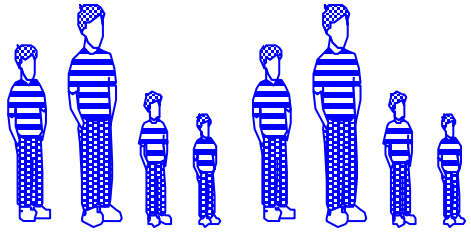
Queue

Queue

- 저장 방식: FIRST IN FIRST OUT (FIFO, 선입선출)
- 큐에 진입한 순서대로 큐에서 제거된다
- Example



- First Come First Serve (FCFS)

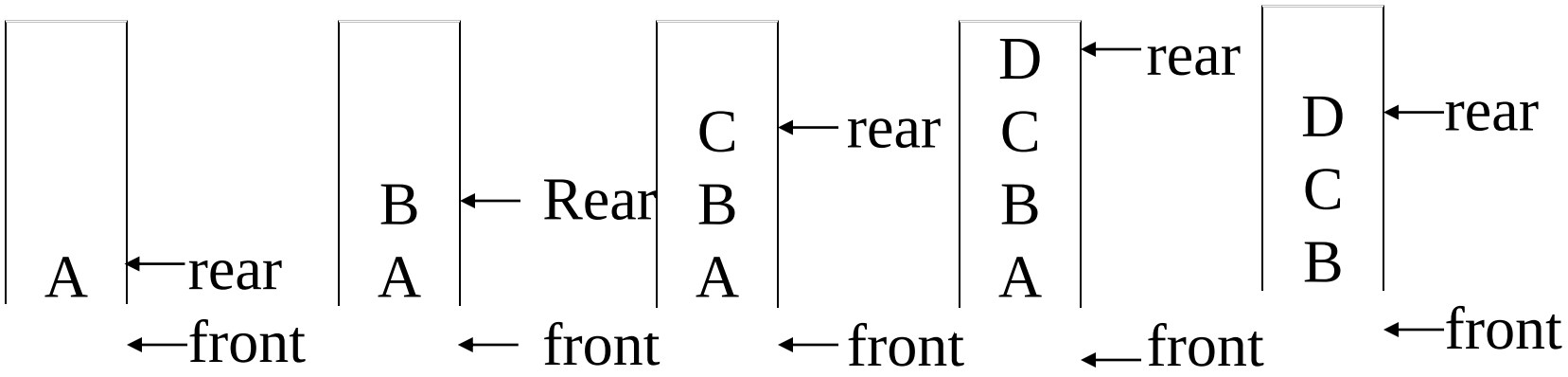
Queue Applications

- 실생활 예
  - Waiting in line
  - Waiting on hold for technical support
- 컴퓨터 운영체제에서의 예
  - Threads
  - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)

Queue 구조

- 선형 리스트(Linear list)
  - infinite queue : 이론적으로만 성립
  - finite queue : 대부분 유한 큐
- 2개의 pointer
  - front (진출 pointer) and
  - rear (진입 pointer)
- 주요 동작
  - AddQueue --- cf. Append, Insert, ..
  - DeleteQueue --- cf. Remove, ...

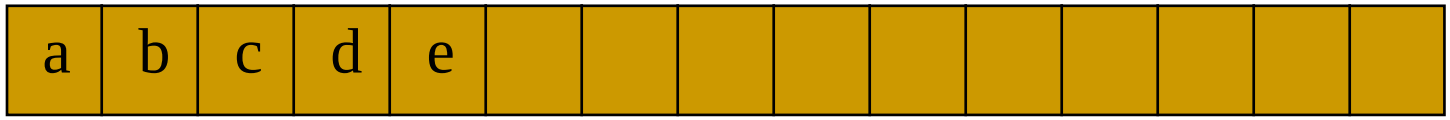
First In First Out



Applications: Job Scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

1차원 배열을 사용한 Queue의 구성



DeleteQ() => delete queue[0]

- **O(sizeof(Queue))** time --- O(N) why?
  - front = front + 1;
  - element = queue[front];
  - Squeeze[queue]; front = front -1 ; rear = rear – 1;

AddQ(element) => if there is capacity, add at right end

- **O(1)** time
  - rear = rear + 1;
  - queue[rear] = element;

