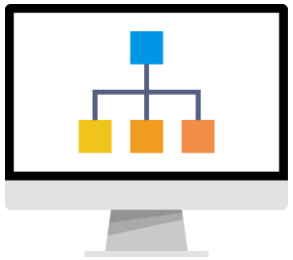


Data Structure & Algorithm
자료구조와 알고리즘

12. 큐 (Queue)

10강 보고서 돌아보기

- 스택의 push, pop, peek, isEmpty 연산의 시간 복잡도는 얼마일까?

- 스택 두 개로 큐를 만들 수 있을까? 가능하다면 어떻게 하면 될까?

재귀 호출 연습 1

- 제곱 합 구하기

```c
#include <stdio.h>

int ssum(int n) {
    if(n == 1) return 1;

    return n * n + ssum(n - 1);
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", ssum(n));
    return 0;
}
```

재귀 호출 연습 2

- 확장된 피보나치 수열

$$f_1 = 1$$
$$f_2 = 1$$
$$f_3 = 2$$

4이상의 n에 대해

$$f_n = f_{n-1} + f_{n-2} + f_{n-3}$$

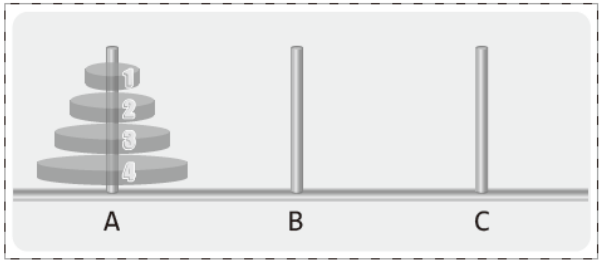
```c
#include <stdio.h>

int f(int n) {
    if(n == 1) return 1;
    if(n == 2) return 1;
    if(n == 3) return 2;

    return f(n-1) + f(n-2) + f(n-3);
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", f(n));
    return 0;
}
```

재귀 호출 연습 3

- 1과 2로 자연수를 만드는 방법

```c
#include <stdio.h>

int fibo(int n) {
    if(n == 1) return 1;
    if(n == 2) return 2;

    return fibo(n-1) + fibo(n-2);
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", fibo(n));
    return 0;
}
```

하노이 타워의 이동 횟수



▶ [그림 02-14: 원반이 4개인 하노이 타워]

큐의 이해와 ADT 정의

큐(Queue)의 이해와 ADT 정의

큐는 'FIFO(First-in, First-out) 구조'의 자료구조이다.
때문에 먼저 들어간 것이 먼저 나오는, 일종의
줄서기에 비유할 수 있는 자료구조이다.

- 큐에 데이터를 넣는 연산 — enqueue
- 큐에서 데이터를 꺼내는 연산 — dequeue

} '큐'의 기본 연산

큐는 운영체제 관점에서 보면 프로세스나 쓰레드의 관리에 활용이 되는 자료구조이다. 이렇듯 운영체제의 구현에도 자료구조가 사용이 된다. 따라서 운영체제의 이해를 위해서는 자료구조에 대한 이해가 선행되어야 한다.

큐의 ADT 정의

ADT를 대상으로 배열 기반의 큐

또는 연결 리스트 기반의 큐를 구현할 수 있다.

- `void QueueInit(Queue * pq);`
  - 큐의 초기화를 진행한다.
  - 큐 생성 후 제일 먼저 호출되어야 하는 함수이다.

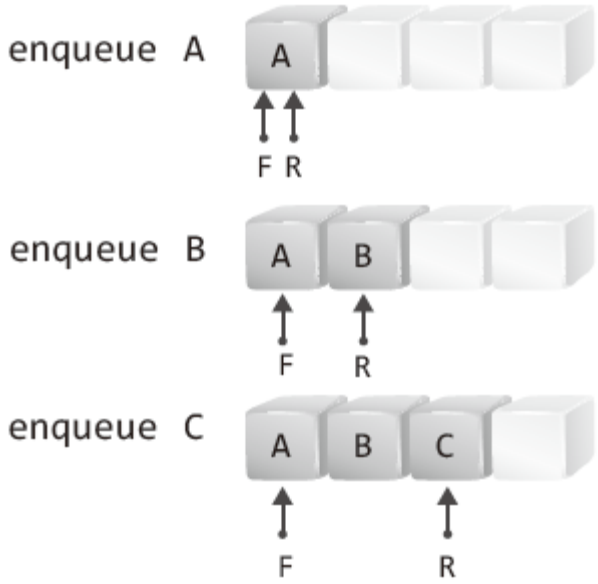
- `int QIsEmpty(Queue * pq);`
  - 큐가 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.

