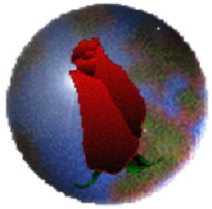
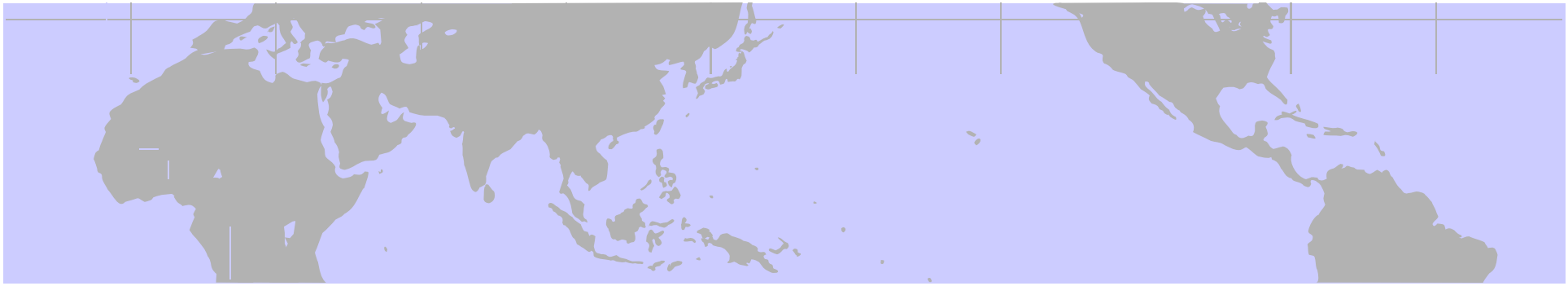
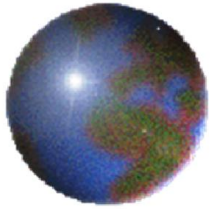


Chương 3

CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN

(*Basic Data Structures*)





Bài giảng của
PGS.TS. NGUYỄN ĐỨC NGHĨA

Khoa học Máy tính
Đại học Bách khoa Hà nội

Nội dung

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

Chương 3. Các cấu trúc dữ liệu cơ bản

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

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

Các kiểu dữ liệu dựng sẵn
(Built-in data types)

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

Dữ liệu đối với kiểu nguyên thuỷ

Trong ngôn ngữ lập trình C

Type	Bits	Minimum value	Maximum value
byte	8	-128	127
short	16	-32768	32767
char	16	0	65535
int	32	$-2147483648 = -2^{31}$	$2147483647 = 2^{31}-1$
long	64	-9223372036854775808	9223372036854775807
float	32	$\approx \pm 1.40 \times 10^{-45}$	$\approx \pm 3.40 \times 10^{38}$
double	64	$\approx \pm 4.94 \times 10^{-324}$	$\approx \pm 1.80 \times 10^{308}$

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

Phép toán đối với kiểu dữ liệu nguyên thuỷ

- **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.

Kiểu dữ liệu trừu tượng
(Abstract Data Types)

- Kiểu dữ liệu trừu tượng (Abstract Data Type -ADT) bao gồm:
  - tập các giá trị (set of *values*) và
  - tập các phép toán (set of *operations*) có thể thực hiện với tất cả các giá trị này
- Phần nào của kiểu dữ liệu (Data Type) đã bị bỏ qua trong ADT ?
  - cách biểu diễn dữ liệu (*data representation*) được sử dụng chung cho tất cả các giá trị này
- Việc làm này có ý nghĩa làm trừu tượng hoá khái niệm kiểu dữ liệu. ADT không còn phụ thuộc vào cài đặt, không phụ thuộc ngôn ngữ lập trình.

Kiểu dữ liệu trừu tượng
(Abstract Data Type - ADT)

- ***Ví dụ:***

ADT	Đối tượng (Object)	Phép toán (Operations)
Danh sách (List)	các nút	chèn, xoá, tìm,...
Đồ thị (Graphs)	đỉnh, cạnh	duyệt, đường đi, ...
Integer	-∞...,-1, 0, 1,... +∞	+, -, *, v.v...
Real	-∞, ..., +∞	+, -, *, v.v...
Ngăn xếp	các phần tử	pop, push, isEmpty,...
Hàng đợi	Các phần tử	enqueue, dequeue,...
Cây nhị phân	các nút	traversal, find,...

Kiểu dữ liệu trừu tượng
(Abstract Data Type - ADT)

- Điều dễ hiểu là các kiểu dữ liệu nguyên thuỷ mà các ngôn ngữ lập trình cài đặt sẵn cũng được coi là thuộc vào kiểu dữ liệu trừu tượng. Trên thực tế chúng là cài đặt của kiểu dữ liệu trừu tượng trên ngôn ngữ lập trình cụ thể.
- **Định nghĩa.** Ta gọi việc *cài đặt* (implementation) một ADT là việc diễn tả bởi các câu lệnh của một ngôn ngữ lập trình để mô tả các biến trong ADT và các thủ tục trong ngôn ngữ lập trình để thực hiện các phép toán của ADT, hoặc trong các ngôn ngữ hướng đối tượng, là các lớp (class) bao gồm cả dữ liệu (data) và các phương thức xử lý (methods).

Kiểu dữ liệu - Kiểu dữ liệu trừu tượng và Cấu trúc dữ liệu
(Data Types, Data Structures and Abstract Data Types)

- Có thể nói những thuật ngữ: kiểu dữ liệu, kiểu dữ liệu trừu tượng và cấu trúc dữ liệu nghe rất giống nhau, nhưng thực ra chúng có ý nghĩa khác nhau.
  - Trong ngôn ngữ lập trình, ***kiểu dữ liệu*** của biến là tập các giá trị mà biến này có thể nhận. Ví dụ, biến kiểu boolean chỉ có thể nhận giá trị đúng hoặc sai. Các kiểu dữ liệu cơ bản có thể thay đổi từ ngôn ngữ lập trình này sang NNLT khác. Ta có thể tạo những kiểu dữ liệu phức hợp từ những kiểu dữ liệu cơ bản. Cách tạo cũng phụ thuộc vào ngôn ngữ lập trình.
  - ***Kiểu dữ liệu trừu tượng*** là mô hình toán học cùng với những phép toán xác định trên mô hình này. Nó là không phụ thuộc vào ngôn ngữ lập trình.
  - Để biểu diễn mô hình toán học trong ADT ta sử dụng ***cấu trúc dữ liệu.***

Cấu trúc dữ liệu
(Data Structures)

- ***Cấu trúc dữ liệu*** (Data Structures) là một họ các biến, có thể có kiểu dữ liệu khác nhau, được liên kết lại theo một cách thức nào đó.
- Việc cài đặt ADT đòi hỏi lựa chọn cấu trúc dữ liệu để biểu diễn ADT.
- Ta sẽ xét xem việc làm đó được tiến hành như thế nào?
- ***Ô*** (cell) là đơn vị cơ sở cấu thành cấu trúc dữ liệu. Có thể hình dung ô như là cái hộp đựng giá trị phát sinh từ một kiểu dữ liệu cơ bản hay phức hợp.
- Cấu trúc dữ liệu được tạo nhờ đặt tên cho một ***nhóm*** các ô và đặt giá trị cho một số ô để mô tả sự liên kết giữa các ô.
- Ta xét một số cách tạo nhóm.

Cấu trúc dữ liệu
(Data Structures)

- Một trong những cách tạo nhóm đơn giản nhất trong các ngôn ngữ lập trình đó là ***mảng*** (array). Mảng là một dãy các ô có cùng kiểu xác định nào đó.
- **Ví dụ:** Khai báo trong Csau đây

```
C
int name[10]
```

khai báo biến *name* gồm 10 phần tử kiểu cơ sở nguyên (integer).

- Có thể truy xuất đến phần tử của mảng nhờ chỉ ra tên mảng cùng với chỉ số của nó.
- Ta sẽ xét kỹ hơn kiểu mảng trong mục tiếp theo.

Cấu trúc dữ liệu
(Data Structures)

- Một phương pháp chung nữa hay dùng để nhóm các ô là ***cấu trúc bản ghi* (*record structure*).**
- ***Bản ghi*** (record) là ô được tạo bởi một họ các ô (gọi là các trường) có thể có kiểu rất khác nhau.
- Các bản ghi lại thường được nhóm lại thành mảng; kiểu được xác định bởi việc nhóm các trường của bản ghi trở thành kiểu của phần tử của mảng.
- **Ví dụ:** Trong C mô tả

C

```
struct record {
        float data;
        int next; } reclist[100];
```

khai báo reclist là mảng 100 phần tử, mỗi ô là một bản ghi gồm 2 trường: *data* và *next*.

Con trỏ
(Pointer)

- Một trong những ưu thế của phương pháp nhóm các ô trong các NNLT là ta có thể biểu diễn mối quan hệ giữa các ô nhờ sử dụng con trỏ.
- **Định nghĩa.** ***Con trỏ*** (pointer) là ô mà giá trị của nó chỉ ra một ô khác.
- Khi vẽ các cấu trúc dữ liệu, để thể hiện ô *A* là con trỏ đến ô *B*, ta sẽ sử dụng mũi tên hướng từ *A* đến *B*.

- **Ví dụ:** Để tạo biến con trỏ *ptr* để trỏ đến ô có kiểu cho trước, chẳng hạn celltype, ta có thể khai báo:

Trong C
celltype *ptr

Phân loại các cấu trúc dữ liệu

- Trong nhiều tài liệu về CTDL thường sử dụng phân loại cấu trúc dữ liệu sau đây:
  - **Cấu trúc dữ liệu cơ sở** (Base data structures).

    Ví dụ: trong C: int, char, float, double,...
  - **Cấu trúc dữ liệu tuyến tính** (Linear data structures).

    Ví dụ: Mảng (Array), Danh sách liên kết (Linked list), Ngăn xếp (Stack), Hàng đợi (Queue),
  - **Cấu trúc dữ liệu phi tuyến** (Nonlinear data structures).

    Ví dụ: Cây (trees), đồ thị (graphs), bảng băm (hash tables),...

Chương 3. Các cấu trúc dữ liệu cơ bản

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

3.2. Mảng

3.2.1. Kiểu dữ liệu trừu tượng mảng

3.2.2. Phân bổ bộ nhớ cho mảng

3.2.3. Các thao tác với mảng

Kiểu dữ liệu trừu tượng mảng

- Mảng được định nghĩa như kiểu dữ liệu trừu tượng như sau:
- **Đối tượng (object):** tập các cặp <*index, value*> trong đó với mỗi giá trị của *index* có một giá trị từ tập *item*. *Index* là tập có thứ tự một chiều hay nhiều chiều, chẳng hạn, {0, … , *n*-1} đối với một chiều, {(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)} đối với 2 chiều, v.v.
- **Các hàm (Functions):**

  Với mọi A ∈ Array, *i* ∈ *index*, x ∈ *item, j, size* ∈ integer

  Khởi tạo mảng:

  Array Create(j, list) ::= trả lại mảng j chiều, trong đó **list** là bộ j thành phần với thành phần thứ i là kích thước của chiều thứ i. **Item** không được định nghĩa.

Kiểu dữ liệu trừu tượng mảng (ADT Array)

- **Truy xuất phần tử:** *Item* Retrieve(A, *i*)

  if (*i* ∈ *index*) **return** item tương ứng với giá trị chỉ số *i* trong mảng A

  else return error

- **Cất giữ phần tử:** *Array Store*(A, *i, x*)

  if (*i* in *index*) **return** mảng giống hệt mảng A ngoại trừ cặp mới <*i, x*> được chèn vào

  else return error

- Ta xét việc cài đặt mảng trong các ngôn ngữ lập trình. Ta hạn chế ở việc xét mảng một chiều và hai chiều

Cấu trúc dữ liệu mảng
Array data structures)

- **Mảng (array)** là dãy các thành phần được đánh chỉ số.
  - Thông thường mảng chiếm giữ một dãy từ máy liên tiếp trong bộ nhớ (cách lưu trữ này được gọi là ***lưu trữ kế tiếp***)
  - Độ dài của mảng được xác định khi khởi tạo và không thể thay đổi.
  - Mỗi thành phần 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
  - Mỗi thành phần của mảng được truy xuất nhờ sử dụng chỉ số của nó
    - Phép toán này được thực hiện một cách hiệu quả: với thời gian $\Theta(1)$

Mảng trong các ngôn ngữ lập trình

- Các chỉ số có thể là số nguyên (C, Java) hoặc là các giá trị kiểu rời rạc (Pascal, Ada)
- Cận dưới là 0 (C, Java), 1 (Fortran), hoặc tuỳ 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 một chiều trong PASCAL/C

C

<kiểu thành phần> <tên biến>[size]

Ví dụ: int list[5];

Khai báo trên sẽ khai báo biến mảng tên name với 5 phần tử có kiểu là số nguyên (2 byte).

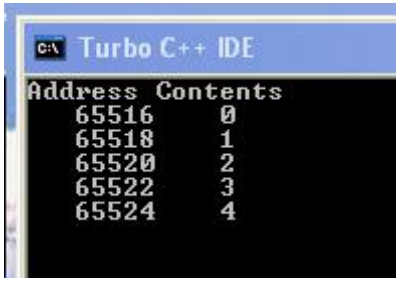
Địa chỉ của các phần tử trong mảng một chiều

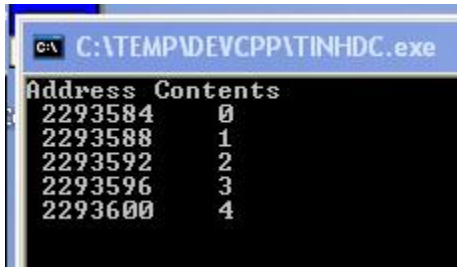
list[0]	địa chỉ gốc = α
list[1]	α + sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*size(int)

Ví dụ:

- Chương trình trên C sau đây đưa ra địa chỉ của các phần tử của mảng một chiều trên C:

Kết quả chạy trong DEVC (sizeof(int)=4)






```
#include <stdio.h>
#include <conio.h>
int main()
{   int one[] = {0, 1, 2, 3, 4};
    int *ptr;   int rows=5;
/* in địa chỉ của mảng một chiều nhờ dùng con trỏ  */
     int i; ptr= one;
     printf("Address Contents\n");
     for (i=0; i < rows; i++)
            printf("%8u%5d\n", ptr+i, *(ptr+i));
     printf("\n");
     getch();
}
```

Kết quả chạy trong TC: (sizeof(int)=2)

```
Address Contents
   65516    0
   65518    1
   65520    2
   65522    3
   65524    4
```

Mảng hai chiều

- **Khai báo mảng hai chiều trong C:**

  <element-type> <arrayName> [size 1][size2];

  - **Ví dụ:**

    double table[5] [4];

  - Truy xuất đến phần tử của mảng: table[2] [4];

Phân bổ bộ nhớ cho mảng hai chiều

- Xét khai báo

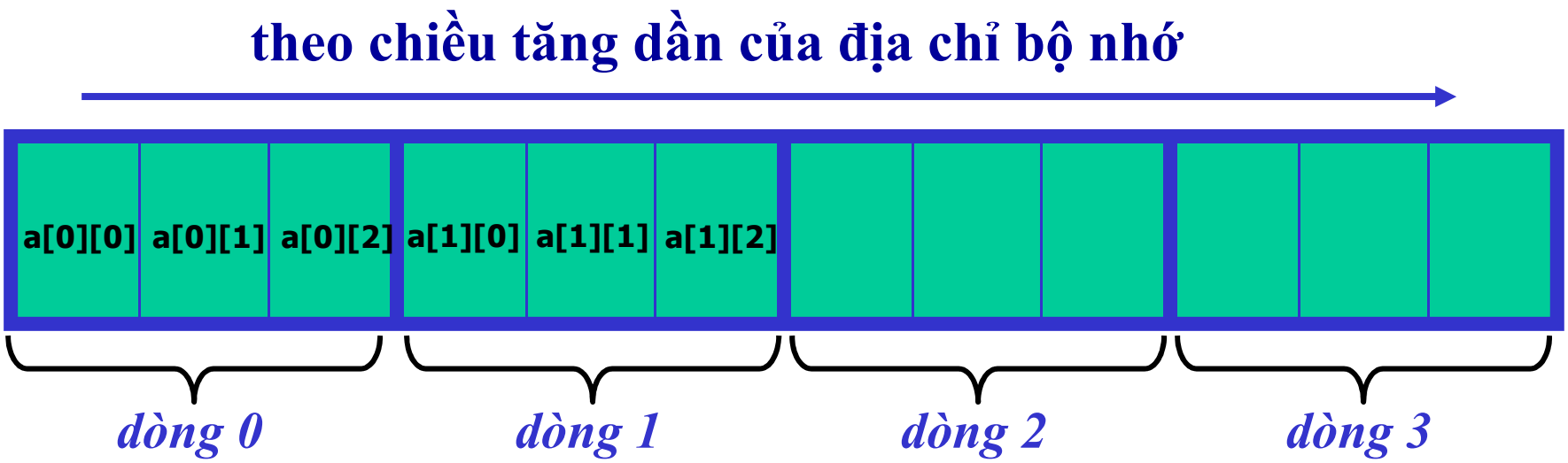
  int a [4] [3];

- Thông thường có thể coi nó như một bảng.

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**

Phân bổ bộ nhớ cho mảng hai chiều

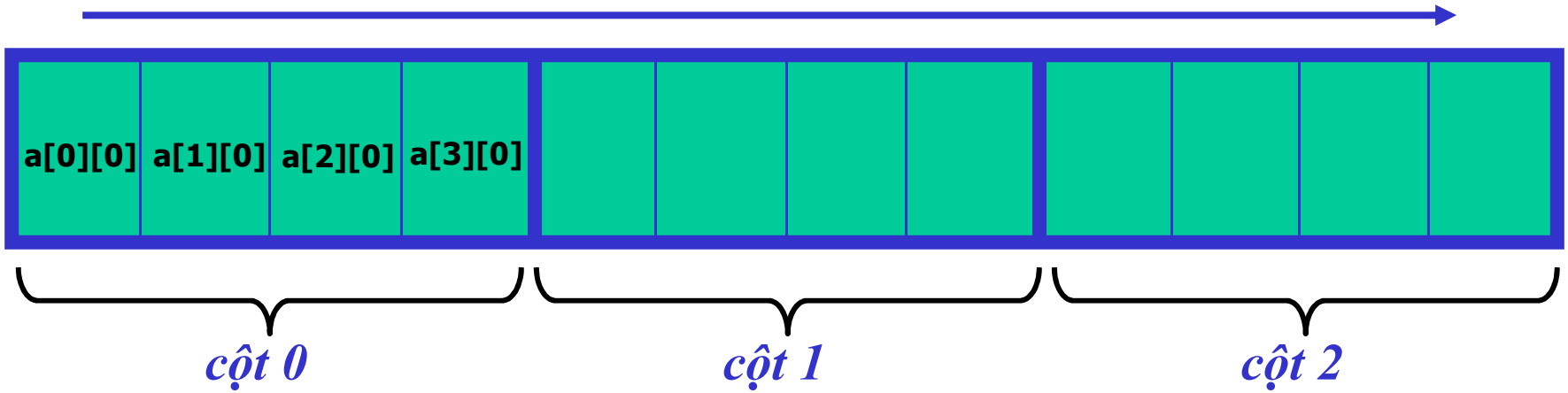
- Trong bộ nhớ (chỉ có một chiều) các ***hàng*** của mảng hai chiều được sắp xếp kế tiếp nhau theo một trong hai cách sau:
  - ***Hết dòng này đến dòng khác:*** thứ tự sắp xếp này được gọi là ***thứ tự ưu tiên dòng*** - *row major order*).
    - Ví dụ: C sử dụng cách sắp xếp này.



Phân bổ bộ nhớ cho mảng hai chiều

- ***Hết cột này đến cột khác:*** Thứ tự sắp xếp này gọi là ***thứ tự ưu tiên cột*** (*column major order*).
  - Ví dụ FORTRAN, MATLAB sử dụng cách sắp xếp này.

theo chiều tăng dần của địa chỉ bộ nhớ

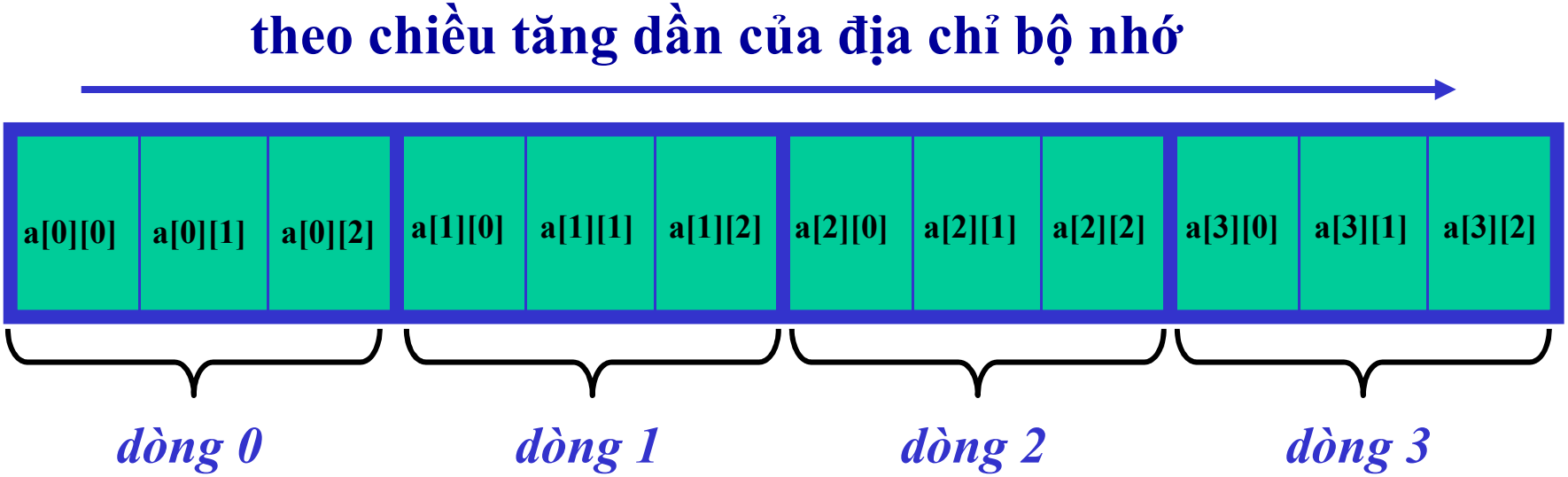


Phân bổ bộ nhớ cho mảng hai chiều

- **Chú ý:** Nếu biết địa chỉ của phần tử đầu tiên của mảng, ta dễ dàng tính được địa chỉ của phần tử tuỳ ý trong mảng.
- **Ví dụ:** Xét cách phân bổ bộ nhớ cho biến mảng khai báo bởi

  int a[4] [3];

  theo thứ tự ưu tiên dòng:



Phân bổ bộ nhớ cho mảng hai chiều

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

  int a[4][3]

  a[0][0] có địa chỉ là α

  a[0][1] α + sizeof(int)

  a[0][2] α + 2*sizeof(int)

  a[1][0] α + 3*sizeof(int)

  a[1][1] α + 4*sizeof(int)

  a[1][2] α + 5*sizeof(int)

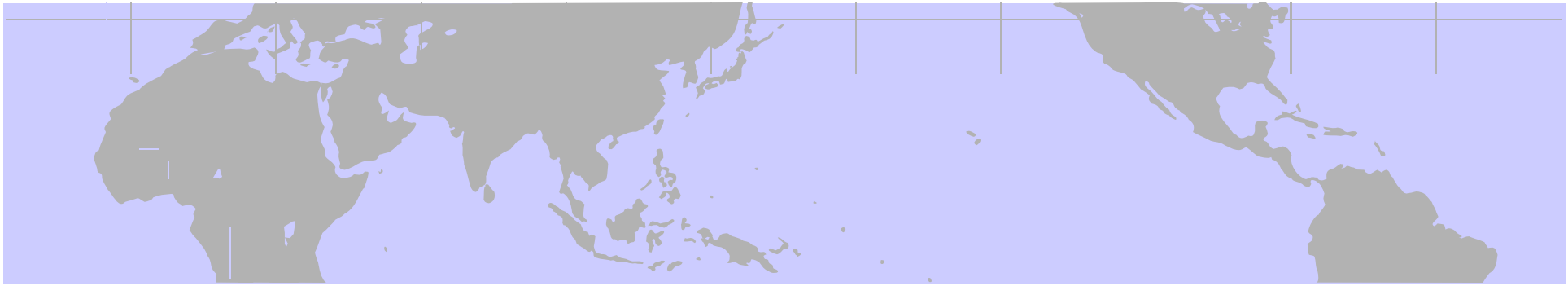
  a[2][0] α + 6*sizeof(int)

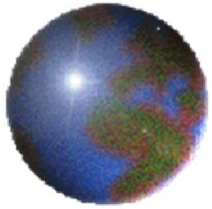
  . . .

- **Tổng quát:** Xét khai báo

  int a[m][n]

- Giả sử: địa chỉ của phần tử đầu tiên của mảng (a[0][0]) là α.
- Khi đó địa chỉ của a[i][j] sẽ là **α + (i*n + j)*sizeof(int)**

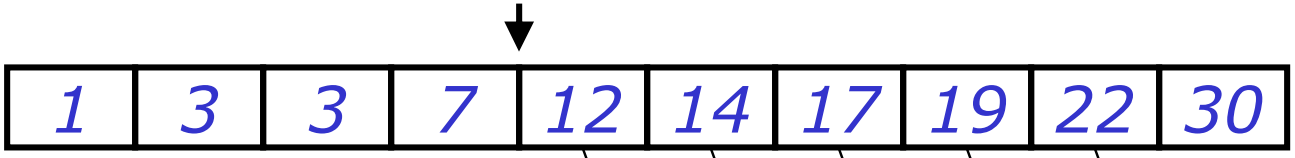




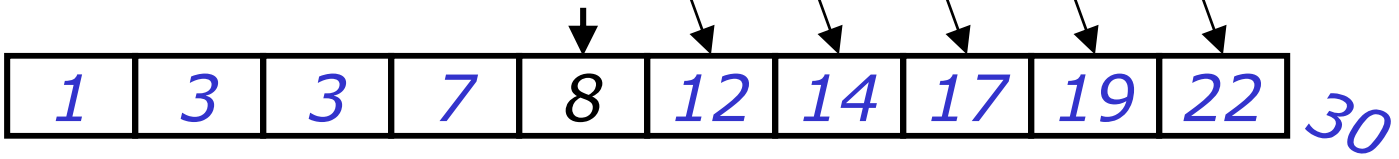
Các thao tác với mảng

Chèn phần tử vào mảng
(Inserting an element into an array)

- Giả sử ta muốn chèn 8 vào một mảng được sắp xếp (và đảm bảo dãy vẫn là được sắp xếp



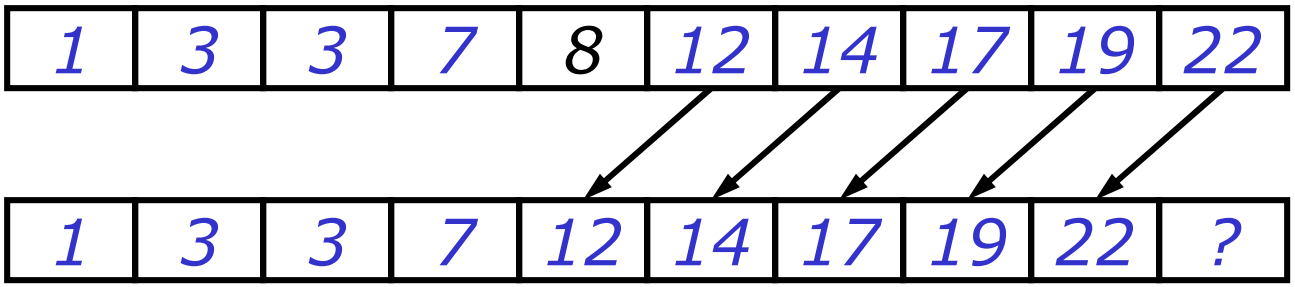
- Ta có thể thực hiện điều này nhờ việc chuyển dịch sang phải một ô tất cả các phần tử đứng sau vị trí đánh dấu
  - Tất nhiên, ta phải loại bỏ 30 khi thực hiện điều này



- **Việc dịch chuyển tất cả các phần tử là một thao tác chậm (đòi hỏi thời gian tuyến tính đối với kích thước mảng)**

Xoá bỏ một phần tử
(Deleting an element from an array)

- Việc xoá bỏ (Deleting) một phần tử được thực hiện tương tự, ta phải dịch sang trái tất cả các phần tử sau nó



- Phép toán xoá là phép toán chậm; việc thực hiện thường xuyên thao tác này là điều không mong muốn.
- Phép xoá sẽ làm xuất hiện một vị trí tự do ở cuối mảng
  - Chúng ta đánh dấu nó là tự do bằng cách nào?
    - Ta cần giữ số lượng phần tử được cất giữ trong mảng

Chương 3. Các cấu trúc dữ liệu cơ bản

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

Danh sách tuyến tính

- **Định nghĩa**
  - Danh sách tuyến tính (Linear List) dãy gồm 0 hoặc nhiều hơn các phần tử cùng kiểu cho trước: $(a_1,a_2,\ldots,a_n)$, $n \geq 0$.
  - a_i là phần tử của danh sách.
  - a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng.
  - n là độ dài của danh sách.
  - Khi $n = 0$, ta có danh sách rỗng (empty list).
  - Các phần tử được *sắp thứ tự tuyến tính* theo vị trí của chúng trong danh sách. Ta nói a_i đi trước a_{i+1}, a_{i+1} đi sau a_i và a_i ở vị trí i.
- **Ví dụ**
  - Danh sách các sinh viên được sắp thứ tự theo tên
  - Danh sách điểm thi sắp xếp theo thứ tự giảm dần

Danh sách tuyến tính

Đưa vào ký hiệu:

L : danh sách các đối tượng có kiểu *element_type*

x : một đối tượng kiểu *element_type*

p : kiểu vị trí

END(L) : hàm trả lại vị trí đi sau vị trí cuối cùng trong danh sách L

Dưới đây ta kể ra một số phép toán đối với danh sách tuyến tính:

0. Creat() Khởi tạo danh sách rỗng

1. Insert (*x*, *p*, *L*)

- Chèn *x* vào vị trí *p* trong danh sách *L*
- Nếu *p* = *END*(*L*), chèn *x* vào cuối danh sách
- Nếu *L* không có vị trí *p*, kết quả là không xác định

Danh sách tuyến tính: Các phép toán

2. Locate (*x*, *L*)

- Trả lại vị trí của *x* trong *L*
- trả lại END(L) nếu *x* không xuất hiện

3. Retrieve (*p*, *L*)

- trả lại phần tử ở vị trí *p* trong *L*
- không xác định nếu *p* không tồn tại hoặc $p = END(L)$

4. Delete (*p*, *L*)

- xoá phần tử ở vị trí *p* trong *L* . Nếu *L* là $a_1, a_2, \ldots, a_n$, thì *L* sẽ trở thành $a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n$.
- kết quả là không xác định nếu *L* không có vị trí *p* hoặc $p = END(L)$

5. Next (*p*, *L*)

- trả lại vị trí đi ngay sau vị trí *p*
- Nếu *p* là vị trí cuối cùng trong *L*, thì NEXT(*p*, *L*) = END(*L*). NEXT không xác định nếu *p* là END(*L*) hoặc *p* không tồn tại.