Queue 동작

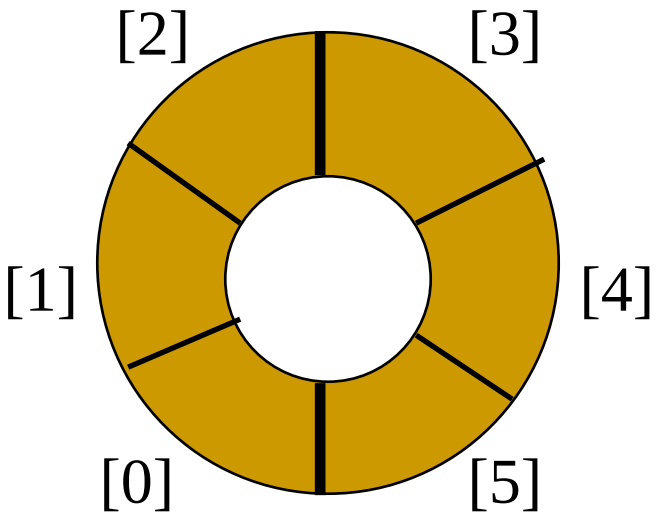
- IsFullQ ... 큐가 full이면 true를 반환
- IsEmptyQ ... 큐가 empty이면 true를 반환
- AddQ ... 큐의 진입구에 데이터를 추가
- DeleteQ ... 큐의 진출구에서 데이터를 삭제

Circular Queue (원형 큐)

- 진입구와 진출구가 연결된 큐
- 1차원 배열을 사용하는 원형 큐

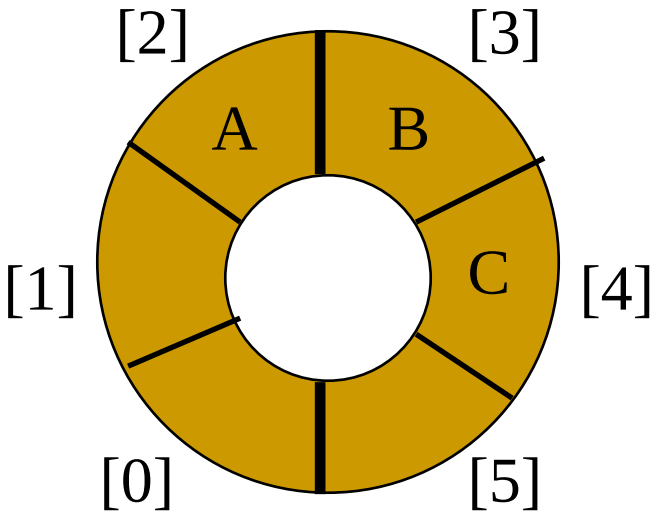
queue[]

- Circular view of array.



