REVIEW

- Dùng định lý thợ để đưa ra các tiệm cận chặt cho các công thức đệ quy sau

a) $T(n) = 3T\left(\frac{n}{2}\right) + n$

b) $T(n) = 5T\left(\frac{n}{2}\right) + n^2$

Chapter 2

Các cấu trúc dữ liệu cơ bản

Nội dung

- Các khái niệm cơ bản
- Mảng và mảng động
- Con trỏ và cấu trúc liên kết
- Danh sách tuyến tính

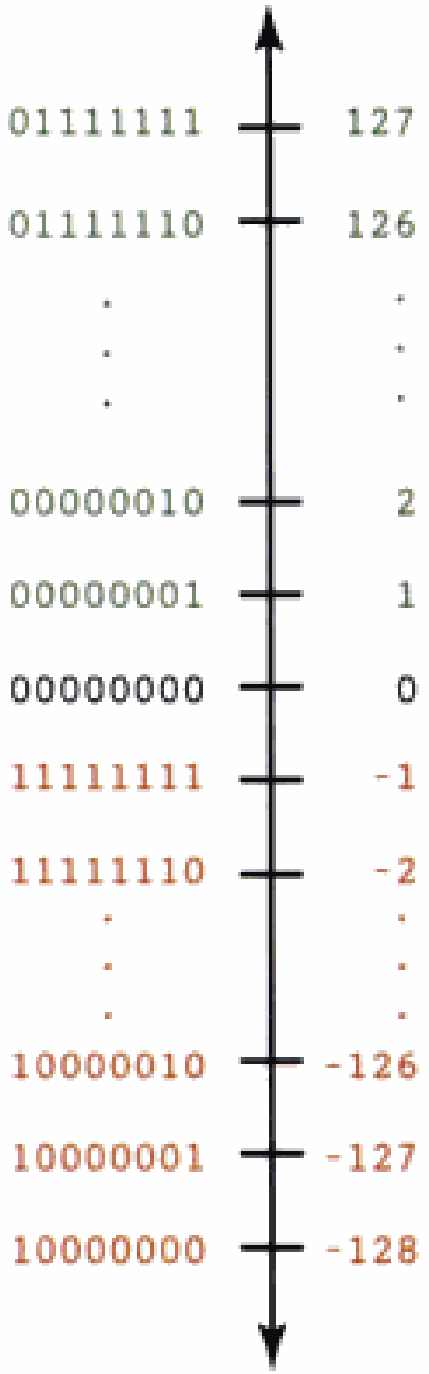
2.1 CÁC KHÁI NIỆM CƠ BẢN

2.1 Khái niệm cơ bản

- Xử lý dữ liệu trên máy tính xét cho cùng là xử lý với các bit

- **Một kiểu dữ liệu** (data type): là một tập **các giá trị** và nhóm **các phép toán** được thực hiện trên các giá trị đó.
  - Chỉ ra cách sử dụng các nhóm bit và các phép toán thực hiện trên các nhóm bit

- VD. Kiểu số nguyên char
  số bit : 8 bit
  các phép toán +, -, *, /, %



Khái niệm cơ bản

- **Các kiểu dữ liệu dựng sẵn** (Built-in data types): được xây dựng sẵn trong ngôn ngữ lập trình

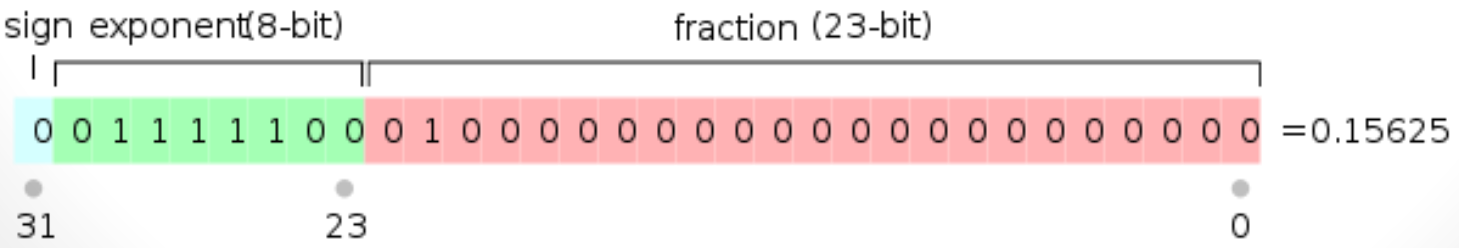
Type	Macintosh Metrowerks CW (Default)	Linux on a PC	IBM PC Windows XP Windows NT	ANSI C Minimum
`char`	8	8	8	8
`int`	32	32	32	16
`short`	16	16	16	16
`long`	32	32	32	32
`long long`	64	64	64	64

Khái niệm cơ bản

- Chuẩn **IEEE754/85:**

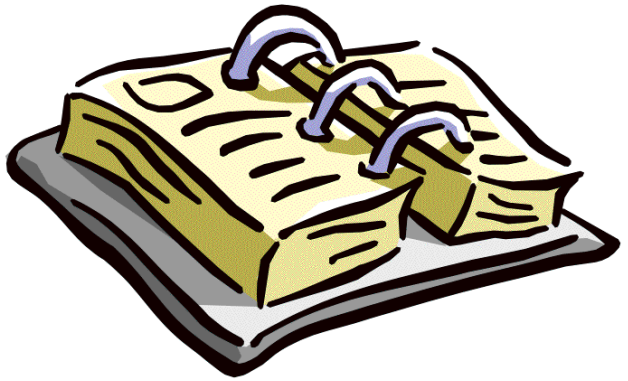
Type	Dấu (sign)	Mũ (Exponent)	Độ lệch mũ (Exponent Bias)	Giá trị (fraction)	Tổng cộng (bit)
Half (IEEE 754-2008)	1	5	15	10	16
Single	1	8	127	23	32
Double	1	11	1023	52	64
Quad	1	15	16383	112	128

$$v = (-1)^{sign} \times 2^{exponent - exponent\,bias} \times (1 + fraction)$$



Khái niệm cơ bản

- Kiểu dữ liệu trừu tượng (Abstract DataType - ADT) gồm:
  - Tập các giá trị
  - Tập các phép toán có thể thực hiện trên các giá trị này
- Cách biểu diễn cụ thể bị bỏ qua khi xét đến ADT.
  - Làm trừu tượng hóa kiểu dữ liệu, không phụ thuộc ngôn ngữ lập trình cụ thể.
- Cài đặt ADT là biểu diễn ADT bởi một ngôn ngữ lập trình cụ thể
  - Xét đến một biểu diễn cụ thể cho ADT
- Các kiểu dữ liệu dựng sẵn chính là cài đặt của các ADT tương ứng bằng ngôn ngữ lập trình cụ thể.



Khái niệm cơ bản

- **Cấu trúc dữ liệu (data structure):** Gồm các kiểu dữ liệu và cách liên kết giữa chúng.

- **Cấu trúc dữ liệu** mô tả cách tổ chức và lưu trữ dữ liệu trên máy tính để sử dụng một cách hiệu quả nhất.

- **Hai vấn đề** của một cấu trúc dữ liệu:
  - **Các thao tác mà nó hỗ trợ, và**
  - **Cách cài đặt các thao tác này**

Khái niệm cơ bản

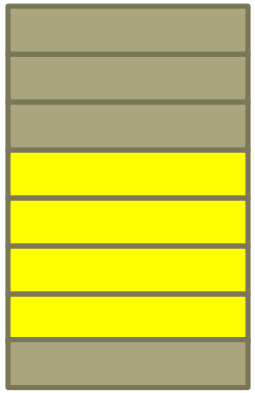
- Thay đổi cấu trúc dữ liệu **không làm thay đổi tính chính xác** của chương trình. Tuy nhiên nó sẽ làm thay đổi **hiệu quả** của chương trình.

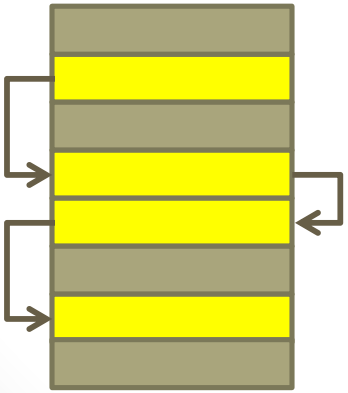
- Tốt nhất nên chọn cấu trúc dữ liệu cho hiệu quả cao nhất ngay từ khi thiết kế chương trình!

