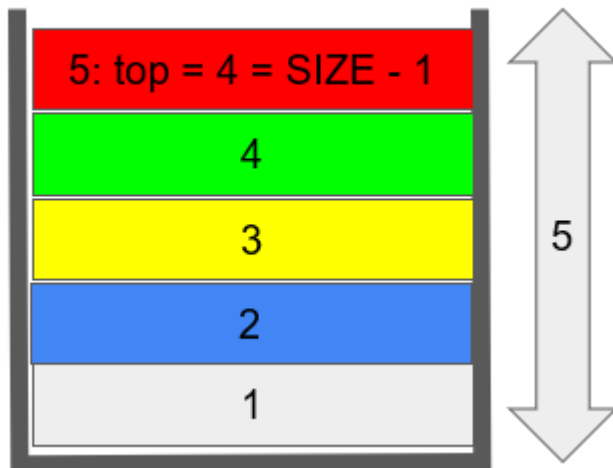


Bài 9_ Stack – Queue

Cấu trúc dữ liệu là cách **tổ chức** và **lưu trữ** dữ liệu trong bộ nhớ để có thể thao tác hiệu quả.

- cấu trúc dữ liệu cơ bản: mảng
- cấu trúc dữ liệu nâng cao: ngăn xếp (Stack), hàng đợi (Queue), danh sách liên kết (Linked List), Cây
- Cấu trúc dữ liệu phi tuyến tính (none linear data structure): Đồ thị graph, cây tree.

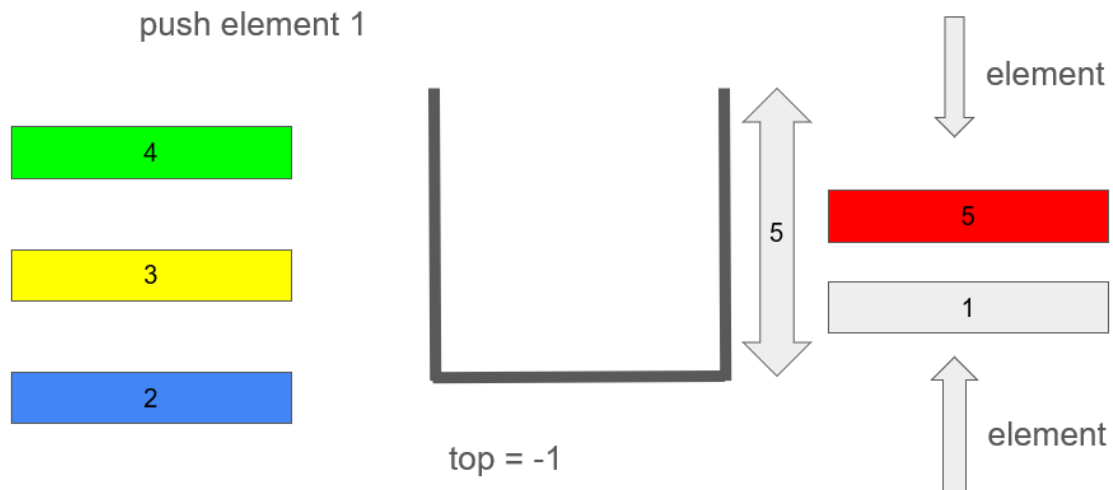


9.2.Stack

Stack (ngăn xếp) là một cấu trúc dữ liệu tuân theo nguyên tắc "**Last In, First Out**" (LIFO), nghĩa là phần tử cuối cùng được thêm vào stack sẽ là phần tử đầu tiên được lấy ra.

Các thao tác cơ bản trên stack bao gồm:

- "**push**" để thêm một phần tử vào **đỉnh** của stack push \rightarrow top++
- "**pop**" để xóa một phần tử ở **đỉnh** stack. pop \rightarrow top--
- "**peek/top**" để lấy giá trị của phần tử ở **đỉnh stack**. top (max) = size - 1 \rightarrow FULL
- Kiểm tra Stack đầy: top = size - 1 top = -1 \rightarrow EMPTY
- Kiểm tra Stack rỗng: top = -1



```
//Stack quản lý dữ liệu theo nguyên lý LIFO : Last in first out.
#include <stdio.h>
#include <stdlib.h>

typedef struct Stack {
    int* items; // con trỏ items sẽ dùng trỏ đếm vùng nhớ cấp phát động heap,
    bản chất con trỏ là 1 mảng lưu data
    int size;    // kích thước stack
    int top;     // đỉnh stack, top = -1 : empty stack, top = size-1 : full
    stack
} Stack; // khai báo stack gồm kích thước size, vị trí hiện tại dữ liệu top,
con trỏ items trỏ đến vùng nhớ và lưu dữ liệu.

void initialize( Stack *stack, int size) { // hàm khởi tạo các giá trị ban
đầu cho stack, tham số truyền vào kiểu con trỏ stack, trả lại kiểu void.
    stack->items = (int*) malloc(sizeof(int) * size); // cấp phát vùng nhớ
động kích thước = size.
    stack->size = size;
    stack->top = -1; // top vừa là đỉnh stack, cũng là vị trí hiện tại của
dữ liệu.
}

int is_empty( Stack stack) { // kiểm tra stack rỗng
    return stack.top == -1;
}

int is_full( Stack stack) { // kiểm tra stack đầy
    return stack.top == stack.size - 1;
}

// Thêm phần tử vào stack (Push)
void push( Stack *stack, int value) { // hàm đẩy giá trị vào đỉnh stack
    if (!is_full(*stack)) { // Dùng *stack cho is_full() vì, push
    khai báo stack là tham số con trỏ, : phép giải tham chiếu trả lại giá trị
    struct, hàm is_full định nghĩa đối tượng truyền vào là struct
```

```

        stack->items[++stack->top] = value; // vị trí hiện tại là top,
// ++stack->top : tăng chỉ số top lên 1 trước. stack->items[]: gán mảng con trỏ
// items ở vị trí top = giá trị phần tử.
    } else {
        printf("Stack đầy\n");
    }
}
// Lấy phần tử ra khỏi stack (Pop)
int pop( Stack *stack) {
    if (!is_empty(*stack)) {
        return stack->items[stack->top--]; //lấy phần tử vị trí top ra và sau
// đó giảm số đỉnh top đi 1.
    }
    else {
        printf("Stack rỗng\n");
        return -1;
    }
}
// Xem phần tử ở đỉnh stack (Peek)
int peek( Stack stack) {
    if (!is_empty(stack)) {
        return stack.items[stack.top]; // trả về giá trị data ở đỉnh top
    } else {
        printf("Stack is empty\n");
        return -1;
    }
}
// Hiển thị toàn bộ stack
void display(struct Stack* stack) {
    if (is_empty(*stack)) {
        printf("Stack rỗng!\n");
        return;
    }
    printf("Nội dung stack (từ đỉnh xuống đáy): ");
    for (int i = stack->top; i >= 0; i--) {
        printf("%d ", stack->items[i]);
    }
    printf("\n");
}
int main() {
    Stack stack1;
    initialize(&stack1, 5);
    push(&stack1, 10);
    push(&stack1, 20);
    push(&stack1, 30);
    push(&stack1, 40);
}

```

```

push(&stack1, 50); // đẩy các phần tử 10,20,30,40,50 vào stack.

display(&stack1); // hiển thị cả stack
printf("Top element: %d\n", peek(stack1)); // hiển thị giá trị đỉnh stack

pop(&stack1); // lấy phần tử đỉnh stack ra
push(&stack1, 60); // đẩy giá trị thay thế vào đỉnh stack.

display(&stack1);
printf("Top element: %d\n", peek(stack1));

return 0;
}

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter

```

Top element: 60

[Done] exited with code=0 in 0.277 seconds

[Running] cd "e:\3.ELECTRONIC\HALA-EMBBED PROGRAMMING_GOOD\
"e:\3.ELECTRONIC\HALA-EMBBED PROGRAMMING_GOOD\C & C++ ADVAN
Nội dung stack (từ đỉnh xuống đáy): 50 40 30 20 10
Top element: 50
Nội dung stack (từ đỉnh xuống đáy): 60 40 30 20 10
Top element: 60

