

Chương 3

Danh sách Tuyển tính

Trong chương này, chúng ta sẽ nghiên cứu danh sách tuyển tính, một trong các mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong việc cài đặt các bài toán đệ quy. Các phương pháp cài đặt danh sách khác nhau sẽ được xem xét. Hai kiểu dữ liệu trừu tượng được biết quan trọng là ngăn xếp (STACK) và hàng đợi (QUEUE) sẽ được nghiên cứu. Chúng ta cũng sẽ trình bày một số ứng dụng phổ biến của danh sách.

3.1 KIỂU CON TRƯ VÀ CÁC KHÁI NIỆM CÓ LIÊN QUAN:

Tất cả các biến có kiểu cấu trúc dữ liệu mà ta đã nghiên cứu như mảng, switch được gọi là biến tĩnh vì chúng được xác định một cách rõ ràng khi khai báo, sau đó chúng được dùng thông qua tên. Thời gian tồn tại của biến tĩnh cũng là thời gian tồn tại của khối chương trình có chứa khai báo biến này. Chẳng hạn, các biến được tĩnh khai báo trong chương trình (biến toàn cục) sẽ tồn tại từ khi chương trình được thực hiện cho đến khi kết thúc chương trình, còn các biến tĩnh được khai báo trong một hàm (biến địa phương) sẽ tồn tại từ khi hàm được thực hiện cho đến khi kết thúc hàm này.

Ngoài các biến tĩnh được xác định trước, ngoài ra ta còn có thể tạo ra các biến trong lúc chạy chương trình, tùy theo nhu cầu. Việc tạo ra các biến theo kiểu này được gọi là cấp phát bộ nhớ động, các biến được tạo ra được gọi là biến động.

Các biến động không có tên. Để tạo ra biến động, ngoài ra sẽ dùng một kiểu biến được biết, gọi là con trỏ và một số hàm cấp phát bộ nhớ động (malloc hoặc calloc) thông qua con trỏ. Khi không sử dụng biến động nữa, ngoài ra có thể xóa nó khỏi bộ nhớ, việc này gọi là thu hồi bộ nhớ động. Để thu hồi bộ nhớ dành cho biến động, ngoài ra dùng hàm Free và thông qua con trỏ đã sử dụng để tạo ra biến động.

So với biến tĩnh, biến số động biến động có ưu điểm là tiết kiệm đáng kể bộ nhớ. Bởi vì, khi cần dùng biến động thì ngỏ ý ta sẽ tạo ra nó và khi không cần nữa ngỏ ý ta lại có thể xóa nó đi. Còn đối với các biến tĩnh, chúng được xác định và cấp phát bộ nhớ khi biên dịch, chúng sẽ chiếm giữ bộ nhớ trong suốt thời gian chạy trình làm việc. Chính hẳn, nếu cần số động một mảng ta phải khai báo ngay ở phần đầu chương trình, ngay lúc này ta đã phải xác định kích thước của mảng và thời gian khai báo đôi ra gây lãng phí bộ nhớ.

Lưu ý: Chúng ta chỉ có thể số động bộ nhớ động khi chúng ta đã khai báo thời gian `std::lib.h`.

3.1.1 Kiểu con trỏ

Chúng ta đã biết các biến chính là các ô nhớ mà chúng ta có thể truy xuất được các tên. Các biến này được lưu trữ tại những chỗ cố định trong bộ nhớ. Đối với chương trình của chúng ta, bộ nhớ máy tính chỉ là một dãy gồm các ô nhớ 1 byte, mỗi ô có một địa chỉ xác định.

Một số mô hình tốt đối với bộ nhớ máy tính chính là một phần trong một thành phần. Trên một phần tốt của các ngôi nhà đầu được đánh số tuần tự với một cái tên duy nhất nên nếu chúng ta nói đến số 27 phần Trọn Hạng Đeo thì chúng ta có thể tìm được nơi đó mà không lo lắng vì chỉ có một ngôi nhà với số nhà vậy.

Cũng với cách tổ chức thông tin nhớ việc đánh số các ngôi nhà, hệ điều hành tổ chức bộ nhớ thành những số đến nhất, tuần tự, nên nếu chúng ta nói đến vị trí 1776 trong bộ nhớ chúng ta biết chính xác ô nhớ đó vì chỉ có một vị trí với địa chỉ nhớ vậy.

a) Toán tử lấy địa chỉ (&).

Vào thời điểm mà chúng ta khai báo một biến thì nó phải được lưu trữ trong một vị trí cố định trong bộ nhớ. Nói chung chúng ta không quyết định nơi nào biến đó được đặt - thật may mắn rằng điều đó đã được làm tốt bằng bộ trình biên dịch và hệ điều hành, nhưng một khi hệ điều hành đã gán một địa chỉ cho biến thì chúng ta có thể mượn biết biến đó được lưu trữ ở đâu.

Điều này có thể được thể hiện bằng cách đặt trước tên biến một dấu và (&), có nghĩa là "địa chỉ của". Ví dụ:

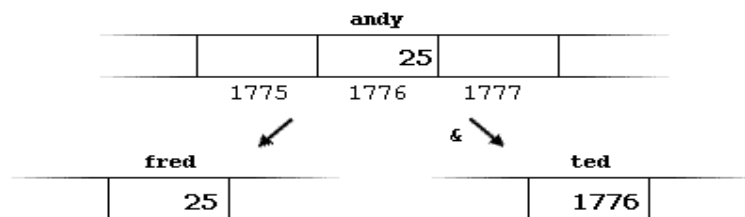
```
ted = &andy;
```

sẽ gán cho biến **ted** địa chỉ của biến **andy**, vì khi đặt trước tên biến **andy** dấu và (&) chúng ta không còn nói đến nội dung của biến đó mà chỉ nói đến địa chỉ của nó trong bộ nhớ.

Giả sử rằng biến **andy** được đặt ở ô nhớ có địa chỉ **1776** và chúng ta viết như sau:

```
andy = 25;  
fred = andy;  
ted = &andy;
```

kết quả sẽ giống như trong sơ đồ dưới đây:



Chúng ta đã gán cho **fred** nội dung của biến **andy** như chúng ta đã làm rất lần nhiều khác trong những phần trước của chương với biến **ted** chúng ta đã gán địa chỉ mà họ đi để hành lưu giá trị của biến **andy**, chúng ta viết nó là **1776**.

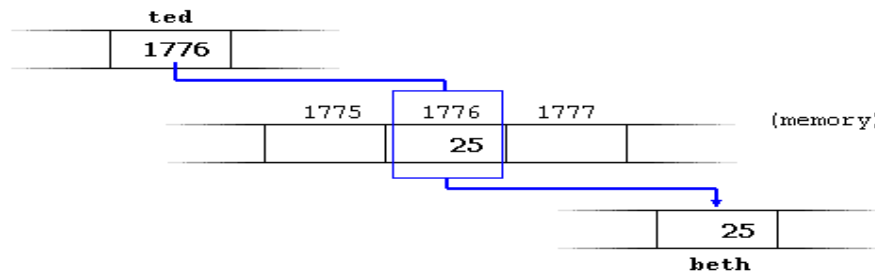
Những biến lưu trữ địa chỉ của một biến khác (như **ted** trong ví dụ trước) được gọi là **con trỏ**. Trong C++ con trỏ có rất nhiều ưu điểm và chúng được sử dụng rất thường xuyên, Tiếp theo chúng ta sẽ thấy các biến kiểu này được khai báo như thế nào.

b) Toán tử tham chiếu (*)

Bằng cách sử dụng con trỏ chúng ta có thể truy xuất trực tiếp đến giá trị được lưu trữ trong biến được trỏ bởi nó bằng cách đặt trước tên biến con trỏ một dấu sao (*) - đây có thể được dịch là "**giá trị được trỏ bởi**". Vì vậy, nếu chúng ta viết:

```
beth = *ted;
```

(chúng ta có thể hiểu nó là: "beth bằng giá trị được trỏ bởi ted" **beth** sẽ mang giá trị 25, vì **ted** bằng 1776 và giá trị trỏ bởi 1776 là 25.



Bạn phải phân biệt được rằng **ted** có giá trị 1776, nhưng ***ted** (với mặt đầu sao đứng trước) trỏ tới giá trị được lưu trữ trong địa chỉ 1776, đó là 25. Hãy chú ý sự khác biệt giữa việc có hay không có dấu sao tham chiếu.

```
beth = ted; // beth bằng ted ( 1776 )
```

```
beth = *ted; // beth bằng giá trị được trỏ bởi ( 25 )
```

Toán tử lấy địa chỉ (&)

Nó được dùng như là một tiền tố của biến và có thể được dịch là "địa chỉ của", vì vậy &variable1 có thể được dịch là "địa chỉ của variable1".

Toán tử tham chiếu (*)

Nó chỉ ra rằng cái cần được tính toán là nội dung được trỏ bởi biểu thức được coi như là một địa chỉ. Nó có thể được dịch là "giá trị được trỏ bởi"..

*mypointer được dịch là "giá trị được trỏ bởi mypointer".

Vào lúc này, với những ví dụ đã viết ở trên

```
andy = 25;
ted = &andy;
```

bạn có thể dễ dàng nhận ra tất cả các biểu thức sau là đúng:

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

c) Khai báo biến kiểu con trỏ

Vì con trỏ có khả năng tham chiếu trực tiếp đến giá trị mà chúng trỏ tới nên cần thiết phải chỉ rõ kiểu dữ liệu nào mà một biến con trỏ trỏ tới khi khai báo nó. Vì vậy, khai báo của một biến con trỏ sẽ có mẫu sau:

type * pointer_name;

trong đó ***type*** là kiểu dữ liệu được trỏ tới, không phải là kiểu của bản thân con trỏ. Ví dụ:

```
int * number;
char * character;
float * greatnumber;
```

Đó là ba khai báo của con trỏ. Mỗi biến đều trỏ tới một kiểu dữ liệu khác nhau nhưng cả ba đều là con trỏ và chúng đều chỉ một lổ hổng bẻ khúc khúc nhau (kích thước của một biến con trỏ tùy thuộc vào hệ điều hành). Nhưng dù là kiểu mà chúng trỏ tới không chỉ một lổ hổng bẻ khúc khúc nhau, một kiểu **int**, một kiểu **char** và cái còn lại kiểu **float**.

Ta phải chú ý lổ hổng khúc khúc sao (*) mà chúng ta đặt khi khai báo một con trỏ chỉ có nghĩa khúc khúc: đó là một con trỏ và hoàn toàn không liên quan đến toán tử tham chiếu mà chúng ta đã xem xét trước đó. Đó được gọi khúc khúc là hai tác vụ khác nhau được biểu diễn bởi cùng một dấu.// *my first pointer*

```
#include <iostream.h>
```

```
int main ()
```

```
{
```

```
int value1 = 5, value2 = 15;
```

```
int * mypointer;
```

```
value1==10 / value2==20
```

```

mypointer = &value1;
*mypointer = 10;
mypointer = &value2;
*mypointer = 20;
cout << "value1==" << value1 << "/
value2==" << value2;

return 0;
}

```

Chú ý rằng giá trị của **value1** và **value2** đã được thay đổi một cách gián tiếp. Đầu tiên chúng ta gán cho **mypointer** địa chỉ của **value1** dùng toán tử lấy địa chỉ (&) và sau đó chúng ta gán **10** cho giá trị được trỏ bởi **mypointer**, đó là giá trị được trỏ bởi **value1** vì vậy chúng ta đã sửa biến **value1** một cách gián tiếp.

Đến bên có thể thấy rằng một con trỏ có thể mang một vài giá trị trong cùng một chương trình chúng ta sẽ lặp lại quá trình với **value2** và với cùng một con trỏ.

Đây là một ví dụ phức tạp hơn một chút:

```

// more pointers
#include <iostream.h>
int main ()
{
    int value1 = 5, value2 = 15;
    int *p1, *p2;

    p1 = &value1;    // p1 = địa chỉ của value1
    p2 = &value2;    // p2 = địa chỉ của value2

    *p1 = 10;        // giá trị trỏ bởi p1 = 10
    *p2 = *p1;        // giá trị trỏ bởi p2 = giá trị trỏ bởi p1
}

```

value1==10 / value2==20

```

p1 = p2;          // p1 = p2 (phép gán
con tr)
*p1 = 20;         // giá trị b i p1 =
20
cout << "value1==" << value1 <<
"/ value2==" << value2;
return 0;
}

```

Một dòng có thể gây sự chú ý của bạn là:

```
int *p1, *p2;
```

dòng này khai báo hai con tr bằng cách đặt dấu sao (*) trước mỗi con tr. Nguyên nhân là kiểu dữ liệu khai báo cho cả dòng là **int** và vì theo thói quen phải sang trái, dấu sao được tính trước tên kiểu

Con tr và mảng.

