Chương III
DANH SÁCH

1. ĐỊNH NGHĨA

- Định nghĩa
  - *Danh sách là một dãy hữu hạn các phần tử cùng loại được xếp theo một thứ tự tuyến tính.*

Danh sách L gồm các phần tử $a_1, a_2, .., a_n$

$$L=(a_1, a_2, .., a_n)$$

a_i*: p.tử thứ i của danh sách,*

n chiều dài của danh sách

1. ĐỊNH NGHĨA

- Danh sách con
- *Cho danh sách $L=(a_1, a_2, .., a_n)$,*

  Danh sách $L'=(a_l,..a_m)$, $1 \le l \le m \le n$ được gọi là danh sách con của danh sách L

  Ví dụ: L=(6, 7, 9, 3, 5, 1,8), L'=(9,3,5)
- Các thao tác trên danh sách

Tạo lập Bổ sung Loại bỏ	Sắp thứ tự Tìm kiếm Ghép danh sách	Trộn danh sách Sao chép danh sách …

1. ĐỊNH NGHĨA

- Danh sách đặc

❑Các phần tử của danh sách nằm liên tiếp trong bộ nhớ.

– Tổ chức lưu trữ: mảng

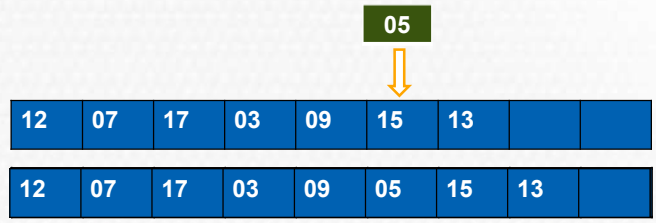
➢ Chú ý: Điểm khác biệt: Cấu trúc mảng và mô hình danh sách: mảng số phần tử cố định, số phần tử của danh sách thay đổi theo các thao tác chèn, xóa

- Danh sách liên kết:

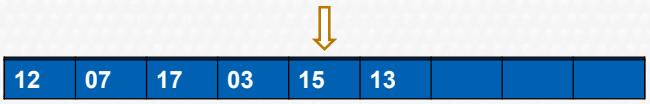
❑Các phần tử của danh sách liên kết với nhau qua thành phần chứa địa chỉ.

2. DANH SÁCH ĐẶC

- Chèn thêm phần tử



- Xóa phần tử

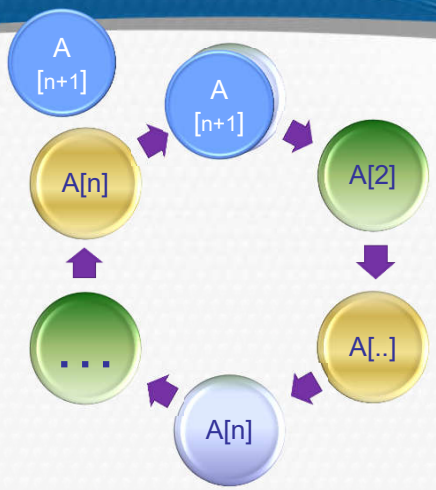


2. DANH SÁCH ĐẶC

- Xác định địa chỉ của phần tử
  - Mảng 1 chiều: Địa chỉ phần tử thứ i
    - Add(i) = l_0+(i-1)d
  - Mảng 2 chiều A_{nxm}: Địa chỉ phần tử thứ (i,j)
    - Add(i,j) = l_0+(i-1)m.d +(j-1).d
- Tổ chức lưu trữ:
  - Mảng
  - Chú ý trường hợp tràn (overflow):

    Tổ chức lưu trữ vòng

2. DANH SÁCH ĐẶC



2. DANH SÁCH ĐẶC

- Ưu nhược điểm của danh sách đặc:

➢ Ưu:

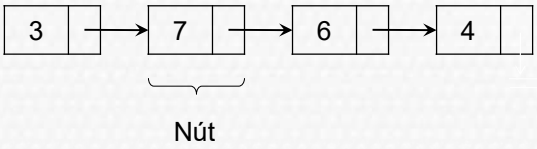
- Truy xuất các phần tử trực tiếp theo vị trí
- Có các thuật toán hiệu năng cao thao tác trên mảng: tìm kiếm, sắp xếp..
- Mật độ sử dụng 100%

➢ Nhược

- Các thuật toán chèn, xóa có độ phức tạp O(n)
- Lãng phí không gian nhớ khi có nhiều phần tử mang cùng giá trị.

