VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu Và Giải Thuật (DSA)

DSA1 - HK241

Lý Thuyết

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 6/2024

Mục lục

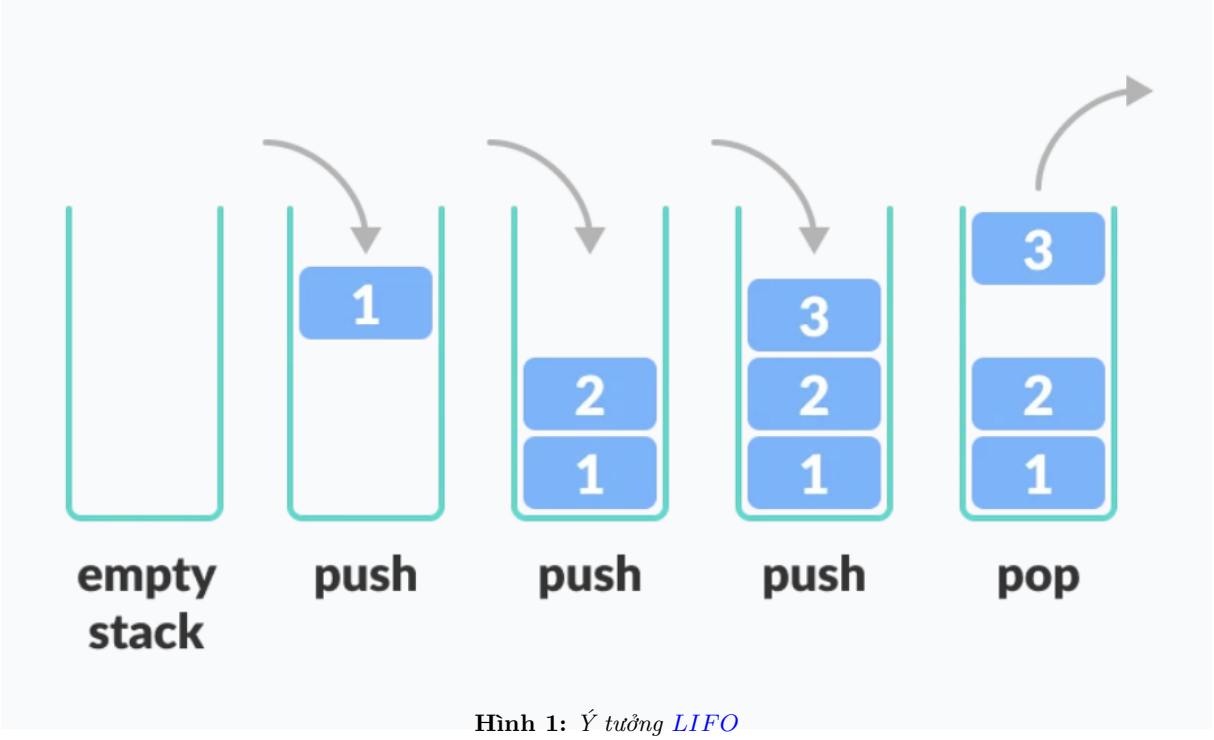
1 Stack

MT22: Ngăn xếp (Stack)

Stack là một cấu trúc dữ liệu tuân theo nguyên tắc Last In First Out (*LIFO*). Điều này có nghĩa là phần tử cuối cùng được thêm vào *stack* sẽ là phần tử đầu tiên được xóa. *Stack* thường được sử dụng để lưu trữ dữ liệu tạm thời, chẳng hạn như thông tin gọi hàm hoặc các cuộc gọi hàm đệ quy. Đảo ngược từ, thông dịch, Giải một số bài toán của lý thuyết đồ thị như giải thuật DFS ...

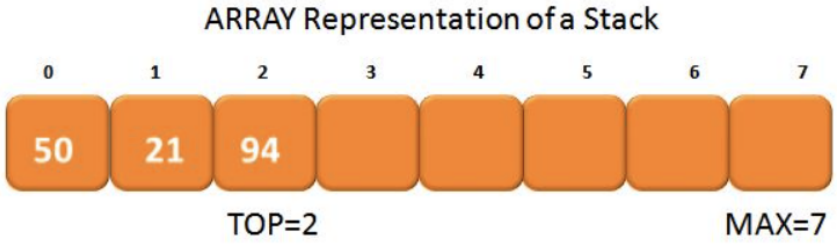
Chúng thường được triển khai bằng mảng (*Array*) hoặc danh sách liên kết (*LinkList*). Dưới đây là một số thao tác có thể được thực hiện trên stack: độ phức tạp tất cả đều là $O(1)$

- *Push*: Thêm một phần tử vào đỉnh của stack.
- *Pop*: Xóa phần tử khỏi đỉnh của stack.
- *Peek*: Trả về phần tử ở đỉnh của stack mà không xóa nó.
- *IsEmpty*: Kiểm tra xem stack có trống không.
- *Size*: Trả về số lượng phần tử trong stack.



