Data structure and algorithm in Python

Stacks, Queues and Deques

Seongjin Lee

Dept of Aerospace and Software Engineering,
Gyeongsang National University

Table of contents

Stacks

Stacks

Definition

A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.

A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack).

Stacks

Example

Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.

Stacks

Example

Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo oper- ation can be accomplished by keeping text changes in a stack.

Stacks

The Stack Abstract Data Type

The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important.

Formally, a stack is an abstract data type (ADT) such that an instance S supports the following 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; an error occurs if the stack is empty.

The Stack Abstract Data Type

Additionally, define the following accessor methods for convenience:

- `S.top()`: Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.
- `S.is_empty()`: Return True if stack S does not contain any elements.
- `len(S)`: Return the number of elements in stack S; in Python, we implement this with the special method `__len__`.

The Stack Abstract Data Type

Example 6.3: *The following table shows a series of stack operations and their effects on an initially empty stack S of integers.*

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]

Stacks

Simple Array-Based Stack Implementation

Simple Array-Based Stack Implementation

We can implement a stack quite easily by storing its elements in a Python list.

The list class already supports

- adding an element to the end with the `append` method,
- removing the last element with the `pop` method,

so it is natural to align the top of the stack at the end of the list, as shown in

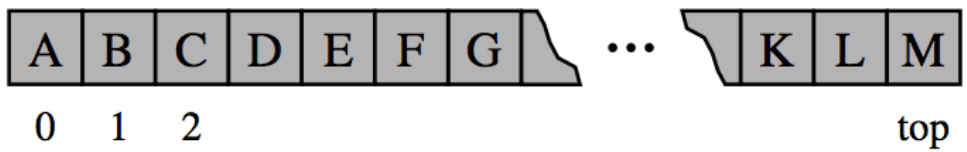


Figure 6.2: Implementing a stack with a Python list, storing the top element in the rightmost cell.

Simple Array-Based Stack Implementation

Although a programmer could directly use the list class in place of a formal stack class,

- lists also include behaviors (e.g., adding or removing elements from arbitrary positions) that would break the abstraction that the stack ADT represents.
- the terminology used by the list class does not precisely align with traditional nomenclature for a stack ADT, in particular the distinction between `append` and `push`.

Simple Array-Based Stack Implementation: The Adapter Pattern

Definition

The adapter design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

Simple Array-Based Stack Implementation: The Adapter Pattern

Definition

The adapter design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable.

Simple Array-Based Stack Implementation: The Adapter Pattern

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)

Table 6.1: Realization of a stack S as an adaptation of a Python list L.

Implementing a Stack Using a Python List

We use the adapter design pattern to define an ArrayStack class that uses an underlying Python list for storage.

Implementing a Stack Using a Python List

One question that remains is what our code should do if a user calls pop or top when the stack is empty. Our ADT suggests that an error occurs, but we must decide what type of error.

```
class Empty(Exception):
    pass
```

Implementing a Stack Using a Python List

```
from exceptions import Empty
class ArrayStack:
    def __init__(self):
        self._data = []

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        return len(self._data) == 0

    def push(self, e):
        self._data.append(e)
```

Implementing a Stack Using a Python List

```
    def pop(self):
        if self.is_empty():
            raise Empty('Stack is empty!')
        return self._data.pop()

    def top(self):
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._data[-1]
```

Implementing a Stack Using a Python List

```
if __name__ == "__main__":
    S = ArrayStack()
    S.push(5)
    S.push(3)
    print(S._data)
    print(S.pop())
    print(S.is_empty())
    print(S.pop())
    print(S.is_empty())
    S.push(7)
    S.push(9)
    S.push(4)
    print(S.pop())
    S.push(6)
    S.push(8)
    print(S._data)
```

Implementing a Stack Using a Python List

```
[5, 3]
3
False
5
True
4
[7, 9, 6, 8]
```

Analyzing the Array-Based Stack Implementation

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

Table 6.2: Performance of our array-based stack implementation. The bounds for push and pop are amortized due to similar bounds for the list class. The space usage is $O(n)$, where n is the current number of elements in the stack.

Analyzing the Array-Based Stack Implementation

The $O(1)$ time for push and pop are amortized bounds.

