פרק 9

עץ בינרי

הרשימה שהכרנו משמשת לטיפול בסדרות. בסדרה, לכל איבר (פרט לראשון) קיים איבר קודם, ולכל איבר (פרט לאחרון), קיים איבר עוקב. יש מצבים שבהם הקשרים בין האיברים מורכבים יותר. בפרק זה נכיר ארגון נתונים חדש, בשם עץ. הנתונים בעץ מאורגנים בצורה היררכית ולא בצורה סדורה כרשימה. בהמשך הפרק נציע מבנה נתונים עבור צורת ארגון זו.

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

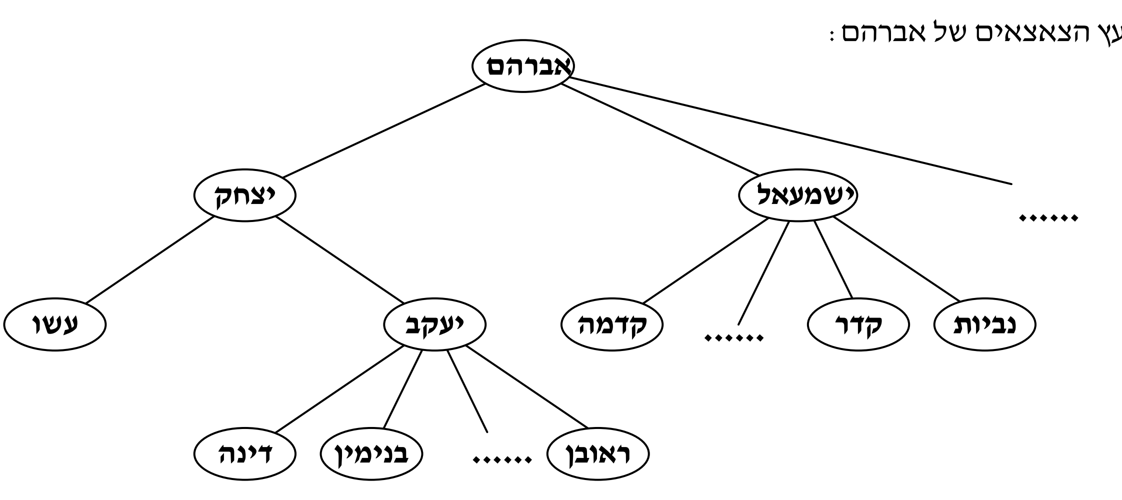
נביות	קדר	קדמה	יעקוב	עשו	ישמעאל	יצחק	אברהם

יש לנו ייצוג של בני המשפחה, אך מה לגבי קשרי המשפחה? יצחק וישמעאל הם הבנים של אברהם, עשו ויעקב הם הבנים של יצחק, קדמה, 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 אל עברו של האדם: הוריו, סביו, רב-סביו וכו'.

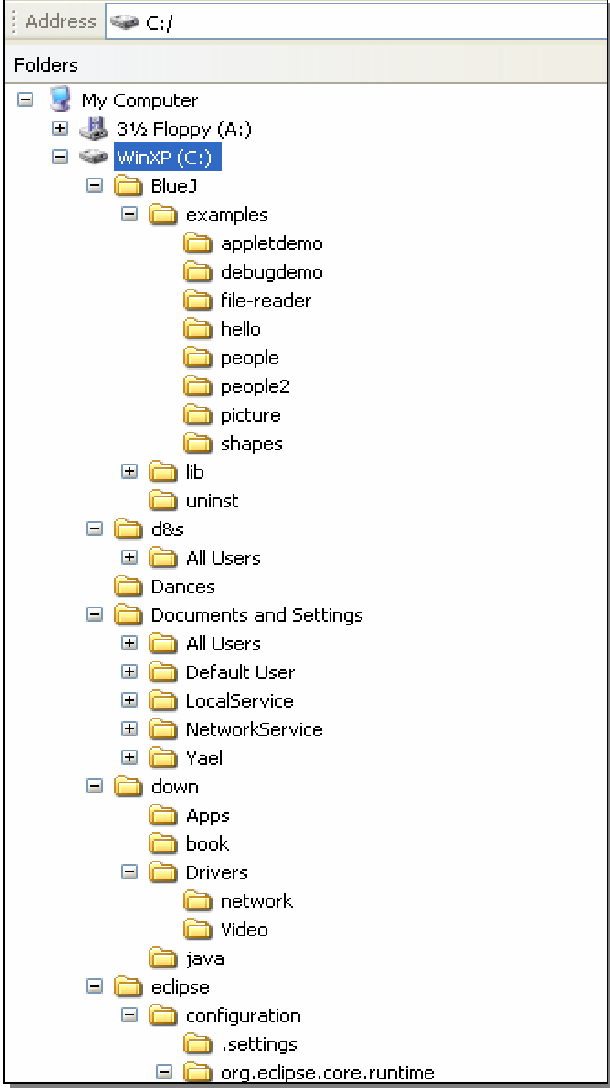
המשפחה של אברהם אבינו, כעץ צאצאים, מתוארת באיור הבא. זהו כמובן רק חלק קטן מהעץ המתאים למשפחה ענפה זו, ואפשר להמשיך ולצייר בו ענפים נוספים.

עץ הצאצאים של אברהם:



© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

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



א. עצים

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 וספריות 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.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

קשרים מהסוג השני הם בין צומת ל**ילדיו (children)**. צמתים שהם ילדים לאותו הורה נקראים, כצפוי, **אחים (siblings)**.

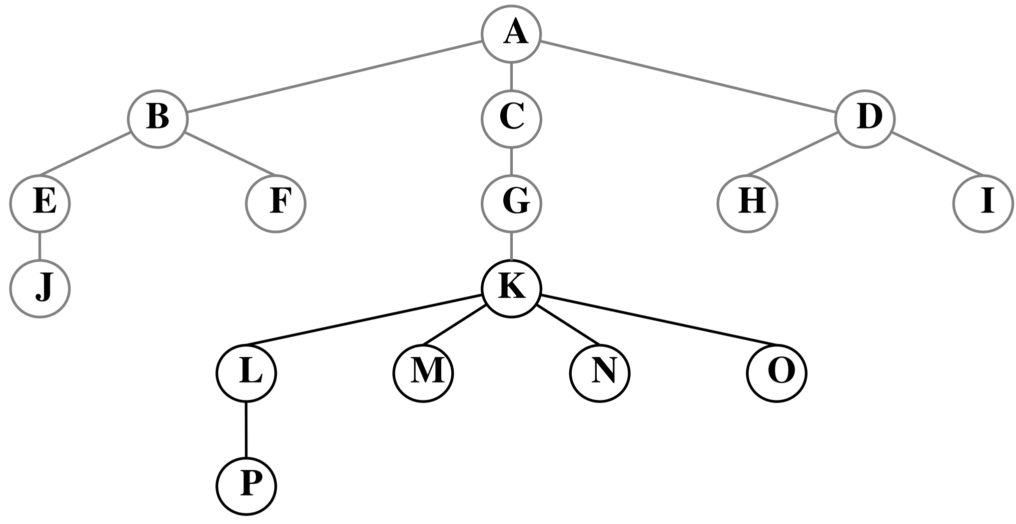
הילדים של צומת בעץ צאצאים, הם ילדיו הביולוגים של האדם המיוצג בצומת. בעץ מערכת הקבצים, לעומת זאת, ילדיה של ספריה המופיעה בצומת, הם הקבצים והספריות שהיא מכילה.

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 הוא **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 הוא צאצא של השורש.

? לכל אחד משלושת התנאים, הצג מבנה שאינו מקיים אותו, אך מקיים את שני התנאים האחרים.

צומת וצאצאיו הם עץ בפני עצמו ; עץ זה הוא **תת-עץ (sub-tree)** של העץ המקורי.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

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

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

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

בעצים שבדוגמאות עד כאן, אין הגבלה על מספר הילדים של צומת. ביישומים רבים עוסקים בעצים שבהם מספר הילדים מוגבל, למשל לכל צומת יש לכל היותר שני ילדים, או בדיוק שני ילדים. בהמשך הפרק נעסוק רק בעצים כאלו.

ב. עץ בינרי

נצמצם את מרחב העצים בו אנו מטפלים ונתמקד בסוג מסוים בלבד. נגדיר **עץ בינרי** (**Binary Tree**) כעץ בו יש לכל היותר שני ילדים לצומת. מקובל להתייחס אליהם כילד שמאלי וילד ימני. כל אחד משני אלו, אם הוא קיים, הוא שורש של עץ. הילד השמאלי הוא שורשו של עץ הנקרא **תת עץ שמאלי** (left sub-tree) של הצומת, והילד הימני הוא שורשו של עץ הנקרא **תת עץ ימני** (right sub-tree). גם כאן כמו בעץ הכללי, לכל צומת אין יותר מהורה אחד (לשורש העץ כולו אין הורה בכלל).

נגדיר עץ בינרי כך:

עץ בינרי הוא **צומת** עם **שני תת עצים לכל היותר**, כאשר:

- כל תת עץ הוא עץ בינרי
- שני תת עצים בינריים אלו זרים זה לזה, כלומר אין להם צמתים משותפים.

הגדרה זו היא הגדרה רקורסיבית שבסיסה הוא: עץ עלה.

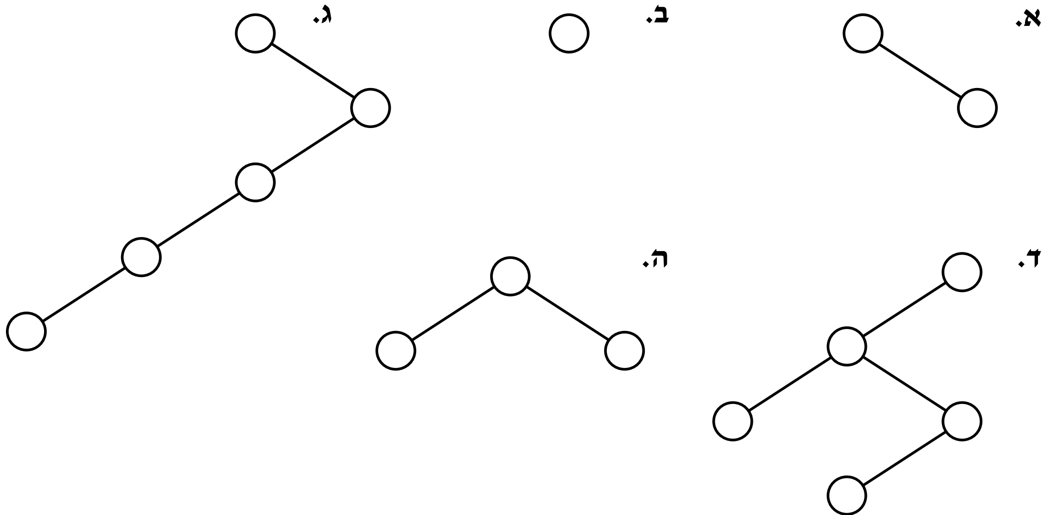
על פי הגדרה זו, אין עץ בינרי ריק. ניתן להגדיר את המושג אחרת כך שעץ בינרי יוכל להיות ריק, אך ההגדרה שבחרנו נוחה יותר לדיונינו בהמשך הפרק.

? אם שני התת עצים של צומת אינם זרים ויש להם צומת משותף, נסתרת ההנחה הבסיסית בעץ, לפיה אין לצומת שני הורים (או יותר מכך). הסבר.

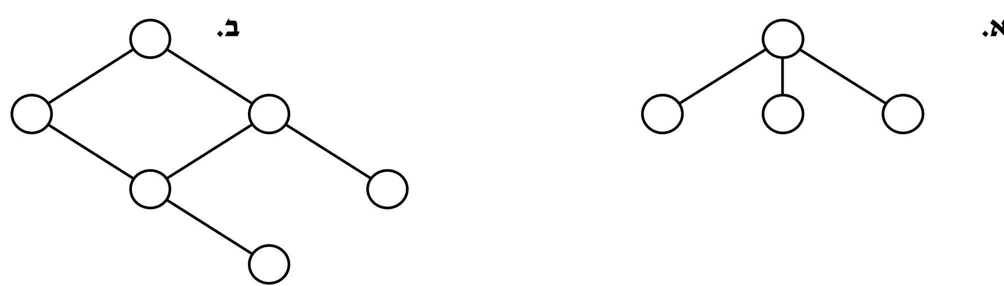
באיור הבא מופיעות מספר דוגמאות לעצים בינריים, ובאיור שאחריו מופיעות שתי דוגמאות לעצמים שאינם עצים בינריים: באחת יש לשורש יותר משני בנים, ובאחרת יש איבר שלו יותר מהורה אחד, ולכן אין זה עץ כלל.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

דוגמאות לעצים בינריים:



דוגמאות לעצמים שאינם עצים בינריים:



מעתה, נדבר רק על **עצים בינריים**, והמונח **עץ** יתפרש תמיד כ**עץ בינרי**.

ב.1. ייצוג העץ הבינרי

בפרק זה נראה ייצוג אפשרי אחד לעץ בינרי, במחלקה שתקרא BinTree. בדומה לייצוג הרשימה בו בחרנו, נאמץ גם כאן את רעיון השימוש בהפניות ונציג כל צומת בעץ על ידי עצם מטיפוס BinTree המורכב משלוש תכונות: תכונה המכילה את המידע השמור בצומת ושתי תכונות שהן הפניות לעצמים נוספים מטיפוס BinTree. כאשר ערך הפניה כזו הוא **null**, פירוש הדבר שהתת עץ המתאים אינו קיים. בניגוד לייצוג הרשימה בו בחרנו במעטפת שתנהל כראוי את אוסף האיברים, כאן אנו בוחרים שלא לעטוף את אוסף הצמתים בעזרת מחלקה נוספת. (משום כך, כמוסבר להלן, במבנה זה לא ניתן לייצג עץ ריק.)

ניתן להגדיר את המחלקה BinTree עבור כל טיפוס של מידע, כלומר באופן גנרי, כפי שנהגנו באוספים הקודמים שהכרנו (מחסנית, תור, רשימה). אולם הפעם אנו מרבים להשתמש בערכים השמורים באוסף ולכן לצורך הנוחות והפשטות בחרנו להגדיר את העץ כעץ של שלמים. כמובן

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

שניתן לשנות את הגדרת תכונת המידע השמור בעץ ולאפשר הגדרת עצים בינריים עם טיפוסי מידע שונים.

איור ה-UML הבא מתאר את ייצוג העץ הבינרי בו בחרנו:

BinTree
private BinTree left **private int** info **private** BinTree right
יושלם בהמשך...

עץ ריק

בייצוג בו בחרנו אין משמעות וקיום למושג: "עץ ריק". ברגע היצירה העץ כבר מכיל צומת. הגדרה זו שונה מהגדרת הרשימה למשל, שם ניתן היה ליצור רשימה ריקה. ההבדל נובע מהגדרת מבנה הנתונים. הרשימה הוגדרה כמעטפת חיצונית המנהלת אוסף חוליות שהוא רכיב שלה. האוסף היה יכול להיות ריק ועדיין היה קיום לעטיפה. לעומת זאת העץ הוא מבנה מקושר שמחזיק את כל הצמתים שבו באמצעות הפניות. אין לעץ מעטפת חיצונית. הוצאת כל הצמתים מהעץ מעלימה את העץ עצמו ואין משמעות למצב שבו מתקיים "עץ ריק".

