게임 자료구조와 알고리즘

-CHAPTER8-

SOULSEEK

목차

스택(STACK)

1. 스택(STACK)

- 후입선출(**LIFO – Last In, First Out**)방식의 자료구조
- 계산기 프로그램이 활용의 예가 된다(대표적인 활용의 예)
- 다른 자료구조를 도구로 사용해서 구현 할 수 있다.

스택 자료구조의 ADT

void StackInit(Stack* pstack);
- 스택의 초기화를 진행한다.
- 스택 생성 후 제일 먼저 호출되어야 하는 함수

int SIsEmpty(Stack* pstack);
- 스택이 빈 경우 **TRUE(1)**을, 그렇지 않은 경우 **FALSE(0)**을 반환한다.

void Spush(Stack* pstack, Data data);
- 스택에 데이터를 저장한다. 매개변수 **data**로 전달된 값을 저장한다.

Data Spop(Stack* pstack);
- 마지막에 저장된 요소를 삭제한다.
- 삭제된 데이터는 반환이 된다.
- 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

Data Speek(Stack* pstack);
- 마지막에 저장된 요소를 반환하되 삭제하지 않는다.
- 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

1. 스택(STACK)

스택의 배열 기반 구현

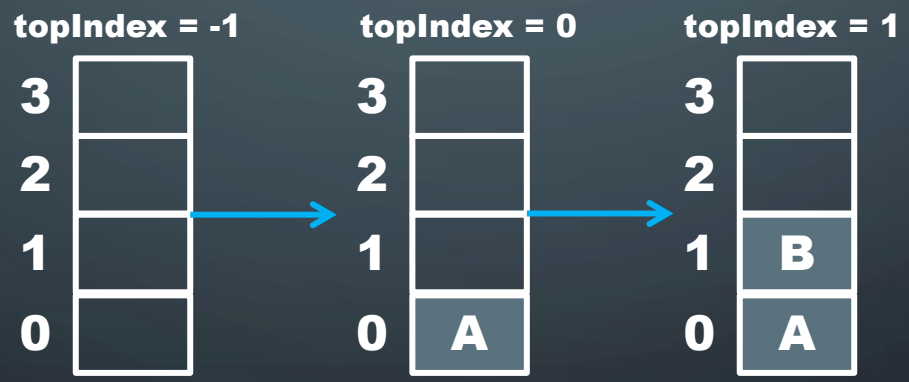
- 데이터를 추가하고 꺼내 쓰고 하는 과정만 있어서 리스트 보다 단순하고 상황이 다양하지 않다.
- 배열의 길이와 상관없이 인덱스 **0**이 항상 바닥에 위치한다.
- 마지막 데이터의 저장 위치를 기억해야 한다.

```
#define TRUE1
#define FALSE0
#define STACK_LEN100

typedef int Data;

typedef struct _arrayStack
{
	Data stackArr[STACK_LEN];
	int topIndex;
}ArrayStack;

typedef ArrayStack Stack;

void StackInit(Stack* pstack);//스택 초기화
int SIsEmpty(Stack* pstack);//스택이 비어있는지 확인

void SPush(Stack* pstack, Data data);//스택의 push 연산
Data SPop(Stack* pstack);//스택의 pop 연산
Data SPeek(Stack* pstack);//스택의 peek 연산
```

1. 스택(STACK)

초기화

```
void StackInit(Stack* pstack)
{
    pstack->topIndex = -1;              // topIndex의 -1은 빈 상태를 의미한다.
}
```



1. 스택(STACK)

삽입 및 삭제, 반환(반환은 제일 위에 있는 인덱스 DATA)

- 스택이 비어있는지 유무를 확인하고 진행해야한다.

```
Int SIsEmpty(Stack* pstack)
{
    if(pstack->topIndex == -1)          //스택이 비어있다면.
        return TRUE;
    else
        return FALSE;
}
```

삽입

```
void Spush(Stack* pstack, Data data)
{
    pstack->topIndex += 1;
    pstack->stackArr[pstack->topIndex] = data;
}
```

1. 스택(STACK)

삭제

```
Data Spop(Stack* pstack)
{
    int rIdx;

    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    rIdx = pstack->topIndex;          // 삭제할 데이터가 저장된 인덱스 값 저장
    pstack->topIndex -= 1;            // pop 연산의 결과로 topIndex 값 하나 감소

    return pstack->stackArr[rIdx];    //삭제되는 데이터 반환
}
```

