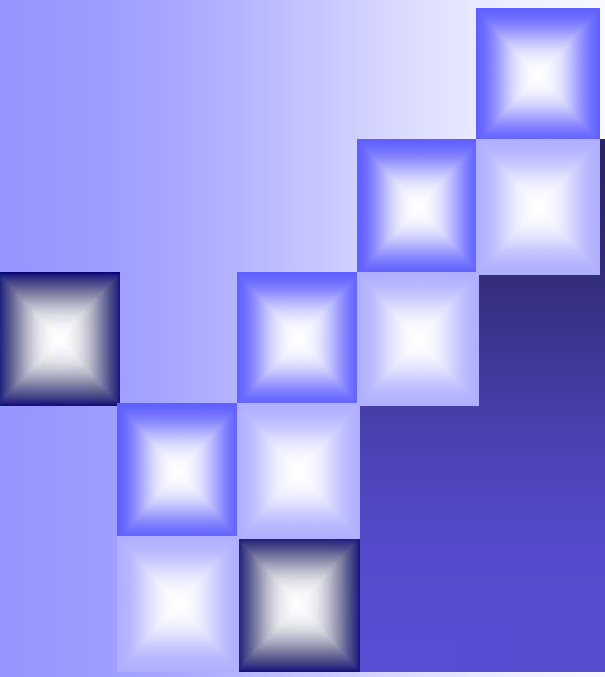


Cấu trúc dữ liệu động

Mục tiêu

- Giới thiệu khái niệm cấu trúc dữ liệu động.
- Giới thiệu danh sách liên kết:
  - Các kiểu tổ chức dữ liệu theo DSLK.
  - Danh sách liên kết đơn: tổ chức, các thuật toán, ứng dụng.

Kiểu dữ liệu tĩnh

- Khái niệm: Một số đối tượng dữ liệu không thay thay đổi được kích thước, cấu trúc, … trong suốt qua trình sống. Các đối tượng dữ liệu thuộc những kiểu dữ liệu gọi là kiểu dữ liệu liệu tĩnh.
- Một số kiểu dữ liệu tĩnh: các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: kiểu thực, kiểu nguyên, kiểu ký tự ... hoặc từ các cấu trúc đơn giản như mẩu tin, tập hợp, mảng ...

➔ Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này thường cứng ngắt, gò bó ➔ khó diễn tả được thực tế vốn sinh động, phong phú.

Ví dụ thực tế

- Mô tả, quản lý một đối tượng ‘con người’ cần thể hiện các thông tin tối thiểu như :
  - Họ tên
  - Số CMND
  - Thông tin về cha, mẹ

Ví dụ thực tế

- Việc biểu diễn một đối tượng có nhiều thành phần thông tin như trên có thể sử dụng kiểu bản ghi. Tuy nhiên, cần lưu ý cha, mẹ của một người cũng là các đối tượng kiểu NGUOI, do vậy về nguyên tắc cần phải có định nghĩa như sau:

```
typedef   struct NGUOI{
  char    Hoten[30];
  int     So_CMND ;
  NGUOI   Cha,Me;
};
```

- Nhưng với khai báo trên, các ngôn ngữ lập trình gặp khó khăn trong việc cài đặt không vượt qua được như xác định kích thước của đối tượng kiểu NGUOI ?

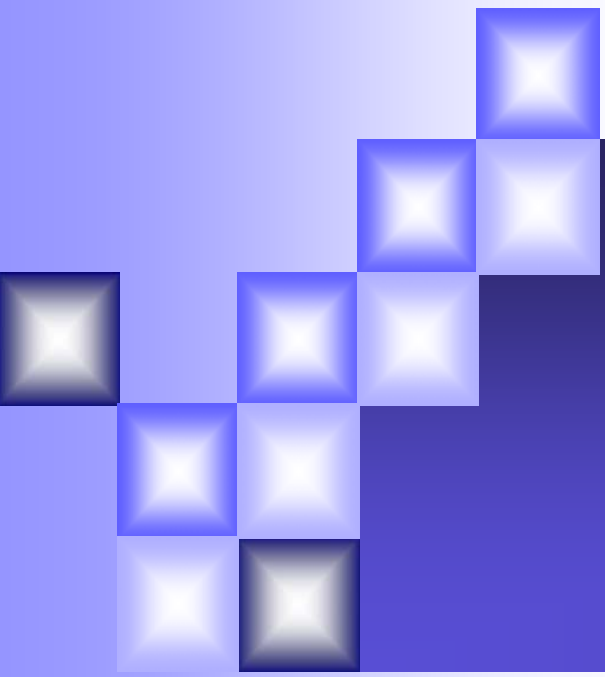
CTDL tĩnh – Một số hạn chế

- Một số đối tượng dữ liệu trong chu kỳ sống của nó có thể thay đổi về cấu trúc, độ lớn, như danh sách các học viên trong một lớp học có thể tăng thêm, giảm đi ... Nếu dùng những cấu trúc dữ liệu tĩnh đã biết như mảng để biểu diễn ➔ Những thao tác phức tạp, kém tự nhiên ➔ chương trình khó đọc, khó bảo trì và nhất là khó có thể sử dụng bộ nhớ một cách có hiệu quả.
- Dữ liệu tĩnh sẽ chiếm vùng nhớ đã dành cho chúng suốt quá trình hoạt động của chương trình ➔ sử dụng bộ nhớ kém hiệu quả.

Hướng giải quyết

- Cần xây dựng cấu trúc dữ liệu đáp ứng được các yêu cầu:
  - Linh động hơn.
  - Có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống.

➔ *Cấu trúc dữ liệu động.*



Kiểu dữ liệu Con trỏ

Biến không động

Biến không động (biến tĩnh, biến nửa tĩnh) là những biến thỏa:

- 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 biến nửa tĩnh - các biến cục bộ).
- Kích thước không thay đổi trong suốt quá trình sống.
- Do được khai báo tường minh, các biến không động có một định danh đã được kết nối với địa chỉ vùng nhớ lưu trữ biến và được truy xuất trực tiếp thông qua định danh đó.
- Ví dụ :

```
int   a;    // a, b là các biến không động
char b[10];
```

Kiểu dữ liệu Con trỏ

- Cho trước kiểu dữ liệu T = <V, O>.
- Kiểu con trỏ - ký hiệu "Tp"- chỉ đến các phần tử có kiểu "T" được định nghĩa: Tp = <Vp, Op>, trong đó:
  - Vp = {{các đia 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 dữ liệu thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó).

Kiểu 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 cuả một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

Kiểu dữ liệu Con trỏ

- 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.
- Ví dụ:
  - Biến con trỏ trong Pascal có kích thước 4 bytes (2 bytes địa chỉ segment + 2 byte địa chỉ offset)
  - Biến con trỏ trong C có kích thước 2 hoặc 4 bytes tùy vào con trỏ near (chỉ lưu địa chỉ offset) hay far (lưu cả segment lẫn offset)

Con trỏ – Khai báo

- Cú pháp định nghĩa một kiểu con trỏ trong ngôn ngữ C :

```
typedef <kiểu cơ sở> * < kiểu con trỏ>;
```

- Ví dụ :

```
typedef   int  *intpointer;
intpointer     p;
```

hoặc

```
int  *p;
```

là những khai báo hợp lệ.

Con trỏ – *Thao tác căn bản*

- Các thao tác cơ bản trên kiểu con trỏ:(minh họa bằng C)
  - Khi 1 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)

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 dữ liệu có đặc điểm kể trên nên được khai báo như biến động. Biến động là những biến thỏa:
  - 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.

Biến động

- Do không được khai báo tường minh nên các biến động không có một định danh được kết buộc với địa chỉ vùng nhớ cấp phát cho nó, do đó gặp khó khăn khi truy xuất đến một biến động.
- Để giải quyết vấn đề, biến con trỏ (là biến không động) được sử dụng để trỏ đến biến động.
- Khi tạo ra một biến động, phải dùng một con trỏ để lưu địa chỉ của biến này và sau đó, truy xuất đến biến động thông qua biến con trỏ đã biết định danh.

Biến động

- Hai thao tác cơ bản trên biến động là tạo và hủy một biến động do biến con trỏ ‘p’ trỏ đến:
  - Tạo ra một biến động và cho con trỏ ‘p’ chỉ đến nó
  - Hủy một biến động do p chỉ đến

Biến động

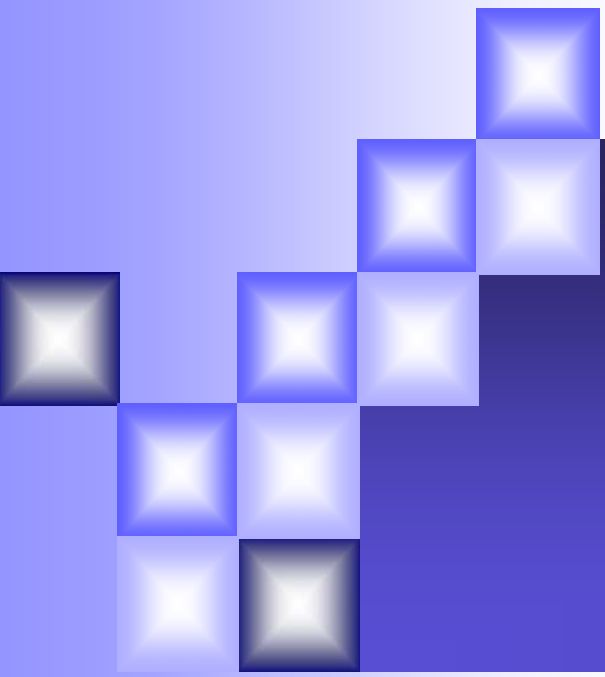
- Tạo ra một biến động và cho con trỏ ‘p’ chỉ đến nó

```
void* malloc(size);   // trả về con trỏ chỉ đến  vùng nhớ
                      // size byte vừa được cấp phát.
void* calloc(n,size);// trả về con trỏ chỉ đến  vùng nhớ
                     // vừa được cấp phát gồm n phần tử,
                     // mỗi phần tử có kích thước size byte

new   // toán tử cấp phát bộ nhớ trong C++
```

- Hàm free(p) huỷ vùng nhớ cấp phát bởi hàm malloc hoặc calloc do p trỏ tới
- Toán tử **delete** p huỷ vùng nhớ cấp phát bởi toán tử **new** do p trỏ tới

Biến động – *Ví dụ*

```
int *p1, *p2;
// cấp phát vùng nhớ cho 1 biến động kiểu int
p1 = (int*)malloc(sizeof(int));
*p1 = 5;   // đặt giá trị 5 cho biến động đang được p1 quản lý
// cấp phát biến động kiểu mảng gồm 10 phần tử kiểu int
p2 = (int*)calloc(10, sizeof(int));
*(p2+3) = 0; // đặt giá trị 0 cho phần tử thứ 4 của mảng p2
free(p1);
free(p2);
```



Danh sách liên kết (List)

Danh sách liên kết (List)
Định nghĩa

- Định nghĩa:
  - Cho T là một kiểu được định nghiã 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 móc nối với nhau theo trình tự tuyến tính};
  - Ox = {Tạo danh sách; Tìm 1 phần tử trong danh sách; Chèn 1 phần tử vào danh sách; Huỷ 1 phần tử khỏi danh sách ; Liệt kê danh sách, Sắp xếp danh sách ...}

Danh sách liên kết (List)
Định nghĩa

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

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- Các hình thức tổ chức danh sách:
  - Mối liên hệ giữa các phần tử được thể hiện ngầm
  - Mối liên hệ giữa các phần tử được thể hiện tường minh

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- **Mối liên hệ giữa các phần tử được thể hiện ngầm:**
  - Mỗi phần tử trong danh sách được đặc trưng bằng chỉ số.
  - Cặp phần tử x_i, x_{i+1} được xác định là kế cận trong danh sách nhờ vào quan hệ giữa cặp chỉ số i và (i+1).
  - Các phần tử của danh sách thường bắt buộc phải lưu trữ liên tiếp trong bộ nhớ để có thể xây dựng công thức xác định địa chỉ phần tử thứ i

    address(i) = address(1) + (i-1)*sizeof(T)
  - Có thể xem mảng và tập tin là những danh sách đặc biệt được tổ chức theo hình thức liên kết "ngầm" giữa các phần tử.

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- **Mối liên hệ giữa các phần tử được thể hiện ngầm:**
  - Cho phép truy xuất ngẫu nhiên, đơn giản và nhanh chóng đến một phần tử bất kỳ trong danh sách
  - Hạn chế về mặt sử dụng bộ nhớ.
  - Đối với mảng, số phần tử được xác định trong thời gian biên dịch và cần cấp phát vùng nhớ liên tục.

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- **Mối liên hệ giữa các phần tử được thể hiện ngầm:**
  - Trong trường hợp tổng kích thước bộ nhớ trống còn đủ để chứa toàn bộ mảng nhưng các ô nhớ trống lại không nằm kế cận nhau thì cũng không cấp phát vùng nhớ cho mảng được.
  - Ngoài ra do kích thước mảng cố định mà số phần tử của danh sách lại khó dự trù chính xác nên có thể gây ra tình trạng thiếu hụt hay lãng phí bộ nhớ.
  - Các thao tác thêm, hủy một phần tử vào danh sách được thực hiện không tự nhiên trong hình thức tổ chức này.

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

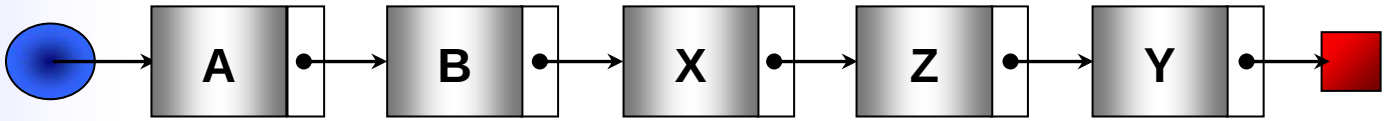
- Mối liên hệ giữa các phần tử được thể hiện tường minh
  - Mỗi phần tử ngoài các thông tin về bản thân còn chứa một liên kết (địa chỉ) đến phần tử kế trong danh sách nên còn được gọi là danh sách móc nối.
  - Do liên kết tường minh, với hình thức này các phần tử trong danh sách không cần phải lưu trữ kế cận trong bộ nhớ
  - Khắc phục được các khuyết điểm của hình thức tổ chức mảng
  - Việc truy xuất đến một phần tử đòi hỏi phải thực hiện truy xuất qua một số phần tử khác.

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

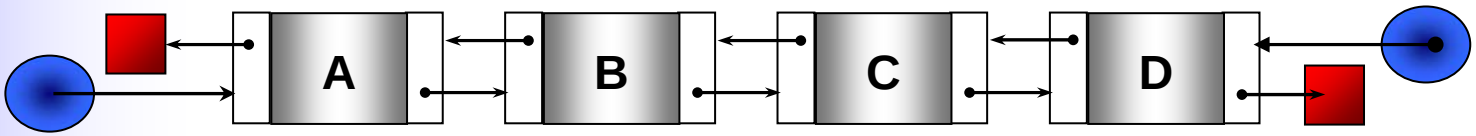
- Mối liên hệ giữa các phần tử được thể hiện tường minh
  - Có nhiều kiểu tổ chức liên kết giữa các phần tử trong danh sách như :
    - Danh sách liên kết đơn
    - Danh sách liên kết kép
    - Danh sách liên kết vòng
    - …

Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:

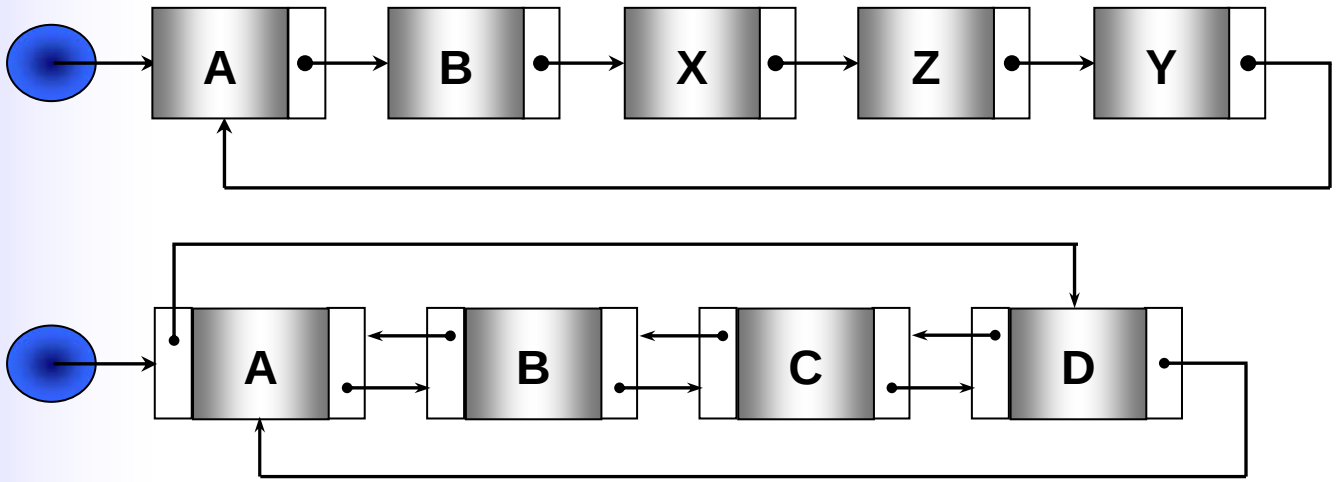


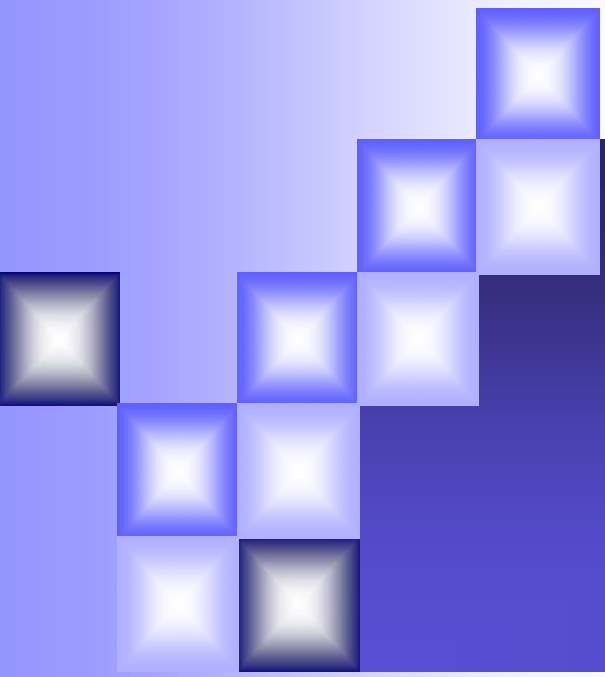
- **Danh sách liên kết kép:** mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:



Danh sách liên kết (List)
Các hình thức tổ chức danh sách

- **Danh sách liên kết vòng :** phần tử cuối danh sách liên kết với phần tử đầu danh sách:





Danh sách đơn - SList (xâu đơn)

SList – Tổ chức 1 phần tử

Mỗi phần tử của danh sách đơn gồm 2 thành phần :

- 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.

```
typedef      struct NODE {
  Data  Info;    // Data là kiểu đã định nghĩa trước
  struct NODE * pNext;  //con trỏ chỉ đến cấu trúc NODE
};
```

SList – Tổ chức 1 phần tử

- Ví dụ : Định nghĩa một phần tử trong danh sách đơn lưu trữ hồ sơ sinh viên:

```
typedef struct SV {
  char  Ten[30];
  int   MaSV;
};
typedef struct SVNode {
  SV    Info;
  struct SVNode * pNext;
};
```

SList – Tổ chức, quản lý

- Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách 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

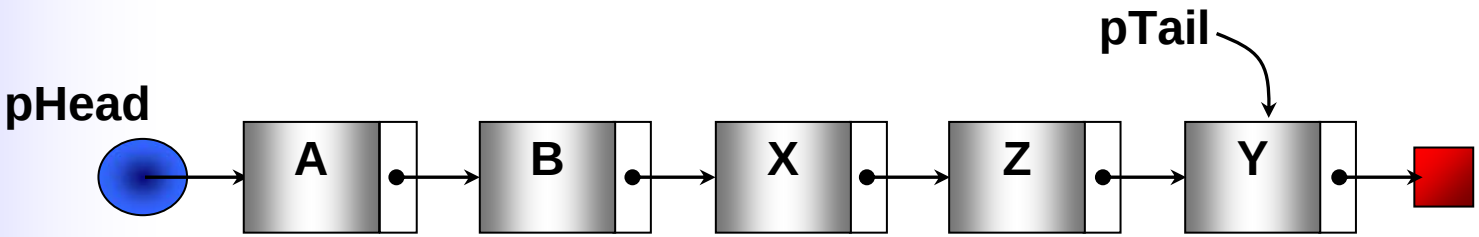
SList – Tổ chức, quản lý

- Để quản lý một xâu đơn chỉ cần biết địa chỉ phần tử đầu xâu.
- Con trỏ **Head** sẽ được dùng để lưu trữ địa chỉ phần tử đầu xâu, ta gọi **Head** là đầu xâu. Ta có khai báo:

```
NODE*    pHead;
```

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

```
NODE*    pTail;
```



SList – Tổ chức, quản lý

Khai báo kiểu của 1 phần tử và kiểu danh sách liên kết đơn:

```
// kiểu của một phần tử trong danh sách
typedef  struct NODE {
  Data           Info;
  struct NODE *  pNext;
};

typedef  struct SLIST        // kiểu danh sách liên kết
{
  NODE* pHead;
  NODE* pTail;
};
```

SList – Tạo một phần tử

- Thủ tục **GetNode** để tạo ra một phần tử cho danh sách với thông tin chứa trong x

```
NODE*  GetNode(Data x)
{
    NODE *p;
    // Cấp phát vùng nhớ cho phần tử
    p = new NODE;
    if (p==NULL)   {
        printf("Không đủ bộ nhớ"); return NULL;
    }
    p->Info = x;  // Gán thông tin cho phần tử p
    p->pNext = NULL;
    return p;
}
```

SList – Tạo một phần tử

Để tạo một phần tử mới cho danh sách, cần thực hiện câu lệnh:

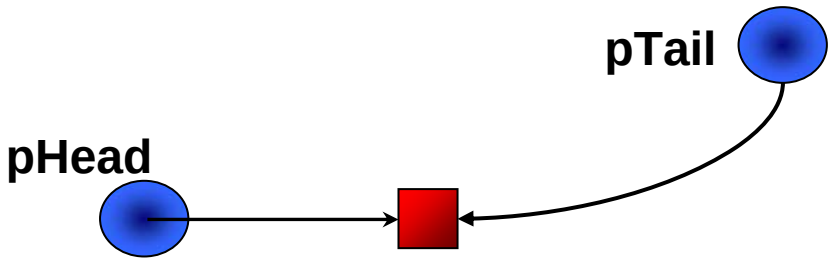
```
new_ele = GetNode(x);
```

→ new_ele sẽ quản lý địa chỉ của phần tử mới được tạo.