- `void Enqueue(Queue * pq, Data data);` enqueue 연산
  - 큐에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.

- `Data Dequeue(Queue * pq);` dequeue 연산
  - 저장순서가 가장 앞선 데이터를 삭제한다.
  - 삭제된 데이터는 반환된다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

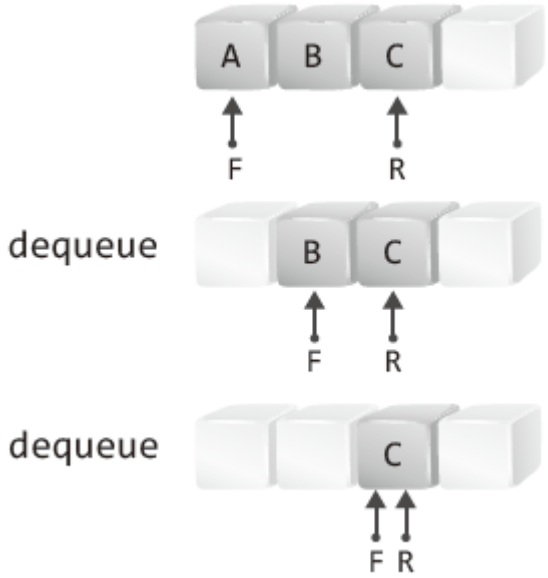
- `Data QPeek(Queue * pq);` peek 연산
  - 저장순서가 가장 앞선 데이터를 반환하되 삭제하지 않는다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

큐의 구현 논리



보편적이고도 올바른
enqueue 연산에 대한 방식

보편적이고도 올바른
dequeue 연산에 대한 방식

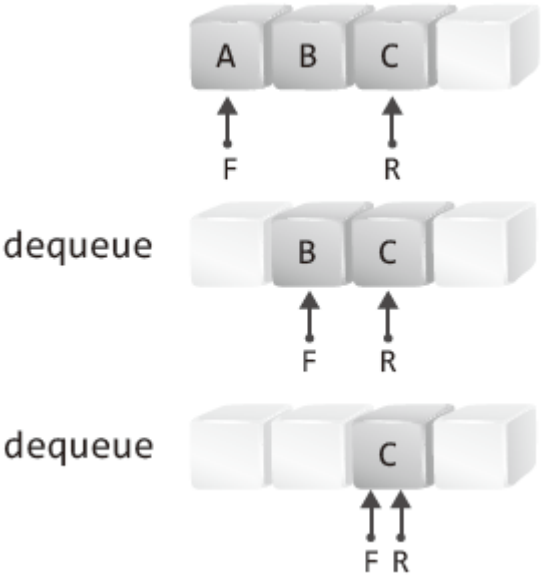


큐의 꼬리를 의미하는 R을 한칸 이동시키고

새 데이터를 저장한다.

큐의 머리를 의미하는 F가 가리키는 데이터를 반환하고 F를 한 칸 이동시킨다.

가장 기본적인 배열 기반 큐의 문제점

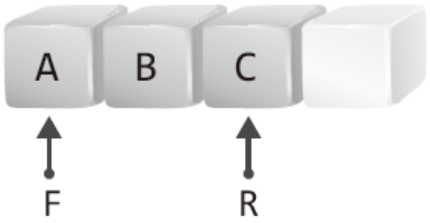


분명 배열은 비어있다. 하지만 R을 오른쪽으로 한칸 이동시킬 수 없어서 더 이상 데이터를 추가할 수 없다!

데이터를 더 추가하기 위해서는 R을 인덱스가 0인 위치로 이동시켜야 한다.

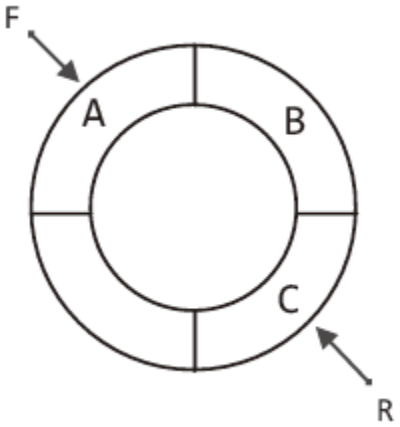
그리고 이러한 방식으로 문제를 해결한 것이 바로 '원형 큐'이다!