Cấu trúc liên tục VS liên kết

- Các cấu trúc dữ liệu có thể được chia thành liên tục (*contiguous*) hoặc liên kết(*linked*), tùy vào việc nó được cài đặt dựa trên mảng hay con trỏ.

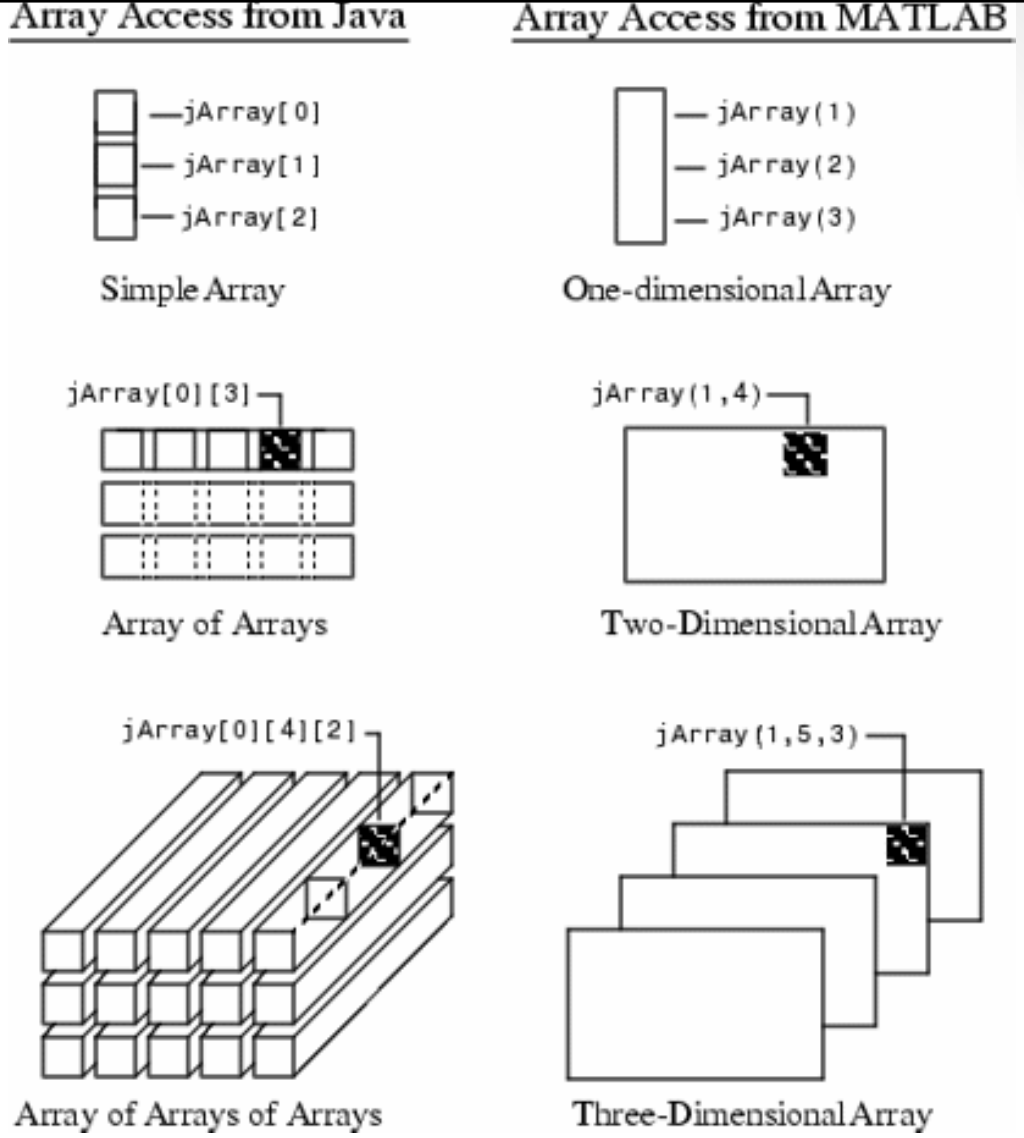
Cấu trúc được cấp phát liên tục: được cấp phát thành vùng bộ nhớ liên tục. VD mảng, ma trận, đống (heap), và bảng băm





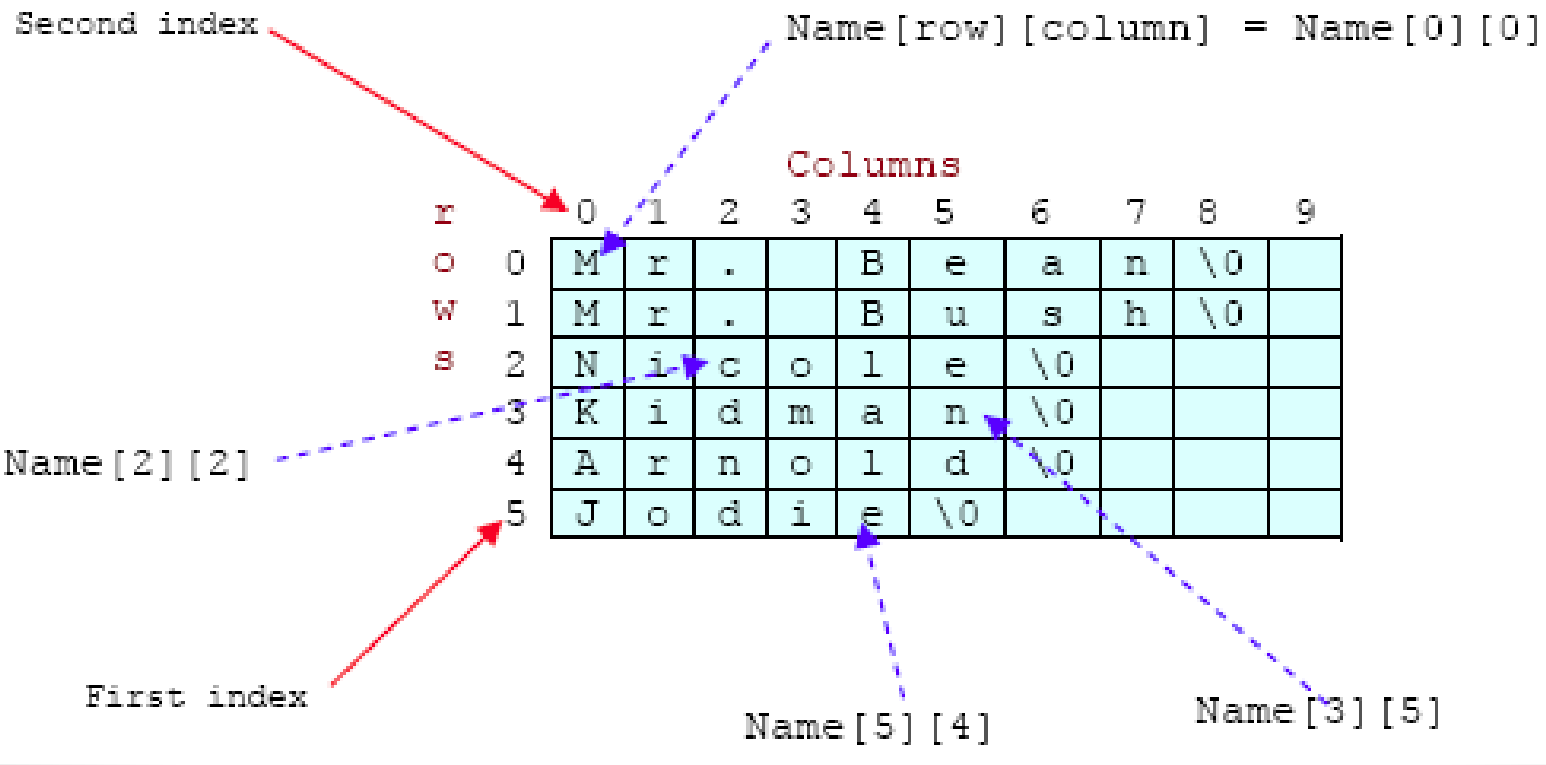
Cấu trúc dữ liệu liên kết: gồm các đoạn(chunk) trong bộ nhớ (không nằm liên tục) và được liên kết với nhau thông qua con trỏ. VD, danh sách, cây, và đồ thị danh sách kề.

