武汉大学
WUHAN UNIVERSITY

数据结构与算法

栈，队列与双端队列

张晓平

武汉大学数学与统计学院

Table of contents

栈

栈

定义：栈 (stack)

栈，又名堆栈，是一种运算受限的线性表，即限定仅在表尾进行插入和删除操作的线性表。允许进行插入和删除的一端被称为栈顶，另一端称为栈底。换句话说，栈遵循后进先出 (Last-In First-Out, LIFO) 原则插入和删除元素。

栈

定义：栈 (stack)

栈，又名堆栈，是一种运算受限的线性表，即限定仅在表尾进行插入和删除操作的线性表。允许进行插入和删除的一端被称为栈顶，另一端称为栈底。换句话说，栈遵循后进先出 (Last-In First-Out, LIFO) 原则插入和删除元素。

定义：入栈和出栈

- 向一个栈插入新元素称为入栈 (也称进栈或压栈)，它是把新元素置于栈顶元素之上，使之成为新的栈顶元素；
- 从一个栈中删除元素称为出栈 (也称退栈或弹栈)，它是把栈顶元素删掉，使其相邻元素成为新的栈顶元素。

栈

例

浏览器将最近访问的网址存储在堆栈中。每次用户访问新站点时，该站点的地址都会“压入”地址栈。然后，浏览器允许用户使用"后退" 按钮将该站点地址从地址栈中" 弹出"，回到上一次访问过的网站。

栈

例

浏览器将最近访问的网址存储在堆栈中。每次用户访问新站点时，该站点的地址都会“压入”地址栈。然后，浏览器允许用户使用" 后退" 按钮将该站点地址从地址栈中" 弹出"，回到上一次访问过的网站。

例

文本编辑器通常提供一个" 撤消" 机制，用于取消最近的编辑操作并恢复到文档以前的状态。此撤消操作操作可以通过在栈中保留文本更改来实现。

栈

栈抽象数据类型

栈抽象数据类型

栈是所有数据结构中最简单的，但也是最重要的。

栈抽象数据类型

栈是所有数据结构中最简单的，但也是最重要的。

形式上，栈是一种抽象数据类型 (ADT)，因此其实例 S 支持以下两种方法：

- $S.push(e)$：将元素 e 添加到栈 S 的顶部。
- $S.pop()$：从栈中移除并返回顶部元素；如果栈为空，则报错。

栈抽象数据类型

栈是所有数据结构中最简单的，但也是最重要的。

形式上，栈是一种抽象数据类型 (ADT)，因此其实例 S 支持以下两种方法：

- $S.push(e)$：将元素 e 添加到栈 S 的顶部。
- $S.pop()$：从栈中移除并返回顶部元素；如果栈为空，则报错。

另外还定义了以下访问器：

- $S.top()$：返回 S 栈顶元素的引用，但不删除它；若栈为空，则报错。
- $S.is\_empty()$：若栈 S 不包含任何元素，则返回 *True*.
- $len(S)$：返回栈 S 的元素个数。在 Python 中，通过特殊方法 $\_\_len\_\_()$ 来实现。

栈抽象数据类型

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]

栈

栈的数组实现

栈的数组实现

通过将栈元素存储在 Python 列表中，可以很容易地实现栈。

列表 (list) 类本身支持

- 使用 *append*() 方法在表尾添加元素
- 使用 *pop*() 方法删除最后一个元素

因此，很自然地会想到将栈顶与表尾对齐，正如下图所示：

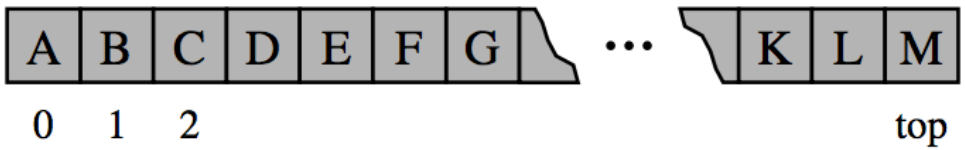


Figure 6.2: Implementing a stack with a Python list, storing the top element in the rightmost cell.

