giá trị hợp lệ mà đối tượng kiểu T có thể nhận được và lưu trữ.
- - Tập O các phép toán (hay thao tác xử lý) xác định có thể thực hiện trên các đối tượng dữ liệu kiểu đó.
- Kí hiệu: T = <V, O>

# Giới thiệu đối tượng dữ liệu con trỏ

**3. Định nghĩa kiểu dữ liệu con trỏ**

Cho trước kiểu T=<V,O>

Kiểu con trỏ - kí hiệu "Tp" – chỉ đến các phần tử có kiểu "T" được định nghĩa:

Tp=<Vp, Op>, trong đó:

- Vp={{các địa chỉ có thể lưu trữ những đối tượng có kiểu T}, NULL} (với NULL là

một giá trị đặc biệt tượng trưng cho một giá trị không biết hoặc không quan tâm).

- Op = {các thao tác định địa chỉ của một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó} (thường gồm các thao tác tạo một con trỏ chỉ đến một đối tượng thuộc kiểu T; hủy một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó)

# Giới thiệu đối tượng dữ liệu con trỏ

- Kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng dữ liệu khác.
- Biến thuộc kiểu con trỏ Tp là biến mà giá trị của nó là địa chỉ của một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.
- Kích thước của biến con trỏ tùy thuộc vào quy ước số byte địa chỉ trong từng mô hình bộ nhớ của từng ngôn ngữ lập trình cụ thể. Chẳng hạn biến con trỏ trong C++ trên môi trường Windows có kích thước 4 bytes.

# Giới thiệu đối tượng dữ liệu con trỏ

- Cú pháp định nghĩa kiểu con trỏ trong ngôn ngữ C, C++:
  typedef <kiểu cơ sở> *<kiểu con trỏ>;
  Ví dụ: typedef int *intpointer; //Kiểu con trỏ
  intpointer p; //Biến con trỏ
- Cú pháp định nghĩa trực tiếp một biến con trỏ trong ngôn ngữ C, C++:
  <kiểu cơ sở> *<tên biến>;
  Ví dụ: int *p;
- Các thao tác cơ bản trên kiểu con trỏ (minh họa bằng C++)
  Khi một biến con trỏ p lưu địa chỉ của đối tượng x, ta nói "p trỏ đến x".
  - Gán địa chỉ của một vùng nhớ con trỏ p:
  p = <địa chỉ>;
  p = <địa chỉ>+<giá trị nguyên>;
  - Truy xuất nội dung của đối tượng do p trỏ đến: *p

# Giới thiệu đối tượng dữ liệu con trỏ

## 4. Biến động

Trong nhiều trường hợp, tại thời điểm biên dịch không thể xác định trước kích thước chính xác của một số đối tượng dữ liệu do sự tồn tại và tăng trưởng của chúng phụ thuộc vào ngữ cảnh của việc thực hiện chương trình.

Các đối tượng có đặc điểm như vậy được khai báo như biến động.

Đặc trưng của biến động

- Biến không được khai báo tường minh.
- Có thể được cấp phát hoặc giải phóng bộ nhớ khi người sử dụng yêu cầu.
- Các biến này không theo qui tắc phạm vi (tĩnh).
- Vùng nhớ của biến được cấp phát trong Heap.
- Kích thước có thể thay đổi trong quá trình sống.

# Giới thiệu đối tượng dữ liệu con trỏ

- Do không được khai báo tường minh nên các **biến động không có một định danh** được liên kết với địa chỉ vùng nhớ cấp phát cho nó, do đó khó truy xuất đến một biến động.
- Để giải quyết vấn đề này, **phải dùng một con trỏ (là biến không động)** để trỏ đến biến động. Khi tạo ra một biến động, phải dùng một **con trỏ** để **lưu địa chỉ** của biến này thông qua biến con trỏ đã biết định danh.
- Thao tác cơ bản trên biến động do biến con trỏ "p" trỏ đến:
  - Tạo ra một biến động và cho con trỏ "p" trỏ đến nó:
  p = new KieuCoSo; (dùng hàm new để cấp phát bộ nhớ)
  - Hủy vùng nhớ cấp phát bởi hàm new do p trỏ tới: delete(p);
  - Cấp phát bộ nhớ cho mảng động: KieuCoSo *p;
  p = new KieuCoSo[Max];/Max là giá trị nguyên dương
  - Thu hồi vùng nhớ của mảng động: delete([]p);

