Python Summary Name:_____

Print Function: the `print()` function takes a list of values to print and writes them to the output, e.g.,

```
print('cat',5,'dog')
print()  # blank line
print('pi is about', 3.14)
```

```
cat 5 dog

pi is about 3.14
```

Optional keyword arguments can be used to replace the defaults: space-character (`' '`) as a `separator`, the new-line-character (`'\n'`) as the `ending` character, and output file of the console (`sys.stdout`). The syntax with default parameters explicitly shown is:

```
print(value,...,sep=' ',end='\n', file=sys.stdout)
```

Print Function	Expected Output

`print('cat',5,end='')`
`print(' horse')`
`print('cow')`
	`cat 5 horse`
	`cow`

`print ('cat',5,'dog',sep='23',end='#')`	`cat23523dog#`
`print ('<',end='')` `print ('cat', 5, 'dog',sep='>, < ', end='>\n')`	`<cat>, <5>, <dog>`
`print ('cat',5,'dog',sep='23','horse')`	error since keyword arguments must be at the end of the parameter list

String Formatting: inside a string we use formatting placeholders (e.g., `%8d, %10s, %5.2f`), follow the string with the format operator (%), and a tuple supplying values for corresponding placeholders. For example:
`print("Name: %10s Age: %-7d GPA: %.2f" % ("Bob", 20, 3.138))` outputs the line:

```
Name:        Bob Age: 20      GPA: 3.14
```

`%10s` means left-justify string in 10 spaces, `%-7d` means right-justify decimal/int in 7 spaces, and `%.2f` left-justify float using the minimum spaces but with 2 decimal places. Another example with a character and exponent:
`print("character: %c float with exponent %e" % ('$', 123.456))` outputs the line:

```
character: $ float with exponent 1.234560e+02
```

Assignment Statement: the assignment statement creates a variable in memory and sets its value. The syntax is: <variable identifier> = <constant or expression value>, where identifiers must start with a letter or underscore (`'_'`), and then can be followed by letters, underscores, or digits.

Assignment Statements and Print Functions	Expected Output

`a = 123`
`b = a`
`a = a + 1`
`print ('a is', a)`
`print ('b is', b)`
	`a is 124`
	`b is 123`

`c = 'cat'`
`d = c`
`c = c + 'fish'  # string concatenation`
`print('c is', c)`
`print('d is', d)`
	`c is catfish`
	`d is cat`

`e = ['cat', 'dog']`
`f = e`
`e.append('cow')`
`print('e is',e)`
`print('f is', f)`
	`e = ['cat','dog','cow']`
	`f = ['cat','dog','cow']`

NOTE: This last example deals with assigning lists. In `e = ['cat', 'dog']` the variable `e` is assigned a reference/pointer to the list, so `f = e` assigns `f` a reference to the same list. There is only a single list! Thus, when we append `'cow'` to the list using `e`'s reference, it also printed in `f`'s list because it's the same list.

The first two examples deal with integers and strings which are *immutable* (i.e., unchangable). New immutable values are created with new references being assigned. After `b = a` both variables reference `123`. When `a = a + 1` executes, a new integer constant of `124` is created and its reference is assign to variable `a`. Variable `b` still references `123`.

Python Summary Name:_____

Input Function: the `input()` function reads a line from the keyboard (`sys.stdin`) and returns it as a string with the trailing new-line stripped. To input numeric values, the string needs to be explicitly cast (e.g. `eval(input()))`. For example, we can input and echo the user's name and age.

```
name = input("Enter your name: ")
age = eval(input("Enter your age: "))
print('Hi', name, end='!')
print('  Your age is', age)
```

```
Enter your name: Bob
Enter your age: 10
Hi Bob!  Your age is 10
```

Control Statements: the body of control statements are indented and there is NO other "end" ("}") marker

`if` statements: An `if` statement allows code to be executed or not based on the result of a comparison. If the condition evaluates to True, then the statements of the **indented body** is executed. If the condition is False, then the body is skipped. The syntax of `if` statements is:

```
if <condition>:
    statement1
    statement2
    statement3
```

```
if <condition>:
    statementT1
    statementT2
else:
    statementF1
    statementF2
```

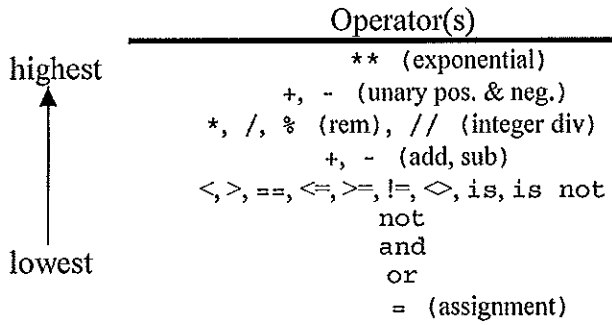
```
if <condition>:
    statementT1
    statementT2
elif <condition2>:
    statement
    statement
else:
    statementF1
    statementF2
```

Typically, the condition involves comparing "stuff" using relational operators (`<, >, ==, <=, >=, !=`).

Complex conditions might involve several comparisons combined using Boolean operators: `not, or, and`. For example, we might want to print "Your grade is B." if the variable score is less than 90, but greater than or equal to 80.

```
if score < 90 and score >= 80:
    print( "Your grade is B." )
```

The precedence for mathematical operators, Boolean operators, and comparisons are given in the table.



`for` loop: the `for` loop iterates once for each item in some sequence type (i.e, list, tuple, string).

```
for value in [1, 3, 9, 7]:
    print(value)
```

```
for character in 'house':
    print( character )
```

The `for` loop iterates over an iterable data-structure object (list, string, dictionary) or a range object created by the built-in `range` function which generate each value one at a time for each iteration of the loop. The syntax of: `range([start,] end, [, step])`, where [ ] are used to denote optional parameters. Examples:

- `range(5)` generates the sequence of values: `0, 1, 2, 3, 4`
- `range(2,7)` generates the sequence of values: `2, 3, 4, 5, 6`
- `range(10,2,-1)` generates the sequence of values: `10, 9, 8, 7, 6, 5, 4, 3`

For example:

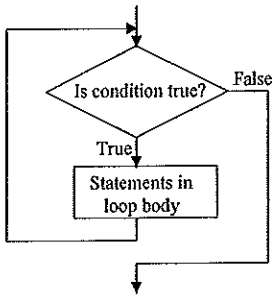
```
for count in range(1,6):
    print( count, end="  " )
print("\nDone")
```

```
1  2  3  4  5
Done
```

Python Summary Name:_____

while loop: A `while` statement allows code to be executed repeated (zero or more times) as long as the condition evaluates to True. The syntax of a `while` statement is:

```
while <condition>:
    statement1
    statement2
    statement3
```



An *infinite loop* is one that would loop forever. (FYI, in a Python shell ctrl-c (^c) can be used to kill the running program.) Most infinite loops are caused by programmer error, but sometimes they are intentional. The following "*sentinel-controlled*" code uses an infinite loop and a *break* statement that immediately causes control to exit the loop.

```
total = 0
counter = 0
while True:      # an infinite loop
    score = eval(input("Enter a score (or negative value to exit): "))
    if score < 0:
        break
    total += score
    counter += 1
print("Average is", total/counter)
```

Strings: Strings in Python are sequential collections of only characters. **Strings are immutable (i.e., cannot be changed), so new strings are generated by string operations.** Operations on strings **(or any sequence collection)** include:

Operation	**Operator**	**Explanation**	**Example** myString = "Hello!!!" aString = "cat"	**Result of Example**
Indexing	`[ <index> ]`	Access the element specified by the index	`myString[1]`	`'e'`
Slicing	`[ : ]`	Extract a part of the string	`myString[ 1:5 ]`	`'ello'`
Concatenation	`+`	Combine strings together	`myString + aString`	`'Hello!!!cat'`
Repetition	`*`	Concatenate a repeated number of times	`aString * 3`	`'catcatcat'`
Membership	`in`	Ask whether a substring is in a string	`'ell' in myString`	`True`
Length	`len(string)`	How many items are in the string?	`len( myString )`	`8`

Indexing of strings starts with 0 on the left end, and -1 on the right end:

```
                    1111
          01234567890123
cheer = 'GO Panthers!!!'
         -4-3-2-1
```

Omitted indexes in a slice means "from the end." For example, `cheer[:4]` generates `'GO P'`.

Omitted indexes in a slice means "from the end." For example, `cheer[-4:]` generates `'s!!!'`.

String objects also have the following methods: (the `string` module can be imported to provide more operations.)

Method	Usage	Explanation
center	myString.center(w)	Returns a string with myString centered in a field of size w
ljust	myString.ljust(w)	Returns a string with myString left-justified in a field of size w
rjust	myString.rjust(w)	Returns a string with myString right-justified in a field of size w
upper	myString.upper( )	Returns a string with myString in all upper-case characters
lower	myString.lower( )	Returns a string with myString in all lower-case characters
strip	myString.strip( )	Returns a string with leading and trailing whitespace (space, tab, new-line) chars. removed. An optional string parameter can be used to supply characters to strip instead of whitespace.
count	myString.count(sub)	Returns number of occurrences of sub in myString (Optional parameters: myString.count(sub [, start [, end]])
endswith	myString.endswith(sub)	Returns True if myString ends with the substring sub; otherwise it returns False
startswith	myString.startswith(sub)	Returns True if myString starts with the substring sub; otherwise it returns False
isdigit	myString.isdigit( )	Returns True if myString contains only digits; otherwise it returns False
isalpha	myString.isalpha( )	Returns True if myString contains only letters; otherwise it returns False
split	myString.split( )	Returns a list of substrings of myString splits at whitespace characters. An optional string parameter can supply characters to split on.
find	myString.find(sub)	Returns the starting index of the first occurrence of sub. (Optional parameters: myString.find(sub [, start [, end]])
replace	myString.replace(old,new)	Returns a string with all occurrences of substring old replaced by substring new. An additional integer parameter can specify the number of replacements to perform, e.g.,myString.replace(old,new, 3)

Lists: A Python list is also a sequence collection, but a list can contain items of any type (e.g., character, strings, integers, floats, other lists, etc.), and **lists are mutable.** Lists are represented by comma-separated values enclosed in square brackets ('[', ']'). Operations on lists (**or any sequence collection**, e.g., strings) include:

Operation	Operator	Explanation	Example myList=[5,6,7,8] ListB=[8,9]	Result of Example
Indexing	[<index>]	Access the element specified by the index	myList[2]	7
Slicing	[:]	Extract a part of the list	myList[1:3]	[6, 7]
Concatenation	+	Combine lists together	myList + ListB	[5, 6, 7, 8, 8, 9]
Repetition	*	Concatenate a repeated number of times	ListB * 3	[8, 9, 8, 9, 8, 9]
Membership	in	Ask whether an item is in a list	3 in myList	False
Length	len(list)	How many items are in the list?	len(myList)	4

Python Summary Name:_____

The following list methods are provided by Python:

Method	Usage	Explanation
append	`myList.append(item)`	Adds item to the end of myList
extend	`myList.extend(otherList)`	Extends myList by adding all items in otherList to myList's end
insert	`myList.insert(i, item)`	Insert item in myList at index i
pop	`myList.pop( )`	Remove and return the last item in myList
pop(i)	`myList.pop(i)`	Remove and return the ith item in myList *
del	`del myList[i]`	Deletes the item in the ith position of myList *
remove	`myList.remove(item)`	Removes the first occurrence of item in myList **
index	`myList.index(item)`	Returns the index of the first occurrence of item in myList **
count	`myList.count(item)`	Returns the number of occurrences of item in myList
sort	`myList.sort( )`	Modifies myList to be sorted
reverse	`myList.reverse( )`	Modifies myList to be in reverse order

* Note: raises an IndexError if the index `i` is not in the list

** Note: raises a ValueError if the `item` is not in the list

Tuples: A tuple is another sequence data type, so the sequence operations of indexing, slicing, concatenation, repetition, membership (in), and len() work on tuples too. Tuples are very similar to lists, i.e., comma-separated items enclosed in parentheses. The main difference is that **tuples are immutable** (cannot be modified).

Create two tuples as:

```
student1 = ('Bob', 123456, 'Jr', 3.12)
student2 = 'Sally', 654321, 'Fr', 0.0
```

In addition to indexing, "fields" of a tuple can be *unpacked* using a single assignment statement as:

```
name, idnum, rank, gpa = student1
```

(NOTE: This allows multiple values to be returned from a function)

Dictionaries: A dictionary is an unordered set of key-value pairs (written as key:value). Keys must be unique and immutable (e.g., numerics, strings, tuples of immutable objects). Dictionaries are typically used to lookup the value corresponding to a specified key. Dictionaries can be written as comma-separated `key:value` pairs enclosed in curly braces. For example,

```
phoneNumbers = {'fienup':35918,'gray':35917,'east':32939,'drake':35811,'schafer':32187}
```

Access to individual `key:value` pairs looks syntactically like a sequence lookup using a key instead of an index. For example, `phoneNumbers['east']` returns 32939, and a new `key:value` pair can be added by `phoneNumbers['wallingford'] = 35919`. Additional, methods on dictionaries are:

Method	Usage	Explanation
keys	`myDictionary.keys()`	Returns keys in an interable `dict_keys` object
values	`myDictionary.values()`	Returns values in an interable `dict_values` object
items	`myDictionary.items()`	Returns key:value tuples in an interable `dict_items` object
get item	`value = myDictionary[myKey]`	Returns the value associated with myKey; otherwise raises a *KeyError* if myKey is not in the dictionary
get	`myDictionary.get(myKey)`	Returns the value associated with myKey; otherwise *None*
get	`myDictionary.get(myKey, alt)`	Returns the value associated with myKey; otherwise alt
in	`myKey in myDictionary`	Returns True if myKey is in myDictionary; otherwise False
del	`del myDictionary[myKey]`	Deletes the key:value pair whose key is myKey

Text Files: Below is a summary of the important text-file operations in Python.

File Operations in Python		
General syntax	**Example**	**Description**
open(filename) open(filename, mode)	`f = open('data.txt', 'w')`	Modes: `'r'` read only; `'w'` write only; `'a'` append; `'r+'` both reading and writing. Default mode is `'r'`
f.close()	`f.close()`	Close the file to free up system resources.
loop over the file object	`for line in f:` `  print (line)`	Memory efficient, fast and simple code to loop over each line in the file.
f.readline()	`nextLine = f.readline()`	Returns the next line from the file. The newline ('\n') character is left at the end of the string.
f.write(string)	`f.write('cats and dogs\n')`	Writes the string to the file.
f.read( )	`all = f.read()`	Returns the whole file as a single string.
f.read(size)	`chunk = f.read(100)`	Returns a string of at most 100 (size) bytes. If the file has been completely read, an empty string is returned.
f.readlines()	`allLines = f.readlines()`	Returns a list containing all the lines of the file.
f.readlines(size)	`someLines = f.readlines(5000)`	Returns the next 5000 bytes of line. Only complete lines will be returned.

Below is a summary of the important file-system functions from the `os` and `os.path` modules in Python.

