

תוכנה 1

תרגיל מספר 10

הנחיות כלליות:

- קראו בעיון את קובץ נוהלי הגשת התרגילים אשר נמצא באתר הקורס.
 - הגשת התרגיל תעשה במערכת ה VirtualTAU בלבד (<http://virtual2002.tau.ac.il/>).
1. יש להגיש קובץ zip יחיד הנושא את שם המשתמש (לדוגמא, עבור המשתמש zvainer יקרא הקובץ zvainer.zip) קובץ ה zip יכיל:
 - א. קובץ פרטים אישיים בשם details.txt המכיל את שמכם ומספר ת.ז. הזהות שלכם.
 - ב. קבצי ה java של התוכניות אותם התבקשתם לממש.
 - ג. קובץ טקסט עם העתק של כל קבצי ה java
 - ד. קובץ טקסט בשם answers עם התשובות לשאלות
-

שימו לב:

1. את כל המחלקות בתרגיל זה יש להציב בחבילה assignment10.
2. יש להקפיד על שמות המחלקות והפונקציות המוזכרים בתרגיל.
3. יש להגיש את קוד מחלקות הבדיקה בנוסף לקוד המימוש.
4. מומלץ בחום לקרוא היטב את כל התרגיל בטרם ניגשים לביצוע. במיוחד חלק ב'.

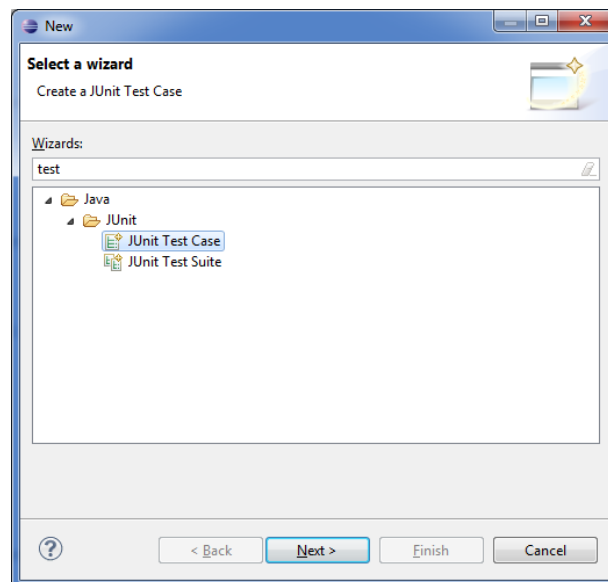
חלק א' – בדוק את עצמך (20 נק')

בחלק זה נבנה מחלקה הממירה מחרוזות המייצגות מספריים רומיים לשלמים ובחזרה. הנכם נדרשים להגיש גם את קוד הבדיקה (TestRomanConverter.java) וגם את מימוש המחלקה הממירה (RomanConverter.java).

המרה ממספר רומי למספר שלם. (15 נקודות)

1. נבנה מחלקת בדיקה ב-eclipse. נקרא למחלקה בשם TestRomanConverter.

ניתן להשתמש ב-CTRL+N ואז לבחור JUnit Test Case



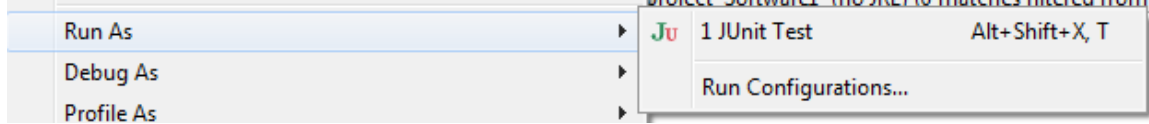
2. נתחיל מבדיקה עבור המקרה הפשוט ביותר בו המחרוזות מכילה רק ספרות של אחדות. הוסיפו את מתודת הבדיקה הבא:

```
@Test
public void testUnits() {
    Assert.assertEquals(1, RomanConverter.valueOf("I"));
    Assert.assertEquals(2, RomanConverter.valueOf("II"));
    Assert.assertEquals(3, RomanConverter.valueOf("III"));
}
```

