Computer Science I — CSCI 141
Stacks and Queues — Lecture

10/18/2020

1 Problem Statement

We will implement a simulation inspired by the Norwegian fairy tale the Billy Goats Gruff. The premise is that a number of goats are hungry and wish to eat the delicious berries in the meadow on the other side of the river. However, there is only one bridge, and it is guarded by a fearsome troll that eats goats. The troll must be defeated before the goats can cross over to the meadow in safety.

In the first stage, the goats 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 goat and does not allow goats to shift around once they enter it. The troll is blocking the exit and the goats must fight their way out, one at a time, until either the troll is defeated or all the goats are defeated.

If any goats survive, they leave the cavern and come to a perilous gorge that separates them from the meadow. An old, narrow, wooden rope bridge spans the gorge. The bridge is weak and can only support so much weight. Like the cavern, it is only wide enough for one goat, Once they step onto the bridge, the goats may not shift around — they can only head towards the other side. The bridge can only hold a limited number of goats. Once the bridge becomes full, the goat that is closest to the other side is allowed to exit. The goats step onto the bridge one at a time. If the bridge supports their total weight, they reach the meadow and eat the delicious berries. Otherwise, the bridge collapses, and the goats who have stepped onto the bridge fall into the ravine below.

While we could use Python data structures, such as lists, tuples, or perhaps dictionaries, we explicitly will not use these tools.

Instead, we will introduce the *linked node* data type as an "implementation foundation" on which we will develop *stack* and *queue* data structures that process a collection of elements in specific ways.

2 The Troll and The Goats

2.1 Representing the Troll

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

```
TROLL_HITPOINTS = randint(1000, 2000)

@dataclass
class Troll:
    hit_points: int

def make_troll():
    """
    Create and return a troll with random strength.
    """
    return Troll(TROLL_HITPOINTS)
```

2.2 Representing the Goats

The goats have a slightly more detailed structure. Each goat has three attributes:

- `name`: A name. This is a string. Our goats will affectionately be named `Goat <num>`, where `<num>` ranges from `1` to the total number of goats.
- `hit_points`: The amount of life. This is an integer in the range of `100-200` and will be randomly generated by a function named `get_goat_hit_points`.
- `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_goat_weight`.

Here is code for defining the structure, `Goat`, and a function, `make_goat`, for creating a new goat:

```
def get_goat_hit_points():
    return randint(100, 200)

def get_goat_weight():
    return randint(50, 300)

@dataclass
class Goat:
    name: str
    hit_points: int
    weight: int

def make_goat(id):
    return Goat("Goat #" + str(id), get_goat_hit_points(), get_goat_weight())
```

3 Stack Overview

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

A stack is a data structure 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 goats will face the troll. The stack supports the following operations, where an `element` refers to an individual goat.

- `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 and return the removed element.
- `is_empty`: Return `True` if the stack is empty and `False` otherwise.
- `size`: Return the number of elements that are in the stack.

3.1 Singly Linked Node: Immutable

Since we are not using a list to store the goats, 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 goat, as well as a reference to the next goat. The top of the stack, therefore, is simply a reference to the first node in the collection.

We will implement our node in the file `node_types.py` and use the `@dataclass` annotation:

```
@dataclass( frozen=True )
class FrozenNode:
    """
    An immutable link node containing a value and a link to the next node
    """
    value: Any
    next: Union[ "FrozenNode" , None ]
```

For reasons that you will see when we get into the stack implementation, the nodes of a stack do not need to change. We set `frozen=True` in the decorator to indicate the fields of the node are immutable and will not change.

When importing, the stack will do the following import and refer to `FrozenNode` as `Node`.

```
from node_types import FrozenNode as Node
```

3.2 Stack Implementation

To implement the actual stack, we will *encapsulate* a reference to the linked sequence. That means we create another data structure that contains, or points to, the linked sequence. Once we have this structure, we can add additional slots to keep track of information about our stack. In particular, we can keep track of the size of the stack. Here is the structure definition, `Stack`, and the function to create an initially empty stack.

```
@dataclass(frozen = True)
class Stack:
    size: int
    top: Union[None, Node]

def make_empty_stack():
    """
    Returns a new stack with size initialized to zero and
    top initialized to the empty list.
    """
    return Stack(0, None)
```

