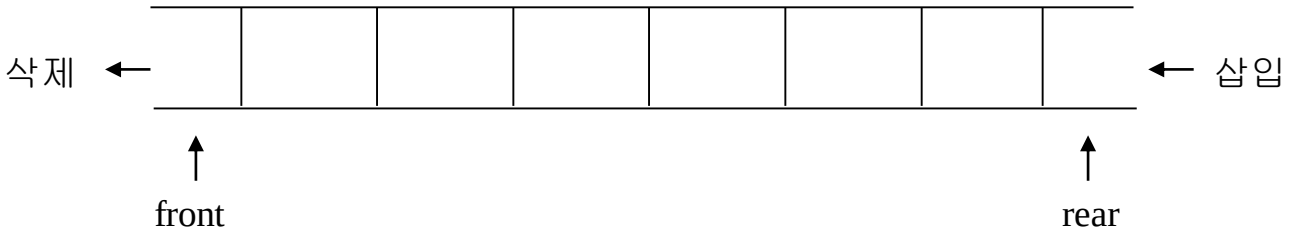
6장 큐

순서

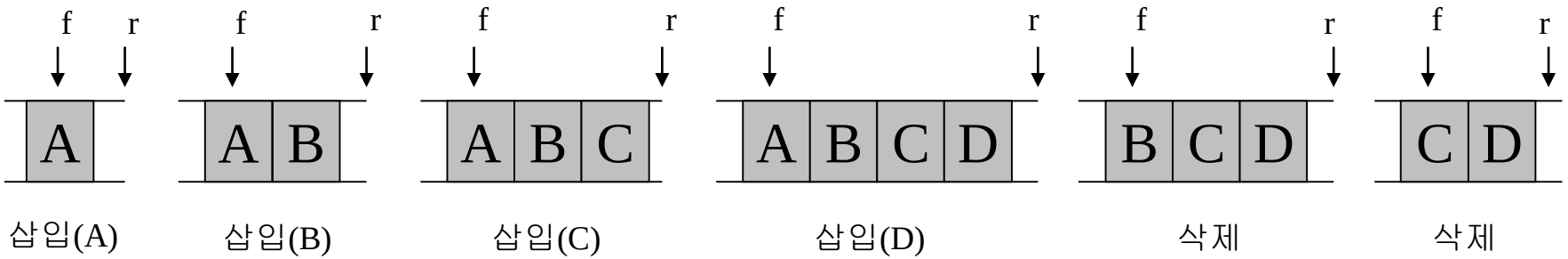
큐 추상 데이타 타입(1)

- **큐(queue)**
  - 한쪽 끝(rear)에서는 삽입(enqueue)만, 또 다른 끝(front)에서는 삭제(dequeue)만 하도록 제한되어 있는 유한 순서 리스트(finite ordered list)
  - 선입선출, First-In-First-Out(FIFO) 리스트
    - 큐에서 제일 먼저 삽입된 원소가 제일 먼저 삭제될 원소가 됨
  - 선착순 서버, first-come-first-serve (FCFS) 시스템
    - 서비스를 받기 위한 대기 행렬로 볼 수 있음
  - 큐의 작동 구조



큐 추상 데이타 타입(2)

- 큐에서의 삽입과 삭제
  - f : front, r : rear



- 큐의 응용 사례
  - 운영 체제 : 작업 큐를 통한 제출 순서에 따른 작업 스케줄
    - 서비스를 기다리는 작업들의 대기 상태를 나타내는 데 적절

큐 추상 데이타 타입(3)

- 큐 추상 데이타 타입(**ADT Queue**)

ADT Queue

데이타 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :

Q ∈ Queue; item ∈ Element;
createQ() ::= create an empty queue;
enqueue(Q, item) ::= insert item at the rear of Q;
isEmpty(Q) ::= if Q is empty then return true
else return false;
dequeue(Q) ::= if isEmpty(Q) then return null
else remove and return the front element of Q;
remove(Q) ::= if isEmpty(Q) then return null
else remove the front element of Q;
peek(Q) ::= if isEmpty(Q) then return null
else return the front element of Q;

End Queue

큐의 순차 표현(1)

