פרק 7
רשימה

בפרק זה נכיר את ה**רשימה** (**List**) המאחסנת אוסף של נתונים ומגדירה את אופן הטיפול בהם. הרשימה כמבנה נתונים דומה למערך בכך שהיא מאחסנת איברים באופן סדרתי, אך היא גמישה מהמערך בפעולותיה. הרשימה גם אינה מוגבלת בגודלה, פרט למגבלה הנובעת מהגבלת הזיכרון המוקצה לתוכנית בזמן ריצה. יחס הסדר ברשימה, מגדיר לכל איבר ברשימה – פרט לאחרון – איבר עוקב, ולכל איבר ברשימה - פרט לראשון – איבר קודם. אם עוברים על איברי הרשימה החל באיבר הראשון, ומכל איבר אל העוקב לו, עוברים על כל איבר בדיוק פעם אחת ומסיימים באיבר האחרון.

ניתן להוסיף או למחוק איברים בכל מקום ברשימה: פעולות הוספה ומחיקה בכל מקום כלולות בממשק הרשימה, ולפיכך תוכנית המבצעת הכנסות ומחיקות ברשימה אינה צריכה להזיז איברים כדי לפנות מקום לאיבר חדש, או "לסגור חור" הנוצר ממחיקתו של איבר. כפי שנראה בהמשך, מימוש הרשימה מבטיח שגם במימוש פעולות הכנסה ומחיקה אלו אין צורך בהזזת איברים.

מכאן שהרשימה מתרחבת באופן טבעי ככל שמתווספים אליה איברים, ומתכווצת עם הוצאת איברים מתוכה. תכונה זו של הרשימה שימושית בזמן עיבוד אוספי נתונים גדולים בהם לא יודעים מראש את מספר האיברים המדויק שבו יש לטפל.

לדוגמה, אם נתונה הסדרה:

1, 4, 5, 7, 5

ורוצים להכניס איבר שערכו 3 בין האיבר הראשון לאיבר השני:

1, 4, 5, 7, 5

הסדרה לאחר ההכנסה תיראה כך:

1, 3, 4, 5, 7, 5

אם היינו רוצים לבצע זאת במערך היינו צריכים להזיז את האיברים החל מהשני ימינה, ורק אחרי שהתפנה מקום הייתה אפשרות לשלב את האיבר החדש למקומו. ברשימה ההכנסה תתבצע באופן מיידי וישיר, במקום הרצוי.

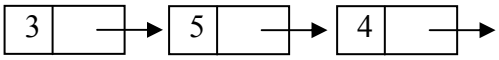
באופן דומה אם רוצים למחוק את האיבר השלישי בסדרה הבאה:

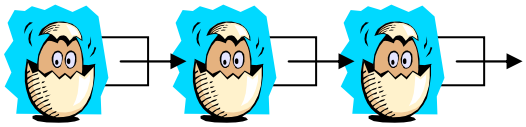
1, 5, 7, 4, 3

כך שאחרי המחיקה הסדרה תיראה כך:

1, 5, 4, 3

במערך היה צורך להזיז את כל האיברים החל מהרביעי שמאלה, אך במחלקה שנכתוב, תוגדר פעולת ההוצאה מרשימה כפעולת ממשק. בייצוג הרשימה שבו נשתמש ניתן יהיה לממש את הפעולה באופן יעיל יותר מאשר פעולת הוצאה ממערך, שאינה פעולת ממשק.
בפרק זה נשתמש בייצוג מקובל (אחד מהייצוגים האפשריים) לרשימה: שרשרת חוליות. ייצוג זה הוא המקנה לרשימה את הגמישות שהזכרנו לעיל. כל חוליה בשרשרת תכיל ערך והפניה לחוליה הבאה. הערך יכול להיות מטיפוס כלשהו: מספרים שלמים, מחרוזות או אובייקטים אחרים, ולפיכך הוא יוגדר באופן הכולל ביותר כטיפוס גנרי.





בשרשרת כזו ניתן להוציא חוליה ממקומה או לצרף לשרשרת חוליות לפי הצורך במספר קטן וקבוע של פעולות. מבנה שכזה פותר את שתי הבעיות שהעלה השימוש במערך: הוא אינו מוגבל בגודלו והוא מאפשר דינמיות של הכנסה והוצאה לכל אורך האוסף. רשימה המיוצגת על ידי שרשרת חוליות נקראת **רשימה מקושרת**. אנו, לשם קיצור נקרא לה פשוט **רשימה**.

א. הגדרת הרשימה

על מנת לבנות רשימה עלינו להגדיר שתי מחלקות. מחלקה אחת – **חוליה (Node)** המגדירה טיפוס של חוליה בודדת. (בשימוש בה ניתן לבנות שרשרות של חוליות, הן שרשרות המייצגות סדרות והן שרשרות המייצגות מבנים אחרים, כגון שרשרות מעגליות.) המחלקה הנוספת – **רשימה (List)** האחראית על המבנה של אוסף החוליות והקשרים ביניהן. הממשקים של שתי המחלקות יספקו לנו את כל הפעולות הנדרשות לטיפול במבנה הנתונים.

א.1. המחלקה חוליה - Node

במחלקת חוליה יש שתי תכונות: תכונה המחזיקה **ערך (info)** מטיפוס כלשהו, ותכונה אחרת המחזיקה הפניה לחוליה. מכיון שבשרשרת החוליות תעניין אותנו ההפנייה לחוליה הבאה, העוקבת לחוליה הנוכחית, נקרא לתכונה זו **עוקב (next)**. תכונת העוקב מאפשרת לשרשר כמה חוליות זו לזו. השימוש בהפניה בייצוג הנתונים הוא החידוש המאפשר את הגמישות והיתרונות של האוסף החדש. כאשר אין בעוקב הפניה לחוליה, יאוחסן בתכונה הערך **null**, ואנו נאמר שאין לחוליה – חוליה העוקבת לה. הערך המוחזק בחוליה יוגדר באופן כללי כטיפוס גנרי ויסומן על ידי T. החוליה כולה תהיה אם כך חוליה גנרית ותסומן כ- Node<T>.

Node<T>
private T info **private** Node<T> next
Node (T x) Node (T x, Node<T> p) T getInfo() Node<T> getNext() **void** setInfo (T x) **void** setNext (Node<T> p) String toString()

ממשק החוליה

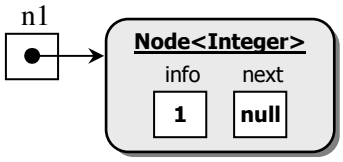
Node (T x)	הפעולה בונה חוליה. הערך של החוליה הוא x, ואין לה חוליה עוקבת
Node (T x, Node<T> p)	הפעולה בונה חוליה. הערך של החוליה הוא x והחוליה העוקבת לה היא p
T getInfo()	הפעולה מחזירה את הערך של החוליה
Node<T> getNext()	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה תחזיר **null**
void setInfo (T x)	הפעולה משנה את ערך החוליה להיות x
void setNext (Node<T> p)	הפעולה משנה את החוליה העוקבת להיות p
String toString()	הפעולה מחזירה מחרוזת המתארת את החוליה

נדגים זימון של הפעולה הבונה חוליות. הזימון מתבצע בזמן ריצת התוכנית ואז עלינו לציין את הטיפוס המדויק ממנו נרצה לייצר עצמים. הטיפוס הגנרי "יוחלף" בטיפוס קונקרטי, ובדוגמה הבאה בטיפוס 'מספרים שלמים':

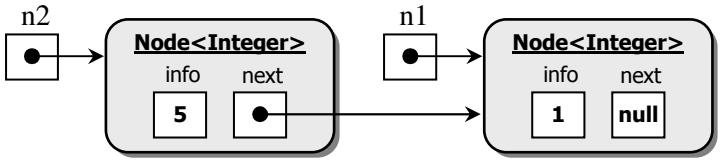
```
Node<Integer> n1;
n1  = new Node<Integer> (1);
Node<Integer> n2 = new Node<Integer> (5 , n1);
```

אחרי השורה הראשונה נוצר משתנה בשם n1 המכיל הפניה ריקה לחוליה:

אחרי השורה השנייה נוצרת חוליה שערכה 1 ואין לה חוליה עוקבת (ערך התכונה 'עוקב' הוא **null**). המשתנה n1 מפנה אל החוליה שנוצרה:



אחרי השורה השלישית נוצרת חוליה חדשה שערכה 5. היא מכילה הפניה לחוליה הראשונה, שאליה מפנה עדיין גם n1. את ההפניה אל החוליה החדשה מחזיק המשתנה n2:



? שימו לב לאופן בו שורשרו שתי החוליות. כתבו קוד המשרשר את שתי החוליות בסדר הפוך.

? כתבו קטע קוד היוצר מהשרשרת המתוארת בטקסט שרשרת מעגלית.

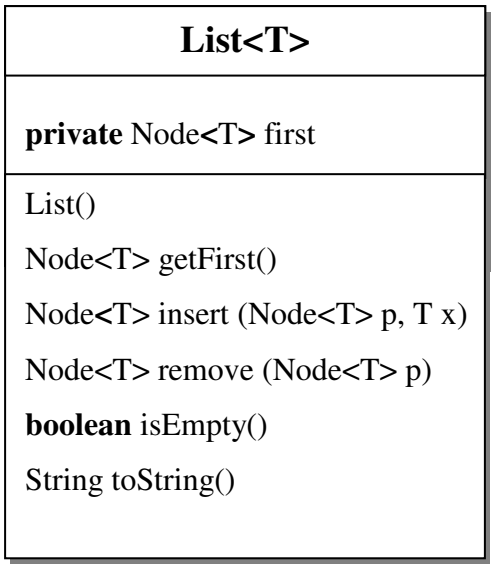
א.2. המחלקה רשימה - List

מחלקת רשימה היא המחלקה האחראית על ניהול שרשרת החוליות כסדרה. ברשימה ניתן להכניס ולמחוק חוליות בכל מקום בשרשרת. באומרנו **מקום (position)** ברשימה אנו מתכוונים למיקומו של איבר בשרשרת החוליות. המקום הוא **הפניה** לחוליה ברשימה. כאשר פעולה של הרשימה פועלת על מקום מסוים, יש להעביר לה את ההפניה לאותו מקום כפרמטר. שימו לב, כי הגדרה זו של מקום שונה מההגדרה במערך, שם מקום היה מספר סידורי. ברשימה, על מנת למחוק איבר במקום השלישי צריך להעביר לפעולת המחיקה את ההפניה לחוליה השלישית.

הרשימה היא עצם מורכב, בעל תכונה יחידה, שערכה הוא הפניה לחוליה הראשונה בשרשרת החוליות, והיא נקראת **ראש-הרשימה (list first)**. אם הרשימה ריקה, ערך ההפניה יהיה **null**. כיון שטיפוס התכונה 'ראש-הרשימה', הוא טיפוס החוליה שהוגדרה כחוליה גנרית, הרי

שהרשימה עצמה הופכת גנרית גם היא. הטיפוס הגנרי יופיע במקומות שונים כטיפוס הפרמטר המועבר לפעולות ובחלק מהפעולות ישמש כערך החזרה.

ככל עצם מורכב הרשימה יכולה להשתמש בפעולות הממשק של חוליה:



נסקור את הפעולות הנחוצות כדי לאפשר ניהול נכון של האיברים ברשימה:

מעבר על הרשימה

סריקה של רשימה נחוצה בהרבה שימושים של רשימות. ניתן לבצע סריקה של רשימה ולהגיע למקום כלשהו בה על ידי שילוב של כמה פעולות: פעולת getFirst() של הרשימה, מחזירה את החוליה הראשונה ברשימה, ופעולת getNext() של החוליה מאפשרת להמשיך ולהתקדם על הרשימה. כדי לדעת אם הרשימה ריקה, שאז אין מה לסרוק, נשתמש בפעולה isEmpty(). אם ברצוננו לעצור את הסריקה כיון שנמצא ערך המקיים תנאי רצוי, נשתמש בפעולה getInfo() של החוליה. דוגמאות של תכנות בסגנון זה מוצגות בהמשך הפרק.

