CSI 2103: Data Structures

Queues and Stacks (Ch 6)

Yonsei University
Spring 2022

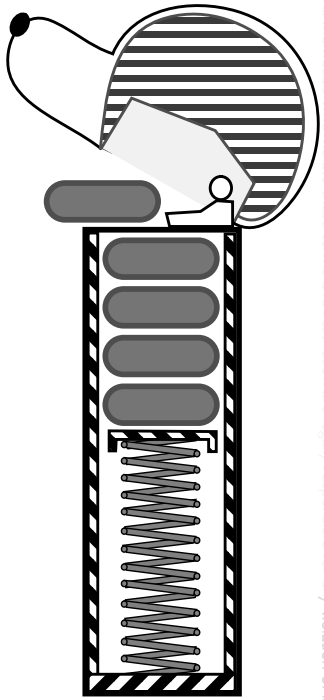
Seong Jae Hwang

Aims

- Study two simple but commonly used data structures
- Stacks
- Queues
- See how some applications naturally leverage the data structure properties
- Again, complexity analysis

Stacks

- Collection of objects that are inserted and removed according to the Last-In, First-Out (LIFO) principle
  - Ex: "Back" button of a web browser: as you visit new sites, they get "pushed" onto the stack. The browser keeps track of this stack. The "back" button "pops" the history from the stack so you can traverse back in the LIFO order.
  - Ex: "Undo" button of text editors: Your changes get "pushed" onto stack. The text editors keeps track of your changes in a stack. The "undo" (ctrl-z) operation reverts the changes in the order which they were done in a LIFO way



Stacks

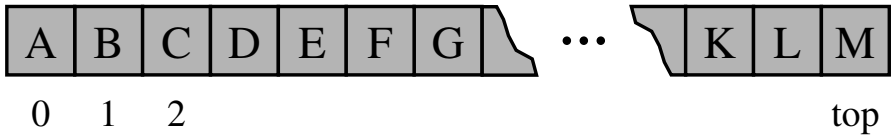
- The simplest of all data structures
- Abstract Data Type (ADT) that stores arbitrary objects with two methods:
  - S.push(e): add element e to the top of stack S
  - S.pop(): remove and return the top element from the stack S; error if empty
- Some other methods for additional information:
  - S.top(): return a reference to the top element of stack S, without removing it; error if empty
  - S.is_empty(): return true if stack S does not contain any elements
  - len(S): return the number of elements in stack S

Example

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

Array-Based Stack in Python

- Using a Python list
  - "top" is the right most cell



  - list has all the functions we need
  - Adapter design pattern: using an existing class
    - Basically all methods can be implemented with the Python list methods

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
len(S)	len(L)

    - Almost…define a new exception class

```
class Empty(Exception):
  """Error attempting to access an element from an empty container."""
  pass
```

Example

```
S = ArrayStack( )          # contents: [ ]
S.push(5)                  # contents: [5]
S.push(3)                  # contents: [5, 3]
print(len(S))              # contents: [5, 3];      outputs 2
print(S.pop( ))            # contents: [5];         outputs 3
print(S.is_empty( ))       # contents: [5];         outputs False
print(S.pop( ))            # contents: [ ];         outputs 5
print(S.is_empty( ))       # contents: [ ];         outputs True
S.push(7)                  # contents: [7]
S.push(9)                  # contents: [7, 9]
print(S.top( ))            # contents: [7, 9];      outputs 9
S.push(4)                  # contents: [7, 9, 4]
print(len(S))              # contents: [7, 9, 4];   outputs 3
print(S.pop( ))            # contents: [7, 9];      outputs 4
S.push(6)                  # contents: [7, 9, 6]
```

Analyzing the Array-Based Stack

- Push and pop: $O(1)$ amortized
  - Most of the time, they take $O(1)$
  - Occasionally (e.g., proportional to n), it is $O(n)$ following the list class
  - Still takes $O(1)$ in the long run
- Maybe too simple for some tasks
- But super fast if stack (and LIFO principle) is appropriate!

