Chap 3. Stacks and Queues (1)

Contents

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

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.

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

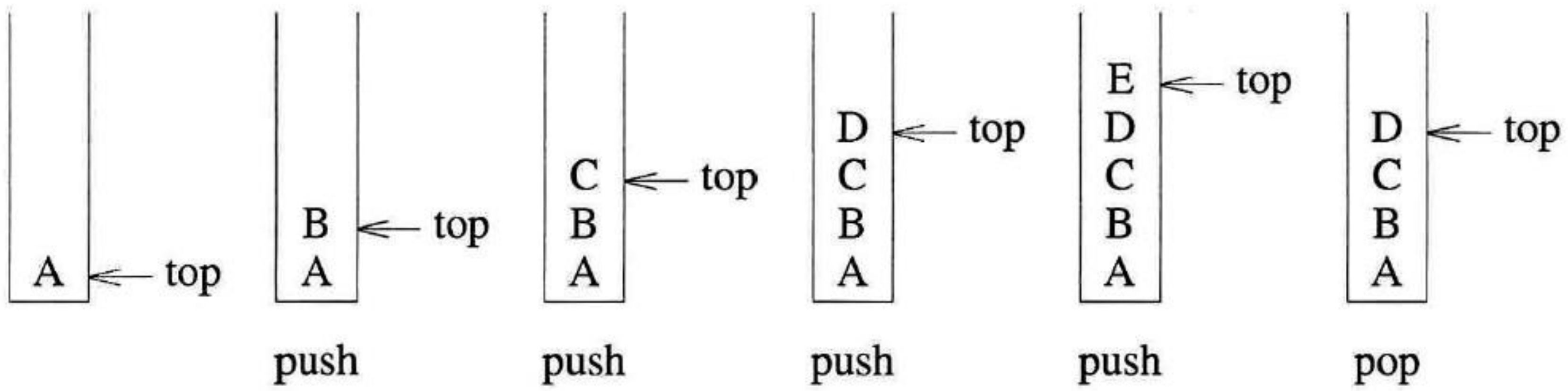


Figure 3.1: Inserting and deleting elements in a stack

Example : System Stack

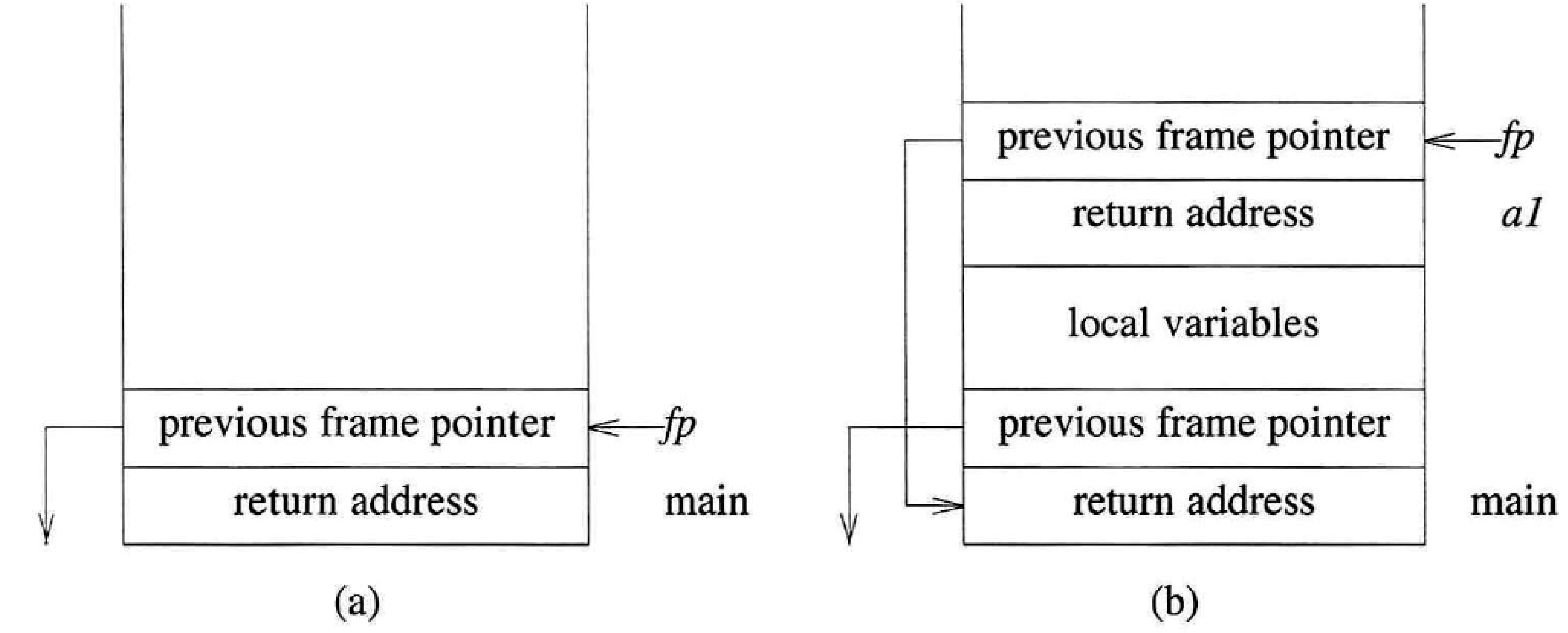
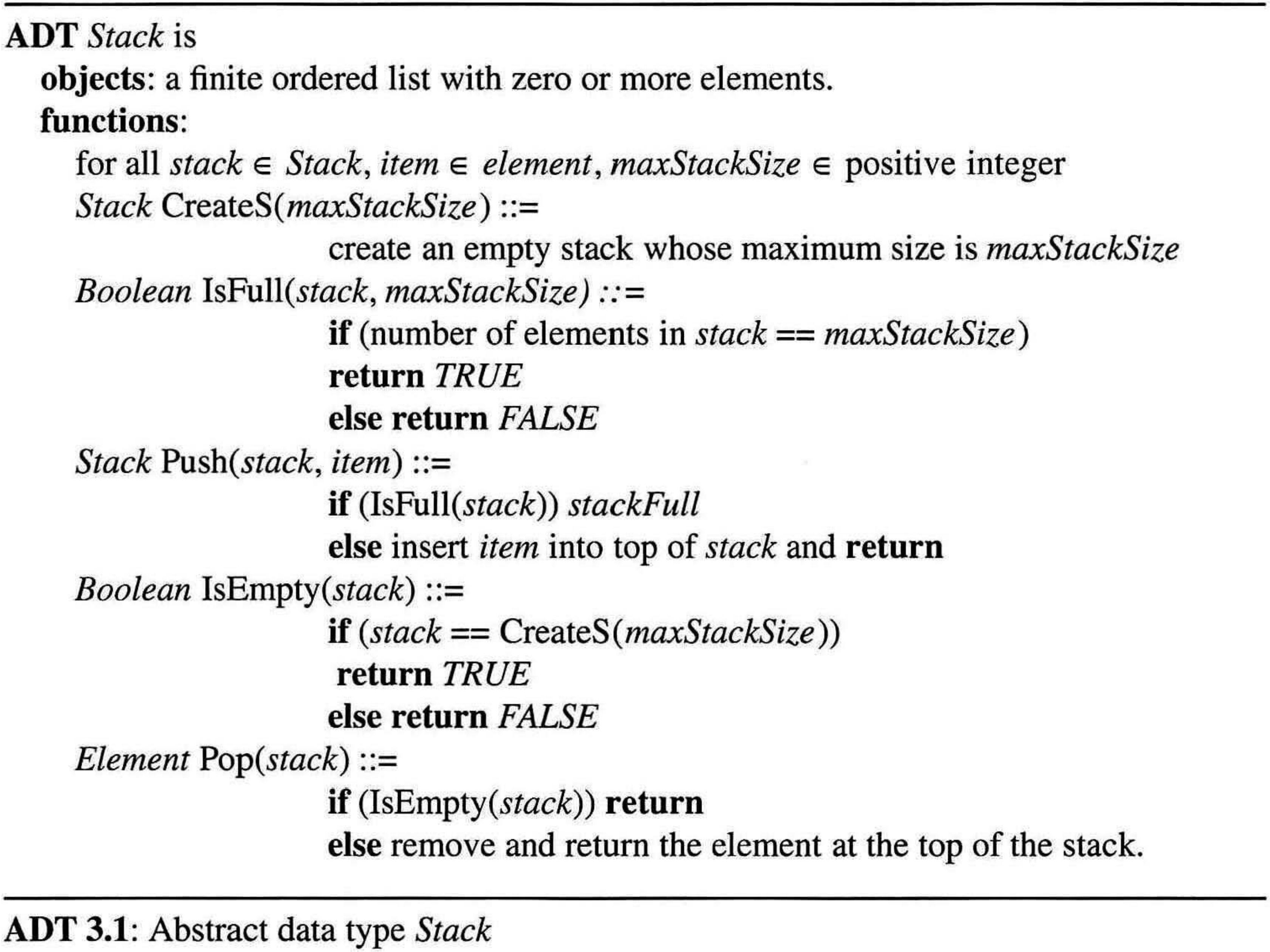


