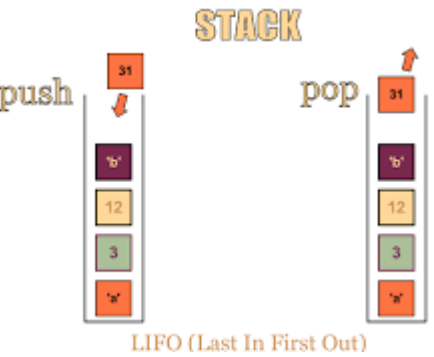
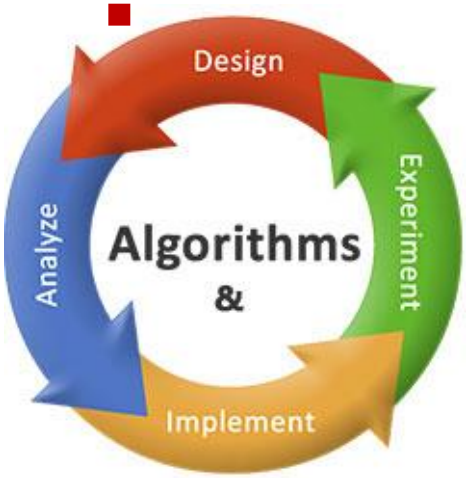
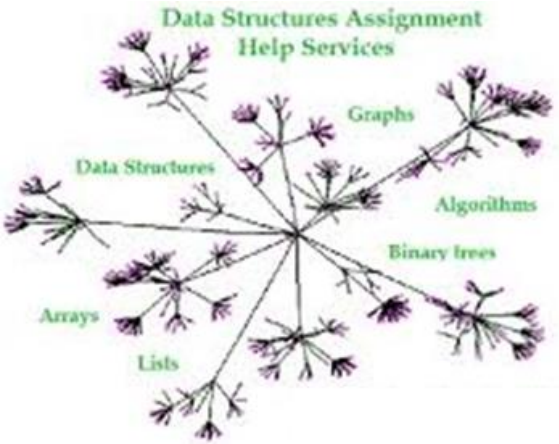


CẤU TRÚC DỮ LIỆU

&

GIẢI THUẬT





Lê Văn Hạnh

levanhanhvn@gmail.com

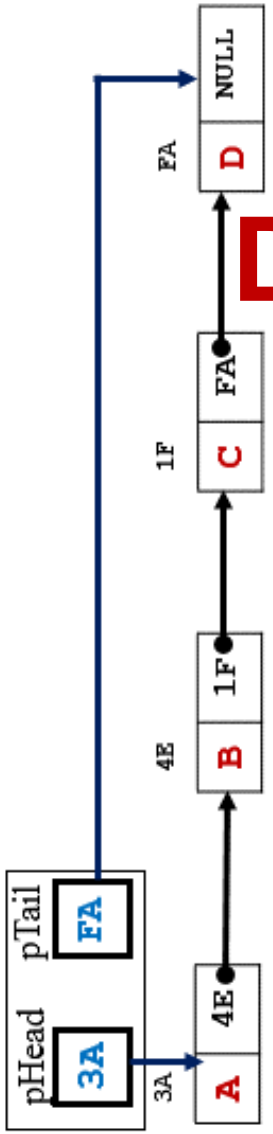
NỘI DUNG MÔN HỌC

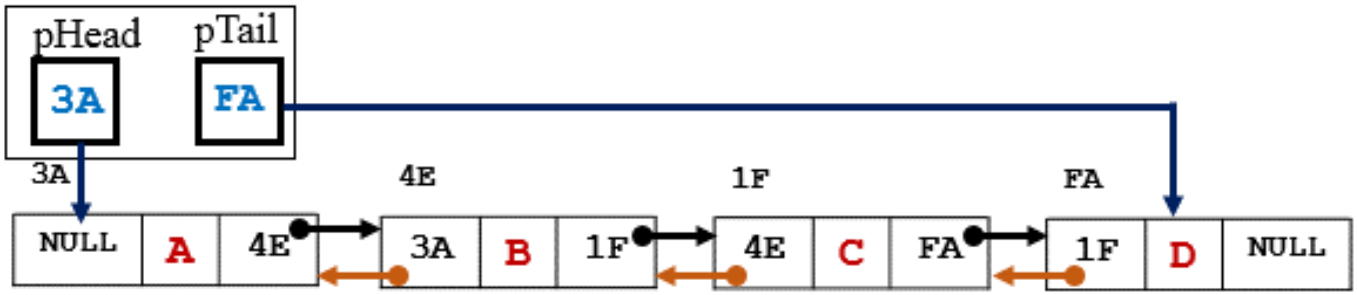
- Chương 1: Ôn tập ngôn ngữ lập trình C
- Chương 2: Kiểu dữ liệu con trỏ
- Chương 3: Tổng quan về cấu trúc dữ liệu và giải thuật
- Chương 4: Danh sách kề (Danh sách tuyến tính)
- Chương 5: Các giải thuật tìm kiếm trên danh sách kề
- Chương 6: Các giải thuật sắp xếp trên danh sách kề
- **Chương 7: Danh sách liên kết động (Linked List)**
- Chương 8: Ngăn xếp (Stack)
- Chương 9: Hàng đợi (Queue)
- Chương 10: Cây nhị phân tìm kiếm (Binary Search Tree)
- Chương 11: Cây cân bằng
- Chương 12: Bảng băm (Hash Table)

Chương 7

DANH SÁCH LIÊN KẾT ĐỘNG

(Dynamically Linked List)





MỤC TIÊU

- Nắm vững khái niệm về kiểu dữ liệu tĩnh và động
- Nắm vững cách tổ chức dữ liệu động bằng danh sách liên kết và minh họa được các thao tác xử lý trên danh sách liên kết đơn
- Cài đặt minh họa được các thao tác của danh sách đơn bằng ngôn ngữ C/ C++

NỘI DUNG

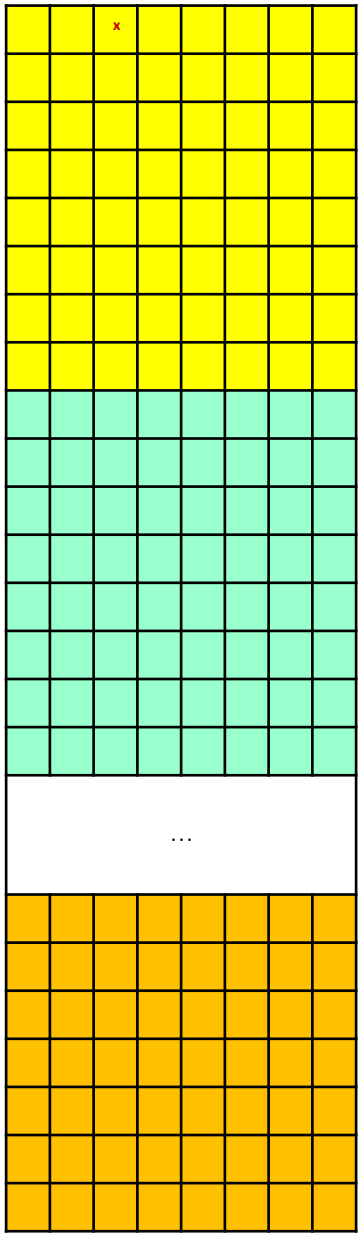
1. Giới thiệu
2. Danh sách liên kết (DSLK)
3. Lập trình với danh sách liên kết đơn
4. Danh sách liên kết không đơn
   i. DSLK đôi
   ii. DSLK đơn vòng
   iii. DSLK đôi vòng
5. Các kiểu phối hợp khác dựa trên DSLK
   i. Mảng các DSLK
   ii. DSLK với nội dung mỗi node chứa 1 DSLK khác

(SV tự cài đặt và trình bày phần 4 và 5)

1. GIỚI THIỆU

1.1. Bộ nhớ chính

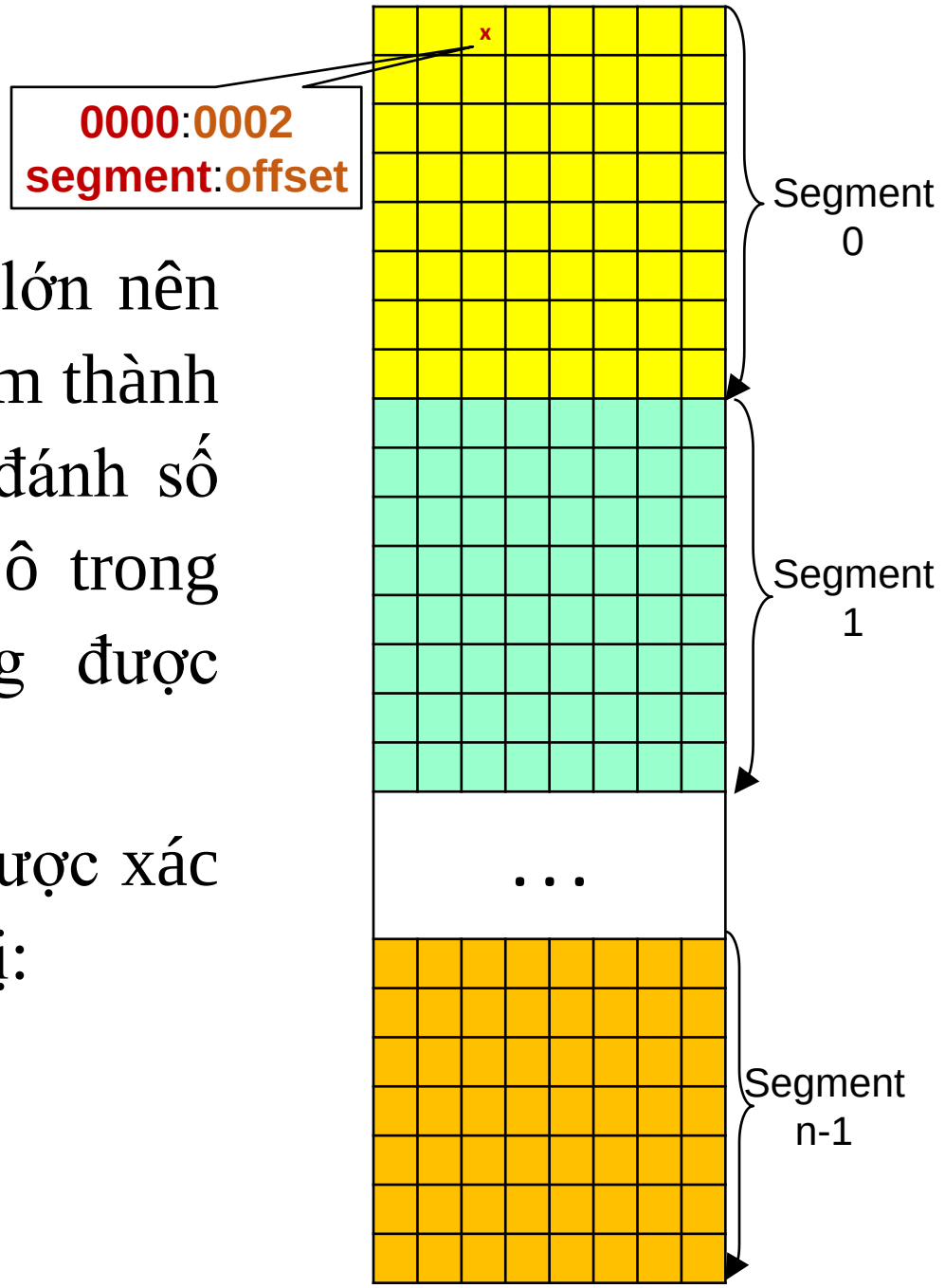
- *Bộ nhớ chính* là thiết bị lưu trữ duy nhất để thông qua đó CPU có thể trao đổi thông tin với môi trường bên ngoài.
- Bộ nhớ chính được tổ chức như một mảng 1 chiều các từ nhớ (word), mỗi từ nhớ có 1 địa chỉ



1.1. Bộ nhớ chính

- Do số lượng ô nhớ quá lớn nên nhiều ô nhớ sẽ được nhóm thành 1 đoạn (*segment*) được đánh số thứ tự từ 0 trở đi. Các ô trong mỗi đoạn (*offset*) cũng được đánh số từ 0 trở đi.
- Vậy địa chỉ mỗi ô nhớ được xác định thông qua cặp giá trị:

segment: offset



1.2. Biến tĩnh (Static variable) và vùng nhớ

- ***Khi khai báo biến***: máy tính sẽ **dành riêng một vùng nhớ** để lưu biến đó. Thông tin máy tính cần lưu trữ về biến gồm:
  - ***Tên biến***
  - ***Kiểu dữ liệu*** của biến
  - ***Giá trị*** của biến
  - ***Địa chỉ vùng nhớ*** nơi chứa giá trị của biến: nếu kích thước của biến gồm nhiều byte thì máy tính sẽ cấp phát một dãy các byte liên tiếp nhau, địa chỉ của biến sẽ là địa chỉ byte đầu tiên trong dãy các byte này.
- ***Khi tên biến được gọi***: máy tính sẽ thực hiện 2 bước sau:
  - *B1*: Tìm kiếm địa chỉ ô nhớ của biến.
  - *B2*: Truy xuất hoặc thiết lập giá trị của biến được lưu trữ tại ô nhớ đó.