栈的数组实现

尽管可以直接使用列表类代替正式的栈类，但

- 列表包括了一些其他行为 (例如，从任意位置添加或删除元素)，这些行为会破坏堆栈 ADT 所表示的抽象。
- 列表类使用的术语与栈的传统术语不完全一致，特别是 *append* 和 *push* 之间的区别。

栈的数组实现: The Adapter Pattern

定义

很多时候我们希望修改现有类，以使得其方法匹配相关但不同的类，这就是所谓的适配器设计模式 (adapter design pattern)。

栈的数组实现: The Adapter Pattern

定义

很多时候我们希望修改现有类，以使得其方法匹配相关但不同的类，这就是所谓的适配器设计模式 (adapter design pattern)。

适配器模式的通用做法是定义一个新类，它将现有类的实例包含为隐藏域，然后使用此隐藏实例变量的方法来实现新类的每个方法。

栈的数组实现: 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.

用 Python 列表实现栈

- 使用适配器模式来定义ArrayStack类，它使用底层 Python 列表进行存储的。

用 Python 列表实现栈

- 使用适配器模式来定义ArrayStack类，它使用底层 Python 列表进行存储的。
- 当栈为空时，若用户调用 *pop*() 或 *top*()，该怎么办?

用 Python 列表实现栈

- 使用适配器模式来定义ArrayStack类，它使用底层 Python 列表进行存储的。
- 当栈为空时，若用户调用 *pop*() 或 *top*()，该怎么办？建议抛出异常，可自定义如下异常类型：

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

用 Python 列表实现栈

```
from exceptions import Empty
class ArrayStack:
```

用 Python 列表实现栈

```
from exceptions import Empty
class ArrayStack:
```

```
    def __init__(self):
        self._data = []
```

用 Python 列表实现栈

```
from exceptions import Empty
class ArrayStack:
```

```
    def __init__(self):
        self._data = []
```

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

用 Python 列表实现栈

```
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
```

用 Python 列表实现栈

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

用 Python 列表实现栈

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

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

用 Python 列表实现栈

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

```
    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]
```

用 Python 列表实现栈

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

```
    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]
```

```
    def __repr__(self):
        return f'{self._data}'
```

用 Python 列表实现栈

```
if __name__ == "__main__":
    S = ArrayStack()
    S.push(5)
    S.push(3)
    print(S)
    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)
```

用 Python 列表实现栈

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

复杂度分析

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.

栈

应用

使用栈反转数据

由于栈遵循后进先出原则，它可用作反转数据序列的通用工具。

例

将 1、2、3 依次压入栈，就可以从栈中依次弹出 3、2、1，这就实现了数据序列的反转。

使用栈反转数据

由于栈遵循后进先出原则，它可用作反转数据序列的通用工具。

例

将 1、2、3 依次压入栈，就可以从栈中依次弹出 3、2、1，这就实现了数据序列的反转。

```
from array_stack import ArrayStack
def reverse_data(L):
    S = ArrayStack()
    for elem in L:
        S.push(elem)
    L = []
    while not S.is_empty():
        L.append(S.pop())
    return L
```

使用栈反转数据

例

很多时候，经常会以逆序打印文件行，比如我们希望以降序而不是升序显示数据集。

应用

```
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()
```

应用

111.txt

```
aaaaaaaa
bbbbbbbb
cccccccc
dddddddd
eeeeeeee
```

应用

111.txt

```
aaaaaaaa
bbbbbbbb
cccccccc
dddddddd
eeeeeeee
```

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

应用

111.txt

```
aaaaaaaa
bbbbbbbb
cccccccc
dddddddd
eeeeeeee
```

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

111.txt

```
eeeeeeee
dddddddd
cccccccc
bbbbbbbb
aaaaaaaa
```

Matching Parentheses

对于可能包含各种分组符号对 (如圆括号、大括号和方括号) 的算术表达式，每个开始符必须与其对应的结束符相匹配。如在表达式

$$[(5+x)-(y-z)]$$

中，[必须匹配].

