SE274 Data Structure

Lecture 3: Stacks, Queues, Deques (+array)

Mar 16, 2020

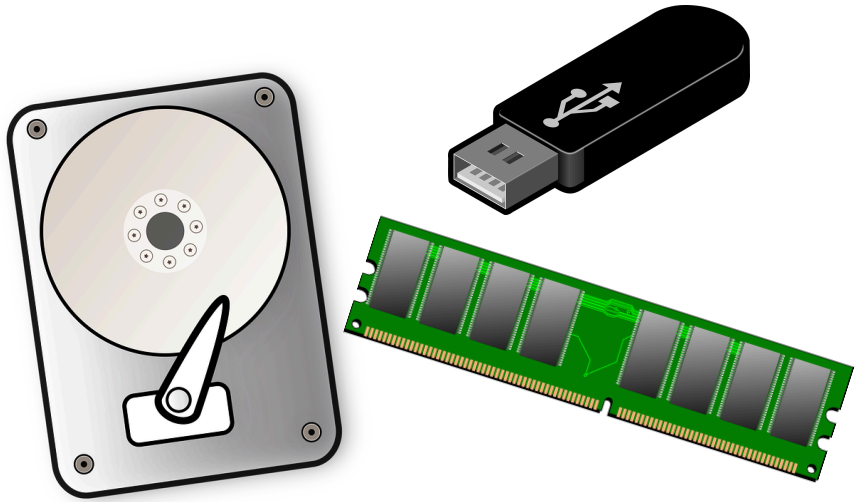
Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

Before we start…

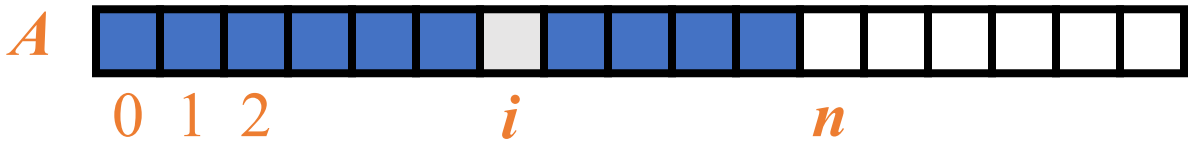
- Memory management of Array in Python

0100101010111011…



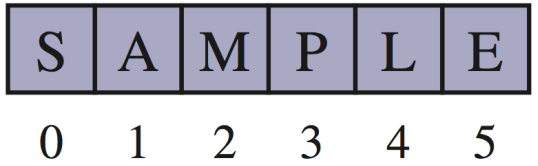
Python Sequence Classes

- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as A[i]
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.

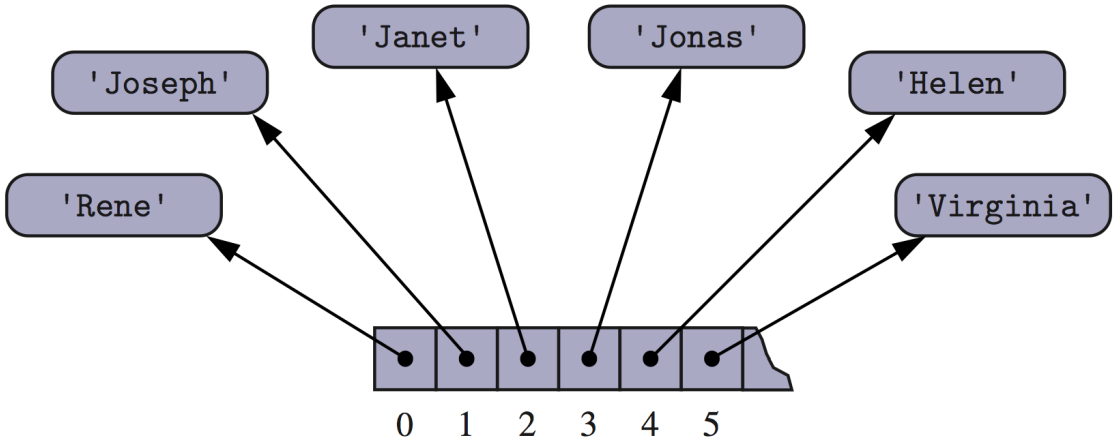


Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



Compact Arrays

- Primary support for compact arrays is in a module named **array**.
  - That module defines a class, also named array, providing compact storage for arrays of primitive data types.
- The constructor for the array class requires a type code as a first parameter, which is a character that designates the type of data that will be stored in the array.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

