

## Data Structures - Test 1

Question 1. (5 points) Consider the following Python code.

```
for i in range(n * n):
    for j in range(0, n, 2):
        print (i, j)
```

What is the big-oh notation  $O()$  for this code segment in terms of  $n$ ?

Question 2. (5 points) Consider the following Python code.

```
i = 1
while i <= n:
    for j in range(n):
        print(j)
        for k in range(n):
            print(k)
    i = i * 2
```

What is the big-oh notation  $O()$  for this code segment in terms of  $n$ ?

Question 3. (5 points) Consider the following Python code.

```
for i in range(n):
    j = n
    while j > 1:
        print(j)
        j = j // 2
    for k in range(n):
        print(k)
```

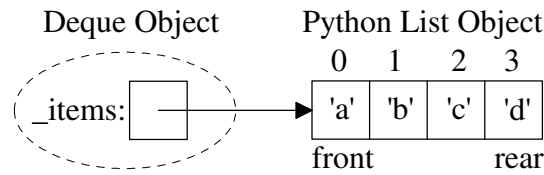
What is the big-oh notation  $O()$  for this code segment in terms of  $n$ ?

Question 4. (10 points) Suppose a  $O(n^2)$  algorithm takes 10 seconds when  $n = 1,000$ . How long would you expect the algorithm to run when  $n = 10,000$ ?

Question 5. (10 points) Why should any method/function having a "precondition" raise an exception if the precondition is violated?

Question 6. A Deque (pronounced “Deck”) is a linear data structure which behaves like a double-ended queue, i.e., it allows adding or removing items from either the front or the rear of the Deque. One possible implementation of a Deque would be to use a built-in Python list to store the Deque items such that

- the front item is **always stored at index 0**,
- the rear item is always at index  $\text{len}(\text{self}.\_items)-1$  or  $-1$



a) (6 points) Complete the big-oh  $O()$ , for each Deque operation, assuming the above implementation. Let  $n$  be the number of items in the Deque.

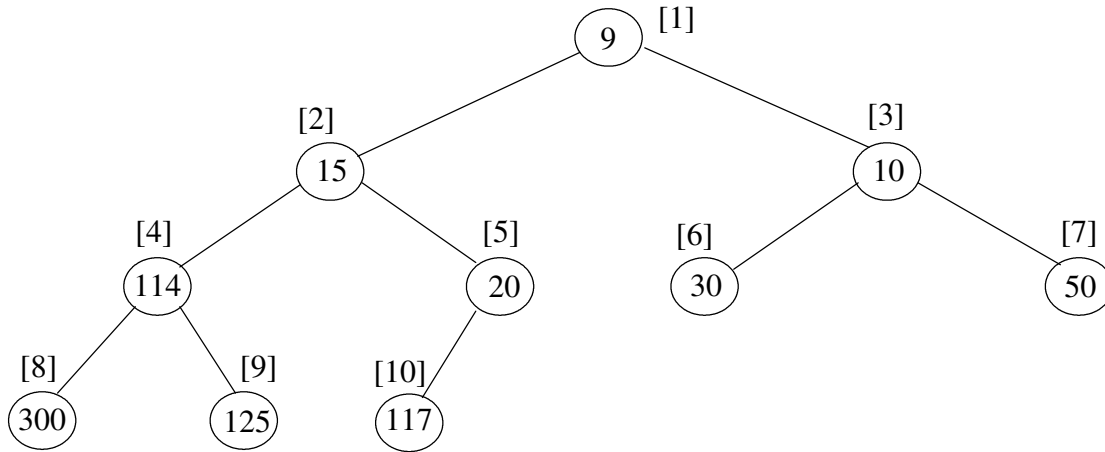
isEmpty	addFront	removeFront	addRear	removeRear	size

b) (9 points) Complete the method for the `removeRear` operation including the precondition check.

```
def removeRear(self):
    """Removes and returns the rear item of the Deque
    Precondition: the Deque is not empty.
    Postcondition: Rear item is removed from the Deque and returned"""
```

c) (5 points) Suggest an alternate Deque implementation to speed up some of its operations.

