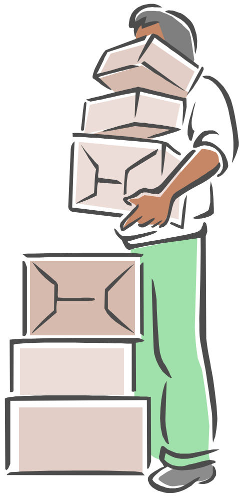
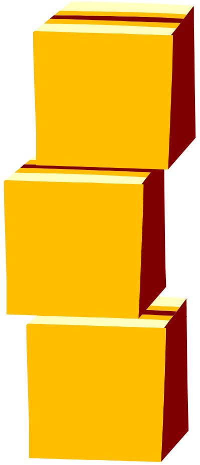
Chap 3. Stacks and Queues (1)

Contents

3. Stacks

3.1 Stacks





3.1 Stacks

- ***Linear list*.**
- **One end is called *top*.**
- **The other end is called *bottom*.**
- **Additions to and removals from the *top* end only.**

3.1 Stacks

- **A stack is a LIFO list.**
  - *Last-In-First-Out*

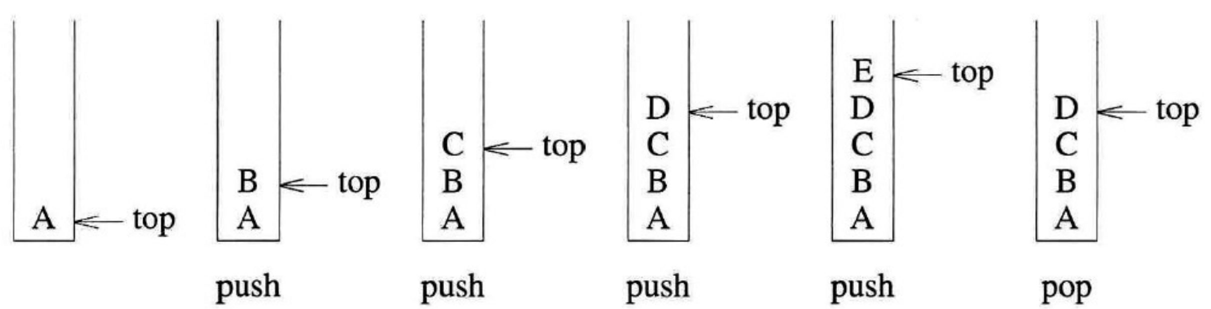


Figure 3.1: Inserting and deleting elements in a stack

3.1 Stacks

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*
        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**
        else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

3.1 Stacks

Creation of Stack in C

- Use a *1D array* to represent a stack.
- Stack elements are stored in *stack[0] through stack[top]*.

```
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;
```

3.1 Stacks

❖ Implementation of Stack Operations

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

Program 3.1: Add an item to a stack

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

Program 3.2: Delete from a stack

Implementation of Stack Operations

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

Program 3.3: Stack full

3.2 Stacks Using Dynamic Arrays

Stack CreateS() ::= typedef struct {

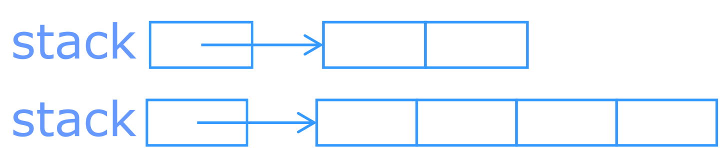
```
int key;
/ * other fields * /
} element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity = 1;
int top = -1;
```

- **Boolean IsEmpty(Stack) ::= top < 0;**
- **BooleanisFull(Stack)::= top>= capacity-1;**
- **capacity : maximum number of stack elements that may be stored in the array**

3.2 Stacks Using Dynamic Arrays

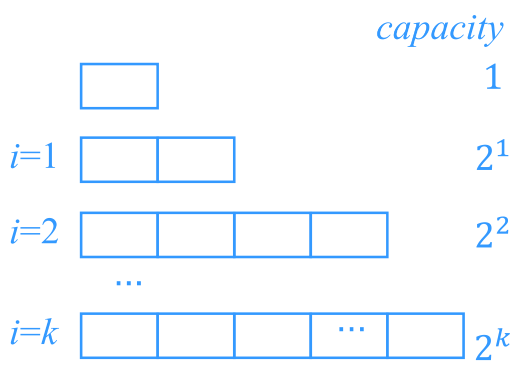
- *pop* : unchanged from Program 3.2
- *push, stackFull*: changed from Program 3.1&3.3
- ***Array Doubling***
  - When stack is full, double the capacity using REALLOC.

