

Queues

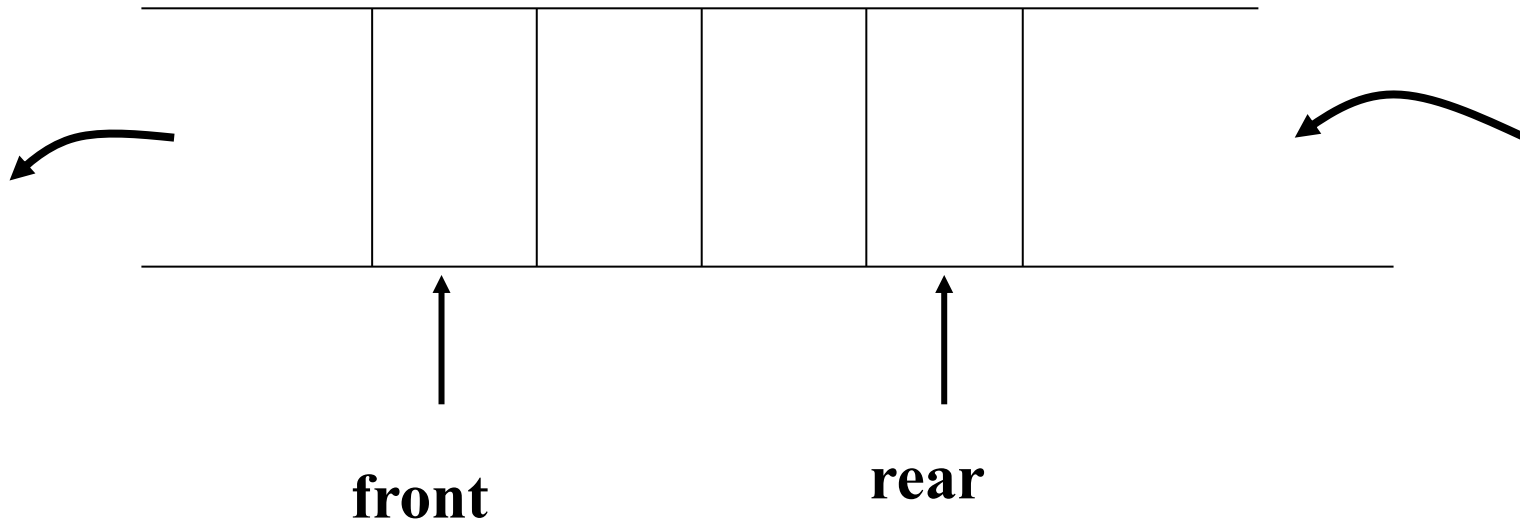
The Abstract Data Type Queue

- A *queue* is a list from which items are deleted from one end (**front**) and into which items are inserted at the other end (**rear**, or **back**)
 - It is like line of people waiting to purchase tickets:
- *Queue* is referred to as a **first-in-first-out (FIFO)** data structure.
 - The first item inserted into a queue is the first item to leave
- Queues have many applications in computer systems:
 - Any application where a group of items is waiting to use a shared resource will use a queue. e.g.
 - jobs in a single processor computer
 - print spooling
 - information packets in computer networks.
 - A *simulation*: a study to see how to reduce the waiting time involved in an application

A Queue

dequeue

enqueue



ADT Queue Operations

- *enqueue(QueueItemType newItem)*
 - Inserts a new item at the end of the queue (at the **rear** of the queue)
 - If array implementation is used and the queue is full, give an error message
 - If linked list implementation is used and memory is full, give an error message.
- *QueueItemType dequeue()*
 - Removes (and returns) the element at the **front** of the queue
 - Remove the item that was added earliest
 - If the queue is empty, give an error message
- *reset()*
 - Empty the queue
 - If linked list implementation is used, free the memory space used by the linked list.

Some Queue Operations

Operation

Queue after operation

an empty queue

front



enqueue(5)

5

enqueue(3)

5 3

enqueue(2)

5 3 2

dequeue()

3 2

→ 5 is returned

enqueue(7)

3 2 7

dequeue()

