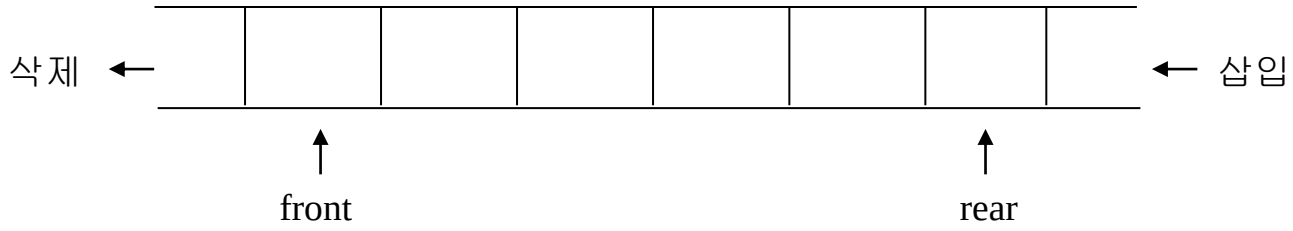
6장 Queue

순서

6.1 큐 추상 데이터 타입

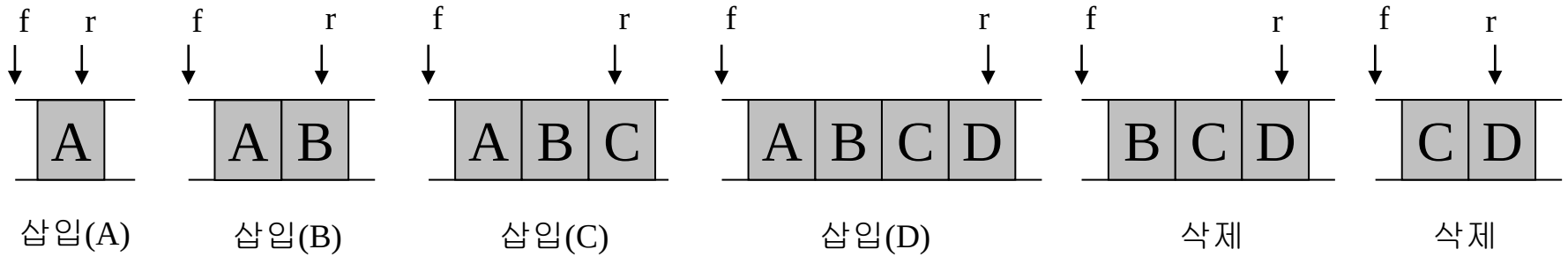
- 한쪽 끝, rear에서는 삽입(enqueue)만, 또 다른 끝, front에서는 삭제(dequeue)만 하도록 제한되어 있는 유한 순서 리스트(finite ordered list)
- 선입선출(First-In-First-Out: FIFO) 리스트
  - 제일 먼저 삽입된 원소가 제일 먼저 삭제될 원소가 됨
- 선착순 서버(first-come-first-serve: FCFS) 시스템
  - 서비스를 받기 위한 대기행렬로 볼 수 있음
  - "Queue here"
- Queue의 작동 구조



Queue 추상 데이타 타입(2)

- Queue에서의 삽입과 삭제
  - 포인터

    f: front(삭제), r: rear(삽입)



- Queue의 응용 사례
  - 운영 체제 : 작업 큐를 통한 제출 순서에 따른

    작업 스케줄(job schedule)
    - 서비스를 기다리는 작업들의 대기 상태를 나타내는 데 적합

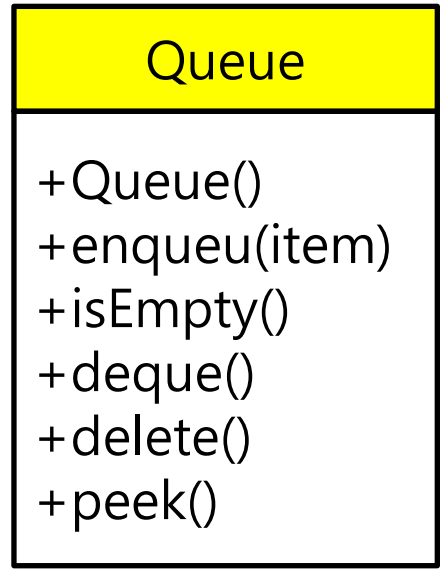
Queue 추상 데이타 타입(3)