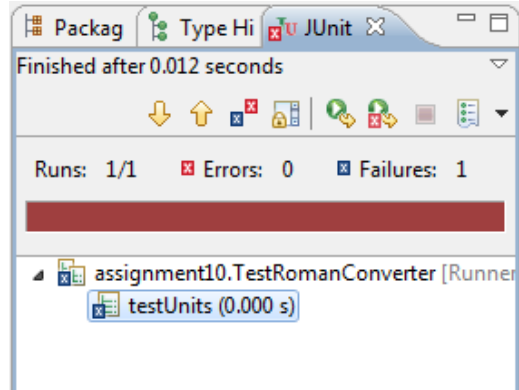
הערה: שימו לב לכך שאנו מסמנים שזו מתודת בדיקה ע"י הכיתוב @Test לפני חתימת המתודה.

3. תקנו את כל שגיאות הקומפילציה שנוצרו בעקבות יצירת מתודת הבדיקה.

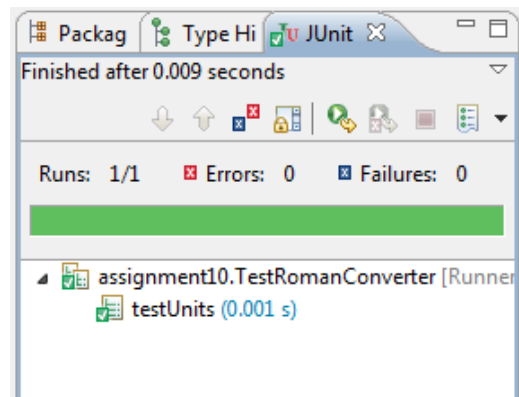
4. הריצו את תוכנת הבדיקה. הקפידו לבקש מ-eclipse להריץ את המחלקה כתוכנית בדיקה:



כצפוי, הרצת תוכנית הבדיקה תכשל שכן עדיין לא מימשנו את הלוגיקה הדרושה:



5. תקנו את המימוש כך שתוכנית הבדיקה תעבור בהצלחה.



6. כעת ניתן להתפנות ולסדר את קוד המימוש (אם צריך).

7. נרחיב את תוכנית הבדיקות כך שתכלול גם את הספרה 'V' המייצגת את המספר 5:

```
@Test
public void testFives() {
    Assert.assertEquals(5, RomanConverter.valueOf("V"));
    Assert.assertEquals(6, RomanConverter.valueOf("VI"));
    Assert.assertEquals(7, RomanConverter.valueOf("VII"));
    Assert.assertEquals(8, RomanConverter.valueOf("VIII"));
}
```

א. הוסיפו את מתודת הבדיקה.

ב. תקנו את הקוד כך שהריצה תעבור בהצלחה.

ג. במידת הצורך, שפרו את המימוש הראשוני והריצו שוב את תוכנית הבדיקה.

8. הוסיפו פונקציית בדיקה חדשה הבודקת את המקרה בו הספרה 'I' נמצאת לפני הספרה 'V' כך שהערך המתקבל הוא בעצם 4:

```
@Test
public void testFour() {
    Assert.assertEquals(4, RomanConverter.valueOf("IV"));
}
```

בצעו את התיקונים הנדרשים עמ"נ לחזור למצב "ירוק".

9. הרחיבו את תוכנית הבדיקה כך שתתמוך גם בספרת העשרות ('X'):

```
@Test
public void testTens() {
    Assert.assertEquals(9, RomanConverter.valueOf("IX"));
    Assert.assertEquals(10, RomanConverter.valueOf("X"));
    Assert.assertEquals(11, RomanConverter.valueOf("XI"));
    Assert.assertEquals(14, RomanConverter.valueOf("XIV"));
    Assert.assertEquals(15, RomanConverter.valueOf("XV"));
    Assert.assertEquals(19, RomanConverter.valueOf("XIX"));
    Assert.assertEquals(20, RomanConverter.valueOf("XX"));
}
```

