

Stacks and Queues

CSE 373

Data Structures

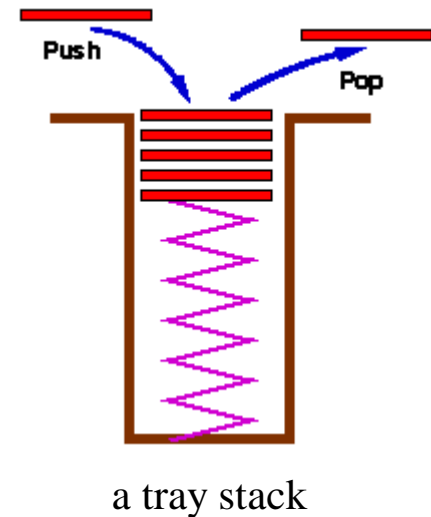
Lecture 6

Readings and References

- Reading
 - › Sections 3.3 and 3.4

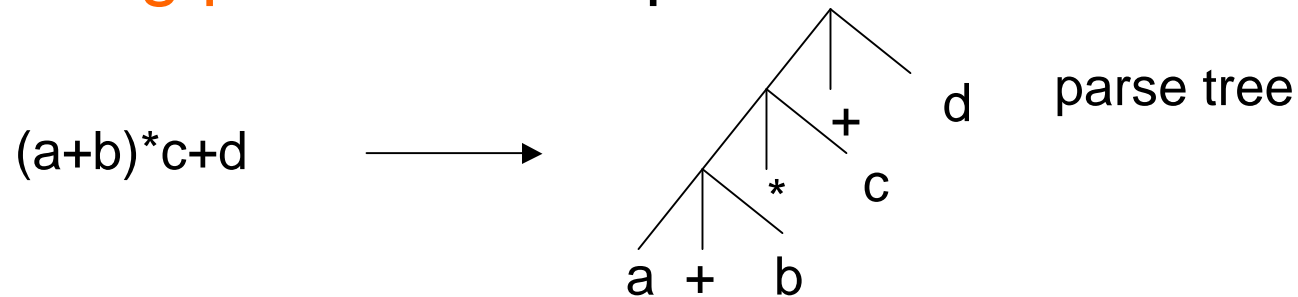
Stacks

- A list for which Insert and Delete are allowed only at one end of the list (the *top*)
 - › the implementation defines which end is the "top"
 - › LIFO – Last in, First out
- **Push**: Insert element at top
- **Pop**: Remove and return top element (aka TopAndPop)
- **IsEmpty**: test for emptiness



Important Applications of Stacks

- **Parsing phase** in compilers



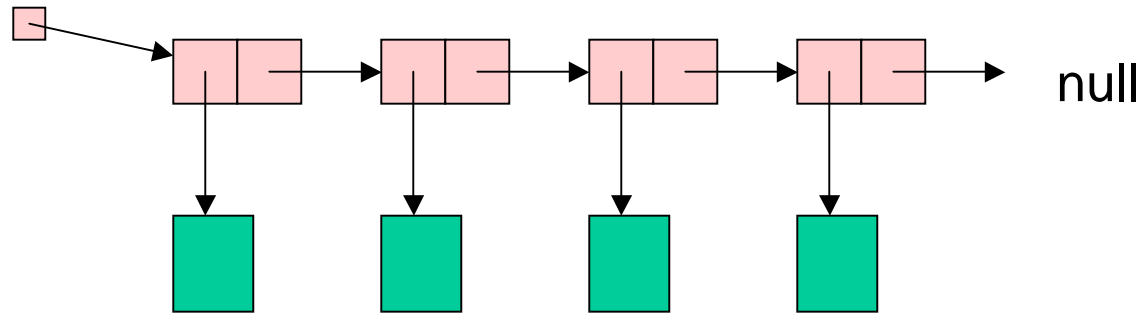
- **Call stack** in run time systems
 - › When a function (method, procedure) is called the work area (local variables, copies of parameters, return location in code) for the new function is pushed on to the stack. When the function returns the stack is popped.

Two Basic Implementations of Stacks

- **Linked List**
 - › Push is InsertFront
 - › Pop is DeleteFront
 - › IsEmpty is test for null
- **Array**
 - › The k items in the stack are the first k items in the array.

