תרגיל בית מספר 5 - להגשה עד 12 בינואר בשעה 23:55

שימו לב שסעיף ג' בשאלה 1 מבוטל, וכן שתוקנה שגיאת הקלדה בדוגמת ההרצה של שאלה 5 סעיף א

קיראו בעיון את הנחיות העבודה וההגשה המופיעות באתר הקורס, תחת התיקייה assignments. חריגה מההנחיות תגרור ירידת ציון / פסילת התרגיל.

הגשה:

- תשובותיכם יוגשו בקובץ pdf ובקובץ py בהתאם להנחיות בכל שאלה.
- השתמשו בקובץ השלד skeleton5.py כבסיס לקובץ ה py אותו אתם מגישים.
  לא לשכוח לשנות את שם הקובץ למספר ת״ז שלכם לפני ההגשה, עם סיומת py.
- בסה״כ מגישים שני קבצים בלבד. עבור סטודנטית שמספר ת״ז שלה הוא 012345678 הקבצים שיש להגיש הם hw5_012345678.pdf ו- hw5_012345678.py.
- הקפידו לענות על כל מה שנשאלתם.
- תשובות מילוליות והסברים צריכים להיות תמציתיים, קולעים וברורים.
  להנחיה זו מטרה כפולה:
  1. על מנת שנוכל לבדוק את התרגילים שלכם בזמן סביר.
  2. כדי להרגיל אתכם להבעת טיעונים באופן מתומצת ויעיל, ללא פרטים חסרים מצד אחד אך ללא עודף בלתי הכרחי מצד שני. זוהי פרקטיקה חשובה במדעי המחשב.

שאלה 1

השאלה עוסקת במימוש בפייתון למטריצה. למי שלא למדו אלגברה ליניארית עדיין, הנה הסבר קצר:

במתמטיקה, מטריצה היא מערך דו מימדי של ערכים. נוח לחשוב על מטריצה כמורכבת משורות ועמודות, ועבור מטריצה M לסמן את האיבר בשורה ה- i ובעמודה ה- j ב- $M_{i,j}$. למשל במטריצה הבאה $M_{0,1} = 13$. כמו כן נהוג לסמן את מספר השורות ב- n ואת מספר העמודות ב- m.

9	13	5	2
1	11	7	6
3	7	4	1
6	0	7	10

ניתן לממש מטריצה בפייתון כרשימה של רשימות (כל רשימה פנימית היא שורה במטריצה). למשל המטריצה הנ"ל תיוצג כ:

M = [[9,13,5,2], [1,11,7,6], [3,7,4,1], [6,0,7,10]]

בשאלה זו נניח כי איברי המטריצה הם מספרים מטיפוס float/int.

בשאלה זו נרצה לממש מחלקה חדשה שנקראת `SparseMatrix`, שמייצגת **מטריצה דלילה**. מטריצה דלילה היא מטריצה של מספרים שבה רוב האיברים הם 0. הרעיון המנחה הוא שאם רוב האיברים הינם 0 אין צורך לשמור $\boldsymbol{n \cdot m}$ ערכים שונים (שרובם ממילא 0), ומספיק לשמור רק את האיברים שערכם שונה מ 0 (כאשר לכל איבר כזה נשמור את מספר השורה והעמודה שלו).

כל אובייקט מטיפוס `SparseMatrix` יחזיק שדה בשם rows מטיפוס מילון (dict של פייתון). המפתחות בשדה rows יהיו מספרי שורות שמכילות איברים שונים מ- 0. הערכים שהמפתחות ממופים אליהם יהיו בעצמם מטיפוס מילון: מפתח המייצג שורה i שיש בה איברים ששונים מ- 0 ימופה למילון d_i כאשר איברי d_i הם זוגות $j:t$ עבורם $M_{i,j} = t \neq 0$. דוגמה:

9	0	5	0
0	0	0	0
0	7	4	0
0	0	0	0

מטריצה זו ניתנת לייצוג ע"י השדות rows באופן הבא:

self.rows = {0:{0:9, 2:5}, 2:{1:7, 2:4}}

להלן מתודת האיתחול של המחלקה `SparseMatrix`:

```
class SparseMatrix():
    def __init__(self, n, m):
       self.rows = {} #empty dict
       self.size = (n,m)
```

בקובץ השלד ממומשת עבורכם גם __repr__, שמדפיסה את המטריצה באופן הטבעי (שורה שורה על פי מספר השורה).

א. השלימו את מימוש המתודות הבאות (ראו דוגמאות הרצה בהמשך). הניחו כי חיפוש , הכנסה או מחיקה במילון של פייתון רצות ב- O(1) זמן בממוצע.

