פרק 10
עץ בינרי
מבנה חוליות הירכי

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.

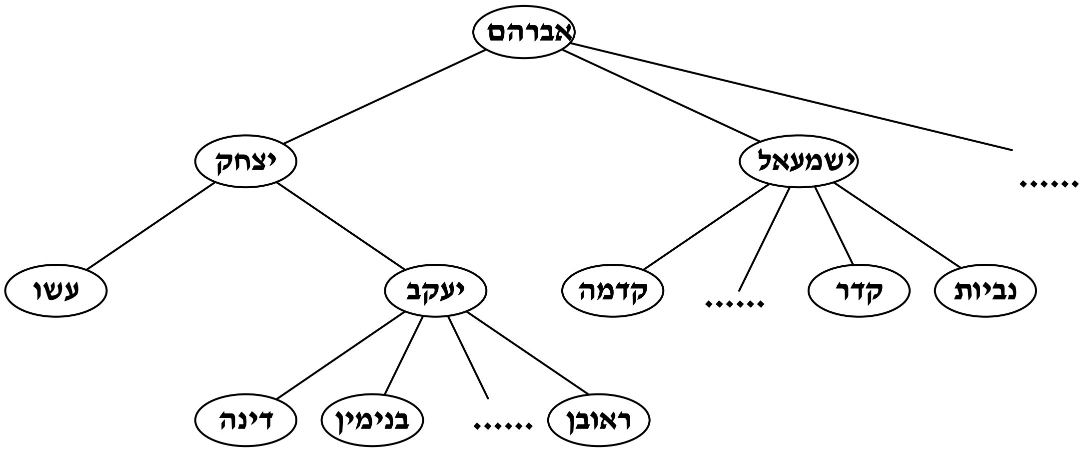
נביות	קדר	קדמה	יעקב	עשו	ישמעאל	יצחק	אברהם

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.

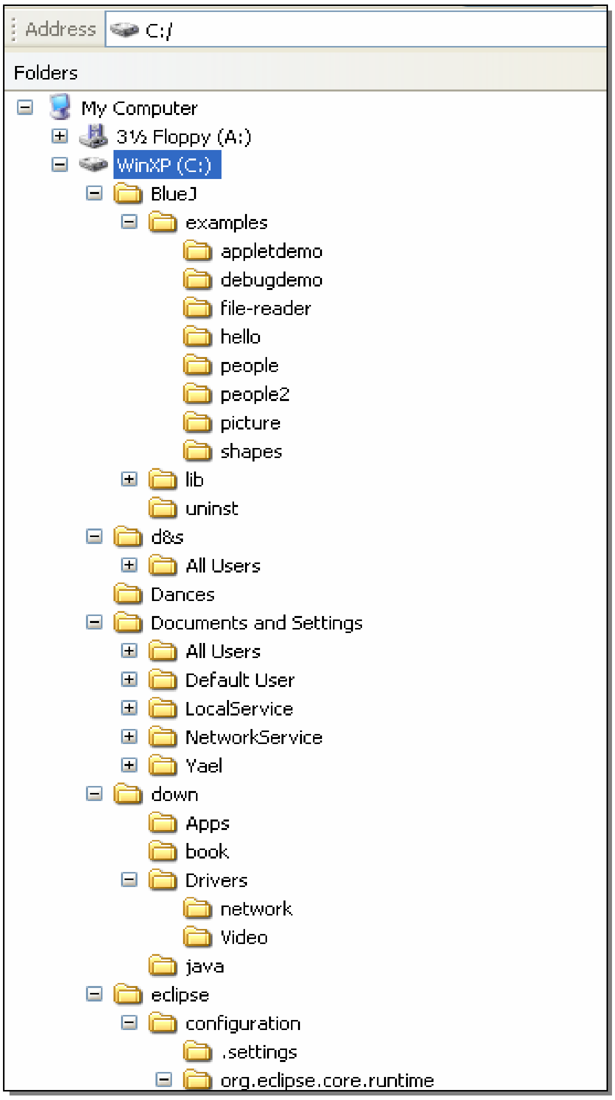
דרך מקובלת לתאר קשרי משפחה של אדם היא באמצעות ציור אילן יוחסין. האילן יכול להיות עץ צאצאים – ראשיתו באדם מסוים והוא מסתעף קדימה אל צאצאיו: ילדיו, נכדיו, ניניו וכו׳, והוא יכול להיות עץ אבות, שמסתעף מן האדם אל עֲבָרוֹ: הוריו, סביו, רב-סביו וכו׳.

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**).

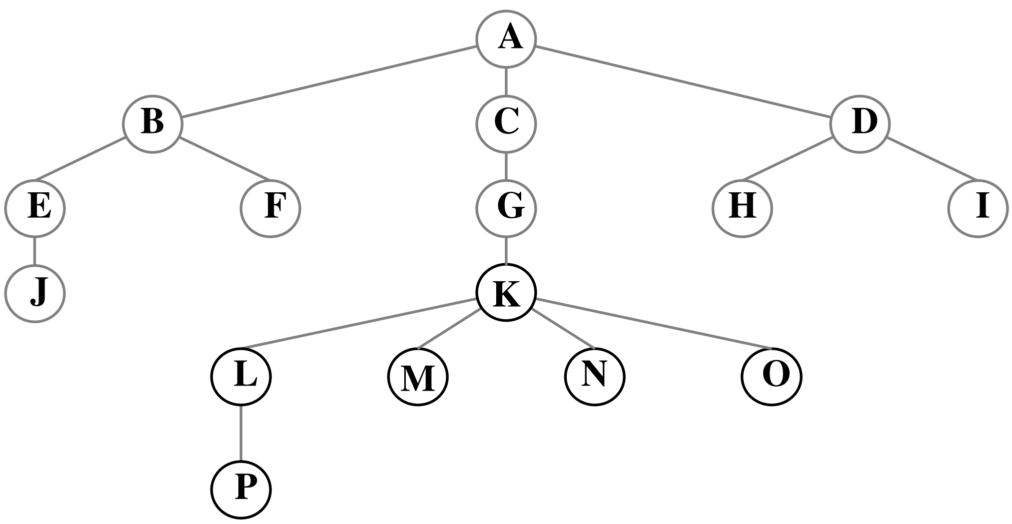
הילדים של צומת אחד בעץ צאצאים הם ילדיו הביולוגים של האדם המיוצג בצומת. בעץ מערכת הקבצים, לעומת זאת, ילדיה של ספרייה המופיעה בצומת הם הקבצים והספריות שהיא מכילה.