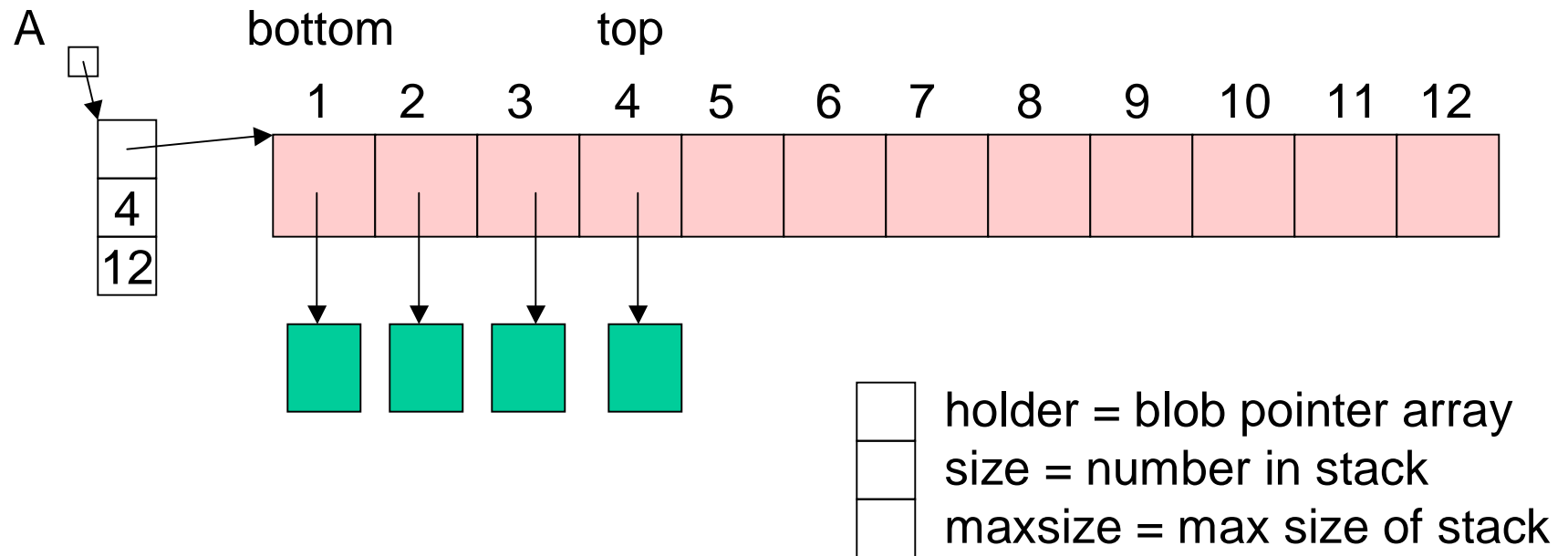
Linked List Implementation

- Stack of blobs



Array Implementation

- Stack of blobs



Push and Pop

```
IsEmpty(A : blobstack pointer) : boolean {
    return A.size = 0
}
IsFull(A : blobstack pointer) : boolean {
    return A.size = A.maxsize;
}
Pop(A : blobstack pointer) : blob pointer {
    // Precondition: A is not empty //
    A.size := A.size - 1;
    return A.holder[A.size + 1];
}
Push(A : blobstack pointer, p : blob pointer): {
    // precondition: A is not full//
    A.size := A.size + 1;
    A.holder[A.size] := p;
}
```


Linked Lists vs Array

- Linked list implementation
 - + flexible – size of stack can be anything
 - + constant time per operation
 - Call to memory allocator can be costly
- Array Implementation
 - + Memory preallocated
 - + constant time per operation.
 - Not all allocated memory is used
 - Overflow possible - Resizing can be used but some ops will be more than constant time.

