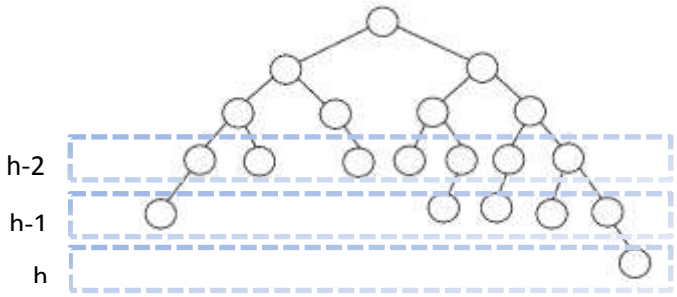
# מבנה נתונים – עבודה 3

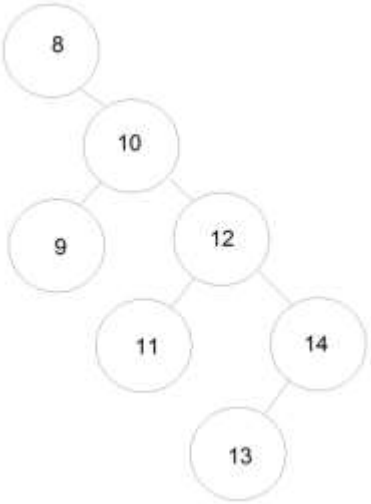
**מגישים: אחיעד שייקר וניצן גואטה**

1. א) לא נכון. דוגמה נגדית:

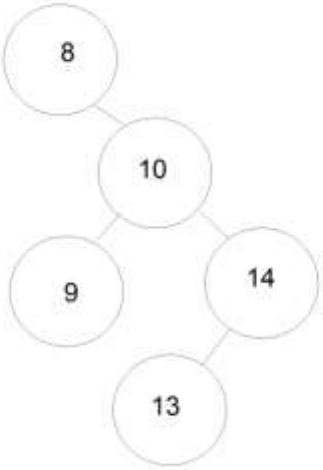



ניתן לראות בדוגמא כי ברמה ה-h2 חסר קודקוד ולכן הרמה אינה מלאה.

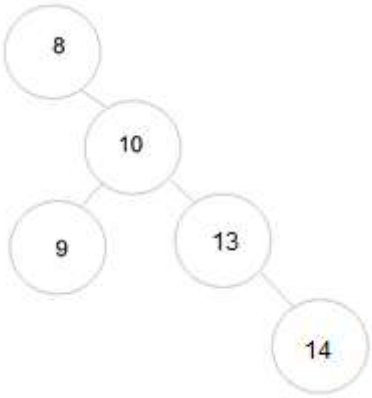
ב) לא נכון. דוגמא נגדית:
בהינתן העץ הבא:




אם נמחק קודם את 11 ואז את 12:

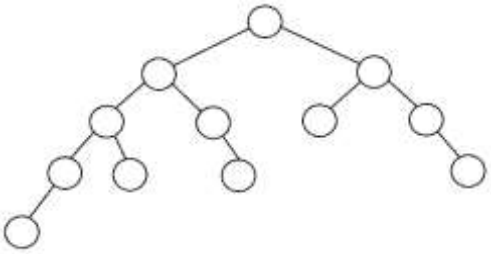


ואילו אם נמחק את 12 ואז את 11 :




ג) נכון.

תחילה נראה שקיים עץ AVL בעל 12 קודקודים בגובה h=4.



כעת, נראה כי לא קיים עץ בעל 11 קודקודים בגובה 4.

נניח בשלילה שקיים עץ AVL בגובה 4 בעל 11 קודקודים.

מההרצאות בכיתה אנו יודעים שגובה של עץ AVL הוא $\theta(logn)$.

אז גובה עץ AVL בעל 11 קודקודים הוא log11 < 4, בסתירה להנחה.

כלומר, מספר הצמתים המינימלי בעץ בגובה 4 הוא 12.

ד) לא נכון.

נניח שלקודקודים בעץ קיים שדה גובה (height).

