# CS 331 Fall 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 primary reason we incorporated a sentinel node in our circular, doubly-linked list implementation?

   (a) to make it easier to locate the first element of the list

   (b) to reduce the runtime complexity of the `__getitem__` method (i.e., indexing)

   (c) to make it possible to remove elements in $O(1)$ time

   (d) to reduce the number of edge cases we have to consider in our implementation

2. What is the worst-case runtime complexity of removing the first element (by index) in a circular, doubly-linked list of $N$ elements?

   (a) $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

   (d) $O(N \log N)$

3. Which of the following operations has slower worst-case runtime complexity in a linked-list, compared to an array-backed list?

   (a) locating an item by index (i.e., `__getitem__`)

   (b) removing the first element (i.e., index 0)

   (c) removing the last element (i.e., index -1)

   (d) all the above operations have the same runtime complexity across the two implementations

4. Assuming uniform hashing, What is the probability of there being *no* collisions in a hashtable with 1000 buckets and 5 total entries?

   (a) $1 - \frac{999}{1000} \times \frac{998}{1000} \times \frac{997}{1000} \times \frac{996}{1000}$

   (b) $\frac{999}{1000} \times \frac{998}{1000} \times \frac{997}{1000} \times \frac{996}{1000}$

   (c) $\frac{995}{1000}$

   (d) $\frac{5}{1000}$

5. What is the worst-case runtime complexity of determining whether a specified key exists in a hashtable containing $N$ key/value entries?

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

6. To improve the performance of a hashtable, we could replace the linked list used to track collisions within a given bucket with another nested hashtable, so that all keys that fall into the same bucket are hashed yet again onto another set of buckets (which, this time, use linked lists for collisions). What would be the worst-case runtime complexity for inserting a key/value pair into a hashtable with this design containing $N$ entries?

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

7. Which of the following data structures maintains entries in first-in-first-out (FIFO) order?

   (a) stack
   (b) queue
   (c) priority queue
   (d) heap

8. In a queue implemented with a circular array of size $N$, where `head` and `tail` are associated with the indices of the next element to dequeue and the most recently enqueued element, respectively, which of the following expressions evaluates to `True` if and only if the queue is full?

   (a) `(tail + 1) % N == head`
   (b) `(head + 1) % N == tail`
   (c) `tail % N == head - 1`
   (d) `(head + N) % N  == head`

9. In a singly-linked queue implementation, which of the following correctly implements the `enqueue` operation when the queue is *not empty*?

   (a) `self.tail.next = Queue.Node(val, self.tail)`

   (b) `self.tail = Queue.Node(val, self.tail)`

   (c) `self.tail = self.tail.next = Queue.Node(val)`

   (d) `self.tail.next = self.tail = Queue.Node(val)`

10. Given an element with index `i` in the array-representation of a heap, which of the following correctly computes the index of the *left child* of that element?

    (a) `(i + 1) // 2`

    (b) `2*i - 1`

    (c) `(i + 1) % 2`

    (d) `2*i + 1`

11. What is the worst-case runtime complexity of adding an element to an array-backed max-heap of $N$ elements and restoring the heap property?

    (a) $O(1)$

    (b) $O(\log N)$

    (c) $O(N)$

    (d) $O(N \log N)$

12. At most how many swaps would be needed to "re-heapify" a max-heap of 100 elements after removing the maximum value?

    (a) 3

    (b) 7

    (c) 12

    (d) 18

## Linked Stack (8 points):

**WP1** For this problem you are to implement the `remove_all` method for the singly-linked stack implementation which, when called with some value `val`, will remove all instances of `val` from the stack. The top of the stack will be adjusted, if necessary.

E.g., calling `remove_all(9)` from a stack with contents `9, 1, 9, 2, 3, 9, 4, 9, 9` (the top of the stack is the leftmost value shown) will result in the stack with contents `1, 2, 3, 4`. `remove_all` should work correctly if the stack is empty, and may also leave the stack empty.

Criteria:

- You should *not* call any other Stack methods
- You should *not* modify or add attributes to the `Stack` or `Node` classes
- You should *not* use any other data structures in your implementation (e.g., lists, dictionaries, etc.)

The base linked stack implementation is provided on the solution sheet for reference.

## Hashtable (8 points):

**WP2** For this problem you are to implement the `collision_ratio` hashtable method, which returns a number $n$ in the range $0 \leq n \leq 1$, computed as the fraction $\frac{c}{k}$, where $c$ is the number of buckets in the hashtable containing a collision, and $k$ is the number of buckets in the hashtable with at least one key/value entry.

E.g., in a hashtable with a total of 1000 buckets, where 50 buckets contain at least one key/value entry, and of which 20 buckets contain two or more entries (i.e., a collision), the method `collision_ratio` should return $\frac{20}{50} = 0.4$.

Criteria:

- You should *not* call any other Hashtable methods
- You should *not* modify or add attributes to the `Hashtable` or `Node` classes
- You should *not* use any other data structures in your implementation (e.g., lists, dictionaries, etc.)

The base hashtable implementation is provided on the solution sheet for reference.

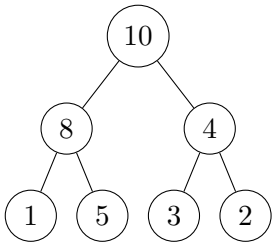
## Heap (8 points):

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

4, 6, 8, 3, 5, 7, 9

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

**WP3 (b)** Consider the following max-heap:




Sketch the heap after removing the maximum value and re-heapifying until the heap is empty. (I.e., show the updated heap after each value is removed.)