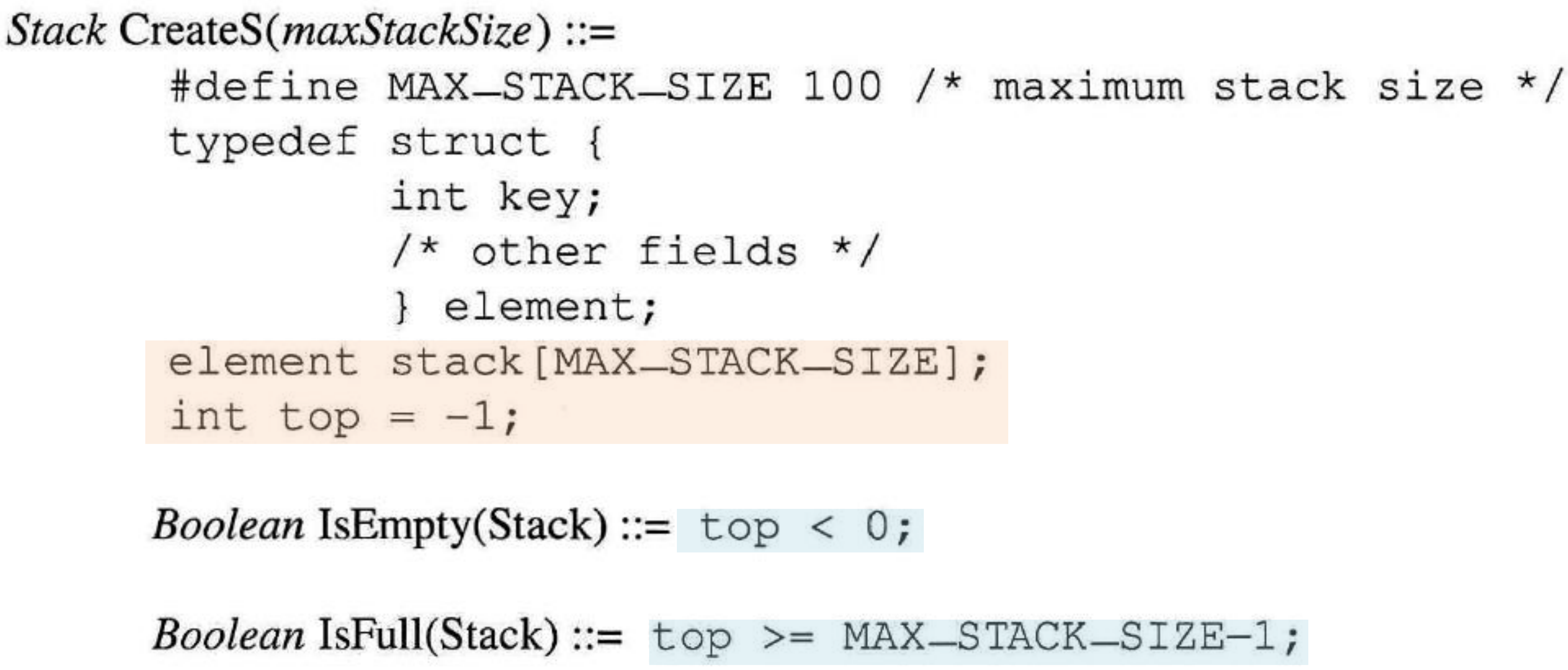
Figure 3.2: System stack after function call



```
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*

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



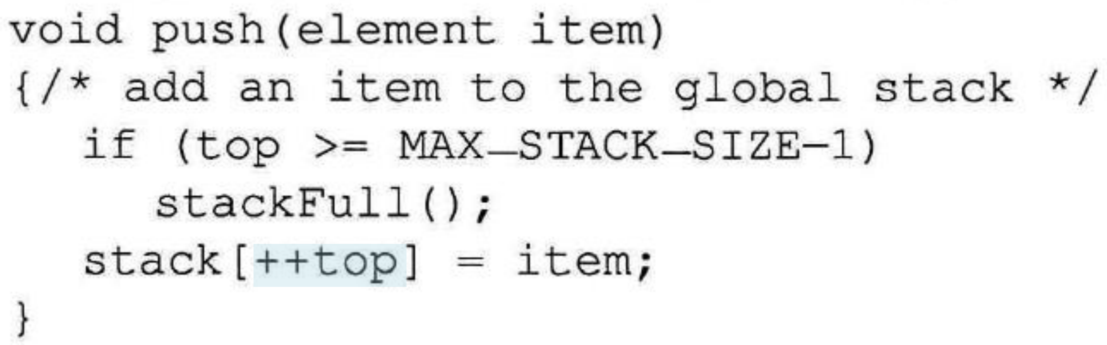
Stack CreateS(*maxStackSize*) ::=

```
#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;`

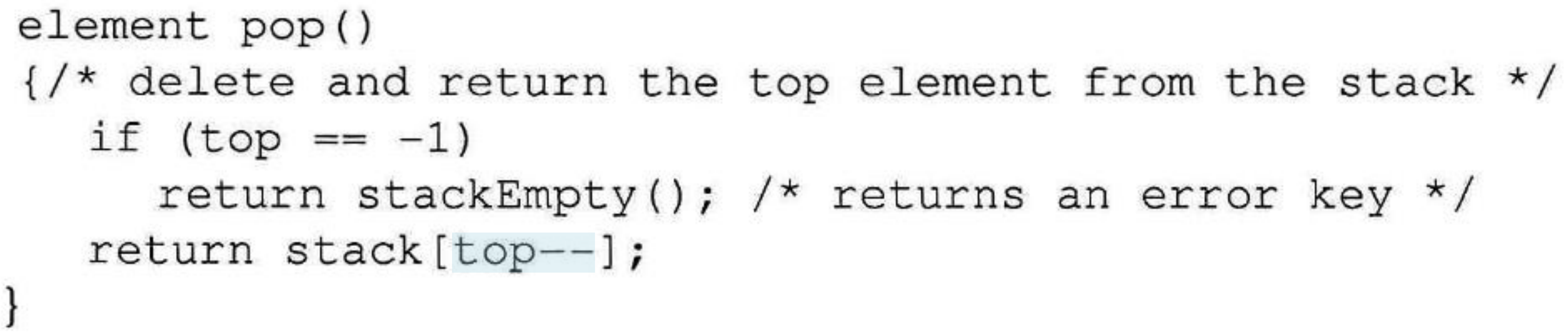
Boolean IsFull(Stack) ::= `top >= MAX_STACK_SIZE-1;`

- Implementation of Stack Operations



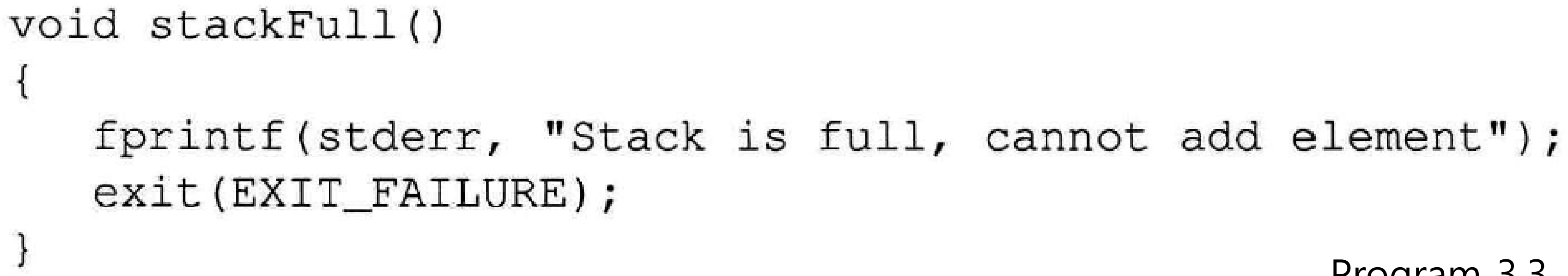
```c
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE-1)
      stackFull();
   stack[++top] = item;
}
```

