CHAPTER 3

STACKS AND QUEUES

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

3.5 A Mazing Problem

3.6 Evaluation of Expressions

- Stack and Queue
  - Special cases of ordered list
  - Ordered list $A=(a_0, a_1, \ldots, a_{n-1})$, $n \geq 0$
    - a_i : element
  - Null or empty list : $A=( )$, $n=0$

- Stack
  - An ordered list in which insertions and deletions are made at one end called the ***top***
    - Stack S= (a_0 , · · · , a_{n-1})
    - a_0: bottom element, a_{n-1}: top element
    - a_i is on top of element a_{i-1} ($0 < i < n$)
  - Known as a Last-In-First-Out (LIFO) list

삽입하고 삭제가 한쪽 끝에서만 이루어진다!

- Ex)
  - Adding the elements *A, B,* C, *D, E* to the stack
  - Then, deleting an element from the stack

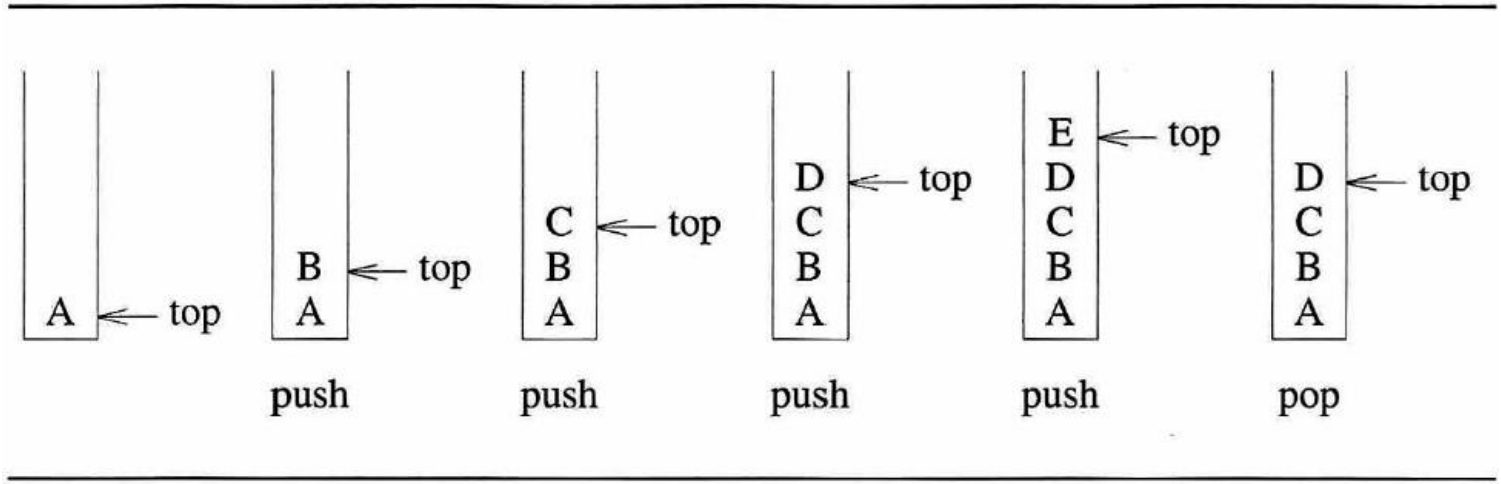


Figure 3.1: Inserting and deleting elements in a stack

- Ex 3.1) [System stack]
  - Used by a program at run-time to process function calls
  - Whenever a function is invoked, the program creates an stack frame (activation record), and places it on top of the system stack
  - When this function terminates, its stack frame is removed
  - Stack frame contains :
    - Pointer to the previous stack frame (of the invoking function)
    - Return address
    - Local variables
    - Parameters

- Ex 3.1) main() invokes *a1*()
  - *fp* (frame pointer) : a pointer to current stack frame
  - System also maintains separately a stack pointer(sp)

