משרד החינוך
המזכירות הפדגוגית – אגף מדעים

הטכניון – מכון טכנולוגי לישראל
מרכז המורים הארצי למדעי המחשב

מינהלת מל"מ, המרכז הישראלי
לחינוך מדעי טכנולוגי ע"ש עמוס דה שליט

פיתוח חומרי עזר למורי תיכון

פיתוח:

מורים מובילים

בהנחיית ד"ר תמר לפידות ועפרה ברנדס

פרויקט 7 פיתוח חומרי עזר למורי תיכון (100 עמודים)

מרכז מורים ארצי במקצוע מדעי המחשב. הפרויקט מבוצע עפ"י מכרז 9/7.2013.
הפרויקט מבוצע עבור האגף לתכנון ופיתוח תוכניות לימודים, המזכירות הפדגוגית, משרד החינוך

יצא לאור במימון ובפיקוח המזכירות הפדגוגית, אגף מדעים במשרד החינוך
ומינהלת מל״מ, המרכז הישראלי לחינוך מדעי טכנולוגי ע״ש עמוס דה-שליט

כל הזכויות שמורות למשרד החינוך©

תוכן עניינים

א. פרק 7 במבני נתונים: יצוג אוספים,

מותאם לתכנית הלימודים החדשה 2

ב. פרק 10 במבני נתונים: עץ בינרי,

מותאם לתכנית הלימודים החדשה 44

ג. שאלות מבחן מותאמות לתכנית החדשה (יסודות ומבני נתונים) 90

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 בַּמקום 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

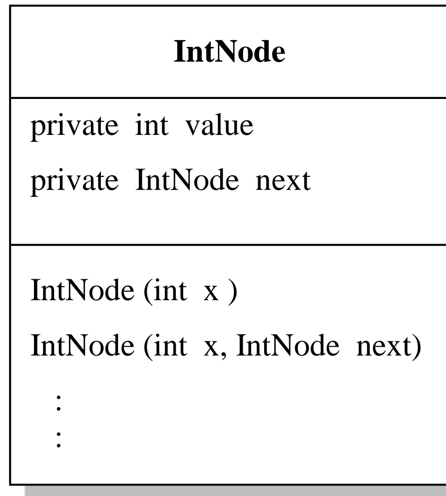
בפרק הקודם, שבו למדנו על הפניות, הכרנו את הגישה של "שרשור עצמי", המשתמשת בהפניה לעצם, כדרך לקישור בין עצמים. נאמץ גישה זו ונעסוק בעצמים שכל אחד מהם מכיל נתון והפניה אחת או יותר לעצמים מאותו הטיפוס. על ידי שרשור של עצמים כאלה נוכל לבנות מבנים לייצוג אוספים של נתונים ולבצע עליהם פעולות כגון חיפוש נתון,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

א.1. 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.

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

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

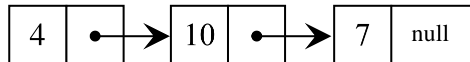
להלן מימוש המחלקה IntNode :

```
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 );
        }
}
```

א.2. 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, ייראה 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.

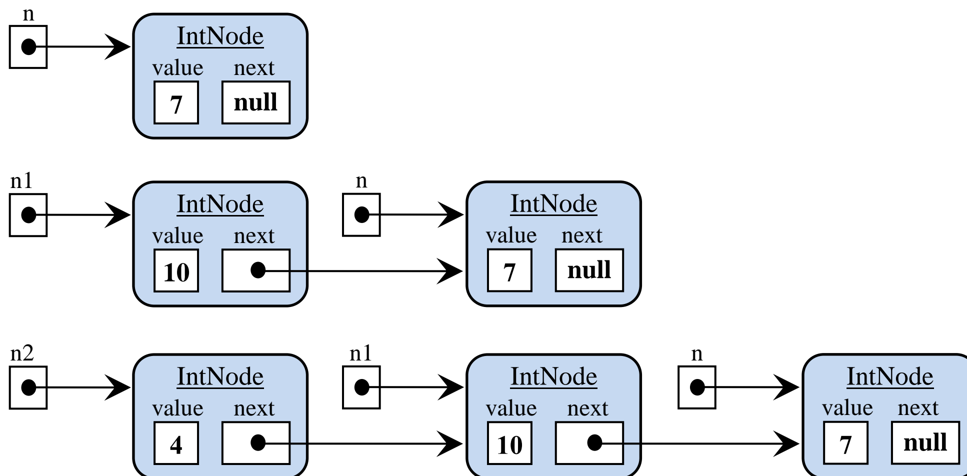
כאמור לעיל, פעולות המחלקה IntNode מאפשרות ליצור רשימות שבהן שמורים מספרים שלמים. בהמשך נראה כי פעולות המחלקה מאפשרות גם ליצור מבנים שאינם רשימות, אך ברובו של הפרק נתמקד ברשימות. בהמשך סעיף זה נראה כי רשימה מקושרת היא מבנה נתונים דינמי, הנותן מענה לחסרונות המערך שמנינו בראשית הפרק: ניתן להוסיף כמה חוליות 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.

א.2.1. 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);
```

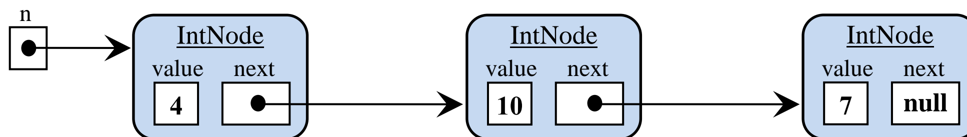
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב את ביצוע קטע התוכנית:



ניתן לקצר את קטע התוכנית הנזכר לעיל ולכותבו כך:

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

תרשים העצמים של התוכנית המקוצרת ייראה כעת כך:



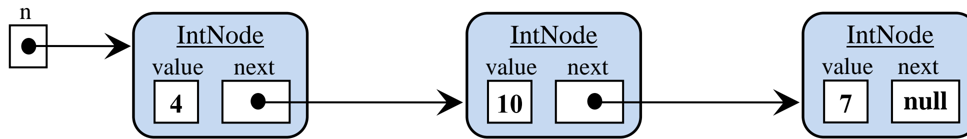
נוסף ליתרון שבכתיבה מקוצרת, דרך כתיבה זו לא נוצרות הפניות נוספות לחוליות שבאמצע הרשימה המקושרת. כפי שכבר הזכרנו בפרק הקודם (6) שדן בהפניות, זהו הישג טוב, שכן ריבוי הפניות עלול לגרום לבעיות שונות. נרחיב את הדיון בנושא זה בהמשך הפרק.

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

א.2.2. 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.

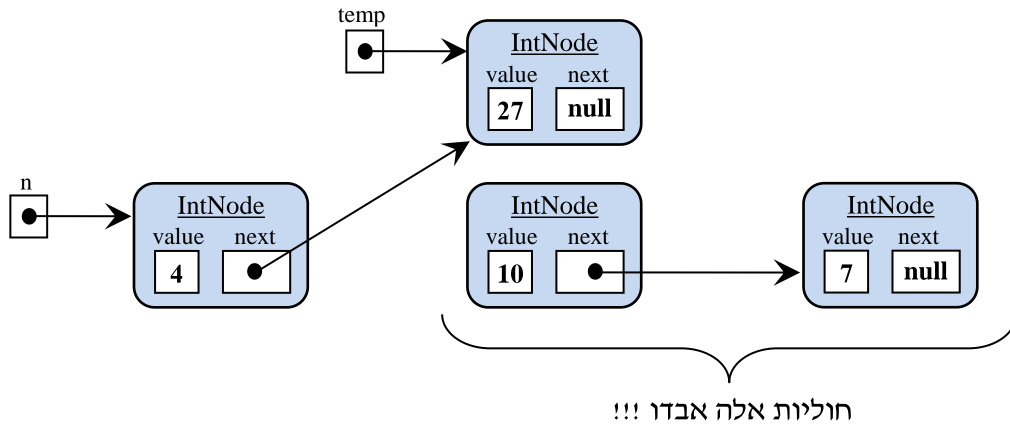
נחזור אל הרשימה המקושרת שאנו עוסקים בה:



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:

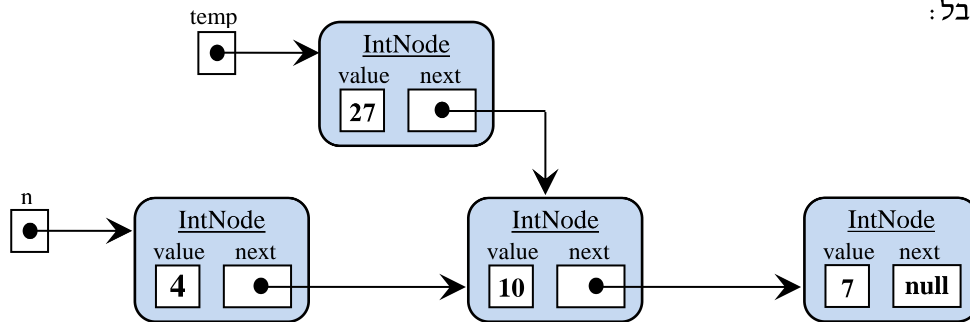
```
IntNode temp = new IntNode(27);
n.setNext(temp);
```



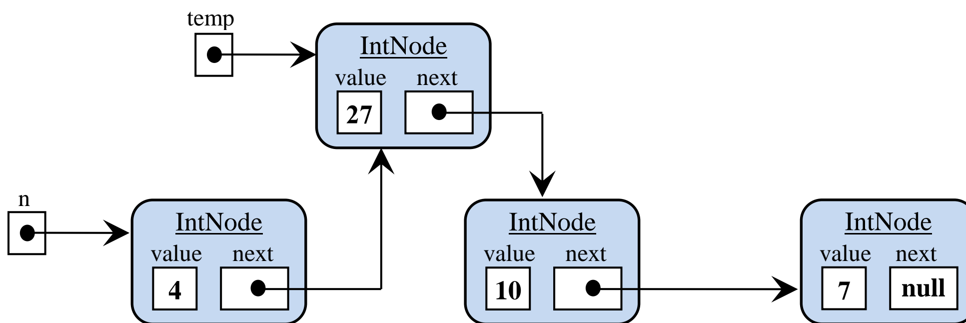
בשלב זה נרצה לבצע את החיבור האחרון ברשימה המקושרת: הפניית החוליה החדשה אל זו העוקבת לה, שהייתה שנייה ברשימה המקורית. אך ערך ההפניה הנחוץ, שהיה שמור בתכונה next של החוליה הראשונה, נמחק כאשר הוכנה אליו ההפניה לחוליה החדשה! כעת אין ביכולתנו לבצע את החיבור המתאים. נראה שהיינו צריכים לבצע את כל מהלך ההוספה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

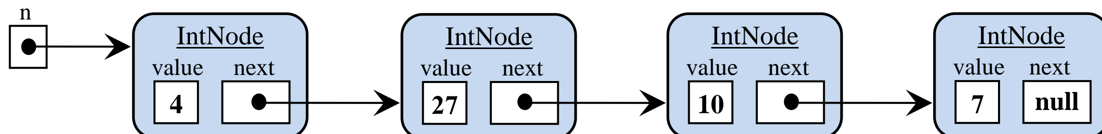
תרשימי העצמים שלפניכם, מתארים שלב אחר שלב, את ביצוע קטע התוכנית. לאחר ההוראה הראשונה נקבל:



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 השניה או השלישית – לפי רצוננו.

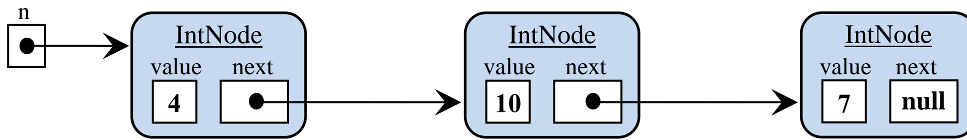
קיום פעולת ההכנסה כפי שתוארה מראה ששתי המגבלות של המערך שתיארנו לעיל אינן קיימות ברשימה מקושרת:

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 :

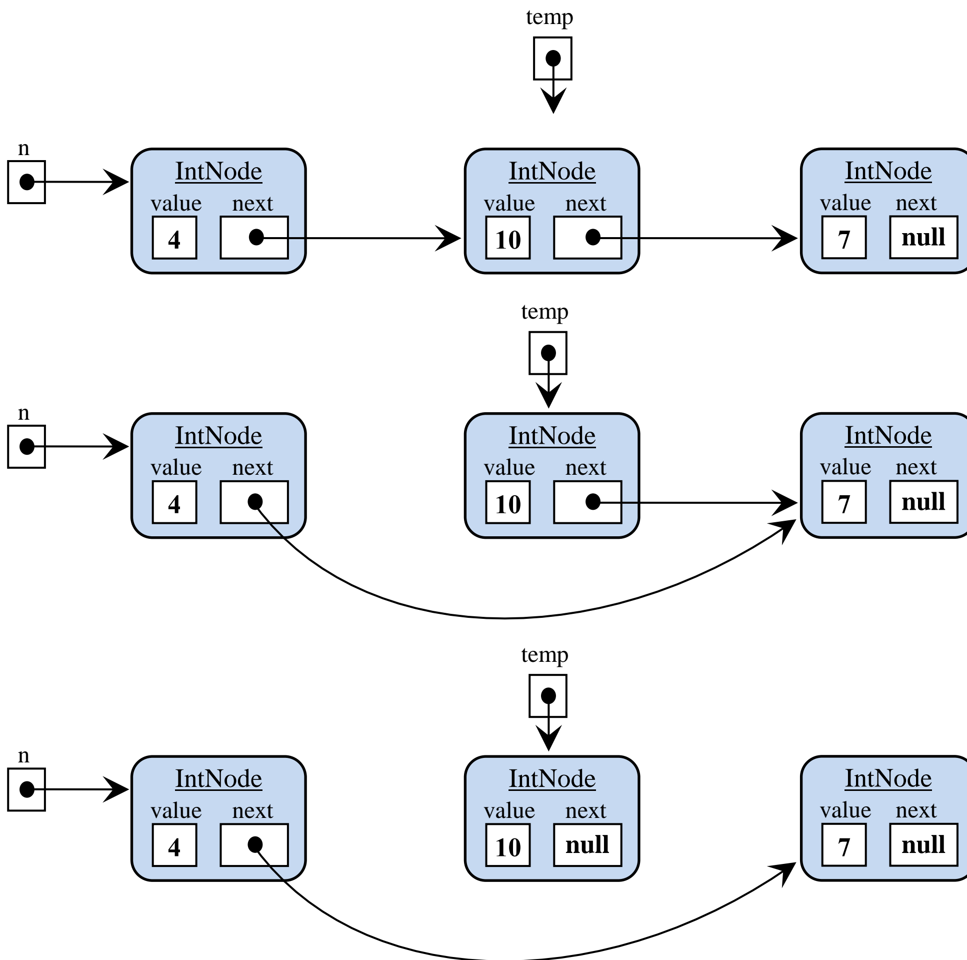
```
IntNode temp = n.getNext();
n.setNext(temp.getNext());
temp.setNext(null);
```

// שימו לב ♥ <u>לא</u> נוצרה כאן חוליה חדשה.

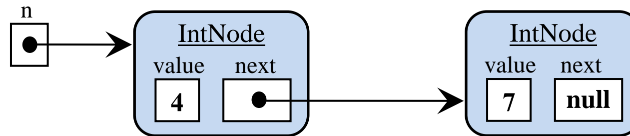
// ההפניה temp מצביעה על חוליה קיימת.

// **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:

שימו לב ♥ <u>לא</u> נוצרה כאן חוליה חדשה. //
אין את הפקודה **new**. //

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

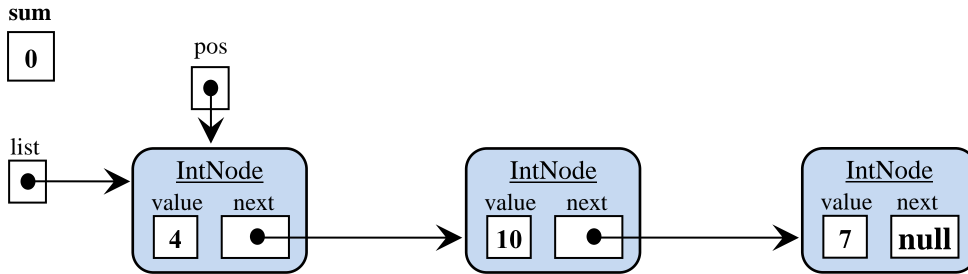
ניישם את התבנית הזו בקטע הקוד שלפניכם, המחשב את סכום המספרים השמורים ברשימה המקושרת ש- list מפנה אליה, החל בחוליה הראשונה וכלה באחרונה:

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

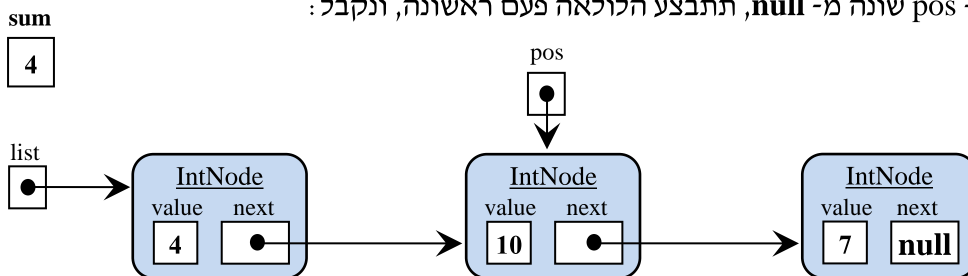
ההפניה החדשה, pos , מסייעת במעבר על הרשימה המקושרת.

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

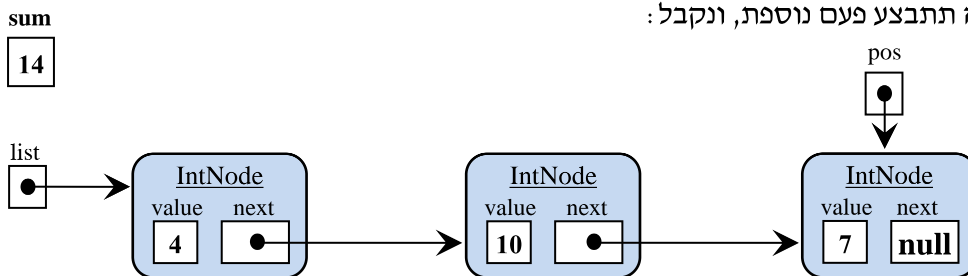
זהו המצב ההתחלתי, אחרי אתחול של pos ושל sum, ולפני שנכנסים ללולאה:



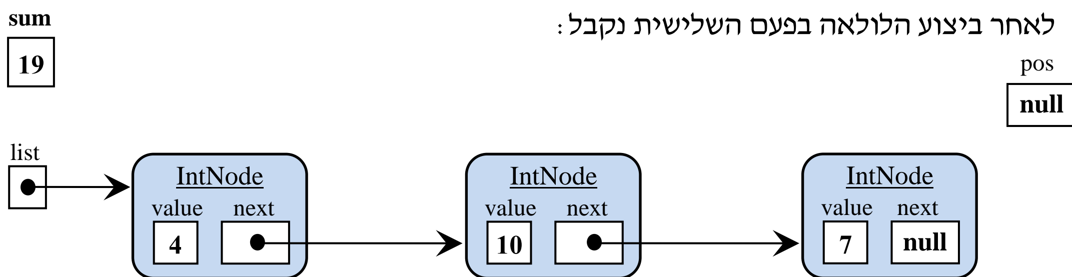
כיוון ש- pos שונה מ- **null**, תתבצע הלולאה פעם ראשונה, ונקבל:



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.

א.3. 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 ♥ **שרשימה 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".

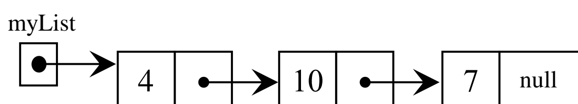
נתבונן בכמה דוגמאות. במהלך הדיון נדון במקרי קצה שונים, ובשגיאות אפשריות.

א.3.1. 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 ;
}
```

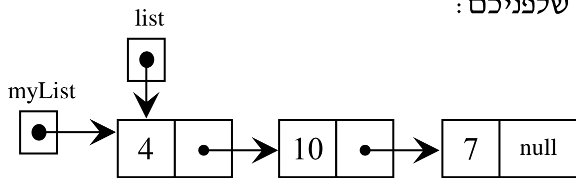
נתונה רשימה מקושרת המוחזקת על ידי משתנה myList :



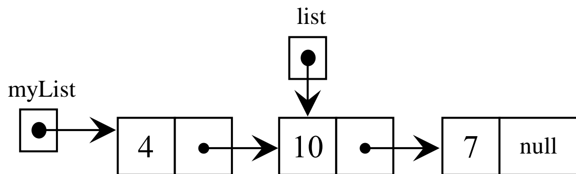
כדי לחשב את סכום המספרים ברשימה המקושרת myList ניתן לזמן את הפעולה getListSum(…) שכתבנו ולשלוח אליה את הרשימה המקושרת myList. הזימון ייראה כך :

```
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 יכיל הפניה לחוליה השנייה:



וכך הלאה. הלולאה תיפסק, כאשר נגיע לסוף הרשימה המקושרת, והמשתנה list יכיל 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.

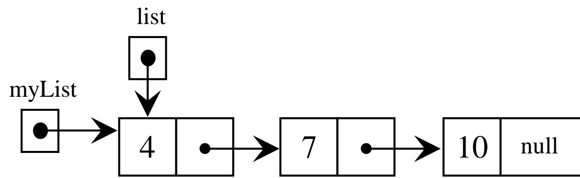
א.3.2. הכנסת ערך לרשימה מקושרת ממוינת

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:

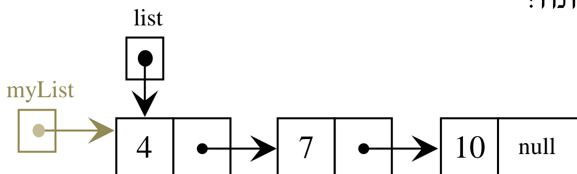
```
public static void insertIntoSortedList(IntNode list , int x)
{
        ...
}
```

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 :

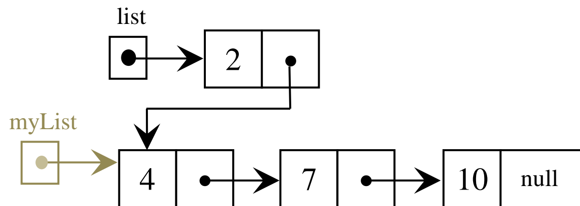


נזמן את הפעולה כך : insertIntoSortedList (myList , 2) במטרה להכניס את הערך 2 לרשימה המקושרת myList הממויינת. הערך 2 הוא הקטן ביותר מבין הערכים 4, 7, 10, ולכן הוא אמור להיכנס כערך הראשון ברשימה המקושרת. בעת זימון הפעולה, הועברה לפרמטר list ההפניה לתחילת הרשימה המקושרת הממויינת. מרגע המעבר לפעולה, ההפניה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



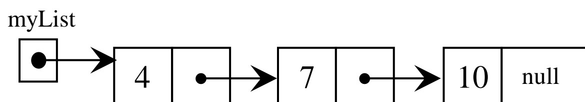
כעת הפעולה insertIntoSortedList (...) עובדת רק עם הפרמטר list כמחזיק הרשימה המקושרת. הפעולה כלל אינה מכירה בתוכה את ההפניה myList (לכן באיורים אלה היא מסומנת בצבע שונה).

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 insertIntoSortedList (...) הפרמטר המקומי list <u>מתבטל</u>! 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.

3.3.א. מציאת ערך מקסימלי ברשימה מקושרת

אם ברצוננו לאתר את הערך הגדול ביותר החֵל 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:

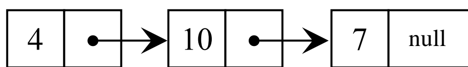
רשימה מקושרת אינה עצם מטיפוס נתונים מוגדר על ידי מחלקה, אלא מבנה המורכב מעצמים מטיפוס חוליה. רשימה מקושרת מוחזקת במשתנה על ידי הפניה לחוליה הראשונה שלה.

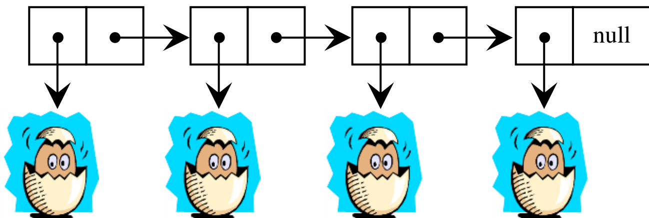
לכן, כשאנו אומרים שפעולה מקבלת רשימה מקושרת, כוונתנו לכך שהפעולה מקבלת הפניה לחוליה הראשונה ברשימה מקושרת.

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.



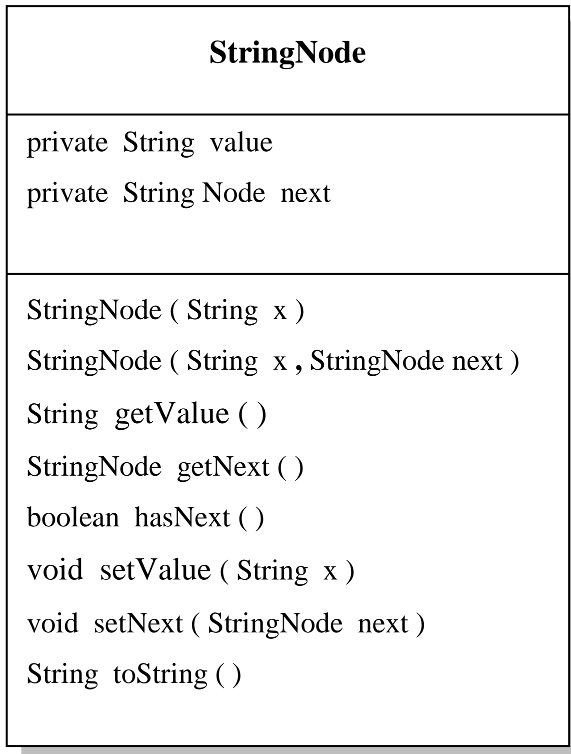


לא נדרשים שינויים משמעותיים כדי להגדיר חוליה שטיפוס הערך שלה שונה ממספר שלם.
למשל, כדי להגדיר מחלקת חוליה שהערך השמור זה הוא מחרוזת, נוכל להעתיק את ההגדרה של המחלקה IntNode ולבצע את השינויים האלה:

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 שעשעינו 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 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 >
{
                              //   (המימוש יתואר בהמשך)  T  מימוש המחלקה באמצעות הטיפוס
}
```

הסימן T אינו שם של טיפוס מסוים, אלא **מחזיק מקום** (**place holder**) לטיפוס שעדין לא נקבע. התוספת <T> אחרי שם המחלקה משמעותה, שזו הגדרה של מחלקה גנרית שבה הטיפוס, שעדיין לא נקבע, מסומן באות T (כפי שנראה בהמשך, נעשָׂה 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>("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:

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 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 תיהיה 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.

