1/2

Stack and Queue

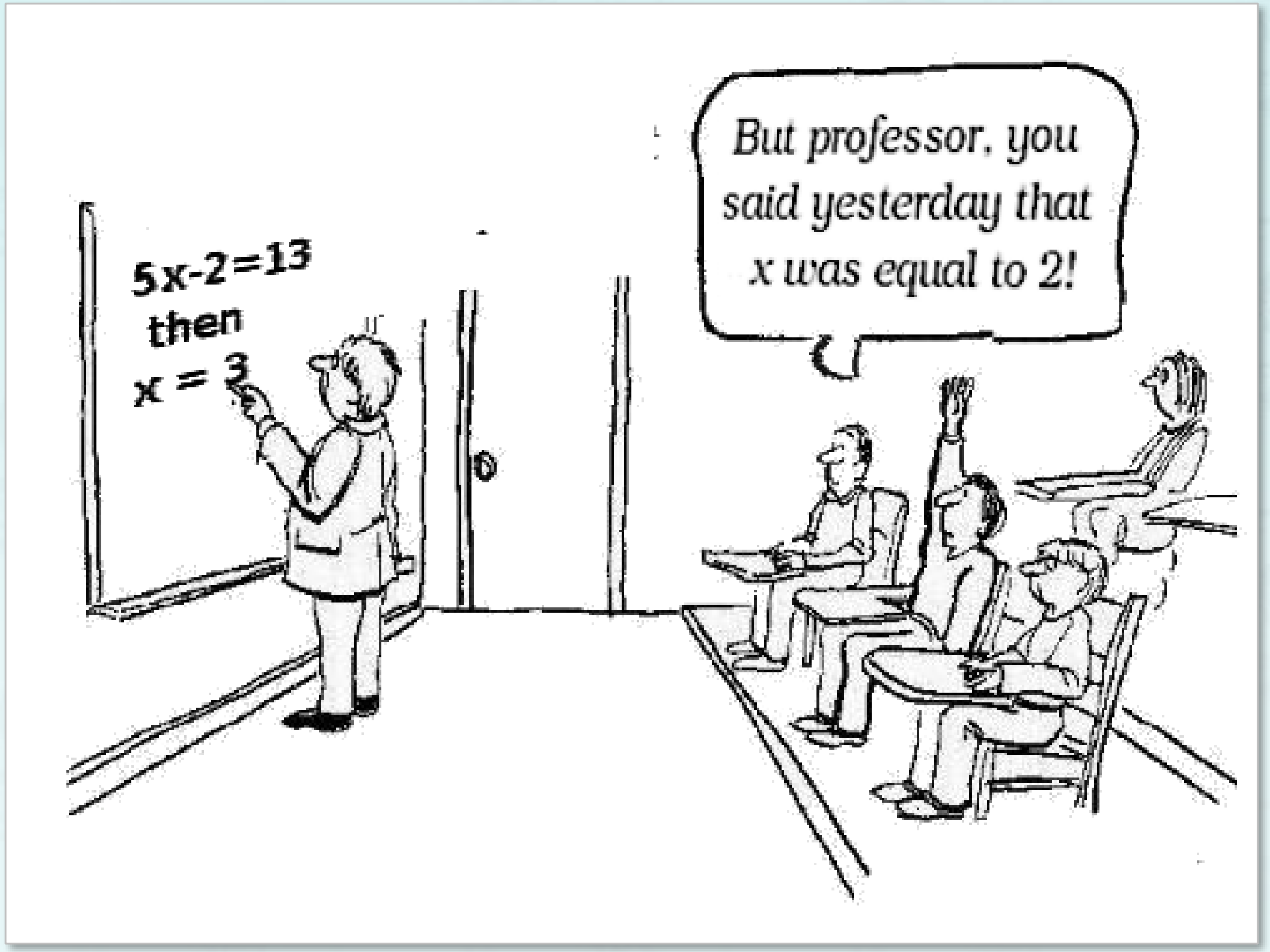
Data Structures

C++ for C Coders

한동대학교 김영섭 교수

idebtor@gmail.com

stacks & queues using dynamic arrays



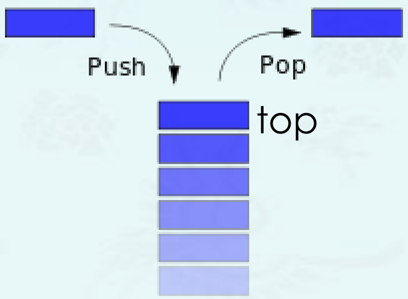
Source http://eng.funiacs.com/funny-pictures/31079

Stacks

Fundamental data types:

- **Value**: collections of objects
- **Operations:** insert, remove, iterate, test if empty, test if full
- Intent is clear when we insert.
- Which item do we remove?

stack



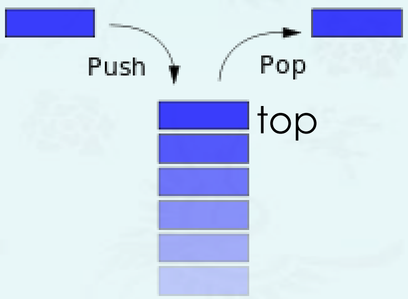
Stack: Examine the item most recently added. LIFO = "last in first out"

Stacks

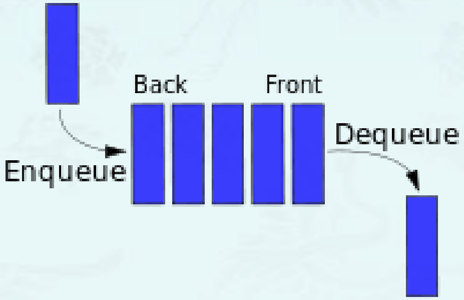
Fundamental data types:

- **Value**: collections of objects
- **Operations:** insert, remove, iterate, test if empty, test if full
- Intent is clear when we insert.
- Which item do we remove?

stack



queue



Stack: Examine the item most recently added. LIFO = "last in first out"
Queue: Examine the item least recently added. FIFO = "first in first out"

Stacks

ADT Stack is
objects: a finite ordered list with zero or more elements
functions: Stack CreateStack(maxStackSize) bool empty() void push(item) void pop() int size;

Why ADTs?