2 7

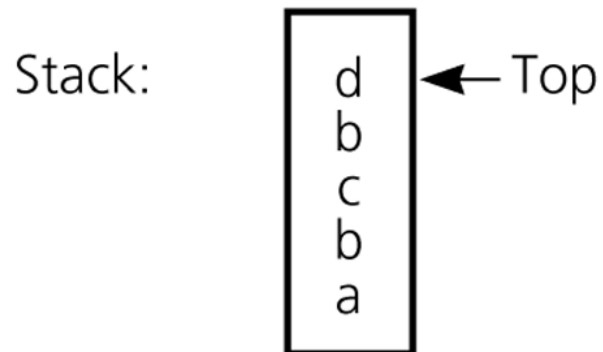
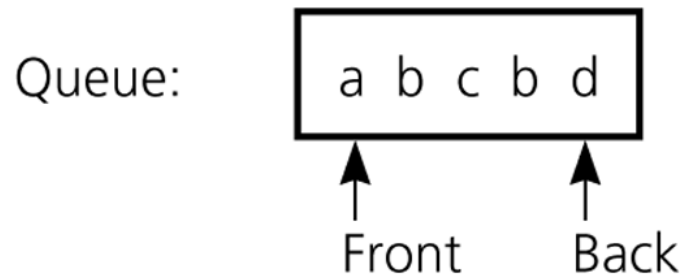
→ 3 is returned

Recognizing Palindromes

- A palindrome
 - A string of characters that reads the same from left to right as it does from right to left
- To recognize a palindrome, a queue can be used in conjunction with a stack
 - A stack reverses the order of occurrences
 - A queue preserves the order of occurrences
- A nonrecursive recognition algorithm for palindromes
 - As you traverse the character string from left to right, insert each character into both a queue and a stack
 - Compare the characters at the front of the queue and the top of the stack

Recognizing Palindromes (cont.)

String: abc bd



The results of inserting a string
into both a queue and a stack

Recognizing Palindromes -- Algorithm

isPal(in str:string) : boolean // Determines whether str is a palindrome or not

```
len = length of str;
for (i=1 through len) {
    nextChar = ith character of str;
    enqueue(nextChar);
    push(nextChar);
}
charactersAreEqual = true;
while (Queue is not empty and charactersAreEqual) {
    if (dequeue( ) != pop( ))
        chractersAreEqual = false;
}
return charactersAreEqual;
```

Implementations of the ADT Queue

- Array-based implementations of queue
 - A naive array-based implementation of queue
 - A circular array-based implementation of queue
- Pointer-based implementations of queue
 - A linear linked list with two external references
 - A reference to the front
 - A reference to the back

A Naive Array-based Implementation of Queue



last: shows the array element where the new item to be inserted.

enqueue(): inserts new element to the place which is pointed by **last**.

dequeue(): removes element from the 0th position, and shifts all the remaining elements left by 1 position.

A Naive Array-based Implementation of Queue

```
#define N 500    // max number of elements
                  // that can be stored in the queue

int Q[N];        // Queue
int last=0;      // initially queue is empty

int enqueue(int item){
    if (last == N){
        printf("Queue is full\n");
        return -1;
    }
    else{
        Q[last] = item;
        last++;
        return 0;
    }
}
```

A Naive Array-based Implementation of Queue

```
int dequeue(){
    int item, i;

    if (last == 0){
        printf("Queue is empty\n");
        return -1;    // if queue has positive values
    }
    else{
        item = Q[0];
        // shift left items in the queue by 1 position
        for (i=1; i< last; i++)
            Q[i-1] = Q[i];
        last--;
        return item;
    }
}
```

A Naive Array-based Implementation of Queue

```
void reset() {  
    last = 0;  
}
```

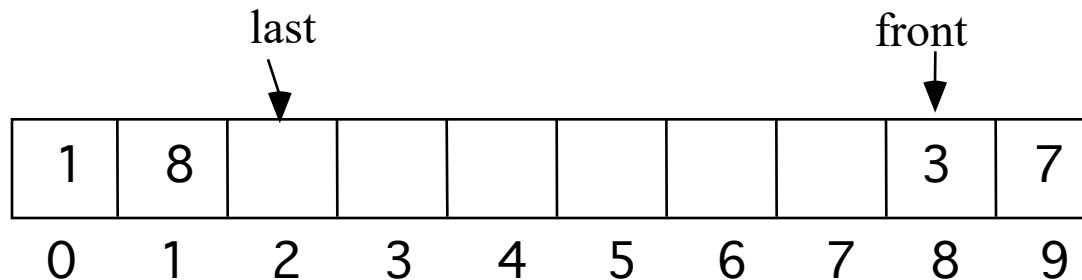
Analysis of the naive array-based implementation:

- Enqueue: $O(1)$
- Dequeue: $O(n)$ because of the shift operation
- Reset: $O(1)$