1. 스택(STACK)

반환 – 가장 위에 있는 데이터를 반환

```
Data Speek(Stack* pstack)
{
    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->stackArr[pstack->topIndex]; // 맨 위에 저장된 데이터 반환
}
```

1. 스택(STACK)

연결 리스트 기반 스택의 구현

- 저장된 정보가 역순으로 조회(삭제)가 가능한 연결 리스트이다.
- 메모리 구조는 똑같은 모양을 하지만 **ADT**정의에서 함수의 역할이 다르다.

```
typedef struct _node                        // 연결 리스트의 노드를 표현한 구조체
{
    Data data;
    struct _node* next;
}Node;

Typedef struct _listStack                   // 연결 리스트 기반 스택을 표현한 구조체
{
    Node* head;
}ListStack;

typedef ListStack Stack;

Void StackInit(Stack* pstack);              // 스택 초기화
Int SIsEmpty(Stack* pstack);                // 스택이 비었는지 확인

Void Spush(Stack* pstack, Data data);       // 스택의 push 연산
Data Spop(Stack* pstack);                   // 스택의 pop 연산
Data Speek(Stack* pstack);                  // 스택의 peek 연산
```

1. 스택(STACK)

초기화와 삽입

초기화

```
void StackInit(Stack* pstack)
{
    pstack->head = NULL;          // 포인터 변수 head는 NULL로 초기화
}
```

삽입준비

```
int SIsEmpty(Stack* pstack)
{
    if(pstack->head == NULL)      // 스택이 비면 head에는 NULL이 저장된다.
        return TRUE;
    else
        return FALSE;
}
```

1. 스택(STACK)

삽입 및 삭제, 반환(반환은 제일 위에 있는 인덱스 DATA)

삽입

```
void Spush(Stack* pstack, Data data)
{
	Node* newNode = (Node*)malloc(sizeof(Node));

	newNode->data = data;
	newNode->next = pstack->head;

	pstack->head = newNode;
}
```

반환

```
Data Spe다(Stack* pstack)
{
	if(SIsEmpty(pstack))
	{
		printf(“Stack Memory Error!”);
		exit(-1);
	}

	return pstack->head->data;
}
```

1. 스택(STACK)

삭제

```
Data Spop(Stack* pstack)
{
    Data rdata;
    Node* rnode;

    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    rdata = pstack->head->data;          // 삭제할 노드의 데이터를 임시로 저장
    rdata = pstack->head;                // 삭제할 노드의 주소 값을 임시로 저장

    pstack->head = pstack->head->next;   // 삭제할 노드의 다음 노드를 head가 가리킴
    free(rnode);                         // 노드 삭제

    return rdata;                        // 삭제된 노드의 데이터 반환
}
```

1. 스택(STACK)

학습과제

- 제공된 Stack 이름이 붙은 프로젝트들을 공부하자.
- CLinkedList.h, CLinkedList.c를 변경없이 활용만해서 스택을 구현해보자.
- 제공된 계산기 프로그램을 확인하고 스택의 활용을 파악해 보자.

큐(QUEUE)

2. 큐(QUEUE)

- 선입선출(**FIFO – First In, First Out**)
- 먼저 기다린 사람이 먼저 배식을 받는다.

큐의 핵심 연산

Enqueue – 큐에 데이터를 넣는 연산
Dequeue – 큐에서 데이터를 꺼내는 연산

큐의 ADT 정의

void QueueInit(Queue* pq);
- 큐의 초기화를 진행, 생성 후 제일 먼저 호출되어야 한다.

int QIsEmpty(Queue* pq);
- 큐가 빈 경우 **TRUE(1)**을, 그렇지 않은 경우 **FALSE(0)**을 반환한다.

void Enqueue(Queue* pq, Data data);
- 큐에 데이터를 저장한다. 매개변수 **data**로 전달된 값을 저장한다.