Why ADTs again?

Separate interface and implementation.
Ex: stack, queue, bag, priority queue, symbol table, union-find, ...

Benefits.

- **Client** can't know details of implementation ⇒
  client has many implementation from which to choose.
- **Implementation** can't know details of client needs ⇒
  many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Interface: description of data type, basic operations.

Stacks

Example: Stack of strings data type (implemented in Java)

	`#include <stack>`	
	`stack<int>`	*creates an empty stack of integers*
`void`	`push(value_type& item)`	*inserts a new item onto stack*
`void`	`pop()`	*removes top item from stack (which is most recently added)*
`value_type&`	`top()`	*returns a reference to the top item*
`bool`	`empty()`	*is the stack empty?*
`int`	`size()`	*returns the number of items in the stack*

Warm-up client: Reverse sequence of strings from standard input.

Stacks

Stack test client:

- Read string from standard input.
  - If string equals “-”, pop string from stack and print.
  - Otherwise, push string onto stack.

```
public static void main (String[] args) {
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty()) {
        String s = StdIn.readString();
        if (s.equals(“-”)
            Stdout.print(stack.pop()));
        else
            stack.push(s);
    }
}
```

Exercise:

```
%more tobe.txt
to be or not to – be - - that - - - is
% java StackOfStrings < tobe.txt
```

```
to be not that or be
```

Stacks using dynamic arrays

Array implementation of a stack:

- Use array s[ ] to store N items on stack.
- push(): add new item at s[N].
- pop(): remove item from s[N-1].

successive doubling
만약에 push 할자리가 부족하게 되면
malloc 해야하는데 얼만큼?
2배
그럼 만약에 capa가 10이라고 하면
부족하게되면 20 거기서 더 부족하면 40
거기서 더 부족하면 80 이렇게 doubling을 한다

s[ ]	to	be	or	not	to	be	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
							N		capacity = 10	

반대로 줄일 때는?
1/4만큼 왔을 때 줄인다

Defect. Stack overflows when N exceeds capacity. *[stay tuned]*

Stacks using dynamic arrays

```
public class FixedSizeStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedSizeStackOfStrings(int capacity) {
        s = new String[capacity];
    }
    public boolean isEmpty() {
        return N == 0;
    }
    public void push(String item) {
        s[N++] = item;
    }
    public String pop() {
        return s[--N];
    }
}
```

a shortcoming (stay tuned)

use to index into array; then increment N

decrement N: then use to index into array

Stacks using dynamic arrays

Things to consider:

- **Overflow and underflow:**
  - Underflow: throw exception if pop from an empty stack or return null;
  - Overflow: use resizing array for array implementation. [stay tuned]

- **Null items:** Allow null items to be inserted or not. Clarify during the design.

- **Loitering:** Holding a reference to an object when it is no longer needed.

```
public String pop() {
    return s[--N];
}
```

loitering

```
public String pop() {
    String item = s[--N];
    s[N] = null;
    return item;
}
```

This version avoids "loitering": Garbage collector can reclaim memory only if no outstanding references.
In C/C++ implementation. free the resources.

Stacks using dynamic arrays

Problem: Requiring client to provide capacity (size of stack) is inappropriate.
Question: How to grow and shrink array?

First try.

- **push(): increase size of array s[] by 1.**
- **pop(): decrease size of array s[] by 1.**

Too expensive.

- Need to copy all items to a new array.
- Inserting first N items takes time proportional to $1 + 2 + 3 + \ldots + N \approx N^2/2$.

infeasible for large N

Challenge: **Ensure that array resizing happens infrequently.**

Stacks using dynamic arrays

"successive doubling"

Q. How to grow and shrink array?
A. If array is full, create a new array of twice the size, and copy items.

```
public class ResizingStackOfStrings {
    s = new String[1];
}

public void push(String item) {
    if (N == s.length) resize(s.length * 2);
    s[N++] = item;
}

private void resize(int capacity) {
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

Consequence: Inserting first N items takes time proportional to N, not N^2

Stacks using dynamic arrays

Q. Cost of inserting first N items by resize(s.length + 10)?
A. $\mathbf{T(N) = 1 + (10 + 20 + 30 + \ \ldots + N)}$

1 array access per push

k array accesses when memory is resized by increment of 10
(ignoring cost to create new array)
(assuming realloc() costs copying each item one by one)