10. הוסיפו גם בדיקות עבור ספרות גבוהות יותר. להזכירכם: 'L' מייצג 50, 'C' מאות, 'D' מייצג 500 ו-'M' מייצג אלפים.

```
@Test
public void testSomeMore() {
    Assert.assertEquals(50, RomanConverter.valueOf("L"));
    Assert.assertEquals(59, RomanConverter.valueOf("LIX"));
    Assert.assertEquals(87, RomanConverter.valueOf("LXXXVII"));
    Assert.assertEquals(44, RomanConverter.valueOf("XLIV"));
    Assert.assertEquals(369, RomanConverter.valueOf("CCCLXIX"));
    Assert.assertEquals(2751, RomanConverter.valueOf("MMDCCCLI"));
}
```

המרה ממספר שלם למספר רומי (5 נקודות)

11. הוסיפו את הפונקציה `toRomanString` המבצעת את ההמרה ההפוכה. הוסיפו פונקציית בדיקה ולאחר מכן תקנו את המימוש בהתאם:

```
@Test
public void testRomans() {
    Assert.assertEquals("I", RomanConverter.toRomanString(1));
    Assert.assertEquals("II", RomanConverter.toRomanString(2));
    Assert.assertEquals("VI", RomanConverter.toRomanString(6));
    Assert.assertEquals("XIV", RomanConverter.toRomanString(14));
    Assert.assertEquals("XXIV", RomanConverter.toRomanString(24));
    Assert.assertEquals("XLIV", RomanConverter.toRomanString(44));
}
```

חלק ב' – עצים (35 נקודות)

בחלק זה נממש עץ מסוג Trie. בעץ זה משתמשים עמ"נ להחזיק מחרוזות בצורה יעילה.

פרטים נוספים על מבנה נתונים זה ניתן למצוא ב:

<http://en.wikipedia.org/wiki/Trie>

בכל אחד מהסעיפים הבאים, כתבו את תוכנית הבדיקה ולאחר מכן ממשו כך שתוכנית הבדיקה תרוץ בהצלחה. לבסוף בדקו האם ניתן לבצע תהליך refactor עמ"נ לשמור על קוד "נכון" יותר.

1. נתחיל ביצירת תוכנית בדיקה. צרו את המחלקה TestTrieTree והגדירו את מתודת הבדיקה הבאה:

```
@Test
public void testCreate() {
    Collection<String> t = new TrieTree();

    Assert.assertEquals(t.isEmpty(), true);
    Assert.assertEquals(t.size(), 0);
}
```

תקנו את שגיאות הקומפילציה וכן את שגיאות זמן הריצה.

2. (8 נקודות) נרצה להוסיף אפשרות להכנסת מחרוזות חדשות לעץ. נגדיר את הפונקציונליות הנדרשת ע"י מתודת בדיקה:

```
@Test
public void testElementAddition() {
    Collection<String> t = new TrieTree();

    t.add("hello");

    Assert.assertEquals(t.size(), 1);
    Assert.assertEquals(t.isEmpty(), false);

    t.add("world");

    Assert.assertEquals(t.size(), 2);

    t.add("world");

    Assert.assertEquals(t.size(), 2);

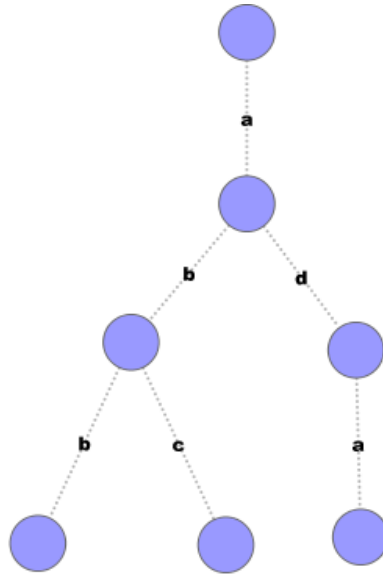
    t.add("worl");

    Assert.assertEquals(t.size(), 3);
}
```

שימו לב, שהעץ אינו תומך בערכים כפולים: כאשר הספנו את המחרוזת "world" פעם נוספת, גודל העץ לא השתנה.

