



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



# Cấu trúc dữ liệu và thuật toán

**Nguyễn Khánh Phương**

**Computer Science department  
School of Information and Communication technology  
E-mail: [phuongnk@soict.hust.edu.vn](mailto:phuongnk@soict.hust.edu.vn)**

# Nội dung khóa học

Chương 1. Các khái niệm cơ bản

Chương 2. Các sơ đồ thuật toán

**Chương 3. Các cấu trúc dữ liệu cơ bản**

**Chương 4. Cây**

**Chương 5. Sắp xếp**

**Chương 6. Tìm kiếm**

**Chương 7. Đồ thị**



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



## Chương 3. Các cấu trúc dữ liệu cơ bản

**Nguyễn Khánh Phương**

Computer Science department  
School of Information and Communication technology  
E-mail: [phuongnk@soict.hust.edu.vn](mailto:phuongnk@soict.hust.edu.vn)

# Kiểu dữ liệu (Data types)

- **Kiểu dữ liệu (data type)** được đặc trưng bởi:
  - Tập các giá trị (a set of *values*);
  - Cách biểu diễn dữ liệu (*data representation*) được sử dụng chung cho tất cả các giá trị này và
  - Tập các phép toán (set of *operations*) có thể thực hiện trên tất cả các giá trị.
- **Chú ý:**
  - Mỗi giá trị có một cách biểu diễn nào đó mà người sử dụng không nhất thiết phải biết
  - Mỗi phép toán được cài đặt theo một cách nào đó mà người sử dụng cũng không cần phải biết

# Các kiểu dữ liệu dựng sẵn

(Built-in data types)

- Trong các ngôn ngữ lập trình thường có một số kiểu dữ liệu nguyên thủy đã được xây dựng sẵn. Ví dụ:
  - Kiểu số nguyên (Integer numeric types)
    - byte, char, short, int, long
  - Kiểu số thực dấu phẩy động (floating point numeric types)
    - float, double
  - Các kiểu nguyên thủy khác (Other primitive types)
    - boolean
  - Kiểu mảng (Array type)
    - mảng các phần tử cùng kiểu

# Dữ liệu đối với kiểu nguyên thủy

## Trong ngôn ngữ lập trình C

Type	Byte	Minimum value	Maximum value
byte	1	-128	127
short	2	-32768	32767
char	2	0	65535
int	4	$-2147483648 = -2^{31}$	$2147483647 = 2^{31}-1$
long	8	-9223372036854775808	9223372036854775807
float	4	$\approx \pm 1.40 \times 10^{-45}$	$\approx \pm 3.40 \times 10^{38}$
double	8	$\approx \pm 4.94 \times 10^{-324}$	$\approx \pm 1.80 \times 10^{308}$

Có thể có kiểu boolean với hai giá trị true hoặc false

# Phép toán đối với kiểu dữ liệu nguyên thủy

- **Đối với kiểu: `byte`, `char`, `short`, `int`, `long`**
  - ✓ `+`, `-`, `*`, `/`, `%`, đổi thành xâu, ...
- **Đối với kiểu: `float`, `double`**
  - ✓ `+`, `-`, `*`, `/`, `round`, `ceil`, `floor`, ...
- **Đối với kiểu: `boolean`**
  - ✓ kiểm giá trị `true`, hay kiểm giá trị `false`
- Nhận thấy rằng: Các ngôn ngữ lập trình khác nhau có thể sử dụng mô tả kiểu dữ liệu khác nhau. Chẳng hạn, PASCAL và C có những mô tả các dữ liệu số khác nhau.

# Nội dung

1. Mảng (Array)
2. Bản ghi (Record)
3. Danh sách liên kết (Linked List)
4. Ngăn xếp (Stack)
5. Hàng đợi (Queue)

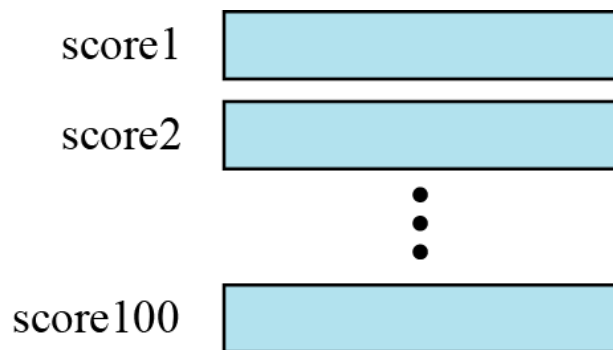


# Nội dung

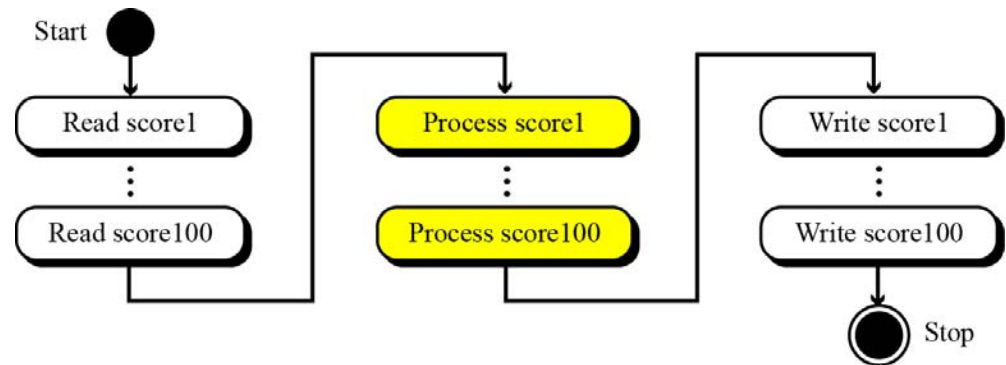
- 1. Mảng (Array)**
2. Bản ghi (Record)
3. Danh sách liên kết (Linked List)
4. Ngăn xếp (Stack)
5. Hàng đợi (Queue)

# 1. Mảng (Array)

- Giả sử có 100 tỉ số (scores) trong một giải đấu bóng chày. Chúng ta cần tiến hành các thao tác: lấy thông tin của 100 tỉ số đó (read them), rồi đem xử lý các tỉ số này (process them), và cuối cùng là in chúng (print/write them).
- Để làm được điều này, ta cần lưu trữ 100 tỉ số này vào bộ nhớ trong suốt quá trình thực hiện chương trình. Do đó, ta sẽ sử dụng 100 biến, mỗi biến một tên tương ứng với 1 tỉ số: (Xem hình 1)



Hình 1 Sử dụng 100 biến

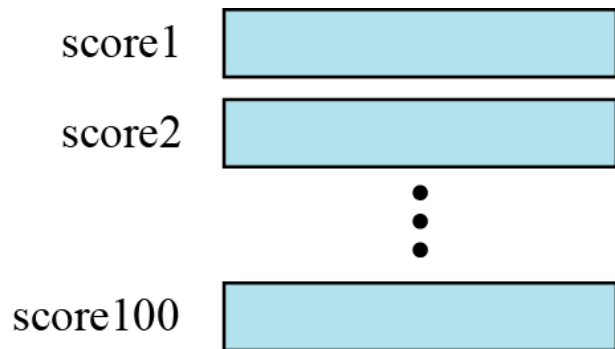


Hình 2 Xử lý dữ liệu tỉ số trên 100 biến

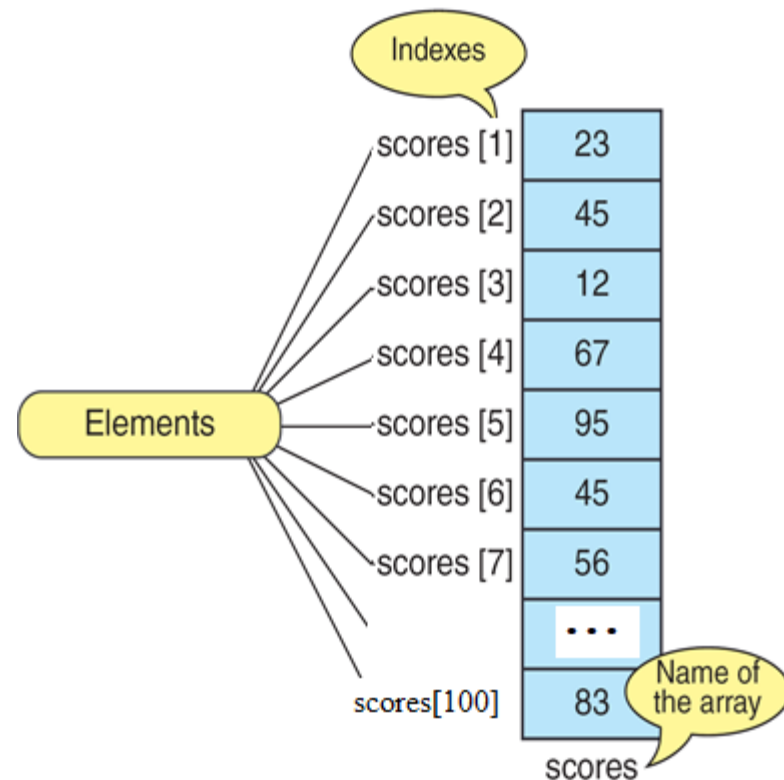
- Sử dụng 100 biến dẫn đến vấn đề: ta cần phải viết lệnh đọc (read) 100 lần, lệnh xử lý (process) 100 lần và lệnh in (write) 100 lần (xem Hình 2).

# 1. Mảng (Array)

- Thay vì khai báo biến một cách rời rạc 100 biến `score1`, `score2`, ..., `score100`, ta có thể khai báo một mảng các giá trị như `scores[1]`, `scores[2]` và ... `scores[100]` để biểu diễn các giá trị riêng biệt.



Hình 1 Sử dụng 100 biến



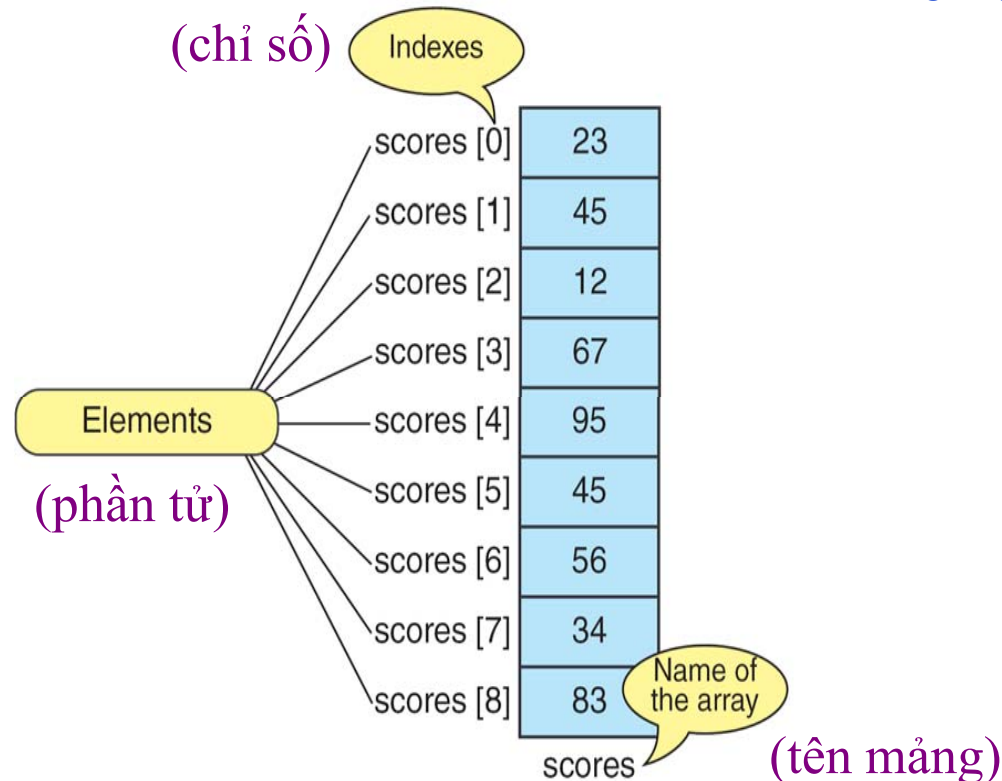
Hình 2 Sử dụng mảng scores gồm 100 phần tử

# 1. Mảng (Array)

**Mảng là một kiểu dữ liệu chứa dãy các phần tử được đánh chỉ số, thông thường các phần tử này có cùng kiểu dữ liệu.**

- Mỗi phần tử của mảng có một chỉ số cố định duy nhất
  - Chỉ số nhận giá trị trong khoảng từ một **cận dưới** đến một **cận trên** nào đó

Ví dụ: Trong ngôn ngữ C, mảng scores kích thước  $N = 9$ , mỗi phần tử được đánh một chỉ số duy nhất  $i$  với điều kiện  $0 \leq i < N$ , tức là chỉ số của mảng luôn luôn được bắt đầu từ 0 và phải nhỏ hơn kích thước của mảng. Để truy cập đến mỗi phần tử của mảng ta chỉ cần biết chỉ số của chúng, khi ta viết **scores[i]** có nghĩa là ta đang truy cập tới phần tử thứ  $i$  của mảng scores



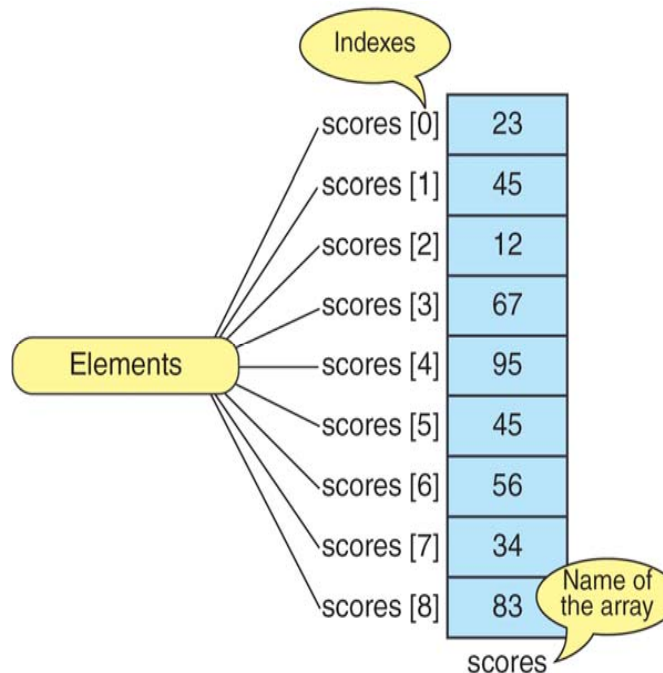
# Tên mảng vs. Tên phần tử của mảng

Trong 1 mảng, ta có 2 kiểu định danh:

- Tên của mảng
- Tên của các phần tử riêng biệt thuộc mảng.

Tên của mảng là tên của toàn bộ cấu trúc, còn tên của một phần tử cho phép ta truy cập đến phần tử đó.

Ví dụ:

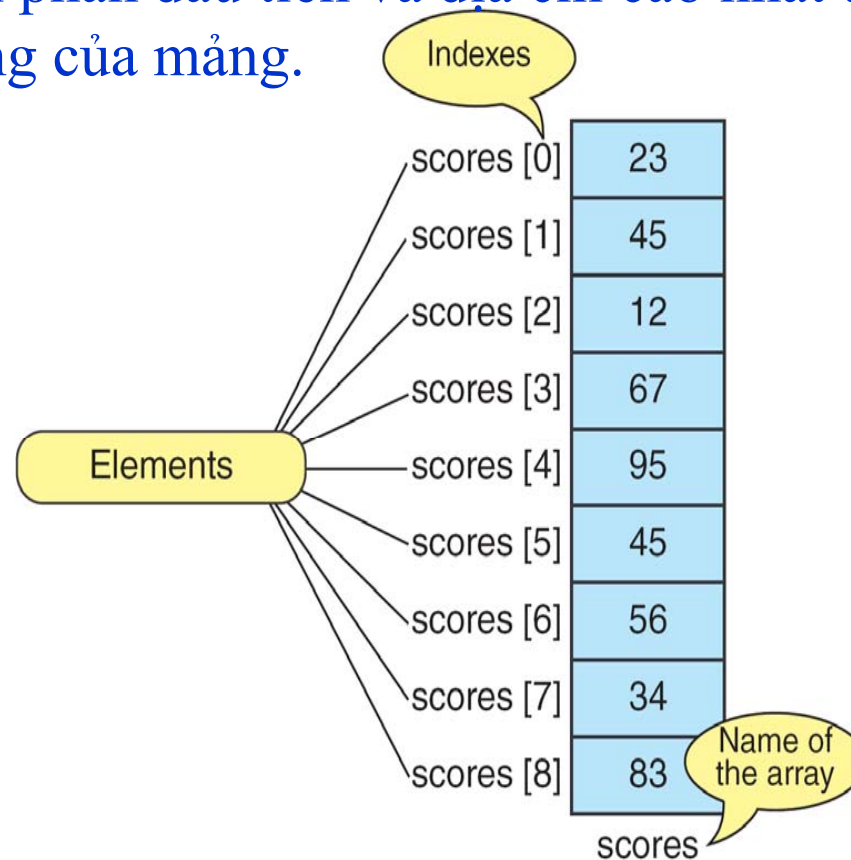


Tên của mảng: *scores*,

Tên mỗi phần tử của mảng: gồm tên của mảng theo sau là chỉ số của phần tử, ví dụ *scores[0]*, *scores[1]*, ...

# 1. Mảng (Array)

- Mảng (Array): tập các cặp (**chỉ số (index)** và **giá trị (value)**)
  - Cấu trúc dữ liệu: mỗi chỉ số, sẽ tương ứng với 1 giá trị.
  - Lưu trữ trong bộ nhớ: mảng được lưu trữ ở các ô nhớ liên kề nhau (cách lưu trữ này được gọi là *lưu trữ kế tiếp*). Địa chỉ thấp nhất tương ứng với thành phần đầu tiên và địa chỉ cao nhất tương ứng với thành phần cuối cùng của mảng.



# Mảng trong các ngôn ngữ lập trình

- Các chỉ số có thể là số nguyên (C/C++, Java) hoặc là các giá trị kiểu rời rạc (Pascal, Ada)
- Cận dưới là 0 (C/C++, Java), 1 (Fortran), hoặc tùy chọn bởi người lập trình (Pascal, Ada)
- Trong hầu hết các ngôn ngữ, mảng là **thuần nhất** (nghĩa là tất cả các phần tử của mảng có cùng một kiểu); trong một số ngôn ngữ (như Lisp, Prolog) các thành phần có thể là không thuần nhất (có các kiểu khác nhau)

# Khai báo mảng 1 chiều (one-dimensional array)

Khi khai báo mảng, ta cần kiểu mảng (type), tên mảng (arrayName) và số phần tử trong mảng (arraySize > 0) :

**type** **arrayName** [**arraySize**] ;



Ví dụ: khai báo **int** **A**[5] ;

tạo mảng A gồm 5 phần tử kiểu số nguyên (vì là kiểu nguyên, nên mỗi phần tử chiếm 4 bytes trong bộ nhớ)

- Nếu kích thước mảng **arraySize** là hằng số thì cho ta mảng có độ dài cố định (fixed length array), nếu là 1 biến (variable) thì cho ta mảng có độ dài thay đổi (variable-length arrays)

– Ví dụ: `double A[10] ;`

Mảng A cố định gồm 10 phần tử

```
int n;  
double B[n] ;
```

Mảng B có độ dài thay đổi qua giá trị của biến *n*

- Chú ý: trước khi sử dụng một mảng, ta luôn phải khai báo và khởi tạo nó

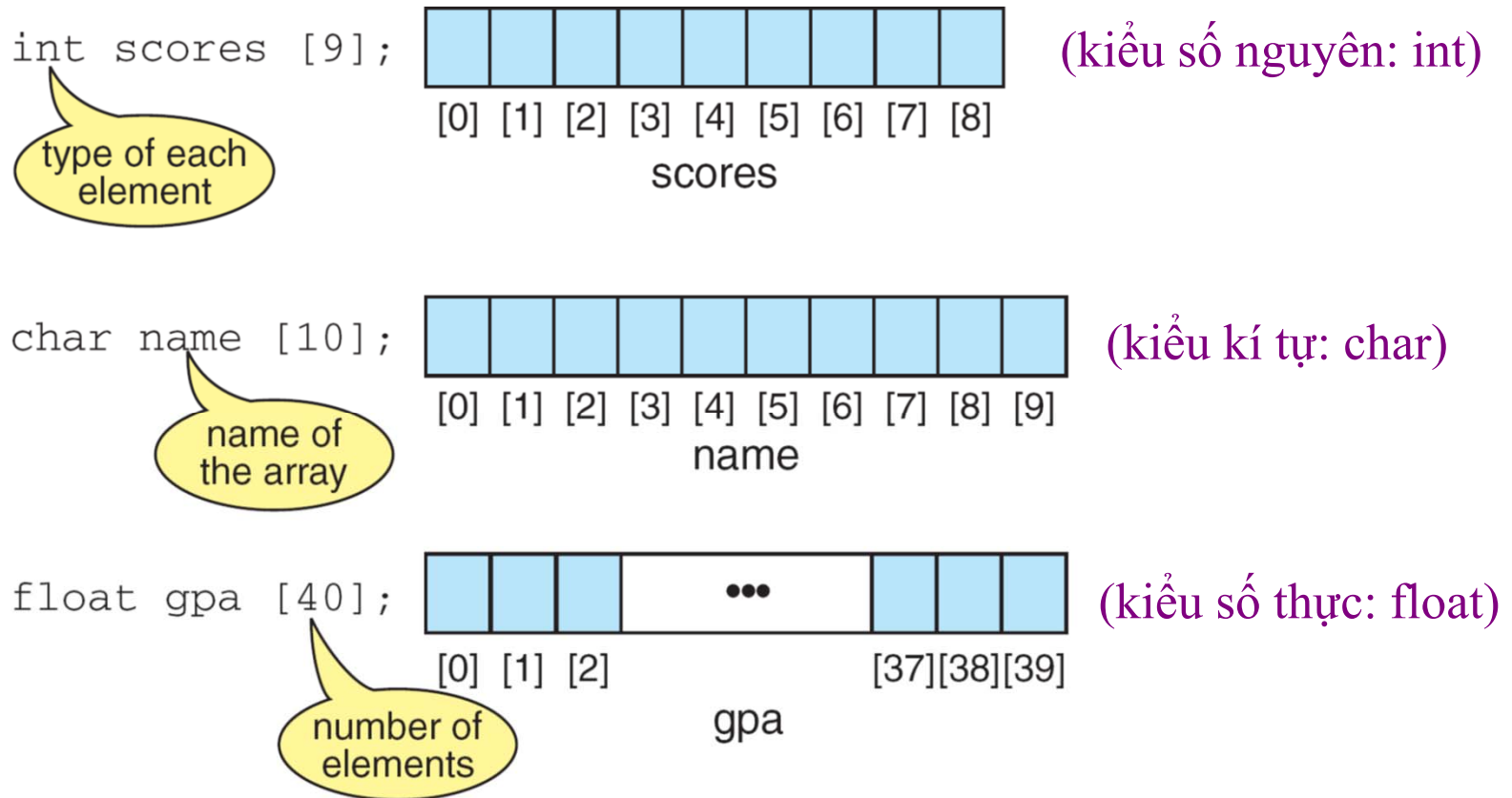


# Mảng trong ngôn ngữ C

**Ngôn ngữ C hỗ trợ 2 kiểu mảng:**

- ✓ *Mảng có độ dài cố định (Fixed Length Arrays)* : lập trình viên “hard codes” (cố định) độ dài của mảng.
- ✓ *Mảng có độ dài thay đổi (Variable-Length Arrays)*: lập trình viên không biết độ dài của mảng cho đến khi chạy chương trình.

# Ví dụ: Khai báo mảng có độ dài cố định (fixed length array)

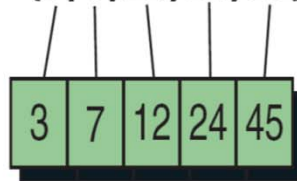


# Khai báo mảng 1 chiều có độ dài cố định

- Ta có thể khởi tạo giá trị cho các phần tử của mảng cùng lúc với khai báo mảng
- Nếu số giá trị dùng khởi tạo nhỏ hơn kích thước mảng, C sẽ tự động gán cho các thành phần còn lại nhận giá trị 0

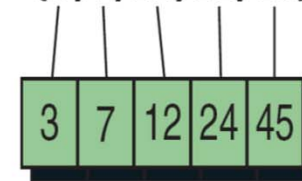
(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```



(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



(c) Partial Initialization

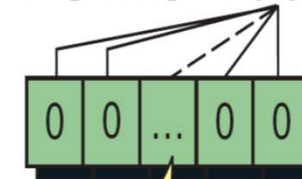
```
int numbers[5] = {3,7};
```



The rest are filled with 0s

(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```



All filled with 0s

# Truy cập phần tử của mảng

Để truy cập vào phần tử của một mảng, ta cần một giá trị nguyên xác định chỉ số của phần tử mà ta muốn truy cập.

- Có thể dùng chỉ số là 1 hằng số: **scores[0];**
- Cũng có thể dùng chỉ số là 1 biến (variable):

```
for (i = 0; i < 9; i++)  
    scoresSum += scores[i];
```

# Truy cập phần tử của mảng

```
#include <stdio.h>

int main ()
{
    int A[ 10 ]; // khai bao A la mot mang gom 10 so nguyen

    // khoi tao gia tri cac phan tu cua mang A:
    for ( int i = 0; i < 10; i++ )
        A[ i ] = i + 100; // thiet lap phan tu tai vi tri i co gia tri la i + 100

    // hien thi gia tri cua moi phan tu trong mang A:
    for ( int i = 0; i < 10; i++ )
        printf("Phan tu thu %d, co gia tri la: %d \n",i,A[i]);
}
```


Kết quả chạy chương trình:

```
Phan tu thu 0, co gia tri la: 100
Phan tu thu 1, co gia tri la: 101
Phan tu thu 2, co gia tri la: 102
Phan tu thu 3, co gia tri la: 103
Phan tu thu 4, co gia tri la: 104
Phan tu thu 5, co gia tri la: 105
Phan tu thu 6, co gia tri la: 106
Phan tu thu 7, co gia tri la: 107
Phan tu thu 8, co gia tri la: 108
Phan tu thu 9, co gia tri la: 109
```

# Con trỏ (pointers)

- Giá trị các biến được lưu trữ trong bộ nhớ máy tính, có thể truy cập tới các giá trị đó qua tên biến, đồng thời cũng có thể qua địa chỉ của chúng trong bộ nhớ.
- Con trỏ thực chất là 1 biến mà nội dung của nó là địa chỉ của 1 đối tượng khác (Biến, hàm, nhưng không phải 1 hằng số) ➔ Việc sử dụng con trỏ cho phép ta truy nhập tới 1 đối tượng gián tiếp qua địa chỉ của nó.
- Có nhiều kiểu biến với các kích thước khác nhau, nên có nhiều kiểu con trỏ. Con trỏ `int` để trỏ tới biến hay hàm kiểu `int`.
- Cách khai báo:

```
type *pointer_name;
```



Chỉ rằng đây là con trỏ

Sau khi khai báo, ta được con trỏ NULL, vì nó chưa trỏ tới 1 đối tượng nào.

- Để sử dụng con trỏ, ta dùng toán tử lấy địa chỉ &

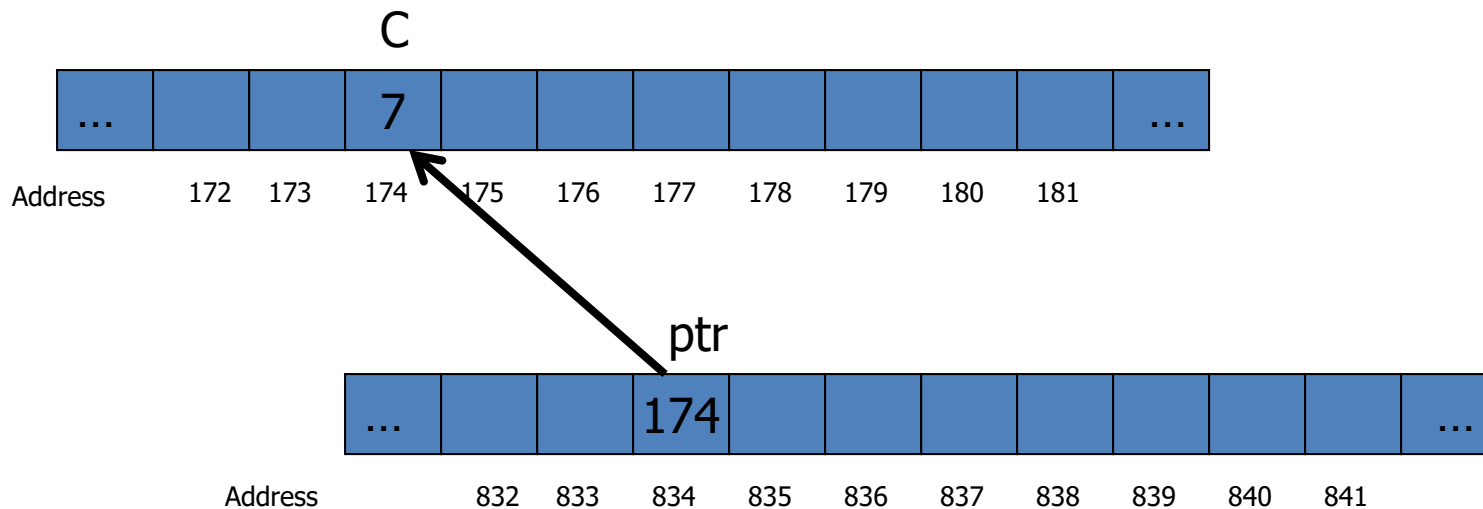
```
pointer_name = &var_name;
```

- Để lấy nội dung biến do con trỏ trỏ tới, ta dùng toán tử lấy nội dung \*

```
*pointer_name
```

# Con trỏ: Ví dụ

```
int c;  
int *ptr; /* Khai báo biến ptr là con trỏ int */  
c = 7;  
ptr = &c;  
  
printf("%d", *ptr); /* in ra số 7 */  
*ptr = 80;  
printf("%d", c); /* in ra số 80 */
```



# Con trỏ void\*

- Là con trỏ không định kiểu.
- Nó có thể trỏ tới bất kì một loại biến nào.
- Thực chất một con trỏ void chỉ chứa một địa chỉ bộ nhớ mà không biết rằng tại địa chỉ đó có đối tượng kiểu dữ liệu gì. ➔ không thể truy cập nội dung của một đối tượng thông qua con trỏ void.
- Để truy cập được đối tượng thì trước hết phải ép kiểu biến trỏ void thành biến trỏ có định kiểu của kiểu đối tượng

## Ví dụ:

```
float x; int y;
void *p; // khai báo con trỏ void
p = &x; // p chứa địa chỉ số thực x
*p = 2.5; // báo lỗi vì p là con trỏ void
/* cần phải ép kiểu con trỏ void trước khi truy cập đối tượng
qua con trỏ */
*((float*)p) = 2.5; // x = 2.5
p = &y; // p chứa địa chỉ số nguyên y
*((int*)p) = 2; // y = 2
```



# Bộ nhớ động

Khi khai báo các biến mảng với độ dài cố định:

- Ta chỉ lưu trữ một số lượng cố định các biến.
- Kích thước không thể thay đổi sau khi biên dịch

Tuy nhiên, không phải lúc nào chúng ta cũng biết trước được chính xác dung lượng chúng ta cần.

➔ Việc dùng bộ nhớ động cho phép xác định bộ nhớ cần thiết trong quá trình thực hiện của chương trình, đồng thời giải phóng chúng khi không còn cần đến để dùng bộ nhớ cho việc khác.

1. Cấp phát: để cấp phát vùng nhớ cho con trỏ ta dùng thư viện chuẩn **stdlib.h**

1. **malloc**

2. **calloc**

3. **realloc**

2. Giải phóng

1. **free**

# Cấp phát động malloc

malloc (memory allocation) : trả về địa chỉ byte đầu tiên của vùng bộ nhớ được cấp phát nếu cấp phát thành công, hoặc trả về NULL nếu cấp phát thất bại → luôn cần kiểm tra bộ nhớ có được cấp phát thành công hay không.

Ví dụ: `int *pointer = (int *) malloc(100);`

- Nếu được cấp phát thành công, 100 bytes bộ nhớ này sẽ nằm trên vùng heap. Vùng nhớ mới được cấp phát này có thể lưu được tối đa  $100/4 = 25$  số nguyên int (4 bytes) hoặc tối đa  $100/2 = 50$  số nguyên int (2 bytes).
- Để tránh khác biệt về kích thước dữ liệu (ví dụ: kiểu int có thể là 2 hoặc 4 bytes tùy thuộc vào kiến trúc máy tính và hệ điều hành), ta có thể sử dụng:

`int *pointer = (int *) malloc (25 * sizeof(int));`

→ cấp phát bộ nhớ cho 1 mảng số nguyên 25 phần tử

Hàm malloc trả về con trỏ kiểu void. Con trỏ kiểu void có thể ép được sang bất kỳ kiểu dữ liệu nào → do đó ta dùng (int \*) để ép sang kiểu int

`kieu_con_trong *ten_con_trong = (kieu_con_trong *) malloc (sizeof(kieu_con_trong));`  
`kieu_con_trong *ten_con_trong = (kieu_con_trong *) malloc (size * sizeof(kieu_con_trong));`

# Cấp phát động calloc

calloc (contiguous allocation) : cũng giống như malloc, calloc được dùng để cấp phát bộ nhớ động. Tuy nhiên, toàn bộ vùng nhớ được cấp phát bởi hàm calloc( ) sẽ được gán giá trị 0.

Ví dụ: Cấp phát bộ nhớ cho 1 mảng nguyên 25 phần tử

```
int *pointer = (int *) calloc(25, sizeof(int));
```

```
int *pointer = (int *) malloc (25 * sizeof(int));
```

# Giải phóng bộ nhớ free

Khác với biến cục bộ và tham số của một hàm nằm trên vùng nhớ stack (sẽ được tự động giải phóng ngay sau khi ra khỏi phạm vi của hàm), vùng nhớ được cấp phát động nằm trên vùng heap sẽ không được giải phóng → Nếu không được giải phóng, chương trình có thể sẽ bị memory leak.

Để giải phóng vùng nhớ được cấp phát bởi malloc( ), calloc( ), realloc( ), ta dùng hàm free( )

Ví dụ:

```
int *pointer = (int *) calloc(25, sizeof(int));  
int *pointer = (int *) malloc (25 * sizeof(int));
```

→ free(pointer);

Chú ý: không xóa một vùng nhớ đã được cấp phát 2 lần  
free( ); → không làm gì cả

# Tái cấp phát realloc

realloc (re-allocation) : cấp phát lại trên chính vùng nhớ đã cấp phát trước đó do vùng nhớ cấp phát trước đó không đủ hoặc quá lớn.

Ví dụ:

Cấp phát bộ nhớ cho 1 mảng nguyên 25 phần tử bằng malloc

```
int *pointer = (int *) malloc (25 * sizeof(int));
```

Cấp phát lại bộ nhớ chỉ cần 20 phần tử:

```
→ pointer = (int *) realloc(pointer, 20*sizeof(int));
```

Cấp phát lại bộ nhớ chỉ cần 50 phần tử:

```
→ pointer = (int *) realloc(pointer, 50*sizeof(int));
```

# Ví dụ: Mảng có độ dài thay đổi

Nhập vào một dãy số nguyên từ bàn phím. In dãy số theo thứ tự ngược lại. Yêu cầu dùng cấp phát động

```
int main(void)
{
    int i, n, *p;

    printf("Nhập số lượng phần tử của mảng = ");
    scanf("%d", &n);

    /* Cấp phát bộ nhớ cho mảng số nguyên gồm n số */
    p = (int *)malloc(n * sizeof(int));
    if (p == NULL)
    {
        printf("Cấp phát bộ nhớ không thành công!\n");
        return 1;
    }
    /* Nhập các phần tử của mảng */
    printf("Hãy nhập các số:\n");
    ...
    for (i = 0; i < n; i++) scanf("%d", &p[i]);
    /* Hiển thị các phần tử của mảng theo thứ tự ngược lại */
    ...
    printf("Các phần tử theo thứ tự ngược lại là:\n");
    for (i = n - 1; i >= 0; --i) printf("%d ", p[i]);

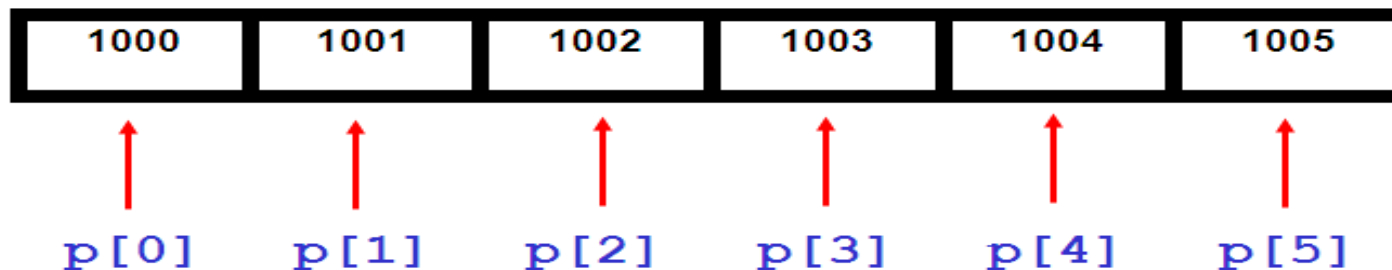
    /* Giải phóng bộ nhớ đã cấp phát */
    free(p);
    return 0;
}
```

# Khai báo mảng

- Điều gì xảy ra khi ta khai báo 1 mảng **p**?

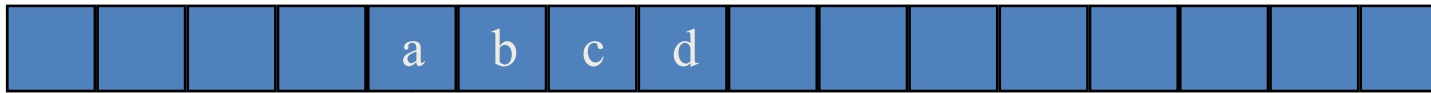
## Bộ nhớ (Memory):

- Các phần tử của mảng được lưu trữ liên kế tiếp nhau trong bộ nhớ.
- Về cơ bản, máy tính sẽ dựa trên địa chỉ **GỐC** (trở bởi biến p – cũng là địa chỉ của phần tử đầu tiên của mảng p) của mảng, rồi đếm tuần tự lần lượt.



# Biểu diễn mảng 1 chiều trong ngôn ngữ C

Bộ nhớ



**start\_address** (địa chỉ gốc)

Ví dụ: Mảng 1 chiều  $x = [a, b, c, d]$

- Lưu trữ vào một khối bộ nhớ liên tiếp (contiguous memory locations)
- Địa chỉ( $x[i]$ ) =  $\text{start\_address} + W * i$

biết rằng:

- **start\_address**: địa chỉ phần tử đầu tiên trong mảng
- **W**: kích thước của mỗi phần tử trong mảng

**Ví dụ:** Xét khai báo **float list[5];**

- Khai báo trên sẽ khai báo biến mảng tên list với 5 phần tử có kiểu là số thực (4 byte).
- **Địa chỉ của các phần tử trong mảng một chiều**

<code>list[0]</code>	địa chỉ gốc = $\alpha$
<code>list[1]</code>	$\alpha + \text{sizeof}(\text{float})$
<code>list[2]</code>	$\alpha + 2 * \text{sizeof}(\text{float})$
<code>list[3]</code>	$\alpha + 3 * \text{sizeof}(\text{float})$
<code>list[4]</code>	$\alpha + 4 * \text{size}(\text{float})$



# Ví dụ

Chương trình trên ngôn ngữ C đưa ra địa chỉ các phần tử trong mảng 1 chiều:

```
#include <stdio.h>
int main()
{   int A[ ] = {5, 10, 12, 15, 4};
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", &A[i], A[i]);
}
```

$\&A[i]$  : địa chỉ của phần tử  $A[i]$   
 $A[i]$  : nội dung của phần tử  $A[i]$

Memory       $\text{Location}(A[i]) = \text{start\_address} + W * i$



$\text{start\_address} = 6487536$

**Result in DevC**  
**(sizeof(int)=4)**

Address	Contents
6487536	5
6487540	10
6487544	12
6487548	15
6487552	4

**Result in turboC**  
**(sizeof(int)=2)**

Address	Contents
65516	5
65518	10
65520	12
65522	15
65524	4

# Ví dụ

Chương trình trên ngôn ngữ C đưa ra địa chỉ các phần tử trong mảng 1 chiều:

```
#include <stdio.h>
int main()
{   int A[ ] = {5, 10, 12, 15, 4};
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", &A[i], A[i]);
```

&A[i] : địa chỉ của phần tử A[i]  
A[i] : nội dung của phần tử A[i]

```
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", A+i, *(A+i));
```

A+i : địa chỉ của phần tử A[i]  
\*(A+i) : nội dung của phần tử A[i]

```
/* print the address of 1D array by using pointer */
int *ptr = A;
printf("Address    Contents\n");
for (int i=0; i < 5; i++)
    printf("%8d %5d\n", ptr+i, *(ptr+i));
```

ptr+i : địa chỉ của phần tử A[i]  
\*(ptr+i) : nội dung của phần tử A[i]

```
}
```

# Ví dụ

Chương trình trên ngôn ngữ C đưa ra địa chỉ các phần tử trong mảng 1 chiều:

```
#include <stdio.h>
int main()
{   int A[ ] = {5, 10, 12, 15, 4};
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", &A[i], A[i]);

    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", A+i, *(A+i));

    /* print the address of 1D array by using pointer */
    int *ptr = A;
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", ptr+i, *(ptr+i));
}
```

&A[i]

A[i]

A+i

\*(A+i)

ptr+i

\*(ptr+i)

Address	Contents
6487536	5
6487540	10
6487544	12
6487548	15
6487552	4
Address	Contents
6487536	5
6487540	10
6487544	12
6487548	15
6487552	4
Address	Contents
6487536	5
6487540	10
6487544	12
6487548	15
6487552	4

```

#include <stdio.h>
int main()
{
    int A[ ] = {5, 10, 12, 15, 4};
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", &A[i], A[i]);

    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", A+i, *(A+i));

    /* print the address of 1D array by using pointer */
    int *ptr = A;
    printf("Address    Contents\n");
    for (int i=0; i < 5; i++)
        printf("%8d %5d\n", ptr+i, *(ptr+i));
}

```

So sánh `int *ptr` và `int A[5]`:

Giống nhau: ptr và A đều là **con trỏ**.

Khác nhau: A chiếm giữ **5 vị trí trong bộ nhớ (5 locations)**.

Kí hiệu:

A : 1 con trỏ trỏ tới phần tử A[0] (a pointer to A[0])

(A + i) : 1 con trỏ trỏ tới phần tử A[i] (cũng có thể viết là **&A[i]**)

\*(A + i) : nội dung chứa trong phần tử **A[i]**

# Mảng trong C

```
int list[5], *plist[5];
```

list[5]: mảng gồm 5 số nguyên

list[0], list[1], list[2], list[3], list[4]

\*plist[5]: một mảng gồm 5 con trỏ, mỗi con trỏ trỏ tới 1 số nguyên

plist[0], plist[1], plist[2], plist[3], plist[4]

**Các thành phần của mảng list được lưu trữ trong bộ nhớ tại các địa chỉ:**

list[0]                      địa chỉ gốc =  $\alpha$

list[1]                       $\alpha + \text{sizeof}(\text{int})$

list[2]                       $\alpha + 2 * \text{sizeof}(\text{int})$

list[3]                       $\alpha + 3 * \text{sizeof}(\text{int})$

list[4]                       $\alpha + 4 * \text{size}(\text{int})$

- So sánh `int *list1` và `int list2[5]`:

Giống nhau: list1 và list2 đều là **con trỏ**.

Khác nhau: list2 chiếm giữ **5 vị trí trong bộ nhớ (5 locations)**.