```
When N = 1, Capacity = 1 → 11          // (?) cost to copy the existing items into the new array
   Cost: 1 + (0)                       // (0) since no copy is needed
When N = 2, Capacity = 11
   Cost: 1 + (0)                       // (0) items to copy into the new array
When N = 3, Capacity = 11
   Cost: 1 + (0)                       // (0) since no copy is needed
When N = 4, Capacity = 11
   Cost: 1 + (0)
....
When N = 11, Capacity = 11 → 21
   Cost: 1 + (10)                      // (10) items to copy into the new array
When N = 12, Capacity = 21
   Cost: 1 + (0)
....
When N = 21, Capacity = 21 → 31
   Cost: 1 + (20)                      // (20) items to copy into the new array
When N = 22, Capacity = 31
   Cost: 1 + (0)
```

Stacks using dynamic arrays

Q. Cost of inserting first N items by resize(s.length + 10)?

A. $\mathrm{T(N)} = \mathbf{1} + (\underbrace{\mathbf{10} + \mathbf{20} + \mathbf{30} + \ldots + \mathbf{N}}_{\text{How many terms? k terms, then N = 10k}}) = ?$

$$T(N) = 1 + (10 + 20 + 30 + \cdots + N)$$

$Let\ N = 10k, then\ it\ becomes$

$$T(N) = 1 + (10 + 20 + 30 + \cdots + 10k)$$
$$= 1 + 10(1 + 2 + 3 + \cdots + k)$$
$$= 1 + 10\frac{k(k+1)}{2}$$
$$= 1 + 10\frac{\frac{N}{10}(\frac{N}{10} + 1)}{2}$$
$$Therefore, T(N) = 1 + \frac{N}{2}\left(\frac{N}{10} + 1\right)$$

The time complexity of the algorithm is $O(n^2)$.

Stacks using dynamic arrays

Q. Cost of inserting first N items by resize(s.length * 2) ?
A. $\mathbf{T}(N) = 1 + (1 + 2 + 4 + 8 + \cdots + N)$

1 array access per push

k array accesses to double to size k
(ignoring cost to create new array)
(assuming realloc() costs copying each item one by one)

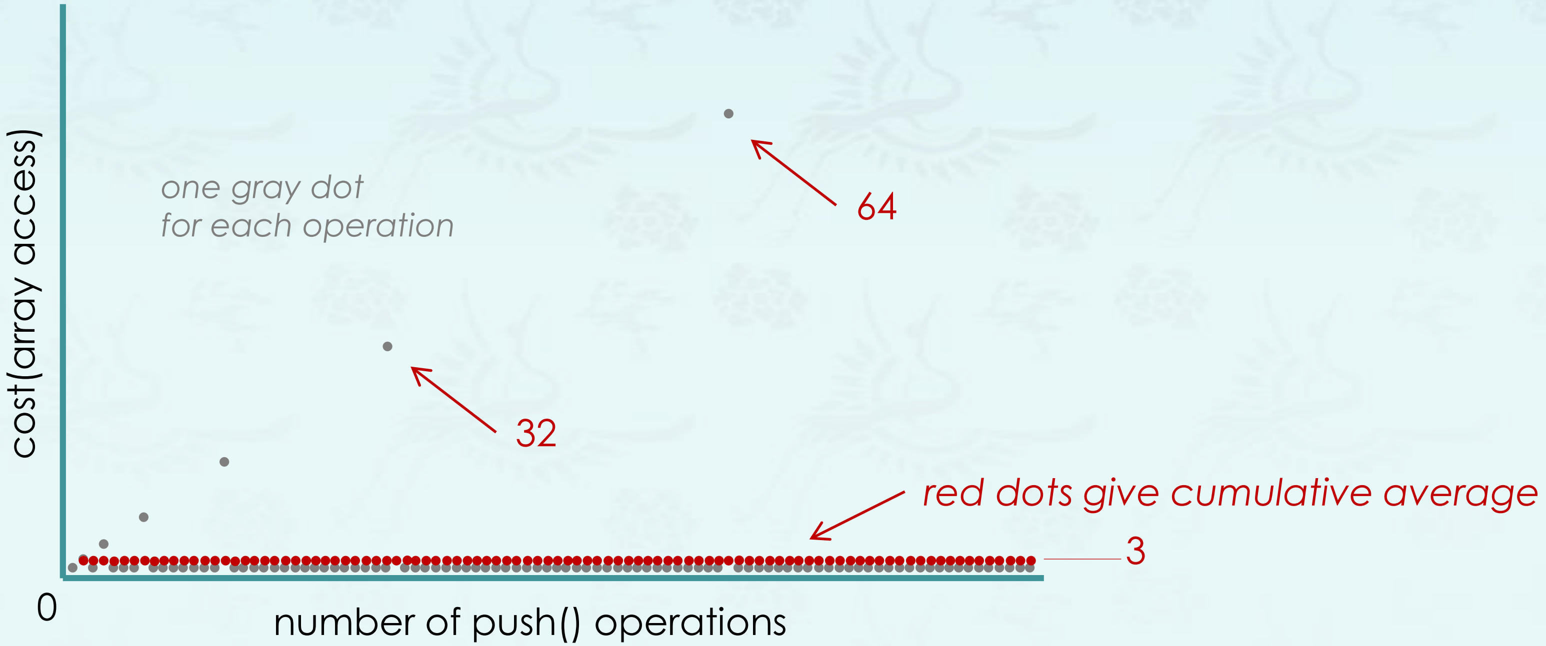
When N = 1, Capacity = 1
Cost: 1 + (0) // (?) cost to copy the existing items into the new array
When N = 2, Capacity = 1
Cost: 1 + (**1**) // (1) items to copy
When N = 3, Capacity = 2
Cost: 1 + (**2**) // (2) items to copy into the new array
When N = 4, Capacity = 4
Cost: 1 + (0) // (0) since no copy is needed
When N = 5, Capacity = 4
Cost: 1 + **(4)** // (4) items to copy into the new array
When N = 6, Capacity = 8
Cost: 1 + (0)
When N = 7, Capacity = 8
Cost: 1 + (0)
When N = 8, Capacity = 8
Cost: 1 + (0)
When N = 9, Capacity = 8
Cost: 1 + **(8)** // (8) items to copy into the new array

Stacks using dynamic arrays

Q. Cost of inserting first N items by resize(s.length * 2) ?
A. $\mathbf{T}(N) = 1 + (1 + 2 + 4 + 8 + \cdots + N) = ?$

Stacks using dynamic arrays

Q. Cost of inserting first N items by resize(s.length * 2) ?
A. $\mathbf{T}(N) = 1 + (1 + 2 + 4 + 8 + \cdots + N)$



Stacks using dynamic arrays

Q: How to shrink array?

First try.
- **push():** double size of array s[] when array is full
- **pop():** halve size of array s[] when array is one-half full.

Too expensive in worst case.
- Consider push-pop-push-pop- … sequence when array is full
- Each operation takes time proportional to N.

N= 5	to	be	or	not	to	be	*null*	*null*
N= 4	to	be	or	not				
N= 5	to	be	or	not	to	be	*null*	*null*
N= 4	to	be	or	not				

Stacks using dynamic arrays

Q: How to shrink array?

Efficient solution

- **push():** double size of array s[] when array is full
- **pop():** halve size of array s[] when array is one-quarter full.