- **1차원 배열**
  - 큐를 표현하는 가장 간단한 방법
  - Q[n]을 이용한 순차 표현
  - 순차 표현을 위한 변수
    - n : 큐에 저장할 수 있는 최대 원소 수
    - 두 인덱스 변수 front, rear
      - 초기화 : front = rear = -1 (공백큐)
      - 공백큐 : front = rear
      - 만원 : rear = n-1

큐의 순차 표현(2)

- 큐의 연산자(1)

```
createQ()
    Q[n];
    front ← -1;     // 초기화
    rear ← -1;
end createQ()

isEmpty(Q)
    return (front = rear);
end isEmpty()

// 이 is Empty 연산자는 아주 간단해서 직접 구현하여 사용할 수도 있다.

enqueue(Q, item)
    // Q에 원소 삽입
    if (rear=n-1) then queueFull();     // Q가 만원이면 큐를 확장
    else rear ← rear + 1;
    Q[rear] ← item;
end enqueue()
```

큐의 순차 표현(3)

- 큐의 연산자(2)

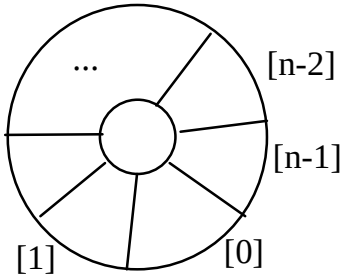
```
dequeue(Q)
    // Q에서 원소를 삭제하고 반환
    if (isEmpty(Q)) then return null;    // Q가 공백일 경우
    front ← front + 1;
    return Q[front];
end dequeue()

remove(Q)
    // Q에서 원소를 삭제
    if (isEmpty(Q)) then return null;
    else return Q[front];
end remove()

peek(Q)
    // Q에서 원소를 검색
    if (isEmpty(Q)) then return null;    // Q가 공백일 경우
    else return Q[front + 1];
end peek()
```

큐의 순차 표현(4)

- **순차 표현의 문제점**
  - Rear=n-1인 경우
    - 만원이지만, 반드시 n개의 원소가 큐에 있지는 않음
      - 큐의 앞에서 삭제로 인해 비 공간이 생길 수 있음
      - 빈공간을 없애기 위해 앞쪽으로 이동, front·rear 재설정
      - → 시간, 연산의 지연 문제
    - 실제로 큐가 만원인 경우
      - 배열의 크기를 확장해야 함
- **원형 큐(circular queue)**
  - 순차 표현의 문제점 해결 위해 배열 Q[n]을 원형으로 운영
  - 원형 큐의 구현
    - 초기화 : front = rear = 0 (공백큐)
    - 공백큐 : front = rear
    - 원소 삽입 : rear를 하나 증가시키고, 그 위치에 원소 저장
    - 만원 : rear를 하나 증가시켰을 때, rear = front
    - (실제 front 공간 하나가 비지만, 편의를 위해 그 공간을 희생)

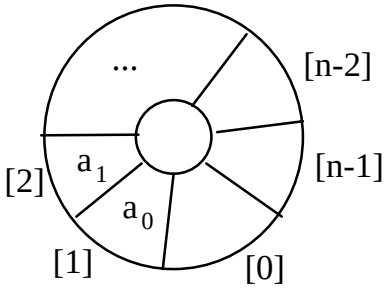
큐의 순차 표현(5)

- 원형 큐의 여러 상태



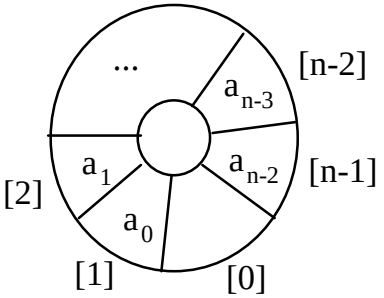
front = [0]
rear = [0]

(a) 공백 원형 큐



front = [0]
rear = [2]

(b) 2개의 원소 저장



front = [0]
rear = [n-1]

(c) 만원 원형 큐

- 1차원 배열을 원형으로 유지하는 방법
  - mod(modulus) 연산자 이용
    - 삽입을 위해 먼저 rear를 증가시킬 때 : rear ← (rear+1) mod n
      - rear 값은 n-1 다음에 n이 되지 않고, 다시 0으로 되돌아감
    - 삭제를 위해 front를 증가시킬 때 : front ← (front+1) mod n
      - rear와 마찬가지로, front 값은 n-1 다음에 0이 되어 원형으로 순환