Kí hiệu:

list2                      : 1 con trỏ trỏ tới list2[0] (a pointer to list2[0])

(list2 + i)                : 1 con trỏ trỏ tới phần tử list2[i] (cũng có thể viết là **&list2[i]**)

\*(list2 + i)              : nội dung chứa trong phần tử **list2[i]**

# Khai báo mảng 2 chiều (two-dimensional array)

- Cách khai báo:

`<element-type> <arrayName> [size1][size2];`

Ví dụ: `double a[3][4];`  
giống như bảng 3 dòng 4 cột

dòng 0	a[0][0]	a[0][1]	a[0][2]
dòng 1	a[1][0]	a[1][1]	a[1][2]
dòng 2	a[2][0]	a[2][1]	a[2][2]
dòng 3	a[3][0]	a[3][1]	a[3][2]
	cột 0	cột 1	cột 2

- Cách khởi tạo:

Ví dụ: `int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

```
int a[3][4] = {  
    {1, 2, 3, 4}, /* khởi tạo giá trị cho hàng có chỉ mục là 0 */  
    {5, 6, 7, 8}, /* khởi tạo giá trị cho hàng có chỉ mục là 1 */  
    {9, 10, 11, 12} /* khởi tạo giá trị cho hàng có chỉ mục là 2 */  
};
```

a[0][0] = 1	a[0][1]=2	a[0][2]=3	a[0][3]=4
a[1][0] = 5	a[1][1]=6	a[1][2]=7	a[1][3]=8
a[2][0] = 9	a[2][1]=10	a[2][2]=11	a[2][3]=12

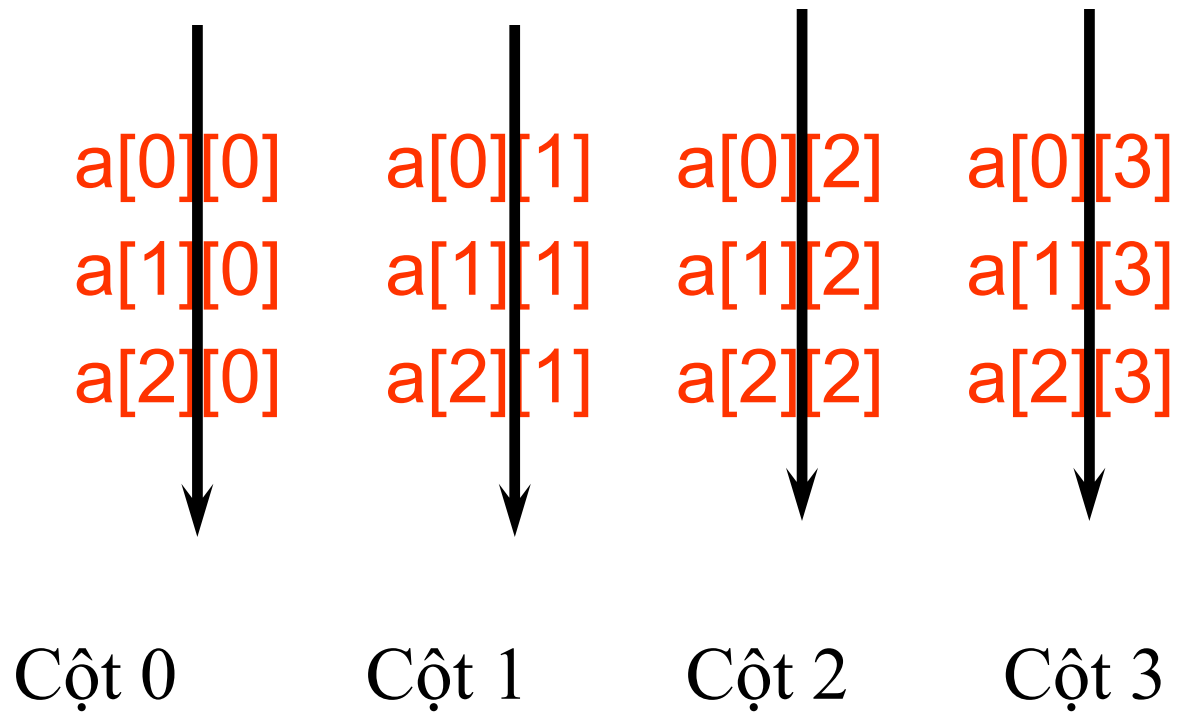
- Truy cập vào 1 phần tử của mảng:

- `a[2][1];`
- `a[i][j];`

# Các dòng của mảng 2 chiều

~~a[0][0]~~ ~~a[0][1]~~ ~~a[0][2]~~ ~~a[0][3]~~ → dòng 0  
~~a[1][0]~~ ~~a[1][1]~~ ~~a[1][2]~~ ~~a[1][3]~~ → dòng 1  
~~a[2][0]~~ ~~a[2][1]~~ ~~a[2][2]~~ ~~a[2][3]~~ → dòng 2

# Các cột của mảng 2 chiều





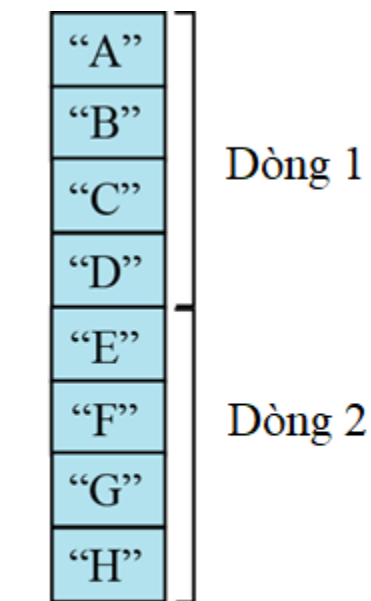
# Bài tập

Hãy khai báo và khởi tạo mảng **a** gồm các số nguyên có giá trị như sau:

```
Gia tri cua a[0][0] la: 0
Gia tri cua a[0][1] la: 0
Gia tri cua a[1][0] la: 1
Gia tri cua a[1][1] la: 2
Gia tri cua a[2][0] la: 2
Gia tri cua a[2][1] la: 4
Gia tri cua a[3][0] la: 3
Gia tri cua a[3][1] la: 6
Gia tri cua a[4][0] la: 4
Gia tri cua a[4][1] la: 8
```

# Phân bổ bộ nhớ cho mảng

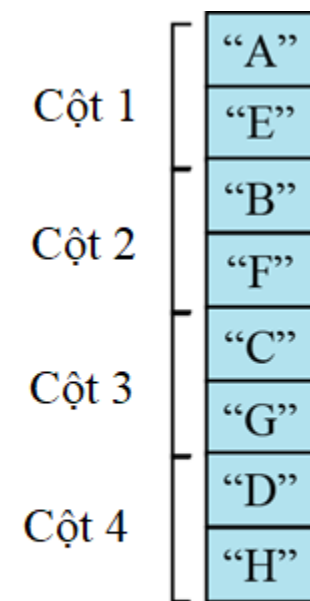
- Trong bộ nhớ, các phần tử của mảng đa chiều thường được lưu trữ kế tiếp nhau theo một trong 2 cách sau:
  - Hết dòng này đến dòng khác: thứ tự này được gọi là thứ tự ưu tiên dòng (**row major order**)
  - Hết cột này đến cột khác: thứ tự này được gọi là thứ tự ưu tiên cột (**column major order**)



Lưu trữ theo  
thứ tự ưu tiên dòng

	[1]	[2]	[3]	[4]
[1]	"A"	"B"	"C"	"D"
[2]	"E"	"F"	"G"	"H"

User's View



Lưu trữ theo  
thứ tự ưu tiên cột

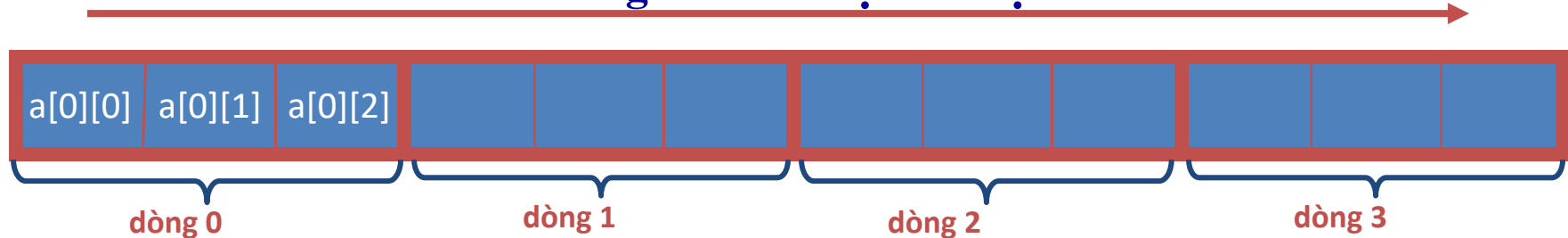
# Thứ tự ưu tiên dòng (Ví dụ: Pascal, C/C++)

- Các phần tử của mảng nhiều chiều được lưu trữ kế tiếp nhau theo bộ nhớ, hết dòng này đến dòng khác. Vì vậy, phần tử ở dòng đầu tiên sẽ chiếm tập các vị trí ô nhớ đầu tiên của mảng, các phần tử ở dòng thứ 2 sẽ chiếm tập các ô nhớ tiếp theo,... cho đến dòng cuối cùng

Các phần tử của dòng 0	Các phần tử của dòng 1	Các phần tử của dòng 2	....	Các phần tử của dòng i	.....
------------------------	------------------------	------------------------	------	------------------------	-------

- Ví dụ: `int a[4][3]`

**Theo chiều tăng dần của địa chỉ bộ nhớ**



- Ví dụ mảng 3 x 4:
- |   |   |   |   |
|---|---|---|---|
| a | b | c | d |
| e | f | g | h |
| i | j | k | l |

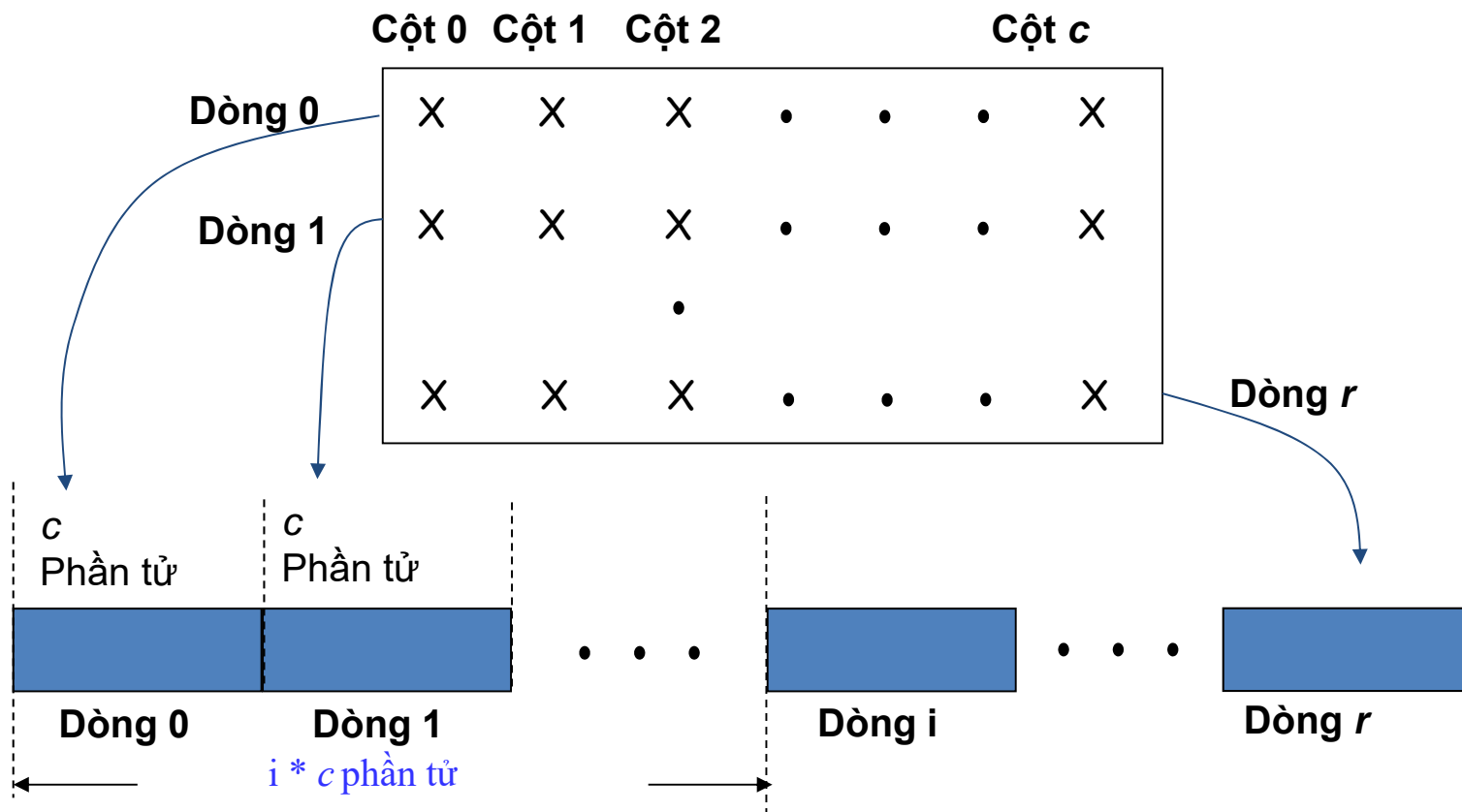
được đổi về mảng 1 chiều **Y** bằng cách gom các phần tử theo dòng:

- Trong mỗi dòng: các phần tử được gom từ trái sang phải.
- Các dòng được gom từ trên xuống dưới.

Khi đó, ta có mảng **Y**[ ] =

{a, b, c, d, e, f, g, h, i, j, k, l}

Mảng 2 chiều theo thứ tự ưu tiên dòng



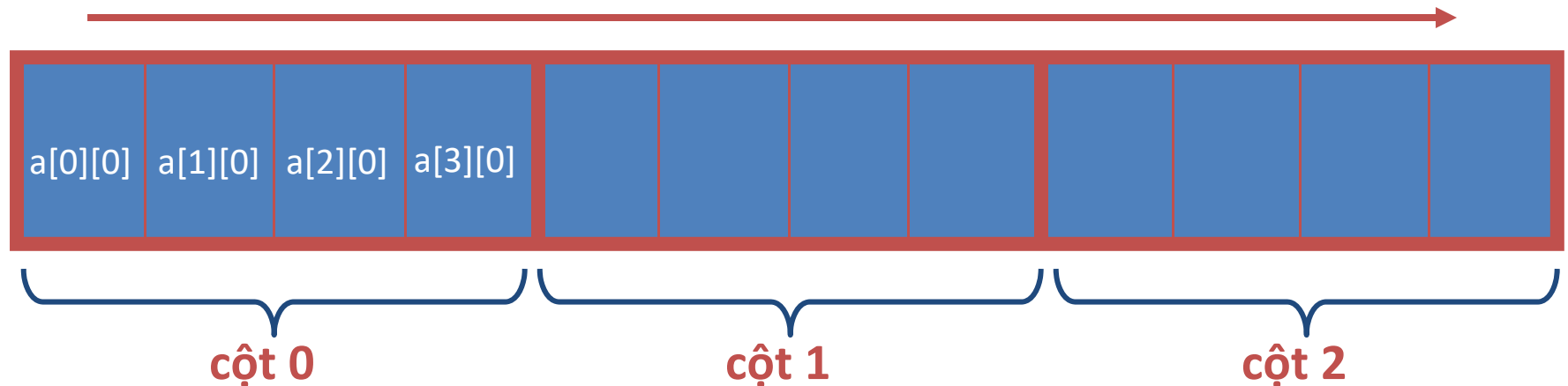
# Thứ tự ưu tiên cột (ví dụ: Matlab, Fortran)

- Các phần tử của mảng nhiều chiều được lưu trữ kế tiếp nhau theo bộ nhớ, hết cột này đến cột khác. Vì vậy, các phần tử ở cột đầu tiên sẽ chiếm tập các vị trí ô nhớ đầu tiên của mảng, các phần tử ở cột thứ 2 sẽ chiếm tập các ô nhớ tiếp theo,... cho đến cột cuối cùng

Các phần tử của cột 0	Các phần tử của cột 1	Các phần tử của cột 2	....	Các phần tử của cột i	.....
-----------------------	-----------------------	-----------------------	------	-----------------------	-------

- Ví dụ: `int a[4][3];`

**Theo chiều tăng dần của địa chỉ bộ nhớ**



# Thứ tự ưu tiên cột (ví dụ: Matlab, Fortran)

- Các phần tử của mảng nhiều chiều được lưu trữ kế tiếp nhau trong bộ nhớ, hết cột này đến cột khác. Vì vậy, các phần tử ở cột đầu tiên sẽ chiếm tập các vị trí ô nhớ đầu tiên của mảng, các phần tử ở cột thứ 2 sẽ chiếm tập các ô nhớ tiếp theo,... cho đến cột cuối cùng

Các phần tử của cột 0	Các phần tử của cột 1	Các phần tử của cột 2	....	Các phần tử của cột i	.....
-----------------------	-----------------------	-----------------------	------	-----------------------	-------

- Ví dụ mảng 3 x 4 :

a b c d  
e f g h  
i j k l

Được chuyển về mảng 1 chiều **Y** bằng cách gom các phần tử theo cột.

- Trong cùng 1 cột: các phần tử được gom từ trên xuống dưới.
- Các cột được gom từ trái sang phải.

Vì vậy, ta thu được mảng **Y**[ ] =

{a, e, i, b, f, j, c, g, k, d, h, l}

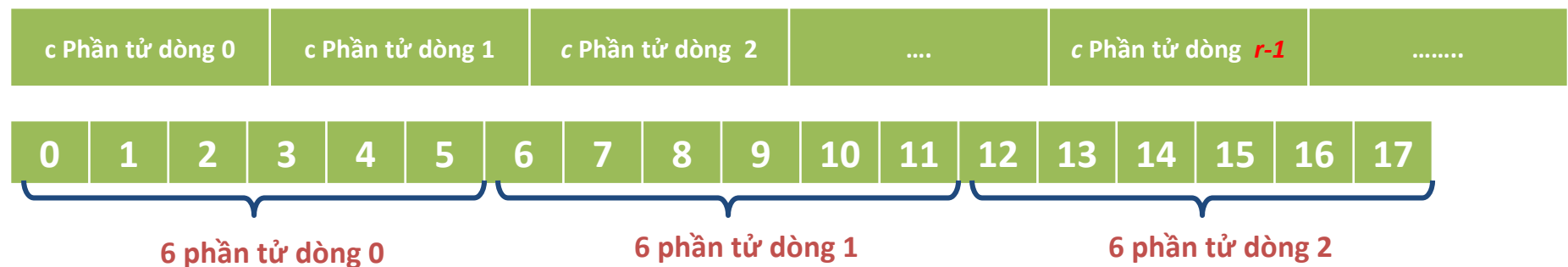
## Ví dụ: Phân bổ bộ nhớ mảng 2 chiều: theo thứ tự ưu tiên dòng và thứ tự ưu tiên cột

Mảng 2 chiều:  $r$  dòng,  $c$  cột

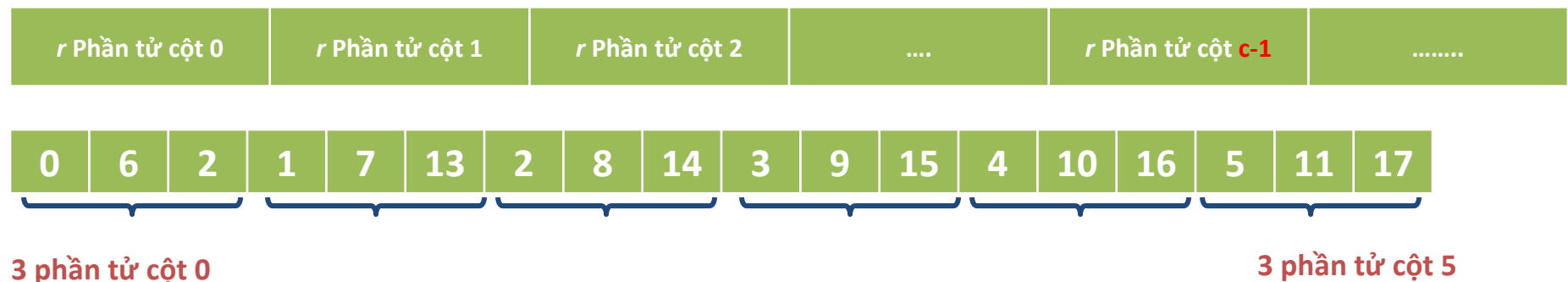
Ví dụ: `int a[3][6]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};`

$a[0][0]=0$     $a[0][1]=1$     $a[0][2]=2$     $a[0][3]=3$     $a[0][4]=4$     $a[0][5]=5$   
 $a[1][0]=6$     $a[1][1]=7$     $a[1][2]=8$     $a[1][3]=9$     $a[1][4]=10$     $a[1][5]=11$   
 $a[2][0]=12$     $a[2][1]=13$     $a[2][2]=14$     $a[2][3]=15$     $a[2][4]=16$     $a[2][5]=17$

### Phân bổ bộ nhớ theo thứ tự ưu tiên dòng



### Phân bổ bộ nhớ theo thứ tự ưu tiên cột



# Xác định địa chỉ phần tử $x[i][j]$ : thứ tự ưu tiên dòng

- Giả sử mảng  $x$ :
  - gồm  $r$  dòng và  $c$  cột (tức là, mỗi dòng gồm  $c$  phần tử)

c Phần tử dòng 0	c Phần tử dòng 1	c Phần tử dòng 2	....	c Phần tử dòng $r-1$	.....
------------------	------------------	------------------	------	----------------------	-------

- Xác định địa chỉ phần tử  $x[i][j]$ :
  - $i$  dòng nằm trước dòng  $j \rightarrow$  có  $i*c$  phần tử nằm trước phần tử  $x[i][0]$
  - $x[i][j]$  đặt ở vị trí:  $i*c + j$  của mảng 1 chiều
  - Địa chỉ của phần tử  $x[i][j]$ :

$$\text{Location}(x[i][j]) = \text{start\_address} + W (i*c + j)$$

Với

- $\text{start\_address}$ : địa chỉ của phần tử đầu tiên ( $x[0][0]$ ) trong mảng
- $W$ : kích thước 1 phần tử
- $c$ : số cột của mảng
- $r$ : số dòng của mảng



# Ví dụ

Chương trình trên ngôn ngữ C in địa chỉ của các phần tử thuộc mảng 2 chiều:

```
#include <stdio.h>
int main()
{ int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
  printf("Address    Contents\n");
  for (int i=0; i < 3; i++)
    for (int j=0; j < 4; j++)
      printf("%8d %5d\n", &a[i][j], a[i][j]);
}
```

**Result in DevC**

(sizeof(int)=4)

Address	Contents
6487488	1
6487492	2
6487496	3
6487500	4
6487504	5
6487508	6
6487512	7
6487516	8
6487520	9
6487524	10
6487528	11
6487532	12

**Bộ nhớ**

$$\text{Location}(a[i][j]) = \text{start\_address} + W * [(i * \text{cols}) + j]$$



↑  
start\_address=6487488

Location(a[1][2]) = ?

## Chương trình trên ngôn ngữ C in địa chỉ của các phần tử thuộc mảng 2 chiều:

```
#include <stdio.h>
int main()
{   int   a[3] [4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

```
    printf("Address      Contents\n");
    for (int i=0; i < 3; i++)
        for (int j=0; j < 4; j++)
            printf("%8d %5d\n", &a[i][j], a[i][j]);
```

Bộ nhớ

$\text{Location}(a[i][j]) = \text{start\_address} + W * [(i * \text{cols}) + j]$

		1	2	3	4	5	6	7	8	9	10	11	12			
--	--	---	---	---	---	---	---	---	---	---	----	----	----	--	--	--



start\_address = địa chỉ phần tử a[0][0]

```
int *ptr = &a[0][0];
printf("Address      Contents\n");
for (int i=0; i < rows; i++)
    for (int j=0; j < cols; j++)
        printf("%8d %5d\n",
```

.....)

????

## Ví dụ thứ tự ưu tiên dòng: phân bổ bộ nhớ cho mảng 2 chiều (kiểu số nguyên int)

- Địa chỉ của các phần tử trong mảng 2 chiều:

**int a[4][3]**

a[0][0]	địa chỉ = $\alpha$
a[0][1]	$\alpha + 1 * \text{sizeof}(\text{int})$
a[0][2]	$\alpha + 2 * \text{sizeof}(\text{int})$
a[1][0]	$\alpha + 3 * \text{sizeof}(\text{int})$
a[1][1]	$\alpha + 4 * \text{sizeof}(\text{int})$
a[1][2]	$\alpha + 5 * \text{sizeof}(\text{int})$
a[2][0]	$\alpha + 6 * \text{sizeof}(\text{int})$
...	

### Tổng quát: Khai báo

**int a[m][n];**

- Giả sử: địa chỉ của phần tử đầu tiên (a[0][0]) là  $\alpha$ .
- Khi đó, địa chỉ của phần tử a[i][j] là:

**$\alpha + (i * n + j) * \text{sizeof}(\text{int})$**

# Xác định địa chỉ phần tử $x[i][j]$ : thứ tự ưu tiên cột

- Giả sử mảng  $x$ :

.....	$r$ Phần tử của cột 0	$r$ Phần tử của cột 1	$r$ Phần tử của cột 2	....	$r$ Phần tử của cột $j$	.....
-------	--------------------------	--------------------------	--------------------------	------	----------------------------	-------

- có  $r$  dòng và  $c$  cột (mỗi cột có  $r$  phần tử)
- Địa chỉ của phần tử  $x[i][j]$ :
  - $j$  cột nằm trước cột  $j \rightarrow$  do đó có  $j*r$  phần tử nằm trước phần tử  $x[0][j]$
  - $x[i][j]$  nằm ở vị trí:  $j*r + i$  tính từ phần tử đầu tiên của mảng
  - $\text{Location}(x[i][j]) = \text{start\_address} + W(j*r + i)$

Với

- $\text{start\_address}$ : địa chỉ của phần tử đầu tiên  $x[0][0]$  của mảng
- $W$ : kích thước mỗi phần tử
- $c$ : số lượng cột của mảng
- $r$ : số lượng dòng của mảng

Ví dụ: mảng : `int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};`

Tìm địa chỉ phần tử  $a[1][2]$  trong bộ nhớ nếu  $\text{start\_address} = 6487488$

# Ví dụ 1: Thứ tự ưu tiên dòng và thứ tự ưu tiên cột

Mảng 2 chiều:

`int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

$a[0][0] = 1$	$a[0][1] = 2$	$a[0][2] = 3$	$a[0][3] = 4$
$a[1][0] = 5$	$a[1][1] = 6$	$a[1][2] = 7$	$a[1][3] = 8$
$a[2][0] = 9$	$a[2][1] = 10$	$a[2][2] = 11$	$a[2][3] = 12$

Phân bố bộ nhớ theo thứ tự ưu tiên dòng

$$\text{Location}(a[i][j]) = \text{start\_address} + W * [(i * \text{cols}) + j]$$

		1	2	3	4	5	6	7	8	9	10	11	12			
--	--	---	---	---	---	---	---	---	---	---	----	----	----	--	--	--



$\text{start\_address} = 6487488$

$\text{Location}(a[1][2]) = ?$

Phân bố bộ nhớ theo thứ tự ưu tiên cột

$$\text{Location}(a[i][j]) = \text{start\_address} + W * [(j * \text{rows}) + i]$$

		1	5	9	2	6	10	3	7	11	4	8	12			
--	--	---	---	---	---	---	----	---	---	----	---	---	----	--	--	--



$\text{start\_address} = 6487488$

$\text{Location}(a[1][2]) = ?$

# Ví dụ 2: Thứ tự ưu tiên dòng và thứ tự ưu tiên cột

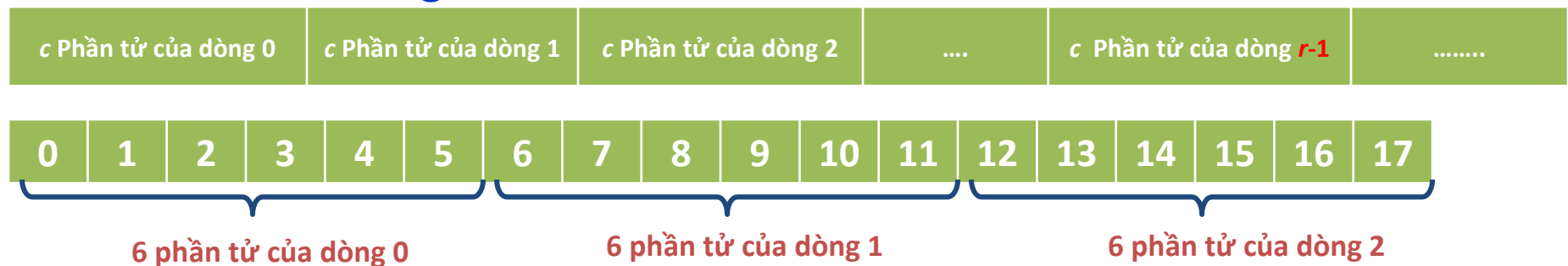
Mảng 2 chiều:  $r$  dòng,  $c$  cột

Ví dụ: `int a[3][6]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};`

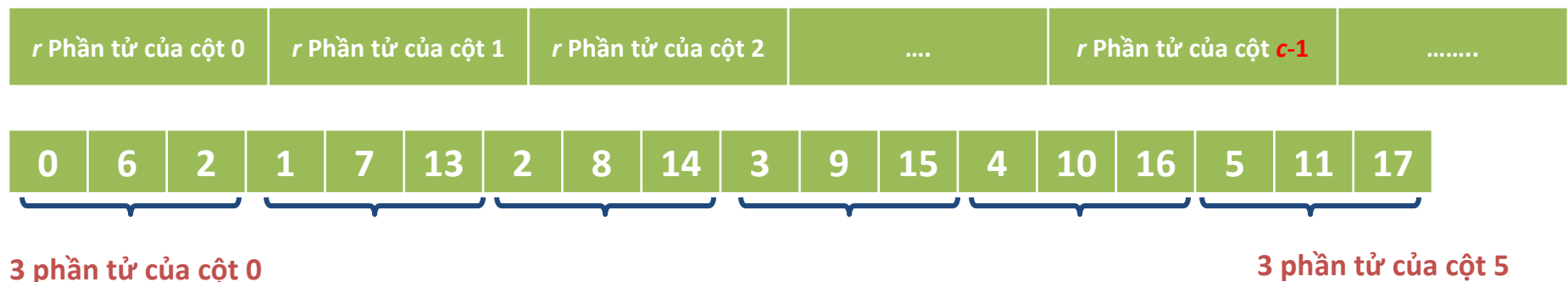
$a[0][0]=0$     $a[0][1]=1$     $a[0][2]=2$     $a[0][3]=3$     $a[0][4]=4$     $a[0][5]=5$   
 $a[1][0]=6$     $a[1][1]=7$     $a[1][2]=8$     $a[1][3]=9$     $a[1][4]=10$     $a[1][5]=11$   
 $a[2][0]=12$     $a[2][1]=13$     $a[2][2]=14$     $a[2][3]=15$     $a[2][4]=16$     $a[2][5]=17$

## Thứ tự ưu tiên dòng

`start_address = 1000` → `Location(a[1][4]) = ?`

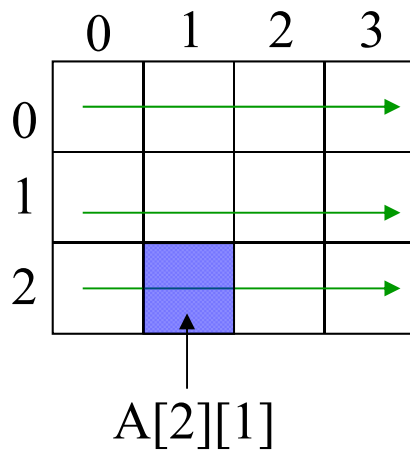


## Thứ tự ưu tiên cột



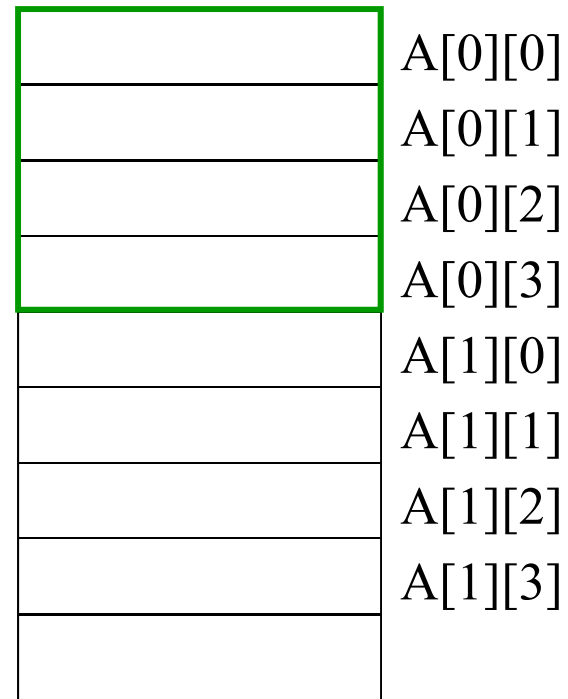
# Phân bổ bộ nhớ

- `char A[3][4];`
- Cấu trúc logic



// Thứ tự ưu tiên dòng

Cấu trúc vật lý



Phân bổ bộ nhớ:

$$\text{Location}(A[i][j]) = \text{Location}(A[0][0]) + i*4 + j$$

## Bài tập

Giả sử trong bộ nhớ có một mảng 2 chiều **students** lưu trữ thông tin sinh viên. Kích thước của mảng là  $100 \times 4$  (100 dòng, 4 cột). Hãy xác định địa chỉ của phần tử `students[5][3]` được lưu trong bộ nhớ biết rằng:

- phần tử `students[0][0]` được lưu ở địa chỉ 1000
- mỗi phần tử chiếm 2 bytes bộ nhớ.

Máy tính phân bổ bộ nhớ theo thứ tự ưu tiên dòng.



# Các thao tác trên mảng

- Các thao tác cơ bản trên mảng bao gồm: tìm kiếm (search), thêm/chèn (insert), xóa (delete), truy xuất thông tin (Retrieval), duyệt (traversal).

Ví dụ: Mảng  $S$  gồm  $n$  số nguyên:  $S[0], S[1], \dots, S[n-1]$

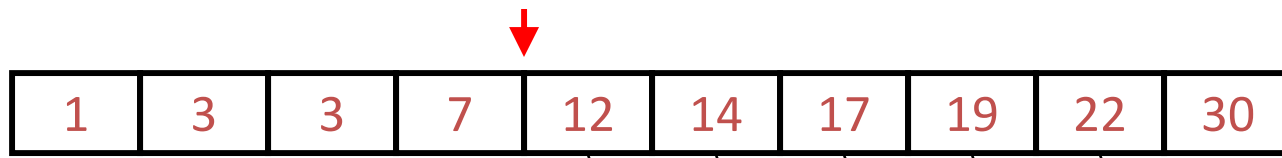
- **Tìm kiếm** : tìm xem giá trị **key** có xuất hiện trong mảng  $S$  hay không  
function  $\text{Search}(S, \text{key})$  trả về giá trị true nếu **key** có trong  $S$ ; false nếu ngược lại
- **Truy xuất thông tin**: xác định giá trị của phần tử tại chỉ số **i** trong mảng  $S$   
function  $\text{Retrieve}(S, i)$  : trả về giá trị của phần tử  $S[i]$  nếu  $0 \leq i \leq n-1$
- **Duyệt**: in ra giá trị của tất cả các phần tử thuộc mảng  $S$   
function  $\text{PrintArray}(S, n)$
- **Thêm**: thêm giá trị **key** vào mảng  $S$
- **Xóa**: xóa phần tử tại chỉ số **i** của mảng  $S$

# Các thao tác trên mảng

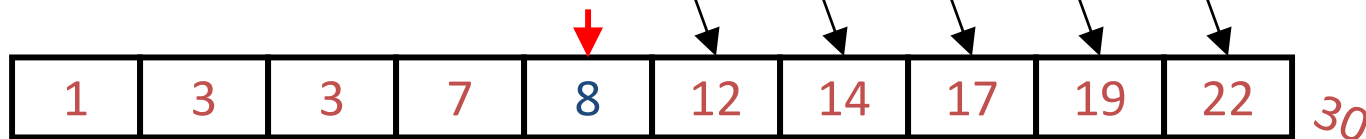
- Các thao tác cơ bản trên mảng bao gồm: tìm kiếm (search), thêm/chèn (insert), xóa (delete), truy xuất thông tin (Retrieval), duyệt (traversal).
- Các thao tác:
  - tìm kiếm, truy xuất thông tin và duyệt mảng là những thao tác đơn giản;
  - thêm và xóa phần tử của mảng tốn thời gian hơn, vì:
    - Trước khi thêm 1 phần tử vào mảng, ta cần dịch chuyển các phần tử sau vị trí chèn về đằng sau (bên phải) 1 vị trí;
    - Sau khi xóa 1 phần tử thuộc mảng, ta cần đẩy các phần tử sau phần tử xóa về phía trước (bên trái) 1 vị trí

# Thêm 1 phần tử vào mảng

- Giả sử cần thêm giá trị 8 vào mảng đã được sắp xếp theo thứ tự tăng dần:



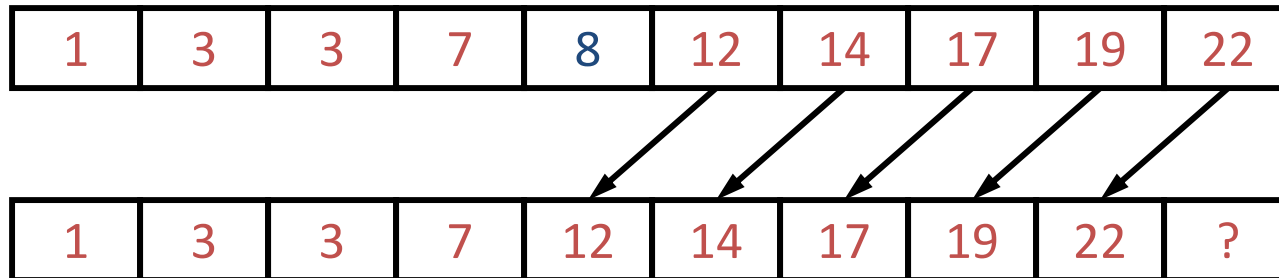
- Dịch chuyển các phần tử sau dấu mũi tên về bên phải 1 vị trí
  - Khi đó, số 30 sẽ bị xóa khỏi mảng (vì mảng có kích thước cố định)



- Việc dịch chuyển các phần tử trong mảng là một thao tác tốn thời gian (đòi hỏi thời gian tuyến tính cỡ  $O(n)$  với  $n$  là kích thước của mảng)

# Xóa 1 phần tử khỏi mảng

- Để xóa 1 phần tử khỏi mảng, ta cần dịch các phần tử phía trước phần tử xóa 1 vị trí về bên trái



- Thao tác xóa là một thao tác tốn thời gian.
- Vì vậy, khi thiết kế thuật toán nếu phải thực hiện thao tác này 1 cách thường xuyên là điều không mong muốn.
- Thao tác xóa làm cho phần tử cuối cùng bị trống
  - Vậy làm thế nào để đánh dấu là phần tử cuối cùng đã trống?
    - Ta cần có biến lưu trữ kích thước của mảng

Ví dụ: biến *size* được dùng để lưu trữ kích thước của mảng. Trước xóa, biến *size* = 10. Sau khi xóa 1 phần tử của mảng, ta cần cập nhật giá trị biến *size*:  $size = 10 - 1 = 9$

# Các thao tác trên mảng

Các thao tác trên mảng đã thảo luận trong phần trước cho ta gợi ý khi nào thì nên sử dụng mảng ?:

- Nếu chúng ta có một danh sách và cần phải thực hiện rất nhiều thao tác thêm/xóa trên danh sách này thì không nên sử dụng mảng
- Nên dùng mảng khi chỉ cần thực hiện ít các thao tác thêm/xóa, nhưng nhiều thao tác tìm kiếm/truy xuất thông tin



Mảng là cấu trúc phù hợp cho trường hợp chỉ thực hiện một số lượng nhỏ các thao tác thêm/xóa, nhưng cần thực hiện nhiều thao tác tìm kiếm/truy xuất thông tin.

# Biểu diễn ma trận bởi mảng

- **Ma trận  $m \times n$**  là một bảng gồm  $m$  dòng và  $n$  cột, nhưng được đánh chỉ số bắt đầu từ 1 thay vì 0.
  - Kí hiệu  $M(i,j)$  là phần tử dòng row  $i$  cột  $j$  của ma trận
  - Các thao tác cơ bản trên ma trận
    - Chuyển vị
    - Cộng/trừ
    - Nhân
  - Ta sẽ sử dụng mảng 2 chiều để biểu diễn ma trận:
    - Chỉ số được đánh số từ 1.
    - Cấu trúc mảng trong ngôn ngữ C không hỗ trợ các thao tác cộng, trừ, nhân, chuyển vị.
      - Giả sử  $x$  và  $y$  là các mảng 2 chiều. Ta không thể gọi lệnh  $x + y$ ,  $x - y$ ,  $x * y$ , ... trong ngôn ngữ C.
- Ta sẽ phải viết các hàm thực hiện các thao tác trên ma trận.

	col 1	col 2	col 3	col 4
row 1	7	2	0	9
row 2	0	1	0	5
row 3	6	4	2	0
row 4	8	2	7	3
row 5	1	4	9	6

# Ma trận thưa (Sparse Matrix)

	col 1	col 2	col 3
row 1	-27	3	4
row 2	6	82	-2
row 3	109	-64	11
row 4	12	8	9
row 5	48	27	47

5\*3

15/15

	col1	col2	col3	col4	col5	col6
row1	15	0	0	22	0	-15
row2	0	11	3	0	0	0
row3	0	0	0	-6	0	0
row4	0	0	0	0	0	0
row5	91	0	0	0	0	0
row6	0	0	28	0	0	0

6\*6

8/36

Ma trận thưa

Nên dùng cấu trúc dữ liệu nào?

# Ma trận thưa (Sparse Matrix)

(Cách1) Biểu diễn bởi mảng 2 chiều (ví dụ: `int M[6][6]`; ma trận kích thước 6x6)

- Ma trận thưa sẽ gây tốn bộ nhớ không cần thiết.

(Cách2) Mỗi phần tử được biểu diễn bởi bộ 3 <dòng, cột, giá trị>

- Các phần tử sẽ được sắp xếp theo thứ tự dựa trên giá trị <dòng, cột>

	Cột1	cột2	cột3	cột4	cột5	cột6			
dòng1	15	0	0	22	0	-15	<i>dòng</i>	<i>cột</i>	<i>giá trị</i>
dòng2	0	11	3	0	0	0	<i>A</i>	<i>[0]</i>	
dòng3	0	0	0	-6	0	0		<i>[1]</i>	
dòng4	0	0	0	0	0	0		<i>[2]</i>	
dòng5	91	0	0	0	0	0		<i>[3]</i>	
dòng6	0	0	28	0	0	0		<i>[4]</i>	
								<i>[5]</i>	
								<i>[6]</i>	
								<i>[7]</i>	

6\*6

<i>A</i>	<i>[0]</i>	1	1	15
	<i>[1]</i>	1	4	22
	<i>[2]</i>	1	6	-15
	<i>[3]</i>	2	2	11
	<i>[4]</i>	2	3	3
	<i>[5]</i>	3	4	-6
	<i>[6]</i>	5	1	91
	<i>[7]</i>	6	2	28

*A*[0][0] = 1;  
*A*[0][1] = 1;  
*A*[0][2] = 15;  
  
*A*[5][0] = 3;  
*A*[5][1] = 4;  
*A*[5][2] = -6;

/\* Sử dụng mảng biểu diễn ma trận thưa

//[ ][0] biểu diễn dòng

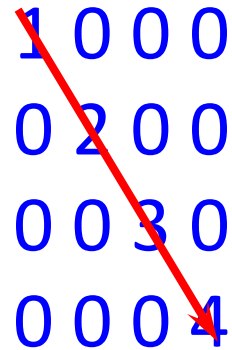
//[ ][1] biểu diễn cột

//[ ][2] biểu diễn giá trị phần tử \*/

```
int MAX = 8; //số lượng phần tử có giá trị != 0 trong ma trận thưa
int A[MAX][3];
```



# Ma trận đường chéo (Diagonal Matrix)



1	0	0	0
0	2	0	0
0	0	3	0
0	0	0	4

- Ma trận  **$n \times n$**  các phần tử có giá trị khác 0 nằm trên đường chéo.
  - $M(i, j)$  là phần tử thuộc đường chéo khi và chỉ khi  $i = j$
  - Số phần tử trên đường chéo của ma trận  $n \times n$  là  $n$
  - Các phần tử không thuộc đường chéo đều bằng 0
- 2 cách biểu diễn:
  - Cách 1: Chỉ lưu trữ  $n$  phần tử trên đường chéo
  - Cách 2: Lưu trữ tất cả  $n^2$  phần tử của ma trận

• `int M[5];`

• `int M[5][5];`

# Ma trận tam giác (Triangular Matrix)

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

- Ma trận kích thước  $n \times n$  có các phần tử khác 0 đều nằm bên dưới hoặc bên trên của đường chéo chính.
  - Phần tử  $M(i,j)$  là phần tử thuộc ma trận tam giác dưới khi và chỉ khi  $i \geq j$ 
    - Số lượng phần tử có giá trị khác 0 của ma trận tam giác dưới là:
$$1 + 2 + \dots + n = n(n+1)/2$$
- 2 cách biểu diễn ma trận tam giác dưới:
  - Cách 1: Lưu trữ tất cả  $n^2$  phần tử:
    - Sử dụng mảng 2 chiều  $A[n][n]$
  - Cách 2: Chỉ lưu trữ các phần tử nằm ở bên dưới đường chéo:
    - Lựa chọn 1: Lưu trữ các phần tử đó vào 1 mảng 1 chiều
    - Lựa chọn 2: Lưu trữ các phần tử đó vào mảng 2 chiều

Lựa chọn 1: Lưu trữ các phần tử nằm ở bên dưới đường chéo vào mảng 1 chiều

Sử dụng mảng 1 chiều theo thứ tự ưu tiên hàng

Ví dụ: Cho ma trận tam giác dưới kích thước 4x4

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

Theo cách biểu diễn này, ta sẽ lưu trữ ma trận trên bằng mảng 1 chiều gồm các phần tử nằm bên dưới đường chéo theo thứ tự:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

# Chỉ số của phần tử $M(i,j)$ trong mảng 1 chiều

Ma trận tam giác dưới  $M_{4 \times 4}$

1 0 0 0  $\longrightarrow$  Hàng 1

2 3 0 0

4 5 6 0

7 8 9 10

Được lưu trữ bằng mảng 1 chiều:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

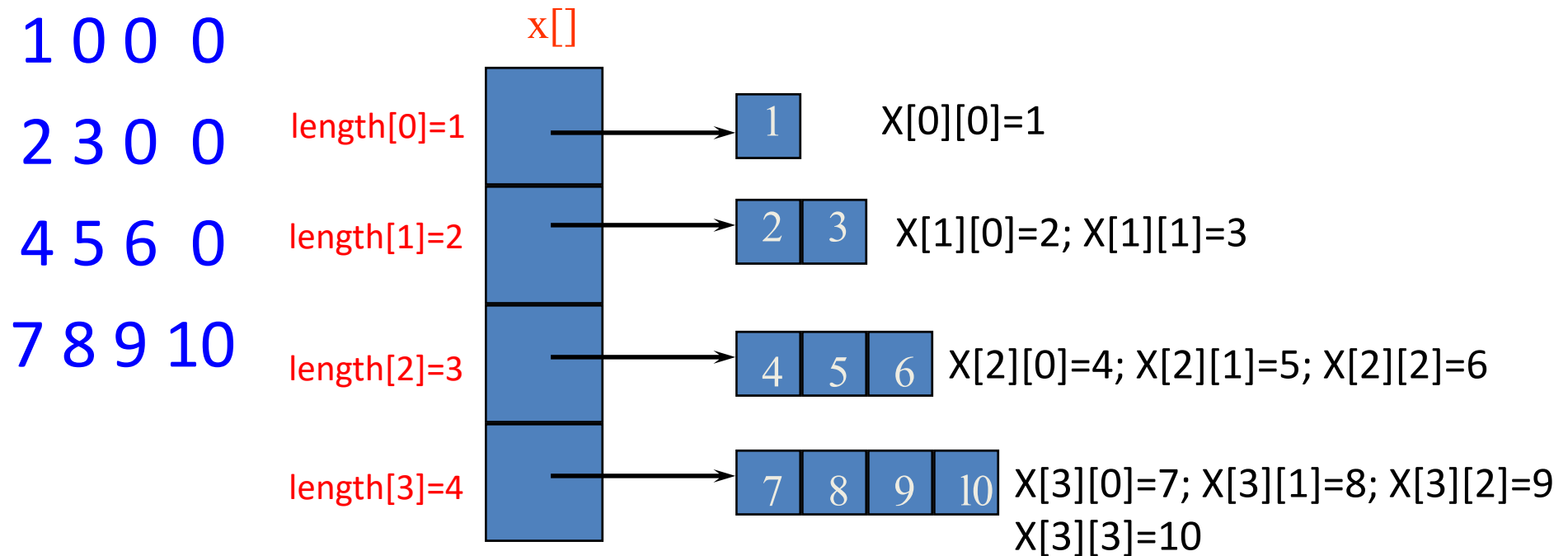
0 1 3 6

h1	h2	h3	...	hàng i		
----	----	----	-----	--------	--	--

- Thứ tự lần lượt là các phần tử thuộc: hàng 1, hàng 2, hàng 3, ...
    - Hàng  $i$  sẽ đi sau các hàng 1, 2, ...,  $i-1$
  - Số lượng phần tử của hàng  $i$  được lưu trữ trong mảng 1 chiều là  $i$
  - Số lượng phần tử lưu trữ trước hàng  $i$  là
$$1 + 2 + 3 + \dots + i-1 = i(i-1)/2$$
- ➔ Phần tử  $M(i,j)$  của ma trận đã cho nằm ở vị trí  $i(i-1)/2 + j-1$  của mảng 1 chiều
- ví dụ:  $M(3, 2)$  nằm ở vị trí ?
$$3(3-1)/2 + 2-1 = 4$$

## Lựa chọn 2: Lưu trữ các phần tử dưới đường chéo bởi mảng 2 chiều không thường quy

Chỉ lưu trữ các phần tử dưới đường chéo:



Mảng 2 chiều không thường quy: độ dài (số phần tử) ở mỗi hàng của mảng không nhất thiết phải bằng nhau.

Vì số phần tử ở mỗi hàng khác nhau → cần thêm 1 biến lưu trữ số lượng phần tử của từng hàng → thêm mảng 1 chiều `length[]`

# Khai báo và sử dụng mảng 2 chiều không thường quy

//BUỐC 1: khai báo mảng 2 chiều

