Data Structures
Fall 2018
Stacks & Queues

Luis de Marcos Ortega

luis.demarcos@uah.es

Contents

- Stack ADT
- Implementation of stacks
- Applications of stacks
- Queue ADT
- Implementation of queues
- Applications of queues

Bibliography

- Chapter 2 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987. “Data Structures and Algorithms.” Addison-Wesley.

Stacks

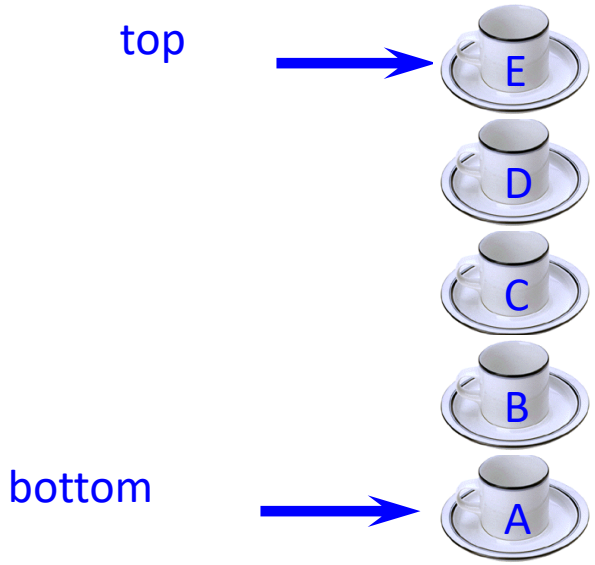
- *Linear sequence* of elements
- One end is called *top*
- Other end is called *bottom*
- *Insertion* and *deletion* of elements take place ONLY at the *top end*.
- Other names:
  - Pushdown list
  - LIFO list (LIFO – Last In, First Out)

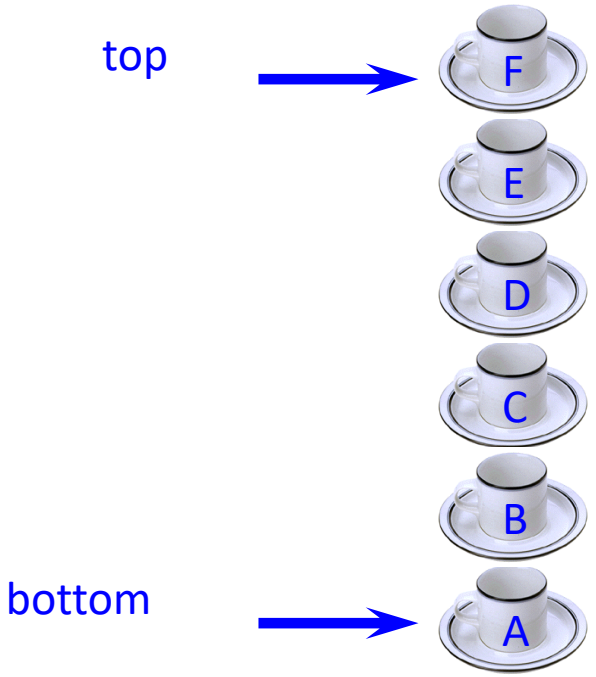
Examples of stacks

- Books on a floor
- Dishes on a shelf
- …

- where it is only convinient to remove the top object on the pile or add a new one above the top

Example of stack





- Add a cup to the stack
- Remove a cup from the stack

Operations on stacks

- *push* – Insert an element at the top of stack.
- *pop* – Delete the element at the top of stack and return it.
  - Some implementations do not return it.
- *top* – Return the element at the top of stack.
  - Sometimes also called *peek*
- *makenull* – Make the stack to be an empty stack.
- *empty* – Return true if the stack is empty, return false otherwise.