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.

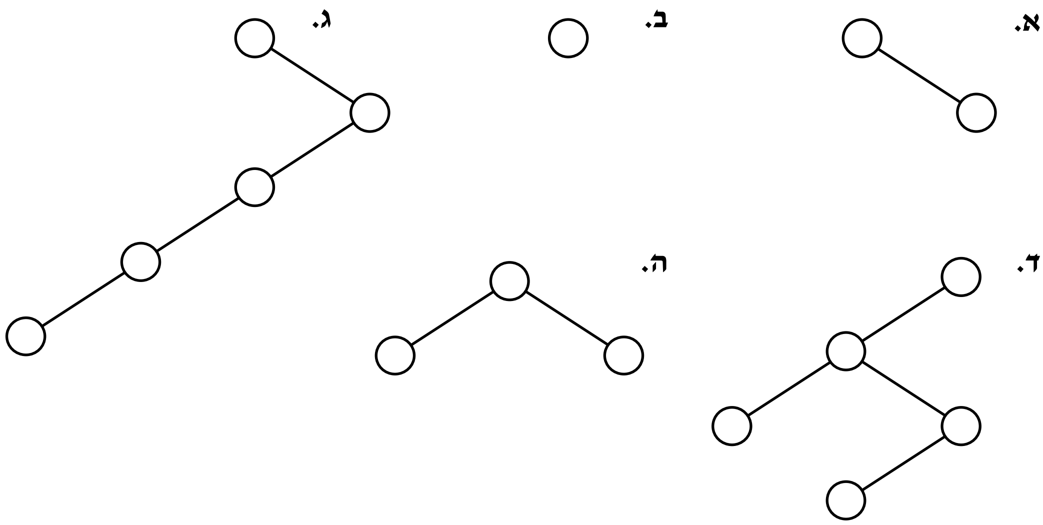
ב. עצים בינריים

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.

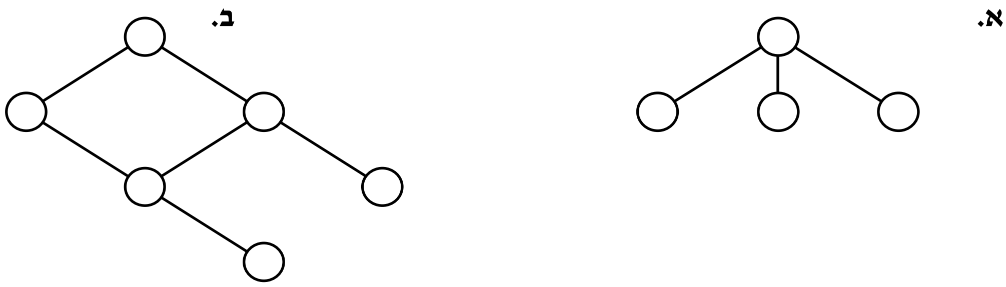
נגדיר **עץ בינרי** (**Binary Tree**) כעץ שיש בו לכל היותר שני ילדים לצומת. מקובל להתייחס אליהם כילד שמאלי וילד ימני. כל אחד משני אלה, אם הוא קיים, הוא שורש של עץ. הילד השמאלי הוא שורשו של עץ הנקרא **תת-עץ שמאלי** (**left subtree**) של הצומת, והילד הימני הוא שורשו של עץ הנקרא **תת-עץ ימני** (**right subtree**). 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 BinTreeNode** (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 BinTreeNode. 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: BinTreeNode<T>.

ג.1. ממשק המחלקה BinTreeNode<T>

כמו לכל מחלקה, גם עבור המחלקה BinTreeNode<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 ()getInfo ו-(…)setInfo 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 BinTreeNode<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**	BinTreeNode (T x)
הפעולה בונה חוליה בינרית שערכה יהיה x. left ו-right הן (הפניות אל) הילד השמאלי והימני שלה. ערכי ההפניות יכולים להיות **null**	BinTreeNode (BinTreeNode<T> left, T x, BinTreeNode<T> right)
הפעולה מחזירה את הערך של החוליה	T getInfo()
הפעולה משנה את הערך השמור בחוליה ל-x	**void** setInfo (T x)
הפעולה מחזירה את הילד השמאלי של החוליה. אם אין ילד שמאלי הפעולה מחזירה **null**	BinTreeNode<T> getLeft()
הפעולה מחזירה את הילד הימני של החוליה. אם אין ילד ימני הפעולה מחזירה **null**	BinTreeNode<T> getRight()
הפעולה מחליפה את הילד השמאלי בחוליה left	**void** setLeft (BinTreeNode<T> left)
הפעולה מחליפה את הילד הימני בחוליה right	**void** setRight (BinTreeNode<T> right)
הפעולה מחזירה מחרוזת המתארת את הערך השמור בחוליה	String toString()

