Queue

Data Structures and Algorithms

목차

- 큐의 이해와 ADT 정의
- 큐의 배열 기반 구현
- 큐의 연결 리스트 기반 구현
- 큐의 활용
- 덱(Deque)의 이해와 구현

큐의 이해와 ADT 정의

큐(Stack)의 이해와 ADT 정의

- 큐는 'LIFO(Last-in, First-out) 구조'의 자료구조
  - 먼저 들어간 것이 먼저 나오는 구조

- 큐의 기본 연산
  - Enqueue: 큐에 데이터를 넣는 연산
  - Dequeu: 큐에서 데이터를 꺼내는 연산

큐의 ADT 정의

- 배열 또는 연결리스트 기반 큐

- `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.

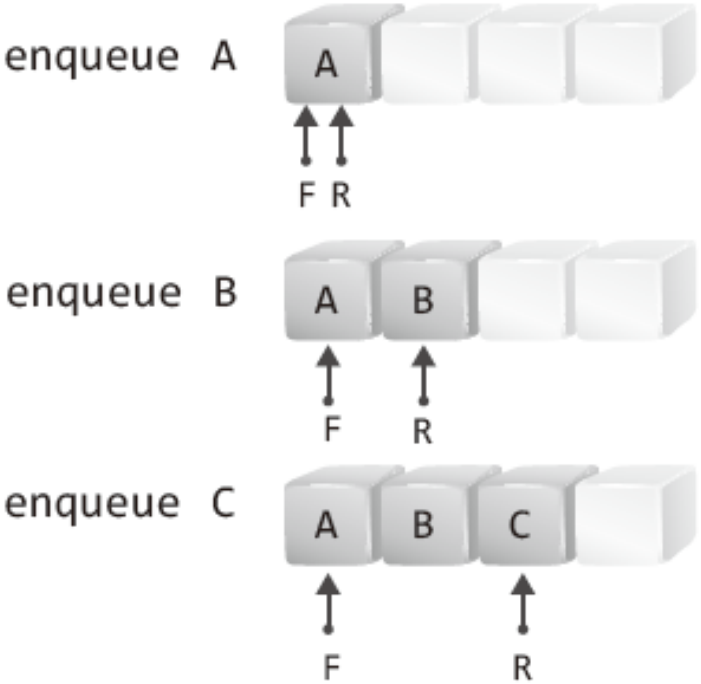
- `Data QPeek(Queue * pq);` peek 연산
  - 저장순서가 가장 앞선 데이터를 반환하되 삭제하지 않는다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