# Giới thiệu đối tượng dữ liệu con trỏ

- Do không được khai báo tường minh nên các **biến động không có một định danh** được liên kết với địa chỉ vùng nhớ cấp phát cho nó, do đó khó truy xuất đến một biến động.
- Để giải quyết vấn đề này, **phải dùng một con trỏ (là biến không động)** để trỏ đến biến động. Khi tạo ra một biến động, phải dùng một **con trỏ** để **lưu địa chỉ** của biến này thông qua biến con trỏ đã biết định danh.
- Thao tác cơ bản trên biến động do biến con trỏ "p" trỏ đến:
  - Tạo ra một biến động và cho con trỏ "p" chỉ đến nó:
  p = new KieuCoSo; (dùng hàm new để cấp phát bộ nhớ)
  - Hủy vùng nhớ cấp phát bởi hàm new do p trỏ tới: delete(p);
  - Cấp phát bộ nhớ cho mảng động: KieuCoSo *p;
  p = new KieuCoSo[Max];/Max là giá trị nguyên dương
  - Thu hồi vùng nhớ của mảng động: delete([]p);

# DANH SÁCH LIÊN KẾT

- **3.2.1 Định nghĩa**

  Cho T là một kiểu được định nghĩa trước, kiểu danh sách Tx gồm các phần tử thuộc kiểu T được định nghĩa là:

  Tx = <Vx, Ox>

  Trong đó:

  - Vx = {tập hợp có thứ tự các phần tử kiểu T được mócnối với nhau theo trình tự tuyến tính};
  - Ox = {các thao tác trên danh sách liên kết như: tạo; tìm kiếm; chèn; xóa; sắp …}

- Ví dụ: Quản lý Hồ sơ học sinh được tổ chức thành danh sách gồm nhiều hồ sơ của từng học sinh, số lượng học sinh có thể thay đổi do vậy cần có các thao tác thêm, hủy một hồ sơ. Để phục vụ cho công tác giáo vụ cần thực hiện các thao tác tìm hồ sơ, in danh sách hồ sơ…xếp danh sách…

# DANH SÁCH LIÊN KẾT

- **3.2.2 Các hạn chế của mảng**

  Xét khai báo: int A[Nmax];

  - Số phần tử của mảng không thể vượt quá Nmax
  - Luôn phải cấp cho A một vùng nhớ Nmax*2B dù cho số phần tử thực tế N rất nhỏ.
  - Các phần tử phải nằm liên tiếp trong bộ nhớ
  - => Không linh hoạt mềm dẻo và lãng phí bộ nhớ, phải có 1 vùng nhớ trống liên tiếp đủ lớn cho mảng.

  Các thao tác chèn, xóa thực hiện khó khăn

  Để khắc phục các hạn chế đó, người ta đề xuất một kiểu dữ liệu mới gọi là DANH SÁCH LIÊN KẾT

# DANH SÁCH LIÊN KẾT

- **Tổ chức danh sách theo cách cấp phát liên kết.**

Một danh sách liên kết bao gồm tập các phần tử (nút), mỗi nút là một cấu trúc chứa hai thông tin:

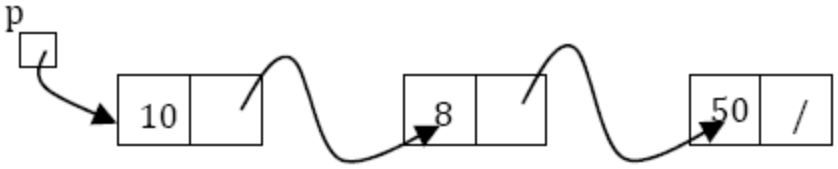
- Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử.
- Thành phần mối liên kết: **lưu trữ địa chỉ** của **phần tử kế tiếp** trong danh sách, hoặc lưu trữ giá trị **NULL** nếu là **phần tử cuối** danh sách.

Mỗi nút như trên có thể được cài đặt như sau:


```
struct tagNode
{ Data Info;          //thành phần dữ liệu
  tagNode* pNext; //thành phần mối liên kết (tự trỏ)
};
//Đổi lại tên kiểu phần tử trong danh sách
typedef  tagNode NODE;
```

# DANH SÁCH LIÊN KẾT

- Mỗi phần tử trong danh sách đơn là một biến động, sẽ được yêu cầu cấp phát bộ nhớ khi cần.
- Danh sách liên kết đơn chính là sự liên kết các biến động này với nhau, do vậy đạt được sự linh động khi thay đổi số lượng các phần tử.
- Ví dụ:

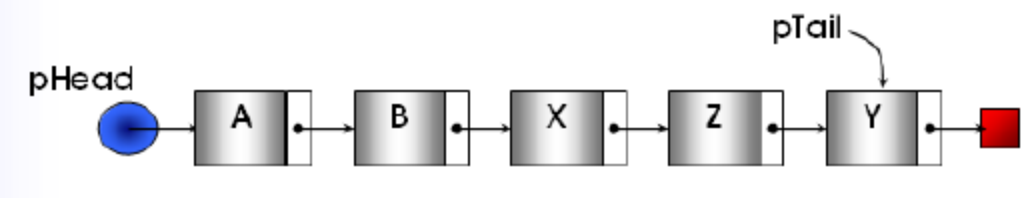



Để quản lý một danh sách liên kết đơn chỉ cần biết địa chỉ phần tử đầu danh sách, từ phần tử đầu danh sách ta có thể đi đến các nút tiếp theo trong danh sách liên kết nhờ vào thành phần địa chỉ của nút.
Con trỏ pHead được dùng để lưu trữ địa chỉ của phần tử ở đầu danh sách, ta gọi Head là đầu của danh sách. Ta có khai báo:
- NODE* pHead;

