ĐỀ CƯƠNG BÀI GIẢNG

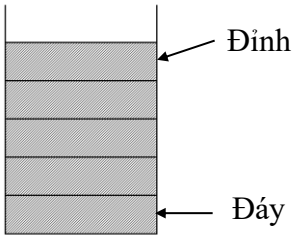
BÀI 4: DANH SÁCH TUYẾN TÍNH

4.5. Stack (Ngăn xếp)

4.5.1.1. Khái niệm

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

Có thể hình dung Ngăn xếp như cơ cấu của một hộp tiếp đạn. Việc đưa đạn vào hộp đạn hay lấy đạn ra khỏi hộp chỉ được thực hiện ở đầu hộp. Viên đạn mới nạp nằm ở đỉnh còn viên đạn nạp đầu tiên nằm ở đáy hộp.



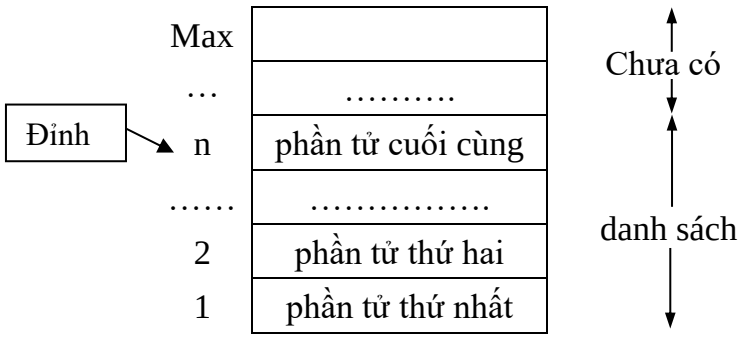
Hình 1: Ngăn xếp

4.5.1.2. Cài đặt ngăn xếp bởi mảng.

Giả sử danh sách được biểu diễn là một ngăn xếp, có độ dài tối đa là một số nguyên dương N nào đó, các phần tử của ngăn xếp có kiểu dữ liệu là ***Item***. ***Item*** có thể là các kiểu dữ liệu đơn, hoặc các kiểu dữ liệu có cấu trúc. Chúng ta biểu diễn ngăn xếp bởi một bản 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. Cấu trúc dữ liệu biểu diễn ngăn xếp được khai báo theo mẫu sau:

```
#define Max  N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct    Stack {
          Item E[Max];
          unsigned int top;
};
struct 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.



Hình 2. Mảng biểu diễn ngăn xếp.

Ví dụ: Đoạn 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.

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

```
void Initialize (struct   Stack *S)
{
          S->top = 0;
}
```

b. Kiểm tra ngăn xếp rỗng

```
int Empty ( struct Stack S)
{
          return  (S.top = = 0);
}
```

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

c. Kiểm tra ngăn xếp đầy

```
int Full ( struct  Stack S)
{
          return (S.top = = Max);
}
```

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

d. Thêm một phần tử mới vào đỉnh ngăn xếp

Để bổ sung phần tử X vào đỉnh của ngăn xếp S, trước hết kiểm tra xem S có đầy không. Nếu S đầy thì bổ sung không thực 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( struct Stack *S,  Item X)
{
        if  (Full(S))   return 0;
        else
        {
                S->top = S->top + 1;
                S->E[S->top] = X;
                return  1;
        }
}
```

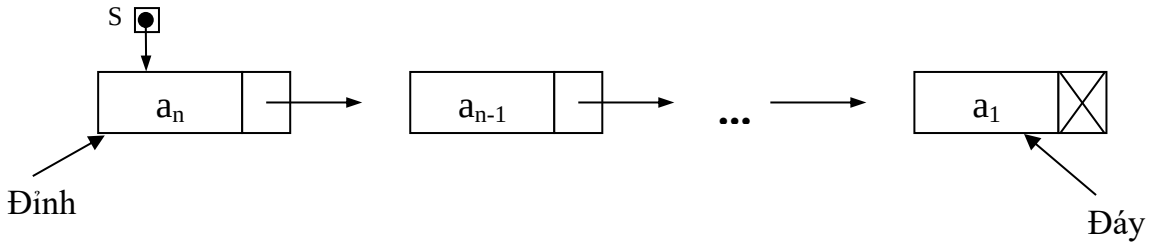
e. Loại bỏ phần tử ở đỉnh của ngăn xếp

Việc loại bỏ được thực hiện nếu S không rỗng, giá trị của phần tử bị loại bỏ được gán cho biến X. Hàm *POP()* trả về 1 nếu loại bỏ thành công, ngược lại trả về 0.