큐의 순차 표현(6)

- 원형 큐에서의 **enqueue**와 **dequeue** 연산

```
enqueue(Q, item)
    // 원형 큐(Q)에 item을 삽입
    rear ← (rear+1) mod n;        // 원형 인덱스
    if (front=rear) then queueFull();    // 큐를 확장
    Q[rear] ← item;
end enqueue()

dequeue(Q)
    // 원형 큐(Q)에서 원소를 삭제하고 그것을 반환
    if (front=rear) then return null;
    front ← (front+1) mod n;    // 원형 인덱스
      return Q[front];
end dequeue()
```

Java 배열을 이용한 큐의 구현(1)

- **큐 ADT의 구현 방법**
  - Java에서 지원하는 interface : 메소드에 대한 선언만 함
  - 실제 구현 : 메소드를 사용하는 클래스에 위임

- **Queue interface 정의**

```
public interface Queue  {
    boolean isEmpty();     // 큐가 공백인가를 검사
    void enqueue(Object x);     // 원소 x를 삽입
    Object dequeue();     // 원소를 삭제하고 반환
    void remove();     // 원소를 삭제
    Object peek();     // 원소값만 반환
}
```

Java 배열을 이용한 큐의 구현(2)

- 배열을 이용하여 원형 큐를 구현하는 **ArrayQueue** 클래스 정의(1)

```
// Java 배열을 이용한 원형 큐의 구현
public class ArrayQueue implements Queue  {
    private int front;                  // 큐의 삭제 장소
    private int rear;                   // 큐의 삽입 장소
    private int count;    // 큐의 원소 수
    private int queueSize;              // 큐(배열)의 크기
    private int increment;              // 배열의 확장 단위
    private Object[] itemArray;     // Java 객체타입의 큐 원소를 위한 배열

    public ArrayQueue()  {          // 무인자 큐 생성자
        front = 0;                      // 초기화
        rear = 0;
        count = 0;
        queueSize = 50;                 // 초기 큐 크기
        increment = 10;                 // 배열의 확장 단위
        itemArray = new Object[queueSize];
    }

    public boolean isEmpty()  {
        return (count == 0);
    }
```

Java 배열을 이용한 큐의 구현(3)

- 배열을 이용하여 원형 큐를 구현하는 **ArrayQueue** 클래스 정의(2)

```
public void enqueue(Object x)  {
    if (count == queueSize) queueFull();
    itemArray[rear] = x;     // 새로운 원소를 삽입
    rear = (rear + 1) % queueSize;
    count++;
}    // end queue( )

public void queueFull()  {     // 배열이 만원이면 increment만큼 확장
    int oldsize = queueSize;       // 현재의 배열 크기를 기록
    queueSize += increment;     // 새로운 배열 크기
    Object[] tempArray = new Object[queueSize];     // 확장된 크기의 임시 배열
    for (int i=0;  i<count;  i++)  {     // 임시 배열로 원소들을 그대로 이동
        tempArray[i] = itemArray[front];
        front = (front + 1) % oldsize
    }
    itemArray = tempArray;     // 배열 참조 변수를 변경
    front = 0;
    rear = count;
}    // end queueFull()
```

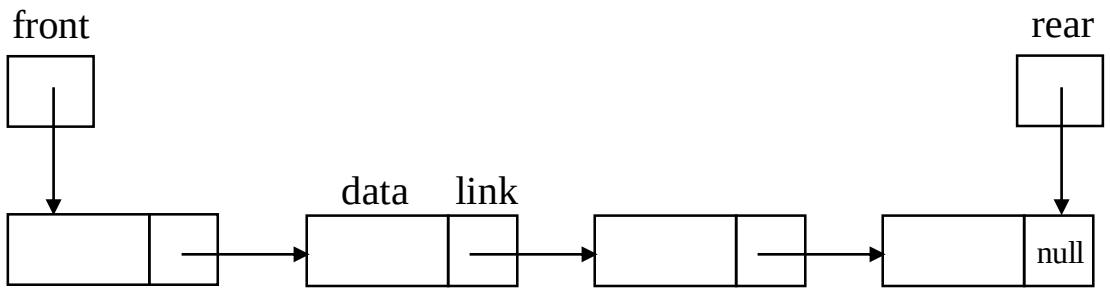
Java 배열을 이용한 큐의 구현(4)