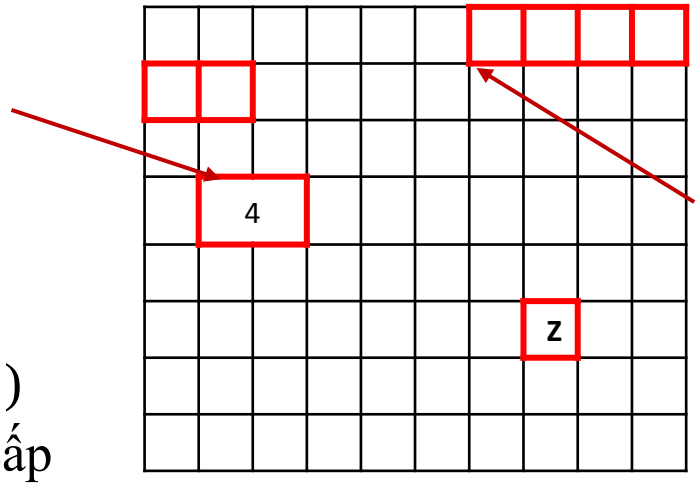
1.2. Biến tĩnh (Static variable) và vùng nhớ

– VD:

```
void main()
{   int x=4;
    char a[6];
    char c='Z';
    …
}
```

Sau khi khai báo biến x, địa chỉ ô nhớ hệ điều hành cấp cho biến **x** là: **A2B9:417C**

Giá trị của biến **x** (=**4**) được lưu tại địa chỉ đã cấp cho biến x (**A2B9:417C**)



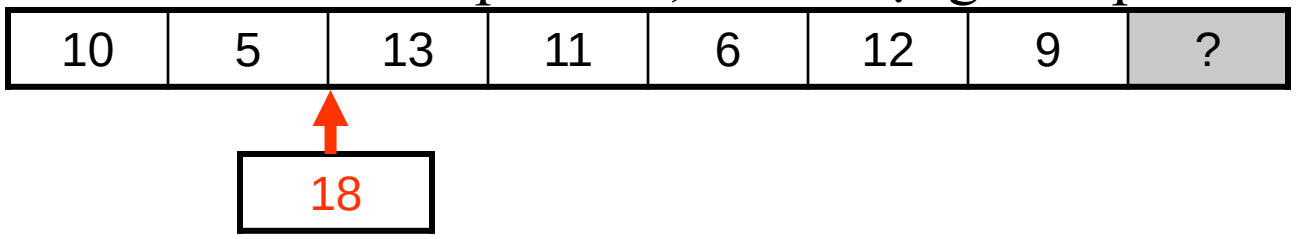
Địa chỉ ô nhớ hệ điều hành cấp cho mảng **a** là: **A2B9:3F8E**

Memory Layout

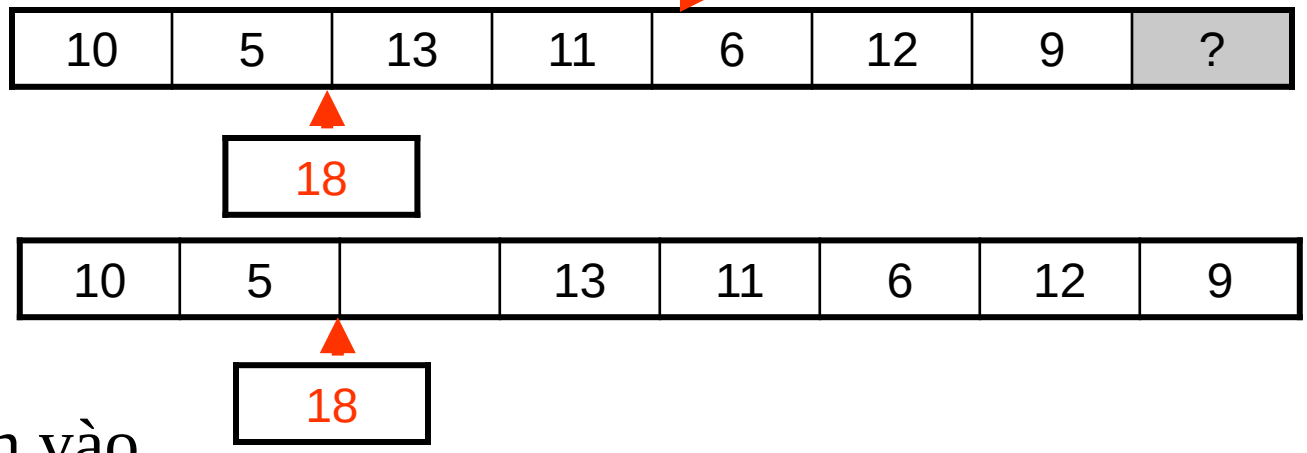
1.3. Một số hạn chế của biến tĩnh

1.3.1. Làm sao để thêm một phần tử vào mảng?

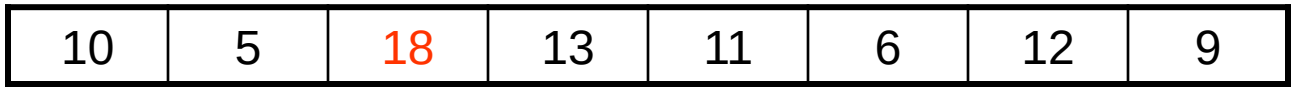
Cho 1 DS kề có SIZE=8 phần tử, đã sử dụng n=7 phần tử.



– Phải chuyển các phần tử về phía sau một vị trí…



– Và thêm vào



– Làm sao thêm một phần tử nữa?

1.3. Một số hạn chế của biến tĩnh

1.3.1. Làm sao để thêm một phần tử vào mảng?

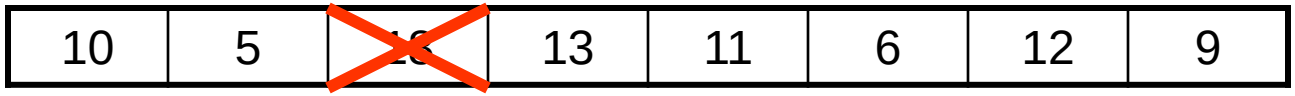
Bài tập: Cho DS kề ***a*** chứa các node có kiểu là số nguyên kích thước ***n***. Hãy cài đặt hàm chèn một phần tử có giá trị ***x*** vào vị trí ***vt*** trong DS kề theo nguyên mẫu hàm như sau:

```
void ChenXVaoViTri(int a[], int &n, int x, int vt)
```

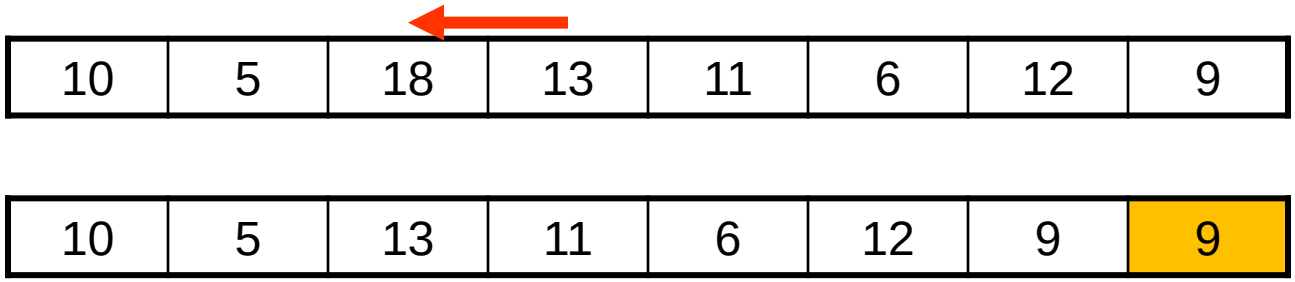
```
void ChenXVaoViTri  (int A[], int &n, int X, int vitri)
{       /* B1: Xuất phát từ cuối mảng tiến hành đẩy lần lượt các phần tử
                                        về phía sau cho đến vị trí cần chèn*/
        for (int i = n; i >vitri ; i--)
                A[i] = A[i-1] ;
        // B2: Đưa phần tử cần chèn vào vị trí chèn
        A[vitri] = X;
        // B3: Tăng kích thước mảng sau khi thêm 1 phần tử
        n++;
}
```

1.3. Một số hạn chế của biến tĩnh

1.3.2. Làm sao xóa một phần tử trong DS kề



- Phải dời các phần tử về trước một vị trí



- Độ phức tạp của thêm và xóa là **O(n)**

1.3.2. Làm sao xóa một phần tử trong DS kề

Bài tập: Cho DS kề ***a*** chứa các node có kiểu là số nguyên kích thước ***n***. Hãy cài đặt hàm xóa một phần tử tại vị trí ***vt*** theo mẫu hàm như sau: void XoaTaiViTri(int a[], int &n, int x)

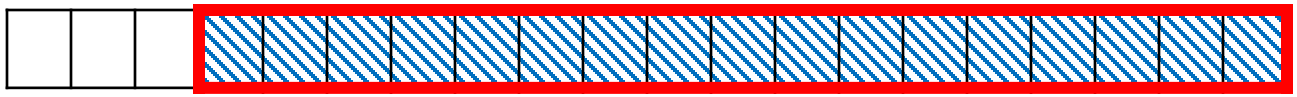
```
void XoaTaiViTri (int A[], int &n, int vitri)
{
        // B1: Dời sang trái từ vitri đến n-1
        for (int i = vitri; i < n ; i++)
                A[i] = A[i+1];

        // B2: Giảm số lượng phần tử của mảng sau khi đã xóa
        n--;
}
```

1.5. Một số hạn chế của biến tĩnh

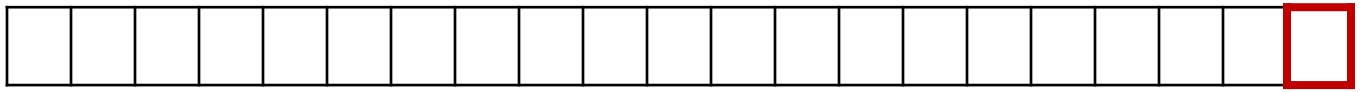
- Cấp phát ô nhớ dư, gây ra lãng phí ô nhớ.

  VD: `char a[20]; int n=3;`



- Cấp phát ô nhớ thiếu, chương trình thực thi bị lỗi.

  VD:
```
char a[20]; int n=3;
a[20]=5;// báolỗi
```



- Các biến tĩnh sẽ tồn tại trong suốt thời gian thực thi chương trình (dù biến chỉ sử dụng 1 lần rồi bỏ)

1.5. Một số hạn chế của biến tĩnh

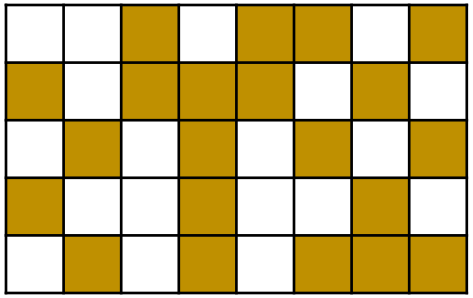
i. Độ phức tạp của chèn/ xóa trên mảng 1 chiều là O(n)

ii. Các vấn đề cần giải quyết:

- Phức tạp khi chèn/ xóa?
- Giới hạn kích thước vùng nhớ tối đa?

```
int A[10]; nA=11;
int B[1000000000], nB=3;
```

- Vùng nhớ không liên tục?



⇒ Có cần một cấu trúc lưu trữ khác?

sử dụng

CẤU TRÚC DỮ LIỆU ĐỘNG

1.6. Biến tĩnh và biến động trong ngôn ngữ C

	Biến tĩnh	Biến động
Khai báo	<kiểu dữ liệu> tên biến;	<kiểu dữ liệu> *tên biến;
VD	**int a; float y; char s[20];**	**int *a; float *y**
Cấp phát	Khi khai báo	Khi cần dùng: **new int [kích thước]**
VD		**int *a;** **a=new int [10];**
Thu hồi	Khi kết thúc hàm	Do người lập trình
VD		**delete a;**
Đặc điểm	✓ Tồn tại trong phạm vi khai báo ✓ Được cấp phát vùng nhớ trong vùng dữ liệu ✓ Kích thước cố định	✓ Chứa địa chỉ của một đối tượng dữ liệu ✓ Được cấp phát hoặc giải phóng bộ nhớ tùy thuộc vào người lập trình ✓ Kích thước có thể thay đổi

2. Danh sách liên kết

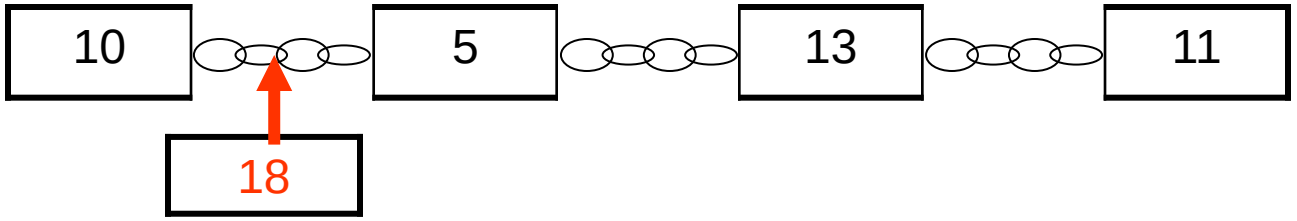
- Các phần tử đều độc lập (rời nhau)

- Được liên kết bằng các "dây xích"

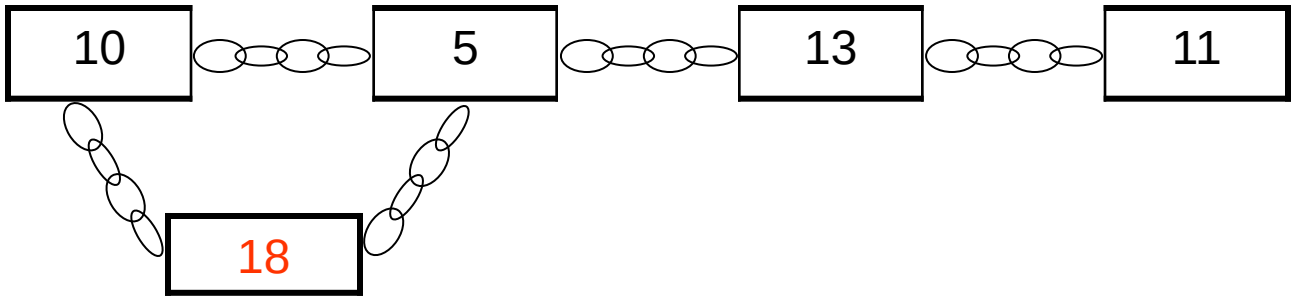
- Các phần tử không cần phải lưu trữ liên tiếp nhau trong bộ nhớ

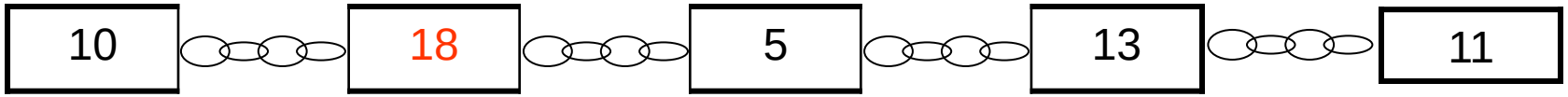
- Có thể mở rộng tuỳ ý (chỉ giới hạn bởi dung lượng bộ nhớ)

