פרק 9
רשימה
אוסף לינארי

בפרקים הקודמים הכרנו שני סוגי אוספים כלליים, מחסנית ותור. ראינו כי ההבדל ביניהם הוא בנוהל ההכנסה וההוצאה של האיברים: במחסנית האיברים הוכנסו והוצאו מצד אחד בלבד של המחסנית (ראש המחסנית), ובתור הוכנסו האיברים מצד אחד והוצאו מצדו האחר. המשותף למחסנית ולתור היה ההגבלה על הגישה לאיברי הסדרות השמורים בהם.

בפרק זה ברצוננו להגדיר אוסף כללי מסוג חדש – **רשימה (List)**. זהו אוסף סדרתי-לינארי, וניתן להכניס אליו ולהוציא ממנו איברים בכל מקום בסדרה ללא הגבלה. סוג אוסף זה שימושי ביישומים רבים שבהם יש צורך בניהול רשימות מסוגים שונים.

כדי לנהל רשימה נזדקק, בין היתר, לפעולות האלה: בנייה של רשימה ריקה ופעולות הכנסה והוצאה של ערכים, שיוכלו להתבצע בכל מקום ברשימה. אחת הפעולות הנפוצות שייעשו על רשימה היא סריקתה, וזאת כדי לאתר נתונים רצויים ולבצע עליהם פעולות נוספות (עדכון, אחזור, הוצאה וכדומה). כדי לבצע סריקה של רשימה יש להתקדם לאורך הרשימה החל באיבר הנמצא במקום הראשון בה.

לפני שנציג ממשק המסכם את הפעולות על רשימה, עלינו להגדיר מושג חדש: **מקום (position)**. פעולות הממשק מזכירות מושג זה ומשתמשות בו, ולכן עלינו להגדירו קודם להצגת הממשק. כדי לבחון את האפשרויות להגדרת המושג מקום, עלינו להקדים ולדון בייצוגים אפשריים לרשימה ובמשמעותו של מקום בכל אחד מייצוגים אלה.

א. ייצוג הרשימה

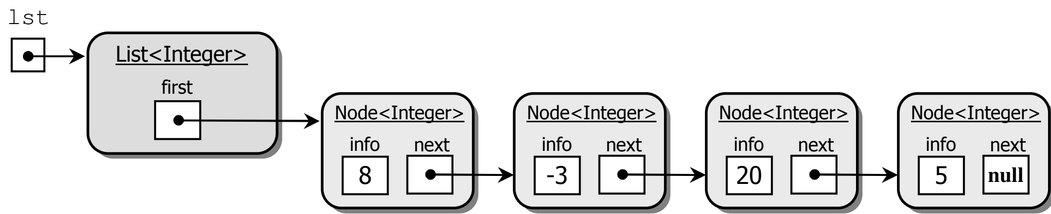
ניתן לייצג את הרשימה באמצעות מערך ולגשת אל האיברים באוסף לפי האינדקסים שלהם. ייצוג המקום כאן הוא אינדקס מספרי. כבר ראינו כי פעולות ההכנסה למערך וההוצאה ממערך מסורבלות ובלתי יעילות, ולכן נעדיף ייצוג אחר התומך בדינמיות של האוסף.

ייצוג כזה יתבסס על מבנה הנתונים המוכר: שרשרת חוליות. איברי הרשימה יאוחסנו בחוליות השרשרת. סדר האיברים יישמר על ידי הגדרת השרשרת, שבה כל חוליה מפנה אל העוקבת לה. ברשימה המיוצגת כך, המושג מקום של איבר הוא הפניה לחוליה המכילה אותו.

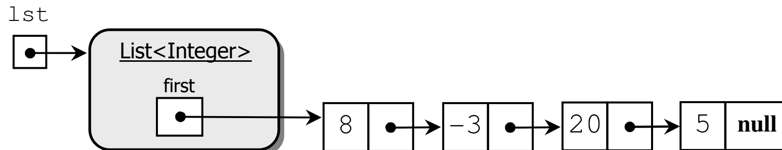
בגישה זו לייצוג, רשימה תיוצג באמצעות תכונה אחת שתחזיק את שרשרת החוליות, כלומר ערכה יהיה הפניה לחוליה הראשונה בשרשרת:

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

לדוגמה, סדרת המספרים [5, 20, -3, 8] תתואר באמצעות הרשימה lst:



או בצורה המופשטת יותר:



רשימה המיוצגת על ידי שרשרת חוליות נקראת **רשימה מקושרת (linked list)**. ייצוג זה ישמש אותנו בפרק זה.

ב. דיון בפעולות הרשימה

כדי לסרוק רשימה, כמו זו המתוארת באיור, יש להתחיל בחוליה הראשונה וממנה להתקדם חוליה אחר חוליה עד סוף הרשימה או עד המקום הרצוי. כלומר, אנו זקוקים לפעולה getFirst()‎ המחזירה את המקום הראשון ברשימה (הפנייה לחוליה הראשונה בשרשרת), ולפעולה שתאפשר להתקדם מחוליה נתונה לחוליה הבאה אחריה. את ההתקדמות ניתן לבצע בעזרת הפעולה getNext()‎ הקיימת בממשק החוליה, המוכר לנו.

הפעולה getNext()‎ היא פעולה על חוליה בודדת, ואולם רוב הפעולות שאנו זקוקים להן אינן מוגדרות על חוליה בודדת, וכמובן אינן קיימות בממשק המחלקה חוליה. פעולות אלה קשורות לניהול שרשרת חוליות שלמה. כזו היא הפעולה getFirst()‎ שהוזכרה לעיל. בפעולות אלה נכללות גם פעולת בנייה של רשימה ריקה, פעולה לבדיקת ריקנות של רשימה – isEmpty()‎, פעולת הכנסה – insert(…)‎, פעולת הוצאה – remove(…)‎, וכן פעולה המחזירה תיאור של הרשימה – toString()‎.
בתִכנות מונחה עצמים מקובל להגדיר מחלקה עבור כל סוג ישות שעובדים איתו, לכן נגדיר מחלקה List, שבה יוגדרו הפעולות העוסקות בניהול שרשרת החוליות. באמצעות מחלקה זו נוכל לבנות עצמים מטיפוס רשימה ולטפל בהם. כיוון שהפעולות אינן מבצעות על איברי הרשימה פעולות ייחודיות לטיפוס מסוים, המחלקה תהיה גנרית.

נציג את הממשק המסכם של המחלקה רשימה, הכולל את הפעולות שהוזכרו לעיל. בתיאור הממשק, משמעות המונח 'מקום' היא הפניה לחוליה.

ממשק המחלקה רשימה List<T>

המחלקה מגדירה אוסף סדרתי-לינארי שהגישה אל ערכיו מתבצעת בכל מקום באוסף.

הפעולה בונה רשימה ריקה	List()
הפעולה מחזירה 'אמת' אם הרשימה הנוכחית ריקה, ו'שקר' אחרת	**boolean** isEmpty()
הפעולה מחזירה את המקום של החוליה הראשונה ברשימה הנוכחית. אם הרשימה ריקה, הפעולה מחזירה **null**	Node<T> getFirst()
הפעולה מכניסה לרשימה הנוכחית את הערך x מקום אחד **אחרי** המקום pos. אם pos הוא **null**, x יוכנס למקום הראשון ברשימה. הפעולה מחזירה את המקום של החוליה החדשה שהוכנסה. **הנחה**: pos הוא מקום ברשימה הנוכחית או **null**	Node<T> insert (Node<T> pos, T x)
הפעולה מוציאה מהרשימה הנוכחית את האיבר הנמצא במקום pos, ומחזירה את המקום העוקב ל-pos. אם הוצא האיבר האחרון – יוחזר **null**. **הנחה**: pos הוא מקום ברשימה הנוכחית ואינו **null**.	Node<T> remove (Node<T> pos)
הפעולה מחזירה תיאור של הרשימה, כסדרה של ערכים, במבנה הזה (x_1 הוא האיבר הראשון ברשימה): [x_1, x_2, …, x_n]	String toString()

שימו לב, הפעולות בממשק המחלקה רשימה אינן מאפשרות כשלעצמן לבצע כל מה שאנו רוצים על רשימה. כדי לשלוף את הערך בחוליה או לשנותו, וכן כדי להתקדם מחוליה אחת לבאה אחריה, עלינו להשתמש בפעולות שבממשק החוליה. אם כן, סוג האוסף רשימה מוגדר על ידי שתי מחלקות: Node ו-List. אוסף הפעולות של שתי המחלקות **יחד** מאפשר לבצע את כל מה שנרצה לבצע על רשימה, כפי שיודגם להלן.

ג. שימוש בפעולות הממשקים

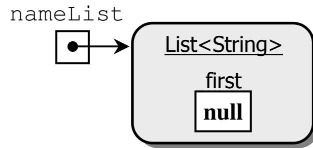
נדגים את פעולות ממשקי הרשימה והחוליה, תוך הסבר מפורט של הפעולות בממשק הרשימה.

ג.1. בניית רשימה

כמו בכל טיפוס גנרי, לפני שאנו בונים רשימה קונקרטית אנו חייבים להחליט מאיזה טיפוס יהיו איבריה.

