Bài giảng môn học

Chương 2: Cấu trúc dữ liệu động

ThS. Nguyen Minh Phuc
Software Technology Department - Faculty of Information Technology – Lac Hong University

Cấu trúc dữ liệu và giải thuật

Mục tiêu

- Giới thiệu khái niệm cấu trúc dữ liệu động.
- Giới thiệu danh sách liên kết:
  - Các kiểu tổ chức dữ liệu theo DSLK.
  - Danh sách liên kết đơn: tổ chức, các thuật toán, ứng dụng.

Kiểu dữ liệu tĩnh

- Khái niệm : Một số đối tượng dữ liệu không thay thay đổi được kích thước, cấu trúc, … trong suốt quá trình sống. Các đối tượng dữ liệu thuộc những kiểu dữ liệu gọi là kiểu dữ liệu tĩnh.
- Một số kiểu dữ liệu tĩnh : các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: kiểu thực, kiểu nguyên, kiểu ký tự ... hoặc từ các cấu trúc đơn giản như mẩu tin, tập hợp, mảng ...

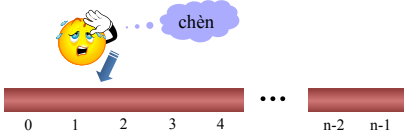
➔ Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này thường cứng ngắt, gò bó ➔ khó diễn tả được thực tế vốn sinh động, phong phú.

CTDL tĩnh – Một số hạn chế

- Một số đối tượng dữ liệu trong chu kỳ sống của nó có thể thay đổi về cấu trúc, độ lớn, như danh sách các học viên trong một lớp học có thể tăng thêm, giảm đi ... Nếu dùng những cấu trúc dữ liệu tĩnh đã biết như mảng để biểu diễn ➔ Những thao tác phức tạp, kém tự nhiên ➔ chương trình khó đọc, khó bảo trì và nhất là khó có thể sử dụng bộ nhớ một cách có hiệu quả.
- Dữ liệu tĩnh sẽ chiếm vùng nhớ đã dành cho chúng suốt quá trình hoạt động của chương trình ➔ sử dụng bộ nhớ kém hiệu quả.

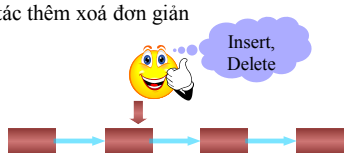
CTDL tĩnh

- Mảng 1 chiều
  - Kích thước cố định (fixed size)
  - Chèn 1 phần tử vào mảng rất khó
  - Các phần tử tuần tự theo chỉ số 0 ⇒ n-1
  - Truy cập ngẫu nhiên (random access)



Cấu trúc dữ liệu động

- Danh sách liên kết
  - Cấp phát động lúc chạy chương trình
  - Các phần tử nằm rải rác ở nhiều nơi trong bộ nhớ
  - Kích thước danh sách chỉ bị giới hạn do RAM
  - Thao tác thêm xoá đơn giản



Hướng giải quyết

- Cần xây dựng cấu trúc dữ liệu đáp ứng được các yêu cầu:
  - Linh động hơn.
  - Có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống.

➔ *Cấu trúc dữ liệu động.*

Biến không động

Biến không động (biến tĩnh, biến nửa tĩnh) là những biến thỏa:

- Được khai báo tường minh,
- Tồn tại khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này,
- Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc là Stack (đối với biến nửa tĩnh - các biến cục bộ).
- Kích thước không thay đổi trong suốt quá trình sống.
- Do được khai báo tường minh, các biến không động có một định danh đã được kết nối với địa chỉ vùng nhớ lưu trữ biến và được truy xuất trực tiếp thông qua định danh đó.
- Ví dụ :

```
int  a;     // a, b là các biến không động
char b[10];
```


Biến động

- Trong nhiều trường hợp, tại thời điểm biên dịch không thể xác định trước kích thước chính xác của một số đối tượng dữ liệu do sự tồn tại và tăng trưởng của chúng phụ thuộc vào ngữ cảnh của việc thực hiện chương trình.
- Các đối tượng dữ liệu có đặc điểm kể trên nên được khai báo như biến động. Biến động là những biến thỏa:
  - Biến không được khai báo tường minh.
  - Có thể được cấp phát hoặc giải phóng bộ nhớ khi người sử dụng yêu cầu.
  - Các biến này không theo qui tắc phạm vi (tĩnh).
  - Vùng nhớ của biến được cấp phát trong Heap.
  - Kích thước có thể thay đổi trong quá trình sống.

Biến động

- Do không được khai báo tường minh nên các biến động không có một định danh được kết buộc với địa chỉ vùng nhớ cấp phát cho nó, do đó gặp khó khăn khi truy xuất đến một biến động.
- Để giải quyết vấn đề, biến con trỏ (là biến không động) được sử dụng để trỏ đến biến động.
- Khi tạo ra một biến động, phải dùng một con trỏ để lưu địa chỉ của biến này và sau đó, truy xuất đến biến động thông qua biến con trỏ đã biết định danh.

Biến động

- Hai thao tác cơ bản trên biến động là tạo và hủy một biến động do biến con trỏ ‘p’ trỏ đến:
  - Tạo ra một biến động và cho con trỏ ‘p’ chỉ đến nó
  - Hủy một biến động do p chỉ đến

Biến động

- Tạo ra một biến động và cho con trỏ 'p' chỉ đến nó

```
void* malloc(size);    // trả về con trỏ chỉ đến vùng nhớ
                       // size byte vừa được cấp phát.
void* calloc(n,size);// trả về con trỏ chỉ đến vùng nhớ
                       // vừa được cấp phát gồm n phần tử,
                       // mỗi phần tử có kích thước size byte
new   // toán tử cấp phát bộ nhớ trong C++
```

- Hàm free(p) huỷ vùng nhớ cấp phát bởi hàm malloc hoặc calloc do p trỏ tới
- Toán tử delete p huỷ vùng nhớ cấp phát bởi toán tử new do p trỏ tới

Biến động – *Ví dụ*

```
int *p1, *p2;
// cấp phát vùng nhớ cho 1 biến động kiểu int
p1 = (int*)malloc(sizeof(int));
*p1 = 5;      // đặt giá trị 5 cho biến động đang được p1
  quản lý
// cấp phát biến động kiểu mảng gồm 10 phần tử kiểu int
p2 = (int*)calloc(10, sizeof(int));
*(p2+3) = 0; // đặt giá trị 0 cho phần tử thứ 4 của
  mảng p2
free(p1);
free(p2);
```


Kiểu dữ liệu Con trỏ

- Kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng dữ liệu khác.
- Biến thuộc kiểu con trỏ Tp là biến mà giá trị của nó là địa chỉ cuả một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

Con trỏ – Khai báo

- Cú pháp định nghĩa một kiểu con trỏ trong ngôn ngữ C :
  typedef <kiểu cơ sở> * < kiểu con trỏ>;
- Ví dụ :

```
typedef    int    *intpointer;
intpointer p;
```

hoặc

```
int    *p;
```

là những khai báo hợp lệ.

