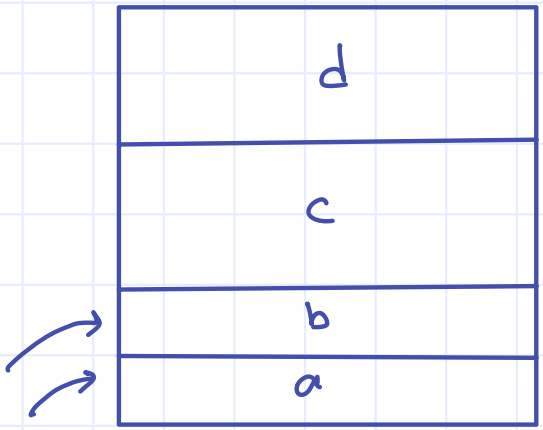
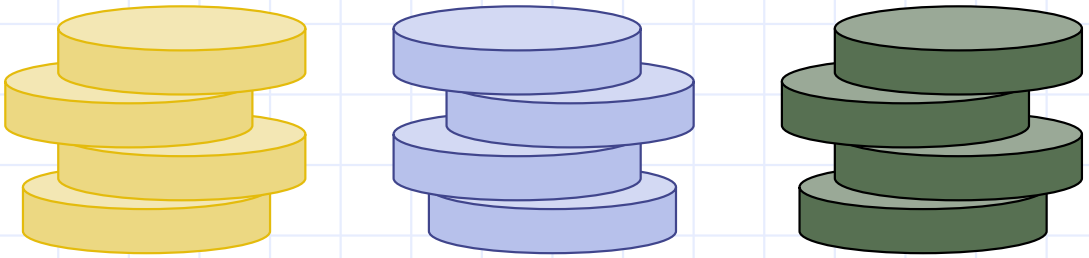
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Stacks

ماخيها ترتيب
ونظرتي للخوق دائم .

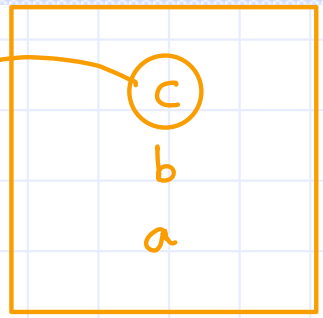
(عكس) + reverse و بيتشتغل



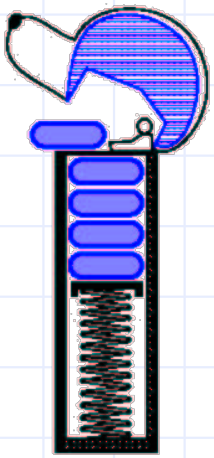


last in first out → اخر واحد دخل اول واحد بيطلع

LIFO



The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element → update size (إضافة)
  - object pop(): removes and returns the last inserted element (حذف)

remove top index

decrement size--

- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it (بدون ما تشيلة) (بترجع الداتا اللي موجودة بالـ last element) copy from element. not do thing in size
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

ADTs :

١- إضافة.

٢- حذف.

٣- استرجاع الداتا.

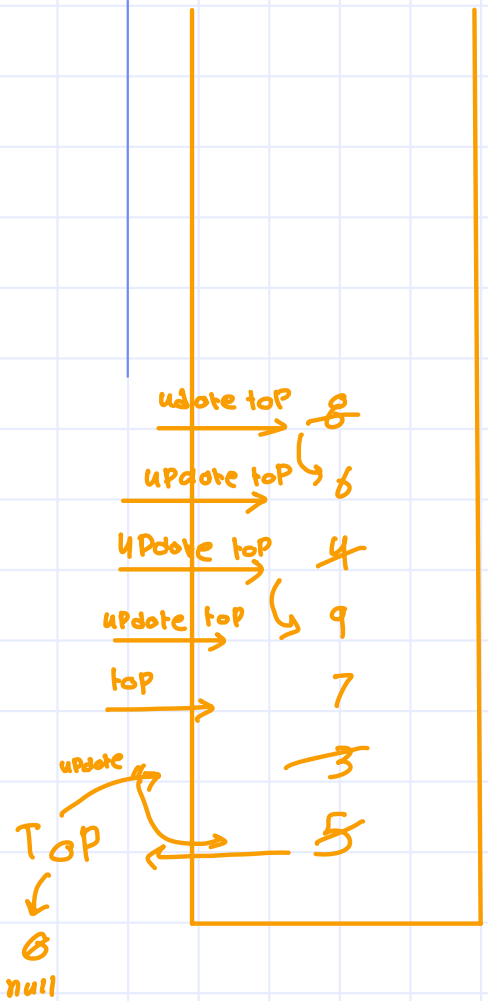
٤- traverse (بمر على كل الداتا)