SList – Các thao tác cơ sở

- Tạo danh sách rỗng
- Thêm một phần tử vào danh sách
- Tìm kiếm một giá trị trên danh sách
- Trích một phần tử ra khỏi danh sách
- Duyệt danh sách
- Sắp xếp danh sách
- Hủy toàn bộ danh sách

SList – Khởi tạo danh sách rỗng



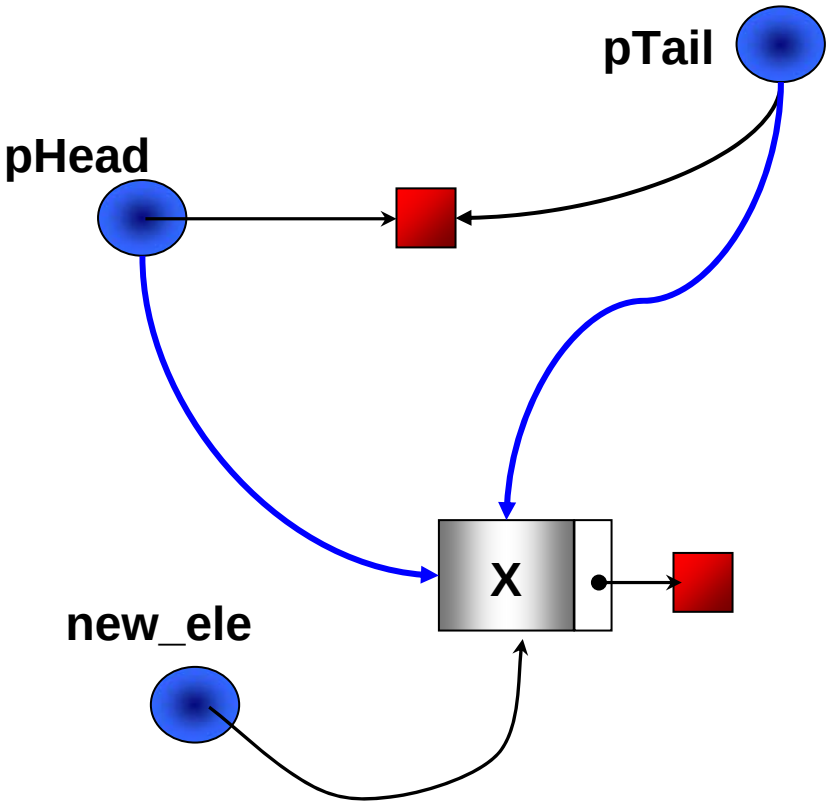

```
void Init(SLIST &l)
{
  l.pHead = l.pTail = NULL;
}
```

SList – Thêm một phần tử

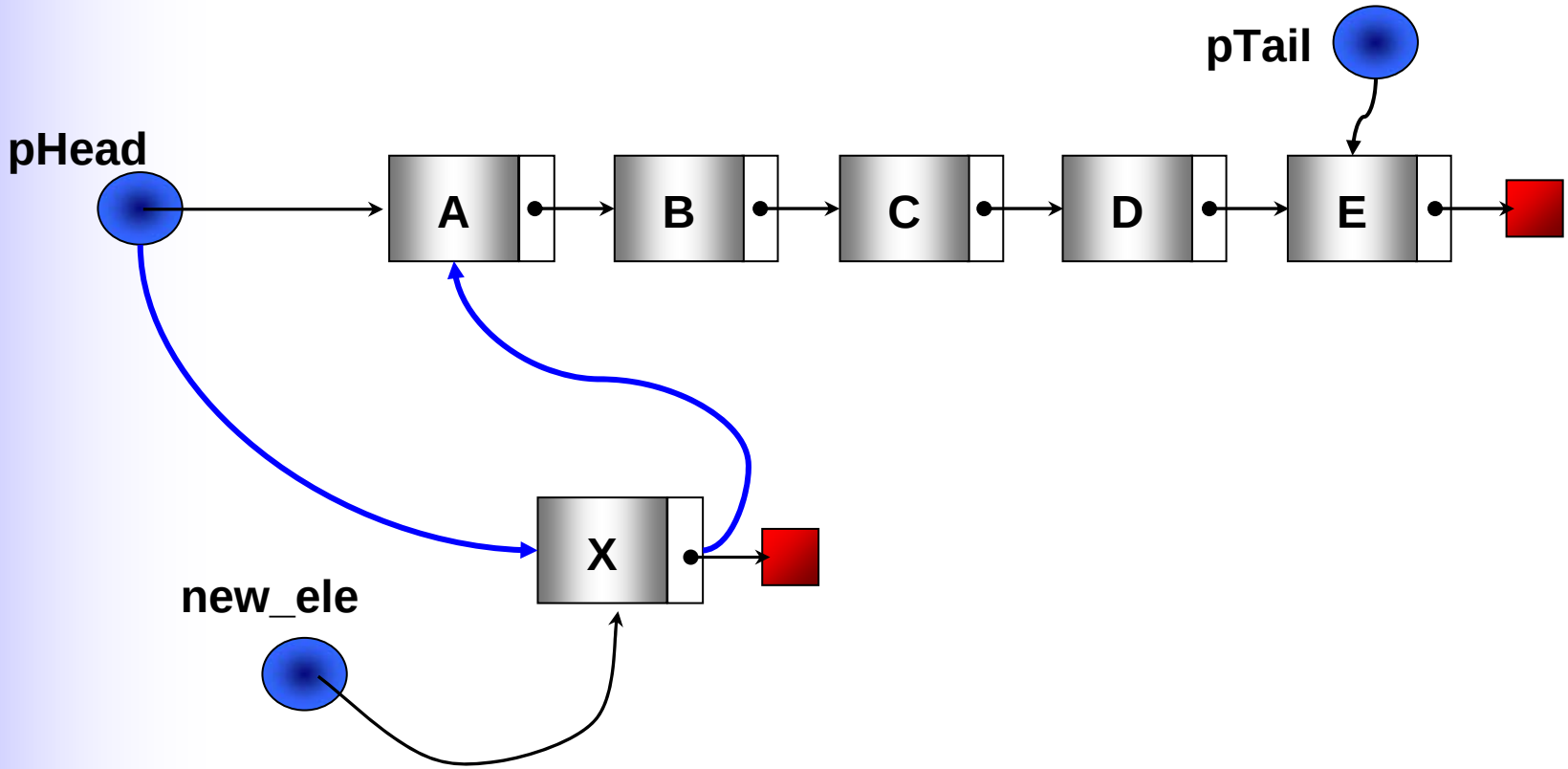
3 vị trí thêm phần tử mới vào danh sách:

- Thêm vào đầu danh sách
- Nối vào cuối danh sách
- Chèn vào danh sách sau một phần tử q

SList – Thêm một phần tử



SList – Thêm một phần tử vào đầu



SList – Thêm một phần tử vào đầu

//input: danh sách (head, tail), phần tử mới new_ele

//output: danh sách (head, tail) với new_ele ở đầu DS

- Nếu Danh sách rỗng Thì
  - Head = new_ele;
  - Tail = Head;
- Ngược lại
  - new_ele ->pNext = Head;
  - Head = new_ele ;

SList – Thêm một phần tử vào đầu

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

SList
Thêm một thành phần dữ liệu vào đầu

//input: danh sách (head, tail), thành phần dữ liệu X

//output: danh sách (head, tail) với phần tử chứa X ở đầu DS

- Tạo phần tử mới new_ele để chứa dữ liệu X
- Nếu tạo được:

  Thêm new_ele vào đầu danh sách
- Ngược lại

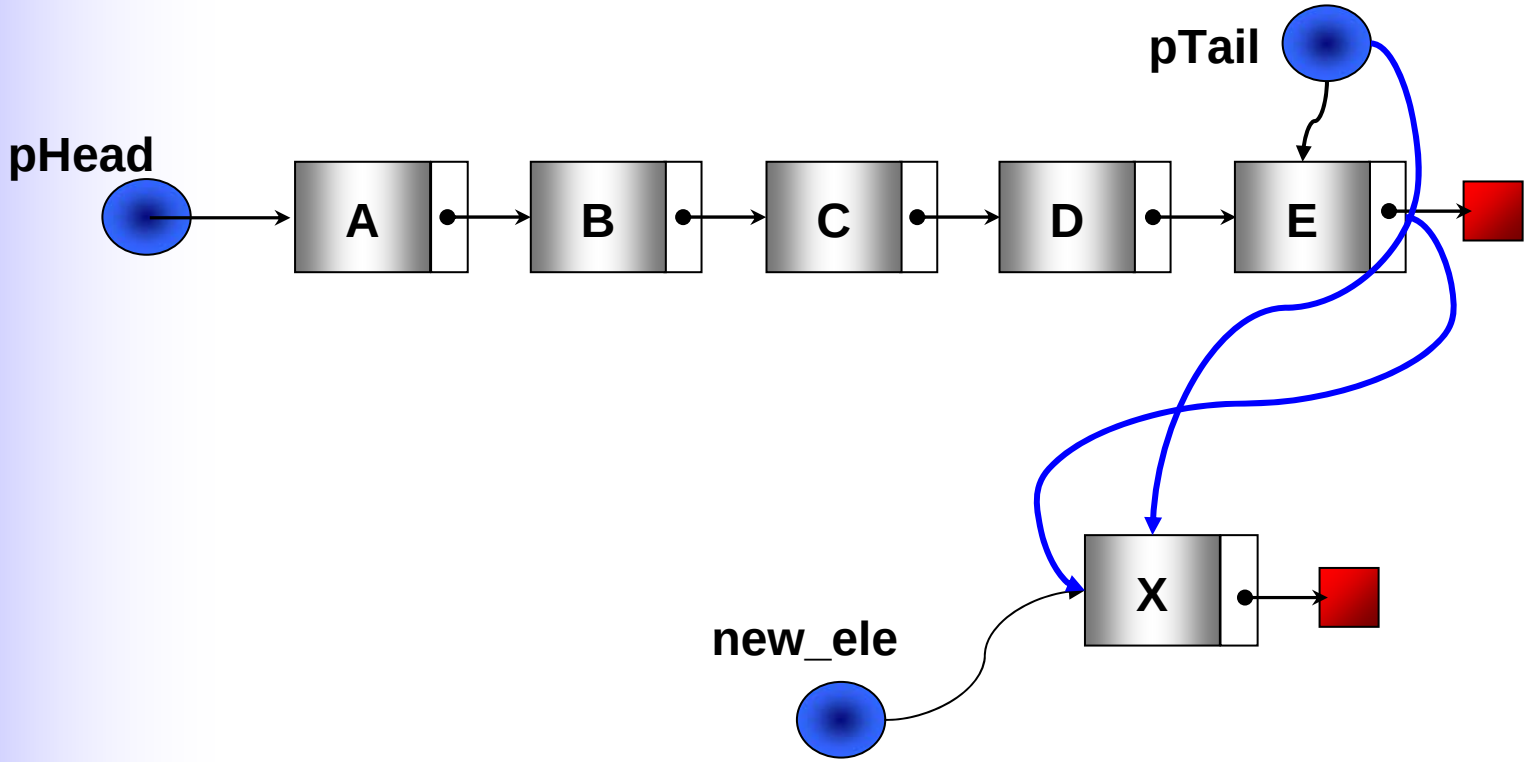
  Báo lỗi

SList

Thêm một thành phần dữ liệu vào đầu

```
NODE* InsertHead(SLIST &l, Data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL)
        return NULL;
    AddFirst(l, new_ele);
    return new_ele;
}
```

SList – Thêm một phần tử vào cuối



SList – Thêm một phần tử vào cuối

//input: danh sách (head, tail), phần tử mới new_ele

//output: danh sách (head, tail) với new_ele ở cuối DS

- Nếu Danh sách rỗng Thì
  - Head = new_ele;
  - Tail = Head;
- Ngược lại
  - Tail->pNext = new_ele ;
  - Tail = new_ele ;

SList – Thêm một phần tử vào cuối

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

}
```

SList
Thêm một thành phần dữ liệu vào cuối

//input: danh sách (head, tail), thành phần dữ liệu X

//output: danh sách (head, tail) với phần tử chứa X ở cuối DS

- Tạo phần tử mới new_ele để chứa dữ liệu X
- Nếu tạo được:

  Thêm new_ele vào cuối danh sách
- Ngược lại

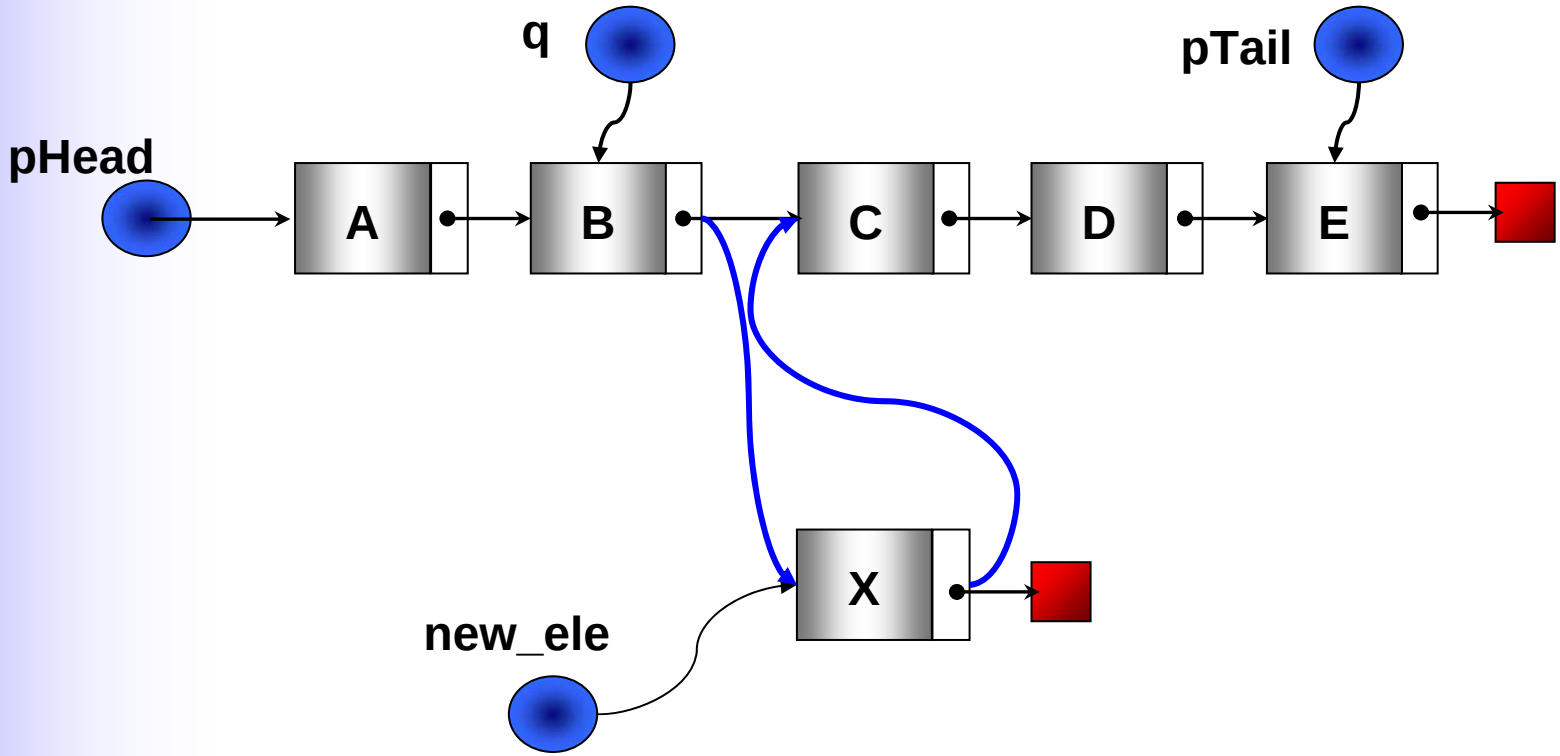
  Báo lỗi

SList

Thêm một thành phần dữ liệu vào cuối

```
NODE* InsertTail(SLIST &l, Data x)
{
    NODE* new_ele = GetNode(x);

    if (new_ele == NULL)
        return NULL;
    AddTail(l, new_ele);
    return new_ele;
}
```

SList – Chèn một phần tử sau q



SList – Chèn một phần tử sau q

//input: danh sách (head, tail), q, phần tử mới new_ele

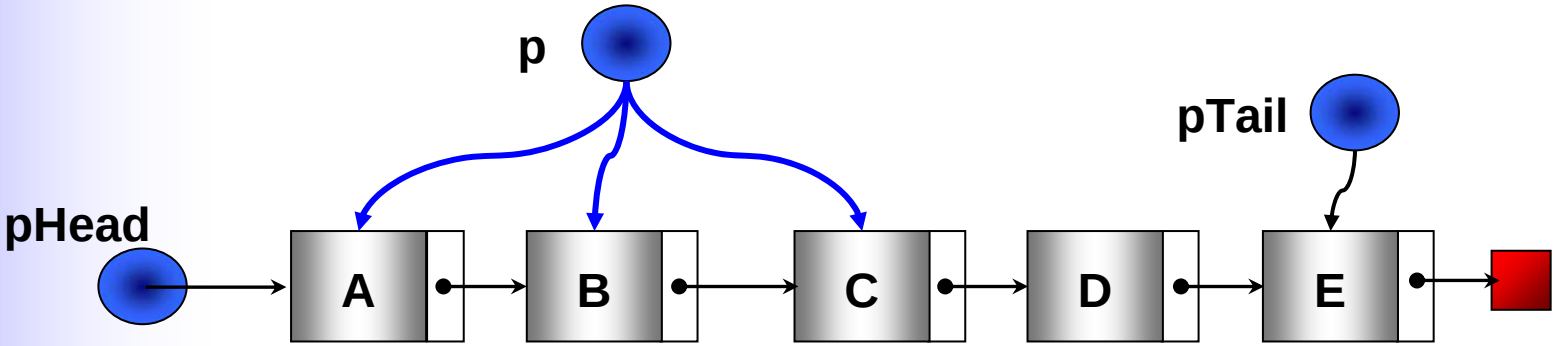
//output: danh sách (head, tail) với new_ele ở sau q

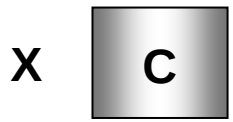
```
Nếu  ( q != NULL) thì
        new_ele -> pNext  = q -> pNext;
        q -> pNext = new_ele ;
        Nếu  ( q == Tail) thì
                Tail = new_ele;
Ngược lại
      Thêm new_ele vào đầu danh sách
```

SList – Chèn một phần tử sau q

```
void AddAfter(SLIST &l,NODE *q, NODE* new_ele)
{
   if (q!=NULL)  {
         new_ele->pNext = q->pNext;
         q->pNext = new_ele;
         if(q == l.pTail)
               l.pTail = new_ele;
   }
   else   // thêm vào đầu danh sách
          AddFirst(l, new_ele);
}
```

SList – Tìm một phần tử





Ok, found

SList – Tìm một phần tử

//input: danh sách (head, tail), dữ liệu cần tìm X

//output: địa chỉ phần tử chứa X

- 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:
  - B21 : p:=p->Next;// Cho p trỏ tới phần tử kế
- Bước 3:
  - Nếu p != NULL thì p trỏ tới phần tử cần tìm
  - Ngược lại: không có phần tử cần tìm.

SList – Tìm một phần tử

```
NODE *Search(SLIST l, Data  X)
{
   NODE     *p;

   for (p=l.pHead;(p)&&(p->Info!=X);p=p->pNext);
   return p;
}
```

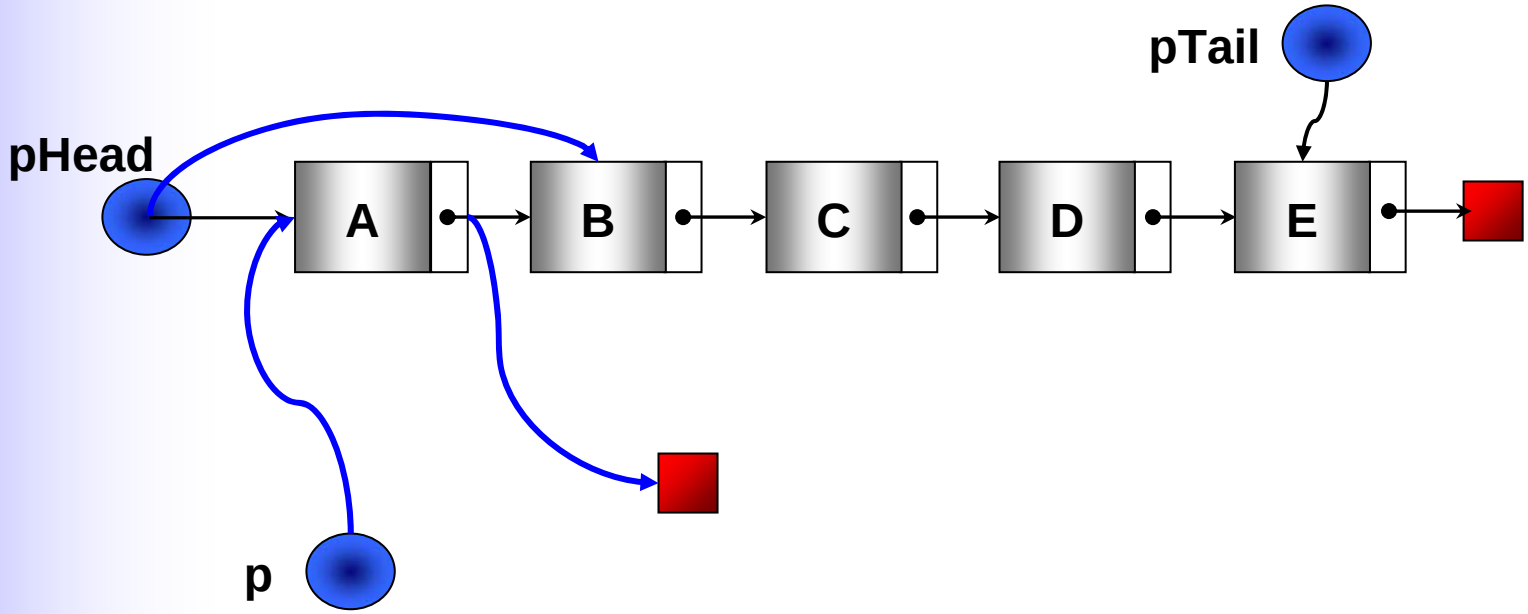
SList – Trích một phần tử

3 trường hợp trích 1 phần tử ra khỏi danh sách:

- Trích phần tử đầu
- Trích phần tử sau một phần tử q
- Trích phần tử có giá trị X

- Lưu ý: Nếu muốn hủy phần tử vừa trích, cần gọi thêm toán tử delete để giải phóng bộ nhớ.

SList – Trích phần tử đầu xâu



SList – Trích phần tử đầu xâu

//input: xâu (head, tail)

//output: xâu đã bị trích phần tử đầu xâu

- Nếu (Head != NULL) thì
  - p = Head; *// p là phần tử cần trích*
  - Head = Head->pNext; *// tách p ra khỏi xâu*
  - p -> pNext = NULL;
  - Nếu Head=NULL thì Tail = NULL; *//Xâu rỗng*

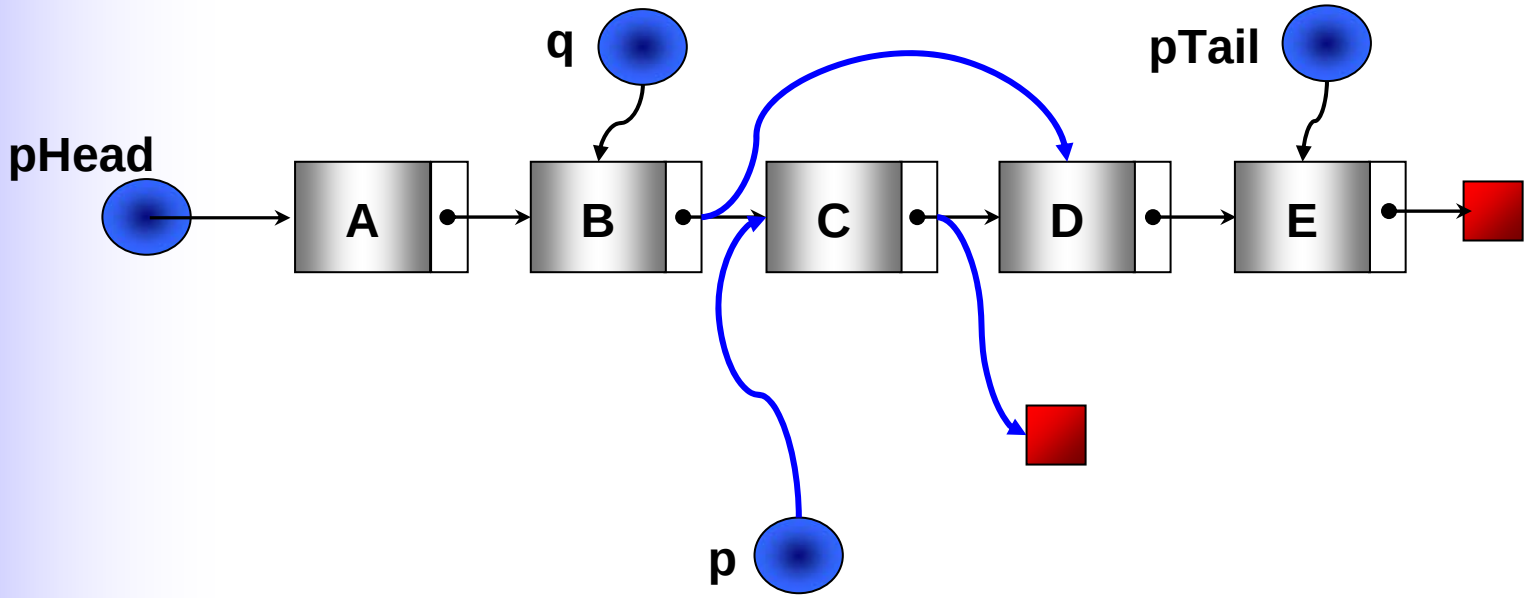
SList – Trích phần tử đầu xâu

```
NODE* PickHead(SLIST &l)
{
    NODE *p = NULL;
    if (l.pHead != NULL)  {
        p = l.pHead;
        l.pHead = l.pHead->pNext;
        p->pNext = NULL;
        if(l.pHead == NULL)
            l.pTail =  NULL;
    }
    return p;
}
```

SList – Trích & hủy phần tử đầu xâu