Hình 1: *Ý tưởng LIFO*

1.1 Hiện thực bằng Array



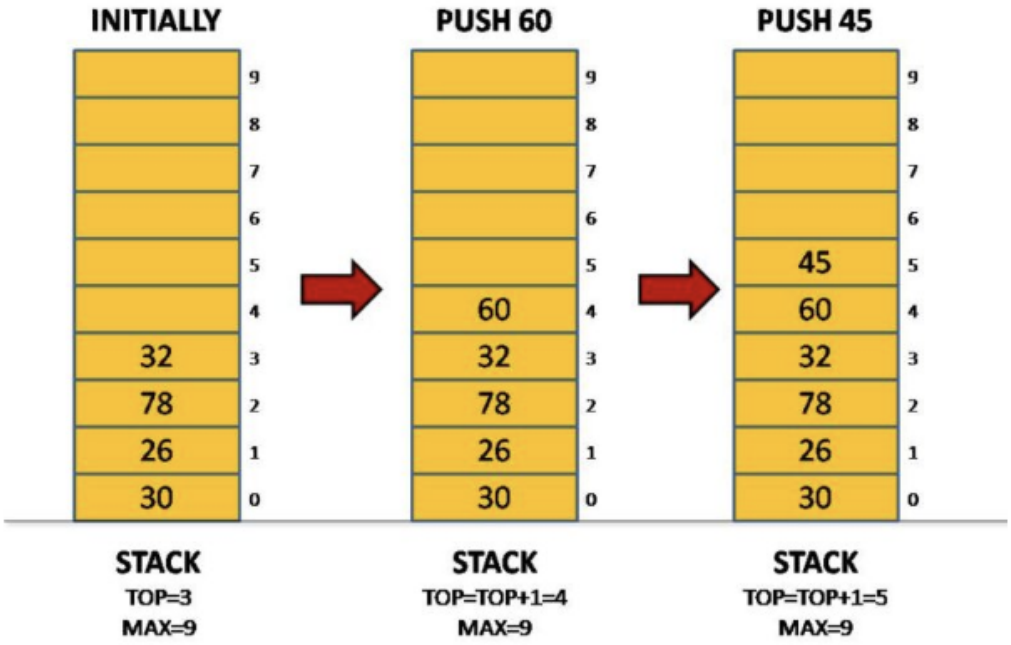
Hình 2: *Hình ảnh Stack hiện thực bằng array*

```
template <typename T>
class Stack_Array
{
private:
    T* array;  //! ngăn xếp
    int Size;  //! kích thước hiện tại
    const int MAXSIZE; //! kích thước tối đa
public:
    Stack_Array(/* args */);
    ~Stack_Array();

    bool empty();
    int size();
    T top();
    void push(int );
    void pop();
};
```

Giải Thích: Stack hiện thực bằng array

- *array* là ngăn xếp chứa đựng dữ liệu cần đưa vào ngăn xếp, *Size* là kích thước hiện tại của ngăn xếp, $MAXSIZE$ là kích thước tối đa được cấp phát cho ngăn xếp nếu vượt quá thì có thể tăng giá trị này nên và khởi tạo lại *array* mới *copy* một hết *array* cũ vào
- *contructor* và *destructor* giống với danh sách liên kết phần trước
- *empty* và *size* xác định ngăn xếp có rỗng hay không hay kích thước hiện tại của ngăn xếp
- *top* lấy từ đầu ngăn xếp tức là vị trí cuối của mảng $array[size-1]$ nếu tính phần tử đầu $array[0]$
- *push* là thêm phần tử mới vào đầu ngăn xếp tức là vị trí cuối của mảng được thêm vào $array[size]$ và tăng $size++$
- *pop* là xóa phần tử mới được thêm vào ra khỏi ngăn xếp là vị trí cuối của mảng $array[size-1]$ và giảm $size--$
- Trong C++ ta có thểm dùng thư viện *vector* để hiện thực
- Trong C++ có sẵn thư viện *stack* các hàm cũng giống như ta hiện thực *size*, *empty*, *top*, *push*, *pop*. **Thư viện** $std::stack$ **:** https://cplusplus.com/reference/stack/stack/