Danh sách tuyến tính: Các phép toán

6. Prev (*p*, *L*)

- trả lại vị trí trước *p*
- Prev là không xác định nếu *p* là 1 hoặc nếu *L* không có vị trí *p*.

7. Makenull (*L*)

- hàm này biến *L* trở thành danh sách rỗng và trả lại vị trí *END*(*L*)

8. First (*L*)

- trả lại vị trí đầu tiên trong *L*. Nếu *L* là rỗng hàm này trả lại END(*L*).

9. Printlist (*L*)

- In ra danh sách các phần tử của *L* theo thứ tự xuất hiện.

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

3.3.2. Các cách cài đặt danh sách tuyến tính
(Implementations of Linear List)

- Dùng mảng (Array-based)
  - Cất giữ các phần tử của danh sách vào các ô liên tiếp của mảng
- Danh sách liên kết (Linked list / Pointer-based)
  - Các phần tử của danh sách có thể cất giữ ở các chỗ tuỳ ý trong bộ nhớ.
  - Mỗi phần tử có con trỏ (hoặc móc nối - link) đến phần tử tiếp theo
- Địa chỉ không trực tiếp (Indirect addressing)
  - Các phần tử của danh sách có thể cất giữ ở các chỗ tuỳ ý trong bộ nhớ
  - Tạo bảng trong đó phần tử thứ *i* của bảng cho biết nơi lưu trữ phần tử thứ *i* của danh sách
- Mô phỏng con trỏ (Simulated pointer)
  - Tương tự như biểu diễn liên kết nhưng các số nguyên được thay bởi con trỏ của C++

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

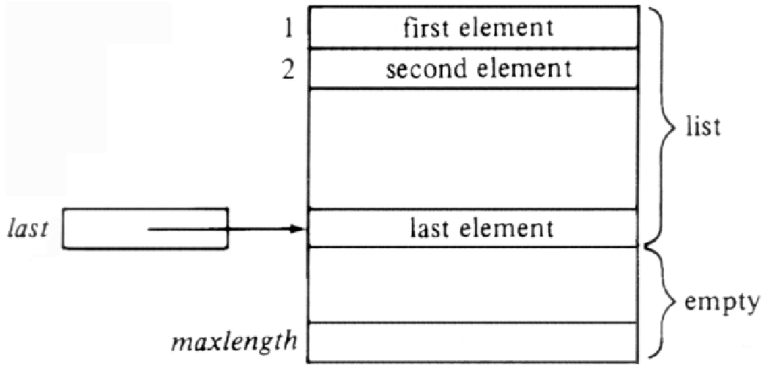
3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

3.3.2.1. Biểu diễn dưới dạng mảng
(Array-based Representation of Linear List)

- Ta cất giữ các phần tử của danh sách tuyến tính vào các ô (liên tiếp) của mảng (array).
- Danh sách sẽ là cấu trúc gồm hai thành phần.
  - Thành phần 1 : là mảng các phần tử
  - Thành phần 2 : **last** - cho biết vị trí của phần tử cuối cùng trong danh sách
- Vị trí có kiểu nguyên (integer) và chạy trong khoảng từ 0 đến *maxlength*-1 . Hàm END(L) trả lại giá trị last+1.



Biểu diễn dưới dạng mảng - C
(Array-based Representation of Linear List)

```
# define maxlength 1000
typedef int elementtype; /* elements are integers */
typedef struct list_tag {
                elementtype elements [maxlength];
                int last;
        } list_type;
```

Hàm End(L)

```
int end (list_type *lp)
   {
       return (lp->last +1)
    }
```

Cài đặt Insert

Insert (x, p,L): Chèn *x* vào vị trí *p* trong danh sách

```
void insert (elementtype x , int p , list_type * lp)
{   int v; // running position
    if (lp -> last >= maxlength - 1)
    {    printf("\n%s ","list is full");
         return;
    }
    if ((p < 0) || (p > lp -> last + 1))
    {    printf("\n%s ","position does not exist");
         return;
    }
    else {
          for (int q = lp -> last; q >= p; q--)
             lp -> elements [q+1] = lp -> elements [q];
          lp -> last = lp -> last +1 ;
          lp -> elements[p] = x;
         }
}
```

Cài đặt DELETE

Delete (p, L): loại phần tử ở vị trí *p* trong danh sách *L*

```
void deleteL(int p , list_type * lp)
{
   int q; /* running position */
   if ((p > lp-> last) || (p < 0))
   {     printf("\n%s ","position does not exist");
         return;
   }
   else /* shift elements */ {
      lp -> last --;
      for (int q = p; q <= lp ->last; q++)
         lp -> elements [q] = lp -> elements [q+1];
   }
}
```

Cài đặt LOCATE

Locate (x, L): trả lại vị trí của *x* trong danh sách *L*

```
int locate (elementtype x , list_type * lp)
{
    int q;
    for (q = 0; q <= lp ->last; q++)
    if (lp -> elements [q] == x)
        return (q);
    return (lp -> last + 1); /* if not found */
}
```

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

Phân tích cách biểu diễn dưới dạng mảng
(Array-based Representation of Linear List)

- Có thể nhận thấy một số ưu - khuyết điểm sau đây của cách tổ chức lưu trữ này:
  - Cách biểu diễn này rất tiện cho việc truy xuất đến các phần tử của danh sách.
  - Do danh sách là biến động, số phần tử trong danh sách là không biết trước. Nên ta thường phải khai báo kích thước tối đa cho mảng để dự phòng (*maxlength*). Điều này dẫn đến lãng phí bộ nhớ.
  - Các thao tác chèn một phần tử vào danh sách và xoá bỏ một phần tử khỏi danh sách được thực hiện chậm (với thời gian tuyến tính đối với kích thước danh sách)

Lưu trữ móc nối đối với danh sách tuyến tính
Linked List)

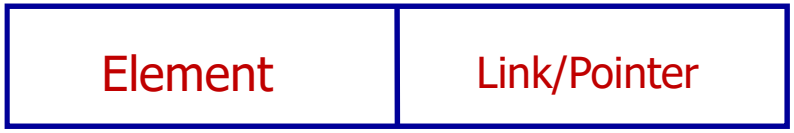
- Lưu trữ kế tiếp có những nhược điểm cơ bản đã được phân tích ở trên: đó là việc bổ sung và loại trừ phần tử là rất tốn kém thời gian, ngoài ra phải kể đến việc sử dụng một không gian liên tục trong bộ nhớ. Việc tổ chức con trỏ (hoặc móc nối) để tổ chức danh sách tuyến tính - mà ta gọi là danh sách móc nối là giải pháp khắc phục nhược điểm này, tuy nhiên cái giá mà ta phải trả là bộ nhớ dành cho con trỏ.
- Ta sẽ xét các cách tổ chức danh sách móc nối sau đây:
  - Danh sách nối đơn (Singly linked list)
  - Danh sách nối vòng (Circulary linked list)
  - Danh sách nối đôi (Doubly Linked List)

Khi nào dùng danh sách móc nối

- **Khi không biết kích thước của dữ liệu - hãy dùng con trỏ và bộ nhớ động (***Unknown data size – use pointers & dynamic storage***).**
- **Khi không biết kiểu dữ liệu - hãy dùng con trỏ void (***Unknown data type – use void pointers***).**
- **Khi không biết số lượng dữ liệu - hãy dùng danh sách móc nối (***Unknown number of data – linked structure***).**

Danh sách móc nối đơn
(Singly linked list)

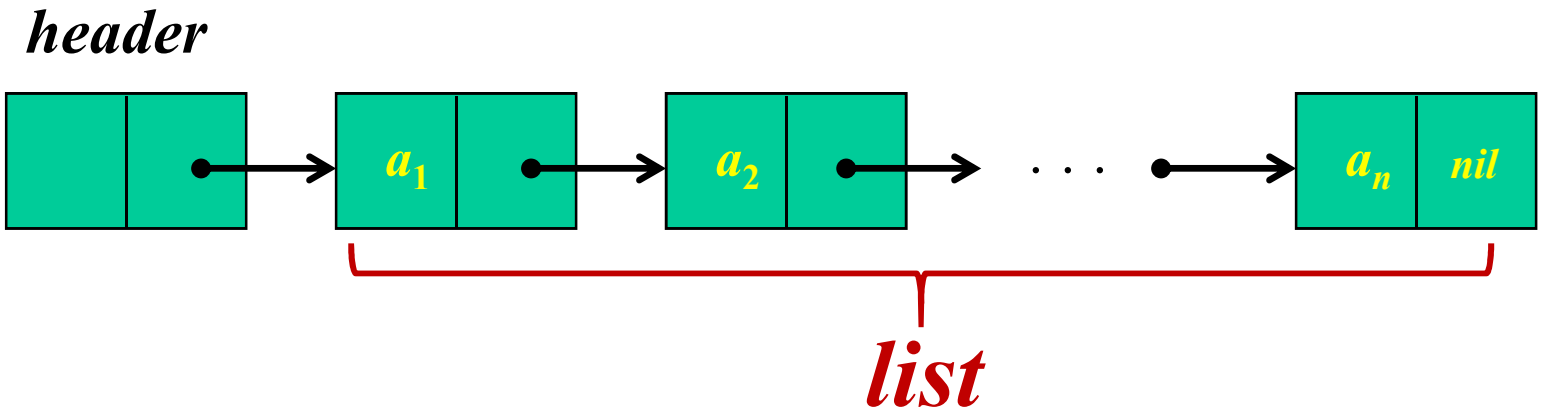
- Trong cách biểu diễn này, danh sách bao gồm các ô (các nút - node), mỗi ô chứa một phần tử của danh sách và con trỏ đến ô tiếp theo của danh sách.
- Nếu danh sách là $a_1, a_2, \ldots, a_n$, thì ô lưu trữ a_i có *con trỏ* (mối nối) đến ô lưu trữ a_{i+1} với $i = 1, 2, \ldots, n\text{-}1$. Ô lưu trữ a_n có con trỏ rỗng, mà ta sẽ ký hiệu là **nil**. Như vậy mỗi ô có cấu trúc:



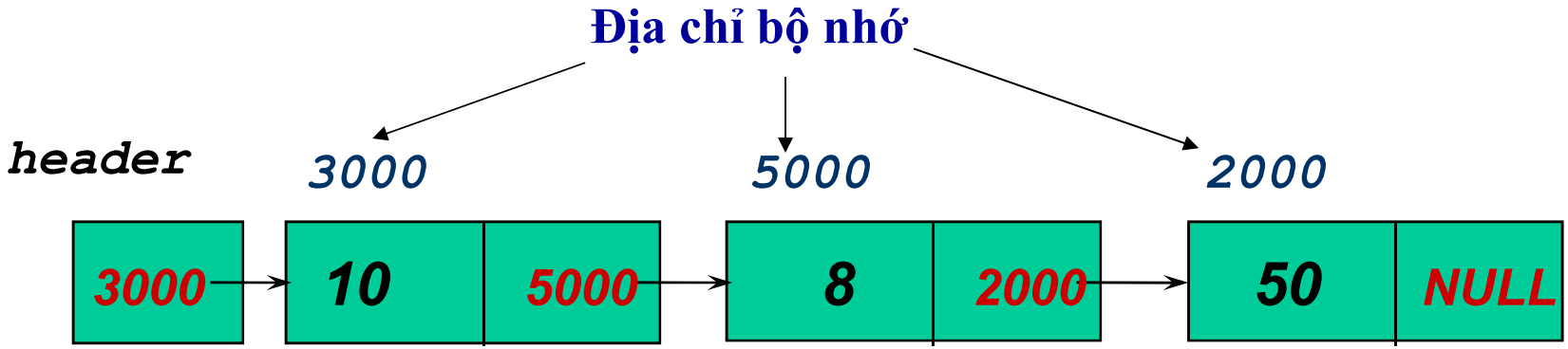
- Có một ô đặc biệt gọi là ô *header* để trỏ ra ô chứa phần tử đầu tiên trong danh sách (a_1); Ô header không lưu trữ phần tử nào cả. Trong trường hợp danh sách rỗng, con trỏ của header là **nil**, và không có ô nào khác.
- Các ô có thể nằm ở vị trí bất kỳ trong bộ nhớ

Danh sách móc nối đơn
(Singly linked list)

- Danh sách móc nối được tổ chức như trong hình vẽ sau:



- *Mối nối chỉ ra địa chỉ bộ nhớ của nút tiếp theo trong danh sách*



Danh sách móc nối đơn
Singly linked list)

- **Mô tả danh sách nối đơn trong C:**

```
typedef  <Kiểu dữ liệu phần tử> ElementType;
struct NodeType{
            ElementType Inf;
            NodeType *Next; };
typedef struct NodeType LIST;
```

- Khai báo trên định nghĩa kiểu NodeType, trong đó NodeType là kiểu bản ghi mô tả một nút gồm hai trường:
  - Inf - lưu dữ liệu có kiểu là ElementType (đã được định nghĩa, có thể gồm nhiều thành phần)
  - Next thuộc kiểu NodeType, lưu địa chỉ của nút kế tiếp.
- Cần có biến con trỏ First lưu địa chỉ của nút đầu tiên trong danh sách:

```
NodeType *First;
```

Hàm INSERT(x,p)

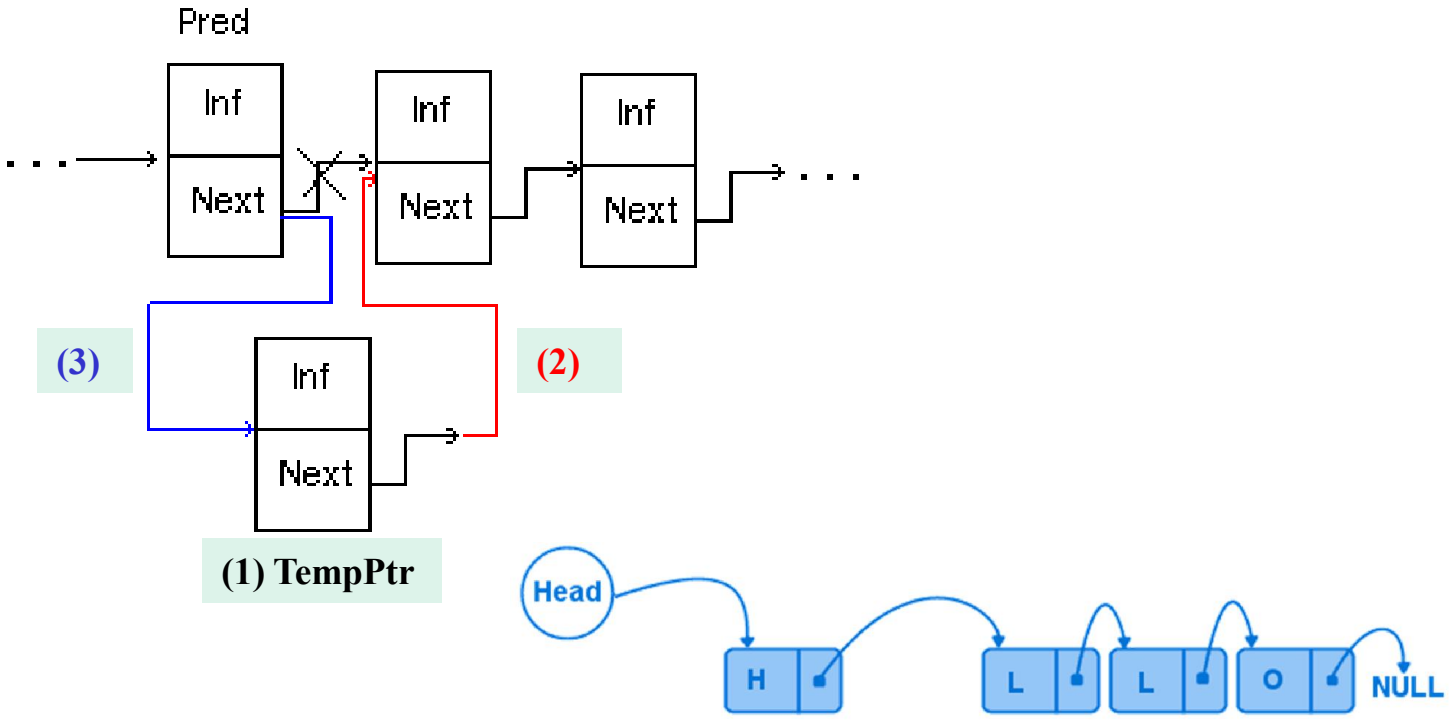
- Giả sử cần chèn một phần tử có nội dung dữ liệu là ***x*** (có kiểu ElementType) vào danh sách. Vị trí cần chèn được xác định là sau nút được trỏ bởi con trỏ **Pred**. Thao tác được tiến hành theo các bước:
  - **(1)** Xin cấp phát một nút mới cho con trỏ **TempNode** để lưu ***x***,
  - Nối nút này vào danh sách tại vị trí cần chèn:

    (2) TempNode->Next bằng **Pred->Next**

    (3) ghi nhận lại **Pred->Next** bằng **TempNode.**

Hàm INSERT(x,p)

- Sơ đồ của thao tác cần làm với danh sách được minh hoạ như sau:



Chú ý:

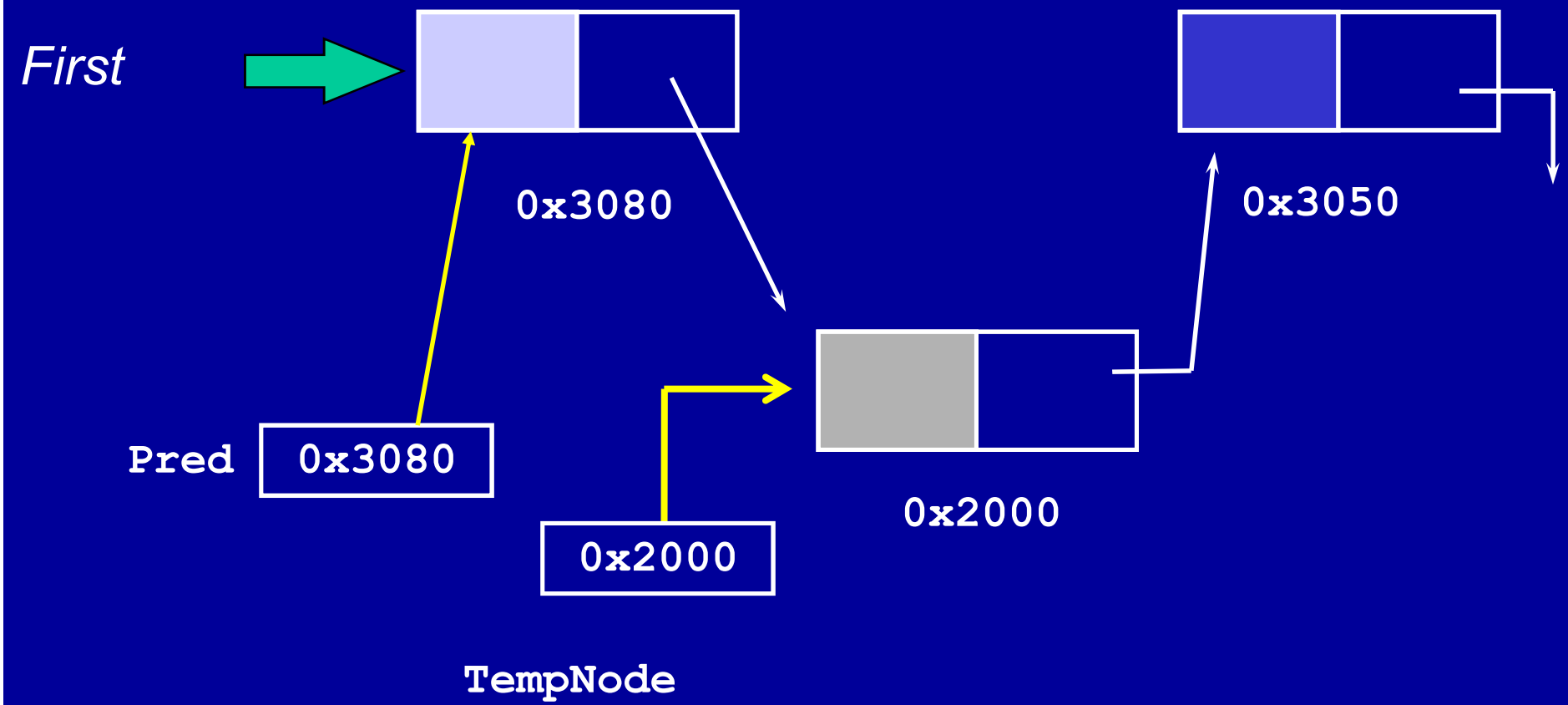
- Việc chèn một nút không ảnh hưởng đến mối liên kết của các nút đứng sau vị trí chèn, do đó không phải thực hiện dồn phần tử như trong cài đặt mảng.

Hàm INSERT(x,p): Cài đặt trên C

```
NodeType *Insert_Middle(NodeType *Pred, ElementType X)
{ Chèn một nút mới với thông tin X vào sau nút được trỏ bởi Pred }
{  NodeType *TempNode;
   TempNode = (NodeType *) malloc(sizeof(NodeType));   //(1)
   TempNode->Inf=X;                                    //(1)
   TempNode->Next=Pred->Next;                          //(2)
   Pred->Next=TempNode;                                //(3)
   return TempNode;
}
```

Hàm INSERT(x,p)

Hàm Insert_Middle



Chèn vào đầu danh sách

```
// Chèn một nút vào đầu danh sách được trỏ bởi First
NodeType *Insert_ToHead(NodeType *First, ElementType X)
{  NodeType *TempNode;
   TempNode = (NodeType *) malloc(sizeof(NodeType));
   TempNode->Inf=X; TempNode->Next=First;
   First=TempNode;
   return First;
}
```

Chèn vào cuối danh sách

// Chèn một nút vào cuối danh sách được trỏ bởi head. Giả sử danh sách khác rỗng

```
NodeType *Insert_ToLast(NodeType *head, ElementType X)
{   NodeType *NewNode; NodeType *TempNode;
    NewNode = (NodeType *) malloc(sizeof(NodeType));
    NewNode->Inf=X; TempNode=head;
    while (TempNode->next != NULL) // go to the end of a list
          TempNode= TempNode->next;
    TempNode->next = NewNode;
    return head;
}
```

Chú ý: *Nếu thao tác này phải thực hiện thường xuyên thì nên đưa vào con trỏ Last trỏ đến nút cuối cùng của danh sách*

Hàm DELETE(p)

- Giả sử cần loại bỏ nút đứng sau nút đang được trỏ bởi **Pred.** Việc xoá chỉ tiến hành khi danh sách là khác rỗng (Cần phải kiểm tra điều kiện này trước khi thực hiện thao tác xoá).
- Sử dụng con trỏ **TempPtr** để ghi nhận nút cần xóa, thao tác xoá được tiến hành theo các bước sau:

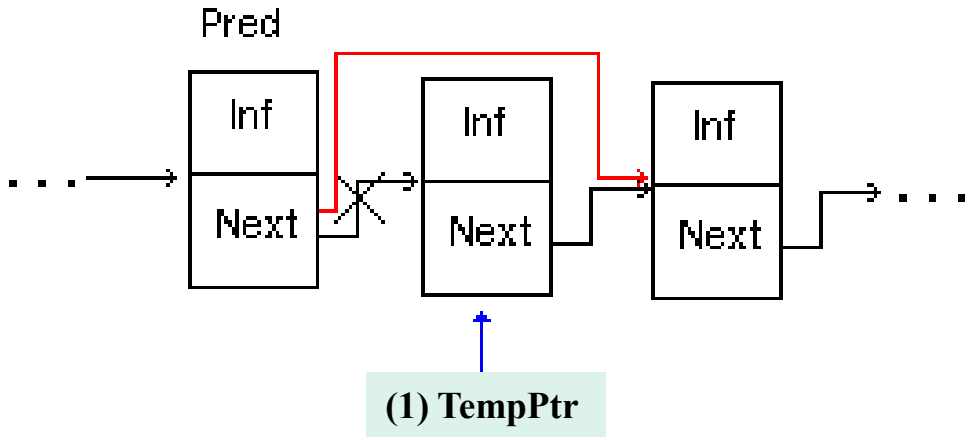
  (1) Gán **TempPtr** bằng **Pred->Next.**

  (2) Đặt lại **Pred->Next** bằng **TempPtr->Next**.

  (3) Thu hồi vùng nhớ được trỏ bởi **TempPtr.**

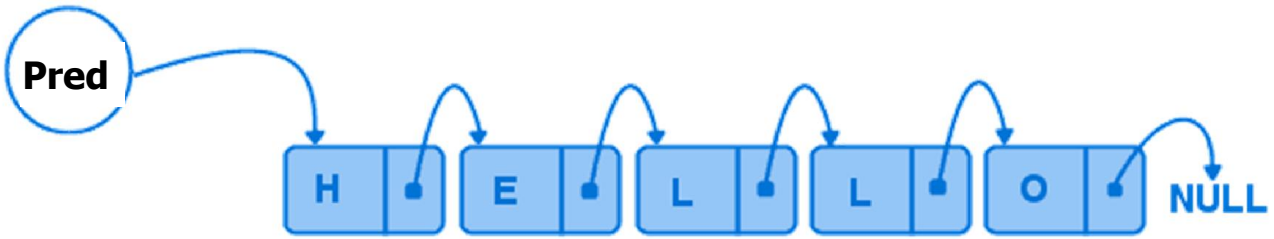
Hàm DELETE(p)

- Sơ đồ của thao tác cần làm với danh sách được minh hoạ trong hình dưới đây:



Hàm DELETE(p): Cài đặt trên C

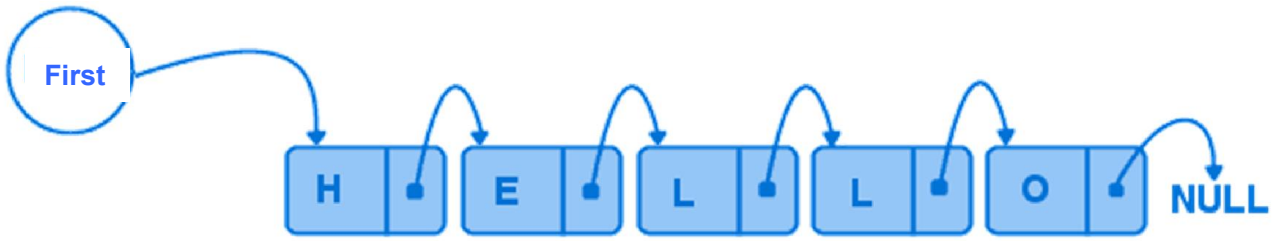
```
ElementType Delete(NodeType *Pred)
{  Xoá nút ở sau nút được trỏ bởi Pred và trả lại giá trị ở nút bị xoá  }
{  ElementType X; NodeType *TempNode;
   TempNode=Pred->Next;                    //(1)
   Pred->Next=Pred->Next->Next;            //(2)
   X=TempNode->Inf;
    free(TempNode);                        //(3)
   return X;
}
```



Xoá phần tử ở đầu danh sách

// Xoá nút ở đầu danh sách được trỏ bởi First

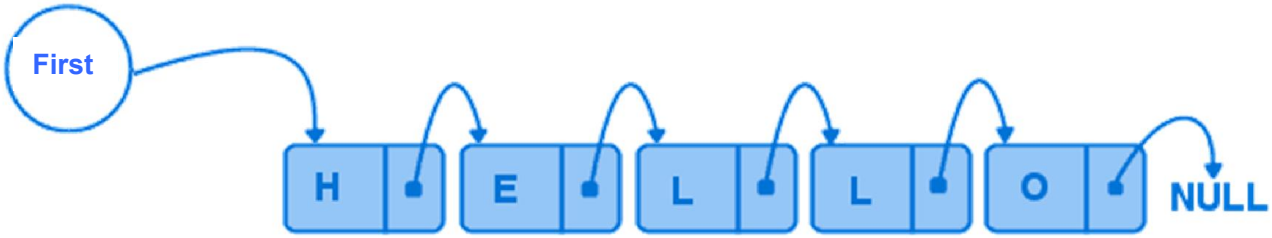
```
NodeType *Delete_Head(NodeType *First)
{ NodeType *TempNode;
  TempNode=First->Next;
  free(First);
  return TempNode;
}
```



Xoá phần tử ở cuối danh sách

```
// Xoá nút cuối của danh sách. Giả sử danh sách khác rỗng
NodeType *Delete_Last(NodeType *First)
{  NodeType *TempNode1, *TempNode2;
   TempNode1= First; TempNode2= First;
    while (TempNode1->next != NULL) // Go to the end of a list
    {  TempNode2 = TempNode1;
       TempNode1= TempNode1->next; }
    TempNode2->next = NULL;
    free(TempNode1);
   return First;
}
```

Chú ý: *Nếu thao tác này phải thực hiện thường xuyên thì nên đưa vào con trỏ Last trỏ đến nút cuối cùng của danh sách*



Cài đặt một số thao tác thường dùng khác

- Chèn một nút vào đầu danh sách
- Chèn một nút vào cuối danh sách
- Xoá nút ở đầu danh sách
- Xoá nút ở cuối danh sách
- Kiểm tra danh sách rỗng
- Huỷ danh sách

Tìm kiếm

- Tìm kiếm trên danh sách móc nối được trỏ bởi *head,* trả lại con trỏ đến nút chứa giá trị *e*.

```
NodeType *Search(NodeType *head, ElementType e)
{
   while (head->inf != e) head=head->next);
   return head;
}
```

IsEmpty và PrintLIST

- **Hàm kiểm tra danh sách rỗng**

```
int IsEmpty(NodeType *head){
        return !head;
}
```

- **Huỷ danh sách**

```
NodeType *MakeNull(NodeType *head){
   while (!IsEmpty(head)) head=Delete_Head(head);
   return head;
}
```

- **Đưa ra toàn bộ thông tin ở các nút**

```
void Print(NodeType *head)
{  NodeType *TempNode;
   TempNode=head; int count = 0;
   while (TempNode) {
       printf("%6d", TempNode->Inf); count++;
       TempNode=TempNode->Next;
       if (count % 12 == 0) printf("\n");
   }
    printf("\n");
}
```

Ví dụ 1. Chương trình minh hoạ trên C

- Xây dựng chương trình thực hiện công việc sau đây:
  - Tạo ngẫu nhiên một danh sách với các phần tử là các số nguyên. Sau đó:
  - Từ danh sách tạo được xây dựng hai danh sách: một danh sách chứa tất cả các số dương còn danh sách kia chứa tất cả các số âm của danh sách ban đầu.
- Chương trình dưới đây sẽ xây dựng một số phép toán cơ bản đối với danh sách móc nối để thực hiện các công việc đặt ra.