Trong thói quen, tên của một mảng thường được viết ở địa chỉ phần tử đầu tiên của nó, giống như một con tr thường được viết ở địa chỉ của phần tử đầu tiên mà nó trỏ tới, vì vậy thói quen chúng hoàn toàn như nhau. Ví dụ, cho hai khai báo sau:

```
int numbers [20];
int *p;
```

Lệnh sau sẽ hợp lệ:

```
p = numbers;
```

Ở đây **p** và **numbers** là biến được định nghĩa và chúng có cùng thuộc tính, sự khác biệt duy nhất là chúng ta có thể gán một giá trị khác cho con tr **p** trong khi **numbers** luôn trỏ đến phần tử đầu tiên trong số 20 phần tử kiểu int mà nó được định nghĩa

. Vì vậy, không giống như **p** - đó là một biến con tr bình thường, **numbers** là một con tr hằng. Lệnh gán sau đây là không hợp lệ:

```
numbers = p;
```

bởi vì **numbers** là mảng (con trỏ hằng) và không có giá trị nào có thể được gán cho các hằng.

Vì con trỏ cũng có một tính chất của một biến nên tất cả các biểu thức có con trỏ trong ví dụ dưới đây là hoàn toàn hợp lệ:

```
// more pointers
#include <iostream.h>

int main ()
{
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

10, 20, 30, 40, 50,

Trong bài "mảng" chúng ta đã dùng dấu ngoặc vuông để chỉ ra phần tử của mảng mà chúng ta muốn truy cập. Cặp ngoặc vuông này được coi như là toán tử offset và ý nghĩa của chúng không đổi khi được dùng với biến con trỏ. Ví dụ, hai biểu thức sau đây:

`a[5] = 0;` *// a [offset of 5] = 0*

`*(a+5) = 0;` *// pointed by (a+5) = 0*

là hoàn toàn tương đương và hợp lệ bất kể **a** là mảng hay là một con trỏ.

d) Khi i t o con tr

Khi khai báo con tr có th chúng ta s mu n ch đ nh r ã ràng chúng s tr t i bi n nào

```
int number;  
int *tommy = &number;
```

là t ã ng đ ã ng v i:

```
int number;  
int *tommy;  
tommy = &number;
```

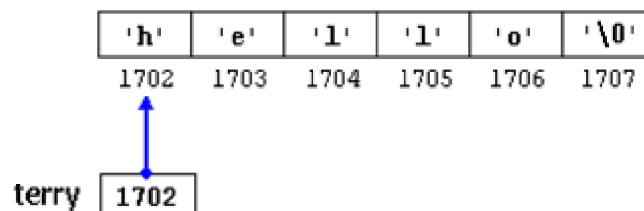
Trong m t phép g n con tr chúng ta ph i luôn luôn g n đ a ch m à nó tr t i ch không ph i là giá tr mà nó tr t i. B n c n ph i nh r ng khi khai báo m t bi n con tr, đ u sao (*) đ ã c dùng đ ch ra nó là m t con tr, và hoàn toàn khác v i toán t tham chi u. Đó là hai toán t khác nhau m c dù chúng đ ã c vi t v i cùng m t đ u. Vì v y, các câu l nh sau là không h p l :

```
int number;  
int *tommy;  
*tommy = &number;
```

Nh đ i v i m ng, trình biên d ch cho phép chúng ta kh i t o giá tr mà con tr tr t i b ng giá tr h ng vào th i đ i m khai báo bi n con tr :

```
char * terry = "hello";
```

trong tr ã ng h p này m t kh i nh t ã c dành đ ch a "hello" và m t con tr tr t i kí t đ u tiên c a kh i nh này (đó là kí t h') đ ã c g n cho **terry**. N u "hello" đ ã c l u t i đ a ch 1702, l nh khai báo trên có th đ ã c hình dung nh th này:



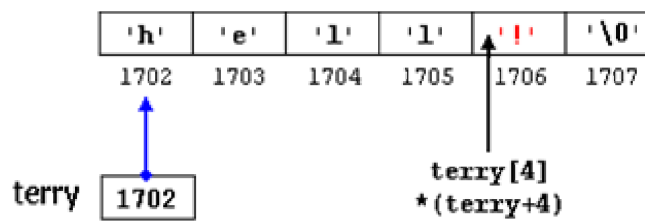
còn phải nhớ lại rằng **terry** mang giá trị **1702** chứ không phải là **'h'** hay **"hello"**.

Biến con trỏ **terry** trỏ tới một chuỗi ký tự và nó có thể được sử dụng như là đối với một mảng (hãy nhớ rằng một mảng chỉ đơn thuần là một con trỏ hằng). Ví dụ, nếu chúng ta muốn thay ký tự **'o'** bằng một dấu chấm than, chúng ta có thể thực hiện việc đó bằng hai cách:

```
terry[4] = '!';
```

```
*(terry+4) = '!';
```

hãy nhớ rằng việc **terry[4]** là hoàn toàn giống với việc ***(terry+4)** mặc dù biểu thức thông dụng nhất là cái đầu tiên. Việc một trong hai lệnh trên chuỗi do **terry** trỏ đến sẽ có giá trị như sau:



e) Các phép toán trên con trỏ :

* Phép gán: Chúng ta nên gán các con trỏ cùng kiểu. Muốn gán các con trỏ khác kiểu phải dùng phép ép kiểu như ví dụ sau:

```
int x;
```

```
char *pc;
```

```
pc=(char *)&x ;
```

* Phép tăng giảm địa chỉ : Ta sẽ giới thiệu quy tắc tăng giảm địa chỉ thông qua ví dụ :

```
float x[30] ,*p ;
```

```
p=&x[10] ;
```

Cho biến con trỏ **p** trỏ tới phần tử **x[10]**. Khi đó

p + i sẽ trỏ đến phần tử **x[10+i]**

p - i sẽ trỏ đến phần tử **x[10- i]**

* Nguyên tắc truy nhập bộ nhớ: Con trỏ float truy nhập tới 4byte, con trỏ int truy nhập 2byte, con trỏ char truy nhập một byte. Ta minh họa qua các ví dụ sau:

```
float *pf;  
int *pi;  
char *pc;
```

Khi đó:

+ Nếu con trỏ pf trỏ đến byte 10001, thì *pf sẽ biểu thị vùng nhớ 4byte liên tiếp từ byte 10001 đến byte 10004.

+ Nếu con trỏ pi trỏ đến byte 10001, thì *pi sẽ biểu thị vùng nhớ 2byte liên tiếp từ byte 10001 đến byte 10002.

+ Nếu con trỏ pc trỏ đến byte 10001, thì *pc sẽ biểu thị vùng nhớ 1byte là byte 10001.

* Phép so sánh: Cho phép so sánh các con trỏ cùng kiểu. Ví dụ như 2 con trỏ p1, p2 có cùng kiểu float thì:

+ $p1 > p2$ nếu địa chỉ p1 trỏ tới cao hơn địa chỉ p2 trỏ tới.

+ $p1 = p2$ nếu địa chỉ p1 trỏ tới cũng là địa chỉ p2 trỏ tới.

+ $p1 < p2$ nếu địa chỉ p1 trỏ tới thấp hơn địa chỉ p2 trỏ tới.

Đó đây là một ví dụ về việc sử dụng phép so sánh con trỏ trong bài toán tính tổng dãy số:

```
float a[100], *pd, *pc, s=0.0;  
int n;  
pc=a+n-1;  
for (pd=a; pd<=pc; pd++)  
    s=s+*pd;
```

f) Thu hồi và cấp phát bộ nhớ động:

* Trong C để cấp phát bộ nhớ động người ta sử dụng hàm malloc() (hoặc calloc() hoặc realloc())

```
void *malloc(n_byte).
```

trong đó n_byte là số byte ta muốn cấp phát cho biến động. Hàm trả về giá trị một con trỏ kiểu void* vì vậy để sử dụng ta phải ép kiểu như trong ví dụ sau:

```
char *pc;
pc=(char*) malloc (5);
```

Đoạn này là c/p phát cho biến pc một khối nhớ 5byte. Tuy nhiên nếu cần phát 1 khối dữ liệu có kiểu khác Char ta phải nhân số phần tử với kích thước của kiểu dữ liệu.

```
int *pi ;
pi=(int *) malloc (5*sizeof(int)) ;
```

Đoạn mã này c/p phát một khối nhớ gồm 5 số nguyên kiểu int cho con trỏ pi.

* Trong C người ta sử dụng hàm Free() để thu hồi bộ nhớ được c/p phát bởi các hàm malloc(), calloc() hoặc realloc().

```
void Free(void * pointer).
```

3.2. Khái niệm danh sách tuyến tính

3.2.1. Khái niệm danh sách

Về mặt toán học, danh sách là một dãy hữu hạn các phần tử thuộc cùng một tập các đối tượng nào đó. Chẳng hạn, danh sách sinh viên của một lớp, danh sách các số nguyên, danh sách các báo xuất bản hàng ngày ở thủ đô v.v...

Giả sử L là một danh sách có n phần tử ($n \geq 0$).

$$L = (a_1, a_2, \dots, a_n)$$

Ta gọi n là độ dài của danh sách. Nếu $n \geq 1$ thì a_1 được gọi là phần tử đầu tiên, a_n được gọi là phần tử cuối cùng của danh sách L . Nếu $n = 0$ thì danh sách L được gọi là danh sách rỗng.

Một tính chất quan trọng của danh sách là các phần tử của nó được sắp xếp tuyến tính: nếu $n > 1$ thì phần tử a_i “đi trước” phần tử a_{i+1} . Ta gọi a_i ($i = 1, 2, \dots, n$) là phần tử ở vị trí thứ i của danh sách. Nghĩa là, một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì danh sách đó được gọi là danh sách tuyến tính.

3.2.2. Các phép toán trên danh sách.

Khi mô tả một mô hình dữ liệu, chúng ta cần xác định các phép toán có thể thực hiện trên mô hình toán học được dùng làm cơ sở cho mô hình dữ liệu.

Có rất nhiều phép toán trên danh sách. Trong các bài giảng, thông thường chúng ta chỉ sẽ dùng một nhóm các phép toán nào đó. Sau đây là một số phép toán cơ bản trên danh sách tuyến tính.

1. *Phép toán Empty: Kiểm tra xem danh sách có rỗng hay không?*

```
Int Empty(struct list ds);
```

2. *Phép toán Full: Kiểm tra xem danh sách có đầy không?*

```
Int Full(struct list ds);
```

3. *Phép thêm vào: Thêm một phần tử có nội dung info vào vị trí thứ k của danh sách:*

```
Void Insert_info(struct list ds, int info, int k);
```

4. *Phép xóa bỏ: Xóa bỏ phần tử thứ k khỏi danh sách tuyến tính. Khi xóa bỏ một phần tử thì danh sách ít nhất phải có 1 phần tử:*

```
Void Delete_info(struct list ds, int k);
```

5. *Duyệt danh sách: Duyệt từ đầu cho đến cuối danh sách, mỗi phần tử đều có duy nhất một lần:*

```
Void Traverse_list(struct list ds);
```

6. *Tìm kiếm: Tìm vị trí đầu tiên của phần tử có giá trị X trong danh sách. Nếu không có thì hàm trả về giá trị -1:*