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

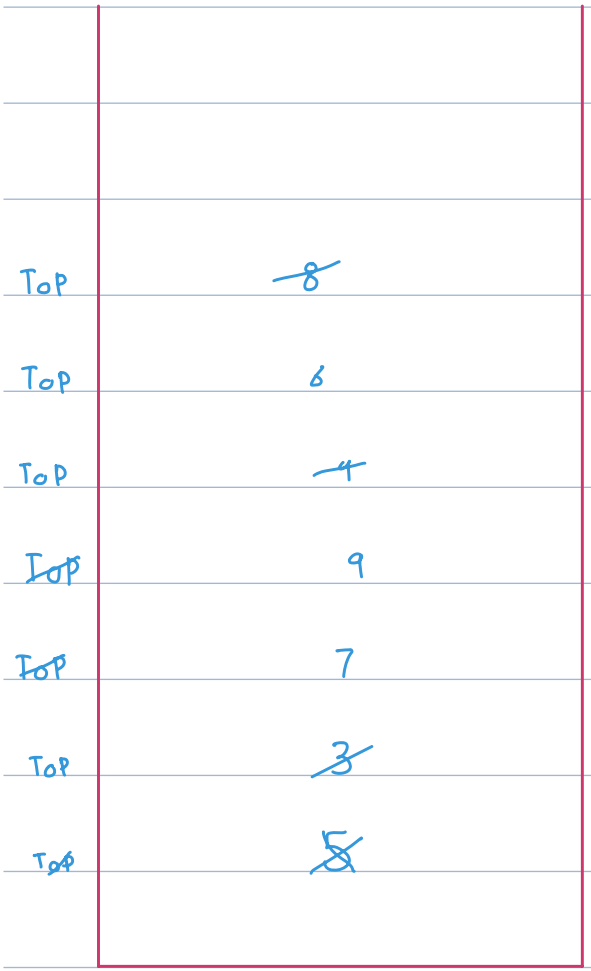
```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E top();
  void push(E element);
  E pop();
}
```

Example



Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Size	Top
1	5
2	3
2	3
1	5
1	5
0	Ø
0	Ø
0	Ø
1	7
2	9
2	9
3	4
3	4
2	9
3	6

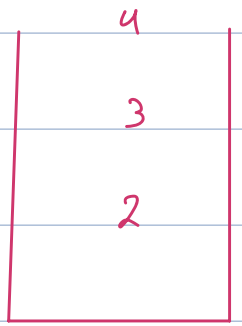


الباقي هي

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	– no return value	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Recursion stack and Queue:

```
f(s)

if (stack Empty()) return

else {

Te = S. Pop

f(s)

S. Push(e)
}
}
e = 4
```



Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        return null
    else
        t ← t – 1
        return S[t + 1]
```

Array-based Stack (cont.)

how much to call size method if we use Array : Constant to PoP element : o (constant) to Push: n

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(o)
  if $t = S.length - 1$ **then**
    ↳ Top
    throw *IllegalStateException*
  else
    $t \leftarrow t + 1$ → t + 1
    $S[t] \leftarrow o$

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$ → اذا ماخيها loop
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Array-based Stack in Java

```
public class ArrayStack<E>
    implements Stack<E> {

  // holds the stack elements
  private E[ ] S;