큐의 배열 기반 구현

큐의 동작

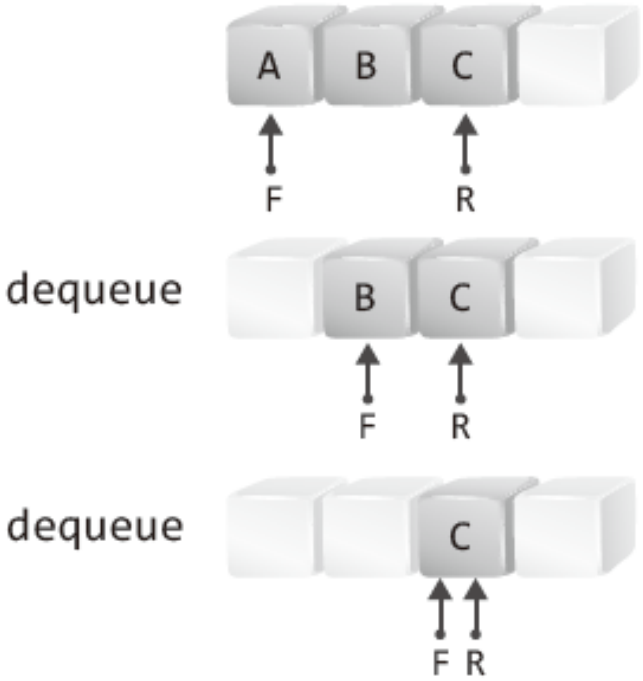
enqueue 연산

- 큐의 꼬리(R)을 한칸 이동 후 새 데이터 저장



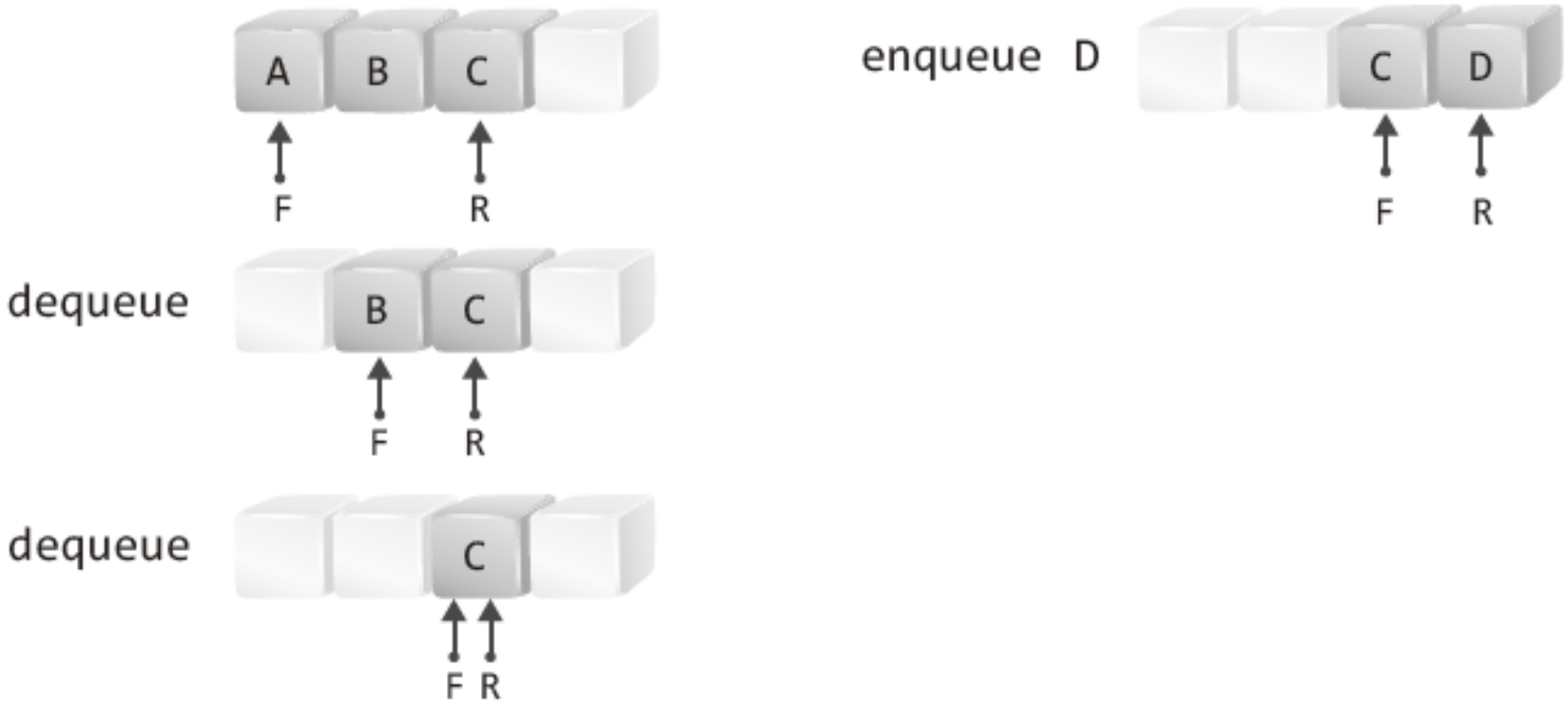
dequeue 연산

- 큐의 머리(F)가 가리키는 데이터 반환 후 F를 한 칸 이동



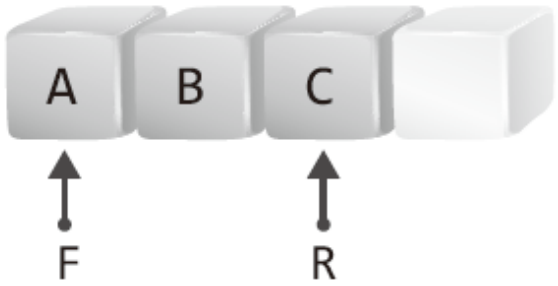
배열 기반 큐의 문제점

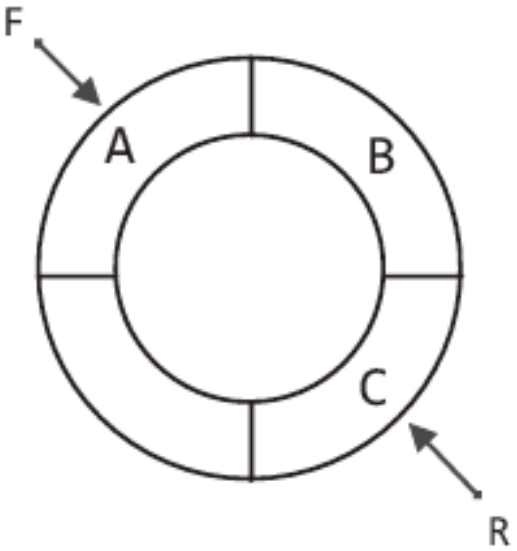
- Dequeue 동작으로 배열이 비더라도 인덱스 이상 증가 불가
  - 데이터 추가를 위해 R을 인덱스 0위치로 이동해야 함



