

**פרק 7 במבני נתונים: ייצוג אוספים,**

**מותאם לתכנית הלימודים החדשה**

הותאם לתכנית הלימודים החדשה על ידי דפנה מינסטר ושרה פונק, קורס מורים

מובילים לעל יסודי תשע"ו

## פרק 7

# ייצוג אוספים

ביישומים רבים יש צורך לשמור אוספים (collections) גדולים של נתונים ולטפל בהם. אוספים אלה הם דינמיים, כלומר כמות הנתונים באוסף גדלה וקטנה במהלך הפעילות, ולרוב היא אינה מוגבלת בגודלה. דוגמה ליישום כזה הייתה המערכת הבית ספרית שעסקנו בה בפרק הקודם. בהמשך הפרק יוצגו דוגמאות לאוספים מסוגים שונים.

גופים רבים מחזיקים מאגר ממוחשב של פרטי אנשים. לדוגמה: בנק מחזיק מאגר לקוחות, לתיאטרון יש מאגר מנויים, למועדון כושר יש מאגר חברים. גם במערכת הבית ספרית שהצגנו בפרק הקודם הוחזק אוסף התלמידים בכל כיתה, וכן מאגר של פרטי ההורים. מאגרים של נתונים מסוגים אחרים כוללים את רשימות הספרים בספרייה ואת רשימת הספרים המושאלים המוחזקים אצל הקוראים. כך גם תוכנת דואר אלקטרוני מחזיק מאגר מסרים ורשימת כתובות, וברשתות מזון ובחנויות אחרות יש מאגר הכולל את המוצרים שבמלאי, וכן את מחירו של כל פריט ואת כמותו במלאי. במחשב כף-יד ובטלפונים סלולריים מוחזקות רשימת כתובות וטלפונים של מכרים.

מאגרים כמו אלה שתיארנו שמורים לעתים קרובות בזיכרון חיצוני. ואולם, כאשר מבצעים פעולות על מאגר - לשם חיפוש וקריאת נתונים או לשם עדכון - התוכנה המטפלת במאגר קוראת תחילה את הנתונים מהזיכרון למאגר נתונים פנימי שלה, ואחר כך מבצעת את הפעולות הדרושות. פעולות אלה יכולות להיות הוספה, מחיקה או שינוי של איבר במאגר, תשובה לבקשת חיפוש, הכנת מכתבים לכל הלקוחות או לחלקם.

עד היום שימש לנו המערך כמבנה נתונים פנימי לשמירת אוספים של נתונים ולכל נתון באוסף נשמר תא במערך. שמירת אוסף נתונים במערך היא סדרתית והגישה אליהם נעשית על ידי שימוש באינדקס. אולם, לשימוש במערך כמה חסרונות. ראשית, המאגרים שבהם אנו רוצים לטפל שונים מאוד זה מזה בגודלם בכיתת בית ספר לכל היותר שלושים וחמישה תלמידים בספרייה גדולה מאות אלפי ספרים ויותר. מכאן שאי אפשר לקבוע מראש את גודל המערך. אפשר להתמודד עם בעיה זו על ידי העברת גודל האוסף כפרמטר לפעולה בונה. אולם, האוספים הם לעתים קרובות דינמיים - איברים נוספים אליהם ונמחקים מהם. מאגר יכול להתחיל כמאגר קטן, לגדול במהירות, ובשלב אחר לקטון שוב, ולעתים קשה לדעת מראש מה יהיה גודלו המקסימלי. הקצאה מראש של מערך שיוכל להכיל את האוסף גם כשיגדל היא בזבוז של מקום רב בזיכרון. גם הקצאת מערך קטן מדי יכולה להיות בעייתית, שכן המערך עלול להתמלא כולו. כדי לבצע פעולת הכנסה נוספת למערך זה יש להקצות מערך חדש, גדול יותר, להעתיק אליו את כל האיברים ורק אז לבצע הכנסה של נתונים נוספים.

בעיה חריפה יותר היא מחירי ביצוע הפעולות. כאשר מדובר באוסף ממוין, יש לבצע פעולות הכנסה לאוסף של איברים חדשים במקום המתאים להם לפי קריטריון המיון, הבה נחשוב על מערך של 10,000 תאים, מכיל אוסף של 9,800 איברים המאוחסנים ב-9,800 התאים הראשונים במערך. נניח שאנו רוצים עתה להכניס איבר חדש, ומקומו לפי המיון הוא אחרי האיבר ה-5,000.

כדי לפנות עבורו את התא ה- 5,001, עלינו להזיז 4,800 איברים נעשה זאת על ידי העתקת כל איבר לתא העוקב.

בדומה לכל, אם מוחקים את האיבר שבתא ה- 4,950, יש להזיז את 4,850 האיברים הנמצאים מעליו במערך, כדי "לסתום את החור". ראינו אם כל שהמערך אינו מתאים כלל לאחסון אוספים דינמיים שבהם פעולות הכנסה והוצאה מתבצעות בכל מקום. לבסוף, כיוון שהמערך הוא מבנה סדרתי, הוא מתאים לאוספים שהם סדרות. אך עולה השאלה כיצד נייצג אוספים שאינם סדרות, אלא אוספים המייצגים היררכיה וסדר בין הנתונים, כגון עץ משפחה או היררכיה של עובדים במפעל?

לסיכום, המערך כאמצעי לאחסון אוספים סובל מכמה חסרונות בולטים:

1. **מגבלת המקום.** מרגע שנוצר מערך, גודלו נקבע, ולא ניתן להגדילו או להקטינו עוד.
2. **סיבוך גבוה לפעולות הכנסה והוצאה.** הוספת נתון למקום שרירותי במערך או הוצאת נתון ממנו הן פעולות יקרות.
3. **מגבלת המבנה הסדרתי.** המערך הוא מבנה סדרתי וקשה לאחסן בו אוספים בעלי מבנה מורכב.

ידועות שיטות המאפשרות להתמודד עם מגבלות אלה, ובפרט עם המגבלה האחרונה. אולם, ככלל, אם נשתמש במערך לשמירת אוספי נתונים נסתכן במחירים גבוהים לחלק מהפעולות, וכן בסיבוך של התוכנה.

לשמחתנו, קיימת חלופה גמישה יותר המאפשרת אחסון אוספים דינמיים בלי לבזבז מקום, עם אפשרות להגדלה דינמית של כל אוסף, כמעט ללא גבול וביצוע זול של רוב או כל הפעולות הדרושות. חלופה זו תאפשר גם אחסון אוספים היררכיים ואוספים מסובכים אף יותר. פגשנו לראשונה חלופה זו בפרק הפניות, בפרק זה נעסוק בהרחבה בשיטה זו לאחסון אוספי נתונים.

## א. החוליה – אבן יסוד למבנה נתונים דינמי

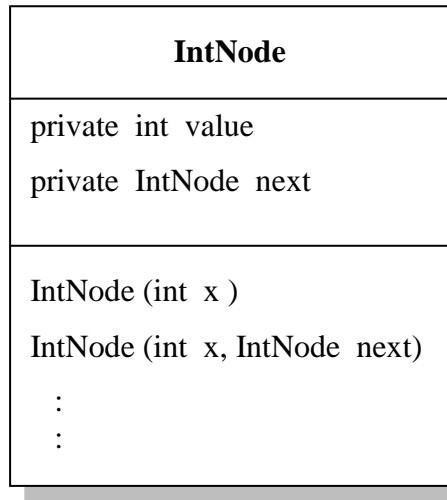
בפרק הקודם, שבו למדנו על הפניות, הכרנו את הגישה של "שרשור עצמי", המשתמשת בהפניה לעצם, כדרך לקישור בין עצמים. נאמץ גישה זו ונעסוק בעצמים שכל אחד מהם מכיל נתון והפניה אחת או יותר לעצמים מאותו הטיפוס. על ידי שרשור של עצמים כאלה נוכל לבנות מבנים לייצוג אוספים של נתונים ולבצע עליהם פעולות כגון חיפוש נתון, גדלת אוסף או הקטנת האוסף וכיוצא בהן. לעצמים כאלה נקרא חוליות.

כאשר מגדירים את המחלקה **חוליה**, Node, יש לקבוע את מספר ההפניות בה. לשם הפשטות נתמקד ברוב הפרק בחוליות שלהן הפניה אחת ויחידה. חוליה כזו היא, אם כן, עצם ובו שתי תכונות:

1. ערך נתון.
2. הפניה לחוליה, בדרך כלל חוליה אחרת, שתיקרא: החוליה העוקבת.

## 1.א. חוליה המכילה מספר שלם

נתחיל בחוליה ששמור בה נתון מסוג מספר שלם. חוליה זו תוגדר על ידי המחלקה `IntNode`.  
לפניכם תרשים UML של המחלקה:



כפי שניתן לראות בתרשים ה-UML, מחלקת החוליה מיוצגת על ידי שתי תכונות:

- התכונה `value` – בה נשמור נתון מספרי.
- התכונה `next` – בה נשמרת הפניה לחוליה העוקבת. אם אין כזו, ערך התכונה יהיה `null`.

### ממשק המחלקה `IntNode`

המחלקה מגדירה חוליה שבה ערך שלם והפניה לחוליה העוקבת.

<code>IntNode ( int x )</code>	הפעולה בונה חוליה. הערך של החוליה הוא <code>x</code> , ואין לה חוליה עוקבת
<code>IntNode (int x , IntNode next )</code>	הפעולה בונה חוליה. הערך של החוליה הוא <code>x</code> , והחוליה העוקבת לה היא <code>next</code> . ערכו של <code>next</code> יכול להיות <code>null</code> .
<code>int getValue ( )</code>	הפעולה מחזירה את הערך השמור בחוליה
<code>IntNode getNext ( )</code>	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה מחזירה <code>null</code> .
<code>boolean hasNext ( )</code>	האם יש חוליה נוספת ?
<code>void setValue ( int x )</code>	הפעולה משנה את הערך השמור בחוליה להיות <code>x</code>
<code>void setNext ( IntNode next )</code>	הפעולה משנה את החוליה העוקבת להיות <code>next</code> . ערכו של <code>next</code> יכול להיות <code>null</code>
<code>String toString ( )</code>	הפעולה מחזירה מחרוזת המתארת את החוליה

למחלקת החוליה יש : שתי פעולות בונות ליצירת חוליה ; פעולות ( ... ) - ו ( ... )  
הקובעות את ערכי התכונות של החוליה ומאחזרת אותן ; פעולת ( toString ) המחזירה מחרוזת  
המתארת את החוליה.

להלן מימוש המחלקה IntNode :

```
public class IntNode
{
    private int value ;
    private IntNode next ;

    public IntNode( int x )
    {
        this.value = x ;
        this.next = null ;
    }

    public IntNode( int x , IntNode next )
    {
        this.value = x ;
        this.next = next;
    }

    public int getvalue ( )
    {
        return ( this.value );
    }

    public IntNode getNext ( )
    {
        return ( this.next);
    }

    public boolean hasNext ( )
    {
        return (this.next != null);
    }

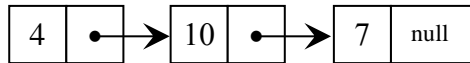
    public void setvalue ( int x )
    {
        this.value = x;
    }

    public void setNext ( IntNode next )
    {
        this.next = next;
    }

    public String toString ( )
    {
        return ( " " + this.value );
    }
}
```

## 2.א. רשימה מקושרת

המחלקה IntNode מאפשרת ליצור רשימה מקושרת של חוליות, לחבר אותן זו לזו באמצעות הפעולה setNext(...), וכך לקבץ יחד קבוצת חוליות של מספרים שלמים. לדוגמה: אוסף המספרים 47, 10, השמור בקבוצה של חוליות, יראה כך:



למבנה כזה מקובל לקרוא **רשימה מקושרת**.

ברשימה מקושרת, כל חוליה שומרת נתון והפניה לחוליה הבאה. החוליה הבאה גם היא שומרת נתון והפניה לחוליה הבאה אחריה וכך הלאה, עד לחוליה האחרונה ברשימה מקושרת המקושרת, השומרת את הנתון האחרון באוסף. מאחר שזו החוליה האחרונה ברשימה מקושרת המקושרת, לא תהיה בה הפניה לחוליה נוספת – ולכן ערך ההפניה בה יהיה **null**.

הרשימה מקושרת המקושרת המתוארת באיור שלעיל מורכבת מ-3 חוליות. בחוליה הראשונה נשמר הנתון המספרי 4 והפניה לחוליה השנייה. בחוליה השנייה שמור הנתון המספרי 10 והפניה לחוליה השלישית. בחוליה השלישית שמור הנתון המספרי 7 והפניה ריקה null המציינת שזו החוליה האחרונה ברשימה מקושרת המקושרת.

כאמור לעיל, פעולות המחלקה IntNode מאפשרות ליצור רשימות שבהן שמורים מספרים שלמים. בהמשך נראה כי פעולות המחלקה מאפשרות גם ליצור מבנים שאינם רשימות, אך ברובו של הפרק נתמקד ברשימות. בהמשך סעיף זה נראה כי רשימה מקושרת היא מבנה נתונים דינמי, הנותן מענה לחסרונות המערך שמנינו בראשית הפרק: ניתן להוסיף כמה חוליות שנרצה לכל מקום ברשימה מקושרת המקושרת וכן להוציא חוליות ממנה – וזאת בצורה פשוטה למדי. נלמד כיצד לעבור על כל החוליות ברשימה מקושרת, להוסיף חוליות ולהוציאן כרצוננו.