Con trỏ – *Thao tác căn bản*

- Các thao tác cơ bản trên kiểu con trỏ:(minh họa bằng C)
  - Khi 1 biến con trỏ p lưu địa chỉ của đối tượng x, ta nói 'p trỏ đến x'.
  - Gán địa chỉ của một vùng nhớ con trỏ p:

```
p = <địa chỉ>;
```

ví dụ :

```
int  i,*p;
p=&i;
```

  - Truy xuất nội dung của đối tượng do p trỏ đến (*p)

Danh sách liên kết (List)

I. Định nghĩa:

Một danh sách (list) là một tập hợp gồm một số hữu hạn phần tử cùng kiểu, có thứ tự.

Có hai cách cài đặt danh sách là :

- Cài đặt theo kiểu kế tiếp : ta có danh sách kề hay danh sách đặc.
- Cài đặt theo kiểu liên kết : ta có danh sách liên kết.

Danh sách liên kết (List)

a. Danh sách kề :

- Các phần tử của danh sách gọi là các node, được lưu trữ kề liền nhau trong bộ nhớ. Mỗi node có thể là một giá trị kiểu int, float, char, … hoặc có thể là một struct với nhiều vùng tin. Mảng hay chuỗi là dạng của danh sách kề.
- Địa chỉ của mỗi node trong danh sách được xác định bằng chỉ số (index). Chỉ số của danh sách là một số nguyên và được đánh từ 0 đến một giá trị tối đa nào đó.
- Danh sách kề là cấu trúc dữ liệu tĩnh, số node tối đa của danh sách kề cố định sau khi cấp phát nên số node cần dùng có khi thừa hay thiếu. Ngoài ra danh sách kề không phù hợp với các thao tác thường xuyên như thêm hay xóa phần tử trên danh sách,

Danh sách liên kết (List)

b. Danh sách liên kết :

- Các phần tử của danh sách gọi là node, nằm rải rác trong bộ nhớ. Mỗi node, ngoài vùng dữ liệu thông thường, còn có vùng liên kết chứa địa chỉ của node kế tiếp hay node trước nó.
- Danh sách liên kết là cấu trúc dữ liệu động, có thể thêm hay hủy node của danh sách trong khi chạy chương trình. Với cách cài đặt các thao tác thêm hay hủy node ta chỉ cần thay đổi lại vùng liên kết cho phù hợp.
- Tuy nhiên, việc lưu trữ danh sách liên kết tốn bộ nhớ hơn danh sách kề vì mỗi node của danh sách phải chứa thêm vùng liên kết. Ngoài ra việc truy xuất node thứ i trong danh sách liên kết chậm hơn vì phải duyệt từ đầu danh sách.

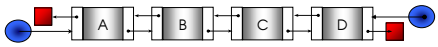
Danh sách liên kết (List)

- Có nhiều kiểu tổ chức liên kết giữa các phần tử trong danh sách như :
  - Danh sách liên kết đơn
  - Danh sách liên kết kép
  - Danh sách liên kết vòng
  - …

Danh sách liên kết (List)

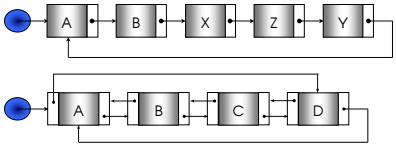
- **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:

- **Danh sách liên kết kép:** mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:



Danh sách liên kết (List)

- **Danh sách liên kết vòng :** phần tử cuối danh sách liên kết với phần tử đầu danh sách:



Danh sách liên kết đơn (xâu đơn)

DSLK - định nghĩa

- DSLK đơn là chuỗi các node, được tổ chức theo thứ tự tuyến tính
- Mỗi node gồm 2 phần:
  - Phần Data, information : lưu trữ các thông tin về bản thân phần tử .
  - Phần link hay con trỏ : 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.

Node

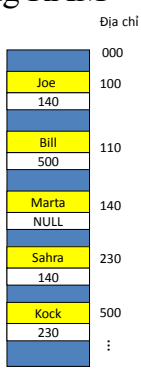
Cấu trúc dữ liệu của DSLK đơn

```
typedef   struct Node
 {
        int     data;  // Data là kiểu đã định nghĩa trước
        Node * link; //con trỏ chỉ đến cấu trúc NODE
 };
```


Lưu trữ DSLK đơn trong RAM

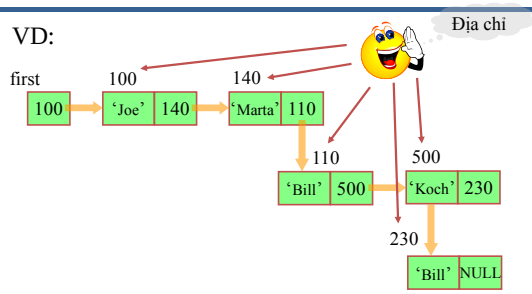
Ví dụ : Ta có danh sách theo dạng bảng sau

Address	Name	Age	Link
100	Joe	20	140
110	Bill	42	500
140	Marta	27	110
230	Sahra	25	NULL
...	...	...	
500	Koch	31	230



DSLK đơn truy xuất – Minh hoạ

- VD:

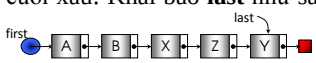


Tổ chức, quản lý

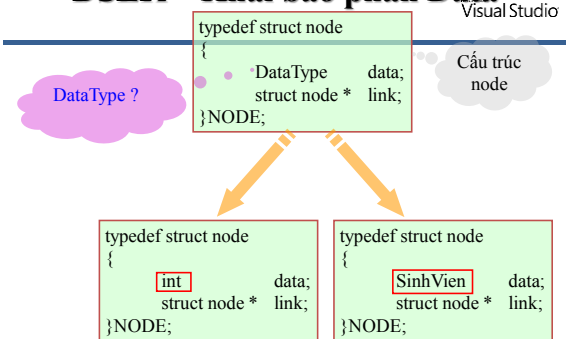
- Để quản lý một xâu đơn chỉ cần biết địa chỉ phần tử đầu xâu.
- Con trỏ **First** sẽ được dùng để lưu trữ địa chỉ phần tử đầu xâu, ta gọi **First** là đầu xâu. Ta có khai báo :

  NODE* first;
- Để tiện lợi, có thể sử dụng thêm một con trỏ **last** giữ địa chỉ phần tử cuối xâu. Khai báo **last** như sau :

  NODE* last;



DSLK – Khai báo phần Data



Tổ chức, quản lý

Khai báo kiểu của 1 phần tử và kiểu danh sách liên kết đơn và để đơn giản ta xét mỗi node gồm vùng chứa dữ liệu là kiểu số nguyên :

```
// kiểu của một phần tử trong danh sách
typedef   struct Node
{
  int     data;
  NODE *  link;
}NODE;

typedef   struct List   // kiểu danh sách liên kết
{
  NODE* first;
  NODE* last;
}LIST;
Trong thực tế biến data là một kiểu struct
```


Tạo một phần tử

- Thủ tục **GetNode** để tạo ra một phần tử cho danh sách với thông tin chứa trong x

```
NODE*   GetNode(int x)
{
    NODE *p;
    // Cấp phát vùng nhớ cho phần tử
    p = (NODE*)malloc(sizeof(NODE))//p= new NODE;
    if (p==NULL)
    {
        cout<<"Khong du bo nho!"; exit(1);
    }
    p->data = x; // Gán thông tin cho phần tử p
    p->link = NULL;
    return p;
}
```


Tạo một phần tử

Để tạo một phần tử mới cho danh sách, cần thực hiện câu lệnh:

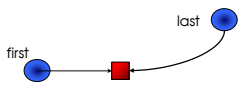
new_ele = GetNode(x);

→ new_ele sẽ quản lý địa chỉ của phần tử mới được tạo.