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

4.5.1.3. Cài đặt ngăn xếp bởi danh sách móc nối đơn.

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



Hình 3: 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;
        Node *Next;
};
struct Node *S;
```

Trong cách cài đặt này, ngăn xếp rỗng khi S = NULL. Ta giả sử việc cấp phát bộ nhớ động cho các phần tử mới luôn thực hiện. Do đó, ngăn xếp không bao giờ đầy và phép toán PUSH luôn thực hiện thành công.

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

a. Khởi tạo ngăn xếp rỗng:

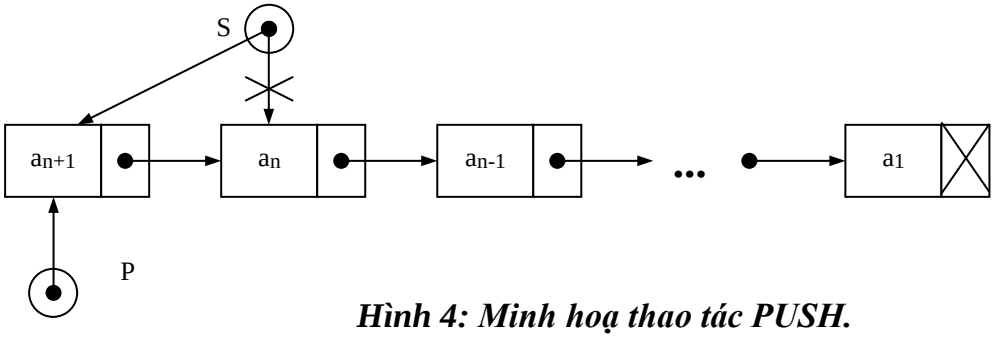
```
void  Create(struct Node **S)
{       *S = NULL;
}
```

b. Kiểm tra ngăn xếp rỗng:

```
int Empty(struct Node *S)
{       return (S = = NULL);
}
```

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

```
void PUSH(struct Node **S, Item X)
{
        struct Node *P;
        P = new Node;
        P->Infor = X;
        P->Next = NULL;
        if (*S= = NULL)     *S = P;
        else
        {       P->Next = *S;  *S = P;
        }
}
```



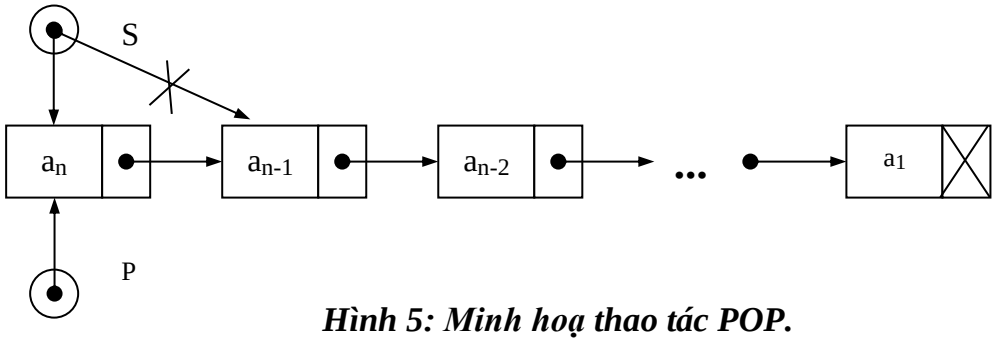
Hình 4: Minh hoạ thao tác PUSH.

d. Lấy ra một phần tử ở đỉnh ngăn xếp:

```
int POP(struct Node **S, Item *X)
{
```

```
        struct Node *P;
        if (Empty(*S))  return 0;
        else
        {
                P = *S;
                X = (*S)->Infor;
                *S = (*S)->Next;
                delete P;
                return 1;
        }
}
```

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

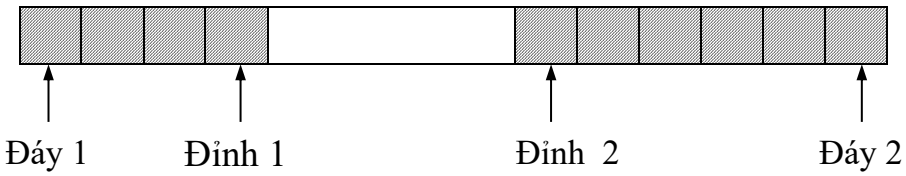


Hình 5: Minh hoạ thao tác POP.

4.5.1.4. 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 dự trữ cho ngăn xếp khác vẫn còn chỗ trống (tràn cục bộ). Làm thế nào để khắc phục được tình trạng này?

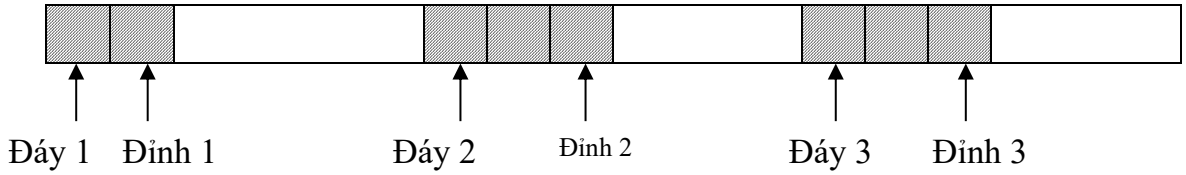
Nếu là hai ngăn xếp thì có thể giải quyết dễ dàng. Ta không qui định kích thước tối đa cho từng ngăn xếp nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta đặt hai ngăn xếp ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như hình dưới đây.



Hình 6: Hai ngăn xếp trên một không gian nhớ

Như vậy, có thể một ngăn xếp này dùng lấn sang quá nửa không gian dự trữ nếu như ngăn xếp kia chưa dùng đến. Do đó hiện tượng tràn chỉ xảy ra khi toàn bộ không gian nhớ 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 nhớ có thể chia đều cho cả 3, như hình dưới đây.



Hình 7: Ba ngăn xếp trên một không gian nhớ