הכנסה לרשימה

כדי להכניס חוליה בעלת ערך לרשימה במקום מסוים, עלינו לשלוח לפעולת ההכנסה את הערך החדש ואת מקום ההכנסה לרשימה ולקבל החלטה. במקום בו אנו רוצים להכניס את החוליה כבר קיימת חוליה אחרת. עלינו להחליט היכן תמוקם החוליה החדשה יחסית לחוליה הקיימת: לפניה או אחריה. נבחר להגדיר את פעולת ההכנסה כך, שהחוליה החדשה תיכנס לאחר החוליה הקיימת, במקום **העוקב** לפרמטר המקום. כשרוצים להכניס את הערך החדש במקום הראשון ברשימה יש לתת סימון מיוחד למקום, כך שההכנסה אכן תתבצע במקום הזה. נבחר ב-**null** כסימון המיוחד.

מכיון שפעמים רבות משולבת ההכנסה בסריקה של הרשימה ואנו מעוניינים להמשיך בסריקה מהמקום בו הכנסנו חוליה, יהיה זה הגיוני לקבוע שהפעולה תחזיר את המקום של החוליה שהוכנסה, כך ניתן יהיה להמשיך את הסריקה של הרשימה ממקום זה והלאה.
אם כך כותרת פעולת ההכנסה תראה כך:

```
public Node<T> insert (Node<T> p, T x)
```

הוצאה מרשימה

פעולת ההוצאה צריכה להוציא מהרשימה את החוליה הגנרית המופיעה במקום p ברשימה (בשונה מפעולת ההכנסה). לעתים קרובות נרצה להוציא חוליה מרשימה, ולאחר המחיקה להמשיך בסריקה של הרשימה. לאחר ההוצאה, אותה חוליה כבר אינה ברשימה, לכן נחוץ לנו שלאחר ההוצאה תחזיר הפעולה את המקום העוקב לחוליה שהוצאה. כך נוכל להמשיך את הסריקה על הרשימה ממקום זה והלאה. לכן תראה כותרת הפעולה כך:

```
public Node<T> remove  (Node<T> p)
```

שימו לב שהחוליה שהוצאה מהרשימה אמנם כבר אינה מהווה חלק מהרשימה, אך קיימת אליה הפנייה חיצונית כלשהי, שהרי הפנייה זו הועברה לפעולה כפרמטר, וערכה לא השתנה.

לפעולות הללו נוסיף את פעולת ה-toString() המוגדרת לכל מחלקה והמאפשרת לקבל תיאור של עצמים מטיפוס המחלקה. פעולה זו מתבצעת על ידי סריקה של הרשימה.

נסכם את ממשק הרשימה המכיל את כל הפעולות שנסקרו לעיל.

ממשק הרשימה

הפעולה בונה רשימה ריקה	List()
הפעולה מחזירה את המקום של החוליה הראשונה ברשימה הנוכחית. אם הרשימה ריקה, הפעולה תחזיר **null**	Node<T> getFirst()
הפעולה מכניסה לרשימה הנוכחית את האיבר x מקום אחד *אחרי* המקום p. הפעולה מחזירה את המקום של החוליה החדשה שהוכנסה. על מנת להכניס חוליה למקום הראשון ברשימה יש לשלוח **null** כפרמטר המקום. **הנחה**: p הוא מקום קיים ברשימה הנוכחית	Node<T> insert (Node<T> p, T x)
הפעולה מוציאה מהרשימה הנוכחית את האיבר הנמצא בה במקום p. מחזירה את המקום העוקב ל- p. אם הוצא האיבר האחרון – יוחזר **null**. **הנחות**: הרשימה אינה ריקה. p הוא מקום קיים (שאינו **null**) ברשימה הנוכחית	Node<T> remove (Node<T> p)
הפעולה מחזירה 'אמת' אם הרשימה הנוכחית ריקה, ו'שקר' אחרת	**boolean** isEmpty()
הפעולה מחזירה מחרוזת המתארת את הרשימה	String toString()

ב. דוגמאות לשימוש בפעולות הרשימה

בחלק הזה נדגים שימושים פשוטים בפעולות הרשימה. שימושים מורכבים יותר יוצגו בהמשך. כל אחת מהפעולות הבאות פועלת על הרשימה שהתקבלה מביצוע הפעולות הקודמות:

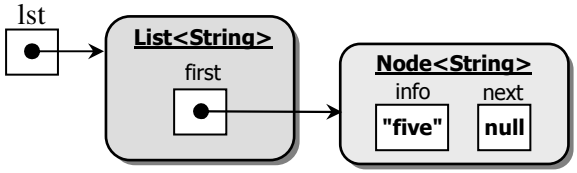
1. בניית רשימה ריקה מטיפוס מחרוזת:

```
List<String> lst = new List<String>();
```

2. הכנסת ערכים לרשימה:

הכנסת המחרוזת "five" במקום הראשון ברשימה:

```
lst.insert (null, "five");
```



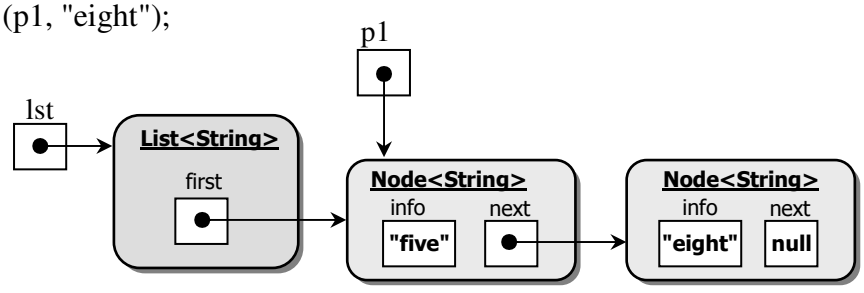
הכנסת הערך "eight" במקום השני ברשימה:

על מנת לעשות זאת עלינו למצוא קודם את ההפניה לחוליה הראשונה (שהיא החוליה הקודמת לשנייה):

```
Node<String> p1 = lst.getFirst();
```

ואז נכניס את הערך.

```
lst.insert (p1, "eight");
```



ניתן לעשות זאת גם בשורה אחת:

```
lst.insert (lst.getFirst(), "eight");
```

אפשרות אחרת היא שבפעולת ההכנסה הקודמת היינו שומרים את ההפניה המוחזרת על ידי פעולת ההכנסה, כך:

```
Node<String> p2 = lst.insert (null, "five");
```

אז יכולנו להכניס את הערך למקום השני כך:

```
lst.insert (p2, "eight");
```

כמובן ששני אופני פעולה אלו יוצרים כאן אותה רשימה, אך אין זה נכון במקרה הכללי: באופן הראשון הערך "eight" יוכנס תמיד למקום השני, ואילו באופן השני נכניס את "eight" אחרי האיבר שהוכנס לפניו, וזה לא חייב להיות המקום השני...)

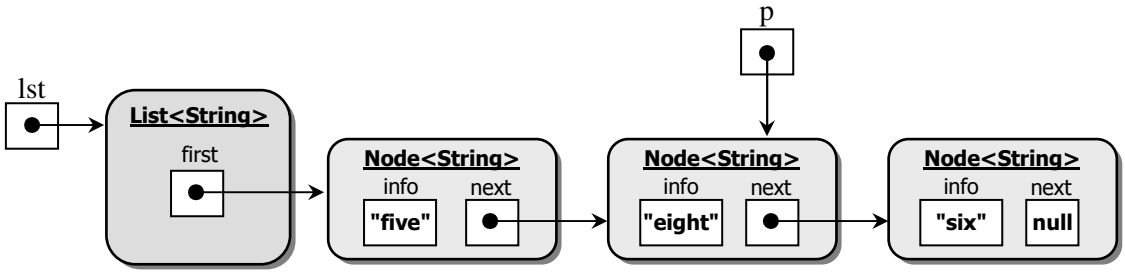
הכנסת הערך "six" במקום השלישי ברשימה:

נמצא את ההפניה לחוליה השנייה:

```
Node<String> p = lst.getFirst().getNext();
```

נכניס את הערך:

```
lst.insert (p, "six");
```



3. הוצאת איבר מרשימה:

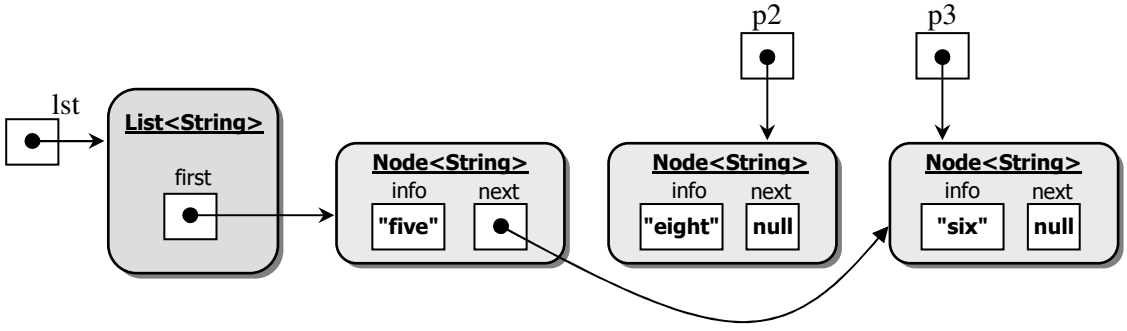
נוציא את האיבר שבמקום השני ברשימה:

שימו לב כי על מנת למחוק חוליה צריך להחזיק את ההפניה לאותה חוליה:

```
Node<String> p2 = lst.getFirst().getNext());
```

חשוב לזכור כי פעולה זו מחזירה את ההפניה לחוליה הבאה ובמידת הצורך נוכל לשמור אותה ולהשתמש בה:

```
Node<String> p3 = lst.remove (p2);
```



ג. מבנה הנתונים רשימה בשפת ג׳אווה

ג.1. המחלקה Node

ייצוג: במחלקת החוליה יש שתי תכונות: תכונת הערך (info) ותכונת העוקב (next). תכונת הערך היא גנרית. מהו הטיפוס של תכונת העוקב? תכונה זו היא משתנה המכיל הפניה לחוליה. כיוון שכך עליה להיות מטיפוס Node שהערך בו גנרי גם הוא:

```
public class Node<T>
{
        private  T  info;
        private Node<T> next;
}
```

האפשרות שעצם מטיפוס מחלקה מסוימת יכיל הפניה לעצם אחר מאותה מחלקה מאפשרת ליצור שרשרת עצמים שכולם מאותו טיפוס.