Các thao tác cơ sở

- Tạo danh sách rỗng
- Thêm một phần tử vào danh sách
- Tìm kiếm một giá trị trên danh sách
- Trích một phần tử ra khỏi danh sách
- Duyệt danh sách
- Hủy toàn bộ danh sách

Khởi tạo danh sách rỗng



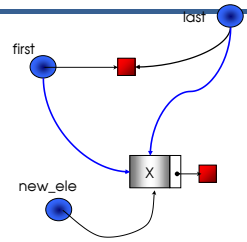

```
void Init(LIST &l)
{
  l.first = l.last = NULL;
}
```


Thêm một phần tử

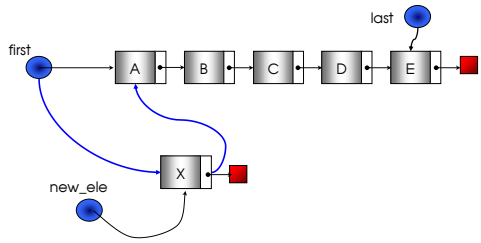
Có 3 vị trí thêm phần tử mới vào danh sách :

- Thêm vào đầu danh sách
- Nối vào cuối danh sách
- Chèn vào danh sách sau một phần tử q

Thêm một phần tử



Thêm một phần tử vào đầu



Thêm một phần tử vào đầu

//input: danh sách (first, last), phần tử mới new_ele

//output: danh sách (first, last) với new_ele ở đầu DS

- Nếu Danh sách rỗng Thì
  - first = new_ele;
  - last = first;
- Ngược lại
  - new_ele ->link = first;
  - first = new_ele ;

Thêm một phần tử vào đầu

```
void AddFirst(LIST &l, NODE* new_ele)
{
    if (l.first == NULL)   //Xâu rỗng
    {
        l.first = new_ele;
        l.last  = l.first;
    }
    else
    {
        new_ele->link = l.first;
        l.first = new_ele;
    }
}
```


Thêm một thành phần dữ liệu vào đầu

//input: danh sách (first, last), thành phần dữ liệu X

//output: danh sách (first, last) với phần tử chứa X ở đầu DS

- Tạo phần tử mới new_ele để chứa dữ liệu X
- Nếu tạo được:

  Thêm new_ele vào đầu danh sách
- Ngược lại

  Báo lỗi

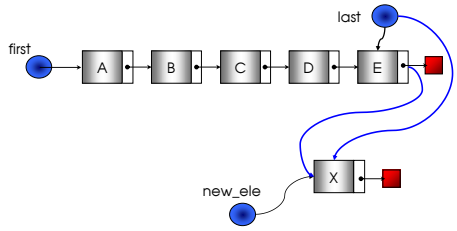
Thêm một thành phần dữ liệu vào đầu

```
NODE* Insertfirst(LIST &l, int x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele == NULL)
        return NULL;
    AddFirst(l, new_ele);
    return new_ele;
}
```

Tạo Link list bằng cách thêm vào đầu

```
void create_list_first(LIST &l, int x)
{
    NODE *p;
    int n;
    cout<<"Nhap so phan tu:";
    cin>>n;
    for (int i=1;i<=n;i++)   // nen dung vong lap do while de viet
        {
          //Insertfirst(l,x);
          p=GetNode(x);
          AddFirst(l,p);
        }
}
```


Thêm một phần tử vào cuối



Thêm một phần tử vào cuối

//input: danh sách (first, last), phần tử mới new_ele

//output: danh sách (first, last) với new_ele ở cuối DS

- Nếu Danh sách rỗng Thì
  - first = new_ele;
  - last = first;
- Ngược lại
  - last->link = new_ele ;
  - last = new_ele ;

Thêm một phần tử vào cuối

```
void InsertLast(LIST &l, NODE *new_ele)
{
    if (l.first==NULL)
    {
        l.first = new_ele;
        l.last = l.first;
    }
    else
    {
        l.last->link = new_ele;
        l.last = new_ele ;
    }
}
```


Thêm một thành phần dữ liệu vào cuối

//input: danh sách (first, last), thành phần dữ liệu X

//output: danh sách (first, last) với phần tử chứa X ở cuối DS

- Tạo phần tử mới new_ele để chứa dữ liệu X
- Nếu tạo được:

  Thêm new_ele vào cuối danh sách
- Ngược lại

  Báo lỗi

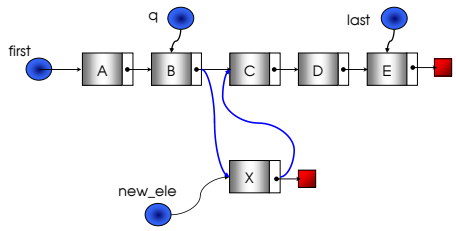
Thêm một thành phần dữ liệu vào cuối

```
NODE* InputLast(LIST &l)
{   int x,n;
    printf("Nhap so phan tu :");
    scanf("%d",&n);
    for(int i=0;i<n;i++)
    {
        printf("Nhap phan tu thu %d :",i);
        scanf("%d",&x);
        NODE* p=GetNode(x);
        InsertLast(l, p);
    }
}
```

Tạo Link list bằng cách thêm vào cuối

```
void create_list_last(list &l, int x)
{
  node *p;
  int n;
  cout<<"Nhap so phan tu:";
  cin>>n;
  for (int i=1;i<=n;i++)   // nen dung vong lap do while de viet
     {
     p=GetNode(x);
     InsertLast(l,p);
     }
}
```


Chèn một phần tử sau q



Chèn một phần tử sau q

//input: danh sách (first, last), q, phần tử mới new_ele
//output: danh sách (first, last) với new_ele ở sau q

Nếu (q != NULL) thì
  new_ele -> link = q -> link;
  q -> link = new_ele ;
  Nếu (q ==last) thì
    last = new_ele;
Ngược lại
  Thêm new_ele vào đầu danh sách

Chèn một phần tử sau q

```
void InsertAfter(LIST &l,NODE *q, NODE* new_ele)
{
        if (q!=NULL && new_ele !=NULL)
        {
             new_ele->link = q->link;
             q->link = new_ele;
             if(q == l.last)
                  l.last = new_ele;
        }
}
```