Nhưng nếu có một ngăn xếp nào phát triển nhanh bị tràn trước mà ngăn xếp khác vẫn còn chỗ thì phải dồn 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 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.

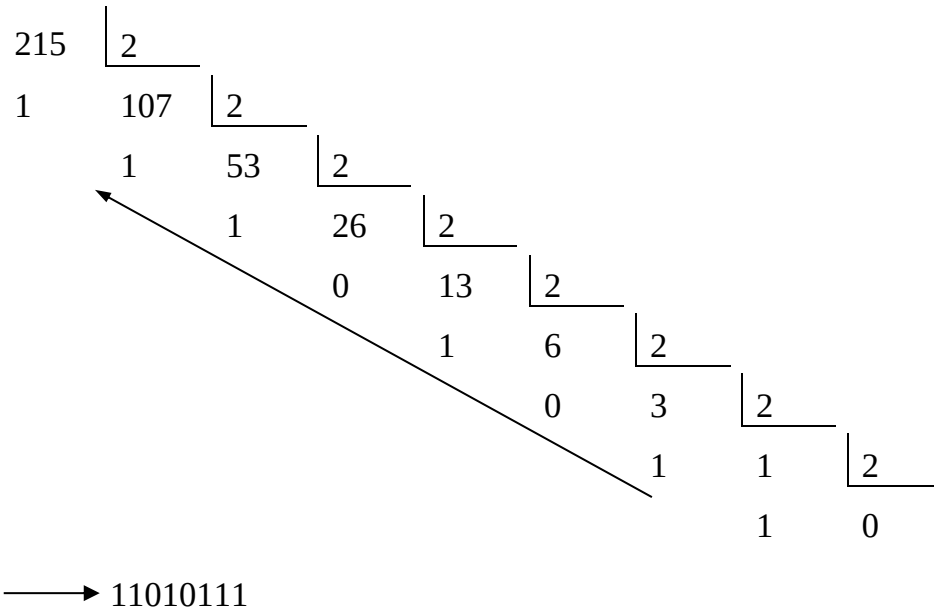
4.5.1.5. Một số ứng dụng của ngăn xếp.

a. Ứng dụng đổi cơ số.

Ta biết rằng dữ liệu lưu trữ trong bộ nhớ của máy tính đều được biểu diện 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 trước khi thực hiện các phép xử lý.

Khi đổi một số nguyên từ hệ thập phân sang hệ nhị phân 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:

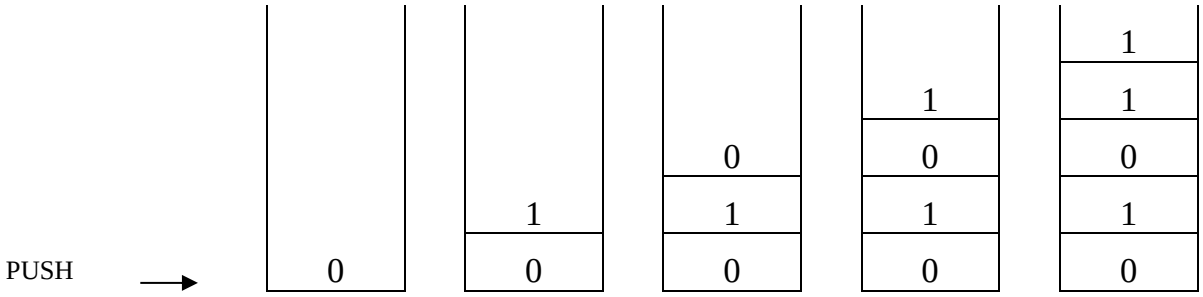


⟶ 11010111

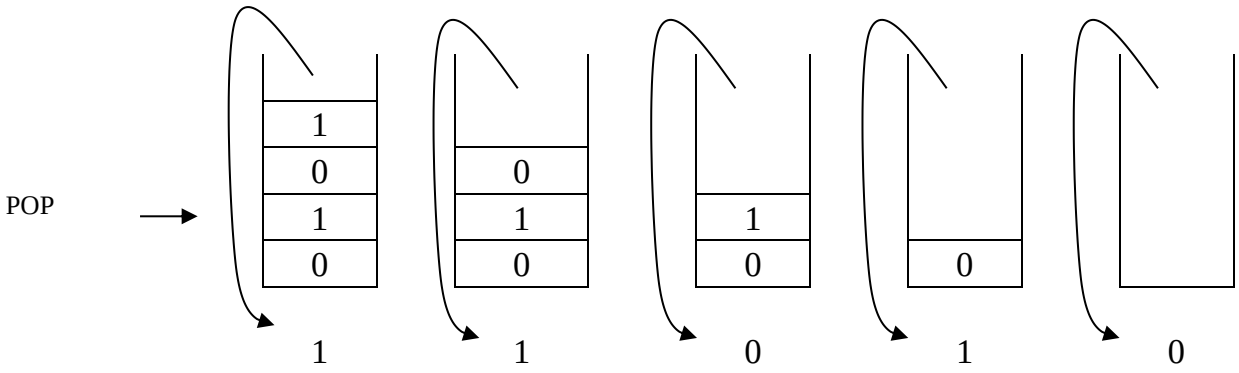
Ta thấy trong cách biến đổi này các số được tạo ra sau lại được hiển thị trước. Cơ chế sắp xếp này chính là cơ chế hoạt động của ngăn xếp. Để thực hiện biến đổi ta sẽ dùng một ngăn xếp để lưu trữ các số dư qua từng phép chia: Khi thực hiện phép chia thì nạp số dư vào ngăn xếp, sau đó lấy chúng lần lượt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

Ví dụ:

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



Hình 8: Mô tả hoạt động lưu trữ dữ liệu vào ngăn xếp.



Hình 9: Mô tả hoạt động lấy dữ liệu từ ngăn xếp.

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

