חלק ו'

ניהול מספרים שלמים - שימוש במבני נתונים קיימים

חברת המעבדים "CPU for you" מעוניינת לנתח את תמונת הזיכרון של המחשב בכדי לתכנן מעבדים יעילים יותר. ראש צוות הפיתוח ביקש משלושה מהנדסים לכתוב מחלקה עם הפונקציונאליות שתפורט בהמשך.

צוות המהנדסים התבקש לבחון מספר פרמטרים שיוגדר במחלקה.

עליכם להשלים את שלד המחלקה שלרשותכם באמצעות שימוש במבני הנתונים שיוגדרו לכל משימה.

ניתן להשתמש במבני הנתונים שיצרתם בחלקים א'-ה', בכל מקום בו אין שימוש במבני נתונים מסעיפים קודמים בעבודה יהיה לרשותכם מימוש למבני הנתונים הנדרשים (מצורף בקובץ נפרד), זכרו כי בכל שימוש במבנה נתונים אין להתייחס לצורת המימוש של מבנה הנתונים אלא ל API שלו בלבד – ADT.

מחלקת NumsManagment:

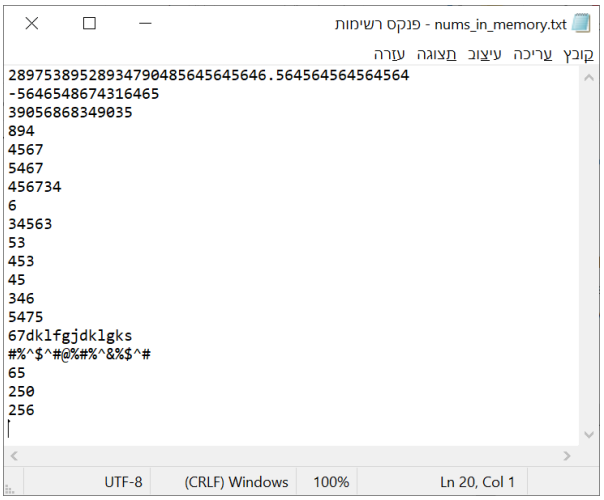
המחלקה ניהול מספרים (NumManagment) מוגדרת על ידי השדה הבא **בלבד**:

- **file_name** –מחרוזת לשם קובץ.

file_name יהיה קובץ טקסט. שורה בקובץ הטקסט שרלבנטית למחלקה זו תהיה רק שורה המכילה מספר שלם וחיובי (בלבד). שורה לא רלבנטית בקובץ הטקסט לא תטופל.

- ניתן להניח כי השורה האחרונה בקובץ ריקה.
- ניתן להניח כי למעט השורה האחרונה לא יהיו שורות ריקות בקובץ.
- ניתן להניח כי אין רווחים בסוף שורה בקובץ וכי יש \n בסוף כל שורה בקובץ.

דוגמה לקובץ טקסט (מסופקת לכם):



לרשותכם הבנאי:

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

בנאי (ממומש) מקבל את שם הקובץ כמחרוזת ושומר אותה בשדה המחלקה .

השלימו את מימוש המחלקה NumManagment:

```
def is_line_pos_int(self, st):
```

שיטה שמטרתה לבדוק האם שורה בקובץ מייצגת מספר שלם וחיובי (למעט התו '\n' בסוף השורה).