`os` Module File-system Functions	
General syntax	**Description**
`getcwd( )`	Returns the complete path of the current working directory
`chdir(path)`	Changes the current working directory to path
`listdir(path)`	Returns a list of the names in directory named path
`mkdir(path)`	Creates a new directory named path and places it in the current working directory
`rmdir(path)`	Removes the directory named path from the current working directory
`remove(path)`	Removes the file named path from the current working directory
`rename(old, new)`	Renames the file or directory named old to new

`os.path` Module File-system Functions	
General syntax	**Description**
`exists(path)`	Returns True if path exists and False otherwise
`isdir(path)`	Returns True if path is a directory and False otherwise
`isfile(path)`	Returns True if path is a file and False otherwise
`getsize(path)`	Returns the size in bytes of the object named path

NOTE: The initial "current working directory" is the directory where the program is located. Typically, it is useful to access files relative to the "current working directory" instead of specifying an absolute (complete) path. You can use the strings:

- `'.'` to specify the current working directory, e.g., `currentDirectoryList = os.listdir('.')`
- `'..'` to specify the parent of current working directory, e.g., `os.chdir('..')` which changes the current working directory to the parent directory

Functions:

A *function* is a procedural abstract, i.e., a named body of code that performs some task when it is called/invoked. Often a function will have one or more parameter that allows it to perform a more general (variable) task. For example, the cube function below can be called with any numeric value with the corresponding cube of that number being returned.

```
# Function to calculate the cube of a number
def cube(num):
    num_squared = num * num
    return num_squared * num

# call the function
value = 2
print('The value', value, 'raised to the power 3 is', cube(value))
print('The value 3 raised to the power 3 is', cube(3))
```

Terminology:

- a *formal parameter* is the name of the variable used in the function definition. It receives a value when the function is called. In the function cube, num is the formal parameter. Formal parameters are only known inside of the function definition. The section of a program where a variable is known is called its *scope*, so the scope of a formal parameter (and other *local variable* defined in the function such as num_squared) is limited to the function in which it is defined.
- an *actual parameter/argument* is the value used in the function call that is sent to the function. In the call to function cube, the variable `value` supplies the actual parameter value of 2.
- a *global variable* is created outside all functions and is known throughout the whole program file, e.g. `value`.

It is helpful to understand the "rules of the game" when a function is called. Memory is used to store the current program and the data associated with it. The memory used to store the data is divided as shown below.

- Global memory is used to store the global variables (and constants).
- The *heap* is used to store dynamically allocated objects as the program runs, e.g. lists, strings, ints, objects
- The *run-time stack* is used to store *call-frames* (or *activation records*) that get *pushed* on the stack when a function is called, and *popped* off the stack when a function returns.

When a function is called the section of code doing the calling is temporarily suspended, and a new call-frame gets pushed on top of the stack before execution of the function body. The call-frame contains the following information about the function being called:

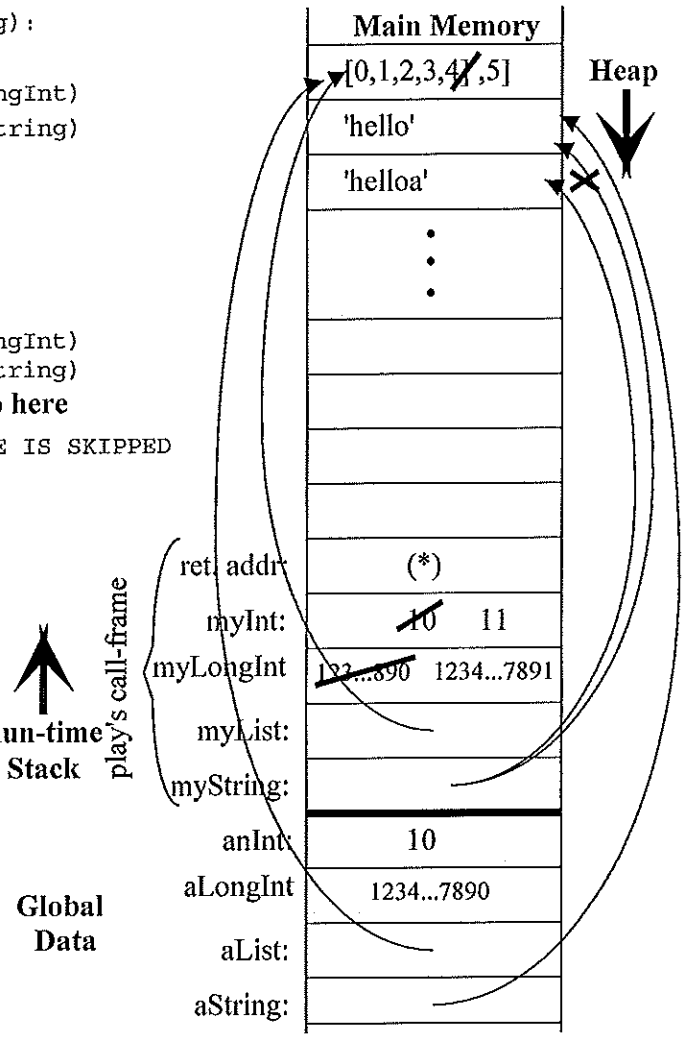
- the *return address* -- the spot in code where the call to the function occurred. This is needed so execution (control) can return there when the end of the function is reached or a `return` statement executes.
- room to store the formal parameters used by the function. In Python, parameters are *passed-by-value* which means that the value of each actual parameter in the function call is assigned to the corresponding formal parameter in the function definition before the function starts executing. However, the memory location for actual parameters for strings, lists, dictionaries, tuples, atomic objects contain only *references* to the heap
- room to store the local variables defined in the function (these are probably references to objects in the heap)

When a function returns, execution resumes at the function call (which is specified by the return address). A function typically sends back a value to the call by specifying an expression after `return` in the return statement. In Python if no expression is specified returned, then the special object `None` is returned.

```
def play(myInt, myLongInt, myList, myString):
    print('START OF play Function')
    print('myInt=',myInt,'myLongInt=',myLongInt)
    print('myList=',myList,'myString=',myString)
    myInt += 1
    myLongInt += 1
    myList.append(1)
    myString +='a'
    print('END OF play Function')
    print('myInt=',myInt,'myLongInt=',myLongInt)
    print('myList=',myList,'myString=',myString)
    return
```

Trace up to here

```
anInt = 10  # 1st STATEMENT EXECUTED, ABOVE IS SKIPPED
aLongInt = 12345678901234567890123456789OL
aList = range(5)
aString = 'hello'
print('BEFORE CALL')
print'anInt=',anInt,'aLongInt=',aLongInt)
print('aList=',aList,'aString=',aString)
play(anInt, aLongInt, aList, aString)
print('AFTER CALL') (*)
print('anInt=',anInt,'aLongInt=',aLongInt)
print('aList=',aList,'aString=',aString)
```



Output of complete program:

```
>>>
BEFORE CALL
anInt= 10 aLongInt= 12345678901234567890123456789O
aList= [0, 1, 2, 3, 4] aString= hello
START OF play Function
myInt= 10 myLongInt= 12345678901234567890123456789O
myList= [0, 1, 2, 3, 4] myString= hello
END OF play Function
myInt= 11 myLongInt= 12345678901234567890123456789O1
myList= [0, 1, 2, 3, 4, 1] myString= helloa
AFTER CALL
anInt= 10 aLongInt= 12345678901234567890123456789O
aList= [0, 1, 2, 3, 4, 1] aString= hello
>>>
```

NOTEs:

- `aList` passes a copy of **its reference to** the list as the initial value of the formal parameter `myList`, so both refer to the single list. In `play` when `myList.append(1)` executes, the single list is changed. Thus, when `play` terminates, `aList` still reflect this change.
- `aString` passes a copy of its reference to the string ('hello') as the initial value of the formal parameter `myString`. Since strings are immutable (cannot be changed) in Python, executing `myString += 'a'` cause a whole new string 'helloa' to be recreated in memory with `myString` referencing it. `aString` still refers to the string 'hello'.
- the `int` objects should actually be stored in the heap like the string objects since `int`'s are immutable .

Classes: A *class* definition is like a blueprint (receipe) for each of the objects of that class.
A class specifies a set of data attributes and methods for the objects of that class

- The values of the data attributes of a given object make up its state
- The behavior of an object depends on its current state and on the methods that manipulate this state
- The set of a class's methods is called its *interface*

The general syntax of class definition is:

```
class MyClass [ ( superClass1 [, superClass2 ]* ) ]:
        '''Document comment which becomes the __.doc__ attribute for the class'''
        def __init__(self, [param [, param]*):
                '''Document comment for constructor method with self be referencing to the object itself'''
                #__init__body

        # defs of other class methods and assignments to class attributes

# end class MyClass
```

```
"""
File: simple_die.py
Description:  This module defines a six-sided Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._currentRoll = randint(1, 6)

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._currentRoll = randint(1, 6)

    def getRoll(self):
        """Returns the face value of the die."""
        return self._currentRoll

    def __str__(self):
        """Returns the string representation of the die."""
        return str(self._currentRoll)
```

Consider the following script to test the Die class and its associated output:

```
# testDie.py - script to test Die class
from simple_die import Die

die1 = Die()
die2 = Die()
print('die1 =', die1)     #calls __str__
print('die2 =', die2)
print()
print('die1.getRoll() = ', die1.getRoll())
print('die2.getRoll() = ', die2.getRoll())
die1.roll()
print('die1.getRoll() = ', die1.getRoll())
print('str(die1): ' + str(die1))
print('die1 + die2:', die1.getRoll() + die2.getRoll())
```

```
>>>
die1 = 2
die2 = 5

die1.getRoll() =  2
die2.getRoll() =  5
die1.getRoll() =  3
str(die1): 3
die1 + die2: 8
>>>
```

Python Summary Name:_____

Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are *public* by default, unless your identifier starts with a single underscores, e.g, `self._currentRoll`
- all data types are objects, so they can be used as inherited base classes
- **objects are passed by reference when used as parameters to functions**
- all classes have a set of standard methods provided, but may not work properly (__str__, __doc__, etc.)
- most built-in operators (+, -, *, <, >, ==, etc.) can be redefined for a class. This makes programming with objects a lot more intuitive. For example suppose we have two Die objects: `die1` & `die2`, and we want to add up their combined rolls. We could use *accessor methods* to do this:

```
diceTotal = die1.getRoll() + die2.getRoll()
```

  Here, the `getRoll` method returns an integer (type int), so the '+' operator being used above is the one for ints. But, it might be nice to "overload" the + operator by defining an __add__ method as part of the Die class, so the programmer could add dice directly as in:

```
diceTotal = die1 + die2
```

The three most important features of *Object-Oriented Programming* (OOP) to simplify programs and make them maintainable are:

1. *encapsulation* - restricts access to an object's data to access only by its methods
   - ⇨ helps to prevent indiscriminant changes that might cause an invalid object state (e.g., 6-side die with a of roll 8)
2. *inheritance* - allows one class (the *subclass*) to pickup data attributes and methods of other class(es) (the *parents*)
   - ⇨ helps code reuse since the subclass can extend its parent class(es) by adding addition data attributes and/or methods, or overriding (through polymorphism) a parent's methods
3. *polymorphism* - allows methods in several different classes to have the same names, but be tailored for each class
   - ⇨ helps reduce the need to learn new names for standard operations (or invent strange names to make them unique)

Consider using inheritance to extend the Die class to a generalized AdvancedDie class that can have any number of sides. The interface for the AdvancedDie class are:

Detail Descriptions of the AdvancedDie Class Methods

Method	Example Usage	Description
`__init__`	`myDie = AdvancedDie(8)`	Constructs a die with a specified number of sides and randomly rolls it (Default of 6 sides if no argument supplied)
`getRoll`	`myDie.getRoll()`	Returns the current roll of the die (inherited from Die class)
`getSides`	`myDie.getSides()`	Returns the number of sides on the die (did not exist in Die class)
`roll`	`myDie.roll()`	Rerolls the die randomly (By overriding the `roll` method of Die, an AdvancedDie can generate a value based on its # of sides)
`__eq__`	`if myDie == otherDie:`	Allows == operations to work correctly for AdvancedDie objects.
`__lt__`	`if myDie < otherDie:`	Allows < operations to work correctly for AdvancedDie objects.
`__gt__`	`if myDie > otherDie:`	Allows > operations to work correctly for AdvancedDie objects.
`__add__`	sum = myDie + otherDie	Allows the direct addition of AdvancedDie objects, and returns the integer sum of their current roll values.

`__str__`	Directly as:
	`myDie.__str__()`
	`str(myDie)`
	or indirectly as:
	`print myDie`
		Returns a string representation for the AdvancedDie. By overriding the __str__ method of the Die class, so the "print" statement will work correctly with an AdvancedDie.

Consider the following script and associated output:

```
# testAdvancedDie.py - script to test
AdvancedDie class
from advanced_die import AdvancedDie

die1 = AdvancedDie(100)
die2 = AdvancedDie(100)
die3 = AdvancedDie()

print( 'die1 =', die1 )     #calls __str__
print( 'die2 =', die2 )
print( 'die3 =', die3 )

print( 'die1.getRoll() = ', die1.getRoll())
print( 'die1.getSides() =', die1.getSides())
die1.roll()
print( 'die1.getRoll() = ', die1.getRoll())
print( 'die2.getRoll() = ', die2.getRoll())
print( 'die1 == die2:', die1==die2)
print( 'die1 < die2:', die1<die2)
print( 'die1 > die2:', die1>die2)
print( 'die1 != die2:', die1!=die2)
print( 'str(die1): ' + str(die1))
print( 'die1 + die2:', die1 + die2)

help(AdvancedDie)
```

```
die1 = Number of Sides=100 Roll=32
die2 = Number of Sides=100 Roll=76
die3 = Number of Sides=6 Roll=5
die1.getRoll() =  32
die1.getSides() = 100

die1.getRoll() =  70
die2.getRoll() =  76
die1 == die2: False
die1 < die2: True
die1 > die2: False
die1 != die2: True
str(die1): Number of Sides=100 Roll=70
die1 + die2: 146
Help on class AdvancedDie in module
advanced_die:

class AdvancedDie(simple_die.Die)
 |  Advanced die class that allows for
any number of sides
 |
 |  Method resolution order:
 |      AdvancedDie
 |      simple_die.Die
 |      __builtin__.object
 |
 |  Methods defined here:
```

Notice that the `testAdvancedDie.py` script needed to `import AdvancedDie`, but not the `Die` class.

The AdvancedDie class that inherits from the Die superclass.

```
"""
File:  advanced_die.py
Description: Provides a AdvancedDie class that allows for any number of sides
Inherits from the parent class Die in module die_simple
"""
from simple_die import Die
from random import randint

class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""

        Die.__init__(self)       # call Die parent class constructor
        self._numSides = sides
        self._currentRoll = randint(1, self._numSides)

    def roll(self):
        """Causes a die to roll itself -- overrides Die class roll"""
        self._currentRoll = randint(1, self._numSides)

    def __eq__(self, rhs_Die):
        """Overrides default '__eq__' operator to allow for deep comparison of Dice"""
        return self._currentRoll == rhs_Die._currentRoll

    def __lt__(self, rhs_Die):
        """Overrides default '__lt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll < rhs_Die._currentRoll

    def __gt__(self, rhs_Die):
        """Overrides default '__gt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll > rhs_Die._currentRoll

   def __str__(self):
        """Returns the string representation of the AdvancedDie."""
        return 'Number of Sides='+str(self._numSides)+' Roll='+str(self._currentRoll)

    def __add__(self, rhs_Die):
        """Returns the sum of two dice rolls"""
        return self._currentRoll + rhs_Die._currentRoll

    def getSides(self):
        """Returns the number of sides on the die."""
        return self._numSides
```

Data Structures (CS 1520) Lecture 1 Name:_____

1. The `print` function has optional *keyword arguments* which can be listed last that modify it behavior. The `print` function syntax: `print(value,...,sep=' ',end='\n', file=sys.stdout)`

a) Predict the expected output of each of the following.