Specification

```
spec STACK[ITEM]
    genres stack, item
    operations
        push: stack item->stack
        pop: stack->item
        top: stack->item
        makenull: stack->stack
        empty: stack->boolean
endspec
```

Example (I)

- Text editor processing:
  - A character (e.g. back-space) to serve as erase character. We will use #.
    - abc##d##e → ae
  - A character (@) to serve as a kill character, whose effect is to cancel all previous characters on the current line.
    - abc@de → de
  - A text editor can process a line of text using a stack. If the character read is:
    - Neither # nor @ → push it onto the stack
    - # → pop the stack (ignore the poped element)
    - @ → make the stack empty

Example (II)

```
void edit(l:line)
     var s:stack, c:char
     s.makenull()
     while not eonl
          read(c)
          if c='#' then s.pop()
          else if c='@' then s.makenull()
          else s.push(c)
     endwhile
     print s in reverse order
endproc
```

Example: Reversing

- Stacks can be used to reverse a list of other elements (even another stack).

```
stack reverse(s:stack)
    e:item
    rs: stack
    while not s.empty()
        e=s.pop()
        rs.push(e)
    endwhile
    return rs
endfunc
```

Implementations of stacks

- Array implementation (or vector implementation)

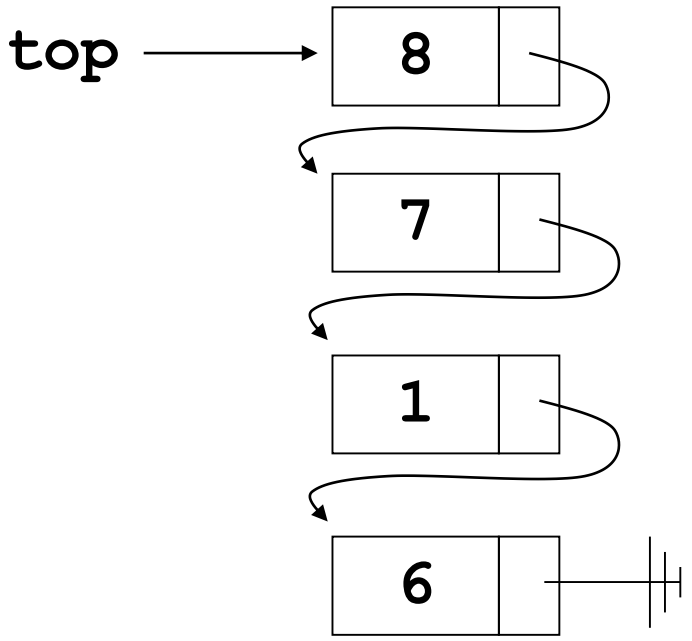
- Pointer implementation (or linked-list implementation)

Array implementation of stacks

- This implementation takes account of the fact that insertions and deletions occur only at the top.
- Anchor the bottom of the stack at the bottom of the array (high-indexed end), and
- Let the stack grow towards the top of the array (low-indexed end).
- A cursor indicates the *top*.

Pointer implementation of stacks

- Use dynamic cells that include a data element and a pointer to the next cell
- The stack is represented as a pointer to the top



Array implementation vs pointer implementation

- Ops are all constant-time operations (O(1)) in both array and pointer implementations
- For array implementation, the operations are performed in very fast constant time
- For array implementation, stack size must be defined statically (compiling-time)
  - Checks to do not overflow the stack shall be included.

Applications of stacks

- Checking expressions
- Converting expressions (e.g. infix to postfix)
- Evaluating expressions
- Method invocation and return
- Backtracking (e.g. in graphs)

- In general: stacks are useful whenever a structure/path/sequence will or could lately be done or performed on reverse order

Checking Expressions

- Example: Balancing symbols
  - To check that every right brace, bracket, and parentheses must correspond to its left counterpart
    - e.g. `[ ( ) ]` is legal, but `[ ( ] )` is illegal