원형 큐의 소개

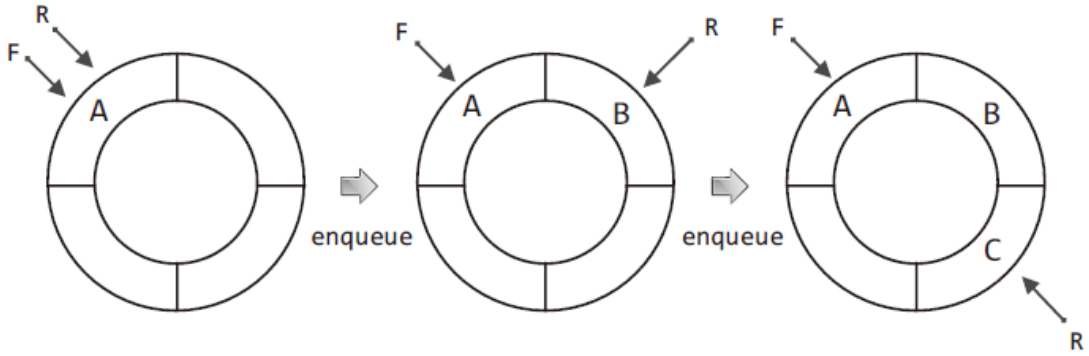


배열의 머리와 끝을 연결한 구조를

원의 형태로 바라본다.



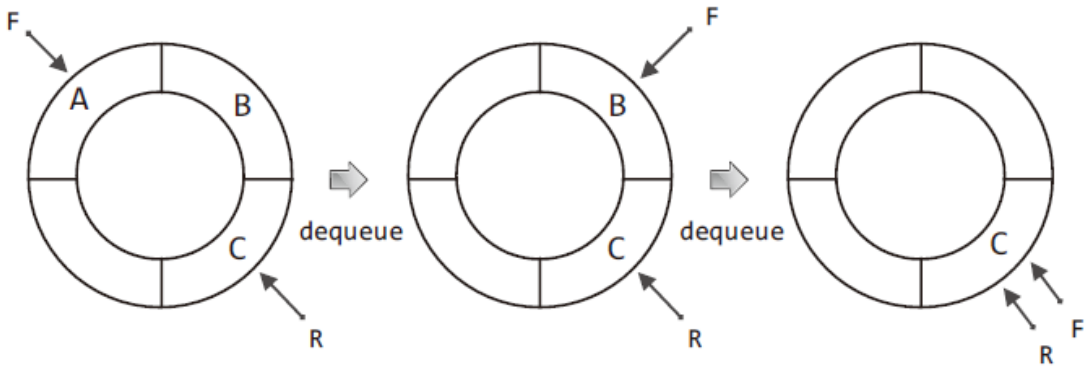
원형 큐의 단순한 연산



▶ [그림 07-6: 원형 큐의 enqueue 연산]

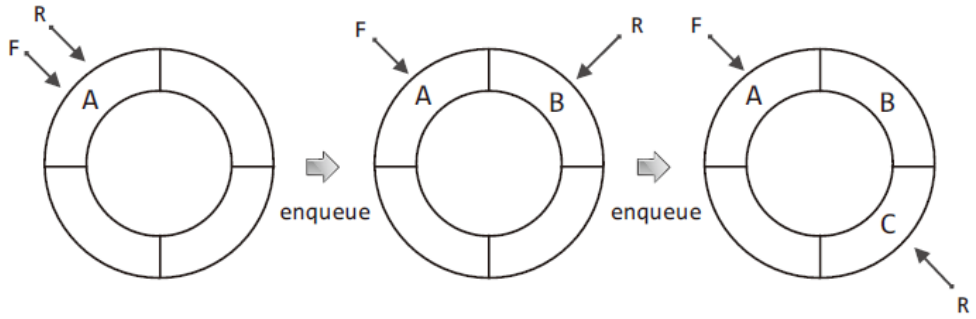
단순 배열 큐와 마찬가지로 R이
이동한 다음에 데이터 저장!

단순 배열 큐와 마찬가지로 F가
가리키는 데이터 반환 후 F 이동!



