האוניברסיטה הפתוחה

מבני נתונים ומבוא לאלגוריתמים
מפגש הנחיה מס' 9

מדעי המחשב, קורס מס' 20407

סמסטר 2016ב

מנחה: ג'ון מרברג

מה ראינו במפגש הקודם?

- טבלאות גיבוב (Hash Tables)
  - פעולות מילוניות
  - מיעון ישיר
  - טבלאות גיבוב
  - פתרון התנגשויות
- עצי חיפוש בינארים (מבוא)
  - סריקה של עץ בינארי
    - סריקה לרוחב
    - סריקה לעומק
  - הגדרת עץ חיפוש בינארי
  - בדיקת חוקיות עץ בינארי

מפגש תשיעי

- נושא השיעור
  - פרק 12 בספר – עצי חיפוש בינריים (המשך)
    - שאילתות: חיפוש, מינימום, מקסימום, עוקב, קודם
    - פעולות: הכנסה, מחיקה
  - פרק 13 בספר – עצים אדומים-שחורים
    - עצי חיפוש בינארי מאוזנים – מוטיבציה
    - הגדרת עץ אדום שחור
    - חסם עליון על גובה של עץ אדום שחור

מבוסס על מצגת של ברוך חייקין ואיציק בייז

עץ חיפוש בינרי (עח"ב) - הגדרה

- עץ חיפוש בינרי (Binary Search Tree) הוא עץ בינרי בו בכל צומת *x* מתקיימת התכונה הבאה (שנקראת תכונת העח"ב):
  - עבור כל צומת *y* בתת-העץ השמאלי של *x*: $key[y] \le key[x]$
  - עבור כל צומת *z* בתת-העץ הימני של *x*: $key[z] \ge key[x]$
- בניגוד לתכונת הערימה, תכונת העח"ב <u>אינה</u> התנהגות מקומית של צומת ושני בניו, אלא התנהגות יותר גלובלית
  - יש לוודא ש- $key[x]$ גדול שווה <u>מכל</u> המפתחות בתת העץ השמאלי, וקטן שווה <u>מכל</u> המפתחות בתת העץ הימני.
  - תכונת העח"ב מכתיבה סדר גלובלי בין כל המפתחות בעץ
- הגדרה אלטרנטיבית לעח"ב:

עץ בינרי שסריקה In-Order שלו מניבה סדרת מפתחות <u>ממוינת</u>

תרגיל: שחזור עץ בינרי לפי סריקות

- עץ בינרי (לאו דווקא עץ חיפוש) נהרס בטעות, אבל נשמרו שני מערכים:
  - המערך $P[1 .. n]$ המכיל את תוצאות הסריקה התחילית של העץ
  - המערך $I[1 .. n]$ המכיל את תוצאות הסריקה התוכית של העץ

  נתון גם שכל המפתחות בעץ שונים זה מזה

- כתבו אלגוריתם Tree-Restore(P, I) המשחזר את העץ המקורי,

  כלומר אלגוריתם הבונה עץ בינרי שהסריקות התחילית והתוכית שלו נתונות על ידי המערכים P ו- I, בהתאמה

- מהי סיבוכיות הזמן של האלג' במקרה הגרוע? עבור עץ בינרי כמעט שלם?

- כאשר ידוע שהעץ הוא עח"ב, ניתן לשחזר אותו באמצעות אחד בלבד משני המערכים P או I. איזה מהם?

האוניברסיטה הפתוחה

פיתרון תרגיל: שיחזור עץ בינרי לפי סריקות

- תאור האלגוריתם:
  - השורש הוא האיבר הראשון בסריקה התחילית. נחפש אותו בסריקה התוכית. לפי אינדקס השורש בסריקה התוכית נוכל לפצל את שתי הסריקות.
  - כך נקבל את הסריקות התחיליות והתוכיות של התת-עצים השמאלי והימני, ונוכל לשחזרם רקורסיבית.
- סיבוכיות הפתרון:
  - במקרה הגרוע: $O(n^2)$
    לדוגמא: עץ שהוא שרוך שמאלי
  - עבור עץ כמעט שלם: $O(n\log n)$
- עח"ב ניתן לשחזור על פי הסריקה התחילית שלו (כיצד?)

