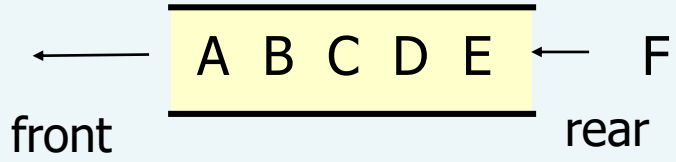
5. 큐 (Queue)

한국항공대학교 안준선

큐 (Queue)

- 큐
  - 저장된 순서대로 꺼내는 자료 저장소 : FIFO (First In First Out)
  - 추상적 자료형으로서 다음 두 가지 조작만이 제공된다.
    - put : 큐의 맨 뒤(rear)에 자료를 집어넣음
    - get : 큐의 맨 앞(front)에서 자료를 빼냄



  - 발생한 순서대로 처리해야 할 대부분의 작업에 대하여 큐를 사용한다.

ex) 은행 창구의 작업, Keyboard 버퍼, 이벤트 버퍼

배열을 이용한 큐의 구현

- 배열을 이용한 큐의 구현
  - 미리 큐에 들어갈 수 있는 원소의 최대 개수를 지정해야 한다.
  - 큐의 front와 rear를 지정하는 변수를 사용한다.

```
#define MAX 10
typedef struct _queue {
    int data[MAX];
    int front=0;  int rear=0;
} queue;
```

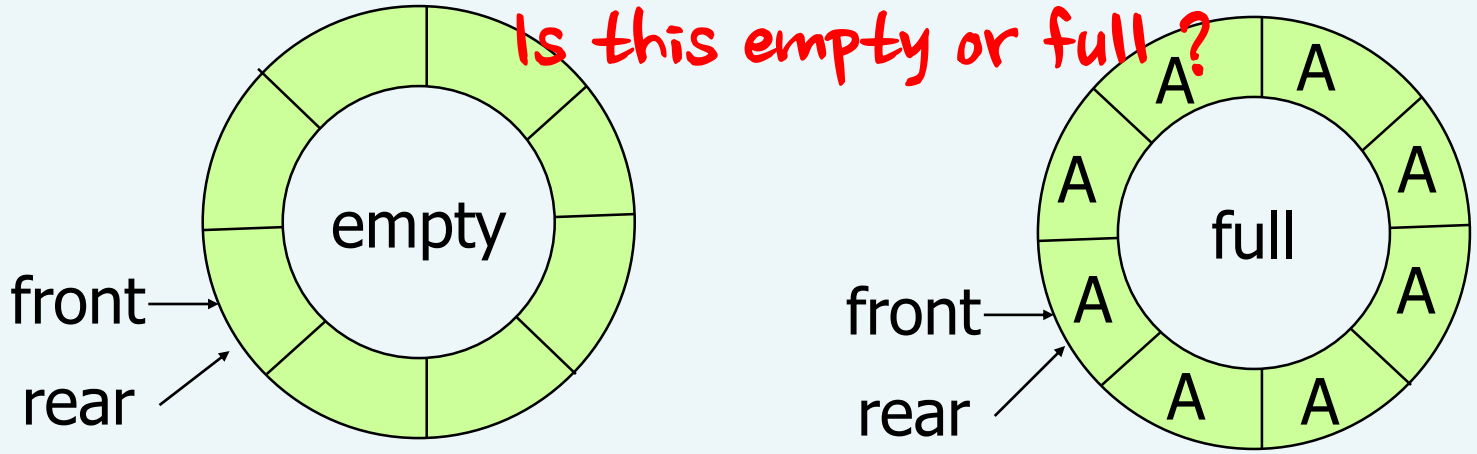
  - 간단한 구현

```
int put(queue *q, int k) {
   <q->rear == MAX이면 오버플로 에러 처리>
   q->data[q->rear++] = k;
}
int get(queue *q) {
   <q->front == q->rear 이면 원소없음 에러 처리>
   return q->data[q->front++];
}
```