```
Int Search_info(struct list ds, int X);
```

7. *Các phép toán khác:*

Còn có thể kể ra nhiều phép toán khác. Chẳng hạn truy cập đến phần tử thứ k của danh sách (để tham khảo hoặc thay thế), kết hợp hai danh sách thành một danh sách, tách một danh sách thành nhiều danh sách v.v...

Ví dụ: Giả sử có danh sách $L = (3, 2, 1, 5)$. Khi đó, thực hiện `Delete_info(L, 3)` ta được danh sách $(3, 2, 5)$. Kết quả của `Insert_info(L, 1, 6)` ta được danh sách $(6, 3, 2, 1, 5)$.

Sau đây ta sẽ xét một số loại danh sách và ứng dụng của chúng.

3.3. List liên kết kép có danh sách tuyến tính.

Ta biết rằng danh sách tuyến tính là một danh sách hai chiều, hai chiều có dạng $L=(a_1, a_2, \dots, a_n)$. Trong danh sách tuyến tính luôn tồn tại một phần tử đầu là **a1** và một phần tử cuối là **an**.

Đối với danh sách tuyến tính trong bộ nhớ máy tính, một phương pháp rất tự nhiên là sử dụng mảng một chiều, trong đó mỗi thành phần của mảng lưu trữ một phần tử nào đó của danh sách, các phần tử kế nhau của danh sách được lưu trữ trong các thành phần kế nhau của mảng. List liên kết theo cách này gọi là list liên kết.

Tuy nhiên việc sử dụng mảng một chiều cũng có nhược điểm và những cải tiến như thế:

- + Vì mảng được lưu trữ liên tiếp nên việc truy cập vào một thành phần nào đó được thực hiện trực tiếp đưa vào địa chỉ tính được (chỉ số), nên tốc độ nhanh và đáng lưu ý với mỗi phần tử.

- + Khi khai báo một mảng ta phải xác định số lượng phần tử của mảng, điều này sẽ phụ thuộc vào số lượng phần tử của danh sách mà mảng lưu trữ, nhược điểm này rất khó thực hiện vì số lượng phần tử của danh sách luôn luôn biến động. Do đó, có thể dẫn đến lãng phí bộ nhớ (có nhược điểm là mảng không được sử dụng) hoặc thiếu bộ nhớ (do tất cả các phần tử mảng đã được sử dụng trong khi ta cần thêm vào danh sách một số phần tử nào đó).

Sau đây ta trình bày phương pháp cài đặt danh sách tuyến tính bởi mảng một chiều:

Để đơn giản, các phép toán trên danh sách sẽ được thao tác trên danh sách các số nguyên với khai báo như sau:

```
#define Maxlist 100;
```

```
typedef struct list {
```

```
    int n; /* số phần tử của danh sách.
```

```
    int nodes[Maxlist]; /* danh sách là mảng 1 chiều.
```

```
}
```

struct list ds;



Hình 3.1: Mô hình biểu diễn danh sách.

Trong cách cài đặt danh sách bởi mảng, các phép toán trên danh sách được thực hiện rất dễ dàng. Để khai tạo danh sách cũng chỉ cần một lệnh gán:

ds.n = 0;

Độ dài thực của danh sách là ds.n, danh sách đầy nếu ds.n=Maxlist.

Ở đây ta cài đặt hai phép toán trên danh sách: phép toán bổ sung một phần tử mới vào danh sách và phép toán loại bỏ một phần tử khỏi danh sách.

1. Phép toán Empty: Kiểm tra danh sách có rỗng hay không?

```
int Empty(struct list ds)
{
    return (ds.n == 0 ? true : false);
}
```

2. Phép toán Full: Kiểm tra danh sách có đầy hay không?

```
Int Full(struct list ds)
{
    return(ds.n==maxlist ? 1 : 0);
}
```

3. *Phép thêm vào: Thêm một phần tử có nội dung info vào vị trí thứ k của danh sách.*

- Nếu $k=0$ thì ta thêm phần tử vào đầu danh sách.
- Nếu $k=ds.n+1$ ta thêm phần tử vào cuối danh sách.

Lưu ý: Khi thêm một phần tử vào danh sách ta phải kiểm tra xem danh sách đã đầy hay chưa.

```
void Insert_info(struct list ds, int info, int k)
```

```
{
    int j;
    if (k<0 || k> ds.n)
        printf("Vị trí chen không phù hợp");
    else
        if (Full(ds)== 0)
            printf("Danh sách đã đầy ");
        else
        {
            for (j= ds.n; j>= k+1;j--)
                ds.nodes[j]=ds.nodes[j-1] ;
            ds.nodes[k]=info ;
            ds.n++ ;
        }
}
```

Hàm trên thực hiện phép bổ sung một phần tử vào vị trí k trong danh sách. Phép toán được thực hiện khi danh sách chưa đầy và k chèn vào một phần tử trong danh sách. Khi bổ sung, ta phải dời các phần tử ở các vị trí k, ..., ds.n lên một vị trí và tăng số lượng phần tử của danh sách lên một $ds.n = ds.n + 1$.

4. *Phép xóa bỏ : Xóa bỏ một phần tử khỏi danh sách tùy tính.*

Lưu ý: Khi xóa bỏ một phần tử khỏi danh sách thì danh sách phải có ít nhất một phần tử.

```
void Delete_info(struct list ds, int k)
```

```

{
    int j;
    if (k<0 || k>ds.n)
        printf("Vi tri loai bo khong thich hop ");
    else
        if (Empty(ds))
            printf("Danh sach khong co phan tu nao ");
        else
            {
                for (j=k; j<ds.n-1; j++)
                    ds.nodes[j]=ds.nodes[j+1] ;
                ds.n-- ;
            }
}

```

5. *Duyệt danh sách* : Duyệt từ đầu cho đến cuối danh sách, mỗi phần tử được duyệt đúng một lần. Khi sẽ ta duyệt danh sách để in giá trị các phần tử :

```

void traverse_list(struct list ds)
{
    int i ;
    if (Empty(ds))
        printf("Danh sach khong co phan tu ");
    else
        for (i=0 ;i<ds.n ; i++)
            printf("Phan tu thu %3d",i,"cua danh sach la %3d",ds.nodes[i]);
}

```

6. *Tìm kiếm* : Tìm vị trí đầu tiên xuất hiện phần tử có giá trị info trong danh sách. Nếu không có info trong danh sách hàm Search_info sẽ trả về giá trị -1:

```

int Search_info(struct list ds)
{
    int i;
    for (i=0;i<ds.n;i++)

```

```

        if (ds.nodes[i]=info)
            return i+1;
        return(-1);
    }

```

Vì cài đặt danh sách bởi mảng cho ta thấy một số ưu điểm và nhược điểm sau:

+ Ưu điểm: Do tính chặt chẽ của mảng phương pháp này cho phép ta truy nhập trực tiếp vào bất kỳ phần tử nào trong danh sách nên tốc độ truy cập nhanh và đúng đắn với mọi phần tử. Các phép toán cũng đều được thực hiện một cách dễ dàng.

+ Nhược điểm: Khi thực hiện các phép toán bổ sung một phần tử vào danh sách hoặc loại bỏ một phần tử ra khỏi danh sách ở vị trí bất kỳ nào đó, ta phải di chuyển tất cả các phần tử sau khi xoá hoặc lên trên một vị trí. Tuy nhiên, nhược điểm chủ yếu của phương pháp cài đặt này là không gian nhớ cần dành giành để lưu trữ các phần tử của danh sách. Không gian nhớ này bị quy định bởi kích thước của mảng (kích thước của mảng được xác định khi khai báo và nó không thay đổi trong khi thực hiện chương trình). Do đó có thể dẫn đến tình huống lãng phí bộ nhớ (do khai báo kích thước mảng quá lớn so với số lượng các phần tử của danh sách) hoặc thiếu bộ nhớ (mảng đã đầy trong khi ta muốn bổ sung thêm một số phần tử nào đó vào danh sách).

Để khắc phục các nhược điểm trên đây người ta sẽ dùng một phương pháp khác để cài đặt danh sách tuy nhiên tính đó là danh sách mố c n i.

3.4. Lưu trữ mố c n i của danh sách tuy n t i n h

Như đã nêu ở phần trên, lưu trữ kiểu tiếp diễn với danh sách tuy n t i n h đã bộc lộ rõ nhược điểm trong trình hợp thực hiện thông xuyên các phép bổ sung hoặc loại bỏ phần tử, trình hợp xử lý đúng thời điểm danh sách v.v...

Vì số lượng con trỏ hoặc m i n i đ i t h c danh sách tuy n t i n h, mà ta gọi là danh sách mố c n i (hay còn gọi là danh sách mố c n i), chính là một giải pháp nhằm khắc phục nhược điểm trên.

3.2. Danh sách móc nối đơn

3.2.1. Nguyên tắc

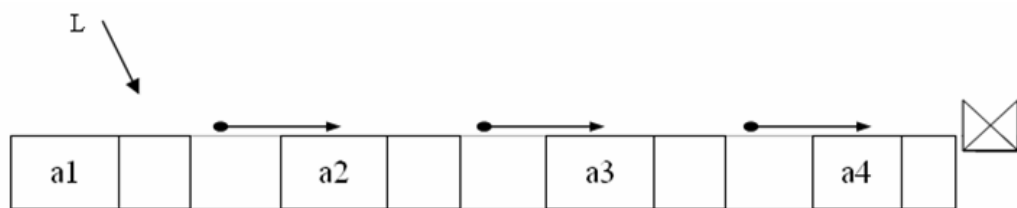
Trong cách cài đặt này, danh sách móc nối đơn có thể tạo nên từ các phần tử nhỏ mà ta gọi là nút (**Node**). Các nút này có thể nằm bất kỳ đâu trong bộ nhớ máy tính. Mỗi nút là một cấu trúc gồm hai thành phần, infor chứa thông tin của phần tử trong danh sách, next là một con trỏ nó trỏ vào nút đứng sau. Qui cách của mỗi nút có thể hình dung như sau:



Riêng nút cuối cùng thì không có nút đứng sau nó nên thành phần next của nút này có giá trị NULL để báo kết thúc danh sách.

Để có thể truy nhập vào mỗi nút trong danh sách, ta phải truy nhập từ nút đầu tiên, nghĩa là cần có một con trỏ L trỏ tới nút đầu tiên này.

Nếu dùng mũi tên để chỉ mỗi nút, ta sẽ có hình ảnh của một danh sách móc nối đơn như sau:



Hình 3.2: Biểu diễn danh sách móc nối đơn.

Để lưu trữ giá trị trong next của phần tử này bằng NULL.

Gọi số các phần tử trong danh sách có kiểu dữ liệu là **Item**. Dưới đây là khai báo cấu trúc dữ liệu biểu diễn danh sách móc nối đơn.

```
struct Node
```

```
{
```

```
    Item infor;
```

```
    Node *next;
```

```
};
```

```
struct Node *L; //Khai báo con trỏ L trỏ vào đầu danh sách
```

$L = \text{NULL}$ nếu danh sách rỗng.

Ví dụ : Khai báo danh sách lưu trữ thông tin về sinh viên

```
struct student      //Item là student
{
    char std_no[10];      //Mã sinh viên
    char std_name[30];    //Họ tên
    int age;              //Tuổi
    float avg_point;      //Điểm trung bình
};

struct Node
{
    student info;
    Node *next;
};

Node *L;
```

3.2.2. Các phép toán trên danh sách móc nối đơn.

Bây giờ chúng ta sẽ xem xét một số phép toán tác động trên danh sách móc nối đơn.

Điều kiện để danh sách móc nối đơn rỗng là $L = \text{NULL}$. Do đó, để khi tạo danh sách rỗng ta chỉ cần lệnh gán: $L = \text{NULL}$;

Danh sách móc nối chỉ đầy khi không còn không gian nhớ để cấp phát cho các phần tử mới của danh sách. Chúng ta giả thiết điều này không xảy ra, nghĩa là danh sách móc nối không bao giờ đầy. Do đó, phép toán bổ sung một phần tử vào danh sách luôn luôn được thực hiện.

3.2.2.1. Bổ sung một nút mới vào danh sách móc nối đơn.

Giả sử Q là một con trỏ, trỏ vào một nút trong danh sách, ta cần bổ sung một phần tử mới với thông tin lưu trong biến X vào sau nút được trỏ bởi Q . Phép toán này được thực hiện bởi thuật toán sau:

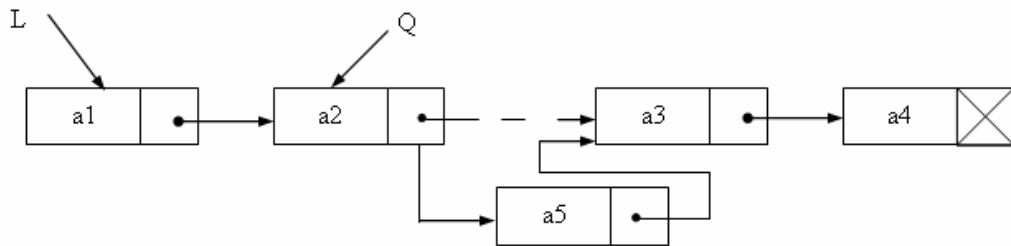
```
void InsertAfter(Node **L, Node *Q, Item X)
{
    Node *P : Tro;
```

