# GIỚI THIỆU MÔN HỌC

**Tóm tắt nội dung:**

Bài 1: Danh sách liên kết

Bài 2: Một số phương pháp sắp xếp

Bài 3: Hàm băm

Bài 4: Cây, cây nhị phân, cây nhị phân tìm kiếm, cây cân bằng

Bài 5: Cây đỏ đen

Bài 6: B-cây, cây 2-3-4

Bài 7: Các đống nhị thức

Bài 8: Các đống Fibonaci

Bài 9: Các tập rời nhau

Bài 10: Các thuật toán so khớp chuỗi

Tài liệu tham khảo:

1) Data Structures, Algorithms, and Object-Oriented Programming. NXB McGraw Hill; Tác giả Gregory Heilleman -1996

2) Advanced Data Structures. NXB McGraw Hill - *1990*; Tác giả Thomas H. C., Charles E.L., and Ronald L.R.

3) Giáo trình thuật toán. NXB Thống kế 2002. Nhóm Ngọc Anh Thư dịch

4) Algorithms and Data Structures in C++; Tác giả *Alan Parker*

# Bài 1: Danh sách liên kết

## I) Danh sách liên kết đơn

### 1. Tổ chức danh sách đơn

Danh sách liên kết bao gồm các phần tử. Mỗi phần tử của danh sách đơn là một cấu trúc chứa 2 thông tin :

- Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử .
- Thành phần mối liên kết: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Ta có định nghĩa tổng quát

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

Ví dụ : Định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên:

```
typedef     struct  SinhVien //Data
{     char  Ten[30];
      int   MaSV;
}SV;

typedef     struct  SinhvienNode
```

```
{       SV    Info;
        struct SinhvienNode* pNext;
}SVNode;
```

Các phần tử trong danh sách sẽ được cấp phát động. Biết phần tử đầu tiên ta sẽ truy xuất được các phần tử tiếp theo. Thường sử dụng con trỏ Head để lưu trữ địa chỉ đầu tiên của danh sách.

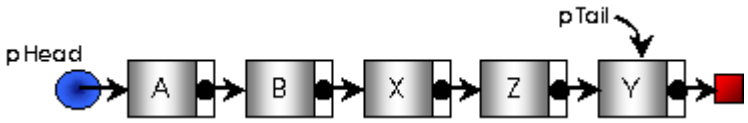
Ta có khai báo:

```
NODE       *pHead;
```

Để quản lý địa chỉ cuối cùng trong danh sách ta dùng con trỏ TAIL. Khai báo như sau:

```
NODE     *pTail;
```

VD:




## II. Các thao tác cơ bản trên danh sách đơn

Giả sử có các định nghĩa:

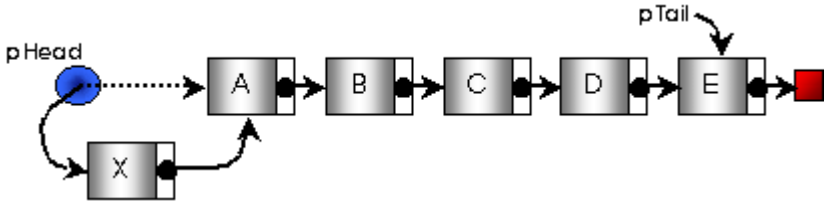
```
typedef      struct  tagNode
{
        Data  Info;
        struct tagNode* pNext;
}NODE;
typedef      struct  tagList
{
```

```
    NODE* pHead;
    NODE* pTail;
}LIST;
```

NODE *new_ele // giữ địa chỉ của một phần tử mới được tạo
Data x; // lưu thông tin về một phần tử sẽ được tạo
LIST lst; // lưu trữ địa chỉ đầu, địa chỉ cuối của danh sách liên kết

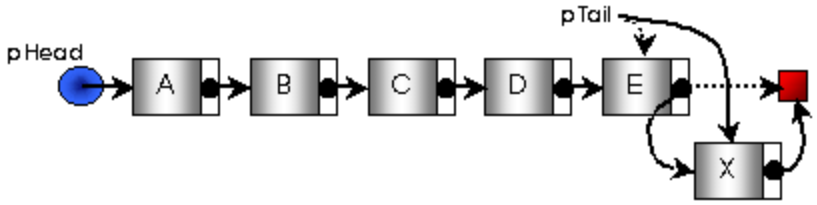
**1.Chèn một phần tử vào danh sách:**

**Có 3 loại thao tác chèn new_ele vào xâu:**
Cách 1: Chèn vào đầu danh sách




Thuật toán :
Bắt đầu:
Nếu Danh sách rỗng Thì
B11 : pHead = new_ele;
B12 : pTail = pHead;
Ngược lại
B21 : new_ele ->pNext = pHead;
B22 : pHead = new_ele ;
Cài đặt:

**Cách 2: Chèn vào cuối danh sách**




Thuật toán :

Bắt đầu :

Nếu Danh sách rỗng thì

B11 : pHead = new_elelment;

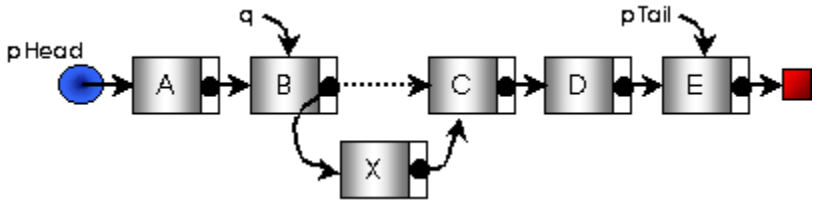
B12 : pTail = pHead;

Ngược lại

B21 : pTail ->pNext = new_ele;

B22 : pTail = new_ele ;

**Cách 3 : Chèn vào danh sách sau một phần tử q**




Thuật toán :

Bắt đầu :

Nếu ( q != NULL) thì

B1 : new_ele -> pNext = q->pNext;

B2 : q->pNext = new_ele ;

Cài đặt :

**2. Tìm một phần tử trong danh sách đơn**

Thuật toán :

Bước 1:

p = pHead; //Cho p trỏ đến  phần tử đầu danh sách

Bước 2:

Trong khi (p != NULL) và (p->Info != k ) thực hiện:

p:=p->pNext;// Cho p trỏ tới phần tử kế

Bước 3:

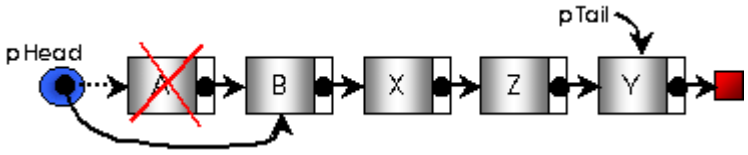
Nếu p != NULL thì p trỏ tới phần tử cần tìm

Ngược lại: không có phần tử cần tìm.

Cài đặt :

## 3. Hủy một phần tử khỏi danh sách

### Hủy phần tử đầu xâu:




Thuật toán :

Bắt đầu:

Nếu  (pHead != NULL)  thì

B1:  p = pHead;  // p là phần tử cần hủy

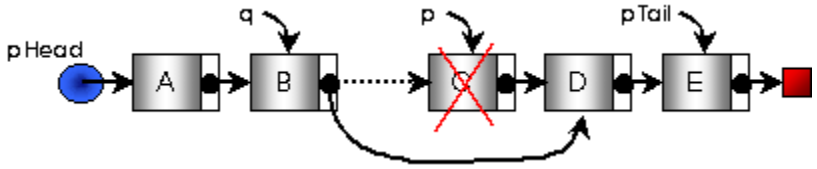
B2:

B21 : pHead = pHead->pNext;  // tách p ra khỏi xâu

B22 :  free(p);  // Hủy biến động do p trỏ đến

B3:  Nếu pHead=NULL thì pTail = NULL; //Xâu rỗng

### Hủy một phần tử đứng sau phần tử q




Thuật toán :

Bắt đầu:

Nếu (q!= NULL) thì

B1: p = q->Next; // p là phần tử cần hủy

B2: Nếu (p != NULL) thì// q không phải là cuối xâu

B21 : q->Next = p->Next; // tách p ra khỏi xâu

B22 : free(p); // Hủy biến động do p trỏ đến

**Hủy 1 phần tử có khoá k**

Thuật toán :

Bước 1:

Tìm phần tử p có khóa k và phần tử q đứng trước nó

Bước 2:

Nếu (p!= NULL) thì // tìm thấy k

Hủy p ra khỏi xâu tương tự hủy phần tử sau q;

Ngược lại

Báo không có k;

## 4. Thăm các nút trên danh sách

- Đếm các phần tử của danh sách,
- Tìm tất cả các phần tử thoả điều kiện,
- Huỷ toàn bộ danh sách (và giải phóng bộ nhớ)

Thuật toán xử lý các nút trên danh sách:

Bước 1:

p = pHead; //Cho p trỏ đến  phần tử đầu danh sách

Bước 2:

Trong khi  (Danh sách chưa hết)  thực hiện

B21 :  Xử lý phần tử p;

B22 :  p:=p->pNext;  // Cho p trỏ tới phần tử kế

Thuật toán hủy toàn bộ danh sách:

Bước 1:

Trong khi  (Danh sách chưa hết)  thực hiện

B11:

p = pHead;

pHead:=pHead->pNext;  // Cho p trỏ tới phần tử kế

B12:

Hủy p;

Bước 2:

Tail  = NULL; //Bảo đảm tính nhất quán khi xâu rỗng

## II. Danh sách liên kết kép

Là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.