→ 문제점 : 저장, 꺼내기 위치가 뒤로만 이동: 앞쪽에 빈 공간이 있어도 overflow 발생

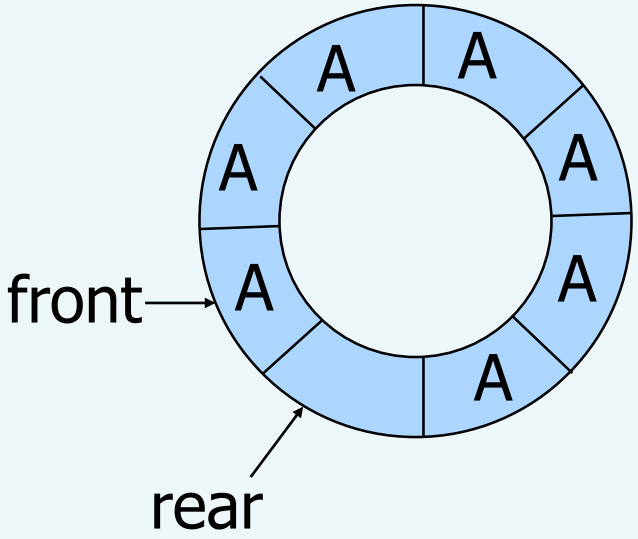
배열을 이용한 큐의 구현

- 문제점의 해결 : 원형 큐의 사용
  - 맨 마지막 위치(MAX)에 put/get한 다음에는 다시 처음(0)으로 돌아가서 put/get을 수행한다. (1 increment 후 modular 연산 수행)
    - front : 실제 자료가 저장된 선두 (남아있는 원소 중 가장 오래 전에 저장됨)
    - rear : 다음에 저장할 빈 공간 (가장 마지막 원소 뒤)
  - 큐가 비어있는 상태와 꽉찬 상태를 구별하기 위하여 완충지대 필요

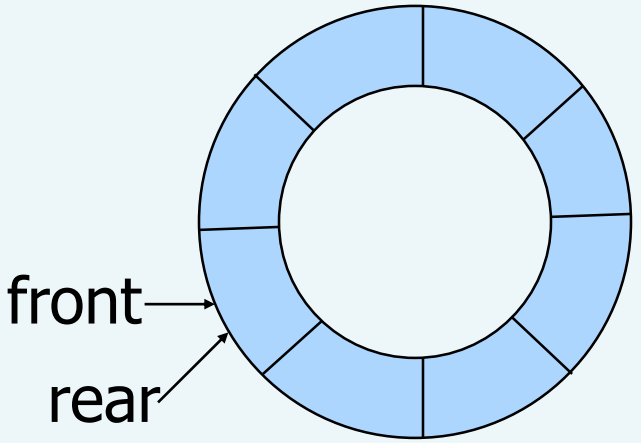


배열을 이용한 큐의 구현

- 완충지대의 사용
  - rear = front−1일 경우에 꽉 찬 것으로 간주 (front 바로 앞을 사용 안함)
  - 하나의 기억장소를 완충지대로 소비함.



(rear+1)%MAX == front → full !



rear == front → empty !

배열을 이용한 큐의 구현

- 큐의 생성과 모든 원소의 삭제

```
queue *create_queue() {
    queue *q = (queue *)malloc(sizeof(queue));
    q->front = q->rear = 0;
    return;
}

void clear_queue(queue *q) {
    q->front = q->rear;
}
```

배열을 이용한 큐의 구현

- put
  - 먼저 큐가 꽉 차 있는지(rear 다음이 front 인지) 검사한다.
  - 다음 저장 장소는 1을 더한 후에 배열의 크기로 %연산을 적용한다.

```
int put(queue *q, int k)
{
    if ((q->rear + 1) % MAX == q->front) /* full */
    {
        printf("\n    Queue overflow.");
        return -1;
    }
    queue->data[q->rear] = k;
    q->rear = ++q->rear % MAX;
    return k;
}
```


https://en.cppreference.com/w/c/language/operator_precedence

배열을 이용한 큐의 구현

- get
  - 먼저 큐가 비어있는지 검사. (front == rear인 경우)
  - front로부터 원소를 빼낸 후에 front를 다음 위치를 가리키도록 함.

```
int get(queue *q){
    int i;
    if (q->front == q->rear)    /* queue is empty */
    {
        printf("\n    Queue underflow.");
        return -1;
    }
    i = q->data[q->front];
    q->front = ++q->front % MAX;
    return i;
}
```

배열을 이용한 큐의 구현

- print_queue : 큐의 모든 원소를 출력한다.

```
void print_queue(queue *q)  {
    int i;
    printf("\n  Queue contents : Front ----> Rear\n");
    for (i = q->front; i != q->rear; i = ++i % MAX)
        printf("%-6d", q->data[i]);
}
```

