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 מחזיקה 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 רשימות 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 בַּמקום המתאים להם לפי קריטריון המיון. הבה נחשוב על מערך של 10,000 תאים, המכיל 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 לכך, אם מוחקים את האיבר שבתא ה-4,950, יש להזיז את 4,850 האיברים הנמצאים מעליו במערך, כדי "לסתום את החור". ראינו אם כן 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

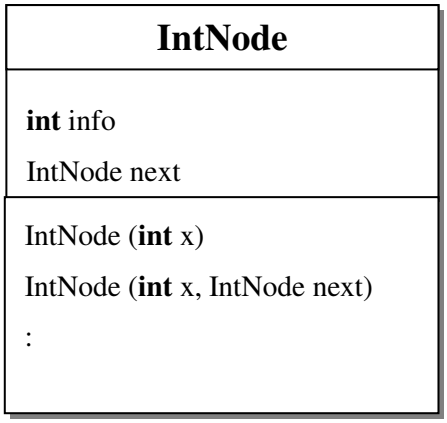
בפרק הקודם, שבו למדנו על הפניות, הכרנו את הגישה של "שרשור עצמים", המשתמשת בהפניה לעצם, כדרך לקישור בין עצמים. נאמץ גישה זו ונעסוק בעצמים שכל אחד מהם מכיל נתון והפניה אחת או יותר לעצמים מאותו הטיפוס. על ידי שרשור של עצמים כאלה נוכל לבנות מבנים לייצוג אוספים של נתונים ולבצע עליהם פעולות כגון חיפוש נתון,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

א.1. חוליה המכילה מספר שלם

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



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

- התכונה info – בה נשמר הנתון המספרי.
- 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.

IntNode (**int** x)	הפעולה בונה חוליה. הערך של החוליה הוא x, ואין לה חוליה עוקבת
IntNode (**int** x, IntNode next)	הפעולה בונה חוליה. הערך של החוליה הוא x והחוליה העוקבת לה היא next. ערכו של next יכול להיות **null**
int getInfo()	הפעולה מחזירה את הערך השמור בחוליה
IntNode getNext()	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה מחזירה **null**
void setInfo (**int** x)	הפעולה משנה את הערך השמור בחוליה להיות x
void setNext (IntNode next)	הפעולה משנה את החוליה העוקבת להיות next. ערכו של next יכול להיות **null**
String toString()	הפעולה מחזירה מחרוזת המתארת את החוליה

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

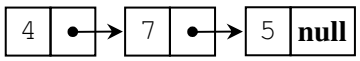
להלן מימוש המחלקה IntNode:

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

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

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

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

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

	public int getInfo()
	{
		return(this.info);
	}

	public void setInfo(int x)
	{
		this.info = x;
	}

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

א.2. שרשרת חוליות

המחלקה IntNode מאפשרת ליצור חוליות, לחבר אותן זו לזו באמצעות הפעולה setNext(…), וכך לקבץ יחד קבוצת חוליות של מספרים שלמים.

לדוגמה: אוסף המספרים 4, 7, 5 השמור בקבוצה של חוליות, ייראה כך:



למבנה כזה מקובל לקרוא **שרשרת חוליות**. בשרשרת, כל חוליה שומרת נתון והפניה לחוליה הבאה. החוליה הבאה גם היא שומרת נתון והפניה לחוליה הבאה אחריה וכך הלאה, עד לחוליה האחרונה בשרשרת, השומרת את הנתון האחרון באוסף. מאחר שזו החוליה האחרונה בשרשרת, לא תהיה בה הפניה לחוליה נוספת – ולכן ערך ההפניה בה יהיה **null**.

שרשרת החוליות המתוארת באיור שלעיל מורכבת מ-3 חוליות. בחוליה הראשונה נשמר הנתון המספרי 4 והפניה לחוליה השנייה. בחוליה השנייה שמור הנתון המספרי 7 והפניה לחוליה השלישית. בחוליה השלישית שמור הנתון המספרי 5 והפניה ריקה **null** המציינת שזו החוליה האחרונה בשרשרת.

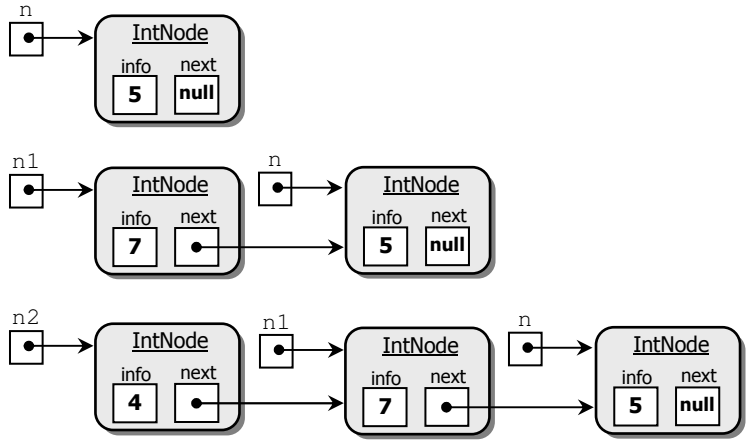
כאמור לעיל, פעולות המחלקה IntNode מאפשרות ליצור שרשרות של חוליות שבהן שמורים מספרים שלמים. בהמשך נראה כי פעולות המחלקה מאפשרות גם ליצור מבנים שאינם שרשרות, אך ברובו של הפרק נתמקד בשרשרות. בהמשך סעיף זה נראה כי שרשרת החוליות היא מבנה נתונים דינמי, הנותן מענה לחסרונות המערך שמנינו בראשית הפרק: ניתן להוסיף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

א.1.2. בניית שרשרת חוליות של מספרים שלמים

נתבונן בקטע התוכנית שלפנינו, הבונה שרשרת חוליות שמאוחסן בה אוסף הנתונים 4, 7, 5:

```
IntNode n  = new IntNode(5);
IntNode n1 = new IntNode(7,n);
IntNode n2 = new IntNode(4,n1);
```

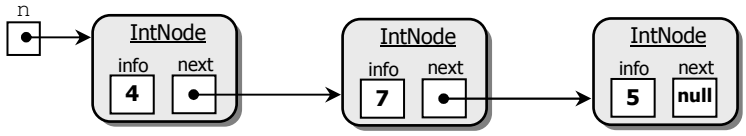
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב, את ביצוע קטע התוכנית:



ניתן לקצר את קטע התוכנית הנזכר לעיל ולכותבו כך:

```
IntNode n = new IntNode(4, new IntNode(7, new IntNode(5)));
```

תרשים העצמים של התוכנית המקוצרת ייראה כעת כך:



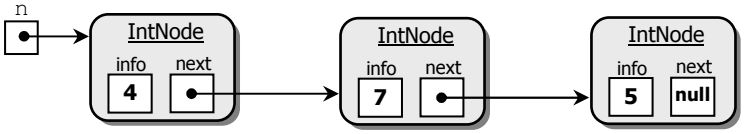
נוסף ליתרון שבכתיבה מקוצרת, דרך כתיבה זו לא נוצרות הפניות נוספות לחוליות שבאמצע השרשרת. כפי שכבר הזכרנו בפרק 6 שדן בהפניות, זהו הישג טוב, שכן ריבוי הפניות עלול לגרום לבעיות שונות. נרחיב את הדיון בנושא זה בהמשך הפרק.