### 2.1.א. בניית רשימה מקושרת של מספרים שלמים

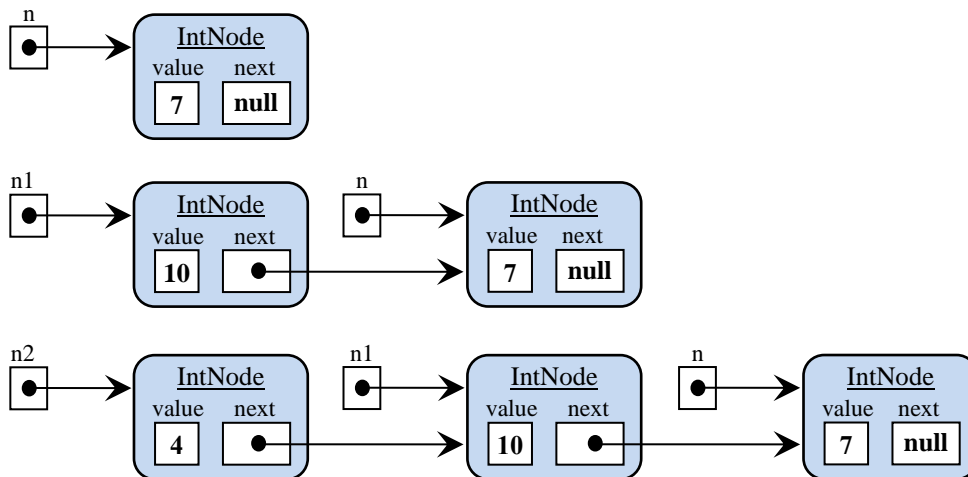
נתבונן בקטע התוכנית שלפנינו, הבונה רשימה מקושרת שמאוחסן בה אוסף הנתונים 4, 10, 7:

```
IntNode n = new IntNode(7);
```

```
IntNode n1 = new IntNode(10,n);
```

```
IntNode n2 = new IntNode(4,n1);
```

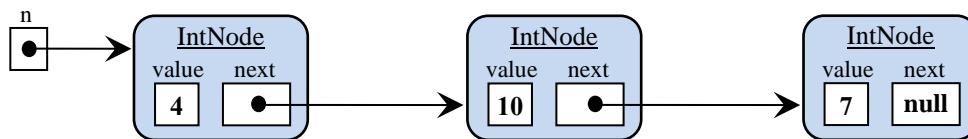
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב את ביצוע קטע התוכנית:



ניתן לקצר את קטע התוכנית הנזכר לעיל ולכתובו כך:

```
IntNode n = new IntNode(4, new IntNode(10, new IntNode(7)));
```

תרשים העצמים של התוכנית המקוצרת יראה כעת כך:



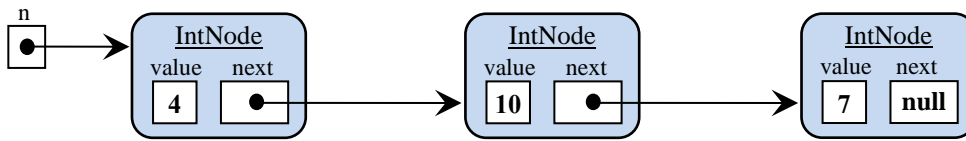
נוסף ליתרון שבכתיבה מקוצרת, דרך כתיבה זו לא נוצרות הפניות נוספות לחוליות שבאמצע הרשימה המקושרת. כפי שכבר הזכרנו בפרק הקודם (6) שדן בהפניות, זהו הישג טוב, שכן ריבוי הפניות עלול לגרום לבעיות שונות. נרחיב את הדיון בנושא זה בהמשך הפרק.

שימו לב ♥, שאת ההפניה לחוליה הראשונה, השמורה במשתנה n, חייבים לשמור. אם הפניה זו תימחק, לא נוכל לגשת יותר לרשימה המקושרת. על פי הגדרת השפה, במקרה זה הרשימה המקושרת לא תיחיה קיימת יותר. מנגנון איסוף הזבל, Garbage Collection, ישחרר את הרשימה המקושרת מהזכרון כשיזהה שהיא עצם שאין אליו הפניה.

### 2.2.א. בניית רשימה מקושרת של מספרים שלמים

על מנת להכניס את הערך x למקום נתון ברשימה מקושרת, עלינו ליצור חוליה חדשה כך שערך התכונה value שלה יהיה x החוליה הקודמת לה תפנה אליה, וערך התכונה next של החוליה החדשה יפנה אל החוליה הבאה ברשימה המקושרת.

נחזור אל הרשימה המקושרת שאנו עוסקים בה :

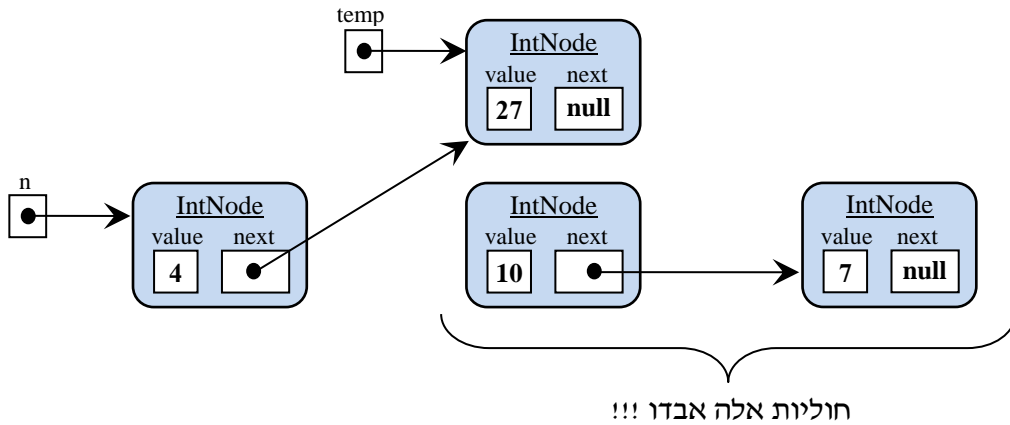


נניח שנרצה להכניס לרשימה המקושרת הזו את המספר 27 לאחר החוליה הראשונה כחוליה שנייה ברשימה המקושרת. נעקוב אחרי שלבי ההכנסה של הערך החדש לרשימה המקושרת.

נגדיר חוליה חדשה שהערך בה הוא 27 ונעדכן את החוליה הקודמת, הראשונה ברשימה המקושרת, כך שתפנה אל החוליה החדשה :

```
IntNode temp = new IntNode(27);
```

```
n.setNext(temp);
```



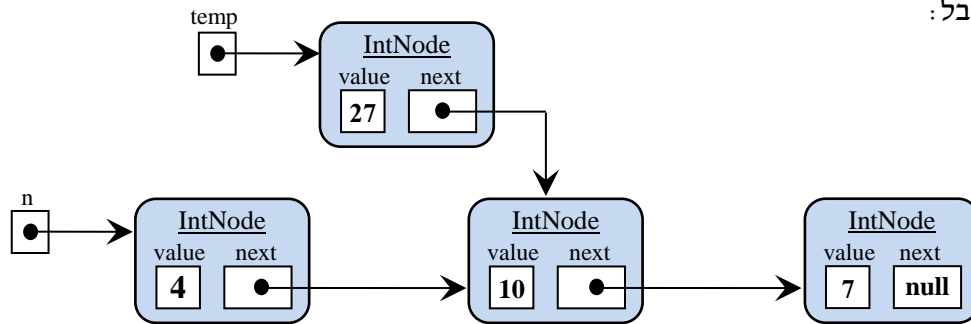
בשלב זה נרצה לבצע את החיבור האחרון ברשימה המקושרת: הפניית החוליה החדשה אל זו העוקבת לה, שהייתה שנייה ברשימה המקורית. אך ערך ההפניה הנחוץ, שהיה שמור בתכונה next של החוליה הראשונה, נמחק כאשר הוכנה אליו ההפניה לחוליה החדשה! כעת אין ביכולתנו לבצע את החיבור המתאים. נראה שהיינו צריכים לבצע את כל מהלך ההוספה "מהסוף להתחלה": ראשית היה עלינו ליצור חוליה חדשה שתכיל את הערך x ותצביע אל החוליה שתבוא אחריה, ורק אז לעדכן את ההפניה אל החוליה החדשה:

```
IntNode temp = new IntNode(27,n.getNext());
```

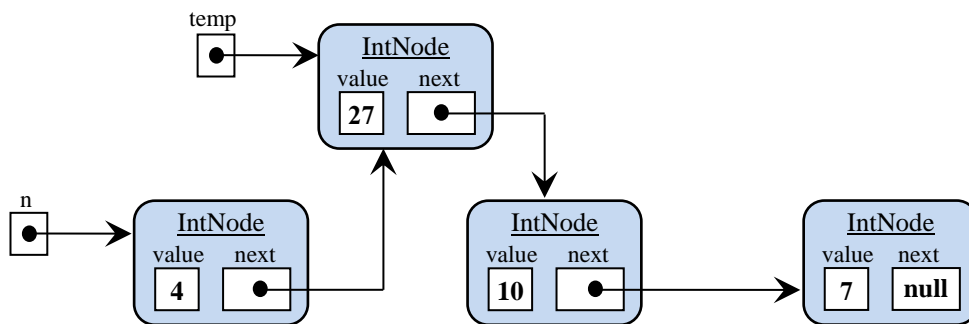
```
n.setNext(temp);
```



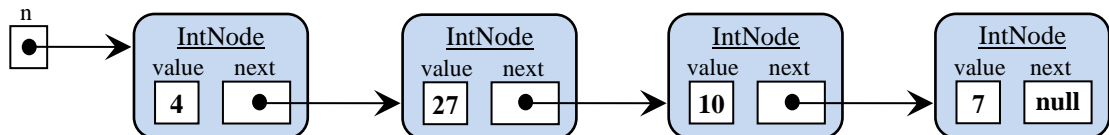
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב, את ביצוע קטע התוכנית. לאחר ההוראה הראשונה נקבל:



לאחר ההוראה השנייה נקבל:



נוח יותר יהיה להסתכל על הרשימה המקושרת "הישרה" שהתקבלה (הפעם ללא temp):



הכנסנו חוליה המכילה נתון חדש אחרי החוליה הראשונה. יכולנו להכניסה גם אחרי החוליה השנייה או השלישית – לפי רצוננו.

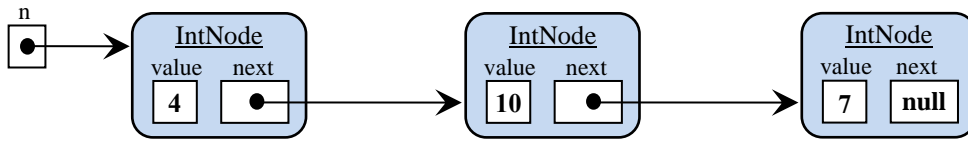
קיום פעולת ההכנסה כפי שתוארה מראה ששתי המגבלות של המערך שתיארנו לעיל אינן קיימות ברשימה מקושרת:

1. מגבלת המקום – ניתן להכניס מספר לא מוגבל של חוליות.
2. סיבוך – לא נדרשות הזזות ימינה כדי לאפשר הכנסה של נתון חדש. כל שיש לעשות הוא לחבר הפניות ולנתקן מחיבוריהן הקודמים, לפי הסדר הרצוי.

בסעיף ה' (בהמשך הפרק) נראה כי הרשימה המקושרת פותרת גם את המגבלה השלישית, ומאפשרת לאחסן אוספים בעלי מורכב שאינו סדרתי.