Tree-Restore(P, I)

Input: Arrays $P[1 .. n]$ and $I[1 .. n]$

Output: A tree that corresponds to P and I

```
n ← length[P]
if n = 0
   then return nil
root ← new tree node
key[root] ← P[1]
r ← Find(P[1], I)
left ← Tree-Restore(P[2..r], I[1..r–1])
right ← Tree-Restore(P[r+1..n], I[r+1..n])
left[root] ← left
right[root] ← right
p[left] ← root
p[right] ← root
return root
```

שאילתות על עח"ב: חיפוש, מינימום, מקסימום

- משפט 12.2 בספר CLRS:
  בעח"ב, הפעולות: חיפוש, מינימום, מקסימום, עוקב, וקודם
  ניתנת למימוש בסיבוכיות זמן $O(h)$

- **מינימום**

Tree-Min(x)

▶ *Input*: tree node $x \neq$ nil

while *left*[x] $\neq$ **nil**
    do x ← *left*[x]
return x

<u>הערה</u>: האלגוריתם **Tree-Max**(x)
הוא סימטרי, תוך שימוש ב- *right*[x]

- **חיפוש**

Tree-Search(x, k)

while $x \neq$ **nil and** $k \neq$ *key*[x]
    do if $k <$ *key*[x]
        then x ← *left*[x]
        else x ← *right*[x]
return x

האוניברסיטה הפתוחה

תרגיל 12.2-4: חלוקת עץ חיפוש בינרי

נניח שחיפוש מפתח k בעץ חיפוש בינרי מסתיים בעלה.

נתבונן ב-3 מפתחות:

a = מפתח כלשהו משמאל למסלול החיפוש,

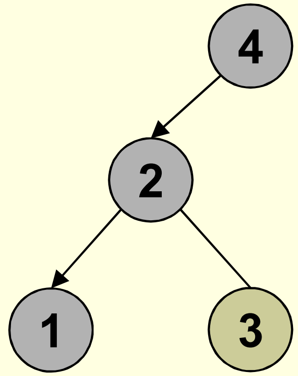
b = מפתח כלשהו השייך למסלול החיפוש,

c = מפתח כלשהו מימין למסלול החיפוש.

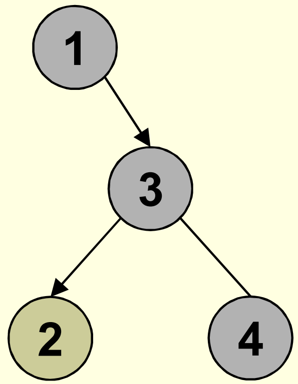
הראו (ע"י דוגמה נגדית) שלא תמיד מתקיים $a \le b \le c$.

פיתרון תרגיל 12.2-4: חלוקת עץ חיפוש בינרי

- דוגמה נגדית מימין
  - נניח שמחפשים את העלה 1
  - המפתח c=3 נמצא מימין למסלול החיפוש, אבל הוא קטן מהמפתח b=4 הנמצא על מסלול החיפוש.



- דוגמה נגדית משמאל
  - נניח שמחפשים את העלה 4
  - המפתח a=2 נמצא משמאל למסלול החיפוש, אבל הוא גדול מהמפתח b=1 הנמצא על מסלול החיפוש.