- Queue 추상 데이타 타입(ADT Queue)

ADT Queue

데이타 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :
queue ∈ Queue; item ∈ Element;
createQ() ::= create an empty queue;
enqueue(queue, item) ::= insert item at the rear of queue;
isEmpty(queue) ::= **if** (queue is empty) **then return** true
else return false;
dequeue(queue) ::= **if** (isEmpty(queue)) **then return** error
else {delete and return the front item of queue};
delete(queue) ::= **if** (isEmpty(queue)) **then return** error
else {delete the front item of queue};
peek(queue) ::= **if** (isEmpty(queue)) **then return** error
else {**return** the front item of queue};
End Queue



6.2 Queue의 순차 표현

- 1차원 배열(array)
  - Queue를 표현하는 가장 간단한 방법
  - 배열 q[n]을 이용한 순차 표현
    - 인덱스는 0에서부터 시작
  - 순차 표현을 위한 변수
    - n : 큐에 저장할 수 있는 최대 원소 수, 큐의 크기
    - 두 인덱스 변수 front, rear
      - 초기화 : front = rear = -1 (공백 큐)
      - 공백 큐(empty queue) : front = rear
      - 만원(queueFull) : rear = n-1

Queue의 연산자(2)

```
// 공백 큐(q[])를 생성
createQ()
   q[n];
   front ← -1;   // 초기화
   rear ← -1;
end createQ()


 // 큐(q)가 공백인지를 검사
isEmpty(q)
   if (front = rear) then return true
   else return false;
end isEmpty()


// 큐(q)에 원소를 삽입
enqueue(q, item)
   if (rear = n-1) then queueFull()
   rear ← rear + 1;
   q[rear] ← item;
end enqueue()
```

```
public class Queue {
   private Object[] q;
   private int front, rear;
   private int n = 10;

   public Queue() {
      q = new Object[n];
      front = rear = -1;
   }



   public boolean isEmpty() {
      return (front == rear);
   }



   public void enqueu(Object item) {
      if (rear == n - 1)
         throw new QueueFullException();
      rear += 1;
      q[rear] = item;
   }
```

Queue의 연산자(2)

```
dequeue(q)
  // 큐(q)에서 원소를 삭제하여 반환
  if (isEmpty(q)) then queueEmpty()
     // 큐(q)가 공백인 상태를 처리
  else {
     front ← front + 1;
     return q[front];
  };
end dequeue()

delete(q)
  // 큐(q)에서 원소를 삭제
  if (isEmpty(q)) then queueEmpty()
     // 큐(q)가 공백인 상태를 처리
  front ← front + 1;
end delete()

peek(q)
  // 큐(q)에서 원소를 검색
  if (isEmpty(q)) then queueEmpty()
     // 큐(q)가 공백인 상태를 처리
  else return q[front+1];
end peek()
```

```
    public Object deque() {
       if (isEmpty())
          throw new QueueEmptyException();
       front += 1;
       return q[front];
    }


    public void delete() {
       if (isEmpty())
          throw new QueueEmptyException();
       front += 1;
    }


    public Object peek() {
       if (isEmpty())
          throw new QueueEmptyException();
       return q[front + 1];
    }

    public class QueueFullException
                extends RuntimeException {
    }

    public class QueueEmptyException
                extends RuntimeException {
    }
}
```