```
int ** iArray = new int*[numberOfRows];
```

Hoặc:

```
int ** iArray;
```

```
malloc(iArray, numberOfRows*sizeof(*iArray));
```

//BUỐC 2: xác định số lượng phần tử cho mỗi hàng

// cấp phát bộ nhớ cho các phần tử ở mỗi hàng

```
for (int i = 0; i < numberOfRows; i++)
```

```
    iArray[i] = new int [length[i]];
```

Hoặc:

```
for (int i = 0; i < numberOfRows; i++)
```

```
    malloc(iArray[i], length[i]*sizeof(int));
```

//BUỐC 3: sử dụng mảng :

```
iArray[2][3] = 5;
```

```
iArray[4][6] = iArray[2][3]+2;
```

```
iArray[1][1] += 3;
```

# Bài tập

Cho ma trận tam giác dưới kích thước 8x8 trong đó các phần tử khác 0 của ma trận này sẽ có giá trị lần lượt là 1, 2, 3,.. (xếp lần lượt từng dòng, từng cột). Hãy viết chương trình trên C lưu trữ ma trận này theo các cách đã học, sau đó in ma trận ra màn hình

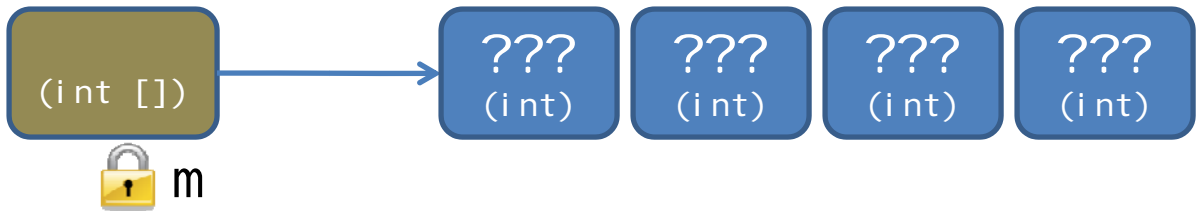
1	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0
4	5	6	0	0	0	0	0
7	8	9	10	0	0	0	0
11	12	13	14	0	0	0	0
15	16	17	18	19	0	0	0
20	21	22	23	24	25	0	0
26	27	28	29	30	31	32	0
33	34	35	36	37	38	39	40

# Con trỏ và mảng

Giả sử ta có : `int m[4];`

➔ Tên của mảng chính là **1 hằng địa chỉ** = địa chỉ của phần tử đầu tiên của mảng:

- $m$  tương đương với  $\&m[0]$ ;
- $m+i$  tương đương với  $\&m[i]$ ;
- $m[i] \sim$  tương đương với  $*(m+i)$



➔  $m$  là 1 hằng  $\Rightarrow$  không thể dùng nó trong câu lệnh gán hay toán tử tăng ( $m++;$ ), giảm ( $m--;$ )

Xét con trỏ: `int *pa;`

$pa = \& m[0]; \Rightarrow$   $pa$  trỏ vào phần tử thứ nhất của mảng  $m$

$pa + 1$  sẽ trỏ vào phần tử thứ 2 của mảng

$*(pa+i)$  sẽ là nội dung của  $m[i]$  (phần tử thứ  $i$  trong mảng  $m$ )



## Con trỏ chuỗi (strings)

```
char tinhthanh[30] = "Da Lat";
```

Tương đương : 

```
char *tinhthanh;  
tinhthanh = "Da lat";
```

Hoặc : 

```
char *tinhthanh = "Da lat";
```

- Ngoài ra các thao tác trên chuỗi cũng tương tự như trên mảng:  

```
*(tinhthanh+3)
```

 có giá trị là kí tự "l"

# Mảng các con trỏ

- Con trỏ cũng là một loại dữ liệu nên ta có thể tạo một mảng các phần tử là con trỏ theo dạng thức:

```
type *pointer_name[size];
```

Ví dụ: `char *ds[3];`

- ➔ `ds` là 1 mảng gồm 3 phần tử, mỗi phần tử là 1 con trỏ kiểu `char`, được dùng để lưu trữ 3 chuỗi ký tự nào đó
- ➔ Cũng có thể khởi tạo trực tiếp các giá trị khi khai báo:

```
char * ds[3] = {"mot", "hai", "ba"};
```

# Mảng các con trỏ

- Ưu điểm của mảng trỏ là ta có thể hoán chuyển các đối tượng (mảng con, cấu trúc..) được trỏ bởi con trỏ này bằng cách hoán chuyển các con trỏ

Ví dụ: Vào danh sách lớp theo họ và tên, sau đó sắp xếp để in ra theo thứ tự ABC

**for (i=0;i<count; i++) printf("\n %d : %s", i+1, ds[i]);**

```
#include <stdio.h>
#include <string.h>
#define MAXHS 50
#define MAXLEN 30
```

```
int main () {
    int i, j, count = 0;
    char ds[MAXHS][MAXLEN];
    char *ptr[MAXHS], *tmp;

    while ( count < MAXHS) {
        printf("Vao hoc sinh thu %d: ", count+1);
        gets(ds[count]);
        if (strlen(ds[count]) == 0) break;
        ptr[count] = ds + count;
        ++count;
    }
```

```
for ( i=0;i<count-1;i++)
    for ( j =i+1;j < count; j++)
        if (strcmp(ptr[i],ptr[j])>0) {
            tmp=ptr[i]; ptr[i] = ptr[j]; ptr[j] = tmp;
        }
```

```
for (i=0;i<count; i++)
    printf("\n %d : %s", i+1, ptr[i]);
```

```
Vao hoc sinh thu 1: Thang
Vao hoc sinh thu 2: Mai
Vao hoc sinh thu 3: Huy
Vao hoc sinh thu 4:
```

```
1 : Huy
2 : Mai
3 : Thang
```

```
Vao hoc sinh thu 1: Thang
Vao hoc sinh thu 2: Mai
Vao hoc sinh thu 3: Huy
Vao hoc sinh thu 4:
```

```
1 : Thang
2 : Mai
3 : Huy
```

Dừng nhập danh sách khi người dùng nhập  
xâu rỗng (Ấn enter)

Dùng mảng các con trỏ **ptr** trỏ đến từng  
thành phần của mảng xâu **ds**

Hoán đổi vị trí các con trỏ ptr sao cho:

**ptr[0]** trỏ đến sinh viên có tên đứng thứ tự  
đầu tiên; và thành phần cuối cùng **ptr[count-1]**  
trỏ đến sinh viên có tên đứng thứ tự cuối cùng  
theo thứ tự ABC

# Con trỏ trỏ tới con trỏ ~ Mảng nhiều chiều

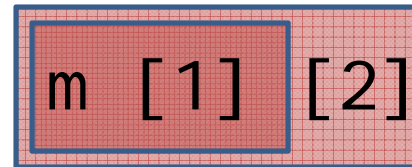
```
int m[2][4];
```

$*(m+1)$ : địa chỉ phần tử  $m[1][0]$

$** (m+1)$ : giá trị phần tử  $m[1][0]$

$*(m+1)+2$ : địa chỉ phần tử  $m[1][2]$

$*(*(m+1)+2)$ : giá trị phần tử  $m[1][2]$



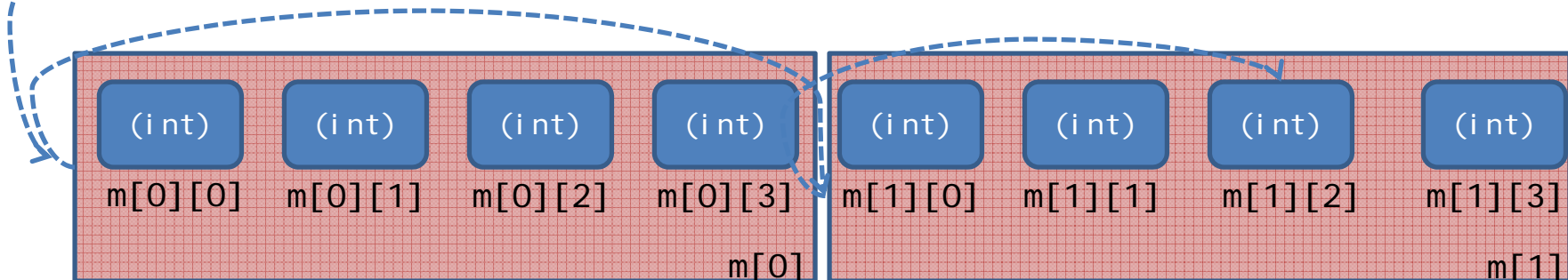
$*(*(m+1)+2)$

+2

Sau đó tăng 1 lượng bằng kích thước 2 phần tử kiểu `int`

+1

Tăng 1 lượng bằng kích thước mảng gồm 4 phần tử kiểu `int`



Con trở trở tới con trở

Mô tả 1 mảng 2 chiều qua con trỏ của con trỏ theo công thức :

$$m[i][j] = *(* (m+i) + j)$$

với

\* $(m+i)+j$  là địa chỉ phần tử  $[i][j]$

Ví dụ: Cho ma trận  $\begin{pmatrix} 7 & 8 & 9 \\ 10 & 13 & 15 \\ 2 & 7 & 8 \end{pmatrix}$ . Viết lệnh cộng mỗi phần tử của ma trận với 5 bằng cách dùng con trỏ của con trỏ:

```
#include <stdio.h>
#define hang 3
#define cot 3

int main() {
    int m[hang][cot] = {{7,8,9},{10,13,15},{2,7,8}};
    for (int i=0; i<hang;i++)
        for (int j=0;j<cot;j++)
            *(&m[i][j]+i*j) = *(m+i)+j +5;
}
```

# Nội dung

1. Mảng (Array)

**2. Bản ghi (Record)**

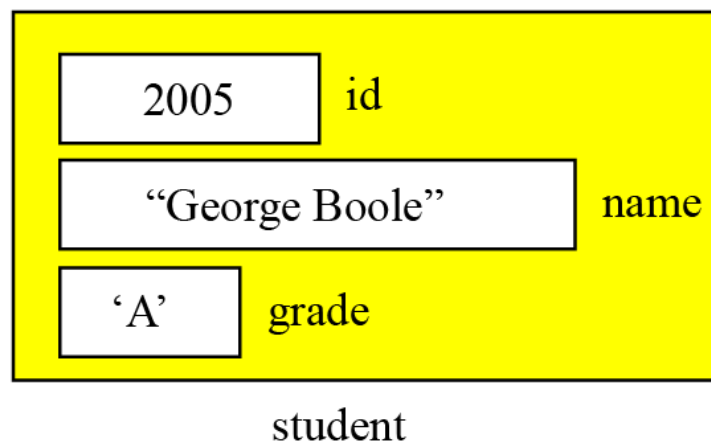
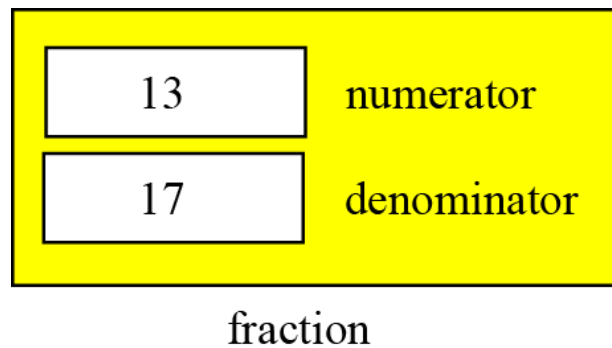
3. Danh sách liên kết (Linked List)

4. Ngăn xếp (Stack)

5. Hàng đợi (Queue)

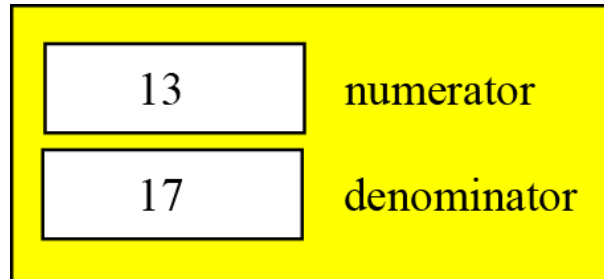
## 2. Bản ghi (record)

- Bản ghi (record) được tạo bởi một họ các trường (field) có thể có kiểu rất khác nhau
- Ví dụ:
  - Ví dụ 1: bản ghi **fraction** (phân số) gồm có 2 trường: **numerator** (tử số) và **denominator** (mẫu số), cả 2 trường này đều có kiểu **int** (số nguyên).
  - Ví dụ 2: bản ghi **student** gồm có 3 trường (**id**, **name**, **grade**), mỗi trường có 1 kiểu dữ liệu riêng:
    - Trường **id**: kiểu **int** (số nguyên)
    - Trường **name**: kiểu **string** (xâu kí tự)
    - Trường **grade**: kiểu **char** (kí tự)

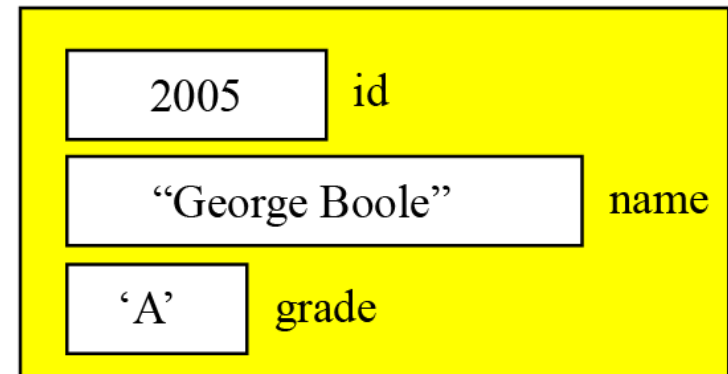


## 2. Bản ghi (Record)

- Ví dụ:



fraction



student

```
struct {  
    int numerator;  
    int denominator;  
} fraction;  
  
fraction.numerator = 13;  
fraction.denominator = 17;
```

```
struct {  
    int id;  
    char* name;  
    char grade;  
} student;  
  
student.id = 2005;  
student.name = "George Boole";  
student.grade = 'A';
```



# Tên của bản ghi vs. Tên của trường

Cũng giống như mảng (array), mỗi bản ghi có 2 định danh:

- Tên của bản ghi, và
- Tên của mỗi trường trong bản ghi.

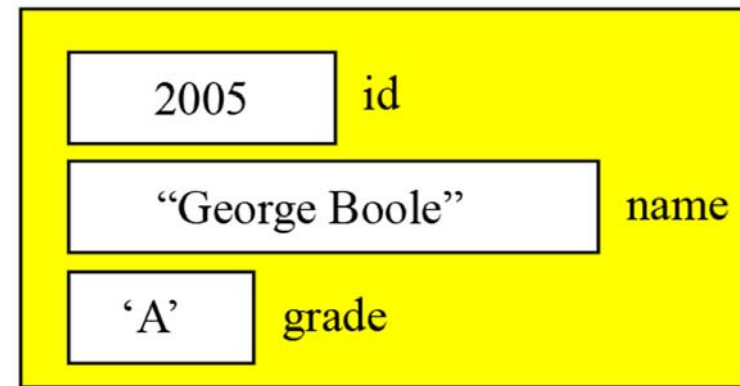
Tên của bản ghi là tên của cấu trúc bản ghi, tên của trường cho phép ta định danh tới trường thuộc bản ghi.

Ví dụ: bản ghi `student`:

- Tên của bản ghi là `student`,
- Tên của mỗi trường trong bản ghi `student` là `student.id`, `student.name` và `student.grade`.

Hầu hết các ngôn ngữ lập trình đều dùng dấu chấm (.) để phân tách tên của cấu trúc bản ghi với tên của các thành phần thuộc vào bản ghi (fields).

```
struct {  
    int id;  
    char* name;  
    char grade;  
} student;  
  
student.id = 2005;  
student.name = "George Boole";  
student.grade = 'A';
```

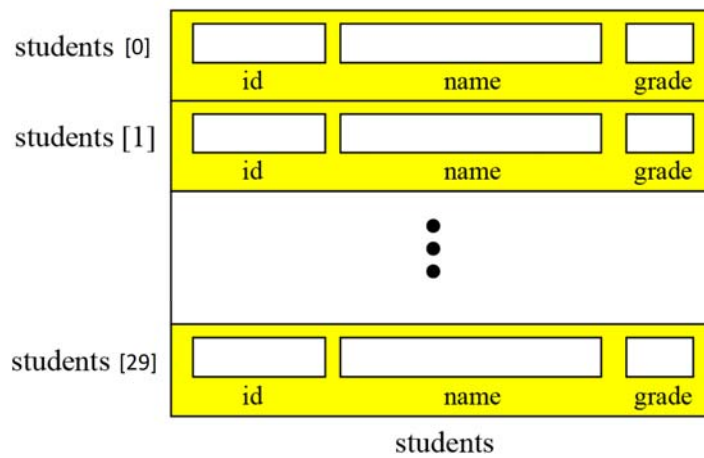


student

# So sánh bản ghi với mảng

Việc so sánh này giúp chúng ta biết khi nào nên dùng bản ghi, khi nào nên dùng mảng:

- Mảng là một tập các phần tử, còn bản ghi là tập các thành phần của 1 phần tử.
- Ví dụ: một mảng lưu trữ tập gồm 30 sinh viên của 1 lớp học, còn 1 bản ghi lưu trữ các thuộc tính khác nhau của 1 sinh viên (ví dụ các thuộc tính của sinh viên gồm: mã số sinh viên (id), tên của sinh viên, điểm của sinh viên...).
- Mảng các bản ghi: Nếu chúng ta cần lưu trữ tập các phần tử và đồng thời các thuộc tính của mỗi phần tử, khi đó ta sẽ phải dùng 1 mảng gồm các bản ghi. Ví dụ, 1 lớp gồm 30 sinh viên, ta sẽ khai báo 1 mảng gồm 30 bản ghi, mỗi bản ghi sẽ lưu trữ thông tin của 1 sinh viên trong lớp.



Hình1. Mảng các bản ghi

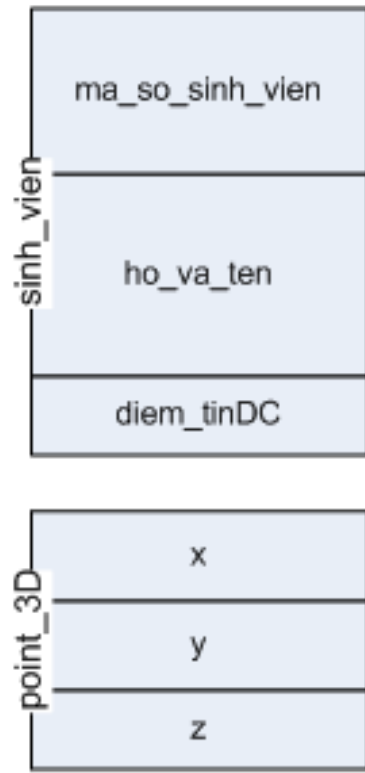
```
struct {  
    int id;  
    char* name;  
    char grade;  
} students[30];
```

```
students[0].id = 1001;  
students[0].name = "J.Aron";  
students[0].grade = 'A';  
  
students[1].id = 1002;  
students[1].name = "F.Bush";  
students[1].grade = 'F';  
...  
students[29].id = 3021;  
students[29].name = "M Blair";  
students[29].grade = 'B';
```

# Khai báo kiểu dữ liệu bản ghi

- Khai báo kiểu bản ghi

```
struct tên_bản_ghi{  
    <khai báo các trường >  
}
```



- Ví dụ

```
struct sinh_vien{  
    char ma_so_sinh_vien[10];  
    char ho_va_ten[30];  
    float diem_tinDC;  
}
```

```
struct point_3D{  
    float x;  
    float y;  
    float z;  
}
```

# Khai báo biến bản ghi

- Cú pháp:

```
struct <tên_cấu_trúc> <tên_biến_cấu_trúc>;
```

- Ví dụ:

```
struct sinh_vien a, b, c;
```

- Kết hợp khai báo

```
struct [tên_cấu_trúc] {  
    <khai báo các trường dữ liệu>;  
} tên_biến_cấu_trúc;
```

```
struct sinh_vien{  
    char ma_so_sinh_vien[10];  
    char ho_va_ten[30];  
    float diem_tinDC;  
}sv1,sv2;
```

```
struct {  
    char ma_so_sinh_vien[10];  
    char ho_va_ten[30];  
    float diem_tinDC;  
}sv1,sv2;
```

# Khai báo biến cấu trúc

- Các cấu trúc có thể được khai báo lồng nhau:

```
struct diem_thi {  
    float dToan, dLy, dHoa;  
}  
  
struct thi_sinh{  
    char SBD[10];  
    char ho_va_ten[30];  
    struct diem_thi ket_qua;  
} thi_sinh_1, thi_sinh_2;
```

- Có thể khai báo trực tiếp các trường dữ liệu của một cấu trúc bên trong cấu trúc khác:

```
struct thi_sinh{  
    char SBD[10];  
    char ho_va_ten[30];  
    struct diem_thi{  
        float dToan, dLy, dHoa;  
    } ket_qua;  
} thi_sinh_1, thi_sinh_2;
```

# Khai báo biến cấu trúc

- Có thể gán giá trị khởi đầu cho một biến cấu trúc, theo nguyên tắc như kiểu mảng:

Ví dụ:

```
struct Date{
    int day;
    int month;
    int year;
};
struct{
    char Ten[20];
    struct Date NS;
} SV = {"Tran Anh", 20, 12, 1990 };
```

```
struct{
    char Ten[20];
    struct Date{
        int day;
        int month;
        int year;
    } NS;
} SV = {"Tran Anh", 20, 12, 1990 };
```

# Định nghĩa kiểu dữ liệu với typedef

- Mục đích
  - Đặt tên mới cho kiểu dữ liệu cấu trúc
  - Giúp khai báo biến “quen thuộc” và ít sai hơn
- Cú pháp

**typedef** <tên\_cũ> <tên\_mới>;

- Ví dụ

```
typedef char message[80];  
message str="xin chao cac ban";
```

# Định nghĩa kiểu dữ liệu với typedef

- Với kiểu bản ghi

```
typedef struct tên_cũ tên_mới
```

```
typedef struct [tên_cũ] {  
    <khai báo các trường >;  
} danh_sách_các_tên_mới;
```

- Chú ý: cho phép đặt tên\_mới trùng tên\_cũ

Ví dụ:

```
struct point_3D{  
    float x, y, z;  
}  
struct point_3D M;  
typedef struct point_3D toa_do_3_chieu;  
toa_do_3_chieu N;
```

```
typedef struct {  
    float x, y, z;  
}point_3D;  
point_3D M;  
point_3D N;
```



# Ví dụ

```
typedef struct tên_cũ {  
    <khai báo các trường >;  
} danh_sách_các_tên_mới;
```

Ví dụ:

```
typedef struct point_2D {  
    float x, y;  
} point_2D, diem_2_chieu, ten_bat_ki;  
point_2D X;  
diem_2_chieu Y;  
ten_bat_ki Z;
```

=> point\_2D, diem\_2\_chieu, ten\_bat\_ki là các tên bản ghi,  
không phải tên biến

# Xử lý dữ liệu bản ghi

- Truy cập các trường dữ liệu
- Phép gán giữa các biến bản ghi

# Truy cập các trường dữ liệu

- Cú pháp

*<tên\_biến\_bản\_ghi>.<tên\_trường>*

- Lưu ý
  - Dấu “.” là toán tử truy cập vào trường dữ liệu trong bản ghi
  - Nếu trường dữ liệu là một bản ghi => sử dụng tiếp dấu “.” để truy cập vào thành phần mức sâu hơn

# Ví dụ

```
#include <stdio.h>
void main() {
    struct {
        char Ten[20];
        struct Date {
            int day;
            int month;
            int year;
        } NS;
    } SV = {"Tran Anh", 20, 12, 1990 };

    printf(" Sinh vien %s (%d/%d/%d)",
SV.Ten, SV.NS.day, SV.NS.month, SV.NS.year) ;
}
```

**Sinh vien Tran Anh (20/12/1990)**

# Phép gán giữa các biến bản ghi

- Muốn sao chép dữ liệu từ biến bản ghi này sang biến bản ghi khác cùng kiểu
  - gán lần lượt từng trường trong hai biến bản ghi → “thủ công”
  - C cung cấp phép gán hai biến bản ghi cùng kiểu:  
 $\text{biến\_cấu\_trúc\_1} = \text{biến\_cấu\_trúc\_2};$

# Phép gán giữa các biến bản ghi

Ví dụ:

- Xây dựng bản ghi gồm họ tên và điểm môn lập trình C của sinh viên
- a, b, c là 3 biến cấu trúc.
- Nhập giá trị cho biến a.
- Gán b=a,
- Gán từng trường của a cho c.
- So sánh a, b và c ?

```
#include<stdio.h>
typedef struct{
    char hoten[20];
    int diem;
}sinhvien;
void main(){
    sinhvien a,b,c;
    printf("Nhap thong tin sinh vien\n");
    printf("Ho ten: ");gets(a.hoten);
    printf("Diem:");scanf("%d",&a.diem);
    b=a;
    strcpy(c.hoten,a.hoten);
    c.diem=a.diem;
    printf("Bien a: %-20s%3d\n",a.hoten,a.diem);
    printf("Bien b: %-20s%3d\n",b.hoten,b.diem);
    printf("Bien c: %-20s%3d\n",c.hoten,c.diem);
}
```

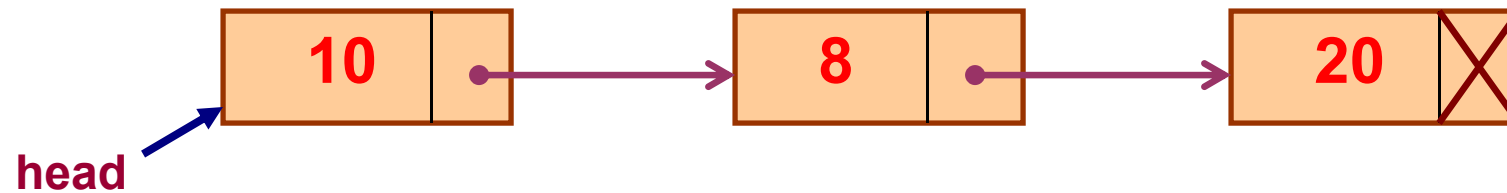
```
Nhap thong tin sinh vien
Ho ten: Nguyen Van Anh
Diem:9
Bien a: Nguyen Van Anh    9
Bien b: Nguyen Van Anh    9
Bien c: Nguyen Van Anh    9
```

# Nội dung

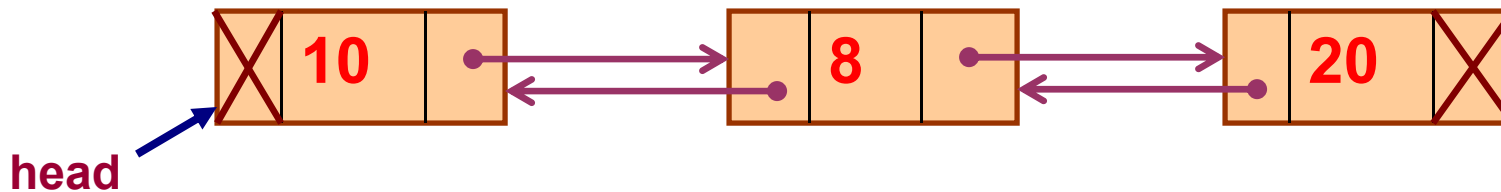
1. Mảng (Array)
2. Bản ghi (Record)
- 3. Danh sách liên kết (Linked List)**
4. Ngăn xếp (Stack)
5. Hàng đợi (Queue)

### 3. Danh sách liên kết/móc nối (Linked list)

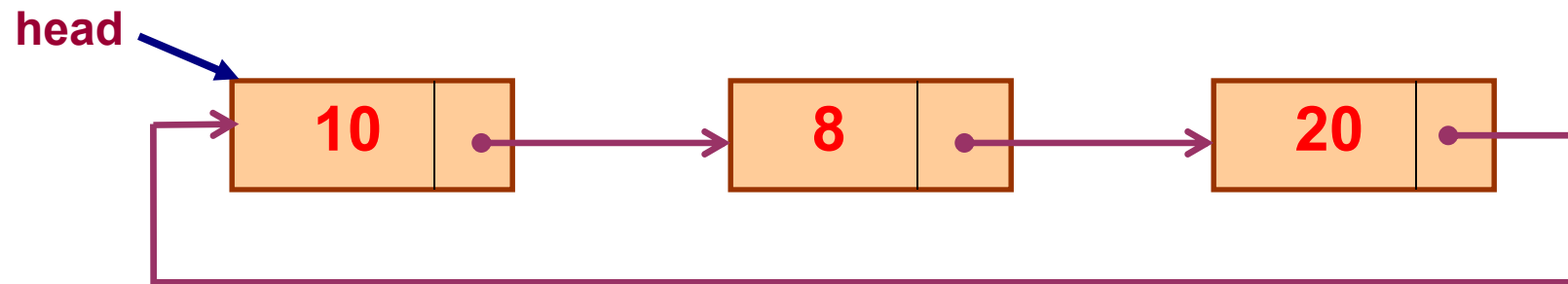
- Danh sách liên kết đơn (Singly linked list)



- Danh sách liên kết đôi (Doubly linked list)



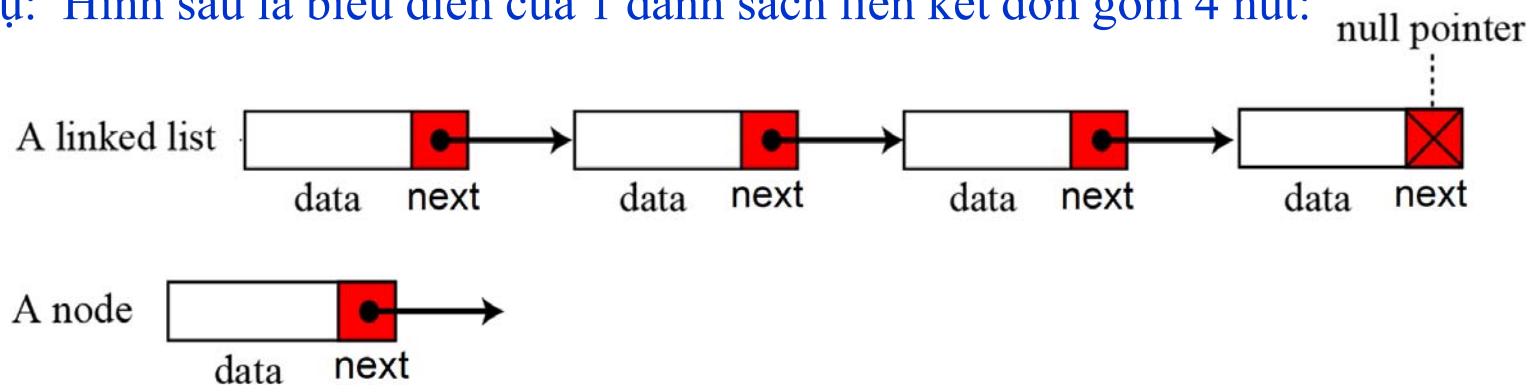
- Danh sách liên kết vòng (Circular linked list)



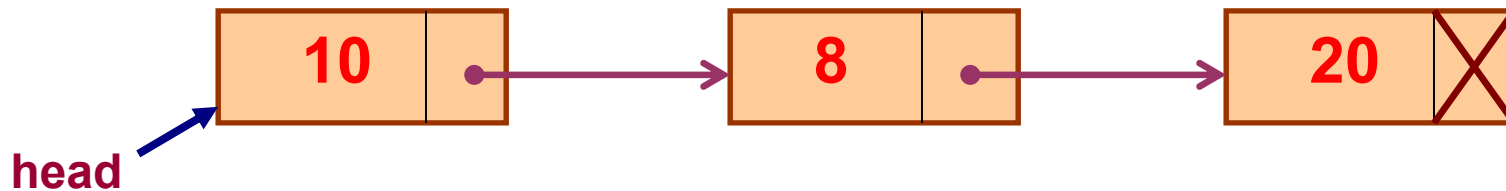


# Danh sách liên kết đơn (Singly Linked list)

- Danh sách liên kết đơn gồm một dãy các nút (node), mỗi nút gồm 2 phần: **dữ liệu (data)** và **con trỏ** trỏ tới nút liền kề tiếp trong danh sách (con trỏ này chứa địa chỉ của nút tiếp theo).
- Ví dụ: Hình sau là biểu diễn của 1 danh sách liên kết đơn gồm 4 nút:

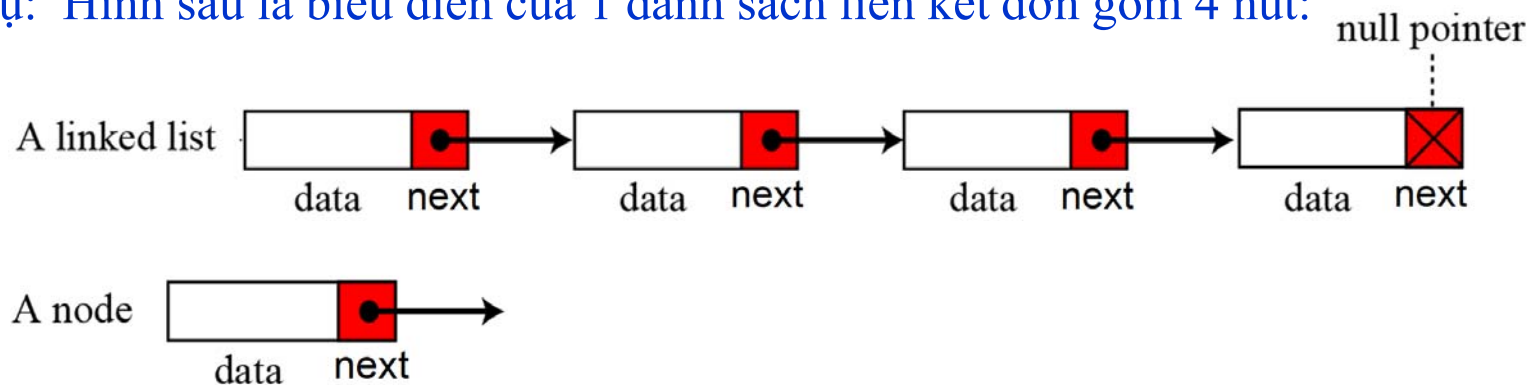


- Để định danh được danh sách liên kết đơn:
  - Ta cần biết con trỏ trỏ đến phần tử đầu tiên của danh sách (cần biết địa chỉ của nút đầu tiên) (thường con trỏ này được kí hiệu là *start* hoặc *head*)
  - Nếu con trỏ head = NULL, danh sách liên kết đơn rỗng (không có nút nào)



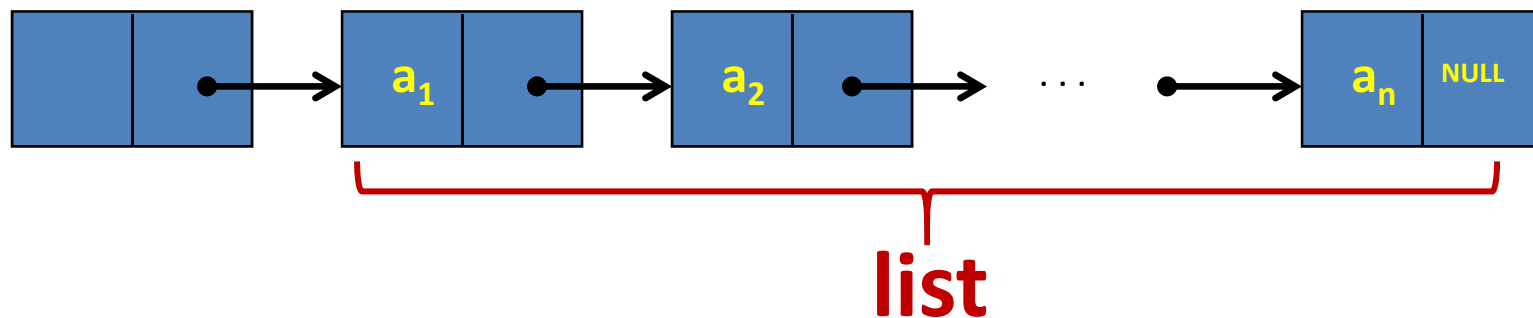
# Danh sách liên kết đơn (Singly Linked list)

- Danh sách liên kết đơn gồm một dãy các nút (node), mỗi nút gồm 2 phần: **dữ liệu (data)** và **con trỏ** trỏ tới nút liền kề tiếp trong danh sách (con trỏ này chứa địa chỉ của nút tiếp theo).
- Ví dụ: Hình sau là biểu diễn của 1 danh sách liên kết đơn gồm 4 nút:



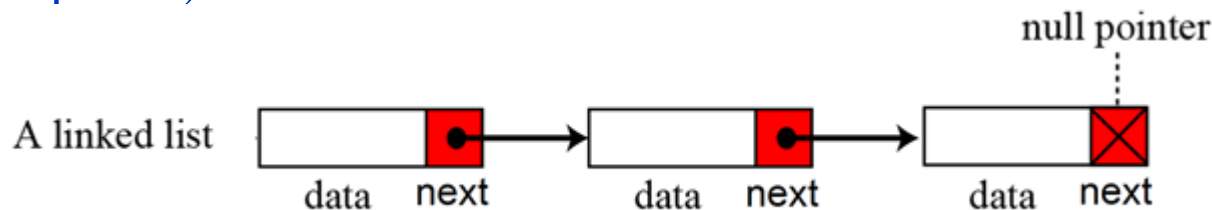
- Để định danh được danh sách liên kết đơn:
  - Ta cần biết con trỏ trỏ đến phần tử đầu tiên của danh sách (cần biết địa chỉ của nút đầu tiên) (thường con trỏ này được kí hiệu là *start* hoặc *head*)
  - Nếu con trỏ *head* = NULL, danh sách liên kết đơn rỗng (không có nút nào)

**head**

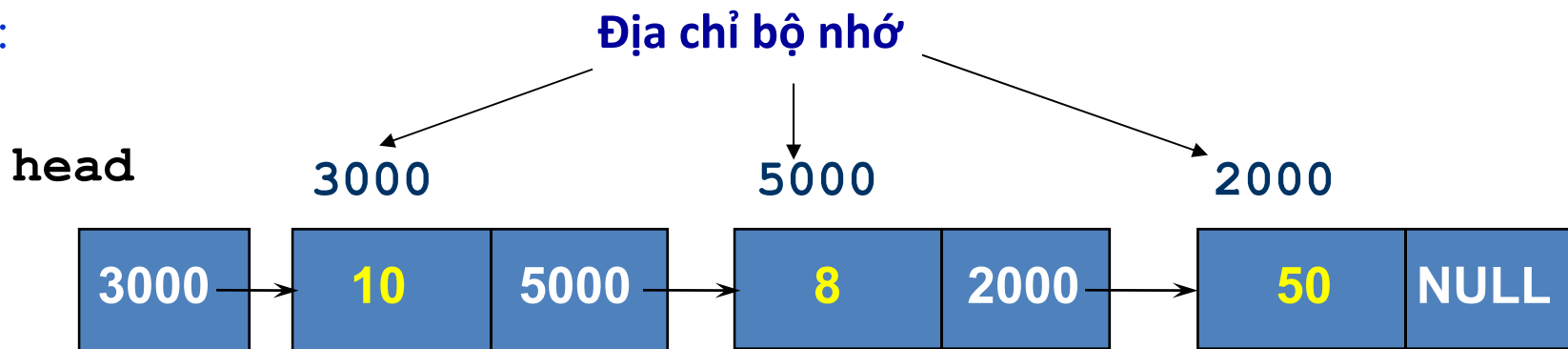


# Danh sách liên kết đơn (Singly Linked list)

- Danh sách liên kết đơn gồm một dãy các nút (node), mỗi nút gồm 2 phần: **dữ liệu (data)** và **con trỏ** trỏ tới nút liên kế tiếp trong danh sách (con trỏ này chứa địa chỉ của nút tiếp theo).



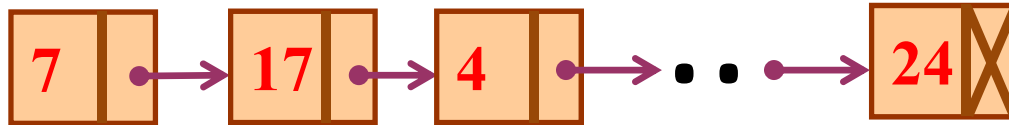
- Hình vẽ trên biểu diễn 1 danh sách gồm 3 nút. Mỗi nối giữa 2 nút liên tiếp được biểu diễn bởi 1 đường thẳng: một đầu có mũi tên, đầu còn lại là một chấm tròn.
- Ví dụ: 3 số nguyên 10, 8, 50 được lưu trữ bởi danh sách liên kết đơn gồm 3 nút như sau:



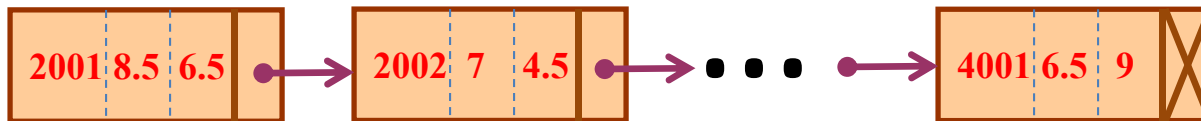
- Các phần tử thuộc danh sách liên kết có thể nằm ở vị trí bất kỳ trong bộ nhớ  
(? So sánh với việc lưu trữ trong bộ nhớ của các phần tử thuộc cùng 1 mảng đã học ở phần trước)

# Cách khai báo danh sách liên kết đơn trong ngôn ngữ C

- Danh sách các số nguyên :



- Danh sách các sinh viên gồm các dữ liệu: mã sinh viên, điểm 2 môn toán và vật lý



- Danh sách các contact trong danh bạ điện thoại gồm các dữ liệu: tên, số điện thoại



## ➔ Cách khai báo 1 danh sách liên kết đơn:

- Đầu tiên cần khai báo kiểu của các dữ liệu lưu trữ trong 1 nút,
- Sau đó, khai báo danh sách liên kết đơn gồm (1) dữ liệu của nút, và (2) con trỏ lưu trữ địa chỉ của nút tiếp theo

## Khai báo danh sách liên kết đơn

→ Cách khai báo 1 danh sách liên kết đơn:

- Đầu tiên cần khai báo kiểu của các dữ liệu lưu trữ trong 1 nút,
- Sau đó, khai báo danh sách liên kết đơn gồm (1) dữ liệu của nút, và (2) con trỏ lưu trữ địa chỉ của nút tiếp theo

```
typedef struct {  
    .....  
}NodeType;
```

Khai báo kiểu của các dữ liệu trong 1 nút

```
typedef struct node {  
    NodeType data;  
    struct node* next;  
}node;  
node* head;
```

Khai báo danh sách liên kết đơn

```
struct node{  
    NodeType element;  
    struct node* next;  
};  
struct node* head;
```

Khai báo trên định nghĩa biến `node` là 1 bản ghi gồm có 2 trường (field):

- `data` : lưu trữ dữ liệu của nút, có kiểu `NodeType` (được định nghĩa trong `typedef...NodeType`, và có thể chứa nhiều thuộc tính)
- `next` : con trỏ lưu trữ địa chỉ của nút tiếp theo trong danh sách

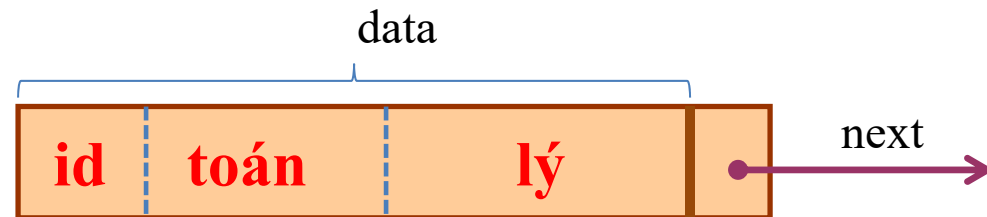
Con trỏ `head` : lưu trữ địa chỉ của nút đầu tiên của danh sách

Ví dụ 1: Danh sách sinh viên gồm các dữ liệu: mã sinh viên (id), điểm 2 môn học: toán, lý

```
typedef struct{  
    char id[15];  
    float toán, lý;  
}student;
```

Khai báo kiểu của các dữ liệu trong 1 nút

```
typedef struct node {  
    student data;  
    struct node* next;  
}node;  
node* head;
```



1 Nút

# Khai báo danh sách liên kết đơn

```
typedef struct {  
    .....  
}NodeType;
```

Khai báo kiểu của các dữ liệu trong 1 nút