1.1.1 Push



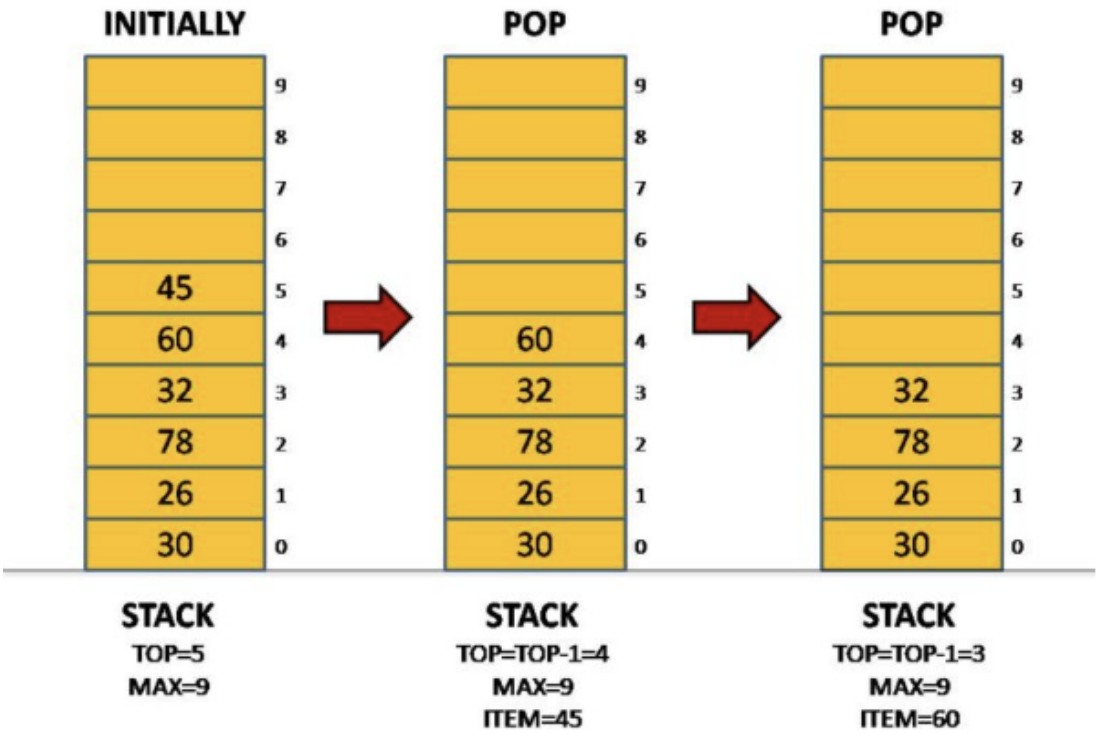
Hình 3: *hàm push in Stack hiện thực bằng array*

```cpp
//! thêm một phần tử vào đầu stack
//^ O(1)
template <typename T>
void Stack_Array<T>::push(int data){
    if(this->MAXSIZE == Size) throw("FULL");  //~ đầy rồi
    array[Size++] = data;
}
```

Giải Thích: Stack hiện thực bằng array hàm push

Nếu mà thêm vào đầu mảng lúc này độ phức tạp sẽ là $O(N)$ nên không dùng được cách này, đây là các bước thêm vào cuối mảng.
Kiểm tra ngăn xếp đã đầy hay chưa nếu đầy có thể thêm hoặc thông báo với người dùng
Thêm $data$ vào cuối mảng $array[Size]$
Tăng kích thước mảng $Size++$

1.1.2 Pop



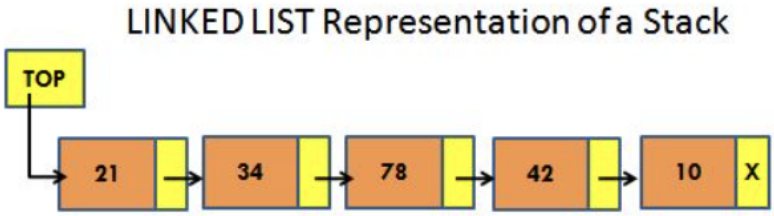
Hình 4: *hàm Pop in Stack hiện thực bằng array*

```
//! xóa phần tử đầu tiên tay đổi size
//^ O(1)
template <typename T>
void Stack_Array<T>::pop(){
    if(this->empty()) throw("Empty");    //~ không có sao xóa
    Size --;
}
```

Giải Thích: Stack hiện thực bằng array hàm pop

Vì *STACK* là cấu trúc *LIFO* nên cần xóa cuối mảng vì lúc thêm thì thêm cuối mảng nên *data* nào được thêm vào gần nhất thì bị xóa trước, đây là các bước xóa cuối mảng.
Kiểm tra ngăn xếp có bị rỗng hay không nếu rỗng thì thông báo cho người dùng
Xóa phần tử cuối mảng $array[Size-1]$ nếu trường hợp là đối tượng được khai báo trong *heap* bằng toán tử *new* thì cần gọi *delete* nếu không thì bỏ qua.
Giảm kích thước mảng $Size--$

