11-stack-and-queue

November 9, 2021

1 Stack and Queue

1.1 Agenda

- Overview

1. Stacks
   - … for delimiter pairing
   - … for postfix expression evaluation
   - … for tracking execution and *backtracking*
2. Queues
   - … for tracking execution and *backtracking*
   - … for fair scheduling (aka "round-robin" scheduling)
   - … for doling out work
3. Run-time analysis

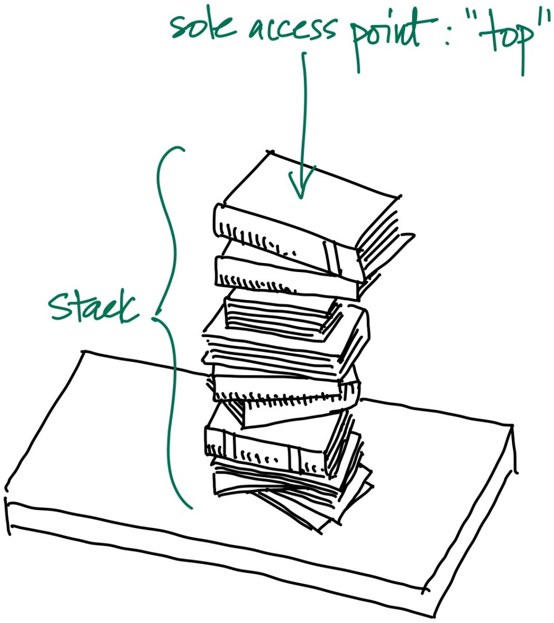
1.2 Overview

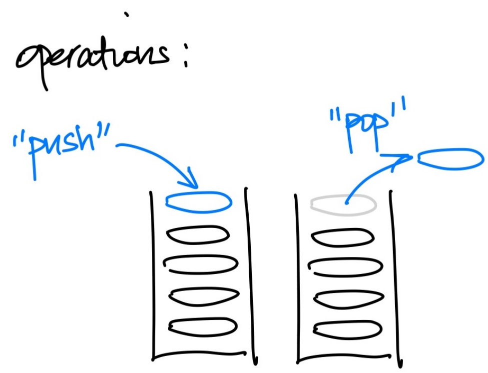
While the list ADT is incredibly useful, both styles of implementation we explored (array-backed and linked) have operations that run in $O(N)$ time, which give them unpredictable runtime behavior.

By further restricting the list API, however — in particular, by *isolating points of access to either the front or end of the underlying data* — we can create data structures whose operations are uniformly $O(1)$, and remain very useful in their own right.

1.3 1. Stacks

The **stack** is an ADT which only permit access to one "end" of the data collection. We can only append ("push") items onto the tail end (a.k.a. the "top") of a stack, and only the most recently added item can be removed ("popped"). The last item to be pushed onto a stack is therefore the first one to be popped off, which is why we refer to stacks as last-in, first out (LIFO) structures.





1.3.1 Array-backed Stack

[1]:
```python
# array-backed implementation

class Stack:
    def __init__(self):
        self.data = []

    def push(self, val):
        self.data.append(val)

    def pop(self):
        assert not self.empty()
        ret = self.data[-1]
        del self.data[-1]
        return ret

    def peek(self):
        assert not self.empty()
        return self.data[-1]

    def empty(self):
        return len(self.data) == 0

    def __bool__(self):
        return not self.empty()
```

[2]:
```
s = Stack()
for x in range(10):
    s.push(x)
```

[3]:
```
s.peek()
```

[3]: 9

[4]:
```
while s:
    print(s.pop())
```

```
9
8
7
6
5
4
3
2
1
0
```

1.3.2 Singly-linked Stack

[5]:
```
# linked implementation

class Stack:
    class Node:
        def __init__(self, val, next=None):
            self.val = val
            self.next  = next

    def __init__(self):
        self.top = None

    def push(self, val):
        self.top = Stack.Node(val, next=self.top)

    def pop(self):
        assert not self.empty()
        ret = self.top.val
        self.top = self.top.next
        return ret

    def peek(self):
        assert not self.empty()
        return self.top.val
```

```
    def empty(self):
        return self.top is None

    def __bool__(self):
        return not self.empty()
```

[6]:
```
s = Stack()
for x in range(10):
    s.push(x)
```

[7]:
```
s.peek()
```

[7]: 9

[8]:
```
while s:
    print(s.pop())
```

```
9
8
7
6
5
4
3
2
1
0
```

1.3.3 … for delimiter pairing

Stacks are used by parsers to decide if expressions which make use of paired delimiters (e.g., `()`, `[]`, `<>`, `<tag></tag>`) are *well-formed*.

e.g., are all the parentheses in `'(1 + 2 * (3 - 4 / 5 + 6) - (7 + 8))'` matched up correctly?