### א.2.3. הוצאת חוליה מרשימה מקושרת

נתבונן שוב ברשימה המקושרת שאנו פועלים עליה:



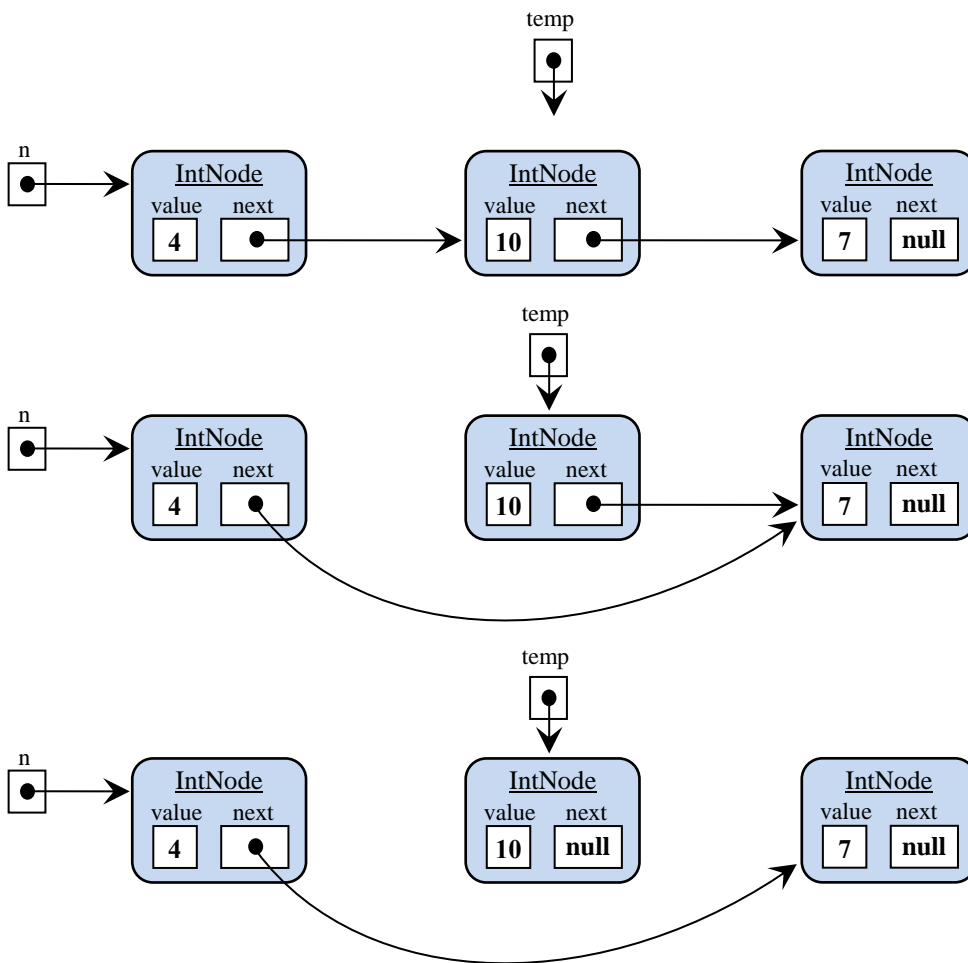
קטע התכנית שלפניכם, מוציא את החוליה השנייה (שערכה 10) מתוך הרשימה המקושרת:

```

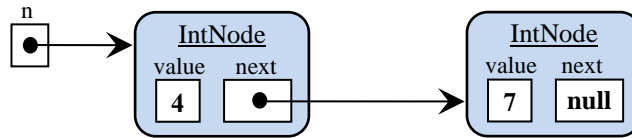
IntNode temp = n.getNext();
n.setNext(temp.getNext());
temp.setNext(null);
    
```

שימו לב ♥ לא נוצרה כאן חוליה חדשה.  
 ההפניה temp מצביעה על חוליה קיימת.  
 אין את הפקודה **new**.

תרשימי העצמים שלפניכם מתארים, שלב אחר שלב, את ביצוע קטע התכנית:



נוח יותר יהיה להסתכל על הרשימה המקושרת המתקבלת לאחר הוצאת המספר 10 כך :



ההוצאה מתבצעת על ידי ניתוק של הפניות בצורה פשוטה וחיבורן מחדש. ההוצאה אינה יוצרת "חור" כפי שהיה קורה במערך, ולכן אין צורך להזיז שמאלה במקום אחד את כל הנתונים שמימין לנקודת ההוצאה, כדי "לסגור" אותו.

שימו לב ♥ להבדל בין הכנסה של חוליות להוצאה שלהן. בהכנסה, החוליה המוכנסת היא חדשה, ואינה קיימת ברשימה המקושרת המקורית. ההכנסה מתבצעת אחרי חוליה קיימת שהפניה אליה נתונה לנו. להפניה זו אנו מגיעים על ידי סריקת הרשימה המקושרת עד החוליה שאחריה אנו רוצים להכניס את החוליה החדשה. בהוצאה, לעומת זאת, אנו מוציאים חוליה קיימת. כדי לבצע את ההוצאה יש למצוא את החוליה **הקודמת** לה ברשימה. רק אז נוכל לבצע את ההוצאה כפי שתיארנו.

#### 2.4. מעבר על מרשימה מקושרת

לעיתים קרובות מאד נרצה לסרוק רשימה מקושרת, כולמר לעבור על החוליות שבה, מתחילתה עד סופה, או עד שנמצא את החוליה המבוקשת. להלן תבנית של קטע קוד המבצע סריקה מלאה על רשימה מקושרת, המוחזקת על ידי `list`. כיוון שידוע שבסוף רשימה מקושרת קיימת חוליה שערכו של העוקב שלה הוא `null`, נסתמך על ערך זה בתנאי העצירה של הלולאה :

```

IntNode pos = list ;
while (pos != null)
{
  //          pos.getValue() חוליה ערך
  pos = pos.getNext();
}
  
```

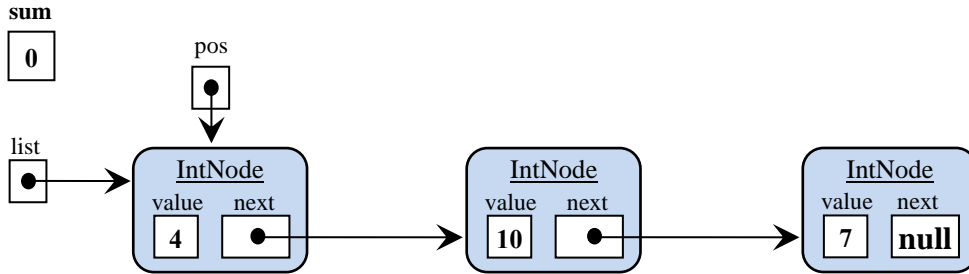
שימו לב ♥ לא נוצרה כאן חוליה חדשה.  
 אין את הפקודה `.new`.

ניישם את התבנית הזו בקטע הקוד שלפניכם, המחשב את סכום המספרים השמורים ברשימה המקושרת ש-`list` מפנה אליה, החל בחוליה הראשונה וכלה באחרונה :

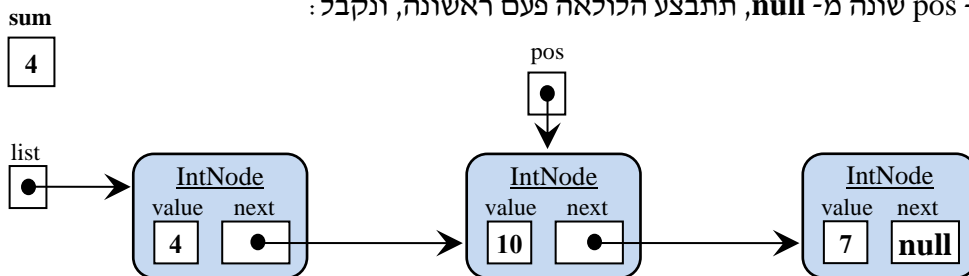
```

int sum = 0;
IntNode pos = list ;
while (pos != null)
{
  sum = sum + pos.getValue()           // סיכום ערכי החוליות
  pos = pos.getNext();
}
  
```

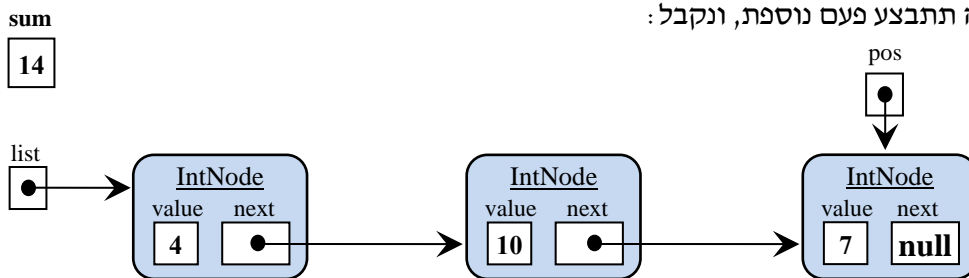
ההפניה החדשה, pos, מסייעת במעבר על הרשימה המקושרת. תרשימי העצמים שלפניכם מתארים, שלב אחר שלב, את ביצוע קטע התכנית: זהו המצב ההתחלתי, אחרי אתחול של pos ושל sum, ולפני שנכנסים ללולאה:



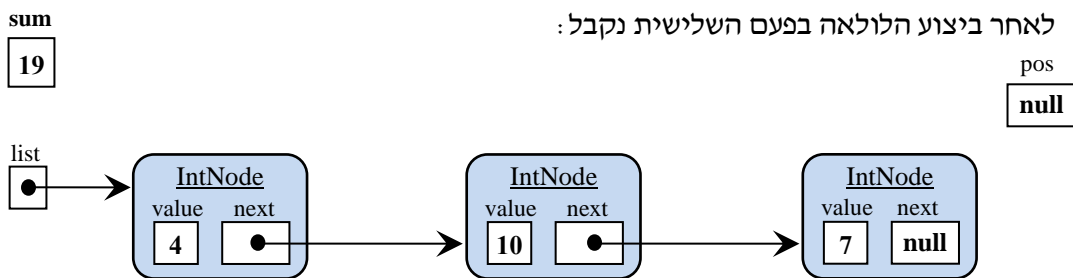
כיוון ש-pos שונה מ-null, תתבצע הלולאה פעם ראשונה, ונקבל:



הלולאה תתבצע פעם נוספת, ונקבל:



לאחר ביצוע הלולאה בפעם השלישית נקבל:



בסיום קטע הקוד, pos יקבל את הערך null (כי ברשימה זו לא קיימת חוליה נוספת), הלולאה תיפסק, וקטע התוכנית יסתיים. במשתנה sum נמצא עכשיו הערך 19 השווה לסכום המספרים ברשימה המקושרת. סכום זה ישמש כערך החזרה של הפעולה.

---

כתבו קטע תוכנית המוצא את המספר הגדול ביותר ברשימה מקושרת המוחזקת על ידי list.

---

### 3.א. רשימה מקושרת כפרמטר לפעולות

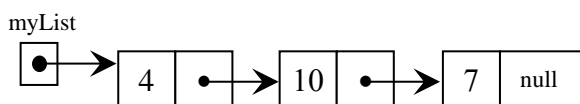
עד כה הדגמנו בעזרת קטעי קוד כמה שינויים אפשריים של רשימה מקושרת. אם הרשימה נוצרה בתוך הפעולה הראשית, אזי קטעי הקוד הללו שולבו אף הם בפעולה זו, אחרי בניית הרשימה. גישה זו נוחה להתנסות ראשונית, אך כדי להשתמש ברשימות מקושרות באופן כללי, נרצה לכתוב פעולות היכולות להתבצע על רשימה כלשהי. לפעמים, די להעביר לפעולה כזו הפניה לחוליה שעליה היא צריכה לפעול; במקרים אחרים, נעביר גם הפניה לרשימה כולה. שימו לב ♥ **רשימה מקושרת אינה טיפוס נתונים, אלא מבנה שנוצר מחוליות, והוא אינו מוגדר בעזרת מחלקה משל עצמו.** כדי להעביר רשימה מקושרת כפרמטר, נעביר הפניה לחוליה הראשונה שלה. כלומר הפניה לרשימה מקושרת גם היא הפניה מטיפוס חוליה. היא מפנה לחוליה הראשונה ברשימה, ומבחינתנו היא "מפנה לרשימה מקושרת". נתבונן בכמה דוגמאות. במהלך הדיון נדון במקרי קצה שונים, ובשגיאות אפשריות.

#### 3.1.א. סכום של רשימה מקושרת

נכתוב פעולה המקבלת רשימה מקושרת של מספרים שלמים (על ידי העברת הפניה לחוליה הראשונה ברשימה), ומחזירה את סכום המספרים ברשימה המקושרת:

```
public static int getListSum (IntNode list)
{
    int sum = 0;
    while ( list != null )
    {
        sum = sum + list.getValue();
        list = list.getNext();
    }
    return sum ;
}
```

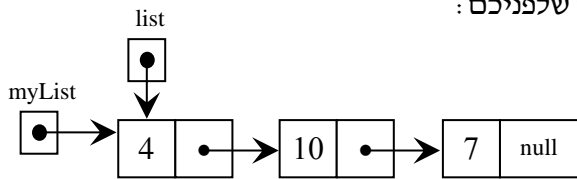
נתונה רשימה מקושרת המוחזקת על ידי משתנה myList:



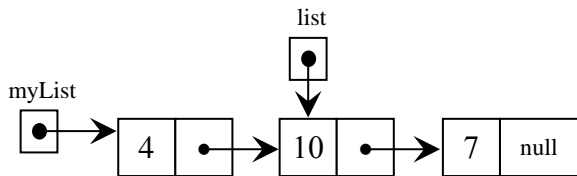
כדי לחשב את סכום המספרים ברשימה המקושרת myList ניתן לזמן את הפעולה `getListSum()` ( ... ) שכתבנו ולשלוח אליה את הרשימה המקושרת myList. הזימון יראה כך :

```
int totalSum = getListSum(list);
```

ברגע הזימון הערך שנמצא במשתנה myList, שהוא הפניה לחוליה הראשונה ברשימה, מועתק לפרמטר list שבכותרת הפעולה (...). getListSum(...). לכן נוצר מצב שבו לחוליה הראשונה ברשימה המקושרת יש שתי הפניות, כמתואר באיור שלפניכם:



כעת גוף הפעולה מתחיל להתבצע. המשתנה list עובר על פני הרשימה המקושרת. לאחר ביצוע הסבב הראשון של הלולאה, המשתנה list יכול הפניה לחוליה השנייה:



וכך הלאה. הלולאה תיפסק, כאשר נגיע לסוף הרשימה המקושרת, והמשתנה list יכול את הערך null. שימו לב ♥, יכולנו לסרוק את הרשימה באמצעות משתנה מקומי נוסף – pos, אך אין צורך לייצר כפילות שכזו. העברת הרשימה לפרמטר הפנימי list מאפשרת לנו לסרוק את הרשימה באמצעותו. התקדמותו של list אינה מפריעה להמשך החזקת הרשימה המקורית על ידי המשתנה החיצוני myList. פעולת הסכימה אינה משנה את מבנה הרשימה המקושרת או את תוכנה.