Duyệt danh sách

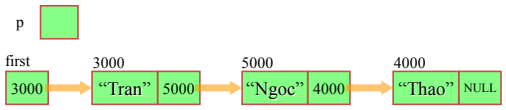
- Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:
  - Đếm các phần tử của danh sách,
  - Tìm tất cả các phần tử thoả điều kiện,
  - Hủy toàn bộ danh sách (và giải phóng bộ nhớ)

Duyệt danh sách

- Bước 1: p = first; *//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->link; *// Cho p trỏ tới phần tử kế*

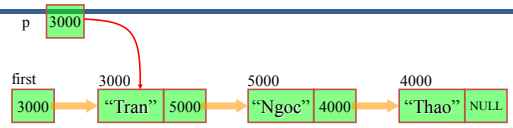
```
void ProcessList (LIST &l)
{
   NODE   *p;
   p = l.first;
   while (p!= NULL){
   ProcessNode(p); // xử lý cụ thể tùy ứng dụng
   p = p->link;
   }
}
```

DSLK – Minh họa in danh sách



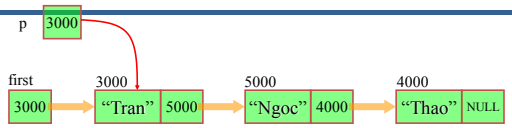

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



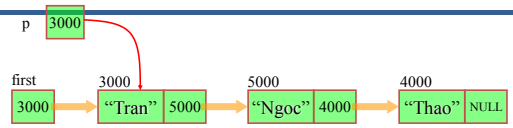

```
p =first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



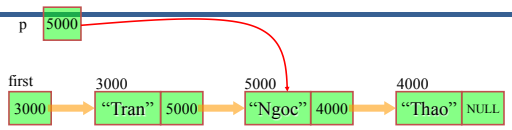

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



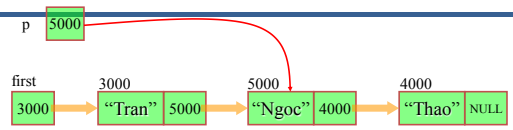

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



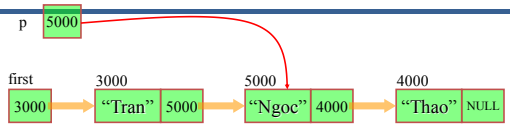

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



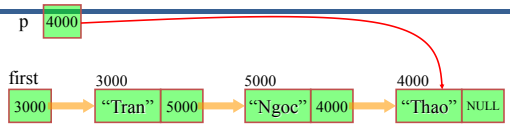

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```

DSLK – Minh họa in danh sách



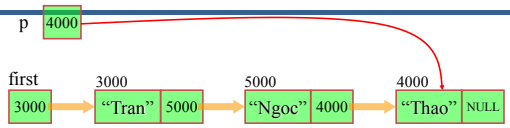

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



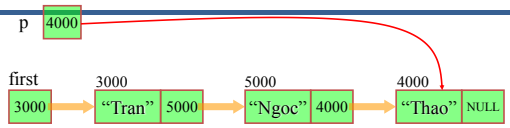

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



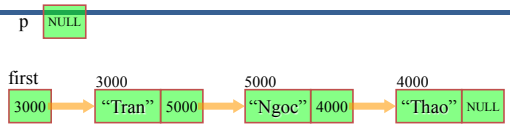

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách



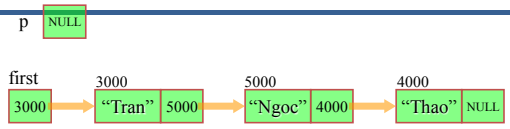

```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách




```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```


DSLK – Minh họa in danh sách




```
p = first;
        while (p!=NULL)
        {
                printf("%d\t",p->data);
                p = p->link;
        }
```

In các phần tử trong danh sách

```
void Output(LIST l)
{
        NODE* p=l.first;
        while(p!=NULL)
        {
                cout<<p->data<<"\t";
                p=p ->link;
        }
        cout<<endl;
}
```


Tìm kiếm một phần tử có khóa x

```
NODE* Search(LIST  l , int  x)
{
        NODE* p=l.first;
        while( p!=NULL && p->data!=x)
                p=p->link;
        return p;
}
```

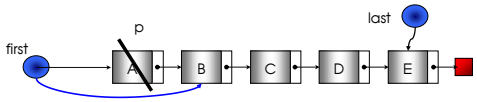

Xóa một node của danh sách

Xóa node đầu của danh sách

Để xóa node đầu danh sách l (khác rỗng)

- Gọi p là node đầu của danh sách (là l.first)
- Cho l.first trỏ vào node sau node p (là p->link)
- Nếu danh sách trở tahnh2 rỗng thì l.last=NULL
- Giải phóng vùng nhớ mà p trỏ tới

Xóa một node của danh sách



Xóa một node của danh sách

```
void  RemoveFirst(LIST &l)
{
   if(l.first != NULL)
   {
        NODE* p=l.first;
        l.first=p->link;
        if(l.first == NULL) l.last=NULL;//Nếu danh sách
   rỗng
        free(p);
   }
}
```


Xóa một node của danh sách

Xóa node sau node q trong danh sách

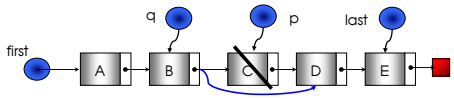
Điều kiện để có thể xóa được node sau q là :

- q phải khác NULL
- Node sau q phải khác NULL

Có 3 thao tác

- Gọi p là node sau q
- Cho vùng link của q trỏ vào node đứng sau p (là l.link)
- Nếu p là phần tử cuối thì l.last trỏ vào q
- Giải phóng vùng nhớ mà p trỏ tới

Xóa node sau node q trong danh sách



Xóa node sau node q trong danh sách

```
void RemoveAfter(LIST &l, NODE *q )
{
  if(q !=NULL && q->link !=NULL)
  {
      NODE*   p = q->link;
      q->link = p->link;
      if(p==l.last)   l.last=q;
      free(p);
  }
}
```


Hủy toàn bộ danh sách

- Để hủy toàn bộ danh sách, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan:
- Bước 1: Trong khi (Danh sách chưa hết) thực hiện
  - B11:
    - p = first;
    - first =first->link; *// Cho p trỏ tới phần tử kế*
  - B12:
    - Hủy p;
- Bước 2:
  - last = NULL;*//Bảo đảm tính nhất quán khi xâu rỗng*

Hủy toàn bộ danh sách