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



3.2 Stacks Using Dynamic Arrays

❖ **Time complexity of array doubling**

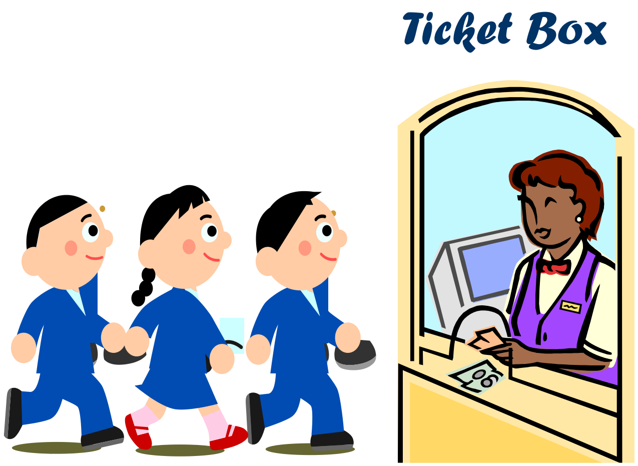


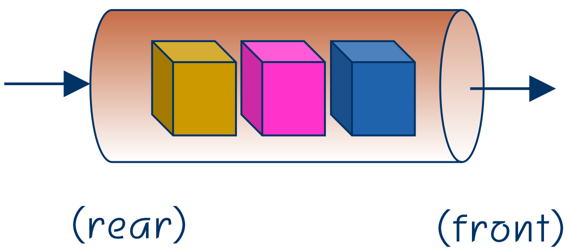
- One array doubling
  - Memory allocation : O(1)
  - Copy of an array element : O(1)
  - Copy of all array elements : O(capacity)
- All array doubling
  - *N 번의 push가 있었고 현재 stack capacity가* 2^k *이라면 (k번 doubling)*
  - $O\left(\sum_{i=1}^{k} 2^i\right) = O\left(2^{k+1}\right) = O\left(2^k\right)$

Doubling 시 copy수

What is difference between stack and Array?

3.3 Queues





3.3 Queues

- ***Linear list*.**
- **One end is called *front*.**
- **The other end is called *rear*.**
- ***Additions* are done at the *rear* only.**
- ***Removals* are made from the *front* only.**

3.3 Queues

- **A queue is a FIFO list.**
  - *First-In-First-Out*

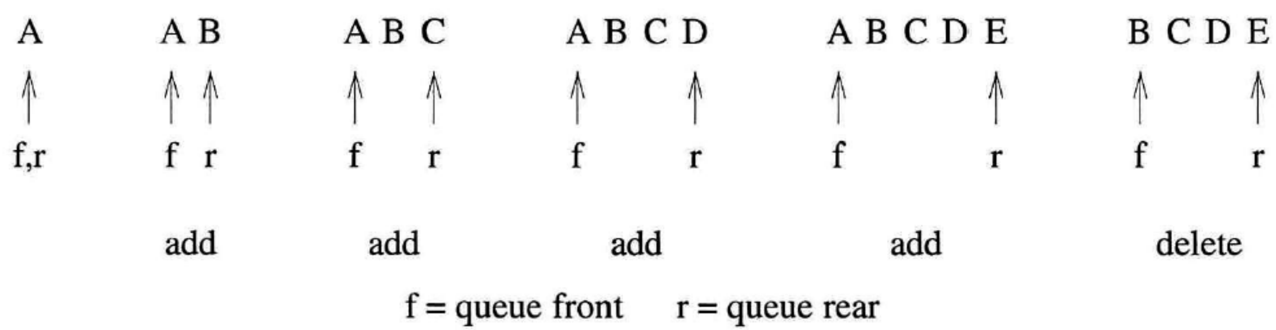


Figure 3.4: Inserting and deleting elements in a queue

3.3 Queues

ADT *Queue* is
objects: a finite ordered list with zero or more elements.
functions:
for all $queue \in Queue$, $item \in element$, $maxQueueSize \in$ 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.

