פרק 8
מחסנית ותור

בפרק זה נציג שני סוגי אוספים כלליים. שני הסוגים מאפשרים ליצור אוספים דינמיים, לא מוגבלים בגודלם. כל אחד מהם מאופיין על ידי פרוטוקול המגדיר דרך גישה ספציפית לנתונים שבאוסף, כולל אופן ההכנסה וההוצאה של נתונים. לכל אחד משני הסוגים נציג שתי מחלקות המממשות אותו: מחלקה אחת מייצגת את הנתונים בעזרת מערך, והאחרת מייצגת אותם בעזרת שרשרת חוליות. שתי המחלקות, המממשות כל אחד משני סוגי האוסף, תומכות בדיוק באותן פעולות ממשק, כולל פרוטוקול הגישה לנתונים, ושתיהן מסתירות לגמרי את הייצוג והמימוש, כך שלקוח אינו יכול לדעת באיזה משני המימושים הוא משתמש, ואינו יכול לפגוע בנכונות המימוש. מכאן ששני סוגי האוספים שיוצגו בפרק זה הם טיפוסי נתונים מופשטים.

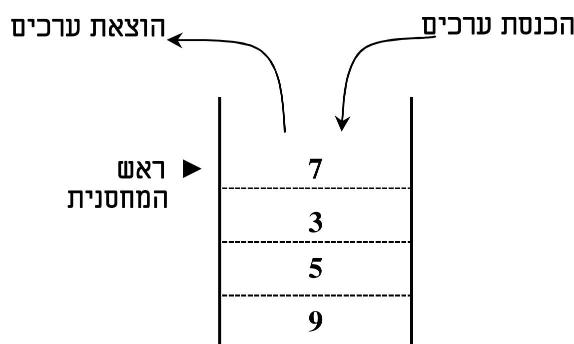
א. המחסנית

מחסנית (**Stack**) היא סוג של אוסף התומך בפעולות הכנסה והוצאה, כך שמתקיימת התכונה הבאה: הוצאת ערך ממחסנית אפשרית רק כאשר היא אינה ריקה והיא מחזירה תמיד את הערך שהוכנס אחרון, מבין הערכים הקיימים במחסנית. ניתן לחשוב על מחסנית כסדרה, שהפעולות עליה מתבצעות רק בקצה אחד של הסדרה, הנקרא **ראש המחסנית** (כמתואר באיור). הכנסת ערך למחסנית מוסיפה ערך חדש בראש המחסנית. הוצאת ערך מהמחסנית, מסירה את הערך המצוי בראש המחסנית וחושפת את הערך הבא בסדרה. נהוג לכנות את פעולת ההכנסה למחסנית בשם דחיפה, ואת פעולת ההוצאה בשם: שליפה.

קל לראות מתיאור זה שהערך הנשלף מהמחסנית הוא תמיד הערך האחרון שנדחף אליה. הפרוטוקול המגדיר את דרך הגישה לערכים נקרא **LIFO** – ראשי התיבות של המילים: **L**ast **I**n **F**irst **O**ut.

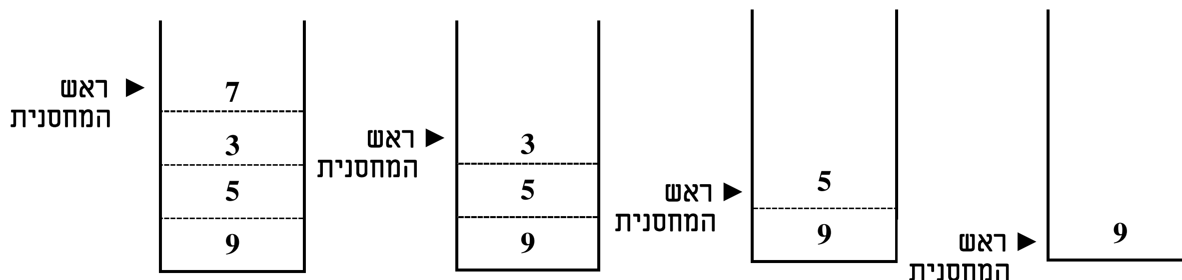
האיור שלפניכם הוא "תצלום" מצב רגעי של מחסנית לאחר שהוכנסו לתוכה הערכים:

9, 5, 3, 7, בזה אחר זה (מימין לשמאל).



ניתן לראות שהערך 7, שהוכנס אחרון, נמצא בראש המחסנית.

באיור הבא ניתן לראות סדרת תצלומי מצב של המחסנית, המתארים (מימין לשמאל), את המחסנית לאחר הכנסת כל אחד מהערכים לתוכה :

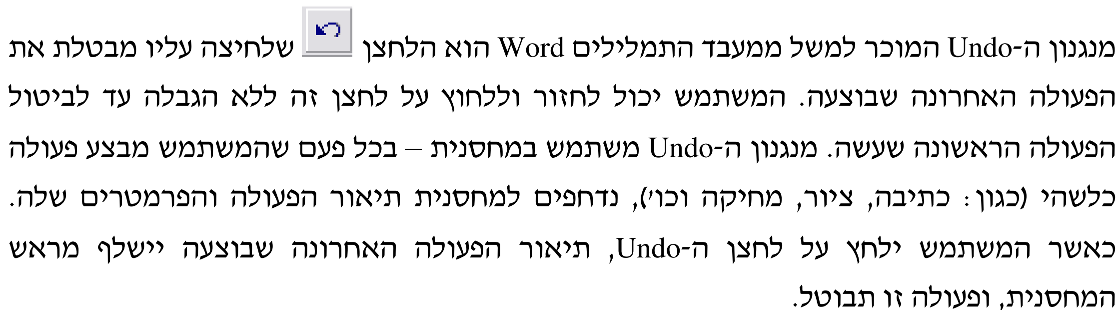


אם לאחר השלב הרביעי המוצג נשלוף ערך מהמחסנית, יוצא הערך 7, והמחסנית תחזור למצב שלפני האחרון.

ב. שימושים במחסנית

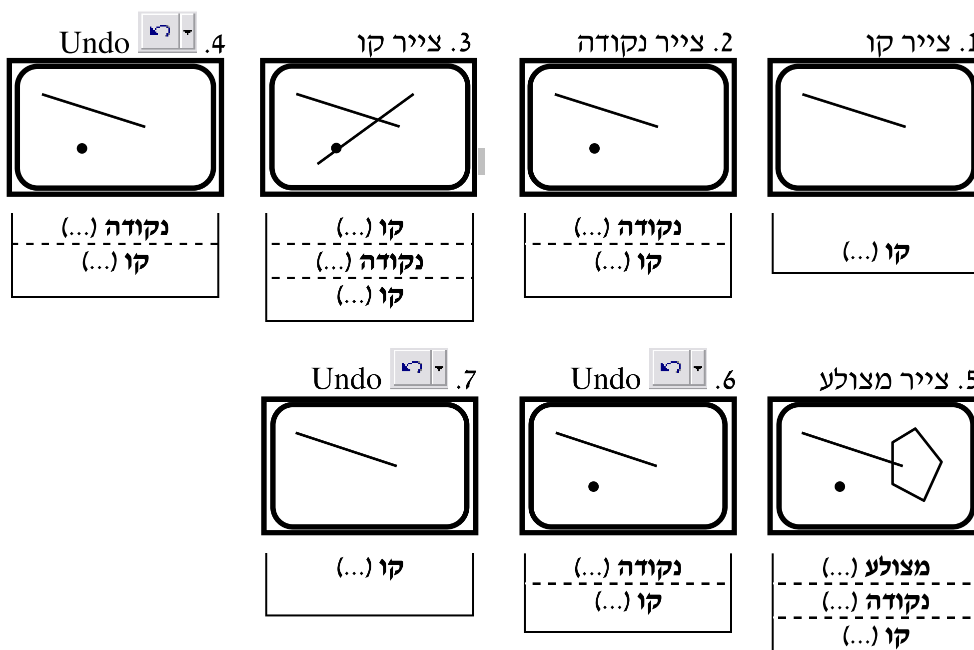
המחסנית משתלבת באופן טבעי ביישומים שבהם יש צורך לאחזר נתונים באופן הפוך לסדר קליטתם. להלן מספר דוגמאות :

דוגמה 1: מנגנון ה-Undo (ביטול פעולה אחרונה)



מנגנון ה-Undo המוכר למשל ממעבד התמלילים Word הוא הלחצן שלחיצה עליו מבטלת את הפעולה האחרונה שבוצעה. המשתמש יכול לחזור וללחוץ על לחצן זה ללא הגבלה עד לביטול הפעולה הראשונה שעשה. מנגנון ה-Undo משתמש במחסנית – בכל פעם שהמשתמש מבצע פעולה כלשהי (כגון : כתיבה, ציור, מחיקה וכו'), נדחפים למחסנית תיאור הפעולה והפרמטרים שלה. כאשר המשתמש ילחץ על לחצן ה-Undo, תיאור הפעולה האחרונה שבוצעה יישלף מראש המחסנית, ופעולה זו תבוטל.

לדוגמה, המשתמש ביצע 7 פעולות בזו אחר זו. באיור שלפניכם ניתן לראות את מצבם של המסך ושל מחסנית ה-Undo לאחר כל פעולה שהמשתמש ביצע :



דוגמה 2: מחסנית זמן ריצה

עוד דוגמה לשימוש במחסנית היא **מחסנית זמן ריצה** (**run time stack**). בעזרת מנגנון זה המחשב עוקב אחר זרימת התוכנית כאשר יש בה פעולות המזמנות זו את זו.

כאשר יש זימון לפעולה מסוימת, נדחפת לתוך המחסנית רשומה ובה בין השאר, ערכי הפרמטרים לזימון והנקודה שאליה צריכה התוכנית לחזור לאחר ביצוע הזימון. כאשר הפעולה מסתיימת, נשלפת הרשומה שבראש המחסנית וביצוע התוכנית ממשיך מנקודת החזרה הרשומה בה.