Program	Expected Output

`print('cat',5,'dog')`
`print()`
`print('cat',5,end='')`
`print(' horse')`
`print('cow')`	

Program	Expected Output
`print ('cat',5,'dog',end='#',sep='23')`	
`print ('cat',5,'dog',sep='23','horse')`	
`print ('cat',5,'dog',sep='>'*3)`	

2. Review of assignment statements. Predict the output of the following programs

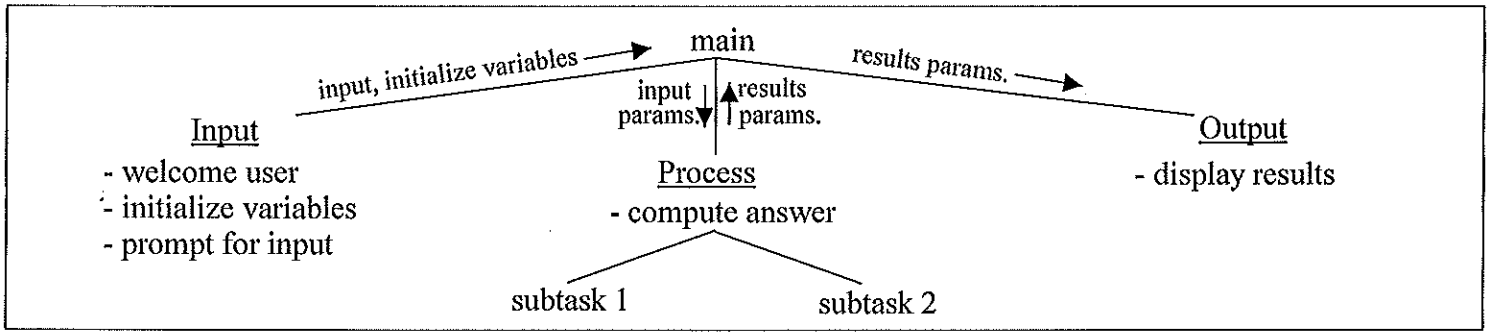
```
a = 123
b = a
a += 1
print ('a is', a)
print ('b is', b)
```

:

```
c = ['cat', 'dog']
d = c
c.append('cow')
print('c is', c)
print('d is', d)
```

```
c = 'cat'
d = c
c += 'fish'
print('c is', c)
print('d is', d)
```

Most simple programs have a similar functional-decomposition design pattern (IPO - Input, Process, Output):




```
""" Simple IPO program to sum a list of numbers. """
def main():
    label, values = getInput()
    total = sum(values)
    displayResults(label, total)

def getInput():
    """ Get label and list of values to sum."""
    label = input("What are we summing? ")
    numberOfValues = int(input("How many values are there? "))
    values = []
    for i in range(numberOfValues):
        values.append(eval(input("Enter the next number: ")))
    return label, values

def displayResults(label, total):
    """ Display sum of values. """
    print("The sum of", label, "values is", total)

main()   # starts the main function running
```

```
What are we summing? money
How many values are there? 4
Enter the next number: 10
Enter the next number: 20
Enter the next number: 30
Enter the next number: 50
The sum of money values is 110
```

3. Design a program to roll two 6-sided dice 1,000 times to determine the percentage of each outcome (i.e., sum of both dice). Report the outcome(s) with the highest percentage.

a) Customize the diagram for the dice problem by briefly describing what each function does and what parameters are passed.

b) An alternative design methodology is to use object-oriented design. For the above dice problem, what objects would be useful and what methods (operations on the objects) should each perform?

Data Structures (CS 1520) Lab 1 Name:_____

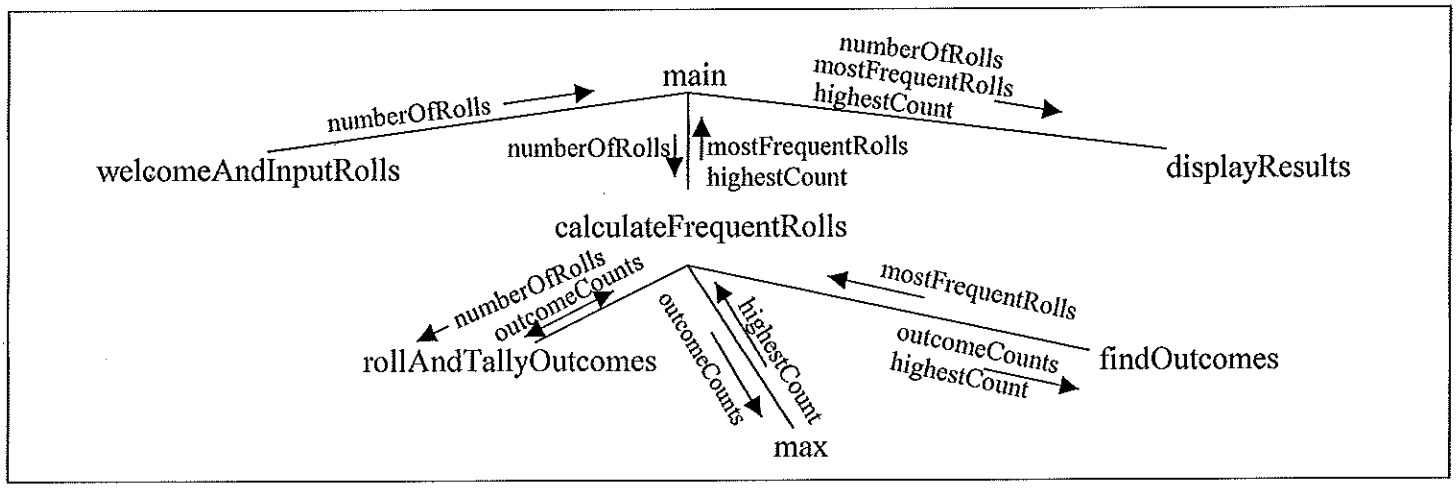
Objective: To practice writing Python code.

To start the lab: Download and unzip the file lab1.zip from
http://www.cs.uni.edu/~fienup/cs1520f18/labs/lab1.zip

Part A: In the folder lab1, open the diceOutcomes.py program in IDLE. (Right-click on diceOutcomes.py | Edit with IDLE) It contains a partial program we started to discuss in class to solve the problem:

> "Write a program to roll two 6-sided dice 1,000 times to determine the percentage of each outcome (i.e., sum of both dice). Report the outcome(s) with the highest percentage."

I decided to functional-decomposition this problem as:



`main` - provides an outline of program by calling top-level functions
`welcomeAndInputRolls` - Displays welcome message for the user. Gets and returns the number of dice rolls from the user.
`calculateFrequentRolls` - Rolls the dice the correct number of times, tallies the outcomes, and returns a list of outcomes with the highest count and highest count.
`rollAndTallyOutcomes` - Rolls the dice the correct number of times and tallies the outcomes. Returns a list of tallies with the index being the outcome.
`max` - built-in function to return the largest item in an iterable data structure like a list.
`findOutcomes` - Returns a list of outcomes with the highest count.
`displayResults` - Displays the outcome(s) with the highest percentage.

Consider running the program with only 10 dice rolls instead of 1,000. The program output with some extra debugging prints showing the two Python lists used: outcomeCounts and mostFrequentRolls.

```
This programs rolls two 6-sided dice many times to
determine the outcome(s) with the highest percentage.
How many times would you like to roll the pair of dice? 10

outcomeCounts: [0, 0, 1, 0, 2, 1, 0, 3, 0, 0, 3, 0, 0]
mostFrequentRolls: [7, 10] and highestCount: 3
The highest percentage is 30.0 for outcome(s): 7 10
```

Your task for lab 1 is to complete the code for the `rollAndTallyOutcomes` and `findOutcomes` functions.

After you have working code, raise your hand and demonstrate your code.

If you complete all parts of the lab, nothing needs to be turned in for this lab. If you do not get done today, then show me the completed lab in next week's lab period. When done, remember save your program to a USB drive, email to yourself, Google drive, etc.

EXTRA CREDIT -- Part B: Rewrite the program using a dictionary instead of a list for the outcomeCounts. Your dictionary will have the outcome for the key and its corresponding tally as its value.

After you have working code, raise your hand and demonstrate your code.

Data Structures (CS 1520) Lecture 2 Name:_____

1. An alternative to functional-decomposition design is to use object-oriented design(OOD). For the following program, what objects would be useful and what methods (operations on the objects) should each support?
"Write a program to roll two 6-sided dice 1,000 times to determine the percentage of each outcome (i.e., sum boths dice). Report the outcome(s) with the highest percentage." (You only need consider the program's OOD)

2. Consider the Die and AdvancedDie classes from the Python Summary handout.

a) What data attributes of AdvancedDie are inherited from the parent Die class?

b) What new data attributes are added as part of the subclass AdvancedDie?

c) Which Die class methods are used directly for an AdvancedDie object?

d) Which Die class methods are redefined/overridden by the AdvancedDie object?

e) Which methods are new to the AdvancedDie class and not in the Die class?

f) If `die1` and `die2` are AdvancedDie objects, then the statement "`if die1 == die2:`" invokes the `__eq__` method of AdvancedDie with `die1` "passed" as `self` and `die2` passed as `rhs_Die`.

```
def __eq__(self, rhs_Die):
    """Overrides default '__eq__' operator to allow for deep comparison of dice"""
    return self._currentRoll == rhs_Die._currentRoll
```

What would the code be for AdvancedDie `__le__` method to allow for the "`if die1 <= die2:`" statement?

g) Good software engineering practice is to include *precondition* and *postcondition* comments on each method/function where the:

- *precondition* - indicates what must be true for the method to work correctly. Typically, the precondition describes the valid values of the parameters. If the precondition is not satisfied, the method does not need to work correctly!
- *postcondition* - describes the expected state after the method has executed

Consider the AdvancedDie constructor:

```
class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""

        Die.__init__(self)  # call Die parent class constructor
        self._numSides = sides
        self._currentRoll = randint(1, self._numSides)
```

What precondition and postcondition comments should we add?

h) If a method/function has a precondition that is not met when invoked (e.g., `die1 = AdvancedDie("six")`), why should the method raise an error?

3. **General "Algorithmic-Complexity Analysis" terminology:**

problem - question we seek an answer for, e.g., "What is the sum of all the items in a list/array?"

parameters - variables with unspecified values

problem instance - assignment of values to parameters, i.e., the specific input to the problem

	0	1	2	3	4	5	6
myList:	5	10	2	15	20	1	11

sum: ?

(number of elements) n: 7

algorithm - step-by-step procedure for producing a solution

basic operation - fundamental operation in the algorithm (i.e., operation done the most) Generally, we want to derive a function for the number of times that the basic operation is performed related to the *problem size.*

problem size - input size. For algorithms involving lists/arrays, the problem size is the number of elements ("n").

Big-oh notation ($O(\ )$) - As the size of a problem grows (i.e., more data), how will our program's run-time grow.

Consider the following `sumList` function.

```
def sumList(myList):
    """Returns the sum of all items in myList"""
    total = 0
    for item in myList:
        total = total + item
    return total
```

a) What is the basic operation of `sumList` (i.e., operation done the most) ?

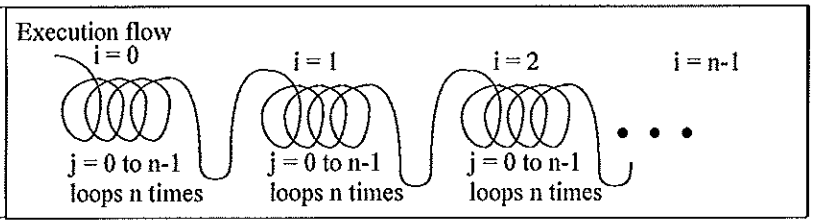
b) What is the problem size of `sumList`?

c) If n is 10000 and `sumList` takes 10 seconds, how long would you expect `sumList` to take for n of 20000?

d) What is the big-oh notation for `sumList`?

4. Consider the following `someLoops` function.

```
def someLoops(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total = total + i + j
    return total
```



a) What is the basic operation of `someLoops` (i.e., operation done the most)?

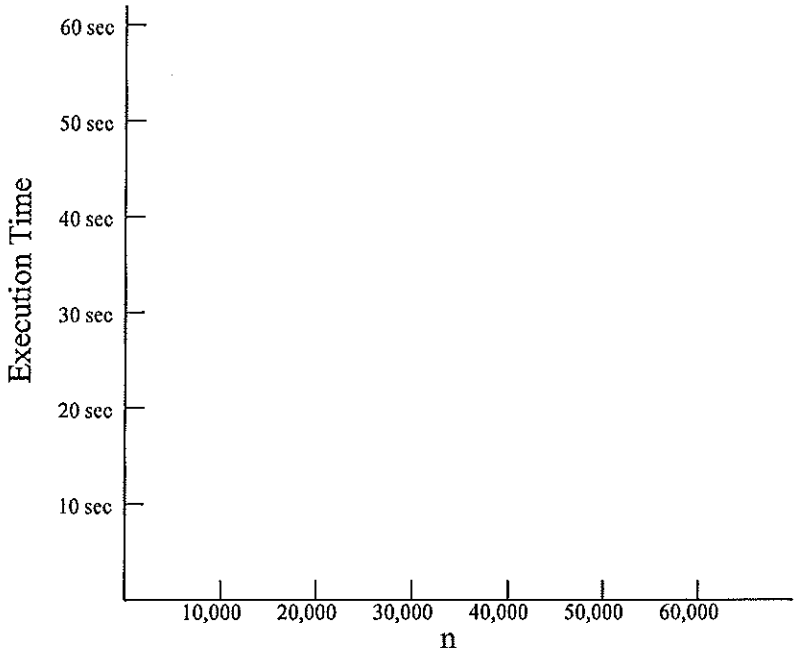
b) How many times will the basic operation execute as a function of n?

c) What is the big-oh notation for `someLoops`?

d) If we input n of 10000 and `someLoops` takes 10 seconds, how long would you expect `someLoops` to take for n of 20000?

Data Structures (CS 1520) Lecture 3 Name:_____

1. Draw the graph for `sumList` ($O(n)$) and `someLoops` ($O(n^2)$) from the previous lecture.



2. Consider the following `sumSomeListItems` function.

```
import time

def main():
    n = eval(input("Enter size of list: "))
    aList = list(range(1, n+1))
    start = time.clock()
    sum = sumSomeListItems(aList)
    end = time.clock()
    print("Time to sum the list was %.9f seconds" % (end-start))

def sumSomeListItems(myList):
    """Returns the sum of some items in myList"""
    total = 0
    index = len(myList) - 1
    while index > 0:
        total = total + myList[index]
        index = index // 2
    return total

main()
```

a) What is the problem size of `sumSomeListItems`?

b) If we input n of 10,000 and `sumSomeListItems` takes 10 seconds, how long would you expect `sumSomeListItems` to take for n of 20,000?

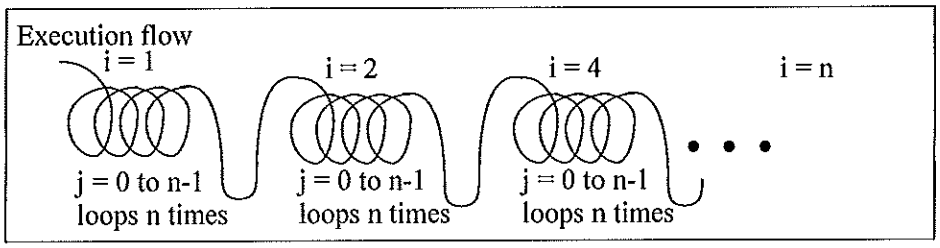
(Hint: For n of 20,000, how many more times would the loop execute than for n of 10,000?)

c) What is the big-oh notation for `sumSomeListItems`?

d) Add the execution-time graph for `sumSomeListItems` to the graph.

3.

```
i = 1
while i <= n:
      for j in range(n):
          # something of O(1)
      # end for
      i = i * 2
# end while
```



a) Analyze the above algorithm to determine its big-oh notation, $O(\ )$.

b) If n of 10,000, takes 10 seconds, how long would you expect the above code to take for n of 20,000?

c) Add the execution-time graph for the above code to the graph.