ב.2. ממשק העץ הבינרי

כמו לכל מחלקה יש להגדיר פעולות בונות מתאימות כך שנוכל לייצר עצמים מתבנית המחלקה. הפעולה הבונה הכללית תקבל פריט מידע ושני תת עצים שישמשו כילדיו של העץ החדש. לשם הנוחיות נגדיר גם פעולה הבונה עץ עלה, שלו מידע בלבד ואין לו ילדים.

לכל אחת מהתכונות שנבחרו כדי לייצג את העץ, יש להגדיר פעולות: get ו-set מתאימות. פעולות getInfo, setInfo מאפשרות לאחזר את המידע שבצומת ולשנותו. פעולות getLeft, setLeft מאפשרות לאחזר ולשנות את התת עץ השמאלי. שתי פעולות דומות קיימות עבור התת עץ הימני. נשים לב כי הפעלת setLeft (tree) משמשת כפעולת מחיקה של התת-עץ שהיה במקום הזה ובו בזמן משמשת להוספת תת-עץ חדש (עלה כמקרה פרטי) באותו מקום. כאשר הפרמטר המועבר לפעולת set הוא **null**, הפעולה מוחקת למעשה את התת-עץ המתאים ושמה **null** במקומו. שימו לב כי לאחר ההחלפה, במצב בו אין הפניה נוספת לעץ המוחלף, "יאבד" תת עץ זה. לכן ככלל יש לבצע את פעולות ההחלפה רק כאשר אין צורך בתת-עץ המקורי, או, אם יש צורך בתת עץ זה, נשמור הפניה אליו במשתנה.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

עד כה דנו בפעולות מקומיות על צומת, אך עץ הוא אוסף של צמתים מחוברים ביניהם בעזרת ההפניות שבכל צומת. יש פעולות "גלובליות" שנרצה להגדיר, הפועלות על העץ כולו, כלומר על אוסף כל הצמתים המאוגדים בעץ. הפעולה toString היא פעולה גלובלית שכזו, הפועלת על אוסף הצמתים כולו. בשונה ממנהגנו עד כה, לא נגדיר את הפעולה toString() של העץ. מכיוון שעץ אינו מבנה סדרתי, קיימת יותר מדרך אחת לסרוק אותו ולהחזיר תיאור של איבריו. על הדרכים השונות של סריקת העץ נדון בהמשך הפרק.

פעולות נוספות שלא יופיעו בממשק העץ הן פעולות הכנסה והוצאה של צמתים בעץ, זאת בניגוד לממשק הרשימה, שבו הופיעו פעולות כאלו. ברשימה ברור לגמרי מה פירוש להכניס איבר חדש אחרי איבר קיים. האיבר החדש ייכנס אחרי אותו איבר ולפני העוקב לו. בעץ, כיוון שיש שני תת עצים, מתבקשות שתי פעולות "הכנסה אחרי": כילד שמאלי או כילד ימני של צומת נתון. אך, בניגוד למצב ברשימה, בעץ לא קיימת הגדרה טבעית-ברורה של פעולות הכנסה אלו. אם נרצה להכניס צומת חדש מתחת ומשמאל לצומת קיים, האם התת-עץ השמאלי שהיה שם קודם לכן, יהיה עכשיו ילד ימני או שמאלי של הצומת החדש? הוא הדין לגבי הגדרת פעולות הוצאה מעץ. לכן איננו כוללים בממשק פעולות הכנסה והוצאה. בפרק על עץ חיפוש בינרי נדון בקבוצת עצים מסוג מסוים עבורה נגדיר פעולת הכנסה הטבעית לה.

לסיכום, הממשק המוצג להלן מכיל רק את הפעולות הבסיסיות, המקומיות, שנדונו לעיל.

להלן איור ה-UML המלא המתאר את מבנה הנתונים עץ בינרי:

BinTree
private BinTree left **private int** info **private** BinTree right
BinTree (**int** info) BinTree (**int** info, BinTree left, BinTree right)
int getInfo() **void** setInfo (**int** info) BinTree getLeft() BinTree getRight() **void** setLeft (BinTree tree) **void** setRight (BinTree tree)

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

להלן ממשק המחלקה המלא:

הפעולה בונה עץ עלה שבשורשו הערך info	BinTree (**int** info)
הפעולה בונה עץ שבשורשו הערך info, התת-עץ השמאלי שלו left והתת-עץ הימני right. כל אחד משני הפרמטרים האחרונים יכול להיות **null**, ואז התת-עץ המתאים אינו קיים. שני פרמטרים **null** מגדירים עץ עלה. **הנחה:** left ו-right זרים זה לזה	BinTree (**int** info, BinTree left, BinTree right)
הפעולה מחזירה את ערך השורש	**int** getInfo()
הפעולה משנה את ערך השורש להיות info	**void** setInfo (**int** info)
הפעולה מחזירה את התת עץ השמאלי. אם אין תת עץ שמאלי, הפעולה תחזיר **null**	BinTree getLeft()
הפעולה מחזירה את התת עץ הימני. אם אין תת עץ ימני, הפעולה תחזיר **null**	BinTree getRight()
הפעולה מחליפה את התת-עץ השמאלי בעץ tree. **הנחה:** המבנה התקין של העץ לא יפגע (שני התת-עצים יישארו זרים זה לזה)	**void** setLeft (BinTree tree)
הפעולה מחליפה את התת-עץ הימני בעץ tree. **הנחה:** המבנה התקין של העץ לא יפגע (שני התת-עצים יישארו זרים זה לזה)	**void** setRight (BinTree tree)

הערה חשובה: תליית תת-עץ חדש על עץ קיים, יכולה לקלקל את מבנה העץ, מכיוון שאין לנו דרך לבדוק שהתת-עץ החדש אינו פוגע בהגדרת המבנה של העץ (לפיה כל שני תת עצים זרים זה לזה). האחריות להתמודדות עם סכנה זו היא של המתכנת המשתמש במחלקה. עליו לוודא שפעולות העוסקות בשינוי מבנה, שתוכניתו מבצעת על עצמים של המחלקה, שומרות על המבנה התקין של העץ.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ג. שימוש בפעולות הממשק

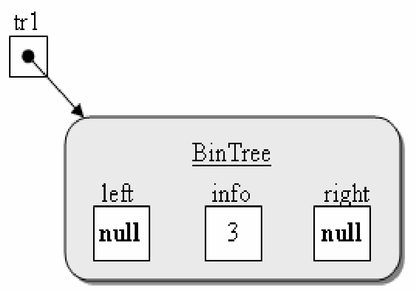
ג.1. בניית עץ

בניית עץ יכולה כאמור להתבצע בשני אופנים. הדרך האחת היא כאשר נתון לנו המידע שבשורש ולא יותר, העץ המתקבל הוא עץ עלה ושתי ההפניות בעץ יקבלו את ערך ההפניה הריקה, **null**. דרך אחרת לבנות עץ היא ממידע שיש לשמור בשורש ושני תת עצים שיהפכו להיות ילדיו של השורש החדש.

נדגים מספר בניות:

בניית עץ עם איבר אחד בלבד שהמידע השמור בו הוא מספר שלם:

BinTree tr1 = **new** BinTree (3);



בניית עץ שבו יותר מאיבר אחד מהשורש לעלים:

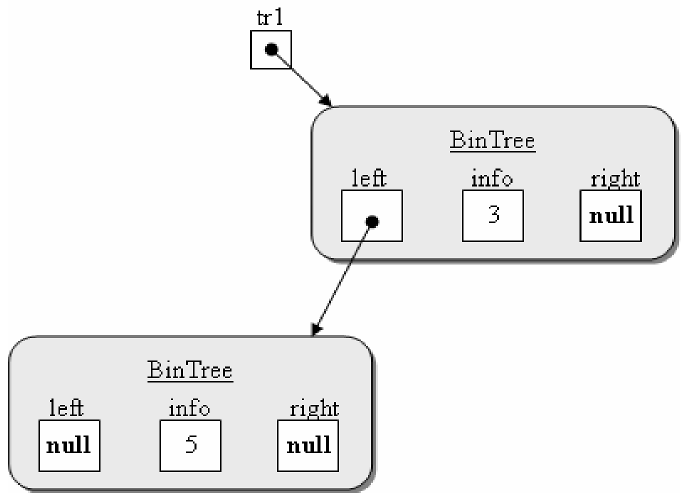
יש לבנות את השורש ולהוסיף עלים במקומות הרצויים על ידי הפעולות setLeft(…) ו-setRight(…):

BinTree tr1 = **new** BinTree (3);

tr1.setLeft (**new** BinTree (5));

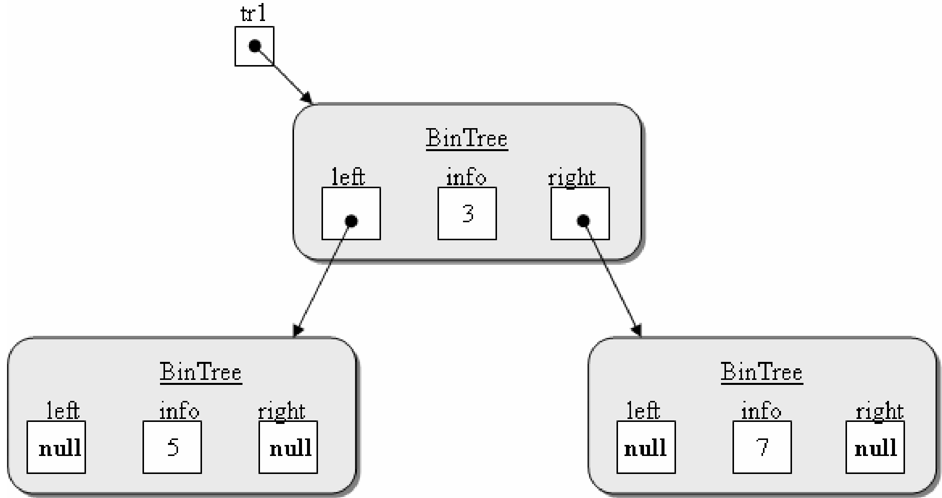
tr1.setRight (**new** BinTree (7));

בשורה הראשונה נבנה העץ כפי שהוא מופיע באיור למעלה. בשורה השנייה נבנה העץ:



© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

בשורה השלישית נבנה העץ:

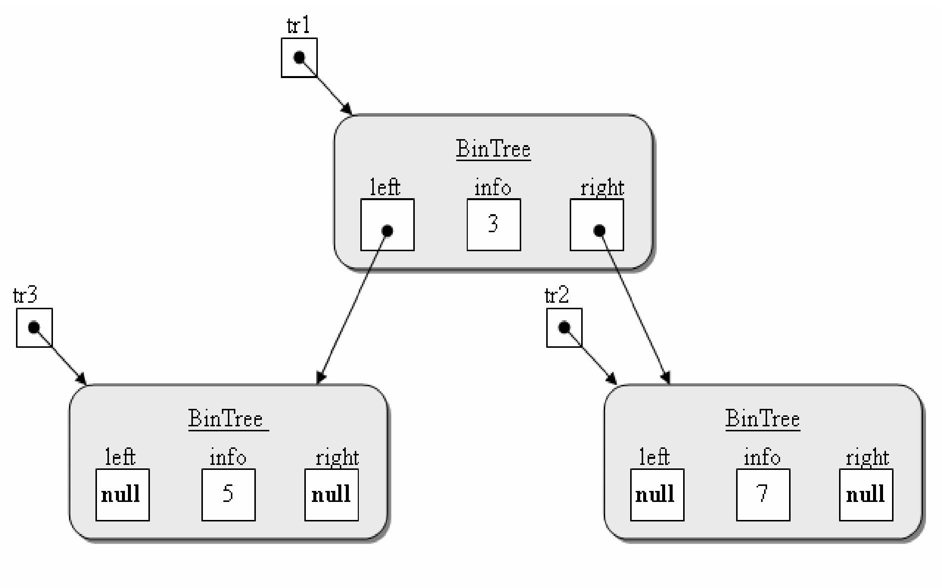


אפשרות אחרת ליצור אותו עץ היא מהעלים אל השורש. יש ליצור שני תת-עצים ולצרפם בעזרת הפעולה הבונה השנייה, לעץ אחד, שבשורשו ערך נתון:

BinTree tr3 **= new** BinTree (5);

BinTree tr2 = **new** BinTree (7);

BinTree tr1 = **new** BinTree (3, tr3, tr2);



ניתן לעשות זאת אף בשורה אחת:

BinTree tr1 = **new** BinTree (3, **new** BinTree (5), **new** BinTree (7));

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ג.2. תנועה על עץ, התייחסות לערכי צמתים

כדי לנוע על עץ נשתמש בשתי פעולות ממשק: האחת - גישה מצומת אל התת-עץ השמאלי שלו, והאחרת - גישה מצומת אל התת-עץ הימני. הפעולה getLeft() הפועלת מתוך שורש של עץ בינרי, מחזירה את התת-עץ השמאלי של שורש זה. באופן דומה הפעולה getRight() מחזירה את התת עץ הימני של שורש זה. באמצעות שתי פעולות אלה נוכל להגיע לכל אחד מצומתי העץ.

במהלך מעבר על עץ נרצה בדרך כלל להתייחס לערכים השמורים בצומת, מכיוון שכל צומת הוא שורש של תת עץ, שניתן להגיע אליו על ידי שימוש בפעולות הגישה, די בפעולות אחזור ועדכון לשורש. לשם כך יש לנו בממשק את הפעולה getInfo() המחזירה את תוכן השורש הנוכחי, והפעולה setInfo(…) המשנה את התוכן של השורש הנוכחי.