원형 큐

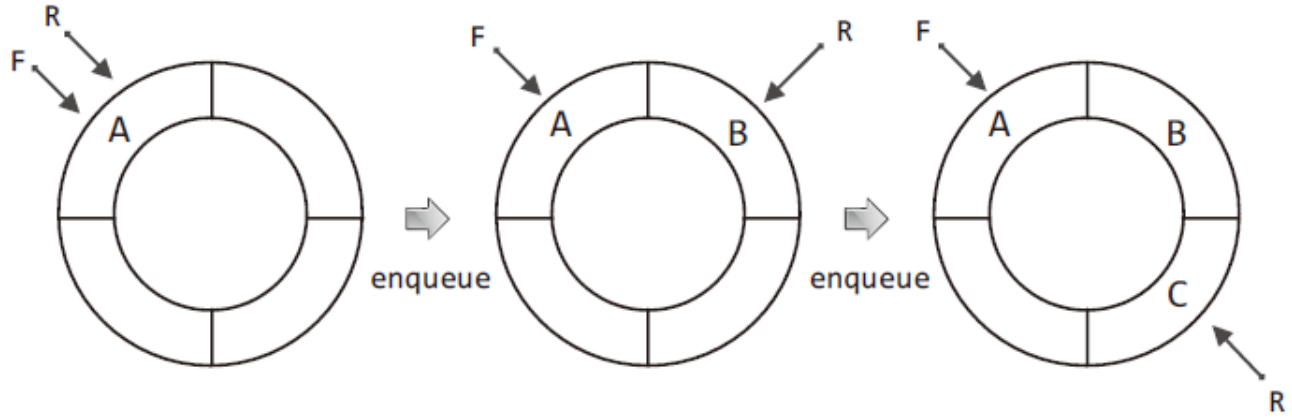
- 배열의 머리와 끝을 연결한 구조





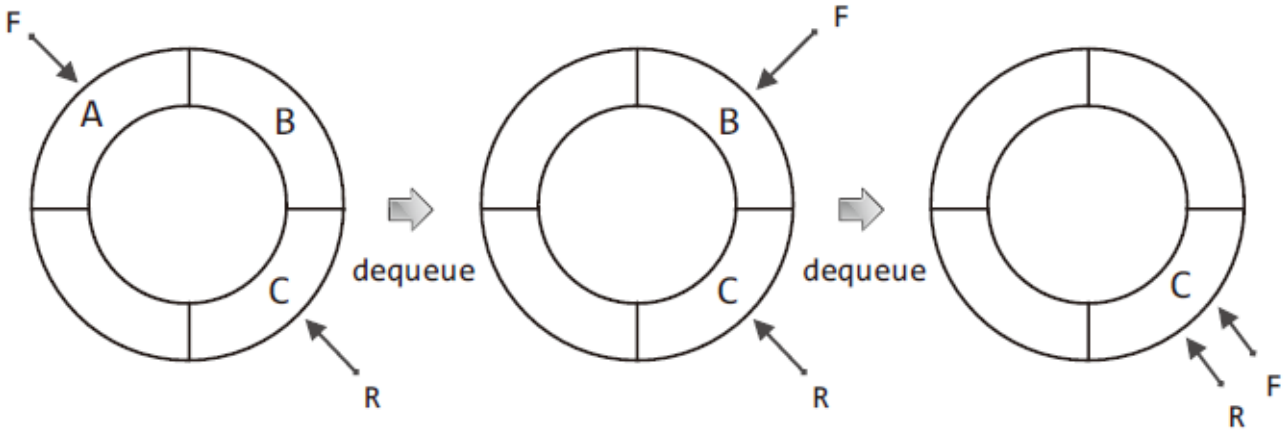
원형 큐의 단순한 연산

- R이 이동한 다음에 데이터 저장