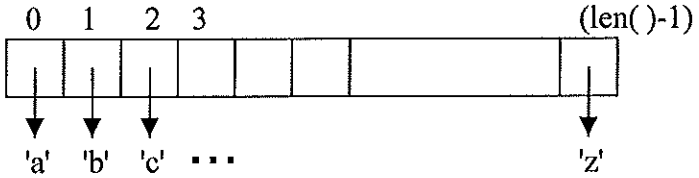
4. Most programming languages have a built-in array data structure to store a collection of same-type items. Arrays are implemented in RAM memory as a contiguous block of memory locations. Consider an array X that contains the odd integers:

address	Memory	
4000	1	X[0]
4004	3	X[1]
4008	5	X[2]
4012	7	X[3]
4016	9	X[4]
4020	11	X[5]
4024	13	X[6]
⋮		

a) Any array element can be accessed randomly by calculating its address. For example, address of X[5] = 4000 + 5 * 4 = 4020. What is the general formula for calculating the address of the ith element in an array?

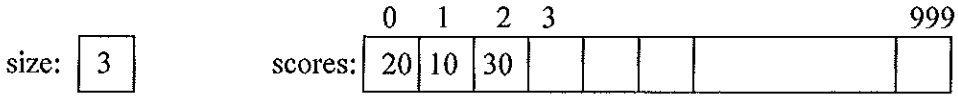
b) What is the big-oh notation for accessing the ith element?

c) A Python list uses an array of references (pointers) to list items in their implementation of a list. For example, a list of strings containing the alphabet:



Since a Python list can contain heterogeneous data, how does storing references in the list aid implementation?

5. Arrays in most HLLs are static in size (i.e., cannot grow at run-time), so arrays are constructed to hold the "maximum" number of items. For example, an array with 1,000 slots might only contain 3 items:



a) The *physical size* of the array is the number of slots in the array. What is the physical size of scores?

b) The *logical size* of the array is the number of items actually in the array. What is the logical size of scores?

c) The *load factor* is faction of the array being used. What is the load factor of scores?

d) What is the $O(\ )$ for "appending" a new score to the "right end" of the array?

e) What is the $O(\ )$ for adding a new score to the "left end" of the array?

f) What is the *average* $O(\ )$ for adding a new score to the array?

g) During run-time if an array fills up and we want to add another item, the program can usually:
- Create a bigger array than the one that filled up
- Copy all the items from the old array to the bigger array
- Add the new item
- Delete the smaller array to free up its memory

When creating the bigger array, how much bigger than the old array should it be?

h) What is the $O(\ )$ of moving to a larger array?

6. Consider the following list methods in Python:

Method	Usage	Average $O(\ )$ for myList containing n items
index []	`itemValue = myList[i]`	
	`myList[i] = newValue`	
append	`myList.append(item)`	
extend	`myList.extend(otherList)`	
insert	`myList.insert(i, item)`	
pop	`myList.pop( )`	
pop(i)	`myList.pop(i)`	
del	`del myList[i]`	
remove	`myList.remove(item)`	
index	`myList.index(item)`	
iteration	`for item in myList:`	
reverse	`myList.reverse( )`	

Dictionary Operations:

Method	Usage	Explanation	Average $O(\ )$ for n keys
get item	`myDictionary.get(myKey)` `value = myDictionary[myKey]`	Returns the value associated with myKey; otherwise *None*	$O(1)$
set item	`myDictionary[myKey]=value`	Change or add myKey:value pair	$O(1)$
in	`myKey in myDictionary`	Returns True if myKey is in myDictionary; otherwise False	$O(1)$
del	`del myDictionary[myKey]`	Deletes the mykey:value pair	$O(1)$

Data Structures (CS 1520) Lab 2 Name:_____

Objective: To get a feel for big-oh notation by analyzing algorithms as well as timing them (Part A), and gain some experience writing Python classes.

Informal Big-oh (and Big-Theta) Definition: As the size of a computational problem grows (i.e., more data), we expect our program to run longer, but this run-time growth is not necessarily linear. Big-oh notation gives us an idea how our program's run-time will grow with respect to its problem size on larger data.

This might seem like a lot of mathematical mumbo-jumbo, but knowing an algorithms big-oh notation can help us predict its run-time on large problem sizes. While running a large size problem, we might want to know if we have time for a quick lunch, a long lunch, a long nap, go home for the day, take a week of vacation, pack-up the desk because the boss will fire you for a slow algorithm, etc.

For example, consider the following algorithm:

```
result = 0
for r in range(n):                          ← loops n times
    for c in range(n):                          ← executes a total of n*n times
        for d in range(n//2):                       ← executes a total of n*n*n/2 times
            result = result + d                         ← executes a total of n*n*n/2 times
        # end for
    # end for
# end for
```

Clearly, the body of the inner-most loop (the "`result = result + d`" statement) will execute $n^3/2$ times, so this algorithm is "big-oh" of n-cubed, $O(n^3)$. Thus, the ***execution-time formula***, T(n), with-respect-to n is:

$$T(n) = c\,n^3 + \text{(slower growing terms)}.$$

For large values of n, the execution time as a function of n, $T(n) \approx c\,n^3$, where c is the *constant of proportionality* on the fastest growing term (the machine dependent time related to how long it takes to execute the inner-most loop once). If we know that T(10,000) = 1 second = c x $10{,}000^3$, then we can calculate c and use it to predict what T(1,000,000). First approximate c as $c \approx T(n) / n^3 = 1 \text{ second} / 10{,}000^3 = 1 \text{ second} / 10^{12} = 10^{-12}$ seconds. Since we are running the algorithm on the same machine, c is unchanged for the larger problem. Thus, $T(1{,}000{,}000) \approx c\ 1{,}000{,}000^3 = c\ 10^{18} = 10^{-12}$ seconds * $10^{18} = 10^6$ seconds or about 11.6 days. (A couple weeks of vacation is appropriate!)

To start the lab: Download and unzip the file lab2.zip from

http://www.cs.uni.edu/~fienup/cs1520f18/labs/lab2.zip

Part A: In the folder lab2, open the timeStuff.py program in IDLE. (Right-click on timeStuff.py | Edit with IDLE) **Start it running** in IDLE by selecting Run | Run Module from the menu. **While it is running**, answer the following questions about each of the algorithms in timeStuff.py.

a) What is the big-oh of Algorithm 0?

```
Algorithm 0:
        result = 0
        for i in range(10000000):
            result = result + i
```

b) What is the big-oh of Algorithm 1?

```
Algorithm 1:
        result = 0
        for i in range(n):
            result = result + i
        # end for
```

c) What is the big-oh of Algorithm 2?

```
Algorithm 2:
        result = 0
        for r in range(n):
            c = n
            while c > 1:
                result = result + c
                c = c // 2
            # end while
        # end for
```

d) What is the big-oh of Algorithm 3?

```
Algorithm 3:
        result = 0
        for r in range(n):
            for c in range(n):
                result = result + c
            # end for
        # end for
```

e) What is the big-oh of Algorithm 4?

```
Algorithm 4:
        result = 0
        for r in range(n):
            for c in range(n):
                for d in range(n*n*n):
                    result = result + d
                # end for
            # end for
        # end for
```

f) What is the big-oh of Algorithm 5?

```
Algorithm 5:
        result = 0
        i = 0
        while i < 2**n:
            result = result + i
            i += 1
        # end while
```

g) Complete the following timing table from the output of timeStuff.py.

Algorithm	Execution Time in Seconds					
	n = 0	**n = 10**	**n = 20**	**n = 30**	**n = 40**	**n = 50**
Algorithm 0						
Algorithm 1						
Algorithm 2						
Algorithm 3						
Algorithm 4						
Algorithm 5				(work on parts h - j while waiting)	These take too long!	

h) For Algorithm 5, use the timing for n = 20 to compute the *constant of proportionality*, c, on the fastest growing term.

i) Using the constant of proportionality computed in (h), predict the run-time of Algorithm 5 for n = 30.

j) How does your prediction in (i) compare to the actual time from (g)?

After you have answered the above questions, raise your hand and explain your answers.

(NOTE: Part B is on the backside of this sheet)

Part B: The `lab2.zip` file also contains:

- A simple `Die` class (in the `simple_die.py` module) for a six-sided die.
- An `AdvancedDie` class (in the module `advanced_die.py` module) for a die which can be constructed with any number of sides. The `AdvancedDie` class inherits from the `Die` class.
- An `averageOutcome.py` program that computes the average outcome (i.e., average total on the pair of dice) on a pair of 10-side dice over 1,000 rolls. Unfortunately, it uses `randint(1,10)+randint(1,10)`.

a) Modify the `averageOutcome.py` program so that it uses the `AdvancedDie` class by:

- creating two 10-sided `AdvancedDie` objects (remember to "`from advanced_die import AdvancedDie`")
- rolls the pair of dice 1,000 times to compute the average outcome

(Note: most of the program will remain unchanged)

Part C:

a) For testing certain dice games, suppose we want to extend the AdvancedDie class to include a new method `setRoll` which takes as a parameter a roll value that is used to set a die's roll to a specified value. We might invoke this method as:

```
myDie.setRoll(3)    # sets myDie's current roll to 3
```

Implement a new subclass MoreAdvancedDie (in a new file `more_advanced_die.py`) which inherits from the AdvancedDie class, and includes the new `setRoll` method. When implementing the MoreAdvancedDie class:

- Include documentation with the MoreAdvancedDie class (comments right below it's `class` line)
- Include documentation with the `setRoll` method including preconditions and postconditions
- Enforce the `setRoll` method's preconditions by raising appropriate exceptions.

(see the AdvancedDie class and its methods for examples of raising exceptions)

b) View the programmer-authored documentation for the MoreAdvancedDie class by typing `help(MoreAdvanceDie)` at the IDLE shell prompt (">>>") after selecting Run | Run Module in the file `more_advanced_die.py` which contains the MoreAdvancedDie class.

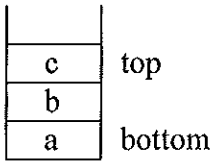
After you have implemented AND fully tested your MoreAdvancedDie class, raise you hand and demonstrate it.

If you complete all parts of the lab, nothing needs to be turned in for this lab. If you do not get done today, then show me the completed lab in next week's lab period. When done save your lab 2 files (USB drive, etc.) and remember to log off.

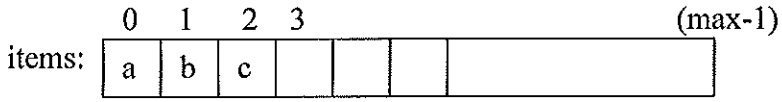
If you have extra time, this would be a good chance to work on Homework #1!

Data Structures (CS 1520) Lecture 4 Name:_____

1. An "abstract" view of the stack:



Using an array implementation would look something like:



top: 2 max: 100

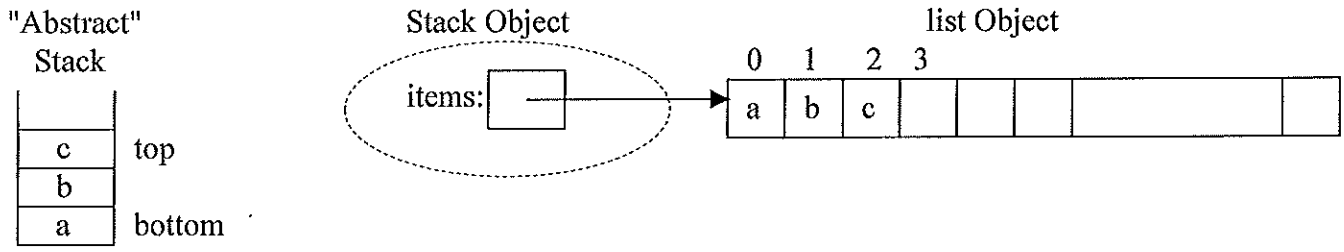
Complete the big-oh notation for the following stack methods assuming an array implementation: ("n" is the # items)

	push(item)	pop( )	peek( )	size( )	isEmpty( )	isFull( )	Constructor
Big-oh							

2. Since Python does not have a (directly accessable) built-in array, we can use a list.

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Since Python uses an array of references (pointers) to list items in their implementation of a list.



a) Complete the big-oh notation for the stack methods assuming this Python list implementation: ("n" is the # items)

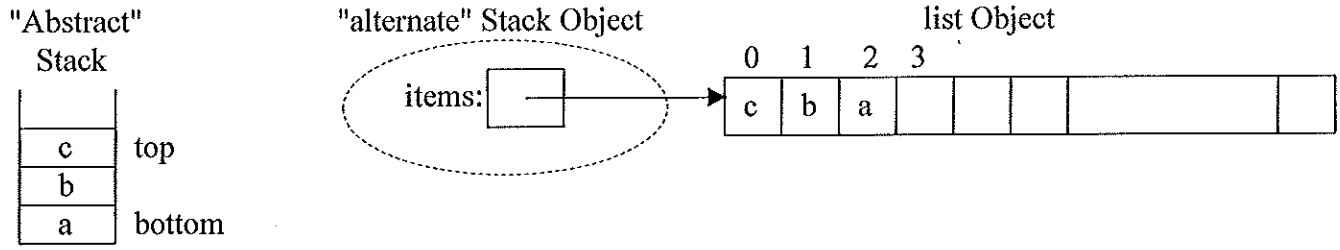
	push(item)	pop( )	peek( )	size( )	isEmpty( )	__init__
Big-oh						

b) Which operations should have what preconditions?