We can now define the stack operations as follows.

```
def push(stack, element):
    """
    Add an element to the top of the stack. Stack changes.
    """
    stack.top = Node(element, stack.top)
    stack.size = stack.size + 1

def top(stack):
    """
    Return top element on stack. Does not change stack.
    """
    if is_empty(stack):
        raise IndexError("top of empty stack")
    return stack.top.value

def pop(stack):
    """
    Remove the top element in the stack and reutns it.
    Stack changes.
    """
    if is_empty(stack):
        raise IndexError("pop on empty stack")
    popped = top(stack)
    stack.top = stack.top.next
    stack.size = stack.size - 1
    return popped

def is_empty(stack):
    """
    Is the stack empty?
    """
```

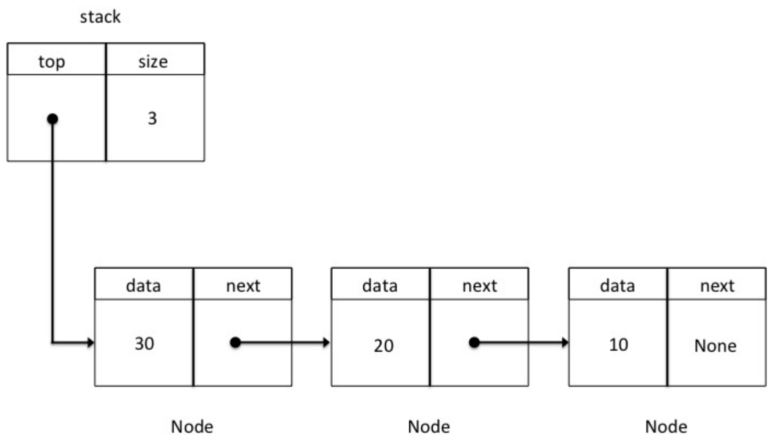
```
        return stack.top is None

    def size(stack):
        """
        Return the number of elements.
        """
        return stack.size
```

We can `push` the values 10, 20, and 30 onto the stack:

```
>>> from stack import *
>>> stack = make_empty_stack()
>>> stack
Stack(size=0, top=None)
>>> push(stack, 10)
>>> stack
Stack(size=1, top=FrozenNode(value=10, next=None))
>>> push(stack, 20)
>>> stack
Stack(size=2, top=FrozenNode(value=20, next=FrozenNode(value=10, next=None)))
>>> push(stack, 30)
>>> stack
Stack(size=3, top=FrozenNode(value=30, next=FrozenNode(value=20,
next=FrozenNode(value=10, next=None))))
```

Here is a box and arrow diagram of that stack.



Because the stack works exclusively with the `top` slot or the `size` slot, all of the operations have a time complexity of $O(1)$.

Take some time to look this over, as well as the test program, `test_stack.py`. **All the files are in a `.zip` file on the course website under the lecture link.**

3.3 Putting It All Together — The Cavern

Now we can implement the first stage of the simulation. The main program can be found in `BillyGoatsGruff/gruff.py`. It prompts for the number of goats and creates the cavern (a stack) by pushing each goat into it. In the end, the cavern, which is a stack, points to the top goat.

```
# create the goats and push them into the cavern
num_goats = int(input("How many goats? "))
cavern = make_empty_stack()
for id in range(1, num_goats+1):
    goat = make_goat(id)
    push(cavern, goat)
```

As the simulation runs, a loop executes so that each goat battles the troll. One at a time, goats are popped off the cavern stack. The goat does battle until either their hit points go to 0, or the troll's hit points go to 0. If the troll survives, the next goat at the front of the cavern is given a shot. This has two possible outcomes:

- The troll dies and the goat who was fighting survives. In this case, the surviving goat is pushed back onto the stack. We then move on to the next phase where all goats leave the cavern and attempt to cross the bridge.
- All the goats have died and the troll survives. In this case, the cavern is empty and the simulation is over.

The implementation of the cavern can be found in `BillyGoatsGruff/cavern.py`. It is implemented in the `survive_the_cavern` function. Here is a simplification that highlights the stack operations:

```
def survive_the_cavern(troll, cavern):
    # push the goats into the cavern
    while not is_empty(cavern) and troll.hit_points > 0:
        goat = pop(cavern)
        while troll.hit_points > 0 and goat.hit_points > 0:
            # goat strikes damage first
            dmg_to_troll = get_goat_damage()
            troll.hit_points -= dmg_to_troll
            if troll.hit_points > 0:
                dmg_to_goat = get_troll_damage()
                goat.hit_points -= dmg_to_goat

        if troll.hit_points <= 0:
            # put them back on the stack
            push(cavern, goat)
    # did anyone survive?
    print(str(size(cavern)), "goat/s survived the dark cavern...")
```

4 Queue Overview

If at least one goat survives the battle against the troll, the next stage of the simulation presents the goats with the bridge obstacle. The bridge has one entrance on the side of the gorge the goats are on, and one exit on the other side of the gorge where the delicious berries lie.

