# Stack and Queue

Wahyono, Ph.D.
Department of Computer Science and Electronics
Faculty of Mathematics and Natural Sciences
Universitas Gadjah Mada, Yogyakarta, Indonesia
Email: wahyo@ugm.ac.id

# Two New ADTs

- Define two new abstract data type
  - Both are restricted lists
  - Can be implemented using arrays or linked list
- Stacks
  - “Last In First Out” (LIFO)
- Queues
  - “First In First Out” (FIFO)

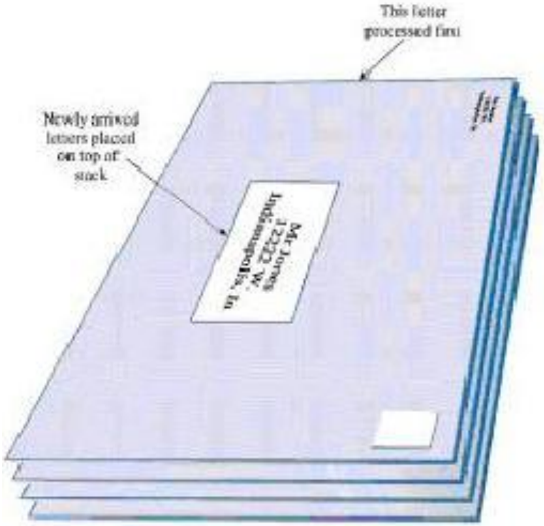
# Stack

- A stack only allows access to the last item inserted
- To get the second-to-last, remove the last
- A stack is what is known as a Last-In, First-Out (LIFO) structure
  - We can only insert to the top (push)
  - We can only access the element on top (peek)
  - We can only delete from the top (pop)

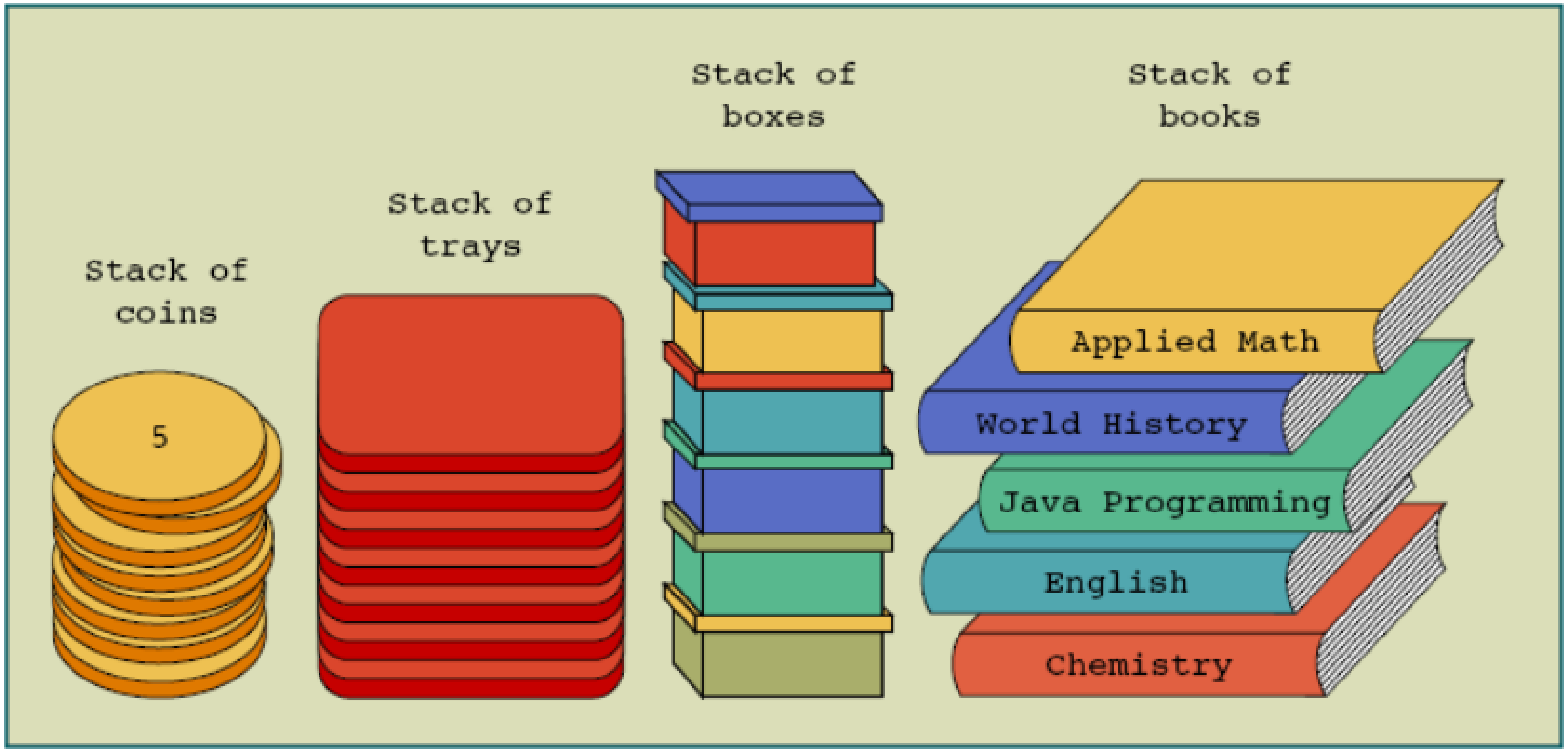
# Performance Implication

- Analogy: US Postal Service
- It is critical that we are able to process mail efficiently
- Otherwise what happens to the letters on the bottom?

If we receive a stack of mail, we typically open the top one first, pay the bill or whatever, then get rid of it and open the second



# Stack Samples

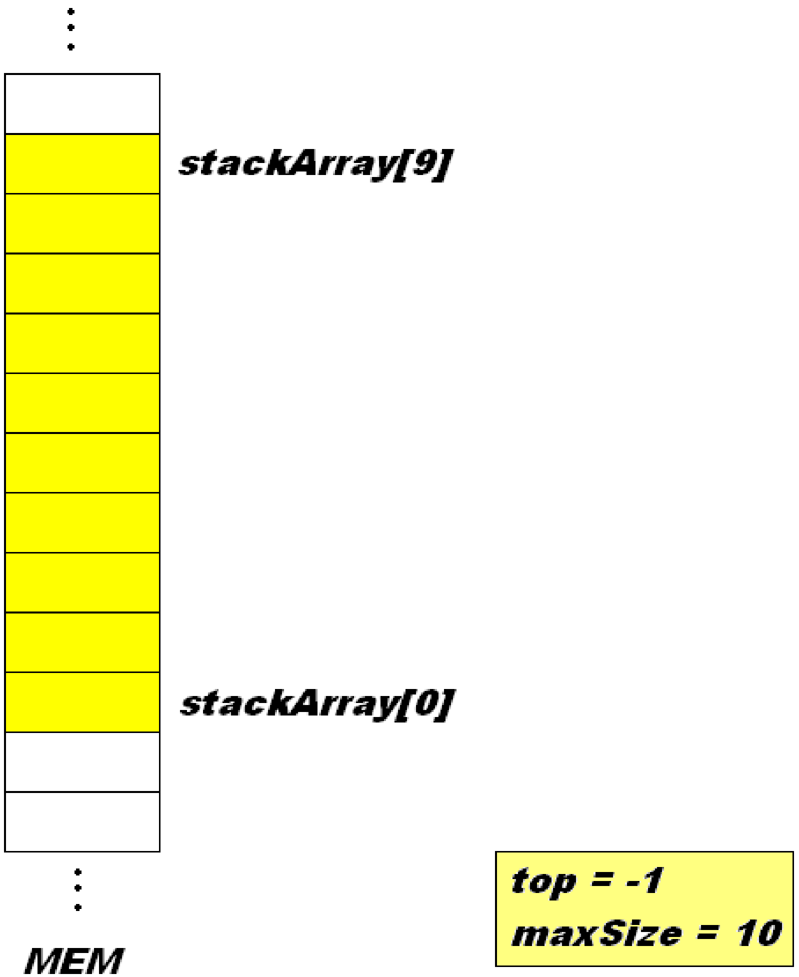


# Stack class methods

- Constructor:
  - Accepts a size, creates a new stack
  - Internally allocates an array of that many slots
- push()
  - Increments top and stores a data item there
- pop()
  - Returns the value at the top and decrements top
  - Note the value stays in the array!  It's just inaccessible (why?)
- peek()
  - Return the value on top without changing the stack
- isFull(), isEmpty()
  - Return true or false

# Pictorally, let's view the execution of main()

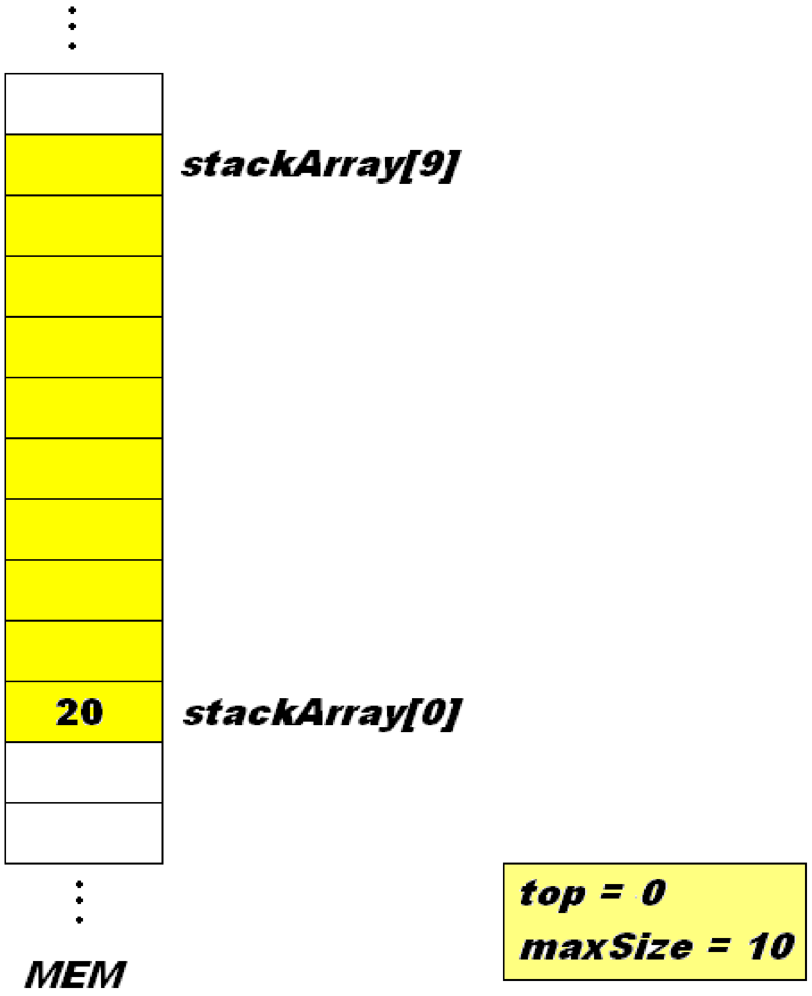
- StackX theStack =
  new StackX(10);