3. The text's alternative stack implementation also using a Python list is:

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0,item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

Since an array is used to implement a Python list, the alternate Stack implementation using a list:



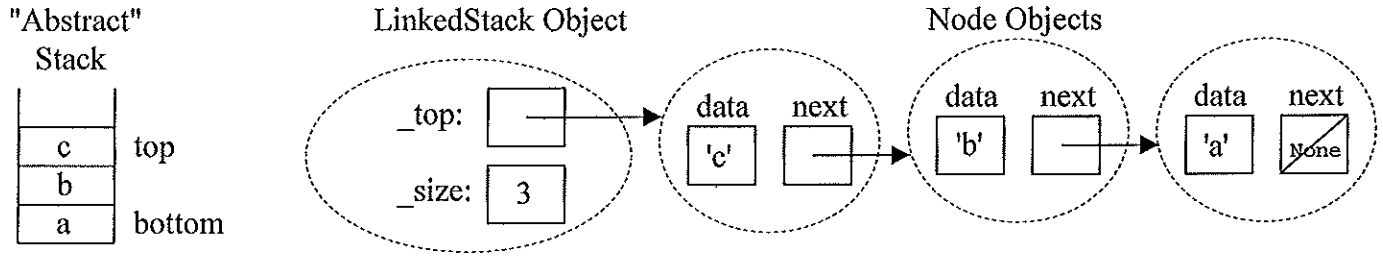
a) Complete the big-oh notation for the "alternate" Stack methods: ("n" is the # items)

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

4. How could we use a stack to check if a word is a palindrome (e.g., radar, toot)?

5. How could we check to see if we have a balanced string of nested symbols? ("((([ ]) { ( ) } [ ])")

1. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. The `LinkedStack` class (in `linked_stack.py`) uses this Node class. Conceptually, a `LinkedStack` object would look like:




```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

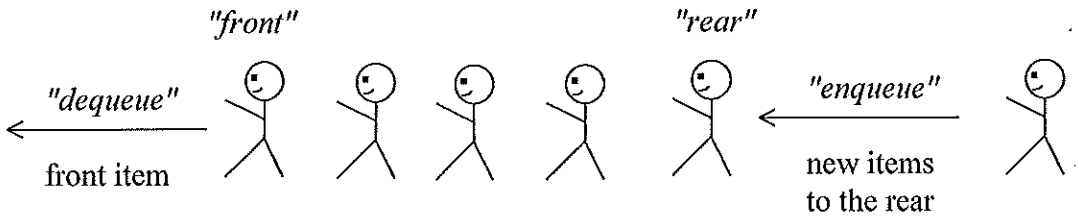
a) Complete the `push`, `pop`, and `__str__` methods.

b) Stack methods big-oh's?
(Assume "n" items in stack)

- constructor `__init__`:
- `push(item)`:
- `pop()`
- `peek()`
- `size()`
- `isEmpty()`
- `str()`

```
class LinkedStack(object):
    """ Link-based stack implementation."""

    def __init__(self):
        self._top = None
        self._size = 0

    def push(self, newItem):
        """Inserts newItem at top of stack."""









    def pop(self):
        """Removes and returns the item at top of the stack.
        Precondition: the stack is not empty."""









    def peek(self):
        """Returns the item at top of the stack.
        Precondition: the stack is not empty."""
        return self._top.getData()

    def size(self):
        """Returns the number of items in the stack."""
        return self._size

    def isEmpty(self):
        return self._size == 0

    def __str__(self):
        """Items strung from top to bottom."""
```

A FIFO *queue* is basically what we think of as a waiting line.



The operations/methods on a queue object, say myQueue are:

Method Call on myQueue object	Description
`myQueue.dequeue()`	Removes and returns the front item in the queue.
`myQueue.enqueue(myItem)`	Adds `myItem` at the rear of the queue
`myQueue.peek()`	Returns the front item in the queue without removing it.
`myQueue.isEmpty()`	Returns `True` if the queue is empty, or `False` otherwise.
`myQueue.size()`	Returns the number of items currently in the queue
`str(myQueue)`	Returns the string representation of the queue

2. Complete the following table by indicating which of the queue operations should have preconditions. Write "none" if a precondition is not needed.

Method Call on myQueue object	Precondition(s)
`myQueue.dequeue()`	
`myQueue.enqueue(myItem)`	
`myQueue.peek()`	
`myQueue.isEmpty()`	
`myQueue.size()`	
`str(myQueue)`	

3. The textbook's Queue implementation use a Python list:

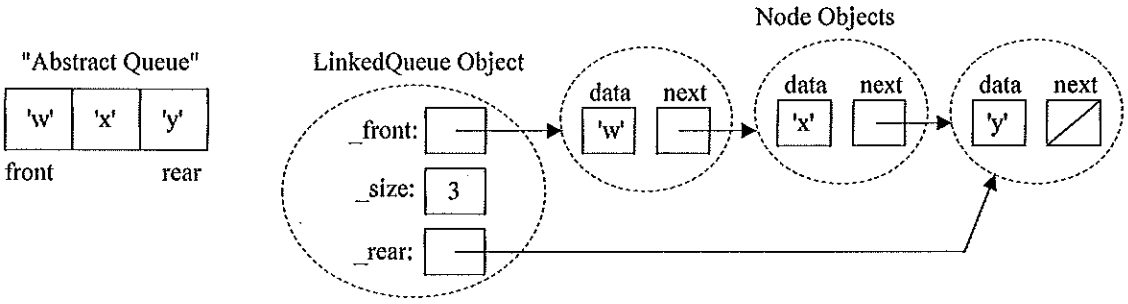
```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def peek(self):


    def size(self):
        return len(self.items)

    def __str__(self):
```

a) Complete the `_peek`, and `__str__` methods

b) What are the Queue methods big-oh's? (Assume "n" items in the queue)

- constructor `__init__`:
- `isEmpty()`
- `enqueue(item)`
- `dequeue()`
- `peek()`
- `size()`
- `str()`

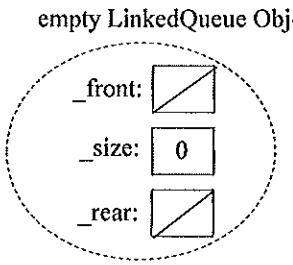
3. An alternate queue implementation using a linked structure (`LinkedQueue` class) would look like:



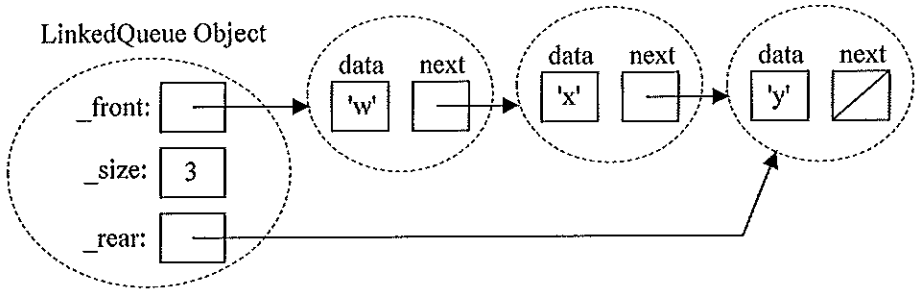
a) Draw on the picture and number the steps for the `enqueue` method of the "normal" case (non-empty queue)

b) Write the `enqueue` method code for the "normal" case:

c) Starting with the empty queue below, draw the resulting picture after your "normal" case code executes.



d) Fix your "normal" case code to handle the "special case" of an empty queue.



e) Draw on the above picture and number the steps for the `dequeue` method of the "normal" case (non-empty queue)

f) Write the `dequeue` method code for the "normal" case:

g) What "special case(s)" does the `dequeue` method code need to handle?

h) Draw the picture for each special case and number the steps for the `dequeue` method in the "special" case(s)

i) Combine the "normal" and special case(s) code for a complete `dequeue` method.

j) Complete the big-oh notation for the `LinkedQueue` methods: ("n" is the # items)

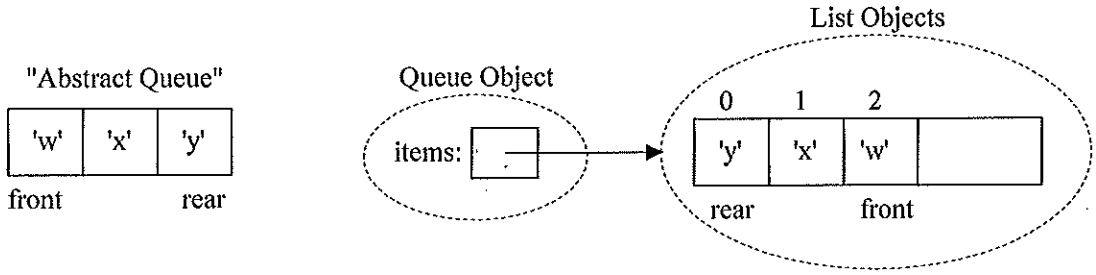
	__init__	enqueue(item)	dequeue( )	peek( )	size( )	isEmpty( )	__str__
Big-oh							

Data Structures (CS 1520) Lab 3 Name:_____

Objective: To understand FIFO (First-In-First-Out) queue implementations in Python including being able to determine the big-oh of each operation.

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs1520f18/labs/lab3.zip

Part A: The textbook's `QueueText` implementation in `lab3/queue_text.py` uses a Python list



a) Complete the big-oh notation for the above `QueueText` implementation: ("n" is the # items)

	__init__	enqueue(item)	dequeue( )	peek( )	size( )	isEmpty( )	__str__
Big-oh							

b) Explain your big-oh answer for enqueue(item).

c) Explain your big-oh answer for dequeue( )

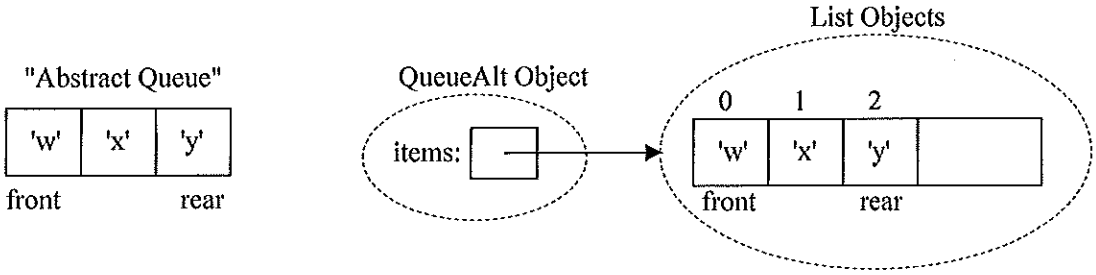
d) Run the timeQueue.py file which times 100,000 enqueues followed by 100,000 dequeues.
Time for 100,000 enqueues: Time for 100,000 dequeues:

e) Why do the enqueues take so much more time?

After answering the above questions, raise you hand and explain your answers.

Part B:

a) Complete the `QueueAlt` implementation in `lab3/queue_alt.py` uses a Python list



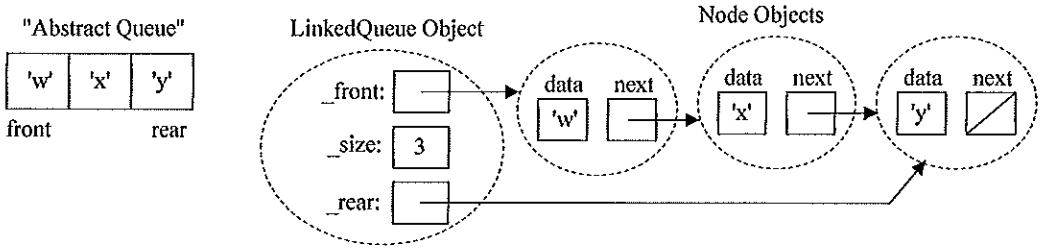
b) Complete the big-oh notation for the above `QueueAlt` implementation: ("n" is the # items)

	__init__	enqueue(item)	dequeue( )	peek( )	size( )	isEmpty( )	__str__
Big-oh							

c) Run the timeQueueAlt.py file which times 100,000 enqueues followed by 100,000 dequeues.
Time for 100,000 enqueues: Time for 100,000 dequeues:

After completing the QueueAlt class, answering the above questions, raise you hand and demonstrate your code.

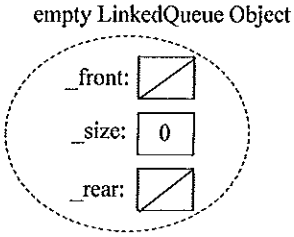
Part C: Consider the `LinkedQueue` implementation in `lab3/linked_queue.py` which uses a linked structure that looks like:



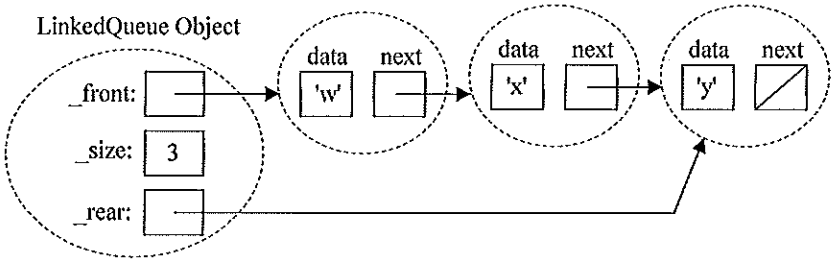
a) Modify the above picture and number the steps for the `enqueue` method's "normal" case (non-empty queue)

b) Complete the `enqueue` method code for the "normal" case in the `lab3/linked_queue.py` file

c) Starting with the empty queue below, draw the resulting picture after your "normal" case code executes.



d) Fix your "normal" case code to handle the "special case" of an empty queue.



e) Modify the above picture and number the steps for the `dequeue` method's "normal" case (non-empty queue)

f) Complete the `dequeue` method code for the "normal" case in the `lab3/linked_queue.py` file

g) What "special case(s)" does the `dequeue` method code need to handle?

h) Complete the big-oh notation for the `LinkedQueue` methods: ("n" is the # items)

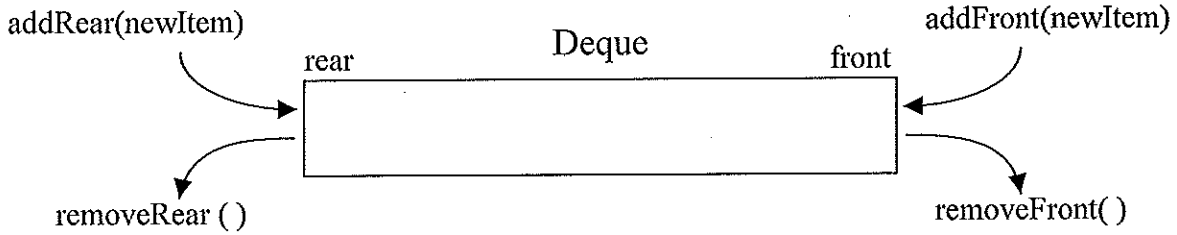
	__init__	enqueue(item)	dequeue( )	peek( )	size( )	isEmpty( )	__str__
Big-oh							

i) Run the `timeLinkedQueue.py` file which times 100,000 enqueues followed by 100,000 dequeues.
Time for 100,000 enqueues: Time for 100,000 dequeues:

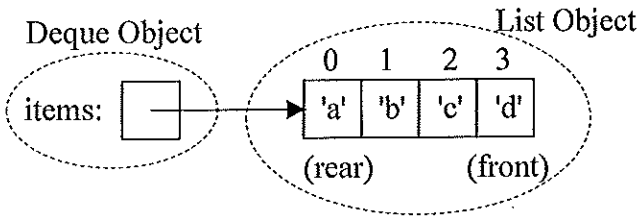
After thoroughly testing your linked implementation, raise you hand and demonstrate your queue.

Data Structures (CS 1520) Lecture 6 Name:_____

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.



1. One possible implementation of a Deque would be to use a Python list to store the Deque items such that
- the rear item is **always stored at index 0**,
- the front item is always stored at the highest index (or -1)




```
class Deque(object):
   def __init__(self):
```

a) Complete the `__init__` method and determine the big-oh, $O(\ )$, for each Deque operation, assuming the above implementation. Let n be the number of items in the Deque.

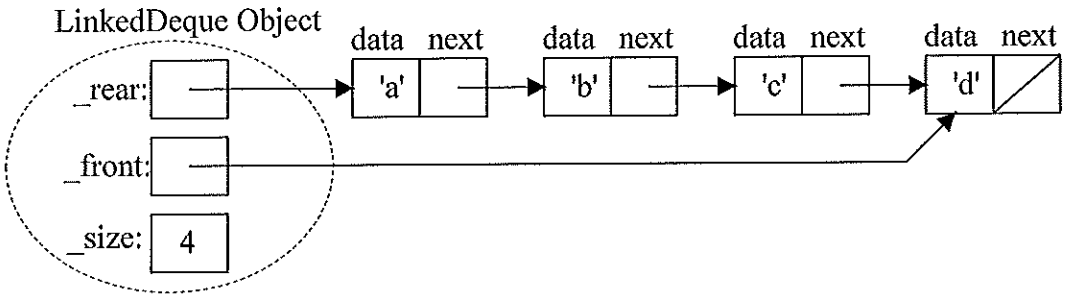
isEmpty	addFront	removeFront	addRear	removeRear	size

b) Write the methods for the `addRear` and `removeRear` operation.