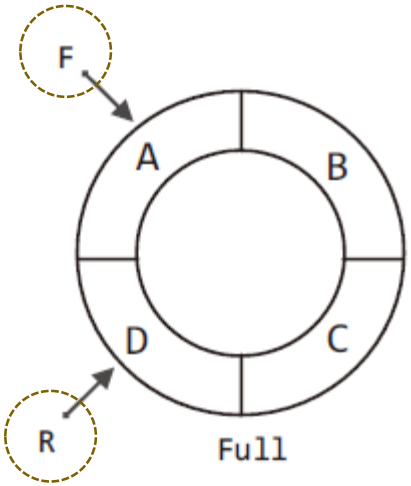
▶ [그림 07-7: 원형 큐의 dequeue 연산]

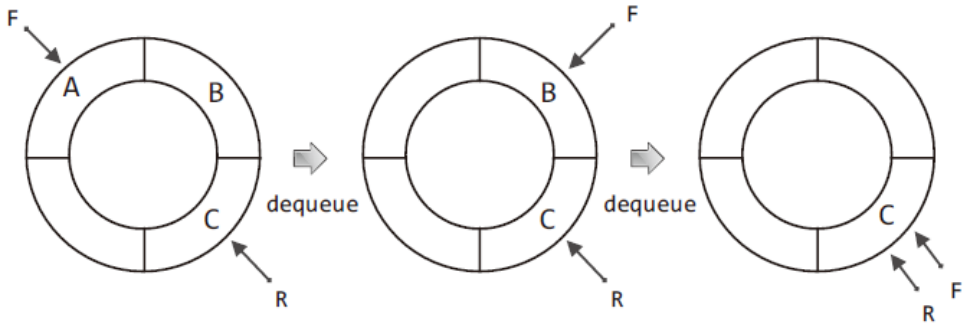
원형 큐의 단순한 연산의 문제점



▶ [그림 07-6: 원형 큐의 enqueue 연산]

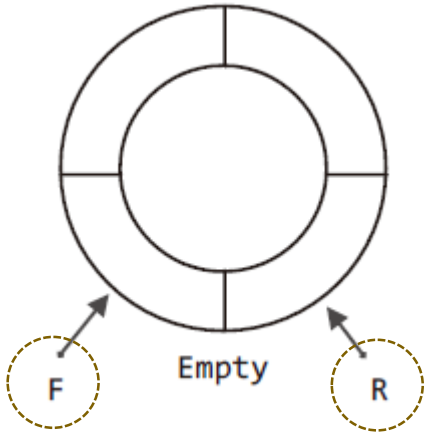
꽉 채운다!



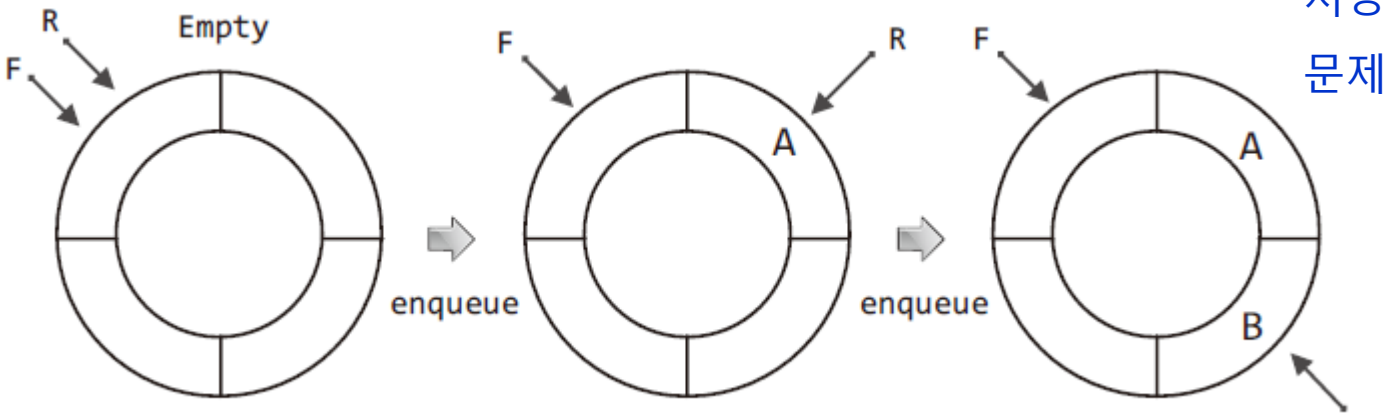


▶ [그림 07-7: 원형 큐의 dequeue 연산]

텅 비운다!



원형 큐의 문제점 해결



저장 공간 하나를 잃고

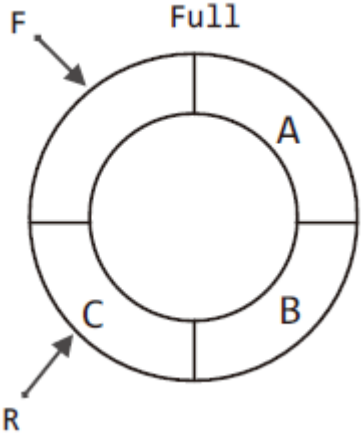
문제점을 해결한다!