ניצור רשימה של שמות, כלומר רשימה מטיפוס מחרוזת:

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



ג.2. הכנסת ערך לרשימה

כותרת פעולת הממשק נראית כך:

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

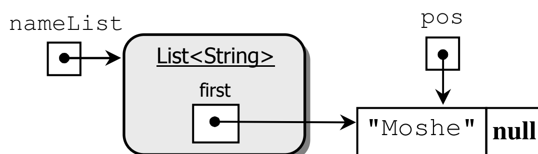
הפרמטר pos הוא מקום ברשימה שאחריו רוצים להכניס איבר חדש המכיל את הערך x. קיים מקרה קצה אחד והוא הכנסת איבר למקום הראשון ברשימה. כיוון שאין איברים לפני האיבר הראשון, אין מקום שאחריו תתבצע ההכנסה. למקרה קצה זה נגדיר שימוש ב-**null** כערך מיוחד ל-pos. כלומר, אם ערכו של pos הוא **null**, הכוונה היא שיש להכניס את האיבר במקום הראשון.

כיוון שפעמים רבות פעולת ההכנסה משולבת בפעולת הסריקה של הרשימה, ואנו מעוניינים להמשיך בסריקה ממקום ההכנסה, נקבע שפעולת ההכנסה תחזיר הפניה לחוליה שהוכנסה. כך ניתן להמשיך את הסריקה של הרשימה ממקום זה.

נראה כמה דוגמאות של הכנסה לרשימה.

דוגמה 1 – הכנסת המחרוזת "Moshe" למקום הראשון ברשימה

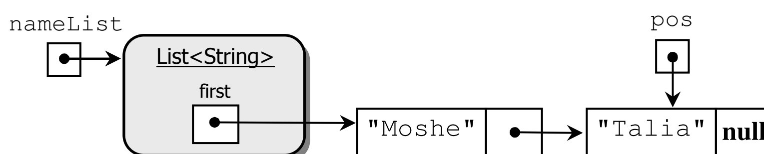
```
Node<String> pos = nameList.insert(null,"Moshe");
```



דוגמה 2 – הכנסת ערכים לרשימה

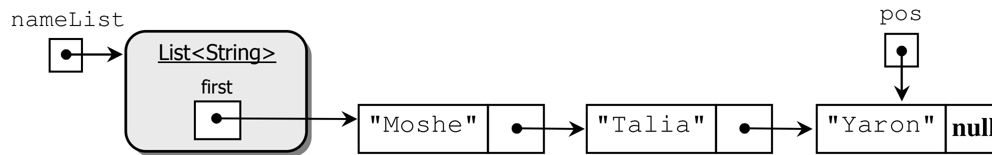
פעולת ההכנסה בסעיף 1 החזירה הפניה לאיבר שהוכנס. ביצענו השׂמה של הפניה זו למשתנה pos. כדי להכניס איבר למקום השני ברשימה, עלינו לשלוח את ההפניה pos כפרמטר להכנסה:

```
pos = nameList.insert(pos,"Talia");
```



ניתן להמשיך באופן זה ולהכניס ערכים נוספים:

```
pos = nameList.insert(pos,"Yaron");
```

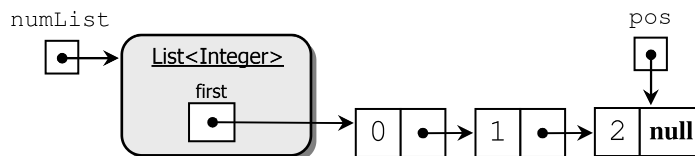


דוגמה 3 – הכנסת ערכים בלולאה

באמצעות לולאה אפשר להכניס לרשימה סדרה של ערכים. הלולאה שלפניכם מכניסה רצף של מספרים שלמים לרשימה ריקה של מספרים:

```
List<Integer> numList = new List<Integer>();
Node<Integer> pos = numList.getFirst();

for (int i=0; i<3; i++)
      pos = numList.insert(pos, i);
```

המשתנה pos מאותחל ל-**null**, כיוון שהפעולה ()getFirst המופעלת על רשימה ריקה מחזירה **null**. לכן בזימון הראשון של הפעולה (…)insert, הערך אפס יוכנס למקום הראשון. pos מתקדם ועכשיו הוא מפנה למקום הראשון ברשימה. הזימון השני מכניס את הערך 1 למקום השני, וכך הלאה.



ניתן להשתמש בלולאות כדי להוסיף לרשימה רצף המתקבל ממערך, מרשימה אחרת או מהקלט. דוגמאות יוצגו בהמשך הפרק.

? תארו את הרשימה המתקבלת מהפעלת הקוד שלפניכם על רשימת המספרים הריקה lst:

```
for (int i=0; i<3; i++)
      lst.insert(null, i);
```

ג.3. הוצאת ערך מרשימה

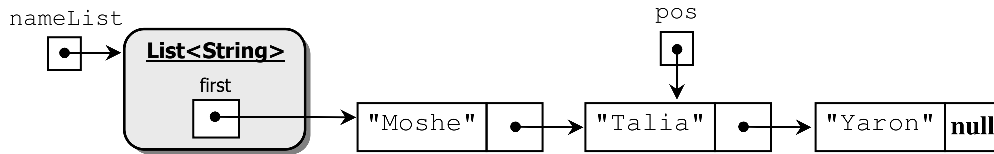
כותרת פעולת הממשק נראית כך:

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

pos הוא המקום של הערך שאנו רוצים להוציא מהרשימה. לעתים קרובות אנו מוציאים ערכים מהרשימה תוך סריקה, ומעוניינים להמשיך בה. הערך שהוצא כבר אינו ברשימה ולכן הפעולה מחזירה את המקום העוקב לו. ממקום זה ניתן להמשיך את הסריקה.

דוגמה: הוצאת ערכים מרשימה

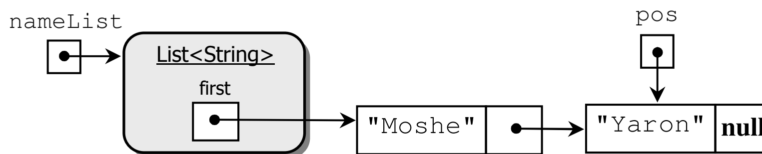
לפניכם מצב רשימת השמות ומשתנה שבו מאוחסן מקום לפני פעולת ההוצאה:



זימון פעולת ההוצאה כך:

```
Node<String> pos = nameList.remove(pos);
```

ישנה את מצב הרשימה והמשתנה, והם ייראו כך:



כפי שניתן לראות באיור, הערך הרצוי הוצא מהרשימה. pos התעדכן והוא נמצא במקום העוקב למקום ההוצאה.

ג.4. סריקה של רשימה

ניתן לבצע סריקה של רשימה מתחילתה (בעזרת getFirst()) או ממקום נתון כלשהו ברשימה.
סריקת הרשימה תיפסק באחד מהמקרים האלה:

א. הגענו לסוף הרשימה.

ב. מצאנו את הערך המקיים את התנאי הרצוי.

נראה כמה דוגמאות של סריקת רשימה.

דוגמה 1: סריקה עד סוף הרשימה

נגדיר פעולה פנימית למחלקה רשימה, המחזירה את גודל הרשימה הנוכחית (מספר האיברים בה):

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

      while (pos != null)
      {
            len++;
            pos = pos.getNext();
      }

      return len;
}
```

דוגמה 2: סריקה עד קיום תנאי רצוי

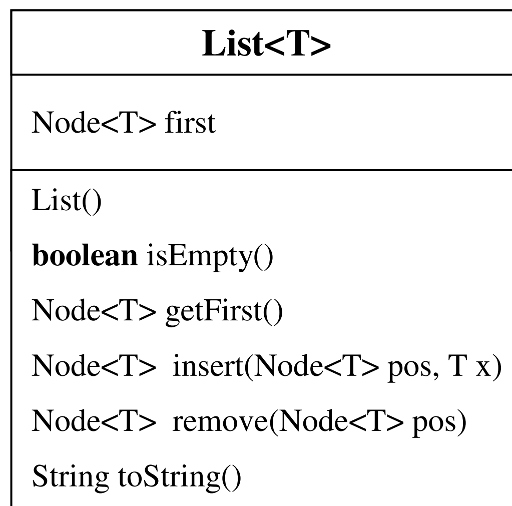
נגדיר פעולה חיצונית המקבלת כפרמטר רשימה וכן ערך x, ומחזירה את מקומו של x ברשימה. אם x אינו מופיע ברשימה, הפעולה תחזיר **null**:

```
public static Node<Integer> getPosition(List<Integer> lst, int x)
{
      Node<Integer> pos = lst.getFirst();

      while ((pos != null) && (pos.getInfo() != x))
            pos = pos.getNext();

      return pos;
}
```

? האם ניתן לכתוב את הפעולה getPosition(…) כפעולה פנימית?

ד. כתיבת המחלקה רשימה

החלטנו לייצג את המחלקה רשימה באמצעות שרשרת חוליות. להלן תרשים UML המתאר את המחלקה רשימה:



פעולה בונה

הפעולה הבונה מייצרת רשימה ריקה, שבה ערך התכונה first הוא **null**:

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

פעולה המחזירה את המקום הראשון ברשימה

הפעולה ()getFirst מחזירה את ערך התכונה first. אם הרשימה אינה ריקה תוחזר הפניה אל החוליה הראשונה ברשימה, אחרת יוחזר הערך **null**:

```
public Node<T> getFirst()
{
      return this.first;
}
```

פעולת הכנסה

פעולת זו מכניסה חוליה חדשה לשרשרת חוליות. דוגמה של הפעולה ראינו בפרק 7 בסעיף א.2.2. כעת עלינו לנסח פעולת הכנסה כללית: ערך החוליה יהיה x מטיפוס כלשהו וההכנסה תתבצע אחרי המקום pos.

בתהליך מימוש הפעולה יש להפריד בין שני מקרים: כאשר ערך הפרמטר pos הוא **null**, במקרה זה ההכנסה תתבצע למקום הראשון בשרשרת; בכל מקרה אחר, ההכנסה תתבצע אחרי המקום pos:

```
public Node<T> insert(Node<T> pos, T x)
{
      Node<T> temp = new Node<T>(x);

      if (pos == null)
      {
            temp.setNext(this.first);
            this.first = temp;
      }

      else
      {
            temp.setNext(pos.getNext());
            pos.setNext(temp);
      }

      return temp;
}
```

פעולת הוצאה

פעולת זו מוציאה חוליה קיימת משרשרת חוליות, כפי שראינו בפרק 7 בסעיף א.3.2. כעת עלינו לנסח פעולת הוצאה כללית: ההוצאה תתבצע על החוליה שבמקום pos, המתקבל כפרמטר, והמציין מקום קיים ברשימה הנוכחית, שאינו **null**. במימוש הפעולה יש להבחין בין שני מקרים: כאשר הפרמטר pos הוא המקום הראשון בשרשרת; בכל מקרה אחר, ההוצאה תתבצע על המקום pos שאינו ראשון בשרשרת:

```
public Node<T> remove(Node<T> pos)
{
      if(this.first == pos)
            this.first = pos.getNext();
      else
      {
            Node<T> prevPos = this.getFirst();
            while(prevPos.getNext() != pos)
                  prevPos = prevPos.getNext();
            prevPos.setNext(pos.getNext());
      }

      Node<T> nextPos = pos.getNext();
      pos.setNext(null);

      return nextPos;
}
```

כפי שניתן לראות ממימוש הפעולה, הטיפול במקרה הראשון (הוצאת החוליה הראשונה) הוא פשוט: אם pos שווה בערכו לתכונה first, כלומר שניהם מפנים לחוליה הראשונה, עלינו לעדכן את התכונה first כך שתכיל את ההפניה למקום העוקב של pos.

לעומת זאת, בכל מקרה אחר, הוצאת החוליה מהמקום pos שאינו הראשון מצריכה תהליך מורכב יותר. בשלב הראשון, יש לאתר את החוליה הקודמת ל-pos ולשמור אותה ב-prevPos. בשלב השני, נעדכן את ההפניה העוקבת של prevPos להיות ההפניה העוקבת של pos.

כדי שלא לשמור הפניות מיותרות לשרשרת החוליות, נסיים את פעולת ההוצאה בניתוק החוליה pos. נעדכן את ההפניה לעוקב של pos להיות **null**.

שימו לב שהחוליה שהוצאה מהרשימה אמנם מפסיקה להיות חלק מהרשימה, אך ייתכן שקיימת אליה הפניה חיצונית אחרת.

פעולה לתיאור הרשימה

כמו בכל טיפוס נתונים, יש להגדיר פעולה המחזירה מחרוזת המתארת את מופעי הטיפוס. לצורך כך יש להגדיר את הפעולה:

```
public String toString()
```

? השלימו את קוד הפעולה ()toString.

ה.יעילות פעולות הממשק

היעילות של כל פעולות הממשק של המחלקה List, פרט ל-()remove ו-()toString, היא O(1).

אם נוציא את האיבר הראשון באמצעות הפעולה (...)remove, אזי גם היעילות שלה תהיה O(1), כיוון שאין צורך לחפש את האיבר הקודם. כאשר n הוא מספר האיברים ברשימה, יעילות ההוצאה של האיבר האחרון היא מסדר גודל O(n), שכן יש לסרוק את כל הרשימה כדי למצוא את המקום הקודם למקום ההוצאה. יעילות הפעולה היא O(n), כיוון שאנו מחשבים יעילות לפי המקרה הגרוע.

ו. פעולות נוספות על רשימות

בסעיף זה נציג כמה פעולות על רשימה. פעולות אלה לא יתוספו לממשק המחלקה אלא יוגדרו כפעולות חיצוניות. הדוגמאות יחדדו נושאים שמן הראוי לתת עליהם את הדעת כאשר דנים ברשימות.

דוגמה 1: יצירת תת-רשימה ובה המספרים הזוגיים מרשימה נתונה

נתונה רשימה של מספרים שלמים. אנו רוצים לקבל תת-רשימה שתכיל את המספרים הזוגיים מתוך הרשימה הנתונה. עומדות לפנינו שתי אפשרויות לביצוע משימה זו: ניתן לשנות את הרשימה הנתונה ולמחוק ממנה את הערכים האי-זוגיים, או להשאיר את הרשימה המתקבלת ללא שינוי וליצור רשימה חדשה.

אם אנו מעוניינים לא לפגוע ברשימה הקיימת, נבחר באפשרות השנייה. נממש פעולה שמקבלת רשימת מספרים שלמים ומחזירה רשימה חדשה המכילה את המספרים הזוגיים מתוך הרשימה שהתקבלה:

```
public static List<Integer> getEvenList(List<Integer> lst)
{
      List<Integer> evenList = new List<Integer>();

      Node<Integer> pos1 = lst.getFirst();
      Node<Integer> pos2 = evenList.getFirst();

      while (pos1 != null)
      {
            if ((pos1.getInfo() % 2) == 0)
                  pos2 = evenList.insert(pos2, pos1.getInfo());
            pos1 = pos1.getNext();
      }

      return evenList;
}
```

הפעולה מחזירה רשימה חדשה ואינה משנה את מצבה של הרשימה שהתקבלה כפרמטר. יעילות הפעולה לינארית בגודל הרשימה lst.

פעולה זו מדגימה אופנים שונים לקידום הפניות המשתתפות בסריקת הרשימה. הקידום של pos1, שבאמצעותו סורקים את הרשימה המקורית, נעשה בעזרת הפעולה ()getNext. לעומת זאת, קידומו של pos2, המציין את מקום ההכנסה ברשימה החדשה, נעשה על ידי עדכונו בערך ההחזרה של פעולת ההכנסה. יש משמעויות שונות לאופן הקידום של ההפניות, ולרוב לא ניתן להחליף את אופני הקידום שלהן.

דוגמה 2: הכנסת ערכים לרשימה ממוינת

בדוגמה זו נכתוב פעולה המקבלת רשימת מחרוזות ממוינת בסדר אלפביתי, ומחרוזת נוספת. הפעולה תכניס את המחרוזת הנוספת למקום המתאים לה ברשימת המחרוזות. (לשם המחשה, חשבו על רשימת שמות אלפביתית שאליה רוצים להכניס שם חדש במקום המתאים). הפעם נניח כי אנו מעוניינים שהשינוי שמבצעת פעולת ההכנסה ייעשה על הרשימה עצמה. אי לכך, לפעולה לא יהיה ערך החזרה. פעולה זו תשמש אותנו כפעולת עזר בדוגמה השלישית.

בפרק 6 – הפניות ועצמים מורכבים, בסעיף ו.3., דנו בהכנסה של ערך לתוך אוסף ממוין. ראינו כי בשלב הראשון יש לבצע סריקה כדי למצוא את המקום שאליו צריך להכניס את הערך החדש. הסריקה מתבצעת כל עוד הערכים באוסף קטנים בערכם מהערך החדש. הסריקה מסתיימת אם הערך במקום הנוכחי גדול מהערך המיועד להכנסה. בפעולה שנכתוב עתה נשמור את המקום הקודם למקום ההכנסה במשתנה שייקרא prev. prev הוא פרמטר המקום לפעולת ההכנסה שמתבצעת במקום שאחריו. לפניכם הקוד המממש את ההכנסה של המחרוזת למקומה:

```
public static void insertIntoSortedList(List<String> lst,
                                                     String str)
{
      Node<String> prev = null;
      Node<String> pos = lst.getFirst();

      while ((pos != null) && (pos.getInfo().compareTo (str)<0))
      {
            prev = pos;
            pos = pos.getNext();
      }

      lst.insert(prev, str);
}
```

יעילות הפעולה לינארית בגודל הרשימה lst, מכיוון שבמקרה הגרוע המחרוזת str תוכנס לסוף הרשימה.

? א. הראו כי הפעולה תפעל כראוי בכל מקרי הקצה הקיימים.

ב. אילו שינויים יש לעשות בפעולה כך שתבצע את משימתה על רשימה של מספרים שלמים?

דוגמה 3: מיון רשימה באמצעות מיון הכנסה (insertion sort)

בדוגמה זו נממש פעולה שתמיין רשימה קיימת של מספרים שלמים בעזרת אלגוריתם שנקרא **מיון הכנסה (insertion sort)**. בפנינו שתי אפשרויות:

1. הפעולה תשאיר את הרשימה המקורית בצורתה, ותחזיר רשימה חדשה המכילה את הערכים המקוריים בסדר ממוין.
2. הפעולה תמיין את הרשימה המקורית ללא שימוש בזיכרון נוסף. מיון כזה נקרא **מיון בַּמקום (in place sort)**.

נתחיל באפשרות הראשונה:

נמיין את הרשימה המתקבלת בעזרת רשימה חדשה שאליה נכניס את הערכים המקוריים תוך שמירה על הסדר שלהם. הפעולה שנכתוב תמיין את רשימת השלמים ותשאיר את הרשימה שהתקבלה כפרמטר ללא שינוי. הפעולה תחזיר רשימה חדשה המכילה את הערכים המקוריים בסדר ממוין. נתחיל כאשר הרשימה החדשה ריקה. במצב זה היא בוודאי ממוינת. מכאן ואילך ניעזר בפעולת העזר insertIntoSortedList(…), שאותה הגדרנו בסעיף הקודם. הפעם תקבל הפעולה רשימה ממוינת של שלמים, ותכניס כל ערך נוסף מהרשימה המקורית למקום המתאים לו ברשימה הממוינת:

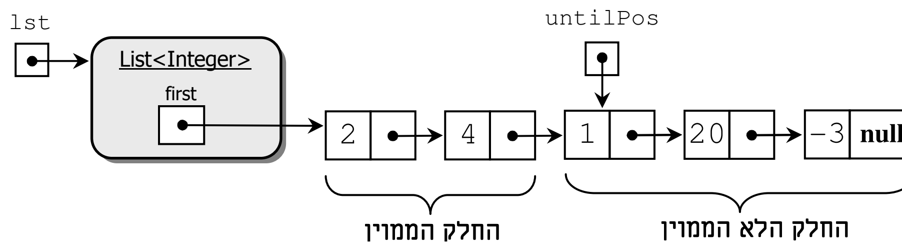
```
public static List<Integer> insertionSort(List<Integer> lst)
{
      List<Integer> sortedList = new List<Integer>();

