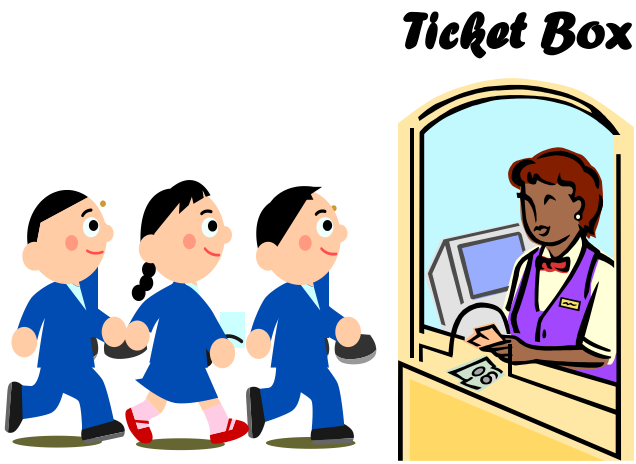
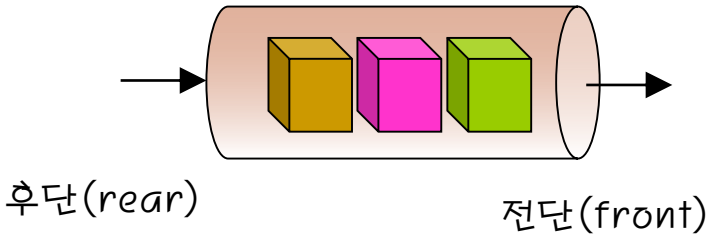
6장: 큐

- 목차
  - 1. 큐
  - 2. 배열로 구현된 큐
  - 3. 연결 리스트로 구현된 큐
  - 4. 덱
  - 5. 큐의 응용

1. 큐(QUEUE)

- 큐: 먼저 들어온 데이터가 먼저 나가는 자료구조
- 선입선출(FIFO: First-In First-Out)
- (예)매표소의 대기열





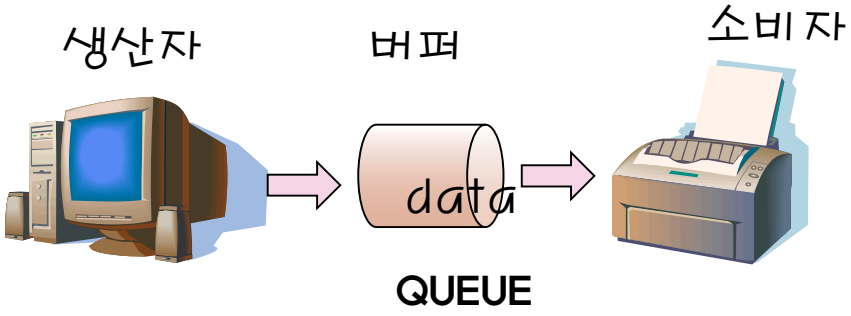
큐 ADT

- 삽입과 삭제는 FIFO순서를 따른다.
- 삽입은 큐의 후단에서, 삭제는 전단에서 이루어진다.

•객체: n개의 element형으로 구성된 요소들의 순서있는 모임
•연산:
- create() ::= 큐를 생성한다.
- init(q) ::= 큐를 초기화한다.
- is_empty(q) ::= 큐가 비어있는지를 검사한다.
- is_full(q) ::= 큐가 가득 찼는가를 검사한다.
- enqueue(q, e) ::= 큐의 뒤에 요소를 추가한다.
- dequeue(q) ::= 큐의 앞에 있는 요소를 반환한 다음 삭제한다.
- peek(q) ::= 큐에서 삭제하지 않고 앞에 있는 요소를 반환한다.

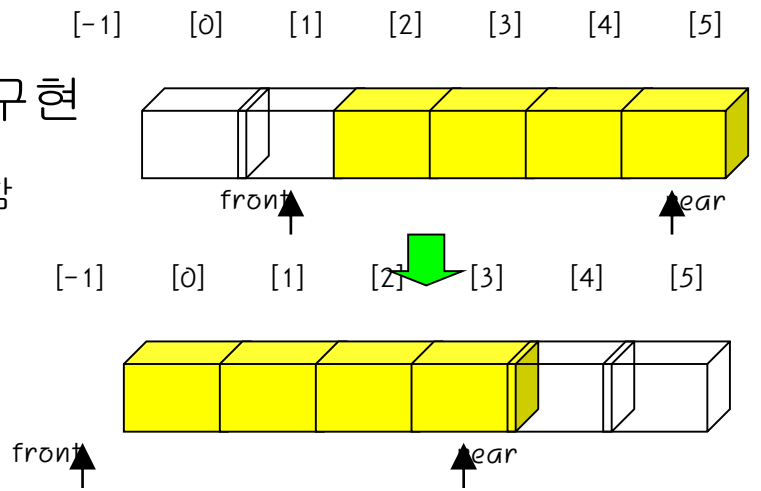
큐의 응용

- 직접적인 응용
  - 시뮬레이션의 대기열(공항에서의 비행기들, 은행에서의 대기열)
  - 통신에서의 데이터 패킷들의 모델링에 이용
  - 프린터와 컴퓨터 사이의 버퍼링
- 간접적인 응용
  - 스택과 마찬가지로 프로그래머의 도구
  - 많은 알고리즘에서 사용됨