1.2 Hiện thực bằng Linked List



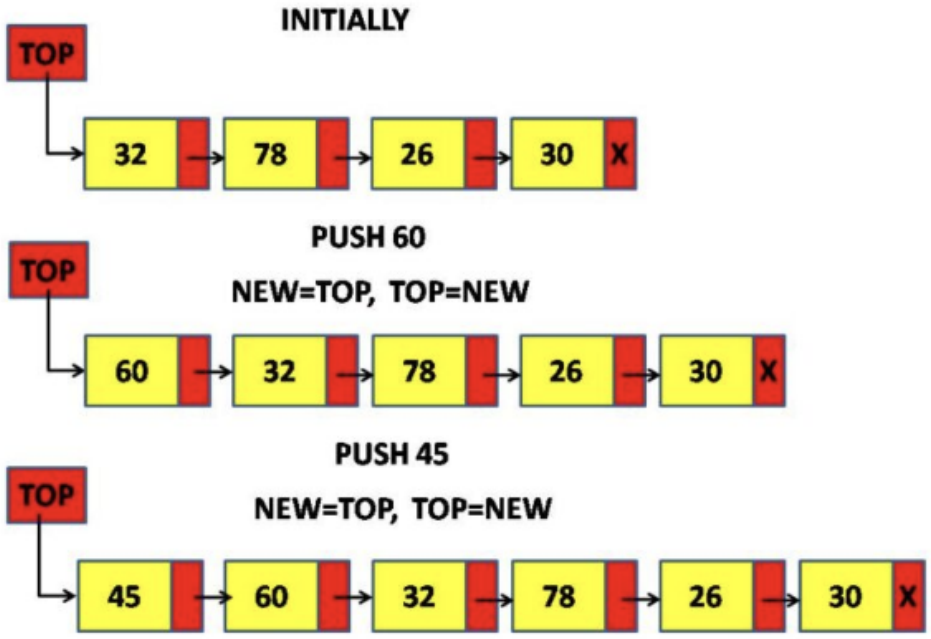
Hình 5: *Hình ảnh Stack hiện thực bằng LinkList*

```
template <typename T>
class Stack_Linked_List
{
private:
    class Node;
private:
    Node* head;
    int Size;
public:
    Stack_Linked_List(/* args */);
    ~Stack_Linked_List();

    bool empty();
    int size();
    T top();
    void push(T );
    void pop();

private:
    class Node{
    public:
        T data;
        Node* next;
        Node(T data, Node* next):data(data), next(next){}
    };
};

```

Giải Thích: Stack hiện thực bằng linklist hàm push

- giống bên danh sách liên kết đơn thôi với *head* là node đầu tiên, *size* là kích thước
- *push* thì thêm một phần tử vào đầu danh sách.
- *pop* thì xóa một phần tử đầu danh sách.
- Vì khi xóa ở cuối danh sách liên kết đơn độ phức tạp $O(n)$ nên ta không chọn được thêm xóa ở cuối danh sách.

1.2.1 Push



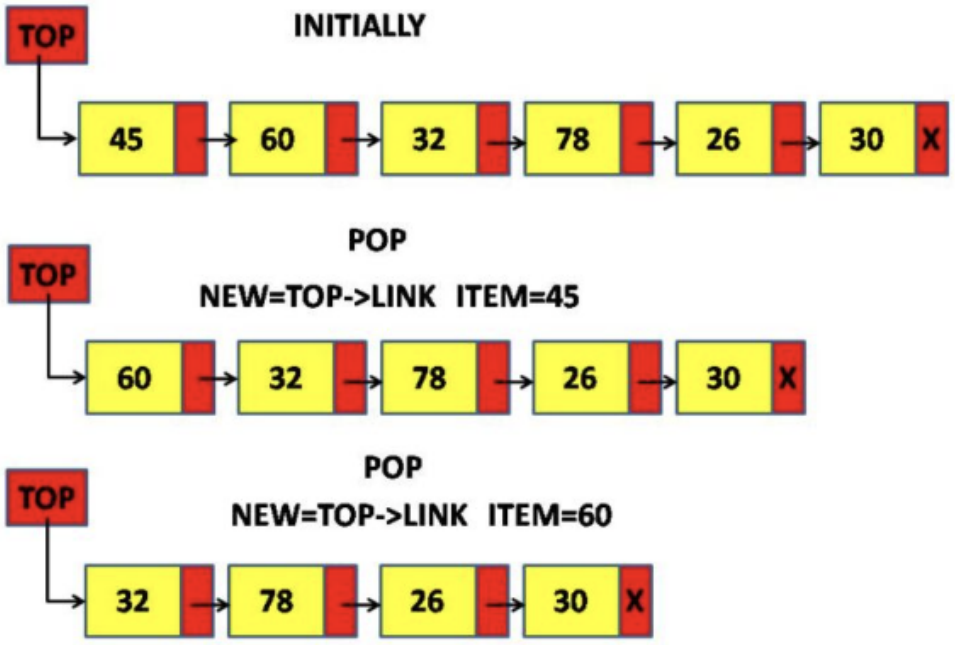
Hình 6: *hàm push in Stack hiện thực bằng LinkList*

```
//! thêm một phần tử vào đầu stack không bao giờ full
//! chèn ở đầu linked list
//^ O(1)
template <typename T>
void Stack_Linked_List<T>::push(T data){
    head = new  Stack_Linked_List<T>::Node(data, head);
    Size ++;
}
```

Giải Thích: Stack hiện thực bằng linklist hàm push

Vì danh sách liên kết không có kích thước tối đa nên bước đầu ta tạo *node* mới
Thêm *node* mới vào đầu danh sách và cập nhật *head* mới
Tăng kích thước mảng *Size* + +