      Node<Integer> pos = lst.getFirst();
      while (pos != null)
      {
            insertIntoSortedList(sortedList,pos.getInfo());
            pos = pos.getNext();
      }

      return sortedList;
}
```

יעילות הפעולה היא ריבועית בגודל הרשימה lst. הפעולה כוללת מעבר על כל איברי הרשימה, ומכניסה כל איבר לרשימה החדשה באמצעות פעולת העזר insertIntoSortedList(…), במחיר O(n), כפי שחישבנו לעיל.

נעבור לאפשרות השנייה, ונבצע מיון בַּמקום:

נסתכל על הרשימה המקורית כבעלת שני חלקים. החלק הראשון, בתחילת הרשימה, מכיל את הערכים הממוינים (בתחילת הפעולה חלק זה ריק – מספר הערכים בו הוא 0). החלק השני יכיל את שארית הרשימה שאותה עדיין יש למיין. בתחילת הפעולה חלק זה מכיל את הרשימה כולה, ובהמשך הוא הולך ומצטמצם עד שאין בו יותר איברים. בשלב זה כל הרשימה ממוינת. את מקום ההפרדה בין שני חלקי הרשימה נשמור במשתנה untilPos. החלק הממוין של הרשימה מתחיל בתחילת הרשימה ועד מקום אחד לפני untilPos. מ-untilPos ועד סוף הרשימה – זה החלק הלא ממוין.

לדוגמה, נתונה הרשימה lst=[4, 2, 1, 20, -3]. לאחר שני שלבים היא תיראה כך:



בכל שלב יש להכניס את הערך הראשון מהחלק הלא ממוין של הרשימה לחלק הממוין של הרשימה בעזרת פעולת עזר הנקראת insertIntoPartialSortedList(…). פעולה זו דומה לפעולה insertIntoSortedList(…). לאחר שלב זה הערך מופיע פעמיים ברשימה: במקומו החדש בחלק הממוין, ובמקומו המקורי. לכן יש לבצע פעולת הוצאה של האיבר המקורי מהחלק הלא ממוין.

להלן פעולה המבצעת מיון של רשימה באמצעות מיון הכנסה, תוך שינוי הרשימה המקורית:

```
public static void insertionSortInPlace(List<Integer> lst)
{
      Node<Integer> untilPos = lst.getFirst();

      while (untilPos !=  null)
      {
            insertIntoPartialSortedList(lst, untilPos);
            untilPos = lst.remove(untilPos);
      }
}

private static void insertIntoPartialSortedList(List<Integer> lst,
                                          Node<Integer> untilPos)
{
      Node<Integer> prev = null;
      Node<Integer> pos = lst.getFirst();

      while ((pos != untilPos) &&
                          (untilPos.getInfo()> pos.getInfo()))
      {
            prev = pos;
            pos = pos.getNext();
      }

      lst.insert(prev, untilPos.getInfo());
 }
```

? א. בפעולה insertIntoSortedList(…) (דוגמה 2) הפרמטר השני הוא ערך, ואילו כאן, בפעולה insertIntoPartialSortedList(…) הפרמטר השני הוא חוליה.

הסבירו מדוע בפעולה insertIntoPartialSortedList(…) יש צורך בפרמטר מטיפוס חוליה.

ב. מהי יעילותה של הפעולה insertionSortInPlace(…)?

בדוגמה האחרונה, מיון במקום, הצגנו רעיון תכנותי המסייע לנו כאשר אנו נדרשים לארגן מחדש אוספי נתונים. כאשר מבצעים קטע קוד המשנה סדר של אוסף נתון המועבר כפרמטר, ואופן ביצוע הפעולה אינו מצריך שימוש בשטח זיכרון שגודלו כגודל האוסף המקורי, נאמר כי התִכנות נעשה בַּמקום, in place. גישה תכנותית זו חוסכת במשאבי המקום בזיכרון. מעבר לכך, היא מועילה כיוון שהעצם המקורי משקף את השינוי והתהליך שהוא עבר. חישבו עד כמה תהיה עבודתנו נוחה, אם הרשימה הכיתתית תמשיך לשקף את המיון, גם לאחר כל פעולות ההכנסה אליה.

בכל בעיה שתעלה, יש לשקול האם תכנות בַּמקום יועיל והאם הוא נחוץ לפתרון הבעיה הנתונה.

ז. ייצוג אוספים באמצעות רשימה

רשימה משמשת בעיקר לייצוג אוספים. בעזרת רשימה ניתן לייצג אוספים ספציפיים הנדרשים ביישומים, אך גם סוגי אוספים כלליים, כפי שנראה בסעיפים הבאים.

ז.1. ייצוג אוסף ספציפי באמצעות רשימה

ניזכר במחלקה StudentList שהכרנו בפרקים הקודמים. כבר ראינו שני ייצוגים שונים למחלקה StudentList. הראשון באמצעות מערך, והשני באמצעות שרשרת חוליות. כעת נראה ייצוג נוסף באמצעות רשימה.

ממשק המחלקה StudentList

המחלקה מגדירה קבוצה של תלמידים הנקראת "רשימה כיתתית". תלמיד ברשימה מזוהה על פי שמו (אין בכיתה תלמידים בעלי שם זהה. אין צורך לבדוק האם קיים ברשימה מקום פנוי להכנסה):

הפעולה בונה רשימה כיתתית ריקה	StudentList()
הפעולה מוסיפה את התלמיד st לרשימה הכיתתית	**void** add (Student st)
הפעולה מוחקת את התלמיד ששמו name מתוך הרשימה הכיתתית. הפעולה מחזירה את התלמיד שנמחק. אם התלמיד אינו קיים ברשימה, הפעולה מחזירה **null**	Student del(String name)
הפעולה מחזירה את התלמיד ששמו name. אם התלמיד אינו קיים ברשימה, הפעולה מחזירה **null**	Student getStudent (String name)
הפעולה מחזירה מחרוזת המתארת דף קשר כיתתי הממוין בסדר אלפביתי, באופן זה: <name1> <tel1> <name2> <tel2>	String toString()

להלן הייצוג החדש של המחלקה ומימוש הפעולה הבונה:

```
public class StudentList
{
      private List<Student> lst;

      public StudentList()
      {
            this.lst = new List<Student>();
      }
      ...
}
```

פעולת ההוספה

ראינו כי פעולת ההוספה של תלמיד לרשימה הכיתתית יכולה להיעשות בשתי דרכים:

א. הוספת התלמיד למקום כלשהו ברשימה.

כדי להוסיף תלמיד לרשימה נוכל להשתמש בפעולת ההכנסה של הרשימה, ותמיד להכניס תלמיד למקום הראשון ברשימה.

```
public void add(Student st)
{
      this.lst.insert(null, st);
}
```

ב. הוספת התלמיד באופן ממוין למקום המתאים לפי הסדר האלפביתי.

כדי לממש את ההכנסה הממוינת לרשימה הכיתתית יש להיעזר בפעולת העזר insertIntoSortedList(...) שהגדרנו בסעיף ו בדוגמה 2.

? ממשו את המחלקה כך שפעולת ההוספה add(…) תוסיף את התלמיד למקומו הנכון לפי הסדר האלפביתי.

מה היתרון של שימוש ברשימה לייצוג רשימה כיתתית, לעומת השימוש בשרשרת חוליות? התשובה פשוטה. לאחר שהגדרנו את המחלקה רשימה ובה קבוצת פעולות שימושיות, נוח להשתמש במחלקה זו ובפעולותיה, במקום להשתמש במבנה נתונים (שרשרת חוליות), ולכתוב שוב את הפעולות שבהן אנו מעוניינים. זהו יישום של עקרון השימוש החוזר בקוד.

ז.2. ייצוג אוסף כללי באמצעות רשימה

ז.1.2. ייצוג מחסנית

בפרק הקודם ראינו כי ניתן לייצג מחסנית באמצעות שרשרת חוליות. באופן דומה ניתן לעשות זאת באמצעות רשימה, שתהיה התכונה היחידה במחלקה מחסנית.

מימוש הפעולה isEmpty() של המחסנית ייעשה בעזרת הפעולה isEmpty() של הרשימה, שהיא תכונת המחלקה. מימוש פעולות ההכנסה וההוצאה של מחסנית ייעשה באמצעות פעולות ההכנסה וההוצאה המתבצעות בתחילת הרשימה. נראה את המימוש כולו:

```
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()
      {
            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
      }