2.1. Chèn một phần tử vào danh sách



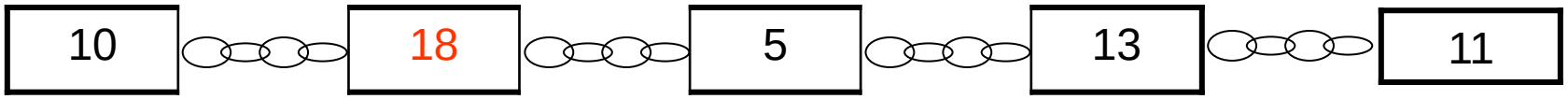
- Chỉ cần "móc" lại các dây xích khi thêm



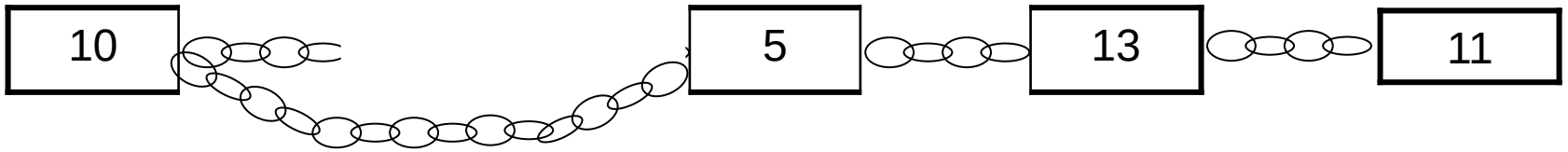


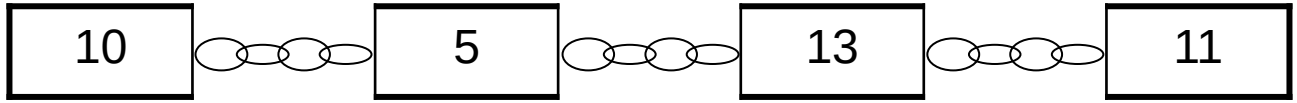
⇒ Thao tác ***chèn*** không cần phải dịch chuyển phần tử mà chỉ cần thay đổi mối liên kết

2.2. Xóa một phần tử khỏi danh sách



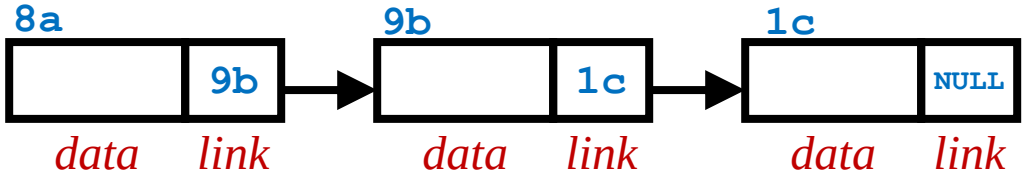
- "Móc" lại các dây xích khi xóa



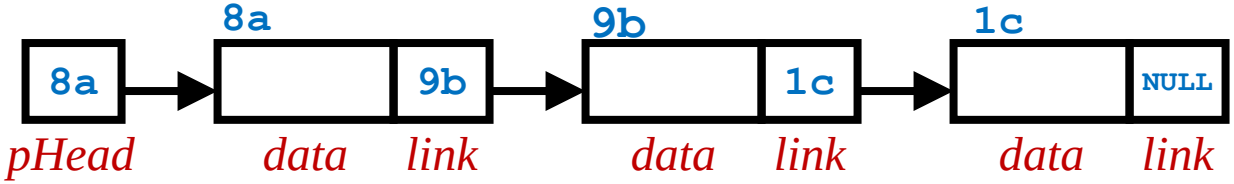


$\Rightarrow$ Thao tác ***xóa*** không cần phải dịch chuyển phần tử mà chỉ cần thay đổi mối liên kết

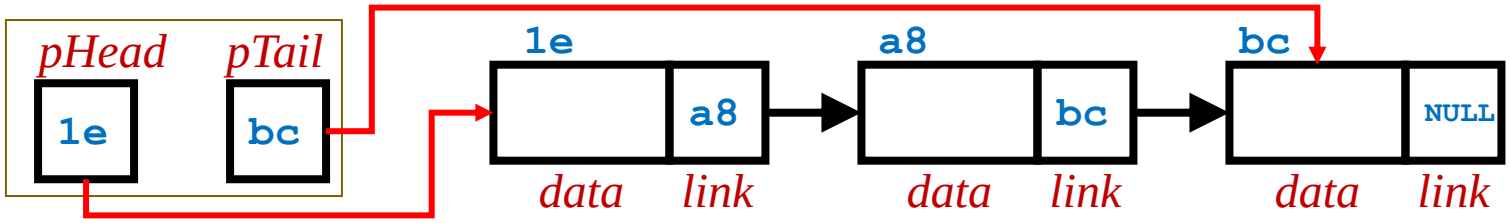
2.3. Khái niệm về danh sách liên kết (DSLK)



- Một dãy tuần tự các node.
- Mỗi node có 2 thông tin:
  - Dữ liệu (*data*).
  - Con trỏ liên kết đến phần tử kế tiếp trong danh sách (*Next pointer link*) ⇒ Phần tử cuối cùng trong danh sách có *Next pointer link* = ***NULL***
- Quản lý phần tử đầu tiên của DSLK bằng con trỏ pHead. pHead không phải là 1 node, mà chỉ là "*con trỏ chỉ đến node*".

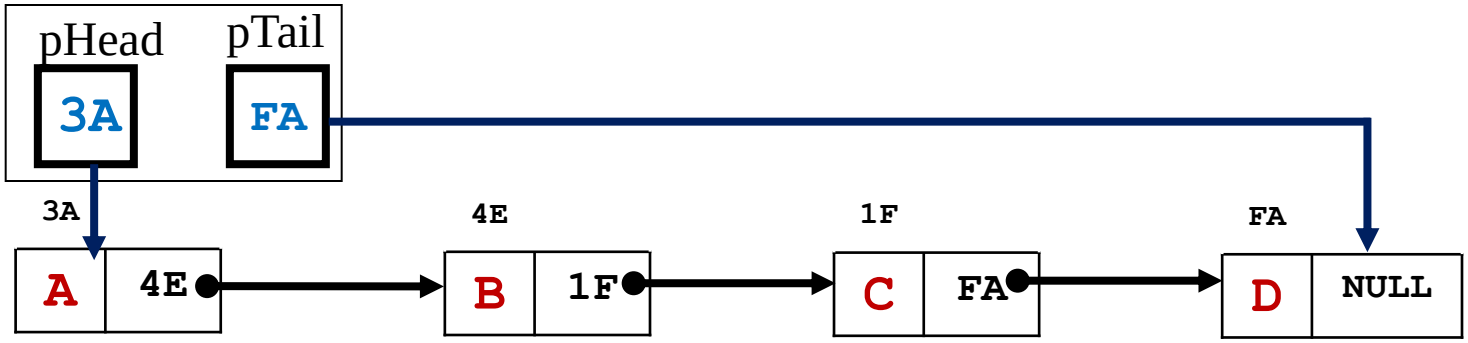


- Có thể sử dụng thêm con trỏ cuối (*pTail*). Tương tự, *pTail* cũng không phải là 1 node

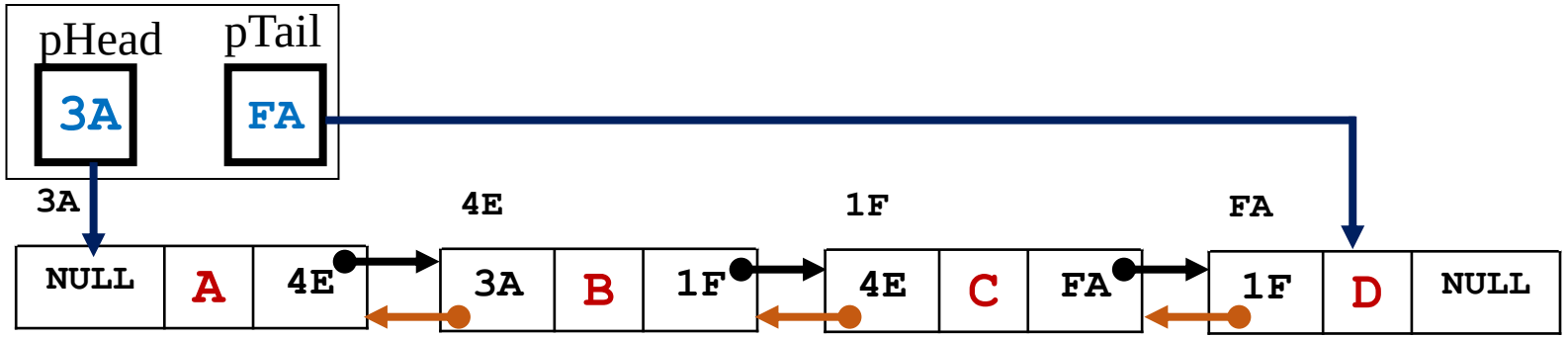


2.4. Các loại danh sách liên kết

- ***Danh sách liên kết đơn*** *(Singly Linked List)*: Mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách

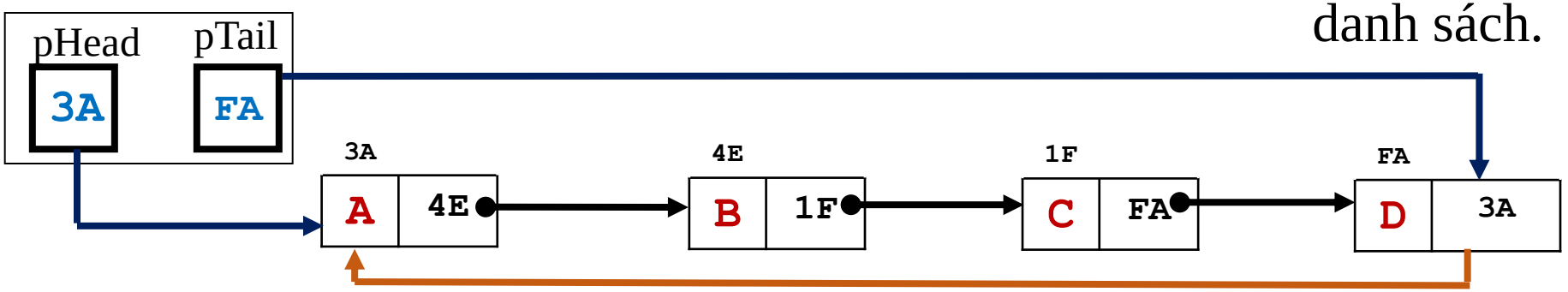


– ***Danh sách liên kết đôi*** *(Doubly-Linked List)*: Mỗi phần tử liên kết với phần tử đứng trước và sau nó trong danh sách



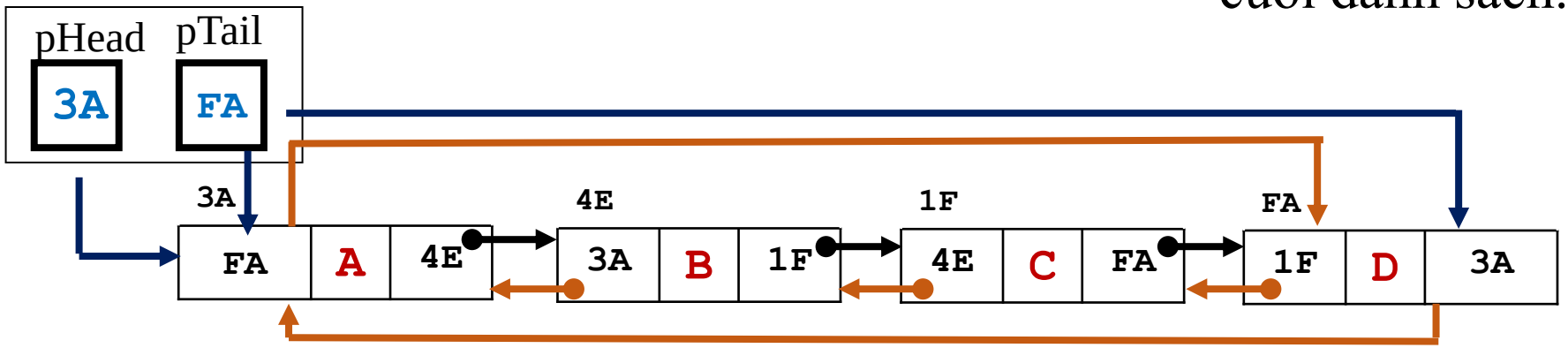
2.4. Các loại danh sách liên kết

– ***Danh sách liên kết đơn vòng (Circular-Linked List)***: Con trỏ pNext của phần tử cuối danh sách liên kết với phần tử đầu danh sách.



– ***Danh sách liên kết đôi vòng (Circular-Double Linked List):***

- Con trỏ *pNext* của phần tử cuối danh sách liên kết với phần tử đầu danh sách.
- Con trỏ *Prev* của phần tử đầu danh sách liên kết với phần tử cuối danh sách.



2.5. So sánh Mảng và Danh sách liên kết

Mảng	Danh sách liên kết
Kích thước cố định	Số phần tử thay đổi tùy ý
Các phần tử lưu trữ tuần tự (địa chỉ tăng dần) trong bộ nhớ	Các phần tử liên kết với nhau bằng con trỏ
Phải dịch chuyển các phần tử khi Thêm/Xóa	Chỉ cần thay đổi con trỏ liên kết khi Thêm/Xóa
Truy xuất ngẫu nhiên	Truy xuất tuần tự

2.6. Ưu điểm

- Các node không cần phải lưu trữ liên tiếp nhau trong bộ nhớ.
- Có thể mở rộng tuỳ ý (chỉ giới hạn bởi dung lượng bộ nhớ).
- Thao tác chèn/xóa không cần phải dịch chuyển các phần tử khác.
- Có thể truy xuất đến các phần tử khác thông qua con trỏ liên kết.