לפניכם המימוש המלא של המחלקה הגנרית Node:

```
/** מחלקה המגדירה חוליה גנרית
public class Node<T>
{
        private T info;
        private Node<T> next;
        /**
         *  הפעולה בונה חוליה. ערך החוליה נתון ואין לה חוליה עוקבת
         * @param info – המידע שיוכנס לתוך החוליה
         */
        public Node (T info)
        {
                this.info = info;
                this.next = null;
        }
        /**
         *  הפעולה בונה חוליה שערכה נתון וכן נתונה החוליה העוקבת לה
         * @param info – המידע שיוכנס לתוך החוליה
         * @param next – החוליה העוקבת
         */
```

```
        public Node (T info, Node<T> next)
        {
                this.info = info;
                this.next = next;
        }
        /**
         * הפעולה מחזירה את ערך החוליה
         * @return - ערך החוליה הנוכחית
         */
        public T getInfo()
        {
                return this.info;
        }
        /**
         * הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת הפעולה תחזיר null
         * @return - החוליה העוקבת
         */
        public Node<T> getNext()
        {
                return this.next;
        }
        /**
         * הפעולה משנה את ערך החוליה הנוכחית לערך הפרמטר
         * @param info – הערך החדש לחוליה
         */
        public void setInfo (T info)
        {
                this.info = info;
        }

        /**
         * הפעולה משנה את החוליה העוקבת להיות כערך הפרמטר
         * @param next – החוליה העוקבת
         */
        public void setNext (Node<T> next)
        {
                this.next = next;
        }
}
```

? ממשו בתוך המחלקה Node את הפעולה toString(). (הערה: הפעולה תחזיר את תיאור הערך השמור בחוליה ולא תיתן ביטוי לערך ההפניה.)

ג.2. המחלקה List

ייצוג: כפי שראינו קודם, המחלקה רשימה היא המחלקה שמחזיקה כתכונה את ההפניה לחוליה הראשונה ברשימה. כאשר הרשימה ריקה, הפניה זו ערכה **null**. כמו הגדרת החוליה, גם הגדרת הרשימה גנרית.

```
/**
 *  מחלקה המגדירה רשימה כאוסף של חוליות
 */
public class List<T>
{
        private Node<T> first;
        /**
         *  הפעולה בונה רשימה ריקה
         */

        public List()
        {
                this.first = null;
        }
        /**
         *  הפעולה מחזירה את המקום של החוליה הראשונה ברשימה הנוכחית.
         *  אם הרשימה ריקה הפעולה תחזיר null
         * @return - null המקום הראשון ברשימה או
         */

        public Node<T> getFirst()
        {
                return this.first;
        }
        /**
         *  הפעולה מכניסה לרשימה הנוכחית את האיבר החדש מקום אחד אחרי המקום הנתון
         * הפעולה מחזירה את המקום של החוליה החדשה.
         *  להכנסת חוליה ראשונה יש לשלוח null
         *  הנחה: המקום הנתון קיים ברשימה
         * @param p – מקום ההכנסה
         * @param x – הערך החדש להכנסה
         * @return  – מקום החוליה החדשה
         */
```

```
        public Node<T> insert (Node<T> p, T x)
        {
                Node<T> temp = new Node<T> (x);
                if (p == null)
                {
                        temp.setNext (this.first);
                        this.first = temp;
                }
                else
                {
                        temp.setNext (p.getNext());
                        p.setNext (temp);
                 }
                 return temp;
        }
        /**
         *  הפעולה מוציאה מהרשימה את האיבר הנמצא בה במקום הנתון ומחזירה את המקום
          * null העוקב. אם הוצא האיבר האחרון יוחזר
         *  null הנחה: הרשימה אינה ריקה והמקום הנתון קיים ברשימה ואינו
         * @param p - the Node to be removed
         * @return the node following the removed node (or null)
         */
        public Node<T> remove  (Node<T> p)          // p is not null
        {
                if (p == this.first)
                {
                        this.first = p.getNext();
                        p.setNext (null) ;
                        return this.first;
                }
                else
                {
                        Node<T> prv = this.first;
                        while (prv.getNext() != p)
                                prv = prv.getNext();
                        prv.setNext (p.getNext());
                        p.setNext (null);
                        return prv.getNext();
                }
        }
        /**
         *  הפעולה מחזירה 'אמת' אם הרשימה הנוכחית ריקה, ו'שקר' אחרת
         * @return אמת או שקר
         */
        public boolean isEmpty()
        {
                return (this.first == null);
        }
}
```

? ממשו בתוך המחלקה List את הפעולה toString().

ד. פעולות נוספות לרשימה

בסעיף זה נציג כמה דוגמאות. כל דוגמה פותרת בעיה על ידי כתיבת פעולה חדשה. כזכור, אנו קוראים לפעולה הכלולה בממשק הרשימה **פעולה פנימית**, או **פעולה של הרשימה**. פעולה פנימית יכולה להשתמש ישירות בייצוג הרשימה. פעולה הנכתבת מחוץ למחלקה תקרא **פעולה חיצונית** ותוכל להיעזר בפעולות הממשק בלבד.

פעולות חדשות שיוגדרו כפנימיות, בתוך המחלקה רשימה, יכולות לפעול על פרמטרים גנריים ולהחזיר ערכים גנריים בהתאמה לטיפוס של הרשימה עצמה. פעולות חדשות שיוגדרו כחיצוניות למחלקה רשימה, יוגדרו עבור טיפוסים קונקרטיים בלבד, כלומר הפרמטרים המועברים לפעולה יהיו ספציפיים ומוגדרים במדויק ולא כלליים. כבר הסברנו החלטה זו בפרק הקודם: הגדרה של פעולות חיצוניות גנריות דורשת שימוש בכלים שאינם כלולים ביחידה זו, ומשום כך אנו נמנעים מהגדרות כאלו.

ד.1. פעולות חיצוניות

דוגמה 1: אורך רשימה

כתבו אלגוריתם לפעולה getLength(), המקבלת רשימה כפרמטר ומחזירה את מספר האיברים ברשימה. ממשו את הפעולה.

שימו לב שעל פי הכלל שעליו החלטנו תוגדר הפעולה getLength() כפעולה חיצונית הפועלת על רשימה קונקרטית בלבד, הפעם נחליט על רשימה של מספרים שלמים.

פתרון

אלגוריתם:

החזר אורך רשימה (lst)

*{**טענת כניסה:** הפרמטר הוא רשימה של מספרים שלמים }*

*{**טענת יציאה:** הפעולה מחזירה את מספר האיברים ברשימה}*

עבור על הרשימה מתחילתה ועד סופה

החזר את מספר צעדי המעבר מחוליה לחוליה שבוצעו.

קוד:

```
public static int getLength (List<Integer> ls)
{
        int len = 0;
        Node<Integer> p = ls.getFirst();
        while ( p!=null)
{
                len++;
                p = p.getNext();
}
return len;
}
```

דוגמה 2: אחזור מקום ברשימה של ערך נתון

פעולות הממשק של חוליה ורשימה מאפשרות לנו לקבל ערך של חוליה לפי מקומה (כזכור, מקום הוא הפניה לחוליה), אך לא להיפך. הפעולה הבאה מקבלת ערך ורשימה ומחזירה מקום של חוליה ברשימה המכילה את הערך.

שימו לב כי ייתכן שאותו הערך קיים כמה פעמים ברשימה, לכן נחדד את הגדרת הפעולה כך:

Node<Integer> getPlace (List<Integer> lst, **int** x)

היא פעולה המקבלת מספר שלם x ורשימה lst של שלמים. הפעולה מחזירה את המקום הראשון ברשימה lst שבו מופיע x. אם x אינו מופיע תחזיר הפעולה **null**.

כתבו אלגוריתם לפעולה וממשו אותה.

פתרון

אלגוריתם:

אחזר-מקום (ls, x)

*{**טענת כניסה:** הפעולה מקבלת מספר שלם x ורשימה של שלמים lst}*

*{**טענת יציאה:** הפעולה מחזירה את המקום הראשון של x ברשימה ls אם x מופיע בה,*

*אחרת מחזירה **null**}*

עבור על חוליות הרשימה lst מתחילתה

אם ערך החוליה הנוכחית שווה ל-x, החזר את מקומה.

*אם הגעת לסוף הרשימה (כולל המקרה שהרשימה ריקה) החזר **null**.*

קוד:

```
public static Node<Integer> getPlace ( int x, List<Integer> ls)
{
        Node<Integer> p = ls.getFirst();
        while ( p!=null )
        { //כל עוד לא הגעת לסוף הרשימה
                if ( p.getInfo() == x)
                        return p;
                else
                        p = p.getNext();
        }
        return null; // הגענו לסוף הרשימה
}
```

המעבר מהאלגוריתם לקוד:

1. המעבר על חוליות הרשימה מתבצע בעזרת ההפניה p. הפניה זו מאותחלת לראש הרשימה ומתקדמת במהלך התוכנית על ידי הפקודה getNext(). המעבר מסתיים כאשר מתקיים התנאי הקבוע בתוכנית או כאשר הגענו לסוף הרשימה (p==**null**).
2. אם הגענו לשורה האחרונה באלגוריתם, פירוש הדבר שהגענו לסוף הרשימה, ומכאן שלא נמצאה חוליה המכילה את הערך x, לכן ערך ההחזרה הוא **null**.

דוגמה 3: ביטול כל מופעי ערך ברשימה

כתבו פעולה removeAllOccurences (בטל-כל-המופעים) המקבלת מספר שלם x ורשימה ls של מספרים שלמים, ומוציאה מהרשימה ls את כל החוליות שערכן x.

פתרון

אלגוריתם:

בטל-כל-המופעים (ls, x)

*{**טענת כניסה**: הפעולה מקבלת מספר שלם x, ורשימה ls של מספרים שלמים}*

*{**טענת יציאה**: הפעולה מוחקת מהרשימה ls את כל החוליות שערכן x}*

עבור על כל חוליות הרשימה ls מתחילתה ועד סופה

אם ערך החוליה הנוכחית שווה ל-x, הוצא אותה מהרשימה.

קוד:

```
public static void deleteAllOccurances (int x, List<Integer> ls)
{
        Node<Integer>  p = ls.getFirst();
        while (p != null)
        {
                if (p.getInfo() == x)
                        p = ls.remove (p);
                else
                        p = p.getNext();
        }
}
```

המעבר מהאלגוריתם לקוד:

1. פעולה זו אינה מחזירה ערך, אלא משנה את הרשימה שהועברה אליה כפרמטר. לכן טיפוס ההחזרה הוא **void**.
2. מעבר על הרשימה מתבצע בעזרת ההפניה p שעוברת על כל איברי הרשימה. קידום ההצבעה אל החוליה הבאה נעשה בשתי דרכים שונות. בפעולת ה-()remove הקידום אוטומטי, לפי הגדרת הפעולה. אבל, אם הערך בחוליה אינו x יש לקדם את ההפניה באופן מפורש בעזרת ()getNext.

ד.2. פעולות פנימיות

נחזור לדוגמה הראשונה שראינו בסעיף הקודם: מציאת אורך של רשימה. הפעם נגדיר אותה כפעולה פנימית המתווספת לפעולות הרשימה הגנרית. הדבר אפשרי שכן הפעולה אינה משתמשת בערכים של איברי הרשימה.

דוגמה 1: אורך רשימה

כתבו פעולה בשם ()getLength, המחזירה את מספר האיברים ברשימה הנוכחית.

פתרון

אלגוריתם:

החזר אורך רשימה

*{**טענת כניסה:** פעולה על הרשימה הנוכחית }*

*{**טענת יציאה:** הפעולה מחזירה את מספר האיברים ברשימה}*

עבור על הרשימה מתחילתה ועד סופה

החזר את מספר צעדי המעבר מחוליה לחוליה שבוצעו.

קוד:

```
public int getLength()
{
        int len = 0;
        Node<T> p = this.first;
        while ( p!=null)
        {
                len++;
                p = p.getNext();
        }
        return len;
}
```

ההבדלים בין המימוש כפעולה חיצונית שהצגנו למעלה ובין מימוש זה, קטנים. ההבדל הראשון: כיון שזו פעולה של הרשימה, המחזירה את האורך של עצם מטיפוס הרשימה עליו היא מופעלת, הרשימה אינה פרמטר. שנית, ההתייחסות לרשימה בגוף התוכנית נעשית בעזרת הפנייה **this**

במקום אזכור שם הפרמטר. שלישית, ניתן להשתמש בשם התכונה first במקום להפעיל את getFirst(), ורביעית, בעוד שהפעולה החיצונית נכתבה עבור טיפוס קונקרטי, כאן הפעולה הפנימית היא גנרית ואמנם היא משתמשת במשתנה מהטיפוס Node<T>.