- dim - מחזירה tuple שהאיבר הראשון בו מייצג את מספר השורות והאיבר השני את מספר העמודות
- __eq__ - לתמיכה באופרטור == . כמו בדוגמה: mat==mat2.
  על המתודה לרוץ בסיבוכיות זמן ממוצעת $O(k_1 + k_2)$, כאשר k_1 הינו מספר הערכים השונים מ-0 במטריצה הראשונה (self), ו-k_2 הינו מספר הערכים השונים מ-0 במטריצה השניה (other).
- __getitem__ שתתמוך בפקודה מהסוג:

```
z = mat[i,j]
```

  שימו לב שהמתודה הזו מקבלת שני קלטים: הראשון self , והשני הינו tuple שמייצג את i,j. המתודה מחזירה את הערך שנמצא בשורה ה i ובעמודה ה j של המטריצה mat. אם הועברו למתודה אינדקסים לא חוקיים (למשל אינדקס שלא קיים במטריצה) על המתודה להחזיר את המחרוזת:
  "index out of bounds". על המתודה לרוץ בסיבוכיות זמן ממוצעת O(1).
- __setitem__ שתתמוך בפקודה מהסוג:

```
mat[i,j] = z
```

  שימו לב שהמתודה הזו מקבלת שלושה קלטים: הראשון self , השני הינו tuple שמייצג את i,j, והשלישי val. המתודה מעדכנת את הערך שנמצא בשורה ה i ובעמודה ה j במטריצה להיות val. אם הועברו למתודה אינדקסים לא חוקיים (למשל אינדקס שלא קיים במטריצה) על המתודה להחזיר את המחרוזת:
  "index out of bounds". על המתודה לרוץ בסיבוכיות זמן ממוצעת O(1).
- __add__ לתמיכה באופרטור + (חיבור מטריצות דלילות שיחזיר מטריצה דלילה חדשה). המתודה פועלת על מטריצות בעלות אותם המימדים (עליכם לוודא זאת באמצעות הפקודה assert).
  על המתודה לרוץ בסיבוכיות זמן ממוצעת $O(k_1 + k_2)$, כאשר k_1 הינו מספר הערכים השונים מ-0 במטריצה הראשונה (self), ו-k_2 הינו מספר הערכים השונים מ-0 במטריצה השניה (other).
- __neg__ לתמיכה באופרטור מינוס אונארי, שמחזיר מטריצה חדשה שאיבריה הם t – לכל איבר t במטריצה self. על המתודה לרוץ בסיבוכיות זמן ממוצעת $O(k_1)$, כאשר k_1 הינו מספר הערכים השונים מ-0 במטריצה (self).
- __sub__ לתמיכה באופרטור - בינארי (חיסור מטריצות דלילות שיחזיר מטריצה דלילה חדשה). המתודה פועלת על מטריצות בעלות אותם המימדים (עליכם לוודא זאת באמצעות הפקודה assert).
  על המתודה לרוץ בסיבוכיות זמן ממוצעת $O(k_1 + k_2)$, כאשר k_1 הינו מספר הערכים השונים מ-0 במטריצה הראשונה (self), ו-k_2 הינו מספר הערכים השונים מ-0 במטריצה השניה (other). ממשו מתודה זו באמצעות שתי הקודמות.

בכל המתודות מומלץ וכדאי להשתמש באופרטור [], שקורא ל __setitem__, __getitem__ לאחר מימוש מתודות אלו.

דוגמאות הרצה (המשך בעמוד הבא):

```
>>> m = SparseMatrix(3,2)
>>> m
[0, 0]
[0, 0]
[0, 0]
>>> m1 = SparseMatrix(3,2)
>>> m1
[0, 0]
[0, 0]
[0, 0]
>>> m == m1
True
```

```
>>> m[1,0] = 30
>>> m[2,1] = -2
>>> m
[0, 0]
[30, 0]
[0, -2]

>>> x = m[1,1]
>>> x
0
>>> m.dim()
(3, 2)
>>> m2 = SparseMatrix(3,2)
>>> m2[0,0] = 100
>>> m2
[100, 0]
[0, 0]
[0, 0]

>>> m3 = m + m2
>>> m3
[100, 0]
[30, 0]
[0, -2]

>>> m
[0, 0]
[30, 0]
[0, -2]

>>> m4 = m- m2
>>> m4
[-100, 0]
[30, 0]
[0, -2]

>>> m
[0, 0]
[30, 0]
[0, -2]

>>> m4 = m-m
>>> m4
[0, 0]
[0, 0]
[0, 0]

>>> m5 = -m
>>> m5
[0, 0]
[-30, 0]
[0, 2]
```