- 배열을 이용하여 원형 큐를 구현하는 **ArrayQueue** 클래스 정의(3)

```
    public Object dequeue( ) {    // 큐에서 원소를 삭제해서 반환
        if (isEmpty()) return null;    // 큐가 공백일 경우

        // 큐가 공백이 아닌 경우
        Object item = itemArray[front];
        front = (front + 1) % queueSize;
        count --;
        return item;
    }    // end dequeue()

    public Object peek( ) {    // 큐에서 원소값을 반환
        if (isEmpty()) return null;
        else return itemArray[front];
    } //    end peek()

} //    end ArrayQueue class
```

큐의 연결 표현(1)

- **연결리스트로 표현된 큐**
  - 여러 개의 큐를 동시에 필요로 하는 경우에 효율적
  - 연결 큐(linked queue)의 구조
    - 단순 연결 리스트를 두 개의 포인터 변수 front, rear로 관리
    - 초기화 : front = rear = null (공백큐)
    - 큐의 공백 여부 : front 또는 rear가 null인지 검사를 통해 알 수 있음



큐의 연결 표현(2)

- 연결 큐에서의 삽입, 삭제, 검색 연산 구현(1)

```
enqueue(Q, item)    // Q에 item을 삽입
    newNode ← getNode();    // 새로운 노드를 생성
    newNode.data ← item;
    newNode.link ← null;
    if (rear = null) then  {    // Q가 공백인 경우
        rear ← newNode;
        front ← newNode;
    }
    else  {
        rear.link ← newNode;
        rear ← newNode;
    }
end Enqueue

dequeue(Q)    // 큐에서 원소를 삭제하고 값을 반환
    if (front = null) then return null;    // Q가 공백인 경우
    oldNode ← front;
    item ← front.data;
    front ← front.link;
    if (front = null) then rear ← null;    // 삭제로 인해 Q가 공백이 되는 이유
    retNode(oldNode);
    return item;
end dequeue
```

큐의 연결 표현(3)

- 연결 큐에서의 삽입, 삭제, 검색 연산 구현(2)

```
remove(Q)     // Q에서 원소를 삭제
    if (front = null) then return null;
    oldNode ← front;
    front ← front.link;
    if (front = null) then rear ← null;
    retNode(oldNode);
end remove()

peek(Q)     // Q의 front 원소를 검색
    if (front = null) then return null;     // Q가 공백인 경우
    else return (front.data);
end peek()
```

큐의 연결 표현(4)

- **연결큐의 특징**
  - 삽입, 삭제로 인한 다른 원소들의 이동이 필요 없음
  - 연산이 신속하게 수행
  - 여러 개의 큐 운영시에도 연산이 간단
  - 링크 필드에 할당하는 추가적인 저장 공간 필요

- **k개의 큐를 사용**
  - 큐0, 큐1, 큐2, …, 큐k-1
  - 각 큐에 대한 front와 rear : 배열 front[i], rear[i]를 사용
  - 초기화
    - front[i] ← null; $0 \le i < k$
    - rear[i] ← null; $0 \le i < k$

큐의 연결 표현(5)

- **k개의 큐에서의 삽입, 삭제, 검색 연산 구현(1)**

```
enqueue(i, item)    // 큐 i에 item을 삽입
    newNode ← getNode();
    newNode.data ← item;
    newNode.link ← null;
    if (front[i] = null) then {    // 큐 i가 공백일 경우
        front[i] ← newNode;
        rear[i] ← newNode;
    }
    else {    // 큐 i가 공백이 아닌 경우
        rear[i].link ← newNode;
        rear[i] ← newNode;
    }
end enqueue()

dequeue(i)    // 큐 i에서 원소를 삭제하고 값을 반환
    if (front[i] = null) then return null;    // 큐 i가 공백일 경우
    else {
        oldNode ← front[i];
        item ← front[i].data;
        front[i] ← front[i].link;
        if (front[i] = null) then rear[i] ← null;    // 큐 i가 공백이 되는 경우
    }
    retNode(oldNode);    // 노드를 가용공간 리스트에 반환
    return item;
end dequeue()
```