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

Example 0: Reversing File Order

- The LIFO naturally reverses a data sequence:
  - Push 1,2,3 -> Pop 3,2,1
- There is no "algorithm" to achieve this; storing and accessing with a specific data structure is enough!
- Aim: Given a list of filenames, reverse the order

```
1  def reverse_file(filename):
2    """Overwrite given file with its contents line-by-line reversed."""
3    S = ArrayStack( )
4    original = open(filename)
5    for line in original:
6      S.push(line.rstrip('\n'))          # we will re-insert newlines when writing
7    original.close( )
8
9    # now we overwrite with contents in LIFO order
10   output = open(filename, 'w')       # reopening file overwrites original
11   while not S.is_empty( ):
12     output.write(S.pop( ) + '\n')  # re-insert newline characters
13   output.close( )
```

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt (top)	

read first line of input -> push to S

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt (top)	
3	C.txt	B.txt (top) A.txt	

read first line of input -> push to S

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt (top)	
3	C.txt	B.txt (top) A.txt	
4		C.txt (top) B.txt A.txt	

read first line of input -> push to S

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt	
3	C.txt	B.txt A.txt	
4		C.txt B.txt A.txt	
5		B.txt A.txt	C.txt

pop from top of S -> write in output file

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt	
3	C.txt	B.txt A.txt	
4		C.txt B.txt A.txt	
5		B.txt A.txt	C.txt
6		A.txt	C.txt B.txt

pop from top of S -> write in output file

Example 0: Reversing File Order

Iteration	Input (Filename list)	Stack (Push/Pop from top)	Output (Filename list)
1	A.txt B.txt C.txt		
2	B.txt C.txt	A.txt	
3	C.txt	B.txt A.txt	
4		C.txt B.txt A.txt	
5		B.txt A.txt	C.txt
6		A.txt	C.txt B.txt
7			C.txt B.txt A.txt

pop from top of S -> write in output file

Example 1: Matching Parentheses

- Let's see an example where a stack is a natural data structure
- Goal: Check if the given arithmetic expression has the matching grouping symbols
  - "( )", "{ }", "[ ]"
  - Correct: ( ) (( )) { ([( )]) }
  - Incorrect:) (( )) { ([( )]) }

```
1  def is_matched(expr):
2    """Return True if all delimiters are properly match; False otherwise."""
3    lefty = '({['                                  # opening delimiters
4    righty = ')}]'                                 # respective closing delims
5    S = ArrayStack( )
6    for c in expr:
7      if c in lefty:
8        S.push(c)                                  # push left delimiter on stack
9      elif c in righty:
10       if S.is_empty( ):
11         return False                             # nothing to match with
12       if righty.index(c) != lefty.index(S.pop( )):
13         return False                             # mismatched
14   return S.is_empty( )                           # were all symbols matched?
```

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:
- Ops:
- S:

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: (
- Ops: S.push("(")
- S: (

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:)
- Ops: S.pop() and check if "(" != ")"
- S: (

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: (
- Ops: S.push("(")
- S: (

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: (
- Ops: S.push("(")
- S: ((

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:)
- Ops: S.pop() and check if "(" != ")"
- S: ({

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:)
- Ops: S.pop() and check if "(" != ")"
- S: {

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: {
- Ops: S.push(“{”)
- S: {

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: [
- Ops: S.push("[")
- S: { [

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:]
- Ops: S.pop() and check if "[" != "]"
- S: { [

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c: }
- Ops: S.pop() and check if "{" != "}"
- S: {

Example 1: Matching Parentheses

- expr: ( ) (( )) { [ ] }
- c:
- Ops:
- S:

- Success if finish without returning False
- We know exactly which "lefty" we need to match for the current "righty"
- Stack is all we need to (1) store the "lefty" in the (2) LIFO order to look for the most recent "lefty" to check
- Time-Complexity? $O(n)$

Example 2: HTML Tag Matching

- Another application of matching delimiters
- HTML: the standard format for hyperlinked documents on the Internet
- A correct HTML has the matching HTML tags:
  - <name> should pair with a matching </name>
- Examples:
  - <body> … </body>: document body
  - <h1> … </h1>: section header
  - … : center justify
  - … : paragraph
  - … : numbered (ordered) list
  - … : list item

Example 2: HTML Tag Matching

```
<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>
```

The Little Boat

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.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Example 2: HTML Tag Matching