- A typical call to either of these methods uses constant time;
- But there is occasionally an $O(n)$-time worst case, where n is the current number of elements in the stack, when an operation causes the list to resize its internal array.

Stacks

Application

Reversing Data Using a Stack

As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence.

Example

If the values 1, 2, and 3 are pushed onto a stack in that order, they will be popped from the stack in the order 3, 2, and then 1.

Reversing Data Using a Stack

Example

We might wish to print lines of a file in reverse order in order to display a data set in decreasing order rather than increasing order.

Application

```
from array_stack import ArrayStack
def reverse_file(filename):
    S = ArrayStack()

    original = open(filename)
    for line in original:
        S.push(line.rstrip('\n'))
    original.close

    output = open(filename, 'w')
    while not S.is_empty():
        output.write(S.pop() + '\n')
    output.close()

if __name__ == "__main__":
    reverse_file('111.txt')
```

Application

111.txt

```
eeeeeeee
dddddddd
cccccccc
bbbbbbbb
aaaaaaaa
```

Application

111.txt

```
eeeeeeee
dddddddd
cccccccc
bbbbbbbb
aaaaaaaa
```

```
>>> ./reverse_file.py
```

Application

111.txt

```
eeeeeeee
dddddddd
cccccccc
bbbbbbbb
aaaaaaaa
```

```
>>> ./reverse_file.py
```

111.txt

```
eeeeeeee
dddddddd
cccccccc
bbbbbbbb
aaaaaaaa
```

Matching Parentheses

Consider arithmetic expressions that may contain various pairs of grouping symbols, such as

- Parentheses: '(' and ')'
- Braces: '{'| and lstinline |'}'
- Brackets: `'[' and ']'

Each opening symbol must match its corresponding closing symbol. For example, a left bracket, '[', must match a corresponding right bracket, ']', as in the expression `[(5+x)-(y+z)]`.

The following examples further illustrate this concept:

- Correct: `()(()){([()])}`
- Correct: `(((()(())){([()])}))`
- Incorrect: `)(()){([()])}`
- Incorrect: `({[])}`
- Incorrect: `(`

Matching Parentheses

```
from array_stack import ArrayStack
def is_matched(expr):
    left  = '({['
    right = ')}]'
    S = ArrayStack()
    for c in expr:
        if c in left:
            S.push(c)
        elif c in right:
            if S.is_empty():
                return False
            if right.index(c) != left.index(S.
            pop()):
                return False
    return S.is_empty()
```

Matching Parentheses

```
if __name__ == "__main__":
    expr = '[(5+x) - (y+z)]'
    if is_matched(expr):
        print("In %s: delimiters is matched" %
        expr)
    else:
        print("In %s: delimiters is NOT matched"
         % expr)
```

Matching Tags in HTML

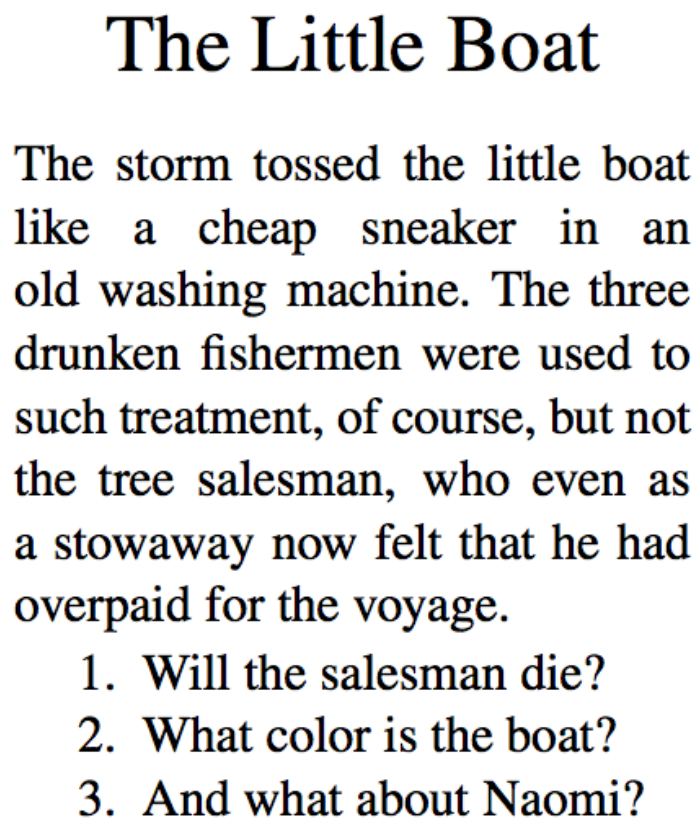
Another application of matching delimiters is in the validation of markup languages such as HTML or XML.