3. DANH SÁCH LIÊN KẾT

- 3.1 Danh sách liên kết đơn



- Các phần tử của một danh sách liên kết không cố định

➡ không gian nhớ cấp phát cho ds liên kết phải được cấp phát động

3. DANH SÁCH LIÊN KẾT

- Cấp phát động, biến kiểu con trỏ

Ô nhớ cấp phát động là những ô nhớ được cấp phát và thu hồi bằng lệnh trong chương trình. Để quản lý các ô nhớ cấp phát động sử dụng biến con trỏ:

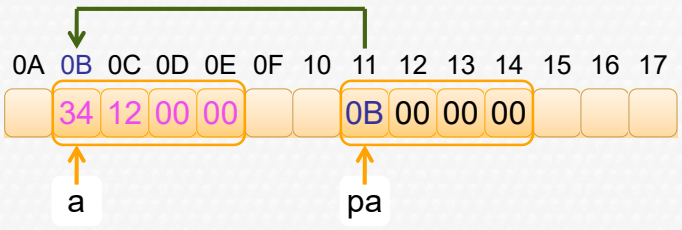
- *Biến kiểu con trỏ dùng để lưu trữ địa chỉ của các biến khác. Mỗi biến kiểu con trỏ quản lý một ô nhớ cấp phát động có kiểu xác định.*

3. DANH SÁCH LIÊN KẾT

- Khai báo kiểu và biến con trỏ trong C

```
<kiểu dữ liệu> *<tên biến con trỏ>;
```

- Ví dụ: int *pa, a; // {a=0x1234; pa=&a;}
- p là biến con trỏ, dùng để chứa địa chỉ của một vùng nhớ (*p) chứa một số nguyên



3. DANH SÁCH LIÊN KẾT

- Sử dụng từ khóa typedef

```
typedef <kiểu dữ liệu> *<tên kiểu con trỏ>;

<tên kiểu con trỏ> <tên biến con trỏ>;
```

- Ví dụ

```
typedef int *pint;
int *p1;
pint p2, p3;
```

- Lưu ý khi khai báo kiểu dữ liệu mới
  - Giảm bối rối khi mới tiếp xúc với con trỏ.
  - Nhưng dễ nhầm lẫn với biến thường.

3. DANH SÁCH LIÊN KẾT

- Tạo vùng nhớ động để lưu trữ dữ liệu:

  <biến con trỏ> = (kiểu dữ liệu biến con trỏ *) malloc (sizeof(kiểu dữ liệu biến con trỏ));

  Ví dụ: `p = (int*)malloc(sizeof(int));`

- Hằng NULL:

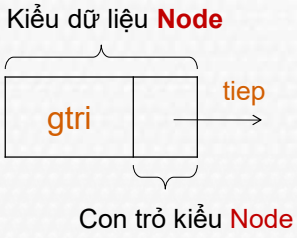
  Khi biến con trỏ không mang địa chỉ của vùng nhớ nào (rỗng), được gán mang giá trị NULL

- Xóa vùng nhớ động

  free(<biến con trỏ>)

3. DANH SÁCH LIÊN KẾT

- a. Khai báo cấu trúc nút




```
struct nut {
    int gtri;
    struct nut *tiep;
}
typedef struct nut Node;
```

3. DANH SÁCH LIÊN KẾT

- b. Khai báo biến con trỏ chứa đ/c nút đầu danh sách

```
Node *dau;
```

- c. Khởi tạo giá trị ban đầu

```
Node *khoitaoDS(void) {
      return NULL;
}
```

- d. Tạo danh sách

Nhập vào một dãy số nguyên, kết thúc việc nhập khi nhập giá trị 0. Các số nguyên được lưu thành một ds liến kết đơn.