קבלת ערך של צומת בעץ המכיל ערכים שלמים:

```
int a = tr1.getLeft().getInfo();
```

כאן, ערך הביטוי tr1.getLeft() הוא התת-עץ השמאלי, והפעולה getInfo() מחזירה את הערך הנמצא בשורש תת עץ זה, כלומר בדוגמה שלעיל יהיה זה הערך 5.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ד. המחלקה עץ בינרי BinTree

```
/** המחלקה מגדירה את מבנה הנתונים עץ בינרי המכיל בצמתיו מספרים שלמים **/
public class BinTree
{
    private int info;
    private BinTree left;
    private BinTree right;

    /** הפעולה בונה עץ עלה שבשורשו info **/
    public BinTree (int info)
    {
        this.info = info;
        this.left = null;
        this.right = null;
    }

    /** הפעולה בונה עץ שבשורשו info, תת העצים שלו הם left, right
    /* אם אין תת עץ מסוים, יועבר null. הנחה: left, right זרים זה לזה */
    public BinTree (int info,  BinTree right, BinTree left)
    {
        this.info = info;
        this.left = left;
        this.right = right;
    }

    /** הפעולה מחזירה את ערך השורש **/
    public int getInfo()
    {
        return this.info;
    }

    /** הפעולה משנה את ערך השורש להיות info **/
    public void setInfo (int info)
    {
        this.info = info;
    }

    /** הפעולה מחזירה את התת עץ השמאלי. אם אין תת עץ שמאלי הפעולה תחזיר null **/
    public BinTree getLeft()
    {
        return this.left;
    }

    /** הפעולה מחזירה את התת עץ הימני. אם אין תת עץ ימני הפעולה תחזיר null **/
    public BinTree getRight()
    {
        return this.right;
    }
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

```
  /** הפעולה מחליפה את התת עץ השמאלי בעץ tree
  /* הנחה: אחרי ההחלפה, שני תתי העץ זרים זה לזה */

  public void setLeft (BinTree tree)
  {
     this.left = tree;
  }

  /** הפעולה מחליפה את התת עץ הימני בעץ tree
  /* הנחה: אחרי ההחלפה, שני תתי העץ זרים זה לזה */

  public void setRight (BinTree tree)
  {
     this.right = tree;
  }
}
```

ד.1. יעילות פעולות הממשק

לאחר שהחלטנו על הייצוג של העץ הבינרי ועל מימושי הפעולות, אפשר לחשב את יעילותן של פעולות הממשק. כל הפעולות מתבצעות מתוך שורש של העץ. אף אחת מהפעולות אינה דורשת "תנועה" על פני העץ, ולכן אף אחת מהן אינה תלויה במספר הצמתים בעץ. משום כך, פעולות הממשק הן בעלות סיבוכיות זמן ריצה קבועה: $O(1)$.

ה. פעולות נוספות לעץ בינרי

בדומה לרשימה, גם בעץ ניתן להוסיף פעולות לממשק בהתאם לבחירת המתכנת ולצורך פונקציונלי כלשהו. אנו נדון במימושי פעולות המרחיבות את הממשק, אך הממשק הרשמי הוא הממשק המוצג בסעיף הקודם. כלומר, לצורך פתרון התרגילים ניתן להשתמש *רק* בממשק המקורי. על מנת להשתמש בפעולות נוספות, יש צורך לממשן, כפעולות פנימיות או חיצוניות, על פי הגדרת המשימה בכל תרגיל.

דוגמה 1: פעולה פנימית פשוטה (לא רקורסיבית):

ממשו את הפעולה isLeaf() הבודקת האם העץ הנוכחי הוא עץ עלה.

פתרון

```
public boolean isLeaf()
{
            return ( this.left == null && this. right == null);
}
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ה.1. העץ כמבנה רקורסיבי

בפעולות על עץ אנו מרבים להשתמש ברקורסיה, משום שגישה זו מתאימה למבנה העץ. אפשר לומר כי מבנה העץ, כפי שהגדרנו אותו, "מזמין" תכנות רקורסיבי. נסביר למה הכוונה ש-"מבנה העץ "מזמין" תכנות רקורסיבי": פעולה היא רקורסיבית אם במהלך ביצועה יש זימון אחד או יותר של הפעולה עצמה. שתיים מתוך התכונות של עץ בינרי הן הפניות מטיפוס עץ, לתת-עץ הימני ולתת-עץ השמאלי, שהם בעצמם עצים. שורשיהם הם צמתים, ולפי הגדרת העץ ניתן להפעיל את פעולות הממשק מכל צומת שהוא. סביר אם כן לכתוב פעולות המבצעות את משימתן על ידי ביצוע פעילות בשורש, בתוספת הפעלות רקורסיביות שלהן בשני התת עצים.

בדוגמה הבאה, וכן בהמשך, נראה פעולות רקורסיביות.

דוגמה 2: פעולה פנימית רקורסיבית:

ממשו פעולה המחזירה את מספר הצמתים בעץ.

פתרון

על מנת לספור את הצמתים בעץ, נספור את הצמתים בתת עץ הימני, נספור את הצמתים בתת עץ השמאלי, ולאחר מכן נחבר את הערכים המתקבלים ונוסיף 1 עבור השורש שגם הוא צומת. ספירת הצמתים בכל תת עץ תיעשה באותה שיטה עצמה (ומשום כך זו פעולה רקורסיבית). כיון שיתכן שתת עץ אינו קיים, יהיה עלינו לבדוק לפני הקריאה הרקורסיבית, האם קיים תת עץ (שמאלי, או ימני).

```
public int numOfNodes ()
{
        int leftCount = 0;
        int rightCount = 0;
        if (this.getLeft() != null)
                leftCount = this.getLeft().numOfNodes();
        if (this.getRight() != null)
                rightCount = this.getRight().numOfNodes();
        return (leftCount + rightCount +1);
}
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ו. מעברים על עץ בינרי

שימושים רבים בעץ בינרי מצריכים מעבר על כל צומתי העץ ללא חזרות, למשל, כדי להדפיס את ערכי הצמתים בעץ. ניתן לעבור על הצמתים בסדרים שונים. את הסדר נבחר בהתאם לצרכי היישום. סדרי מעבר שונים יתאימו לצרכים שונים. ביישומים מסוימים אין חשיבות לסדר, ובלבד שלא נבקר יותר מפעם אחת בכל צומת. לפעמים, כשנמצא את מה שחיפשנו, נפסיק את המעבר לפני סופו.

לדוגמה, אילו רצינו לבדוק האם פלוני נמצא בעץ משפחה, היה עלינו לעבור על העץ ולחפש את שמו בצמתיו. ברור שנדרש חיפוש יעיל במהלכו 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 דורות: ראשית הילד, אחריו הוריו, אחריהם סבותיו וכן הלאה, נזדקק למעבר לרוחב העץ, לפי רמות, תוך מעבר על כל 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 אלו כתבניות שאותן ניישם בהתאם לצרכינו.

ו.1. סריקות עומק של עץ בינרי

קיימות 3 סריקות עומק של עץ: **סריקה תחילית (preorder traversal)**, **סריקה תוֹכית (inorder traversal)** ו**סריקה סופית (postorder traversal)**. בשלושת הסריקות מתבצעות הפעולות הבאות:

1. ביקור בשורש העץ
2. סריקה רקורסיבית של התת-עץ השמאלי
3. סריקה רקורסיבית של התת-עץ הימני

אם הביקור בשורש מתבצע ראשון, הסריקה נקראת **תחילית**. אם הביקור בשורש מתבצע שני (בין שתי הסריקות הרקורסיביות), הסריקה נקראת **תוֹכית**, ואם הביקור בשורש מתבצע בסוף, לאחר שתי הסריקות הרקורסיביות, הסריקה נקראת **סופית**. 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 תת עץ שמאלי לפני ימני.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

באמרנו *ביקור* 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 עוברים בצמתים פעמים נוספות בלי לבצע "ביקור", רק כדי להגיע אל ילדיהם של אותם צמתים, או כדי לחזור דרכם להוריהם.

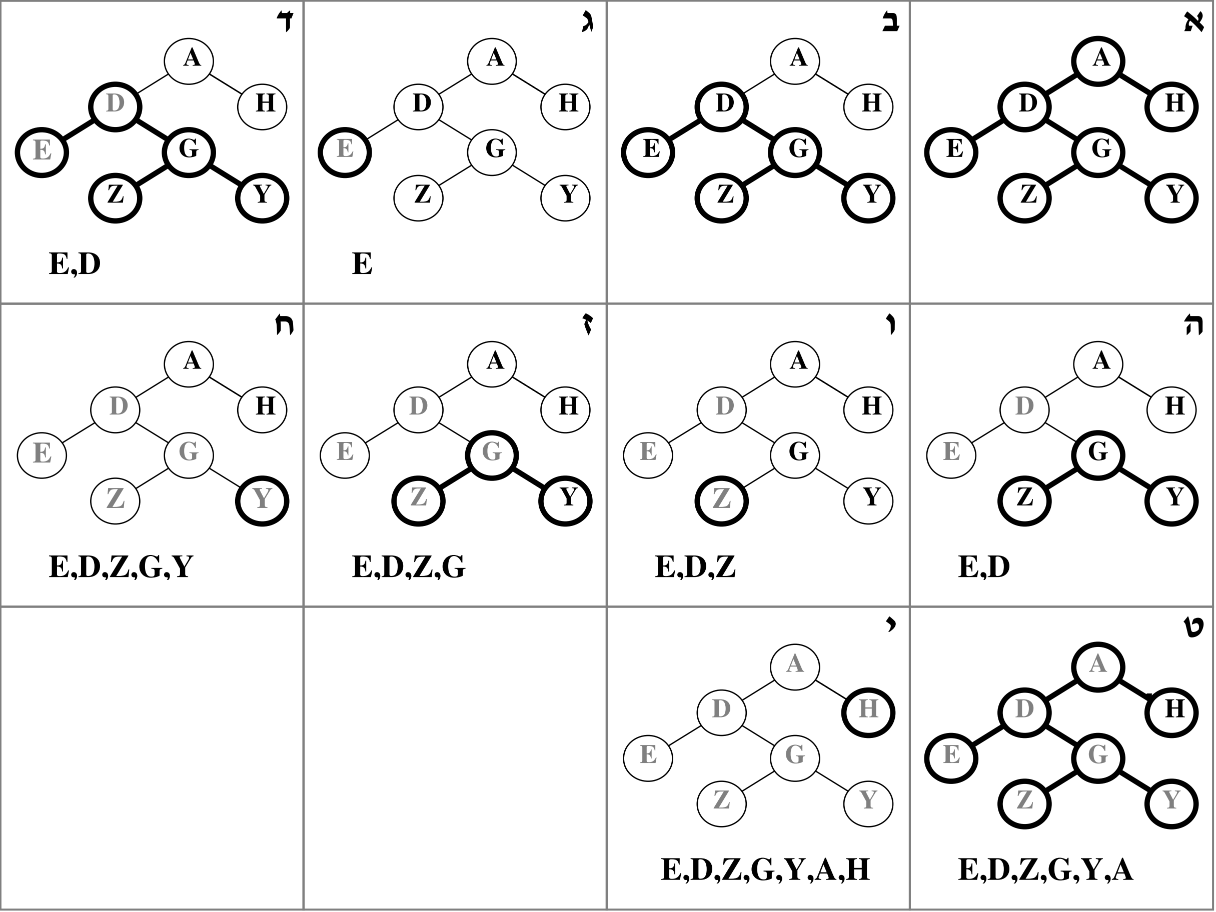
בכל שלוש הסריקות, כאשר תת עץ (שמאלי או ימני, או שניהם) אינו קיים, אזי כמובן אין מבצעים את הקריאה הרקורסיבית עבורו. מכאן שכאשר מגיעים לעלה, הרקורסיה (בתת-עץ הנוכחי) מסתיימת.

בכל שלושת הסריקות קיימת אפשרות להפסיק את הסריקה כאשר התקיים תנאי מסוים, למשל כאשר נמצא הנתון שאותו אנו מחפשים.

לפני שנעבור לניסוח פורמלי מדויק של הסריקות, נבחן את האיור הבא המדגים סריקה בסדר תוֹכי של עץ נתון. 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 כולו. בשלב ב יש לבצע סריקה תוֹכית רקורסיבית של התת עץ השמאלי (ששורשו D). בשלב ג נסרק התת-עץ השמאלי של תת-עץ זה: העלה E. כיון שזה עלה, מתבצע בו ביקור. בשלב ד חוזרים בסריקה לצומת D ומבקרים בו. בשלבים ה-ח נסרק התת-עץ הימני ששורשו G, כאשר סדר הביקור הוא: 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 והביקור בו הוא צעד יחיד.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים



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

הרי שסריקה בסדר תחילי תחזיר סדרה המכילה אותם איברים אך בסדר שונה. הסדרה שתתקבל תהיה: A, D, E, G, Z, Y, H

? עקבו אחרי סריקה בסדר סופי של העץ המופיע באיור שלעיל, וכתבו את סדרת האיברים המתקבלת.

ו.2. כתיבה אלגוריתמית

סריקה יכולה להיות פעולה פנימית או חיצונית. סגנונות הכתיבה האלגוריתמית בשתי הגישות שונים מעט זה מזה. נראה את הסגנונות השונים.

ו.3. סריקות כפעולות פנימיות

מכיוון שסריקת העץ נעשית בכל שלב דרך צומת מסוים, עלינו לציין את הצומת המפעיל ברגע מסוים את פעולת הסריקה. בשלב הכתיבה האלגוריתמית אנו משתמשים בנוסח:

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

*'הפעל **ב-** [צומת מסוים] את הסריקה'*

או כאשר הצומת ידוע:

*'הפעל **בו** את הסריקה'*

ומציינים את סוג הסריקה המבוצעת.

סריקה תחילית (preorder):

סריקה זו מתחילה בביקור בשורש ואז באופן רקורסיבי סורקת את התת-עץ השמאלי ואחר כך את התת-עץ הימני.

סרוק-בסדר-תחילי()
{טענת כניסה: פעולה על העץ הנוכחי }
{טענת יציאה: האלגוריתם סורק את העץ הנוכחי בסדר תחילי}
בקר בשורש העץ
*אם קיים תת עץ שמאלי, הפעל בו את **סרוק-בסדר-תחילי()***
*אם קיים תת עץ ימני, הפעל בו את **סרוק-בסדר-תחילי()***

סריקה תּוֹכית (inorder):

סרוק-בסדר-תּוֹכי ()
{טענת כניסה: פעולה על העץ הנוכחי }
{טענת יציאה: האלגוריתם סורק את העץ הנוכחי בסדר תּוֹכי}
*אם קיים תת עץ שמאלי, הפעל בו את **סרוק-בסדר-תּוֹכי()***
בקר בשורש העץ
*אם קיים תת עץ ימני, הפעל בו את **סרוק-בסדר-תּוֹכי()***

? נסחו את האלגוריתם לסריקה בסדר סופי.

ו.4. סריקות כפעולות חיצוניות

מכיוון שפעולות חיצוניות צריכות לקבל עץ בינרי כפרמטר, אנו משתמשים בשלב הכתיבה האלגוריתמית בנוסח:

'הפעל את הפעולה החיצונית (עץ בינרי כפרמטר)'.

ראשית כותבים את שם הפעולה החיצונית ולאחר מכן בסוגריים, מציינים מיהו העץ הנשלח כפרמטר לפעולה.

מכיון שכאמור אין סיבה לנסח את הסריקות דווקא כפעולות פנימיות, נראה איך תנוסח אחת מאותן הסריקות לעומק שראינו לעיל, כפעולה חיצונית.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

סרוק-בסדר-תוכי *(tree)*

{ טענת כניסה: האלגוריתם מקבל עץ tree של שלמים }

{ טענת יציאה: האלגוריתם סורק את העץ על פי סדר תוכי }

*אם קיים תת עץ שמאלי של tree, הפעל את **סרוק-בסדר-תוכי** (תת עץ שמאלי של tree)*

בקר בשורש העץ

*אם קיים תת עץ ימני של tree, הפעל את **סרוק-בסדר-תוכי** (תת עץ ימני של tree)*

ו.5. יעילות הסריקות

נעריך את סיבוכיות זמן הריצה של האלגוריתמים הרקורסיביים המתוארים לעיל לסריקה של עץ בינרי. הפעולה שמבצע כל אחד מהאלגוריתמים היא הביקור בצומת (שורש של תת עץ). נכלול בחישוב המחיר של הביקור גם את הבדיקות האם קיימים תת עצים בשמאל ובימין, את שני המעברים דרך הצומת שאינם ביקור וכן את החזרה לאחר סיום ביצוע הפעולה כולה על התת עץ שהצומת הוא שורשו. בסך הכל הביקור, כולל כל מה שכללנו בו, מכיל מספר פעולות קבוע, ולכן נמשך זמן קבוע. הביקור הוא הצעד הבסיסי. מכיוון שהאלגוריתמים נבנו כך שהסריקה תכלול ביקור יחיד בכל צומת של העץ, מספר הפעמים שמבוצע הצעד הבסיסי הוא כמספר צומתי העץ. לכן סיבוכיות זמן הריצה של כל אחת מהסריקות היא ליניארית בגודל העץ.

ז. בעיות המשתמשות במעברים על עץ

ז.1. פעולות חיצוניות

כאשר אנו מגדירים פעולה חיצונית למחלקה BinTree, היא מקבלת כפרמטר עץ בינרי שעליו היא תפעל. כאמור בחרנו בשלב זה לטפל בעצים המכילים מספרים שלמים בלבד. גם אם היינו מטפלים בעצים גנריים בפרק, הרי שבשלב הגדרת הפעולות החיצוניות היינו מוכרחים לטפל בעצים קונקרטיים בלבד כמו שהוסבר בפרקים הקודמים.

בייצוג שבחרנו לעץ אין משמעות לעץ ריק. כל עצם של המחלקה BinTree מייצג עץ לא ריק.

כאשר אנו מגדירים פעולות חיצוניות אנו מוכרחים לקחת בחשבון שבטעות עלולה להיות מועברת לפעולה ההפניה **null**. אנו מציינים עובדה זו בתיעוד על ידי הוספת הערה לפיה אנו מניחים שפרמטר מטיפוס עץ בינרי אינו **null**. בקוד עצמו, אנו מסתמכים על הנחה זו, ואיננו בודקים שהערך הפרמטר אינו **null**. (ניתן כמובן להוסיף בדיקות כאלו לקוד, אך בג׳אווה יש שיטה טובה יותר לטיפול בבעיות כאלו, ולכן איננו עושים זאת. שיטה זו המשתמשת בחריגות אינה כלולה בחומר של יחידה זו).

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

דוגמה 1: בדיקת קיום איבר בעץ

ממשו פעולה המקבלת עץ שצמתיו מכילים מספרים שלמים, וערך שלם נוסף, ומשתמשת בסריקה תחילית כדי לבדוק האם הערך נמצא בעץ.

הערה: ברור שבעיה זו אינה מחייבת שימוש בסריקה תחילית דווקא. כל מעבר רקורסיבי פותר אותה. כמו כן, כאשר נמצא הערך הנתון, אין צורך להמשיך את הסריקה.

פתרון

קוד:

```
/** הנחה: ערך הפרמטר tree אינו null */
public static boolean existsIn (BinTree tree, int x)
{
      boolean resL = false;
      boolean resR = false;
      if (tree.getInfo() == x)
            return true;
      if (tree.getLeft() != null)
      {
            resL = existsIn (tree.getLeft(), x);
            if (resL)
                  return true; // הערך המבוקש נמצא ואין צורך להמשיך בחיפוש
      }
      // הערך ודאי לא נמצא בצידו השמאלי של העץ, נמשיך את החיפוש בצד ימין
      if (tree.getRight() != null)
            resR = existsIn (tree.getRight(), x);
      return (resR);
}
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

דוגמה 2: סכום ערכי הצמתים

ממשו פעולה *המקבלת עץ* שצמתיו מכילים מספרים ומחזירה את סכום כל ערכי הצמתים שבעץ.

פתרון

קוד: /* **הנחה**: ערך הפרמטר tree אינו **null** **/

```
public static int sum (BinTree tree)
{
    int leftSum=0;
    int rightSum=0;
    if (tree.getLeft()!=null)
        leftSum = sum (tree.getLeft());
    if (tree.getRight()!=null)
        rightSum = sum (tree.getRight());
    return  tree.getInfo() + leftSum + rightSum;
}
```

ז.2. פעולות פנימיות

הדפסת עצם בג׳אווה נעשית בדרך כלל בעזרת מחרוזת שמחזירה הפעולה toString() המשרשרת את כל תכונות העצם למחרוזת אחת. בעץ בינרי השרשור יכול להעשות בהתאם לסוגי המעברים השונים: preorder, postorder, inorder.

נגדיר שלוש פעולות המחזירות מחרוזת המתארת עץ קיים עליו הן פועלות: preorderString(), postorderString() ו-inorderString(). פעולות אלו ישרשרו את ערכי העץ בהתאם לסדר שהוגדר בשמן, ויחזירו מחרוזות מתאימות.

נדגים את הפעולה preorderString():

דוגמה 1: **מחרוזת של איברי העץ**

כתבו אלגוריתם לפעולת preorderString() המחזירה את איברי העץ הנוכחי על פי סריקה תחילית. ממשו את האלגוריתם.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

פתרון

אלגוריתם:

{ טענת כניסה: האלגוריתם פועל על העץ הנוכחי }

{ טענת יציאה: האלגוריתם מחזיר מחרוזת המורכבת מאיברי העץ הנוכחי על פי סריקה בסדר תחילי }

סרוק את העץ בסריקה תחילית

בכל ביקור בצומת,

צרף מחרוזת המתארת את הצומת (בצירוף סמל מפריד לפי בחירתך),

לסופה של מחרוזת התוצאה.

קוד:

```
public String preorderString()
{
    String str = "";
    str += this.info + " ";
    if (this.left!= null)
        str += this.left.preorderString();
    if (this.right != null)
        str += this.right.preorderString();
    return str ;
}
```

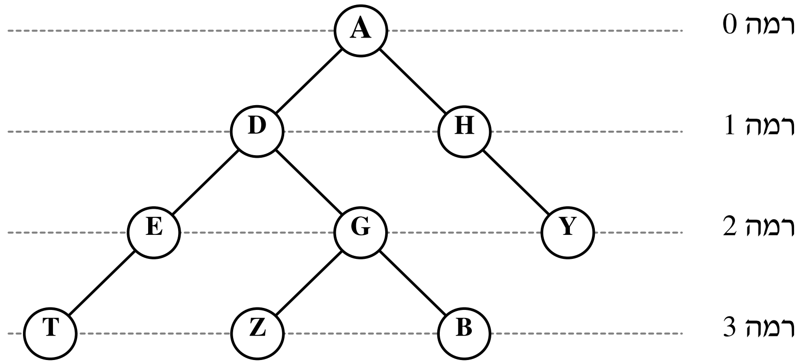
? *ממשו את הפעולות postorderString() ו-inorderString().*

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ח. 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, כלומר זו הרמה הגבוהה ביותר בעץ. גובה העץ הבא הוא 3.



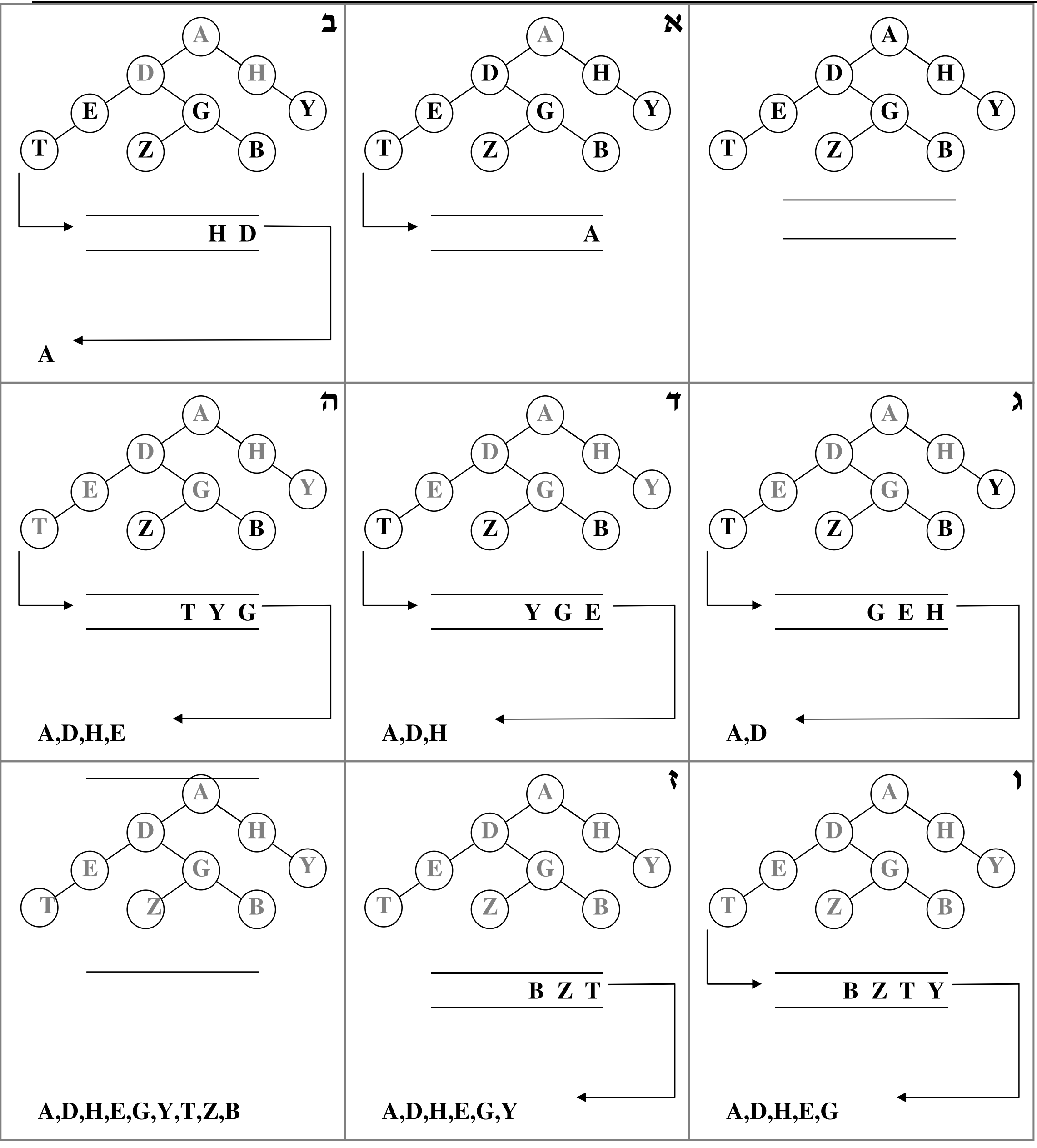
כדי לסרוק עץ לפי רמות, נצטרך להשתמש ברעיון אלגוריתמי חדש. הפעם הסריקה תהיה רמה אחת אחרי קודמתה החל בשורש, כשבכל רמה נסרקים הצמתים משמאל לימין.

נעיין מעט 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 פעולות הממשק שהגדרנו ניתן לגשת לילד רק דרך ההורה שלו, ואילו E הוא ילד של D ולא של H. וכיצד נעבור מ-G ל-Y כשנמשיך בסריקה? הרי אין הם אחים.

נשים לב כי כשאנו מבקרים ב-D, ידוע לנו כי בעתיד נרצה לבקר בילדיו E ו-G, בסדר זה. כאשר אנו ממשיכים ל-H, גם פה נרצה לבקר בילדיו (במקרה זה רק אחד) בעתיד. באופן כללי, כאשר אנו מבקרים בצמתים שברמה מסוימת משמאל לימין, אנו יודעים כי נרצה לבקר בילדי צמתים אלו, שהם הצמתים ברמה הבאה, באותו סדר. כדי לממש רצון זה נשתמש בטיפוס הנתונים תור. כזכור, האיברים מתווספים לתור בסופו ויוצאים מתחילתו, ולכן האיבר שנכנס ראשון לתור הוא זה שיוצא ממנו ראשון. התור שניעזר בו יאחסן תת-עצים שטרם נסרקו. תחילה נכניס לתור הריק את (שורש) העץ כולו. בהמשך, נוציא בכל צעד צומת (דהיינו שורש תת-עץ) מהתור, נבקר בצומת, ונכניס את ילדיו (שניים, אחד או אפס) לתור, בסופו. נמשיך כך עד שיתרוקן התור.

האיור הבא מציג את אופן הסריקה של עץ לפי רמות. בכל שלב מוצגים מצבו של העץ, שורשי העצים הנמצאים בתור ורשימת האיברים שנסרקו.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים



האלגוריתם הבא מציג סריקה של עץ בינרי נתון, לפי רמות, כפעולה פנימית. האלגוריתם משתמש בפעולות הממשק של טיפוס הנתונים תור.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

סרוק-לפי-רמות ()

{ טענת כניסה: האלגוריתם מתבצע על העץ הנוכחי שהוא עץ של שלמים }

{ טענת יציאה: האלגוריתם סורק את צומתי העץ הנתון לפי רמות, משמאל לימין. }

בנה תור חדש של עצים

הכנס את שורש העץ tree לתוך התור

כל עוד התור אינו ריק, בצע את הפעולות:

הוצא עץ מתוך התור

בקר בשורשו של העץ

אם קיים תת-עץ שמאלי לעץ, הכנס אותו לתור

אם קיים תת-עץ ימני לעץ, הכנס אותו לתור

נראה את הקוד המממש פעולה זו כפעולה פנימית הפועלת על העץ הנוכחי שהוא עץ בינרי של שלמים. שימו לב שכדי להתייחס לעץ הנוכחי נשתמש כפרמטר בערך **this** המפנה אל העץ הנוכחי.

```
public String levelOrderTraversal()
    {
        BinTree t;
        String str = new String();
        Queue<BinTree> q = new Queue<BinTree>();

        q.insert (this);
        while (!q.isEmpty())
        {
            t = q.remove();
            str = str + t.getInfo() + " , ";
            if (t.getLeft() != null)
                q.insert (t.getLeft());
            if (t.getRight() != null)
                q.insert (t.getRight());
        }

        return (str);
    }
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ח.1. יעילות הסריקה לפי רמות

נעריך את יעילותו של האלגוריתם. בסריקה לפי רמות הצעד הבסיסי הוא גוף הלולאה שבשורת "כל עוד התור אינו ריק...", והוא מכיל את הפעולות האלה: הוצאת עץ מהתור, ביקור בשורשו והכנסת שני התת-עצים שלו לתור. הנחה (סבירה מאוד) היא שמימוש התור מבטיח שפעולות ההוצאה וההכנסה של איברים אורכות זמן קבוע, ואם כך מחירו של הצעד הבסיס הוא קבוע.

כמו האלגוריתמים הרקורסיביים לסריקת עצים, גם אלגוריתם הסריקה לפי רמות מבקר פעם אחת בדיוק בכל אחד מצומתי העץ, ולכן גם בו מספר הפעמים שמבוצע הצעד הבסיסי הוא כמספר צומתי העץ. סיבוכיות זמן הריצה של האלגוריתם היא לינארית בגודל העץ.

ט. עץ ביטוי

עץ בינרי משמש לפתרון בעיות ממשיות. למשל תכנית המקבלת ביטוי חשבוני ומחשבת את ערכו מייצגת את הביטוי כעץ בינרי, ואז מחשבת את ערכו על ידי אלגוריתם רקורסיבי פשוט (גם מהדר, כגון המהדר של ג'אווה, מייצר קוד לחישוב ביטוי חשבוני בגישה דומה). נבחן מהלך זה לעומקו בפירוט.

כאמור, המהלך מורכב משני חלקים: ייצוג הביטוי כעץ בינרי, וחישוב ערך הביטוי בהינתן עץ זה.

ביטוי חשבוני מורכב ממספרים ומפעולות החשבון: חיבור, חיסור, כפל וחילוק. פעולות החשבון מופעלות על אופרנדים שהם מספרים או ביטויים מורכבים יותר. בייצוג ביטוי על ידי עץ בינרי יש להתחשב בכללים כגון מתן קדימות לפעולות כפל וחילוק על פעולות חיבור וחיסור, וכן קיבוץ נכון של פעולות שיש להם אותה קדימות. כדי להימנע מהסיבוך הכרוך בנושאים אלו, נניח שהביטוי החשבוני "ממוסגר לחלוטין", כלומר סביב כל פעולה ושני האופרנדים שלה, מופיעים סוגריים.

לדוגמה, הביטויים א-ג חוקיים:

א. (7 + 5)

ב. ((3 * 4) + 2)

ג. ((3 – 2) * ((4 * 1) + 8))

ולעומתם הביטויים ד-ז אינם חוקיים (לפי הכללים שהוגדרו לעיל):

ד. (3 + 5) * 4)

ה. 3 – 2

ו. (7–)

ז. (8)

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

לפי כללים אלו אפשר להגדיר באופן פורמלי:

ביטוי חשבוני הוא: **A** כאשר A הוא מספר

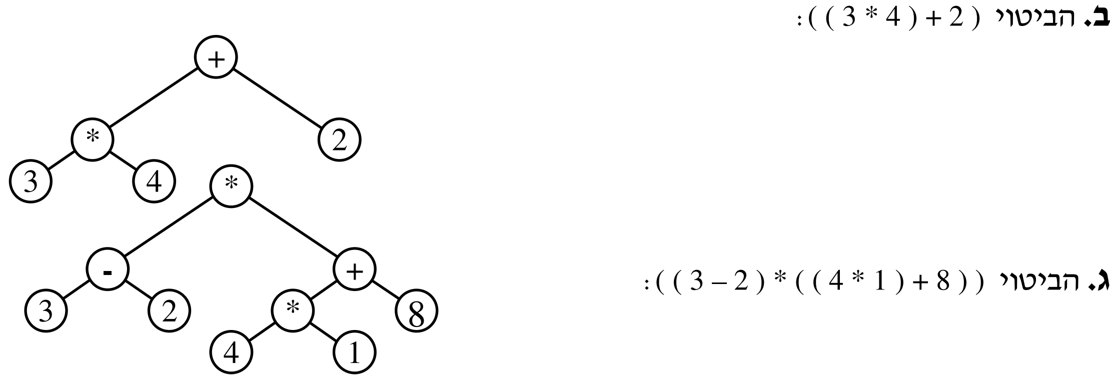
או: **(X op Y)** כאשר X ו-Y הם ביטויים חשבוניים, האופרנדים של הפעולה, ו-op הוא פעולה.

שימו לב כי זוהי הגדרה רקורסיבית: האופרנדים X ו-Y בביטוי מורכב הם ביטויים חשבוניים בעצמם. כיצד ניתן לייצג ביטויים שכאלו? קל לראות התאמה בין הגדרת ביטוי חשבוני להגדרה של עץ בינרי: הענף הראשון – ביטוי שהוא מספר – מתאים לשורש ללא ילדים; הענף 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 או מספר. צומת שלו תת עצים (צומת פנימי) יכיל פעולה, שאותה צריך להפעיל על האופרנדים שהם הביטויים המיוצגים על ידי התת-עצים השמאלי והימני של אותו צומת. עלי העץ יכילו מספרים. באיור הבא מיוצגים ביטויים חשבוניים באמצעות עצים בינריים.

א. הביטוי (7 + 5):

ב. הביטוי (2 + (4 * 3)):

ג. הביטוי ((8 + (1 * 4)) * (2 – 3)):



? *כיצד ייוצג הביטוי ((3 / 9) – ((8 * 7) + 4))?*

בהינתן עץ המייצג ביטוי, חישוב ערך הביטוי על ידי פעולה על העץ קל לביצוע. בתרגילים המצורפים לפרק זה תתרגלו בנייה של עץ ביטוי מביטוי חשבוני נתון, וכן את חישוב ערכו.

יש לשים לב שלצורך תרגול זה אנו זקוקים לעצים בינריים שטיפוס המידע בהם הוא **char**. ניתן להעתיק לשם כך את המחלקה שהוגדרה בפרק, תוך שינוי הטיפוס. בעתיד נגדיר את המחלקה כגנרית וכך נוכל לחסוך את הצורך בהעתקות נוספות לצורך פתרון תרגילים אחרים.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

י. עץ בינרי – מבנה נתונים

עץ בינרי, כמוהו כרשימה, הוא מבנה נתונים אך אינו טיפוס נתונים מופשט.

הפעולות setLeft, setRight מאפשרות למשתמש במחלקה לשנות את המבנה של עצם המייצג עץ בינרי כך שהמגבלות לא יתקיימו יותר (ובמילים אחרות לקלקל את מבנה העץ!). לדוגמה, אם tree הוא עצם כזה, אזי אחרי ביצוע הפעולה:

```
tree.setLeft (tree.getRight());
```

שני התת עצים של tree – זהים ממש. אם, במקום זאת, נבצע על tree את הפעולה:

```
tree.setLeft (this);
```

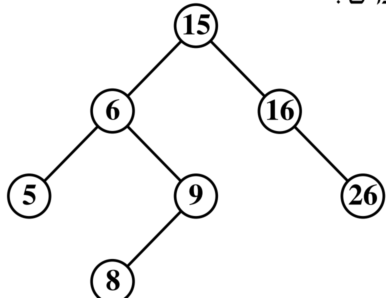
אזי נקבל עצם שבו לכל צומת יש הורה, כלומר אין יותר שורש.

כמו ברשימה, קיום המגבלות תלוי ברצון הטוב (ובידע) של המשתמש.

שני המבנים, רשימה ועץ בינרי, הם מבני נתונים המבוססים על אותו רעיון: שימוש בעצם המכיל הפניה או הפניות לעצמים אחרים מאותו טיפוס. חשיבותם כמבני נתונים היא בכך שהם מאפשרים לייצג ארגוני נתונים נפוצים ושימושיים: סדרות ומבנים היררכיים. בשניהם ניתן להשתמש כייצוגים של נתונים במחלקות המממשות טיפוסי נתונים מופשטים. (ראינו שימושים כאלו ברשימה בפרקים קודמים.)

יא. עץ חיפוש בינרי

לאחר שהכרנו את העצים הבינריים, תכונותיהם ופעולותיהם, נפנה לעיין בסוג מיוחד של עצים בינריים. הסתכלו בעץ הבינרי הבא המאחסן מספרים שלמים:



בעץ זה קיימת תכונת סדר מיוחדת: כל הערכים בצמתים הנמצאים בתת-עץ השמאלי של צומת כלשהו קטנים ממנו, וכל הערכים בצמתים הנמצאים בתת-עץ הימני של הצומת גדולים ממנו. הערכים הנמצאים בתת-עץ השמאלי של העץ הם: 5, 6, 9 ו-8 וכולם קטנים מ-15, שהוא הערך בשורש. הערכים 16 ו-26, הנמצאים בתת-עץ הימני, גדולים מ-15. באותו אופן הערך 5, הנמצא בתת-עץ השמאלי של הצומת 6, קטן מ-6, ואילו הערכים 9 ו-8, הנמצאים בתת-עץ הימני של הצומת, גדולים ממנו.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

עץ בינרי המקיים את התכונה הזו, כלומר לכל צומת בו, כל הערכים בצומתי התת-עץ השמאלי שלו קטנים מהערך בצומת, וכל הערכים בצומתי התת-עץ הימני שלו גדולים מהערך בצומת, נקרא **עץ חיפוש בינרי (binary search tree)**. נעבור לדון בתכונות של עצים שכאלו, תוך הדגשת היתרונות הנובעים מהארגון המיוחד שלהם.

יא.1. איתור ערך בעץ חיפוש בינרי

הגדרתו המיוחדת של עץ חיפוש בינרי מאפשרת ביצוע חיפוש של ערך בעץ באופן מהיר ופשוט הרבה יותר מחיפוש בעץ בינרי כללי. כל השוואה של הערך אותו מחפשים מול שורש נוכחי בעץ חיפוש בינרי, נותנת מענה לשאלות הבאות:

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 בגלל העובדה שהמבנה המיוחד של העץ, מאפשר לבצע בו חיפוש יעיל.

נגדיר אלגוריתם ***אתר-בעץ-חיפוש (x)*** הפועל על עץ חיפוש בינרי ובודק האם הערך x נמצא בו. האלגוריתם מחזיר 'אמת' אם x נמצא בעץ, ו-'שקר' אם אינו נמצא בו.

להלן האלגוריתם לחיפוש רקורסיבי:

אתר-בעץ-חיפוש (x)

אם הערך השמור בשורש הוא x, החזר 'אמת'

אחרת,

אם x קטן מהערך השמור בשורש, וקיים לשורש תת עץ שמאלי,

*החזר את הערך של הפעלת **אתר-בעץ-חיפוש (x)** בתת עץ השמאלי*

אם x גדול מהערך השמור בשורש, וקיים לשורש תת עץ ימני,

*החזר את הערך של הפעלת **אתר-בעץ-חיפוש (x)** בתת עץ הימני*

אחרת, החזר 'שקר'. (הערך לא נמצא בעץ)

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

שימו לב כי קיים דמיון רב בין שיטת חיפוש זו ובין החיפוש הבינרי, שם משווים את הערך המבוקש לאיבר האמצעי במערך, ולפי התוצאה מחליטים באיזה חלק של המערך להמשיך ולחפש.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

החיפוש על עץ חיפוש בינרי, פשוט ויעיל יותר מהפעולה החיצונית existsIn שהוצגה בפרק זה (בסעיף ז.1.). במקרה הגרוע ביותר, כאשר הערך המבוקש אינו קיים בעץ, הפעולה existsIn עוברת על כל הצמתים. גם כאשר הערך קיים בעץ, הפעולה יכולה במקרים מסוימים לעבור על רוב או אפילו על כל הצמתים, לכן סדר הגודל שלה הוא לינארי במספר הצמתים שבעץ. בעץ חיפוש בינרי, מכיוון שאנו יכולים להסתמך על סדר היחס של הערכים בעץ, אנו "עוזבים" בכל שלב בחיפוש את התת עץ שבו ברור שאין לנו מה לחפש, וממשיכים רק בתת עץ בו יש סיכוי לאתר את הערך המבוקש. בכל מקרה, בין אם הערך קיים בעץ ובין אם אינו קיים בו, החיפוש עובר רק על **שביל** אחד, המתחיל בשורש העץ. אם הערך קיים בעץ, השביל מסתיים בצומת המכיל אותו. אם הערך אינו קיים בעץ, השביל מסתיים בצומת שהערך המבוקש צריך היה להימצא בתת עץ שלו, אך אותו תת עץ אינו קיים.

יא.2. מציאת ערך מינימלי בעץ חיפוש בינרי

ברשימה כללית, מציאת הערך המינימלי דורשת מעבר על כל חוליות הרשימה. ברשימה ממוינת בסדר עולה, מציאת הערך הקטן ביותר, פשוטה: הוא ממוקם בראשית הרשימה. באופן דומה, מציאת הערך המינימלי בעץ בינרי כללי, דורשת במקרה הגרוע לעבור על כל צמתיו. מה לגבי מציאת הערך הקטן ביותר בעץ-חיפוש? קל לראות, כי ערך זה ממוקם בצומת השמאלי ביותר בעץ. (באופן סימטרי, הערך הגדול ביותר בעץ החיפוש נמצא בצומת הימני ביותר בעץ.)

כלומר, ניצול המידע הנתון לנו על מבנהו של עץ החיפוש, מוזיל מאד את הפעולות המחזירות את הערכים הקיצוניים השמורים בעץ.

? *נמקו את הטענה שהערך המינימלי בעץ נמצא בצומת השמאלי ביותר.*

? *האם הצומת השמאלי ביותר הוא תמיד עלה? אם לא, האם יתכן שיש לו תת עץ שמאלי? האם יתכן שיש לו תת עץ ימני? אם התשובות לשתי שאלות אלו חיוביות, הצדיקו אותן בעזרת דוגמאות מתאימות.*

? *כתבו את האלגוריתם למציאת הערך המינימלי בעץ חיפוש בינרי.*

הערה: ראינו כי בעץ חיפוש בינרי יש פעולת חיפוש יעילה של ערך נתון, וכן פעולת חיפוש יעילה של הערך המינימלי בעץ. אך מדוע שיהיה לנו עניין בחיפושים אלו? יכולים להיות שימושים מעניינים לחיפושים אלו אם לכל ערך מספרי השמור בעץ יוצמד מידע נוסף, למשל, יתכן כי הערך בעץ הוא מספר תעודת זהות, והמידע הנוסף הוא פרטי בעל התעודה. בהמשך הפרק נרחיב בנושא זה.

כמו כן, בעוד שאנו עוסקים בעצים שערכיהם הם מספרים, ניתן באופן דומה לבנות עצי חיפוש עם ערכים מכל תחום בו מוגדר יחס סדר, למשל מחרוזות. כך, נוכל לבנות גם עץ חיפוש בינרי שבצמתיו שמות (ואליהם מצורפים מספרי טלפון).

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

יא.3. הכנסת ערכים לעץ חיפוש בינרי ובניית עץ

עץ חיפוש בינרי הוא *מקרה* פרטי של עץ בינרי כללי, בו מוגדר סדר על האיברים השמורים בעץ. בעץ בינרי כללי לא הצגנו פעולת הכנסה של ערך לעץ, כיון שתוצאת הפעולה אינה מוגדרת היטב. לא כך הדבר בעץ חיפוש בינרי. התנאי המגדיר את מבנה עץ החיפוש קובע כיצד נוסיף לו ערך: אם הערך זהה לזה שבשורש העץ הנתון, יש להודיע שהערך כבר קיים בעץ. אם הערך קטן מזה שבשורש, יש להכניסו לתת-עץ השמאלי של השורש: אחרת, יש להכניסו לתת-עץ הימני. כך ממשיכים לרדת בעץ, עד שמגיעים לצומת שהערך צריך להיות משמאלו (או מימינו) אך התת-עץ השמאלי (ימני) שלו אינו קיים, ושם מכניסים צומת חדש המכיל את הערך. קל לראות שפעולה זו שומרת על תכונת העץ: אחרי ההכנסה הוא עדיין עץ חיפוש בינרי. מבנהו של עץ חיפוש בינרי מוביל לפעולת הכנסה המוגדרת באופן פשוט וברור. שימו לב שכמו בפעולת החיפוש, גם פעולת ההכנסה עוברת על שביל המתחיל בשורש ויורד בעץ.

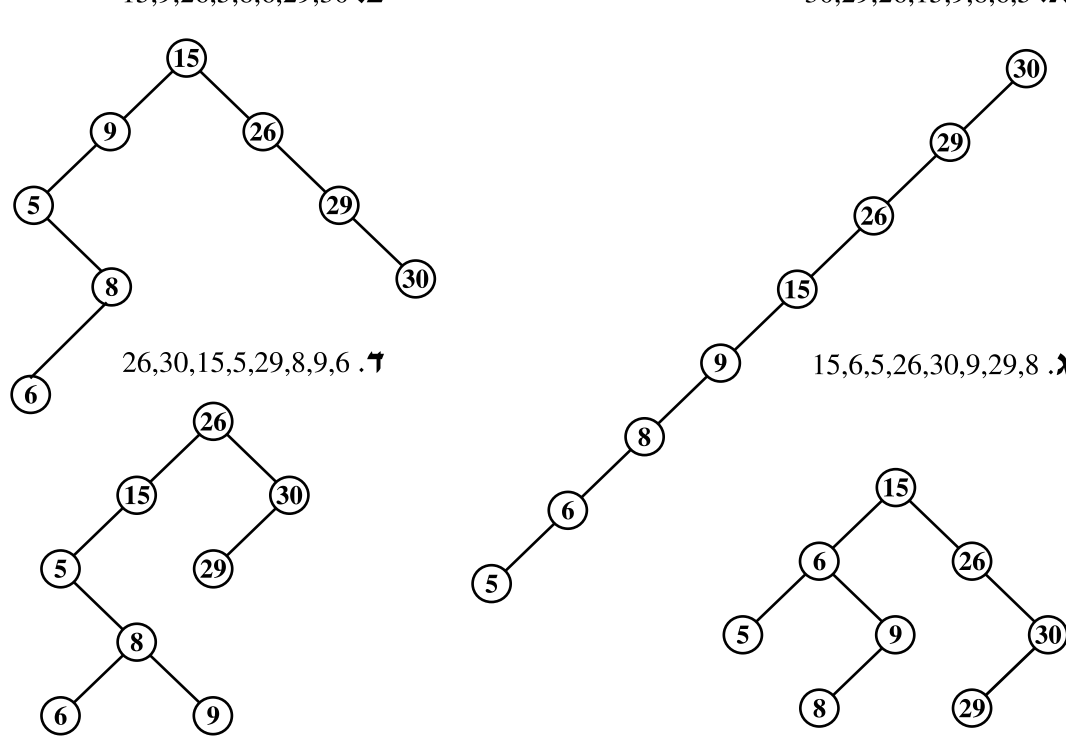
? *כתבו במדויק את האלגוריתם להכנסה של ערך לעץ חיפוש בינרי.*

כאשר עסקנו בעצים בינריים כלליים, בנינו אותם על ידי צירוף צמתים זה לזה באופן שרירותי. העובדה שיש לנו פעולת הכנסה יעילה לעץ חיפוש בינרי, השומרת על תכונותיו, מאפשרת לנו לבנות עץ חיפוש בינרי המכיל קבוצה נתונה של ערכים באופן הבא: נבנה עץ עלה, המכיל את אחד הערכים. לאחר מכן נוסיף את שאר הערכים אחד אחרי השני, תוך שימוש בפעולת ההכנסה המתוארת. מובטח לנו שנקבל עץ חיפוש בינרי המכיל את כל הערכים (וגם את המידע הנוסף הצמוד לכל ערך, אם יש כזה). עץ החיפוש הבינרי יאפשר חיפוש יעיל של ערכים מהקבוצה.

האם לקבוצה נתונה של ערכים מתקבל תמיד אותו העץ, בלי תלות בסדר ההכנסה של האיברים?

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

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים



? *בדקו בעזרת האלגוריתם לבניית עץ חיפוש בינרי, על ידי הכנסת הערכים אחד אחד, כי העצים שבאיורים אכן נוצרו על פי הסדרות המתאימות, על ידי הכנסת הערכים שבהן משמאל לימין. הציגו סדרה המכילה אותם ערכים, אך עץ החיפוש הבינרי הנבנה ממנה שונה מכל העצים שבאיור.*

אם כן, קיימים עצי חיפוש בינריים שונים המכילים אותם ערכים, ומסתבר שצורת העץ הסופית תלויה בסדר הכנסת הערכים לתוכו בעת הבנייה. 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 מראה עץ שנוצר על פי סדרת ערכים הממוינת בסדר יורד. מה תהיה צורתו של העץ שייווצר על ידי אותם ערכים הממוינים בסדר עולה?*

יא.4. גובה העץ כמדד ליעילות

אם נשווה את העצים שבאיורים א ו-ג לעיל, הרי ברור שלמרות שהם מכילים אותה קבוצת ערכים, חיפוש בראשון יהיה פחות יעיל מאשר בשני. עד כה לא דנו בפירוט בסיבוך פעולות החיפוש וההכנסה בעץ חיפוש בינרי, אם כי ההשוואה של פעולות החיפוש לפעולות חיפוש דומות בעץ בינרי

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

כללי נטתה בבירור לטובתו של עץ החיפוש הבינרי. הגיע הזמן לעסוק בנושא זה, ובפרט בשאלה מה הקשר בין צורת העץ לסיבוכיות של פעולות החיפוש בו.

באיור הבא, ארבעה עצים בינריים שבכל אחד מהם שבעה צמתים. ההבדל בין העצים מתבטא בסידורם של הצמתים. הסידור משפיע על מספר הצמתים בכל אחת מרמות העץ, ולכן גם על גובהו של העץ.

עץ א הוא המאוזן מבין העצים. הוא מכיל את מספר הצמתים המקסימלי בכל אחת מהרמות, ולכן הוא הנמוך מביניהם - גובהו 2. עץ ד הוא הפחות מאוזן. הוא מכיל צומת אחד בכל רמה, ולכן גובהו מקסימלי - 6. שני העצים האחרים שבאיור מתארים מצבי ביניים, וגובהם 3 ו-4.

עץ בינרי שכל רמותיו מלאות (מכילות את מספר האיברים 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 מרמת השורש (רמה 0) ועד לרמה k:

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

? *נסו להוכיח את השיוויון באמצעות אינדוקציה או על ידי חישוב של סכום הנדסי.*

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

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 ל-$2^{k+1}-1$.

בדיון הקודם עסקנו בשאלה מהו מספר הצמתים המינימלי והמקסימלי בעץ שמספר רמותיו, k, נתון. נוכל גם לשאול את השאלה ההפוכה: מהו טווח הגבהים של עץ בינרי שבו n צמתים? למעשה, זו השאלה המעניינת מבין השתיים, שכן מה שמעניין אותנו הן הצורות האפשריות של העץ, בהינתן n ערכים שאנו רוצים לאחסן בו.

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

או: $n + 1 = 2^{k+1}$

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

ונקבל לבסוף: $k = \log_2(n+1) - 1$

אולם, ברור כי לא כל מספר n יכול להיות מספר הצמתים בעץ בינרי מלא, אלא רק מספר n המקיים ש: 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, נמלא רמות אחת אחרי השנייה. אם לא מתקיים השוויון $n = 2^{k+1} - 1$, עבור k מסוים, אזי הרמה האחרונה לא תהיה מלאה לגמרי. זהו אם כן **עץ בינרי כמעט מלא**. במקרה זה, $\log_2(n+1)$ אינו מספר שלם. אם מספר הרמות בעץ כזה הוא k, אזי מתקיים $2^k - 1 < n < 2^{k+1} - 1$ ומכאן נובע כי $k < \log_2(n+1) < k+1$. מכאן 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 את המסקנה הבאה: 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 שסיבוכיות החיפוש בעץ בינרי המכיל n צמתים נעה בין *O(log n)* במקרה הטוב ביותר, ל- *O(n)* במקרה הרע, מכיון שכפי שהסברנו לעיל, פעולת חיפוש ערך בעץ בינרי עוברת רק על שביל המתחיל בשורש. מספר הצמתים בשביל זה הוא לכל היותר אחד יותר מגובה העץ. פעולת ההכנסה של ערך לעץ אף היא עוברת על שביל בעץ, ולכן מספר הצמתים שעליו היא עוברת אף הוא נקבע על ידי גובה העץ, ולכן סיבוכיותה זהה לזו של פעולת החיפוש.

כמובן שאם n מספר קטן, למשל 5 או 10, אין הבדל גדול בין סדרי גודל אלו. אך אם n=1000, למשל, ההבדל משמעותי: $2^{10} = 1024$, ולכן *log(n)* כאן הוא (כמעט) 10, בעוד ש-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 גדל.

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

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

מסקנה זו מעלה שאלה חשובה: במקרה הגרוע, סיבוך הפעולות בעץ חיפוש בינרי הוא מאותו סדר גודל כמו בעץ בינרי רגיל, כך שאין במקרה זה ניצול של התכונה המיוחדת של עץ חיפוש בינרי. האם ניתן להימנע מהמקרה הגרוע? התשובה לשאלה זו חיובית. קיים אלגוריתם הכנסה לעץ חיפוש בינרי הבודק אם פעולת ההכנסה מקלקלת את האיזון בעץ. אם כן, האלגוריתם משנה את מבנה העץ כדי להחזיר את האיזון. שימוש באלגוריתם זה מבטיח שהעץ יישאר מאוזן גם אחרי הכנסות רבות, וסיבוך פעולות החיפוש בו יישאר לוגריתמי. אלגוריתם זה אינו כלול ביחידת לימוד זו. (המעוניינים מוזמנים לחפש באינטרנט את האלגוריתם המדובר תחת המונח: red black tree, ולהתרשם מהדגמות נאות של תהליך האיזון.)

יא.5. מחלקה וממשק לעץ חיפוש בינרי

עד כה הגדרנו לגבי עץ חיפוש בינרי שלוש פעולות: בנייה, הכנסה וחיפוש. מימוש האלגוריתם להוצאה של ערך מעץ חיפוש בינרי מורכב, ולכן איננו מציגים אותו בפרק, אך פעולת ההוצאה מעץ חיפוש בינרי ניתנת בהחלט לביצוע.

פעולה נוספת שכדאי לשקול את הגדרתה עבור עץ חיפוש בינרי, היא קבלת אוסף כל הערכים שבעץ, כך יתאפשר לבצע על העץ פעולות מועילות רבות הקשורות לערכים אלו. כיון שבעץ יש סדר, קל לממש פעולה זו כך שתחזיר אוסף ממוין, בסדר עולה, של ערכים אלו. לשם כך, די לעבור על העץ בסדר תוֹכי.

נשוב לדיון בבניית עץ חיפוש בינרי. כפי שהוסבר לעיל, הדרך הנכונה לבניית עץ חיפוש בינרי כך שתכונת הסדר שלו תישמר, היא על ידי בניית עץ עלה והכנסת שאר הערכים לעץ בעזרת פעולת ההכנסה. בניית עץ חיפוש תוך שימוש בפעולות setLeft, setRight, כפי שעשינו בעץ כללי, עלולה לחבל בתכונת עץ החיפוש. לכן סביר ומתבקש להסתיר פעולות אלה כאשר מתייחסים לעץ חיפוש בינרי.

מסיבות דומות, רצוי להסתיר את הפעולות getLeft, getRight, getInfo, setInfo, ולהשתמש רק בפעולות החיפוש שהצגנו לעיל. אי הסתרת פעולות אלו תאפשר למשתמש לשנות ערכים בעץ באופן שרירותי, וכך להרוס את מבנהו.

במילים פשוטות, לפי העקרונות הבסיסיים של תכנות מונחה עצמים, כיון שיש לנו טיפוס נתונים עם פעולות שייחודיות לו, הרי שזהו טיפוס נתונים חדש, ולכן יש לבנות עבורו מחלקה חדשה, BinSearchTree. ממשק המחלקה יורכב מפעולות הבנייה, החיפוש, ההכנסה, ההוצאה ואחזור כל הערכים שבעץ, כפי שהוגדרו לעיל.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

BinSearchTree
ייצוג ...
BinSearchTree (**int** x)
void insert (**int** x) **void** remove (**int** x) **boolean** exists (**int** x) **int**[] getAllNumbers()

ייצוג המחלקה נתון לבחירתנו. ניתן לחשוב על ייצוג עץ חיפוש בינרי באופן דומה מאד לייצוג עץ כללי. BinSearchTree ייוצג על ידי שלוש תכונות: תכונה המכילה את המידע השמור בצומת ושתי תכונות שהן הפניות לעצמים נוספים מטיפוס BinSearchTree. המחלקה תיכתב מחדש תוך הגדרת פעולות המתאימות לממשק עץ החיפוש בלבד.