例：匹配的例子

- 正确: `()(()){([()])}`
- 正确: `(((()(())){([()])}))`
- 错误: `)(()){([()])}`
- 错误: `({[])}`
- 错误: `(`

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(f"In {expr}: delimiters is matched
        ")
    else:
        print(f"In {expr}: delimiters is NOT
        matched")
```

Matching Parentheses

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

```
In [(5 + x) - (y + z)]: delimiters is matched
```

HTML 中的标记匹配

匹配分隔符的另一个应用是验证标记语言，如 HTML 或 XML。

HTML 是 Internet 上超链接文档的标准格式，XML 是用于各种结构化数据集的可扩展标记语言。

HTML 中的标记匹配

匹配分隔符的另一个应用是验证标记语言，如 HTML 或 XML。

HTML 是 Internet 上超链接文档的标准格式，XML 是用于各种结构化数据集的可扩展标记语言。

在 HTML 文档中，部分文本由 HTML 标记分隔。一个简单的开始标记为<name>，相应的结尾标记为</name>。

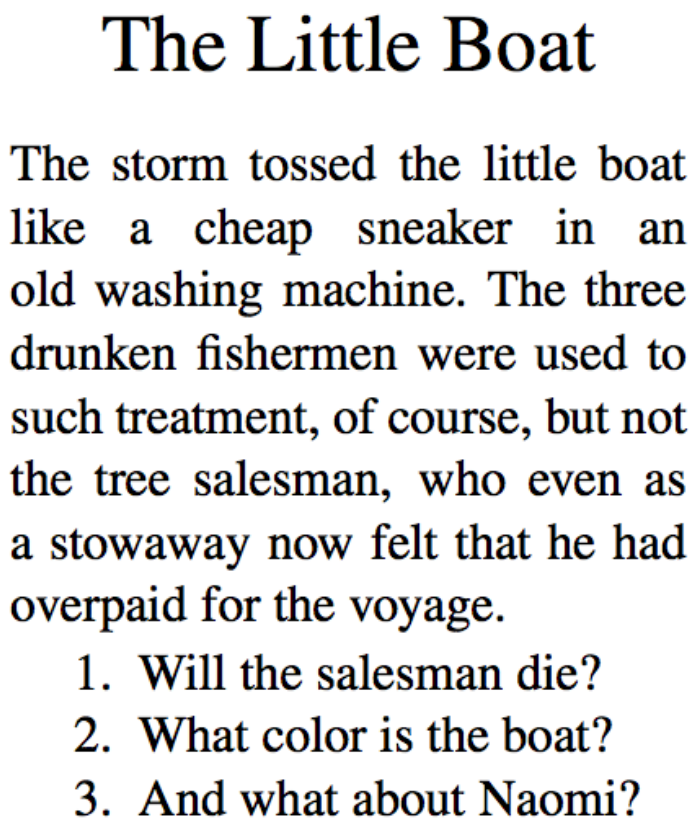
常用 HTML 标记有

body	文件正文
h1	节标题
center	居中对齐
p	段落
ol	编号 (排序) 列表
li	列表项

HTML 中的标记匹配

```
<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.

HTML 中的标记匹配

```
from array_stack import ArrayStack
def is_matched_html(raw):
    S = ArrayStack()
    j = raw.find('<')
    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()
```

HTML 中的标记匹配

'example.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 sunc treatment, of course, but not the tree
     sales man, who even as a stowaway now felt that he had
    overpaid for the voyage.
  </p>
  <ol>
    <li> Will the salesman die? </li>
    <li> What color is the boat? </li>
    <li> And what about Naomi? </li>
  </ol>
</body>
```

HTML 中的标记匹配

```
if __name__ == "__main__":
    fname = 'example.html'
    raw = open(fname)
    if(is_matched_html(raw.read())):
        print(f"In {fname}: Matched")
    else:
        print(f"In {fname}: Not Matched")
```

HTML 中的标记匹配