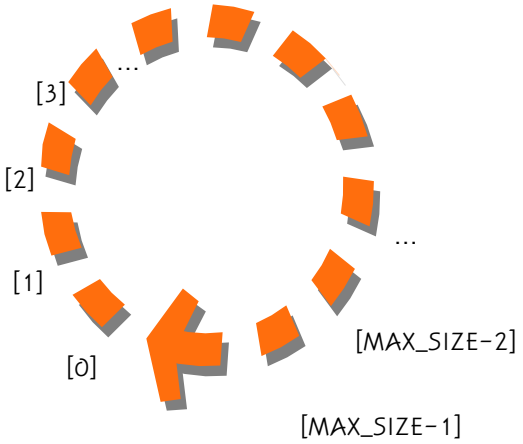


2. 배열로 구현된 큐

- 선형큐: 배열을 선형으로 사용하여 큐를 구현
  - 삽입을 계속하기 위해서는 요소들을 이동시켜야 함
  - 문제점이 많아 사용되지 않음

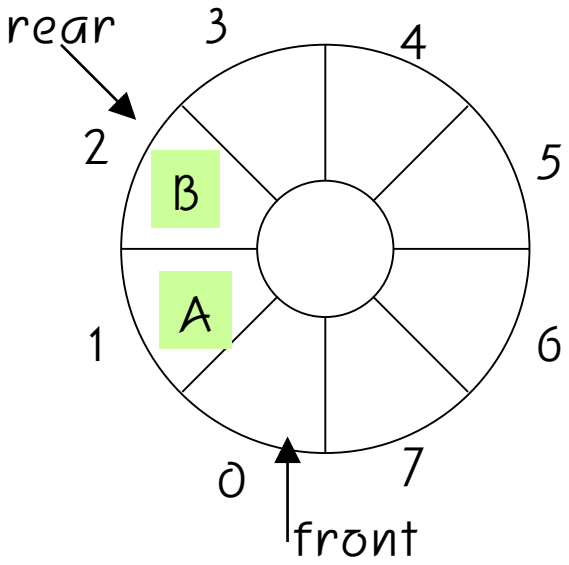


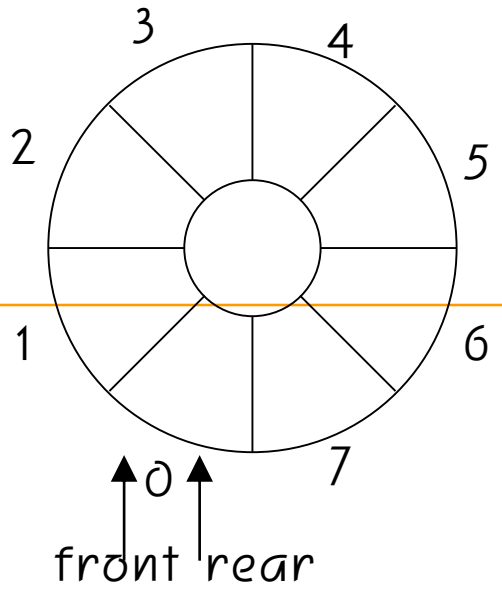
- 원형큐: 배열을 원형으로 사용하여 큐를 구현



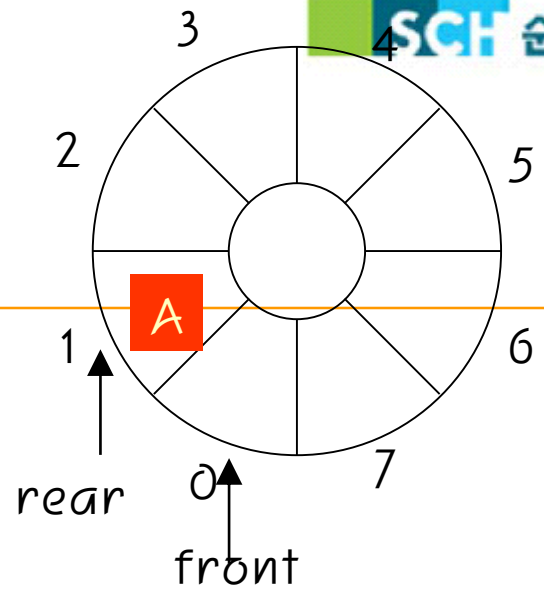
큐의 구조

- 큐의 전단과 후단을 관리하기 위한 2개의 변수 필요
- front: 첫번째 요소 하나 앞의 인덱스
- rear: 마지막 요소의 인덱스