# DANH SÁCH LIÊN KẾT

Để tiện lợi, ta có thể sử dụng thêm một con trỏ pTail để giữ địa chỉ phần tử cuối danh sách. Khai báo pTail như sau:

- NODE* pTail;




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

```
struct SV                    //Data
{ char Ten[30]; int MaSV; };
//Kiểu phần tử trong DS: SinhvienNode
struct SinhvienNode
{ SV Info; SinhvienNode* pNext; };
-//Đổi lại tên kiểu phần tử trong DS: SVNode
        typedef SinhvienNode SVNode;
```

# DANH SÁCH LIÊN KẾT

- //Định nghĩa danh sách liên kết: LIST

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

- **Các thao tác cơ bản trên danh sách liên kết đơn**
  - Tạo một phần tử cho danh sách liên kết với thông tin x.
  - Khởi tạo danh sách rỗng.
  - Kiểm tra danh sách rỗng.
  - Duyệt danh sách.
  - Tìm một phần tử trong danh sách.
  - Chèn một phần tử vào danh sách
  - Hủy một phần tử khỏi danh sách
  - Sắp xếp danh sách

# DANH SÁCH LIÊN KẾT

**1. Tạo một phần tử cho danh sách liên kết với thông tin x.**

Hàm CreateNode(x): tạo một nút trong DSLK với thành phần dữ liệu x, hàm trả về con trỏ lưu trữ địa chỉ của phần tử vừa tạo (nếu thành công):

Khai báo con trỏ p là một NODE;

Cấp phát bộ nhớ cho p bằng hàm new

Gán thông tin cho p: p->Info=x;

Thành phần con trỏ của p chưa được trỏ: p->pNext=NULL;

- Sử dụng hàm này: Giả sử new_ele là một phần tử mới:

NODE *new_ele = CreateNode(x);

**2. Khởi tạo danh sách rỗng L**

```
void CreatList(LIST &L)
{  L.pHead = L.pTail = NULL; }
```

# DANH SÁCH LIÊN KẾT

**3. Kiểm tra danh sách rỗng**

Hàm IsEmpty(L) = 1 nếu danh sách rỗng, bằng 0 nếu không rỗng

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

**4. Duyệt danh sách**

Duyệt danh sách thường sử dụng trong các thao tác như:

- • Đếm các phần tử của danh sách.
- In danh sách
- Tìm kiếm một phần tử

# DANH SÁCH LIÊN KẾT

**4. Duyệt danh sách**

Thuật toán có thể mô tả như sau:

Bước 1: p = Head;  *//Cho p trỏ đến phần tử đầu danh sách*

Bước 2: Trong khi (Danh sách chưa hết) thực hiện

Xử lý phần tử p;

p=p->pNext;  *// Cho p trỏ tới phần từ kế tiếp*

```
void ProcessList (LIST L)
    {  NODE *p;
        p = L.pHead;
        while (p!= NULL) //Chưa hết DS
            {
              ProcessNode(p); // xử lý cụ thể phụ thuộc từng ứng dụng
               p = p->pNext;
             }
     }
```

# DANH SÁCH LIÊN KẾT

**5. Tìm phần tử có thành phần dữ liệu x trong danh sách**

Mô tả

Bước 1: p = Head;  //Cho p trỏ đến phần tử đầu danh sách

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

p = p->Next;  // Cho p trỏ tới phần tử kế tiếp

Bước 3:

Nếu p != NULL thì p trỏ tới phần tử cần tìm

Ngược lại: không có phần tử cần tìm.

# DANH SÁCH LIÊN KẾT

**6. Thao tác chèn một phần tử vào danh sách**

Có ba vị trí để có thể chèn một phần tử new_ele vào danh sách:

Chèn phần tử vào đầu danh sách

Chèn phần tử vào cuối danh sách

Chèn phần tử vào danh sách sau một phần tử q.