ב. המתודה __mul__ לתמיכה באופרטור * מומשה עבורכם. מתודה זו תומכת בכפל מטריצות (כלומר גם other הוא אובייקט מטיפוס מטריצה דלילה). שימו לב שמספר השורות של other שווה למספר העמודות של self, אחרת כפל המטריצות אינו מוגדר. הפלט יהיה מטריצה דלילה חדשה.
קישור להסבר על מכפלת מטריצות.
המימוש של __mul__ מופיע בעמוד הבא, ולאחריו דוגמאות הרצה.
נניח שהמטריצה הראשונה (self) בעלת n שורות ו-m עמודות, והשניה (other) בעלת m שורות ו r עמודות.
נסמן ב k_1 את מספר הערכים השונים מ-0 במטריצה הראשונה (self), וב-k_2 את מספר הערכים השונים מ-0 במטריצה השניה (other).
בקובץ ה pdf נתחו את סיבוכיות הזמן של __mul__ כתלות ב n, m, r, k_1, k_2 (או בחלק מהם). תנו תשובה במונחי O(.) ונמקו.

```
def __mul__(self,other):
    assert isinstance(other, SparseMatrix)
    assert self.dim()[1] == other.dim()[0]

    n,m = self.dim()[0], other.dim()[1]
    new = SparseMatrix(n,m)

    #create columns dict for other
    other_cols = {}
    for i in other.rows:
        for j in other.rows[i]:
            if j not in other_cols:
                other_cols[j] = {}
            other_cols[j][i] = other.rows[i][j]

    for i in self.rows:
        for j in other_cols:
            s_ij = 0
            rowd = self.rows[i]
            for k in self.rows[i]:
                if k in other_cols[j]:
                    s_ij += self.rows[i][k] * other_cols[j][k]
            if s_ij != 0:
                if i not in new.rows:
                    new.rows[i] = {}
                new.rows[i][j] = s_ij

    return new
```

דוגמת הרצה (המשך בעמוד הבא):

```
>>> m = SparseMatrix(3,2)
>>> m[1,0] = 30
>>> m[2,1] = -2
>>> m
[0, 0]
[30, 0]
[0, -2]
>>> m1 = SparseMatrix(2,3)
>>> m1[0,0] = 100
>>> m2 = m*m1
>>> m2
[0, 0, 0]
```

```
[3000, 0, 0]
[0, 0, 0]
```

~~ג. בקובץ ה pdf הציעו כיצד ניתן לממש את __neg__ בזמן קבוע. תוכלו להוסיף שדות למחלקה אבל אין לשנות שדות קיימים. הסבירו כיצד ישתנו שאר המתודות של המחלקה כתוצאה משינוי זה.~~

ד. להלן 3 מימושים למתודה __hash__ של המחלקה SparseMatrix (כזכור מתודה זו היא האחראית על מה שיקרה כאשר נפעיל את הפונקציה hash המובנית של פייתון על אובייקט מהמחלקה).

מימוש אפשרי ראשון:

```
def __hash__(self):
   sorted_keys = sorted([(i,j) for i in self.rows for j in
self.rows[i]])
   #the keys are now sorted first by the value of the leftmost
          element
   #and then by the value of the rightmost element
   vals_str = ",".join(str(self[i,j]) for (i,j) in sorted_keys)
   return hash(vals_str)
```

מימוש אפשרי שני:

```
def __hash__(self):
   sorted_keys = sorted([(i,j) for i in self.rows for j in
          self.rows[i]])
   #the keys are now sorted first by the value of the leftmost
          element
   #and then by the value of the rightmost element
   keys_str = ",".join(str(i) + "," + str(j)  for (i,j) in sorted_keys)
   vals_str = ",".join(str(self[i,j]) for (i,j) in sorted_keys)
   return hash(keys_str + "," + vals_str)
```

מימוש אפשרי שלישי:

```
def __hash__(self):
   keys_sum = sum([i+j for i in self.rows for j in self.rows[i]])
   vals_sum = sum([self[i,j] for i in self.rows for j in
self.rows[i]])
   return hash(keys_sum + vals_sum)
```

להלן מטריצה דלילה, לה נקרא mat1:

```
0 6 0
0 0 1
0 3 0
```

מטריצה זו תתקבל ע"י הרצת הפקודות הבאות:

```
>>> mat1 = SparseMatrix(3,3)
>>> mat1[0,1] = 6
>>> mat1[1,2] = 1
>>> mat1[2,1] = 3
```

לכל אחד משלושת המימושים הללו של __hash__, כתבו בקובץ השלד דוגמה למטריצה אחרת באותם המימדים, **<u>שבהכרח</u>** תיצור התנגשות עם mat1 תחת hash.

את הדוגמאות יהיה עליכם להשלים בפונקציות הייעודיות <u>בקובץ השלד</u>: ()mat2a עבור המימוש הראשון, ()mat2b עבור המימוש השני, ו- ()mat2c עבור המימוש השלישי.

אם עבור סעיף מסויים לדעתכם אין מטריצה כזו, דאגו שהפונקציה הרלוונטית לסעיף תחזיר None.