HTML is the standard format for hyperlinked documents on the Internet and XML is an extensible markup language used for a variety of structured data sets.

Matching Tags in HTML

```
<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>
```

(a)



(b)

Figure 6.3: Illustrating HTML tags. (a) An HTML document; (b) its rendering.

Matching Tags in HTML

In an HTML document, portions of text are delimited by HTML tags. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>".

Other commonly used HTML tags that are used in this example include:

- `body`: document body
- `h1`: section header
- `center`: center justify
- `p`: paragraph
- `ol`: numbered (ordered) list
- `li`: list item

Matching Tags in HTML

```
from array_stack import ArrayStack

def is_matched_html(raw):
    S = ArrayStack()
```

Matching Tags in HTML

```
    while j != -1:
        k = raw.find('>', j+1)
        if k == -1:
            return False
        tag = raw[j+1:k]
        if not tag.startswith('/'):
            S.push(tag)
        else:
            if S.is_empty():
                return False
            if tag[1:] != S.pop():
                return False
        j = raw.find('<', k+1)
    return S.is_empty()
```

Matching Tags in HTML

```
if __name__ == "__main__":
    raw = "<a> <b> </b> </a>"
    if(is_matched_html(raw)):
        print("Matched")
    else:
```

Queue

Queue

Definition

A queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle.

Elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

Queue

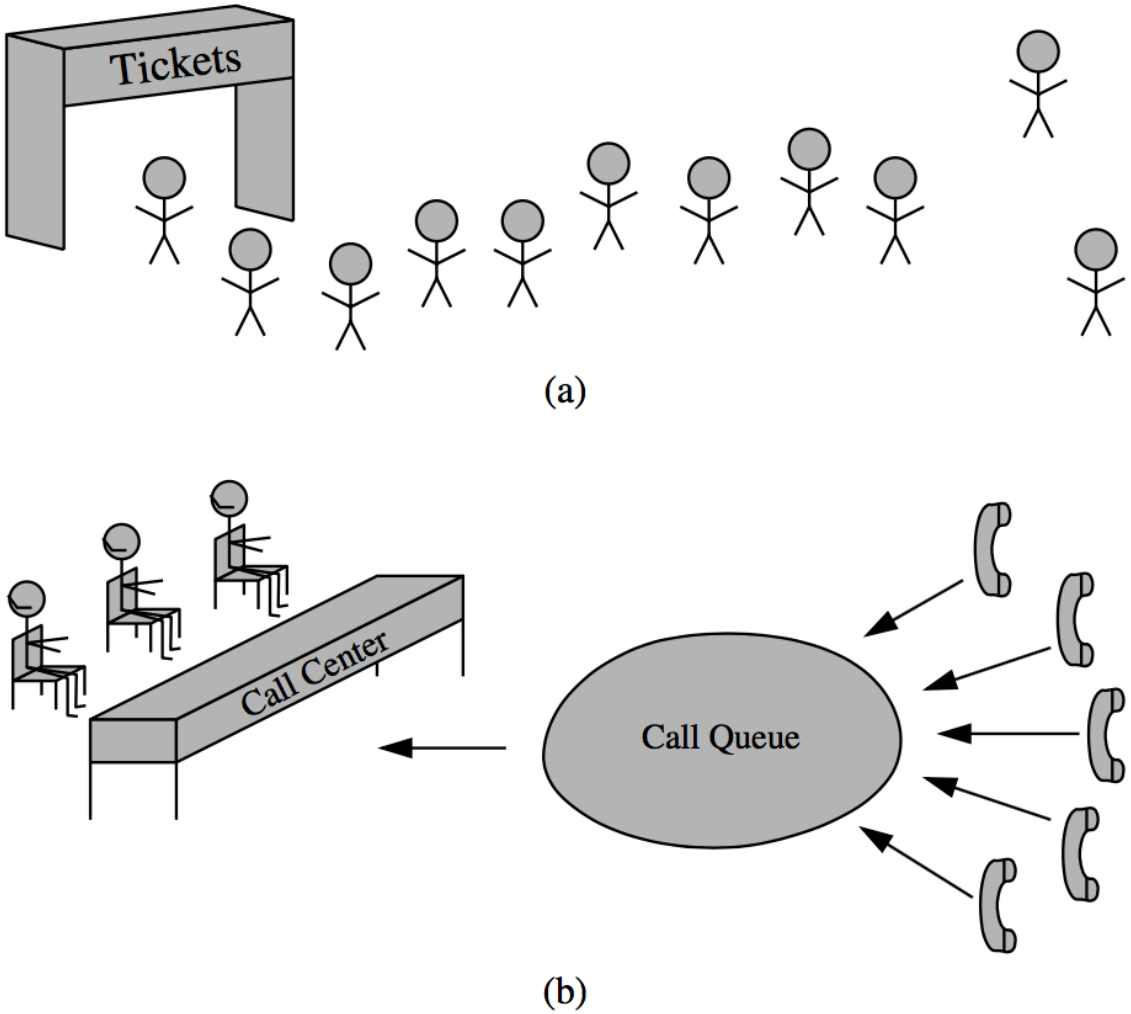


Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

Queue

The Queue Abstract Data Type

The Queue Abstract Data Type

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where

- element access and deletion are restricted to the first element in the queue;
- and element insertion is restricted to the back of the sequence.

The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q:

- `Q.enqueue(e)`: Add element e to the back of queue Q.
- `Q.dequeue(e)`: Remove and return the first element from queue Q; an error occurs if the queue is empty.

The queue ADT also includes the following supporting methods

- `Q.first()`: Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.
- `Q.is_empty()`: Return True if queue Q does not contain any elements.
- `len(Q)`: Return the number of elements in queue Q; in Python, we implement this with the special method `__len__`.

The Queue Abstract Data Type

By convention, we assume that a newly created queue is empty, and that there is no a priori bound on the capacity of the queue. Elements added to the queue can have arbitrary type.

The Queue Abstract Data Type

Example 6.4: *The following table shows a series of queue operations and their effects on an initially empty queue Q of integers.*

Operation	Return Value	first ← Q ← 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]

Queue

Array-Based Queue Implementation

Array-Based Queue Implementation

For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage. It may be very tempting to use a similar approach for supporting the queue ADT.

- We could enqueue element `e` by calling `append(e)` to add it to the end of the list.
- We could use the syntax `pop(0)`, as opposed to `pop()`, to intentionally remove the first element from the list when dequeuing.

Array-Based Queue Implementation

For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage. It may be very tempting to use a similar approach for supporting the queue ADT.

- We could enqueue element `e` by calling `append(e)` to add it to the end of the list.
- We could use the syntax `pop(0)`, as opposed to `pop()`, to intentionally remove the first element from the list when dequeuing.

As easy as this would be to implement, it is tragically inefficient.

Array-Based Queue Implementation

As discussed before, when `pop` is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the pop. Therefore, a call to `pop(0)` always causes the worst-case behavior of $O(n)$ time.

Array-Based Queue Implementation

We can improve on the above strategy by avoiding the call to `pop(0)` entirely.

Array-Based Queue Implementation

We can improve on the above strategy by avoiding the call to `pop(0)` entirely.

We can

- replace the dequeued entry in the array with a reference to None, and
- maintain an explicit variable `f` to store the index of the element that is currently at the front of the queue.

Such an algorithm for dequeue would run in $O(1)$ time.

Matching Tags in HTML

After several dequeue operations, this approach might lead to

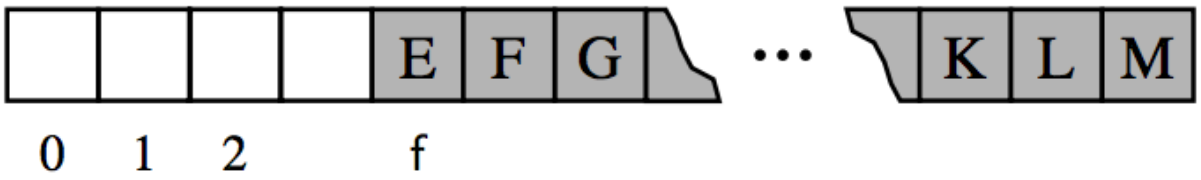


Figure 6.5: Allowing the front of the queue to drift away from index 0.

