# CS 331 Fall 2017

# 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. Which data structure would you choose to help maintain multiple key/value associations, and where a full-ordering of keys or values is not important?

   (a) a list
   (b) a stack
   (c) a queue
   (d) a hashtable
   (e) a heap

2. Which data structure would you choose to help track a collection of objects where only the object with the largest value needs to be readily accessible?

   (a) a list
   (b) a stack
   (c) a queue
   (d) a hashtable
   (e) a heap

3. Which data structure would you choose to help track a collection of objects where only the object that has been tracked the longest needs to be readily accessible?

   (a) a list
   (b) a stack
   (c) a queue
   (d) a hashtable
   (e) a heap

4. What is the run-time complexity of locating and retrieving the element in the middle (by index) of a circular, doubly-linked list with a sentinel head?

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

5. What is the run-time complexity of determining whether a specified key exists in a hashtable containing $N$ key/value pairs?

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

6. What is the run-time complexity of locating (but not removing) the maximum element in a max-heap of $N$ elements?

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

7. What is the run-time complexity of inserting a new element into a max-heap of $N$ elements and ensuring the heap property is maintained?

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

8. A student has proposed the following method as a faster way of counting the number of elements in a doubly-linked list with a sentinel head node.

```
def fast_count(self):
    n = 0
    h = self.head.next
    while h is not self.head:
        if h.next is not self.head:
            n, h = n+2, h.next.next
        else:
            n, h = n+1, h.next
    return n
```

   What is the time complexity of `fast_count` when run on a list with $N$ elements?

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

9. In what scenario will constructing a max-heap from $N$ elements (added one at a time) be carried out most efficiently?

   (a) when the elements are added in ascending order
   (b) when the elements are added in descending order
   (c) when the elements are added in random order
   (d) when a separate min-heap is used to simultaneously keep track of the smallest element
   (e) all the above yield the same runtime efficiency

10. For this and the next problem, consider the following definition of a circular, doubly-linked list with a sentinel head, in which we have defined a `set_cursor` method, which sets the `cursor` attribute to a specified index in the linked list where subsequent operations can be performed.

```
class LinkedList:
  def set_cursor(self, idx):
      assert(idx < len(self))
      self.cursor = self.head.next
      for _ in range(idx):
          self.cursor = self.cursor.next

  def cursor_insert(self, val):
      n = LinkedList.Node(val, prior=self.cursor.prior, next=self.cursor)
      ------------------------------------------------
      self.length += 1

  def cursor_delete(self):
      ------------------------------------------------
      ------------------------------------------------
      self.cursor = self.cursor.next
      self.length -= 1
```

`cursor_insert` should insert the given value `val` at the current cursor position, and `cursor_delete` should delete the value at the current cursor position. In both cases, the cursor should refer to the same index as it did before the operation (except in the case of deletion of the last node, which will invalidate the cursor).

Which correctly completes the implementation of `cursor_insert`?

(a) `self.cursor.prior.next = self.cursor.next.prior = n`

(b) `self.cursor = self.cursor.next = self.cursor.prior = n`

(c) `self.cursor = self.cursor.prior = self.cursor.next.prior = n`

(d) `self.cursor.next.prior = self.cursor.prior = self.cursor = n`

(e) `self.cursor.prior.next = self.cursor.prior = self.cursor = n`

11. Which correctly completes the implementation of `cursor_delete`?

(a)
```
self.cursor.prior = self.cursor.next
self.cursor.next = self.cursor.prior
```

(b)
```
self.cursor = self.cursor.next
self.cursor.prior = self.cursor
```

(c)
```
self.cursor.prior.next = self.cursor.next.prior
self.cursor.next.prior = self.cursor.prior.next
```

(d)
```
self.cursor.prior.next = self.cursor.next
self.cursor.next.prior = self.cursor.prior
```

(e)
```
self.cursor.next = self.cursor.next.prior
self.cursor.prior = self.cursor.prior.next
```

12. Consider the following definition of a hashtable with a partially implemented `rehash` method, which will either grow or shrink the `buckets` array to the provided value `n_buckets` (while keeping all existing key/value mappings).

```
class Hashtable:
  def rehash(self, n_buckets):
      new_buckets = [None] * n_buckets
      for b in self.buckets:
          while b:
              b_next = b.next
              --------------------------------
              --------------------------------
              --------------------------------
              b = b_next
      self.buckets = new_buckets
```

Which correctly completes the `rehash` method?