1.2.2 Pop



Hình 7: *hàm pop in Stack hiện thực bằng LinkList*

```cpp
//! xóa phần tử đầu tiên tay đổi size
//! xóa ở đầu linked list
//^ O(1)
template <typename T>
void Stack_Linked_List<T>::pop(){
    if(this->empty()) throw("Empty");    //~ không có sao xóa
    Stack_Linked_List<T>::Node* temp = head;
    head = head->next;
    delete temp;
    Size --;
}
```

Giải Thích: Stack hiện thực bằng linklist hàm pop

Kiểm tra ngăn xếp có bị rỗng hay không nếu rỗng thì thông báo cho người dùng
Xóa phần tử đầu mảng dùng toán tử *delete* và cập nhật lại *head*
Giảm kích thước mảng *Size – –*

2 Queue

MT22: Hàng đợi (Queue)

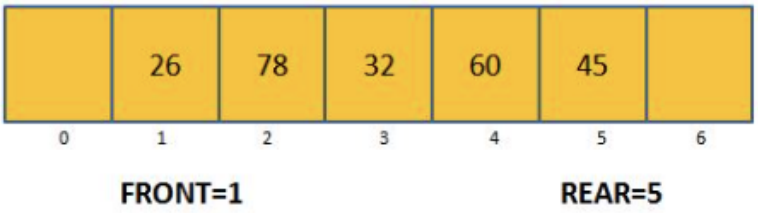
Queue là một cấu trúc dữ liệu tuân theo nguyên tắc First In First Out (*FIFO*). Điều này có nghĩa là phần tử đầu tiên được thêm vào *queue* sẽ là phần tử đầu tiên được xóa. *Queue* thường được sử dụng để lưu trữ dữ liệu cần được xử lý theo thứ tự đến trước phục vụ trước..

Chúng thường được triển khai bằng mảng (*Array*) hoặc danh sách liên kết (*LinkList*). Dưới đây là một số thao tác có thể được thực hiện trên *queue*: độ phức tạp tất cả đều là $O(1)$

- *Enqueue*: Thêm một phần tử vào đuôi của queue.
- *Dequeue*: Xóa một phần tử khỏi đầu của queue.
- *Peek*/*top*: Trả về phần tử ở đầu của queue mà không xóa nó.
- *IsEmpty*: Kiểm tra xem Queue có trống không.
- *Size*: Trả về số lượng phần tử trong Queue.

Hình 8: *Ý tưởng FIFO*

2.1 Hiện thực bằng Array



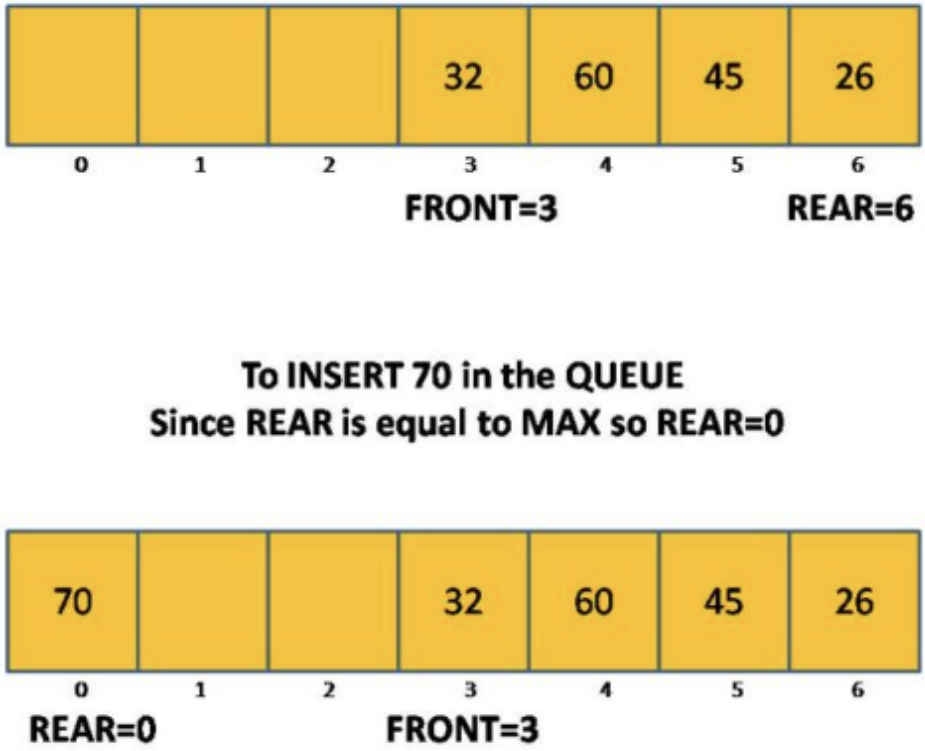
Hình 9: *Hình ảnh Queue hiện thực bằng array*

```
template <typename T>
class Queue_Array
{
private:
    T* array;
    int Size;
    int index;
    const int MAXSIZE;
public:
    Queue_Array(/* args */);
    ~Queue_Array();

    bool empty();
    int size();
    T top();
    void Enqueue(T );
    void Dequeue();
};
```