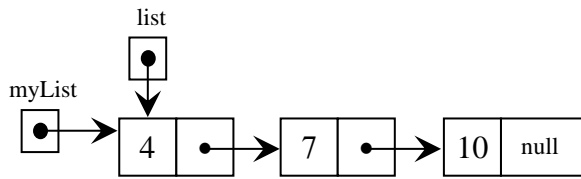
### א.3.2. הכנסת ערך לרשימה מקושרת ממוינת

נניח כי נתונה לנו רשימה מקושרת ממוינת, כלומר הערך בכל חוליה (פרט לראשונה) גדול מהערך שבקודמתה, ואנו מעוניינים בפעולה שתאפשר להכניס ערך חדש למקומו הנכון ברשימה. כדי להכניס ערך לרשימה זו עלינו להעביר את הערך להכנסה ואת הרשימה עצמה כפרמטרים לפעולה. לא נדון כאן בפירוט מימוש הפעולה, פרט לכך שהיא תבצע סריקה של הרשימה החל במקום הראשון בה, עד מציאת המקום המתאים לביצוע ההכנסה, ואז הפעולה תבצע את ההכנסה:

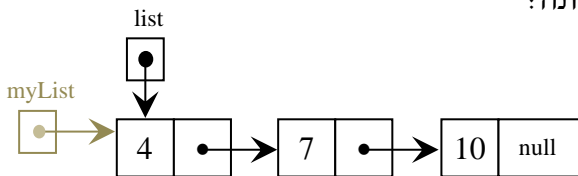
```
public static void insertIntoSortedList(IntNode list , int x)
{
    ...
}
```

נבחן את מקרה הקצה שבו יש להכניס את הערך החדש לראש הרשימה המקושרת. נראה כי במקרה זה אין הפעולה יכולה לבצע זאת. להלן דוגמה המתארת את הבעיה.

נתונה הרשימה המקושרת הממויינת myList :

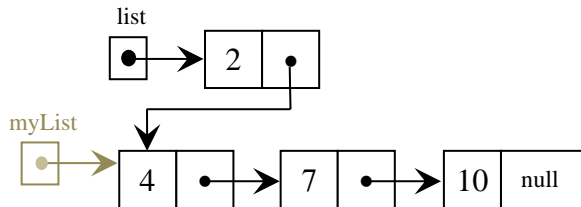


נזמן את הפעולה כך:  $\text{insertIntoSortedList}(\text{myList}, 2)$  במטרה להכניס את הערך 2 לרשימה המקושרת myList הממויינת. הערך 2 הוא הקטן ביותר מבין הערכים 4, 7, 10, ולכן הוא אמור להיכנס כערך הראשון ברשימה המקושרת. בעת זימון הפעולה, הועברה לפרמטר list ההפניה לתחילת הרשימה המקושרת הממויינת. מרגע המעבר לפעולה, ההפניה החיצונית myList אינה מוכרת עוד, ולכן היא נצבעת בצבע שונה :



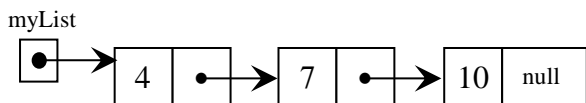
כעת הפעולה  $\text{insertIntoSortedList}(\dots)$  עובדת רק עם הפרמטר list כמחזיק הרשימה המקושרת. הפעולה כלל אינה מכירה בתוכה את ההפניה myList (לכן באיורים אלה היא מסומנת בצבע שונה).

לאחר איתור מקום ההכנסה, הפעולה יוצרת חוליה חדשה המכילה את הערך 2, ומכניסה אותו למקום הראשון ברשימה המקושרת :



איור זה מראה את הבעיה. המשתנה list מכיל הפניה לחוליה הראשונה החדשה, בעוד המשתנה myList לא עודכן בתוך הפעולה. בתום הפעולה  $\text{insertIntoSortedList}(\dots)$  הפרמטר המקומי list מתבטל! אנו חוזרים למצב התחלתי בו myList הוא ההפניה היחידה לרשימה המקושרת, אך myList לא השתנה.

בסיכום כל התהליך, הערך 2 לא הוכנס לרשימה המקושרת כמתבקש :



---

✍ הסבירו מדוע הבעיה שתיארנו אינה מתעוררת אם הכנסת הערך מתבצעת במקום שאינו ראשון ברשימה המקושרת.

---

אותה בעיה תתעורר במקרה קצה אחר, כאשר נרצה לנתק את החוליה הראשונה ברשימה מקושרת. הבעייתיות המוצגת תתעורר למעשה בכל פעולה של שינוי מבנה הרשימה כאשר הדבר נוגע לחוליה הראשונה בה.

פתרון לבעיות שהעלינו יוצג בהמשך הפרק ובפרקים הבאים.

### א.3.3. מציאת ערך מקסימלי ברשימה מקושרת

אם ברצוננו לאתר את הערך הגדול ביותר החל מחוליה מסוימת ברשימה מקושרת, לא נצטרך להעביר את הרשימה כולה כפרמטר לפעולה, כיוון שממילא אין לנו צורך לחפש ברשימה כולה. די יהיה בהעברת הפניה לחוליה שממנה אנו רוצים להתחיל את החיפוש.

לפניכם מימוש הפעולה המחזירה הפניה לחוליה שמכילה את הערך המקסימלי ברשימה:

```
public static IntNode getMaxPosition (IntNode pos)
{
    IntNode maxPos = pos;

    while ( pos != null )
    {
        if (pos.getValue() > maxPos.getValue())
            maxPos = pos;
        pos = pos.getNext();
    }
    return maxPos ;
}
```

---

✍ נתונה רשימה מקושרת המוחזקת על ידי `myList`.

א. כתבו את הזימון הנדרש כדי שהפעולה תחזיר הפניה לחוליה בה נמצא הערך הגדול ביותר ברשימה החל מהמקום השביעי.

ב. כתבו את הזימון הנדרש כדי שהפעולה תחזיר את מיקום הערך הגדול ביותר ברשימה המקושרת `myList` כולה.

---

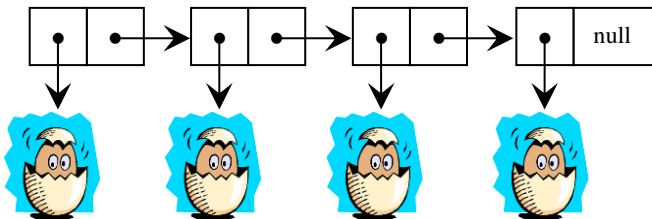
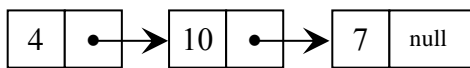


## סיכום ביניים:

רשימה מקושרת אינה עצם מטיפוס נתונים מוגדר על ידי מחלקה, אלא מבנה המורכב מעצמים מטיפוס חוליה. רשימה מקושרת מוחזקת במשתנה על ידי הפניה לחוליה הראשונה שלה. לכן, כשאנו אומרים שפעולה מקבלת רשימה מקושרת, כוונתנו לכך שהפעולה מקבלת הפניה לחוליה הראשונה ברשימה מקושרת. בנוסף, פעולות (כמו למשל הכנסה והוצאה) אינן יכולות לשנות את החוליה הראשונה כך ששינוי זה יראה מחוץ לפעולה.

## ב. חוליה גנרית

כפי שראינו, כל חוליה ברשימה מקושרת מכילה ערך והפניה לחוליה הבאה. עד כה עסקנו בחוליה שמכילה ערך שהוא מספר שלם. לכל טיפוס ערך אחר, כגון מחרוזת או אובייקט, ניתן להגדיר מחלקת חוליה שבה הערך בחוליה הוא מטיפוס זה.



לא נדרשים שינויים משמעותיים כדי להגדיר חוליה שטיפוס הערך שלה שונה ממספר שלם. למשל, כדי להגדיר מחלקת חוליה שהערך השמור זה הוא מחרוזת, נוכל להעתיק את ההגדרה של

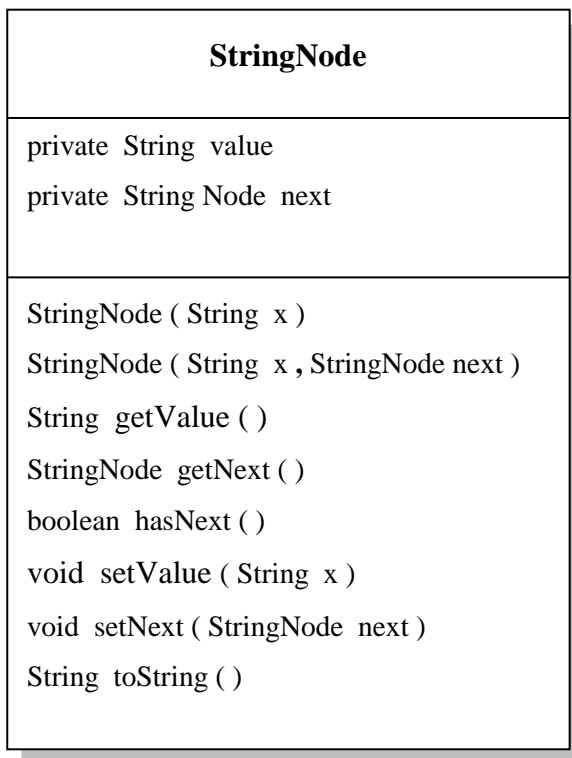
המחלקה `IntNode` ולבצע את השינויים האלה:

1. שינוי טיפוס התכונה `value` לטיפוס `String`.

2. שינוי המחלקה ל- `StringNode` (יש לשמור אותה בקובץ בשם זה).

בהתאמה נשנה את הטיפוס `int` ואת המחלקה `IntNode` בכל מקום שבו טיפוס התכונה או המחלקה מופיעים כערכי הפרמטרים או ערכי החזרה. פרט לכך לא יידרש שום שינוי. נוכל לבנות ממחלקה זו רשימה מקושרת בדיוק כמו שבנינו רשימה מקושרת מחוליות של `IntNode`.

להלן תרשים UML של המחלקה החדשה:



האם בכל פעם שנרצה ליצור רשימה מקושרת שערכיה טיפוס מסוים נצטרך לכתוב מחלקה חדשה כפי שעשינו עד כה? העתקה של קטעי קוד, תוך הכנסת שינויים מזעריים נראית מיותרת, ויש בה גם סכנה של יצירת שגיאות. לשמחתנו, קיים בשפה מנגנון הקרוי מנגנון הגנריות (genericity), שיאפשר לנו להמנע מכך. מנגנון זה מאפשר לכתוב הגדרה יחידה של מחלקה גנרית, ולהשתמש בה לטיפוסי ערכים שונים.

כך תיראה כותרת המחלקה הגנרית של החוליה:

```
public class Node < T >  
{  
    // מימוש המחלקה באמצעות הטיפוס T (המימוש יתואר בהמשך)  
}
```

הסימן T אינו שם של טיפוס מסוים, אלא **מחזיק מקום (place holder)** לטיפוס שעדיין לא נקבע. התוספת <T> אחרי שם המחלקה משמעותה, שזו הגדרה של מחלקה גנרית שבה הטיפוס, שעדיין לא נקבע, מסומן באות T (כפי שנראה בהמשך, נעשה שימוש ב-T בתוך קוד המחלקה). אין חשיבות לאות T, ניתן לסמן את הטיפוס בכל אות או מזהה אחרים (מקובל להשתמש באות יחידה). בעת השימוש במחלקה יש לקבוע טיפוס קונקרטי במקום הטיפוס T.

אם כן, מנגנון הגנריות מאפשר לנו להימנע בזמן הגדרת המחלקה `Node < T >` מהתחייבות על טיפוס הערך המדויק שיאוחסן בחוליה. את הטיפוס צריך לקבוע רק בזמן השימוש במחלקה, ובעת שימושים שונים ניתן לקבוע טיפוסים שונים. כך אנו יכולים להגדיר מחלקה כללית של חוליה, שניתן להשתמש בה לכל טיפוס הערכים (במגבלה קלה שתידון בהמשך). רק כאשר נשתמש בהגדרת המחלקה ליצירת עצם או להכרזת טיפוס של משתנה, נצטרך לציין את הטיפוס המדויק של הערך השמור בחוליה. באותו הרגע יבצע המעבד "הצבה", ובכל מקום שניכתב `T`, הוא יוחלף בטיפוס הקונקרטי שהגדרנו.