האוניברסיטה הפתוחה

שאילתות בעח"ב: עוקב וקודם

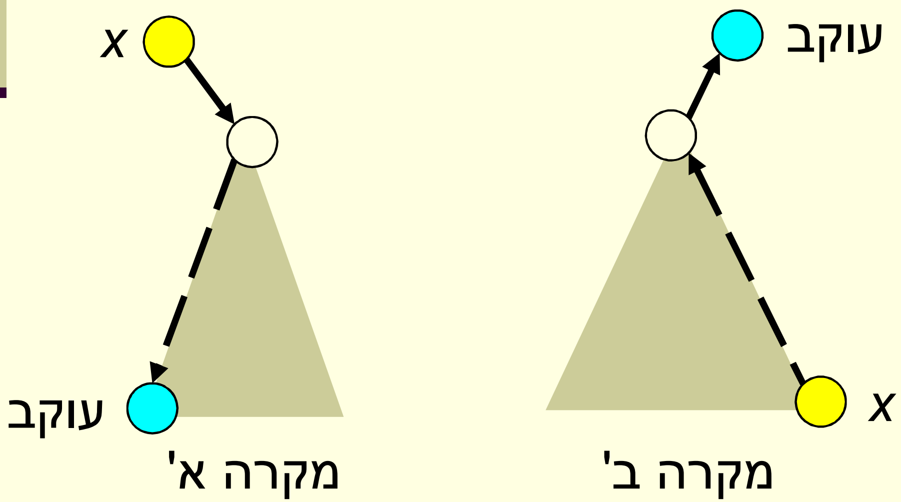
- **עוקב** (successor)

Tree-Successor(x)

```
1. if right[x] ≠ nil
2.     then return Tree-Min(right[x])
3. y ← p[x]
4. while y ≠ nil and x = right[y]
5.     do x ← y
6.        y ← p[y]
7. return y
```

- המימוש של השגרה למציאת הקודם (predecessor) סימטרי

- מקרה א' (שורה 2): ל-x יש בן ימני
  - העוקב הוא המינימום של תת-העץ הימני של x
- מקרה ב' (שורות 3-7): ל-x אין בן ימני
  - העוקב (אם קיים) הוא האב הקדמון y הקרוב ביותר ל- x, כך ש- x נמצא בתת העץ השמאלי של y



תרגיל 12.2-7: מימוש אחר של סריקה תוכית בעח"ב

ניתן לממש סריקה תוכית של עץ חיפוש בינרי בעל n צמתים
על-ידי מציאת האיבר המינימלי בעץ בעזרת Tree-Min,
ואז ביצוע $n-1$ קריאות ל- Tree-Successor.

הוכיחו שזמן הריצה של אלגוריתם זה הוא $\Theta(n)$.

פיתרון תרגיל 12.2-7:
מימוש אחר של סריקה תוכית בעח"ב

- החסם התחתון הוא לינארי
  - ברור שכדי להדפיס מפתחות של n צמתים יש צורך ב-$\Omega(n)$ פעולות
- החסם העליון הוא לינארי
  - ניתן להראות שכל קשת בעץ נסרקת לכל היותר פעם אחת לכל כיוון
    - הקשת מצומת נתון לבן השמאלי שלו נסרקת
      - בכיוון מטה ע"י Tree-Min,
      - בכיוון מעלה ע"י האיטרציה האחרונה של הלולאה בשורות 4-6 של Tree-Successor המופעל על הצומת המקסימאלי בתת העץ של הבן
    - הקשת מצומת נתון לבן הימני שלו נסרקת
      - בכיוון מטה ע"י שורות 1-2 של Tree-Successor,
      - בכיוון מעלה ע"י איטרציה לא-אחרונה של הלולאה בשורות 4-6 של Tree-Successor המופעל על הצומת המקסימאלי בתת העץ של הבן
  - מספר הקשתות בעץ כלשהו בגודל n הוא $n - 1$, ולכל קשת מתבצע מספר קבוע של פעולות, לכן מספר הפעולות הוא $O(n)$
  - משתי הטענות הקודמות נובע שזמן הסריקה הוא $\Theta(n)$

תרגיל: חיפוש איבר קרוב

- הפעולה Tree-FindClose(x, k) על עץ חיפוש בינארי מוגדרת כך:
  - אם קיים בעה"ב ששורשו x איבר בעל מפתח k, מוחזר איבר זה
  - אחרת, מוחזר איבר בעץ שהמפתח שלו הוא הקטן ביותר מבין כל המפתחות הגדולים מ- k, או nil אם לא קיים איבר כזה.