```
typedef struct node {  
    NodeType data;  
    struct node* next;  
}node;  
node* head;
```

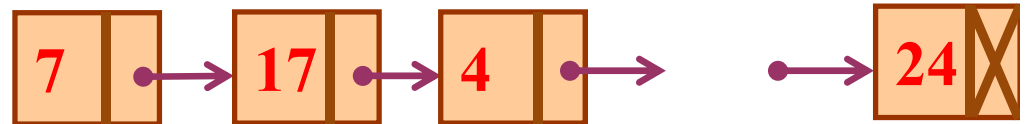
Khai báo danh sách liên kết đơn

Khai báo trên định nghĩa biến `node` là 1 bản ghi gồm có 2 trường (field):

- `data` : lưu trữ dữ liệu của nút, có kiểu `NodeType` (được định nghĩa trong `typedef...NodeType`, và có thể chứa nhiều thuộc tính)
- `next` : con trỏ lưu trữ địa chỉ của nút tiếp theo trong danh sách

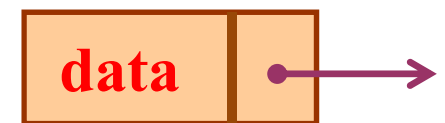
Con trỏ `head` : lưu trữ địa chỉ của nút đầu tiên của danh sách

Ví dụ 2: Danh sách chứa các số nguyên (int)



```
typedef struct node{  
    int data;  
    struct node* next;  
}node;  
node* head;
```

“`int`” là kiểu dữ liệu của nút, nên trong trường hợp này ta không cần sử dụng cấu trúc “`typedef...NodeType`” để khai báo kiểu dữ liệu của nút



1 nút

## Khai báo danh sách liên kết đơn

→ Cách khai báo 1 danh sách liên kết đơn:

- Đầu tiên cần khai báo kiểu của các dữ liệu lưu trữ trong 1 nút,
- Sau đó, khai báo danh sách liên kết đơn gồm (1) dữ liệu của nút, và (2) con trỏ lưu trữ địa chỉ của nút tiếp theo

```
typedef struct {  
    .....  
}NodeType;  
  
typedef struct node {  
    NodeType data;  
    struct node* next;  
}node;  
node* head;
```

Khai báo kiểu của các dữ liệu trong 1 nút

Khai báo danh sách liên kết đơn

Khai báo trên định nghĩa biến `node` là 1 bản ghi gồm có 2 trường (field):

- `data` : lưu trữ dữ liệu của nút, có kiểu `NodeType` (được định nghĩa trong `typedef...NodeType`, và có thể chứa nhiều thuộc tính)
- `next` : con trỏ lưu trữ địa chỉ của nút tiếp theo trong danh sách

Con trỏ `head` : lưu trữ địa chỉ của nút đầu tiên của danh sách

Ví dụ 3: Danh sách lưu trữ danh bạ điện thoại gồm các dữ liệu: tên và số điện thoại

```
typedef struct{  
    char name[15];  
    char phone[20];  
}contact;
```

Định nghĩa kiểu dữ liệu của nút

```
typedef struct node{  
    contact data;  
    struct node* next;  
}node;  
node* head;
```

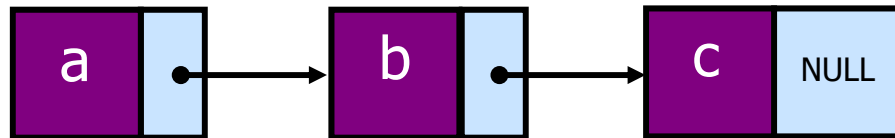


1 nút

# Danh sách liên kết đơn – Ví dụ

```
typedef struct list {  
    char data;  
    struct list *next;  
}list;  
list node1, node2, node3;
```

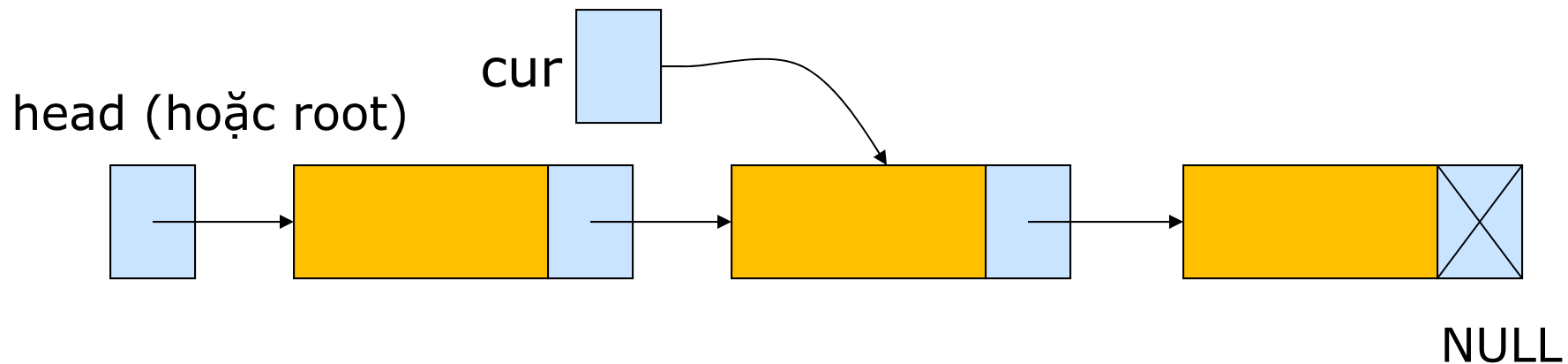
```
node1.data='a';  
node2.data='b';  
node3.data='c';  
node1.next=node2;  
node2.next=node3;  
node3.next=NULL;
```





# Các thành phần quan trọng trong 1 danh sách liên kết đơn

- head: biến con trỏ lưu trữ địa chỉ của nút đầu tiên trong danh sách liên kết
- cur: biến con trỏ lưu trữ địa chỉ của nút hiện tại
- NULL: giá trị của trường con trỏ của nút cuối cùng



```
typedef struct{
    char *ma;
    struct{
        float toan, ly, hoa, tong;
    } DT;
}thisinh;
```

```
typedef struct node{
    thisinh data;
    struct node* next;
}node;
node* head;
node* cur;
```

# Cấp phát bộ nhớ cho 1 nút

- Khai báo con trỏ newNode

```
node *newNode;      (1)
```

- Phân bổ bộ nhớ cho 1 nút mới trỏ bởi biến con trỏ newNode trong danh sách:

```
newNode = (node *) malloc(sizeof(node));    (2)
```

hoặc có thể gộp lệnh (1) và (2) thành một lệnh duy nhất:

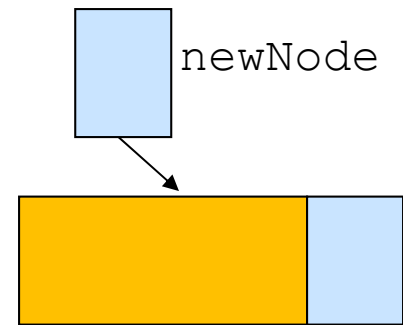
```
node *newNode = (node *) malloc(sizeof(node));
```

- Truy cập dữ liệu (data) của nút trỏ bởi con trỏ newNode :

```
newNode->data hoặc (*newNode).data
```

- Giải phóng bộ nhớ phân bổ cho nút trỏ bởi con trỏ newNode:

```
free (newNode) ;
```



```
typedef struct{  
    char *ma;  
    struct{  
        float toan, ly, hoa, tong;  
    } DT;  
}thisinh;  
typedef struct node{  
    thisinh data;  
    struct node* next;  
};  
node* head;  
node* cur;
```

```
node *newNode;  
newNode = (node*) malloc(sizeof(node));  
newNode->data.DT.ma = "12345";  
newNode->data.DT.toan = 4.5;  
newNode->data.DT.ly = 7.0;  
(*newNode).data.DT.hoa = 8.5;  
newNode->next = NULL;
```

# Khai báo structure không dùng typedef

```
typedef struct{
    char ma[15];
    struct{
        float toan, ly, hoa, tong;
    } DT;
}thisinh;
typedef struct node{
    thisinh data;
    struct node* next;
};
node* head;
node* cur;
```



```
typedef struct{
    char ma[15];
    struct{
        float toan, ly, hoa, tong;
    } DT;
}thisinh;
struct node{
    thisinh data;
    struct node* next;
};
struct node* head;
struct node* cur;
```

```
node *newNode;
newNode = (node*) malloc(sizeof(node));
newNode->data.DT.ma = "12345";
newNode->data.DT.toan = 4.5;
newNode->data.DT.ly = 7.0;
newNode->data.DT.hoa = 8.5;
newNode->next = NULL;
```

```
struct node *newNode;
newNode = (struct node*) malloc(sizeof(struct node));
.....
```

# Bài tập

```
#include<stdio.h>
#include <stdlib.h>
typedef struct
{
    char * hoten;
    float diemthi;
    char grade;
}RECORD;

int main()
{
    RECORD *sv;
    sv = (RECORD *) malloc (sizeof(RECORD));
    (*sv).hoten = "Nguyen Phuong";
    (*sv).diemthi = 8.5;
    (*sv).grade = 'A';
    printf("Ho ten thi sinh: %s\n", (*sv).hoten);
    printf("Diem toan: %f\n", (*sv).diemthi);
    printf("Phan loai: %c\n", (*sv).grade);
    free(sv);
    return 0;
}
```

- Cho chương trình như hình bên.
1. Đưa ra nội dung trên màn hình khi chạy chương trình.
  2. Sửa lại chương trình:
    - a. Khai báo structure mà không dùng typedef
    - b. Truy cập dữ liệu của nút dùng qua con trỏ dạng:

**newNode->data**

# Các thao tác trên danh sách liên kết đơn

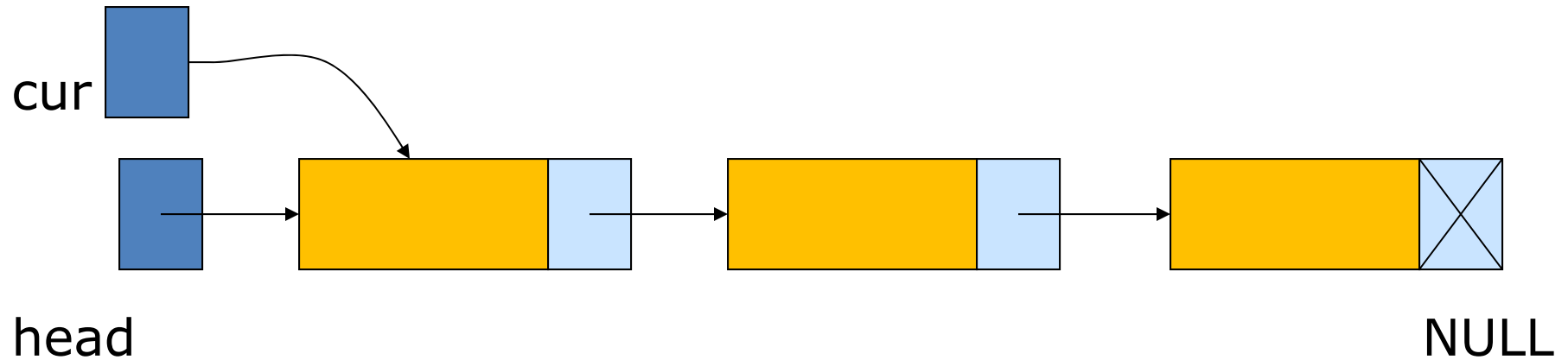
- Duyệt (Traverse) danh sách
- Thêm (Insert) 1 nút mới vào danh sách
- Xóa (Delete) 1 nút khỏi danh sách
- Tìm kiếm dữ liệu (Search) trong danh sách

# Các thao tác trên danh sách liên kết đơn

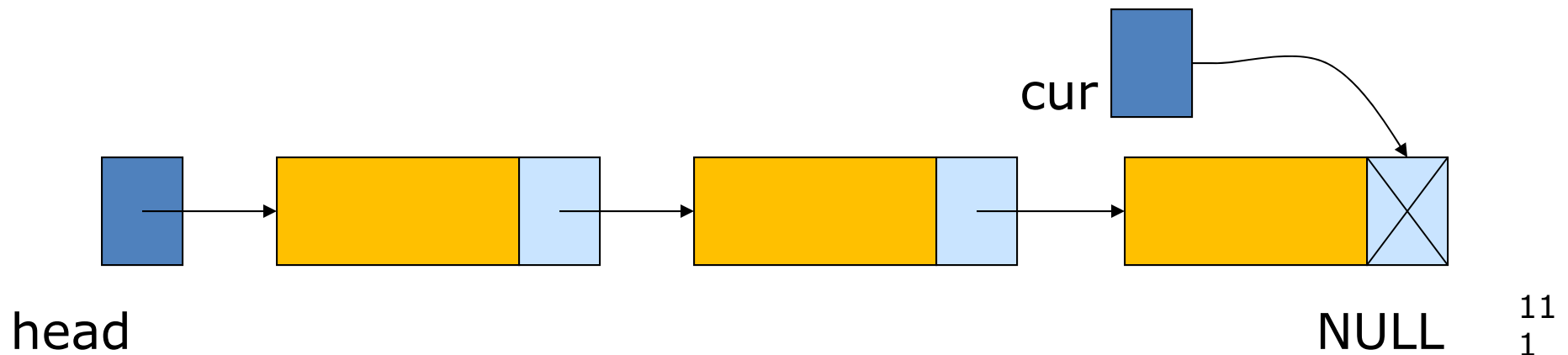
- **Duyệt (Traverse) danh sách**
- Thêm (Insert) 1 nút mới vào danh sách
- Xóa (Delete) 1 nút khỏi danh sách
- Tìm kiếm dữ liệu (Search) trong danh sách

# Duyệt danh sách liên kết đơn

```
for ( cur = head; cur != NULL; cur = cur->next )  
    showData_Of_Current_Node( cur->data );
```



- Lần lượt thay đổi giá trị của biến con trỏ **cur**
- Thao tác duyệt kết thúc khi con trỏ **cur** có giá trị NULL



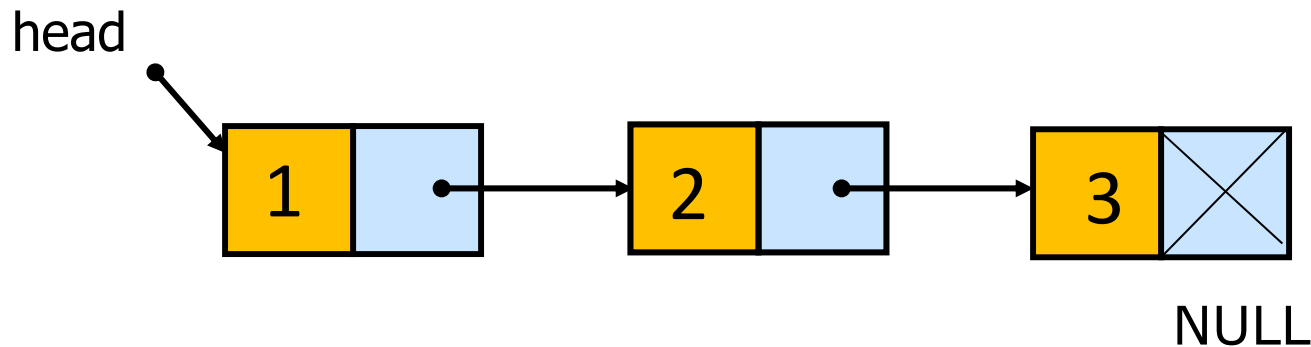
# Bài 1

- Một danh sách các số nguyên được lưu trữ bởi danh sách liên kết đơn.

```
typedef struct Node{  
    int data;  
    struct Node *next;  
}Node;  
Node *head;
```

```
struct Node {  
    int data;  
    struct Node *next;  
};  
struct Node *head;
```

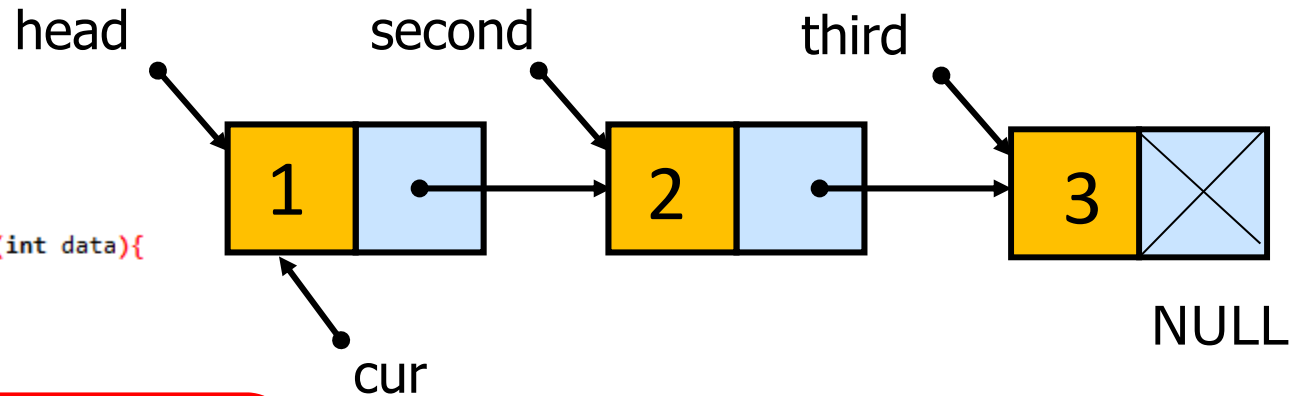
- Tạo danh sách lưu trữ 3 số nguyên: 1, 2, 3
- In các số nguyên có trong danh sách ra màn hình (Duyệt danh sách)





# Bài 1

```
1 #include<stdio.h>
2
3 typedef struct Node
4 {
5     int data;
6     struct Node *next;
7 }Node;
8
9 void showData_Of_Current_Element(int data){
10     printf("Data = %d\n",data);
11 }
12
13 int main()
14 {
15     struct Node* head;
16     Node* second = NULL;
17     Node* third = NULL;
18
19     // Cap phat bo nho cho 3 node trong vung nho heap
20     head = (Node*)malloc(sizeof(Node));
21     second = (Node*)malloc(sizeof(Node));
22     third = (Node*)malloc(sizeof(Node));
23
24     head->data = 1; //gan du lieu cho node dau tien trong danh sach
25     head->next = second; //noi node dau tien voi node thu 2
26
27     second->data = 2; //gan du lieu cho node thu 2
28     second->next = third; //noi node thu 2 voi node thu 3
29
30     third->data = 3; //gan du lieu cho node thu 3
31     third->next = NULL;
32
33     Node* cur;
34     for (cur = head; cur != NULL; cur = cur->next)
35         showData_Of_Current_Element(cur->data);
36     return 0;
37 }
38
```



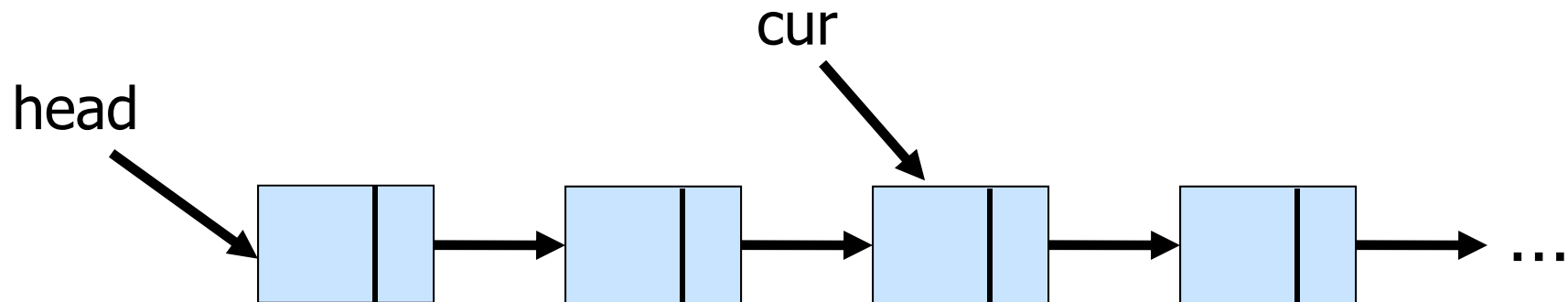
# Các thao tác trên danh sách liên kết đơn

- Duyệt (Traverse) danh sách
- **Thêm (Insert) 1 nút mới vào danh sách**
- Xóa (Delete) 1 nút khỏi danh sách
- Tìm kiếm dữ liệu (Search) trong danh sách

# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào danh sách:

- Thêm vào đầu danh sách
- Thêm vào ngay phía sau nút đang trỏ bởi con trỏ **cur**
- Thêm vào ngay phía trước nút đang trỏ bởi con trỏ **cur**
- Thêm vào cuối danh sách

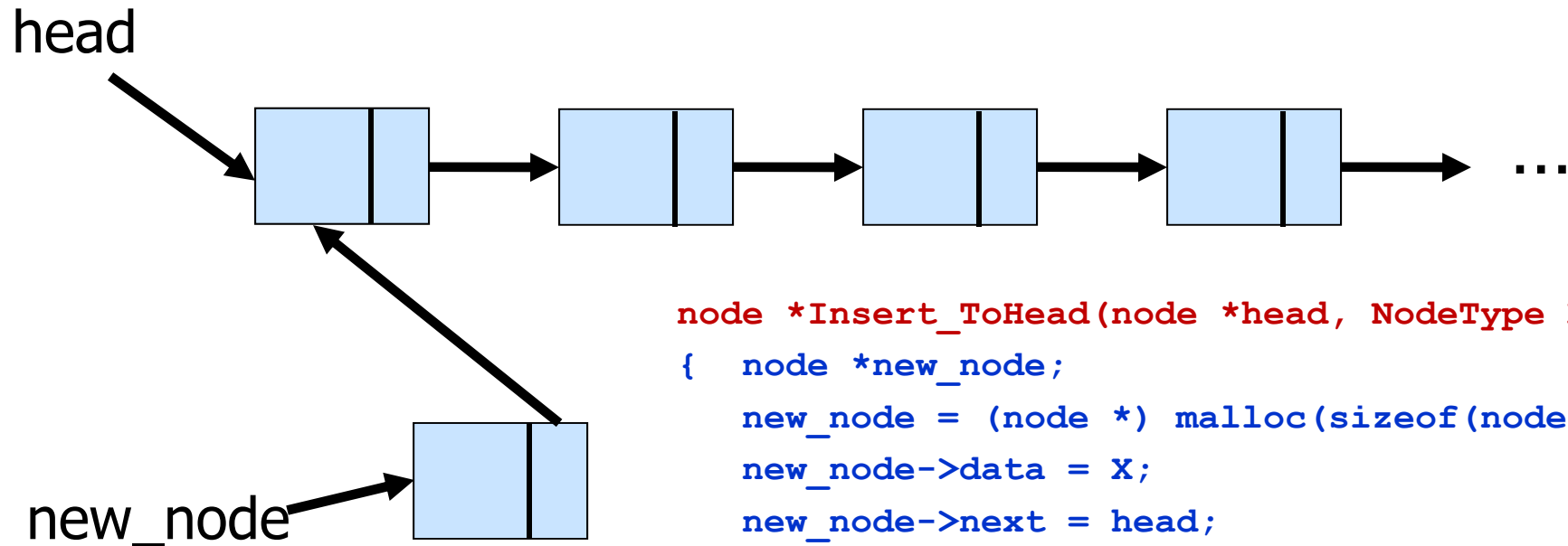


# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào danh sách:

- Thêm vào đầu danh sách

```
<tạo nút mới new_node>;  
new_node->next = head;  
head = new_node;
```

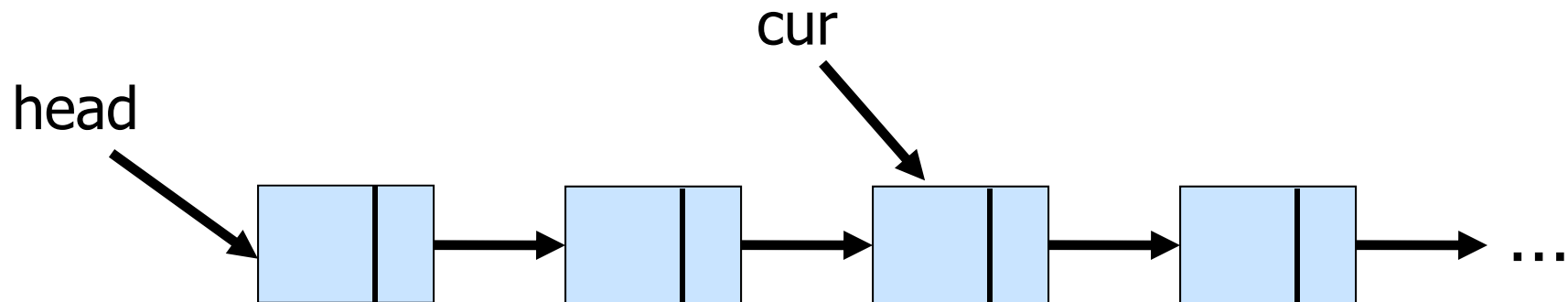


```
node *Insert_ToHead(node *head, NodeType X)  
{  
    node *new_node;  
    new_node = (node *) malloc(sizeof(node));  
    new_node->data = X;  
    new_node->next = head;  
    head = new_node;  
    return head;  
}
```

# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào danh sách:

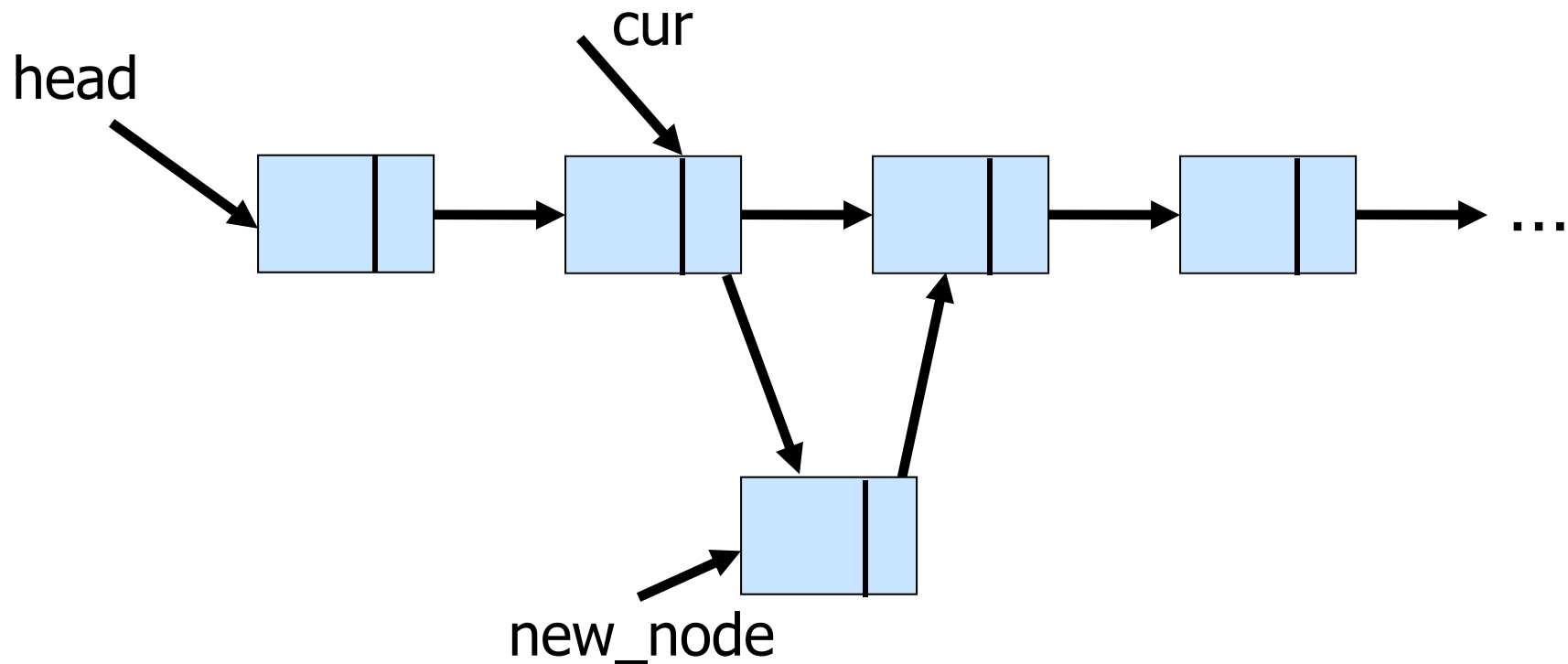
- Thêm vào đầu danh sách
- **Thêm vào ngay phía sau nút đang trỏ bởi con trỏ `cur`**
- Thêm vào ngay phía trước nút đang trỏ bởi con trỏ `cur`
- Thêm vào cuối danh sách



# Các thao tác trên danh sách liên kết đơn: THÊM

- Thêm 1 nút mới vào **ngay sau nút trỏ bởi con trỏ cur**:

```
< tạo nút mới new_node >;  
new_node ->next = cur->next;  
cur->next = new_node;
```



# Các thao tác trên danh sách liên kết đơn: THÊM

- Thêm 1 nút mới vào **ngay sau nút trở bởi con trỏ cur**:

```
< tạo nút mới new_node>;  
new_node ->next = cur->next;  
cur->next = new_node;
```

Viết hàm thêm 1 nút mới có trường dữ liệu data = X (có kiểu «NodeType») vào ngay sau nút trở bởi con trỏ cur. Hàm trả về địa chỉ của nút mới:

```
node *Insert_After(node *cur, NodeType X)  
{  
    node *new_node;  
    new_node = (node *) malloc(sizeof(node)); // (1)  
    new_node -> data = X;                      // (1)  
    new_node->next = cur->next;                 // (2)  
    cur->next = new_node;                      // (3)  
    return new_node;  
}
```

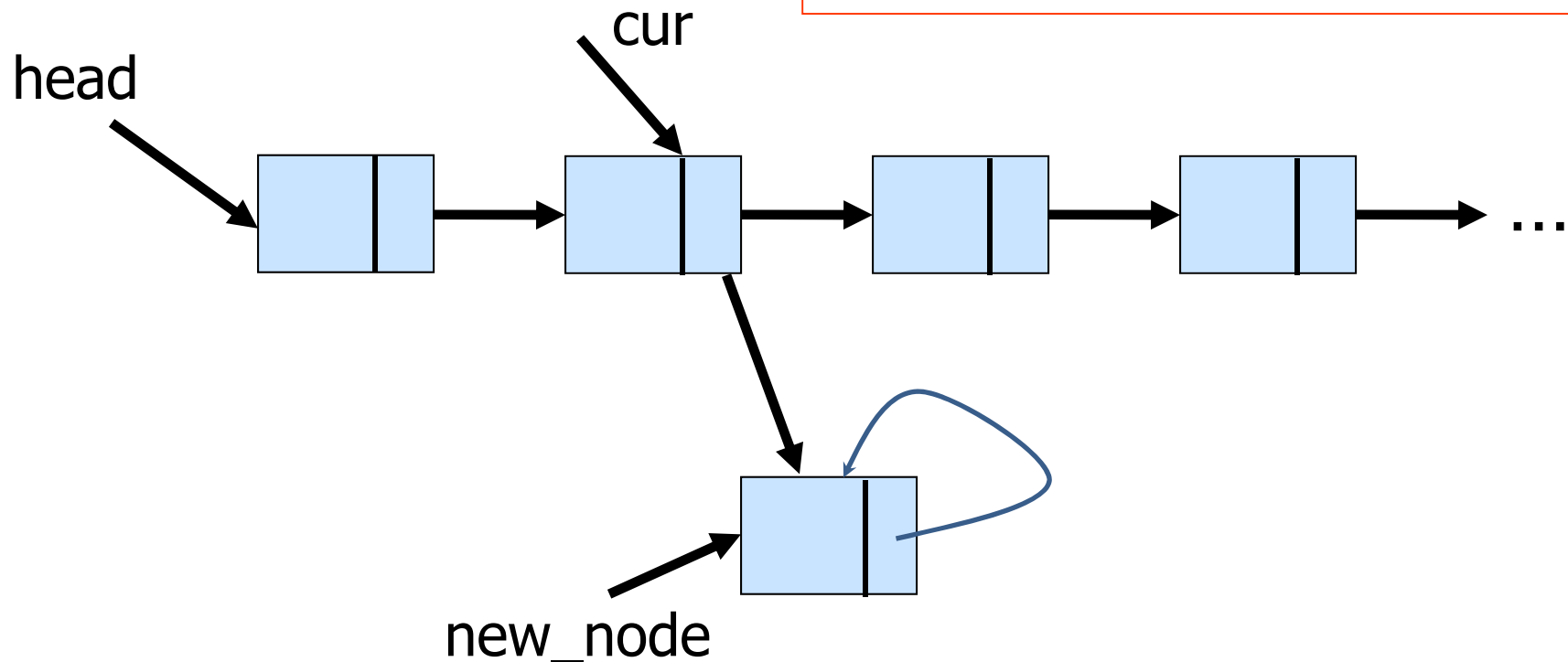
# Các thao tác trên danh sách liên kết đơn: THÊM

- Thêm 1 nút mới vào **ngay phía sau nút trỏ bởi con trỏ cur**:

```
<tạo nút mới new_node>;  
new_node ->next = cur->next;  
cur->next = new_node;
```

?? Danh sách rỗng

// cài đặt sai:  
cur->next = new\_node;  
new\_node ->next = cur->next;





# Các thao tác trên danh sách liên kết đơn: THÊM

- Thêm 1 nút mới vào ngay phía sau nút trỏ bởi con trỏ cur:

```
<tạo nút mới new_node>;  
new_node ->next = cur->next;  
cur->next = new_node;
```

?? Danh sách rỗng

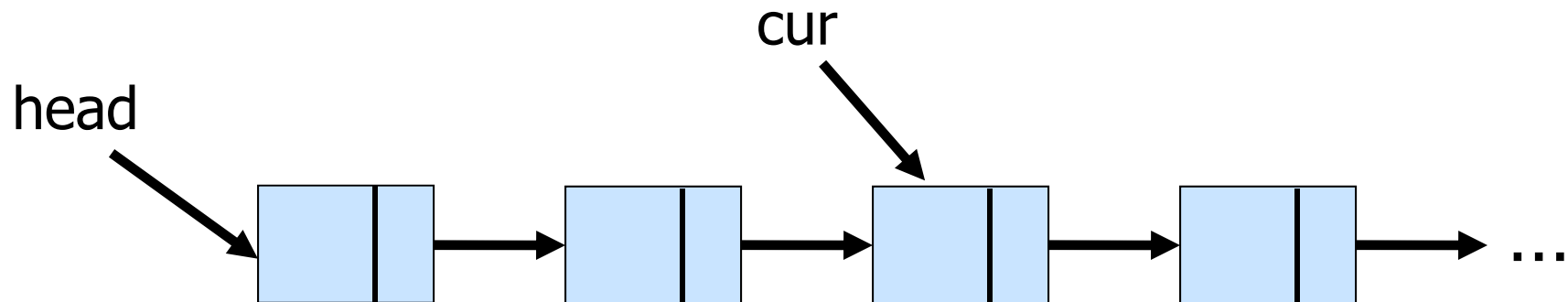


```
<tạo 1 nút mới new_node>;  
if (head == NULL) { /* danh sách đang không có nút nào */  
    head = new_node;  
    cur = head;  
}  
else {  
    new_node ->next = cur->next;  
    cur->next = new_node;  
}
```

# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào danh sách:

- Thêm vào đầu danh sách
- Thêm vào ngay phía sau nút đang trỏ bởi con trỏ `cur`
- **Thêm vào ngay phía trước nút đang trỏ bởi con trỏ `cur`**
- Thêm vào cuối danh sách



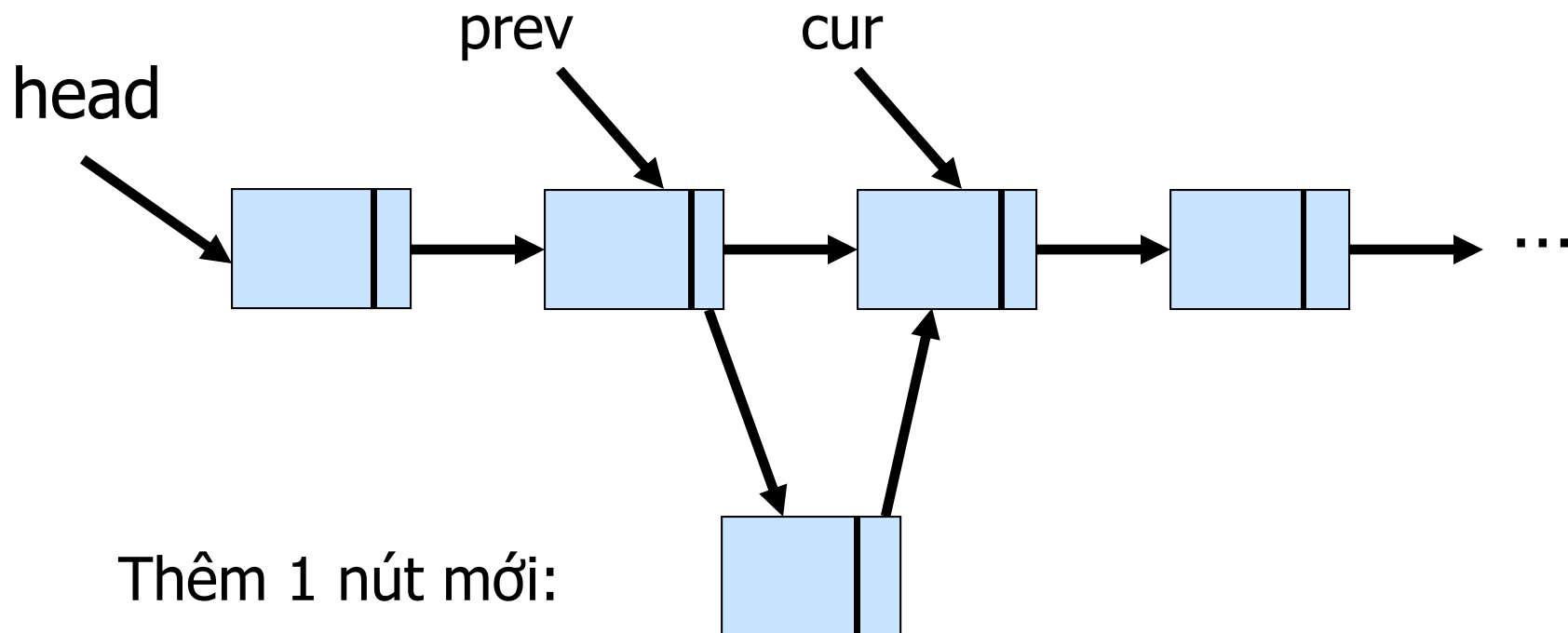
# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào **ngay trước nút trở bởi con trỏ cur**

```
<tạo nút mới new_node>;  
prev->next = new_node;  
new_node->next = cur;
```

?? Danh sách đang không có nút nào

?? cur trỏ đến nút đầu tiên trong danh sách



# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào **ngay trước nút trở bởi con trỏ cur**

```
<tạo nút mới new_node>;  
prev->next = new_node;  
new_node->next = cur;
```

?? Danh sách đang không có nút nào

?? cur trỏ đến nút đầu tiên trong danh sách



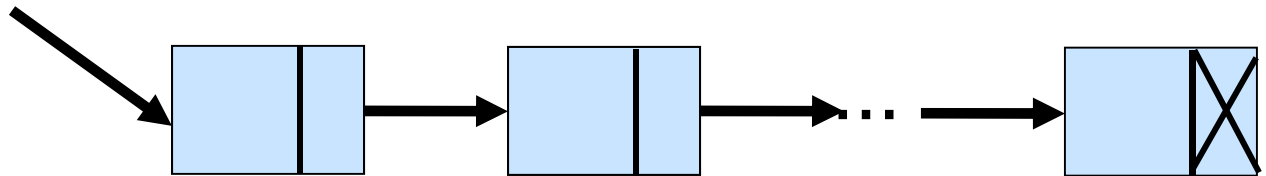
```
<tạo nút mới new_node>;  
if (head == NULL) { /* danh sách đang không có nút nào */  
    head = new_node;  
    cur = head;  
}  
else if (cur == head) { //cur trỏ đến nút đầu tiên trong ds  
    head = new_node;  
    new_node->next = cur;  
}  
else {  
    prev->next = new_node;  
    new_node->next = cur;  
}
```

# Các thao tác trên danh sách liên kết đơn: THÊM

Thêm 1 nút mới vào:

- Đầu tiên
- Sau nút trở bởi cur
- Trước nút trở bởi cur
- **Cuối danh sách**

head



```
<tạo 1 nút mới new_node>;
if (head == NULL) { /* danh sách đang không có nút nào*/
    head = new_node;
}
else {
    //dịch chuyển con trỏ về cuối danh sách:
    node *last = head;
    while (last->next != NULL) last = last->next;
    //Gán giá trị cho con trỏ next của nút cuối cùng:
    last->next = new_node;
}
```

**Độ phức tạp = ....**

```
node *Insert_ToLast(node *head, NodeType X)
{
    node *new_node;
    new_node = (node *) malloc(sizeof(node));
    new_node->data = X;
    if (head == NULL) head = new_node;
    else
    {
        node *last;
        last = head;
        while (last->next != NULL) //di chuyển đến nút cuối
            last = last->next;
        last->next = new_node;
    }
    return head;
}
```

# Các thao tác trên danh sách liên kết đơn

- Duyệt (Traverse) danh sách
- Thêm (Insert) 1 nút mới vào danh sách
- **Xóa (Delete) 1 nút khỏi danh sách**
- Tìm kiếm dữ liệu (Search) trong danh sách

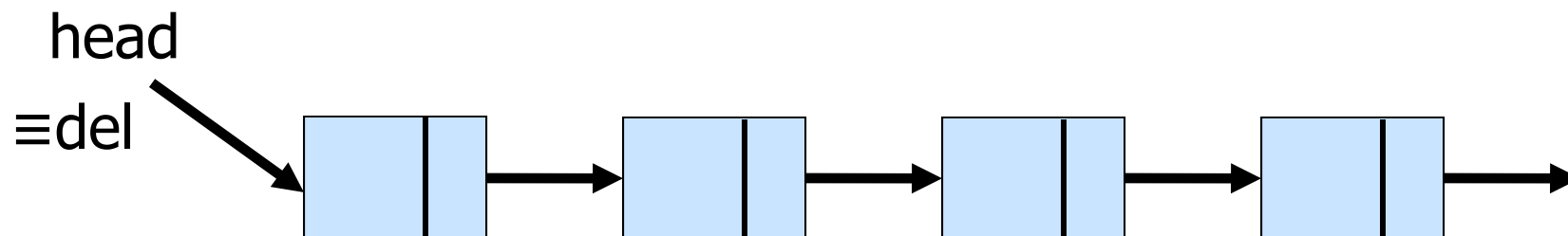
# Các thao tác trên danh sách liên kết đơn: XÓA

- **Xóa một nút**
- Xóa tất cả các nút

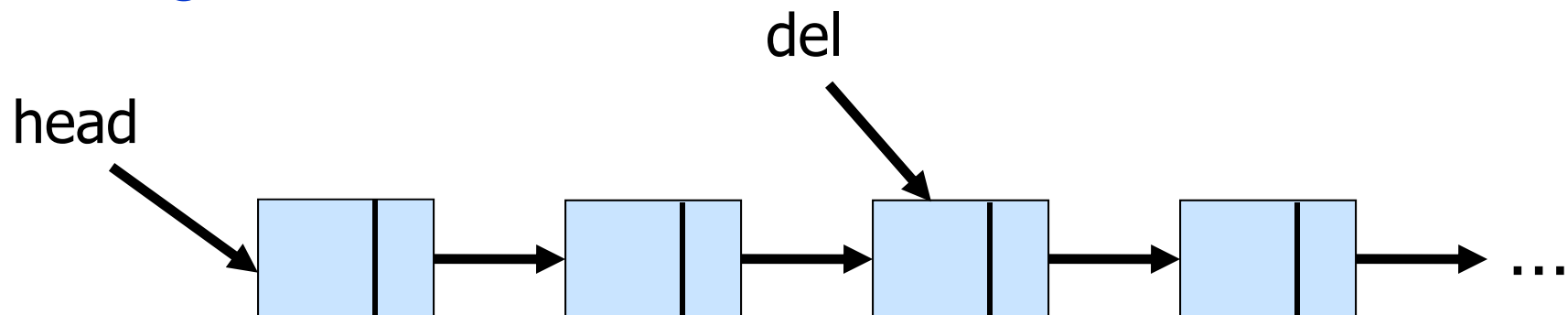
# Các thao tác trên danh sách liên kết đơn: XÓA

Xóa 1 nút:

- Nút đầu tiên trong danh sách



- Nút giữa / cuối danh sách

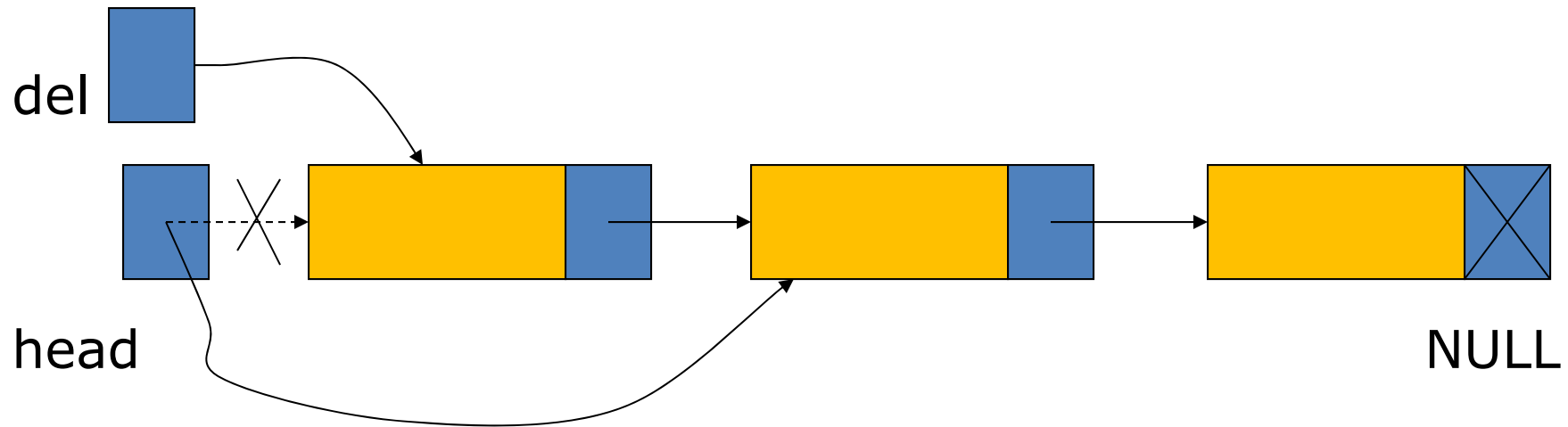




# Xóa nút đầu tiên khỏi danh sách

- Xóa nút `del` đang là nút đầu tiên của danh sách:

➡ `head = del->next;`  
`free(del);`



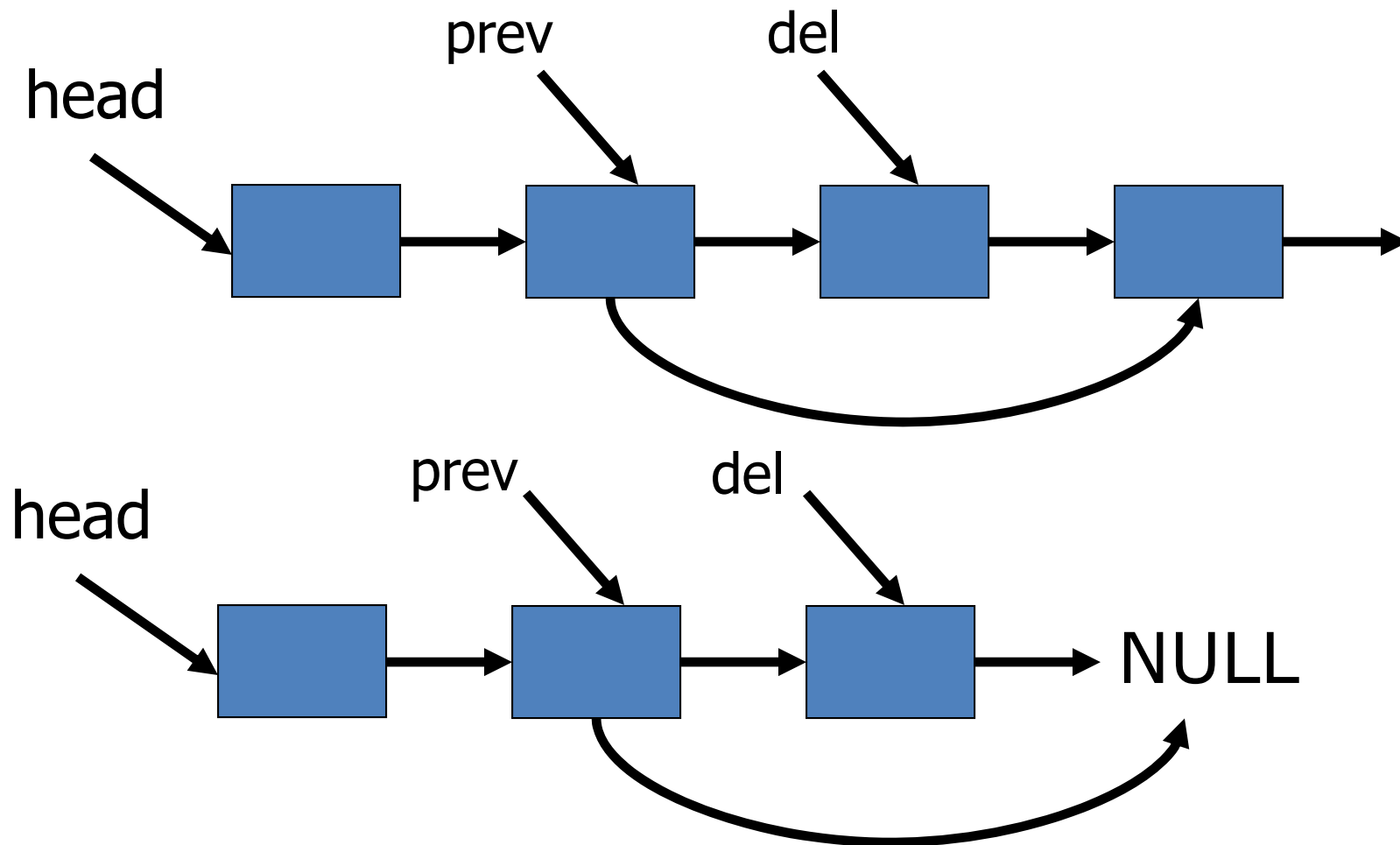
# Xóa nút giữa/cuối danh sách

Xóa nút `del` đang là nút ở giữa/cuối danh sách

**<Xác định con trỏ `prev` trở tới nút ngay trước nút `del`>;**

`prev->next = del->next; //sửa đổi kết nối`

`free(del); //xóa nút del để giải phóng bộ nhớ`

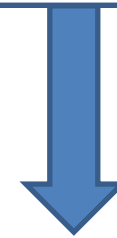


# Xóa nút giữa/cuối danh sách

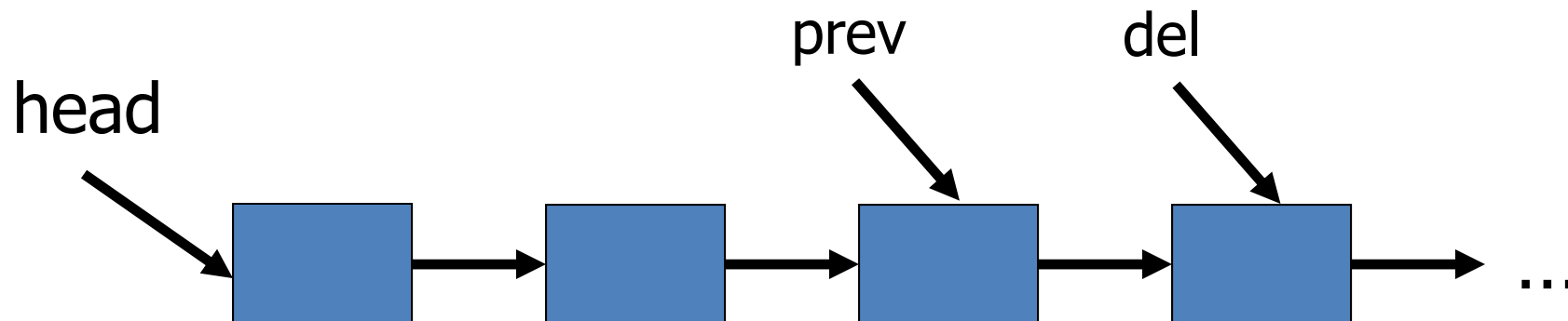
Xóa nút `del` đang là nút ở giữa/cuối danh sách

**<Xác định con trỏ `prev` trở tới nút ngay trước nút `del`>;**

