Spring 2013 Name: ______________________

Data Structures - Test 1

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

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

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 = 2**n   # this is 2^n
while i > 1:

    for j in range(n):
        print(j)

    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.

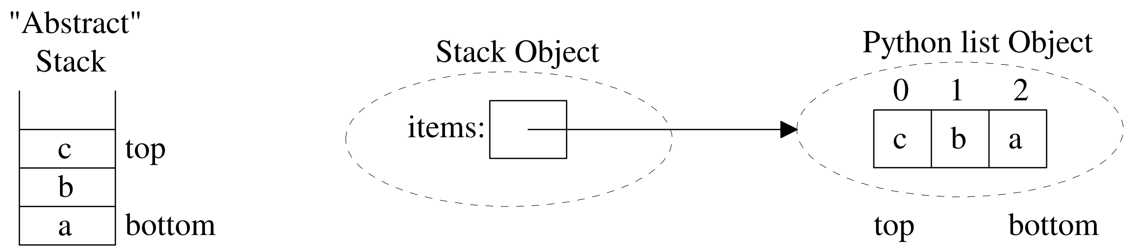
```
def main(n):
    for i in range(n):
        doSomething(n)

def doSomething(n):
    for k in range(n):
        print(k)

main(n)
```

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

Question 4. (10 points) Suppose a $O(n^4)$ 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 you design a program instead of “jumping in” by start writing code?

Question 6. Consider the following Stack implementation utilizing a Python list:



a) (6 points) Complete the big-oh notation for the Stack methods assuming the above implementation: ("n" is the # items)

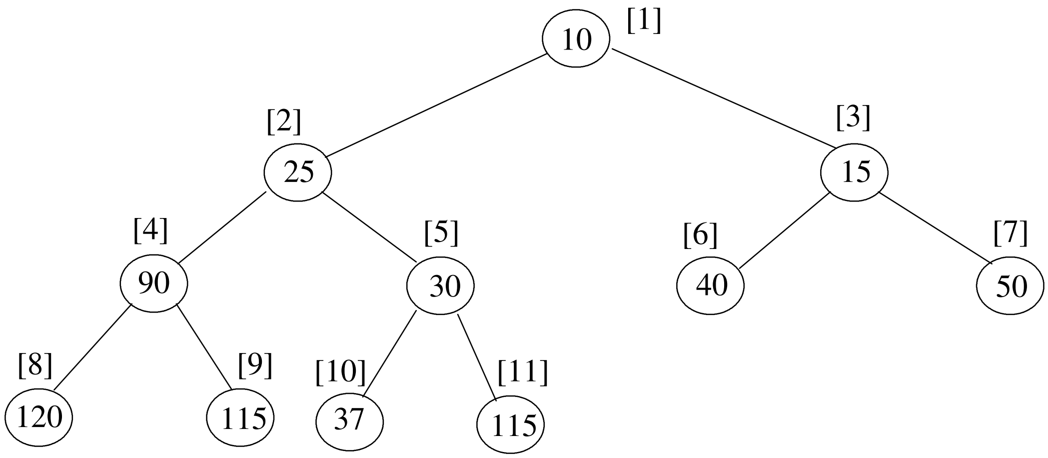
	push(item)	pop( )	peek( )	size( )	isEmpty( )	__init__
Big-oh						

b) (9 points) Complete the code for the `pop` method including the precondition check.

```
class Stack:

   def __init__(self):
      self._items = []

   def pop(self):
        """Removes and returns the top item of the stack
          Precondition:  the stack is not empty.
          Postcondition: the top item is removed from the stack and returned"""
```

c) (5 points) Suggest an alternate Stack 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 arranges by *heap-order property*, i.e., each node is ≤ either of its children. An example of a *min* heap "viewed" as a complete binary tree would be:



	0	1	2	3	4	5	6	7	8	9	10	11
Python List actually used to store heap items	not used	10	25	15	90	30	40	50	120	115	37	115

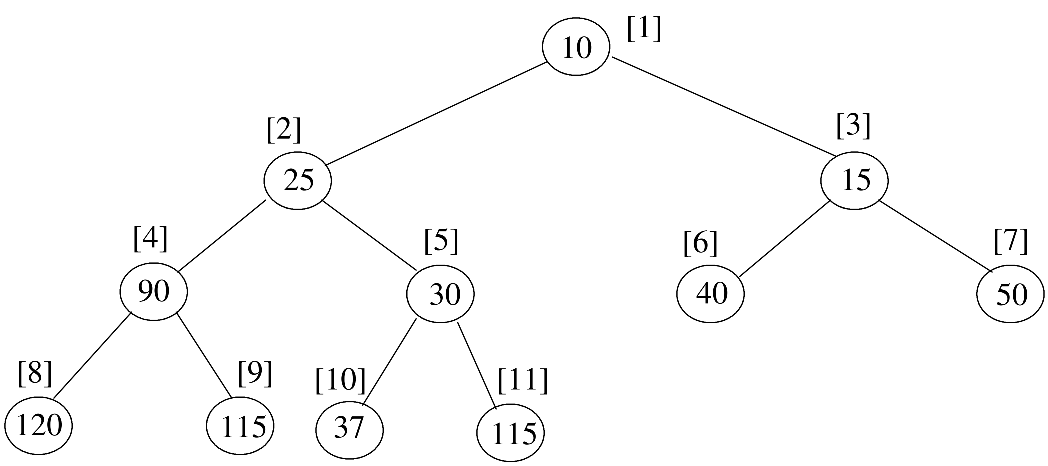
a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:

- 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 35 and then 12 (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.



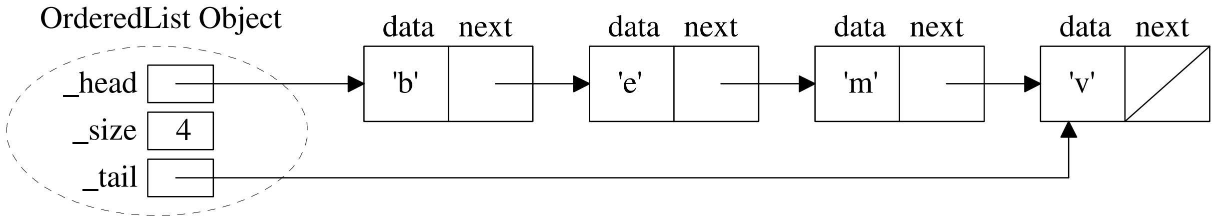
	0	1	2	3	4	5	6	7	8	9	10	11
Python List actually used to store heap items	not used	10	25	15	90	30	40	50	120	115	37	115

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 ordered list ADT uses a singly-linked list implementation. I added the `_size` and `_tail` attributes:



a) (15 points) The `pop(position)` method removes and returns the item at the specified `position`. The precondition is that `position` is a nonnegative integer corresponding to an actual list item (e.g., for the above list 0 ≤ `position` ≤ 3). Complete the `pop(position)` method code including the precondition check.

```
class OrderedList:

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

   def pop(self, position):
```

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

`pop(position)` removes and returns the item at the specified position	`pop()` removes and returns tail item	`length()` returns number of items in the list	`index(item)` returns the position of item in the list	`add(item)` adds item to its sorted spot in the list