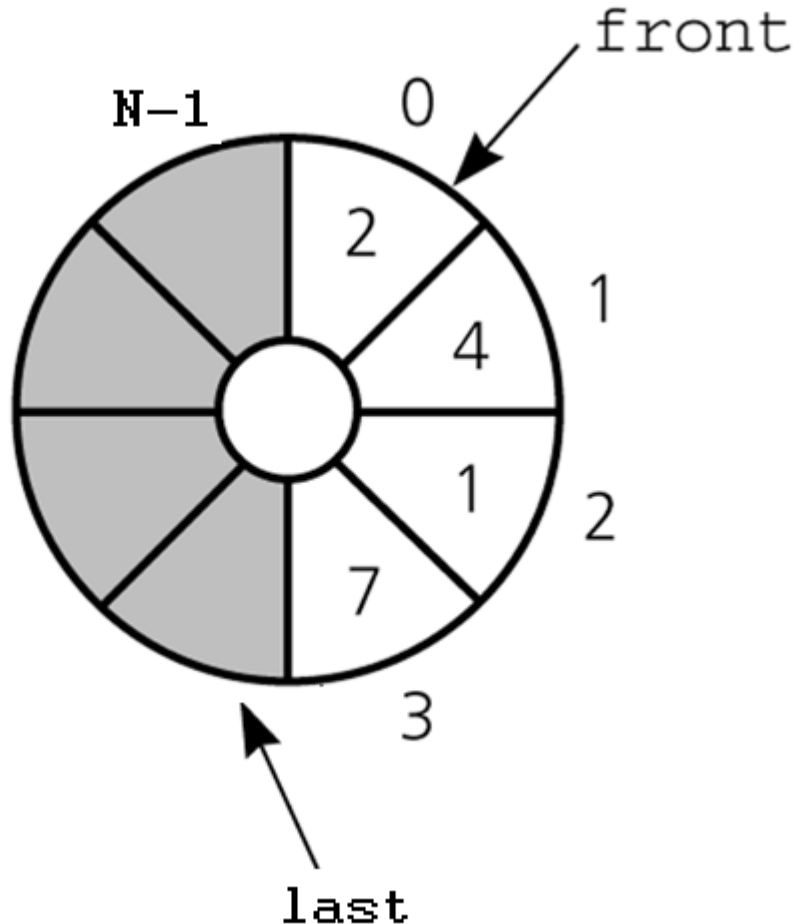
→ Circular array implementation

Circular Array Implementation

- The front and last are the same as the basic model, except: The queue wraps around when the end of the array is reached.

