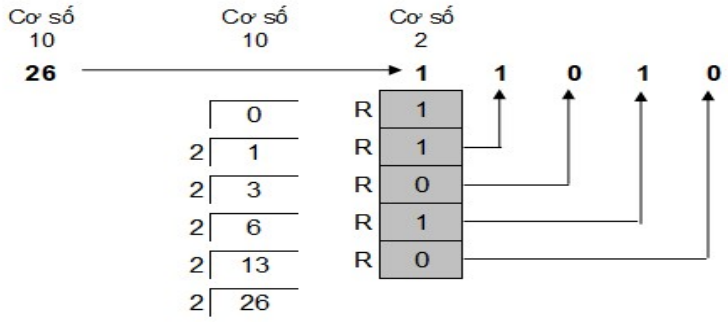
ỨNG DỤNG CỦA DANH SÁCH: NGĂN XẾP VÀ HÀNG ĐỢI

I. NGĂN XẾP

Các mục dữ liệu được lưu trữ trong bộ nhớ máy tính bằng biểu diễn nhị phân. Điều đó có nghĩa là các số nguyên xuất hiện trong một lệnh của chương trình hay trong một tập tin dữ liệu phải được chuyển đổi sang biểu diễn cơ số 2. Một thuật toán để thực hiện phép chuyển đổi này là dùng phép chia liên tiếp cho 2 với các số dư là các chữ số nhị phân từ phải sang trái.

Ví dụ: Biểu diễn cơ số 2 của 26 là 11010 như phép tính sau:



Hình 3.1: Chuyển đổi từ dạng thập phân sang nhị phân

Để ý rằng các bit tạo nên dạng nhị phân của 26 được tính theo thứ tự ngược lại, từ phải sang trái. Như vậy các bit phải được hiển thị theo cách "*được tạo ra cuối cùng - được hiển thị đầu tiên*".

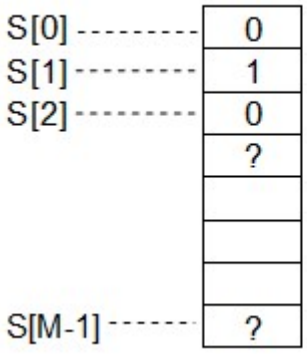
Kiểu xử lý "*vào sau ra trước*" này xuất hiện trong nhiều ứng dụng, vì vậy một cấu trúc dữ liệu trừu tượng thể hiện ý tưởng này là cần thiết. Cấu trúc như thế này được gọi là ngăn xếp. Thực chất nó là một danh sách trong đó các phép toán thêm vào, bớt đi đều được thực hiện ở một đầu gọi là đỉnh của ngăn xếp.

Một số phép toán cơ bản của một ngăn xếp:

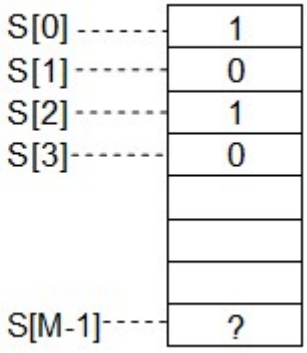
1. Khởi tạo một ngăn xếp rỗng.
2. Kiểm tra một ngăn xếp có rỗng hay không?
3. Kiểm tra một ngăn xếp có đầy hay không?
4. Xóa phần tử tại đỉnh ngăn xếp
5. Trả về nội dung phần tử tại đỉnh ngăn xếp
6. Thêm một phần tử mới vào đỉnh ngăn xếp.

Cài đặt ngăn xếp bằng mảng (danh sách đặc)

Ta có thể dùng mảng để lưu trữ những mục dữ liệu của ngăn xếp, mỗi phần tử chiếm một vị trí trong mảng và vị trí 0 phục vụ như đỉnh của ngăn xếp. Ví dụ, trong bài toán chuyển đổi sang cơ số 2 ở phần trước, nếu ba số dư đầu tiên 0, 1, 0 được đưa vào ngăn xếp, ngăn xếp sẽ có dạng như hình 3.2:



Hình 3.2: Ngăn xếp sau khi thêm 0, 1, 0

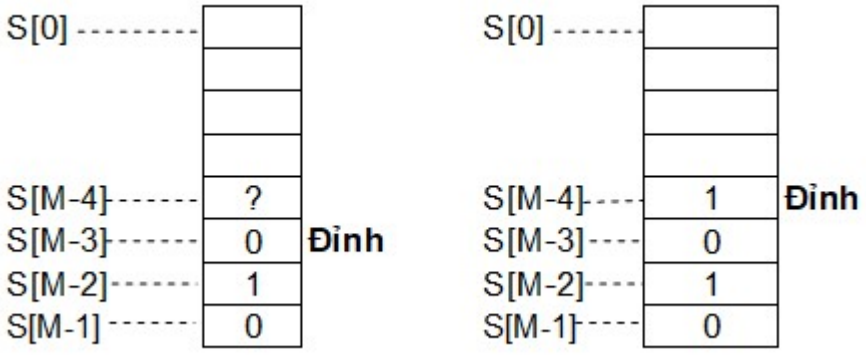


Hình 3.3: Các phần tử trong mảng dịch chuyển ra sau để thêm phần tử mới

Tuy nhiên, việc đưa số dư tiếp theo vào ngăn xếp đòi hỏi phải dịch chuyển các phần tử của mảng tại các vị trí 0, 1, 2 sang các vị trí 1, 2, 3 theo thứ tự đó (hình 3.3).

Tương tự như thế, khi một mục được lấy ra từ ngăn xếp, tất cả các phần tử của mảng phải được đẩy lên một vị trí để cho phần tử đỉnh nằm ở vị trí 0.

Việc dịch chuyển các phần tử của mảng như trên là quá mất thời gian và không cần thiết; ta có thể tránh được điều này bằng cách lật ngược ngăn xếp để cho các phần tử chứa từ đáy trở lên và dùng một biến Top để ghi lại vị trí của đỉnh ngăn xếp trong mảng (hình 3.4).



Hình 3.4: Ngăn xếp cài đặt bằng mảng với 1 biến ghi vị trí đỉnh

Cài đặt các phép toán trên ngăn xếp

1. Khai báo ngăn xếp

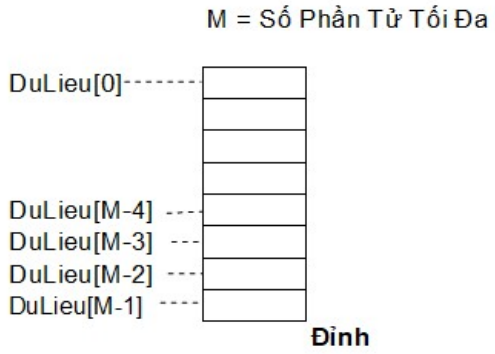
```
#define SoPhanTu 100
typedef struct{
      int DuLieu[SoPhanTu];
      int Dinh;
}NganXep;
```

2. Khởi tạo ngăn xếp rỗng (hình 3.5)

```
void khoitao(NganXep *pS){
      pS->Dinh = SoPhanTu;
}
```

3. Kiểm tra ngăn xếp rỗng (hình 3.5)

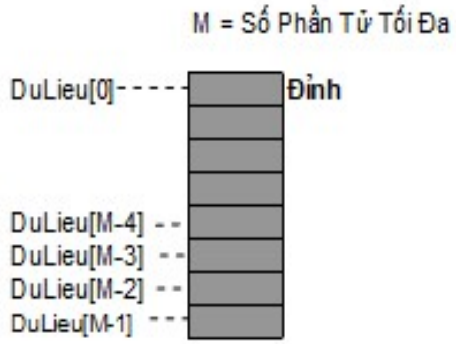
```
int ktRong(NganXep S){
      return S.Dinh == SoPhanTu;
}
```