3. DANH SÁCH LIÊN KẾT

```
Node *nhapDS_First(Node *dau){
  int x;
  Node *new;
  do {
     printf("Nhap 1 so nguyen (0:dung) ");
     scanf("%d",&x);
     if (x!=0) {
          new = (Node*)malloc(sizeof(Node));
          new->gtri = x;
          new->tiep = dau;
          dau = new;
     }
  }  while (x!=0);
  return dau;
}
```

3. DANH SÁCH LIÊN KẾT

- e. Duyệt danh sách

C1:

```
void duyetDS(Node *dau) {
        Node *tam;
        tam = dau;
        while (tam != NULL) {
              // Xử lý nút tam;
              tam = tam->tiep;
        }
    }
```

C2:

```
void duyetDSdq (Node *dau) {    // đệ quy
        if (dau != NULL)  {
              // Xử lý nút dau;
              duyetdsdq(dau->tiep);
        }
    }
```


3. DANH SÁCH LIÊN KẾT

- e1. Xem danh sách

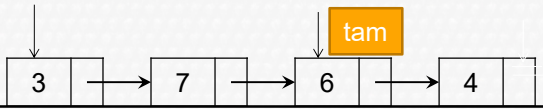
```
void xemDS(Node *dau){
     Node *tam;
     tam = dau;
     while (tam != NULL){
           printf("%d ",tam->gtri);
           tam = tam->tiep;
     };
}
```

3. DANH SÁCH LIÊN KẾT

- e2. Tìm nút đầu tiên mang giá trị x

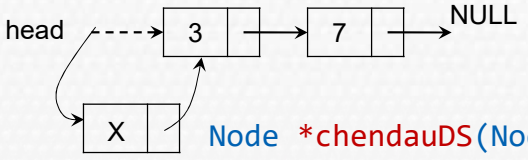
```
Node *timDS(Node *dau, int x, int *pos) {
        Node *tam;
        tam = dau;
        *pos = -1;
        while (tam != NULL) && (tam->gtri != x){
              tam = tam->tiep;
              *pos++;
        }
        if (tam == NULL) *pos == -1
        return tam;
}
```

Vị trí nút tìm thấy



3. DANH SÁCH LIÊN KẾT

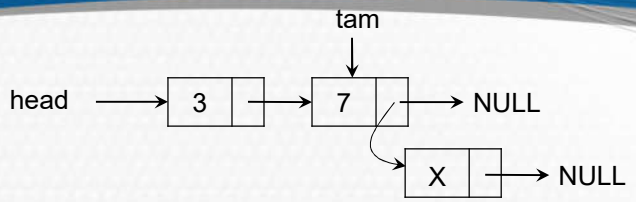
- f. Chèn một nút vào danh sách
- f1. Chèn một nút vào đầu danh sách:




```
Node *chendauDS(Node *dau, int x){
        new = (Node*)malloc(sizeof(Node));
        new->gtri = x;
        new->tiep = dau;
        dau = new;
        return dau;
}
```

3. DANH SÁCH LIÊN KẾT

- f2. Chèn một nút vào cuối danh sách:




```
Node *chencuoiDS(Node *dau, int x){
      new = (Node*)malloc(sizeof(Node));
      new->gtri = x;
      new->tiep = NULL;
      if (dau == NULL) dau = new;
      else {
         while (tam->tiep != NULL) tam = tam->tiep;
         tam->tiep = new;
      }
      return dau;
}
```


3. DANH SÁCH LIÊN KẾT

- g3. Chèn một nút vào sau nút thứ tự pos

3. DANH SÁCH LIÊN KẾT

- h. Xóa nút
  - h1. Xóa nút đầu ds
  - h2. Xóa nút cuối ds
  - h3. Xóa nút thứ tự pos

Thao tác tương tự như chèn

3. DANH SÁCH LIÊN KẾT

i. Sắp xếp danh sách, Ý tưởng: Buble Sort

```
Node *BubleSort(Node *dau) {
        Node *p,*q;
        int tam;
        p = dau;
        while (p->tiep != NULL) {
                q = p->tiep;
                while (q!=NULL) {
                        if (p->gtri > q->gtri) {
                                tam = p->gtri;
                                p->gtri = q->gtri;
                                q->gtri = tam;
                        }
                        q = q->tiep;
                }
        p = p->tiep;
        }
        return dau;
}
```