[9]:
```
def check_parens(expr):
    s = Stack()
    for c in expr:
        if c == '(':
            s.push(c)
        elif c == ')':
            if s.empty():
                return False
            elif s.pop() != '(':
                return False
    return s.empty()
```

[10]:
```
check_parens('()')
```

[10]: True

[11]:
```
check_parens('((()))')
```

[11]: True

[12]:
```
check_parens('()(()()(()))')
```

[12]: True

[13]:
```
check_parens('(')
```

[13]: False

[14]:
```
check_parens('())')
```

[14]: False

[15]:
```
check_parens('(1 + 2 * (3 - 4 / 5 + 6) - (7 + 8))')
```

[15]: True

1.3.4 … for postfix expression evaluation

Arithmetic expressions are commonly written in *infix form* (e.g., "(1 + 2 * (3 - 4 / 5 + 6) - (7 + 8))"), as it is more intuitive for humans to read and write. However, to evaluate such expressions implicit rules of precedence and associativity must be known and correctly applied. For this reason, it is not an ideal notation for automated evaluation.

Postfix form (aka "reverse polish notation") allows arithmetic expressions to be specified unambiguously and evaluated without applying any rules of precedence or associativity. The infix expression "(1 + 2 * (3 - 4 / 5 + 6) - (7 + 8))" in postfix looks like this: "1 2 3 4 5 / - 6 + * + 7 8 + -".

Stacks are used to help evaluate postfix arithmetic expressions.

[16]:
```
def eval_postfix(expr):
    s = Stack()
    toks = expr.split()
    for t in toks:
        if t.isdigit():
            s.push(int(t))
        elif t == '*':
            s.push(s.pop() * s.pop())
        elif t == '+':
            s.push(s.pop() + s.pop())
    return s.pop()
```

[17]:
```
# (1 + 2) * 5
eval_postfix('1 2 + 5 *')
```

[17]: 15

[18]:
```
# 1 + 2 * 5
eval_postfix('1 2 5 * +')
```

[18]: 11

[19]:
```
# 10 + (1 + 2) * (3 + 2)
eval_postfix('10 1 2 + 3 2 + * +')
```

[19]: 25

Note: a stack can also be used to translate infix expressions to postfix!

1.3.5 … for tracking execution and backtracking

[20]:
```
maze_str = """######
              I    #
              # ## #
              # ####
              #    O
              ######"""

def parse_maze(maze_str):
    '''Parses a string representing a maze into a 2D array.'''
    grid = []
    for line in maze_str.split('\n'):
        grid.append(['# IO'.index(c) for c in line.strip()])
    return grid

def print_maze(grid):
    '''Takes a 2D array maze representation and pretty-prints it.
       The contents of the 2D maze are in the range 0-5, which are interpreted␣
 ↪as:

       0: a wall
       1: an unvisited (i.e., not previously traversed) path
       2: the maze entrance
       3: the maze exit
       4: a discovered but unvisited path
       5: a visited path
    '''
    for r in grid:
        print(''.join('# IO!+'[c] for c in r))
```

[21]:
```
parse_maze(maze_str)
```

[21]:
```
[[0, 0, 0, 0, 0, 0],
 [2, 1, 1, 1, 1, 0],
 [0, 1, 0, 0, 1, 0],
 [0, 1, 0, 0, 0, 0],
 [0, 1, 1, 1, 1, 3],
 [0, 0, 0, 0, 0, 0]]
```

[22]:
```python
print_maze(parse_maze(maze_str))
```

```
######
I    #
# ## #
# ####
#    O
######
```

[23]:
```python
maze = parse_maze(maze_str)
maze[1][0] = maze[1][1] = 5
maze[1][2] = maze[2][1] = 4
print_maze(maze)
```

```
######
++!  #
#!## #
# ####
#    O
######
```