}
```

ז.2.2. ייצוג תור

נייצג את התור באמצעות רשימה, בדומה לייצוג המחסנית בסעיף הקודם. נגדיר את הרשימה כתכונה פרטית של התור, כך שמשתמש במחלקת התור לא יוכל לגשת אליה ישירות, 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. גישה לסוף הרשימה כדי להכניס איבר חדש היא מורכבת יותר, כיוון שאין ברשימה הפניה לסוף הרשימה. הגישה לסוף הרשימה תיעשה על ידי פעולת עזר פרטית שתוגדר במחלקה תור. הפעולה תסרוק כל פעם את הרשימה ותחזיר את ההפניה לסוף התור. יעילותה של פעולה זו תהיה מסדר גודל $O(n)$, שכן יש צורך לעבור על כל החוליות ברשימה. לפיכך, יעילותה של פעולת ההכנסה לתור תהיה $O(n)$.

? כתבו את פעולת העזר המתאימה, הסורקת את הרשימה ומחזירה את המקום האחרון בה.

כדי לשפר את יעילות פעולת ההכנסה, ניתן לשנות את ייצוג התור ולהגדיר בו תכונה נוספת. תכונה זו תכיל הפניה למקום האחרון ברשימה. הפניה זו תתעדכן בעת הכנסה של איברים לסוף התור. הגדרת תכונה זו תשפר את יעילות פעולת ההכנסה לכדי O(1), כיוון ששוב אין צורך לחפש את החוליה האחרונה, והפניה אליה היא ישירה.

לפניכם מוצג המימוש המלא של תור באמצעות רשימה והתכונה last :

```
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()
      {
            Node<T> first = this.data.getFirst();
            this.data.remove(first);
            if (first == this.last)
                              // עם הוצאת האיבר האחרון התרוקן התור
                  last = null;
            return first.getInfo();
      }

      public T head()
      {
            return this.data.getFirst().getInfo();
      }

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

      }

}
```

לפני סיום נדון עוד ביעילותה של הפעולה (...)remove של תור. על פי ניתוח פשוט של הפעולה, נראה שיעילותה היא מסדר גודל O(n). זאת כיוון שאנו פונים לפעולה (...)remove של הרשימה שיעילותה במקרה הגרוע היא O(n). אם נבחן היטב את הפעולה (...)remove של תור נראה שהיא מוציאה תמיד את האיבר שבראש התור, שהוא האיבר הראשון ברשימה המייצגת את התור. יעילותה של הוצאה מראש הרשימה היא O(1), כיוון שבמקרה זה איננו מחפשים את המקום שלפני מקום ההוצאה. לכן יעילות ההוצאה מתוך תור תהיה מסדר גודל O(1).

ז.3.2. סיכום: ייצוג באמצעות שרשרת חוליות או ייצוג באמצעות רשימה

הערה: ניתן להסתכל על מחסנית ועל תור כסוגים מיוחדים של רשימות, שעל דרך ההכנסה וההוצאה של איברים אליהם ומהם יש הגבלות. אם, ביישום שבו אנו זקוקים למחסנית, נשתמש ישירות ברשימה, אזי קיום ההגבלות יהיה תלוי ברצונו הטוב ובזיכרונו של מתכנת היישום. זו אינה גישה טובה. הגישה שהצגנו כאן היא ייצוג של טיפוסים אלה על ידי שימוש ברשימות. גישה זו עדיפה, שכן היא מונעת שימוש שגוי ביכולות הכלליות של רשימה, במקום ששימוש כזה אינו רצוי.

כאשר משתמשים ברשימה לייצוג בתוך מחלקה A, המממשת טיפוס נתונים מופשט כגון מחסנית או תור, הפעולות האפשריות על רשימה מוסתרות מהמשתמש. כך המשתמש יכול לבצע רק את הפעולות המוגדרות בממשק המחלקה A. הסתרה זו מתקיימת בגלל עקרון הסתרת המידע הקיים במחלקות. על פי עיקרון זה, אין גישה ישירה לתכונת המחלקה והפעולות שהיא מאפשרת (ובלבד שתוגדר כתכונה פרטית), אלא רק דרך הפעולות המוגדרות בממשק המחלקה שבה מוגדרת התכונה: הפעולות המוגדרות במחלקה Stack תומכות בפרוטוקול ה-LIFO, הפעולות המוגדרות במחלקה Queue תומכות בפרוטוקול ה-FIFO, ובשני המקרים פעולות הממשק אינן מאפשרות גישה חופשית לכל האיברים. כך אין אפשרות לבצע סריקה או הוספת חוליה בכל מקום, אף על פי שהייצוג נעשה באמצעות רשימה. כך הדבר גם לגבי השימוש ברשימה לייצוג תור. אין בכך כל פגם – כאשר הרשימה משמשת לייצוג של מחלקה אחרת, אותה המחלקה תחשוף בפני המשתמש את פעולותיה שלה, ולא את אלה של הרשימה.

כפי ששאלנו על רשימה כיתתית, נשאל גם כאן: מה היתרון של ייצוג מחסנית או של תור באמצעות רשימה, ולא באמצעות שרשרת חוליות?

התשובה ישימה לכל מחלקה, המממשת סוג אוסף כלשהו. אם מימשנו את מחלקה A באמצעות שרשרת חוליות, אין כל סיבה שלא נמשיך להשתמש בה. אבל אם עומדת לרשותנו המחלקה רשימה עם פעולותיה, ואנו מבקשים לכתוב את המחלקה A, נעדיף לייצגה באמצעות רשימה. בעשותנו זאת אנו משתמשים שוב בקוד שכבר כתבנו ובדקנו את נכונותו, ואין לנו צורך לכתוב קוד חדש. גם אם המחלקה החדשה A משתמשת רק בחלק קטן מהפעולות של הרשימה, אין בכך כל רע. עדיף להשתמש בקוד קיים ובדוק מאשר לכתוב קוד חדש. כל זה מותנה בכך שרשימה היא ייצוג מתאים למחלקה A.

ח. מימוש רקורסיבי של פעולות ברשימה

כפי שלמדנו בפרק ייצוג אוספים, ניתן לכתוב פעולות רקורסיביות על שרשרת חוליות. כיוון שבייצוג שאנו מציעים לרשימה בפרק זה התכונה של המחלקה היא שרשרת חוליות, ניתן להציע מימושים רקורסיביים לחלק מפעולות הממשק של הרשימה, וכן לפעולות פנימיות אחרות. כיוון שניתן להגיע לשרשרת החוליות גם מחוץ למחלקה רשימה, על ידי שימוש בפעולות כגון getFirst() ו-getNext(), ניתן לממש גם פעולות חיצוניות מסוימות באופן רקורסיבי. עם זאת, אין זה סביר להגדיר את הפעולות הדרושות לנו כפעולות על שרשרת החוליות, כיוון שמעצם הגדרתן הפעולות הן על רשימה ולא על הייצוג שלה. לכן, הגישה הכללית שאנו מציעים למימוש רקורסיבי של פעולה ברשימה היא: יש לכתוב תחילה פעולת עזר רקורסיבית לביצוע הפעולה על שרשרת החוליות. אחר כך יש לכתוב את הקוד לפעולה שבה אנו מעוניינים, ובו זימון לפעולת העזר הרקורסיבית.

נדגים את הגישה על ידי מימוש הפעולה size(), שהגדרנו בסעיף ג.4., כפעולה פנימית המחזירה את גודל הרשימה הנוכחית. הפעולה size() היא פעולה על רשימה, ולכן הסבירות שהיא תקבל את שרשרת החוליות המייצגת את הרשימה כפרמטר – נמוכה. כדי להציג מימוש רקורסיבי לפעולה נגדיר פעולת עזר במחלקה רשימה:

```
//     הנחה: node אינו שווה null
private int sizeHelp(Node<T> node)
{
      if (node.getNext() == null)
            return 1;
      return 1 + sizeHelp(node.getNext());
}
```

פעולה זו היא למעשה הכללה לפעולה גנרית של הפעולה:

```
getChainLength(Node<String> chain)
```

את הפעולה הזו הגדרנו בפרק אוספים, בסעיף ג. הפעולה מוגדרת כפעולה פרטית, כיוון שכל תפקידה הוא לשמש כפעולת עזר לפעולה size().

הפעולה size() של מחלקת רשימה תפעיל את פעולת העזר ותשלח אליה כפרמטר את החוליה הראשונה בשרשרת החוליות. לפעולה של הרשימה יש להוסיף בדיקה למקרה מיוחד של רשימה ריקה. בדיקה זו לא נדרשה כאשר חישבנו אורך של שרשרת חוליות כיוון ששרשרת חוליות לעולם אינה ריקה:

```
public int size()
{
      if (this.first == null)
            return 0;
      return sizeHelp(this.first);
}
```

נשים לב כי הפעולה ()size היא הפעולה היחידה המזמנת את (...)sizeHelp (כיוון שפעולת העזר פרטית ולכן אינה מוכרת מחוץ למחלקה), והיא תמיד מעבירה פרמטר שערכו אינו **null**. אם כך ההנחה של פעולת העזר מתקיימת תמיד. הנחה זו הכרחית לפעולה תקינה של פעולת העזר. אם יועבר אליה כפרמטר הערך **null** נקבל שגיאת זמן ריצה.

ניתן לממש גם פעולות חיצוניות באופן רקורסיבי, על פי אותה הגישה. הנה לדוגמה, מימוש של הפעולה (...)size כפעולה חיצונית המקבלת רשימה של מספרים.

