CS 331 Summer 2017

Final Exam

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 (30 points):

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

2. 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) $O(N \log N)$

3. 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) $O(N \log N)$

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

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

5. 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 heap
   (d) a binary search tree

6. 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 heap
   (d) a binary search tree

7. 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 heap
   (d) a binary search tree

8. 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 heap
   (d) a binary search tree

9. 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) $O(N \log N)$

10. Which 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]`

11. Which 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`

12. Consider the following function:

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

    Which 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])`

13. 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],))
```

14. 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]`

15. 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) $O(N^2)$

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 to the nth 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:

- Your implementation should *not* change the value contained in any of the nodes nor create any new nodes. Instead, it should work by re-linking nodes.
- You may assume the argument, n, to `roll` is at least 2, and that the stack contains at least n values.

Heaps (8 points):

WP2 (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.

WP2 (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.

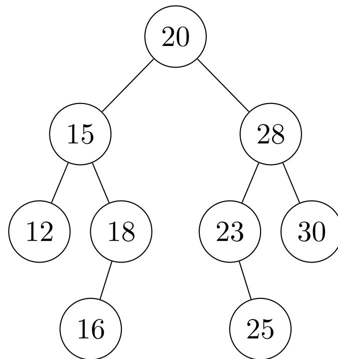
Binary Search Trees (12 points):

WP3 (a) Consider the following ordered sequence of values to be added to a (unbalanced) binary search tree:

8, 4, 9, 5, 7, 6

Sketch the tree after adding each value.

WP3 (b) Consider the following binary search tree:



Sketch the resulting tree after removing the value 20 and replacing it with a suitable candidate (as in the deletion algorithm presented in class).

WP3 (c) Implement the binary search tree method `predecessor`, which returns the largest value in the tree that is smaller than the argument. If there is no such value, the method returns `None`. You should not assume that the argument value itself is contained within the tree.

E.g., calling `predecessor(17)` on the tree above should return the value 16.

Note that a recursive helper function is already defined and called for you.