3. DANH SÁCH LIÊN KẾT

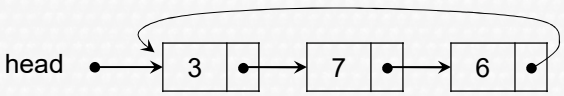
j. Đảo ngược danh sách:

```
Node *daonguoc(Node *dau) {
    Node *p,*q;
    q = NULL;
    while (dau!=NULL) {
            p = dau;
            dau = dau->tiep;
            p->tiep = q;
            q = p;
    }
    return p;
}
```


3. DANH SÁCH LIÊN KẾT

2. Danh sách liên kết vòng:

Nút cuối của danh sách liên kết vòng không trỏ đến NULL mà trỏ về lại nút đầu.



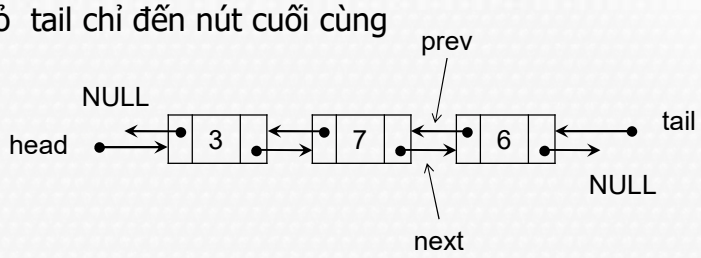
3. DANH SÁCH LIÊN KẾT

3. Danh sách liên kết kép:

Nút của danh sách liên kết kép gồm 2 con trỏ

- next : trỏ đến nút đứng sau
- prev: trỏ đến nút đứng trước

Con trỏ tail chỉ đến nút cuối cùng



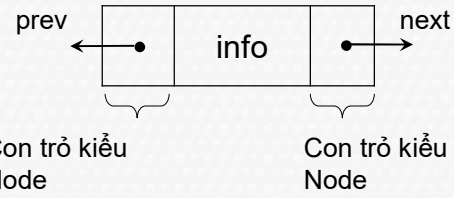
Page ▪ 101

101

3. DANH SÁCH LIÊN KẾT

Cấu trúc nút của danh sách liên kết kép:

```
struct Node {
    int info;
    struct Node *prev;
    struct Node *next;
};
```



Page ▪ 102

102

3. DANH SÁCH LIÊN KẾT

a. Ưu điểm:

- Không thực hiện các thao tác dời mảng khi thêm hay xóa phần tử.
- Không hao phí bộ nhớ.

b. Nhược điểm:

- Không truy cập trực tiếp phần tử bằng chỉ số.

4. ỨNG DỤNG DS

A. Ngăn Xếp (Stack)

1) Khái niệm ngăn xếp
2) Ngăn xếp thực hiện bằng mảng
3) Ứng dụng của ngăn xếp

B. Hàng Đợi (Queue)

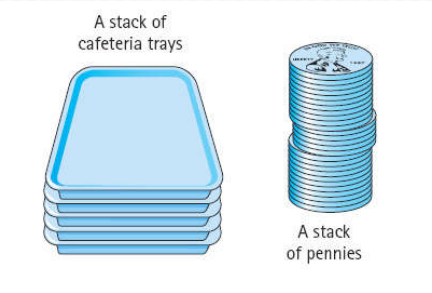
4) Khái niệm hàng đợi
5) Hàng đợi thực hiện bằng mảng
6) Ứng dụng của hàng đợi

a. Ngăn Xếp (Stack)

1. Khái niệm ngăn xếp:

Ngăn xếp là danh sách đặc biệt: thêm và xóa phần tử được thực hiện theo nguyên tắc **vào sau ra trước** (LIFO – Last In First Out)

Ví dụ: chồng đĩa



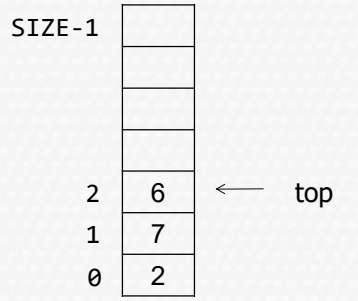
Thao tác trên Stack

- Push(x): thêm phần tử x vào Stack
- Pop(x) : lấy ra phần tử từ Stack cho vào biến x.
- View(x): xem phần tử kế tiếp sẽ được lấy ra.