[24]:
```python
class Move:
    '''Represents a move in the maze between orthogonally adjacent locations
      `frm` and `to`, which are both (row,col) tuples.'''
    def __init__(self, frm, to):
        self.frm = frm
        self.to  = to

    def __repr__(self):
        return f'({self.frm[0]},{self.frm[1]}) -> ({self.to[0]},{self.to[1]})'

def moves(maze, loc):
    '''Returns all possible moves within a maze from the provide location.'''
    moves = [Move(loc, (loc[0]+d[0], loc[1]+d[1]))
             for d in ((-1, 0), (1, 0), (0, -1), (0, 1))
             if loc[0]+d[0] in range(len(maze)) and
                loc[1]+d[1] in range(len(maze[0])) and
                maze[loc[0]+d[0]][loc[1]+d[1]] in (1, 2, 3)]
    return moves
```

[25]:
```python
maze = parse_maze(maze_str)
print_maze(maze)
```

```
######
I    #
# ## #
# ####
#    O
######
```

[26]:
```python
moves(maze, (1, 0))
```

[26]:
```
[(1,0) -> (1,1)]
```

[27]:
```python
moves(maze, (1, 1))
```

[27]:
```
[(1,1) -> (2,1), (1,1) -> (1,0), (1,1) -> (1,2)]
```

[28]:
```python
maze[1][0] = 5
moves(maze, (1, 1))
```

[28]:
```
[(1,1) -> (2,1), (1,1) -> (1,2)]
```

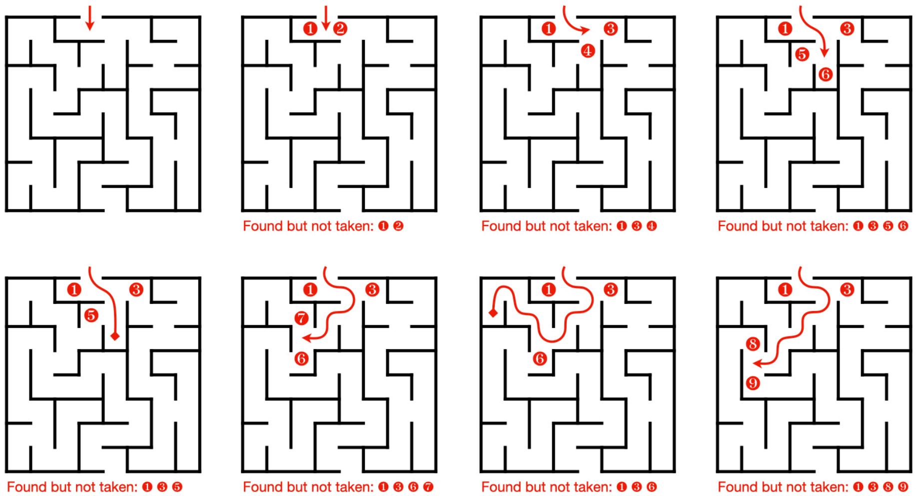
[29]:
```python
from time import sleep
from IPython.display import clear_output

def mark(maze, loc):
    '''Marks a loc in the maze as having been discovered'''
    if maze[loc[0]][loc[1]] != 3:
        maze[loc[0]][loc[1]] = 4

def visit(maze, loc):
    '''Marks a loc in the maze as having been visited'''
    maze[loc[0]][loc[1]] = 5

def display(maze):
    '''Prints out the maze after clearing the cell -- useful for animation.'''
    clear_output(True)
    print_maze(maze)
    sleep(0.5)
```

[30]:
```python
def solve_maze(maze, entry):
    '''Searches for the exit in a maze starting from the given entry point.

       The algorithm works as follows:

       1. Visit the entry point and save all possible moves from that location.
```

```
        2. Remove and consider one of the saved moves. If it is the exit, we are␣
↪done,
           otherwise visit the destination and save all possible moves from␣
↪there.
        3. If we run out of saved moves, we can't find an exit.

        When we save a move, we also mark it as "discovered" in the maze.

        The data structure used to save moves plays a critical role in how maze
        exploration proceeds!
      '''
    for m in moves(maze, entry):
        save_move(m)
    visit(maze, entry)
    while not out_of_moves():
        move = next_move()
        if maze[move.to[0]][move.to[1]] == 3:
            break
        display(maze)
        visit(maze, move.to)
        for m in moves(maze, move.to):
            mark(maze, m.to)
            save_move(m)
    display(maze)
```