שימו לב, שאת ההפניה לחוליה הראשונה, השמורה במשתנה n, חייבים לשמור. אם הפניה זו תימחק, לא נוכל לגשת יותר לשרשרת החוליות. על פי הגדרת השפה, במקרה זה השרשרת לא תהיה קיימת יותר. מנגנון איסוף הזבל, garbage collection, ישחרר את השרשרת מהזיכרון כשיזהה שהיא עצם שאין אליו הפניה.

א.2.2. הכנסת חוליה לשרשרת חוליות

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

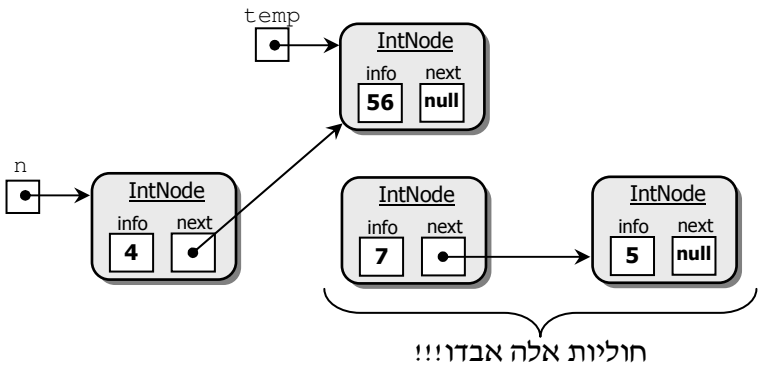
נחזור אל שרשרת החוליות שאנו עוסקים בה:



נניח שנרצה להכניס לשרשרת זו את המספר 56 לאחר החוליה הראשונה כחוליה שנייה בשרשרת. נעקוב אחרי שלבי ההכנסה של הערך החדש לשרשרת.

נגדיר חוליה חדשה שהערך בה הוא 56 ונעדכן את החוליה הקודמת, הראשונה בשרשרת, כך שתפנה אל החוליה החדשה:

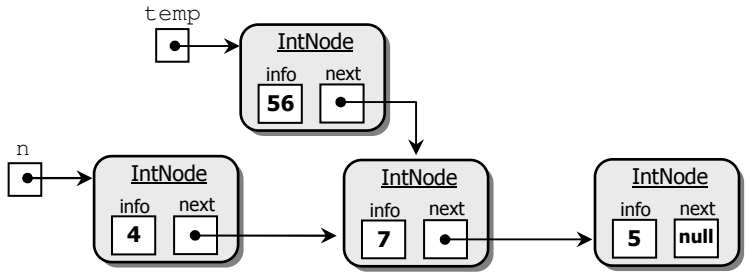
```
IntNode temp = new IntNode(56);
n.setNext(temp);
```



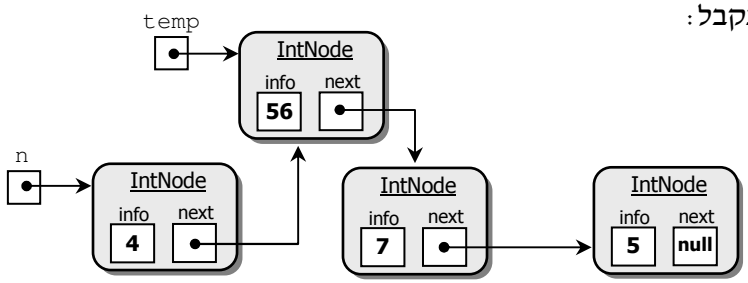
בשלב זה נרצה לבצע את החיבור האחרון בשרשרת: הפניית החוליה החדשה אל זו העוקבת לה, שהייתה שנייה בשרשרת המקורית. אך ערך ההפניה הנחוץ, שהיה שמור בתכונה next של החוליה הראשונה, נמחק כאשר הוכנסה אליו ההפניה לחוליה החדשה! כעת אין ביכולתנו לבצע את החיבור המתאים. נראה שהיינו צריכים לבצע את כל מהלך ההוספה "מהסוף 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(56,n.getNext());
n.setNext(temp);
```

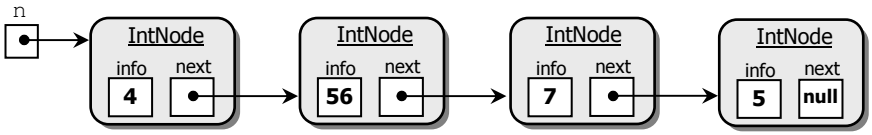
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב, את ביצוע קטע התוכנית. לאחר ההוראה הראשונה נקבל:



לאחר ההוראה השנייה נקבל:



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



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

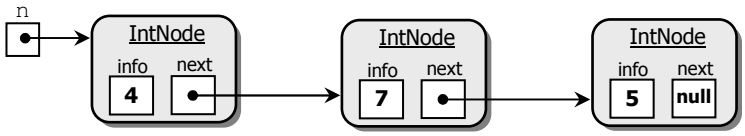
קיום פעולת ההכנסה כפי שתוארה מראה ששתי המגבלות של המערך שתיארנו לעיל אינן קיימות בשרשרת חוליות:

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 שרשרת החוליות פותרת גם את המגבלה השלישית, ומאפשרת לאחסן אוספים בעלי מבנה מורכב שאינו סדרתי.

א.3.2. הוצאת חוליה משרשרת החוליות

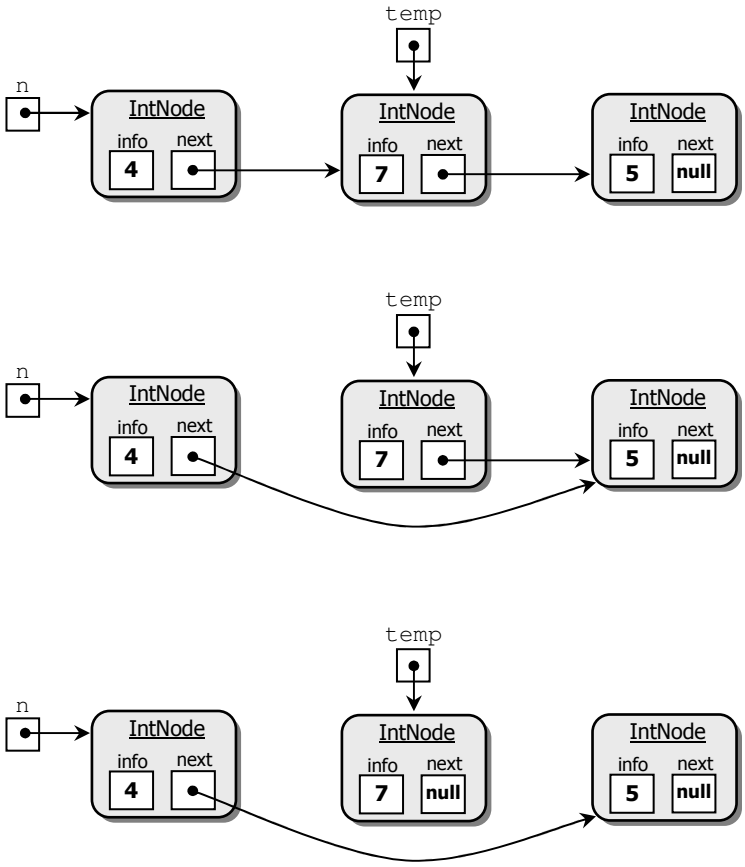
נתבונן שוב בשרשרת החוליות שאנו פועלים עליה:



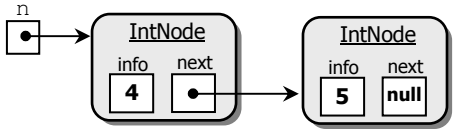
קטע התוכנית שלפניכם, מוציא את החוליה השנייה (שערכה 7) מתוך שרשרת החוליות:

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

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



נוח יותר יהיה להסתכל על שרשרת החוליות המתקבלת לאחר הוצאת המספר 7 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 בשרשרת המקורית. ההכנסה מתבצעת **אחרי** חוליה קיימת שהפניה אליה נתונה לנו. להפניה זו אנו מגיעים על ידי סריקת השרשרת 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 בשרשרת. רק אז נוכל לבצע את ההוצאה כפי שתיארנו.

א.4.2. מעבר על שרשרת החוליות

לעתים קרובות מאוד נרצה לסרוק שרשרת, כלומר לעבור על החוליות שבה, מתחילתה עד סופה, או עד שנמצא את החוליה המבוקשת. להלן תבנית של קטע קוד המבצע סריקה מלאה של שרשרת החוליות, המוחזקת על ידי chain. כיוון שידוע שבסוף שרשרת החוליות קיימת חוליה שערכו של העוקב שלה הוא **null**, נסתמך על ערך זה בתנאי העצירה של הלולאה:

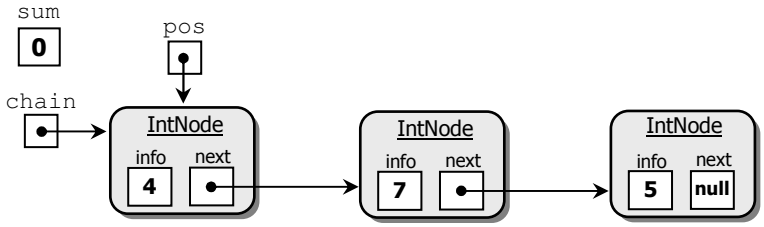
```
IntNode pos = chain;