Program 3.1



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

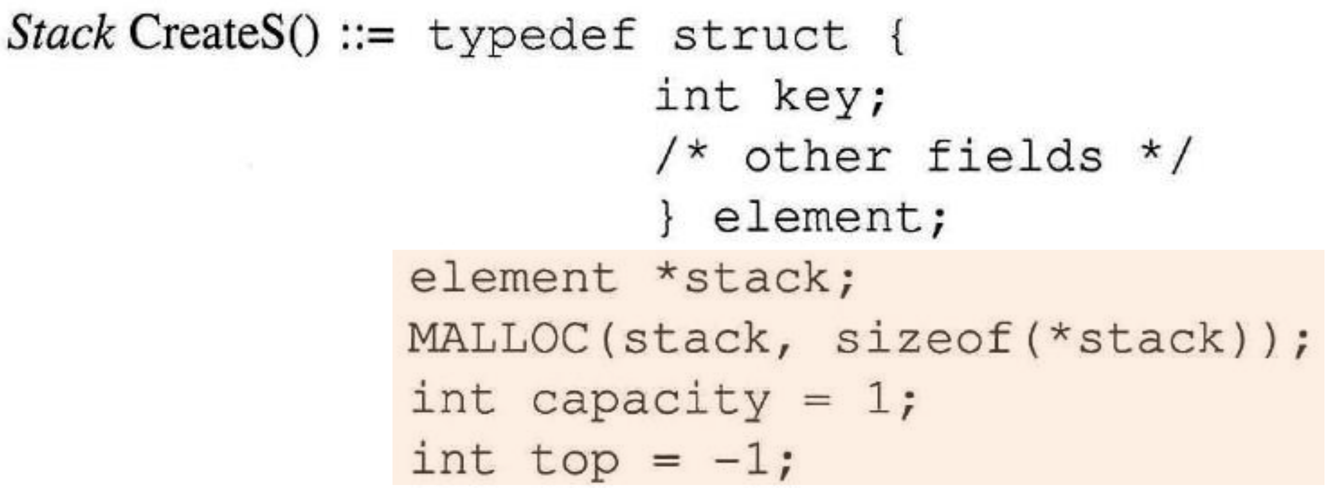
Program 3.2



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

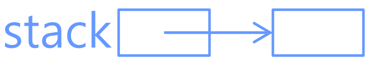
Program 3.3

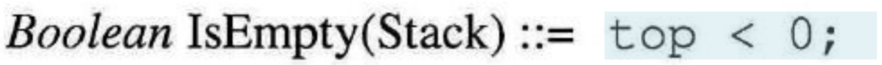
3.2 Stacks Using Dynamic Arrays



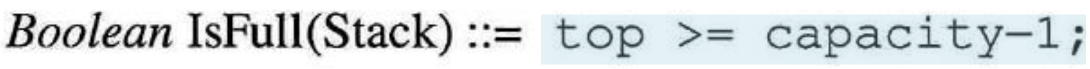
```
Stack CreateS() ::= typedef struct {
                             int key;
                             /* other fields */
                             } element;
                    element *stack;
                    MALLOC(stack, sizeof(*stack));
                    int capacity = 1;
        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

※ **capacity** : maximum number of stack elements that may be stored in the array



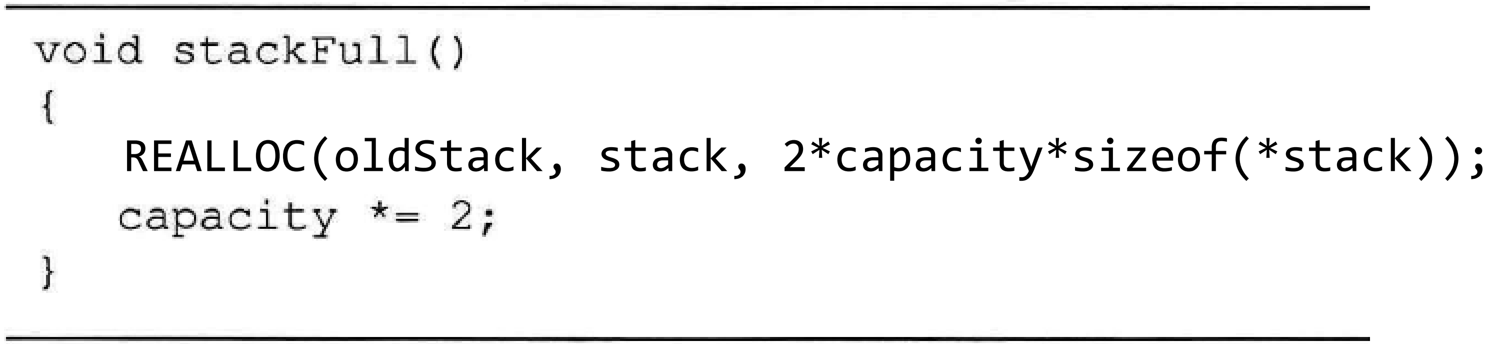


```
Boolean IsEmpty(Stack) ::= top < 0;
```



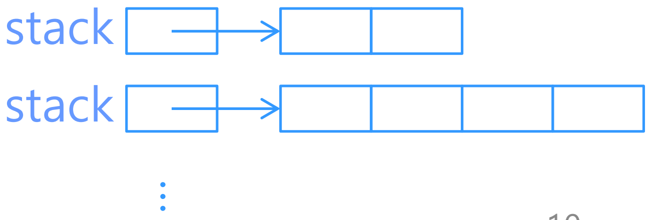
```
Boolean IsFull(Stack) ::= top >= capacity-1;
```

- *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(oldStack, stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

Program 3.4: Stack full with array doubling



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.

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

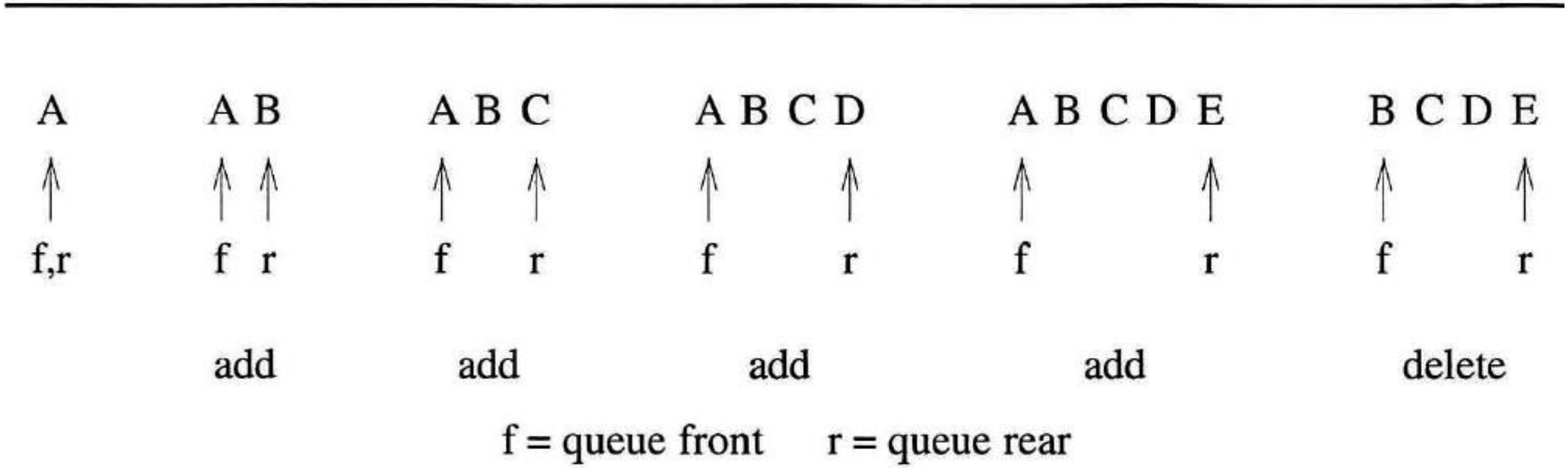
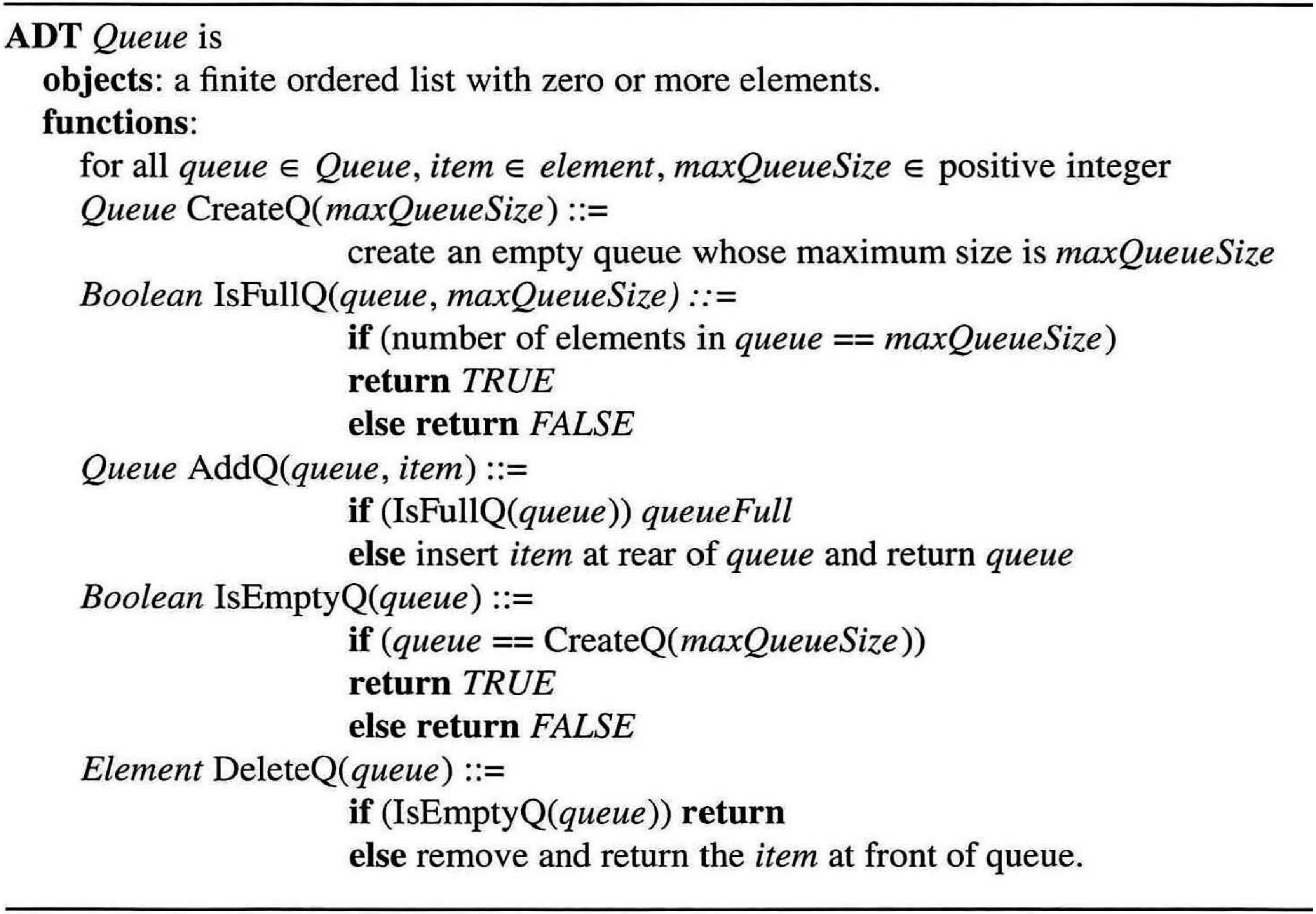


Figure 3.4: Inserting and deleting elements in a queue