  // index to top element
  private int top = -1;

  // constructor
  public ArrayStack(int capacity) {
     S = (E[ ]) new Object[capacity];
  }
```

```
  public E pop() {
   if isEmpty()
       return null;
    E temp = S[top];
    // facilitate garbage collection:
    S[top] = null;
    top = top – 1;
    return temp;
  }

… (other methods of Stack interface)
```

Example Use in Java

```
public class Tester {
  // … other methods
  public intReverse(Integer a[]) {
      Stack<Integer> s;
      s = new ArrayStack<Integer>();
      … (code to reverse array a) …
  }
```

```
  public floatReverse(Float f[]) {
      Stack<Float> s;
      s = new ArrayStack<Float>();
      … (code to reverse array f) …
  }
```

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

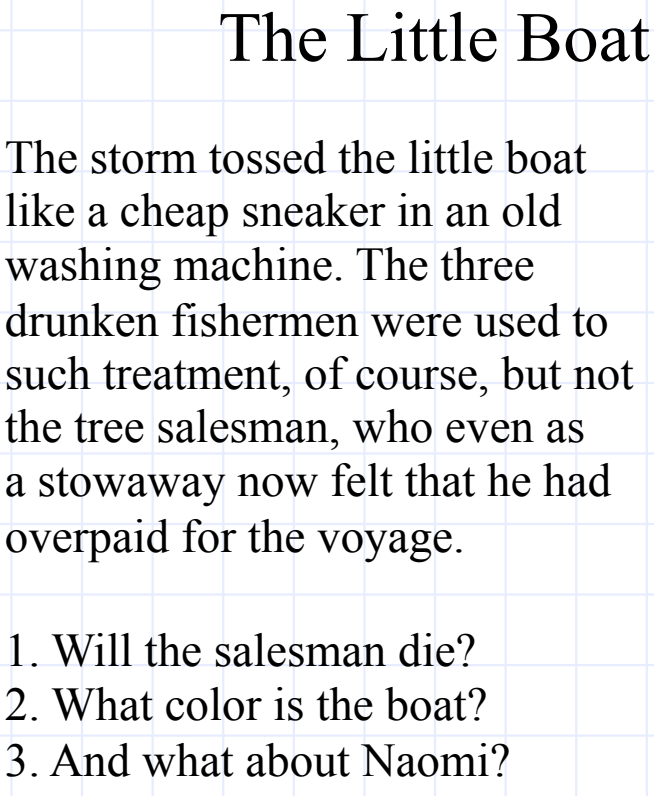
Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) { // this is a right delimiter
      if (buffer.isEmpty( )) // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false; // mismatched delimiter
      }
    }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k= html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next '<' character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

opening tag
< Center >

if it # it is closing tag

↳ closing tag

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

left to right Stovt

$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

1- first operand 3- another operand
2- operation

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
    prec(opStk.top()))
    doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Array Stack Big-oh

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Table 6.2: Performance of a stack realized by an array. The space usage is $O(N)$

Linked List Stack Big-oh

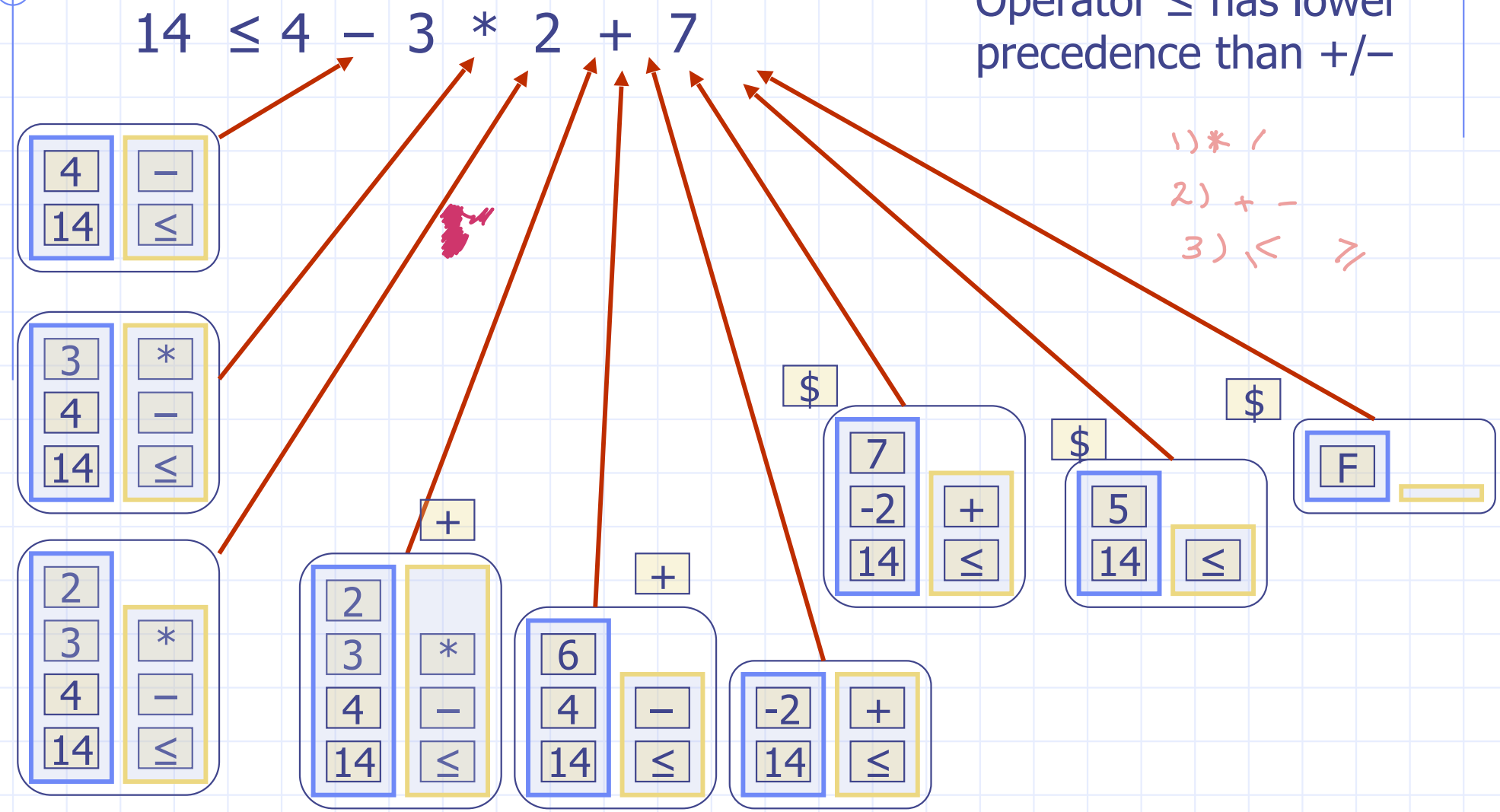
Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

O(1)
O(1)
O(1)
O(1)
O(1)

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/–



Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

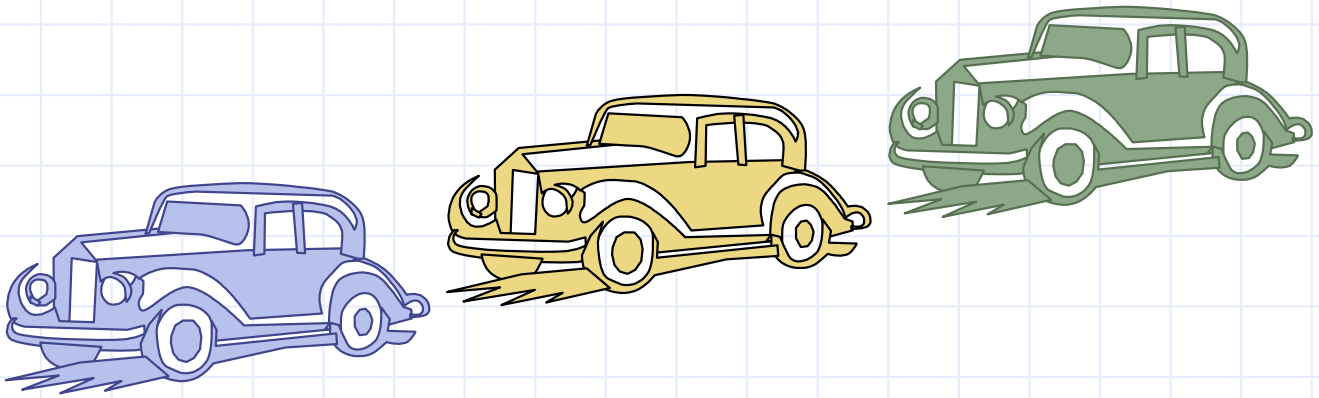
Queues

first n

F I F O → first in first out. first Process

first in , first out (اول واحد بيدخل ، هو اخ يطلع)

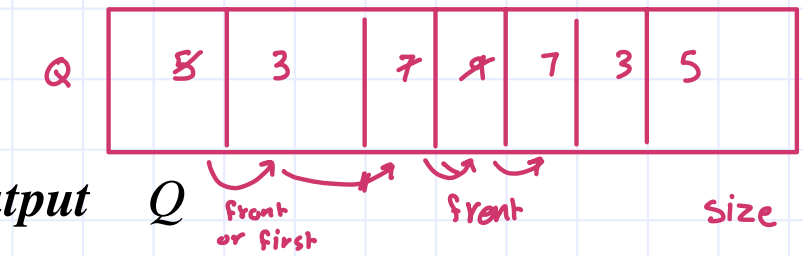
waiting



The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

Example



Operation	return	*Output*	*Q*	size
enqueue(5)	–	(5)	5	1
enqueue(3)	–	(5, 3)	5	2
dequeue()	*5*	(3)	3	1
enqueue(7)	–	(3, 7)	3	2
