Computer Science I — CSCI141
Stacks and Queues — Lecture

08/26/2013

1 Problem Statement

This week we will be implementing a simulation inspired by the popular 80's action movie, Predator. The premise is that a number of hostages are trapped in the jungle and are being ruthlessly hunted by a superior alien predator. Their goal is to reach "da choppa" (a famous quote from the movie by Arnold Schwarzenegger) and be flown away to safety.

In the first stage, the hostages are chased into a dark, narrow cavern that seemingly goes on forever, with only one entrance (which is also the only exit). The cavern is only wide enough for one person and does not allow people to shift around once they enter it. The predator is blocking the exit and the hostages must fight their way out, one at a time, until either the predator or the hostages are all defeated.

If any hostages survive, they leave the cavern and come to a perilous gorge that separates them from "da choppa". An old, narrow, wooden rope bridge spans the gorge. The bridge is very weak and can only support so much weight. Like the cavern, it is only wide enough for one hostage, Once they step onto the bridge, the hostages may not shift around - they can only head towards the other side. The bridge can only hold a limited number of people. Once it becomes full, the person that is closest to the other side is allowed to exit. The hostages step onto the bridge one at a time. If the bridge supports their total weight, they reach "da choppa" and survive. Otherwise, the bridge collapses, and the hostages who have stepped onto the bridge fall into the ravine below. Ironically, "da choppa" also randomly explodes if the bridge collapses, thus leaving the remaining hostages stranded in the jungle forever.

For this problem, we will only use the data structures we develop in this writeup to handle the collection of hostages. We explicitly will not use lists, tuples or dictionaries.

2 The Predator and The Hostages

2.1 Representing the Predator

Our predator is a very simple class. It only has a random number of hit points between `1000-2000`:

```
def GET_PREDATOR_HITPOINTS():
    return randint(1000, 2000)

class Predator():
    __slots__ = ('hitPoints')

def mkPredator():
    p = Predator()
    p.hitPoints = GET_PREDATOR_HITPOINTS()
    return p
```

2.2 Representing the Hostages

The hostages are a slightly more detailed class. Each hostage has three attributes:

- `name`: A name. This is a string. As a homage to the original Star Trek television series, our hostages will affectionately be named `Red Shirt <num>`, where `<num>` ranges from `1` to the total number of hostages.
- `hitPoints`: The amount of life. This is an integer in the range of `100-200` and will be randomly generated by a function named `GET_HOSTAGE_HITPOINTS`.
- `weight`: The weight (in pounds). This is an integer in the range of `50-300` and will be randomly generated by a function named `GET_HOSTAGE_WEIGHT`.

This results in the following code for defining the class, `Hostage`, and a function, `mkHostage`, for creating a new hostage:

```
from random import *    # randint

def GET_HOSTAGE_HITPOINTS():
    return randint(100, 200)

def GET_HOSTAGE_WEIGHT():
    return randint(50, 300)
```

```
class Hostage():
    __slots__ = ('hitPoints', 'name', 'weight')

def mkHostage(id):
    h = Hostage()
    h.hitPoints = GET_HOSTAGE_HITPOINTS()
    h.name = "Red Shirt #" + str(id)
    h.weight = GET_HOSTAGE_WEIGHT()
    return h
```

3 Stack Overview

First we will deal with the problem of putting the hostages in the cavern and having them fight their way out against the predator. We will use a *stack* to represent the cavern.

A stack is a simplified kind of list where items can only be added and removed from one end. It is referred to as either a Last In First Out (LIFO) or First in Last Out (FILO) structure. That end is called, not surprisingly, the *top*. This defines the order the hostages will face the predator. The stack supports the following operations. Here, an `element` refers to an individual hostage.

- `push`: Insert an element onto the top of the stack.
- `top`: Peek at the top element on the stack, without removing it.
- `pop`: Remove the top element from the stack.
- `emptyStack`: A boolean function that indicates whether the stack is empty or not
- `size`: Get the number of elements that are in the stack.

3.1 Singly Linked Node

Since we are not using a list to store the hostages, we need to come up with a way to store them ourselves. We will use a *singly linked node* based representation. Each node in the stack will hold a hostage, as well as a "pointer" to the next hostage. The top of the stack, therefore, is simply a pointer to the first node in the collection.