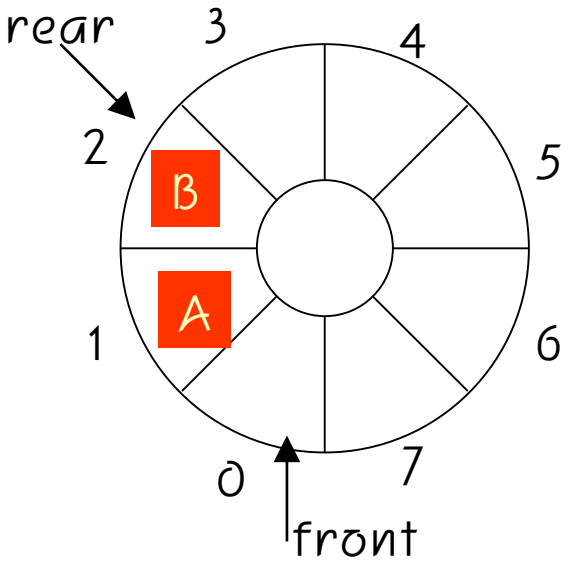




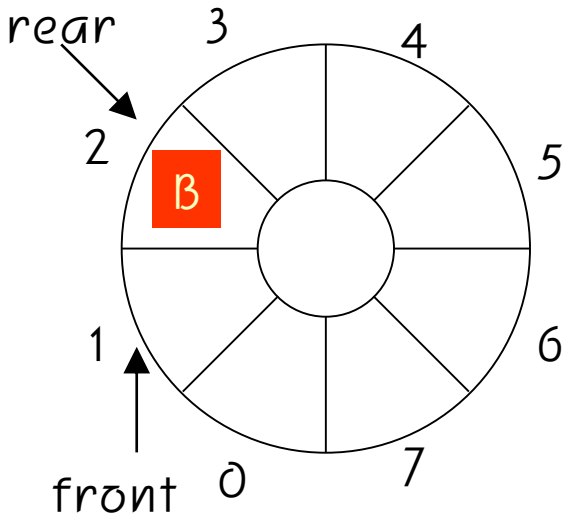
(a) 초기상태



(b) A 삽입



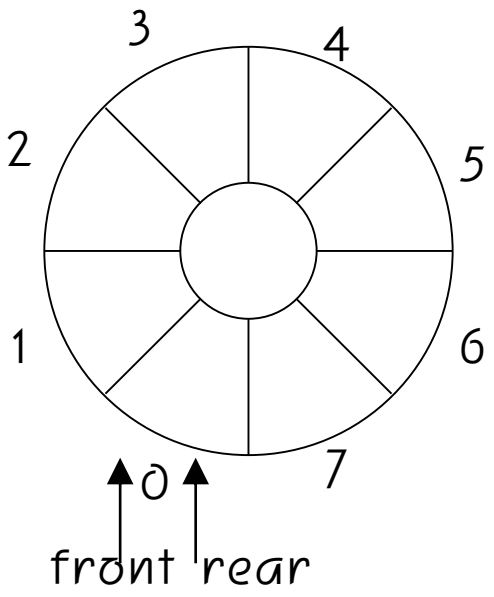
(c) B 삽입



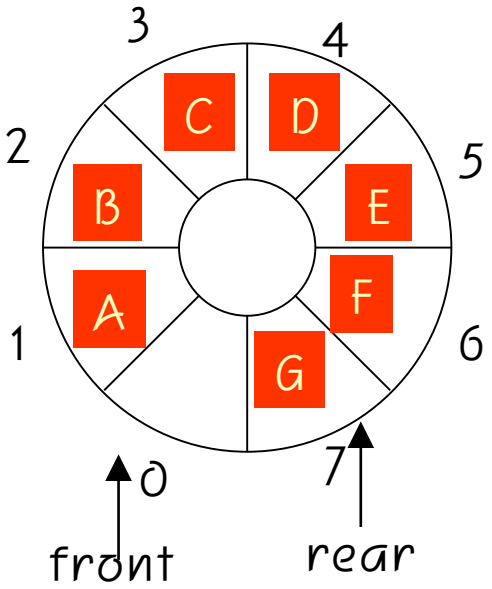
(d) A 삭제

공백상태, 포화상태

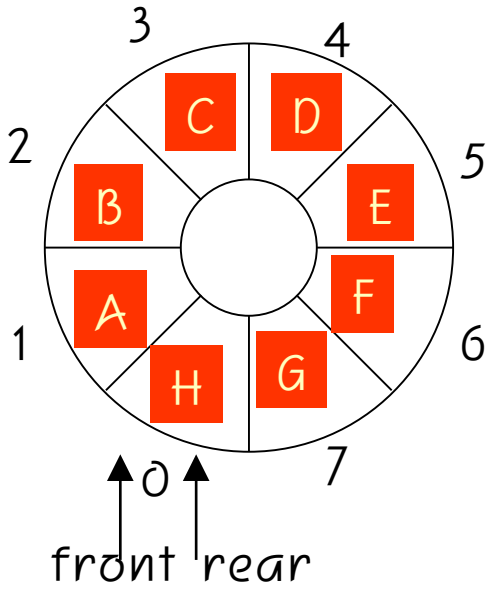
- 공백상태: front == rear
- 포화상태: front == (rear+1) % M
- 공백상태와 포화상태를 구별하기 위하여 하나의 공간은 항상 비워둔다.