- כתוב אלגוריתם עבור Tree-FindClose אשר מתבצע בזמן $O(h)$ במקרה הגרוע (h הוא גובה העץ)

פתרון תרגיל: חיפוש איבר קרוב

הרעיון:

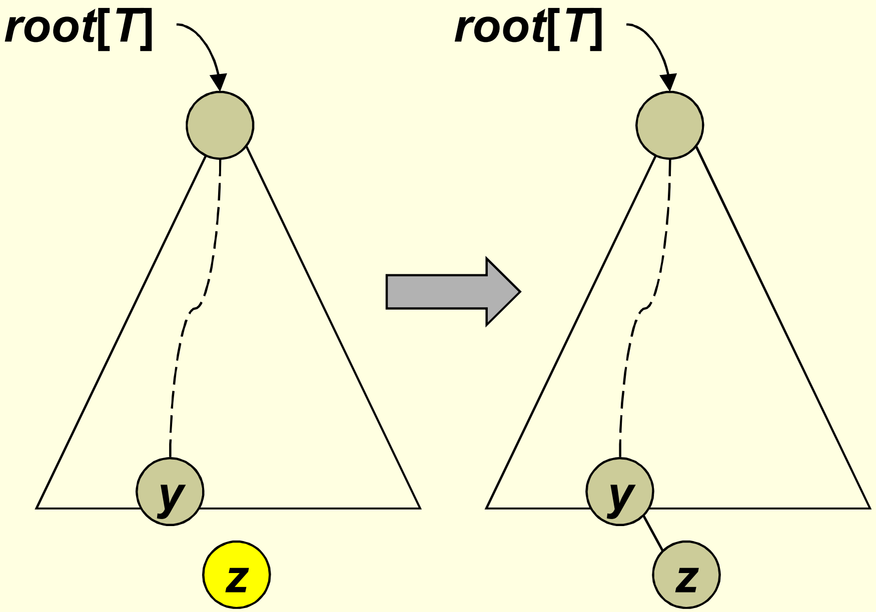
- מחפשים את *k* בעץ בדומה לאלגוריתם החיפוש Tree-Find (שורות 2-6)
- אם *k* לא נמצא, בודקים את הצומת האחרונה *y* שלפני הכישלון
  - אם *k* קטן מהמפתח של *y* (כלומר איבר שמפתחו *k* צריך היה להיות בן שמאלי של *y*) אז האיבר הקרוב הוא *y*
  - אחרת האיבר הקרוב הוא העוקב של *y*
  - נכונות שתי הטענות לעיל מבוססות על אלגוריתם העוקב Tree-Successor
- זמן ריצה: $O(h)$ במקרה הגרוע

Tree-FindClose(x, k)

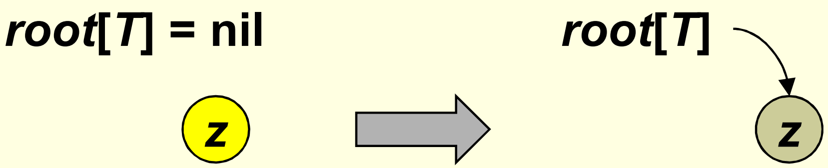
1. $y \leftarrow$ **nil**
2. **while** $x \neq$ **nil and** $k \neq key[x]$
3.   **do** $y \leftarrow x$
4.   **if** $k < key[x]$
5.   **then** $x \leftarrow left[x]$
6.   **else** $x \leftarrow right[x]$
7. **if** $x \neq$ **nil or** $y =$ **nil**
8.   **then return** x
9. **if** $k < key[y]$
10.   **then return** y
11. **return** Tree-Successor(y)