`T` בכותרת המחלקה הוא למעשה פרמטר טיפוס של המחלקה, המוחלף בכל שימוש בטיפוס קונקרטי.

לדוגמה, לאחר שנגדיר את מחלקת החוליה הגנרית נוכל לכתוב בתוכנית הראשית את השורה הזו:

```
Node<String> nodeStr = new Node<String>("daf");
```

כאן השתמשנו במחלקה `Node` פעמיים: להכרזת טיפוס המשתנה `nodeStr`, וכן ליצירת עצם חדש מטיפוס המחלקה. בשני האגפים השתמשנו בשם המחלקה ואחריו הסימון `<String>`.

המשמעות היא שאנו משתמשים כאן במחלקה `Node`, ו-`T` נקבע להיות הטיפוס `String`. בשורת הקוד הגדרנו משתנה שיכול להחזיק הפניות לחוליות שבהן מחרוזות, יצרנו חוליה שכזו והכנסנו את ההפניה אליה למשתנה.

הערה לגבי השימוש במנגנון הגנריות: אפשר לקבוע את `T` להיות טיפוס של עצם כלשהו, אך לא להיות טיפוס בסיסי (`primitive`), כגון `int` (מגבלה זו נובעת מימוש רעיון הגנריות, אך נושא זה רחב ולא נעסוק בו כאן).

בהמשך נראה שלמרות מגבלה זו ניתן לייצר רשימה מקושרת של מספרים שלמים או כל טיפוס בסיסי אחר, ולהשתמש בה ללא קושי.

לפניכם ממשק החוליה הגנרית:

### **מימשק החוליה הגנרית `Node < T >`**

המחלקה מגדירה חוליה גנרית שבה ערך מטיפוס `T` והפניה לחוליה העוקבת.

Node ( T x )	הפעולה בונה חוליה. הערך של החוליה הוא x, ואין לה חוליה עוקבת
Node ( T x , Node< T > next )	הפעולה בונה חוליה. הערך של החוליה הוא x, והחוליה העוקבת לה היא next. ערכו של next יכול להיות null.
T getValue ( )	הפעולה מחזירה את הערך השמור בחוליה
Node< T > getNext ( )	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה מחזירה null.
boolean hasNext ( )	האם יש חוליה נוספת ?
void setValue ( T x )	הפעולה משנה את הערך השמור בחוליה להיות x
void setNext (Node< T > next )	הפעולה משנה את החוליה העוקבת להיות next. ערכו של next יכול להיות null
String toString ( )	הפעולה מחזירה מחרוזת המתארת את החוליה

בממשק אנו מציינים מצב חריג למקובל ביחידה זו. בפעולה הבונה השנייה ובפעולה ( setNext ( ... ) ערך ההפניה הנשלחת כפרמטר יכול להיות שווה null. המשמעות תהיה שאין חוליה עוקבת לחוליה הנוכחית.

שימו לב ♥ לעובדה שבכל פעולות ממשק שמתייחסת לערך החוליה יש שימוש בסימון T כטיפוס ערך החוליה. קודם שנראה את מימוש המחלקה, נבחן שימושים שונים שניתן לעשות בה.

### 1.1. שימוש בחוליה הגנרית

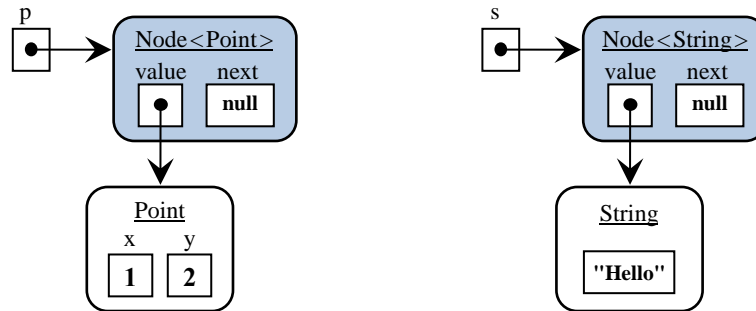
ניתן ליצור רשימה מקושרת שהערכים בהן מטיפוס גנרי, ובעת היצירה יש לציין את הטיפוס הקונקרטי הרצוי. יתירה מכך, ניתן ליצור באותה התוכנית כמה רשימות מטיפוסים שונים:

```
public static void main (String [ ] args)
{
    // 1 שלב
    Node<Point> p = new Node<Point>( new Point(1,2));
    Node<String> s = new Node<String>("Hello");

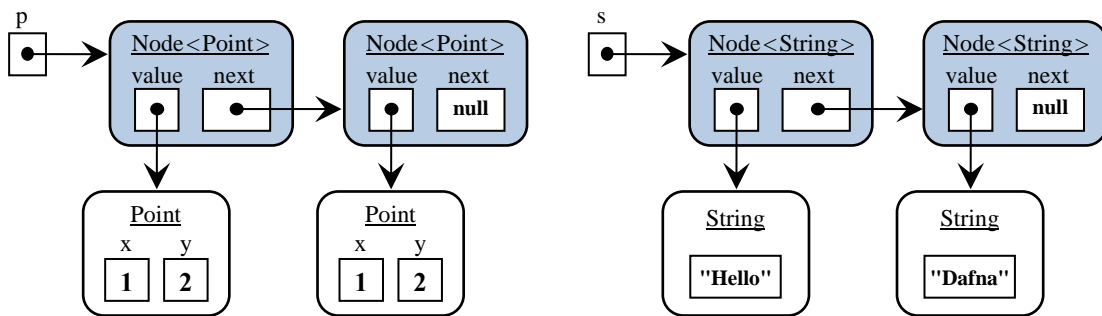
    // 2 שלב
    p.setNext (new Node<Point>(new Point(3,4)));
    s.setNext (new Node<String>("Dafna"));
}
```

תרשימי העצמים שלפניכם מתארים את בניית החוליות והרשימות בתוכנית :

לאחר שלב 1 :



לאחר שלב 2 :



שימו לב ♥ כאשר מבצעים ביישום את הפעולה `getValue()` על חוליה מתקבל עצם מטיפוס ידוע (קונקרטי). כדי לטפל בו יש לדעת מה הטיפוס של עצם זה, וכך אפשר להיעזר בפעולותיו.

## 2.ב. מחלקות עוטפות

כאמור לעיל, טיפוס הערך בחוליה חייב להיות טיפוס של עצם. דרישה זו מעלה בעיה: כיצד נייצר רשימה מקושרת שהערכים השמורים בה הם מטיפוס פשוט, כגון `int`? לשם כך נשתמש במנגנון נוסף הקיים בשפה.

לכל טיפוס פשוט מוגדרת מחלקה מקבילה: `Integer` עבור `int`, `Double` עבור `double`, `Character` עבור `char`, וכן הלאה. מחלקות אלה מאפשרות להתייחס לערכים של טיפוסים בסיסיים כאל עצמים. מחלקות אלה נקראות **מחלקות עוטפות (type wrapper classes)** (מידע נוסף עליהן ניתן למצוא בתיעוד on-line של השפה). כדי להקל על המתכנת, ההמרות בין ערכים מטיפוסי בסיס לעצמים עוטפים, וחזרה, נעשות אוטומטית על ידי מערכת, בהתאם לצורך, והמתכנת אינו נדרש לכתוב המרות בתוכניתו.

👉 **חשוב לדעת:** המרה אוטומטית כמתואר כאן קיימת בג'אווה רק מגרסה 1.5.

כדי לייצר ולהשתמש ברשימה מקושרת של מספרים שלמים, מגדירים חוליה מטיפוס Integer. ניתן להתייחס לערך המאוחסן בחוליה הן כעצם מטיפוס Integer, והן כערך מטיפוס **int**:

```
Node<Integer> n = new Node<Integer>(101);  
int x = n.getValue( );
```

בשורה הראשונה יצרנו חוליה מטיפוס Integer ובתוכה אוחסן המספר 101, שהוא ערך מטיפוס **int**. התבצעה המרה אוטומטית לעצם מטיפוס Integer (המערכת יודעת שקיים צורך בהמרה כיוון שהארגומנט של הפעולה הוא מטיפוס **int**, ואילו טיפוס הערך של החוליה הוא Integer). בשורה הבאה אחזרנו את ערך החוליה והצבנו אותו במשתנה x. שוב התבצעה המרה אוטומטית, הפעם מטיפוס Integer לערך מטיפוס **int**.

### ב.3. פעולות פנימיות וחיצוניות

פעולה הכלולה בהגדרת המחלקה A, בין אם היא פומבית, כלומר שייכת לממשק המחלקה, ובין אם היא פרטית, נקראת פעולה **פנימית** של המחלקה A. פעולה המופיעה בכל מחלקה אחרת היא פעולה **חיצונית** (יחסית ל-A). אם פעולה חיצונית פועלת על עצם מטיפוס A, בהכרח יש לה פרמטר מטיפוס A, אף שבפעולה פנימית אין עצם המפעיל את הפעולה מצויין כפרמטר. אם  $A < T$  היא מחלקה גנרית, אזי השימוש בטיפוס הגנרי T מותר בפעולות הפנימיות שלה, ורק בהן (על פי החלטתנו ביחידת לימוד זו). אי לכך, בפעולות חיצוניות ל-A, שמשתמשות בעצמים של A, יש להחליף את T בטיפוס קונקרטי, כפי שנעשה בדוגמות הקודמות.

### ב.4. על שימוש במחלקות גנריות בפעולות חיצוניות

כפי שכבר אמרנו, כדי להגדיר משתנים או לייצר עצמים חדשים יש לקבוע טיפוס קונקרטי שיחליף את הטיפוס הגנרי בהגדרת המחלקה הגנרית. בהגדרת פעולה, שורת הכותרת מגדירה פרמטרים מטיפוסים שונים. לכן, גם בהגדרת פעולה, לכל פרמטר מטיפוס מחלקה גנרית יש לקבוע טיפוס קונקרטי. למשל, כאשר נכתוב את פעולת הסכימה שהגדרנו בסעיף א.3.1 כך שתסכום רשימה מקושרת של מספרים שלמים, כותרת הפעולה שהייתה במקור:

```
public static int getChainSum( IntNode chain )
```

תיראה כך:

```
public static int getChainSum( Node<Integer> chain )
```

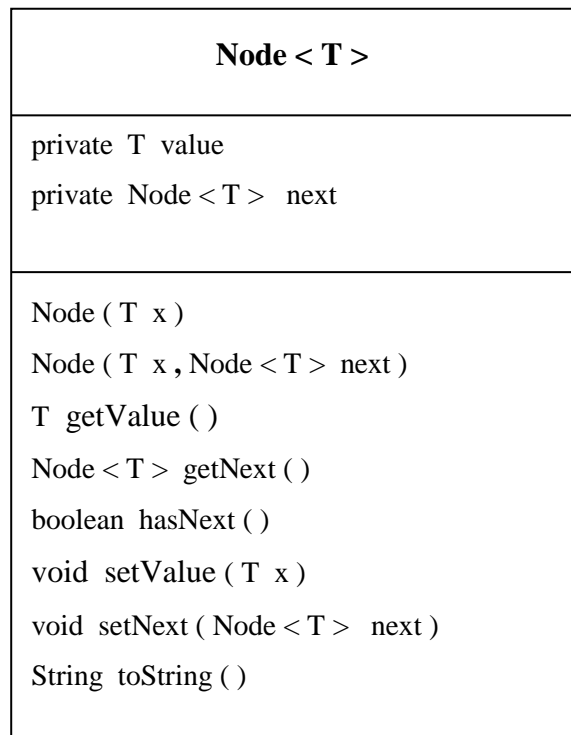
לפרמטרים שונים ניתן לבחור טיפוסים קונקרטיים שונים, כמו בכותרת זו:

```
public static void doIt( Node<Integer> pos1, Node<String> pos2 )
```

**לסיכום:** ביחידה זו, עבור כל פעולה המקבלת פרמטרים מטיפוס מחלקה גנרית, יש לקבוע טיפוסים קונקרטיים לפרמטרים.

### ב.5. מימוש המחלקה גנרית

מימוש המחלקה הגנרית Node יהיה דומה למימוש המחלקה IntNode שראינו, למעט שינוי אחד: בכל מקום שאנו מתייחסים לטיפוס ערך החוליה נחליף אותו בסימון T. להלן תרשים UML של המחלקה הגנרית:



ראינו כבר את הגדרת הכותרת של המחלקה הגנרית, נדון עתה בתכולתה.

```

public class Node< T >
{
    private T value;
    private Node < T > next;

    public Node( T value )
    {
        this.value = value;
        this.next = null;
    }

    public Node( T value, Node< T > next )
    {
        this.value = value;
        this.next = next;
    }

    public T getValue()
    {
        return this.value;
    }

    public Node< T > getNext( )
    {
        return this.next;
    }

    public boolean hasNext ( )
    {
        return (this.next != null);
    }

    public void setValue( T value )
    {
        this.value = value;
    }

    public void setNext( Node< T > next )
    {
        this.next = next;
    }

    public String toString ( )
    {
        return this.value.toString( );
    }
}

```



כפי שניתן לראות בהגדרת טיפוס הערך (value) מופיעה האות T. כאשר אנו מגדירים טיפוס של פרמטר או ערך החזרה שהוא מטיפוס המחלקה אנו מציינים `Node< T >`. כאשר טיפוס פרמטר או ערך החזרה הוא T עצמו, אנו כותבים T. בכותרת המחלקה, T אינו חלק משם המחלקה, ולכן הוא אינו מופיע בכותרת הפעולה הבונה.