Queue

- Insert at one end of List, remove at the other end
- Queues are “FIFO” – first in, first out
- Primary operations are Enqueue and Dequeue
- A queue ensures “fairness”
 - › customers waiting on a customer hotline
 - › processes waiting to run on the CPU

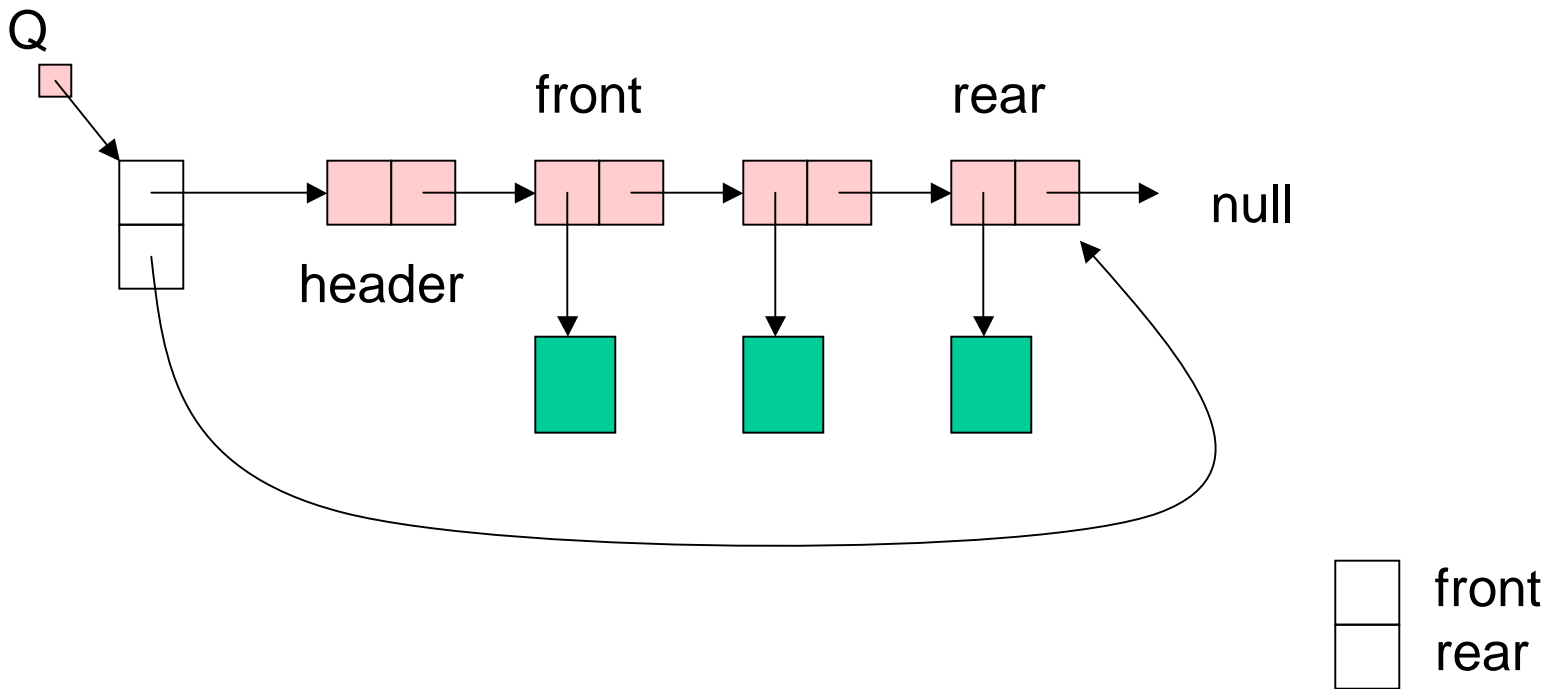
Queue ADT

- Operations:
 - › Enqueue - add an entry at the end of the queue
 - › Dequeue - remove the entry from the front of the queue
 - › IsEmpty
 - › IsFull may be needed

Applications of Queues

- File servers: Users needing access to their files on a shared file server machine are given access on a FIFO basis
- Printer Queue: Jobs submitted to a printer are printed in order of arrival
- Phone calls made to customer service hotlines are usually placed in a queue