**Khai báo:**

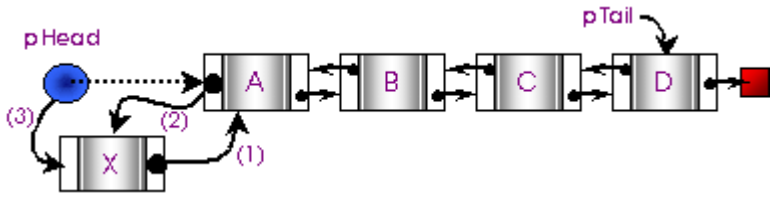
```
typedef      struct  tagDNode
{
Data  Info;
struct tagDNode* pPre;       // trỏ đến phần tử đứng trước
struct tagDNode* pNext;      // trỏ đến phần tử đứng sau
}DNODE;

typedef      struct  tagDList
{
DNODE* pHead;       // trỏ đến phần tử đầu danh sách
DNODE* pTail;// trỏ đến phần tử cuối danh sách
}DLIST;
```

### 1. Chèn một phần tử vào  danh sách:

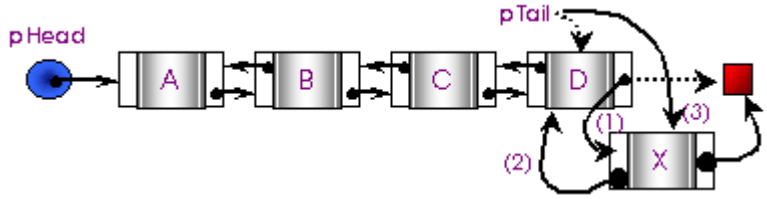
Có 4 loại thao tác chèn  new_ele vào danh sách:

Cách 1: Chèn vào đầu danh sách



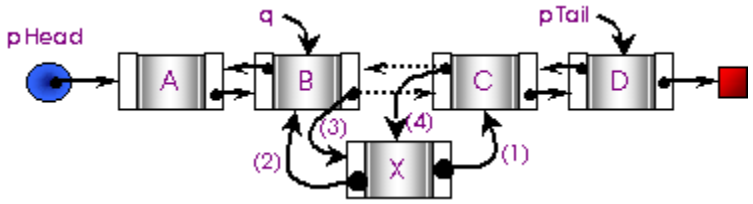
Cài đặt :

Cách 2: Chèn vào cuối danh sách



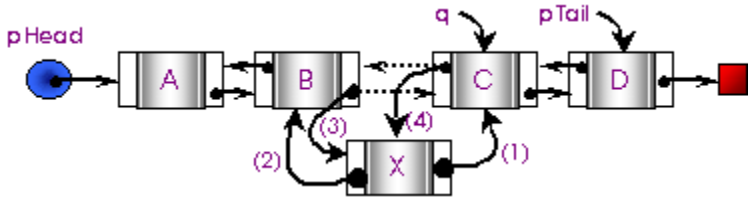

Cài đặt :

Cách 3 : Chèn vào danh sách sau một phần tử q




Cài đặt :

Cách 4 : Chèn vào danh sách trước một phần tử q




Cài đặt :

**2. Hủy một phần tử khỏi danh sách**

- Hủy phần tử đầu xâu
- Hủy phần tử cuối xâu
- Hủy một phần tử đứng sau phần tử q
- Hủy một phần tử đứng trước phần tử q
- Hủy 1 phần tử có khoá k

**3. Xử lý các nút trên danh sách:**

- Tìm nút có khóa k
- Hiển thị giá trị khóa của các nút trong danh sách
- Hủy tòan bộ danh sách

## III. Ngăn xếp (stack)

Stack chứa các đối tượng làm việc theo cơ chế LIFO (Last In First Out) nghĩa là việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "Vào sau ra trước".

Thao tác thêm 1 đối tượng vào stack thường được gọi là "Push".

Thao tác lấy 1 đối tượng ra khỏi stack gọi là "Pop".

Trong tin học, CTDL stack có nhiều ứng dụng: khử đệ qui, lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, ứng dụng trong các bài toán tính toán biểu thức, .