We will implement our node in the file `myNode.py`. Initially we will use the value `None` to indicate that a pointer doesn't point to anything (for example, if the stack is empty, or we've reached the last hostage in the stack).

```
class Node():
    __slots__ = ( 'data', 'next' )

def mkNode(dataVal, nextVal):
    """Create and return a newly initialized Node object"""
    node = Node()
    node.data = dataVal
```

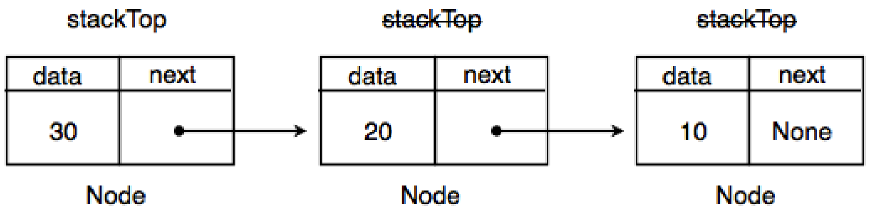
```
        node.next = nextVal
        return node
```

3.2 Stack Implementation - Phase 1

With this node representation, we can now create a simple stack. Eventually we will be using `Hostage`'s as the `data` attribute, but for now we will just use integers. Let's see what happens when we `push` the values 10, 20 and 30 onto the stack:

```
from myStack import *

stackTop = None                       # create an empty stack
stackTop = mkNode(10, None)           # the bottom node in the stack
stackTop = mkNode(20, stackTop)       # the middle node in the stack
stackTop = mkNode(30, stackTop)           # the top node in the stack
```

This results in a stack that looks like the following:



Alternately, we could create the entire stack in one statement, without needing temporary variables:

```
stackTop = mkNode(30, mkNode(20, mkNode(10, None)))
```

We can now see how the operation `top` works. It simply accesses the data attribute of the top node in the stack:

```
print("Top element:", stackTop.data)
```

But what happens if the stack is empty? If the top points to `None`, it will cause a run-time error:

```
AttributeError: 'NoneType' object has no attribute 'data'
```

This is easy enough to detect, and defines the `emptyStack` function. We can rewrite the code above as follows:

```
if stackTop == None:
    print("The stack is empty!")
else:
    print("Top element:", stackTop.data)
```

Since the `top` operation is required to return the top element, and the `pop` operation is required to remove the top element, it is considered a fatal operation to call these if the stack is empty. If this happens, we will cause the program to terminate by *raising an exception*.

```
if stackTop == None:
    raise IndexError("Error, the stack is empty!")
else:
    print("Top element:", stackTop.data)
```

When run on an empty stack:

```
line 2, IndexError: Error, the stack is empty!
```

Finally, how does the `pop` operation work? Well, the element at the top needs to get removed. This is achieved by ”advancing” the top to point to the next node in the stack. In other words, if the stack is not empty you can pop as follows:

```
stackTop = stackTop.next
```

Putting this all together results in our first pass at a singly linked node based stack:

```
from myNode import *

def push(node, element):
    """Add an element to the top of the stack"""
    newnode = mkNode(element, node)
    return newnode

def top(node):
    """Return top element on stack.  Does not change stack"""
    if emptyStack(node):
        raise IndexError("top on empty stack")
    return node.data

def pop(node):
    """Remove the top element in the stack.  Returns new top"""
    if emptyStack(node):
        raise IndexError("pop on empty stack")
    return node.next

def emptyStack(node):
    """Is the stack empty?"""
    return node == None
```

Take some time to look this over, as well as the test program, `testStack.py`. All three files are on the course website under the lecture link for week 10 in the `Stack/Phase1` directory.

3.3 A Different Approach - The Problem With None

In the second phase of our stack implementation, we will address the poor programming practice of using `None`, as well as implement the `size` function.

Many software experts advocate against using `None`. One reason is that Python's `None`, and other similar values in other programming languages, are used as a catch-all for anything that means "there is nothing here". However, if during execution of your program you encounter `None` in a spot where you did not expect it, the debugging process can be quite time-consuming. The reason is that the value stored in your structure gives you no clue as to what piece of code executed and put it there. If instead, for each distinct application you develop a "null class" instead of using None, then if that value pops up at the wrong time, you have a better idea of which part of your system put it there.