[31]:

```
move_stack = Stack()

def save_move(move):
    move_stack.push(move)

def next_move():
    return move_stack.pop()

def out_of_moves():
    return move_stack.empty()
```



[32]:
```
maze_str = """######
              I    #
              # ## #
              # ####
              #    O
              ######"""
solve_maze(parse_maze(maze_str), (1, 0))
```

```
######
+++++#
#+##+#
#+####
#++++O
######
```

[33]:
```
maze_str = """#################
              I #       #     #
              # ##### # # # # #
              #     # # # # # #
              # ### ### # # ###
              #   #       #   O
              #################"""

solve_maze(parse_maze(maze_str), (1, 0))
```

```
#################
++#       #+++++#
#+##### # #+#+#+#
#+++++# # #+#+#+#
#!###+###!#+#+###
```

```
#   #+++++++#+++O
#################
```

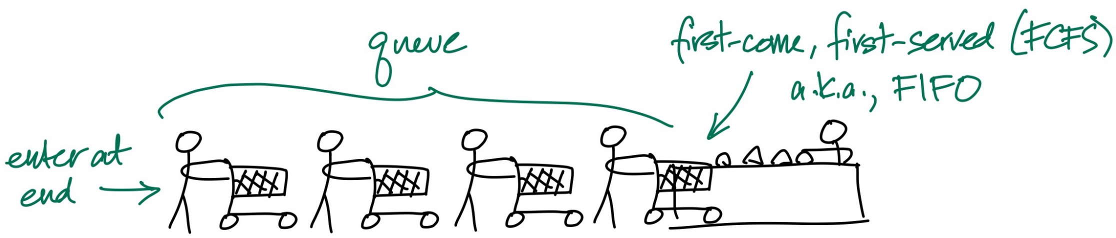
```
[34]: maze_str = """#################
                 I               #
                 # # # # # # # # #
                 # # # # # # # # #
                 # ###############
                 #               O
                 #################"""

      solve_maze(parse_maze(maze_str), (1, 0))
```

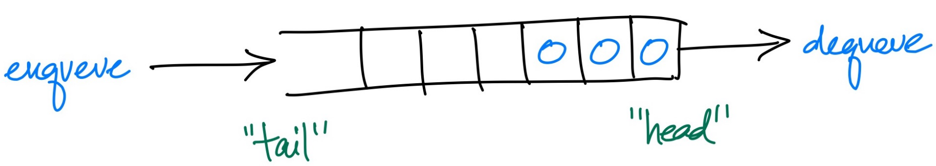
```
#################
++++++++++++++++#
#+#+#+#+#+#+#+#+#
#+#+#+#+#+#+#+#+#
#+###############
#+++++++++++++++O
#################
```

Intuitively, because the stack is a last-in, first-out data structure, it keeps following moves down the most recently discovered path until it either reaches the exit or reaches a dead end. It then picks up from the previously discovered path. We call this type of exploration *depth-first traversal*.

1.4 2. Queues

The **queue** is an ADT which only permits us to append ("enqueue") items at the tail end, and remove ("dequeue") items from the front. The oldest item still in a queue is therefore the next one to be dequeued, which is why we refer to a queue as a first-in, first-out (FIFO) structure. It is helpful to think of a queue as being the model for a line at a typical supermarket checkout aisle (first customer in, first customer to be checked out).





1.4.1 Array-backed Queue

[35]:
```python
# array-backed implementation

class Queue:
    def __init__(self):
        self.data = []
        self.head = -1

    def enqueue(self, val): # O(1)
        self.data.append(val)

    def dequeue(self): # O(1), but very space inefficient!
        assert not self.empty()
        self.head += 1
        ret = self.data[self.head]
        self.data[self.head] = None
        return ret

    def empty(self):
        return self.head + 1 == len(self.data)

    def __bool__(self):
        return not self.empty()
```