```
    //    null הנחה: node אינו שווה
public static int sizeHelp(Node<Integer> node)
{
      if (node.getNext() == null)
            return 1;
      return 1 + sizeHelp(node.getNext());
}

public static int size(List<Integer> lst)
{
      if (lst.isEmpty())
            return 0;
      return sizeHelp(lst.getFirst());
}
```

גם כאן, אם הפעולה (...)size היא היחידה הקוראת ל-(...)sizeHelp, אזי ההנחה שערך הפרמטר אינו **null** מתקיימת. כיוון שפעולת העזר במקרה זה היא פעולה פומבית, יש חשש מסוים (אם כי מזערי), שהיא תזומן על ידי פעולה אחרת שתעביר אליה כפרמטר את **null**.

? הפעולה שלפניכם מומשה בסעיף ג.4. דוגמה 2.

```
public static Node<Integer> getPosition(List<Integer> lst, int x)
```

כתבו את הפעולה כפעולה רקורסיבית (מותר להגדיר פעולות עזר רקורסיביות).

ט. רשימה דו-כיוונית

כפי שראינו בפרק 7 – ייצוג אוספים, שרשרת חוליות יכולה להיות דו-כיוונית, כלומר כל חוליה בה תכיל הפניה שתצביע "קדימה" אל החוליה העוקבת, והפניה נוספת שתצביע "אחורה" אל החוליה הקודמת לה. רשימה המיוצגת באמצעות שרשרת חוליות דו-כיוונית נקראת "רשימה דו-כיוונית" (doubly linked list). בתרגול המצורף לפרק תתבקשו לממש רשימה שכזו.

אם מייצגים את הרשימה באמצעות שרשרת חוליות דו-כיוונית, ניתן לשפר את היעילות של פעולת ההוצאה שבממשק המחלקה, הפעולה (...)remove, בסדר גודל, מ-O(n) ל-O(1). כזכור, פעולה זו דורשת את מציאת האיבר הקודם לזה שמוציאים. במימוש שהצגנו לפעולה, דבר זה מתבצע על ידי סריקה הרשימה מתחילתה. במקרה של שרשרת חוליות דו-כיוונית מציאת איבר קודם בשרשרת מתבצעת ביעילות O(1) – עוברים מהחוליה הנוכחית לקודמתה. שימו לב כי

הפעולות האחרות תהפוכנה למורכבות יותר, כיוון שבכל שינוי שייעשה על הרשימה יהיה צורך לטפל בשתי ההפניות.

? איך תשתנה היעילות של הפעולה insertIntoSortedList(…) שראינו בדוגמה 2, בסעיף ו, כאשר הרשימה תיוצג באמצעות שרשרת חוליות דו-כיוונית? הסבירו.

י. הרשימה – טיפוס נתונים מופשט?

הגדרנו שתי תכונות מרכזיות המאפיינות טיפוס נתונים מופשט:

1. הממשק של טיפוס נתונים מופשט מסתיר את אופן הייצוג והמימוש של הטיפוס. כתוצאה מכך ניתן לשנות את הייצוג מבלי לשנות את הממשק.
2. פעולות הממשק של טיפוס נתונים מופשט שומרות על נכונותו של הטיפוס (מבחינת המבנה שלו והגישה אל איבריו). לא ניתן לקלקל את מבנה טיפוס הנתונים המופשט אגב שימוש בפעולותיו.

האם הרשימה, כפי שהגדרנו אותה בפרק זה, מקיימת את התכונות הללו וניתן להגדירה כטיפוס נתונים מופשט?

ייצגנו את הרשימה באמצעות שרשרת חוליות. ייצוג זה חשוף למשתמש, כיוון שהפעולה getFirst() מחזירה ערך מטיפוס Node – הפניה לחוליה הראשונה. יתרה מזו, הפעולות לטיפול ברשימות כוללות הן את פעולות המחלקה רשימה והן את פעולות המחלקה Node, כגון הפעולה getNext() המאפשרת לעבור מחוליה אחת לחוליה העוקבת לה, והפעולה getInfo() השולפת ערך מחוליה. כפי שהובהר על ידי הדוגמאות שהצגנו, אין אפשרות לוותר על פעולות המחלקה Node. כלומר טיפול ברשימה מתבצע גם על ידי פעולות החוליות המרכיבות אותה. ייצוג הרשימה אינו מוסתר, ולא ניתן לשנותו מבלי לשנות את הממשק של הרשימה.

כתוצאה ישירה מכך ניתן להרוס את הרשימה על ידי שימוש בפעולות החוליה. הדבר יכול לקרות בשגגה או במזיד. נדגים שימוש בפעולות החוליה הגורם לתוצאות לא רצויות לרשימה.

דוגמה 1: יעקב יוצר רשימה lst של מספרים. בכוונתו להכניס אליה את המספרים הראשוניים, לפי סדרם. לאחר כמה פעולות הכנסה, הרשימה נראית כך: lst = [2, 3, 7, 11]. יעקב שם לב שהמספר 5 חסר בסדרה וכדי לתקן את המעוות הוא כותב את קטע הקוד:

```
Node<Integer> pos = lst.getFirst();
pos = pos.getNext();
Node<Integer> n = new Node<Integer>(5);
pos.setNext(n);
n.setNext(pos.getNext());
```

לאחר הרצת קטע הקוד, יעקב משוכנע שהבעיה נפתרה וכעת הרשימה מכילה גם את המספר 5. ליתר ביטחון, הוא מוסיף את השורה: System.out.print(lst) לסוף הקוד, כדי שהערכים יוצגו ברשימה על המסך.

? בדקו מה יתקבל בהדפסה. מה מקור הבעיה?

דוגמה 2: כדי לשפר את היעילות של פעולת ההוצאה מרשימה, הוחלט לייצג את הרשימה כרשימה דו-כיוונית, וכמובן לממש מחדש את כל פעולות הרשימה בהתאם לייצוג החדש. בינתיים, תיקן יעקב את הטעות שהייתה בתוכניתו, על ידי החלפת סדר שתי הפעולות האחרונות. אך המחלקה החדשה של הרשימה הדו-כיוונית שכתב שוב אינה נכונה, מדוע?

המימוש שלנו לרשימה ולפעולותיה נכון – הפעולות כולן שומרות על מבנה הרשימה, אך במימוש שלנו אין הסתרה. המימוש חשוף ומתכנת המשתמש ברשימה יכול להגיע למימוש ולהשתמש בו ישירות, כפי שעשה יעקב. ייתכן שמתכנת כזה טועה בשוגג, וכמובן שיש מצבים של פגיעה במזיד. בכל מקרה, קוד הפועל ישירות על הייצוג הפנימי עלול לפגוע ברשימה: עלולים לאבד חלק מאיברי הרשימה, עלולים להפוך את הרשימה או חלק ממנה למעגל, וכן הלאה. כמו כן, אם בעתיד יוחלף ייצוג הרשימה בייצוג אחר, תוכניות המשתמשות ישירות בייצוג לא יהיו נכונות.

מסקנה: הרשימה היא טיפוס נתונים מועיל ומעניין, אך זכרו שהיא משמשת כמבנה נתונים בלבד, ולא טיפוס מופשט! שמירה על הרשימה על ידי כתיבת תוכניות המשתמשות רק בפעולות הממשק אפשרית כמובן, אך היא עניין של הסכמה ורצון טוב, ואינה נכפית על ידי מנגנוני השפה. תוכנית שאינה מקפידה על הסכמה זו עלולה לסכן את נכונות מבנה הרשימה.

שימו לב, אף על פי שהרשימה אינה טיפוס נתונים מופשט, היא עדיפה לצרכים שהועלו בפרק זה, לצורך ייצוג רשימות – על שרשרת חוליות – כיוון שהיא מחלקה העוטפת שרשרת חוליות. הרשימה מציעה פתרונות לכמה מהבעיות המתעוררות בשימוש בשרשרת חוליות. כך למשל רשימה יכולה להיות ריקה, בעוד שרשרת חוליות אינה יכולה להיות ריקה. כמו כן, ניתן להוסיף איבר לתחילת הרשימה, או להוציא את האיבר הראשון מהרשימה, מה שלא אפשרי לביצוע בשרשרת חוליות, כפי שראינו.

במה שונה הרשימה מהמחסנית והתור? מה מאפשר למחסנית ולתור להיות טיפוסי נתונים מופשטים? התשובה פשוטה למדי. גם מחסנית ותור הם אוספים לינאריים. אך, בניגוד לרשימה, אין צורך בייצוג של מקום בתור או ברשימה מחוץ למחלקות אלה, ואין צורך בהעברת מקום כפרמטר לפעולות המחסנית והתור. באוספים אלה פעולות הכנסה, הוצאה או שליפת ערך נעשים תמיד במקומות קבועים – בראש המחסנית או בראש התור וזנבו. הידע על המקום בו תתבצע פעולה נמצא בקוד של הפעולה, ואין צורך להעבירו כפרמטר. כיוון שכך, במחלקות אלה אין צורך לחשוף את הייצוג.

יא. סיכום

- בפרק זה הצגנו את סוג האוסף **רשימה**. רשימה היא אוסף לינארי גנרי של נתונים שאינו מוגבל בגודלו, והוא מאורגן כסדרה. ניתן להכניס איברים לכל מקום ולהוציא איברים מכל מקום ברשימה.
- ניתן לייצג את הרשימה באופנים שונים. אנו הצגנו בפרק את הייצוג באמצעות שרשרת חוליות. בייצוג זה למחלקת הרשימה יש תכונה אחת: first, שערכה הפניה לחוליה הראשונה ברשימה, או **null** כאשר הרשימה ריקה. רשימה הממומשת באמצעות חוליות נקראת **רשימה מקושרת**.
- אוסף הפעולות על רשימה כולל את הפעולות שבממשקים של המחלקות **חוליה** ו**רשימה** גם יחד.
- סריקת רשימה מקושרת נעשית באמצעות הפניה מטיפוס Node, שניתן לאתחל לכל מקום ברשימה ולקדם באמצעות הפעולה getNext(). בכל שלב, ניתן לאחזר את הערך שבחוליה הנוכחית על ידי שימוש בפעולה getInfo(), או לשנותו על ידי שימוש בפעולה setInfo(). כמו כן, ניתן להוסיף חוליה חדשה אחרי החוליה הנוכחית, או להוציא את החוליה הנוכחית מן הרשימה.
- ניתן להשתמש ברשימה לייצוג אוספים במגוון יישומים, וכן לייצוג של סוגי אוספים אחרים, כגון מחסנית או תור.

מושגים

מקום	position
רשימה	List
רשימה מקושרת	linked list

תרגילים

שאלה 1

כתבו את הפלט המתקבל לאחר ביצוע כל קוד:

א.