```
prev->next = del->next; //modify the link  
free(del); //delete node del to free memory
```



```
node *prev = head;  
while (prev->next != del) prev = prev->next;
```



# Xóa nút trở bởi con trở del

Viết hàm **node \*Delete\_Node(node \*head, node \*del)**

Xóa nút trở bởi con trở “del” của danh sách có nút đầu tiên trở bởi con trở “head”.

Hàm trả về địa chỉ của nút đầu tiên của danh sách sau khi xóa

- Xóa nút **del** đang là nút đầu tiên của danh sách:

```
head = del->next;  
free(del);
```

**node \*Delete\_Node(node \*head, node \*del)**

```
{  
    if (head == del) //del là nút đầu tiên của danh sách:  
    {  
        head = del->next;  
        free(del);  
    }  
    else{  
        node *prev = head;  
        while (prev->next != NULL) prev = prev->next;  
        prev->next =del->next;  
        free(del);  
    }  
    return head;  
}
```

Xóa nút del đang là nút ở giữa/cuối danh sách

```
<Xác định con trở prev trở tới nút ngay trước nút del>;  
prev->next = del->next; //modify the link  
free(del); //delete node del to free memory
```



```
node *prev =head;  
while (prev->next != del) prev = prev->next;
```

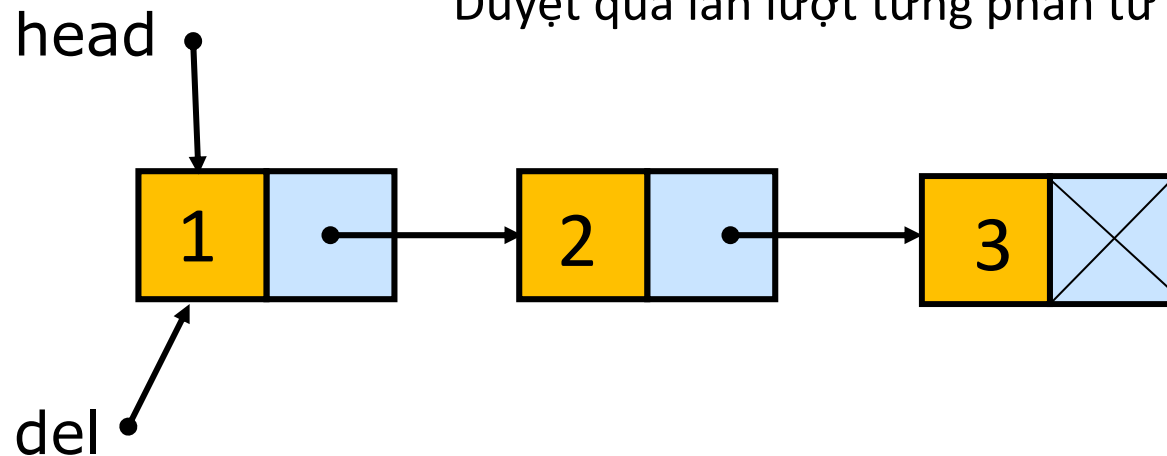
# Các thao tác trên danh sách liên kết đơn: XÓA

- Xóa một nút
- **Xóa tất cả các nút**

# Xóa tất cả các nút trong danh sách

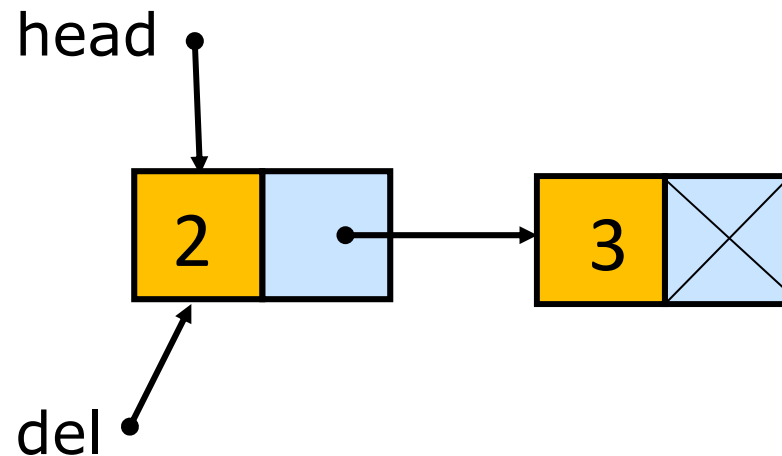
```
➡ del = head ;  
while (del != NULL)  
{  
    head = head->next;  
    free(del);  
    del = head;  
}
```

Duyệt qua lần lượt từng phần tử từ đầu đến cuối danh sách



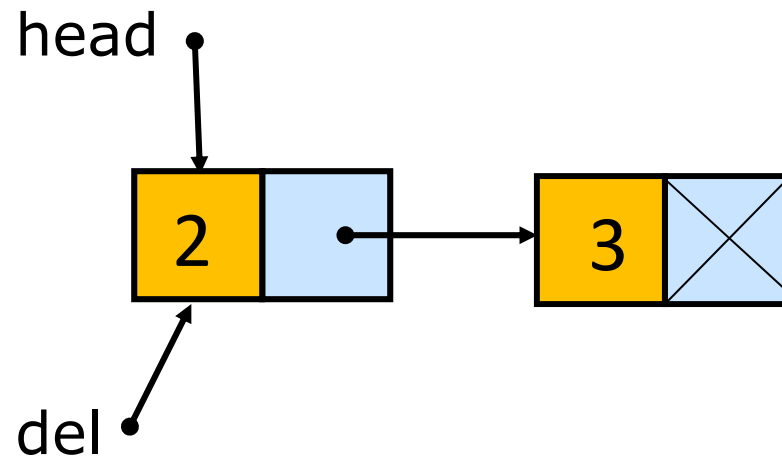
# Xóa tất cả các nút trong danh sách

```
del = head ;  
while (del != NULL)  
{  
    head = head->next;  
    free(del);  
    del = head;  
}
```



# Xóa tất cả các nút trong danh sách

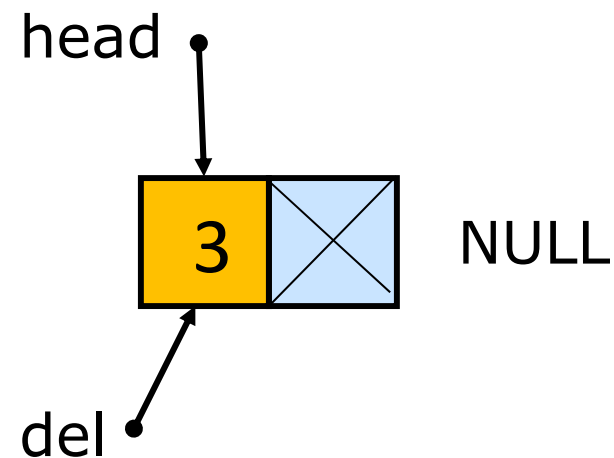
```
del = head ;  
➔ while (del != NULL)  
{  
    head = head->next;  
    free(del);  
    del = head;  
}
```





# Xóa tất cả các nút trong danh sách

```
del = head ;  
while (del != NULL)  
{  
    head = head->next;  
    free(del);  
    del = head;  
}
```



# Xóa tất cả các nút trong danh sách

Viết hàm `node* deleteList (node* head)`

Xóa tất cả các nút trong danh sách có nút đầu trỏ bởi con trỏ head

Hàm trả về con trỏ head sau khi xóa

`node* deleteList (node* head)`

```
{
    node *del = head ;
    while (del != NULL)
    {
        head = head->next;
        free(del);
        del = head;
    }
    return head;
}
```

# Kiểm tra xem danh sách liên kết đơn rỗng hay không

Viết hàm `int IsEmpty(node *head)`

Để kiểm tra xem danh sách liên kết đơn rỗng hay không (biến con trỏ `head` trỏ đến nút đầu tiên trong danh sách).

Hàm trả về giá trị 1 nếu danh sách rỗng; giá trị 0 nếu ngược lại

```
int IsEmpty(node *head) {  
    if (head == NULL)  
        return 1;  
    else return 0;  
}
```

# Các thao tác trên danh sách liên kết đơn

- Duyệt (Traverse) danh sách
- Thêm (Insert) 1 nút mới vào danh sách
- Xóa (Delete) 1 nút khỏi danh sách
- **Tìm kiếm dữ liệu (Search) trong danh sách**

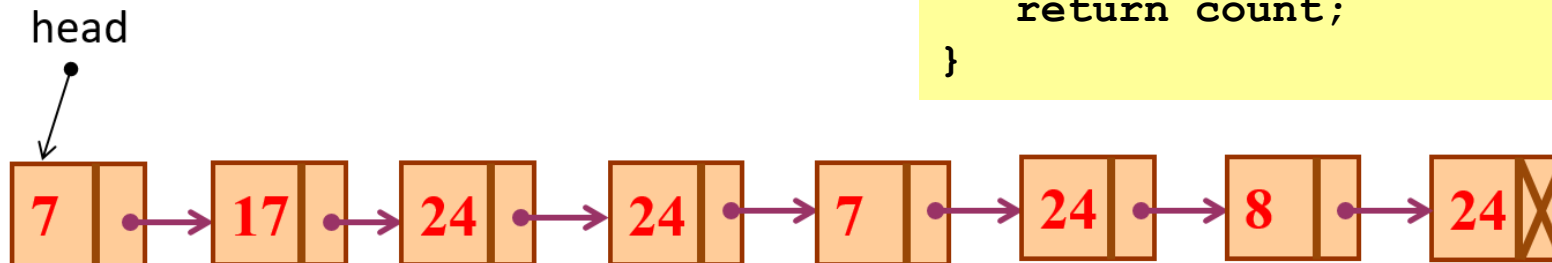
# Tìm kiếm

- Để tìm kiếm một phần tử, ta cần duyệt danh sách từ phần tử đầu tiên cho đến khi ta tìm ra hoặc đến khi đến cuối danh sách mà vẫn không tìm thấy.

Ví dụ: Cho danh sách liên kết chứa các số nguyên. Đếm số nút có giá trị bằng x.

```
typedef struct node {  
    int data;  
    struct node* next;  
}node;  
node* head;
```

```
int countNodes(int x){  
    int count = 0;  
    node* e = head;  
    while(e != NULL){  
        if(e->data == x) count++;  
        e = e->next;  
    }  
    return count;  
}
```



```
int Result1 = countNodes(24);
```

```
int a = 7;
```

```
int Result2 = countNodes(a);
```

**Result1 = ?**

**Result2 = ?**

# Độ phức tạp tính toán: Danh sách liên kết đơn và mảng 1 chiều

Thao tác	Mảng 1 chiều	Danh sách liên kết đơn
Thêm vào đầu	$O(n)$	$O(1)$
Thêm vào cuối	$O(1)$	$O(1)$ nếu có con trỏ tail trỏ vào phần tử cuối của danh sách $O(n)$ nếu danh sách không có con trỏ <b>tail</b>
Thêm vào giữa*	$O(n)$	$O(n)$
Xóa phần tử đầu	$O(n)$	$O(1)$
Xóa phần tử cuối	$O(1)$	$O(n)$
Xóa phần tử giữa*	$O(n)$ : $O(1)$ cho thao tác xóa, và $O(n)$ để dịch chuyển các phần tử	$O(n)$ : $O(n)$ cho thao tác tìm kiếm, và sau đó là $O(1)$ cho thao tác xóa
Tìm kiếm	$O(n)$ tìm kiếm tuyến tính $O(\log n)$ tìm kiếm nhị phân	$O(n)$
Chỉ số: Tìm phần tử tại vị trí thứ $k$	$O(1)$	$O(n)$

\* giữa: không phải cuối, cũng không phải đầu

## Phân tích sử dụng danh sách liên kết so với mảng 1 chiều

- Những ưu điểm của việc dùng danh sách liên kết:
  - Không xảy ra vượt mảng, ngoại trừ hết bộ nhớ.
  - Chèn và Xoá được thực hiện dễ dàng hơn là cài đặt mảng.
  - Với những bản ghi lớn, thực hiện di chuyển con trỏ là nhanh hơn nhiều so với thực hiện di chuyển các phần tử của danh sách.
- Những bất lợi khi sử dụng danh sách liên kết:
  - Dùng con trỏ đòi hỏi bộ nhớ phụ.
  - Linked lists không cho phép truy cập trực tiếp.
  - Tốn thời gian cho việc duyệt và biến đổi con trỏ.
  - Lập trình với con trỏ là *khá rắc rối*.

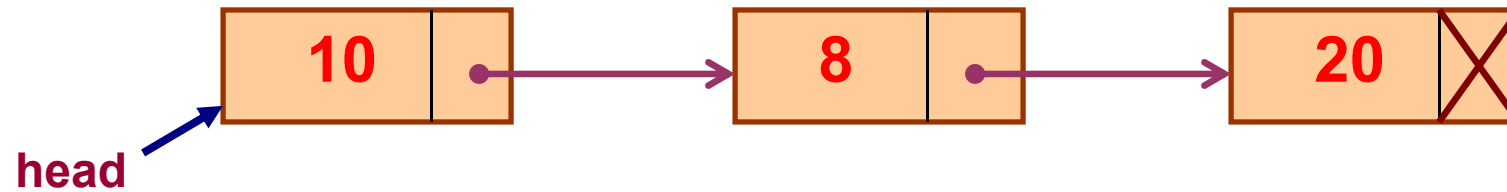
# Phân tích sử dụng danh sách liên kết so với mảng 1 chiều

1D-array	Singly-linked list
Kích thước cố định: thay đổi kích thước là thao tác tốn thời gian	Kích thước thay đổi dễ dàng
Thao tác thêm và xóa tốn nhiều thời gian: vì các phần tử thường phải dịch chuyển	Thao tác thêm và xóa thực hiện dễ dàng: không cần phải dịch chuyển các phần tử
Cho phép truy cập trực tiếp (dễ dàng xác định phần tử tại vị trí thứ k của mảng)	Không cho phép truy cập trực tiếp → Không phù hợp với các thao tác đòi hỏi truy cập phần tử qua chỉ số (vị trí) trong danh sách, ví dụ như sắp xếp
Không lãng phí bộ nhớ nếu số phần tử trong mảng bằng đúng hoặc gần bằng kích thước mảng khi khai báo mảng; nếu không sẽ rất lãng phí bộ nhớ.	Cần thêm bộ nhớ để lưu trữ các con trỏ chứa địa chỉ của nút kế tiếp; tuy nhiên nhờ đó cho phép sử dụng đúng dung lượng bộ nhớ vừa đủ cho yêu cầu lưu trữ các phần tử
Truy cập tuần tự nhanh hơn vì các phần tử được lưu trữ liên tiếp kế tiếp nhau trong bộ nhớ	Truy cập tuần tự chậm vì các phần tử không được lưu trữ kế tiếp nhau trong bộ nhớ

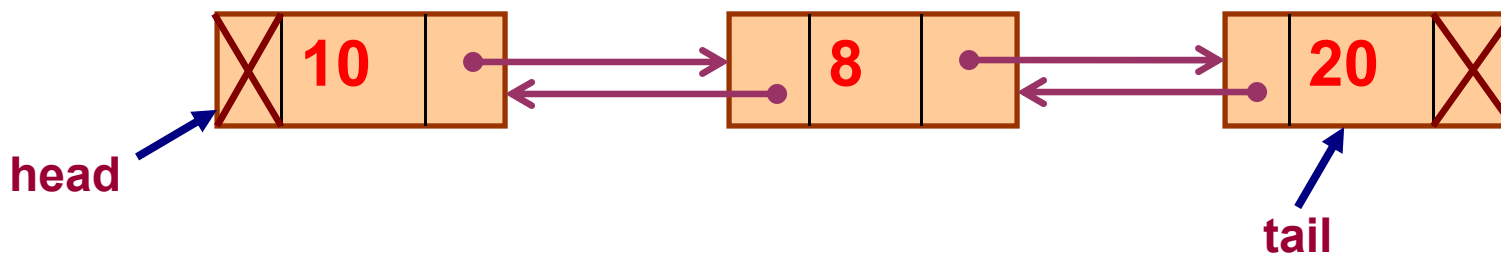


### 3. Danh sách liên kết (Linked list)

- Danh sách liên kết đơn

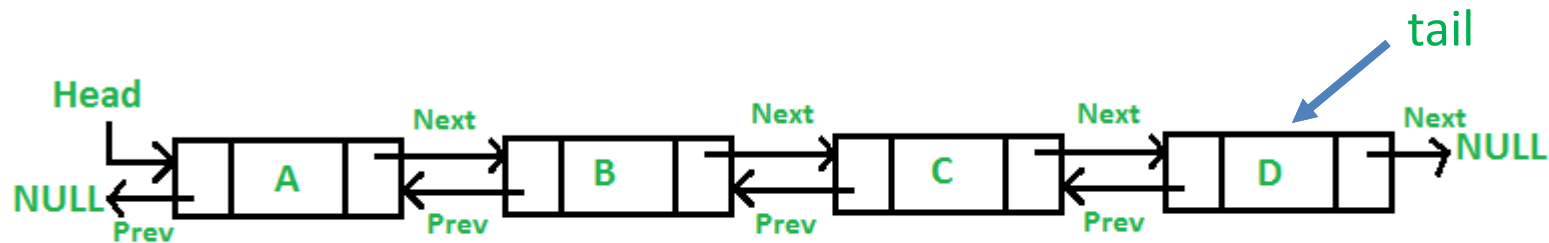


- Danh sách liên kết đôi (Doubly linked list)

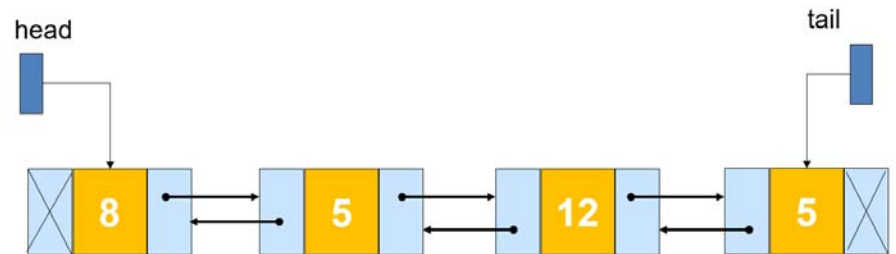


# Danh sách liên kết đôi (Doubly linked list)

- Danh sách liên kết đôi (**D**oubly **L**inked **L**ist (DLL)) chứa con trỏ **next** trỏ tới nút kế tiếp và dữ liệu (data) giống như trong danh sách liên kết đơn, ngoài ra còn có thêm một con trỏ **prev** để trỏ vào phần tử ngay trước



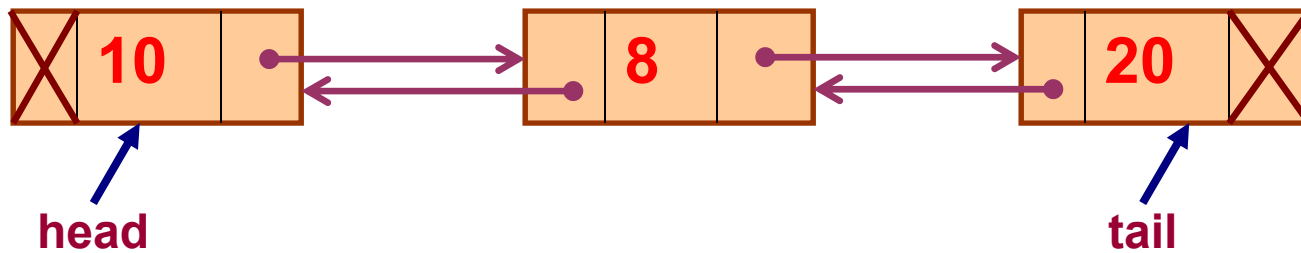
- 2 nút đặc biệt: **tail** và **head**
  - head có con trỏ **prev** = NULL
  - tail có con trỏ **next** = NULL



- Các thao tác cơ bản (thêm, xóa, duyệt, tìm kiếm) được thực hiện tương tự như trong danh sách liên kết đơn

# Danh sách liên kết đôi (Doubly linked list)

- Khai báo 1 danh sách liên kết đơn lưu trữ các số nguyên (kiểu `int`) :

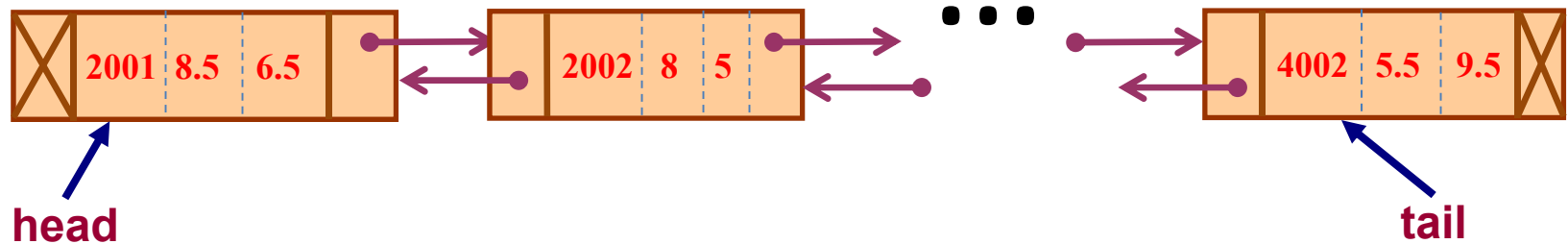


```
typedef struct dllist{  
    int number;  
    struct dllist *next;  
    struct dllist *prev;  
} dllist;  
dllist *head, *tail;
```



# Danh sách liên kết đôi (Doubly linked list)

- Khai báo danh sách liên kết đôi lưu trữ dữ liệu: mã sinh viên (id), điểm 2 môn học toán và lý



```
typedef struct{
    char id[15];
    float math, physics;
}student;
```

```
typedef struct dllist {
    student data;
    struct ddlist* next;
    struct ddlist* prev;
}dllist;
dllist *head, *tail;
```

```
struct student{
    char id[15];
    float math, physics;
};
```

```
typedef struct dllist{
    struct student data;
    struct ddlist* next;
    struct ddlist* prev;
}dllist;
dllist *head, *tail;
```

# Danh sách liên kết đôi – Ví dụ

```
typedef struct dllist{  
    char data;  
    struct dllist *prev;  
    struct dllist *next;  
}dllist;  
dllist *node1, *node2, *node3;
```

```
node1->data='a';  
node2->data='b';  
node3->data='c';  
node1->prev=NULL;  
node1->next=node2;  
node2->prev=node1;  
node2->next=node3;  
node3->prev=node2;  
node3->next=NULL;
```



# Khai báo danh sách liên kết đôi (doubly linked list)

```
typedef ... NodeType;
typedef struct node{
    NodeType data;
    struct node* prev;
    struct node* next;
}node;
node *head, *tail;
```

```
typedef ... NodeType;
struct node{
    NodeType data;
    struct node* next;
};
struct node *head, *tail;
```

Khai báo trên định nghĩa kiểu node, trong đó node là kiểu bản ghi mô tả một nút gồm hai trường:

- **element** : lưu dữ liệu có kiểu là ElementType (đã được định nghĩa, có thể gồm nhiều thành phần)
- **prev** : con trỏ, lưu địa chỉ của nút kề trước.
- **next** : con trỏ, lưu địa chỉ của nút kế tiếp.

Biến con trỏ **head** : lưu địa chỉ của nút đầu tiên trong danh sách

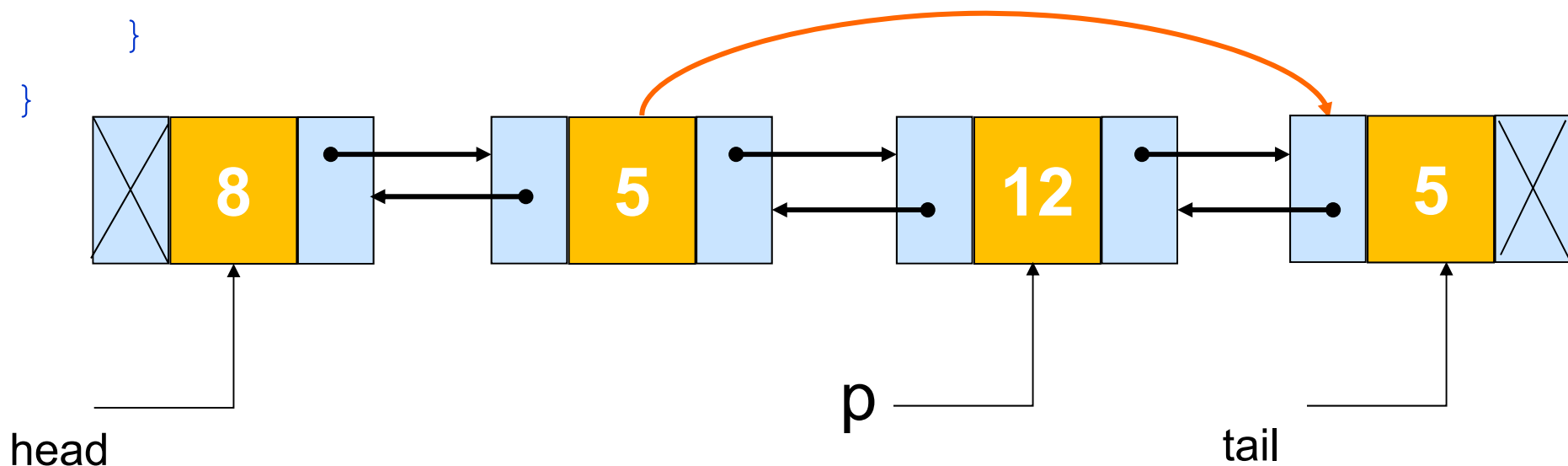
Biến con trỏ **tail** : lưu địa chỉ của nút cuối cùng trong danh sách

```
typedef struct{
    char ma[15];
    struct{
        float toan, ly, hoa, tong;
    } DT;
}thisinh;
```

```
typedef struct node{
    thisinh data;
    struct node *prev;
    struct node *next;
}node;
node *head, *tail;
```

# Xóa 1 nút trở bởi con trỏ p

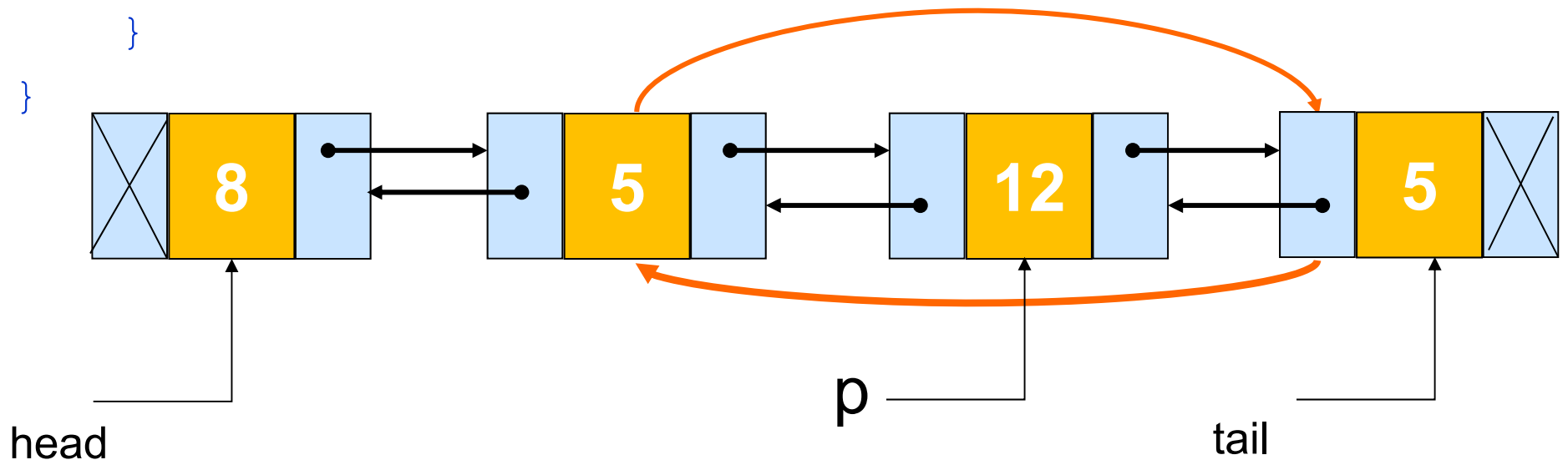
```
void Delete_Node (ddlist *p){  
    if (head == NULL) printf("Danh sách rỗng");  
    else {  
        if (p==head) head = head->next; //Xóa phần tử đầu ds  
        else p->prev->next = p->next;  
        if (p->next!=NULL) p->next->prev = p->prev;  
        else tail = p->prev;  
        free(p);  
    }  
}
```



head là con trỏ trỏ đến đầu danh sách đã cho (ddlist \*head;)

# Xóa 1 nút trở bởi con trỏ p

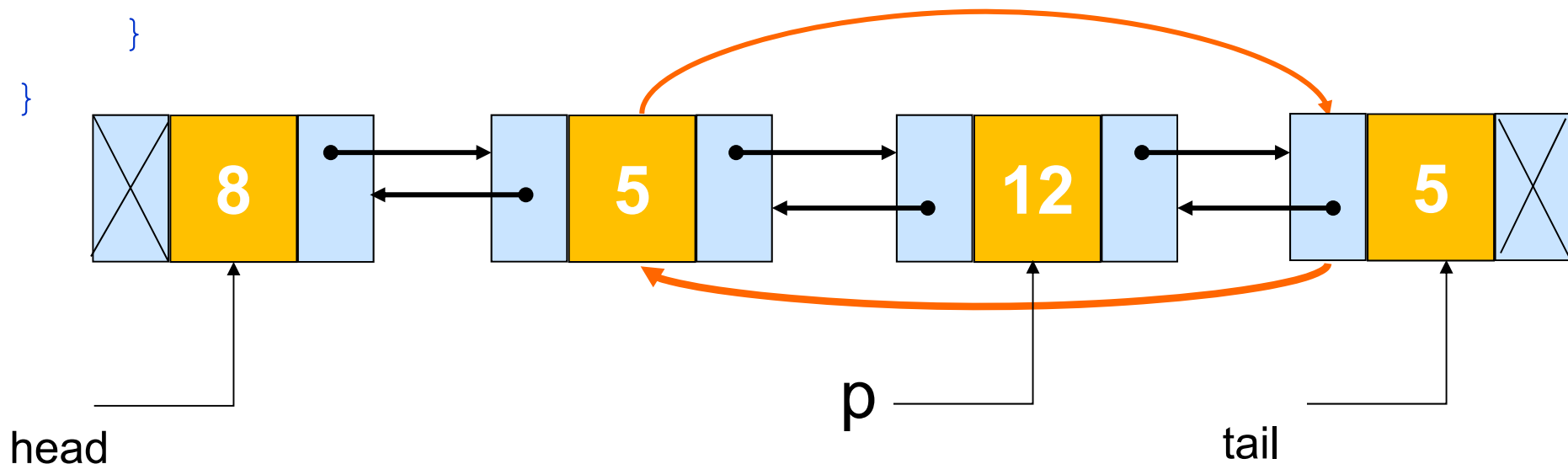
```
void Delete_Node (ddlist *p){  
    if (head == NULL) printf(" Danh sách rỗng");  
    else {  
        if (p==head) head = head->next; // Xóa phần tử đầu ds  
        else p->prev->next = p->next;  
        if (p->next!=NULL) p->next->prev = p->prev;  
        else tail = p->prev;  
        free(p);  
    }  
}
```





# Xóa 1 nút trở bởi con trỏ p

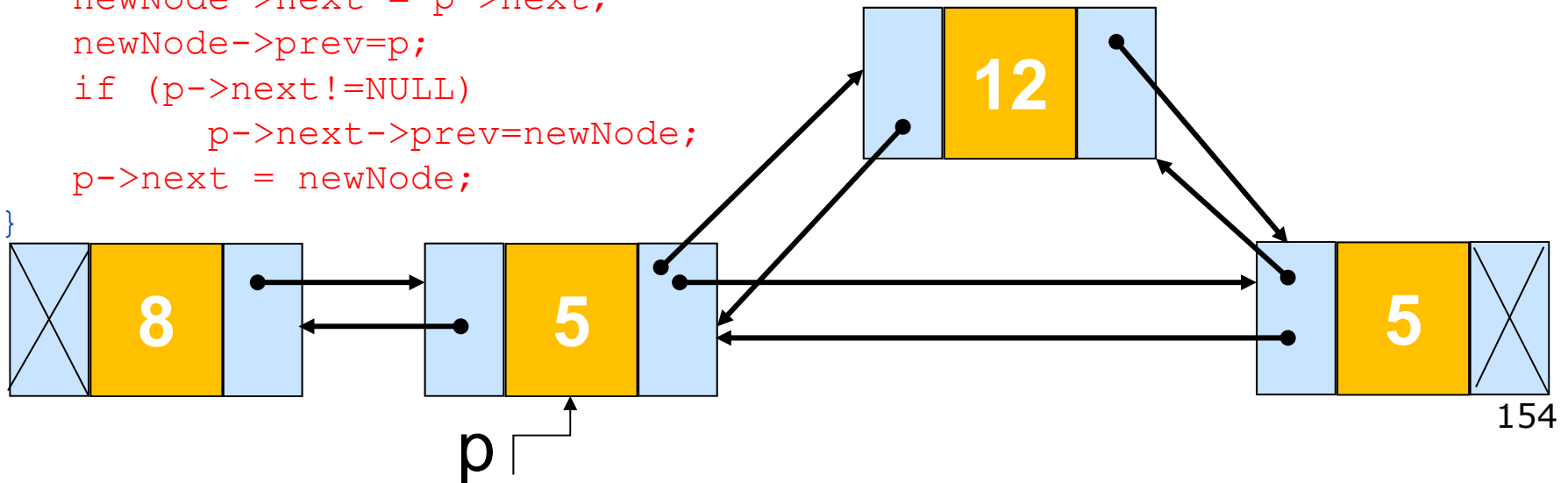
```
void Delete_Node (ddlist *p){  
    if (head == NULL) printf(" Danh sách rỗng");  
    else {  
        if (p==head) head = head->next; // Xóa phần tử đầu ds  
        else p->prev->next = p->next;  
        if (p->next!=NULL) p->next->prev = p->prev;  
        else tail = p->prev;  
        free (p) ;  
    }  
}
```



# Thêm 1 nút vào sau nút đang trỏ bởi con trỏ p

```
void Insert_Node (NodeType X, ddlist *p){  
    if (head == NULL){ // List is empty  
        head =(ddlist*)malloc(sizeof(ddlist));  
        head->data = X;  
        head->prev =NULL;  
        head->next =NULL;  
    }  
    else{  
        ddlist *newNode;  
        newNode=(ddlist*)malloc(sizeof(ddlist));  
        newNode->data = X;  
        newNode->next = NULL;
```

```
        newNode->next = p->next;  
        newNode->prev=p;  
        if (p->next!=NULL)  
            p->next->prev=newNode;  
        p->next = newNode;  
    }  
}
```



## Bài tập: Tạo danh sách liên kết đôi chứa các số nguyên

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct dllist {
    int number;
    struct dllist *next;
    struct dllist *prev;
} dllist;
dllist *head, *tail;

/* Thêm 1 nút p vào cuối danh sách */
void append_node(dllist *p);
/* Thêm 1 nút mới p vào sau nút đang trỏ bởi con trỏ after */
void insert_node(dllist *p, dllist *after);
/* Xóa 1 nút đang trỏ bởi con trỏ p */
void delete_node(dllist *p);
```

```
/* Thêm 1 nút p vào cuối danh sách */  
void append_node(dllist *p) {  
    if(head == NULL)  
    {  
        head = p;  
        p->prev = NULL;  
    }  
    else {  
        tail->next = p;  
        p->prev = tail;  
    }  
    tail = p;  
    p->next = NULL;  
}
```

/\* Thêm 1 nút mới p vào sau nút đang trỏ bởi con trỏ after \*/

void insert\_node(dllist \*p, dllist \*after) {

    p->next = after->next;

    p->prev = after;

    if(after->next != NULL)

        after->next->prev = p;

    else

        tail = p;

    after->next = p;

}

/\* Xóa 1 nút đang trỏ bởi con trỏ p \*/

void delete\_node(dllist \*p) {

    if(p->prev == NULL)

        head = p->next;

    else p->prev->next = p->next;

    if(p->next == NULL)

        tail = p->prev;

    else p->next->prev = p->prev;

}

```

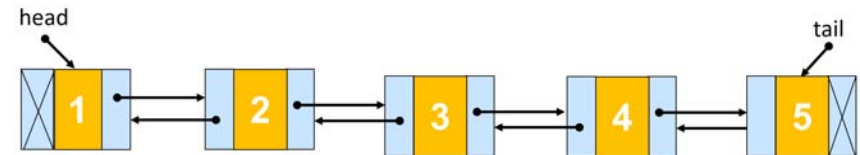
int main( ) {
    dllist *tempnode; int i;
    /* tạo danh sách liên kết đôi gồm 5 số nguyên 1,2,3,4,5 */
    for(i = 1; i <= 5; i++) {
        tempnode = (dllist *)malloc(sizeof(dllist));
        tempnode->number = i;
        append_node(tempnode);
    }

    /* in các số trong danh sách theo thứ tự từ đầu đến cuối */
    printf(" Traverse the dll list forward \n");
    for(tempnode = head; tempnode != NULL; tempnode = tempnode->next)
        printf("%d\n", tempnode->number);

    /* in các số trong danh sách theo thứ tự ngược từ cuối lên đầu ds*/
    printf(" Traverse the dll list backward \n");
    for(tempnode = tail; tempnode != NULL; tempnode = tempnode->prev)
        printf("%d\n", tempnode->number);

    /* Giải phóng bộ nhớ cấp phát cho danh sách liên kết */
    while(head != NULL) delete_node(head);
    return 0;
}

```



# Ứng dụng của danh sách liên kết – Ma trận thưa

Để biểu diễn ma trận thưa, ta dùng 1 danh sách liên kết gồm 2 danh sách:

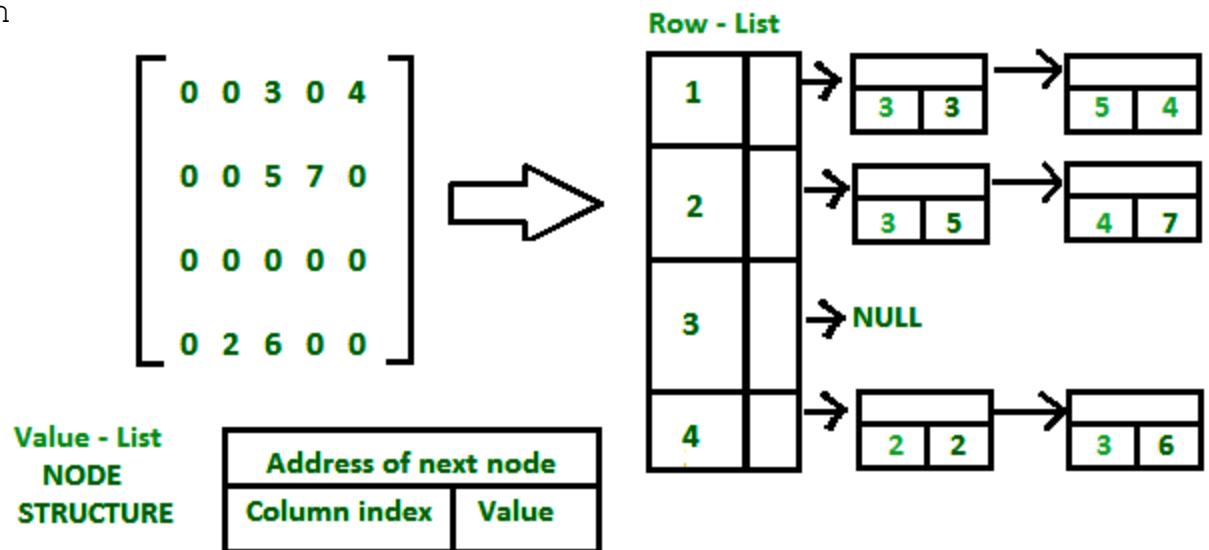
- 1 danh sách liên kết **row\_list** biểu diễn các hàng thuộc ma trận,
- mỗi hàng được biểu diễn bởi 1 danh sách liên kết **value\_list** chứa bộ 3 thông tin: **chỉ số cột (column index)**, **giá trị của phần tử (value)** và **trường địa chỉ**, cho các phần tử khác 0 của ma trận thưa.

Hai danh sách này được lưu trữ theo thứ tự tăng dần của khóa.