*a. Chèn phần tử vào đầu danh sách*

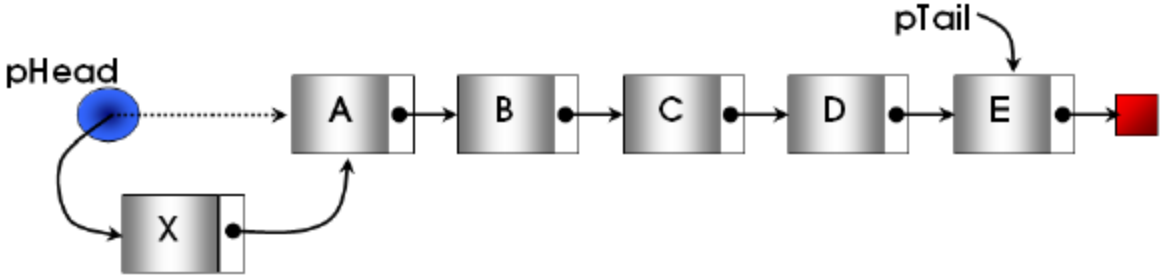
Nếu danh sách rỗng thì nút này vừa là nút đầu vừa là nút cuối: Cho cả 2 con trỏ trỏ vào nút đó: Head = new_ele; Tail = Head;

Ngược lại:

Kết nối phần tử mới vào đầu DS: new_ele ->pNext = Head;

Chuyển đầu DS sang nút mới: Head = new_ele;

Minh họa



# DANH SÁCH LIÊN KẾT

**6. Thao tác chèn một phần tử vào danh sách**

*b. Chèn một phần tử vào cuối danh sách*

Mô tả:

Nếu danh sách rỗng thì

Head = new_ele;

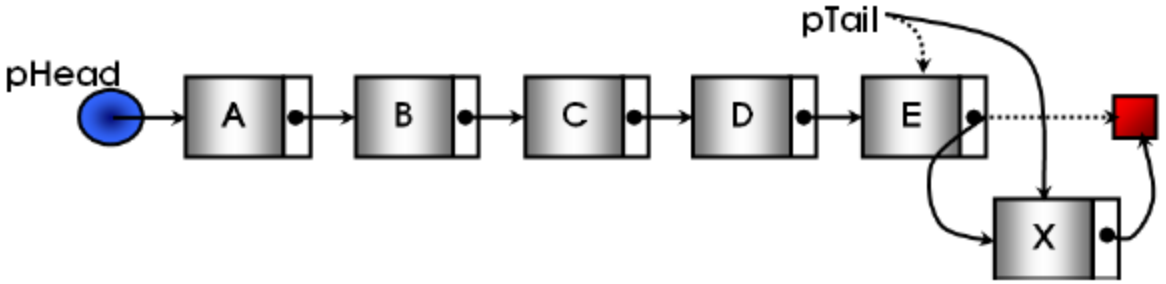
Tail = Head;

Ngược lại

Tail ->Next = new_ele;

Tail = new_ele ;

Minh họa



# DANH SÁCH LIÊN KẾT

## 6. Thao tác chèn một phần tử vào danh sách

*c. Chèn một phần tử x vào danh sách sau một phần tử q*

Mô tả

Nếu ( q != NULL) thì

new_ele -> pNext = q->pNext;

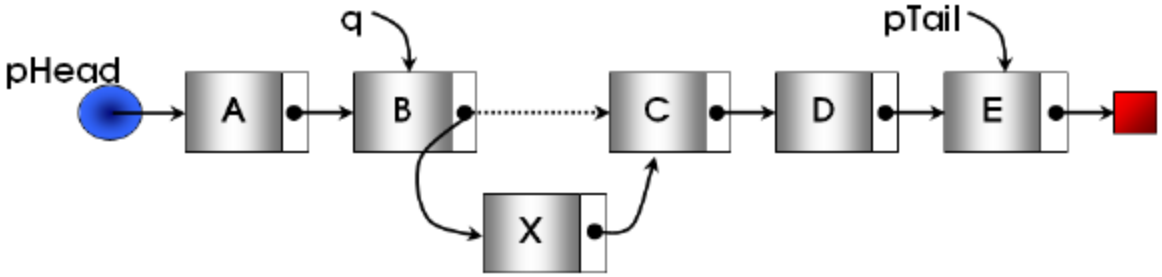
q->pNext = new_ele;

Nếu q = l.Tail thì l.Tail = new_ele

Ngược lại

Chèn phần tử x vào đầu danh sách

Minh họa



# DANH SÁCH LIÊN KẾT

## 6. Thao tác chèn một phần tử vào danh sách

*c. Chèn một phần tử x vào danh sách sau một phần tử q*

Mô tả

Nếu ( q != NULL) thì

new_ele -> pNext = q->pNext;

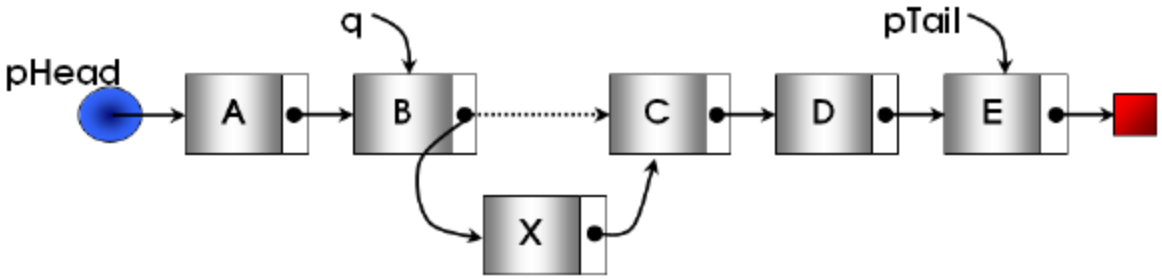
q->pNext = new_ele;