2.2 ARRAY – MẢNG



Mảng

- **Mảng** : gồm các bản ghi có kiểu giống nhau, có kích thước cố định. Mỗi phần tử được xác định bởi chỉ số (địa chỉ)
- Mảng là cấu trúc dữ liệu được cấp phát liên tục cơ bản.



Mảng

- **Ưu điểm** của mảng:
  - **Truy cập phần tử với thời gian hằng số $O(1)$**: vì thông qua chỉ số của phần tử ta có thể truy cập trực tiếp vào ô nhớ chứa phần tử.
  - **Sử dụng bộ nhớ hiệu quả**: chỉ dùng bộ nhớ để chứa dữ liệu nguyên bản, không lãng phí bộ nhớ để lưu thêm các thông tin khác.
  - **Tính cục bộ về bộ nhớ**: các phần tử nằm liên tục trong 1 vùng bộ nhớ, duyệt qua các phần tử trong mảng rất dễ dàng và nhanh chóng.
- **Nhược điểm**: không thể thay đổi kích thước của mảng khi chương trình đang thực hiện.

Mảng

- **Mảng động (dynamic array)**: cấp phát bộ nhớ cho mảng một cách động trong quá trình chạy chương trình.
  Trong C là ***malloc*** và ***calloc***, trong C++ là ***new***
- Sử dụng mảng động ta bắt đầu với mảng chỉ có 1 phần tử, mỗi khi số lượng phần tử vượt quá khả năng của mảng thì ta lại **gấp đôi kích thước** của mảng cũ và **copy các phần tử** mảng cũ vào nửa đầu của mảng mới.

- **Ưu điểm**: tránh lãng phí bộ nhớ khi phải khai báo mảng có kích thước lớn ngay từ đầu
- **Nhược điểm**: phải thực hiện thêm các thao tác copy phần tử mỗi khi thay đổi kích thước.

Mảng động

- Từ mảng 1 phần tử tới n phần tử, số lần phải thay đổi kích thước là $\log_2 n$

- Số phần tử phải di chuyển

$$M = \sum_{i=1}^{\log n} i * \frac{n}{2^i} = n * \sum_{i=1}^{\log n} \frac{i}{2^i} < n * \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

  Thời gian để duy trì mảng chỉ là $O(n)$

- **Nhược điểm**: một số thời gian thực hiện một số thao tác không còn đúng là hằng số nữa

2.3 CON TRỎ VÀ CẤU TRÚC LIÊN KẾT

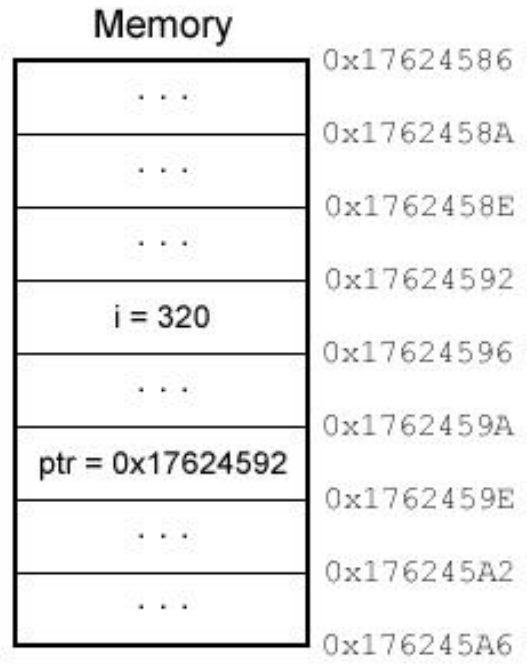
- Con trỏ và cấu trúc liên kết
- Danh sách liên kết đơn
- Các dạng khác của danh sách liên kết

Con trỏ và cấu trúc liên kết

- **Con trỏ** lưu trữ địa chỉ của một vị trí trong bộ nhớ.
  VD. Visiting card có thể xem như con trỏ trỏ đến nơi làm việc của một người nào đó.

- Trong cấu trúc liên kết con trỏ được dùng để liên kết giữa các phần tử.

- Trong C/C++ :
  - ***p** chỉ **p** là một biến con trỏ
  - **&x** chỉ địa chỉ của biến **x** trong bộ nhớ
  - Con trỏ **NULL** chỉ biến con trỏ chưa được gán giá trị (không trỏ vào đâu cả)




```
int i = 320;
int* ptr = &i;
```

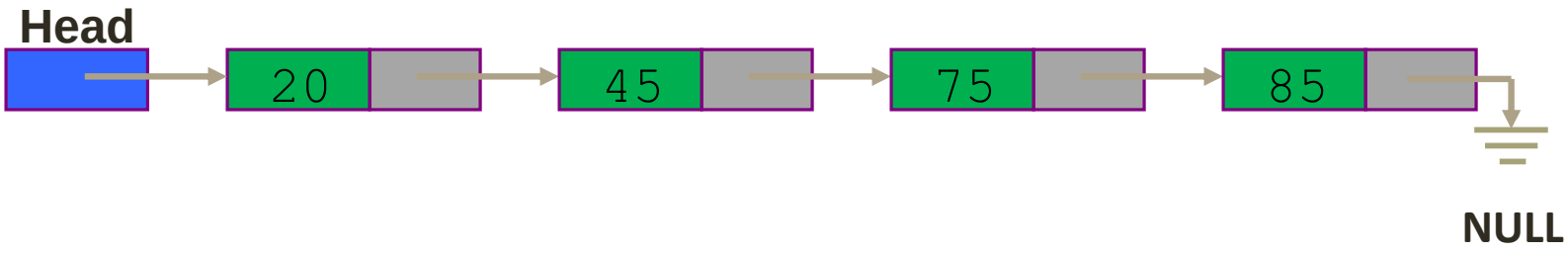
Con trỏ và cấu trúc liên kết

- Tất cả các cấu trúc liên kết đều có đặc điểm giống với khai báo danh sách liên kết đơn (singly-linked list)sau:

```
typedef struct list {
        item_type item;      /* data item */
        struct list *pNext;  /* point to successor */
} list;
```

- Mỗi nút có 1 hay nhiều trường dữ liệu (item) chứa dữ liệu ta cần lưu trữ
- Mỗi nút có ít nhất 1 con trỏ trỏ đến nút tiếp theo (pNext). Do đó cấu trúc kết nối cần nhiều bộ nhớ hơn cấu trúc liên tục.
- Cuối cùng, ta cần 1 con trỏ trỏ đến đầu cấu trúc để chỉ ra phần tử bắt đầu của cấu trúc.