כאשר T מופיע בכותרת המחלקה הוא משמש למעשה כפרמטר של המחלקה.

אי לכך, כאשר נפעיל את מנגנון ה-javadoc על המחלקה הגנרית תתקבל שורת תיעוד שתגדיר את הפרמטר של המחלקה:

```

Class Node<T>
  java.lang.Object
    └─ Node<T>

Type Parameters:
  T - טיפוס ערכי החוליה

public class Node<T>
  extends java.lang.Object
  
```

### ב.5. יעילות פעולות המחלקה `Node < T >`

ניתוח יעילות פעולות החוליה מראה שכולן מסדר גודל קבוע,  $O(1)$ . זאת משום שכל פעולה נמשכת זמן קבוע ללא תלות באורך קלט כלשהו.

### ג. רשימה מקושרת כמבנה רקורסיבי

ההגדרה שהשתמשנו בה לרשימה מקושרת, ולפיה הרשימה מקושרת היא אוסף של חוליות, הייתה בסיס טוב לתכנות איטרטיבי של פעולות על רשימות, כפי שעשינו עד כה. אך ניתן להתייחס אל רשימה מקושרת גם כאל מבנה רקורסיבי. להלן הגדרה רקורסיבית של רשימה מקושרת:

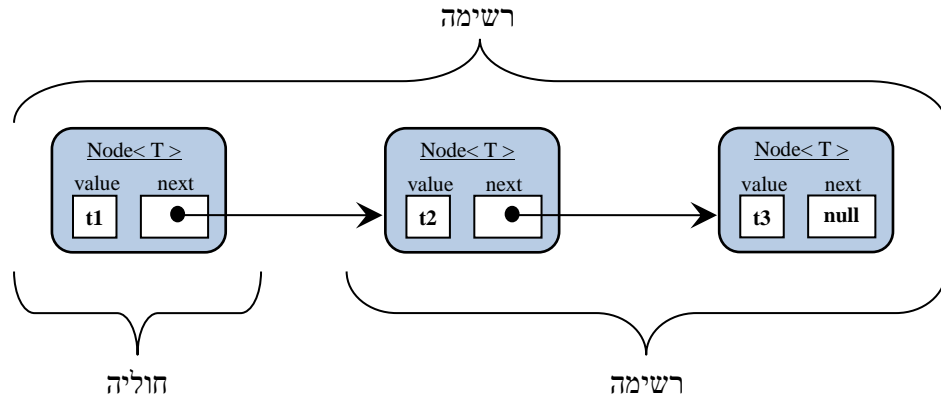
**רשימה מקושרת של חוליות היא:**

- חוליה יחידה

או

- חוליה שיש בה הפנייה לרשימה מקושרת של חוליות

ההגדרה הרקורסיבית מתאפשרת משום שבחוליה קיימת תכונה בשם `next`, המכילה הפניה לחוליה הבאה שהיא מאותו הטיפוס. כלומר, כל חוליה ברשימה מקושרת היא בעצמה התחלה של רשימה מקושרת.



ההגדרה הרקורסיבית מאפשרת לכתוב פעולות רקורסיביות על רשימה מקושרת.

### דוגמה 1: חישוב סכום ברשימה מקושרת

כדי לחשב סכום מספרים שלמים ברשימה מקושרת יש צורך לסרוק את כל החוליות ברשימה מקושרת ולסכם את ערכיהן. בסעיף א.2.4 הצגנו קטע תוכנית שביצע משימה זו תוך שימוש בלולאת **while**. חישוב סכום המספרים ברשימה מקושרת ניתן למימוש גם באופן רקורסיבי, בהסתמך על ההגדרה הרקורסיבית של רשימה מקושרת שהוצגה לעיל. להלן פעולה רקורסיבית המקבלת רשימה מקושרת של מספרים שלמים, ומחזירה את סכום המספרים ברשימה מקושרת. כפי שציינו ביחידת לימוד זו, פעולה המקבלת רשימה מקושרת תקבל רק רשימה מקושרת מטיפוס קונקרטי ולא רשימה מקושרת גנרית.

```
public static int getListSum (Node<Integer> list)
{
    if ( list == null )
        return 0;
    return ( list.getValue() + getListSum(list.getNext()));
}
```

ניתן לראות שמימוש הפעולה בנוי על פי ההגדרה הרקורסיבית של רשימה מקושרת. כלומר, סכום החוליות ברשימה מקושרת הוא:

- אם ההפנייה של הרשימה המקושרת list היא **null** – יוחזר הערך 0 אחרת,
- יוחזר הערך שבחוליה הראשונה + סכום האיברים ברשימה המקושרת ללא החוליה הראשונה.

## דוגמה 2: אורך רשימה מקושרת

לפניכם פעולה המקבלת רשימה מקושרת של מחרוזות, ומחזירה את אורך הרשימה מקושרת, כלומר, מספר החוליות ברשימה מקושרת.

```
public static int getListCount (Node<String> list)
{
    if ( list == null )
        return 0;
    return ( 1 + getListCount(list.getNext()));
}
```

גם בדוגמה זו ניתן לראות שמימוש הפעולה בנוי על פי ההגדרה הרקורסיבית של רשימה מקושרת. כפי שהצגנו אותה למעלה. כלומר, אורך הרשימה מקושרת הוא:

- אם ההפנייה של הרשימה המקושרת list היא **null** – יוחזר הערך 1 אחרת,
- יוחזר הערך 1 (עבור החוליה הראשונה) + אורך הרשימה המקושרת ללא החוליה הראשונה

**שימו לב ♥** הפעולה לחישוב אורך הרשימה מקושרת אינה משתמשת בפעולה ייחודית לטיפוס הפרמטר. לכן, אם נרצה לכתוב פעולה רקורסיבית זו על רשימה מקושרת של מספרים שלמים, השינוי היחיד שנבצע הוא בכותרת הפעולה – טיפוס החוליה ישתנה מ- Integer ל- String. טענה זו אינה נכונה עבור פעולת חישוב הסכום, שכן פעולה זו משתמשת בהוראת החיבור, הישימה למספרים, אך אינה ישימה לטיפוסים אחרים. טענות אלה נכונות כמובן גם לגרסאות האיטרטיביות של פעולות אלה. אם כך, מתעוררת השאלה: אם פעולה אינה משתמשת בהוראות ייחודיות לטיפוס קונקרטי כלשהו, כגון הפעולה לחישוב אורך רשימה מקושרת, האם ניתן לכתוב אותה כפעולה גנרית? התשובה היא חיובית. יש בשפה אפשרות כזו, אך פעולות גנריות אינן כלולות ביחידה זו. אנו נמשיך לכתוב פעולות חיצוניות רק עבור טיפוסים קונקרטיים.

## דוגמה 3: מיקום ערך x ברשימה מקושרת

הפעולה שלפניכם מקבלת רשימה מקושרת של מספרים ומספר נוסף x, ומחזירה הפניה לחוליה שבה נמצא המופע הראשון של הערך x. אם x אינו מופיע ברשימה מקושרת, תוחזר הפניה ריקה (null).

שימו לב ♥ כי זהו סוג סריקה שונה מזה שבו דנו קודם – הסריקה נפסקת כאשר נמצאה החוליה המכילה את הערך שחיפשנו.

לפניכם קוד הפעולה:

```

public static Node<Integer> getPosition(Node<Integer> list, int x)
{
    if ( list == null )
        return null;
    if ( list.getValue() == x )
        return list;
    return (getPosition(list.getNext(),x));
}

```

---

משימה: ממשו את הפעולה getPosition( ... ) בגישה איטרטיבית (לא רקורסיבית).

---

#### דוגמה 4: הדפסת רשימה מקושרת

לפניכם פעולה המקבלת רשימה מקושרת של מחרוזות, ומדפיסה מחרוזות המורכבת משרשור המחרוזות שברשימה מקושרת, וביניהן המפריד " -> ":

```

public static void printList (Node<String> list)
{
    if ( list != null )
    {
        System.out.print( " -> " + list.getValue());
        printList(list.getNext());
    }
}

```

## ד. שימוש ברשימה מקושרת לייצוג אוסף

בפרק הקודם הכרנו את המערכת לניהול בית ספרי שתלויה אותנו כדוגמה לאורך יחידת הלימוד. הצגנו את הרשימה הכיתתית וייצגנו אותה בעזרת מערך. כעת, לאחר שהכרנו את הרשימה המקושרת, נציג ייצוג שונה של הרשימה הכיתתית, המשתמש ברשימה מקושרת. ייצוג זה ייתן מענה למגבלות ולסרבול הכרוכים בייצוג רשימה כיתתית בעזרת מערך, כפי שהוצגו בתחילת הפרק. יתרה מזאת, נראה כי כאשר משתמשים ברשימה מקושרת בתוך מחלקה, גם בעיות של שימוש ברשימה מקושרת שעליהן הערנו במהלך הפרק, נפתרות.

ניזכר במחלקה StudentList ונתמקד בחלק הפעולות המוגדרות בה.

### 1.4. המחלקה StudentList

המחלקה StudentList 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:

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

כזכור, הרשימה הכיתתית היא עצם המייצג אוסף של עצמים מטיפוס Student. בייצוג הנוכחי נשתמש בתוכנה מטיפוס חוליה של Student:

```
public class StudentList
{
    private Node<Student> first;
}
```

זוהי התכונה היחידה של רשימת התלמידים. כאשר ערכה **null**, פירוש הדבר הוא שהרשימה ריקה. אחרת, פירוש הדבר שהיא מכילה הפניה לחוליה הראשונה ברשימה המקושרת, שבה כל חוליה מכילה נתוני תלמיד אחד. הרשימה מכילה את כל התלמידים שברשימה הכיתתית.

הפעולה הבונה של **StudentList** תבנה רשימה כיתתית ריקה (כלומר ערכה של התכונה **first** יהיה **null**):

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

## 2.4. פעולת ההוספה

בפרק הקודם התלבטנו אם כדאי להגדיר פעולת הוספה השומרת את הרשימה הכיתתית ממוינת, והגענו למסקנה שנחליט על דרך שמירת הרשימה בהתאם למרבית השימושים הנעשים ברשימה. נתייחס עתה לפעולת הוספה שאינה שומרת את הרשימה ממוינת. הוספה של תלמיד חדש לרשימה תתבצע תמיד בתחילת הרשימה ולכן יעילותה תהיה קבועה.

כיוון שצמצמנו את העולם האמיתי לעולם שאין בו שמות פרטיים כפולים, איננו צריכים לחשוש לקיומם של שני תלמידים בעלי שם זהה בכיתה אחת. לכן אין צורך לבצע בדיקה לפני הוספה של תלמיד חדש לרשימה הכיתתית.

נזכיר כי אנו מניחים גם שהמזכיר/ה האחראי/ת על הרשימה הכיתתית מוודאת לפני ביצוע הפעולה שעדין יש מקום בכיתה. בדיקה זו נחוצה כדי לא למלא כיתה מעבר למכסה המותרת. לבדיקה זו אין קשר לייצוג הרשימה, כיוון שבניגוד למערך, לרשימה המקושרת ניתן להוסיף חוליות ללא הגבלה.

כדי לבצע את ההוספה עלינו ליצור חוליה חדשה המכילה את פרטי התלמיד החדש:

```
Node<Student> temp = new Node<Student>(Student);
```

את החוליה החדשה נכניס לראש הרשימה המקושרת:

```
temp.setNext(this.first);
```

```
this.first = temp;
```

ניתן לקצר את הכתיבה, ולבצע את יצירת החוליה החדשה ואת ההכנסה בשתי פקודות בלבד:

```
Node<Student> temp = new Node<Student>(Student, this.first);
```

```
this.first = temp;
```

ואף ניתן לכתוב הכל בפקודה אחת:

```
this.first = new Node<Student>(Student,this.first);
```

---

ממשו את פעולת ההוספה כך שהרשימה הכיתתית תישמר ממוינת.

---

### 3.4. פעולת הסריקה לאיתור תלמיד

הפעולה `getStudent()` המופיעה בממשק מבצעת סריקה על רשימה מקושרת במטרה לאתר תלמיד על פי שמו. לצורך סריקה זו יש להגדיר משתנה שיעבור על כל החוליות. נגדיר משתנה `pos` מטיפוס חוליה של `Student`. המשתנה ישמש לצורך הסריקה של הרשימה המקושרת. לפניכם מימוש הפעולה. קוד זה מטפל היטב במקרה של רשימה מקושרת ריקה, וכן במקרה שהתלמיד בעל השם הרצוי אינו קיים ברשימה:

```
public Student getStudent (String name)
{
    Node<Student> pos = this.first;
    while (pos != null)
    {
        if(pos.getValue().getName().compareTo(name)== 0)
            return pos.getValue();
        else
            pos = pos.getNext();
    }
    return null;
}
```

כיצד ניתן לשלוף את שם התלמיד מתוך החוליה?