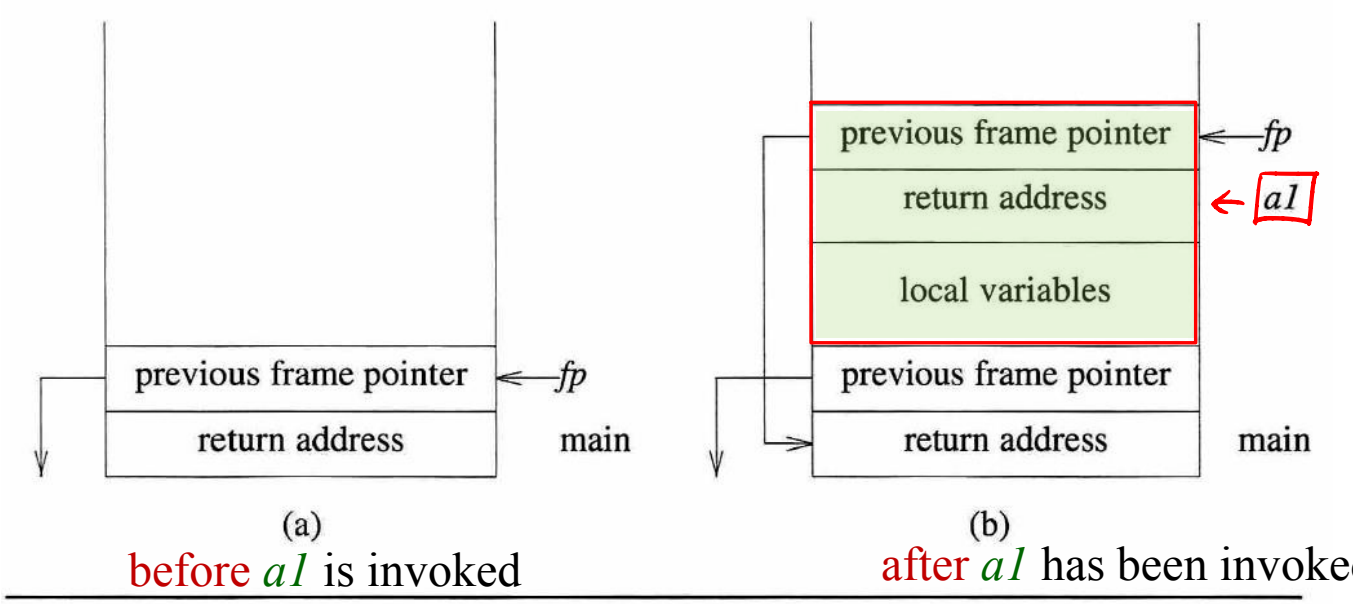


Figure 3.2: System stack after function call

```
#include <stdio.h>
int sum(int n);

int main() {
    int number, result;
    scanf("%d", &number);

    result = sum(number);

    printf("%d", result);
}
int sum(int num) {
    if (num!=0)
        return num + sum(num-1);
    else
        return num;
}
```

What about recursive function calls? …

- Stack Implementation in C
  - Use a 1D array, stack [MAX-STACK-SIZE]
  - 1st element of the stack is stored in ..., 2nd in ..., and i^{th} in ...
  - A variable, ***top*** points to the top element in the stack
  - Initially, *top* = **-1** (empty stack)

ADT *Stack* is

objects: a finite ordered list with zero or more elements.

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer

Stack CreateS(*maxStackSize*) ::=
create an empty stack whose maximum size is *maxStackSize*

Boolean IsFull(*stack*, *maxStackSize*) ::=
if (number of elements in *stack* == *maxStackSize*)
return *TRUE*
else return *FALSE*

Stack Push(*stack*, *item*) ::=
if (IsFull(*stack*)) *stackFull* → 스택이 다 찼는지 안 찼는지 확인
else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=
if (*stack* == CreateS(*maxStackSize*))
return *TRUE*
else return *FALSE*

Element Pop(*stack*) ::=
if (IsEmpty(*stack*)) **return** → 스택에 데이터가 있는지 없는지 확인
else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

- Implementation of Stack Operations:

```
Stack CreateS(maxStackSize) ::=
        #define MAX_STACK_SIZE 100  /* maximum stack size*/
        typedef struct {
            int key;
            /* other fields */
        } element;
        element  stack[MAX_STACK_SIZE];
        int top = -1;

Boolean IsEmpty(Stack) ::= top < 0;

Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE-1;
```

스택의 각 원소를 구조체로 구현

top<0 이면 true 반환

index이기 때문에 -1을 해주어야함

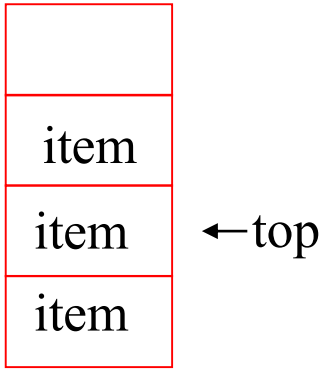
ex) stack size가 100이면 top(배열의 끝)은 99

```
void push(element item)
{/* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1)
        stackFull();
    stack[++top] = item;
}
```

++하고 저장! ⇒ top이 가리키는 것은 저장된 element의 index

Program 3.1: Add an item to a stack

Stack::push



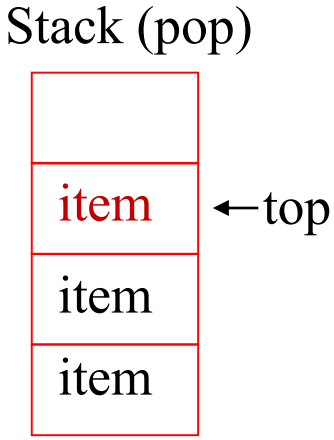

```
void stackFull()
{
    fprintf(stderr, "Stack is full, cannot add element");
    exit(EXIT_FAILURE);
}
```

Program 3.3: Stack full

```
element pop()
{/* delete and return the top element from the stack */
    if (top == -1)
        return stackEmpty();  /* returns an error key */
    return stack[top--];
}
```

top을 리턴하고 top이 1 감소

Program 3.2: Delete from a stack



STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

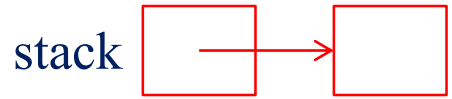
3.5 A Mazing Problem

3.6 Evaluation of Expressions

- A drawback of the previous stack implementation: …

- Implementation using dynamic arrays

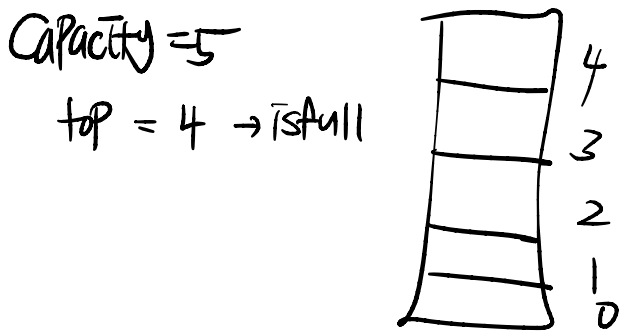
```
Stack CreateS() ::= typedef struct {
                        int key;
                        /* other fields */
                    } element;
                    element *stack;
                    MALLOC(stack, sizeof(*stack));
                    int capacity = 1;
                    int top = -1;

Boolean IsEmpty(Stack) ::= top < 0;

Boolean isFull(Stack) ::= top >= capacity-1;
```