3.4 Stack Implementation - Phase 2

We will start by creating a global constant variable, `NONE_NODE`, a variable of the type `NoneNode`, to represent a pointer that points to nothing,

```
class NoneNode():
    __slots__ = ()

NONE_NODE = NoneNode()
```

Using this, we can create an empty stack as follows:

```
stack = NONE_NODE
```

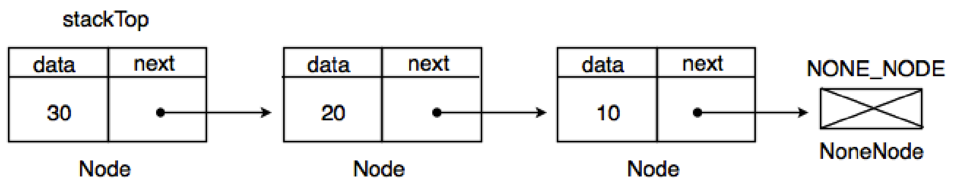
Internally, there are two ways that we can check whether the stack is empty. We can check directly for an equivalence:

```
def emptyStack(node):
    return node == NONE_NODE
```

Or alternately, we can use the `isinstance(obj, type)` method. This is a built-in boolean function that tells whether `obj` is of `type`, or not.

```
def emptyStack(node):
    return isinstance(node, NoneNode)
```

Here is a picture of the same stack using the `NONE_NODE` as the terminating value:



Finally, let's write the `size` function. Hopefully you see that this is a simple recursive function. We will traverse through the stack, from the top node to the bottom node, adding 1 each time we recurse and returning the result.

```
def size(node):
    """Return the # of elements including this node"""
    if emptyStack(node):
        return 0
    else:
        return 1 + size(node.next)
```

Take some time to look this over, as well as the test program, `testStack.py`. All three files are on the course website under the lecture link for week 10 in the `Stack/Phase2` directory.

3.5 Putting It All Together - The Cavern

Now we can implement the first stage of the simulation. The main program can be found in `Choppa/choppa.py` on the course website. It prompts for the number of hostages and then creates the cavern (a stack) by pushing each of them into it. In the end, the cavern, which is the top, points to the top hostage on the stack.

```
    # create the hostages and push them into the cavern
    numHostages = int(input("How many hostages? "))
    cavern = NONE_NODE
    for id in range(1, numHostages+1):
        hostage = mkHostage(id)
        cavern = push(cavern, hostage)
```

As this simulation runs, there is a loop that executes as each hostage battles the predator. One at a time, a hostage is `top`'d and `pop`'d off the stack. The hostage does battle until either their hit points go to 0, and/or the predator's does. This results in one of two scenarios:

- The predator dies and the hostage who was at the top survives. In this case, the surviving hostage is pushed back onto the top of the stack.
- The predator survives and all the hostages die. In this case, the stack is empty and the simulation is over.

The implementation of the cavern can be found in `Choppa/cavern.py` on the course website. It is implemented in the `surviveTheCavern` function. Here is a simplification that highlights the stack operations:

```
    def surviveTheCavern(predator, cavern):
        """Returns the surviving hostages in the cavern (stack), if any"""
        while not emptyStack(cavern) and predator.hitPoints > 0:
            hostage = top(cavern)
            cavern = pop(cavern)
            while predator.hitPoints > 0 and hostage.hitPoints > 0:
                dmgToPredator = GET_HOSTAGE_DAMAGE()
                predator.hitPoints -= dmgToPredator
                if (predator.hitPoints > 0):
                    dmgToHostage = GET_PREDATOR_DAMAGE()
                    hostage.hitPoints -= dmgToHostage

            if predator.hitPoints <= 0:
                print(hostage.name, "has defeated the Predator!")
                cavern = push(cavern, hostage)
            if hostage.hitPoints <= 0:
                print(hostage.name, "has fallen!")

        # did anyone survive?
        print(str(size(cavern)), "hostage/s survived the dark cavern...")
        return cavern
```

4 Queue Overview

Assuming at least one hostage survives the battle against the predator, the next stage of the simulation presents the hostages with a bridge obstacle. The bridge has one entrance on the side of the gorge the hostages are on, and an exit on the other side of the gorge where ”da choppa” is.

The bridge can be represented as a *queue*. A queue is a First In First Out (FIFO) or Last In Last Out (LILO) structure. Like the stack, we will use the same singly linked node representation. Unlike the stack, we have to maintain two pointers. The first pointer is to the `front`, the front of the queue (the exit of the bridge). The second pointer is to the `back`, the back of the queue (the entrance to the bridge). The queue supports the following operations:

- `enqueue`: Insert an element onto the back of the queue
- `dequeue`: Remove the front element from the queue
- `front`: Peek at the front element in the queue, without removing it
- `back`: Peek at the back element in the queue, without removing it
- `emptyQueue`: A boolean function that indicates whether the queue is empty or not
- `size`: Get the number of elements that are in the queue