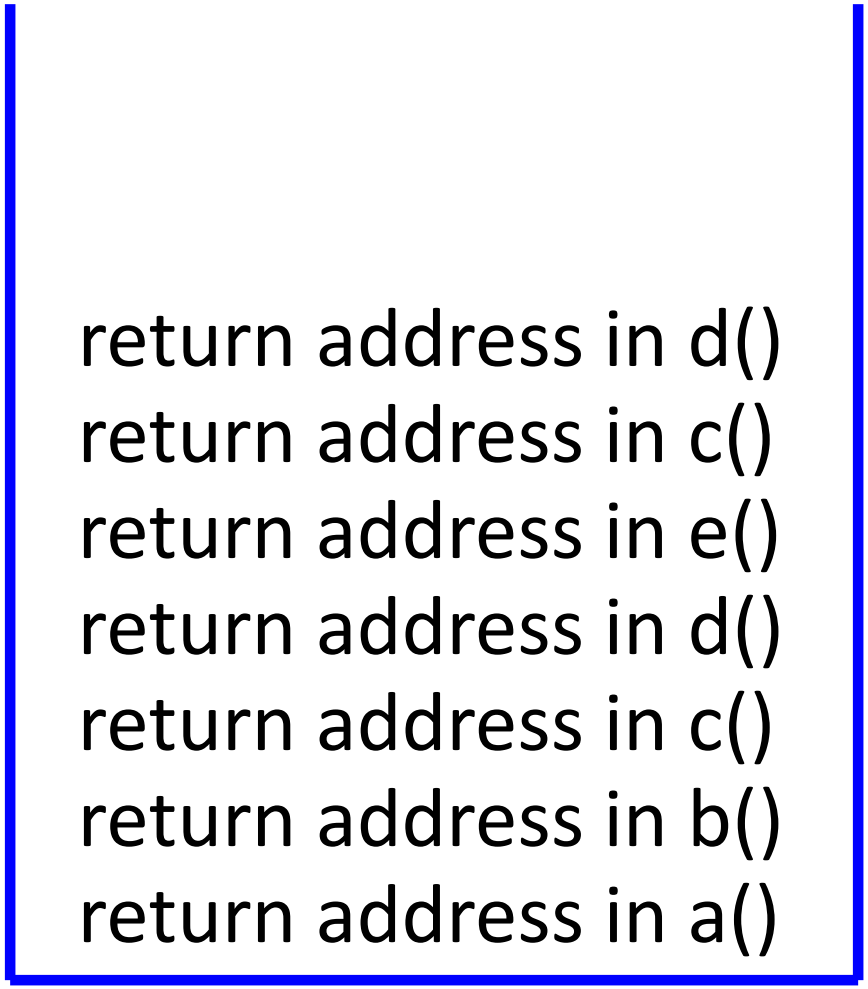
Checking Expressions

- Example: Balancing symbols

```
void checkexpression(l:line)
  var s:stack, c,d:char
  s.makenull()
  while not eonl
    read(c)
    if c=openingsymbol then s.push(c)
    if c=closingsymbol then
      if s.empty() then error()
      else
        d=s.pop()
        if d is not the corresponding closing symbol of c
        then error()
      endelse
    endif
  endwhile
endproc
```

Method invocation and return

```
public void a()
{ …; b(); …}
public void b()
{ …; c(); …}
public void c()
{ …; d(); …}
public void d()
{ …; e(); …}
public void e()
{ …; c(); …}
```