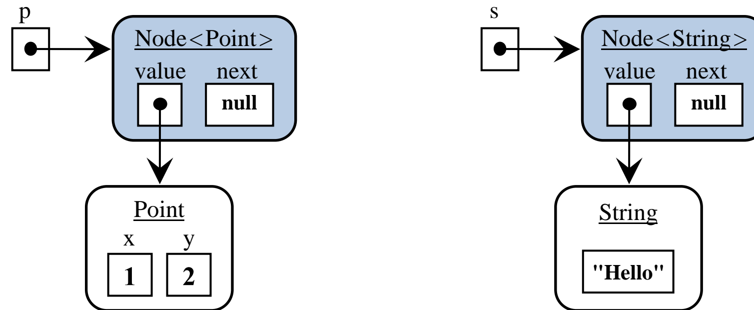
ב.1. שימוש בחוליה הגנרית

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 :

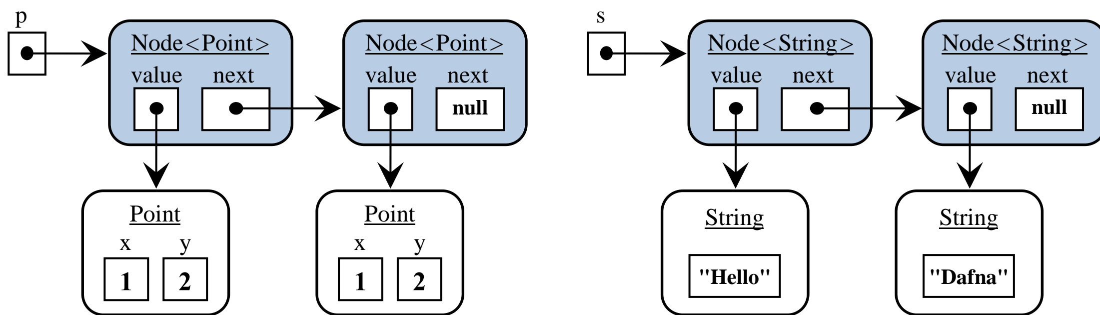
```
public static void main (String [ ]  args)
{
        // שלב 1
        Node<Point> p = new Node<Point>( new Point(1,2));
        Node<String> s = new Node<String>("Hello");

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

ב.2. 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 )
```

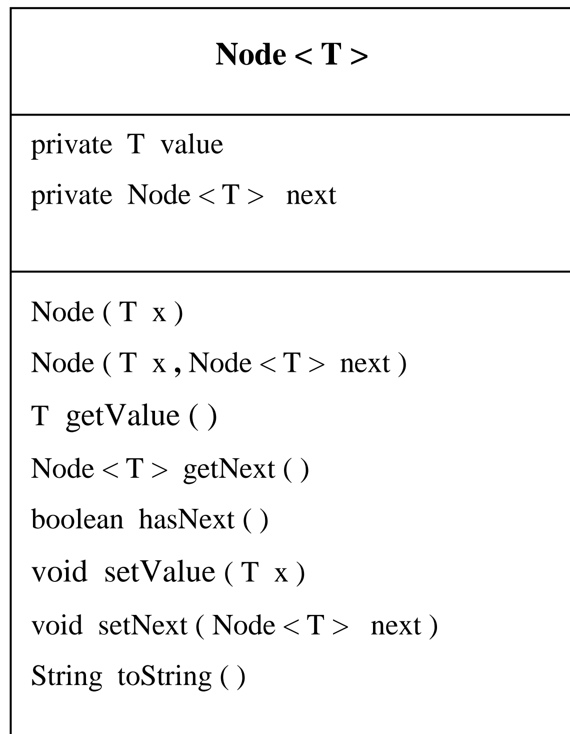
לפרמטרים שונים ניתן לבחור טיפוסים קונקרטיים שונים, כמו בכותרת זו:

```
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:

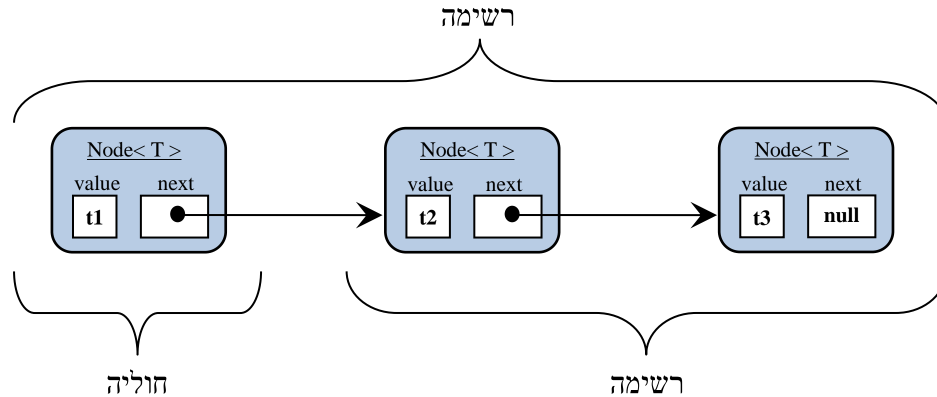
רשימה מקושרת של חוליות היא:

- חוליה יחידה

או

- חוליה שיש בה הפנייה ל**רשימה 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- 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 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 מחרוזת 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 שתלווה 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.

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, 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;
}
```

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 שהמזכיר/ה האחראי/ת 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, 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.

ד.3. 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 **null** == pos, 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 שתתבצע 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.

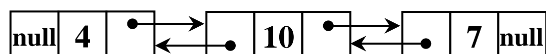
שימו לב שבפרק זה הצגנו ייצוג ומימוש חדשים לרשימה הכיתתית, StudentList. כיוון שאין שינוי בממשק המחלקה, ברור שהרשימה הכיתתית היא טיפוס נתונים מופשט. הממשק של המחלקה אינו חושף בפני המשתמש את דרך הייצוג והמימוש של המחלקה, ולכן המשתמש כלל אינו מודע לשינוי הייצוג ואינו מוטרד ממנו.

ה. מבנים אחרים מבוססי חוליות

הרעיון של שימוש בהפניה אל עצם נוסף מאותו הטיפוס מאפשר ליצור 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. 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 (**Bi**nary) המפנות לכיוונים (**Dir**ections) שונים, נקרא לה BiDirNode ונגדיר אותה כחוליה גנרית.

לפניכם תרשים UML של חוליה דו-כיוונית כזו :

BiDirNode<T>

T value

BiDirNode <T> next

BiDirNode <T> prev

פעולות...

פעולות מסוימות ניתן לממש באופן יעיל יותר על רשימה מקושרת דו-כיוונית מאשר על רשימה 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 :

null

null
null

null
null

null
null

✍ 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** מְמַפְתֵּחַ (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 הירככי, 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

- 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 שמשתמשת 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.

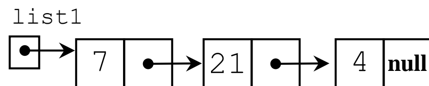
מושגים

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

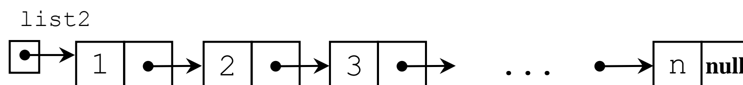
שאלה 1

א. בנו את הרשימה המקושרת הזו :



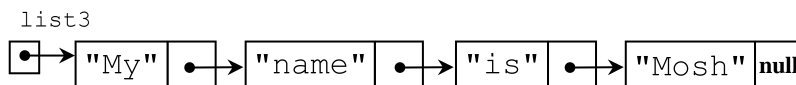
ב. בנו רשימה מקושרת שבה מאוחסנים המספרים השלמים מ-1 ועד n.

n הוא מספר אקראי בין 2 ל-100 :



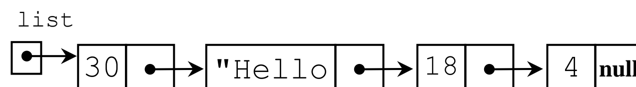
ג. בנו רשימה מקושרת של מחרוזות המייצגת משפט שייקלט. כל מילה תישמר בחוליה נפרדת.

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:

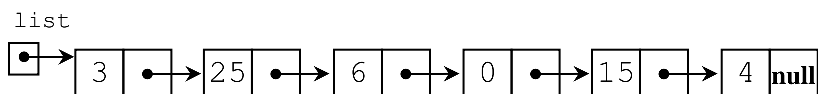
```
public static boolean secret(Node<Integer> list)
{
    if (!list.hasNext())
        return true;

    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.

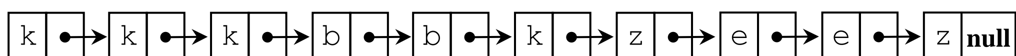
שאלה 8

ממשו את הפעולה הזו :

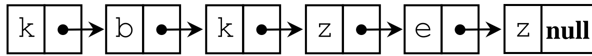
```
public static void compressSequences (Node<Character>  list)
```

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 :

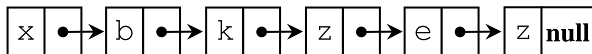


שאלה 9

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:

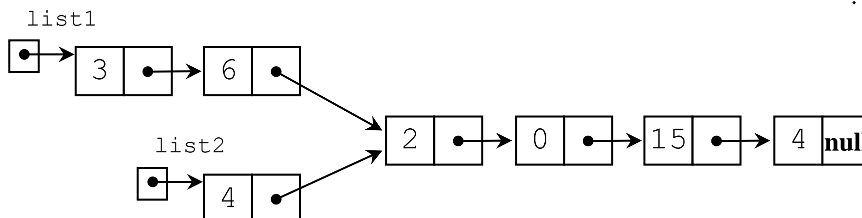
```
public static Node<Integer> merge(Node <Integer> list1,
                                   Node <Integer> list2)
```

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:

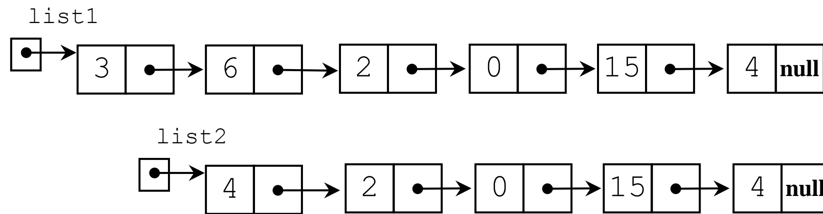


שימו לב כי מספר החוליות הנפרדות בכל אחת מהרשימות אינו זהה בהכרח.

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.

פרק 10

עץ בינרי

מבנה חוליות היררכי

דמיינו לעצמכם משפחה: הורים, ילדים, נכדים וכן הלאה. אנו רוצים לשמור מידע על בני המשפחה ועל קשרי המשפחה ביניהם. כל מבני הנתונים שהכרנו עד עכשיו אינם מתאימים למטרה זו. למשל ננסה לשמור את הנתונים של המשפחה התנ״כית המפורסמת של אברהם אבינו, בתוך מערך או רשימה, כמופיע באיור להלן.

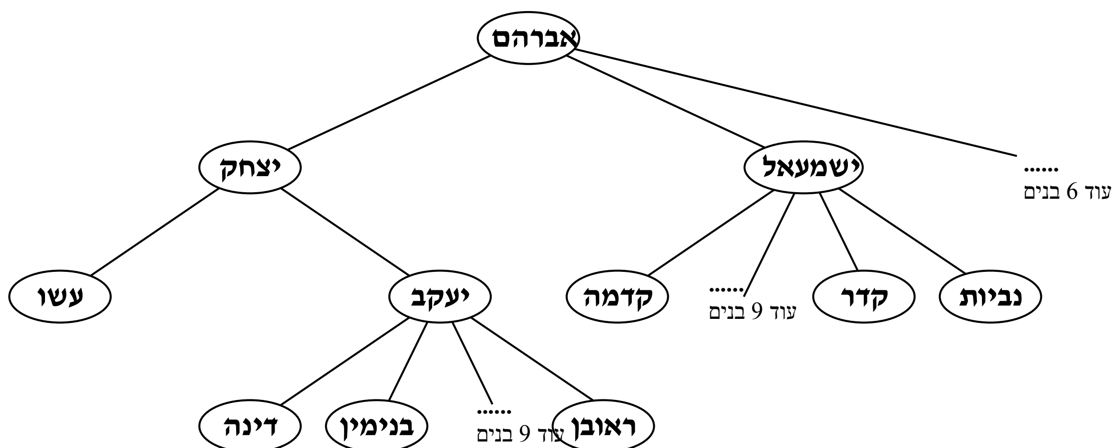
נביות	קדר	קדמה	יעקב	עשו	ישמעאל	יצחק	אברהם

יש לנו ייצוג של בני המשפחה, אך מה לגבי קשרי המשפחה? יצחק וישמעאל הם הבנים של אברהם, עשו ויעקב הם הבנים של יצחק, וקדמה, קדר ונביות הם הבנים של ישמעאל. לא ניתן לראות זאת במערך, ואף לא נוכל לייצג מידע זה ברשימה. משפחה, עם הקשרים בין חבריה, אינה סדרה.

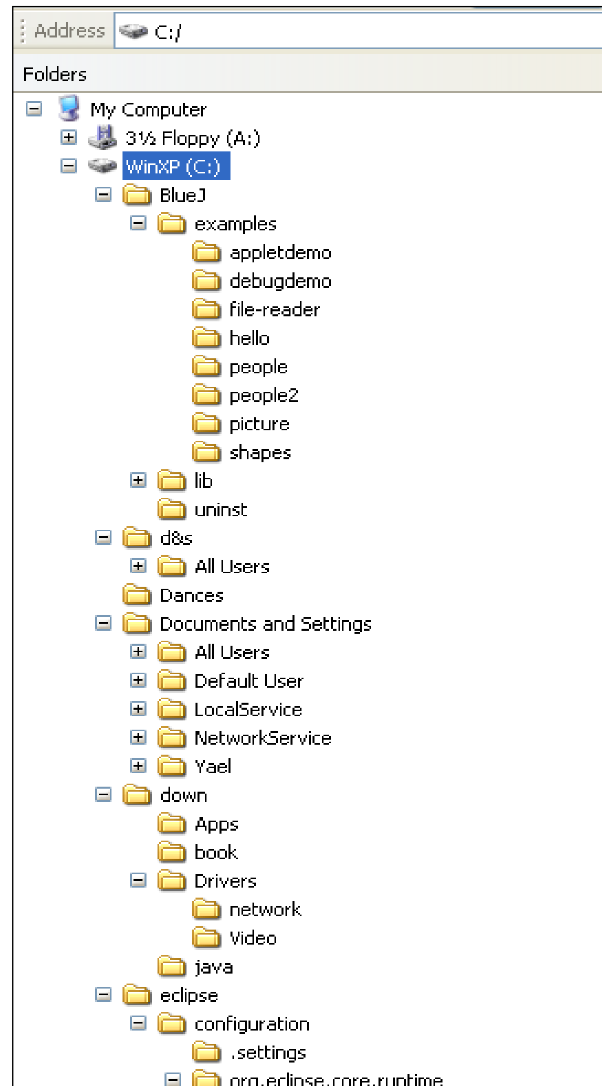
דרך מקובלת לתאר קשרי משפחה של אדם היא באמצעות ציור אילן יוחסין. האילן יכול להיות עץ צאצאים – ראשיתו באדם מסוים והוא מסתעף קדימה אל צאצאיו: ילדיו, נכדיו, ניניו וכו׳, והוא יכול להיות עץ אבות, שמסתעף מן האדם אל עֲבָרוֹ: הוריו, סביו, רב-סביו וכו׳.

המשפחה של אברהם אבינו, כעץ צאצאים, מתוארת באיור שלפניכם. זהו כמובן רק חלק קטן מהעץ המתאים למשפחה ענפה זו, ואפשר להמשיך ולצייר בו ענפים נוספים.

עץ הצאצאים של אברהם:



עץ הוא מבנה נפוץ. גם מערכת קבצים במחשב היא עץ: באיור הבא אנו רואים תיאור של חלק ממערכת קבצים ששורשה הוא MyComputer. בכל ספרייה במערכת הקבצים יש ספריות נוספות או קבצים.



דוגמה נוספת לעץ היא מבנה ארגוני של חברה שבראשו עומד המנהל ומתחתיו כל הכפופים לו.

בפרק זה נציג עצים, נדון בתכונותיהם, נציג משפחה של עצים שהגבלה על המבנה שלהם מאפשרת מימוש יעיל וכן נציג אלגוריתמים שונים על עצים כאלה.

א. עצים

נתחיל בדיון כללי בעצים, ובמונחים המשמשים לדיון בהם. שני המבנים שהצגנו – אילן יוחסין ומערכת קבצים – הם דוגמאות לאוספים המאורגנים כ**עץ** (**Tree**). למשל, אוסף האיברים במקרה של עץ אבות הוא האנשים המופיעים בעץ, והקשרים ביניהם הם יחסי הורות. במקרה של מערכת קבצים, אוסף האיברים מורכב מקבצים ומספריות, והקשר ביניהם הוא קשר של הכלה. מה שמגדיר אותם כעצים הן ההגבלות על הקשרים כפי שיתוארו להלן.

את האיברים בעץ נהוג לכנות בשם **צמתים**, זאת כיוון שבדומה לצומת בדרך, אנחנו יכולים לבחור באחד מכמה כיוונים להמשך דרכנו. הקשרים בין צומת לצמתים הסמוכים לו הם משני סוגים, הנקראים בשמות הלקוחים מעולם המשפחה: סוג אחד מקשר בין צומת ל**הורה** (**parent**) שלו. קשרים מהסוג השני הם בין צומת ל**ילדיו** (**children**). צמתים שהם ילדים לאותו ההורה נקראים, כצפוי, **אחים** (**siblings**).

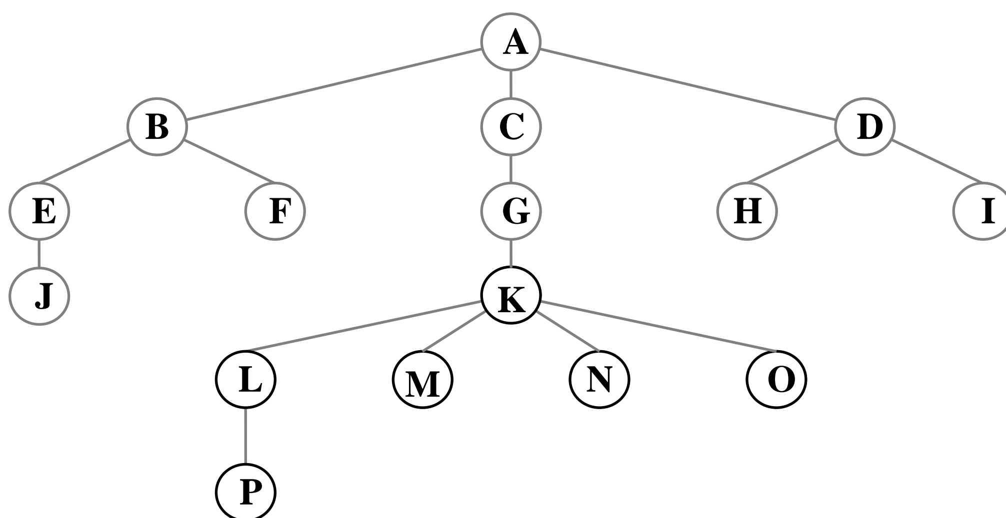
הילדים של צומת אחד בעץ צאצאים הם ילדיו הביולוגים של האדם המיוצג בצומת. בעץ מערכת הקבצים, לעומת זאת, ילדיה של ספרייה המופיעה בצומת הם הקבצים והספריות שהיא מכילה.

תכונה אופיינית לעץ היא שלכל צומת בעץ, פרט לצומת יחיד, הקרוי **שורש** (**root**) העץ, יש הורה אחד בלבד, אולם יכולים להיות לו כמה ילדים.

באיור הבא, שורש העץ הוא הצומת A. ילדיו של A הם B, C ו-D. ילדיו של D הם H ו-I. M ו-L הם אחים. לעומתם, הצמתים G ו-H אינם אחים.

צומת נקרא **צאצא** (**descendant**) של צומת אחר, אם הוא ילד שלו או שהוא צאצא של ילד שלו. (שימו לב, זו הגדרה רקורסיבית). היחס ההפוך לצאצא נקרא **הורה-קדמון** (**ancestor**).

בעץ הבא, O הוא צאצא של C, ולכן C הוא הורה-קדמון של O.



אחרי שהצגנו את המונחים, נציג את המגבלות המגדירות מבנה של עץ:

1. קיים צומת אחד בדיוק ללא הורה; צומת זה קרוי **שורש** העץ.
2. לכל צומת שאינו השורש יש הורה יחיד.
3. כל צומת (לבד מהשורש) הוא צאצא של השורש.

כל צומת בעץ יחד עם צאצאיו הם עץ בפני עצמו. כאשר הצומת אינו השורש של העץ, עץ זה נקרא **תת-עץ** (**subtree**) של העץ המקורי. כיוון שאף הוא עץ, יש לו שורש משלו, כלומר כשנדבר על שורש הכוונה תהיה שורש של עץ או של תת-עץ שלו.

השורש של כל העץ שמופיע באיור הקודם הוא A, השורש של התת-עץ שמכיל את B, E, F ו-J הוא B. בעץ המתאר מערכת קבצים, השורש של תת-עץ המכיל קבצים וספריות הוא הספרייה שמכילה אותם. השורש של העץ המתאר את מערכת הקבצים שבאיור למעלה הוא הספרייה MyComputer.

צומת שאין לו ילדים נקרא **עלה** (**leaf**). עץ בעל צומת אחד בלבד נקרא **עץ עלה**.

העלים בעץ שבאיור הקודם הם J, F, P, M, N, O, H ו-I.

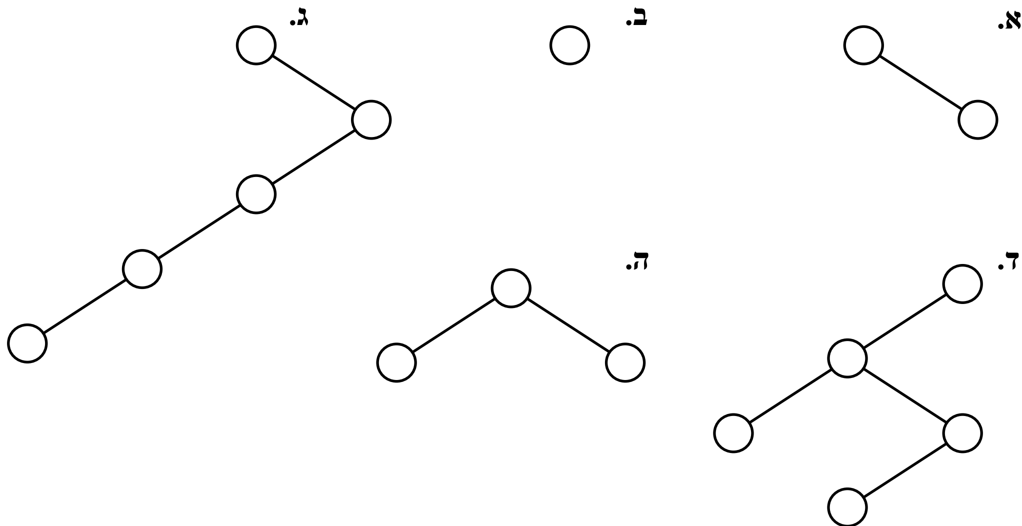
ב. עצים בינריים

בעצים כלליים, שבהם דנו בסעיף הקודם, אין הגבלה על מספר הילדים של צומת. כדי לאפשר ייצוג פשוט בשפת תכנות, נצמצם את מרחב העצים שבו אנו מטפלים ונתמקד בהמשך הפרק בסוג מסוים בלבד – עצים שבהם לכל צומת יש לכל היותר שני ילדים. לעצים אלה נקרא **עצים בינריים**. נעיר כי ליישומים רבים די לעסוק בעצים בינריים בלבד. יתרה מכך, שיטה מקובלת לייצוג עצים כלליים משתמשת בעצים בינריים. לכן, דיוננו בעצים בינריים מספק בסיס טוב לטיפול בעצים כלליים.

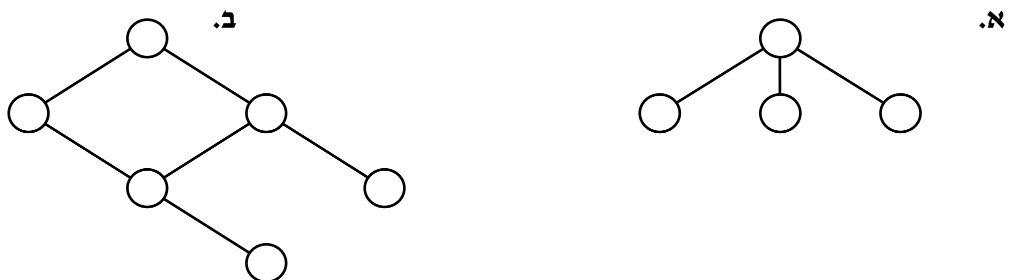
נגדיר **עץ בינרי** (**Binary Tree**) כעץ שיש בו לכל היותר שני ילדים לצומת. מקובל להתייחס אליהם כילד שמאלי וילד ימני. כל אחד משני אלה, אם הוא קיים, הוא שורש של עץ. הילד השמאלי הוא שורשו של עץ הנקרא **תת-עץ שמאלי** (**left subtree**) של הצומת, והילד הימני הוא שורשו של עץ הנקרא **תת-עץ ימני** (**right subtree**). גם כאן כמו בעץ הכללי, לכל צומת אין יותר מהורה אחד (לשורש העץ כולו אין הורה בכלל).

באיור שלפניכם מוצגות חמש דוגמאות לעצים בינריים. באיור שאחריו מוצגות שתי דוגמאות לעצמים שאינם עצים בינריים: באחת יש לשורש יותר משני בנים, ובאחרת יש איבר שלו יותר מהורה אחד, ולכן אין זה עץ כלל.

דוגמאות לעצים בינריים:



דוגמאות לעצמים שאינם עצים בינריים:



מעתה נעסוק רק ב**עצים בינריים**, ובכל מקום שנשתמש במונח **עץ** הכוונה היא ל**עץ בינרי**.

ג. חוליה בינרית

הבסיס למבני נתונים משורשרים סדרתיים הוא השימוש בחוליה שבה יש ערך והפניה אחת לחוליה מאותו הטיפוס. לייצוג עצים בינריים נשתמש בחוליות עם שתי הפניות. חוליה כזו נקראת **חוליה בינרית BinNode** (לחוליה עם הפניה יחידה קוראים בהתאם **חוליה אונרית**). החוליה הבינרית תשמש לייצוג צומת עץ. לחוליה כזו יש שלוש תכונות: תכונה המכילה את הערך השמור בצומת ושתי תכונות שהן הפניות לעצמים נוספים מטיפוס BinNode. כאשר ערכה של הפניה כזו הוא **null** פירוש הדבר שהתת-עץ המתאים אינו קיים. כאשר ערך ההפניה אינו **null**, העצם שאליו היא מפנה הוא ילד של הצומת, שהוא שורש של תת-עץ.

כפי שעשינו בעבר, גם כאן נגדיר את המחלקה עבור כל טיפוס ערך, כלומר באופן גנרי: BinNode<T>.

ג.1. ממשק המחלקה BinNode <T>

כמו לכל מחלקה, גם עבור המחלקה BinNode<T> יש להגדיר פעולות בונות מתאימות כך שנוכל לייצר עצמים מהתבנית שלה. הפעולה הבונה הכללית תקבל ערך ושתי הפניות. לשם הנוחיות נגדיר גם פעולה הבונה חוליה שבה ערך בלבד ושערך שתי ההפניות שלה הוא **null**. כיוון שהערך השמור בחוליה יכול להיות מטיפוס כלשהו, תוגדר החוליה הבינרית כגנרית.

לכל אחת מהתכונות נוסיף פעולות : get() ו-set(…) מתאימות. פעולות getValue() ו-setValue(…) מאפשרות לאחזר את הערך שבצומת ולשנותו. פעולות getLeft() ו-setLeft(…) מאפשרות לאחזר את הילד השמאלי ולשנותו. שתי פעולות דומות קיימות גם עבור הילד הימני. כמו עבור כל עצם, נגדיר פעולת toString() שתחזיר את תיאור החוליה הבינרית.

ממשק המחלקה BinNode <T>

המחלקה מגדירה חוליה בינרית שבה ערך מטיפוס T ושתי הפניות לחוליות בינריות.

הפעולה בונה חוליה בינרית. ערך החוליה הוא x וערך שתי ההפניות שלה הוא **null**	BinNode (T x)
הפעולה בונה חוליה בינרית שערכה יהיה x. left ו-right הן (הפניות אל) הילד השמאלי והימני שלה. ערכי ההפניות יכולים להיות **null**	BinNode (BinNode <T> left **,** T x , BinNode <T> right)
הפעולה מחזירה את הערך של החוליה	T getValue( )
הפעולה משנה את הערך השמור בחוליה ל-x	**void** setValue (T x)
הפעולה מחזירה את הילד השמאלי של החוליה. אם אין ילד שמאלי הפעולה מחזירה **null**	BinNode <T> getLeft()
הפעולה מחזירה את הילד הימני של החוליה. אם אין ילד ימני הפעולה מחזירה **null**	BinNode <T> getRight()
הפעולה מחליפה את הילד השמאלי בחוליה left	**void** setLeft (BinNode <T> left)
הפעולה מחליפה את הילד הימני בחוליה right	**void** setRight (BinNode <T> right)
הפעולה מחזירה מחרוזת המתארת את הערך השמור בחוליה	String toString( )

ג.2. המחלקה BinNode <T>

להלן מימוש המחלקה:

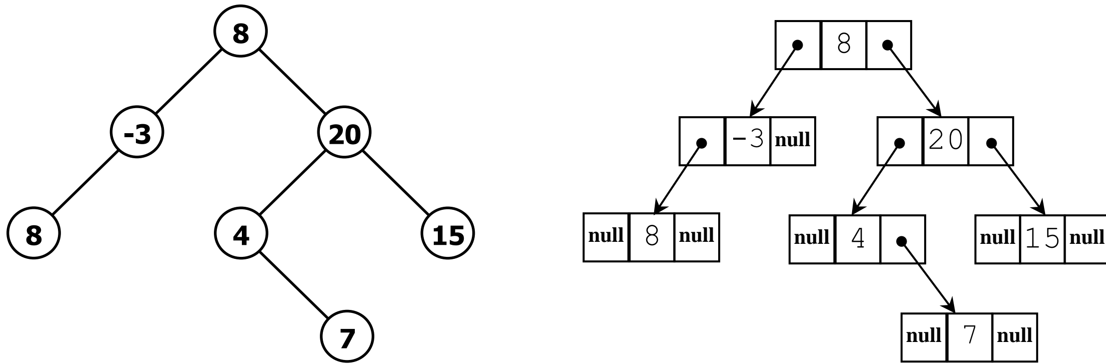
```
public class BinNode<T>
 {
        private BinNode<T> left;
        private T value;
        private BinNode<T> right;