2.7. Cấu tạo của nút (node)

- Node có 1 field dữ liệu


```
typedef struct tagNODE
{       int number;
        tagNODE *pNext;
} NODE;
```

- Node có nhiều field dữ liệu


```
typedef struct tagNODE
{
        char  name[30];
        int   id;
        float grdPts;
        tagNODE *pNext;
} NODE;
```

2.7. Cấu tạo của nút (node)

- Node có dữ liệu là 1 struct gồm nhiều field

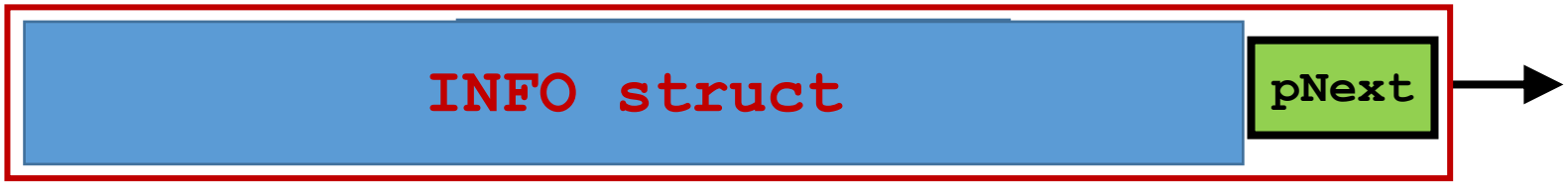
```
typedef struct tagINFO
{
        char        name[30];
        int         id;
        float       grdPts;
} INFO;

typedef struct tagNODE
{
        INFO        data;
        tagNODE*    pNext;
} NODE;
```

2.7. Cấu tạo của nút (node)

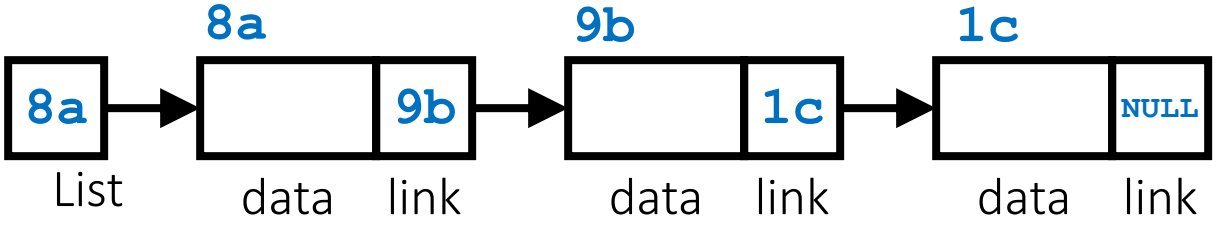
- Node có dữ liệu là 1 struct, trong đó lại có thành phần là 1 struct khác

```
typedef struct DATE
{       int ngay, thang, nam;
};
typedef struct tagINFO
{       char   name[30];
        int    id;
        DATE   NgaySinh
        float  grdPts;
} INFO;
typedef struct tagNODE
{
        INFO           data;
        tagNODE*       pNext;
} NODE;
```



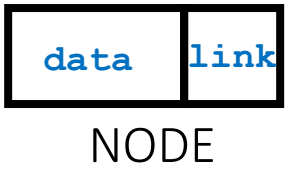
2.8. Cấu tạo của danh sách liên kết

2.8.1. Quản lý DSLK đơn với phần tử đầu là pHead



– Định nghĩa cấu trúc Node

```
typedef struct tagNode
{    data      data;
     tagNode   *pNext;
}NODE;
typedef NODE* NodePointer;
```

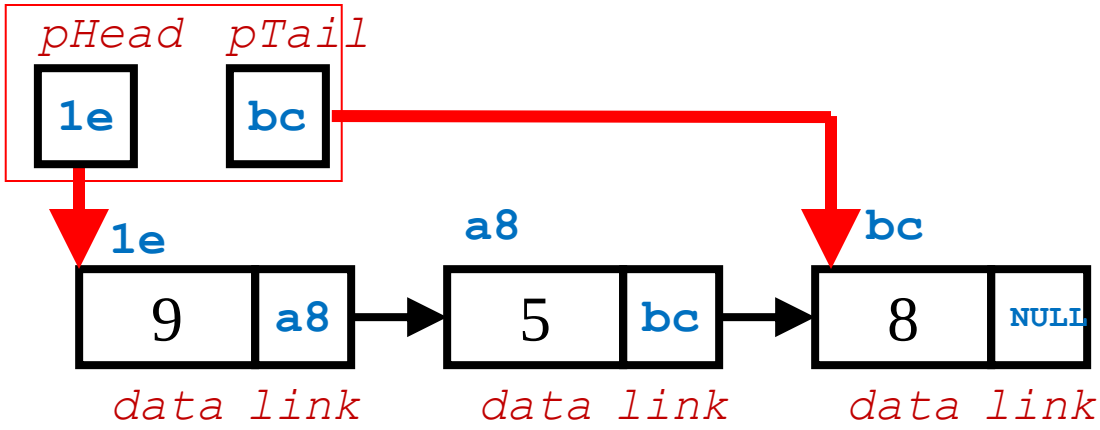


– Định nghĩa biến quản lý danh sách

```
NodePointer  phead;
```

2.8. Cấu tạo của danh sách liên kết

2.8.2. Quản lý DSLK đơn với 1 struct để quản lý phần tử đầu và phần tử cuối



- Danh sách rỗng: **pHead = NULL; pTail = NULL**
- Định nghĩa cấu trúc lưu trữ

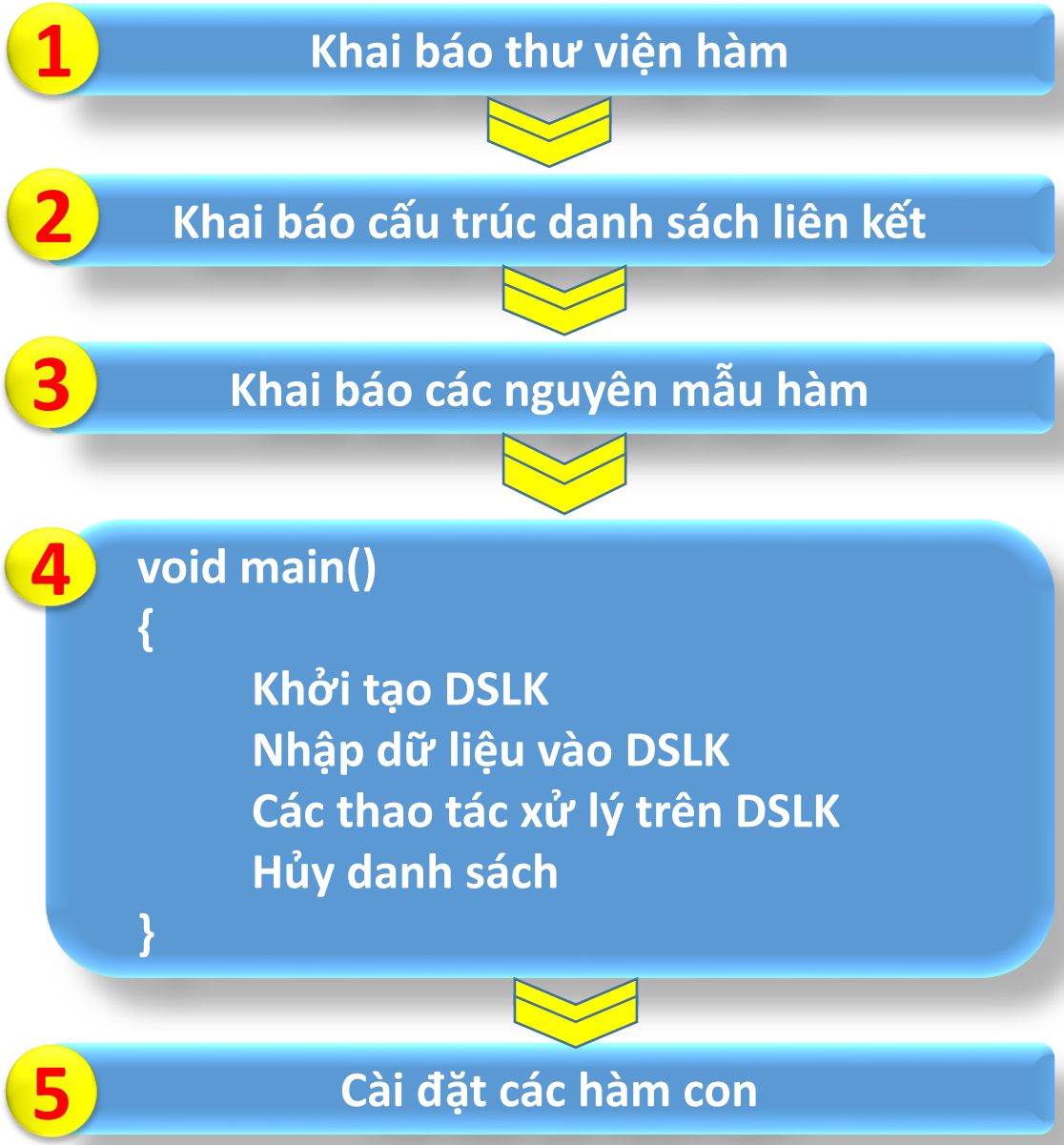
```
typedef struct tagNode
{    Data      data;
     tagNode   *pNext;
}NODE;
```

- Định nghĩa biến quản lý danh sách:

```
typedef struct  tagList
{    Node   *pHead;
     Node   *pTail;
}List;
```

3. Lập trình với danh sách liên kết đơn (*Linked List*)

Cấu trúc tổng quát chương trình

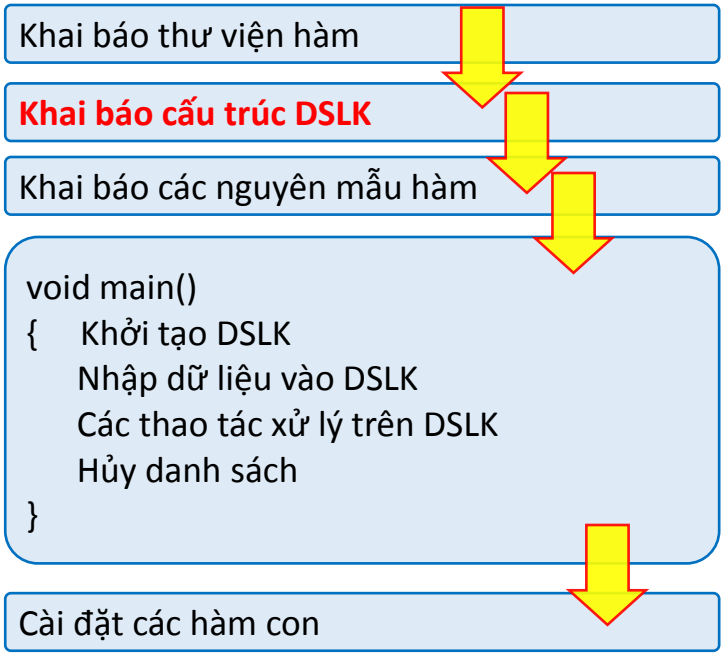


3.1. Khai báo cấu trúc DSLK

3.1.1. Khai báo cấu trúc node cần dùng

- Khai báo khi thành phần cuả node là đơn
  - ***VD1***: Quản lý danh sách với data chỉ là số nguyên

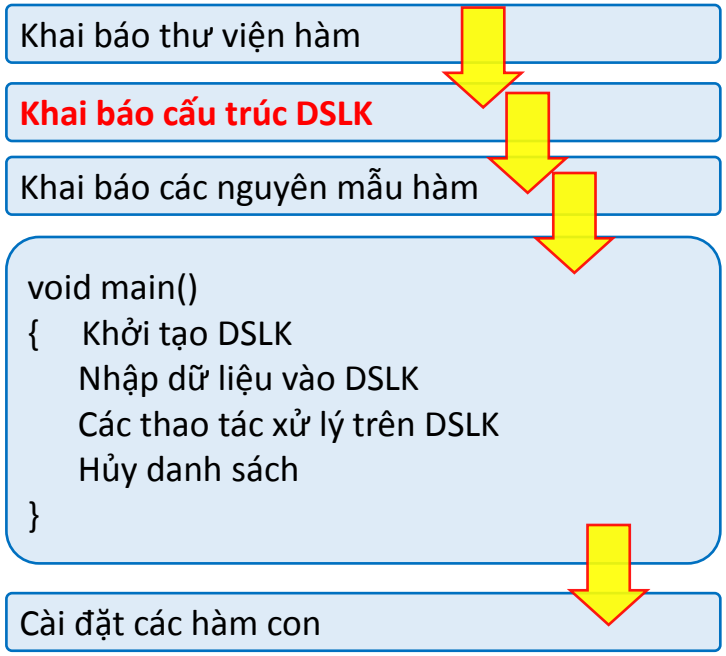
```
typedef struct tagNode
{    // Lưu thông tin bản thân node
     int data;
     /* Lưu địa chỉ của node đứng sau*/
     tagNode* pNext;
}NODE;
```



3.1. Khai báo cấu trúc DSLK

3.1.1. Khai báo cấu trúc node cần dùng

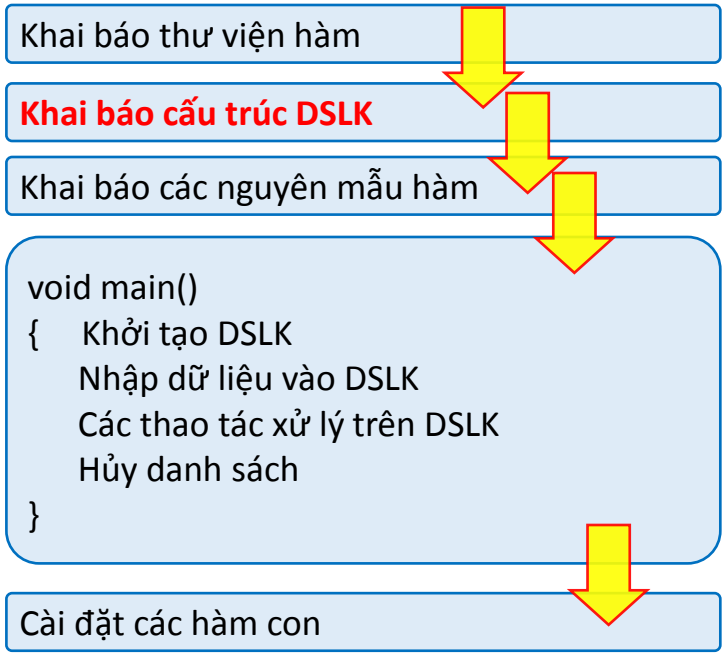
- Khai báo khi thành phần cuả node là một *struct*
  - ***VD2***: DSLK với data là 1 struct




```
//Cấu trúc thông tin của một SV
typedef struct SinhVien
{   char MSSV[10];
    char HoTen[40];
    float ĐTB;
}SV;

//Cấu trúc của node với thành phần chứa cấu trúc SV
typedef struct tagNode
{   SV        data;
    struct    tagNode* pNext;
}Node;
```