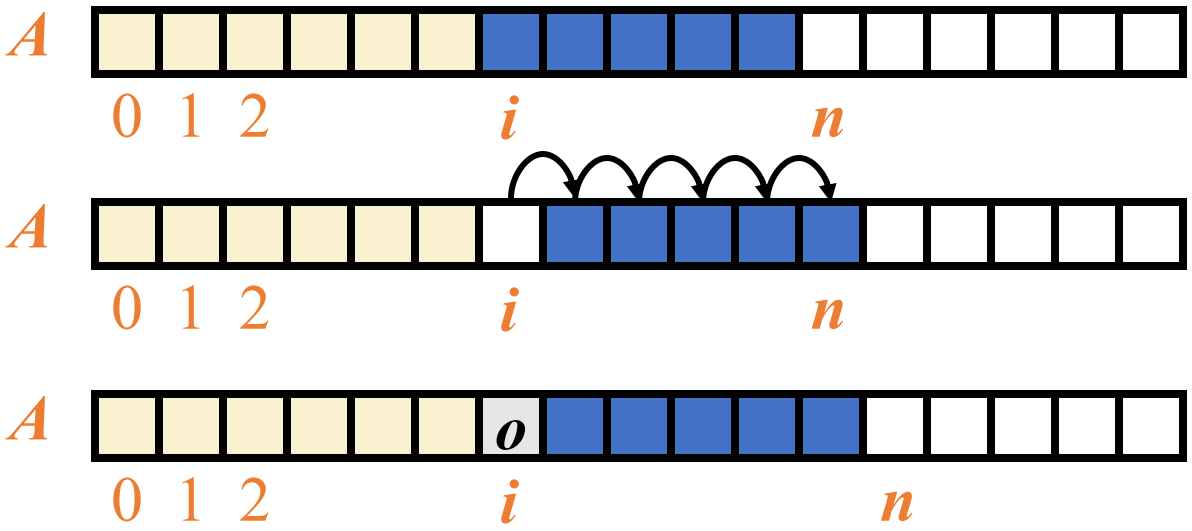
Type Codes in the array Class

- Python's array class has the following type codes:

Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

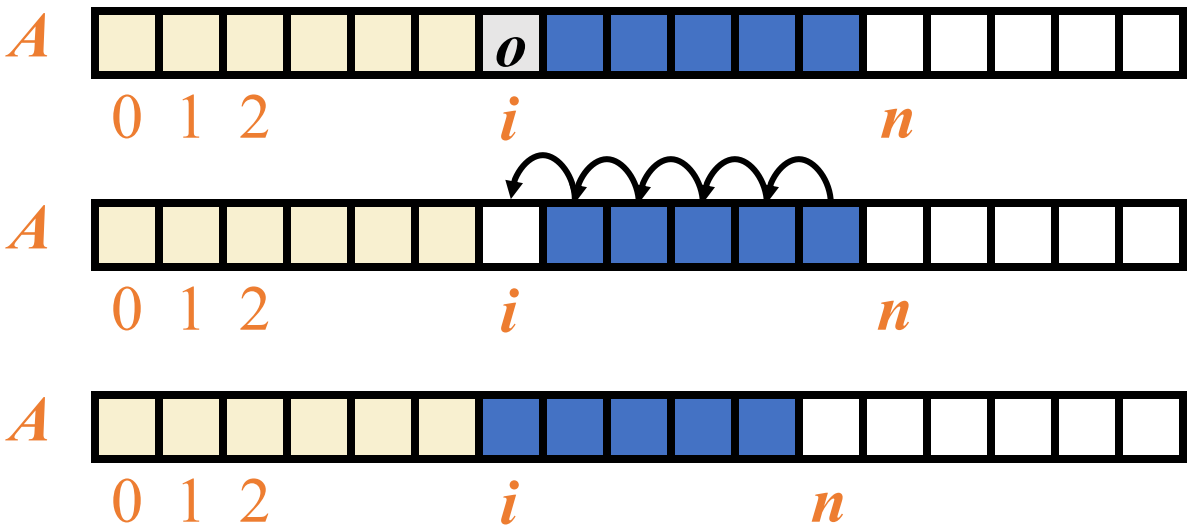
Insertion

- In an operation *add*(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In an operation *remove*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In an array based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at I takes $O(1)$ time
  - *add* and *remove* run in $O(n)$ time in worst case
- In an *add* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

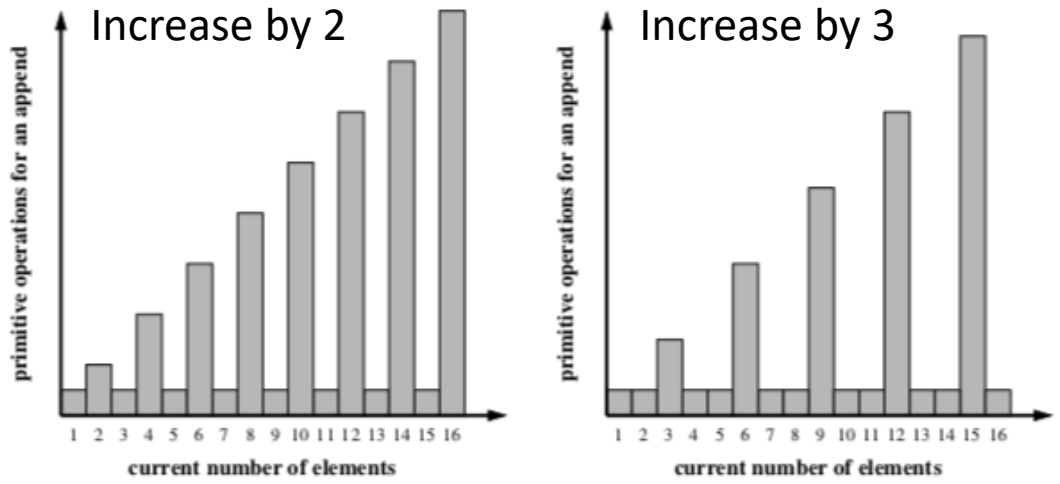
Algorithm *add*(o)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