```
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.
```

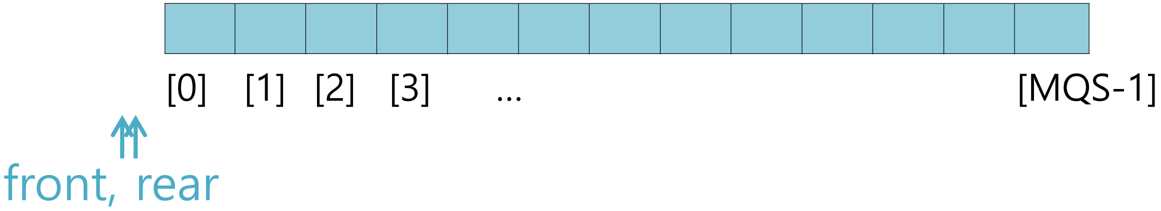
ADT 3.2: Abstract data type *Queue*

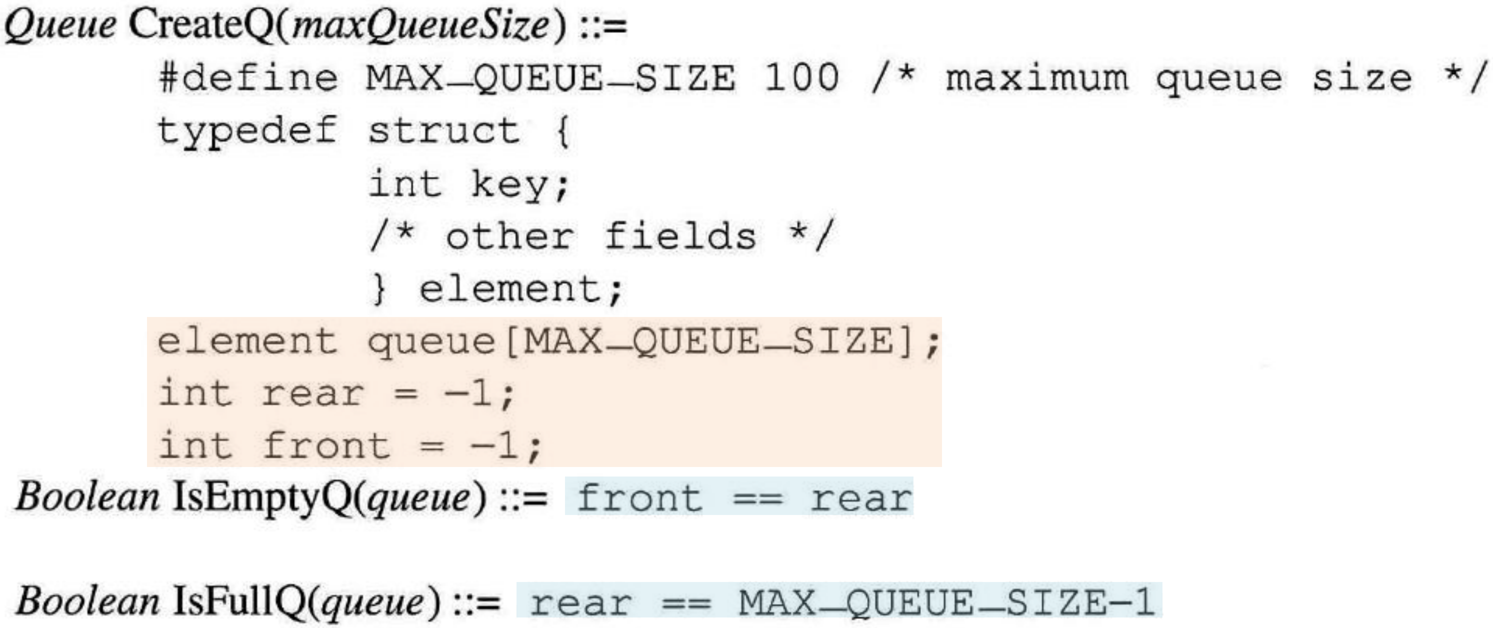
Representations of Queue

- 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*) ::=

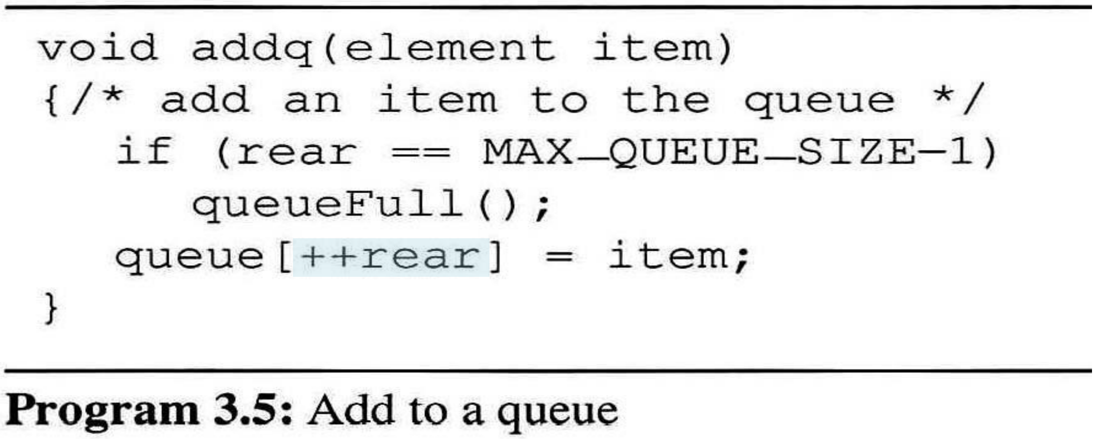
```
#define MAX_QUEUE_SIZE 100 /* maximum queue size */
typedef struct {
        int key;
        /* other fields */
        } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