3.1. Khai báo cấu trúc DSLK



3.1.2. Khai báo cấu trúc quản lý DSLK

- Khai báo khi chỉ dùng pHead

VD3:

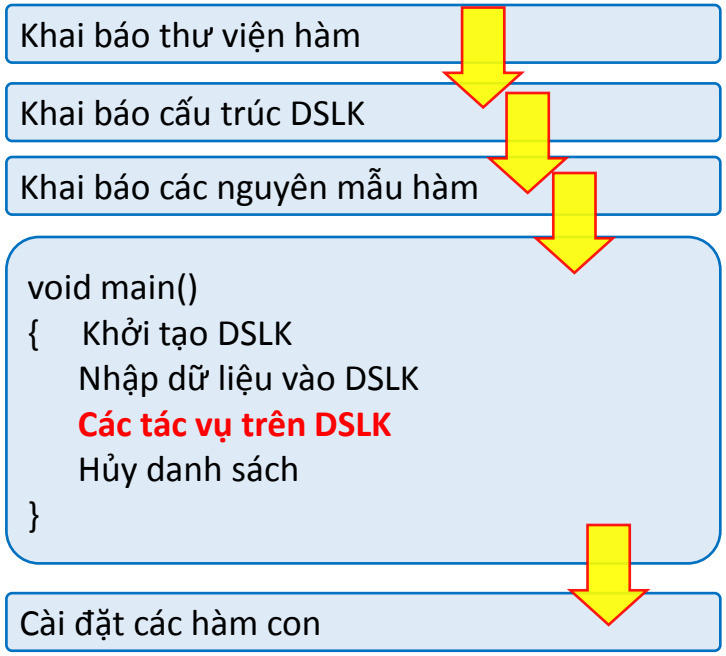
```
typedef struct NODE *NodePointer;
NodePointer  pHead;
```

- Khai báo cấu trúc khi dùng cả *pHead* và *pTail*

VD4:

```
typedef struct  tagList
{    Node  *pHead;
     Node  *pTail;
}List;
```

Chọn cấu trúc đơn giản này để minh họa cho các tác vụ của DSLK đơn



3.2. Các tác vụ cơ bản trên DSLK đơn

i. Tác vụ Initialize
ii. Tác vụ IsEmpty
iii. Tác vụ CreateNode
iv. Tác vụ Insert
- Tác vụ InsertFirst
- Tác vụ InsertAfter

v. Tác vụ ShowList
vi. Tác vụ Search
vii. Tác vụ Delete
- Tác vụ DeleteFirst
- Tác vụ DeleteAfter

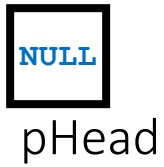
viii. Tác vụ ClearList
ix. Tác vụ Sort
x. Một số tác vụ trên danh sách liên kết có thứ tự
- Tác vụ InsertOrderly
- Tác vụ Merge2List

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.1. Tác vụ Initialize

- *Công dụng*: Khởi tạo danh sách ban đầu là rỗng
- *Tham số đầu vào*: biến pHead làm tham biến
- *Kết quả trả về*: không
- *Cài đặt*

```
void Initialize(NodePointer &pHead)
{
    pHead = NULL;
}
```



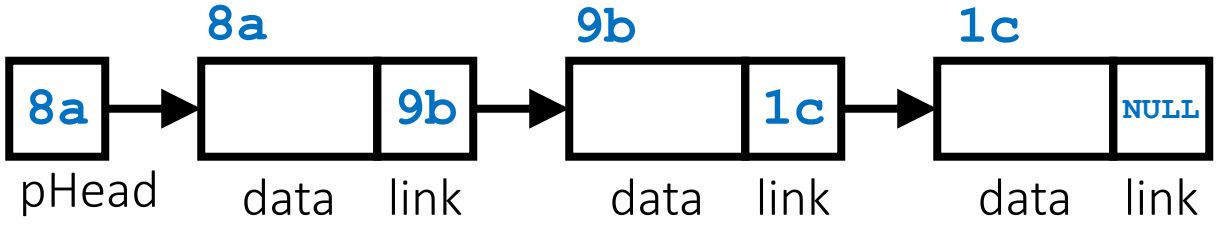
3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.2. Tác vụ IsEmpty

- *Công dụng*: kiểm tra danh sách có rỗng hay không?
- *Tham số đầu vào*: biến pHead làm tham trị.
- *Kết quả trả về*: true | false
- *Cài đặt*:

```
bool isEmpty(NodePointer pHead)
{       return (pHead==NULL)
}
```

⇒ **true**



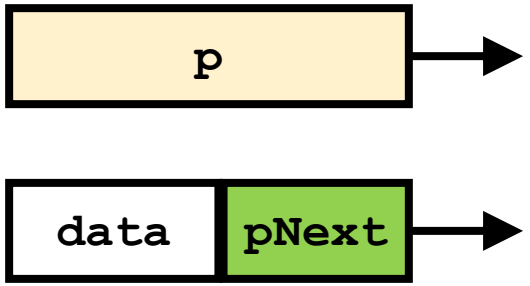
⇒ **false**

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.3. Tác vụ CreateNode

- *Công dụng*: Cấp phát vùng nhớ cho 1 node.
- *Tham số đầu vào*: data của node
- *Kết quả trả về*: địa chỉ của node vừa cấp phát.
- *Cài đặt*:

```
NodePointer CreateNode(int x)
{    NodePointer p=new Node;
     if (p==NULL)
     {    printf("Không cấp được node mới");
          return NULL;
     }
     p->data=x;
     p->pNext=NULL;
     return p;
}
```

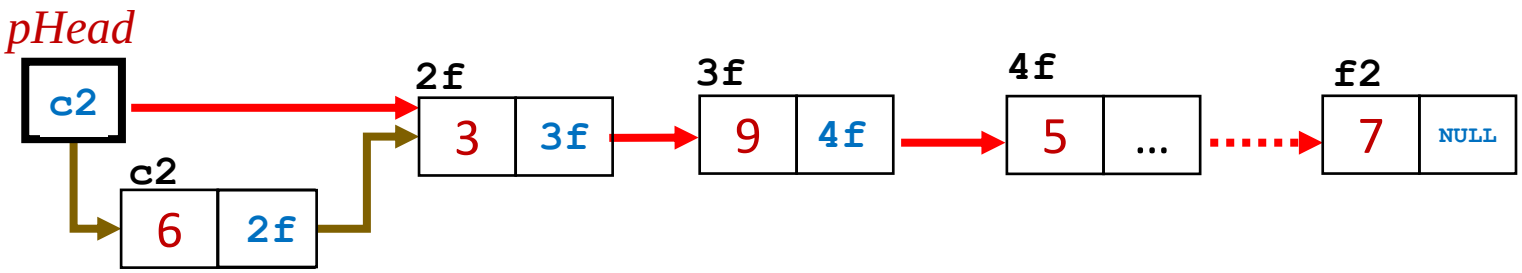


3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.4. Tác vụ Insert

- *Công dụng*: Thêm 1 node vào DSLK.
- *Kết quả trả về*: không
- *Phân loại*: gồm 2 trường hợp:
  - ***InsertFirst***
    - *Công dụng*: Thêm 1 node có data là x vào **đầu** DSLK.
    - *Tham số đầu vào*: pHead làm tham biến và data làm tham trị.
    - *Cài đặt:*

```
void InsertFirst(NodePointer &pHead, int x)
{    NodePointer p=CreateNode(x);
     if (p!=NULL)
     {    p->pNext=pHead;
          pHead=p;
     }
}
```



3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.4. Tác vụ Insert

- *Thực hành:* viết hàm nhận 2 tham số là pHead và số nguyên n. Hàm thực hiện tạo DSLK gồm n node (node có cấu trúc như đã khai báo)

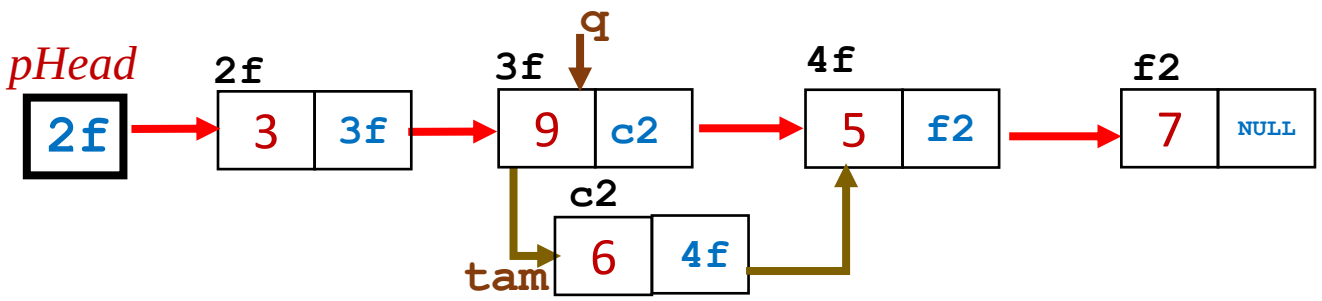
```
void CreateList(NodePointer &pHead)
{  int n;
   //B1: nhập số node cần tạo cho DSLK
   printf("\nNhap n: ");
   scanf("%d", &n);
   //B2: tạo DSLK
   for (int i = 0; i < n; i++)
   { int x;
     printf("Nhap gia tri cho Node thu %d: ",i+1);
     scanf("%d", &x);
     //Tạo 1 node chứa x rồi chén node vào đầu DSLK
     InsertFirst(pHead,x);
   }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.4. Tác vụ Insert

- ***InsertAfter***
  - *Công dụng*: Thêm 1 node có data là x vào sau node có địa chỉ là q. Do đó tác vụ này thường được gọi sau khi đã thực hiện tác vụ tìm kiếm được địa chỉ q.
  - *Tham số đầu vào*: biến q và data cùng làm tham trị.
  - *Cài đặt*

```
void InsertAfter (NodePointer q, int x)
{    NodePointer tam;
     if(q!=NULL)
     {    tam = createNode(x);
          tam->pNext = q->pNext;
          q->pNext = tam;
     }
}
```



3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.4. Tác vụ Insert

- ***InsertAfter***
  - *Thực hành*: Viết hàm thêm 1 node có data là x vào sau số nguyên tố đầu tiên có trong DSLK. Nếu DSLK không có SNTố, sẽ không thực hiện chèn). Yêu cầu sử dụng hàm InsertAfter đã có

```
void ChenXVaoSauSNT_DauTien (NodePointer &pHead)
{ NodePointer q=pHead;
      int x;
      //B1 nhap X
      printf("Nhap X: ");
      scanf_s("%d", &x);
      //B2: duyệt DSLK
      while(q!=NULL)
      {     //nếu data là số chẵn
            if (LaSNTo(q->data))
            {     //gọi hàm InsertAfter(q,X)
                  InsertAfter(q,X);
                  break;
            }
            q=q->pNext;
      }
}
```

```
bool LaSNTo (int k)
{ NodePointer q=pHead;
  if (k<=1)
      return false;
  for (int i=2; i<=sqrt(k);i++)
      if (k%i==0)
            return false;
  return true;
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.4. Tác vụ Insert

- ***InsertAfter***
  - *Thực hành*: Viết hàm thêm 1 node có data là x vào sau tất cả các node là số chẵn đang có trong DSLK.

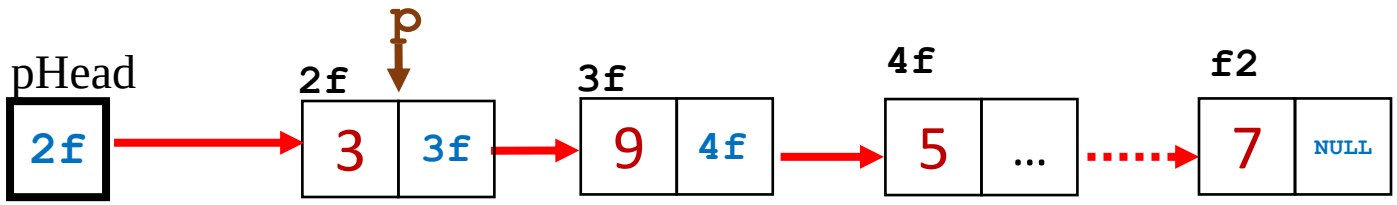
```
void ChenXVaoSauCacSoChan (NodePointer &pHead)
{ NodePointer q=pHead;
     int x;
     //B1 nhap X
     printf("Nhap X: ");
     scanf_s("%d", &x);
     //B2: duyệt DSLK
     while(q!=NULL)
     {    //nếu data là số chẵn
          if (q->data%2==0)
          {    //gọi hàm InsertAfter(q,X)
               InsertAfter(q,X);
               q=q->pNext;
          }
          if (q!=NULL)
               q=q->pNext;
     }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.5. Tác vụ ShowList

- *Công dụng*: duyệt danh sách liên kết, hiển thị thông tin các nút.
- *Tham số đầu vào*: biến pHead làm tham trị.
- *Kết quả trả về*: không
- *Cài đặt*

```
void ShowList(NodePointer pHead)
{    NodePointer p=pHead;
     if(p==NULL)
          printf("\n Danh sach bi rong");
     else
     {
          while(p!=NULL)
          {    printf("%5d",p->data);
               p=p->pNext;
          }
     }
}
```



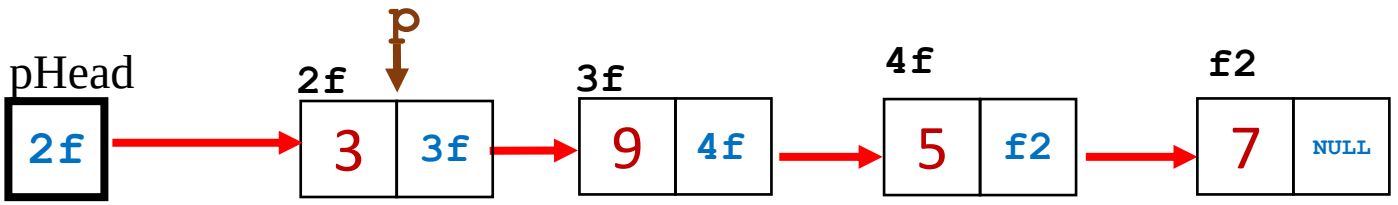
3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.6. Tác vụ Search

- *Công dụng*: Tìm node đầu tiên trong danh sách có data bằng với x trên DSLK bằng phương pháp tìm tuyến tính.
- *Tham số đầu vào*: biến pHead và data x làm tham trị.
- *Kết quả trả về*: địa chỉ của node có data=x nếu tìm thấy hoặc NULL khi không tìm thấy.
- *Cài đặt*