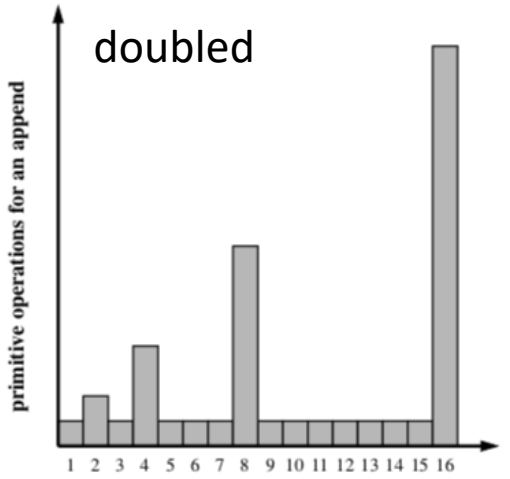
Incremental vs. Doubling

Incremental



Add: O(n)
* amortized time analysis

Doubling



Add: O(1)
* amortized time analysis

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

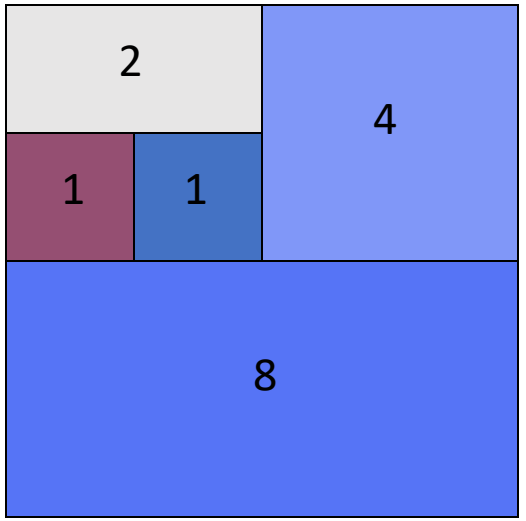
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



List and Tuple

The same: sequence data types (=stored as an array)

List

- Defined as [A, B, C, …]
- Mutable

```
1 list_var = ['A', 'B', 'C']
2 print(list_var)
```

```
['A', 'B', 'C']
```

```
1 list_var[0] = 'AA'
2 print(list_var)
```

```
['AA', 'B', 'C']
```

```
1 list_var.append('D')
2 print(list_var)
```

```
['AA', 'B', 'C', 'D']
```

Tuple

- Defined as (A, B, C, …)
- Immutable

```
1 tuple_var = ('A', 'B', 'C')
2 print(tuple_var)
```

```
('A', 'B', 'C')
```

```
1 tuple_var[0] = 'AA'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-415b8969b049> in <module>
----> 1 tuple_var[0] = 'AA'

TypeError: 'tuple' object does not support item assignment
```