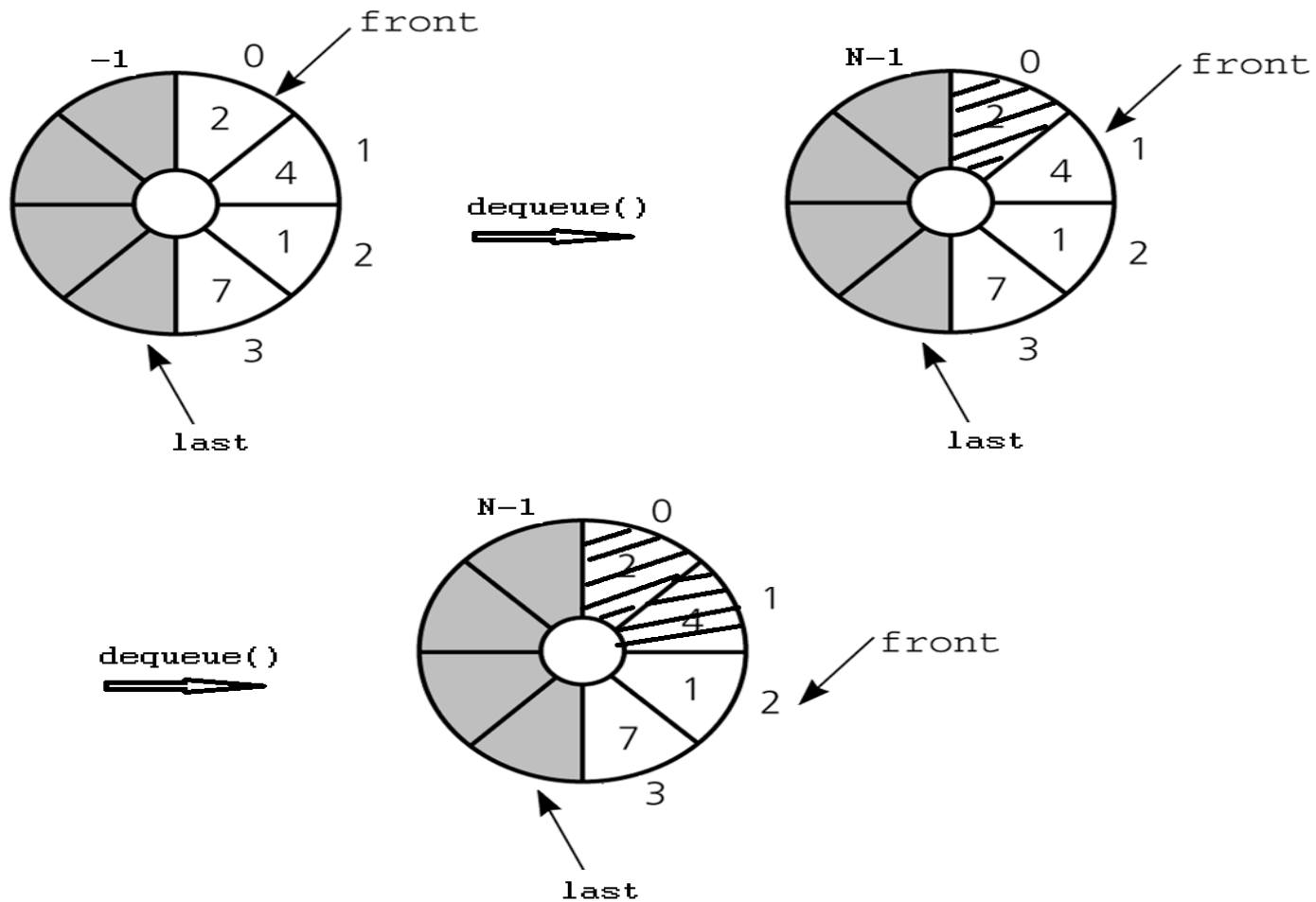


A Circular Array-Based Implementation



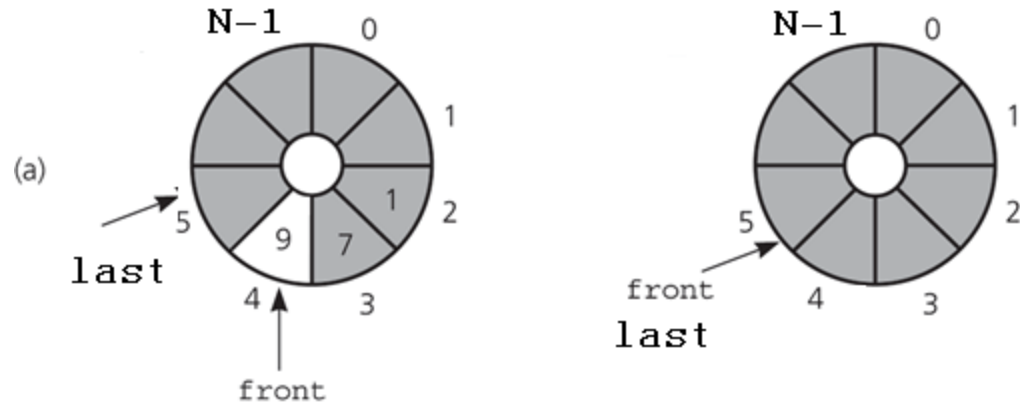
When either **front** or **last**
past **N-1**
it wraps around to 0.

The effect of some operations of the queue



PROBLEM – Queue is Empty or Full

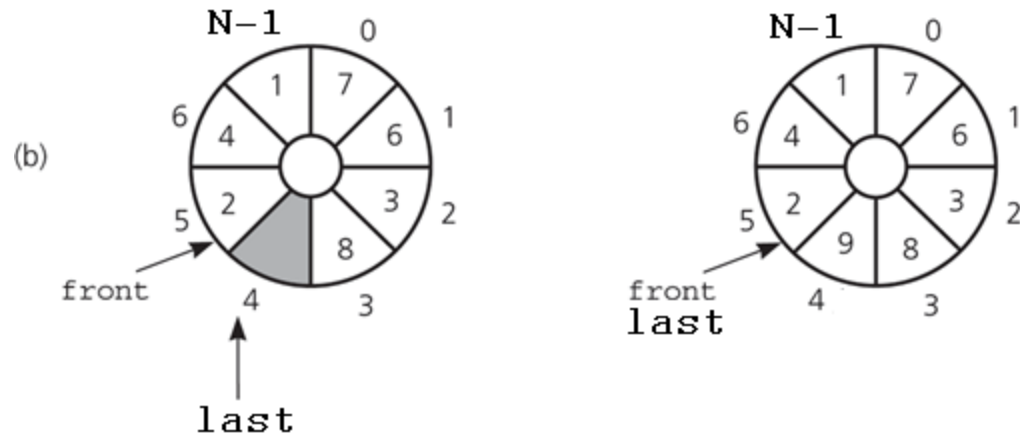
Queue with single item → Delete item—queue becomes empty



front and **last** cannot be used to distinguish between *queue-full* and *queue-empty* conditions.