לדוגמה: אם למשל החלטנו שהפעלת hash על המטריצה שכולה אפסים תחזיר את אותו הערך כמו הפעלת hash על המימוש הראשון אז נשלים את הפונקציה המתאימה ()mat2a באופן הבא:

```
def mat2a():
     mat = SparseMatrix(3,3)
     return mat
```

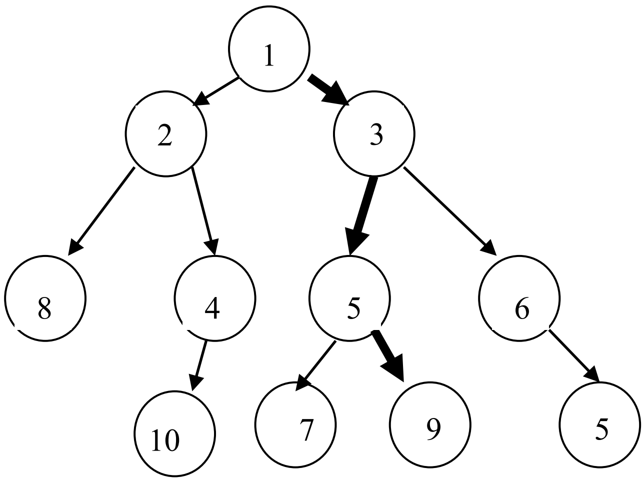
כלומר, יתקיים:

```
>>> hash(mat1) == hash(mat2a())
True
```

<u>שאלה 2</u>

השאלה עוסקת בעצים בינאריים, ובמחלקה Binary_search_tree. הניחו בשאלה זו שהמפתחות (השדה key) בצמתים הינם מחרוזות והערכים (השדה val) בצמתים הינם מספרים (int או float) גדולים ממש מ- 0 וקטנים או שווים ל- 100 . שימו לב שבכל הדוגמאות בסעיפים הבאים המפתחות הינם מטיפס str.

א. ממשו את המתודה `max_sum(self)` שמחזירה את הסכום המקסימלי של ערכים (values) במסלול משורש לעלה בעץ. ראו בדוגמא להלן את המסלול המודגש שסכומו הוא המקסימלי (שימו לב: **בכל צומת מופיע הערך בשדה val ולא ה key**!) . הסכום המקסימלי בעץ ריק הינו 0.

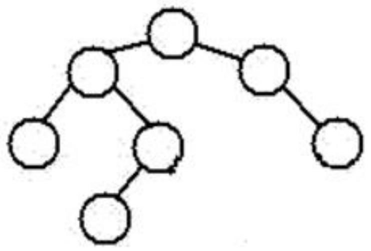


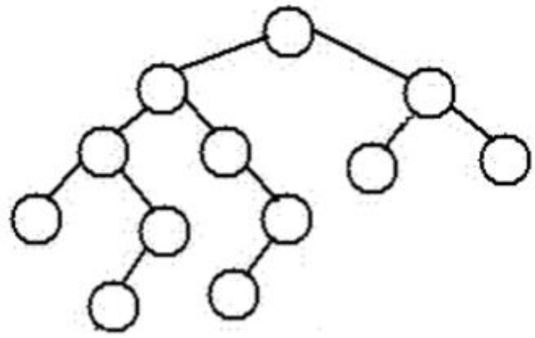
דוגמת הרצה (שמתאימה לעץ מעלה):

```
>>> t = Binary_search_tree()
>>> t.max_sum()
0
>>> t.insert('e', 1)
>>> t.insert('b', 2)
>>> t.max_sum()
3
>>> t.insert('a', 8)
>>> t.insert('d', 4)
>>> t.insert('c', 10)
>>> t.insert('i', 3)
>>> t.insert('g', 5)
>>> t.insert('f', 7)
>>> t.insert('h', 9)
>>> t.insert('j', 6)
>>> t.insert('k', 5)
>>> t.max_sum()
18
```

ב. עץ בינארי הוא <u>מאוזן</u> אם הוא עץ ריק או אם אינו ריק ומתקיימים שני התנאים הבאים:
1. ההפרש בין גובהו של תת-העץ השמאלי לבין גובהו של תת-העץ הימני הוא לכל היותר 1.
2. תת-העץ השמאלי שלו וגם תת-העץ הימני שלו הם עצים מאוזנים

לדוגמה, בתמונה שבעמוד הבא העץ השמאלי הוא מאוזן, אבל העץ הימני אינו מאוזן, כי ההפרש בין גובה תת-העץ השמאלי של השורש לבין גובה תת-העץ הימני של השורש הוא 2.





ממשו את המתודה is_balanced(self) אשר מחזירה True אם העץ מאוזן, ו False אחרת.

<u>הנחיה מחייבת</u>:

1. על המתודה לפעול ב O(n) כאשר n מספר הצמתים בעץ.
2. אין להוסיף שדות ל Tree_node.