```
NodePointer Search(NodePointer pHead, int x)
{    NodePointer p=pHead;
     while(p->data !=x && p!=NULL)
          p=p->pNext;
     return p;
}
```

X=8



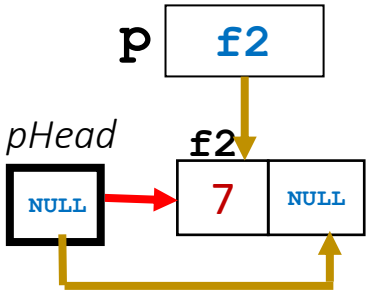
3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.7. Tác vụ Delete

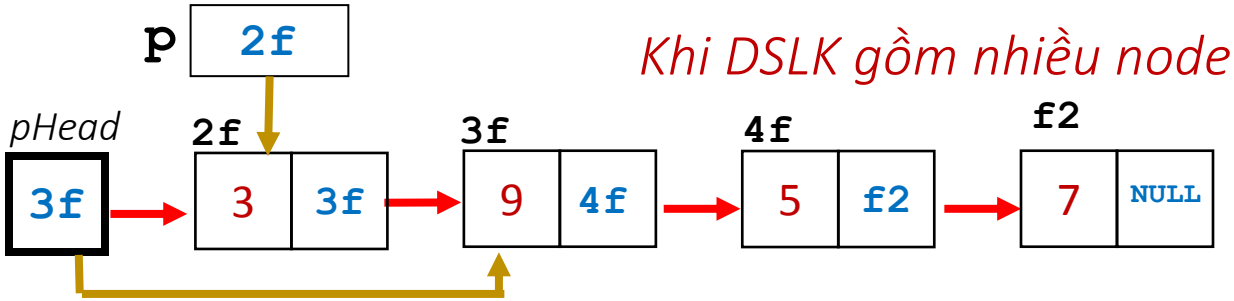
- *Công dụng*: Xóa 1 node khỏi DSLK.
- *Kết quả trả về*: không
- *Phân loại*: gồm 2 trường hợp
  - ***Tác vụ DeleteFirst***
    - *Công dụng*: Xóa node đầu tiên của DSLK.
    - *Tham số đầu vào*: biến pHead làm tham biến.
    - *Cài đặt*

```
void DeleteFirst(NodePointer &pHead)
{     NodePointer p;
      if (IsEmpty(pHead))
            printf("List is empty!");
      else
      {     p = pHead;
            pHead = pHead->pNext;
            delete p;
      }
}
```

Khi DSLK chỉ gồm 1 node



Khi DSLK gồm nhiều node



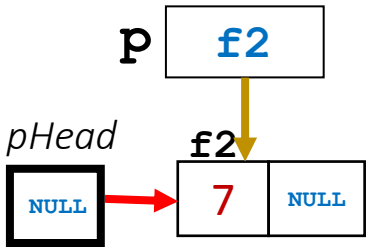
3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.7. Tác vụ Delete

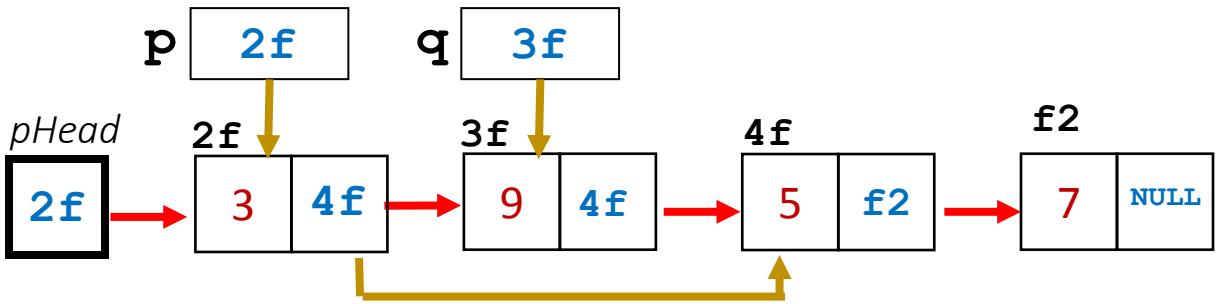
- ***Tác vụ DeleteAfter***
  - *Công dụng*: Xóa node sau node p trong danh sách DSLK.
  - *Tham số đầu vào*: biến p thuộc kiểu NodePointer làm tham trị.
  - *Cài đặt*

```
void DeleteAfter(NodePointer p)
{     //kiem tra nut sau p co ton tai hay khong?
      if ( (p==NULL) || (p->pNext ==NULL) )
            printf("Cannot delete node!");
      else
      {     NodePointer q = p->pNext;
            p->pNext = q->pNext;
            delete q;
      }
}
```

Khi p->pNext==NULL



Khi p->pNext!=NULL



3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.8. Tác vụ ClearList

- *Công dụng*: giải phóng tất cả các nút có trên DSLK. Ta có thể sử dụng lệnh pHead = NULL để xóa toàn bộ danh sách, nhưng trong bộ nhớ, các vùng nhớ đã cấp phát cho các node không giải phóng về lại cho memory heap, nên sẽ lãng phí vùng nhớ.
- *Tham số đầu vào*: biến pHead làm tham biến.
- *Kết quả trả về*: không.
- *Cài đặt*

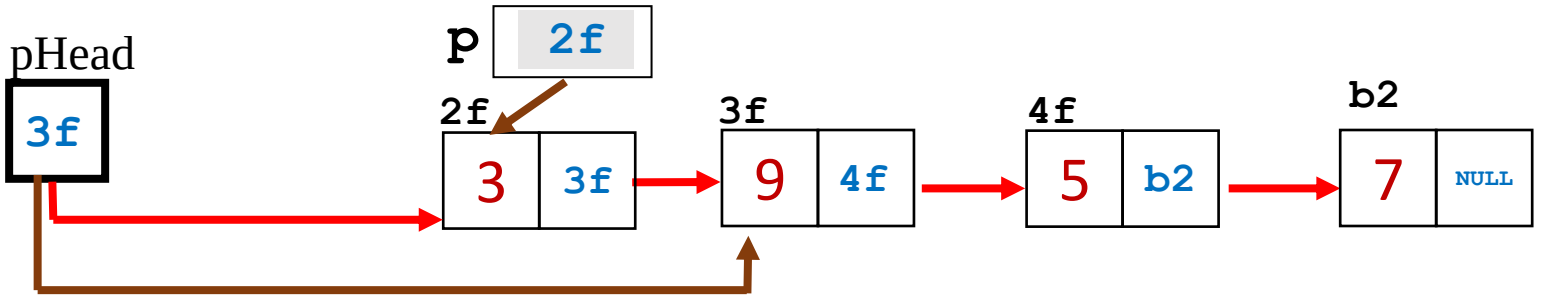
```
void ClearList(NodePointer &pHead)
{    NodePointer p;
     while (pHead!=NULL)
     {    p = pHead;
          pHead = p->pNext;
          delete p;
     }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.8. Tác vụ ClearList

- *Cài đặt*

```
void ClearList(NodePointer &pHead)
{   NodePointer p;
    while (pHead!=NULL)
    {   p = pHead;
        pHead = p->pNext;
        delete p;
    }
}
```

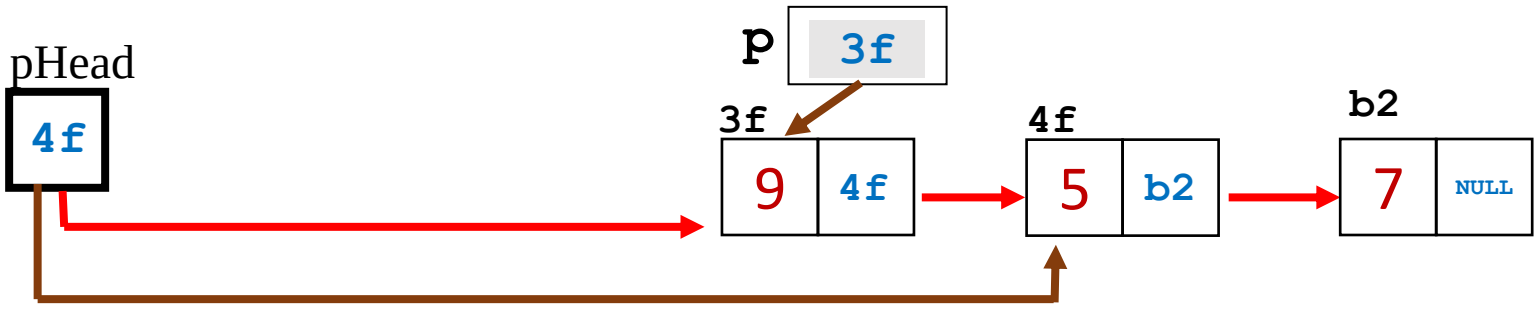


3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.8. Tác vụ ClearList

- *Cài đặt*

```
void ClearList(NodePointer &pHead)
{   NodePointer p;
    while (pHead!=NULL)
    {   p = pHead;
        pHead = p->pNext;
        delete p;
    }
}
```



3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.8. Tác vụ ClearList

- *Cài đặt*

```
void ClearList(NodePointer &pHead)
{   NodePointer p;
    while (pHead!=NULL)
    {   p = pHead;
        pHead = p->pNext;
        delete p;
    }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.8. Tác vụ ClearList

- *Cài đặt*

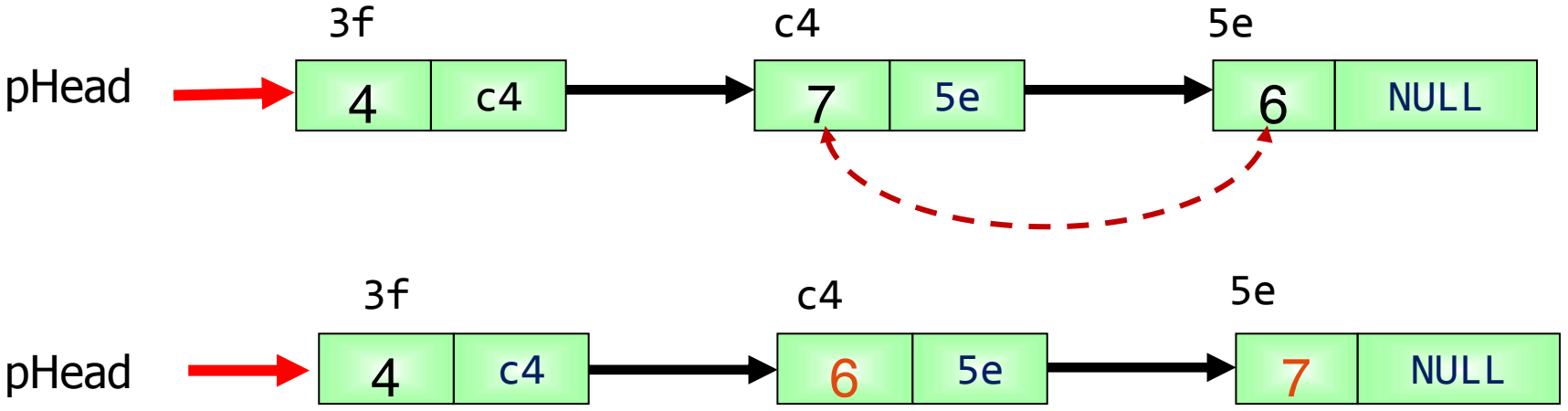
```
void ClearList(NodePointer &pHead)
{   NodePointer p;
    while (pHead!=NULL)
    {   p = pHead;
        pHead = p->pNext;
        delete p;
    }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.9. Tác vụ Sort

Có hai cách tiếp cận

- **Cách 1**: Thay đổi thành phần `data`



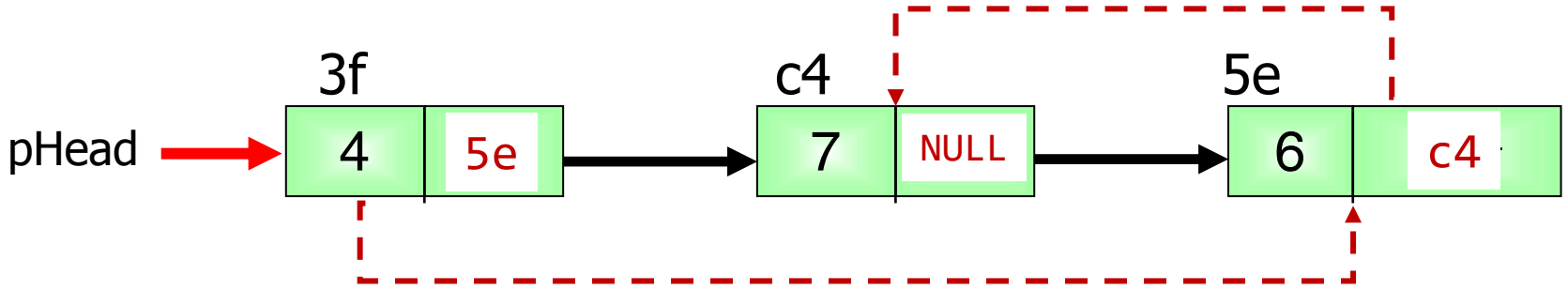
- *Ưu*: Cài đặt đơn giản, tương tự như sắp xếp mảng
- *Nhược*:
  - Đòi hỏi thêm vùng nhớ khi hoán vị nội dung của 2 phần tử ⇒ chỉ phù hợp với những DSLK có kích thước data nhỏ.
  - Khi kích thước data (dữ liệu) lớn chi phí cho việc hoán vị thành phần data lớn ⇒ thao tác sắp xếp chậm

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.9. Tác vụ Sort

Có hai cách tiếp cận

- **Cách 2**: Thay đổi thành phần **`pNext`** (thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn)