큐의 연결 표현(6)

- **k**개의 큐에서의 삽입, 삭제, 검색 연산 구현(2)

```
remove(i)    // 큐 i에서 원소를 삭제
    if (front[i] = null) then return null;
    else  {
        oldNode ← front[i];
        item ← front[i].link;
        front[i] ← front[i].link;
        if (front[i] = null) then rear[i] ← null;
    }
    retNode(oldNode);
    return item;
end remove()

peek(i)    // 큐 i에서 원소 값을 검색
    if (front[i] = null) then return null;
    else return (front[i].data);
end peek()
```

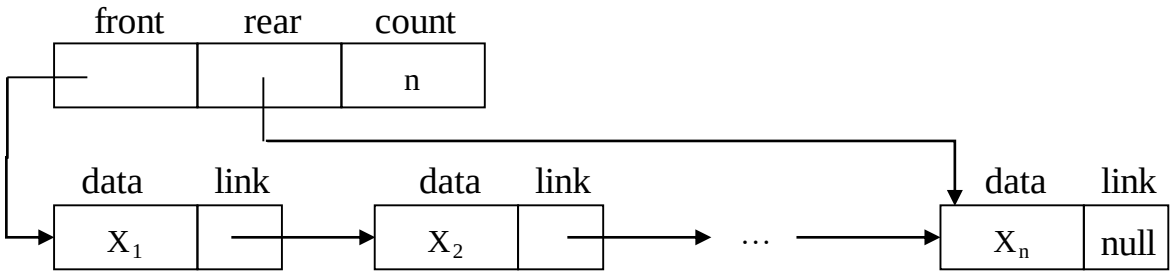
Java 리스트를 이용한 큐의 구현(1)

- 먼저 노드의 구조를 결정 : **ListNode** 클래스

```
public class ListNode {
    Object data;
    ListNode Link;
}
```

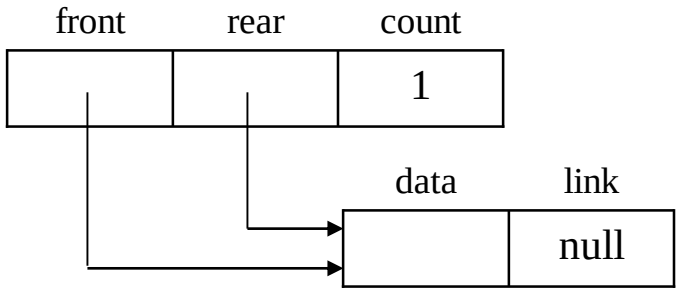
Java 리스트를 이용한 큐의 구현(2)

- 큐를 구현한 연결 리스트 표현



공백이 아닌 큐를 구현한 연결 리스트 표현

공백 큐를 구현한 연결 리스트 표현



하나의 노드를 가진 큐의 연결 리스트 표현

Java 리스트를 이용한 큐의 구현(3)

- 연결 리스트로 큐를 구현한 **ListQueue** 클래스(1)

```
public class ListQueue implements Queue  {
    private ListNode front;      // 큐에서의 front 원소
    private ListNode rear;       // 큐에서의 rear 원소
    private int count;           // 큐의 원소수

    public ListQueue()  {     // 공백 큐
        front = null;
        rear = null;
        count = 0;
    }
```

Java 리스트를 이용한 큐의 구현(4)

- 연결 리스트로 큐를 구현한 **ListQueue** 클래스(2)

```
public void enqueue(Object x) {     // 큐에 원소 x를 삽입
    ListNode newNode = new ListNode();
    newNode.data = x;
    newNode.link = null;
    if (count==0)  {     // 큐(리스트)가 공백인 경우
        front = rear = newNode;
    } else {
        rear.link = newNode;
        rear = newNode;
    }
    count++;
}     // end enqueue()

public Object dequeue()  {     // 큐에서 원소를 삭제하고 반환
    if (count==0) return null;
    Object item = front.data;
    front = front.link;
    if (front == null)  {     // 리스트의 노드를 삭제 후 공백이 된 경우
        rear = null;
    }
    count-- ;
    return item;
}     // end dequeue()
```