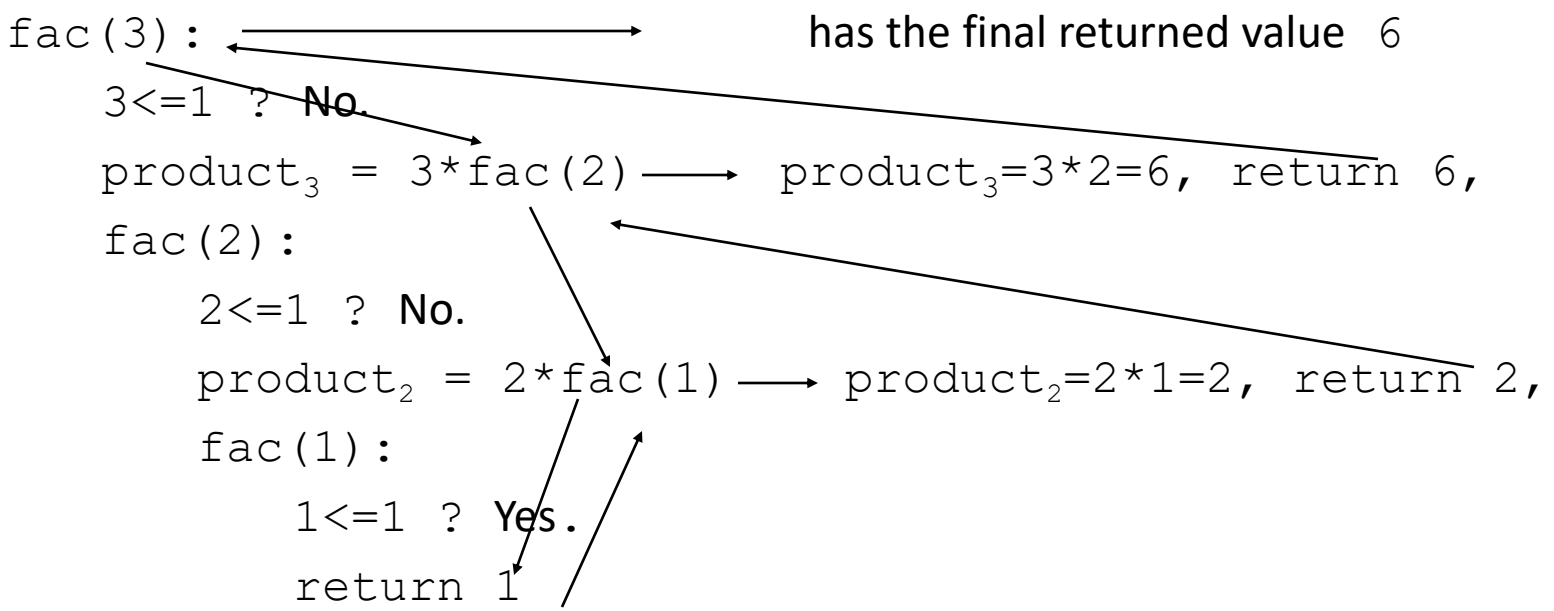


Method invocation and return

```
#include <iostream>
using namespace std;
int fac(int n){
   int product;
   if(n <= 1) product = 1;
   else product = n * fac(n-1);
   return product;
}
void main(){
   int number;
   cout << "Enter a positive integer : " << endl;;
   cin >> number;
   cout << fac(number) << endl;
}
```

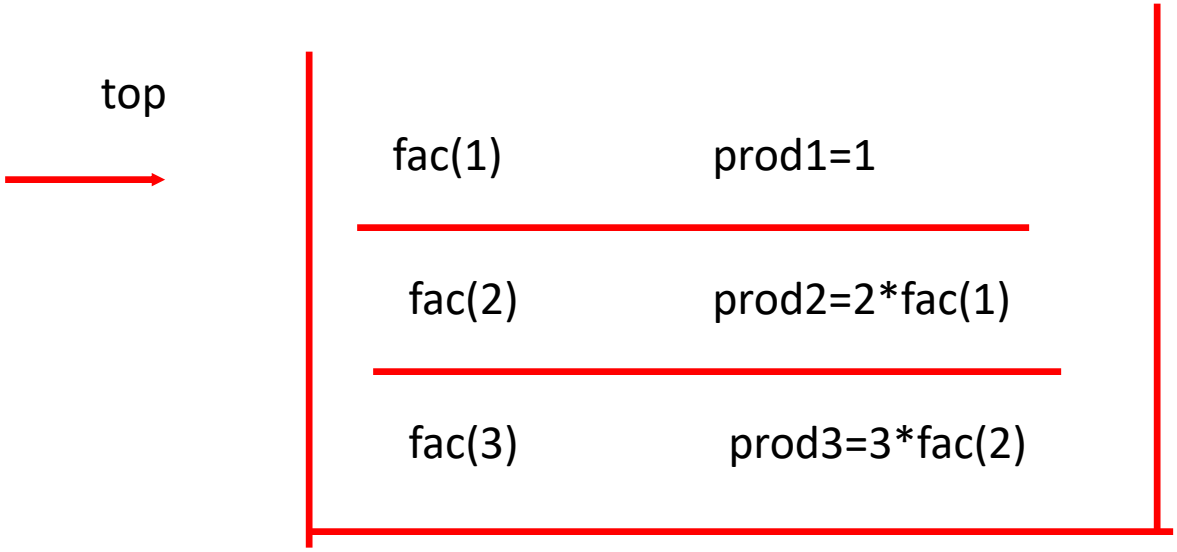
Method invocation and return

Assume the number typed is 3.