כיצד בנוי עץ Trie? כיצד מכניסים אליו ערכים? כל מחרוזת תיוצג על ידי מסלול מהשורש אל אחד הצמתים הפנימיים. לכן, מספר הבנים של צומת נתון עלול להיות כגודל הא"ב אותו אנו מייצגים. לשם פשטות נניח שכל המחרוזות שלנו הן מעל האותיות a-z, כלומר, אותיות לטיניות קטנות בלבד.

לדוגמא, העץ הבא מכיל את המחרוזות "abc", "ada" ו-"abb":



כיצד נדע שהעץ איננו מכיל גם את המחרוזות "ab" או "ad"? בתצורה זו איננו יודעים. הפתרון: "נסמן" בכל צומת האם מסתיימת בה מילה או לא.

כדי להכניס מילה לעץ נתקדם לאורך המסלול המתאים בעץ. אם הגענו לצומת פנימית נסמן אותה שמסתיימת בה מילה. אם הגענו לקצה העץ, כלומר לצומת אשר אין לו בן לאורך קשת מתאימה, אז נבנה את המשך העץ כשרשרת.

לדוגמא, נניח כי בעץ הקודם היו המחרוזות "abc" ו-"abb". לאחר מכן רוצים להוסיף את המחרוזת "ada". עבור ה-"a" הראשון יש קשת ולכן נותר להוסיף את המחרוזת "da" לתת העץ ששורשו בבן התלוי על קשת זו. (בדוגמא שלנו יש רק בו יחיד לשורש העץ). כעת, נרצה ללכת לאורך קשת בעלת תווית "d". אולם, אין קשת כזו בצומת אליו הגענו. לכן, נוסיף צומת חדש, ונצביע אליו עם קשת המסומנת ב-"d". בצורה דומה, נוסיף את תחתיו צומת התלוי על קשת המסומנת ב-"a".

3. (8 נקודות) נרצה לתמוך באפשרות לעבור על כל המילים שאוחסנו בעץ. נכתוב את מתודת הבדיקה הבאה:

```
@Test
public void testIteration() {
    String[] values = {"course", "hello", "software", "world" };
    List<String> permutation = Arrays.asList(values.clone());
    Collections.shuffle(permutation);

    Collection<String> t = new TrieTree();

    for(String word : permutation)
        t.add(word);

    int j =0;
    for(String word : t) {
        Assert.assertEquals(values[j++], word);
    }
}
```

נשים לב לכך שהמחרוזות חוזרות בסדר לקסיקוגרפי. בנוסף, חוזרות רק מילים שהכנסנו לעץ ללא קשר למבנה הפנימי שאנו מחזיקים.

כדי לעמוד בבדיקה זו אנו נדרשים לממש את הפונקציה `iterator()`. פונקציה זו מחזירה אובייקט המממש את הממשק `Iterator<String>` המאפשר מעבר על רשימות מילים.

האובייקט יצטרך לסרוק את העץ, כלומר לעבור על כל הצמתים וכאשר מגיעים לצומת מסומן יודעים כי נמצאה המילה הבאה. כדי להחזיר את המילים בסדר לקסיקוגרפי, יש לרדת לאורך קשתות בסדר זה. כלומר, כאשר סורקים את העץ ונמצאים בצומת מסויים, יורדים קודם כל לאורך קשת 'a' (אם קיימת), לאחר מכן לאורך הקשת המסומנת ב-'b' וכו'.

רמז: רצוי להשתמש במחלקה פנימית (ראו הרצאה 7) כדי לממש איטרטורים.

4. 8 נקודות) נרצה לתמוך בשאילתה הבודקת אם מילה מסויימת נמצאת בעץ. נכתוב את תוכנית הבדיקה הבאה:

```
@Test
public void testContains() {
    Collection<String> t = new TrieTree();

    t.add("europe");
    t.add("africa");
    t.add("asia");
    t.add("america");
    t.add("australia");

    Assert.assertEquals(t.contains("europe"), true);
    Assert.assertEquals(t.contains("china"), false);
}
```

השלימו את המימוש כנדרש.

5. 11 נקודות) נרצה לממש חיפוש יעיל, כדוגמת אלו המצויים בתיבות טקסט בעלות השלמה (AutoComplete). בתיבות כאלה, כאשר המשתמש מכניס חלק מהמילה, מוצגות בפניו כל המילים המתחילות במחרוזת אותה הוא הקליד.