דוגמה 3: בדיקת תקינות סוגריים

ביטוי חשבוני מכיל לעתים קרובות סוגריים מסוגים שונים. נגדיר מהו ביטוי חשבוני תקין מבחינת סוגריים: ביטוי המכיל סוגריים מסוגים שונים, במספר לא מוגבל, ובלבד שיהיו מאוזנים. איזון הסוגריים מחייב שמספר הסוגריים-הפותחים והסוגריים-הסוגרים יהיה שווה, וכן לכל פותח יימצא סוגר מאותו סוג במקום המתאים.

לדוגמה, הביטויים האלה תקינים: ((a))

(b + a - 2 * 7)

{ + 32 * (37 *) / [5 + 1]} - 4

(שימו לב: הביטוי האחרון תקין מבחינת הסוגריים, אף שכביטוי חשבוני הוא אינו תקין).

ואילו הביטויים האלה אינם תקינים: a + ((c

([3 + a) + 4]

[ ) (5 - 3] * [2 - 3]

בבדיקת תקינות סוגריים אנו מתמקדים רק בפותחים ובסוגרים המופיעים בביטוי. כאשר קיימים בביטוי סוגי סוגריים שונים, אין זה מספיק למנות את הסוגריים, צריך גם לדעת מהו סדר הופעתם. בעיה זו דומה למנגנון ה-Undo שעסקנו בו קודם. גם שם היה צורך לזכור את הפעולות שהופיעו ואת סדר הופעתן.

מתיאור הבעיה נראה שמחסנית היא הכלי הנכון לבדיקת האיזון. נסרוק את הביטוי משמאל לימין. בכל פעם שניתקל בפותח, נדחוף אותו למחסנית. כאשר נגיע לסוגר נשלוף איבר מראש המחסנית ונבדוק אם הוא הפותח המתאים בסוגו לסוגר הנוכחי. אם כן, התת-ביטוי שבין הפותח לסוגר תקין, שכן אין בו סוגריים נוספים, או שיש בו זוגות מאוזנים של סוגריים. אם הסוגר והפותח אינם מאותו הסוג, הביטוי בוודאי אינו תקין, אחרי הודעה מתאימה נעצור את תהליך הבדיקה.

נמשיך בבדיקה באותו האופן. כל פותח שנקרא מהקלט יידחף למחסנית, וכל סוגר יוביל לניסיון לשלוף פותח מהמחסנית.

תהליך הבדיקה יסתיים באחד משלושת המקרים האלה:

- אם בשלב מסוים הסוגר והפותח הנבחנים אינם מתאימים, הביטוי אינו תקין.
- אם הגענו לסוגר והמחסנית ריקה, הביטוי אינו תקין.
- כאשר הגענו לסוף הקלט: אם המחסנית לא התרוקנה, סימן שנותר פותח שטרם הותאם לו סוגר, והביטוי אינו תקין; אם המחסנית ריקה, הביטוי תקין.

בדיקת תקינות סוגריים משמעותית גם בעת הידור של תוכניות מחשב. גם כאן מופיעים סוגי סוגריים שונים, וחוסר איזון והתאמה בין כמות הפותחים והסוגרים יגרום לבעיות הידור.

לסיכום: בכל הדוגמאות שראינו בא לידי ביטוי פרוטוקול ה-LIFO. בדוגמת ה-Undo הפעולה האחרונה שהתבצעה ממוקמת בראש המחסנית. פעולה זו תישלף בעת ביצוע ה-Undo, ואנו נחזור למצב שהיה לפני ביצועה. בדוגמה של מחסנית זמן ריצה, הרשומה המתארת את מיקום הזימון האחרון בתוכנית ממוקמת בראש המחסנית. בתום זימון זה ועם שליפת הרשומה ניתן להמשיך בריצה תקינה של התוכנית. בדוגמה לבדיקת איזון סוגריים, הפותח האחרון ממוקם בראש המחסנית. כאשר ייקרא סוגר, יישלף הפותח ויושוֶוה לסוגר. על פי ההשוואה נחליט כיצד להתקדם בבדיקה.

ג. ממשק המחלקה מחסנית

לפני שנציג מחלקה Stack, המממשת את סוג האוסף שתיארנו, הבה נבחן אילו פעולות יש לכלול בממשק שלה. פעולות הממשק ודאי יכללו את פעולות ההכנסה וההוצאה שהזכרנו לעיל. לצורכי נוחות, מקובל להפריד את השליפה ממחסנית מפעולת הצצה לתוכה המאפשרת לראות את הערך השמור בראש המחסנית מבלי להוציאו ממנה. מהדוגמאות לשימוש במחסנית, שהוצגו למעלה, עולה בבירור הצורך להגדיר פעולה הבודקת את ריקנות המחסנית: פעולת השליפה ממחסנית אפשרית רק אם יש במחסנית איברים. כמו כן סיום תוכניות המשתמשות במחסנית יהיה תלוי פעמים רבות בהגעה למצב של מחסנית ריקה. לכן יש להציע בממשק בדיקה האם המחסנית ריקה או לא. ולסיום, כמקובל לגבי כל טיפוס נתונים המוגדר בעזרת מחלקה, יש להגדיר פעולה המחזירה את תיאור המחסנית וכן להגדיר פעולה בונה.

המחסנית היא סוג של אוסף כללי, והפעולות עליה אינן משתמשות בתכונות או בפעולות ייחודיות לנתונים שבה, לכן נגדיר את המחסנית כמחלקה גנרית.

ממשק המחלקה מחסנית <Stack<T>

המחלקה מגדירה טיפוס אוסף בעל פרוטוקול LIFO להכנסה והוצאה של ערכים.

הפעולה בונה מחסנית ריקה	Stack()
הפעולה מחזירה 'אמת' אם המחסנית הנוכחית ריקה, 'שקר' אחרת	**boolean** isEmpty()
הפעולה מכניסה את הערך x לראש המחסנית הנוכחית (דחיפה)	**void** push (T x)
הפעולה מוציאה את הערך שבראש המחסנית הנוכחית ומחזירה אותו (שליפה). **הנחה**: המחסנית הנוכחית אינה ריקה	T pop()
הפעולה מחזירה את הערך שבראש המחסנית הנוכחית מבלי להוציאו. **הנחה**: המחסנית הנוכחית אינה ריקה	T top()
הפעולה מחזירה תיאור של המחסנית, כסדרה של ערכים, במבנה הזה (x_1 הוא האיבר שבראש המחסנית): $[x_1, x_2, \ldots, x_n]$	String toString()

הממשק <Stack<T> הוא הגדרה של טיפוס נתונים מופשט, שכן הוא מתייחס רק לפעולות של המחסנית ולתכונותיהן, ואינו עוסק כלל בייצוג המחסנית.

נזכיר כי בעת יצירת מחסנית יש לציין את הטיפוס הקונקרטי של המחסנית. אנו נשתמש בנוסחים מקוצרים בסגנון: "מחסנית מטיפוס מסוים" במקום "מחסנית שאיבריה מטיפוס מסוים". למשל, באומרנו "מחסנית מטיפוס תו" או "מחסנית תווים" נתכוון למחסנית שאיבריה הם מטיפוס תו. הקיצורים הללו ישמשו אותנו גם בדיונים על אוספים נוספים בהמשך היחידה.

ד. שימוש בפעולות הממשק

ד.1. קוד לדוגמה

בקוד שלפניכם נוצרות מחסניות מטיפוסים שונים. המחסניות מפעילות את פעולות הממשק. לצד הקוד מופיעים איורים המדגימים את מצב המחסניות תוך כדי ריצת התוכנית:

```java
public static void main(String[] args)
{
    // יצירת מחסניות מטיפוסים שונים //
    Stack<Integer> s1 = new Stack<Integer>();
    Stack<Integer> s2 = new Stack<Integer>();
    Stack<String>  s3 = new Stack<String>();

    // דחיפת ערכים למחסניות //
    s1.push(4);
    s1.push(7);
    s1.push(2);
    s2.push(11);
    s3.push("Moshe");

    // שליפת ערך ממחסנית אחת ודחיפתו לאחרת //
    int x = s1.pop();
    s2.push(x);

    // העתקת ערך מראש מחסנית למחסנית אחרת //
    x = s1.top();
    s2.push(x);

    // שליפה מותנית //
    if (!s2.isEmpty())
        s1.push(s2.pop());
}
```

מחסנית מחרוזות	מחסנית שלמים	מחסנית שלמים
s3: (ריקה)	s2: (ריקה)	s1: (ריקה)
s3: Moshe	s2: 11	s1: 2, 7, 4
s3: Moshe	s2: 2, 11	s1: 7, 4
s3: Moshe	s2: 7, 2, 11	s1: 7, 4
s3: Moshe	s2: 2, 11	s1: 7, 7, 4

ד.2. היפוך סדרת תווים שאורכה לא ידוע

נכתוב תוכנית הקולטת סדרת של תווים, תו אחר תו. הסדרה מסתיימת בלחיצת מקש ה-Enter. בתום הקליטה תודפס סדרת התווים בסדר הפוך לסדר קליטתה.

את המשימה מתאים לפתור בעזרת מחסנית מכמה סיבות. ראשית, המטרה הסופית של התוכנית היא היפוך סדר התווים, ולכן שימוש במחסנית עולה על הדעת שכן הוצאת האיברים ממחסנית הופכת את הסדר שבו נקלטו התווים. שנית, אורך הקלט אינו מוגבל, ולכן האופי הדינמי של המחסנית ישרת אותנו היטב בביצוע משימה זו.

עקבו אחר התוכנית שלפניכם, המציגה פתרון לבעיה תוך שימוש במחסנית של תווים:

```
public static void main(String[] args)
{
    // בניית מחסנית עזר לשמירת התווים הנקלטים
    Stack<Character> stk = new Stack<Character>();

    // קליטת סדרת תווים
    System.out.print("Enter a sequence of characters: ");
    Scanner in = new Scanner(System.in);
    in.useDelimiter("");// קליטת סדרת תווים רציפה

    char ch = in.next().charAt(0);
    while (ch != '\r')
    {
        stk.push(ch);
        ch = in.next().charAt(0);
    }

    // הדפסת סדרת התווים שנקלטה במהופך
    System.out.print("The reversed sequence: ");
    while (!stk.isEmpty())
        System.out.print(stk.pop());
}
```

ה. כתיבת המחלקה מחסנית

עצם מטיפוס מחסנית שומר בתוכו את אוסף הערכים שהוכנסו למחסנית תוך שמירה על סדר הכנסתם. לכן בכתיבת מחלקה למימוש הממשק שלעיל עלינו להחליט, ראשית לכול, על ייצוג מתאים לאחסון ערכים אלה. לאחר מכן נממש את פעולות ממשק המחסנית על פי ייצוג זה.