[36]:
```python
q = Queue()
for x in range(10):
    q.enqueue(x)
```

[37]:
```python
while q:
    print(q.dequeue())
```

```
0
1
2
3
4
5
6
7
8
9
```

1.4.2 Circular Array-backed Queue

[38]:
```
# circular array-backed implementation (partial)

class Queue:
    def __init__(self, size):
        self.data = [None] * size
        self.head = self.tail = -1

    def enqueue(self, val): # O(1)
        self.tail = (self.tail + 1) % len(self.data)
        self.data[self.tail] = val

    def dequeue(self): # O(1)
        self.head = (self.head + 1) % len(self.data)
        ret = self.data[self.head]
        self.data[self.head] = None # not really needed
        return ret
```

[39]:
```
q = Queue(10)
for x in range(6):
    q.enqueue(x)
```

[40]:
```
q.data
```

[40]:
```
[0, 1, 2, 3, 4, 5, None, None, None, None]
```

[41]:
```
for x in range(5):
    print(q.dequeue())
```

```
0
1
2
3
4
```

[42]:
```
q.data
```

[42]:
```
[None, None, None, None, None, 5, None, None, None, None]
```

[43]:
```
for x in range(6, 12):
    q.enqueue(x)
```

[44]:
```
q.data
```

[44]:
```
[10, 11, None, None, None, 5, 6, 7, 8, 9]
```

1.4.3 Singly-linked Queue

[45]:
```python
# linked implementation

class Queue:
    class Node:
        def __init__(self, val, next=None):
            self.val = val
            self.next  = next

    def __init__(self):
        self.head = self.tail = None

    def enqueue(self, val): # O(1)
        if self.tail:
            self.tail.next = self.tail = Queue.Node(val)
        else:
            self.head = self.tail = Queue.Node(val)

    def dequeue(self): # O(1)
        assert not self.empty()
        ret = self.head.val
        self.head = self.head.next
        if not self.head:
            self.tail = None
        return ret

    def empty(self):
        return self.head is None

    def __bool__(self):
        return not self.empty()
```

[46]:
```python
q = Queue()
for x in range(10):
    q.enqueue(x)
```

[47]:
```python
while q:
    print(q.dequeue())
```

```
0
1
2
3
4
5
6
7
```

8
9

1.4.4 … for tracking execution and backtracking

[48]:
```
move_queue = Queue()

def save_move(move):
    move_queue.enqueue(move)

def next_move():
    return move_queue.dequeue()

def out_of_moves():
    return move_queue.empty()
```

[49]:
```
maze_str = """######
              I    #
              # ## #
              # ####
              #    O
              ######"""

solve_maze(parse_maze(maze_str), (1, 0))
```

```
######
+++++#
#+##+#
#+####
#++++O
######
```

[50]:
```
maze_str = """#################
              I #       #     #
              # ##### # # # # #
              #     # # # # # #
              # ### ### # # ###
              #   #       #   O
              #################"""

solve_maze(parse_maze(maze_str), (1, 0))
```

```
#################
++#+++++++#+++++#
#+#####+#+#+#+#+#
#+++++#+#+#+#+#+#
#+###+###+#+#+###
#+++#+++++++#+++O
#################
```

```
[51]: maze_str = """#################
              I               #
              # # # # # # # # #
              # # # # # # # # #
              # ###############
              #               O
              #################"""

      solve_maze(parse_maze(maze_str), (1, 0))
```

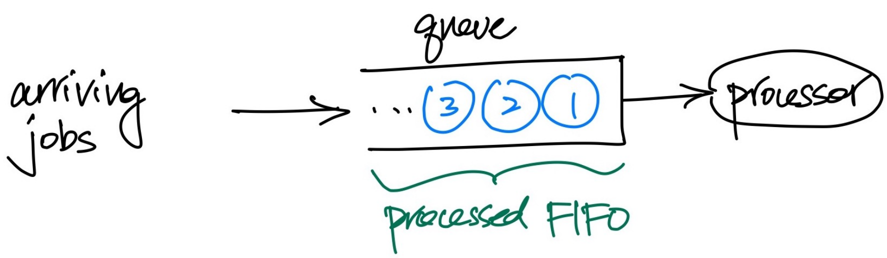
```
#################
++++++++++++++++#
#+#+#+#+#+#+#+#+#
#+#+#+#+#+#+#+#+#
#+###############
#+++++++++++++++O
#################
```