```
if __name__ == "__main__":
    fname = 'example.html'
    raw = open(fname)
    if(is_matched_html(raw.read())):
        print(f"In {fname}: Matched")
    else:
        print(f"In {fname}: Not Matched")
```

```
In example.html: Matched
```

队列

队列

定义：队列 (queue)

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端 (front) 进行删除操作，而在表的后端 (rear) 进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。

队列

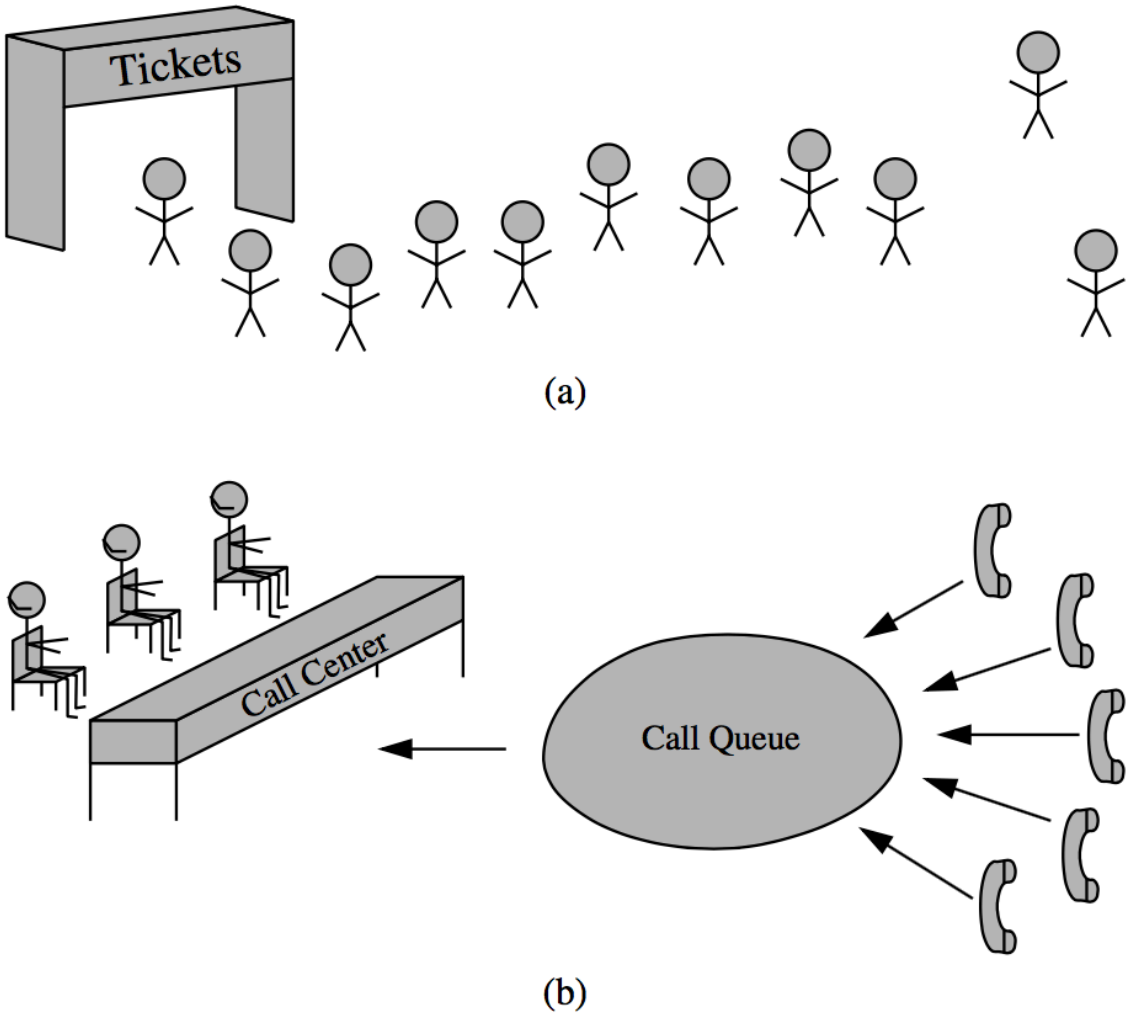


Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

队列

队列抽象数据类型

队列抽象数据类型

在形式上，队列抽象数据类型定义了一个集合，该集合将对象保持在一个序列中，其中

- 访问和删除操作仅限于队头元素；
- 插入操作仅限于队尾。

队列抽象数据类型

队列抽象数据类型（ADT）支持队列 Q 的以下两种基本方法：

- $Q.enqueue(e)$：　将元素 e 添加到队列 q 的后面；
- $Q.dequeue(e)$：　从队列 Q 中移除并返回第一个元素；如果队列为空，则会报错。

队列 ADT 还包括以下支持方法：

- $Q.first()$：　返回对队头元素的引用，而不删除它；如果队列为空，则会报错。
- $Q.is\_empty()$：　如果队列 Q 不包含任何元素，则返回 $True$。
- $len(Q)$：　返回队列 Q 中的元素数；在 Python 中，使用特殊的方法 $\_\_len\_\_$ 来实现。

队列抽象数据类型

按照惯例，我们假设

- 初始队列为空
- 队列的容量没有预先限制