דוגמת הרצה:

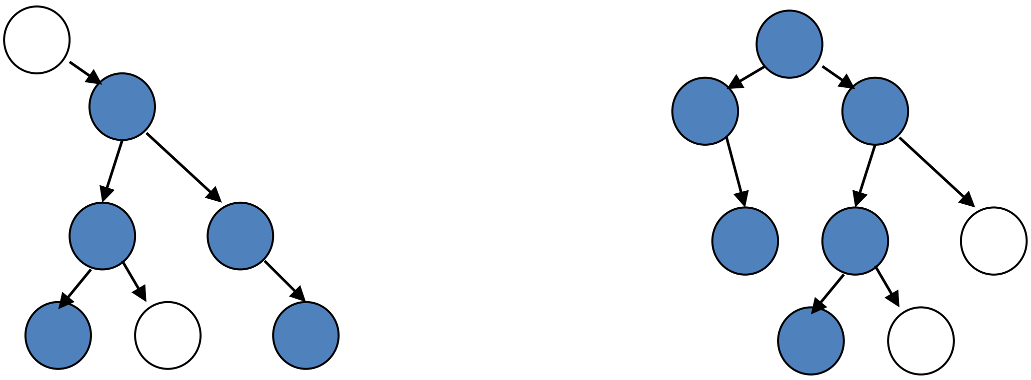
```
>>> t = Binary_search_tree()
>>> t.is_balanced()
True
>>> t.insert("b", 10)
>>> t.insert("d", 10)
>>> t.insert("a", 10)
>>> t.insert("c", 10)
>>> t.is_balanced()
True
>>> t.insert("e", 10)
>>> t.insert("f", 10)
>>> t.is_balanced()
False
```

ג. קוטר של עץ הינו אורכו של המסלול הארוך ביותר בין שני צמתים בעץ, כאשר צומת לא מופיע במסלול יותר מפעם אחת. אורך המסלול במקרה זה נקבע לפי מספר הצמתים במסלול. ייתכנו מספר מסלולים שונים שאורכם הוא קוטר העץ. ממשו את המתודה `diam(self)` שמחזירה את קוטר העץ. עבור עץ ריק תחזיר 0.

<u>הנחיה מחייבת</u>:

1. על המתודה לפעול ב O(n) כאשר n מספר הצמתים בעץ.
2. אין להוסיף שדות ל Tree_node.

בעמוד הבא מופיעות שתי דוגמאות בהן מודגש בכחול מסלול שאורכו הינו הקוטר:



דוגמאות הרצה:

```
>>> t2 = Binary_search_tree()
>>> t2.insert('c', 10)
>>> t2.insert('a', 10)
>>> t2.insert('b', 10)
>>> t2.insert('g', 10)
>>> t2.insert('e', 10)
>>> t2.insert('d', 10)
>>> t2.insert('f', 10)
>>> t2.insert('h', 10)
>>> t2.diam()
6
>>> t3 = Binary_search_tree()
>>> t3.insert('c', 1)
>>> t3.insert('g', 3)
>>> t3.insert('e', 5)
>>> t3.insert('d', 7)
>>> t3.insert('f', 8)
>>> t3.insert('h', 6)
>>> t3.insert('z', 6)
>>> t3.diam()
5
```

<u>הנחיות והבהרות לכלל השאלה</u>:

- בכל מקום בו נדרש לממש מתודה באופן רקורסיבי, מותר להגדיר פונקצית עזר רקורסיבית שתיקרא על ידי המתודה (שבמקרה זה תהיה "מתודת מעטפת").
  רצוי ואף מומלץ לממש את פונקצית העזר הרקורסיבית כפונקציה פנימית של המתודה. לדוגמה:

```
def some_method(self):
    def helper_rec( <some parameters> ):
        # some code
    return helper_rec( <some arguments> )
```

- המחלקה Tree_node שראיתם בכיתה מופיעה בקובץ השלד. כאמור, אין לשנות מחלקה זו.

שאלה 3

בשאלה זו נעסוק בהרחבה למבנה של רשימה מקושרת: רשימה מקושרת **דו-כיוונית** שתיוצג ע"י המחלקה DLList (doubly linked list).

צומת ברשימה מקושרת דו-כיוונית, שייצוג ע"י המחלקה Node, מכיל ערך (value) ומצביע לצומת הבא ברשימה (next). בנוסף על כך מכיל הצומת גם מצביע לאיבר הקודם ברשימה (prev). הרשימה עצמה (מטיפוס DLList) מכילה מצביע לצומת הראשון ברשימה (head), מצביע לצומת האחרון ברשימה (tail) ואת אורך הרשימה (כאמור, מספר הצמתים בה, גם כאן נקרא למשתנה זה len).

להלן מתודות האתחול של שתי המחלקות שהצגנו והמתודה len של המחלקה DLList:

```
class Node():

    def __init__(self, val):
        self.value = val
        self.next = None
        self.prev = None

class DLList():

    def __init__(self, seq=None):
        self.head = None
        self.tail = None
        self.len = 0
        if seq != None:
            for item in seq:
                self.insert(item)

    def __len__(self):
        return self.len
```

בקובץ השלד מומשה עבורכם המתודה __repr__ של המחלקה DLList ושל המחלקה Node.
שימו לב – מתודת האתחול של DLList משתמשת במתודה insert אותה נממש כעת.