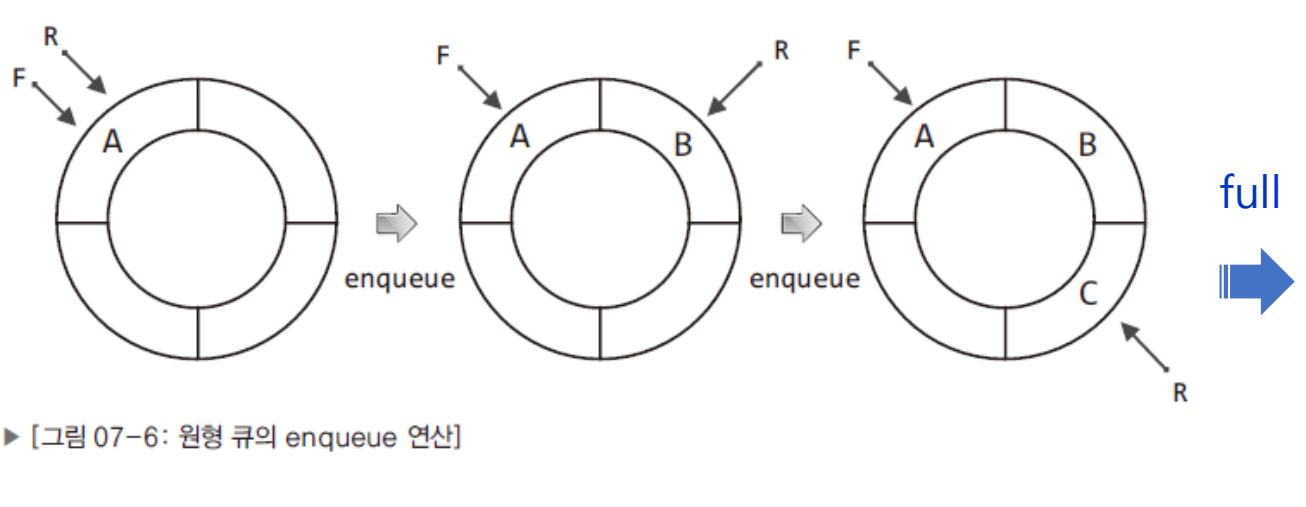
▶ [그림 07-6: 원형 큐의 enqueue 연산]

- F가 가리키는 데이터 반환 후 F 이동

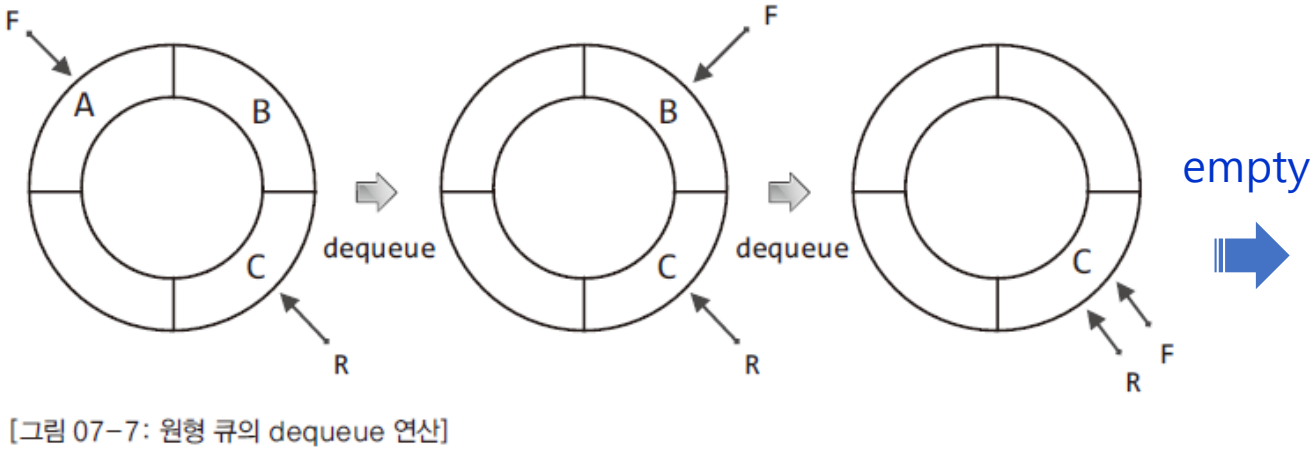


▶ [그림 07-7: 원형 큐의 dequeue 연산]