```

//1. Tạo mới nút mới
p=(Node *)malloc(sizeof(Node));
p->infor = X;
/*2. Thêm mới vào danh sách, nếu danh sách rỗng thì thêm nút mới vào
thành nút đầu tiên, ngược lại thêm nút mới vào sau nút đầu tiên là Q*/

if (L == NULL)
{
    p->next = NULL;
    L = p;
}
else {
    p->next = Q->next;
    Q->next = p;
}
}

```



Hình 3.3 Mô tả phép thêm mới phần tử vào sau nút trước là Q trong danh sách.

Giả sử bây giờ ta cần thêm nút mới vào trước nút đầu tiên là Q. Phép toán này phức tạp hơn. Khó khăn là nếu Q không phải là nút đầu tiên của danh sách ($Q \neq L$) thì ta không thể xác định được nút đứng trước Q để kết nối nó với nút mới. Có thể giải quyết khó khăn bằng cách, đầu tiên ta vẫn thêm nút mới vào sau Q, sau đó trao đổi giá trị chứa trong phần infor giữa nút mới và nút đứng trước Q. Khi thực hiện phép toán này xin dành cho bạn đọc.

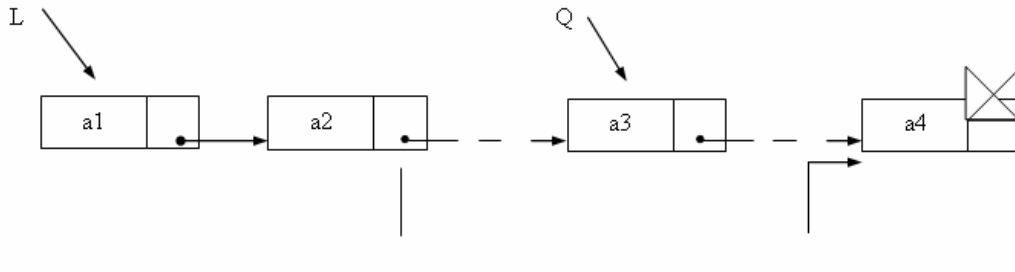
3.2.2.2. Loại bỏ một nút ra khỏi danh sách móc nối đơn.

Cho danh sách móc nối đơn đầu tiên là L. Q là một con trỏ, trỏ vào một nút trong danh sách. Giả sử ta cần loại bỏ nút đứng trước Q. Ở đây ta cũng gặp khó khăn là nếu Q không phải là nút đầu tiên thì không xác định được nút đứng trước Q. Trong trường hợp này ta phải tìm được nút đứng trước Q và cho

con trỏ R trỏ vào nút đó, tức là $Q = R \rightarrow \text{next}$. Sau đó ta mới thực hiện xóa khỏi nút Q. Ta có thể thực hiện sau:

```
void DeleteL(Node **L, Node *Q, Item &X)
{
    Node *R;
    //1. Kiểm tra danh sách rỗng
    if (L == NULL)
    {
        print( "Danh sach rong"); return;
    }
    X = Q->infor; //lưu thông tin của nút cần xóa vào biến X
    //2. Kiểm tra nút trỏ bởi Q là nút đầu tiên
    if (Q == L)
    {
        L = Q->next; free(Q);
        return;
    }
    //3. Tìm đến nút đang trỏ tới nút trỏ bởi Q
    R = L;
    while (R->next != Q)
        R = R->next;
    //4. Xóa khỏi nút trỏ bởi Q
    R->next = Q->next; free(Q);
}
```

Phép toán xóa khỏi được mô tả bởi hình sau:



Hình 3.4: Mô tả phép xóa khỏi một phần tử khỏi danh sách.

3.2.2.3. Ghép hai danh sách móc nối đôi n.

Giả sử có hai danh sách móc nối đôi liên tiếp được trỏ bởi L1 và L2. Sau đó thực hiện việc ghép hai danh sách đó thành một danh sách móc nối đôi trỏ bởi L1.

```
void COMBINE(Node ** L1, Node *L2)
{ Node *R;
  //1. Trỏ về địa chỉ đầu của danh sách trỏ bởi L2
  if (L2 == NULL) return;
  //2. Trỏ về địa chỉ đầu của danh sách trỏ bởi L1
  if (L1 == NULL)
  {
    *L1 = L2; return;
  }
  //3. Tìm địa chỉ nút cuối của danh sách trỏ bởi L1
  R = *L1;
  while (R->next != NULL) R = R->next;
  //4. Ghép
  R->next = L2;
}
```

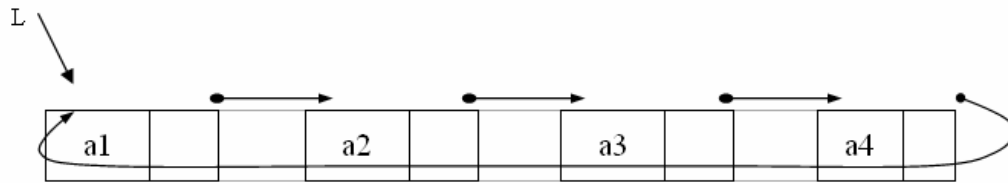
Nhận xét: Rõ ràng với các danh sách tuyến tính mà kích thước luôn biến động trong quá trình xử lý hay thao tác xuyên có các phép bổ sung và loại bỏ tác động, thì việc cài đặt bằng danh sách móc nối như trên tỏ ra thích hợp. Tuy nhiên cách cài đặt này cũng có nhược điểm như sau:

Chỉ có phần tử đầu tiên trong danh sách được truy cập trực tiếp, các phần tử khác chỉ được truy cập sau khi đã qua các phần tử đứng trước nó.

Mỗi nút trong danh sách phải có thêm trường Next để lưu trữ địa chỉ của nút tiếp theo, do đó với cùng một danh sách thì việc cài đặt bằng danh sách móc nối sẽ tốn bộ nhớ hơn.

3.3. Danh sách nối vòng

Một cái tên của danh sách móc nối đơn là kiểu danh sách móc nối vòng. Nó khác với danh sách móc nối đơn ở chỗ: trường Next của nút cuối cùng trong danh sách không phải bằng NULL, mà nó trỏ đến nút đầu tiên trong danh sách, tạo thành một vòng tròn. Hình ảnh của nó như sau:

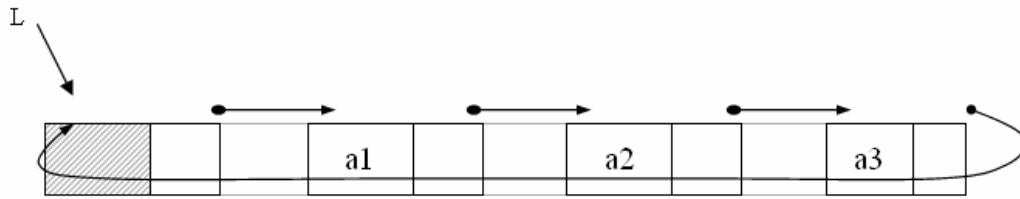


Hình 3.5: Mô tả danh sách móc nối vòng.

Cái tên này làm cho việc truy nhập vào các nút trong danh sách dễ dàng hơn. Ta có thể truy nhập vào mọi nút trong danh sách bất kể từ nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng được. Nhờ vậy, đối với danh sách móc nối vòng chỉ cần cho biết con trỏ trỏ tới nút muốn loại bỏ ta vẫn thực hiện được vì vẫn tìm được đến nút đang trỏ tới đó. Với phép ghép, phép tách cũng có những thuận lợi như thế.

Tuy nhiên, danh sách móc nối vòng có một nhược điểm rất rõ là trong khi xử lý, nếu không cần thận sơ đoán thì một chu trình không kết thúc, bởi vì không biết được vị trí kết thúc danh sách.

Để khắc phục nhược điểm này, người ta đã thêm vào danh sách một nút đặc biệt gọi là “nút đầu danh sách”. Trường infor của nút này không chứa dữ liệu của phần tử nào và con trỏ L bây giờ trỏ tới nút đầu danh sách này. Việc dùng thêm nút đầu danh sách đã khiến cho danh sách vẫn giữ hình thức không bao giờ rỗng. Hình ảnh của nó như sau:



Hình 3.6: Mô tả danh sách móc nối vòng có nút đầu.

Sau đây là đồ án ghi thu t b sung m t nút vào thành nút đầu tiên trong danh sách có “nút đầu danh sách” tr b i L.

```
P=(Node*) malloc(sizeof(Node));
```

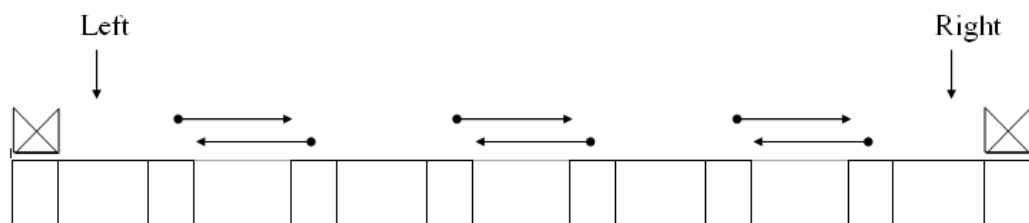
```
P->infor = X;
```

```
P->next = L->next;
```

```
L->next = P;
```

3.4. Danh sách móc n i hai chi u.

Khi làm vi c v i danh sách, có nh ng x lý tr ên m i nút c a danh sách l i liên quan đ n c nút đ ng tr c và nút đ ng sau. Trong nh ng tr ng h p nh th, đ thu n ti n, ng i ta đ a vào m i nút c a danh sách hai con tr : Next_left tr đ n nút đ ng tr c và Next_right tr đ n nút đ ng sau nó. Đ truy nh p vào danh sách ta dùng hai con tr : con tr Left tr vào nút đầu tiên và con tr Right tr vào nút cuối cùng c a danh sách. Hình nh c a danh sách móc n i hai chi u nh sau:



Hình 3.7: Mô t danh sách móc n i đôi.

Ta có th khai báo c u trúc d li u danh sách móc n i hai chi u nh sau:

```

Typedef struct Node{
    Item Infor;

    Struct Node *Next_left, *Next_right;

};

Node *Left, *Right;

```

Việc cài đặt danh sách móc nối hai chiều sẽ tiêu tốn nhiều bộ nhớ hơn so với danh sách móc nối đơn. Song bù lại, danh sách móc nối đôi có những ưu điểm mà danh sách móc nối đơn không thể có được, chẳng hạn: khi xem xét danh sách móc nối đôi ta có thể lùi lại sau, hoặc tiến lên trước.

Các phép toán trên danh sách móc nối hai chiều được thực hiện dễ dàng hơn. Chẳng hạn, khi thực hiện phép toán loại bỏ, với danh sách móc nối đơn, ta không thể thực hiện được nếu không biết nút đang trỏ tới nút cần loại bỏ. Trong khi đó, ta có thể tiến hành dễ dàng trên danh sách móc nối hai chiều.

Đó chính là một số giải thuật tác động lên danh sách móc nối hai chiều nói trên.

3.4.1. Phép bổ sung một nút mới

Cho hai con trỏ Left và Right lần lượt trỏ tới nút đầu và nút cuối của một danh sách móc nối hai chiều, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện bổ sung một nút mới, mà dữ liệu chứa ở biến X, vào trước nút trỏ bởi M.

```

void Bo_sung(Node **Left, Node **Right, Node *M, Item X)
{
    Node P;
    //1. Tạo nút mới
    P = (Node*)malloc(sizeof(Node);
    P->infor = X;
    {Trỏ vào hộp danh sách rỗng}
    if (*Right == NULL)
    {
        P->Nextleft = NULL; P->Next_right = NULL;
    }
}

```

```

        Left = P; Right = P;
    }
    else
//Trình ng h p M tr t i nút đ u tiên}
        if (M == Left)
        {
            P->Next_left = NULL;
            P->Next_right = M
            M->Next_left = P;
            Left = P;
        }
        else //B sung vào gi a
        {
            P->Next_left = M->Next_left;
            P->Next_right = M;
            M->Next_left = P;
            P->Next_left->Next_right = P;
        }
    }
}

```

3.4.2. Loại bỏ nút trên danh sách

Cho hai con tr Left và Right lần l ột tr t i nút đ u và nút cuối c a m t danh sách móc n i hai chi u, M là con tr tr t i m t nút trong danh sách này. Gi i thu t này th c hi n lo i b nút tr b i M ra kh i danh sách.