האוניברסיטה הפתוחה

הכנסת צומת חדש



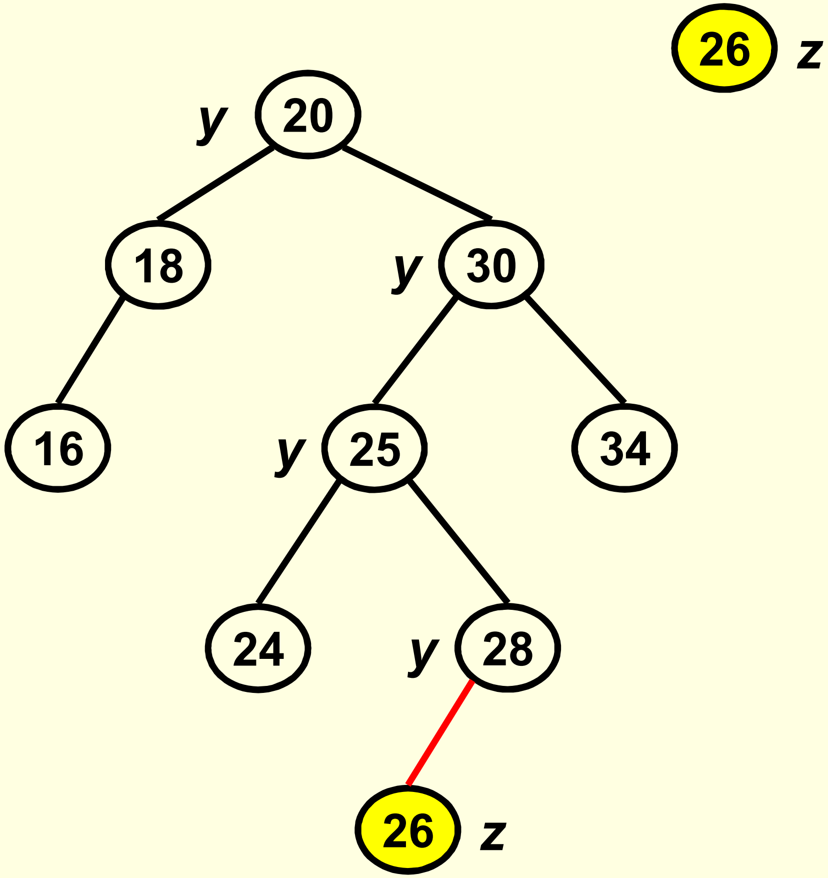
- הכנסה לעץ לא ריק
  - הצומת החדש z נכנס כעלה תחת אב כלשהו y
  - ל-y יכולים להיות 0 או 1 בנים



- הכנסה לעץ ריק
  - z נכנס כשורש

הכנסה – דוגמה

- נוסיף לעץ את הצומת z (מפתח 26)