[Done] exited with code=0 in 0.274 seconds

```

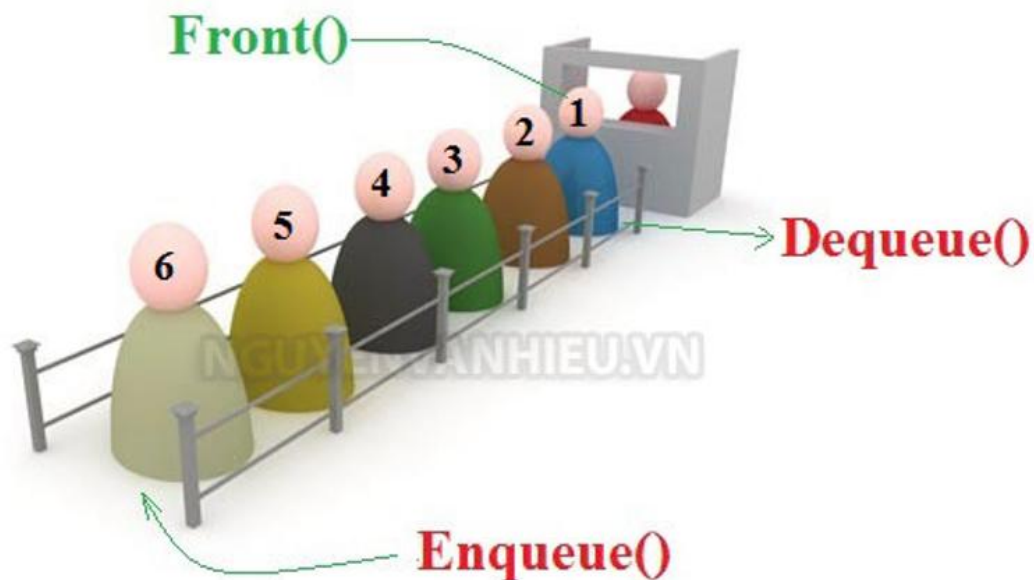
9.3 Queue

Queue là một cấu trúc dữ liệu tuân theo nguyên tắc "**First In, First Out**" (FIFO), nghĩa là phần tử đầu tiên được thêm vào hàng đợi sẽ là phần tử đầu tiên được lấy ra.

Các thao tác cơ bản trên hàng đợi bao gồm:

- "enqueue" (thêm phần tử vào **cuối** hàng đợi)
- "dequeue" (lấy phần tử từ **đầu** hàng đợi).
- "front" để lấy giá trị của phần tử đứng đầu hàng đợi.
- "rear" để lấy giá trị của phần tử đứng cuối hàng đợi.
- Kiểm tra hàng đợi đầy/rỗng.

Linear Queue
Circular Queue
Priority Queue



Phân biệt các kiểu hàng đợi

Linear Queue (Hàng đợi tuyến tính)

- **Đặc điểm:**
 - Là loại hàng đợi đơn giản nhất, hoạt động theo nguyên tắc **FIFO (First In First Out)**: phần tử được thêm vào từ cuối (rear) và lấy ra từ đầu (front).
 - Khi lấy ra một phần tử, các phần tử còn lại không được dịch chuyển để lấp đầy khoảng trống, dẫn đến lãng phí bộ nhớ nếu không quản lý tốt.
- **Ứng dụng:**
 - Dùng trong các tình huống đơn giản như xử lý tác vụ tuần tự.


Circular Queue (Hàng đợi vòng)

- **Đặc điểm:**
 - Là cải tiến của Linear Queue, tận dụng không gian trống bằng cách nối đầu và cuối hàng đợi thành một vòng tròn.

- *Khi chỉ số rear đạt đến cuối mảng, nó sẽ quay lại đầu nếu có không gian trống*, tối ưu hóa việc sử dụng bộ nhớ.
- **Ứng dụng:**
 - Dùng trong các hệ thống tuần hoàn như quản lý bộ đệm (buffer).

Priority Queue (Hàng đợi ưu tiên)

- **Đặc điểm:**
 - Không tuân theo nguyên tắc FIFO, phần tử có mức ưu tiên cao nhất được xử lý trước.
 - Mỗi phần tử được gán một giá trị ưu tiên (thường là số nguyên, số càng nhỏ thì ưu tiên càng cao).
 - Có thể được triển khai bằng mảng, danh sách liên kết hoặc heap.
- **Ứng dụng:**
 - Dùng trong lập lịch CPU, quản lý giao thông, hoặc các thuật toán như Dijkstra

- 
- so sánh khác nhau các kiểu hàng đợi, và viết chương trình c cho các kiểu hàng đợi Linear Queue Circular Queue Priority Queue

- [Thought for 23s](#)
- [Expand for details](#)

• So sánh các kiểu hàng đợi và triển khai bằng ngôn ngữ C