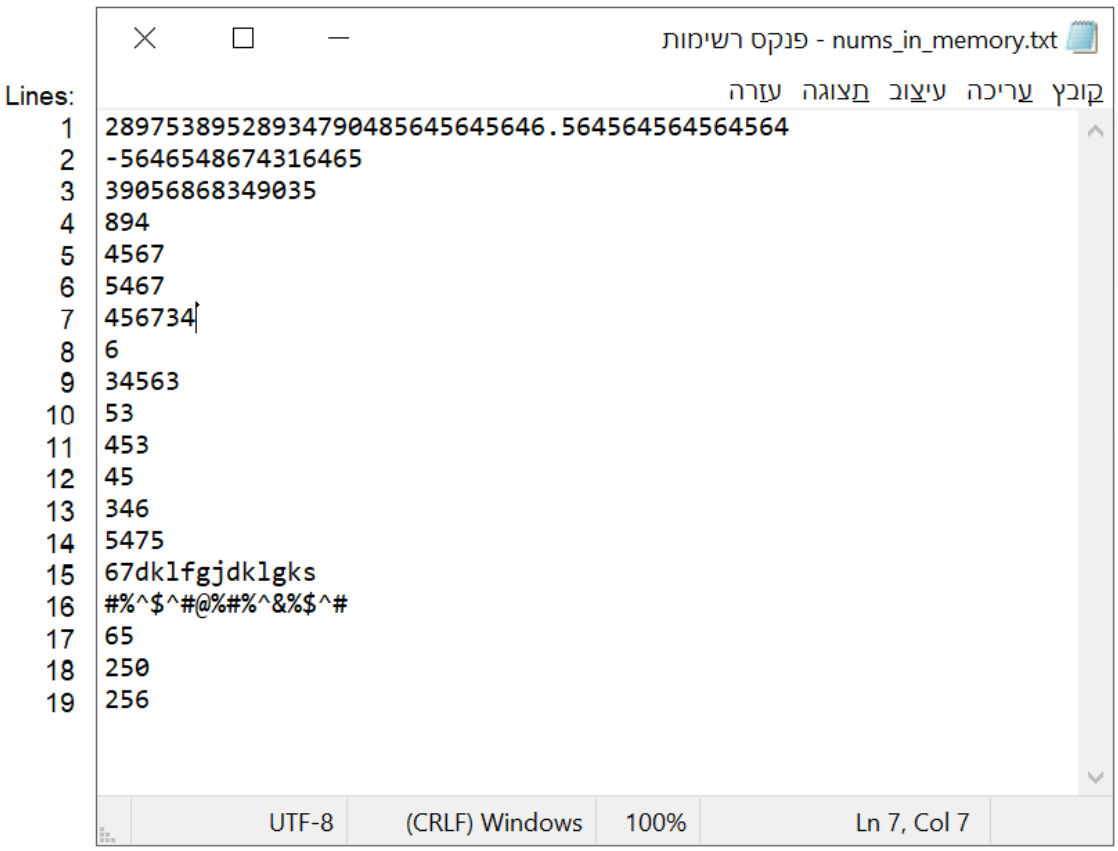
קלט:

- **[str]** `st` מחרוזת שנקראה מקובץ הטקסט באמצעות שיטה שתפורט בהמשך.

פלט:

- **[bool]** – True אם המחרוזת מייצגת מספר חיובי ושלם, False אם אחרת.

לדוגמא עבור הקובץ שתואר קודם, השורות המייצגות מספרים שלמים וחיוביים הן שורות 3-14 ושורות 17-19, בעוד שורות 1-2 ו-15-16 לא מייצגות מספרים שלמים וחיוביים:



יש לכתוב שיטה המחזירה גנרטור הקורא מהקובץ מספרים תקינים, ממשו את השיטה:

```
def read_from_file_gen(self):
```

שיטה המחזירה גנרטור שבכל איטרציה יחזיר ייצוג של המספר התקין הבא בקובץ. בקריאה ל-__next__ הגנרטור יחזיר את המספר הבא בייצוג בינארי באמצעות LinkedListBinaryNum. עליכם לזרוק חריגה מסוג FileNotFoundError כאשר נתקלתם בתקלה בעבודה עם הקובץ.

<u>פלט:</u>

- **[generator]** כפי שמפורט למעלה.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
nm = NumsManagment('nums_in_memory.txt')
gen = nm.read_file_gen()
print(type(gen))
print(next(gen).__repr__())
print(next(gen).__repr__())
```

יודפס:

```
<class 'generator'>
|00100011|10000101|10100010|11000011|10010100|01101011|
|00000011|01111110|
```

כאשר המחרוזת הראשונה מייצגת את 39056868349035 והשנייה את 894, שני המספרים התקינים הראשונים בקובץ.

עבור חקר תפוסת הזיכרון של מספרים בינאריים בזיכרון (יפורט בהמשך) ניתנה לכם גישה למימוש של מבנה נתונים מסוג מחסנית, עם הממשק הבא:

- Stack – בנאי של מחסנית ריקה
- push – שיטה המקבלת ערך ומכניסה למחסנית
- pop – שיטה המחזירה ומוציאה ערך מהמחסנית
- top – שיטה המחזירה אך לא מוציאה ערך מהמחסנית
- __len__ - שיטה המחזירה את כמות הערכים במחסנית
- is_empty – שיטה המחזירה האם אין ערכים במחסנית
- __repr__ - מחזירה מחרוזת המייצגת המחסנית עם ציון ראש המחסנית בפורמט שיוצג בהמשך

יש לממש שיטה המשתמשת בגנרטור מהסעיף הקודם ומחזירה מחסנית המכילה את המספרים התקינים מהקובץ – **בשיטה יש להשתמש במבנה נתונים Stack בלבד**, ניתן להגדיר יותר מאובייקט אחד מסוג Stack אם אתם מוצאים לנכון, ממשו את השיטה:

