POLYTECH©
NICE-SOPHIA

Algorithms & Data Structures I

Lesson 4: ADT, stacks, queues

Marc Gaetano

Edition 2015-2016

Data structures

A data structure is a (often *non-obvious*) way to organize information to enable *efficient* computation over that information

A data structure supports certain *operations*, each with a:

- Meaning: what does the operation do/return
- Performance: how efficient is the operation

Examples:

- ***List*** with operations **`insert`** and **`delete`**
- ***Stack*** with operations **`push`** and **`pop`**

Trade-offs

A data structure strives to provide many useful, efficient operations

But there are unavoidable trade-offs:

- Time vs. space
- One operation more efficient if another less efficient
- Generality vs. simplicity vs. performance

We ask ourselves questions like:

- Does this support the operations I need efficiently?
- Will it be easy to use (and reuse), implement, and debug?
- What assumptions am I making about how my software will be used? (E.g., more lookups or more inserts?)

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations
  - Not concerned with implementation details

- Algorithm
  - A high level, language-independent description of a step-by-step process

- Data structure
  - A specific organization of data and family of algorithms for implementing an ADT

- Implementation of a data structure
  - A specific implementation in a specific language

Example: Stacks

- The ***Stack*** ADT supports operations:
  - **`isEmpty`**: have there been same number of pops as pushes
  - **`push`**: adds an item to the top of the stack
  - **`pop`**: raises an error if empty, else removes and returns most-recently pushed item not yet returned by a pop
  - **`peek`**: the same as **`pop`** but without removing the item
  - What else?

- A Stack data structure could use a linked-list or an array and associated algorithms for the operations

- One implementation is in the library **`java.util.Stack`**

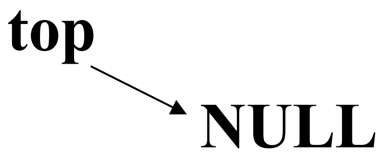
Why useful

The Stack ADT is a useful abstraction because:

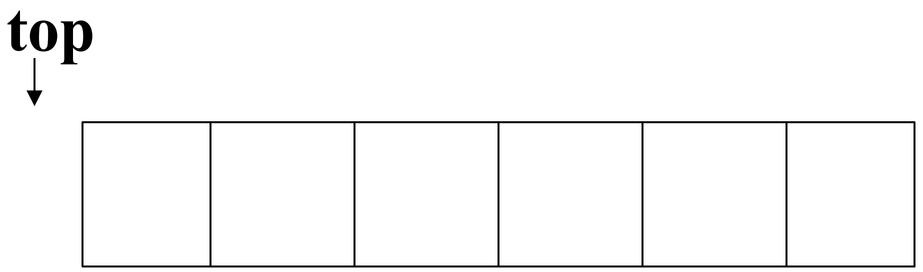
- It arises all the time in programming (e.g., see Weiss 3.6.3)
  - Recursive function calls
  - Syntax analysis of pairwise tags (XML)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion (see text)
- We can code up a reusable library
- We can communicate in high-level terms
  - "Use a stack and push numbers, popping for operators…"
  - Rather than, "create an array and keep indices to the…"