원형 큐의 단순한 연산의 문제점



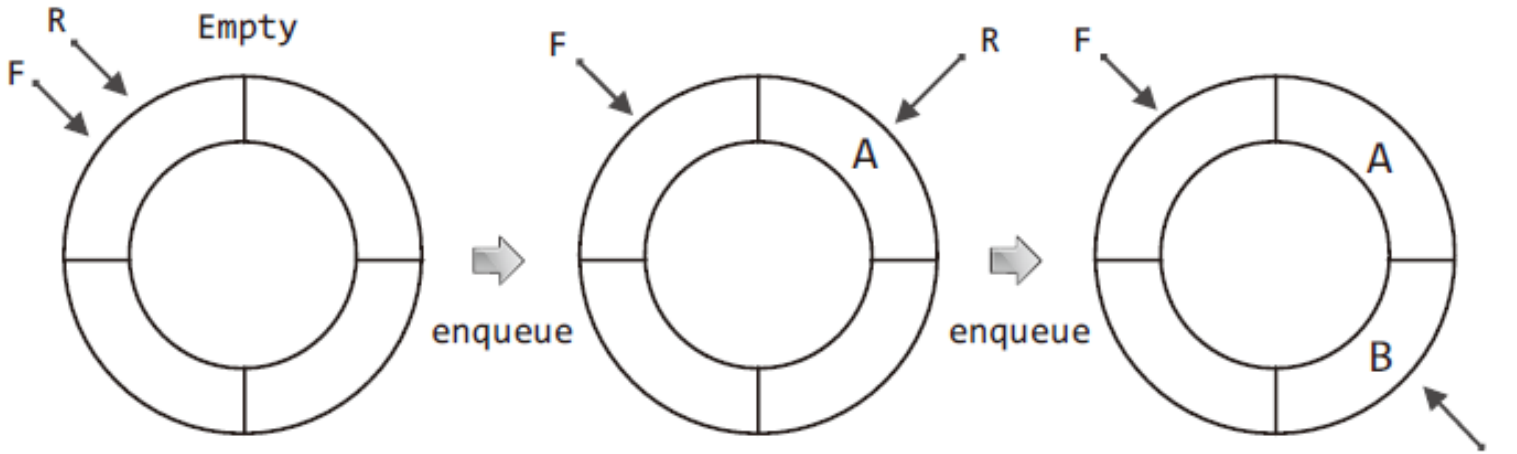
▶ [그림 07-6: 원형 큐의 enqueue 연산]



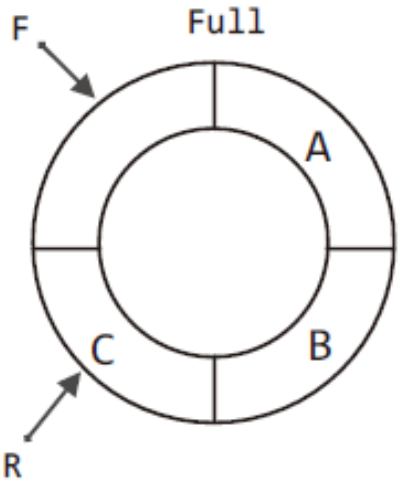
▶ [그림 07-7: 원형 큐의 dequeue 연산]