Ví dụ 1. Chương trình trên C

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef  long  ElementType;
struct PointerType{
       ElementType Inf;
       PointerType *Next; };
PointerType *Insert_ToHead(PointerType *First,
   ElementType X)
{  PointerType *TempNode;
   TempNode = (PointerType *) malloc(sizeof(PointerType));
   TempNode->Inf=X; TempNode->Next=First;
   First=TempNode;
   return First;
}
```

Ví dụ 1.

```
PointerType *Insert_Middle(PointerType *Pred, ElementType X)
{  PointerType *TempNode;
   TempNode = (PointerType *) malloc(sizeof(PointerType));
   TempNode->Inf=X;
   TempNode->Next=Pred->Next;
   Pred->Next=TempNode;
   return TempNode;
}
```

Ví dụ 1.

```
PointerType *Delete_Head(PointerType *First)
{  PointerType *TempNode;
   TempNode=First->Next;
   free(First);
   return TempNode;
}

ElementType Delete(PointerType *Pred)
{  ElementType X;
   PointerType *TempNode;
   TempNode=Pred->Next;
   Pred->Next=Pred->Next->Next;
   X=TempNode->Inf; free(TempNode);
   return X;
}
```

Ví dụ 1.

```
void Print(PointerType *First)
{  PointerType *TempNode;
   TempNode=First; int count = 0;
   while (TempNode) {
       printf("%6d", TempNode->Inf); count++;
       TempNode=TempNode->Next;
       if (count % 12 == 0) printf("\n");
   } printf("\n");
}
int IsEmpty(PointerType *First) {
     return !First;
}
PointerType *MakeNull(PointerType *First) {
   while (!IsEmpty(First)) First=Delete_Head(First);
   return First;
}
```

Ví dụ 1.

```
int main() {
   PointerType *S1, *S2, *S3, *V1, *V2, *V3;
   ElementType a; int i, n;
   clrscr(); randomize(); S1=NULL;
   // Tao phan tu dau tien
   a=-100+random(201);
   S1=Insert_ToHead(S1, a);
   printf("Nhap vao so luong phan tu n = ");
   scanf("%i", &n);  printf("\n");
   // Tao ngau nhien danh sach va dua ra man hinh
   V1=S1;
   for (i=2; i<=n; i++) {
      a=-100+random(201);
      V1=Insert_Middle(V1, a);
   }
   printf("====>  Danh sach ban dau: \n"); Print(S1);
   printf("\n");
```

Ví dụ 1.

```
    V1 = S1;  S2 = NULL; S3 = NULL;
    while (V1) {
      if (V1->Inf > 0)
        if (!S2) {  S2=Insert_ToHead(S2, V1->Inf); V2 = S2; }
        else { Insert_Middle(V2, V1->Inf); V2 = V2->Next; }
      if (V1->Inf < 0)
          if (!S3) { S3=Insert_ToHead(S3, V1->Inf); V3 = S3;}
          else { Insert_Middle(V3, V1->Inf); V3 = V3->Next;}
      V1= V1->Next;
    }
    printf("====>  Danh sach so duong: \n"); Print(S2);
    printf("\n");
    printf("====>  Danh sach so am: \n"); Print(S3);
    printf("\n");
    S1=MakeNull(S1); S2=MakeNull(S2); S3=MakeNull(S3);
    getchar(); getchar();
}
```

Ví dụ 2.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{  int        data;
   struct node *next;
};
typedef struct node node;

void printlist(node* head);

int main()
{ node a, b, c;
   node* pcurr;
   node* phead;
   node* pnew;   node* pdel;
   int i;
```

```
/*** A: Static Memory Allocation ***/
  printf("Static Memory Allocation:\n");
  /* initialise nodes */
  a.data = 1;
  b.data = 2;
  c.data = 3;
  a.next = b.next = c.next = NULL;

  /* link to form list a - b - c */
  a.next = &b;    b.next = &c;

  printlist(&a);
```

Ví dụ (tiếp)

```
/*** B: Dynamic Memory Allocation ***/
  printf("\n\nDynamic Memory Allocation:\n");
 /* Tạo nút đầu tiên với dữ liệu = 10 */
  if ((phead = (node*)malloc(sizeof(node))) ==
     NULL)
    exit(1); // lỗi phân bổ bộ nhớ
  phead->data = 10;
  pcurr = phead; // khởi tạo con trỏ đến nút đầu tiên
/* Tạo tiếp 5 nút, với dữ liệu = 20, 30, ... */
  for (i=20; i<=60; i+=10)
  {    // phân bổ bộ nhớ
   if ((pnew = (node*)malloc(sizeof(node))) ==
     NULL)
      exit(1); // lỗi phân bổ bộ nhớ
       // tạo dữ liệu
   pnew->data = i;
// nối nút hiện tại với nút mới,
// và đặt nút mới thành nút hiện tại
    pcurr->next = pnew;
    pcurr = pnew;
  }
 /*  phần tử cuối LIST có  next là NULL */
 /* Chú ý là pcurr  trỏ đến nút cuối của danh sách */
  pcurr->next = NULL;
 /* Ta có thể in danh sách */
  printlist(phead);
/*** C: Insert và Delete Nodes ***/
  printf("\n\nInsert and Delete Nodes:\n");
 /* Insert a new node (data = 100) after the third one */
 // Tạo và gán giá trị nút mới
  if ((pnew = (node*)malloc(sizeof(node)))==
     NULL)
     exit(1);
  pnew->data = 100;
```

Ví dụ (tiếp)

```
// Làm nút hiện tại trở thành nút mà ta sẽ chèn vào sau nó
  pcurr = phead->next->next;
 //Bắt nút mới trỏ đến nút được trỏ bởi nút hiện tại
  pnew->next = pcurr->next;
 // Đặt nút hiện tại trỏ đến nút mới
  pcurr->next = pnew;
 /* Loại bỏ nút thứ hai*/
 // Làm nút hiện tại trở thành nút đứng trước nút cần xoá
  pcurr = phead;
  pdel = pcurr->next;
 // Loại nút bởi việc bỏ qua nó
  pcurr->next = pcurr->next->next;
 // Giải phóng bộ nhớ của nút bị bỏ qua
  free(pdel);
/* Lại đưa ra danh sách */
  printlist(phead);
  return 0;
}
```

```
void printlist(node* head)
{
/* duyệt qua danh sách, đưa ra dữ liệu của mỗi nút */

  node* curr = head;  // start at head
  while (curr != NULL)
  {
printf("  Dữ liệu của nút hiện tại: %d\n", curr-
     >data);
    curr = curr->next;
  }
 getchar();
}
```

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

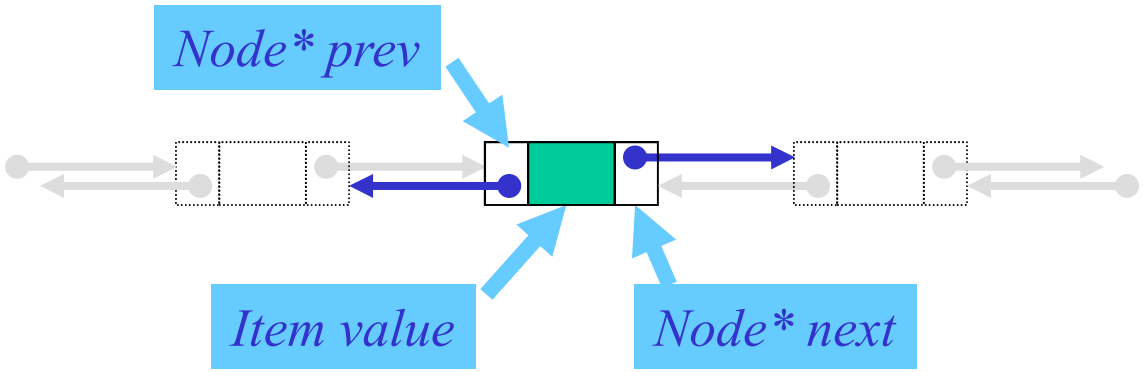
3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

Danh sách nối đôi
(Doubly linked list)

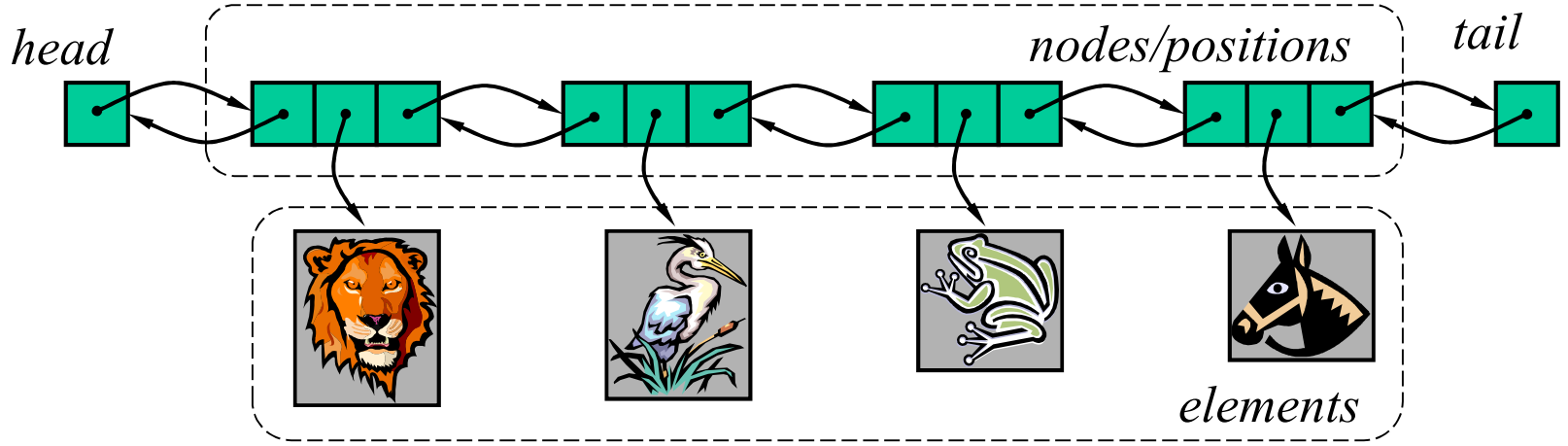
- Trong nhiều ứng dụng ta muốn duyệt danh sách theo cả hai chiều một cách hiệu qủa. Hoặc cho một phần tử, ta cần xác định cả phần tử đi trước lẫn phần tử đi sau nó trong danh sách một cách nhanh chóng.
- Trong tình huống như vậy ta có thể gán cho mỗi ô trong danh sách con trỏ đến cả phần tử đi trước lẫn phần tử đi sau nó trong danh sách.



- Cách tổ chức này được gọi là ***danh sách nối đôi.***

Danh sách nối đôi
(Doubly linked list)

- Cách tổ chức ***danh sách nối đôi*** được minh hoạ trong hình vẽ sau:



- Có hai nút đặc biệt: tail (đuôi) và head (đầu)
  - head có con trỏ trái prev là null
  - tail có con trỏ phải next là null
- Các phép toán cơ bản được xét tương tự như danh sách nối đơn.

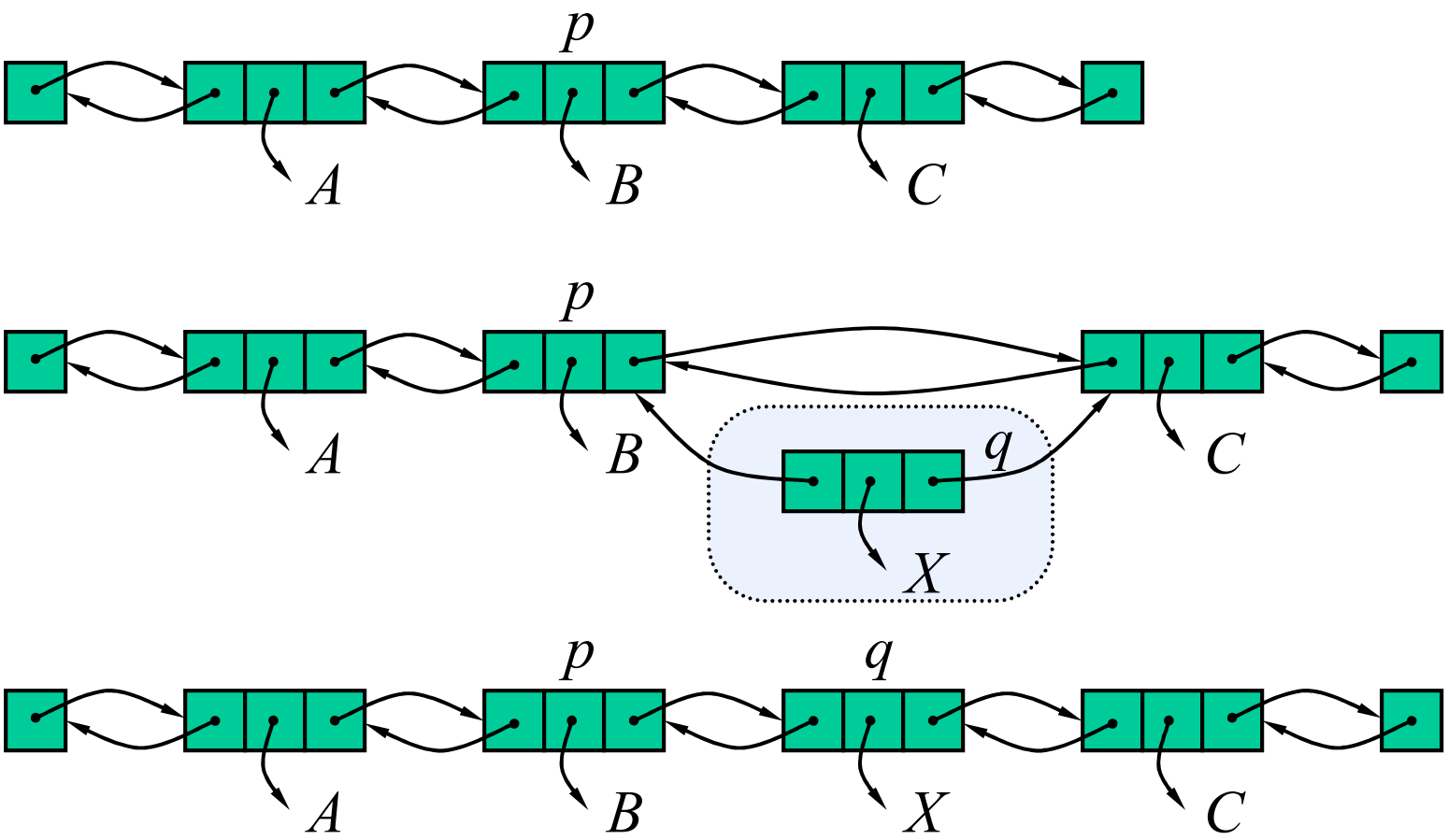
Danh sách nối đôi
(Doubly linked list)

- Cách mô tả ***danh sách nối đôi*** trên C:

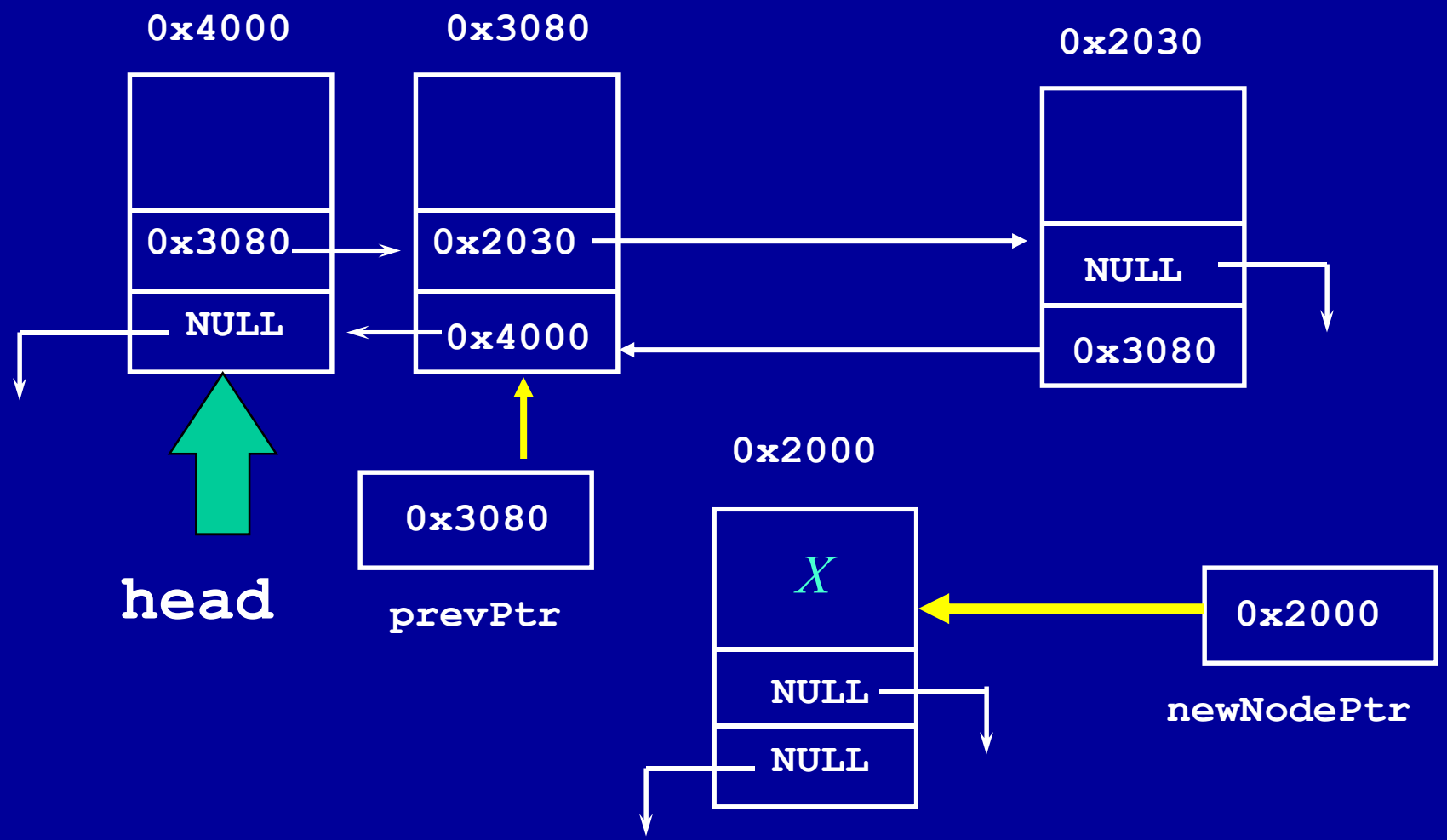
```
struct dllist {
 int number;
 struct dllist *next;
 struct dllist *prev;
};

struct dllist *head, *tail;
```

Phép toán chèn (Insertion)

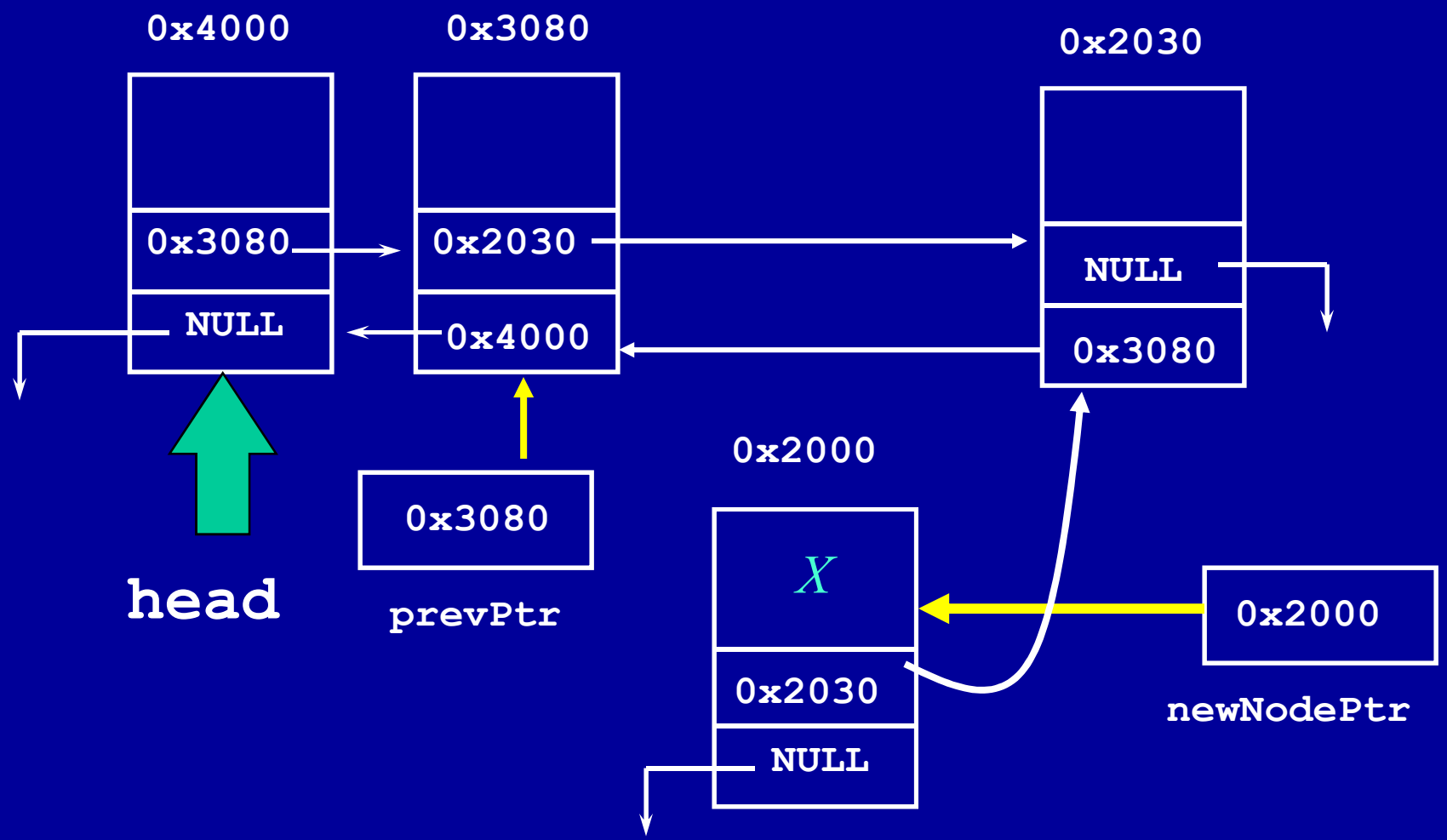
- Ta mô tả phép toán chèn insertAfter(p, X)



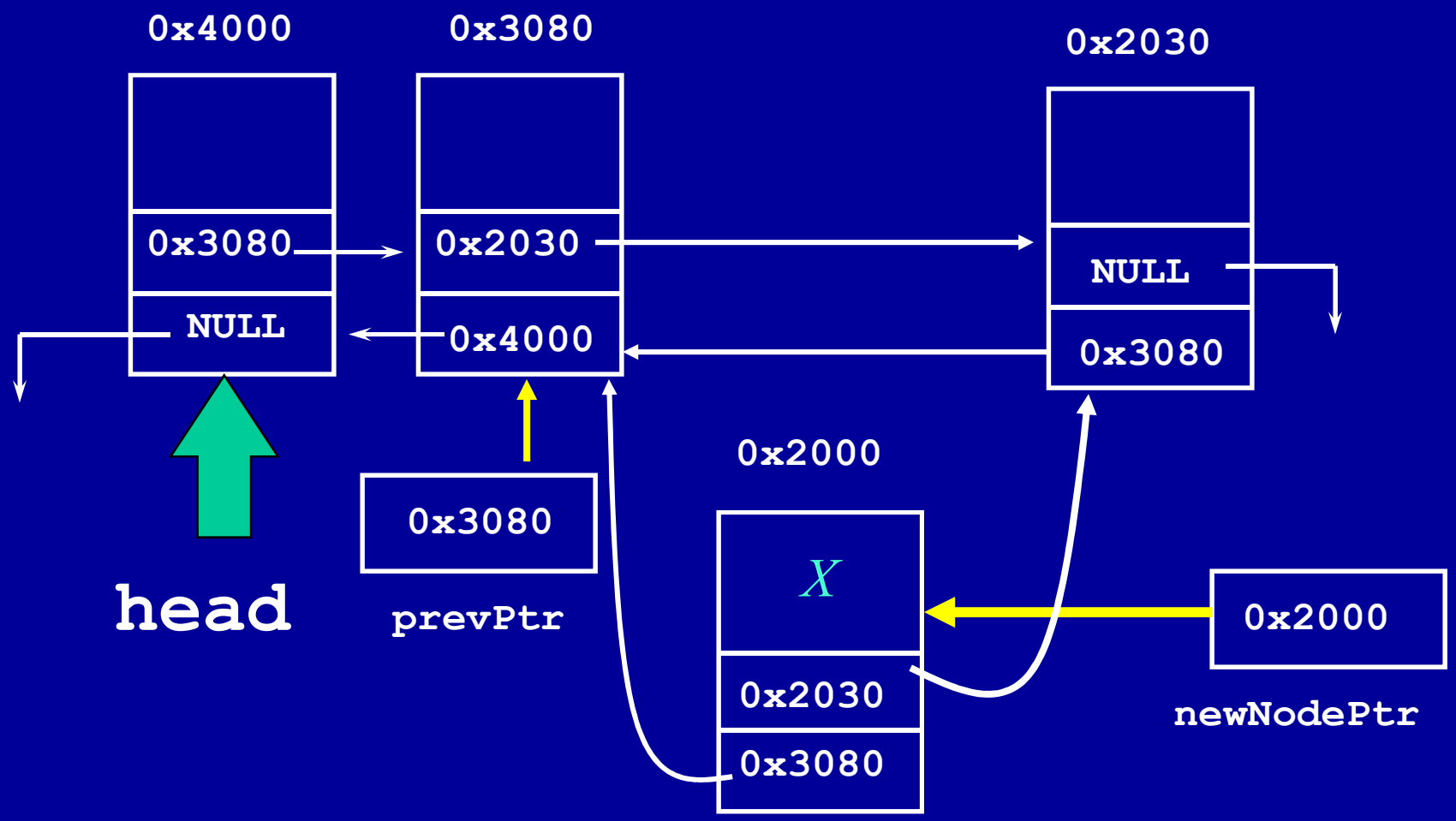
Chèn sau prevPtr: Insert(X,prevPtr)



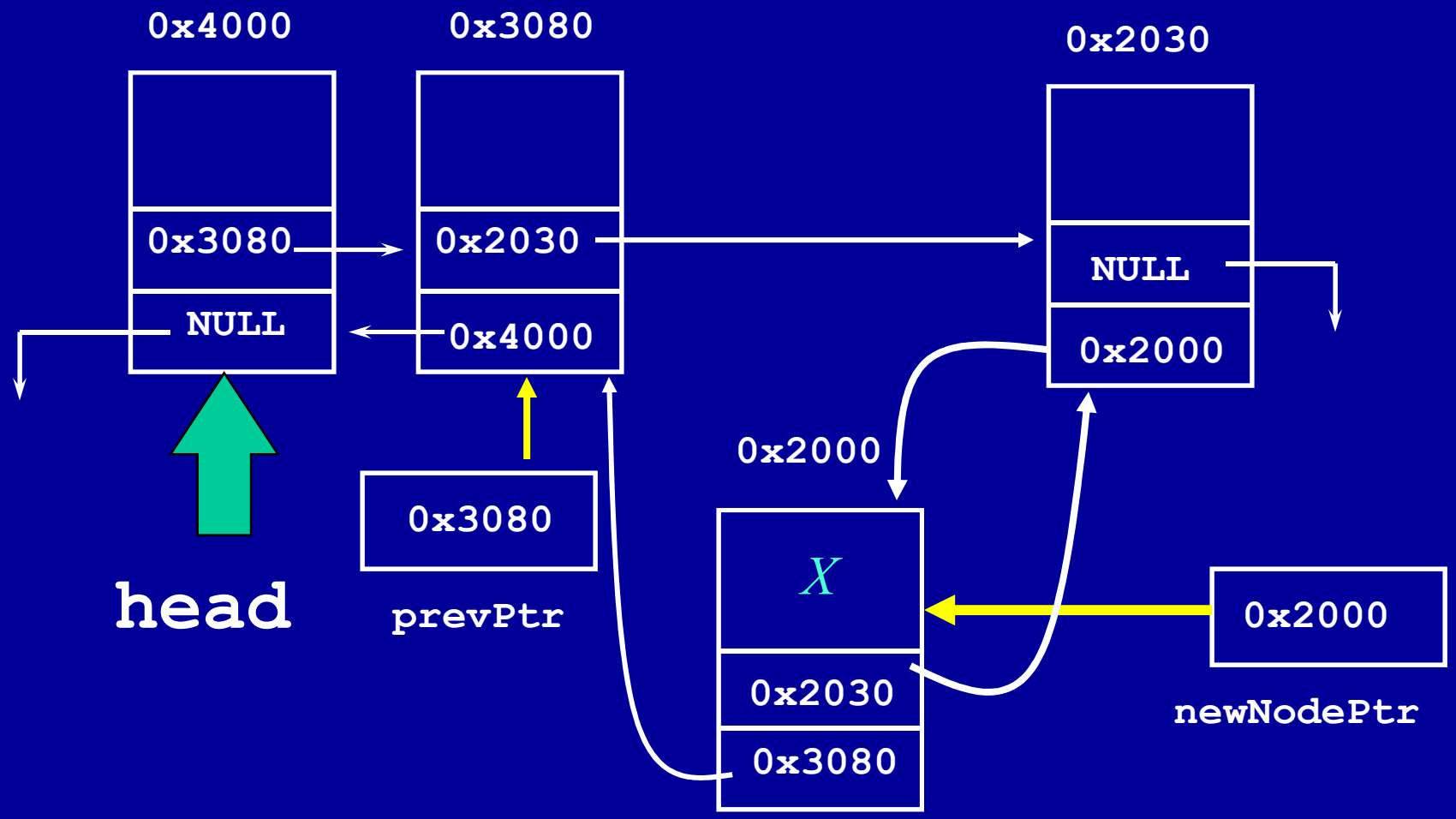
Chèn sau prevPtr: Insert(X,prevPtr)



