# CS 331 Final Exam

## Monday, December 5th, 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') on the answer sheet.

1. One way to reverse the order of elements in a stack is to pop off and enqueue each element one by one into a queue, then to dequeue the elements while pushing them back onto the stack. What is the time complexity of this when applied to a stack of *N* elements?

   (a) O(log N)

   **(b) O(N)**

   (c) O(N log N)

   (d) $O(N^2)$

2. An algorithm calls for appending *N* elements to an array-backed list, while invoking merge sort on the list after each append operation to keep the list sorted. What is the time complexity of the algorithm?

   (a) O(log N)

   (b) O(N log N)

   **(c) $O(N^2 \log N)$**

   (d) $O(N^3)$

3. What is the 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)

4. What is the 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)

5. What is the time complexity for inserting a new element into an unbalanced binary search tree containing *N* elements, assuming that the new element's value is neither the new minimum nor maximum value after insertion?

   (a) O(1)

   (b) O(log N)

   **(c) O(N)**

   (d) O(N log N)

6. 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)

7. 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 queue

**(b) a hashtable**

(c) a max-heap

(d) a binary search tree

8. Which data structure would you choose for an application where it is necessary to track multiple objects, but where only the object with the largest value of some specified attribute needs to be retrieved and processed?

(a) a queue

(b) a hashtable

**(c) a max-heap**

(d) a binary search tree

9. Which data structure would you choose for an application where it is necessary to track multiple objects, but where only the object that has been tracked for the longest needs to be retrieved and processed?

**(a) a queue**

(b) a hashtable

(c) a max-heap

(d) a binary search tree

10. Which data structure would you choose for an application where it is necessary at all times to maintain a full ordering of multiple objects across insertions and deletions?

(a) a queue

(b) a hashtable

(c) a max-heap

**(d) a binary search tree**

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

```
4 2 9 5 3 8
```

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

(a) `[2, 3, 4, 5, 8, 9]`

(b) `[9, 4, 8, 5, 3, 2]`

(c) `[9, 8, 5, 4, 3, 2]`

**(d) `[9, 5, 8, 2, 3, 4]`**

12. If adding the value 7 to the max-heap constructed in problem (11), how many swaps would need be performed to restore the heap property?

**(a) 0**

(b) 1

(c) 2

(d) 3

13. What would be the new array representation of the max-heap originally constructed in problem (11), after removing the root and re-heapifying?

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

(b) `[5, 8, None, 2, 3, 4]`

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

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

14. If we were to continue adding values to the max-heap until the data structure contained 100 values, what would be the maximum number of swaps needed upon adding the 101st value, in order to maintain the heap property?

(a) 2

**(b) 6**

(c) 10

(d) 15

15. Consider the non-balanced binary search tree constructed from the following sequence of values (starting with the leftmost):

```
4 8 10 1 3 5
```

What is the value stored in the root of the tree?

(a) 3

**(b) 4**

(c) 5

(d) 10

16. What is the height of the tree (i.e., the longest chain of nodes from root to leaf) from problem (15)?

**(a) 3**

(b) 4

(c) 5

(d) 6

17. If inserting the value 6 into the tree constructed in problem (15), where would it go?

(a) Above node (4)

(b) As the right child of node (3)

**(c) As the right child of node (5)**

(d) As the left child of node (8)

18. If deleting the value 1 from the tree constructed in problem (15), how should we best go about updating the tree?

(a) Set the left child of (3) to None

(b) Set the right child of (3) to None

(c) Set the left child of (5) to node (4)

**(d) Set the left child of (4) to node (3)**

19. Which choice completes the following implementation of `insert`, which inserts a new value `val` at (positive) position `idx` within an array-backed list?

```
def insert(self, idx, val):
    self.data.append(None)

    ________________________
    self.data[idx] = val
```

(a)
```
for i in range(idx, len(self.data)):
    self.data[i+1] = self.data[i-1]
```

(b)
```
for i in range(idx, len(self.data)):
    self.data[i+1] = self.data[len(self.data)-i]
```

(c)
```
for i in range(len(self.data)-1, idx, -1):
    self.data[len(self.data)-i] = self.data[i]
```

**(d)**
```
for i in range(len(self.data)-1, idx, -1):
    self.data[i] = self.data[i-1]
```

20. Which choice removes the first element from a circular doubly-linked list with a sentinel head node?

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

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

**(c)**
```
self.head.next.next.prior = self.head
self.head.next = self.head.next.next
```

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

21. Which choice implements the enqueue operation in a singly-linked, non-empty queue?

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

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

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

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

22. Which choice correctly implements the parent and left/right child index computation functions for the array-backed heap?

**(a)**
```
def _parent(idx): return (idx-1)//2
def _left(idx):   return idx*2+1
def _right(idx):  return idx*2+2
```

(b)
```
def _parent(idx): return (idx-1)//2
def _left(idx):   return idx*2
def _right(idx):  return idx*2+1
```

(c)
```
def _parent(idx): return (idx-1)%2
def _left(idx):   return idx*(2+1)
def _right(idx):  return idx*(2+2)
```

(d)
```
def _parent(idx): return (idx-1)//2
def _left(idx):   return (idx+1)*2
def _right(idx):  return (idx+2)+2
```

23. Given that we are to delete the value stored in some binary search tree node n containing non-empty left and right subtrees, which choice locates a node `t` that contains a suitable replacement value for n and can also be more easily removed from the tree?