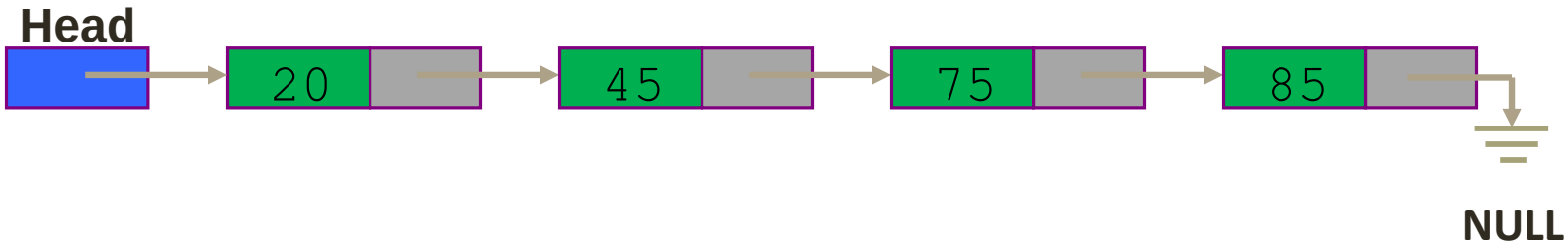
Cấu trúc liên kết



- Đặc điểm của cấu trúc liên kết:
  - Cần thêm bộ nhớ phụ để lưu các con trỏ
  - Không cho phép truy cập phần tử một cách ngẫu nhiên

Cấu trúc liên kết

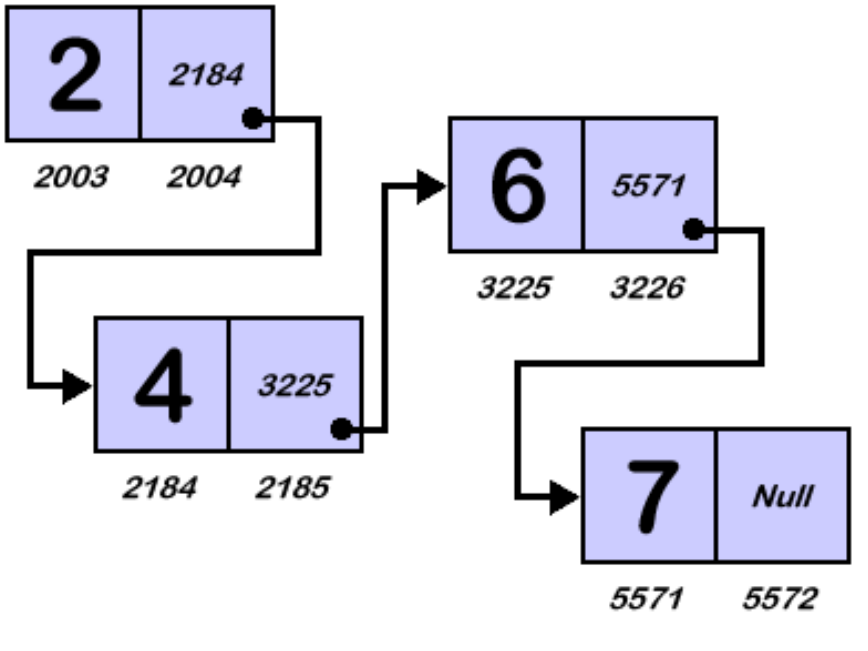
- **Danh sách liên kết đơn**




```
typedef struct list {
        DATA_TYPE item;      /* data item */
        struct list *pNext;  /* point to successor */
} LIST;
```

Danh sách liên kết đơn

- Một số thao tác thông dụng trên danh sách liên kết đơn
  - Chèn một phần tử mới
  - Xóa một phần tử
  - Tìm kiếm một phần tử



Danh sách liên kết đơn

- Chèn một phần tử mới vào đầu danh sách
  - **Đầu vào**: **pHead** con trỏ trỏ tới đầu danh sách,
    Value giá trị cần chèn vào
  - **Đầu ra** : danh sách thu được sau khi chèn thêm

```
void insert_list(LIST  *&l,   DATA_TYPE  x)
{
   LIST *p;                          /* temporary pointer */
   p = (LIST *)malloc( sizeof(LIST) );
   p->item = x;
   p->pNext = l;
   l = p;
}
```

Danh sách liên kết đơn

- Tìm kiếm một phần tử trong danh sách
  - **Đầu vào**: danh sách L và một khóa k
  - **Đầu ra**: phần tử trong L có giá trị khóa bằng khóa k, hoặc NULL nếu không tìm thấy.

```
LIST *search_list(LIST *l, DATA_TYPE  x)
{
        if (l == NULL) return(NULL);
        if (l->item == x)
                return(l);
        else
                return( search_list(l->pNext, x) );
}
```

- Thời gian thực hiện $O(n)$

Danh sách liên kết đơn

- Xóa phần tử khỏi danh sách
  - Đầu vào: Danh sách L và giá tị phần tử cần xóa
  - Đầu ra: Danh sách thu được sau khi xóa.

```
LIST *predecessor_list(LIST *l, DATA_TYPE x)
{
    if ((l == NULL) || (l->pNext == NULL)) {
        printf("Error: Danh sach rong hoac co 1 phan tu.\n");
        return(NULL);
    }
    if ((l->pNext)->item == x)
        return(l);
    else
        return( predecessor_list(l->pNext, x) );
}
```

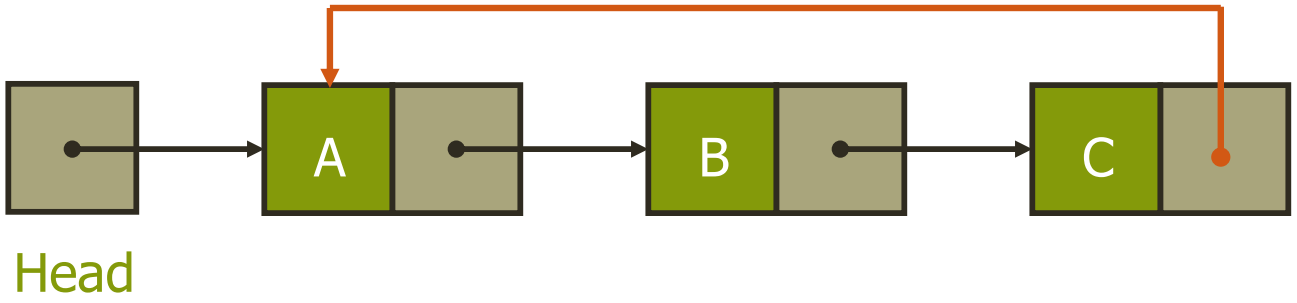
Danh sách liên kết đơn

```
void  delete_list(LIST  *&l, DATA_TYPE  x)
{
    LIST *p;          /* item pointer */
    LIST *pred;     /* predecessor pointer */
    p = search_list(l, x);
    if (p != NULL) {
            pred = predecessor_list(l, x);
            if (pred == NULL)              /* splice out out list */
            l = p->pNext;
        else
            pred->pNext = p->pNext;
        free(p);                    /* free memory used by node */
    }
}
```

Con trỏ và cấu trúc liên kết

Một số dạng khác của danh sách liên kết

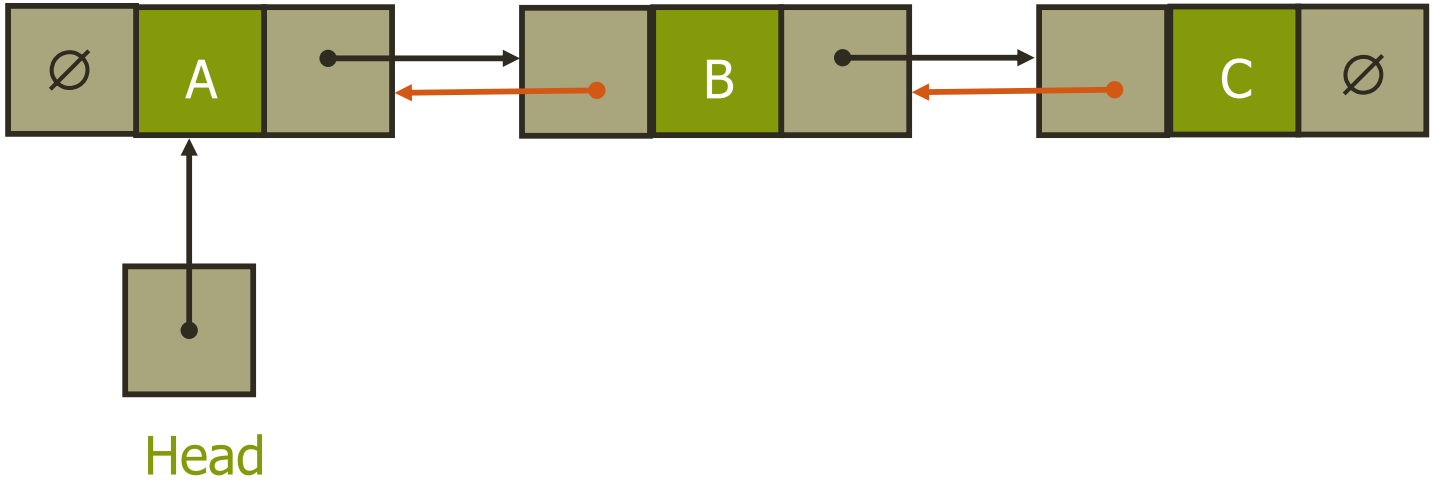
- **Danh sách liên kết đơn nối vòng**: Con trỏ của phần tử cuối trỏ về đầu danh sách



- Tác dụng: có thể quay lại từ đầu khi đã ở cuối dãy
- Kiểm tra ở đầu dãy : currentNode == Head ?