ניתן לחשוב גם על ייצוג שונה שייעזר בעץ הכללי. בגישה זו תהיה למחלקה BinSearchTree, תכונה פרטית שערכה יהיה עץ מטיפוס BinTree. המחלקה BinSearchTree תעטוף את העץ הכללי ותמנע את השימוש בפעולות שלו שאיננו רוצים לחשוף למשתמש (פעולות הפוגעות במבנה עץ החיפוש), כמובן שמימושי פעולות הממשק החדש יהיו חופשיים להשתמש בכל פעולות המחלקה BinTree, ואם יש צורך בכך, גם בפעולות נוספות, כגון סקירות (למשל כדי להחזיר רשימה של כל הערכים בעץ). כיון שפעולות המחלקה BinTree, ובפרט כל פעולות השינוי של מחלקה זו, מוסתרות מהמשתמש במחלקה BinSearchTree (למעשה הוא אינו חייב כלל לדעת שנעשה שימוש במחלקה זו במימוש), הרי שאין למשתמש במחלקה BinSearchTree שום דרך לעקוף את הממשק, או לקלקל את מבנהו של הטיפוס החדש. מכאן שמחלקה זו, בניגוד למחלקה BinTree, מגדירה טיפוס נתונים מופשט.

את כתיבת המחלקה אנו משאירים לשלב התרגול.

יא.6. שימושים בעץ חיפוש בינרי

אם נסרוק עץ חיפוש בינרי בסדר תוֹכי, הרי שנקבל סדרה ממוינת בסדר עולה של כל האיברים הנמצאים בו. אם כך, אפשר לנצל את הסריקה של עץ חיפוש בינרי בסדר תוֹכי לצורך מיון של רשימה: תחילה נבנה עץ חיפוש מאיברי הרשימה המקורית, ולאחר מכן נסרוק את עץ החיפוש שנוצר בסדר תוֹכי, כך שבכל ביקור בצומת יצורף הערך שבו, לסופה של רשימה חדשה. הרשימה

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

המתקבלת תכיל את איברי הרשימה המקורית כשהם ממוינים בסדר עולה. מיון כזה נקרא **מיון-עץ** (tree sort).

למעשה, מכיוון שכללנו בפעולות של עץ חיפוש בינרי, פעולה המחזירה את אוסף הערכים שבו, ממוינת בסדר עולה, הרי שכל שעלינו לעשות זה להכניס את איברי הרשימה הנתונה לעץ, אחד, אחד, ואחר כך להשתמש בפעולה זו כדי לקבל את הרשימה הממוינת.

מהי היעילות של שיטת מיון זו? אם ברשימה n איברים, הרי שיש לנו n פעולות הכנסה, ואחריהן סריקה תוכית של עץ המכיל n צמתים, פעולה שאת סיבוכיותה כבר חישבנו והיא $O(n)$. לגבי ההכנסות, אם מבנה העץ נשאר מאוזן במשך כל ההכנסות, אזי מחיר כל הכנסה הוא $O(\log n)$, ולכן סיבוך כל ההכנסות הוא $O(n \log n)$. במקרה זה, זו שיטת מיון יעילה. אם איזון העץ התקלקל, אזי יתכן שסיבוך רוב ההכנסות הוא $O(n)$, סיבוך כל ההכנסות יחד הוא $O(n^2)$ ואז זו שיטת מיון סבירה, אך יש יעילות ממנה. כפי שהערנו קודם, בגישה פשוטה להכנסה אי אפשר להבטיח שהעץ יישאר מאוזן, אך ידועות שיטות המבצעות הכנסה באופן יעיל וגם שומרות על האיזון.

היסטורית, חוקרי מדעי המחשב המציאו את מושג עץ החיפוש הבינרי, ומצאו עבורו אלגוריתמים יעילים לחיפוש, הכנסה והוצאה כדי לקבל מבנה שאפשר לשמור בו נתונים ולאחזר אותם באופן יעיל יותר מאשר ברשימה, אפילו כשהיא ממוינת. לצורך הדיון העקרוני במבנה העץ ומימוש הפעולות עליו, די לעסוק בעץ שצמתיו מכילים מספרים שלמים. אבל ביישומים אין הרבה עניין לבנות עץ של מספרים ואחר כך לחפש אותם בו. הרבה יותר מעניין ומועיל לטפל בעץ שכל צומת בו מכיל מפתח ומידע נוסף, למשל תעודות זהות שאליהן מקושרים נתוני אנשים במרשם התושבים, או מספרי סטודנט המקושרים לרשומות הסטודנטים באוניברסיטה וכד׳. קל לראות שכל מימושי הפעולות של עץ חיפוש בינרי שעסקנו בהן ניתנים להכללה בקלות, כך שישתמשו ברכיב המפתח של תוכן צומת. כמו כן, כשם שהגדרנו עצים שהערכים בהם (כלומר המפתחות) הם מספרים שלמים, אפשר להפעיל אותם אלגוריתמים גם על עצים שהערכים (מפתחות) בהם, הם מחרוזות או ערכים מכל תחום שיש בו סדר.

היכולת לשמור מפתחות וערכים בעץ שיכול לגדול ולקטון ללא הגבלת גודל, יחד עם היכולת לאתר ערכים דרך מפתחותיהם בקלות, נראית בסיס טוב למגוון יישומים מעניינים. זה הרגע להיזכר שאת הצירוף הזה של שמירת מפתחות וערכים הקשורים אליהם, יחד עם פעולות הכנסה, חיפוש, הוצאה והחזרה של כל הנתונים, אנו מכירים מטיפוס הנתונים מפה שהגדרנו בפרק הקודם. במילים פשוטות, הממשק של המחלקה ה״חדשה״ כבר קיים. מה שגילינו בפרק זה למעשה היא דרך ייצוג ומימוש חדשה למפה, באמצעות עץ חיפוש בינרי.

הפעולות של טיפוס הנתונים מפה מכילות את הפעולות שאנו ביקשנו להגדיר לגבי עץ חיפוש בינרי המכיל מפתחות וערכים: בניה, בדיקה האם מפתח נמצא בעץ, הכנסה והוצאה וכן קבלת אוסף ממוין של כל המפתחות. יש במפה פעולה אחת נוספת: החזרת ערך הקשור למפתח, שבעץ שיש בו רק מפתחות אין בה תועלת. מימוש פעולה זו כאשר המפה ממומשת על ידי עץ בינרי דומה מאוד למימוש פעולת החיפוש. ואמנם, אחת הסיבות שלא עסקנו בהרחבה בהגדרת המחלקה

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

BinSearchTree הייתה שידענו שאין היא אלא גרסה מעט מנוונת של Map ורצינו להגיע להבנה זו כאן, ולהתמקד במימוש החדש של Map.

האם שיפרנו במשהו את יעילות הפעולות של המפה כאשר עברנו מייצוגה בעזרת רשימה של זוגות לייצוג בעזרת עץ בינרי של זוגות המאורגן כעץ חיפוש בינרי? כפי שראינו בדיוננו הקודם במפה, סיבוך פעולות חיפוש ברשימה הוא מסדר גודל *O(n)*. בעץ חיפוש בינרי, אם הוא מאוזן, סדר הגודל הוא *O(log n)*. מצד שני, הכנסה לרשימה, כאשר המיקום אינו חשוב, נעשית ב-*O(1)* בעוד שבעץ חיפוש בינרי המיקום חשוב, ולכן סיבוך ההכנסה הוא *O(log n)*. אם נניח תרחיש (טיפוסי למדי) שבו בונים בתחילה עץ בינרי מאוזן, ואחר כך רוב הפעולות הן חיפושים, כאשר מדי פעם יש גם לבצע הכנסה או הוצאה, הרי שהרווחנו, כי המחיר הממוצע של הפעולות ירד משמעותית.

יא.6. מבני נתונים לעומת טיפוסי נתונים מופשטים – דיון מסכם

עם סיום הפרק האחרון, מן הראוי לשוב ולדון 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 טיפוס. על ידי "חיבור" חוליות כאלו זו לזו, אנו בונים מבנים משורשרים שונים, שהם מבני נתונים.

בפרק "רשימה", המחלקה המגדירה את טיפוס החוליה היא Node. עצמים מטיפוס זה הם חוליות עם הפניה יחידה. על ידי שרשור חוליות כאלו ניתן לבנות רשימות, כפי שאמנם עשינו. מחוליות שנוצרות מ-Node ניתן לבנות גם מבנים אחרים, כגון מעגל המורכב משרשרת חוליות, כמה רשימות או כמה מעגלים, ועוד. כל אלו הם מבני נתונים. לרשימות של חוליות הגדרנו מחלקה עוטפת בשם List. מחלקה זו אינה מסתירה את מבנה שרשרת החוליות שבה, היא רק מציעה מגוון פעולות מועילות על רשימה של חוליות. כיון שהמבנה, כאוסף חוליות משורשרות, חשוף והמשתמש במחלקה יכול לטפל בו ישירות, גם עם מחלקה זו אנו עדיין עוסקים במבנה נתונים.

בפרק זה, המחלקה המגדירה את טיפוס החוליה היא BinTree, ולה שלוש תכונות (מידע ושתי הפניות ל-BinTree).

המחלקה מספקת פעולות לקריאה ושינוי של כל אחת משלוש התכונות. על ידי שרשור חוליות כאלו זו לזו ניתן לבנות מבנה של עץ בינרי, ואז אנו מתייחסים לחוליות כאל צמתים של עץ. שימו לב, כי למרות שם המחלקה, עצם ממחלקה זו הוא חוליה עם שתי הפניות ולא עץ! אולם, כיון שלא הגדרנו מעטפת מיוחדת לעץ כזה, משתנה בתוכנית יכול להפנות לעץ בינרי רק על ידי הפניה לחוליה שכזו, וזו הסיבה שקראנו לה BinTree. שימו לב כי ניתן לבנות בעזרת חוליות מטיפוס BinTree גם מבנים אחרים, למשל רשימות דו-כיווניות, שבהן לכל חוליה יש הפניה אחת

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

לחוליה העוקבת, והפניה אחרת לחוליה הקודמת. כלומר, השם BinTree אינו משקף שום משמעות פנימית של המחלקה, אלא רק את השימוש שאנו עושים בה בפרק זה.

לסיכום, עץ בינרי הוא מבנה הבנוי מחוליות כאלו. גם אם נכלול במחלקה פעולות על העץ, כגון סקירות, המבנה נשאר חשוף, ומשתמש במחלקה יודע על המבנה ומשתמש בו ישירות. לכן כשאנו בונים עצים כאלו ומשתמשים בהם, אנו עוסקים במבני נתונים.

נתבונן עתה במחלקה Map. בממשק מחלקה זו יש פעולה בונה, וכן פעולות הכנסה, חיפוש והוצאה של מפתחות וערכים, ופעולה לייצוא כל המפתחות באוסף. כל הפעולות מתייחסות לערכים (של המפתחות או הערכים) באוסף בלבד, ואינן מזכירות כלל חוליות. אין שום חשיפה של המבנה הפנימי, האם מדובר במימוש בעזרת מערך, או אולי על ידי חוליות מסוג זה או אחר. ואמנם, ראינו כי ניתן לממש מפה על ידי שימוש ברשימה כתכונה פרטית לאחסון האוסף, ובפרק זה ראינו כי ניתן גם לממש מפה בעזרת עץ חיפוש בינרי, כלומר על ידי עץ בינרי המקיים תכונה מסוימת. למעשה, בשפת ג׳אווה, המימוש היעיל של מפה הוא מימוש שלישי, לא על ידי רשימה ולא על ידי עץ חיפוש, אלא על ידי מבנה הקרוי טבלת ערבול. מכאן ברור שמפה היא טיפוס נתונים מופשט: היא מגדירה טיפוס על ידי פעולות ממשק בלבד, אינה חושפת אף פרט מהמימוש ומאפשרת למעשה מימושים שונים מאד זה מזה.

לבסוף, נבקש לציין דבר נוסף לגבי המחלקה BinSearchTree. כפי שכבר נאמר, היא מגדירה טיפוס נתונים מופשט שהוא מקרה מנוון של מפה (=מפתחות ללא ערכים). הפעולות הכלולות בממשק של המחלקה BinSearchTree, שהן פעולות חיפוש, הכנסה והוצאה של ערכים, אינן מחייבות כלל שמימושה יהיה על ידי שימוש בעץ בינרי! ניתן בהחלט לממש מחלקה זו על ידי שימוש בתכונה פנימית שהיא רשימה ממוינת המחזיקה את אוסף הערכים.

אם כן, עץ חיפוש בינרי הוא שמו של מבנה נתונים מסוים, שתכונתו האופיינית תומכת במספר פעולות מועילות עליו. ברגע שאנו מתרכזים רק בפעולות אלו ומתעלמים מהמבנה, אנו עולים ברמת ההפשטה, ומקבלים טיפוס נתונים מופשט.

אפשר להצדיק את השם BinSearchTree למחלקה, כיוון שהיא ממומשת על ידי מבנה הנתונים עץ חיפוש בינרי. שימו לב שאין שום איסור על משתמש במחלקה המגדירה טיפוס נתונים מופשט לדעת כיצד מומשה, ובלבד שלא תהיה לו כל דרך לעקוף את הממשק ולפעול ישירות על המימוש. יכול יצרן תוכנה בהחלט לומר ללקוח: אני מוכר לך מחלקה המאפשרת לבנות מאגרי נתונים דינמיים, לחפש בהם, ולשנות את תכולתם, וכל זאת ביעילות, כיוון שהיא ממומשת על ידי עץ חיפוש בינרי. אם המוכר מקפיד שהמחלקה תגדיר טיפוס נתונים מופשט, יוכל לשנות בעתיד את המימוש ליעיל יותר, בלי לפגוע בתוכנות שהלקוח פיתח על בסיס המחלקה שלו.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

יב. סיכום

- עץ הוא מבנה שבו לכל צומת, פרט לשורש, יש צומת אחד המכונה "הורה" שלו, ומספר כלשהו של צמתים המכונים "ילדים" שלו.
- **עץ בינרי** הוא **שורש** עם **שני תתי עצים לכל היותר**, כאשר:
  - כל תת עץ הוא עץ בינרי
  - שני תת עצים בינריים אלו זרים זה לזה, כלומר אין להם צמתים משותפים.
- העץ הבינרי הקטן ביותר נקרא **עץ עלה.**
- ממשק המחלקה BinTree המגדירה עץ בינרי של מספרים שלמים:

הפעולה בונה עץ עלה שבשורשו הערך info	BinTree (**int** info)
הפעולה בונה עץ שבשורשו הערך info, התת-עץ השמאלי שלו left והתת-עץ הימני right. כל אחד משני הפרמטרים האחרונים יכול להיות **null**, ואז התת-עץ המתאים אינו קיים. שני פרמטרים **null** מגדירים עץ עלה. **הנחה:** left ו-right זרים זה לזה	BinTree (**int** info, BinTree left, BinTree right)
הפעולה מחזירה את ערך השורש	**int** getInfo()
הפעולה משנה את ערך השורש להיות info	**void** setInfo (**int** info)
הפעולה מחזירה את התת עץ השמאלי. אם אין תת עץ שמאלי, הפעולה תחזיר **null**	BinTree getLeft()
הפעולה מחזירה את התת עץ הימני. אם אין תת עץ ימני, הפעולה תחזיר **null**	BinTree getRight()
הפעולה מחליפה את התת-עץ השמאלי בעץ tree. **הנחה:** המבנה התקין של העץ לא יפגע (שני התת-עצים יישארו זרים זה לזה)	**void** setLeft (BinTree tree)
הפעולה מחליפה את התת-עץ הימני בעץ tree. **הנחה:** המבנה התקין של העץ לא יפגע (שני התת-עצים יישארו זרים זה לזה)	**void** setRight (BinTree tree)

- מבנהו של העץ הבינרי מאפשר טיפול נוח באמצעות אלגוריתמים רקורסיביים.
- קיימים שני סוגים עיקריים של אלגוריתמים לסריקה של עץ בינרי: אלגוריתמים הסורקים את העץ באופן רקורסיבי ואלגוריתם לסריקה לפי רמות. יעילותם של אלגוריתמים אלו לינארית במספר הצמתים.
- עץ בינרי הוא מבנה נתונים אך אינו טיפוס נתונים מופשט.
- מספר הרמות בעץ בינרי שבו n צמתים נע בין $O(n)$ ל-$O(\log n)$.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

- 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 מהערך בצומת.
- מחלקה המציעה פעולות בנייה, הכנסה, חיפוש והוצאה, הממומשת על ידי עץ חיפוש בינרי, מגדירה טיפוס נתונים מופשט. גם אם קוראים לה BinSearchTree היא אינה חייבת להיות מיוצגת בעזרת עץ בינרי (למרות שמהּ).
- ניתן לייעל את פעילות המחלקה Map, כאשר מממשים אותה על ידי עץ חיפוש בינרי.

מושגים ומילות מפתח

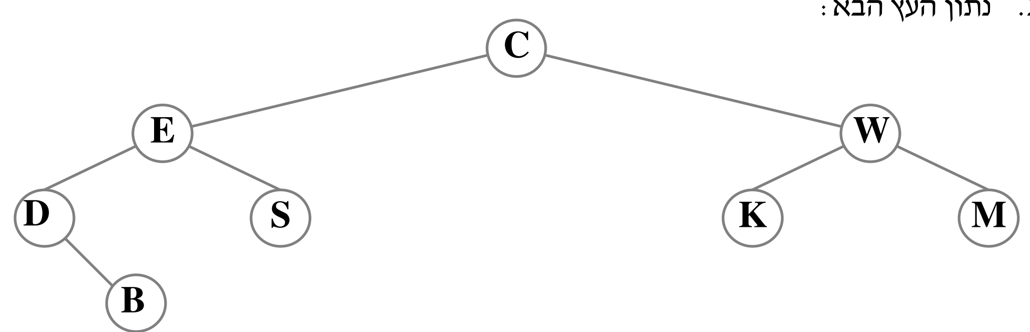
אח	sibling
גובה עץ	tree height
הורה	parent
הורה-קדמון	ancestor
ילד (שמאלי, ימני)	child (left, right)
סריקה בסדר סופי	postorder traversal
סריקה בסדר תוֹכי	inorder traversal
סריקה בסדר תחילי	preorder traversal
סריקה לפי רמות	level traversal
עלה	leaf
עץ	tree
עץ בינרי	binary tree
עץ חיפוש בינרי	binary search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

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

תרגילים

א. מושגים המגדירים עץ בינרי

1. נתון העץ הבא:



א. השלם:

i. מספר הצמתים בעץ הוא _____

ii. מספר העלים בעץ הוא _____

iii. השורש של התת-עץ השמאלי של E הוא _________

iv. W הוא ____ של C

v. C הוא _____של B

vi. הגובה של העץ הוא _____

vii. הצאצאים של E הם __________

ב. איזה משפט מבין המשפטים הבאים הוא משפט נכון:

i. B הוא צאצא של E, C ו-W

ii. D, S, K ו-M הם אחים

iii. K הוא ברמה 3

iv. אם הצמתים נמצאים באותה רמה אזי הם אחים

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ב. סריקות לעומק

2. ממשו את האלגוריתם הבא המדפיס את הערכים השמורים בעץ בינרי בעזרת סריקה בסדר תוֹכי של צמתי העץ.

אלגוריתם

הדפס-בסדר-תוֹכי *(tree)*

{ טענת כניסה: האלגוריתם מקבל עץ tree של שלמים }

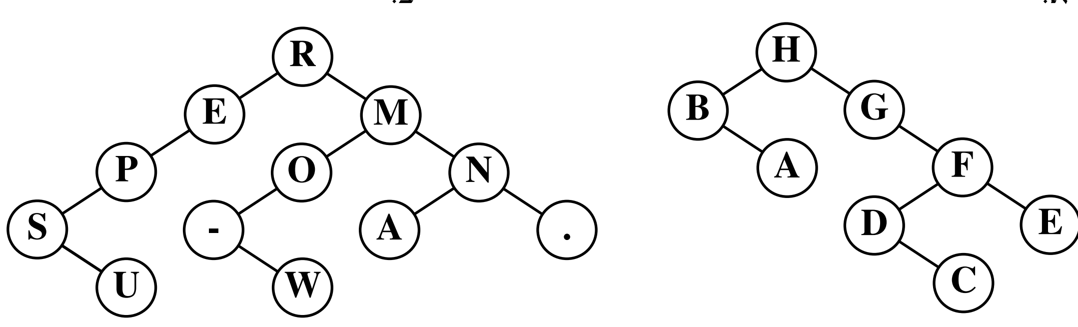
{ טענת יציאה: האלגוריתם מדפיס את כל הערכים שבעץ על פי סדר תוכי }

אם קיים תת עץ שמאלי של tree, הפעל את ***הדפס-בסדר-תוֹכי*** *(תת עץ שמאלי של tree)*

הדפס את שורש העץ

אם קיים תת עץ ימני של tree, הפעל את ***הדפס-בסדר-תוֹכי*** *(תת עץ ימני של tree)*

3. לפניכם שני עצים בינריים:



רשמו את רשימת הצמתים שתתקבל לפי כל אחת משיטות הסריקה לעומק של עצים אלו (סדר תוֹכי, סדר תחילי וסדר סופי).

4. כתבו פעולה חיצונית המדפיסה את כל המספרים הזוגיים המופיעים בעץ.

5. כתבו פעולה חיצונית המקבלת עץ ומחזירה את מספר הצמתים שבו.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ג. תרגילי מעקב

6. הגדירו מה מבצעת הפעולה secret() הבאה המוגדרת בתוך המחלקה BinTree?