- Hàng đợi (queue) là một cấu trúc dữ liệu quan trọng trong lập trình, được sử dụng để lưu trữ và quản lý dữ liệu theo một trật tự nhất định. Trong bài này, chúng ta sẽ so sánh ba kiểu hàng đợi phổ biến: **Linear Queue (Hàng đợi tuyến tính)**, **Circular Queue (Hàng đợi vòng)**, và **Priority Queue (Hàng đợi ưu tiên)**. Sau đó, chúng ta sẽ cung cấp các chương trình C để minh họa cách triển khai từng loại.

- _____

- 1. So sánh các kiểu hàng đợi
- Dưới đây là bảng so sánh chi tiết giữa ba loại hàng đợi theo các tiêu chí khác nhau:

Tiêu chí	Linear Queue (Hàng đợi tuyến tính)	Circular Queue (Hàng đợi vòng)	Priority Queue (Hàng đợi ưu tiên)
Nguyên tắc hoạt động	FIFO (First In, First Out): phần tử thêm vào cuối, lấy ra từ đầu	FIFO: phần tử thêm vào cuối, lấy ra từ đầu, nhưng di chuyển vòng tròn	Không tuân theo FIFO, phần tử có ưu tiên cao được xử lý trước
Triển khai	Có thể dùng mảng hoặc danh sách liên kết	Chủ yếu dùng mảng, với cơ chế vòng tròn	Có thể dùng mảng, danh sách liên kết hoặc heap
Hiệu quả không gian	Thấp: có thể lãng phí không gian khi mảng đầy, dù có phần tử đã bị xóa	Cao: tận dụng không gian tốt hơn nhờ cơ chế vòng tròn	Trung bình: phụ thuộc vào cách triển khai
Độ phức tạp triển khai	Thấp: đơn giản, dễ hiểu	Trung bình: phức tạp hơn do cơ chế vòng tròn	Cao: cần quản lý ưu tiên, có thể tốn tài nguyên
Ứng dụng	Các tình huống đơn giản, không cần quản lý không gian chặt chẽ	Các tình huống cần quản lý không gian hiệu quả, như buffer	Các tình huống cần xử lý theo mức độ ưu tiên, như lập lịch tác vụ

9.3.1 Linear Queue (Hàng đợi tuyến tính)

- Sử dụng mảng để lưu trữ các phần tử.
- Theo dõi hai chỉ số: front (đầu hàng đợi) và rear (cuối hàng đợi).

Đặc điểm hàng đợi tuyến tính

- Khi enqueue (thêm phần tử vào cuối queue), chỉ số rear tăng. Nếu tăng đến max size của hàng đợi không thêm được nữa, hàng đợi đầy.
- khi dequeue (xóa phần tử từ đầu queue), front tăng (chờ xóa phần tử tiếp theo)
- $front = rear$: hàng đợi còn 1 phần tử
- queue đầy: $rear = size - 1$
- queue rỗng: $rear = front = -1$.
- Queue bắt đầu có phần tử : tăng $rear = front = 0$.
- Khi queue đầy thì phải xóa hết phần tử trong queue mới thêm được phần tử mới vào=> rất lãng phí tài nguyên, bộ nhớ.

