CS 331 Midterm Exam 2

Friday, November 4th, 2016

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 time complexity for retrieving the value for an element at a specified index in a circular, doubly-linked list of *N* elements?

   (a) $O(1)$

   (b) $O(\log N)$

   [✓ (c)] $O(N)$

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

2. What is the time complexity for removing the last element 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. What is the time complexity for popping off the most recent (i.e., newest) element pushed onto a stack which contains *N* elements?

   [✓ (a)] $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

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

4. Which of the following data structures would best be used for the evaluation of a postfix arithmetic expression (e.g., “8 4 + 3 /“)?

   (a) an array

   [✓ (b)] a stack

   (c) a queue

   (d) a hashtable

5. Which of the following data structures manages elements in a strictly first-in, first-out manner?

   (a) an array

   (b) a stack

   [✓ (c)] a queue

   (d) a hashtable

6. If we assume uniform hashing, what is the probability that a collision will occur in a hashtable with 1000 buckets and 3 keys?

   (a) $\frac{3}{1000}$

   (b) $1 - \frac{3}{1000}$

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

   [✓ (d)] $1 - (\frac{999}{1000} \times \frac{998}{1000})$

7. What is ***not*** a possible result of the expression `hash(key) % 10000`, where key is an arbitrary, hashable Python object?

   (a) 0

   (b) 1

   (c) 9999

   (d) **`10000`**

8. Which correctly implements the `__init__` method in a circular, doubly-linked list with a sentinel head?

   (a)
   ```
   def __init__(self):
       self.head = LinkedList.Node(None, next=self.head.next, prior=self.head.prior)
       self.head.next = self.head.prior = self.head
   ```

   (b)
   ```
   def __init__(self):
       self.head = LinkedList.Node(None)
       self.head = self.head.next
   ```

   (c)
   `def __init__(self):`
   `    self.head = LinkedList.Node(None)`
   `    self.head.next = self.head.prior = self.head`

   (d)
   ```
   def __init__(self):
       self.head.next = self.head.prior = self.head = LinkedList.Node(None)
   ```

9. Which correctly implements the `prepend` method in a circular, doubly-linked list with a sentinel head?

(a)
```
def prepend(self, val):
    n =  LinkedList.Node(val)
    self.head.prior = self.head.next = n
```

(b)
```
def prepend(self, val):
    n =  LinkedList.Node(val, next=self.head.next, prior=self.head)
   self.head.next.prior = self.head.next = n
```

(c)
```
def prepend(self, val):
    n =  LinkedList.Node(val, next=self.head.next.next, prior=self.head.prior)
    self.head.prior = self.head.next = n
```

(d)
```
def prepend(self, val):
    n =  LinkedList.Node(val, next=self.head.next, prior=self.head.prior)
    self.head.next.prior = self.head.prior.next = n
```

10. Which correctly implements the `clear` method (to remove all elements) in a circular, doubly-linked list with a sentinel head?

(a)
```
def clear(self):
   self.head.next = self.head.prior = self.head
```

(b)
```
def clear(self):
    self.head = self.head.next = self.head.prior
```

(c)
```
def clear(self):
    self.head.next.prior = self.head.prior.next
```

(d)
```
def clear(self):
    self.head = None
```

11. Consider the following implementation of `__delitem__` in a circular, doubly-linked list with a sentinel head:

```
def __delitem__(self, idx):
    n = self.head.next
    for _ in range(idx):
        n = n.next
    ________________________
    ________________________
```

Which correctly completes the implementation?

(a)
```
n.prior, n.next = n.next, n.prior
```

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

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

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

12. Which correctly implements an iterator over all the values in a hashtable?

(a)
```
def values(self):
    for b in self.buckets:
        yield b.val
        yield b.next
```

(b)
```
def values(self):
    for b in self.buckets:
        while b:
            yield b.val
            b = b.next
```

(c)
```
def values(self):
    b = self.buckets[0]
    while b:
        yield b.val
        b = b.next
```

(d)
```
def values(self):
    for i in range(len(self.buckets)):
        yield self.buckets[i].val
```

13. 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
```

14. Which correctly implements the `push` operation in an array-backed stack with a time complexity of O(1)?

(a) **`self.data.append(val)`**

(b) `self.data.insert(0, val)`

(c) `self.data[-1] = val`

(d)
```
self.data.append(None)
for i in range(0, len(self.data)-1):
    self.data[i+1] = self.data[i]
self.data[0] = val
```

15. Which correctly implements the `push` operation in a singly-linked stack?

    (a) `self.top.next = Stack.Node(val, next=self.top)`

    (b) **`self.top = Stack.Node(val, next=self.top)`**

    (c) `self.top.next = self.top = Stack.Node(val)`

    (d) `self.top = self.top.next = Stack.Node(val, next=self.top.next)`

16. Which correctly exchanges the top two elements in a singly-linked stack (assuming that the stack has at least 2 elements)?

    (a) `self.top.next, self.top = self.top, self.top.next`

    (b) `self.top.next, self.top.next.next = self.top.next.next, self.top.next`

    (c) **`self.top.next.next, self.top.next, self.top = self.top, self.top.next.next, self.top.next`**

    (d) `self.top, self.top.next, self.top.next.next = self.top.next.next, self.top, self.top.next`

17. Which correctly implements a method to re-queue — i.e., dequeue, then enqueue again — the front-most/oldest element in an array-backed queue (assuming the queue is not empty)?

    (a)
    ```
    def requeue(self):
        self.data[0], self.data[-1] = self.data[-1], self.data[0]
    ```

    (b)
    ```
    def requeue(self):
        del self.data[-1]
        self.data.append(self.data[0])
        del self.data[0]
    ```

    (c)
    ```
    def requeue(self):
        x = self.data[0]
        for i in range(len(self.data)-1, 0, -1):
            self.data[i-1] = self.data[i]
       self.data[-1] = x
    ```

    (d)
    ```
    def requeue(self):
        x = self.data[0]
        for i in range(1, len(self.data)):
            self.data[i-1] = self.data[i]
       self.data[-1] = x
    ```

18. Which correctly implements the `enqueue` operation in a singly-linked queue such that enqueue and dequeue can each be implemented with a time complexity of O(1)?

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

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

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

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

19. Which correctly implements a method to discard the front-most/oldest `n` elements from a singly-linked queue (assuming that the queue has at least `n+1` elements)?

(a)
```
def drop(self, n):
    for _ in range(n):
        self.head = self.head.next
```

(b)
```
def drop(self, n):
    for _ in range(n-1):
        self.head.next = self.head.next.next
```

(c)
```
def drop(self, n):
    node = self.head
    for _ in range(n-1):
        node = node.next
    node.next = self.head
```

(d)
```
def drop(self, n):
    node = self.head
    for _ in range(n):
        node.next.val = node.val
        node = node.next
```

20. Consider the following implementation of a method to reverse the elements in a singly-linked queue (assuming that the queue is not empty):

```
def reverse(self):
    p, q = self.head, self.head.next
    while q:
        _________________________________
    self.tail, self.head = self.head, self.tail
    self.tail.next = None
```

Which correctly completes the implementation?

(a) `p, q = q, p`

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

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

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