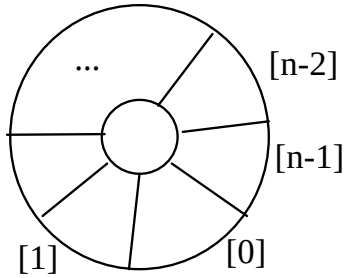
Queue의 순차 표현의 문제점

- 순차 표현의 문제점
  - rear = n-1인 경우
    - 만원이지만, 반드시 n개의 원소가 큐에 있지는 않음
      - 큐의 앞에 삭제로 인한 빈 공간이 있을 수 있음
      - 빈공간을 활용하기 위해 front와 rear를 재설정 필요. 시간, 연산의 지연 문제
    - 실제로 큐가 만원인 경우
      - 배열의 크기를 확장해야 됨

- 원형 큐(circular queue)
  - 순차 표현의 문제를 해결하기 위해 배열 q[n]을 원형으로 운영
  - 원형 큐의 구현
    - 초기화: front = rear = 0 (공백 큐)
    - 공백 큐: front = rear
    - 원소 삽입: 먼저 rear를 하나 증가시키고, 그 위치에 원소 저장
    - 만원 큐: rear를 하나 증가시켰을 때, rear = front
      (이때 실제로 front의 공간 하나가 공백으로 있지만,
      구현의 편의를 위해 이 공간을 희생)

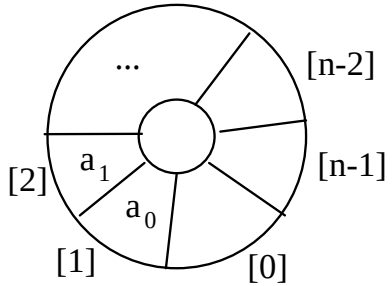
Queue의 순차 표현(5)

- 원형 큐의 여러 상태



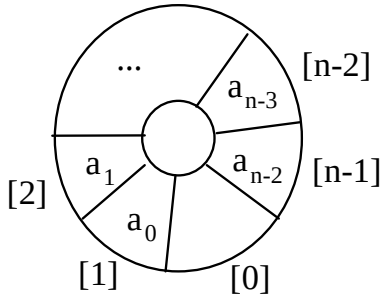
front = [0]
rear = [0]

(a) 공백 원형 큐



front = [0]
rear = [2]

(b) 2개의 원소 저장



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

(c) 만원 원형 큐

- 1차원 배열을 원형으로 구현하는 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 됨
    - 삭제를 위해 front를 증가시킬 때 : front ← (front+1) *mod* n
      - rear와 마찬가지로, front 값은 n-1 다음에 0이 되어 원형으로 순환

Queue의 순차 표현(6)

- 원형 큐에서의 enqueue와 dequeue 연산

```
public class CircularQueue {
    private static final int SIZE = 10;
    private Object[] q = new Object[SIZE];
    private int rear = 0;
    private int front = 0;

    // 원형 큐에 item을 삽입
    public void enqueu(Object item) {
        rear = (rear + 1) % SIZE;
        if (front == rear)
            throw new QueueFull();
        q[rear] = item;
    }

    // 원형 큐에서 원소를 삭제하고 반환
    public Object dequeue() {
        if (front == rear)
            throw new QueueEmpty();
        front = (front + 1) % SIZE;
        return q[front];
    }
}
```

```
public class QueueEmpty extends RuntimeException {
}

public class QueueFull extends RuntimeException {
}
```

6.3 배열을 이용한 Queue의 구현(1)

- Queue ADT의 구현 방법
  - Java에서 지원하는 interface : 메소드에 대한 선언만 함
  - 실제 구현은 이 메소드들을 사용하는 클래스에 위임

- Queue interface 정의

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

배열을 이용한 Queue의 구현(2)

```
/**
 * 배열을 이용한 Queue interface의 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 객체 타입의 큐 원소를 위한 배열

    public ArrayQueue() {
        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);
    }
```

배열을 이용한 Queue의 구현(3)