- Similar logic to the parenthesis checker
  - Instead of checking the parenthesis characters, we check the tags

```
1  def is_matched_html(raw):
2    """Return True if all HTML tags are properly match; False otherwise."""
3    S = ArrayStack( )
4    j = raw.find('<')                    # find first '<' character (if any)
5    while j != -1:
6      k = raw.find('>', j+1)             # find next '>' character
7      if k == -1:
8        return False                     # invalid tag
9      tag = raw[j+1:k]                   # strip away < >
10     if not tag.startswith('/'):        # this is opening tag
11       S.push(tag)
12     else:                              # this is closing tag
13       if S.is_empty( ):
14         return False                   # nothing to match with
15       if tag[1:] != S.pop( ):
16         return False                   # mismatched delimiter
17     j = raw.find('<', k+1)             # find next '<' character (if any)
18   return S.is_empty( )                 # were all opening tags matched?
```

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

<h1>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<h1>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~ ~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

<p>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<p>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<li>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<li>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<li>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

<body>

~~<ol>~~

Example 2: HTML Tag Matching

```
<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>
```

Stack:

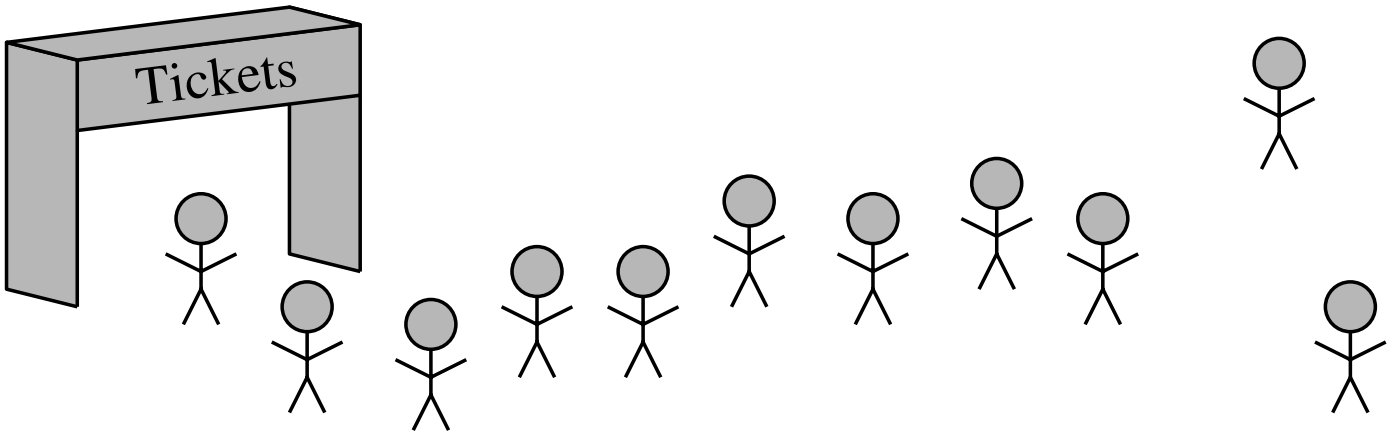
~~<body>~~

When finished, check if the stack is empty

Time Complexity? $O(n)$

Queues

- Queue: another simple data structure; a collection of objects that are inserted and removed according to the First-In, First-Out (FIFO) principle
- Very common "data structure" seen in daily life
  - waiting in line
  - "First come, first serve"



Queues

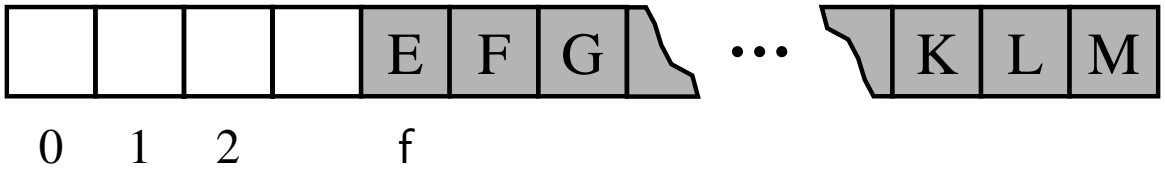
- Also a simple data structure
- Another ADT that stores arbitrary objects with two methods:
  - **Q.enqueue(e)**: add element e to the back of queue Q
  - **Q.dequeue()**: remove and return the first element from queue Q; error if empty
- Some other methods for additional information:
  - **Q.first()**: return a reference to the front element of queue Q, without removing it; error if empty
  - **Q.is_empty()**: return true if queue Q does not contain any elements
  - **len(Q)**: return the number of elements in queue Q