```
def stack_from_file(self):
```

פלט:

- **[Stack]** מחסנית המכילה את כל המספרים התקינים מהקובץ.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
nm = NumsManagment('nums_in_memory.txt')
s = nm.stack_from_file()
print(s)
```

יודפס:

```
[|00100011|10000101|10100010|11000011|10010100|01101011|,|00000011|01111110|
,|00010001|11010111|,|00010101|01011011|,|00000110|11111000|00011110|,|000001
10|,|10000111|00000011|,|00110101|,|00000001|11000101|,|00101101|,|00000001|010
11010|,|00010101|01100011|,|01000001|,|11111010|,|00000001|00000000|]<=Top
```

כחלק מחקר תפוסת הזיכרון של המספרים הבינאריים תוך מתן חשיבות למספרים הגדולים יותר, כתבו שיטה המקבלת מחסנית המכילה מספרים בינאריים ומחזירה מחסנית ממוינת של אותם מספרים כך שהאיבר בראש המחסנית יהיה המספר הגדול ביותר – אין להשתמש בשיטה במבני נתונים מלבד Stack, **ניתן להשתמש במספר אובייקטים מסוג Stack** אם אתם מוצאים לנכון, ממשו את השיטה:

```
def sort_stack_descending(self, s):
```

המקבלת מחסנית המכילה מספרים בינאריים בלבד ומחזירה מחסנית המכילה את אותם המספרים ממוינים.

קלט:

- **s [Stack[LinkedListBinaryNum]]** מחסנית המכילה מספרים בינאריים.

פלט:

- **[Stack]** מחסנית המכילה את הערכים שהיו ב – s אך ממויינים כך שהאברים יצאו מהמחסנית בסדר יורד.

הנחיות:

- ניתן לשנות את הפרמטר s.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
nm = NumsManagment('nums_in_memory.txt')
s = nm.sort_stack_descending(nm.stack_from_file())
print(s)
```

יודפס:

```
[|00000110|,|00101101|,|00110101|,|01000001|,|11111010|,|00000001|00000000|,|000
00001|01011010|,|00000001|11000101|,|00000011|01111110|,|00010001|11010111|,|0
0010101|01011011|,|00010101|01100011|,|10000111|00000011|,|00000110|11111000|
00011110|,|00100011|10000101|10100010|11000011|10010100|01101011|]<=Top
```

בכדי לבחון מופעים של ערכים של בתים בזיכרון כתבו שיטה היוצרת תור של מספרים בינאריים, ממשו את השיטה:

```
def queue_from_file(self):
```

<u>פלט:</u>

- **[Queue]** תור המכיל את כל המספרים התקינים מהקובץ.

<u>הנחיות:</u>

- ניתן להשתמש בתור שהגדרתם בעבודה, כולל האיטרטור.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
q = nm.queue_from_file()
q.enqueue(LinkedListBinaryNum(17))
q.enqueue(LinkedListBinaryNum(3))
print(q)
```

יודפס:

```
Newest=>[|00000011|,|00010001|,|00000001|00000000|,|11111010|,|01000001|,|0001
0101|01100011|,|00000001|01011010|,|00101101|,|00000001|11000101|,|00110101|,|1
0000111|00000011|,|00000110|,|00000110|11111000|00011110|,|00010101|01011011|
,|00010001|11010111|,|00000011|01111110|,|00100011|10000101|10100010|1100001
1|10010100|01101011|]<=Oldest
```

יש לייצר כל ערכי הבתים המופיעים בתור של מספרים בינאריים (פלט של הסעיף הקודם) אך מכיוון שכל בית יכול להופיע מספר רב של פעמים על אוסף ערכי הבתים להיות ללא כפילויות, ממשו את השיטה:

```
def set_of_bytes(self, q_of_nums):
```

המקבלת תור מספרים בינאריים ומחזירה set של ערכי בתים ללא כפילויות תוך **שימוש במבנה נתונים מסוג תור ו – set בלבד**.

קלט:

- **`q_of_nums`** **[Queue[LinkedListBinaryNum]]** תור המכיל מספרים בינאריים כפי שתואר בסעיף הקודם.