- main 프로그램

```
void main(void){
    int i;
    queue *q = create_queue();
    printf("\nPut 5, 4, 7, 8, 2, 1");
    put(q, 5); put(q, 4); put(q, 7);
    put(q, 8); put(q, 2); put(q, 1);
    print_queue();
    /* 계속 */
```

배열을 이용한 큐의 구현

- main 프로그램 (계속)

```
    printf("\nGet"); i = get(q);
    print_queue(q);
    printf("\n   getting value is %d", i);
    printf("\nPut 3, 2, 5, 7");
    put(q,3); put(q, 2); put(q, 5); put(q, 7);
    print_queue(q);
    printf("\nNow queue is full, put 3");
    put(q, 3);
    print_queue(q);
    printf("\nInitialize queue");
    clear_queue(q);
    print_queue(q);
    printf("\nNow queue is empty, get");
    i = get(q);
    print_queue(q);
    printf("\n   getting value is %d", i);
}
```

연결리스트를 이용한 큐의 구현

- 연결 리스트를 이용한 큐의 구현
  - 스택과 마찬가지로 노드를 동적으로 할당하면서 메모리의 한계까지 큐의 크기를 확장할 수 있음 (오버플로우가 없음)
  - 제일 끝(tail 앞)에 원소를 삽입해야 하므로 이중 연결리스트를 사용하여 구현
  - 노드의 구조 정의

```
typedef struct _dnode
    {
    int key;
    struct _dnode *prev;
    struct _dnode *next;
    } dnode;

dnode *head, *tail;
```

연결리스트를 이용한 큐의 구현

- 큐의 초기화
  - 이중 연결리스트의 초기화와 같음.
  - head 다음 노드가 front가 되고 tail 이전 노드가 rear가 된다.

```
void create_queue(void)
{
    head = (dnode*)malloc(sizeof(dnode));
    tail = (dnode*)malloc(sizeof(dnode));
    head->prev = head;
    head->next = tail;
    tail->prev = head;
    tail->next = tail;
}
```

연결리스트를 이용한 큐의 구현

- put
  - tail 앞에 새로운 노드를 삽입한다.

```
int put(int k){
    dnode *t;
    if ((t = (dnode*)malloc(sizeof(dnode))) == NULL)
        {
        printf("\n    Out of memory.");
        return -1;
        }
    t->key = k;
    tail->prev->next = t;
    t->prev = tail->prev;
    tail->prev = t;
    t->next = tail;
    return k;
}
```

연결리스트를 이용한 큐의 구현

- get
  - 큐가 비어있는지 검사한 후에 헤드 다음의 노드의 삭제하고 값을 리턴

```
int get(void)
{
    dnode *t;
    int i;
    t = head->next;
    if (t == tail)
        {
        printf("\n    Queue underflow.");
        return -1;
        }
    i = t->key;
    head->next = t->next;
    t->next->prev = head;
    free(t);
    return i;
}
```

연결리스트를 이용한 큐의 구현

- clear_queue : 큐의 모든 노드를 삭제
  - 큐의 모든 노드를 순회하면서 노드들이 차지하고 있는 메모리를 돌려줌

```
void clear_queue(void)
{
     dnode *t;
     dnode *s;
     t = head->next;
     while (t != tail)
          {
          s = t;
          t = t->next;
          free(s);
          }
     head->next = tail;
     tail->prev = head;
}
```

연결리스트를 이용한 큐의 구현

- main 프로그램

```
typedef struct _dnode {
    int key;
    struct _dnode *prev;
    struct _dnode *next;
    } dnode;
dnode *head, *tail;

void main(void){
   create_queue();
   put(1); put(2);
   printf("%d \n", get())
   put(3);
   printf("%d \n", get())
   printf("%d \n", get())
}
```

연결리스트를 이용한 큐의 구현 : 추상적 자료구조

```
typedef struct _dnode {
    int key;
    struct _dnode *prev;
    struct _dnode *next;
    } dnode;

typedef struct _queue {
   struct _dnode *head;
   struct _dnode *tail;
} queue;

queue *create_queue();
int put(queue *q, int k);
int get(queue *q);
void clear_queue(queue *q);
```

정리

- 큐 자료구조
- 큐 자료구조의 구현 (배열과 연결리스트)
- 추상적 자료구조의 구현