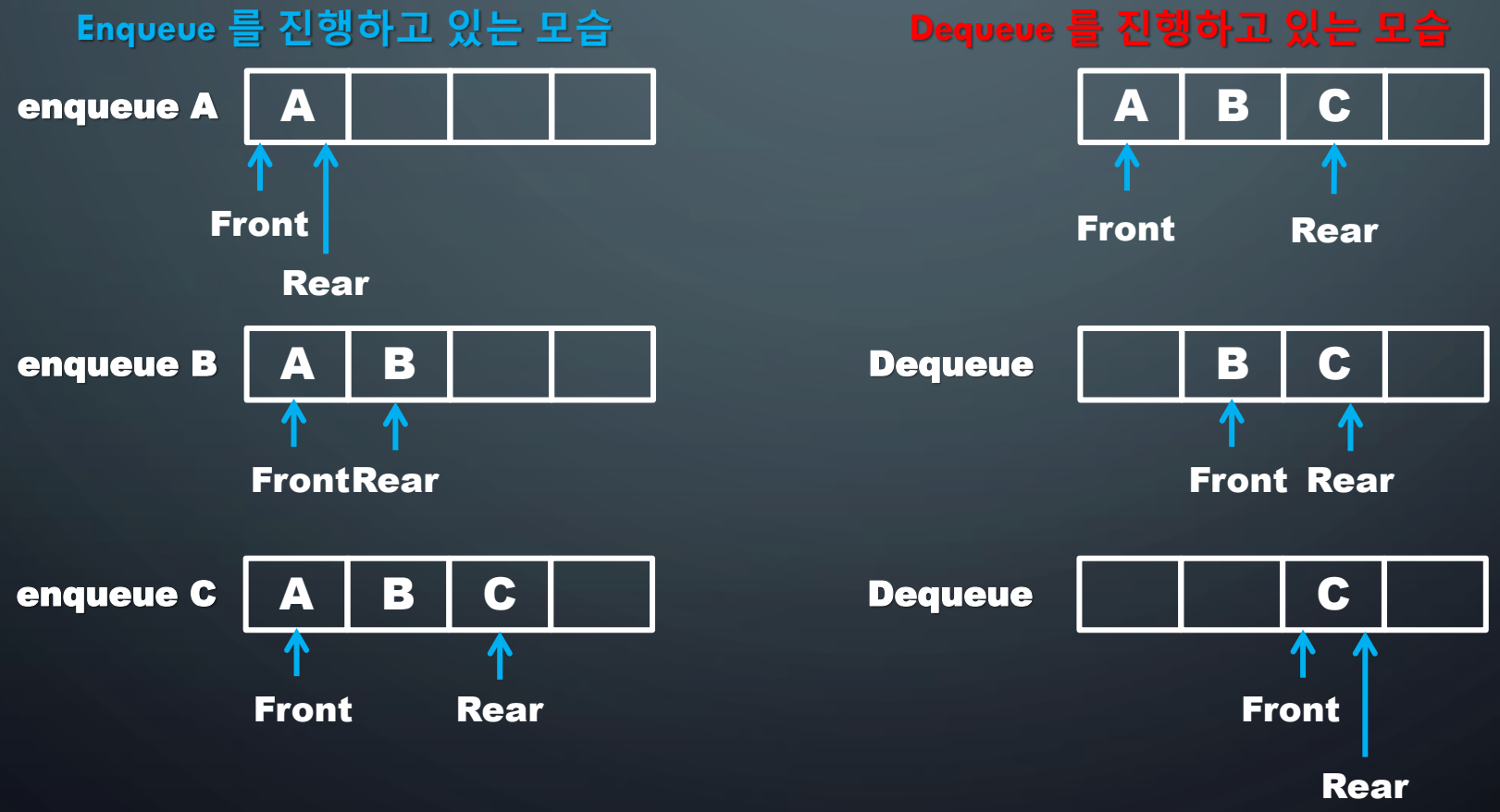
Data Dequeue(Queue* pq);
- 저장 순서가 가장 앞선 데이터를 삭제한다.
- 삭제된 데이터는 반환된다.
- 본 함수는 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

Data QPeek(Queue* pq);
- 저장순서가 가장 앞선 데이터를 반환하되 삭제하지 않는다.
- 본 함수는 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

2. 큐(QUEUE)

배열 기반의 큐

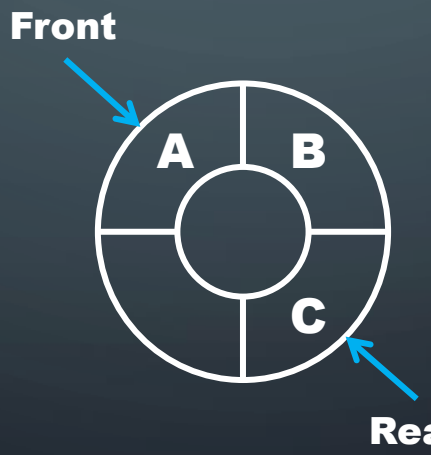
- 데이터가 증가할 때는 뒤에서 부터 붙고 반환할 때는 앞에서 부터 반환한다.