The bridge can be represented as a *queue*. A queue is a First In First Out (FIFO) or Last In Last Out (LILO) structure. The queue supports the following operations:

- `enqueue`: Insert an element onto the back of the queue.
- `dequeue`: Remove the front element from the queue and return the removed value.
- `front`: Peek at the front element in the queue without removing it.
- `back`: Peek at the back element in the queue without removing it.
- `is_empty`: Return `True` if the queue is empty and `False` otherwise.
- `size`: Return the number of elements that are in the queue.

4.1 Singly Linked Node: Mutable

Since we are not using a list to store the goats, we need to come up with a way to store them ourselves. We will again use a *singly linked node* based representation. Each node in the queue will hold a goat, as well as a reference to the next goat. The queue will hold references to the goat at the front and back of the queue.

We will implement our node in the file `node_types.py` and use the `@dataclass` annotation:

```
@dataclass( frozen=False )
class MutableNode:
    """
    A mutable link node containing a value and a link to the next node
    """
    value: Any
    next: Union[ "MutableNode" , None ]
```

For reasons that you will see when we get into the queue implementation, the nodes of a queue do need to change. We set `frozen=False` in the decorator to indicate the fields of the node are mutable and can be changed after creation.

When importing, the queue will do the following import and refer to `MutableNode` as `Node`.

```
from node_types import MutableNode as Node
```

4.2 Queue Implementation

Like the stack, we will reuse the mutable linked sequence for the representation. Unlike the stack, the queue will require more subtle connections and surgery to achieve the performance we want. If we do nothing, then we would have to traverse the entire sequence to reach the end and enqueue a new element, producing a new sequence. If we keep track of the last element however, we can quickly add the new element. This is efficient although it changes the existing sequence.

That means our implementation has to maintain two pointers. The first pointer is called `front`, referring to the front of the queue (the exit of the bridge). The second pointer is called `back`, referring to the back of the queue (the entrance to the bridge).

We will create a structure that encapsulates both pointers, as well as the size of the queue. Here is the structure definition, `Queue`, and the function to create an initially empty queue.

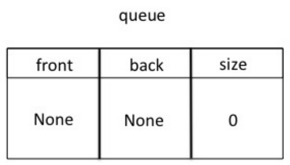
```
@dataclass(frozen = True)
class Queue:
    size: int
    front: Union[None, Node]
    back: Union[None, Node]

def make_empty_queue():
    """
    Returns a new queue with size initialized to zero and
    front & back initialized to be an empty queue.
    """
    return Queue(0, None, None)
```

When created:

```
    queue = make_empty_queue()
```

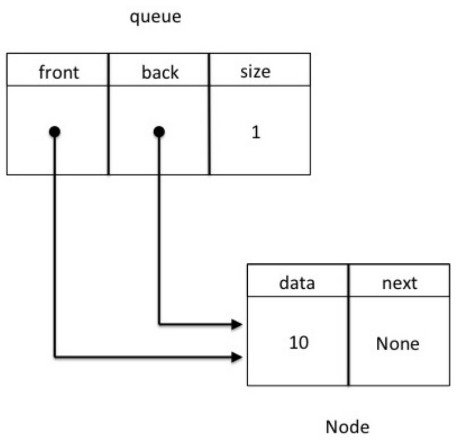
It looks like:



The check for `is_empty` is similar to the stack. We can either check whether the front or back pointer refer to `None` (the queue is empty), or not (the queue is not empty).

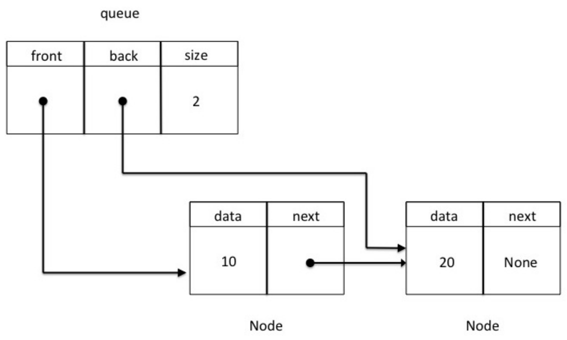
There are two conditions that we have to deal with when enqueueing a new element into a queue. The first condition is to handle what happens when the queue is initially empty. In this scenario, both the front and back pointer need to be updated to point to the new node. The new node's next should point to `None`:

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



In the second condition, the queue is not empty, and 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`). But first, the next slot of the old back node should be set to point to the new node.