```

void Loai_bo(Node **Left, Node **Right, Node *M)
{
    if (Left == NULL)
        printf( "Danh sach rong");
    else
        if (Left == Right)
        {
            Left = NULL; Right = NULL;
        }
        else
            if (M == Left)

```

```

    {
        Left = Left->Next_right;
        Left->Next_left = NULL;
    }
    else if (M == Right)
    {
        Right = Right^.Next_left;
        Right^.Next_right = NULL;
    }
    else {
        M->Next_left->Next_right = M->Next_right;
        M->Next_right->Next_left = M->Next_left;
    }
}

```

Chú ý: Trong các loại đồ thị, người ta cũng thường sử dụng các danh sách móc nối hai chiều vòng tròn, có nút đầu danh sách. Với loại danh sách này, ta có thể có các lưu đồ tìm kiếm danh sách móc nối hai chiều và danh sách vòng tròn.

3.4. Loại đồ thị danh sách móc nối: các phép tính số học trên đa thức

Trong mục này ta sẽ xét các phép tính số học cơ bản (cộng, trừ, nhân, chia) đối với đa thức mà ta có định nghĩa:

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (1)$$

Mỗi hàm thức của đa thức được biểu diễn bởi hai số và số mũ của x. Giả sử các hàm thức trong đa thức được sắp xếp theo thứ tự giảm dần của số mũ, như trong đa thức (1). Ta thấy đa thức như một danh sách tuyến tính với các phần tử của danh sách là các hàm thức của đa thức. Khi ta thực hiện các phép toán trên đa thức ta sẽ nhận được đa thức có bậc không thể đoán trước được. Ngay cả hàm đa thức có bậc xác định thì số các hàm thức của nó cũng biến đổi rất nhiều từ một đa thức này đến một đa thức khác. Do đó, phương pháp tốt nhất là biểu diễn đa thức dưới dạng một danh sách móc nối. Mỗi nút của danh sách là một bản ghi gồm ba trường: coef chứa hệ số, exp chứa số mũ của x và con trỏ Next để trỏ tới nút tiếp theo. Cấu trúc dữ liệu mô tả một hàm thức (một nút) như sau:

```
struct Node
```

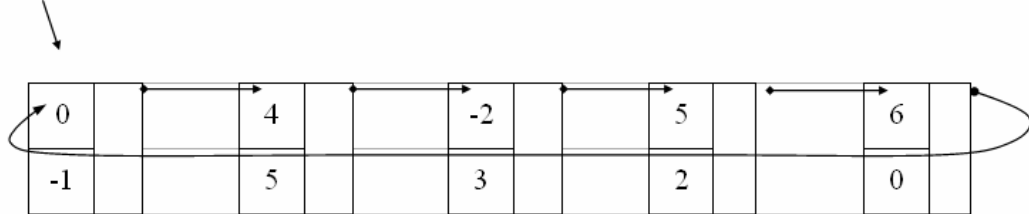
```

{
    float coef;
    int exp;
    Node *Next;
};

```

Vì nhúng vào điểm của danh sách vòng tròn có nút đầu danh sách (không cần kiểm tra danh sách rỗng, mỗi thành phần đều có thành phần đi sau), ta chọn danh sách móc nối vòng tròn để biểu diễn đa thức. Với cách chọn này việc thực hiện các phép toán đa thức sẽ rất gọn. Nút đầu danh sách là nút đặc biệt, có $\text{exp} = -1$.

Như vậy với đa thức: $A(x) = 4x^5 - 2x^3 + 5x^2 + 6$ sẽ được biểu diễn như sau



Hình 3.8: Danh sách móc nối đôi biểu diễn đa thức.

Sau đây chúng ta sẽ xét phép cộng hai đa thức $A(x)$ và $B(x)$. Con trỏ A trỏ tới đầu danh sách biểu diễn đa thức $A(x)$, con trỏ B trỏ tới đầu danh sách biểu diễn đa thức $B(x)$. Sau khi thực hiện phép cộng hai đa thức trên ta được đa thức $C(x)$ và con trỏ C trỏ tới đầu danh sách biểu diễn $C(x)$.

Ghi chú

Trước hết cần phải suy nghĩ để thực hiện phép cộng đa thức $A(x)$ với đa thức $B(x)$ ta phải tìm được từng hạng thức của các đa thức đó, nghĩa là phải dùng hai biến con trỏ P và Q để duyệt qua hai danh sách từng bước với hai đa thức $A(x)$ và $B(x)$ trong quá trình tìm này.

Ta thấy có những trường hợp sau:

1. $\text{EXP}(P) = \text{EXP}(Q)$, ta sẽ phải thực hiện cộng giá trị coef của hai nút đó, nếu giá trị khác không thì phải tạo ra nút mới thể hiện hạng thức tương ứng và gán vào danh sách tiếp theo của $C(x)$.

2. $EXP(P) > EXP(Q)$ (hệ số mũ của P cũng lớn hơn Q): phải sao chép nút P và gán vào danh sách của $C(x)$.

3. Nếu nút danh sách kết thúc trước: phần còn lại của danh sách kia sẽ được sao chép và gán vào danh sách của $C(x)$.

Mũi tên nút mới được tạo ra đều phải gán vào cuối danh sách $C(x)$. Do đó, cần một con trỏ R trỏ vào nút cuối của danh sách $C(x)$.

Công việc này được thực hiện nhiều lần, vì vậy cần được thực hiện bằng một thủ tục gọi là $Attack(h, m, R)$. Nó thực hiện: lấy nút mới, đưa vào trường hệ số của nút này giá trị h (hệ số), đưa vào trường exp giá trị m (số mũ) và gán nút mới đó vào sau nút trỏ bởi con trỏ R .

```
void Attack(float h, int m, Node *R)
{
    Node *N;
    N=(Node*)malloc(sizeof(Node));
    N->coef = h;
    N->exp = m;
    R->Next = N;
    R = N;
}
```

Sau đây là thủ tục cộng hai đa thức.

```
void Add(Node *A, Node *B, Node **C)
{
    Node *P, *Q, *R;
    float x;
    P = A; Q = B;
    *C=(Node*)malloc(sizeof(Node));
    *C->exp = -1; R = *C;
    while (P->exp != -1 && Q->exp != -1)
    if (P->exp == Q->exp)
    {
        x = P->coef + Q->coef;
        if (x != 0)
            Attack(x, P->exp, R);
    }
}
```

```

        P = P->Next; Q = Q->Next;
    }
    else
        if (P->exp < Q->exp)
        {
            Attack(Q->coef, Q->exp, R);
            Q = Q->Next;
        }
        else {
            Attack(P->coef, P->exp, R);
            P = P->Next;
        }
    while (P->exp != -1) //Danh sách đang vớ i B(x) đã hết
    {
        Attack(P->coef, P->exp, R);
        P = P->Next;
    }
    while (Q->exp != -1) //Danh sách đang vớ i A(x) đã hết
    {
        Attack(Q->coef, Q->exp, R);
        Q = Q->Next;
    }
    R->Next = *C;
}

```

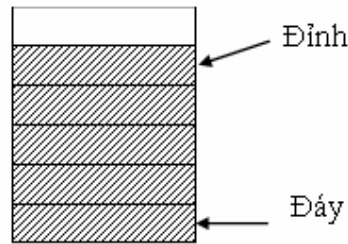
4. STACK VÀ QUEUE

4.1. Stack

4.1.1. Khái niệm:

Ngăn xếp (STACK) là một danh sách tuyến tính, trong đó phép bổ sung mới phần tử vào ngăn xếp và phép loại bỏ một phần tử khi ngăn xếp luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

Có thể hình dung Ngăn xếp như cấu trúc mảng tĩnh điển hình. Việc đưa dữ liệu vào ngăn xếp hay lấy dữ liệu ra khỏi ngăn xếp chỉ đơn giản là thao tác di chuyển dữ liệu. Viên dữ liệu mới nhập nằm ở đỉnh còn viên dữ liệu nhập đầu tiên nằm ở đáy ngăn xếp.



Hình 3.9 ngăn xếp

4.1.2. Cài đặt ngăn xếp bằng mảng.

Giả sử danh sách được biểu diễn là mảng ngăn xếp, có thể dài tối đa là mảng số nguyên điển hình N nào đó, các phần tử của ngăn xếp có kiểu dữ liệu là **Item**. **Item** có thể là các kiểu dữ liệu điển hình, hoặc các kiểu dữ liệu có cấu trúc, thông thường **Item** là biến ghi. Chúng ta biểu diễn ngăn xếp bằng một biến ghi gồm 2 trường. Trường thứ nhất là mảng các **Item**, trường thứ 2 ghi chỉ số của thành phần mảng lưu trữ phần tử ở đỉnh của ngăn xếp. Chúng ta khai báo cấu trúc dữ liệu biểu diễn ngăn xếp như sau:

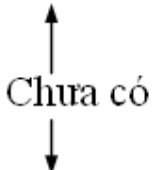
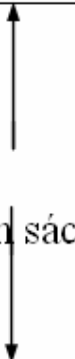
```
#define Max N

struct Stack
{
    Item E[Max];

    unsigned int top;
};

Stack S; //Khai báo ngăn xếp S
```

Với cách cài đặt này, nếu $S.top = 0$ thì S là ngăn xếp rỗng, $S.top = Max$ thì S là ngăn xếp đầy.

Max		
		
top (Đỉnh)	phần tử cuối cùng	
.....		
2	phần tử thứ hai	
1 (Đáy)	phần tử thứ nhất	

Hình 3.10. Mảng biểu diễn ngăn xếp.

Ví dụ: Đơn giản chương trình.

```
#define Max 100
struct Hoc_sinh
{
    char ho_ten[25];
    int tuoi;
};
Struct Stack
{
    Struct Hoc_sinh E[max];
    unsigned int top;
};
Struct Stack S;
```

Khai báo ngăn xếp S có thể chứa tối đa 100 phần tử, mỗi phần tử (*Item*) là một cấu trúc Hoc_sinh gồm 2 thành phần ho_ten và tuoi.

Các phép toán trên ngăn xếp.

Giả sử S là ngăn xếp, các phần tử của nó có kiểu *Item* và X là một phần tử có cùng kiểu với các phần tử của ngăn xếp. Ta có các phép toán sau với ngăn xếp S.

1. Khởi tạo ngăn xếp rỗng (ngăn xếp không chứa phần tử nào)

```
void Initialize (struct Stack *S)
```

```
{  
    S->top = 0;  
}
```

2. Kiểm tra ngăn xếp rỗng

```
int Empty ( struct stack S)
```

```
{  
    return (S.top == 0?1:0);  
}
```

Hàm Empty nhận giá trị true nếu S rỗng và false nếu S không rỗng.

3. Kiểm tra ngăn xếp đầy

```
int Full ( struct S)
```

```
{  
    return (S.top == Max?1:0);  
}
```

Hàm Full nhận giá trị true nếu S đầy và false nếu không.

4. Thêm một phần tử mới vào đỉnh ngăn xếp

Đầu tiên bổ sung phần tử X vào đỉnh của ngăn xếp S, trước hết phải kiểm tra xem S có đầy không. Nếu S đầy thì việc bổ sung không thể hiện được, ngược lại X được bổ sung vào đỉnh của S. Hàm PUSH trả về 1 nếu bổ sung thành công, ngược lại trả về 0.