Matching Tags in HTML

Unfortunately, there remains a drawback to the revised approach.

Matching Tags in HTML

Unfortunately, there remains a drawback to the revised approach.

We can build a queue that has relatively few elements, yet which are stored in an arbitrarily large list. This occurs, for example, if we repeatedly enqueue a new element and then dequeue another (allowing the front to drift rightward). Over time, the size of the underlying list would grow to $O(m)$ where m is the total number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue.

Using an Array Circularly

In developing a more robust queue implementation, we allow

- the front of the queue to drift rightward,
- the contents of the queue to "wrap around" the end of an underlying array.

Using an Array Circularly

We assume that our underlying array has fixed length N that is greater that the actual number of elements in the queue.

Using an Array Circularly

We assume that our underlying array has fixed length N that is greater that the actual number of elements in the queue.

New elements are enqueued toward the "end" of the current queue, progressing from the front to index $N-1$ and continuing at index 0, then 1.

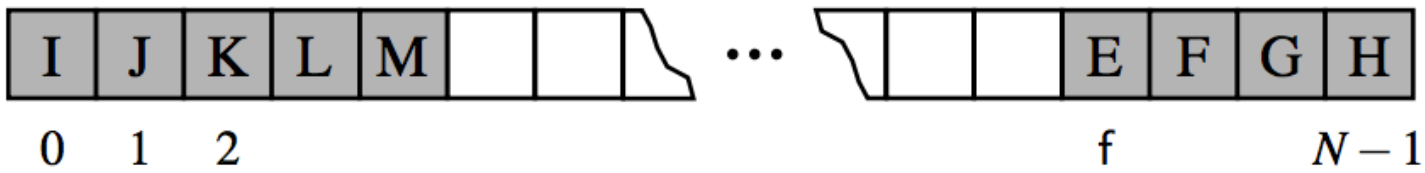


Figure 6.6: Modeling a queue with a circular array that wraps around the end.

Using an Array Circularly

Implementing this circular view is not difficult.

- When we dequeue an element and want to "advance" the front index, we use the arithmetic `f = (f + 1)% N`.

A Python Queue Implementation

Internally, the queue class maintains 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).

A Python Queue Implementation

```
from exceptions import Empty

class ArrayQueue:
    DEFAULT_CAPACITY = 10

    def __init__(self):
        self._data = [None] * ArrayQueue.
        DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0
```

A Python Queue Implementation

```
    def first(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]

    def dequeue(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        result = self._data[self._front]
        self._front = (self._front + 1) % len(
        self._data)
        self._size -= 1
        return result
```

A Python Queue Implementation

```
    def dequeue(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        result = self._data[self._front]
        self._front = (self._front + 1) % len(
        self._data)
        self._size -= 1
        return result
```

Resizing the Queue

When enqueue is called at a time when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list.

Resizing the Queue

After creating a temporary reference to the old list of values, we allocate a new list that is twice the size and copy references from the old list to the new list. While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array, as shown in