Method invocation and return

- Call is a push, return is a pop



- Program stack can be overflowed

Evaluating expressions

- Infix expression (fully parenthesized)
  - Input: Expression
    - Five types of input characters
      - Opening bracket: (
      - Numbers: 0..9
      - Operators : +, -. * and /
      - Closing bracket:)
      - New line character
  - Output: Value of the expression
  - Assumption: Expression is correct

Evaluating expressions

- Algorithm

```
real evaluateexpression(e:expresion)
  var s:stack; op1, op2, op:char
  makenull(s)
  while not eonl
    read(c)
    case
      c=opening bracket: s.push(c)
      c=number: s.push(c)
      c=operation: s.push(c)
      c=closing bracket
        op2 = s.pop()
        op = s.pop()
        op1 = s.pop()
        s.pop() {discard opening bracket}
        s.push(Evaluate(op1 op op2))
      end c=closing bracket
  endwhile
  return s.pop()
endproc
```

Evaluating expressions

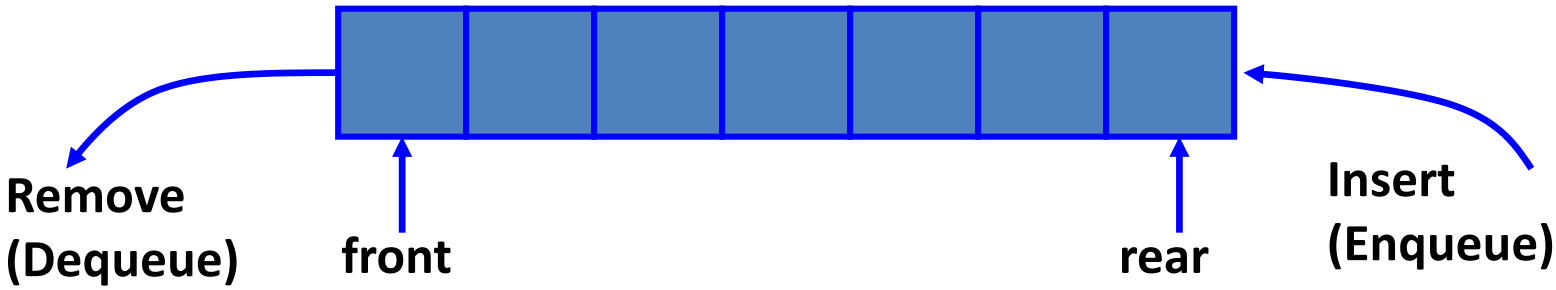
- Input: ((2 * 5) - (1 * 2))

Input Symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
2	((2	
*****	((2 *	
5	((2 * 5	
)	(10	2 * 5 = 10 and push
-	(10 -	
(	(10 - (	
1	(10 - (1	
*****	(10 - (1 *	
2	(10 - (1 * 2	
)	(10 - 2	1 * 2 = 2 & Push
)	8	10 - 2 = 8 & Push
New line	Empty	Pop & return

Queues

- *Linear sequence* of elements
- One end is called *front*
- Other end is called *rear*
- *Insertions* are done at *rear* only
- *Deletions* are made from the *front* end only
- Also known as FIFO List
  - FIFO – First in, First out

Queues



Examples of queues

- Bus stop
- Print queue
- Requests' queues
  - Web server
  - Operating system
  - …

- Any real-life queue

Queues vs stacks

- Operations for a queue are analogous to those for a stack, the substantial differences being that insertions go at the end of the list, rather than the beginning.
- Traditional terminology for stacks and queues is different

Operations on queues

- *enqueue* – Insert an element at the rear of the queue.
- *dequeue* – Delete the element at the front of queue and return it.
  - Some implementations do not return it.
- *front* – Return the element at the front of the queue.
- *makenull* – Make the queue to be an empty queue.
- *empty* – Return true if the queue is empty, return false otherwise.