데이터가 하나 존재하는 경우 F와 R이 같은 위치를

가리켰는데, 초기화 직후에 이 상태가 되게 한다.

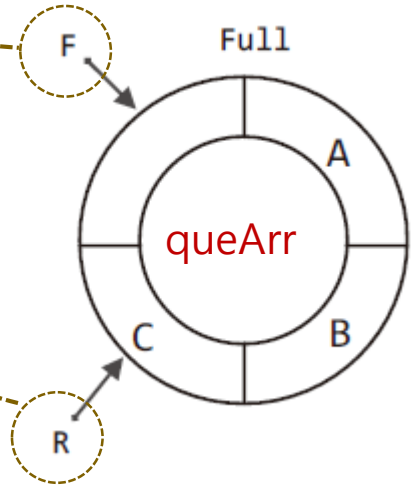
그냥 하나 비운 상태에서 FULL로 인정한다!

그러면 Empty와 Full의 구분이 가능하다!



원형 큐의 구현: 헤더파일

```
#define QUE_LEN 100
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);
```



원형 큐의 구현: Helper Function

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

```
int NextPosIdx(int pos)
{
    return (pos + 1) % QUE_LEN;
}
```

큐의 연산에 의해서 F(front)와 R(rear)이 이동할때 이동해야 할

위치를 알려주는 함수! 원형 큐를 완성하게 하는 실질적인 함수이다!

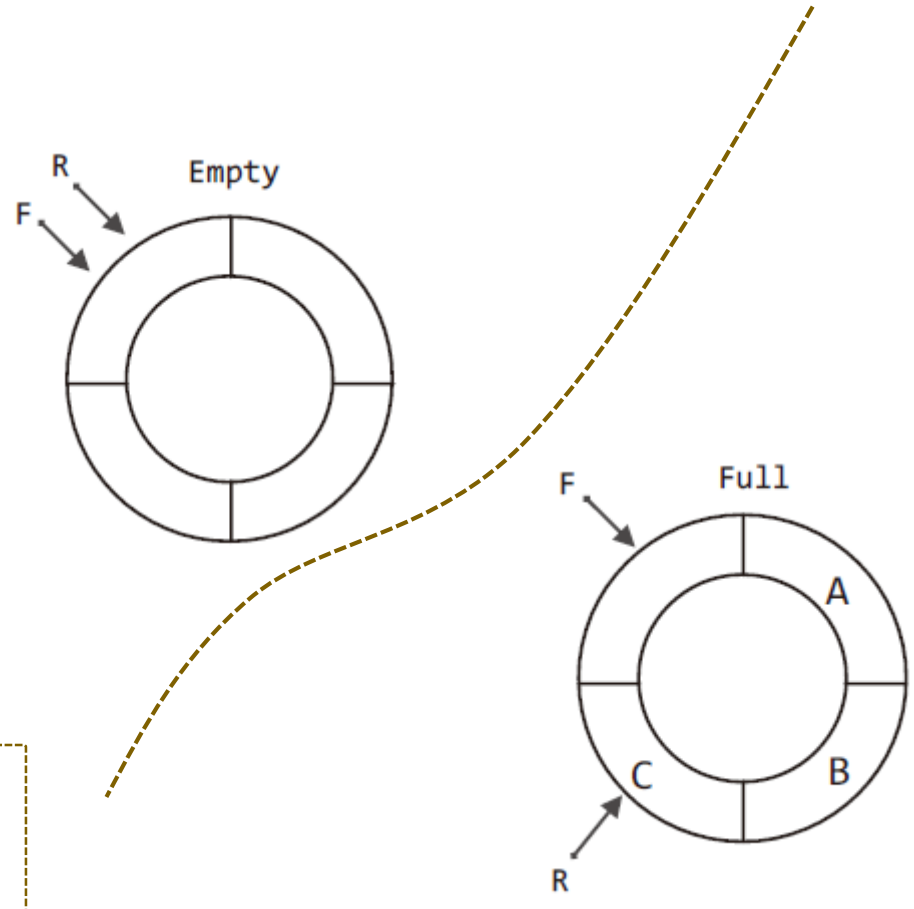
위 함수의 정의를 통해서 원형 큐의 나머지 구현은 매우 간단해진다.