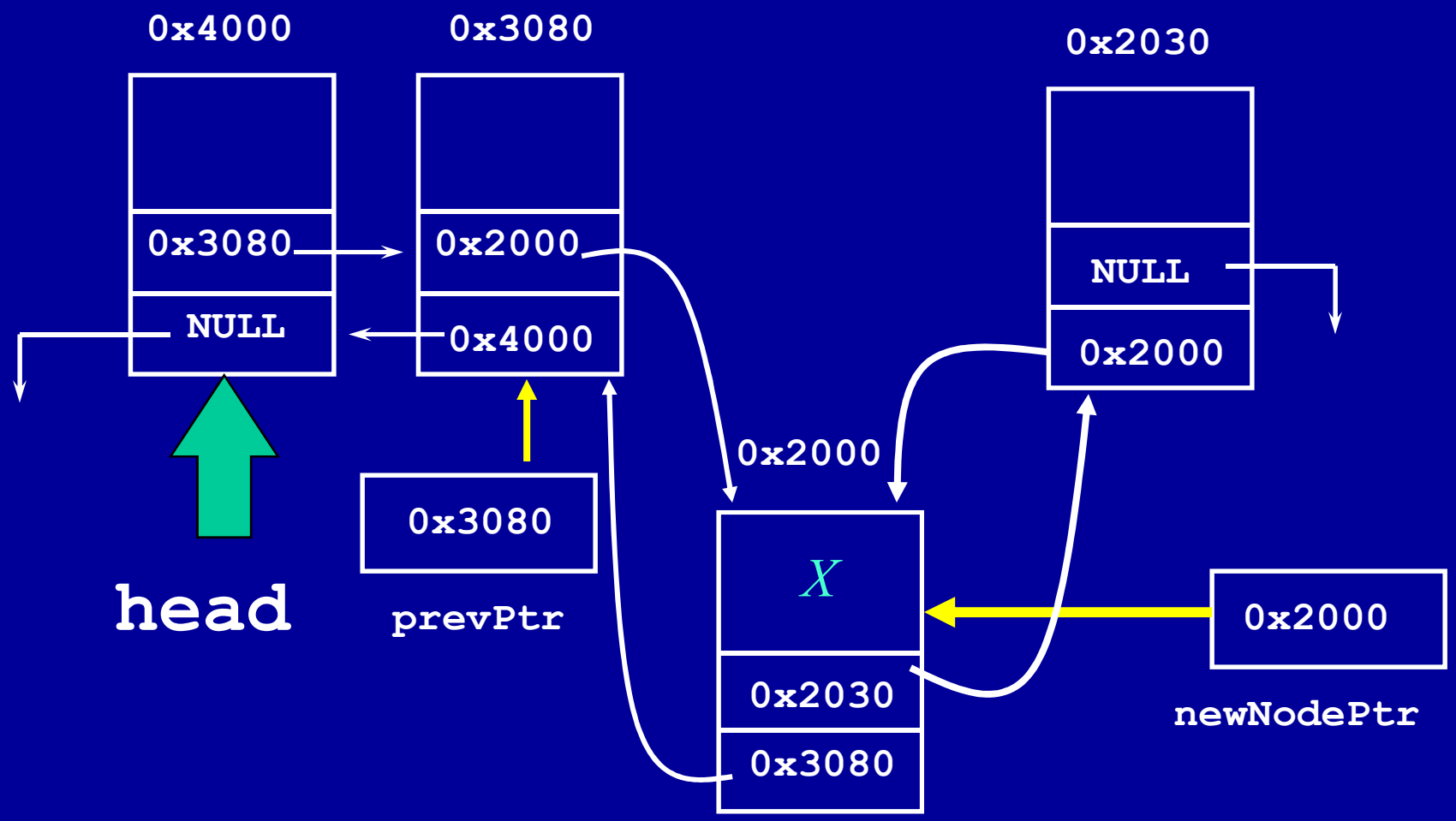
Chèn sau prevPtr: Insert(X,prevPtr)



Chèn sau prevPtr:Insert(X,prevPtr)

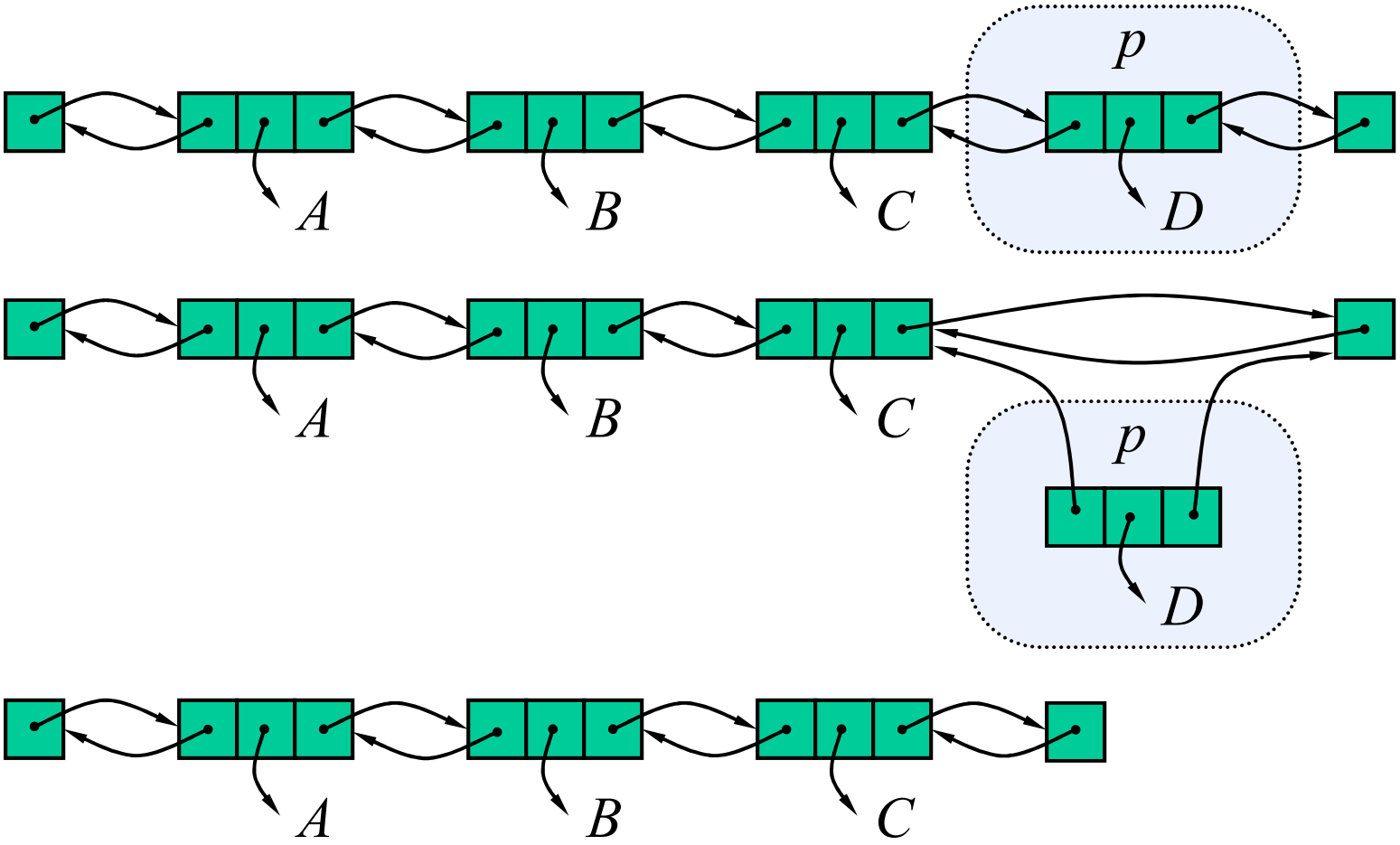


Chèn sau prevPtr: Insert(X,prevPtr)



Phép toán Xoá (Deletion)

- Ta mô tả phép toán remove(p), trong đó p = last()



Chương trình trên C minh hoạ một số phép toán

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

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

/* Nối đuôi một phần tử mới */
void append_node(struct dllist *lnode);
/* Chèn một phần tử mới vào sau ô trỏ bởi after */
void insert_node(struct dllist *lnode, struct dllist *after);
/* Xoá ô trỏ bởi lnode */
void remove_node(struct dllist *lnode);
```

Chương trình minh hoạ trên C

```
int main(void) {
struct dllist *lnode; int i = 0;
/* add some numbers to the double linked list */
for(i = 0; i <= 5; i++) {
   lnode = (struct dllist *)malloc(sizeof(struct dllist));
   lnode->number = i;
   append_node(lnode);
}
/* print the dll list forward */
 printf(" Traverse the dll list forward \n");
 for(lnode = head; lnode != NULL; lnode = lnode->next)
 { printf("%d\n", lnode->number); }

 /* print the dll list backward */
printf(" Traverse the dll list backward \n");
for(lnode = tail; lnode != NULL; lnode = lnode->prev)
{  printf("%d\n", lnode->number); }
```

Chương trình minh hoạ trên C

```
/* destroy the dll list */
 while(head != NULL)
    remove_node(head);
    getch(); /* Wait for ...*/
    return 0;
}
void append_node(struct dllist *lnode) {
   if(head == NULL) {
      head = lnode;
      lnode->prev = NULL; }
   else {
     tail->next = lnode;
     lnode->prev = tail;
   }
   tail = lnode;
   lnode->next = NULL;
}
```

Chương trình minh hoạ trên C

```
void insert_node(struct dllist *lnode, struct dllist *after) {
 lnode->next = after->next;
 lnode->prev = after;
 if(after->next != NULL)
     after->next->prev = lnode;
 else
     tail = lnode;
 after->next = lnode;
}
void remove_node(struct dllist *lnode) {
 if(lnode->prev == NULL)
     head = lnode->next;
 else  lnode->prev->next = lnode->next;
 if(lnode->next == NULL)
     tail = lnode->prev;
 else  lnode->next->prev = lnode->prev;
}
```

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

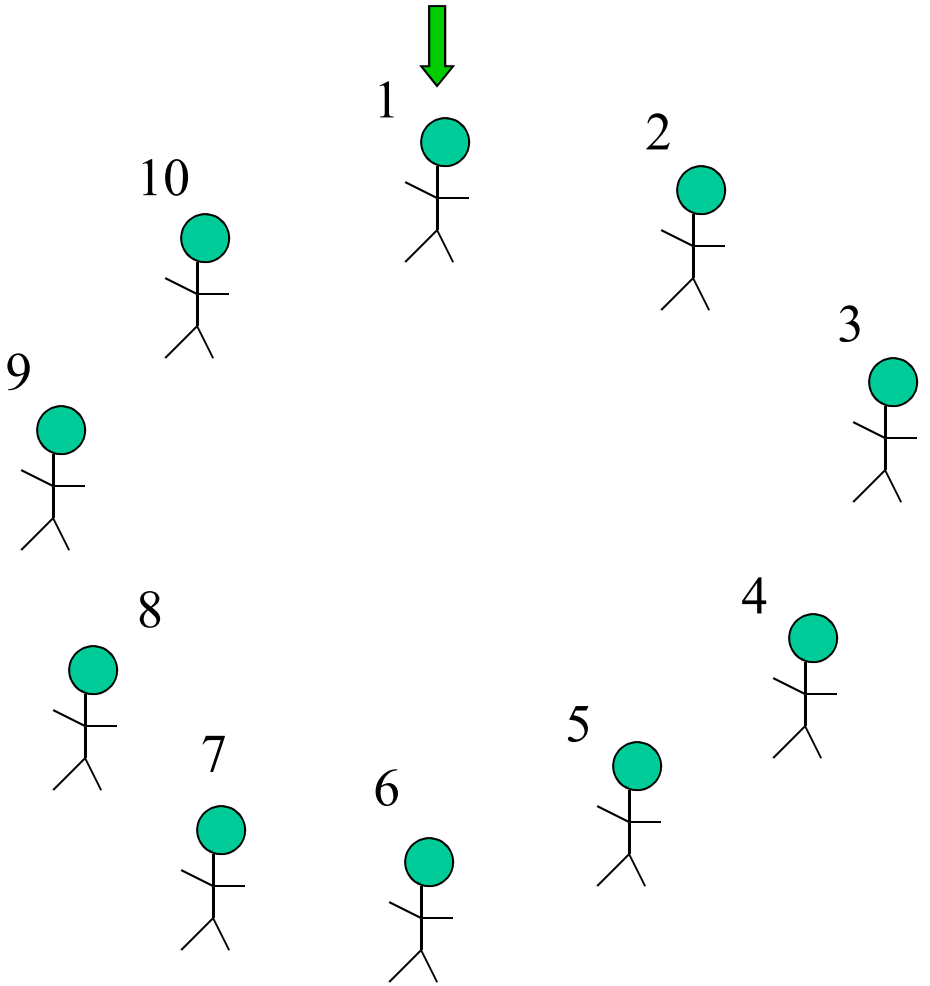
Ví dụ 1. The Josephus Problem

- n khách hàng tham gia vào vòng quay trúng thưởng của Công ty X.
- Các khách hàng được xếp thành một vòng tròn.
- Giám đốc Công ty lựa chọn ngẫu nhiên một số m ($m \leq n$).
- Bắt đầu từ một người được chọn ngẫu nhiên trong số các khách hàng, Giám đốc đếm theo chiều kim đồng hồ và dừng lại mỗi khi đếm đến m.
- Khách hàng ở vị trí này sẽ dời khỏi cuộc chơi.
- Quá trình được lặp lại cho đến khi chỉ còn một người.
- Người cuối cùng còn lại là người trúng thưởng!

Ví dụ

$n = 10$

$m = 5$



Người thắng cuộc!

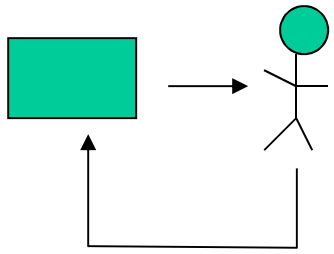
Có thể giải bài toán này nhờ danh sách nối đôi

Thuật toán giải Josephus Problem

```
void josephus(int n, int m){
    // Khai báo danh sách nối kép
    struct dllist { int number;
        struct dllist *next; struct dllist *prev;};
    struct dllist *dList, *curr;
    int i, j;
    // khởi tạo danh sách gồm các khách hàng 1 2 3 ... n
    for (i = 1; i <= n; i++)
        insert(dList, i); // Phải xây dựng hàm này
    // khởi động vòng đếm bắt đầu từ ngưởi 1
    curr = dList->next;
    // thực hiện vòng đếm để loại tất cả chỉ để lại 1 người trong danh sách
    for (i=1; i < n; i++) {
        // đếm bắt đầu từ người hiện tại curr, đi qua m người.
        // ta phải thực hiện điều này m-1 lần.
        for (j=1; j <= m-1; j++)
        {  // con trỏ kế tiếp
            curr = curr->next;
```

Thuật toán (continued)

```
                    // nếu curr dừng tại header, cần thực hiện di chuyển tiếp
                    if (curr == dList)
                            curr = curr->next;
            }
            printf("Xoa khach hang %i  \n", curr->nodeValue);
            // triển khai tiếp curr và xoá nút tại điểm dừng
            curr = curr->next;
            erase(curr->prev);  // Phải xây dựng hàm này
            // có thể loại bỏ nút ở cuối danh sách, vì thế curr phải trỏ về head và tiếp tục
            if (curr == dList)      curr = curr->next;
    }
    printf("\n Khach hang %i  la nguoi thang cuoc\n",curr->nodeValue);
    // xoá bỏ nút cuối cùng và đầu danh sách
    delete curr; delete dList;
}
```



Chương trình chính

```
void main()
{
        // n - số lượng người chơi
        // m - là số cần đếm
        int n, m;

        printf("Nhap vao n = ");
        scanf("%i",n); printf("\n");

        // tạo ngẫu nhiên số m: 1<=m<=n
        m = 1 + random(n);
        printf(" m = %i",m);
        // giải bài toán và đưa ra người thắng
        josephus(n, m);
}
```

```
Nhap vao n =  10
m = 5
Xoá khách hàng 5
Xoá khách hàng 10
Xoá khách hàng 6
Xoá khách hàng 2
Xoá khách hàng 9
Xoá khách hàng 8
Xoá khách hàng 1
Xoá khách hàng 4
Xoá khách hàng 7

Khach hang 3 la nguoi thang.
```

Ví dụ 2. Biểu diễn đa thức

- Xét đa thức

$$P(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

- Để biểu diễn đa thức đơn giản nhất là dùng mảng a[i] cất giữ hệ số của x^i. Các phép toán với các đa thức như: Cộng hai đa thức, Nhân hai đa thức, ..., khi các đa thức được biểu diễn bởi mảng, có thể cài đặt một cách đơn giản.
- 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.
- Tuy nhiên, khi đó việc cài đặt các phép toán lại phức tạp hơn.

Biểu diễn đa thức

- **Ví dụ:** Có thể sử dụng khai báo sau đây để khai báo danh sách móc nối của hai đa thức Poly1 và Poly2

```
struct Polynom {
    int coeff;
    int pow;
    struct Polynom *link;
} *Poly1, *Poly2
```

- Việc cài đặt phép cộng hai đa thức Poly1 và Poly2 đòi hỏi ta phải duyệt qua hai danh sách móc nối của chúng để tính hệ số của đa thức tổng PolySum (cũng được biểu diễn bởi danh sách móc nối).
- Đây là bài tập tốt để luyện tập cài đặt các thao tác với danh sách móc nối.

Thuật toán tính tổng hai đa thức

Thuật toán SumTwoPol

```
node=(TPol *)malloc (sizeof(TPol));
PolySum=node;
ptr1=Poly1; ptr2=Poly2;
while(ptr1!=NULL && ptr2!=NULL){
    ptr=node;
    if  (ptr1->pow > ptr2->pow ) {
       node->coeff=ptr2->coeff;
       node->pow=ptr2->pow;
       ptr2=ptr2->link;   //update ptr list 2
    }
    else if ( ptr1->pow < ptr2->pow )
      {
          node->coeff=ptr1->coeff;
          node->pow=ptr1->pow;
          ptr1=ptr1->link;   //update ptr list 1
      }
```

Thuật toán tính tổng hai đa thức (tiếp)

```
  else
      {  node->coeff=ptr2->coeff+ptr1->coeff;
         node->pow=ptr2->pow;
         ptr1=ptr1->link;   //update ptr list 1
         ptr2=ptr2->link;   //update ptr list 2
      }

   node=(TPol *)malloc (sizeof(TPol));
   ptr->link=node;   //update ptr list 3
} //end of while
if (ptr1==NULL)     //end of list  1
{ while(ptr2!=NULL){
     node->coeff=ptr2->coeff; node->pow=ptr2->pow;
     ptr2=ptr2->link;   //update ptr list 2
     ptr=node; node=(TPol *)malloc (sizeof(TPol));
     ptr->link=node; }  //update ptr list 3
}
```

Thuật toán tính tổng hai đa thức (tiếp)

```
    else if (ptr2==NULL)      //end of list  2
      {
        while(ptr1!=NULL) {
           node->coeff=ptr1->coeff;
           node->pow=ptr1->pow;
           ptr1=ptr1->link;   //update ptr list 2
           ptr=node;
           node=(TPol *)malloc (sizeof(TPol));
           ptr->link=node;   //update ptr list 3
         }
     }
   node=NULL;
   ptr->link=node;
}
```

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

3.3.5. Một số biến thể của danh sách móc nối

Phân tích sử dụng linked list

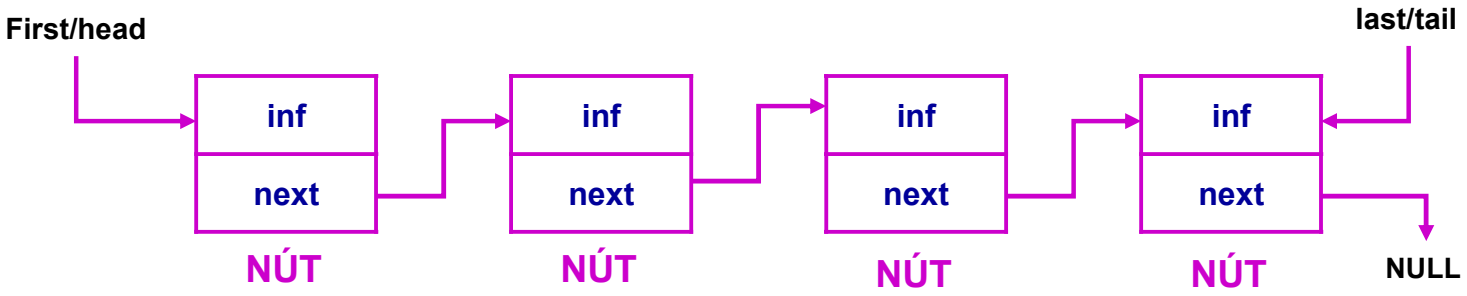
- Những ưu điểm của việc dùng linked lists:
  - Không xảy ra vượt mảng, ngoại trừ hết bộ nhớ.
  - Chèn và Xoá được thực hiện dễ dàng hơn là cài đặt mảng.
  - Với những bản ghi lớn, thực hiện di chuyển con trỏ là nhanh hơn nhiều so với thực hiện di chuyển các phần tử của danh sách.
- Những bất lợi khi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

So sánh các phương pháp

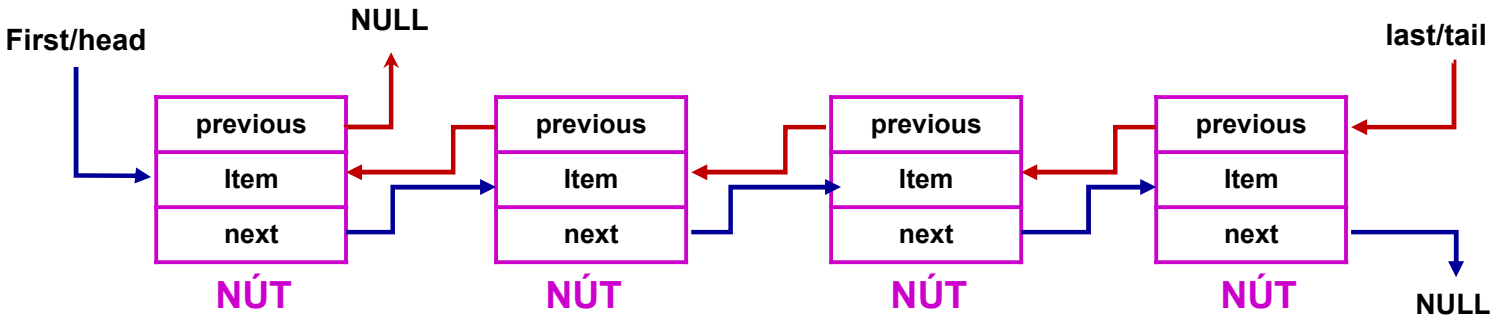
- Việc lựa chọn cách cài đặt mảng hay cài đặt con trỏ để biểu diễn danh sách là tuỳ thuộc vào việc thao tác nào là thao tác thường phải dùng nhất. Dưới đây là một số nhận xét về hai cách cài đặt:
  - Cách cài đặt mảng phải khai báo kích thước tối đa. Nếu ta không lường trước được giá trị này thì nên dùng cài đặt con trỏ.
  - Có một số thao tác có thể thực hiện nhanh trong cách này nhưng lại chậm trong cách cài đặt kia: INSERT và DELETE đòi hỏi thời gian hằng số trong cài đặt con trỏ nhưng trong cài đặt mảng đòi hỏi thời gian $O(N)$ với N là số phần tử của danh sách. PREVIOUS và END đòi hỏi thời gian hằng số trong cài đặt mảng, nhưng thời gian đó là $O(N)$ trong cài đặt con trỏ.
  - Cách cài đặt mảng đòi hỏi dành không gian nhớ định trước không phụ thuộc vào số phần tử thực tế của danh sách. Trong khi đó cài đặt con trỏ chỉ đòi hỏi bộ nhớ cho các phần tử đang có trong danh sách. Tuy nhiên cách cài đặt con trỏ lại đòi hỏi thêm bộ nhớ cho con trỏ.

Tóm tắt hai cách biểu diễn bởi con trỏ

- Trong cách biểu diễn Singly-linked list hoặc doubly-linked list, cho dù bằng cách nào **header** và **tail** chỉ là các con trỏ đến nút đầu tiên và nút cuối cùng, chúng không là nút.
- Singly linked list:



- Doubly linked list:



ADT List: So sánh các cách cài đặt

	Array	Singly-L	Doubly-L
Creation	O(1) nếu dùng kiểu có sẵn O(n) nếu trái lại	O(1)	O(1)
Destruction	giống khởi tạo	O(n)	O(n)
isEmpty()	O(1)	O(1)	O(1)
getLength()	O(1)	O(1) có biến *size* O(n) không có	O(1) có *size* O(n) không có
insertFirst()	O(n)	O(1)	O(1)
insertLast()	O(1)	O(1) có *tail* O(n) không có	O(1) có *tail* O(n) không có
insertAtIndex() Chèn vào vị trí i	O(n)	O(n)	O(n)

ADT List: So sánh các cách cài đặt

	Array	Singly-L	Doubly-L
deleteFirst()	O(n)	O(1)	O(1)
deleteLast()	O(1)	O(n) ngay cả có *tail*	O(1) có tail O(n) không có
deleteAtIndex()	O(n)	O(n)	O(n)
getFirst()	O(1)	O(1)	O(1)
getLast()	O(1)	O(1) có tail O(n) không có	O(1) có tail O(n) không có
getAtIndex() lấy pt tại vị trí	O(1)	O(n)	O(n)

ADT List: So sánh các cách cài đặt

- Nhiều khi cùng cần xác định thêm một số phép toán khác
- Chúng là có ích khi các phép toán insertions/deletions phải thực hiện nhiều trong danh sách. Trong các tình huống như vậy nên sử dụng doubly-linked list.

	Array	Singly-L	Doubly-L
insertBefore()	O(n)	O(n)	O(1)
insertAfter()	O(n)	O(1)	O(1)
deleteBefore()	O(n)	O(n)	O(1)
deleteAfter()	O(n)	O(1)	O(1)
next()	O(1)	O(1)	O(1)
previous()	O(1)	O(n)	O(1)

3.3. Danh sách

3.3.1. Danh sách tuyến tính

3.3.2. Cài đặt danh sách tuyến tính

Biểu diễn dưới dạng mảng

Danh sách móc nối

Danh sách nối đôi

3.3.3. Các ví dụ ứng dụng

3.3.4. Phân tích sử dụng linked list

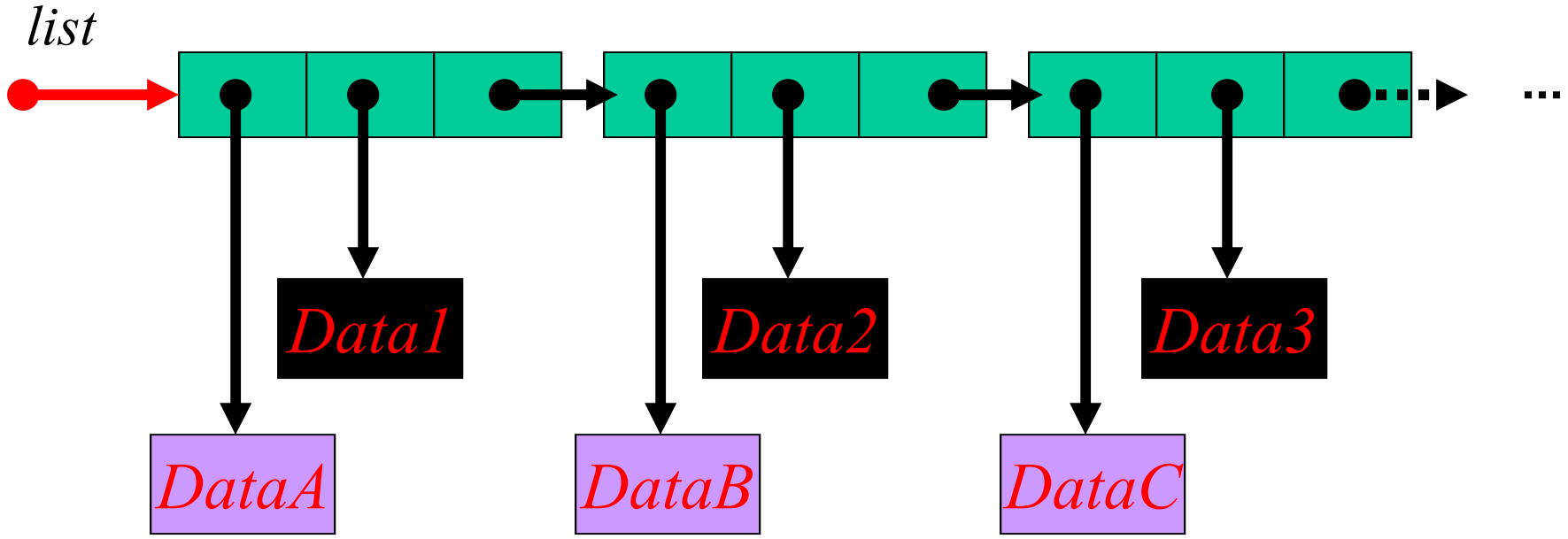
3.3.5. Một số biến thể của danh sách móc nối

Một số biến thể của danh sách móc nối

- Có nhiều biến thể của danh sách móc nối. Ta kể ra một số biến thể thường gặp:
  - Danh sách móc nối đa dữ liệu (Linked List-Multiple data)
  - Danh sách nối vòng (Circular Linked Lists)
  - Danh sách nối đôi vòng (Circular Doubly Linked Lists)
  - Danh sách móc nối của các danh sách (Linked Lists of Lists)
  - Danh sách đa móc nối (Multiply Linked Lists)
- Các phép toán cơ bản với các biến thể này được xây dựng tương tự như đối với danh sách móc nối đơn và danh sách móc nối kép mà ta xét ở trên.

Danh sách móc nối đa dữ liệu

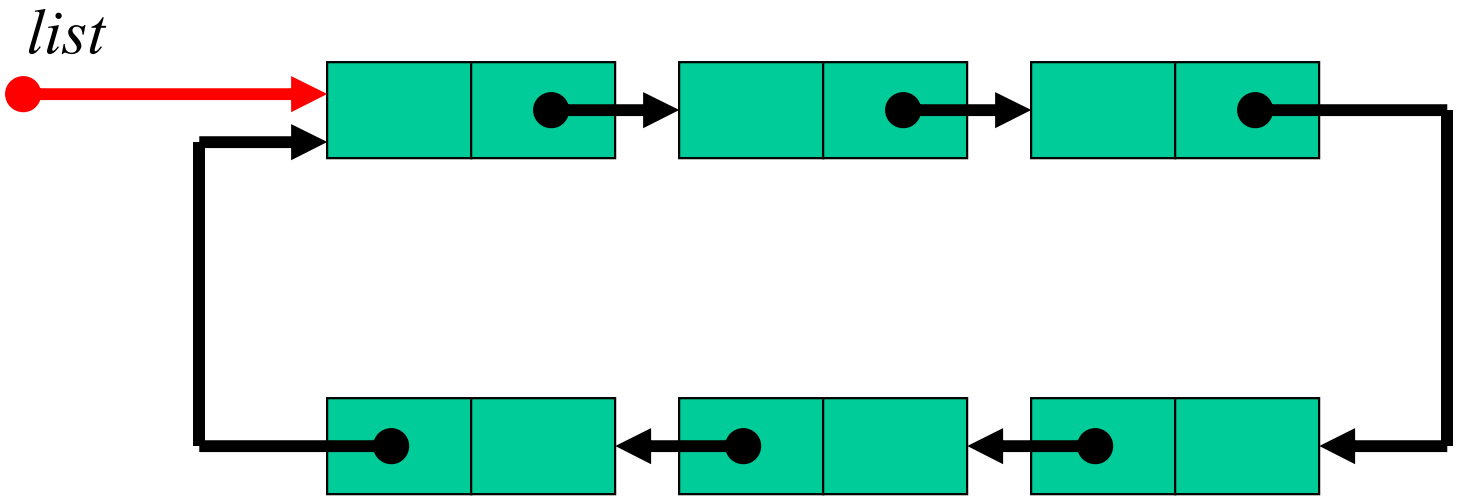
Linked Implementation- Multiple data (pointers)