List and Tuple

List

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

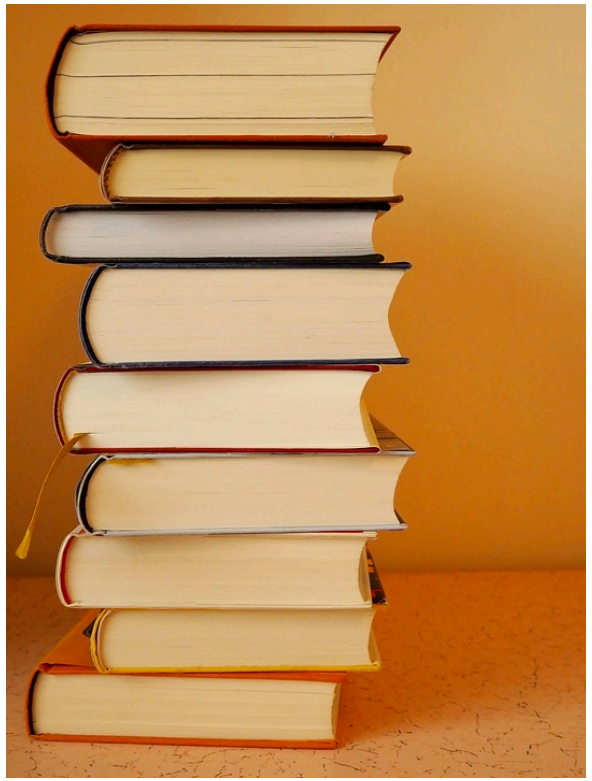
Tuple

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

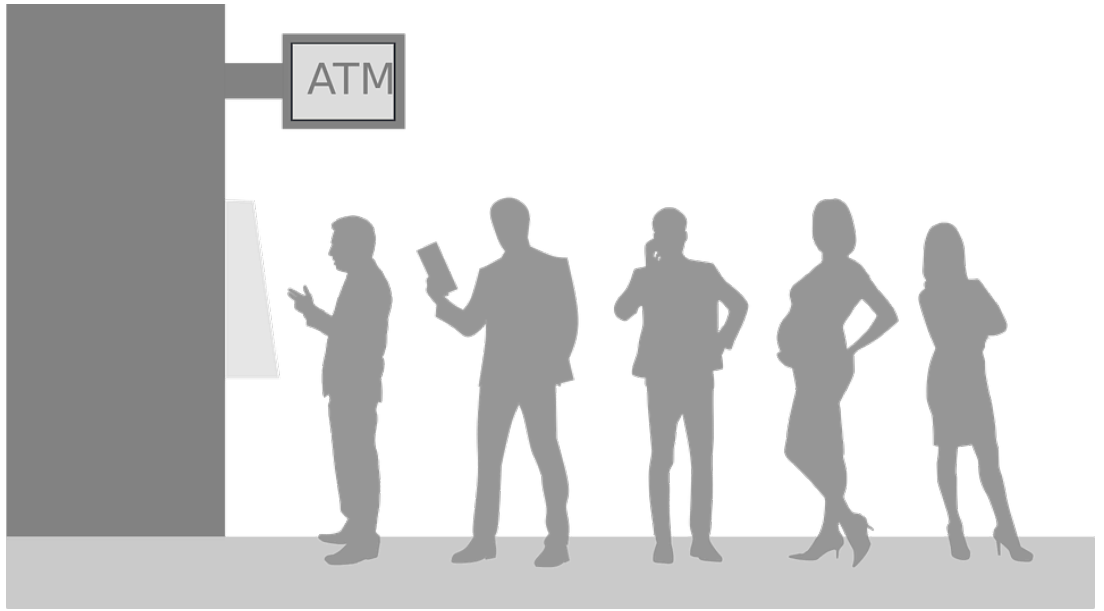
Stack

Stack and Queue, the concepts

Stack



Queue

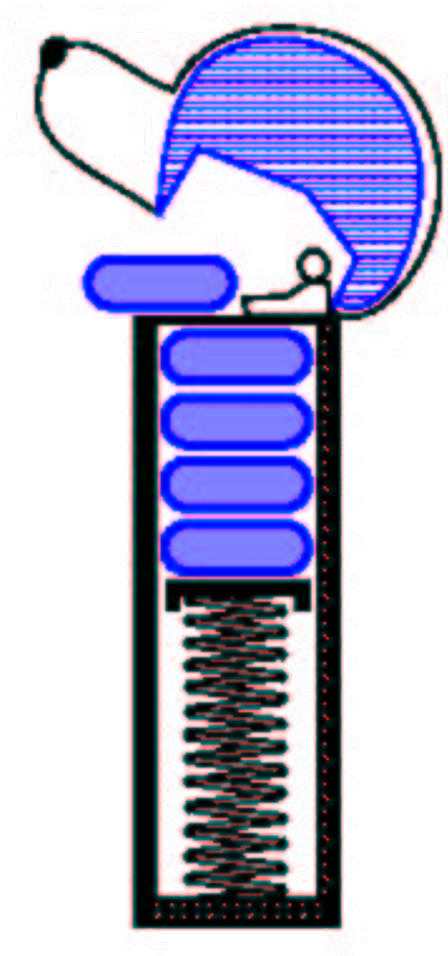


Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

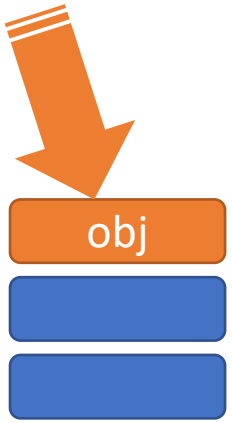
The Stack ADT



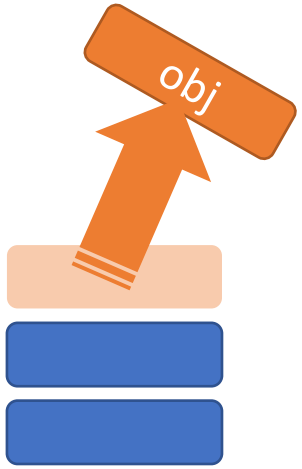
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- **LIFO**
  = Last In First Out

- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

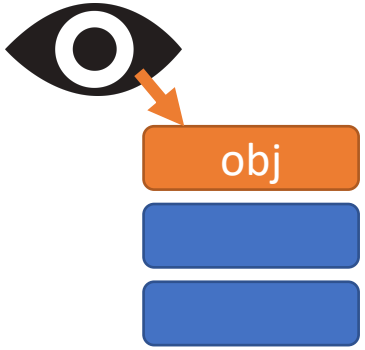
The Stack ADT



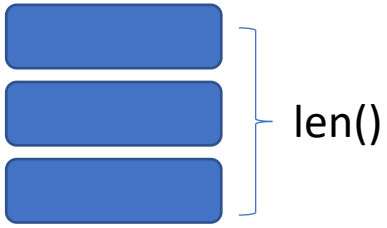
push(obj)



obj = pop()



obj = top()



- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

Example

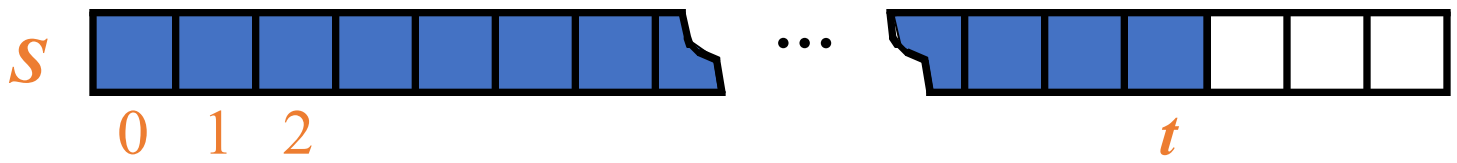
Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)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

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language that supports recursion
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

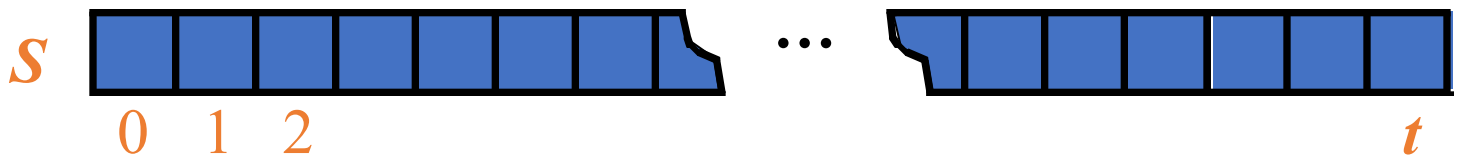
Array-based Stack

- 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



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.