Stack Implementations

- stack as a linked list

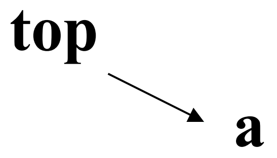


- stack as an array

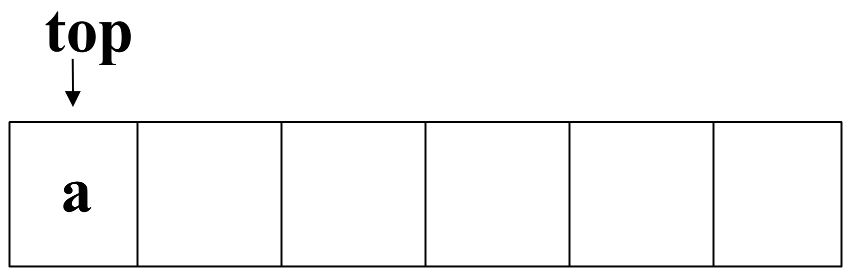


Stack Implementations

- stack as a linked list

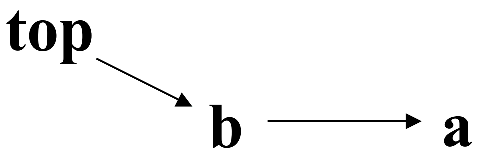


- stack as an array

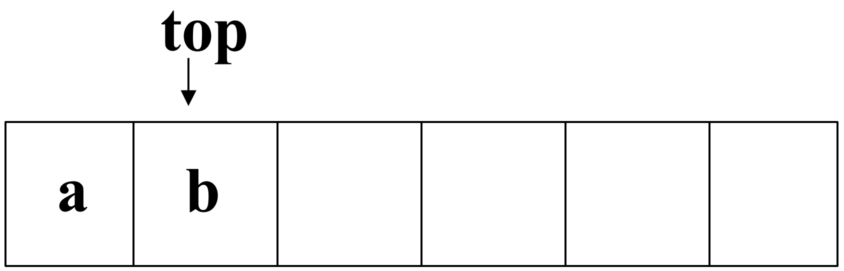


Stack Implementations

- stack as a linked list



- stack as an array



The Queue ADT

- Operations

  `create`

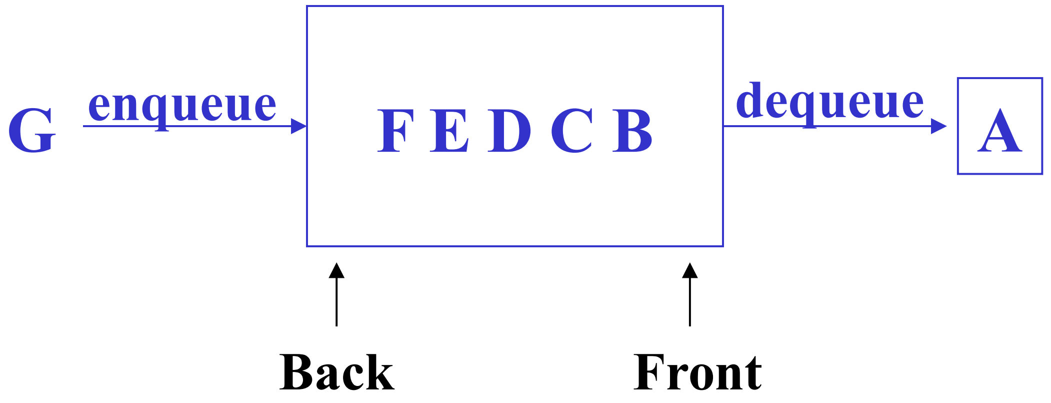
  `destroy`

  `enqueue`

  `dequeue`

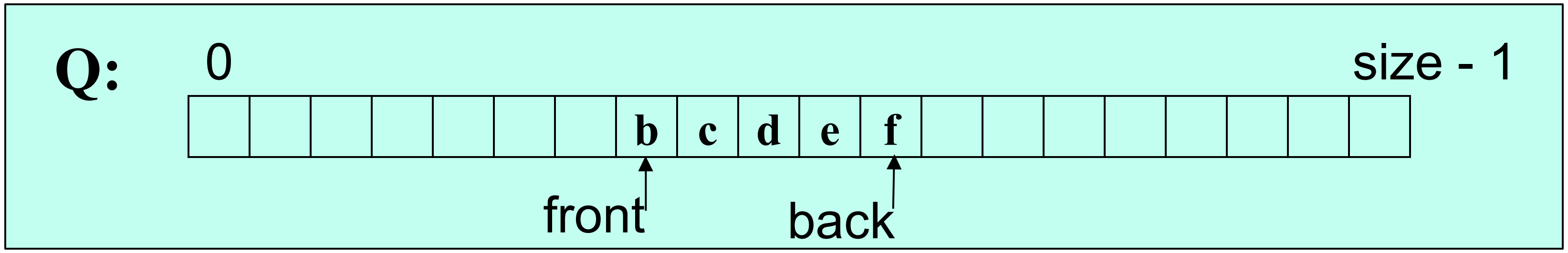
  `is_empty`

`What else?`



- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)