```

// Chương trình viết các hàm thao tác với hàng đợi (kiểm tra hàng đợi đầy,
hàng đợi rỗng, thêm vào cuối hàng đợi, xóa phần tử ở đầu hàng đợi, hiển thị
các giá trị trong hàng đợi)
// Note: thêm vào queue : rear tăng
// xóa phần tử đầu queue: front tăng(chờ xóa phần tử tiếp)
// hàng đợi có 1 phần tử: front = rear
// queue đầy: rear = size-1
// queue rỗng: rear = front = -1.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct
{
    int *item; // con trỏ item có chức năng đa dụng trỏ đến địa chỉ vùng nhớ
động, lưu mảng giá trị các phần tử. con trỏ 1 cấp vai trò mảng 1 chiều
    int size; // số lượng phần tử tối đa có thể đưa vào
    int front; // chỉ số của phần tử đầu hàng đợi
    int rear; // chỉ số của phần tử cuối hàng đợi
} Queue; // Như vậy struct hàng đợi sẽ 4 thành viên chính (1.thành viên lưu
mảng giá trị,2.kích thước, chỉ số hàng đợi trước, sau)

// khởi tạo hàng đợi
void initialize(Queue *queue, int size) // tham số truyền vào là con trỏ queue
và kích thước hàng đợi
{
    queue->size = size;
    queue->item = (int*)malloc(size * sizeof(int)); // cấp phát động vùng nhớ
cho struct queue, malloc trả về void, nên phải ép kiểu int* cùng kiểu dữ liệu
item
    queue->front = queue->rear = -1; // không có phần tử trong hàng đợi front
= rear = -1,
}

// kiểm tra hàng đợi đầy
bool isFull(Queue queue) //tham số truyền vào là 1 struct queue.
{
    return (queue.rear == queue.size - 1); // queue đầy : rear = size-1 :
(kích thước hàng đợi)-1
}

// kiểm tra hàng đợi rỗng
bool isEmpty(Queue queue)
{
    return (queue.front == -1 || queue.front > queue.rear);
}

```



```

// thêm phần tử vào cuối hàng đợi
void enqueue(Queue *queue, int data) // note : tham số truyền vào 1. con trỏ
queue, 2. data
{
    if (isFull(*queue)) // *queue : giả tham chiếu về kiểu struct, cho khớp
với định nghĩa của hàm.
    {
        printf("Hàng đợi đầy!\n");
        return;
    }
    else
    {
        if (queue->front == -1) queue->front = queue->rear = 0; // nếu chưa có
phần tử nào front = rear = 0
        else queue->rear++;
        queue->item[queue->rear] = data; // thêm data vào mảng item ở vị trí
rear.
        printf("Enqueue data %d\n", data);
    }
}

#define QUEUE_EMPTY -1

// xóa phần tử đầu hàng đợi
// = xóa giá trị tại vị trí front hiện tại, tăng chỉ số front lên 1 (front++),
chờ xóa tiếp phần tử.
int dequeue(Queue *queue)
{
    if (isEmpty(*queue)) // kiểm tra hàng đợi rỗng thì không có gì để xóa cả
    {
        printf("Hàng đợi rỗng, không có gì để xóa :))!\n");
        return QUEUE_EMPTY;
    }
    else
    {
        int dequeue_value = queue->item[queue->front]; // gán giá trị phần tử
vào biến trung gian trước khi xóa

        queue->item[queue->front] = 0; // xóa giá trị ở phần tử front, front
tăng lên 1.

        if (queue->front == queue->rear && queue->rear == queue->size - 1) //
kiểm tra nếu xóa đến phần tử cuối cùng thì reset chỉ số front, rear về 1.
        {
            queue->front = queue->rear = -1;
        }
        else
        {

```

```

        queue->front++;
    }
    return dequeue_value;
}
}

void display(Queue queue) // in ra các phần tử trong hàng đợi line
{
    if (isEmpty(queue))
    {
        printf("Hàng đợi rỗng!\n");
    }
    else
    {
        printf("Queue: ");

        for (int i=queue.front; i<=queue.rear; i++)
        {
            printf("%d ", queue.item[i]);
        }
        printf("\n");
    }
}

int front(Queue queue) // đọc giá trị đầu hàng đợi
{
    if (isEmpty(queue))
    {
        printf("Hàng đợi rỗng!\n");
        return QUEUE_EMPTY;
    }
    else
    {
        return queue.item[queue.front];
    }
}

int rear(Queue queue) // đọc giá trị cuối hàng đợi queue
{
    if (isEmpty(queue))
    {
        printf("Hàng đợi rỗng!\n");
        return QUEUE_EMPTY;
    }
    else
    {
        return queue.item[queue.rear];
    }
}

```