```
Data RemoveHead(SLIST &l)
{
    if (l.pHead == NULL)
        return NULLDATA;
    NODE* p = PickHead(l);
    Data x = p->Info;
    delete p;
    return x;
}
```

SList – Trích phần tử sau q



SList – Trích phần tử sau q

//input: xâu (head, tail), con trỏ q

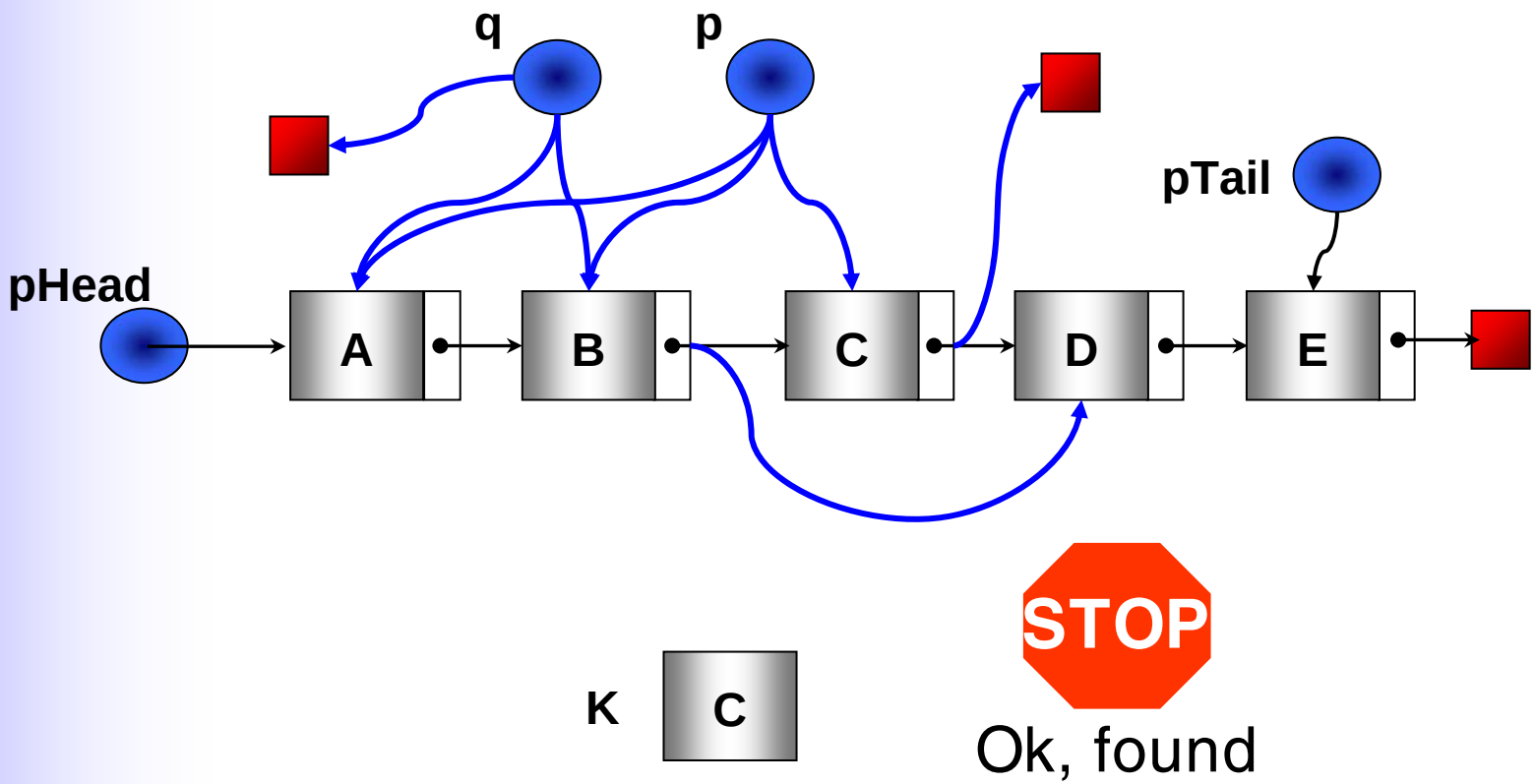
//output: xâu đã bị trích phần tử sau phần tử q

- Nếu (q!= NULL)
  - p = q->pNext; *// p là phần tử cần trích*
  - Nếu (p != NULL) *// q không phải là cuối xâu*
    - q->pNext = p->pNext; *// tách p ra khỏi xâu*
    - p->pNext = NULL;
    - Nếu (p==Tail)
      - Tail = q;
- Ngược lại:
  - Trích phần tử đầu xâu

SList – Trích phần tử sau q

```
NODE  * PickAfter (SLIST &l, NODE *q)
{
   NODE       *p;
   if ( q != NULL) {
       p = q ->pNext ;
       if ( p != NULL) {
             if (p == l.pTail)  l.pTail = q;
             q->pNext = p->pNext;
             p->pNext = NULL;
       }
   }
   else p = PickHead(l);
   return p;
}
```

SList – Trích phần tử có khóa K



SList – Trích phần tử có khóa K

//input: xâu (head, tail), khóa k

//output: xâu đã bị trích phần tử có khóa k

- Bước 1:
  - Tìm phần tử p có khóa k và phần tử q đứng trước nó
- Bước 2:
  - Nếu (p!= NULL) thì // tìm thấy k
    - Trích p ra khỏi xâu tương tự trích phần tử sau q;
  - Ngược lại
    - Báo không có k;

SList – Trích phần tử có khóa K

```
NODE * PickNode(SLIST &l, Data k)
{
    NODE   *p = l.pHead;
    NODE   *q = NULL;
    while ((p != NULL) && (p->Info != k))
    {
        q = p;
        p = p->pNext;
    }
    if(p == NULL) return NULL;
    return PickAfter(l, q);
}
```

SList – Duyệt danh sách

- Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:
  - Đếm các phần tử của danh sách,
  - Tìm tất cả các phần tử thoả điều kiện,

SList – Duyệt danh sách

- Bước 1:p = Head; *//Cho p trỏ đến phần tử đầu danh sách*
- Bước 2: Trong khi (Danh sách chưa hết) thực hiện
  - B21 : Xử lý phần tử p;
  - B22 : p:=p->pNext; *// Cho p trỏ tới phần tử kế*

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

SList – 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 = Head;
    - Head:=Head->pNext; *// Cho p trỏ tới phần tử kế*
  - B12:
    - Hủy p;
- Bước 2:
  - Tail = NULL; *//Bảo đảm tính nhất quán khi xâu rỗng*

SList – 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

Sắp xếp danh sách

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)

Sắp xếp danh sách

Hoán vị nội dung các phần tử trong danh sách

- Cài đặt lại trên xâu một trong những thuật toán sắp xếp đã biết trên mảng
- 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.
- Do thực hiện hoán vị nội dung của các phần tử nên đòi hỏi sử dụng thêm vùng nhớ trung gian $\Rightarrow$ chỉ thích hợp với các xâu 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 phân tử sẽ chiếm chi phí đáng kể.
- Không tận dụng được các ưu điểm của xâu!

Slist – Sắp xếp

Hoán vị nội dung các phần tử trong danh sách

```
void  ListSelectionSort (LIST &l)
{
   NODE  *min, *p,*q;
   p = l.pHead;
   while(p != l.pTail) {
      q = p->pNext; min = p;
      while(q != NULL) {
         if(q->Info< min->Info )
            min = q; // ghi nhận vị trí phần tử min hiện hành
         q = q->pNext;
      }
      Hoanvi(min->Info, p->Info);
      p = p->pNext;
   }
}
```

Slist – Sắp xếp Thay đổi các mối liên kết

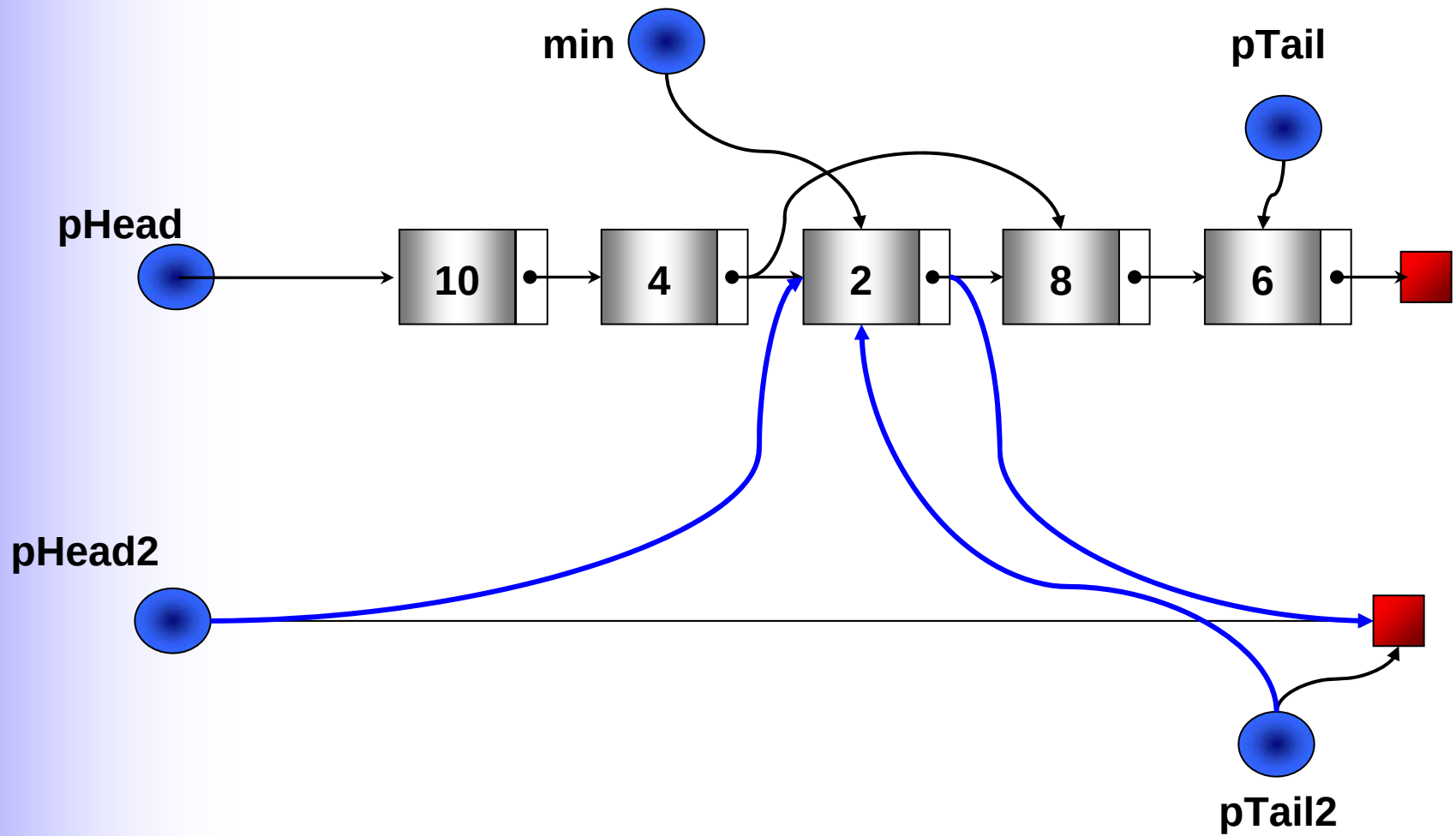
- Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn ⇒ chỉ thao tác trên các móc nối (pNext).
- Kích thước của trường pNext:
  - Không phụ thuộc vào bản chất dữ liệu lưu trong xâu
  - Bằng kích thước 1 con trỏ (2 hoặc 4 byte trong 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.

⇒Cần cân nhắc khi chọn cách tiếp cận: Nếu dữ liệu không quá lớn thì nên chọn phương án 1 hoặc một thuật toán hiệu quả nào đó.

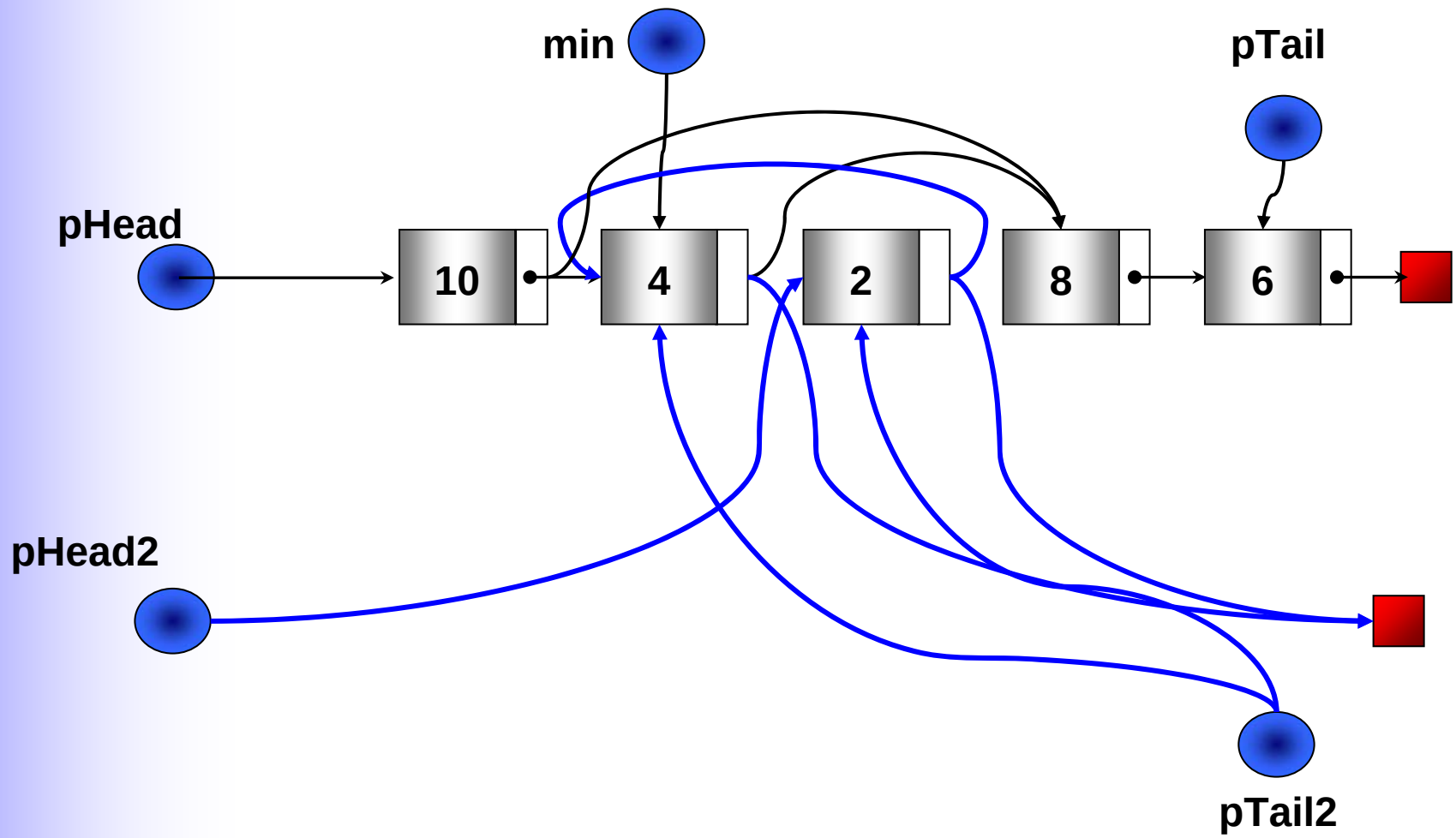
Slist – Sắp xếp Thay đổi các mối liên kết

- Một trong những cách thay đổi móc nối đơn giản nhất là tạo một danh sách mới là danh sách có thứ tự gồm các phần tử trích từ danh sách cũ.
- Giả sử danh sách mới sẽ được quản lý bằng con trỏ đầu xâu Result, ta có thuật toán chọn trực tiếp của phương án 2 như sau:
  - B1: Khởi tạo danh sách mới Result là rỗng;
  - B2: Tìm trong danh sách cũ l phần tử nhỏ nhất – min;
  - B3: Tách min khỏi danh sách cũ;
  - B4: Chèn min vào cuối danh sách Result;
  - B5: Lặp lại bước 2 khi chưa hết danh sách cũ;

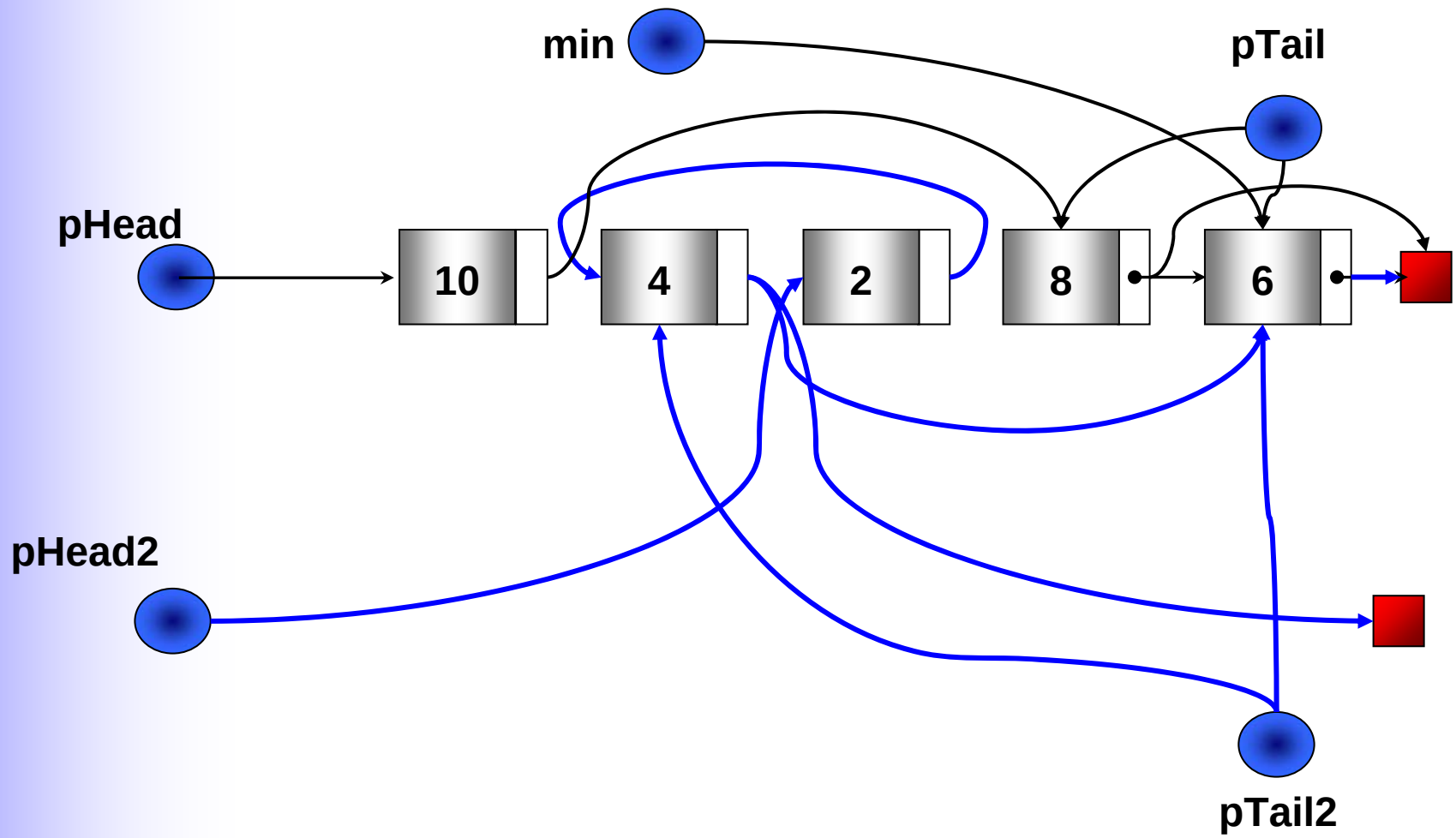
SList – Sắp xếp chọn trực tiếp



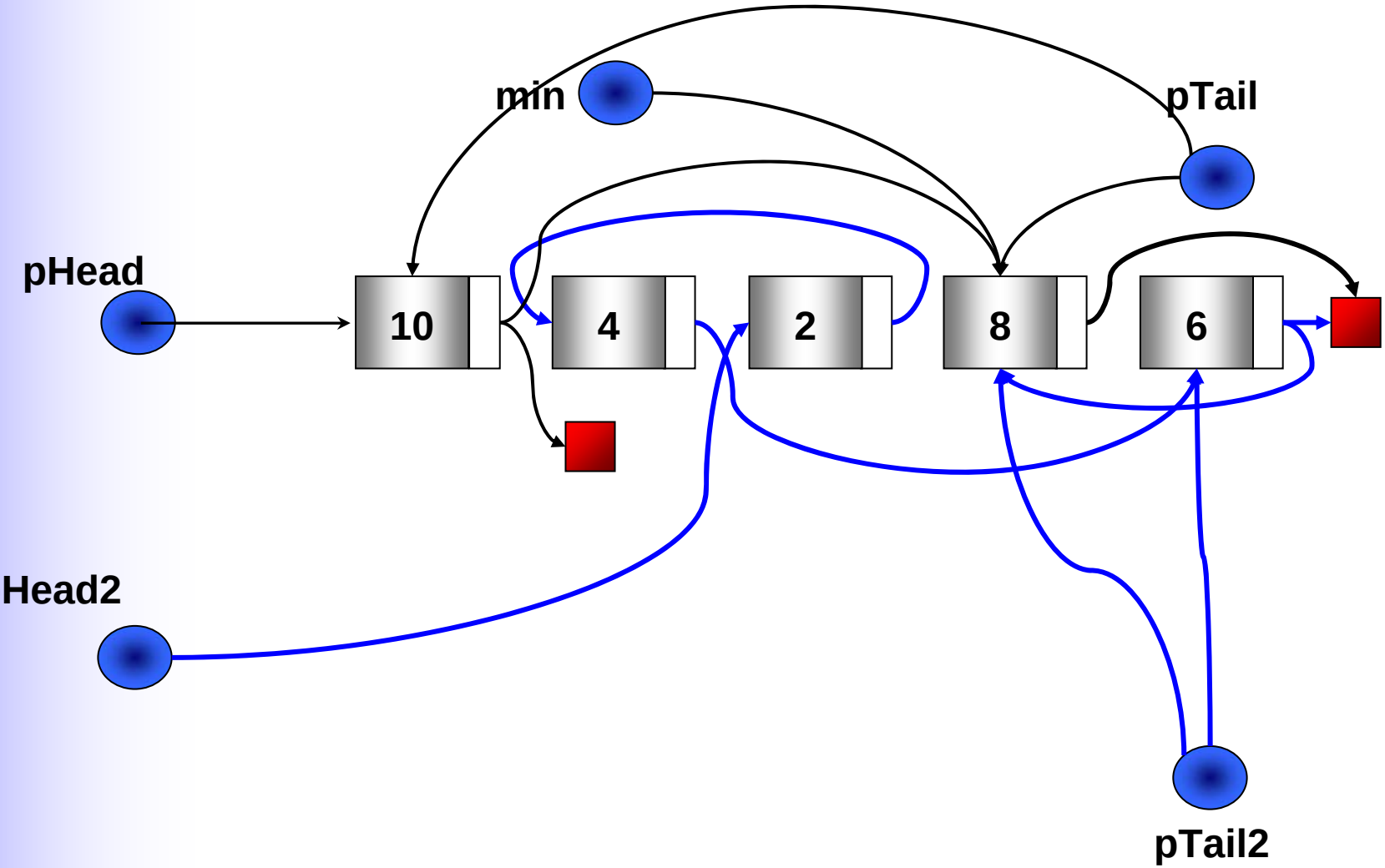
SList – Sắp xếp chọn trực tiếp



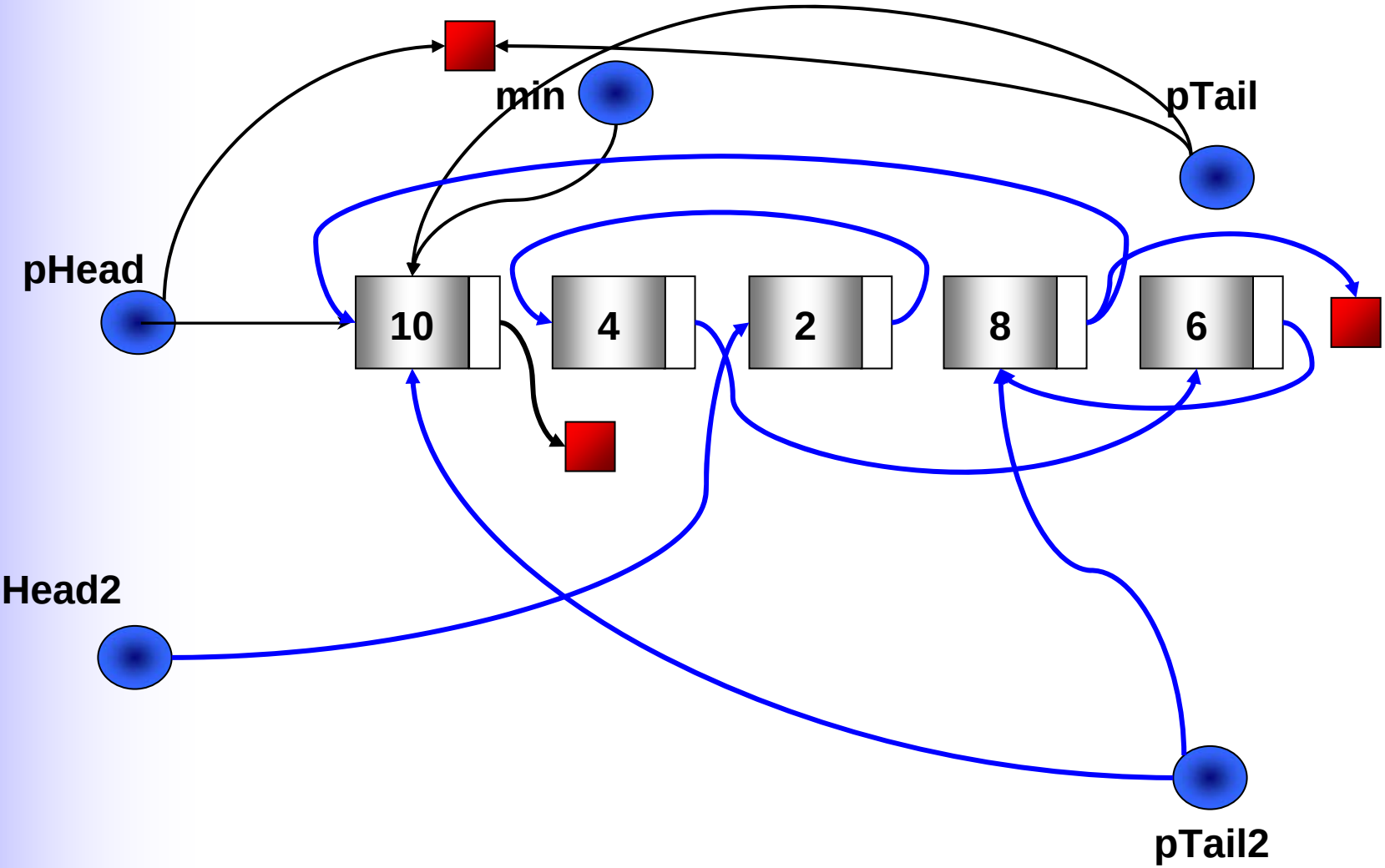
SList – Sắp xếp chọn trực tiếp



SList – Sắp xếp chọn trực tiếp



SList – Sắp xếp chọn trực tiếp



SList – Sắp xếp chọn trực tiếp