- *Ưu*:
  - Kích thước của trường này không thay đổi ⇒ không phụ thuộc vào kích thước bản chất dữ liệu lưu tại mỗi node.
  - Thao tác sắp xếp nhanh
- *Nhược*: Cài đặt phức tạp

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.9. Tác vụ Sort

(Cách 1: thay đổi thành phần data)

```
void SelectionSort(NodePointer &pHead)
{   NodePointer p, q, min;
    p = pHead;
    while (p->pNext != NULL)
    {   min = p;
        q = p->pNext;
        while (q != NULL)
        {       if (q->data < min->data)
                        min = q;
                q = q->pNext;
        }
        swap(min->data, p->data);
        p = p->pNext;
    }
}
void swap(int &a, int &b)
{       int tmp = a ;
        a = b;
        b = tmp;
}
```

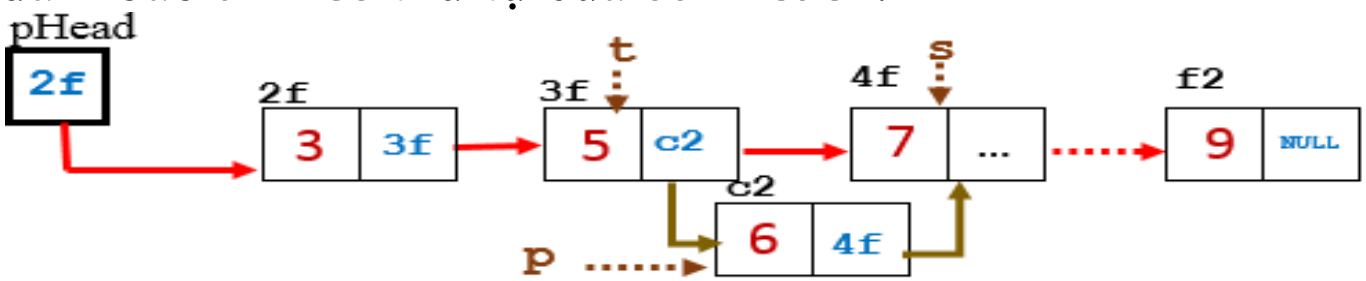
```
void SelectionSort(int a[],int n )
{  int min,i,j;
   for(i=0;i<n-1;i++)
   {
      min = i;
      for(j = i+1; j <n ; j++)
          if (a[j ] < a[min])
             min=j;
      swap(a[min],a[i]);
   }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.10. Một số tác vụ trên danh sách liên kết có thứ tự

i. Tác vụ InsertOrderly

- Thêm vào DSLK có thứ tự một phần tử có data (x=6) sao cho sau khi thêm vào vẫn đảm bảo tính có thứ tự của danh sách.



- Cài đặt

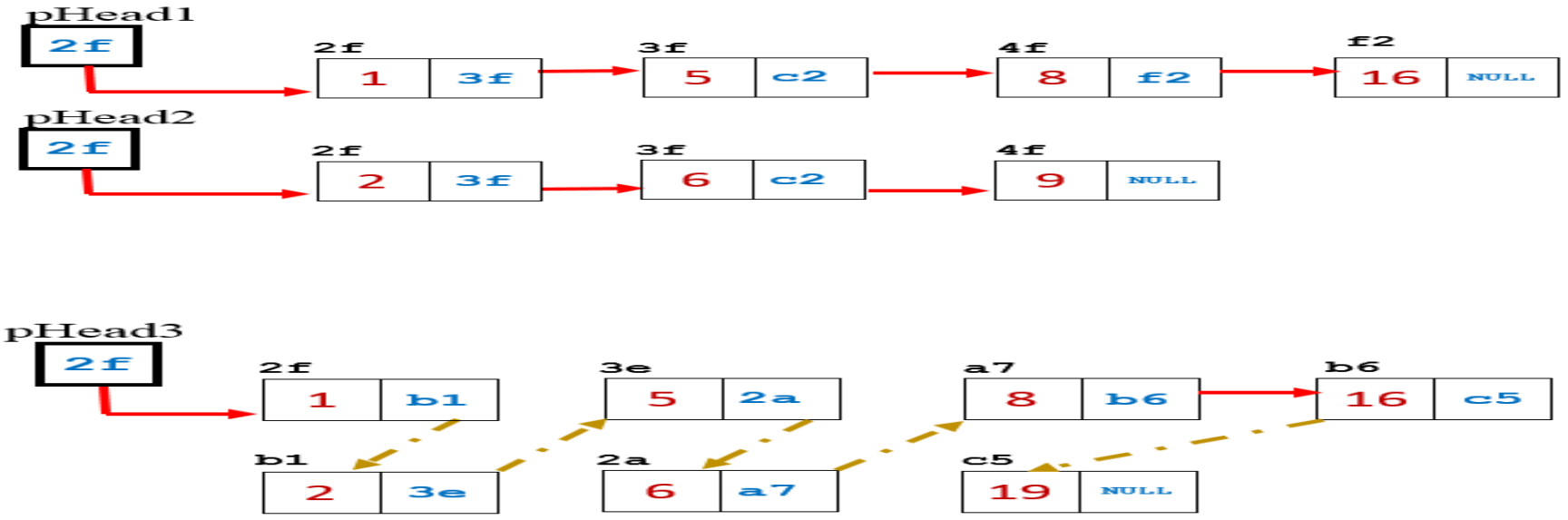
```
void InsertOrderly(NodePointer &pHead, int x)
{   NodePointer p, t, s;  //t la nut truoc, s la nut sau
    p=new NODE;
    p->data=x;
    for(s = pHead; s!=NULL && s->data<x ; t=s, s=s->pNext);
        if(s == pHead)//them nut vao dau DSLK
        {       p->pNext = pHead;
                pHead = p;
        }
        else     // them nut p vao truoc nut s
        {       p->pNext= s;
                t->pNext= p;
        }
}
```

3.2. Các tác vụ cơ bản trên DSLK đơn

3.2.10. Một số tác vụ trên danh sách liên kết có thứ tự

ii. Tác vụ Merge2List

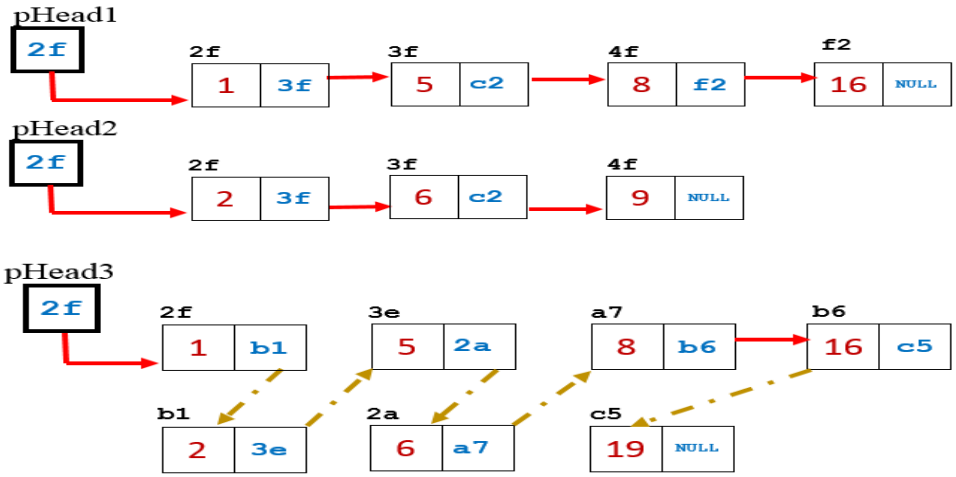
- Cho hai danh sách liên kết pHead1, pHead2 đã có thứ tự. Hãy trộn hai danh sách này lại thành một danh sách liên kết mới pHead3 sao cho pHead3 cũng có thứ tự.



ii. Tác vụ Merge2List

- Cài đặt

```
NodePointer Merge2List(NodePointer &pHead1, NodePointer &pHead2)
{  NodePointer p1, p2, p3;
   NodePointer pHead3= new NODE ;
   p1=pHead1; p2 = pHead2; p3=pHead3;
   while (p1 !=NULL && p2 !=NULL)
      if (p1->data < p2->data)
      {  p3->pNext = p1;
         p3=p1;
         p1=p1->pNext ;
      }
      else
      {   p3->pNext = p2;
          p3=p2;
          p2=p2->pNext ;
      }
      if (p1==NULL)    p3->pNext=p2;
      else             p3->pNext=p1;
      p3 = pHead3;
      pHead3=p3->pNext;
      delete p3;
      pHead1=pHead2=NULL;
      return pHead3;
```



4. THỰC HÀNH

4.1. Cần quản lý 1 DSLK với thành phần dữ liệu chỉ là 1 số nguyên. Thực hiện các yêu cầu sau:

i. Khai báo cấu trúc *NODE* và kiểu con trỏ (*NodePointer*) cần dùng để quản lý DSLK.

ii. Sử dụng các tác vụ đã có là *CreateNode* và *InsertFirst*, viết hàm *CreateList1* hực hiện tạo DSLK với giá trị các node do người dùng nhập vào. Việc nhập kết thúc khi giá trị nhập vào =0.

```
void CreateList1 (NodePointer &pHead) { ... }
```

iii. Viết hàm *CreateList2* thực hiện tạo n node với giá trị được phát sinh ngẫu nhiên trong khoảng từ -100 đến +100.

```
void CreateList2 (NodePointer & pHead, int n) { ... }
```

Mở rộng: phát sinh ngẫu nhiên giá trị các node thỏa điều kiện:

- Các trường hợp số âm, số chẵn, số vừa âm vừa chẵn, …
- Nằm trong khoảng từ 50 đến 100.
- Nhỏ hơn 10 hoặc lớn hơn hay bằng 50.
- Giá trị các phần tử tăng dần.

4. THỰC HÀNH

4.1. Quản lý DSLK với thành phần dữ liệu chỉ là 1 số nguyên. Viết các hàm thực hiện chức năng:

iv. Xuất toàn bộ danh sách ra màn hình

```
void ShowList (NodePointer pHead) { … }
```

v. Đếm các phần tử trong danh sách.

```
int DemNode (NodePointer pHead) { … }
```

vi. Tính trung bình các số có trong DSLK

```
float TinhTrungBinh (NodePointer pHead) { … }
```

vii. Tìm xem DSLK có chứa data bằng giá trị X cho trước.

```
bool TimX(NodePointer pHead, int X) { … }
```

viii. Tìm giá trị lớn/nhỏ nhất trong DSLK

```
int TimMax(NodePointer pHead) { … }
```

ix. Tìm vị trí đầu tiên chứa số lớn/nhỏ nhất (vị trí tính từ 1, có thể có nhiều node cùng có giá trị lớn/nhỏ nhất)

```
int TimViTriMax(NodePointer pHead) { … }
```

4. THỰC HÀNH

4.1. Quản lý DSLK với thành phần dữ liệu chỉ là 1 số nguyên. Viết các hàm thực hiện chức năng:

x. Kiểm tra DSLK:

- Chứa toàn số dương (toàn dương trả về true, ngược lại trả về false)

```
bool ToanDuong(NodePointer pHead) { ... }
```

- Chứa toàn số âm.
- Chứa toàn số nguyên tố.
- Đã được sắp tăng dần (tăng dần trả về true, ngược lại trả về false)

xi. Cho nhập số nguyên X, chèn X vào sau số nguyên tố đầu tiên có trong DSLK. Yêu cầu: trong hàm ChenXSauSNT sẽ gọi hàm InsertAfter.

```
void ChenXSauSNT (NodePointer & pHead) { ... }
```

xii. Cho nhập số nguyên X, chèn X vào sau tất cả các số chẵn có trong DSLK. Yêu cầu: trong hàm ChenXSauChan sẽ gọi hàm InsertAfter.

```
void ChenXSauChan (NodePointer & pHead) { ... }
```

4. THỰC HÀNH

4.1. Quản lý DSLK với thành phần dữ liệu chỉ là 1 số nguyên. Viết các hàm thực hiện chức năng:

xiii. Xóa node đầu tiên có giá trị X (DSLK có thể có nhiều giá trị X)

```
void XoaX (NodePointer &List, int X) { ... }
```

xiv. Xóa các node có giá trị (data) trùng nhau

```
bool XoaTrung(NodePointer &List) { ... }
```

xv. Tách DSLK A thành 2 DSLK B và C. Với B chỉ chứa các số nguyên tố, C chỉ chứa các số không là nguyên tố.

```
void TachSNTo(NodePointer pHeadA, NodePointer & pHeadB,
                                  NodePointer & pHead C) { ... }
```

4. THỰC HÀNH

4.2. Cần quản lý 1 danh mục các sinh viên. Biết thông tin cần quản lý của sinh viên gồm mã số (kiểu số nguyên), tên sinh viên (chiều dài 30 ký tự), điểm lý thuyết (kiểu số thực), điểm thực hành (kiểu số thực).

Sử dụng DSLK để quản lý danh mục sinh viên. Thực hiện các yêu cầu sau, trong đó có thể viết thêm các hàm khác để phục vụ cho yêu cầu đang thực hiện:

i. Khai báo các cấu trúc (NODE) và kiểu con trỏ cần dùng (*NodePointer*) để quản lý DSLK.

ii. Viết hàm *Init* để khởi tạo tạo DSLK.

```
void Init (NodePointer &pHead) { ... }
```

iii. Viết hàm *IsEmpty* để kiểm tra DSLK có rỗng hay không? Nếu rỗng trả về true, ngược lại trả về false.

```
bool IsEmpty (NodePointer pHead) { ... }
```

4.2. Quản lý DSLK Sinh viên:

iv. Viết hàm *CreateNode* để tạo 1 node chứa thông tin về 1 sinh viên.