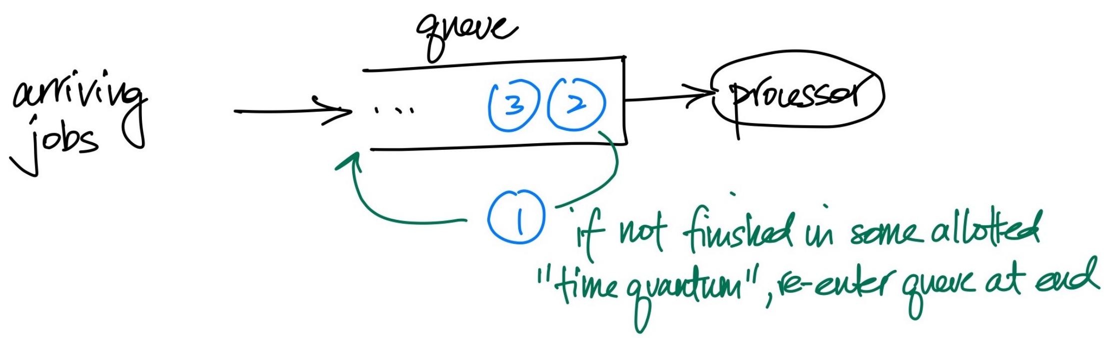
Intuitively, because the queue is a first-in, first-out – i.e., *fair* – data structure, it keeps rotating through all the paths which haven't yet dead-ended, making just one move further down each time. We call this type of exploration *breadth-first traversal*.

Are there types of mazes which might be more suitably tackled using one approach over the other (i.e., depth-first vs. breadth-first)?

1.4.5 … for fair scheduling (aka "round-robin" scheduling)

Queues are often used to help allocate resources in a fair way to different entities that require them. E.g., an operating system may use a queue to allocate processing time to different jobs running on a computer. A **round-robin scheduler** allows each job to run for a fixed *time quantum* on the processor; if it does not complete in that time then it re-enters the queue at the end:





Here we implement a "round-robin" scheduler for permitting different tasks to run for small, fixed periods of time until they complete:

[52]:
```
from random import randint

# create a bunch of jobs with random lengths
job_queue = Queue()
for i in range(5):
    job_queue.enqueue((f'Job {i}', randint(1, 5)))

# manually print out the jobs
n = job_queue.head
while n:
    print(n.val)
    n = n.next
```

```
('Job 0', 1)
('Job 1', 2)
('Job 2', 1)
('Job 3', 5)
('Job 4', 4)
```

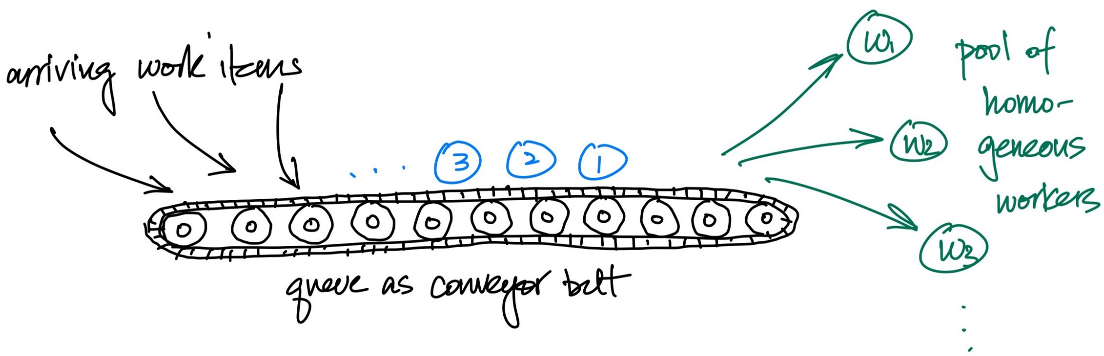
[53]:
```
from time import sleep

# scheduler loop
while job_queue:
    job, time_left = job_queue.dequeue()  # grab job at front of queue
    print(f'[\x1b[31mRUNNING\x1b[0m] {job}')
```

```
    sleep(1)  # run it for 1 second
    time_left -= 1

    # requeue if necessary
    if time_left > 0:
        print(f'[\x1b[33mREQUEUE\x1b[0m] {job} with time remaining =␣
↪{time_left}')
        job_queue.enqueue((job, time_left))
    else:
        print(f'[\x1b[32mCOMPLETED\x1b[0m] {job}')
```

```
[RUNNING] Job 0
[COMPLETED] Job 0
[RUNNING] Job 1
[REQUEUE] Job 1 with time remaining = 1
[RUNNING] Job 2
[COMPLETED] Job 2
[RUNNING] Job 3
[REQUEUE] Job 3 with time remaining = 4
[RUNNING] Job 4
[REQUEUE] Job 4 with time remaining = 3
[RUNNING] Job 1
[COMPLETED] Job 1
[RUNNING] Job 3
[REQUEUE] Job 3 with time remaining = 3
[RUNNING] Job 4
[REQUEUE] Job 4 with time remaining = 2
[RUNNING] Job 3
[REQUEUE] Job 3 with time remaining = 2
[RUNNING] Job 4
[REQUEUE] Job 4 with time remaining = 1
[RUNNING] Job 3
[REQUEUE] Job 3 with time remaining = 1
[RUNNING] Job 4
[COMPLETED] Job 4
[RUNNING] Job 3
[COMPLETED] Job 3
```