```
int PUSH ( Stack *S, Item X)
```

```
{  
    if (Full(*S)==1) return 0;  
    else  
    {  
        S->top = S->top + 1;  
        S->E[S->top-1] = X;  
        return 1;  
    }  
}
```

5. Loại bỏ phần tử ở đỉnh của ngăn xếp

Việc lỗi bị chấp nhận có thể hiển nhiên nếu S không rỗng, giá trị của phần tử bị lỗi bị gán cho biến X . Hàm POP trả về 1 nếu việc lỗi bị thành công, ngược lại trả về 0.

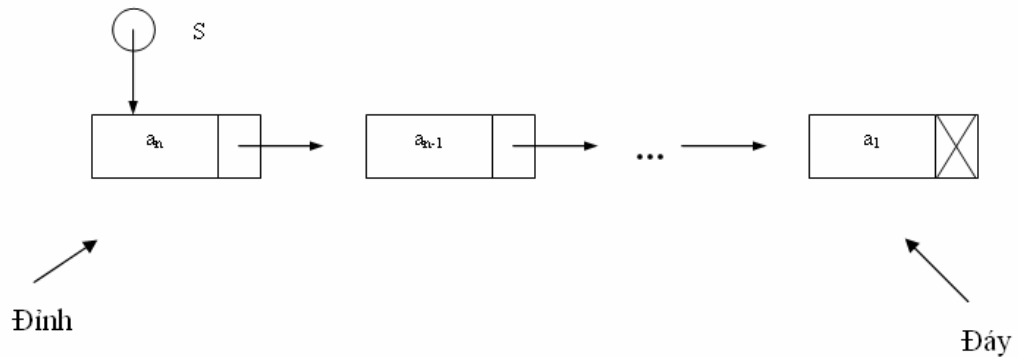
```

int POP (Stack *S; Item *X)
{
    if (Empty(*S)==1) return 0;
    else
    {
        *X = S->E[S->top-1];
        S->top = S->top - 1;
        return 1;
    }
}

```

4.1.3. Cài đặt ngăn xếp bằng danh sách móc nối đơn.

Để cài đặt ngăn xếp bằng danh sách móc nối đơn, ta sẽ dùng con trỏ S trỏ vào phần tử ở đỉnh của ngăn xếp (hình 3.11).



Hình 3.11: Danh sách móc nối đơn biểu diễn ngăn xếp.

Cấu trúc dữ liệu của ngăn xếp được khai báo như sau:

```

struct Node
{
    Item Infor;
    Struct Node *Next;
};
Struct Node *S;

```

Trong cách cài đặt này, ngăn xếp rỗng khi $S = \text{NULL}$. Ta giữ số vị trí của phát biểu để dùng cho các phần tử mới luôn thể hiện. Do đó, ngăn xếp không bao giờ đầy và phép toán PUSH luôn thể hiện thành công.

***) Các hàm và thủ tục thể hiện các phép toán trên ngăn xếp:**

1. Khởi tạo ngăn xếp rỗng:

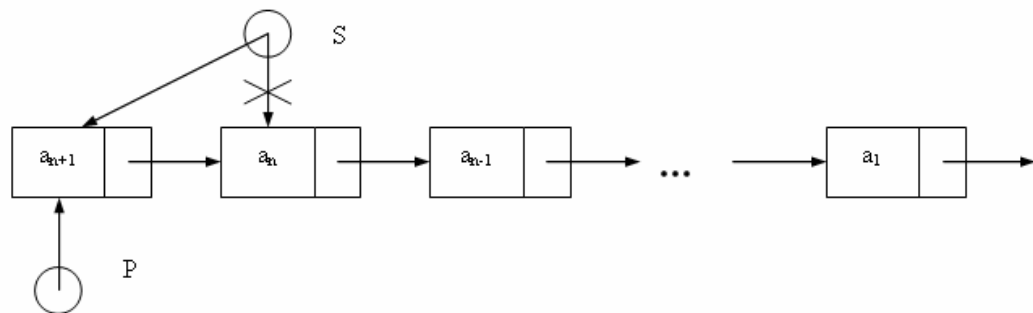
```
void Create(struct Node **S)
{
    *S = NULL;
}
```

2. Kiểm tra ngăn xếp rỗng:

```
int Empty(Node *S)
{
    return (S == NULL?1:0);
}
```

3. Bổ sung một phần tử vào đỉnh ngăn xếp:

```
void PUSH(Node **S, Item X)
{
    Node *P;
    P = (Node *) malloc (sizeof(Node));
    P->Infor = X;
    P->Next = *S;
    *S = P;
}
```



Hình 3.12: Minh họa thao tác PUSH.

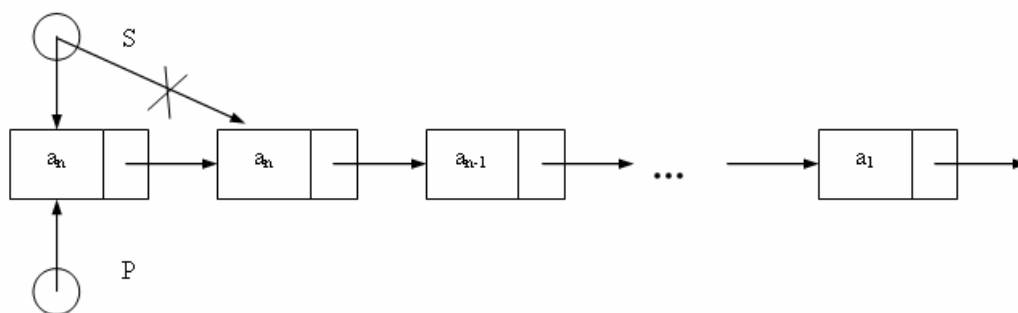
4. Lấy ra một phần tử từ đỉnh ngăn xếp:

```

int POP(Node **S, Item *X)
{
    Node *P;
    if (Empty(*S)==1) return 0;
    else
    {
        P = *S;
        *X = *P->Infor;
        *S = *P->Next;
        free(P);
        return 1;
    }
}

```

Hàm POP trả về 1 nếu việc xóa thành công, ngược lại trả về 0

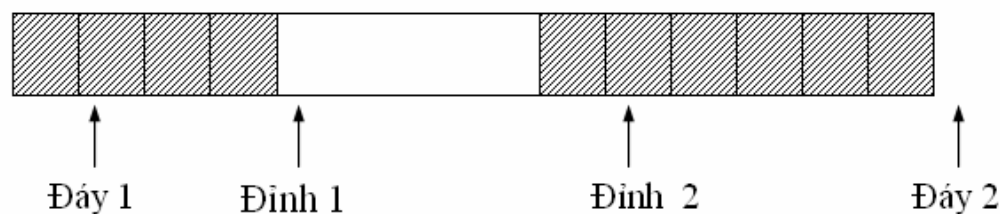


Hình 3.13: Minh họa thao tác POP.

4.1.4. Xử lý với nhiều ngăn xếp

Có những trường hợp cùng một lúc ta phải xử lý nhiều ngăn xếp trên cùng một không gian nhớ. Như vậy, có thể xảy ra tình trạng một ngăn xếp này đã bị tràn trong khi không gian dành cho ngăn xếp khác vẫn còn chỗ trống (trần cục bộ). Làm thế nào để khắc phục được tình trạng này?

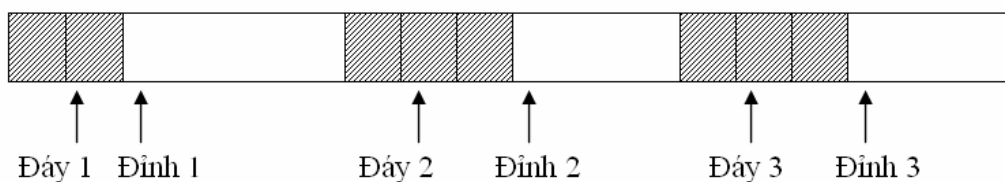
Nếu là hai ngăn xếp thì có thể giải quyết dễ dàng. Ta không qui định kích thước tối đa cho từng ngăn xếp nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta đặt hai ngăn xếp ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như hình dưới đây.



Hình 3.14: Hai ngăn xếp trên mặt không gian nhô

Như vậy, có thể mặt ngăn xếp này dùng lên sang quá nửa không gian dưới như ngăn xếp kia chỉ dùng đến. Do đó hiện tượng tràn chệch xảy ra khi toàn bộ không gian nhô dành cho chúng đã được dùng hết.

Nhưng nếu số lượng ngăn xếp từ 3 trở lên thì không thể làm theo kiểu như vậy được, mà phải có giải pháp linh hoạt hơn nữa. Chẳng hạn có 3 ngăn xếp, lúc đầu không gian nhô có thể chia đều cho cả 3, như hình dưới đây.



Hình 3.15: Ba ngăn xếp trên mặt không gian nhô

Nhưng nếu có mặt ngăn xếp nào phát triển nhanh bỏ phần trống mà ngăn xếp khác vẫn còn chỗ thì phải dọn chỗ cho nó bằng cách hoặc đẩy ngăn xếp đứng sau nó sang bên phải hoặc lùi chính ngăn xếp đó sang trái trong trường hợp hợp có thể. Như vậy thì đáy của các ngăn xếp phải được phép di động và dĩ nhiên các giải thuật bổ sung hoặc loại bỏ phần tử đối với các ngăn xếp hoạt động theo kiểu này cũng phải thay đổi.

4.1.5. Một số ứng dụng của ngăn xếp.

4.1.5.1. Ứng dụng để kiểm tra.

Ta biết rằng dữ liệu lưu trữ trong bộ nhớ của máy tính đều được biểu diễn dưới dạng mã nhị phân. Như vậy các số xuất hiện trong chương trình đều phải

chuyển đổi từ hệ thập phân sang hệ nhị phân trực khi thực hiện các phép xử lý.

Khi đổi một số nguyên từ hệ thập phân sang hệ nhị phân thì người ta dùng phép chia liên tiếp cho 2 và lấy các số dư (là các chữ số nhị phân) theo chiều ngược lại.

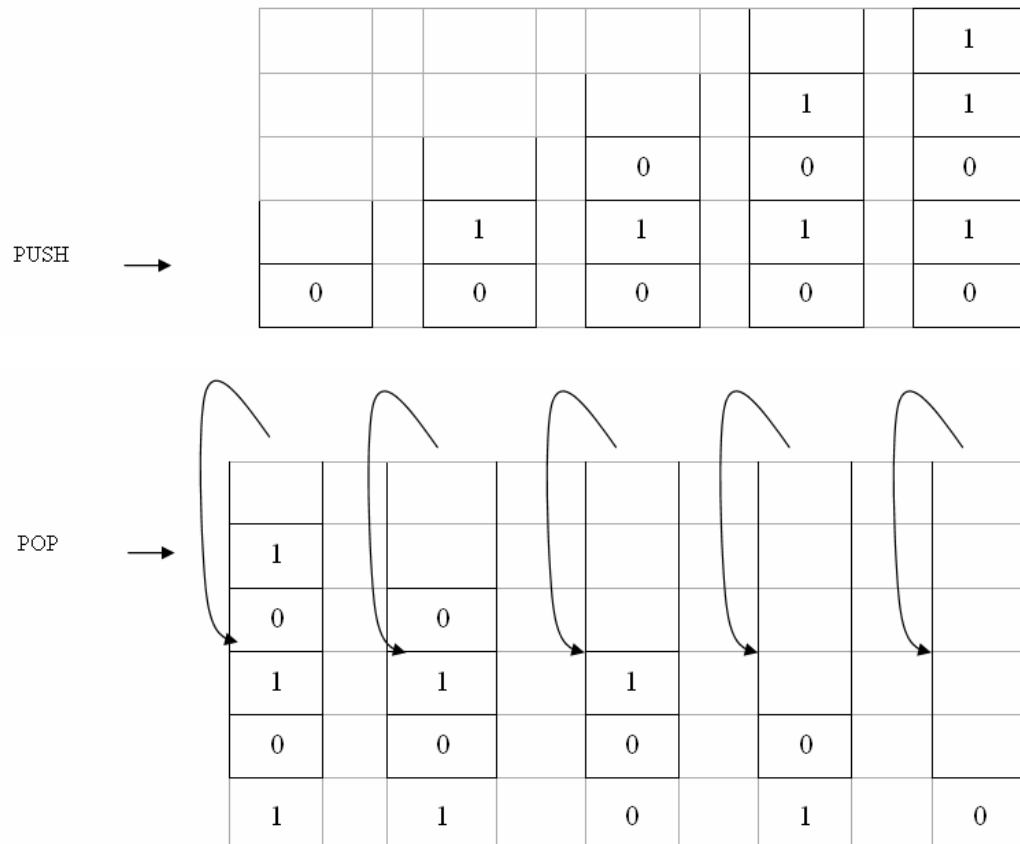
Ví dụ:

215	2							
1	107	2						
	1	53	2					
		1	26	2				
			0	13	2			
				1	6	2		
					0	3	2	
						1	1	2
							1	0

————→ 11010111

Ta thấy trong cách biến đổi này các số được tạo ra sau lại được hiển thị trực. Cách sắp xếp này chính là cách hoạt động của ngăn xếp. Để thực hiện biến đổi ta sẽ dùng một ngăn xếp để lưu trữ các số dư qua từng phép chia: Khi thực hiện phép chia thì nhập số dư vào ngăn xếp, sau đó lấy chúng lên một ngăn xếp ra.

Ví dụ: S: $(26)_{10} = (11010)_2$ trong quá trình biến đổi các số dư lần lượt sẽ là: (0 1 0 1 0 1) .



Hình 3.16: Mô tả hoạt động lưu trữ của ngăn xếp.

Ta khai báo cấu trúc dữ liệu cho bài toán này như sau:

#define Max 16 //thì c hiên đ i s nguyên có kích th ì c 2 byte

typedef int Item; //Item là ch ì s nh ì phân

struct Stack

{ **Item** E[Max];

int top;

};

Stack S;

*/*S là ngăn ch ì a các s ì d ì qua các phép chia trong quá trình chuy ìn đ ì i*/*

Gi ì i thu ì t s ì d ì ng ngăn x ì p th ì c hi ì n chuy ìn đ ì i s nguyên đ ì ì ng N t ì h ì c ì s ì 10 sang h ì c ì s ì 2. Trong gi ì i thu ì t này có s ì d ì ng các phép toán trên ngăn x ì p.

void Chuyen_doi(N)

{