אנחנו חייבים לעבור על כל הקודקודים בעץ (במקרה הגרוע ביותר) לצורך בדיקה שכל אחד משני תתי העצים של הקודקוד מקיים באופן רקורסיבי את תכונת האיזון של עץ AVL (הפרש גבהים של לכל היותר 1).

בכל בדיקה של קודקוד נצטרך לבדוק שגובה abs(left.height-right.height) $\geq$ 1 ולאחר מכן לבצע את אותה פעולה על left ו-right (בדיקה באופן רקורסיבי לכל קודקוד בעץ).

נסתכל על הפסאודו קוד הבא, פונקציה זו תבדוק מהשורש עד העלה, כך שבמקרה הגרוע ביותר זמנה הוא $\theta(n)$- כמספר הצמתים הקיימים בעץ:

```
isAVLTree(BinaryNode bn){
    int lh // height of left subtree
    int rh // height of right subtree

  if(root == NULL)   // If tree is empty return true
      return  true

  lh = height(root->left)
  rh = height(root->right)
  if( abs(lh-rh)<=1 AND isAVLTree(root->left) AND isAVLTree(root->right))
      return true

  return false   // tree isn't balanced
}
```

2. א) נתאר אלגוריתם:

נתון לנו עץ חיפוש "מקולקל קלות" בו יש צומת אחד (v) בדיוק שלא מקיים את תכונת החיפוש. כלומר, בתת העץ השמאלי של v קיים לפחות מפתח אחד שגדול מ-v.key. או שקיים בתת העץ הימני של v לפחות מפתח אחד שקטן מ-v.key.

כדי למצוא את הצומת הבעייתי נרצה לגלות היכן התכונה לא מתקיימת, נעבור על העץ בסריקת inorder ולכל צומת t אליו נגיע, נכניס למערך את t.key (לשם נוחות נקרא לו מערך inorder). נעבור על כל צמתי העץ (n) ולכן זמן הבנייה של המערך הוא O(n).
באותו אופן, נבנה מערך המכיל את ערכי העץ בסדר preorder (נקרא לו מערך preorder). בדומה למקודם, זמן הבנייה של המערך הוא O(n).
נעבור על ערכי מערך inorder, בכל פעם נשווה את האיבר הנוכחי לאיבר הבא אחריו, מכיוון שהעץ מקולקל קלות, ובנוסף המערך היה צריך להיות ממויין, במעבר על איברי המערך נגלה שיש איבר שמפר את הסדר. בשלב זה אנחנו לא יכולים לדעת איזה צומת בעץ מפר את הסדר (האיבר אותו בעל הערך אותו בדקנו או האיבר בעל הערך שאחריו).
במקרה הגרוע ביותר נעבור על כל איברי המערך, ולכן סריקת המערך היא O(n) .
על מנת שנדע איזה צומת מבין השניים הוא הצומת המקולקל קלות, נעבור על מערך הpreorder, האיבר הראשון מבין השניים שנתקל בו הוא הערך של הצומת הבעייתי, אותו נרצה למצוא בעץ כעת.
סריקת המערך היא O(n) לכן זמן הריצה הוא O (n).
כעת נסרוק את העץ, כשנמצא את הצומת שמכיל את הערך הבעייתי אותו מצאנו קודם, נחזיר מצביע לצומת. לכל היותר נעבור על כל הצמתים בעץ, לכן O(n).
סה"כ זמן הריצה הוא 5O(n)=O(n)

ב) כעת מצאנו את הצומת הבעייתי, ויש בידינו מערך inorder שאם נחלקו ל3 מערכים, נקבל 3 מערכים ממויינים (מתחילת המערך עד לאיבר הקודם לאיבר שמפר את הסדר – מערך ראשון, האיבר שמפר את הסדר- מערך שני, החל מהאיבר העוקב לאיבר שמפר את

הסדר ועד לסוף המערך – מערך שלישי). אנו יודעים שמיון מיזוג של שני מערכים ממויינים הוא O(n), במקרה שלנו נמזג בכל פעם זוג מערכים שזה $2O(n) = O(n)$.
אחרי שמיינו נעבור על העץ בשנית בסדר *inorder* ונכניס את איברי המערך הממויין *(inorder)* ובצורה זו נקבל עץ ממויין.
סריקה של כל הצמתים בעץ בסדר *inorder O(n)*, החלפת ערכים היא *O(1)* לכל צומת ולכן זמן הריצה הכולל הוא *O(n)*.