이 부분에의 강점
: stack이 element가 아니라도
같은 방식이라면도 그대로 쓸 수 있다

- pop: unchanged

```
element pop()
{/* delete and return the top element from the stack */
    if (top == -1)
        return stackEmpty();  /* returns an error key */
    return stack[top--];
}
```



- push : changed

```
void push(element item)
{/* add an item to the global stack */
    if ( top >= capacity-1 )
        stackFull();
    stack[++top] = item;
}
```

- stackFull : changed

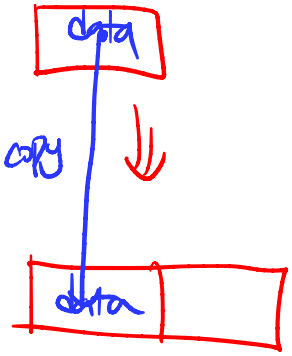
```
void stackFull()
{
        REALLOC( stack, 2*capacity*sizeof(*stack) );
        capacity *= 2;
}
```

Program 3.4: Stack full with array doubling

```
void stackFull()  {
      REALLOC(stack, 2*capacity*sizeof(*stack))
      capacity *= 2;
}
```

- Time Complexity of Array Doubling
  - One array doubling : O(capacity)
    - Memory allocation : O(1)
    - Copy of all array elements : O(capacity)

realloc 할때 필요



  - When capacity = 2^k
    - k array doublings : $O(\sum_{i=1}^{k} 2^i) = O(2^{k+1}) = O(2^k)$

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

3.5 A Mazing Problem

3.6 Evaluation of Expressions

- Queue
  - An ordered list
    - Insertions and deletions take place at different ends
  - First-In-First-Out (FIFO) lists
  - **Insertions** take place, called the *rear*
  - **Deletions** take place at the opposite end, called the *front*

삽입과 삭제하는 쪽이 다름!

ex) 프린트 할때

삽입이 일어나는 쪽 = rear

삭제가 " = front

- Ex)
  - Insert *A, B,* C, *D,* and *E*
  - Then, *A* is the first element deleted from the queue

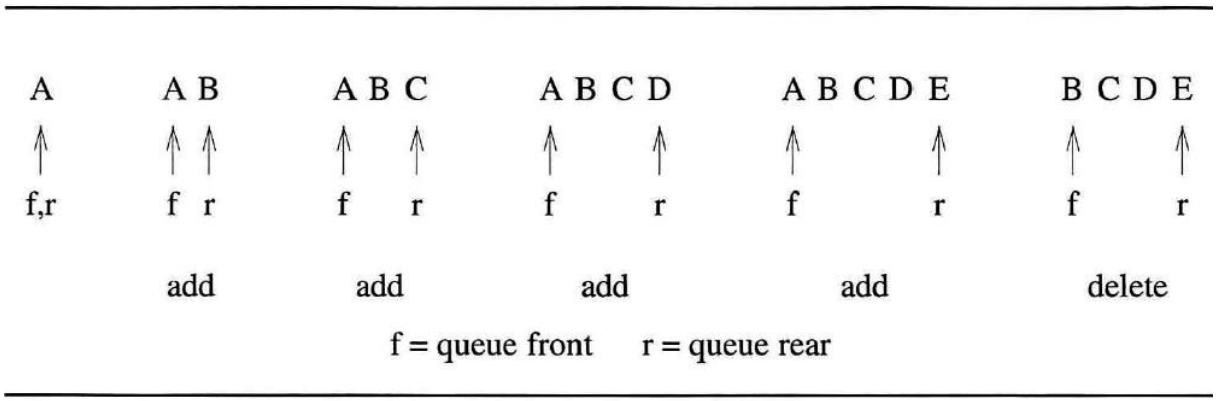


Figure 3.4: Inserting and deleting elements in a queue

ADT *Queue* is

objects: a finite ordered list with zero or more elements.

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer

Queue CreateQ(*maxQueueSize*) ::=
create an empty queue whose maximum size is *maxQueueSize*

Boolean IsFullQ(*queue*, *maxQueueSize*) ::=
if (number of elements in *queue* == *maxQueueSize*)
return *TRUE*
else return *FALSE*

Queue AddQ(*queue*, *item*) ::=
if (IsFullQ(*queue*)) *queueFull*
else insert *item* at rear of *queue* and return *queue*

Boolean IsEmptyQ(*queue*) ::=
if (*queue* == CreateQ(*maxQueueSize*))
return *TRUE*
else return *FALSE*

Element DeleteQ(*queue*) ::=
if (IsEmptyQ(*queue*)) **return**
else remove and return the *item* at front of queue.

ADT 3.2: Abstract data type *Queue*

- Queue
  - Representation I:
    Using a 1D array and two variables, front and rear
  - Representation II:
    Regard an array as a circular queue;
    more efficient

- Representation I
  - Using a 1D array and two variables, front and rear

```
Queue CreateQ(maxQueueSize) ::=
        #define MAX_QUEUE_SIZE 100 /* maximum queue size */
        typedef struct {
            int key;
            /* other fields */
        } element;
        element queue[MAX_QUEUE_SIZE];
        int rear = -1;
        int front = -1;

Boolean IsEmptyQ(queue) ::=  front == rear

Boolean IsFullQ(queue) ::=  rear == MAX_QUEUE_SIZE-1
```

```
void addq(element item)
{/* add an item to the queue */
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull();
    queue[++rear] = item;
}
```

Stack에서 Push랑 같음
(⇒ ++ 하고 저장)

Program 3.5: Add to a queue

```
element deleteq()
{/* remove element at the front of the queue */
    if (front == rear)
        return queueEmpty();  /* returns an error key */
    return queue[++front];
}
```

Program 3.6: Delete from a queue

Function Calls:

```
addq (item);
…
item = deleteq ();
```

- Ex 3.2 [Job scheduling]:

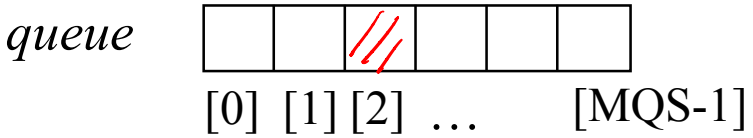
front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
−1	0	J1				Job 1 is added
−1	1	J1	J2			Job 2 is added
−1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure 3.5: Insertion and deletion from a sequential queue

- Problems :
  - Queue gradually shifts to right 오른쪽으로 계속 쉬프트됨
    - rear index equals MAX_QUEUE_SIZE–1 (queue is full) 사실은 안 찼는데 찬거처럼 인식..
  - Should move the entire queue to the left (array shifting)
    - 1st element → Q[0], front → - 1, rear → …
    - Time-consuming: O(MAX_QUEUE_SIZE)

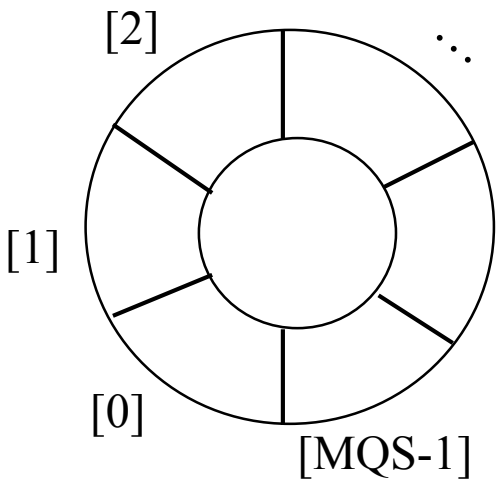
⇒ 다시 왼쪽으로 쉬프트 시켜주어야함

- Representation II: Circular Queue
  - Using 1D array



처음과 끝이 붙어있는 형태로 생각하자

  - Circular view of array:
    Queue wrapped around the end of the array



Initial values :
front = rear = 0

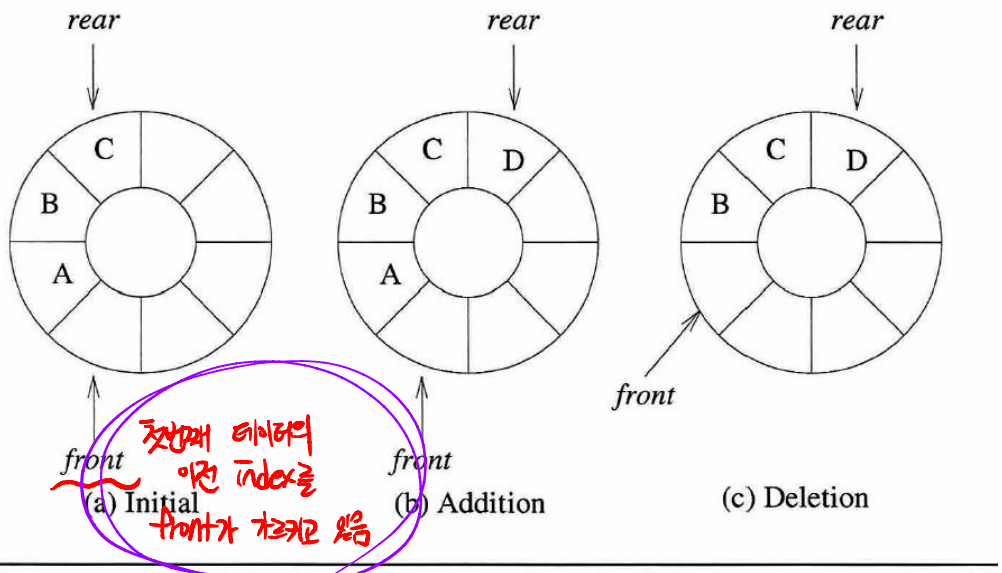
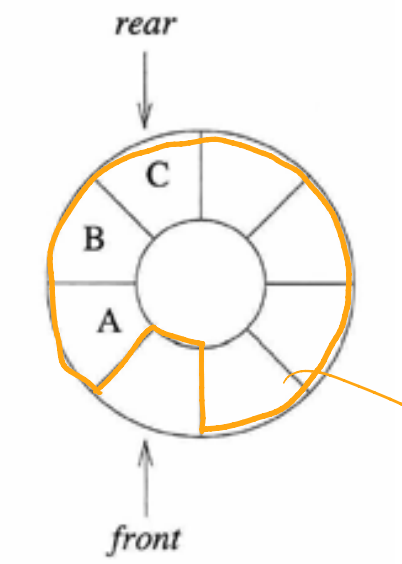


Figure 3.6: Circular queue

★ 이 페이지를 이해 잘 해보자!



If we do 5 additions(D, E, F, G, H) : …

front보다 rear가 같아짐

If we perform 3 deletions : …

front보다 rear가 같아짐

이만큼만 쓰자

Problem: … full일때랑 empty일때 구분이안감

→ *front*== *rear* iff the queue is empty ★ empty로 정하자

↳ 데이터의 개수

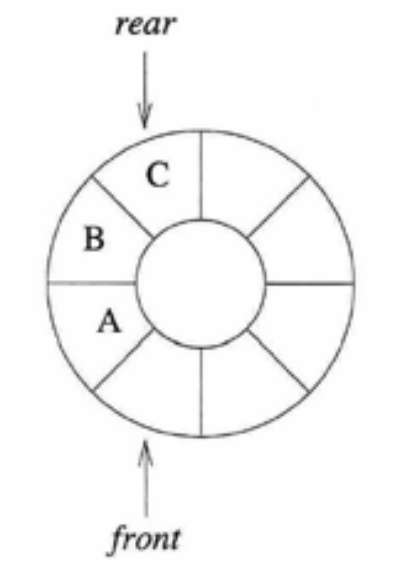
→ max capacity: MAX_QUEUE_SIZE-1

↳ 그리고 전체 데이터 중 한칸을 비워놓자

- 원형 큐가 텅 빈 상태 — F와 R이 동일한 위치를 가리킨다.
- 원형 큐가 꽉 찬 상태 — R이 가리키는 위치의 앞을 F가 가리킨다.