Example

Operation	**Return Value**	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

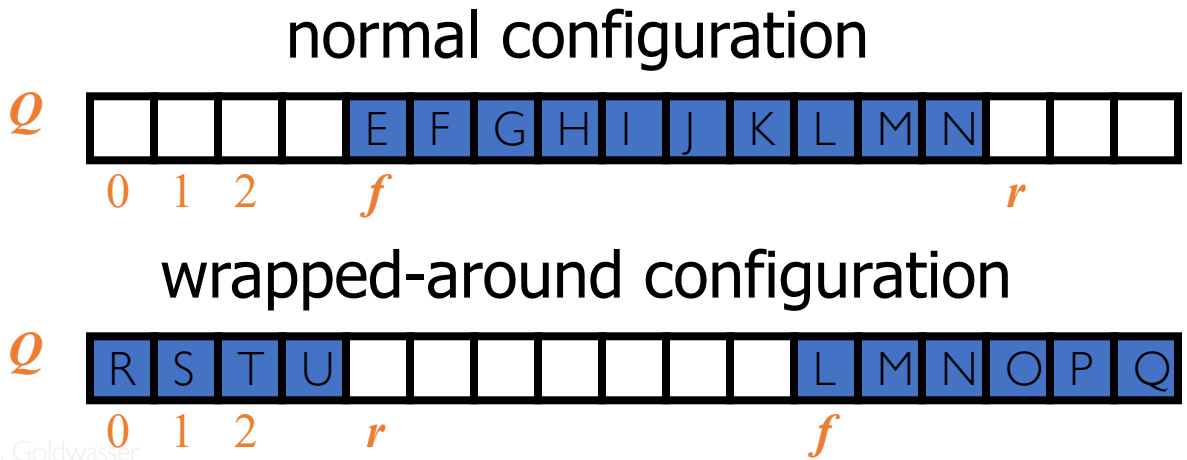
Array-Based Queue in Python

- For the stack ADT, we used append(e) and pop()
- Can we do the same to implement a queue?
  - enqueue(e): append(e) to add e to the end of the list
  - dequeue(e): pop(0) to intentionally remove the first element of the list
    - This requires n pop() operations to reach the first element!
    - Time complexity? $O(n)$ Good or bad?
- Slight modification: Avoid using pop by
  - Replacing the dequeued entry with None reference and
  - Keep tracking the "front index" f
- Another issue: The list keeps growing!
  - Some applications will not need very long queues