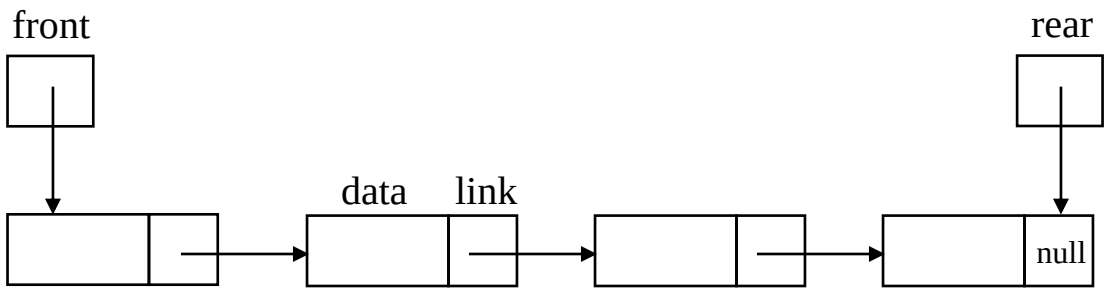
```
/**
 * 큐에 원소 x를 삽입
 */
public void enqueue(Object x) {
    if (count == queueSize)
        queueFull();

    itemArray[rear] = x;            // 원소를 삽입
    rear = (rear + 1) % queueSize;
    count++;
}

/**
 * 배열이 만원이면 increment만큼 확장
 */
public void queueFull() {
    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 이름을 변경
    front = 0;
    rear  = count;
}
```

배열을 이용한 Queue의 구현(4)

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

        Object item = itemArray[front];
        front = (front + 1) % queueSize;
        count --;
        return item;
    }

    /** 큐에서 원소를 삭제 */
    public Object delete( ){
        if (isEmpty()) return null;

        front = (front + 1) % queueSize;
        count --;
    }

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

}
```

6.4 Queue의 연결 표현(1)

- 연결 리스트(linked list)로 표현된 Queue
  - 여러 개의 큐를 동시에 필요로 하는 경우에 효율적
  - 연결 큐(linked queue)의 구조
    - 단순 연결 리스트를 두 개의 포인터 front, rear로 관리
    - 초기화: front = rear = null (공백 큐)
    - 큐의 공백 여부: front 또는 rear가 null인지 검사해서 알 수 있음



Queue의 연결 표현(2)

- 연결 큐의 특징
  - 삽입, 삭제로 인한 다른 원소들의 이동이 필요 없음
  - 삽입, 삭제 연산이 신속하게 수행
  - 여러 개의 큐 운영 시에도 연산이 간단

- m개의 큐 구현
  - 여러 개의 큐 객체 이용

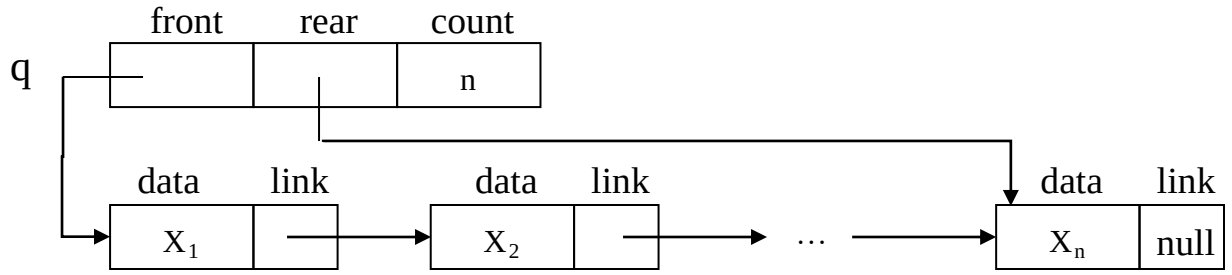
6.5 리스트를 이용한 Queue의 구현(1)

- 리스트 노드의 구조: ListNode class

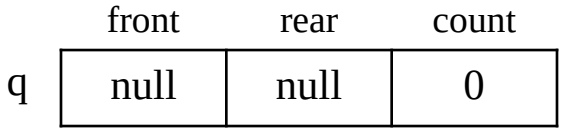
```
public class ListNode {
    Object  data;
    ListNode  Link;
}
```

리스트를 이용한 Queue 의 구현(2)

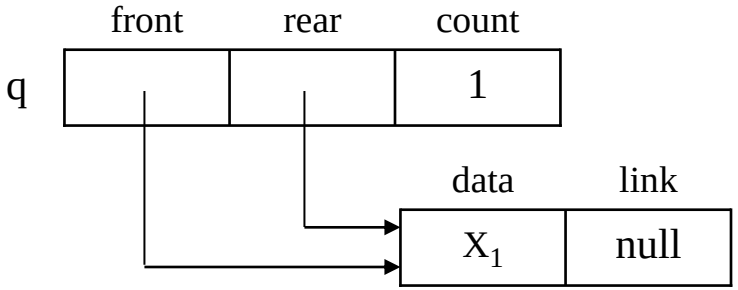
- 연결 리스트 구현된 Queue



공백이 아닌 큐를 구현한 연결 리스트



공백 큐를 구현한 연결 리스트



하나의 노드를 가진 큐의 연결 리스트

연결 리스트로 구현한 Queue (1)