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 $O(1)$ (amortized in the case of a push)

Array-based Stack in Python

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

Parentheses Matching

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

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
    if *X*[*i*] is an opening grouping symbol **then**
        S.push(*X*[*i*])
    else if *X*[*i*] is a closing grouping symbol **then**
        if *S*.is_empty() **then**
            return false {nothing to match with}
        if *S*.pop() does not match the type of *X*[*i*] **then**
            return false {wrong type}
if *S*.isEmpty() **then**
    return true {every symbol matched}
else return false {some symbols were never matched}

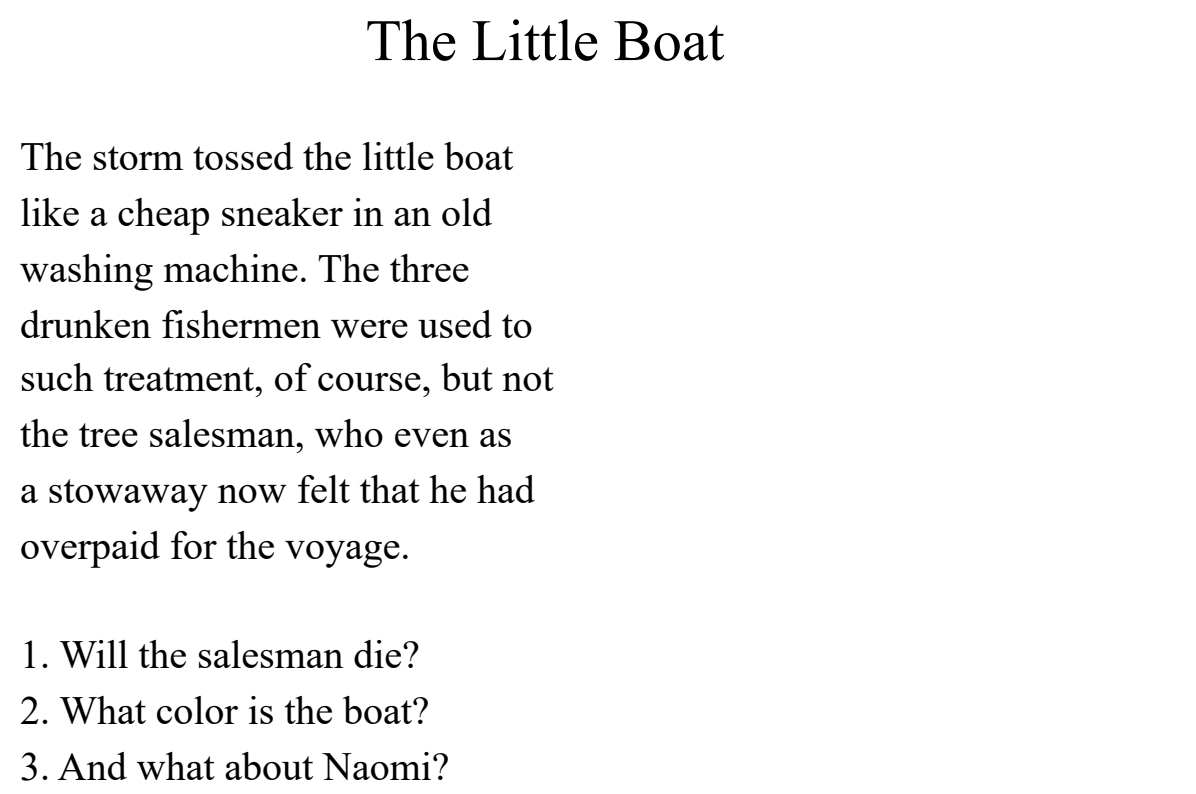
Parentheses Matching in Python

```
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?
```

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>
```



Tag Matching Algorithm in Python