2) Biểu diễn ngăn xếp bằng mảng

- ***Ngăn xếp các số nguyên:***

```
#define SIZE 20
struct Stack {
    int a[SIZE];      // Stack có kích thước là SIZE
    int top;          // vị trí của đầu Stack
};
typedef  struct Stack StackType;
```



top= -1 : stack rỗng
top= SIZE-1 : stack đầy

▪ Cài đặt stack số nguyên

```
void Init(StackType *s) {
  s->top = -1;
}
int Push(StackType *s, int x) {
  if (s->top < SIZE-1) {  // stack chưa đầy
    s->top++;             // vị trí phần tử mới
    s->a[s->top] = x;     // đưa phần tử mới vào stack
    return 1;             // thêm thành công
  } else   return 0;      // stack đầy
}
```

```
int Pop(StackType *s, int *x) {
   if (s->top == -1)              // stack rỗng
      return 0;
   else {
      *x = s->a[s->top];          // lấy phần tử khỏi stack
      s->top = s->top - 1;        // chỉ đến phần tử tiếp theo
      return 1;                   // Pop thành công
   }
}
int main(int argc, char *argv[]) {
   StackType s;
   int x;
   Init(&s);
   Push(&s, 2);
   Push(&s, 3);
   while (Pop(&s, &x)==1) printf("%d ", x);    return 0;
}
```

phần tử của Stack được đưa vào x

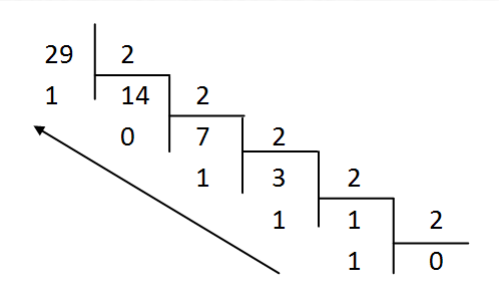
Giống hàm Pop, xem nhưng không lấy phần tử khỏi stack

```
int View(StackType *s, int *x) {
   if (s->top == -1)
         return 0;
   else {
         *x = s->a[s->top];
         return 1;
   }
}
```

3) Ứng dụng của ngăn xếp

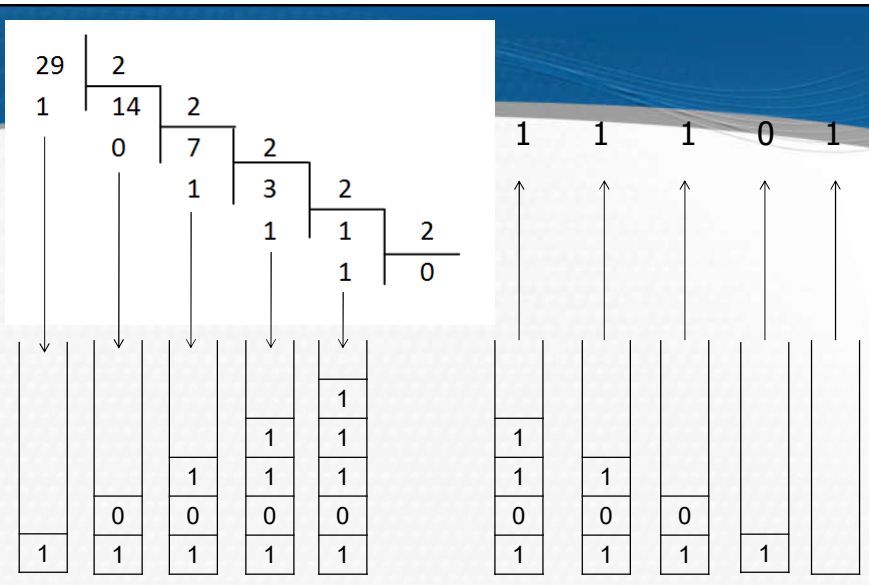
(Đổi số thập phân sang nhị phân)