נוסיף, אם כן, את האפשרות לעבור על כל המחרוזות בעץ המתחילות ברישא (prefix) נתונה.

נכתוב תוכנית בדיקה:

```

@Test
public void testEnumWords() {
    TrieTree t = new TrieTree();

    t.add("anemia");
    t.add("anomaly");
    t.add("anagram");
    t.add("anorexia");
    t.add("anathema");
    t.add("anthrax");
    t.add("anecdote");

    boolean anomaly = false;
    boolean anorexia = false;

    for(String word : t.enumerateWords("ano"))
        if(word.equals("anomaly"))
            anomaly = true;
        else if (word.equals("anorexia"))
            anorexia = true;
        else
            Assert.fail();

    Assert.assertEquals(anomaly && anorexia, true);

    boolean anemia = false;
    boolean anecdote = false;
    for(String word : t.enumerateWords("ane"))
        if(word.equals("anemia"))
            anemia = true;
        else if (word.equals("anecdote"))
            anecdote = true;
        else
            Assert.fail();

    Assert.assertEquals(anecdote && anemia, true);
}

```

שימו לב לכך שחיפוש כל המילים המתחילות ברישא (prefix) נתונה שקול למעבר על כל המילים הנמצאות בתת העץ ששורשו הוא בצומת אליו מגיעים כאשר מבצעים הילוך מהשורש לפי הרישא הנתונה. בנוסף, ראינו כבר כיצד לעבור על כל המילים (בכל העץ).

הערה: על הערך המחוזר מ-`enumerateWords` לממש את הממשק `Iterable<String>` כדי שנוכל להשתמש בו בלולאת `for-each`.

חלק ג' – מרוב עצים... (45 נק')

בחלק זה נממש Collection גנרי ע"י עץ חיפוש בינארי. שימו לב, לא נקפיד כי עץ החיפוש יהיה מאוזן. בנוסף, לשם פשטות אינכם נדרשים לספק מימוש חוקי לכל המתודות בממשק Collection, אלא רק לאלו המוזכרות בתרגיל.

פרטים נוספים על חיפוש בינארי ניתן למצוא בקישור:

http://en.wikipedia.org/wiki/Binary_search_tree

1. נתחיל, כמובן, בתוכנית בדיקה. צרו את המחלקה TestTree. הוסיפו את מתודת הבדיקה הבאה:

```
@Test
public void testCreate() {
    Collection<Integer> t = new Tree<Integer>();

    Assert.assertEquals(t.isEmpty(), true);
    Assert.assertEquals(t.size(), 0);
}
```

הביאו את הקוד למצב בו תוכנית הבדיקה רצה ללא שגיאות.

2. (7 נק') נרצה ליצוק תוכן למבנה הנתונים שלנו. נתחיל בהוספת אלמנטים למבנה הנתונים. נרצה לעמוד בתוכנית הבדיקה הבאה:

```
@Test
public void testElementAddition() {
    Collection<Integer> t = new Tree<Integer>();

    t.add(4);

    Assert.assertEquals(t.size(), 1);
    Assert.assertEquals(t.isEmpty(), false);

    t.add(5);

    Assert.assertEquals(t.size(), 2);

    t.add(4);

    Assert.assertEquals(t.size(), 3);
}
```

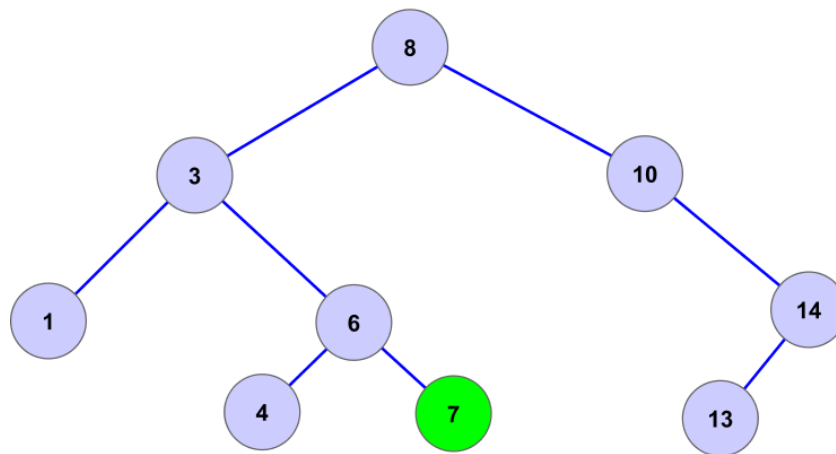
שימו לב, מלאו תחילה את הקוד ההכרחי לעמידה בתוכנית הבדיקה ורק לאחר מכן ממשו את מבנה הנתונים.