```
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?
```

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

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

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.

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)

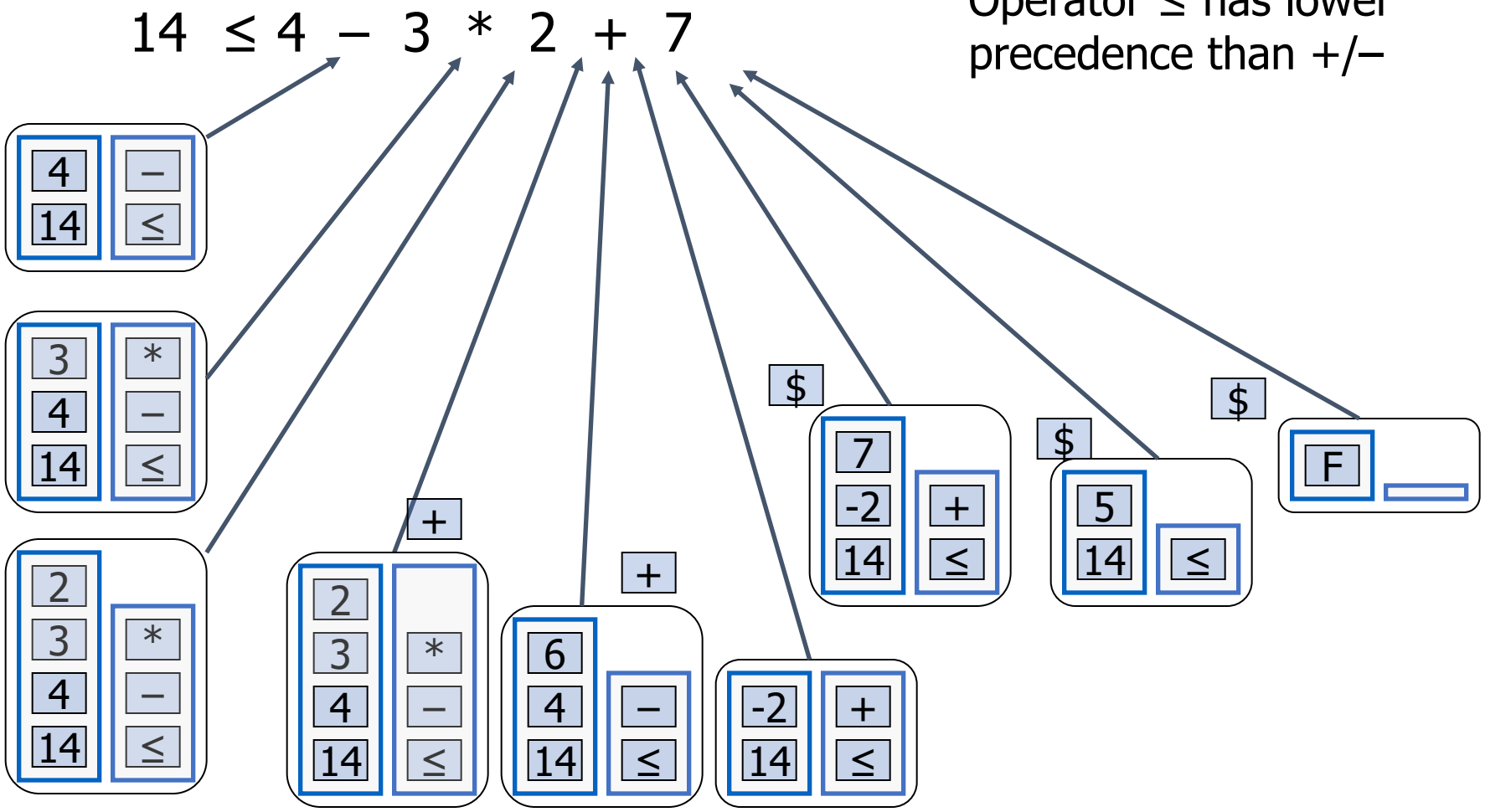
Output: the value of the expression

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

Algorithm on an Example Expression

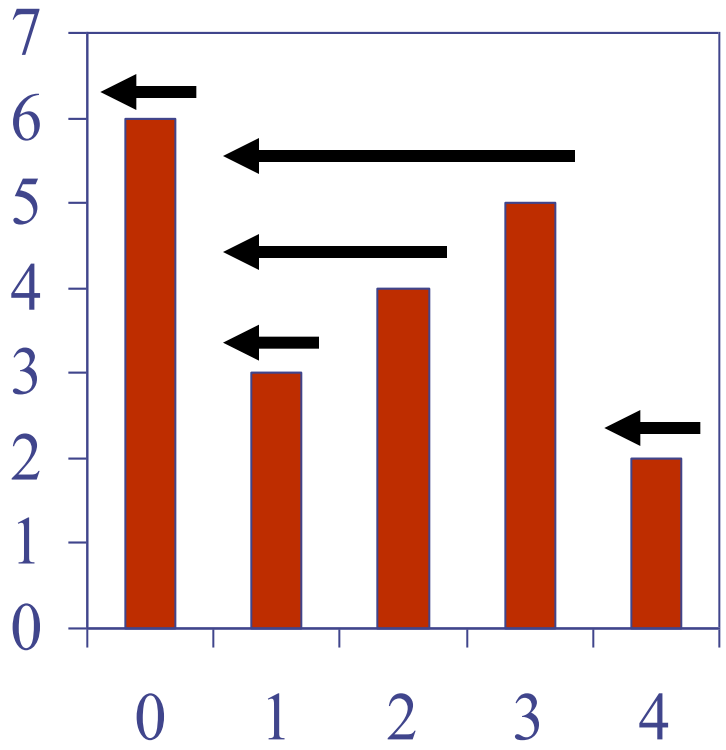
Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/–



Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

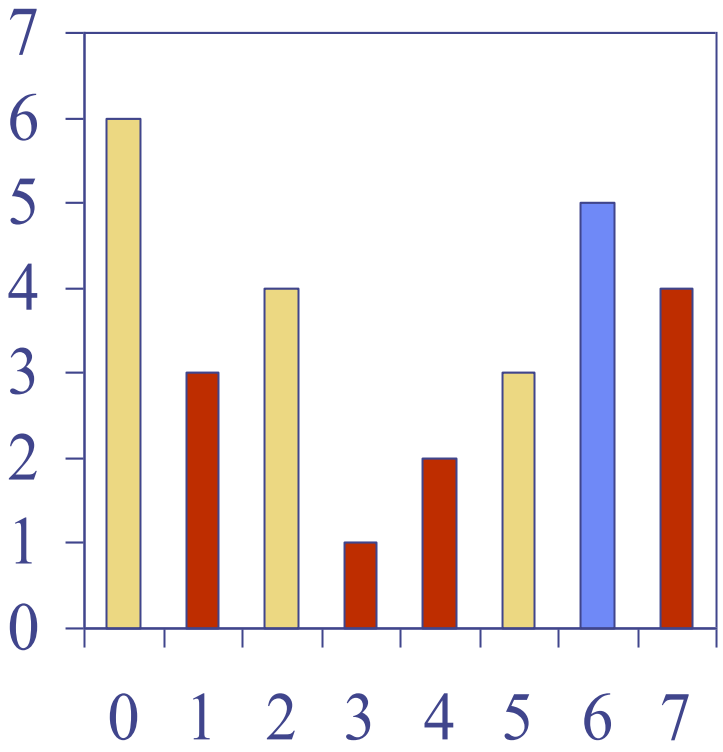
Quadratic Algorithm