3. נממש בעזרת פסאודו קוד:

```
Sort(Queue q1,int n){
    int min
    int curr, counter<-n
    queue q2
    while(!q1.isEmpty()){
        min<-q1.dequeue()
        Counter<-counter-1
        If (!q1.isEmpty())
            for(i=0 upto counter)
                curr <- q1.dequeue()
                if(min>curr)
                    q1.enqueue(min)
                    min<-curr
                else q1.enqueue(curr)
        Q2.enqueue(min)
    }
    While(!q2.isEmpty())
        Q1.enqueue(q2.dequeue())
    Return q1
}
```

הסבר על מימוש הקוד:
בשלב הראשון משלוף את האיבר הראשון מהתור ונגדיר אותו להיות המינימום ((O(1)).
כל עוד התור המקורי לא ריק (יש n איברים לכן O(n) ריצות), נרוץ כמספר האיברים ((O(n) ) בו ונבדוק בכל פעם האם האיבר הנוכחי קטן מהמינימום((O(1)). אם כן, אנחנו נכניס בחזרה לתור את המינימום הנוכחי ונגדיר את המינימום החדש להיות האיבר שמצאנו((O(1)).
לאחר מכן, האיבר המינימלי שהיה בתור נמצא במשתנה מינימום, נכניס אותו לתור העזר ונוריד את כמות האיברים בתור המקורי ב-1 (שכן האיבר המינימלי נכנס לתור החדש) ((O(1)).
לאחר שבכל שלב מצאנו את המינימום, נגיע למצב בו התור המקורי ריק. כעת, נרצה להעביר את האיברים מהתור החדש אל התור המקורי: נשלוף מהתור החדש ונכניס לתור המקורי ((O(n)) .

באופן כללי:

$O(n)\ (O(n)+4O(1))+ O(n) = O(n^2)+ O(n) = O(n^2)$

4. נתאר אלגוריתם אשר הופך עץ חיפוש בינארי כלשהו בעל n צמתים לעץ חיפוש בינארי אחר שצורתו נתונה.

בשלב הראשון נהפוך את עץ החיפוש הבינארי שלנו לשרשרת ימנית בעזרת O(n) רוטציות ולאחר מכן נעבור על השרשרת לפי הסדר ונכניס לעץ החיפוש הבינארי שצורתו נתונה את הערכים בסדר inOrder.

<u>שלבי האלגוריתם</u>:

א. נבצע rightRotation על שורש העץ עד אשר אין לו בן שמאלי.

ב. לכל צומת שאין לו בן שמאלי, נעבור לבן הימני.

ג. במידה וקיים בן שמאלי לצומת נבצע rightRotation עד אשר לא קיים בן שמאלי לצומת (בדומה לסעיף א').

ד. במידה ושני הבנים של הצומת הם null, כלומר הצומת הוא עלה, נעצור את התהליך (סיימנו לבנות את השרשרת הימנית).

ה. נעבור בסדר inOrder על העץ שצורתו נתונה, ונכניס את הצומת עליה אנו מצביעים בשרשרת שיצרנו.

ו. בשרשרת נקדם את המצביע על הצומת להיות המצביע על הבן הימני ונבצע שוב את ה' עד אשר הגענו לnull בשרשרת ונעצור.

נתאר את זמני הריצה:

א. זמן הריצה של rightRotation הוא O(1) , דבר שנלמד בכיתה. במקרה הגרוע ביותר (שרשרת שמאלית) נבצע n-1 פעמים, כלומר בסך הכל O(n).

ב. O(1) קריאה לבן הימני. לכל היותר נקרא לn בנים ימניים, לכן O(n).