Con trỏ và cấu trúc liên kết

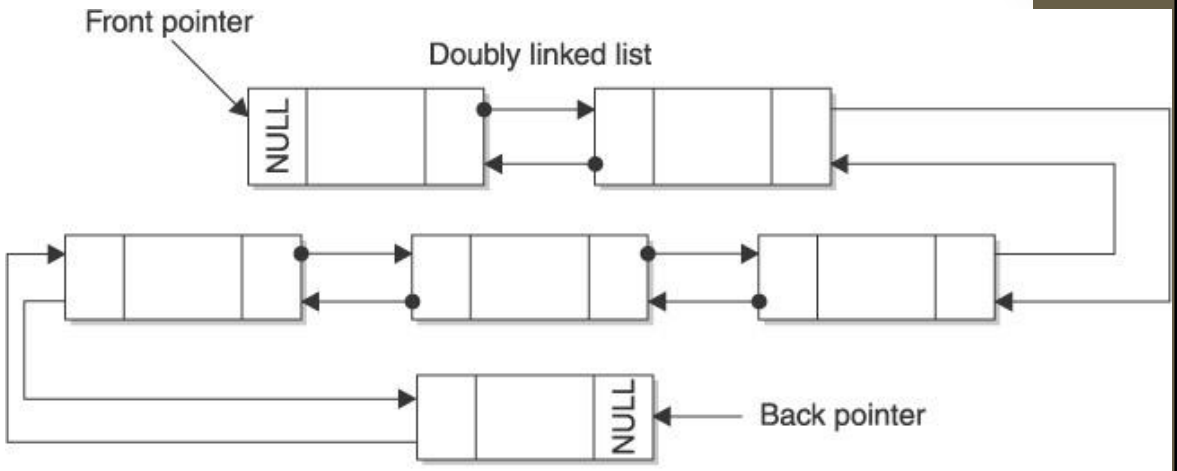
- **Danh sách liên kết đôi:**
  - Mỗi nút có 2 con trỏ: con trỏ phải trỏ đến phần tử tiếp sau, con trỏ trái trỏ đến phần tử ngay trước.

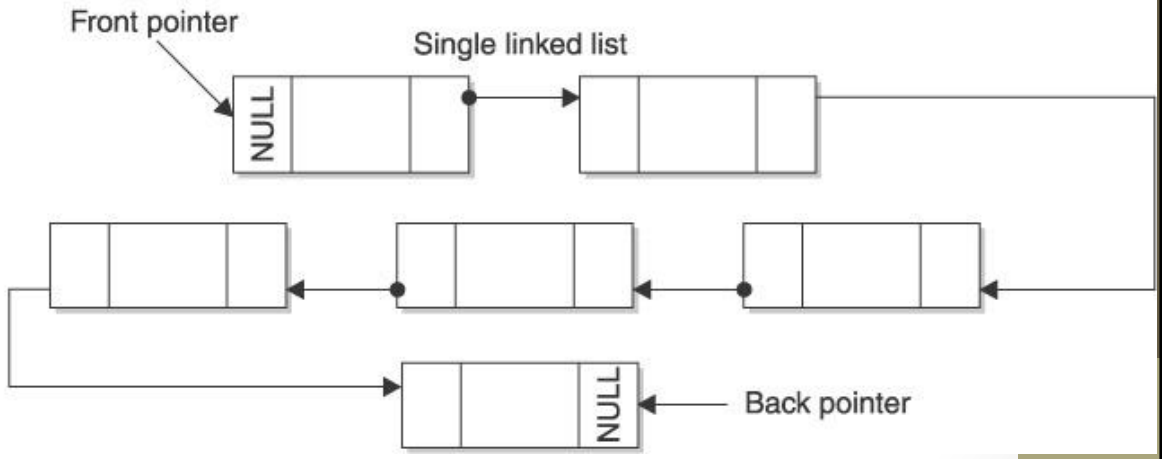


  - **Ưu điểm**: có thể duyệt danh sách theo cả hai chiều
  - Kiểm tra cuối danh sách: con trỏ phải là NULL
  - Đầu danh sách: con trỏ trái là NULL

Con trỏ và cấu trúc liên kết

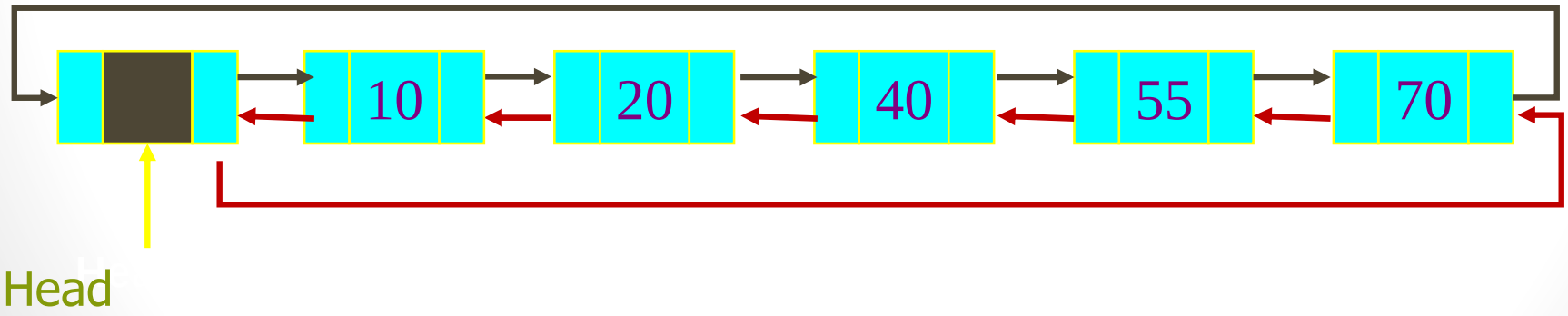
```
typedef struct list {
        DATA_TYPE item;
        struct list *pNext;
        struct list *pPrev;
} LIST;
```





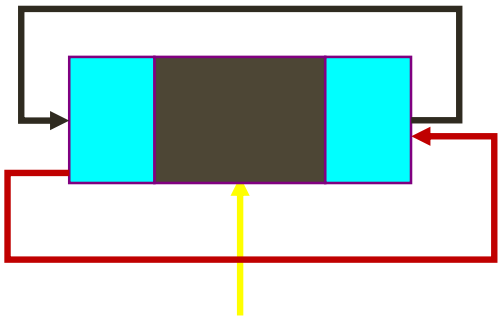
Con trỏ và cấu trúc liên kết

- **Danh sách liên kết đôi nối vòng** (danh sách liên kết đôi với nút giả):
  - Con trỏ pNext của nút cuối trỏ vào nút đầu và con trỏ pPrev của nút đầu trỏ vào nút cuối.
  - Nút đầu danh sách mà nút giả