원형 큐의 구현: 함수의 정의1

```
void QueueInit(Queue * pq)
{
    pq->front = 0;
    pq->rear = 0;
}
```

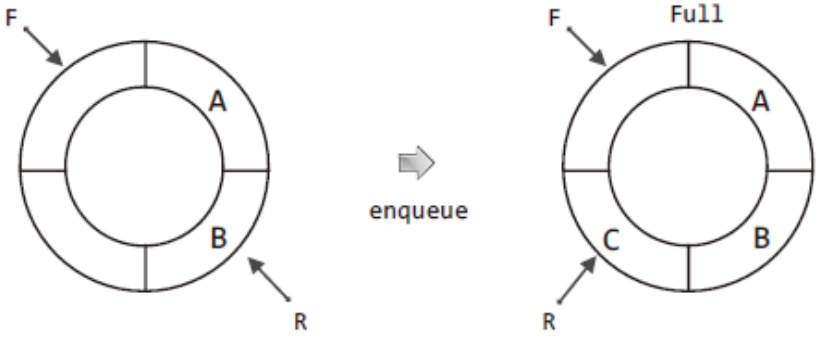
```
int QIsEmpty(Queue * pq)
{
    if(pq->front == pq->rear)
        return TRUE;
    else
        return FALSE;
}
```

```
int QIsEmpty(Queue *pq)
{
     return pq->front == pq->rear;
}
```



원형 큐의 구현: 함수의 정의2

```
void Enqueue(Queue * pq, Data data)
{
    if(NextPosIdx(pq->rear) == pq->front)
    {
        printf("Queue Memory Error!");
        exit(-1);
    }
    pq->rear = NextPosIdx(pq->rear);
    pq->queArr[pq->rear] = data;
}
```



rear을 이동시키고(NextPosIdx 함수의 호출을 통해),
그 위치에 데이터 저장

두 연산 모두 rear와 front를 우선 이동시키는 구조이다!

front를 이동시키고(NextPosIdx 함수의 호출을 통해),
그 위치의 데이터 반환

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

    pq->front = NextPosIdx(pq->front);
    return pq->queArr[pq->front];
}
```

원형 큐의 실행

```
int main(void)
{
    // Queue의 생성 및 초기화 ///////
    Queue q;
    QueueInit(&q);

    // 데이터 넣기 ///////
    Enqueue(&q, 1); Enqueue(&q, 2);
    Enqueue(&q, 3); Enqueue(&q, 4);
    Enqueue(&q, 5);

    // 데이터 꺼내기 ///////
    while(!QIsEmpty(&q))
        printf("%d ", Dequeue(&q));

    return 0;
}
```

CircularQueue.h
CircularQueue.c
원형 큐의 구현 결과

CircularQueueMain.c main 함수의 정의

```
1 2 3 4 5
```

실행결과

큐의 연결 리스트 기반 구현

연결 리스트 기반 큐의 헤더파일

```
typedef int Data;

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

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

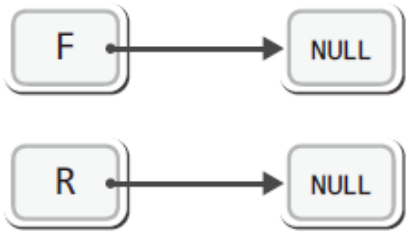
```
typedef LQueue Queue;

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

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

배열 기반의 구현보다 연결 리스트 기반의
구현에서 논의할 내용이 더 적다!

"스택과 큐의 유일한 차이점이 앞에서 꺼내느냐 뒤에서 꺼내느냐에 있으니, 이전에 구현해 놓은 스택을 대상으로 꺼내는 방법만 조금 변경하면 큐가 될 것 같다."

} 연결 리스트 기반 큐의 경우!

연결 리스트 기반 큐의 구현: 초기화

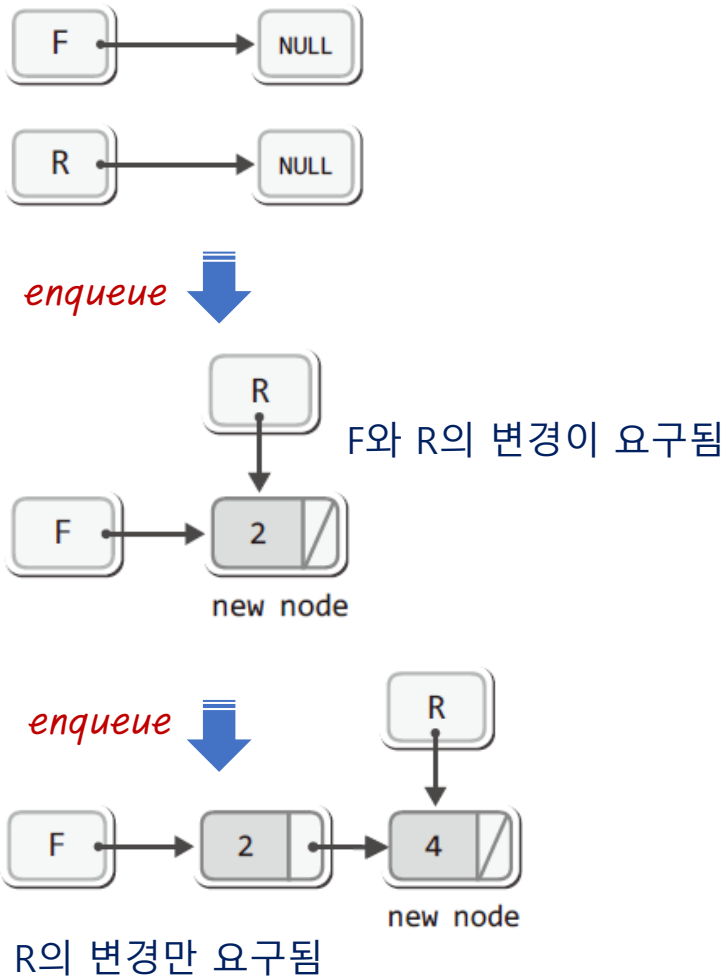


▶ [그림 07-12: 리스트 기반 큐의 초기상태]

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

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

연결 리스트 기반 큐의 구현: enqueue




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

    if(QIsEmpty(pq))
    {
        pq->front = newNode;
        pq->rear = newNode;
    }
    else
    {
        pq->rear->next = newNode;
        pq->rear = newNode;
    }
}
```