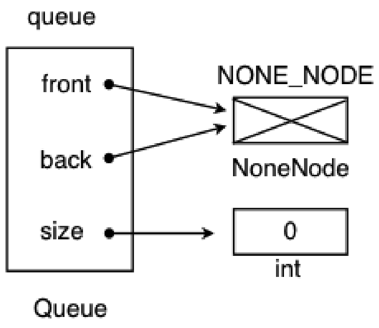
For our queue implementation, we will create a class that ”encapsulates” both pointers, as well as the size of the queue. Here is the class definition, `Queue`, and the function to create an initially empty queue.

```
    from myNode import *

    class Queue():
        __slots__ = ( 'front', 'back', 'size' )

    def mkQueue():
        q = Queue()
        q.front = NONE_NODE
        q.back = NONE_NODE
        q.size = 0
        return q
```

When created:

```
queue = mkQueue()
```

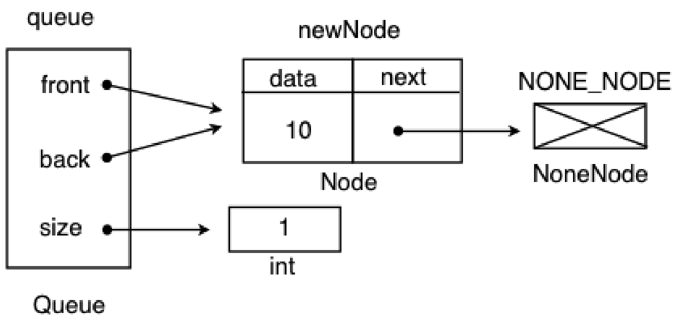
It looks like:



The check for `emptyQueue` is similar to the stack. You can either check whether the front or back pointer refer to `NONE_NODE` (the queue is empty), or not (the queue is not empty).

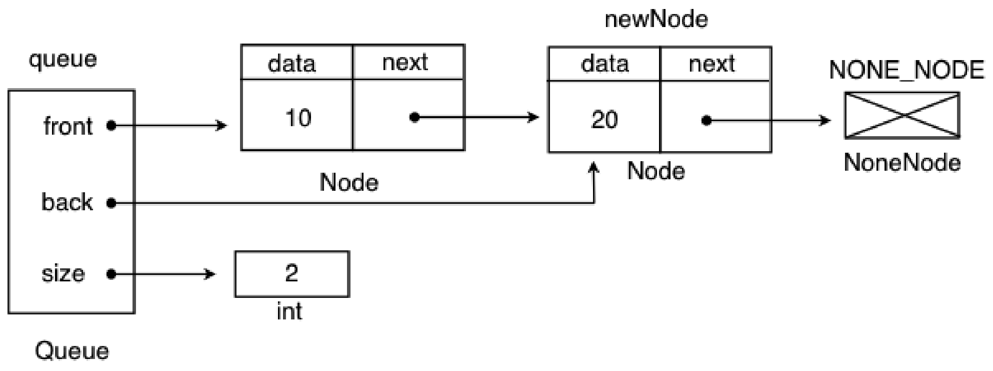
There are two conditions that we have to deal with when enqueueing a new element into a queue. The first condition is to handle what happens when the queue is initially empty. In this scenario, both the front and back pointer need to be updated to point to the new node that was just added. The new node's next should point to the `NONE_NODE`:

```
enqueue(queue, 10)
```



In the second condition, the queue is not empty, so an enqueue only needs to modify the back pointer to point to the new node (with the new node's next pointing to the `NONE_NODE`). But first, the old back should be set to point to the new node.

```
enqueue(queue, 20)
```



Notice how in both cases, an enqueue should increase the size by one. Unlike the stack, our size is constantly being updated. And here, `size` is not a function, it is merely an attribute of the `Queue` that can be accessed at any time:

```
print("There are", queue.size, "elements in the queue")
```

As long as there are elements in the queue, the `front` and `back` operations are trivial. They simply return the data value associated with the node they point to.

```
print(There front element in the queue is", queue.front.data)
print(There back element in the queue is", queue.back.data)
```

The `dequeue` operation is very similar to the stack's `pop` operation. Here, the front element needs to be removed from the queue. To do this, you can ”advance” the front pointer to point to the next element.

```
queue.front = queue.front.next
```

There is one special condition. If you dequeue the last element in the queue, the front pointer will correctly go to the `NONE_NODE`. However, the back pointer will still point to that node. In this case, you should change the back pointer to also point to the `NONE_NODE`.

```
queue.back= NONE_NODE
```

With both cases, make sure to decrement the size of the queue by one.

