תרגיל 8-ב: המשך של Abstract data type ועיבוד תמונה

תאריך פרסום: 5.1.2023

תאריך הגשה: 15.1.2023 בשעה 23:59

מתרגלת אחראית: אסראא נסאסרה

משקל תרגיל: 3 נקודות

הנחיות לתרגיל:

אין לייבא ספריות, חיצוניות או מובנות (למעט itertools).

שאלה 1:

בחלק הראשון של התרגיל מימשתם מבני נתונים המייצגים דנ״א ורנ״א. כעת תשתמשו במבני הנתונים שמימשתם כדי לייצג ולנתח דנ״א. מימוש של מבני הנתונים שיש להשתמש בהם נתון לרשותכם בקבצים Stack.py ו-BinarySearchTree.py.

סעיף א:

אחד מהניתוחים החישוביים של רצפי דנ״א (התחום נקרא ״ביואינפורמטיקה״) הוא מיונם. ממשו את הפונקציה:

```
def sort_DNA_Stack(DNA_stack):
```

הפונקציה מקבלת מבנה נתונים מטיפוס מחסנית, המכילה אובייקטי דנ״א DNA, ומבצעת להם מיון על סמך האורך והמסה שלהם. הקריטריון הראשון למיון יהיה אורך הדנ״א, כאשר אורכי הדנ״א זהים, הסדר נקבע על פי המסה.
הפונקציה תחזיר מחסנית חדשה ממוינת, כך שהדנ״א הגדול ביותר (לפי הקריטריונים שהוגדרו) יהיה בראש המחסנית.
לטובת מימוש המיון יש לרשותכם מחסנית עזר אחת בלבד. אין להשתמש במבני עזר נוספים\אחרים!

דוגמת הרצה:

```
stack_1 = Stack()
stack_1.push(DNA("TGGGTA"))
stack_1.push(DNA("ATGGAA"))
stack_1.push(DNA("AGTCAAAGGGCTTTATATAT"))
stack_1.push(DNA("AGGGGGGCCCCCC"))
stack_1.push(DNA("AGGGGGCCC"))
print(f"The original DNA stack: ")
print(stack_1)
print("Sorted DNA are: ")
sortedst = sort_DNA_Stack(stack_1)
print(sortedst)
```

תתקבל התוצאה הבאה:

```
The original DNA stack:
|TGGGTA ATGGAA AGTCAAAGGGCTTTATATAT AGGGGGGCCCCCC AGGGGGCCC <-top
Sorted DNA are:
|ATGGAA TGGGTA AGGGGGCCC AGGGGGGCCCCCC AGTCAAAGGGCTTTATATAT <-top
```

סעיף ב:

להזכירכם, כפי שלמדתם בחלק א׳ של התרגיל, הדנ״א מורכב משני גדילים, כאשר האחד משלים את השני. לבניית הגדיל המשלים יש לזווג לכל ״A״ את הנוקלאוטיד ״T״, לכל ״T״ את ה-״A״, לכל ״C״ את ה-״G״ ולכל ״G״ את ה-״C״. הגדיל המשלים נקרא באופן הופכי (מהסוף להתחלה).

ניתן להשתמש בעץ חיפוש בינארי כדי להציג ולנתח גדילי דנ״א. השימוש בעץ חיפוש בינארי רלוונטי כאשר לא כל הנוקלאוטידים ידועים מראש ויש צורך לעדכנם באופן יעיל.

ממשו את הפונקציה:

```
def complement_BST_DNA(sequence_BST):
```

המקבלת עץ חיפוש בינארי המייצג גדיל דנ״א. כל קודקוד (node) בעץ מורכב ממפתח מטיפוס מספר שלם חיובי המייצג את מיקום הנוקלאוטיד בגדיל, וערך מטיפוס מחרוזת המייצגת את הנוקלאוטיד הבודד בקודקוד. הפונקציה מחזירה מחרוזת, המייצגת את הגדיל המשלים בסדר ההפוך.