2. 큐(QUEUE)

- 배열의 끝까지 갔지만 꽉 차지 않은 상황이 발생한 경우

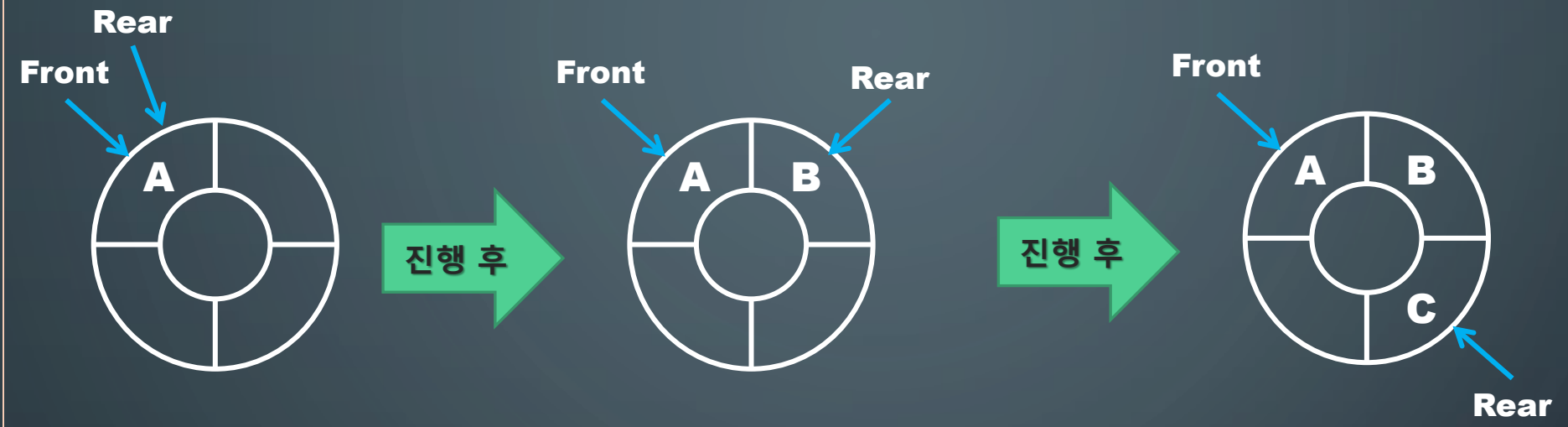
- 아직 채워야 할 공간이 있기 때문에 **Rear**를 다시 오른쪽으로 옮겨줘야 하고 선입선출을 하기 위해서 배열처럼 옮겨주거나 **Rear**가 다시 처음으로 돌아가고 **Front**도 같이 따라가는 구조가 되어야 한다.