1.4.6 … for doling out work

Queues are also frequently used as a sort of conveyer belt for a pool of homogeneous workers to draw from.



Here we implement this "work queue" pattern and use it to communicate work items to a pool of concurrent threads of execution:

[54]:
```
from threading import Thread
from queue import Queue
from time import sleep
from random import random


class Worker(Thread):
    def __init__(self, wid, queue):
        super().__init__()
        self.wid= wid
        self.queue = queue

    def run(self):
        print(f'Worker {self.wid} starting up')
        while True:
            work = self.queue.get()     # retrieve a work item from the queue
            if work == 'Stop':
                print(f'Worker {self.wid} stopping.')
                break
            else:
                print(f'Worker {self.wid} processing {work}')
                sleep(random())        # pretend to do some work (with random␣
↪duration)
                self.queue.task_done() # indicate that we've finished the work␣
↪item


# create a work queue
work_queue = Queue()

# create a bunch of workers that monitor the queue for work items
for i in range(5):
    w = Worker(i, work_queue)
```

```
    w.start()
```

```
Worker 0 starting up
Worker 1 starting up
Worker 2 starting up
Worker 3 starting up
Worker 4 starting up
```

[55]:

```python
# add a bunch of work items to the queue
for i in range(50):
    work_queue.put(i)

# wait for all work items to be processed
work_queue.join()
```

```
Worker 0 processing 0Worker 1 processing 1Worker 2 processing 2Worker 3
processing 3Worker 4 processing 4




Worker 0 processing 5
Worker 2 processing 6
Worker 0 processing 7
Worker 3 processing 8
Worker 2 processing 9
Worker 1 processing 10
Worker 4 processing 11
Worker 1 processing 12
Worker 0 processing 13
Worker 4 processing 14
Worker 2 processing 15
Worker 1 processing 16
Worker 0 processing 17
Worker 4 processing 18
Worker 3 processing 19
Worker 2 processing 20
Worker 3 processing 21
Worker 1 processing 22
Worker 4 processing 23
Worker 0 processing 24
Worker 0 processing 25
Worker 2 processing 26
Worker 3 processing 27
Worker 0 processing 28
Worker 2 processing 29
Worker 1 processing 30Worker 3 processing 31
```

```
Worker 4 processing 32
Worker 3 processing 33
Worker 1 processing 34
Worker 2 processing 35
Worker 0 processing 36
Worker 2 processing 37
Worker 4 processing 38
Worker 1 processing 39
Worker 2 processing 40
Worker 3 processing 41
Worker 0 processing 42
Worker 2 processing 43
Worker 2 processing 44
Worker 3 processing 45
Worker 4 processing 46
Worker 1 processing 47
Worker 0 processing 48
Worker 0 processing 49
```

[56]:
```
# order all workers to terminate
for i in range(5):
    work_queue.put('Stop')
```

```
Worker 2 stopping.Worker 1 stopping.Worker 3 stopping.Worker 4 stopping.Worker 0
stopping.
```

1.5 3. Run-time analysis

Stack & Queue implementations:

- Insertion (push and enqueue) = $O(1)$
- Deletion (pop and dequeue) = $O(1)$