```
Struct {
    Node * next;
    void * item1;
    void * item2;
}
```

Cất giữ hai loại phần tử 1 và 2

Danh sách nối vòng
Circular Linked Lists

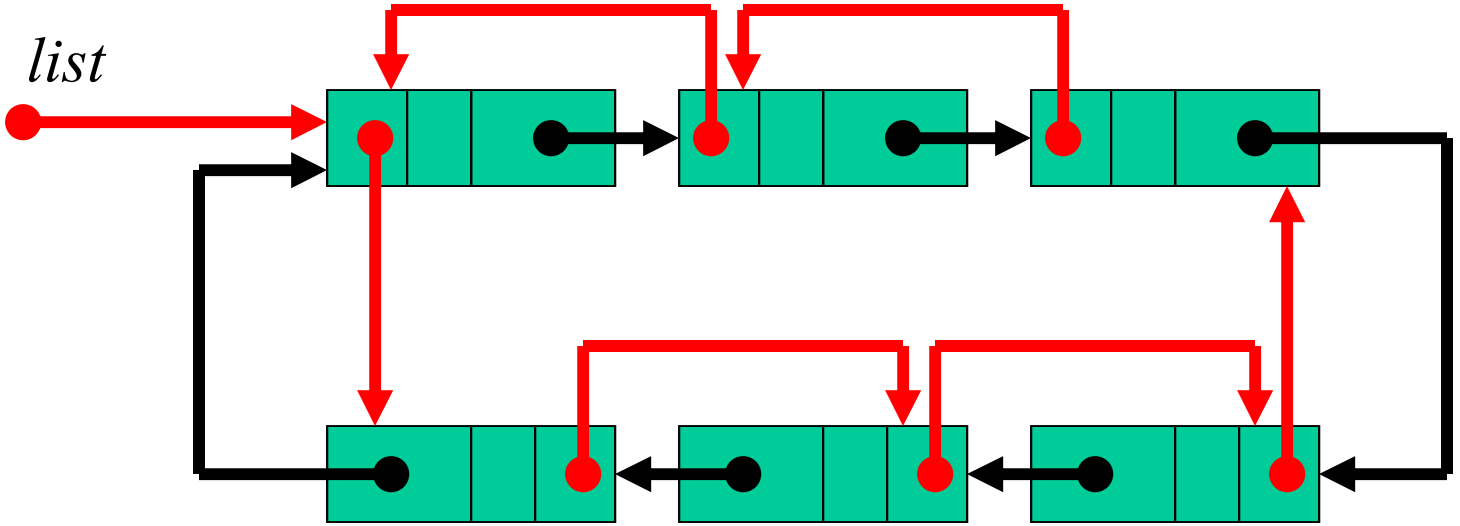



```
Struct {
   DataType * item;
   Node * next;
}
```

Cất giữ phần tử

Danh sách nối đôi vòng

Circular Doubly Linked Lists

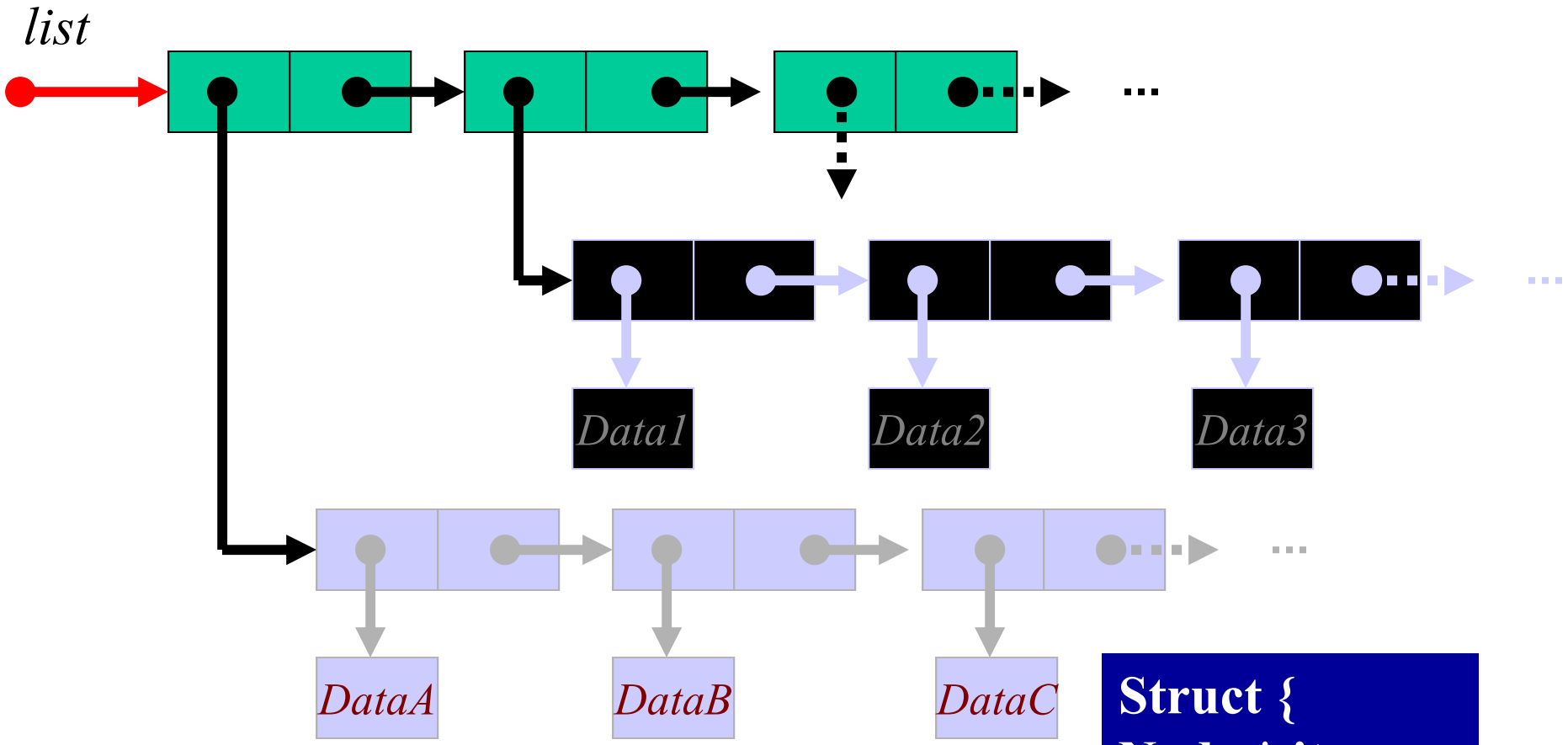



```
Struct {
    void * item;
    Node * prev;
    Node * next;
}
```

Cất giữ phần tử

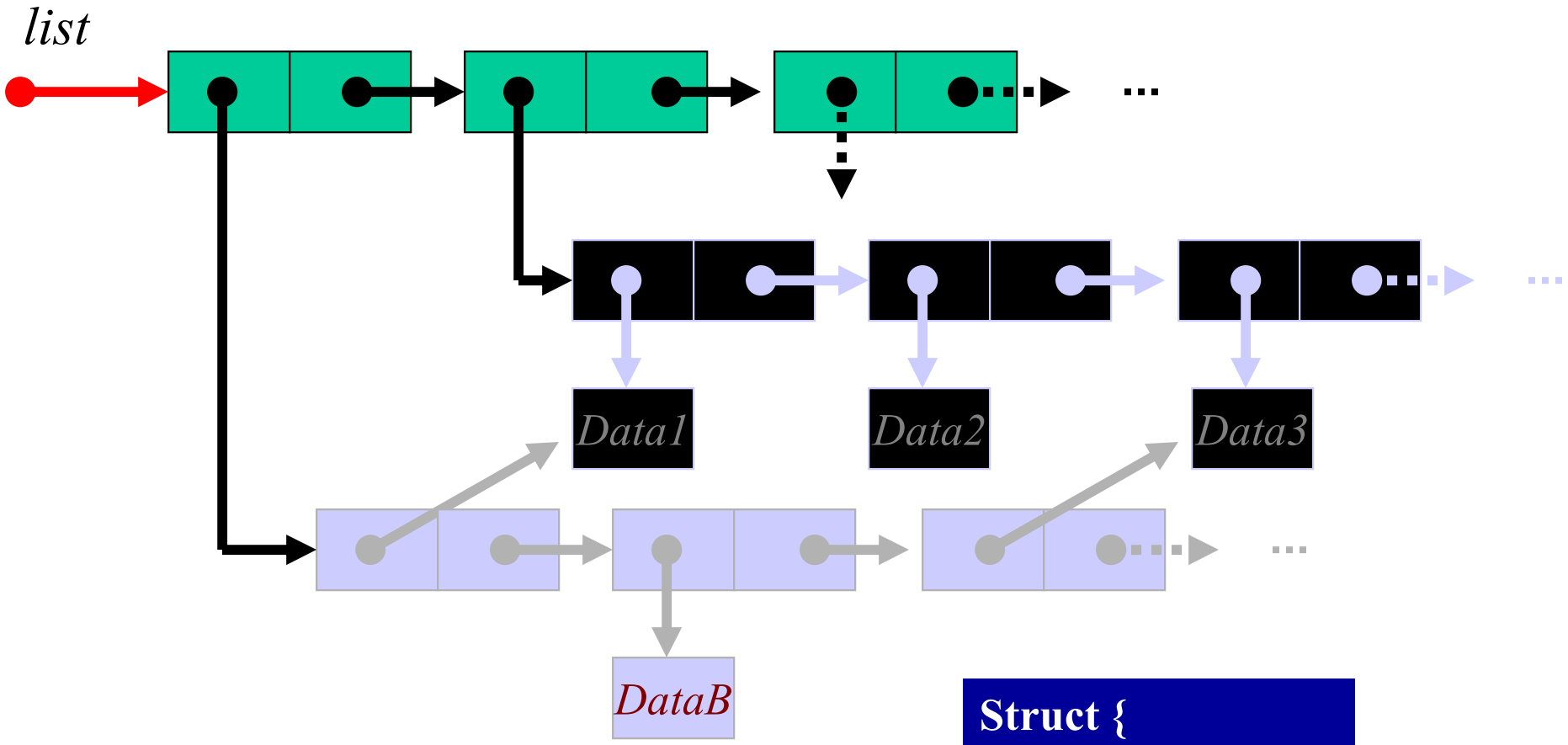
Danh sách móc nối của các danh sách

Linked Lists of Lists




```
Struct {
Node * item;
Node * next;
}
```

Danh sách đa móc nối
Multiply Linked Lists

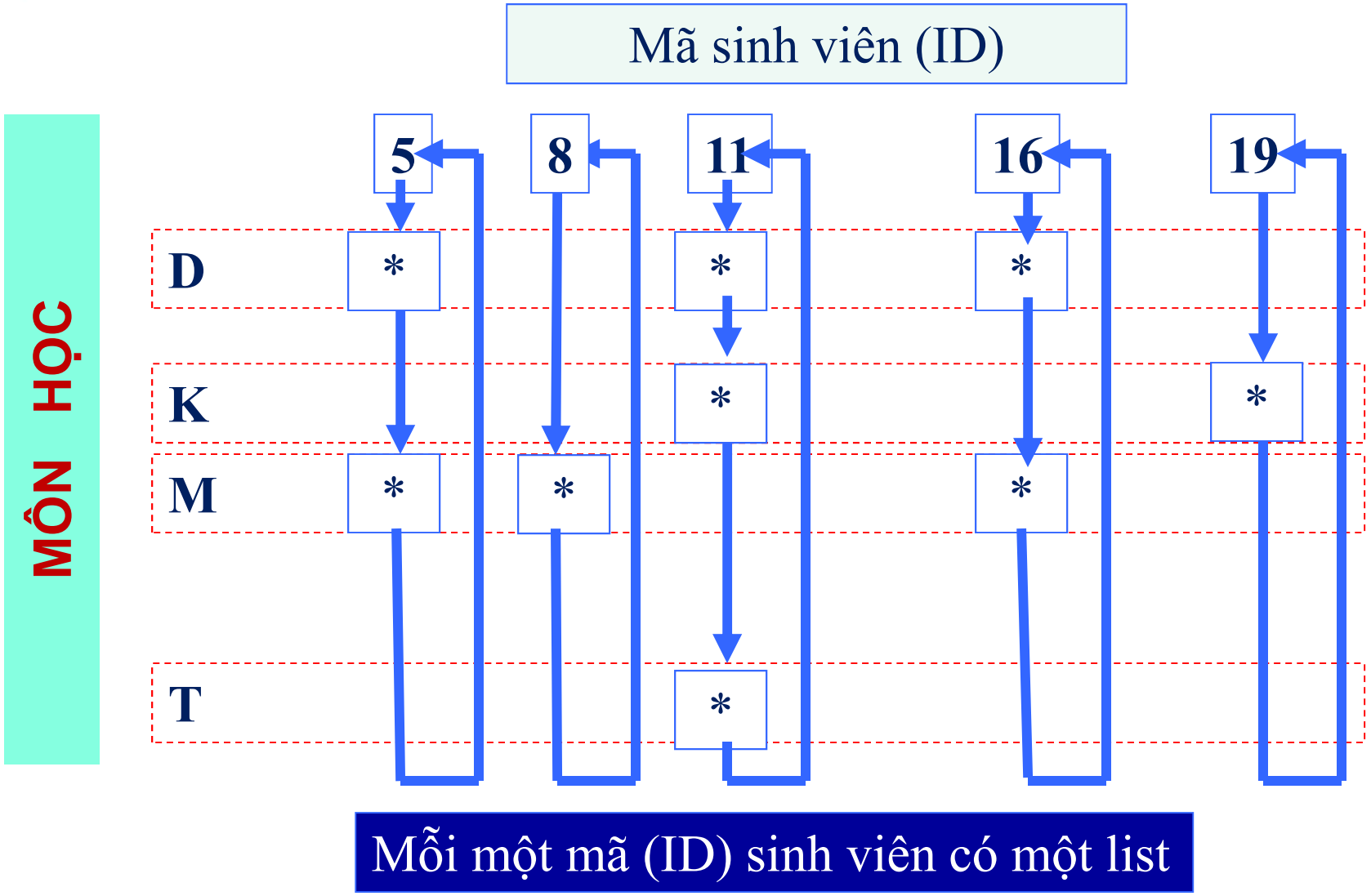



```
Struct {
    Node * item;
    Node * next;
}
```

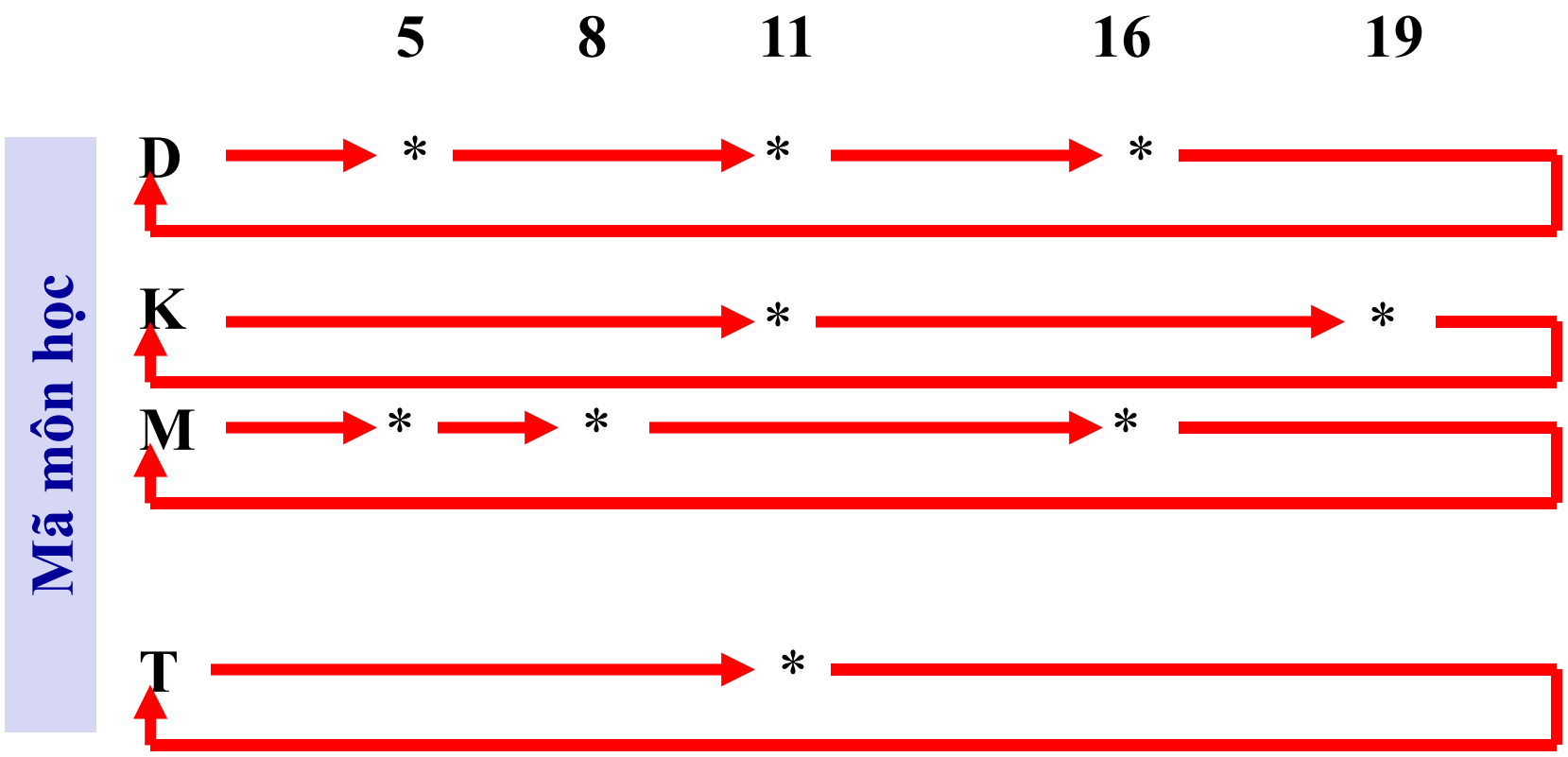
Ứng dụng Multilists - Ma trận thưa (Sparse matrix)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A																			
B																			
C																			
D		*																	
E		*								*							*		
F					*														
G		*								*									
H																			
I																			
J																			
K							*										*		
L					*									*					
M																			
N																			
O																			
P																			
Q																			
R																			
S																			

Multilists - ỨNG DỤNG

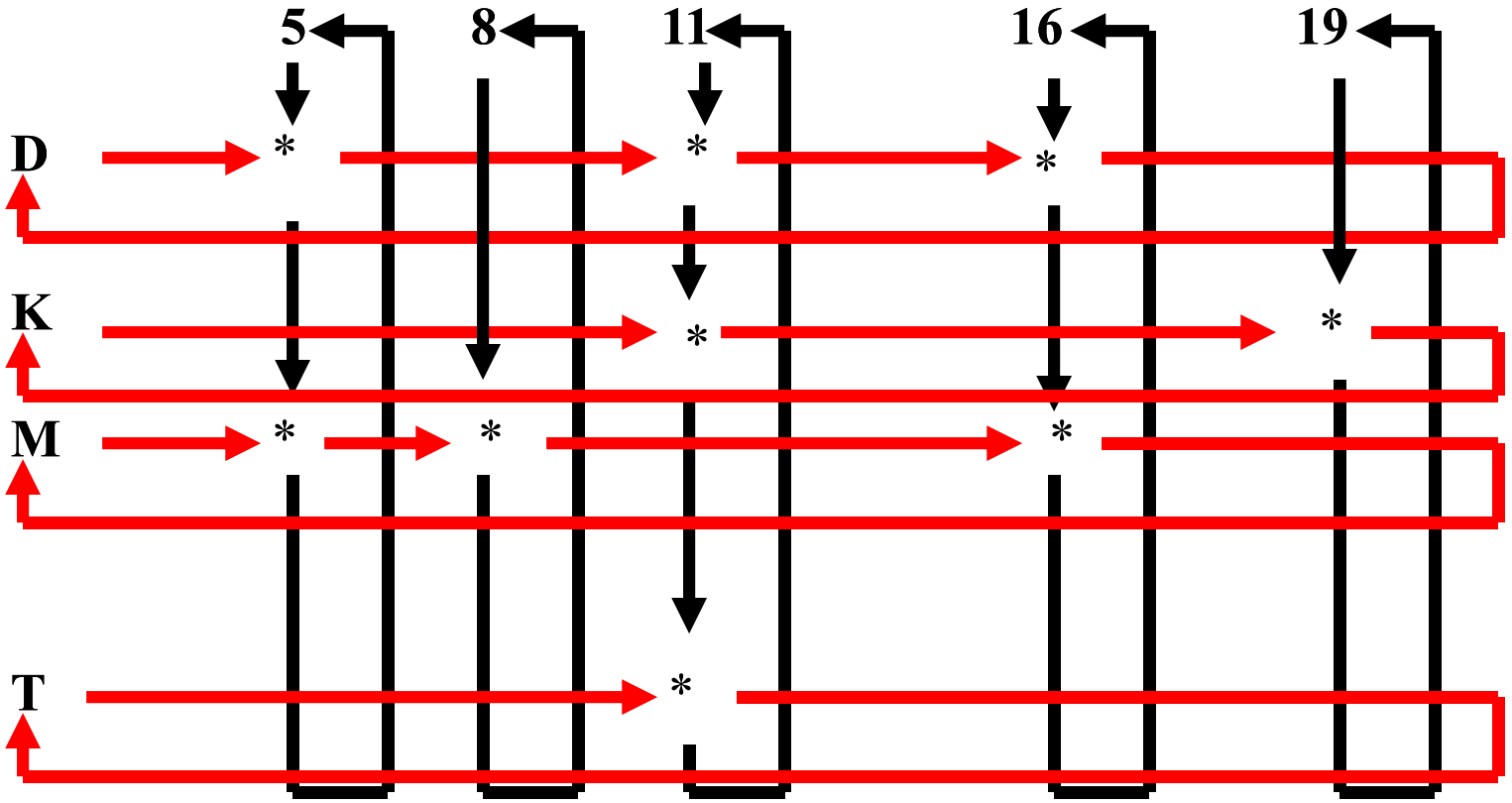


Multilists - ứng dụng



Mỗi môn học có một list

Multilists



Multilists - duyệt danh sách

- Chọn một danh sách tương ứng với một header, chẳng hạn "Mã Sinh Viên = 5"
- Lặp lại:
  - Duyệt danh sách, tại mỗi nút đi theo danh sách môn học.
  - Định vị được các đầu danh sách, chẳng hạn "Mã môn học = D".
- Thu được: Sinh viên 5 đăng ký học môn D và M.

Chương 3. Các cấu trúc dữ liệu cơ bản

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

3.4. Ngăn xếp

3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

3.4.2. Ngăn xếp dùng mảng

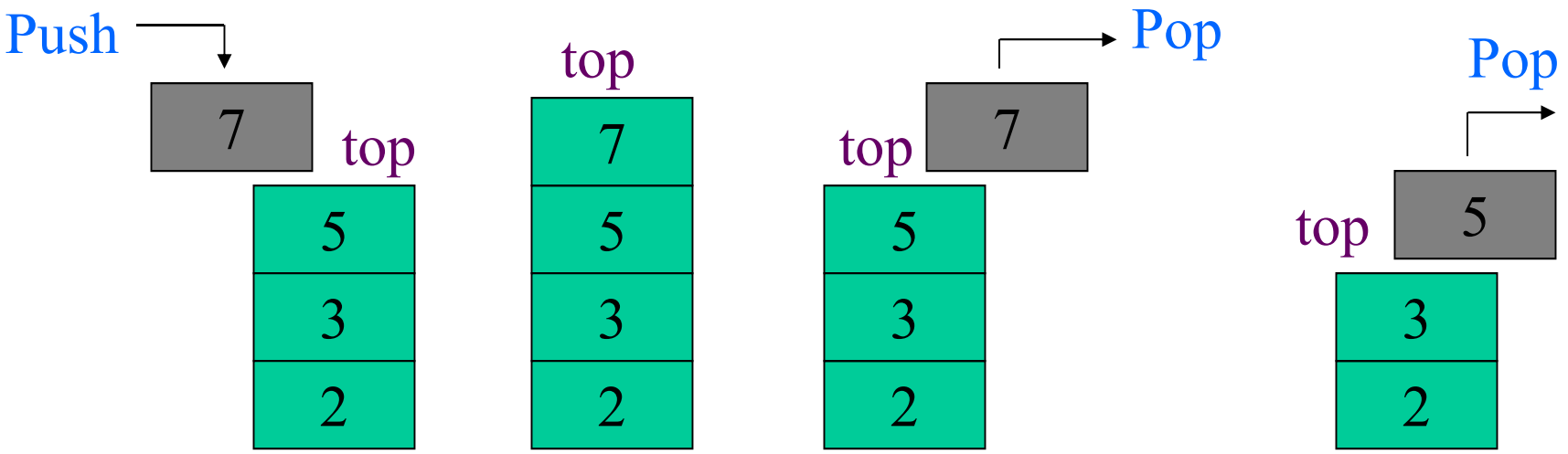
3.4.3. Cài đặt ngăn xếp với danh sách móc nối

3.4.4. Một số ứng dụng của ngăn xếp

- Ứng dụng 1: Ngoặc hợp cách
- Ứng dụng 2: Sánh đôi thẻ trong HTML
- Ứng dụng 3. Bài toán đổi cơ số
- Ứng dụng 4. Bài toán tính giá trị biểu thức số học
- Ứng dụng 5. Ngăn xếp và đệ qui

3.4.1. ADT ngăn xếp (Stacks ADT)

- **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.



Nguyên tắc: Vào sau - Ra trước

Last-in, first-out (LIFO)

3.4.1. ADT ngăn xếp (Stacks ADT)

- **Các phép toán cơ bản (stack operations):**
  - **push**(object): nạp vào một phần tử (inserts an element)
  - object **pop**(): lấy ra phần tử nạp vào sau cùng (removes and returns the last inserted element)
- **Các phép toán bổ trợ:**
  - object **top**(): trả lại phần tử nạp vào sau cùng mà không loại nó khỏi ngăn xếp
  - integer **size**(): trả lại số lượng phần tử được lưu trữ
  - boolean **isEmpty**(): nhận biết có phải ngăn xếp rỗng
- **Có hai cách tổ chức ngăn xếp:**
  - Sử dụng mảng
  - Sử dụng danh sách móc nối

3.4. Ngăn xếp

3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

3.4.2. Ngăn xếp dùng mảng

3.4.3. Cài đặt ngăn xếp với danh sách móc nối

3.4.4. Một số ứng dụng của ngăn xếp

- Ứng dụng 1: Ngoặc hợp cách
- Ứng dụng 2: Sánh đôi thẻ trong HTML
- Ứng dụng 3. Bài toán đổi cơ số
- Ứng dụng 4. Bài toán tính giá trị biểu thức số học
- Ứng dụng 5. Ngăn xếp và đệ qui

Ngăn xếp dùng mảng
Array-based Stack

- Cách đơn giản nhất để cài đặt ngăn xếp là dùng mảng (*S*)
- Ta nạp các phần tử theo thứ tự từ trái sang phải
- Có biến lưu giữ chỉ số của phần tử ở đầu ngăn xếp (*N*)

```
Algorithm size()
  return N + 1

Algorithm pop()
  if isEmpty() then
    Error("EmptyStack")
  else
    N ← N – 1
  return S[N + 1]
```

Ngăn xếp dùng mảng
Array-based Stack

- Mảng cất giữ ngăn xếp có thể tràn
- Khi đó thao tác nạp vào phải thông báo lỗi: FullStack
  - Đây là hạn chế của cách cài đặt dùng mảng
  - Không phải là bản chất của Stack ADT

```
Algorithm push(x)
  if N = S.length – 1 then
    error("FullStack")
  else
    N ← N + 1
    S[N] ← x
```

Cài đặt Ngăn xếp dùng mảng trên C
(Stack Array Implementation)

- **Các phép toán cơ bản:**
- **void STACKinit(int);**
- **int STACKempty();**
- **void STACKpush(Item);**
- **Item STACKpop();**