원형 큐의 문제점 해결

- 초기화: F와 R이 같은 위치를 가리킴

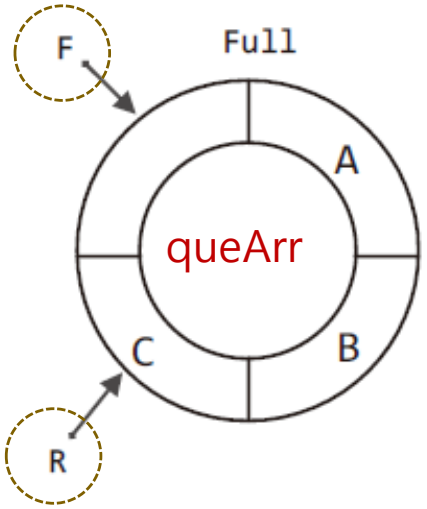


- R이 F를 가리키려고 하면 Full



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);
```



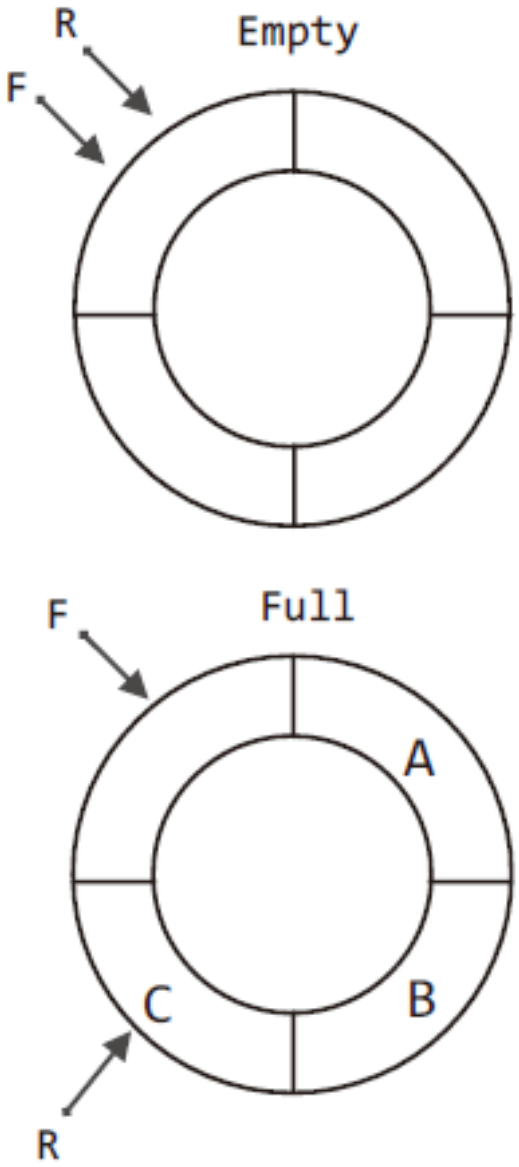
원형 큐의 구현: Helper Function

- 큐의 연산에 의해서 F(front)와 R(rear)이 이동할때 이동해야 할 위치를 알려주는 함수

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

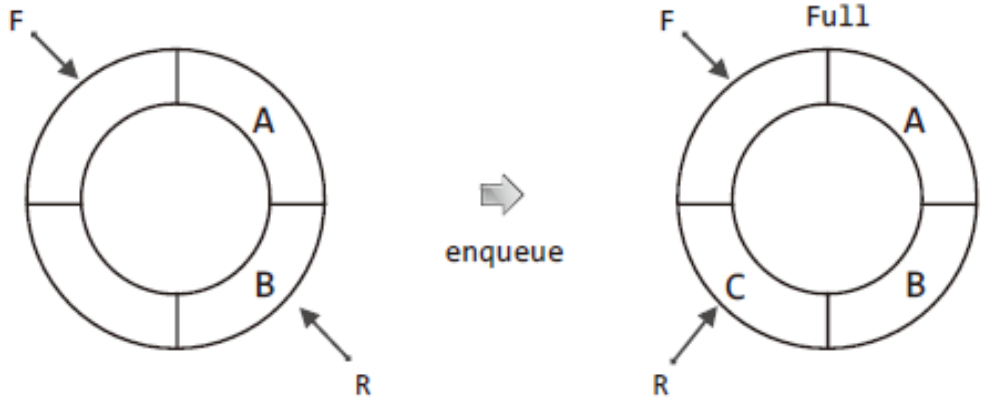
원형 큐의 구현: 함수 정의1

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


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



원형 큐의 구현: 함수 정의2

```
void Enqueue(Queue * pq, Data data)
{ // rear이동 후 데이터 저장
    if(NextPosIdx(pq->rear) == pq->front)
    {
        printf("Queue Memory Error!");
        exit(-1);
    }

    pq->rear = NextPosIdx(pq->rear);