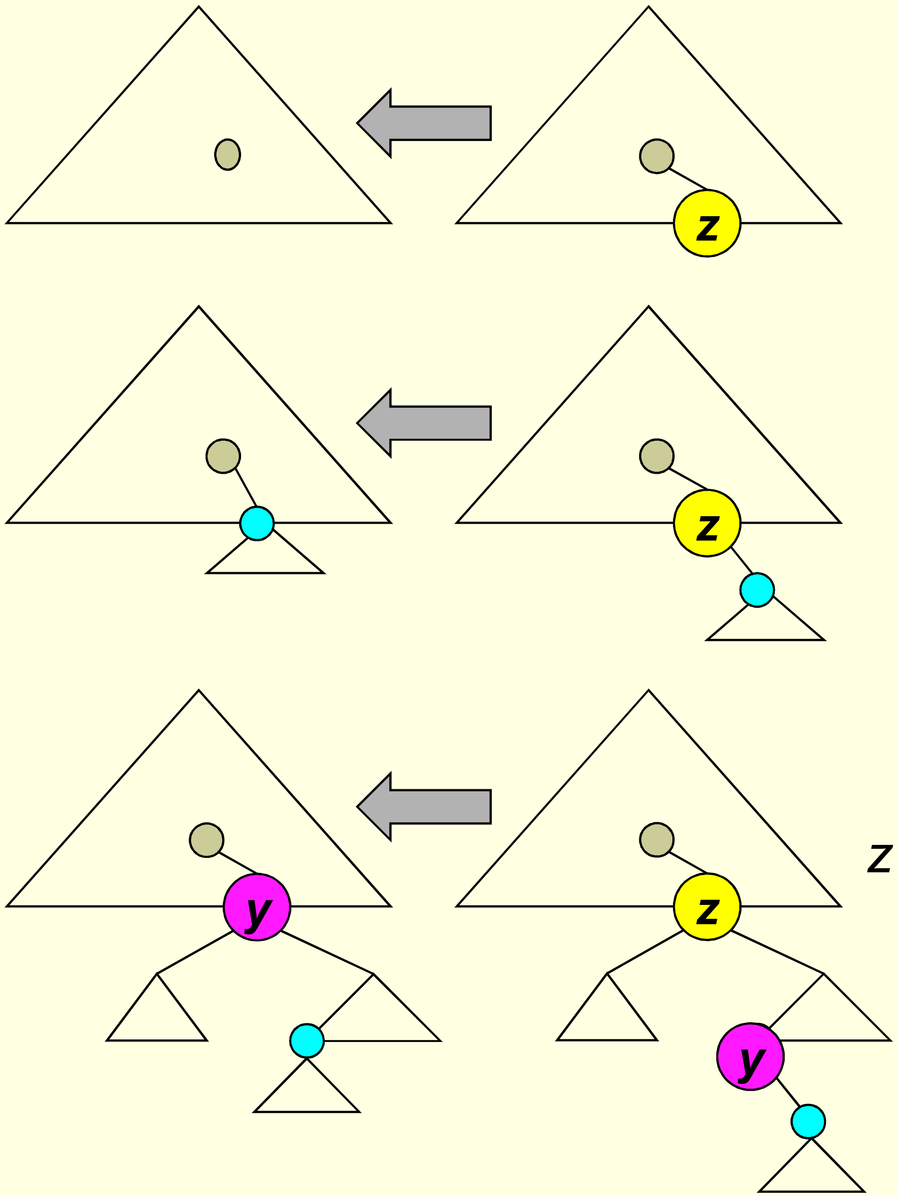
הכנסה – האלגוריתם

- מכניסים את הצומת z כעלה של T
  - שורות 1-7: מחפשים אב y ל-z כך שתישמר תכונת עץ החיפוש הבינרי
  - שורות 8-13: מציבים את z כבן שמאלי או ימני של y
  - שורה 10: טיפול במקרה הקצה ש-z הוכנס לעץ ריק
- זמן ריצה: $O(h)$ במקרה הגרוע

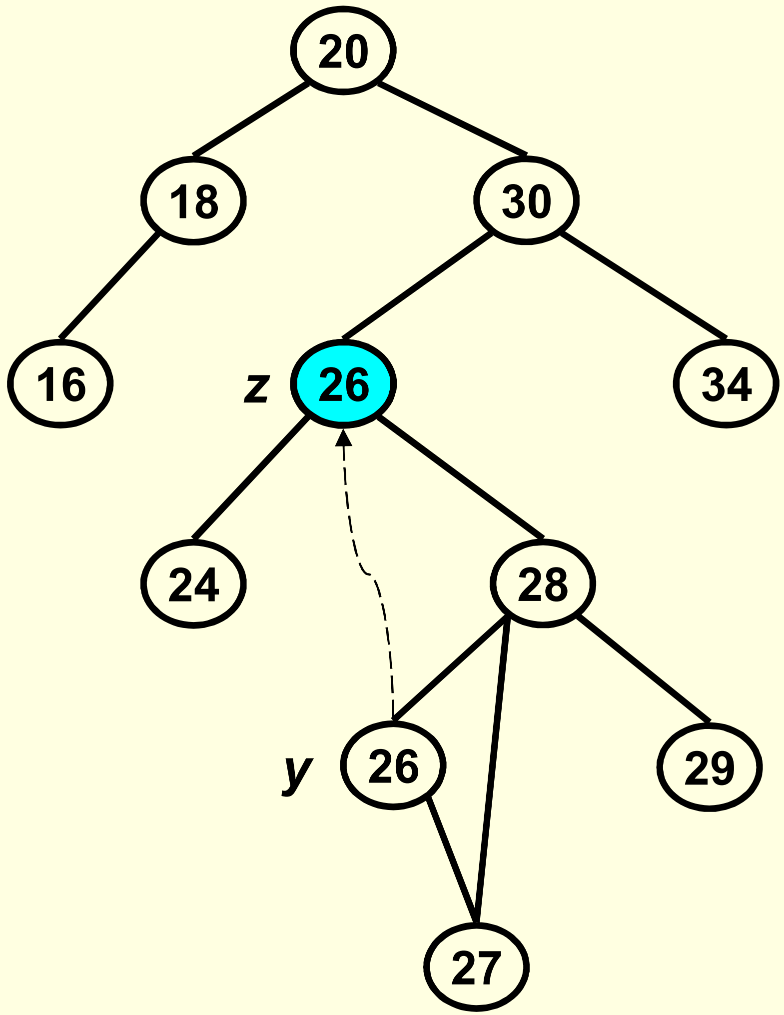
```
Tree-Insert(T, z)
1.  y ← nil
2.  x ← root[T]
3.  while x ≠ nil
4.     do y ← x
5.        if key[z] < key[x]
6.          then x ← left[x]
7.          else x ← right[x]
8.  p[z] ← y
9.  if y = nil
10.   then root[T] ← z
11.   else if key[z] < key[y]
12.          then left[y] ← z
13.          else right[y] ← z
```