פלט:

- **[set]** סט של ערכי בתים מהמספרים הבינאריים.

הנחיות:

- **אין משמעות לסדר הבתים המוחזרים בסט.**
- ניתן ליצור אובייקטים נוספים מסוג תור – בלבד - אם אתם מוצאים לנכון.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
q = nm.queue_from_file()
q.enqueue(LinkedListBinaryNum(17))
q.enqueue(LinkedListBinaryNum(3))
bytes_set = nm.set_of_bytes(q)
print(isinstance(bytes_set, set))
print(sorted(bytes_set))
```

יודפס:

```
True
['00000000', '00000001', '00000011', '00000110', '00010001', '00010101', '00011110',
'00100011', '00101101', '00110101', '01000001', '01011010', '01011011', '01100011',
'01101011', '01111110', '10000101', '10000111', '10010100', '10100010', '11000011',
'11000101', '11010111', '11111000', '11111010']
```

עבור בדיקה של מספרים עוקבים יש ליצור שיטה המחזירה עץ חיפוש בינארי המכיל את כל המספרים השלמים והחיוביים בקובץ, לצורך כך מסופקת לכם מחלקת עץ חיפוש בינארי עם ה API הבא:

TreeNode – מחלקת חוליית עץ חיפוש בינארי

- TreeNode(key, val) – בנאי חוליית עץ חיפוש בינארי המקבל מפתח וערך
- successor – שיטה המחזירה את החולייה עם הערך הנמוך ביותר שגבוה מהחולייה הנוכחית

BinarySearchTree – מחלקת עץ חיפוש בינארי

- BinarySearchTree – בנאי היוצר עץ ריק
- insert(key, val) – שיטה המקבלת מפתח וערך ומכניסה אותם לעץ ממוינים ע"פ המפתח
- minimum – שיטה המחזירה את החולייה של המפתח המינימלי בעץ
- __iter__ - שיטה המחזירה איטרטור למחלקה
- __next__ - in_order שיטה המחזירה את האיבר הבא לפי
- __repr__ - in_order שיטה המחזירה מחרוזת המייצגת את העץ, לפי

ממשו את השיטה:

```
def nums_bst(self):
```

מחזירה עץ חיפוש בינארי המכיל את כל המספרים התקינים מהקובץ, עבור שגיאה בפתיחת הקובץ יש לזרוק חריגה מסוג FileNotFoundError, המפתח של כל חוליה בעץ יהיה הערך של המספר בבסיס דצימלי (int) וערכו יהיה LinkedListBinaryNum המייצג את אותו המספר.

<u>פלט:</u>

- **[BinarySearchTree]** כפי שמפורט בסעיף.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא:

```
bst = nm.eng3_nums_bst()
print(bst)
```

יודפס:

```
(6,|00000110|)
(45,|00101101|)
(53,|00110101|)
(65,|01000001|)
(250,|11111010|)
```

(256,|00000001|00000000|)

(346,|00000001|01011010|)

(453,|00000001|11000101|)

(894,|00000011|01111110|)

(4567,|00010001|11010111|)

(5467,|00010101|01011011|)

(5475,|00010101|01100011|)

(34563,|10000111|00000011|)

(456734,|00000110|11111000|00011110|)

(39056868349035,|00100011|10000101|10100010|11000011|10010100|01101011|)

בכדי לקבל מספרים עוקבים יש למצוא את שני המספרים הקרובים ביותר בקובץ ולייצר עבורם את הטווח, ממשו את השיטה:

```
def bst_closest_gen(self, bst):
```

המקבלת עץ חיפוש בינארי (אין צורך לבדוק) ומחזירה גנרטור.

<u>קלט:</u>

- **bst [BinarySearchTree[LinkedListBinaryNum]]** – עץ חיפוש של מספרים בינאריים.

<u>פלט:</u>

- **[generator]** – גנרטור כפי שפורט.

לדוגמא עבור הקובץ המצורף והרצת הקוד הבא (שני המספרים הקרובים ביותר הם 250 ו-256):

```
bst = nm.nums_bst()
gen = nm.bst_closest_gen(bst)
print(type(gen))
for num in gen:
    print('(' + str(num[0]) + ',' + str(num[1]) + ')')
```

יודפס:

```
<class 'generator'>
(250,|11111010|)
(251,|11111011|)
```

(252,|11111100|)

(253,|11111101|)

(254,|11111110|)

(255,|11111111|)

(256,|00000001|00000000|)

בהצלחה בעבודה ובבחינות! אוריאל.