Pointer Implementation

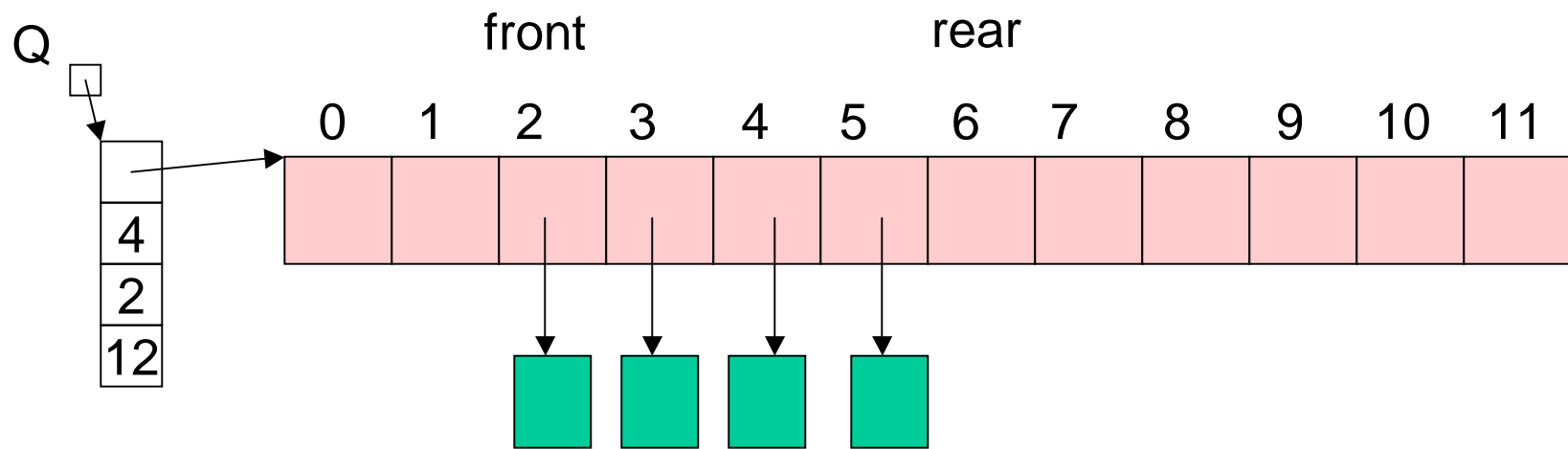


List Implementation

```
IsEmpty(Q : blobqueue pointer) : boolean {
    return Q.front = Q.rear
}
Dequeue(Q : blobqueue pointer) : blob pointer {
// Precondition: Q is not empty //
    B : blob pointer;
    B := Q.front.next;
    Q.front.next := Q.front.next.next;
    return B;
}
Enqueue(Q : blobqueue pointer, p : blob pointer): {
    Q.rear.next := new node;
    Q.rear := Q.rear.next;
    Q.rear.value := p;
}
```