while(pos != null)
{
   // pos.getInfo() ביצוע פעולה עם ערך החוליה
   pos = pos.getNext();
}
```

ניישם את התבנית הזו בקטע הקוד שלפניכם, המחשב את סכום המספרים השמורים בשרשרת חוליות ש-chain מפנה אליה, החל בחוליה הראשונה וכלה באחרונה:

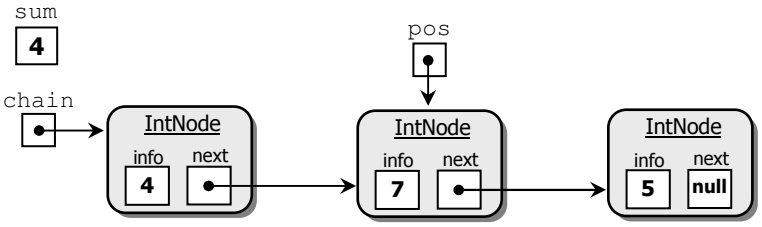
```
int sum = 0;
IntNode pos = chain;

while(pos != null)
{
   sum = sum + pos.getInfo();       // סיכום ערכי החוליות
   pos = pos.getNext();
}
```

ההפניה החדשה, pos, מסייעת במעבר על השרשרת. תרשימי העצמים שלפניכם מתארים, שלב אחר שלב, את ביצוע קטע התוכנית:

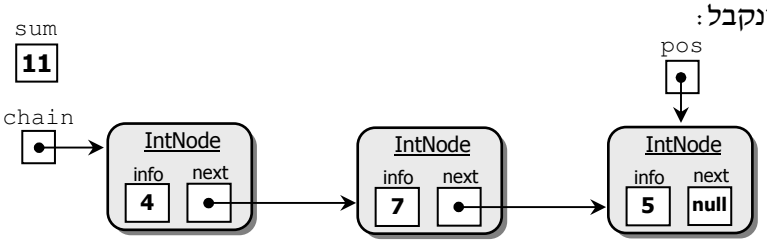
זהו המצב ההתחלתי, אחרי אתחול של pos ושל sum, ולפני שנכנסים ללולאה:



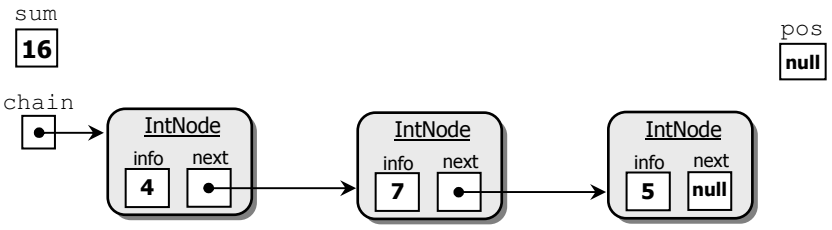
מכיוון ש- pos שונה מ-**null**, תתבצע הלולאה פעם ראשונה, ונקבל:



הלולאה תתבצע פעם נוספת ונקבל:



לאחר ביצוע הלולאה בפעם השלישית נקבל:



בסיום קטע הקוד, pos יקבל את הערך **null** (כי בשרשרת זו לא קיימת חוליה נוספת), הלולאה תיפסק, וקטע התוכנית יסתיים. במשתנה sum נמצא עכשיו הערך 16 השווה לסכום המספרים בשרשרת. סכום זה ישמש כערך ההחזרה של הפעולה.

? כתבו קטע תוכנית המוצא את המספר הגדול ביותר בשרשרת חוליות המוחזקת על ידי chain.

א.3. שרשרת חוליות כפרמטר לפעולות

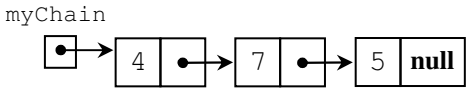
עד כה הדגמנו בעזרת קטעי קוד כמה שינויים אפשריים של שרשרת חוליות. אם השרשרת נוצרה בתוך הפעולה הראשית, אזי קטעי הקוד הללו שולבו אף הם בפעולה זו, אחרי בניית השרשרת. גישה זו נוחה להתנסות ראשונית, אך כדי להשתמש בשרשרות חוליות באופן כללי, נרצה לכתוב פעולות היכולות להתבצע על שרשרת כלשהי. לפעמים, די להעביר לפעולה כזו הפניה לחוליה שעליה היא צריכה לפעול; במקרים אחרים, נעביר גם הפניה לשרשרת כולה. שימו לב ששרשרת החוליות אינה טיפוס נתונים, אלא היא מבנה שנוצר מחוליות, והוא אינו מוגדר בעזרת מחלקה משל עצמו. כדי להעביר שרשרת כפרמטר, נעביר הפניה לחוליה הראשונה שלה. כלומר הפניה לשרשרת גם היא הפניה מטיפוס חוליה. היא מפנה לחוליה הראשונה בשרשרת, ומבחינתנו היא "מפנה לשרשרת חוליות". שרשרת חוליות מכילה לפחות חוליה אחת, ולכן בכל הדוגמאות שנעסוק בהן תקפה ההנחה הסמויה המלווה את יחידת הלימוד, לפיה ערך הפרמטר המחזיק את השרשרת בזימון הפעולה אינו שווה **null**, ואיננו צריכים לבדוק את ההנחה בתוך מימוש הפעולה. נתבונן בכמה דוגמאות. במהלך הדיון נדון במקרי קצה שונים, ובשגיאות אפשריות.

א.1.3. סכום של שרשרת חוליות

נכתוב פעולה המקבלת שרשרת חוליות של מספרים שלמים (על ידי העברת הפניה לחוליה הראשונה בשרשרת), ומחזירה את סכום המספרים בשרשרת:

```
public static int getChainSum(IntNode chain)
{
      int sum = 0;

      while(chain != null)
      {
            sum = sum + chain.getInfo();
            chain = chain.getNext();
      }

      return sum;
}
```

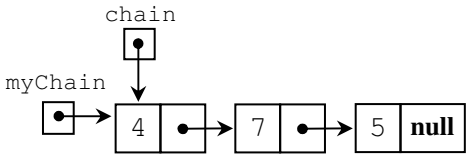
נתונה שרשרת חוליות המוחזקת על ידי המשתנה myChain:



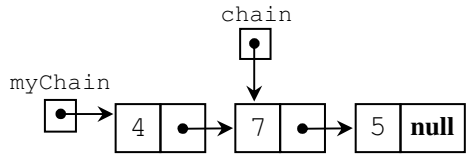
כדי לחשב את סכום המספרים בשרשרת החוליות myChain ניתן לזמן את הפעולה getChainSum(...) שכתבנו ולשלוח אליה את שרשרת החוליות myChain. הזימון ייראה כך:

```
int totalSum = getChainSum(myChain);
```

ברגע הזימון הערך שנמצא במשתנה myChain, שהוא הפניה לחוליה הראשונה בשרשרת, מועתק לפרמטר chain שבכותרת הפעולה getChainSum(...). לכן נוצר מצב שבו לחוליה הראשונה בשרשרת יש שתי הפניות, כמתואר באיור שלפניכם:



כעת גוף הפעולה מתחיל להתבצע. המשתנה chain עובר על פני שרשרת החוליות. לאחר ביצוע הסבב הראשון של הלולאה, המשתנה chain יכיל הפניה לחוליה השנייה:



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

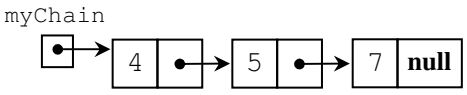
א.2.3. הכנסת ערך לשרשרת חוליות ממוינת

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

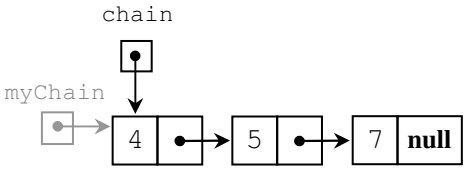
```
public static void insertIntoSortedChain(IntNode chain, int x)
{
      ...
}
```

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

נתונה שרשרת החוליות הממוינת myChain:

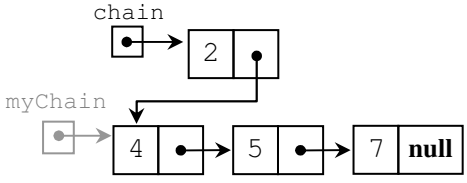


נזמן את הפעולה כך: insertIntoSortedChain (myChain, 2) במטרה להכניס את הערך 2 לשרשרת החוליות myChain הממוינת. הערך 2 הוא הקטן ביותר מבין הערכים 4, 5, 7, ולכן הוא אמור להיכנס כערך הראשון בשרשרת החוליות. בעת זימון הפעולה, הועברה לפרמטר chain ההפניה לתחילת שרשרת החוליות הממוינת. מרגע המעבר לפעולה, ההפניה החיצונית myChain אינה מוכרת עוד, ולכן היא נצבעת בצבע שונה:



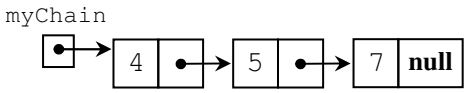
כעת הפעולה insertIntoSortedChain(…) עובדת רק עם הפרמטר chain כמחזיק שרשרת החוליות. הפעולה כלל אינה מכירה בתוכה את ההפניה myChain (לכן באיורים אלה היא מסומנת בצבע שונה).

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



איור זה מראה את הבעיה. המשתנה chain מכיל הפניה לחוליה הראשונה החדשה, בעוד המשתנה myChain לא עודכן בתוך הפעולה. בתום הפעולה insertIntoSortedChain(…) הפרמטר המקומי

chain מתבטל! אנו חוזרים למצב ההתחלתי בו myChain הוא ההפניה היחידה לשרשרת, אך myChain לא השתנה. בסיכום כל התהליך, הערך 2 לא הוכנס לשרשרת החוליות כמתבקש:



? הסבירו מדוע הבעיה שתיארנו אינה מתעוררת אם הכנסת הערך מתבצעת במקום שאינו הראשון בשרשרת.

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

פתרון לבעיות שהעלינו יוצג בהמשך הפרק ובפרקים הבאים.

א.3.3. מציאת ערך מקסימלי בקטע של שרשרת חוליות

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

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

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

      pos = pos.getNext();
      while(pos != null)
      {
            if(pos.getInfo() > maxPos.getInfo())
                  maxPos = pos;
            pos = pos.getNext();
      }

      return maxPos;
}
```

? נתונה שרשרת חוליות המוחזקת על ידי myChain.

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

ב. כתבו את הזימון הנדרש כדי שהפעולה תחזיר את מיקום הערך הגדול ביותר בשרשרת myChain כולה.

סיכום ביניים:

שרשרת חוליות אינה עצם מטיפוס נתונים המוגדר על ידי מחלקה, אלא מבנה המורכב מעצמים מטיפוס חוליה. שרשרת מוחזקת במשתנה על ידי הפניה לחוליה הראשונה שלה.

לכן, כשאנו אומרים שפעולה מקבלת שרשרת חוליות, כוונתנו לכך שהפעולה מקבלת הפניה לחוליה הראשונה בשרשרת. כמו בכל היחידה, גם כאן מתקיימת ההנחה הסמויה שהפרמטר המועבר אינו שווה **null** (פירוש הדבר למעשה שהשרשרת מכילה לפחות חוליה אחת).

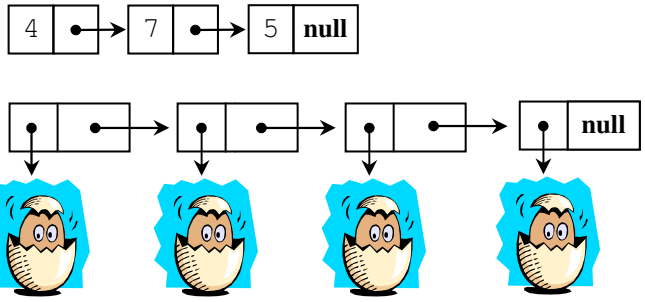
בנוסף, פעולות (כמו למשל הכנסה והוצאה) אינן יכולות לשנות את החוליה הראשונה כך ששינוי זה ייראה מחוץ לפעולה. הרחבה והעמקה בנושא תמצאו בהמשך הפרק ובפרקים הבאים.

ניתן לבצע פעולות הכנסה והוצאה על שרשרת חוליות (במגבלה שתיארנו) ביעילות, במספר קטן וקבוע של פעולות. אין הגבלה על גודל שרשרת (פרט למגבלת הזיכרון במחשב). לכן, שרשרת החוליות היא מבנה נתונים הפותר את מגבלותיו של המערך, שכן מבנה הנתונים הוא דינמי ותומך בביצוע יעיל של פעולות עדכון.