```
NodePointer CreateNode (SinhVien x) { ... }
```

v. Viết hàm InsertFirst để thêm 1 node chứa X vào đầu DSLK

```
void InsertFirst (NodePointer &pHead, SV x)
{     //B1: gọi hàm CreateNode(x) để tạo 1 node p mới chứa thông tin SV x
      //B2: Nếu tạo được node mới thì thực hiện đưa node p vào đầu DSLK
}
```

vi. Viết hàm CreateList để tạo DSLK gồm n node

```
void CreateList (NodePointer &pHead)
{     //B1: nhập n
      //B2: nhập DLK
      for( int i=0; i<n; i++)
      {       //B2.1: gọi hàm nhập thông tin 1 SV  s
              //B2.2: gọi hàm InsertFirst để đưa node chứa SV x vào đầu DSLK
      }
}
```

4.2. Quản lý DSLK Sinh viên:

vii. Viết hàm xuất thông tin của 1 node ra màn hình

void Xuat1SV (SV p) { … }

viii. Xuất toàn bộ danh sách ra màn hình. Hàm này sẽ thực hiện gọi hàm *Xuat1SV*

void ShowList(NodePointer List) { … }

ix. Đếm số lượng các phần tử có trong danh sách.

int Dem (NodePointer pHead) { … }

x. Tính điểm trung bình của tất cả các SV có trong DSLK, biết rằng:

DTB = (Điểm lý thuyết + Điểm thực hành)/2

float DTB (NodePointer pHead) { … }

xi. Xuất ra thông tin của những SV có điểm trung bình lớn nhất trong DSLK

float DanhSachDTBmax (NodePointer pHead) { … }

4. THỰC HÀNH

4.2. Quản lý DSLK sinh viên

xii. Tìm và xuất ra thông tin của SV có mã số là '123'. Nếu không tìm thấy, in ra thông báo.

```
float TimSV (NodePointer pHead, int MaSo) { ... }
```

xiii. Tách DSLK A thành 2 DSLK B và C. Với B chỉ chứa các SV có điểm trung bình >=5, C chỉ chứa các SV có điểm trung bình <5.

```
float TachDS (NodePointer pHeadA, NodePointer pHeadB,
                                  NodePointer pHeadC) { ... }
```

xiv. Xóa SV có mã số 123. Hàm trả về true nếu xóa thành công, ngược lại khi không tìm thấy trả về false. Hàm sẽ sử dụng 2 hàm *DeleteFirst* và *DeleteAfter* (đã có)

```
bool XoaSV (NodePointer pHead, int MaSo) { ... }
```

4. THỰC HÀNH

4.2. Quản lý DSLK sinh viên

xv. Sử dụng thuật toán *Interchange Sort* để sắp xếp DSLK giảm dần theo điểm trung bình.

```
float TimSV (NodePointer pHead, int MaSo) { … }
```

4. THỰC HÀNH

4.3. Cần quản lý 1 danh mục các hàng hóa. Biết thông tin cần quản lý của hàng hóa gồm mã số (chiều dài 6 ký tự), tên hàng hóa(chiều dài 30 ký tự), Giá vốn (kiểu số thực), Số lượng tồn (kiểu số nguyên) và ngày hết hạn (là 1 biến kiểu cấu trúc gồm các thành phần là ngày tháng, năm; tất cả đều là kiểu số nguyên).

Sử dụng DSLK để quản lý danh mục hàng hóa. Thực hiện các yêu cầu sau, trong đó SV có thể viết thêm các hàm khác để phục vụ cho yêu cầu đang thực hiện:

i. Khai báo các cấu trúc (NODE) và kiểu con trỏ cần dùng (NodePointer) để quản lý DSLK.

ii. Viết hàm CreateNode để tạo 1 node chứa thông tin về 1 hàng hóa.

```
NodePointer CreateNode(HANGHOA x) { ... }
```

iii. Viết hàm CreateList để tạo DSLK gồm n node

```
void CreateList (NodePointer &List) { ... }
```

4. THỰC HÀNH

4.3. Quản lý DSLH hàng hóa.

iv. Viết hàm xuất thông tin của 1 node ra màn hình

```
void ShowNode (NodePointer p) { … }
```

v. Viết hàm xuất toàn bộ danh sách ra màn hình

```
void ShowList (NodePointer List) { … }
```

vi. Đếm số lượng hàng hóa có trong DSLK.

vii. Xuất ra thông tin của những hàng hóa trị giá hàng hóa lớn nhất trong DSLK. Biết Trị giá hàng hóa= Đơn giá X Số lượng tồn.

viii. Tính tổng trị giá hàng hóa có trong DSLK

ix. Tìm và xuất ra thông tin của hàng hóa có mã số là ‘C0123’. Nếu không tìm thấy, in ra thông báo.

iv. Tạo ra 1 DSLK mới chứa các hàng hóa có trị giá hàng hóa >=1 tỷ.

5. BÀI TẬP VỀ CÁC DSLK KHÔNG ĐƠN

(SV tự cài đặt các dạng danh sách liên kết và báo cáo)

i. *Bài tập 3.11*: DSLK đôi
ii. *Bài tập 3.12*: DSLK đơn vòng
iii. *Bài tập 3.13*: DSLK đôi vòng

Các thao tác thực hiện tương tự như trên DSLK đơn như:

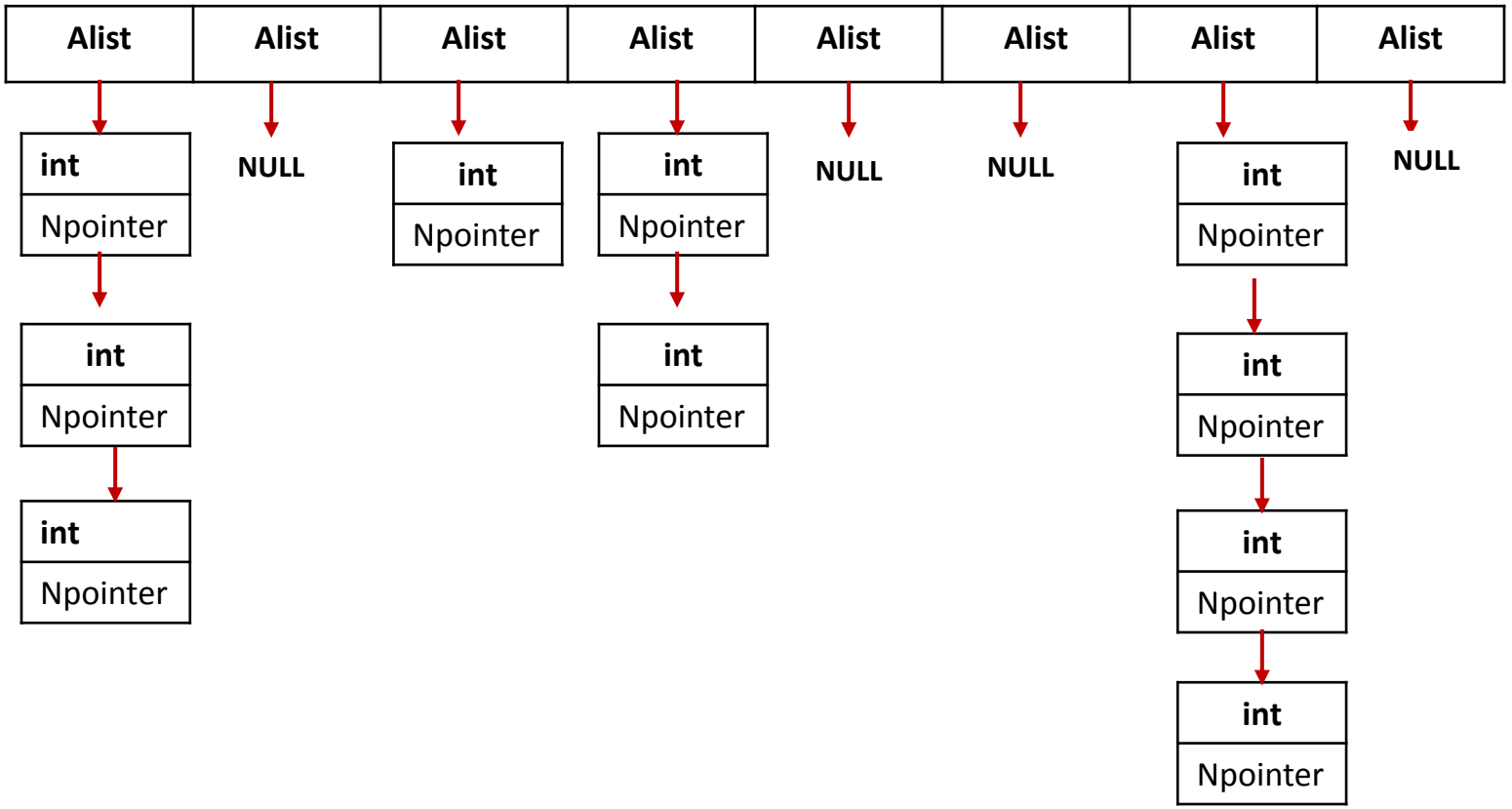
i. Khai báo cấu trúc cần dùng
ii. Khởi tạo 1 DSLK rỗng
iii. Tạo 1 node có data bằng X
iv. Thêm một phần tử có data là X vào danh sách
    - Thêm vào đầu danh sách
    - Thêm vào cuối danh sách
    - Thêm vào trước/sau phần tử có khóa là Y
    - Thêm vào danh sách sao cho giữ đúng thứ tự tăng/giảm dần
v. Duyệt DSLK
vi. Tìm một phần tử có data bằng X
vii. Hủy một phần tử trong DSLK
viii. Sắp xếp DSLK

6. CÁC DẠNG PHỐI HỢP KHÁC DỰA TRÊN DSLK

(SV cài đặt các dạng danh sách liên kết và báo cáo)

i. Mảng các DSLK: trong đó, thành phần dữ liệu:

- *Mảng*: mỗi phần tử là 1 cấu trúc gồm pHead và pTail.
- *Node*: gồm Data là số nguyên và liên kết là pointer chỉ đến Node tương ứng.



6. Các dạng phối hợp khác dựa trên DSLK

(SV cài đặt các dạng danh sách liên kết và báo cáo)

ii. DSLK với nội dung chứa 1 DSLK khác

Có nhu cầu lưu các số nguyên từ 0-1000. Trong đó, thành phần của các loại node như sau:

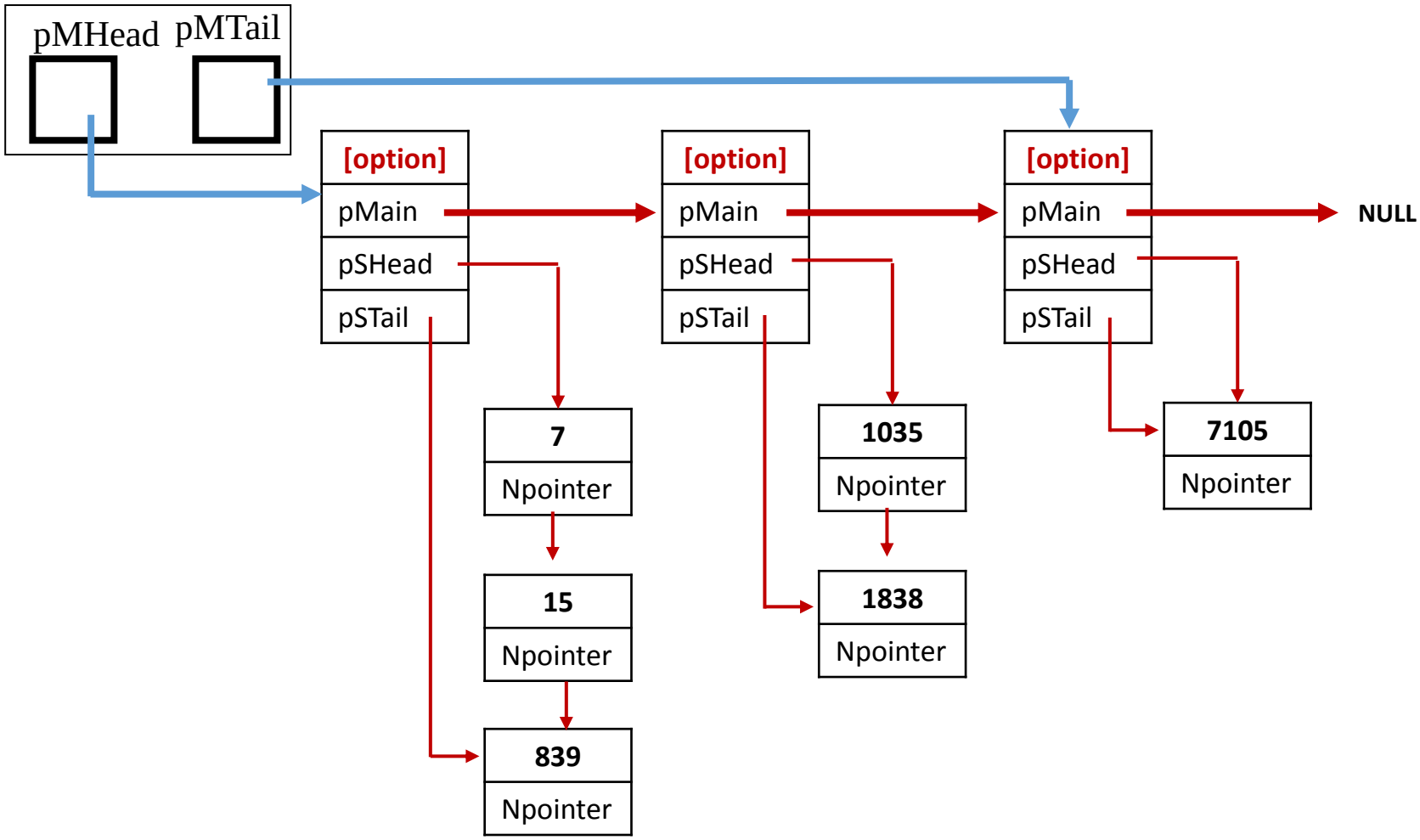
- DSLK cấp 1 (main): mỗi phần tử là 1 cấu trúc gồm:
  - Liên kết: các node liên kết qua 1 pointer.
  - *pHead*, *pTail* để quản lý các DSLK con (Sub).
  - Node thứ 1 sẽ quản lý các số (do người dùng nhập vào) từ 0-999, node thứ 2 sẽ quản lý các số từ 1000-1999, … Khi số nhập vào đã có, chương trình sẽ báo lỗi.
  - Khởi đầu DSLK là rỗng. Node chỉ được tạo ra khi có phát sinh 1 giá trị thuộc phạm vi node quản lý.

- DSLK cấp 2 (sub):
  - *Data*: là số nguyên (do người dùng nhập vào), Khi giá trị được thêm vào danh sách sẽ luôn được sắp xếp tang dần.
  - *Liên kết:* là pointer chỉ đến Node tương ứng.

6. Các dạng phối hợp khác dựa trên DSLK

(SV cài đặt các dạng danh sách liên kết và báo cáo)

ii. Minh họa gợi ý DSLK với nội dung chứa 1 DSLK khác.




Q&A
How
Why
When
Where
What
Who