```
/**
 * 연결 리스트를 이용한 Queue interface의 구현
 */
public class ListQueue implements Queue {

    private ListNode front;         // 큐에서의 front 원소
    private ListNode rear;          // 큐에서의 rear 원소
    private int count;              // 큐의 원소 수

    /** 공백 큐를 생성 */
    public ListQueue() {
        front = null;
        rear  = null;
        count = 0;
    }

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

연결 리스트로 구현한 Queue (2)

```
/** 큐에 원소 x를 삽입 */
public void enqueue(Object x) {
    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++;
}

/** 큐에서 원소를 삭제하고 반환 */
public Object dequeue() {
    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;
  }
```

연결 리스트로 구현한 Queue (3)

```
    /** 큐에서 원소를 삭제 */
    public void delete() {
        if (count == 0)
            return null;

        front = front.link;
        if (front == null) {   // 리스트의 노드를 삭제 후 공백이 된 경우
            rear = null;
        }
        count-- ;
    }

    public Object peek() {
        return (count == 0) ? null : front.data;
    }

}
```

6.6 Queue의 응용 - 컴퓨터 운영 체제

- 운영 체제에서 큐의 응용
  - 상이한 속도로 실행하는 두 프로세스 간의 상호작용을 조화시키는 버퍼 역할을 담당
    - 예: CPU와 프린터 사이의 프린트 버퍼(printBufferQueue)
    - consumer/producer problem

```
/** 프린트해할 라인을 CPU가 프린트 버퍼 큐에 삽입(생산) */
writeLine()
    if (there is a line L to print) and (printBufferQueue ≠ full)
                                    and (printBufferQueue ≠ busy)
    then enqueue(printBufferQueue, L);
end writeLine()

/** 프린터가 프린트 버퍼 큐의 라인들을 프린트(소비) */
readLine()
    if (printBufferQueue ≠ empty) and (printBufferQueue ≠ busy)
    then {
        L ← dequeue(printBufferQueue);
        print L;
    }
end readLine()
```

컴퓨터 시뮬레이션(1)

- 컴퓨터 시뮬레이션(simulation)의 정의
  - 어떤 물리적 시스템의 행태(behavior)를 분석하고 예측하기 위해 컴퓨터 모델을 통해 시뮬레이트하는 것
  - 물리적 시스템
    - 특정 목적을 달성하기 위해 동작하고 상호작용하는 독립적인 원소나 개체의 집합

- 물리적 시스템의 상태
  - 상태 변수(state variable)들을 사용하여 표현
    - 공항의 항공기 교통 시뮬레이션 예
      - 개체: 항공기
      - 상태: 항공기의 상태 (공중, 지상에서 착륙이나 이륙)