דוגמת הרצה:

```
BST = BinarySearchTree()
BST.insert(4, "T")
BST.insert(2, "A")
BST.insert(6, "C")
BST.insert(1, "G")
BST.insert(3, "A")
BST.insert(7, "G")
BST.insert(5, "T")
print(BST)
print(complenet_BST_DNA(BST))
```

תודפס התוצאה הבאה:

```
                    4:T
           ________/   \________
        2:A                     6:C
     __/   \__               __/   \__
  1:G         3:A         5:T         7:G
 /   \       /   \       /   \       /   \
#     #     #     #     #     #     #     #

CGAATTC
```

שאלות 2-4 הן שאלות בלתי תלויות ואינן קשורות לחלק א של עבודת בית 8.

שאלה 2:

ממשו את הפונקציה :

```
def print_right_view(binary_tree):
```

הפונקציה מקבלת עץ בינארי שכל קודקוד בו מכיל מפתח וערך. הפונקציה מחזירה רשימה המכילה את ערכי הקודקודים הימניים ביותר בכל קבוצת קודקודים במרחק מסוים מהשורש (כלומר בכל רמה בעץ). למשל, בדוגמת ההרצה 1, הקודקוד 6 הוא הימני ביותר במרחק 0 מהשורש, הקודקוד 2 הוא הימני ביותר במרחק 1 מהשורש, והקודקוד 4 הוא הימני ביותר במרחק 2 מהשורש.

(המימוש של עץ בינארי יינתן לכם בקובץ BinaryTree.py)

דוגמאות הרצה:

דוגמה 1:

```
bin_tree = BinaryTree()
n1 = TreeNode(1, 'a')
n2 = TreeNode(2, 'b')
n3 = TreeNode(3, 'c')
n4 = TreeNode(4, 'd')
n5 = TreeNode(5, 'e')
n6 = TreeNode(6, 'f')
bin_tree.root = n6
n6.left = n3
n6.right = n2
n2.right = n4
n3.left = n1
n3.right = n5
print(bin_tree)
print(print_right_view(bin_tree))
```

יודפסו התוצאות הבאות למסך:

```
                    6:f
          ________/   \__
       3:c              2:b
     __/   \__          /   \__
  1:a         5:e      #       4:d
 /   \       /   \            /   \
#     #     #     #          #     #

['f', 'b', 'd']
```

דוגמה 2 :

```
bin_tree = BinaryTree()
n1 = TreeNode(1, 'a')
n2 = TreeNode(2, 'b')
n3 = TreeNode(3, 'c')
n4 = TreeNode(4, 'd')
n5 = TreeNode(5, 'e')
n6 = TreeNode(6, 'f')
n7 = TreeNode(12, 'x')
n8 = TreeNode(23, 'y')
bin_tree.root = n6
n6.left = n3
n6.right = n2
n2.right = n4
n3.left = n1
n3.right = n5
n5.left = n7
n1.right = n8
print(bin_tree)
print(print_right_view(bin_tree))
```

יודפסו התוצאות הבאות למסך :

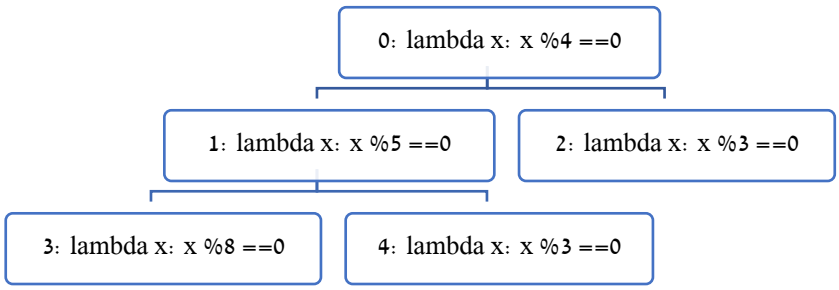
```
                                6:f
                ______________/   \__
            3:c                     2:b
    _________/   \_________           /   \__
  1:a                       5:e       #       4:d
 /   \__                 __/   \             /   \
#       23:y         12:x       #           #     #
       /    \       /    \
      #      #     #      #

['f', 'b', 'd', 'x']
```

שאלה 3:

עץ החלטה בינארי הוא עץ שכל קודקוד בו מכיל כערך פונקציית lambda, וכל קודקוד פנימי מכיל רק פונקציית lambda בינארית המחזירה True/False. בהינתן קלט מסוים, ההחלטה לגבי איזה כיוון להמשיך בעץ ניתנת על סמך הערך המוחזר מהפונקציה בקודקוד הנוכחי על הקלט. אם הפונקציה בקודקוד מחזירה True יש להמשיך לבן השמאלי ואחרת לבן הימני.