```
def addRear(self, newItem):
```

```
def removeRear(self):
```

2. An alternative implementation of a Deque would be a linked implementation as in:




```
class LinkedDeque(object):
   def __init__(self):
```

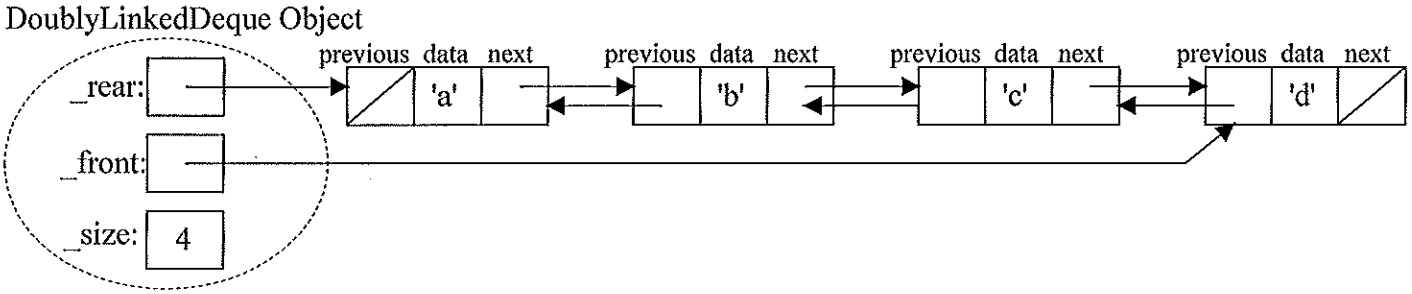
a) Complete the `__init__` method and determine the big-oh, $O(\ )$, for each Deque operation assuming the above linked implementation. Let n be the number of items in the Deque.

isEmpty	addFront	removeFront	addRear	removeRear	size

b) Suggest an improvement to the above linked implementation of the Deque to speed up some of its operations.

```
from node import Node

class Node2Way(Node):
    def __init__(self,initdata):
        Node.__init__(self, initdata)
        self.previous = None

    def getPrevious(self):
        return self.previous

    def setPrevious(self,newprevious):
        self.previous = newprevious
```

3. An alternative implementation of a Deque would be a doubly-linked implementation as in:



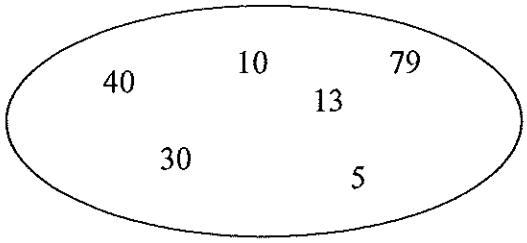
a) Determine the big-oh, *O* ( ), for each Deque operation assuming the above doubly-linked implementation. Let n be the number of items in the Deque.

isEmpty	addFront	removeFront	addRear	removeRear	size

4. A *priority queue* has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a proirity that determines the order they are removed. A hospital emergence room operates like a priority queue -- the person with the most serious injure has highest priority even if they just arrived.

a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next?

priority queue:



b) To implement a priority queue, we could use an **unorder Python list**. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

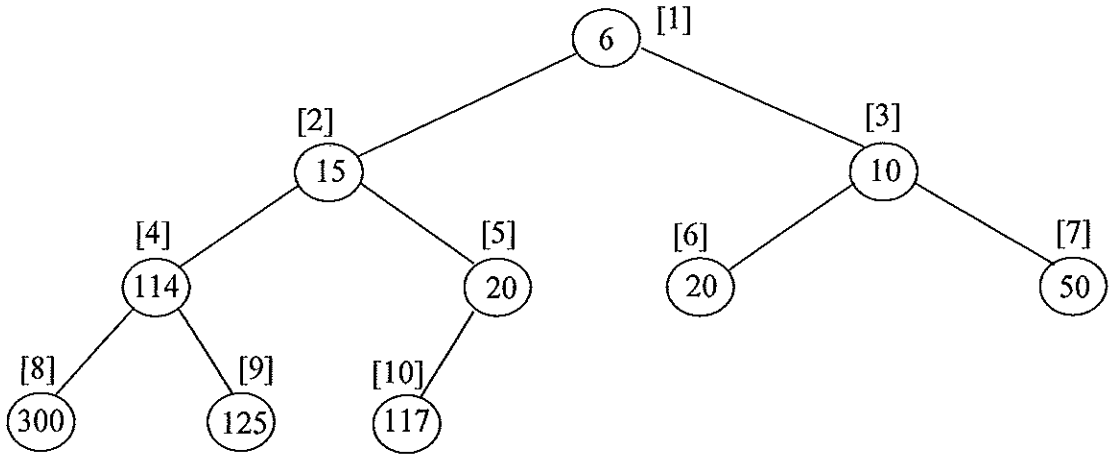
- enqueue:
- dequeue:

c) To implement a priority queue, we could use a **Python list order by priorities** in decending order. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue:
- dequeue

Data Structures (CS 1520) Lecture 7 Name:_____

1. Section 6.6 discusses a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array 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" an 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	6	15	10	114	20	20	50	300	125	117

a) For the above heap, the list/array 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) What would the above heap look like after inserting 13 and then 3? (show the changes on above tree)

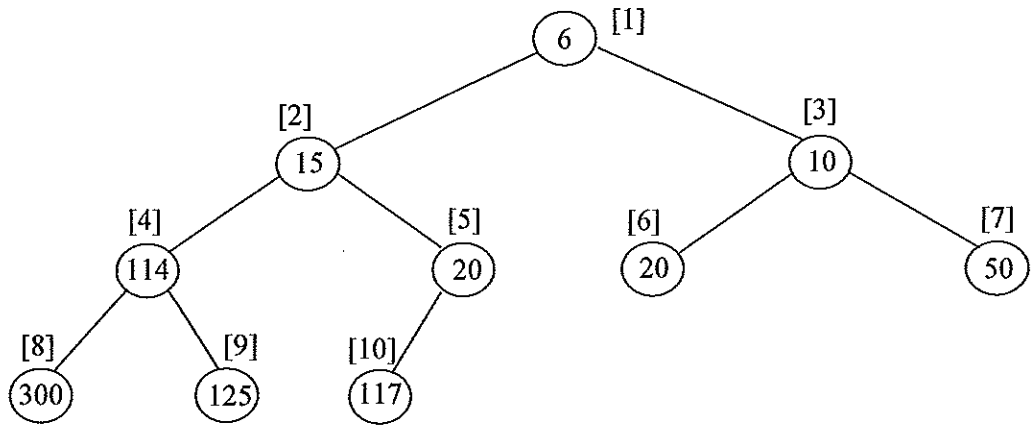
General Idea of insert(newItem):
- append newItem to the end of the list (easy to do, but violates heap-order property)
- restore the heap-order property by repeatedly swapping the newItem with its parent until it *percolates* to correct spot

c) What is the big-oh notation for inserting a new item in the heap?

d) Complete the code for the percUp method used by insert.

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self,currentIndex):
        parentIndex =
        while









    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
```

2. Now let us 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	6	15	10	114	20	20	50	300	125	117

a) What item would `delMin` remove and return from the above heap?

b) What is the quickest way to fill the hole left by `delMin`?

c) What new problem does this cause?

General Idea of `delMin()`:

- remember the minimum value so it can be returned later (easy to find - at index 1)
- copy the last item in the list to the root, delete it from the right end, decrement size
- restore the heap-order property by repeatedly swapping this item with its smallest child until it *percolates down* to the correct spot
- return the minimum value

d) What would the above heap look like after `delMin`? (show the changes on above tree)

e) Complete the code for the percDown method used by `delMin`.

```
class BinHeap:

            .
            .
            .

  def minChild(self,i):
     if i * 2 + 1 > self.currentSize: # if only left child
         return i * 2
     else:
         if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
             return i * 2
         else:
             return i * 2 + 1

  def delMin(self):
     retval = self.heapList[1]
     self.heapList[1] = self.heapList[self.currentSize]
     self.currentSize = self.currentSize - 1
     self.heapList.pop()
     self.percDown(1)
     return retval
```

```
  def percDown(self,currentIndex):
```

f) What is the big-oh notation for `delMin`?

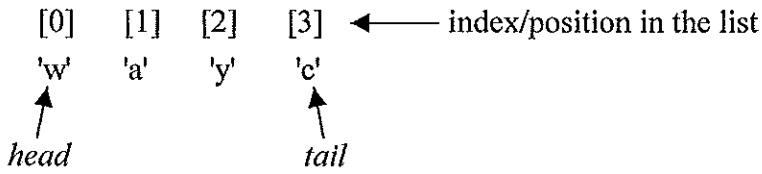
Once we have a working `BinHeap`, then implementing the `PriorityQueue` class using a `BinHeap` is a piece of cake:

```
### File: priority_queue.py
from binheap import BinHeap

class PriorityQueue:
    def __init__(self):
        self._heap = BinHeap()

    def isEmpty(self):
        return self._heap.isEmpty()

    def enqueue(self, item):
        self._heap.insert(item)

    def dequeue(self):
        return self._heap.delMin()

    def size(self):
        return self._heap.size()

    def __str__(self):
        return str(self._heap)
```

```
>>> q = PriorityQueue()
>>> print(q)
[]
>>> q.enqueue(5)
>>> q.enqueue(1)
>>> q.enqueue(7)
>>> print(q)
[1, 5, 7]
>>> q.dequeue()
1
>>> print(q)
[5, 7]
```

3. A "list" is a generic term for a sequence of items in a linear arrangement. Unlike stacks, queues and deques access to list items is not limited to either end, but can be from any position in the list. The general terminology of a list is illustrated by:

"Abstract view of a list"



There are three broad categories of list operations that are possible:

- **index-based operations** - the list is manipulated by specifying an index location, e.g.,
  myList.insert(3, item) # insert item at index 3 in myList
- **content-based operations** - the list is manipulated by specifying some content (i.e., item value), e.g.,
  myList.remove(item) # removes the item from the list based on its value
- **cursor-base operations** - a *cursor* (current position) can be moved around the list, and it is used to identify list items to be manipulated, e.g.,
  myList.first() # sets the cursor to the head item of the list
  myList.next() # moves the cursor one position toward the tail of the list
  myList.remove() # deletes the second item in the list because that's where the cursor is currently located

The following table summarizes the operations from the three basic categories on a list, L:

Index-based operations	Content-based operations	cursor-based operations

`L.insert(index, item)`
`item = L[index]`
`L[index] = newValue`
`L.pop(index)`
	`L.add(item)`
	`L.remove(item)`
	`L.search(item) #return Boolean`
	`i = L.index(item)`
		`L.hasNext()`
		`L.next()`
		`L.hasPrevious()`
		`L.previous()`
		`L.first()`
		`L.last()`
		`L.insert(item)`
		`L.replace(item)`
		`L.remove()`

Built-in Python lists are unordered with a mixture of index-based and content-based operations. We know they are implemented using a contiguous block of memory (i.e., an array). The textbook talks about an unordered list ADT, and a sorted list ADT which is more content-based. Both are implemented using a singly-linked list.

a) Why would a singly-linked list be a bad choice for implementing a cursor-based list ADT?

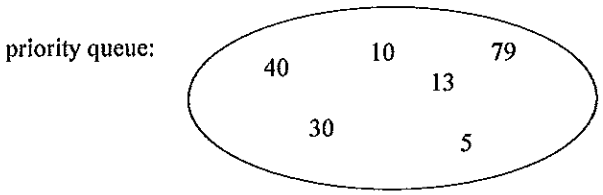
Data Structures (CS 1520) Lab 4 Name:_____

Objective: To understand priority queue implementations in Python including being able to determine the big-oh of each operation.

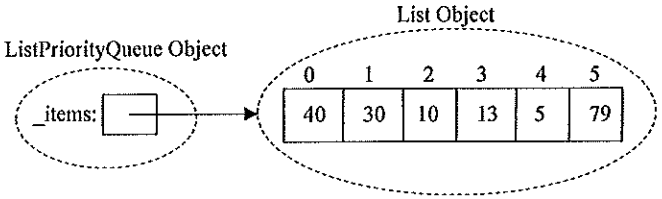
To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs1520f18/labs/lab4.zip

Part A:

a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next?



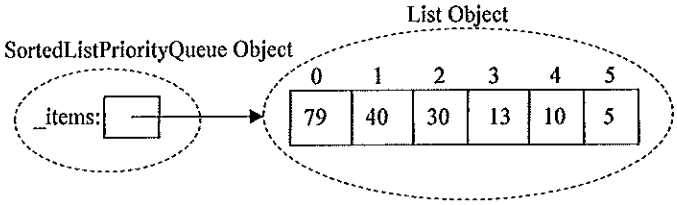
b) The `ListPriorityQueue` implementation in `lab4/list_priority_queue.py` uses an **unorder Python list.**



What would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue:
- dequeue:

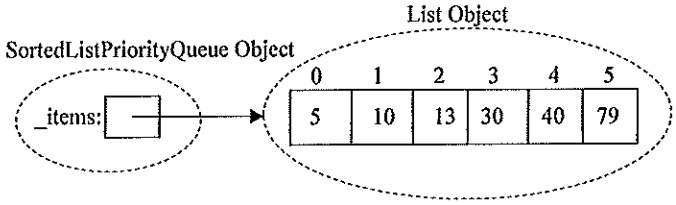
c) The `SortedListPriorityQueue` implementation in `lab4/sorted_list_priority_queue.py` uses a **Python list order by priorities** in decending order.



What would be the big-oh notation for each of the following methods: (justify your answer)

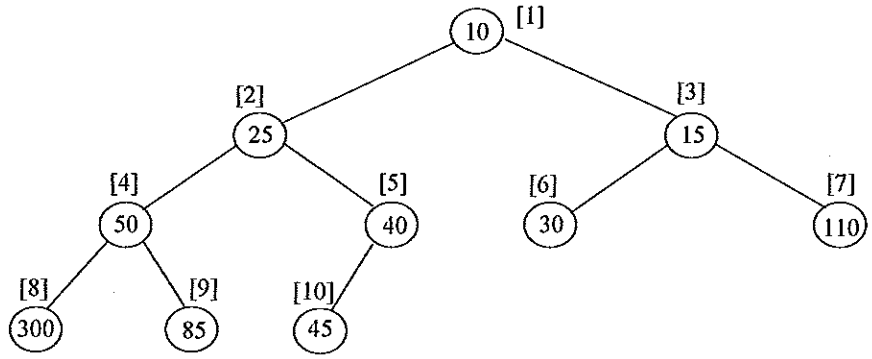
- enqueue:
- dequeue

d) Why would it be a bad idea to implement a priority queue using a **Python list order by priorities** in reverse (ascending) order? (HINT: What is the big-oh notations for enqueue and dequeue?)



Answer the above questions, then raise your hand. Explain your answers to an instructor or TA.

Part B: (Lecture 7 and) Section 6.6 discusses a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array 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" an 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	10	25	15	50	40	30	110	300	85	45

a) For the above heap, the list/array 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:

Recall the General Idea of insert(newItem):

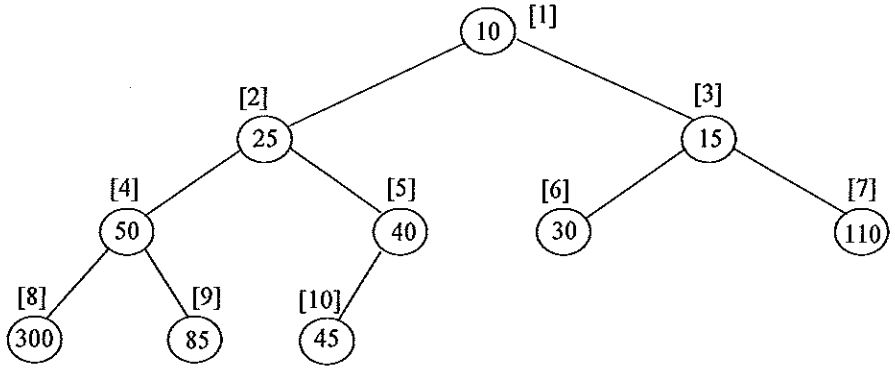
- append newItem to the end of the list (easy to do, but violates heap-order property)
- restore the heap-order property by repeatedly swapping the newItem with its parent until it *percolates up* to the correct spot

b) What would the above heap look like after inserting 18 and then 27? (show the changes on above tree)

c) What is the big-oh notation for inserting a new item in the heap?