ג.2. המחלקה BinTreeNode<T>

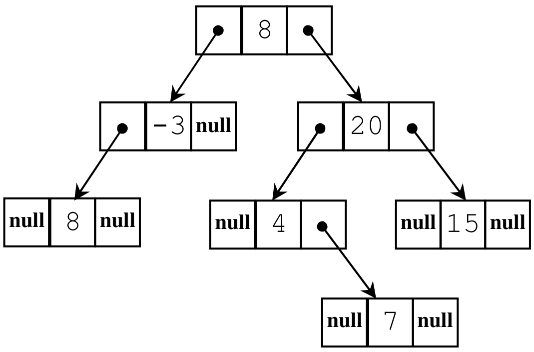
להלן מימוש המחלקה:

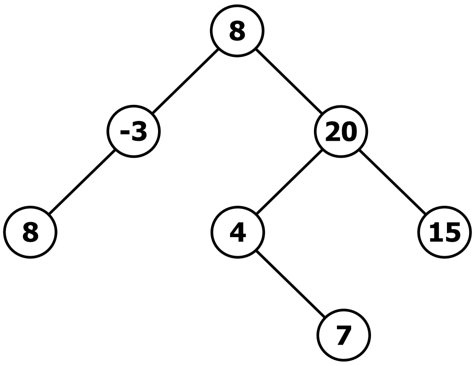
```
public class BinTreeNode<T>
{
      private BinTreeNode<T> left;
      private T info;
      private BinTreeNode<T> right;

      public BinTreeNode(T x)
      {
            this.left = null;
            this.info = x;
            this.right = null;
      }

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

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

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

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

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

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

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

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

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

ד. עץ חוליות בינרי

המחלקה BinTreeNode 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".

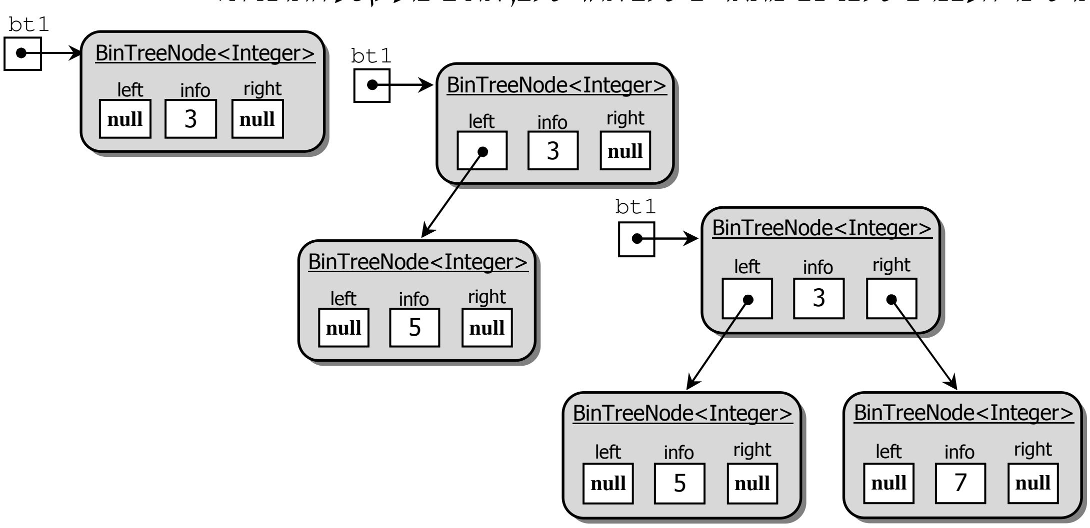
בסעיפים הבאים נדגים כיצד ניתן לבנות עץ חוליות בינרי ונדון בתכונותיו.

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(…):

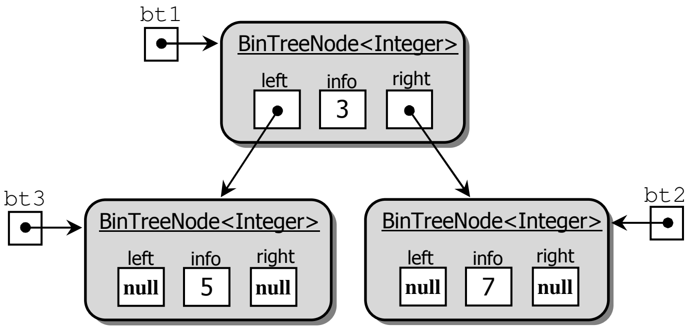
```
BinTreeNode<Integer> bt1 = new BinTreeNode<Integer>(3);
bt1.setLeft(new BinTreeNode<Integer>(5));
bt1.setRight(new BinTreeNode<Integer>(7));
```

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:

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



ניתן לעשות זאת אף בפקודה אחת:

```
BinTreeNode<Integer> bt1 = new BinTreeNode<Integer>
       (new BinTreeNode<Integer>(5), 3, new BinTreeNode<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 (BinTreeNode<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().getInfo();
```

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

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

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

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

```
BinTreeNode<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 שרשרת? התשובה היא אמנם חיובית, אך הוספה כזו היא מסובכת. נניח כי לחוליה binTreeNode 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 binTreeNode או בצד ימין שלה. נניח כי החלטנו על הוספה בשמאל. עכשיו עולה השאלה מה נעשה עם התת-עץ השמאלי הנוכחי של binTreeNode? 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 בשרשרת, 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 binTreeNode יש תת-עץ שמאלי ששורשו הוא החוליה binTreeNode1, ואנו רוצים להוציא את binTreeNode1 מהעץ. ל-binTreeNode1 יש במקרה הכללי שני תת-עצים, שאת החוליות שלהם (פרט ל-binTreeNode1 עצמה) אנו רוצים להשאיר בעץ. אנו יכולים לחבר אחד מהם כתת-עץ שמאלי של binTreeNode, 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 שרשרות של חוליות. כל זמן ששרשרת החוליות הייתה חשופה, לא יכולנו להבטיח שמירה על המבניות הלינארית שלה. הפתרון עבור שרשרות, כפי שנוכחנו לדעת בפרקים הקודמים, היה הגדרת מחלקה עוטפת, שיש לה הפניה פנימית לשרשרת ופעולות המטפלות בשרשרת. 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 השרשרת, שכן למשתמש במחלקה אין גישה ישירה לשרשרת. 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 לשרשרת החוליות הלינארית. כשם ששרשרת חוליות מכילה לפחות חוליה אחת, כך גם עץ חוליות בינרי מכיל לפחות חוליה אחת.