(a) 공백상태



(b) 포화상태



(c) 오류상태

큐의 연산

- 나머지(modulo) 연산을 사용하여 인덱스를 원형으로 회전시킨다.

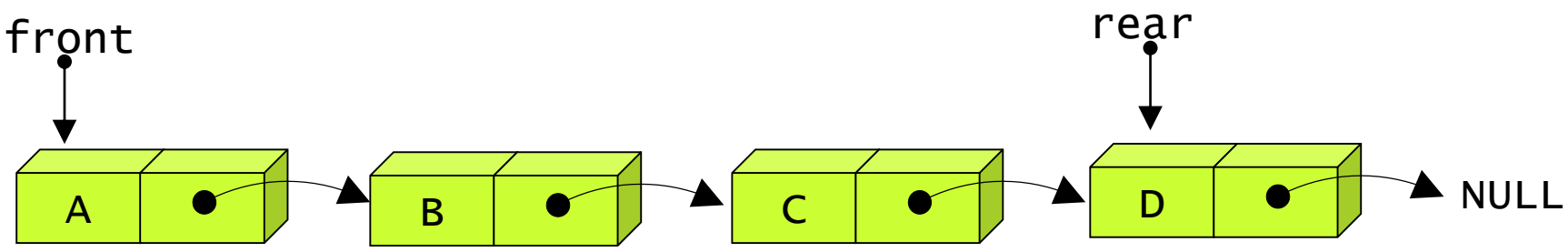
front = (front+1) % M;

rear = (rear+1) % M;

```
// 공백 상태 검출 함수
int is_empty(QueueType *q)
{
        return (q->front == q->rear);
}
// 포화 상태 검출 함수
int is_full(QueueType *q)
{
        return ((q->rear+1)%MAX_QUEUE_SIZE == q-
>front);
}
// 삽입 함수
void enqueue(QueueType *q, element item)
{
        if( is_full(q) )
                error("큐가 포화상태입니다");
        q->rear = (q->rear+1) % MAX_QUEUE_SIZE;
        q->queue[q->rear] = item;
}
// 삭제 함수
element dequeue(QueueType *q)
{
        if( is_empty(q) )
                error("큐가 공백상태입니다");
        q->front = (q->front+1) % MAX_QUEUE_SIZE;
        return q->queue[q->front];
}
```

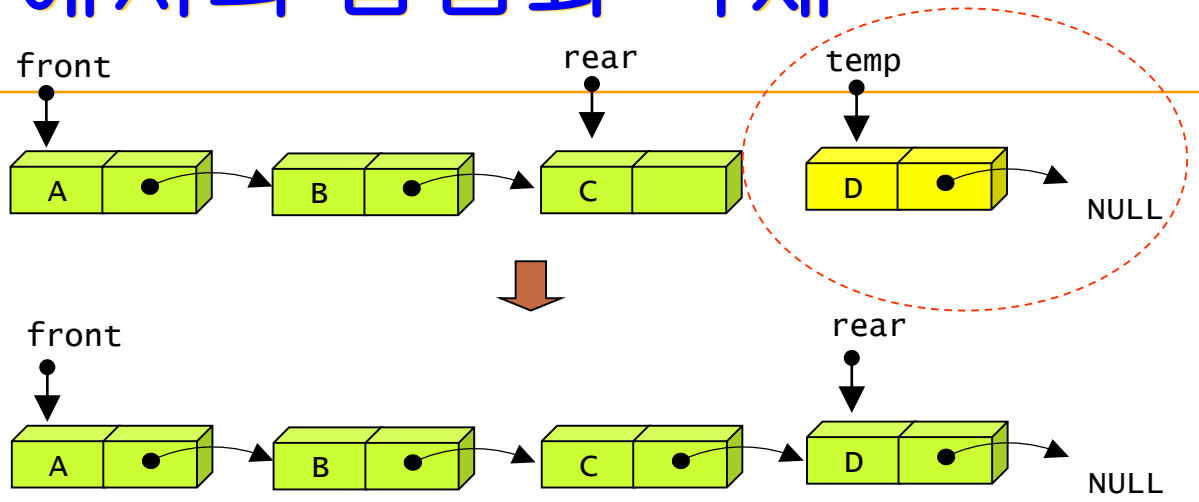
3. 연결 리스트로 구현된 큐

- 연결된 큐(linked queue): 연결리스트로 구현된 큐
- front 포인터는 삭제와 관련되며 rear 포인터는 삽입
- front는 연결 리스트의 맨 앞에 있는 요소를 가리키며, rear 포인터는 맨 뒤에 있는 요소를 가리킨다
- 큐에 요소가 없는 경우에는 front와 rear는 NULL