Circular Array Queue Data Structure

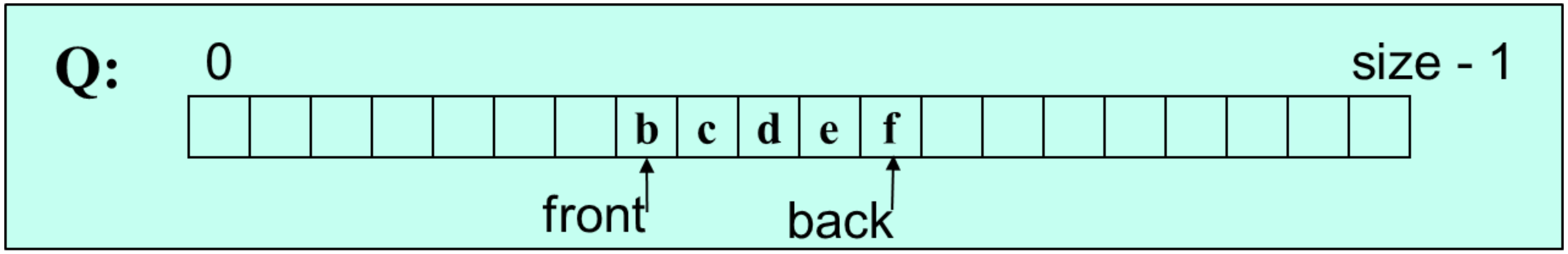


```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

```
// Basic idea only!
dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- What if ***array*** is full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the kth element in the queue?

Circular Array Example



enqueue('g')

o1 = dequeue( )

o2 = dequeue( )

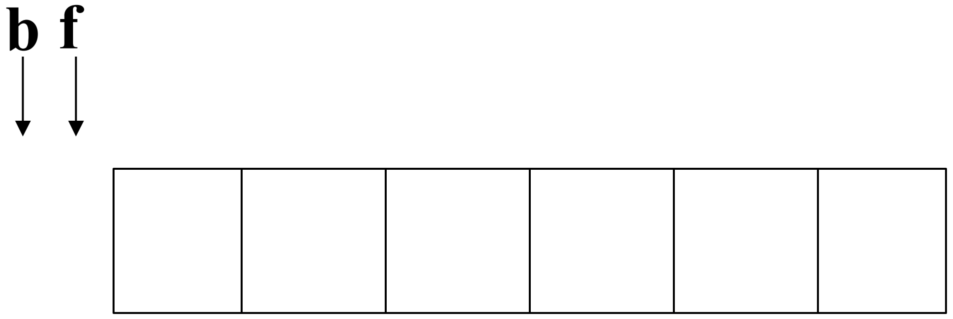
o3 = dequeue( )

o4 = dequeue( )

