# CS 331 Spring 2019

# Midterm Exam 2

**Instructions:**

- This exam is closed-book, closed-notes. Calculators are not permitted.
- For numbered, multiple-choice questions, fill your answer in the corresponding row on the "bubble" sheet.
- For problems that require a written solution (labeled with the prefix "WP"), write your answer in the space provided on the written solution sheet. Please write legibly and clearly indicate your final answer.
- Turn in the exam question packet, bubble sheet, and written solution sheet separately.
- Good luck!

## Concepts (24 points):

1. What is the run-time complexity of inserting a new element at the beginning of a circular, doubly-linked list with a sentinel head?

   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N^2)$

2. What is the run-time complexity of locating and deleting a specified key from a hashtable containing $N$ key/value pairs?

   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N^2)$

3. What is the run-time complexity of increasing the number of buckets and *rehashing* all elements into the new buckets in a hashtable, given $N$ key/value pairs?

   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N^2)$

4. Given a circular, doubly-linked list whose contents are sorted in ascending order, what is the run-time complexity for inserting a new element into the list so that it remains correctly sorted? (Including the time required to search for the element's correct position.)

   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N^2)$

5. Which implementation of the list ADT is best suited to an application where the most common operations are to insert and remove values from the beginning and end of the list?

   (a) built-in Python list
   (b) array-backed list
   (c) singly-linked list
   (d) doubly-linked list

6. Which of the following best describes the behavior of the Stack ADT?

   (a) first-in, first-out
   (b) last-in, first-out
   (c) first-in, second-out
   (d) last-in, last-out

7. Which of the following best describes the behavior of the Queue ADT?

   (a) first-in, last-out
   (b) last-in, first-out
   (c) first-in, second-out
   (d) last-in, last-out

8. If we assume uniform hashing, what is the probability that a collision will occur in a hashtable with 100 buckets and 2 keys?

   (a) $\frac{2}{100}$
   (b) $1 - \frac{2}{100}$
   (c) $1 - \frac{99}{100}$
   (d) $1 - \frac{99}{100} \times \frac{98}{100}$

9. What are the contents of the list `l` after executing the following code?

```
s = Stack()
q = Queue()
l = []

for x in range(10):
    s.push(x)
while s:
    q.enqueue(s.pop())
while q:
    l.append(q.dequeue())
```

   (a) `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`
   (b) `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
   (c) `[1, 0, 3, 2, 5, 4, 7, 6, 9, 8]`
   (d) `[]`

10. Which correctly *prepends* `value` to a circular, doubly-linked list with a sentinel head?

    (a)
    ```
    n = LinkedList.Node(value, prior=self.head.prior, next=self.head.next)
    n.prior = n.prior = n
    ```

    (b)
    ```
    n = LinkedList.Node(value, prior=self.head, next=self.head.next)
    n.prior.next = n.next.prior = n
    ```

    (c)
    ```
    n = LinkedList.Node(value, prior=self.head, next=self.head.prior)
    n.next, n.prior = n, n.next.prior
    ```

    (d)
    ```
    n = LinkedList.Node(value, prior=self.head.next, next=self.head.next.next)
    n.next.prior = n.prior.next = n
    ```

11. If `to_del` refers to a node in a circular, doubly-linked list with a sentinel head, which of the following correctly removes the node from the list?

    (a)
    ```
    to_del.next = to_del.prior
    to_del.prior = to_del.next
    ```

    (b)
    ```
    to_del.prior.next = to_del.next
    to_del.next.prior = to_del.prior
    ```

    (c)
    ```
    to_del.prior.prior = to_del.next.next
    to_del.next.next = to_del.prior.prior
    ```

    (d)
    ```
    to_del.next, to_del.prior = to_del.prior, to_del.next
    ```

12. Which correctly computes the index for the provided `key`'s bucket in a hashtable implementation?

    (a) `bucket_idx = hash(key) % len(self.buckets)`

    (b) `bucket_idx = hash(key) // len(self.buckets)`

    (c) `bucket_idx = hash(key + value) % len(self.buckets)`

    (d) `bucket_idx = hash(key) % len(self)`

## Linked Lists (8 points):

**WP1** Implement the `merge` method for the LinkedList data structure (implemented as a circular, doubly-linked list with a sentinel head). `merge` will be called with a list argument (also a LinkedList), whose contents will be merged in after the existing elements. This operation will be carried out in O(1) time, without creating any additional nodes and only by adjusting links. After the operation, the argument list can no longer be used reliably. Sample usage below:

```
l1 = LinkedList()
for x in range(5):
    l1.append(x)

# l1 contains [0, 1, 2, 3, 4]

l2 = LinkedList()
for x in range(5,10):
    l2.append(x)

# l2 contains [5, 6, 7, 8, 9]

l1.merge(l2)

# l1 contains [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Criteria:

- You should *not* call any other LinkedList methods.
- You should *not* change the value contained in any of the nodes nor create any new nodes. Instead, your implementation should work by re-linking existing nodes.

## Queues (8 points):

**WP2** Implement the `promote` method for the singly-linked Queue data structure, which takes an integer and moves the element at that position in the queue (where the head is at position 0) one position closer to the head. Sample usage below — the queue contents are shown in list form, with the head on the left and the tail on the right:

```
q = Queue()
for x in 'abcdefg'
    q.enqueue(x)

# q contains [a, b, c, d, e, f, g]

q.promote(1)

# q contains [b, a, c, d, e, f, g]

q.promote(3)

# q contains [b, a, d, c, e, f, g]
```

Criteria:

- You should *not* call any other Queue methods.
- You should *not* change the value contained in any of the nodes nor create any new nodes. Instead, your implementation should work by re-linking existing nodes.

## Hashtables (8 points):

**WP3** Re-implement the `__setitem__` and `__getitem__` methods of the Hashtable data structure so that it keeps track of *all* past values associated with a given key, which can then be accessed – in the order they were added – using the iterator returned by `__getitem__`. Below is a sample interaction with the updated Hashtable:

```
ht = Hashtable()

ht['a'] = 'apple'
ht['b'] = 'banana'
ht['a'] = 'avocado'
ht['b'] = 'bamboo'
ht['c'] = 'cabbage'
ht['a'] = 'artichoke'

for v in ht['a']: # prints 'apple', 'avocado', 'artichoke' (in that order)
  print(v)

for v in ht['c']: # prints 'cabbage'
  print(v)
```

Criteria:

- Note that the implementation you are updating is based on the basic Hashtable presented in class, not the sorted Hashtable you built in lab — for this problem you do not need to guarantee that the ordering of keys is preserved!