```
NODE* FindMinprev (LIST list)
{
   NODE *min, *minprev, *p, *q;
   minprev = q = NULL;    min=p = list.pHead;
   while(p != NULL) {
      if (p->Info < min->Info) {
         min = p;     minprev = q;
      }
      q = p;     p = p->pNext;
   }
   return minprev;
}
```

SList – Sắp xếp chọn trực tiếp

```
void  ListSelectionSort2 (SLIST &list)
{
   SLIST    lRes;
   NODE     *min, *minprev;
   lRes.pHead = lRes.pTail = NULL;
   while(list.pHead != NULL)  {
      minprev = FindMinprev(list);
      min = PickAfter(list, minprev);
      AddTail(lRes, min);
   }
   list = lRes;
}
```

SList

Một số thuật toán sắp xếp hiệu quả

- Thuật toán Quick Sort
- Thuật toán Merge Sort
- Thuật toán Radix Sort

SList –Quick Sort: Thuật toán

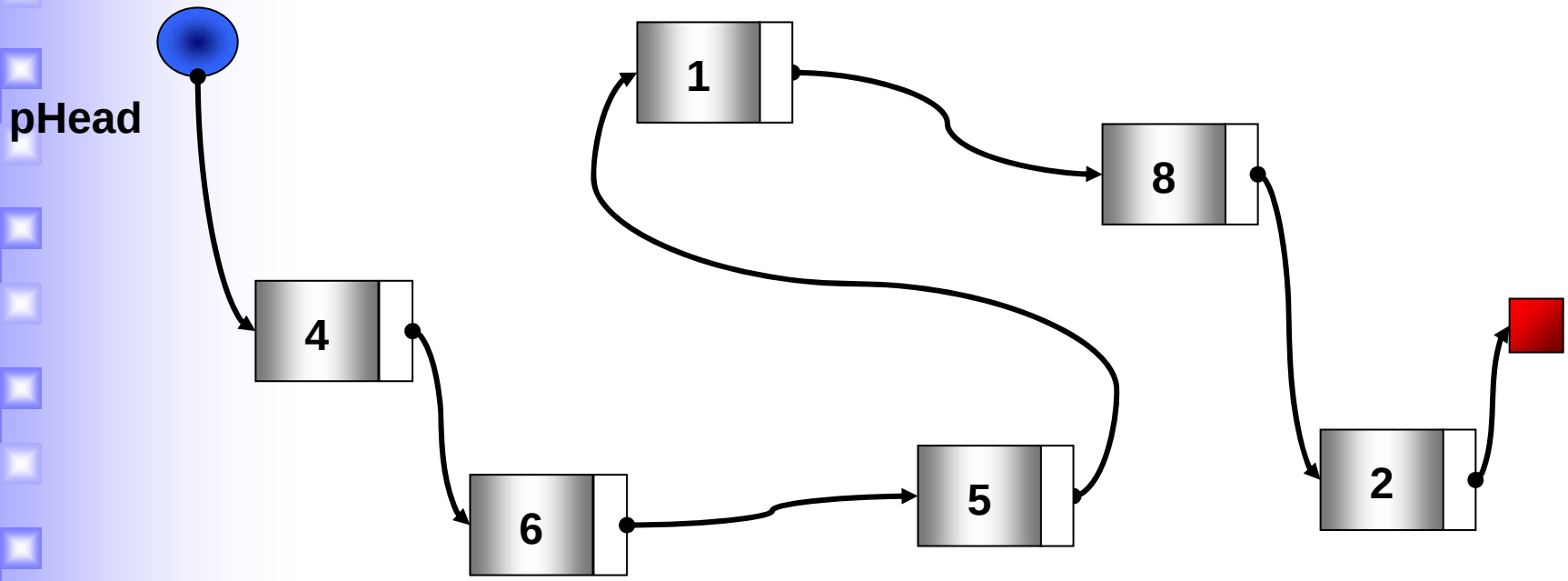
//input: xâu (head, tail)

//output: xâu đã được sắp tăng dần

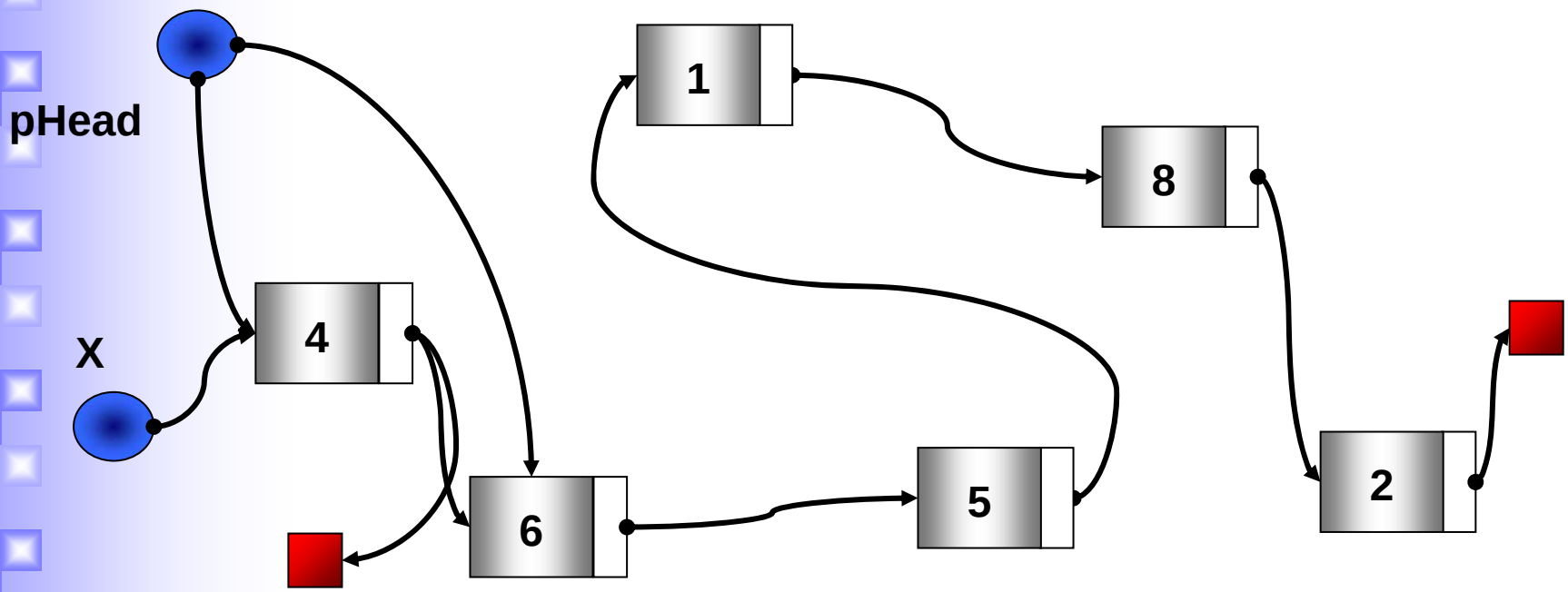
- Bước 1: Nếu xâu có ít hơn 2 phần tử

  Dừng; *//xâu đã có thứ tự*
- Bước 2: Chọn X là phần tử đầu xâu L làm ngưỡng. Trích X ra khỏi L.
- Bước 3: Tách xâu L ra làm 2 xâu L_1 (gồm các phần tử nhỏ hơn hay bằng X) và L_2 (gồm các phần tử lớn hơn X).
- Bước 4: Sắp xếp Quick Sort (L_1).
- Bước 5: Sắp xếp Quick Sort (L_2).
- Bước 6: Nối L_1, X, và L_2 lại theo trình tự ta có xâu L đã được sắp xếp.

SList – Sắp xếp quick sort

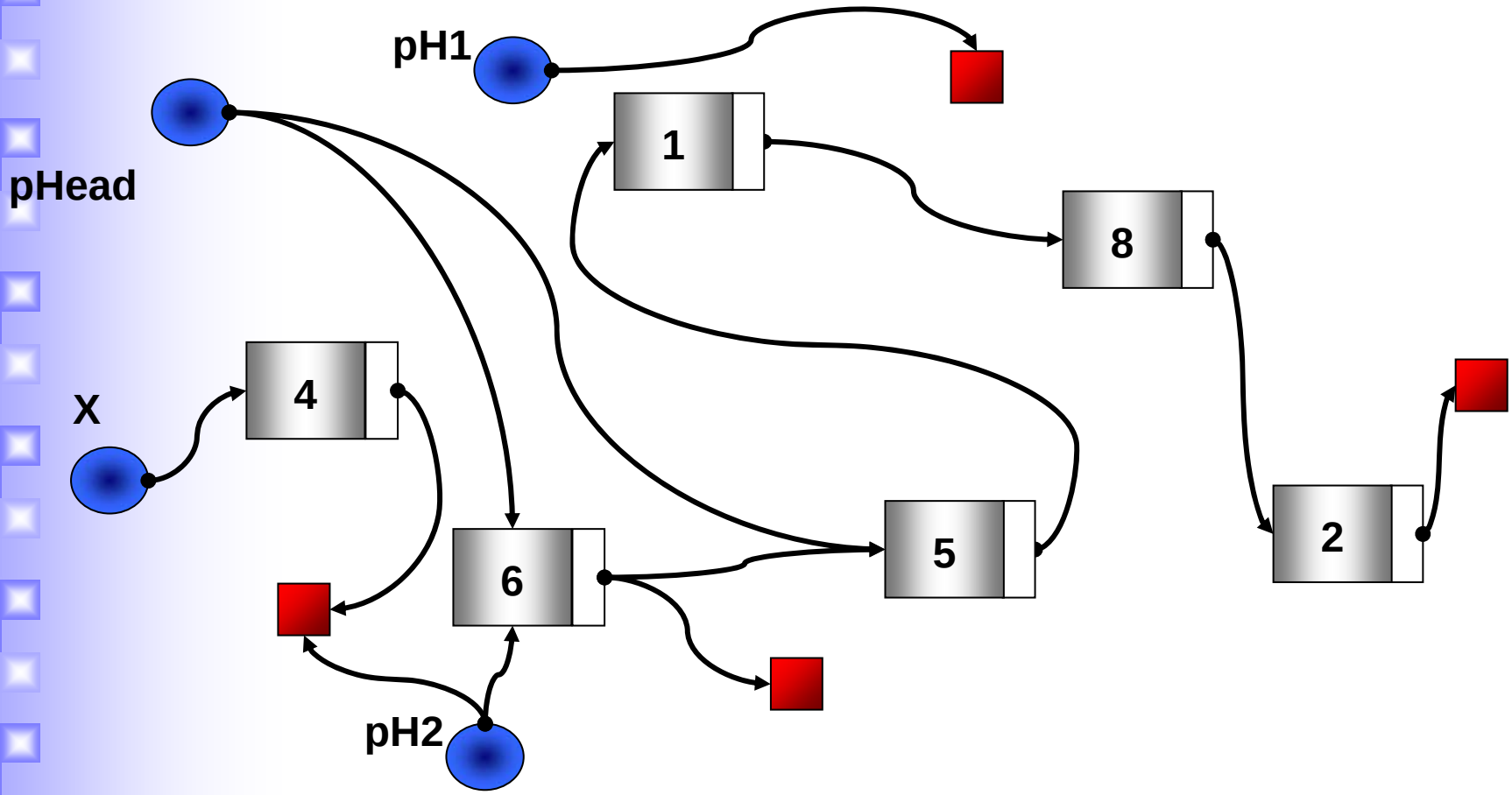


SList – quick sort: phân hoạch



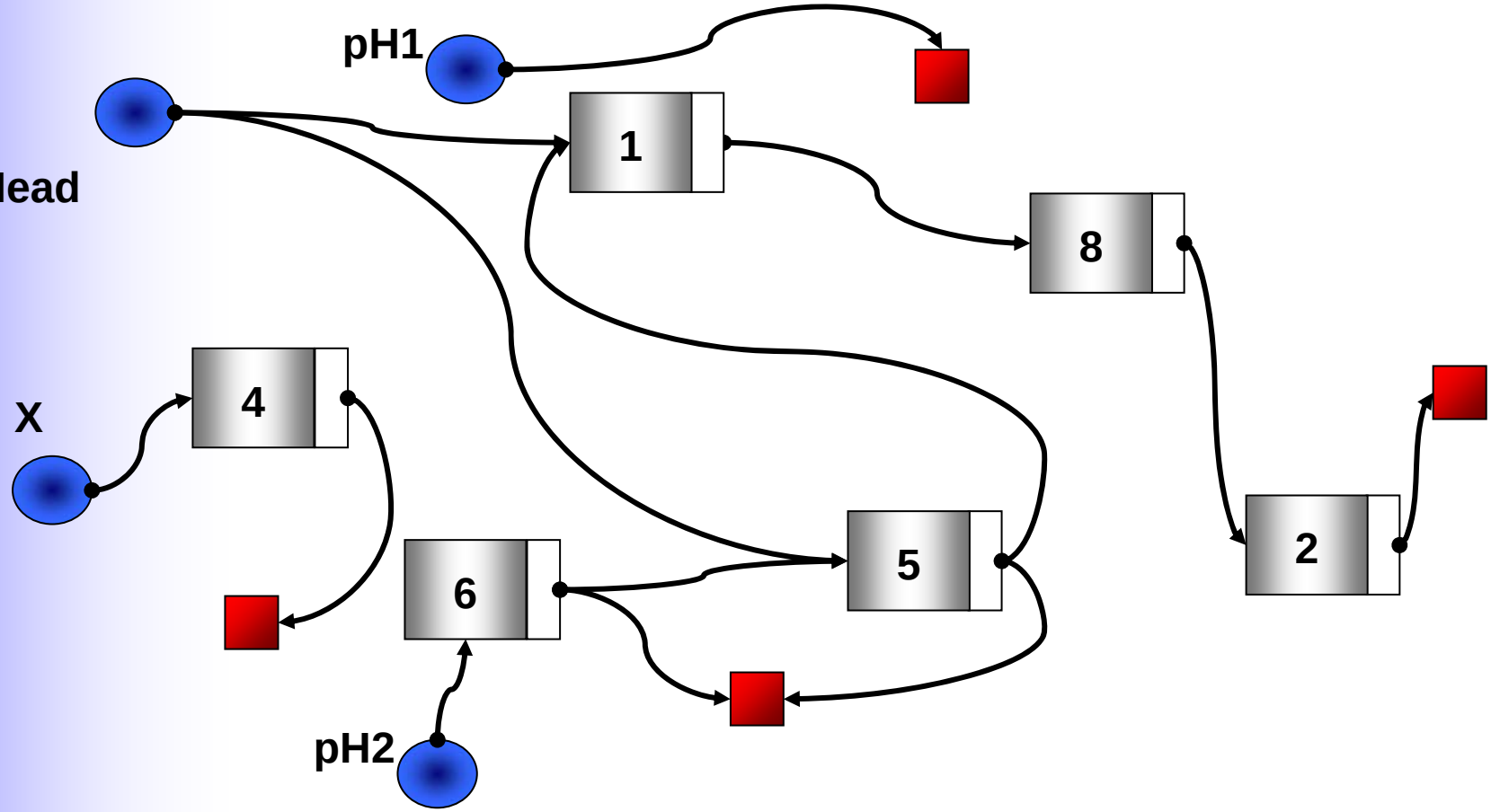
Chọn phần tử đầu xâu làm ngưỡng

SList – quick sort: phân hoạch



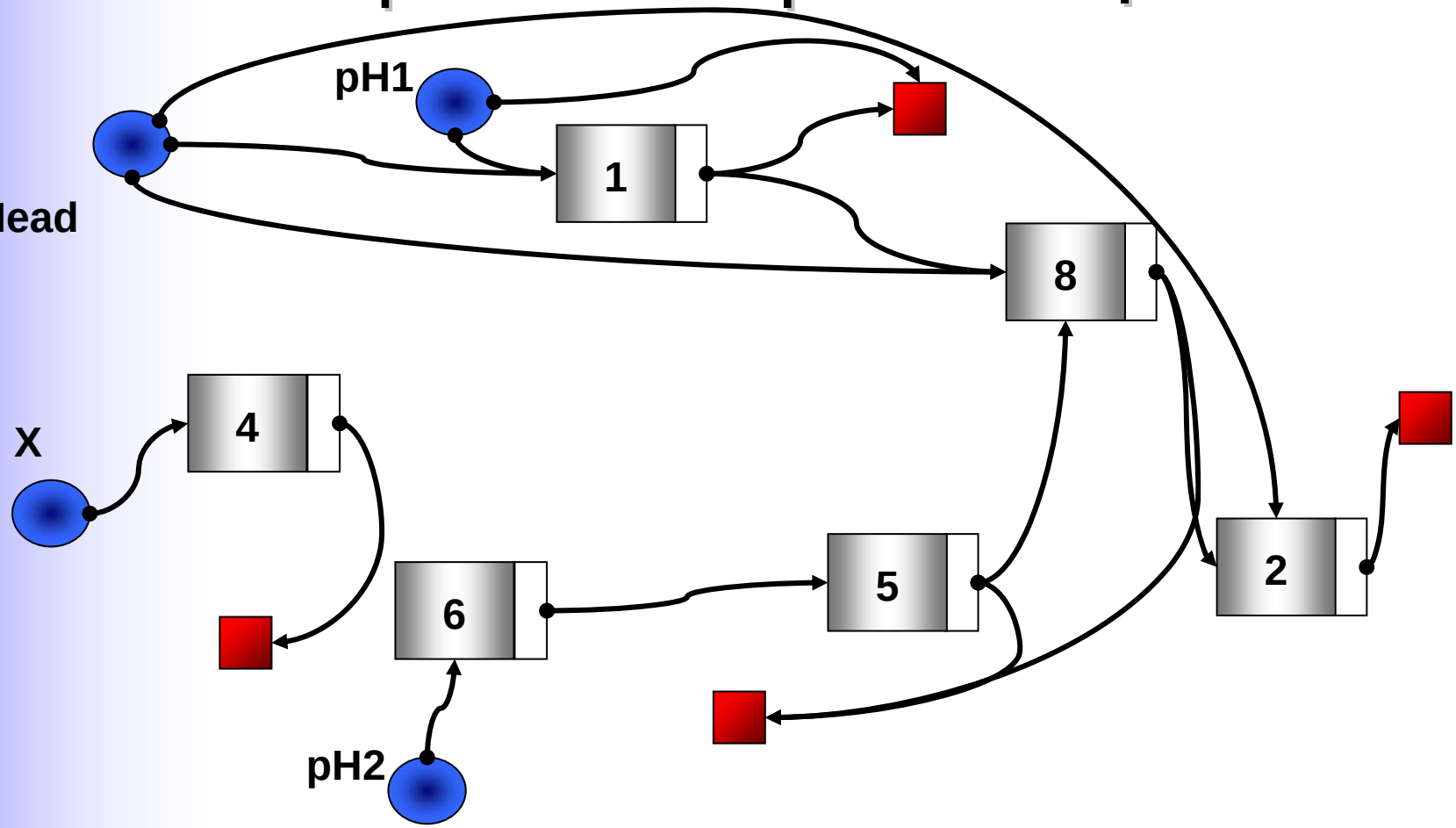
Tách xâu hiện hành thành 2 xâu

SList – quick sort: phân hoạch



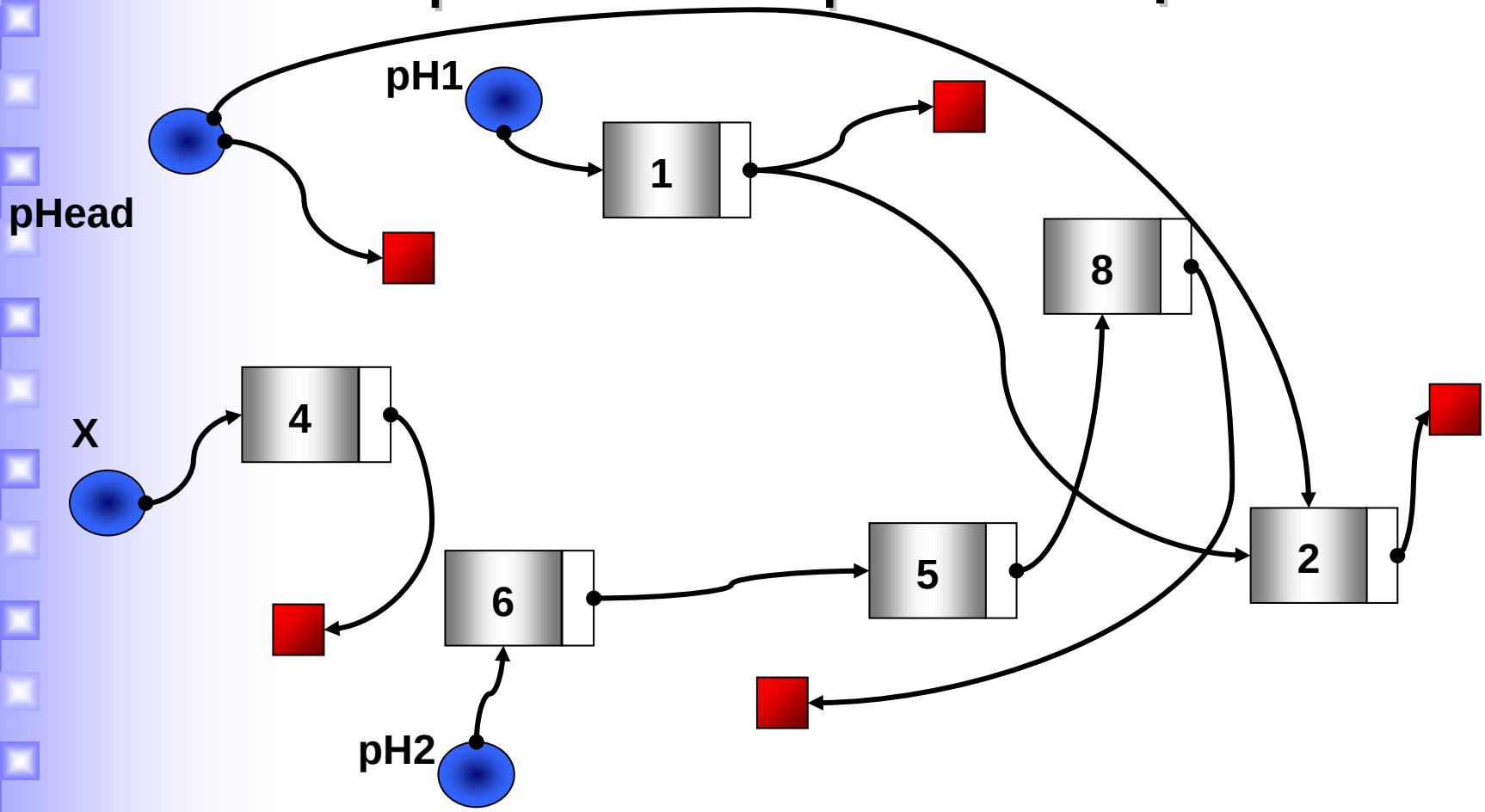
Tách xâu hiện hành thành 2 xâu

SList – quick sort: phân hoạch



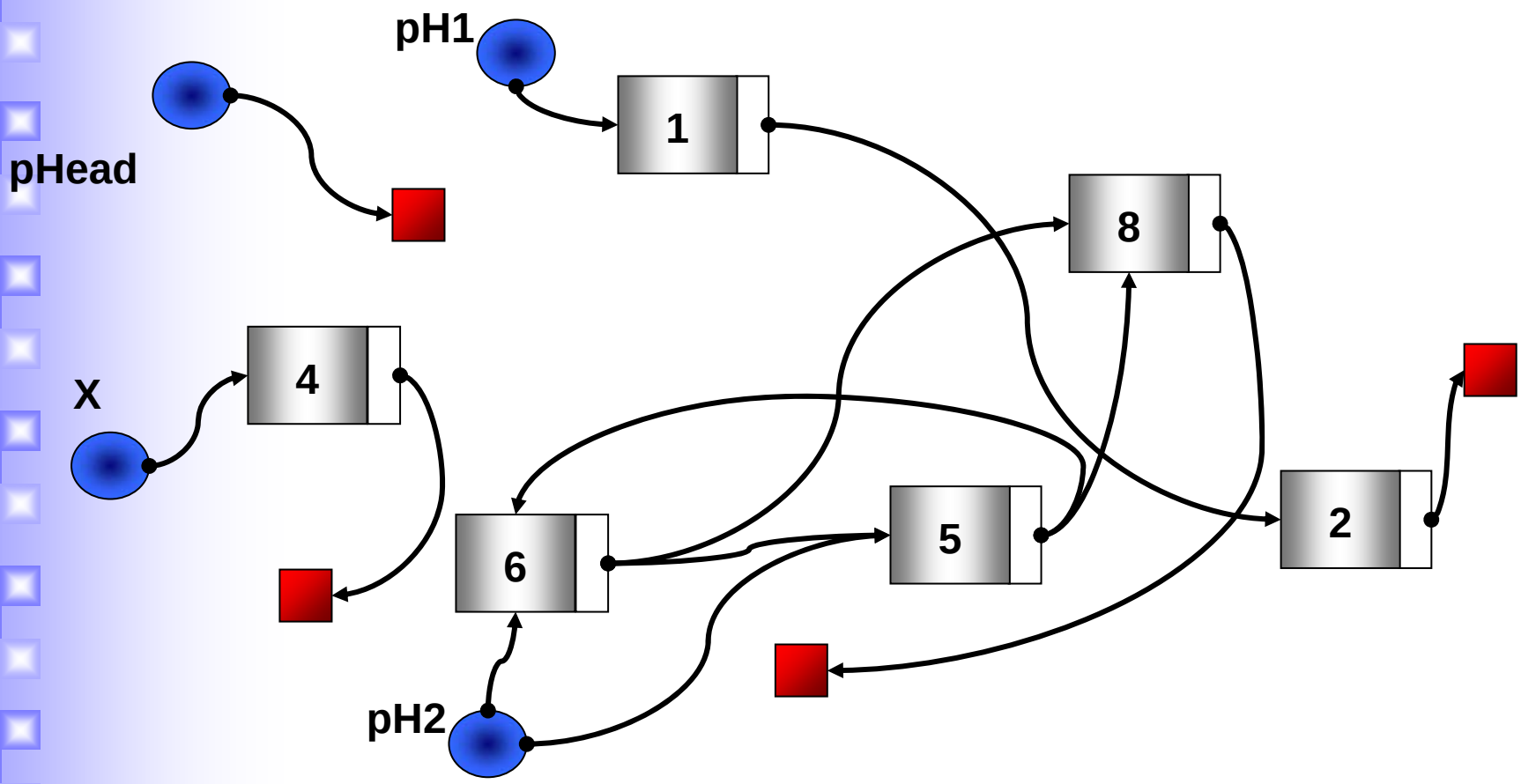
Tách xâu hiện hành thành 2 xâu

SList – quick sort: phân hoạch



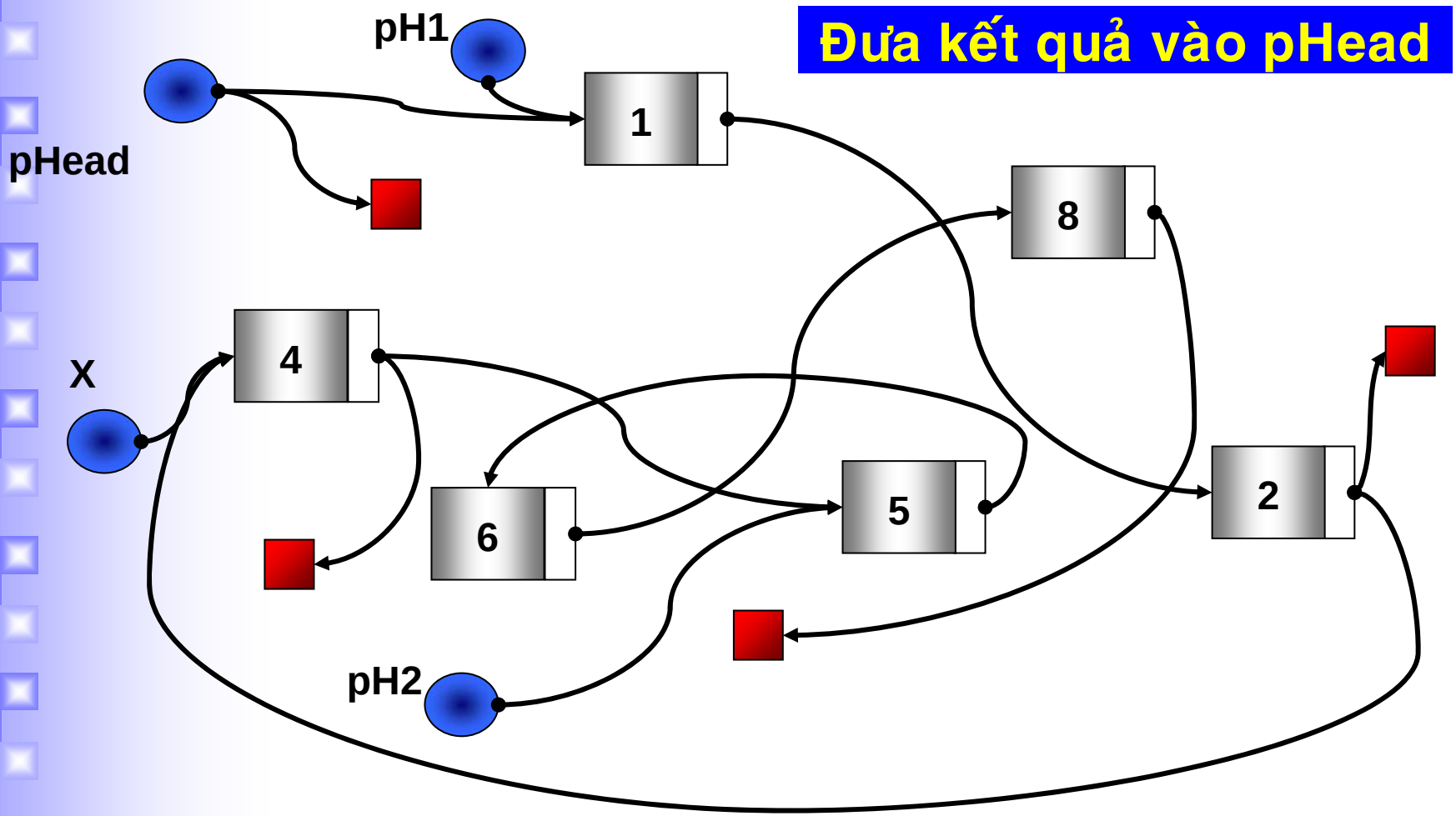
Tách xâu hiện hành thành 2 xâu

SList – quick sort



Sắp xếp các xâu pH1, pH2

SList – quick sort



SList – Nối 2 danh sách