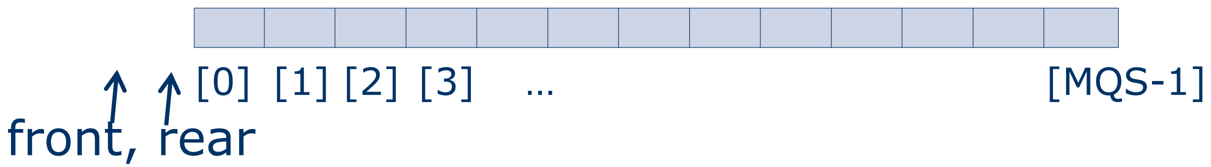
ADT 3.2: Abstract data type *Queue*

Representations of Queue

- **Sequential representation**
  - Uses an *1D array*
- **Circular representation : *circular queue***
  - Uses an *1D array*
  - More efficient

Sequential Representation

- **Creation of Queue in C**
  - Uses an 1D array, *queue*



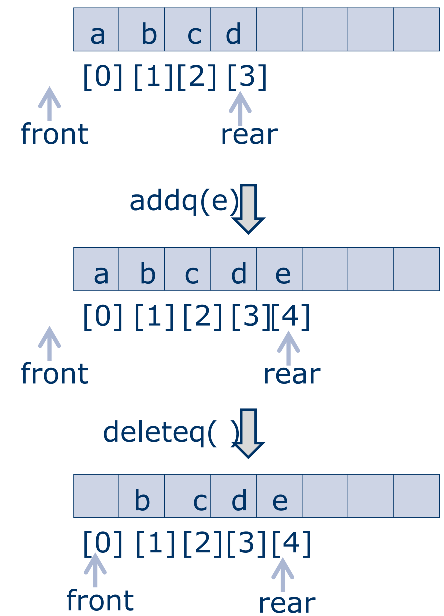
- ***Queue CreateQ(maxQueueSize)* ::=**
- **#define MAX-QUEUE-SIZE 100 /* maximum queue size */**
- **typedef struct {**
- **int key;**
- **/* other fields */**
- **} element;**
- **element queue[MAX-QUEUE-SIZE];**
- **int rear = -1;**
- **int front = -1;**
- ***Boolean IsEmptyQ(queue)* ::= front == rear**
- ***Boolean IsFullQ(queue)* ::= rear == MAX-QUEUE-SIZE-1**

Sequential Representation

❖ Implementation of Queue Operations

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

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



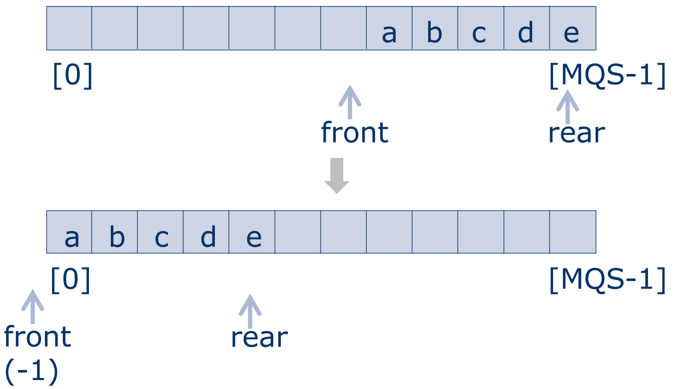
Sequential Representation

❖ Example: Job Scheduling by an OS

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

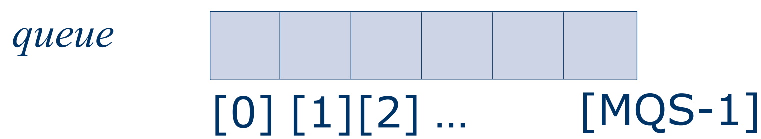
Figure 3.5: Insertion and deletion from a sequential queue