Hình 3.5: Ngăn xếp rỗng

4. Kiểm tra ngăn xếp đầy (hình 3.6)

```
int krDay(NganXep S){
      return S.Dinh ==0;
}
```



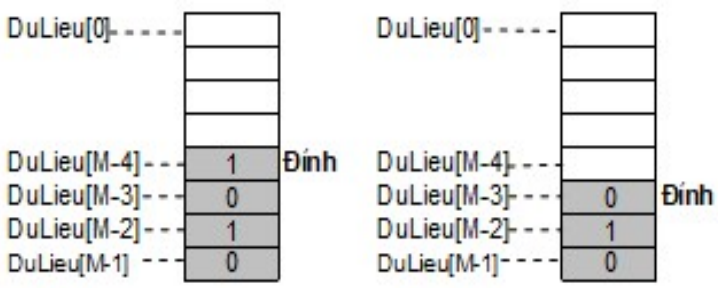
Hình 3.6: Ngăn xếp đầy

5. Trả về nội dung tại đỉnh ngăn xếp (hình 3.7)

```
int giatriDinh(NganXep S){
      return S.DuLieu[S.Dinh];
}
```

6. Xóa nội dung tại đỉnh ngăn xếp (hình 3.7)

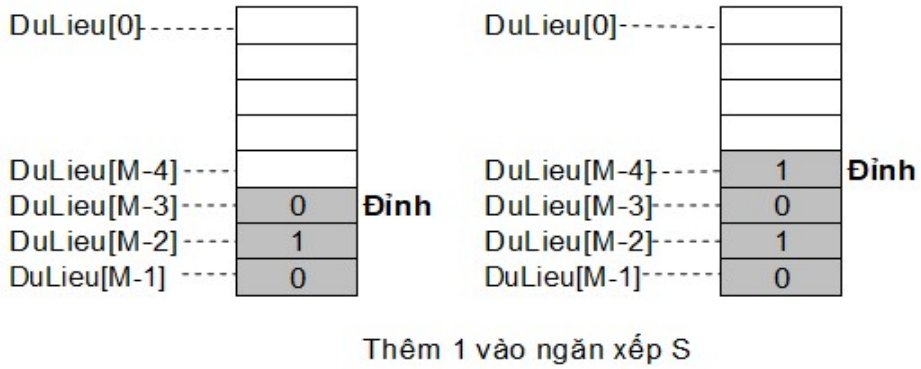
```
void xoaDinh(NganXep *pS){
      if (!ktRong(*pS))
            pS->Dinh++;
}
```



Hình 3.7: Truy xuất nội dung phần tử tại đỉnh và xóa đỉnh ngăn xếp bằng cách di chuyển vị trí đỉnh 1 vị trí

7. Thêm phần tử vào đỉnh ngăn xếp (hình 3.8)

```
void them(int X, NganXep *pS){
      if(!ktDay(*pS)){
            pS->Dinh--;
            pS->DuLieu[pS->Dinh] = X;
      }
}
```



Hình 3.8: Minh họa việc thêm 1 phần tử vào ngăn xếp, sau khi thêm vị trí đỉnh di chuyển 1 vị trí

8. Hiển thị ngăn xếp

```
void hienthi(NganXep *pS){
      while(!ktRong(*pS)){
            printf("%d\t",giatriDinh(*pS));
            xoaDinh(pS);
      }
      printf("\n");
}
```

Hàm hiển thị ngăn xếp ở trên là điển hình của phép toán duyệt qua 1 ngăn xếp. Giải thuật cơ bản của phép duyệt này như sau:

```
void Visit_Stack(Stack S){
    while(S ≠ Φ){
      <Xử lý phần tử tại đỉnh ngăn xếp S >
      <Xóa phần tử tại đỉnh ngăn xếp S >
   }
}
```

Sau khi duyệt qua 1 ngăn xếp, ngăn xếp trở nên rỗng

Ứng dụng của ngăn xếp: Ký pháp nghịch đảo Ba Lan.

Trong đa số ngôn ngữ lập trình, các biểu thức số học được viết theo ký pháp trung tố (infix notation), nghĩa là mỗi ký hiệu của một phép toán nhị phân được đặt giữa các toán hạng. Nhiều bộ dịch trước hết chuyển các biểu thức trung tố này sang ký pháp hậu tố (postfix hay tiền tố - prefix), trong đó mỗi toán tử đi sau các toán hạng, sau đó mới tạo ra các chỉ thị máy để đánh giá biểu thức hậu tố này. Các biểu thức hậu tố, một cách tổng quát, dễ đánh giá hơn là các biểu thức trung tố.

Khi dùng ký pháp trung tố, dấu ngoặc là cần thiết để tăng độ ưu tiên của toán tử. Tuy nhiên, những năm đầu 1950 nhà logic học Ba Lan Jan Lukasiewicz đã phát hiện rằng dấu ngoặc là không cần thiết trong ký pháp hậu tố, còn được gọi là ký pháp nghịch đảo Ba Lan (Reverse Polish Notation - RPN).