enqueue의 과정은 두 가지로 나뉜다!

연결 리스트 기반 큐의 구현: dequeue 논리

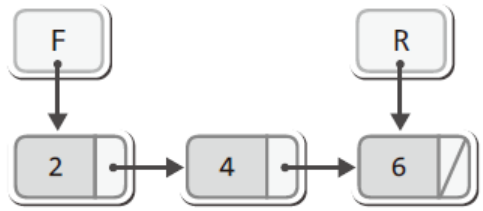
F가 다음 노드를 가리키게 하고, F가 이전에 가리키던 노드를 소멸시킨 결과!

마찬가지로! F가 다음 노드를 가리키게 하고, F가 이전에 가리키던 노드를 소멸시킨 결과!

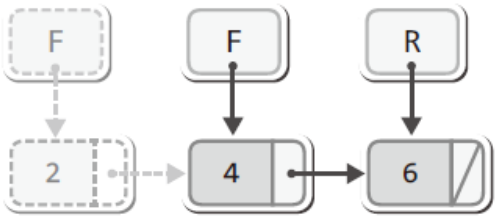
R은 그냥 내버려 둔다! 그래야 dequeue의 흐름을 하나로 유지할 수 있다!

그리고 R은 그냥 내버려 두어도 된다!

연결 리스트 기반 큐의 구현: dequeue 정의



dequeue




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

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

    delNode = pq->front;
    retData = delNode->data;
    pq->front = pq->front->next;

    free(delNode);
    return retData;
}
```

F가 다음 노드를 가리키게 하고!

노드를 삭제한다.

연결 리스트 기반 큐의 실행

```
int main(void)
{
    // Queue의 생성 및 초기화 ///////
    Queue q;
    QueueInit(&q);

    // 데이터 넣기 ///////
    Enqueue(&q, 1); Enqueue(&q, 2);
    Enqueue(&q, 3); Enqueue(&q, 4);
    Enqueue(&q, 5);

    // 데이터 꺼내기 ///////
    while(!QIsEmpty(&q))
        printf("%d ", Dequeue(&q));

    return 0;
}
```

ListBaseQueue.h
ListBaseQueue.c
} 연결 리스트 기반 큐의 구현결과

ListBaseQueueMain.c main 함수의 정의