- queueFull
  - array shifting : time-consuming
  - Worst case time complexity, *O(MAX_QUEUE_SIZE)*

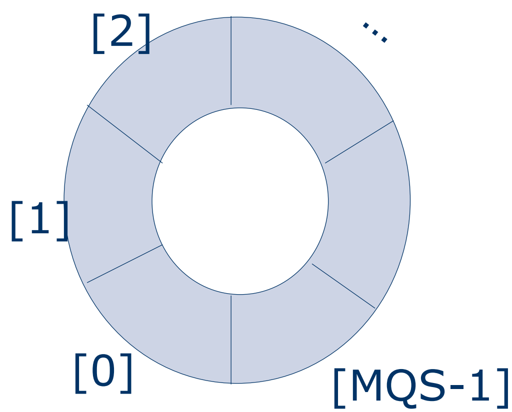


Circular Queue

- Uses an 1D array, ***queue***



- Circular view of an 1D array

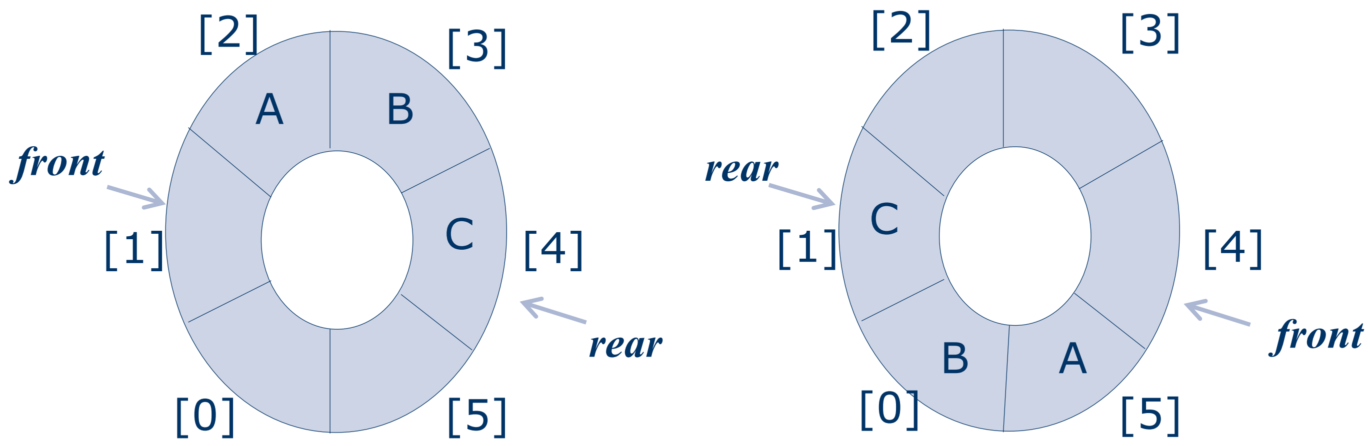


Initial values

$front = rear = 0$

Circular Queue

- **integer variables *front* and *rear*.**
  - *front* is *one position counterclockwise from first element*
  - *rear* gives position of last element