Ví dụ biểu thức trung tố 2 * (3+4) được viết dạng RPN như sau:

2 3 4 + *

Giải thuật chuyển từ dạng trung tố sang dạng RPN như sau:

1. Khởi động một ngăn xếp rỗng

2. while (Không xảy ra lỗi và chưa đạt đến kết thúc của biểu thức trung tố)

a. Đọc *Token* (hằng số, biến số, toán tử số học, các dấu ngoặc trái và ngoặc phải) tiếp theo trong biểu thức trung tố

b. Nếu phần tử (*Token*) là:

i. Một dấu ngoặc trái : đưa nó vào ngăn xếp

ii. Một dấu ngoặc phải : lấy ra và hiển thị các phần tử của ngăn xếp cho đến khi dấu ngoặc trái được đọc (nếu ngăn xếp là rỗng thì xảy ra lỗi).

iii. Một toán tử :

Nếu ngăn xếp là rỗng hay *Token* được ưu tiên hơn phần tử ở đỉnh ngăn xếp, đẩy *Token* vào ngăn xếp

Nếu khác, lấy ra và hiển thị phần tử đỉnh của ngăn xếp ; lặp lại việc so sánh *Token* với phần tử ở đỉnh ngăn xếp

iv. Một toán hạng : hiển thị nó

3. Khi đạt đến kết thúc của biểu thức trung tố, lấy ra và hiển thị các phần tử của ngăn xếp cho đến khi ngăn xếp rỗng.

II. HÀNG ĐỢI

Có thể tưởng tượng hàng đợi như là một hàng người đang chờ tính tiền ở một siêu thị, một dãy các máy bay đang chờ hạ cánh ở một sân bay, hay một dãy các công việc trong mạng máy tính đang chờ một thiết bị ra nào đó, chẳng hạn là máy in. Trong các ví dụ này, các mục sẽ được phục vụ theo thứ tự khi chúng đến; nghĩa là mục đầu tiên trong hàng sẽ được phục vụ đầu tiên. Như vậy hàng đợi là một cấu trúc "*Vào trước, ra trước*".

Hàng đợi là một dạng danh sách đặc biệt ở đó việc lấy ra được thực hiện tại 1 đầu của hàng đợi gọi là đầu (Front), và việc thêm vào tại đầu kia gọi là đuôi (Rear).

Cài đặt hàng đợi bằng mảng

Ta dùng một mảng để lưu trữ các phần tử của hàng đợi và duy trì 2 biến : Front ghi lại vị trí của phần tử có thể lấy ra trong mảng (vị trí đầu tiên của hàng đợi) ; và Rear lưu giữ vị trí trong mảng ở đó một phần tử có thể thêm vào được, vị trí sau vị trí cuối cùng của hàng đợi.

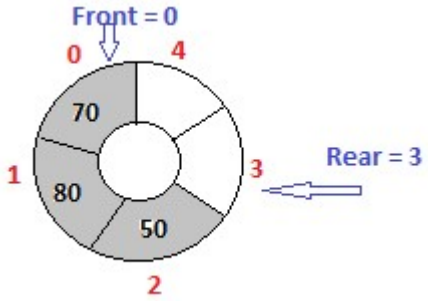
Một phần tử được lấy ra khỏi hàng đợi bằng cách tìm phần tử trong mảng ở vị trí Front và tăng Front lên 1. Một mục được thêm vào mảng bằng cách lưu nó vào vị trí Rear trong mảng và tăng Rear lên 1 (nếu Rear không vượt quá độ dài mảng).

Một khó khăn của cách cài đặt này là các phần tử có xu hướng "*dịch sang phải*" mảng. Để giải quyết tình trạng này ta:

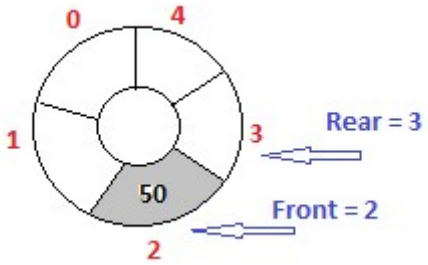
- Tịnh tiến các phần tử trong mảng về đầu
- Xem mảng như là vòng tròn, phần tử tiếp theo cũng là phần tử cuối cùng. Điều này có thể thực hiện bằng phép chia dư MaxSize với Front và Rear.