```
void RemoveList(LIST &l)
{
  NODE  *p;
  while (l.first!= NULL) {
      p = l.first;
      l.first = p->link;
      free( p);
  }
  l.last = NULL;
}
```


Sắp xếp trên danh sách liên kết đơn

Sắp xếp danh sách

Cách tiếp cận:

- **Phương án 1:**
  Hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng data).
- **Phương án 2 :**
  Thay đổi các mối liên kết (thao tác trên vùng link)

Sắp xếp danh sách

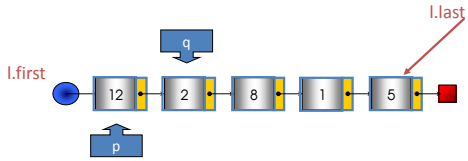
Hoán vị nội dung các phần tử trong danh sách

- Cài đặt lại trên danh sách liên kết một trong những thuật toán sắp xếp đã biết trên mảng
- Điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên danh sách liên kết thông qua liên kết thay vì chỉ số như trên mảng.
- Do thực hiện hoán vị nội dung của các phần tử nên đòi hỏi sử dụng thêm vùng nhớ trung gian ⇒ chỉ thích hợp với các xâu có các phần tử có thành phần data kích thước nhỏ.
- Khi kích thước của trường data lớn, việc hoán vị giá trị của hai phần tử sẽ chiếm chi phí đáng kể.

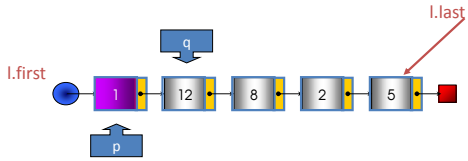
Sắp xếp bằng phương pháp đổi chổ trực tiếp (*Interchange Sort*)

```
void   SLL_InterChangeSort ( List  &l )
{
   for ( Node* p=l.first ; p!=l.last ; p=p->link )
        for ( Node* q=p->link ; q!=NULL ; q=q->link )
              if ( p->data > q->data )
                    Swap( p->data , q->data );
}
```

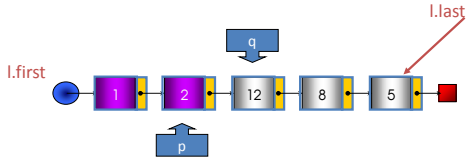

Sắp xếp đổi chổ trực tiếp (*Interchange Sort*)



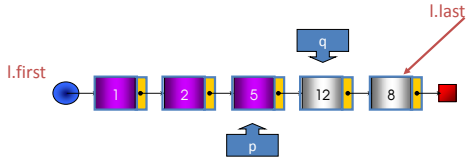
Sắp xếp đổi chổ trực tiếp (*Interchange Sort*)



Sắp xếp đổi chổ trực tiếp (*Interchange Sort*)



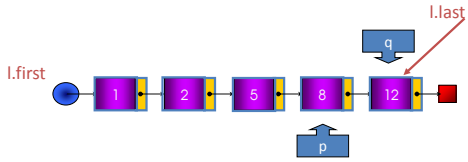
Sắp xếp đổi chổ trực tiếp (*Interchange Sort*)



Sắp xếp đổi chỗ trực tiếp (*Interchange Sort*)

Visual Studio

Dừng



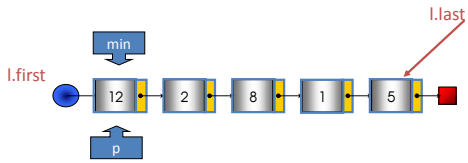
Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

Visual Studio

```
void   ListSelectionSort (LIST &l)
{
   for ( Node* p = l.first ; p != l.last ; p = p->link )
   {
      Node* min = p;
      for ( Node* q = p->link ; q != NULL ; q = q->link )
            if ( min->data > q->data ) min = q ;
                  Swap(min->data, p->data);
   }
}
```

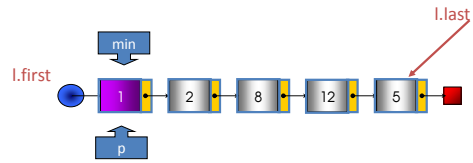

Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

Visual Studio



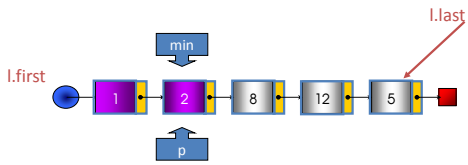
Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

Visual Studio



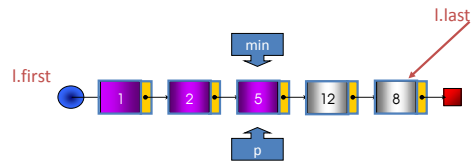
Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

Visual Studio



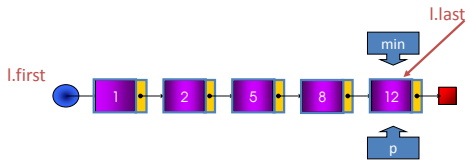
Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

Visual Studio



Sắp xếp bằng phương pháp chọn trực tiếp (*Selection sort*)

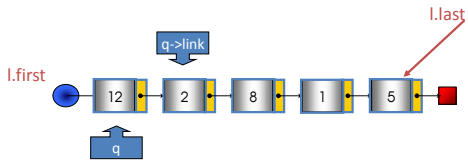
Dừng



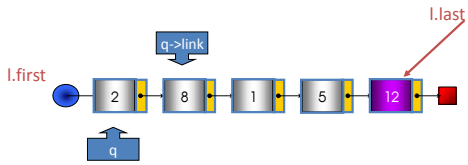
Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)