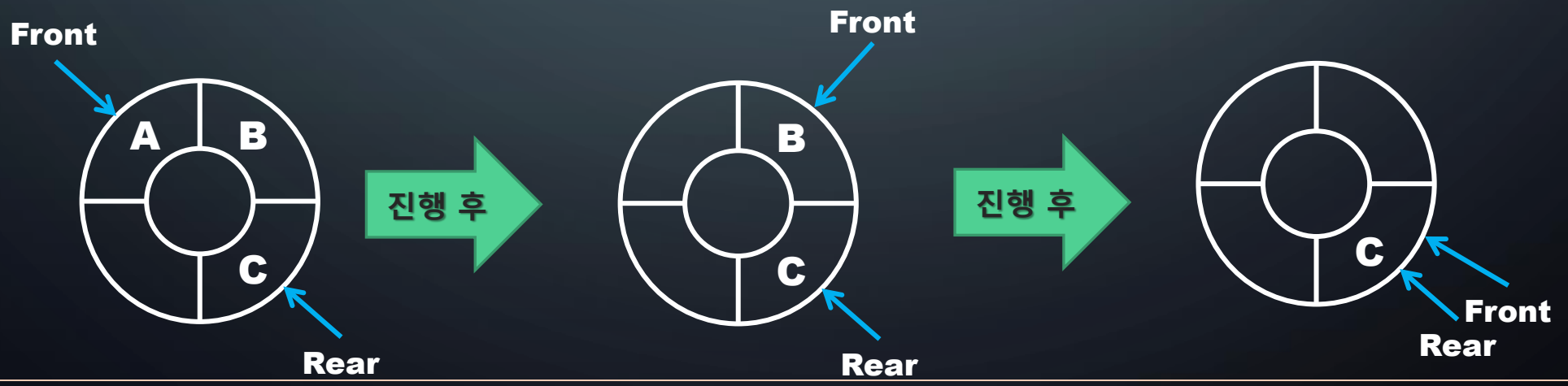
원형 큐

2. 큐(QUEUE)

원형 큐 Enqueue 연산

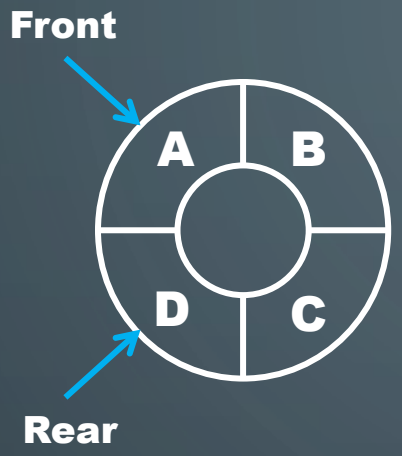


원형 큐의 Dequeue 연산

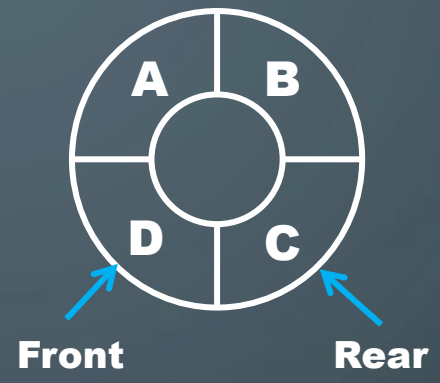


2. 큐(QUEUE)

- **4**곳이 **Full**로 차버린 경우와 모두 비운 경우가 된다면 꽉 찼는지 다 비었는지 알 수 없다.
- **Front**와 **Rear**의 위치로는 절대 알 수가 없다.

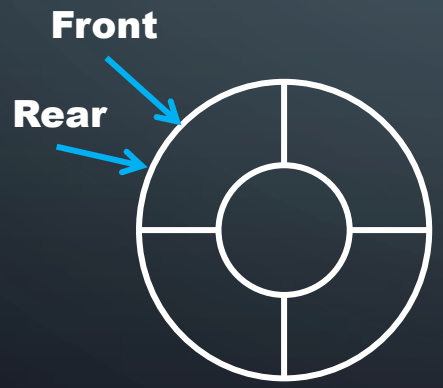


FULL

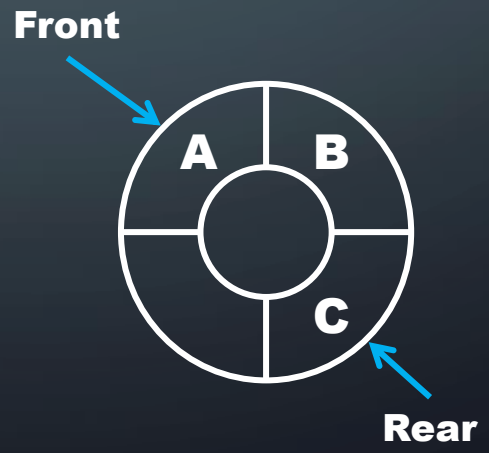


EMPTY

- 전체의 양보다 **1**작은 양**(n - 1)**을 채우게 설정하면 다 비었는지 꽉 찼는지 구분 할 수 있다.