אין למעשה שום הבדל בין היעילות של שני המימושים – בשניהם מספר הצעדים המתבצע הוא (בקירוב) כאורך הרשימה. אך מימושה כפעולה פנימית פותח לפנינו אפשרויות שאינן קיימות במימוש פעולה חיצונית. בדוגמה זו למשל, יכול המתכנת להוסיף לרשימה תכונה נוספת, הנקראת length, לשנות את מימושי פעולות ההכנסה והמחיקה כך שפעולת ההכנסה תגדיל ערך תכונה זו ב-1, ופעולת המחיקה תקטין אותו ב-1. אז ניתן לממש את הפעולה המחזירה את האורך כך:

```
public int getLength()
{
    return this.length;
}
```

מציאת אורך של רשימה היא בעיה נפוצה למדי ולכן הפתרון שלה כלול כפעולה פנימית ברבות מהספריות המממשות רשימות. לעומת פעולה זו, את הפעולות 2 ו-3 שבדוגמאות 3 לעיל, אי אפשר לכתוב כפעולות פנימיות של הרשימה הגנרית List<T> כיון שהן מפעילות פעולות על ערכים ברשימה. שתיהן משוות את הפרמטר לערך שבמקום הנוכחי בעזרת ==. כיון שהטיפוס T גנרי, אין בטחון שכל טיפוס המחליף אותו תומך בפעולה זו, עם המשמעות שאנו מעוניינים בה. לכן את הפעולות הללו אפשר להוסיף רק כפעולות חיצוניות.

בהמשך נציג בחלק מהדוגמאות פעולות פנימיות ובאחרות חיצוניות. אך כפי שציינו הממשק המקורי של הרשימה שהוצג לעיל, לא ישתנה גם כאשר נתרגל הוספה של פעולות פנימיות.

ד.3. ממשק מינימלי או ממשק מורחב?

בנקודה זו, ניתן לשאול, מדוע לא נכללו הפעולות שפרטנו לעיל בממשק הרשימה ובמימושה? אילו היו כלולות, לא היה נזקק כל מתכנת המעוניין בהן לכתוב אותן שוב בעצמו. טענה זו נכונה, אך מצד שני יש לזכור כי יש מגוון גדול מאוד, למעשה אין-סופי, של פעולות על רשימות, שכל אחת מהן שימושית ליישומים מסוימים ולא ניתן לכלול את כולן בממשק הרשימה.

אנו הצגנו ממשק מסוים המספק אותנו ביחידה זו, אך בפועל ייתכנו ממשקים שונים לרשימה. בשפות תכנות מודרניות קיימות ספריות תוכנה המממשות מבני נתונים כגון רשימה. בדרך כלל הן מציעות אוסף הרבה יותר גדול של פעולות בממשק הרשימה ממה שאנו הצענו, כדי לספק למשתמשים ספריה עשירה של פעולות. הפעולות הכלולות בספריה נבחרות על ידי בוני הספרייה, כשהשיקול העיקרי הוא האם הן ישימות למגוון רחב של יישומים. שיקול נוסף הוא, האם מימוש

פעולה כפעולה של הרשימה, היכולה לגשת ישירות לייצוג מבנה הנתונים יאפשר מימוש יעיל יותר. בהמשך נדגים שיקול זה.

בסעיפים הקודמים ראינו שלוש דוגמאות של בעיות שפתרונותיהן מומשו כפעולות חיצוניות, ואת הפעולה הראשונה מימשנו גם כפעולה פנימית תוך הצבעה על הבדל בין שני הפתרונות. בפתרון זה הנחנו למעשה שאוסף פעולות הממשק הורחב ומימוש הפעולה הוכלל במימוש המחלקה רשימה. אך יש להבין, שמתכנת המקבל ספריה סגורה ובה מימוש של רשימה אינו יכול לשנות את המימוש, ומוגבל לשימוש בממשק הנתון לו בלבד.

ה. פעולות מתקדמות

ה.1. יצירת רשימה חדשה תוך כדי מעבר על רשימה נתונה

ניתן ליצור רשימה חדשה מרשימה נתונה על פי חוקיות מסוימת: רשימה שהיא תת-רשימה של רשימה מסוימת, רשימה הכוללת רק את האיברים הזוגיים או אי-זוגיים, או שהיא מכילה רק את האיברים הגדולים מאיבר מסוים וכיו"ב.

דוגמה 1

כתבו פעולה המקבלת רשימה של שלמים ומספר נוסף. הפעולה מחזירה רשימה חדשה הכוללת את המספרים ברשימה המתחלקים במספר זה.

פתרון

אלגוריתם:

*{**טענת כניסה**: הפעולה מקבלת רשימה ls של שלמים ומספר x}*

*{**טענת יציאה**: הפעולה מחזירה רשימה של המספרים השלמים מתוך רשימה ls*
המתחלקים במספר הנתון, על פי סדר הופעתם המקורי ברשימה}

רשימת_המתחלקים (lst, x)

בנה רשימה חדשה ריקה lst2.

עבור על הרשימה lst מתחילתה,

אם הערך של החוליה הנוכחית מתחלק ב-x, הכנס אותו לסוף lst2.

החזר את lst2.

קוד:

```
public  static List<Integer>  dividedByList (List<Integer> lst, int x)
{
        List<Integer> lst2 = new List<Integer>();
        Node<Integer> p2 = null;        // reference to the last node in lst2

        Node<Integer> p1 = lst.getFirst();
        while (p1 != null)
        {
                if ((p1.getInfo() % x) == 0)
                {
                        p2 = lst2.insert (p2, p1.getInfo());
                }
                p1 = p1.getNext ();
        }
        return lst2;
}
```

המעבר מאלגוריתם לקוד:

1. כשמטפלים בשתי רשימות צריך שני משתנים שיכילו הפניות נפרדות, אחת לכל רשימה.
2. שימו לב כי קידום ההפניה p2 נעשה באמצעות הפעולה insert בעוד שהקידום של ההפניה p1 נעשה על ידי הפעולה getNext().
3. p2 משמש לצורך פעולת הכנסה המקדמת את ההפניה לחוליה החדשה שהוכנסה, לכן פעולת הקידום שלו מתקיימת באופן אוטומטי. לא ניתן לקדם את p2 באמצעות הפעולה getNext(), כיוון שבכניסה הראשונה ללולאה הערך של p2 הוא **null**, ולכן הפניה אל p2 כדי לקבל את הבא אחריו היא בלתי אפשרית ותגרום לחריגה. השימוש בערך החזרה של פעולת ה-insert חוקי תמיד, ומבטיח את נכונות התוכנית כיון שהוא קושר את הקידום לפעולה שבגללה הוא נדרש.

ה.2. טיפול בשתי רשימות

דוגמה 2: שיוויון בין רשימות

לעתים קרובות נרצה לסרוק שתי רשימות במקביל ולבצע עליהן פעולה מסוימת. לא כל פעולה שכזו יכולה להיכתב כפעולה פנימית. למשל, אם נגדיר שוויון של רשימות כשוויון של סדרות האיברים שלהן, נצטרך לממש את פעולת השוויון בין רשימות על ידי מעבר במקביל על שתי הרשימות, והשוואת כל זוג איברים. אין פעולת השוואה אחת המתאימה לכל סוגי הטיפוסים

שיכולים להיות טיפוסי הערך באיברים ולכן פעולה זו תוכל להתבצע רק כפעולה חיצונית לגבי רשימות מטיפוס קונקרטי שמוגדרת לגביו פעולת השוואה. פעולות אחרות יוכלו להתבצע כפעולות פנימיות, למשל כאשר נרצה למזג שתי רשימות לרשימה משולבת אחת בהתאמה. כאן, כאשר איננו מתייחסים באופן מפורט לערך החוליה אין בעיה להגדיר את הפעולה כפנימית.
מכאן שכאשר נרצה לטפל בשתי רשימות נצטרך לבחון את הבעיה המסוימת ולהחליט אם לממשה כפעולה פנימית או חיצונית.

בעיה: כתבו פעולה חיצונית equals המקבלת כפרמטרים שתי רשימות. הפעולה תחזיר "אמת" אם שתי הרשימות שוות, ו-"שקר" אחרת. כיון שהפעולה חייבת לקבל רשימות מטיפוס איבר קונקרטי, נחליט כי זהו הטיפוס String.

פתרון

אלגוריתם:

*{**טענת כניסה**: הפעולה מקבלת שתי רשימות מחרוזתיות }*

*{**טענת יציאה**: הפעולה מחזירה "אמת" אם שתי הרשימות שוות ו-"שקר" אחרת}*

עבור על שתי הרשימות במקביל,

*אם ערכי החוליות הנוכחיות שונים, החזר **false**.*

*אם הגעת לסוף שתי הרשימות, החזר **true***

*אחרת החזר **false**.*

קוד:

```
public static boolean equals (List<String> one, List<String> other)
{
        Node<String> one_p = one.getFirst();
        Node<String>  other_p = other.getFirst();
        while (one_p!= null  &&  other_p != null)
        {
                if (one_p.getInfo().compareTo (other_p.getInfo()) != 0)
                        return false;
                one_p = one_p.getNext();
                other_p = other_p.getNext();
        }
                return ((one_p ==  null) && (other_p == null));
}
```

מעבר מאלגוריתם לקוד:

בטיפול בשתי רשימות משתמשים בשתי הפניות, אחת לכל רשימה.

דוגמה 3: מעבר מקביל על שתי רשימות ויצירת רשימה שלישית

לעתים קרובות נרצה לסרוק שתי רשימות במקביל, וליצר מהן רשימה שלישית: למשל, ניתן לבצע פעולת חשבון כלשהי על הערכים במקום ה-n בשתי הרשימות, ולהכניס את התוצאה לרשימה שלישית.

בעיה: כתבו פעולה המקבלת שתי רשימות של מספרים שלמים השונים זה מזה. הפעולה מחזירה רשימה חדשה המוגדרת באופן הבא: הערך במקום ה-n ברשימה החדשה הוא המקסימום בין הערכים שבמקומות ה-n בשתי הרשימות הנתונות. אם החל ממקום כלשהו קיימים איברים רק באחת משתי הרשימות, הם יצורפו לרשימה החדשה.

פתרון

אלגוריתם:

*{**טענת כניסה**: הפעולה מקבלת שתי רשימות של מספרים שלמים השונים זה מזה}*

*{**טענת יציאה**: הפעולה מחזירה רשימה חדשה המכילה את האיבר המקסימלי מבין האיברים במקום ה-n ברשימות המקוריות}*

בנה רשימה ריקה lst3.

עבור על שתי הרשימות במקביל,

אם קיימים ערכים בשתי הרשימות, הכנס את הגדול מביניהם למקום האחרון ב lst3.

אם הגעת לסוף אחת הרשימות, ובאחרת עדיין יש איברים -

העתק את יתר האיברים לפי הסדר ל-lst3.