```
public String pop() {
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4)
        resize(s.length/2);
    return item;
}
```

❖ **Invariant**. Array is between 25% and 100% full.

Stacks using dynamic arrays

Amortized analysis: Average running time per operation over a worst-case sequence of operations.

Proposition: Starting from an empty stack, any sequence of N push and pop operations takes time proportional to N.

	best	worst	amortized
construct	O(1)	O(1)	O(1)
push	O(1)	O(n)	O(1)
pop	O(1)	O(n)	O(1)
size	O(1)	O(1)	O(1)

doubling and halving operations

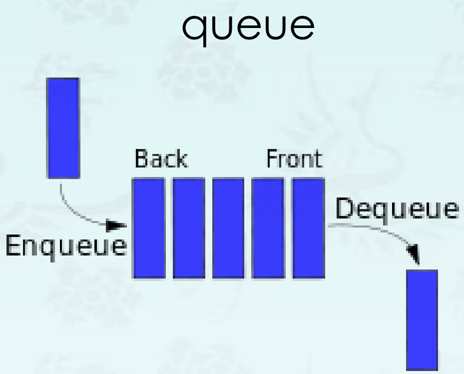
order of growth of running time
for resizing stack with N items

Data Structures

- ***stacks*** & ***queues***
  using dynamic arrays
- *some applications*

Queues

Queue: An ordered list in which **enqueues** (insertion or add) at the **rear** and **dequeues** (deletion or remove) take place at different end or **front**. It is also known as a Fist-in-first-out(FIFO) list.



❖ Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

Queues