```
void  SListAppend(SLIST &list, LIST &list2)
{
   if (list2.pHead == NULL) return;
   if (list.pHead == NULL)
       list = list2;
   else {
       list.pTail->pNext = list2.pHead;
       list.pTail = list2.pTail;
   }
   Init(list2);
}
```

```
void  SListQSort(SLIST &list) {
   NODE *X, *p;
   SLIST list1, list2;
   if (list.pHead == list.pTail) return;
   Init(list1);  Init(list2);
   X = PickHead(list);
   while (list.pHead != NULL) {
      p = PickHead(list);
      if (p->Info <= X->Info) AddTail(list1, p);
      else       AddTail(list2, p);
   }
   SListQSort(list1);  SListQSort(list2);
   SListAppend(list, list1);
   AddTail(list, X);
   SListAppend(list, list2);
}
```

```
MYLIST:
void  QuickSort() {
   NODE X, p;
   MYLIST list1, list2;
   if (Head == Tail) return;
   X = PickHead();
   while (Head != NULL) {
      p = PickHead();
      if (p.Info <= X.Info)      list1.AddTail(p);
      else                       list2.AddTail(p);
   }
   list1.QuickSort();  list2.QuickSort();
   Append(list1);
   AddTail(X);
   Append(list2);
}
```

SList – Quick sort: nhận xét

- **Nhận xét:**
  - Quick sort trên xâu đơn đơn giản hơn phiên bản của nó trên mảng một chiều
  - Khi dùng quick sort sắp xếp một xâu đơn, chỉ có một chọn lựa phần tử cầm canh duy nhất hợp lý là phần tử đầu xâu. Chọn bất kỳ phần tử nào khác cũng làm tăng chi phí một cách không cần thiết do cấu trúc tự nhiên của xâu.

SList – Merge sort: thuật toán

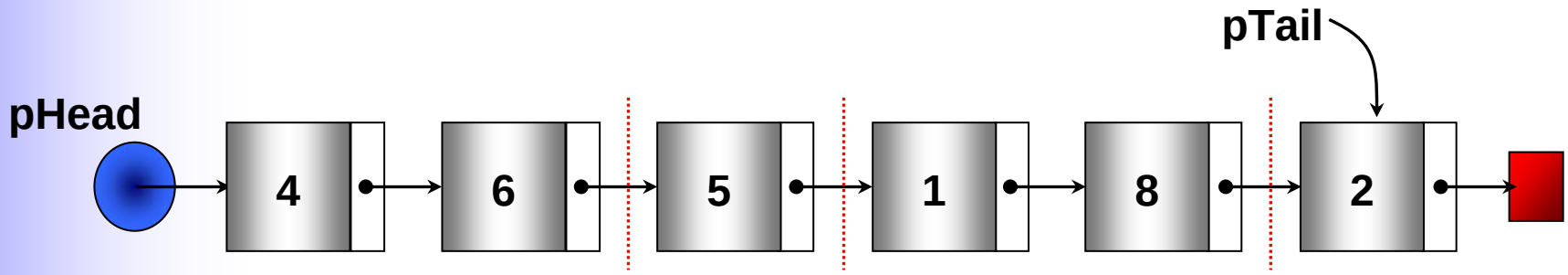
//input: xâu L

//output: xâu L đã được sắp tăng dần

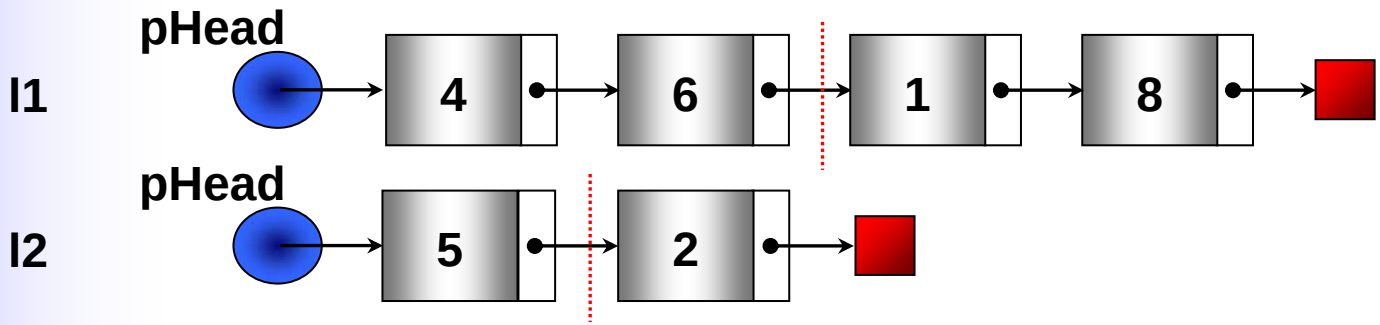
- Bước 1: Nếu xâu có ít hơn 2 phần tử

  Dừng; *//Xâu đã có thứ tự*
- Bước 2: Phân phối luân phiên từng đường chạy của xâu L vào 2 xâu con L1 và L2.
- Bước 3: Nếu L2 không rỗng
  - B31: Sắp xếp Merge Sort (L1).
  - B32: Sắp xếp Merge Sort (L2).
  - B33: Trộn L1 và L2 đã sắp xếp lại ta có xâu L đã được sắp xếp.
- Bước 4: Ngược lại: L = L1;

SList – Merge sort: Ví dụ

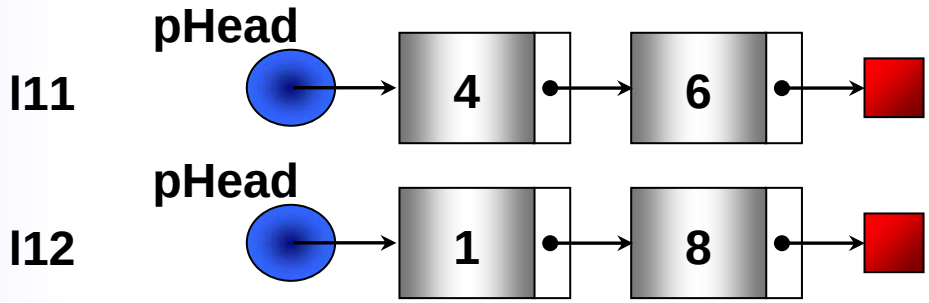


- Phân phối các đường chạy của l vào l1, l2:

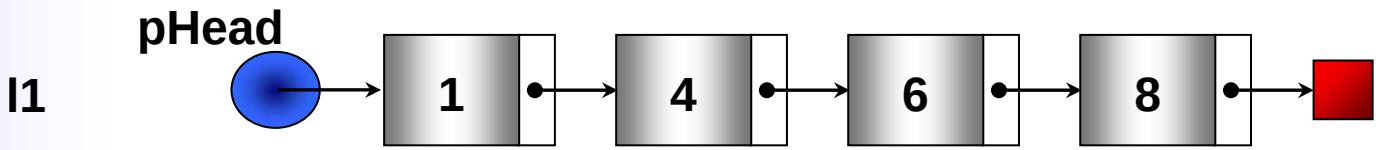


SList – Merge sort: Ví dụ

- Sắp xếp l1:
  - Phân phối các đường chạy của l1 vào l11, l12:

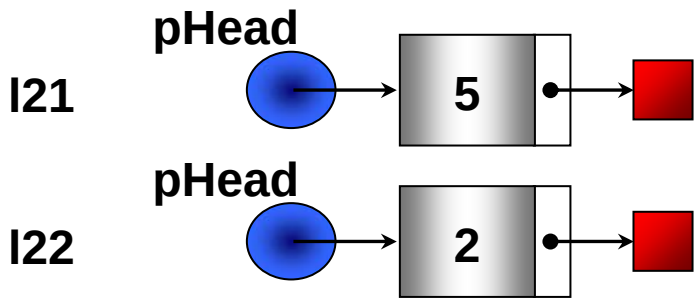


  - Trộn l11, l12 lại thành l1:

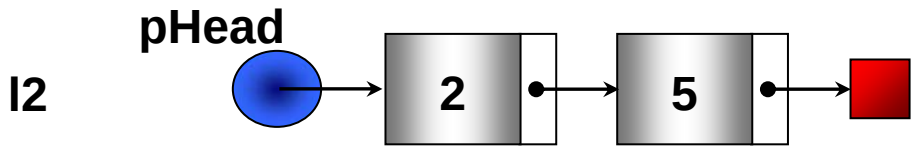


SList – Merge sort: Ví dụ

- Sắp xếp l2:
  - Phân phối các đường chạy của l2 vào l21, l22:

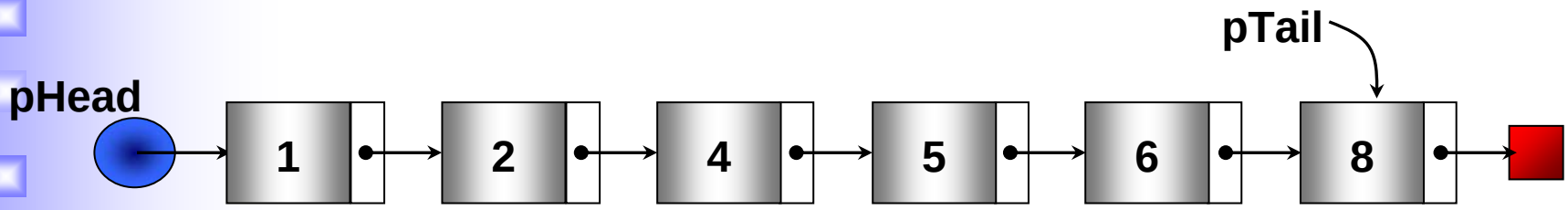


  - Trộn l21, l22 lại thành l2:



SList – Merge sort: Ví dụ

- Trộn l1, l2 lại thành l:



SList – Merge sort: cài đặt

```
void  SListMergeSort(SLIST & list)
{
   SLIST    l1, l2;
   if (list.pHead == list.pTail) //đã có thứ tự
      return;
   Init(l1); Init(l2);
   //Phân phối list thành l1 và l2 theo từng đường chạy
   SListDistribute(list, l1, l2);
   if (l2.pHead) {
      SListMergeSort(l1);      //Gọi đệ qui để sort l1
      SListMergeSort(l2);      //Gọi đệ qui để sort l2
      //Trộn l1 và l2 đã có thứ tự thành list
      SListMerge(list, l1, l2);
   }
   else     SListAppend(list, l1);
}
```

SList – Merge sort: cài đặt

```
void SListDistribute(SLIST &list,
                     SLIST &l1, SLIST &l2)
{
   NODE *p = PickHead(list);
   AddTail(l1, p);
   if (list.pHead)
      if (p->Info <= list.pHead->Info)
         SListDistribute(list, l1, l2);
      else
         SListDistribute(list, l2, l1);
}
```

SList – Merge sort: cài đặt

```
void SListMerge(SLIST& list,
                SLIST& l1, SLIST& l2)
{
    NODE     *p;
    while ((l1.pHead) && (l2.pHead))
    {
        if(l1.pHead->Info <= l2.pHead->Info)
            p = PickHead(l1);
        else
            p = PickHead(l2);
        AddTail(list, p);
    }
    SListAppend(list, l1);
    SListAppend(list, l2);
}
```

SList – Merge sort: cài đặt

```
MYLIST:
void  MergeSort()
{
   MYLIST   l1, l2;
   if (Head == Tail) //đã có thứ tự
      return;
   //Phân phối list thành l1 và l2 theo từng đường chạy
   Distribute(l1, l2);
   if (l2.pHead) {
      l1.MergeSort();    //Gọi đệ qui để sort l1
      l2.MergeSort();    //Gọi đệ qui để sort l2
      //Trộn l1 và l2 đã có thứ tự thành list
      Merge(l1, l2);
   }
   else      Append(l1);
}
```

SList – Merge sort: cài đặt

```
MYLIST:
void Distribute(ref MYLIST l1, ref MYLIST l2)
{
   NODE p = PickHead();
   l1.AddTail(p);
   if (Head)
      if (p.Info <= Head.Info)
         Distribute(l1, l2);
      else
         Distribute(l2, l1);
}
```

SList – Merge sort: cài đặt