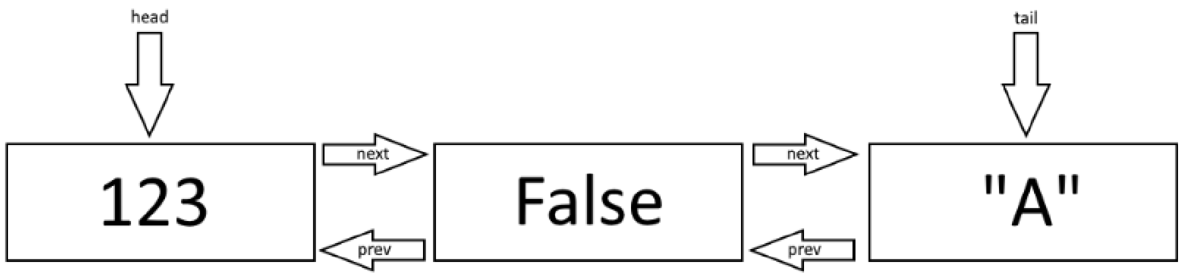
א. השלימו בקובץ השלד את המתודה insert של המחלקה DLList. המתודה מקבלת אובייקט מטיפוס DLList (self), ערך (val) ופרמטר בוליאני אופציונלי (first, שהינו False באופן דיפולטי). ופועלת באופן הבא:

- אם first == False המתודה מוסיפה את הערך val כצומת חדש בסוף הרשימה
- אם first == True המתודה מוסיפה את הערך val כצומת חדש בתחילת הרשימה

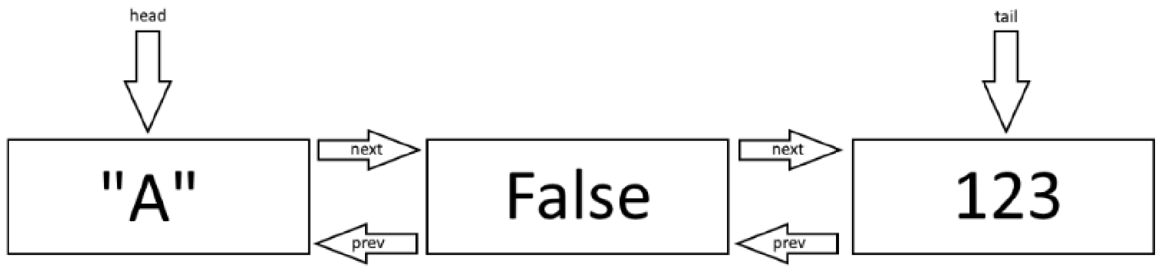
הנחיות:

- על המתודה לפעול בזמן O(1) ובזכרון O(1).
- על המתודה לעדכן את אורך הרשימה.

ב. השלימו בקובץ השלד את המתודה reverse של המחלקה DLList. המתודה פועלת על רשימה מקושרת דו-כיוונית והופכת את סדר האיברים שלה. לדוגמא, אם הרשימה lst נראית כך:



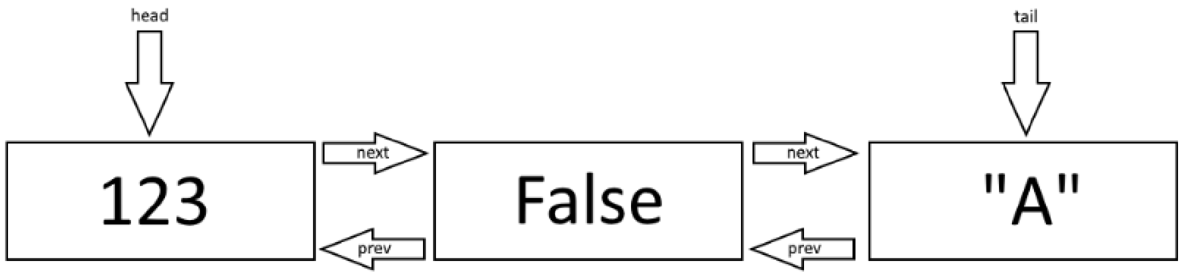
אזי לאחר הפעלת ()lst.reverse הרשימה תראה כך:



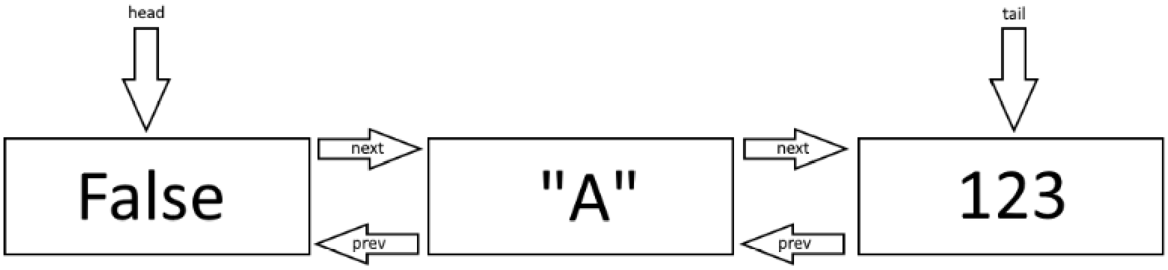
הנחיות:

- על המתודה לפעול בזמן O(n) ובזכרון O(1) כאשר n הוא מספר האיברים ברשימה
- על המתודה לשנות את הרשימה עליה היא נקראת (כלומר – לא ליצור רשימה חדשה)

ג. השלימו בקובץ השלד את המתודה rotate של המחלקה DLList. המתודה מקבלת אובייקט מטיפוס DLList (בשם self) ומספר שלם אי-שלילי (k) ומסיטה את איברי הרשימה k "צעדים" קדימה באופן מעגלי.
כלומר, לרשימה בת n איברים, אם נמספר את איברי הרשימה מ-0 עד n-1 ונבצע הזזה של k צעדים אזי האיבר ה-i ברשימה המקורית יהיה האיבר ה-$(i + k)\ \%\ n$ ברשימה החדשה.
לדוגמא, עבור הרשימה מהסעיף הקודם:



ופרמטר k=2 נקבל את הרשימה הבאה (מופיעה בעמוד הבא):



הנחיות:

- על המתודה לפעול בזמן O(m) כאשר $m = \min\{k, n-k\}$ ובסיבוכיות זכרון שקטנה מ O(n), כאשר n הוא מספר האיברים ברשימה (כלומר, פתרון שמשתמש בזיכרון עזר שגודלו $c \cdot n$ עבור קבוע כלשהו $c > 0$ אינו עומד בדרישה).
- כאמור, על המתודה לשנות את הרשימה עליה היא נקראת (כלומר – לא ליצור רשימה חדשה)
- ניתן להניח כי $k < n$

ד. השלימו בקובץ השלד את המתודה delete_node של המחלקה DLList. המתודה מקבלת אובייקט מטיפוס DLList (בשם self) ואובייקט מטיפוס צומת (node) אשר נמצא ברשימה. המתודה מוחקת את הצומת מהרשימה.

הנחיות:

- ניתן להניח כי הצומת node נמצא ברשימה ובפרט כי הרשימה לא ריקה
- על המתודה לפעול בזמן ובזכרון O(1)
- על המתודה לעדכן את אורך הרשימה.

שאלה 4

א. הוכיחו כי לכל $a, b, c \in \mathbb{N}$ מתקיים: $(a \bmod b)^c \bmod b = a^c \bmod b$

השתמשו בנוסחה הבאה, הידועה כנוסחת הבינום, עבור n טבעי:

$$(x+y)^n = \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} x^i y^{n-i}$$

ב. בתרגול ראינו את הפונקציה crack_DH אשר בהינתן $g, p, x = g^a \bmod p$ מנסה למצוא a' כך ש-$g^{a'} \bmod p = g^a \bmod p$. להלן תזכורת לקוד:

```python
def crack_DH(p, g, x):
    ''' find secret "a" that satisfies g**a%p == x
        Not feasible for large p '''
    for a in range(1, p - 1):
        if a % 100000 == 0:
            print(a) #just to estimate running time
        if pow(g, a, p) == x:
            return a
    return None #should never get here
```

בנוסף, ראינו כי יתכן שהפונקציה תחזיר $a' \neq a$ עבורו מתקיים השיוויון לעיל.

הוכיחו כי בהינתן $g,\ p,\ g^a \bmod p,\ g^b \bmod p$ ו-a' כך ש-$g^{a'} \bmod p = g^a \bmod p$ ניתן לחשב ביעילות את $g^{ab} \bmod p$.

רמז – היעזרו בסעיף הקודם.

שאלה 5

נתונה רשימה של n מחרוזות $[s_0, s_1, \dots, s_{n-1}]$, לאו דווקא שונות זו מזו. הניחו כי אורך כל אחת מהמחרוזות הוא לכל היותר m. בנוסף נתון k חיובי, וידוע שכל המחרוזות באורך לפחות k (ניתן להניח זאת בכל הפתרונות שלכם ואין צורך לבדוק או לטפל במקרים אחרים). אנו מעוניינים למצוא את כל הרביעיות הסדורות (i, j, a, b) כך שמתקיים:

א. $i < j$

ב. קיימת תת מחרוזת (רצופה) של s_i באורך k שחופפת לתת מחרוזת (רצופה) של s_j. תת המחרוזת החופפת מתחילה באינדקס a במחרוזת s_i ובאינדקס b במחרוזת s_j. כלומר, $s_i[a: a+k] == s_j[b: b+k]$.

לדוגמה, אם האוסף מכיל את המחרוזות

s0 = "aaac"

s1 = "caaaab"

s2 = "edcaac"

אז עבור k=3 יש חפיפה באורך k בין s0[0:3] לבין s1[1:4], וכן בין s0[0:3] לבין s1[2:5]. בנוסף, יש חפיפה באורך k בין s0[1:4] לבין s2[3:6], וגם בין s1[0:3] לבין s2[2:5].