Nếu q = l.Tail thì l.Tail = new_ele

Ngược lại

Chèn phần tử x vào đầu danh sách

Minh họa



# DANH SÁCH LIÊN KẾT

- **7. Hủy một phần tử khỏi danh sách liên kết**

  Có ba thao tác thông dụng khi hủy một phần tử ra khỏi danh sách:

  • Hủy phần tử đầu danh sách

  • Hủy một phần tử đứng sau phần tử q trong danh sách

  • Hủy một phần tử có dữ liệu x.

  *a. Hủy phần tử đầu danh sách*

  Mô tả: Nếu danh sách khác rỗng

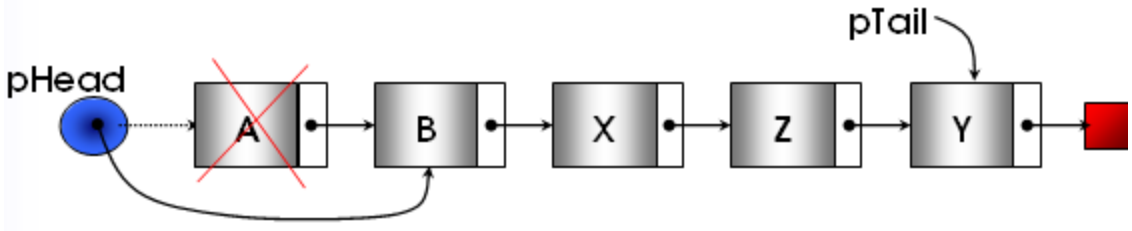
  p = Head;  *// p là phần tử cần hủy*

  Head = Head->pNext; *// tách p ra khỏi xâu*

  free(p); *// hủy biến động do p trỏ đến*

  Nếu Head=NULL thì Tail = NULL; *//Xâu rỗng*

  Minh họa



# DANH SÁCH LIÊN KẾT

- **7. Hủy một phần tử khỏi danh sách liên kết**

*b. Hủy một phần tử đứng sau phần tử q trong danh sách*

Mô tả

Nếu (q!= NULL) thì

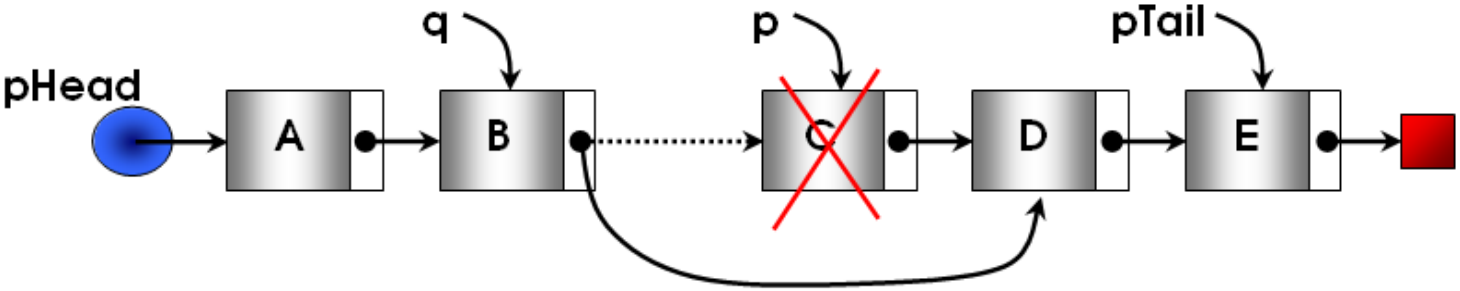
p = q->Next; *// p là phần tử cần hủy*

Nếu (p != NULL) thì *// q không phải là cuối xâu*

q->Next = p->Next; *// tách p ra khỏi xâu*

free(p); *// Hủy biến động do p trỏ đến*

Minh họa



# DANH SÁCH LIÊN KẾT

- **7. Hủy một phần tử khỏi danh sách liên kết**

  *c. Hủy một phần tử có dữ liệu x*

  Mô tả

  Bước 1: tìm phần tử p có dữ liệu x và phần tử q đứng trước nó

  Bước 2:

  Nếu (p!=NULL) thì //tìm thấy x

  Hủy p ra khỏi xâu như hủy phần tử sau q;

  Ngược lại

  Báo không có phần tử có dữ liệu x.