```
}

int main(int argc, char const *argv[])
{
    Queue liQueue;

    initialize(&liQueue, 3);

    enqueue(&liQueue, 3);
    enqueue(&liQueue, 6);
    enqueue(&liQueue, 9);
    enqueue(&liQueue, 4);
    enqueue(&liQueue, 5);
    enqueue(&liQueue, 7);

    printf("Front: %d\n", front(liQueue)); // in ra phần tử đầu
    printf("Rear: %d\n", rear(liQueue)); // in ra phần tử cuối

    display(liQueue); // hiển thị toàn bộ phần tử trong queue

    printf("Dequeue %d\n", dequeue(&liQueue)); // lấy 1 phần đầu ra khỏi hàng
    // đợi và in ra giá trị vừa lấy ra
    // printf("Dequeue %d\n", dequeue(&liQueue));

    display(liQueue); //

    enqueue(&liQueue, 369);
    display(liQueue);

    return 0;
}
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter
[Running] cd "e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD\C & C++
linequeue && "e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD\C & C++
Enqueue data 3
Enqueue data 6
Enqueue data 9
Hàng đợi đầy!
Hàng đợi đầy!
Hàng đợi đầy!
Front: 3
Rear: 9
Queue: 3 6 9
Dequeue 3
Dequeue 6
Dequeue 9
Hàng đợi rỗng!
Enqueue data 369
Queue: 369

[Done] exited with code=0 in 0.254 seconds
```

9.3.2.Circular Queue (Hàng đợi vòng)

Đặc điểm hàng đợi tuyến tính

- Size là kích thước hàng, số phần tử tối đa của queue
- front, rear lần lượt là chỉ số đầu và cuối hàng đợi, 2 chỉ số này sẽ chạy quay vòng tròn đầu nối nhau liên tục từ : $0 - (size - 1)$ (quay vòng liên tục).
- queue rỗng: $rear = front = -1$.
- Queue bắt đầu có phần tử : $rear = front = 0$.
- $front = rear$: queue còn 1 phần tử hoặc queue đầy (full phần tử)
- Khi enqueue (thêm phần tử vào cuối queue), chỉ số rear tăng từ $0 - (size - 1)$, sau đó quay về vị trí $0, 1, \dots$ (nếu vị trí này còn trống). vị trí 0 trống khi $front > 0$,
- khi dequeue (xóa phần tử từ đầu queue), front tăng (chỉ đến phần tử tiếp theo chờ xóa)
- queue đầy: $rear \% (size - 1) = front$
- Trong hàng đợi vòng sử dụng phép chia module % (chia lấy phần dư): để xử lý tính chất vòng của hàng đợi. đây là thuật toán rất hay!

Ví dụ kiểm tra hàng đợi đầy :

```
int queue_IsFull(Queue queue)
{
```

```
return (queue.rear + 1) % queue.size == queue.front;
```

Trường hợp 1: Hàng đợi đầy

Giả sử `queue.size = 5`, `queue.front = 0`, `queue.rear = 4`.

$(4 + 1) \% 5 = 0$.

`0 == queue.front` (vì `front = 0`), nên hàng đợi đầy.

Hàm trả về 1.

Trường hợp 2: Hàng đợi đầy (vị trí vòng)

Giả sử `queue.size = 5`, `queue.front = 2`, `queue.rear = 1`.

$(1 + 1) \% 5 = 2$.

`2 == queue.front` (vì `front = 2`), nên hàng đợi đầy.

Hàm trả về 1.

Trường hợp 3: Hàng đợi chưa đầy

Giả sử `queue.size = 5`, `queue.front = 1`, `queue.rear = 3`.

$(3 + 1) \% 5 = 4$.

`4 != queue.front` (vì `front = 1`), nên hàng đợi chưa đầy.

Hàm trả về 0.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int *items; // mảng lưu trữ giá trị các phần tử
    int size;   // kích thước của hàng đợi
    int front;  // chỉ số phần tử đầu hàng đợi
    int rear;   // chỉ số phần tử cuối hàng đợi
} Queue;

// khởi tạo hàng đợi
void queue_Init(Queue *queue, int size)
{
    queue->items = (int*)malloc(size * sizeof(int)); // cấp phát vùng nhớ động
    // cho mảng items
    queue->size = size;
    queue->front = queue->rear = -1;
}

// kiểm tra hàng đợi rỗng
int queue_IsEmpty(Queue queue)
{
    return (queue.front == -1);
}

// kiểm tra hàng đợi đầy
```

```

int queue_IsFull(Queue queue)
{
    return (queue.rear + 1) % queue.size == queue.front;
}

// thêm phần tử vào cuối hàng đợi
void enqueue(Queue *queue, int data)
{
    if (queue_IsFull(*queue))
    {
        // nếu queue đầy thì không cho thêm phần tử vào
        printf("Hàng đợi đầy!\n");
    }
    else
    {
        if (queue->front == -1)
        {
            queue->front = queue->rear = 0;
        }
        else
        {
            queue->rear = (queue->rear + 1) % queue->size;
            /*
            Phép toán module % : chia lấy dư => giúp rear quay vòng
            giả sử size = 2
            Thêm phần tử đầu tiên:
            rear = (-1 + 1) % 2 = 0 % 2 = 0
            Phần tử được thêm vào vị trí 0.
            Thêm phần tử thứ 2:
            rear = (0 + 1) % 2 = 1 % 2 = 1
            Phần tử được thêm vào vị trí 1.
            Thêm phần tử thứ ba:
            rear = (1 + 1) % 2 = 2 % 2 = 0
            phần tử thứ 3 sẽ quay lại vị trí 0: đầu mảng
            */
        }
        queue->items[queue->rear] = data;
        printf("Enqueued %d\n", data);
    }
}

#define QUEUE_EMPTY -1

// xóa phần tử từ đầu hàng đợi
int dequeue(Queue *queue)
{
    if (queue_IsEmpty(*queue))
    {

