Stacks

1

Stacks

- A stack is a data structure of items such that items **can be inserted and removed only at one end.**
- Stacks allows **access to only one item at a time**.
- It is a LIFO data structure.
- Extremely useful tool in programming:
  - Evaluate arithmetic expressions with lots of parentheses
  - Traversing binary trees
  - Searching vertices of a graph, and much more.
- There are two ways we can implement a stack:
  - Using an array/list. Check difference between array and list here.
  - Using a linked list
- We will look at array implementation of stack only.

2

Stacks Basic Operations

- Push - place an item on the stack
- Peek / Top - Look at the item on top of the stack, but do not remove it
- Pop - Look at the item on top of the stack and remove it

3

Stack Abstract Data Type (ADT)

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.

Additionally, let us 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__.

4

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For both underflow and overflow errors, exceptions should be thrown with an appropriate message

5

5

Error Checking

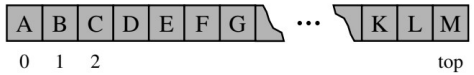
- Always check to see if there are items on the stack before popping the stack or if there is room before pushing the stack
  - isFull?
  - isEmpty?
  - Appropriate error routines are up to the developer / user
- Best approach: in Push() and Pop(), check for isEmpty and isFull within these methods.
- Regardless, they need to be specified.

6

6

Implementing a Stack Using a Python List

- 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, and 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 below



7

7

Realization of a Stack Using a Python List

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)

8

8

Implementing a Stack Using a Python List

```
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
```

9

9

Implementing a Stack Using a Python 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
```

10

10

Example Usage

Below, we present an example of the use of our ArrayStack class, mirroring the operations at the beginning of Example 6.3 on page 230.

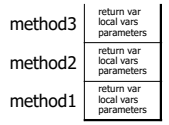
```
S = ArrayStack( )              # contents: [ ]
S.push(5)                      # contents: [5]
S.push(3)                      # contents: [5, 3]
print(len(S))                  # contents: [5, 3];      outputs 2
print(S.pop( ))                # contents: [5];         outputs 3
print(S.is_empty( ))           # 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]
```

11

11

Stack Applications

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome
- Undo and redo operations in text editors.



12

12

Queues

13

Introduction

- A queue is FIFO data structure
- A stack, on the other hand, is a LIFO data structure.
- A queue can be implemented using an array or a linked list
- We are going to look at array implementation of queue data structure
- Queues: Very useful for many different applications:
  - Typically used to model waiting lines, such as planes waiting to take off; waiting lines at banks, hospitals etc.
  - Many queues in your operating system:
    - Print queues
    - Router packet queues

14

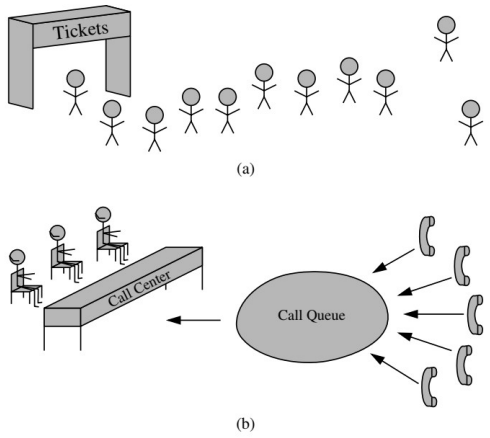


Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

15

Queues Terminology

- For queues:
  - data entered in one end and
  - data extracted from the other end
- Rear of queue is the tail (back or tail or end)
- Front of queue is head.
- Thus, queue operations are insert() and remove():
  - Insert() at the rear (back or tail) of the queue
  - Remove() at the front (head) of the queue.

16

Queue Abstract Data Type (ADT)

Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue( ): 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 (with first being analogous to the stack's top method):

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__`.

17

17

Example

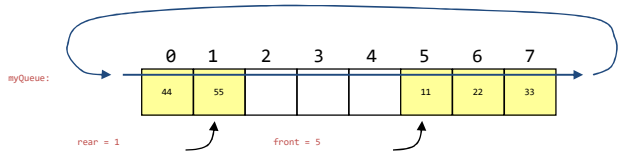
Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3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

18

18

Circular arrays

- We can treat the array holding the queue elements as a circular (joined at the ends)



- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order

19

19

Implementing a Queue Using Circular Array

In developing a more robust queue implementation, we allow the front of the queue to drift rightward, and we allow the contents of the queue to "wrap around" the end of an underlying array. 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 illustrates such a queue with first element `E` and last element `M`.

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

20

20

Implementing a Queue Using Circular Array

Implementing this circular view is not difficult. When we dequeue an element and want to "advance" the front index, we use the arithmetic $f = (f + 1) \% N$. Recall that the % operator in Python denotes the ***modulo*** operator, which is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is, $\frac{14}{3} = 4\frac{2}{3}$. So in Python, 14 // 3 evaluates to the quotient 4, while 14 % 3 evaluates to the remainder 2. The modulo operator is ideal for treating an array circularly. As a concrete example, if we have a list of length 10, and a front index 7, we can advance the front by formally computing $(7+1) \% 10$, which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute $(9+1) \% 10$, which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).

21

21

Queues Potential Errors

- Attempt to remove an item from an empty queue?
  - Want 'empty queue' message returned.
- Attempt to insert an item into a 'full queue'
  - Want full queue message returned.
  - Or increase size of array
- Must be careful to keep track of the front and rear of the queue so that they do not wrap!

22

22

Pyhton Queue Implementation

- A complete implementation of a queue ADT using a Python list in circular fashion is presented in Code Fragments 6.6 and 6.7. 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).
- We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the front index to zero.
- When front or dequeue are called with no elements in the queue, we raise an instance of the Empty exception, defined in Code Fragment 6.1 for our stack.

23

23

Python Queue Implementation

```
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
```

24

24

Python Queue Implementation

```
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
```

25

25

Python Queue Implementation

```
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
```

26

26

Python Queue Implementation

```
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
```

27

27

Application of Queues

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

28

28