הפעולה `pos.getValue()` מחזירה הפניה לעצם מטיפוס `Student`. ניתן להפעיל על עצם זה את הפעולות של המחלקה `Student`:

```
String name = pos.getValue().getName();
```

כיוון שהשמות הם מטיפוס מחרוזת, אזי כדי להשוות את `name` לשם המועבר כפרמטר, נפעיל את הפעולה `compareTo(...)` המשמשת להשוואה של מחרוזות.

אם התלמיד אינו נמצא, ועדיין לא הגענו לסוף הרשימה, ניתן לקדם את pos באמצעות הפעולה:  
pos = pos.getNext();

#### ד.4. פעולת המחיקה

כדי למחוק מתוך רשימה מקושרת תלמיד ששמו נתון, עלינו לסרוק את הרשימה המקושרת ולמצוא את ההפניה לחוליה המכילה תלמיד שזה שמו. לאחר שנמצאה החוליה, ננתק אותה מהרשימה. כמו בכל סריקה, נגדיר משתנה מטיפוס החוליה שבאמצעותו נבצע את הסריקה, ונפנה אותו לאיבר הראשון ברשימה הכיתתית. אחר כך נסרוק את הרשימה מקושרת, כפי שראינו בסעיף הקודם. אם סרקנו את כל הרשימה מקושרת ולא מצאנו את השם, פירוש הדבר ששם זה אינו קיים ברשימה. במקרה זה הפעולה תסתיים כאשר `pos == null`, אחרת `pos` יפנה לחוליה שבה תלמיד בשם זה. עתה יש לבצע הוצאה של החוליה ש-`pos` מצביע אליה.

כדי להוציא חוליה כלשהי (פרט לראשונה) עלינו להשתמש בהפניה לחוליה שלפניה. האם יש צורך לסרוק מחדש את כל הרשימה המקושרת כדי למצוא את החוליה הקודמת ל-`pos`? סריקה נוספת תייקר את יעילות פעולת ההוצאה.

---

א. ממשו את פעולת המחיקה בעזרת שתי סריקות. מה סדר הגודל של יעילות פעולה זו?  
ב. ממשו את פעולת המחיקה באופן שונה, כך שתבצע רק סריקה אחת של הרשימה המקושרת.  
מה סדר הגודל של יעילות הפעולה במימוש זה?

---

#### ד.5. דיון

נדון בקצרה בכמה נקודות שמעלה דוגמת הרשימה הכיתתית.  
ראשית, רשימה כיתתית יכולה להיות ריקה, ולמעשה היא נוצרת ריקה. אין כאן כל סתירה. המחלקה רשימה כיתתית מיוצגת על ידי תכונה המפנה לרשימה מקושרת, כאשר יש בה תלמידים. כאשר הרשימה הכיתתית ריקה, ערך התכונה הוא `null`.

שנית, נתקלנו בקושי להגדיר פעולת הכנסה לרשימה מקושרת שתוכל לבצע הכנסה של חוליה חדשה כחוליה ראשונה ברשימה. כאן, אין כל קושי לכתוב פעולה של הוספה לרשימה כיתתית המכניסה חוליה חדשה למקום הראשון ברשימה. הסיבה לכך היא שהמשתנה היחיד המכיל הפניה לרשימה הוא התכונה `first`. כיוון שפעולת ההכנסה פנימית למחלקה, המשתנה מוכר לה, והיא יכולה לשנות תכונה זו כך שהיא תצביע על החוליה החדשה. במצב זה לא ייתכן שמשתנה אחר ימשיך להפנות לחוליה שהיא עכשיו השנייה.

באופן דומה, אין קושי לכתוב פעולת מחיקה מן הרשימה הכיתתית, גם אם המחיקה כרוכה בהוצאת החוליה הראשונה.



**לסיכום:** מחלקה מגדירה ייצוג לרעיון מופשט ומממשת פעולות על עצמים שלה. כאשר אנו בוחרים לייצג את האוסף על ידי מבנה כגון רשימה מקושרת, אנו רשאים להוסיף לייצוגים החוקיים גם את הערך **null** במשמעות המתאימה לנו. ניתן לפעול על התכונה המכילה את הרשימה המקושרת ולשנותה לפי הצורך.

בניסוח מדויק פחות, אנו יכולים לומר שלרשימה מקושרת, כמבנה נתונים יש מגבלות, אך כאשר אורזים רשימה בתוך מחלקה, כייצוג פנימי, המגבלות אינן קיימות.

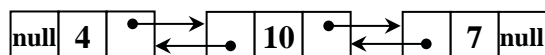
שימו לב שבפרק זה הצגנו ייצוג ומימוש חדשים לרשימה הכיתתית, StudentList. כיוון שאין שינוי בממשק המחלקה, ברור שהרשימה הכיתתית היא טיפוס נתונים מופשט. הממשק של המחלקה אינו חושף בפני המשתמש את דרך הייצוג והמימוש של המחלקה, ולכן המשתמש כלל אינו מודע לשינוי הייצוג ואינו מוטרד ממנו.

## ה. מבנים אחרים מבוססי חוליות

הרעיון של שימוש בהפניה אל עצם נוסף מאותו הטיפוס מאפשר ליצור שרשרת בין כמה עצמים מאותו הטיפוס, ובכך לבטא קשר בין כמה איברים של אוסף אחד. טכניקה זו יכולה לשמש אותנו גם לייצוג אוספים אחרים שבהם קיימים קשרים מורכבים יותר בין האיברים. נציג בקצרה שני שימושים ברעיון ההפניות אל עצמים מטיפוס המחלקה עצמה. בפרקים הבאים נעמיק את ההיכרות עם מבנים אלה ועם משמעויותיהם.

### ה.1. רשימה מקושרת דו-כיוונית

להרשימה המקושרת שבה דנו בפרק קוראים לפעמים רשימה מקושרת חד-כיוונית (singly linked list), כיוון שניתן לנוע עליה רק בכיוון אחד. ניתן להגדיר מחלקת חוליה שבה שתי הפניות, ולהשתמש בחוליות כאלה לבניית **רשימה מקושרת דו-כיוונית (doubly-linked list)**, שבה בכל חוליה יש הפניה אחת המובילה לחוליה הבאה ברשימה (next), ואחת המובילה לחוליה הקודמת ברשימה (prev). בשימוש כזה, נתייחס לחוליה כאל חוליה דו-כיוונית. רשימה מקושרת דו-כיוונית תיראה כך:



מכיוון שלחוליה שתי הפניות (Binary) המפנות לכיוונים (Directions) שונים, נקרא לה BiDirNode ונגדיר אותה כחוליה גנרית.



---

חשבו על עץ משפחה שבו כל אדם מחזיק הפניות לשני הוריו. כיצד נאתר את הסבתא מצד האם של אדם נתון?

---

### ה.3. מבט קדימה

יש הרבה יישומים שבהם יש צורך לנהל אוספים; רק את מקצתם תיארנו לעיל. כיוון שכך חשוב ללמוד כיצד להגדיר מחלקות שהעצמים שנוצרים מהן מייצגים אוספים, והמחלקות מגדירות עבור האוספים את הפעולות המתאימות.

במחשבה ראשונה, היינו רוצים לדעת כיצד להגדיר מחלקות עבור אוספים ספציפיים, כגון הרשימה הכיתתית.

במחשבה שנייה, נרצה יותר מזה. רשימה כיתתית ואוספים ספציפיים אחרים שהזכרנו מיועדים למטרות מסוימות. ואולם, יש הרבה דמיון בין האוספים השונים. למשל, סביר שאוספים של לקוחות בנק, מנויי תיאטרון או מנויי מכון כושר, מאורגנים באופן דומה, ויכולים לבצע אותן פעולות כלליות של הוספה, של מחיקה, של שינוי ושל חיפוש. אם כך, חשוב יותר להגדיר מחלקות המייצגות סוגי אוספים נפוצים. ספרייה של מחלקות כאלה תהווה ארגז כלים שיעמוד לרשותנו לצורך תכנות יישומים שונים. למשל, ממחלקה המגדירה "תור" נוכל לייצר עצם לייצוג כל תור ספציפי הדרוש ליישום: תור של שיחות טלפון ממתנות, תור של תהליכים במפעל ייצור וכדומה.

כדי להבין כיצד נוכל להגדיר מחלקות כלליות כאלה לייצוג סוגי אוספים, נחשוב על האוספים שמנינו וננסה לעמוד על תכונותיהם, ועל הדומה והשונה שבהם.

תכונה חשובה באוסף היא הצורך בקיום סדר. בחלק מהאוספים שהזכרנו, אין חשיבות לסדר האיברים באוסף, ניתן לחשוב על האוסף כעל קבוצה של איברים. אך לעתים קרובות האוסף ממוין ומהווה **סדרה**. כך הדבר כאשר מסדרים את רשימת התלמידים בכיתה בסדר אלפביתי, או את רשימת התנועות של לקוח בבנק לפי תאריכי הביצוע שלהן.

ספרייה תחזיק בדרך כלל את רשימת הספרים המושאלים ממוינת על פי שמות הספרים, אך יש עניין גם בסידור הרשימה לפי תאריכי השאלה, או תאריכי החזרה. דוגמאות נוספות של אוספים ממוינים הן רשימת כתובות וטלפונים של מכרים, ממוינת לפי סדר אלפביתי של השמות, ותיקיות של מסרי דואר אלקטרוני, הממוינות לפי תאריך קבלה, או לפי קריטריון אחר הנקבע על ידי המשתמש.

תכונה מסוג אחר, שאינה קשורה לסדר, היא אופן השימוש באוסף. באוספים רבים יש צורך לעבור על כל האיברים באוסף ולבצע עבור כל אחד פעולה מסוימת. פעולה כזו נקראת **סריקה**. באוספים רבים נזדקק לאפשרות לבצע **חיפוש**, למשל המשתמש מספק שם ומקבל כתשובה את פרטי האדם (הלקוח, המנוי) שזה שמו. על אוספים כאלה אנו חושבים במונחים של **מיפוי מִמְפָּתָח** (כגון שם) לערך הקשור אליו (למשל כתובת או מספר טלפון).

השוו בין תור למאגר לקוחות. במאגר לקוחות של בנק או במאגר מנויי תיאטרון, הלקוח נשאר זמן רב. יש לספק באוספים אלה פעולות של הוספה והוצאה של לקוח, אך הפעולות החשובות הן חיפוש לקוח לפי פרטים מזהים, או מעבר על כל מאגר הלקוחות לצורך ביצוע פעולה עבור כל לקוח. תור אף הוא אוסף, ולמעשה סדרה שבה הסדר נקבע לפי מועד הכניסה לתור. אולם, בניגוד למאגרים, איבר בתור נשאר בו לרוב זמן קצר בלבד, עד שהוא מגיע לראש התור ויוצא ממנו. בדרך כלל אין צורך בפעולות חיפוש או סריקה של תור. שתי הפעולות החשובות עבור תור הן הכנסה והוצאה, והן מתבצעות במקומות קבועים – הכנסה לזנב התור והוצאה מראשו. אם כן, לתור יש אפיון חשוב שאינו קיים בסוגי המאגרים האחרים: המקום להכנסה או להוצאה אינו נקבע על ידי ערך פרמטר המועבר לפעולה, וגם לא על ידי ערכי נתונים (כמו בפעולות הכנסה והוצאה על אוסף ממוין), אלא הוא נקבע בהגדרת האוסף. תור הוא סוג אוסף שיש לו **נוהל גישה** המגביל את הפעולות עליו.

על האוספים שהזכרנו עד כה ניתן לחשוב כקבוצות, שבהן אין קשר בין האיברים השונים. אפשר לחשוב על האוספים גם כסדרות שבהן יש קשר חד-ממדי בין האיברים. נזכיר כי קיימים אוספים שבהם מבנה הקשרים בין האיברים מורכב יותר. בעץ משפחה (אילן יוחסין) האיברים מייצגים בני אדם, וכולם ממשפחה אחת. אם באוסף מיוצגים רק קשרים בין הורים לילדיהם, אזי הוא באמת מבנה היררכי, וכינויו "עץ משפחה" מוצדק. לעתים קרובות אוסף זה מייצג גם קשרי נישואים ומבנהו מורכב יותר, אם כי עדיין מקובל לכנותו "עץ משפחה".

בארגונים רבים, רשומות העובדים וקשרי הניהול מאורגנים אף הם כאוסף שצורתו כתרשים עץ. מפעלים רבים מייצרים מוצרים מורכבים – מכלולים – כגון מנועים או מטוסים. מכלול מורכב מרכיבים פשוטים יותר, שחלקם אף הם מכלולים וחלקם רכיבים אטומיים. מאגר המכלולים וקשרי ההכלה ביניהם נקרא "עץ המוצר". הפעולות לטיפול במלאי ובתזמון הייצור, כולן מתבססות על עץ המוצר. למשל, חישוב מחירו של מנוע מתבצע על ידי סיכום מחירי החלקים המרכיבים אותו, בתוספת עלות עבודת ההרכבה. פעולת חישוב עלותו של מנוע דורשת סריקה של עץ המוצר שלו, מהעלים (החלקים הבסיסיים) ועד לשורש (המוצר המוגמר).