"enqueue 연산 시, R이 가리키는 위치를 한 칸 이동시킨 다음에, R이 가리키는 위치에 데이터를 저장한다."

"dequeue 연산 시, F가 가리키는 위치를 한 칸 이동시킨 다음에, F가 가리키는 위치에 저장된 데이터를 반환 및 소멸한다."



rear을 0~MAX_QUEUE_SIZE-1로 정

```
void addq(element item)
{ /* add an item to the queue */
     rear = (rear+1)%MAX_QUEUE_SIZE;
     if (front == rear)
          queueFull();   /* print error and exit */
     queue[rear] = item;
}
```

Program 3. 7: Add to a circular queue

max capacity: MAX_QUEUE_SIZE-1

```
element deleteq()
{ /* remove front element from the queue */
        if (front == rear)
                return queueEmpty();  /* return an error key */
        front = (front+1) % MAX_QUEUE_SIZE;
        return queue[front];
}
```

Program 3.8: Delete from a circular queue

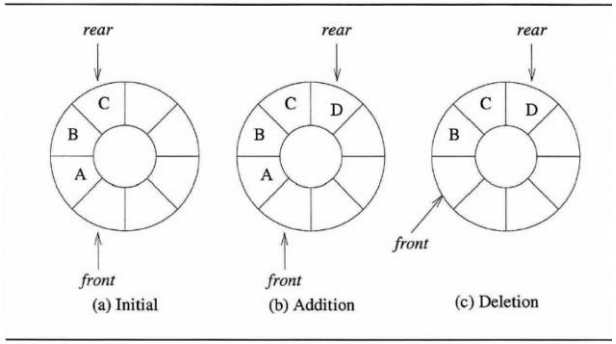


Figure 3.6: Circular queue

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

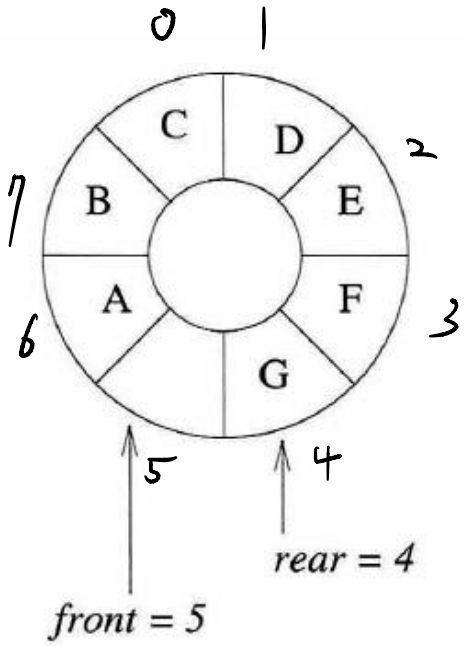
3.4 Circular Queues Using Dynamic Arrays

3.5 A Mazing Problem

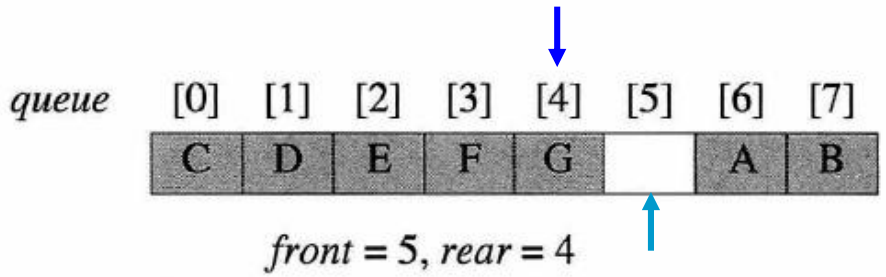
3.6 Evaluation of Expressions

- Suppose that a **dynamically allocated array** is used to hold the queue elements

