CS 331 Midterm Exam 2

Please bubble your answers in on the provided answer sheet. Also be sure to write and bubble in your student ID number (without the leading 'A').

1. What is the worst-case runtime complexity for enqueuing N elements onto an empty singly-linked queue implementation (as given in class), then dequeuing until empty?

   (a) O(1)

   (b) O(log N)

   (c) O(N)

   (d) O(N log N)

2. What is the worst-case runtime complexity for removing the first element (at index 0) from a doubly-linked list containing N elements?

   (a) O(1)

   (b) O(log N)

   (c) O(N)

   (d) O(N log N)

3. What is the worst-case runtime complexity for determining whether a hashtable of N elements contains a given key?

   (a) O(1)

   (b) O(log N)

   (c) O(N)

   (d) O(N log N)

4. What is the worst case runtime complexity of removing a key from a hashtable of N elements, given that the bucket index for said key has already been computed?

   (a) O(1)

   (b) O(log N)

   (c) O(N)

   (d) O(N log N)

5. What is the worst case runtime complexity for a function that efficiently determines the *second-largest* element in a max-heap of N elements?

   (a) O(1)

   (b) O(log N)

   (c) O(N)

   (d) O(N log N)

6. Assuming that p and q refer to adjacent nodes (with p preceding q) within a circular doubly-linked list, which of the following swaps the two?

   (a)
   ```
   p.next, p.prior, q.next, q.prior = q.next, q.prior, p.next, p.prior
   ```

   (b)
   ```
   p, q = q, p
   p.next, p.prior = q.next, q.prior
   ```

   (c)
   ```
   p.prior.next, q.next.prior = p.prior, q.next
   p.next.prior, q.prior.next = p.next, q.prior
   ```

   (d)
   ```
   p.prior.next, q.next.prior = q, p
   p.next, p.prior, q.next, q.prior = q.next, q, p, p.prior
   ```

7. Which of the following implements a generator-based iterator for a circular doubly-linked list, where `self.head` refers to the sentinel head?

   (a)
   ```
   for i in len(self.head):
       yield self.head.val[i]
   ```

   (b)
   ```
   n = self.head.next
   while n:
       yield n.next.val
   ```

   (c)
   ```
   yield from self.head
   while iter(self):
       yield n.val
       n = next(self)
   ```

   (d)
   ```
   n = self.head.next
   while n is not self.head:
       yield n.val
       n = n.next
   ```

8. Which choice completes the following function so that it returns `True` only for strings that contain properly balanced pairs of parentheses, brackets, and curly braces?

```
def check_parens(str):
    pairs = {'(': ')', '[': ']', '{': '}'}
    stack = Stack()
    for c in str:
        if c in pairs.keys():
            stack.push(c)
        elif c in pairs.values():

            ____________________________________
    if stack:
```

```
            return False
        else:
            return True
```

(a)
```
if not stack:
    return False
```

(b)
```
if not stack or c != pairs[stack.pop()]:
    return False
```

(c)
```
if c == pairs[stack.pop()]:
    return True
```

(d)
```
while stack.pop() != c
    if not stack:
        return False
```

9. What are the contents of the list `lst` after the following program is executed?

```
s = Stack()
q = Queue()
lst = []
for i in range(10):
    s.push(i)
while s:
    q.enqueue(s.pop())
while q:
    lst.append(q.dequeue())
```

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

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

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

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

10. When using a "circular" array to implement a queue, which of the following statements would best serve to increment the index of the tail element? Assume that `self.data` refers to the backing array.

(a) `self.tail_idx = (self.tail_idx % len(self.data)) + 1`

(b) `self.tail_idx = (self.tail_idx + 1) % len(self.data)`

(c) `self.tail_idx = self.tail_idx + (1 % len(self.data))`

(d) `self.tail_idx = (self.tail_idx + 1) % (len(self.data) + 1)`

11. When using singly-linked nodes (i.e., each containing only a value and a "next" reference) to implement a queue, which of the following statements best implements the "enqueue" operation?

(a) `self.tail.next = self.tail = Node(new_val)`

(b) `self.tail = Node(new_val, self.tail.next)`

(c) `self.tail = self.tail.next = Node(new_val, self.tail)`

(d) `self.tail.next = self.tail = Node(new_val, self.tail.next)`

12. What can we say, definitively, about the hash code of the string `'hello'` (i.e., the result of `hash('hello')`) in Python?