Specification

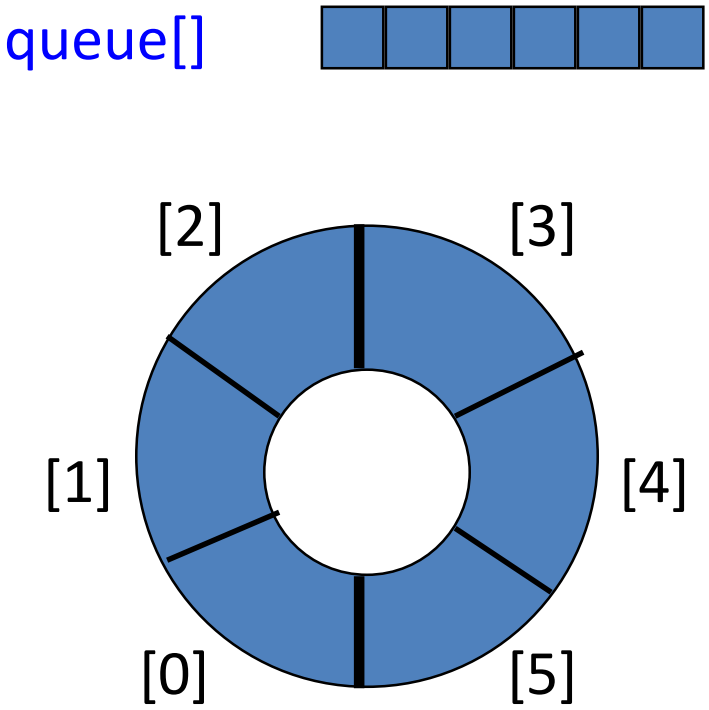
```
spec QUEUE[ITEM]
    genres queue, item
    operations
        enqueue: queue item->queue
        dequeue: queue->item
        front: queue->item
        makenull: queue->queue
        empty: queue->boolean
endspec
```

Implementations of queues

- Array implementation of queues
  - Front can be always at position 1.
  - Rear will be a cursor to the last element.
  - Enqueue will take O(1).
  - Dequeue will take O(n).
- Circular array implementation of queues
- Pointer implementation of queues

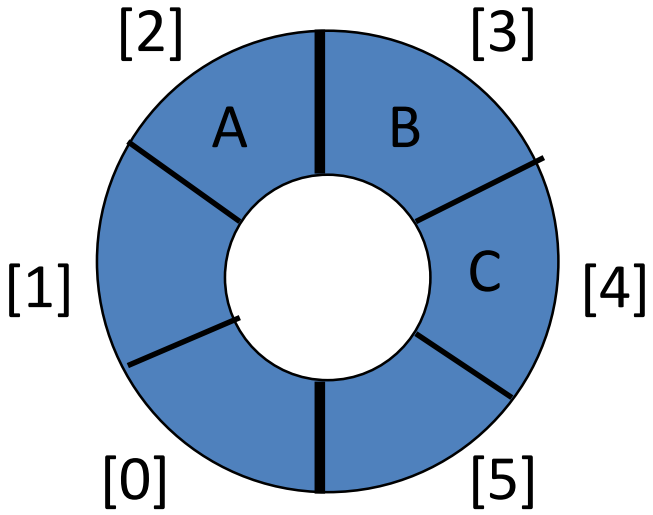
Circular array implementation of a queue