דוגמה:



למשל, המסלול במורד העץ עבור הערך 20 הוא: קודקוד 0 (True) -> קודקוד 1 (True) -> קודקוד 3 (False).
המסלול במורד העץ עבור הערך 9 הוא: קודקוד 0 (False) -> קודקוד 2 (True) .

סעיף א:

ממשו את המחלקה BinaryDecisionTreeNode שמכילה את ארבעת השדות הבאים:

1) key: מטיפוס int, מחזיק את המפתח של החוליה
2) val: מטיפוס function
3) right המצביע לבן הימני
4) left המצביע לבן השמאלי

ממשו את הבנאי של המחלקה המאתחל את השדות:

```
def __init__(self, key, val):
```

עליכם לדרוס את האופרטור __repr__ כך שהוא יחזיר מחרוזת המתארת את החוליה לפי התבנית הבאה:
"key: val"

```
def __repr__(self):
```

ממשו את השיטה:

```
def is_leaf(self):
```

השיטה מחזירה True אם החוליה היא עלה, ו-False אחרת.

סעיף ב:

ממשו את המחלקה BinaryDecisionTree שתייצג עץ ההחלטה בינארי:

המחלקה מכילה שדה אחד בשם decision_tree_root והוא מצביע על שורש העץ.

ממשו את בנאי המחלקה:

```
def __init__(self, elements):
```

בנאי המחלקה מקבל רשימה של פונקציות הלמבדה כקלט ומאתחל לפיה את עץ ההחלטה הבינארי לפי הכללים הבאים:

עבור כל אינדקס i ברשימה:

- Parent(i) = $\lfloor \frac{i-1}{2} \rfloor$
- Left(i) = 2i+1
- Right(i) = 2i+2

לכל קודקוד בעץ (מטיפוס BinaryDecisionTreeNode) המפתח יהיה האינדקס ה-i ברשימת הקלט (int) והערך הוא פונקציית הלמדה במקום ה-i ברשימת הקלט.

סעיף ג:

ממשו את השיטה:

```
def decide(self, input_value)
```

המקבלת משתנה מטיפוס כלשהו, שפונקציות הלמבדה בכל קודקודי העץ יודעות לקבל.

המשתנה מוזן לקודקוד השורש של העץ, ומתקדם לקודקוד הבא במסלול לפי תוצאת הפעלת פונקציית הלמדה בקודקוד הנוכחי עליו (עד שהוא מגיע לקודקוד עלה). על השיטה להחזיר כמחרוזת את מסלולו של הערך במורד העץ, שמיוצג ע"י שרשור מפתחות הקודקודים בהם הקלט עבר.

סעיף ד:

בקודקודי העלים של העץ, יכולות להיות פונקציות lambda שאינן בינאריות. ממשו את השיטה:

```
def output(self, input_value):
```

השיטה מקבלת קלט ובודקת אם קודקוד העלה במורד המסלול שלו מכיל פונקציה lambda שאינה בינארית. אם הפונקציה בקודקוד העלה אינה בינארית, השיטה תחזיר את התוצאה של ההפעלה של אותה הפונקציה בעלה על הערך שניתן כקלט. אחרת השיטה תחזיר None.

סעיף ה:

לעץ החלטה בינארי יש תצוגה שימושית אחרת, כרשימה מקוננת. האינדקסים ברשימה החיצונית בתצוגה זו מייצגים את מרחקי הקודקודים האפשריים משורש העץ (רמות העץ). הרשימה הפנימית באינדקס ה-j מכילה את כל פונקציות הלמבדה שנמצאות במרחק j משורש העץ. ממשו את השיטה:

```
def compact(self):
```

השיטה compact מחזירה את התצוגה הזו של עץ ההחלטה כרשימה מקוננת.