```
#define  Max  16  //thực hiện đổi số nguyên có kích thước 2 byte
typedef  int Item; //Item là chữ số nhị phân
struct    Stack
{
        Item  E[Max];
        int top;
};
struct Stack  S;
/*S là ngăn chứa các số dư qua các phép chia trong quá trình chuyển đổi*/
```

Giải thuật sử dụng ngăn xếp thực hiện chuyển đổi số nguyên dương N từ hệ cơ số 10 sang hệ cơ số 2. Trong giải thuật này có sử dụng các phép toán trên ngăn xếp.

```
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 – cout<<"\nma nhi phan: ";
        while (!Empty(S))
        {
                call POP(S, R);
                cout<<R;
        }
}
```

b. Ứng dụng định giá biểu thức số học theo ký pháp nghịch đảo Balan.

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

Bộ dịch phải tạo ra các chỉ thị máy tương ứng 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 tương ứng của X, trong bộ nhớ.

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

Chẳng hạn: 5*(7+3)

Dấu ngoặc là cần thiết vì nếu viết 5*7 + 3 thì theo qui ước về thứ tự 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 Balan Lukasiewicz đã đưa ra dạng biểu thức số học theo ký pháp hậu tố (postfix notation) và tiền tố (prefix notation) mà được 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 khẳng định rằng đối với các dạng ký pháp này dấu ngoặc là không cần thiết.

Nhiều bộ dịch khi định giá biểu thức số học thường thực biện: trước hết chuyển các biểu thức dạng trung tố có dấu ngoặc sang dạng hậu tố, sau đó mới tạo các chỉ thị máy để

định giá biểu thức ở dạng hậu tố. Việc biến đổi từ dạng trung tố sang dạng 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 hoạ ta xét định giá của biểu thức sau:

1 5 + 8 4 1 - - *

tương ứng 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 vi dụ của chúng ta thì toán tử đầu tiên được đọc là + và hai toán hạng tương ứng với nó là 1 và 5, sau khi kết hợp biểu thực con này có giá trị là 6, thay vào ta có biểu thức rút gọn:

6 8 4 1 - - *

Lại đọc 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 - *

Lại tiếp tục ta đi tới:

6 5 *

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

Phương pháp định giá biểu thức hậu tố như trên đòi hỏi phải lư trữ các toán hạng cho tới khi một toán tử dược đọc, tại thời 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 cơ chế hoạt động "vào sau ra trước" nghĩa là ta sẽ phải sử dụng tới ngăn xếp dể lưu trữ các toán hạng. Cứ mỗi lần đọc được một toán tử thì hai giá trị sẽ được lấy ra từ ngăn xếp để áp đặt 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 sử dụng ngăn xếp S để lưu trữ các toán hạng đọc từ biểu thức*/
    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);
        POP(S, R);
        cout<<R;
}
```

Dưới đây là hình ảnh 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 1	Đẩy 5 vào ngăn xếp
+ 8 4 1 - - * ↑	6 ← T	Lấy 5 và 1 từ ngăn xếp cộng lại rồi đẩy kết quả vào ngăn xếp
8 4 1 - - * ↑	8 ← T 6	Đẩy 8 vào ngăn xếp
4 1 - - * ↑	4 ← T 8 6	Đẩy 4 vào ngăn xếp
1 - - * ↑	1 ← T 4 8 6	Đẩy 1 vào ngăn xếp
- - * ↑	3 ← T 8 6	Lấy 1 và 4 từ ngăn xếp Thực hiện (4 - 1) rồi đẩy kết quả vào ngăn xếp
- * ↑	5 ← T 6	Lấy 3 và 8 từ ngăn xếp 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 10: Biểu diễn giải thuật tính giá trị đa thức

4.6. Queue (Hàng đợi)

4.6.1. 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 một phần tử vào hàng đợi được thực hiện ở một đầu, gọi là lối sau (rear) và phép loại bỏ một phần tử được thực hiện ở đầu kia, gọi là lối trước (front).