Array Implementation

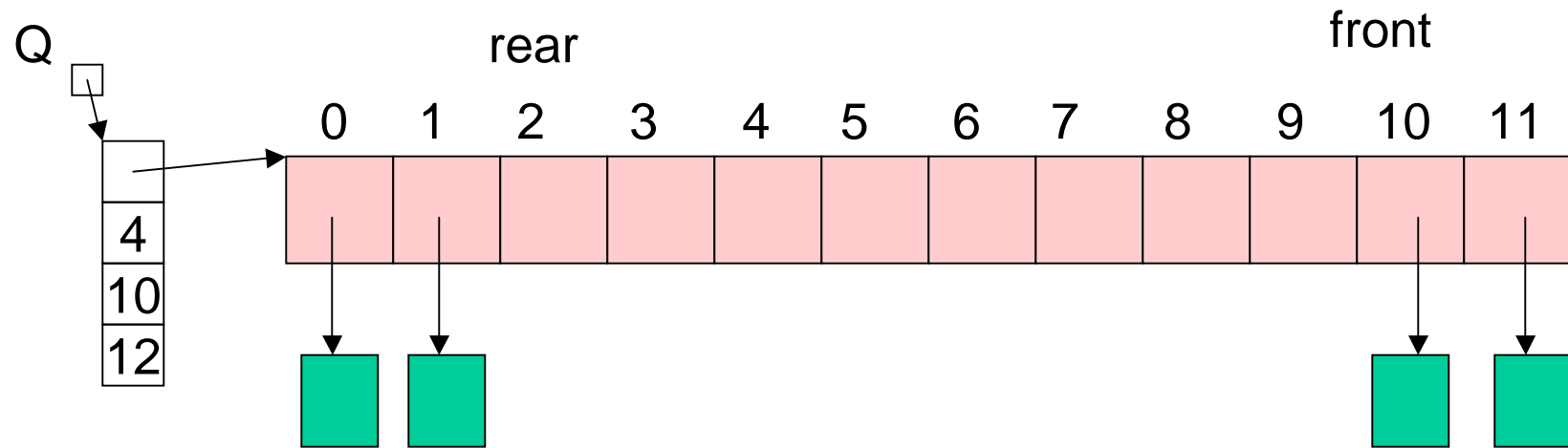
- Circular array



$$\text{rear} = (\text{front} + \text{size}) \bmod \text{maxsize}$$

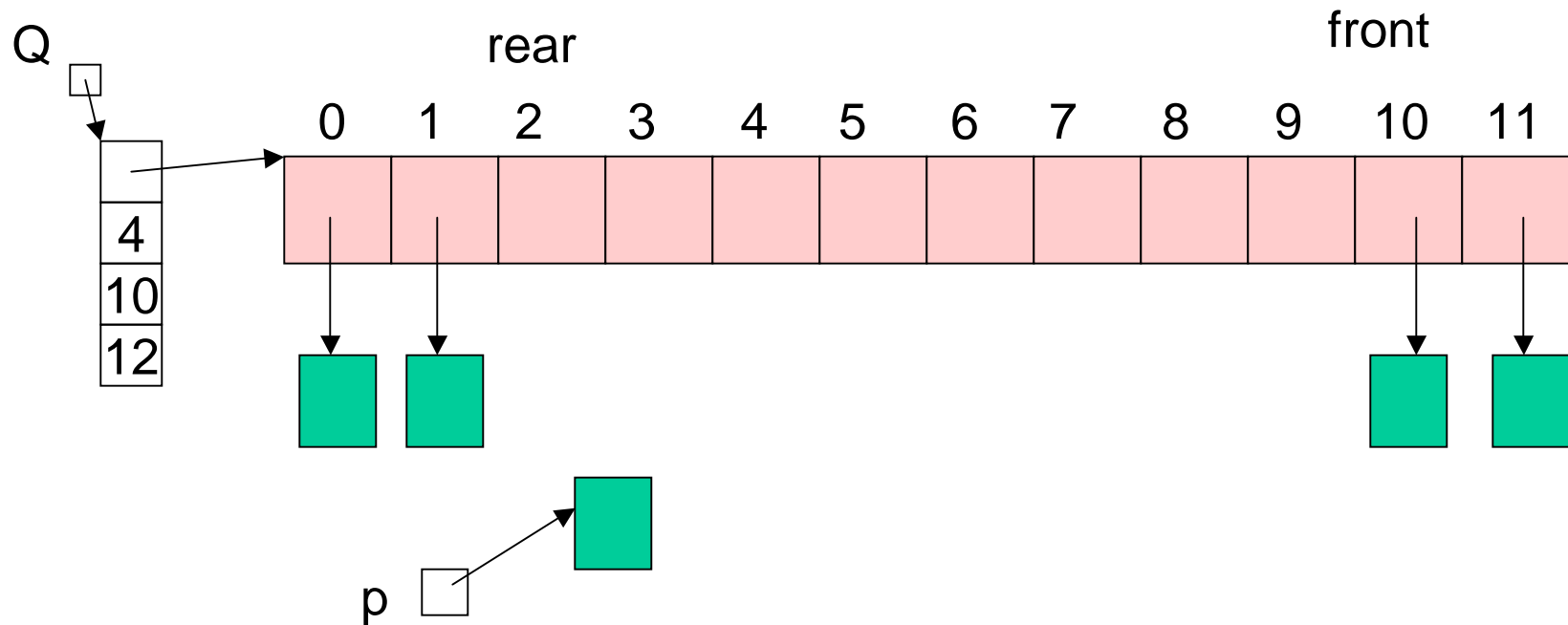
- holder = blob pointer array
- size = number in stack
- front = index of front of stack
- maxsize = max size of stack