Now let us consider the `delMin` operation that removes and returns the minimum item. Recall the General Idea of `delMin()`:

- remember the minimum value so it can be returned later (easy to find - at index 1)
- copy the last item in the list to the root, delete it from the right end, decrement size
- restore the heap-order property by repeatedly swapping this item with its smallest child until it *percolates down* to the correct spot
- return the minimum value



d) What would the above heap look like after `delMin`? (show the changes on above tree)

Answer the above questions, then raise your hand. Explain your answers to an instructor or TA.

Part C: Run the `lab4/timePriorityQueues.py` program that enqueues 20,000 random integers followed by dequeueing all 20,000 integers from various priority queues discussed above. Complete the following timing table from the output of `timePriorityQueues.py`.

Priority Queue Implementation	Execution Time in Seconds	
	Enqueuing 20,000 Random ints	**Dequeuing 20,000 ints**
Unsorted Python list		
Sorted Python list in descending order		
"Reverse" sorted Python list in ascending order		
Binary heap stored in a Python list		

b) Why does it take more time to enqueue 20,000 items in the "unsorted" Python list version than dequeue 20,000 in the sorted Python list version?

c) Why does it take more time to dequeue 20,000 items in the heap version than enqueue 20,000 in the heap version?

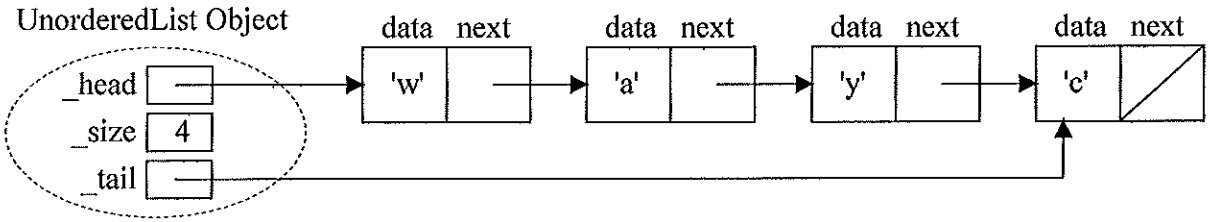
d) Why is the heap implementation of the priority queue considered "better" than the other three?

After you have answered the above questions, raise your hand and explain your answers.

If you have extra time, complete previous labs (1 to 3) or work on homework!

Data Structures (CS 1520) Lecture 8 Name:_____

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



a) The `search(targetItem)` method searches for `targetItem` in the list. It returns `True` if `targetItem` is in the list; otherwise it returns `False`. Complete the `search(targetItem)` method code:

```
class UnorderedList:

   def search(self, targetItem):
```

b) The textbook's unordered list ADT **does not** allow duplicate items, so operations `add(item)`, `append(item)`, and `insert(pos, item)` would have what precondition?

c) Complete the `append(item)` method including a check of it's precondition(s)?

```
   def append(self, item):
```

d) Why do you suppose I added a `_tail` attribute?

e) The textbook's `remove(item)` and `index(item)` operations "Assume the item is present in the list." Thus, they would have a precondition like "Item is in the list." When writing a program using an UnorderedList object (say `myGroceryList = UnorderedList()`), how would the programmer check if the precondition is satisfied?

```
itemToRemove = input("Enter the item to remove from the Grocery list: ")
if
   myGroceryList.remove(itemToRemove)
```

f) The `remove(item)` and `index(item)` methods both need to look for the `item`. What is inefficient in this whole process?

g) Modify the `search(targetItem)` method code in (a) to set additional data attributes to aid the implementation of the `remove(item)` and `index(item)` methods.

h) Write the `index(item)` method including a check of its precondition(s).

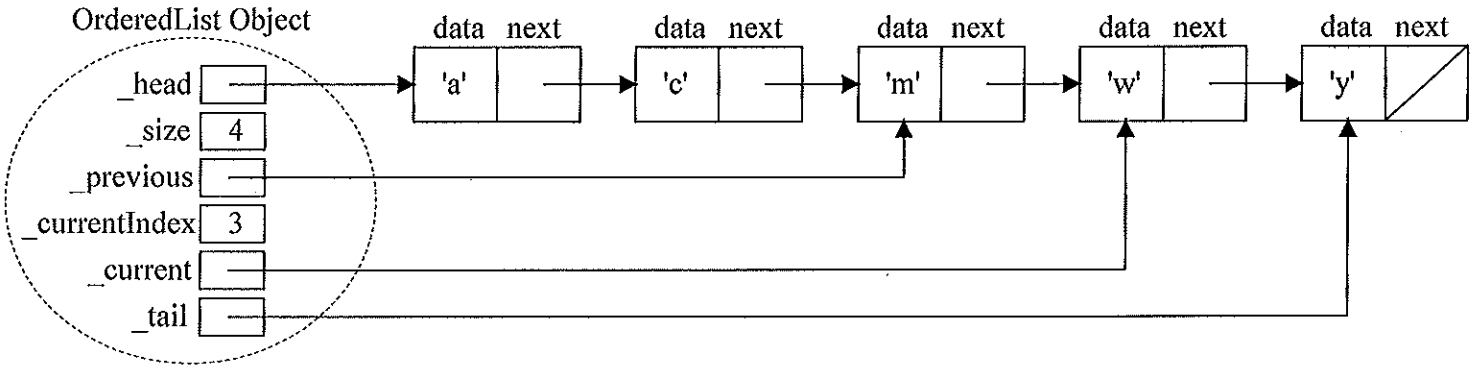
```
def index(self, item):
```

i) Write the `remove(item)` method including a check of its precondition(s).

```
def remove(self, item):
```

Data Structures (CS 1520) Lecture 9 Name:_____

1. The textbook's **ordered list** ADT uses a singly-linked list implementation. I added the `_size`, `_tail`, `_current`, `_previous`, and `_currentIndex` attributes:



The `search(targetItem)` method searches for `targetItem` in the list. It returns `True` if `targetItem` is in the list; otherwise it returns `False`. Additionally, it has the side-effects of setting `_current`, `_previous`, and `_currentIndex`. The complete `search(targetItem)` method code for the `OrderedList` is:

```
class OrderedList:

   def search(self, targetItem):
       if self._current != None and self._current.getData() == targetItem:
          return True

       self._previous = None
       self._current = self._head
       self._currentIndex = 0
       while self._current != None:
           if self._current.getData() == targetItem:
              return True
           elif self._current.getData() > targetItem:
              return False
           else: #inch-worm down list
              self._previous = self._current
              self._current = self._current.getNext()
              self._currentIndex += 1
       return False
```

a) What's the purpose of the "`elif self._current.getData() > targetItem:`" check?

Consider the `add(item)` method with the precondition: `item` is not in the list.

b) Write the precondition check at the start of the `add(item)` method.

c) Suppose you are adding the item value of 's'. Update the above picture for this "normal" case, and number the steps in the drawing.

d) What special cases need to be considered for the add method?

2. A *recursive function* is one that calls itself. Complete the recursive code for the `countDown` function that is passed a starting value and proceeds to count down to zero and prints "Blast Off!!!".

Hint: The `countDown` function, like most recursive functions, solves a problem by splitting the problem into one or more simpler problems of the same type. For example, `countDown(10)` prints the first value (i.e, 10) and then solves the simpler problem of counting down from 9. To prevent "infinite recursion", if-statement(s) are used to check for trivial *base case*(s) of the problem that can be solved without recursion. Here, when we reach a `countDown(0)` problem we can just print "Blast Off!!!".

```
""" File:  countDown.py """

def main():
    start = eval(input("Enter count down start: "))
    print("\nCount Down:")
    countDown(start)

def countDown(count):









main()
```

Program Output:

```
Enter count down start: 10

Count Down:
10
9
8
7
6
5
4
3
2
1
Blast Off!!!
```

a) Trace the function call `countDown(5)` on paper by drawing the run-time stack and showing the output.

b) What do you think will happen if your call `countDown(-1)`?

c) Why is there a limit on the depth of recursion?

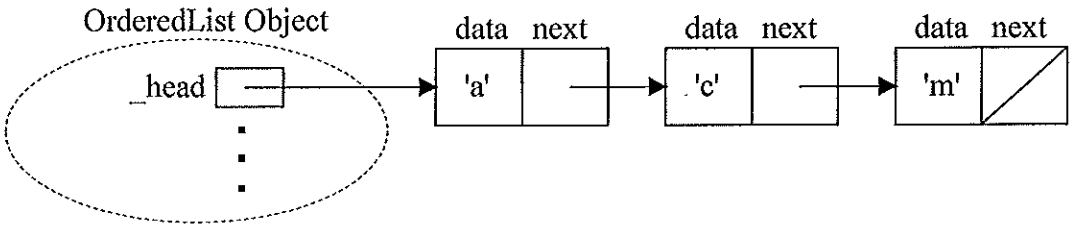
3. The non-recursive __str__ method for `OrderedList` object below would return: `"(head) a c m (tail)"`

```
def __str__(self):
    resultStr = "(head) "
    current = self._head
    while current != None:
        resultStr += str(current.getData()) + " "
        current = current.getNext()
    return resultStr + "(tail)"
```

We can thing of building the string for the list as "a " + (string for the rest of the list)

a) Complete the recursive `strHelper` function in the __str__ method for our `OrderedList` class.

```
    def __str__(self):
        """ Returns a string representation of the list with a space between each item. """

        def strHelper(current):









    # Start of __str__ method execution
    return "(head) " + strHelper(self._head) + "(tail)"
```



4. Some mathematical concepts are defining by recursive definitions. One example is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

$Fib_0 = 0$

$Fib_1 = 1$

$Fib_N = Fib_{N-1} + Fib_{N-2}$ for $N \geq 2$.

a) Complete the recursive function: `def fib (n):`

b) Draw the *call tree* for fib(5).

c) On my office computer, the call to fib(40) takes 22 seconds, the call to fib(41) takes 35 seconds, and the call to fib(42) takes 56 seconds. How long would you expect fib(43) to take?

d) How long would you guess calculating fib(100) would take on my office computer?

e) Why do you suppose this recursive fib function is so slow?

f) What is the computational complexity? $O(\quad)$

g) How might we speed up the calculation of the Fibonacci series?

5. A VERY POWERFUL concept in Computer Science is *dynamic programming*. Dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating them.

We can use a list to store the answers to smaller problems of the Fibonacci sequence.
To transform from the recursive view of the problem to the dynamic programming solution you can do the following steps:

1) Store the solution to smallest problems (i.e., the base cases) in a list
2) Loop (no recursion) from the base cases up to the biggest problem of interest. On each iteration of the loop we:
   - solve the next bigger problem by looking up the solution to previously solved smaller problem(s)
   - store the solution to this next bigger problem for later usage so we never have to recalculate it

a) Complete the dynamic programming code:

```
def fib(n):
    """Dynamic programming solution to find the nth number in the Fibonacci seq."""

    # List to hold the solutions to the smaller problems
    fibonacci = []

    # Step 1:  Store base case solutions
    fibonacci.append(    )
    fibonacci.append(    )

    # Step 2:  Loop from base cases to biggest problem of interest
    for position in range(                                ):

        fibonacci.append(                                                    )

    # return nth number in the Fibonacci sequence
    return
```

Running the above code to calculate fib(100) would only take a fraction of a second.

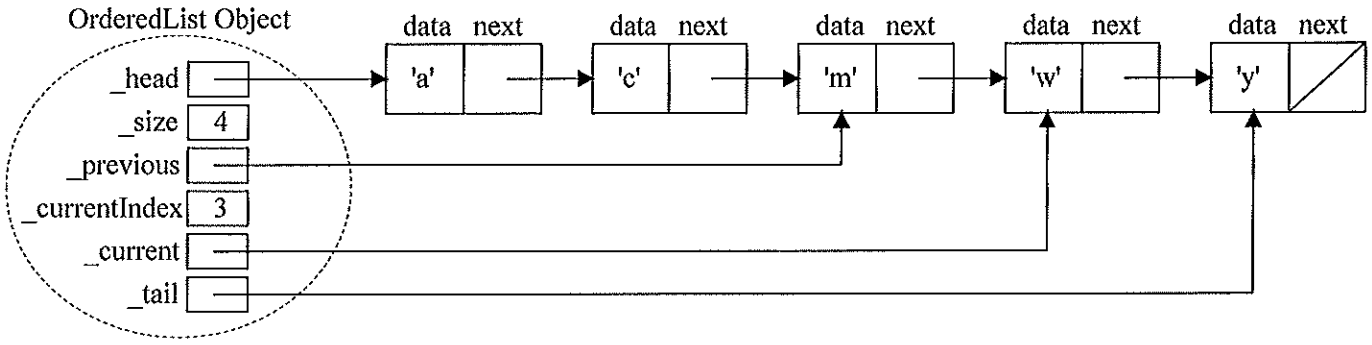
b) One tradeoff of simple dynamic programming implementations is that they can require more memory since we store solutions to **all** smaller problems. Often, we can reduce the amount of storage needed if the next larger problem (and all the larger problems) don't really need the solution to the really small problems, but just the larger of the smaller problems. In fibonacci when calculating the next value in the sequence how many of the previous solutions are needed?

Data Structures (CS 1520) Lab 5 Name:_____

Objective: To understand recursion by writing simple recursive solutions.

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs1520f18/labs/lab5.zip

Part A: Recall: We modified the textbook's ordered list ADT that uses a singly-linked list implementation by adding the `_size`, `_tail`, `_current`, `_previous`, and `_currentIndex` attributes:




```
##    NON-RECURSIVE CODE WE ARE REPLACING
 def search(self, targetItem):
    if self._current != None and \
        self._current.getData() == targetItem:
          return True

      self._previous = None
      self._current = self._head
      self._currentIndex = 0
      while self._current != None:
          if self._current.getData() == targetItem:
              return True
          elif self._current.getData() > targetItem:
              return False
          else: #inch-worm down list
              self._previous = self._current
              self._current = self._current.getNext()
              self._currentIndex += 1
      return False
```

```
def search(self, targetItem):
   def searchHelper():
      """ Recursive helper function that moves down the linked list.
          It has no parameters, but uses self._current, self._previous,
          self._currentIndex."""
       # ADD CODE HERE




    # START OF SEARCH - DO NOT MODIFY BELOW CODE
    if self._current != None and \
       self._current.getData() == targetItem:
        return True

    self._previous = None
    self._current = self._head
    self._currentIndex = 0
    return searchHelper()  # Returns the result of searchHelper
```

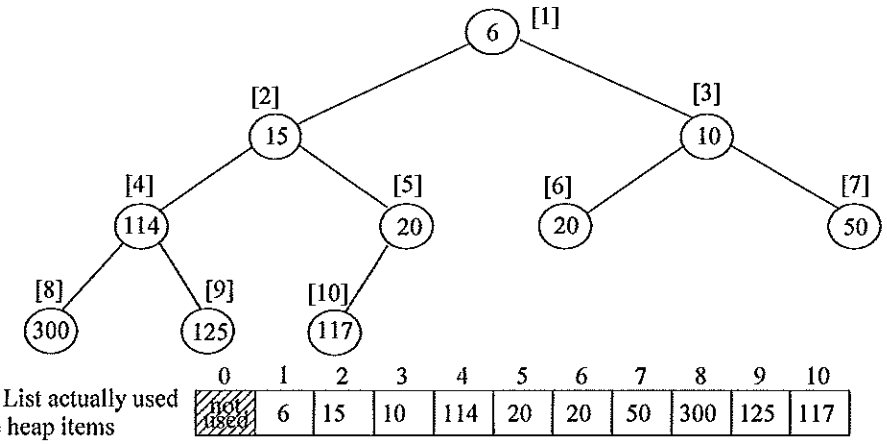
a) What are the base case(s) for the `searchHelper` that **halt** the while-loop of the non-recursive search code?

b) What are the recursive case(s) for the `searchHelper` that replaces the while-loop of the non-recursive search code?

c) Complete the recursive `searchHelper` function in the `search` method of our `OrderedList` class in `ordered_linked_list.py`. Test it with the `listTester.py` program.