//Nút biểu diễn bộ 3 thông tin

```
typedef struct value_list
{
    int column_index;
    int value;
    struct value_list *next;
}value_list;
```

```
typedef struct row_list
{
    int row_number;
    struct row_list *link_down;
    struct value_list *link_right;
} row_list;
```



# Ứng dụng của danh sách liên kết: Cộng đa thức

Đa thức: được định nghĩa bởi 1 danh sách các hệ số và số mũ

- *Bậc* của đa thức = số mũ cao nhất có trong đa thức

$$p(x) = a_1x^{e_1} + a_2x^{e_2} + \dots + a_nx^{e_n}$$

Ví dụ:

Đa thức  $A(x) = 3x^{10} + 2x^5 + 6x^4 + 4$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

Để biểu diễn đa thức, cách đơn giản nhất là sử dụng mảng:  $a[i]$  lưu trữ hệ số của số hạng  $x_i$ . Các thao tác trên đa thức như: cộng hai đa thức, nhân hai đa thức, ... khi đó được thực hiện dễ dàng trên 2 mảng

Mảng a ~A(x)	a[10]	a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
	3	0	0	0	0	2	6	0	0	0	4

Mảng b ~B(x)	b[4]	b[3]	b[2]	b[1]	b[0]
	1	10	3	0	1

$$C(x) = A(x) + B(x) =$$

$$3x^{10} + 2x^5 + 7x^4 + 3x^2 + 5$$

Mảng c ~C(x)	c[10]	c[9]	c[8]	c[7]	c[6]	c[5]	c[4]	c[3]	c[2]	c[1]	c[0]
	=a[10] =3	=a[9] =0	=a[8] =0	=a[7] =0	=a[6] =0	=a[5] =2	=a[4]+b[4] =6+1=7	=a[3]+b[3] =0+10=10	=a[2]+b[2] =0+3=3	=a[1]+b[1] =0+0=0	=a[0]+b[0] =4+1=5



# Ứng dụng của danh sách liên kết: Cộng đa thức

$$A(x) = 2x^{1000} + x^3$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

Sử dụng mảng để lưu trữ hệ số của tất cả các hệ số tương ứng của tất cả các số mũ:

...	2	...	0	1	0	0	0	A
...	0	...	1	10	3	0	1	B
	1000	...	4	3	2	1	0	

$$A(x) + B(x) =$$

Ưu điểm: dễ cài đặt

Nhược điểm: lãng phí bộ nhớ khi đa thức thưa

Tuy nhiên khi đa thức với nhiều hệ số bằng 0 cách biểu diễn đa thức dưới dạng mảng là tốn kém bộ nhớ, chẳng hạn, việc biểu diễn đa thức  $x^{1000} - 1$  đòi hỏi mảng gồm 1001 phần tử.

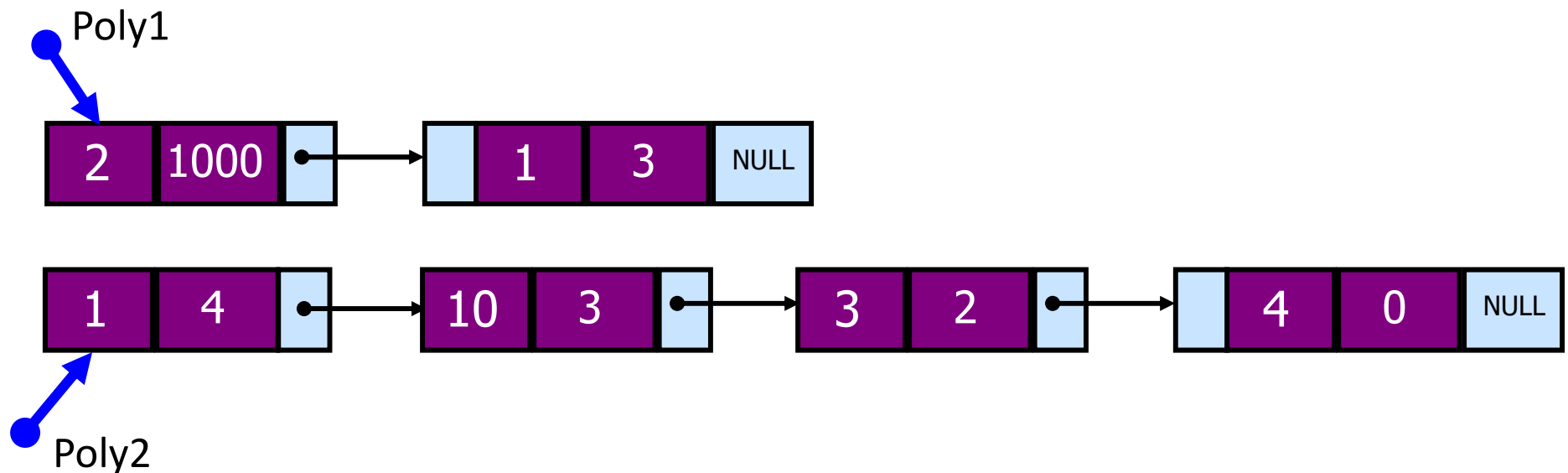
Trong trường hợp đa thức thưa (có nhiều hệ số bằng 0) có thể sử dụng biểu diễn đa thức bởi danh sách móc nối: Ta sẽ xây dựng danh sách chỉ chứa các hệ số khác không cùng số mũ tương ứng.

# Ứng dụng của danh sách liên kết: Cộng đa thức

$$A(x) = 2x^{1000} + x^3$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 4$$

Sử dụng danh sách liên kết để biểu diễn đa thức (chỉ lưu trữ hệ số khác 0 và số mũ tương ứng):

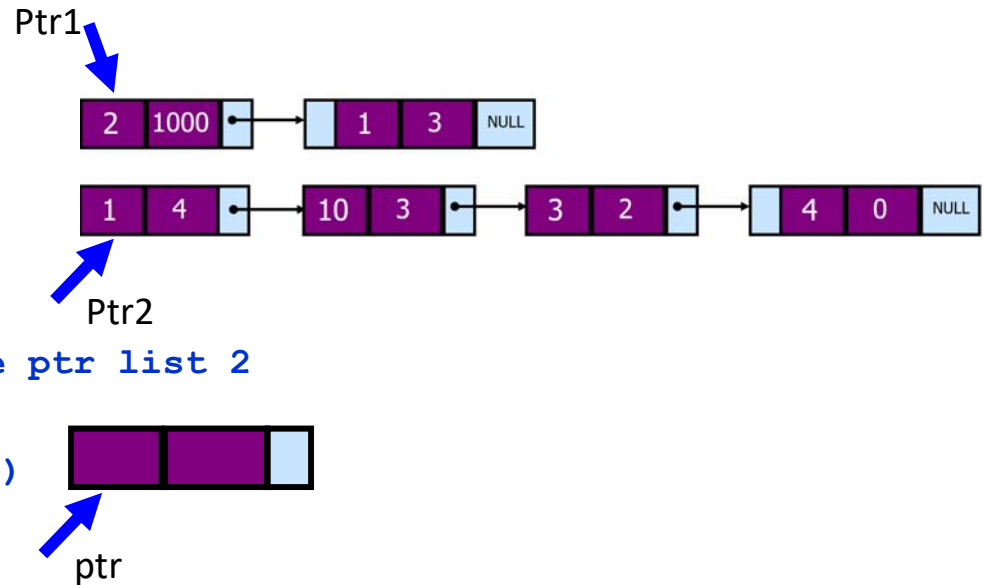


```
struct Polynom {  
    int coef;  
    int exp;  
    struct Polynom *next;  
} *Poly1, *Poly2;
```

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->nextp = node;    //update ptr listResult
} //end of while

```



```

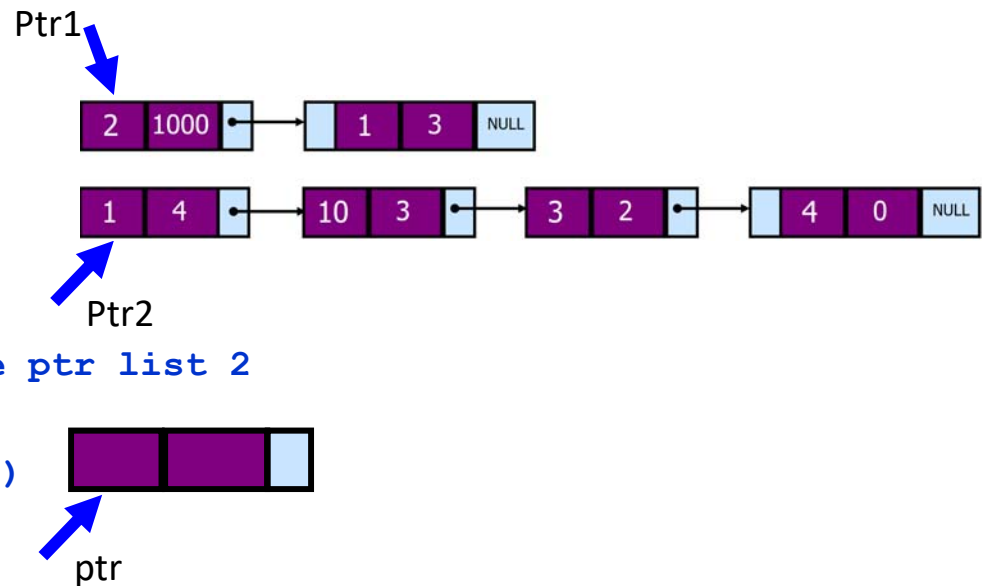
if (ptr1 == NULL)          //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)      //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (TPol *)malloc (sizeof(TPol));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    { node->coef = ptr2->coef + ptr1->coef;
      node->exp = ptr2->exp;
      ptr1 = ptr1->next;    //update ptr list 1
      ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->nextp = node;    //update ptr listResult
} //end of while

```

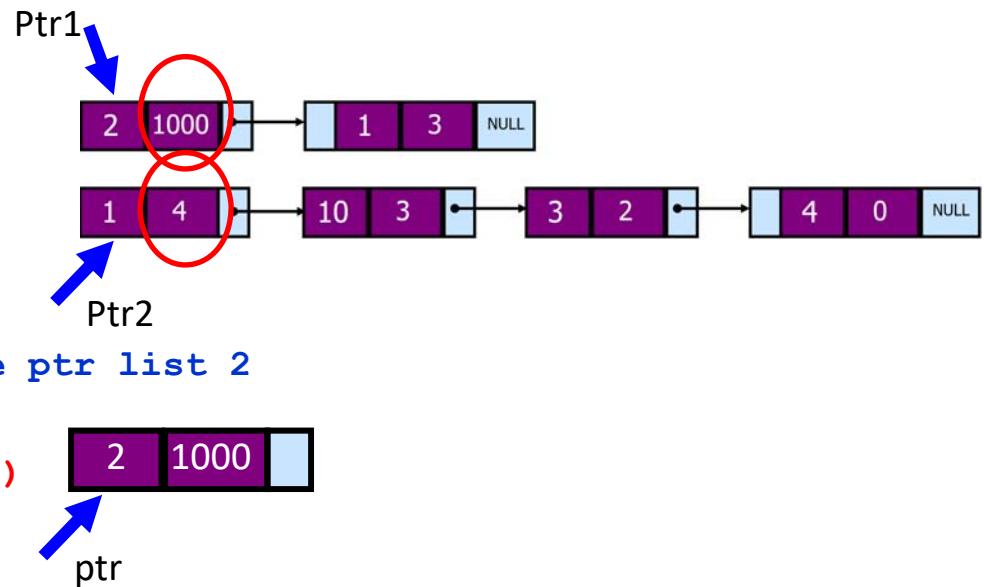


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

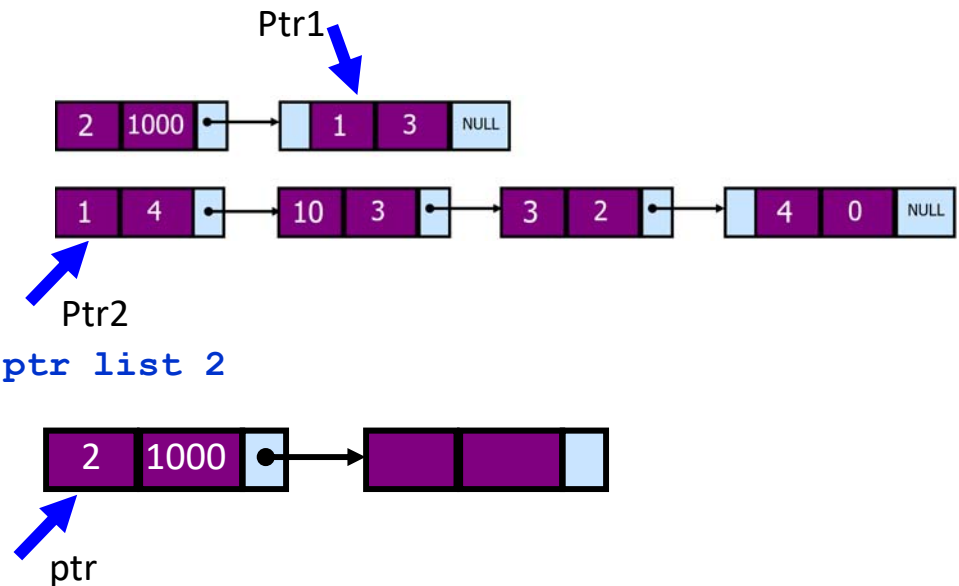


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

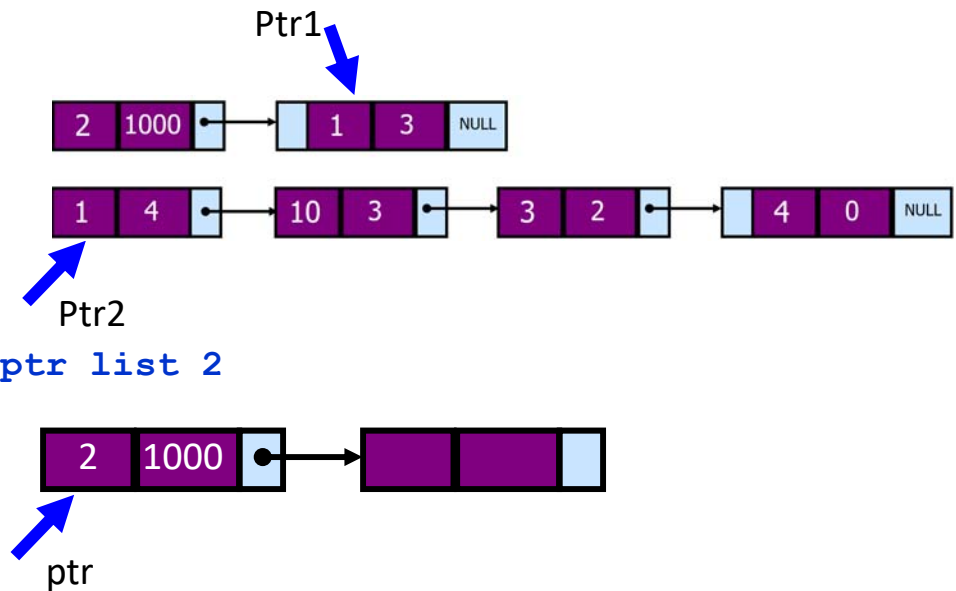


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    { node->coef = ptr2->coef + ptr1->coef;
      node->exp = ptr2->exp;
      ptr1 = ptr1->next;    //update ptr list 1
      ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```



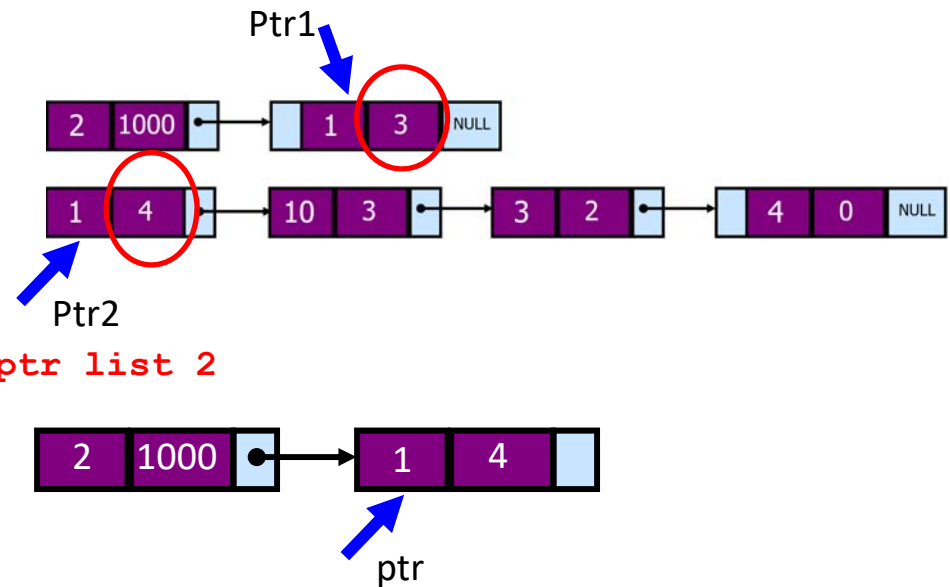
**Minh họa**



```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

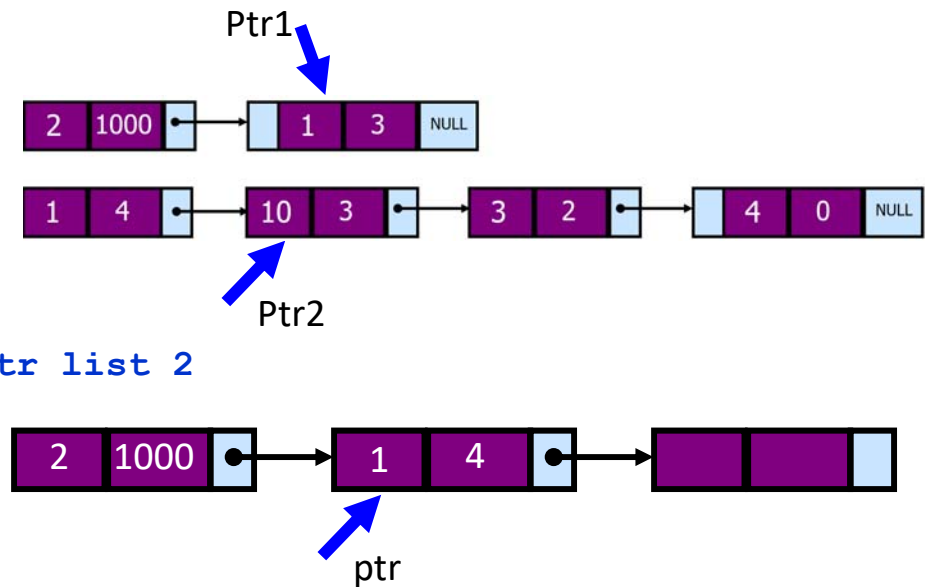


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {   node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

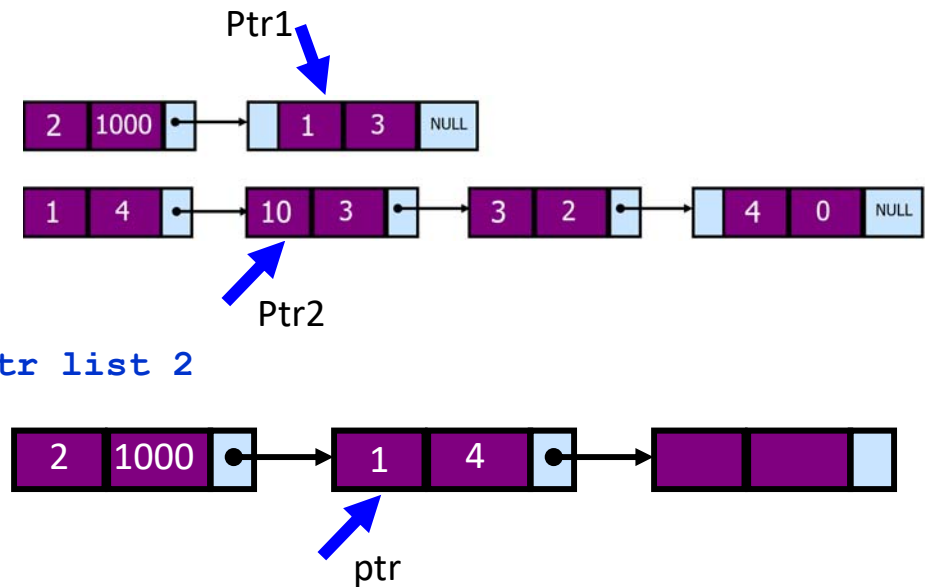


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    { node->coef = ptr2->coef + ptr1->coef;
      node->exp = ptr2->exp;
      ptr1 = ptr1->next;    //update ptr list 1
      ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

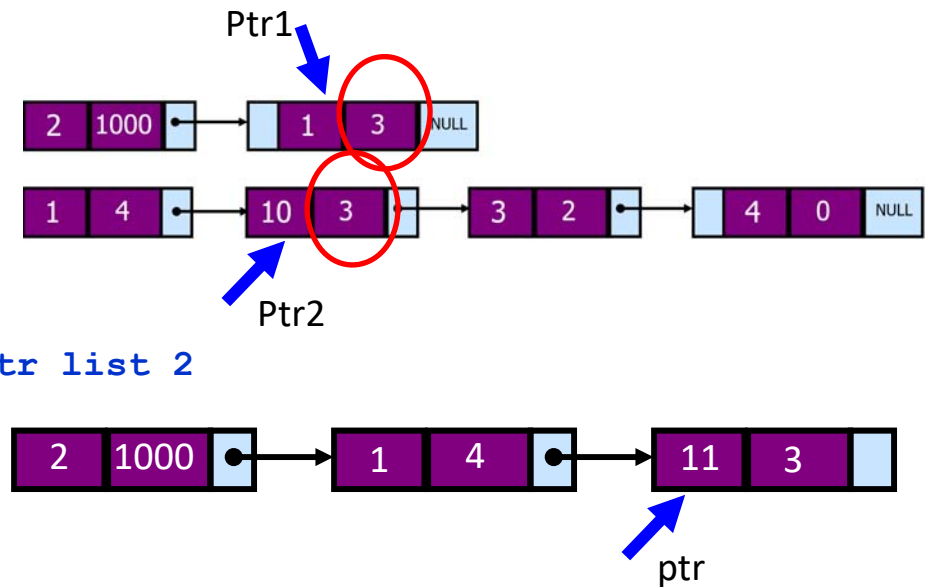


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

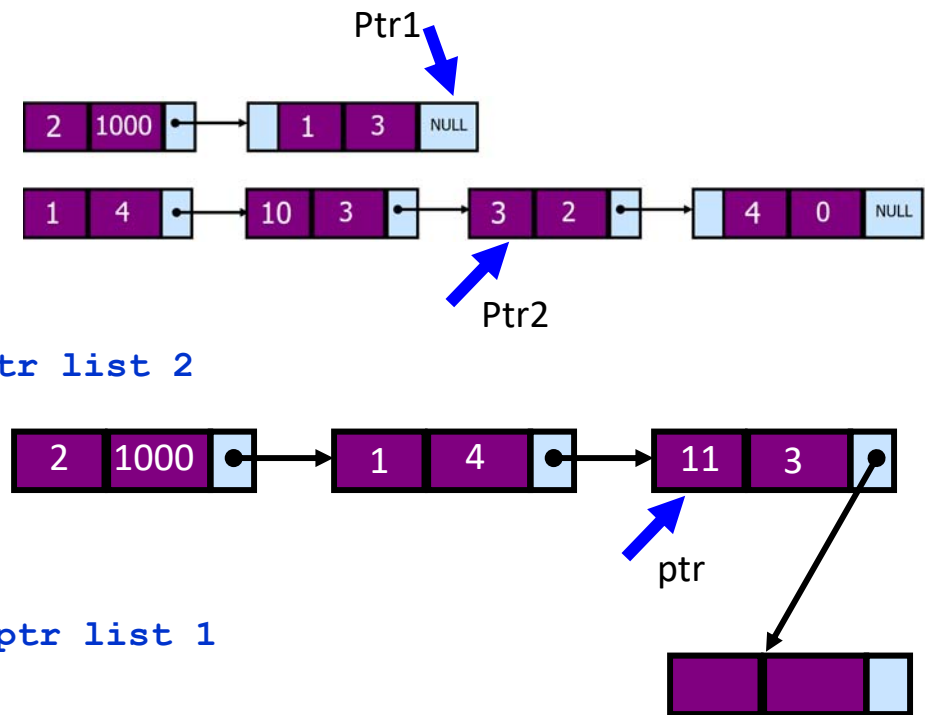


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {   node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

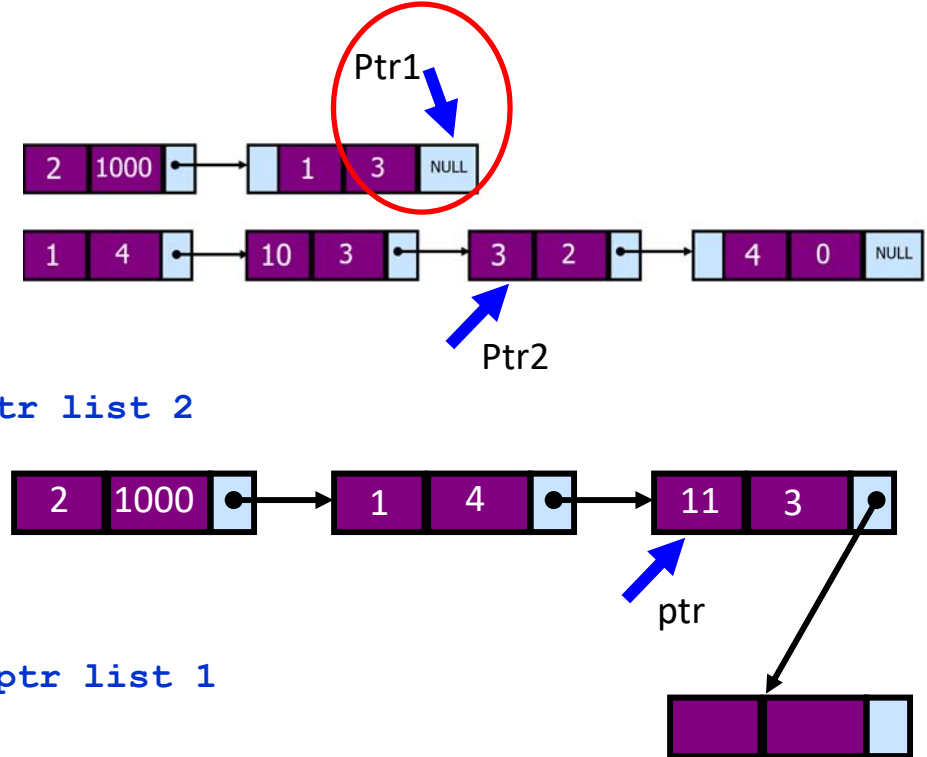


**Minh họa**

```

Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL) {
    ptr = node;
    if (ptr1->exp < ptr2->exp ) {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp ) {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 1
    }
    else
    {
        node->coef = ptr2->coef + ptr1->coef;
        node->exp = ptr2->exp;
        ptr1 = ptr1->next;    //update ptr list 1
        ptr2 = ptr2->next;    //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;    //update ptr listResult
} //end of while

```

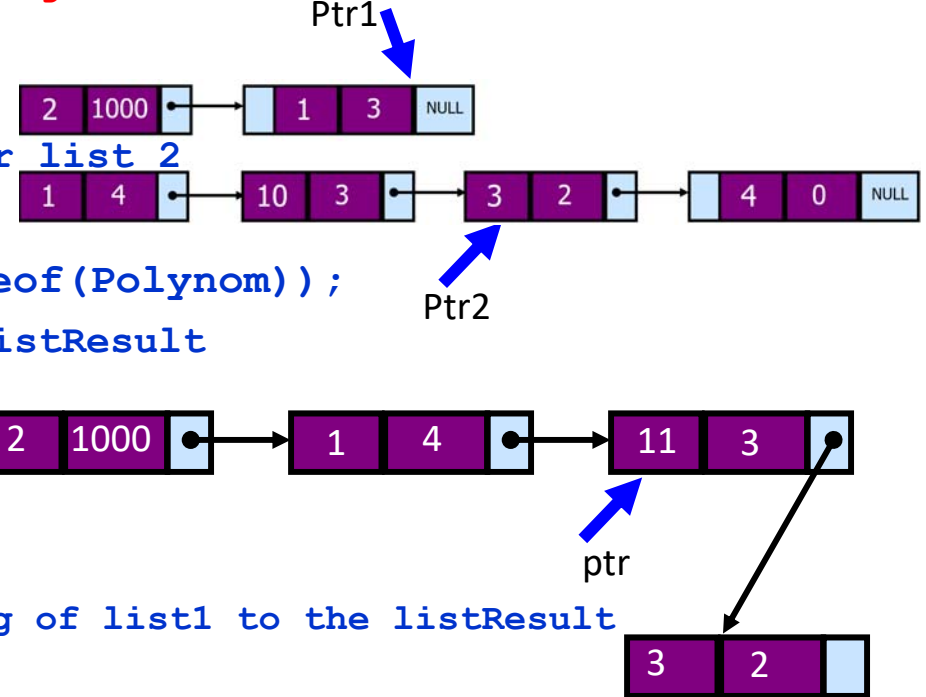


**Minh họa**

```

if (ptr1 == NULL)      //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)    //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *)malloc (sizeof(Polynom));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```



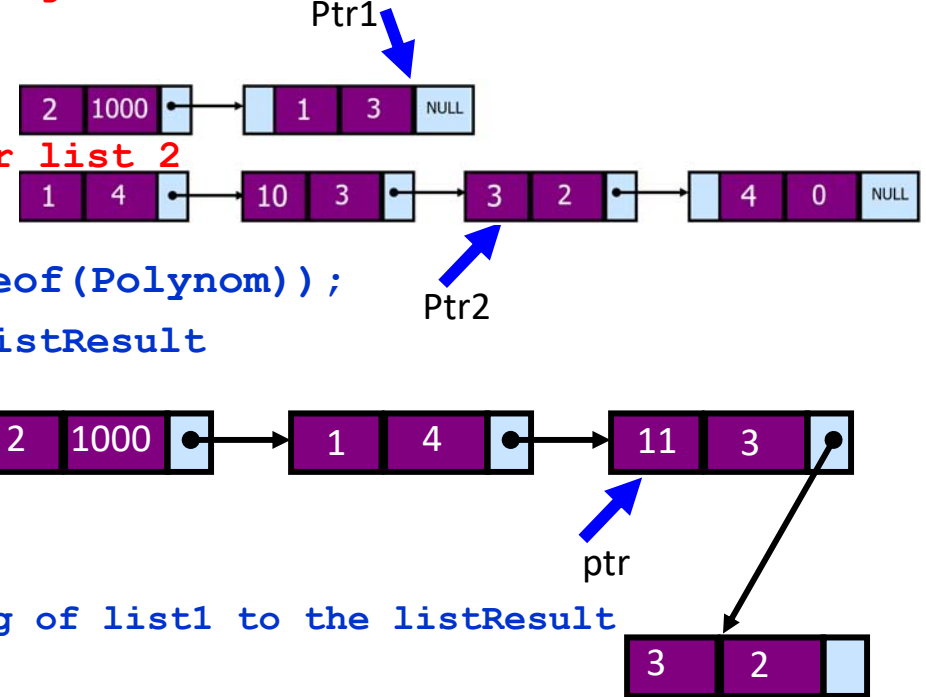
**Minh họa**

```

if (ptr1 == NULL)      //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr list 2
        ptr = node;

        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)    //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *)malloc (sizeof(Polynom));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```



**Minh họa**

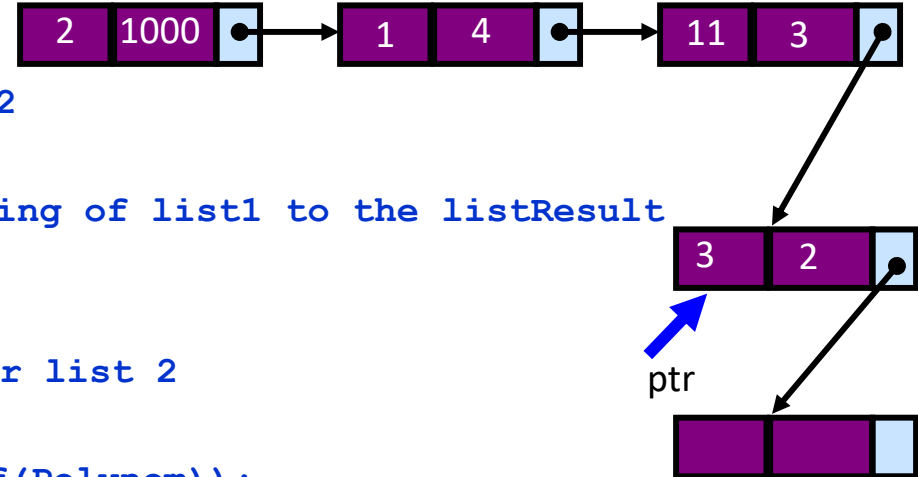
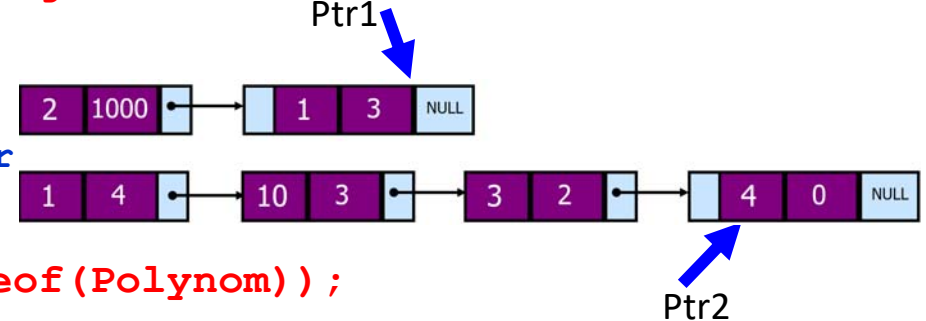


```

if (ptr1 == NULL)      //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr
        ptr = node;

        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)    //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *)malloc (sizeof(Polynom));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```



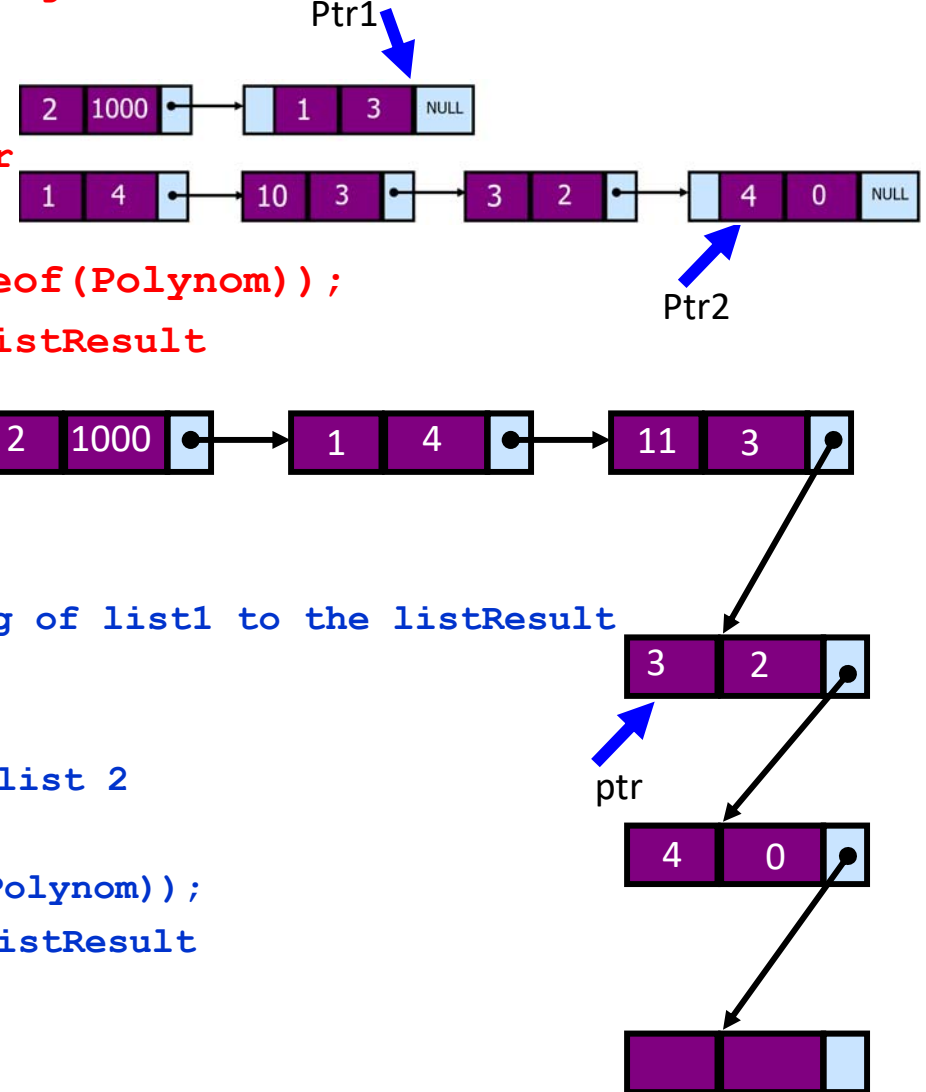
**Minh họa**

```

if (ptr1 == NULL)      //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr
        ptr = node;

        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)    //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *)malloc (sizeof(Polynom));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```



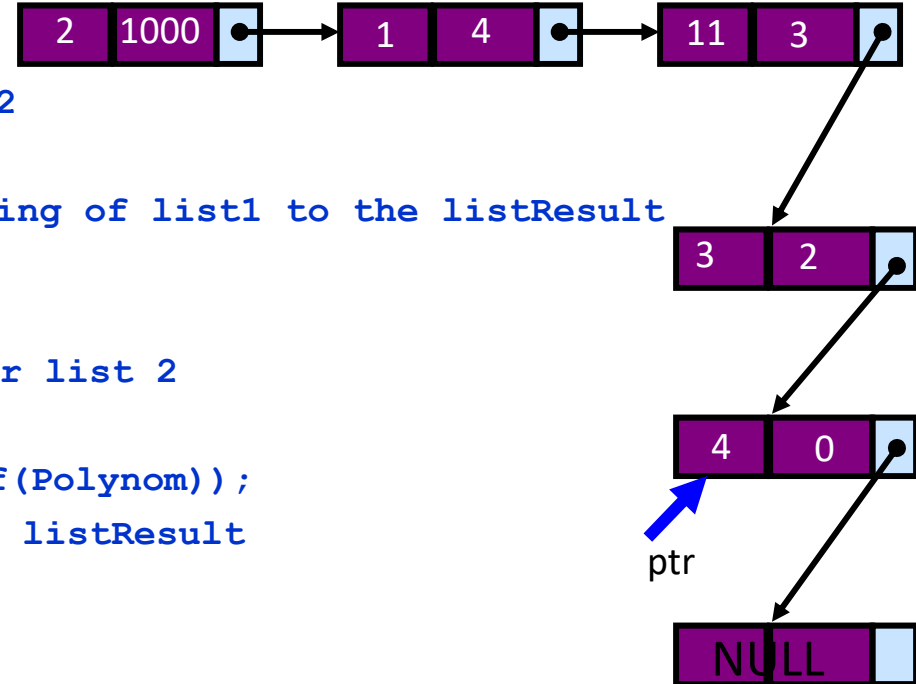
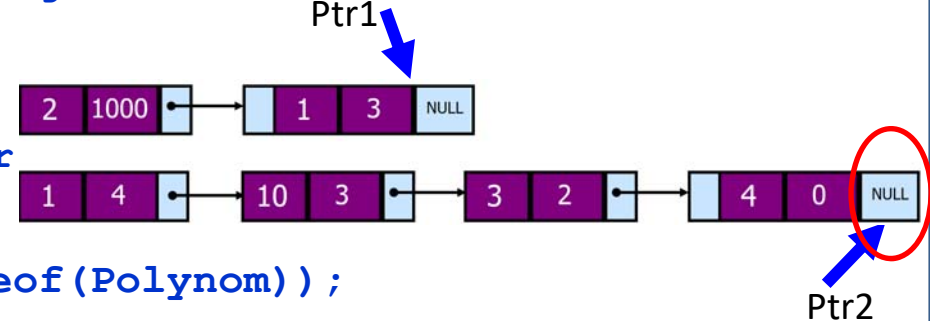
**Minh họa**

```

if (ptr1 == NULL)      //end of list 1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {
        node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;    //update ptr
        ptr = node;

        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
    }
}
else if (ptr2==NULL)    //end of list 2
{
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {
        node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;    //update ptr list 2
        ptr = node;
        node = (Polynom *)malloc (sizeof(Polynom));
        ptr->next = node;    //update ptr listResult
    }
}
ptr->next = NULL;
}

```



$$A(x) = 2x^{1000} + x^3$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 4$$

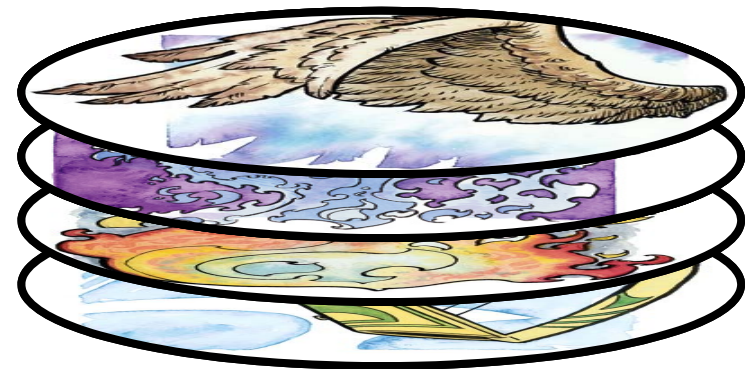
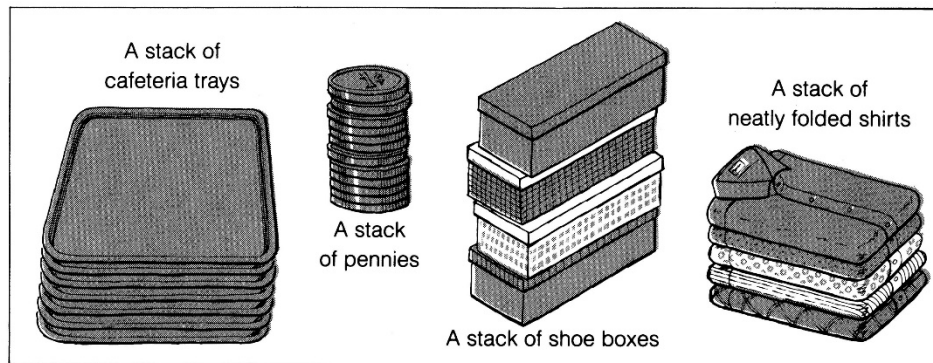
$$\text{Result}(x) = 2x^{1000} + x^4 + 11x^3 + 3x^2 + 4$$

# Nội dung

1. Mảng (Array)
2. Bản ghi (Record)
3. Danh sách liên kết (Linked List)
- 4. Ngăn xếp (Stack)**
5. Hàng đợi (Queue)

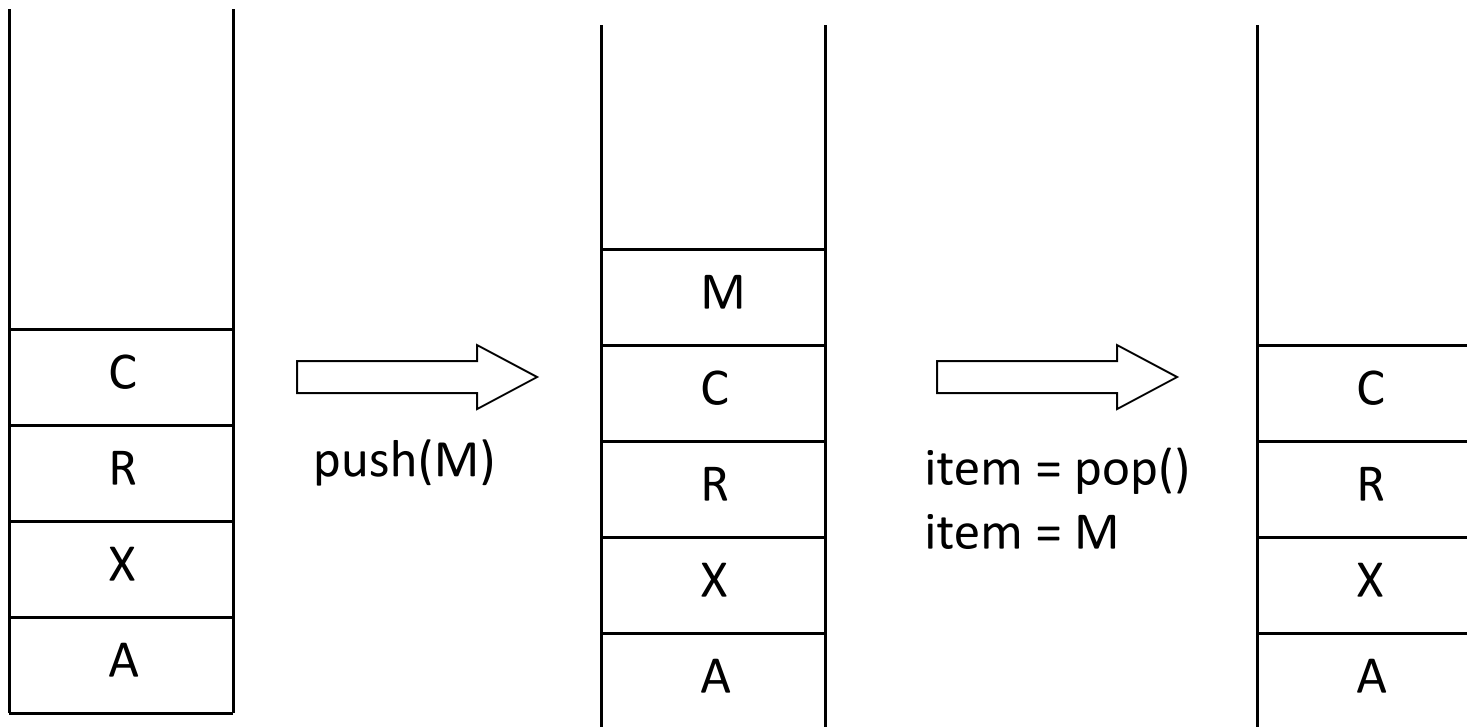
# What is a stack?