ה.1. שני ייצוגים למחלקה מחסנית

בפרקים הקודמים ראינו שקיימות שתי דרכים לאחסן אוסף בזיכרון: הדרך הסטטית באמצעות מערך, והדרך הדינמית באמצעות שרשרת חוליות.

תחילה, נסקור בקצרה את ייצוג המחסנית באמצעות מערך.

ה.1.1. ייצוג מחסנית באמצעות מערך

הייצוג של המחלקה:

```
public class Stack<T>
{
     public static final int STACK_SIZE = 100;
     private T[] data;
     private int top;
```

משתנה בשם top מטיפוס **int** יכיל את מספר התא במערך שבו נמצא הערך שבראש המחסנית. כאשר המחסנית ריקה ערכו של top יהיה שווה ל-(1-). בעת דחיפת ערך למחסנית נגדיל את ערכו של top באחד, ונציב את הערך החדש בתא שמספרו top. כאשר נרצה לשלוף ערך מן המחסנית, נחזיר את הערך שבמקום top, ונקטין את ערכו של top באחד.

כיוון שבעת יצירת המערך עלינו לקבוע את גודלו, נגדיר את המשתנה STACK_SIZE. באופן שרירותי נקבע את ערכו ל-100.

מה יקרה אם בפועל המשתמש יכניס למחסנית רק 3 ערכים במשך כל התוכנית? המערך עדיין יהיה בגודל 100 (שכן המערך מוגדר בפעולה הבונה של המחסנית, לפני פעולות ההכנסה) והזיכרון של 97 תאים נוספים יתבזבז. מה יקרה אם המשתמש ירצה להכניס 101 ערכים? אזי כאשר המשתמש יפעיל את הפעולה (...)push להכנסת הערך ה-101, קוד הפעולה יגלה כי אין מקום במערך, יקצה מערך גדול יותר, יעתיק לשם את הערכים מהמערך הקיים, ורק אז יוסיף את הערך הנוסף. ברור שמחירה של פעולת ההכנסה במקרה זה גבוה באופן משמעותי יחסית למקרה שבו המערך אינו מלא.

? כתבו את המחלקה Stack<T> כאשר היא מיוצגת על ידי מערך.

לסיכום: החיסרון שבבחירת מערך כמייצג של המחסנית הוא בכך שהמערך הוא זיכרון סטטי ואינו יכול לגדול ולקטון כרצוננו. "הגדלת" מערך פירושה למעשה הקצאת מערך חדש, גדול יותר, והעתקת תוכן המערך הנתון אל המערך החדש.

ה.2.1. ייצוג מחסנית באמצעות שרשרת חוליות

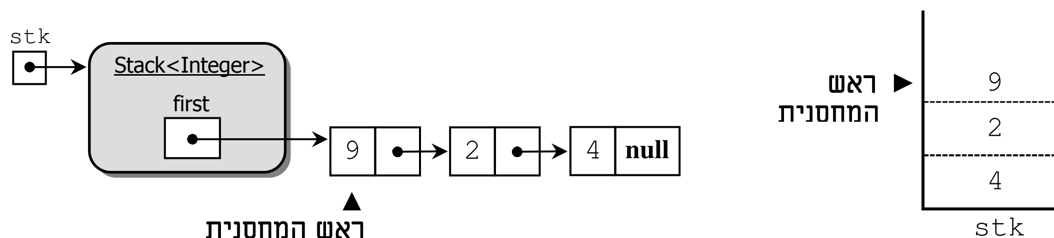
כיוון שהמחסנית דינמית, ואפשר להכניס אליה ערכים ולהוציאם ממנה, ואין מגבלה על מספר הערכים שבתוכה, מתאים מאוד לייצגה בעזרת שרשרת חוליות. השרשרת מאפשרת לשמור בכל רגע נתון בדיוק את מספר הערכים הקיימים במחסנית ואינה מבזבזת זיכרון נוסף. אנו נציג ייצוג המשתמש בשרשרת חוליות חד-כיוונית.

להלן הייצוג החדש:

```
public class Stack<T>
{
      private Node<T> first;
      ...
```

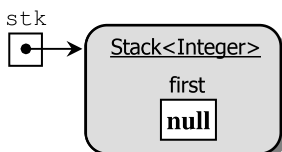
את שרשרת החוליות מחזיקה תכונה פנימית פרטית של המחלקה מחסנית. התכונה היא מטיפוס חוליה ונקראת first, וכאשר המחסנית אינה ריקה, היא תְּפנה לחוליה הראשונה בשרשרת. הפעילות על המחסנית מתבצעת רק בקצה אחד של שרשרת החוליות, שכן דחיפה ושליפה של ערכים ממחסנית נעשית רק דרך ראש המחסנית. ראש המחסנית יכול להיות תחילת השרשרת או סופה. החלטה על הקצה הרצוי תיעשה לאור שיקולי יעילות שבהם נעסוק בהמשך. בייצוג הנוכחי נקבע כי ראש המחסנית יהיה תחילת שרשרת החוליות. כלומר, התכונה first מכילה הפניה לחוליה שבראש המחסנית ומאפשרת גישה ישירה אליה.

נראה מחסנית של שלמים (מימין), ואת ייצוגה בעזרת תרשים עצמים (משמאל):



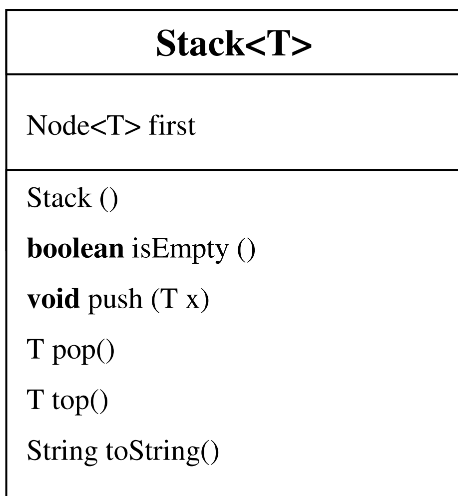
כאמור, הכנסה והוצאה של איברים מהמחסנית יתבצעו בתחילת שרשרת החוליות, שהיא ראש המחסנית.

שימו לב כי בייצוג זה השרשרת "עטופה" במחלקה Stack, המציגה למשתמש הפשטה של רעיון המחסנית. כך, בעוד ששרשרת חוליות לעולם אינה ריקה, המחסנית יכולה להיות ריקה, בדיוק כפי שהרשימה הכיתתית יכולה הייתה להיות ריקה. מצב זה של מחסנית ריקה מיוצג על ידי כך שהערך בתכונה first הוא **null**.



ה.2. מימוש הפעולות

לאחר שבחרנו לייצג את המחסנית באמצעות שרשרת חוליות, נעבור לממש את פעולות המחלקה על פי ייצוג זה. להלן תרשים UML של ממשק המחלקה מחסנית:



הפעולה הבונה

נבנה מחסנית ריקה על ידי כך שנציב ב-first את הערך **null**. ערך זה משמעותו שהמחסנית ריקה:

```
public Stack()
{
      this.first = null;
}
```

בדיקת ריקנות

נבדוק האם המחסנית ריקה באמצעות בדיקת ההפניה שבתכונה first :

```
public boolean isEmpty()
{
      return this.first == null;
}
```

הכנסה (דחיפה)

כאשר נרצה להכניס ערך למחסנית יהיה עלינו לייצר חוליה חדשה המכילה אותו, ולדחוף אותה לראש המחסנית, כלומר למקום הראשון בשרשרת החוליות. נזכיר כי בפרק הקודם ראינו כי פעולה אינה יכולה להכניס חוליה למקום הראשון בשרשרת חוליות המועברת אליה כפרמטר. מיד נראה כי עכשיו, כאשר השרשרת עטופה על ידי המחלקה מחסנית, אין בעיה להכניס חוליה למקום הראשון בשרשרת.

כדי להכניס את החוליה החדשה כחוליה ראשונה בשרשרת החוליות, מבלי לאבד את שרשרת החוליות, עלינו לקבוע את הערך next של החוליה החדשה, כך שיצביע לחוליה הראשונה בשרשרת הקיימת. זאת ניתן לעשות בעת יצירת החוליה. לאחר מכן יש לעדכן ההפניה first, כך שתצביע אל החוליה החדשה.

לפניכם הקוד המבצע את הדחיפה :

```
public void push (T x)
{
      this.first = new Node<T>(x, this.first);
}
```

? הסבירו מדוע אין בקוד זה שום בעיה בהכנסת חוליה למקום הראשון בשרשרת.

הוצאה (שליפה)

כדי לשלוף איבר מהמחסנית עלינו לנתק את החוליה הראשונה מהשרשרת, ולעדכן את first, כך שיפנה אל החוליה העוקבת (אם יש כזו). כמו בהכנסה, גם הוצאת חוליה ניתנת לביצוע ללא בעיות. לפני ניתוק החוליה אנו שומרים את ערכה במשתנה שערכו יהיה ערך ההחזרה של הפעולה. נזכיר כי לא ניתן לבצע הכנסה של חוליה או הוצאת חוליה משרשרת חוליות המועברת כפרמטר לפעולה, כאשר מדובר בחוליה הראשונה בשרשרת.

לפניכם קוד הפעולה :

```
public T pop()
{
      T x = this.first.getInfo();
      this.first = this.first.getNext();

      return x;
}
```

התכונה first היא המשתנה היחיד המפנה לחוליה הראשונה בשרשרת, ואין שום הפניה חיצונית אל חוליה כלשהי בשרשרת. בסיום הפעולה, התכונה first אינה מפנה עוד אל החוליה הראשונה המקורית, ולכן אין אליה שום הפניה. 'אספן הזבל' ידאג לשחרר את החוליה מהזיכרון.

לא ניתן לבצע את הפעולה אם המחסנית ריקה, ובכל זאת איננו בודקים מצב זה בתחילת הפעולה. אנו מסתמכים על ההנחה, כפי שהיא מופיעה בהגדרת הפעולה בממשק, ולפיה אין לזמן את הפעולה אם המחסנית ריקה. האחריות על בדיקת הריקנות היא על המשתמש במחסנית.