EMPTY



FULL

2. 큐(QUEUE)

- **Enqueue** 연산 시, **R**이 가리키는 위치를 한 칸 이동시킨 다음에, **R**이 가리키는 위치에 데이터를 저장한다. – **원형 큐가 꽉 찬 상태는 R이 가리키는 위치의 앞을 F가 가리킨다.**
- **Dequeue** 연산 시, **F**가 가리키는 위치를 한 칸 이동시킨 다음에, **F**가 가리키는 위치에 저장된 데이터를 반환 및 소멸한다. – **원형 큐가 텅 빈 상태는 F와 R이 동일한 위치를 가리킨다.**

원형 큐의 구현

```
#define QUE_LEN100
typedef int Data;

typedef struct _cQueue
{
	int front;
	int rear;
	Data queArr[QUE_LEN];
} CQueue;

typedef CQueue Queue;

void QueueInit(Queue * pq);
int QIsEmpty(Queue * pq);

void Enqueue(Queue * pq, Data data);
Data Dequeue(Queue * pq);
Data QPeek(Queue * pq);
```

2. 큐(QUEUE)

```
int NextPosIdx(int pos)
{
	if (pos == QUE_LEN - 1)
		return 0;
	else
		return pos + 1;
}
```

- 큐에서는 다음 위치를 체크해서 위치를 반환하는 함수가 구현의 핵심**!**
- **Enqueue, Dequeue** 함수에서 **Front**와 **Rear**을 옮겨주는 역할을 해준다**.**

2. 큐(QUEUE)

LinkedList기반 큐의 구현

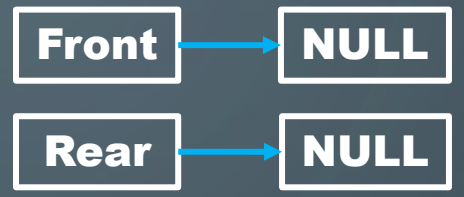
- **LinkedList**기반의 스택에서 반환하는 부분만 변경하면 큐가 된다.
- 스택의 **push**와 **pop**이 이뤄지는 위치가 같지만 큐는 **enqueue**와 **dequeue**가 이뤄지는 위치가 다르다.

```
typedef struct _node
{
	Data data;
	struct _node * next;
} Node;

typedef struct _lQueue
{
	Node * front;
	Node * rear;
} LQueue;

typedef LQueue Queue;

void QueueInit(Queue * pq);
int QIsEmpty(Queue * pq);

void Enqueue(Queue * pq, Data data);
Data Dequeue(Queue * pq);
Data QPeek(Queue * pq);
```

2. 큐(QUEUE)

초기화

- **Front**와 **Rear**이 가리키는 대상이 없는 상태가 비어 있는 상태이므로 **NULL**로 초기화




```
void QueueInit(Queue * pq)
{
	pq->front = NULL;
	pq->rear = NULL;
}
```

삽입

- **Rear**을 제외한 **Front**만 참조하여 큐가 비었는지 판단 하면, 텅 비게 되는 경우에도 **Front**만을 신경 쓰면 되기 때문에 여러모로 편리하다.

```
int QIsEmpty(Queue * pq)
{
	if (pq->front == NULL)
		return TRUE;
	else
		return FALSE;
}
```

2. 큐(QUEUE)

Enqueue

- 첫 번째 노드가 추가될 때에는 Front뿐만 아니라 Rear도 새 노드를 가리키게 설정
- 두 번째 이후의 노드가 추가될 때에는 F는 변함없지만 Rear는 새 노드를 가리키게 설정.

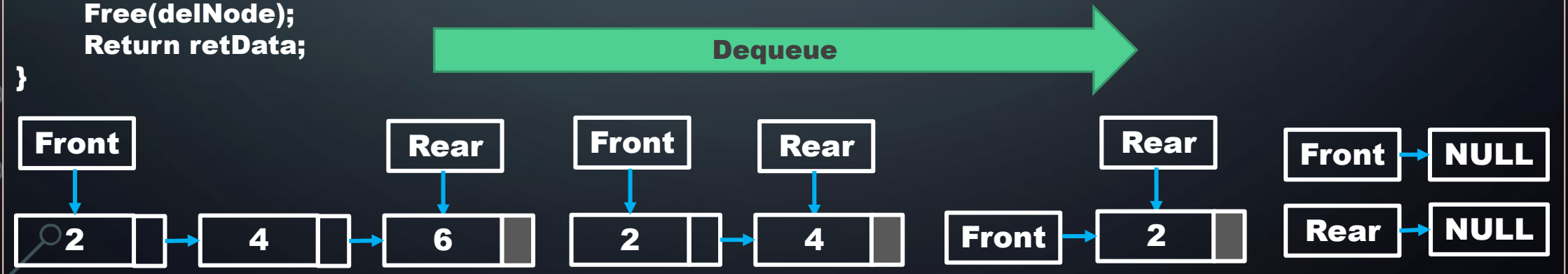
```
void Enqueue(Queue * pq, Data data)
{
	Node * newNode = (Node*)malloc(sizeof(Node));
	newNode->next = NULL;
	newNode->data = data;

	if (QIsEmpty(pq))                    // 첫 번째 노드의 추가라면,
	{
		pq->front = newNode;         // Front가 새 노드를 가리키게 하고,
		pq->rear = newNode;          // Rear도 새 노드를 가리키게 한다.
	}
	else                                 // 두 번째 이후의 노드 추가라면.
	{
		Pq->rear->next = newNode;    // 마지막 노드가 새 노드를 가리키게 하고,
		Pq->rear = newNode;          // Rear가 새 노드를 가리키게 한다.
	}
}
```