- Ngăn xếp là dạng đặc biệt của danh sách tuyến tính trong đó các đối tượng được nạp vào (**push**) và lấy ra (**pop**) chỉ từ một đầu được gọi là đỉnh (**top**) của danh sách.
- *Ngăn xếp có 2 đầu:*
  - Đỉnh(**top**)
  - Đuôi (bottom)
- Phần tử cuối cùng được thêm vào ngăn xếp sẽ được lấy ra khỏi ngăn xếp đầu tiên (nguyên tắc: vào sau – ra trước LIFO)  
(The last element to be added is the first to be removed (**LIFO**: Last In, First Out))



# Các thao tác trên ngăn xếp

- *Push*: thao tác thêm 1 phần tử mới vào đầu ngăn xếp
- *Pop*: thao tác loại bỏ phần tử đang ở đầu ngăn xếp ra khỏi nó



## 4. Ngăn xếp (Stacks)

- Các phép toán cơ bản (stack operations):
  - **push**(object): Nạp vào (bổ sung) một phần tử
  - object **pop**(): Lấy ra (loại bỏ và trả lại) phần tử nạp vào sau cùng
- Các phép toán hỗ trợ:
  - object **top**(): trả lại phần tử nạp vào sau cùng mà không loại nó khỏi ngăn xếp
  - int **size**(): trả lại số lượng phần tử được lưu trữ
  - boolean **isEmpty**(): nhận biết có phải ngăn xếp rỗng

# Cài đặt Stack

- Hai cách thường dùng cài đặt stack:
  - Mảng
  - Danh sách liên kết
- Sử dụng cách nào còn phụ thuộc vào yêu cầu cụ thể của từng bài toán ứng dụng
  - Ưu điểm và nhược điểm của mỗi cách cài đặt?



# Stack: Cài đặt Stack dùng mảng Array

- Ta cần dùng 1 biến `numItems` để lưu trữ số phần tử có trong stack
- Nếu sử dụng mảng để cài đặt stack, vậy phần tử trên đỉnh stack nằm ở chỉ số nào của mảng là tốt ?
  - Vị trí 0 ?
  - Hay vị trí `numItems - 1` ?
- Chú ý thao tác `push` và `pop` phải thực hiện trong thời gian cỡ  $O(1)$
- Để cài đặt stack dùng mảng, ta sẽ tiến hành nạp các phần tử từ trái sang phải:
  - Phần tử đáy stack nằm ở phần tử `S[0]`
  - Phần tử đỉnh stack nằm ở phần tử `S[numItems - 1]`
  - *push* thực hiện tại `S[numItems]`
  - *pop* thực hiện tại `S[numItems - 1]`



# Stack: Cài đặt Stack dùng mảng Array

## Thao tác cơ bản:

- **void STACKinit(int);**
- **int STACKempty();**
- **void STACKpush(Item);**
- **Item STACKpop();**

```
typedef    .... Item;
static Item *s;
static int maxSize;//số lượng phần tử tối đa mà stack có thể chứa
static int numItems; //số lượng phần tử hiện tại có trong stack
void STACKinit(int maxSize)
{
    s = (Item *) malloc(maxSize*sizeof(Item));
    N = 0;
}
int STACKempty(){return numItems==0;}
int STACKfull() {return numItems==maxSize;}
```

```
void STACKpush(Item item)
{
    if (Stackfull()) ERROR("Stack đã đầy (Stack is full)");
    else
    {
        s[numItems] = item;
        numItems++;
    }
}
Item STACKpop()
{
    if (STACKempty()) ERROR("Stack không có phần tử nào (Stack is empty)")
    else
    {
        numItems--;
        return s[numItems+1];
    }
}
```

# Stack: Cài đặt Stack dùng mảng Array

- Ưu điểm
  - Dễ cài đặt
  - Thời gian tính toán tốt nhất: thao tác push và pop được thực hiện trong thời gian cố
- Nhược điểm
  - Độ dài cố định: kích thước mảng phải được khởi tạo cố định vì
    - Kích thước mảng phải được biết trước khi mảng được khởi tạo, và nó phải cố định,
    - Khi mảng đã đầy, không phần tử mới nào được chèn thêm vào mảng nữa
- Nếu kích thước tối đa của mảng không được biết trước (hoặc lớn hơn rất nhiều so với dự kiến), ta có thể sử dụng mảng động (dynamic array)
  - Đôi khi thao tác push có thể mất thời gian cỡ  $O(n)$



`maxSize`: số lượng phần tử tối đa của mảng

## Stack overflow

- Xảy ra khi thực hiện thao tác push (thêm) một phần tử vào stack đã đầy.

```
if (STACKfull())  
    STACKpush(item);
```

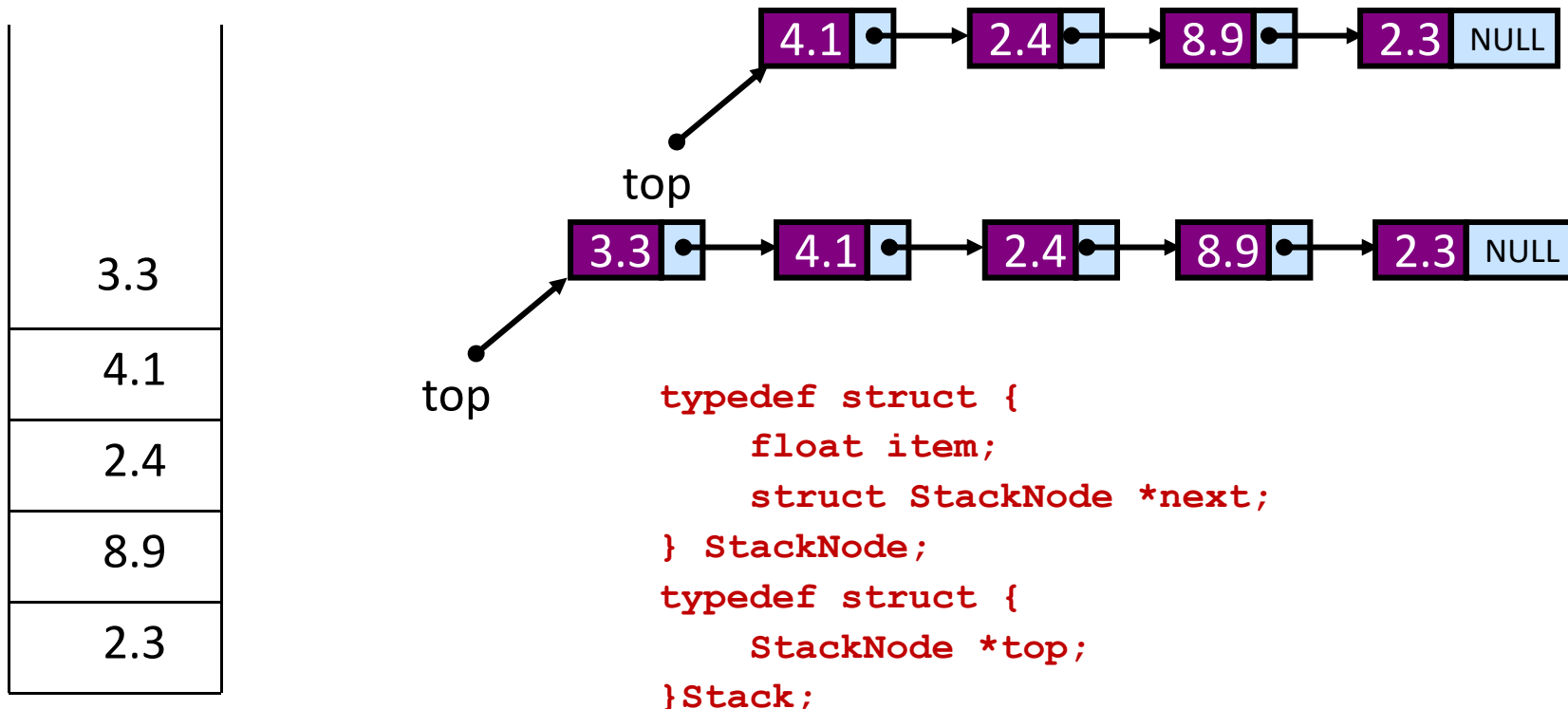
## Stack underflow

- Xảy ra khi thực hiện thao tác pop (lấy) một phần tử ra khỏi stack rỗng.

```
if (STACKempty())  
    STACKpop(item);
```

# Stack: Cài đặt Stack dùng danh sách liên kết

- Trong cách cài đặt stack dùng danh sách móc nối, stack được cài đặt như danh sách móc nối với các thao tác bổ sung và loại bỏ luôn làm việc với nút ở đầu danh sách:
  - push* thêm phần tử vào đầu danh sách
  - pop* xóa phần tử đầu danh sách
- Vị trí đỉnh stack là nút **head**, vị trí cuối stack là nút cuối



# Các thao tác

1. Khởi tạo:

```
Stack *StackConstruct();
```

2. Kiểm tra rỗng:

```
int StackEmpty(Stack* s);
```

3. Kiểm tra đầy:

```
int StackFull(Stack* s);
```

4. Thêm 1 phần tử vào stack (Push): *thêm 1 phần tử mới vào đỉnh stack*

```
int StackPush(Stack* s, float* item);
```

5. Xóa 1 phần tử khỏi stack (Pop): *xóa phần tử đang ở đỉnh stack khỏi stack và trả về phần tử này:*

```
float pop(Stack* s);
```

6. In ra màn hình tất cả các phần tử của stack

```
void Disp(Stack* s);
```

# Khởi tạo stack

```
Stack *StackConstruct() {  
    Stack *s;  
    s = (Stack *)malloc(sizeof(Stack));  
    if (s == NULL) {  
        return NULL;    // Không đủ bộ nhớ  
    }  
    s->top = NULL;  
    return s;  
}
```

```
/**** Destroy stack ****/  
void StackDestroy(Stack *s) {  
    while (!StackEmpty(s)) {  
        StackPop(s);  
    }  
    free(s);  
}
```



```
/** Check empty */
```

```
int StackEmpty(const Stack *s) {  
    return (s->top == NULL);  
}
```

```
/** Check full */
```

```
int StackFull() {  
    printf("\n KHÔNG ĐỦ BỘ NHỚ! STACK ĐÃ ĐẦY");  
    return 1;  
}
```



# Hiển thị tất cả các phần tử trong stack

```
void disp(Stack* s) {
    StackNode* node;
    int ct = 0; float m;
    printf("\n\n  Danh sách các phần tử trong stack \n\n");
    if (StackEmpty(s))
        printf("\n\n  ----- STACK RÕNG ----- \n");
    else {
        node = s->top;
        do {
            m = node->item;
            printf("%8.3f \n", m);
            node = node->next;
        } while (!(node == NULL));
    }
}
```

# Thuật toán thực hiện thao tác Push

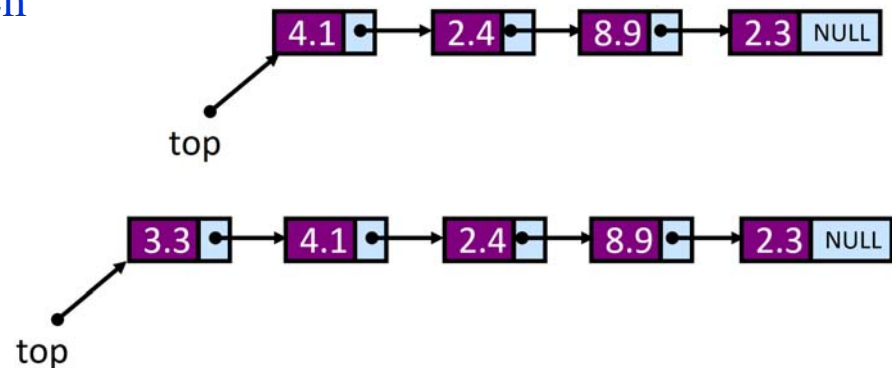
Cần thực hiện lần lượt các bước sau:

- (1) Thêm nút mới: phân bổ bộ nhớ, rồi gán dữ liệu cho nút mới
- (2) Kết nối nút mới này tới nút đầu (head) của danh sách
- (3) Gán nút mới này thành nút đầu (head) của danh sách

```
int StackPush(Stack *s, float item) {  
    StackNode *node;  
    node = (StackNode *)malloc(sizeof(StackNode)); // (1)  
    if (node == NULL) {  
        StackFull(); return 1; // overflow: out of memory (không đủ bộ nhớ)  
    }  
    node->item = item; // (1)  
    node->next = s->top; // (2)  
    s->top = node; // (3)  
    return 0;  
}
```

# Thuật toán thực hiện thao tác Pop

1. Kiểm tra xem **stack có rỗng hay không**
2. Ghi nhớ **địa chỉ** của nút đang ở đầu (top) danh sách
3. Ghi nhớ **giá trị** của nút đang ở đầu (top) danh sách
4. Cập nhật nút đầu (top) của danh sách: nút top trở tới nút kế tiếp của nó
5. **Giải phóng bộ nhớ** nút top cũ của danh sách
6. Trả về giá trị của nút top cũ của danh sách



```
float StackPop(Stack *s) {  
    float data;  
    StackNode *node;  
    if (StackEmpty(s))           // (1)  
        return NULL;           // Stack rỗng, không thực hiện được pop  
    node = s->top;                // (2)  
    data = node->item;            // (3)  
    s->top = node->next;          // (4)  
    free(node);                  // (5)  
    return data;                 // (6)  
}
```

# Toàn bộ chương trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
// tất cả các hàm StackConstruct, StackDestroy, ....đặt ở đây
int main() {
    int ch,n,i;    float m;
    Stack* stackPtr;
    while(1)
    {   printf("\n\n=====\\n");
        printf(" STACK TEST PROGRAM \\n");
        printf("=====\\n");
        printf(" 1.Create\\n 2.Push\\n 3.Pop\\n 4.Display\\n 5.Exit\\n");
        printf("-----\\n");
        printf("Nhập số để chọn chức năng tương ứng: ");
        scanf("%d",&ch); printf("\\n\\n");
```

```
switch(ch) {
    case 1:    printf("KHỞI TẠO STACK");
               stackPtr = StackConstruct(); break;
    case 2:    printf("Nhập số thực để thêm vào stack: ");
               scanf("%f", &m);
               StackPush(stackPtr, m); break;
    case 3:    m=StackPop(stackPtr);
               if (m != NULL)
                   printf("\n Giá trị của số đẩy ra khỏi stack: %8.3f\n", m);
               else {
                   printf("\n Stack rỗng, không thực hiện được thao tác pop \n");
               }
               break;
    case 4:    disp(stackPtr); break;
    case 5:    printf("\n Kết thúc! \n\n");
               exit(0);    break;
    default:   printf("Bấm sai nút số");
} //switch
} // end while
} //end main
```

# Stack: Cài đặt Stack dùng danh sách liên kết

- Ưu điểm:
  - Thời gian thực hiện thao tác push và pop 1 phần tử từ stack luôn là hằng số
  - Kích thước của stack có thể lên rất lớn (phụ thuộc vào bộ nhớ máy)
- Nhược điểm
  - Khó cài đặt

# Các ứng dụng của ngăn xếp

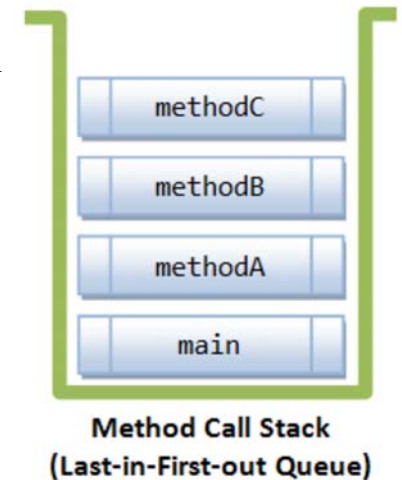
- **Ứng dụng trực tiếp**
  - Lịch sử duyệt trang trong trình duyệt Web
  - Dãy Undo trong bộ soạn thảo văn bản
  - Chuỗi các hàm được gọi (Chain of method calls)
  - Kiểm tra tính hợp lệ của các dấu ngoặc trong biểu thức
  - Đổi cơ số
  - Ứng dụng trong cài đặt chương trình dịch (Compiler implementation)
  - Tính giá trị biểu thức (Evaluation of expressions)
  - Quay lui (Backtracking)
  - Khử đệ qui
  - ...
- **Các ứng dụng khác (Indirect applications)**
  - Cấu trúc dữ liệu hỗ trợ cho các thuật toán
  - Thành phần của các cấu trúc dữ liệu khác

# Các ứng dụng của ngăn xếp: Ví dụ 1

- **Chuỗi các hàm được gọi:** hầu hết các ngôn ngữ lập trình đều sử dụng “call stack” để thực thi việc gọi hàm
  - Khi chương trình gọi một hàm nào đó để thực thi, chỉ số dòng lệnh và các thông tin hữu ích liên quan đến hàm này sẽ được thêm vào “call stack”.
  - Khi hàm kết thúc, nó sẽ được loại bỏ khỏi “call stack” và chương trình tiếp tục thực thi dòng lệnh được chỉ ra tại hàm tiếp theo đang nằm ở đỉnh stack.

```
void methodC()  
{  
    printf("Enter methodC ");  
}  
void methodB()  
{  
    methodC();  
}  
void methodA()  
{  
    methodB();  
}  
void main() {  
    methodA();  
}
```

1. Hàm main được đẩy vào “call stack”, chương trình thực hiện hàm main
2. methodA được đẩy vào “call stack”, chương trình thực hiện methodA
3. methodB được đẩy vào “call stack”, chương trình thực hiện methodB
4. methodC được đẩy vào “call stack”, chương trình thực hiện methodC; khi thực hiện xong, methodC sẽ bị lấy ra khỏi stack
5. methodB thực hiện xong và được lấy ra khỏi stack.
6. methodA thực hiện xong và được lấy ra khỏi stack.
7. main thực hiện xong và được lấy ra khỏi stack. Chương trình kết thúc.





# Các ứng dụng của ngăn xếp: Ví dụ 2

- Chương trình dịch (compilers) kiểm tra lỗi cú pháp
  - Kiểm tra các dấu ngoặc “(”, “{”, hoặc “[” có được đi cặp với “)”, “}”, hoặc “]” hay không

Ví dụ:

- Đúng : `()(()) {[()]}`
- Đúng : `((())(())[()])`
- Sai: `)(()) {[()]}`
- Sai: `( {[ ]})`
- Sai: `(`

Checking for balanced parentheses is one of the most important task of a compiler.