הערה: בחרנו לממש את המחלקה כך שגם ההכנסה וגם ההוצאה של הערכים בשרשרת החוליות ייעשו במקום הראשון בשרשרת. ניתן היה לבחור שהן ההכנסה והן ההוצאה ייעשו מהקצה האחר של שרשרת החוליות, כלומר להתייחס לחוליה האחרונה כאל ראש המחסנית.

? חשבו את היעילות של פעולות הממשק עבור הייצוג החלופי המתואר לעיל.

תיאור המחסנית

הפעולה ()toString מחזירה את סדרת הערכים השמורים במחסנית כך שהם מופיעים בתוך סוגריים מרובעים ומופרדים בעזרת פסיקים. האיבר השמאלי ביותר הוא זה הנמצא בראש המחסנית.

לסיכום, נתבונן בממשקי הפעולות ובמימושיהן. קל לראות כי ייצוג המחסנית אינו חשוף למשתמש במחלקה: כל הפעילות על החוליות היא פנימית למימוש, ופעולות הממשק אינן מאפשרות גישה לחוליות, ולכן לא ניתן לפגוע במבנה המחסנית.

ה.3. המחלקה <Stack<T (בייצוג שרשרת חוליות)

```
public class Stack<T>
{
      private Node<T> first;

      public Stack()
      {
           this.first = null;
      }

      public boolean isEmpty()
      {
            return this.first == null;
      }

      public void push(T x)
      {
            this.first = new Node<T>(x, this.first);
      }

      public T pop()
      {
            T x = this.first.getInfo();
            this.first = this.first.getNext();

            return x;
      }

      public T top()
      {
            return this.first.getInfo();
      }

      public String toString()
      {
            String str = "[";

            Node<T> pos = this.first;
            while (pos != null)
            {
                  str = str + pos.getInfo().toString();
                  if (pos.getNext() != null)
                        str = str + ",";
                  pos = pos.getNext();
            }
            str = str + "]";

            return str;
      }
}
```

ו. יעילות פעולות המחסנית

יעילות הפעולות של טיפוס נתונים נקבעת על פי מימושן, שנעשה על סמך הייצוג שנבחר עבור הטיפוס. פעולות מסוימות יכולות להיות יעילות מאוד במימוש מסוים ויעילות הרבה פחות במימוש אחר. שינוי ייצוג יכול "לגבות עלויות" בנושא יעילות זמן הריצה. כאשר דנים במחסנית, קיימת פעולה אחת שיעילותה תושפע מאוד מבחירת הייצוג, הפעולה push(…). אם מייצגים את המחסנית באמצעות מערך, ובמקרה הגרוע ביותר שבו המערך מלא לחלוטין באיברים, הפעולה push(…) צריכה ליצור מערך חדש גדול יותר מהמקורי ולהעתיק לתוכו את כל האיברים הקיימים במחסנית. כלומר יעילות הפעולה תהיה O(n) – n מייצג את מספר האיברים במחסנית. לעומת זאת, כאשר המחסנית מיוצגת באמצעות שרשרת חוליות, יעילות ההכנסה היא O(1), כיוון שתמיד קיימת בידינו הפניה בדיוק למקום שבו צריך להכניס את האיבר החדש.

שימו לב, דרך מימוש שונה על פי אותו ייצוג יכולה ליצור הבדלי יעילות. ייצוג מחסנית בעזרת שרשרת חוליות, כאשר ראש המחסנית הוא האיבר האחרון בשרשרת, ישנֶה את יעילות הפעולות push(…), pop(), top() כך שתהיה O(n) לכל אחת מהן. אם נשנה את הייצוג של המחסנית ונוסיף את התכונה last, מטיפוס חוליה, שתשמור הפניה לאיבר האחרון בשרשרת החוליות, תחזור יעילות הפעולות push(…), top() להיות O(1). אולם, מה לגבי הפעולה pop()? פעולה זו חייבת לעדכן את ערכה של התכונה last, ולכן תידרש סריקה של כל חוליות השרשרת עד לחוליה שתעמוד בראש המחסנית, כלומר יעילות הפעולה תישאר O(n).

מכאן והלאה נתעלם מאפשרות ייצוג זו, כיוון שהיא טובה פחות מהייצוג האחר שבו ראש המחסנית הוא החוליה הראשונה.

מרבית פעולות המחסנית מתבצעות ביעילות של O(1). נסקור לדוגמה את הפעולה pop() המתבצעת בכל אחד מהייצוגים ב-O(1):

בייצוג של המערך: אנו מעדכנים את האינדקס top.

בשרשרת החוליות: אנו שולפים את החוליה שבראש המחסנית, שאליה כאמור יש לנו הפניה.

יעילות הפעולה toString() תהיה O(n) בכל הייצוגים, כיוון שהיא דורשת סריקה של כל איברי המחסנית.

	ייצוג על ידי שרשרת חוליות	**ייצוג על ידי מערך**
Stack()	O(1)	O(1)
boolean isEmpty()	O(1)	O(1)
void push (T x)	**O(1)**	**O(n)**
T pop()	O(1)	O(1)
T top()	O(1)	O(1)
String toString()	O(n)	O(n)

ז. פעולות נוספות

לפעמים אנו מעוניינים לממש פעולות נוספות המתייחסות למחסנית. היכן נממש את הפעולות הללו? האם נרחיב את הממשק המוגדר או שמא נימנע מכך? כיצד נגדיר פעולה כזו מחוץ לממשק המחסנית? נבחן את הדוגמה הבאה.

ז.1. גודל המחסנית

ברצוננו לממש את הפעולה המחזירה את גודל האוסף, כלומר את מספר התאים במחסנית. עומדות לפנינו שתי אפשרויות:

אפשרות ראשונה: פעולה פנימית

נממש את הפעולה כ**פעולה פנימית** של המחלקה Stack המוצגת בסעיף ה.3., כלומר נוסיף עוד פעולה לממשק המחלקה. הפעולה הפנימית פועלת על המחסנית הנוכחית, ולכן מותר לנו להשתמש בייצוג כדי לספור את מספר החוליות בשרשרת:

```
public int size()
{
      int counter = 0;
      Node<T> temp = this.first;

      while (temp != null)
      {
            counter++;
            temp = temp.getNext();
      }

      return counter;
}
```

דרך נוספת למימוש הפעולה יכולה להיות שינוי הייצוג של המחסנית. אם היינו מוסיפים לייצוג המחסנית את התכונה size, שהייתה שומרת את המספר המייצג את כמות האיברים שבמחסנית בכל רגע נתון, היינו יכולים לבצע פעולה קצרה ותכליתית בשם ()getSize. יש לשים לב שבעוד שינוי הייצוג היה מוריד את מחיר היעילות של מימוש הפעולה ()getSize, היינו צריכים להוסיף עדכונים של התכונה size בפעולות (…)push ו-()pop. כמו כן, לא תמיד נתונה לנו היכולת לשנות את הייצוג של מחלקה, ולכן יש לבחון בכל מצב האם אפשרות זו מתאימה לבעיה הנתונה.

אפשרות שנייה: פעולה חיצונית

הוספת הפעולה ()size מרחיבה את אוסף פעולות הממשק של מחסנית, מעבר למינימום המגדיר תפקוד תקין של מחסנית. אך האומנם נהיה מעוניינים להרחיב את הממשק של טיפוס נתונים עבור כל בעיה פרטית שבה נתקל? עומדת בפנינו אפשרות נוספת.

נממש את הפעולה המחזירה את גודל המחסנית כ**פעולה חיצונית** למחלקה Stack. הגדרת פעולה חיצונית פירושה שהפעולה מוגדרת וממומשת מחוץ למחלקה מחסנית. הפעולה פועלת על פרמטר שהוא עצם מטיפוס מחסנית. במקרה זה לא ניתן להשתמש בייצוג המחלקה אלא בפעולות

הממשק בלבד. לכן גם לא נאמר שאנו סופרים חוליות, כיוון שכלל איננו יודעים כיצד מיוצג האוסף השמור במחסנית, אלא נאמר שאנו סופרים את הערכים השמורים במחסנית. כיוון שהפעולה היא חיצונית, ובהתאם להחלטתנו ביחידת לימוד זו, היא פועלת על מחסנית קונקרטית, למשל על Stack<Integer>.

בתכנון פעולה זו עלינו להביא בחשבון שלשם ספירת הערכים שבמחסנית, עלינו לשלוף אותם אחד-אחד מן המחסנית. עם זאת, אחרי ביצוע הפעולה, סביר ביותר שנצטרך להמשיך ולהתייחס אל המחסנית המקורית בהמשך התוכנית. עלינו לוודא שמצב המחסנית בסיום הפעולה יהיה כמו בראשיתה. דיון זה מוביל למימוש שלפניכם:

```
public static int size(Stack<Integer> s)
{
     int counter = 0;
     Stack<Integer> temp = new Stack<Integer>();

     while (!s.isEmpty())
     {
          temp.push(s.pop()); // שמירת האיברים במחסנית העזר
          counter++;
     }

     while (!temp.isEmpty()) // החזרת האיברים למחסנית המקורית
          s.push(temp.pop());

     return counter;
}
```

במימוש זה השתמשנו במחסנית עזר כדי לשמור על תוכן ומבנה המחסנית המקורית. כל מימוש של פעולה המחייב שינוי הסדר של איברי המחסנית, או כניסה לעומקה, מחייב לשמור את איברי המחסנית המקורית כך שניתן יהיה להחזירם למחסנית תוך שמירת הסדר המקורי שלהם. לצורך כך יש להשתמש במבנה עזר שיאפשר את שחזור המחסנית המקורית קודם לתום הפעולה. מחסנית היא מבנה נוח ומתאים למטרה זו.

ז.2. סכום הערכים במחסנית

נדון בפעולה נוספת על מחסנית, המחזירה את סכום הערכים השמורים במחסנית (למשל, אם הערכים במחסנית הם 1, 6, 18 אזי הפעולה תחזיר 25). פעולה זו אינה מתאימה למחסנית בעלת ערכים שלא ניתנים לסכימה. למשל, לא ניתן לסכום ערכים שהם צבים או קוביות. לכן, פעולה זו אינה יכולה להיות ממומשת כפעולה פנימית של המחלקה מחסנית, שכן היא לא מתאימה לכל טיפוס של ערכים.