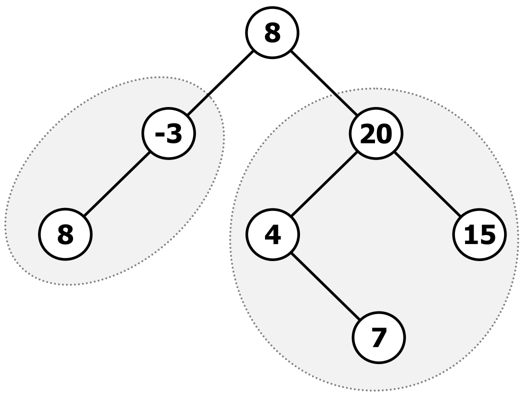
כלומר עץ חוליות בינרי הוא:

- חוליה בינרית יחידה

או

- חוליה בינרית שבה יש הפניה אחת לעץ חוליות בינרי או שתי הפניות לעצי 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 הַפְּעֻלּוֹת 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. בכל פעולה המקבלת עץ חוליות בינרי אנו מניחים את ההנחה הסמויה המקובלת עלינו ביחידת לימוד זו: הפרמטר המועבר לפעולה אינו **null**. כמו כן 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(BinTreeNode<Integer> bt)
{
      if (bt == null)
            return 0;
      return numNodes(bt.getLeft()) + numNodes(bt.getRight()) + 1;
}
```

נעיר שתי הערות:

א. למרות תנאי העצירה המחזיר את הערך 0, לא ייתכן שזה יהיה ערך ההחזרה הסופי של הפעולה. החזרת הערך 0 כערך סופי פירושה שלא שמרנו על ההנחה הסמויה והפרמטר שנשלח לפעולה היה שווה **null**. זוהי חריגה מההנחה, ולכן מתקבל ערך לא צפוי שהוא ערך חסר משמעות עבור הדוגמה הזו. 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.

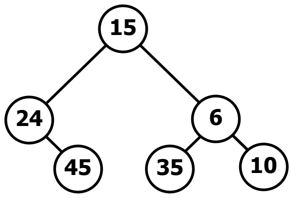
דוגמה 2: הדפסת ערכי העץ

נכתוב פעולה המדפיסה את הערכים השמורים בצומתי העץ.

```
public static void printNodes(BinTreeNode<Integer> bt)
{
      if (bt != null)
      {
            System.out.print(bt.getInfo() + " ");
            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 getInfo()

המחזירה את תוכן השורש הנוכחי, ואת הפעולה setInfo(…) 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

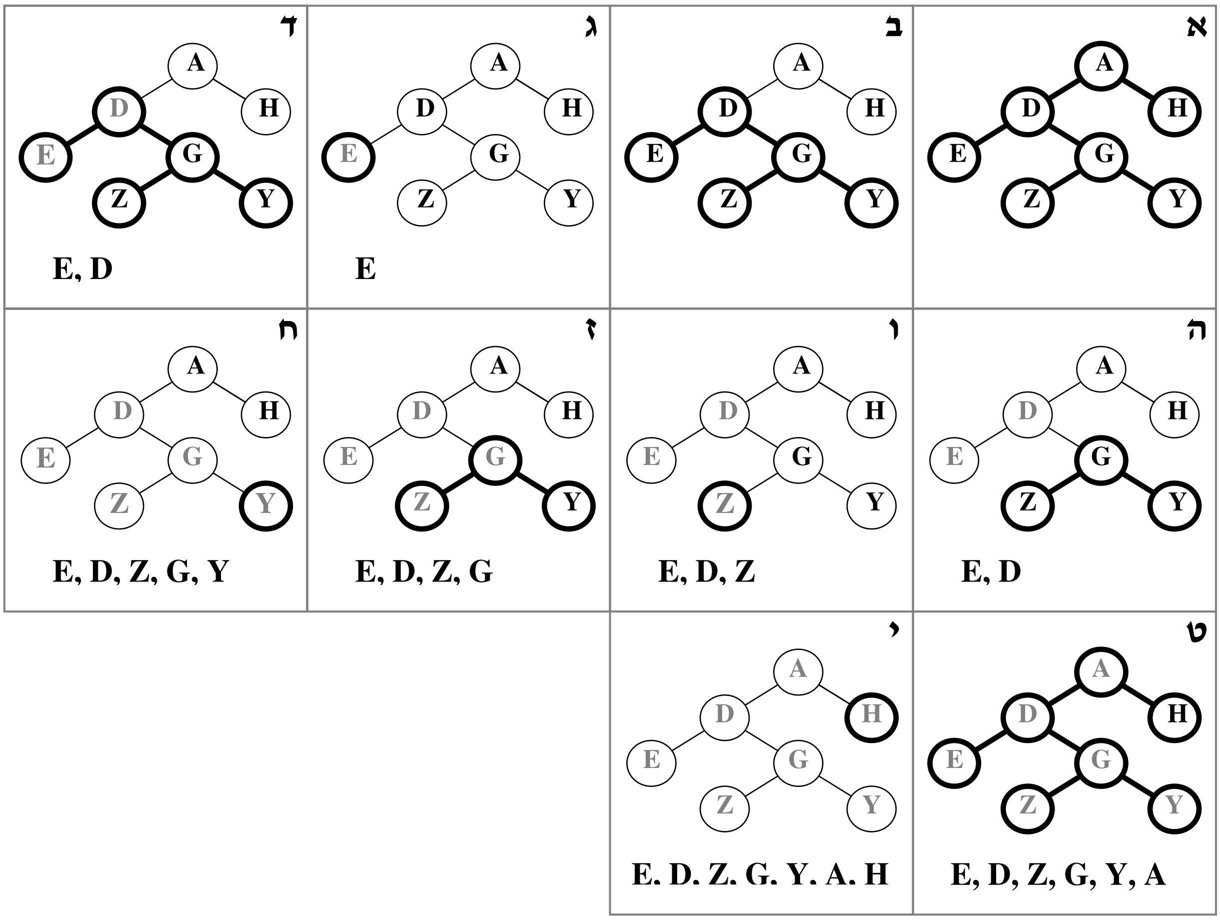
באמרנו "ביקור בשורש", אנו מתכוונים לביצוע פעולה כלשהי בצומת תוך כדי הסריקה, למשל הדפסת ערכו של הצומת. נשים לב כי במהלך סריקה אנו מבקרים פעם אחת בכל צומת, אך אנו חולפים על פני הצמתים פעמים נוספות בלי לבצע "ביקור", וזאת כדי להגיע אל ילדיהם של אותם צמתים או כדי לחזור דרכם אל הוריהם.