So, we need extra mechanism to distinguish between *queue-full* and *queue-empty* conditions.

Queue with single empty slot → Insert 9—queue becomes full



Solutions for Queue-Empty/Queue-Full Problem

1. Using a counter to keep the number of items in the queue.
 - Initialize count to 0 during creation; Increment count by 1 during insertion; Decrement count by 1 during deletion.
 - $\text{count}=0 \rightarrow \text{empty}$; $\text{count}=N \rightarrow \text{full}$
2. Using isFull flag to distinguish between the full and empty conditions.
 - When the queue becomes full, set isFullFlag to true; When the queue is not full set isFull flag to false;
3. Using an extra array location (and leaving at least one empty location in the queue).
 - Declare $N+1$ locations for the array items, but only use N of them. We do not use one of the array locations.
 - *Full*: $\text{front equals to } (\text{last}+1)\%(N+1)$
 - *Empty*: front equals to last

Using a counter

- To initialize the queue, set

```
front = 0
last = 0
count = 0
```
- Inserting into a queue

```
Q[last] = newItem;
last = (last+1) % N;
count++;
```
- Deleting from a queue

```
front = (front+1) % N;
count--;
```
- Full: `count == N`
- Empty: `count == 0`

Circular Array-Based Implementation

Using a counter

```
#define N 500
int Q[N];
int front=0, last=0, count=0;

int enqueue(int item){
    if (count >= N){
        printf("Queue is full\n");
        return -1;
    }
    else{
        Q[last] = item;
        last = (last + 1) % N;
        count++;
        return 0;
    }
}
```

Circular Array-Based Implementation

Using a counter

```
int dequeue() {
    int item;
    if (count == 0) {
        printf("Queue is empty\n");
        return -1;
    }
    else{
        item = Q[front];
        front = (front + 1) % N;
        count--;
        return item;
    }
}
```

Circular Array-Based Implementation

Using a counter

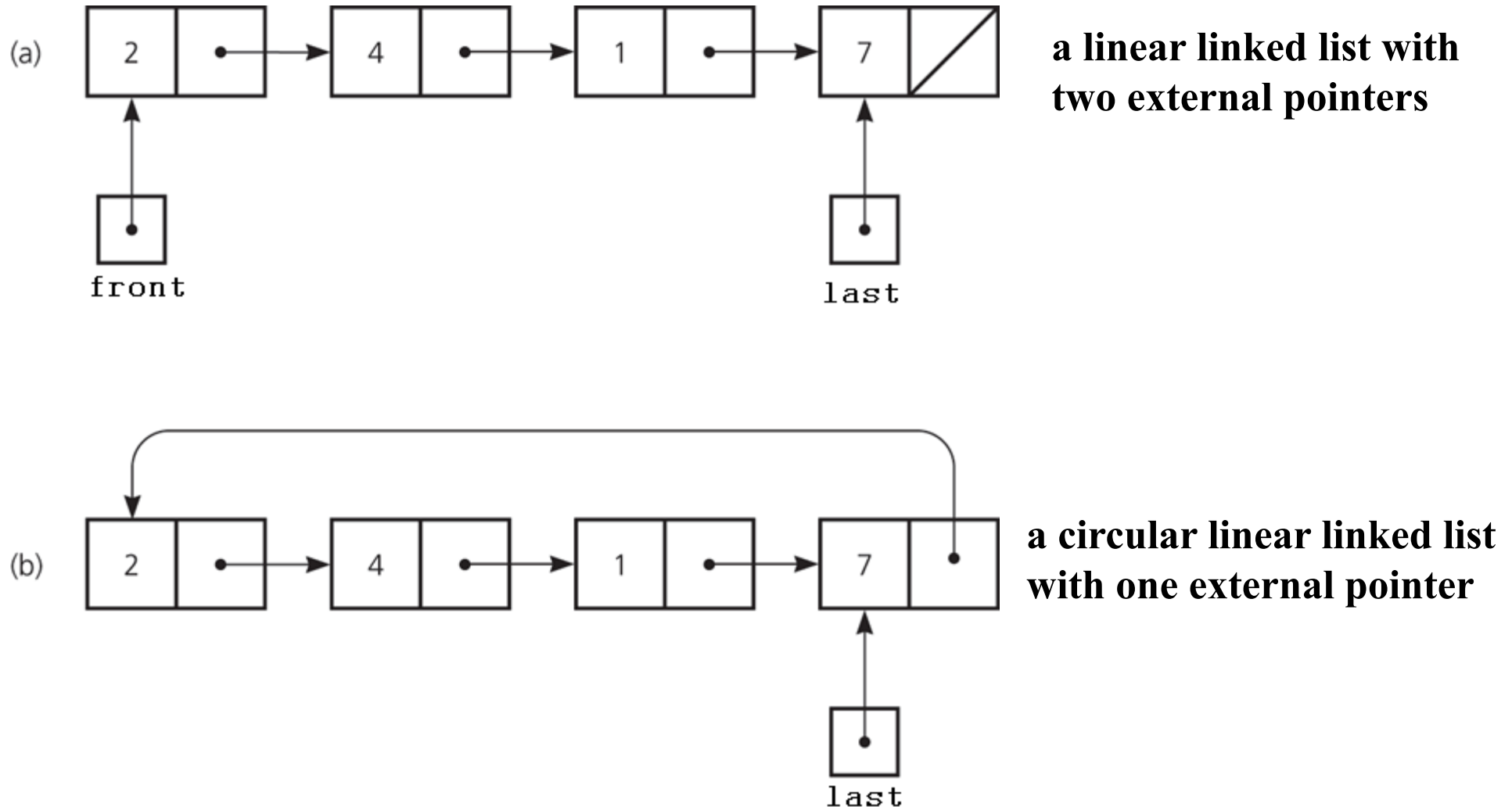
```
void reset() {  
    count = 0;  
    front = 0;  
    last = 0;  
}
```