כמו הפעולה הקודמת, תוגדר פעולה זו כפעולה חיצונית (פעולה סטטית), למחלקה Stack. גם כאן נזדקק למחסנית עזר כדי לשמר את המחסנית המקורית:

```
public static int sumItems(Stack<Integer> s)
{
     int x, sum = 0;
     Stack<Integer> temp = new Stack<Integer>();

     while (!s.isEmpty())
     {
          x = s.pop();
          temp.push(x);
          sum = sum + x;
     }
     while (!temp.isEmpty())
          s.push(temp.pop());

     return sum;
}
```

נסכם: ההחלטה באיזה פתרון נבחר לצורך מימוש פעולה נוספת על אוסף נתונים, האם נגדיר בממשק פעולה פנימית נוספת או נגדיר פעולה חיצונית, תלויה בצרכים הספציפיים של הבעיה ובהתאם להנחיות שנקבל. בכל מקרה, ברור שפעולה פנימית יכולה להיבחן כאפשרות רק במקרים שבהם לא נדרשים שום טיפול או התייחסות ייחודית לערכים השמורים במחסנית.

ח. המחסנית – טיפוס נתונים מופשט

בפרק מחלקות הגדרנו "טיפוס נתונים מופשט" (טנ"מ) כטיפוס שהממשק שלו אינו חושף את דרך הייצוג שלו. מכאן נובע שלא ניתן לגשת ישירות לתכונות העצמים של הטיפוס ולשנות את מצבם, אלא רק דרך הפעולות המוגדרות על הטיפוס. כמו כן, אפשר להסיק מכך שניתן להחליף את דרך הייצוג של הטיפוס מבלי שמשהו בפעולות הממשק ישתנה. כך אפשר להעלים מהמשתמש בַּטיפוס את כל נושא הייצוג והמימוש של הטיפוס, ולשמור בהקפדה את העיקרון של הסתרת מידע שהוא עקרון בסיסי בתכנות מונחה עצמים.

הגדרה זו של טיפוס נתונים מופשט הספיקה והתאימה לכל מחלקה שהגדרנו למעשה ביחידת לימוד זו עד לכאן. לפי הגדרה זו, היו הדלי, הדרכון והנקודה טיפוסי נתונים מופשטים, שהפעילות עליהם נעשתה רק בעזרת פעולות הממשק. הפעולות הסתירו את אופן הייצוג והמימוש של הטיפוסים. ניתן היה לשנות את הייצוג והמימוש בלי שהמשתמש היה צריך לשנות משהו בתוכניות שכבר התבססו על עצמים מטיפוסים אלה.

כשעברנו לטפל באוספים דינמיים הרחבנו את ההגדרה של טיפוס הנתונים המופשט בדרישה נוספת. כיוון שההגדרה של אוסף נתונים כוללת את הרעיון של קיום נוהל מחייב של אופן הגישה אל האיברים השמורים באוסף (פרוטוקול), חייב טיפוס נתונים מופשט המגדיר אוסף להיות מוגן מפני כל שינוי במבנה שלו העלול לפגוע בקיום הנוהל. על הטיפוס מוטלת הדרישה לוודא שכל הפעולות שנעשות עליו לא יפגעו במבנהו באופן שיוביל להפרת נוהל הגישה. בַּמחסנית, פירוש הדבר, שכל הפעולות שהמשתמש יכול לבצע יבטיחו שאיברים יתוספו למחסנית רק בראשה

ויוצאו ממנה בסדר הפוך לסדר הכנסתם. נוסף על כך, אי-אפשר יהיה לפנות לאיברים השמורים בעומק המחסנית.

המחסנית, כפי שהוגדרה בפרק זה, מקיימת את כל הנדרש מטיפוס נתונים מופשט: ממשק המחלקה Stack<T> מגדיר פעולות, אך אינו חושף את דרך הייצוג והמימוש של המחלקה. ראינו כי ניתן לממש את המחסנית לפחות בשני ייצוגים שונים: במערך ובשרשרת חוליות. שינוי הייצוג אינו משנה את הממשק של המחסנית. לכן אין צורך לבצע שינוי כלשהו בתוכנית המשתמשת במחסנית, גם אם מחליפים מימוש אחד במשנהו.

כיוון שהפעולות האפשריות על מחסנית הן רק כאלה המאפשרות הכנסה לראש המחסנית והוצאה ממנו, הרי גם הדרישה החדשה לגבי שמירת המבנה ונוהל הגישה של האוסף מתקיימת. מכל אלה נובע שהמחסנית אכן עונה על ההגדרה של טיפוס נתונים מופשט:

1. קיימת הפרדה מלאה בין הממשק למימוש, ולכן המימוש והייצוג מוסתרים לחלוטין מהמשתמש.
2. בשום דרך לא ניתן לקלקל את המבנה של המחסנית.

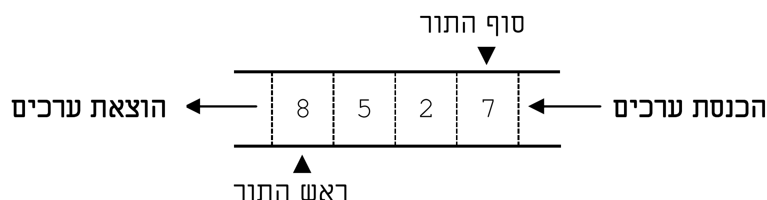
המחלקה מחסנית מגדירה טיפוס אוסף כללי עם פרוטוקול LIFO להכנסה של ערכים ולהוצאתם. המחסנית היא הטיפוס הראשון בארגז הכלים שלנו, האמור להכיל מחלקות המאפשרות להחזיק אוסף של נתונים ולטפל בהם בעזרת נוהל קבוע. המחסנית תשמש לפתרון בעיות יישומיות ספציפיות המקיימות את פרוטוקול LIFO.

את הפרק נסיים בהצגת טיפוס אוסף כללי נוסף המקיים פרוטוקול שונה. סוג אוסף זה יהיה הטיפוס הבא בארגז הכלים שלנו.

ט. התור

תור (**Queue**) הוא סוג של אוסף. הערכים בתור מאורגנים כסדרה, והפרוטוקול עבור פעולות ההכנסה וההוצאה הוא זה: הוצאת ערכים בתור מותרת רק מצד אחד, הקרוי **ראש התור**, והכנסת ערכים מותרת רק לצד האחר, הקרוי **סוף התור** (או 'זנב התור').

גישה כזו לטיפול בערכים נקראת **FIFO** – ראשי התיבות של המילים: **F**irst **I**n **F**irst **O**ut: האיבר הראשון שנכנס לתור הוא הראשון לצאת ממנו.



ט.1. ממשק המחלקה התור

הפעולות שיש צורך להגדיר בממשק התור הן פעולות הכנסה והוצאה התומכות בפרוטוקול FIFO. כפי שעשינו במחסנית, נפריד בין פעולת הוצאה המחזירה את הערך שבראש התור לבין פעולה המשאירה את האיבר בתור ורק מאפשרת לברר את ערכו. כמו במחסנית, יש צורך להגדיר בממשק גם פעולה הבודקת ריקנות, שכן פעולת ההוצאה תלויה בבדיקה זו. כיוון שהתור הוא אוסף כללי ופעולותיו אינן משתמשות בתכונות או בפעולות ייחודיות לערכים השמורים בו, נגדיר אותו כמחלקה גנרית.

ממשק המחלקה תור Queue<T>

המחלקה מגדירה טיפוס אוסף עם פרוטוקול FIFO להכנסה והוצאה של ערכים.

הפעולה בונה תור ריק	Queue()
הפעולה מחזירה 'אמת' אם התור הנוכחי ריק, ו'שקר' אחרת	**boolean** isEmpty()
הפעולה מכניסה את הערך x לסוף התור הנוכחי	**void** insert (T x)
הפעולה מוציאה את הערך שבראש התור הנוכחי ומחזירה אותו. **הנחה**: התור הנוכחי אינו ריק	T remove()
הפעולה מחזירה את ערכו של האיבר שבראש התור מבלי להוציאו. **הנחה**: התור הנוכחי אינו ריק	T head()
הפעולה מחזירה מחרוזת המתארת את התור כסדרה של ערכים, במבנה הזה (x_1 הוא האיבר שבראש התור): $[x_1, x_2, \ldots, x_n]$	String toString()

ט.2. ייצוג המחלקה תור

כמו במחסנית, גם תור ניתן לייצוג בעזרת מערך. אם רוצים להכניס ערך נוסף והמערך מלא, יש להקצות מערך חדש, גדול יותר, להעתיק אליו את המערך הנוכחי, ורק אז לבצע את ההכנסה. כיוון שכבר דנו בגישה זו עבור מחסנית, והתור הוא סדרה שאינה מוגבלת בגודלה, נעסוק כאן בייצוג בעזרת שרשרת חוליות.

כיוון שההכנסה וההוצאה מהתור מתבצעות משני קצוות שונים, נשמור שתי הפניות, הפניה אחת לראש התור והפניה נוספת לסופו:

```
public class Queue<T>
{
            private Node<T> first;
            private Node<T> last;
            ...
```

? א. היכן בשרשרת, כדאי לקבוע את first והיכן את last? הסבירו את בחירתכם.

ב. כתבו את המחלקה תור כך שיעילות כל פעולות הממשק, למעט toString(), תהיה O(1).

ג. נבחן שינוי אפשרי של ייצוג התור: נסתפק בהפניה אחת לראש התור. לשם ביצוע הפעולה המתבצעת בקצה התור האחר, "נרוץ" עם הפניה נוספת עד סוף התור. ייצוג זה יגרום לפחות לאחת מהפעולות המוגדרות על התור להתבצע ביעילות O(n) במקום ב-O(1), ולכן להפוך ללא יעילה. זהו את הפעולה.

ט.3. יעילות פעולות התור

בהינתן הייצוג המופיע בסעיף הקודם, נסכם את יעילות פעולות הממשק של תור (n מייצג את מספר האיברים בתור).