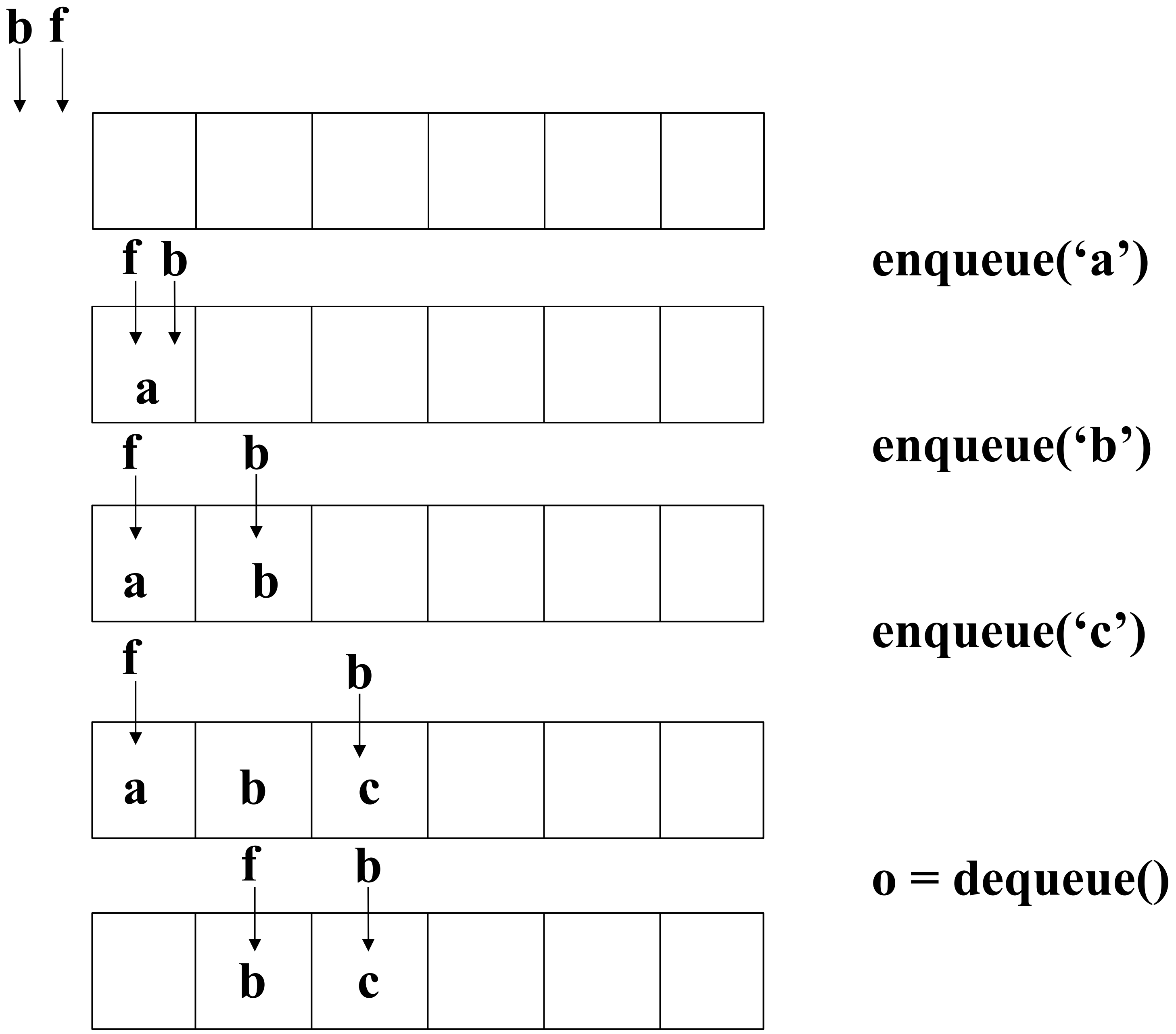
o5 = dequeue( )