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(BinTreeNode<Integer> bt, int x)

{
      if (bt == null)
            return false;

      if (bt.getInfo() == 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:

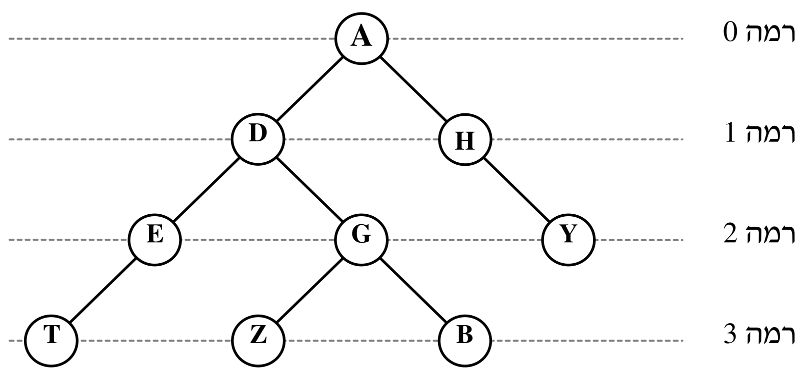
```
public static String preorderString(BinTreeNode<String> bt)
{
      if (bt == null)
            return "";
      return bt.getInfo() + " " + preorderString(bt.getLeft()) +
                                   preorderString(bt.getRight());
}
```

? 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.