Analysis of the circular array-based implementation:

- Enqueue: $O(1)$
- Dequeue: $O(1)$
- Reset: $O(1)$

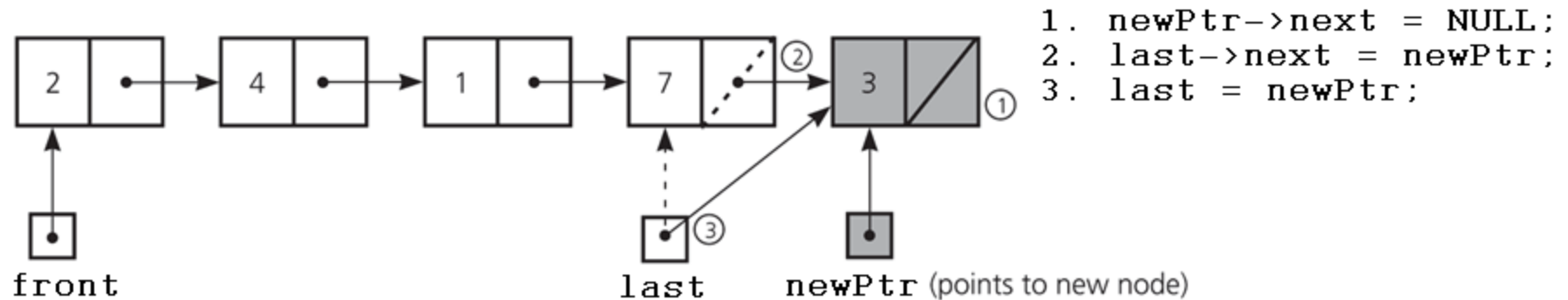
→ Circular array implementation should be preferred