```
public boolean secret()

{

   if (this.right == null && this.left == null)

                        return true;

   if (this.right == null || this.left == null)

                        return false;

   return (this.right.secret() && this.left.secret());

}
```

7. מה מבצעת הפעולה הבאה המוגדרת בתוך המחלקה BinTree?

```
public int what ()
{
   if (this.getLeft() == null  &&  this.getRight() == null)
      return 1;
   int nl = 0;
   int nr = 0;
   if (this.getLeft() != null)
      nl = this.getLeft().what();
   if (this.getRight() != null)
      nr = this.getRight().what();
   return (nl + nr);
 }
```

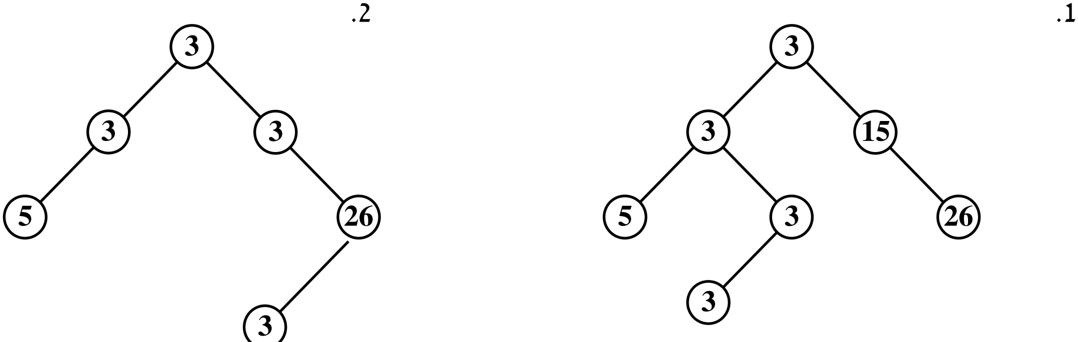
8. קראו את קטע הקוד הבא המגדיר פעולה פנימית במחלקה BinTree :

```
   public boolean mystery (int x)
   {
      if (this.getInfo() !=  x)
            return false;
      if (this.getLeft() == null  &&  this.getRight() == null)
            return true;
      if (this.getLeft() == null)
            return this.getRight().mystery (x);
```


© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

```
    if (this.getRight() == null)
        return this.getLeft().mystery (x);
    return (this.getLeft().mystery (x) || this.getRight ().mystery (x));
}
```

א. איזה ערך תחזיר הפעולה: mystery (3) כשתפעל על העצים הבאים:



ב. מה מטרת הפעולה?

ד. בניית עצים

9. ממשו את הפעולה הבאה:

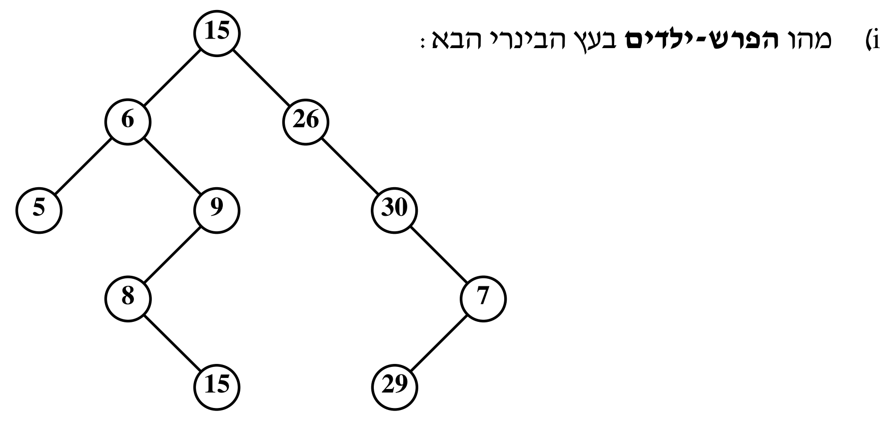
```
public static BinTree buildIdentTree(BinTree tree)
```

הפעולה בונה ומחזירה **עץ בינרי זהה** (במבנה ובתוכן) לעץ tree המתקבל כפרמטר.

ה. פעולות נוספות

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

i) מהו **הפרש-ילדים** בעץ הבינרי הבא:



© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

ii) ממשו פעולה המקבלת עץ בינרי של שלמים ומחשבת **הפרש-ילדים**.

11. כתבו פעולה פנימית המחזירה את גובהו של העץ הנוכחי. (אם המושג **גובה** לא זכור לכם, חזרו לסעיף "סריקה לפי רמות").

12. כתבו פעולה המקבלת עץ בינרי ומחזירה 'אמת' אם העץ מלא ו-'שקר' אחרת.

13. ממשו את הפעולה הבאה:

public static BinTree getParent (BinTree tree, BinTree child)

הפעולה מחזירה את ההורה של child, שהוא אחד מצמתיו של העץ tree.

ו. עצים על פי הגדרה

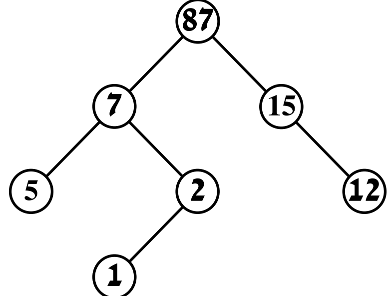
14. **צומת סכום** הוא צומת שערכו גדול מסכום ערכי כל צאצאיו.

א. כתבו את הפעולה **סכום-צמתים (T)** המקבלת עץ בינרי של שלמים, ומחזירה את סכום כל צמתיו.

ב. כתבו פעולה המקבלת צומת בעץ בינרי ומחזירה 'אמת' אם הוא צומת סכום ו-'שקר' אחרת. ניתן להיעזר בפעולה שמימשתם בסעיף א'.

ג. **עץ סכום** הוא עץ שכל צומת בו הוא **צומת סכום**.

i. האם העץ הבא הוא עץ סכום?

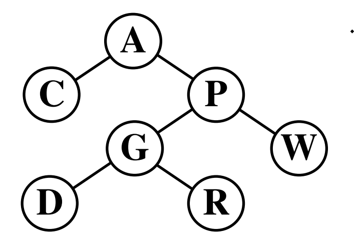


ii. כתבו פעולה המקבלת עץ ומחזירה 'אמת' אם הוא עץ סכום ו-'שקר' אחרת.

15. **עצים דומים** הם עצים שהמבנה הפנימי שלהם, כלומר סדר הצמתים בתוך העץ, זהה. לדוגמה, העצים א ו-ב הם עצים דומים , אף שהם מכילים איברים שונים.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

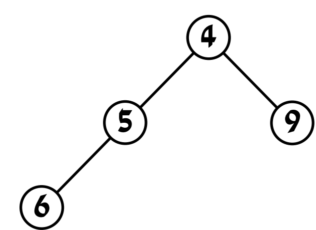
א.

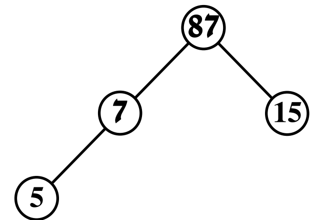


ב.

C R A H E T T

i. האם העצים הבאים הם עצים דומים?





ii. כתבו אלגוריתם **עצים-דומים?** (t1, t2), המחזיר 'אמת' אם העצים t1 ו-t2 הם עצים דומים, ו-'שקר' אחרת.

iii. ממשו את האלגוריתם שכתבתם בסעיף א.

ז. עץ ביטוי

16.

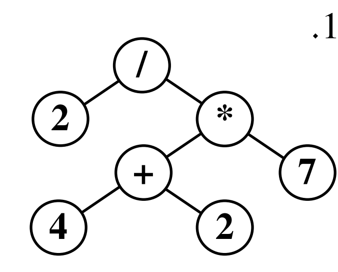
א. ציירו את העצים הבינריים המתאימים לביטויים החשבוניים הבאים:

i. ((2 - (6 / 2)) * (4 * 3))

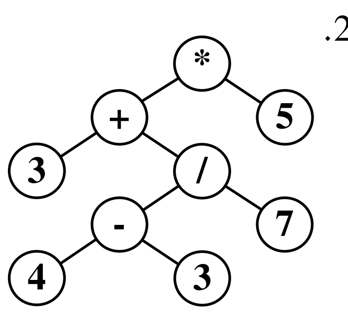
ii. (3 + (6 * ((2 / 4) - 5)))

ב. מהו ערכו של הביטוי המיוצג על ידי כל אחד מהעצים הבאים:

1.



2.



© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

17. לפניכם אלגוריתם המחשב את ערכו של ביטוי חשבוני המיוצג כעץ בינרי.

i. ראשית קבעו היכן תוגדר הפעולה ולפי זה, האם עליה לקבל פרמטרים ואלו?

ii. ממשו את האלגוריתם.

הערה: שימו לב שלא ניתן לפתור בעיה זו תוך שימוש בעץ של שלמים.

חשב-ערך-ביטוי *(???)*

{ האלגוריתם מחשב את ערכו של הביטוי החשבוני המיוצג על ידי עץ ביטוי }

{ב-rightVal וב-leftVal יאוחסנו ערכים מספריים, וב-op יאוחסן אופרטור }

*{**הנחות**: העץ מכיל ביטוי חשבוני תקין }*

אם העץ הוא עלה, החזר את ערכו

אחרת,

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

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

שמור ב-op את הערך של השורש

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

18. נגדיר פעולה הבונה עץ ביטוי חשבוני מביטוי חשבוני. כדי לפשט את הטיפול בביטוי נניח שהוא ממוסגר לחלוטין באופן תקין, מורכב משלמים בלבד ואינו מכיל ביטויים שליליים מהצורה x–.

i. היכן תוגדר הפעולה?

ii. כתבו את הפעולה.

הערה: שימו לב שלא ניתן לפתור בעיה זו תוך שימוש בעץ של שלמים.

ח. עץ חיפוש בינרי

19. א. בנו עצי חיפוש בינריים על פי הרשימות i - iii (משמאל לימין):

i. 5, 15, 7, 10, 14, 12, 11

ii. G ,D ,J ,E ,A ,C ,F

iii. 13, 10, 4, 50, 59, 10, 2, 3, 59, 35, 55, 35

ב. מצאו רשימה נוספת שתיצור עץ חיפוש זהה לזה שנוצר בסעיף iii.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

20. כתבו את המחלקה BinSearchTree. בשלב ראשון החליטו על הייצוג למחלקה (האפשרויות מפורטות בפרק בסעיף יא.5).

21. ממשו פעולה המקבלת אוסף של מספרים שלמים ומחזירה את אוסף המספרים, ממוינים ללא כפילויות.

רמז: יש להשתמש בעץ חיפוש בינרי .

22. ברצוננו להרחיב את הפעולה שבשאלה 21.

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

הנחיה: הערכים שיוחזרו יהיו זוגות. המפתח (=הערך הראשון בזוג) יהיה מספר שלם, והערך של הזוג (=הערך השני בו) יהיה מספר הפעמים בהם הופיע המספר באוסף המקורי.

ט. שיחזור עץ על פי סריקותיו

23. כאשר נתונה סריקה כלשהי, ייתכנו כמה עצים המתאימים לה.

נתונה סריקה תחילית: 3, 7, 9, 5

מצאו שני עצים שונים המתאימים לסריקה זו.

24. כאשר נתונות שתי סריקות: תחילית וסופית, ייתכנו כמה עצים המתאימים לשתי הסריקות:

נתונה סריקה תחילית: 3, 7, 9, 5

נתונה סריקה סופית: 9, 5, 7, 3

מצאו שני עצים *שונים* המתאימים לשתי הסריקות האלו יחד.

25. כאשר נתונות שתי סריקות, ואחת מהן היא סריקה תוֹכית, קיים *רק עץ אחד* המתאים לשתי הסריקות (**בתנאי** שהמספרים בכל סריקה שונים לחלוטין זה מזה). על מנת להבין איך ניתן לשחזר אותו צריך להבין איזה מידע ניתן להפיק משתי הסריקות.

i. בכל סריקה תחילית נתונה, שורש העץ הוא ________________________.

ii. בכל סריקה סופית שורש העץ הוא ________________________________.

iii. בהינתן סריקה תוֹכית. אם ידוע כי שורש העץ הוא מספר העומד במקום מסוים, מה ניתן להגיד לגבי כל המספרים שלפני המקום הזה, ולגבי כל המספרים המופיעים אחרי מקום זה בסריקה? ________________________________.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים

26. נתונות שתי סריקות:

סריקה תחילית: 1, 5, 7, 3, 8, 4, 6

סריקה תוֹכית: 7, 5, 8, 3, 1, 4, 6

ציירו את העץ היחיד המתאים לשתי הסריקות האלו גם יחד.

27. ממשו את הפעולה הבאה:

public static BinTree find (**int**[] preorder, **int**[] inorder)

הפעולה מקבלת שני מערכים של מספרים שלמים שהתקבלו כתוצאה מסריקת עץ בינרי בסריקה תחילית ובסריקה תוֹכית, ומחזירה את העץ המקורי. הניחו שהמספרים בכל מערך שונים זה מזה.

© כל הזכויות שמורות למשרד החינוך ולאוניברסיטה העברית בירושלים