    pq->queArr[pq->rear] = data;
}

Data Dequeue(Queue * pq)
{
    if(QIsEmpty(pq))
    {
        printf("Queue Memory Error!");
        exit(-1);
    }
    // front 이동 후 데이터 반환
    pq->front = NextPosIdx(pq->front);
    return pq->queArr[pq->front];
}
```



큐의 연결 리스트 기반 구현

연결 리스트 기반 큐의 헤더파일

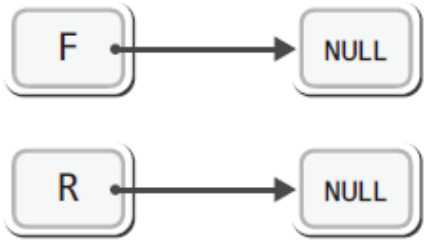
- 연결 리스트 기반 스택의 응용

```
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);
```

연결 리스트 기반 큐의 구현: 초기화

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



▶ [그림 07-12: 리스트 기반 큐의 초기상태]

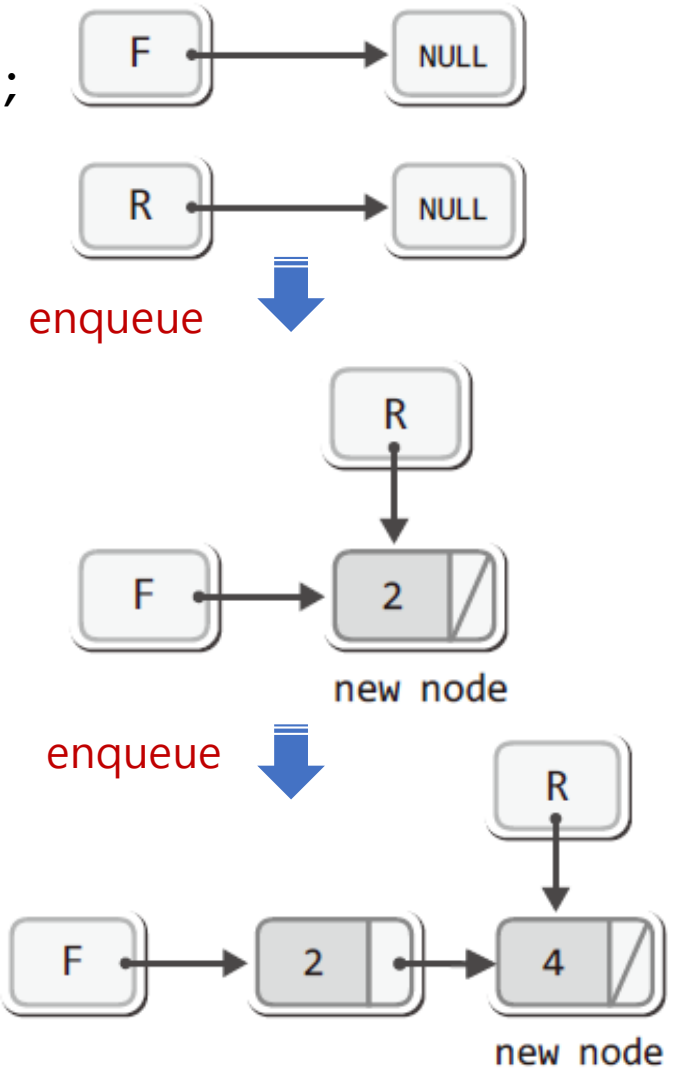
```
int. QIsEmpty(Queue * pq)
{
    if(pq->front == NULL)
        return TRUE;
    else
        return FALSE;
}
```

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;
    }
}
```