Queue()	O(1)
boolean isEmpty()	O(1)
void insert (T x)	O(1)
T remove()	O(1)
T head()	O(1)
String toString()	O(n)

ט.4. התור – טיפוס נתונים מופשט

העבודה עם עצמים מטיפוס תור נעשית אך ורק דרך ממשק הפעולות, שאינו חושף את דרך הייצוג של התור ואינו מאפשר לגשת ישירות לתכונות התור ולשנותן. לפיכך ניתן לשנות את דרך הייצוג והמימוש של התור מבלי שתוכניות המשתמשות בפעולות התור יהיו מודעות לשינוי או ידרשו לבצע שינויים בהתאמה. עקרון הסתרת המידע מתקיים ונשמר.

יתרה מזאת, התור שומר על מבנהו, ופעולות התור מבטיחות את קיומו התקין של פרוטוקול הגישה FIFO, המחייב שהכנסה לתור וההוצאה ממנו יתבצעו בשני קצוות שונים של התור. כלומר, בדומה למחסנית, גם תור הוא טיפוס נתונים מופשט.

התור הוא אם כן טיפוס אוסף נוסף המצטרף לארגז הכלים שלנו. הוא ישמש אותנו בהמשך בדוגמאות השונות, וביישומים שבהם יש צורך לטפל באוספי נתונים ספציפיים המתנהגים על פי פרוטוקול FIFO.

י. סיכום

- המחסנית היא טיפוס אוסף כללי ואינה מוגבלת בגודלה. ניתן לחשוב על מחסנית כסדרה שבה פעולות הדחיפה והשליפה של ערכים מתבצעות רק דרך קצה אחד של הסדרה, הנקרא ראש המחסנית. הכנסת ערך למחסנית מוסיפה ערך חדש בראש המחסנית. הוצאת ערך מהמחסנית מסירה את הערך המצוי בראש המחסנית וחושפת את הערך הבא בסדרה.
- הפרוטוקול המגדיר את דרך הגישה לערכים במחסנית נקרא **LIFO** – **L**ast **I**n **F**irst **O**ut.
- המחסנית משתלבת באופן טבעי בכתיבת יישומים שבהם יש צורך לאחזר נתונים באופן הפוך לסדר קליטתם.
- **תור** הוא טיפוס אוסף כללי שאינו מוגבל בגודלו. ניתן לחשוב על תור כסדרה שבה פעולות ההכנסה וההוצאה מתבצעות דרך שני קצוות שונים של הסדרה, ראש התור וזנבו. הכנסת ערך נעשית אל זנב התור, והוצאת ערך נעשית מראש התור.
- הפרוטוקול המגדיר את דרך הגישה לערכים בתור נקרא **FIFO** – **F**irst **I**n **F**irst **O**ut.
- ניתן לייצג מחסנית ותור באמצעות מערך או באמצעות שרשרת חוליות. הייצוג באמצעות מערך דורש כתיבת פעולת הכנסה מורכבת, הכוללת יצירת מערך חדש והעתקת איברים אליו אם המערך הקיים מלא. בעיה זו אינה מתעוררת בייצוג בעזרת שרשרת חוליות.
- מחסנית ותור הם טיפוסי נתונים גנריים. טיפוס המחלקה ייקבע רק בעת השימוש בה, למשל בעת יצירת עצם. ממחסנית ותור גנריים אפשר ליצור מחסניות ותורים מטיפוסים שונים ללא צורך בשינוי המחלקות.
- פעולות חיצוניות על אוספים שיוגדרו ביחידת לימוד זו יטפלו רק בטיפוסים קונקרטיים ולא בטיפוסים גנריים.
- ניתן להרחיב את אוסף הפעולות האפשריות על מחסנית או תור, אך יש להחליט אם להגדיר פעולות אלה כפעולות פנימיות המרחיבות את הממשק המקורי או להגדיר פעולות חיצוניות המקבלות את המחסנית או התור כפרמטר ועובדות עליהם. במחסנית ובתור אנו מעדיפים שלא להרחיב את ממשק הטיפוס המקורי ולהגדיר פעולות נוספות כפעולות חיצוניות.
- טיפוס אוסף הוא טיפוס מופשט אם הוא מקיים את הדרישות האלה:
  1. הממשק שלו אינו חושף את הייצוג שנבחר.
  2. ניתן לממש את הטיפוס בכמה ייצוגים שונים.
  3. הממשק שומר על הפרוטוקול הכלול בתיאור הטיפוס – ביצוע פעולות הממשק אינו יכול לקלקל את מבנה האוסף ואינו יכול לפגוע בקיום הפרוטוקול.
- המחסנית והתור, כפי שהוצגו בפרק זה, הם טיפוסי נתונים מופשטים.
- המחסנית והתור הם טיפוסי האוספים הכלליים הראשונים המוצגים ביחידה זו. הם ישמשו אותנו בהמשך לפתרון יישומים מתקדמים.

מושגים

מחסנית	Stack
מחסנית זמן ריצה	run time stack
נכנס אחרון יוצא ראשון	Last In First Out
נכנס ראשון יוצא ראשון	First In First Out
תור	Queue

תרגילים

שאלה 1

תארו את תכולת המחסניות s1 ו-s2 לאחר כל סדרת פעולות :

א.

```
Stack<Character> s1 = new Stack<Character>();
Stack<Character> s2 = new Stack<Character>();
s1.push('a');
s1.push('b');
s1.push('c');
s1.push('d');
char ch = s1.pop();
s2.push(ch);
ch = s2.pop();
s1.push('e');
```

ב.

```
Stack<Integer> s1 = new Stack<Integer>();
Stack<Integer> s2 = new Stack<Integer>();
s1.push(1);
s2.push(2);
s1.push(3);
s2.push(4);
s1 = s2;
s1.push(5);
s2.push(6);
```

שאלה 2

תארו את תכולת התורים q1 ו-q2 לאחר כל סדרת פעולות :

א.

```
Queue<Integer> q1 = new Queue<Integer>();
Queue<Integer> q2 = new Queue<Integer>();
q1.insert(32);
q1.insert(-6);
q2.insert(q1.head());
q1.insert(q1.remove()+q1.remove());
if (q1.isEmpty())
      q1.insert(9);
else
      q2.insert(17);
```

ב.

```
Queue<String> q1 = new Queue<String>();
Queue<String> q2 = new Queue<String>();
q1.insert("Hello");
q1.insert("World");
q2.insert(q1.remove());
q2.insert(q1.head());
q1.remove();
q1.insert("Hello" + q2.head());
```

שאלה 3

נתונות שתי מחסניות st1 ו-st2, של מספרים שלמים. המחסנית st2 ריקה, ואילו st1 מכילה מספר לא ידוע של ערכים. נתון קטע הקוד הזה:

```
int x, y;
while (!st1.isEmpty())
{
      x = st1.pop();
      if (!st1.isEmpty())
      {
            y = st1.pop();
            st2.push(y);
      }
      else
            st2.push(x);

      st2.push(x);
}
System.out.println("st2=" + st2);
```

בהנחה שהמחסנית st1 מכילה את סדרת הערכים: [8, 2, 1, 7, 3] (הערך 8 נמצא בראש המחסנית), כתבו מה יהיה הפלט שיתקבל בעקבות הרצת קטע הקוד.

שאלה 4

הפעולה שלפניכם משתמשת במחסנית של מספרים שלמים לפתרון בעיה מסוימת:

```
public static int mystery(int num)
{
      int x = 0, y = 1;
      Stack<Integer> s = new Stack<Integer>();

      while (num != 0)
      {
            s.push(num % 10);
            num = num / 10;
      }
      while (!s.isEmpty())
      {
            x = x + y * s.pop();
            y = y * 10;
      }

      return x;
}
```

א. מהו המספר שיוחזר עבור הזימון mystery(2047)?

ב. כתבו את טענת היציאה של הפעולה mystery(...).

ג. נתחו את יעילות הפעולה mystery(...).

שאלה 5

הפעולה שלפניכם מקבלת שני מספרים שלמים וחיוביים, m ו-n, m>n. הפעולה נעזרת בתור של מספרים שלמים:

```
public static void printSomething(int m, int n)
{
      Queue<Integer> q = new Queue<Integer>();

      for (int i = 1; i <= n; i++)
            q.insert(i);

      while (!q.isEmpty())
      {
            for (int i = 1; i <= m - 1; i++)
                  q.insert(q.remove());
            System.out.print(q.remove() + " ");
      }
}
```

א. עקבו אחר ביצוע הפעולה עבור הזימון printSomething (5, 9) וכתבו את הפלט המתקבל.

ב. כתבו את טענת היציאה של הפעולה printSomething(…).

ג. נתחו את יעילות הפעולה printSomething(…).

שאלה 6

לפניכם הפעולה:

```
public static boolean secret(Stack<Integer> s1, Stack<Integer> s2)
{
      while (!s1.isEmpty() && !s2.isEmpty())
      {
            if (s1.pop() != s2.pop())
                  return false;
      }
      if (!s1.isEmpty() || !s2.isEmpty())
            return false;

      return true;
}
```

א. תנו דוגמה לשתי מחסניות s1 ו-s2, שעבורן הזימון secret (s1, s2) יחזיר **false**, ודוגמה נוספת לזימון שיחזיר **true**.

ב. כתבו את טענת היציאה של הפעולה secret(…).

ג. נתחו את יעילות הפעולה secret(…).

שאלה 7

נתונה תוכנית המשתמשת במחסנית של מחרוזות:

```
public class Test
{
      public static void main(String[] args)
      {
            Stack<String> st1 = new Stack<String>();

            st1.push("i");
            st1.push("world");
            st1.push("here");
            st1.push("hello");
            st1.push("am");
            doSomething(st1);
            while (!st1.isEmpty())
                  System.out.print(st1.pop() + " ");
      }

      public static void doSomething(Stack<String> st2)
      {
            Stack<String> st3 = new Stack<String>();

            while (!st2.isEmpty())
            {
                  String str = st2.pop();
                  if (str.length() > 4)
                        st3.push(str);
            }
            while (!st3.isEmpty())
                  st2.push(st3.pop());
      }

}
```

עקבו אחרי ריצת התוכנית, וכתבו את הפלט.