# Push

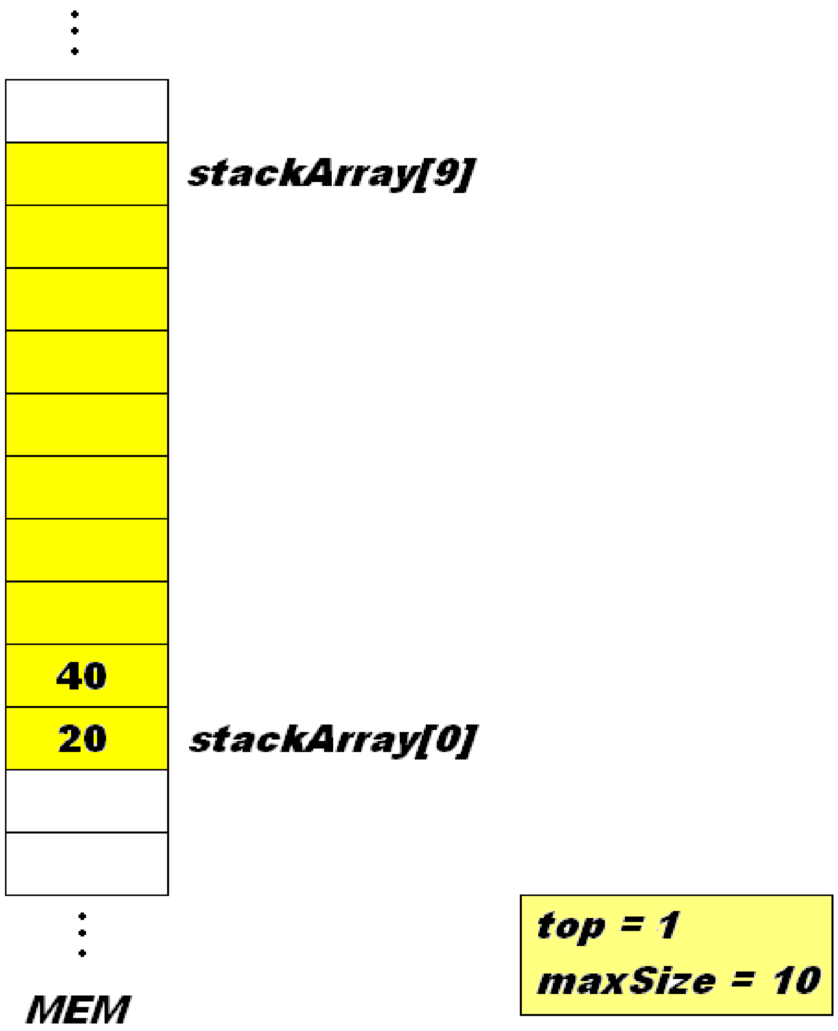
- theStack.push(20);

- **top** gets bumped up
- 20 gets stored in the slot with index **top**



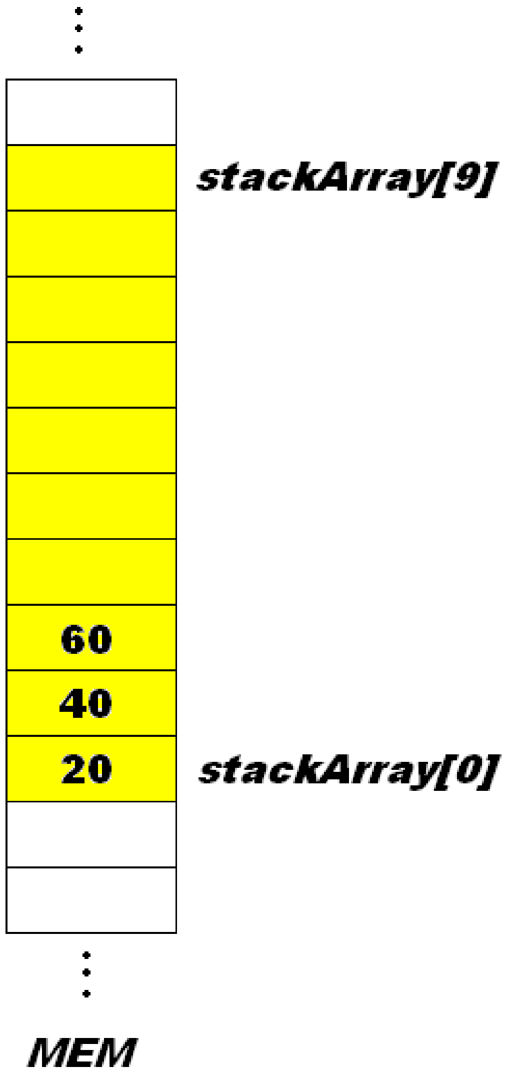
- theStack.push(40);

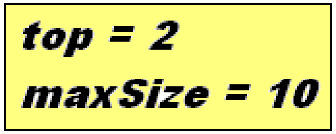
- **top** gets bumped up
- 40 gets stored in the slot with index **top**



- theStack.push(60);

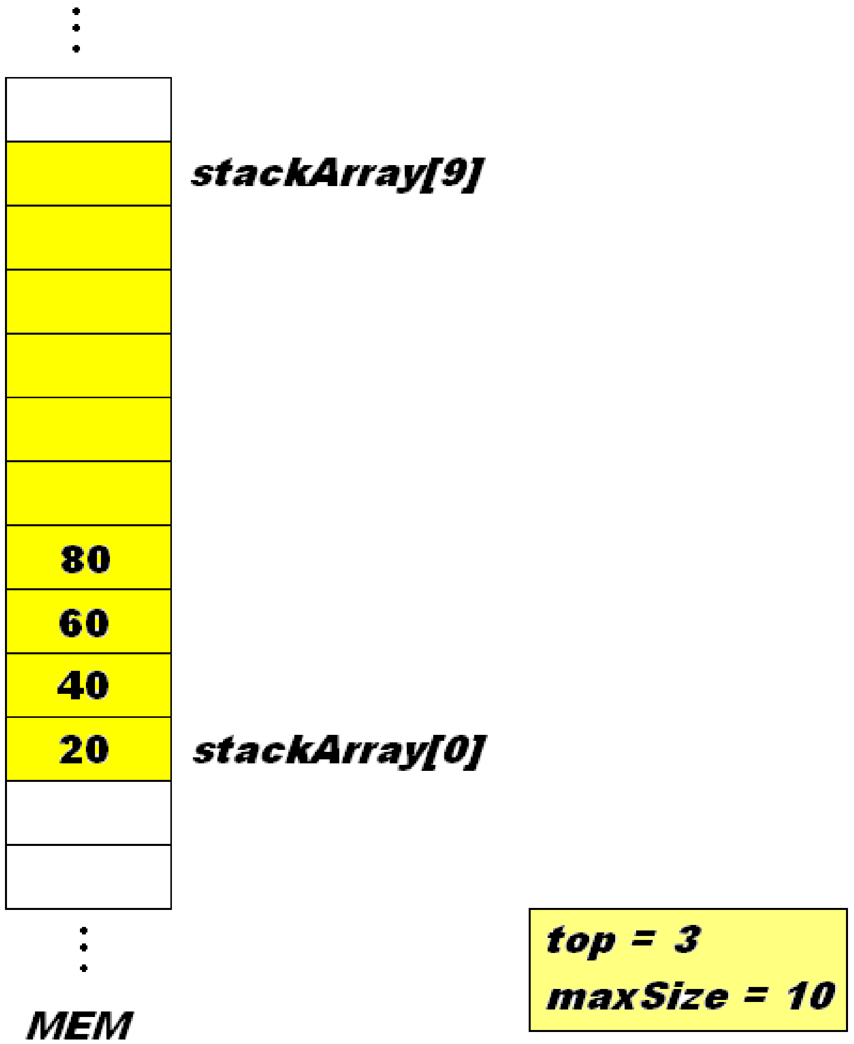
- **top** gets bumped up
- 60 gets stored in the slot with index **top**



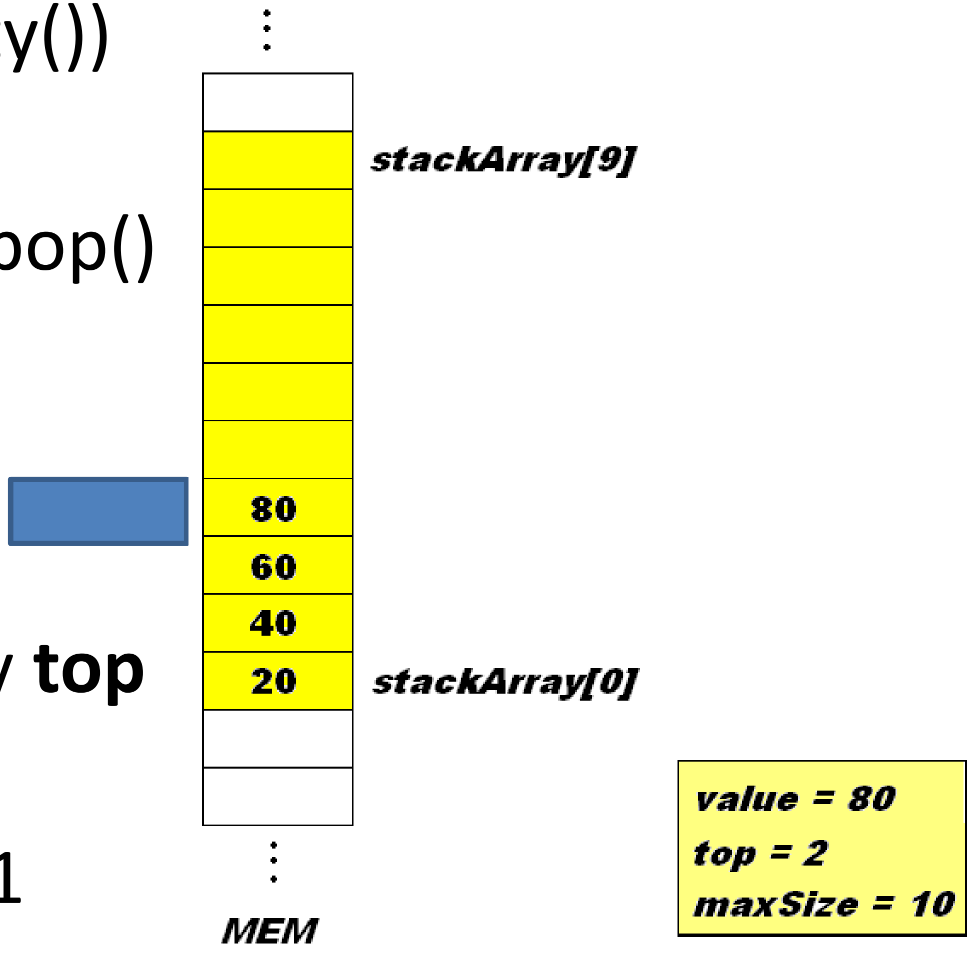
- theStack.push(80);