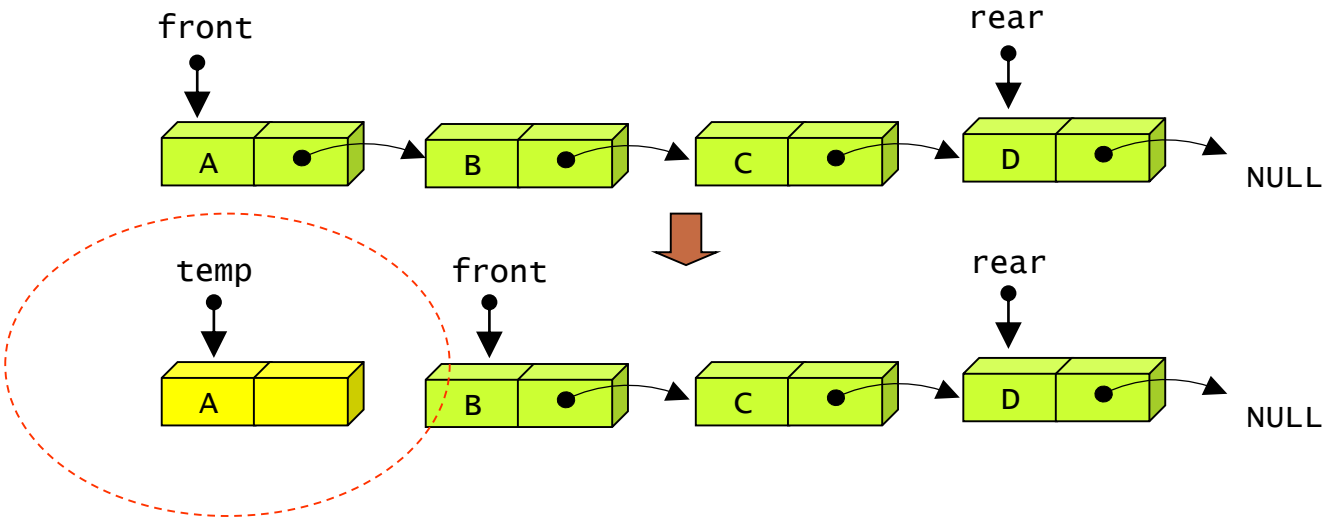


연결된 큐에서의 삽입과 삭제

삽입

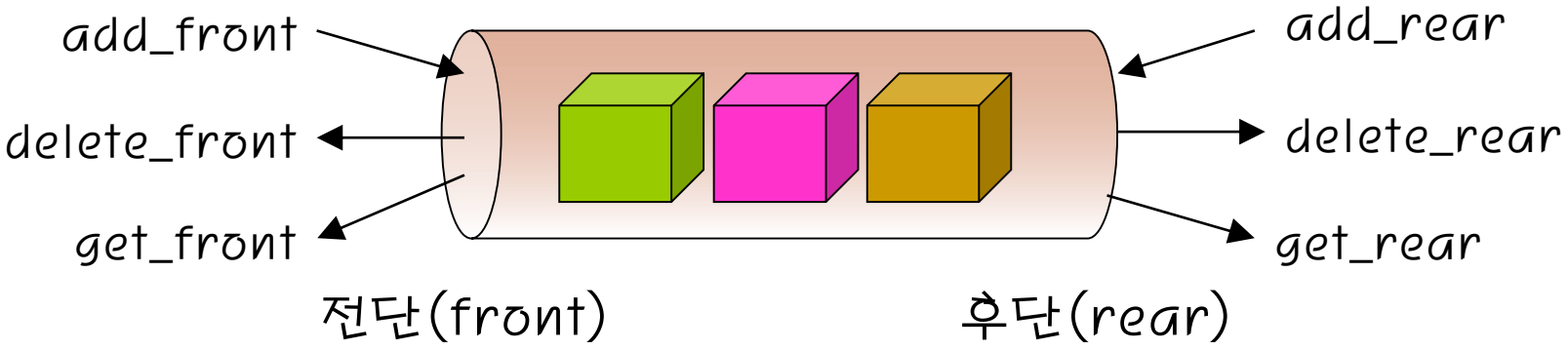


삭제



4. 덱(deque)