קוד:

```
public static List<Integer> maxValue (List<Integer> lst1, List<Integer> lst2)
{
        List<Integer> lst3 = new List<Integer>();
        Node<Integer> p1 = lst1.getFirst();
        Node<Integer> p2 = lst2.getFirst();
        Node<Integer> p3 =  null;           // reference to last position in lst3
        while ( p1 != null  &&  p2 != null)
        {
                if (p1.getInfo() > p2.getInfo())
                        p3 = lst3.insert (p3, p1.getInfo());
                else
                        p3 = lst3.insert (p3, p2.getInfo());
                p1 = p1.getNext();
                p2 = p2.getNext();
        }
```

```
        Node p;
        if (p2 == null)
                p = p1;
        else
                p = p2;
        while (p != null)
        {
                p3 = lst3.insert (p3, p.getInfo());
                p = p.getNext();
        }
        return lst3;
}
```

מעבר מאלגוריתם לקוד:

1. בטיפול בשלוש רשימות משתמשים בשלוש הפניות, אחת לכל רשימה.
2. קידום ההפניות p1 ו-p2 נעשה באמצעות הפעולה getNext().
3. קידום ההפניה p3 נעשה באמצעות הפעולה insert.
4. ההפניה p משמשת כהפניית עזר אך למעשה היא p1 או p2.

ה.3. שתי הפניות לאותה רשימה

ראינו כי כאשר מטפלים במספר רשימות, יש צורך בדרך כלל, לפחות בהפניה אחת לכל רשימה. בדוגמה הבאה נראה כי לפעמים יש צורך ביותר מהפניה אחת לאותה רשימה.

דוגמה 4: מיון בחירה

כתבו אלגוריתם לפעולה הממיינת רשימה נתונה של מספרים לפי סדר עולה באמצעות "**מיון בחירה**" (המיון נעשה על גבי הרשימה הנתונה). ממשו את האלגוריתם כפעולה חיצונית.

פתרון

אלגוריתם:

*{**טענת כניסה**: הפעולה מקבלת רשימה lst לא ריקה של מספרים שלמים}*

*{**טענת יציאה**: הפעולה מחזירה את הרשימה לאחר שמיינה את איבריה בסדר עולה בעזרת "מיון בחירה"}*

עבור על הרשימה lst באמצעות המצביע p1 מתחילתה עד החוליה לפני האחרונה

עבור על הרשימה lst באמצעות מצביע p2 החל מהחוליה העוקבת ל-p1 ועד סופה

אם האיבר במקום p2 קטן מהאיבר במקום p1 , החלף את ערכי האיברים.

קוד:

```
public static void selectionSort (List<Integer> list)
{
        int tmp;
        Node<Integer> pos1, pos2;

        pos1 = list.getFirst();
        while (pos1 != null)
        {
                pos2 = pos1.getNext();
                while (pos2 != null)
                {
                        if (pos2.getInfo() < pos1.getInfo())
                        {
                                // החלפה בין הערכים של שני האיברים
                                tmp = pos1.getInfo();
                                pos1.setInfo (pos2.getInfo());
                                pos2.setInfo (tmp);
                        }
                        pos2 = pos2.getNext();
                }
                pos1 = pos1.getNext();
        }
}
```

מעבר בין אלגוריתם לקוד:

שתי ההפניות שהוגדרו הן הפניות לאותה רשימה, המצביעות על איברים שונים אותם ברצוננו להשוות.

ה.4. רשימות עם טיפוסים שונים

לפעמים באותה תוכנית נצטרך לטפל בכמה רשימות שטיפוסי הערכים בהן שונים. הטיפוסים יכולים להיות בסיסיים, או עצמים. השימוש במחלקת הרשימה הגנרית מאפשר לנו ליצור מופעים שונים של אותה מחלקה כאשר כל מופע יכול להכיל ערכים מטיפוס שונה.

דוגמה 5: כיווץ

ביישומים רבים, רוצים לכווץ מידע קיים. בתרגיל זה נבצע כיווץ של רשימה. לדוגמה, ניתן לכווץ את הרשימה list1 ולקבל את הרשימה list2:

list1: A, A, A, A, B, F, F, F, A, A, B, B, B, B, B, B, B, B, T, T ,T…

list2: A-4, B-1, F-3, A-2, B-8, T-3 …

הרשימה list2 היא כיווץ של הרשימה list1 כאשר עבור כל אות הוספנו את מספר המופעים שלה ברצף.

בעיה: בהינתן רשימה של תווים, יש לבנות רשימה חדשה שבה ערך כל חוליה הוא עצם המורכב משני שדות: האחד הוא תו והאחר הוא מספר. הרשימה החדשה נוצרת באופן הבא: מכל רצף של

איברים המכילים אותו תו ברשימה הנתונה, נוצרת ברשימה החדשה חוליה שהעצם שבה מכיל את התו ואת מספר מופעיו ברצף הזה (ראו את האיורים בהמשך).

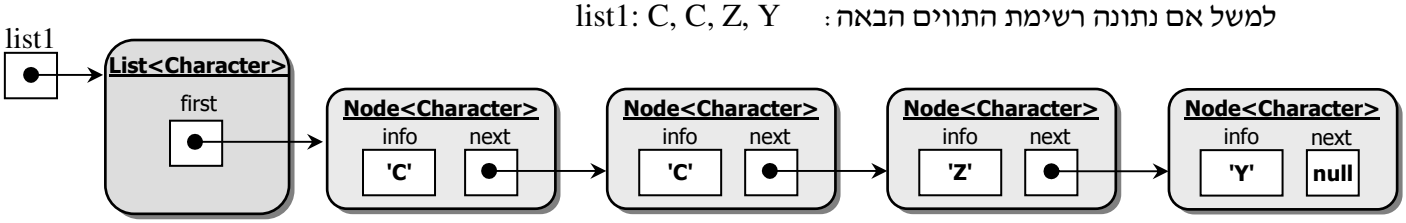
פתרון

ניצור מחלקה חדשה בשם ZipInfo שתאפשר ליצור עצמים המכילים תו ומספר.

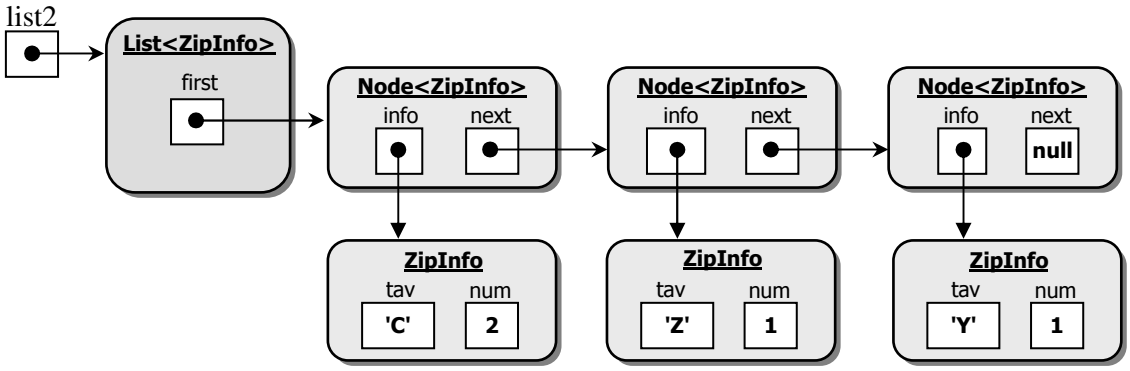
המחלקה מתוארת על ידי ה-UML הבא:

ZipInfo
private char tav; **private long** num;
ZipItem(**char** tav, **long** num) **char** getTav() **long** getNum() **void** setTav(**char** tav) **void** setNum(**long** num) String toString()

למשל אם נתונה רשימת התווים הבאה: list1: C, C, Z, Y



הפעולה תחזיר את הרשימה הבאה שאבריה מטיפוס ZipInfo: list2: C-2, Z-1, Y-1



אלגוריתם:

*{**טענת כניסה**: האלגוריתם מקבל רשימה של תווים}*

*{**טענת יציאה**: האלגוריתם מחזיר רשימה חדשה שבה כל חוליה מכילה עצם מטיפוס ZipInfo ובו תו ומספר המופעים שלו ברצף ברשימה המקורית}*

בנה רשימה חדשה שערך איבריה מטיפוס ZipInfo. p1 מתקדם על רשימה זו מראשה.

הצב הפניה p לתחילת הרשימה של התווים.

כל זמן ש-p לא הגיע לסוף הרשימה בצע:

קדם את p עד שהתו בחוליה שאליה הוא מצביע - משתנה,

או שהוא הגיע לסוף הרשימה.

הכנס לרשימה החדשה (מטיפוס ZipInfo), במקום p1,

חוליה עם התו שנקרא עד כאן ומספר מופעיו.

החזר את הרשימה החדשה.

קוד:

אנו מביאים רק חלקים מקוד הפתרון:

```
public class ZipInfo
{
	private char tav;
	private long num;
	...
}
```

הפעולה הבאה תופיע במחלקה ListUtil המספקת שירותים עבור רשימות או במחלקה ספציפית ListZip האמורה להציע פתרונות לנושא כיווץ ופתיחת מחרוזות:

```
public static List<ZipInfo> zip (List<Character> lst)
{
	List< ZipInfo> lst1 = new List< ZipInfo> ();
	Node< ZipInfo> p1 = lst1.getFirst();
	Node<Character> p = lst.getFirst();

	int c;
	char t;
	while (p != null)
	{
		c=1;
		t = p.getInfo();
		p = p.getNext();
		while((p != null) && (t == p.getInfo()))
		{
			c++;
			p = p.getNext();
```

```
            }
            ZipInfo x = new ZipInfo (t, c);
            p1 = lst1.insert (p1,x);
        }
        return (lst1);
}
```

ו. ייצוג טיפוסי נתונים מופשטים בעזרת מבנה הנתונים רשימה

ו.1.ייצוג מחסנית בעזרת רשימה

בפרק הקודם ייצגנו את המחסנית באמצעות מערך. נראה כעת ייצוג אחר באמצעות רשימה, וכך באופן טבעי לא תהיה המחסנית מוגבלת בגודלה שכן הרשימה אינה מוגבלת בגודלה.

טיפוס הנתונים מחסנית הוא בעל חוקים וכללי התנהגות ברורים. על מנת לאכוף חוקים אלו, נגדיר רשימה מטיפוס המתאים לטיפוס איברי המחסנית, כתכונה *פרטית* של המחסנית. המשתמש במחלקת המחסנית לא יוכל לגשת לתכונה הפרטית הזאת ולהשתמש בכל היכולות של הרשימה, אלא רק בפעולות שהגדרנו על המחסנית:

```
public class Stack<T>
{
        private List<T> data;

        public Stack()
        {
                this.data = new List<T>();
        }

        public boolean isEmpty()
        {
                return this.data.isEmpty();
        }

        public void push (T x)
        {
                … // השלימו בעצמכם!!
        }

        public T pop()
        {  // הנחה: המחסנית אינה ריקה
                Node<T> first = this.data.getFirst();
                T x = first.getInfo();
                this.data.remove (first);
                return x;
        }

        public T top()
        {
```

```
            return this.data.getFirst().getInfo();
        }

        public String toString()
        {
            השלימו בעצמכם!!  // ...
        }

}
```

הערות:

1. בייצוג הנוכחי אין צורך להגדיר את תכונת ה-top, כיוון שבתוך רשימה תמיד ניתן לגשת לראש הרשימה.
2. בייצוג הנוכחי פעולת ה-pop מוחקת בפועל את האיבר מהרשימה, לעומת הייצוג במערך, שם לא בוצעה מחיקה, אלא המקום של האיבר סומן כלא שייך למחסנית וניתן היה להכניס למקום זה איבר חדש.
3. בפועל, אנו משתמשים כאן רק בחלק (קטן) מאוסף הפעולות של הרשימה. אין בכך פגם. אם טרחנו ובנינו מבנה נתונים כרשימה עם אוסף פעולות, אין סיבה שלא נשתמש בו גם ליישומים שבהם איננו זקוקים לכל הפעולות. (חלופה אחרת היא לממש את המחסנית ישירות כשרשרת של חוליות, בלא להזדקק לרשימה עוטפת. עבור המחסנית יש לכתוב רק מעט קוד למימוש ישיר כזה, ולכן זו חלופה סבירה.)