Java 리스트를 이용한 큐의 구현(5)

- 연결 리스트로 큐를 구현한 **ListQueue** 클래스(3)

```
    public void remove()  {     // 큐에서 원소를 삭제
        if (count==0) return null;
        else front = front.link;
    }     // end remove()

    public Object peek() {
        if (count==0) return null;
        else return front.data;
    }     // end peek()

}     // end ListQueue class
```

덱(1)

- **덱(deque : double-ended queue)**
  - 스택과 큐의 성질을 종합한 순서 리스트
  - 삽입과 삭제가 리스트의 양끝에서 임의로 수행될 수 있는 자료구조
  - 스택이나 큐 ADT이 지원하는 연산을 모두 지원

덱(2)

- 덱의 추상 데이타 타입(ADT)

```
createDeque() ::= create an empty deque;
insertFirst(Deque,e) ::= insert new element e at the beginning of Deque;
insertLast(Deque,e) ::= insert new element e at the end of Deque;
isEmpty(Deque) ::= if Deque is empty then return true
                   else return false;
deleteFirst(Deque) ::= if isEmpty(Deque) then return null
                       else remove and return the first element of Deque;
deleteLast(Deque) ::= if isEmpty(Deque) then return null
                      else remove and return the last element of Deque;
removeFirst(Deque) ::= if isEmpty(Deque) then return null
                       else remove the first element of Deque;
removeLast(Deque) ::= if isEmpty(Deque) then return null
                      else remove the last element of Deque;
peekLast(Deque) ::= return the last element of Deque;
peekFirst(Deque) ::= return the first element of Deque;
```

덱(3)

- 공백 덱에 대한 일련의 연산 수행

연산	덱(Deque)
insertFirst(Deque,3)	(3)
insertFirst(Deque,5)	(5, 3)
deleteFirst(Deque)	(3)
insertLast(Deque,7)	(3, 7)
deleteFirst(Deque)	(7)
deleteLast(Deque)	()
insertFirst(Deque,9)	(9)
insertLast(Deque,7)	(9, 7)
insertFirst(Deque,3)	(3, 9, 7)
insertLast(Deque,5)	(3, 9, 7, 5)
deleteLast(Deque)	(3, 9, 7)
deleteFirst(Deque)	(9, 7)

덱(4)

- 스택과 큐 ADT 연산에 대응하는 덱의 연산
  - 스택 ADT 연산에 대응하는 연산

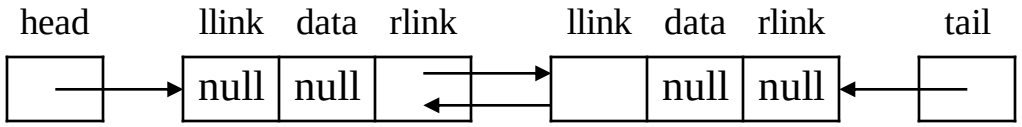
스택 연산	덱 연산
createStack()	createDeque()
push(S,e)	insertLast(Deque,e)
isEmpty(S)	isEmpty(Deque)
pop(S)	deleteLast(Deque)
remove(S)	removeLast(Deque)
peek(S)	peekLast(Deque)

  - 큐 ADT 연산에 대응하는 연산

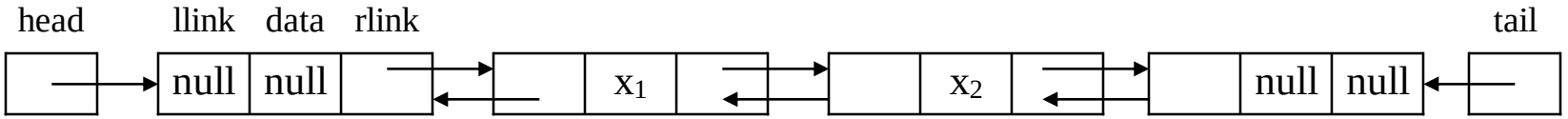
큐 연산	덱 연산
createQ()	createDeque()
enqueue(Q,e)	insertLast(Deque,e)
isEmpty(Q)	isEmpty(Deque)
dequeue(Q)	deleteFirst(Deque)
remove(Q)	removeFirst(Deque)
peek(Q)	peekFirst(Deque)