Raise your hand when done. Demonstrate and explain your code to an instructor.

Part B: Recall that Lecture 7 and Section 6.6 discussed a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array 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" an 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	6	15	10	114	20	20	50	300	125	117

Recall the General Idea of `insert(newItem)`:

- append newItem to the end of the list (easy to do, but violates heap-order property)
- restore the heap-order property by repeatedly swapping the newItem with its parent until it *percolates up* to the correct spot

Recall the General Idea of `delMin()`:

- remember the minimum value so it can be returned later (easy to find - at index 1)
- copy the last item in the list to the root, delete it from the right end, decrement size
- restore the heap-order property by repeatedly swapping this item with its smallest child until it *percolates down* to the correct spot
- return the minimum value

Originally, we used iteration (i.e., a loop) to percolate up (see `percUp`) and percolate down (see `percDown`) the tree. (textbook code below)

```
##    NON-RECURSIVE CODE WE ARE REPLACING

def percUp(self,i):
    while i // 2 > 0:
        if self.heapList[i] < self.heapList[i//2]:
           tmp = self.heapList[i // 2]
           self.heapList[i // 2] = self.heapList[i]
           self.heapList[i] = tmp
        i = i // 2
```

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc
```

For part B, I want you to complete the recursive `percUpRec` and recursive `percDownRec` methods in `binHeap.py`. Run the `binHeap.py` file to test your code.

Raise your hand when done. Demonstrate and explain your code to an instructor.

(If you have extra time, work on previous labs or homeworks!)

1. Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the fewest number of coins for this amount of change.

a) What "greedy" algorithm would you use to solve this problem with US coin types of {1, 5, 10, 25, 50} and a change amount of 29-cents?

b) Do you get the correct solution if you use this algorithm for coin types of {1, 5, 10, 12, 25, 50} and a change amount of 29-cents?

2. One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.
Solve a problem by:

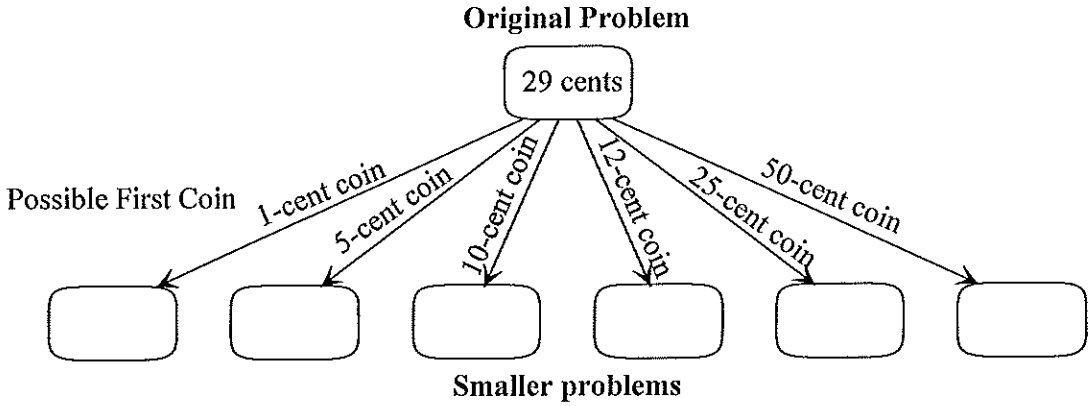
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

a) For the coin-change problem, what determines the size of the problem?

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

3. After we give back the first coin, which smaller amounts of change do we have?

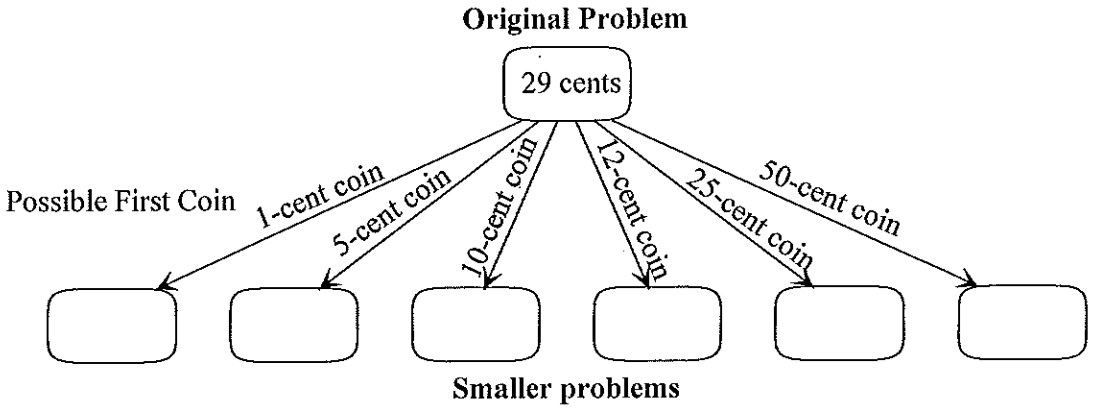


4. If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

5. Complete a recursive relationship for the fewest number of coins.

$$\text{FewestCoins(change)} = \begin{cases} \min_{\text{coin} \in \text{CoinSet and coin} \leq \text{change}} (\text{FewestCoins(}\quad\text{)}) + & \text{if change} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

6. Complete a couple levels of the recursion tree for 29-cents change using the set of coins {1, 5, 10, 12, 25, 50}.



Test 1 will be Thursday September 27th in class. It will be closed-book and notes, except for one 8.5" x 11" sheet of paper (you can use front and back) containing any notes that you want AND the Python Summary handout. The test will cover the following topics (and maybe more).

Chapter 1. Introduction
Python control structures: if, while, for,
Python built-in data structures: list, dictionary, string
Preconditions, postconditions, and raising exceptions to enforce the precondition
Defining classes (e.g., Die) including inheritance, overriding methods

Chapter 2. Algorithm Analysis
Machine dependent measures of performance: program running time and instruction count
Machine independent measures of performance: big-oh, orders of complexity: constant O(1), logarithmic O(log n), linear O(n), "n log n"/log linear O(n log n), quadratic $O(n^2)$, cubic $O(n^3)$, exponential $O(2^n)$
Complexity analysis of an algorithm to determine its big-oh notation
Implementation of Python lists as an array of object references with implications on operation big-oh (e.g., pop() is O(1) while pop(0) is O(n), etc.)

Chapter 3. Basic Data Structures
General concept of a stack: LIFO, top and bottom
Stack Operations: pop, push, peek, size, isEmpty, __str__
Stack Implementations: Python list to store stack items and linked list of Nodes to store stack items including big-oh of operations
Stack Applications: general idea of using a stack to do parentheses matching and palindrome checking

General concept of a queue: FIFO, front and rear
Queue Operations: enqueue, dequeue, peek, size, isEmpty, __str__
Queue Implementations: Python list to store queue items and linked list of Nodes to store stack items including big-oh of operations

General concept of a deque: double ended queue, front and rear
Deque Operations: addFront, addRear, removeFront, removeRear, size, isEmpty, __str__
Deque Implementations: Python list to store deque items, singly-linked list of Nodes to store deque items, and doubly-linked list of Nodes (e.g, Node2Way) including big-oh of operations
Deque Applications: general idea of using deque to do palindrome checking

General concept of a list: head, tail, index
Categories of List operations: index-based, content-based, cursor-based
Unordered List operations and implementation with a singly-linked list of Nodes including big-oh of operations
Ordered List operations and implementation with a singly-linked list of Nodes including big-oh of operations

Section 6.6. Priority Queue with Binary Heaps
General concept of a priority queue: remove highest priority next
Priority Queue Operations: enqueue, dequeue, peek, size, isEmpty, __str__
Priority Queue Implementations: Python list unordered, Python list ordered by priority, Binary Heap including big-oh of operations
Binary Heap implementation: insert, findMin, delMin, isEmpty, size, buildHeap including big-oh of operations

Question 1. (4 points) Consider the following Python code.

```
i = n
while i > 1:
    print(i)
    i = i // 2

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

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

Question 2. (4 points) Consider the following Python code.

```
for i in range(n):

    for j in range(n):

        k = 1
        while k < n:
            print(i, j, k)
            k = k + 3
```

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

Question 3. (4 points) Consider the following Python code.

```
def main(n):
    i = n
    while i > 0:
       for j in range(n):
           doSomething(n)
       i = i // 2

def doSomething(n):
    for k in range(n):
       doMore(n*n)

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

main(n)
```

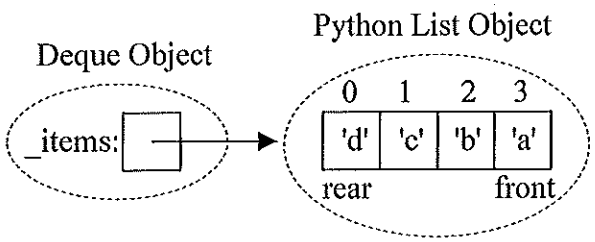
What is the big-oh notation $O(\ )$ for this code segment in terms of n?

Question 4. (5 points) Suppose a $O(n^5)$ algorithm takes 1 second when n = 1000. How long would the algorithm run when n = 10,000?

Question 5. (8 points) Why should medium/large size programs be written using function definitions instead of a single block of monolithic code written at the top-level (i.e., all statements written outside of any function)?

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 **rear** item is **always stored at index 0,**
- the front item is always at index len(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	addRear	removeRear	addFront	removeFront	size

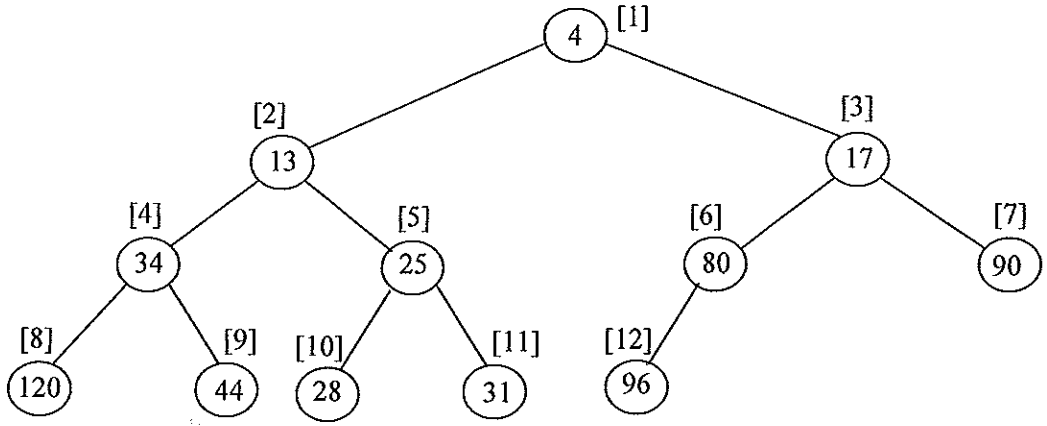
b) (9 points) Complete the code for the `addRear` method, including any precondition check needed by raising an exception if it is violated.

```
def addRear(self, newItem):
    """Adds the newItem to the rear of the Deque
       Precondition:  none
       Postcondition: newItem has been added to the rear of the Deque"""
```

c) (10 points) Complete the method for the `__str__` operation.

```
def __str__(self):
    """Returns the string representation of the Deque.
       Precondition:  none
       Postcondition: Returns a string representation of the Deque from the
       front item thru the rear item with a blank space between each item."""

    resultStr = "(front) "
```

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:



Python List actually used to store heap items

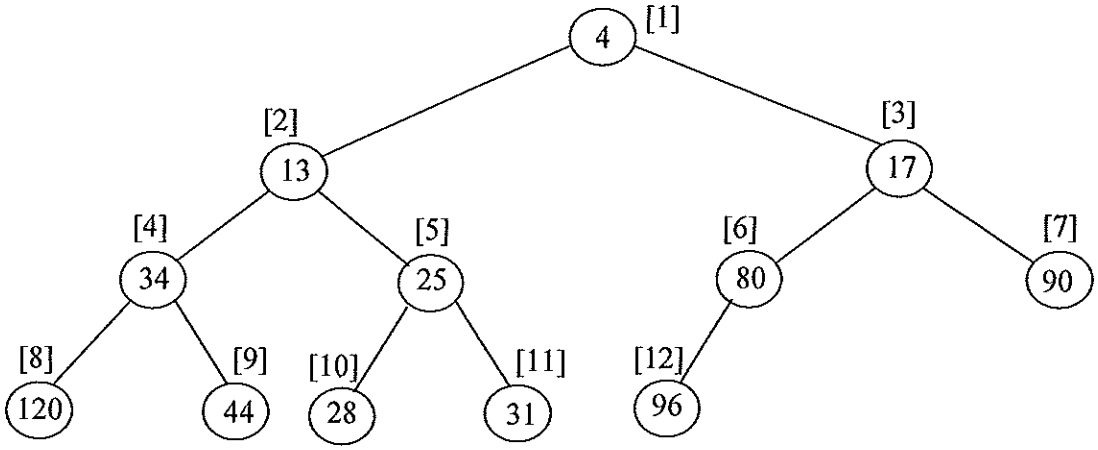
0	1	2	3	4	5	6	7	8	9	10	11	12
not used	4	13	17	34	25	80	90	120	44	28	31	96

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) (7 points) What would the above heap look like after inserting 10 **and then** 20 (show the changes on above tree)

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

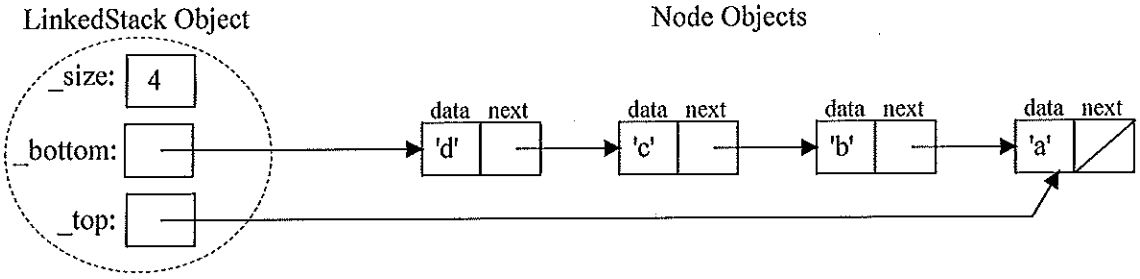


c) (2 point) What item would `delMin` remove and return from the above heap?

d) (7 points) What would the heap look like after `delMin`? (show the changes on tree in the middle of the page)

e) (6 points) What is the big-oh notation for the `delMin` operation? **(EXPLAIN YOUR ANSWER)**

Question 8. The `Node` class can be used to dynamically create storage for each new item added to a Stack using a singly-linked implementation as in:



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

isEmpty	size	pop	push(item)	__init__	__str__

b) (12 points) Complete the `push` method for the above LinkedStack implementation.

```
class LinkedStack(object):
    """ Singly-linked list based Stack implementation."""

    def __init__(self):
        self._size = 0
        self._bottom = None
        self._top = None

    def push(self, item):
        """ Adds the item to the top of the Stack.
            Precondition:  none """
```

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

c) (7 points) Suggest an improvement to the above implementation to speed up some of the stack operations enough to change their big-oh notation? (Justify your answer)