א. נציע תחילה את השיטה הבאה למציאת כל החפיפות הנ"ל: לכל מחרוזת s_i נבדוק כל אחת מתתי המחרוזות באורך k שלה אל מול כל תתי המחרוזות באורך k של כל המחרוזות s_j האחרות כך ש $j > i$. ממשו את הפתרון הזה בקובץ השלד, בפונקציה substr_overlap (lst, k), אשר מקבלת רשימה (מסוג list של פייתון) של מחרוזות, וערך מספרי k, ומחזירה רשימה עם כל הרביעיות כמתואר מעלה. **אין חשיבות לסדר הרביעיות ברשימה, אך יש כמובן חשיבות לסדר הפנימי של הערכים בכל רביעיה.**

דוגמאות הרצה:

```
>>> s0 = "aaac"
>>> s1 = "caaaab"
>>> s2 = "edcaac"
>>> substr_overlap ([s0,s1,s2], 3)
[(0, 1, 0, 1), (0, 1, 0, 2), (1, 2, 0, 2), (0, 2, 1, 3)]
```

ב. ציינו מהי סיבוכיות הזמן של הפתרון הזה במקרה הגרוע, כתלות ב- m, n וב- k במונחים של $O(\ldots)$. הניחו כמובן כי השוואה בין שתי תת מחרוזות באורך k דורשת $O(k)$ פעולות במקרה הגרוע. ציינו גם מתי מתקבל המקרה הגרוע, בהנחה שהשוואת מחרוזות עוברת תו תו בשתי המחרוזות במקביל משמאל לימין, ומפסיקה ברגע שהתגלו תווים שונים.

ג. כעת נייעל את המימוש ונשפר את סיבוכיות הזמן (בממוצע), ע"י שימוש במנגנון של טבלאות hash. לשם כך נשתמש במחלקה חדשה בשם Dict, שחלק מהמימוש שלה מופיע בקובץ השלד. מחלקה זו מזכירה מאוד את המחלקה Hashtable שראיתם בהרצאה, אבל ישנם שני הבדלים: 1) בקוד מההרצאה האיברים בטבלה הכילו רק מפתחות (keys), בדומה ל- set של פייתון, ואילו אנחנו צריכים לשמור גם מפתחות וגם ערכים נלווים (values), בדומה לטיפוס dict של פייתון. המפתחות במקרה שלנו יהיו תתי מחרוזות באורך k. חישבו מה צריך להיות הערך אותו נשמור במילון עם כל מפתח. חישוב ה- hash לצורך הכנסה וחיפוש במילון מתבצע על המפתח בלבד. 2) מכיוון שיכולות להיות תת מחרוזות זהות, נרצה לאפשר חזרות של מפתחות ב- Dict (ראו בדוגמה בהמשך).

השלימו בקובץ השלד את המימוש של המתודה find(self, key) של המחלקה Dict, המתודה מחזירה רשימה (list של פייתון) עם **כל** ה- values שמתאימים למפתח key הנתון (לא חשוב באיזה סדר). אם אין כאלו תוחזר רשימה ריקה.

דוגמאות הרצה (המשך בעמוד הבא):

```
>>> d = Dict(3)
>>> d.insert(56, "a")
>>> d.insert(56, "b")
>>> d #calls __repr__
```

```
0 []
1 []
2 [[56, 'a'], [56, 'b']]

>>> d.find(56)
['a', 'b'] #order does not matter
>>> d.find(34)
[]
```

ד. השלימו את מימוש הפונקציה substr_overlap_hash1(lst, k), שהגדרתה זהה לזו של substr_overlap(lst, k), אלא שהיא תשתמש במחלקה Dict מהסעיף הקודם. נעבור על המחרוזות מהאחרונה לראשונה, ונשמור את תתי המחרוזות באורך k של כל מחרוזת במילון.

ה. ציינו מהי סיבוכיות הזמן של הפתרון הזה **בממוצע**, כתלות ב- n, k, m וכן באורך רשימת הפלט (מספר הרביעיות) אותו נסמן ב s. תנו תשובה במונחים של O(…). הניחו כמובן כי השוואה בין שתי תת מחרוזות באורך k דורשת $O(k)$ פעולות במקרה הגרוע, וכך גם חישוב hash על מחרוזת באורך k. נמקו את תשובתכם בקצרה.

ו. לסיום נרצה להשתמש במילון של פייתון (טיפוס dict), כדי לפתור את אותה הבעיה שוב. שימו לב להבדל מהפתרון הקודם: כאן אין לנו אפשרות לשנות את המימוש הפנימי של המחלקה dict, ובפרט איננו יכולים להחליט שחזרות של מפתחות מותרות (כזכור במילון של פייתון אין חזרות של מפתחות). השלימו את הפונקציה substr_overlap_hash2(lst, k) בהתאם למגבלה זו.

ז. בצעו השוואה של זמני ריצה בין שלושת הפתרונות, עבור רשימת מחרוזות באורכים של לפחות 500 תווים, הציגו את תוצאות ההשוואה בטבלה וצרפו את מסקנותיכם **כולל הסברים** (בקצרה) על סיבת ההבדלים בזמני הריצה. אין צורך לצרף את זמני הריצה עצמם.

סוף