- To add an element to a full queue
  → increase the size of array: …



(a) A full circular queue



(b) Flattened view of circular full queue

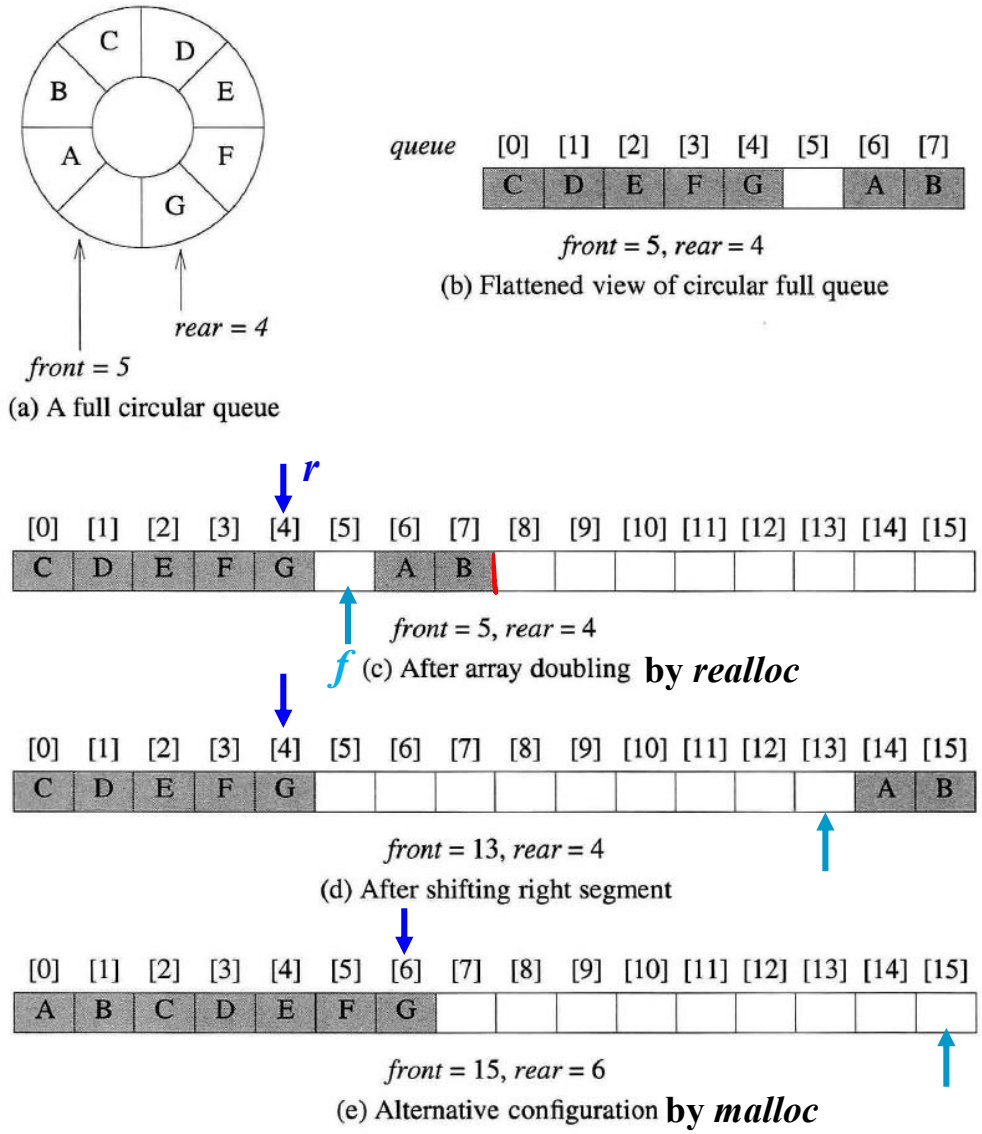
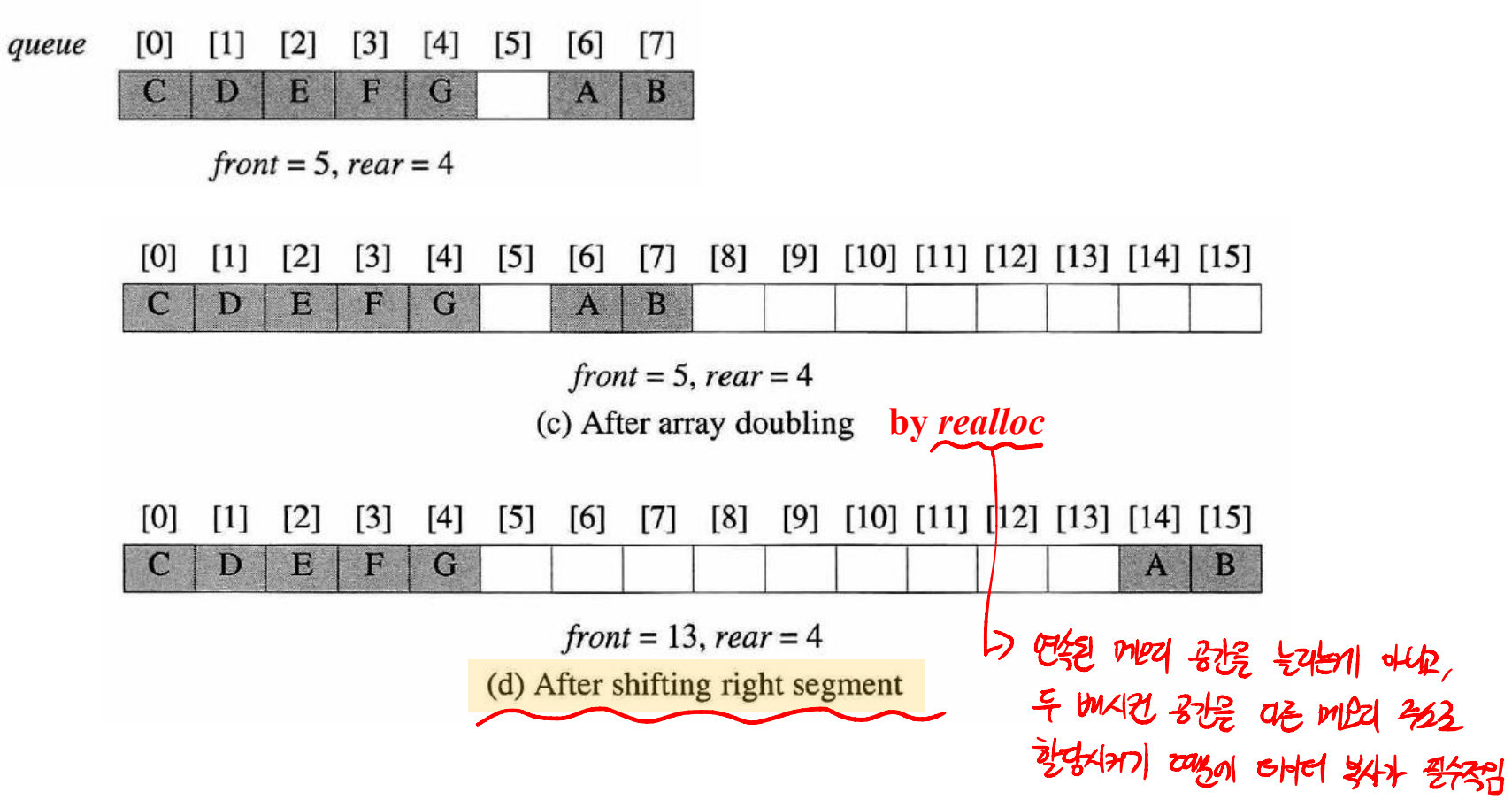