שאלה 8

לפניכם פעולה רקורסיבית:

```
public static void mystery(Queue<Integer> q)
{
      int x;

      if (!q.isEmpty())
      {
            x = q.remove();
            mystery(q);
            q.insert(x);
      }
}
```

כתבו את טענת היציאה של הפעולה.

שאלה 9

הפעולה שלפניכם מקבלת תור ומחזירה את גודלו – מספר האיברים בתור. בסיום הפעולה התור נשאר כפי שהתקבל:

```
public static int size(Queue<Integer> q)
```

א. ממשו את הפעולה תוך שימוש בתור עזר.

ב. נתחו את יעילות הפעולה size(…).

ג. האם ניתן לממש את הפעולה size(…) ללא שימוש בתור עזר או בכל טיפוס אוסף אחר? אם כן, ממשו את הפעולה בדרך זו. אם לא, הסבירו מדוע.

שאלה 10

בפרק הוצגה הפעולה החיצונית:

```
public static int size(Stack<Integer> s)
```

המקבלת מחסנית של מספרים ומחזירה את גודל המחסנית, כלומר את מספר האיברים במחסנית. בסיום הפעולה המחסנית נשארה כפי שהייתה במקור, וזאת משום שהשתמשנו במחסנית עזר.

ממשו את הפעולה מחדש, אך הפעם ללא שימוש במחסנית עזר או בכל טיפוס אוסף אחר.

רמז: השתמשו ברקורסיה.

שאלה 11

נניח כי הפעולה top() אינה קיימת בממשק המחסנית. כתבו פעולה חיצונית המבצעת את הפעולה הזו על מחסנית של מחרוזות (Stack<String>). חשבו מהו הפרמטר שתקבל הפעולה ומה יהיה ערך ההחזרה שלה.

שאלה 12

לעתים קרובות אנו מעוניינים לשכפל תור, כלומר ליצור העתק מדויק של תור קיים. כתבו פעולה חיצונית בשם cloneQueue(…), המקבלת תור של תווים ומחזירה תור חדש שהוא העתק של התור שהתקבל.

שאלה 13

ממשו פעולה בשם getStringsLength(…) המקבלת מחסנית של מחרוזות ומחזירה מחסנית שבה מופיעים אורכי המחרוזות לפי הסדר שלהן במחסנית המקורית.

שימו לב: המחסנית המתקבלת לא תשתנה בסיום הפעולה.

דוגמה: אם הפעולה מקבלת את המחסנית:

"Maradona"
"Pele"
"Messy"
"Ronaldinho"

הפעולה תחזיר מחסנית ובה הערכים האלה:

8
4
5
10

שאלה 14

בשאלה זו עליכם לממש שתי פעולות כפעולות פנימיות של המחלקה מחסנית Stack<T>, כאשר היא מיוצגת בעזרת שרשרת חוליות.

א. ממשו את הפעולה הפנימית הזו:

```
public T getItemAt(int k)
```

טענת כניסה: הפעולה מקבלת מספר שלם חיובי k.

טענת יציאה: הפעולה מחזירה את האיבר שנמצא בעומק k במחסנית הנוכחית. לאחר ביצוע הפעולה, מצב המחסנית יהיה זהה למצבה לפני ביצוע הפעולה. בעומק k=1 נמצא האיבר שבראש המחסנית. אם במחסנית פחות מ-k ערכים יוחזר **null**.

לדוגמה, אם המחסנית st נראית כך:

&
@
#
$
%

לאחר הזימון st.getItemAt (4) יוחזר התו '$'.

לאחר הזימון st.getItemAt (1) יוחזר התו '&'.

ב. ממשו את הפעולה הפנימית הזו:

```
public T removeItemAt(int k)
```

טענת כניסה: הפעולה מקבלת מספר שלם חיובי k.

טענת יציאה: הפעולה מוציאה את האיבר שנמצא בעומק k במחסנית הנוכחית ומחזירה אותו. לאחר ביצוע הפעולה, במחסנית יישארו כל איבריה, למעט האיבר שהוצא. בעומק k=1 נמצא האיבר שבראש המחסנית. אם במחסנית פחות מ-k ערכים יוחזר **null**.

לדוגמה, אם המחסנית st נראית כך:

&
@
#
$
%

לאחר הזימון st.removeItemAt (4) יוחזר התו '$', והמחסנית תראה כך:

&
@
#
%

שאלה 15

ממשו את הפעולה שלפניכם כפעולה פנימית של המחלקה תור Queue<T> על פי הייצוג המופיע בפרק בסעיף ט.2.:

```
public void reverse()
```

הפעולה הופכת את סדר איברי התור.

יש לממש את הפעולה בסדר גודל O(n) לכל היותר. n הוא מספר האיברים בתור.

שאלה 16

ממשו את הפעולה:

```
public static int getTopsSum(Stack<Integer>[] stacks)
```

הפעולה מקבלת מערך מחסניות של מספרים שלמים, ומחזירה את סכום האיברים שנמצאים בראשי המחסניות. שימו לב: ייתכן שמחסנית תהיה ריקה, ולכן אין לה איבר בראש המחסנית.

שאלה 17

נזכיר את העקרונות העומדים מאחורי פעילותה של מחסנית זמן ריצה:

1. בתחילת התוכנית "מחסנית זמן הריצה" ריקה.
2. בכל פעם שמתבצעת קריאה (זימון) לפעולה כלשהי, נדחפת לראש "מחסנית זמן הריצה" הכתובת של הפקודה העוקבת לפעולה שזומנה.
3. בכל פעם שמסתיים ביצוע של פעולה שזומנה, נשלפת מראש "מחסנית זמן הריצה" הכתובת של הפקודה העוקבת לפעולה זו, והתוכנית ממשיכה ממקום זה.

עקבו אחר פעילותה של מחסנית זמן הריצה בזמן ביצוע הקוד שלפניכם. לשם פשטות המעקב, נתייחס לכתובת של פקודה כאל מספר השורה שבה היא מופיעה. שרטטו ציור המתאר את המעקב.

```
1     public class RunTimeStack
2     {
3          public static void main(String[] args)
4          {
5              int n = 4;

6              func1(n);
7              func3(n);
8              func2(n);
9              func3(n);
10             System.out.println("end of program");
11         }

12         public static void func1(int n)
13         {
14             n = n + 1;
15         }

16         public static void func2(int n)
17         {
```

```
18              func1(n);
19              n = n + 2;
20          }

21          public static void func3(int n)
22          {
23              System.out.println(n);
24          }
25      }
```

שאלה 18

בפרק הוצגה דוגמה לשימוש במחסנית עבור בדיקת תקינות סוגריים של ביטוי חשבוני.

א. ממשו את הפעולה:

```
public static boolean isBracketBalanced(String str)
```

הפעולה מקבלת מחרוזת המייצגת ביטוי חשבוני ומחזירה **true** אם הביטוי תקין מבחינת סוגריים, ו-**false** אחרת.

ב. זמנו את הפעולה עבור כל אחד מהביטויים החשבוניים האלה, וודאו שהפעולה אכן מחזירה את התשובה הנכונה.

- {(a + b) + c } * [2 * {5 - a}] תקין
- q - (r - (s + 2)) * q) לא תקין
- [a +]] * [b - {a * 3}] + (a -(i) לא תקין
- {1 + (a *4)} - ((2 + (3 -[5 +b])) / 2) תקין

שאלה 19

כתבו תוכנית שתקלוט סדרת תווים, תו אחר תו, ותסתיים בלחיצה על מקש ה-Enter (היזכרו בקליטת התווים המתבצעת בקוד בדוגמה שבסעיף ד.2.). התוכנית תדפיס את התווים בסדר שבו הם נקלטו, עד שיופיע התו '@'. תו זה לא יודפס, אבל יגרום להיפוך סדר ההדפסה של התווים בינו ובין ה-'@' הבא. הניחו שסדרת התווים שנקלטה מכילה מספר זוגי של '@'.

לדוגמה, עבור כל אחד מהקלטים האלה: @ven@@im re@nd

ne@m rev@ind

יתקבל הפלט: never mind

ועבור הקלט: se@tp@emb@re@ so@gn@

יתקבל הפלט: september song

שאלה 20

כתבו תוכנית שתקלוט סדרת תווים, תו אחר תו, שתסתיים בלחיצת מקש ה-Enter (היזכרו בקליטת התווים המתבצעת בקוד בדוגמה שבסעיף ד.2.). סדרת התווים שתיקלט מייצגת משפט המורכב ממילים המופרדות על ידי רווח אחד בדיוק. התוכנית תדפיס את המשפט כך שכל מילה בו תודפס במהופך. יש לשמור על סדר המילים.

לדוגמה, עבור המשפט: yM eman si ehsoM

התוכנית תדפיס: My name is Moshe

שאלה 21

כתבו תוכנית שתקלוט סדרת תווים, תו אחר תו, שתסתיים בלחיצת מקש ה-Enter (היזכרו בקליטת התווים המתבצעת בקוד בדוגמה שבסעיף ד.2.). התוכנית תבדוק האם סדרת התווים שנקלטה היא מהצורה xZyZx, לפי ההגדרות האלה:

- x היא סדרת תווים לא ריקה באורך לא ידוע, המורכבת מהתווים: a, b ו-c.
- y היא סדרת תווים המכילה אותם התווים שב-x, אך בסדר הפוך (y הוא היפוך של x).
- Z הוא התו Z.

לדוגמה, המחרוזות האלה הן מהצורה xZyZx:

aZaZa

aacbaZabcaaZaacba

ואילו המחרוזת הזו אינה מהצורה xZyZx:

aZbZa

רמז: אפשר להשתמש ביותר ממחסנית אחת.

פרק 8 דף עבודה מס׳ 1
מיון בסיס

מבוא

בדף עבודה זה נכיר שיטת מיון המכונה **מיון בסיס (radix sort)**. מיון זה נעזר בטיפוס האוסף תור כדי לבצע את המיון. עקרון פעולתו של המיון מבוסס על ערכן של הספרות ועל מיקומן במספרים שאותם רוצים למיין.

המטרה בדף עבודה זה היא לממש פעולה שתקבל מערך של מספרים שלמים וחיוביים ותמיין אותו בסדר עולה על פי שיטת מיון בסיס המוצגת להלן.