2.ו. ייצוג תור בעזרת רשימה

תור הוא טיפוס נתונים הניתן לתיאור כרשימה עם חוקים המגבילים הכנסה והוצאה. הוצאת איברים בתור מותרת רק בצד אחד, הקרוי ראש התור, והכנסת האיברים מותרת רק בצד האחר, הקרוי סוף התור (או 'זנב התור').

לפניכם ממשק התור:

Queue()	הפעולה בונה תור ריק
boolean isEmpty()	הפעולה מחזירה 'אמת' אם התור הנוכחי ריק, ו'שקר' אחרת
void insert (T x)	הפעולה מכניסה את הערך x לסוף התור הנוכחי
T remove()	הפעולה מוציאה את הערך שבראש התור הנוכחי ומחזירה אותו. הנחה: התור הנוכחי אינו ריק
T head()	הפעולה מחזירה את ערכו של האיבר שבראש התור מבלי להוציאו. הנחה: התור הנוכחי אינו ריק
String toString()	הפעולה מחזירה מחרוזת המתארת את התור

נייצג את התור בעזרת רשימה בדומה למה שעשינו עם המחסנית בסעיף הקודם. נגדיר את הרשימה כתכונה פרטית data של התור, כך שמשתמש במחלקת התור לא יוכל לגשת אליה, אלא רק להשתמש בפעולות של התור.

את הוצאת האיברים מתחילת התור קל לממש בעזרת הפעולות של הרשימה, כיוון שקל להגיע לתחילת הרשימה ולהוציא את האיבר הנמצא שם. יותר מורכב לגשת לסוף הרשימה כדי להכניס איבר חדש, כיוון שאין ברשימה הפניה לסוף הרשימה. ניתן לעשות זאת על ידי הגדרת פעולת עזר פרטית בתוך התור:

```
private Node<T> getLast()
{  //  הנחה: התור אינו ריק
        Node<T> p = this.data.getFirst();
        while (p.getNext() != null )
                p = p.getNext();
        return p;
}
```

הסיבוכיות של הפעולה הזו היא O(n) שכן יש צורך לעבור על כל החוליות. לכן ההכנסה לתור תתבצע בסיבוכיות של O(n).

על מנת להקטין את הסיבוכיות של ההכנסה, ניתן להגדיר לתור תכונה נוספת. תכונה זו תכיל הפניה למקום האחרון ברשימה. הפניה זו תתעדכן בעת ההכנסה של איברים לסוף התור. כיון שלעולם מוציאים איברים רק מראש התור קל לשמור על ערך נכון של תכונה זו. הגדרת תכונה זו תוריד את סיבוכיות פעולת ההכנסה לכדי $O(1)$, כיוון שאין צורך לחפש את החוליה האחרונה.

```
public class Queue<T>
{
        private List<T> data;
        private Node<T> last;

        public Queue()
        {
                this.data = new List<T>();
                this.last = null;
        }

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

        public void insert (T x)
        {
                this.last = this.data.insert (this.last, x);
        }

        public T remove()
        {   //      הנחה: התור אינו ריק
                Node<T> first = this.data.getFirst();
                T x = first.getInfo();
                if (first = = last){   // התור התרוקן עם הוצאת האיבר האחרון
                        this.data.remove (first);
                        last = null;
                }
                else   // last is unaffected
                        this.data.remove (first);
                return x;
        }

        public int head()
        {    //  הנחה: התור אינו ריק
                return this.data.getFirst().getInfo();
        }

        public String toString()
        {
                … // השלימו בעצמכם!!
        }

}
```

ז. חוליות בעלות שתי הפניות

החידוש של שימוש בהפניות בייצוג אוספי נתונים, יכול להתרחב ולאפשר לנו גמישות רבה יותר. לעיתים נמצא כמועילה את האפשרות לטייל על הרשימה לא רק בכיוון אחד, ליניארי, אלא בשני הכיוונים באופן דומה. לצורך כך אנו יכולים להגדיר שחוליה בשרשרת החוליות תכיל שתי הפניות: האחת תצביע "קדימה" אל החוליה העוקבת והאחרת תצביע "אחורה" אל החוליה שקדמה לנוכחית. פעולת הקודם ברשימה תהפוך באופן שכזה לפעולה יעילה שסיבוכיותה $O(1)$, בהינתן מקום ברשימה (לעומת $O(n)$ באופן הייצוג הקודם).

רשימה שחוליותיה מכילות שתי הפניות שכאלה תקרא "רשימה דו כיוונית". בתרגול המצורף לפרק תתבקשו לממש רשימה שכזו.

ח. הרשימה - טיפוס נתונים מופשט?

הבה נסכם את תכונותיה של הרשימה ואת הפעולות שלה, תוך הדגשת הקשרים בין הפעולות וההגבלות על הפעולות השונות, לבין התכונות.

תיאור הרשימה:

1. הרשימה היא אוסף של ערכים (או עצמים); האוסף יכול להיות ריק (כמוגדר בנקודה 4 להלן).
2. קיים סדר ליניארי על האוסף, כאשר הוא אינו ריק. קריאת אברי הרשימה בסדר זה ניתנת לביצוע על ידי שימוש בשלוש פעולות: האחת מחזירה את המקום הראשון ברשימה, השנייה מאפשרת לעבור ממקום שאינו המקום האחרון לעוקב לו בסדר, והשלישית מאפשרת לקרוא את הערך המאוחסן במקום נתון.
3. לרשימה פעולות נוספות:

   פעולה לשינוי ערך המאוחסן במקום נתון.

   פעולה להוספת ערך לרשימה במקום נתון (ליתר דיוק אחרי מקום נתון) וכמקרה חריג גם לפני המקום הראשון.

   פעולה להוצאת ערך ממקום נתון ברשימה.

   פעולות אלו משמרות את תכונה 2. אחרי ביצוע עדכון: שינוי, הכנסה, או הוצאה, האוסף משקף את העדכון, ועדיין קיים סדר ליניארי על אברי הרשימה, המשקף את הפעולה שבוצעה.
4. הרשימה ריקה בדיוק כאשר כל ערך שהוכנס אליה הוצא ממנה (זה כולל את המקרה שעדיין לא הוכנס אף ערך).
5. פעולות השינוי וההוצאה אינן מוגדרות כאשר הרשימה ריקה.

נבהיר את הכוונה בנקודה השלישית. נניח שברשימה שלושה איברים המכילים את המספרים 3, 5, 11, בסדר זה. אם הכנסנו את המספר 7 אחרי המקום בו נמצא 5, אזי סקירה של הרשימה שהתקבלה תדפיס את הסדרה 3, 5, 7, 11. כך, תוכן הרשימה משקף את הוספת המספר 7, ומיקומו מתאים למקום בו הוכנס, שהוא אחרי מקומו של 5.

המימוש שבחרנו למחלקה רשימה נאמן להגדרה זו, במילים אחרות, זהו מימוש נכון.
ניווכח בזאת על ידי השוואה בין מימושי הפעולות לתיאור הפעולות והקשרים ביניהן.
ראשית, קל לראות כי דרישות 1, 2, 4, 5 מתקיימות מיד לאחר יצירת רשימה על ידי הפעולה הבונה.
במימוש שלנו קיימות הפעולות המנויות בנקודה 3. קל לראות, על ידי בחינה ישירה של הקוד המממש כל אחת מהן שהמימוש מקיים את הדרישה לגבי הפעולה.

הוכחנו את נכונות המימוש. כעת עלינו לבחון עוד נקודה חשובה והיא רמת ההסתרה של הייצוג שמספקת הגדרת המחלקה. לשם כך נעיין במספר דוגמאות.
דוגמה 1: צביקה יוצר רשימה של מספרים ששמה lst. בכוונתו להכניס אליה את המספרים הראשוניים, לפי סדרם. לאחר כמה פעולות הכנסה, הרשימה מכילה את המספרים 2, 3, 5, 11, בסדר זה. צביקה שם לב שהמספר 7 חסר בסדרה וכדי לתקן את המעוות הוא כותב את התוכנית הבאה:

```
void addSeven (lst)
{
    p = lst.getFirst();
    p = p.getNext();
    p =p.getNext();
    n = new Node (7);
    p.setNext (n);
    n.setNext (p.getNext());
}
```

לאחר הקריאה addSeven(lst), צביקה משוכנע שהבעיה נפתרה. ליתר בטחון, הוא מפעיל את תכנית הסקירה כדי שתציג את הערכים ברשימה על המסך, אך מה מתקבל?

דוגמה 2: כיון שבפעולת המחיקה יש צורך בפעולה המחזירה את המקום הקודם למקום נתון (בתנאי שאינו הראשון) הוחלט לממשה כפעולה פנימית. כיון שמימושה יקר, הוחלט לממש מחדש את החוליה כחוליה דו-כיוונית, וכמובן לממש מחדש את כל פעולות הרשימה על בסיס החוליה החדשה. בינתיים, תיקן צביקה את הטעות שהיתה בתוכניתו על ידי החלפת סדר שתי הפעולות האחרונות. אך במחלקה החדשה של הרשימה, תכניתו שוב אינה נכונה, מדוע?

אמנם המימוש שלנו לרשימה ופעולותיה נכון – הפעולות כולן שומרות על מבנה הרשימה, אך במימוש שלנו אין הסתרה. המימוש חשוף ומתכנת המשתמש ברשימה יכול להגיע אליו ולהשתמש בו ישירות, כפי שעשה צביקה. יתכן שמתכנת כזה טועה בשוגג, וכמובן שיש מצבים של טעות במזיד. בכל מקרה, תכנות ישיר על המימוש יכול לפגוע בתכונות הרשימה: ניתן לאבד חלק מאברי

הרשימה, ניתן להפוך את הרשימה או חלק ממנה למעגל, וכך הלאה. כמו כן, אם יוחלף בעתיד מימוש הרשימה במימוש אחר, תכניות המשתמשות ישירות במימוש לא יהיו נכונות.

מסקנה:
הרשימה אינה טיפוס נתונים מופשט! הרשימה היא מבנה נתונים, מועיל ומעניין, אך רק מבנה נתונים, ולא טיפוס מופשט. שמירה על תכונות הרשימה, על ידי כתיבת תכניות המשתמשות רק בפעולות הממשק, אפשרית כמובן, אך היא עניין של הסכמה ורצון טוב, ואינה נכפית על ידי השפה. ניתן להשתמש ברשימה לייצוג טיפוסי נתונים אחרים, שכן אז היא מוסתרת כתכונה פנימית של הטיפוס אותו היא ממשת. שימוש ישיר בה יש בו סכנה.

ט. סיכום

- בפרק הזה הכרנו מבנה נתונים בשם **רשימה**. רשימה היא אוסף נתונים שאינו מוגבל בגודל, המאורגן כסדרה. ניתן להכניס איברים לכל מקום ולהוציא איברים מכל מקום ברשימה.
- ניתן לייצג את הרשימה באופנים שונים. אנו הצגנו בפרק את הייצוג המקובל באמצעות שרשרת חוליות. רשימה הממומשת באמצעות חוליות נקראת **רשימה מקושרת**.
- למחלקת החוליה יש שתי תכונות : **info** (ערך)– שיכול להיות מכל טיפוס (גם פשוט וגם מטיפוס מחלקה כלשהי) ו-**next** (עוקב) – שהוא הפניה לחוליה הבאה ברשימה.
- ניתן להגדיר את הרשימה כמחלקה גנרית, אז טיפוס הערך הוא טיפוס גנרי T, ובכל שימוש ניתן להחליפו בטיפוס מחלקה, אך לא בטיפוס פשוט. מחלקות עוטפות מאפשרות שימוש לא מוגבל ברשימה גנרית גם עבור טיפוסים פשוטים.
- למחלקת הרשימה יש תכונה אחת : first- והיא הפניה לחוליה הראשונה ברשימה.
- הממשק של **מבנה הנתונים** רשימה נוצר מצירוף הממשקים של **חוליה** ו**רשימה** גם יחד.