Giải Thích: Stack hiện thực bằng array

- *array* là hàng đợi chứa đựng dữ liệu cần đưa vào hàng đợi, *Size* là kích thước hiện tại của hàng đợi, $MAXSIZE$ là kích thước tối đa được cấp phát cho hàng đợi nếu vượt quá thì có thể tăng giá trị này nên và khởi tạo lại *array* mới *copy* một hết *array* cũ vào
- *index* dùng để xác định vị trí của đầu hàng đợi đang ở đâu trong mảng vì ta sẽ xử lí mảng vòng tròn nên *index* có gia trị thua đổi từ $0 -> MAXSIZE - 1 -> 0 -> ....$ Do đó ta xác định được $array[index]$ là đầu hàng đợi và $array[Size + index]$ là phần tử cuối hàng đợi.
- *contructor* và *destructor* giống với danh sách liên kết phần trước
- *empty* và *size* xác định hàng đợi có rỗng hay không hay kích thước hiện tại của hàng đợi
- *top* lấy từ cuối hàng đợi tức là vị trí đầu của mảng đang xét $array[index]$ nếu tính phần tử đầu $array[0]$
- *Enqueue* là thêm phần tử mới vào đầu hàng đợi tức là vị trí cuối của hàng đợi được thêm vào $array[Size + index]$ và tăng $size++$
- *Dequeue* là xóa phần tử lâu nhất ra khỏi hàng đợi là vị trí đầu của hàng đợi $array[index]$ và giảm $size--$ và tăng *index*
- Trong C++ ta có thểm dùng thư viện *vector* để hiện thực
- Trong C++ có sẵn thư viện *queue* các hàm cũng giống như ta hiện thực *size*, *empty*, *back*, *front*, *pop*. **Thư viện** $std :: queue$ **:** https://cplusplus.com/reference/queue/queue/

2.1.1 Enqueue



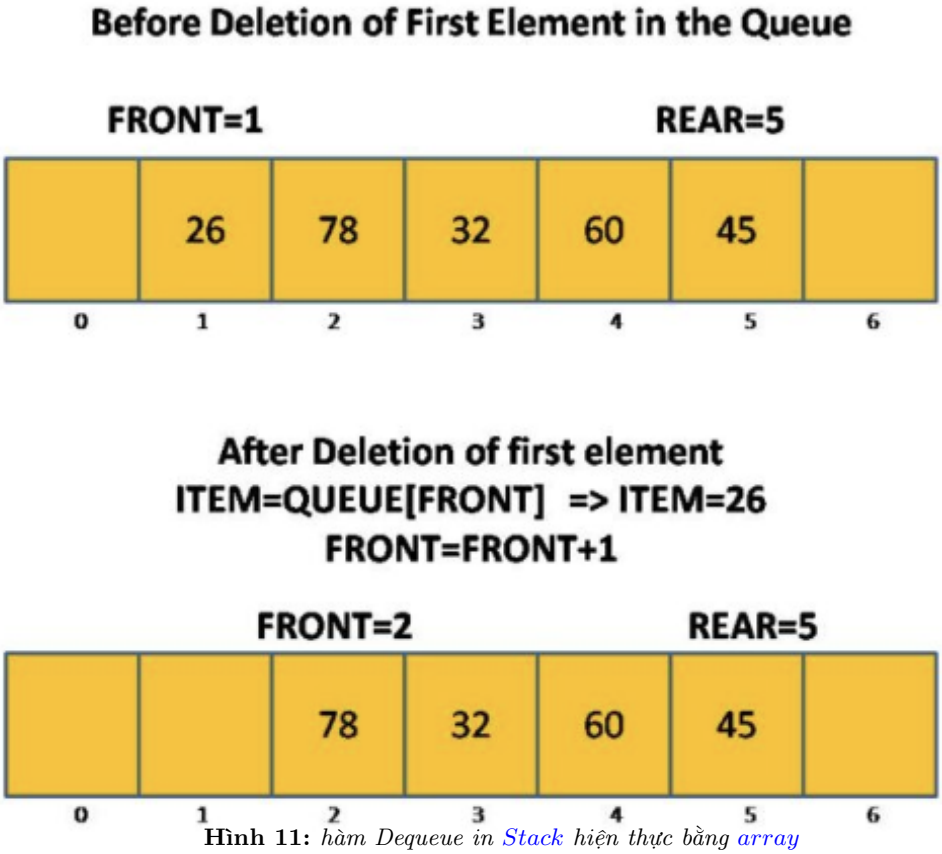
Hình 10: *hàm Enqueue in Queue hiện thực bằng array*

```cpp
//! thêm một phần tử vào cuối stack
//^ O(1)
template <typename T>
void Queue_Array<T>::Enqueue(T data){
    if(this->MAXSIZE == Size) throw("FULL");  //~ đầy rồi
    array[Size + index] = data;
    Size ++;
}
```

Giải Thích: Stack hiện thực bằng array hàm Enqueue

Vì thiết kế theo kiểu vòng nên thêm xóa ở đầu cuối gì cũng được. code trên thêm ở cuối *queue* cho giống lý thuyết
Kiểm tra hàng đợi đã đầy hay chưa nếu đầy có thể thêm hoặc thông báo với người dùng
Thêm *data* vào cuối hàng đợi $array[Size + index]$
Tăng kích thước mảng $Size++$