מחיקת צומת קיים

- נסמן ב- *z* את הצומת שיש למחוק
- **מקרה א': *z* הוא עלה**
  - נמחק אצל אביו של z את המצביע אל *z*
- **מקרה ב': ל-*z* יש בן אחד**
  - נחליף אצל אביו של z את המצביע אל *z* במצביע אל בנו של *z*
- **מקרה ג': ל-*z* יש שני בנים**
  - נסמן ב- y את הצומת העוקב של z
  - נעתיק את תכולת הצומת *y לתוך הצומת z (למעט המצביעים)*
  - נמחק את הצומת *y מהעץ*
    - ל- *y* יש לכל היותר בן אחד (מדוע?), ולכן מחיקתו שייכת למקרה א' או ב'



מחיקה – דוגמה



- מחיקת הצומת z (מפתח 25)
  - ל-z יש שני בנים (מקרה ג'), לכן מחליפים את תכולתו בתכולת העוקב y שלו (מפתח 26)
  - ל-y יש בן אחד (מקרה ב'), ולכן מחברים את הבן שלו לאביו
  - לבסוף מוחקים את y
- ניתן היה להשתמש **בקודם** במקום בעוקב (תרגיל 12.3-6)

האוניברסיטה הפתוחה

מחיקה – האלגוריתם

```
Tree-Delete(T, z)
► y denotes the node to actually delete
if left[z] = nil or right[z] = nil
    then y ← z
    else y ← Tree-Successor(z)
if left[y] ≠ nil
    then x ← left[y]
    else x ← right[y]
if x ≠ nil ► case b
    then p[x] ← p[y]
if p[y] = nil
    then root[T] ← x
    else if y = left[p[y]]
            then left[p[y]] ← x
            else right[p[y]] ← x
if y ≠ z ► case c
    then key[z] ← key[y]
            ►copy y's satellite data to z ...
return y ►for recycling
```

- זמן ריצה: $O(h)$ במקרה הגרוע

עצי חיפוש בינריים שנבנים אקראית

- הגדרה: עץ חיפוש בינרי שנבנה אקראית
  - עץ המתקבל ע"י הכנסת *n* מפתחות שונים בסדר אקראי לעץ ריק
  - הנחה: לכל אחת מ- *n*! התמורות של המפתחות סיכוי שווה
  - לא מתבצעות מחיקות (אחרת קשה לנתח את המבנה)

- משפט 12.4: תוחלת הגובה של עץ חיפוש בינרי שנבנה אקראית עם *n* מפתחות היא $O(\lg n)$
  - משמעות: זמן הריצה של הפעולות היסודיות הוא *לוגריתמי בממוצע* (למרות שהוא לינארי במקרה הגרוע)
  - לא ידוע אם זה המצב גם כאשר מתבצעות מחיקות אקראיות מהעץ

תרגיל 12.4-3:
"בניה אקראית" לעומת "בחירה אקראית"

- הגדרה: עץ חיפוש בינרי <u>שנבחר</u> אקראית
  - עץ המתקבל ע"י בחירה אקראית מבין כל העח"ב האפשריים עם n מפתחות
  - כל אחד מהעח"ב האפשריים הוא בעל סיכוי שווה להיבחר

- הוכח שהמושג "עץ חיפוש בינרי <u>שנבחר</u> אקראית" שונה מהמושג "עץ חיפוש בינרי <u>שנבנה</u> אקראית"