Figure 3.7: Doubling queue capacity

- Case 1: (b)→(c)→(d)

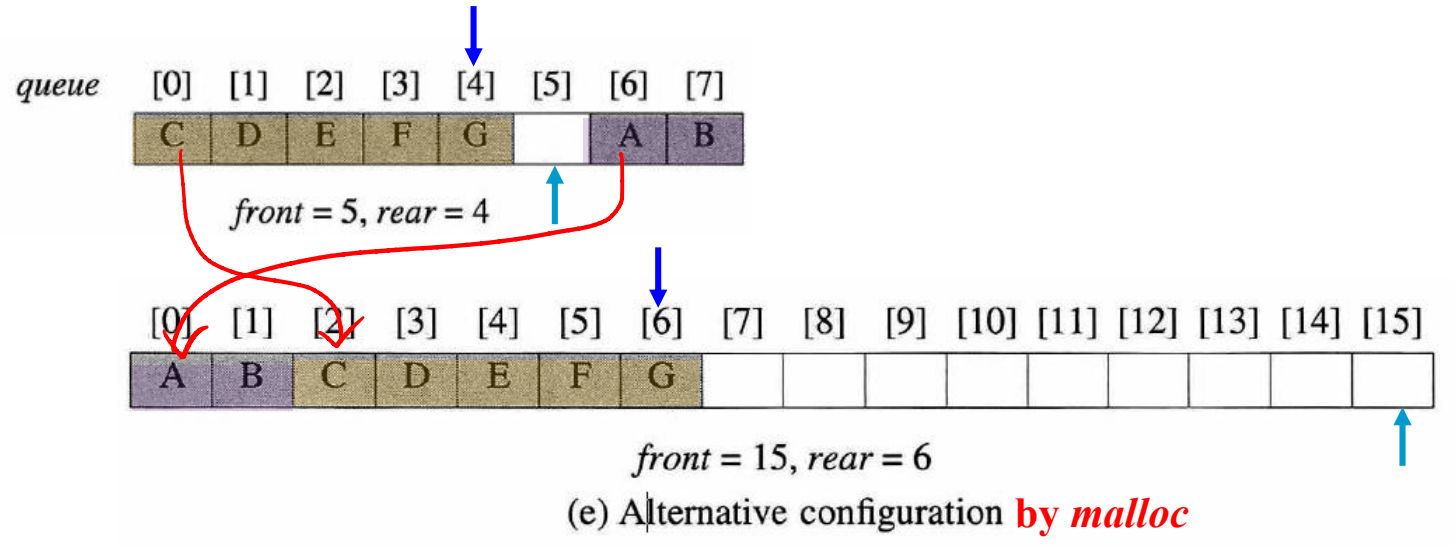


The number of elements copied:

→ at most (2·capacity – 2) elements

$= 2 \times (\text{capacity} - 1)$

- Case 2: (b)→(e)
  - The number of elements copied can be limited to *capacity* - 1



(1) Create a new array *newQueue* of twice the capacity
(2) Copy **the second segment** to positions in newQueue beginning at **0**
(3) Copy **the first segment** to positions in newQueue beginning at **capacity-front-1**

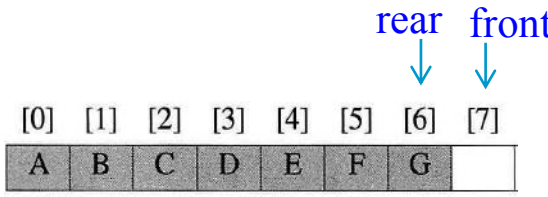
```
void addq(element item)
{
        rear = (rear+1)%MAX_QUEUE_SIZE;
        if (front == rear)
             queueFull();   /* double capacity */
        queue[rear] = item;
}
```

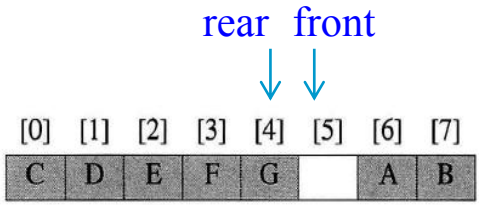
Program 3.9: Add to a circular queue

CAPACITY = MAX_QUEUE_SIZE

```
void queueFull() {
    /* allocate an array with twice the capacity */
    element* newQueue;
    MALLOC(newQueue, 2*capacity*sizeof(*queue));
    /* copy from queue to newQueue */
    int start = (front+1) % capacity;
    if (start < 2)
        /* no wrap around */
        copy(queue+start, queue+start+capacity-1, newQueue);
    else
        /* queue wraps around */
        copy(queue+start, queue+capacity, newQueue);
        copy(queue, queue+rear+1, newQueue+capacity-start);
     }
    /* switch to newQueue */
    front = 2*capacity-1;
    rear = capacity-2;              // queue size = capacity-1
    capacity *= 2;
    free(queue);
    queue = newQueue;
}
```

→ 데이터가 순차적으로 될 경우 (circular가 안될경우)