Pointer-based implementations of queue

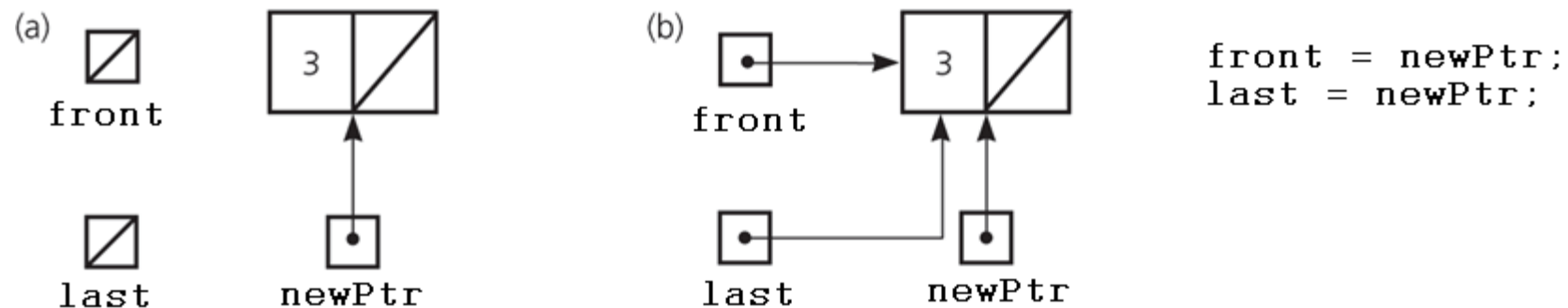


A Pointer-Based Implementation -- enqueue

Inserting an item into a nonempty queue



Inserting an item into an empty queue

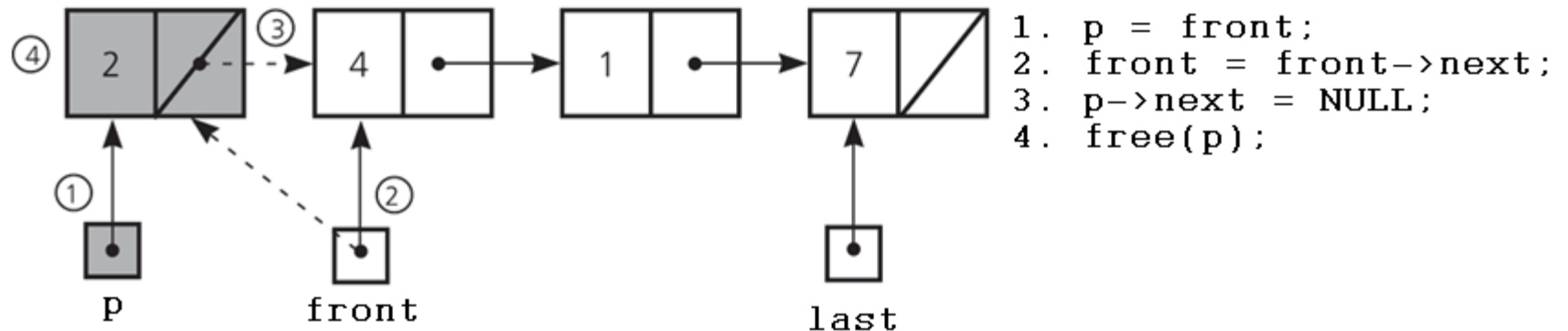


a) before insertion

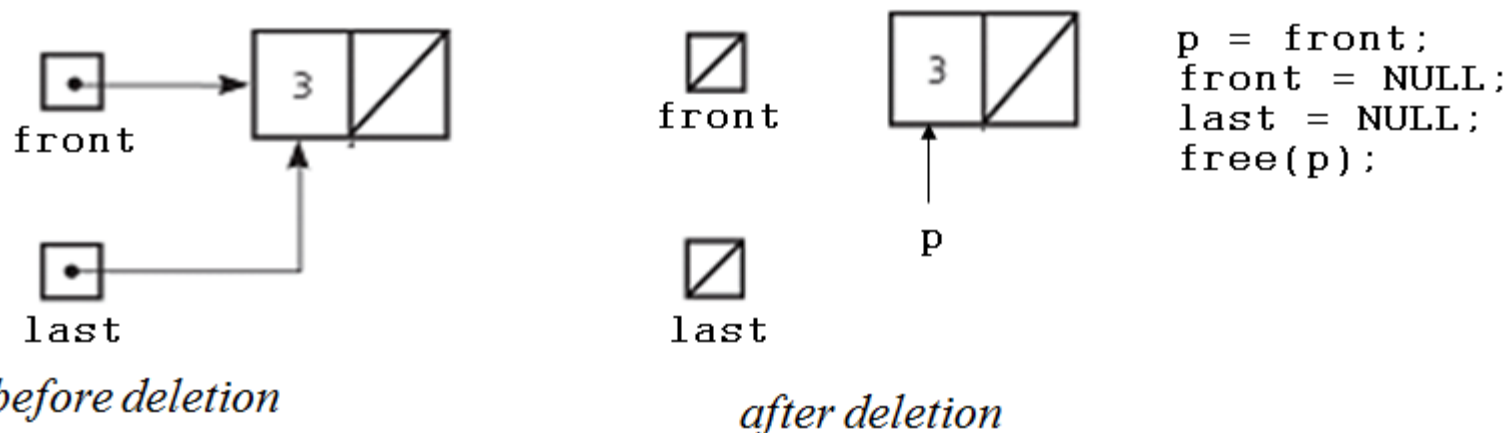
b) after insertion

A Pointer-Based Implementation -- dequeue

Deleting an item from a queue of more than one item



Deleting an item from a queue with one item



A Pointer-Based Implementation

```
typedef struct q{  
    int item;  
    struct q *next;  
}QUEUE;
```

```
QUEUE *front = NULL, *last = NULL;
```