Wrap Around

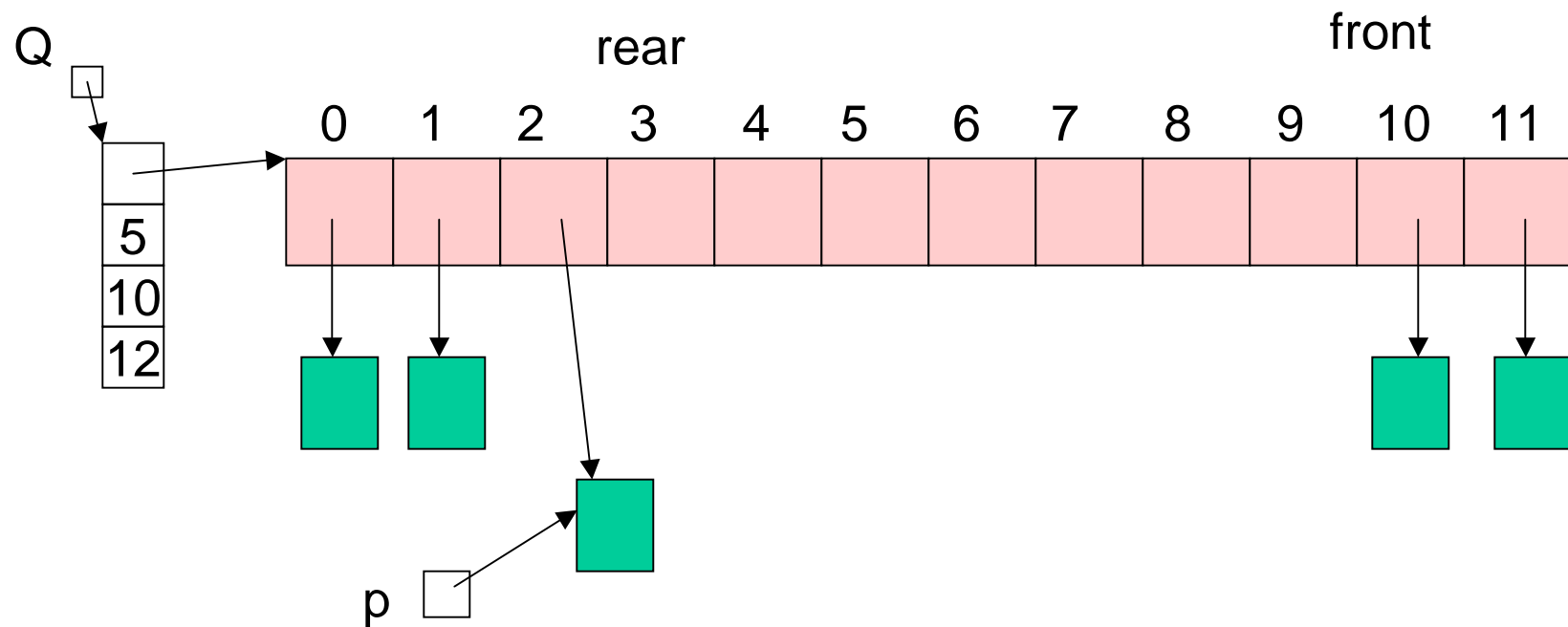


$$\begin{aligned} \text{rear} &= (\text{front} + \text{size}) \bmod \text{maxsize} \\ &= (10 + 4) \bmod 12 = 14 \bmod 12 = 2 \end{aligned}$$