Chẳng hạn, ban đầu là hàng đợi rỗng, lần lượt thực hiện các phép toán sau :

- Thêm 70, 80, 50 vào hàng : hình 3.9 :

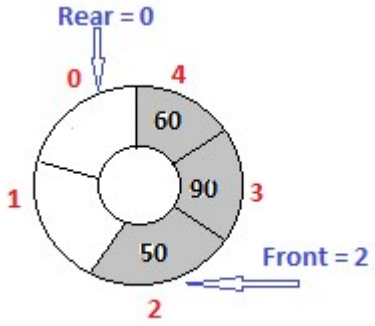


Hình 3.9: Thêm 70, 80, 50

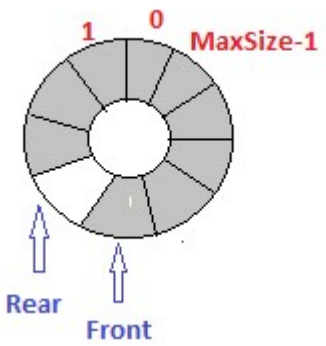


Hình 3.10: Xóa 2 phần tử trong hàng

- Xóa 2 phần tử đầu hàng đợi : hình 3.10
- Thêm 90, 60 vào hàng : hình 3.11



Hình 3.11: Thêm 70, 90, 60



Hình 3.12: Hàng đợi đầy

Cài đặt hàng đợi bằng ***mảng vòng*** lưu trong tập tin CircularQueue.c :

a. Khai báo hàng đợi:

Để cài đặt hàng đợi, một struct gồm một mảng chứa các phần tử của hàng đợi và 2 trường Front và Rear để ghi lại vị trí đầu và vị trí sau phần tử cuối cùng.

```
#define MaxSize 100
typedef struct {
      int Data[MaxSize];
      int Front, Rear;
}Queue;
```

b. Khởi tạo một hàng đợi rỗng

```
void makenullQueue(Queue *pQ){
      pQ->Front=pQ->Rear=0;
}
```

c. Kiểm tra hàng đợi rỗng hay không ?

Nếu hàng đợi có một phần tử, nó sẽ ở vị trí Front của mảng và Rear ở vị trí trống tiếp theo. Nếu xóa phần tử này, Front được tăng thêm 1 và có cùng giá trị với Rear. Như vậy để kiểm tra hàng đợi có rỗng hay không, chỉ cần kiểm tra Front == Rear hay không ?

```
int emptyQueue(Queue Q){
      return Q.Front == Q.Rear;
}
```

d. Kiểm tra hàng đợi có đầy không ?

Khi hàng đợi gần đầy như hình 3.12, tức hàng chỉ còn 1 vị trí trống. Nếu 1 phần tử được thêm vào ô trống này, Rear được tăng thêm 1 và sẽ có cùng giá trị với Rear. Tuy nhiên điều kiện Front == Rear chỉ ra hàng đợi rỗng. Do đó ta không thể phân biệt được hàng rỗng và hàng đầy. Để vượt qua, ta duy trì một vị trí trống trong mảng ; lúc đó điều kiện hàng đầy sẽ là : (Rear+1)%MaxSize == Front.

```
int fullQueue(Queue Q){
      return (Q.Rear + 1) % MaxSize == Q.Front;
}
```

e. Thêm 1 phần tử vào hàng đợi

```
void enQueue(int X, Queue *pQ){
      if (!fullQueue(*pQ)){
            pQ->Data[pQ->Rear] = X;
            pQ->Rear = (pQ->Rear+1)%MaxSize;
      }
}
```

f. Trả về nội dung phần tử đầu hàng đợi và xóa phần tử đầu hàng.

```
void deQueue(Queue *pQ, int *pTop){
      if (!emptyQueue(*pQ)){
            (*pTop) = pQ->Data[pQ->Front];
            pQ->Front = (pQ->Front+1)%MaxSize;
      }
}
```

g. Hiển thị một hàng đợi

```
void printQueue(Queue *pQ){
      while(!emptyQueue(*pQ)){
            int top;
            deQueue(pQ, &top);
            printf("%d\t",top);
      }
      printf("\n");
}
```

Hàm hiển thị hàng đợi ở trên là điển hình của phép toán duyệt qua 1 hàng đợi. Giải thuật cơ bản của phép duyệt này như sau:

```
void Visit_Queue (Queue Q){
      while(Q ≠ Φ){
        <Xử lý phần tử tại đầu hàng Q >
        <Xóa phần tử đầu hàng Q>
   }
}
```

Sau khi duyệt qua 1 hàng đợi, hàng đợi trở nên rỗng