דוגמאות הרצה:

דוגמה 1:

```
lambda_elements_1 = [lambda x: x % 2 == 0, lambda x: x % 3 == 0, lambda x: x % 4 == 0, lambda x: x % 6 == 0, lambda x: x // 20 == 9, lambda x: x % 8 == 0, lambda x: x % 3 ]
D_tree_1 = BinaryDecisionTree(lambda_elements_1)
print(D_tree_1)
print(D_tree_1.compact()[2][1](20))
print(D_tree_1.output(60))
print(D_tree_1.output(9))
print(D_tree_1.decide(12))
```

יודפסו התוצאות הבאות:

```
            0
      ______/ \______
     1               2
   __/ \__         __/ \__
  3       4       5       6
 / \     / \     / \     / \
#   #   #   #   #   #   #   #

False
None
0
0->1->3
```

דוגמה 2:

```
lambda_elements_2 = [lambda x: len(x) >= 4, lambda x: x.upper() == x, lambda x: x.capitalize() == x, lambda x: x.startswith("A"), lambda x: "I am " + x,  lambda x: x.endswith("t")]
D_tree_2 = BinaryDecisionTree(lambda_elements_2)
print(D_tree_2)
print(D_tree_2.decide("Esraa"))
print(D_tree_2.decide("Tom"))
print(D_tree_2.output("Assaf"))
print(D_tree_2.output("Tom"))
print(D_tree_2.compact()[2][0]("Noy"))
```

יודפסו התוצאות הבאות:

```
            0
      ______/ \______
     1               2
   __/ \__         __/ \
  3       4       5     #
 / \     / \     / \
#   #   #   #   #   #

0->1->4
0->2->5
I am Assaf
None
False
```

שאלה 4:

זיהוי גבולות הוא משימה בסיסית בהרבה מודלים של למידה חישובית. גילוי הגבולות בתמונה דורש קונבולוציה של המטריצות הבאות על התמונה:

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

לשם גילוי הגבולות בתמונה יש לייצר שתי תמונות, וזאת ע"י:

1) קונבולוציה של המטריצה הימנית על תמונת הקלט בציר ה-x. כיוון הקונבולוציה הוא ימינה בציר ה-x. ונסמן את התוצאה כ-Gx.
2) קונבולוציה של המטריצה השמאלית על תמונת הקלט בציר ה-y. כיוון הקונבולוציה הוא למטה בציר ה-y. ונסמן את התוצאה כ-Gy.

את שתי התמונות יש לאחד ע"י מעבר בכל אינדקס בשתיהן במקביל וביצוע החישוב הבא:

$$G = \sqrt{Gx^2 + Gy^2}$$

```
def edge_detection(image_nested_list, x_matrix, y_matrix):
```

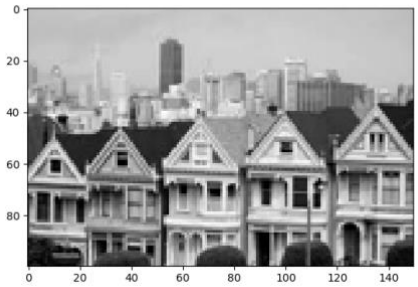
ממשו את הפונקציה המקבלת תמונה המיוצגת ע"י רשימה דו ממדית ומפעילה עליה את אלגוריתם גילוי הגבולות. הפונקציה מחזירה רשימה דו ממדית של ערכים מספריים שלמים, המייצגת את התמונה לאחר הפעלת האלגוריתם.

<u>דוגמת הרצה:</u>

לרשותכם, תמונת דוגמה השמורה כקובץ SanFrancisco.txt. התמונה היא בגוון אפור (grayscale עם ערכים מספריים שלמים 0-255). פונקציית העזר read_image מיועדת לקבל נתיב לקובץ ומחזירה רשימה דו ממדית המייצגת את התמונה. בנוסף לכך, לרשותכם פונקציית עזר שתאפשר לכם להציג את התמונה (בשם display).

```
SanFrancisco_image = read_image("SanFrancisco.txt")
display(SanFrancisco_image)
x_structure = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
y_structure = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]
SanFrancisco_post_ED_image = edge_detection(SanFrancisco_image, x_structure, y_structure)
display(SanFrancisco_post_ED_image)
```

התמונה טרם ביצוע גילוי הגבולות:



התמונה אחרי גילוי הגבולות: