CSC6013 Final Exam - Summer 2 2025 - instructor: Paulo Fernandes

You have four hours to solve the questions and upload your .pdf with answers

1) Linked Lists - Create a Swap Method

Given the LinkedList implementation seen in class (here is a copy and also the link: linkedlist.py), create a method that swaps the node pointed by the *current* pointer by the next node (the *current* will be the formerly next node). This method shall return -1 if the swap is impossible (either the list is empty or the *current* node has no node next), or it should return 0 if the swap was performed.

For example, for a linked list:

20   40   50(current)   60   70

The method swap will change the linked list to

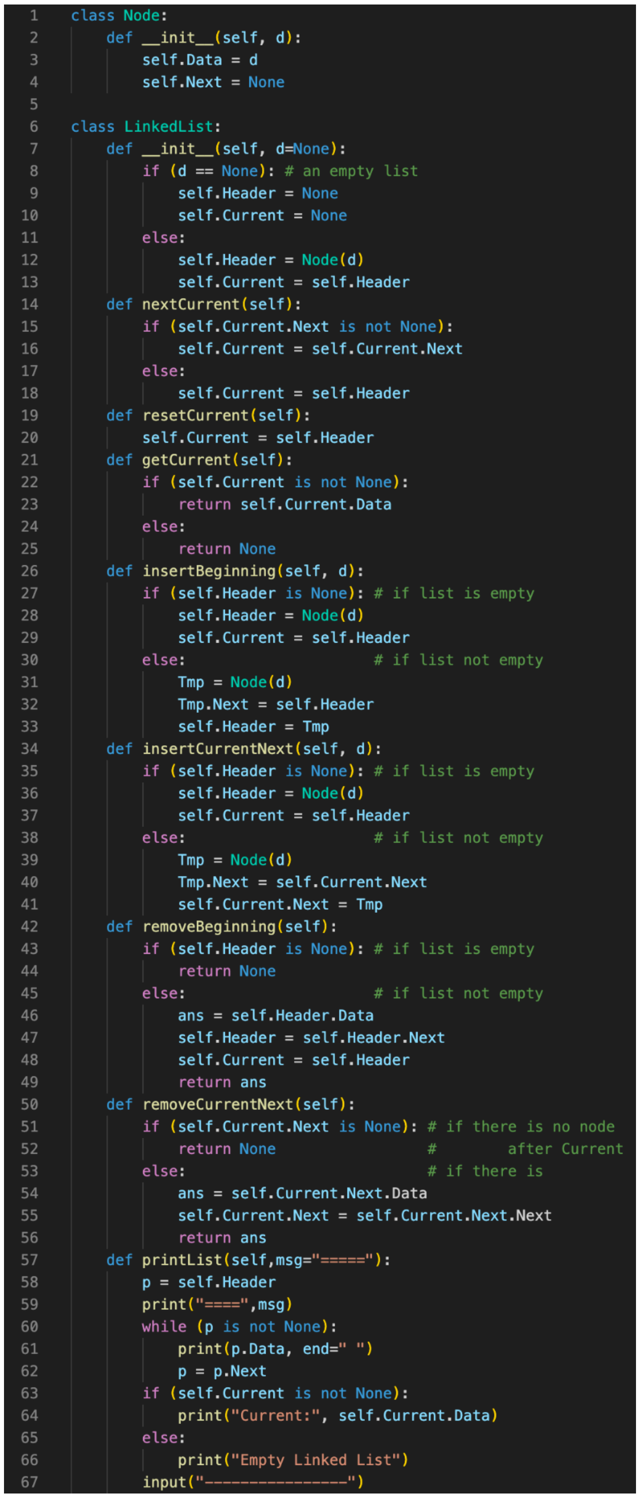
20   40   60(current)   50   70

… and it returns the value 0

Another example, for a linked list:

20   40   50   60   70(current)

The method swap will not change the linked list and it returns the value -1



```python
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
    def nextCurrent(self):
        if (self.Current.Next is not None):
            self.Current = self.Current.Next
        else:
            self.Current = self.Header
    def resetCurrent(self):
        self.Current = self.Header
    def getCurrent(self):
        if (self.Current is not None):
            return self.Current.Data
        else:
            return None
    def insertBeginning(self, d):
        if (self.Header is None): # if list is empty
            self.Header = Node(d)
            self.Current = self.Header
        else:                     # if list not empty
            Tmp = Node(d)
            Tmp.Next = self.Header
            self.Header = Tmp
    def insertCurrentNext(self, d):
        if (self.Header is None): # if list is empty
            self.Header = Node(d)
            self.Current = self.Header
        else:                     # if list not empty
            Tmp = Node(d)
            Tmp.Next = self.Current.Next
            self.Current.Next = Tmp
    def removeBeginning(self):
        if (self.Header is None): # if list is empty
            return None
        else:                     # if list not empty
            ans = self.Header.Data
            self.Header = self.Header.Next
            self.Current = self.Header
            return ans
    def removeCurrentNext(self):
        if (self.Current.Next is None): # if there is no node
            return None                 #        after Current
        else:                           # if there is
            ans = self.Current.Next.Data
            self.Current.Next = self.Current.Next.Next
            return ans
    def printList(self,msg="====="):
        p = self.Header
        print("====",msg)
        while (p is not None):
            print(p.Data, end=" ")
            p = p.Next
        if (self.Current is not None):
            print("Current:", self.Current.Data)
        else:
            print("Empty Linked List")
        input("----------------")
```

2) **Asymptotic Notations - Computing the Complexity**

Answer the following questions explaining in a short sentence your rationale to find the answer. Consider that all relevant tasks to each algorithm is

a) A given algorithm **A** is an iterative one that has two loops disposed sequentially (one after the other) each going over the ***n*** iterations. What is the complexity of **A**?
b) A given algorithm **B** is an iterative one that has two nested loops (one inside the other) each going over the ***n*** iterations. What is the complexity of **B**?
c) A given algorithm **C** is a recursive one that for a problem of size ***n*** executes ***O(n)*** recursive calls and to each recursive call it executes a certain number of tasks adding up a ***$O(n^2)$*** complexity each. What is the complexity of **C**?
d) A given algorithm **D** is an iterative one that for a problem of size ***n*** executes ***$O(n^2)$*** calls of a function that has complexity ***O(log n)***. What is the complexity of **D**?

3) **Brute-Force Algorithm - Create the Difference of Two Sets**

Given two arrays of Integers **A** and **B** with ***len(A) = n*** and ***len(B) = m***, create a third array **C** that includes all elements of **A** that are not in **B**. We write this operation as "**A – B**"; we call the operation set difference; and we call the result the difference of the two sets **A** and **B** (or simply "**A minus B**"). Assume that in each array, each element is listed only once (there are no duplicates within the same array), but the elements are not sorted.

a) Write a brute force function that uses nested for loops to repeatedly check if each element in **A** matches any of the elements in **B**. If the element from **A** does not match any element in **B**, then copy it into the next available slot of array **C**. Do not sort any of the arrays at any time.

Example: With ***A = [2, 4, 6]*** and ***B = [3, 4, 5]***, your algorithm should produce ***C = [2, 6]***.

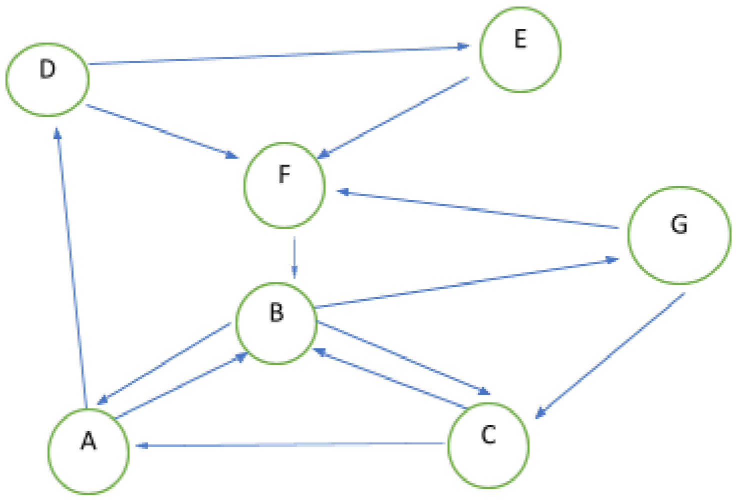
b) Trace the algorithm with

A = [20, 40, 70, 30, 10, 80, 50, 90, 60]

B = [35, 45, 55, 60, 50, 40]

c) Perform asymptotic analysis to determine the maximum number of comparisons of array elements that are needed. What is the Big-Oh class for this algorithm in terms of ***m*** and ***n***?

4) Recursion - Breadth First Search and Depth First Search



a) Represent this graph using adjacency lists. Arrange the neighbors of each vertex in alphabetical order.

b) Show the steps of a breadth first search with the graph using the technique given in the class notes. Use the adjacency lists representation that you created. Start at vertex **A**. As part of your answer, produce a graph that has the vertices numbered according to the order in which they were processed/visited.

c) Show the steps of a depth first search with the graph using the technique given in the class notes. Use the adjacency lists representation that you created. Start at vertex **A**. As part of your answer, produce a graph that has the vertices numbered according to the order in which they were processed/visited.

5) **Recursion - Master Method**

Use the master method to determine the Big-Oh class for an algorithm whose worst-case performance is given by each of these recurrence relations.

a) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

b) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

c) $T(n) = 4T\left(\frac{n}{2}\right) + n$

6) Decrease-and-Conquer Algorithm – Maximum Element in Array

a) Write a recursive decrease-and-conquer algorithm to calculate the maximum element in a non-empty array of real numbers. Your algorithm should work by comparing the last element in the array with the maximum of the "remaining front end" of the array.

For example, to find the largest element in the array [5, 13, 9, 10] your algorithm should call itself to find the maximum of [5, 13, 9] and return either 10 or the result of the recursive call, whichever is larger.

- Do not use Python's built-in max() function.
- Do not rearrange the elements of the array by sorting or partially sorting them.
- Do not use any loops.

You can assume that the array has at least one element in it.

Your function call should call should be

Maximum(A, right)

where the two input parameters are the array and right index. With these input parameters, the function should return the maximum array element from A[0] to A[right]. Return the value of the array element, not the index where it occurs in the array.

b) Trace your algorithm with

A = [17, 62, 49, 73, 26, 51]

c) Write a recurrence relation for the number of comparisons of array elements that are performed for a problem of size n. Then perform asymptotic analysis to determine the Big-Oh class for this algorithm

7) **Divide-and-Conquer Algorithms – Mergesort and Quicksort**

a) For each of these two sorting algorithms, what is its Big-Oh class in the worst case?

b) For each of these two sorting algorithms, what is its Big-Oh class in the average case?

c) Trace the mergesort algorithm for the following array of values.

A = [127, 48, 62, 51, 198, 17, 52, 209]

Rather than keep track of the values of individual variables, follow the graph-like that was used in the slides to trace the Mergesort algorithm.

d) Trace the Quicksort algorithm for the same array of values.

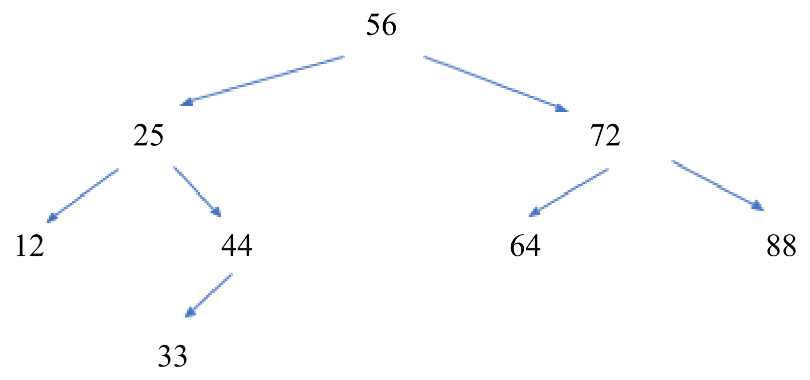
A = [127, 48, 62, 51, 198, 17, 52, 209]

Indicate the pivots in red as was done in the class notes.

8) **Transform-and-Conquer Algorithms – AVL Trees**

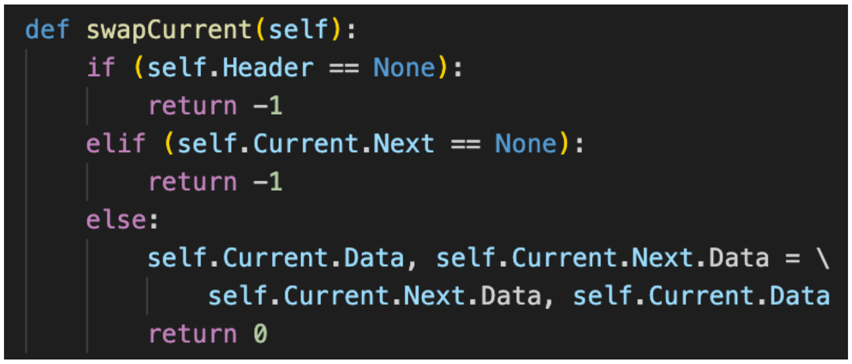
Considering the AVL Tree below, what happens if the value 38 is inserted in this tree?

a) Depict the tree immediately after the insertion of 38 (without balancing);
b) Describe what possible rotations, if necessary, need to be taken to balance the tree, indication the rotation kind and target node;
c) Depict the tree after the balancing operations (rotations).



SOLUTIONS

solution to #1) Linked Lists - Create a Swap Method
One possible solution is:



```
def swapCurrent(self):
    if (self.Header == None):
        return -1
    elif (self.Current.Next == None):
        return -1
    else:
        self.Current.Data, self.Current.Next.Data = \
            self.Current.Next.Data, self.Current.Data
        return 0
```

solution to #2) Asymptotic Notations - Computing the Complexity
Answer the following questions explaining in a short sentence your rationale to find the answer. Consider that all relevant tasks to each algorithm is

a) A given algorithm **A** is an iterative one that has two loops disposed sequentially (one after the other) each going over the ***n*** iterations. What is the complexity of **A**?
   i) ***O(n)***
b) A given algorithm **B** is an iterative one that has two nested loops (one inside the other) each going over the ***n*** iterations. What is the complexity of **B**?
   i) ***$O(n^2)$***
c) A given algorithm **C** is a recursive one that for a problem of size ***n*** executes ***O(n)*** recursive calls and to each recursive call it executes a certain number of tasks adding up a ***$O(n^2)$*** complexity each. What is the complexity of **C**?
   i) ***$O(n^3)$***
d) A given algorithm **D** is an iterative one that for a problem of size ***n*** executes ***$O(n^2)$*** calls of a function that has complexity ***O(log n)***. What is the complexity of **D**?
   i) ***$O(n^2 \log n)$***

solution to #3) Brute-Force Algorithm - Create the Difference of Two Sets

```
# Input: Arrays A and B of integers.
# Output: Array C containing the elements of A that are not in B.
def DifferenceOfArrays(A, B, C):
    i = -1
    for j in range(len(A)):
        match = False
        for k in range(len(B)):
            if A[j] == B[k]:
                match = True
        if match == False:
            i = i+1
            C[i] = A[j]
    return C
```

b) Trace the algorithm with
A = <20, 40, 70, 30, 10, 80, 50, 90, 60>
B = <35, 40, 45, 50, 55, 60>

20-35, 20-40, 20-45, 20-50, 20-55, 20-60 thus 20 is not in B, so C[0] = 20
40-35, 40=40, thus 20 is in B, so do not add it to C
70-35, 70-40, 70-45, 70-50, 70-55, 70-60 thus 70 is not in B, so C[1] = 70
30-35, 30-40, 30-45, 30-50, 30-55, 30-60 thus 30 is not in B, so C[2] = 30
10-35, 10-40, 10-45, 10-50, 10-55, 10-60 thus 10 is not in B, so C[3] = 10
80-35, 80-40, 80-45, 80-50, 80-55, 80-60 thus 80 is not in B, so C[4] = 80
50-35, 50-40, 50-45, 50=50 thus 50 is in B, so do not add it to C
90-35, 90-40, 90-45, 90-50, 90-55, 90-60 thus 90 is not in B, so C[5] = 90
60-35, 60-40, 60-45, 60-50, 60-55, 60=60 thus 60 is in B, so do not add it to C

c) Max work occurs when A and B have no common elements, so C = A.

In this case, every element of A must be compared with every element of B before adding it to C. This requires n * m comparisons, so O(mn).

solutions to #4) DFS and BFS

a) adjacency lists
A -> B >- D
B -> A -> C -> G
C -> A -> B
D -> E -> F
E -> F
F-> B
G -> C -> F

b) BFS: A1, B2, D3, C4, G5, E6, F7

c) DFS: A1, B2, C3, G4, F5, D6, E7

solutions to #5) Master Method

a) a=4, b=2, d=3   4 < 2^3   so O(n^3)

b) a=4, b=2, d=2   4 = 2^2   so O(n^2 * lg n)

c) a=4, b=2, d=1   4 > 2^1   so O(n^lg 4) = O(n^2)

solution to #6 – Decrease-and-Conquer Algorithm – Maximum Element in Array

a)

```
def Maximum(A, right)
    if right == 0:
        return A[0]
    else:
        tempmax = Maximum(A, right–1)
        if A[right] > tempmax;
            return A[right]
        else:
            return tempmax
```

```
b) compare 51 with Maximum(<17, 62, 49, 73, 26>)
    which requires a comparison of 26 with Maximum(<17, 62, 49, 73>)
        which requires a comparison of 73 with Maximum(<17, 62, 49>)
            which requires a comparison of 49 with Maximum(<17, 62>)
                which requires a comparison of 62 with Maximum(<17>)
                    which is 17
                so unstacking the function calls 62 > 17 so tempmax = 62
            and 49 < 62 so keep tempmax = 62
        and 73 > 62 so tempmax = 73
    and 26 < 73 so tempmax = 73
and 51 < 73 so max = 73
```

c) $T(n) = T(n-1) + 1, T(1) = 0$

$= [T(n-2) + 1] + 1 = T(n-2) + 2$

$= [T(n-3) + 1] + 2 = T(n-3) + 3$

in general

$= T(n-k) + k$

reach base case when $n-k = 1, n-1 = k$

$= T(1) + n-1$

$= 0 + n-1$

$= n-1$

$= O(n)$

solutions to #7) Mergesort and Quicksort

a) In the worst case, Mergesort is O(nlg(n)) and quicksort is O(n^2)
b) In the average case, Mergesort is O(nlg(n)) and quicksort is O(nlg(n))
c) Mergesort

$$\begin{tikzcd}
 & & & \langle 127, 48, 62, 51, 198, 17, 52, 209\rangle \arrow[dll] \arrow[drr] & & & & \\
 & \langle 127, 48, 62, 51\rangle \arrow[dl] \arrow[dr] & & & & \langle 198, 17, 52, 209\rangle \arrow[dl] \arrow[dr] & & \\
\langle 127, 4\rangle \arrow[d] \arrow[dr] & & \langle 62, 51\rangle \arrow[d] \arrow[dr] & & \langle 198, 17\rangle \arrow[d] \arrow[dr] & & \langle 52, 209\rangle \arrow[d] \arrow[dr] & \\
\langle 127\rangle \arrow[d] & \langle 4\rangle \arrow[dl] & \langle 62\rangle \arrow[d] & \langle 51\rangle \arrow[dl] & \langle 198\rangle \arrow[d] & \langle 17\rangle \arrow[dl] & \langle 52\rangle \arrow[d] & \langle 209\rangle \arrow[dl] \\
\langle 4, 127\rangle \arrow[dr] & & \langle 51, 62\rangle \arrow[dl] & & \langle 17, 198\rangle \arrow[dr] & & \langle 52, 209\rangle \arrow[dl] & \\
 & \langle 4, 51, 62, 127\rangle \arrow[drr] & & & & \langle 17, 52, 198, 209\rangle \arrow[dll] & & \\
 & & & \langle 4, 17, 51, 52, 62, 127, 198, 209\rangle & & & &
\end{tikzcd}$$

d) Quicksort

<127, 48, 62, 51, 198, 17, 52, 209>
209 becomes the pivot

[127, 48, 62, 51, 198, 17, 52] 209
52 becomes the pivot

[48, 51,17] 52 [198, 62, 127] 209
17 and 127 become pivots

17 [51, 48] 52 [62] 127 [198] 209
48 becomes a pivot

17 48 [51] 52 [62] 127 [198] 209
everyone is sorted

solutions to #8) Transform-and-Conquer Algorithms – AVL Trees

$$\begin{tikzcd}
 & & & & 56 \arrow[dlll] \arrow[drr] & & & \\
 & 25 \arrow[dl] \arrow[drr] & & & & & 72 \arrow[dl] \arrow[dr] & \\
12 & & & 44 \arrow[dl] & & 64 & & 88 \\
 & & 33 & & & & &
\end{tikzcd}$$

a) After inserting 38:

$$\begin{tikzcd}
 & & & & & 56 \arrow[dllll] \arrow[drr] & & & \\
 & 25 \arrow[dl] \arrow[drrr] & & & & & & 72 \arrow[dl] \arrow[dr] & \\
12 & & & & 44 \arrow[dll] & & 64 & & 88 \\
 & & 33 \arrow[dr] & & & & & & \\
 & & & 38 & & & & &
\end{tikzcd}$$

b) Operations needed to balance:
   i) rotate right 44
   ii) rotate left 25

c) After balancing:

$$\begin{tikzcd}
 & & & & & 56 \arrow[dlll] \arrow[drr] & & & \\
 & & 33 \arrow[dl] \arrow[drr] & & & & & 72 \arrow[dl] \arrow[dr] & \\
 & 25 \arrow[dl] & & & 44 \arrow[dl] & & 64 & & 88 \\
12 & & & 38 & & & & &
\end{tikzcd}$$