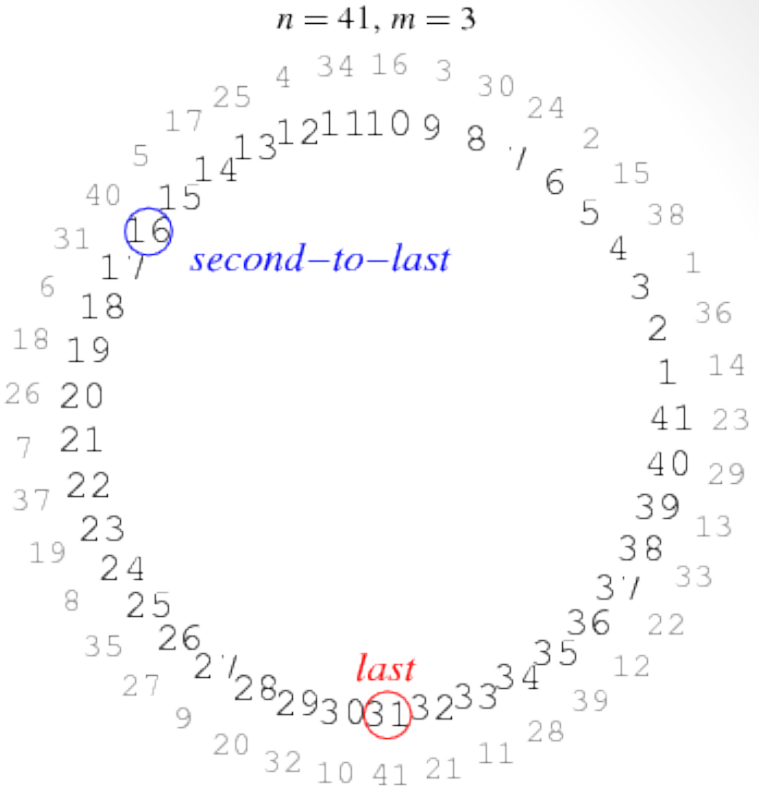
Danh sách liên kết đôi nối vòng

- **Ưu điểm**: có thể di chuyển theo hai chiều, và từ phần tử cuối (đầu) có thể nhảy ngay đến phần tử đầu (cuối) dãy.
- Kiểm tra danh sách rỗng: pNext, pPrev đều trỏ vào 1 phần tử Head.
- Kiểm tra phần tử cuối dãy: pNext trỏ tới Head



Danh sách rỗng

Ứng dụng



Ví dụ. Bài toán Josephus

- Có một nhóm gồm *n* người được xếp theo một vòng tròn. Từ một vị trí bất kỳ đếm theo chiều ngược chiều kim đồng hồ và loại ra người thứ *m* trong vòng. Sau mỗi lần loại lại bắt đầu đếm lại vào loại tiếp cho đến khi chỉ còn lại 1 người duy nhất.

- **Cài đặt** : sử dụng danh sách liên kết đơn nối vòng

MỘT SỐ CẤU TRÚC DỮ LIỆU CƠ BẢN

Danh sách tuyến tính

Ngăn xếp – Stack

Hàng đợi – Queue

…

CLIENT LIST

4 Wheeling America
4 Wheel Sport Utility
Acom Healthcare
American Ski
American Superconductor
Aptuit
Ardais
Bacou-Dalloz
Biogen Idec
Bioventures
B Fletcher & Associates
Bon-Ton
Bradford Soap Works
Browne & Ware Overland
CuraGen
Chomerics
STATS ChipPAC
CJ Higgins Engineering
Clarus Ventures
Cobra Golf
Corning
Courier
Danaher
Drohan, Huges & Toccio
Dynagraf Printing
EMC
Entegris
Epix Medical
Everett Design
Feinstein Kean Healthcare
First Act
First Marblehead
Genworth
Genzyme
The Gillette Co.
Hingham Recreation Dept.
Harvard Business School
JFK Library
Land Rover Lifestyle
Land Rover Owners Int.
Land Rover Hanover
Linear Air
Locke DMD
Mass Army National Guard
Merk
MicroEdge
Millipore
Milton Academy
MIT RLE
MIT Physics Dept.
Monosol RX
Morgan
MPM Capitol
Mykrolis
Nabi Biopharmaceuticals
National Cancer Institute
Netgear
New Bedford EMS
NMT Medical
Numerics
OmniGuide
Oscient Pharmaceuticals
Parexcel
Parthenon Group
Parthenon Capitol
Partners and Simons
Rattle the Market
Raytheon
RI Air National Guard
RSA
Savage Arms
Senko
Sentillion
Skadden, Arps, Slate...
TA Associates
Thales Raytheyon Systems
Therion Biologics
Titliest
TJX Companies
Urban Improv
US Shiping
Uvex
Vanerweil Engineers
Vertex
Weiu
Weymouth Design
White Builders

2.4 DANH SÁCH TUYẾN TÍNH

Danh sách tuyến tính

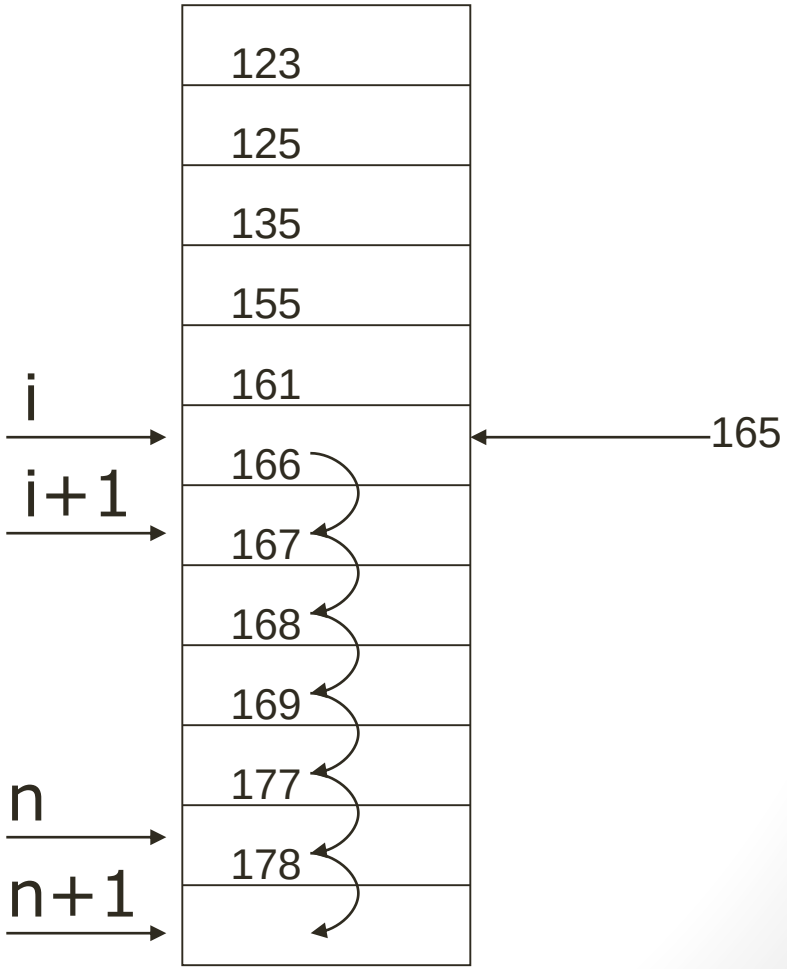
- **Danh sách tuyến tính** (linear list) là tập hợp các đối tượng có cùng kiểu, được gọi là các phần tử. Các phần tử trong danh sách tuân theo thứ tự tuyến tính.
  - VD. Danh sách các sinh viên sắp theo thứ tự tên
- Các phép toán cơ bản
  - Chèn thêm phần tử
  - Xóa phần tử
  - Tìm kiếm
  - Kiểm tra rỗng
  - …
- Cài đặt danh sách tuyến tính
  - Dùng mảng
  - Cấu trúc liên kết

Danh sách tuyến tính

- **Cài đặt dùng mảng**: sử dụng mảng 1 chiều
  - Tìm kiếm dễ dàng (tuần tự hoặc tìm kiếm nhị phân)
  - Duyệt các phần tử dễ dàng sử dụng chỉ số:
  - Chèn và xóa KHÔNG dễ dàng
  - Số lượng phần tử biến động => Phải khai báo số lượng phẩn tử tối đa hoặc dùng mảng động