```
typedef   .... Item;
static Item *s;
static int N;

void STACKinit(int maxN){
    s = (Item *) malloc(maxN*sizeof(Item));
    N = 0;}

int STACKempty()
    {return N==0;}

void STACKpush(Item item)
    {s[N++] = item;}

Item STACKpop()
    { return s[--N];}
```

3.4. Ngăn xếp

3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

3.4.2. Ngăn xếp dùng mảng

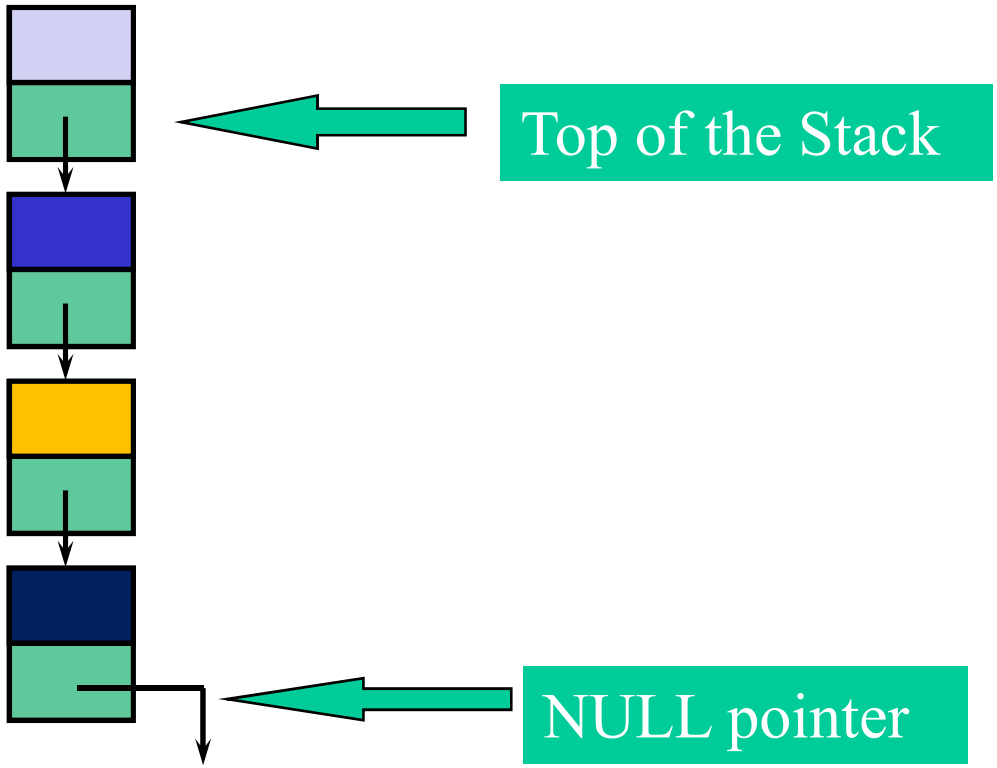
3.4.3. Cài đặt ngăn xếp với danh sách móc nối

3.4.4. Một số ứng dụng của ngăn xếp

- Ứng dụng 1: Ngoặc hợp cách
- Ứng dụng 2: Sánh đôi thẻ trong HTML
- Ứng dụng 3. Bài toán đổi cơ số
- Ứng dụng 4. Bài toán tính giá trị biểu thức số học
- Ứng dụng 5. Ngăn xếp và đệ qui

Cài đặt ngăn xếp với danh sách móc nối
(Linked Stack)

- Trong cách cài đặt ngăn xếp dùng danh sách móc nối, Ngăn xếp được cài đặt như danh sách móc nối với các thao tác bổ sung và loại bỏ luôn làm việc với nút đầu tiên.



Cài đặt ngăn xếp với danh sách móc nối
Linked Stack)

MÔ TẢ NGĂN XẾP

```
struct StackNode {
    float item;
    StackNode *next;
};

struct Stack {
    StackNode *top;
};
```

Các phép toán cơ bản
(Linked Stack)

Cài đặt các phép toán:

1. Khởi tạo: `Stack *StackConstruct();`

2. Kiểm tra ngăn xếp rỗng:

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

3. Kiểm tra tràn ngăn xếp:

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

4. Nạp vào (Push): *Nạp phần tử vào đầu ngăn xếp*

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

5. Lấy ra (Pop): *Trả lại giá trị phần tử ở đầu ngăn xếp*

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

6. Đưa ra các phần tử của ngăn xếp

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

Khởi tạo ngăn xếp
(Initialize Stack)

```
Stack *StackConstruct() {
    Stack *s;
    s = (Stack *)malloc(sizeof(Stack));
    if (s == NULL) {
      return NULL;  // No memory
    }
    s->top = NULL;
    return s;
}
/****   Huỷ ngăn xếp   *****/
void StackDestroy(Stack *s) {
    while (!StackEmpty(s)) {
      StackPop(s);
    }
    free(s);
}
```

Các hàm bổ trợ

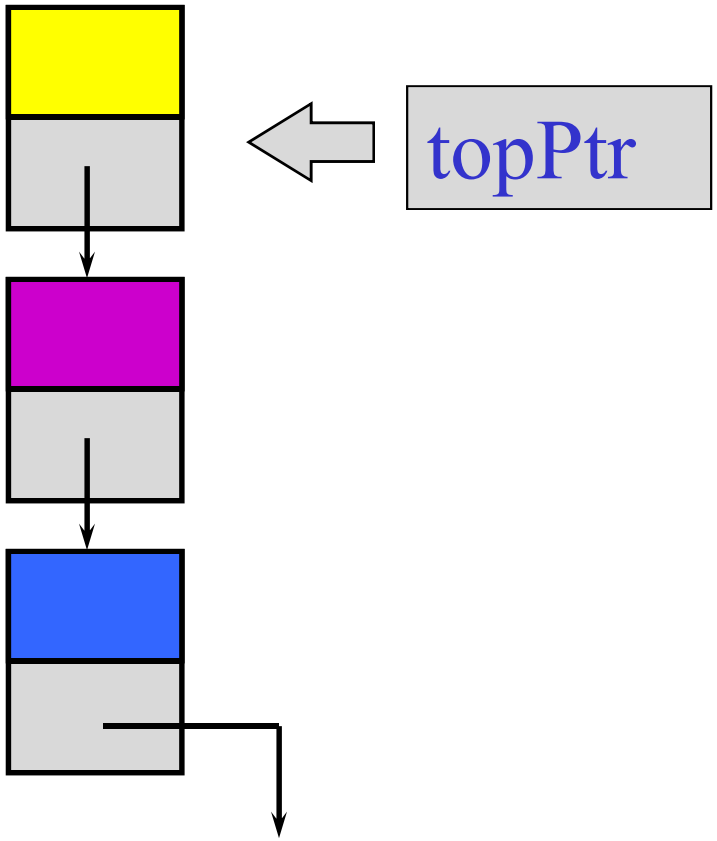
```
/*** Kiểm tra Stack rỗng   ***/
int StackEmpty(const Stack *s) {
    return (s->top == NULL);
}

/*** Thông báo Stack tràn   ***/
int StackFull() {
   printf("\n NO MEMORY! STACK IS FULL");
    getch();
    return 1;
}
```

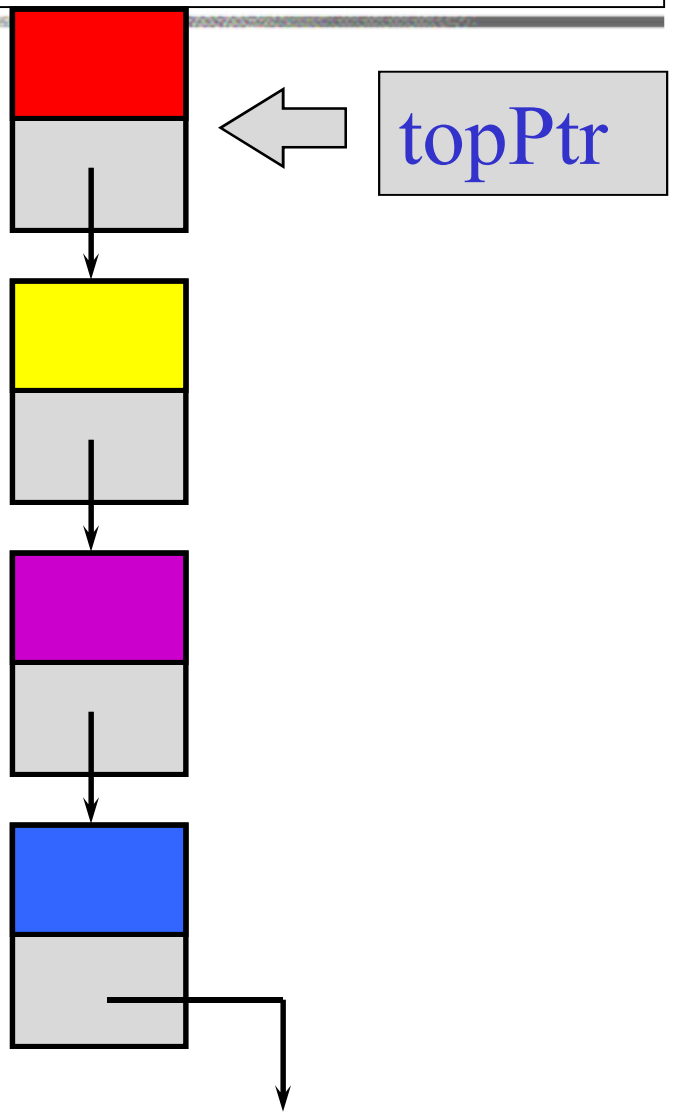
Đưa ra các phần tử của ngăn xếp

```
void disp(Stack* s) {
   StackNode* node;
   int ct = 0; float m;
   printf("\n\n  DANH SACH CAC PHAN TU CUA STACK \n\n");
   if (StackEmpty(s)) {
       printf("\n\n  >>>>>  EMPTY STACK   <<<<<\n");
       getch();  }
   else {
      node= s->top;
      do {
         m=node->item;  printf("%8.3f ", m); ct++;
         if (ct % 9 == 0) printf("\n");
         node = node->next;
       } while (!(node == NULL));
       printf("\n");
   }
}
```

Nạp vào (Push)

- Bổ sung vào đầu Stack





Nạp vào - Push

Cần thực hiện các thao tác sau:

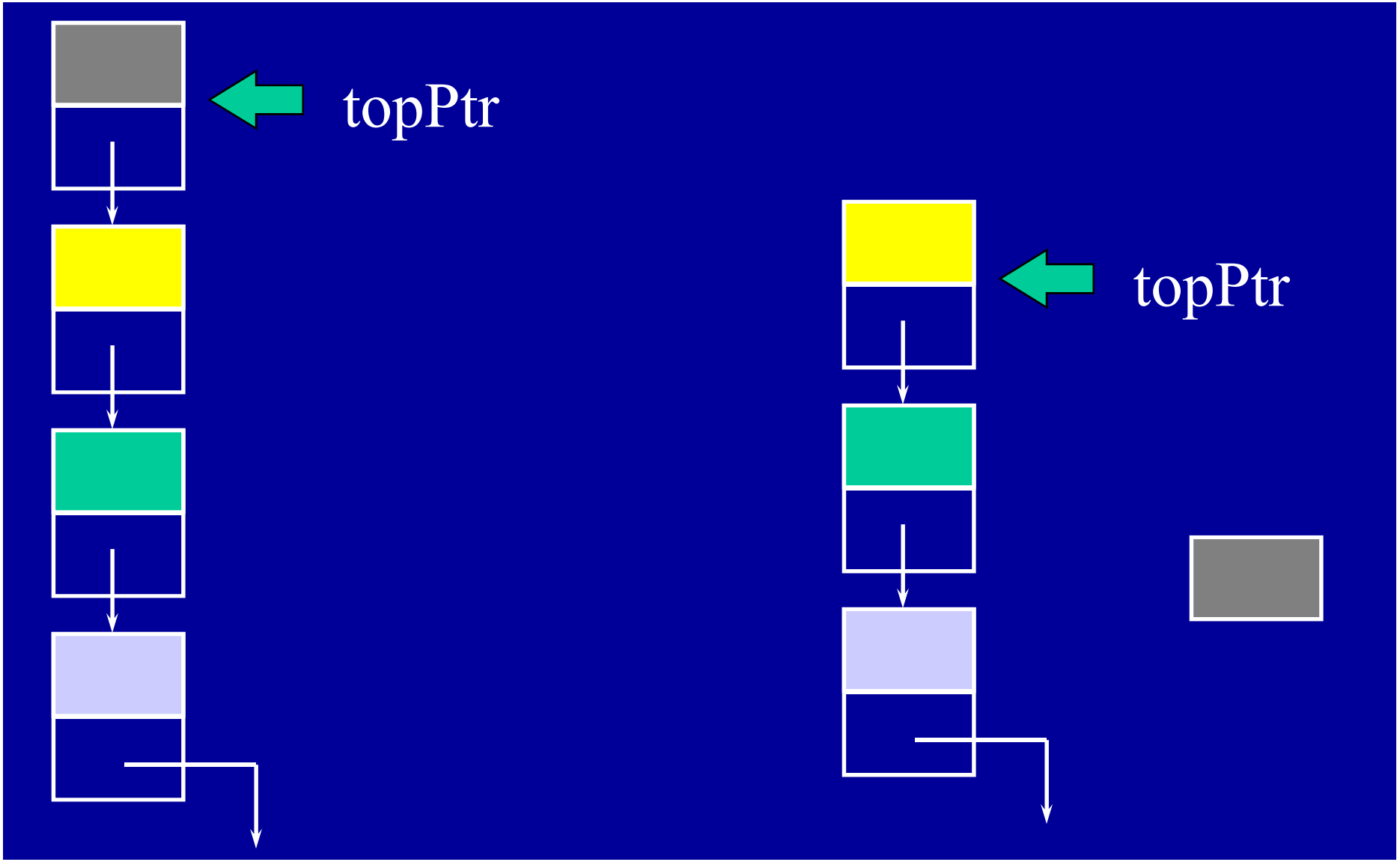
(1) Tạo nút mới cho item

(2) Móc nối nút mới đến nút ở đầu

(3) Đặt nút mới thành nút đầu mới

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

Lấy ra - Pop



Thuật toán thực hiện Pop

- **Thuật toán:**
  1. kiểm tra có phải ngăn xếp là rỗng
  2. ghi nhớ địa chỉ của nút đầu hiện tại
  3. ghi nhớ giá trị phần tử ở nút đầu
  4. chuyển nút tiếp theo thành nút đầu mới (new top)
  5. giải phóng nút đầu cũ
  6. trả lại giá trị phần tử ở nút đầu cũ

Cài đặt Pop trên C

```
float StackPop(Stack *s) {
    float item;
    StackNode *node;
    if (StackEmpty(s)) {          //(1)
  // Empty Stack, can't pop
  return NULL;
    }
    node = s->top;                //(2)
    item = node->item;            //(3)
    s->top = node->next;          //(4)
    free(node);                   //(5)
    return item;                  //(6)
}
```

Chương trình thử nghiệm

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
// Các mô tả và hàm liên quan đến Stack
int main() {
   int ch,n,i;   float m;
   Stack* stackPtr;
   while(1)
   {  printf("\n\n======================\n");
      printf("CHUONG TRINH THU STACK\n");
      printf("======================\n");
      printf(" 1.Create\n 2.Push\n 3.Pop\n 4.Display\n 5.Exit\n");
      printf("----------------------\n");
      printf("Chon chuc nang: ");
      scanf("%d",&ch); printf("\n\n");
```

Chương trình thử nghiệm

```
   switch(ch) {
      case 1:    printf("Da khoi tao STACK");
                 stackPtr = StackConstruct(); break;
      case 2:    printf("Vao gia tri phan tu: "); scanf("%f",&m);
                 StackPush(stackPtr, m); break;
      case 3:     m=StackPop(stackPtr);
                  if (m != NULL)
                  {printf("\n\n Gia tri phan tu lay ra: %8.3f\n",m);
                  } else {
                   printf("\n\n  >>> Empty Stack, can't pop <<<\n");}
                 break;
      case 4:    disp(stackPtr); break;
      case 5:    printf("\n  Bye! Bye! \n\n"); getch();
                exit(0);    break;
      default:   printf("Wrong choice"); getch();
    } //switch
  } // end while
} //end main
```

File chương trình: c:\temp\devcpp\STACK_N0.CPP

3.4. Ngăn xếp

3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

3.4.2. Ngăn xếp dùng mảng

3.4.3. Cài đặt ngăn xếp với danh sách móc nối

3.4.4. Một số ứng dụng của ngăn xếp

- Ứng dụng 1: Ngoặc hợp cách
- Ứng dụng 2: Sánh đôi thẻ trong HTML
- Ứng dụng 3. Bài toán đổi cơ số
- Ứng dụng 4. Bài toán tính giá trị biểu thức số học
- Ứng dụng 5. Ngăn xếp và đệ qui

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
  - 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

Một số ứng dụng của STACK

- Ta xét ứng dụng của STACK vào việc cài đặt thuật toán giải một số bài toán sau:
  - Bài toán kiểm tra dấu ngoặc hợp lệ
  - Bài toán đổi cơ số
  - Bài toán tính giá trị biểu thức số học
  - Chuyển đổi biểu thức dạng trung tố về hậu tố
  - Khử đệ qui

Ứng dụng 1: Parentheses Matching

- Mỗi "(", "{", hoặc "[" phải cặp đôi với ")", "}", hoặc "]"
- **Ví dụ:**
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}))
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Thuật toán giải bài toán Parentheses Matching

Algorithm ParenMatch(*X,n*):

Đầu vào: Mảng *X* gồm *n* ký hiệu, mỗi ký hiệu hoặc là dấu ngoặc, hoặc là biến, hoặc là phép toán số học, 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 ký hiệu mở ngoặc)
            push(S, X[i]);    // gặp dấu mở ngoặc thì đưa vào Stack
    else
       if (X[i] là ký hiệu đóng ngoặc)
                                // gặp dấu đóng ngoặc thì so với dấu mở ngoặc ở đầu Stack
          if  isEmpty(S)
                    return false {không tìm được cặp đôi}
          if  (pop(S)  không đi 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}
```

Ứng dụng 2: Sánh đôi thẻ trong HTML
HTML Tag Matching

- *Trong HTML, mỗi <name> phải đi cặp đôi với </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?

HTML Tag Matching Algorithm

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

Ứng dụng 3. Bài toán đổi cơ số

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

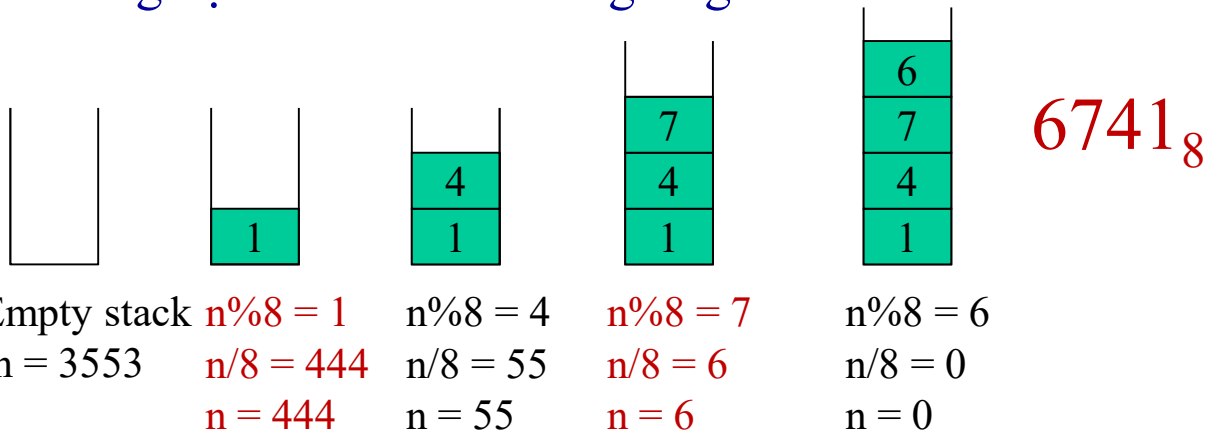
(cơ số 8) $28_{10} = 3 \bullet 8 + 4 = 34_8$

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

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

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:**



6741_8

1. *Chữ số phải nhất của n là n % b. Push 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.*

Chuyển phần dư thành ký tự chữ số tương ứng trong hệ đếm 16:

```
string digitChar = "0123456789ABCDEF";
// chữ số cho13 trong hệ đếm 16 là digitChar[13]=D
```

Ứng dụng 4. Bài toán tính giá trị biểu thức số học

- Xét việc tính giá trị của biểu thức số học trong đó có các phép toán hai ngôi: Cộng, trừ, nhân, chia, luỹ thừa giữa các toán hạng (gọi là biểu thức số học trong ký pháp trung tố - infix notation).
- Thông thường, đối với biểu thức trong ký pháp trung tố, trình tự thực hiện tính biểu thức được chỉ ra bởi các cặp dấu ngoặc hoặc theo thứ tự ưu tiên của các phép toán.
- Vào năm1920, Łukasiewicz (nhà toán học Ba lan) đã đề xuất ký pháp Ba lan cho phép không cần sử dụng các dấu ngoặc vẫn xác định được trình tự thực hiện các phép toán trong biểu thức số học.

Jan Łukasiewicz
1878 - 1956

Ký pháp trung tố (Infix Notation)

- *Mỗi phép toán hai ngôi được đặt giữa các toán hạng.*
- *Mỗi phép toán một ngôi (unary operator) đi ngay trước toán hạng.*

$$-2 + 3 * 5 \iff (-2) + (3 * 5)$$

- **Biểu thức trong ký pháp trung tố sẽ được tính nhờ sử dụng hai ngăn xếp có kiểu dữ liệu khác nhau:**

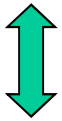
- một ngăn xếp để giữ các toán hạng

- ngăn xếp kia giữ các phép toán.

Ký pháp hậu tố (Postfix Notation)

Ký pháp hậu tố còn được gọi là ký pháp đảo Balan (Reverse Polish Notation - RPN) trong đó các toán hạng được đặt trước các phép toán.

*a b * c +* — *không cần dấu ngoặc trong RPN.*



*a * b + c* — *(ký pháp trung tố tương đương)*

Ví dụ.

infix	*postfix*
*a*b*c*d*e*f*	*ab*c*d*e*f**
*1 + (-5) / (6 * (7+8))*	*1 5 - 6 7 8 + * / +*
*(x/y – a*b) * ((b+x) – y) ^{y}*	*x y / a b * – b x + y-y^ **
*(x*y*z – x^2 / (y*2 – z^3) + 1/z) * (x – y)*	*xy*z*x2^y2*z3^ – / – 1z/+xy – **

Tính giá trị biểu thức hậu tố

A Postfix Calculator

biểu thức trung tố: (7 – 11) * 2 + 3

dạng hậu tố tương đương: 7 11 – 2 * 3 +

Sử dụng *ngăn xếp toán hạng*

step 1	7	11 – 2 * 3 +	step 4	2 -4	* 3 +
step 2	11 7	– 2 * 3 +	step 5	-8	3 +
step 3	-4	2 * 3 +	step 6	3 -8	+
			step 7	-5	Kết quả!

Tính giá trị biểu thức hậu tố nhờ dùng ngăn xếp

Nếu đầu vào là số

Đầu vào là phép toán

push 5
push 9
push 8

pop 8
pop 9
eval 1
push 1

Dãy đầu vào

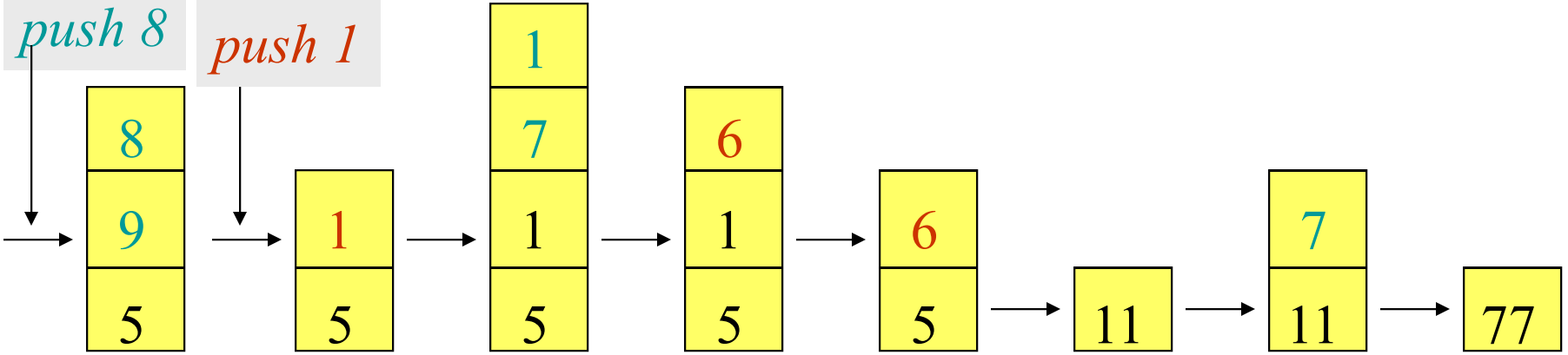
- 5 9 8 – 7 1 – * + 7 *

= 5 (9 – 8) (7 – 1) * + 7 *

= 5 ((9 – 8) * (7 – 1)) + 7 *

= (5 + ((9 – 8) * (7 – 1))) 7 *

= (5 + ((9 – 8) * (7 – 1))) * 7



Thuật toán

- Giả sử ta có xâu gồm các toán hạng và các phép toán.
  1. Duyệt biểu thức từ trái sang phải
  2. Nếu gặp toán hạng thì đưa (push) giá trị của nó vào ngăn xếp
  3. Nếu gặp phép toán thì thực hiện phép toán này với hai toán hạng được lấy ra (pop) từ ngăn xếp.
  4. Cất giữ (push) giá trị tính được vào ngăn xếp (như vậy, 3 ký hiệu được thay bởi một toán hạng)
  5. Tiếp tục duyệt cho đến khi trong ngăn xếp chỉ còn một giá trị duy nhất - chính là kết quả của biểu thức
- Thời gian tính là $O(n)$ bởi vì mỗi toán hạng và mỗi phép toán được duyệt qua đúng một lần.

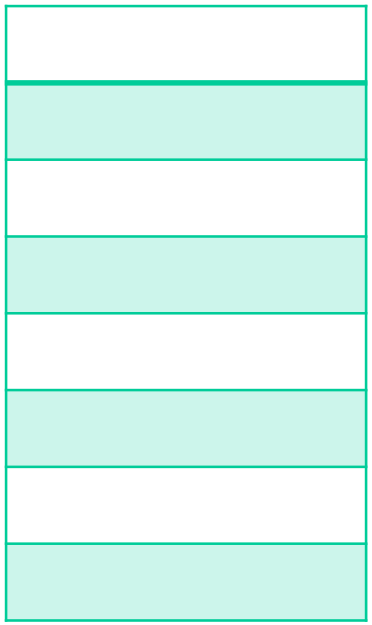
Thuật toán

- **Thuật toán có thể mô tả hình thức hơn như sau:**