```
int main(){  
    for ( int i=0; i < 10; i++)  
    {  
        //some code  
    }  
}  
} ← Compiler generates error
```

# Các ứng dụng của ngăn xếp: Ví dụ 2

**Kiểm tra dấu ngoặc hợp lệ trong 1 biểu thức (Parentheses Matching):** Cho 1 biểu thức dưới dạng một dãy các kí tự, viết chương trình kiểm tra xem cặp các dấu ngoặc “{“,”}”,“(“,”)”, “[“,”]” và thứ tự của chúng trong biểu thức đã cho có hợp lệ hay không.

Ví dụ 1: cho biểu thức = [O]{} {[OO]O} → chương trình trả về true;

Ví dụ 2: cho biểu thức = [(]) → chương trình trả về false

## Thuật toán:

- 1) Khai báo stack S chứa các kí tự.
- 2) Duyệt lần lượt từng kí tự của biểu thức, từ trái sang phải:
  - a) Nếu kí tự hiện tại là dấu mở ngoặc (‘(‘ hoặc ‘{‘ hoặc ‘[‘) thì push nó vào stack.
  - b) Nếu kí tự hiện tại là dấu đóng ngoặc (‘)’ hoặc ‘}’ hoặc ‘]’) thì thực hiện thao tác pop để lấy kí tự ở đỉnh stack ra, nếu kí tự lấy ra này là dấu mở ngoặc và là cặp với kí tự hiện tại thì thuật toán tiếp tục, nếu không ta khẳng định dấu ngoặc trong biểu thức đã cho không hợp lệ và thuật toán kết thúc
- 3) Nếu đã duyệt qua tất cả các kí tự trong biểu thức, vẫn còn dấu mở ngoặc ở trong stack, ta kết luận “dấu ngoặc trong biểu thức không hợp lệ”

## Ví dụ 2: Parentheses Matching

**Algorithm** ParenMatch( $X, n$ ):

**Input:** Mảng  $X$  gồm  $n$  kí tự, mỗi kí tự hoặc là dấu ngoặc, hoặc là biến, hoặc là phép toán số hoặc, hoặc là con số.

**Output:** **true** khi và chỉ khi các dấu ngoặc trong  $X$  là có đôi

$S =$  ngăn xếp rỗng;

**for**  $i=0$  to  $n-1$  **do**

**if** ( $X[i]$  là kí tự mở ngoặc)

        push( $S, X[i]$ ); // gặp dấu mở ngoặc ('(' hoặc '{' hoặc '[') thì đưa vào stack

**else**

**if** ( $X[i]$  là kí tự đóng ngoặc) // so sánh  $X[i]$  với kí tự đang ở đầu stack

**if** isEmpty( $S$ )

**return false** // Không tìm được cặp đôi

**if** ( pop( $S$ ) không là cặp với dấu ngoặc trong  $X[i]$  )

**return false** //Lỗi kiểu dấu ngoặc

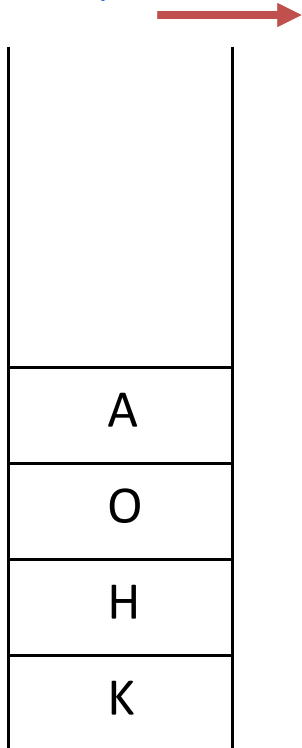
**if** isEmpty( $S$ )

**return true** //mỗi dấu ngoặc đều có cặp

**else return false** //có dấu ngoặc không tìm được cặp

# Các ứng dụng của ngăn xếp: Ví dụ 3

- Đảo ngược từ cho trước: ta có thể sử dụng 1 stack để đảo thứ tự các chữ cái trong 1 từ cho trước.
- Làm thế nào?
- Ví dụ: KHOA



Push(K)


Push(H)

Push(O)

Push(A)

- Đọc lần lượt từng chữ cái và push (đẩy) vào stack

## Ví dụ 3: Đảo ngược từ

- Đảo ngược từ cho trước: Ta có thể sử dụng 1 stack để đảo thứ tự các chữ cái trong 1 từ cho trước.
- Làm thế nào?
- Ví dụ: KHOA 

A
O
H
K

Push(K)	Pop(A)
Push(H)	Pop(O)
Push(O)	Pop(H)
Push(A)	Pop(K)

A O H K

- Đọc lần lượt từng chữ cái và push (đẩy) vào stack
- Khi các chữ cái của từ đều đã được đưa vào stack, tiếp theo ta lại lần lượt pop (lấy ra) các chữ cái khỏi stack, và in chúng ra

# Các ứng dụng của ngăn xếp:

## Ví dụ 4: HTML Tag Matching

- Trong HTML, mỗi thẻ `<name>` phải đi cặp đôi với thẻ `</name>`

`<body>`

`<center>`

`<h1> The Little Boat </h1>`

`</center>`

`<p> The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage. </p>`

`<ol>`

`<li> Will the salesman die? </li>`

`<li> What color is the boat? </li>`

`<li> And what about Naomi? </li>`

`</ol>`

`</body>`

### The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

- Thuật toán hoàn toàn tương tự như thuật toán giải bài toán ngoặc hợp lệ
- Bài tập: Hãy thiết kế và cài đặt chương trình giải bài toán đặt ra!

## Các ứng dụng của ngăn xếp:

### Ví dụ 5: Tính giá trị biểu thức

Ví dụ: Tính giá trị biểu thức :  $(4/(2-2+3))*(3-4)*2)$

Chú ý: thứ tự ưu tiên các phép toán từ cao đến thấp:

1. Phép mũ ^
2. Phép nhân chia \* /
3. Phép cộng trừ + -

Thuật toán tính giá trị biểu thức: **Sử dụng 2 stack:**

- Stack S1 lưu trữ các toán hạng
- Stack S2 lưu trữ các phép toán (+, -, \*, /, ^) và dấu mở ngoặc (

#### **Thủ tục Process:**

- Thực hiện Pop(S1) 2 lần: Đẩy 2 giá trị đang ở đầu stack S1 ra khỏi S1, gọi hai giá trị đó lần lượt là A và B
- Thực hiện Pop(S2): Đẩy phép toán đang ở đầu stack S2 ra khỏi S2, gọi phép toán đó là OPT.
- Thực hiện  $A \text{ OPT } B$ , thu được kết quả kí hiệu là R
- Rồi đẩy kết quả thu được R vào S1: tức là thực hiện Push(S1, R)

# Thuật toán: tính giá trị biểu thức sử dụng 2 stack

Duyệt biểu thức từ trái qua phải:

- Nếu gặp *toán hạng (Operands)*: đưa nó vào stack S1.
- Nếu gặp *phép toán (Operator)*, kí hiệu là *OPT*:
  - Nếu stack S2 đang rỗng: đưa phép toán OPT vào stack S2, tức là thực hiện Push(S2, OPT)
  - Else Nếu phép toán OPT có độ ưu tiên lớn hơn hoặc bằng độ ưu tiên của top(S2) thì đưa OPT vào stack S2, tức là thực hiện Push(S2, OPT)
  - Else Nếu phép toán OPT có độ ưu tiên ít hơn độ ưu tiên của top(S2) thì
    - do Process .... Until phép toán OPT ưu tiên  $\geq$  độ ưu tiên của top(S2) Hoặc stack toán hạng S1 rỗng
- Nếu gặp *dấu mở ngoặc*: thì nạp nó vào stack S2.
- Nếu gặp *dấu đóng ngoặc*: thì
  - do Process ... Until tìm thấy dấu mở ngoặc đầu tiên trên S2
  - Lấy dấu mở ngoặc đó ra khỏi S2
- *Khi duyệt hết biểu thức và Stack S1 vẫn chưa rỗng*:
  - Do Process ... Until Stack S1 rỗng



Ví dụ: Tính giá trị biểu thức  $2 * ( 5 * ( 3 + 6 ) ) / 15 - 2$

Kí tự	Thực hiện	Stack toán hạng S1	Stack phép toán S2	Giải thích
2	Push 2 vào S1	2	RỖNG	
*	Push * vào S2	2	*	
(	Push ( vào S2	2	( *	
5	Push 5 vào S1	5 2	( *	
*	Push * vào S2	5 2	* ( *	
(	Push ( vào S2	5 2	( * ( *	
3	Push 3 vào S1	3 5 2	( * ( *	
+	Push + vào S2	3 5 2	+ ( * ( *	
6	Push 6 vào S1	6 3 5 2	+ ( * ( *	
)	Gọi Process:			Thực hiện Process cho đến khi tìm thấy dấu mở ngoặc ( đầu tiên trong S2
	Pop 6 và 3 khỏi S1	5 2	+ ( * ( *	
	Pop + khỏi S2	5 2	( * ( *	
	Thực hiện 3+6=9	5 2	( * ( *	
	Push 9 vào S1	9 5 2	( * ( *	
	Pop ( khỏi S2	9 5 2	* ( *	

Ví dụ: Tính giá trị biểu thức  $2 * ( 5 * ( 3 + 6 ) ) / 15 - 2$

Kí tự	Thực hiện	Stack toán hạng S1	Stack phép toán S2	Giải thích
)	Gọi Process:			Thực hiện Process cho đến khi tìm thấy dấu mở ngoặc ( đầu tiên trong S2
	Pop 9 và 5 khỏi S1	2	* ( *	
	Pop * ra khỏi S2	2	( *	
	Thực hiện $5*9=45$	2	( *	
	Push 45 vào S1	45 2	( *	
	Pop ( khỏi S2		*	
/	Push / vào S2	45 2	/ *	/ và * có cùng thứ tự ưu tiên
15	Push 15 vào S1	15 45 2	/ *	

Ví dụ: Tính giá trị biểu thức  $2 * (5 * (3 + 6)) / 15 - 2$

Kí tự	Thực hiện	Stack toán hạng S1	Stack phép toán S2	Giải thích
-	Gọi Process		/ *	- có thứ tự ưu tiên nhỏ hơn /, do đó thực hiện Process
	Pop 15 và 45 khỏi S1	2		
	Pop / ra khỏi S2	2	*	
	Thực hiện $45/15=3$	2	*	
	Push 3 vào S1	3 2	*	
	Gọi Process			- có thứ tự ưu tiên nhỏ hơn *, do đó thực hiện Process
	Pop 3 và 2 khỏi S1	RỖNG	*	
	Pop * khỏi S2	RỖNG	RỖNG	
	Thực hiện $2*3=6$			
	Push 6 vào S1, push - vào S2	6	-	S1 đã rỗng
2	Push 2 vào S1	2 6	-	
	Gọi Process			Đã duyệt hết các kí tự trong biểu thức mà S1 vẫn chưa rỗng, do đó gọi Process cho đến khi S1 rỗng
	Pop 2 và 6 khỏi S1	RỖNG	-	
	Pop - khỏi S2	RỖNG	RỖNG	
	Thực hiện $6-2=4$			
	Push 4 vào S1	4		

Kết quả biểu thức = 4

Bài tập: Tính giá trị:  $(4/(2-2+3))*(3-4)*2$

Kí tự	Thực hiện	Stack toán hạng S1	Stack phép toán S2	Giải thích
(	Push ( vào S2		(	

## Các ứng dụng của ngăn xếp:

### Ví dụ 6: Đổi cơ số

**Bài toán:** Viết một số trong hệ đếm thập phân thành số trong hệ đếm cơ số  $b$ .

**Ví dụ:**

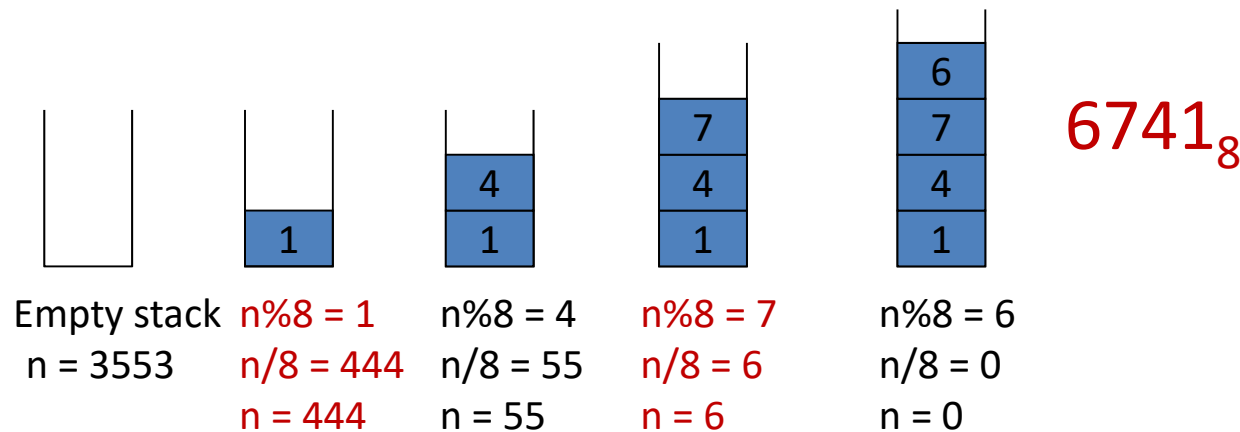
$$(\text{cơ số } 8) \quad 28_{10} = 3 \cdot 8 + 4 = 34_8$$

$$(\text{cơ số } 4) \quad 72_{10} = 1 \cdot 64 + 0 \cdot 16 + 2 \cdot 4 + 0 = 1020_4$$

$$(\text{cơ số } 2) \quad 53_{10} = 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 = 110101_2$$

# Thuật toán dùng ngăn xếp

- **Input** số trong hệ đếm thập phân  $n$
- **Output** số trong hệ đếm cơ số  $b$  tương ứng
- **Ví dụ:**

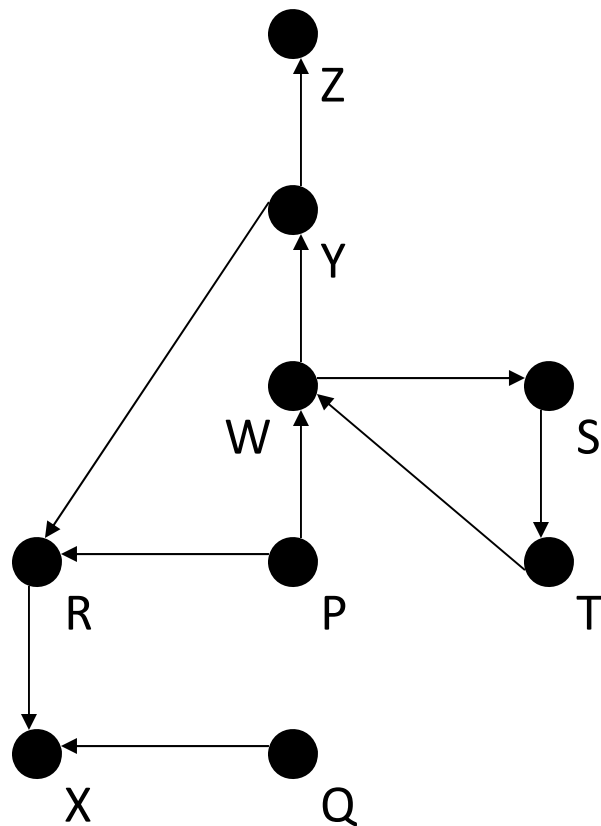


1.  $A = n \% b$ . Push A vào stack.
2. Thay  $n$  bởi  $n / b$  (để tiếp tục xác định các chữ số còn lại).
3. Lặp lại các bước 1-2 đến khi còn số 0 ( $n/b = 0$ ).
4. Đẩy các ký tự ra khỏi ngăn xếp và in chúng.

# Các ứng dụng của ngăn xếp:

## Ví dụ 7: Tìm đường đi

Cho đồ thị mô tả các chuyến bay như sau



● : thành phố



Chuyến bay từ thành phố W tới thành phố S

# Thuật toán tìm đường đi bằng cách dùng ngăn xếp

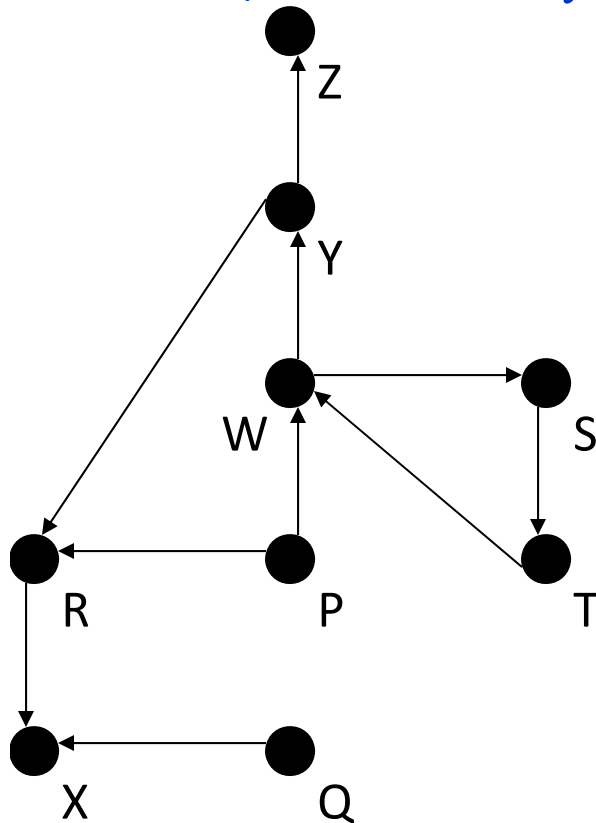
- Nếu tồn tại, ta có thể tìm đường đi từ thành phố  $C_1$  tới thành phố  $C_2$  bằng 1 stack
  - Đẩy thành phố xuất phát vào stack
    - Đánh dấu thành phố đó là đã đến
    - Chọn một thành phố  $K$  bất kì có chuyến bay đến nó từ thành phố này (trên đồ thị có mũi tên chỉ từ thành phố xuất phát này đến thành phố  $K$  ta chọn)
      - Thành phố  $K$  vừa chọn này phải chưa được đánh dấu là đã đến
    - Đẩy thành phố  $K$  này vào stack
      - Đánh dấu thành phố  $K$  thành đã đến
    - Nếu thành phố  $K$  là thành phố  $C_2$ , thuật toán kết thúc
      - Nếu không, chọn một thành phố  $K'$  bất kì có chuyến bay đến nó từ thành phố  $K$ 
        - » Thành phố  $K'$  phải chưa bao giờ được đánh dấu là đã đến; thuật toán bắt đầu từ thành phố  $K'$
        - » Nếu không tìm được thành phố  $K'$  nào như vậy, thực hiện thao tác POP trên stack, để đẩy thành phố  $K$  ra khỏi stack; thuật toán bắt đầu lại từ thành phố ngay trước  $K$
  - Lặp lại cho đến khi đạt được thành phố  $C_2$  hoặc tất cả các thành phố đều đã đến



# Các ứng dụng của ngăn xếp:

## Ví dụ 7: Tìm đường đi

Cho đồ thị mô tả các chuyến bay như sau



● : thành phố

● → ● Chuyển bay từ thành phố W tới thành phố S  
W S

- Cần đi từ P tới Y
  - Đẩy P vào stack và đánh dấu nó thành đã đến
  - Chọn R là thành phố tiếp theo từ P (chọn ngẫu nhiên trong số các thành phố đến được từ P là: W, R)
    - Đẩy R vào stack và đánh dấu nó thành đã đến
  - Chọn X là thành phố tiếp theo đến được từ P (lựa chọn duy nhất)
    - Đẩy X vào Stack, và đánh dấu nó thành đã đến
  - Từ X, không đến được thành phố nào khác: thực hiện thao tác POP: lấy X ra khỏi stack; thuật toán bắt đầu lại từ thành phố ngay trước X (tức là thành phố R)
  - Từ R không đến được thành phố nào khác: thực hiện POP: lấy R ra khỏi stack; thuật toán bắt đầu lại từ thành phố ngay trước R (tức là thành phố W)
  - Chọn W là thành phố tiếp theo đến được từ P (lựa chọn duy nhất)
    - Đẩy W vào stack và đánh dấu nó thành đã đến
  - Chọn Y là thành phố tiếp theo đến được từ W (chọn ngẫu nhiên trong số các thành phố đến được từ W là: Y và S)
    - Y là thành phố đích => thuật toán kết thúc

Kết luận: đường đi từ P đến Y tìm được là:

# Nội dung

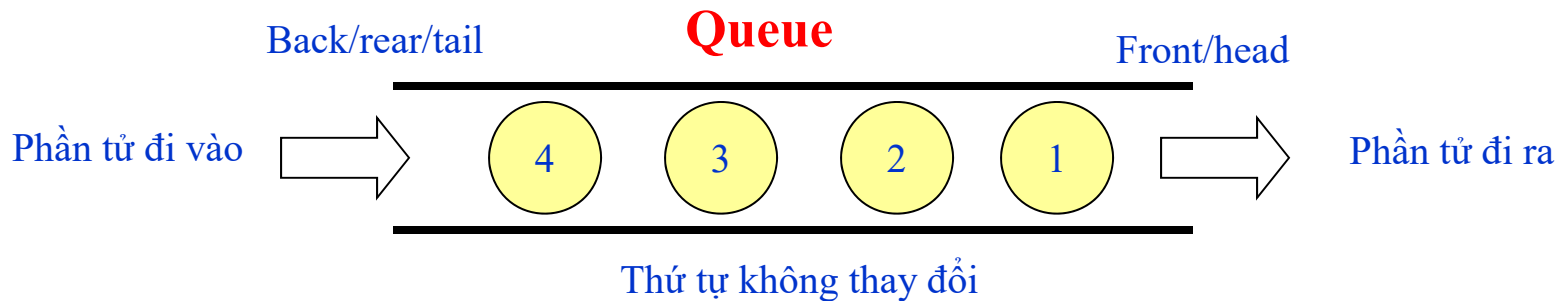
1. Mảng (Array)
2. Bản ghi (Record)
3. Danh sách liên kết (Linked List)
4. Ngăn xếp (Stack)
- 5. Hàng đợi (Queue)**

# What is a Queue?



## 5. Hàng đợi (Queue)

- Hàng đợi: Là danh sách có thứ tự trong đó phép toán chen luôn thực hiện chỉ ở một phía gọi là **phía sau** (**back** hoặc **rear** hoặc **tail**), còn phép toán xóa chỉ thực hiện ở phía còn lại gọi là **phía trước** hay **đầu** (**front** hoặc **head**).



Ví dụ: Hàng đợi thanh toán tại siêu thị

- Các thao tác thực hiện trên hàng đợi queue:
  - **Enqueue** (đưa vào) – Thêm 1 phần tử vào queue (còn có thể gọi là insert)
  - **Dequeue** (đưa ra) – Xóa 1 phần tử khỏi queue (còn có thể gọi là getFront)
- Các phần tử được lấy ra khỏi hàng đợi theo qui tắc Vào trước - Ra trước. Vì thế hàng đợi còn được gọi là danh sách vào trước ra trước (**First-In-First-Out (FIFO) list**).

## 5. Hàng đợi (Queue)

- Các thuật ngữ liên quan đến hàng đợi được mô tả trong hình vẽ sau đây:



- Hàng đợi có tính chất như là các hàng đợi chờ được phục vụ trong thực tế.

# Các thuộc tính của hàng đợi Queue

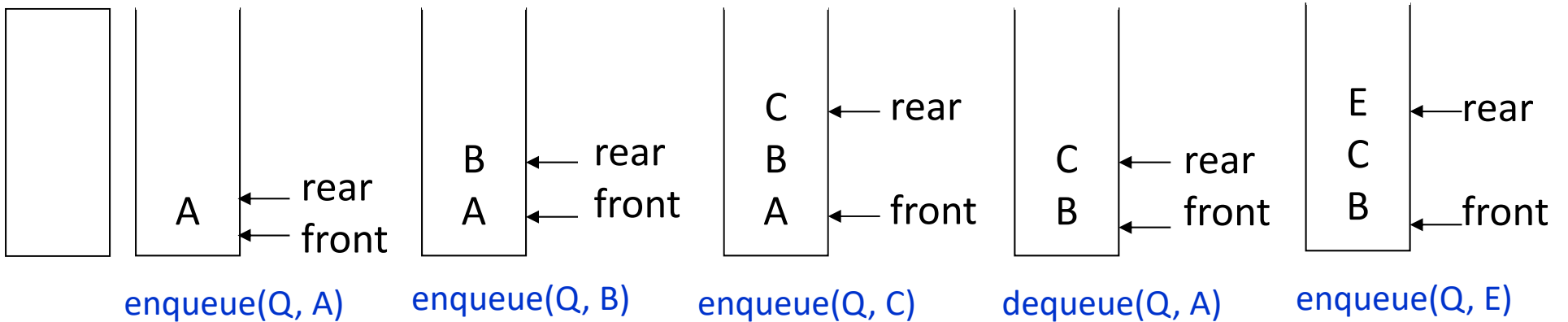
## Định nghĩa:

- *maxSize*: số lượng phần tử tối đa mà queue có thể chứa
- *ItemType*: kiểu dữ liệu của các phần tử thuộc queue

## Các phép toán:

- `Q = init()` ; khởi tạo Q là hàng đợi rỗng
- `isEmpty(Q)` ; hàm trả về giá trị “true” nếu hàng đợi Q là rỗng
- `isFull(Q)` ; hàm trả về giá trị “true” nếu hàng đợi Q đã đầy, cho ta biết là đã sử dụng tối đa bộ nhớ dành cho Q; nếu không hàm trả về giá trị “false”
- `frontQ(Q)` ; hàm trả về phần tử ở phía trước (front/head) của hàng đợi Q hoặc trả về lỗi nếu hàng đợi Q đang rỗng (không chứa phần tử nào).
- `enqueue(Q, x)` ; hàm thực hiện thêm phần tử x vào phía sau (back/rear) của hàng đợi Q. Nếu trước khi thêm, hàng đợi Q đã đầy, thì cần đưa ra thông báo là: Không thể thêm vì Q đã đầy.
- `x = dequeue(Q)` ; hàm xóa phần tử ở phía trước (front/head) hàng đợi, trả lại x là dữ liệu chứa trong phần tử này. Nếu hàng đợi rỗng trước khi thực hiện dequeue, thì cần đưa ra thông báo lỗi.
- `print(Q)` ; hàm đưa ra danh sách tất cả các phần tử của hàng đợi Q theo thứ tự từ phía trước ra phía sau.
- `sizeQ(Q)` ; trả về số lượng phần tử đang có trong hàng đợi Q.

# FIFO





# Queues everywhere!!!!





# Queues

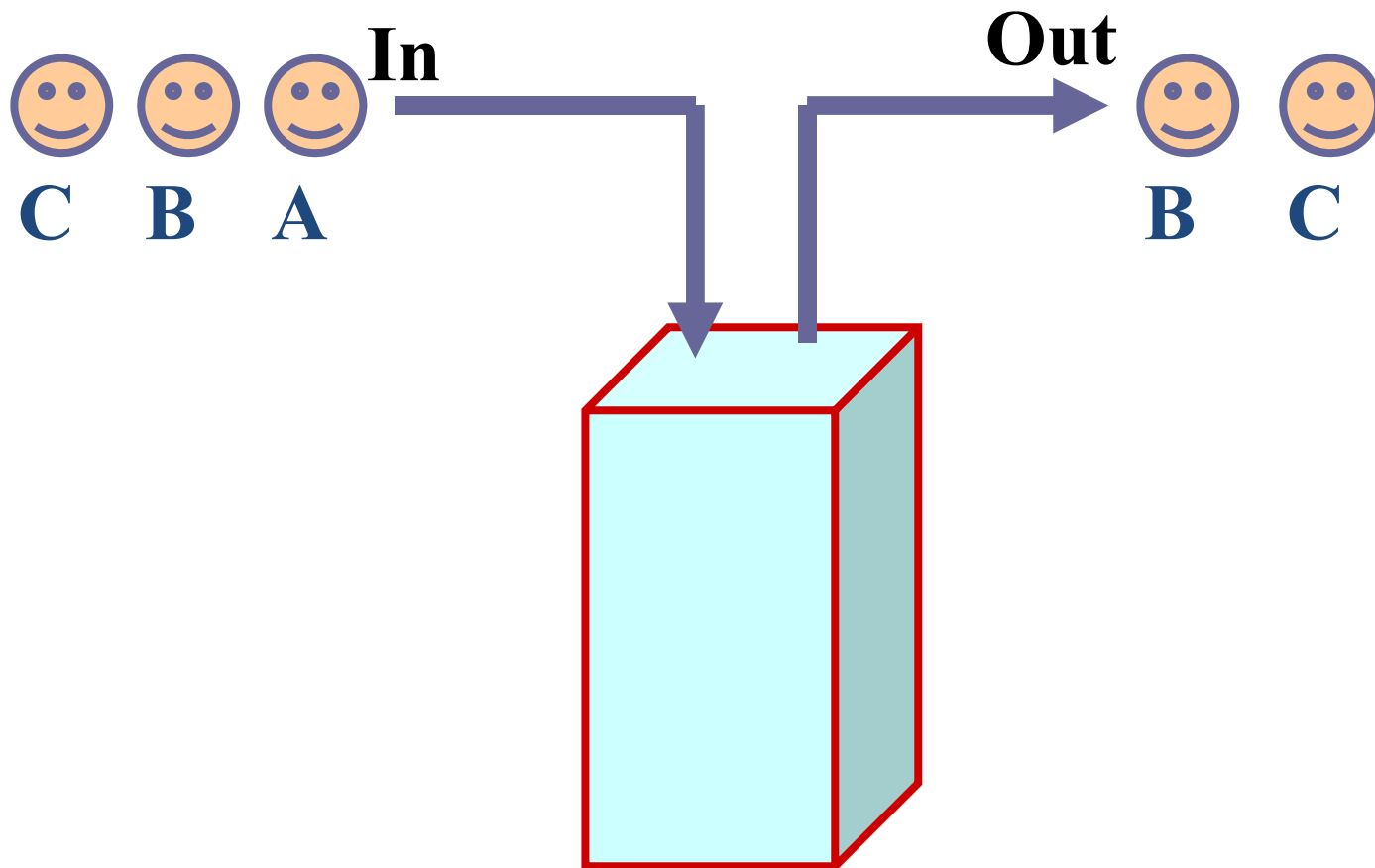
- What are some applications of queues?
  - Round-robin scheduling in processors
  - Input/Output processing
  - Queueing of packets for delivery in networks

Cho dãy các thao tác trên hàng đợi Q như sau. Hãy xác định đầu ra (output) và dữ liệu trên Q sau mỗi thao tác

	Thao tác	Output	Queue Q
1	enqueue (5)	-	(5)
2	enqueue (Q, 3)	-	(5, 3)
3	dequeue (Q)	5	(3)
4	enqueue (Q, 7)	-	(3, 7)
5	dequeue (Q)	3	(7)
6	front (Q)	7	(7)
7	dequeue (Q)	7	( )
8	dequeue (Q)	error	( )
9	isEmpty (Q)	true	( )
10	size (Q)	0	( )
11	enqueue (Q, 9)	-	(9)
12	enqueue (Q, 7)	-	(9, 7)
13	enqueue (Q, 3)	-	(9, 7, 3)
14	enqueue (Q, 5)	-	(9, 7, 3, 5)
15	dequeue (Q)	9	(7, 3, 5)

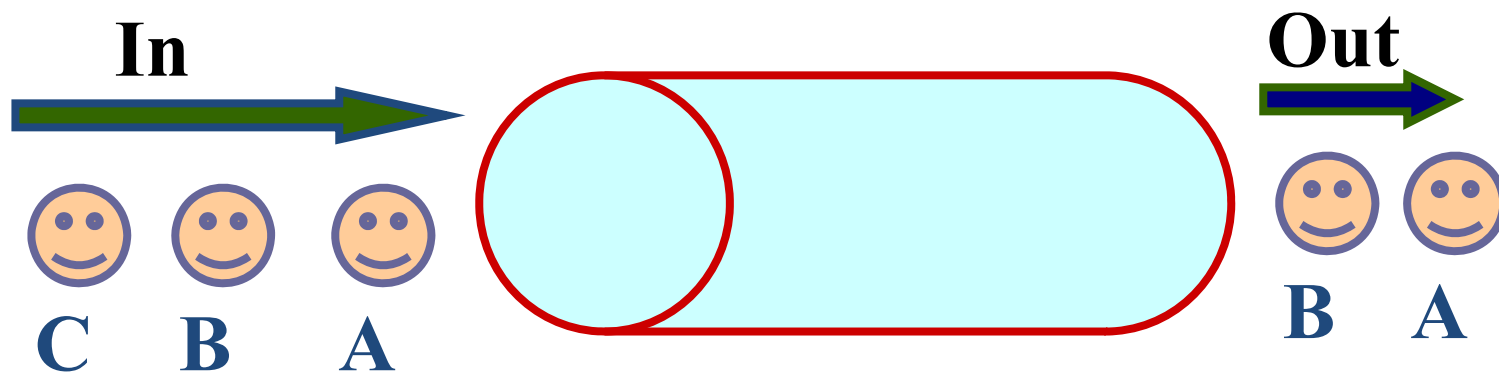
# Stack

Cấu trúc dữ liệu với thuộc tính vào sau ra trước  
**Last-In First-Out (LIFO)**



# Queue

Cấu trúc dữ liệu với thuộc tính vào trước-ra trước  
First-In First-Out (FIFO)

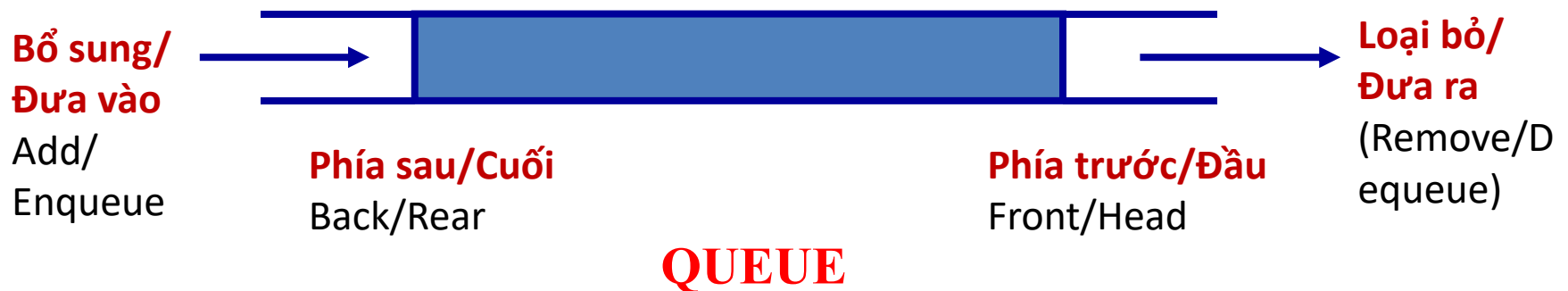


# Cài đặt hàng đợi Queue

- Cũng tương tự stack, ta có thể cài đặt Queue bằng 2 cách:
  - Sử dụng mảng (array)
  - Sử dụng danh sách liên kết (linked list)

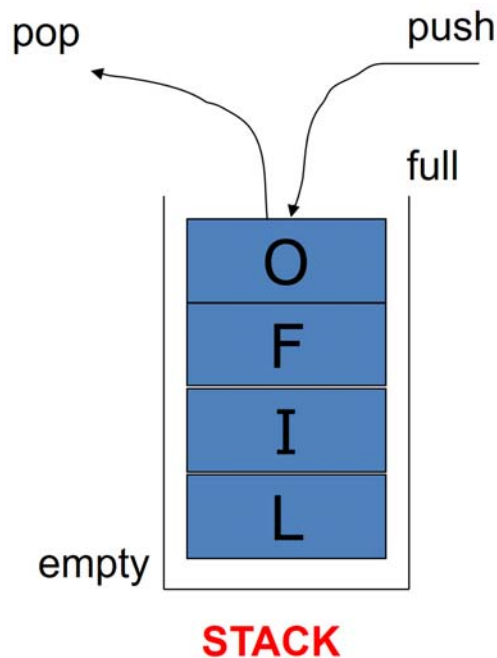
# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Dùng mảng để cài đặt hàng đợi khó hơn rất nhiều so với khi cài đặt stack. Vì sao?
  - Ngăn xếp stack: thêm và xóa phần tử đều thực hiện ở cùng 1 đầu,
  - Hàng đợi queue: thêm thực hiện ở 1 đầu, và xóa thực hiện ở đầu còn lại



# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Dùng mảng để cài đặt hàng đợi khó hơn rất nhiều so với khi cài đặt stack. Vì sao?
  - Ngăn xếp stack: thêm (push) và xóa (pop) phần tử đều thực hiện ở cùng 1 đầu,
  - Hàng đợi queue: thêm thực hiện ở 1 đầu, và xóa thực hiện ở đầu còn lại



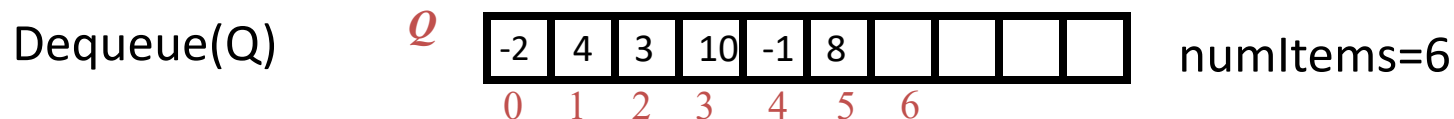
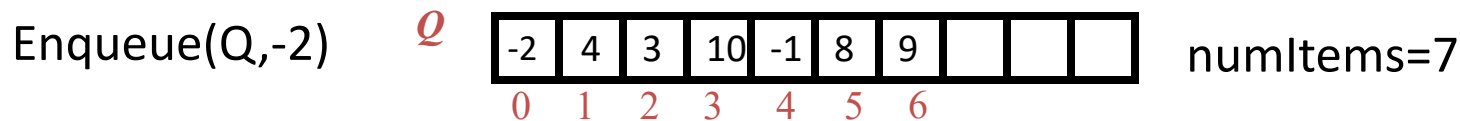
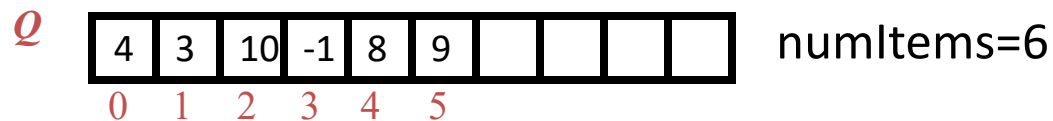
# Cài đặt hàng đợi Queue: dùng mảng (Array)

Ta có thể sử dụng 1 trong 3 cách sau để cài đặt queue:

- Cách 1:
  - Thêm (Enqueue) 1 phần tử mới vào  $Q[0]$ , và dịch chuyển tất cả các phần tử còn lại sang trái 1 vị trí để lấy chỗ cho phần tử mới thêm.
  - Xóa (Dequeue) 1 phần tử : xóa phần tử  $Q[\text{numItems}-1]$



Ví dụ:





# Cài đặt hàng đợi Queue: dùng mảng (Array)

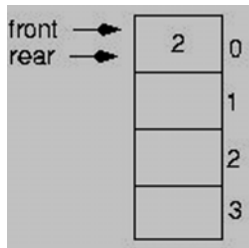
- Cách 2: hàng đợi Q vòng tròn, [quấn quanh]  
(circular queue ["wrap around"])

Ví dụ 1: Mảng có maxSize = 4;

Khởi tạo: Q rỗng: front = 0; rear = -1;

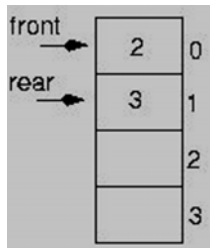
- Q[front]: phần tử đầu tiên của Q
- Q[rear]: phần tử cuối cùng của Q
- Thêm phần tử vào Q (Enqueue): rear+=1; Q[rear]=item;
- Xóa phần tử khỏi Q (Dequeue): xóa Q[front]; sau đó front+=1

enqueue(Q,2)



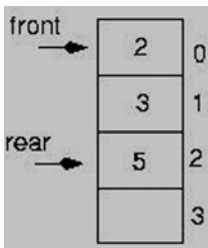
rear ++;  
Q[rear] = 2;  
Q = (2)

enqueue(Q,3)



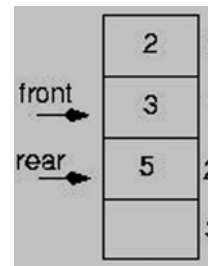
rear++;  
Q[rear] = 3;  
Q = (2, 3)

enqueue(Q,5)



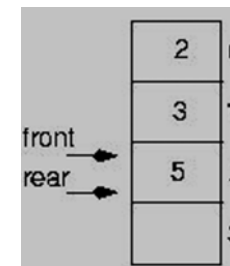
rear++;  
Q[rear] = 5;  
Q = (2, 3, 5)

dequeue(Q)



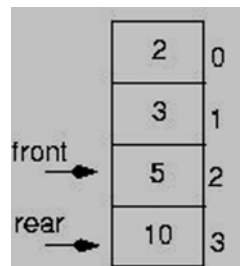
Dequeue Q[front]  
Q = (3, 5)  
front++;

dequeue(Q)



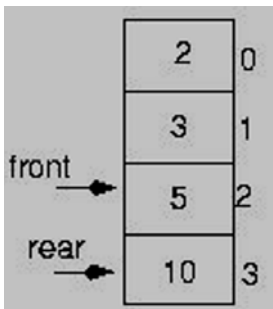
Dequeue Q[front]  
Q = (5)  
front++;

enqueue(Q,10)



rear++;  
Q[rear] = 10;  
Q = (5,10)

enqueue(Q,20) ???

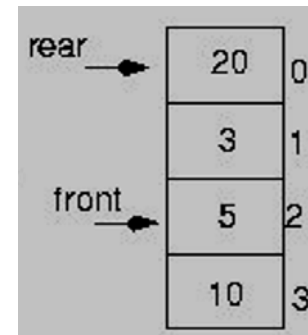


rear ++;  
Q[rear] = 20;  
→ rear = 4 = maxSize  
→ Mảng Q tràn (over flow)

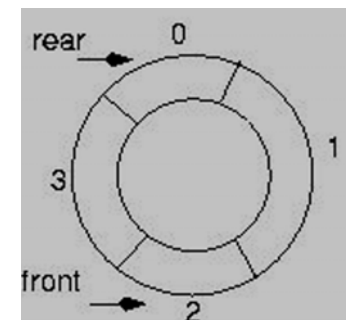
Thực hiện quấn quanh  
"wrap around"

If (rear == maxSize -1)  
    rear = 0;  
else  
    rear = rear +1;

Or  
rear = (rear + 1) % maxSize;



Q = (5,10, 20)



Q vòng tròn  
(Circular queue)

# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn, [quấn quanh]

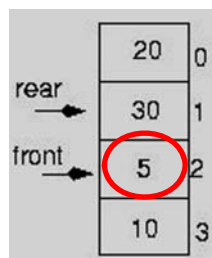
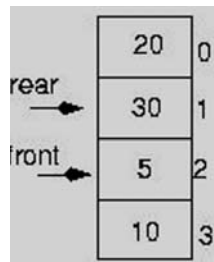
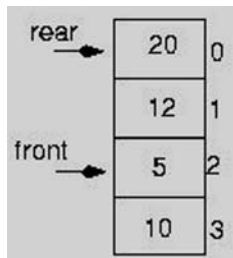
Ví dụ 2: Mảng có kích thước maxSize = 4

Q gồm 3 phần tử:  $Q = (5, 10, 20)$

- $Q[\text{front}]$ : phần tử đầu tiên của Q
- $Q[\text{rear}]$ : phần tử cuối cùng của Q
- Thêm phần tử vào Q (Enqueue):  $\text{rear} += 1; Q[\text{rear}] = \text{item};$
- Xóa phần tử khỏi Q (Dequeue): xóa  $Q[\text{front}]$ ; sau đó  $\text{front} += 1$

enqueue(Q, 30)

enqueue(Q, 50) ?? Thêm 50 vào Q khi Q đã đầy

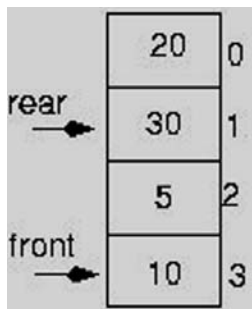


Q đã đầy!!!  
Làm thế nào để kiểm tra được Q đã đầy ?

$\text{rear} + 1 == \text{front}$

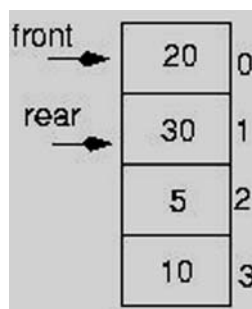
Nếu vậy ta không phân biệt được 2 trường hợp: hàng đợi Q RỖNG VÀ ĐẦY!!

dequeue(Q)



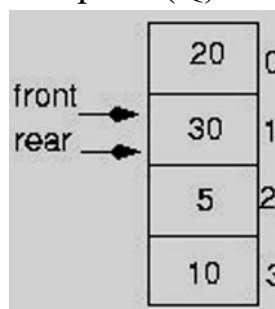
Dequeue  $Q[\text{front}]$   
 $Q = (10, 20, 30)$   
 $\text{front}++;$

dequeue(Q)



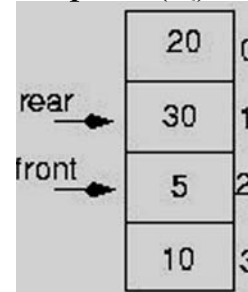
Dequeue  $Q[\text{front}]$   
 $Q = (20, 30)$   
 $\text{front}++;$   
 $\Rightarrow \text{front} = 4 = \text{maxSize}$   
 $\Rightarrow$  "wrap around"  
 $\Rightarrow \text{front} = 0$

dequeue(Q)



Dequeue  $Q[\text{front}]$   
 $Q = (30)$   
 $\text{front}++;$

dequeue(Q)



Dequeue  $Q[\text{front}]$   
 $Q = \text{empty}$   
 $\text{front}++;$

Q đã rỗng!!!  
Làm thế nào để kiểm tra được Q rỗng ?

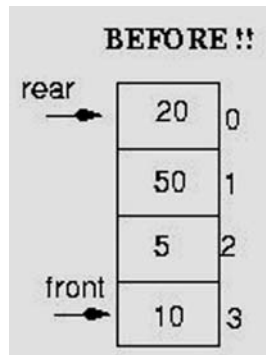
$\text{rear} + 1 == \text{front}$

# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn,[quấn quanh]

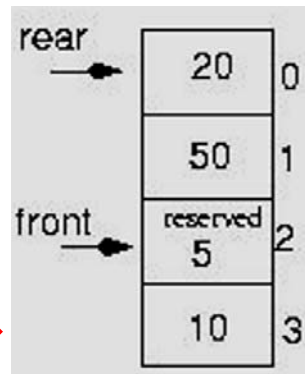
Giải pháp 1: Dùng biến *front* trở tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ).

# Ví dụ 3: mô phỏng giải pháp 1



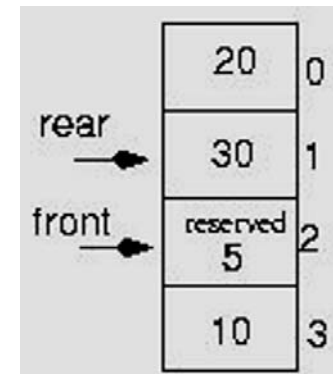
Q = (10, 20)

Giải pháp 1: Dùng biến *front* trở tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ lãng phí không được dùng tới)



Q = (10, 20)

enqueue(Q, 30)

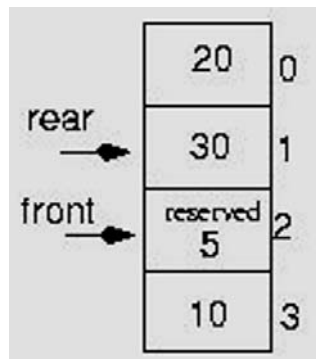


rear++; Q[rear] = 30;  
Q = (10, 20, 30)

Q đã đầy!!!  
Làm thế nào để kiểm tra Q đã đầy ?

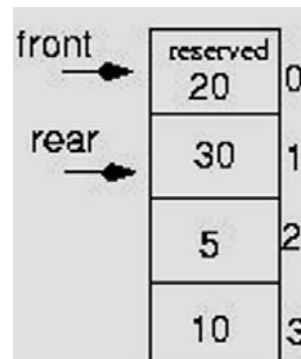
**rear + 1 == front**

dequeue(Q)



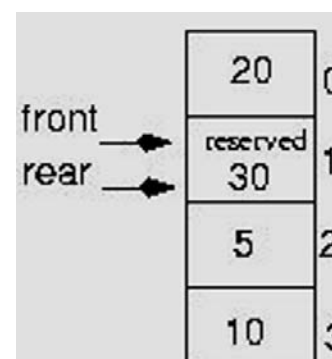
Q = (10, 20, 30)

dequeue(Q)



front++;  
⇒ front = 4 = maxSize  
⇒ wrap around: front = 0  
Dequeue Q[front]  
Q = (30)

dequeue(Q)



front++;  
Dequeue Q[front]  
Q = empty

Q đã rỗng!!!  
Làm thế nào để kiểm tra Q đã rỗng ?

**rear == front**

**Dùng giải pháp 1, một ô nhớ của mảng sẽ bị lãng phí vì không được dùng tới!!!**

# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn,[quấn quanh]

Giải pháp 1: Dùng biến *front* trở tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

**Giá trị khởi tạo ban đầu của 2 biến *front* và *rear* ?**

**`front = rear = maxSize - 1;`**

**Hàng đợi Q chỉ chứa được tối đa (`maxSize - 1`) phần tử, các phần tử nằm lần lượt từ `Q[front+1]` ... đến `Q[rear]`**

**Hàng đợi Q đầy khi: `rear + 1 == front`**

**Hàng đợi Q rỗng khi: `rear == front`**

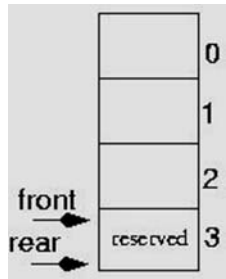
- `Q[front+1]` : phần tử đầu tiên của hàng đợi
- `Q[rear]` : phần tử cuối cùng của hàng đợi
- Thêm (Enqueue) 1 phần tử mới vào Q : `rear+=1; Q[rear]=item`
- Xóa (Dequeue) 1 phần tử khỏi Q : `front+=1`; sau đó xóa `Q[front]` khỏi hàng đợi

Ví dụ 4: Mảng có  $\text{maxSize} = 4$

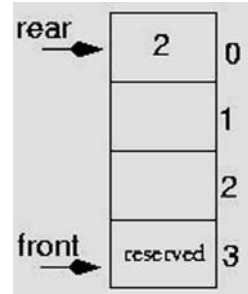
Khởi tạo:  $\text{front} = \text{rear} = \text{maxSize} - 1 = 3$ ;  
Q rỗng

## Mô phỏng giải pháp 1

Khởi tạo:  $\text{front} = \text{rear} = 3$ ;

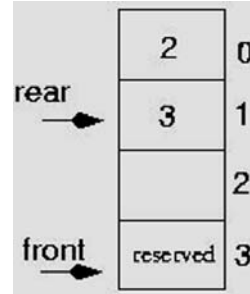


enqueue(Q, 2)



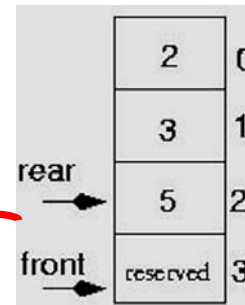
$\text{rear}++$ ;  
 $Q[\text{rear}] = 2$ ;  
 $Q = (2)$

enqueue(Q, 3)



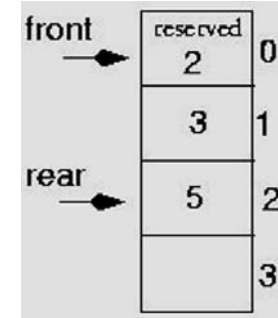
$\text{rear}++$ ;  
 $Q[\text{rear}] = 3$ ;  
 $Q = (2, 3)$

enqueue(Q, 5)



$\text{rear}++$ ;  
 $Q[\text{rear}] = 5$ ;  
 $Q = (2, 3, 5)$

dequeue(Q)



$\text{front}++$ ;  
 $\Rightarrow \text{front} = 4 = \text{maxSize}$   
 $\Rightarrow \text{wrap around} : \text{front} = 0$   
Dequeue  $Q[\text{front}]$   
 $Q = (3, 5)$

$\text{rear} + 1 == \text{front}$   
Hàng đợi Q đầy!!!

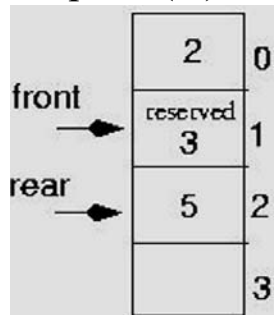
Tổng quát: kiểm tra Q đầy:

$(\text{rear} + 1) \% \text{maxSize} == \text{front}$

enqueue(Q, 30) ??

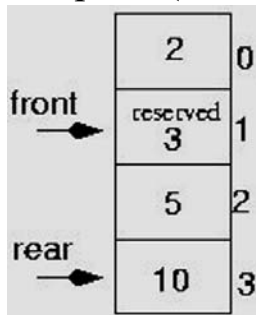
ERROR: Q đầy

dequeue(Q)



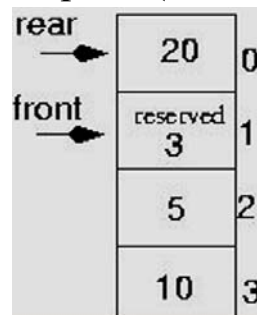
$\text{front}++$ ;  
dequeue  $Q[\text{front}]$   
 $Q = (5)$

enqueue(Q, 10)



$\text{rear}++$ ;  
 $Q[\text{rear}] = 10$ ;  
 $Q = (5, 10)$

enqueue(Q, 20)

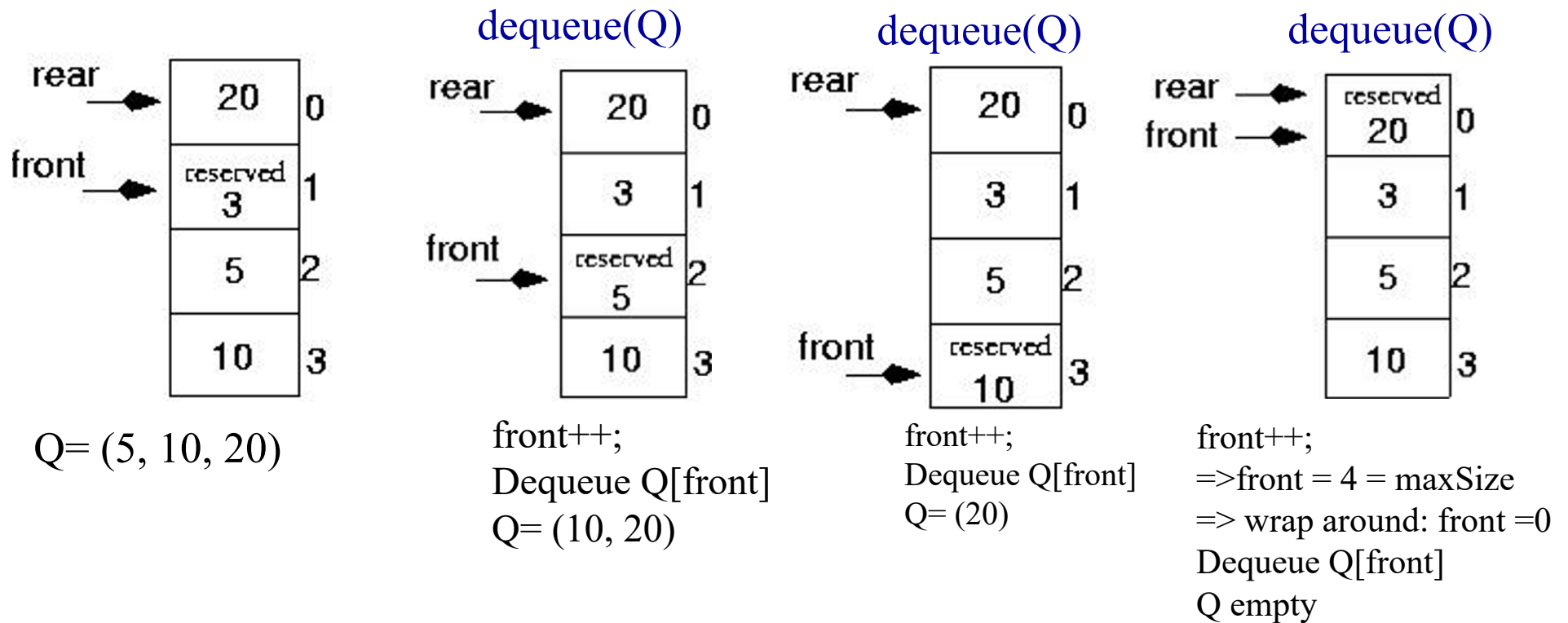


$\text{rear}++$ ;  
 $Q[\text{rear}] = 20$ ;  
 $Q = (5, 10, 20)$

Ví dụ 4: Mảng có  $\text{maxSize} = 4$

Khởi tạo:  $\text{front} = \text{rear} = \text{maxSize} - 1 = 3$ ;  
Q rỗng

Mô phỏng giải pháp 1



**Hàng đợi Q rỗng !!!**

**$\text{rear} == \text{front}$**

# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn,[quấn quanh]

Giải pháp 1: Dùng biến *front* trỏ tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

- Giá trị ban đầu của biến *front* và *rear* :

**`front = rear = maxSize - 1;`**

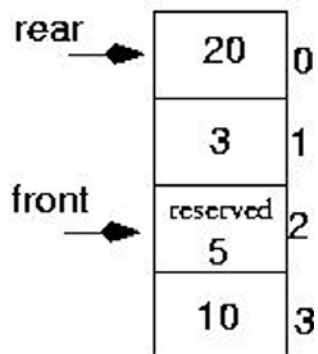
- `Q[front+1]` : phần tử đầu hàng đợi
- `Q[rear]` : phần tử cuối hàng đợi
- Thêm (Enqueue) 1 phần tử *item* vào hàng đợi Q :
  - `rear+=1; if (rear == maxSize) rear = 0;`
  - `Q[rear]=item`
- Xóa (Dequeue) 1 phần tử khỏi hàng đợi Q :
  - `front = (front + 1) % maxSize;`
  - sau đó xóa phần tử `Q[front]` khỏi hàng đợi
- Hàng đợi rỗng nếu : `rear == front`
- Hàng đợi đầy nếu: `(rear + 1) % maxSize == front`



Cài đặt cho Giải pháp 1: Dùng biến *front* trỏ tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

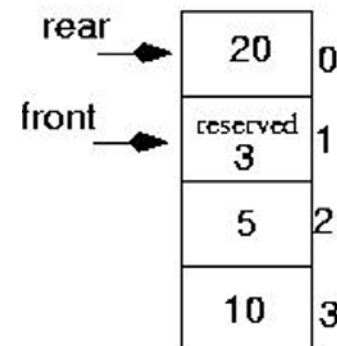
```
init(int max) //khởi tạo hàng đợi Q rỗng
{
    maxSize = max;
    front = maxSize - 1;
    rear = maxSize - 1;
    Q = new ItemType[maxSize];
}

int sizeQ(Q) //hàm trả về số lượng phần tử đang có trong hàng đợi Q
{
    int size = (maxSize - front + rear) % maxSize;
    return size;
}
```



Q = (10, 20)

số phần tử =  $(4 - 2 + 0) \% 4 = 2$



Q = (5, 10, 20)

số phần tử =  $(4 - 1 + 0) \% 4 = 3$

Cài đặt cho Giải pháp 1: Dùng biến *front* trỏ tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

```
isEmpty(Q) // trả về giá trị "true" nếu hàng đợi Q là rỗng(empty)
```

```
{  
    if (rear == front) return true;  
    else return false;  
}
```

```
isFull(Q) /*hàm trả về giá trị "true" nếu hàng đợi Q đầy (full), tức là đã dùng hết bộ nhớ dành cho Q; ngược lại  
hàm trả về giá trị "false" */
```

```
{  
    if ((rear + 1) % maxSize == front) return true;  
    else return false;  
}
```

```
frontQ(Q) //hàm trả về giá trị phần tử đầu (front (head)) hàng đợi, hoặc trả về thông báo error nếu Q đã đầy.
```

```
{  
    return Q[front + 1];  
}
```

Cài đặt cho Giải pháp 1: Dùng biến *front* trỏ tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

`enqueue(Q, x)` /\*hàm thêm phần tử x vào phía sau (back (rear)) hàng đợi Q. Nếu Q đã đầy trước khi thêm, cần thông báo: Q đã đầy, không thể thêm\*/

```
{
    if (isFull(Q)) ERROR("Queue is FULL");
    else
    {
        rear++;
        if (rear == maxSize) rear = 0;
        Q[rear] = x;
    }
}

enqueue(Q, x)
{
    if (isFull(Q)) ERROR("Queue is FULL");
    else
    {
        rear = (rear + 1) % maxSize;
        Q[rear] = x;
    }
}
```

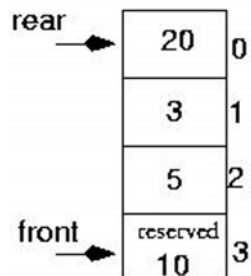
Cài đặt cho Giải pháp 1: Dùng biến *front* trỏ tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

`dequeue(Q)` /\*deletes the element at the front (head) of the queue Q, then returns x which is the data of this element. If the queue Q is empty before dequeue, then give the error notification\*/

```
{
    if (isEmpty(Q)) ERROR("Queue is EMPTY");
    else
    {
        front = (front + 1);
        if (front == maxSize) front = 0;
        return Q[front];
    }
}
```

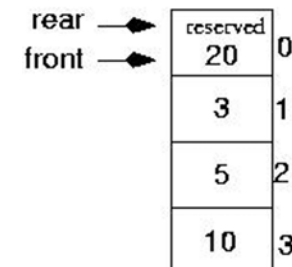
`dequeue(Q)` //Một cách viết khác:

```
{ if (isEmpty(Q)) ERROR("Queue is EMPTY");
    else
    {
        front = (front + 1) % maxSize;
        return Q[front];
    }
}
```



Dequeue (Q)

front = front + 1;  
 => front = 4 = maxSize  
 => wrap around: front = 0  
 Dequeue Q[front]  
 Q empty



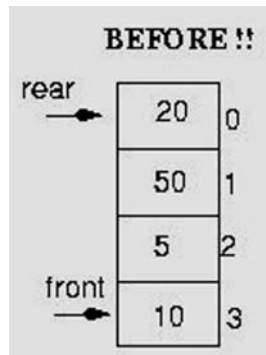
# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn,[quấn quanh] (circular queue [“wrap around”])

Giải pháp 1: Dùng biến *front* trở tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

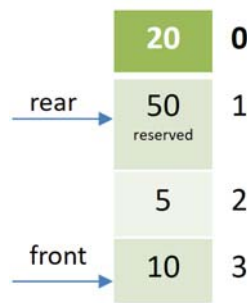
Solution 2: Dùng biến *rear* trở tới phần tử ngay sau phần tử cuối Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

# Ví dụ 5: Mô phỏng giải pháp 2



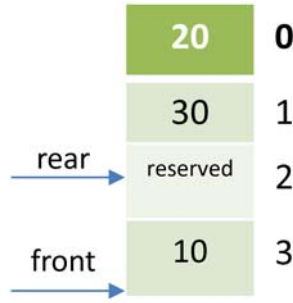
Q = (10, 20)

Giải pháp 2: Dùng biến *rear* trở tới phần tử ngay sau phần tử cuối Q (khi đó, 1 ô nhớ sẽ lãng phí vì không được dùng tới)



Q = (10, 20)

enqueue(Q, 30)



Q[rear] = 30; rear++;  
Q = (10, 20, 30)

Q đã đầy!!!  
Làm thế nào để biết được Q đã đầy ?

**rear + 1 == front**

dequeue(Q)

?

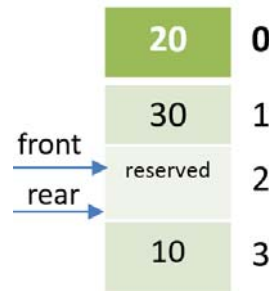
front++;  
⇒ front = 4 = maxSize  
⇒ wrap around: front = 0  
Q = (20, 30)

dequeue(Q)

?

front++;  
Q = (30)

dequeue(Q)



front++;  
Q = empty

Q đã rỗng!!!  
Làm thế nào để biết được Q đang rỗng ?

**rear == front**

**Dùng giải pháp 2, một ô nhớ của mảng sẽ bị lãng phí vì không được dùng tới!!!**

# Cài đặt hàng đợi Queue: dùng mảng (Array)

- Cách 2: hàng đợi Q vòng tròn,[quấn quanh] (circular queue [“wrap around”])

Giải pháp 1: Dùng biến *front* trở tới phần tử ngay trước phần tử đầu Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

Vậy với giải pháp 2: giá trị ban đầu cho biến *front* và *rear* ?

```
front = rear = maxSize - 1;
```

Giải pháp 2: Dùng biến *rear* trở tới phần tử ngay sau phần tử cuối Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ)

Vậy với giải pháp 2, giá trị ban đầu cho biến *front* và *rear* ?

```
front = rear = 0;
```

Ví dụ 6: Mảng có kích thước  $\text{maxSize} = 4$

Khởi tạo:  $\text{front} = \text{rear} = 0$ ;

Queue Q rỗng

Minh họa giải pháp 2

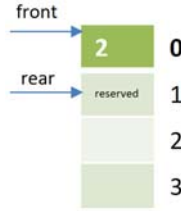
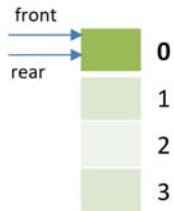
Init:  $\text{front} = \text{rear} = 0$ ;

$\text{enqueue}(Q, 2)$

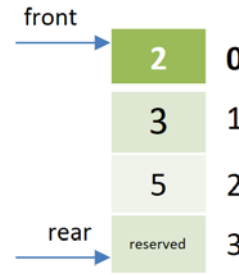
$\text{enqueue}(Q, 3)$

$\text{enqueue}(Q, 5)$

$\text{dequeue}(Q)$



?



?

$Q[\text{rear}] = 2$ ;  
 $\text{rear}++$ ;  
 $Q = (2)$

$Q[\text{rear}] = 3$ ;  
 $\text{rear}++$ ;  
 $Q = (2, 3)$

$Q[\text{rear}] = 5$ ;  
 $\text{rear}++$ ;  
 $Q = (2, 3, 5)$

Dequeue  $Q[\text{front}]$   
 $\text{front}++$ ;  
 $Q = (3, 5)$

$\text{dequeue}(Q)$

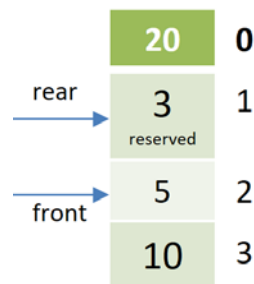
$\text{enqueue}(Q, 10)$

$\text{enqueue}(Q, 20)$

$(\text{rear} + 1) \% \text{maxSize} == \text{front}$   
**Queue Q đầy!!!**

?

?



$\text{enqueue}(Q, 30) ??$

ERROR: Queue đầy

dequeue  $Q[\text{front}]$   
 $\text{front}++$ ;  
 $Q = (5)$

$Q[\text{rear}] = 10$ ;  
 $\text{rear}++$ ;  
 $\text{rear} == \text{maxSize}$   
 $\rightarrow$  wrap around:  $\text{rear} = 0$   
 $Q = (5, 10)$

$Q[\text{rear}] = 20$ ;  
 $\text{rear}++$ ;  
 $Q = (5, 10, 20)$

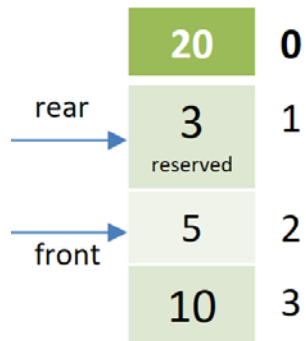


Ví dụ 6: Mảng có kích thước `maxSize = 4`

Khởi tạo: `front = rear = 0;`

Queue Q rỗng

Minh họa giải pháp 2



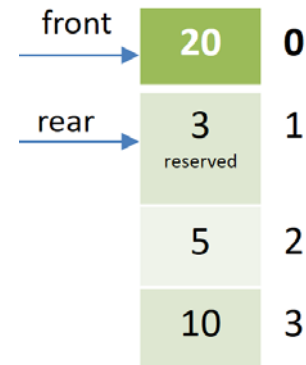
$Q = (5, 10, 20)$

`dequeue(Q)`

?

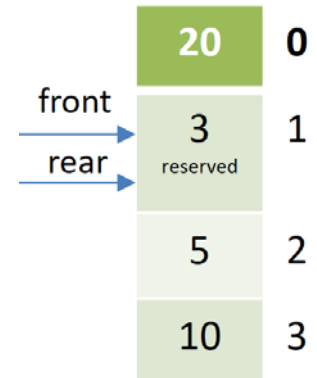
Dequeue  $Q[\text{front}]$   
`front++;`  
 $Q = (10, 20)$

`dequeue(Q)`



Dequeue  $Q[\text{front}]$   
`front++;`  
 $\Rightarrow \text{front} = 4 = \text{maxSize}$   
 $\Rightarrow \text{wrap around: front} = 0$   
 $Q = (20)$

`dequeue(Q)`



Dequeue  $Q[\text{front}]$   
`front++;`  
 $Q$  empty

**Queue Q rỗng!!!**

**`rear == front`**

Giải pháp 2: Dùng biến *rear* trở tới phần tử ngay sau phần tử cuối Q (khi đó, 1 ô nhớ sẽ không được dùng tới -> lãng phí 1 ô nhớ) in

Bài tập: Viết các hàm sau trên hàng đợi Q cho trường hợp giải pháp 2

- `isEmpty(Q)` ; hàm trả về “true” nếu hàng đợi Q đang rỗng (empty)
- `isFull(Q)` ; hàm trả về “true” nếu hàng đợi Q đang đầy (full), tức là đã sử dụng tối đa bộ nhớ dành cho hàng đợi Q; ngược lại hàm trả về giá trị “false”
- `frontQ(Q)` ; hàm trả về phần tử đầu (front (head)) hàng đợi Q; hoặc báo lỗi error nếu Q rỗng.
- `enqueue(Q, x)` ; hàm thêm phần tử x vào phía sau (back (rear)) hàng đợi Q. Nếu trước khi thêm mà Q đã đầy, thì không thể thêm được, cần đưa ra thông báo: Q đã đầy, không thể thêm phần tử.
- `x = dequeue(Q)` ; hàm thực hiện xóa phần tử nằm đầu (front (head)) hàng đợi Q, trả về x là giá trị của phần tử này. Nếu trước khi xóa mà hàng đợi Q đã rỗng, thì đưa ra thông báo lỗi (error).
- `sizeQ(Q)` ; hàm trả về số lượng phần tử đang có trong hàng đợi Q.

## Ứng dụng 1: nhận dạng palindrome (recognizing palindromes)

- **Định nghĩa.** Ta gọi *palindrome* là xâu mà đọc nó từ trái qua phải cũng giống như đọc nó từ phải qua trái.

Ví dụ: NOON, DEED, RADAR, MADAM

*Able was I ere I saw Elba*

- Để nhận biết một xâu cho trước có phải là palindrome hay không: ta thực hiện 2 bước sau:
  - Bước 1: đưa các ký tự của nó đồng thời vào một hàng đợi và một ngăn xếp.
  - Bước 2: Sau đó lần lượt loại bỏ các ký tự khỏi hàng đợi và ngăn xếp và tiến hành so sánh:
    - Nếu phát hiện sự khác nhau giữa hai ký tự, một ký tự được lấy ra từ ngăn xếp còn ký tự kia lấy ra từ hàng đợi, thì xâu đang xét không là palindrome.
    - Nếu tất cả các cặp ký tự lấy ra là trùng nhau thì xâu đang xét là palindrome.

# Ví dụ 1: kiểm tra “RADAR” có phải là palindrome

Bước 1: Đẩy “RADAR” vào Queue và Stack:

**Kí tự hiện tại**

**Queue**

(front ở bên trái,  
rear ở bên phải)

**Stack**

(top ở bên trái)

R

R

R

A

R A

A R

D

R A D

D A R

A

R A D A

A D A R

R

R A D A R

R A D A R

↑  
front

↑  
rear

↑  
top

## Ví dụ 1: kiểm tra “RADAR” có phải là palindrome

Bước 2: Xóa “RADAR” khỏi Queue và Stack:

- Dequeue lần lượt từng kí tự cho đến khi queue rỗng
- Pop lần lượt từng kí tự khỏi stack cho đến khi stack rỗng

Queue (front ở bên trái)	Front của Queue	Top của Stack	Stack (top ở bên trái)
R A D A R	R	R	R A D A R
A D A R	A	A	A D A R
D A R	D	D	D A R
A R	A	A	A R
R	R	R	R
empty	empty	empty	empty

**Kết luận: xâu "RADAR" là palindrome**

## Ví dụ 2: phát hiện palindrome

*Able was I ere I saw Elba*

a
b
I
E
⋮
e
I
b
A

**Stack**

A	b	I	e		.....		E	I	b	a
---	---	---	---	--	-------	--	---	---	---	---

**Queue**

## Ứng dụng 2: Chuyển đổi xâu số về số thập phân

**Thuật toán được mô tả trong sơ đồ sau:**

```
// Chuyển dãy chữ số trong Q thành số thập phân n  
// Loại bỏ các dấu cách/trống ở đầu nếu có  
do { dequeue(Q, ch)  
} until ( ch != blank)  
// ch lúc này chứa chữ số đầu tiên của xâu đã cho  
// Tính n từ dãy chữ số trong hàng đợi  
n = 0;  
done = false;  
do { n = 10 * n + số nguyên mà ch biểu diễn;  
      if (! isEmpty(Q) )  
        dequeue(Q, ch)  
      else  
        done = true  
} until ( done || ch != digit)  
// Kết quả: n chứa số thập phân cần tìm
```

# Cài đặt hàng đợi Queue

- Cũng tương tự stack, ta có thể cài đặt Queue bằng 2 cách:
  - Sử dụng mảng (array)
  - **Sử dụng danh sách liên kết (linked list)**



# Cài đặt hàng đợi Queue: dùng danh sách liên kết

```
typedef struct {  
    DataType element;  
    struct node *next;  
} node;  
typedef struct {  
    node *front;  
    node *rear;  
} queue;
```

với `DataType` là kiểu dữ liệu của phần tử sẽ lưu trữ trong hàng đợi;  
`DataType` cần phải được khai báo trước khi khai báo hàng đợi queue.

- Cài đặt hàng đợi sử dụng danh sách liên kết:
  - Phần tử ở đầu (Front/head) hàng đợi được lưu trữ là nút đầu (head) của danh sách liên kết, phần tử ở cuối (rear) của hàng đợi là nút cuối (tail) của danh sách.
  - *Thao tác Thêm (Enqueue)*: thêm phần tử vào cuối danh sách.
  - *Thao tác Xóa (Dequeue)*: xóa phần tử đang ở đầu danh sách.