- 添加到队列的元素可以具有任意类型

队列抽象数据类型

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]

队列

基于数组的队列实现

基于数组的队列实现

对于堆栈 ADT，创建了一个非常简单的适配器类，它使用 Python 列表作为底层存储。使用类似的方法也可支持队列 ADT。

- 入队列时，使用 $append(e)$ 将元素 e 添加到列表的末尾；
- 出队列时，使用 $pop(0)$ 删除列表的第一个元素。

基于数组的队列实现

对于堆栈 ADT，创建了一个非常简单的适配器类，它使用 Python 列表作为底层存储。使用类似的方法也可支持队列 ADT。

- 入队列时，使用 $append(e)$ 将元素 e 添加到列表的末尾；
- 出队列时，使用 $pop(0)$ 删除列表的第一个元素。

尽管这很容易实现，但它的效率却很低，因调用 $pop(0)$ 将导致 $O(n)$ 的时间复杂度。

基于数组的队列实现

问题

如何避免调用 $pop(0)$ 来改进上述策略呢?

基于数组的队列实现

问题

如何避免调用 $pop(0)$ 来改进上述策略呢?

解决方案

- 将数组中的出列项替换为对 *None* 的引用
- 维护一个显式变量 f 以存储当前队头元素的索引

这种出队列算法的时间复杂度为 $O(1)$.

HTML 中的标记匹配

在多次出列操作之后，这种方法可能会导致

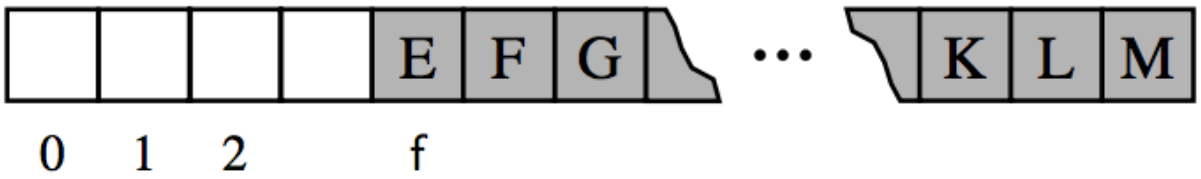


Figure 6.5: Allowing the front of the queue to drift away from index 0.

HTML 中的标记匹配

在多次出列操作之后，这种方法可能会导致

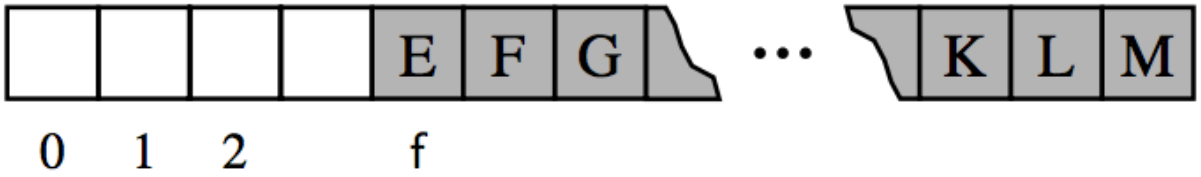


Figure 6.5: Allowing the front of the queue to drift away from index 0.

不幸的是，这种改进策略仍有缺陷。

HTML 中的标记匹配

在多次出列操作之后，这种方法可能会导致