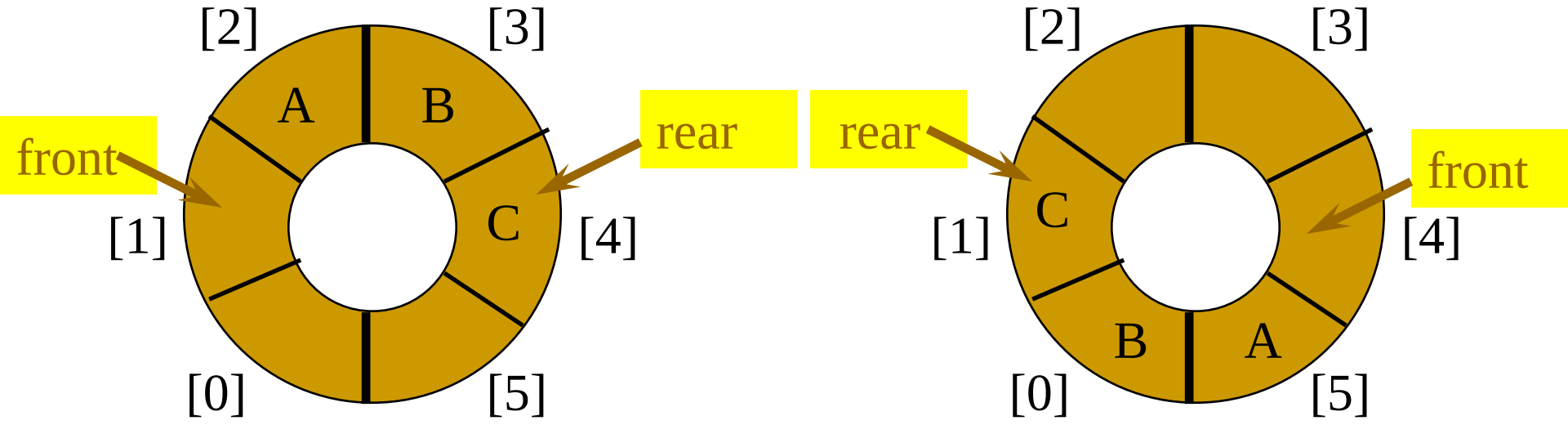
Circular Array

- 3 elements를 가지는 원형 큐의 예



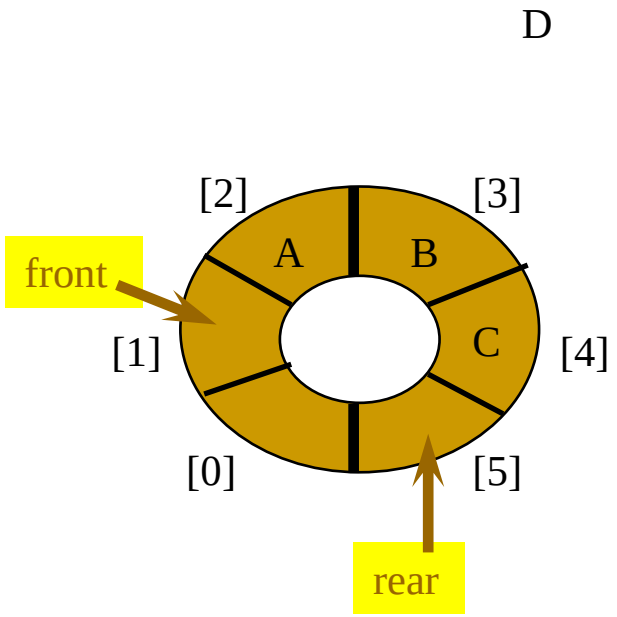
Circular Array

- 2개의 index pointer front 와 rear 를 사용한다
  - 초기에는 front = rear



Add an Element

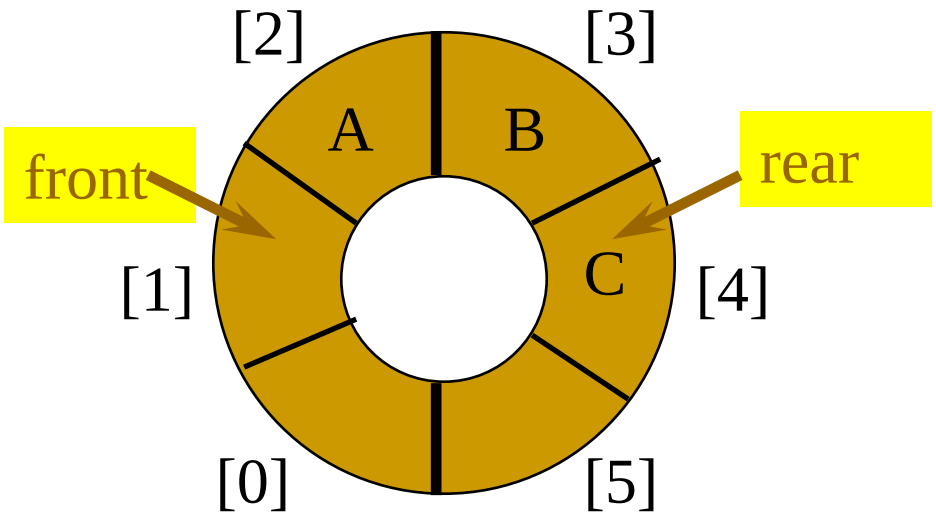
- Add an element
  - rear = rear + 1
  - queue[rear] = element
- Delete an element
  - front = front + 1
  - element = queue[front]



Moving rear Clockwise

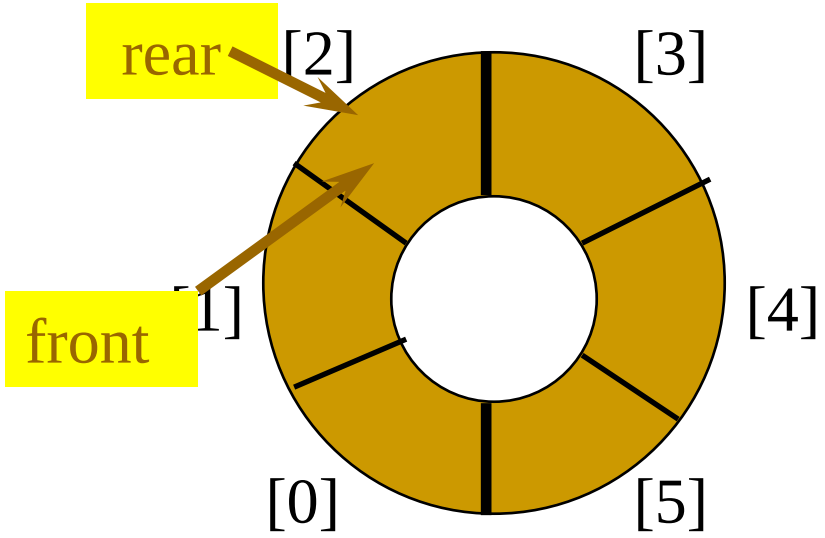
- rear++;

  if (rear = = capacity) rear = 0;