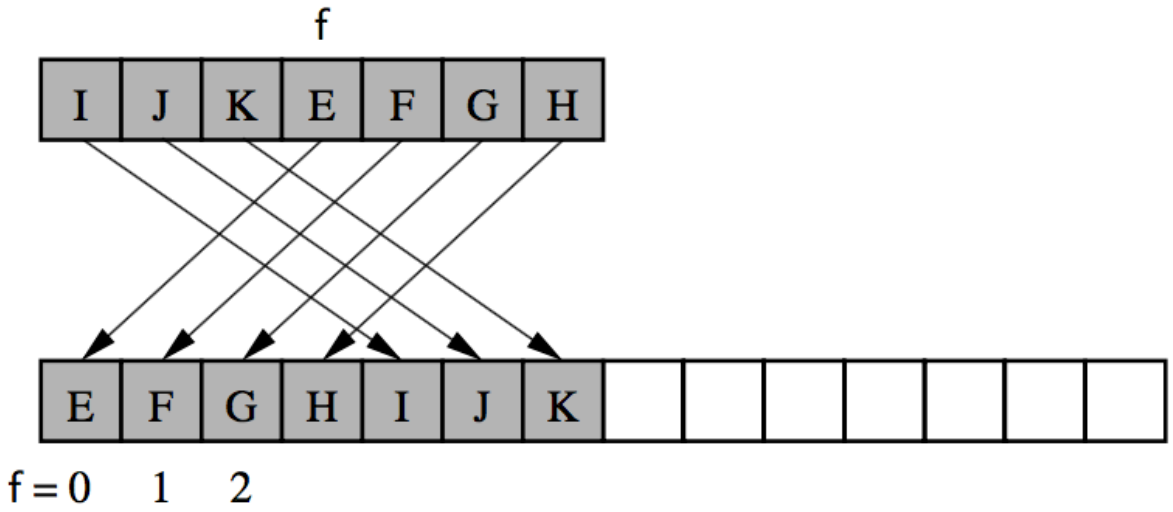


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

Analyzing the Array-Based Queue Implementation

With the exception of the resize utility, all of the methods rely on a constant number of statements involving arithmetic operations, comparisons, and assignments. Therefore, each method runs in worst-case $O(1)$ time, except for enqueue and dequeue, which have amortized bounds of $O(1)$ time

Analyzing the Array-Based Queue Implementation

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

Table 6.3: Performance of an array-based implementation of a queue. The bounds for enqueue and dequeue are amortized due to the resizing of the array. The space usage is $O(n)$, where n is the current number of elements in the queue.

Double-Ended Queues

Double-Ended Queues

Definition

A dequeue (i.e., double-ended queue) is a queue-like data structure that supports insertion and deletion at both the front and the back of the queue.

Deque is usually pronounced "deck" to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation "D.Q.".

Double-Ended Queues

The deque abstract data type is more general than both the stack and the queue ADTs.

Example

A restaurant using a queue to maintain a waitlist.

- Occassionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will re-insert the person at the first position in the queue.
- It may also be that a customer at the end of the queue may grow impatient and leave the restaurant.

Double-Ended Queues

The Deque Abstract Data Type

The Deque Abstract Data Type

To provide a symmetrical abstraction, the deque ADT is defined so that deque D supports the following methods:

- `D.add_first(e)`: Add element e to the front of deque D.
- `D.add_last(e)`: Add element e to the back of deque D.
- `D.delete_first()`: Remove and return the first element from deque D; an error occurs if the deque is empty.
- `D.delete_last()`: Remove and return the last element from deque D; an error occurs if the deque is empty.

The Deque Abstract Data Type

Additionally, the deque ADT will include the following accessors:

- `D.first()`: Return (but do not remove) the first element of deque D; an error occurs if the deque is empty.
- `D.last()`: Return (but do not remove) the last element of deque D; an error occurs if the deque is empty.
- `D.is_empty()`: Return True if deque D does not contain any elements.
- `len(D)`: Return the number of elements in deque D; in Python, we implement this with the special method `__len__`.

The Deque Abstract Data Type

Example 6.5: *The following table shows a series of operations and their effects on an initially empty deque D of integers.*

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

Double-Ended Queues

Implementing a Deque with a Circular Array

Implementing a Deque with a Circular Array

We can implement the deque ADT in much the same way as the ArrayQueue class.

- We recommend maintaining the same three instance variables: `_data`, `_size`, and `_front`.
- Whenever we need to know the index of the back of the deque, or the first available slot beyond the back of the deque, we use modular arithmetic for the computation.
  - in `last()` method, uses the index

```
back = (self._front + self._size - 1) % len(
self._data)
```

  - in `add_first()` method, circularly decrement the index

```
self._front = (self._front - 1) % len(self.
_data)
```

Double-Ended Queues

Deques in the Python Collections Module

Deques in the Python Collections Module

An implementation of a deque class is available in Python's standard collections module. A summary of the most commonly used behaviors of the collections.deque class is given in

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first( )	D.appendleft( )	add to beginning
D.add_last( )	D.append( )	add to end
D.delete_first( )	D.popleft( )	remove from beginning
D.delete_last( )	D.pop( )	remove from end
D.first( )	D[0]	access first element
D.last( )	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear( )	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Table 6.4: Comparison of our deque ADT and the collections.deque class.