- Think of an array as a circle



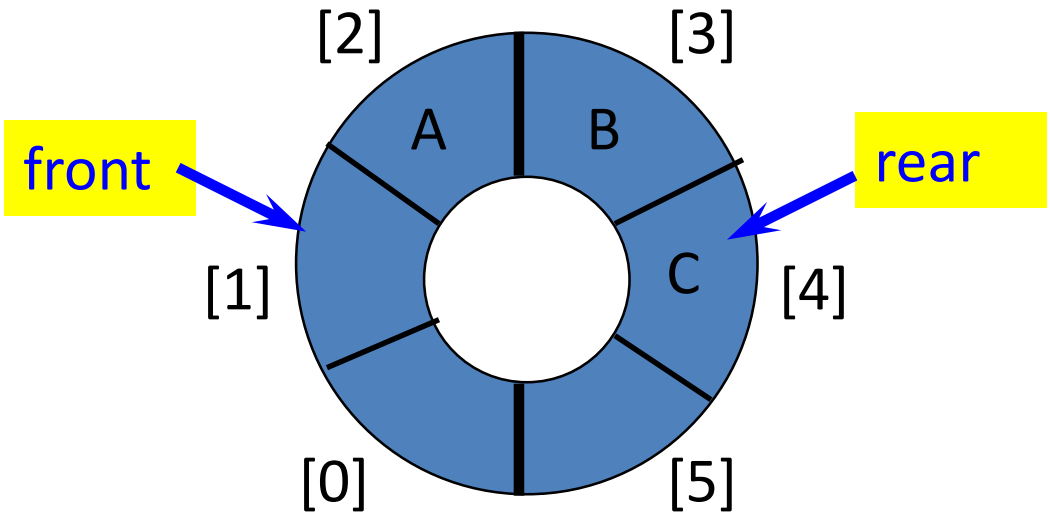
Circular array implementation of a queue

- The queue is found somewhere around the circle in conscutive positions, with the rear of the queue somewhere clockwise from the front.



Circular array implementation of a queue

- Use cursors for front and rear
  - Front is one position counterclokwise from first element
  - Rear gives the position of the last element

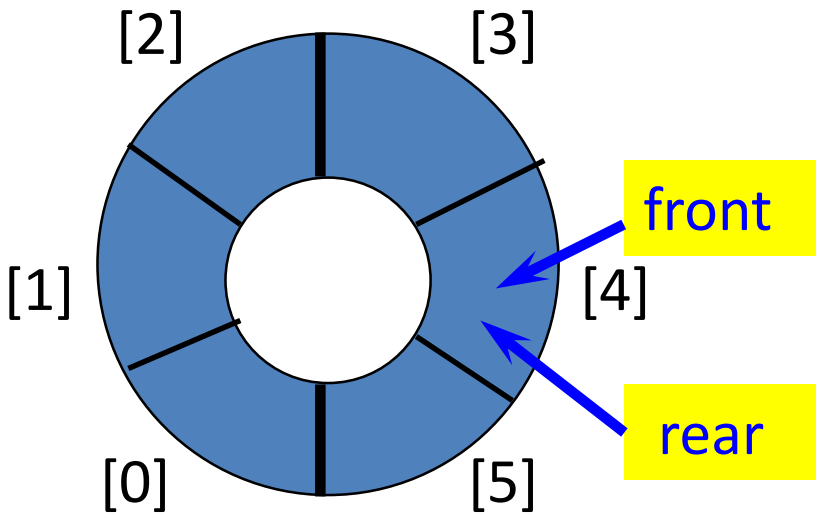


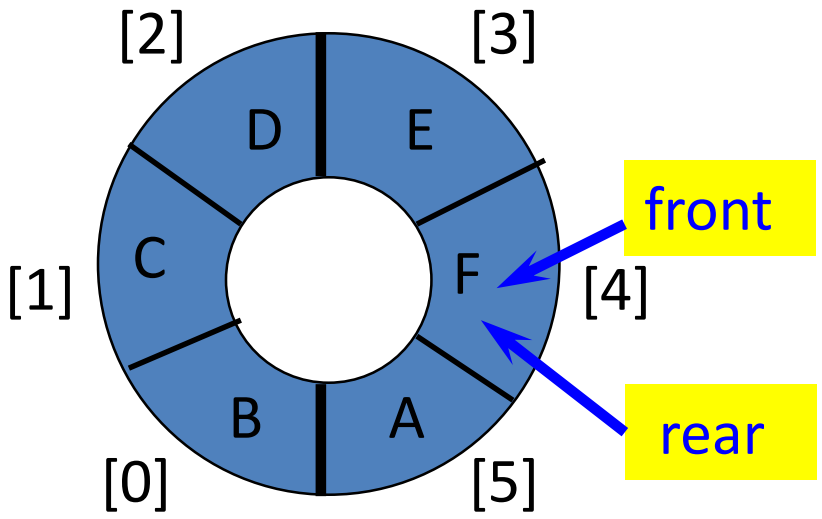
Circular array implementation of a queue