- **rear = (rear + 1) % capacity;**

Empty or Full



- element의 진입과 진출이 이루어지다 보면 큐가 empty가 될 수도 있고 full이 될 수 있다. 그 경우 **front = rear**.
- 그 경우 empty인지 full인지 판단해야 한다

- 개선책
  - 큐가 full이 되지 않도록 하는 것이 중요
    - 만약 큐 full 현상이 발생하면 큐가 크기를 늘려야 한다
  - 큐 full과 큐 empty의 구별을 위한 1가지 방법은 boolean 변수를 사용한다, 예를 들면 lastOperationIsAddQ
    - AddQ 가 실행될 때 마다 이 변수를 true 로 세트
    - DeleteQ 가 실행될 때 마다 false 로 세트
    - 그러므로, (front == rear) && !lastOperationIsAddQ 이면 큐는 empty
    - (front == rear) && lastOperationIsAddQ 이면 큐는 full

- 또 다른 방법 integer 변수를 사용, 예를 들면 size.
  - AddQ 시 size++.
  - DeleteQ 시 size--.
  - 이때, (size == 0) 이면 큐는 empty
  - (size == arrayLength) 이면 queue는 full

원형 Queue의 구현

```
#include <stdio.h>
int *queue;
int size;
int head;
int tail;

void InitQ(int _size) {
   size = _size;
   queue = (int *)malloc(sizeof(int)*size);
   head = tail = 0;
}

bool InsertQ(int data) {
   if( (tail+1)%size == head) {
      return -1;
   }
   queue[tail] = data;
   tail = (tail+1)%size;
   return true;
}
```

```
int DelQ( ) {
   int temp;
   if( ((head+1)%size == tail) {
      return -1;
   }
   temp = queue[head];
   head = (head+1)%size;
   return temp;
}

int main( ) {
   int i, temp;

   InitQ(MAX_SIZE);
   for (i=1; i<MAX_SIZE; i++) {
      InsertQ(i);
   }
   for (; (temp=DelQ( )) != -1;) {
      printf("%d  ", temp);
   }
   printf("₩n");
}
```

Linked List로 구현한 Queue

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
                int data ;
                struct Node *next ;
};
struct Node *front;
struct Node *rear;

void Enqueue(int x) {
   struct Node *temp =
                (struct Node *)malloc(sizeof(struct Node));
   temp->data = x;
   temp->next = null;
   if(front == NULL && rear == NULL) {
      front = rear = temp;
      return;
   }
   rear->next = temp;
   rear = temp;
}
```

```
void Dequeue( ) {
   struct Node *temp = front;
   if(front == NULL) {
      printf("Queue is empty₩n");
      return;
   }
   if(front == rear) {
      front = rear = NULL;
   }
   else {
      front = front -> next;
   }
   free(temp);
}
int Front( ) {
   if(front == NULL) {
      printf("Queue is empty₩n");
      return;
   }
   return front->data;
}
```

```
void Print( ) {
    struct Node *temp = front;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("₩n");
}

int main( ) {
    Enqueue(2);
    Print();
    Enque(4);
    Print();
    Enque(6);
    Print();
    Dequeue();
    Print();
    Enqueue(8);
    Print();
}
```

큐의 응용

- 동작 속도가 다른 두 장치 프로세스 간의 효율개선을 위한 Buffer로서의 역할
  - CPU와 프린터 사이, CPU와 키보드 사이, 디스크와 메인메모리 사이, 통신용
- 생산자 프로세스와 소비자 프로세스 간의 효율적 데이터 전달
  - 대부분 원형 큐를 사용

Deque (데크)

- 덱(**deque**)
  - Double-ended queue의 줄임말
  - 큐의 전단(front)와 후단(rear)에서 모두 삽입과 삭제가 가능한 큐
  - 양쪽에서 삽입, 삭제가 가능하여야 하므로 일반적으로 이중연결 리스트를 사용한다

```
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;
```

- 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