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



Notice how in both cases, an enqueue increases the size by one. This addition is similar to `push` for the stack.

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

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

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

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

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

There is one special condition. If we dequeue the last element in the queue, the front pointer will correctly go to `None`. However, the back pointer will still point to that node. In this case, we must also change the back pointer to point to the `None`.

```
queue.back = None
```

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

This is the resulting queue implementation using a linked node sequence. It is located in the `Queue` subdirectory.

```
def enqueue(queue, element):
    """
    Insert an element into the back of the queue. (Returns None).
    """
    newnode = Node(element, None)
    if is_empty(queue):
        queue.front = newnode
    else:
        queue.back.next = newnode
    queue.back = newnode
    queue.size = queue.size + 1

def dequeue(queue):
    """
    Remove the front element from the queue. (Returns the removed value)
    """
    if is_empty(queue):
        raise IndexError("dequeue on empty queue")
    removed = queue.front.value
    queue.front = queue.front.next
    if is_empty(queue):
        queue.back = None
    queue.size = queue.size - 1
    return removed

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

def back(queue):
    """
    Access and return the last element in the queue without removing it.
```

```
        """
        if is_empty(queue):
            raise IndexError("back on empty queue")
        return queue.back.value

    def is_empty(queue):
        """
        Is the queue empty?
        """
        return queue.front is None
```

All of the queue operations have a time complexity of $O(1)$ because we added the `back` slot; there are no loops or recursive functions.

4.3 Putting It All Together — The Bridge

Now we can represent the bridge as a queue. This is implemented in the `BillyGoatsGruff/bridge.py` source file. There is a function, `crossTheBridge` that takes the surviving goats 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 800-1000. Second, the bridge can hold a random number of goats between 3 and 5.

Next, we enter a loop where each goat 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 can enter the bridge. Goats on the bridge fall to their demise, and the remaining goats from the cavern become stranded. In the other condition, the bridge does not break and the cavern becomes empty. When the bridge becomes full, the first goat to have entered the bridge leaves the bridge, and reaches the meadow with the delicious berries.

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

```
    MAX_WEIGHT = randint(700, 1100)
    MAX_GOATS = randint(3, 5)

    def cross_the_bridge(cavern, meadow):
        """
        Goats leave the cavern and enter the bridge.
        """

        bridge = make_empty_queue()

        # take goats one at a time from the cavern and enqueue
        # them onto the bridge, until either it breaks or they
        # all make it on
        total_weight = 0
```

```
broken_bridge = False
while not broken_bridge and not stack.is_empty(cavern):
    # if the bridge holds the max goats, remove the front goat from it
    if bridge.size == MAX_GOATS:
        goat = front(bridge)
        enqueue(meadow, front(bridge))
        dequeue(bridge)
        total_weight -= goat.weight

    # process the next goat in the cavern
    goat = pop(cavern)
    enqueue(bridge, goat)
    total_weight += goat.weight
    if total_weight > MAX_WEIGHT:
        broken_bridge = True

# if the bridge breaks, separate those who fall from those
# who are stranded
if broke_bridge:
    while not cs_queue.is_empty(bridge):
        # the goats on the bridge are in trouble
        dequeue(bridge)
    while not stack.is_empty(cavern):
        goat = top(cavern)
        pop(cavern)
else:
    while not cs_queue.is_empty(bridge):
        goat = front(bridge)
        enqueue(meadow, front(bridge))
        dequeue(bridge)

print(str(goats.size), "goat/s survived the bridge crossing...")
```

5 Complexity: Running time analysis

Because the stack works exclusively with the `top` slot or the `size` slot, all of the operations have a time complexity of $O(1)$.

All of the queue operations have a time complexity of $O(1)$ because the `back` slot was 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 goats).

6 Testing

The program uses randomization so we may have to run it a few times.

- The troll wins. This is easy to demonstrate. Run the program with a small number of goats (e.g. 5).
- No one wins. This is also easy to demonstrate. Run the program with a large number of goats (e.g. 60).
- Some goats win. Play with this one. It usually works within the range of 20-30 goats.