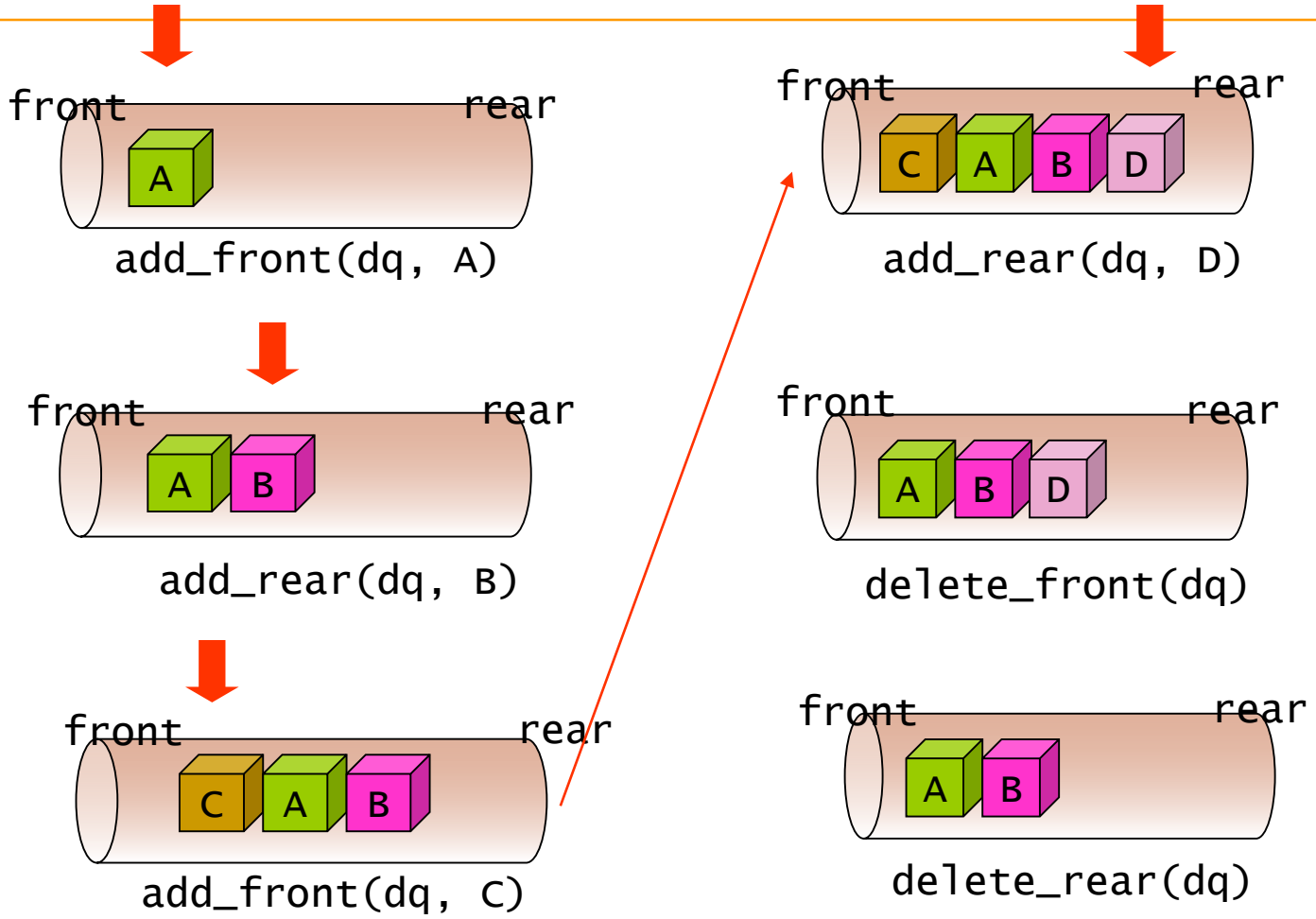
- 덱(deque)은 double-ended queue의 줄임말로서 큐의 전단(front)와 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



덱 ADT

- 객체: n개의 element형으로 구성된 요소들의 순서있는 모임
- 연산:
  - create() ::= 덱을 생성한다.
  - init(dq) ::= 덱을 초기화한다.
  - is_empty(dq) ::= 덱이 공백상태인지를 검사한다.
  - is_full(dq) ::= 덱이 포화상태인지를 검사한다.
  - add_front(dq, e) ::= 덱의 앞에 요소를 추가한다.
  - add_rear(dq, e) ::= 덱의 뒤에 요소를 추가한다.
  - delete_front(dq) ::= 덱의 앞에 있는 요소를 반환한 다음 삭제한다
  - delete_rear(dq) ::= 덱의 뒤에 있는 요소를 반환한 다음 삭제한다.
  - get_front(q) ::= 덱의 앞에서 삭제하지 않고 앞에 있는 요소를 반환한다.
  - get_rear(q) ::= 덱의 뒤에서 삭제하지 않고 뒤에 있는 요소를 반환한다.