2.1.2 Dequeue



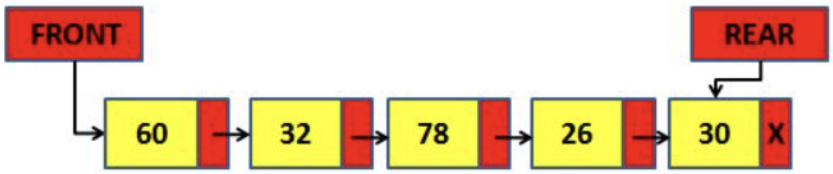
Hình 11: *hàm Dequeue in Stack hiện thực bằng array*

```cpp
//! xóa phần tử đầu tiên tay đổi size
//^ O(1)
template <typename T>
void Queue_Array<T>::Dequeue(){
    if(this->empty()) throw("Empty");    //~ không có sao xóa
    Size --;
    index = (index + 1) % MAXSIZE;
}
```

Giải Thích: Stack hiện thực bằng array hàm pop

Vì $QUEUE$ là cấu trúc $FIFO$ nên chọn thêm thì vô cuối thì sẽ xóa ở đầu.
Kiểm tra hàng đợi có bị rỗng hay không nếu rỗng thì thông báo cho người dùng
Xóa phần tử đầu hàng đợi $array[index]$ nếu trường hợp là đối tượng được khai báo trong $heap$ bằng toán tử new thì cần gọi $delete$ nếu không thì bỏ qua.
Giảm kích thước mảng $Size--$
cập nhật $index = (index+1)\%MAXSIZE$ vì khi $index$ lớn hơn $MaxSize$ có thể quay lại ban đầu 0

2.2 Hiện thực bằng Linked List



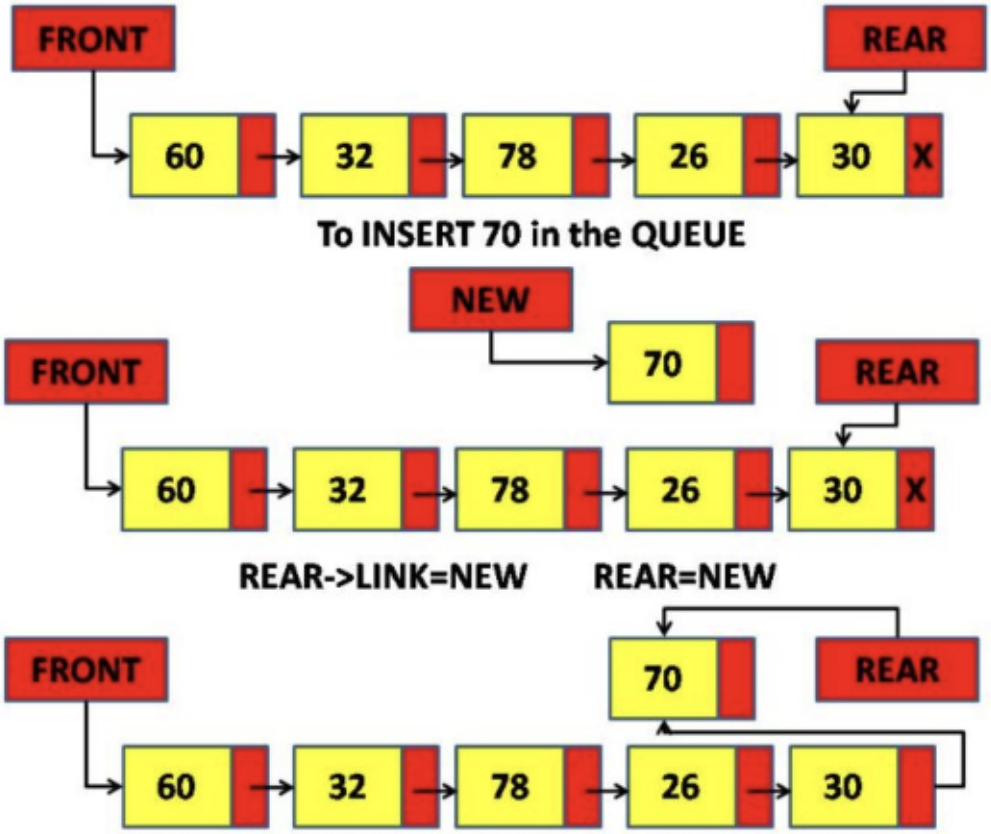
Hình 12: *Hình ảnh Queue hiện thực bằng LinkList*

```
template <typename T>
class Queue_Linked_List
{
public:
    class Node;
private:
    Node* head, *tail;
    int Size;

public:
    Queue_Linked_List(/* args */);
    ~Queue_Linked_List();

    bool empty();
    int size();
    T peek();
    void Enqueue(T );
    void Dequeue();

public:
    class Node{
    public:
        T data;
        Node* next;
        Node(T data, Node* next):data(data), next(next){}
    };
};
```

Giải Thích: Queue hiện thực bằng linklist

- giống bên danh sách liên kết đơn thôi với *head* là node đầu tiên, *tail* là node cuối cùng, *size* là kích thước
- *Enqueue* thì thêm một phần tử vào cuối danh sách.
- *Dequeue* thì xóa một phần tử đầu danh sách.
- Vì khi xóa ở cuối danh sách liên kết đơn độ phức tạp $O(n)$ nên ta không chọn được xóa ở cuối.