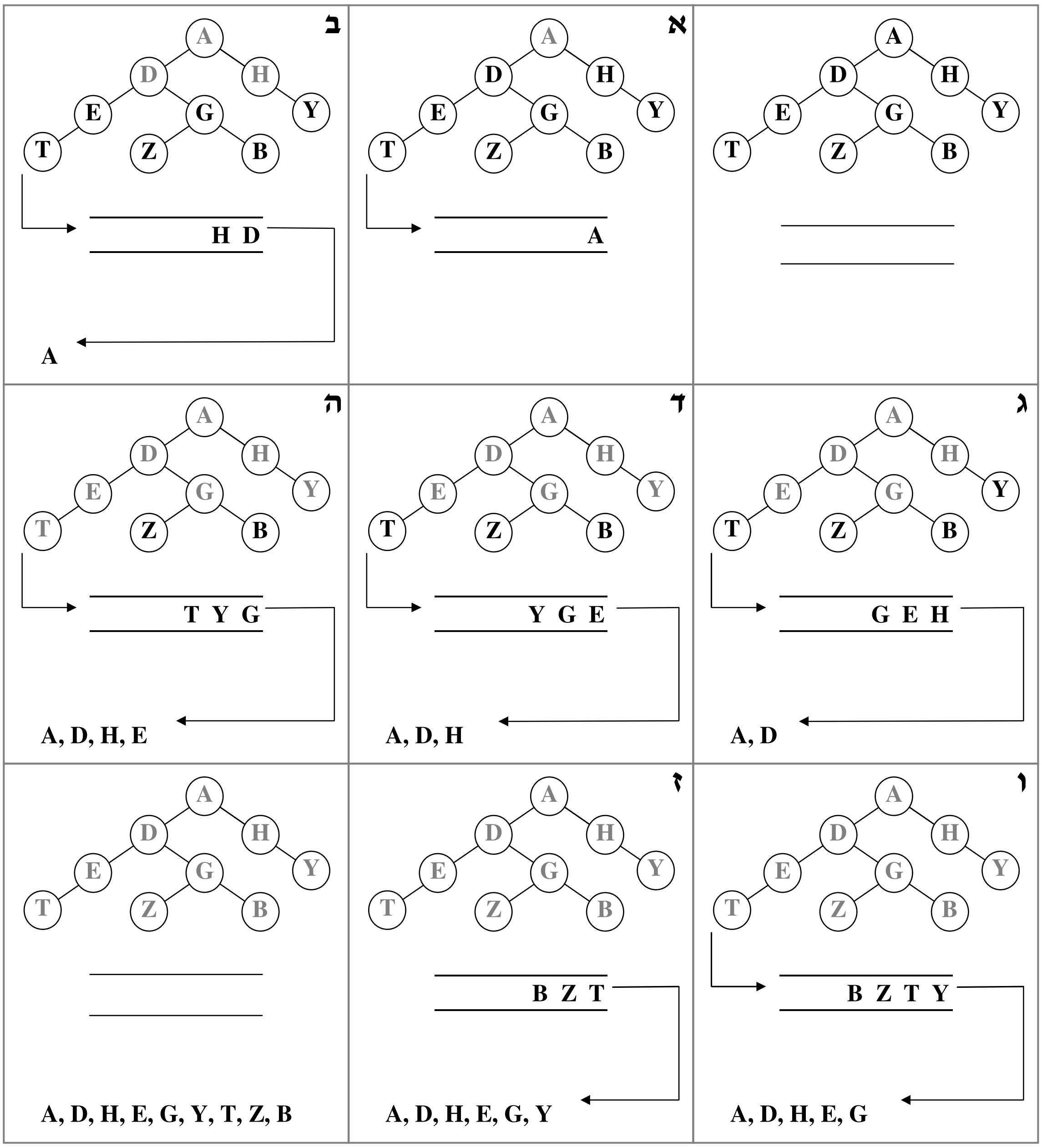
ו.1.2. אלגוריתם סריקה לפי רמות

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<BinTreeNode<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. $(4 * (3 + 5)$

ה. $3 – 2$

ו. $(–7)$

ז. (8)

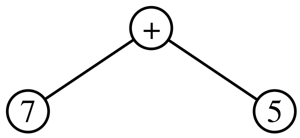
את הביטויים המקיימים את ההנחות לעיל אפשר להגדיר כך:

ביטוי חשבוני הוא: **A** כאשר A הוא מספר חד-ספרתי

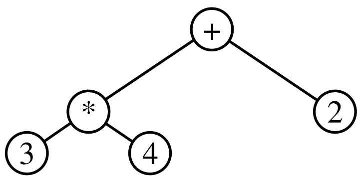
או: **(X op Y)** כאשר X ו-Y הם ביטויים חשבוניים,
האופרנדים של הפעולה, ו-op הוא פעולה.

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.

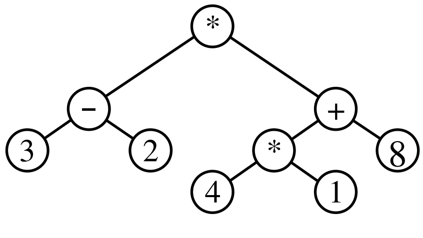
א. הביטוי $(7 + 5)$:



ב. הביטוי $((3 * 4) + 2)$:



ג. הביטוי $((3 – 2) * ((4 * 1) + 8))$:



? 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.

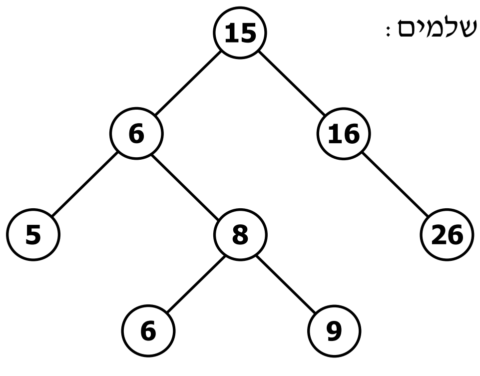
חשב-ערך-ביטוי *(tree)*
אם העץ הוא עלה, החזר את ערכו
אחרת,
חשב-ערך-ביטוי (תת-עץ שמאלי של *tree) ושמור את התוצאה ב- leftVal.*
חשב-ערך-ביטוי (תת-עץ ימני של *tree) ושמוראת התוצאה ב- rightVal.*
שמור ב-op את הערך של השורש
החזר את: (leftVal op rightVal). {תוצאת הפעולה op על שני הערכים}.

? 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.