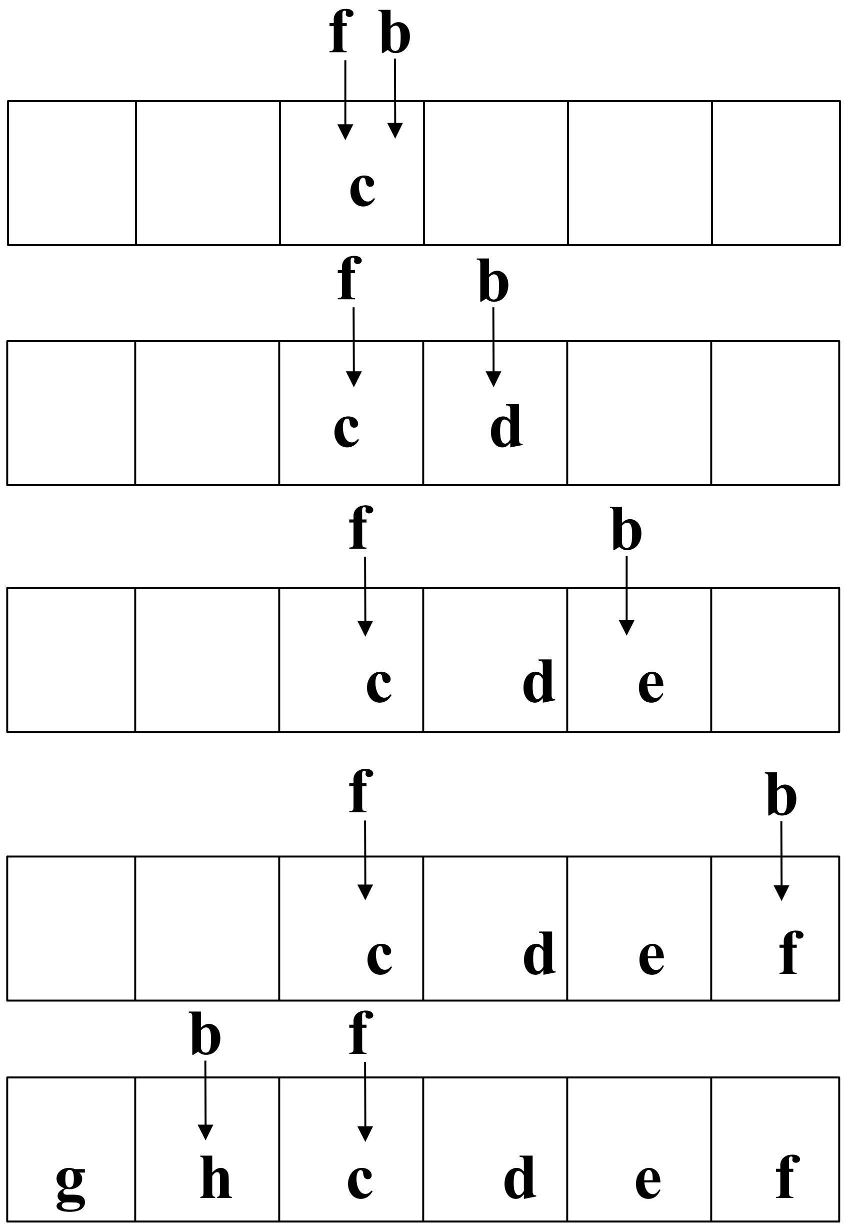
o6 = dequeue( )

In Class Practice



enqueue('a')
enqueue('b')
enqueue('c')
o = dequeue()
o = dequeue()
enqueue('d')
enqueue('e')
enqueue('f')
enqueue('g')
enqueue('h')
enqueue('i')


b f
f b
a
f b
a b
f
b
a b c
f b
b c
enqueue('a')
enqueue('b')
enqueue('c')
o = dequeue()



o = dequeue

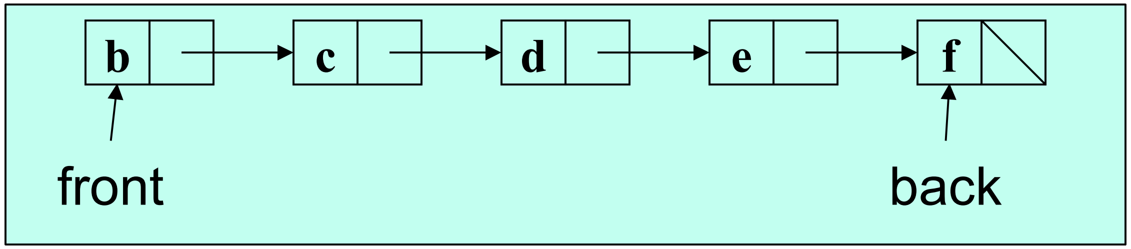
enqueue(‘d’)

enqueue(‘e’)

enqueue(‘f’)

enqueue(‘g’), enqueue(‘h’)

Linked List Queue Data Structure



```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// Basic idea only!
dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- Can ***list*** be full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the kth element in the queue?

Circular Array vs. Linked List

Array

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast
- Constant-time access to kth element (not in ADT!!)

List

- Always just enough space
- But more space per element
- Operations very simple / fast
- No constant-time access to kth element (not in ADT!!)

Conclusion

- Abstract data structures allow us to define a new data type and its operations.
- Each abstraction will have one or more implementations.
- Which implementation to use depends on the application, the expected operations, the memory and time requirements.
- Both stacks and queues have array and linked implementations.
- We'll look at other ordered-queue implementations later.