```
MYLIST:
void Merge(ref MYLIST l1, ref MYLIST l2)
{
    NODE     p;
    while ((l1.pHead) && (l2.pHead))
    {
        if(l1.pHead.Info <= l2.pHead.Info)
            p = PickHead(l1);
        else
            p = PickHead(l2);
        AddTail(p);
    }
    Append(l1);
    Append(l2);
}
```

SList – Merge sort: nhận xét

- **Nhận xét:**
  - Merge sort trên xâu đơn giản hơn phiên bản trên mảng một chiều.
  - Khi dùng Merge sort sắp xếp một xâu đơn, không cần dùng thêm vùng nhớ phụ như khi cài đặt trên mảng một chiều.
  - Thủ tục Merge trên xâu không phức tạp như trên mảng vì chỉ phải trộn hai xâu đã có thứ tự, trong khi trên mảng phải trộn hai mảng bất kỳ.

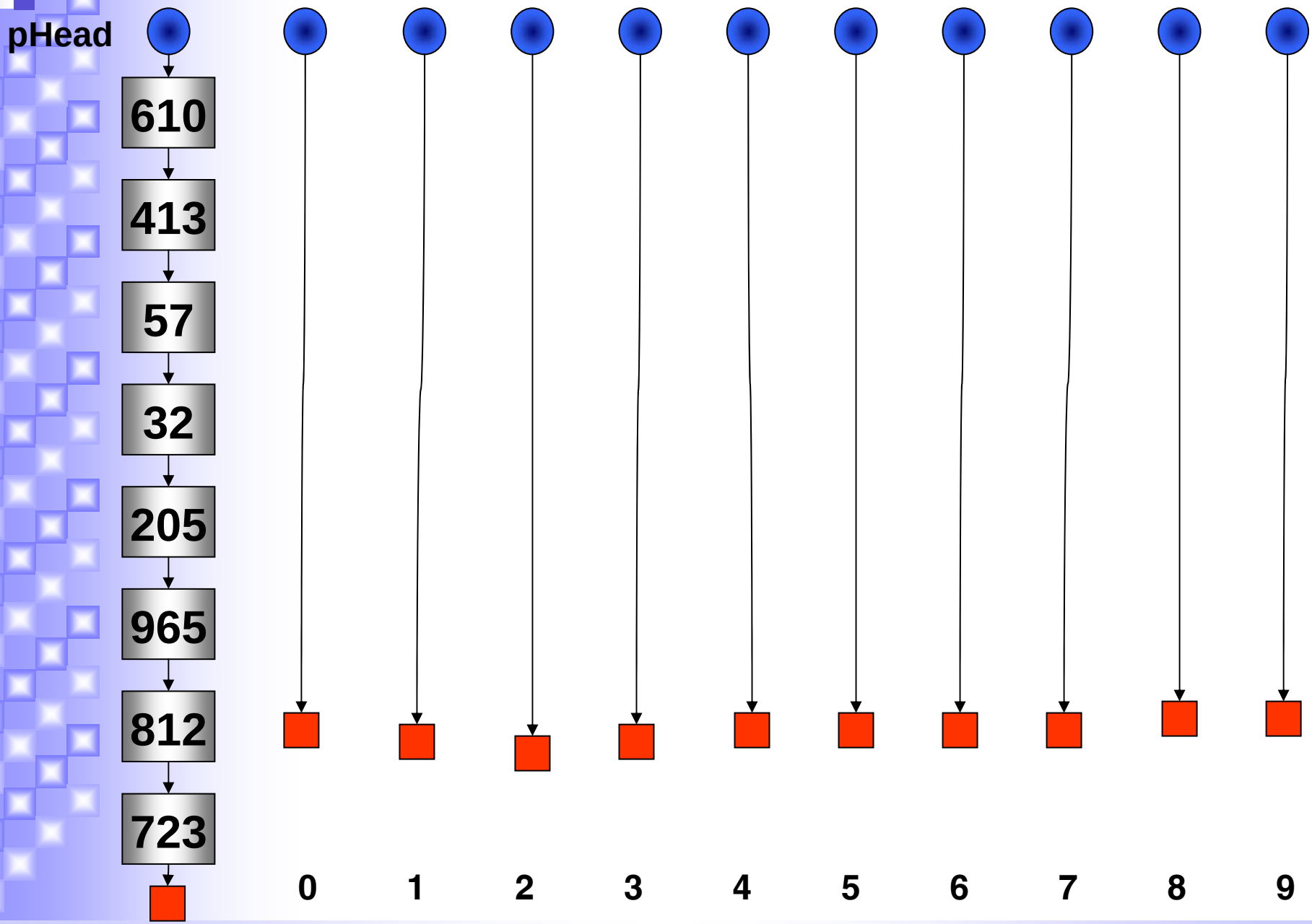
SList – Radix sort: thuật toán

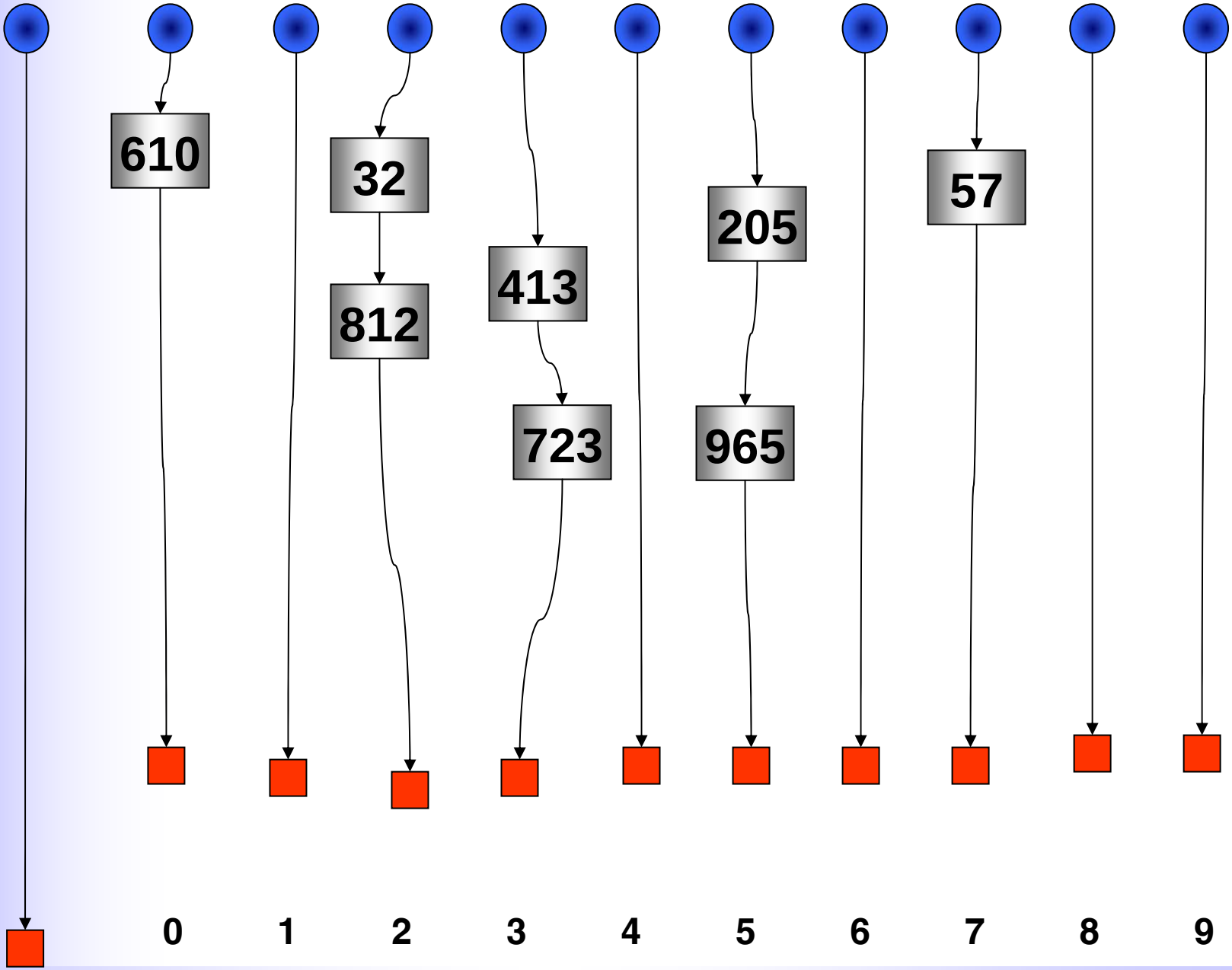
//input: xâu L

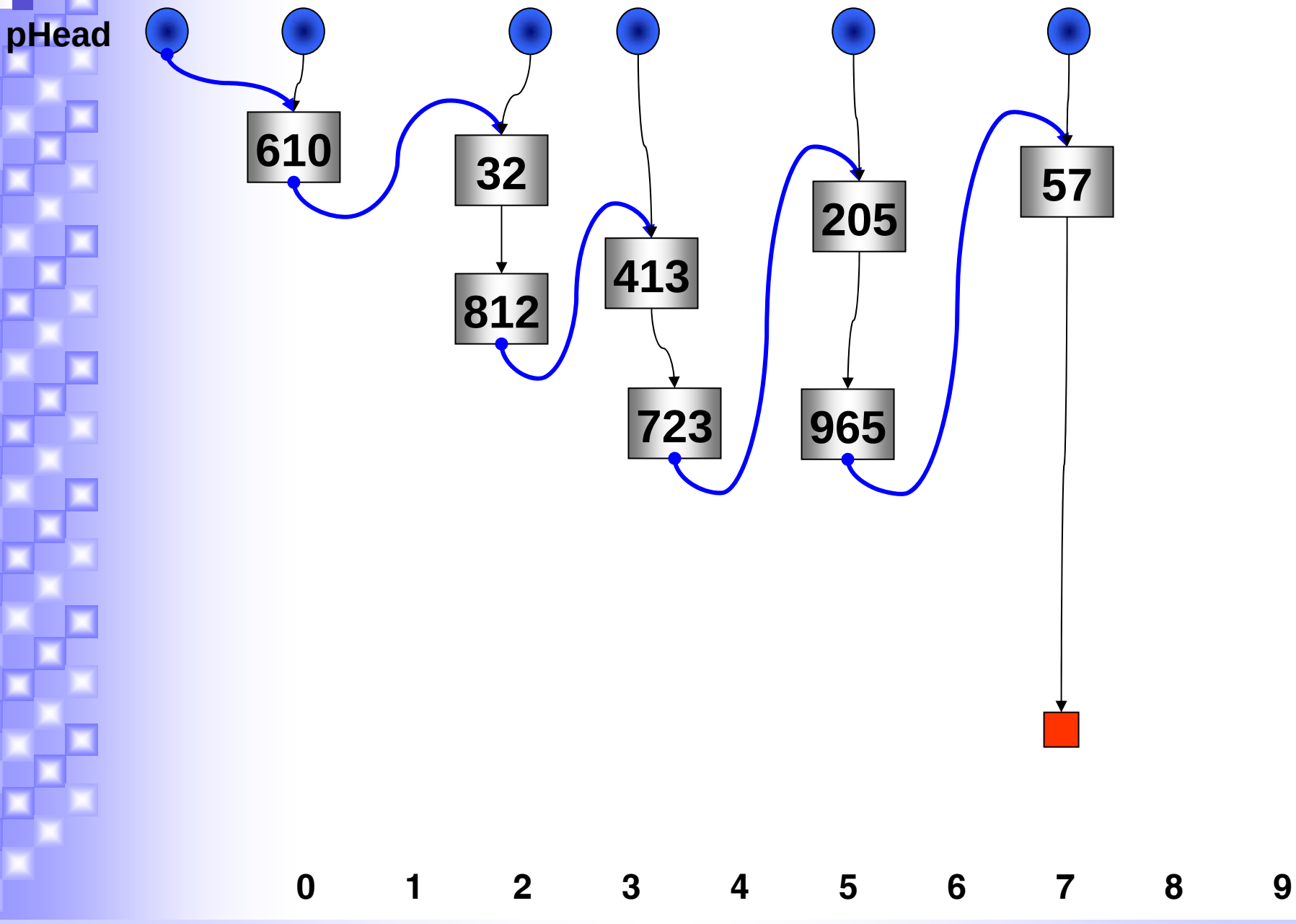
//output: xâu L đã được sắp tăng dần

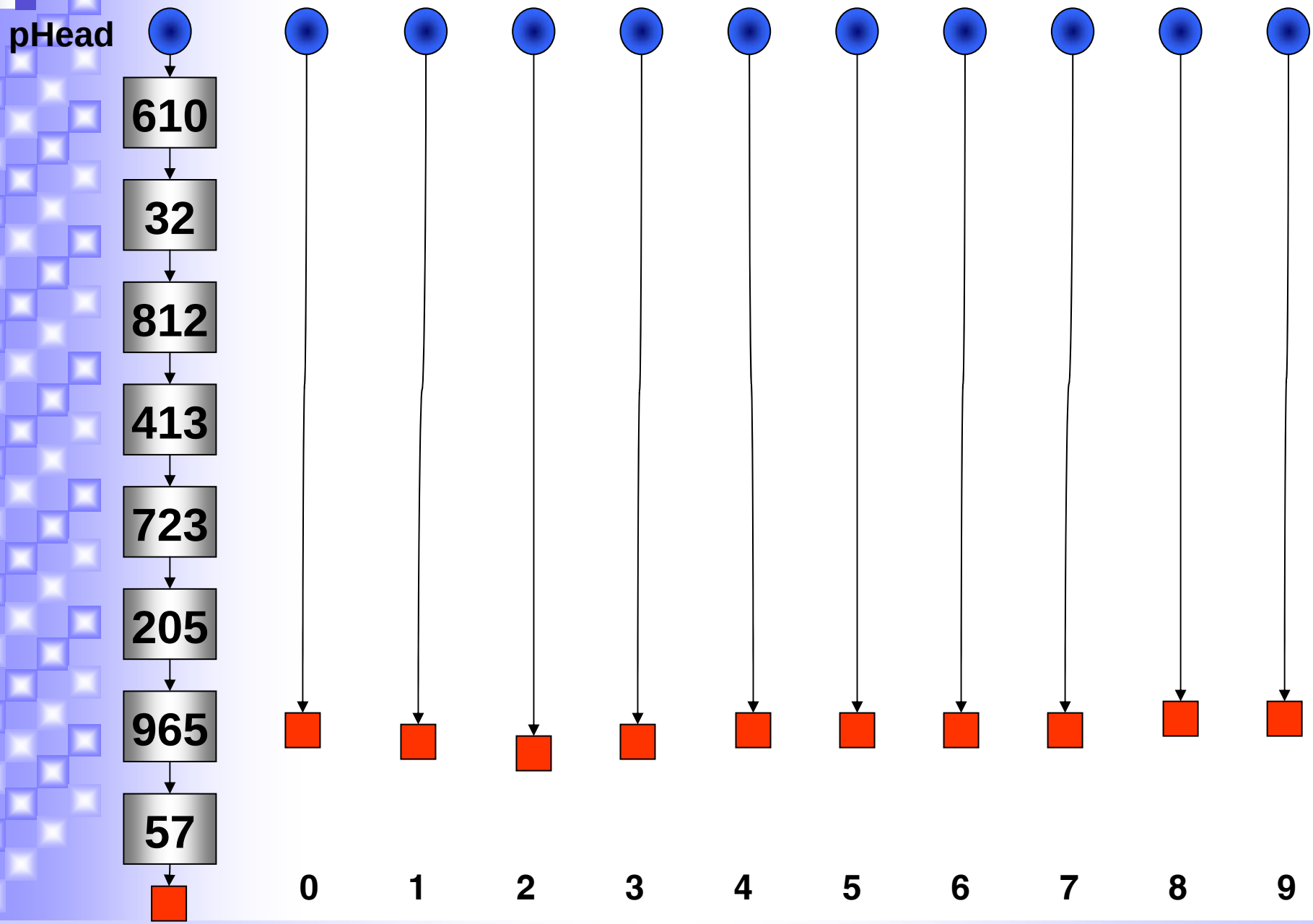
//dữ liệu trung gian: mười danh sách rỗng $B_0, B_1, \ldots, B_9$

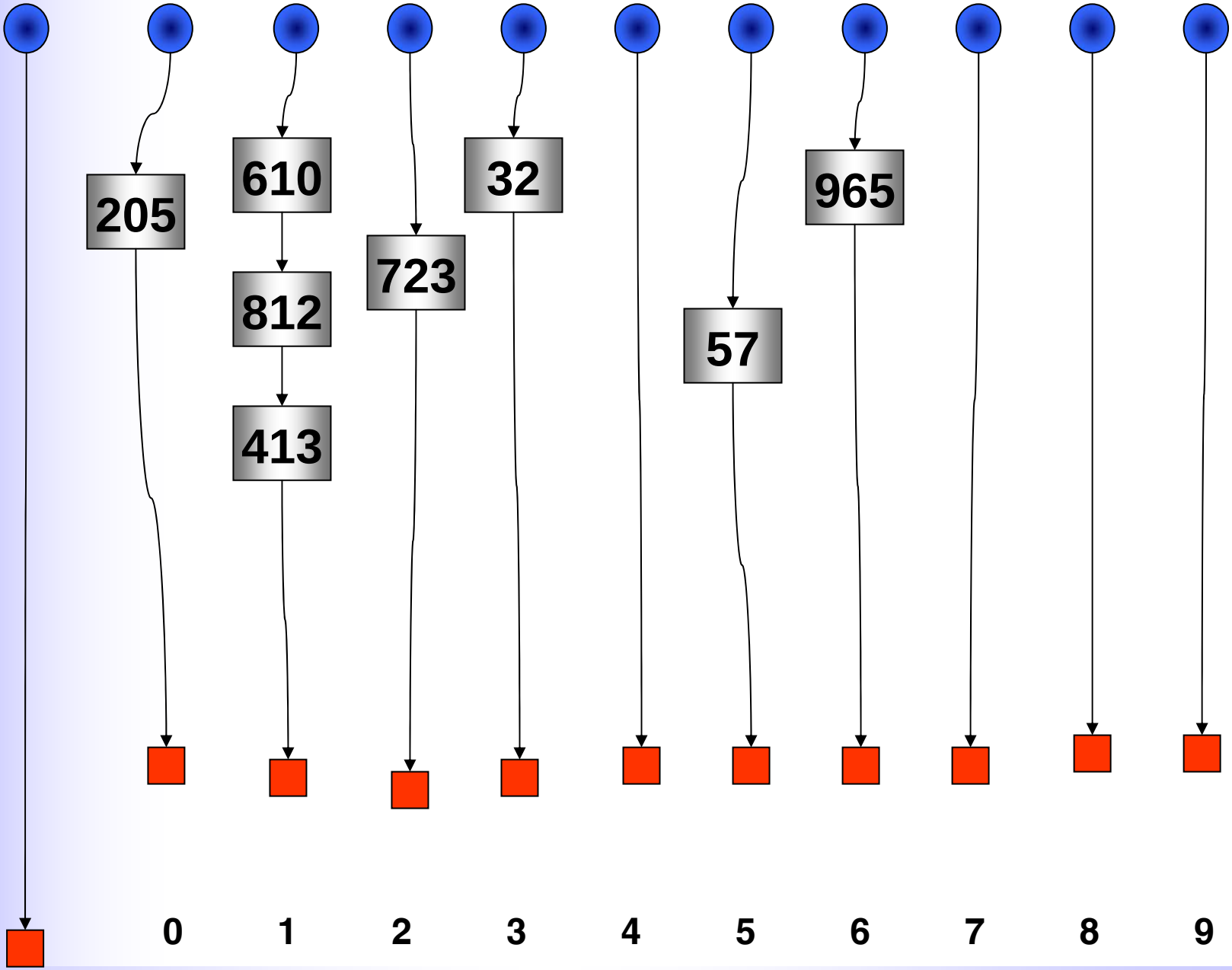
- B1: + Khởi tạo các danh sách (lô) rỗng $B_0, B_1, \ldots, B_9$;
  + k = 0;
- B2: Trong khi L khác rỗng:
  + p ← phần tử đầu xâu của L;
  + Nối p vào cuối danh sách B_d với d là chữ số thứ k của p;
- B3: Nối $B_0, B_1, \ldots, B_9$ lại thành L;
- B4: + k = k+1;
  + Nếu k < m: quay lại Bước 2

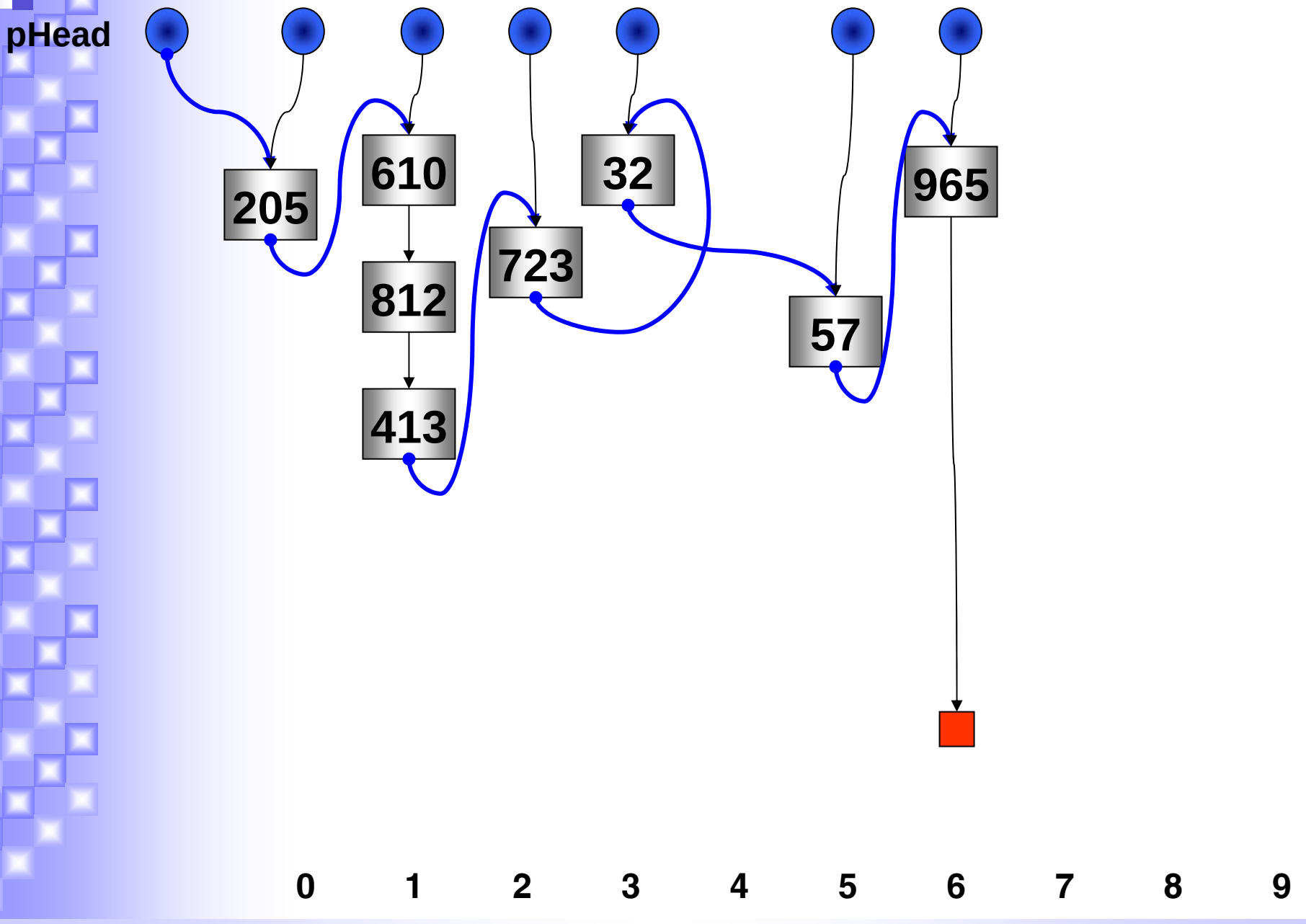

pHead
610
413
57
32
205
965
812
723
0
1
2
3
4
5
6
7
8
9

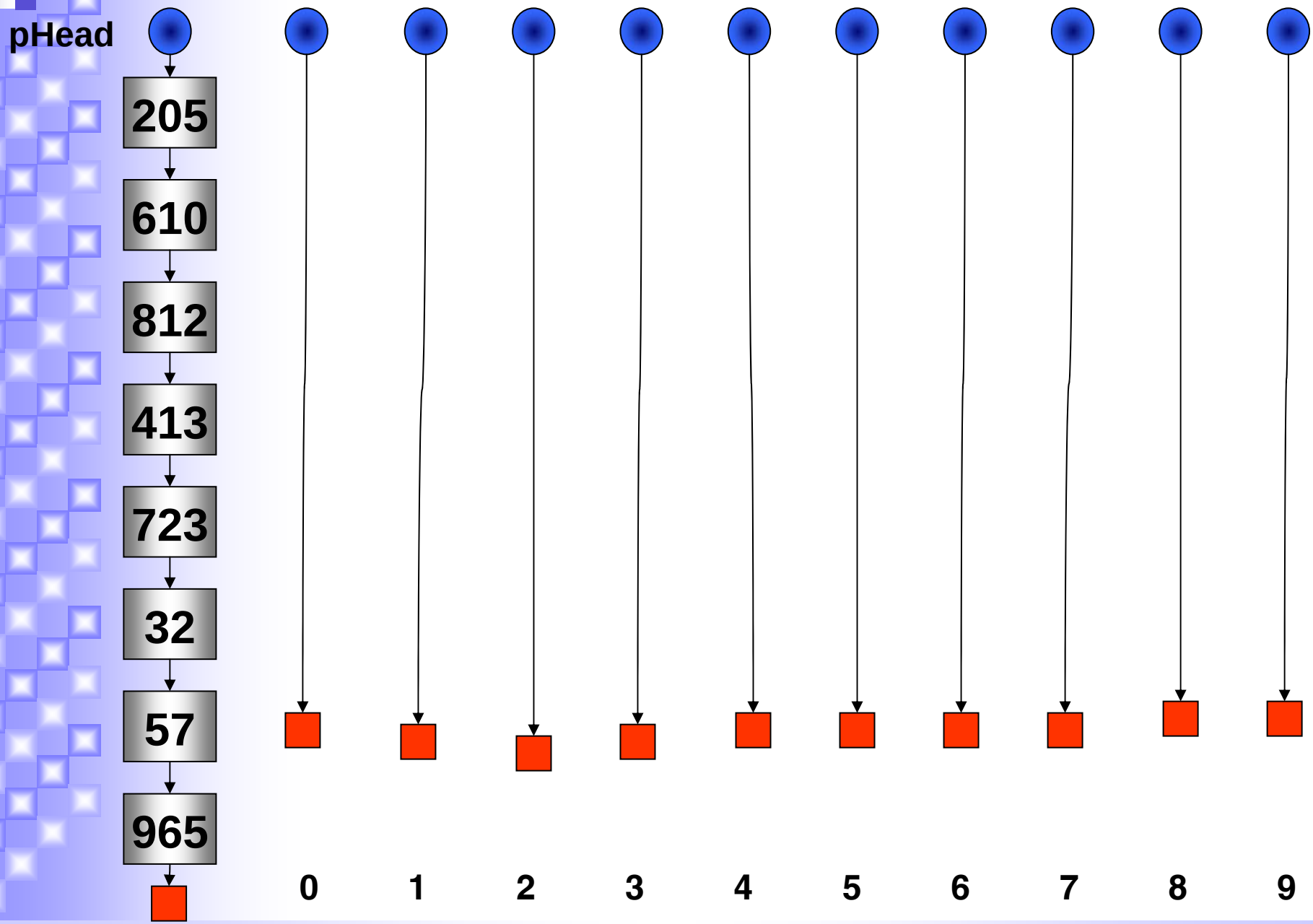

pHead
610
32
812
413
723
205
965
57
0
1
2
3
4
5
6
7
8
9

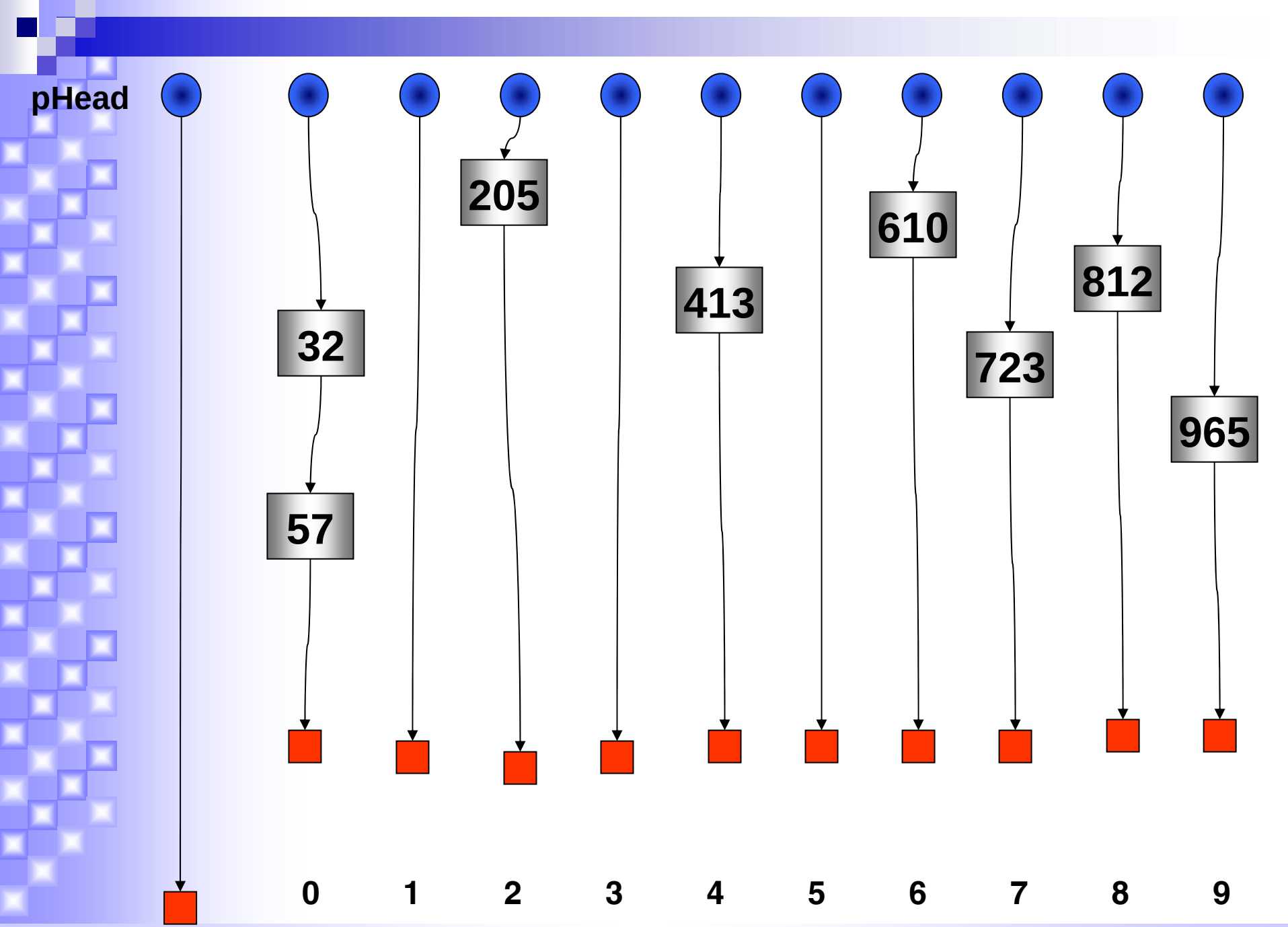

pHead
610
32
812
413
723
205
965
57
0
1
2
3
4
5
6
7
8
9

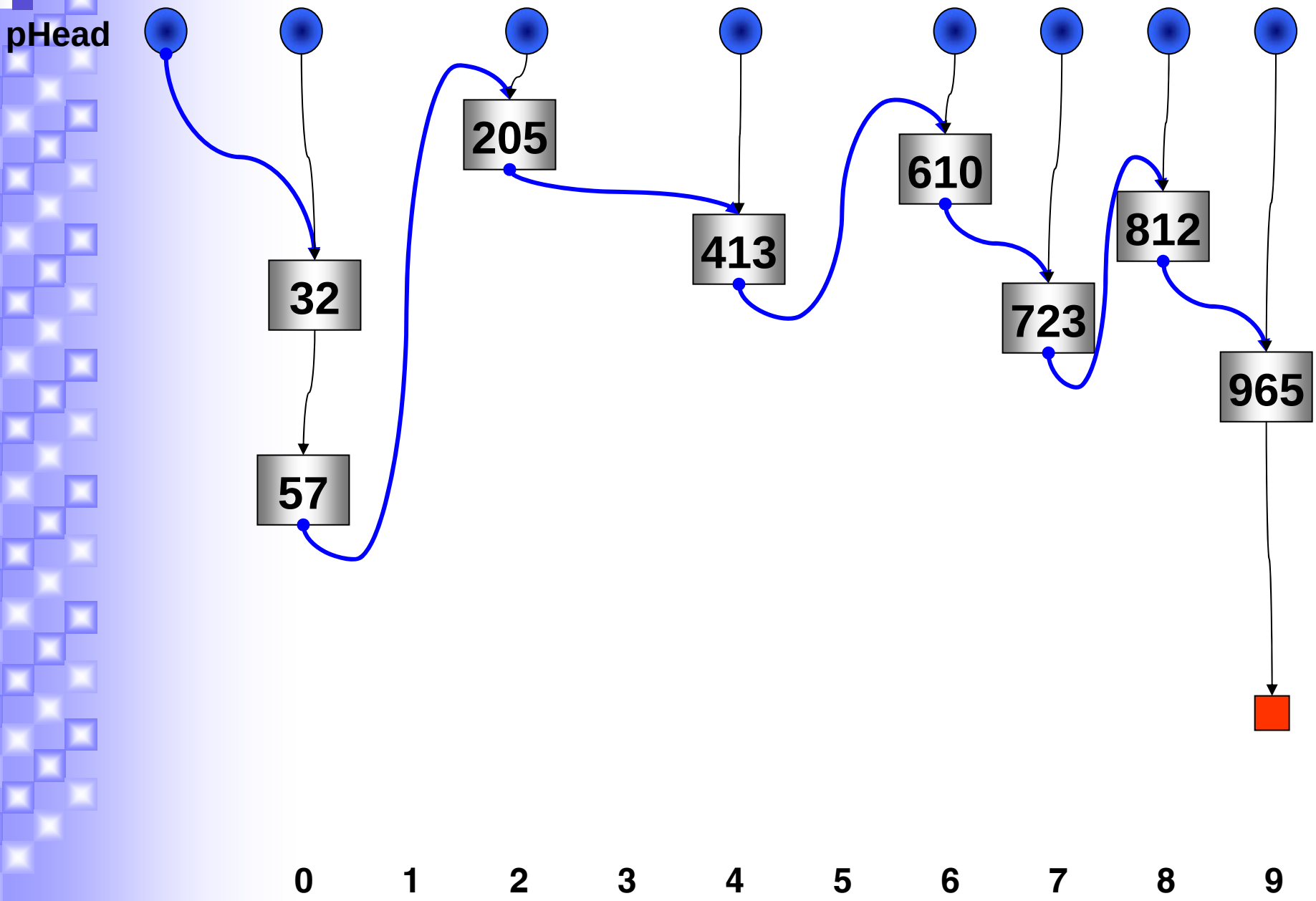

pHead
610
32
812
413
723
205
965
57
0
1
2
3
4
5
6
7
8
9


pHead
205
610
812
413
723
32
57
965
0
1
2
3
4
5
6
7
8
9


pHead
205
610
812
413
723
32
57
965
0
1
2
3
4
5
6
7
8
9


pHead
205
610
812
413
723
32
57
965
0
1
2
3
4
5
6
7
8
9


pHead
32
57
205
413
610
723
812
965
0
1
2
3
4
5
6
7
8
9


pHead
32
57
205
413
610
723
812
965
0
1
2
3
4
5
6
7
8
9

SList – Radix sort: cài đặt