(a)
```
idx = hash(b.key) % n_buckets
b.next = new_buckets[idx]
new_buckets[idx] = b
```

(b)
```
idx = hash(b.key) % len(self.buckets)
b.next = new_buckets[idx]
self.buckets[idx] = b
```

(c)
```
idx = hash(b.key) % len(self.buckets)
self.buckets[idx] = new_buckets[idx]
b.next = new_buckets[idx]
```

(d)
```
idx = hash(b.key) % n_buckets
new_buckets[idx].next = b
b.next = new_buckets[idx]
```

(e)
```
idx = hash(b.key) % n_buckets
b.next = new_buckets[idx].next
new_buckets[idx].next = b
```

## Stacks (6 points):

Consider the following linked implementation of the Stack ADT:

```
class Stack:
    class Node:
        def __init__(self, val, next):
            self.val = val
            self.next = next

    def __init__(self):
        self.top = None

    def push(self, val):
        self.top = Stack.Node(val, self.top)

    def pop(self):
        val = self.top.val
        self.top = self.top.next
        return val
```

**WP1** Implement the method `roll` as part of the linked stack data structure. With an argument $n \geq 2$, the method moves the top of the stack (which is in position 1) to the $n^{th}$ position, shifting all intervening elements towards the top.

Examples:

- invoking `roll(3)` on a stack containing the values `A, B, C, D, E` (where `A` is the top-most value) would result in the stack `B, C, A, D, E`
- invoking `roll(5)` on a stack containing the values `A, B, C, D, E` would result in the stack `B, C, D, E, A`

Restrictions/Assumptions:

- You should *not* create or use any data structures.
- You should *not* change the value contained in any of the nodes nor create any new nodes. Instead, your implementation should only re-link existing nodes.
- You may assume the argument, $n$, to `roll` is at least 2, and that the stack contains at least $n$ values.

## Linked Lists (6 points):

**WP2** For this problem, you are to implement the `guided_iter` method of a circular, doubly-linked list with a sentinel head. When called with an array of integers (called "`steps`"), `guided_iter` will return an iterator over elements in the list reached by traversing (backwards or forwards) over the number of elements specified by each integer, starting at index 0.

E.g., given the list `l = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']`,

- `l.guided_iter([3, 1, 2])` will return an iterator over `['d', 'e', 'g']`
- `l.guided_iter([0, -2, -4])` will return an iterator over `['a', 'i', 'e']`
- `l.guided_iter([4, -1, 0, 8, -5])` will return an iterator over `['e', 'd', 'd', 'b', 'g']`

Note that negative step values indicate backwards traversal and positive step values indicate forward traversal — large values may cause multiple "loops" to be taken around the elements in a list (ignoring, of course, the sentinel head).

Restrictions/Assumptions:

- Your implementation should *not* make use of any other list methods (e.g., `__getitem__`).
- You may assume that the list contains at least one element.
- You may assume the input steps are all integers.

## Heaps (8 points):

**WP3 (a)** Consider the following ordered sequence of values to be added to a max-heap:

4, 8, 6, 5, 7, 9

Sketch the heap after adding each value and re-heapifying.

**WP3 (b)** Consider the following implementation of `third_largest`, which is intended to return the third largest value from a max-heap (assuming the heap contains at least three values):

```
class Heap:
    def __init__(self):
        self.data = []

    def third_largest(self):
        if self.data[1] > self.data[2]:
            return self.data[2]
        else:
            return self.data[1]
```

The method does not work reliably, however. Sketch a valid max-heap for which the above implementation fails.