        public BinNode(T x)
        {
                this.left = null;
                this.value = x;
                this.right = null;
        }

        public BinNode(BinNode<T> left, T x, BinNode<T> right)
        {
                this.left = left;
                this.value = x;
                this.right = right;
        }

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

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

```
public BinNode<T> getLeft()
{
    return this.left;
}

public BinNode<T> getRight()
{
    return this.right;
}

public void setLeft(BinNode<T> left)
{
    this.left = left;
}

public void setRight(BinNode<T> right)
{
    this.right = right;
}

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

קל לראות כי יעילותן של כל פעולות הממשק היא קבועה O(1).

ד. עץ חוליות בינרי

המחלקה BinNode מאפשרת ליצור חוליות בינריות, לחבר אותן זו לזו באמצעות הפעולות setLeft(…) ו-setRight(…), וכך לבנות מבני חוליות היררכיים שייצגו עצים בינריים. לדוגמה, מבנה החוליות הבינריות מימין מייצג את העץ הבינרי המופיע משמאל:



לשם הפַּשטות, במקום הביטוי המדויק אך המסובך: "מבנה חוליות המייצג עץ בינרי", נשתמש מכאן והלאה בביטוי הפשוט יותר: "עץ חוליות בינרי", ואף בביטוי המקוצר: "עץ בינרי". כמו כן, נשתמש לעתים קרובות במונח "צומת" במקום במונח "חוליה".

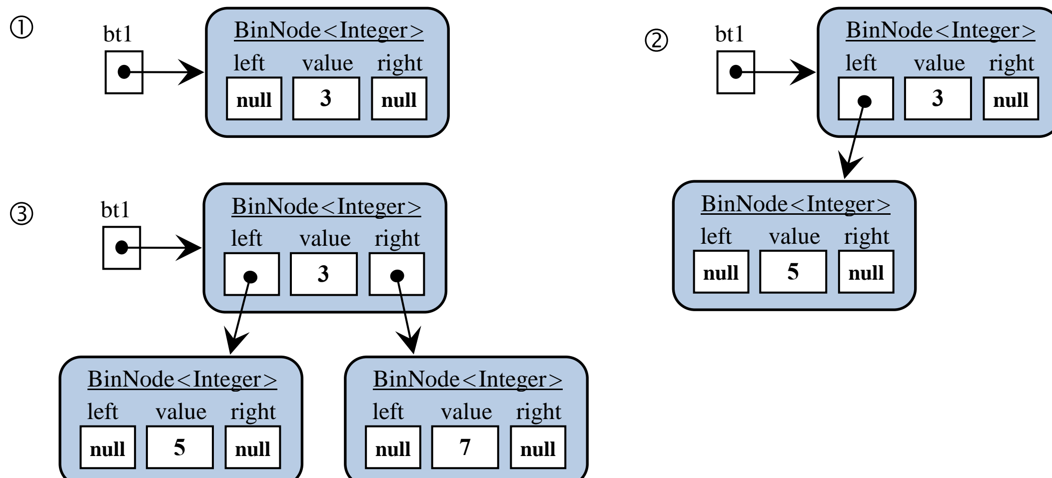
בסעיפים הבאים נדגים כיצד ניתן לבנות עץ חוליות בינרי ונדון בתכונותיו.

ד.1. בניית עץ חוליות בינרי

נתבונן בקטע התוכנית שלפנינו, הבונה עץ חוליות בינרי. עץ החוליות יכיל כמה צמתים ובהם מספרים שלמים. הבנייה תיעשה מהשורש לכיוון העלים. תחילה נבנה את השורש ואחר כך נוסיף צמתים על ידי הפעולות setLeft(…) ו- setRight(…):

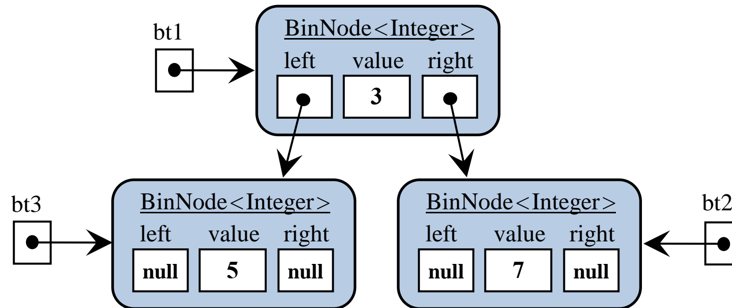
```
①    BinNode<Integer> bt1 = new BinNode<Integer>(3);
②    bt1.setLeft(new BinNode<Integer>(5));
③    bt1.setRight(new BinNode<Integer>(7));
```

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



אפשרות אחרת ליצירת אותו העץ היא מהעלים לכיוון השורש. יש ליצור שני תת-עצים ולצרפם בעזרת הפעולה הבונה השנייה לעץ אחד, שבשורשו ערך נתון:

```
BinNode<Integer> bt3 = new BinNode<Integer>(5);
BinNode<Integer> bt2 = new BinNode<Integer>(7);
BinNode<Integer> bt1 = new BinNode<Integer>(bt3, 3, bt2);
```



ניתן לעשות זאת אף בפקודה אחת:

```
BinNode<Integer> bt1 = new BinNode<Integer>
       (new BinNode<Integer>(5), 3, new BinNode<Integer>(7));
```

✍ עץ עלה הוא העץ הקטן ביותר שיכול להתקיים. פעמים רבות ניעזר בבדיקה האם עץ מסוים הוא עץ עלה. כתבו פעולה הבודקת האם חוליה בינרית נתונה היא עץ עלה:

```
public static boolean isLeaf (BinNode<Integer> node)
```

ד.2. תנועה על עץ חוליות בינרי ושינויו

לאחר שבנינו עץ חוליות בינרי, נוכל לעבור על העץ משורשו לכיוון מטה. זאת נעשה בעזרת הפעולות getLeft()‎ ו- getRight()‎ של חוליה בינרית, המובילות מצומת אל הילד השמאלי או הימני שלו. אם ערך ההפניה שפעולה כזו מחזירה הוא **null** – פירוש הדבר שלצומת אין ילד שמאלי (או ימני בהתאמה) והתנועה על העץ תיפסק. בהגיענו לצומת, אנו יכולים לשלוף את המידע השמור בו או לשנותו.

לדוגמה, בהינתן העץ שבנינו קודם, ביצוע שורת הקוד:

```
int a = bt1.getLeft().getValue();
```

יחזיר את הערך 5. כאן, ערך הביטוי bt1.getLeft()‎ הוא הפניה לילד שהוא שורש התת-עץ השמאלי, והפעולה getValue()‎ מחזירה את הערך הנמצא בו.

באופן דומה נוכל לשנות את הערך שבילד השמאלי של העץ מ-5 ל-7:

```
bt1.getLeft().setValue(7);
```

נעיר כי אנו יכולים לבצע גם פעולות "נועזות" יותר, למשל להחליף את שני התת-עצים של העץ הנתון, זה בזה:

```
BinNode<Integer> temp = bt1.getLeft();
bt1.setLeft(bt1.getRight());
bt1.setRight(temp);
```

ד.3. הכנסת ערכים לעץ חוליות בינרי

כאשר לצומת אין ילד שמאלי אפשר לשנות את ערך ההפניה left שבו, על ידי שימוש בפעולה setLeft(…). הערך המקורי, **null**, יהפוך להפניה לחוליה שהיא שורש של עץ, שיהפוך על ידי כך לתת-עץ שמאלי של העץ הנתון. באופן דומה אפשר להוסיף תת-עץ ימני. שינוי ההפניות מהווה למעשה הכנסה של צמתים (המכילים ערכים) לעץ המקורי. גם אם הילד השמאלי או הימני קיימים ואנו מבצעים את התהליך המתואר, אנו מבצעים למעשה הכנסה של צמתים לעץ. עם זאת, בו בזמן אנו מוחקים צמתים שהיו בעץ קודם לכן, ומכאן שאי אפשר להסתכל על פעולה זו כפעולת הכנסה, אלא כפעולת החלפה.

הדגמנו והסברנו רק הוספה בקצוות של העץ, כאשר ערך ההפניה **null** הוחלף בהפניה לעץ. האם אפשר להוסיף חוליה במרכז העץ, כשם שהוספנו חוליות באמצע רשימה מקושרת? התשובה היא אמנם חיובית, אך הוספה כזו היא מסובכת. נניח כי לחוליה BinNode יש תת-עצים שמאלי וימני, ואנו רוצים להוסיף לעץ הבינרי חוליה חדשה (ובה ערך כלשהו). שימו לב שהמושג "הוסף אחרי" אינו מוגדר כאן, ואם כך עלינו להחליט האם אנו רוצים להוסיף את החוליה החדשה בצד שמאל של החוליה BinNode או בצד ימין שלה. נניח כי החלטנו על הוספה בשמאל. עכשיו עולה השאלה מה נעשה עם התת-עץ השמאלי הנוכחי של BinNode? שוב עלינו לבחור האם הוא יהפוך להיות תת-עץ שמאלי של החוליה החדשה או תת-עץ ימני שלה. לאחר שבחרנו גם כאן באחת משתי האפשרויות, נוכל לבנות קוד שיבצע את ההכנסה.

לפעולת הכנסה באמצע עץ חוליות בינרי אין שימוש רב. בגלל זה, ובגלל סיבוכה, לא נעסוק בה עוד בפרק זה.

ד.4. הוצאת ערכים מעץ חוליות בינרי

הוצאת עלה מעץ היא פעולה קלה לביצוע, בתנאי שיש לנו הפניה לחוליית ההורה שלו. מצב זה דומה למצב ברשימה מקושרת, שממנה אפשר להוציא חוליה אם יש לנו הפניה לחוליה הקודמת לה. באופן דומה, ניתן להוציא תת-עץ שלם המחובר לחוליה נתונה. אבל, הוצאת חוליה בודדת מאמצע העץ, היא פעולה מסובכת. נניח כי לחוליה BinNode יש תת-עץ שמאלי ששורשו הוא החוליה binNode1, ואנו רוצים להוציא את binNode1 מהעץ. ל-binNode1 יש במקרה הכללי שני תת-עצים, שאת החוליות שלהם (פרט ל-binNode1 עצמה) אנו רוצים להשאיר בעץ. אנו יכולים לחבר אחד מהם כתת-עץ שמאלי של binNode, אך איזה מהם? ומה נעשה עם השני? גם כאן, לאחר שנקבל החלטות מתאימות, נוכל לכתוב קוד לביצוע הפעולה הדרושה. החלטות כאלה יש לקבל במסגרת יישום המשתמש בעץ. ברובו של פרק זה כלל לא נעסוק בהוצאות של ערכים מתוך עץ.

ד.5. שמירה על מבנה העץ

תנועה על עץ, המתבצעת מצומת אל ילד שלו, שליפת מידע מצומת או החלפתו באחר – כל הפעולות הללו אינן משנות את מבנה העץ. הכנסת חוליה או תת-עץ חדש, וכן הוצאת חוליה או תת-עץ, המתרחשות עקב שימוש בפעולות setLeft(…) ו- setRight(…) על חוליה שמייצגת צומת בעץ, משנות את המבנה.

כאמור, עץ חוליות בינרי הוא מבנה המורכב מחוליות בינריות ומייצג עץ בינרי. מבנה כזה חייב לקיים את ההנחות והתנאים שתוארו בהגדרת המושג "עץ בינרי" בראשית סעיף ב'. קל לקלקל את המבנה, כך שיהפוך למבנה שאינו עץ חוליות בינרי, תוך שימוש בפעולות שינוי המבנה. לדוגמה, נסתכל שוב בעץ שבנינו קודם, שהמשתנה bt1 מכיל הפניה אליו. הפקודה שלפניכם הופכת אותו למבנה שאינו עץ חוליות בינרי:

```
bt1.setLeft(bt1);
```

☝ שימו לב: שתי התכונות של חוליה בינרית קרויות left ו- right, ובתחילת סעיף ג' אמרנו ש- left,

אם אינה **null**, מפנה לתת-עץ השמאלי, וכך לגבי right. אולם בפועל אין זו אלא הבעת כוונות בלבד. השמירה על המבנה התקין של העץ תלויה ברצון הטוב (ובידע) של המשתמש; כאשר אחד מאלה או שניהם חסרים, עלול להתקבל מבנה חוליות שאינו מייצג עץ בינרי.

בעיות דומות התעוררו גם בייצוג סדרות על ידי רשימות מקושרות. כל זמן שרשימה מקושרת הייתה חשופה, לא יכולנו להבטיח שמירה על המבניות הלינארית שלה. הפתרון עבור רשימות, כפי שנוכחנו לדעת בפרקים הקודמים, היה הגדרת מחלקה עוטפת, שיש לה הפניה פנימית לרשימה מקושרת ופעולות המטפלות ברשימה מקושרת. דוגמאות למחלקות כאלו הן מחסנית ותור. לאחר שנכתב הקוד למחלקה כזו ונבדק היטב, ולאחר שהבטחנו שקוד זה שומר על המבנה הרצוי, אנו יכולים להיות בטוחים שלא תקרה תקלה שתפגע במבנה הרשימה המקושרת, שכן למשתמש במחלקה אין גישה ישירה לרשימה מקושרת. כאשר השתמשנו ברשימה שבה נשארה גישה ישירה לחוליות, לא יכולנו להבטיח שלא יקרו תקלות ושיבושים במבנה הרשימה.

בעצים בינריים המצב דומה לרשימה. כל זמן שהעץ מיוצג על ידי מבנה חוליות בינריות, וקוד המשתמש בעץ יכול לפעול ישירות על מבנה זה, לא נוכל להבטיח שמבנהו התקין לא ייפגע. כאשר נגדיר מחלקות המשתמשות בעצי חוליות בינריים כתכונות פנימיות, והמחלקות יגדירו טיפוסי נתונים מופשטים, נוכל להיות בטוחים שמבנה העץ יישמר בקפידה.

ה. עץ חוליות בינרי – מבנה רקורסיבי

נגדיר עץ חוליות בינרי בצורה רקורסיבית, בדומה להגדרה שנתנו לרשימה המקושרת הלינארית. כשם שרשימה מקושרת מכילה לפחות חוליה אחת, כך גם עץ חוליות בינרי מכיל לפחות חוליה אחת.

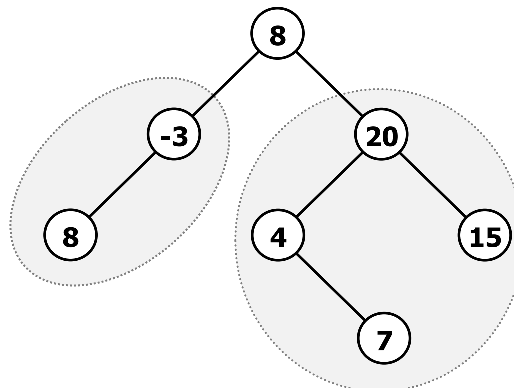
כלומר **עץ חוליות בינרי** הוא:

- חוליה בינרית יחידה

או

- חוליה בינרית שבה יש הפניה אחת ל**עץ חוליות בינרי** או שתי הפניות ל**עצי חוליות בינריים** הזרים זה לזה (שאין להם חוליות משותפות)

ההגדרה הרקורסיבית מתאפשרת משום שבחוליה הבינרית קיימות שתי תכונות left ו-right המכילות הפניות לחוליות בינריות, שהן שורשים של תת-עצים בינריים. כלומר כל חוליה במבנה היא בעצמה שורש של עץ חוליות בינרי. באיור שלפניכם מוצג עץ בינרי ששורשו 8, ולו שני תת-עצים שגם הם עצים בינריים: תת-עץ ימני ששורשו 20 ותת-עץ שמאלי ששורשו 3-.



ההגדרה הרקורסיבית מאפשרת לכתוב פעולות רקורסיביות על עץ חוליות בינרי. נבחן שתי פעולות המנצלות את המבנה הרקורסיבי של העץ. הפעולות יבצעו את משימתן על ידי ביצוע פעילות בשורש, בתוספת הַפְּעֻלות רקורסיביות שלהן בַּתת-עצים.

ה.1. פעולות על עצי חוליות בינריים

פעולות על עצי חוליות יקבלו את העץ כפרמטר. למעשה הפרמטר המועבר הוא הפניה לחוליה בינרית המהווה שורש של עץ חוליות. טיפוס החוליה הוא טיפוס קונקרטי.

דוגמה 1: ספירת מספר הצמתים בעץ

נכתוב פעולה המחזירה את מספר הצמתים בעץ.

נכתוב את הפעולה עבור עץ שצמתיו מכילים מספרים שלמים. כדי לספור את הצמתים בעץ, נספור הן את הצמתים בתת-עץ הימני והן את הצמתים בתת-עץ השמאלי, ונחבר את הערכים המתקבלים. לסכום שהתקבל נוסיף 1 עבור השורש של העץ, שגם הוא צומת. ספירת הצמתים בכל תת-עץ תיעשה באותה השיטה עצמה (ומשום כך זו פעולה רקורסיבית).

```
public static int numNodes(BinNode<Integer> bt)
{
	if (bt == null)
		return 0;
	return numNodes(bt.getLeft()) + numNodes(bt.getRight()) + 1;
}
```

נעיר שתי הערות:

א. הבדיקה (**if** (bt == **null** משמשת למקרי הסיום של המעבר על המבנה ולא כתנאי פתיחה לבדיקת הפרמטר שנשלח לפעולה. הזימונים הרקורסיביים מסתמכים על הגדרת המבנה, כיוון שהפעולה ()getLeft (או ()getRight בהתאמה) יכולה להחזיר **null** כחלק מהגדרתה. ערך החזרה כזה פירושו שאין חוליה נוספת בכיוון זה במבנה.

ב. הפעולה בדוגמה זו התמקדה במבנה העץ. לכאורה ניתן היה להגדיר את הפעולה כגנרית משום שהיא אינה מבצעת דבר על הערכים השמורים בעץ. בכל זאת לפי הכללים שקבענו ביחידה זו, עצים המועברים כפרמטרים לפעולות חיצוניות חייבים להיות עצים קונקרטיים.

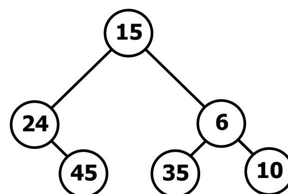
דוגמה 2: הדפסת ערכי העץ

נכתוב פעולה המדפיסה את הערכים השמורים בצומתי העץ.