```
int M = 10; //số chữ số tối đa của một phần tử
void  SListRadixSort(SLIST & list)
{
   SLIST B[10];   NODE *p;    int k;
   if (list.pHead == list.pTail) return;//đã có thứ tự
   for (i = 0; i < 10; i++)
      Init(B[i]);
   for (k = 0; k < M; k++) {
      while(list.pHead) {
         p = PickHead(list);
         AddTail(B[GetDigit(p->Info, k)], p);
      }
      for(i = 0; i < 10; i++)
         SListAppend(list, B[i]); //Nối B[i] vào cuối l
   }
}
```

SList – Radix sort: cài đặt

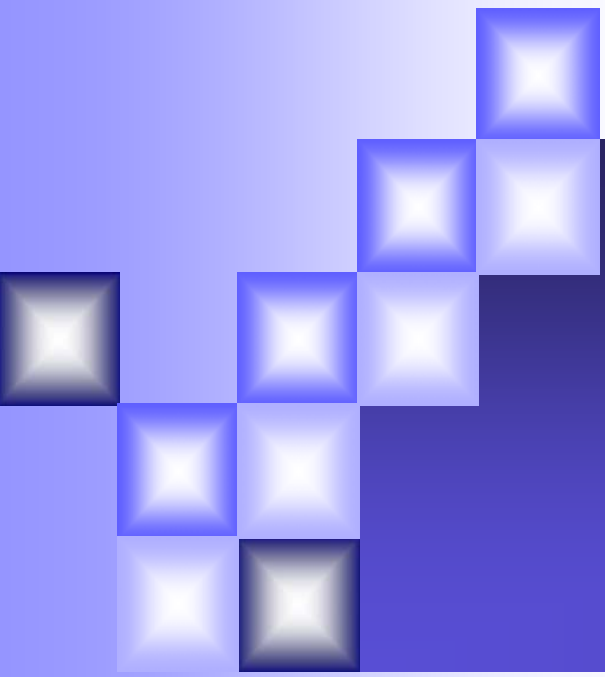
```
unsigned long base[10] = {
      1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000,
      100000000, 10000000000};

int GetDigit(unsigned long N, int k)
{
   return ((N / base[k]) % 10);
}
```

SList – Radix sort: nhận xét

- **Nhận xét:**
  - Radix sort trên xâu đơn giản hơn phiên bản trên mảng một chiều.
  - Khi dùng Radix sort sắp xếp một xâu đơn, không cần dùng thêm vùng nhớ phụ như khi cài đặt trên mảng một chiều.
  - Không có thao tác chép dữ liệu, chỉ có thay đổi các mối liên kết.

Một số cấu trúc dữ liệu trừu tượng

- Stack
- Hàng đợi (Queue)



Stack

Stack

- Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế LIFO (*Last In First Out*) ⇒ Việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "*Vào sau ra trước*".
- Các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.
- "Push": Thao tác thêm 1 đối tượng vào stack
- "Pop": Thao tác lấy 1 đối tượng ra khỏi stack.
- Stack có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, ứng dụng trong các bài toán tính toán biểu thức, …

Stack

- Stack là một CTDL trừu tượng (ADT) tuyến tính 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 dùng mảng

- Có thể tạo một stack bằng cách khai báo một mảng 1 chiều với kích thước tối đa là N (ví dụ: N =1000).
- Stack có thể chứa tối đa N phần tử đánh số từ 0 đến N-1.
- Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử)
- Để khai b
  nguyên t
  biết kích thước tối đa của stack.


```
Data    S [N];
int     t;
```

Biểu diễn Stack dùng mảng

- Lệnh t = 0 sẽ tạo ra một stack S rỗng.
- Giá trị của t sẽ cho biết số phần tử hiện hành có trong stack.
- Khi cài đặt bằng mảng 1 chiều, stack có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho stack:
  - `IsFull`(): Kiểm tra xem stack có đầy chưa.
  - Khi stack đầy, việc gọi đến hàm push() sẽ phát sinh ra lỗi.

Biểu diễn Stack dùng mảng

- Kiểm tra stack rỗng hay không

```
char IsEmpty()
{ if(t == 0) // stack rỗng
    return 1;
  else
    return 0;
}
```

- Kiểm tra stack đầy hay không

```
char IsFull()
{ if(t >= N) // stack đầy
    return 1;
  else
    return 0;
}
```

Biểu diễn Stack dùng mảng

- Thêm một phần tử x vào stack S

```
void Push(Data x)
{ if(t < N) // stack chưa đầy
  {S[t] = x; t++; }
  else puts("Stack đầy");
}
```

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

```
Data Pop()
{ Data    x;
  if(t > 0) // stack khác rỗng
  { t--; x = S[t]; return x;}
  else puts("Stack rỗng")
}
```

Biểu diễn Stack dùng mảng

- Xem thông tin của phần tử ở đỉnh stack S

```
Data Top()
{ Data    x;
  if(t > 0) // stack khác rỗng
  { x = S[t-1];
    return   x;
  }
  else  puts("Stack rỗng")
}
```

Biểu diễn Stack dùng mảng

- **Nhận xét:**
  - Các thao tác trên đều làm việc với chi phí O(1).
  - Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả.
  - Tuy nhiên, hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack N. Giá trị của N có thể quá nhỏ so với nhu cầu thực tế hoặc quá lớn sẽ làm lãng phí bộ nhớ.

Biểu diễn Stack dùng danh sách liên kết

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

```
SLIST * S;
```

Biểu diễn Stack dùng danh sách liên kết

- Tạo Stack S rỗng:

```
S.pHead=l.pTail= NULL;
```

- Kiểm tra stack rỗng :

```
char  IsEmpty(LIST &S)
{ if (S.pHead == NULL) // stack rỗng
    return 1;
  else
    return 0;
}
```

Biểu diễn Stack dùng danh sách liên kết

- Thêm một phần tử x vào stack S

```
void      Push(LIST &S, Data x)
{ InsertHead(S, x); }
```

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

```
Data      Pop(LIST &S)
{ Data    x;
  if(isEmpty(S))      return NULLDATA;
  x = RemoveFirst(S); return x;
}
```

- Xem thông tin của phần tử ở đỉnh stack S

```
Data      Top(LIST &S)
{ if(isEmpty(S)) return NULLDATA;
  return l.Head->Info;
}
```

Ứng dụng của Stack

- Stack thích hợp lưu trữ các loại dữ liệu mà trình tự truy xuất ngược với trình tự lưu trữ
- Một số ứng dụng của Stack:
  - Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, Stack được sử dụng để lưu môi trường của các thủ tục.
  - Lưu dữ liệu khi giải một số bài toán của lý thuyết đồ thị (như tìm đường đi)
  - Khử đệ qui
  - ...

Ứng dụng của Stack

Ví dụ: thủ tục Quick_Sort dùng Stack để khử đệ qui:

- 1. left:=1; r:=n;
- 2. Chọn phần tử giữa x:=a[(left+r) div 2];
- 3. Phân hoạch (left,r) thành (left1,r1) và (left2,r2) bằng cách xét:
  - y thuộc (left1,r1) nếu y≤x;
  - y thuộc (left2,r2) ngược lại;
- 4. Nếu phân hoạch (left2,r2) có nhiều hơn 1 phần tử thực hiện:
  - Cất (left2,r2) vào Stack;
  - Nếu (left1,r1) có nhiều hơn 1 phần tử thực hiện:
    - left=left1;
    - r=r1;
    - Goto 2;
  - Ngược lại
    - Lấy (left,r) ra khỏi Stack nếu Stack khác rỗng và Goto 2;
    - Nếu không dừng;

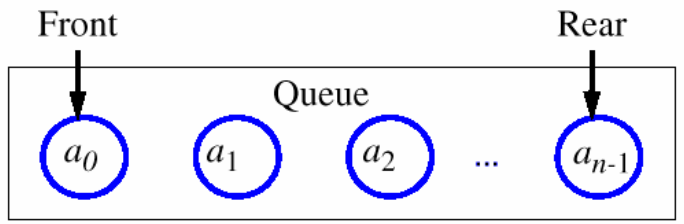
Hàng đợi (Queue)

Hàng đợi (Queue)

- Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế FIFO (*First In First Out*) ⇒ việc thêm một đối tượng vào hàng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được thực hiện theo cơ chế "*Vào trước ra trước*".
- Các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi.
- "**Enqueue**":Thao tác thêm một đối tượng vào hàng đợi
- "**Dequeue**":Thao tác lấy một đối tượng ra khỏi hàng đợi.

Hàng đợi (Queue)

- Việc thêm một đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



- Trong tin học, CTDL hàng đợi có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui, vét cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, …

Hàng đợi (Queue)

- Hàng đợi là một CTDL trừu tượng (ADT) tuyến tính
- Hàng đợi hỗ trợ các thao tác:
  - **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.
  - **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 Queue dùng mảng

- Có thể tạo một hàng đợi bằng cách sử dụng một mảng 1 chiều với kích thước tối đa là N (ví dụ, N=1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0) $\Rightarrow$ Hàng đợi chứa tối đa N phần tử.
- Phần tử ở đầu hàng đợi (front element) sẽ có chỉ số f.
- Phần tử ở cuối hàng đợi (rear element) sẽ có chỉ số r .

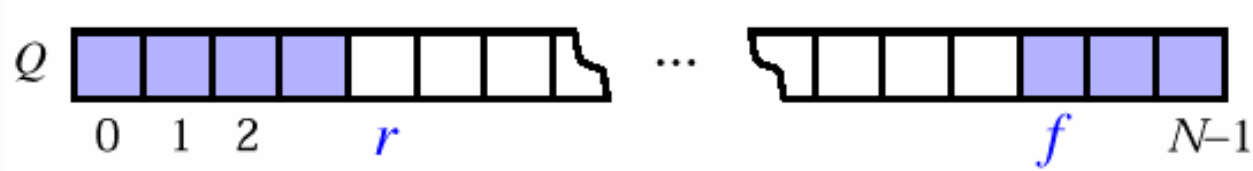
Biểu diễn Queue dùng mảng

- Để khai báo một hàng đợi, ta cần:
  - một mảng một chiều Q,
  - hai biến nguyên f, r cho biết chỉ số của đầu và cuối của hàng đợi
  - hằng số N cho biết kích thước tối đa của hàng đợi.
- Ngoài ra, khi dùng mảng biểu diễn hàng đợi, cần dùng một giá trị đặc biệt, ký hiệu là NULLDATA, để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu lưu trong hàng đợi..

Biểu diễn Queue dùng mảng

- Trạng thái hàng đợi lúc bình thường:

- Trạng thái hàng đợi lúc xoay vòng:



Biểu diễn Queue dùng mảng

- Khi giá trị f=r cho ta điều gì?
- Lúc này hàng đợi chỉ có thể ở một trong hai trạng thái là rỗng hoặc đầy.
- Coi như một bài tập các bạn hãy tự suy nghĩ tìm câu trả lời trước khi đọc tiếp để kiểm tra kết quả.

Biểu diễn Queue dùng mảng

- Hàng đợi có thể được khai báo cụ thể như sau:

```
Data      Q[N] ;
int       f, r;
```

- Do khi cài đặt bằng mảng một chiều, hàng đợi có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho hàng đợi:
  - `IsFull`(): Kiểm tra xem hàng đợi có đầy chưa.

Biểu diễn Queue dùng mảng

- Tạo hàng đợi rỗng

```
void   InitQueue()
{
   f = r = 0;
   for(int i = 0; i < N; i++)
      Q[i] = NULLDATA;
}
```

Biểu diễn Queue dùng mảng

- Kiểm tra hàng đợi rỗng hay không

```
char  IsEmpty()
{
  return (Q[f] == NULLDATA);
}
```

- Kiểm tra hàng đợi đầy hay không

```
char IsFull()
{
  return (Q[r] != NULLDATA);
}
```

Biểu diễn Queue dùng mảng

- Thêm một phần tử x vào cuối hàng đợi Q

```
char      EnQueue(Data x)
{
  if(IsFull()) return -1; //Queue đầy
  Q[r++] = x;
  if(r == N)      // xoay vòng
     r = 0;
}
```

Biểu diễn Queue dùng mảng

- Trích, huỷ phần tử ở đầu hàng đợi Q

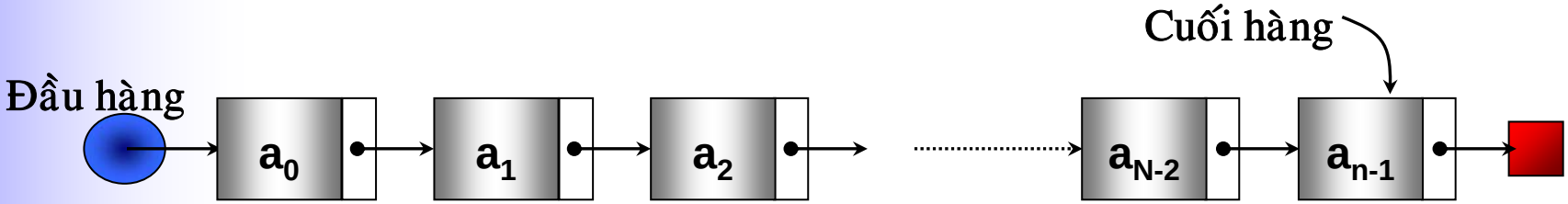
```
Data      DeQueue()
{ Data x;
  if(IsEmpty()) return NULLDATA; //Queue rỗng
  x = Q[f]; Q[f++] = NULLDATA;
  if(f == N)     f = 0; // xoay vòng
  return x;
}
```

- Xem thông tin của phần tử ở đầu hàng đợi Q

```
Data      Front()
{
  if(IsEmpty()) return NULLDATA; //Queue rỗng
  return Q[f];
}
```

Biểu diễn hàng đợi dùng danh sách liên kết

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



Biểu diễn hàng đợi dùng danh sách liên kết

- Tạo hàng đợi rỗng:

```
Q.pHead=Q.pTail= NULL;
```

- Kiểm tra hàng đợi rỗng :

```
char  IsEmpty(LIST Q)
{ if (Q.pHead == NULL) // hàng đợi rỗng
    return 1;
  else
    return 0;
}
```

Biểu diễn hàng đợi dùng danh sách liên kết

- Thêm một phần tử p vào cuối hàng đợi

```
void      EnQueue(LIST Q, Data x)
{ InsertTail(Q, x);}
```

- Trích/Hủy phần tử ở đầu hàng đợi

```
Data      DeQueue(LIST Q)
{ Data    x;
  if (IsEmpty(Q))    return NULLDATA;
  x = RemoveFirst(Q);return x;
}
```

- Xem thông tin của phần tử ở đầu hàng đợi

```
Data      Front(LIST Q)
{ if (IsEmpty(Q)) return NULLDATA;
  return Q.pHead->Info;
}
```

Biểu diễn hàng đợi dùng danh sách liên kết

- **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 xâu, thao tác **Enqueue** sẽ có độ phức tạp O(n).

Ứng dụng của hàng đợi

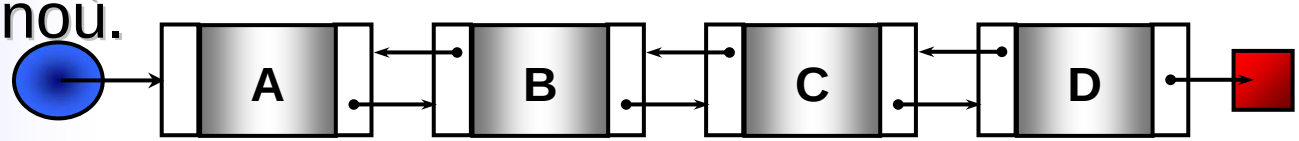
- Hàng đợi có thể được sử dụng trong một số bài toán:
  - Bài toán 'sản xuất và tiêu thụ' (ứng dụng trong các hệ điều hành song song).
  - Bộ đệm (ví dụ: Nhấn phím $\Rightarrow$ Bộ đệm $\Rightarrow$ CPU xử lý).
  - Xử lý các lệnh trong máy tính (ứng dụng trong HĐH, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, ….

Một số cấu trúc dữ liệu dạng danh sách liên kết khác

- Danh sách liên kết kép
- Hàng đợi hai đầu (double-ended queue)
- Danh sách liên kết có thứ tự (Odered List)
- Danh sách liên kết vòng
- Danh sách có nhiều mối liên kết
- Danh sách tổng quát

Danh sách liên kết kép

- Danh saùch lieân keát keùp laø danh saùch maø moãi phaàn töû trong danh saùch coù keát noái vôùi 1 phaàn töû ñöùng tröôùc vaø 1 phaàn töû ñöùng sau noù.



Danh sách liên kết kép

- Dùng hai con trỏ:
  - pPrev liên kết với phần tử đứng trước
  - pNext liên kết với phần tử đứng sau:

```
typedef   struct  tagDNode
{
   Data    Info;
   struct tagDNode* pPre;  // trỏ đến phần tử đứng trước
   struct tagDNode* pNext;// trỏ đến phần tử đứng sau
}DNODE;
typedef   struct  tagDList
{
   DNODE* pHead;   // trỏ đến phần tử đứng trước
   DNODE* pTail;   // trỏ đến phần tử đứng sau
}DLIST;
```

Danh sách liên kết kép

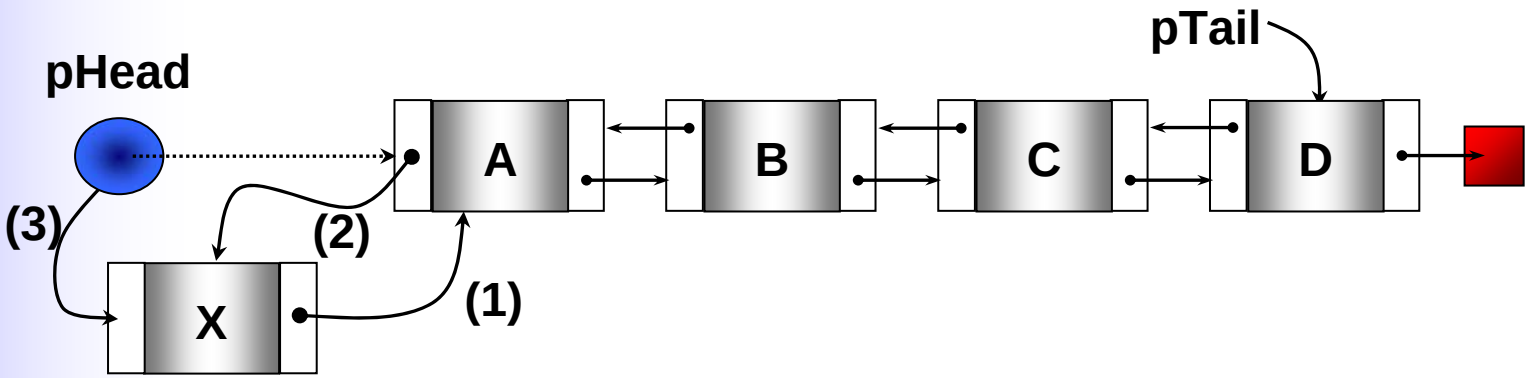
- Thủ tục khởi tạo 1 phần tử cho danh sách liên kết kép:

```
DNODE*    GetNode(Data x)
{ DNODE *p;
  // Cấp phát vùng nhớ cho phần tử
  p = new DNODE;
  if ( p==NULL)
  {   printf("Không đủ bộ nhớ");
      exit(1);
  }
  // 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

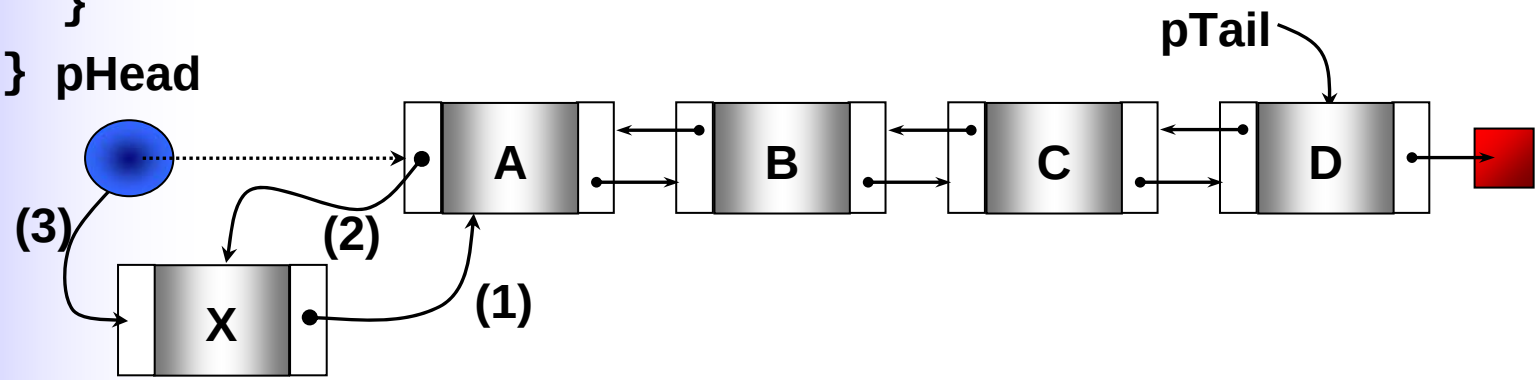
- Có 4 loại thao tác chèn new_ele vào danh sách:
  - Cách 1: Chèn vào đầu danh sách
  - Cách 2: Chèn vào cuối danh sách
  - Cách 3 : Chèn vào danh sách sau một phần tử q
  - Cách 4 : Chèn vào danh sách trước một phần tử q

Chèn một phần tử vào đầu danh sách liên kết kép



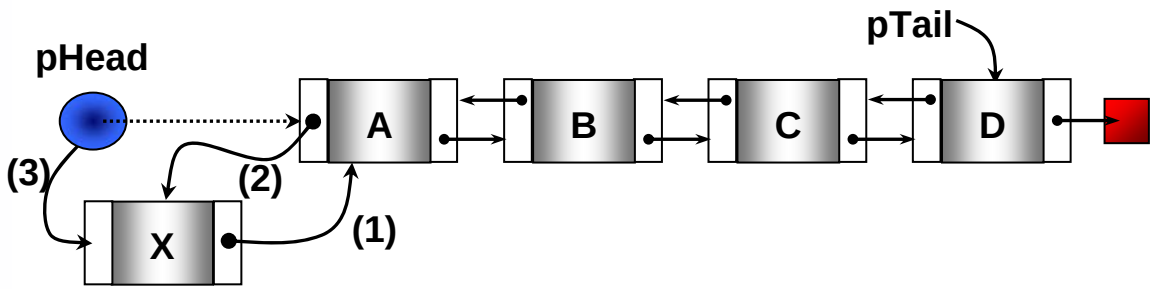
Chèn một phần tử vào đầu danh sách liên kết kép

```
void AddFirst(DLIST &l, DNODE* new_ele)
{ if (l.pHead==NULL)  //Xâu 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)
  }
} pHead
```

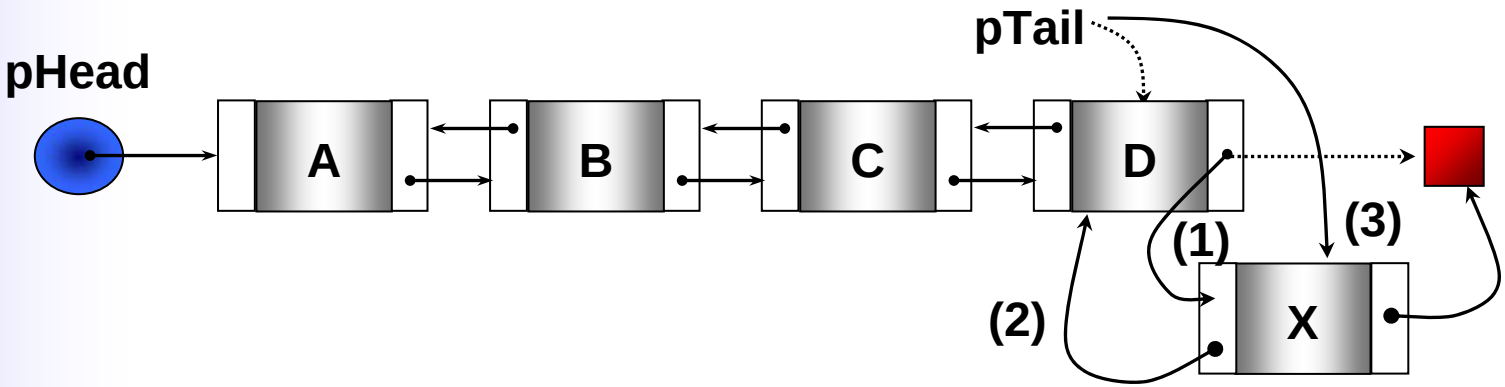


Chèn một phần tử vào đầu danh sách liên kết kép

```
NODE* InsertHead(DLIST &l, Data x)
{ NODE* new_ele = GetNode(x);
  if (new_ele ==NULL)  return NULL;
  AddFirst(l, new_ele)
  return new_ele;
}
```