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

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

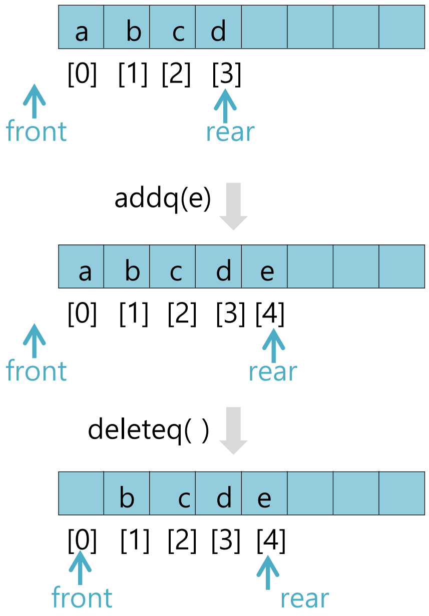
Sequential Representation

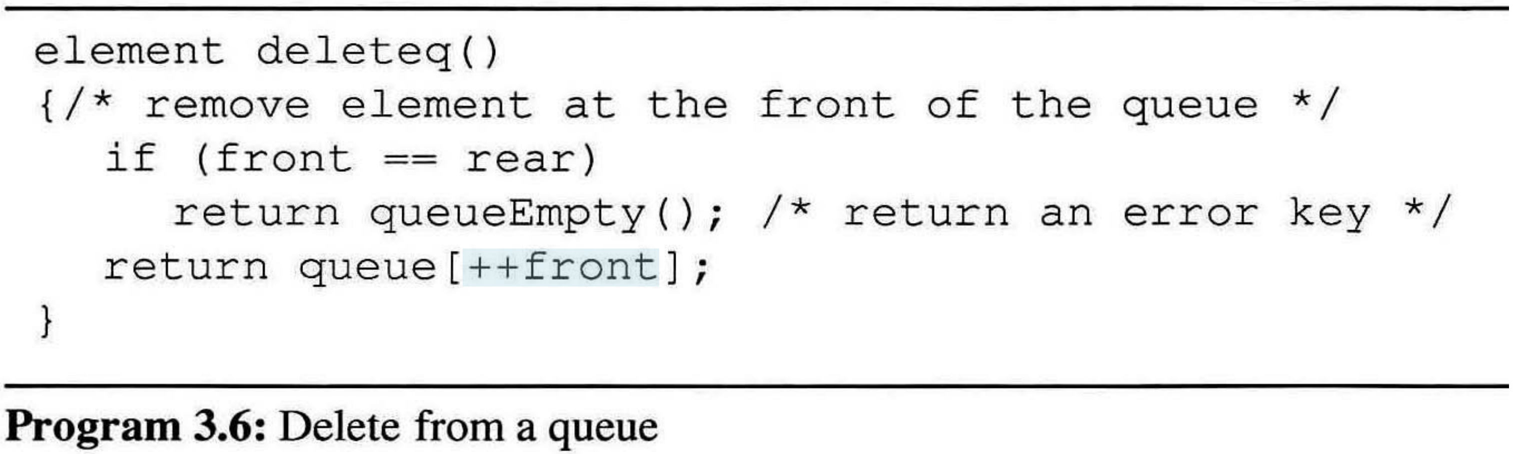
- Implementation of Queue Operations



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

Program 3.5: Add to a queue



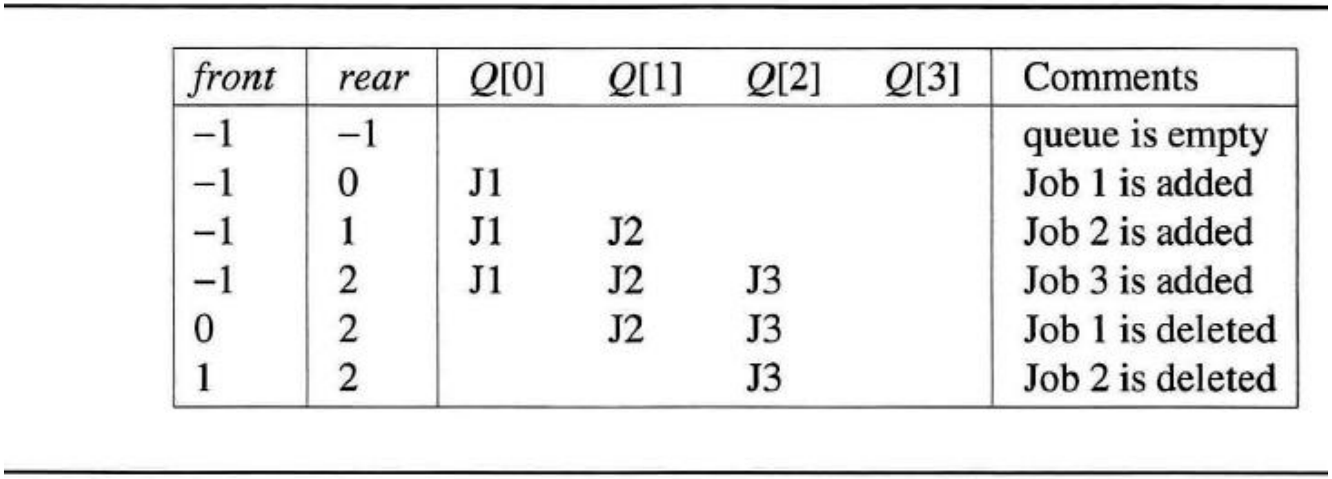


```
element deleteq()
{/* remove element at the front of the queue */
   if (front == rear)
      return queueEmpty(); /* return an error key */
   return queue[++front];
}
```

Program 3.6: Delete from a queue

Sequential Representation

- Example: Job Scheduling by an OS

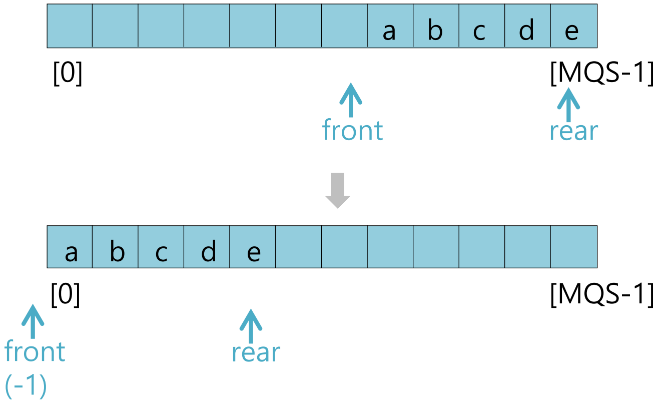


front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
−1	0	J1				Job 1 is added
−1	1	J1	J2			Job 2 is added
−1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure 3.5: Insertion and deletion from a sequential queue