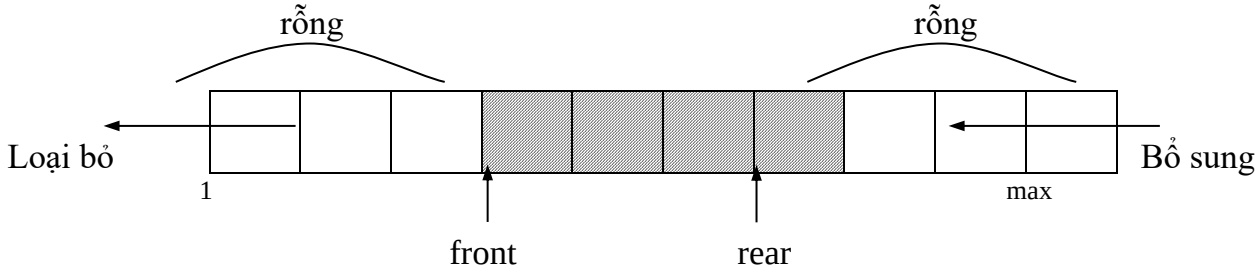
Như vậy, cơ cấu của hàng đợi là vào ở một đầu, ra ở đầu khác, phần tử vào trước thì ra trước, phần tử vào sau thì ra sau. Do đó, hàng đợi còn được gọi là danh sách kiểu ***FIFO*** (*First In First Out*). Trong thực tế ta cũng thấy có những hình ảnh giống hàng đợi, chẳng hạn, hàng người chờ mua vé tầu,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

4.6.2. Cài đặt hàng đợi bởi mảng.

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

```
#define  max  N
//Khai báo kiểu dữ liệu Item nếu cần
struct   Queue
{
        unsigned int front, rear;
        Item  E[max];
};
struct Queue Q; //Khai báo hàng đợi Q lưu trữ các phần tử của danh sách
```

Trong cách cài đặt như trên, hàng đợi Q là rỗng nếu Q.rear = 0, và hàng đầy nếu Q.rear = Max.



Hình 11: Mảng 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 đặc biệt được định nghĩa trên hàng đợi. Đó là các phép toán sau:

a. Khởi tạo hàng đợi rỗng

```
void Initialize(struct Queue *Q)
{       Q->front = 1; Q->rear = 0;
}
```

b. Kiểm tra hàng đợi rỗng

```
int Empty(struct Queue Q)
{
        return (Q.rear = =0);
}
```

c. Kiểm tra hàng đợi đầy

```
int Full(struct Queue Q)
{
        return (Q.rear = = max);
}
```

d. Bổ sung một 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)) return 0;
        else
        {
                Q->rear = Q->rear + 1;
                Q->E[Q->rear] = X;
                return 1;
        }
}
```

Hàm AddQ thực hiện bổ sung phần tử mới vào cuối hàng đợi, hàm trả về 1 nếu bổ sung thành công, và trả về 0 nếu ngược lại.

e. 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 ở 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 khởi tạo lại hàng.

Sau đây là thủ tục thực hiện việc loại bỏ.

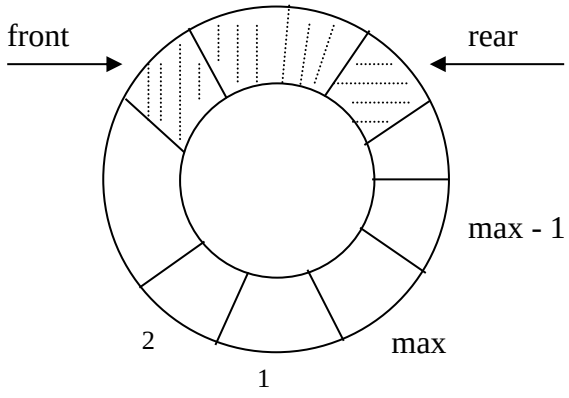
```
int DeleteQ(struct Queue *Q, Item X)
{
        if  (Empty(*Q))
                return 0;
        else
        {
                X = Q->E[Q->front];
                if (Q->front = = Q->rear)
                {
                        Q->front = 1; Q->rear = 0;  //khởi tạo lại hàng đợi
                }
                else    Q->front = Q->front + 1;
                return 1;
        }
}
```

Hàm DeleteQ thực hiện lấy ra một phần tử ở đầu hàng đợi, hàm trả về 1 nếu phép lấy ra thành công, ngược lại trả về 0.

Nhận xét: Phương pháp cài đặt hàng đợi bởi mảng với hai chỉ số như trên có nhược điểm lớn. Nếu phép loại bỏ không thường xuyên làm cho hàng rỗng, thì các chỉ số front và rear sẽ tăng liên tục và sẽ vượt quá cỡ của mảng. Hàng sẽ trở thành đầy, mặc dù các vị trí trống trong mảng có thể vẫn còn nhiều (do việc loại bỏ các phần tử ở đầu hàng). Để tránh tình trạng này, mỗi khi hàng đợi đầy ta lại kiểm tra không gian nhớ phía trước hàng đợi, nếu còn trống thì đẩy hàng đợi về phía trước để tạo ra không gian nhớ trống ở phía sau. Tuy nhiên việc này tiêu tốn rất nhiều thời gian.

4.6.3. Cài đặt hàng đợi bởi mảng vòng tròn.

Để khắc phục nhược điểm nêu trên, người ta đưa ra phương pháp cài đặt hàng đợi bởi mảng vòng tròn. Đó là một mảng với chỉ số chạy trong miền 1..max, với mọi i = 1, 2, ..., max – 1, phần tử thứ i của mảng đi trước phần tử thứ i + 1, còn phần tử thứ max đi trước phần tử đầu tiên, tức là các phần tử của mảng được xếp thành vòng tròn (xem hình dưới).



Hình 12: Hàng đợi 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  N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct    Queue
{
        int count;
        int front, rear;
        Item  E[max];
};
struct Queue Q;
```