```
1 2 3 4 5
```

실행결과

덱(Deque)의 이해와 구현

Double-Ended Queue

덱의 이해

"덱은 앞으로도 뒤로도 넣을 수 있고, 앞으로도 뒤로도 뺄 수 있는 자료구조!"

- 앞으로 넣기
- 앞에서 빼기
- 뒤로 넣기
- 뒤에서 빼기

덱의 4가지 연산!

모든 연산은 Pair를 이루지 않는다!

개별적으로 연산 가능!

Deque는 double ended queue의 줄인 표현으로, 양쪽 방향으로 모두 입출력이 가능함을 의미한다. 그리고 스택과 큐의 특성을 모두 지니고 있다고도 말한다. 덱을 스택으로도 큐로도 활용할 수 있기 때문이다.

덱의 ADT

- `void DequeInit(Deque * pdeq);`
  - 덱의 초기화를 진행한다.
  - 덱 생성 후 제일 먼저 호출되어야 하는 함수이다.

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

앞으로 넣기

- `void DQAddFirst(Deque * pdeq, Data data);`
  - 덱의 머리에 데이터를 저장한다. data로 전달된 값을 저장한다.

뒤로 넣기

- `void DQAddLast(Deque * pdeq, Data data);`
  - 덱의 꼬리에 데이터를 저장한다. data로 전달된 값을 저장한다.

앞에서 빼기

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

뒤로 빼기

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

- `Data DQGetFirst(Deque * pdeq);` 앞에서 PEEK
  - 덱의 머리에 위치한 데이터를 소멸하지 않고 반환한다.

- `Data DQGetLast(Deque * pdeq);` 뒤에서 PEEK
  - 덱의 꼬리에 위치한 데이터를 소멸하지 않고 반환한다.

덱의 구현: 헤더파일 정의

```
typedef int Data;

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

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

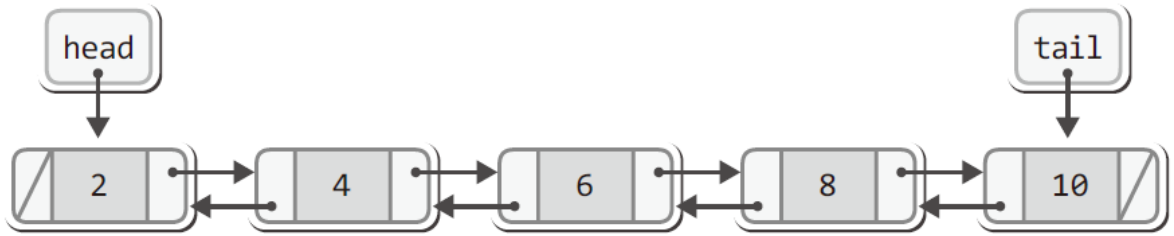
덱의 구현에 가장 어울리는 자료구조는 양방향 연결 리스트이다!

```
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);
```

앞과 뒤로 입출력 연산이 이뤄지기 때문에

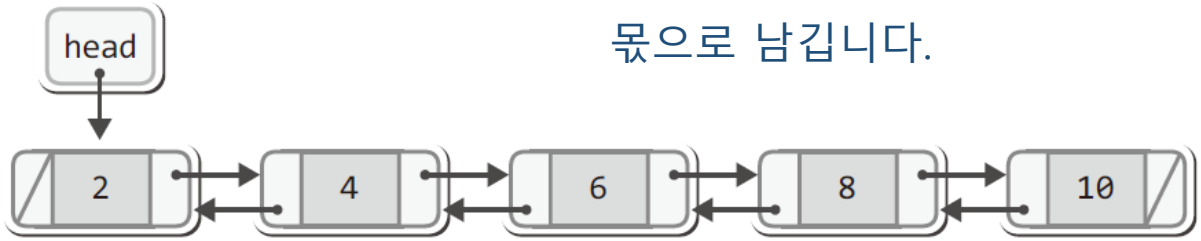
head뿐만 아니라 tail도 정의되어야 한다.

덱의 구현: 함수의 정의



덱의 구현을 위한 양방향 연결 리스트의 구조

이전에 구현한 양방향 연결 리스트와의 차이점은 tail의 존재 유무가 전부이니! 코드에 대한 해석은 여러분의 몫으로 남깁니다.



이전에 구현한 양방향 연결 리스트의 구조

요약

- 배열로 큐를 구현할 경우 메모리의 낭비를 막기 위해 원형 큐를 구현한다.

- 연결 리스트로 큐를 구현할 경우 연결리스트의 enqueue는 연결리스트의 꼬리에 데이터 추가, dequeue는 연결리스트의 머리에서 데이터 삭제로 구현한다.

- 덱은 큐와 스택의 특성을 모두 가지고 있으며 이중 연결리스트의 양 끝에서 삽입/삭제를 하는 것으로 구현한다.

출석 인정을 위한 보고서 작성

- A4 반 장 이상으로 아래 질문에 답한 후 포털에 있는 과제 제출 란에 PDF로 제출

- 큐의 enqueue, dequeue, peek, isEmpty 연산의 시간 복잡도는 얼마일까?

- 큐 두 개로 스택을 만들 수 있을까? 가능하다면 어떻게 하면 될까?