Enqueue



Enqueue

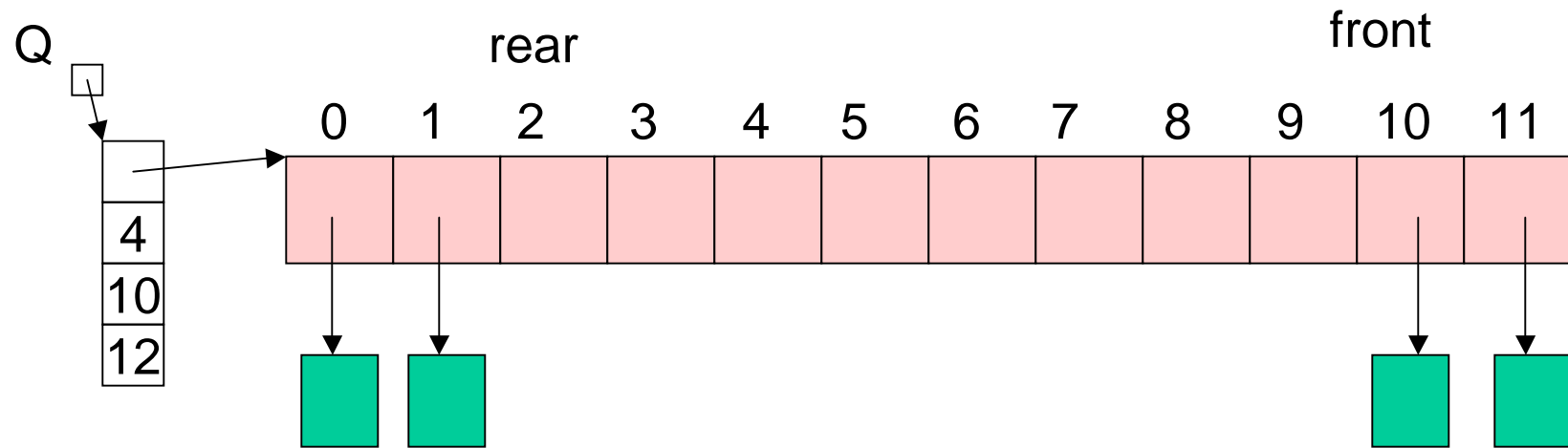


Enqueue

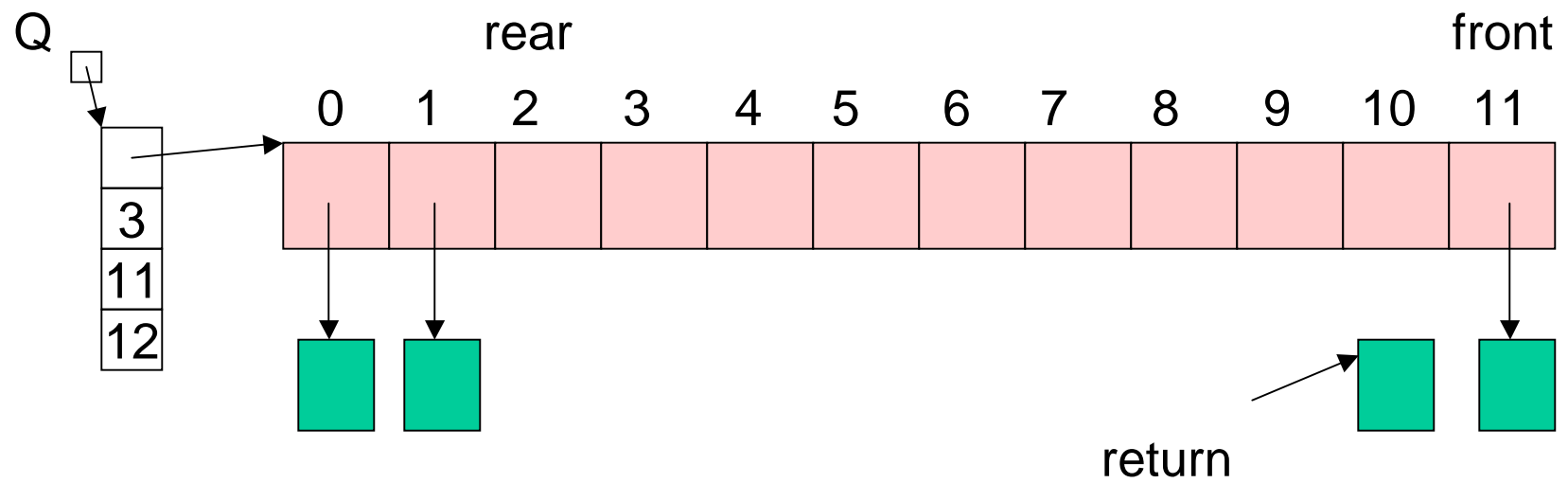
```
Enqueue(Q : blobqueue pointer, p : blob pointer) : {  
  // precondition : queue is not full //  
  Q.holder[(Q.front + Q.size) mod Q.maxsize] := p;  
  Q.size := Q.size + 1;  
}
```

Constant time!

Deque



Deque



Try Dequeue

- Class Participation
- Define the circular array implementation of Dequeue

Solution to Dequeue

```
Dequeue(Q : blobqueue pointer) : blob pointer {  
  // precondition : queue is not empty //  
  p : blob pointer  
  p := Q.holder[Q.front];  
  Q.front := (Q.front + 1) mod Q.maxsize;  
  Q.size := Q.size - 1;  
  return p;  
}
```