- **top** gets bumped up
- 80 gets stored in the slot with index **top**



# Pop

- while (!theStack.isEmpty())
  {
    long value = theStack.pop()
    …




- The element indexed by **top** is stored in **value**
- **top** is decremented by 1

# Print

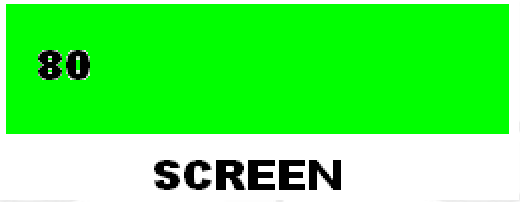
- while (!theStack.isEmpty())

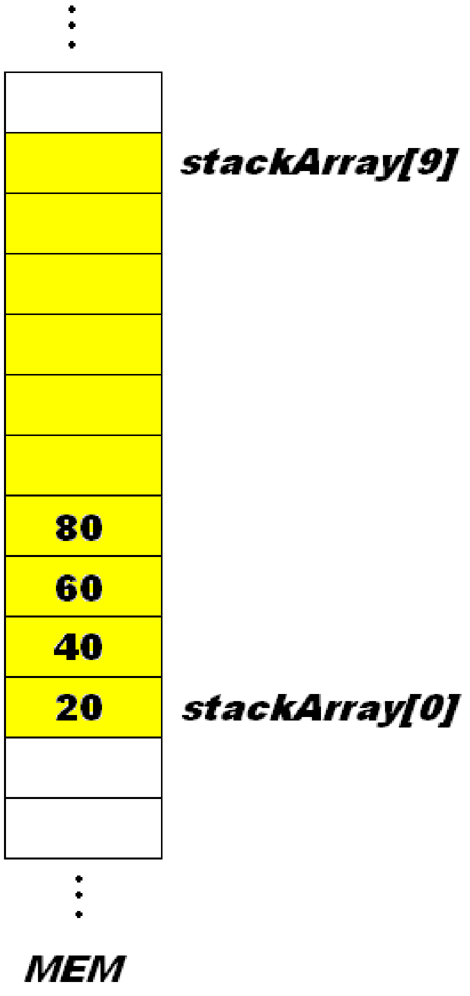
  {

  …

  System.out.print(value)

  System.out.print(“”)




*value = 80*

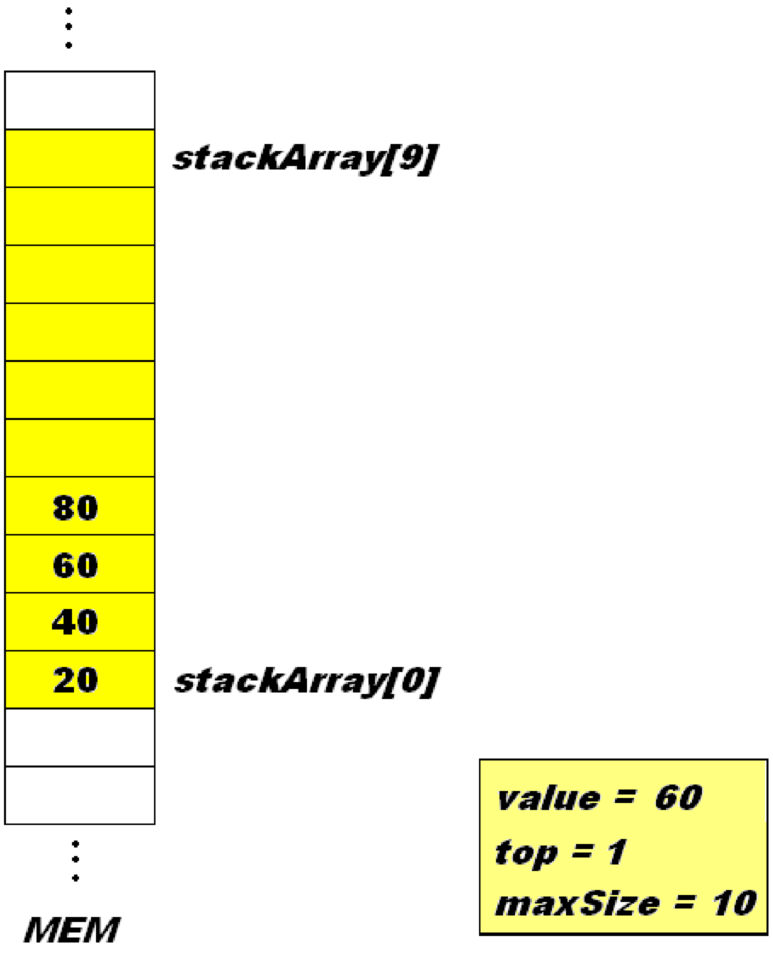
*top = 2*

*maxSize = 10*

# Pop

- while (!theStack.isEmpty())
  {
    long value = theStack.pop()
    …

- The element indexed by **top** is stored in **value**
- **top** is decremented by 1



# Print

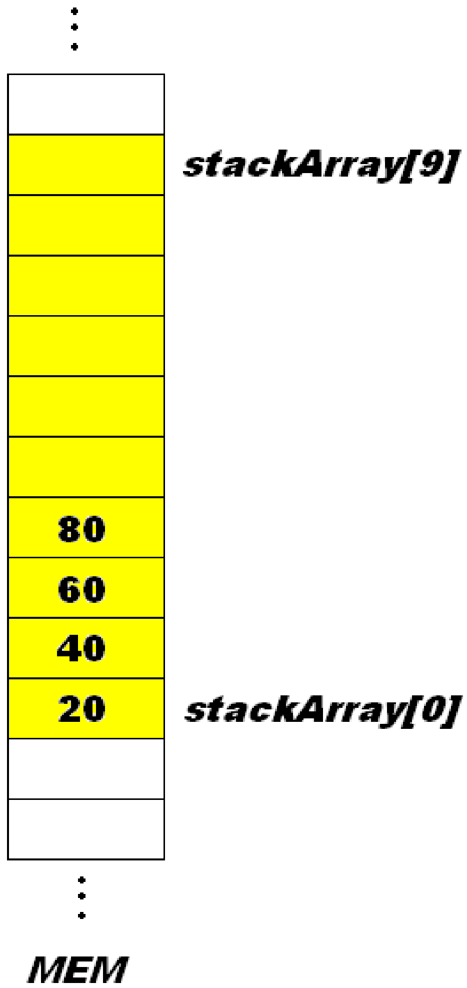
- while (!theStack.isEmpty())

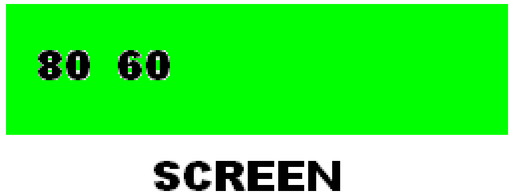
  {

  …

  System.out.print(value)

  System.out.print(“”)



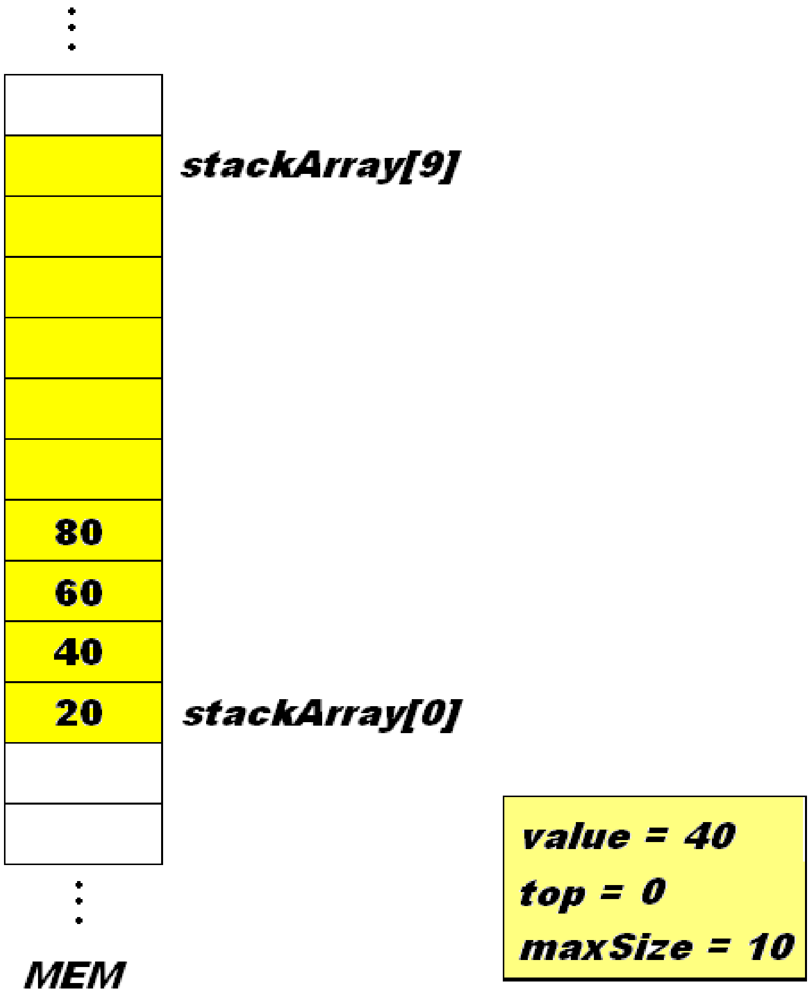




value = 60
top = 1
maxSize = 10

# Pop

- while (!theStack.isEmpty())
  {
    long value = theStack.pop()
    …

- The element indexed by **top** is stored in **value**
- **top** is decremented by 1



# Print

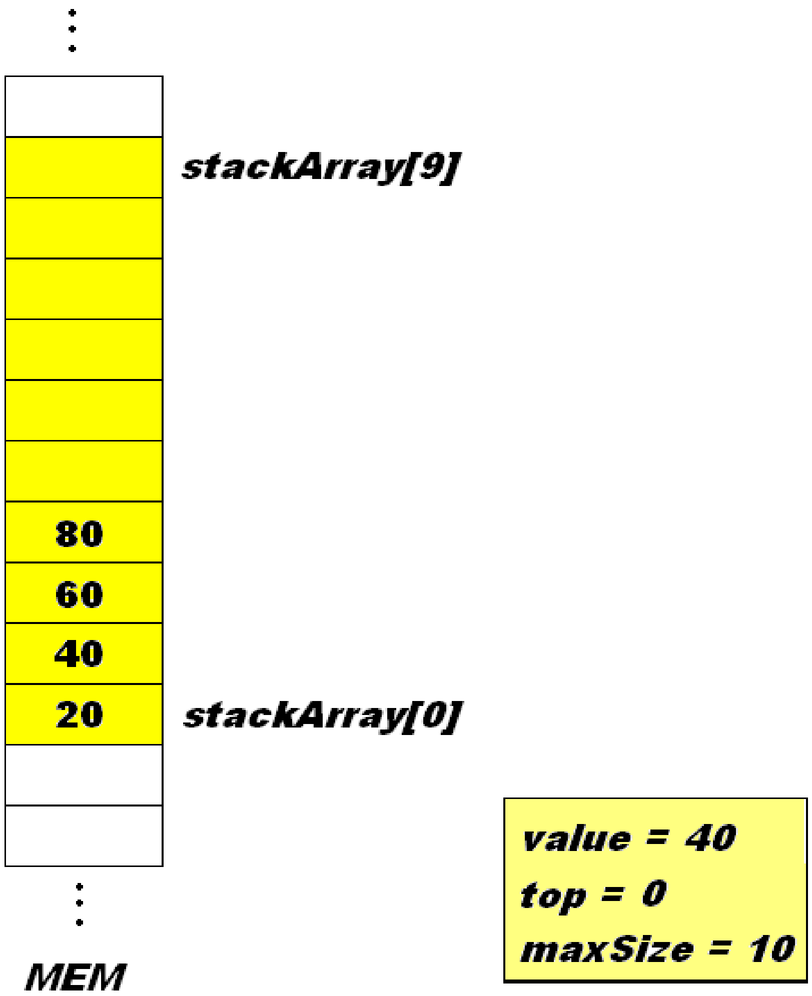
- while (!theStack.isEmpty())