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

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

      if (bst.getInfo() < 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(BinTreeNode<Integer> bst, int x)
```

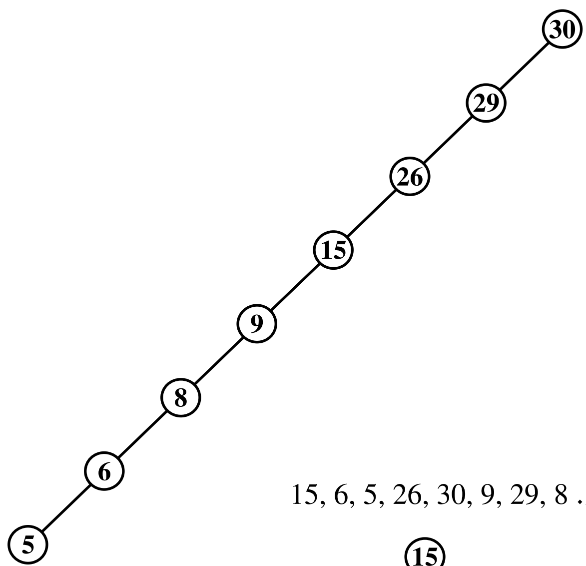
הפעולה מכניסה מספר שלם לעץ-חיפוש-בינרי המכיל מספרים.

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).

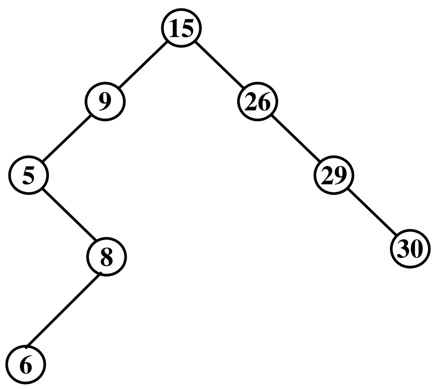
האם לסדרה נתונה של ערכים מתקבל תמיד אותו העץ, בלי תלות בסדר ההכנסה של הערכים?

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.

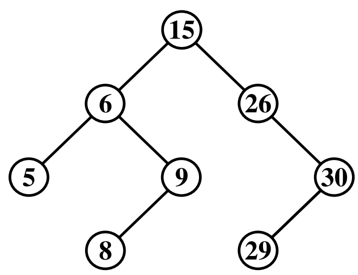
א. 30, 29, 26, 15, 9, 8, 6, 5



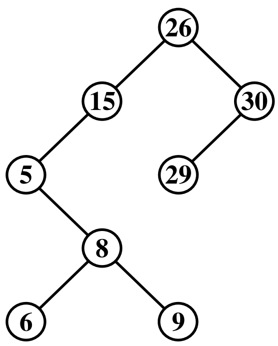
ב. 15, 9, 26, 5, 8, 6, 29, 30



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, 29 ,8 ,5 ,30 6, 9 (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.

שימו לב שמבנה העץ אינו יכול להיות בן פחות מחוליה בינרית אחת. לכן, אם העץ המכיל את x הוא עץ עלה, לא נוכל לנתק אותו מהעץ. בעיה זו תיפתר כאשר מבנה החוליות יהיה עטוף בעזרת מחלקה.

ח.5. יעילות הפעולות על עץ-חיפוש-בינרי

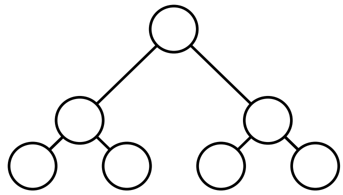
אם נשווה זה לזה את העצים שבאיורים א ו-ג לעיל, נראה שהם אמנם מכילים אותם ערכים, אך חיפוש בעץ הראשון יהיה יעיל פחות מאשר בעץ השני, כיוון שהראשון גבוה יותר. גם פעולת הכנסה לעץ גבוה יעילה פחות מאשר לעץ נמוך, כיוון שגם פעולת ההכנסה עוברת על מסלול בעץ. עד כה לא דנו בפירוט ביעילותן של פעולות החיפוש וההכנסה 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

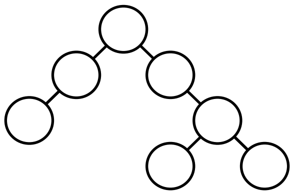
המקסימלי לעץ המכיל שבעה צמתים. שני העצים באיורים ב ו-ג מתארים מצבי ביניים, וגובהם 3 ו-4 בהתאמה.

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

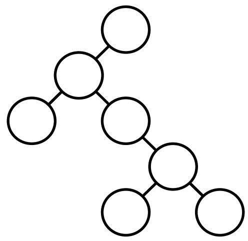
א.



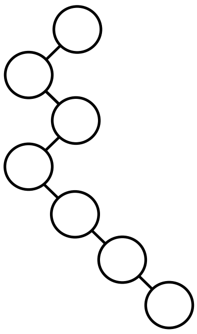
ב.



ג.



ד.



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. 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 לשרשרת חוליות 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 BinTreeNode 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 ששרשרת 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 (שרשרת

חוליות 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, שרשרת 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-BinTreeNode; מבני הנתונים: שרשרת חוליות, 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, המחלקה List לא מימשה טיפוס נתונים מופשט, ונשארה בגדר טיפוס נתונים. גם עץ חוליות בינרי ועץ-חיפוש-בינרי הם מבני נתונים. בגלל איזה הבדל בין סוגי האוספים הצלחנו בהסתרת המימוש ברוב האוספים, אך למשל לא ברשימות?

מתברר שההבדל פשוט למדי. 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.

לא כך המצב ברשימה. ברשימה רצינו לאפשר למשתמש גישה לאיבר על פי מקומו. אילו הסתפקנו במקום שהוא אינדקס מספרי, יכולנו לייצג רשימה בעזרת מערך או בעזרת שרשרת חוליות, ולממש את הפעולות בלי שהמשתמש ידע באיזה ייצוג השתמשנו. אז היינו מקבלים הסתרה מלאה של המקום, ולכן מקבלים מימוש של טיפוס נתונים מופשט. אולם, לייצוג בעזרת מערך יש חסרונות, כפי שתואר בפרקים הקודמים. מחירו של שימוש באינדקס מספרי לשם גישה לאיברי שרשרת חוליות הוא גבוה. כיוון שכך, בחרנו לייצג מקום בעזרת הפניה לחוליה, ולכן נאלצנו לחשוף את הייצוג של המחלקה למשתמשים. עקב כך, רשימה אינה מימוש של טיפוס נתונים מופשט.

נסיים דיון זה בכמה הערות לגבי מימוש מחלקות המממשות סוגי אוספים כאלה בג׳אווה. מקובל שמחלקה חייבת בהסתרה מלאה. במחלקה רשימה הגישה לאיבר היא בעזרת אינדקס מספרי. גישה בעזרת הפניה לחוליה קיימת גם כן, אך בשיטה אחרת לגמרי מזו שאנו הצגנו. את 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 לעץ חוליות בינרי:
  - חוליה בינרית יחידה;

  או

  - חוליה בינרית שבה לכל היותר שתי הפניות לעצי 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.

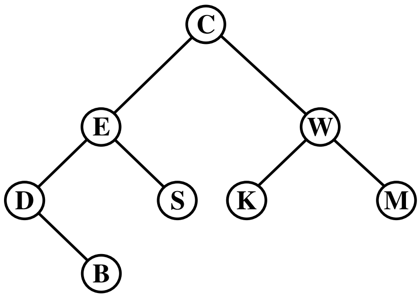
מושגים

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

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 __________.

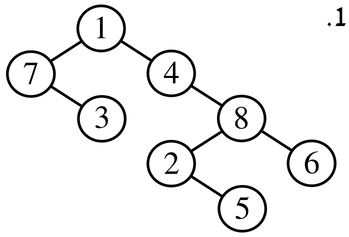
ב. עבור כל אחד מהמשפטים ציינו אם הוא נכון או לא נכון, ונמקו :

- 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

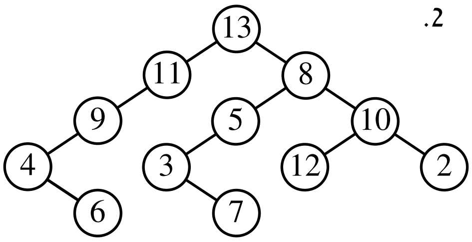
שאלה 2

לפניכם שני עצים בינריים שבצומתיהם מספרים:

.1



.2



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(BinTreeNode<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.

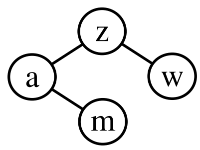
שאלה 5

לפניכם הפעולה:

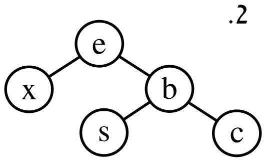
```
public static boolean secret(BinTreeNode<Character> bt)
{
      if (bt.getLeft() == null && bt.getRight() == null)
            return true;