ממשק החוליה :

Node (T x)	הפעולה בונה חוליה. הערך של החוליה הוא x, ואין לה חוליה עוקבת
Node (T x, Node<T> p)	הפעולה בונה חוליה. הערך של החוליה הוא x והחוליה העוקבת לה היא p
T getInfo()	הפעולה מחזירה את הערך של החוליה
Node<T> getNext()	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה תחזיר **null**
void setInfo (T x)	הפעולה משנה את ערך החוליה להיות x

void setNext (Node<T> p)	הפעולה משנה את החוליה העוקבת להיות p
String toString()	הפעולה מחזירה מחרוזת המתארת את החוליה

ממשק הרשימה:

List()	הפעולה בונה רשימה ריקה
Node<T> getFirst()	הפעולה מחזירה את המקום של החוליה הראשונה ברשימה הנוכחית. אם הרשימה ריקה, הפעולה תחזיר **null**
Node<T> insert (Node<T> p, T x)	הפעולה מכניסה לרשימה הנוכחית את האיבר x מקום אחד *אחרי* המקום p. הפעולה מחזירה את המקום של החוליה החדשה שהוכנסה. על מנת להכניס חוליה למקום הראשון ברשימה יש לשלוח **null** כפרמטר המקום. **הנחה**: p הוא מקום קיים ברשימה הנוכחית
Node<T> remove (Node<T> p)	הפעולה מוציאה מהרשימה הנוכחית את האיבר הנמצא בה במקום p. מחזירה את המקום העוקב ל- p. אם הוצא האיבר האחרון – יוחזר **null**. **הנחות**: הרשימה אינה ריקה. p הוא מקום קיים (שאינו **null**) ברשימה הנוכחית
boolean isEmpty()	הפעולה מחזירה 'אמת' אם הרשימה הנוכחית ריקה, ו'שקר' אחרת
String toString()	הפעולה מחזירה מחרוזת המתארת את הרשימה

- מעבר על רשימה או סריקת רשימה נעשה באמצעות הפניה מטיפוס Node שניתן לשים בכל מקום ברשימה ולקדם באמצעות הפעולה getNext().
- ניתן להוסיף למחלקת הרשימה פעולות ממשק (פנימיות) נוספות. כאשר הרשימה גנרית, ניתן להוסיף רק כאלה שאינן משתמשות בפעולות על הערכים שברשימה (פרט להחזרתם כערך חזרה).
- ניתן להסתכל על מחסנית ועל תור כסוגים מיוחדים של רשימות, שבהן יש הגבלות על דרך ההכנסה של איברים ודרך הוצאתם. מדרך הסתכלות זו נגזרים ייצוגים של טיפוסים אלו על ידי שימוש ברשימות, בכך הדגמנו שימוש במבנה נתונים לצורך ייצוג של טיפוס נתונים.

מושגים

חוליה	Node
עוקב	next
ערך	info
ראש רשימה	list first
רשימה	List
רשימה מקושרת	LinkedList
תור	Queue

תרגילים

א. פעולות חיצוניות

1. כתבו פעולה המקבלת רשימה של מספרים שלמים ומחזירה את סכום ערכי החוליות במקומות הזוגיים.

2. א. כתבו אלגוריתם המקבל רשימה lst של מחרוזות ומחזיר 'אמת' אם הרשימה ממוינת לקסיקוגרפית ו-'שקר' אחרת.
   ב. ממשו את האלגוריתם מסעיף א כפעולה. (חפשו פעולה המשווה שתי מחרוזות בתוך המחלקה String של Java).

3. כתבו פעולה המקבלת רשימה של תווים ומנפה **אותה** מערכים חוזרים. כלומר, הרשימה הסופית תכיל רק ערך אחד מכל המופעים החוזרים.

4. כתבו פעולה המקבלת רשימה של נקודות (Point) ומחזירה הדפסה של כל הנקודות שסכום הקואורדינטות שלהן אינו גדול מעשרים.

5. הנקודות (Points) השמורות ברשימה נתונה הן נקודות על גרף רציף.
   i. כתבו פעולה המחזירה את הנקודה הנמצאת בראש הגרף (הנקודה הגבוהה ביותר על פני העקומה). הניחו שיש רק נקודה אחת שכזו.
   ii. העבירו את רשימת הנקודות לצב מטיפוס Turtle והפעילו אותו כך שיצייר על המסך את כל הגרף השמור ברשימת הנקודות.

6. א. כתבו אלגוריתם המקבל שתי רשימות של מספרים שלמים, ממוינות בסדר עולה, וממזג אותן לרשימה אחת חדשה, שגם היא ממוינת בסדר עולה. האלגוריתם יחזיר את הרשימה החדשה.
   ב. ממשו את האלגוריתם כפעולה.

7. א. כתבו אלגוריתם המקבל שתי רשימות של מספרים שלמים ומחזיר רשימה חדשה הנבנית באופן הבא: ערך של כל חוליה במקום ה–n ברשימה החדשה הוא סכום הערכים בחוליות שבמקום ה-n ברשימות הנתונות. אם החל ממקום מסוים נותרו איברים ברשימה אחת בלבד הם יועתקו לרשימה החדשה.
   לדוגמה:
   נתונות שתי רשימות:

L1: 3, 6, 8, 1, -5, 3

L2: -4, 6, 2, 76

האלגוריתם יחזיר את הרשימה:

L3: -1, 12, 10, 77, -5, 3

ב. ממשו את הפעולה.

8. **פעולה בונה מעתיקה** היא פעולה המקבלת עצם כלשהו ומחזירה עצם חדש מאותו טיפוס המכיל בדיוק אותם ערכים פנימיים.

א. כתבו פעולה בונה מעתיקה המקבלת חוליה של שלמים ומחזירה חוליה חדשה בעלת אותו תוכן בדיוק.

ב. הסבירו מדוע לא נוכל לכתוב את הפעולה כפעולה פנימית במחלקה Node.

9. בדוגמה המופיעה בפרק, בסעיף **ו.4**. (רשימות עם טיפוסים שונים), ראינו את פעולת הכיווץ על רשימה. כתבו פעולה בשם unZip המבצעת את התהליך ההפוך (פריסת רשימה): הפעולה תקבל רשימה מכווצת ותחזיר את הרשימה המקורית שבה מופיעות חוליות בעלות אותו תוכן על פי מספר הפעמים הנדרש.

ב. מעקב

10. לפניכם פעולה:

```
/* טענת כניסה: הפעולה מקבלת שתי שרשרות של חוליות nd1 ו-nd2 של מספרים שלמים */
/* טענת יציאה: ???? */

public static Node<Integer> change (Node<Integer> nd1, Node<Integer> nd2)
{
        Node<Integer> first;
        Node<Integer> p = nd1;
        if (p == null)
                first = nd2;
        else
        {
                while (p.getNext() != null)
                        p = p.getNext();
                p.setNext (nd2);
                first = nd1;
        }
        return first;
}
```

א. איך יראו שתי השרשרות בתום ביצוע הפעולה? הדגימו על שתי שרשרות.

ב. מהי מטרת הפעולה? השלימו את טענת היציאה של הפעולה.

ג. מהי סיבוכיות זמן הריצה של הפעולה על פי הייצוג המוזכר בפרק?

11. לפניכם פעולה פנימית של רשימה:

*{**טענת כניסה**: ...}*

*{**טענת יציאה**: הפעולה מחזירה 1 אם..., 2 אם...}*

```
public int secret()
{
	Node<T> p = this.getFirst();
	int num = 2;
	while (p != null)
	{
		num = 3 - num;
		p = p.getNext();
	}
	return num;
}
```

א. תנו דוגמה לרשימה עבורה יוחזר הערך 1, ולרשימה עבורה יוחזר הערך 2.

ב. השלימו את טענות הכניסה והיציאה של הפעולה.

12. נתונה הפעולה הבאה המוגדרת במחלקה ListUtil:

```
public static boolean secret (Node<Integer> p)
{
	int x, y;
	if (p.getNext() == null)
		return true;
	else
	{
		x = p.getInfo();
		y = p.getNext().getInfo();
	}
	if (x*y > 0)
		return false;
	else
		return secret (p.getNext());
}
```

א. נתונה רשימה lst של מספרים שלמים. מזמנים את הפעולה:

```
boolean answer = ListUtil.secret (lst.getFirst())
```

תנו דוגמה עבורה יוחזר ל-answer ערך **true** ודוגמה עבורה יוחזר ל-answer ערך **false**.

ב. השלימו את תיעוד הפעולה כך שיוסבר מה היא עושה. ציינו במפורש בתיעוד מהי הנחת היסוד העומדת בבסיס הגדרת הפעולה.

ג. פעולות פנימיות

13. הפעולה getPrev(...) (אחזר קודם ברשימה) מקבלת מקום של חוליה ברשימה ומחזירה את המקום של החוליה הקודמת לה ברשימה.
    א. כתבו פעולה פנימית getPrev(...) (אחזר קודם ברשימה), במחלקה רשימה.
    ב. אילו הנחות עומדות בבסיס ההגדרה של הפעולה?
    ג. ממשו מחדש את הפעולה remove של הרשימה בעזרת הפעולה getPrev(...).
    ד. מדוע הפעולה getPrev(...) היא פעולה של המחלקה רשימה ולא של המחלקה חוליה, למרות שלכאורה היא נראית כפעולה סימטרית של פעולת החוליה getNext()?

14. הפעולה getLast() מחזירה את המקום של האיבר האחרון ברשימה.
    א. כתבו את הפעולה getLast() כפעולה פנימית של מחלקת הרשימה.
    ב. האם ניתן לשנות את הייצוג של הרשימה, על מנת שסיבוכיות הפעולה תהיה O(1). אם כן ייצגו את הרשימה מחדש והסבירו את השינויים.
    ג. כתבו את הפעולה כפעולה חיצונית על פי הייצוג של רשימה כפי שהוא מופיע בפרק.

15. כתבו במחלקה רשימה את הפעולה replace(…), המקבלת מקום ברשימה ומחליפה בין החוליה הנמצאת במקום זה לחוליה העוקבת לה. אם אין חוליה עוקבת, הפעולה לא תבצע דבר.

16. כתבו פעולה פנימית למחלקה, בשם reverse(), המחזירה רשימה חדשה שבה סדר החוליות הפוך מהסדר ברשימה הנוכחית.

17. לעתים רוצים לממש רשימה שנוח לטייל בה לשני הכיוונים (גם הלוך וגם חזור). רשימה כזו נקראת **רשימה דו-כיוונית**. לצורך כך יש להגדיר בחוליה שתי הפניות: אחת לחוליה הבאה, ואחת לחוליה הקודמת.
    א. ייצגו את החוליה החדשה.
    ב. ממשו את המחלקה חוליה על פי הייצוג החדש.
    ג. ממשו את הפעולה Node<T> insert (Node<T> p, T x) של המחלקה: רשימה דו-כיוונית.
    ד. אילו פעולות נוספות של המחלקה רשימה יש לממש מחדש במחלקה DoubleLinkedList.

