הוכן ע"י אמיר רובינשטיין

מבני נתונים ומבוא לאלגוריתמים

נושא 10

הרחבה של מבני נתונים ועצי דרגות
Augmenting Data Structures and Rank trees

בתוכנית

- נלמד כיצד מרחיבים מבני נתונים קיימים ומוכרים כדי שיתאימו לפתרון בעיות חדשות

- נדגים זאת באמצעות עצי דרגות – הרחבה של עצי AVL

מוטיבציה

- במקרים פשוטים ADT ניתן למימוש יעיל באמצעות מבנה נתונים מוכר (ערימה, מערך, AVL...)
- לעיתים קרובות נדרש שילוב כלשהו של מבני נתונים

- ישנם מקרים בהם מימוש יעיל אפשרי ע"י הרחבה של מבני נתונים מוכרים.
- למשל:

נניח שברצוננו לממש מילון - $\text{Insert}(S, x)$, $\text{Delete}(S, x)$, $\text{Search}(S, k)$

התומך גם בפעולות הבאות:

- $\text{Select}(S,i)$ – החזרת האיבר ה- i הקטן ביותר ב- S
- $\text{Rank}(S,x)$ – החזרת הדירוג של x מבין איברי S (דירוג של איבר הוא מיקומו בסדר הממוין)

למשל:

$S = \{1, 5, 3, 6, 22, 10\}$

$\text{Select}(S,4) = 6$ $\text{Rank}(S,10) = 5$

הערה: לשם פשטות אנו מניחים כי האיברים שונים זה מזה.

פתרונות מוכרים

עץ AVL

- פעולות המילון ימומשו כרגיל.
- Select(S,i) – נבצע סריקה in-order בעץ, ונחזיר את האיבר ה- i שבו נבקר
- Rank(S,x) – באופן דומה, ע"י סריקה in-order. נספור בכמה צמתים ביקרנו עד שהגענו ל- x.

שתי הפעולות האחרונות רצות, במקרה הגרוע, כאשר $i=n$, בזמן ליניארי.

מערך

- חיפוש והוצאה יתבצע בזמן ליניארי
- הכנסה בזמן קבוע (בסוף המערך)
- Select(S,i) – נשתמש באלגוריתם Select (עם חציון החציונים), שרץ בזמן ליניארי
- Rank(S,x) – נעבור על המערך ונספור כמה איברים קטנים מ- x, בזמן ליניארי.

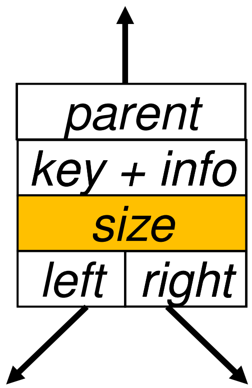
ערימה

- חיפוש, Select, Rank בזמן ליניארי...

פתרון יעיל יותר – עץ AVL מורחב

נראה כעת פתרון המבוסס על הרחבה של עצי AVL.

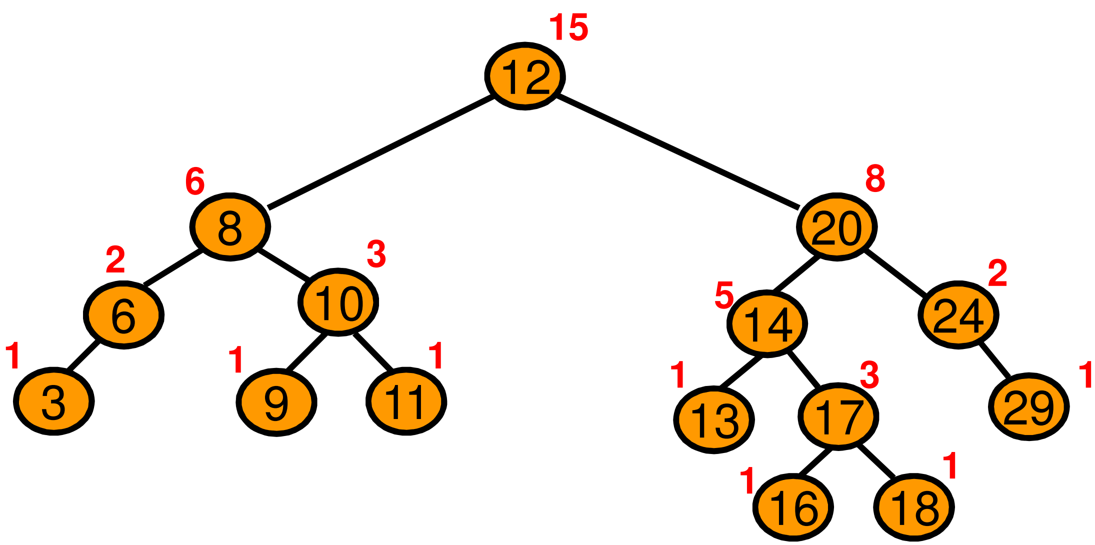
כל הפעולות ירוצו בזמן לוגריתמי.



נשתמש בעץ AVL, שבו בכל צומת נוסיף שדה אחד – ***size***.

שדה זה יחזיק את כמות הצמתים בתת העץ של הצומת (כולל הצומת עצמו).

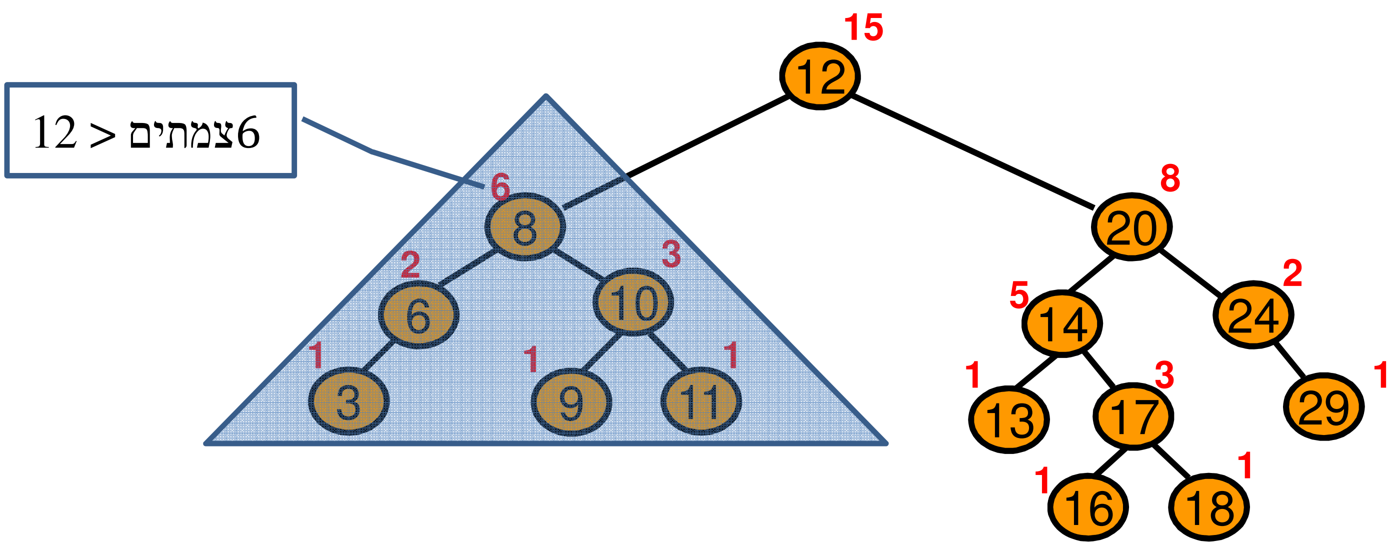
עץ כזה נקרא **עץ דרגות** (rank tree).



איך מידע נוסף זה עוזר במימוש Select ו- Rank ?

Tree-Select

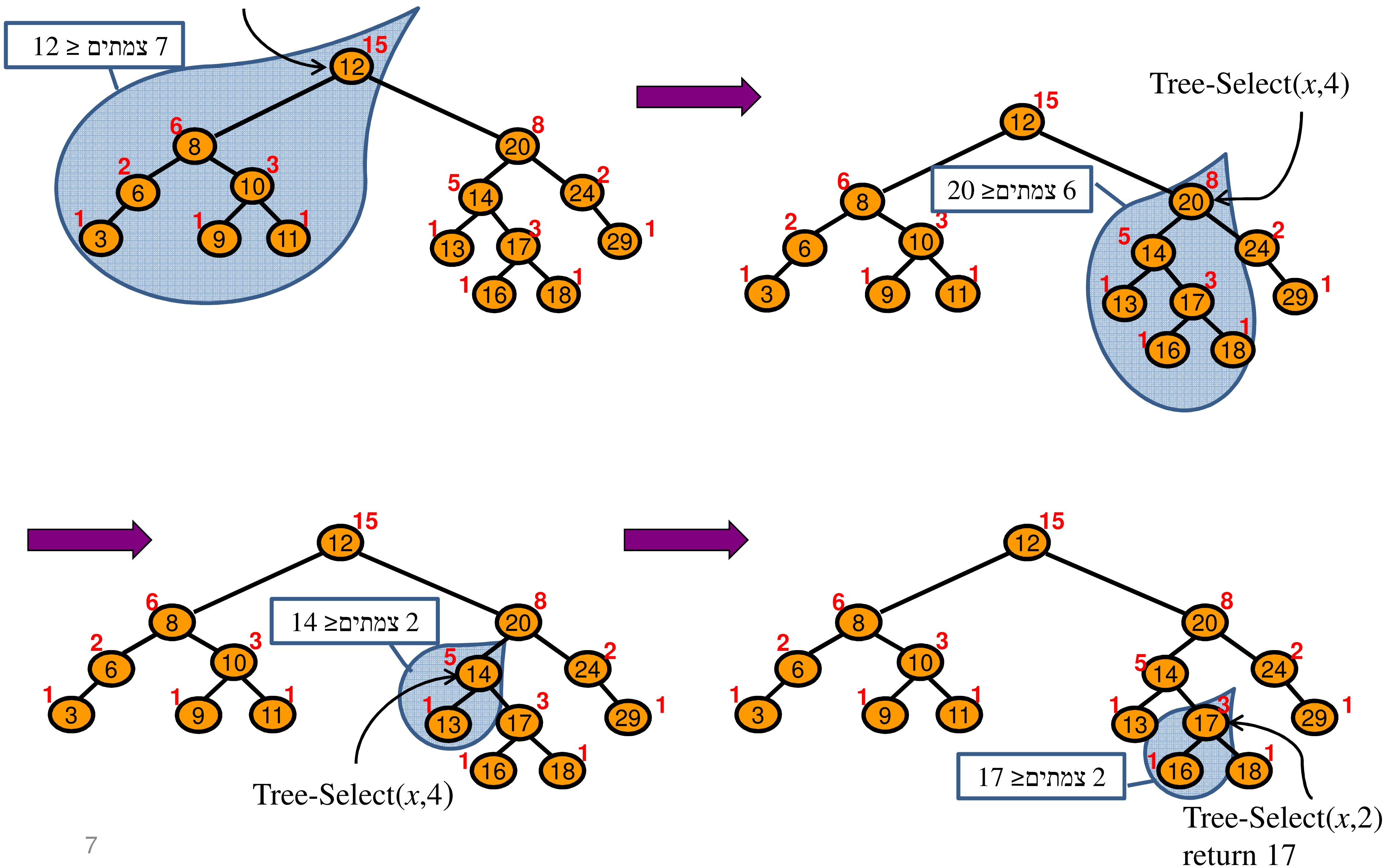
נשים לב ששורש העץ הוא האיבר ה- 7 הכי קטן.



- אם אנחנו מחפשים את האיבר ה- $i=7$ הכי קטן, הרי שזהו השורש.
- אחרת, אם $i<7$, נחפש בתת-העץ <u>השמאלי</u> של השורש <u>את האיבר ה- i הכי קטן</u>.
- אחרת ($i>7$), נחפש בתת העץ <u>הימני</u> של השורש <u>את האיבר ה- i-7 הכי קטן</u>.

Tree-Select

נדגים: Tree-Select(S,11)



Tree-Select

האלגוריתם:

Tree-Select(x, i)
1. $r \leftarrow size[left[x]] + 1$
2. **if** $i = r$
3. **return** x
4. **else if** $i < r$
5. **return** Tree-Select($left[x]$, i)
6. **else return** Tree-Select($right[x]$, $i - r$)

בקריאה הראשית x הוא השורש

סיבוכיות זמן:

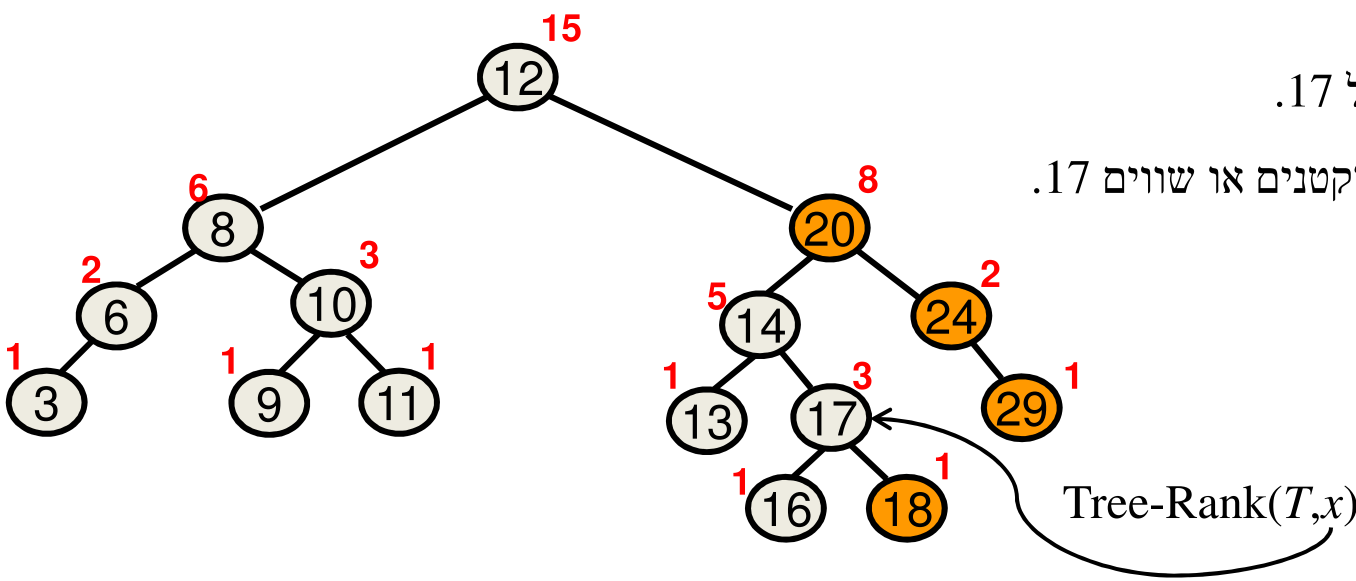
בכל רמה של העץ "מבזבזים" זמן קבוע. לכן סיבוכיות הזמן ליניארית בגובה העץ – $\Theta(\log n)$.

סיבוכיות זיכרון נוסף:

מחסנית הרקורסיה - $\Theta(\log n)$. אבל אפשר לממש גרסה איטרטיבית, בזיכרון נוסף $\Theta(1)$.

Tree-Rank

כיצד נמצא את דירוגו של איבר, בהינתן מצביע אליו?



דוגמה: נמצא את דירוגו של 17.

הצמתים הלבנים הם אלו שקטנים או שווים 17.

<u>הרעיון</u>:

נספור תחילה כמה צמתים יש בתת העץ השמאלי של 17, פלוס 1 (עבור 17 עצמו).

אח"כ נטפס מ- x עד לשורש:

בכל פעם שנעלה שמאלה לצומת, נוסיף את כמות הצמתים בתת העץ השמאלי פלוס 1.

Tree-Rank

```
Tree-Rank(T, x)
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
    if y = right[parent[y]]
        r ← r + size[left[parent[y]]] + 1
    y ← parent[y]
return r
```

<u>האלגוריתם</u>:

T הוא שורש העץ.

x מצביע לאיבר שאת דירוגו רוצים למצוא.

<u>סיבוכיות זמן</u>:

בכל רמה של העץ "מבזבזים" זמן קבוע. לכן סיבוכיות הזמן ליניארית בגובה העץ – $\Theta(\log n)$.

<u>סיבוכיות זיכרון נוסף</u>:

$\Theta(1)$.

Tree-Rank-Key

<u>שאלה</u>:

רוצים להחזיר את דירוגו של איבר בעץ דרגות, בעל <u>מפתח</u> נתון.

הציעו פתרון לבעיה זו.

<u>תשובה</u>:

אפשרות א': נמצא את האיבר, ונעביר את המצביע אליו ל- Tree-Rank.

```
Tree-Rank-Key(T, k)
1.   x ← AVL-Search(T, k)
2.   return  Tree-Rank (T, x)
```

זמן: $\Theta(\log n)$.

זיכרון נוסף: אם נשתמש בגרסה האיטרטיבית לחיפוש, $\Theta(1)$.

אפשרות ב': נאתחל $r \leftarrow 0$. נרד מהשורש במסלול החיפוש אחר המפתח, ובכל פעם <u>שנרד ימינה</u>, נוסיף ל- r את כמות הצמתים בתת-העץ השמאלי ועוד 1. נעשה זאת שוב בהגיענו לצומת המבוקש.

זמן: $\Theta(\log n)$.

זיכרון נוסף: $\Theta(1)$.

תחזוקת השדה size בעת הכנסה והוצאה

עד כה ראינו כיצד תוספת השדה *size* מאפשרת לממש את הפעולות Select ו- Rank בזמן לוגריתמי.

כעת עלינו להראות, כי בעת הכנסה או הוצאה של איברים, ניתן לעדכן את השדה הזה, מבלי לפגוע בסיבוכיות של פעולות ההכנסה וההוצאה !

תחזוקת השדה size בעת הכנסה

הכנסה לעץ AVL מורכבת משני שלבים:

שלב 1 - ירידה מהשורש כלפי מטה והכנסת צומת חדש

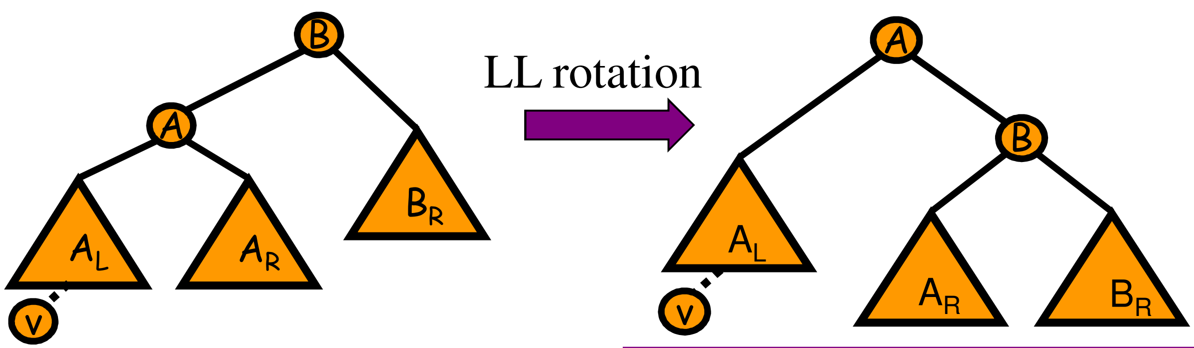
שלב 2 – עלייה מהצומת החדש לכיוון השורש כדי לאתר "עברייני AVL", ואולי ביצוע גלגול אחד.

כיצד נעדכן את *size* בכל אחד מהשלבים הללו?

שלב 1 – נוסיף 1 לשדה *size* של כל צומת שעברנו דרכו (בצומת החדש $size[z] \leftarrow 1$).

שלב 2 – עדכון $\Theta(1)$ צמתים.

למשל בגלגול LL:



$$size[A] \leftarrow size[B]$$
$$size[B] \leftarrow size[left[B]] + size[right[B]] + 1$$

בכל יתר הגלגולים מעדכנים באופן דומה.

העדכונים דורשים תוספת של קבועים בכל רמה בעץ, ולפיכך לא משנים את סיבוכיות הזמן של ההכנסה.

תחזוקת השדה size בעת הוצאה

הוצאה מעץ AVL מורכבת משני שלבים:

שלב 1 – מחיקה כרגיל מעץ חיפוש בינארי.

שלב 2 – עלייה מהצומת שנמחק פיזית לכיוון השורש כדי לאתר "עבריייני AVL", ואולי ביצוע גלגולים.

כיצד נעדכן את size בכל אחד מהשלבים הללו?

שלב 1 – כלום.

שלב 2 – תוך כדי העלייה מהצומת <u>שנמחק פיזית</u>, נחסיר 1 מהשדה *size* של כל צומת שעברנו דרכו.

אם התבצעו גלגולים תוך כדי, נעדכן $\Theta(1)$ צמתים בכל גלגול (בדיוק כמו בהכנסה).

העדכונים דורשים תוספת של קבועים בכל רמה בעץ, ולפיכך לא משנים את סיבוכיות הזמן של ההוצאה.

הרחבה של מבנה נתונים

נסכם את מה שעשינו עד כה:

ביקשנו לממש ADT, שאינו נתמך בזמן לוגריתמי ע"י אף מבנה נתונים "פשוט" אחד שמוכר לנו, או שילוב של כאלו.

לשם כך:

1. בחרנו תשתית כלשהי של מבנה נתונים מוכר — AVL
2. הרחבנו אותו ע"י תוספת כלשהי — שדה *size*
3. וידאנו שהפעולות הדינמיות לא נפגעו — הכנסה והוצאה
4. הראינו כיצד לממש את הפעולות הנוספות — Tree-Select, Tree-Rank

בעיה נוספת

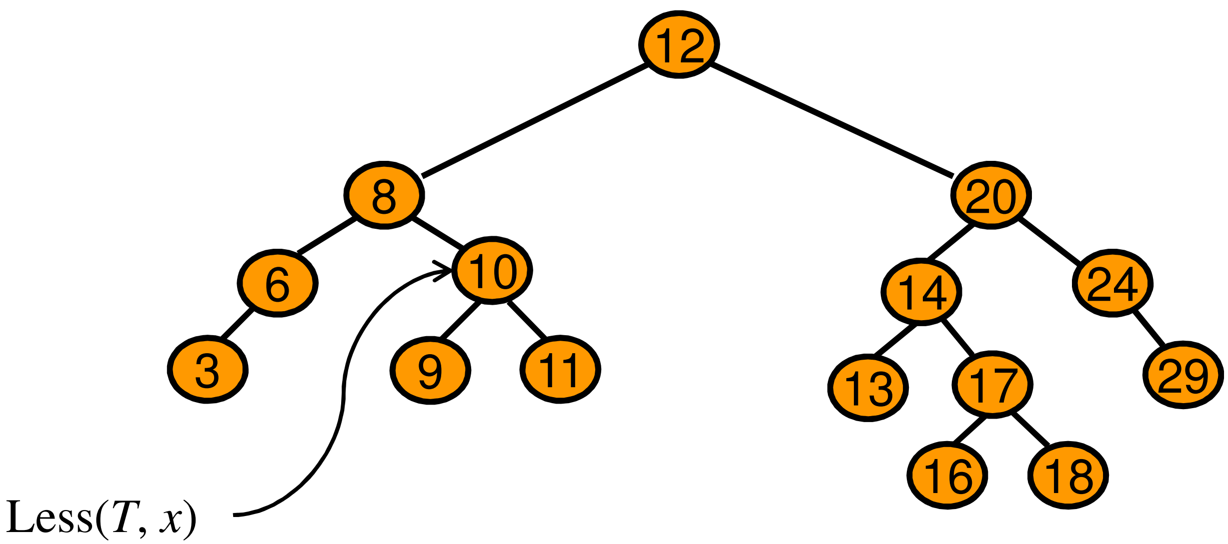
נניח שברצוננו לממש מילון ללא חזרות, התומך גם בפעולה הבאה:

- $\text{Less}(S, x)$ – החזרת סכום המפתחות שקטנים/שווים למפתח של האיבר x.

גם כאן כדאי להשתמש בעץ AVL כתשתית.

איך נממש את Less?

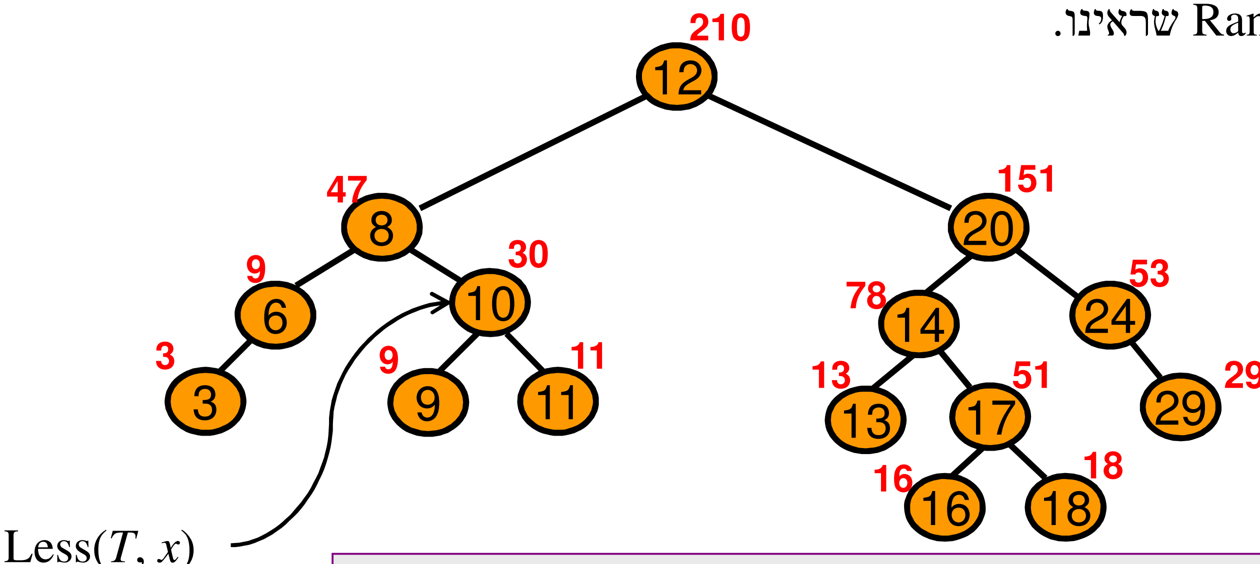
לדוגמה:



מימוש Less

הרעיון: עץ AVL שבו בכל צומת שדה נוסף – *sum*, שמכיל את <u>סכום</u> המפתחות בתת העץ של הצומת (כולל).

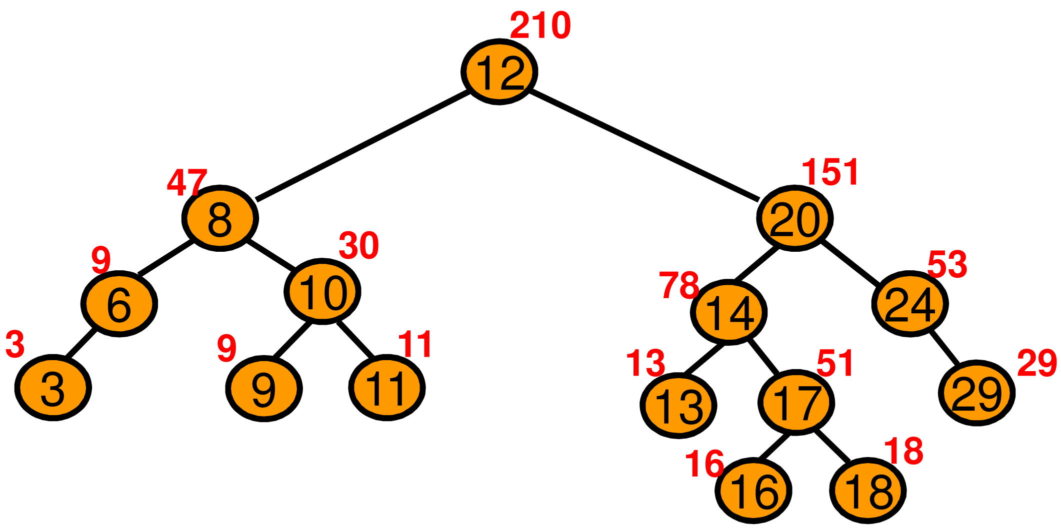
מימוש Less דומה מאוד למימוש Rank שראינו.




```
Less(T, x)
    s ← sum[left[x]] + key[x]
    y ← x
    while y ≠ root[T]
        if  y = right[parent[y]]
            s ← s + sum[left[parent[y]]] + key[parent[y]]
        y ← parent[y]
    return  s
```

תחזוקת sum

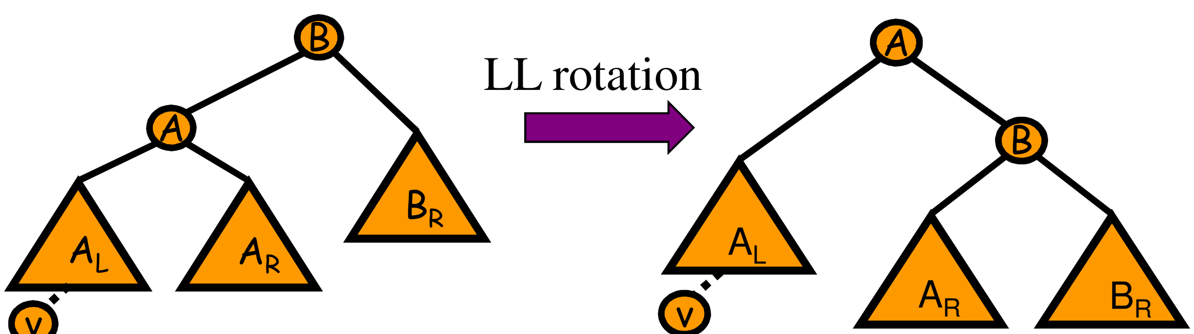
כעת עלינו להראות שניתן לתחזק את השדה *sum* מבלי לפגוע בסיבוכיות ההכנסה וההוצאה.



<u>הכנסה</u>:

שלב 1 – במהלך הירידה נוסיף את מפתח האיבר החדש לכל צומת דרכו עברנו (בצומת החדש $sum[z] \leftarrow key[z]$)

שלב 2 – נדגים על גלגול LL:



$$sum[A] \leftarrow sum[B]$$
$$sum[B] \leftarrow sum[left[B]] + sum[right[B]] + key[B]$$

תחזוקת sum

<u>הוצאה</u>:

שלב 1 – כלום

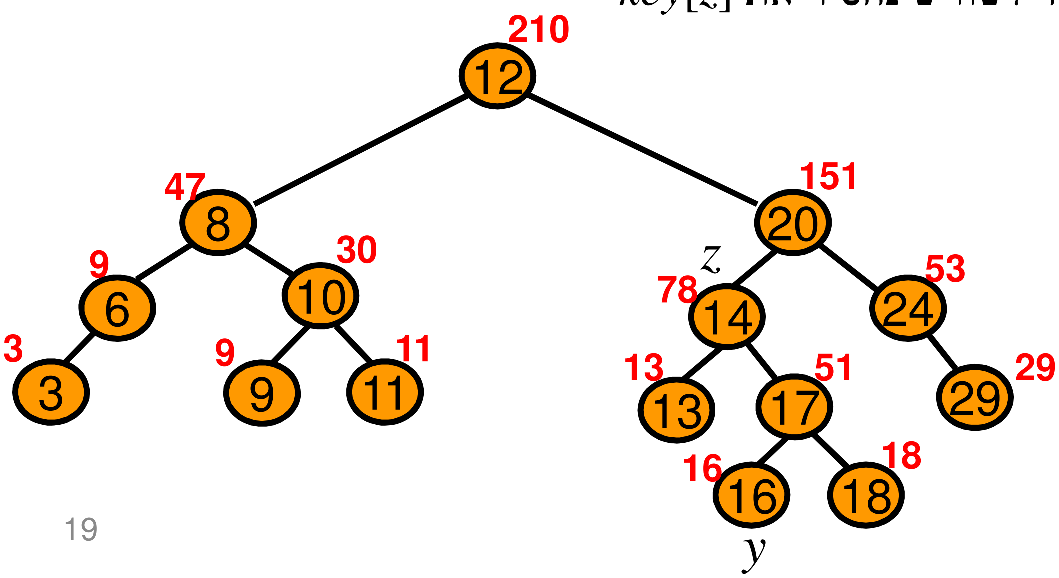
שלב 2 - **אם** לצומת שנמחק z היה לכל היותר בן אחד:

- נעלה מהצומת שנמחק עד לשורש ונחסיר $key[z]$ מהשדה sum של כל צומת

אחרת: נסמן ב- y את הצומת שנמחק פיזית (שימו לב שהמפתח של z השתנה).

- נחסיר $key[y]$ מכל הצמתים שבין y ל- z (לא כולל)

- מכל יתר הצמתים החל ב- z ועד לשורש נחסיר את $key[z]$



בגלגולים נטפל כמו בהכנסה.

מה אפשר לתחזק ביעילות?

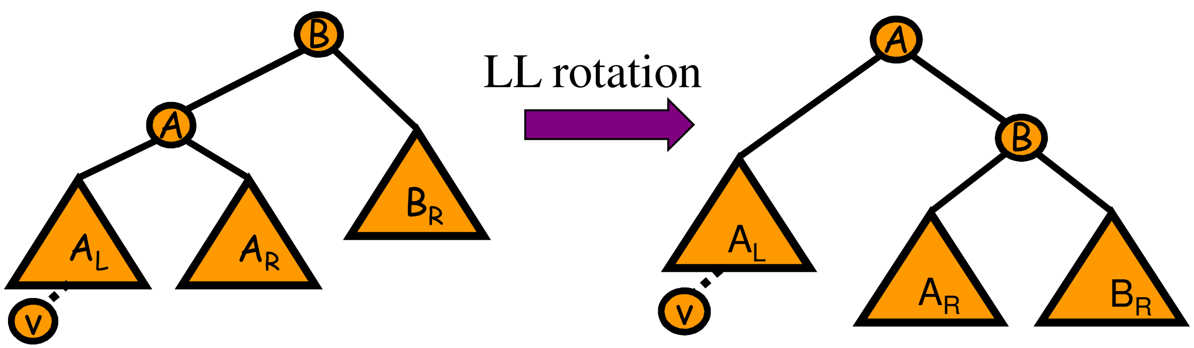
<u>שאלה</u>:

האם ניתן לתחזק ביעילות (מבלי לפגוע בסיבוכיות הזמן של הכנסה והוצאה) שדות המכילים את <u>עומקיהם</u> של צמתים בעץ AVL?

<u>תשובה</u>:

לא. ישנם מקרים (למשל גלגולים, מחיקה) שבעקבותיהם צריך לעדכן את שדות העומקים של $\Theta(n)$ צמתים !

לדוגמה, עומקיהם של אילו צמתים משתנים בגלגול LL ?



לכן סיבוכיות הכנסה/הוצאה נפגעת.

מה אפשר לתחזק ביעילות?

<u>משפט</u>

יהי f שדה המרחיב עץ AVL T בן n צמתים.

אם שינוי במידע המאוכסן בצומת מסוים (כולל שדה f שלו) משפיע רק על נכונות f של אבותיו הקדמונים

אז ניתן לתחזק את ערכי f בהכנסה והוצאה מבלי להשפיע על זמן הריצה האסימפטוטי $\Theta(\log n)$ של פעולות אלו.

<u>הערות</u>:

- המשפט מציין רק תנאי מספיק, לא תנאי הכרחי
- המשפט לא מספק אלגוריתם לעדכון השדות, רק מציין מתי ניתן לעשות זאת

סיכום

ראינו כיצד מרחיבים עצי AVL כדוגמה להרחבה של מבני נתונים.

זוהי מתודולוגיה שניתן להרחיב באמצעותה כל מבנה נתונים שלמדנו, תוך ביצוע 4 השלבים שראינו.

כדי לדעת באיזה מבנה נתונים לבחור וכיצד להרחיב אותו, נדרשת לפעמים לא מעט יצירתיות...

<u>שאלות חזרה</u>

1. הראו כיצד יש לעדכן את שדות ה- *size* של עץ דרגות בעקבות גלגול LR.
2. כיתבו גרסה לא רקורסיבית של Tree-Select.

תשובות לשאלות חזרה

2.

```
Iterative-Tree-Select(T, i)
x ← root[T]
r ← size[left[x]] + 1
while i ≠ r
      if  i < r
            x ← left[x]
      else  x ← right[x]
            i ← i - r
       r ← size[left[x]] + 1
return  x
```

<u>תרגילים</u>

<u>תרגילים</u>

1. בהינתן איבר x בעץ דרגות בעל n צמתים, ומספר טבעי i, כיצד ניתן למצוא את העוקב ה- i של x בזמן $O(\log n)$?

2. כיצד ניתן לממש מילון, שבו פעולת העוקב מתבצעת בזמן $O(1)$, וכל יתר פעולות המילון בזמן לוגריתמי במספר האיברים במבנה?

3. הציעו מימוש למבנה נתונים התומך בפעולות הבאות (n הוא מספר האיברים ברגע נתון):
   - $\text{Insert}(x)$ – הוספת האיבר x למבנה, בזמן $O(\log n)$
   - $\text{Delete}(x)$ – מחיקת האיבר x מהמבנה, בזמן $O(\log n)$
   - $\text{Find}(k)$ – מציאת איבר בעל מפתח k במבנה, בזמן $O(\log n)$
   - $\text{Min}()$ – מציאת האיבר בעל מפתח מינימלי במבנה, בזמן $O(1)$
   - $\text{Between}(k_1, k_2)$ – החזרת מספר המפתחות בין k_1 ל- k_2 (כולל), בזמן $O(\log n)$

פתרון 1

```
Tree-Successor-i(T, x, i)
1.   r ← Tree-Rank(T, x)
2.   return Tree-Select(T, r+i)
```

האם הפתרון של הפעלת Tree-Successor (החל ב- x) i פעמים גם כן נכון?

פתרון 2

נשתמש בעץ AVL מורחב, כאשר בכל צומת נשמור בנוסף גם מצביע *succ* לאיבר העוקב שלו (בצומת המקסימלי מצביע זה יכיל Nil).

פעולת העוקב ניתנת למימוש בקלות באמצעות המצביע הזה, בזמן O(1).

תחזוקת המצביע *succ* בהכנסה: נוסיף את השורות האלו בסוף אלגוריתם ההכנסה הרגיל:

$succ[z] \leftarrow$ Tree-Successor(T, z)
$pre \leftarrow$Tree-Predecessor(T, z)
$succ[pre] \leftarrow z$

(z הוא הצומת החדש)

בעת גלגולים אין צורך לבצע שום עדכון נוסף (מדוע?)

תחזוקת המצביע *succ* בהוצאה: יש להפריד למקרים.

אם נמחק צומת עם לכל היותר בן אחד – נפנה *succ* של קודמו לעוקב שלו.

אחרת?...

פתרון 3

נשתמש בעץ דרגות, כפי שראינו בהרצאה (עץ AVL עם שדה נוסף *size*).

בנוסף, נשמור מצביע לאיבר המינימלי בעץ (שיעודכן בעת הכנסה והוצאה, כפי שיוסבר).

- Insert(x) – נכניס לעץ דרגות כפי שראינו בשקפים – O(log$n$), ואם מפתח האיבר שהוכנס קטן מהמינימום נעדכן את המצביע למינימום – O(1).
- Delete(x) – נוציא מעץ דרגות כפי שראינו בשקפים – O(log$n$), ואם יש צורך נמצא את המינימום החדש ע"י קריאה ל- AVL-Minimum – O(log$n$).
- Find(k) – כרגיל בעץ AVL – O(log$n$).
- Min() – נחזיר את האיבר המינימלי בעזרת המצביע, ב- O(1).
- Between(k_1, k_2) – נשתמש בפעולה Tree-Rank שמחזירה את דירוגו של איבר נתון.
  1. איננו יודעים אם קיימים במבנה איברים בעלי המפתחות הנתונים, לכן תחילה נבדוק זאת, ואם לא – נכניס אותם (נקרא להם x_1 ו- x_2).
  2. נחשב את Tree-Rank(x_2) – Tree-Rank(x_1) + 1.
  3. נפחית מהערך שקיבלנו את כמות האיברים (בין 0 ל- 2) שהכנסנו, וזוהי התוצאה המבוקשת.
  4. לבסוף, אם יש צורך, נמחק את האיברים שהכנסנו.

כל הפעולות הנ"ל רצות בזמן O(log$n$).