덱(5)

- 덱의 구현
  - 단순 연결 리스트
    - 이점 : 리스트의 마지막 노드를 가리키는 포인터를 이용
    - 단점 : 리스트 마지막 노드의 삭제를 상수 시간에 수행할 수 없음
  - 이중 연결 리스트
    - 이점 : 리스트 양쪽 끝에서 삽입과 삭제가 상수 시간에 수행
    - head 노드, tail 노드 사용
      - 리스트의 첫 번째 노드와 마지막 노드만을 가리키는 공백 노드
      - 다른 데이타를 저장할 목적이 아님



공백 이중 연결 리스트



2개의 원소를 포함한 이중 연결 리스트

덱(6)

- 이중 연결 리스트를 이용한 **Deque** 구현(1)

```
public class DoubleListNode  {
    Object data;
    DoubleListNode rlink;
    DoubleListNode llink;
}

public class Deque  {     // 이중 연결 리스트로 구현
    private DoubleListNode head, tail;
    private int count;

    public Deque()  {
        head = new DoubleListNode();
        tail = new DoubleListNode();

        head.llink = null;
        head.data = null;
        head.rlink = tail;

        tail.rlink = null;
        tail.data = null;
        tail.llink = head;
    }
```

덱(7)

- 이중 연결 리스트를 이용한 **Deque** 구현(2)

```
public void insertFirst(Object value)  {
    DoubleListNode newNode = new DoubleListNode();
    newNode.data = value;
    DoubleListNode second = head.rlink;
    second.llink = newNode;
    newNode.rlink = second;
    newNode.llink = head;
    head.rlink = newNode;
    count++;
}

public void insertLast(Object value)  {
    DoubleListNode newNode = new DoubleListNode();
    newNode.data = value;
    DoubleListNode secondtolast = tail.llink;
    newNode.llink = secondtolast;
    secondtolast.rlink = newNode;
    newNode.rlink = tail;
    count++;
}
```

덱(8)

- 이중 연결 리스트를 이용한 **Deque** 구현(3)

```
public Object deleteFirst()  {
    if (isEmpty()) return null;
    DoubleListNode first = head.rlink;
    Object value = first.data;
    DoubleListNode second = first.rlink;
    second.llink = head;
    head.rlink = second;
    count--;
    return value;
}

public Object deleteLast()  {
    if (isEmpty()) return null;
    DoubleListNode last = tail.llink;
    DoubleListNode secondtolast = last.llink;
    Object value = last.data;
    secondtolast.rlink = tail;
    tail.llink = secondtolast;
    count--;
    return value;
}
```

덱(9)

- 이중 연결 리스트를 이용한 **Deque** 구현(4)

```
    public void removeFirst()  {
                    :
    }
    public void removeLast()  {
                    :
    }
    public Object peekFirst()  {
                    :
    }
                    :
    public Object peekLast()  {
                    :
    }
}    // end Deque class
```

덱(10)

- 덱을 이용한 Stack 구현(1)
  - Depue 클래스를 이용해 스택을 구현한 DequeStack 클래스

```
public class DequeStack implements Stack {     // 스택을 덱을 이용해 구현
    private Deque d;     // Deque 타입의 참조 변수

    public DequeStack() {     // 생성자, 스택을 초기화
        d = new Deque();
    }

    public boolean isEmpty() {     // 스택이 공백인가를 검사
        return d.isEmpty();
    }

    public void push(Object x) {     // 스택에 원소 삽입
        d.insertLast(x);
    }
```

덱(11)

- 덱을 이용한 **Stack** 구현(2)

```
    public Object peek()  {     // 스택의 톱 원소를 검색
        if (isEmpty()) return null;     // 스택이 공백인 경우
        else return d.Last();
    }

    public Object pop()  {     // 스택의 톱 원소를 삭제하고 반환
        if (isEmpty()) return null;     // 스택이 공백인 경우
        else return d.deleteLast();
    }

    public void remove()  {     // 스택의 톱 원소를 삭제
        if (isEmpty()) return;
        else d.deleteLast();
    }

}     // end DequeStack class
```