dequeue()	*3*	(7)	7	1
first()	*7*	(7)	7	1
dequeue()	*7*	()	null	0
dequeue()	*null*	()	null	0
isEmpty()	*true*	()	9	1
enqueue(9)	–	(9)	9	2
enqueue(7)	–	(9, 7)	9	3
size()	*2*	(9, 7)	9	5
enqueue(3)	–	(9, 7, 3)	7	3
enqueue(5)	–	(9, 7, 3, 5)		
dequeue()	*9*	(7, 3, 5)		

return

Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first() or front	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	*9*	(7, 3, 5)	

Queues

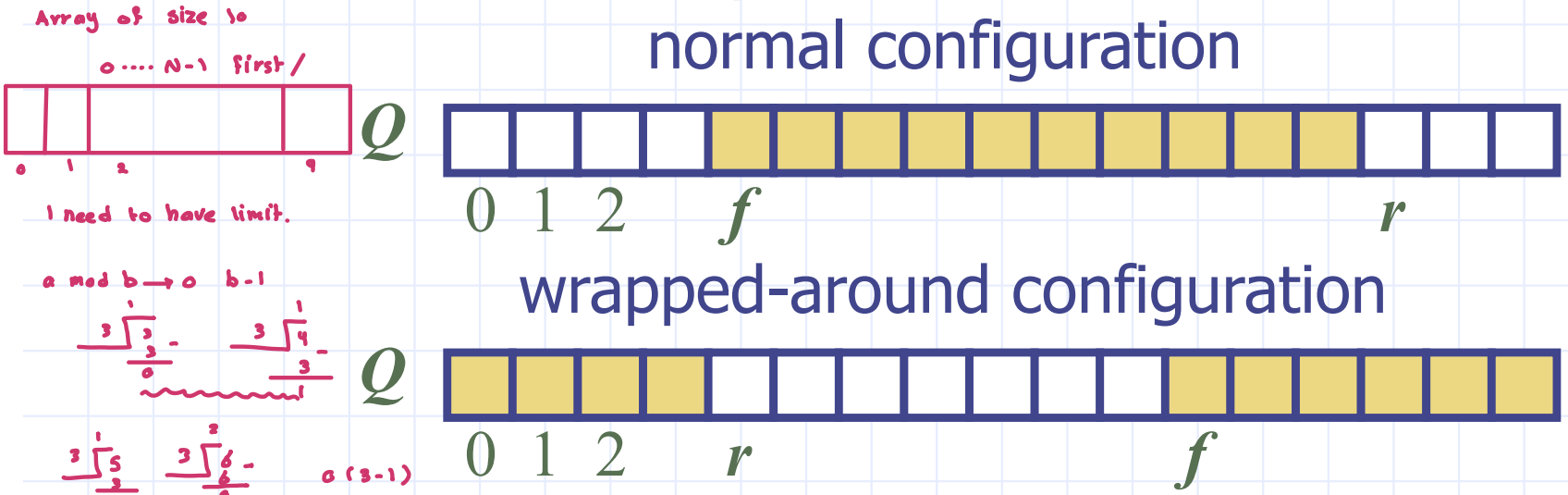
Applications of Queues

first in
first out.

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

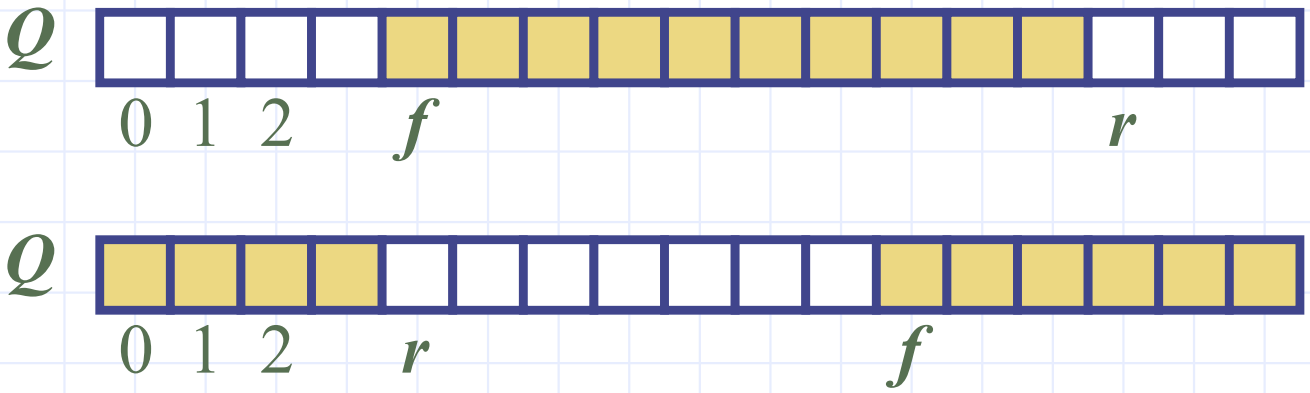
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



Queue Operations

- We use the modulo operator (remainder of division)

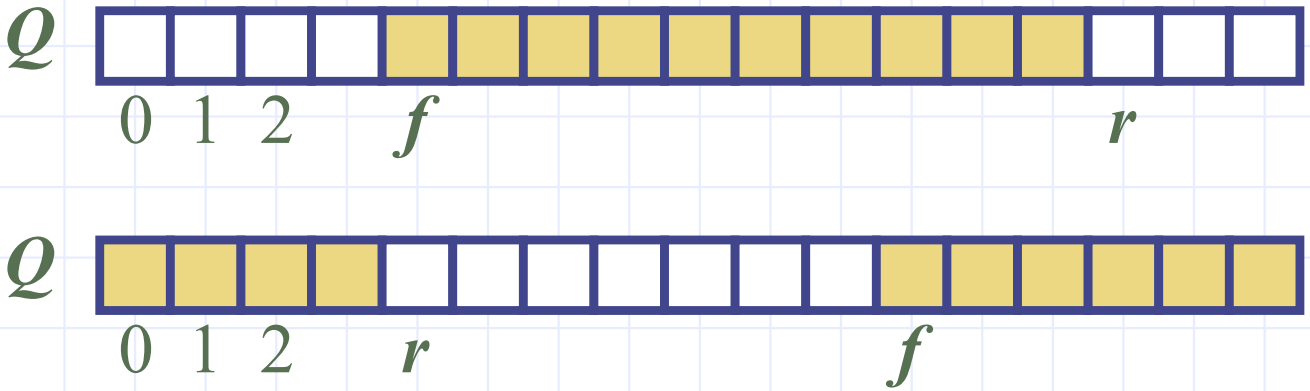
```
Algorithm size()
  return sz