# DANH SÁCH LIÊN KẾT

## 8. Sắp xếp danh sách

Có hai cách tiếp cận sắp xếp trên danh sách liên kết:

- 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 trường Next)

- Phương án 1: hoán vị nội dung các phần tử trong danh sách liên kết

  Ta có thể áp dụng một thuật giải sắp xếp đã biết trên mảng, chẳng hạn thuật toán chọn trực tiếp. Điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên xâu thông qua liên kết thay vì chỉ số như trên mảng.

# DANH SÁCH LIÊN KẾT

## 8. Sắp xếp danh sách

- Phương án 2: thay đổi 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, do đó chỉ thao tác trên các móc nối (pNext).

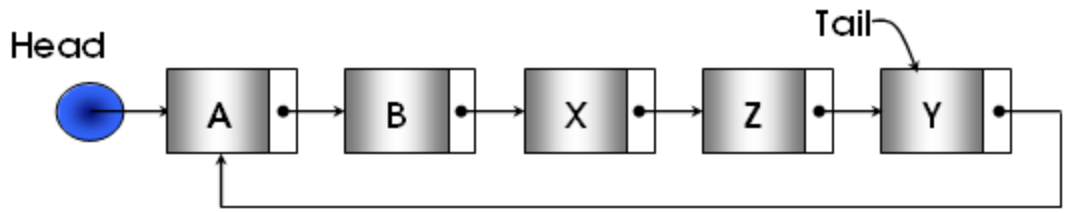
  Tuy nhiên, thao tác trên các móc nối thường phức tạp hơn thao tác trực tiếp trên dữ liệu, do đó cần cân nhắc khi chọn cách tiếp cận. Nếu dữ liệu không quá lớn thì nên chọn phương án 1.

  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.

# DANH SÁCH LIÊN KẾT ĐƠN NỐI VÒNG

## 1. Định nghĩa

- Danh sách liên kết vòng (xâu vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL thì nó trỏ tới phần tử đầu danh sách.
- Đối với danh sách vòng, ta có thể xuất phát từ một phần từ bất kì để duyệt toàn bộ danh sách.
- Biểu diễn danh sách



# DANH SÁCH LIÊN KẾT ĐƠN NỐI VÒNG

**2. Các thao tác trên danh sách liên kết vòng (biểu diễn bằng DSLK đơn)**

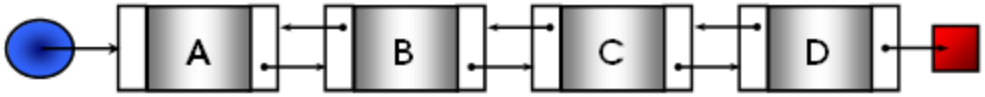
- Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kì trên danh sách xem như phần tử đầu danh sách để kiểm tra việc duyệt đã hết phần tử của danh sách hay chưa.
- VD: Duyệt DS – Tìm kiếm

```
NODE* Search(LIST L, Data x)
  { NODE *p;
  p = L.pHead;
  do   { if ( p->Info == x) return p;   p = p->pNext;
  } while (p != L.pHead); // chưa đi hết vòng
  return NULL;                  //Khong co
  }
```

# DANH SÁCH LIÊN KẾT KÉP

## 1. Định nghĩa

- 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.




## 2. Cài đặt:

pPre liên kết với phần từ dứng trước.

pNext liên kết với phần từ dứng sau.

```
struct tagDNode
 {
Data Info;
tagDNode* pPre;
 tagDNode* pNext;
 };
```

# DANH SÁCH LIÊN KẾT KÉP

**2. Cài đặt:**

```
typedef tagDNode DNODE;
struct DLIST
 {
 DNODE* pLeft;   // trỏ đến phần tử cực  trái danh sách
 DNODE* pRight;    // trỏ đến phần tử cực phải danh sách
 };
```

**3. Chèn một phần tử vào DSLK kép**

Có 4 cách chèn một nút new_ele vào danh sách kép:

- Chèn bên trái danh sách
- Chèn bên phải danh sách
- Chèn nút vào sau một phần tử p