שיטת המיון

נעבור על המספרים במערך ונכניס אותם לעשרה תורים נפרדים (הממוספרים מ-0 ועד 9) על פי ערכי הספרות שלהן מהמשמעותית פחות (אחדות) למשמעותית ביותר. נכניס כל מספר מהמערך באחד מעשרת התורים, לפי ערך הספרה שבה מטפלים. לאחר מכן נחזיר את המספרים למערך המקורי. נתחיל בתור המספרים בעלי הספרה 0 ונסיים בתור המספרים בעלי הספרה 9. מכל תור נוציא את המספרים כך שהסדר ביניהם יישמר. אם מבצעים פעולות אלה עבור כל ספרה, מהמשמעותית פחות למשמעותית ביותר, יתקבל מערך ממוין. כלומר, סיום המיון תלוי בערך בעל מספר הספרות הגדול ביותר שקיים במערך.
הערה: אם המספרים במערך אינם בעלי מספר ספרות זהה, יש להתייחס אליהם כאילו יש להם אפסים מובילים.

לדוגמה, כאשר נתון המערך:

170	45	75	90	2	24	802	66

אפשר להסתכל עליו גם כך:

170	045	075	090	002	024	802	066

לצורך מיון המערך המתואר, בשיטת מיון בסיס, נצטרך לסרוק את כל המערך 3 פעמים – כיוון שהערכים הגבוהים ביותר במערך הם בעלי 3 ספרות לכל היותר. בכל סריקה יש לבצע את הפעולות שתוארו לעיל.

לצורך הפתרון יש להגדיר מערך נוסף של עשרה תורים בשם queues.

<u>סריקה 1</u>: נסדר בתורים את כל המספרים במערך על פי **ספרת האחדות**, ונקבל:

```
queues[0]:  170 , 090
queues[1]:
queues[2]:  002 , 802
queues[3]:
queues[4]:  024
queues[5]:  045 , 075
queues[6]:  066
queues[7]:
queues[8]:
queues[9]:
```

כעת נוציא את כל המספרים מהתורים, על פי הסדר. נכניס אותם חזרה למערך ונקבל:

170	090	002	802	024	045	075	066

<u>סריקה 2</u>: נסדר בתורים את כל המספרים מהמערך שהתקבל, הפעם על פי **ספרת העשרות**:

```
queues[0]:  002 , 802
queues[1]:
queues[2]:  024
queues[3]:
queues[4]:  045
queues[5]:
queues[6]:  066
queues[7]:  170 , 075
queues[8]:
queues[9]:  090
```

נוציא את כל המספרים מהתורים, על פי הסדר, ונכניס אותם חזרה למערך. נקבל:

002	802	024	045	066	170	075	090

<u>סריקה 3</u>: נסדר בתורים את כל המספרים מהמערך שהתקבל, הפעם על פי **ספרת המאות**:

```
queues[0]:  002 , 024 , 045 , 066 , 075 , 090
queues[1]:  170
queues[2]:
queues[3]:
queues[4]:
queues[5]:
queues[6]:
queues[7]:
queues[8]:  802
queues[9]:
```

נוציא את כל המספרים מהתורים, על פי הסדר. נכניס אותם חזרה למערך ונקבל:

2	24	45	66	75	90	170	802

סיימנו. כפי שניתן לראות, קיבלנו מערך ממוין.

מה עליכם לעשות?

א. כתבו תוכנית שתיצור מערך של מספרים שלמים בגודל שייקלט על ידי המשתמש, ותאתחל אותו במספרים חיוביים אקראיים. התוכנית תמיין את המערך בעזרת הפעולה:

```
public static void radixSort(int[] arr)
```

הפעולה מקבלת מערך של מספרים שלמים וחיוביים וממיינת אותו בסדר עולה, על פי האלגוריתם למיון בסיס שהוסבר לעיל. מאחר שבמערך יכולים להיות מספרים שלמים בין 0 ל-Integer.MAX_VALUE, מספר הספרות הגדול ביותר יכול להגיע ל-10 ספרות (כלומר יתבצעו 10 סריקות מלאות על המערך עד תום המיון). לבדיקת נכונות מימוש הפעולה radixSort(…), הדפיסו את המערך לפני המיון ואחריו.

ב. נתחו את יעלות הפעולה radixSort(…) שכתבתם. מה החיסרון הבולט של מיון זה?

בהצלחה!

פרק 8 דף עבודה מסי 2
מגדלי האנוי

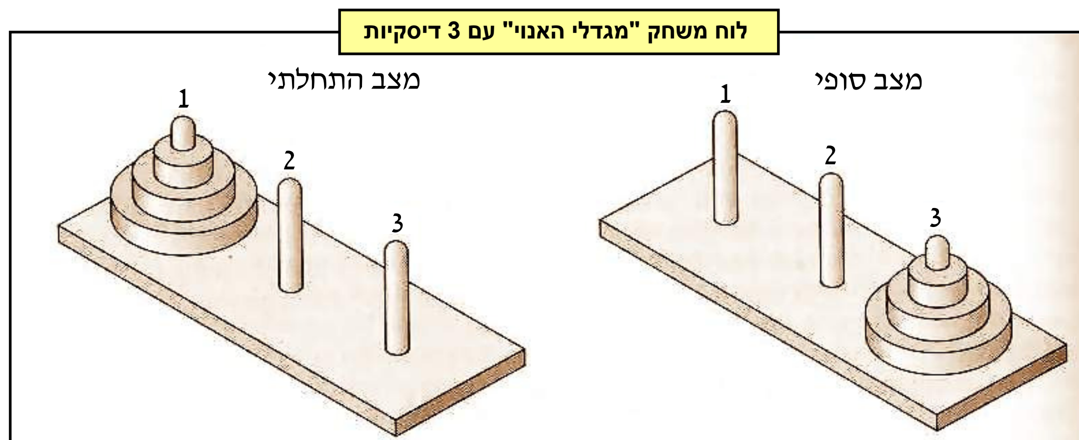
נשוב לבעיית מגדלי האנוי שפתרתם בפרק 4 – רקורסיה. הפעם יינתן לכם הפתרון במחלקה מוכנה. הדגש בדף העבודה יושם על בניית לוח המשחק תוך תרגול נושא המחסנית. הפתרון שיתקבל יוצג באופן גרפי.

חידת ״מגדלי האנוי״ (Towers of Hanoi) היא חידה מתמטית שהומצאה על ידי המתמטיקאי הצרפתי אדוארד לוקאס ב-1883, והפכה למשחק חביב. המשחק בנוי מלוח שעליו נעוצים שלושה מוטות הממוספרים 1, 2, 3. המשחק בנוי מלוח שבו נעוצים שלושה מוטות הממוספרים מ-1 עד 3 ומדיסקיות בהיקפים שונים. בתחילת המשחק, על מוט מספר 1 מושחלות n דיסקיות מן הדיסקית שהיקפה גדול בסדר יורד (כמתואר באיור). הדיסקיות יוצרות צורה של מגדל ומכאן שמו של המשחק. מטרת המשחק היא להעביר את מגדל הדסקיות בשלמותו ממוט 1 למוט 3 על פי הכללים האלה:

א. ניתן להעביר דיסקית אחת בלבד בכל שלב.

ב. באף שלב אסור שדיסקית גדולה תהיה מונחת על דיסקית קטנה ממנה.

לוח משחק "מגדלי האנוי" עם 3 דיסקיות



מה עליכם לעשות?

כתבו את המחלקה HanoiTowers המגדירה את לוח המשחק "מגדלי האנוי", על פי הממשק :

הפעולה בונה לוח מגדלי האנוי ובו 3 מוטות. על מוט מספר 1 מושחלות numOfDiscs דיסקיות בסדר יורד ביחס להיקפן – היקף הדיסקית העליונה הוא 1 ס"מ, מתחתיה דיסקית שהיקפה 2 ס"מ, עד הדיסקית התחתונה ביותר שהיקפה numOfDiscs. שני המוטות האחרים 2 ו-3 ריקים (כמתואר באיור "מצב התחלתי") יש לוודא שערכו של numOfDiscs יהיה בין 3 ל-10 כולל. אם לא, יש לקבוע אותו ל-3 כברירת מחדל	HanoiTowers(**int** numOfDiscs)
הפעולה שולפת את הדיסקית העליונה ממוט מספר fromPoleNum ומשחילה אותו על מוט מספר toPoleNum. הפעולה מחזירה **true** אם העברת הדיסקית התבצעה בצורה תקינה (דיסקית גדולה לא תונח על דיסקית קטנה ממנה, והמוט fromPoleNum אינו ריק), אחרת הפעולה תחזיר **false**. **הנחה** : קיימים מוטות שמספרם fromPoleNum ו-toPoleNum	**boolean** moveDisc(**int** fromPoleNum, **int** toPoleNum)
הפעולה מחזירה את מספר הדיסקיות שעל לוח המשחק	**int** getNumOfDiscs()
הפעולה מחזירה את מספר הדיסקיות המושחלות על מוט מספר poleNum. **הנחה** : קיים מוט שמספרו poleNum	**int** getNumOfDiscs(**int** poleNum)
הפעולה מחזירה את גודל הדיסקית שנמצאת בראש המוט שמספרו poleNum. אם המוט poleNum ריק, יוחזר הערך 0 **הנחה** : קיים מוט שמספרו poleNum	**int** getSizeTopDisc(**int** poleNum)
הפעולה מחזירה **true** אם המוט שמספרו poleNum ללא דיסקיות, ו-**false** אחרת	**boolean** isEmpty(**int** poleNum)

מהלך העבודה

1. כתבו את המחלקה HanoiTowers על פי הממשק לעיל. יצגו את 3 המוטות בעזרת 3 מחסניות. הקפידו שערכי המחסניות יהיו מספרים שלמים המייצגים את גודל הדיסקיות המושחלות על המוטות.

2. כדי לבדוק את המחלקה שכתבתם, הריצו את התוכנית HanoiTowersApplication.java הנמצאת במחשב שלכם בתיקייה HelpFiles. התוכנית תשתמש במחלקה שכתבתם ותבצע סימולציה גרפית של המשחק מגדלי האנוי.

בהצלחה!