```
List<Integer> lst = new List<Integer>();
lst.insert(null, 10);
Node<Integer> pos = lst.insert(null, -5);
lst.insert(lst.getFirst(), 17);
pos = lst.insert(pos, 8);
lst.insert(null, pos.getInfo());
System.out.println(lst);
```

ב.

```
List<Integer> lst = new List<Integer>();
for (int i = 0; i < 4; i++)
      lst.insert(null, i*i);
System.out.println(lst);
```

ג.

```
List<Integer> lst = new List<Integer>();
Node<Integer> pos = lst.getFirst();
for (int i = 0; i < 4; i++)
      pos = lst.insert(pos, i*i);
System.out.println(lst);
```

שאלה 2

נתונה רשימת המספרים: lst = [6, -3, 7, -8, -5, 10, 14, -1]. איך תיראה הרשימה lst לאחר ביצוע הקוד שלפניכם:

```
Node<Integer> pos = lst.getFirst();
while (pos != null)
{
      int x = pos.getInfo();
      if (x%2 != 0)
            lst.remove(pos);
      else
            if (x < 0)
                  pos.setInfo(-x);
      pos = pos.getNext();
}
```

שאלה 3

רשמו את טענת היציאה של הפעולה:

```
// הנחה: הרשימה אינה ריקה //
public static String mystery(List<String> lst)
{
      Node<String> pos1 = lst.getFirst();
      Node<String> pos2 = pos1.getNext();

      while (pos2 != null)
      {
            if (pos2.getInfo().length() > pos1.getInfo().length())
                  pos1 = pos2;
            pos2 = pos2.getNext();
      }

      return pos1.getInfo();
}
```

שאלה 4

נתונה הפעולה:

```
public static List<Character> mystery(List<String> lst)
{
      Node<String> pos1 = lst.getFirst();
      List<Character> lstRes = new List<Character>();
      Node<Character> pos2 = null;

      while (pos1 != null)
      {
            pos2 = lstRes.insert(pos2, pos1.getInfo().charAt(0));
            pos1 = pos1.getNext();
      }

      return lstRes;
}
```

א. איך תיראה רשימת התווים שתוחזר לאחר זימון הפעולה mystery(...), עבור הרשימה:

[“Hello”, “world”, “here”, “I”, “am”]

ב. רשמו את טענת היציאה של הפעולה.

ג. נתחו את יעילות הפעולה.

שאלה 5

נתונה הפעולה הרקורסיבית:

```
public static boolean secret(List<Integer> lst, int x)
{
      boolean res;
      int tmp;

      if (lst.isEmpty())
            res = false;
      else
      {
            tmp = lst.getFirst().getInfo();
            lst.remove(lst.getFirst());
            res = (tmp == x) || secret(lst, x);
            lst.insert(null,tmp);
      }
      return res;
}
```

א. מה תחזיר הפעולה עבור הזימון secret(lst,10), עבור הרשימה: lst = [60, 13, 97, 15, 36] ?

ב. תנו דוגמה לזימון הפעולה secret(…) כך שתחזיר ערך שונה מזה שהוחזר בסעיף א, אך עבור אותה רשימה lst.

ג. האם לאחר ביצוע הפעולה secret(…), הרשימה שהועברה כפרמטר השתנתה? הסבירו.

ד. רשמו את טענת היציאה של הפעולה.

שאלה 6

נתונה הפעולה:

```
public static void what(List<Integer> lst)
{
      int x;
      Node<Integer> pos = lst.getFirst();

      while (pos != null)
      {
            x = pos.getInfo();
            pos = lst.remove(pos);
            lst.insert(null,x);
      }
}
```

א. רשמו את טענת היציאה של הפעולה.

ב. נתחו את יעילות הפעולה.

ג. ממשו את הפעולה מחדש כך שתשפר את יעילות הפעולה המקורית בסדר גודל.

ד. ממשו את הפעולה המקורית כפעולה פנימית למחלקה List<T> המופיעה בסעיף ד בפרק זה.

שאלה 7

כתבו פעולה המקבלת רשימת מספרים שלמים ומחזירה את סכום הערכים במקומות הזוגיים. הניחו שהרשימה המתקבלת אינה ריקה.

שאלה 8

כתבו פעולה המקבלת רשימת מחרוזות ומחזירה 'אמת' אם הרשימה ממוינת בסדר אלפביתי ו-'שקר' אחרת.

שאלה 9

כתבו פעולה המקבלת רשימה של תווים ומנפה **אותה** מערכים חוזרים. כלומר, הרשימה המתקבלת תעודכן כך שתכיל רק ערך אחד מכל המופעים החוזרים.

שאלה 10

א. כתבו פעולה המקבלת רשימה של נקודות (Point) ומדפיסה את כל הנקודות שסכום ערכי הקואורדינטות שלהן אינו גדול מעשרים.

ב. נניח שהנקודות שנשמרו ברשימה בסעיף א הן נקודות על גרף רציף. כתבו פעולה המחזירה את הנקודה הגבוהה ביותר בגרף המתקבל (זו שערך ה-y שלה הגדול ביותר). הניחו שיש רק נקודה אחת כזו.

שאלה 11

כתבו פעולה המקבלת שתי רשימות של מספרים שלמים ומחזירה רשימה חדשה הנבנית כך: ערך של כל חוליה במקום ה-n ברשימה החדשה הוא סכום הערכים בחוליות שבמקום ה-n ברשימות הנתונות. אם החל ממקום מסוים ייוותרו איברים ברשימה אחת בלבד הם יועתקו לרשימה החדשה.

לדוגמה:

נתונות שתי רשימות:

lst1 = [-3, 6, 8, 10, -5, 3]

lst2 = [-4, 6, 2, 76]

הפעולה תחזיר את הרשימה החדשה הזו: [-7, 12, 10, 86, -5, 3]

שאלה 12

כתבו פעולה המקבלת שתי רשימות של מספרים שלמים, ממוינות בסדר עולה, וממזגת אותן לרשימה אחת חדשה, שגם היא ממוינת בסדר עולה. הפעולה תחזיר את הרשימה החדשה.

שאלה 13

נוסיף את הפעולה שלפניכם כפעולה פנימית במחלקה List<T>:

```
public Node<T> getPrev (Node<T> pos)
```

הפעולה מקבלת מקום ברשימה ומחזירה את המקום של החוליה הקודמת ברשימה. אם pos הוא החוליה הראשונה יוחזר הערך **null**.

א. ממשו את הפעולה בתוך המחלקה List<T> המופיעה בסעיף ד בפרק זה.

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, אף שלכאורה היא נראית כפעולה סימטרית של פעולת החוליה ()getNext?

שאלה 14

נוסיף את הפעולה שלפניכם כפעולה פנימית במחלקה List<T>:

```
public Node<T> getLast()
```

הפעולה מחזירה את המקום של האיבר האחרון ברשימה. הנחה: הרשימה אינה ריקה.

א. ממשו את הפעולה בתוך המחלקה List<T> המופיעה בסעיף ד בפרק זה.

ב. האם ניתן לשנות את הייצוג של הרשימה, כך שסדר הגודל של הפעולה יהיה O(1). אם כן, יצגו את הרשימה מחדש והסבירו את השינויים שהכנסתם.

שאלה 15

כתבו פעולה בונה מעתיקה למחלקה StudentList שהוצגה בפרק. אתם רשאים להוסיף פעולות נוספות למחלקה Student.

שאלה 16

בסעיף ט בפרק זה הצגנו את הרשימה הדו-כיוונית DoublyLinkedList, המאפשרת סריקה של האיברים בשני הכיוונים (הלוך וחזור). המחלקה DoublyLinkedList<T> תיוצג על ידי שרשרת חוליות דו-כיוונית. שרשרת זו מבוססת על החוליה שהכרתם בפרק ייצוג אוספים BiDirNode, שהכילה ערך ושתי הפניות: אחת לחוליה הבאה, ואחת לחוליה הקודמת.

א. כתבו את המחלקה BiDirNode<T>.

ב. כתבו את המחלקה DoublyLinkedList<T> תוך שימוש במחלקה BiDirNode<T>. ממשו במחלקה DoublyLinkedList<T> את הפעולות המופיעות בממשק המחלקה – רשימה ופעולות נוספות שיאפשרו גם את סריקת הרשימה הדו-כיוונית הלוך וחזור.

ג. נתחו את פעולות הממשק של הרשימה הדו-כיוונית. מה תוכלו להגיד על יעילות הפעולות בהשוואה לפעולות הרשימה List<T>?

שאלה 17

ביישומים רבים אנו רוצים לכווץ (zip) מידע קיים ובשלב מסוים לפרוס (unzip) אותו חזרה לגודלו המקורי. בתרגיל זה נעסוק בכיווץ ופריסה של סדרות תווים.

בהינתן סדרת תווים, ניתן לכווץ אותה באופן זה: כל רצף תווים זהים בסדרה יהפוך לזוג המורכב מתו ומכמות מופעיו ברצף הזה.

לדוגמה, את סדרת התווים (משמאל לימין):

A, A, A, A, G, G, G, A, A, C, C, C, C, C, C, C, C, T, T, T, A, A, A, A, A, A, A, A, A, A, C

ניתן לכווץ ולהפוך לסדרה המכווצת:

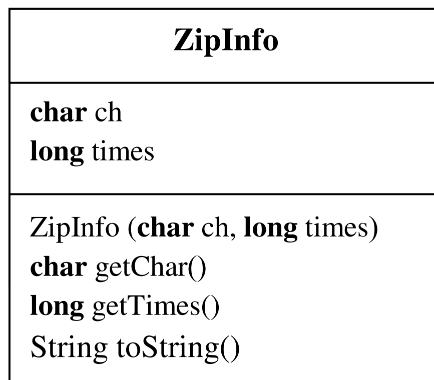
A:4, G:3, A:2, C:8, T:3, A:10, C:1

ניתן לראות שבסדרת התווים הראשונה קיימים 7 רצפים, והם הפכו ל-7 זוגות תווים בסדרה השנייה. ברצף הראשון מופיע התו A 4 פעמים, ברצף השני מופיע התו G 3 פעמים וכן הלאה.

א. ממשו את הפעולה שלפניכם, המכווצת רשימת תווים:

```
public static List<ZipInfo> zip(List<Character> lst)
```

הפעולה מקבלת רשימה של תווים ומחזירה רשימה מכווצת. איבריה של הרשימה המכווצת הם מטיפוס המחלקה ZipInfo (שאותה יש לכתוב). הרשימה מגדירה זוג: תו וכמות מופעיו ברצף, כמתואר בתרשים ה-UML:



ZipInfo
char ch **long** times
ZipInfo (**char** ch, **long** times) **char** getChar() **long** getTimes() String toString()

ב. תהליך הפוך לפעולת הכיווץ היא פעולת הפריסה. ממשו את הפעולה:

```
public static List<Character> unzip(List<ZipInfo> lst)
```

הפעולה מקבלת רשימה מכווצת, פורסת אותה, ומחזירה את רשימת התווים המקורית כפי שהייתה לפני הכיווץ.

שאלה 18

קבוצה היא אוסף לא סדור של ערכים ללא חזרות (כלומר כל ערך מופיע פעם אחת בלבד ואין סדר מחייב של הערכים). הכנסה של ערך שקיים בקבוצה לא תשנה את הקבוצה. הקבוצה יכולה להיות ריקה – ללא ערכים. לפניכם ממשק המחלקה IntSet.

ממשק המחלקה IntSet

המחלקה מגדירה קבוצה של מספרים שלמים.

הפעולה בונה קבוצה ריקה	IntSet()
הפעולה מוסיפה לקבוצה את המספר x, בתנאי שאינו נמצא בה. אם x נמצא בקבוצה הפעולה אינה מבצעת דבר	**void** add (**int** x)
הפעולה מחזירה 'אמת' אם המספר x נמצא בקבוצה ו-'שקר' אחרת	**boolean** exists (**int** x)
הפעולה מוחקת את המספר x מהקבוצה. אם x אינו מופיע בקבוצה הפעולה אינה מבצעת דבר	**void** delete (**int** x)
הפעולה מקבלת קבוצה set ומחזירה את **קבוצת האיחוד** של set ושל הקבוצה הנוכחית	IntSet unify (IntSet set)
הפעולה מקבלת קבוצה set ומחזירה את **קבוצת החיתוך** של set ושל הקבוצה הנוכחית	IntSet intersect (IntSet set)
הפעולה מחזירה מחרוזת המתארת את הקבוצה כך: $\{x_1, x_2, x_3, \ldots\}$	String toString()

הערות:

1. **קבוצת האיחוד** של שתי קבוצות set1 ו-set2 היא קבוצת כל המספרים המופיעים בקבוצה set1 **או** בקבוצה set2.
2. **קבוצת החיתוך** של שתי קבוצות set1 ו-set2 היא קבוצת המספרים המופיעים בקבוצה set1 **וגם** בקבוצה set2.

א. כתבו את המחלקה IntSet. הסבירו את השיקולים בבחירת הייצוג.

ב. נתחו את היעילות של כל אחת מפעולות המחלקה IntSet. ציינו מה סדר הגודל של כל פעולה.

ג. השאלה עוסקת בקבוצות של מספרים שלמים. האם ניתן להכליל את ההגדרה על קבוצה גנרית? נמקו.

שאלה 19

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

ממשק המחלקה IntSortedCollection

המחלקה מגדירה 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 כל המופעים של המספר x מהאוסף הממוין. אם x אינו קיים באוסף הפעולה אינה מבצעת דבר
int[] getAll()	הפעולה מחזירה מערך המכיל את כל המספרים באוסף ממוינים בסדר עולה
String toString()	הפעולה מחזירה מחרוזת המתארת את האוסף הממוין: $[x_1, x_2, x_3, \ldots]$ המספרים ממוינים בסדר עולה.

א. כתבו את המחלקה IntSortedCollection. הסבירו את השיקולים בבחירת הייצוג.

ב. נתחו את היעילות של כל אחת מפעולות המחלקה IntSortedCollection. ציינו מה סדר הגודל של כל פעולה ונמקו.

ג. השאלה עוסקת באוסף של מספרים שלמים. האם ניתן להכליל את ההגדרה על אוסף גנרי? נמקו.

שאלה 20

חיזרו ובצעו שוב את דף עבודה 5 מפרק 6 – "ספר טלפונים". בצעו מחדש את כל הסעיפים תוך התייחסות לנקודות אלה:

א. השתמשו ברשימה לייצוג אוסף אנשי הקשר בספר הטלפונים.

ב. ממשו את פעולות המחלקה כך שהפעולה toString() תתבצע ביעילות לינארית בגודל ספר הטלפונים. **רמז**: בדקו האם הפעולה insertIntoSortedList(…) שראיתם בפרק, יכולה לסייע לכם.

שאלה 21

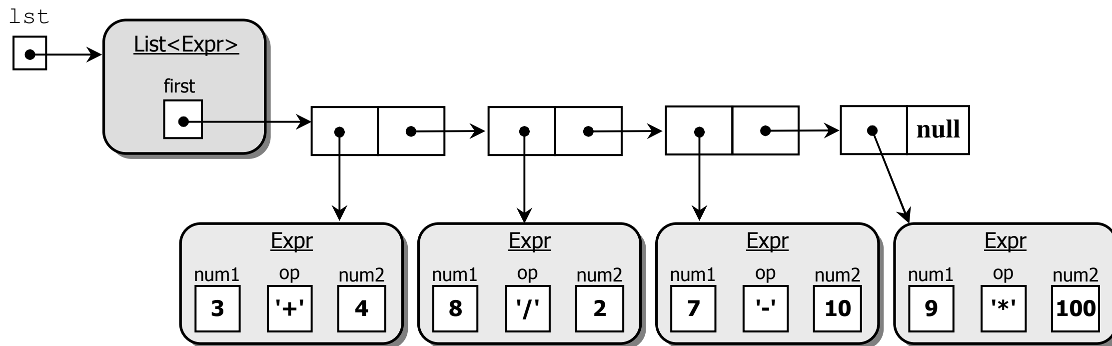
(שאלה זו מבוססת על בחינת הבגרות של קיץ 2005)

רשימה "חשבונית" היא רשימה שאיבריה מטיפוס המחלקה Expr והיא מגדירה ביטוי חשבוני פשוט. ביטוי חשבוני פשוט בנוי משלושה חלקים: <num1> <op> <num2>.

num1, num2 הם מספרים שלמים גדולים מ-0. op הוא אחד מארבעת התווים: / ,* ,- ,+ המייצגים פעולה חשבונית המופעלת על שני המספרים.

להלן ארבע דוגמאות לביטויים חשבוניים פשוטים: 3 + 4, 8 / 2, 7 - 10, 9*100.

הרשימה החשבונית lst מכילה את ארבעת הביטויים החשבוניים הנזכרים:



א. כתבו את המחלקה Expr. המחלקה תכלול פעולה בונה ליצירת ביטוי חשבוני פשוט, פעולות לאחזור המספרים ולאחזור הפעולה החשבונית, וכן פעולה המחזירה מחרוזת המתארת את הביטוי החשבוני הפשוט.

ב. הפעולה **public double** calculate() מחזירה את ערכו של ביטוי חשבוני פשוט.
1. היכן תוגדר פעולה זו? הסבירו.
2. ממשו את הפעולה.

ג. נגדיר את הפעולה sumExpressions(...) המחזירה את **הסכום** הכולל של ערכי הביטויים החשבוניים הנמצאים ברשימה "חשבונית". אם הרשימה "החשבונית" ריקה, יוחזר הערך 0. לדוגמה, עבור הרשימה ה"חשבונית" lst המתוארת באיור הנ"ל, הפעולה sumExpressions() תחזיר את הערך 908.
1. היכן תוגדר הפעולה (באיזו מחלקה) – הסבירו.
2. ממשו את הפעולה (הפעולה צריכה להשתמש בפעולה calculate() שכתבתם בסעיף ב).