Math Sec 2.2

Rex McArthur
Math 320

September 28, 2015

Exercise. 2.8

```
import time
start = time.time()
A = []
for i in xrange(10000):
    A.append(i)
print time.time() - start
#Returns .0012832
start = time.time()
B = []
for i in xrange(10000):
    B.insert(0,i)
print time.time() - start
#Returns .02349239
```

Thus, as we would expect, appending at the end is quicker, because one does not have to copy the list over when you hit the predetermined maximum allowed.

Exercise. 2.9

```
Given the maximum size is n,
Class stack():
    Intialize an empty array of length n
    and a an element value  = 0
def push(data):
    if element value = n, Return error, array full
    else: put data at the element value index of array
    element value += 1
def pull():
    return the element value index element, and -1 from element value
```

Both of these have best case scenario of O(1), and worst case O(1)
If you don't know the max length of the list, you would need to copy over the whole array everysingle time you reached a certain length to a larger lengthed array. This

canges the best case of push to still O(n), and worst case is O(n). On the other hand, pull will always pull at O(1).

Exercise. 2.10

```
Given the maximum size is n,
Class stack():
    Intialize an empty array of size n
    and length = 0 attribute
def enque(data):
    Put data at length index
    length += 1
def deque():
    Take element at the 1st element,
    Push everythin up 1 index
    length -= 1
    return taken first element
```

Enque and has best and worse case scenario of O(1)
If you don't know what the max of the list will be, you will have to add elements at the end of the list once you hit the max. This only affects the deque function, making the worst case scenario O(n).

Exercise. 2.11
You would want to use a deque, because you can pull from the front and back.

```
def Palindrome_finder(word):
    i = 0
    while len(word)<2:
        if word[i] != word [-i]
            return False
        else:
            delete(word(end))
            delete(word(begining))
    return True
```

This gives it a temporal complexity of O(.5n). If you used a stack, it would make the temporal complexity O(2n), because you would have to make a copy of the stack, invert it one element at a time, then compare each element as you went along, until one didn't match, or the end was reached. Linked Lists would be dreadful to do this. Deque is the best option.