  {

  …

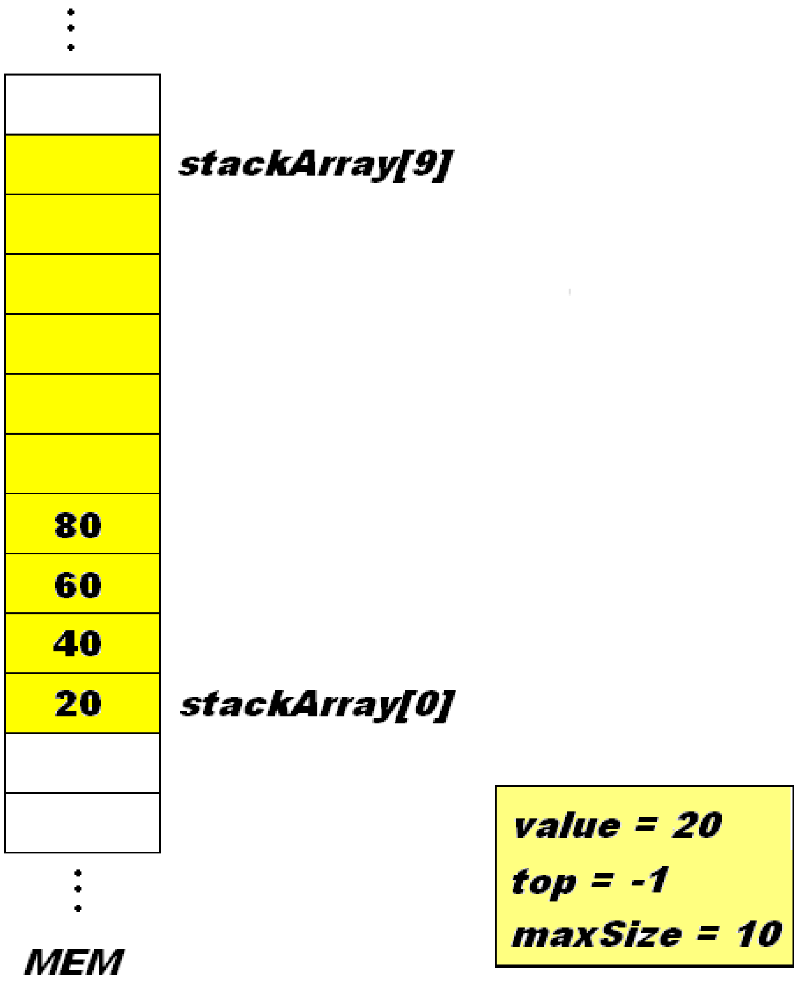
  System.out.print(value)

  System.out.print(“”)

# Pop

- while (!theStack.isEmpty())
  {
    long value = theStack.pop()
    …




- The element indexed by **top** is stored in **value**
- **top** is decremented by 1

# Print

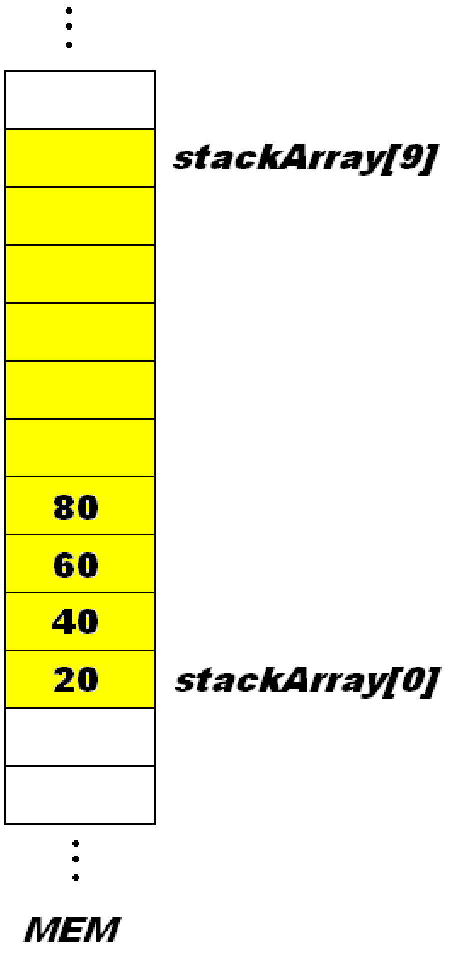
- while (!theStack.isEmpty())

  {

  …

  System.out.print(value)

  System.out.print(“”)


value = 20
top = -1
maxSize = 10

# Example Application: Word Reversal

- Let’s use a stack to take a string and reverse its characters
  - How could this work? Let’s look.
- Reminder of the available operations with Strings:
- If I have a string s
  - s.charAt(j) <- Return character with index j
  - s + “…” <- Append a string (or character to s)
  - What would we need to change about our existing stack class?
- Reverser, page 125

# Example Application: Delimiter Matching

- This is done in compilers!
- Parse text strings in a computer language
- Sample delimiters in Java:
  - {, }
  - [, ]
  - (, )

- All opening delimiters should be matched by closing ones
- Also, later opening delimiters should be closer before earlier ones
  - See how the stack can help us here?

# Example Strings

- c[d]
- a{b[c]d}e
- a{b(c]d}e
- a[b{c}d]e}
- a{b(c)

- Which of these are correct?
- Which of these are incorrect?

# Algorithm

- Read each character one at a time
- If an opening delimiter, place on the stack
- If a closing delimiter, pop the stack
  - If the stack is empty, error
  - Otherwise if the opening delimiter matches, continue
  - Otherwise, error
- If the stack is not empty at the end, error

# Example

- Let’s look at a stack for a{b(c[d]e)f}

| Character | Stack | Action |
|---|---|---|
| a | | x |
| { | { | push ‘{‘ |
| b | { | x |
| ( | {( | push ‘(‘ |
| c | {( | x |
| [ | {([ | push ‘[‘ |
| d | {([ | x |
| ] | {( | pop ‘[‘, match |
| e | {( | x |
| ) | { | pop ‘(‘, match |
| f | { | x |
| } | | pop ‘{‘, match |

# Example

- Let’s do one that errors: a[b{c}d]e}
- Together on the board

# Stacks: Evaluation

- For the tools we saw: reversing words and matching delimiters, what about stacks made things easier?
  - i.e. What would have been difficult with arrays?
  - Why does using a stack make your program easier to understand?

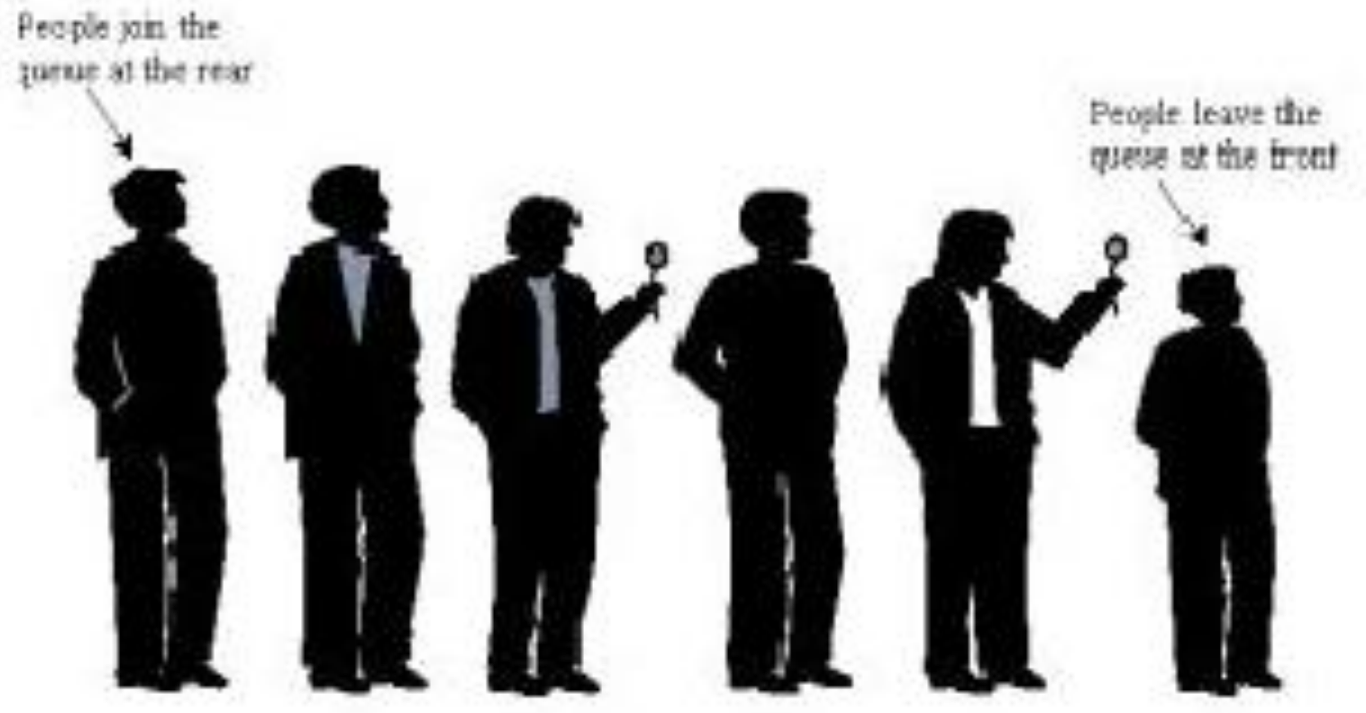
- Efficiency
  - Push -> O(1) (Insertion is fast, but only at the top)
  - Pop -> O(1) (Deletion is fast, but only at the top)
  - Peek -> O(1) (Access is fast, but only at the top)

# Queues

- British for “line”
- Somewhat like a stack
  - Except, first-in-first-out
  - Thus this is a FIFO structure.