```

```

        // nếu queue rỗng thì không cho xóa
        printf("Hàng đợi rỗng\n");
        return QUEUE_EMPTY;
    }
    else
    {
        int dequeue_value = queue->items[queue->front]; // gán giá trị phần tử
        // vào biến trung gian trước khi xóa
        if (queue->front == queue->rear && (queue->rear == queue->size - 1))
        //kiểm tra nếu còn 1 phần tử cuối cùng
        {
            queue->front = queue->rear = -1; // reset về trạng thái queue rỗng
        }
        else
        {
            queue->front = (queue->front + 1) % queue->size; // Dùng phép toán
            // module giúp front quay vòng. nếu xóa hết phần tử front cũng quay về 0 : vị trí
            // đầu mảng
        }
        return dequeue_value;
    }
}

// lấy giá trị của phần tử đứng đầu hàng đợi (front)
int front(Queue queue)
{
    if (queue_IsEmpty(queue))
    {
        printf("Queue is empty\n");
        return QUEUE_EMPTY;
    }
    else
    {
        return queue.items[queue.front];
    }
}

// lấy giá trị của phần tử đứng cuối hàng đợi (rear)
int rear(Queue queue)
{
    if (queue_IsEmpty(queue))
    {
        printf("Queue is empty\n");
        return QUEUE_EMPTY;
    }
    else
    {
        return queue.items[queue.rear];
    }
}

```

```

    }
}

// Hiển thị các phần tử
void display(Queue q)
{
    if (queue_IsEmpty(q))
    {
        printf("Hàng đợi rỗng\n");
        return;
    }
    printf("Hàng đợi: ");
    int i = q.front;

    while (1) // do front, rear lúc này có giá trị quay vòng nên phải dùng
    vòng lặp while.
    {
        printf("%d ", q.items[i]);
        if (i == q.rear) break; // điều kiện để thoát khỏi vòng lặp vô tận
        i = (i + 1) % q.size; // phép toán module % giúp quay vòng giá trị i.
    }
    printf("\n");
}

int main(int argc, char const *argv[])
{
    Queue queue;

    queue_Init(&queue, 5);

    enqueue(&queue, 3);
    enqueue(&queue, 6);
    enqueue(&queue, 9);
    enqueue(&queue, 33);
    enqueue(&queue, 66);

    display(queue);

    printf("Front element: %d\n", front(queue));
    printf("Rear element: %d\n", rear(queue));

    printf("Dequeued %d\n", dequeue(&queue));
    // printf("Dequeued %d\n", dequeue(&queue));
    // printf("Dequeued %d\n", dequeue(&queue));
    // printf("Dequeued %d\n", dequeue(&queue));

    display(queue);
}

```



```

enqueue(&queue, 50);
display(queue);

printf("Front element: %d\n", front(queue));
printf("Rear element: %d\n", rear(queue));

return 0;
}