```
Algorithm spans1(X, n)                                  #
    Input array X of n integers
    Output array S of spans of X                        #
    S ← new array of n integers                         n
    for i ← 0 to n − 1 do                               n
        s ← 1                                           n
        while s ≤ i ∧ X[i − s] ≤ X[i]                   1 + 2 + …+ (n − 1)
            s ← s + 1                                   1 + 2 + …+ (n − 1)
        S[i] ← s                                        n
    return S                                            1
```

- Algorithm *spans1* runs in $O(n^2)$ time

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack



Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                          #
    S ← new array of n integers                 n
    A ← new empty stack                         1
    for i ← 0 to n − 1 do                       n
        while (¬A.is_empty() ∧
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.is_empty() then                    n
            S[i] ← i + 1                        n
        else
            S[i] ← i − A.top()                  n
        A.push(i)                               n
    return S                                    1
```

Queue

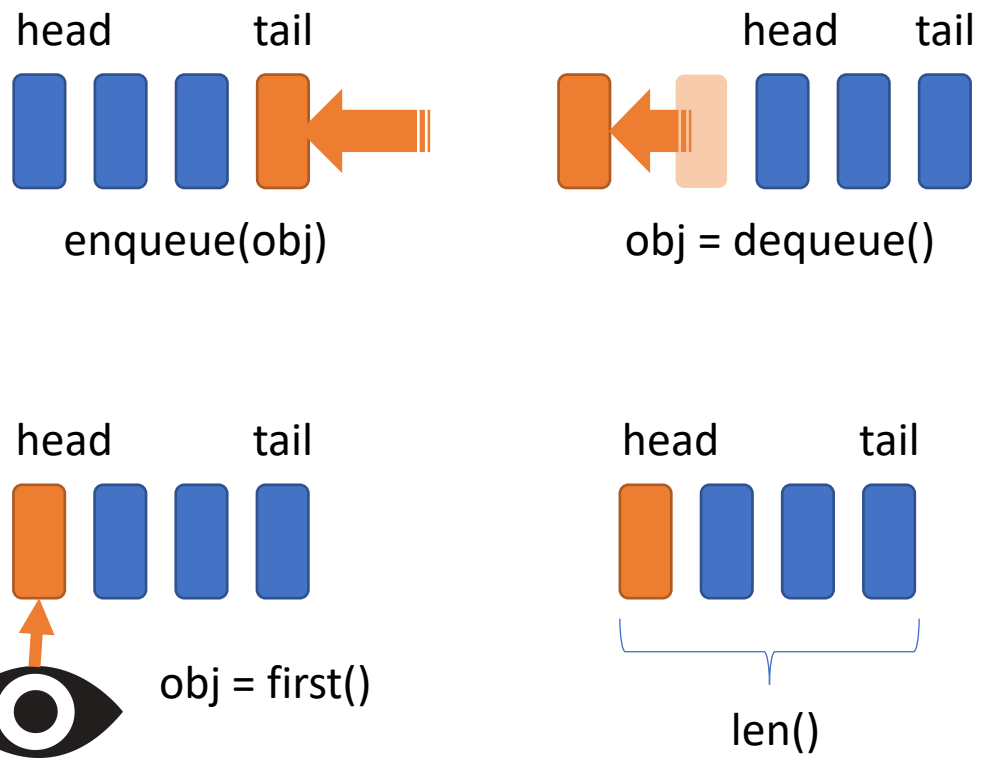
The Queue ADT

- 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
- **FIFO**
  = First In First Out

- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

- 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 len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

The Queue ADT



- 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 len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

Example

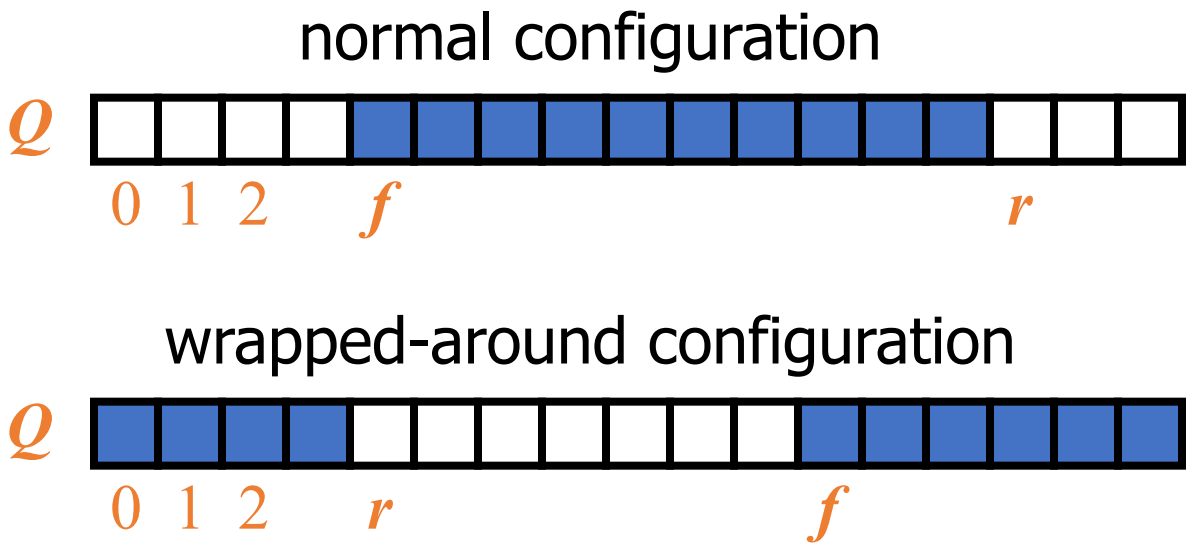
Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)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