Circular Queue

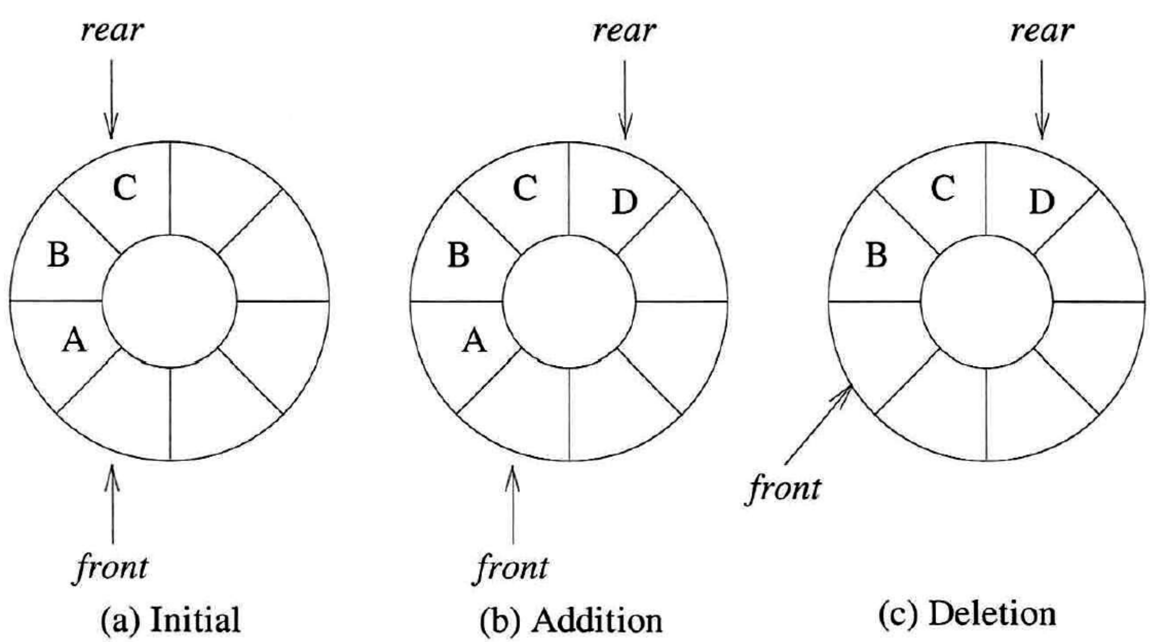
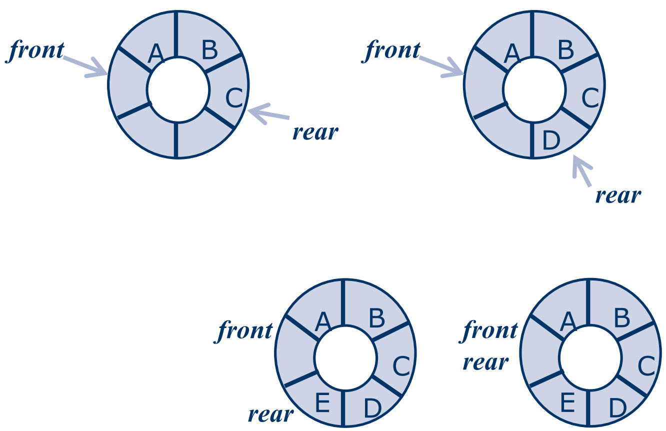


Figure 3.6: Circular queue

Circular Queue

- **Add an element in the circular queue.**
  - Move *rear* one clockwise.
  - Queue Full Check
  - Then put into *queue*[*rear*].



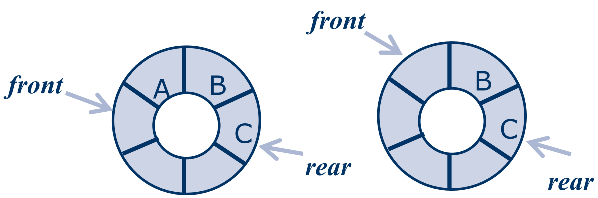

```
void addq(element item)
{/* add an item to the queue */
        rear = (rear+l) % MAX-QUEUE-SIZE;
        if (front == rear)
                queueFull(); /*print error and exit*/
        queue[rear] =item;
}
```

- **a maximum of MAX_QUEUE_SIZE-1 elements in the queue at any time!**

Circular Queue

❖ Delete an element from the circular queue.

- Queue Empty check
- Move *front* one clockwise.
- Then extract from *queue[front]*.




```
element deleteq()
{/* remove front element from the queue */
    element item;
    if (front == rear)
        return queueEmpty(); /*return an error key*/
    front = (front+l) % MAX-QUEUE-SIZE;
    return queue[front];
}
```