```

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Filter

[Running] cd "e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD" && "e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD"
Enqueued 3
Enqueued 6
Enqueued 9
Enqueued 33
Enqueued 66
Hàng đợi: 3 6 9 33 66
Front element: 3
Rear element: 66
Dequeued 3
Hàng đợi: 6 9 33 66
Enqueued 50
Hàng đợi: 6 9 33 66 50
Front element: 6
Rear element: 50

[Done] exited with code=0 in 0.273 seconds

```

9.3.3 Hàng Đợi Ưu Tiên (Priority Queue)

Đặc điểm hàng đợi ưu tiên:

.....(bỏ xung)

```

#include <stdio.h>
#include <stdlib.h>

```

```

#define MAX_SIZE 100

// Định nghĩa cấu trúc cho một phần tử trong priority queue
typedef struct {
    int data;        // Dữ liệu của phần tử
    int priority;    // Mức độ ưu tiên
} Element;

// Định nghĩa cấu trúc cho priority queue
typedef struct {
    Element elements[MAX_SIZE];
    int size;
} PriorityQueue;

// Khởi tạo priority queue
void init(PriorityQueue *pq) {
    pq->size = 0;
}

// Hàm hoán đổi hai phần tử
void swap(Element *a, Element *b) { // khai báo tham số là con trỏ nên tham số
    // truyền vào là địa chỉ
    Element temp = *a; // phép giải tham chiếu, biến temp gán bằng giá trị,
    // con trỏ a trỏ tới.
    *a = *b;
    *b = temp;
}

// Hàm heapify để duy trì tính chất min-heap
void heapify(PriorityQueue *pq, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < pq->size && pq->elements[left].priority < pq->elements[smallest].priority) {
        smallest = left;
    }

    if (right < pq->size && pq->elements[right].priority < pq->elements[smallest].priority) {
        smallest = right;
    }

    if (smallest != i) {
        swap(&pq->elements[i], &pq->elements[smallest]);
        heapify(pq, smallest);
    }
}

```

```

}

// Thêm phần tử vào priority queue
void enqueue(PriorityQueue *pq, int data, int priority) {
    if (pq->size == MAX_SIZE) {
        printf("Priority Queue is full!\n");
        return;
    }

    Element newElement = {data, priority};
    int i = pq->size;
    pq->elements[i] = newElement;
    pq->size++;

    // Heapify từ dưới lên
    while (i != 0 && pq->elements[(i - 1) / 2].priority > pq->elements[i].priority) {
        swap(&pq->elements[i], &pq->elements[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

// Xóa và trả về phần tử có mức độ ưu tiên cao nhất
Element dequeue(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is empty!\n");
        Element empty = {-1, -1};
        return empty;
    }

    if (pq->size == 1) {
        pq->size--;
        return pq->elements[0];
    }

    Element root = pq->elements[0];
    pq->elements[0] = pq->elements[pq->size - 1];
    pq->size--;
    heapify(pq, 0);

    return root;
}

// Kiểm tra priority queue có rỗng không
int isEmpty(PriorityQueue *pq) {
    return pq->size == 0;
}

```

```

// In priority queue
void printQueue(PriorityQueue *pq) {
    if (isEmpty(pq)) {
        printf("Priority Queue is empty!\n");
        return;
    }

    printf("Priority Queue: ");
    for (int i = 0; i < pq->size; i++) {
        printf("(%d, %d) ", pq->elements[i].data, pq->elements[i].priority);
    }
    printf("\n");
}

// Hàm main để kiểm tra
int main() {
    PriorityQueue pq;
    init(&pq);

    // Thêm các phần tử vào priority queue
    enqueue(&pq, 10, 2);
    enqueue(&pq, 20, 1);
    enqueue(&pq, 30, 3);
    enqueue(&pq, 40, 0);

    // In priority queue
    printQueue(&pq); // Kết quả: (40, 0) (20, 1) (30, 3) (10, 2)

    // Xóa phần tử có ưu tiên cao nhất
    Element e = dequeue(&pq);
    printf("Dequeued: (%d, %d)\n", e.data, e.priority); // Kết quả: (40, 0)

    // In lại priority queue sau khi xóa
    printQueue(&pq); // Kết quả: (20, 1) (10, 2) (30, 3)

    return 0;
}

```