בפרקים הבאים נעסוק במחלקות המגדירות סוגים כלליים של אוספים. מחלקות אלה הן כעין ארגז כלים היכול לשמש אותנו בבניית יישומים מתקדמים.

## 1. סיכום

- רשימה מקושרת היא מבנה נתונים דינמי המאפשר שמירת אוסף נתונים, כך שכל נתון נשמר בתוך חוליה. מידע נוסף שנשמר בכל חוליה מאפשר את חיבור החוליות זו לזו. הרשימה המקושרת אינה עצם מטיפוס מחלקה עצמאית.
- כדי לגשת לנתון ברשימה יש להגיע תחילה לחוליה שבה הוא שמור. לכל רשימה מקושרת יש התחלה – ההפניה אל החוליה הראשונה ברשימה.
- שימוש באוספי נתונים הוא עניין שבשגרה ביישומי מחשב רבים. לצורך טיפול באוסף שלו תכונות מסוימות, מגדירים מחלקה מתאימה. איברי האוסף מיוצגים על ידי תכונה של המחלקה. במקום להשתמש במערך לתכונה זו ניתן להשתמש ברשימה מקושרת. מחלקת אוסף שמשמשת ברשימה לאחסון איברי האוסף מגדירה את נוהל הגישה המיוחד לכל אוסף (הפרוטוקול): פעולות לבניית האוסף, פעולות לחיפוש באוסף ופעולות לאחזור ערכים השמורים בו. בשימוש בעצמים מטיפוס המחלקה נפתרות בעיות השינוי של הרשימה המקושרת שהוצגו בפרק, כיוון שהרשימה המקושרת ארוזה בתוך המחלקה.
- מנגנון הגנריות מאפשר להגדיר מחלקה שבה טיפוס הערך אינו נקבע בזמן הגדרת המחלקה אלא רק בעת השימוש בה, למשל בעת יצירת עצם. בפרק זה הצגנו רשימה מקושרת גנרית ושימושיה.
- ביחידת לימוד זו, פעולות המקבלות פרמטרים חייבות לקבל פרמטרים קונקרטיים ולא פרמטרים גנריים. גם ערכי החזרה של הפעולות יהיו קונקרטיים.
- במהלך הפרקים הבאים נבנה ארגז כלים שיכיל הגדרות שונות שישמשו אותנו לטיפול באוספים שונים. לצורך ייצוג אוספים אלה נשתמש לעתים ברשימה מקושרת.

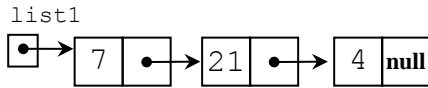
## מושגים

collection	אוסף
genericity	גנריות
node	חוליה
type wrapper classes	מחלקות עוטפות
external method	פעולה חיצונית
internal method	פעולה פנימית
doubly linked list	רשימה מקושרת דו-כיוונית
(singly) linked list	רשימה מקושרת (חד-כיוונית)

# תרגילים

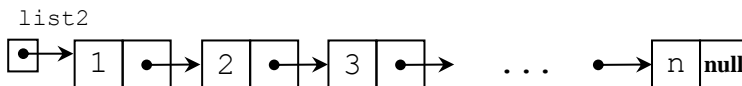
## שאלה 1

א. בנו את הרשימה המקושרת הזו:



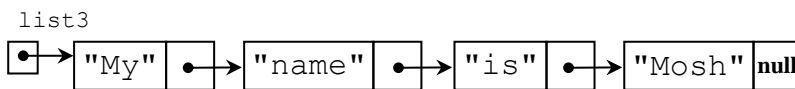
ב. בנו רשימה מקושרת שבה מאוחסנים המספרים השלמים מ-1 ועד n.

n הוא מספר אקראי בין 2 ל-100:



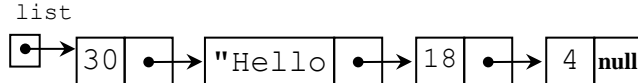
ג. בנו רשימה מקושרת של מחרוזות המייצגת משפט שייקלט. כל מילה תישמר בחוליה נפרדת.

לדוגמה, אם נקלט המשפט "My name is Moshe" יש לבנות את הרשימה המקושרת הזו:



## שאלה 2

בנו את הרשימה המקושרת הזו:



אם לא הצלחתם, הסבירו מהי הבעיה לבנות את הרשימה.

## שאלה 3

א. הוסיפו למחלקה `IntNode`, המופיעה בפרק, פעולה בונה מעתיקה. תזכורת: פעולה בונה מעתיקה היא פעולה המקבלת עצם כלשהו ומאתחלת עצם חדש מאותו הטיפוס כך שיכיל בדיוק את ערכיו הפנימיים של העצם המתקבל כפרמטר.

ב. הסבירו מה המשמעות של פעולה בונה מעתיקה הנכתבת כפעולה פנימית במחלקה

`.Node < T >`

## שאלה 4

ממשו את הפעולה הזו:

```
public static Node<Integer> createRandomList (int numNodes)
```

הפעולה מחזירה רשימה מקושרת של מספרים שלמים שבה `numNodes` ערכים אקראיים בין 0

ל-100. הניחו ש: `numNodes > 0`.

## שאלה 5

הפעולה הבאה מקבלת שתי רשימות מקושרות:

```
public static void change(Node<Integer> list1,
                          Node<Integer> list2)
{
    Node<Integer> pos = list1;

    while(pos.hasNext())
        pos = pos.getNext();
    pos.setNext(list2);
}
```

- א. איך ייראו שתי הרשימות, list1 ו-list2, בתום הפעולה? הדגימו על שתי רשימות.
- ב. כתבו את טענת היציאה של הפעולה.
- ג. נתחו את יעילות הפעולה.

## שאלה 6

נתונה פעולה רקורסיבית המקבלת רשימה מקושרת:

```
public static boolean secret(Node<Integer> list)
{
    if (!list.hasNext())
        return true;

    int x = list.getValue();
    int y = list.getNext().getValue();

    if (x*y > 0)
        return false;

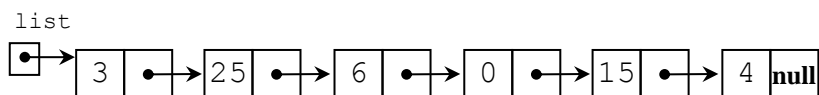
    return secret(list.getNext());
}
```

- א. תנו דוגמה לרשימה מקושרת שעבורה זימון הפעולה secret(...) יחזיר true, ותנו דוגמה נוספת לרשימה מקושרת שעבורה זימון הפעולה secret(...) יחזיר false.

ב. כתבו את טענות הכניסה והיציאה של הפעולה. ציינו במפורש בתיעוד מהי הנחת היסוד העומדת בבסיס הגדרת הפעולה.

## שאלה 7

נתונה רשימה מקושרת של מספרים שלמים המוחזקת על ידי `list`:



קטע התוכנית שלפניכם מבצע שינוי כלשהו על הרשימה המקושרת:

```
Node<Integer> pos1 = list;
```

```
Node<Integer> pos2 = null;
```

```
Node<Integer> pos3 = null;
```

```
while(pos1 != null)
```

```
{
```

```
    pos2 = pos1.getNext();
```

```
    pos1.setNext(pos3);
```

```
    pos3 = pos1;
```

```
    pos1 = pos2;
```

```
}
```

```
list = pos3;
```

א. עקבו אחר קטע התוכנית וציירו את הרשימה המקושרת המתקבלת בסיומו.

ב. כתבו מה מבצע קטע התוכנית.

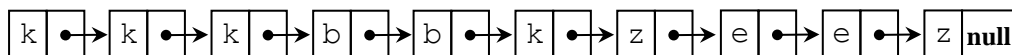
## שאלה 8

ממשו את הפעולה הזו:

```
public static void compressSequences (Node<Character> list)
```

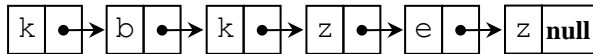
הפעולה מקבלת רשימה מקושרת של תווים. הפעולה תצמצם רצפי תווים, כך שלא יופיעו תווים זהים ברצף. התו הראשון בכל רצף יישאר וכל השאר יוסרו. יש לשמור על סדר התווים.

לדוגמה, לאחר זימון הפעולה עבור הרשימה המקושרת האלה:





הרשימה המקושרת תיראה כך :

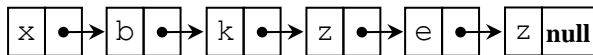


### שאלה 9

לפניכם פעולה רקורסיבית המקבלת רשימה מקושרת של תווים ומחזירה מחרוזת :

```
public static String mystery (Node<Character> list)
{
    if(!list.hasNext())
        return list.getValue().toString();
    return mystery(list.getNext()) + "," + list.getValue();
}
```

א. מהי המחרוזת שתוחזר לאחר זימון הפעולה `mystery(...)` עבור הרשימה המקושרת הזו :



ב. כתבו את טענת היציאה של הפעולה המתוארת בשאלה.

ג. ממשו את הפעולה ללא שימוש ברקורסיה.

### שאלה 10

רשמו את טענת היציאה של הפעולה שלפניכם :

```
public static void mystery(Node<Integer> list)
{
    int temp;
    Node<Integer> pos1 = list;
    Node<Integer> pos2;

    while(pos1.hasNext())
    {
        pos2 = pos1.getNext();
        while(pos2 != null)
        {
            if(pos1.getValue() > pos2.getValue())

```

```

    {
        temp = pos1.getValue();
        pos1.setValue(pos2.getValue());
        pos2.setValue(temp);
    }
    pos2 = pos2.getNext();
}
pos1 = pos1.getNext();
}
}

```

## שאלה 11

לפניכם הפעולה:

```

public static Node<Integer> merge(Node <Integer> list1,
                                   Node <Integer> list2)

```

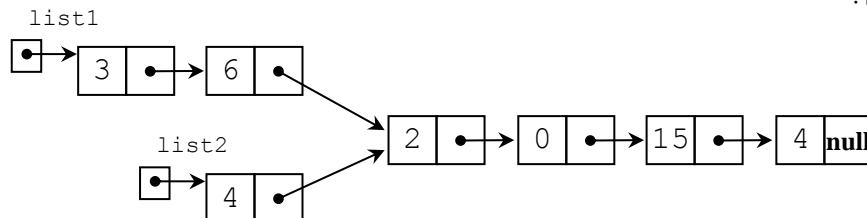
הפעולה מקבלת שתי רשימות מקושרות של חוליות עם מספרים שלמים, ממוינות בסדר עולה. הפעולה מבצעת מיזוג של שתי הרשימות. עליכם לממש את הפעולה בשתי צורות שונות:

א. הפעולה תחזיר הפניה לרשימה מקושרת חדשה שהיא מיזוג של שתי הרשימות.

ב. הפעולה תחזיר הפניה לרשימה מקושרת שהיא מיזוג של שתי הרשימות תוך שימוש בחוליות הקיימות של שתי השרשרות – ללא יצירת חוליות חדשות.

## שאלה 12

שתי רשימות מקושרות של מספרים מחוברות ביניהן. חוליה כלשהי בכל אחת משתי הרשימות מפנה אל חוליה משותפת, החל בחוליה זו הרשימות זהות. האיור שלפניכם מתאר שתי הרשימות המחוברות באופן זה:

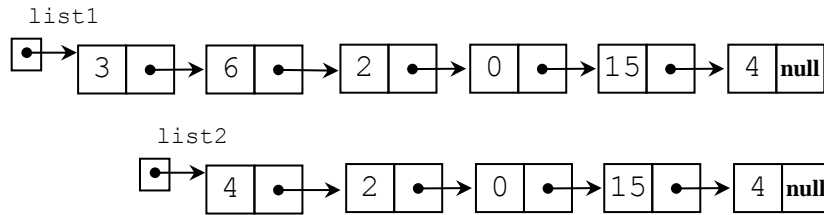


שימו לב כי מספר החוליות הנפרדות בכל אחת מהרשימות אינו זהה בהכרח.

א. ממשו את הפעולה:

```
public static void disconnect(Node <Integer> list1,  
    Node <Integer> list2)
```

הפעולה מקבלת שתי רשימות מקושרות המחוברות ביניהן על ידי חוליה כלשהי, ומנתקת אותן. בתום הפעולה תכיל כל אחת מהרשימות בסופה את האיברים המשותפים. האיור הבא מתאר את שתי הרשימות מהאיור הקודם לאחר ביצוע פעולת הניתוק:



ב. נתחו את יעילות הפעולה disconnect(...) שכתבתם בסעיף א.

### שאלה 13

חזרו ובצעו מחדש את כל הסעיפים שבדף עבודה 5 מפרק 6 – "ספר טלפונים". השתמשו ברשימה מקושרת לייצוג אוסף אנשי הקשר בספר הטלפונים.