```
Khởi tạo ngăn xếp rỗng S;
while(dòng vào khác rỗng){
    token = <phần tử tiếp theo của biểu thức>;
    if (token là toán hạng){
        push(S,token);
    }
    else if (token là phép toán){
                op2 = pop(S);
                op1 = pop(S);
                result = calc(token, op1, op2);
                push(S,result);
    }
} //end of while
return pop(S);
```

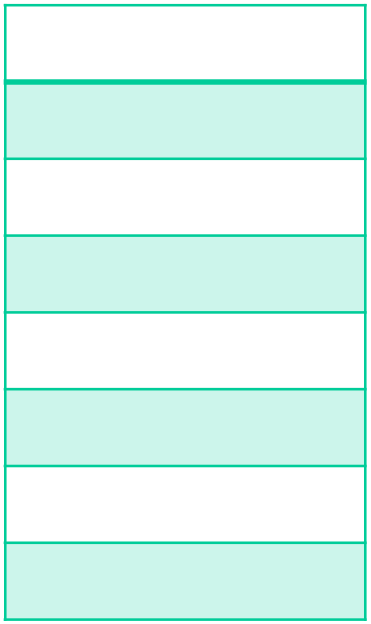
Ví dụ: Tính giá trị biểu thức hậu tố

a	*b*	*c*	-	*d*	+	/	*e*	*a*	-	*	*c*	*



Ví dụ: Tính giá trị biểu thức hậu tố

5	10	7	-	2	+	/	15	5	-	*	7	*

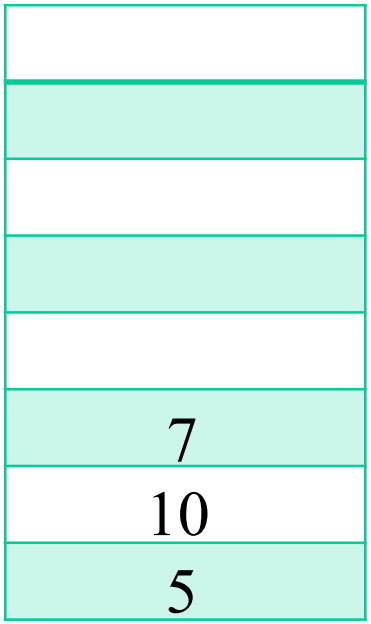


Nạp các ký hiệu trong mảng vào stack

Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.

Ví dụ: Tính giá trị biểu thức hậu tố

			-	2	+	/	15	5	-	*	7	*



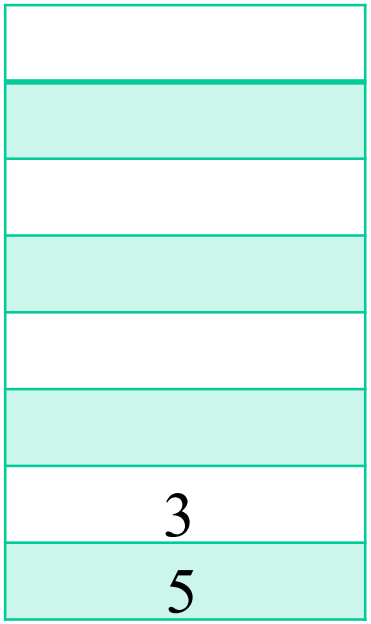
Nạp các ký hiệu trong mảng vào stack

Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.

= 3

Ví dụ: Tính giá trị biểu thức hậu tố

				2	+	/	15	5	-	*	7	*



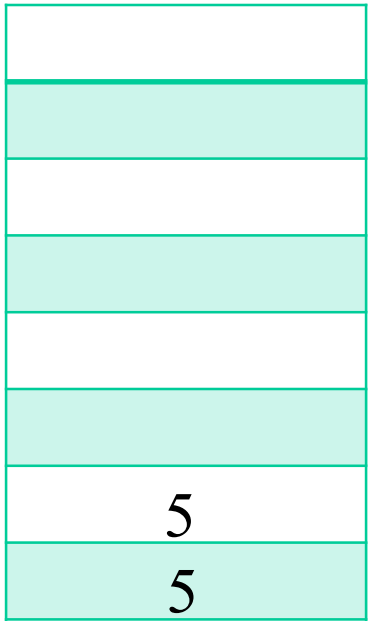
$= 5$

** Nạp các ký hiệu trong mảng vào stack*

** Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.*

Ví dụ: Tính giá trị biểu thức hậu tố

						/	15	5	-	*	7	*



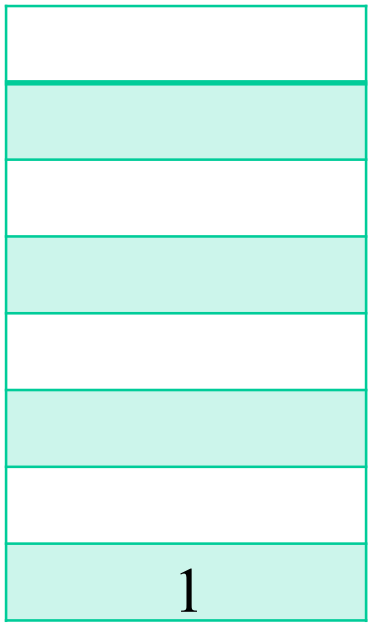
$= 1$

** Nạp các ký hiệu trong mảng vào stack*

** Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.*

Ví dụ: Tính giá trị biểu thức hậu tố

							15	5	-	*	7	*



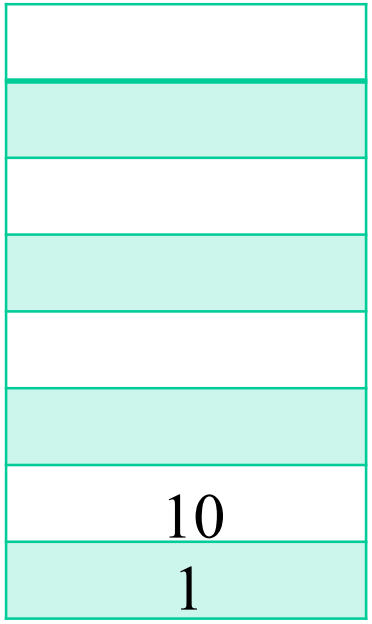
$= 10$

** Nạp các ký hiệu trong mảng vào stack*

** Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.*

Ví dụ: Tính giá trị biểu thức hậu tố

										*	7	*

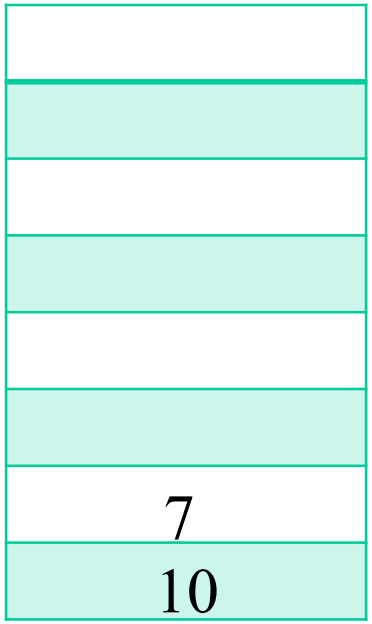


=10

** Nạp các ký hiệu trong mảng vào stack*

** Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.*

Ví dụ: Tính giá trị biểu thức hậu tố



=70 là giá trị cuả biểu thức

** Nạp các ký hiệu trong mảng vào stack*

** Nếu ký hiệu là phép toán, thì thực hiện phép toán này với hai toán hạng ở đầu stack và nạp kết quả vào stack.*

Cài đặt PostfixCalcul tính giá trị biểu thức hậu tố đơn giản

Đầu vào: Xâu chứa biểu thức hậu tố có độ dài không quá 80 ký tự. Các toán hạng và phép toán phân cách nhau bởi đúng một dấu cách

Kết quả: Đưa ra giá trị của biểu thức.

Toán hạng được giả thiết là:

số nguyên không âm

Phép toán chỉ có:

+, -, *

Cài đặt PostfixCalcul

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
int stack [1000];
int top;

// Tính giá trị của biểu thức
int eval (char *s);
// Kiểm tra có phải phép toán
int isop (char op);
// Thao tác đẩy ra của ngăn xếp
int pop (void);
// Thao tác đẩy vào của ngăn xếp
void push (int a);
// Thực hiện phép toán
int do_op (int a, int b, char op);
```

Cài đặt PostfixCalcul

```
int main (void)
{
   char expression[80];
   int value;

   printf ("Nhap vao xau bieu thuc: ");
   gets(expression);
   printf ("\nBieu thuc nhap vao: %s", expression);
   value=eval (expression);
   printf ("\nGia tri cua bieu thuc = %i", value);
   getch();
   return 0;
}
```

Cài đặt PostfixCalcul

```
int eval (char *s){
   char *ptr;
   int first, second, c;
   ptr = strtok (s, " ");
   top = -1;
   while (ptr) {
        if (isop (*ptr))  {
                second = pop(); first = pop();
                c = do_op (first, second, *ptr);
                push(c);
        }
        else {  c = atoi(ptr); push(c);
        }
        ptr = strtok (NULL, " ");
   }
   return (pop ());
}
```

Cài đặt PostfixCalcul

```
int do_op (int a, int b, char op)
{
   int ans;
   switch (op) {
   case '+':
        ans = a + b;
        break;
   case '-':
        ans = a - b;
        break;
   case '*':
        ans = a * b;
        break;
   }
   return ans;
}
```

Cài đặt PostfixCalcul

```
int pop (void){
   int ret;
   ret = stack [top];
   top--;
   return ret;
}

void push (int a){
   top++;
   stack [top] = a;
}

int isop (char op){
   if (op == '+' || op == '-' || op == '*')
        return 1;
   else
        return 0;
}
```

Chuyển biểu thức dạng trung tố về dạng hậu tố (Infix to Postfix Conversion)

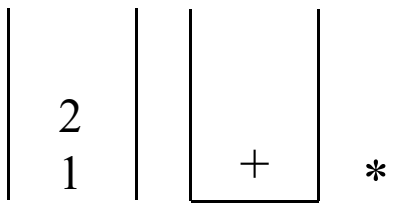
- Chuyển đổi biểu thức dạng trung tố về dạng hậu tố để có thể tính được dễ dàng.
- Ta xét cách chuyển đổi biểu thức trung tố với các phép toán ***cộng, trừ, nhân, chia, luỹ thừa và các dấu ngoặc*** về dạng hậu tố.
- Trước hết nhắc lại qui tắc tính giá trị biểu thức trung tố như vậy:
  - ***Thứ tự ưu tiên***: Luỹ thừa; Nhân/Chia; Cộng/Trừ
  - ***Qui tắc kết hợp***: Cho biết khi hai phép toán có cùng thứ tự ưu tiên thì cần thực hiện phép toán nào trước.
    - Luỹ thừa: Phải qua trái. Ví dụ: 2^2^3= 2^(2^3)=256
    - Nhân/Chia: Trái qua phải.
    - Cộng/Trừ: Trái qua phải
  - ***Dấu ngoặc*** được ưu tiên hơn cả thứ tự ưu tiên và qui tắc kết hợp

Chuyển biểu thức dạng trung tố về dạng hậu tố (Infix to Postfix Conversion)

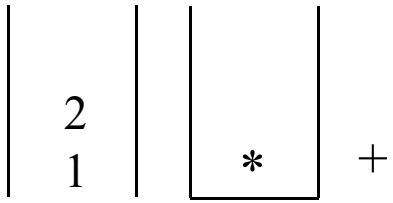
- Thuật toán cơ bản là *operator precedence parsing*.
- Ta sẽ duyệt biểu thức từ trái qua phải.
- Khi gặp toán hạng, lập tức đưa nó ra.
- Còn khi gặp phép toán thì cần làm gì?
  - Khi gặp phép toán không thể đưa nó ra ngay được, bởi vì toán hạng thứ hai còn chưa được xét.
  - Vì thế, phép toán cần được cất giữ và sẽ được đưa ra đúng lúc.
  - Sử dụng ngăn xếp để cất giữ phép toán đang xét nhưng còn chưa được đưa ra.

Ngăn xếp giữ phép toán

- Xét biểu thức `1+2*3^4+5`, biểu thức hậu tố tương đương là `1 2 3 4 ^ * + 5 +.`
- Biểu thức `1*2+3^4+5` có biểu thức hậu tố tương đương là `1 2 * 3 4 ^ + 5 +.`
- Trong cả hai trường hợp, khi phép toán thứ hai cần được xử lý thì trước đó chúng ta đã đưa ra 1 2, và có một phép toán đang nằm trong ngăn xếp. Câu hỏi là: *Các phép toán dời khỏi ngăn xếp như thế nào*?



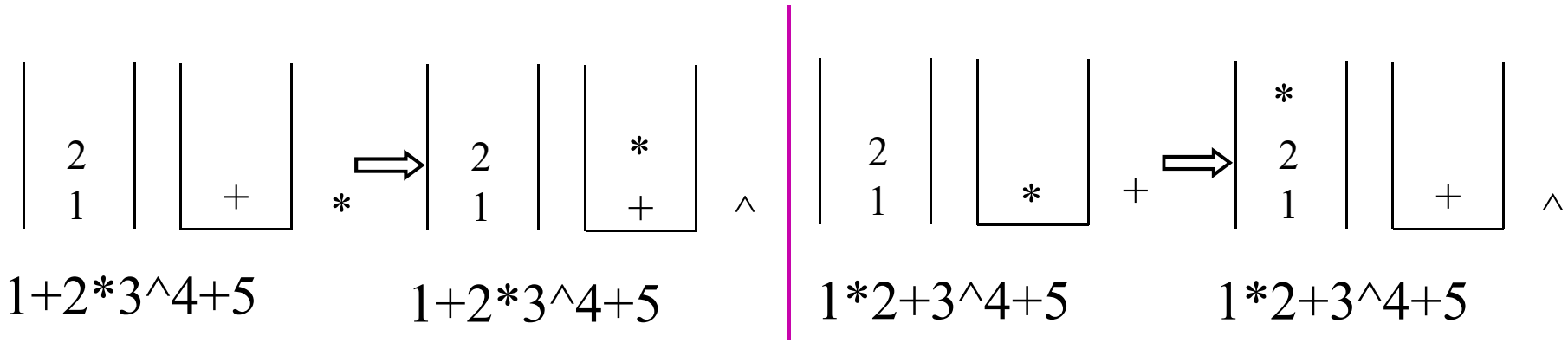
1+2*3^4+5



1*2+3^4+5

Khi nào đưa phép toán ra khỏi ngăn xếp

- Phép toán dời khỏi ngăn xếp nếu các qui tắc về trình tự và kết hợp cho thấy rằng nó cần được xử lý thay cho phép toán đang xét.
- **Qui tắc cơ bản**: Nếu phép toán đang xét có thứ tự ưu tiên ***thấp hơn*** so với phép toán ở đầu ngăn xếp, thì phép toán ở đầu ngăn xếp phải dời ngăn xếp.



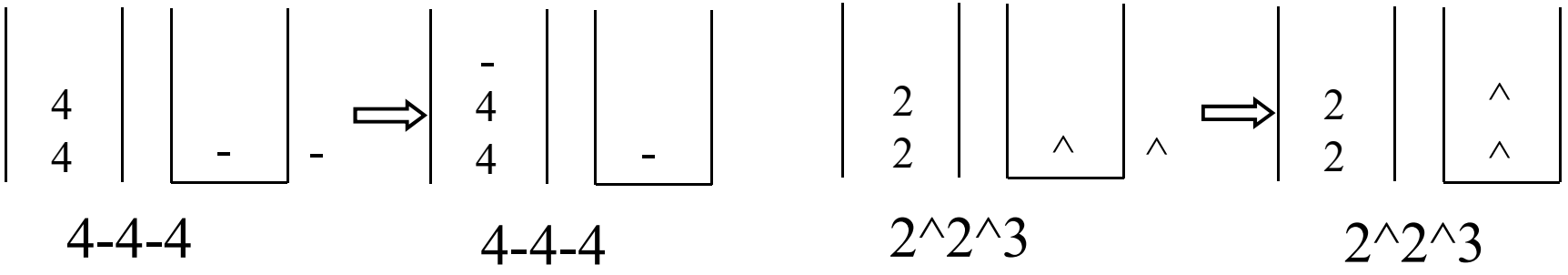
Cùng mức ưu tiên

- **Qui tắc kết hợp** cho biết cần làm gì khi phép toán đang xét có cùng thứ tự ưu tiên với phép toán ở đỉnh ngăn xếp.
  - Nếu phép toán có tính kết hợp trái (left associative), thì phép toán ở đỉnh ngăn xếp cần đưa ra

    4-4-4 => 4 4 - 4 -

  - Nếu phép toán có tính kết hợp phải (right associative), thì không đưa phép toán ở đỉnh ngăn xếp ra.

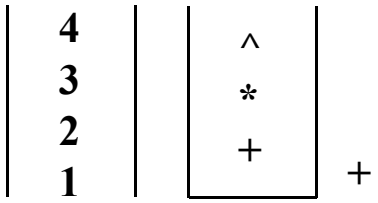
    2^2^3 => 2 2 3 ^ ^



Hàng loạt phép toán dời ngăn xếp

- Xét biểu thức **1+2*3^4+5**, với biểu thức hậu tố tương đương là

  1 2 3 4 ^ * + 5 +



Khi gặp phép toán + thứ hai, các phép toán ^ * + lần lượt được đưa ra khỏi ngăn xếp.

- Như vậy, có thể xảy ra tình huống hàng loạt phép toán dời ngăn xếp đối với cùng một phép toán đang xét.

Dấu ngoặc

- Dấu mở ngoặc có thứ tự ưu tiên hơn là phép toán khi nó được xét như là *ký tự đầu vào (input symbol)* (nghĩa là không có gì dời khỏi ngăn xếp). Dấu mở ngoặc có thứ tự ưu tiên thấp hơn phép toán khi nó *ở ngăn xếp*.
- Dấu đóng ngoặc sẽ đẩy phép toán ra khỏi ngăn xếp cho đến khi gặp dấu mở ngoặc dời khỏi ngăn xếp. Các phép toán sẽ được ghi ra còn các dấu ngoặc thì không được ghi ra.

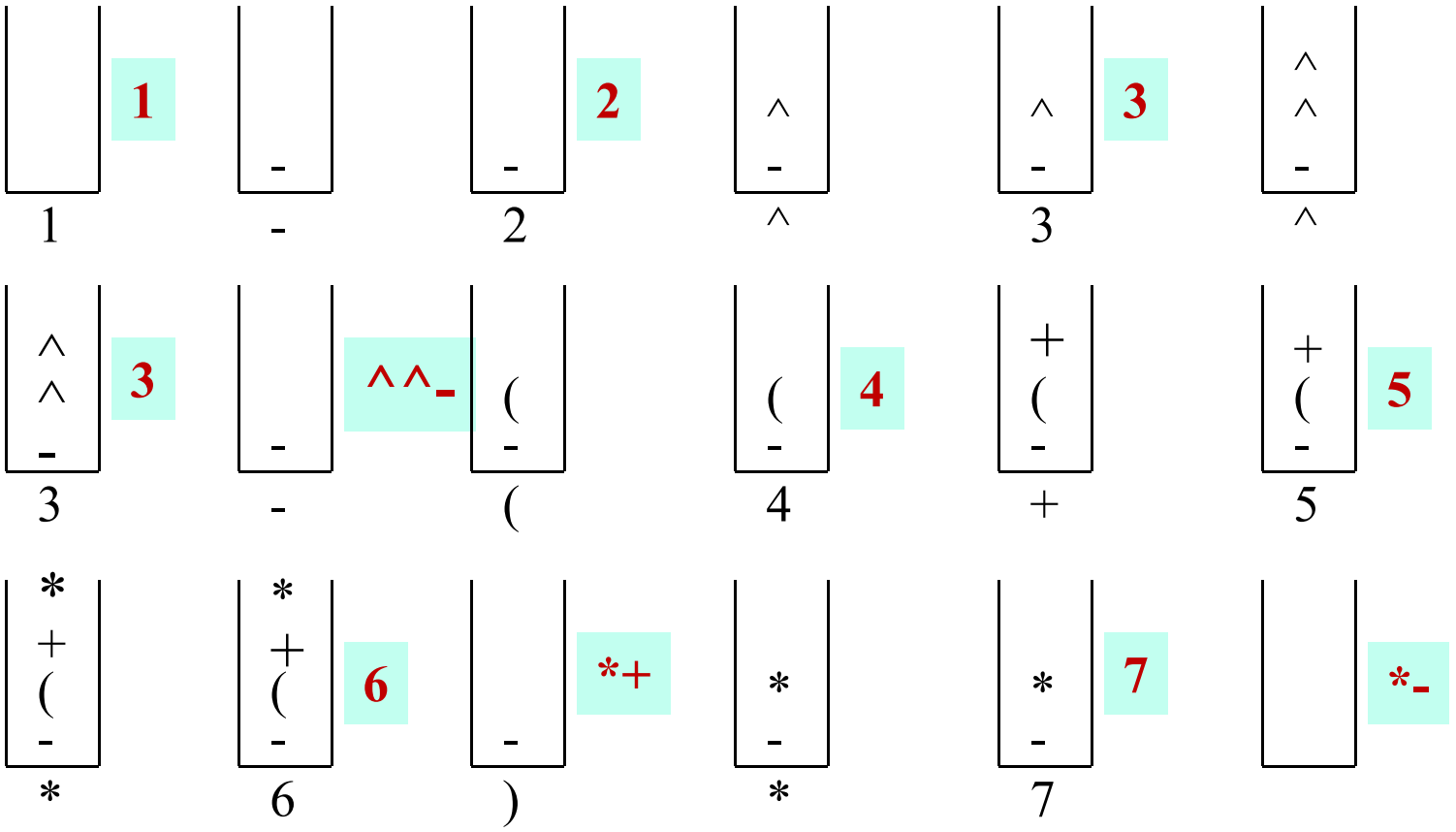
Thuật toán

- Duyệt biểu thức từ trái qua phải:
- Nếu gặp *toán hạng (Operands)*: đưa ra tức thì.
- Nếu gặp *dấu mở ngoặc* thì nạp nó vào ngăn xếp.
- Nếu gặp *dấu đóng ngoặc*: Đẩy ký hiệu ra khỏi ngăn xếp cho đến khi gặp dấu mở ngoặc đầu tiên được đẩy ra.
- Nếu gặp *phép toán (Operator)*: Đưa ra khỏi ngăn xếp tất cả các phép toán cho đến khi gặp phép toán có thứ tự ưu tiên thấp hơn hoặc gặp phép toán có tính kết hợp phải và có cùng thứ tự ưu tiên. Sau đó nạp phép toán đang xét vào ngăn xếp.
- *Khi duyệt hết biểu thức*: Đưa tất cả các phép toán còn lại ra khỏi ngăn xếp.

Ví dụ chuyển đổi trung tố về hậu tố

Infix: 1-2^3^3-(4+5*6)*7

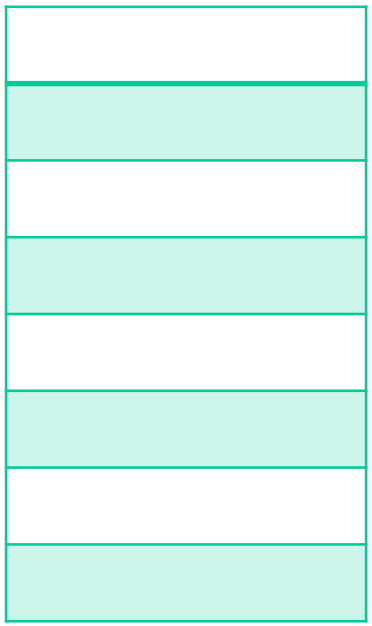
Postfix: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -



Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về postfix

(	a	/	(	b	-	c	+	d	)	)	*	(	e	-	a	)	*	c	\0



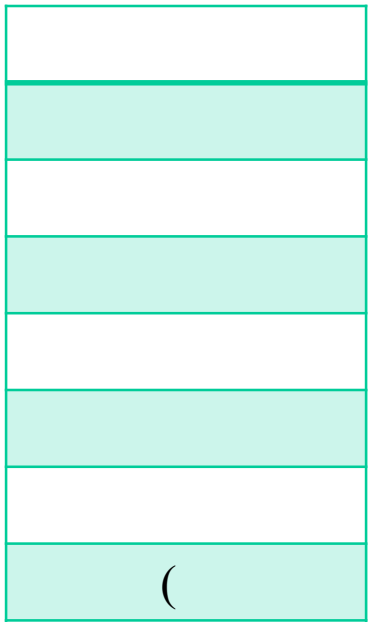
Trước hết, tạo một stack và một mảng.

Nạp toán tử vào stack và toán hạng vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

		/	(	b	-	c	+	d	)	)	*	(	e	-	a	)	*	c	\0



a												

Trước hết, tạo một stack và một mảng.

Nạp toán tử vào stack còn toán hạng vào mảng

Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.

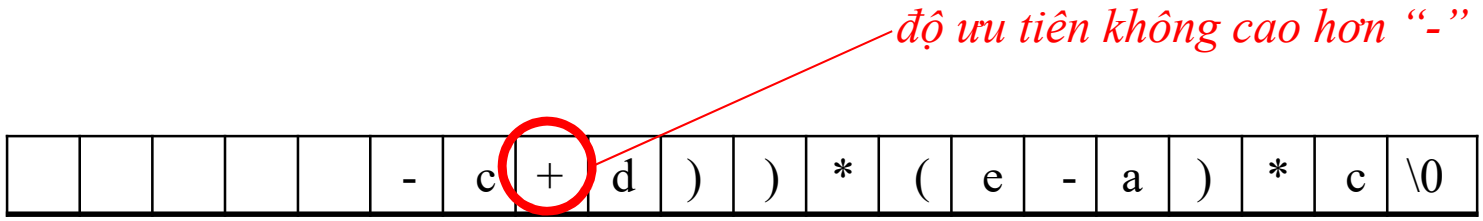
Gặp dấu "(" thì nạp ngay vào stack,

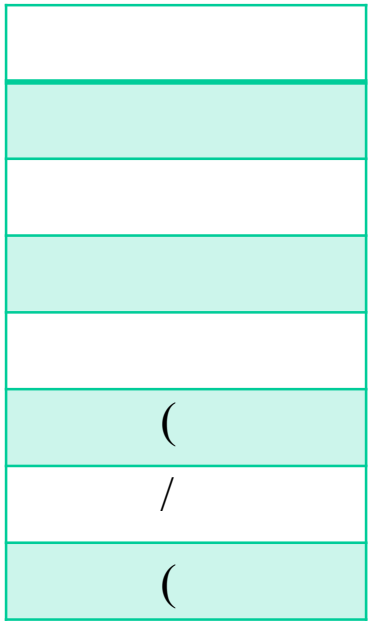
"(" chỉ dời stack khi gặp ")" .

Chú ý: Dấu "(" ở ngoài stack được coi là có độ ưu tiên cao hơn bất cứ toán tử nào, nhưng khi ở trong stack thì nó lại được coi là có độ ưu tiên thấp hơn bất cứ toán tử nào

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix



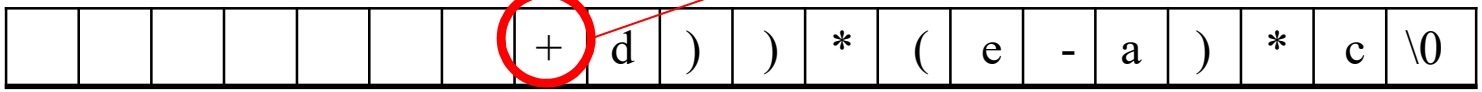


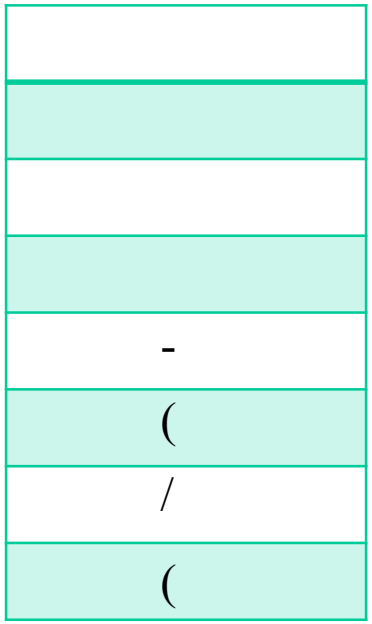
Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack,
thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

độ ưu tiên không cao hơn "-"





Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack,
thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

										)	*	(	e	-	a	)	*	c	\0

)
+
(
/
(

a	b	c	-	d								

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

										)	*	(	e	-	a	)	*	c	\0

/
(

a	b	c	-	d	+							

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

											*	(	e	-	a	)	*	c	\0

/

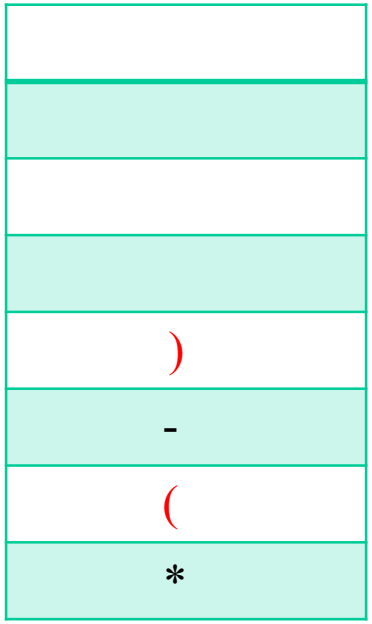
a	b	c	-	d	+							

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

độ ưu tiên không cao hơn ""*



a	b	c	-	d	+	/	e	a				

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

"\0" is lowest order operator, so pop all the operators in stack.

																		\0

*

a	b	c	-	d	+	/	e	a	-	*	c	

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack, "(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

Ví dụ: Chuyển trung tố về hậu tố

Ta sử dụng stack để chuyển infix về posfix

a	b	c	-	d	+	/	e	a	-	*	c	*

Trước hết, tạo một stack và một mảng.
Nạp toán tử vào stack còn toán hạng vào mảng
Nếu toán hạng có độ ưu tiên cao hơn toán hạng ở đầu stack, thì nạp nó vào stack, trái lại đưa toán hạng ở đầu stack ra mảng.
Gặp dấu "(" thì nạp ngay vào stack,
"(" chỉ dời stack khi gặp ")" .
Nếu nạp dấu ")" thì đưa tất cả các toán tử nằm giữa ( ) trong stack ra mảng, các dấu ( ) cũng được đưa ra khỏi stack nhưng không cất vào mảng.

3.4. Ngăn xếp

3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

3.4.2. Ngăn xếp dùng mảng

3.4.3. Cài đặt ngăn xếp với danh sách móc nối

3.4.4. Một số ứng dụng của ngăn xếp

- Ứng dụng 1: Ngoặc hợp cách
- Ứng dụng 2: Sánh đôi thẻ trong HTML
- Ứng dụng 3. Bài toán đổi cơ số
- Ứng dụng 4. Bài toán tính giá trị biểu thức số học
- **Ứng dụng 5. Ngăn xếp và đệ qui**

Stacks và đệ qui

- Mỗi hàm đệ qui đều có thể cài đặt sử dụng ngăn xếp và lặp (*Every recursive function can be implemented using a stack and iteration*).