```
void  SLL_BubleSort ( List  l )
{
   Node*  t = l.last ;
   for ( Node* p = l.first ;  p != NULL ; p = p->link)
   {    Node*  t1;
        for ( Node*  q=l.first ; p!=t ; q=q->link )
        {
                if( q->data  > q->link->data )
                        Swap( q->data , q->link->data );
                t1 = q ;
        }
        t = t1;
   }
}
```

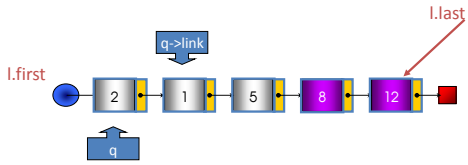

Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)



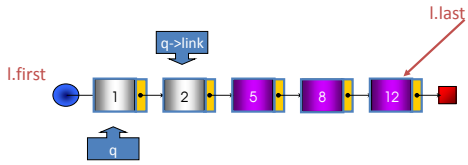
Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)



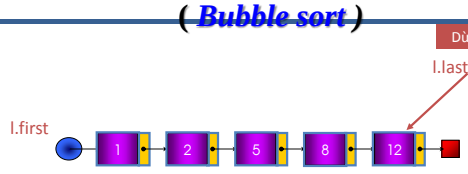
Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)



Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)



Sắp xếp bằng phương pháp nổi bọt (*Bubble sort*)

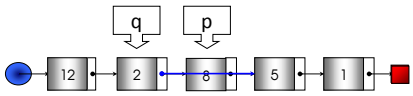


Sắp xếp Thay đổi các mối liên kết

- Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn ⇒ chỉ thao tác trên các móc nối (link).
- Kích thước của trường link:
  - Không phụ thuộc vào bản chất dữ liệu lưu trong xâu
  - Bằng kích thước 1 con trỏ (2 hoặc 4 byte trong môi trường 16 bit, 4 hoặc 8 byte trong môi trường 32 bit…)
- Thao tác trên các móc nối thường phức tạp hơn thao tác trực tiếp trên dữ liệu.

⇒Cần cân nhắc khi chọn cách tiếp cận: Nếu dữ liệu không quá lớn thì nên chọn phương án 1 hoặc một thuật toán hiệu quả nào đó.

Phương pháp lấy Node ra khỏi danh sách giữ nguyên địa chỉ của Node



1 . q->link = p->link ; // *p->link chứa địa chỉ sau p*

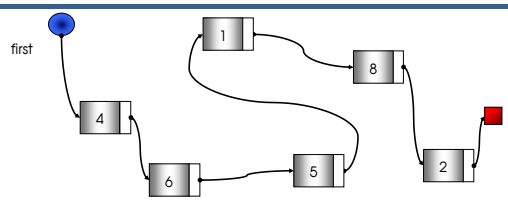
2 . q->link = NULL ; // *p không liên kết phần tử Node*

Quick Sort : Thuật toán

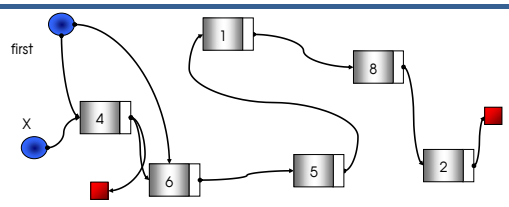
//input: xâu (first, last)
//output: xâu đã được sắp tăng dần

- Bước 1: Nếu xâu có ít hơn 2 phần tử
  Dừng; *//xâu đã có thứ tự*
- Bước 2: Chọn X là phần tử đầu xâu L làm ngưỡng. Trích X ra khỏi L.
- Bước 3: Tách xâu L ra làm 2 xâu L_1 (gồm các phần tử nhỏ hơn hay bằng X) và L_2 (gồm các phần tử lớn hơn X).
- Bước 4: Sắp xếp Quick Sort (L_1).
- Bước 5: Sắp xếp Quick Sort (L_2).
- Bước 6: Nối L_1, X, và L_2 lại theo trình tự ta có xâu L đã được sắp xếp.

Sắp xếp quick sort

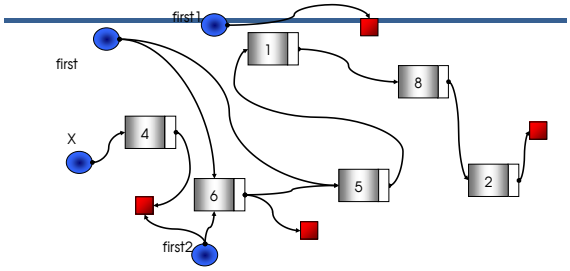


Quick sort : phân hoạch



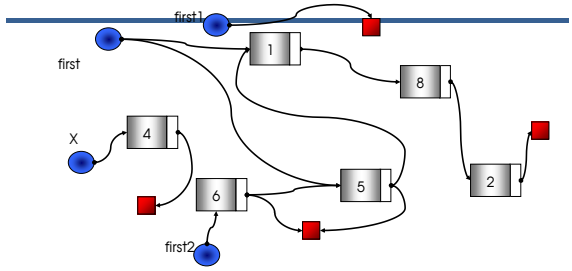
Chọn phần tử đầu xâu làm ngưỡng

Quick sort : phân hoạch



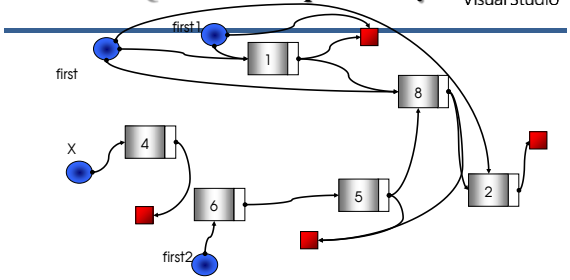
Tách xâu hiện hành thành 2 xâu

Quick sort : phân hoạch



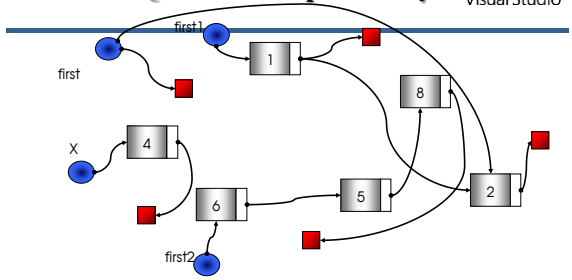
Tách xâu hiện hành thành 2 xâu

Quick sort : phân hoạch



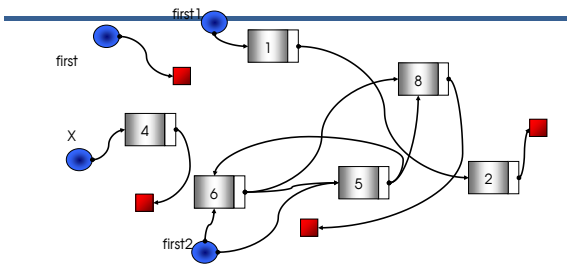
Tách xâu hiện hành thành 2 xâu

Quick sort : phân hoạch



Tách xâu hiện hành thành 2 xâu

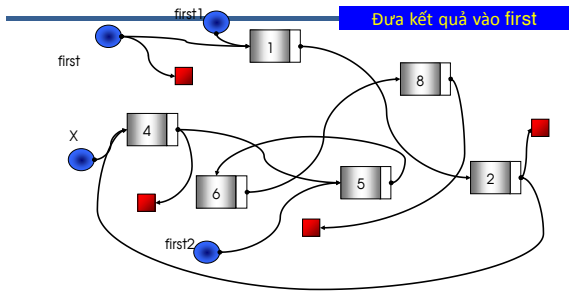
Quick sort



Sắp xếp các xâu l1, l2

Quick sort

Đưa kết quả vào first



Nối 2 danh sách

```
void SListAppend(SLIST &l, LIST &l2)
{
  if (l2.first == NULL) return;
  if (l.first == NULL)
     l = l2;
  else {
     l.first->link = l2.first;
     l.last = l2.last;
  }
  Init(l2);
}
```



