Lecture 10 Stack and Queue

§1 Stack

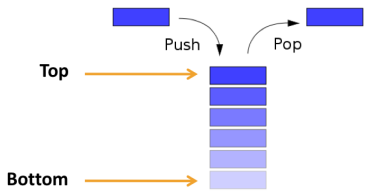
1. Basic information

1° A basic data structure

2° A linear structure

3° The last-in, first-out (LIFO) Principle

Insertion and deletion of items takes place at one end: top of the stack



EXAMPLE: WEB BROWSER

- 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.
- Using the "back" button: Back to previously visited sites
  - The browser then allows the user to "pop"

EXAMPLE: TEXT EDITOR

- Undo Operation
  - cancels recent editing operations
  - reverts to former states of a document
  - can be accomplished by keeping text changes in a stack
- How about Redo?

2. Operators for stacks

1° Push: Insert a data item to the top of the stack

2° Pop: Get a data item on the top of the stack and remove it from the stack.

3° Top: Get a data item on the top of the stack and do not remove it from the stack

3. The stack class

- Generally, a stack may contain the following 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.

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 CODE OF STACK CLASS

```
class ListStack:

    def __init__(self):
        self.__data = list()

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

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

    def push(self,e):
        self.__data.append(e)

    def top(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data[self.__len__()-1]

    def pop(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data.pop()
```

THE CODE TO USE STACK CLASS

```
def main():
    s = ListStack()
    print('The stack is empty? ',s.is_empty())
    s.push(100)
    s.push(200)
    s.push(300)
    print(s.top())
    print(s.pop())
    print(s.top())
```

300
300
200

PRACTICE: REVERSE A LIST USING STACK

- Write a program to reverse the order of a list of numbers using the stack class

SOLUTION

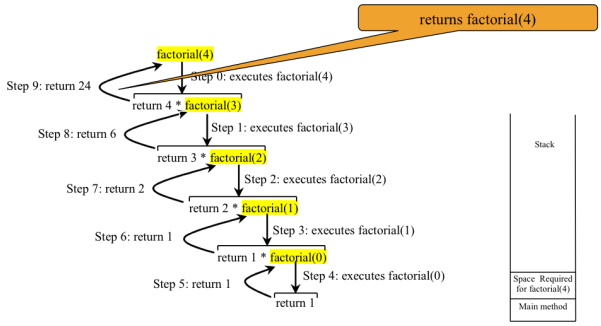
```
from stack import ListStack

def reverse_data(oldList):
    s = ListStack()
    newList = list()

    for i in oldList:
        s.push(i)

    while (not s.is_empty()):
        mid = s.pop()
        newList.append(mid)

    return newList

def main():
    oldList = [1,2,3,4,5]
    newList = reverse_data(oldList)
    print(newList)
```

STACK IN RECURSIONS

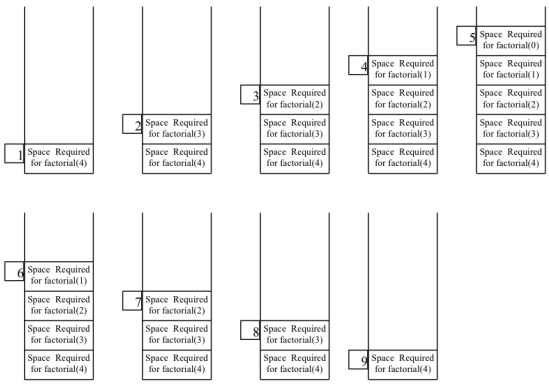
factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * (6)
= 24

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

TRACE OF RECURSIVE FACTORIAL



TRACE OF RECURSIVE FACTORIAL



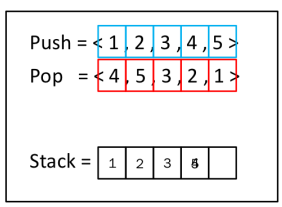
SOLVE PROBLEM WITH STACK: POP SEQUENCE

Given two integer lists, one of which is a sequence of numbers pushed into a stack (supposing all numbers are unique). Please write a program to check whether the other list is a corresponding sequence popped from the stack.

For example, if the pushing sequence is {1,2,3,4,5}, the sequence {4,5,3,2,1} is a corresponding sequence, but {4,3,5,1,2} is not.

SOLUTION IDEA

- Push the elements from push sequence to a stack one by one
- After pushing each element, peek if it is the head of pop sequence
- If the top of stack equals to top of head of pop sequence, keep poping stack elements and moving cursor of pop forward
- Once finishing push sequence, check if stack is empty
- If empty, return true; else, return false



CASE STUDY WITH STACK: CALCULATOR

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open (and closing parentheses), the plus + , minus -, multiply *, divide / , non-negative single digit integers and empty spaces.

You may assume that the given expression is always valid.
Examples:
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23

Note: Do not use the eval built-in library function.

ALGORITHM

Phase 1: Scanning the expression
The program scans the expression from left to right to extract operands, operators, and the parentheses.
1.1. If the extracted item is an operand, push it to **operandStack**.
1.2. If the extracted item is a + or - operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.
1.3. If the extracted item is a * or / operator, process the * or / operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.
1.4. If the extracted item is a (symbol, push it to **operatorStack**.
1.5. If the extracted item is a) symbol, repeatedly process the operators from the top of **operatorStack** until seeing the (symbol on the stack.

Phase 2: Clearing the stack
Repeatedly process the operators from the top of operatorStack until operatorStack is empty.