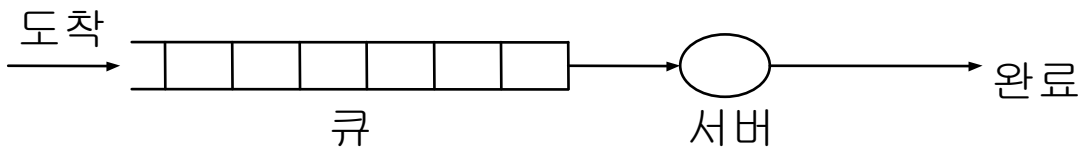
  - 상태 측정 시간
    - 연속적 시스템(continuous system) 시뮬레이션
      - 시스템 상태는 시간에 따라 연속적으로 변함
    - 이산 시스템(discrete system) 시뮬레이션
      - 상태 변수들은 어떤 특정 사건 발생 시점에서만 변함

컴퓨터 시뮬레이션(2)

- 시뮬레이션에서의 시간
  - 물리적 시스템에서의 시간을 의미하는 시뮬레이트 되는 시간 (simulated time)과 시뮬레이션 프로그램에서의 연산 시간 (computation time)을 분명하게 구별해야 됨
  - 대부분의 경우는 시뮬레이트 되는 시간보다 연산 시간이 훨씬 짧음
    - 예 : 기상 예측 시스템

컴퓨터 시뮬레이션(1)

- 단일 서버 큐잉 시스템

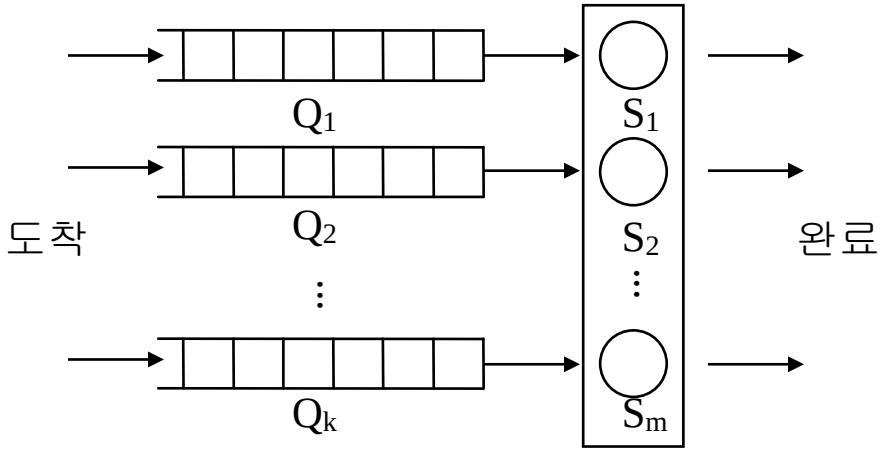


컴퓨터 시뮬레이션(1)