- Mỗi hàm lặp có sử dụng ngăn xếp đều có thể cài đặt sử dụng đệ qui. (*Every iterative function which uses a stack can be implemented using recursion.*)

Ví dụ

Xét hàm tính luỹ thừa được cài đặt đệ qui:

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
            tmp = tmp*tmp;
        else
            tmp = tmp*tmp*x;
    }
    return tmp;
}
```

Xét việc thực hiện power(2,5)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5

tmp = 1

power(2,5) → power(2,2)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5

tmp = 1

x = 2, n = 2

tmp = 1

power(2,5) → power(2,2) → power(2,1)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5

tmp = 1

x = 2, n = 2

tmp = 1

x = 2, n = 1

tmp = 1

power(2,5) → power(2,2) → power(2,1) ⇄ power(2,0)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

x = 2, n = 2
tmp = 1

x = 2, n = 1
tmp = 1

x = 2, n = 0
tmp = 1

power(2,5) → power(2,2) ← power(2,1)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

x = 2, n = 2
tmp = 1

x = 2, n = 1
tmp = 1*1*2
= 2

power(2,5) ← power(2,2)

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5

tmp = 1

x = 2, n = 2

tmp = 2*2
= 4

power(2,5) trả lại 32

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5

*tmp = 4*4*2*
= 32

Tổ chức ngăn xếp cất giữ trạng thái của các lần gọi đệ qui

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

Lần gọi 4	tmp	1
	n	0
	x	2
Lần gọi 3	tmp	1
	n	1
	x	2
Lần gọi 2	tmp	1
	n	2
	x	2
Lần gọi 1	tmp	1
	n	5
	x	2

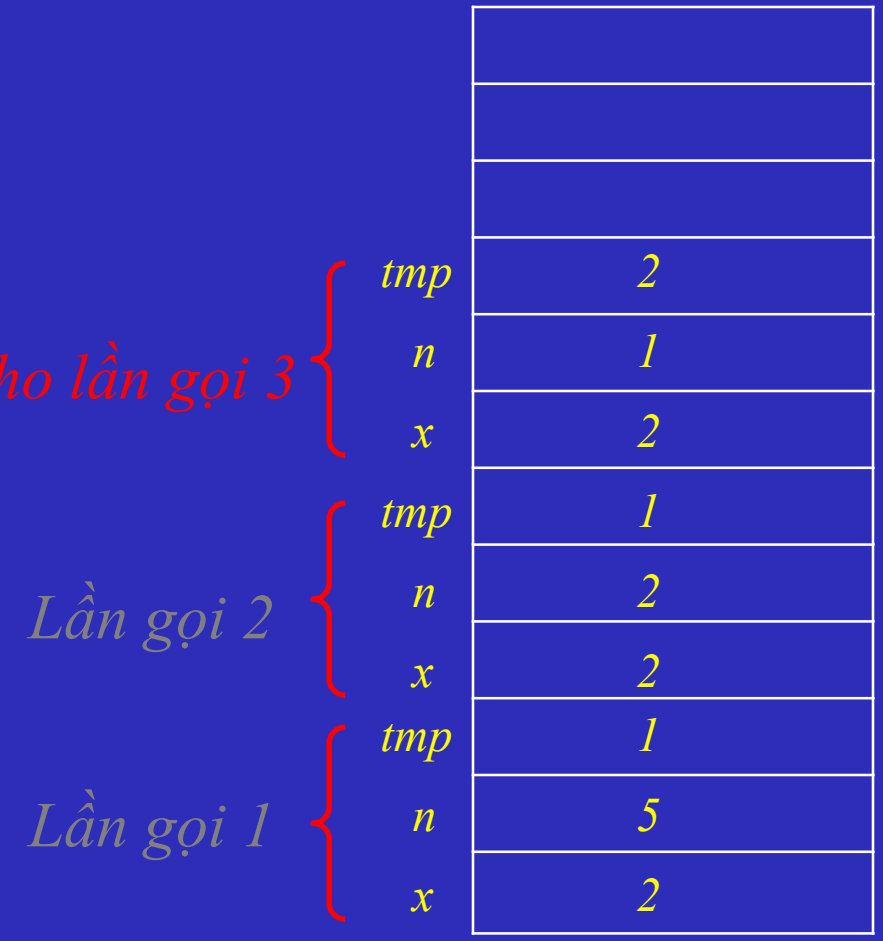
Cất giữ vào ngăn xếp

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

Trả lại cho lần gọi 3

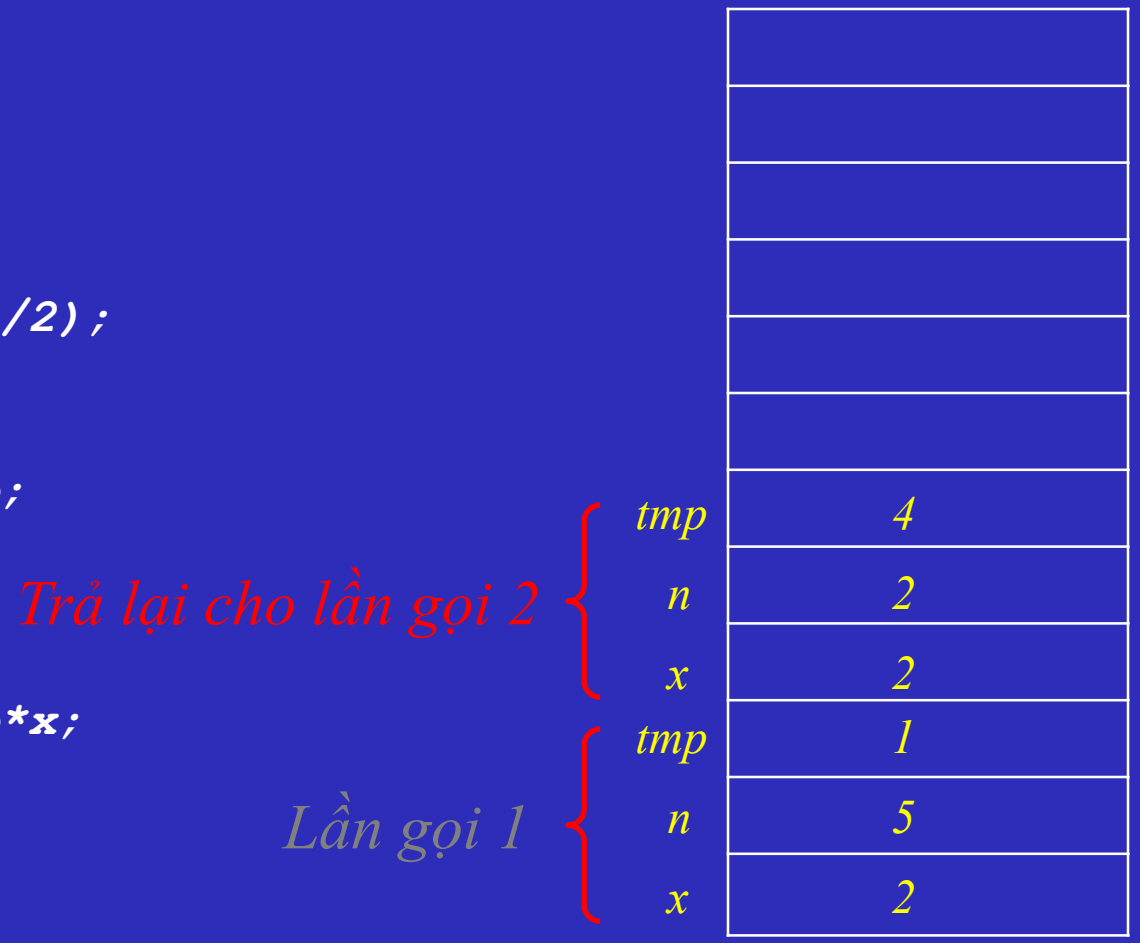
Lần gọi 2

Lần gọi 1



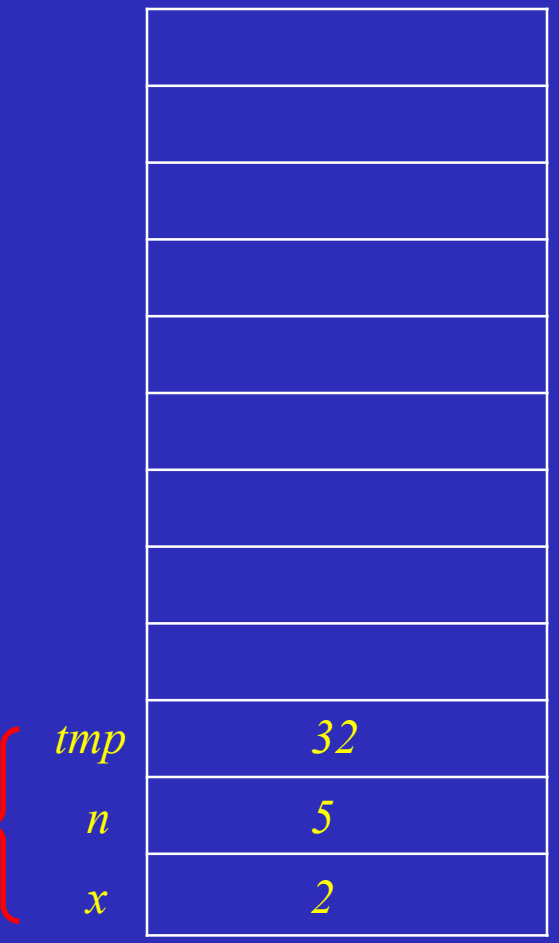
Ngăn xếp

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```



```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

Trả lại cho lần gọi 1

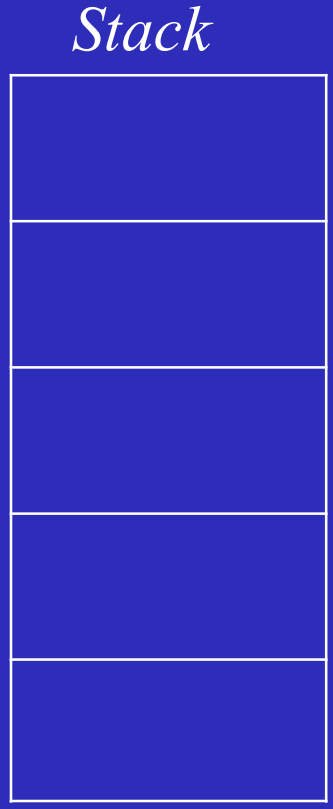


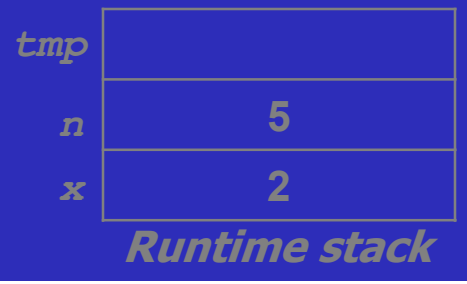
Ngăn xếp

Thuật toán lặp power(x,n) tính x^n sử dụng Stack

$n = 5, x = 2$

```
module power(x, n)
{
    create a Stack
⟶  initialize a Stack
    loop{
       if (n == 0) then {exit loop}
       push  n  onto Stack
       n = n/2
    }
    tmp = 1
    loop {
       if (Stack is empty) then {return tmp}
       pop n off Stack
       if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

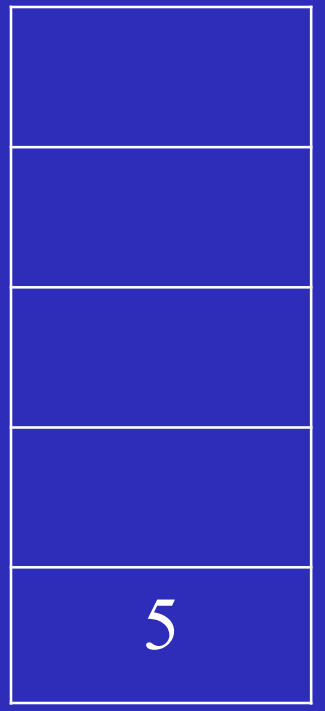


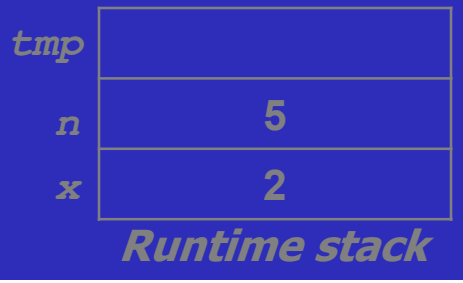


```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push  n  onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

$n = 5, x = 2$

Stack

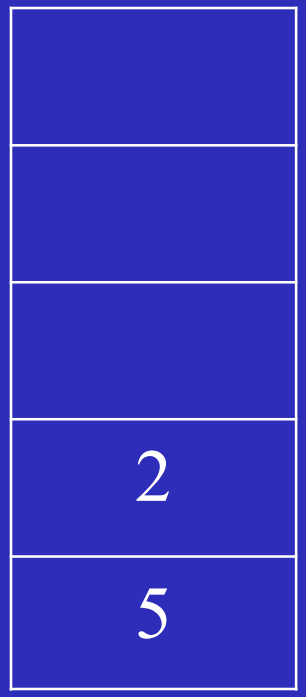


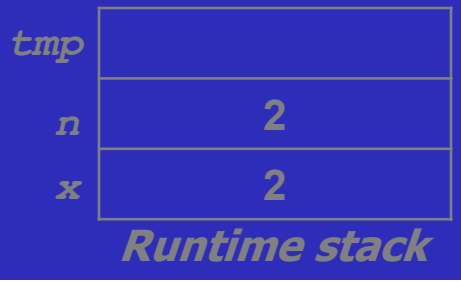


```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
  ⟶    push  n  onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

n = 2, x = 2

Stack

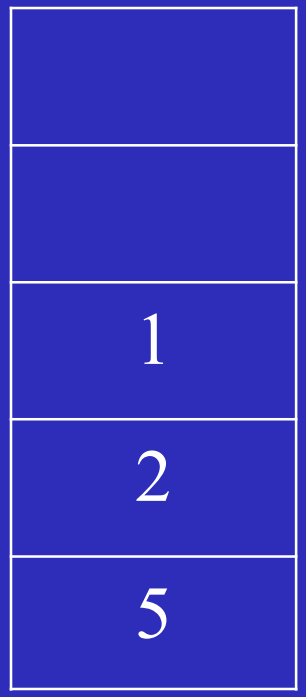


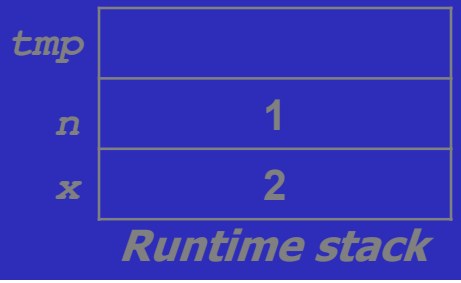


```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
  ⟶    push  n  onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

n = 1, x = 2

Stack



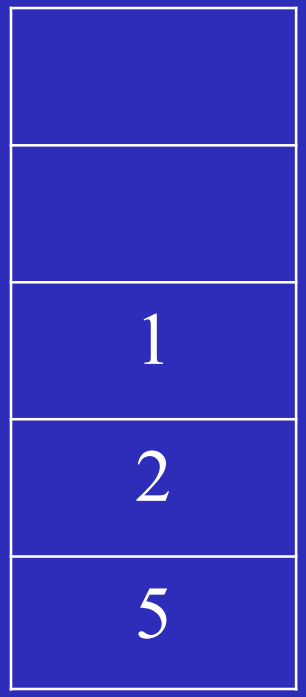


```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push  n  onto Stack
        n = n/2
    }
⟶  tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

n = 0, x = 2

tmp = 1

Stack

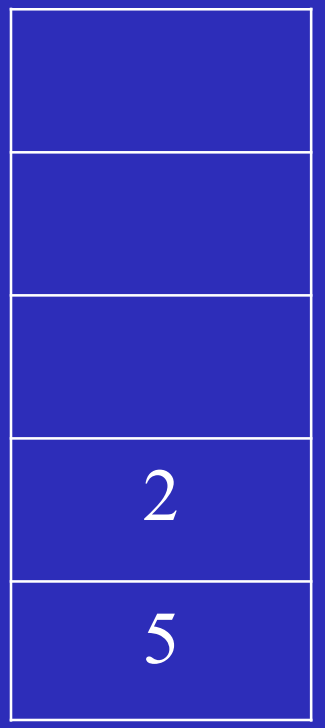


```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push  n  onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
→       if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

n = 0, x = 2

tmp = 2

Stack

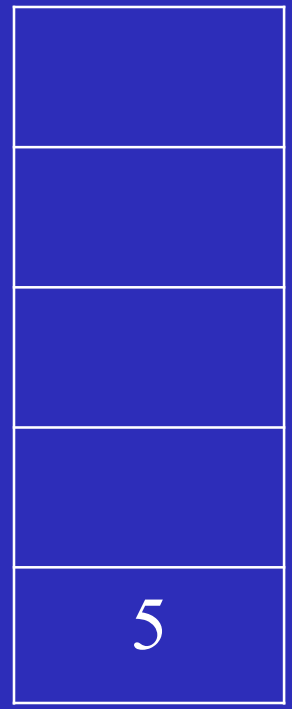


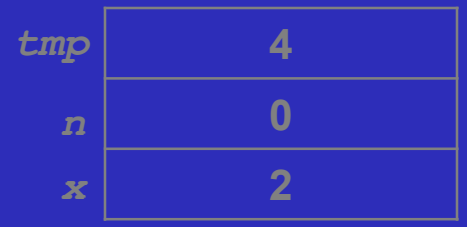
n = 0, x = 2

tmp = 4

```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
       if (n == 0) then {exit loop}
       push  n  onto Stack
       n = n/2
    }
    tmp = 1
    loop {
       if (Stack is empty) then {return tmp}
       pop n off Stack
→      if (n is even) {tmp = tmp*tmp}
     else {tmp = tmp*tmp*x}
    }
}
```

Stack





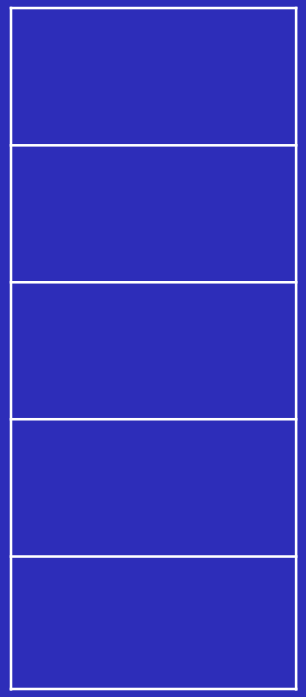
Runtime stack

```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push  n  onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
      else {tmp = tmp*tmp*x}
    }
}
```

n = 0, x = 2

tmp = 32

Stack



tmp	32
n	0
x	2

Runtime stack

Cài đặt không đệ qui sử dụng stack

```
double power(double x, int n)
{
   double tmp = 1;
   Stack theStack;

   initializeStack(&theStack);
   while (n != 0) {
      push(&theStack, n);
      n /= 2;
   }

   while (!stackEmpty(&theStack)) {
      n = pop(&theStack);
      if (n % 2 == 0)
      { tmp = tmp*tmp;}
      else
      { tmp = tmp*tmp*x;}
   }
   return tmp;
}
```

Cách mô tả này gần với C hơn !

Chương 3. Các cấu trúc dữ liệu cơ bản

3.1. Các khái niệm

3.2. Mảng

3.3. Danh sách

3.4. Ngăn xếp

3.5. Hàng đợi

3.5. Hàng đợi

3.5.1. Kiểu dữ liệu trừu tượng hàng đợi

3.5.2. Cài đặt hàng đợi bằng mảng

3.5.3. Cài đặt hàng đợi bởi danh sách móc nối

3.5.4. Một số ví dụ ứng dụng hàng đợi

Ứng dụng 1. Chuyển đổi xâu chữ số thành số thập phân

Ứng dụng 2. Nhận biết Palindromes

ADT hàng đợi (ADT queues)

- **Hàng đợi** là *danh sách có thứ tự* trong đó phép toán chèn luôn thực hiện chỉ ở một phía gọi là **phía sau** hay **cuối** (**back or rear**), còn phép toán xoá chỉ thực hiện ở phía còn lại gọi là **phía trước** hay **đầu** (**front or head**).
- Thuật ngữ thường dùng cho hai thao tác **chèn và xoá** đối với hàng đợi tương ứng là **đưa vào** (**enqueue**) và **đưa ra** (**dequeue**).
- 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**).

ADT hàng đợi (ADT queues)

- 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.

Using a Queue

Queues Everywhere!

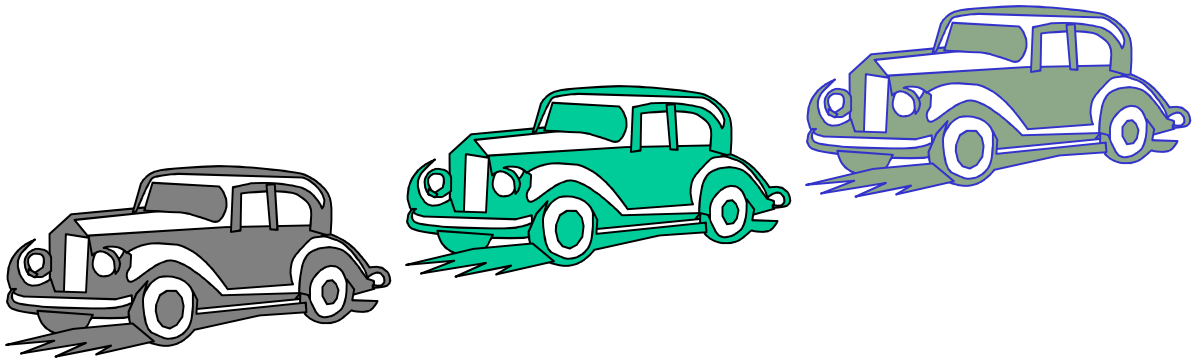
Các phép toán

- Q = init(); Khởi tạo Q là hàng đợi rỗng.
- isEmpty(Q); Trả lại "true" khi và chỉ khi hàng đợi Q là rỗng.
- isFull(Q); Trả lại "true" khi và chỉ khi hàng đợi Q là tràn, cho biết là ta đã sử dụng vượt quá kích thước tối đa dành cho hàng đợi.
- front(Q); Trả lại phần tử ở phía trước (front) của hàng đợi Q hoặc gặp lỗi nếu hàng đợi rỗng.
- enqueue(Q,x); Chèn phần tử x vào phía sau (back) hàng đợi Q. Nếu việc chèn dẫn đến tràn hàng đợi thì cần thông báo về điều này.
- dequeue(Q,x); Xoá phần tử ở phía trước hàng đợi, trả lại x là thông tin chứa trong phần tử này. Nếu hàng đợi rỗng thì cần đưa ra thông báo lỗi.
- print(Q); Đưa ra danh sách tất cả các phần tử của hàng đợi Q theo thứ tự từ phía trước đến phía sau.
- size(Q); Trả lại số lượng phần tử trong hàng đợi Q.

Ví dụ

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	5	(3)
enqueue(7)	–	(3, 7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	“error”	()
isEmpty()	true	()
size()	0	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

Queues



Ứng dụng của hàng đợi

- Ứng dụng trực tiếp
  - Danh sách xếp hàng chờ mua vé tàu xe, chờ gửi xe, chờ được phục vụ ở hàng ăn, chờ mượn sách ở thư viện, ...
  - Chia sẻ các tài nguyên (ví dụ, printer, CPU, bộ nhớ, ...)
  - Tổ chức thực hiện đa chương trình (Multiprogramming)
  - ...
- Ứng dụng gián tiếp (Indirect applications)
  - Cấu trúc dữ liệu bổ trợ cho các thuật toán
  - Là thành phần của những cấu trúc dữ liệu khác
  - ...

3.5. Hàng đợi

3.5.1. Kiểu dữ liệu trừu tượng hàng đợi

3.5.2. Cài đặt hàng đợi bằng mảng

3.5.3. Cài đặt hàng đợi bởi danh sách móc nối

3.5.4. Một số ví dụ ứng dụng hàng đợi

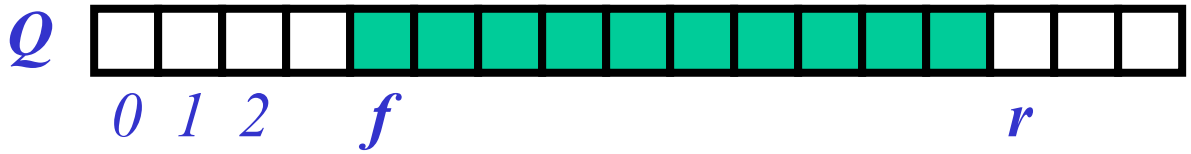
Ứng dụng 1. Chuyển đổi xâu chữ số thành số thập phân

Ứng dụng 2. Nhận biết Palindromes

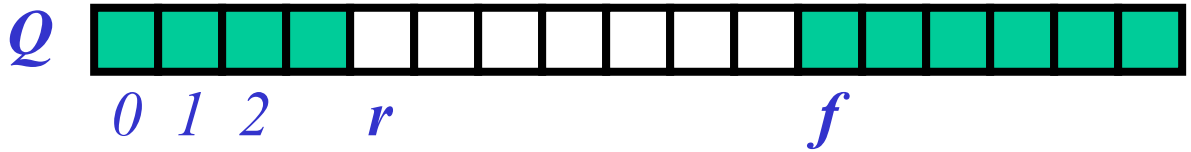
Cài đặt hàng đợi bằng mảng
Array-based Queue

- Sử dụng mảng *Q* kích thước *N* theo thứ tự vòng tròn
- Có hai biến để lưu trữ vị trí **đầu** và **cuối** (***f***ront and ***r***ear)
  - ***f*** chỉ số của phần tử ở đầu hàng đợi
  - ***r*** chỉ số của vị trí ở ngay sau vị trí của phần tử cuối cùng của hàng đợi
- Vị trí ***r*** được giữ là rỗng

cấu hình bình thường



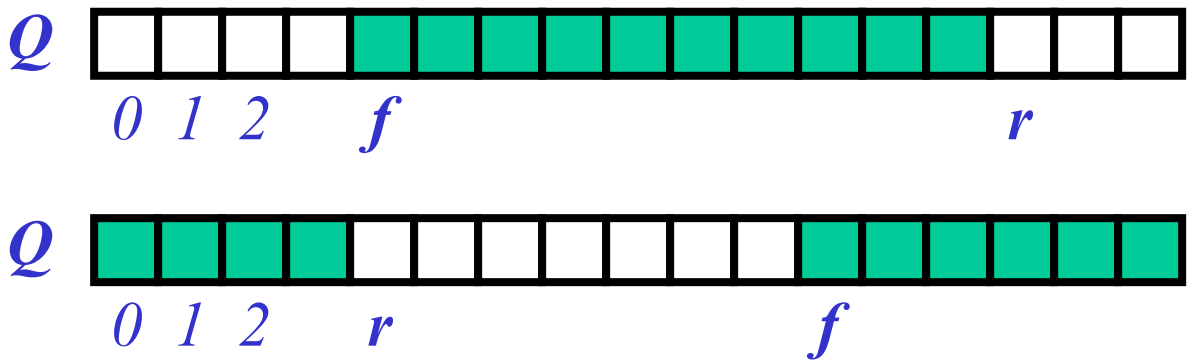
cấu hình xoay vòng tròn



Các phép toán

- Ta sử dụng phép toán theo modulo (phần dư của phép chia)

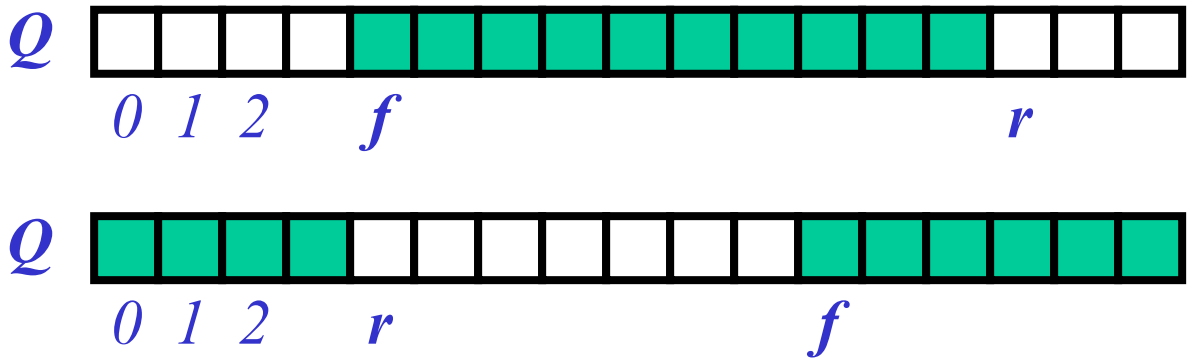
```
Algorithm size()
    return (N – f + r) mod N

Algorithm isEmpty()
    return (f = r)
```



Phép toán chèn - Đưa vào (enqueue)

- Phép toán enqueue phải để ý đến lỗi tràn hàng đợi.
- Lỗi này cần xử lý bởi người sử dụng.

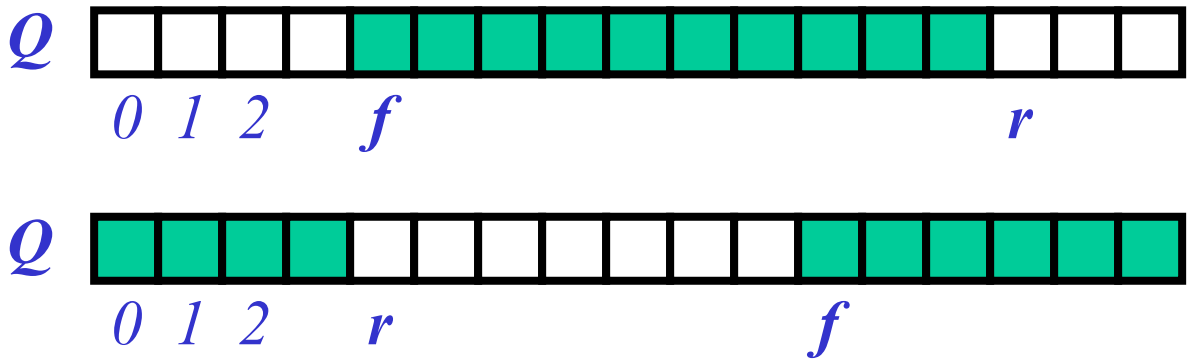
```
Algorithm enqueue(o)
  if size() = N – 1 then
    Error("FullQueue")
  else
    Q[r] ← o
    r ← (r + 1) mod N
```



Phép toán loại bỏ - Đưa ra (dequeue)

- Phép toán loại bỏ (dequeue) cần xử lý lỗi hàng đợi rỗng

```
Algorithm dequeue()
  if isEmpty() then
     Error("EmptyQueue")
  else
     o ← Q[f]
     f ← (f + 1) mod N
     return o
```



Demo: d:\..\0DEMOCODE\QueueArray.c

3.5. Hàng đợi

3.5.1. Kiểu dữ liệu trừu tượng hàng đợi

3.5.2. Cài đặt hàng đợi bằng mảng

3.5.3. Cài đặt hàng đợi bởi danh sách móc nối

3.5.4. Một số ví dụ ứng dụng hàng đợi

Ứng dụng 1. Chuyển đổi xâu chữ số thành số thập phân

Ứng dụng 2. Nhận biết Palindromes

Cài đặt hàng đợi bởi danh sách móc nối

- Ta có thể cài đặt hàng đợi bởi danh sách móc nối đơn hoặc đôi.
- **Ví dụ:** khai báo sau đây được sử dụng để mô tả hàng đợi bởi danh sách móc nối đơn:

```
typedef struct _node {
          DataType element;
          struct _node *next; } node;
typedef struct { node *front; node *back; } queue;
```

trong đó DataType là kiểu dữ liệu của đối tượng cần lưu giữ, được khai báo trước.

- Các phép toán đối với hàng đợi mô tả bởi danh sách móc nối được cài đặt tương tự như đối với danh sách móc nối đã trình bày ở trên. Ta sẽ không trình bày lại chi tiết ở đây.

3.5. Hàng đợi

3.5.1. Kiểu dữ liệu trừu tượng hàng đợi

3.5.2. Cài đặt hàng đợi bằng mảng

3.5.3. Cài đặt hàng đợi bởi danh sách móc nối

3.5.4. Một số ví dụ ứng dụng hàng đợi

Ứng dụng 1. Chuyển đổi xâu chữ số thành số thập phân

Ứng dụng 2. Nhận biết Palindromes

Ví dụ: Chuyển đổi xâu chữ số thành 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 ở đầu (nếu có)
do {        dequeue(Q, ch)
} until ( ch != blank)
//  ch chứa chữ số đầu tiên
// 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 cần tìm
```

Ví dụ: Nhận biết *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
- Một trong những cách nhận biết một xâu cho trước có phải là palindrome hay không là ta đưa các ký tự của nó đồng thời vào một hàng đợi và một ngăn xếp. 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 minh hoạ thuật toán: "RADAR"

Bước 1: Đưa "RADAR" vào Queue và Stack:

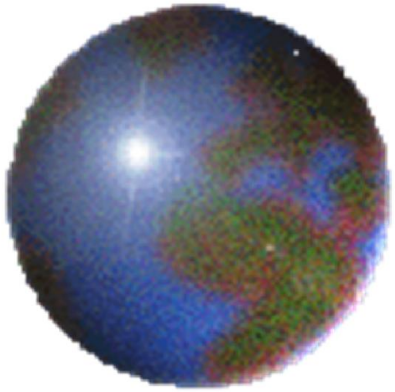
Ký tự hiện thời	Queue (cuối ở bên phải)	Stack (top ở bên trái)
R	R	R
A	R A	A R
D	R A D	D A R
A	R A D A	A D A R
R	R A D A R	R A D A R

Nhận biết "RADAR" có phải palindrome

Bước 2: Xoá bỏ “RADAR” khỏi Queue và Stack:

Queue (head ở bên trái)	head 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



UESTION?

Hết chương 3