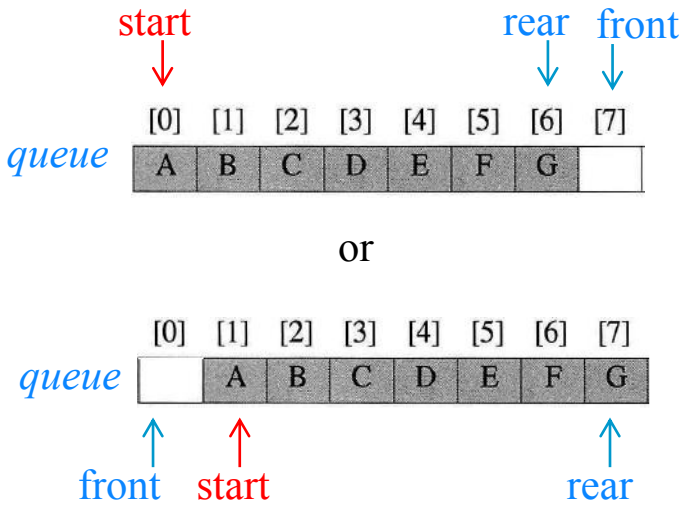
Program 3.10: Doubling queue capacity

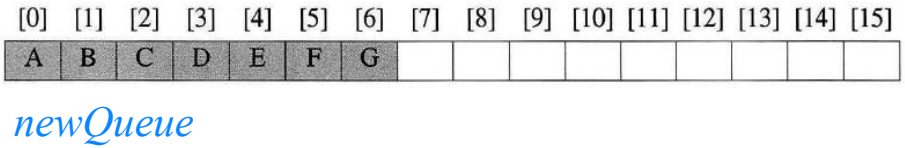
copy (a,b,c);
// copies elements from locations **a** through **b-1** to locations beginning at **c**

index a부터 b-1까지 다 copy

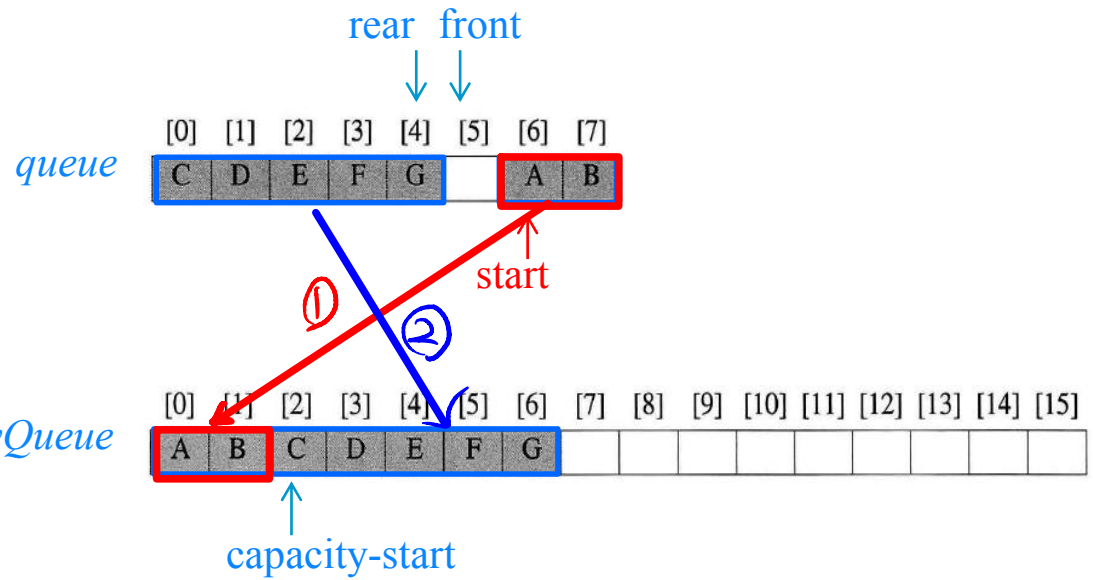
↳8

```
int start = (front+1) % capacity;
if (start < 2)
    /* no wrap around */
    copy(queue+start, queue+start+capacity-1, newQueue);
```





```
int start = (front+1) % capacity;
if (start < 2)
      /* no wrap around */
      copy(queue+start, queue+start+capacity-1, newQueue);
else
      /* queue wraps around */
      copy(queue+start, queue+capacity, newQueue);
      copy(queue, queue+rear+1, newQueue+capacity-start);
}
```

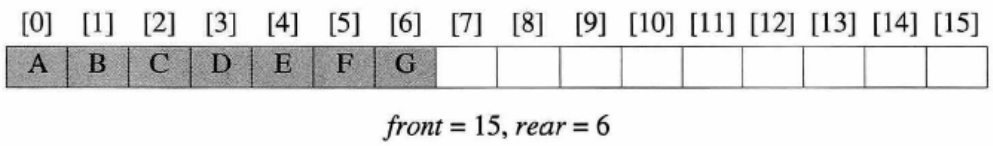


```
void queueFull() {
    /* allocate an array with twice the capacity */
    element* newQueue;
    MALLOC(newQueue, 2*capacity*sizeof(*queue));
    /* copy from queue to newQueue */
    int start = (front+1) % capacity;
    if (start < 2)
        /* no wrap around */
        copy(queue+start, queue+start+capacity-1, newQueue);
    else
        /* queue wraps around */
        copy(queue+start, queue+capacity, newQueue);
        copy(queue, queue+rear+1, newQueue+capacity-start);
    }
    /* switch to newQueue */
    front = 2*capacity-1;
    rear = capacity-2;
    capacity *= 2;
    free(queue);
    queue = newQueue;
}
```

capacity = 원래크기 +1

front = 2*capacity-1; → 전방에 들어온 방의 앞 index (여기서는 15!)

rear = capacity-2; → 최근에 들어온 방의 Index // queue size = capacity-1



Program 3.10: Doubling queue capacity

STACKS AND QUEUES