להזכרכם, עץ חיפוש בינארי הוא עץ בו כל צומת מקיים את התכונות הבאות:

- א. לכל צומת יש ערך ולכל היותר שני תתי עצים.
- ב. לכל הצמתים בתת העץ השמאלי יש ערכים קטנים (או שווים) מזה שבצומת הנוכחי.
- ג. לכל הצמתים בתת העץ הימני יש ערכים גדולים יותר מזה שבצומת הנוכחי.

כדי להוסיף ערך לעץ קיים נתקדם במורד העץ, לאורך המסלול בו היינו מצפים למצוא את הערך המוכנס. כאשר מגיעים לקצה העץ, מוסיפים את הערך כעלה במקום זה.

לדוגמא נניח שאנו רוצים להכניס את הערך 7 לעץ הנתון, אזי 7 קטן משמונה, לכן נרד לתת העץ השמאלי, 7 גדול מ-3 ולכן נמשיך לתת העץ הימני. 7 גדול מ-6 ולכן היינו רוצים להמשיך לבן הימני של 6, אולם מכיוון שאין בן כזה, נוסיף את 7 כבן הימני.



הערה: אנו ממשים מבנה נתונים גנרי, כלומר המבנה יכול להחזיק נתונים מסוג כלשהו. נניח כי הטיפוס הגנרי מממש את הממשק $\text{Comparable}\langle E \rangle$ כך שניתן להשוות בין הנתונים המוחזקים בעץ.

3. (7 נק') כעת נוסיף גם אפשרות למחיקת נתונים ממבנה הנתונים שלנו :

```
@Test
public void testRemovalAndClear() {
    Collection<Integer> t = new Tree<Integer>();

    t.add(4);
    t.add(5);

    t.remove(4);

    Assert.assertEquals(t.size(), 1);

    t.remove(4);

    Assert.assertEquals(t.size(), 1);

    t.add(10);
    t.add(11);
    t.add(12);
    t.add(13);

    Assert.assertEquals(t.size(), 5);

    List<Integer> toRemove = Arrays.asList(new Integer[]{11, 12, 5, 4000});

    t.removeAll(toRemove);

    Assert.assertEquals(t.size(), 2);

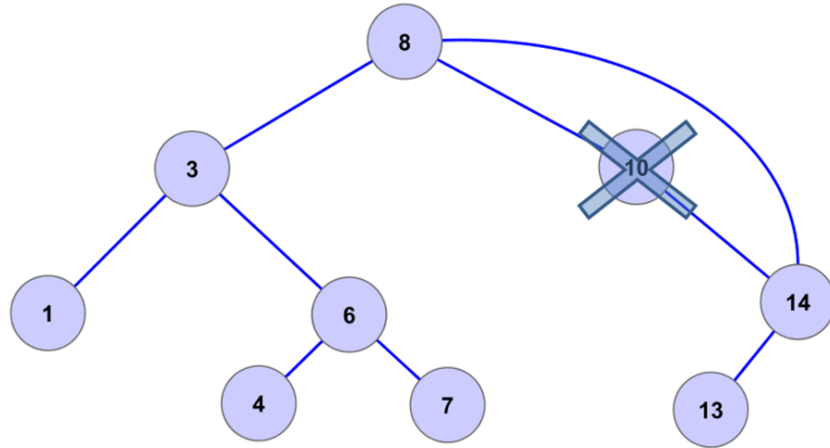
    t.clear();

    Assert.assertEquals(t.size(), 0);
}
```

כיצד מוחקים מעץ חיפוש?