Một hình ảnh một stack

### Các thao tác

Push(o): Thêm đối tượng o vào đầu stack

Pop(): Lấy đối tượng ở đỉnh stack ra khỏi stack và trả về giá trị của nó. Nếu stack rỗng thì lỗi sẽ xảy ra.

isEmpty(): Kiểm tra xem stack có rỗng không.

Top(): Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

**Biểu diễn Stack dùng mảng**

Ta có thể tạo một stack bằng cách khai báo một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000).

VD:


Tạo stack S và quản lý đỉnh stack bằng biến t – chỉ số của phần từ trên cùng trong stack:

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

**Biểu diễn Stack dùng danh sách liên kết đơn**

VD:

```
LIST S;
```

**Các thao tác:**

Tạo Stack S rỗng (S.pHead=l.pTail= NULL sẽ tạo ra một Stack S rỗng)

Kiểm tra stack rỗng: int IsEmpty(LIST &S)

Thêm một phần tử p vào stack S:void Push(LIST &S, Data x)

Trích huỷ phần tử ở đỉnh stack S: Data Pop(LIST &S)

Xem thông tin của phần tử ở đỉnh stack S: Data Top(LIST &S)

**Ứng dụng của Stack:**

Biến đổi biểu thức:

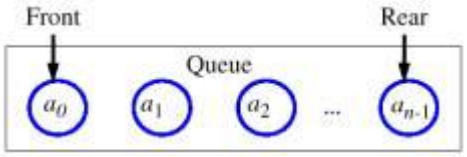
| Dạng trung tố | Dạng hậu tố |
|---|---|
| a+b | ab+ |
| a*b | ab* |

| | |
|---|---|
| a*(b+c)-d/e | abc+*de-/ |

Tính giá trị của biểu thức ở dạng hậu tố.

## IV. Hàng đợi ( Queue)

Hàng đợi chứa các đối tượng làm việc theo cơ chế FIFO (First In First Out) nghĩa là việc thêm một đối tượng vào hàng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được thực hiện theo cơ chế "Vào trước ra trước".




Hàng đội

Các thao tác:

EnQueue(o): Thêm đối tượng o vào cuối hàng đợi

DeQueue(): Lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

IsEmpty(): Kiểm tra xem hàng đợi có rỗng không.

Front(): Trả về giá trị của phần tử nằm ở đầu hàng đợi mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

### Biểu diễn dùng mảng:

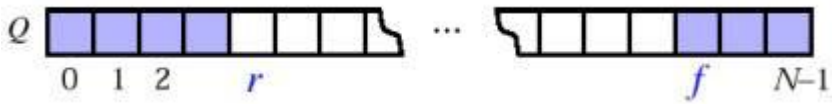
Ta có thể tạo một hàng đợi bằng cách sử dụng một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000) theo kiểu xoay vòng (coi phần tử an-1 kề với phần tử a0).

Ta ký hiệu nó là NULLDATA như ở những phần trước.

Trạng thái hàng đợi lúc bình thường:


Q – biến hàng đợi, f quản lý đầu hàng đợi, r quản lý phần tử cuối hàng đợi.

Trạng thái hàng đợi lúc xoay vòng (mảng rỗng ở giữa):




Câu hỏi đặt ra: khi giá trị f=r cho ta điều gì ? Ta thấy rằng, lúc này hàng đợi chỉ có thể ở một trong hai trạng thái là rỗng hoặc đầy.

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

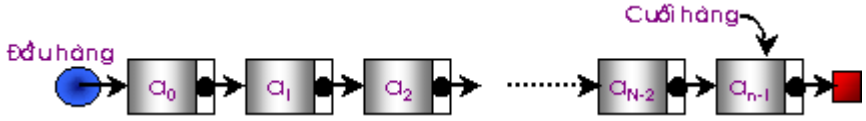
Data Q[N] ;

int f, r;

**Dùng danh sách liên kết**

Ta có thể tạo một hàng đợi bằng cách sử dụng một danh sách liên kết đơn.

LIST Q;




**Các thao tác:**

Tạo hàng đợi rỗng: Lệnh Q.pHead = Q.pTail = NULL sẽ tạo ra một hàng đợi rỗng.

-Kiểm tra hàng đợi rỗng :

int IsEmpty(LIST Q)

- Thêm một phần tử p vào cuối hàng đợi :

void EnQueue(LIST Q, Data x)

- Trích/Hủy phần tử ở đầu hàng đợi:

  Data DeQueue(LIST Q)

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

  Data Front(LIST Q)

**Ứng dụng của hàng đợi**

- Bài toán quản lý tồn kho
- Bài toán xử lý các lệnh trong máy tính điện tử.

**Bài tập:**