# Analogy:

- Line at the movie theatre
- Last person to line up is the last person to buy

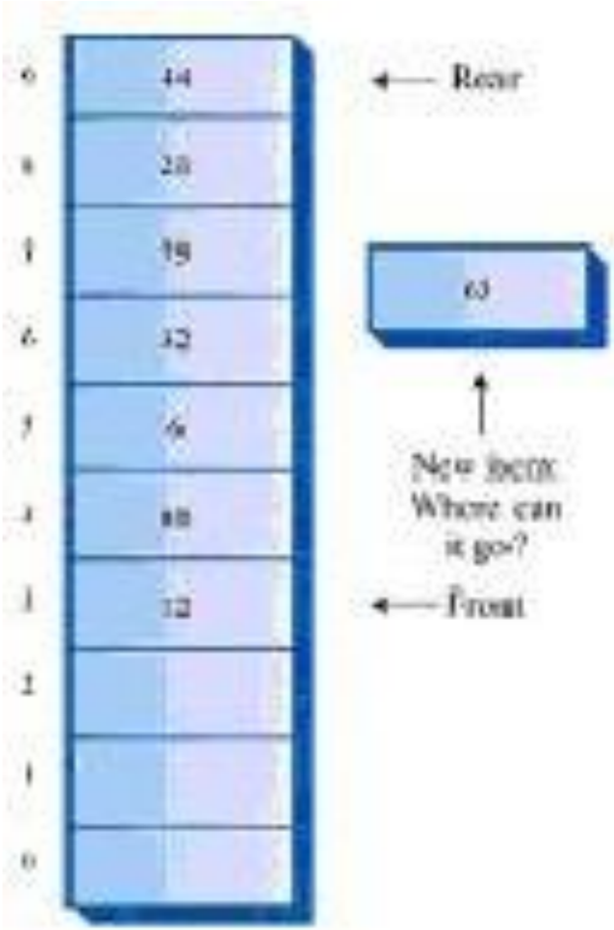


# Queue Operations

- insert()
  - Also referred to as put(), add(), or enque()
  - Inserts an element at the back of the queue
- remove()
  - Also referred to as get(), delete(), or deque()
  - Removes an element from the front of the queue
- peekRear()
  - Element at the back of the queue
- peekFront()
  - Element at the front of the queue

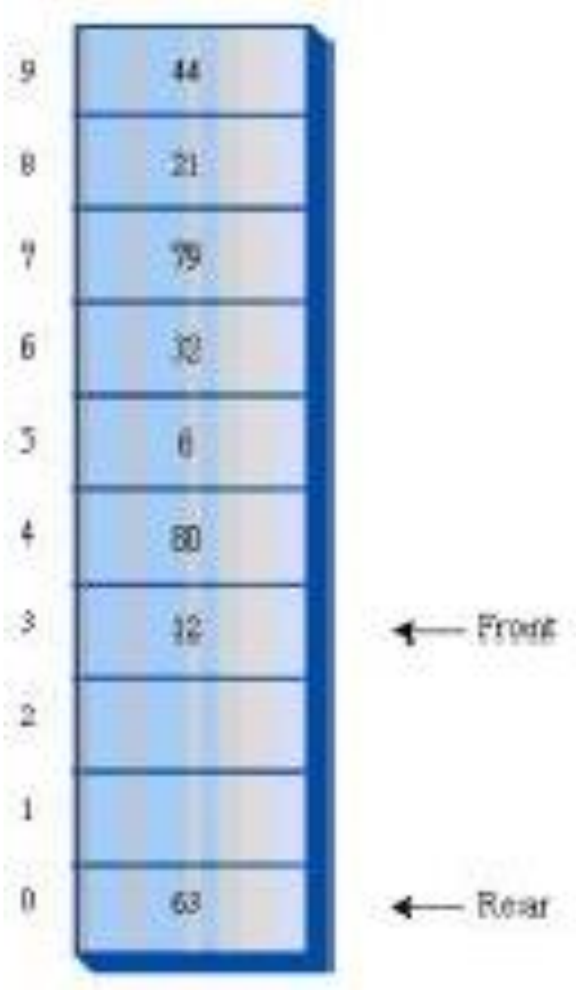
# Insert and remove occur at opposite ends!!!

- Whereas with a stack, they occurred at the same end
  - That means that if we remove an element we can reuse its slot
- With a queue, you cannot do that
- Unless….

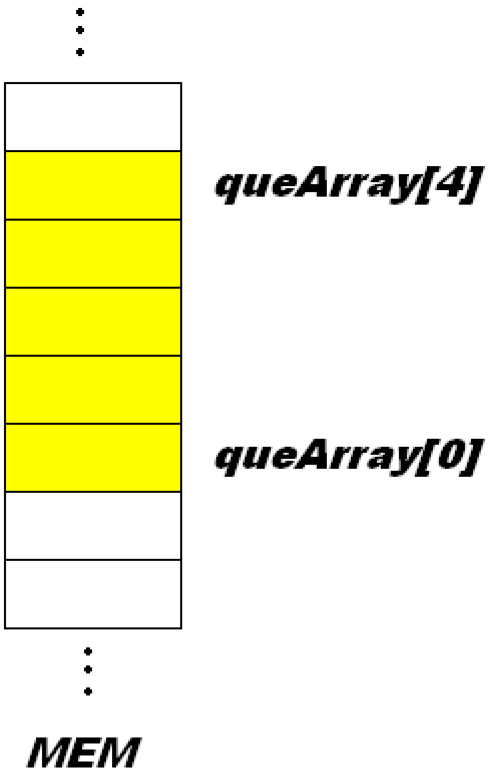


# Circular Queue

- Indices 'wraparound'

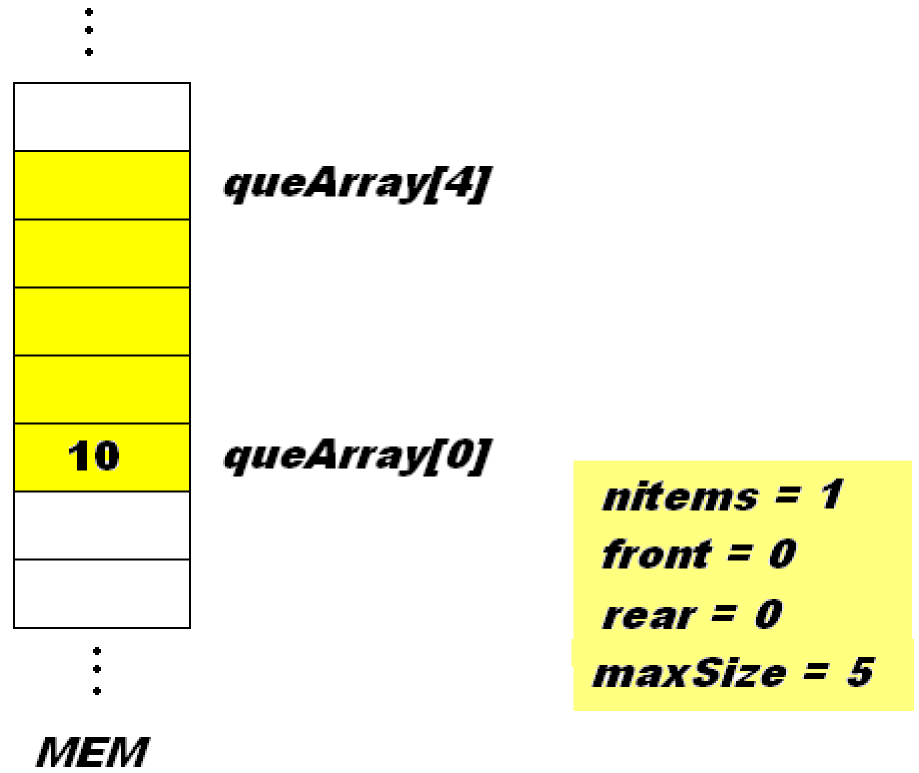


# Queue theQueue = new Queue(5);

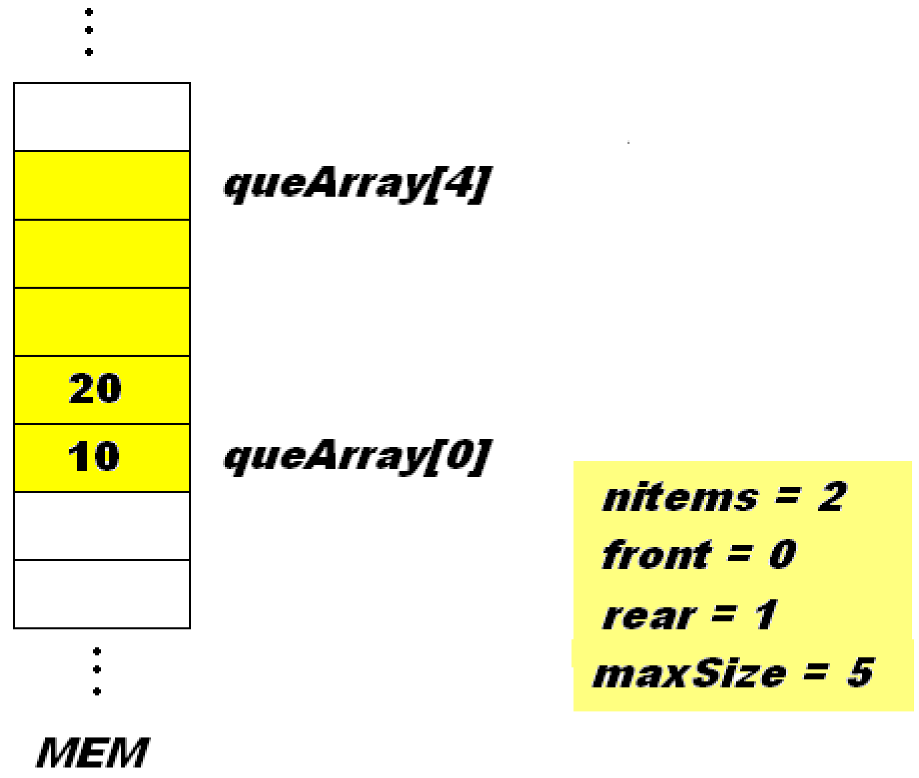



nitems = 0
front = 0
rear = -1
maxSize = 5

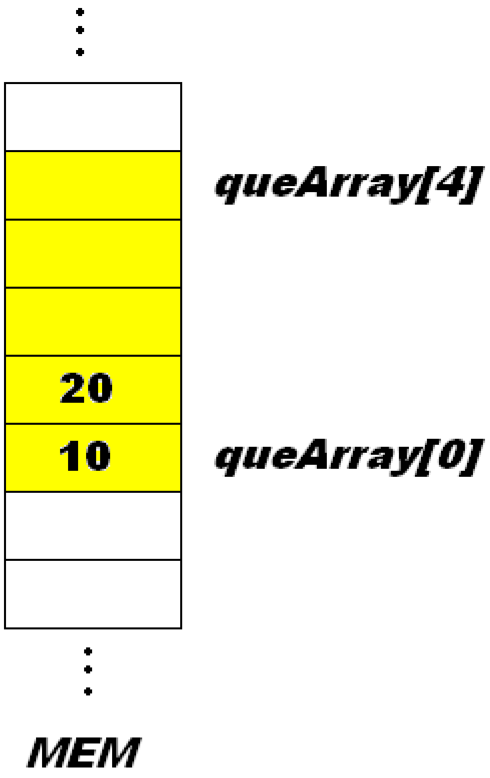
# theQueue.insert(10);