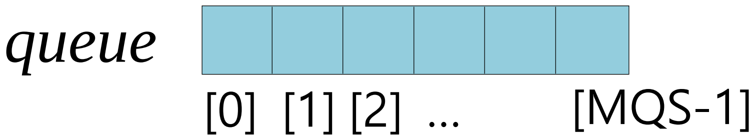
– queueFull

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

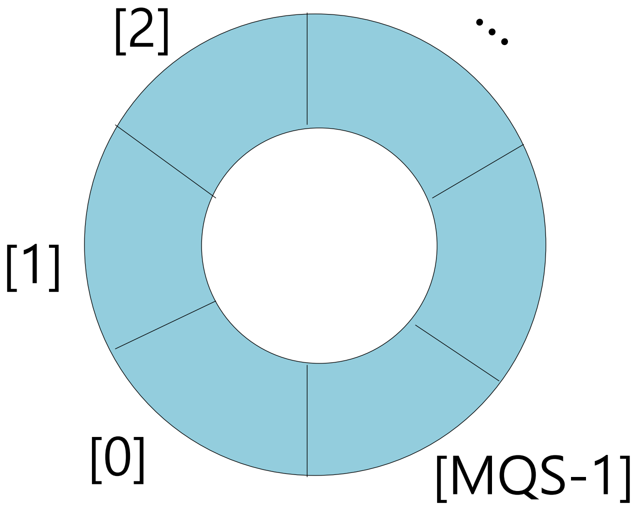


Circular Queue

- Uses an 1D array, *queue*



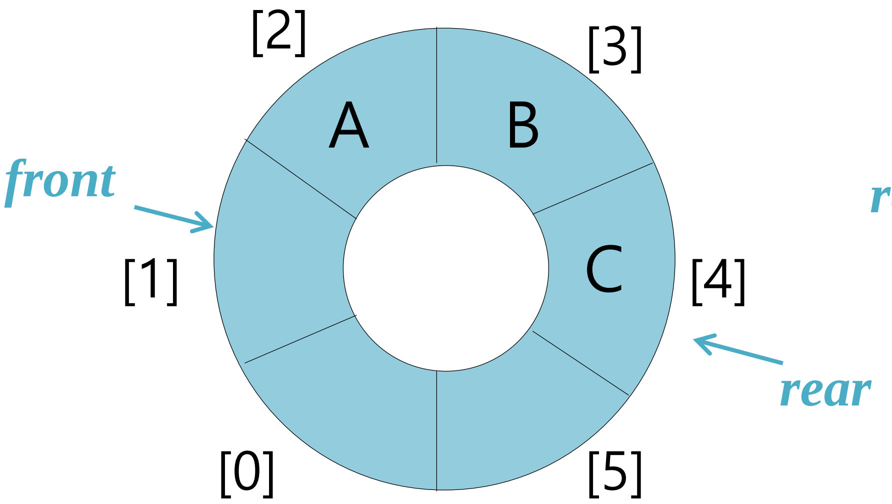
- Circular view of an 1D array

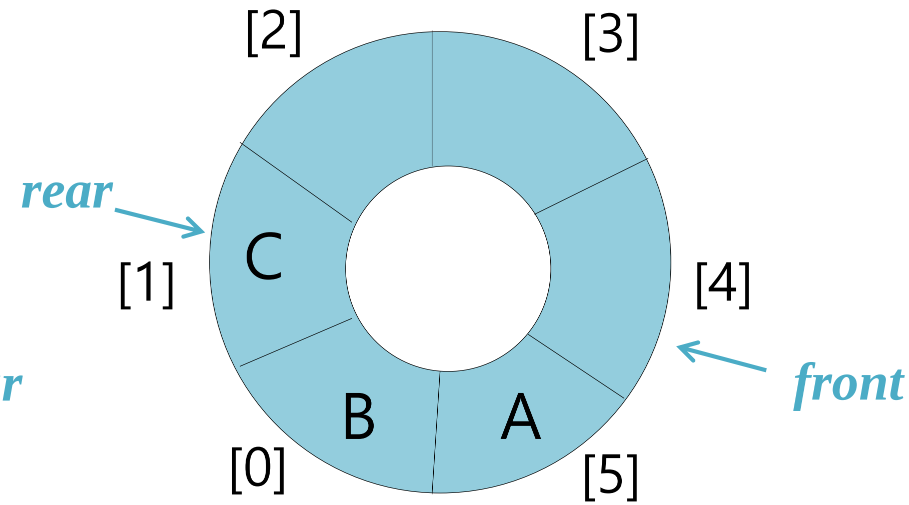


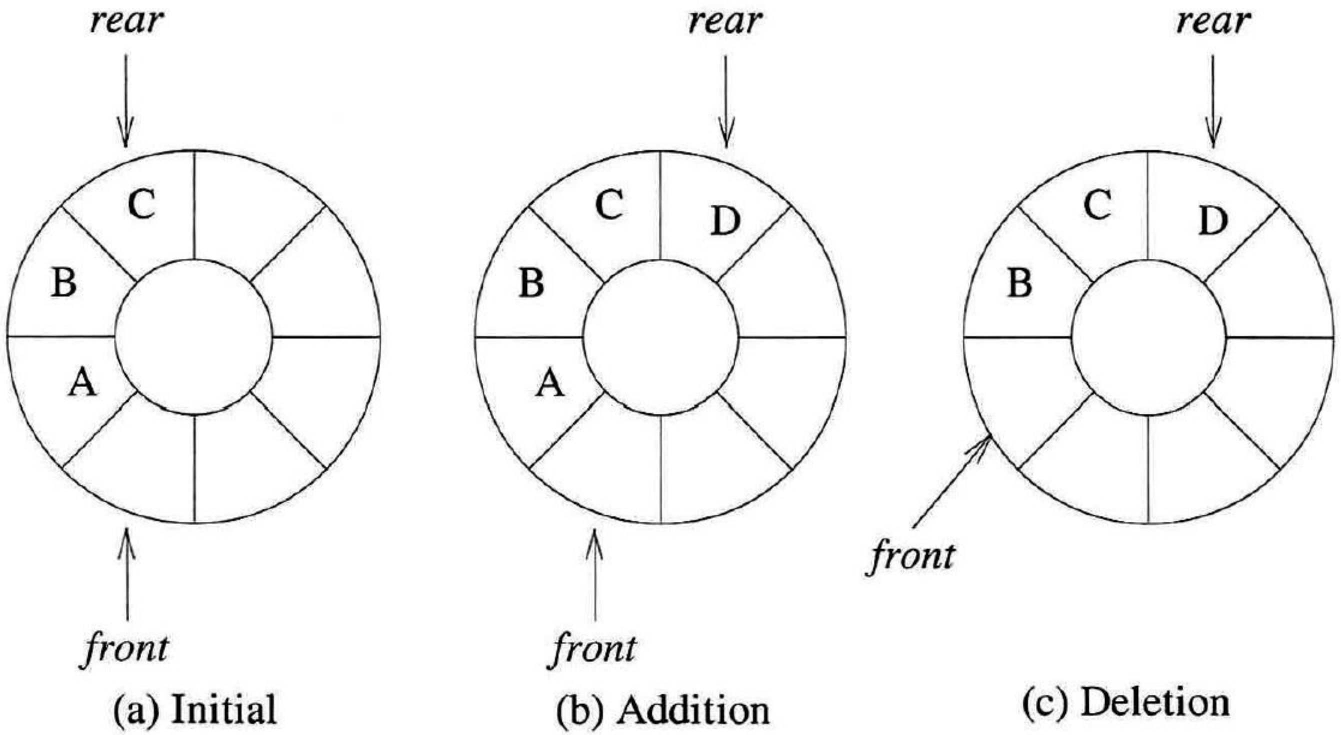
Initial values
front = *rear* = 0

Circular Queue

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







(a) Initial (b) Addition (c) Deletion

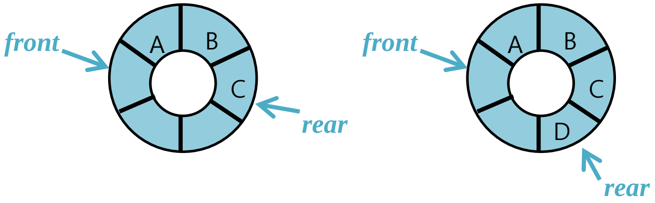
Figure 3.6: Circular queue

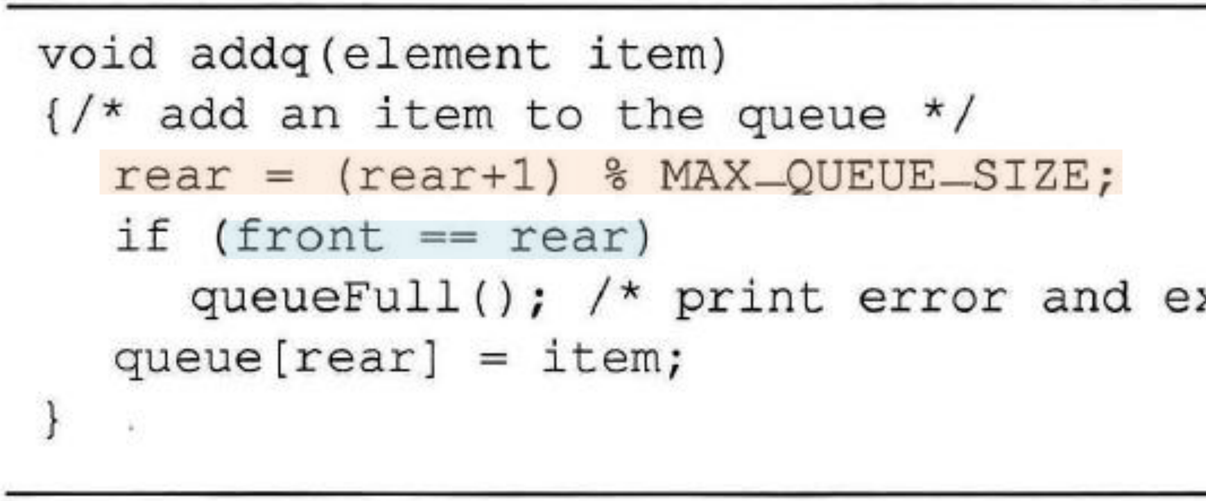
- (b) → 4 additions ➔ queue full, front == rear
- (c) → 3 deletions ➔ queue empty, front == rear
- We cannot distinguish between an empty and a full queues.

- To avoid the resulting confusion, **we shall increase the capacity of a queue just before it becomes full**.

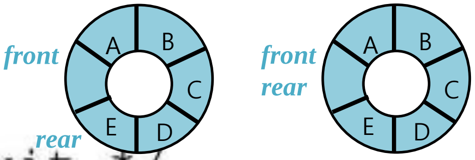
Circular Queue

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





```
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;
}
```

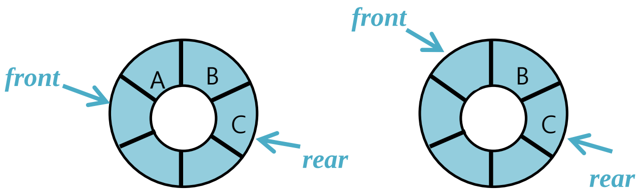


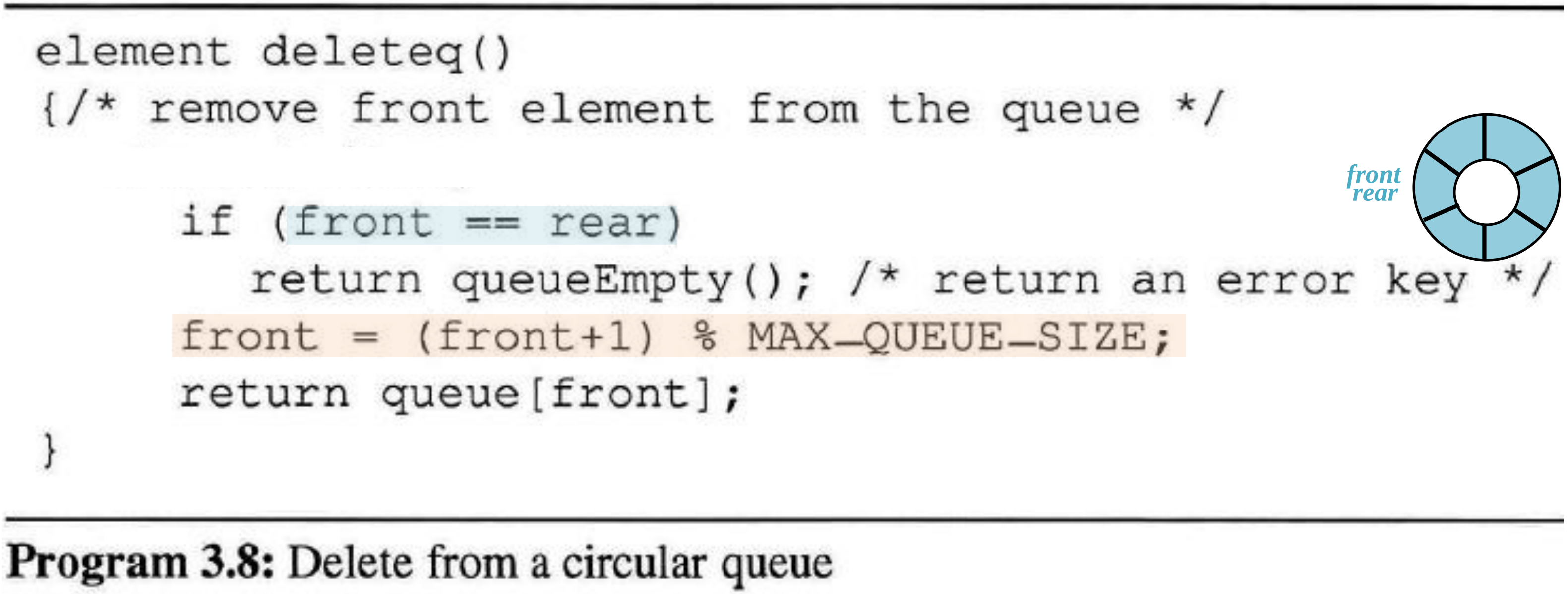
Program 3.7: Add to a circular queue

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

Circular Queue

- Delete an element from the circular queue.
  - Move *front* one clockwise.
  - Then extract from *queue[front]*.