אפשר להסתכל על הפניה לחוליה בשרשרת חוליות כהפניה לחוליה בתוך שרשרת קיימת או כהפניה לתת-שרשרת. ההסתכלות תלויה בהקשר של הבעיה הנידונה.

ב. חוליה גנרית

כפי שראינו, כל חוליה בשרשרת 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.

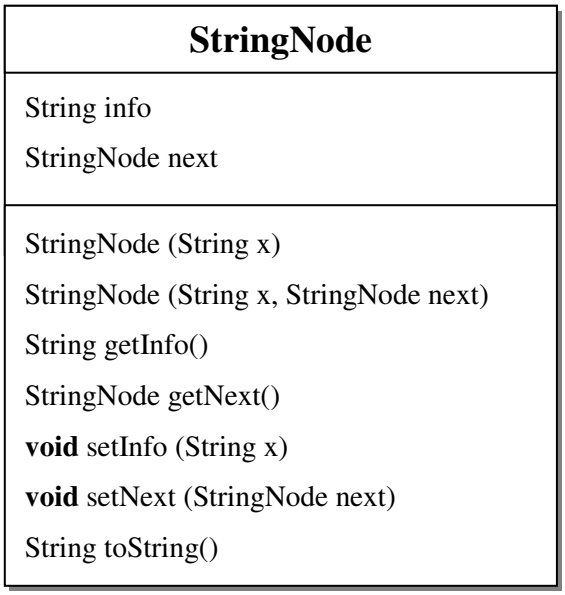


לא נדרשים שינויים משמעותיים כדי להגדיר חוליה שטיפוס הערך שלה שונה ממספר שלם. למשל, כדי להגדיר מחלקת חוליה שהערך השמור בה הוא מחרוזת, נוכל להעתיק את ההגדרה של המחלקה IntNode ולבצע את השינויים האלה:

1. שינוי טיפוס התכונה info לטיפוס String.
2. שינוי שם 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 פרמטרים או ערכי החזרה. פרט לכך לא יידרש שום שינוי. נוכל לבנות ממחלקה זו שרשרות חוליות בדיוק כמו שבנינו שרשרות מחוליות של IntNode.

להלן תרשים UML של המחלקה החדשה:



האם בכל פעם שנרצה ליצור שרשרת חוליות שערכיה מטיפוס 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 ה**גנריות** (**genericity**), שיאפשר לנו להימנע 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>
{
      //(המימוש יתואר בהמשך) מימוש המחלקה באמצעות הטיפוס T
}
```

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 טיפוסי 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>("abc");
```

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 ממימוש רעיון הגנריות, אך נושא זה רחב ולא נעסוק בו כאן). בהמשך נראה שלמרות מגבלה זו ניתן לייצר שרשרת חוליות של מספרים שלמים או של כל טיפוס בסיסי אחר, ולהשתמש בה ללא קושי.

לפניכם ממשק החוליה הגנרית.

ממשק 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 getInfo()	הפעולה מחזירה את הערך של החוליה
Node<T> getNext()	הפעולה מחזירה את החוליה העוקבת. אם אין חוליה עוקבת, הפעולה מחזירה **null**
void setInfo (T x)	הפעולה משנה את הערך השמור בחוליה ל-x.
void setNext (Node<T> next)	הפעולה משנה את החוליה העוקבת ל-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 פעולת ממשק שמתייחסת לערך החוליה יש שימוש בסימון T כטיפוס ערך החוליה. קודם שנראה את מימוש המחלקה, נבחן שימושים שונים שניתן לעשות בה.

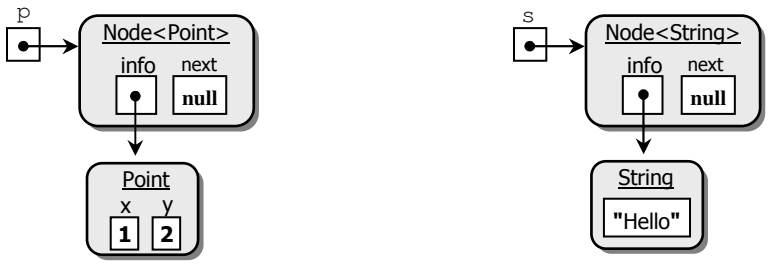
ב.1. שימוש בחוליה הגנרית

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

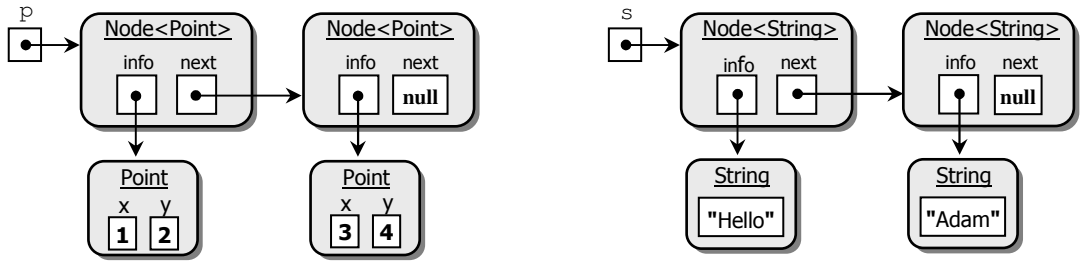
```
public static void main(String[] args)
{
      // שלב 1
      Node<Point> p = new Node<Point>(new Point(1,2));
      Node<String> s = new Node<String>("Hello");

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

תרשימי העצמים שלפניכם מתארים את בניית החוליות והשרשרות בתוכנית:

לאחר שלב 1:



לאחר שלב 2:



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

ב.2. מחלקות עוטפות

כאמור לעיל, טיפוס הערך בחוליה חייב להיות טיפוס של עצם. דרישה זו מעלה בעיה: כיצד נייצר שרשרת חוליות 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 המערכת, בהתאם לצורך, והמתכנת אינו נדרש לכתוב המרות בתכניתו.

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

כדי לייצר ולהשתמש בשרשרת חוליות 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 a = n.getInfo();
```

בשורה הראשונה ייצרנו 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 a. שוב התבצעה המרה אוטומטית, הפעם מעצם מטיפוס Integer לערך מטיפוס **int**.

ב.3. פעולות פנימיות וחיצוניות

פעולה הכלולה בהגדרת מחלקה 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 העצם המפעיל את הפעולה מצוין 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 בדוגמאות 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.1.3 כך שתסכום שרשרת חוליות של מספרים שלמים, כותרת הפעולה שהייתה במקור:

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

תיראה כך:

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

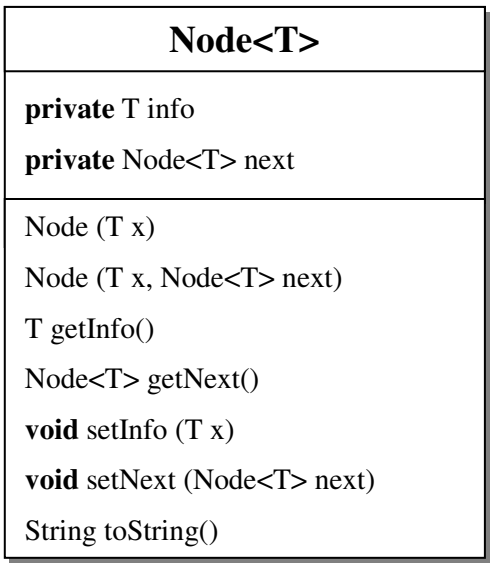
לפרמטרים שונים ניתן לבחור טיפוסים קונקרטיים שונים, כמו בכותרת זו:

```
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 הגנרית

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 חוליה גנרית:



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

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

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

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

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

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


	public T getInfo()
	{
		return this.info;
	}

	public void setInfo(T x)
	{
		this.info = x;
	}

	public String toString()
	{
		return this.info.toString();
	}
}
```