A Pointer-Based Implementation (cont.)

```
int enqueue(int item){
    QUEUE *p;
    p = (QUEUE *)malloc(sizeof(QUEUE));
    if (p == NULL){
        printf("Memory is full\n");
        return -1;
    }
    p->item = item;
    p->next = NULL;
    if (front == NULL){ // if queue is empty
        front = p;      // after enqueue, it has 1 element
        last = p;
    }
    else {               // otherwise, insert the new element
        last->next = p;  // at the end
        last = p;
    }
    return 1;
}
```

A Pointer-Based Implementation (cont.)

```
int dequeue() {
    QUEUE *p;
    int item;
    if (front == NULL) {
        printf("Queue is empty\n");
        return -1;
    }
    p = front;
    front = front->next;
    if (front == NULL) {
        last = NULL;
    }
    item = p->item;
    free(p);
    return item;
}
```

A Pointer-Based Implementation (cont.)

```
void reset() {  
    QUEUE *p;  
    p = front;  
    while (p) {  
        front = front->next;  
        free(p) ;  
        p = front;  
    }  
    last = NULL;  
}
```

Comparing Implementations

- Fixed size versus dynamic size
 - A statically allocated array
 - Prevents the `enqueue` operation from adding an item to the queue if the array is full
 - A resizable array or a reference-based implementation
 - Does not impose this restriction on the `enqueue` operation
- Pointer-based implementations
 - A linked list implementation
 - More efficient; no size limit

Priority Queues

- New items are inserted into the queue with respect to their priority.
- Elements having the highest priority are inserted at the beginning.

```
// linked list implementation of priority queues
```

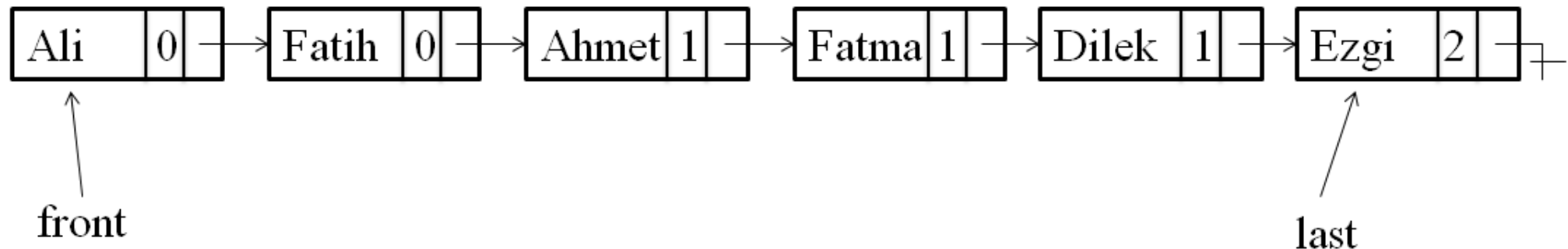
```
typedef struct pq{  
    char name[10];  
    int priority;  
    struct pq *next;  
} PQUEUE;
```

```
PQUEUE *front = NULL, *last = NULL;
```

Linked List Implementation of Priority Queues

- Insert the below items into the priority queue

<u>Name</u>	<u>priority</u>	
Ali	0	←———— highest priority
Ahmet	1	
Fatma	1	
Ezgi	2	←———— lowest priority
Fatih	0	
Dilek	1	



Linked List Implementation of Priority Queues

- ***Enqueue()***: As shown in the previous figure. Each item is inserted according to its priority value.
 - If the list is empty, it is inserted as the front and last element.
 - Otherwise, search for the place to insert the new item, it can be front, last, or middle.
- ***Dequeue()***: Always remove item from the front.
- ***Reset()***: Remove each item in the list one by one.

Priority Queue Using Heap Structure

- Use **min heap** where the root has the minimum item. Each node must be less than or equal to its children.
- ***Enqueue()***:
 - Insert the new element at the end of the heap
 - Call `heapify()` for the parent of the new element.
 - Continue to call `heapify()` for each node until the root node.
- ***Dequeue()***:
 - Return the element from the root node.
 - Remove the last element from the heap and copy it into the root node.
 - Call `heapify()` for the root node.