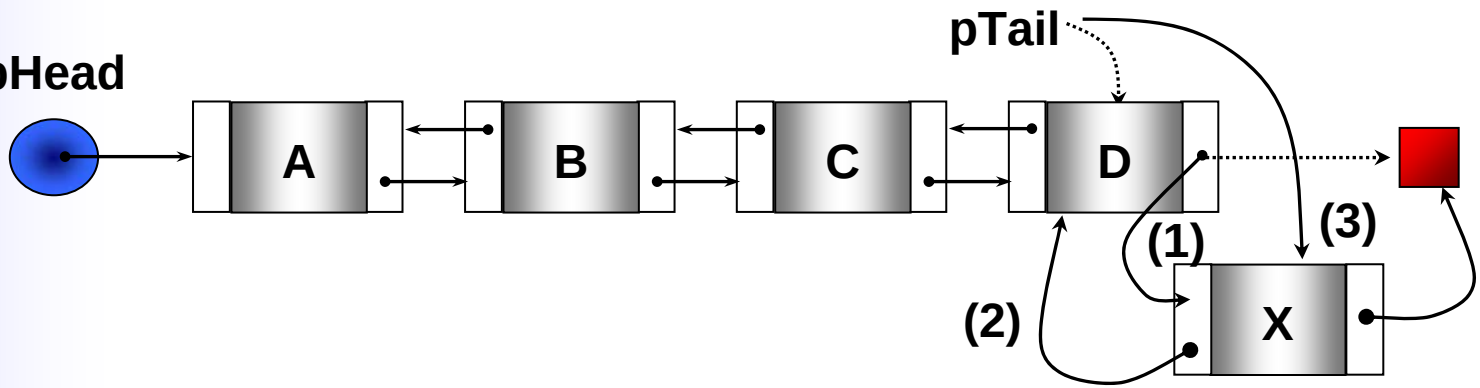


Chèn một phần tử vào đầu danh sách liên kết kép



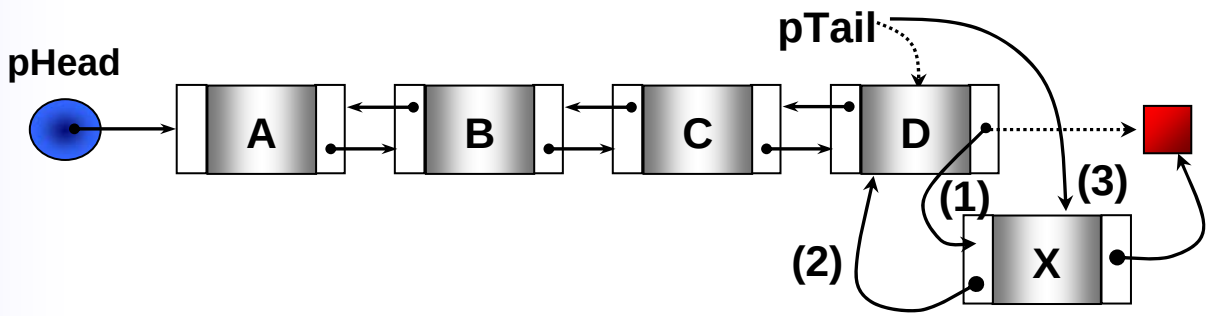
Chèn một phần tử vào đầu danh sách liên kết kép

```
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)
  }
}
```

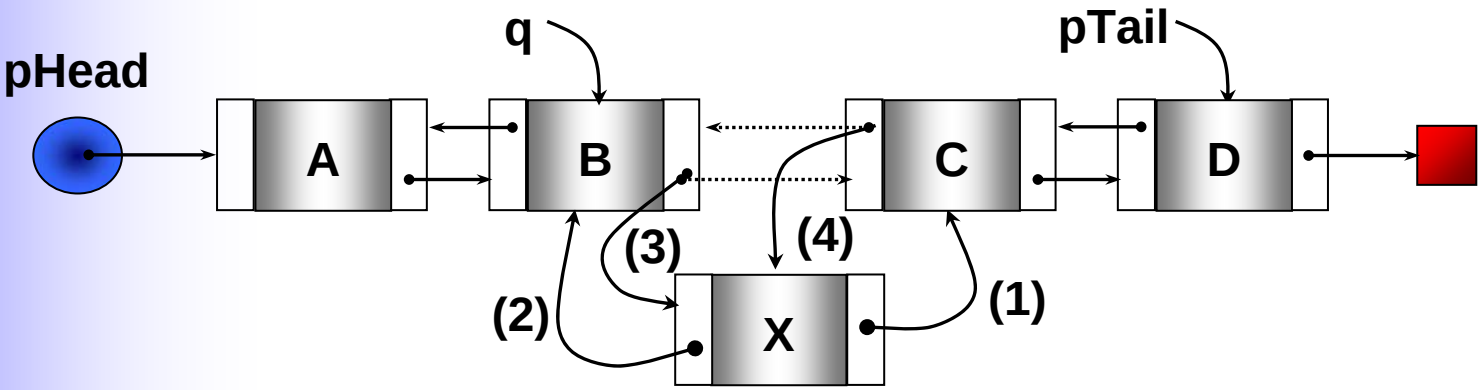


Chèn một phần tử vào cuối danh sách liên kết kép

```
NODE* InsertTail(DLIST &l, Data x)
{ NODE* new_ele = GetNode(x);
  if (new_ele ==NULL)  return NULL;
  AddTail(l, new_ele)
  return new_ele;
}
```



Chèn vào DSLK kép sau một phần tử q



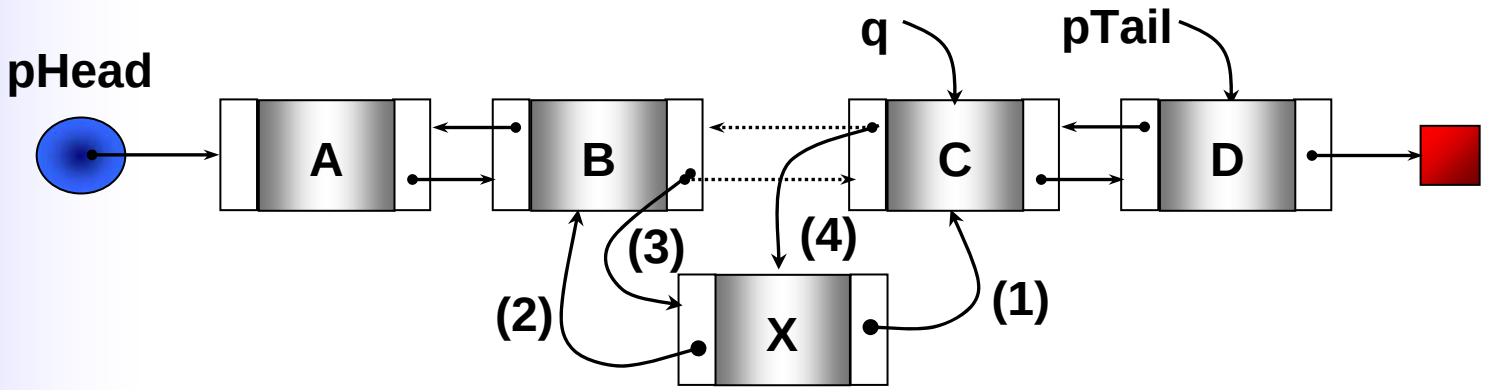
Chèn vào DSLK kép sau một phần tử q

```
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;
  }
  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)
{ NODE* new_ele = GetNode(x);
  if (new_ele ==NULL)  return NULL;
  AddAfter(l, q, 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)
{
  NODE* new_ele = GetNode(x);
  if (new_ele ==NULL)  return NULL;
  DNODE* p = q->pPrev;
  AddBefore(l, q, 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 xâu
  - Hủy phần tử cuối xâu
  - 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 xâu

```
Data RemoveHead(DLIST &l)
{ DNODE   *p;
  Data    x = NULLDATA;
  if ( l.pHead != NULL)
  {  p = l.pHead; x = p->Info;
     l.pHead = l.pHead->pNext;
     l.pHead->pPrev = NULL;
     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 xâu

```
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.pHead == NULL) l.pTail        = NULL;
     else                l.pHead->pPrev = 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

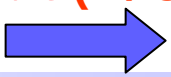
Hủy một phần tử đứng trước phần tử q

```
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
Sắp xếp danh sách

- Việc sắp xếp danh sách trên xâu kép về cơ bản không khác gì trên xâu đơn.
- Ta chỉ cần lưu ý một điều duy nhất là cần bảo toàn các mối liên kết hai chiều trong khi sắp xếp.

Danh sách liên kết kép

Sắp xếp danh sách - DListQSort(1/2)

```
void      DListQSort(DLIST & l)
{ DNODE   *p, *X;        // X chỉ đến  phần tử cầm canh
  DLIST   l1, l2;
  if(l.pHead == l.pTail) return;//đã có thứ tự
  l1.pHead == l1.pTail = NULL; //khởi tạo
  l2.pHead == l2.pTail = NULL;
  X = l.pHead; l.pHead = X->pNext;
  while(l.pHead != NULL) //Tách l thành l1, l2;
  {  p = l.pHead;
     l.pHead = p->pNext; p->pNext = NULL;
     if (p->Info <= X->Info) AddTail(l1, p);
     else                    AddTail(l2, p);
  }
.....................................DListQSort(2/2)
}
```

Danh sách liên kết kép

Sắp xếp danh sách - DListQSort(2/2)

```
void      DListQSort(DLIST & l)
{ ..........................................
  DListQSort(l1);          //Gọi đệ qui để sort l1
  DListQSort(l2);          //Gọi đệ qui để sort l2
  //Nối l1, X và l2 lại thành l đã sắp xếp.
  if(l1.pHead != NULL)
  {  l.pHead = l1.pHead; l1.pTail->pNext = X;
     X->pPrev = l1.pTail;
  } else l.pHead = X;
  X->pNext = l2;
  if(l2.pHead != NULL)
  {  l.pTail = l2.pTail;
     l2->pHead->pPrev = X;
  } else l.pTail = X;
}
```

Danh sách liên kết kép

- Xâu kép về mặt cơ bản có tính chất giống như xâu đơn.
- Tuy nhiên nó có một số tính chất khác xâu đơn như sau:
  - Xâu kép có mối liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên xâu 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 xâu kép, còn trên xâu đơn thao tác này tồn chi phí O(n).
  - Bù lại, xâu kép tốn chi phí gấp đôi so với xâu 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.

Hàng đợi hai đầu (double-ended queue)

- Hàng đợi hai đầu (gọi tắt là Deque) là một vật chứa các đối tượng mà việc thêm hoặc hủy một đối tượng được thực hiện ở cả 2 đầu của nó.
- Deque là một CTDL trừu tượng (ADT) hỗ trợ các thao tác chính sau:
  - `InsertFirst`(e): Thêm đối tượng e vào đầu deque
  - `InsertLast`(e): Thêm đối tượng e vào cuối deque
  - `RemoveFirst`(): Lấy đối tượng ở đầu deque ra khỏi deque và trả về giá trị của nó.
  - `RemoveLast`(): Lấy đối tượng ở cuối deque ra khỏi deque và trả về giá trị của nó.

Hàng đợi hai đầu (double-ended queue)

- Ngoài ra, deque cũng hỗ trợ các thao tác sau:
  - `IsEmpty`(): Kiểm tra xem deque có rỗng không.
  - `First`(): Trả về giá trị của phần tử nằm ở đầu deque mà không hủy nó.
  - `Last`(): Trả về giá trị của phần tử nằm ở cuối deque mà không hủy nó.

Dùng deque để cài đặt stack và queue

- Dùng Deque để cài đặt Stack

STT	Stack	Deque
1	Push	InsertLast
2	Pop	RemoveLast
3	Top	Last
4	IsEmpty	IsEmpty

- Dùng Deque để cài đặt Queue

STT	Queue	Deque
1	Enqueue	InsertLast
2	Dequeue	RemoveFist
3	Front	First
4	IsEmpty	IsEmpty

Cài đặt deque

- Do đặc tính truy xuất hai đầu của deque, việc xây dựng CTDL biểu diễn nó phải phù hợp.
- Có thể cài đặt CTDL deque bằng danh sách liên kết đơn. Tuy nhiên, khi đó thao tác RemoveLast hủy phần tử ở cuối deque sẽ tốn chi phí O(n). Điều này làm giảm hiệu quả của CTDL.
- Thích hợp nhất để cài đặt deque là dùng danh sách liên kết kép. Tất cả các thao tác trên deque khi đó sẽ chỉ tốn chi phí O(1).

Danh sách liên kết có thứ tự (Odered List)

- Danh sách liên kết có thứ tự (gọi tắt là OList) là một vật chứa các đối tượng theo một trình tự nhất định.
- Trình tự này thường là một khóa sắp xếp nào đó. Việc thêm một đối tượng vào OList phải bảo đảm tôn trọng thứ tự này.
- Ta có thể cài đặt OList bằng DSLK đơn hoặc DSLK đôi với việc định nghĩa lại duy nhất một phép thêm phần tử: thêm bảo toàn thứ tự. Nghĩa là trên OList chỉ cho phép một thao tác thêm phần tử sao cho thứ tự định nghĩa trên OList phải bảo toàn.

Danh sách liên kết có thứ tự (Odered List)

```
NODE* InsertNode(LIST & l, Data X)
{ NODE* q = NULL, *p = l.pHead;
  if(IsEmpty(l)) return InsertHead(l, X);
  while(p)
  {  if(p->Info >= X)  break;
     q = p;
     p = q->pNext;
  }
  NODE* pT = new(NODE);
  pT->Info = X; pT->pNext= p;
  if(q) q->pNext = pT;
  else  l.pHead  = pT;
  if(q == l->pTail) l.pTail   = pT;
  return pT;
}
```

Danh sách liên kết có thứ tự (Odered List)

```
NODE* SearchNode(LIST l, Data X)
{
  NODE*p = l.pHead;

  while(p)
  {  if(p->Info == X)  break;
     else if(p->Info > X) return NULL;
     p = p->pNext;
  }
  return p;
}
```

Danh sách liên kết có thứ tự (Odered List)

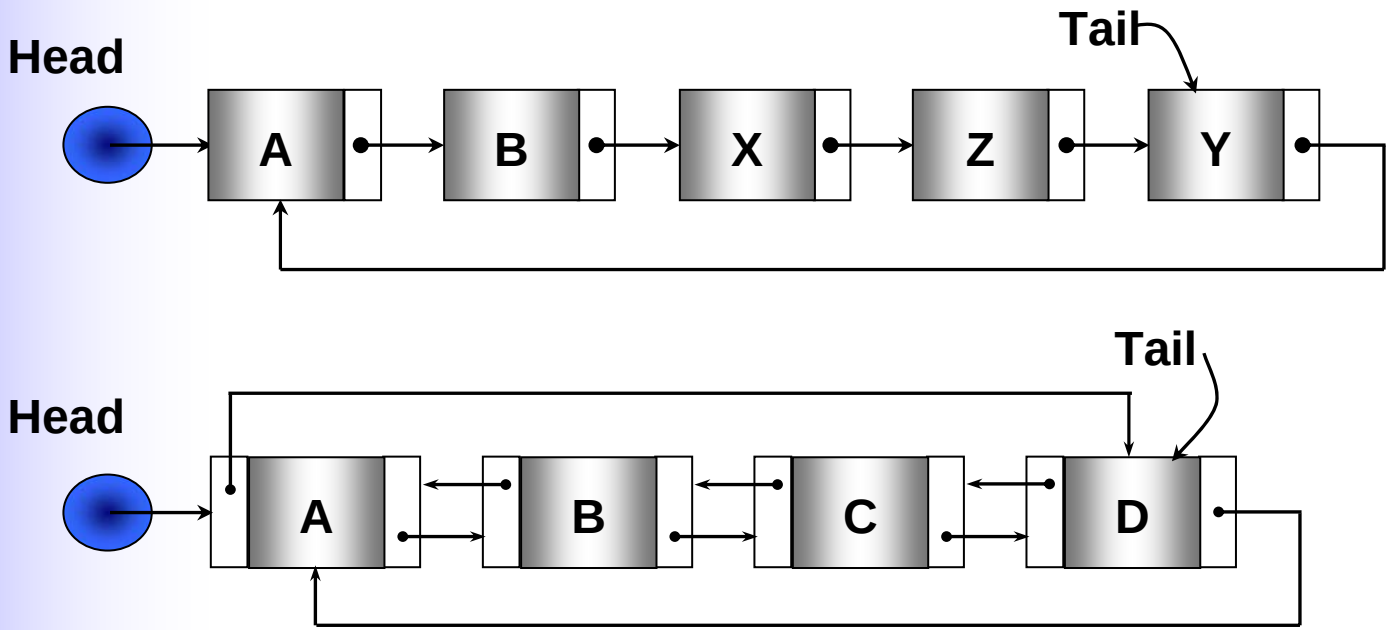
- Có thể dùng OList để cài đặt CTDL hàng đợi có độ ưu tiên.
- Trong hàng đợi có độ ưu tiên, mỗi phần tử được gán cho một độ ưu tiên.
- Hàng đợi có độ ưu tiên cũng giống như hàng đợi bình thường ở thao tác lấy một phần tử khỏi hàng đợi (lấy ở đầu queue) nhưng khác ở thao tác thêm vào. Thay vì thêm vào ở cuối queue, việc thêm vào trong hàng đợi có độ ưu tiên phải bảo đảm phần tử có độ ưu tiên cao đứng trước, phần tử có độ ưu tiên thấp đứng sau.
- Hàng đợi có độ ưu tiên có nhiều ứng dụng, ví dụ như dùng quản lý hàng đợi các tiến trình chờ được xử lý trong các hệ điều hành đa nhiệm.

Danh sách liên kết vòng

- 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, trỏ 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.

Danh sách liên kết vòng

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



Danh sách liên kết vòng
Tìm phần tử trên danh sách vòng

- Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phân tử đầu xâu để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
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 giáp vòng
  return p;
}
```

Danh sách liên kết vòng
Thêm phần tử đầu xâu

```
void      AddHead(LIST &l, NODE *new_ele)
{
  if(l.pHead == NULL)  //Xâu 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;
  }
}
```

Danh sách liên kết vòng

Thêm phần tử cuối xâu

```
void      AddTail(LIST &l, NODE *new_ele)
{
  if(l.pHead == NULL)  //Xâu 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;
  }
}
```

Danh sách liên kết vòng

Thêm phần tử sau nút q

```
void      AddAfter(LIST &l, NODE *q, NODE *new_ele)
{
  if(l.pHead == NULL)  //Xâu 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)
      l.pTail = new_ele;
  }
}
```

Danh sách liên kết vòng
Hủy phần tử đầu xâu

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

Danh sách liên kết vòng

Hủy phần tử đứng sau nút q

```
void RemoveAfter(LIST &l, NODE *q)
{ NODE    *p;
  if(q != NULL)
  {
   p = q ->Next ;
   if ( p == q)  l.pHead = l.pTail = NULL;
   else
   {
     q->Next = p->Next;
     if(p == l.pTail)
        l.pTail = q;
   }
   delete p;
  }
}
```

Danh sách có nhiều mối liên kết

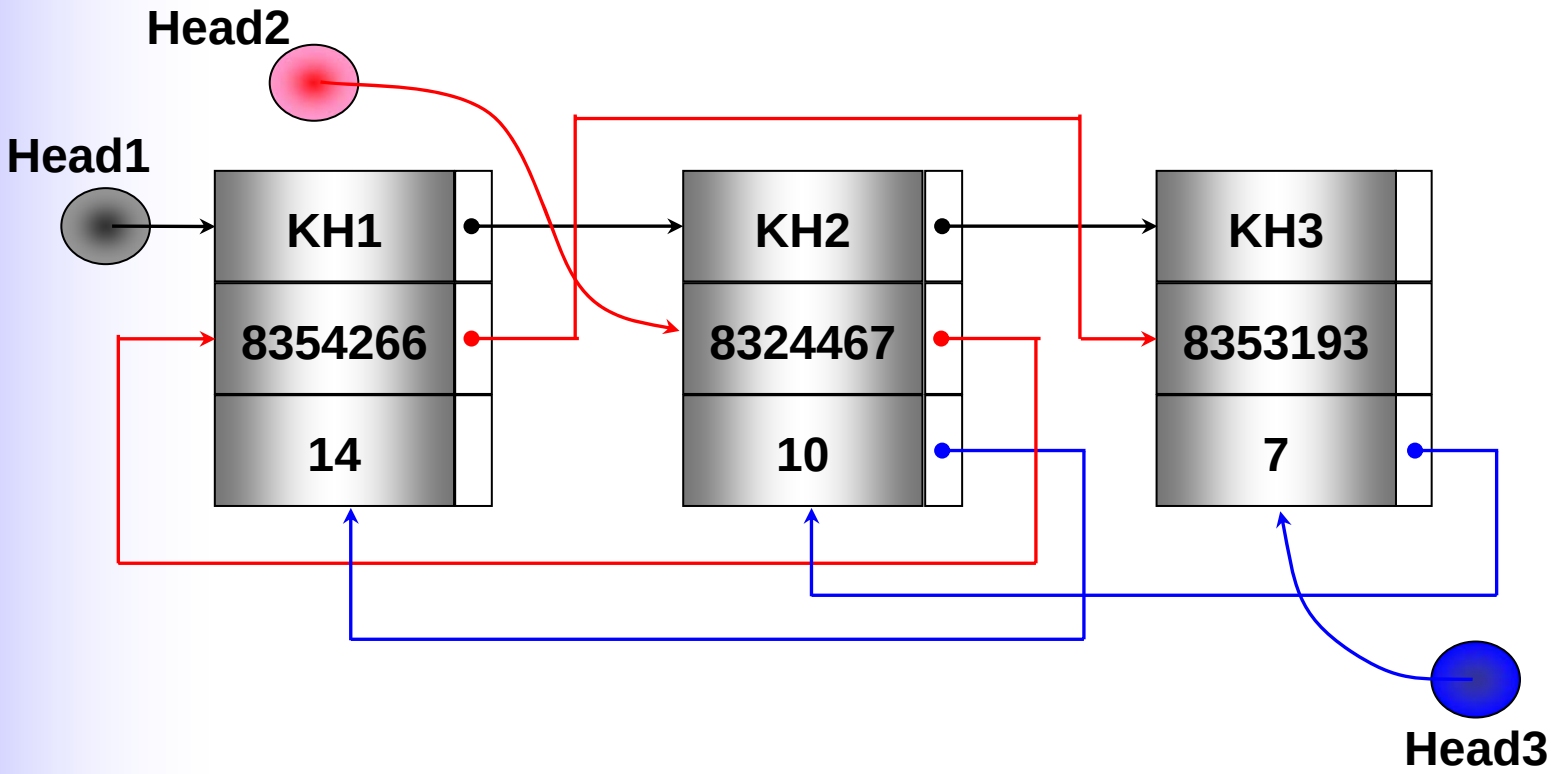
- Danh sách có nhiều mối liên kết là danh sách mà mỗi phần tử có nhiều khóa và chúng được liên kết với nhau theo từng loại khoá.
- Danh sách có nhiều mối liên kết thường được xử dụng trong các ứng dụng quản lý một cơ sở dữ liệu lớn với những nhu cầu tìm kiếm dữ liệu theo những khoá khác nhau.

Danh sách có nhiều mối liên kết

- Ví dụ: Để quản lý danh mục điện thoại thuận tiện cho việc in danh mục theo những trình tự khác nhau : tên khách tăng dần, theo số điện thoại tăng dần, thời gian lắp đặt giảm dần, ta có thể tổ chức dữ liệu như hình trên: một danh sách vơí 3 mối liên kết:
  - một cho họ tên khách hàng,
  - một cho số điện thoại
  - một cho thời gian lắp đặt

Danh sách có nhiều mối liên kết

- Ví dụ: quản lý danh mục điện thoại



Danh sách tổng quát

- Danh sách tổng quát là một danh sách mà mỗi phần tử của nó có thể lại là một danh sách khác.
- Các ví dụ sau minh hoạ các cấu trúc danh sách tổng quát.
- Các thao tác trên một danh sách được xây dựng dựa trên cơ sở các thao tác trên danh sách liên kết chúng ta đã khảo sát.

Danh sách tổng quát

- Ví dụ 1:

```
typedef   struct tagNode
{
   DATA            Info;
   struct tagNode *pNext;
   void            *Sublist;
}NODE;
typedef   NODE   *PNODE;
```

Danh sách tổng quát

- Ví dụ 2:

```
typedef   struct tagNguoi
{
   char     Ten[35];
   char     GioiTinh;
   int      NamSinh;
   struct tagNguoi  *Cha, *Me;
   struct tagNguoi  *Anh, *Chi, *Em;
}NGUOI;
typedef   NGUOI  *PNGUOI;
```

Danh sách tổng quát

- Ví dụ 3:

```
typedef   struct tagGNode
{
   DATA     Info;
   struct tagGNode **pLink;
}GNODE;
typedef GNODE    *PGNODE;
```