```
public static void printNodes(BinNode<Integer> bt)
{
	if (bt != null)
	{
	    System.out.print(bt.getValue() + " ");
		printNodes(bt.getLeft());
		printNodes(bt.getRight());
	}
}
```

כמו בדוגמה הקודמת, גם פעולה זו מסתמכת על ההגדרה הרקורסיבית של עץ חוליות, אלא שהפעם ההתייחסות היא לערכים השמורים בצמתים.

✍ בצעו מעקב רקורסיבי על העץ שלפניכם. הפעילו עליו את הפעולות (...)numNodes ו-(...)printNodes וכתבו מה מתקבל.



בסעיף הבא נגדיר באופן מסודר את האופנים השונים למעבר על עץ, המאפשרים ביצוע פעולות רקורסיביות על עצים. כך גם נבין כיצד התקבל סדר ההדפסה של הערכים השמורים בעץ על ידי הפעולה printNodes(…).

ו. מעברים על עץ בינרי

שימושים רבים בעץ בינרי מצריכים מעבר על כל צומתי העץ ללא חזרות. למשל, הדפסת ערכי הצמתים בעץ, ביצוע פעולות חיפוש או ביצוע פעולה כלשהי על הערכים שבצמתים כדוגמת הפעולות שכתבנו בסעיף הקודם.

מאחר שהעץ הוא מבנה היררכי ניתן לעבור על הצמתים בסדרים שונים. את הסדר נבחר בהתאם לצורכי היישום. סדרי מעבר שונים יתאימו לצרכים שונים. ביישומים מסוימים אין חשיבות לסדר, ובלבד שלא נבקר יותר מפעם אחת בכל צומת. לפעמים נפסיק את המעבר לפני סופו אם נמצא את מה שחיפשנו.

לדוגמה, אם נרצה לבדוק האם פלוני נמצא בעץ משפחה, עלינו לעבור על העץ ולחפש את שמו בצמתיו. ברור שנדרש לנו חיפוש יעיל שבמהלכו נבקר בכל צומת פעם אחת לכל היותר. המעבר ייפסק אם נמצא את נתוני אותו פלוני בצומת מסוים.

אם נרצה להדפיס את שמות כל האנשים בעץ אבות לפי סדר הדורות: ראשית הילד, אחריו הוריו, אחריהם סביו וסבותיו וכן הלאה, נזדקק למעבר על העץ לרוחבו, לפי רמות, תוך מעבר בכל הצמתים.

באילו מהדוגמאות שהבאנו לעיל יש חשיבות לסדר הביקור בצמתים, ובאילו אין?

שתי הדוגמאות לעיל דורשות לכל היותר ביקור יחיד בכל אחד מצומתי העץ, שבמהלכו מבוצעת פעולה כלשהי. סוג זה של מעבר מכונה: **סריקה של העץ**.

קיימות כמה דרכים לסריקה של עץ, ולהלן נציג אחדות מהן. ניתן להתייחס לגישות אלה כתבניות שאותן ניישם בהתאם לצרכינו.

סריקה מתחילה משורש עץ. כדי לעבור מהשורש אל אחד מילדיו, אם הם קיימים, יש להשתמש בפעולות getLeft() או getRight(). באמצעות שתי פעולות אלה נוכל להגיע לכל אחד מצומתי העץ. במהלך סריקת עץ נרצה בדרך כלל להתייחס לערכים השמורים בצומת. כיוון שכל צומת הוא שורש של תת-עץ, די בפעולות אחזור ועדכון לשורש. לשם כך יש לנו בממשק את הפעולה getValue() המחזירה את תוכן השורש הנוכחי, ואת הפעולה setValue(…) המשנה את התוכן של השורש הנוכחי.

ו.1. סריקות עומק של עץ בינרי

קיימים 3 סוגי סריקות עומק של עץ: **סריקה בסדר תחילי (preorder traversal)**, **סריקה בסדר תוֹכי (inorder traversal)** ו**סריקה בסדר סופי (postorder traversal)**. בשלושת סוגי הסריקות מתבצעות הפעולות הבאות, אך בכל סריקה הן מתבצעות בסדר שונה:

- ביקור בשורש העץ
- סריקה רקורסיבית של התת-עץ השמאלי
- סריקה רקורסיבית של התת-עץ הימני

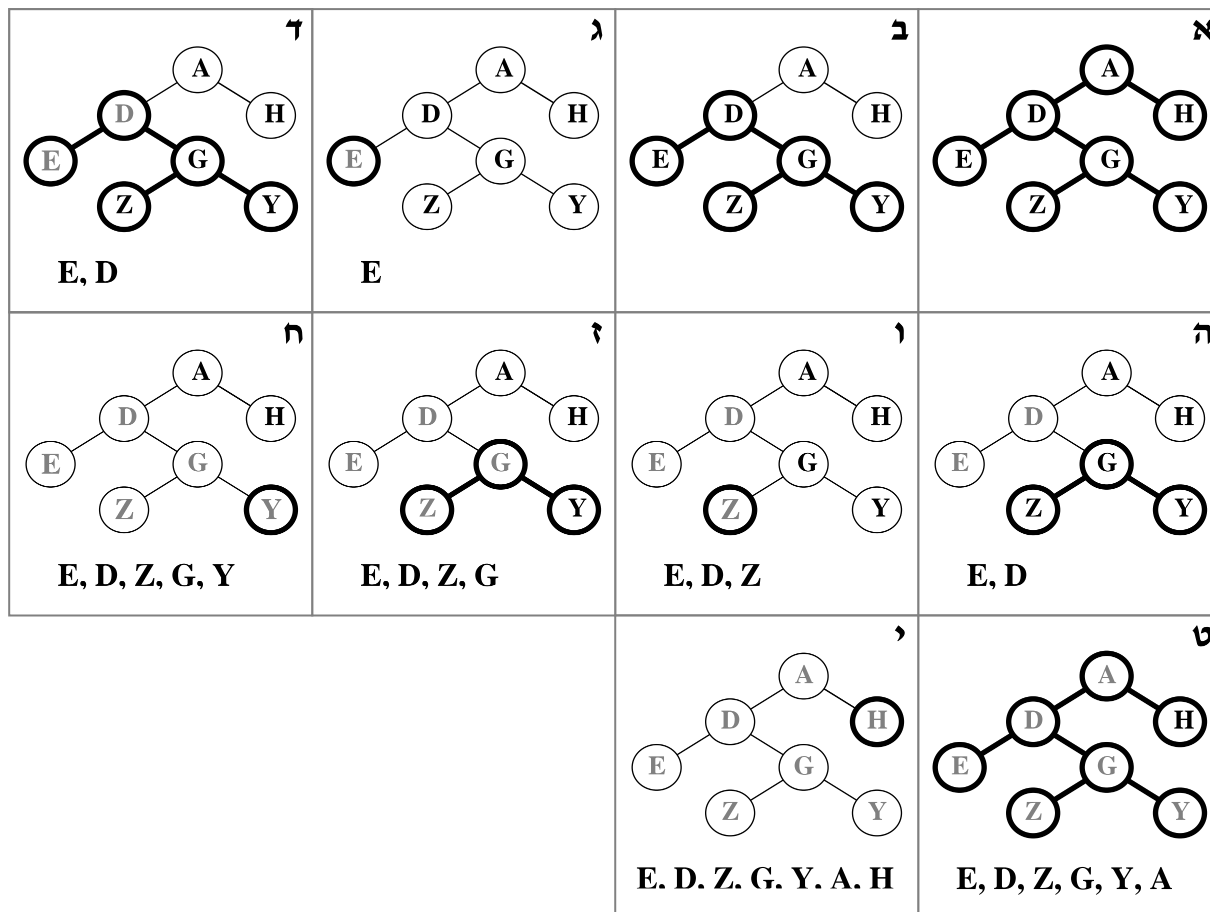
באמרנו "ביקור בשורש", אנו מתכוונים לביצוע פעולה כלשהי בצומת תוך כדי הסריקה, למשל הדפסת ערכו של הצומת. נשים לב כי במהלך סריקה אנו מבקרים פעם אחת בכל צומת, אך אנו חולפים על פני הצמתים פעמים נוספות בלי לבצע "ביקור", וזאת כדי להגיע אל ילדיהם של אותם צמתים או כדי לחזור דרכם אל הוריהם.

אם הביקור בשורש מתבצע ראשון, הסריקה נקראת **סריקה בסדר תחילי**. אם הביקור בשורש העץ מתבצע בשלב שני (בין שתי הסריקות הרקורסיביות), הסריקה נקראת **סריקה בסדר תוֹכי**. כאשר הביקור בשורש העץ מתבצע אחרון, לאחר שתי הסריקות הרקורסיביות, הסריקה נקראת **סריקה בסדר סופי**.

שימו לב כי בכל הסריקות נעשית קודם סריקה של התת-עץ השמאלי ולאחר מכן סריקה של התת-עץ הימני. ניתן כמובן לשנות את הסדר ולסרוק קודם את התת-עץ הימני לפני השמאלי. במקרה זה יתקבלו שלוש סריקות נוספות, סימטריות לשלוש הקודמות. בספרות מקובל לסרוק את התת-עץ השמאלי לפני הימני.

בכל שלוש הסריקות, כאשר תת-עץ (שמאלי, ימני או שניהם) אינו קיים, לא ממשיכים לרדת בכיוון זה של העץ. בכל שלוש הסריקות קיימת אפשרות להפסיק את הסריקה כאשר התקיים תנאי מסוים, למשל כאשר נמצא הנתון שאותו אנו מחפשים.

נבחן את האיור הבא המדגים סריקה בסדר תוֹכי של עץ נתון המכיל מספרים שלמים. הביקור המתבצע בשורש הוא הדפסת ערכו, והוא מתבצע בין שתי הסריקות הרקורסיביות. הצמתים המודגשים באיור באים להבליט את התת-עץ המטופל ברגע מסוים, התת-עץ הנוכחי. למעשה גם מתוך תת-עץ זה רק שורשו הוא החשוב לנו בכל שלב בסריקה.



בשלב א מתחילה הסריקה בשורש העץ – הצומת A. נשים לב כי אנו אמנם עוברים ב-A, אך איננו מבקרים בו, שכן הביקור יתבצע רק לאחר סריקת התת-עץ השמאלי. בשלב ב יש לבצע סריקה בסדר תוֹכי של התת-עץ השמאלי (ששורשו D). בשלב ג נסרק התת-עץ השמאלי של תת-עץ זה, כלומר העלה E. כיוון שזה עלה מתבצע בו ביקור. בשלב ד חוזרים בסריקה לצומת D ומבקרים בו. בשלבים ה–ח נסרק התת-עץ הימני ששורשו G, לפי סדר זה: Z, אחריו G ואחריו Y. בשלב ט מבוצע הביקור ב-A, שהוא שורש העץ כולו. בשלב י מבוצעת הסריקה של התת-עץ הימני של העץ. כאן יש צומת יחיד H והביקור בו הוא צעד יחיד.

בעוד סריקה בסדר תוֹכי של העץ שבאיור החזירה את הסדרה: E, D, Z, G, Y, A, H

הרי סריקה בסדר תחילי תחזיר סדרה המכילה אותם ערכים אך בסדר שונה:

A, D, E, G, Z, Y, H

✍ עקבו אחרי סריקה בסדר סופי של העץ המופיע באיור לעיל וכתבו את סדרת הערכים המתקבלת.

ו.1.1. יעילות הסריקות

נעריך את סדר הגודל של זמן הריצה של האלגוריתמים הרקורסיביים המתוארים לעיל לסריקה של עץ בינרי. הפעולה הבסיסית שמבצע כל אחד מהאלגוריתמים היא הביקור בצומת (שורש של תת-עץ). ההוראות הבסיסיות הנלוות לביקור כוללות את הבדיקות האם קיימים תת-עצים בשמאל ובימין, את שני המעברים (לכל היותר) דרך הצומת שאינם ביקור וכן את החזרה לאחר סיום ביצוע הפעולה על התת-עץ שהצומת הוא שורשו. בסך הכול, מספר ההוראות הנלוות לביקור הוא קבוע, ולכן משך הזמן שיידרש לביצוען הוא קבוע גם כן. בשלושת האלגוריתמים הסריקה מבצעת ביקור יחיד בכל צומת של העץ, לכן זמן הריצה של כל אחת מהסריקות הוא לינארי בגודל העץ (שהוא מספר צמתיו). סדר הגודל של היעילות הוא $O(n)$.

ו.2.1. שימוש בסריקות של עץ

כאשר אנו רוצים לבצע משימה המצריכה מעבר על פני כל הצמתים בעץ, עלינו להשתמש באחת מהסריקות שהוצגו. בסעיף ה.1. בדוגמה 1 השתמשנו בסריקה בסדר סופי, ובדוגמה 2 השתמשנו בסריקה בסדר תחילי של עץ. בחירה בכל סריקה אחרת הייתה מסייעת בביצוע המשימות באופן דומה. נבחן כמה דוגמאות נוספות של פעולות המשתמשות בסריקות של עצים.

דוגמה 1: בדיקת הימצאות איבר בעץ

נכתוב פעולה המקבלת עץ שצמתיו מכילים מספרים שלמים וערך שלם נוסף. הפעולה משתמשת בסריקה בסדר תחילי כדי לבדוק האם הערך נמצא בעץ.

```
public static boolean exists(BinNode<Integer> bt, int x)
{
      if (bt == null)
            return false;

      if (bt.getValue() == x)
            return true;

      return exists(bt.getLeft(), x) || exists(bt.getRight(), x);
}
```

בעיה זו אינה מחייבת שימוש בסריקה בסדר תחילי דווקא. כל מעבר רקורסיבי מתאים לפתרונה. כאשר יימצא הערך הנתון לא יהיה צורך להמשיך את הסריקה, ולכן לפני שבודקים אם הערך נמצא באחד מהתת-עצים שמתחת לחוליה הנוכחית, עדיף לבדוק האם הערך נמצא בחוליה עצמה. כך, למשל, אם הערך נמצא בשורש העץ, הבדיקה תסתיים בהצלחה אחרי שנבדוק את חוליית השורש בלבד.

דוגמה 2: מחרוזת המתארת את העץ

נכתוב פעולה המקבלת עץ ומחזירה מחרוזת המתארת את תוכנו. אנו מסתמכים על העובדה כי לחוליה יש פעולת toString() המחזירה את הערך שבה כמחרוזת. כיוון שניתן לסרוק את העץ בשלושה אופנים, ניתן להגדיר למעשה שלוש פעולות מתאימות:

preorderString(…), postorderString(…), inorderString(…)

פעולות אלה יחזירו תיאורים שונים של העץ על פי הסדר שהוגדר בשמן.

נממש את הפעולה המחזירה מחרוזת שבה ערכי העץ מסודרים על פי סריקה בסדר תחילי:

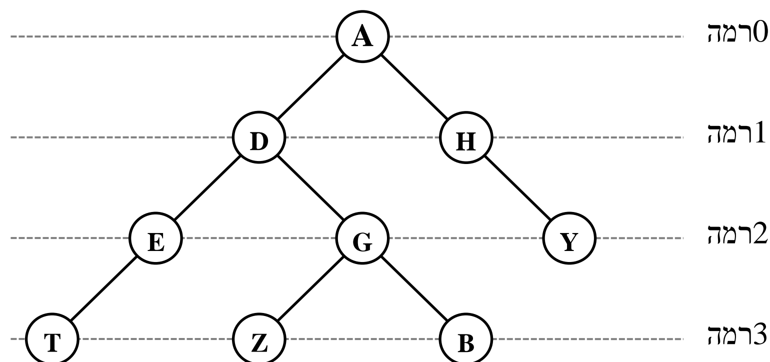
```
public static String preorderString(BinNode<String> bt)
{
   if (bt == null)
           return "";
      return bt.getValue() + " " + preorderString(bt.getLeft()) +

preorderString(bt.getRight());
}
```

✍ ממשו את הפעולות postorderString(…) ו- inorderString(…).

ו.2. סריקה לפי רמות (סריקה לרוחב)

עץ הוא מבנה היררכי המחולק לרמות. השורש נמצא ברמה אחת, ילדיו ברמה הבאה וכן הלאה. במקרים מסוימים עולה הצורך לסרוק את העץ לרוחבו, לפי רמות.

רמה (level) של צומת מסוים בעץ היא אורך המסלול מהשורש אל צומת זה, כלומר המרחק של הצומת מהשורש. רמת השורש היא 0, והרמה של כל צומת אחר בעץ גדולה באחד מהרמה של ההורה שלו. **גובה עץ (tree height)** הוא המרחק הגדול ביותר מהשורש לעלה כלשהו של העץ, כלומר זו הרמה הגבוהה ביותר של עץ. גובה העץ שלפניכם הוא 3.



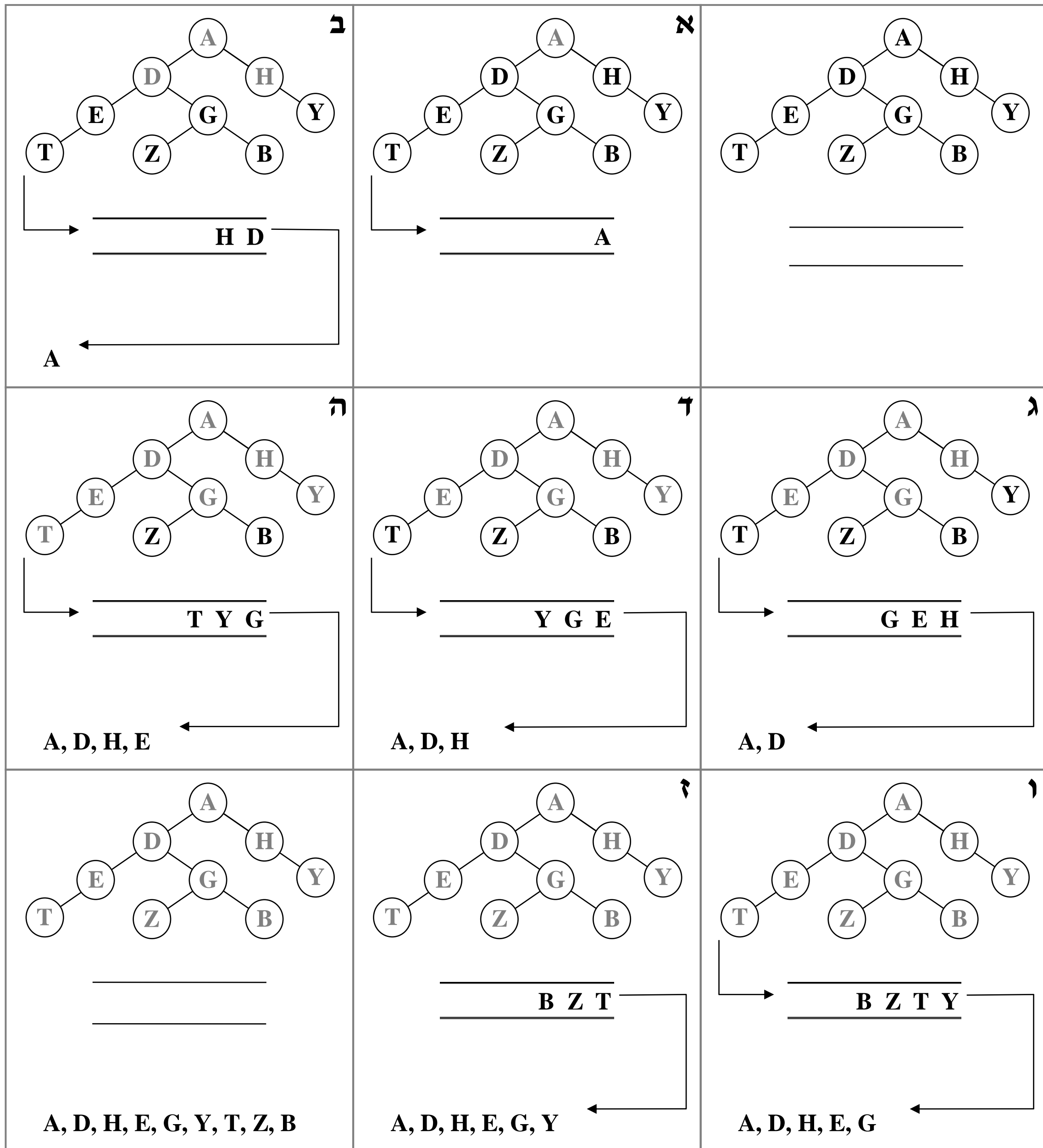
1.2.ו. אלגוריתם סריקה לפי רמות

בסריקת עץ לפי רמות, הסריקה מתבצעת רמה אחת אחרי קודמתה החל בשורש, כשבכל רמה נסרקים הצמתים משמאל לימין. כדי לממש סריקה כזו, נצטרך להשתמש ברעיון אלגוריתמי חדש.

נעיין במהלך הסריקה של העץ שבאיור. תחילה נבקר ב-A ואחר כך בשני ילדיו, D ו-H. לאחר הביקור ב-H עלינו לבקר ב-E. כיצד נעבור מהצומת H ל-E? הרי באמצעות הפעולות שברשותנו ניתן לגשת לילד רק דרך ההורה שלו, ואילו E אינו ילד של H אלא של D. כיצד נעבור מ-G ל-Y כשנמשיך בסריקה, הרי הם אינם אחים?

שימו לב כי כשאנו מבקרים ב-D, ידוע לנו כי בעתיד נרצה לבקר בילדיו E ו-G, בסדר זה. גם כאשר אנו ממשיכים ל-H, אנו יודעים שבעתיד נרצה לבקר בילדיו (במקרה זה רק אחד). באופן כללי, כאשר אנו מבקרים משמאל לימין בצמתים השייכים לרמה מסוימת בעץ, אנו יודעים כי בעתיד נרצה לבקר באותו סדר בילדים של צמתים אלה, שהם הצמתים ברמה הבאה. כדי לממש רצון זה נשתמש בטיפוס האוסף תור. כזכור, האיברים מתוספים לתור בסופו ויוצאים מתחילתו, ולכן האיבר שנכנס ראשון לתור הוא זה שיוצא ממנו ראשון. התור שניעזר בו יהיה מטיפוס חוליה בינרית כדי שנוכל לאחסן בו את שורשי התת-עצים שטרם נסרקו. תחילה נכניס לתור הריק את שורש העץ. בהמשך, בכל צעד נוציא שורש של תת-עץ מהתור, נבקר בו ונכניס את ילדיו (שניים, אחד או אפס) לסוף התור. נמשיך כך עד שיתרוקן התור.

האיור שלפניכם מציג את אופן הסריקה של עץ לפי רמות. בכל שלב מוצגים מצבו של העץ, שורשי העצים הנמצאים בתור ורשימת האיברים שנסרקו. שימו לב שראשו של התור מוצג בצד ימין וזנבו בשמאל.



להלן האלגוריתם הסורק את העץ הבינרי לפי רמות:

***סרוק-לפי-רמות** (tree)*
בנה תור חדש של חוליות בינריות
הכנס את החוליה tree לתוך התור
כל עוד התור אינו ריק, בצע את הפעולות:
הוצא חוליה מתוך התור
בקר בחוליה
אם קיים לחוליה ילד שמאלי, הכנס אותו לתור
אם קיים לחוליה ילד ימני, הכנס אותו לתור

✍ ממשו פעולה בשם levelOrderString המקבלת עץ חוליות בינרי של מחרוזות ומחזירה מחרוזת המתארת את תוכן העץ המסודר לפי רמות העץ.

כדי לכתוב את הקוד יהיה עליכם להגדיר תור של חוליות בינריות מטיפוס מחרוזת. נזכור כי גם החוליה הבינרית וגם התור הם מחלקות גנריות, ויש לקבוע טיפוס קונקרטי לכל אחת מהן. כלומר, את הטיפוס של החוליה הבינרית עלינו לכתוב עבור הטיפוס הקונקרטי מחרוזת. הטיפוס המתקבל הוא הטיפוס הקונקרטי לתור. הגדרת התור תיראה כך:

```
Queue<BinNode<String>>
```

ו.2.2. יעילות הסריקה לפי רמות

נעריך את יעילותו של האלגוריתם. בסריקה לפי רמות הפעולה הבסיסית היא הוצאה מהתור. נלוות אליה הפעולות האלה: ביקור בשורש העץ והכנסת שני התת-עצים שלו לתור (שניים לכל היותר). הנחה (סבירה מאוד) היא שמימוש התור מבטיח שפעולות ההוצאה וההכנסה של איברים אורכות זמן קבוע. אם כך, מחיר הפעולה הבסיסית והפעולות הנלוות אליה הוא קבוע. כל שורש עץ מוכנס לתור בדיוק פעם אחת. מכאן שכמו באלגוריתמים הרקורסיביים לסריקת עצים, גם אלגוריתם הסריקה לפי רמות מבקר פעם אחת בדיוק בכל אחד מצומתי העץ, ולכן יעילות זמן הריצה של האלגוריתם היא לינארית בגודל העץ.

ז. שימוש בעץ חוליות בינרי – ייצוג ביטוי חשבוני

בסעיף זה נציג שימוש בעץ חוליות בינרי לייצוג ביטוי חשבוני (שהוא מחרוזת המכילה פעולות חשבון ומספרים).

שימוש נפוץ בייצוג הזה הוא תוכנית המקבלת ביטוי חשבוני ומחשבת את ערכו. בשלב הראשון, התוכנית מייצגת את הביטוי כעץ בינרי. בשלב השני היא מחשבת את ערכו על ידי אלגוריתם רקורסיבי פשוט (גם מהדר, כגון המהדר של ג׳אווה, מייצר קוד לחישוב ביטוי חשבוני על פי גישה דומה). נבחן מהלך דו-שלבי זה בפירוט.

נניח כי ביטוי חשבוני מכיל מספרים ואת פעולות החשבון: חיבור, חיסור, כפל וחילוק. הפעולות נקראות אופרטורים, ואילו הארגומנטים שלהן (שהם מספרים או ביטויים) נקראים אופרנדים.

כדי לפשט את הטיפול בביטוי נניח שהמספרים הם חד-ספרתיים, וכן נניח שהביטוי עצמו וכל תת-ביטוי שלו אינם ביטויים מהצורה x-. כמו כן, כדי לחסוך את הטיפול בקדימויות שבין האופרטורים, נניח שהביטוי החשבוני "ממוסגר לחלוטין", כלומר סביב כל אופרטור ושני האופרנדים שעליהם הוא פועל מופיעים סוגריים.

לדוגמה, הביטויים א–ג חוקיים (מקיימים את ההנחות):

א. $(7 + 5)$

ב. $((3 * 4) + 2)$

ג. $((3 – 2) * ((4 * 1) + 8))$

ולעומתם הביטויים ד–ז אינם חוקיים (אינם מקיימים את ההנחות):

ד. $(3 + 5) * 4$

ה. $3 – 2$

ו. $(–7)$

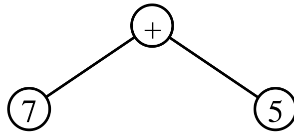
ז. (8)

את הביטויים המקיימים את ההנחות לעיל אפשר להגדיר כך:

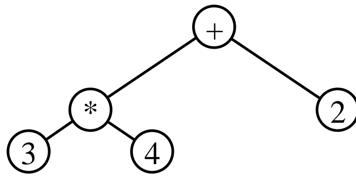
ביטוי חשבוני הוא:	**A**	כאשר A הוא מספר חד-ספרתי
או:	**(X op Y)**	כאשר X ו-Y הם ביטויים חשבוניים, האופרנדים של הפעולה, ו-op הוא פעולה.

שימו לב כי זוהי הגדרה רקורסיבית: האופרנדים X ו-Y בביטוי מורכב הם ביטויים חשבוניים בעצמם, כלומר תת-ביטויים של הביטוי המלא. כיצד ניתן לייצג ביטויים כאלה? קל לראות התאמה בין הגדרת ביטוי חשבוני להגדרה של עץ בינרי: החלק הראשון של ההגדרה – ביטוי שהוא מספר – מתאים לעץ עלה (שורש ללא ילדים); החלק השני – ביטוי מורכב – מתאים לשורש עם שני ילדים. משום כך, טבעי לייצג ביטוי חשבוני בעזרת עץ בינרי. צומת בעץ יוכל להכיל אחד משני סוגי נתונים: פעולה חשבונית או מספר. צומת שיש לו תת-עצים (צומת פנימי), יכיל פעולה, שאותה צריך להפעיל על האופרנדים שהם הביטויים המיוצגים על ידי התת-עצים השמאלי והימני של אותו הצומת. עלי העץ יכילו מספרים. נשים לב כי בעץ בינרי המייצג ביטוי חשבוני כפי שהגדרנו אותו אין צומת שיש מתחתיו תת-עץ אחד בלבד. באיור שלפניכם מיוצגים ביטויים חשבוניים באמצעות עצים בינריים.

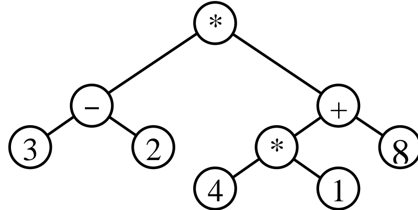
א. הביטוי (5 + 7):



ב. הביטוי (2 + (4 * 3)):



ג. הביטוי ((8 + (1 * 4)) * (2 – 3)):



✍ ציירו את עץ הביטוי עבור: ((4 + (7 * 8)) – (9 / 3))

בהינתן עץ המייצג ביטוי חשבוני תקין, ניתן לחשב את ערכו על פי האלגוריתם הבא, המנצל את מבנהו הרקורסיבי של עץ הביטוי, כדי להעריך אותו. הסריקה של העץ נעשית הפעם בסדר סופי, משום שזהו הסדר היחיד המאפשר את חישוב ערכו של הביטוי בצורה נכונה.

חשב-ערך-ביטוי *(tree)*

אם העץ הוא עלה, החזר את ערכו

אחרת,

חשב-ערך-ביטוי (תת-עץ שמאלי של *tree) ושמור את התוצאה ב- leftVal.*

חשב-ערך-ביטוי (תת-עץ ימני של *tree) ושמוראת התוצאה ב- rightVal.*

שמור ב-op את הערך של השורש

החזר את: (leftVal op rightVal). {תוצאת הפעולה op על שני הערכים}.

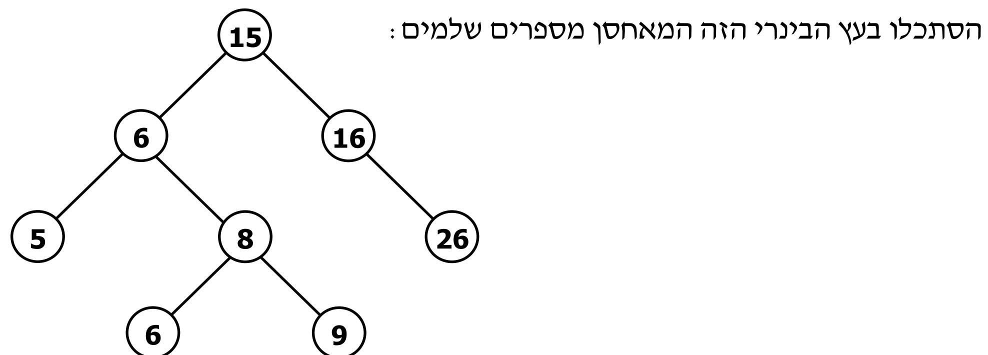
✍ ממשו את האלגוריתם **חשב-ערך-ביטוי** כפעולה בשם computeExprTree(…).

ח. עץ-חיפוש-בינרי

ייצוג של קבוצה או של סדרת ערכים באמצעות עץ בינרי יהיה ייצוג נחות לעומת ייצוגם בעזרת רשימה מקושרת. אמנם ניתן לסרוק עץ במחיר לינארי של מספר איבריו, בדיוק כמו בסריקת רשימה, אך נראה שקל יותר להכניס איבר לרשימה במקום רצוי מאשר לעץ, וכן להוציא ממנה איבר שמקומו נתון. אלה אינן פעולות קלות לביצוע בעץ בינרי. בסעיף זה נראה כי אם הערכים

שאנו רוצים לשמור באוסף לקוחים מתחום שיש בו יחס סדר, כגון מספרים, תווים ומחרוזות, אזי יש סוג מיוחד של עצים בינריים שמאפשר ייצוג יעיל של האוסף, בעזרת שימוש מושכל בסדר של הערכים. ייצוג זה בעזרת עץ בינרי יהיה עדיף על ייצוג בעזרת רשימה.

הסתכלו בעץ הבינרי הזה המאחסן מספרים שלמים:



הערכים הנמצאים בַּתת-עץ השמאלי הם: 5, 9, 8 ופעמיים הערך 6, וכולם קטנים מ-15, שהוא הערך בשורש. הערכים 16 ו-26, הנמצאים בתת-עץ הימני, גדולים מ-15. באותו האופן, הערך 5, הנמצא בתת-עץ השמאלי של הצומת 6, קטן מ-6, ואילו הערכים 6, 9 ו-8, הנמצאים בתת-עץ הימני של הצומת, גדולים ממנו או שווים לו. למעשה, העץ מאורגן באופן זה: כל הערכים הנמצאים בתת-עץ השמאלי של צומת כלשהו קטנים ממש מהערך שבצומת, וכל הערכים הנמצאים בתת-עץ הימני של הצומת גדולים מערך זה או שווים לו.

עץ בינרי המאורגן בצורה זו – כלומר שלכל צומת בו, כל הערכים בתת-עץ השמאלי שלו קטנים מהערך בצומת, וכל הערכים בתת-עץ הימני שלו גדולים או שווים לערך בצומת – נקרא **עץ-חיפוש-בינרי (binary search tree)**. כמובן, דרך ארגון זו יכולה להתקיים רק אם הערכים שבעץ לקוחים מתחום שיש בו יחס סדר, כלומר שהערכים הם ברי השוואה.

נדון בתכונות של עצים כאלה, תוך הדגשת היתרונות הנובעים מהארגון המיוחד שלהם.

הערה: יש שימושים של עץ-חיפוש-בינרי שבהם אסור שערך יופיע בעץ יותר מפעם אחת. בעץ כזה מתקיים שכל הערכים הנמצאים בַּתת-עץ השמאלי של צומת כלשהו קטנים מהערך שבצומת, וכל הערכים הנמצאים בתת-עץ הימני של הצומת גדולים מערך זה. אנו נמשיך את דיוננו עבור ההגדרה המקורית של העץ.

ח.1. איתור ערך בעץ-חיפוש-בינרי

מבנהו המיוחד של עץ-חיפוש-בינרי מאפשר ביצוע חיפוש של ערך בעץ, כלומר בירור האם הערך קיים בצומת כלשהו של העץ באופן מהיר ופשוט הרבה יותר מחיפוש בעץ בינרי רגיל. לדוגמה, נציג את החיפוש של הערך 25 בעץ שהצגנו למעלה. השוואה של הערך 25 לערך 15 שבשורש העץ מספרת לנו כי: 25 אינו הערך שבשורש העץ, ואם הוא קיים בעץ, אזי מקומו בתת-עץ הימני של השורש. בצעד הבא נשווה את 25 לערך שבילד הימני של השורש. כיון שילד זה מכיל את הערך 16, אנו לומדים כי 25 אינו הערך שבו, ואם 25 קיים בעץ, אזי מקומו בתת-עץ הימני שלו. השוואה שלישית, הפעם לילד הימני של צומת זה, המכיל 26, תוביל אותנו לתת-עץ השמאלי של צומת זה.

כיון שתת-עץ זה אינו קיים, אנו מגיעים למסקנה שהערך 25 אינו קיים בעץ. באופן דומה, אם הערך בחיפוש היה 26, היינו מוצאים אותו בעץ כבר בהשוואה השלישית, ומפסיקים את החיפוש.

באופן כללי, חיפוש בעץ-חיפוש-בינרי מתחיל בשורש, ובכל שלב יורד רמה אחת שמאלה או ימינה, עד למציאת הערך או עד להגעה לתת-עץ שאינו קיים, המעיד שהערך אינו קיים בעץ. אם הערך קיים בעץ יותר מפעם אחת, החיפוש ייעצר כאשר יגיע לצומת המכיל ערך זה, ולא ימשיך לחפש צמתים נוספים המכילים אותו. בחיפוש כזה, השוואה של הערך שאותו מחפשים לערך שבצומת העץ נותנת מענה לשאלות האלה:

1. האם מצאנו צומת המכיל את הערך?
2. אם לא, באיזה תת-עץ יש להמשיך את החיפוש?

שמו של סוג עץ זה, **עץ-חיפוש-בינרי**, נבחר משום שהמבנה המיוחד של העץ, המסתמך על סדר הקיים בין הערכים, מאפשר לבצע בו חיפוש יעיל.

להלן פעולה המקבלת עץ המאורגן כעץ-חיפוש-בינרי ובודקת האם הערך x נמצא בו. הפעולה מחזירה 'אמת' אם x נמצא בעץ, ו-'שקר' אם אינו נמצא בו. הפרמטר המייצג את העץ שבו החיפוש מתבצע נקרא bst (שהם ראשי התיבות של **b**inary **s**earch **t**ree). הפעולה מניחה כי ערך הפרמטר בזימון כלשהו הוא הפניה לעץ המאורגן כעץ-חיפוש-בינרי.

```
public static boolean existsInBST(BinNode<Integer> bst, int x)
{
            if (bst == null)
                  return false;

      if (bst.getValue() == x)
                  return true;

            if (bst.getValue() < x)
                  return existsInBST(bst.getLeft(), x);

            return existsInBST(bst.getRight(), x);
}
```

הערה: ניתן לבצע את תהליך החיפוש גם באופן איטרטיבי, כאשר על סמך תוצאות ההשוואה של x לשורש נתון מחליטים אם להמשיך את החיפוש בתת-עץ הימני או השמאלי שלו.

כתבו גרסה איטרטיבית של הפעולה existsInBST(…).

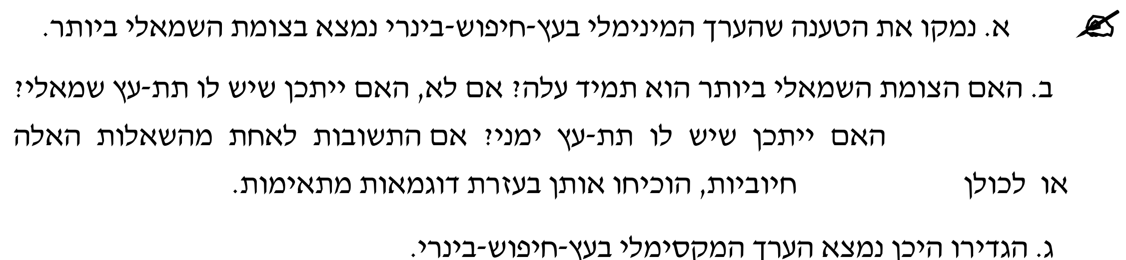
שימו לב כי קיים דמיון רב בין שיטת חיפוש זו ובין חיפוש בינרי במערך ממוין. בחיפוש בינרי במערך ממוין משווים את הערך המבוקש לאיבר האמצעי במערך, ולפי התוצאה מחליטים באיזה חלק של המערך להמשיך את החיפוש, בשמאלי או בימני. בעץ–חיפוש-בינרי משווים את הערך המבוקש לערך שבשורש העץ, ולפי התוצאה מחליטים אם להמשיך את החיפוש משמאלו או מימינו.

החיפוש בעץ-חיפוש-בינרי יעיל יותר משימוש בפעולה exists(...) לחיפוש ערך בעץ בינרי רגיל (שהוצגה בדוגמה 2 בסעיף ו.2.1.). במקרה הגרוע ביותר, כאשר הערך המבוקש אינו קיים בעץ, הפעולה exists(...) עוברת על כל הצמתים. גם כאשר הערך קיים בעץ, הפעולה יכולה במקרים מסוימים לעבור על רוב הצמתים או אפילו על כולם, ולכן סדר הגודל שלה הוא לינארי במספר הצמתים שבעץ. בעץ-חיפוש-בינרי בכל שלב בחיפוש אנו "עוזבים" את התת-עץ שבו ברור שאין לנו מה לחפש, וממשיכים לחפש רק בתת-עץ שבו יש סיכוי לאתר את הערך המבוקש. בכל מקרה, בין אם הערך קיים בעץ ובין אם אינו קיים בו, החיפוש עובר רק על מסלול אחד, המתחיל בשורש העץ. אם הערך קיים בעץ, המסלול מסתיים בצומת המכיל אותו. אם הערך אינו קיים בעץ, המסלול מסתיים בצומת שהערך המבוקש אמור היה להימצא בתת-עץ שלו, אלא שאותו התת-עץ אינו קיים. אורכו של המסלול חסום על ידי גובה העץ. דיון מפורט יותר ביעילות החיפוש בעץ-חיפוש-בינרי מוצג בהמשך.

הערה: עבור עץ-חיפוש-בינרי מוגדרת פעולת חיפוש יעילה אחר ערך נתון. אך מדוע שיהיה לנו עניין בחיפושים אלה? לחיפושים אלה יכולים להיות שימושים מעניינים אם לכל ערך השמור בעץ יוצמד מידע נוסף, למשל, ייתכן כי הערך בעץ הוא מספר תעודת זהות, והמידע הנוסף הוא פרטי בעל התעודה, או שהערכים בצומתי העץ הם שמות ואליהם מצורפים מספרי טלפון. בתרגיל המסכם – "מפה", נרחיב בנושא זה.

ח.2. מציאת ערך מינימלי בעץ-חיפוש-בינרי

ברשימה כללית מציאת הערך המינימלי דורשת סריקה של כל החוליות ברשימה. באופן דומה, מציאת הערך המינימלי בעץ בינרי דורשת סריקה של כל הצמתים שלו. כאשר המבנה סדור, נצפה למצוא את הערך המינימלי בלי שנצטרך לעבור על כל הערכים בעץ. ואמנם ברשימה ממוינת בסדר עולה, מציאת הערך הקטן ביותר היא פעולה פשוטה ביותר, משום שהוא ממוקם בראשית הרשימה. מה לגבי מציאת הערך הקטן ביותר בעץ-חיפוש-בינרי? קל לראות, כי ערך זה ממוקם בצומת השמאלי ביותר בעץ, וניתן להגיע אליו בירידה עקבית שמאלה, המתחילה בשורש העץ.


✍ א. נמקו את הטענה שהערך המינימלי בעץ-חיפוש-בינרי נמצא בצומת השמאלי ביותר.

ב. האם הצומת השמאלי ביותר הוא תמיד עלה? אם לא, האם ייתכן שיש לו תת-עץ שמאלי? האם ייתכן שיש לו תת-עץ ימני? אם התשובות לאחת מהשאלות האלה או לכולן חיוביות, הוכיחו אותן בעזרת דוגמאות מתאימות.

ג. הגדירו היכן נמצא הערך המקסימלי בעץ-חיפוש-בינרי.

ראינו שמציאת ערך קיצוני בעץ-חיפוש-בינרי דורשת לעבור על מסלול אחד בעץ. כלומר אם ננצל את המידע הנתון לנו על מבנהו של עץ-החיפוש נוזיל מאוד את הפעולות המחזירות את הערכים הקיצוניים השמורים בעץ.

ח.3. הכנסת ערכים לעץ-חיפוש-בינרי ובניית העץ

לעץ בינרי רגיל אפשר להכניס ערך בתוך עלה חדש, כילד שמאלי של כל צומת שאין לו ילד שמאלי, וכילד ימני של כל צומת שאין לו ילד ימני. אילו היינו כותבים פעולה המבצעת הכנסת ערך היינו צריכים לכלול בפרמטרים את הערך, את צומת ההורה ואת מיקום הצומת החדש מתחת להורה. לא כך הדבר בעץ-חיפוש-בינרי. כאן התנאי המגדיר את מבנה עץ-החיפוש קובע כיצד נוסיף לו ערך: אם הערך קטן מזה שבשורש, יש להכניסו לתת-עץ השמאלי של השורש; אחרת, יש להכניסו לתת-עץ הימני. כך ממשיכים לרדת בעץ, עד שמגיעים למצב שבו אין תת-עץ בכיוון הנדרש, ושם ניתן להוסיף את הערך החדש כעלה.

קל לראות שפעולת ההכנסה, כפי שתיארנו אותה עכשיו, שומרת על תכונת העץ: שכן אחרי הכנסת הערך, העץ עדיין עץ-חיפוש-בינרי. יותר מזה, המקום להכנסה שהפעולה בוחרת הוא **היחידי** שבו ניתן להכניס את הערך החדש תוך שמירה על מבנה העץ. כלומר, מבנהו של עץ-חיפוש-בינרי מאפשר, ואף מחייב, להגדיר פעולת הכנסה המקבלת את הערך כפרמטר יחיד; מקום הצומת החדש נקבע על ידי הפעולה עצמה, בהתאם למבנה העץ, ואינו נתון כלל לשיקול דעתו של מזמן הפעולה. לעובדה זו יתרון נוסף: בעץ בינרי רגיל, פעולה של בניית העץ יכולה לגרום לקלקול מבנהו, למשל אם מצרפים לתת-עץ השמאלי חוליה הקיימת כבר בתת-עץ הימני. בעץ-חיפוש-בינרי אין מצרפים חוליות, אלא רק מכניסים ערכים, ומיקום הערך נקבע על ידי הפעולה. כך מובטח שהמבנה יישאר עץ-חיפוש-בינרי. שימו לב שכמו פעולת החיפוש, גם פעולת ההכנסה סורקת את המסלול המתחיל בשורש ויורד בעץ.

ממשו את הפעולה:

```
public static void insertIntoBST(BinNode<Integer> bst, int x)
```

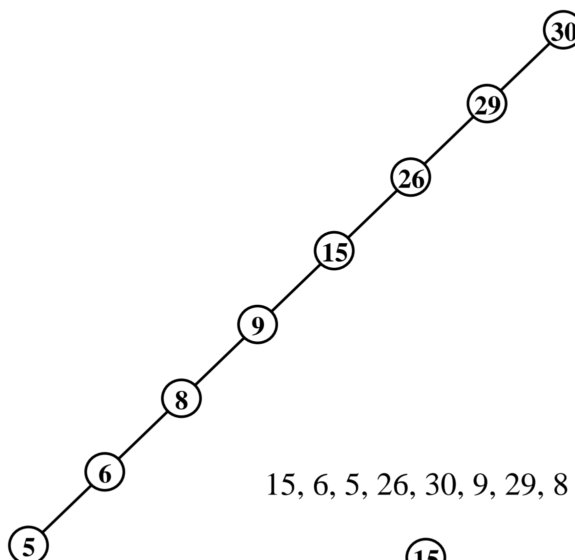
הפעולה מכניסה מספר שלם לעץ-חיפוש-בינרי המכיל מספרים.

כאשר עסקנו בעצים בינריים רגילים בנינו אותם על ידי צירוף צמתים זה לזה באופן שרירותי. בעץ-חיפוש-בינרי כמובן אי אפשר לצרף צמתים באופן שרירותי, ולמעשה אין מצרפים צמתים כלל, אלא מכניסים ערכים לעץ. רק הפעולה להכנסת ערך יוצרת חוליה חדשה ומצרפת אותה לעץ במקום המתאים. העובדה שיש לנו פעולת הכנסה לעץ-חיפוש-בינרי, השומרת על תכונותיו, מאפשרת לנו לבנות עץ-חיפוש-בינרי המכיל אוסף נתון של ערכים באופן זה: נבנה עץ עלה, המכיל את אחד הערכים. לאחר מכן נוסיף את שאר הערכים אחד אחרי השני, תוך שימוש בפעולת ההכנסה. מובטח לנו שנקבל עץ-חיפוש-בינרי המכיל את כל הערכים (וגם את המידע הנוסף הצמוד לכל ערך, אם יש כזה).

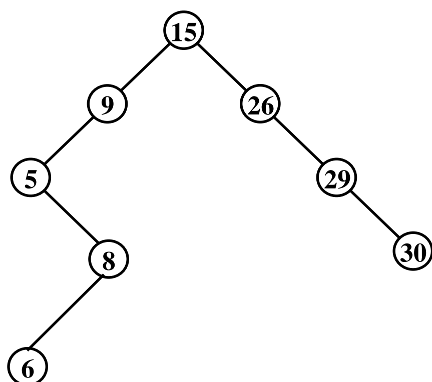
האם לסדרה נתונה של ערכים מתקבל תמיד אותו העץ, בלי תלות בסדר ההכנסה של הערכים?

האיור שלפניכם מציג כמה עצי חיפוש שנבנו על פי סדרות ערכים שונות, שכולן מכילות בדיוק את הערכים 5, 6, 8, 9, 15, 26, 29, 30. הערכים בכל סדרה הוכנסו לפי סדרם, משמאל לימין.

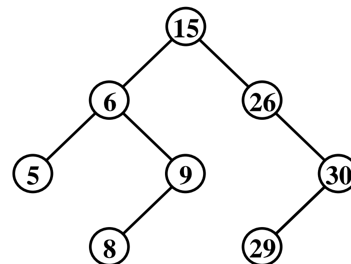
א. 30, 29, 26, 15, 9, 8, 6, 5



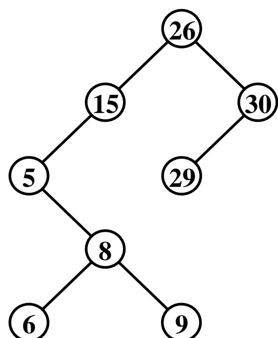
ב. 15, 9, 26, 5, 8, 6, 29, 30



ג. 15, 6, 5, 26, 30, 9, 29, 8



ד. 26, 30, 15, 5, 29, 8, 9, 6



✍ בדקו כי העצים שבאיורים אכן נוצרו על פי הסדרות המתאימות, על ידי הכנסת הערכים שבהן משמאל לימין. הציגו סדרה נוספת המכילה אותם הערכים, אך עץ-החיפוש הבינרי הנבנה ממנה שונה מכל העצים שבאיור.

ראינו שקיימים עצי חיפוש בינריים שונים המכילים אותם הערכים, וראינו שצורת העץ הסופית תלויה בסדר הכנסת הערכים לתוכו. ייתכן מצב ששני סידורים שונים של ערכים ברשימה יובילו לבנייתו של אותו עץ חיפוש. כך, למשל, עץ ג באיור יתקבל גם מהכנסת הערכים שבסדרה

15, 26, 6, 9 30, 5, 8, 29 (משמאל לימין).

✍ עץ א לעיל נוצר על פי סדרת ערכים הממוינת בסדר יורד. מה תהיה צורתו של העץ שייווצר על ידי אותם ערכים הממוינים בסדר עולה?

ח.4. הוצאת ערך מעץ-חיפוש-בינרי

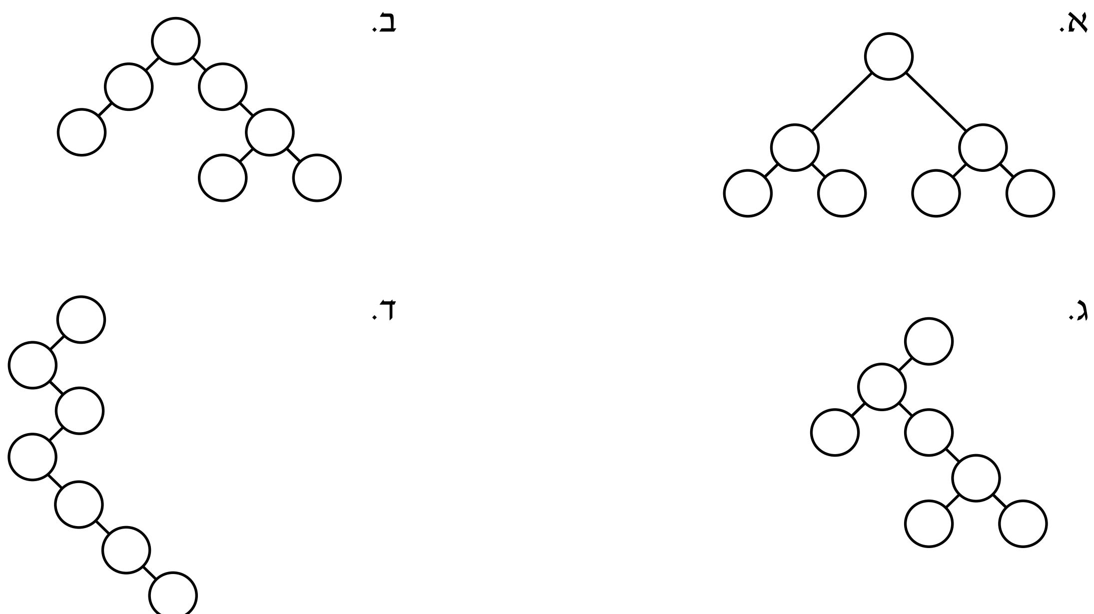
בהינתן ערך x, נחפש האם הוא קיים בעץ. אם x אינו קיים בעץ, אין צורך לעשות דבר. אחרת, אלגוריתם החיפוש מגלה את הצומת הראשון (בסריקה בסדר תוֹכי), המכיל אותו. בשלב ראשון ננתק מן העץ את התת-עץ שצומת זה הוא שורשו. בשלב שני נתחיל להכניס בחזרה לעץ-החיפוש את הערכים שבתת-עץ שנותק. בעת ההכנסה לעץ לא יוכנס הערך שבשורש התת-עץ (זהו ה-x שרצינו להוציא). שימו לב ש-x יכול להימצא בעץ יותר מפעם אחת (בתת-עץ הימני של השורש המכיל את x). מכיוון שאנו מוציאים מהעץ רק את המופע הראשון של x, יש לוודא בשלב ההכנסה שהמופעים הנוספים של x, אם קיימים, יוכנסו חזרה לעץ.

ח.5. יעילות הפעולות על עץ-חיפוש-בינרי

אם נשווה זה לזה את העצים שבאיורים א ו-ג לעיל, נראה שהם אמנם מכילים אותם ערכים, אך חיפוש בעץ הראשון יהיה יעיל פחות מאשר בעץ השני, כיוון שהראשון גבוה יותר. גם פעולת הכנסה לעץ גבוה יעילה פחות מאשר לעץ נמוך, כיוון שגם פעולת ההכנסה עוברת על מסלול בעץ. עד כה לא דנו בפירוט ביעילותן של פעולות החיפוש וההכנסה בעץ-חיפוש-בינרי, אם כי השוואתן לפעולות חיפוש דומות בעץ בינרי רגיל מראה בבירור שעדיף להשתמש בעץ-חיפוש-בינרי. כעת נעסוק בנושא זה, ובפרט בשאלה מה הקשר בין צורת העץ ליעילות של פעולות החיפוש בו.

לפניכם ארבעה איורים של עצים בינריים שבכל אחד מהם שבעה צמתים. עץ א הוא המאוזן מבין העצים. הוא מכיל את מספר הצמתים המקסימלי בכל אחת מהרמות, ולכן הוא הנמוך מביניהם – גובהו 2. קל לראות שזה הגובה המינימלי האפשרי לעץ המכיל שבעה צמתים. עץ ד הוא הפחות מאוזן מבין העצים. הוא מכיל צומת אחד בכל רמה, ולכן גובהו מקסימלי – 6. זהו הגובה המקסימלי לעץ המכיל שבעה צמתים. שני העצים באיורים ב ו-ג מתארים מצבי ביניים, וגובהם 3 ו-4 בהתאמה.

עץ בינרי שכל רמותיו מלאות (מכילות את מספר הצמתים המקסימלי האפשרי) נקרא **עץ בינרי מלא** (**full binary tree**).



כמה צמתים יש ברמה ה- i של עץ בינרי מלא? ברמה 0 קיים איבר אחד (השורש); בכל רמה אחרת מספר האיברים הוא פי שניים ממספר האיברים ברמה הקודמת.

רמה 0: $2^0 = 1$

רמה 1: $2^1 = 2$

רמה 2: $2^2 = 4$

⋮

רמה k: 2^k

מספר הצמתים הכולל בעץ מלא בגובה k הוא סכום מספר הצמתים בכל רמה, החל ברמת השורש (רמה 0) וכלה ברמה k:

$$2^0 + 2^1 + 2^2 + ... + 2^k = 2^{k+1}-1$$

ואמנם, בעץ א באיור למעלה מספר הצמתים הוא 7, גובהו הוא 2, ומתקיים עבורו: $2^{2+1}-1= 7$

✍ נסו להוכיח את השוויון באמצעות אינדוקציה או על ידי חישוב של סכום הנדסי.

כאשר יש בעץ בעל k רמות צומת אחד בכל רמה, קל לראות כי מספר הצמתים הוא k+1. מספר הצמתים בכל העצים האחרים בגובה k ינוע בין שני המקרים הגבוליים, ויהיה בין k+1 ובין $2^{k+1}-1$. גם לערך k שאינו גדול, ההפרש בין שני ערכים אלה משמעותי. למשל, כאשר k=5 מספר הצמתים הוא בין 6 ל-63; וכאשר k=6 מספר הצמתים הוא בין 7 ל-127.

ענינו, אם כן, על השאלה מהו מספר הצמתים המינימלי והמקסימלי בעץ שמספר רמותיו נתון (k). אבל, לצורך הדיון ביעילות הפעולות על עץ-חיפוש-בינרי, השאלה המעניינת אותנו היא השאלה ההפוכה: מהו טווח הגבהים של עץ בינרי שבו n צמתים? עלינו לשאול שאלה זו, שכן מה שמעניין אותנו הן הצורות האפשריות של העץ והגבהים האפשריים שלו, בהינתן n ערכים שאנו רוצים לאחסן בו.

אם נניח שלפנינו עץ מלא בגובה k המכיל n צמתים הרי יתקיים השוויון הזה: $n = 2^{k+1} - 1$

או השוויון הזה: $n + 1 = 2^{k+1}$

אם נפעיל log על שני האגפים: $\log_2(n+1) = \log_2(2^{k+1})$

נקבל: $k = \log_2(n+1) -1$

אולם, ברור כי לא כל מספר n יכול להיות מספר הצמתים בעץ בינרי מלא, אלא רק מספר n המקיים: n+1 הוא חזקה של שתיים.

אם אנו רוצים לבנות עץ מאוזן ככל האפשר המכיל מספר נתון n של צמתים, נעשה זאת כך: נתחיל מהשורש, וכל זמן שעוד לא הכנסנו את כל n הצמתים, נכניס ערכים לעץ לפני רמות, רמה אחר רמה. אם השוויון:

$n = 2^{k+1} - 1$

אינו מתקיים עבור k מסוים, אזי הרמה האחרונה לא תהיה מלאה לגמרי. במקרה זה נקבל עץ בינרי כמעט מלא והביטוי $\log_2(n+1)$ לא יהיה מספר שלם. אם מספר הרמות בעץ כזה הוא k, אזי מתקיים הביטוי: $2^k -1 < n < 2^{k+1} - 1$

מכאן נובע כי: $k < \log_2(n+1) < k+1$

אם נבטא את גובה העץ כפונקציה של מספר הצמתים בעץ (כולל המקרה של עץ מלא), נקבל:

$\log_2(n+1)-1 \le k < \log_2(n+1)$

מכאן נוכל להסיק שגובהו של עץ כזה אף הוא לוגריתמי במספר הצמתים.

לעומת זאת, גובהו של עץ שבו n צמתים, המכיל בכל רמה צומת אחד, יהיה n-1.

לסיכום נוכל לנסח מסקנה: פעולת חיפוש ערך בעץ-חיפוש-בינרי עוברת רק על המסלול המתחיל בשורש. פעולת ההכנסה של ערך לעץ-חיפוש-בינרי אף היא עוברת על מסלול בעץ. מספר הצמתים במסלול הוא לכל היותר אחד יותר מגובה העץ. לכן סדר הגודל של פעולות אלה חסום על ידי גובה העץ. גובהו של עץ שבו n צמתים נע בין סדר גודל לוגריתמי $O(\log n)$ בעץ מאוזן ככל האפשר, לסדר גודל לינארי $O(n)$ בעץ שאינו מאוזן כלל, ובו צומת יחיד בכל רמה. מכאן שיעילות פעולות החיפוש וההכנסה בעץ-חיפוש-בינרי המכיל n צמתים נעה בין $O(\log n)$ במקרה הטוב ביותר, ל- $O(n)$ במקרה הגרוע ביותר.

כמובן שאם n מספר קטן, למשל 5 או 10, אין הבדל גדול בין סדרי גודל אלה. אך אם n=1000, ההבדל משמעותי: $2^{10} = 1024$, ולכן $\log(n)$ כאן הוא (כמעט) 10, בעוד n הוא 1000. יש כמובן הבדל גדול אם פעולת חיפוש צריכה לעבור על 10 צמתים או על 1000 צמתים. ההבדל יהיה משמעותי יותר ככל שמספר הצמתים יגדל.

כפי שראינו קודם, ניתן להכניס אותה קבוצת ערכים בסדר שונה, ולקבל עצי-חיפוש-בינריים שונים. המקרה הטוב ביותר הוא כאשר העץ המתקבל מאוזן ככל האפשר, כלומר עץ כמעט מלא, אז יעילות פעולת החיפוש או הכנסה של ערך היא לוגריתמית במספר הצמתים. המקרה הגרוע ביותר הוא כאשר העץ אינו מאוזן כלל, ואז יעילות הפעולות היא לינארית.

סדר גודל לינארי לביצוע פעולות אינו נותן לעץ-חיפוש-בינרי יתרון על מבנים אחרים, כגון רשימה מקושרת. האם ניתן להימנע מהמקרה הגרוע? התשובה לשאלה זו חיובית. ידועות כמה גישות למימוש פעולות הכנסה והוצאה של ערכים לעץ-חיפוש-בינרי ביעילות מסדר גודל לוגריתמי, המבטיחות שהעץ יישאר מאוזן לאחר ביצוע הפעולה. הרעיון המשותף להן הוא שהאלגוריתם המבצע פעולה בודק אם ביצועה מקלקל את האיזון בעץ. אם כן, האלגוריתם משנה את מבנה העץ כדי להחזיר את האיזון. כמובן, אלגוריתמים אלה להכנסה ולחיפוש מסובכים יותר מהאלגוריתמים הפשוטים, אך השימוש בהם מבטיח שהעץ יישאר מאוזן גם אחרי הכנסות והוצאות רבות, ולכן סדר הגודל של פעולות החיפוש, ההכנסה וההוצאה יישאר לוגריתמי (חישוב זה כולל גם את מחיר פעולת האיזון מחדש). אלגוריתמים אלה אינם כלולים ביחידת לימוד זו

(המעוניינים בהרחבה מוזמנים לחפש באינטרנט את המונח: AVL tree. באתרים מסוימים ניתן גם להתרשם מהדגמות של תהליך האיזון של עץ).

ח.6. מיון בעזרת עץ-חיפוש-בינרי

מה יקרה אם נסרוק עץ-חיפוש-בינרי בסדר תוֹכי? כל האיברים בתת-עץ השמאלי של כל צומת בעץ-חיפוש קטנים מהאיבר שבצומת, וכל האיברים בתת-עץ הימני גדולים מהאיבר שבצומת או שווים לו, לכן הסריקה תעבור על כל אחד ואחד מהאיברים בעץ מהקטן ביותר ועד לגדול ביותר בסדר עולה.

אפשר לנצל את הסריקה של עץ-חיפוש-בינרי בסדר תוכי לצורך מיון של רשימה. תחילה נבנה עץ-חיפוש מאיברי הרשימה המקורית, ולאחר מכן נסרוק את עץ-החיפוש שנוצר בסדר תוכי, כך שבכל ביקור בשורש יצורף האיבר שבו לסופה של רשימה. הרשימה המתקבלת תכיל את איברי הרשימה המקורית ממוינים בסדר עולה. מיון כזה נקרא **מיון עץ (tree sort)**.

נחשב את יעילות הפעולה **מיון עץ**. כדי לחשב זאת עלינו להתחיל בחישוב היעילות של בניית עץ-חיפוש מרשימה לא ממוינת. כפי שראינו בסעיף הקודם, הכנסה של איבר בודד לעץ-חיפוש מתבצעת ביעילות מסדר גודל $O(n)$ במקרה הגרוע, ו-$O(\log n)$ במקרה הטוב. כאשר בונים עץ שלם ובו n איברים, יש להכפיל את היעילות במספר זה. לכן יעילותה של פעולת הבנייה של עץ-חיפוש מרשימה לא ממוינת היא במקרה הגרוע מסדר גודל $O(n^2)$ ובמקרה הטוב מסדר גודל $O(n \cdot \log n)$. בשלב הבא מתבצעת סריקת העץ בסדר תוכי, המבטיחה שסדר האיברים המתקבל יהיה ממוין בסדר עולה. במהלך הסריקה מבקר האלגוריתם פעם אחת בכל צומת ומכניס את האיבר שבו לרשימה. האיברים מוכנסים תמיד לסופה של הרשימה החדשה, ולכן סדר הגודל של פעולת ההכנסה הוא $O(1)$. הסריקה כולה היא מסדר גודל $O(n)$, ואינה משפיעה על יעילות המיון כולו.

בסך הכול זמן הריצה של מיון עץ יהיה מסדר גודל ריבועי במקרה הגרוע, ו-$O(n \cdot \log n)$ במקרה הטוב. אם משתמשים באלגוריתמים המבטיחים שהעץ יישאר מאוזן, אזי זמן הריצה יהיה מסדר גודל $O(n \cdot \log n)$ בכל מקרה.

הערה: מיון רשימה באופן המתואר יהיה מעניין במיוחד אם לכל ערך השמור בעץ יוצמד מידע נוסף, למשל אם למספר תעודת הזהות של אדם יוצמדו פרטיו האישיים. בפרק 11 – מפה, נרחיב בנושא זה.

✍ כתבו פעולה המקבלת עץ-חיפוש-בינרי ומדפיסה את איבריו בצורה ממוינת בסדר יורד.

ח.7. ייצוג אוספים בעזרת עץ-חיפוש-בינרי

עץ-חיפוש-בינרי הוא מבנה יעיל מאוד כאשר יש צורך בחיפוש ובמיון איברים שביניהם מתקיים יחס סדר. בדומה לרשימה מקושרת ולעץ בינרי, עץ-חיפוש-בינרי אינו טיפוס נתונים מופשט, אלא מבנה נתונים. הסכנה בשימוש ישיר במבני נתונים היא שקל לקלקל אותם אם מבצעים פעולות

שינוי מבנה באופן לא זהיר. בתכנות מונחה עצמים מתמודדים עם סכנה זו על ידי אריזת מבני נתונים ופעולותיהם במחלקות מתאימות, המונעות גישה ישירה למבנים. לכן, כאשר נרצה להשתמש בעץ-החיפוש-הבינרי לייצוג אוספים, נגדיר אותו כתכונה במחלקה עוטפת.

בתרגילים המצורפים לפרק תתבקשו לממש מחדש מחלקות שאתם מכירים מפרקים קודמים: IntSortedCollection ו-StudentList, אך הפעם הייצוג והמימוש יהיו באמצעות עץ-חיפוש-בינרי.

ט. מבני נתונים לעומת טיפוסי נתונים מופשטים

עם סיום פרק זה, נשוב לדון במונחים "מבנה נתונים" ו"טיפוס נתונים מופשט". כעת יש בידינו די דוגמאות כדי להבהיר את המשמעות של כל אחד מהם, ולחדד את ההבדל ביניהם.

הרעיון העיקרי המשמש אותנו בבניית מבני נתונים ביחידה זו הוא השימוש בחוליה, שהיא עצם עם תכונה אחת המכילה מידע, ותכונות נוספות המכילות הפניות לחוליות נוספות מאותו הטיפוס. על ידי "חיבור" חוליות כאלה זו לזו אנו בונים מבנים משורשרים שונים, שהם מבני נתונים.

בפרק 7 – ייצוג אוספים ראינו שהמחלקה Node מגדירה חוליות עם הפניה יחידה. על ידי שרשור חוליות כאלה ניתן לבנות מבנים לינאריים של חוליות, אבל גם מבנים אחרים, כגון מעגל, כמה רשימות או כמה מעגלים, ועוד. כל אלה הם מבני נתונים. בפרק הנוכחי, המחלקה BinNode מגדירה חוליות בינריות, שמהן ניתן לבנות עצי חוליות בינריים (כלומר מבני חוליות היררכיים המייצגים עצים בינריים), אך גם מגוון מבנים אחרים. עצי חוליות בינריים עם הגבלות מסוימות מייצגים עצי חיפוש בינריים. גם אלה הם מבני נתונים.

אחת התכונות המשותפות למבנים אלה היא שהם בעלי חוליה אחת לפחות, אחרת הם אינם קיימים. עובדה זו נובעת מכך שרשימה מקושרת ועץ חוליות אינם עצם, אלא מבנה המורכב מקבוצת עצמים המשורשרים זה לזה. התכונה השנייה המשותפת להם היא שניתן להגדיל אותם, כמעט ללא גבול, על ידי הוספת עוד ועוד חוליות, וכן אפשר גם להקטינם על ידי הוצאת חוליות (רשימה מקושרת גמישה יותר מעץ חוליות בינרי לגבי פעולות הוספה והוצאה של חוליות, אך הבדל זה אינו חשוב לדיון הנוכחי). התכונה השלישית המשותפת להם היא שאם מאפשרים פעילות ישירה עליהם קיימת סכנה שהמבנה שלהם יתקלקל. למשל, רשימה מקושרת יכולה להפוך למעגל סגור, ועץ בינרי יכול להפוך למבנה שאינו עץ.

גם את מבני הנתונים האלה הוספנו לארגז הכלים שהלך ונבנה לאורך היחידה. בארגז נאספו אם כן המחלקות: Node ו- BinNode; מבני הנתונים: רשימה מקושרת, עץ חוליות בינרי ועץ-חיפוש-בינרי, הנבנים על ידי שרשור של חוליות אונריות או בינריות; וטיפוסי הנתונים: מחסנית, תור ורשימה. כל אלה יוכלו לשמש אותנו לפי הצורך לכתיבת תוכנה לניהול סוגים שונים של אוספים כלליים שימושיים. השימוש במבני הנתונים כמבנים פנימיים בייצוג אוסף, הוא בעל יתרון חשוב: ניתן לנצל את הגמישות של מבני נתונים אלה, תוך הימנעות מהסכנה של קלקול מבנה על ידי מימוש לא זהיר של פעולות.

במחלקות Stack ו-Queue הצלחנו להשיג את היתרון הזה בשלמותו – מחלקות אלה מסתירות לחלוטין את הייצוג הפנימי שלהן ואת מימושי הפעולות. הן מממשות טיפוסי נתונים מופשטים.

לעומת זאת, עץ חוליות בינרי ועץ-חיפוש-בינרי הם מבני נתונים. בגלל איזה הבדל בין סוגי האוספים הצלחנו בהסתרת המימוש ברוב האוספים, אך למשל לא ברשימות.

במחסנית ובתור הפעולות עוסקות בערכים בלבד, והידע של המשתמש מוגבל לקשרים בין הפעולות. במחסנית המשתמש יודע רק שפעולת הוצאה תחזיר תמיד את הערך שהוכנס אחרון, ובתור – שפעולת הוצאה תחזיר את הערך הוותיק ביותר. אין במידע זה כל התייחסות לארגון פנימי של האוסף. כמו כן, משתמש אינו צריך או אינו יכול לקבוע את מקומו של ערך המוכנס למחסנית או לתור, שכן אלה נקבעים על ידי הפרוטוקול, כך שהקשר בין הכנסות והוצאות ממשיך להתקיים. לכן, ניתן לייצג אוסף מסוגים אלה בדרכים שונות, וכאשר מוסיפים ערך לאוסף, מקומו נקבע על ידי מצב הייצוג של האוסף. למשתמש המבצע את פעולת ההכנסה אין כל השפעה או ידע על ייצוג האוסף ועל מקומו של הערך החדש.

נסיים דיון זה בכמה הערות לגבי מימוש מחלקות הממששות סוגי אוספים כאלה בג׳אווה. מקובל שמחלקה חייבת **בהסתרה** מלאה. עץ-החיפוש-הבינרי לא נמצא כמחלקה בשפה, אך הרעיון עצמו ממומש במחלקות אחרות. גישת השפה משתמשת ברעיונות שאינם כלולים ביחידת לימוד זו ומאפשרים הסתרה מלאה.

י. סיכום

- **עץ בינרי** הוא עץ שיש לכל צומת בו שני ילדים לכל היותר. מקובל להתייחס אליהם כילד שמאלי וילד ימני. כל אחד משני אלה, אם הוא קיים, הוא שורש של עץ.
- העץ הבינרי הקטן ביותר נקרא **עץ עלה**.
- הגדרה רקורסיבית ל**עץ חוליות בינרי**:
    - חוליה בינרית יחידה;

    או
    - חוליה בינרית שבה לכל היותר שתי הפניות ל**עצי חוליות בינריים** הזרים זה לזה.
- מבנהו של העץ הבינרי מאפשר טיפול נוח באמצעות אלגוריתמים רקורסיביים.
- קיימים שני סוגים עיקריים של אלגוריתמים לסריקה של עץ בינרי: אלגוריתמים הסורקים את העץ לעומק ואלגוריתמים לסריקה לפי רמות (לרוחב). יעילותם של אלגוריתמים אלה לינארית במספר הצמתים.
- עץ חוליות בינרי הוא מבנה נתונים היררכי המורכב מחוליות בינריות. עץ החוליות אינו טיפוס נתונים מופשט, ולכן הוא אינו עטוף במחלקה.
- גובהו של עץ שבו n צמתים נע בין $\log n$ (בעץ מאוזן ככל האפשר) לבין n (בעץ שאינו מאוזן כלל, ובו צומת יחיד בכל רמה).
- עץ-חיפוש-בינרי הוא עץ שבו כל הערכים המאוחסנים בתת-עץ שמאלי של צומת כלשהו קטנים מהערך בצומת, וכל הערכים המאוחסנים בתת-עץ ימני של צומת גדולים או שווים מהערך בצומת.

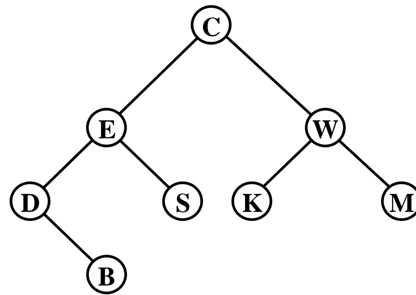
מושגים

sibling	אח
tree height	גובה עץ
parent	הורה
ancestor	הורה קדמון
BinNode	חוליה בינרית
child (left , right)	ילד (שמאלי, ימני)
tree sort	מיון עץ
postorder traversal	סריקה בסדר סופי
inorder traversal	סריקה בסדר תוֹכי
preorder traversal	סריקה בסדר תחילי
level traversal	סריקה לפי רמות
leaf	עלה
tree	עץ
binary tree	עץ בינרי
full binary tree	עץ בינרי מלא
binary search tree	עץ-חיפוש-בינרי
descendant	צאצא
level	רמה
root	שורש
subtree (left , right)	תת-עץ (שמאלי, ימני)

תרגילים

שאלה 1

התייחסו לעץ הבינרי הזה וענו על השאלות:



א. השלימו:

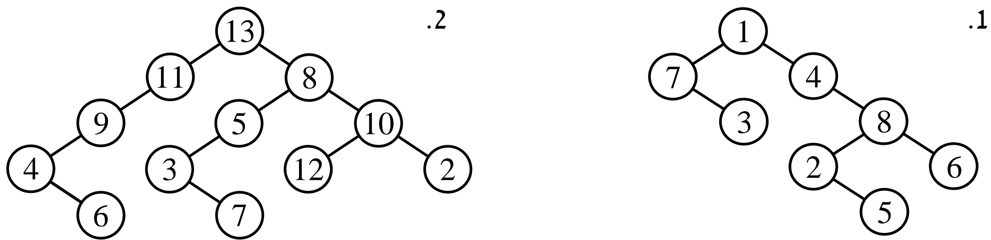
- מספר הצמתים בעץ הוא __________.
- מספר העלים בעץ הוא __________.
- השורש של התת-עץ השמאלי של E הוא __________.
- W הוא __________של C.
- C הוא __________של B.
- הגובה של העץ הוא __________.
- הצאצאים של E הם __________.
- הצמתים ברמה 2 בעץ הם __________.

ב. עבור כל אחד מהמשפטים ציינו אם הוא נכון או לא נכון, ונמקו:

- B הוא צאצא של E, C ו-W
- S ו-K הם אחים
- ל-E יש שני ילדים
- K הוא ברמה 3
- אם הצמתים נמצאים באותה רמה אזי הם אחים
- C הוא הורה קדמון של E ו-B

שאלה 2

לפניכם שני עצים בינריים שבצומתיהם מספרים:



כתבו מה יתקבל לאחר שתסרקו כל אחד מהעצים בסריקה בסדר תחילי, בסדר תוֹכי, בסדר סופי ובסריקה לפי רמות.

שאלה 3

א. נתונה סדרת ערכים שהתקבלו מסריקה כלשהי של עץ. ייתכנו כמה עצים המתאימים לה. ציירו שני עצים **שונים**, ובהם ערכים, כך שאם נסרוק את העצים בסדר תחילי נקבל את הסדרה (משמאל לימין): 3, 7, 9, 5.

ב. כאשר נתונים ערכים שהתקבלו משתי סריקות: בסדר תחילי ובסדר סופי, ייתכנו כמה עצים המתאימים לשתי הסריקות. ציירו שני עצים **שונים**, ובהם ערכים, כך שאם נסרוק אותם בסדר תחילי נקבל 3, 7, 9, 5; ואם נסרוק אותם בסדר סופי נקבל: 9, 5, 7, 3 (הסדרות נקראות תמיד משמאל לימין).

ג. כאשר נתונים ערכים שהתקבלו משתי סריקות, ואחת מהן היא סריקה בסדר תוֹכי, קיים **רק עץ אחד** המתאים לשתי הסריקות (**בתנאי** שהמספרים בכל סריקה שונים לחלוטין זה מזה). כדי להבין איך ניתן לשחזר אותו צריך להבין איזה מידע ניתן להפיק משתי הסריקות.

- בכל סריקה בסדר תחילי נתונה, שורש העץ הוא ________________________.
- בכל סריקה בסדר סופי, שורש העץ הוא ______________________________.
- נתונה סריקה בסדר תוֹכי. אם ידוע כי שורש העץ הוא מספר העומד במקום מסוים, מה ניתן להגיד על כל המספרים שלפני המקום הזה, ועל כל המספרים המופיעים אחרי מקום זה בסריקה? __________________________________.

ציירו את העץ היחיד האפשרי, שאם נסרוק אותו בסדר תחילי נקבל: 1, 5, 7, 3, 8, 4, 6; ואם נסרוק אותו בסדר תוכי נקבל: 7, 5, 8, 3, 1, 4, 6.

שאלה 4

הפעולה שלפניכם מקבלת עץ חוליות בינרי שערכיו מספרים שלמים:

```
public static int mystery(BinNode<Integer> bt)
{
      if (bt == null)
            return 0;

      if (bt.getLeft() == null && bt.getRight() == null)
            return 1;

      return mystery(bt.getLeft()) + mystery(bt.getRight());
}
```

א. כתבו את טענת היציאה של הפעולה.

ב. **טענה**: אם היינו מחליפים את טיפוס ערכי הצמתים מ-Integer לכל טיפוס אחר, הדבר לא היה משפיע על ביצוע הפעולה, והיא הייתה מבצעת בדיוק אותה המשימה.

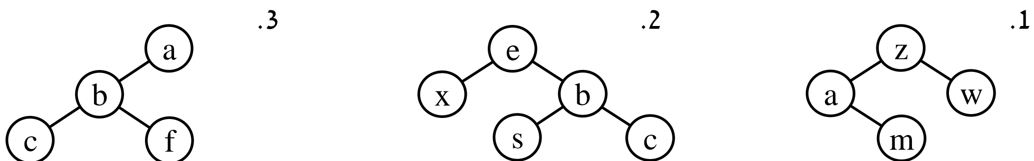
האם טענה זו נכונה? נמקו.

שאלה 5

לפניכם הפעולה:

```
public static boolean secret(BinNode<Character> bt)
{
      if (bt.getLeft() == null && bt.getRight() == null)
            return true;

      if (bt.getLeft() == null || bt.getRight() == null)
            return false;

      return secret(bt.getLeft()) && secret(bt.getRight());
}
```

א. מה תחזיר הפעולה secret(...) אם נזמן אותה עבור כל אחד מהעצים:



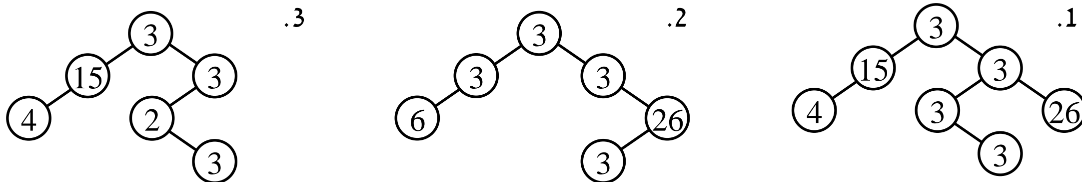
ב. כתבו את טענת היציאה של הפעולה.

שאלה 6

לפניכם הפעולה:

```
public static boolean what(BinNode<Integer> bt, int x)
{
      if (bt == null)
            return false;

      if (bt.getValue() !=  x)
            return false;

      if (bt.getLeft() == null  && bt.getRight() == null)
            return true;

      return what(bt.getLeft(), x) || what(bt.getRight (), x);
}
```

א. איזה ערך יוחזר כשנזמן את הפעולה כך: what (bt, 3) עבור כל אחד מהעצים:



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

ג. מה תהיה טענת היציאה של הפעולה (...)what אם נחליף את האופרטור or (||) באופרטור and (&&) בשורה האחרונה.

שאלה 7

לפניכם פעולה שמקבלת מספר שלם גדול או שווה לאפס, ומחזירה עץ חוליות בינרי של מספרים שלמים:

```
public static BinNode<Integer> build(int n)
{
      if (n == 0)
                  return new BinNode<Integer>(0);

      return new BinNode<Integer>(build(n-1), n, build(n-1));
}
```

א. ציירו את העץ הבינרי שיוחזר עבור הזימון: build(3).

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

שאלה 8

לפניכם הפעולה:

```
public static int sod(BinNode<Integer> bt)
{
      int ml = bt.getValue();
      int mr = bt.getValue();

      if (bt.getLeft() != null)
            ml = Math.max(bt.getValue(), sod(bt.getLeft()));

      if (bt.getRight() != null)
            mr = Math.max(bt.getValue(), sod(bt.getRight()));

      return Math.max(ml, mr);
}
```

כתבו את טענת היציאה של הפעולה.

שאלה 9

נתונה הפעולה הזו:

```
public static void mystery(BinNode<Integer> bt)
{
      BinNode<Integer> node = bt;
   Stack<BinNode<Integer>> stack = new Stack<BinNode<Integer>>();

      do
      {
            while (node != null)
            {
                  stack.push(node);
                  node = node.getLeft();
            }
            if (!stack.isEmpty())
            {
                  node = stack.pop();
                  System.out.print(node.getValue() + " ");
                  node = node.getRight();
            }
   } while (!stack.isEmpty() || node != null);
}
```

מהי טענת היציאה של הפעולה? הדגימו בעזרת עץ של מספרים שלמים.

שאלה 10

ממשו את הפעולה:

```
public static Node<Integer> levelOrderList(BinNode<Integer> bt)
```

הפעולה תחזיר רשימה מקושרת ובה כל הערכים השמורים בעץ, מופיעים על פי סדר הסריקה לפי רמות.

שאלה 11

ממשו את הפעולה:

```
public static int count(BinNode<Character> bt, char ch)
```

הפעולה תחזיר את מספר המופעים של התו ch בצומתי העץ bt.

שאלה 12

ממשו את הפעולה:

```
public static void printStrings(BinNode<String> bt, char ch)
```

הפעולה תדפיס, בשורות נפרדות, את כל המחרוזות בצומתי העץ bt, שבהן מופיע התו ch.

שאלה 13

ממשו את הפעולה:

```
public static void replace(BinNode<String> bt,
                                    String s1, String s2)
```

הפעולה תחליף כל מחרוזת שמופיעה בצומתי העץ bt וערכה s1, במחרוזת s2.

שאלה 14

ממשו את הפעולה:

```
public static int height(BinNode<Integer> bt)
```

הפעולה תחזיר את גובהו של העץ הבינרי bt.

שאלה 15

ממשו את הפעולה:

```
public static int numNodesInLevel(BinNode<Integer> bt, int level)
```

הפעולה תחזיר את מספר הצמתים ברמה level בעץ הבינרי bt.

הנחה: level>=0.

שאלה 16

ממשו את הפעולה:

```
public static BinNode<String> buildIdent(BinNode<String> bt)
```

הפעולה תבנה **עץ בינרי זהה** (במבנה ובתוכן) לעץ bt המתקבל כפרמטר, ותחזיר אותו.

שאלה 17

ממשו את הפעולה:

```
public static boolean isFull(BinNode<Integer> bt)
```

הפעולה תחזיר 'אמת' אם העץ הבינרי bt הוא עץ מלא. אחרת, תחזיר 'שקר'.

שאלה 18

ממשו את הפעולה:

```
public static BinNode<Integer> parent(BinNode<Integer> bt,
                                      BinNode<Integer> child)
```

הפעולה תחזיר את ההורה של child, שהוא אחד מצמתיו של העץ bt, או **null** אם child הוא שורש העץ.

שאלה 19

ממשו את הפעולה:

```
public static BinNode<Integer> buildRandomTree(int maxLevels)
```

הפעולה תחזיר עץ בינרי שבו לכל היותר maxLevels רמות. מבנהו המדויק של העץ ותוכן הצמתים שלו אקראיים.

הערה: חשבו תחילה כיצד תוכלו לקבוע האם לצומת יהיו שני בנים, בן בודד (ימני או שמאלי) או אף לא בן אחד.

שאלה 20

ממשו את הפעולה:

```
public static BinNode<Integer> buildTree(int[] preorder,
                                         int[] inorder)
```

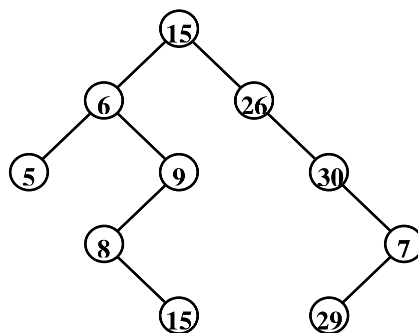
הפעולה מקבלת שני מערכים של שלמים שהתקבלו כתוצאה מסריקת עץ בינרי בסריקה בסדר תחילי ובסריקה בסדר תוֹכי, ומחזירה את העץ המקורי.

הנחה: המספרים בכל מערך שונים זה מזה.

שאלה 21

ילד יחיד בעץ בינרי הוא צומת שלהורה שלו אין ילד נוסף. נגדיר **הפרש ילדים** בעץ כהפרש בין סכום כל הילדים היחידים הימניים בעץ של שלמים, לבין סכום כל הילדים היחידים השמאליים בעץ (נפחית את סכום השמאליים מסכום הימניים).

א. מהו **הפרש ילדים** בעץ הבינרי הזה:



ב. ממשו פעולה המקבלת עץ בינרי של שלמים ומחזירה את **הפרש הילדים** בו.

שאלה 22

עצים דומים הם עצים שהמבנה הפנימי שלהם, כלומר סדר הצמתים בתוך כל עץ, זהה. לדוגמה, העצים שלפניכם הם עצים דומים, אף שהם מכילים ערכים שונים:

ממשו פעולה המקבלת שני עצים בינריים של שלמים. הפעולה תחזיר 'אמת' אם העצים דומים, ו-'שקר' אחרת.

שאלה 23

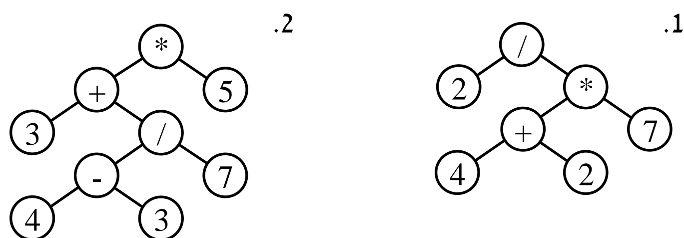
בסעיף ז בפרק ראינו שימוש בעץ חוליות בינרי לייצוג של ביטוי חשבוני.

א. ציירו את העצים הבינריים המתאימים לביטויים החשבוניים:

- ((2 - (6 / 2)) * (4 * 3))
- (3 + (6 * ((2 / 4) - 5)))

כמו כן, הראינו בסעיף ז בפרק את האלגוריתם **חשב-ערך-ביטוי** המקבל עץ ביטוי, מחשב את ערך הביטוי ומחזיר אותו.

ב. העזרו באלגוריתם **חשב-ערך-ביטוי** וכתבו מהו ערכו של הביטוי החשבוני המיוצג על ידי כל אחד מהעצים:



ג. ממשו פעולה שמחזירה עץ בינרי המייצג ביטוי חשבוני. העץ ייבנה על פי קלט של סדרת תווים המהווה ביטוי חשבוני תקין (הביטוי החשבוני ימוסגר לחלוטין באופן תקין, יורכב מספרות בלבד ולא יכיל ביטויים שליליים מהצורה x–).

שאלה 24

בסעיף ח בפרק ראינו את ההגדרה של **עץ-חיפוש-בינרי**:

עץ חוליות בינרי שערכיו מסודרים כך שערך כל צומת בעץ גדול מכל אחד מצאצאיו בַּתת-עץ השמאלי וקטן או שווה לכל אחד מצאצאיו בתת-עץ הימני.

א. בנו עצי-חיפוש-בינריים על פי הסדרות (קראו משמאל לימין):

1. 5, 15, 7, 10, 14, 12, 11
2. G, D, J, E, A, C, F
3. 13, 10, 4, 50, 59, 10, 2, 3, 59, 35, 55, 35

ב. מצאו סדרת מספרים נוספת שתיצור עץ-חיפוש זהה לזה שנוצר על ידי הסדרה ה-3.

ג. ממשו פעולה המחזירה את הערך הגדול ביותר בעץ-חיפוש-בינרי.

ד. ממשו פעולה המקבלת אוסף של מספרים שלמים ומחזירה את אוסף המספרים ממוינים ללא כפילויות (חזרו וקראו את סעיף ח.5. בפרק).

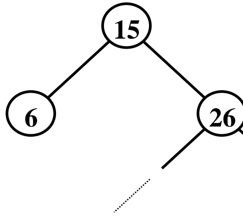
ה. נרחיב את הפעולה הקודמת: ממשו פעולה המקבלת אוסף של מספרים שלמים, כך שהאוסף המוחזר יכיל את כל המספרים ממוינים, ללא כפילויות, תוך ציון מדויק של כמה פעמים הופיע כל ערך באוסף המקורי.

הנחיה: הערכים שיוחזרו יהיו זוגות. בכל זוג, הערך הראשון יהיה מספר שלם, והערך השני בו יהיה מספר המופעים של המספר באוסף המקורי.

שאלה 25

עץ-בינרי-משופע הוא עץ שבו הבן הימני של כל צומת גדול מערך אביו, ובנו השמאלי של כל צומת קטן מערך אביו.

א. לפניכם עץ-בינרי-משופע (הצומת 6 הוא עלה, התת-עצים ששורשם 26 אינם מצוירים).



1. האם ניתן להסיק ממבנה העץ, שהמספר 13 אינו נמצא בו? נמקו.
2. מה הייתה התשובה אילו הנחנו שהעץ הוא עץ-חיפוש-בינרי?

ב. נסחו במדויק מהו ההבדל בין עץ-בינרי-משופע לעץ-חיפוש-בינרי.

ג. כתבו פעולה שמקבלת עץ בינרי כלשהו ומחזירה 'אמת' אם הוא עץ-בינרי-משופע, ו-'שקר' אחרת.

שאלה 26

א. חזרו לפרק רשימה, לסעיף ז.1., וכתבו מחדש את המחלקה StudentList. השתמשו בעץ-חיפוש-בינרי לייצוג אוסף התלמידים.

ב. נתחו את יעילות הפעולות של המחלקה בייצוג החדש בהשוואה ליעילותן בפרק רשימה.

שאלה 27

א. חזרו לשאלה 19 בפרק רשימה, וכתבו מחדש את המחלקה IntSortedCollection. השתמשו בעץ-חיפוש-בינרי לייצוג אוסף המספרים הממוין.

ב. נתחו את יעילות הפעולות של המחלקה בייצוג החדש בהשוואה ליעילותן בפרק רשימה.

שאלות מבחן מותאמות לתכנית הלימודים החדשה

<u>שאלה 1</u> (ברכה דאום רייטר)

מדריך (Madrich) הוא טיפוס שתכונותיו הם שם, מין ושם השבט.

- בנה מחלקה לטיפוס Madrich הכוללת תכונות, פעולה בונה, פעולות מאחזרות וקובעות(get,set) ופעולת toString.
- כתוב תכנית הבונה מערך חד מימדי של עצמים באורך 20 של "מדריכים" שנתוניהם התקבלו כקלט. יש להציג כפלט:
  - א. את שם השבט בו הכי הרבה מדריכים(בנים ובנות),
  - ב. את שם השבט בו הכי הרבה מדריכות.

<u>שאלה 2</u> (ברכה דאום רייטר)

Panuy הוא טיפוס שתכונותיו הן גיל, מין וגובה.

- בנה מחלקה לטיפוס Panuy הכוללת תכונות, פעולה בונה, פעולות מאחזרות (get) ופעולת toString.

במשרד שידוכים שומרים במערך בעל 20 מקומות את נתוני ה"פנויים" שבמשרד.

כתוב תכנית הקולטת נתוני Panuy חדש ומאתרת עבורו את השידוך הראשון המתאים. על השידוך להיות מהמין השני, בגיל דומה (הפרש עד שנתיים לכל כיוון) ובגובה מתאים (על הגבר להיות גבוה יותר מהאישה).

אם נמצא מועמד מתאים על התכנית להוציא כפלט את פרטיו. באם אין מועמד מתאים על התכנית להוציא הודעה מתאימה.

<u>שאלה 3</u> (נטליה קיפניס)

נתונה המחלקה תאריך שלי - MyDate שיש לה 3 תכונות:

יום - day מטיפוס שלם

חודש - month מטיפוס שלם

שנה - year מטיפוס שלם

הנה שלכל תכונה הוגדרו ב- Java - פעולות get ו-set וב-C# פעולת Get ו- Set.

א. כתוב ב-Java או C# פעולה בונה של המחלקה MyDate , שתקבל כפרמטרים ערכים לכל אחת משלוש התכונות.

ב. נתונה המחלקה חייל- Soldier שיש לה 4 תכונות :

num - מספרו האישי של חייל מטיפוס שלם

born - תאריך לידה מטיפוס MyDate

gius - תאריך גיוס מטיפוס MyDate

endSadir – תאריך שחרור מטיפוס MyDate

הנח שלכל תכונה הוגדרו ב- Java - פעולות get ו-set וב-C# פעולות Get ו- Set.

1) כתוב ב-Java או C# את כותרת המחלקה Soldier ואת תכונות שלה.

2) כתוב ב-Java או C# פעולה במחלקה Soldier שתקבל תאריך מטיפוס MyDate ומחזירה true אם החייל כבר משוחרר ו– false במקרה אחרת.

3) כתוב ב-Java או C# במחלקה Main פעולה חיצונית בשם updateSoldier שתקבל מערך חד ממדי של גדוד כלשהו. כל תא בו מטיפוס Soldier וגם תאריך מטיפוס MyDate.

הפעולה תדפיס את שמות החיילים שצריכים להשתחרר מהצבא לפי התאריך, תמחק את החיילים האלו מהגדוד ותחזיר את המספר החיילים האלו.

שאלה 4: (גלית שריקי)

לפניך המחלקה **Recycling**

מחלקה זו מכילה את קבוצת המחזור ואת כמות הסוללות והפחיות בסל המחזור.

Recycling()	פעולה בונה המאתחלת את מספר הסוללות ואת מספר הפחיות ל-0
void addBattery(int num)	פעולה המוסיפה num סוללות
void addCans(int num)	פעולה המוסיפה num פחיות
int getBattery()	פעולה המחזירה את מספר הסוללות
int getCans()	פעולה המחזירה את מספר הפחיות
int getPoints()	פעולה המחזירה את מספר הנקודות שמקבלים על הפחיות ועל הסוללות יחד. פחית מזכה ב-2 נקודות וסוללה מזכה ב-6 נקודות

א. הוסיפו פעולה בונה מעתיקה למחלקה **Recycling** .

לפניך המחלקה **RecycWin**

מחלקה זו מדמה תחרות בין מספר קבוצות למחזור(כל קבוצה מסוג **Recycling)**

מספר הקבוצות המקסימלי שיכול להשתתף בתחרות הוא 10.

כמו כן מחלקה זו שמורת נתונים על כמות הסוללות והפחיות הכללי שמוחזר.

RecycWin()	פעולה הבונה תחרות חדשה ללא קבוצות מחזור.
void addTeam(int num)	פעולה המוסיפה קבוצה לתחרות, מספר הקבוצה הוא num
void addBattery(int num, int batteryNum)	פעולה המוסיפה batteryNum סוללות לקבוצה שמספרה num הנחה: קיימת קבוצה מספר num
void addCans(int num, int cansNum)	פעולה המוסיפה cansNum פחיות לקבוצה שמספרה num

הנחה: קיימת קבוצה מספר num	
פעולה המחזירה את מספר הסוללות הכללי	static int getBattery()
פעולה המחזירה את מספר הפחיות הכללי	static int getCans()
פעולה המוסיפה num למספר הסוללות הכללי	static void setBattery(int num)
פעולה המוסיפה num למספר הפחיות הכללי	static void setCans(int num)
פעולה המחזירה את מספר הנקודות לקבוצה שמספרה num הנחה קיימת קבוצה מספר num	int getPoint(int num)
פעולה המחזירה את הקבוצה המנצחת בתחרות הנחה: קיימת קבוצה אחת שניצחה	Recycling win()

ב.1 . ייצגו את המחלקה RecycWin

2. ממשו את הפעולות addCans, setBattery , getPoints ו- win.

ג. בבית ספר קיימים 5 כיתות בשכבת י' ובכל כיתה מספר התלמידים שונה.
הוחלט לקיים תחרות בשיתוף המשרד לאיכות הסביבה.
במסגרת התחרות מביאים התלמידים מוצרים למחזור: סוללות או פחיות. פחית מזכה ב-2 נקודות וסוללה מזכה ב-6 נקודות.
כתוב תכנית הקולטת עבור כל תלמיד בכל כיתה זוג מספרים כאשר המספר הראשון מייצג את מספר הפחיות שהתלמיד הביא והמספר השני מייצג את מספר הסוללות שהתלמיד הביא. סיום הקלט יהיה הזוג 0,0.
התכנית תדפיס את מספר הכיתה המנצחת ואת הניקוד שלה, וכן כמה כתות צברו לפחות 450 נקודות, כמו כן תדפיס התכנית את כמות הפחיות והסוללות הכללי שמוחזר.

<u>שאלה 5</u> (ישראל אברמוביץ)
בחנות פרחים יש פרחים בצבעים: red, Blue, Pink
בכל בוקר בעל החנות מעדכן כמה פרחים הוא קיבל מכל צבע, לכל צבע של פרח יש עצם אשר מכיל את כמות הפרחים שיש בחנות באותו זמן.
כאשר המוכר בחנות מוכר פרחים מצבע מסוים הוא מעדכן את העצם המתאים.

א. כתבו מחלקה בשם Flower

למחלקה יש תכונה מסוג int ששמה count אשר שומרת את כמות הפרחים
הפעולה הבונה מקבלת את כמות הפרחים שהמוכר קיבל בבוקר
הוסיפו פעולה אשר מקבלת את מספר הפרחים שנמכרו לקונה ומעדכנת כמה פרחים נשארו.
הוסיפו פעולה שמאחזרת את כמות הפרחים שנשארו.

הפעולה ()ToString מחזירה מחרוזת עם כמות הפרחים שנשארו.

ב. כתבו בתוכנית הראשית קטע קוד שקולט בתחילת היום כמה פרחים המוכר קיבל מכל צבע (red, Blue, Pink),

לאחר מיכן מגיעים הלקוחות, כל לקוח מעדכן את צבע הפרח המבוקש (red / Blue / Pink) וכמות הפרחים שהלקוח קנה מאותו צבע, אם הלקוח ביקש פרחים מצבע מסוים בכמות יותר גדולה ממה שנשאר, הלקוח לא יקבל פרחים ותודפס הודעה מתאימה.

החנות נסגרת כאשר נגמרו הפרחים מצבע מסוים.

בסוף היום המוכר מדפיס כמה פרחים נשארו מכל צבע

- יש ליצור עצמים מסוג Flower ולהשתמש בכל הפעולות הפנימיות

שאלה 6: (זהבית בר לוי)

נתונה המחלקה **Date** – תאריך.

לפניך פעולה ראשית :

```
public static void main(String[] args)
{
        Date d1 = new Date(1,1,2007);
        Date d2 = new Date(2,2,2006);
        Date [] d3 = new Date[5] ;
        d3[d3.length-1]= d1;
        d3[d3.length-2]=d2 ;
        //→  1
        //→  2  System.out.println(d3[0].getDay());
        for(int k =0 ; k<d3.length ;k++)
        {
            System.out.println(d3[k]);
        }
        //→3
      Date d4 = new Date(1,1,2007);
      Date d5=d4;
      d4.setDay(20);
      d5.setYear(1980);
      System.out.println(d4);
      System.out.println(d5);
      //→4
```

}

א. תאר בצורה סכימטית את העצמים השונים ואת כל ההפניות אל העצמים, עד השורה המסומנת ב **1 →**.

ב. ההוראה בשורה המסומנת ב - **2 →** שגויה.

האם זוהי שגיאת הידור (קומפילציה) או שגיאת ריצה , נמקו בקצרה .

ג. מה הפלט המתקבל עד השורה המסומנת ב **3 →** (מתקבל פלט תקין) .

ד. מה יודפס בקטע התוכנית בין הסימנים **3→** ו- **4→**?

ה. כמה עצמים מסוג **Date** נוצרו בין שני הסימנים הנ"ל? הסבר בקצרה.

ו. מה ניתן להוסיף במחלקה **Date** כדי שתהיה מנייה אוטומטית של כל עצם חדש שנוצר? איזה שינויים צריכים לערוך לשם כך ואיפה?

שאלה 7 (גלית סלוצקי)

באולימפיאדה יש תחרות "מינימרתון" שבה כל משתתף במתחרה בכמה מקצועות ספורט. גברים מתחרים ב- 10 מקצועות, ונשים מתחרות ב- 7 מקצועות ספורט.

כל משתתף מקבל ציון בכל אחד מהמקצועות, וציונו הסופי הוא ממוצע הציונים שקיבל.

המחלקה MiniMaratonTicket מכילה את התכונות:

1. genader - טיפוס מחרוזת
2. totalGrade - טיפוס ממשי
3. name – טיפוס מחרוזת
4. numOfSports – טיפוס שלם

ממש במחלקה את הפעולה הבונה, כאשר הקלט הוא:

- שם המשתתף בתחרות
- מין המשתתף (גבר/אישה)
- בהתאם למין קלוט את הציון בכל מקצוע ספורט וחשב את ציוו הסופי.

הקפד לעדכן את כל אחד מ- 4 התכונות של המחלקה.

שאלה 8 (גלית סלוצקי)

סוכנות הנסיעות "מסעות" מציעה טיול לחופשת הפסח.

לטיול יכולים להירשם עד 50 נוסעים מעל גיל 18 (כולל).

כשנוסע מבקש להירשם לטיול נבדק גילו והתוקף של דרכונו.

נוסע יכול להירשם אם דרכונו תקף עד 1/10/2010 ויש מקום פנוי בטיול שהוא רוצה.

עבור כל לקוח יש לקלוט: גיל ואם הגיל מתאים ויש מקום בטיול

לקלוט חודש, שנה של דרכון.

אם הדרכון בתוקף אז לרשום לטיול.

התוכנית תעצור כאשר אין מקומות בטיול או שנקלט במשתנה גיל המספר 999.

השתמש במחלקה Date כדי לבדוק את תוקף הדרכון.

המחלקה PassportDate מכילה את התכונות:

יום – סוג מספר שלם

חודש – סוג מספר שלם

שנה – סוג מספר שלם

למחלקה PassportDate קיימת פעולה SetDate :

```
void  SetDate(int day, int month, int year);
```

ממש במחלקה זו את הפעולה

```
bool  validDate(int day, int month, int year);
```

המקבלת את התאריך הרצוי ובודקת את הדרכון בתוקף עד תאריך זה.

שלב את המחלקה בתוכנית הראשית המתוארת.

<u>שאלה 9</u> (דורית בן דוד)

המחלקה Collec היא אוסף (רשימה) של מספרים שלמים וגדולים מ־ 0 .

כתוב את כותרת המחלקה Collec ואת התכונות שלה.

ממש במחלקה את הפעולה MaxRetzef המחפש את הרצף הארוך ביותר הקיים ברשימה ומחזיר את האיבר הראשון של הרצף הזה.

אם רוצים להעלות רמה בשאלה אז.....

המחלקה Collec היא אוסף (רשימה) של תלמידים.

לכל תלמיד יש אוסף של ציונים.

כתוב את כותרת המחלקה Student ואת התכונות שלה.

כתוב את כותרת המחלקה Collec ואת התכונות שלה.

במחלקת Student ממש את הפעולה AvgGrade המחשב את ממוצע ציוני התלמיד ומחזירה A אם הציון הממוצע הוא 90 עד 100, B אם הוא בין 80 ל 90 C אם הוא בין 70 ל 80 D בין 60 ל 70 ו F אם פחות מ60.

ממש במחלקה את הפעולה MaxRetzef המחפש את הרצף הארוך ביותר של דרגת ציונים הקיים ברשימה ומחזיר את האיבר הראשון של הרצף הזה.

<u>שאלה 10</u> (אביטל Evi גרינולד)

נתונות המחלקות הבאות : Moo, Goo, TestGoo

public class Moo	public class Goo

``java
public class Moo
{
  private int n;
  private int ch;
 	
 public Moo(int n, char ch)
  {
    this.n = n;
    this.ch = ch;
  }
 	
  public int getN()
  {   return this.n;   }
 	
  public int getCh()
  {   return this.ch; }
 	
  public void setN(int n)
  {   this.n = n;   }
 	
  public void setCh(char ch)
  {   this.ch = ch;   }
 	
  public String toString()
  {
    String str= "Moo:("+this.n+") :";
    for (int k=1; k<=this.n; k++)
      str+= (char)this.ch + " ";   //(1)
    return str;
  }
}
``	

```java
public class Moo
{
  private int n;
  private int ch;

 public Moo(int n, char ch)
  {
    this.n = n;
    this.ch = ch;
  }

  public int getN()
  {    return this.n;       }

  public int getCh()
  {    return this.ch; }

  public void setN(int n)
  {    this.n = n;          }

  public void setCh(char ch)
  {    this.ch = ch;        }

  public String toString()
  {
    String str= "Moo:("+this.n+") :";
    for (int k=1; k<=this.n; k++)
      str+= (char)this.ch + " ";   //(1)
    return str;
  }
}
```

```java
public class Goo
{
   private Moo moo;

   public Goo(Moo moo)
   {
      this.moo = moo;
   }
   public Moo getMoo()
   {
      return this.moo;
   }
   public String toString()
   {
      return "Goo["+this.moo+"]";
   }
}
```

```java
public class TestGoo
{
   public static void main(String[] args)
   {
      Moo moo1 =new Moo(3,'d');
      Goo goo = new Goo(moo1);
      System.out.println(goo);
      Moo moo2 = goo.getMoo();
      moo1.setCh('x');
      moo2.setN(2);
      System.out.println(moo1);
      System.out.println(moo2);
      System.out.println(goo);
   }
}
```

א- הוסף פעולה בונה מעתיקה למחלקה Moo .

ב- הסבר מה משמעות של (char) בהוראה (1) במחלקה Moo ? מה היה קורה ללא (char) ?

ג- עקוב אחר הפעולה הראשית במחלקה TestGoo בעזרת תרשים עצמים ורשום את הפלט.

ד- תקן את קוד המחלקה Goo כך שפלט הפעולה הראשית של TestGoo יהיה:

הוסף הסבר לתיקון שבצעת.