Question 7. Consider the binary heap approach to implement a priority queue. A Python list is used to store a *complete binary tree* (a full tree with any additional leaves as far left as possible) with the items being arranged by *heap-order property*, i.e., each node is  $\leq$  either of its children. An example of a *min* heap “viewed” as a complete binary tree would be:

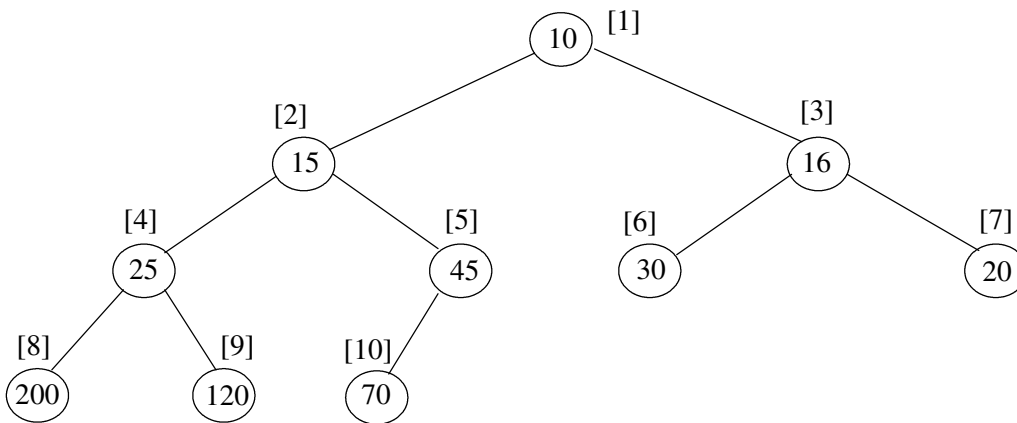


Python List actually used to store heap items

	0	1	2	3	4	5	6	7	8	9	10
	not used	9	15	10	114	20	30	50	300	125	117

- a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index  $i$ , what is the index of:
- its left child if it exists:
  - its right child if it exists:
  - its parent if it exists:
- b) (6 points) What would the above heap look like after inserting 7 and then 8 (show the changes on above tree)
- c) (2 points) What is the big-oh notation for inserting a new item in the heap?

Now consider the `delMin` operation that removes and returns the minimum item.

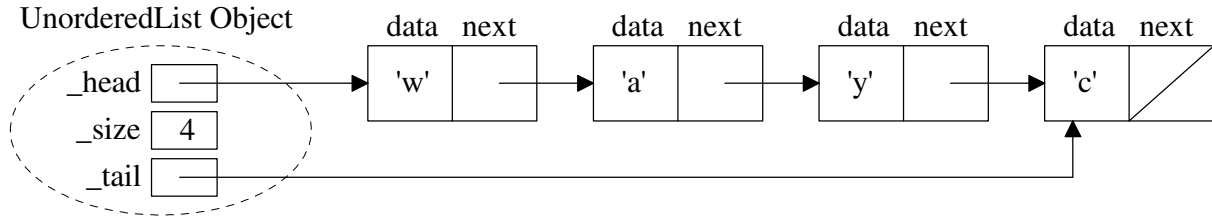


Python List actually used to store heap items

	0	1	2	3	4	5	6	7	8	9	10
	not used	10	15	16	25	45	30	20	200	120	70

- d) (1 point) What item would `delMin` remove and return from the above heap?
- e) (6 points) What would the above heap look like after `delMin`? (show the changes on above tree)
- f) (2 points) What is the big-oh notation for `delMin`?

Question 8. The textbook's unordered list ADT uses a singly-linked list implementation. I added the `_size` and `_tail` attributes:



a) (15 points) The `insert(position, item)` method adds the `item` to the list at the specified `position`. Unlike the textbook's implementation, **ASSUME** that the list may contain duplicate items!!! The precondition is that `position` is a nonnegative integer. If `position` is 0, then add it to the head of the list. If `position` is `_size` or bigger, then add it to the tail of the list. Complete the `insert(position, item)` method code including the precondition check.

```
class UnorderedList:
    def __init__(self):
        self._head = None
        self._size = 0
        self._tail = None

    def insert(self, position, item):
```

b) (10 points) Assuming the unordered list ADT described above **that allows duplicate items**. Complete the big-oh  $O()$  for each operation. Let  $n$  be the number of items in the list.

<code>insert(position, item)</code>	<code>pop()</code> removes and returns tail item	<code>length()</code> returns number of items in list	<code>append(item)</code> adds item to the tail of list	<code>add(item)</code> adds item to the head of list