המקרים הפשוטים יותר הם כאשר לצומת אותו מבקשים למחוק הוא עלה או בעל בן יחיד :

- במקרה הראשון ניתן למחוק את העלה ללא שינוי נוסף.
- במקרה השני יש להחליף את הצומת הנוכחי בבנו (בציוור מודגם מחיקת הערך 10 עפ"י מקרה זה).



המקרה המסובך יותר הוא כאשר יש לצומת שני בנים. במקרה זה נמצא את הצומת בעץ המכיל את הערך העוקב לצומת זה. נרשום בצומת הנוכחי את ערכו של העוקב ונמחק את הצומת העוקב עפ"י הכללים שהוזכרו לעיל.

- כיצד מוצאים את העוקב? זהו הצומת בעל הערך הקטן ביותר בתת העץ הימני.
- מהו הצומת בעל הערך הקטן ביותר בתת-עץ נתון? זה הבן השמאלי ביותר בתת-העץ.

הערות:

- את removeAll ניתן לממש ע"י הסרת האיברים הנתונים אחד אחרי השני.
- את clear ניתן לממש ע"י איפוס מבנה הנתונים.
- המתודה remove של הממשק Collection<E> מקבלת כארגומנט עצם מטיפוס Object. מותר להמיר את הארגומנט לטיפוס אותו אנו מחזיקים:

```
public boolean remove(Object o) {
    E item = (E)o;
    ...
}
```

במקרה זה נקבל אזהרה מהקומפיילר. מותר להתעלם מאזהרה זאת. נסמן זאת ע"י האנוטציה @SupperssWarnings, כפי שמציע Eclipse:

```
public boolean remove(Object o) {
    @SuppressWarnings("unchecked")
    E item = (E)o;
    ...
}
```

4. (7 נק') נמשיך במימוש. נוסיף אפשרות לעבור על האיברים המוחזקים בעץ. נרשום את מתודת הבדיקה הבאה:

```
@Test
public void testIteration() {
    Integer[] values = {1,2,3,4,5,6,7,8,9,10,50,60,70,80,90};
    List<Integer> permutation = Arrays.asList(values.clone());
    Collections.shuffle(permutation);

    Collection<Integer> t = new Tree<Integer>();

    for(Integer i : permutation)
        t.add(i);

    int j =0;
    for(Integer i : t)
        Assert.assertEquals(values[j++], i);
}
```

שימו לב, כאשר עוברים על האיברים יש להחזיר אותם בסדר עולה, ללא קשר לסדר ההכנסה. כיצד נעשה זאת? אנו כבר יודעים למצוא את האיבר הקטן ביותר בעץ, מהו האיבר הבא אחריו?

אם יש לצומת הנוכחי בן ימני, אז אנו כבר יודעים מיהו העוקב. אחרת, העוקב נמצא במעלה העץ: יש לעלות במעלה העץ כל עוד מתקדמים לאורך קשתות מאב לבן ימני, ואז להחזיר את האב של הצומת המתקבל. לדוגמא: מיהו העוקב של 7 בצירור? עולים ל-6 דרך קשת של בן ימני. ממשיכים בצורה דומה ל-3. כעת 3 הוא בן שמאלי ולכן נחזיר את צומת האב של 3, שהוא 8. ואכן, 8 הוא העוקב של 7 בעץ.

הילוך זה על מבנה העץ נקרא הילוך inorder, מכיוון שהוא שקול להילוך המוגדר באופן רקורסיבי כך (הטיפול בערך הנוכחי מתבצע בים הטיפול בענף הימני לשמאלי):

- בצע רקורסיה על תת העץ השמאלי.
- טפל בערך בצומת הנוכחי.
- בצע רקורסיה על תת העץ הימני.

רמז: יש לממש את המתודה `iterator()` כדי לעמוד בבדיקה זו.