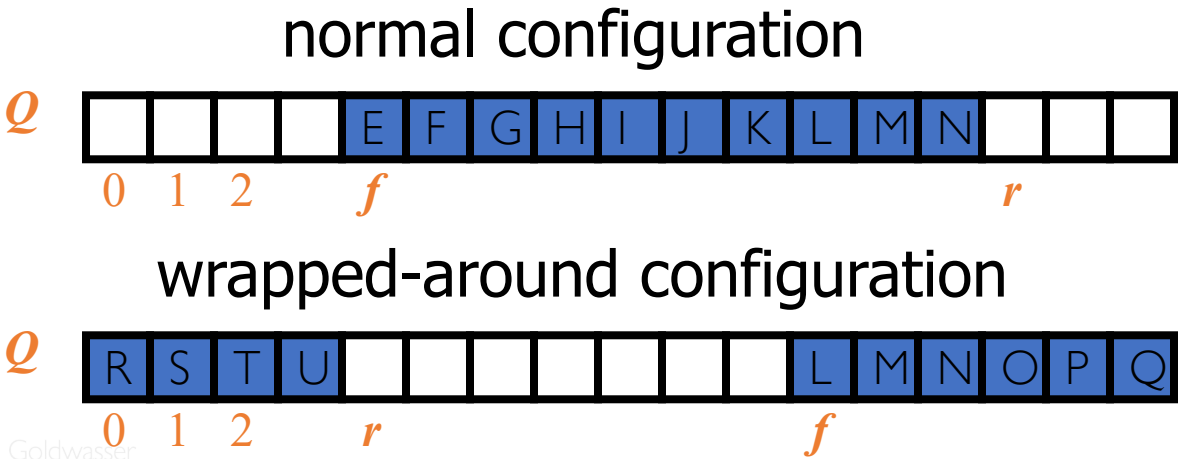
Array-Based Queue in Python

- Use an array of size *N* in a circular fashion
  - Assume *N* > number of elements
  - dequeue shifts the front as usual
  - New elements are enqueued toward the "end" (or rear), progressing from the front to index N-1 and continuing at index 0
    - See the example below: R,S,T,U are enqueued at index 0,1,2,3
- This is easy to implement! Use modulo operator



Array-Based Queue in Python

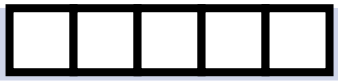
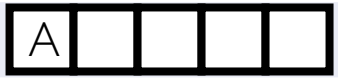
- dequeue example:
  - We dequeue at f and also "advance" the front index: f = (f + 1) % N
  - % is the modulo operator in Python to compute the remainder
    - 14 % 3 returns 2
  - Ex: when N = 10,
    - f = 8: dequeue at f = 8, and move f = (8 + 1) % 10 => 9
    - f = 9: dequeue at f = 9, and move f = (9 + 1) % 10 => 0 (wrapped!)
    - f = 0: dequeue at f = 0, and move f = (0 + 1) % 10 => 1



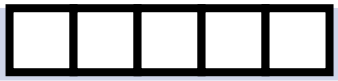
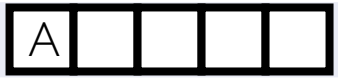
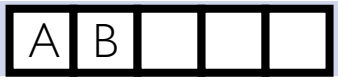
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		

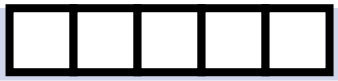
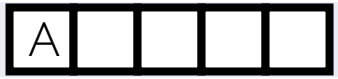
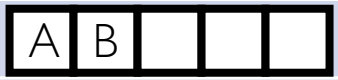
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0

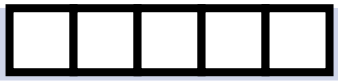
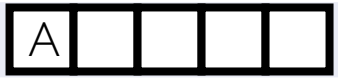
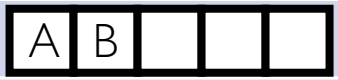
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0

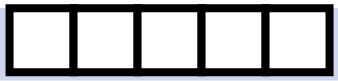
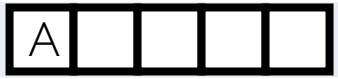
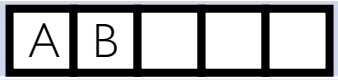
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0

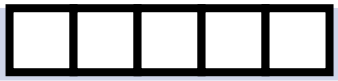
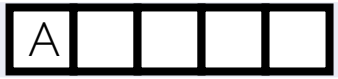
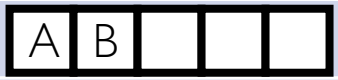
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1

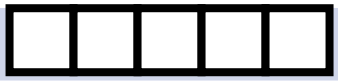
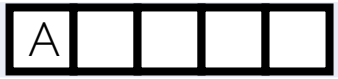
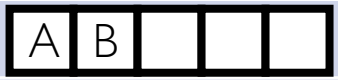
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2

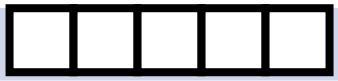
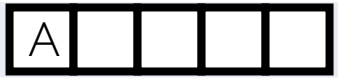
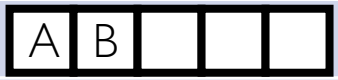
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2

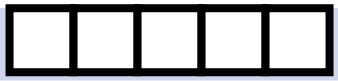
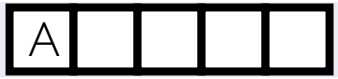
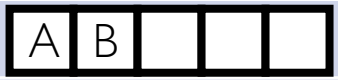
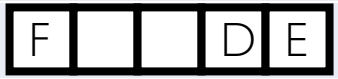
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2