This is the resulting singly node based queue implementation. It is located on the course website in the `Queue` sub-directory.

```
def enqueue(queue, element):
    """Insert an element into the back of the queue"""
    newnode = mkNode(element, NONE_NODE)
    if emptyQueue(queue):
        queue.front = newnode
    else:
        queue.back.next = newnode
    queue.back = newnode
    queue.size += 1

def dequeue(queue):
    """Remove the front element from the queue (returns None)"""
    if emptyQueue(queue):
        raise IndexError("dequeue on empty queue")
    queue.front = queue.front.next
    if emptyQueue(queue):
        queue.back = NONE_NODE
    queue.size -= 1

def front(queue):
    """Access and return the first element in the queue without removing it"""
    if emptyQueue(queue):
        raise IndexError("front on empty queue")
    return queue.front.data

def back(queue):
    """Access and return the last element in the queue without removing it"""
    if emptyQueue(queue):
        raise IndexError("back on empty queue")
    return queue.back.data

def emptyQueue(queue):
    """Is the queue empty?"""
    return queue.front == NONE_NODE
```

4.1 Putting It All Together - The Bridge

Now we can represent the bridge as a queue. This is implemented in the `Choppa/cavern.py` source file. There is a function, `crossTheBridge` that takes the surviving hostages from the cavern (the stack), and an initially empty bridge (the queue).

First, the total weight the bridge can hold is randomly determined in the range `700-1100`. Second, the bridge can hold a random number of people between `3-5`.

Next, we enter a loop where each hostage exits the cavern and enters the bridge. There are two conditions that could cause this loop to exit. First, if the bridge breaks, the simulation is over and no one else should try to enter the bridge. This results in hostages on the bridge falling to their demise, and the remaining hostages from the cavern being stranded. In the other condition, the bridge does not break and the cavern becomes empty. Any time the bridge is full, the first person to have entered the bridge exits it and reaches ”da choppa”.

This is a summary that highlights the stack and queue operations:

```
    def crossTheBridge(cavern, survivors):
        """Hostages leave the cavern and enter the bridge"""
```

```
bridge = mkQueue()
MAX_WEIGHT = GET_BRIDGE_MAX_WEIGHT()
MAX_PEOPLE = GET_BRIDGE_MAX_PEOPLE()

# take hostages one at a time from the cavern and enqueue
# them onto the bridge, until either it breaks or they
# all make it on
totalWeight = 0
brokeBridge = False
while not brokeBridge and not emptyStack(cavern):
    # if bridge at max capacity, remove the front person from it
    if bridge.size == MAX_PEOPLE:
        survivor = front(bridge)
        enqueue(survivors, front(bridge))
        dequeue(bridge)
        totalWeight -= survivor.weight

    # process the next person in the cavern
    hostage = top(cavern)
    cavern = pop(cavern)
    if not brokeBridge:
        enqueue(bridge, hostage)
        totalWeight += hostage.weight
        if totalWeight > MAX_WEIGHT:
            brokeBridge = True
            if survivors.size == 0:
                # the choppa explodes!

# if the bridge breaks, separate those who fall from those
# who are stranded
if brokeBridge:
    while not emptyQueue(bridge):
        # those who fall are toast
        dequeue(bridge)
    while not emptyStack(cavern):
        # those remaining in the cavern are stranded
        hostage = top(cavern)
        cavern = pop(cavern)
# deal with the survivors
else:
    while not emptyQueue(bridge):
        survivor = front(bridge)
        enqueue(survivors, front(bridge))
```

```
                dequeue(bridge)

        print(str(survivors.size), "hostage/s survived the bridge crossing...")
```

5 Complexity: Running time analysis

Because the stack works exclusively with the top, all of the operations, except `size`, have a time complexity of $O(1)$. Because `size` iterates through all of the nodes in the stack, it is $O(N)$ (where N is the number of elements in the stack).

All of the queue operations have a time complexity of $O(1)$ because the size and back pointer were added; there are no loops or recursive functions among the queue functions.

Overall, the entire simulation runs in $O(N)$ time (where N is the number of hostages).

6 Testing

The program uses randomization so you may have to try running it a few times.

- The predator wins. This is easy to demonstrate. Run the program with a small number of hostages (e.g. 5).
- No one wins. This is also easy to demonstrate. Run the program with a large number of hostages (e.g. 60).
- Some hostages win. Play with this one. I've found it usually works within the range of 20-30 hostages.