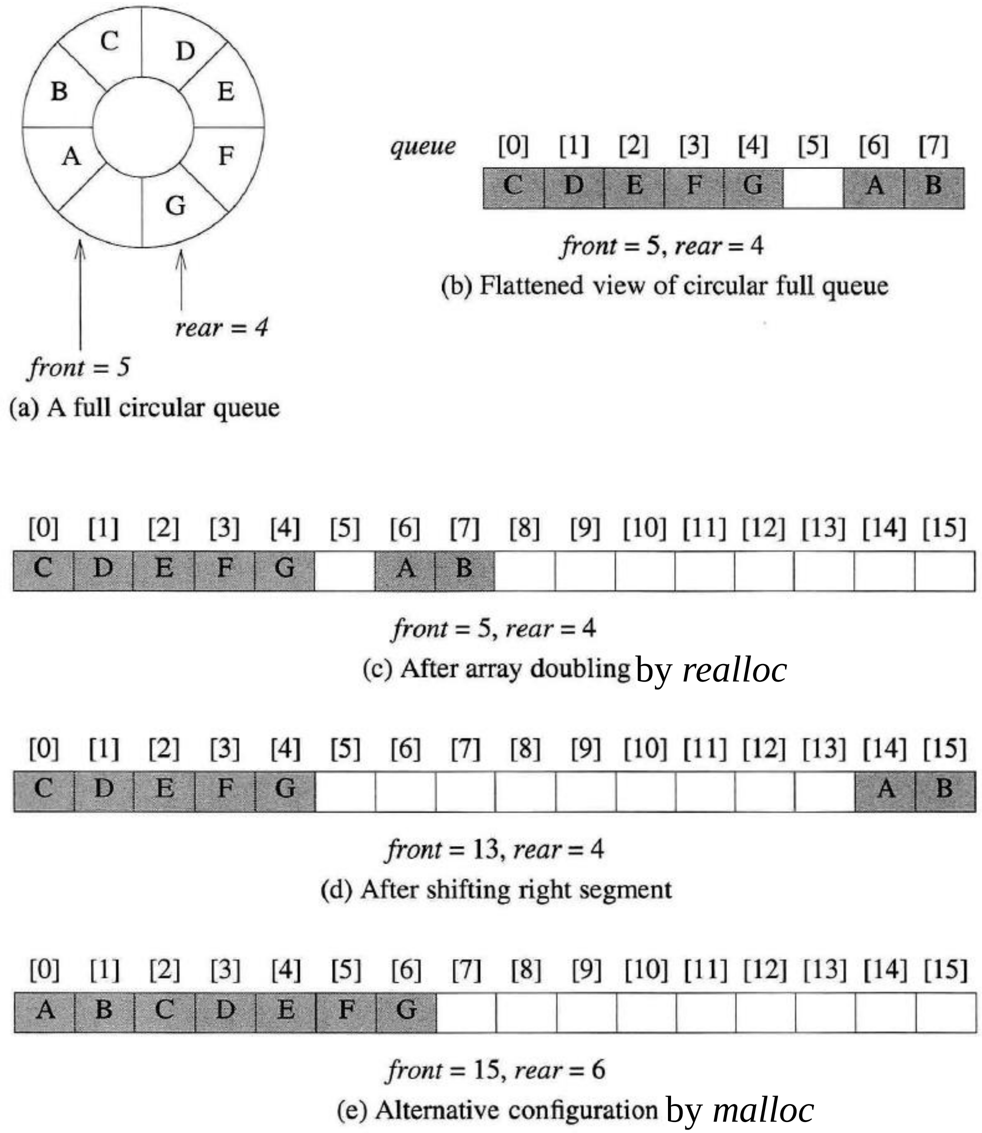




```
element deleteq()
{/* remove front element from the queue */

     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

3.4 Circular Queues Using Dynamically Allocated Arrays



(a) A full circular queue

queue	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	C	D	E	F	G		A	B

front = 5, *rear* = 4

(b) Flattened view of circular full queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G		A	B								

front = 5, *rear* = 4

(c) After array doubling by *realloc*

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G										A	B

front = 13, *rear* = 4

(d) After shifting right segment

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									

front = 15, *rear* = 6

(e) Alternative configuration by *malloc*

Figure 3.7: Doubling queue capacity

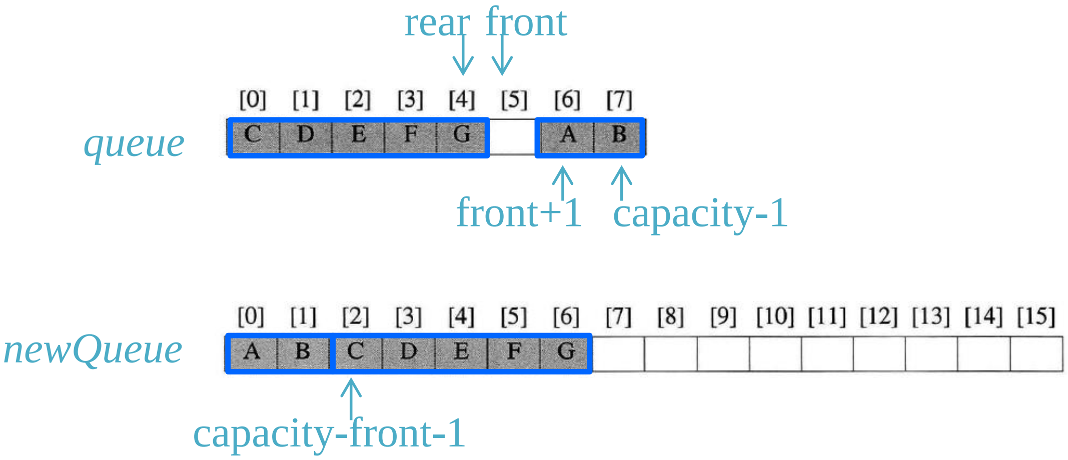
(case 1) (b)→(c)→(d)
(case 2) (b)→(e) , Program 3.10

- Figure 3.7 (b)→(e)

(1) Create a new array *newQueue* of twice the capacity.

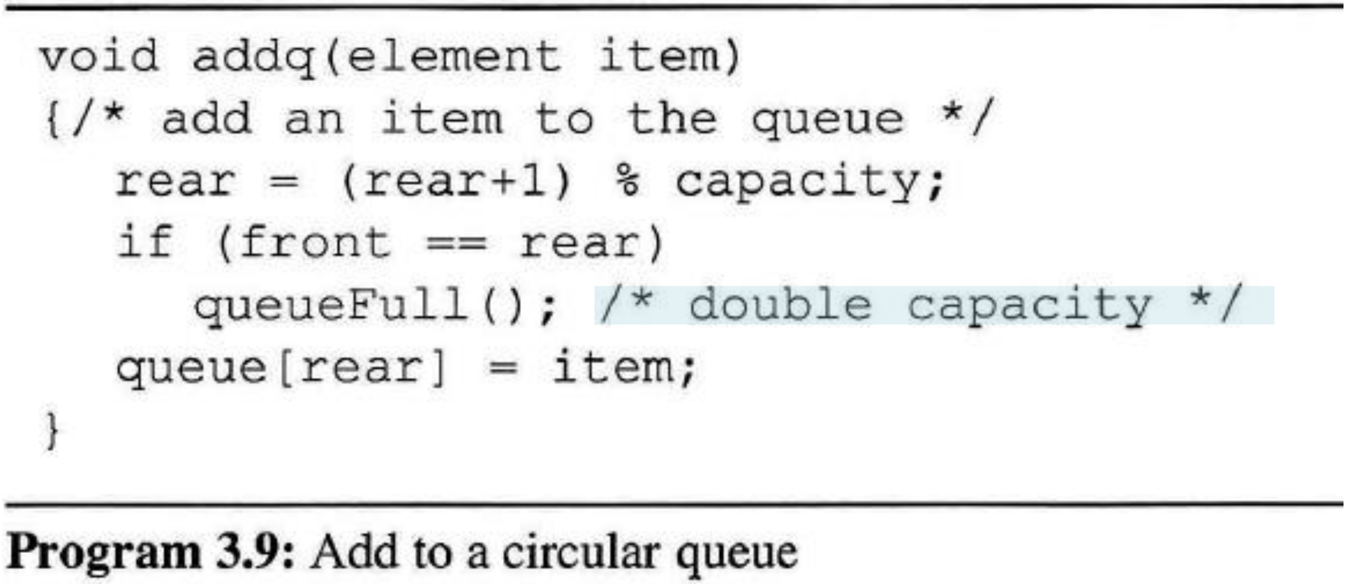
(2) Copy the second segment (i.e., the elements *queue* [*front* +1] through *queue* [*capacity* −1]) to positions in *newQueue* beginning at 0.

(3) Copy the first segment (i.e., the elements *queue* [0] through *queue* [*rear*]) to positions in *newQueue* beginning at *capacity* −*front*−1.



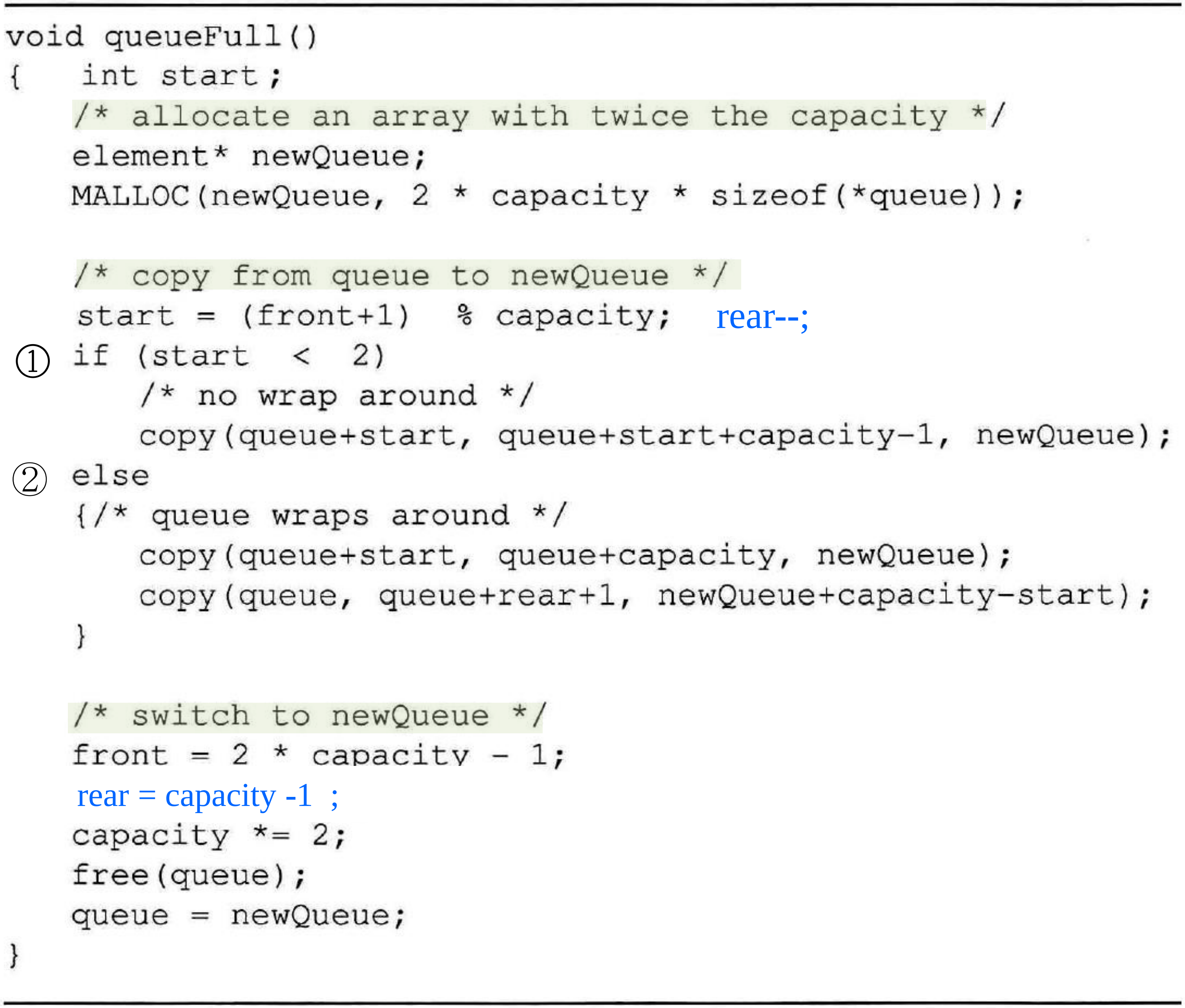
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	D	E	F	G		A	B

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									



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

Program 3.9: Add to a circular queue

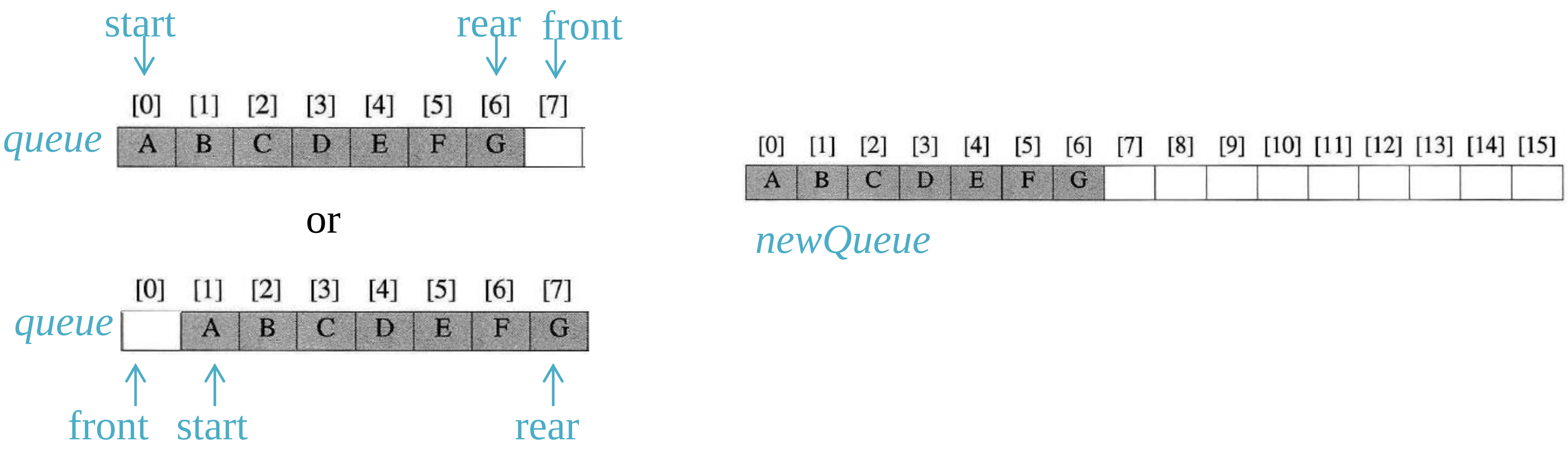