2.2.1 Enqueue



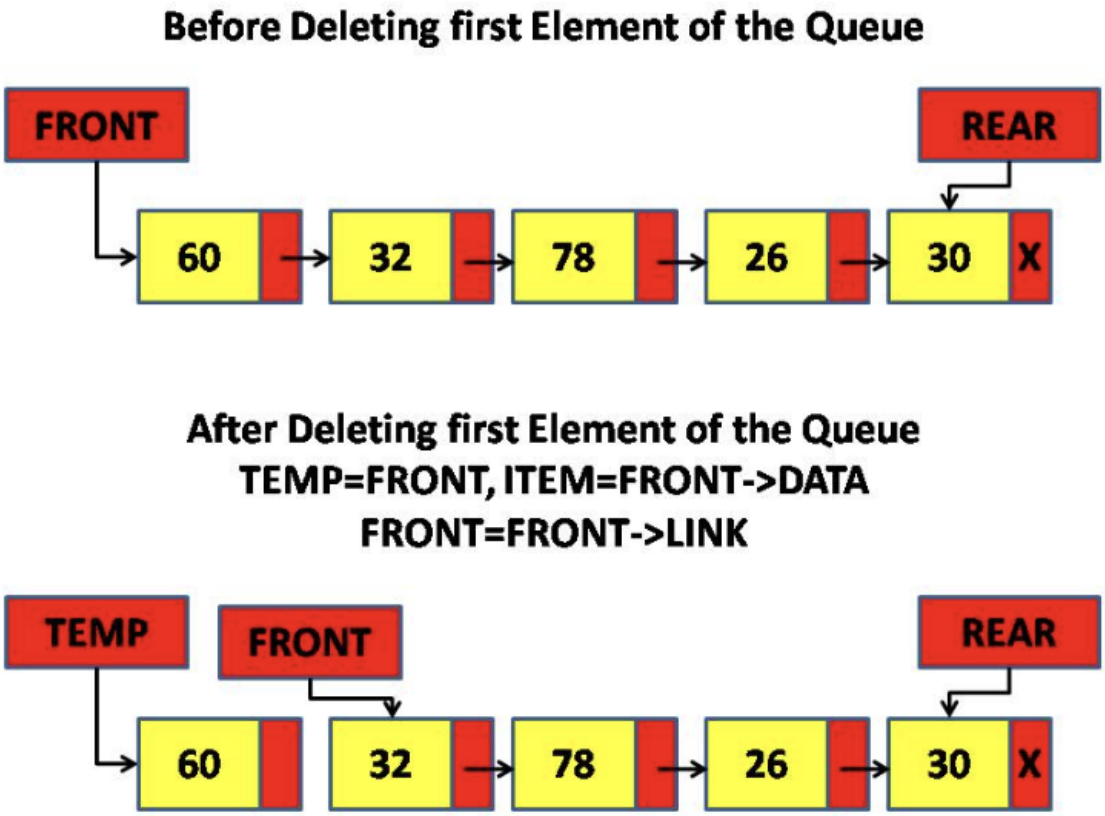
Hình 13: *hàm push in Enqueue hiện thực bằng LinkList*

```
//! thêm một phần tử vào cuối danh sách
//^ O(1)
template <typename T>
void Queue_Linked_List<T>::Enqueue(T data){
    if(this->Size == 0) head = tail = new Node(data, nullptr);
    else{
        tail->next = new Node(data, nullptr);
        tail = tail->next;
    }
    this->Size ++;
}
```

Giải Thích: Queue hiện thực bằng linklist hàm Enqueue

xết xem danh sách có rỗng hay không cập nhật *head*, *tail*
Thêm *node* mới vào cuối danh sách và cập nhật *tail* mới
Tăng kích thước mảng $Size++$

2.2.2 Dequeue



Hình 14: *hàm Dequeue in Queue hiện thực bằng LinkList*

```
//! xóa ở đầu linked list
//^ O(1)
template <typename T>
void Queue_Linked_List<T>::Dequeue(){
    if(this->empty()) throw("Empty");    //~ không có sao xóa
    Node* temp = head;
    head = head->next;
    delete temp;
    this->Size --;
}
```

Giải Thích: Stack hiện thực bằng linklist hàm Dequeue

Kiểm tra ngăn xếp có bị rỗng hay không nếu rỗng thì thông báo cho người dùng
Xóa phần tử đầu mảng dùng toán tử *delete* và cập nhật lại *head*
Giảm kích thước mảng *Size* – –