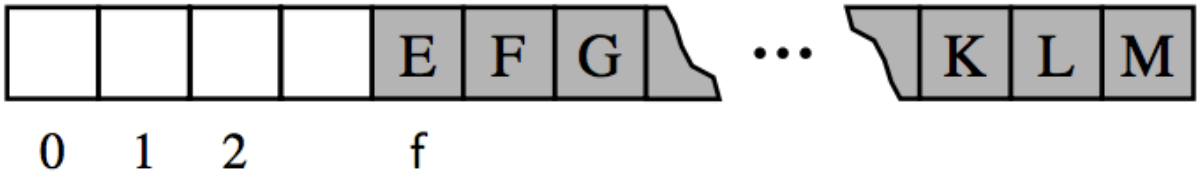


Figure 6.5: Allowing the front of the queue to drift away from index 0.

不幸的是，这种改进策略仍有缺陷。

若重复地入队列和出队列，就会导致该队列中的元素个数较少，而底层列表却很大的情形。随着时间的推移，底层列表的大小将增至 $O(m)$，其中 m 是入队列操作的总数，而不是队列中当前元素的个数。

循环使用数组

为开发更健壮的队列实现，可允许队头元素向右移动，使队列内容“环绕”至底层数组的末尾。

循环使用数组

为开发更健壮的队列实现，可允许队头元素向右移动，使队列内容“环绕”至底层数组的末尾。

设底层数组具有固定长度 n，该长度大于队列中的实际元素个数。

循环使用数组

为开发更健壮的队列实现，可允许队头元素向右移动，使队列内容“环绕”至底层数组的末尾。

设底层数组具有固定长度 n，该长度大于队列中的实际元素个数。

新元素在当前队列的“末尾”处入队列，从前面进入索引 $n-1$，并在索引 0 处继续，然后是 1。

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

循环使用数组

实现循环队列并不困难。设 f 为当前队头元素的索引，

- 出队列一个元素后，队头元素的索引变为

$$f = (f+1)\%N.$$

- 入队列时，若队列中的元素个数等于底层列表的长度，将对底层列表加倍扩容。在复制内容时，会将队头元素调整至索引为 0 的位置。

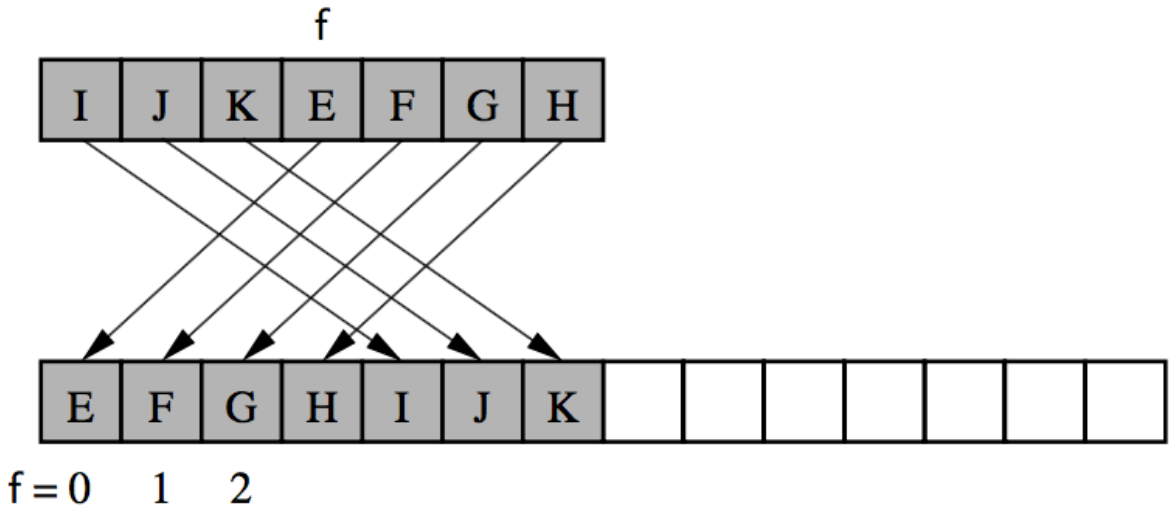


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

循环队列的实现

在内部，队列类维护以下三个实例变量：

- *_data*：对具有固定容量的列表实例的引用。
- *_size*：一个整数，表示队列中存储的当前元素数 (与数据列表的长度相反)。
- *_front*：是一个整数，表示队列第一个元素的数据中的索引（假设队列不是空的）。

循环队列的实现