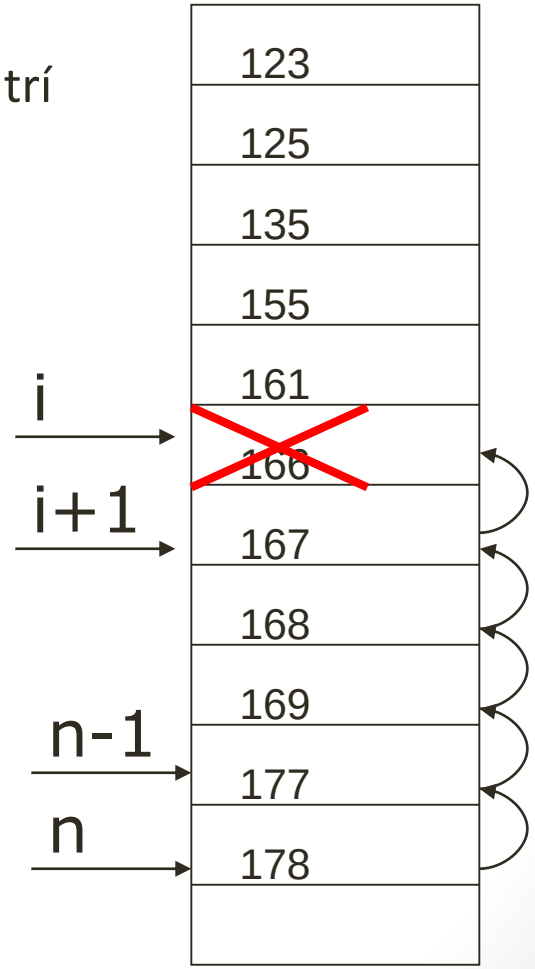
Danh sách tuyến tính

- Chèn thêm phần tử vào danh sách:
  - Dịch chuyển các phần tử đứng sau từ vị trí cần thêm xuống 1 vị trí
  - Thêm phần tử mới vào
  - Tăng số lượng phần tử hiện tại thêm 1
- Trung bình cần $n/2$ dịch chuyển mỗi khi thêm
- Thời gian thực hiện $O(n)$



Danh sách tuyến tính

- Xóa phần tử khỏi danh sách:
  - Chuyển các phần tử đứng sau lên trước 1 vị trí
  - Giảm số lượng phần tử hiện tại đi 1
- Trung bình cần $n/2$ dịch chuyển mỗi khi xóa 1 phần tử
- Thời gian thực hiện $O(n)$



Danh sách tuyến tính

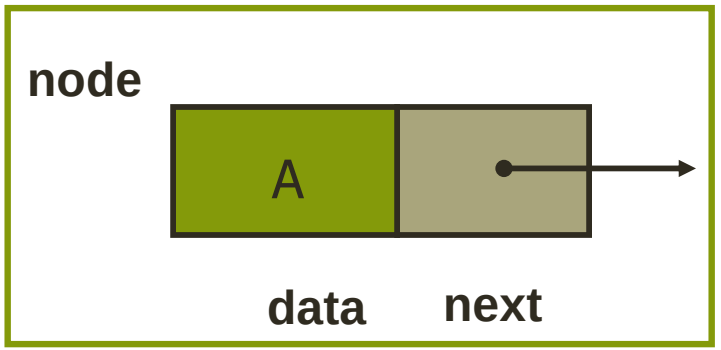
- **Cài đặt danh sách tuyến tính dùng mảng:**
- **Ưu điểm**:
  - Thời gian truy cập từng phần tử nhanh $O(1)$
  - Tìm kiếm phần tử nhanh (tìm kiếm nhị phân)

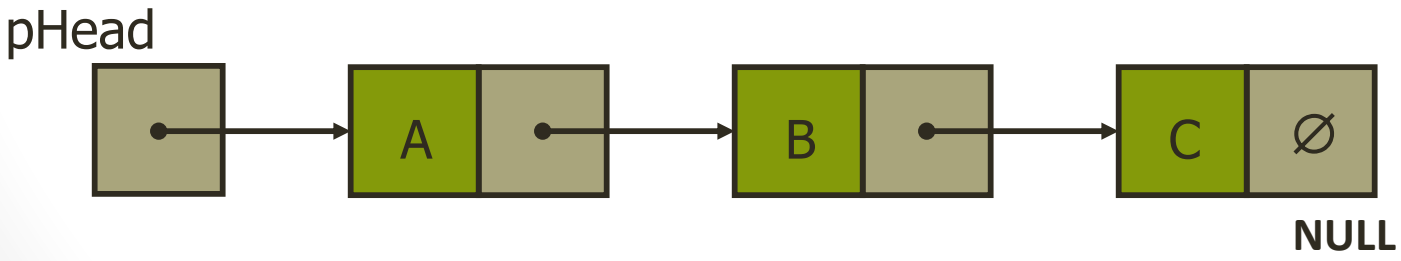
- **Nhược điểm:**
  - Chèn và xóa phần tử mất nhiều thời gian (trung bình là $O(n)$)
  - Cần phải biết trước số lượng phần tử tối đa của danh sách hoặc sẽ lãng phí bộ nhớ cho các phần tử chưa được dùng đến trong mảng

Danh sách tuyến tính

- **Cài đặt dùng cấu trúc liên kết**: danh sách liên kết đơn (singly linked list)

```
typedef struct list {
        DATA_TYPE   item;
        struct list   *pNext;
} LIST;
```





Cài đặt dùng danh sách liên kết

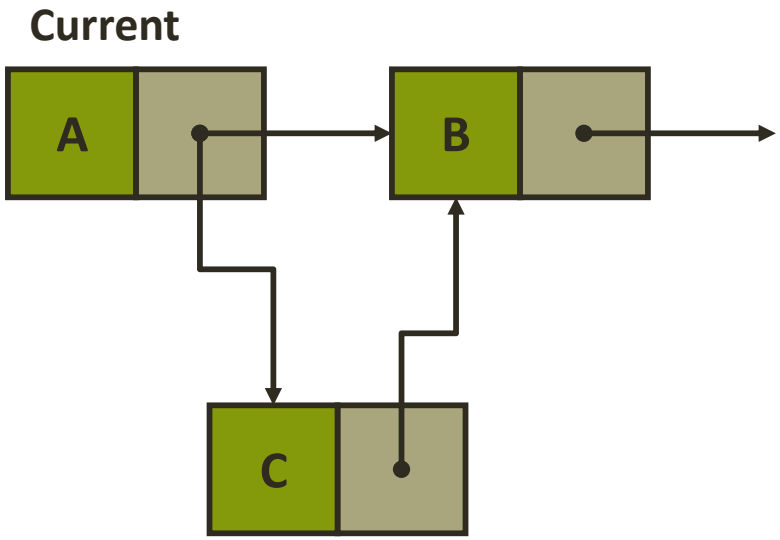
- **Tìm kiếm**: tìm kiếm phần tử x trong danh sách.
  Phương pháp: duyệt lần lượt từng phần tử trong danh sách và so sánh với x. Thực hiện bằng vòng lặp hoặc đệ quy.

```
LIST *search_list(LIST *l, DATA_TYPE  x)
{
        if (l == NULL) return(NULL);
        if (l->item == x)
                return(l);
        else
                return( search_list(l->next, x) );
}
```

- Giống tìm kiếm trên danh sách liên kết đơn

Cài đặt dùng danh sách liên kết

- **Chèn vào giữa**: thêm một phần tử mới vào giữa danh sách



1. Cấp phát bộ nhớ để lưu trữ phần tử mới
2. Cho con trỏ của phần tử mới trỏ vào phần tử sau
3. Cho con trỏ của phần tử hiện tại trỏ vào phần tử mới