```

1 – while (N != 0)
    R = N % 2; //tính số dư trong phép chia N cho 2
    call PUSH(S, R);
    N = N / 2; //thay N bằng thương của phép chia N cho 2
2 – printf(“\nma nhì phan:”);
    while (!Empty(S))
    {
        call POP(S, R);
        printf(R);
    }
}

```

4.1.5.2. Nguyên tắc định giá biểu thức số học theo ký pháp nghịch đảo o Balan.

Nhiệm vụ của bộ dịch là tạo ra các chương trình máy tính để thực hiện các lệnh của chương trình nguồn. Một phần trong nhiệm vụ này là tạo ra các chương trình định giá các biểu thức số học. Chương trình câu lệnh gán $X = A * B + C$.

Bộ dịch phải tạo ra các chương trình máy tính như sau:

- 1 – LOA A: Tìm giá trị của A lưu trữ trong bộ nhớ và tải nó vào thanh ghi.
- 2 – MUL B: Tìm giá trị của B và nhân nó với giá trị đang ở thanh ghi.
- 3 – ADD C: Tìm giá trị của C và cộng nó với giá trị trong thanh ghi.
- 4 – STO X: Đưa giá trị trong thanh ghi vào lưu trữ ở vị trí tiếp theo của X, trong bộ nhớ.

Trong các ngôn ngữ lập trình, biểu thức số học được viết như dòng thông thường của toán học nghĩa là theo ký pháp trung tâm (infix notation) mà ký hiệu của phép toán hai ngôi được đặt giữa hai toán hạng, có thể thêm dấu ngoặc.

Chương trình: $5 * (7 + 3)$

Dấu ngoặc là cần thiết vì nếu viết $5 * 7 + 3$ thì theo qui tắc ưu tiên của phép toán (mà các ngôn ngữ lập trình đều chấp nhận) thì biểu thức trên nghĩa là lấy 5 nhân 7 được kết quả cộng với 3.

Nhà logic học người Ba Lan Lukasiewicz đã đưa ra dạng biểu thức số học theo ký pháp hậu tố (postfix notation) và tiền tố (prefix notation) mà chúng ta gọi là dạng ký pháp Balan.

Ở dạng hậu tố các toán tử đi sau các toán hạng. Như biểu thức $5*(7+3)$ sẽ có dạng: $5\ 7\ 3\ -\ *$

Còn ở dạng tiền tố thì các toán tử sẽ đi trước các toán hạng. Khi đó biểu thức $5*(7+3)$ có dạng: $*\ 5\ -\ 7\ 3$

Ông cũng không định riêng đối với các dạng ký pháp này đâu ngoài là không cần thiết.

Nhiều bạn thắc khi định giá biểu thức số học thông thường thì bạn: trước hết chuyển các biểu thức dạng trung tố có đâu ngoài sang dạng hậu tố, sau đó mới tạo các chỗ trống máy để định giá biểu thức ở dạng hậu tố. Việc biến đổi từ dạng trung tố sang dạng hậu tố không khó khăn gì, còn việc định giá theo dạng hậu tố thì dễ dàng hơn, “máy móc” hơn so với dạng trung tố.

Để minh họa ta xét định giá của biểu thức sau:

$1\ 5\ +\ 8\ 4\ 1\ -\ -\ *$

Tổng cộng với biểu thức thông thường: $(1 + 5) * (8 - (4 - 1))$

Biểu thức này được đọc từ trái sang phải cho tới khi tìm ra một toán tử. Hai toán hạng được đọc cuối cùng, trước toán tử này, sẽ được kết hợp với nó. Trong ví dụ của chúng ta thì toán tử đầu tiên được đọc là $+$ và hai toán hạng tổng cộng với nó là 1 và 5, sau khi kết hợp biểu thức con này có giá trị là 6, thay vào ta có biểu thức rút gọn:

$6\ 8\ 4\ 1\ -\ -\ *$

Lưu ý từ trái sang phải, toán tử tiếp theo là $-$ và ta xác định được 2 toán hạng của nó là 4 và 1. Thực hiện phép toán ta có dạng rút gọn:

$6\ 8\ 3\ -\ *$

Lưu ý tiếp tục ta đi tới:

6 5 *

và cuối cùng thực hiện phép toán * ta có kết quả là 30.

Phương pháp đánh giá biểu thức hậu tố như trên đòi hỏi phải lưu trữ các toán hạng cho tới khi mặt toán tử được đọc, thời điểm điểm này hai toán hạng cuối cùng phải được tìm ra và kết hợp với toán tử này. Như vậy ở đây đã xuất hiện cách cho hoạt động “vào sau ra trước” nghĩa là ta sẽ phải sử dụng tới ngăn xếp để lưu trữ các toán hạng. Có mặt liên đến đến mặt toán tử thì hai giá trị sẽ được lấy ra từ ngăn xếp để áp dụng toán tử đó lên chúng và kết quả lại được đẩy vào ngăn xếp.

Giải thuật sau đây thể hiện các ý trên.

Void Dinh_Gia()

*/*giải thuật này sẽ dùng ngăn xếp S để lưu trữ các toán hạng được tính biểu thức*/*

1 – Do

//Đọc phần tử X tiếp theo trong biểu thức;

if (X là toán hạng) **call** PUSH(S, X)

else {

call POP(S, Y);

call POP(S, Z);

*/*tác động toán tử X lên hai toán hạng Y và Z rồi gán cho biến W*/*

W = Y (X) Z;

call PUSH(S, W);

}

while (gặp dấu kết thúc biểu thức);

2- POP(S, R);

write(R);

}

Điểm này đây là hình minh họa việc thực hiện giải thuật này với biểu thức:

1 5 + 8 4 1 - - *

Biểu thức		Ngăn xếp		Chú thích
1 5 + 8 4 1 - - *		1	← T	Đẩy 1 vào ngăn xếp

↑				
1 5 + 8 4 1 - - *		5	← T	Đẩy 5 vào ngăn xếp
↑		1		
+ 8 4 1 - - *		6	← T	Lấy 5 và 1 từ ngăn xếp cùng lần rồi đẩy kết quả vào ngăn xếp
↑				
		8	← T	
8 4 1 - - *		6		Đẩy 8 vào ngăn xếp
↑				
		4	← T	Đẩy 4 vào ngăn xếp
4 1 - - *		8		
↑		6		
		1	← T	Đẩy 1 vào ngăn xếp
1 - - *		4		
↑		8		
		6		
		3	← T	Lấy 1 và 4 từ ngăn xếp
- - *		8		Thực hiện $(4 - 1)$ rồi đẩy kết quả vào ngăn xếp
↑		6		
- *		5	← T	Lấy 3 và 8 từ ngăn xếp
↑		6		Thực hiện $(8 - 3)$ rồi đẩy kết quả vào ngăn xếp
*		30	← T	Lấy 5 và 6, thực hiện $(5 * 6)$ rồi đẩy kết quả vào ngăn xếp
↑				

Hình 3.17: Biểu diễn ghi nhớ thuật toán tính giá trị đa thức

4.2. Queue

4.2.1. Khái niệm

Một kiểu dữ liệu trừu tượng quan trọng khác được xây dựng trên cơ sở mô hình dữ liệu danh sách tuyến tính là hàng đợi. Hàng đợi là kiểu danh sách tuyến tính trong đó, phép bổ sung mới phần tử vào hàng đợi được thực hiện ở một đầu, gọi là đầu sau (rear) và phép loại bỏ mới phần tử được thực hiện ở đầu kia, gọi là đầu trước (front).

Như vậy, cấu trúc của hàng đợi là vào ở một đầu, ra ở đầu khác, phần tử vào trước thì ra trước, phần tử vào sau thì ra sau. Do đó, hàng đợi còn được gọi là danh sách kiểu FIFO (First In First Out). Trong thực tế ta cũng thấy có nhiều hình ảnh giống hàng đợi, chẳng hạn, hàng người chờ mua vé tàu, học sinh xếp hàng đi vào lớp v.v...

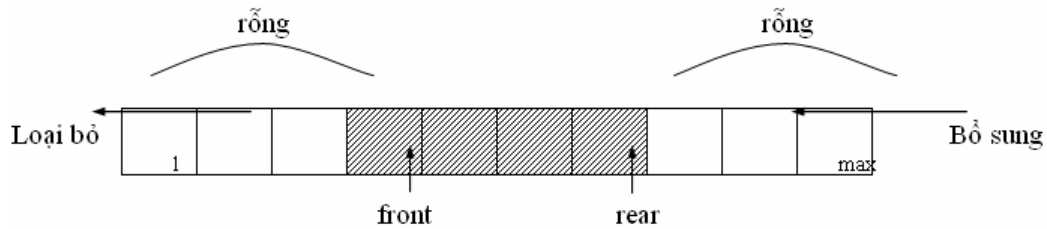
4.2.2. Cài đặt hàng đợi bằng mảng.

Ta có thể biểu diễn hàng đợi bằng mảng với việc sử dụng hai chỉ số front để chỉ vị trí đầu hàng đợi (đầu trước) và rear để chỉ vị trí cuối hàng đợi (đầu sau). Cấu trúc dữ liệu hàng đợi được biểu diễn như sau:

```
#define Max 100 ;
typedef struct Queue
    Int front, rear;
    Item E[Max];
};
Struct Queue Q; {Khai báo hàng đợi Q lưu trữ các phần tử của danh sách}
```

- Vị trí để loại bỏ phần tử gọi là Front.
- Vị trí để thêm vào phần tử gọi là Rear.

Trong cách cài đặt như trên, hàng đợi Q là mảng nullo Q.rear = -1, và hàng đợi đầynullo Q.rear = Max-1.



Hình 3.18: Mô hình biểu diễn hàng đợi

***) Các phép toán trên hàng đợi.**

Giống như ngăn xếp, khi thực hiện các thao tác với hàng đợi ta không được phép truy nhập tùy tiện vào các phần tử của hàng đợi, mà phải sử dụng các phép toán được biết đến với định nghĩa trên hàng đợi. Đó là các phép toán sau:

1. Khởi tạo hàng đợi rỗng

Procedure Initialize(Struct Queue *Q)

```
{
    Q->front = -1;
    Q->rear = -1;
}
```

2. Kiểm tra hàng đợi rỗng

int Empty(Struct Queue Q)

```
{
    Return (Q.rear == -1 ? 1:0);
}
```

3. Kiểm tra hàng đợi đầy

int Full(struct Queue Q)

```
{
    Return (Q.rear == Max-1 ? 1:0);
}
```

4. Bổ sung một phần tử mới vào đầu hàng đợi.

Khi bổ sung một phần tử mới (với thông tin lưu trong biến X) vào hàng đợi cần kiểm tra xem hàng có đầy không, nếu hàng chưa đầy thì bổ sung phần tử mới vào hàng, ngược lại việc bổ sung không được thực hiện.

```
int AddQ(struct Queue *Q; Item X)
```

```
{
    If (Full(*Q)==1)
        Return 0;
    Else
    {
        Q->rear = Q->rear + 1;
        Q->E[Q->rear] = X;
        Return 1;
    }
}
```

5. *Loại bỏ một phần tử ra khỏi hàng đợi.*

Khi loại bỏ một phần tử cần phải kiểm tra xem hàng đợi có rỗng không, nếu hàng đợi rỗng thì không thể thực hiện việc loại bỏ, ngược lại thì loại bỏ phần tử đầu hàng, nội dung của phần tử này được lưu trong biến X. Thêm nữa, khi loại bỏ, nếu hàng chỉ có một phần tử (nghĩa là hàng sẽ rỗng sau khi loại bỏ) thì cần khôi phục lại hàng.

Sau đây là thủ tục thực hiện việc loại bỏ.