18. הפעולה insert

ברצוננו לשנות את פעולה ההכנסה לרשימה המופיעה בממשק המחלקה. במקום פעולת הכנסה אחת המשמשת להכנסה במקום כלשהו ברשימה, נגדיר שתי פעולות הכנסה: האחת

להכנסה במקום הראשון ברשימה, והאחרת להכנסה למקום כלשהו ברשימה שאינו המקום הראשון בה.

א. כתבו כותרות מתאימות לשתי הפעולות.

ב. ממשו את שתי פעולות ההכנסה.

ג. בנו רשימה ריקה והכניסו אליה את האיברים 1,2,3,4,5 באמצעות שתי הפעולות שמימשתם.

19. הפעולה remove

פעולת ההוצאה המוגדרת בממשק הרשימה בפרק, מקבלת מקום p ברשימה ומוציאה את האיבר הנמצא במקום זה:

Node<T> remove (Node<T> p)

נשנה את הגדרת הפעולה ונגדיר אותה כך:

הפעולה מוציאה את ה**עוקב** של האיבר הנמצא במקום p.

על מנת להוציא את האיבר הראשון, מותר להגדיר פעולה נוספת (removeFirst או לקבוע שכאשר הפעולה הקיימת מקבלת **null**, היא מוציאה את האיבר הראשון.

כתבו את פעולת ההוצאה החדשה. אם החלטתם על שתי פעולות נפרדות – כתבו את שתיהן.

ד. טיפוסי נתונים מופשטים

בשאלות הבאות תתבקשו להשתמש במבני הנתונים שהכרתם, לצורך ייצוגם של טנ"מ.

20. מחסנית

א. ייצגו את הטיפוס מחסנית באמצעות Node בלבד.

ב. ממשו את הפעולות של המחסנית בהתאם לייצוג של סעיף א.

ג. האם שינוי הייצוג והמימוש משנה את ממשק המחלקה מחסנית? פרטו והסבירו.

21. תור

א. ייצגו את הטיפוס תור באמצעות Node בלבד.

ב. ממשו את הפעולות של התור בהתאם לייצוג של סעיף א.

ג. אם על פי הייצוג שבחרתם בסעיף א, סיבוכיות פעולת ההכנסה לתור גדולה מ-O(1), בצעו שינוי בייצוג כך שהסיבוכיות תהיה בדיוק O(1).

22. קבוצה

קבוצה היא אוסף לא סדור של ערכים ללא חזרות (כלומר כל ערך מופיע פעם אחת בלבד ואין סדר מחייב בין הערכים). הכנסה של ערך שכבר קיים בקבוצה לא תשנה את הקבוצה. הקבוצה הריקה, המסומנת כ-{} היא קבוצה שלא מכילה ערכים.

לפניכם הממשק של הטיפוס "קבוצה של מספרים שלמים" IntSet:

IntSet()	הפעולה בונה קבוצה ריקה
void insert (**int** x)	הפעולה מכניסה לקבוצה את האיבר x, בתנאי שאינו נמצא בה. אם x נמצא בקבוצה הפעולה אינה מבצעת דבר
boolean exists (**int** x)	הפעולה מחזירה 'אמת' אם האיבר נמצא בקבוצה ו-'שקר' אחרת
void delete (**int** x)	הפעולה מוחקת את האיבר x מהקבוצה. אם x לא מופיע בקבוצה הפעולה אינה מבצעת דבר
IntSet unify (IntSet set)	הפעולה מקבלת כפרמטר קבוצה set ומחזירה את **קבוצת האיחוד** של set עם הקבוצה הנוכחית
IntSet intersect (IntSet set)	הפעולה מקבלת כפרמטר קבוצה set ומחזירה את **קבוצת החיתוך** של set עם הקבוצה הנוכחית

הערות:

1. **קבוצת איחוד** של שתי קבוצות set1 ו-set2 היא קבוצת כל האיברים המופיעים בקבוצה set1 או בקבוצה set2.
2. **קבוצת חיתוך** של שתי קבוצות set1 ו-set2 היא קבוצת האיברים המופיעים בקבוצה set1 *וגם* בקבוצה set2.

מה עליכם לעשות?

א. ייצגו את טיפוס הנתונים קבוצה. הסבירו את בחירתכם.

ב. ממשו את טיפוס הקבוצה על פי הייצוג שבחרתם בסעיף א.

ג. ציינו מהי הסיבוכיות של כל אחת מפעולות הממשק על פי הייצוג שבחרתם. הסבירו.

ד. השאלה עוסקת בקבוצות של מספרים שלמים. האם ניתן להכליל את ההגדרה לקבוצה גנרית כלשהי? נמקו.

23. אוסף ממוין SortedCollection

אוסף ממוין הוא אוסף של ערכים הממוינים בסדר עולה או יורד.

נגדיר את המחלקה **IntSortedCollection** המייצגת אוסף ממוין בסדר עולה של מספרים שלמים:

IntSortedCollection()	הפעולה בונה אוסף ממוין ריק
void insert (**int** x)	הפעולה מכניסה למקום המתאים באוסף הממוין את המספר x
boolean exists (**int** x)	הפעולה מחזירה 'אמת' אם המספר x נמצא באוסף הממוין ו-'שקר' אחרת
void delete (**int** x)	הפעולה מוחקת את המספר x מהאוסף הממוין. אם x לא קיים באוסף הפעולה לא תבצע דבר
int[] getAll()	הפעולה מחזירה מערך המכיל את כל אברי האוסף ממוינים בסדר עולה
int getMinimum()	הפעולה מחזירה את האיבר הקטן ביותר באוסף. **הנחה**: האוסף הממוין אינו ריק
int getNextValue (**int** x)	הפעולה מחזירה את האיבר העוקב לאיבר הנתון [**עוקב** במקרה הזה פירושו הבא אחריו מבחינת הגודל] **הנחה**: x קיים באוסף

מה עליכם לעשות?

א. ייצגו את טיפוס הנתונים אוסף ממוין. הסבירו את בחירתכם.

ב. ממשו את טיפוס הנתונים על פי הייצוג שבחרתם בסעיף א.

ג. ציינו מהי הסיבוכיות של כל אחת מפעולות הממשק על פי הייצוג שבחרתם. הסבירו.

ד. אם הסיבוכיות של הפעולה getMinimum() על פי הייצוג שבחרתם, אינה O(1), הציעו ייצוג אחר שיאפשר לבצע את הפעולה בסיבוכיות זו. הסבירו.

24. וקטור

וקטור הוא אוסף של ערכים שהגישה אל ערכיו נעשית באמצעות מקומם הסידורי באוסף.

לפניכם ממשק המחלקה Vector המכיל מספרים שלמים:

Vector()	הפעולה בונה וקטור ריק
void insert (**int** place, **int** x)	הפעולה מכניסה למקום place בוקטור את הערך x. **הנחה**: המקום place קיים בוקטור
boolean exists (**int** x)	הפעולה מחזירה 'אמת' אם x נמצא בוקטור ו-'שקר' אחרת
void delete (**int** x)	הפעולה מוחקת את האיבר x מהוקטור. אם x אינו קיים בוקטור הפעולה לא מבצעת דבר

מה עליכם לעשות?

א. ייצגו את המחלקה Vector. הסבירו.

ב. ממשו את הפעולה **void** insert(**int** place, **int** x) בהתאם לייצוג שבחרתם בסעיף א.

ג. מהי סיבוכיות זמן הריצה של הפעולה שמימשתם בסעיף ב.

ה. מבני נתונים

בשאלות הבאות תתבקשו להשתמש במבני הנתונים שהכרתם, לצורך ייצוג מבני נתונים אחרים.

25. ייצוג רשימה בעזרת מערך:

ניתן לייצג סדרה ליניארית באמצעות מערך גנרי. במקרה כזה המערך, המשמש לשמירת ערכי הרשימה, יהיה התכונה של הרשימה. המקום של איבר בייצוג שכזה אינו הפניה לחוליה אלא מספר שלם המציין את המקום של האיבר ברשימה.

```
public class List<T>
{
    private T[] arrList;
}
```

א. האם שינוי הייצוג של הרשימה משנה את הממשק? הסבירו ונמקו. האם אתם יכולים להסיק מכך משהו לגבי היות הרשימה טיפוס נתונים מופשט?

ב. אילו מפעולות הממשק של הרשימה יתמכו בשינוי הייצוג ואילו יצטרכו לעבור התאמה?

ג. ממשו את הפעולה הבונה של הרשימה על פי ייצוג זה. הגבילו את גודל הרשימה ל-30 כדי לחסוך את התכנות הקשור להגדלת המערך.

ד. ממשו את הפעולה **void** insert (**int** place, T info) המכניסה איבר בעל ערך info למקום place.

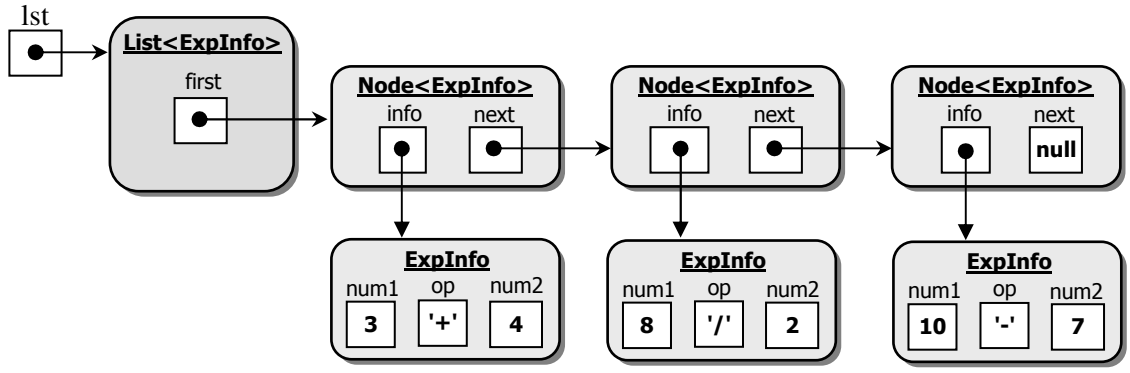
ו. שאלות כלליות

27. **הערה**: השאלה הבאה לקוחה מתוך בחינת הבגרות של קיץ 2005 והותאמה לרוח היחידה החדשה.

רשימה "חשבונית" היא רשימה שאיבריה מכילים ביטויים חשבוניים.
ביטוי חשבוני מורכב משלושה חלקים:

- מספר שלם גדול מאפס
- תו אחד מבין ארבעת התווים:
  - \+ המייצג חיבור
  - \- המייצג חיסור
  - * מייצג כפל
  - / מייצג חילוק
- מספר שלם גדול מאפס

דוגמה לרשימה "חשבונית" lst:



א. ייצגו את הביטוי החשבוני.

ב. הפעולה **public int** calculate() מחזירה את ערכו של ביטוי חשבוני.

i) היכן תוגדר פעולה זו? הסבירו.

ii) כתבו את הפעולה.

ג. נגדיר את הפעולה sumExpressions(…) המחזירה את הסכום הכולל של תוצאות הביטויים החשבוניים הנמצאים ברשימה. בעבור רשימה ריקה יוחזר 0.

i) היכן תוגדר הפעולה (באיזו מחלקה) – הסבירו.

ii) כתבו את הפעולה (הפעולה צריכה להשתמש בפעולה calculate שהגדרת בסעיף ב).

בעבור הרשימה ה"חשבונית" lst בדוגמה הנתונה, הפעולה ()sumExpressions תחזיר 14.

הערה: אין צורך לממש את פעולות הממשק של המחלקה רשימה.