```
void SListQSort(SLIST &l) {
   NODE *X, *p;
   SLIST l1, l2;
   if (list.first == list.last) return;
   Init(l1);     Init(l2);
   X = l.first; l.first=x->link;
   while (l.first != NULL) {
      p = l.first;
      if (p->data <= X->data) AddFirst(l1, p);
      else       AddFirst(l2, p);
   }
   SListQSort(l1);     SListQSort(l2);
   SListAppend(l, l1);
   AddFirst(l, X);
   SListAppend(l, l2);
}
```


Quick sort : nhận xét

Nhận xét:

- Quick sort trên xâu đơn đơn giản hơn phiên bản của nó trên mảng một chiều
- Khi dùng quick sort sắp xếp một xâu đơn, chỉ có một chọn lựa phần tử cầm canh duy nhất hợp lý là phần tử đầu xâu. Chọn bất kỳ phần tử nào khác cũng làm tăng chi phí một cách không cần thiết do cấu trúc tự nhiên của xâu.

Danh sách hạn chế

- Stack (Chồng)
- Hàng đợi (Queue)

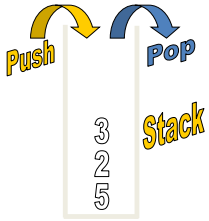
Stack (Chồng)

Stack (Chồng)

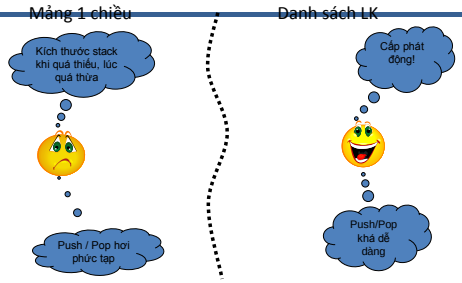
- Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế LIFO (*Last In First Out*) ⇒ Việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "Vào *sau ra trước*".
- Các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.
- "Push": Thao tác thêm 1 đối tượng vào stack
- "Pop": Thao tác lấy 1 đối tượng ra khỏi stack.
- Stack có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, ứng dụng trong các bài toán tính toán biểu thức, …

Giới thiệu

- LIFO: Last In First Out
- Thao tác Pop, Push chỉ diễn ra ở 1 đầu



Hiện thực stack

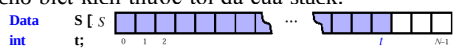


Stack (Chồng)

- Stack là một CTDL trừu tượng (ADT) tuyến tính hỗ trợ 2 thao tác chính:
  - Push(o): Thêm đối tượng o vào đầu stack
  - Pop(): Lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó. Nếu stack rỗng thì lỗi sẽ xảy ra.
- Stack cũng hỗ trợ một số thao tác khác:
  - Empty(): 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

- Có thể tạo một stack bằng cách khai báo một mảng 1 chiều với kích thước tối đa là N (ví dụ: N =1000).
- Stack có thể chứa tối đa N phần tử đánh số từ 0 đến N-1.
- Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử)
- Để khai báo một stack, ta cần một mảng 1 chiều S, biến nguyên t cho biết chỉ số của đầu stack và hằng số N cho biết kích thước tối đa của stack.



Biểu diễn Stack dùng mảng

- Lệnh t = 0 sẽ tạo ra một stack S rỗng.
- Giá trị của t sẽ cho biết số phần tử hiện hành có trong stack.
- Khi cài đặt bằng mảng 1 chiều, stack có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho stack:
  - Full(): Kiểm tra xem stack có đầy chưa.
  - Khi stack đầy, việc gọi đến hàm push() sẽ phát sinh ra lỗi.

Khai báo stack

```
typedef struct node
{
        int data;
};

typedef struct stack
{
        int top;
        node list[N];
};
```

Biểu diễn Stack dùng mảng

```
void Init(stack &s)
{
   s.top=UNDERFLOW;//nhaän giaù trò -1
}
```


Biểu diễn Stack dùng mảng

- Kiểm tra stack rỗng hay không

```
int Empty(stack s)
{
  return s.top=UNDERFLOW ? 1 : 0;
}
```

- Kiểm tra stack đầy hay không

```
int Full(stack s)
{
   return s.top = N-1 ? 1 : 0;
}
```


Biểu diễn Stack dùng mảng

- Thêm một phần tử x vào stack S

```
void Push(stack &s,node x)
{
   if(!Full(s)) // stack chưa đầy
         s.list[++s.top]=x;
}
```

- Trích thông tin và huỷ phần tử ở đỉnh stack S

```
node Pop(stack &s)
{
  if(!Empty(s)) // stack khác rỗng
   return s.list[s.top--];
}
```


Biểu diễn Stack dùng mảng

Nhận xét:

- Các thao tác trên đều làm việc với chi phí O(1).
- Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả.
- Tuy nhiên, hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack N. Giá trị của N có thể quá nhỏ so với nhu cầu thực tế hoặc quá lớn sẽ làm lãng phí bộ nhớ.

Biểu diễn Stack dùng danh sách liên kết

- Có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn (DSLK).
- DSLK có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác trên stack đều diễn ra ở đầu stack.

```
stack *s;
```


Khai báo stack

```
typedef struct node
{
        int data;
        node *link;
};
typedef struct stack
{
        node *top;
};
```

Biểu diễn Stack dùng danh sách liên kết

- Khôûi taïo stack

```
void Init(stack &top)
{
    top=NULL;
}
```

- Kiểm tra stack rỗng :

```
int  Empty(stack top)
{
   return top == NULL ? 1 : 0; // stack rỗng
}
```


Biểu diễn Stack dùng danh sách liên kết

- Thêm một phần tử x vào stack S

```
void      Push(stack &top, node x)
{
  stack p;
  p=(node*)malloc(sizeof(node));
  if(p!=NULL)
  {
        p->data=x.data;
        p->link=top;
        top=p;
  }
}
```


Biểu diễn Stack dùng danh sách liên kết

Trích thông tin và huỷ phần tử ở đỉnh stack S

```
Int Pop(stack &top)
{
    if(!Empty(top))
    {
        stack p=top;
        return p->data;
        top=p->link;
        free(p);
    }
}
```


Ứng dụng của Stack

- Stack thích hợp lưu trữ các loại dữ liệu mà trình tự truy xuất ngược với trình tự lưu trữ
- Một số ứng dụng của Stack:
  - Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, Stack được sử dụng để lưu môi trường của các thủ tục.
  - Lưu dữ liệu khi giải một số bài toán của lý thuyết đồ thị (như tìm đường đi)
  - Khử đệ qui
  - …

Ứng dụng của Stack

Ví dụ:ï thủ tục Quick_Sort dùng Stack để khử đệ qui:

- 1. l:=1; r:=n;
- 2. Chọn phần tử giữa x:=a[(l+r) div 2];
- 3. Phân hoạch (l,r) thành (l1,r1) và (l2,r2) bằng cách xét:
  - y thuộc (l1,r1) nếu y≤x;
  - y thuộc (l2,r2) ngược lại;
- 4. Nếu phân hoạch (l2,r2) có nhiều hơn 1 phần tử thực hiện:
  - Cất (l2,r2) vào Stack;
  - Nếu (l1,r1) có nhiều hơn 1 phần tử thực hiện:
    - l=l1;
    - r=r1;
    - Goto 2;
  - Ngược lại
    - Lấy (l,r) ra khỏi Stack nếu Stack khác rỗng và Goto 2;
    - Nếu không dừng;

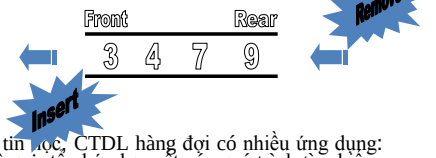
Hàng đợi (Queue)

Hàng đợi (Queue)

- Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế FIFO (*First In First Out*) ⇒ 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*".
- Các đối tượng có thể được thêm vào hàng đợi bất kỳ thời điểm nào, nhưng chỉ có thêm và khỏi hàng

Giới thiệu

- FIFO
- Thêm vào cuối và lấy ra ở đầu



- Trong tin học, CTDL hàng đợi có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui, vét cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, …

Hàng đợi (Queue)

- Hàng đợi là một CTDL trừu tượng (ADT) tuyến tính
- Hàng đợi hỗ trợ 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.
  - **Empty**(): 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 Queue dùng mảng

- Có thể tạo một hàng đợi bằng cách sử dụng một mảng 1 chiều với kích thước tối đa là N (ví dụ, N=1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0) ⇒ Hàng đợi chứa tối đa N phần tử.
- Phần tử ở đầu hàng đợi (front element) sẽ có chỉ số f.
- Phần tử ở cuối hàng đợi (rear element) sẽ có chỉ số r .

Biểu diễn Queue dùng mảng

- Dùng một array: Có xu hướng dời về cuối array
- Hai cách hiện thực đầu tiên:
  - Khi lấy một phần tử ra thì đồng thời dời hàng lên một vị trí

  - Chỉ dời hàng về đầu khi cuối hàng không còn chỗ

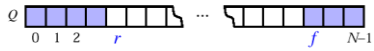
Biểu diễn Queue dùng mảng

- Để khai báo một hàng đợi, ta cần:
  - một mảng một chiều Q,
  - hai biến nguyên f, r cho biết chỉ số của đầu và cuối của hàng đợi
  - hằng số N cho biết kích thước tối đa của hàng đợi.
- Ngoài ra, khi dùng mảng biểu diễn hàng đợi, cần dùng một giá trị đặc biệt, ký hiệu là NULLDATA, để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu lưu trong hàng đợi..

Biểu diễn Queue dùng mảng

- Trạng thái hàng đợi lúc bình thường:

- Trạng thái hàng đợi lúc xoay vòng:



Biểu diễn Queue dùng mảng

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