Với 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 cài đặt thao tác bổ sung một phần tử vào hàng đợi, và loại bỏ một phần tử khỏi hàng đợi, các thao tác khác dành cho bạn đọc.

```
void AddQ(struct Queue *Q, Item X)
{
        if  (Q->rear = = max) cout<<"\nHang day"<<endl;
        else
        {
                if  (Q->rear = = max)
                        Q->rear = 1;
                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) cout<<"\nhang rong"<<endl;
```

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

Hàng đợi thường dùng để thực hiện các "tuyến chờ" (waiting lines) trong xử lý động, đặc biệt trong các hệ mô phỏng (simulation), đó là các hệ mô hình hoá các quá trình động và người ta dùng mô hình này để nghiên cứu hoạt động của các quá trình ấy.

4.6.4. Cài đặt hàng đợi bởi danh sách móc nối đơn.

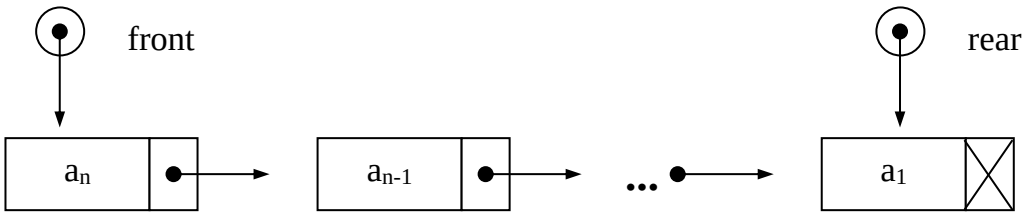
Như ta đã biết, đối với ngăn xếp việc truy nhập chỉ được thực hiện ở một đầu (đỉnh). Vì vậy, việc cài đặt ngăn xếp bằng danh sách móc nối là khá tự nhiên. Chẳng hạn, với danh sách móc nối đơn trỏ bởi L thì có thể coi L như con trỏ trỏ tới đỉnh của ngăn xếp. Bổ sung một nút vào ngăn xếp chính là việc 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 dạng này không cần kiểm tra hiện tượng tràn như với ngăn xếp lưu trữ kế tiếp.

Đối với hàng đợi thì loại bỏ ở một đầu, còn bổ sung thì ở 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 thực gắn nút mới vào cuối hàng đợi, nghĩa là phải tìm đến nút cuối cùng. Trong trường hợp này, để lưu trữ danh sách người ta dùng hai con trỏ, một con trỏ trỏ vào nút đầu danh sách và một con trỏ trỏ vào nút cuối danh sách.

