The queue class

MAY 4TH, 2015

Remember searchWordNetwork

```
def searchWordNetwork(source, target, D):

    processed = {source:0}
    reached = {}
    for e in D[source]:
        reached[e] = source # the value in the dictionary of a key k is the "parent" of k

    # Repeat until reached set becomes empty or target is reached
    while reached:
        # Check if target is in reached; this would imply there is path from source to target
        if target in reached:
            processed.update({target:reached[target]})
            return processed

        # Pick an item in reached and process it
        item = reached.popitem() # returns an arbitrary key-value pair as a tuple
        newWord = item[0]
        parent = item[1]

        # Find all neighbors of this item and add new neighbors to reached
        processed[newWord] = parent
        for neighbor in D[newWord]:
            if neighbor not in reached and neighbor not in processed:
                reached[neighbor] =  newWord

    return {}
```

The fact that popItem returns an arbitrary node makes this function return arbitrarily long paths from source to target.

How to get shortest paths?

- If we pull out the “oldest” item from reached, we will be guaranteed to get a shortest path.

- Nodes are inserted into reached in some order – the order in which they are reached by the exploration algorithm. So we have a notion of how long each item has been in reached.

- The network exploration algorithm with this feature is called *breadth-first search*.

A new data structure

- So we need a data structure that maintains a collection of items and supports the following operations:
  - enqueue: inserts the given item into the data structure
  - dequeue: deletes from the data structure the element that was inserted earliest and returns this element.

Example:

```
>>> Q = queue()
>>> Q.enqueue(10)
>>> Q.enqueue(20)
>>> Q.enqueue(11)
>>> Q.dequeue()
10
>>> Q.enqueue(10)
>>> Q.dequeue()
20
```

“FIFO” data structure

- This is called a *First-in First-out (FIFO)* data structure. Also called a *queue* data structure.

- How to implement this data structure?

- We’ll discuss a *list-based* implementation and a *dictionary-based* implementation.

- **GOAL:** To ensure that both operations (enqueue and dequeue) run in *constant* number of rounds, independent of the length of the queue.

List-based implementation

- Idea:
  - When a new element arrives, append it to the (back of the) list
  - This means that the oldest elements are at the front and newest elements at the back.
  - So we delete (dequeue) elements from the front

Implementation

```
class queue():

    # Constructs an empty queue
    def __init__(self):
        self.L = []

    # Enqueue appends items at back of list
    def enqueue(self, item):
        self.L.append(item)

    # Dequeue removes items from front of list. This method is not efficient
    def dequeue(self):
        item = self.L.pop(0)
        return item

    # Shows the queue as a list
    def __repr__(self):
        return str(self.L)
```

A more efficient list-based implenentation

- Let us keep an index called `start` that will always point to the first (earliest) element in the list.
- So we do not explicitly remove elements from the list in response to `dequeue`; instead we simply move `start`.
- Now both enqueue and dequeue are quite efficient.

Implementation

```
class queue():

    # Constructs an empty queue
    def __init__(self):
        self.L = []
        self.start = -1 # initialize start to point to before the first valid index

    # Enqueue appends items at back of list
    def enqueue(self, item):
        self.L.append(item)
        # If the queue was empty prior to this insertion, update start
        if self.start == -1:
            self.start = self.start + 1

    # Dequeue removes items from front of list. This method is not efficient
    def dequeue(self):
        self.start = self.start + 1
        item = self.L[self.start - 1]
        return item

    # Shows the queue as a list
    def __repr__(self):
        return str(self.L[self.start:])

    # Queue is empty is there if the list if physically empty
    # or start points to the end of the list
    def isEmpty(self):
        return len(self.L) == 0 or self.start == len(self.L)
```

But wait…

- …even this implementation has a problem.
- We may have a very large self.L even though the queue may have very few elements.
- Thus we have traded off space (memore) for time (speed).