# theQueue.insert(20);

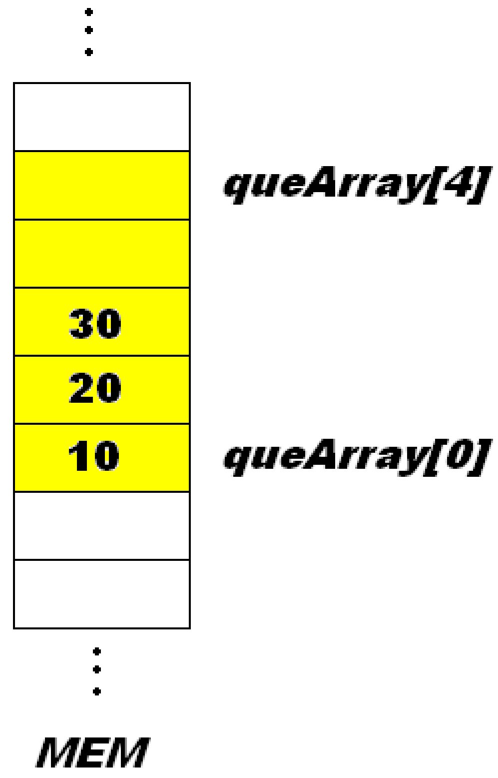


# theQueue.insert(30);



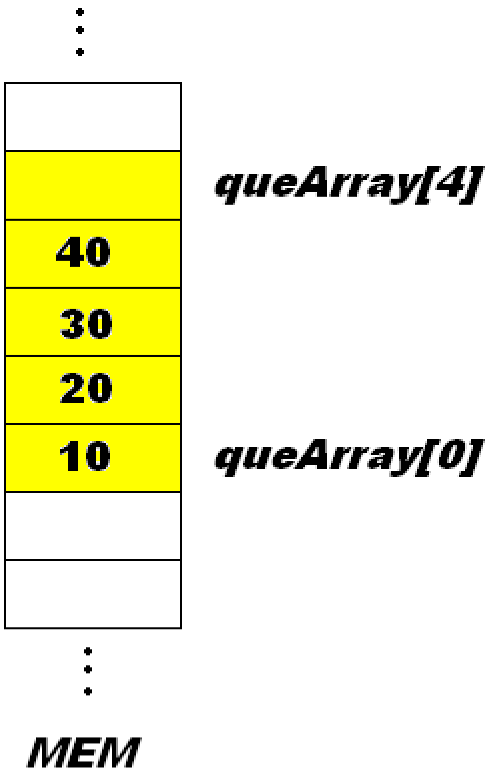

nitems = 2
front = 0
rear = 1
maxSize = 5

# theQueue.insert(30);



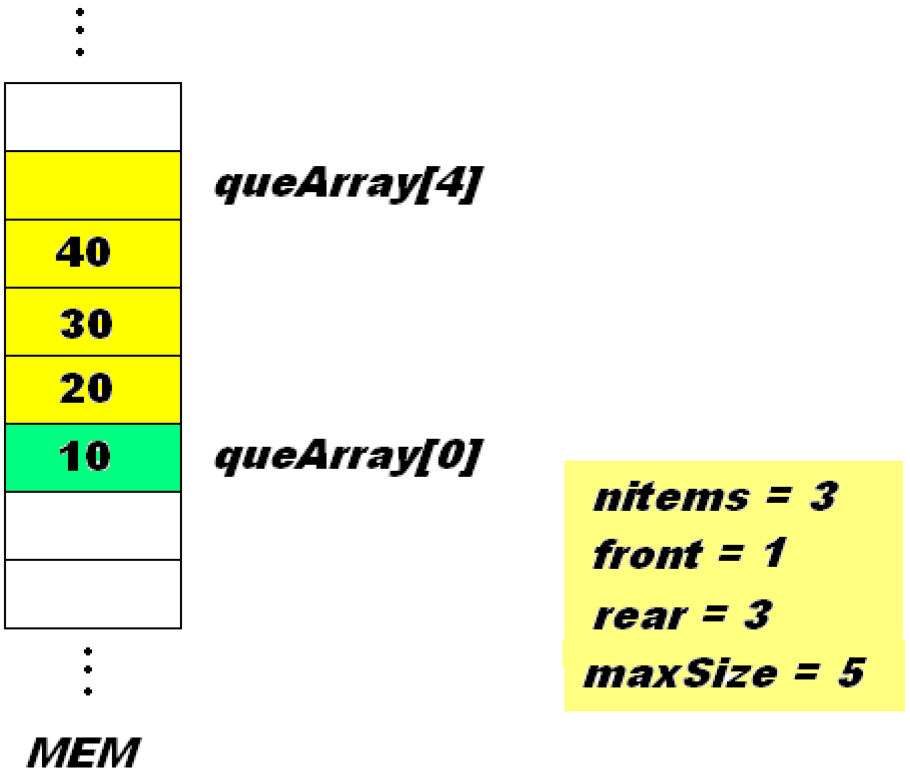

nItems = 3
front = 0
rear = 2
maxSize = 5

# theQueue.insert(40);

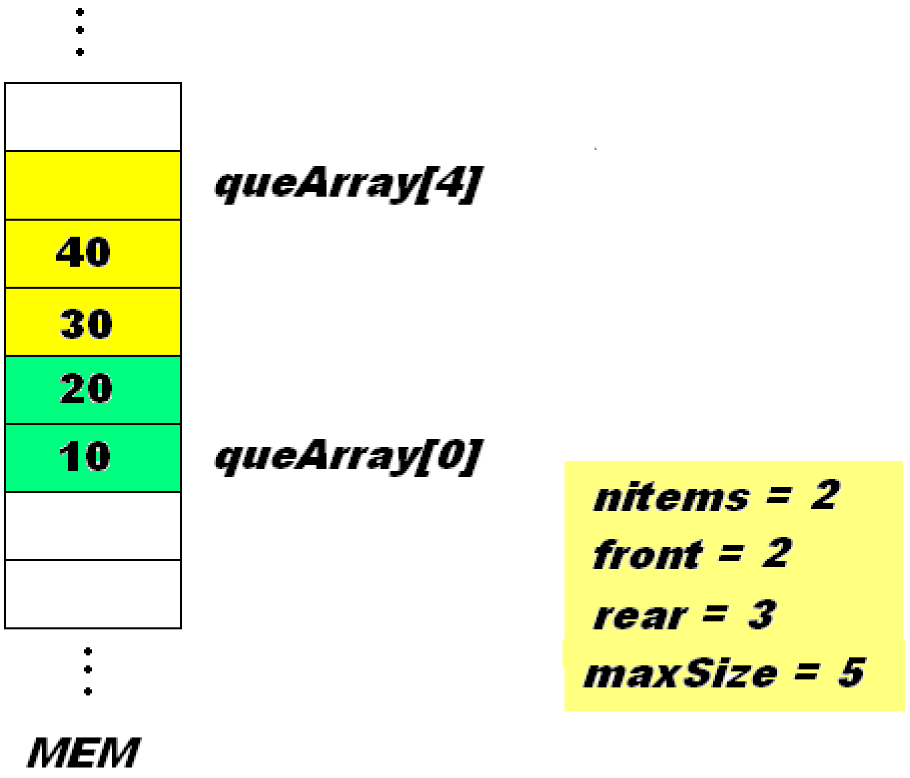



nitems = 4
front = 0
rear = 3
maxSize = 5

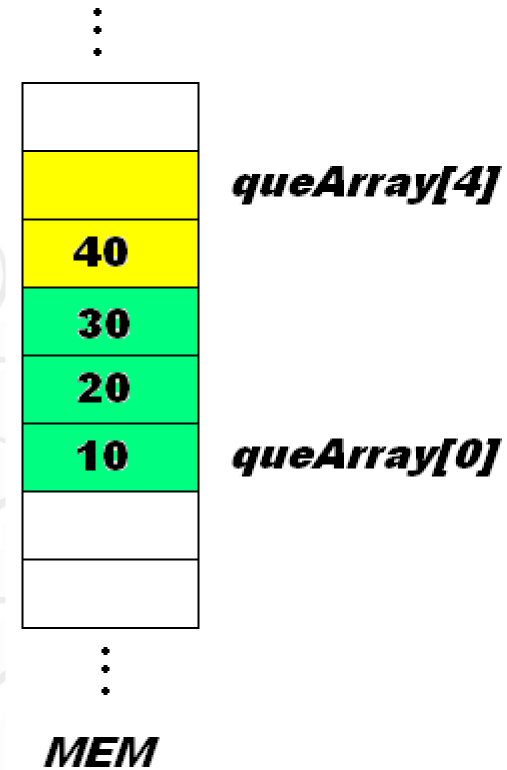
# theQueue.remove();