(a)
```
t = n.left.right
```

(b)
```
t = n
while t:
    t = t.left
```

**(c)**
```
t = n.right
while t.left:
    t = t.left
```

(d)
```
t = n.left
while t:
    t = t.right
t = t.left
```

24. Which choice completes the following method, which returns the total number of nodes in a binary search tree containing a value less than (and not equal to) `val`?

```
def count_lt(self, val):
    def count_lt_rec(n):



        ____________________________________
    return count_lt_rec(self.root)
```

**(a)**
```
if not n:
    return 0
elif n.val < val:
    return 1 + count_lt_rec(n.left) + count_lt_rec(n.right)
else:
    return count_lt_rec(n.left)
```

(b)
```
if not n:
    return 0
elif n.val < val:
    return 1 + count_lt_rec(n.right)
else:
    return count_lt_rec(n.left)
```

(c)
```
if not n:
    return 0
elif n.val >= val:
    return 1 + count_lt_rec(n.left)
else:
    return 1 + count_lt_rec(n.right)
```

(d)
```
if not n:
    return 0
elif n.val >= val:
    return 1 + count_lt_rec(n.left) + count_lt_rec(n.right)
else:
    return count_lt_rec(n.left)
```

25. Which completes the following iterator implementation, which is to yield the values of the associated binary search tree in ascending order?

```
def __iter__(self):
    def iter_rec(node):
        if node:


            _____________________________
    yield from iter_rec(self.root)
```

(a)
```
yield node.val
yield from iter_rec(node.left)
yield from iter_rec(node.right)
```

**(b)**
```
yield from iter_rec(node.left)
yield node.val
yield from iter_rec(node.right)
```

(c)
```
yield from iter_rec(node.right)
yield from iter_rec(node.left)
yield node.val
```

(d)
```
yield from iter_rec(node.left)
yield from iter(self)
yield from iter_rec(node.right)
```

26. Which of the following functions demonstrate *linear* recursion?

**(a)**
```
def foo_rec(n):
    if n == 1:
        return
    elif n % 2 == 0:
        return foo_rec(n / 2)
    else:
        return foo_rec(3*n + 1)
```

(b)
```
def foo_rec(x, y):
    if x == 0:
        return y+1
    elif y == 0:
        return foo_rec(x-1, 1)
    else:
        return foo_rec(x-1, foo_rec(x, y-1))
```

(c)
```
def foo_rec(n):
    if n <= 1:
```

```
            return 1
        else:
            return n * foo_rec(n-1) * foo_rec(n-2)
```

(d)
```
def foo_rec(x, y):
    if x < 0 or y < 0:
        return False
    elif foo_rec(x-1, y):
        return foo_rec(x, y-1)
    return True
```

27. Consider the following function:

```
def last(a):
    if not a:
        return None
    elif len(a) == 1:
        return a[0]
    else:
        return _______________
```

Which recursive call completes the implementation so that when passed a non-empty list as the initial argument, the last element in the list is returned?

(a) `last(a[0])`

**(b) `last(a[1:])`**

(c) `last(a[:-1])`

(d) `last(a[1:-1])`

28. Consider the following function:

```
def permutations(lst, p=()):
    if not lst:
        print(p)
    else:
        _____________________________
        _____________________________
```

Which completes the implementation so that when passed a list, all permutations of the list elements are printed out? E.g., when called with the list `[1, 2, 3]`, the output would consist of `(1, 2, 3)`, `(1, 3, 2)`, `(2, 1, 3)`, `(2, 3, 1)`, `(3, 1, 2)`, `(3, 2, 1)` — though not necessarily in that order.

(a) `permutations(lst[1:], (lst[0],) + p)`

(b)
```
permutations(lst[1:], p + (lst[0],))
permutations(lst[1:], p)
```

**(c)**
```
for i in range(len(lst)):
    permutations(lst[:i] + lst[i+1:], p + (lst[i],))
```

(d)
```
for i in range(1, len(lst)):
    permutations(lst[i:], p + (lst[i],))
```

29. For this and the next question, consider the following recursive function, devised by a student as an implementation of an alternative sort algorithm:

```
def swap_sort(lst):
    if not lst or len(lst) == 1:
```

```
        return lst
    elif lst[0] > lst[-1]:
        return [ lst[-1] ] + swap_sort(lst[1:-1]) + [ lst[0] ]
    else:
        return [ lst[0] ] + swap_sort(lst[1:-1]) + [ lst[-1] ]
```

The algorithm is flawed, however. What does it return given the input list `[10, 3, 4, 1, 8, 16, 2]`?

(a) `[1, 3, 2, 4, 8, 10, 16]`

**(b) `[2, 3, 4, 1, 8, 16, 10]`**

(c) `[3, 10, 1, 4, 8, 16, 2]`

(d) `[10, 16, 8, 4, 3, 2, 1]`

30. We can approximate the runtime complexity of `swap_sort` by estimating the number of recursive calls (i.e., the depth of recursion). What is the runtime complexity of `swap_sort` given an input list of N elements?

**(a) O(N)**

(b) O(log N)

(c) O(N log N)

(d) $O(N^2)$