Queue: An ordered list in which **enqueues** (insertion or add) at the **rear** and **dequeues** (deletion or remove) take place at different end or **front**. It is also known as a Fist-in-first-out(FIFO) list.

ADT Queue is
objects: a finite ordered list with zero or more elements

functions:
Queue CreateQueue(maxQueueSize)
bool full(queue, maxQueueSize)
bool empty(queue)
void add(queue, item) // Enqueue
item pop(queue) // Dequeue

Stacks

Example: Stack of strings data type (implemented in Java)

	`#include <stack>`	
	`stack<int>`	*creates an empty stack of integers*
`void`	`push(value_type& item)`	*inserts a new item onto stack*
`void`	`pop()`	*removes top item from stack (which is most recently added)*
`value_type&`	`top()`	*returns a reference to the top item*
`bool`	`empty()`	*is the stack empty?*
`int`	`size()`	*returns the number of items in the stack*

Warm-up client: Reverse sequence of strings from standard input.

Array implementation of a queue:

- Use array q[] to store items in queue.
- enqueue(): add new item at q[tail].
- dequeue(): remove item from q[head].
- Update head and tail modulo the capacity.
- Add resizing array.

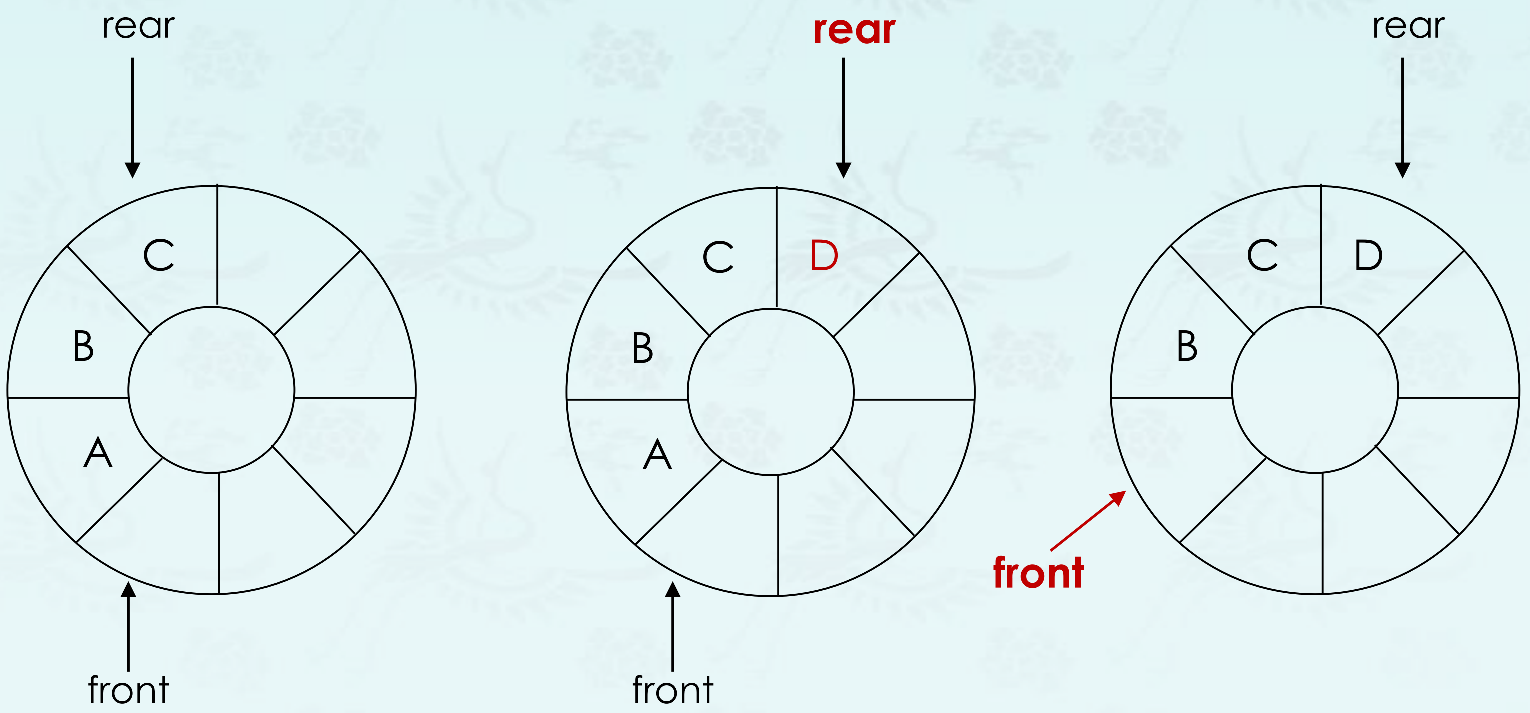
q[ ]	null	null	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
			head front				tail rear			

capacity = 10

Q. How to resize?

Circular queue:

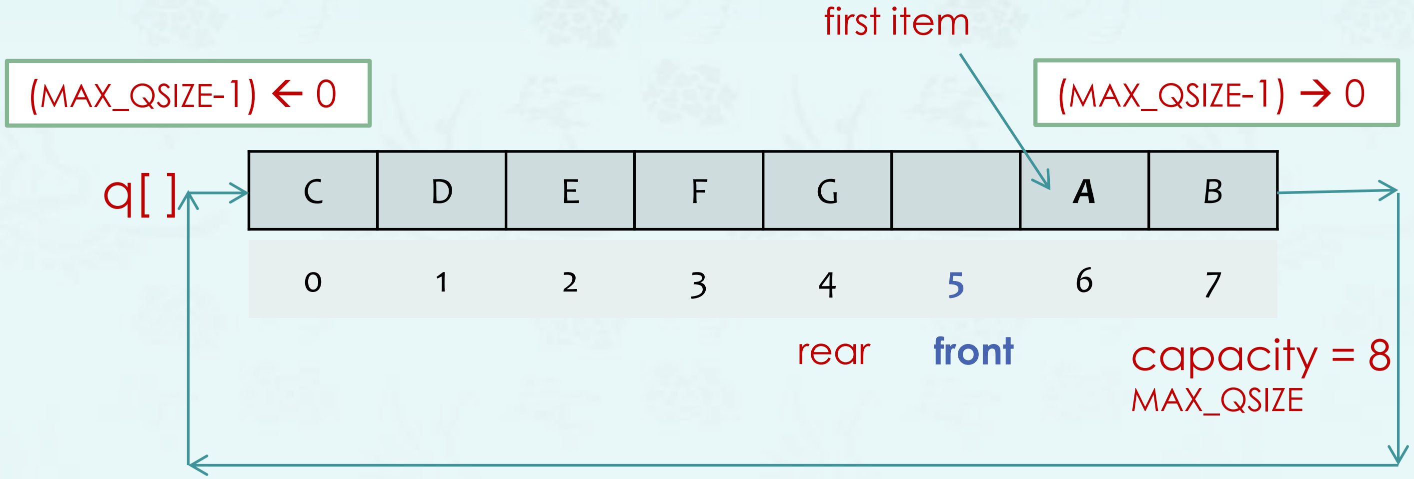
- To avoid shifting array and resizing array which is costly
- The array positions are handled in a circle rather than in a straight line.



(a) Initial (b) Addition/Enqueue (c) Deletion/Dequeue

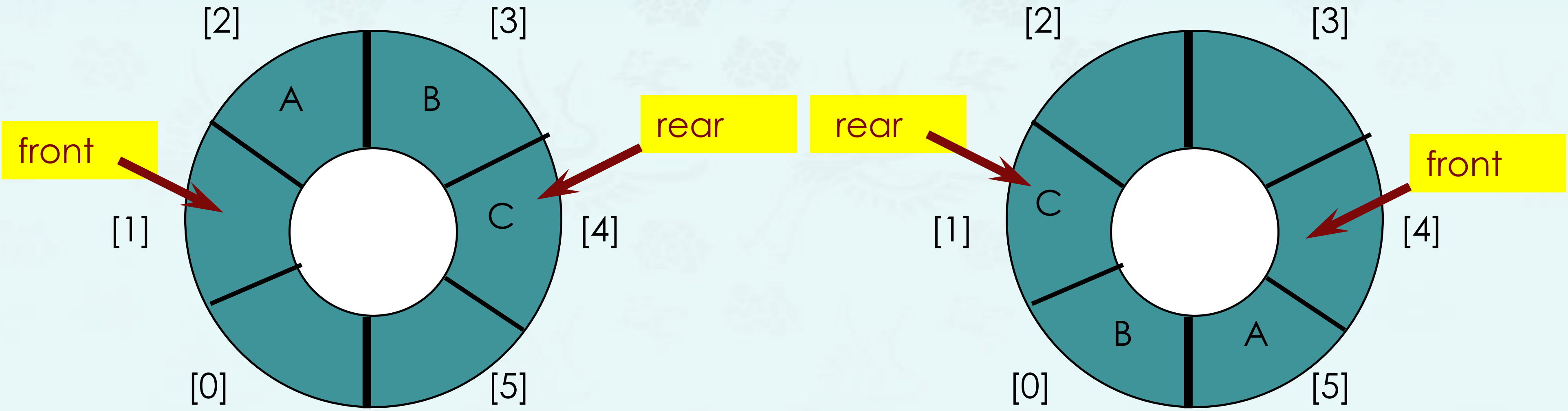
Circular queue:

- To be circular, the position next to position MAX_QSIZE-1 is 0, the position that precedes 0 is MAX_QSIZE-1
- By convention, the **first item** is located at (front + 1) position.
- If front = rear, then queue is **empty** or **full**?
  To distinguish this case, double the queue size just before it becomes full.
- The initial value for **front** and **rear** is 0.



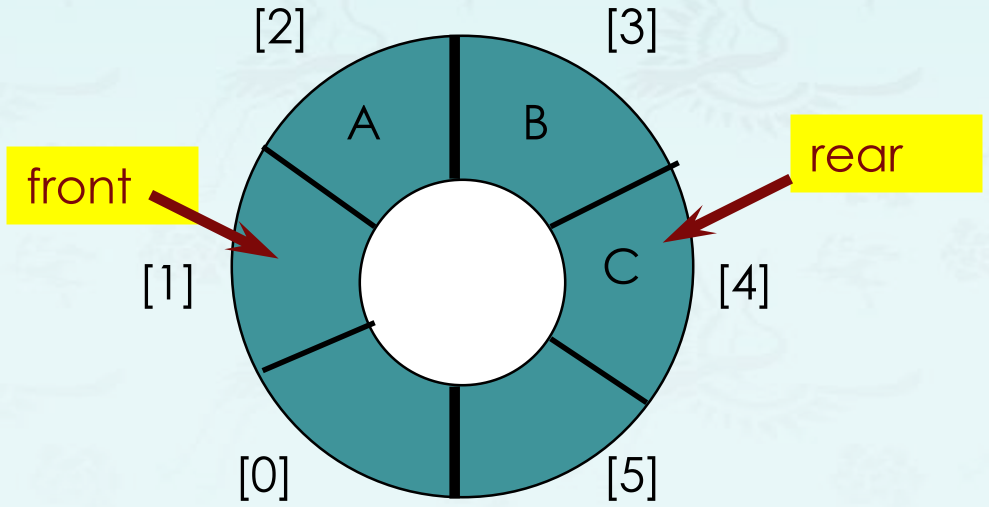
Circular queue:

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



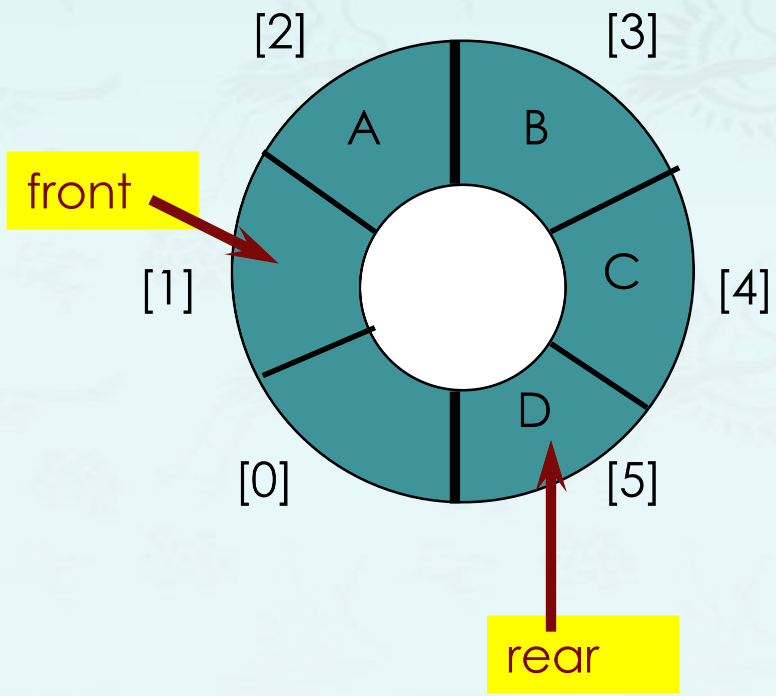
Circular queue:

- Add an element
  - Move rear one clockwise.



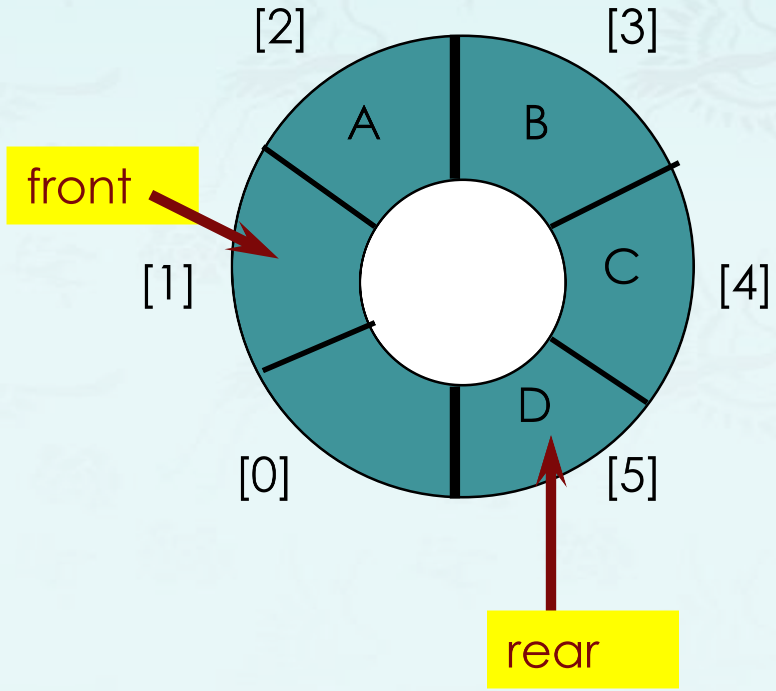
Circular queue:

- Add an element
  - Move rear one clockwise.
  - Then put into queue[rear]



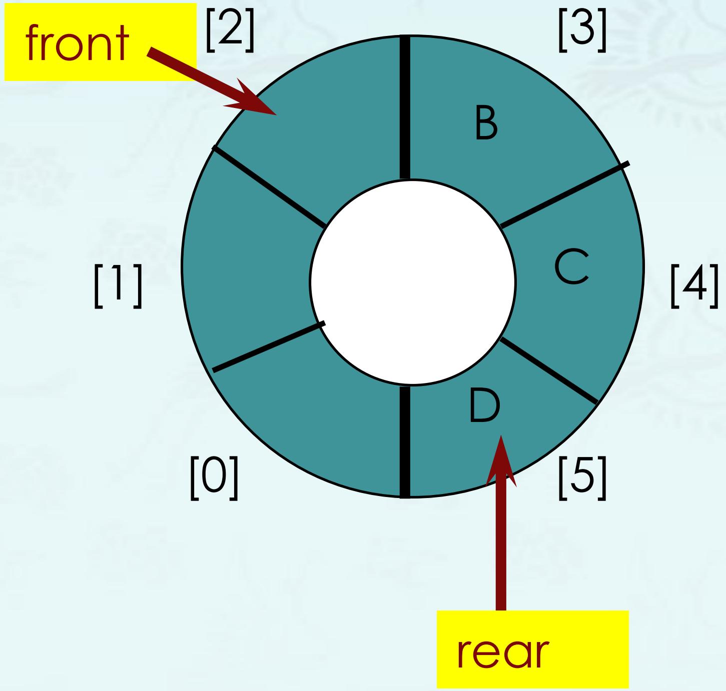
Circular queue:

- Delete an element
  - Move front one clockwise.



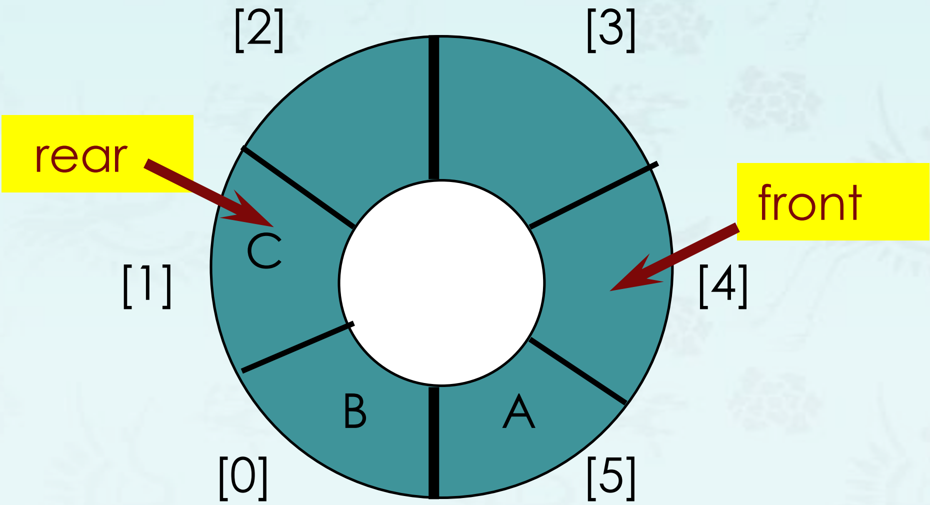
Circular queue:

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



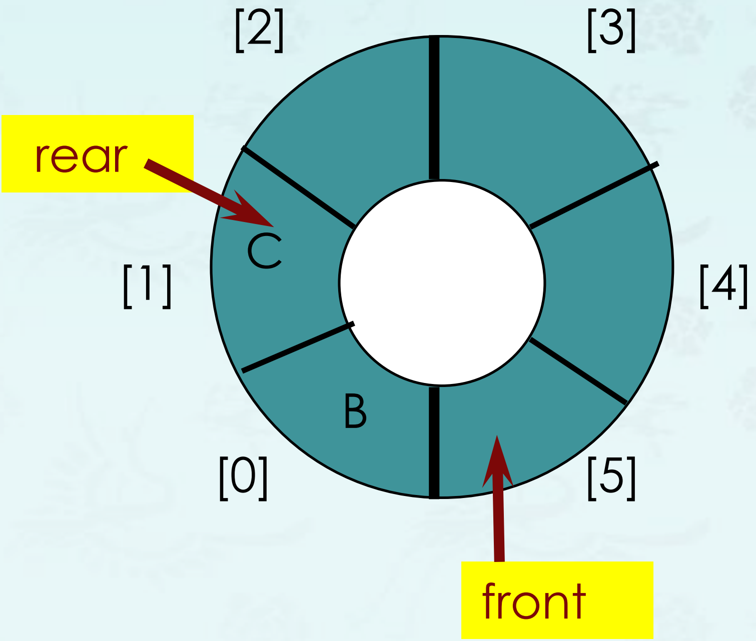
Circular queue:

- Empty that queue



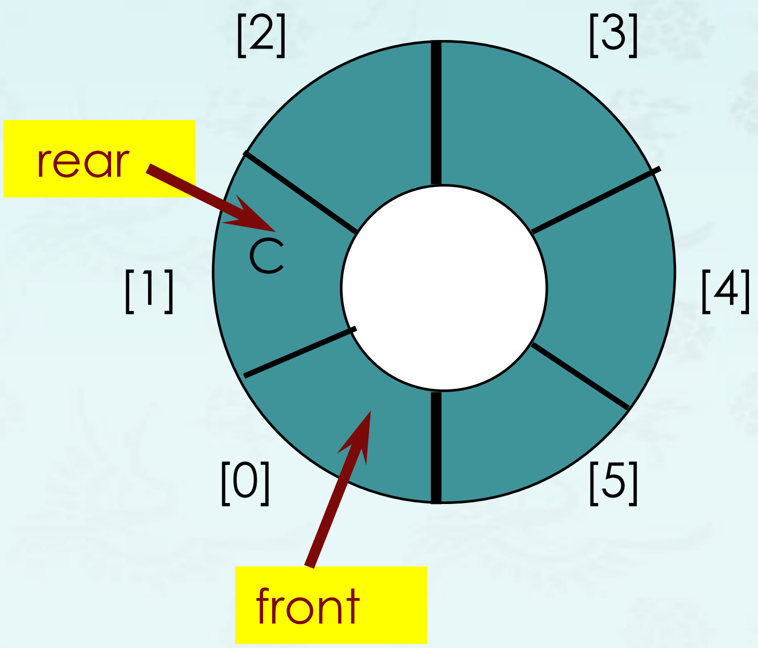
Circular queue:

- Empty that queue



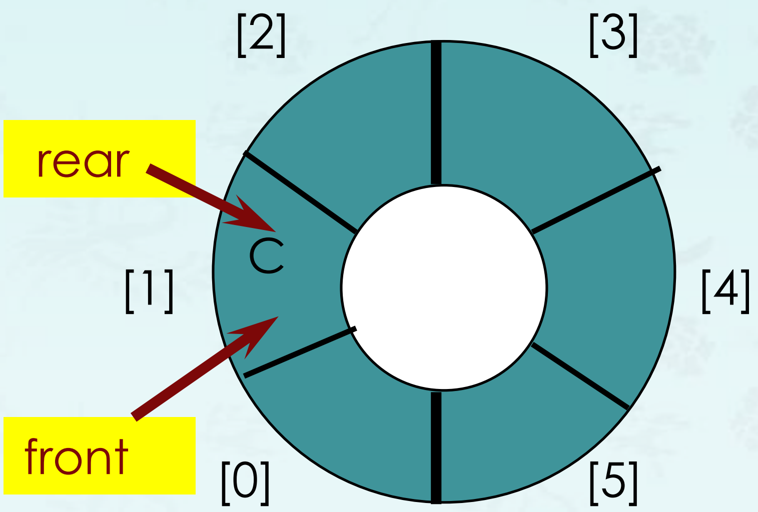
Circular queue:

- Empty that queue



Circular queue:

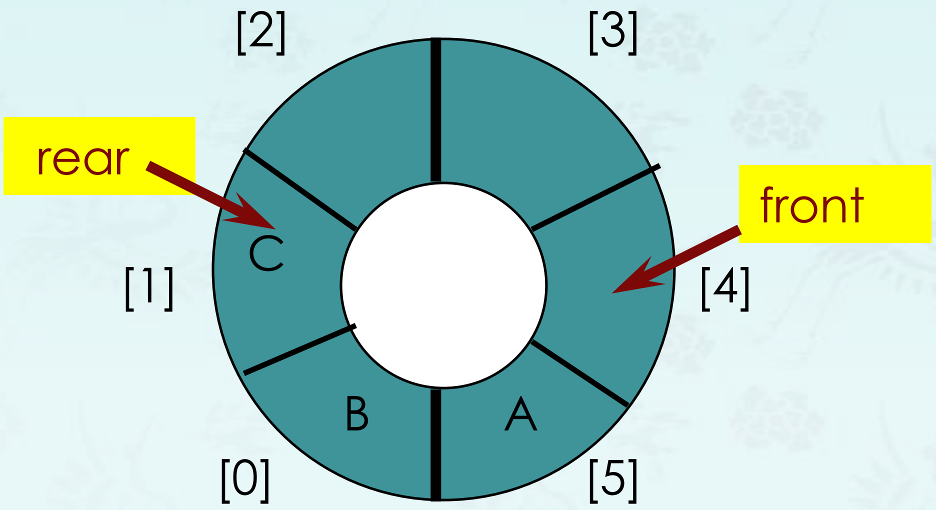
- Empty that queue



- When a series of removes causes the queue to become empty, **front = rear**.
- When a queue is constructed, it is empty.
- So initialize **front = rear = 0**.

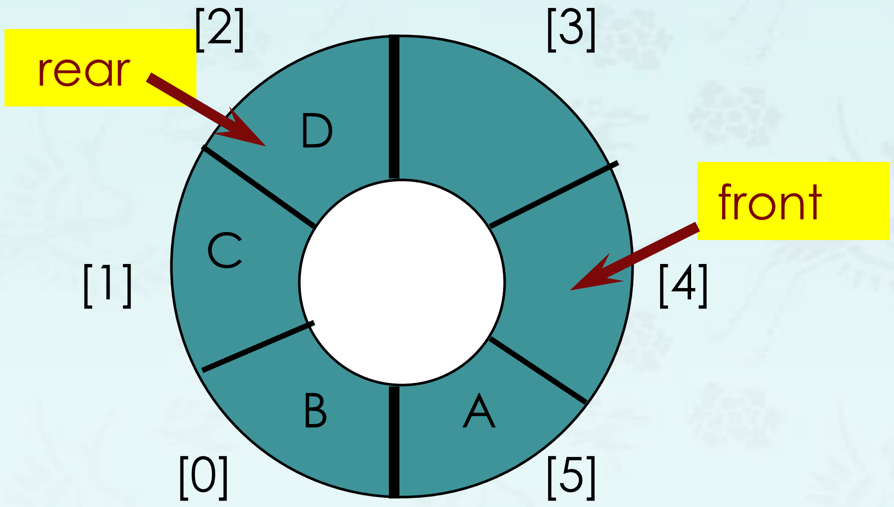
Circular queue:

- Full that queue



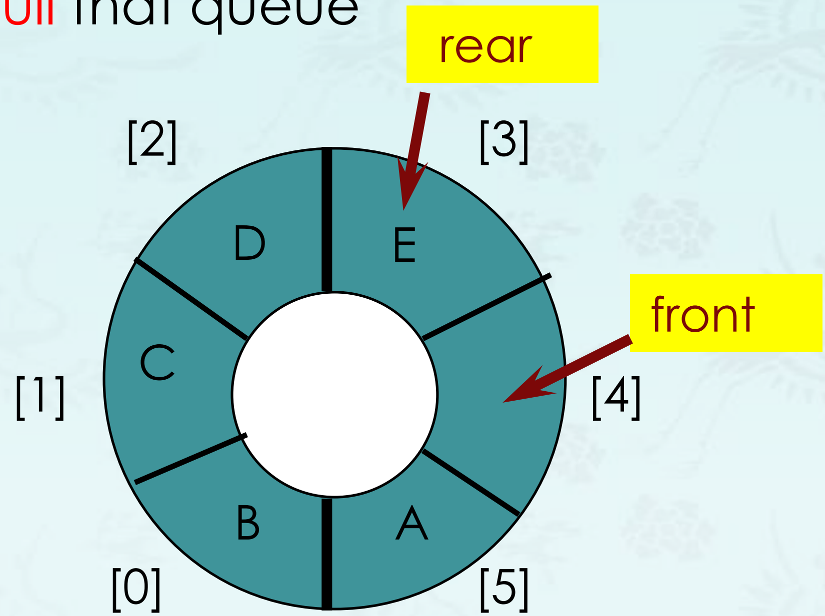
Circular queue:

- Full that queue



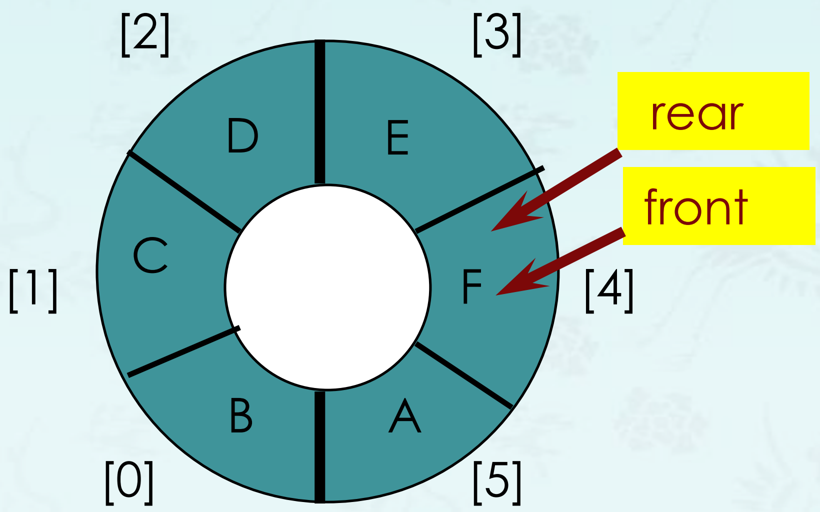
Circular queue:

- Full that queue



Circular queue:

- Full that queue



- When a series of adds causes the queue to become full, **front = rear**.
- So we cannot distinguish between a full queue and an empty queue!

Circular queue:

- Challenge: **front = rear** when queue is empty and full.
- Solutions:
  - Don't let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsAddQ.
    - Following each AddQ set this variable to true.
    - Following each DeleteQ set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsAddQ
    - Queue is full iff (front == rear) && lastOperationIsAddQ

Circular queue:

- Challenge: **front = rear** when queue is empty and full.
- Solutions: (continued)
  - Define an integer variable size.
    - Following each AddQ do size++.
    - Following each DeleteQ do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == arrayLength)

  - Performance is slightly better when first strategy is used.

1/2

Stack and Queue

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

stacks & queues using dynamic arrays
some applications – infix and postfix