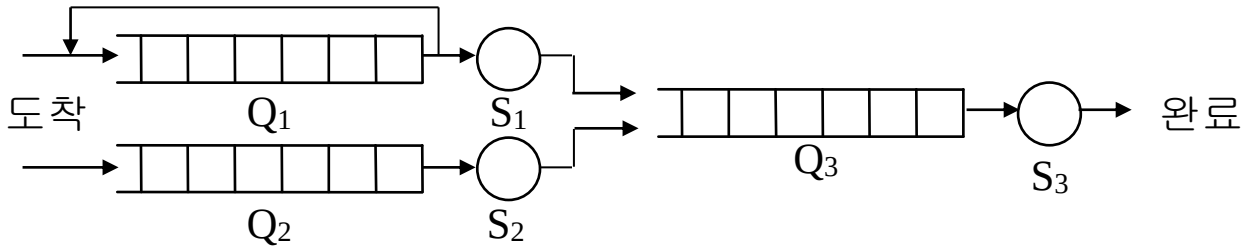
- 큐잉 시스템 시뮬레이션 프로그램
  - 구성 요소
    - 고객 큐(queue) : 대기선을 모델링
    - 서버(server) : 고객들을 서비스
    - 스케줄러(scheduler) : 시뮬레이션하는 동안 사건이 일어날 시간을 스케줄
  - 시뮬레이션에 필요한 자료
    - 새로운 고객 도착 : 확률 분포 함수 A(t)에 따라 정해지는 시간
    - 서비스를 받고 시스템을 빠져 나감 : 확률 분포 함수 S(t)에 따라 결정되는 시간
  - 시뮬레이션 방법
    - 시간 중심 시뮬레이션(time-driven simulation)
      - 일정 단위 시간을 시뮬레이션 시계에 계속적으로 증가시키면서 시뮬레이션을 수행
    - 사건 중심 시뮬레이션(event-driven simulation)
      - 사건이 발생할 때마다 경과된 시간을 시뮬레이션 시계에 증가시키고, 상태 변수를 갱신하면서 시뮬레이션을 수행

컴퓨터 시뮬레이션(2)

- 다중 큐 다중 서버 큐잉 시스템



- 큐잉 네트워크



6.7 우선순위 큐

- 생략 (8장에서 배움)

6.8 덱(Deque : double-ended queue)

- Stack과 Queue의 성질을 종합한 순서 리스트
- 삽입과 삭제가 리스트의 양끝에서 임의로 수행될 수 있는 자료구조
- Stack이나 Queue ADT가 지원하는 연산은 모두 지원

Deque의 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;
```

공백 Deque에 대한 일련의 연산 수행 예

Deque 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)

Deque(4)

- Stack과 Queue ADT 연산에 대응하는 Deque의 연산
  - Stack ADT 연산에 대응하는 연산

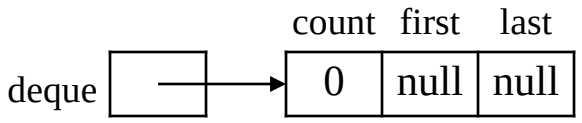
Stack 연산	Deque 연산
createStack()	createDeque()
push(stack,e)	insertLast(deque,e)
isEmpty(stack)	isEmpty(deque)
pop(stack)	deleteLast(deque)
delete(stack)	removeLast(deque)
peek(stack)	peekLast(deque)

  - Queue ADT 연산에 대응하는 연산

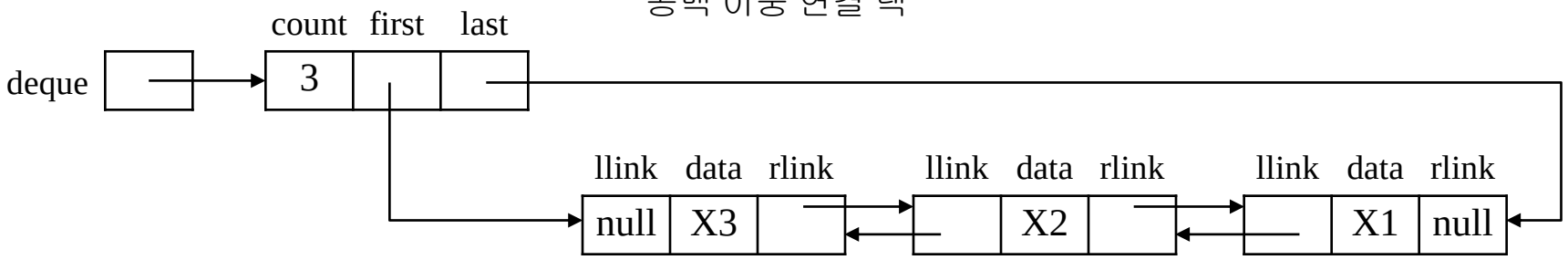
Queue 연산	Deque 연산
createQ()	createDeque()
enqueue(queue,e)	insertLast(deque,e)
isEmpty(queue)	isEmpty(deque)
dequeue(queue)	deleteFirst(deque)
delete(queue)	removeFirst(deque)
peek(queue)	peekFirst(deque)

Deque의 구현

- 단순 연결 리스트로 구현
  - 이점 : 리스트의 마지막 노드를 가리키는 포인터를 이용
  - 단점 : 리스트 마지막 노드의 삭제를 상수 시간에 수행할 수 없음
- 이중 연결 리스트로 구현
  - 이점 : 리스트 양쪽 끝에서 삽입과 삭제가 상수 시간에 수행
  - first 링크와 last 링크를 사용
    - 리스트의 첫 번째 노드와 마지막 노드만을 가리키는 링크
    - 다른 데이타를 저장할 목적이 아님



공백 이중 연결 덱



3개의 원소를 포함한 이중 연결 덱

이중 연결 리스트를 이용한 Deque 구현 (1)