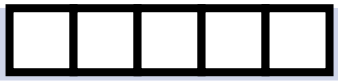
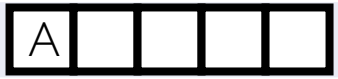
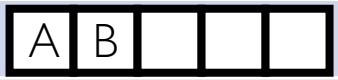
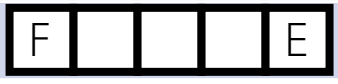
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2

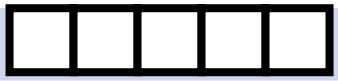
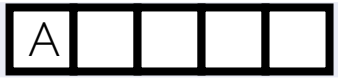
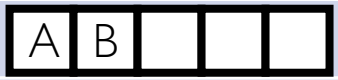
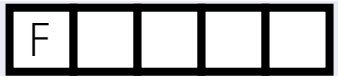
Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2
Q.dequeue()		3

Example

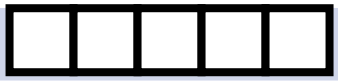
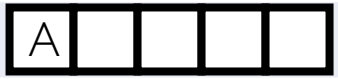
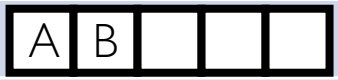
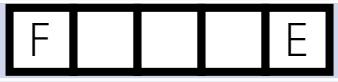
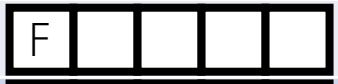
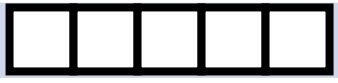
Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2
Q.dequeue()		3
Q.dequeue()		4

Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2
Q.dequeue()		3
Q.dequeue()		4
Q.dequeue()		0

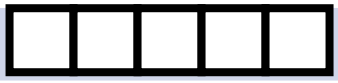
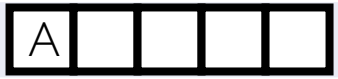
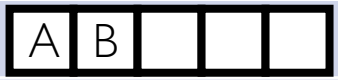
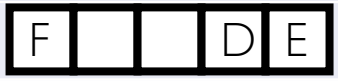
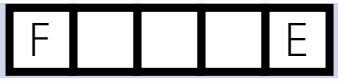
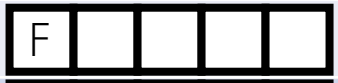
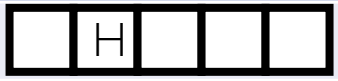
when f wraps around →

Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2
Q.dequeue()		3
Q.dequeue()		4
Q.dequeue()		0
Q.dequeue()		1

when f wraps around →

Example

Operation	Queue	f
Q = CircularArrayQueue(5)		
Q.enqueue(A)		0
Q.enqueue(B)		0
Q.enqueue(C)		0
Q.dequeue()		1
Q.dequeue()		2
Q.enqueue(D)		2
Q.enqueue(E)		2
Q.enqueue(F)		2
Q.dequeue()		3
Q.dequeue()		4
Q.dequeue()		0
Q.dequeue()		1
Q.enqueue(H)		1

when f wraps around →

Array-Based Queue in Python

reference to the actual list instance of data with a default capacity

two variables to keep track of front and size
size: 0
front: 0

trivial methods using size

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
```

Array-Based Queue in Python

- L35: save "answer" before front is updated
- L36: remove the reference to that object
  - Python maintains a count of the number of references to each object. Reaching it to be 0 reclaim that memory
  - Since we are not responsible for storing a dequeued element, we explicitly remove the reference to it from our list
- L37: update front
- L38: update size