Algorithm isEmpty()
  return (sz == 0)
```



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

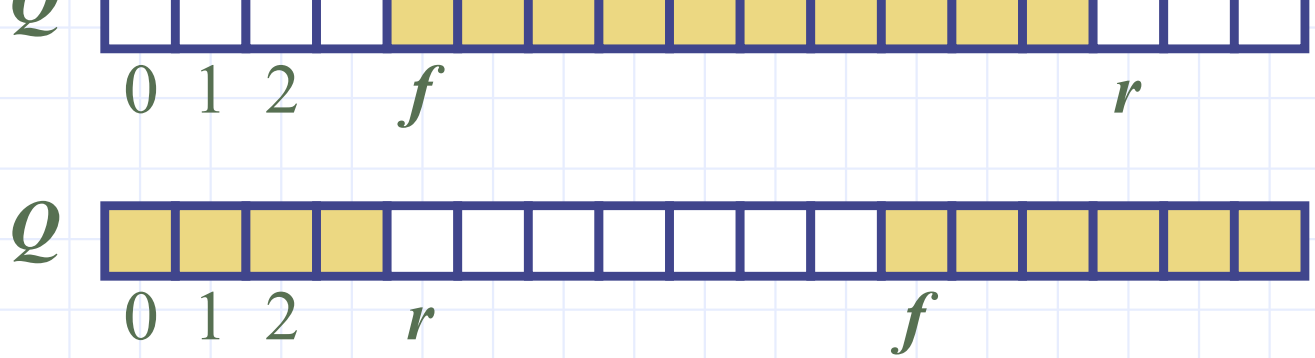
Algorithm *enqueue*(*o*)
  if *size*() = $N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$



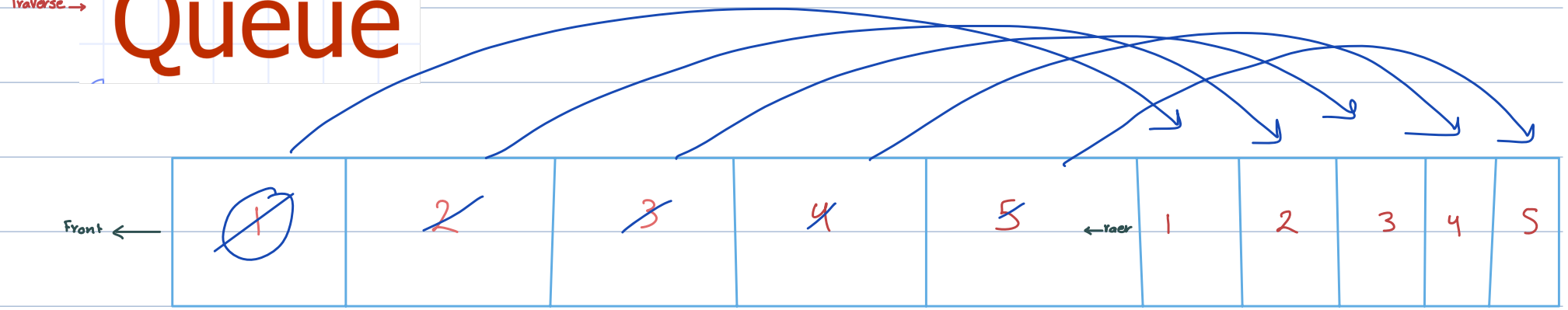
Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty

Algorithm *dequeue*()
  if *isEmpty*() **then**
    return *null*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow (sz - 1)$
    return o



Queue



باخذ معلوماتهم وارجعهم وراء وبوقف على عددهم .