# theQueue.remove();

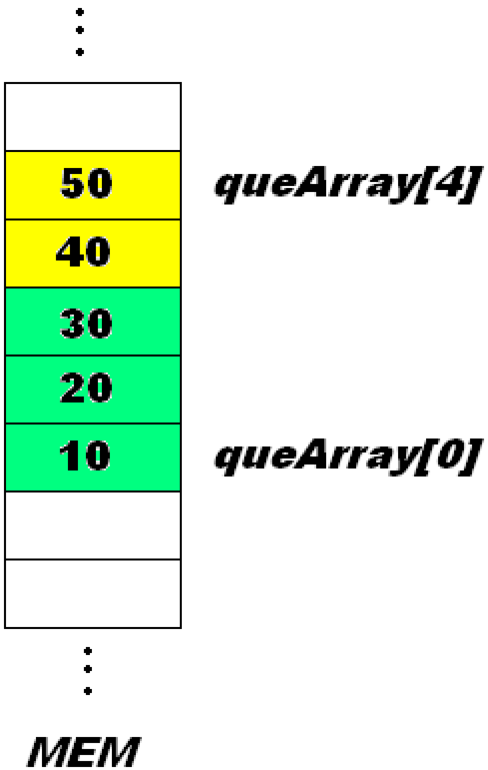


# theQueue.remove();



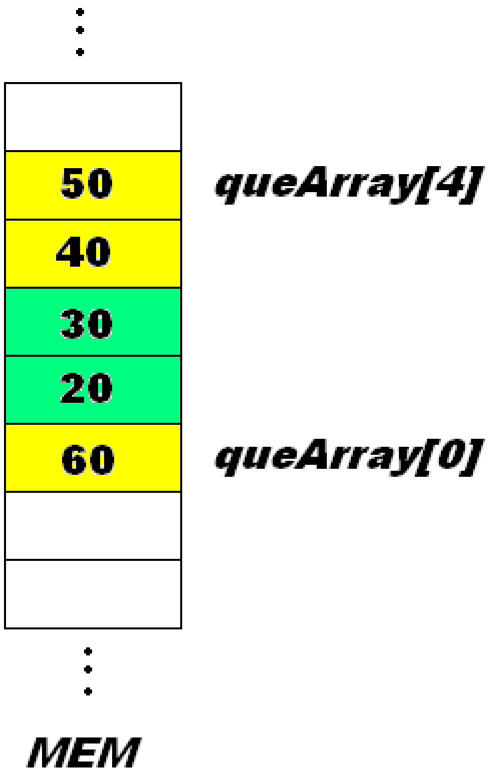

nitems = 1
front = 3
rear = 3
maxSize = 5

# theQueue.insert(50);



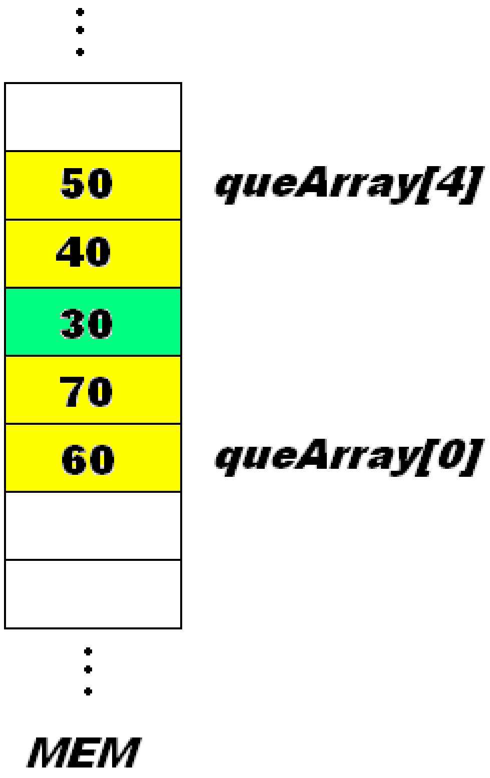

***nitems = 2***
***front = 3***
***rear = 4***
***maxSize = 5***

# theQueue.insert(60);



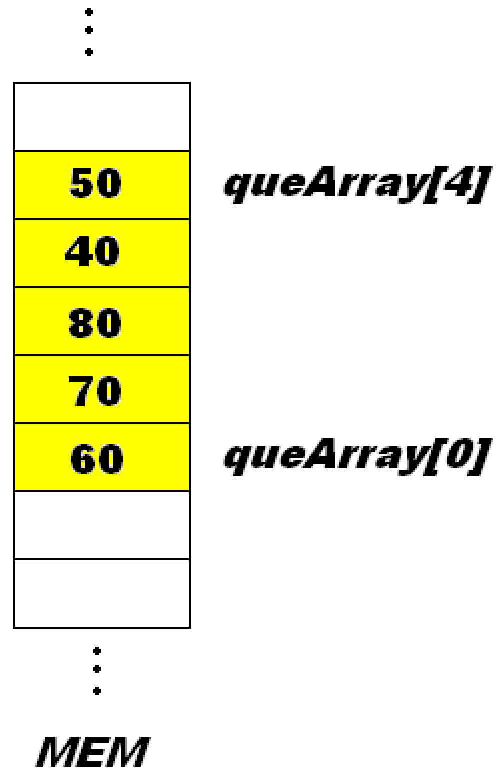

nitems = 3
front = 3
rear = 0
maxSize = 5

# theQueue.insert(70);




nItems = 4
front = 3
rear = 1
maxSize = 5

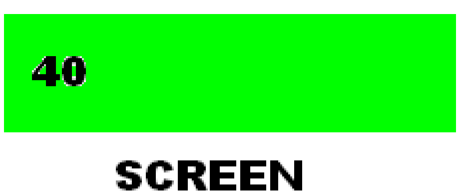
# theQueue.insert(80);

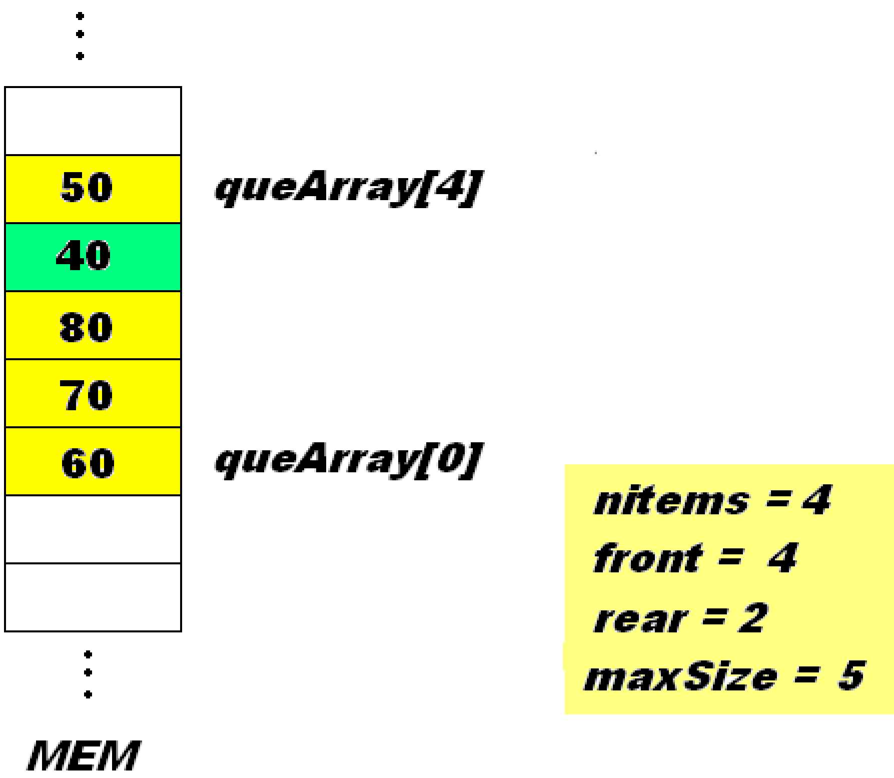



***nitems = 5***
***front = 3***
***rear = 2***
***maxSize = 5***

# Remove and print…

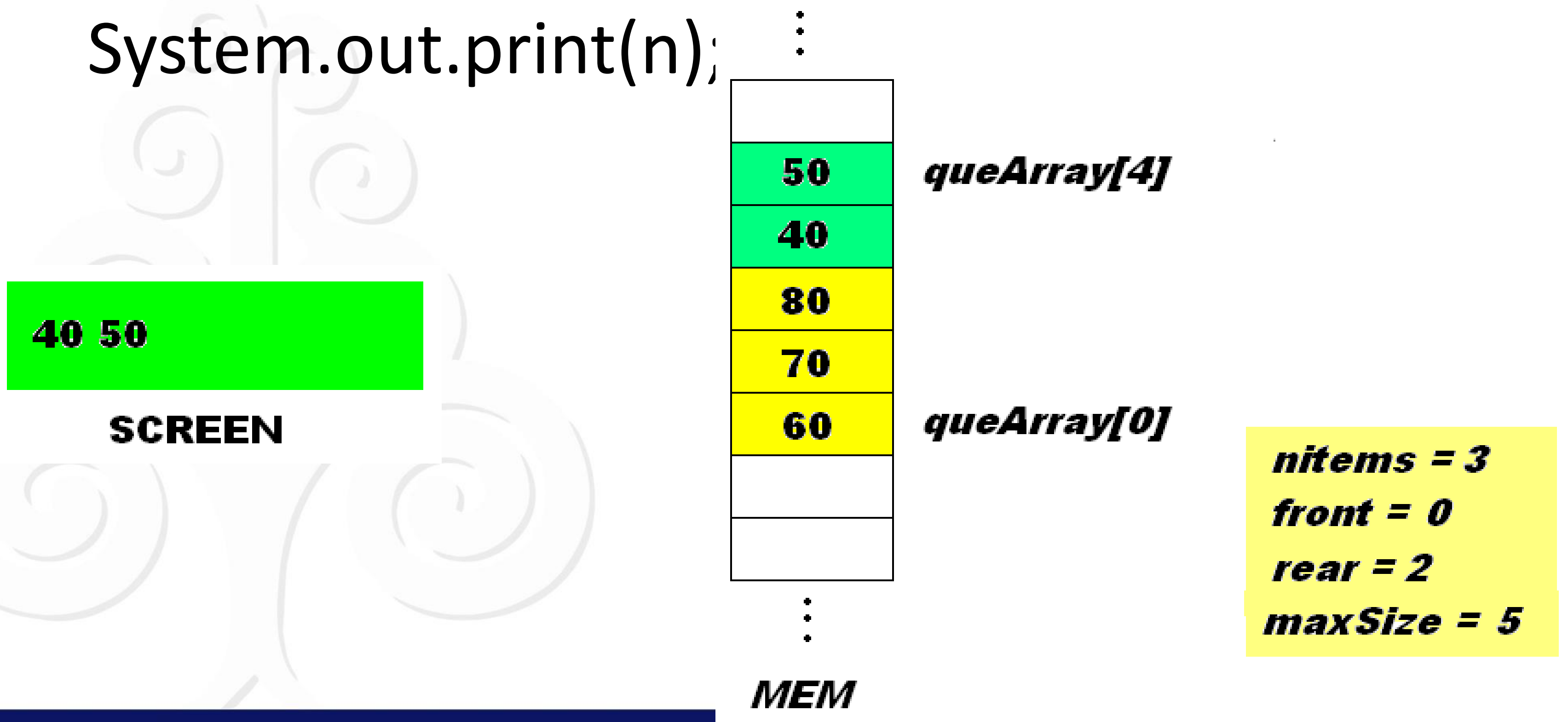
- while (!theQueue.isEmpty())
- long n = theQueue.remove();
- System.out.print(n);



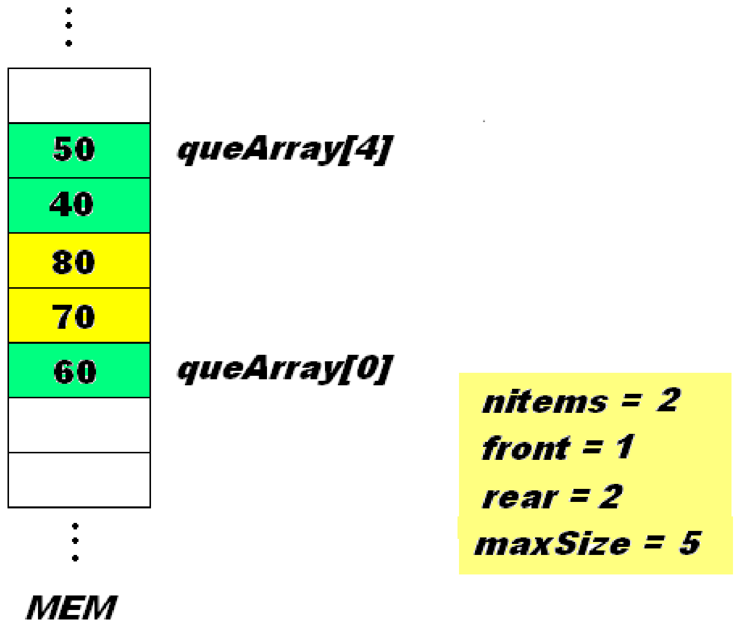
# Remove and print…

- while (!theQueue.isEmpty())
-     long n = theQueue.remove();
-     System.out.print(n);