ג. RightRotation הוא O(1) ומספר הפעמים שזה יבוצע הוא מספר הבנים השמאליים שיש לצומת. במקרה הגרוע ביותר O(n).

ד. עצירת התהליך היא O(1).

ה. מעבר בסדר inOrder על העץ שצורתו נתונה הוא O(n), הכנסת הצומת O(1)

ו. קריאה לבן הימני O(1), נבצע n פעמים, לכן O(n) בסך הכל.

זמן הריצה הכולל הוא $O(n)+O(n)+O(n)+O(1)+O(n)+O(n)=O(n)$

5. מבנה נתונים יורכב משני עצי AVL.

עבור הגמדים – dwarfTree .

עבור הענקים - giantTree.

בעץ dwarfTree כל צומת בנוסף למצביעים רגילים יכיל מצביעים לעוקב וקודם.

init() - אתחול שני עצים ריקים, O(1).

insertDwarf(location) - הכנסת גמד חדש לעץ dwarfTree, *O(log(n)) במקרה הגרוע.*

insertGiant(location) - הכנסת ענק חדש לעץ giantTree, *O(log(n)) במקרה הגרוע.*

isTalking(L1,L2) - נבצע חיפוש אחר L1 ו L2-בעץ dwarfTree.
במידה ואחד מהם (או שניהם) לא נמצאים, נחזיר הודעת שגיאה. *O(log(n)) לחיפוש.*
אחרת, ניקח את min(L1,L2) ונבצע insertGiant(min(L1,L2)) *O(log(n)) להכנסה.*
נמצא את העוקב שלו ב giantTree- נחפש את ערך המינימום ב giantTree *O(log(n)) לחיפוש*, ונחזיר את מיקום של הענק הקרוב ביותר מימין *O(1) להחזרת הערך.*
נסמן את מיקומו ע"י Suc .
נמחק מהעץ את min(L1,L2), *O(log(n)) למחיקה.*
אם max(L1,L2) > Suc אז הענק נמצא בין L1 ל- L2 והם לא יכולים לדבר.
אחרת L1, ו L2 יכולים לדבר.
במידה ולא קיים עוקב ל min(L1,L2) בעץ giantTree , אז אין ענק מימין לגמד min(L1,L2) ולכן L1 ו- L2 יכולים לדבר.

*בסה"כ זמן ריצה : O(log(n)) במקרה הגרוע.*

remove(location) – נחפש את location ב- dwarfTree *O(log(n)) לחיפוש*, אם מצאנו גמד, נמחק אותו *O(log(n)) למחיקה* לפי אלגוריתם מחיקה של עצי AVL.
אחרת, נחפש את location ב- giantTree *O(log(n)) לחיפוש*, אם מצאנו ענק, נמחק אותו *O(log(n)) למחיקה.*
אחרת, לא קיים ענק או גמד בlocation ונחזיר הודעת שגיאה *O(1).*

*בסה"כ זמן הריצה : O(log(n)) במקרה הגרוע.*

whomTalking(location) – תחילה, נבצע insertGiant(location). *O(log(n)) להכנסה.*

נחפש את location ב giantTree *O(log(n)) לחיפוש*, ונחזיר את העוקב והקודם שלו *O(1)* , נסמן אותם highLimit ו-lowLimit בהתאם.

נמחק את location מgiantTree. *O(log(n)) למחיקה.*

נחפש את location בdwarfTree, *O(log(n)) לחיפוש,* מכיוון שקיימים מצביעים לעוקב והקודם של כל צומת, נתקדם מlocation ימינה (עוקב) ונדפיס כל ערך עד אשר נגיע לערך שגדול מ – highLimit, בנוסף נתקדם שמאלה (קודם) ונדפיס עד אשר נגיע לערך שקטן מ- lowLimit. *O(k) כאשר k הוא מספר הגמדים שיש לנו בטווח.*

אם הגענו לצומת כלשהו שהעוקב או הקודם שלה הם null, נפסיק את ההדפסה באותו הכיוון.

*בסה"כ זמן הריצה: O(log(n) + k) במקרה הגרוע.*