Applications of Queues

- 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 rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

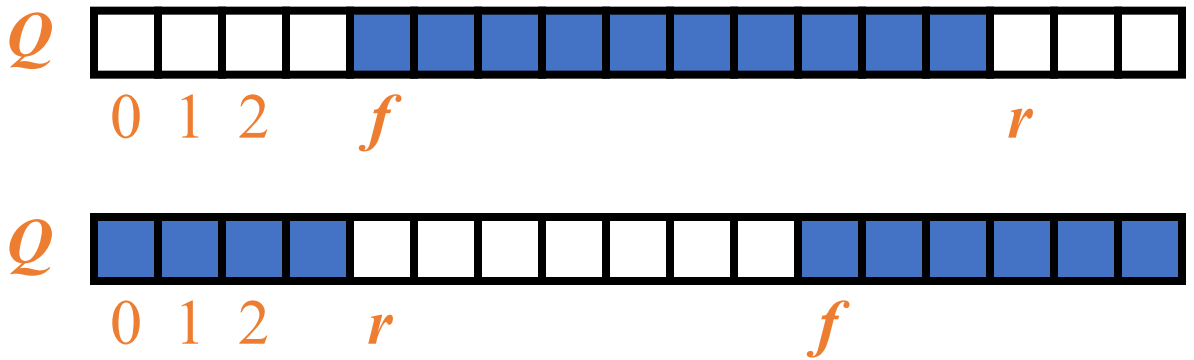


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

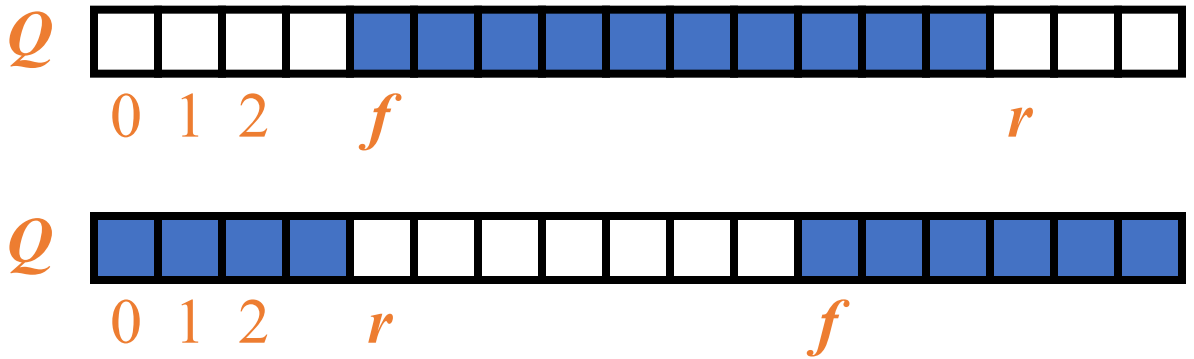
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

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

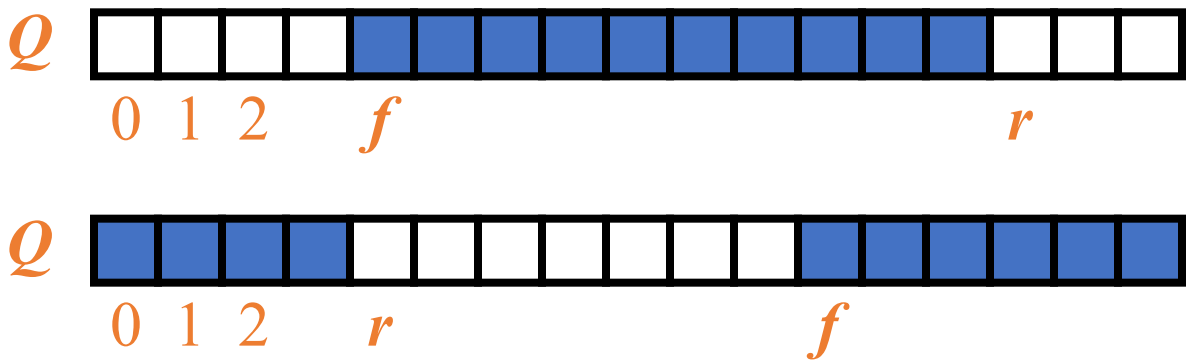
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *FullQueueException*
  else
    Q[*r*] ← *o*
    r ← (*r* + 1) mod *N*



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

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



Queue in Python

- Use the following three instance variables:
  - _data: is a reference to a list instance with a fixed capacity.
  - _size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - _front: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).

Queue in Python, Beginning

```
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]
27
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
```

Queue in Python, Continued

```
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
47
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
```

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)