```
void queueFull()
{   int start;
    /* allocate an array with twice the capacity */
    element* newQueue;
    MALLOC(newQueue, 2 * capacity * sizeof(*queue));

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

Program 3.10: Doubling queue capacity < Figure 3.7 (b)→(e) >

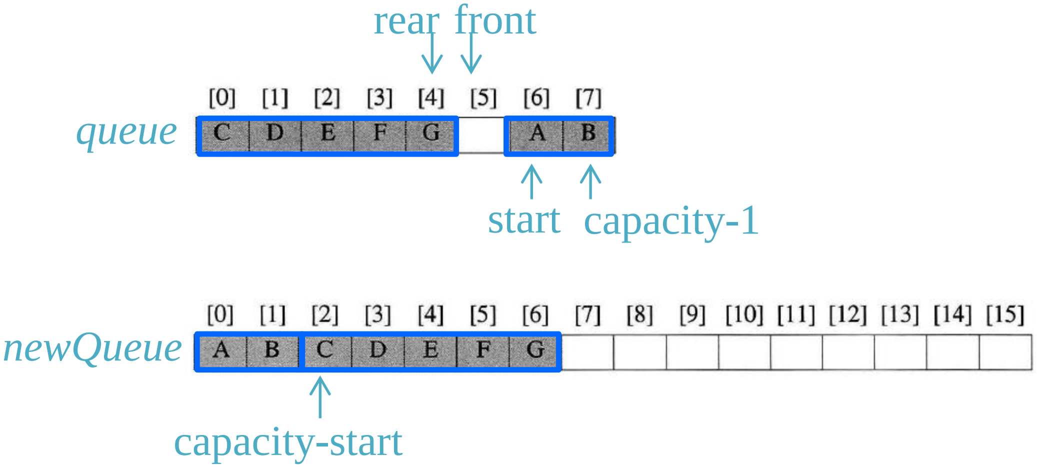
- *copy*(*a*, *b*, *c*)
  - – copies elements from locations *a* through *b*-1 to locations beginning at *c*.
- ①

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



- ②

```
else
{/* queue wraps around */
    copy(queue+start, queue+capacity, newQueue);
    copy(queue, queue+rear+1, newQueue+capacity-start);
}
```



rear
front
[0] [1] [2] [3] [4] [5] [6] [7]
queue
C D E F G A B
start
capacity-1
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]
newQueue
A B C D E F G
capacity-start