# DANH SÁCH LIÊN KẾT KÉP

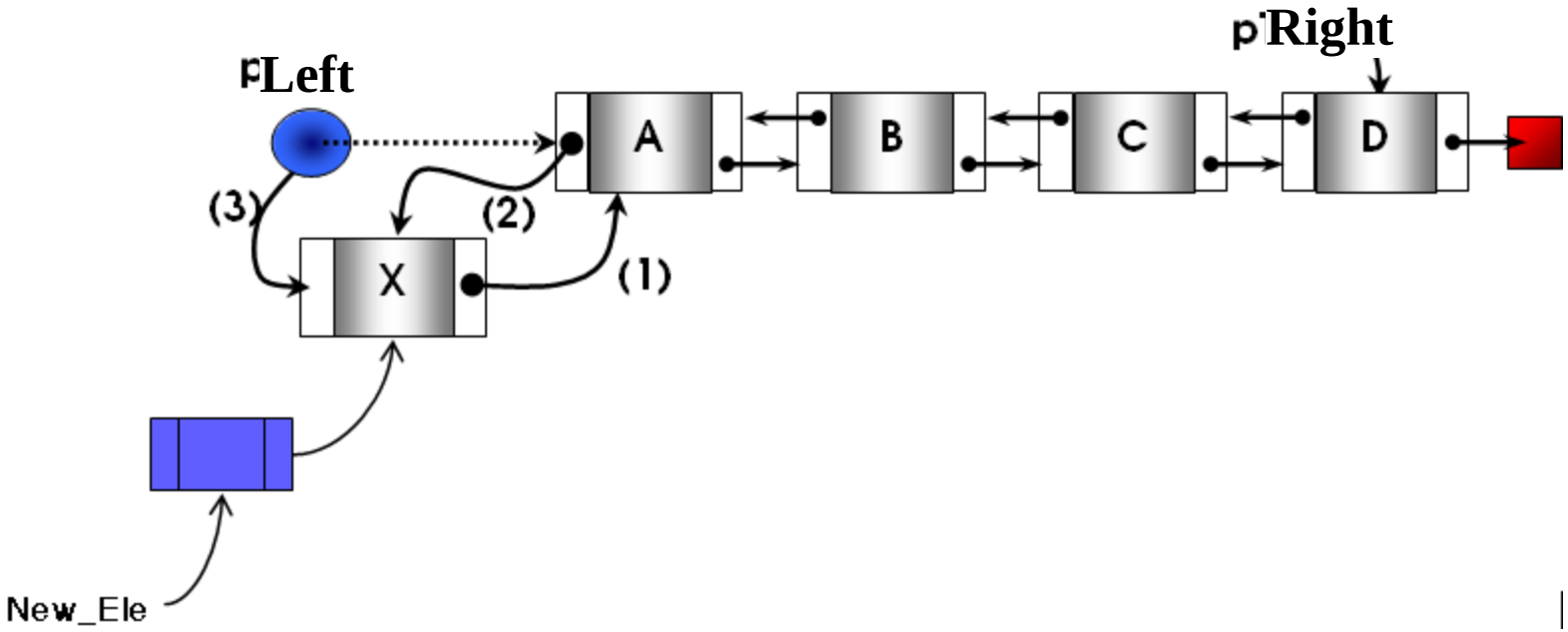
-Chèn bên trái danh sách

-Nếu DS rỗng: pLeft=NULL thì nút mới vừa là nút cực trái, vừa là nút cực phải: pLeft=pRight = new_ele.

-Ngược lại: new_ele->pNext=pLeft;

pLeft->pPrev = new_ele;

pLeft=new_ele;



# DANH SÁCH LIÊN KẾT KÉP

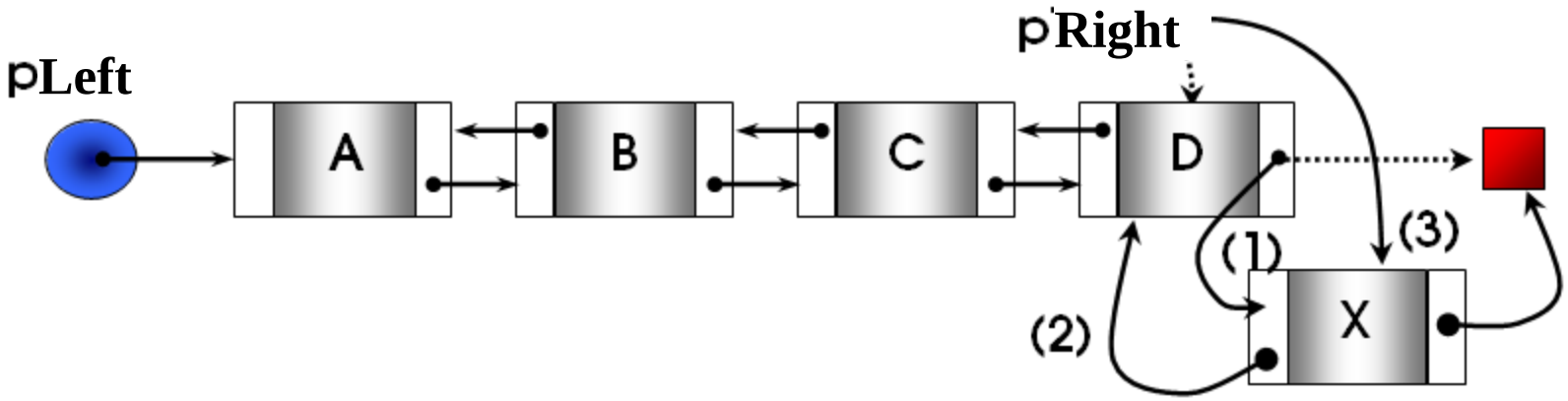
-Chèn bên phải danh sách

-Nếu DS rỗng: Như trên

-Ngược lại: pRight->pNext = new_ele;

new_ele->pPrev=pRight;

pRight=new_ele;