כפי שניתן לראות בהגדרת טיפוס הערך (info) 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.6. 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. שרשרת חוליות כמבנה רקורסיבי

ההגדרה שהשתמשנו בה לשרשרת חוליות, ולפיה השרשרת היא אוסף של חוליות, הייתה בסיס טוב לתכנות איטרטיבי של פעולות על שרשרות, כפי שעשינו עד כה. אך ניתן להתייחס אל שרשרת החוליות גם כאל מבנה רקורסיבי. להלן הגדרה רקורסיבית של שרשרת החוליות:

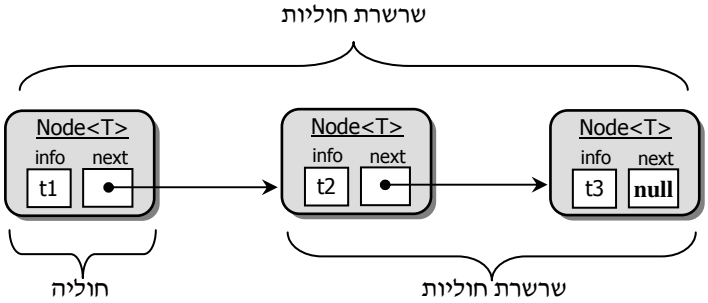
שרשרת חוליות היא:

- חוליה יחידה

  או

- חוליה שיש בה הפניה לשרשרת חוליות

ההגדרה הרקורסיבית מתאפשרת משום שבחוליה קיימת תכונה בשם next, המכילה הפניה לחוליה הבאה שהיא מאותו הטיפוס. כלומר כל חוליה בשרשרת החוליות היא בעצמה התחלה של שרשרת חוליות.



ההגדרה הרקורסיבית מאפשרת לכתוב פעולות רקורסיביות על שרשרת חוליות.

דוגמה 1: חישוב סכום בשרשרת חוליות

כדי לחשב סכום מספרים שלמים בשרשרת חוליות יש צורך לסרוק את כל החוליות בשרשרת ולסכם את ערכיהן. בסעיף א.4.2. הצגנו קטע תוכנית שביצע משימה זו תוך שימוש בלולאת **while**. חישוב סכום המספרים בשרשרת חוליות ניתן למימוש גם באופן רקורסיבי, בהסתמך על ההגדרה הרקורסיבית של שרשרת חוליות שהוצגה לעיל.

להלן פעולה רקורסיבית המקבלת שרשרת חוליות המכילות מספרים, ומחזירה את סכום המספרים בשרשרת. כפי שציינו ביחידת לימוד זו, פעולה המקבלת שרשרת חוליות תקבל רק שרשרת מטיפוס קונקרטי ולא שרשרת גנרית. כאן, כמו בפעולות הקודמות, אנו מניחים שההפניה המועברת כפרמטר אינה **null**, שכן בשרשרת יש לפחות חוליה אחת.

```
public static int getChainSum(Node<Integer> chain)
{
      if(chain.getNext() == null)
            return chain.getInfo();
      return chain.getInfo() + getChainSum(chain.getNext());
}
```

ניתן לראות שמימוש הפעולה בנוי על פי ההגדרה הרקורסיבית של שרשרת חוליות. כלומר, סכום החוליות בשרשרת חוליות הוא:

- אם השרשרת היא חוליה יחידה , כלומר ערך התכונה next של הפרמטר chain הוא **null** – יוחזר הערך שבחוליה הראשונה (והיחידה)

אחרת:

- יוחזר הערך שבחוליה הראשונה + סכום שרשרת החוליות ללא החוליה הראשונה

דוגמה 2: אורך שרשרת חוליות

לפניכם פעולה המקבלת שרשרת חוליות של מחרוזות, ומחזירה את אורך השרשרת, כלומר את מספר החוליות שבשרשרת.

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

גם בדוגמה זו ניתן לראות שמימוש הפעולה בנוי על פי ההגדרה הרקורסיבית של שרשרת החוליות, כפי שהצגנו אותה למעלה. כלומר, אורך השרשרת הוא:

- אם השרשרת היא חוליה יחידה – יוחזר 1

אחרת,

- יוחזר הערך 1 (עבור החוליה הראשונה) + אורך שרשרת החוליות ללא החוליה הראשונה

שימו לב: הפעולה לחישוב אורך השרשרת אינה משתמשת בפעולה ייחודית לטיפוס הפרמטר. לכן, אם נרצה לכתוב פעולה רקורסיבית זו על שרשרת חוליות של מספרים שלמים, השינוי היחיד שנבצע הוא בכותרת הפעולה – טיפוס החוליה ישתנה מ-String ל-Integer. 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 לגִרסאות האיטרטיביות של פעולות אלה.

אם כך, מתעוררת השאלה: אם פעולה אינה משתמשת בהוראות ייחודיות לטיפוס קונקרטי כלשהו, כגון הפעולה לחישוב אורך שרשרת, 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 בשרשרת חוליות

הפעולה שלפניכם מקבלת שרשרת חוליות של מספרים ומספר נוסף x, ומחזירה הפניה לחוליה שבה נמצא המופע הראשון של הערך x. אם x אינו מופיע בשרשרת, 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> chain, int x)
{
      if(chain.getInfo() == x)
            return chain;

      if(chain.getNext() == null)
            return null;
      return getPosition(chain.getNext(), x);
}
```

? ממשו את הפעולה getPosition(...) בגישה איטרטיבית (לא רקורסיבית).

דוגמה 4: הדפסת שרשרת חוליות

הפעולה שלפניכם מקבלת שרשרת חוליות של מחרוזות, ומדפיסה מחרוזת המורכבת משרשור המחרוזות שבחוליות השרשרת, וביניהן המפריד " -> ":

```
public static void printChain(Node<String> chain)
{
      System.out.print(chain.getInfo());

      if (chain.getNext() != null)
      {
            System.out.print(" -> ");
            printChain(chain.getNext());
      }
}
```

ד. שימוש בשרשרת חוליות לייצוג אוסף

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

ניזכר במחלקה StudentList ונתמקד בחלק מהפעולות המוגדרות בה.

ד.1. 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 ברשימה הפעולה מחזירה **null**
Student getStudent (String name)	הפעולה מחזירה את התלמיד ששמו name. אם התלמיד אינו קיים ברשימה, הפעולה מחזירה **null**
String toString()	הפעולה מחזירה מחרוזת המתארת דף קשר כיתתי ממוין בסדר אלפביתי, באופן זה: <name1> <tel1> <name2> <tel2>

כזכור, הרשימה הכיתתית היא עצם המייצג אוסף של עצמים מטיפוס Student. בייצוג הנוכחי נשתמש בתכונה 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 ראשונה בשרשרת חוליות, שבה כל חוליה מכילה נתוני תלמיד אחד. השרשרת 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;
}
```

ד.2. 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 שהמזכירה האחראית על הרשימה הכיתתית מוודאת לפני ביצוע הפעולה שעדיין יש מקום בכיתה. בדיקה זו נחוצה כדי לא למלא כיתה מעבר למכסה המותרת. לבדיקה זו אין קשר לייצוג הרשימה, כיוון שבניגוד למערך, לשרשרת 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 שרשרת החוליות:

```
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 הכול בפקודה אחת:

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

? ממשו את פעולת ההוספה כך שהרשימה הכיתתית תישמר ממוינת.

ד.3. פעולת הסריקה לאיתור תלמיד

הפעולה getStudent() המופיעה בממשק מבצעת סריקה על שרשרת החוליות 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 שרשרת החוליות.

לפניכם מימוש הפעולה. קוד זה מטפל היטב במקרה של רשימה 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.getInfo().getName().compareTo(name)== 0)
                  return pos.getInfo();
            else
                  pos = pos.getNext();
      }
      return null;
}
```

כיצד ניתן לשלוף את שם התלמיד מתוך החוליה?