```
from exceptions import Empty
class ArrayQueue:
    DEFAULT_CAPACITY = 5
```

循环队列的实现

```
from exceptions import Empty
class ArrayQueue:
    DEFAULT_CAPACITY = 5
```

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

循环队列的实现

```
from exceptions import Empty
class ArrayQueue:
    DEFAULT_CAPACITY = 5
```

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

```
    def __len__(self):
        return self._size
```

循环队列的实现

```
from exceptions import Empty
class ArrayQueue:
    DEFAULT_CAPACITY = 5
```

```
    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
```

循环队列的实现

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

循环队列的实现

```
    def dequeue(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        result = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(
        self._data)
        self._size -= 1
        return result
```

循环队列的实现

```
    def dequeue(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        result = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(
        self._data)
        self._size -= 1
        return result
```

```
    def enqueue(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        avail = (self._front + self._size) % len
        (self._data)
        self._data[avail] = e
        self._size += 1
```

循环队列的实现

```
    def _resize(self, cap):
        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0
```

循环队列的实现

```
    def _resize(self, cap):
        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0
```

```
    def __repr__(self):
        data = []
        walk = self._front
        for k in range(self._size):
            data.append(self._data[walk])
            walk = (1 + walk) % len(self._data)
        return f"{data}"
```

循环队列的实现

```
if __name__ == '__main__':
    Q = ArrayQueue()
    Q.enqueue(1)
    Q.enqueue(2)
    Q.enqueue(3)
    Q.enqueue(4)
    Q.enqueue(5)
    Q.dequeue()
    Q.dequeue()
    print(Q._data)
    Q.enqueue(6)
    Q.enqueue(7)
    print(Q._data)
    Q.enqueue(8)
    print(Q._data)
```

循环队列的实现

```
[None, None, 3, 4, 5]
[6, 7, 3, 4, 5]
[3, 4, 5, 6, 7, 8, None, None, None, None]
```

复杂度分析

除了 *resize*() 之外，所有方法都仅依赖于算术运算、比较和赋值等常量操作。因此，每个方法的最坏复杂度均为为 $O(1)$。

复杂度分析

除了 *resize*() 之外，所有方法都仅依赖于算术运算、比较和赋值等常量操作。因此，每个方法的最坏复杂度均为为 $O(1)$。

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.

双端队列

双端队列

定义：双端队列 (Double-Ended Queues, Deque)

双端队列是一种类似于队列的数据结构，允许在队列的两端进行插入和删除操作。

双端队列

定义：双端队列 (Double-Ended Queues, Deque)

双端队列是一种类似于队列的数据结构，允许在队列的两端进行插入和删除操作。

双端队列比栈和队列更通用。

例：餐厅的等待列表

- 有时，第一位顾客从队列中删除，但却发现没有空位；通常，餐厅会在队头处重新插入该顾客。
- 排在队尾的顾客可能会变得不耐烦，离开餐馆。

双端队列

双端队列 ADT

双端队列 ADT

为了提供对称的抽象，定义双端队列 ADT，以使得双端队列 D 支持以下方法：

- $D.add\_first(e)$　将元素 e 添加到 D 的前面。
- $D.add\_last(e)$　将元素 e 添加到 D 的后面。
- $D.delete\_first()$　从 D 中移除并返回第一个元素；如果 D 为空，则报错。
- $D.delete\_last()$　从 D 中移除并返回最后一个元素；如果 D 为空，则报错。

双端队列 ADT

此外，双端队列 ADT 将包括以下访问器：

- $D.first()$　返回 (但不要删除)D 的第一个元素；如果 D 为空，则报错。
- $D.last()$　返回 (但不要删除)D 的最后一个元素；如果 D 为空，则报错。
- $D.is\_empty()$　若 D 不包含任何元素，则返回 $True$。
- $len(D)$　返回 D 中的元素个数；在 Python 中，使用特殊方法 $\_\_len\_\_$ 来实现。

双端队列 ADT

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]

双端队列

用循环数组实现双端队列

用循环数组实现双端队列

我们可以以与 ArrayQueue 类几乎相同的方式实现队列 ADT。

- 建议维护相同的三个实例变量：*_data*, *_size*, *_front*。
- 当我们需要知道双端队列后面的索引或双端队列后面第一个可用的槽位时，使用模运算进行计算。
  - 在 *last*() 中，使用索引

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

  - 在 *add_first*() 中，循环递减索引

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

用循环数组实现双端队列