덱의 연산


front
rear
A
add_front(dq, A)
front
rear
A
B
add_rear(dq, B)
front
rear
C
A
B
add_front(dq, C)
front
rear
C
A
B
D
add_rear(dq, D)
front
rear
A
B
D
delete_front(dq)
front
rear
A
B
delete_rear(dq)

덱의 구현

- 양쪽에서 삽입, 삭제가 가능하여야 하므로 일반적으로 이중연결 리스트 사용

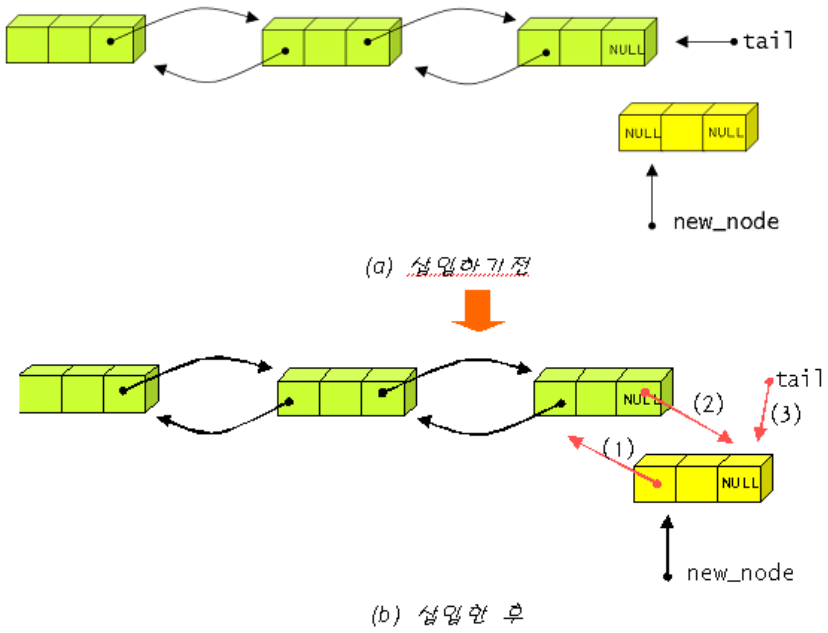
```
typedef int element;              // 요소의 타입


typedef struct DlistNode {      // 노드의 타입
     element data;
     struct DlistNode *llink;
     struct DlistNode *rlink;
} DlistNode;


typedef struct DequeType {      // 덱의 타입
     DlistNode *head;
     DlistNode *tail;
} DequeType;
```

덱에서의 삽입연산

- 연결리스트의 연산과 유사
- 헤드포인터 대신 head와 tail 포인터 사용



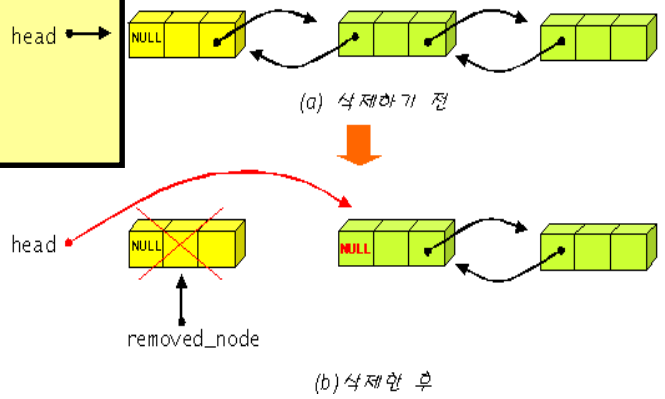

```
void add_rear(DequeType *dq, element item)
{
  DlistNode *new_node =  create_node(dq-
>tail, item, NULL);
  if( is_empty(dq))
     dq->head = new_node;
  else
     dq->tail->rlink = new_node;
  dq->tail = new_node;
}
//
void add_front(DequeType *dq, element item)
{
  DlistNode *new_node = create_node(NULL,
item, dq->head);

  if( is_empty(dq))
          dq->tail = new_node;
  else
          dq->head->llink = new_node;
  dq->head = new_node;
}
```

덱에서의 삭제연산

```
// 전단에서의 삭제
element delete_front(DequeType *dq)
{
        element item;
        DlistNode *removed_node;

        if (is_empty(dq))  error("공백 덱에서 삭제");
        else {
                removed_node = dq->head;    // 삭제할 노드
                item = removed_node->data; // 데이터 추출
                dq->head = dq->head->rlink; // 헤드 포인터 변경
                free(removed_node);                // 메모리 공간 반납
                if (dq->head == NULL) // 공백상태이면
                        dq->tail = NULL;
                else        // 공백상태가 아니면
                        dq->head->llink=NULL;
        }
        return item;
}
```



5. 큐의 응용 : 버퍼

- 큐는 서로 다른 속도로 실행되는 두 프로세스 간의 상호 작용을 조화시키는 버퍼 역할을 담당
  - CPU와 프린터 사이의 프린팅 버퍼, 또는 CPU와 키보드 사이의 키보드 버퍼
- 대개 데이터를 생산하는 생산자 프로세스가 있고 데이터를 소비하는 소비자 프로세스가 있으며 이 사이에 큐로 구성되는 버퍼가 존재