```
28    def dequeue(self):
29      """Remove and return the first element of the queue (i.e., FIFO).
30
31      Raise Empty exception if the queue is empty.
32      """
33      if self.is_empty( ):
34        raise Empty('Queue is empty')
35      answer = self._data[self._front]
36      self._data[self._front] = None                # help garbage collection
37      self._front = (self._front + 1) % len(self._data)
38      self._size -= 1
39      return answer
```

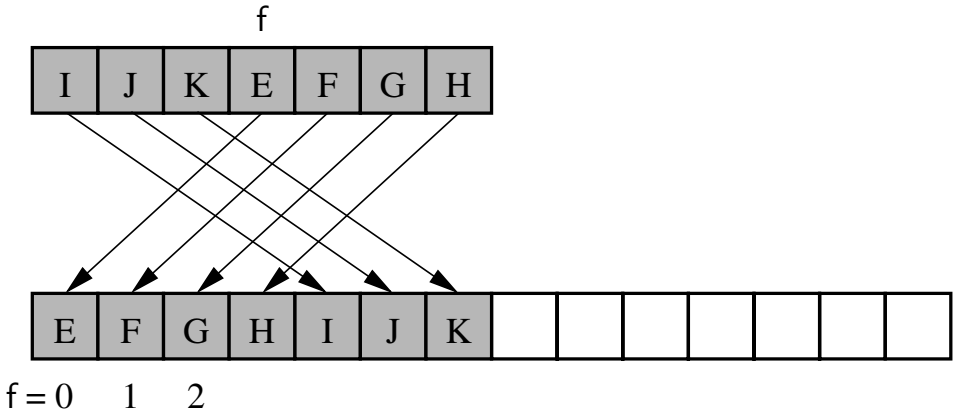
Array-Based Queue in Python

- Key to enqueue is to compute the proper index to place the new element:
  - avail = (front + size) % N
- L42-43: resize if full (next slide)
- L44: compute the index (avail) to enqueue
  - self._size is prior to adding this new element
  - ex: front = 8, size = 3, N = 10. Then, avail should be 1 since indices 8, 9, 0 are occupied. Thus, avail = (8 + 3) % 10 => 1
- L45: add the element
- L46: update size

```
40    def enqueue(self, e):
41      """Add an element to the back of queue."""
42      if self._size == len(self._data):
43        self._resize(2 * len(self.data))          # double the array size
44      avail = (self._front + self._size) % len(self._data)
45      self._data[avail] = e
46      self._size += 1
```

Array-Based Queue in Python

- When the array size needs to grow, we rely on a standard technique of doubling the storage capacity
- Note: cannot naively copy/paste since the circular indexing needs to hold after copying the elements to a larger array



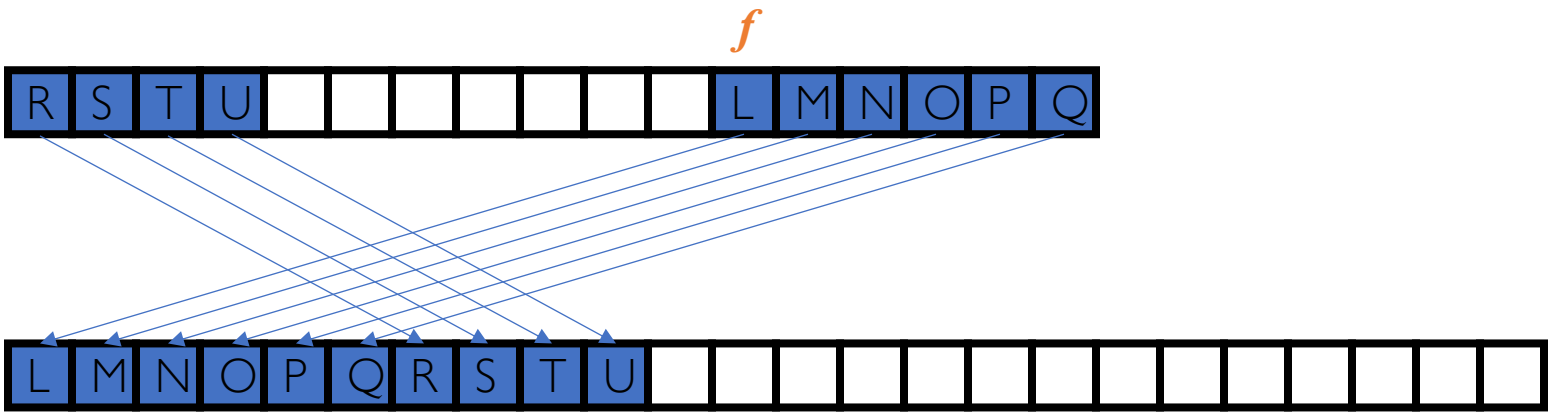
- old queue: "walk" from f until end of queue using modulo
- new queue: 0 to size

