Project 1 Array-based lists

CECS 274 Data Structures Fall 2020

Deadline: 11:55pm, September 20 (Sunday), 2020

Note: the sample codes are given in Python format, you can flexibly choose other languages like Java, etc.

1. (5pt) In the following sample code of figure 1.1, we perform an experiment to compare the length of a Python list to its underlying memory usage. Determining the sequence of array sizes requires a manual inspection of the output of that program. Redesign the experiment so that the program outputs only those values of k at which the existing capacity is exhausted. For example, on a system consistent with the results of Figure 1.2, your program should output that the sequence of array capacities are 0, 4, 8, 16, ... Name your code to a file as *1_1_length.py*.

```
1 import sys                                  # provides getsizeof function
2 data = [ ]
3 for k in range(n):                          # NOTE: must fix choice of n
4   a = len(data)                             # number of elements
5   b = sys.getsizeof(data)                   # actual size in bytes
6   print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
7   data.append(None)                         # increase length by one
```

Figure 1.1

```
Length:   0; Size in bytes:   72
Length:   1; Size in bytes:  104
Length:   2; Size in bytes:  104
Length:   3; Size in bytes:  104
Length:   4; Size in bytes:  104
Length:   5; Size in bytes:  136
Length:   6; Size in bytes:  136
Length:   7; Size in bytes:  136
Length:   8; Size in bytes:  136
Length:   9; Size in bytes:  200
Length:  10; Size in bytes:  200
Length:  11; Size in bytes:  200
Length:  12; Size in bytes:  200
Length:  13; Size in bytes:  200
Length:  14; Size in bytes:  200
Length:  15; Size in bytes:  200
Length:  16; Size in bytes:  200
Length:  17; Size in bytes:  272
Length:  18; Size in bytes:  272
```

Figure 1.2

2. (10pt) Design an implementation of the DynamicArray. The general implementation of insert for the DynamicArray, as given in the Figure 1.3, has the following inefficiency: in the case when a resize occurs, the resize operation takes time to copy all the elements from an old array to a new array, and then the subsequent loop in the body of insert shifts many of those elements. Give an improved implementation of the insert method, so that, in the case of a

resize, the elements are shifted into their final position during that operation, thereby avoiding the subsequent shifting. Name your code file to *1_2_dynamicArray.py* (or .java, .etc).

```
def insert(self, k, value):
  """Insert value at index k, shifting subsequent values rightward."""
  # (for simplicity, we assume 0 <= k <= n in this verion)
  if self._n == self._capacity:                # not enough room
    self._resize(2 * self._capacity)           # so double capacity
  for j in range(self._n, k, -1):              # shift rightmost first
    self._A[j] = self._A[j-1]
  self._A[k] = value                           # store newest element
  self._n += 1
```

Figure 1.3

3.(15pt) Design a code for array-based implementation of a FIFO queue that uses modular arithmetic. Uses a doubling strategy for resizing when it becomes full. You can use the given sample "*1_3_array_queue.py*" in *sample_project_1* folder and test with the test code. (you can also write your own code with same name).

4.(10pt) Design the code for an array-based list implementation with O(1) amortized update time. Stores the list in an array, *a*, so that the *i*'th list item is stored at *a[(j+i)%len(a)]*. The amortized cost of resizing *a* should be O(1). You can use the given sample "*1_4_array_stack.py*" in *sample_project_1* folder and test with the test code. (you can also write your own code with same name).

5.(10pt) Implement the queue using two stacks as instance variables, such that all queue operations execute in amortized O(1) time. Give a proof of the amortized bound at the beginning comment before the code. Please use the name "*1_5_queue.py*" in the submission (or use .java, etc.).

Submission Requirements

You need to strictly follow the instructions listed below:

1) Submit a .zip/.rar file that contains all files.

2) The submission should include your **source code**. Do not submit your binary code.
3) Your code must **be able to compile**; otherwise, you will receive a grade of zero.
4) Your code should not produce anything else other than the required information in the output file.
6) If you code is partially completed, also explain in the begin comment of specific submission what has been completed and the status of the missing parts.
7) Provide **sufficient comments** in your code to help the TA understand your code. This is important for you to get at least partial credit in case your submitted code does not work properly.

Grading criteria:

Details	Points
Submission follows the right formats	10 %
Have a README file shows how to compile and test your submission	15 %
Submitted code has proper comments to show the design details	15 %
Code can be compiled and shows right outputs	60 %

Rubrics for code outputs

	Level 4 (100%)	Level 3 (70%)	Level 2 (40%)	Level 1 (20%)
Code outputs	It is always correct without crashes	It is always correct and eventually it crashes	It is not always correct and eventually it crashes	It is not correct or incomplete