```
typedef struct node
{
    int data;
}
typedef struct queue
{
   int        front, rear;
   node list[N];
}
```

- Do khi cài đặt bằng mảng một chiều, hàng đợi có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho hàng đợi:
  - **Full**(): Kiểm tra xem hàng đợi có đầy chưa.

Biểu diễn Queue dùng mảng

- Tạo hàng đợi rỗng

```
void   Init(queue &q)
{
   q.front = q.rear = UNDERFLOW;
}
```

- Kiểm tra queue rỗng

```
int Empty(queue q)
{
    if(q.front == q.rear == UNDERFLOW) return 1;
    return 0;
}
```


Biểu diễn Queue dùng mảng

- Kiểm tra hàng đợi đầy hay không

```
int Full(queue q)
{
  if( q.front== 0 && q.rear== N-1) return 1;
  if( q.front == q.rear+1) return 1;
  return 0;
}
```


Biểu diễn Queue dùng mảng

- Thêm một phần tử x vào cuối hàng đợi Q

```
void      EnQueue(queue &q, node x)
{
  if(!Full(q))   //Queue chưa đầy
  {
   if(Empty(q)   // Queue rỗng
          q.front=q.rear=0;
   else
          if( q.rear==N-1) q.rear=0;
          else q.rear++;
          q.list[q.rear]=x;
  }
}
```

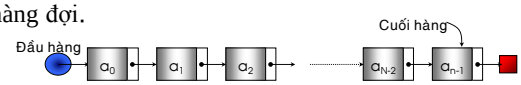

Biểu diễn Queue dùng mảng

- Trích, huỷ phần tử ở đầu hàng đợi Q

```
node      DeQueue(queue &q)
{
  if(!Empty(q))
  {
   node t=q.list[q.front];
   if(q.front == q.rear) Init(q);
   else
          if(q.fornt == N -1) q.front = 0;
          else q.front++;
   return t;
  }
}
```

Biểu diễn hàng đợi dùng danh sách liên kết

- Có thể tạo một hàng đợi sử dụng một DSLK đơn.
- Phần tử đầu DSKL (head) sẽ là phần tử đầu hàng đợi, phần tử cuối DSKL (tail) sẽ là phần tử cuối hàng đợi.



Biểu diễn hàng đợi dùng danh sách liên kết

```
typedef struct node
{
    int data;
    node *link;
};
typedef struct queue
{
    node *front, *rear;
};
```


Biểu diễn hàng đợi dùng danh sách liên kết

- Tạo hàng đợi rỗng:

```
void Init(queue &q)
{
  q.front=q.rear= NULL;
}
```

- Kiểm tra hàng đợi rỗng :

```
int  Empty(queue &q)
{
  if (q.front == NULL) return 1; // hàng đợi rỗng
  else      return 0;
}
```


Biểu diễn hàng đợi dùng danh sách liên kết

- Thêm một phần tử p vào cuối hàng đợi

```
void      EnQueue(queue &q, node *new_else)
{
  if(q.front == NULL)
  {
   q.front=new_else;
   q.rear=q.front;
  }
  else
  {
   q.front->link=new_else;
   q.rear=new_else;
  }
}
```


Biểu diễn hàng đợi dùng danh sách liên kết

- Trích phần tử ở đầu hàng đợi

```
node* DeQueue(queue &q)
{
  if (!Empty(q))
  {
   node *p=q.front;
   q.front=p->link;
   return p
  }
}
```


Biểu diễn hàng đợi dùng danh sách liên kết

Nhận xét:

- Các thao tác trên hàng đợi biểu diễn bằng danh sách liên kết làm việc với chi phí O(1).
- Nếu không quản lý phần tử cuối xâu, thao tác **Dequeue** sẽ có độ phức tạp O(n).

Ứng dụng của hàng đợi

- Hàng đợi có thể được sử dụng trong một số bài toán:
  - Bài toán 'sản xuất và tiêu thụ' (ứng dụng trong các hệ điều hành song song).
  - Bộ đệm (ví dụ: Nhấn phím ⇒ Bộ đệm ⇒ CPU xử lý).
  - Xử lý các lệnh trong máy tính (ứng dụng trong HĐH, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, ….

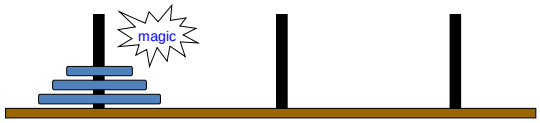
Ứng dụng STACK để khử đệ quy cho bài toán tháp Hà Nội

Bài toán Tháp Hà nội

- Luật:
  - Di chuyển mỗi lần một đĩa
  - Không được đặt đĩa lớn lên trên đĩa nhỏ

Bài toán Tháp Hà nội – Thiết kế hàm

- Hàm đệ qui:
  - Chuyển (count-1) đĩa trên đỉnh của cột start sang cột temp
  - Chuyển 1 đĩa (cuối cùng) của cột start sang cột finish
  - Chuyển count-1 đĩa từ cột temp sang cột finish



Bài toán Tháp Hà nội – Mã C++

```
void move(int n, char start, char finish, char temp)
{
        if (n > 0)
        {
                move(n − 1, start, temp, finish);
                cout << "Move disk " << n << " from " ;
                cout<< start << " to " << finish << "." << endl;
                move(n − 1, temp, finish, start);
        }
}
```


Bài giảng môn học

Cấu trúc dữ liệu và giải thuật