Ví dụ: đổi số 29 sang nhị phân



Kết quả: 11101

Page ▪ 111



Page ▪ 112

```
int main(int argc, char *argv[]) {
        StackType s;
        int n,c;
        Init(&s);

        printf("Nhap vao so nguyen n: ");
        scanf("%d", &n);
        while (n>0) {
                Push(&s, n%2);    // đưa phần dư vào stack
                n = n/2;          // tính phần thương
        }
        printf("So nhi phan: ");
        while (Pop(&s, &c))
                printf("%d", c);
        return 0;
}
```


4) Ứng dụng của ngăn xếp

(Biểu thức hậu tố - ký pháp Ba Lan - *Polish notation*)

Jan Łukasiewicz -1920

Biểu thức trung tố:

Toán hạng	Toán tử	Toán hạng

Ví dụ: 6+7, A*B, 3^8, A+B*C - D

Để thay đổi thứ tự tính toán, dùng dấu ngoặc: (A+B)*(C – D)

b) Biểu thức hậu tố:

Toán hạng	Toán hạng	Toán tử

Ví dụ: 6 7 +, A B *, 3 8 ^,

Trung tố	Hậu tố	Bỏ ngoặc
A + B * C	A (B C *) +	A B C * +
(A + B) * C	(A B +) C *	A B + C *

Tính chất của biểu thức hậu tố:

- Được tính từ trái qua phải
- Không có thứ tự ưu tiên giữa các phép tính → không cần dấu ngoặc

Page ▪ 115

Ví dụ: tính giá trị biểu thức: 1 2 3 * + 4 -

Lần tính	Kết quả
1	1 6 + 4 -
2	7 4 -
3	3

Bài tập:

1. Tính: 1 2 3 4 - * +
2. Tính: 1 5 + 8 4 1 - - *

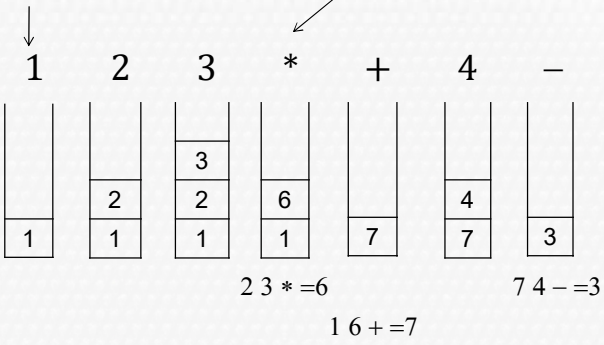
Page ▪ 116

b) Dùng Stack tính giá trị biểu thức hậu tố:

Ví dụ: tính giá trị biểu thức 1 2 3 * + 4 -

Toán tử: - Lấy 2 phần tử từ stack, tính.
- Đưa kết quả vào lại stack

Toán hạng: đưa vào stack



Bài tập:

1. Tính: 1 2 3 4 - * +
2. Tính: 1 5 + 8 4 1 - - *

b) Chuyển đổi biểu thức trung tố thành hậu tố:

Xét biểu thức $A + (B * E - C * D / F)$

Page ▪ 119

Toán hạng: → kết quả

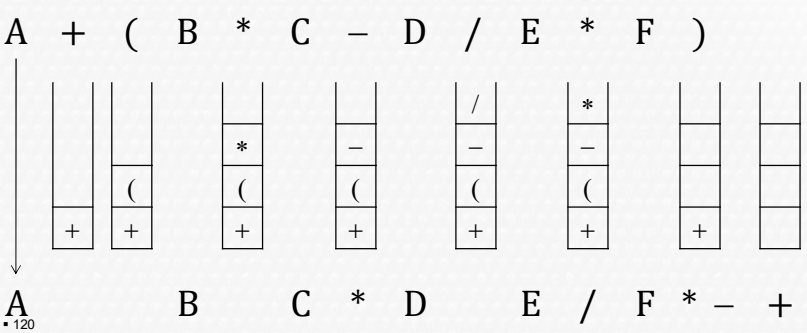
Kết thúc: pop hết stack

Toán tử: push stack

Dấu): pop stack cho đến dấu (

Dấu (: push stack

Độ ưu tiên ≤ toán tử ở stack: pop stack → kết quả



Page ▪ 120

Các thao tác thực hiện khi duyệt biểu thức trung tố

- Nếu là toán hạng: ghi ra biểu thức hậu tố kết quả
- Nếu là dấu (: push stack
- Nếu là dấu) : pop stack cho đến khi gặp dấu (, ghi ra biểu thức kết quả
- Nếu kết thúc biểu thức trung tố: pop hết stack, ghi ra biểu thức kết quả
- Nếu là toán tử:
  - Nếu toán tử trên đỉnh stack có độ ưu tiên không thấp hơn toán tử đang xét của biểu thức → pop stack, ghi ra biểu thức kết quả
  - Push toán tử vào stack

Bài tập đổi biểu thức trung tố thành hậu tố

1) A / (B + C * D - E) * F
2) (A - B) * (C - D * E + F) + G

b. Hàng Đợi (Queue)

1. Khái niệm hàng đợi:

Hàng đợi là danh sách đặc biệt: thêm và xóa phần tử được thực hiện theo nguyên tắc **vào trước ra trước** (FIFO – First In First Out)

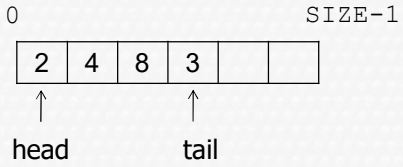
Ví dụ: xếp hàng

Thao tác trên Queue:

- Push(x): thêm một phần tử x vào Queue
- Pop(x) : lấy ra phần tử x ra khỏi Queue
- View(x): xem phần tử kế tiếp.

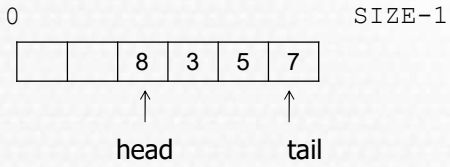
2) Biểu diễn hàng đợi bằng mảng

```
#define SIZE 20
struct Queue {
  int a[SIZE];    // Queue có kích thước là SIZE
  int head, tail; // Vị trí của đầu và cuối của queue
};
typedef struct Queue QueueType;
```



Vấn đề 1:

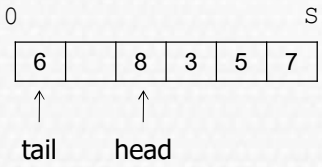
Khi pop Queue: nếu giữ head cố định và dồn mảng → tốn kém → tăng giá trị head sau khi pop



Vấn đề 2:

Nếu tail chỉ đến cuối mảng: cần push queue?

→ cho tail chỉ về đầu mảng



Các trường hợp của head và tail:

- Nếu head = tail: Queue có 1 phần tử
- Nếu (tail + 1) mod SIZE = head: Queue đầy
- Queue rỗng? Head = tail = -1

Cài đặt Queue

```
void Init(QueueType *q){
   q->head = q->tail = -1;
}

int Push(QueueType *q, int x) {
   if ((q->tail + 1)%SIZE == q->head)   // Queue đầy
          return 0;
   else {
          q->tail = (q->tail + 1)%SIZE;
          q->a[q->tail] = x;
          if (q->head == -1) q->head = 0;
          return 1;
   }
}
```

nếu trước đó queue rỗng cần cập nhật head

```
int Pop(QueueType *q, int *x){
  if (q->head == -1)  return 0;
  else {
        *x = q->a[q->head];
        if (q->head == q->tail)
              q->head = q->tail = -1;
        else
              q->head = (q->head + 1)% SIZE;
        return 1;
  }
}
```

Queue rỗng

Nếu phần tử lấy ra là pt cuối thì cập nhật queue rỗng

Vị trí phần tử tiếp theo sẽ được Pop

Page ▪ 129

```
int View(QueueType *q, int *x){
  if (q->head == -1) return 0;
  else {
        *x = q->a[q->head];
        return 1;
  }
}
```

Page ▪ 130

BÀI TẬP

Dùng stack, viết chương trình phân tích 1 số thành thừa số nguyên tố theo thứ tự lớn trước nhỏ sau.
Ví dụ n = 3960, hiển thị: 3960= 11*5*3*3*2*2*2