F와 R 변경

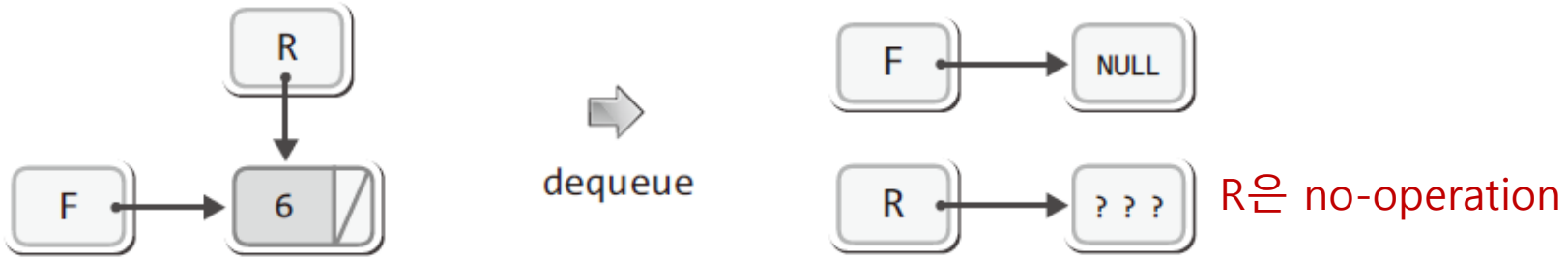
R만 변경



연결 리스트 기반 큐의 구현: dequeue

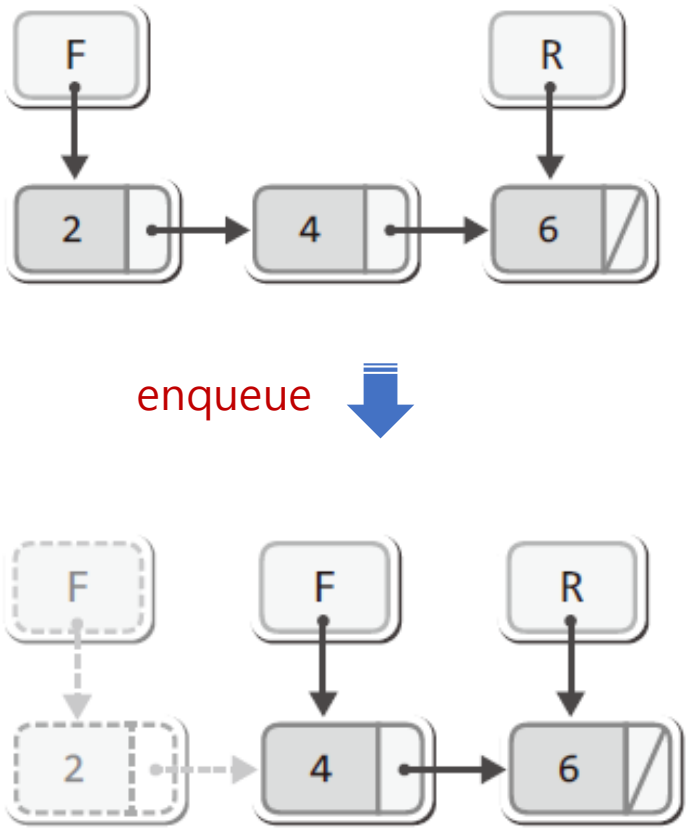
- F가 다음 노드를 가리킴, 이전 노드 삭제

- F가 다음 노드를 가리킴, 이전 노드 삭제



연결 리스트 기반 큐의 구현: dequeue 정의

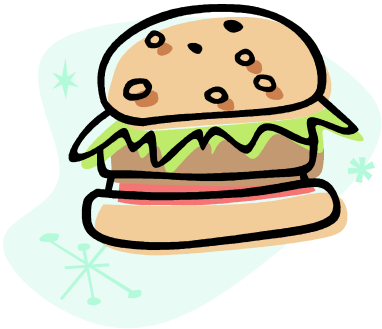
```
Data Dequeue(Queue * pq)
{
    Node * delNode;
    Data retData;

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

    delNode = pq->front;
    retData = delNode->data;
    // F가 다음 노드를 가리킴
    pq->front = pq->front->next;

    free(delNode); // 노드 삭제
    return retData;
}
```



큐의 활용

주제

- 점심시간 1시간 동안에 고객이 15초당 1명씩 주문
  - 종류별 햄버거를 만드는데 걸리는 시간
    - 치즈버거-12초
    - 불고기버거-15초
    - 더블버거-24초



- 시뮬레이션: 대기의 길이를 결정하는데 필요한 정보 확보

- 시뮬레이션을 통해서 추출된 정보의 형태!
  - 수용인원이 30명인 공간 안정적으로 고객을 수용할 확률 50%
  - 수용인원이 50명인 공간 안정적으로 고객을 수용할 확률 70%
  - 수용인원이 100명인 공간 안정적으로 고객을 수용할 확률 90%
  - 수용인원이 200명인 공간 안정적으로 고객을 수용할 확률 100%

시뮬레이션 예제

- 점심시간은 1시간이고 그 동안 고객은 15초에 1명씩 주문
- 한 명의 고객은 하나의 버거 만을 주문
- 주문하는 메뉴에는 가중치 없음
- 모든 고객은 무작위로 메뉴 선택
- 햄버거를 만드는 사람은 1명이다. 그리고 동시에 둘 이상의 버거가 만들지 않음
- 주문한 메뉴를 받을 다음 고객은 대기실에서 나와서 대기

CircularQueue.h

CircularQueue.c

HambugerSim.c

실행결과1

```
Simulation Report!
- Cheese burger: 80
- Bulgogi burger: 72
- Double burger: 88
- Waiting room size: 100
```

실행결과2

```
Queue Memory Error!
```

덱(Deque)의 이해와 구현

덱 (Deque, double ended queue)

- 덱은 양방향으로 enqueue와 dequeue가 되는 자료구조
  - 스택과 큐의 특성을 모두 갖음
  - 덱을 스택과 큐로 활용

- 덱의 4가지 연산
  - 앞으로 넣기
  - 앞에서 빼기
  - 뒤에서 빼기
  - 뒤에서 넣기

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.

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

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

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

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

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

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

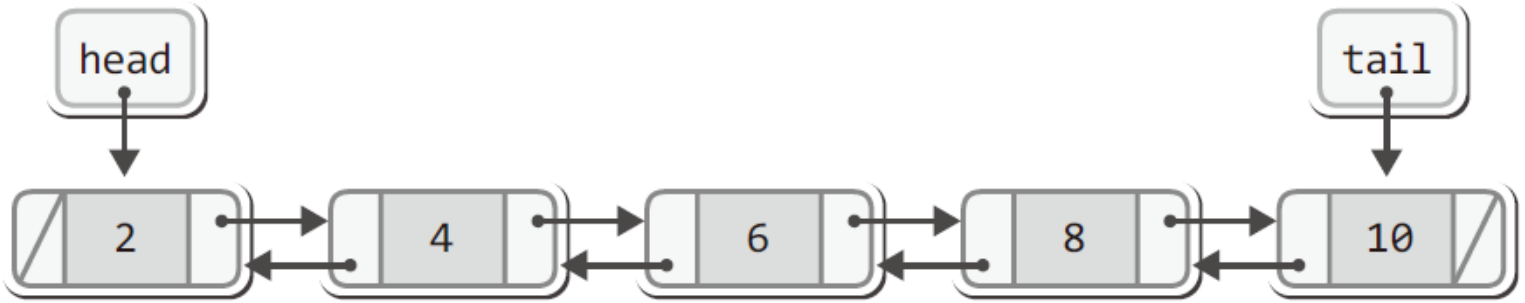
덱의 구현: 헤더파일 정의 Dequeu.h

```
typedef int Data;

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

typedef struct _dlDeque
{   // 앞 뒤 연산
    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);
```

덱의 구현: 함수의 정의

- 양방향 연결 리스트의 구조



- 이전에 구현한 양방향 연결 리스트의 구조