5. (3 נק') נממש את המתודה contains הבודקת האם מבנה הנתונים מכיל ערך נתון. נרצה לעמוד במתודת הבדיקה הבאה:

```
@Test
public void testContains() {
    Collection<Integer> t = new Tree<Integer>();

    t.add(3);
    t.add(10);
    t.add(15);
    t.add(4);

    Assert.assertEquals(t.contains(4), true);
    Assert.assertEquals(t.contains(new Integer(4)), true);
    Assert.assertEquals(t.contains(13), false);
    Assert.assertEquals(t.contains(20), false);

    t.remove(4);

    Assert.assertEquals(t.contains(4), false);
    Assert.assertEquals(t.contains(15), true);

    List<Integer> other = Arrays.asList(new Integer[]{10,15,3});
    Assert.assertEquals(t.containsAll(other), true);
}
```

הערה: נממש את containsAll ע"י בדיקת contains של כל אחד מהאיברים הנתונים בנפרד.

6. (3 נק') נמשיך לעבות את המימוש. השתמשו במתודת הבדיקה הבאה. בצעו את השנויים הנדרשים כך בתוכנית הבדיקה תעבור ללא שגיאות.

```
@Test
public void testRetains() {
    Collection<Integer> t = new Tree<Integer>();

    t.add(1);
    t.add(2);
    t.add(3);
    t.add(10);

    List<Integer> keep = Arrays.asList(new Integer[]{10,15,3});

    t.retainAll(keep);

    Assert.assertEquals(t.size(), 2);
}
```

7. (7 נק') נרצה לאפשר סידור כלשהו של האיברים בעץ. על מנת לאפשר זאת, נוסיף בנאי המקבל מימוש של הממשק Comparator אשר ישמש להשוואה בין האיברים.
נגדיר את הבדיקה הבאה:

```
@Test
public void testComparison() {
    Integer[] values = {1,2,3,4,5,6,7,8,9,10,50,60,70,80,90};
    List<Integer> permutation = Arrays.asList(values.clone());
    Collections.shuffle(permutation);

    Collection<Integer> t = new Tree<Integer>(new ReverseIntegerCompare());

    for(Integer i : permutation)
        t.add(i);

    int j = values.length -1 ;
    for(Integer i : t)
        Assert.assertEquals(values[j--], i);
}
```

שימו לב:

א. יש להשלים את המימוש של ReverseIntegerCompare עפ"י קטע הקוד הבא:

```
public class ReverseIntegerCompare implements Comparator<Integer> {
}
}
```

מחלקה זאת ממשת השוואה הפוכה, כך שהסדר של האיברים בעץ יהיה הפוך, כלומר כאשר נשתמש נעבור על הערכים בעץ נקבל אותם מהגדול לקטן.

ב. יש לוודא שתוכנית הבדיקה רצה כנדרש.

8. (8 נק') בדומה להגדרת הילוך inorder שראינו קודם, ניתן להגדיר הילוכים נוספים על העץ. נוסיף תמיכה גם בהילוך preorder. המוגדר על ידי ההגדרה הרקורסיבית הבאה:

- טפל בצומת הנוכחי
- בצע רקורסיה על תת עץ שמאל
- בצע רקורסיה על תת עץ ימין

את המימוש נבדוק ע"י מתודת הבדיקה הבאה:

```
@Test
public void testPreorder() {
    Integer[] values = {3,4,5,6,1,9,10,50,60,70,2,7,8,80,90};

    Tree<Integer> t = new Tree<Integer>();

    for(Integer i : values)
        t.add(i);

    Integer[] preorder = {3,1,2,4,5,6,9,7,8,10,50,60,70,80,90};

    int j = 0;
    for(Integer i : t.preOrder())
        Assert.assertEquals(preorder[j++], i);
}
```

תזכורת! הפונקציה `preOrder()` צריכה להחזיר מחלקה הממשת את הממשק של `Iterable<E>` של Java כדי שנוכל להשתמש בערך החוזר בלולאת `for-each`.

9. (3 נק') הוסיפו מתודת בדיקה נוספת, `testPreorder2`, הבודקת את הילוך `preorder` עבור העץ בדוגמא. התוצאה צריכה להיות (עפ"י הסדר):

8, 3, 1, 6, 4, 7, 10, 14, 13

ב ה צ ל ח ה !