```
int DeleteQ(struct Queue *Q; Item *X)
```

```
{
    If (Empty(*Q)==1) return(0);
    Else
    {
        *X = Q->E[Q->front];
        if (Q->front == Q->rear)
        {
            Q->front = -1; Q->rear = -1; {khôi phục lại hàng đợi}
        }
    }
}
```

```

        else    Q->front = Q->front + 1;
    Return(1);
}
}

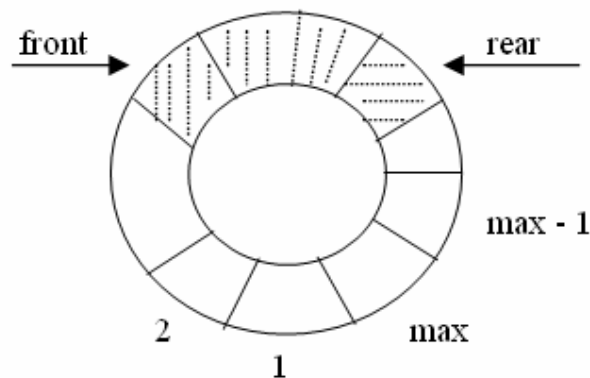
```

Trong 2 hàm trên t₁ sẽ dùng dùng hàm tr₁ và giá tr₁, nếu hàm tr₁ và giá tr₁ 1 thì là phép toán th₁ c₁ hi₁n thành công còn ngược lại là không thành công.

Nh₁n xét: Ph₁ng pháp cài đ₁t hàng đ₁i b₁i m₁ng v₁i hai ch₁s₁ nh₁ trên có nh₁ c₁ đ₁i m₁ n₁. Nếu phép lo₁i b₁ không th₁ng xuyên làm cho hàng r₁ng, thì các ch₁s₁ front và rear sẽ tăng liên t₁c và sẽ v₁ t₁ quá c₁ c₁a m₁ng. Hàng sẽ tr₁ thành đ₁y, m₁c dù các v₁ trí tr₁ng trong m₁ng có th₁ v₁n còn nh₁u (do v₁c lo₁i b₁ các ph₁n t₁ đ₁u hàng). Đ₁ tránh tình tr₁ng này, m₁i khi hàng đ₁i đ₁y ta l₁i ki₁m tra không gian nh₁ phía tr₁ c₁ hàng đ₁i, nếu còn tr₁ng thì đ₁y hàng đ₁i v₁ phía tr₁ c₁ đ₁ t₁o ra không gian nh₁ tr₁ng v₁ phía s₁au. Tuy nhiên v₁c này tiêu t₁n r₁t nh₁u th₁i gian.

4.2.3. Cài đ₁t hàng đ₁i b₁i m₁ng vòng tròn.

Đ₁ kh₁c ph₁c nh₁ c₁ đ₁i m₁ nêu trên, ng₁ i ta đ₁a ra ph₁ng pháp cài đ₁t hàng đ₁i b₁i m₁ng vòng tròn. Đó là m₁t m₁ng v₁i ch₁s₁ ch₁y trong mi₁n 0..Max-1, v₁i m₁i i = 0, 1, ..., Max - 1, ph₁n t₁ th₁ i c₁a m₁ng đ₁ tr₁ c₁ ph₁n t₁ th₁ i + 1, còn ph₁n t₁ th₁ Max-1 đ₁ tr₁ c₁ ph₁n t₁ đ₁u tiên, t₁c là các ph₁n t₁ c₁a m₁ng đ₁ c₁ x₁p thành vòng tròn (xem hình d₁ i).



Hình 3.20: Hàng đ₁i vòng tròn

Khi biểu diễn hàng bởi mảng vòng tròn, để biết khi nào hàng đầy, khi nào hàng rỗng ta cần đưa thêm vào biến count để đếm số phần tử trong hàng. Chúng ta có khai báo cấu trúc dữ liệu sau:

```
#define Max 100 ;  
Typedef struct Queue  
{  
    Int count;  
    Int front, rear;  
    Item E[Max];  
};  
Struct Queue Q;
```

Trong cách cài đặt này, hàng rỗng khi Q.count = 0, hàng đầy khi Q.count = Max.

Khi làm việc với mảng vòng tròn, cần lưu ý rằng, phần tử đầu tiên của mảng đi sau phần tử thứ max. Sau đây chúng ta sẽ viết các thủ tục bổ sung một phần tử vào hàng đợi, và loại bỏ một phần tử khỏi hàng đợi, các thủ tục khác dành lại cho bạn đọc.

```
Void AddQ(Struct Queue *Q;Item X);  
{  
    if (*Q.count == Max) printf(“\nHang day”);  
    else {  
        if (Q->rear == Max-1)  
            Q->rear = 0;  
        else  
            Q->rear = Q->rear + 1;  
        Q->E[Q->rear] = X;  
        Q->count = Q->count + 1;  
    }  
}  
  
Void DeleteQ(Struct Queue *Q;Item *X)  
{  
    if (Q->count == 0) printf(“\n hang rong”)
```

```

else {
    *X = Q->E[Q->front];
    if (Q->front == Q->rear)
    {
        Q->front = -1; Q->rear = -1;
    }
    else
        if (Q->front == max-1) Q->front = 0;
        else Q->front = Q->front + 1;
    Q->count = Q->count - 1;
}
}

```

Hàng đợi thi công dùng để thể hiện các “tuyến chờ” (waiting lines) trong mô phỏng, được biểu diễn trong các hệ mô phỏng (simulation), đó là các hệ mô hình hoá các quá trình đợi và ngược lại ta dùng mô hình này để nghiên cứu hoạt động của các quá trình.

4.2.4. Cài đặt hàng đợi bằng danh sách móc nối đơn.

Như ta đã biết, để tránh gặp việc truy nhập chung đến các thể hiện của một đầu (đỉnh). Vì vậy, việc cài đặt ngăn xếp bằng danh sách móc nối là khá tự nhiên. Chẳng hạn, với danh sách móc nối đơn thì có thể coi L như con trỏ trỏ đến đỉnh của ngăn xếp. Bổ sung một nút vào ngăn xếp chính là việc bổ sung một nút vào thành nút đầu tiên của danh sách, còn loại bỏ một nút ra khỏi ngăn xếp chính là loại bỏ nút đầu tiên của danh sách đang trỏ bởi L . Trong việc bổ sung với ngăn xếp dùng này không cần kiểm tra hiện trạng của ngăn xếp lưu trữ kịp.

Để với hàng đợi thì loại bỏ một đầu, còn bổ sung thì đầu kia. Nếu coi danh sách móc nối đơn như một hàng đợi thì việc loại bỏ một nút cũng giống như với ngăn xếp, nhưng bổ sung một nút thì phải thêm nút mới vào cuối hàng đợi, nghĩa là phải tìm đến nút cuối cùng. Trong trường hợp này, để lưu trữ danh sách ngược lại ta dùng hai con trỏ, một con trỏ trỏ vào nút đầu danh sách và

mặt con trỏ trỏ vào nút cuối danh sách. Cấu trúc dữ liệu biểu diễn hàng đợi như sau:

```

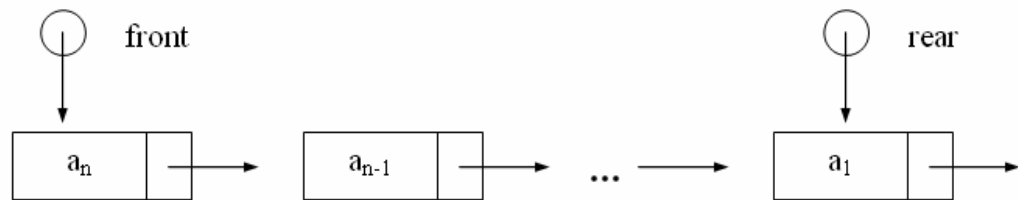
Struct Node
{
    Item Info;
    Struct Node *Next;
};

Struct Queue
{
    Struct Node *front;
    Struct Node *rear;
};

Struct Queue Q ;

```

Trong cách biểu diễn trên, front là con trỏ trỏ đến nút đầu hàng đợi, rear là con trỏ trỏ đến nút cuối hàng đợi.



Hình 3.21: Danh sách móc nối biểu diễn hàng đợi.

Với cách cài đặt này, hàng đợi được xem là không khi nào đầy. Hàng đợi rỗng khi $Q.front = NULL$.

Sau đây là các hàm và thủ tục thực hiện các phép toán trên hàng đợi:

1. Khởi tạo hàng đợi rỗng:

```

Void Create(Struct Queue *Q)
{
    *Q.front = NULL;
    *Q.rear = NULL;
}

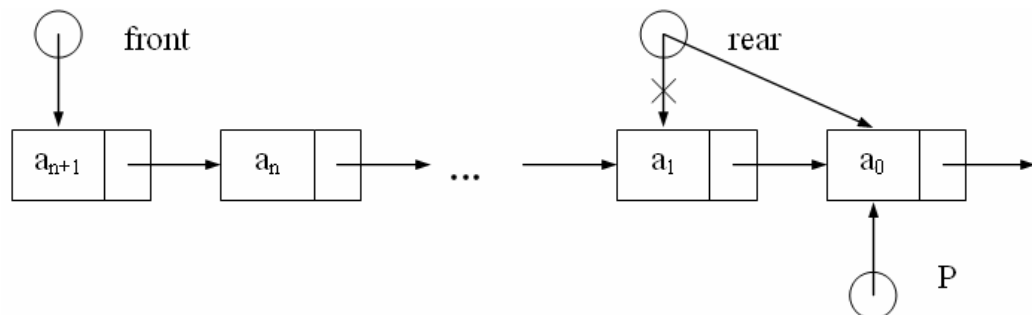
```

2. Kiểm tra hàng đợi trống:

```
Int Empty(Struct Queue Q)
{
    Return (Q.front == NULL? 1: 0);
}
```

3. Thêm phần tử vào cuối hàng đợi:

```
Void ADD(Struct Queue *Q; Item X );
    Struct Node *P;
    {
        P=(struct Node*)malloc(sizeof(struct Node));
        (P*).Info = X;
        (P*).Next = NULL;
        If ( Empty(*Q) ==1)
        {
            *Q.front = P;
            *Q.rear = P;
        }
        Else {
            *Q.rear ->Next = P;
            *Q.rear -> P;
        }
    }
```



Hình 3.22: Minh họa thao tác thêm vào.

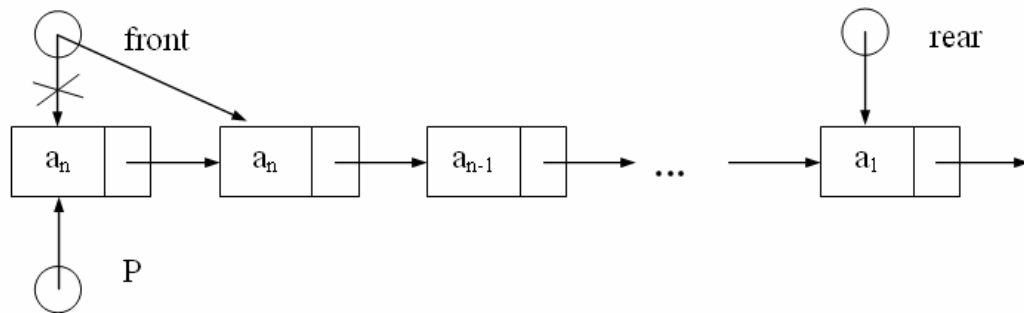
3. Lấy ra phần tử đầu hàng đợi:

```
Void Del(Struct Queue *Q; Item *X );
```

```

Struct Node *P;
{
If (Empty(*Q) == 1)
{
    printf(“\n Hang doi khong co phan tu nao!!!”);
    return;
}
Else {
    P = *Q.front;
    *X = (*Q -> front) -> Info;
    *Q -> front = (*Q -> front) ->Next;
    Free(P) ;
}
}

```



Hình 3.23: Minh họa thao tác lấy ra một phần tử .

Trên đây chúng ta đã xem xét một loại cấu trúc dữ liệu, được sử dụng rất phổ biến trong các ứng dụng, đó là danh sách tuyến tính với các ứng dụng khác nhau được cài đặt theo hai cách: bằng mảng (liên tiếp) và bằng con trỏ (liên tiếp móc nối), cùng với các phép toán xử lý thông thường trên mỗi loại. Tuy nhiên, để hiểu rõ hơn về nó, ta cần cài đặt một số ứng dụng như trên máy tính với mỗi loại.