- To enqueue an element:
  - Move the rear clockwise
  - Then put the new element in queue[rear]
  - O(1)
- To dequeue an element:
  - Move the front clockwise
  - Then extract from queue[front]
  - O(1)

Circular array implementation of a queue

- Problem: No way to tell an empty queue from one that occupies the entire circle



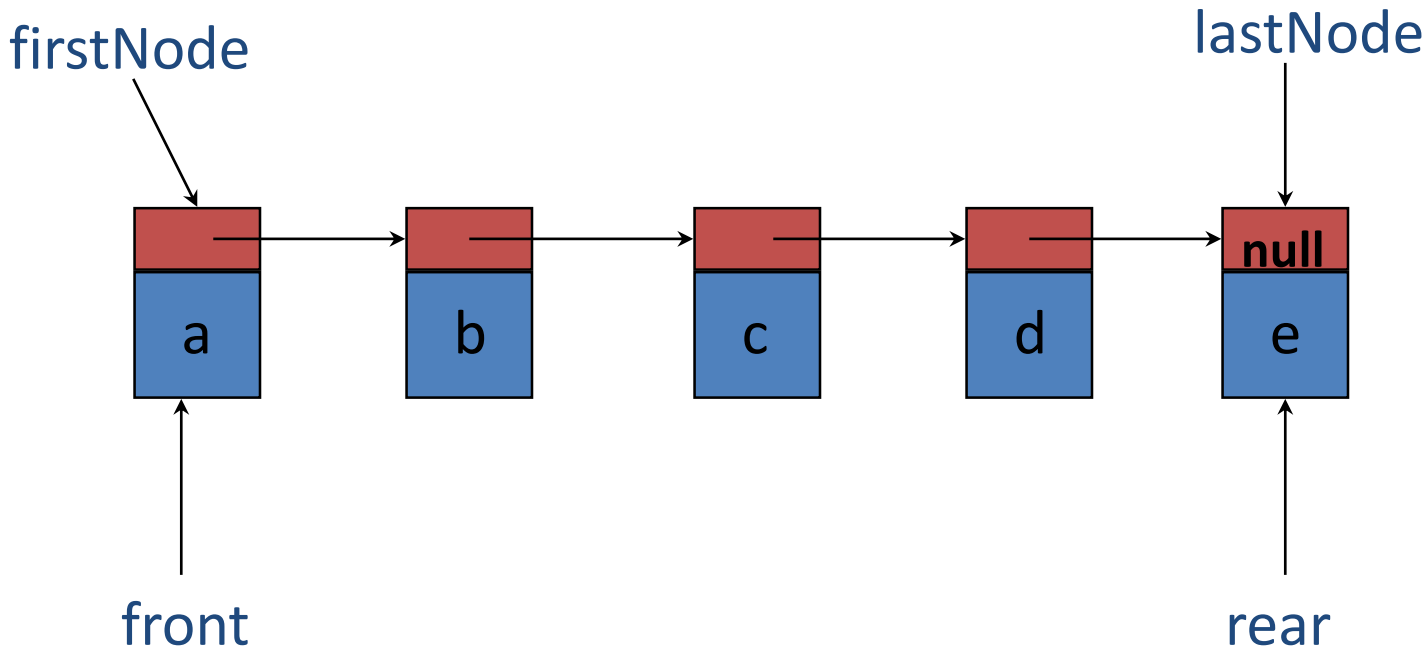


Circular array implementation of a queue

- Problem: No way to tell an empty queue from one that occupies the entire circle
- Remedies:
  - Don’t let the queue get full
  - Use a boolean variable
    - That can be true if and only if the queue is empty

Pointer implementation of queues

- Keep pointers to the front and rear elements



Applications of queues

- Queues provide many services in computer science, transport, logistics, operations research… where various entities such as data, objects, persons, or events are stored and held to be processed later.
- Anything served on first-come first-serviced basis can be modeled as a queue

Stacks & Queues

Luis de Marcos Ortega

luis.demarcos@uah.es