```
Count = q.size ( ) ;          5  4  3

While ( (count > 0 ) {

  e = q.dequeue          +  2

[ Process : found min or max or Print ] ⟶ if ( e != Sara )

  q. enqueue (e);   ← مرتبطة بشرط معين

  count--;
```

3

Functio

enqueue

dequeue

front

size

is empt

Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

return copy

return copy

of first class

Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                    // generic array used for storage
5    private int f = 0;  -1 no element yet. // index of the front element
6    private int sz = 0;                   // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {      // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];  // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

Array-based Implementation (2)

```
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

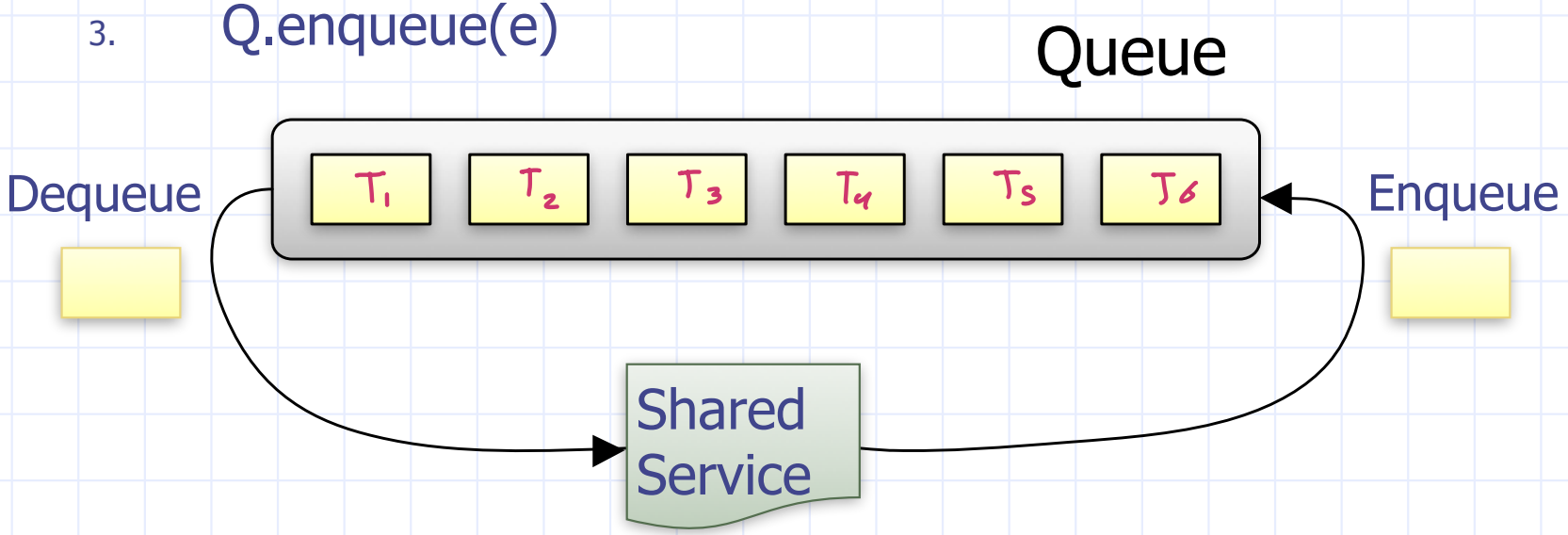
Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Array Queue Big-oh

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

f a queue realized by an array. The space usage is $O(N)$,

DEQueue Big-oh

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Performance of a deque realized by either a circular array or a doubly
The space usage for the array-based implementation is $O(N)$, where N

LinkedList Queue Big-oh

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Linked list