```
Goo[Moo:(3) :d d d ]
Moo:(3) :x x x
Moo:(2) :d d
Goo[Moo:(3) :d d d ]
```

שאלה 11: (אביטל Evi גרינולד)

המורה למדעי המחשב מעוניינת לנתח את חיסורי התלמידים משיעורים.

לשם כך יצרה את המחלקות הבאות:

```
public class StudentAbsence
{
   private String name;
   private int absences;
   private static int counterAbsences = 0;
   private static int countLessons=0;

   public StudentAbsence(String name)
   {
      this.name = name;
      this.absences = 0;
   }

   public void increaseAbsence()
   {
      StudentAbsence.counterAbsences++;              // (1)
      this.absences++;                               // (2)
   }

   public static void addLesson(int lessons)
   {
      StudentAbsence.countLessons += lessons;
   }

  public double percentAbsence()
   {
      if (StudentAbsence.countLessons == 0)          // (3)
         return 0;
      return (double)this.absences/StudentAbsence.countLessons * 100;  // (4)
   }

   public static int getCounterAbsences()
   {
```

```
        return StudentAbsence.counterAbsences;
    }

    public static int getCountLessons()
    {
        return StudentAbsence.countLessons;
    }

    public String toString()
    {
        String str="Studnet name: "+this.name+"\n";
        str+="number of absences: "+this.absences+", ";
        str += "pecent of absences is: "+ this.percentAbsence();
        return str;
    }
}
```

```
public class TestStudentAbsence
{
    public static void main(String[] args)
    {
        (1)  StudentAbsence stud1 = new StudentAbsence("Din");
        (2)  StudentAbsence stud2 = new StudentAbsence("Gil");
        (3)  StudentAbsence.addLesson(12);
        (4)  stud1.increaseAbsence();
        (5)  stud2.increaseAbsence();
        (6)  stud1.increaseAbsence();
        (7)  stud1.increaseAbsence();
        (8)  stud2.increaseAbsence();
        (9)  System.out.println("Number of lessons is: "+ StudentAbsence.getCountLessons());
        (10) System.out.println(stud1);
        (11) System.out.println(stud2);
        (12) System.out.println("Total absences is: "+StudentAbsence.getCounterAbsences());
    }
}
```

א) הסבר מה ההבדל בין התכונה absences לתכונה *counterAbsences*.

ב) הסבר מה תקפיד התכונה *countLessons* ומדוע הוגדרה כ **static** ?

ג) בפעולה ()**increaseAbsence** .

הסבר מה התפקיד של כל אחת מההוראות (1),(2) ומדוע הן כתובות בצורה שונה.

ד) הסבר את ההוראות שבפעולה ()**percentAbsence**.

לשם מה קיימת ההוראה (3) שבפעולה? מה היה קורה לולא הייתה קיימת?

מה משמעות של (**double**) בהוראה (4) ? מה היה קורה ולא הייתה קיימת?

ה) עקוב אחר הפעולה הראשית שבמחלקה **TestStudentAbsence** בעזרת תרשים עצמים ורשום את הפלט.

<u>שאלה 12:</u> (זהבית בר לוי)

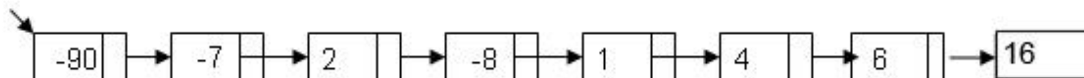
א. כתוב פעולה רקורסיבית המקבלת כפרמטרים מיקום pos (הפנייה) ברשימה מקושרת lst של מספרים שלמים ומספר שלם num ומחזירה את סכום המספרים הנמצאים במקומות שהם כפולה של num החל מ-pos.

<u>הנחה</u>: num קטן מאורך הרשימה.

ב. **רשימת סכומים עולה** היא רשימה מקושרת באורך 2n המכילה מספרים שלמים שבה מתקיים התנאי הבא: סכום כל האיברים במקומות הזוגיים גדול מסכום כל האיברים ברשימה, סכום כל האיברים במקומות שהם כפולה שלמה של 3 גדול מסכום כל האיברים במקומות הזוגיים וכך הלאה <u>עד סכום המספרים במקומות שהם כפולה של n</u> .

כתוב פעולה חיצונית המקבלת רשימה מקושרת lst ומחזירה 'אמת' אם היא רשימת סכומים עולה ו'שקר' אחרת תוך שימוש בפעולה שכתבת בסעיף א'.

דוגמה לרשימה כזו עבור n=4:



-90-7+2-8+1+4+6+16=-76

-7-8+4+16=5

2+4=6

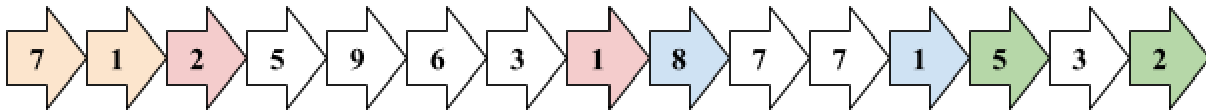
-8+16=8

שאלה 13 (אסף צדיק)

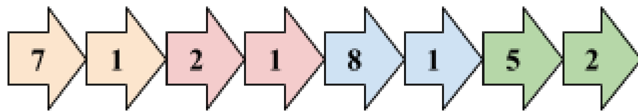
כתבו **פעולה** המקבלת תור של מספרים שלמים, **הפעולה** תחזיר תור חדש שבו עבור כל תת-סדרה יורדת של מספרים מתוך תור המקורי, בתת-סדרה יורדת כוללת לפחות שני מספרים, יהיו רק זוג מספרים, המספר הראשון והמספר האחרון מתת-הסדרה. התור המקורי יישאר ללא שינוי.

דוגמה:

עבור תור המספרים השלמים להלן:



יתקבל תור המספרים השלמים הבא:



שאלה 14 (אסף צדיק)

כתבו **פעולה** המקבלת מחסנית. הפעולה תחזיר מחסנית בה כל 2 איברים צמודים מסוכמים והסכום מוחזר למחסנית המקור. אם מספר הערכים במחסנית אי-זוגי, יוחזר המספר הבודד (כסכום ערכו + אפס).

שאלה 15 (דפנה לוי רשתי)

נתונות 4 פעולות. הראשונה מעבירה את כל איברי התור הראשון לתור השני.

```
public static void Na(Queue<int> q, Queue<int> q1)
{
                    // הפעולה מעבירה את כל איברי התור q1 לתור q
    if (!q1.IsEmpty())
    {
        q.Insert(q1.Remove());
        Na(q, q1);
    }
}
public static void Nach(Queue<int> q)
{
    Queue<int> q1 = new Queue<int>();
    while (!q.IsEmpty())
```

```
    {
        q1.Insert(q.Head());
        q1.Insert(q.Remove());
    }
    Na(q, q1);
}
public static Queue<int> Nachman(Queue<int> q)
{
    Queue<int> q1 = new Queue<int>();
    MeUman(q, q1);
    Na(q1, q);
    return q1;
}
private static void MeUman(Queue<int> q, Queue<int> q1)
{
    if (!q.IsEmpty())
    {
        int x = q.Remove();
        q1.Insert(q.Remove());
        MeUman(q, q1);
        q.Insert(x);
    }
}
```

להלן הפעולה הראשית:

```
static void Main(string[] args)
{
    1.    Queue<int> q = new Queue<int>();
    2.    q.Insert(1);
    3.    q.Insert(2);
    4.    q.Insert(3);
    5.    Console.WriteLine(q);
    6.    Nach(q);
    7.    Console.WriteLine(q);
    8.    q = Nachman(q);
```

```
9.        Console.WriteLine(q);
}
```

א. (0.5 נקודה) כיצד יראה הפלט בשורה 5?

ב. **הפעולה Nach**:

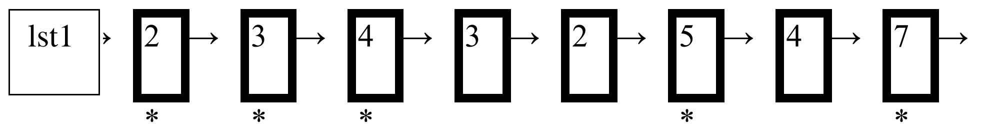
1. (2 נקודה) עקוב אחר הפעולה Nach(q) בשורה 6, בכל דרך מסודרת המתאימה לך. הראה את מצב התור בכל שלב. אין צורך לעקוב אחר הפעולה Na(q,q1)
2. (0.5 נקודה) כיצד יראה הפלט בשורה 7?
3. (2 נקודה) מה עושה הפעולה Nach(q)?

ג. **הפעולות Nachman ו-MeUman**:

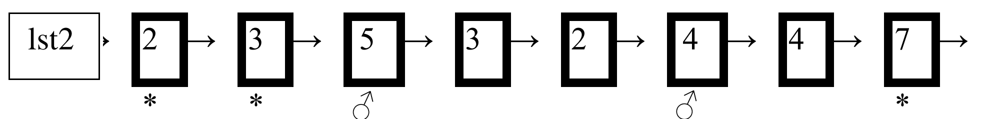
4. (6 נקודות) עקוב אחר הזימון q=Nachman(q) בשורה 8. אין צורך לעקוב אחר הזימון Na(q1,q) שבתוך הפעולה Nach. חובה לעקוב אחר הזימון Meuman(q,q1) שבפעולה.
5. (1 נקודה) כיצד יראה הפלט בשורה 9?
6. (3 נקודות) מה עושות הפעולות ?

<u>שאלה 16</u> - (דפנה לוי רשתי)

רשימה ממוינת על פי ראש היא רשימה מטיפוס שלם שבה עבור כל זוג איברים ברשימה, המופע הראשון של האיבר הקטן נמצא לפני המופע הראשון של האיבר הגדול:



lst1 היא רשימה ממוינת על פי ראש. ברשימה סומנו המופעים הראשונים של איברים זהים.



lst2 אינה רשימה ממוינת על פי ראש. ברשימה סומנו המופעים הראשונים של איברים זהים. שימו לב כי המופע הראשון של 4 הינו אחרי המופע הראשון של 5.

א. (8 נקודות) כתבו פעולה אשר בודקת האם רשימה נתונה היא ממוינת על פי ראש. אין לעבור על הרשימה הנתונה יותר מפעם אחת.

ב. (2 נקודות) מה סיבוכיות זמן ריצה של הפעולה שכתבת? נמק?

ג. (15 נקודות) כתבו פעולה המקבלת מחסנית שבה רשימות של מספרים שלמים, הפעולה תוציא מהמחסנית את **הרשימות הממוינות על פי ראש**, תחבר אותן **לרשימה אחת**, שתוחזר בפעולה.

סדר הרשימות ברשימה יהיה על פי סדר הכנסתן למחסנית. סדר האיברים בתוך כל רשימה אינו משתנה.

יש להשתמש בפעולה מסעיף א.

שאלה מס' 17 (דפנה מינסטר)

נתונה הפעולה הבאה:

```
public static Queue<Integer> test (Queue<Integer> Q1 , Queue<Integer> Q2)
{
        // הפעולה מקבלת 2 תורים: התור הראשון עם ערכים, והתור השני מאותחל וריק
                if (Q1.isEmpty())
                        return Q2;
                int x = Q1.remove();
                if ( x % 2  == 0 )
                        Q2.insert(x);
                Queue<Integer> temp = test(Q1 , Q2);
                if ( x % 2  == 1 )
                        temp.insert(x);
                return temp;
}
```

נתונה גם התכנית הראשית הבאה:

```
public static void main(String[] args)
{
                Queue<Integer> Q = new Queue<Integer>();
                Q.insert(4);
                Q.insert(7);
                Q.insert(2);
                Q.insert(-6);
                Q.insert(3);
                Q.insert(9);
                System.out.println("Queue Data -> " + Q.toString());
                System.out.println("Queue Data -> " +
```

```
test(Q , new Queue<Integer>() ).toString());
}
```

א. עקב/י אחר התכנית הראשית , מה יהיה פלט התכנית ? (חובה להראות מעקב)

ב. תן/י דוגמה לתור (המכיל לפחות 4 איברים) שהפעולה test תחזיר את אותו התור שהתקבל בתור Q1.

ג. תן/י דוגמה לתור (המכיל לפחות 4 איברים) שהפעולה test תחזיר תור הפוך לתור שהתקבל בתור Q1.

ד. כתב/י ב- 2-3 שורות מה מבצעת הפעולה **test** (Q1 **,** Q2).

ה. מהי סיבוכיות זמן הריצה (סדר הגודל) של השגרה **test** ? נמק /י !

שאלה מס' 18 (דפנה מינסטר)

המחלקה "מחסנית כפולה" **DoStacks** מכילה מערך של שתי מחסניות מטיפוס מחרוזות.

לפניך חלק מהממשק העברי לטיפול ב- **DoStacks**:

פעולה בונה המגדירה "מחסנית כפולה".	DoStacks ( )
הפעולה מקבלת איבר x ומספר המחסנית place לתוכה יוכנס האיבר.	void insert (String x **,** int place)
פעולה רקורסיבית המקבלת איבר x ומספר מחסנית place, מחזירה 'אמת' אם האיבר x נמצא במחסנית place, אחרת מחזירה 'שקר'.	boolean isExist (String x **,** int place)
הפעולה מחזירה מחסנית עם האיברים המשותפים של שתי המחסניות (פעולת חיתוך בין שתי המחסניות).	Stack < String > intersect ( )
הפעולה מחזירה 'אמת' אם 2 ה"מחסניות הכפולות" זהות, אחרת מחזירה 'שקר'.	boolean equals (DoStacks ds)
הפעולה מחזירה מחרוזת המתארת את ה"מחסנית הכפולה".	String toString ( )

א. ממשו ב- JAVA את תכונה/ות המחלקה DoStacks , ואת הפעולה הבונה.

ב. ממשו את הפעולה boolean **isExist** (String x **,** int place).

הערה: אין לפגוע בקלט ה"מחסנית הכפולה".

ג. ממשו את הפעולה Stack < String > **intersect** ( ).

הערה: אין לפגוע בקלט ה"מחסנית הכפולה".

ד. מהו סדר הגודל (סיבוכיות זמן הריצה) של הפעולות בסעיפים ב'-ג' ? נמק/י!

<u>שאלה 19</u> (ויליאם פרג'ון)

לפניך הפעולה sum הכתובה במחלק הראשית

```
publicstaticint sum(Stack <Integer>s)   {
if (s.isEmpty())
return 0;
int x=s.pop();
if(x%6==0)
return x+sum(s);
else return sum(s);
  }
```

נתונה המחסנית s.

12	4	33	6	30	0

א. עקוב אחר הפעולה sum עבור מחסנית s, ורשום את הערך שיוחזר, במעקב הראה את המעבר על המחסנית s.

לפניך הפעולה sod. קטע תכנית המשתמש בפעולה

```
publicstatic void sod(Queue<Stack<Integer>>qq,Queue<Integer>qm){
if(!qq.isEmpty()){
int x=sum(qq.remove());
      qm.insert(x);
     sod (qq, qm);
   }
  }
publicstaticvoid main(String [] args) {
   Stack<Integer> s1= new Stack<Integer>();
   Stack<Integer> s2= new Stack<Integer>();
   Queue<Stack<Integer>>q1= new Queue<Stack<Integer>>();
    s1.push(0);   s1.push(30);   s1.push(6);   s1.push(33);   s1.push(4);   s1.push(12);
    s2.push(23);   s2.push(36);   s2.push(1);
   q1.insert(s1);
   q1.insert(s2);
Queue<Integer> qr= new Queue<Integer>();
    sod(q1,qr);
```

}

ב. סרטט את כל אחד משני התורים q1-ו q2 לפני הקריאה לפעולה sod ואחרי ביצוע הפעולה sod.

ג. חשב את סיבוכיות הפעולה sum וסיבוכיות הפעולה sod.

<u>שאלה 20</u> (ויליאם פרג'ון)

נתונה שרשרת שבה כל חוליה מכילה הפנייה לשרשרת של ספרות חיוביות בטווח 0-9 (שרשרת של שרשרות).

כל אחת מהשרשראות מייצגת מספר כאשר האיבר הראשון הוא ספרת האחדות, השני הוא ספרת העשרות וכו' (כלומר – צריך להפוך את סדר האיברים ברשימה כדי לפענח את המספר). ייתכן שחוליה מחזיקה null (משמעות – אין מספר).

לדוגמה: שרשרת שאיבריה הם השרשראות: [[1, 6, 9],[null], [2, 8], [4, 2, 7], [6, 2]] מייצגת את המספרים:961, 28, 427, 62

א. כתוב פעולה המקבלת את מבנה הנ"ל ומחזירה את המספר הגדול ביותר המיוצג בשרשרת.
עבור השרשרת שתוארה בדוגמא לעיל תחזיר הפעולה **961**.
כותרת הפעולה תהיה **publicstaticint max(Node<Node<Integer>> lst)**

ב. כתוב פעולה המקבלת שרשרת של שרשראות במבנה הנ"ל ומחזירה את שרשרת החדשה המורכבת מהספרות המשמעותיות ביותר. עבור דוגמה הנ"ל הפעולה תחזיר את שרשרת הבאה: [6,4,2,9].

ג. מה סדר הגודל [O(?)] של סיבוכיות זמן הריצה של הפעולה שכתבת בסעיף ב'?
נמק את תשובתך והגדר מפורשות מהו **n**

<u>שאלה 21</u> (לילך לגזיאל)

הפעולה Sum הכתובה במחלקה ראשית.

```
public static int Sum (Stack<int> s)
{ if (s.IsEmpty()) return 0;
  int x=s.Pop();
  If (x%6==0)
       return x+ Sum(s);
  return Sum(s);
}
```

נתונה המחסנית s

12
4
33
6
30
0

א. עקוב אחר הפעולה Sum בעבור המחסנית s, ורשום את הערך שיוחזר.
במעקב הראה את המעבר על המחסנית s.

ב. לפניך הפעולה Sod.

```
public static void Sod(Queue<Stack<int>> qq, Queu<int> qm)
 { if (!qq.IsEmpty())
     {  int x=Sum (qq.Remove());
        qm.Insert(x);
        Sod (qq,qm);
     }
 }
```

לפניך קטע תכנית המשתמש בפעולה Sod.

סרטט את כל אחד משני התורים q1 ו- qr, לפני הקריאה לפעולה Sod ואחרי ביצוע הפעולה Sod.

```
public static void Main(string[] args)
{   Stack<int> s1=new Stack<int>();
   Stack<int> s2=new Stack<int>();
  Queue<Stack<int>> q1= new Queue<Stack<int>>();
  s1.Push(0);    s1.Push(30);
  s1.Push(6);    s1.Push(33);
  s1.Push(4);    s1.Push(12);
  s2.Push(23);    s2.Push(36);
  s2.Push(1);
  q1.Insert(s1);
  q1.Insert(s2);
  Queue<int>gr=new Queue<int>();
  Sod (q1, qr);
}
```

<u>שאלה 22</u> (אביטל Evi גרינולד)

נגדיר **רשימה-טריו** רשימה של מספרים שלמים ריקה או בנויה משלשות של חוליות, בכל שלשה הערכים הם בסדר עולה ממש, וסכום כל שלשה גדול מסכום השלשה הקודמת.

דוגמה ל **רשימת-טריו** → 2 → 5 → 8 → 0 → 9 → 7 →

הסבר: הרשימה בנויה משתי שלשות של מספרים. בכל שלשה , המספרים הם בסדר עולה ממש.
סכום שלשה ראשונה הוא 15 וסכום שלשה שנייה הוא 16

הרשימות הבאות הן לא **רשימות-טריו**

→ 2 → 5 → 8 → 0 → 4 → 7

סכום שלשה שנייה אינו גדול מסכום שלשה ראשונה

המספרים בשלשה ראשונה אינם בסדר עולה. → 2 → 5 → 4 → 3 → 5 → 6

הרשימה לא בנויה משלשות של מספרים. → 2 → 5 → 8 → 4 → 7 →

א- כתוב פעולה אשר מקבלת כפרמטר הפנייה לרשימה של שלמים.

הפעולה מחזירה 'אמת' אם הרשימה היא **רשימת-טריו**, 'שקר' – אחרת.

ב- מה יעילות הפעולה שכתבת בסעיף א? נמק קביעתך. בנימוק התייחס לאורך הקלט ולמעברים על המבנה.

<u>שאלה 23</u> (לילך לגזיאל)

נגדיר "**סדרת מספרי x**", כרצף המורכב מ- x מופעים של המספר x.

לדוגמא: "סדרת מספרי 3" היא רצף : 333

כתוב <u>פעולה חיצונית</u> המקבלת מחסנית s של מספרים שלמים ובונה מחסנית חדשה כך שעבור כל מספר x של מחסנית s תכיל המחסנית החדשה "**סדרת מספרי x**".

לדוגמא, עבור מחסנית s:

1
1
4
3
1
2

תוחזר המחסנית

1
1
4
4
4
4
3
3
3
1
2
2

ניתן להשתמש בכל פעולות הממשק

של המחלקה Stack<T> מבלי לממש אותן.

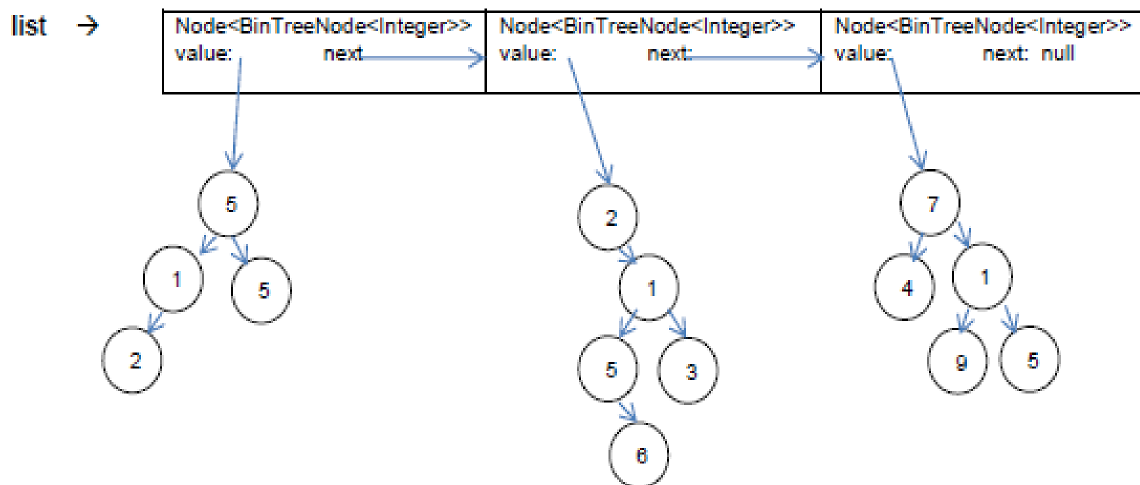
<u>שאלה 24</u>: (אביטל Evi גרינולד)

כתוב פעולה המקבלת רשימה, כל איבר ברשימה הוא הפנייה לעץ חוליות בינארי מטיפוס שלמים.

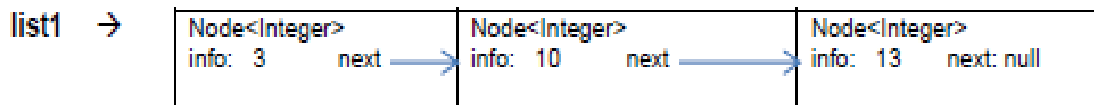
הפעולה תחזיר רשימה של שלמים באופן הבא:

עבור כל איבר מהרשימה שערך השורש של העץ שבו הוא זוגי, תכניס לרשימה החדשה איבר עם סכום הבנים הימניים של העץ, ואילו עבור כל איבר מהרשימה שערך השורש של העץ שבו הוא אי-זוגי, תכניס לרשימה החדשה איבר עם סכום הבנים השמאליים.

לדוגמה, עבור הרשימה הבאה:



תתקבל הרשימה הבאה



הסבר:

- עבור החוליה הראשונה יתקבל סכום הבנים השמאליים של העץ כי ערך השרש הוא אי-זוגי והוא 1+2=3
- עבור החוליה השנייה יתקבל סכום הבנים הימניים של העץ כי ערך השרש הוא זוגי 1+3+6 = 10
- עבור החוליה הראשונה יתקבל סכום הבנים השמאליים של העץ כי ערך השרש הוא אי-זוגי 4+9=13

<u>שאלה 25</u> (אביטל Evi גרינולד)

לפניך שתי פעולות חיצוניות:

```
// הפעולה מקבלת תור לא ריק של מספרים שלמים ומספר האיברים בתור
public static void sod1 (Queue<Integer> q, int n)
{
        if (n > 1)
        {
                int x=q.remove();
                int y=sod2(q,n-1,x);
                sod1(q,n-1);
                q.insert(y);
        }
}
```

```
// הפעולה מקבלת תור לא ריק של מספרים שלמים, מספר אברים ומספר שלם נוסף
public static int sod2 (Queue<Integer> q,int n,int x)
{
	while (n>0)
	{
		int y=q.remove();
		n--;
		if (x > y)
		{
			q.insert(x);
			x = y;
		}
		else
			q.insert(y);
	}
	return x;
}
```

א. מה יחזיר הזימון sod2(q, n, x) כאשר : q = [8,10,2,5], n = 4 , x = 3
ומה מצב התור עם סיום הפעולה. חובה להראות מעקב.

ב. מה מבצעת הפעולה sod2(q, n, x) באופן כללי?

ג. מה יהיה מצב התור אחרי הזימון sod1(q, n) כאשר: q = [5,7,10,2,4] , n = 5
חובה להראות מעקב. אין צורך להראות שוב מעקב עבור sod2.

ד. מה מבצעת הפעולה sod1(q, n) באופן כללי?

ה. כתוב את הפעולה sod2 מחדש כפעולה רקורסיבית.