```
48    def _resize(self, cap):                    # we assume cap >= len(self)
49      """Resize to a new list of capacity >= len(self)."""
50      old = self._data                         # keep track of existing list
51      self._data = [None] * cap                # allocate list with new capacity
52      walk = self._front
53      for k in range(self._size):              # only consider existing elements
54        self._data[k] = old[walk]              # intentionally shift indices
55        walk = (1 + walk) % len(old)           # use old size as modulus
56      self._front = 0                          # front has been realigned
```

Array-Based Queue in Python

front = 11
size = 10
N = 17

	L	M	N	O	P	Q	R	S	T	U
Old	11	12	13	14	15	16	0	1	2	3
New	0	1	2	3	4	5	6	7	8	9



Shrinking the Array?

- The space complexity may be inefficient
  - array size proportional to the max # of elements we ever stored
  - array size >> current # of elements
- Shrinking can also be achieved using resize
  - Rule of thumb: reduce to half its current size when the # of elements is ¼ its capacity

```
if 0 < self._size < len(self._data) // 4:
  self._resize(len(self._data) // 2)
```

Complexity Analysis

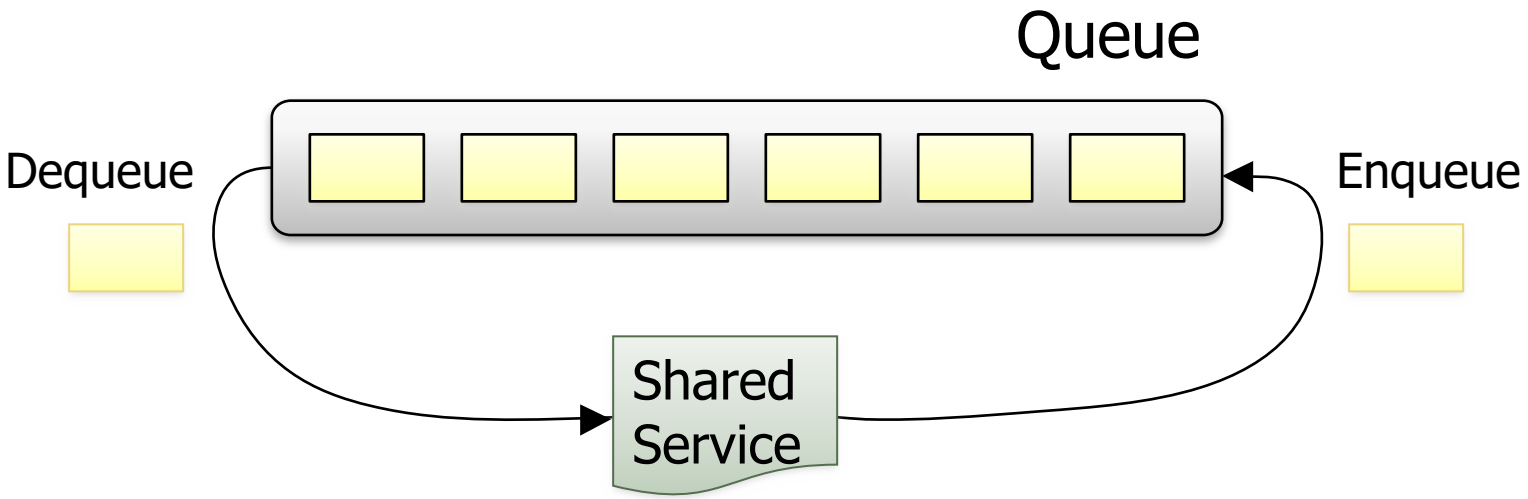
- All main methods have constant time complexity
  - enqueue and dequeue have amortized $O(1)^*$
  - Each resize is $O(n)$ but we don't do this often
    - Resize is $O(n)^*$ for n appends
    - Resize is $O(1)^*$ for 1 append
  - Space complexity is $O(n)$ assuming we occasionally shrink

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Example: Round Robin Scheduler

- Another example where the task naturally fits the data structure
  - Simple data structure for simple tasks
- To continuously service the elements in an order
  - Take the element from the queue: e = Q.dequeue()
  - Service element e
  - Put the element back into the queue: Q.enqueue(e)



Summary

- Simple but commonly used data structures with fast time complexity
- Stacks
  - LIFO
  - Surprisingly useful applications such as parenthesis matching, HTML tag matching
- Queues
  - FIFO

- We will continuously see how data structures with seemingly limited functionalities nicely enable certain tasks