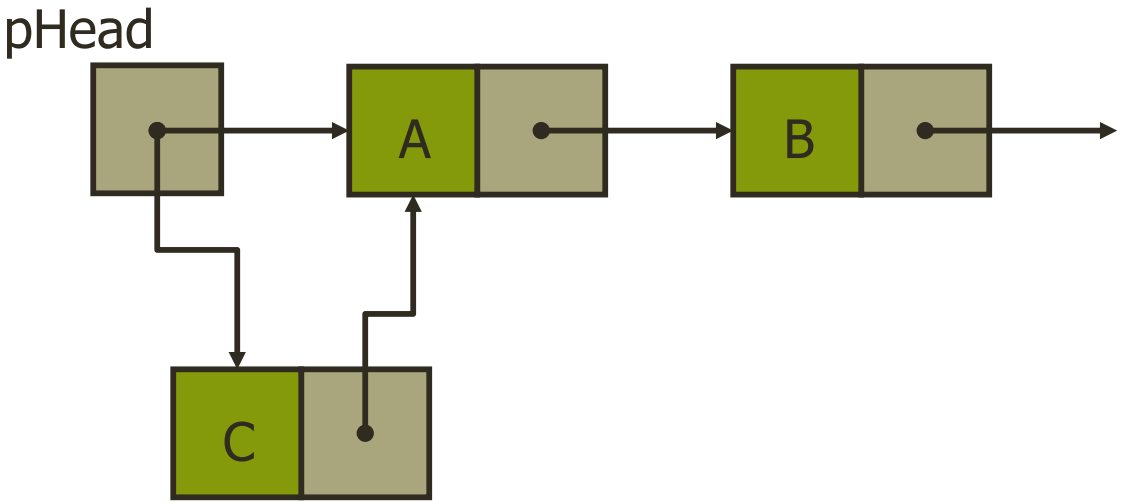
Cài đặt dùng danh sách liên kết

```
void insert_list(LIST  *&Current,   DATA_TYPE  x)
{
   LIST *p;                    /* temporary pointer */
   p = (LIST*)malloc( sizeof(LIST) );
   p->item = x;
   p->pNext = Current->pNext;
   Current->pNext = p;
}
```

Cài đặt dùng danh sách liên kết

- **Chèn vào đầu danh sách**

insert_list (Head, x)

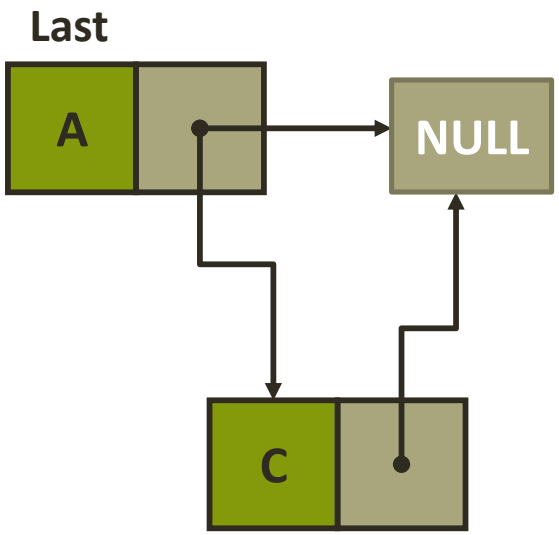


1. Cấp phát bộ nhớ để lưu trữ phần tử mới
2. Cho con trỏ của phần tử mới trỏ vào phần tử đầu
3. Cho pHead trỏ vào phần tử mới

Cài đặt dùng danh sách liên kết

- **Chèn vào cuối danh sách**

insert_list (Last, x)



1. Cấp phát bộ nhớ để lưu trữ phần tử mới
2. Cho con trỏ của phần tử mới trỏ vào NULL
3. Cho con trỏ của phần tử cuối trỏ vào phần tử mới

Cài đặt dùng danh sách liên kết

- **Xóa**: xóa phần tử khỏi danh sách.
- Phương pháp giống với xóa phần tử trong danh sách liên kết đơn.

- **Kiểm tra danh sách rỗng:** kiểm tra xem danh sách có chứa phần tử nào hay không.

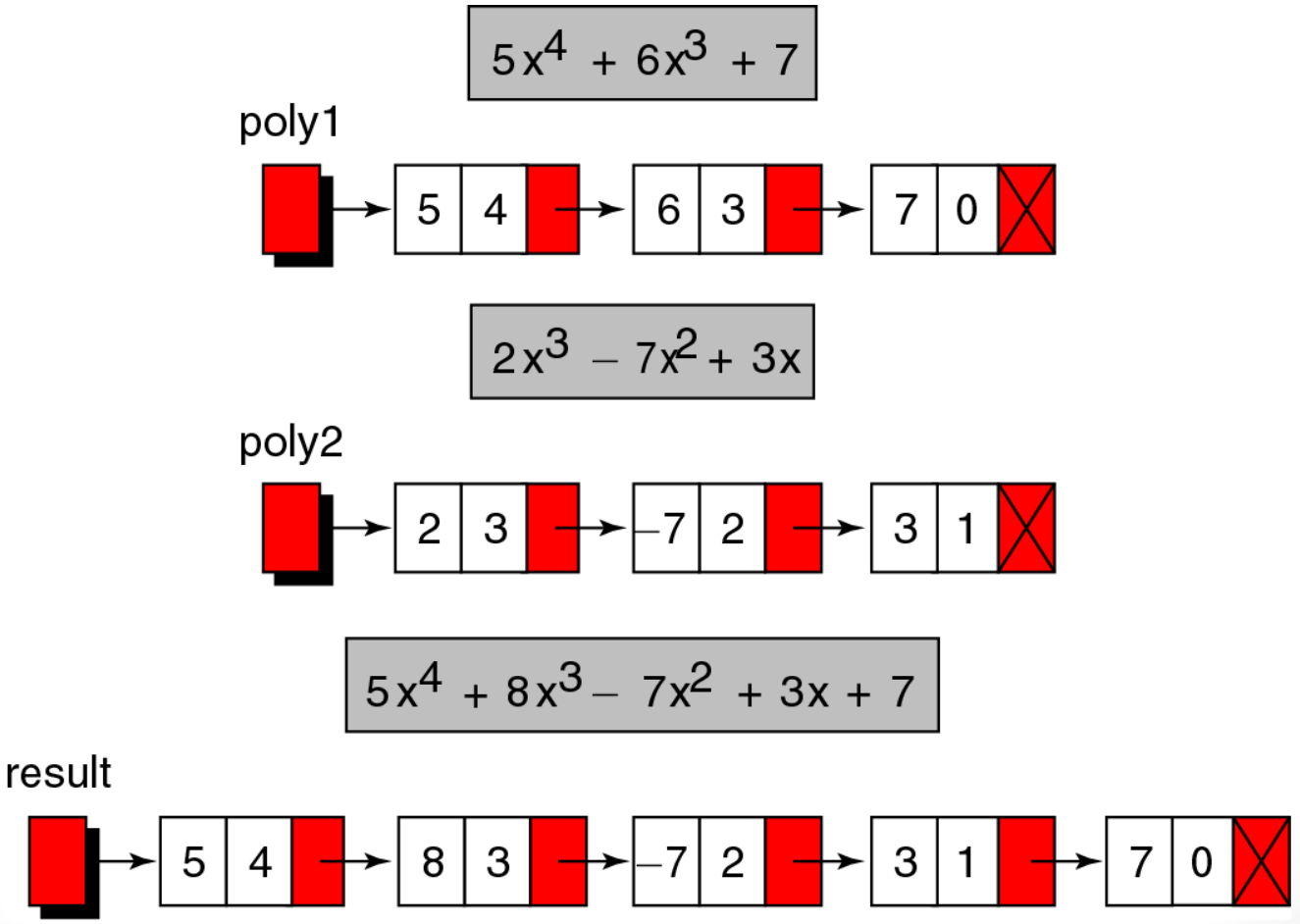
```
bool isEmpty(LIST *Head)
{
    if(Head==NULL) return true;
    return false;
}
```

Cài đặt dùng danh sách liên kết

- **Cài đặt danh sách tuyến tính dùng danh sách liên kết:**
- **Ưu điểm**:
  - Chèn và xóa nhanh do chỉ cần thao tác với một vài con trỏ
  - Không cần biết trước số lượng phần tử của danh sách, khi cần lưu trữ phần tử mới cấp phát bộ nhớ (danh sách chỉ đầy khi bộ nhớ trên máy hết)
- **Nhược điểm:**
  - Không cho phép truy nhập trực tiếp từng phần tử
  - Theo tác tìm kiếm mất nhiều thời gian (cỡ $O(n)$)
  - Cần bộ nhớ để lưu thêm các con trỏ

Ứng dụng

- VD1. Biểu diễn đa thức



Ứng dụng

- **typedef struct poly{**
  float heSo;
  float soMu;
  struct poly *nextNode;
  } POLY;

- **Các thao tác:**
  - Nhập đa thức
  - Hiển thị
  - Cộng
  - Trừ
  - Nhân
  - Tính giá trị đa thức
  - Chia
  - ….