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

```
String name = pos.getInfo().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 תלמיד ששמו נתון, עלינו לסרוק את הרשימה ולמצוא את ההפניה לחוליה המכילה תלמיד שזה שמו. לאחר שנמצאה החוליה, ננתק אותה מהשרשרת. כמו בכל סריקה, נגדיר משתנה מטיפוס החוליה שבאמצעותו נבצע את הסריקה, ונפנה אותו לאיבר הראשון ברשימה הכיתתית. אחר כך נסרוק את הרשימה, כפי שראינו בסעיף הקודם. אם סרקנו את כל הרשימה 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 שרשרת החוליות 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 שתתבצע רק סריקה אחת של שרשרת החוליות. מה סדר הגודל של יעילות הפעולה במימוש זה?

ד.5. דיון

נדון בקצרה בכמה נקודות שמעלה דוגמת הרשימה הכיתתית.

ראשית, נזכיר כי אין משמעות לשרשרת חוליות ריקה. שרשרת חוליות תמיד מכילה לפחות חוליה אחת. בניגוד לכך, רשימה כיתתית יכולה להיות ריקה, ולמעשה היא נוצרת ריקה. אין כאן כל סתירה. המחלקה רשימה כיתתית מיוצגת על ידי תכונה המפנה לשרשרת חוליות, כאשר יש בה תלמידים. כאשר הרשימה הכיתתית ריקה, ערך התכונה הוא **null**.

שנית, נתקלנו בקושי להגדיר פעולת הכנסה לשרשרת חוליות שתוכל לבצע הכנסה של חוליה חדשה כחוליה ראשונה בשרשרת. כאן, אין כל קושי לכתוב פעולה של הוספה לרשימה כיתתית המכניסה חוליה חדשה למקום הראשון בשרשרת. הסיבה לכך היא שהמשתנה היחיד המכיל הפניה לשרשרת 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 וממשת פעולות על עצמים שלה. כאשר אנו בוחרים לייצג את האוסף על ידי מבנה כגון שרשרת חוליות, אנו רשאים להוסיף לייצוגים החוקיים גם את הערך **null** במשמעות המתאימה לנו. ניתן לפעול על התכונה המכילה את שרשרת החוליות ולשנותה לפי הצורך.

בניסוח מדויק פחות, אנו יכולים לומר שלשרשרת, כמבנה נתונים יש מגבלות, אך כאשר אורזים שרשרת 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.

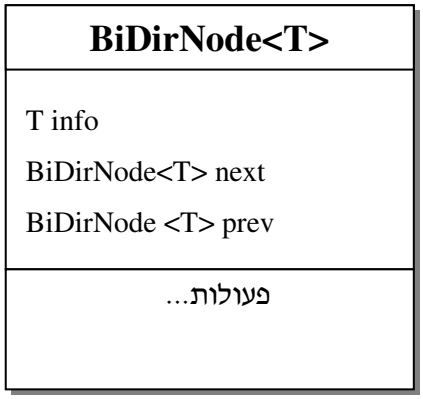
ה. מבנים אחרים מבוססי חוליות

הרעיון של שימוש בהפניה אל עצם נוסף מאותו הטיפוס מאפשר ליצור שרשור 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. שרשרת דו-כיוונית

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

מכיוון שלחוליה שתי הפניות (**Bi**nary) המפנות לכיוונים (**Dir**ections) שונים, נקרא לה BiDirNode ונגדיר אותה כחוליה גנרית. לפניכם תרשים UML של חוליה דו-כיוונית כזו:



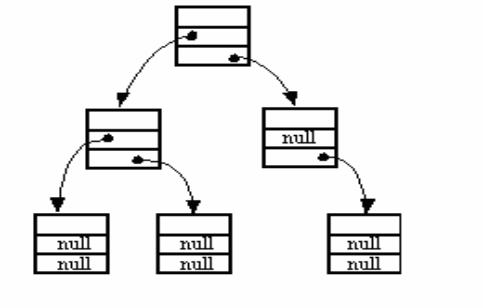
פעולות מסוימות ניתן לממש באופן יעיל יותר על שרשרת חוליות דו-כיוונית מאשר על שרשרת חד-כיוונית, ונעסוק בכך בהמשך.

? חשבו על רשימה כיתתית דו-כיוונית ממוינת בסדר אלפביתי של שמות התלמידים. חוליה המכילה תלמיד st תכיל הפניה next לחוליה שהתלמיד בה הוא העוקב ל-st בסדר האלפביתי, ותכיל הפניה נוספת prev לתלמיד שקודם לו לפי סדר זה. האם תהליך המחיקה של תלמיד מתוך הרשימה יהפוך להיות פשוט יותר?

ה.2. מבנים היררכיים

ניתן להשתמש בחוליה שלה שתי הפניות לאותו הטיפוס באופן שונה, כרכיב יסוד של מבנה היררכי. חוליה תכיל הפניה אחת, left, לחוליה שעומדת באוסף לשמאלה), והפניה נוספת, right, לחוליה שלימינה). שרשור של חוליות דו-כיווניות כאלה ייצור מבנה היררכי המזכיר אילן יוחסין, מעין תרשים של בעלי תפקידים בארגון או במפעל. גם בשרשור זה נעסוק בהרחבה בפרקים הבאים.

המבנה שלפניכם הוא דוגמה למבנה שיכול להתקבל משרשור חוליות כאלה:



? 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 ממתינות, 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** מַמַפְתֵחַ (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.

ו. סיכום

- שרשרת חוליות 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. שרשרת החוליות אינה עצם מטיפוס מחלקה עצמאית.
- כדי לגשת לנתון בשרשרת יש להגיע תחילה לחוליה שבה הוא שמור. לכל שרשרת חוליות יש התחלה – ההפניה אל החוליה הראשונה בשרשרת. סוף השרשרת היא החוליה האחרונה.
- 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 בשרשרת חוליות. מחלקת אוסף שמשתמשת בשרשרת 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 שרשרת החוליות שהוצגו בפרק, מכיוון שהשרשרת 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 שרשרת חוליות 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 בשרשרת חוליות.

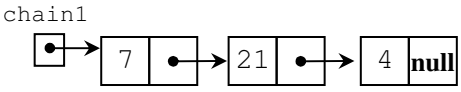
מושגים

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

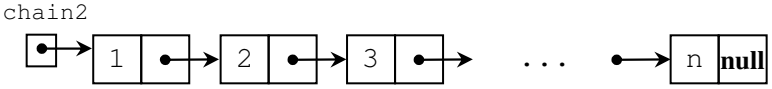
תרגילים

שאלה 1

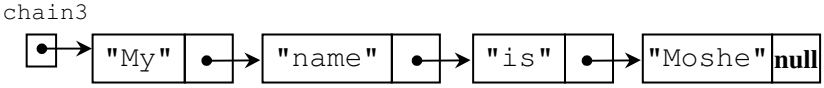
א. בנו את שרשרת החוליות הזו:



ב. בנו שרשרת חוליות שבה מאוחסנים המספרים השלמים מ-1 ועד n. n הוא מספר אקראי בין 2 ל-100:

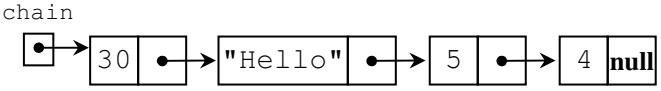


ג. בנו שרשרת חוליות של מחרוזות המייצגת משפט שייקלט. כל מילה תישמר בחוליה נפרדת. לדוגמה, אם נקלט המשפט "My name is Moshe" יש לבנות את שרשרת החוליות הזו:



שאלה 2

בנו את שרשרת החוליות הזו:



אם לא הצלחתם, הסבירו מהי הבעיה לבנות את השרשרת.

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> createRandomChain(int numNodes)
```