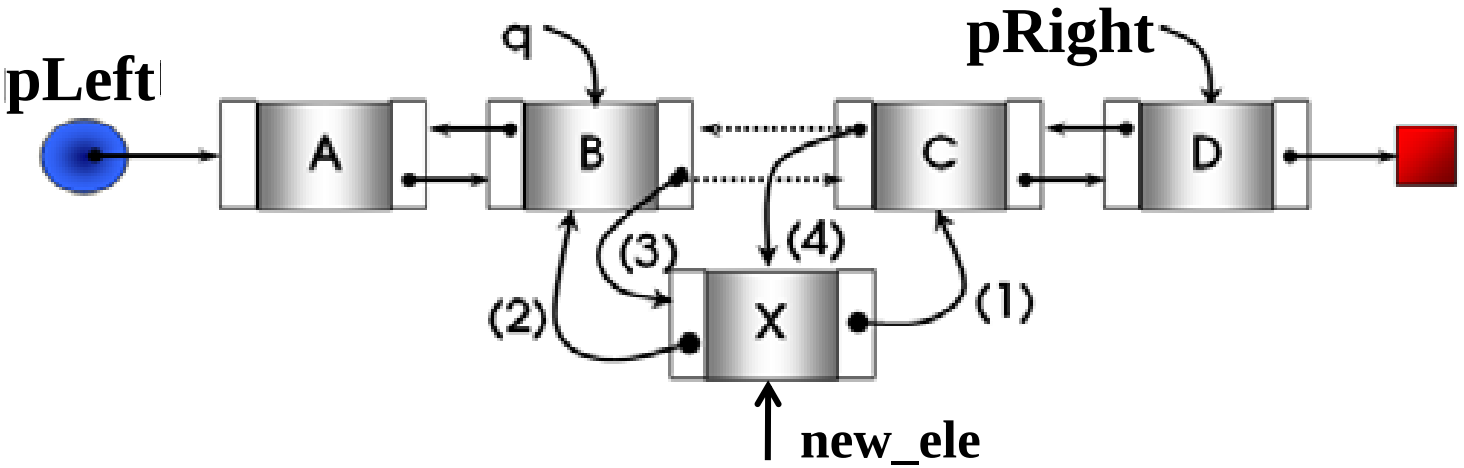
# DANH SÁCH LIÊN KẾT KÉP

-Chèn một phần tử sau một phần tử q trong danh sách: Có nhiều cách móc nối, một trong các cách có thể như sau:

p=q->pNext;

1) new_ele->pNext = q->pNext;
2) new_ele->pPrev = q;
3) q->pNext = new_ele;
4) if(p!=NULL) p->pPrev=new_ele;

Nếu đó là nút cực phải: if(q==pRight) pRight=new_ele;



# DANH SÁCH LIÊN KẾT KÉP

**4. Hủy một phần tử ra khỏi danh sách**

Có 5 thao tác thông dụng để hủy một phần tử ra khỏi danh sách liên kết kép:

- Hủy phần tử cực trái danh sách
- Hủy phần tử cực phải danh sách
- Hủy phần tử sau phần tử q
- Hủy phần tử trước phần tử q

-Hủy phần tử có khóa k

Chúng ta chỉ xét một số t/h, phần còn lại tự nghiên cứu.

# DANH SÁCH LIÊN KẾT KÉP

- **Hủy phần tử cực trái danh sách:**

p= pLeft;
pLeft=pLeft->pNext; //Chuyển pLeft sang bên phải
delete p;

- **Hủy phần tử cực phải danh sách:**

p= pRight;
pRight=pRight->pPrev; //Chuyển pRight sang bên trái
delete p;

- **Hủy một phần tử đứng sau phần tử q:**

Chỉ cần nối vòng các liên kết qua phần tử cần hủy: cho p trỏ vào nút cần hủy
q->pNext = p->pNext;
p->pNext->pPrev = p->pPrev; hoặc
q->pNext->pPrev = p->pPrev;

**pLeft** **pRight**

# Tìm kiếm nhị phân (Binary Search)

- **Lưu ý:**
  - Thuật toán **tìm nhị phân** chỉ có thể vận dụng trong trường hợp **dãy/mảng đã có thứ tự**. Trong trường hợp **tổng quát** chúng ta chỉ có thể áp dụng thuật toán **tìm kiếm tuần tự**.
  - Các thuật toán đệ quy có thể ngắn gọn song tốn kém bộ nhớ để ghi nhận mã lệnh chương trình (mỗi lần gọi đệ quy) do vậy có thể làm cho chương trình chạy chậm lại. Trong thực tế, khi viết chương trình nếu có thể chúng ta **nên sử dụng thuật toán không đệ quy**.

BÀI TẬP