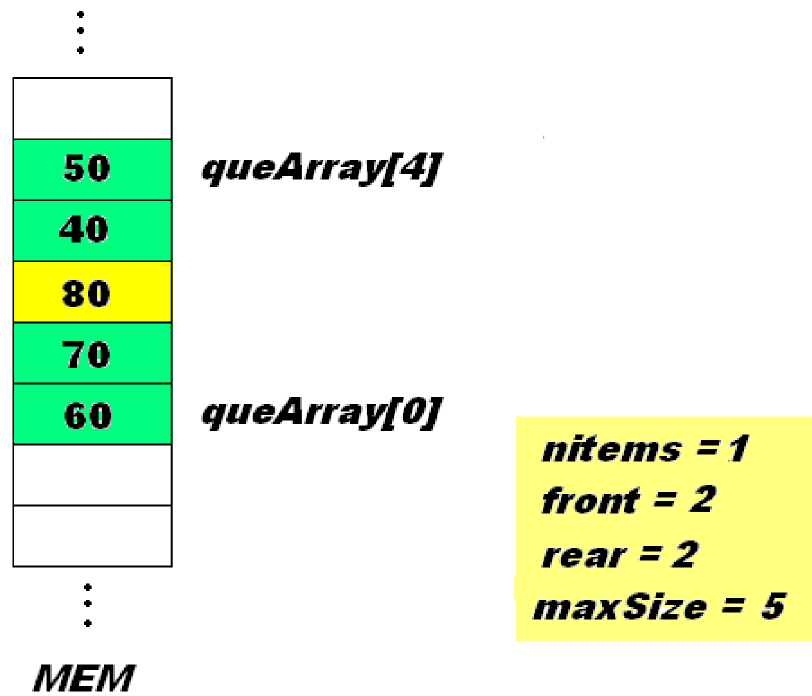
# Remove and print…

- while (!theQueue.isEmpty())
- long n = theQueue.remove();
- System.out.print(n);

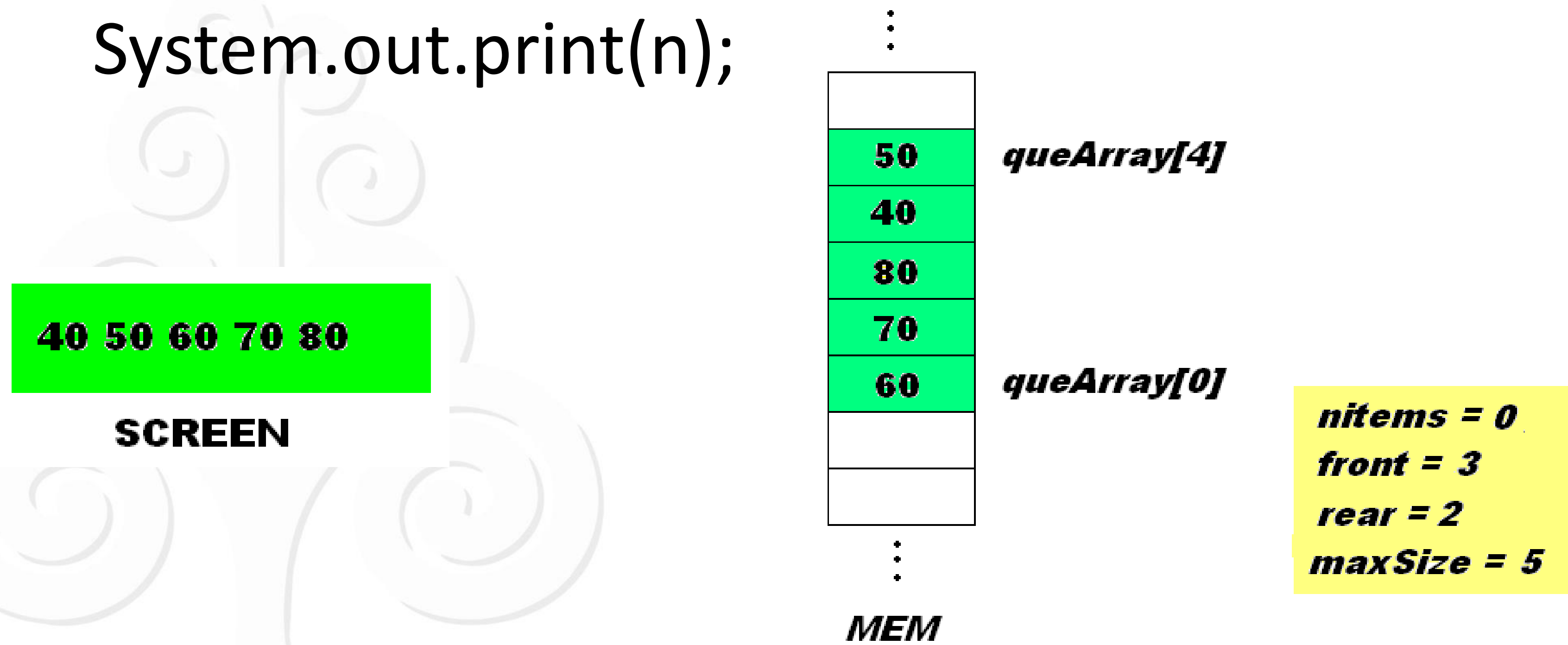
# Remove and print…

- while (!theQueue.isEmpty())
-     long n = theQueue.remove();
-     System.out.print(n);

# Remove and print…

- while (!theQueue.isEmpty())
-     long n = theQueue.remove();
-     System.out.print(n);



# Queues: Evaluation

- Some implementations remove nItems
  - Allow front and rear indices to determine if queue is full or empty, or size
    - Queue can appear to be full and empty (why?)
    - Additional overhead when determining size (why?)
    - Can remedy these by making array one size larger than the max number of items
- Efficiency
  - Same as stack:
    - Push: O(1) only at the back
    - Pop: O(1) only at the front
    - Access: O(1) only at the front

# Priority Queues

- Like a Queue
  - Has a front and a rear
  - Items are removed from the front
- Difference
  - No longer FIFO
  - Items are ordered
- We have seen ordered arrays. A priority queue is essentially an 'ordered queue'

- Mail analogy: you want to answer the most important first

# Priority Queue Implementation

- Almost NEVER use arrays. Why?
- Usually employs a heap (we'll learn these later)
- Application in Computing
  - Programs with higher priority, execute first
  - Print jobs can be ordered by priority
- Nice feature: The min (or max) item can be found in O(1) time

# Parsing Arithmetic Expressions

- A task that must be performed by devices such as computers and calculators
- Parsing is another word for *analyzing*, that is, piece by piece

- For example, given the expression 2*(3+4)
- We have to know to first evaluate 3+4
- Then multiple the result by 2

# How it's done…

- 1. Transform the arithmetic expression into postfix notation
  - Operators follow their two operands, i.e.
  - 3+4 = 34+ (in postfix)
  - 2*(3+4) = 234+* (in postfix)
  - May seem silly, but it makes the expression easier to evaluate with a stack

- 2. Use a stack and evaluate

# Some practice

- Convert the following to postfix:

- 3*5
- 3+8*4  (remember the rules of precedence!)
- (3+4)*(4+6)

# Translating infix to postfix

- Think conceptually first. How do we evaluate something like: 2*(3+4) to get 14?
  - Read left to right
  - When we've read far enough to evaluate two operands and an operator - in the above case, 3+4
    - Evaluate them: 3+4=7
    - Substitute the result: 2*7 = 14
  - Repeat as necessary

# Parsing in our Heads

| | |
|---|---|
| • 2*(3+4) | |
| • We have to evaluate anything in parentheses before using it | |
| • Read | Parsed |
| • 2 | 2 |
| • 2* | 2* |
| • 2*( | 2*( |
| • 2*(3 | 2*(3 |
| • 2*(3+ | 2*(3+ |
| • 2*(3+4) | 2*(3+4) |
| • | 2*7<br>14 |

# Precedence

- 3+4*5
- Note here we don’t evaluate the ‘+’ until we know what follows the 4 (a ‘*’)
- So the ‘parsing’ proceeds like this:

| Read | Parsed |
| --- | --- |
| 3 | 3 |
| + | 3+ |
| 4 | 3+4 |
| * | 3+4* |
| 5 | 3+4*5 |
| | 3+20 |
| | 23 |

# Infix to Postfix: Algorithm

- Start with your infix expression, and an empty postfix string
  - Infix: 2*(3+4)  Postfix:
- Go through the infix expression character-by-character
- For each operand:
  - Copy it to the postfix string
- For each operator:
  - Copy it at the 'right time'
  - When is this? We'll see

# Example: 2*(3+4)

| *Read* | *Postfix* | *Comment* |
|---|---|---|
| 2 | 2 | Operand |
| * | 2 | Operator |
| ( | 2 | Operator |
| 3 | 23 | Operand |
| + | | Operator |
| 4 | 234 | Operand |
| ) | 234+ | Saw ), copy + |
| | 234+* | Copy remaining ops |

# Example: 3+4*5

| *Read* | *Postfix* | *Comment* |
|---|---|---|
| 3 | 3 | Operand |
| + | 3 | Operator |
| 4 | 34 | Operand |
| * | 34 | Operator |
| 5 | 345 | Operand |
| | 345* | Saw 5, copy * |
| | 345*+ | Copy remaining ops |

# Rules on copying operators

- You cannot copy an operator to the postfix string if:
  - It is followed by a left parenthesis '(‘
  - It is followed by an operator with higher precedence (i.e., a '+' followed by a '*')
- If neither of these are true, you can copy an operator once you have copied both its operands

- We can use a stack to hold the operators before they are copied. Here's how:

# How can we use a stack?

- Suppose we have our infix expression, empty postfix string and empty stack S. We can have the following rules:
  - If we get an operand, copy it to the postfix string
  - If we get a ‘(‘, push it onto S
  - If we get a ‘)’:
    - Keep popping S and copying operators to the postfix string until either S is empty or the item popped is a ‘(‘
  - Any other operator:
    - If S is empty, push it onto S
    - Otherwise, while S is not empty and the top of S is not a ‘(‘ or an operator of lower precedence, pop S and copy to the postfix string
    - Push operator onto S

  - To convince ourselves, let’s try some of the expressions

# Example: 3+4*5

- ***<u>Read Postfix Stack</u>***

# Evaluating postfix expressions

- If we go through the trouble of converting to postfix, there's got to be a reason, right?
- Well, there is! The resulting expression is much easier to evaluate, once again using a stack

- Take one example: 345*+
  - For every operand push it onto a stack
  - Everytime we encounter an operator, apply it to the top two items and pop them, then push the result on the stack
  - We're done when we have a result and the stack is empty
  - Let's do some examples!

# Example: 234*+

| *Read* | *Stack* | *Comment* |
|---|---|---|
| 2 | 2 | Operand |
| 3 | 2 3 | Operand |
| 4 | 2 3 4 | Operand |
| * | 2 12 | Apply * to 3 and 4 push result |
| + | 24 | Apply * to 2 and 12 push result |

# THANK YOU

Mengakar Kuat
Menjulang
Tinggi