```
/**
 * 이중 연결 리스트로 구현
 */
public class Deque {

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

    private DoubleListNode first;
    private DoubleListNode last;
    private int count;

    public Deque() {
        first = null;
        last  = null;
        count = 0;
    }

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

이중 연결 리스트를 이용한 Deque 구현 (2)

```
/**
 * 연결 덱에 첫 번째 원소를 삽입
 */
public void insertFirst(Object value) {
    DoubleListNode newNode;
    newNode = new DoubleListNode();
    newNode.data = value;

    if (count == 0) {          //덱이 공백인 경우
        first = newNode;
        last  = newNode;
        rlink = llink = null;
    } else {
        first.llink   = newNode;
        newNode.rlink = first;
        newNode.llink = null;
        first = newNode;
    }

    count++;
}
```

이중 연결 리스트를 이용한 Deque 구현 (3)

```
/**
 * 연결 덱에 마지막 원소로 삽입
 */
public void insertLast(Object value) {
    DoubleListNode newNode;
    newNode = new DoubleListNode();
    newNode.data = value;

    if (count == 0) {    //덱이 공백인 경우
        first = newNode;
        last  = newNode;
        rlink = null;
        llink = null;
    } else {
        last.rlink = newNode;
        newNode.rlink = null;
        newNode.llink = last;
        last = newNode;
    }
    count++;
}
```

이중 연결 리스트를 이용한 Deque 구현 (4)

```
/**
 * 연결 덱에서 첫 번째 원소를 삭제하고 반환
 */
public Object deleteFirst() {
    if (count == 0) {                    // 연결 덱이 공백인 경우
        return null;
    } else {
        Object value = first.data;
        if (first.rlink == null) {       //원소가 1개인 경우

            first = null;
            last  = null;
         } else {                        // 원소가 2개 이상인 경우
            first = first.rlink;
            first.llink = null;
         }
         count--;

         return value;
    }
}
```

이중 연결 리스트를 이용한 Deque 구현 (5)

```
/**
 * 연결 덱에서 마지막 원소를 삭제하고 반환
 */
public Object deleteLast() {
    if (count = = 0) {          // 연결 덱이 공백인 경우
        return null;
    } else {
        Object value = last.data;
        if (last.llink = = null) {   // 원소가 1개인 경우
            first = null;
            last  = null;
        } else {                // 원소가 2개 이상인 경우
            last = last.llink;
            last.rlink = null;
        }
        count--;
        return value;
    }
}
```

이중 연결 리스트를 이용한 Deque 구현 (6)

```
    public void removeFirst() {
            ⋮
    }

    public void removeLast() {
            ⋮
    }

    public Object peekFirst() {
            ⋮
    }

    public Object peekLast() {
            ⋮
    }
}
```

Deque을 이용한 Stack 구현(1)

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

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

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

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

Deque을 이용한 Stack 구현(2)

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

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

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

}
```