(a) it is greater than the hash code of `'goodbye'`

(b) it is equivalent to the hash code of `'olleh'[::-1]`

(c) it is smaller than the value 1,000,000

(d) it is equivalent to `hash('h') + hash('e') + hash('l') + hash('l') + hash('o')`

13. If we assume uniform hashing, what is the probability that two keys being inserted into an empty hashtable of 100 buckets will NOT collide?

(a) 1/100

(b) 99/100

(c) (99/100) × (98/100)

(d) 1 - 99/100

14. Consider the following hashtable method, intended to compute and return the number of key/value mappings:

```
def count(self):
    n = 0
    ____________________________
    ____________________________
    ____________________________
    return n
```

Which correctly completes the implementation?

(a)
```
for b in self.buckets:
    while b:
```

```
            n += 1
            b = b.next
```

(b)
```
for i in len(self.buckets):
    if self.buckets[i] is not None:
        n += 1
```

(c)
```
for b in self.buckets:
    for k in b:
        n += 1
```

(d)
```
b = self.buckets
while b:
    if b.key:
        n += 1
    b = b.next
```

15. Consider the following implementation of `__setitem__` in a hashtable:

```
def __setitem__(self, key, val):
    bucket_idx = hash(key) % len(self.buckets)
    if not self.buckets[bucket_idx]:
        self.buckets[bucket_idx] = Hashtable.Node(key, val)
    else:
        n = self.buckets[bucket_idx]
        while n:
            ____________________________
            ____________________________
            ____________________________
            n = n.next
```

Which correctly completes the implementation?

(a)
```
if n.key == key and n.val == val:
    n.val = val
    return
```

(b)
```
if n.key != key:
    n = Hashtable.Node(key, val, next=n.next)
    return
```

(c)
```
if n.next is None:
    n.next = Hashtable.Node(key, val)
    return
elif not n.next:
    n.val = val
    return
```

(d)
```
if n.key == key:
    n.val = val
    return
elif not n.next:
    n.next = Hashtable.Node(key, val)
    return
```

16. Consider the max-heap constructed from the following sequence of values (starting with the leftmost):

```
2, 4, 3, 8, 1, 7, 5
```

What is the resulting array representation of the max-heap (based on the `add` method covered in class)?

(a) `[8, 4, 7, 2, 1, 3, 5]`

(b) `[8, 7, 5, 4, 3, 2, 1]`

(c) `[8, 5, 7, 4, 3, 1, 2]`

(d) `[8, 2, 4, 3, 1, 7, 5]`

17. If adding the value 6 to the max-heap constructed in the previous problem, how many swaps would need be performed to restore the heap property?

(a) 0

(b) 1

(c) 2

(d) 3

18. What would be the new array representation of the max-heap originally constructed in problem (16), after removing the root and re-heapifying?

(a) `[7, 4, 2, 1, 3, 5]`

(b) `[7, 5, 4, 3, 2, 1]`

(c) `[7, 2, 4, 3, 1, 5]`

(d) `[7, 4, 5, 2, 1, 3]`

19. Which of the following returns the *second largest* element in a max-heap, assuming the heap contains at least 2 elements?

(a)
```
return self.data[-1]
```

(b)
```
return self.data[1]
```

(c)
```
if len(self.data) == 2 or self.data[1] > self.data[2]:
    return self.data[1]
else:
    return self.data[2]
```

(d)
```
i = 1
while i < len(self.data):
    i = Heap._right(i)
return self.data[i-1]
```

20. Consider the following __init__ and add methods for a heap:

```
def __init__(self, key):
    self.data = []
    self.key  = key

def add(self, x):
    self.data.append(x)
    i = len(self.data) - 1
    p = Heap._parent(i)
    while i > 0 and self.key(self.data[p]) < self.key(self.data[i]):
        self.data[p], self.data[i] = self.data[i], self.data[p]
        i = p
        p = Heap._parent(i)
```

Which of the following creates a heap to which strings may be added, such that self.data[0] (the root of the heap) always refers to the string with the *smallest* length?

(a)
```
Heap(key=lambda s: s[::-1])
```

(b)
```
Heap(key=lambda s: len(s))
```

(c)
```
Heap(key=lambda s: -len(s))
```

(d)
```
Heap(key=lambda s: len(s) - len(key))
```