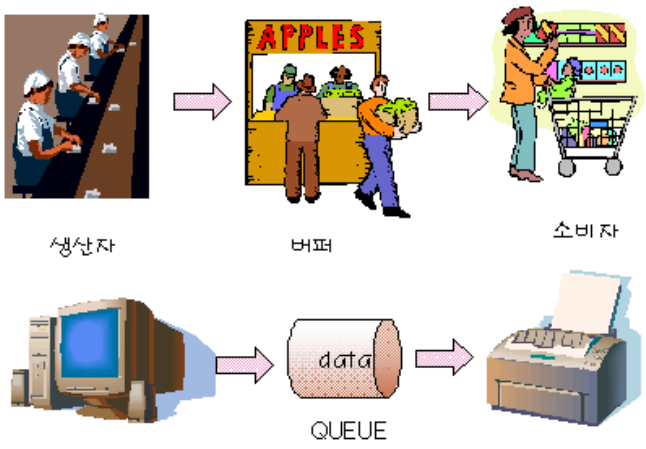


그림 6.17 생산자와 버퍼, 소비자의 개념

```
QueueType buffer;

/* 생산자 프로세스 */
producer()
{
     while(1){
              데이터 생산;
              while( lock(buffer) != SUCCESS ) ;
              if( !is_full(buffer) ){
                       enqueue(buffer, 데이터);
              }
              unlock(buffer);
     }
}
/* 생산자 프로세스 */
consumer()
{
     while(1){
              while( lock(buffer) != SUCCESS )  ;
              if( !is_empty(buffer) ){
                       데이터 = dequeue(buffer);
                       데이터 소비;
              }
              unlock(buffer);
     }
}
```

큐의 응용: 시뮬레이션

- 큐잉이론에 따라 시스템의 특성을 시뮬레이션하여 분석하는 데 이용
- 큐잉모델은 고객에 대한 서비스를 수행하는 서버와 서비스를 받는 고객들로 이루어진다
- 은행에서 고객이 들어와서 서비스를 받고 나가는 과정을 시뮬레이션
  - 고객들이 기다리는 평균시간을 계산

그림 6.18 은행에서의 서비스 대기큐

큐의 응용: 시물레이션

- 시뮬레이션은 하나의 반복 루프
- 현재시각을 나타내는 clock이라는 변수를 하나 증가
- is_customer_arrived 함수를 호출한다. is_customer_arrived 함수는 랜덤 숫자를 생성하여 시뮬레이션 파라미터 변수인 arrival_prov와 비교하여 작으면 새로운 고객이 들어왔다고 판단
- 고객의 아이디, 도착시간, 서비스 시간 등의 정보를 만들어 구조체에 복사하고 이 구조체를 파라미터로 하여 큐의 삽입 함수 enqueue()를 호출한다.
- 여기서 고객이 필요로 하는 서비스 시간은 역시 랜덤숫자를 이용하여 생성된다.
- 지금 서비스하고 있는 고객이 끝났는지를 검사: 만약 service_time이 0이 아니면 어떤 고객이 지금 서비스를 받고 있는 중임을 의미한다.
- clock이 하나 증가했으므로 service_time을 하나 감소시킨다.
- 만약 service_time이 0이면 현재 서비스받는 고객이 없다는 것을 의미한다. 따라서 큐에서 고객 구조체를 하나 꺼내어 서비스를 시작한다.

큐의 응용: 시뮬레이션

```
// 시뮬레이션 프로그램
void main()
{
  int service_time=0;

  clock=0;
  while(clock < duration){
    clock++;
    printf("현재시각=%d\n",clock);
    if (is_customer_arrived()) {
      insert_customer(clock);
    }
    if (service_time > 0)
      service_time--;
    else {
      service_time = remove_customer();
    }
  }
  print_stat();
}
```

```
현재시각=1
고객 0이 1분에 들어옵니다. 서비스시간은 3분입니다.
고객 0이 1분에 서비스를 시작합니다. 대기시간은 0분이었습니다.
현재시각=2
고객 1이 2분에 들어옵니다. 서비스시간은 5분입니다.
현재시각=3
고객 2이 3분에 들어옵니다. 서비스시간은 3분입니다.
현재시각=4
고객 3이 4분에 들어옵니다. 서비스시간은 5분입니다.
고객 1이 4분에 서비스를 시작합니다. 대기시간은 2분이었습니다.
현재시각=5
현재시각=6
현재시각=7
고객 4이 7분에 들어옵니다. 서비스시간은 5분입니다.
현재시각=8
현재시각=9
고객 5이 9분에 들어옵니다. 서비스시간은 2분입니다.
고객 2이 9분에 서비스를 시작합니다. 대기시간은 6분이었습니다.
현재시각=10
고객 6이 10분에 들어옵니다. 서비스시간은 1분입니다.
서비스받은 고객수 = 3
전체 대기 시간 = 8분
1인당 평균 대기 시간 = 2.666667분
아직 대기중인 고객수 = 4
```