Cấu trúc dữ liệu biểu diễn hàng đợi như sau:

```
//Khai báo kiểu dữ liệu Item (nếu cần)
struct  Node
{       Item  Infor;
        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 13: 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:

a. Khởi tạo hàng đợi rỗng:

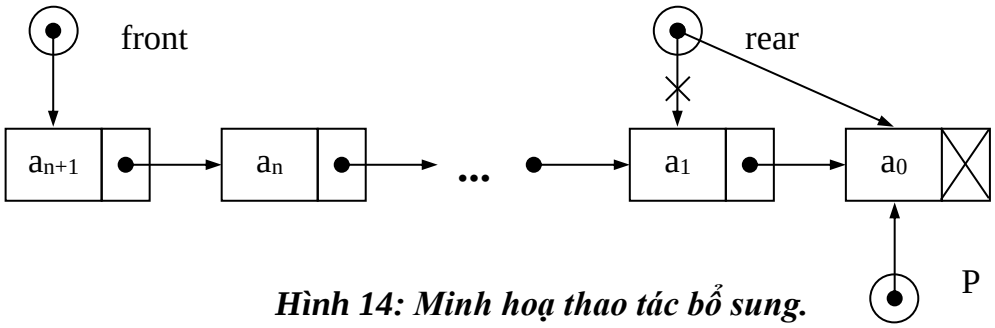
```
void Create(struct Queue *Q)
{
        Q->front = NULL;
        Q->rear = NULL;
}
```

b. Kiểm tra hàng đợi rỗng:

```
int Empty(struct Queue Q)
{
        return (Q.front = = NULL);
}
```

c. Bổ sung một phần tử vào cuối hàng đợi:

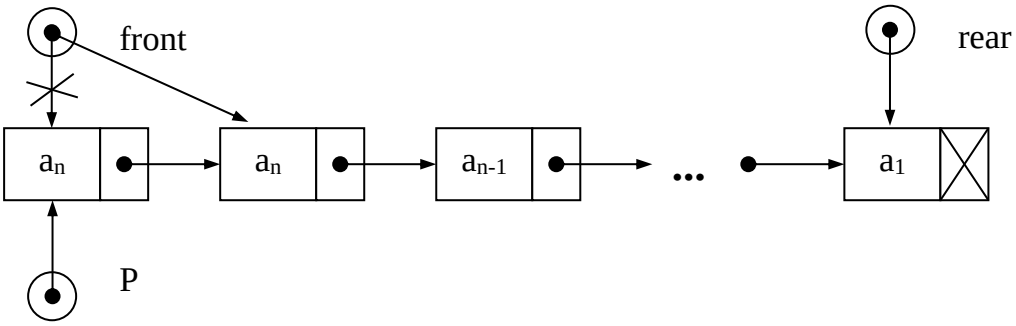
```
void ADD(struct Queue *Q, Item X)
{
  struct Node *P;
        P = (struct Node*)malloc(sizeof(struct Node)) ;
        P->Infor = X;
        P->Next = NULL;
        if  (Empty(*Q))
        {       Q->front = P;
                Q->rear = P;
        }
        else    {
                        Q->rear->Next = P;
                        Q->rear = P;
        }
}
```



Hình 14: Minh hoạ thao tác bổ sung.

d. Lấy ra một phần tử ở đầu hàng đợi:

```
int Del(struct Queue *Q, Item X)
{
  struct Node P;
  if  (Empty(*Q)) return 0;
  else {
        P = Q->front;
        X = Q->front->Infor;
        Q->front = Q->front->Next;
        free(P);
        return 1;
  }
}
```



Hình 15: Minh hoạ 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 dạng khác nhau được cài đặt theo hai cách: bằng mảng (lưu trữ kế tiếp) và bằng con trỏ (lưu trữ móc nối), cùng với các phép toán xử lý tương ứng trên mỗi loại. Bạn đọc cũng được tìm hiểu hai cách thức lưu trữ đặc biệt đó là danh sách kiểu ngăn xếp và kiểu hàng đợi. Mỗi loại danh sách và cách cài đặt chúng có những ưu và nhược điểm khác nhau. Tuy nhiên, để hiểu rõ hơn về nó, ta cần cài đặt một số ứng dụng nhỏ trên máy đối với mỗi loại. Khi cài đặt hãy lựa chọn cấu trúc lưu trữ phù hợp với dữ liệu của bài toán.