2. 큐(QUEUE)

Dequeue

- **Rear**은 고정시키고 **Front**가 다음 노드를 가리키게 하면 된다.
- **Front**가 이전에 가리키고 있던 노드를 소멸시킨다.

```
Data Dequeue(Queue * pq)
{
    Node * delNode;
    Data retData;

    if (QIsEmpty(pq))
    {
        printf("Queue Memory Error!");
        exit(-1);
    }

    delNode = pq->front;            // 삭제할 노드의 주소 값 저장
    retData = delNode->data;        // 삭제할 노드가 지닌 값 저장
    Pq->front = pq->front->next;    // 삭제할 노드의 다음 노드를 Front가 가리킴

    Free(delNode);
    Return retData;
}
```



2. 큐(QUEUE)

제공된 예제와 Queue의 활용 코드를 살펴보자.

덱(DEQUE)

2. 덱(DEQUE)

- 양쪽으로 삽입 및 조회 삭제가 가능하다.
- **head**와 **tail**이 모두 존재하는 양방향 연결 리스트를 활용할 수 있다.
- **Deque**를 이용해서 **Queue**를 구현 할 수 있다.

Deque의 ADT 정의

void DequeInit(Deque * pdeq);
- **Deque**의 초기화를 진행한다.
- **Deque** 생성 후 제일 먼저 호출되어야 하는 함수.

int DQIsEmpty(Deque * pdeq);
- **Deque**이 빈 경우 **TRUE(1)**을, 그렇지 않은 경우 **FALSE(0)**을 반환한다.

void DQAddFirst(Deque * pdeq, Data data);
- **Deque**의 머리에 데이터를 저장한다. **Data**로 전달된 값을 저장한다.

Void DQAddLast(Deque * pdeq, Data data);
- **Deque**의 꼬리에 데이터를 저장한다. **Data**로 전달된 값을 저장한다.

Data DQRemoveFirst(Deque * pdeq);
- **Deque**의 머리에 위치한 데이터를 반환 및 소멸한다.

Data DQRemoveLast(Deque * pdeq);
- **Deque**의 꼬리에 위치한 데이터를 반환 및 소멸한다.

Data DQGetFirst(Deque * pdeq);
- **Deque**의 머리에 위치한 데이터를 소멸하지 않고 반환한다.

Data DQGetLast(Deque * pdeq);
- **Deque**의 꼬리에 위치한 데이터를 소멸하지 않고 반환한다.

2. 덱(DEQUE)

```
typedef struct _node
{
	Data data;
	struct _node * next;
	struct _node * prev;
} Node;

typedef struct _dlDeque
{
	Node * head;
	Node * tail;
} DLDeque;

typedef DLDeque Deque;

void DequeInit(Deque * pdeq);
int DQIsEmpty(Deque * pdeq);

void DQAddFirst(Deque * pdeq, Data data);
void DQAddLast(Deque * pdeq, Data data);

Data DQRemoveFirst(Deque * pdeq);
Data DQRemoveLast(Deque * pdeq);

Data DQGetFirst(Deque * pdeq);
Data DQGetLast(Deque * pdeq);

#endif
```

2. 덱(DEQUE)

학습과제

- 양방향 연결 리스트의 구조로 만들어진 **Deque**를 확인해보자**.**
- **Deque**를 이용한 **Queue**를 만들어보자**.**