הפעולה מחזירה שרשרת חוליות של מספרים שלמים שבה numNodes ערכים אקראיים בין 0 ל-100. הניחו ש: numNodes > 0.

שאלה 5

הפעולה הבאה מקבלת שתי שרשרות של חוליות:

```
public static void change(Node<Integer> chain1,
                                        Node<Integer> chain2)
{
      Node<Integer> pos = chain1;

      while(pos.getNext() != null)
            pos = pos.getNext();
      pos.setNext(chain2);
}
```

א. איך ייראו שתי השרשרות, chain1 ו-chain2, בתום הפעולה? הדגימו על שתי שרשרות.

ב. כתבו את טענת היציאה של הפעולה.

ג. נתחו את יעילות הפעולה.

שאלה 6

נתונה פעולה רקורסיבית המקבלת שרשרת חוליות:

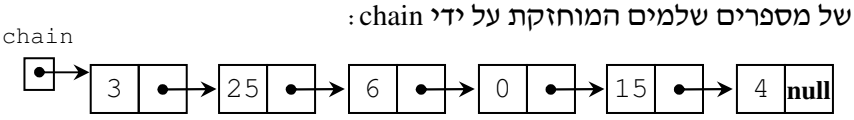
```
public static boolean secret(Node<Integer> chain)
{
      if (chain.getNext() == null)
            return true;

      int x = chain.getInfo();
      int y = chain.getNext().getInfo();

      if (x*y > 0)
            return false;

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

א. תנו דוגמה לשרשרת חוליות שעבורה זימון הפעולה secret(...) יחזיר **true**, ותנו דוגמה נוספת לשרשרת חוליות 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 שרשרת חוליות של מספרים שלמים המוחזקת על ידי chain:



קטע התוכנית שלפניכם מבצע שינוי כלשהו על שרשרת החוליות:

```
Node<Integer> pos1 = chain;
Node<Integer> pos2 = null;
Node<Integer> pos3 = null;

while(pos1 != null)
{
      pos2 = pos1.getNext();
      pos1.setNext(pos3);
      pos3 = pos1;
      pos1 = pos2;
}
chain = pos3;
```

א. עקבו אחר קטע התוכנית וציירו את שרשרת החוליות המתקבלת בסיומו.

ב. כתבו מה מבצע קטע התוכנית.

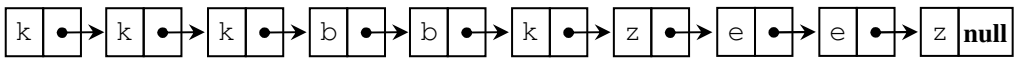
שאלה 8

ממשו את הפעולה הזו:

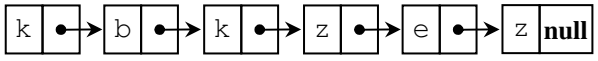
```
public static void compressSequences(Node<Character> chain)
```

הפעולה מקבלת שרשרת חוליות 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 שרשרת החוליות האלה:



שרשרת החוליות תיראה כך:

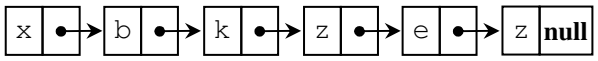


שאלה 9

לפניכם פעולה רקורסיבית המקבלת שרשרת חוליות של תווים ומחזירה מחרוזת:

```
public static String mystery(Node<Character> chain)
{
     if(chain.getNext() == null)
          return chain.getInfo().toString();
     return mystery(chain.getNext()) + "," + chain.getInfo();
}
```

א. מהי המחרוזת שתוחזר לאחר זימון הפעולה mystery(…) עבור שרשרת החוליות 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> chain)
{
      int temp;
      Node<Integer> pos1 = chain;
      Node<Integer> pos2;

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

שאלה 11

לפניכם הפעולה:

```
public static Node<Integer> merge(Node<Integer> chain1,
                                  Node<Integer> chain2)
```

הפעולה מקבלת שתי שרשרות של חוליות של מספרים שלמים, ממוינות בסדר עולה. הפעולה מבצעת מיזוג של שתי השרשרות.

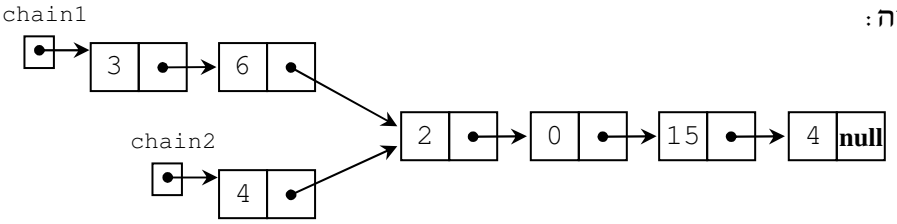
עליכם לממש את הפעולה בשתי צורות שונות:

א. הפעולה תחזיר הפניה לשרשרת חוליות חדשה שהיא מיזוג של שתי השרשרות.

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

שאלה 12

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

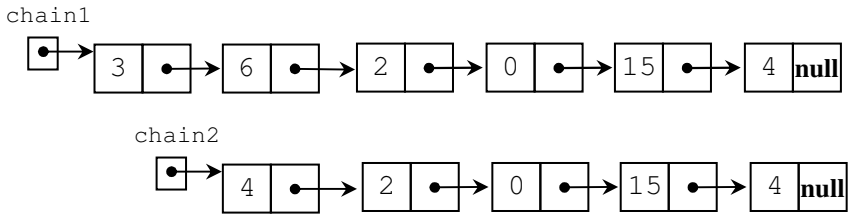


שימו לב כי מספר החוליות הנפרדות בכל אחת מהשרשרות אינו זהה בהכרח.

א. ממשו את הפעולה:

```
public static void disconnect(Node<Integer> chain1,
                              Node<Integer> chain2)
```

הפעולה מקבלת שתי שרשרות המחוברות ביניהן על ידי חוליה כלשהי, ומנתקת אותן. בתום הפעולה תכיל כל אחת מהשרשרות בסופה את האיברים המשותפים. האיור הבא מתאר את שתי השרשרות 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 בשרשרת חוליות לייצוג אוסף אנשי הקשר בספר הטלפונים.