```
from exceptions import Empty
class ArrayDeque:
    DEFAULT_CAPACITY = 5
```

用循环数组实现双端队列

```
from exceptions import Empty
class ArrayDeque:
    DEFAULT_CAPACITY = 5
```

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

用循环数组实现双端队列

```
from exceptions import Empty
class ArrayDeque:
    DEFAULT_CAPACITY = 5
```

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

```
    def __len__(self):
        return self._size
```

用循环数组实现双端队列

```
from exceptions import Empty
class ArrayDeque:
    DEFAULT_CAPACITY = 5
```

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

```
    def __len__(self):
        return self._size
```

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

用循环数组实现双端队列

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

用循环数组实现双端队列

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

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

用循环数组实现双端队列

```
    def delelte_first(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        result = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(
        self._data)
        self._size -= 1
        return result
```

用循环数组实现双端队列

```
    def delelte_last(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        back = (self._front + self._size - 1) %
        len(self._data)
        result = self._data[back]
        self._data[back] = None
        self._size -= 1
        return result
```

用循环数组实现双端队列

```
    def add_last(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        avail = (self._front + self._size) % len
        (self._data)
        self._data[avail] = e
        self._size += 1
```

用循环数组实现双端队列

```
    def add_last(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        avail = (self._front + self._size) % len
        (self._data)
        self._data[avail] = e
        self._size += 1
```

```
    def add_first(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        self._front = (self._front - 1) % len(
        self._data)
        self._data[self._front] = e
        self._size += 1
```

用循环数组实现双端队列

```
    def _resize(self, cap):
        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0
```

用循环数组实现双端队列

```
    def _resize(self, cap):
        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0
```

```
    def __repr__(self):
        data = []
        walk = self._front
        for k in range(self._size):
            data.append(self._data[walk])
            walk = (1 + walk) % len(self._data)
        return f"{data}"
```

用循环数组实现双端队列

```
if __name__ == '__main__':
    D = ArrayDeque()
    D.add_last(5)
    D.add_first(3)
    D.add_first(2)
    D.add_first(1)
    D.add_last(4)
    print(D._data)

    D.add_last(6)
    print(D._data)

    D.delelte_first()
    D.delelte_first()
    D.delelte_last()
    print(D._data)
```

用循环数组实现双端队列

```
[5, 4, 1, 2, 3]
[1, 2, 3, 5, 4, 6, None, None, None, None]
[None, None, 3, 5, 4, None, None, None, None,
None]
```

双端队列

Python 的 collections 模块中的双端队列

Python 的 collections 模块中的双端队列

在 Python 的标准集合模块中提供了一个 *deque* 类的实现。*collections.deque* 类的最常用行为总结如下

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.