      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:

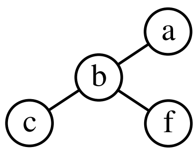
.1



.2



.3



ב. כתבו את טענת היציאה של הפעולה.

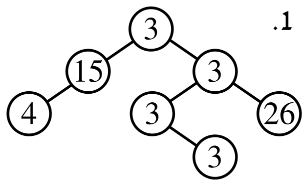
שאלה 6

לפניכם הפעולה:

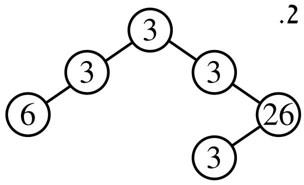
```
public static boolean what(BinTreeNode<Integer> bt, int x)
{
      if (bt == null)
            return false;

      if (bt.getInfo() !=  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:

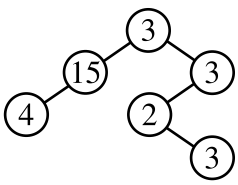
.1



.2



.3



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

שאלה 7

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

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

      return new BinTreeNode<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(BinTreeNode<Integer> bt)
{
      int ml = bt.getInfo();
      int mr = bt.getInfo();

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

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

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

כתבו את טענת היציאה של הפעולה.

שאלה 9

נתונה הפעולה הזו:

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

      do
      {
            while (node != null)
            {
                  stack.push(node);
                  node = node.getLeft();
            }
            if (!stack.isEmpty())
            {
                  node = stack.pop();
                  System.out.print(node.getInfo() + " ");
                  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 List<Integer> levelOrderList(BinTreeNode<Integer> bt)
```

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

שאלה 11

ממשו את הפעולה:

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

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

שאלה 12

ממשו את הפעולה:

```
public static void printStrings(BinTreeNode<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(BinTreeNode<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(BinTreeNode<Integer> bt)
```

הפעולה תחזיר את גובהו של העץ הבינרי bt.

שאלה 15

ממשו את הפעולה:

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

הפעולה תחזיר את מספר הצמתים ברמה level בעץ הבינרי bt.

הנחה: level>=0.

שאלה 16

ממשו את הפעולה:

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

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

שאלה 17

ממשו את הפעולה:

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

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

שאלה 18

ממשו את הפעולה:

```
public static BinTreeNode<Integer> parent(BinTreeNode<Integer> bt,
                                          BinTreeNode<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 BinTreeNode<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 BinTreeNode<Integer> buildTree(int[] preorder,
                                              int[] inorder)
```

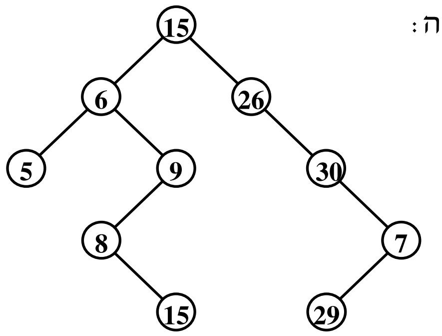
הפעולה מקבלת שני מערכים של שלמים שהתקבלו כתוצאה מסריקת עץ בינרי בסריקה בסדר תחילי ובסריקה בסדר תוכי, ומחזירה את העץ המקורי.

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 :

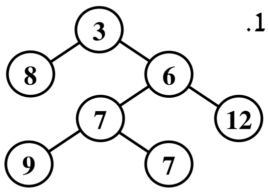


ב. ממשו פעולה המקבלת עץ בינרי של שלמים ומחזירה את **הפרש הילדים** בו.

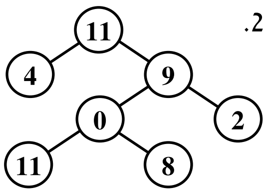
שאלה 22

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 :

.1



.2



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.

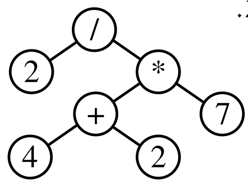
א. ציירו את העצים הבינריים המתאימים לביטויים החשבוניים :

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

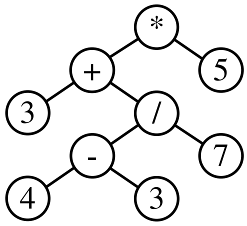
כמו כן, הראינו בסעיף ז בפרק את האלגוריתם **חשב-ערך-ביטוי** המקבל עץ ביטוי, מחשב את ערך הביטוי ומחזיר אותו.

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

.1



.2



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).

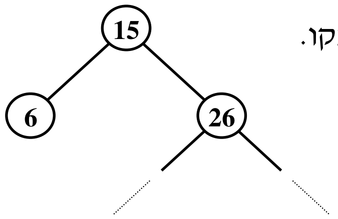
ה. נרחיב את הפעולה הקודמת: ממשו פעולה המקבלת אוסף של מספרים שלמים, כך שהאוסף המוחזר יכיל את כל המספרים ממוינים, ללא כפילויות, תוך ציון מדויק של כמה פעמים הופיע כל ערך באוסף המקורי.

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.