EXAMPLE

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3 ↑	(	Phase 1.4		(
(1 + 2)*4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2)*4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3 ↑	)	Phase 1.5	3	
(1 + 2)*4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2)*4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3 ↑	−	Phase 1.2	12	−
(1 + 2)*4 − 3 ↑	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3 ↑	none	Phase 2	9	

§2 Queue

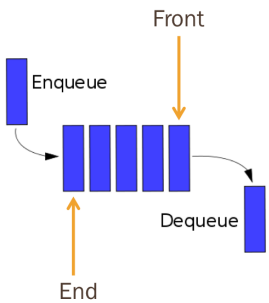
1. Basic information

1° Another basic data structure

2° A linear structure

3° The first-in, first-out (FIFO) priciple

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



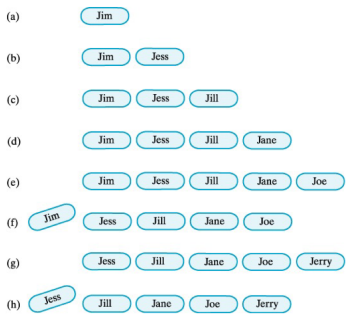
2. Operators for queues

1° Enqueue: insert a data item to the rear of the queue.

2° Dequeue: remove a data item at the front of the queue.

3° First: get a data item at the front of the queue without removing.

A SIMPLE EXAMPLE



Queue of strings after
(a) **enqueue** adds *Jim*;
(b) *Jess*;
(c) *Jill*;
(d) *Jane*;
(e) *Joe*;
(f) **dequeue** retrieves, removes *Jim*;
(g) **enqueue** adds *Jerry*;
(h) **dequeue** retrieves, removes *Jess*.

3. The queue class

- The queue class may contain the following methods:

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.

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 CODE OF QUEUE CLASS

```
class ListQueue:
    default_capacity = 5

    def __init__(self):
        self.__data = [None]*ListQueue.default_capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size ==0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]
```

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

```
def dequeue(self):

    if self.is_empty():
        print('Queue is empty.')
        return None

    answer = self.__data[self.__front]
    self.__data[self.__front] = None
    self.__front = (self.__front+1) \
                   % ListQueue.default_capacity
    self.__size -=1
    return answer

def enqueue(self,e):
    if self.__size == ListQueue.default_capacity:
        print('The queue is full.')
        return None

    self.__data[self.__end] = e
    self.__end = (self.__end+1) \
                 % ListQueue.default_capacity
    self.__size += 1

def outputQ(self):
    print(self.__data)
```

PRACTICE: SIMULATING A WEB SERVICE

- An online video website handles service requests in the following way:
  1) It maintains a service queue which stores all the unprocessed service requests.
  2) When a new service request arrives, it will be saved at the end of the service queue.
  3) The server of the website will process each service request on a "first-come-first-serve" basis.

- Write a program to simulate this process. The processing time of each service request should be randomly generated.

SOLUTION

```
from ListQueue import ListQueue
from random import random
from math import floor

class WebService():
    default_capacity = 5
    def __init__(self):
        self.nameQ = ListQueue()
        self.timeQ = ListQueue()

    def taskArrive(self,taskName,taskTime):
        if self.nameQ.__len__() < WebService.default_capacity:
            self.nameQ.enqueue(taskName)
            self.timeQ.enqueue(taskTime)
            print('A new task 《'+taskName+'》 has arrived and is waiting for processing...')
        else:
            print('The service queue of our website is full, the new task is dropped.')

    def taskProcess(self):
        if (self.nameQ.is_empty() == False):
            taskName = self.nameQ.dequeue()
            taskTime = self.timeQ.dequeue()
            print('Task 《'+taskName+'》 has been processed, it costs '+str(taskTime)+' seconds.')
```

```
def main():
    ws = WebService()
    taskNameList = ['Dark knight','X-man','Kungfu','Shaolin Soccer','Matrix','Walking in the clouds'\
                    ,'Casino Royale','Bourne Supremacy','Inception','The Shawshank Redemption']

    print('Simulation starts...')
    print('----------------------------------------')
    for i in range(1,31):
        rNum = random()
        if rNum<=0.6:
            taskIndex = floor(random()*10)
            taskTime  = floor(random()*1000)/100
            ws.taskArrive(taskNameList[taskIndex],taskTime)
        else:
            ws.taskProcess()
    print('----------------------------------------')
    print('Simulation finished.')
```

4. Types of queuing system

1° Single queue, single server

2° Single queue, multiple servers

e.g. Bank

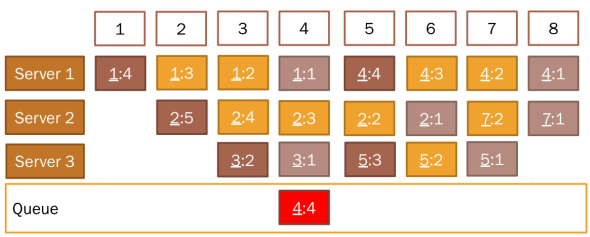
3° Multiple queus, single servers

e.g. Traffic light

4° Multiple queues, multiple servers

SOLVE PROBLEM WITH QUEUE: PROCESSING TIME

- Given a single queue for n guests and 3 bank tellers. The *i* th guests will arrived at k(i) th minutes and their transactions will be completed in t(i) minutes. The bank tellers will try their best to serve guests if the queue is not empty. Calculate the overall processing time.
- For example, given 6 guests and corresponding k = [1, 2, 3, 4, 5, 7] and t = [4, 5, 2, 4, 3, 2], the overall processing time is 8 minutes, calculated as follows:



SOLUTION IDEA

- Initialize a clock with minute = 1
- Create an empty queue
- Create tellers for no workload
- Run the clock to simulate the status of each minute
  - If comes a new guest, put them into the queue
  - For each teller
    - If it is serving, then reduce the transaction minute by 1
    - If it it not serving and the queue is not empty, get a guest to serve
  - if tellers are free and the queue is empty, stop simulation