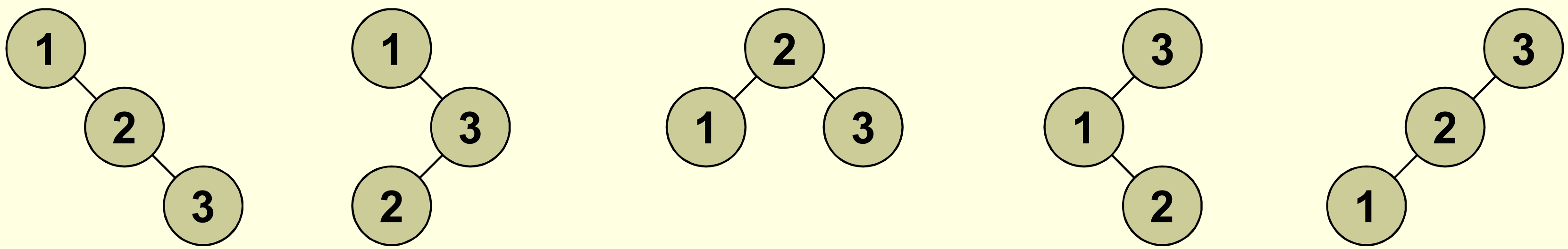
רמז: חשב את הסיכויים של כל האפשרויות עבור $n = 3$.

פיתרון תרגיל 12.4-3:
"בניה אקראית" לעומת "בחירה אקראית"

נתבונן בעצים הנבנים ע"י סדרת קלט בגודל 3 (יש 6 = !3 סדרות כאלה).

עבור סדרת הקלט:

<1,2,3> <1,3,2> <2,1,3> and <2,3,1> <3,1,2> <3,2,1>



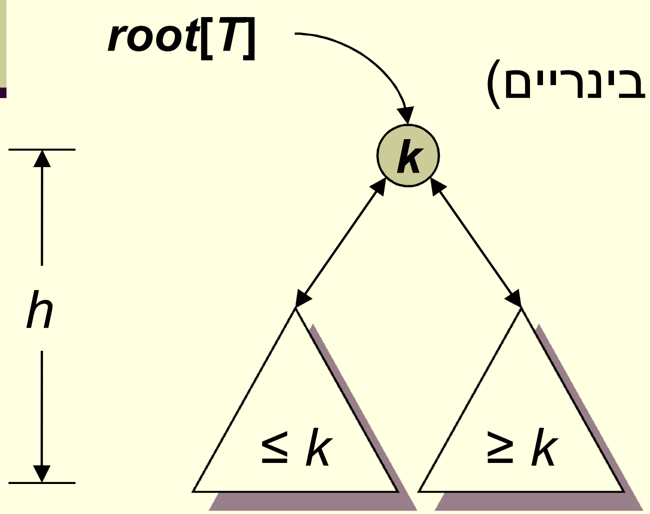
קיימים בדיוק 5 עצים שונים בגודל 3.

הסיכוי לבנות את העץ האמצעי הוא 1/3, והסיכוי לבנות כל אחד מהעצים האחרים הוא 1/6.

לעומת זאת, הסיכוי <u>לבחור</u> עץ מסוים הוא 1/5.

עצי חיפוש מאוזנים

- עץ חיפוש בינארי של n צמתים יכול להיות בגובה $h=O(n)$
  - לכן זמן הריצה של הפעולות על ע"חב לינארי במספר הצמתים במקרה הגרוע $O(h)=O(n)$
- דרוש עץ חיפוש "מאוזן" (balanced search tree)
  - רצוי שגובה העץ יהיה $O(\lg n)$
  - לשם כך יש "לאזן" את העץ
    גובה שני התת-עצים של כל צומת צריך להיות "בערך" שווה
- עץ אדום-שחור (Red-Black Tree) הוא סוג של עץ חיפוש בינרי מאוזן
  - האיזון מושג על-ידי כללים של צביעת צמתים וביצוע פעולות איזון מקומיות
    על העץ (רוטציות) בעת הכנסה ומחיקה
  - יש סוגים נוספים של עצי חיפוש מאוזנים (לאו דווקא בינריים)



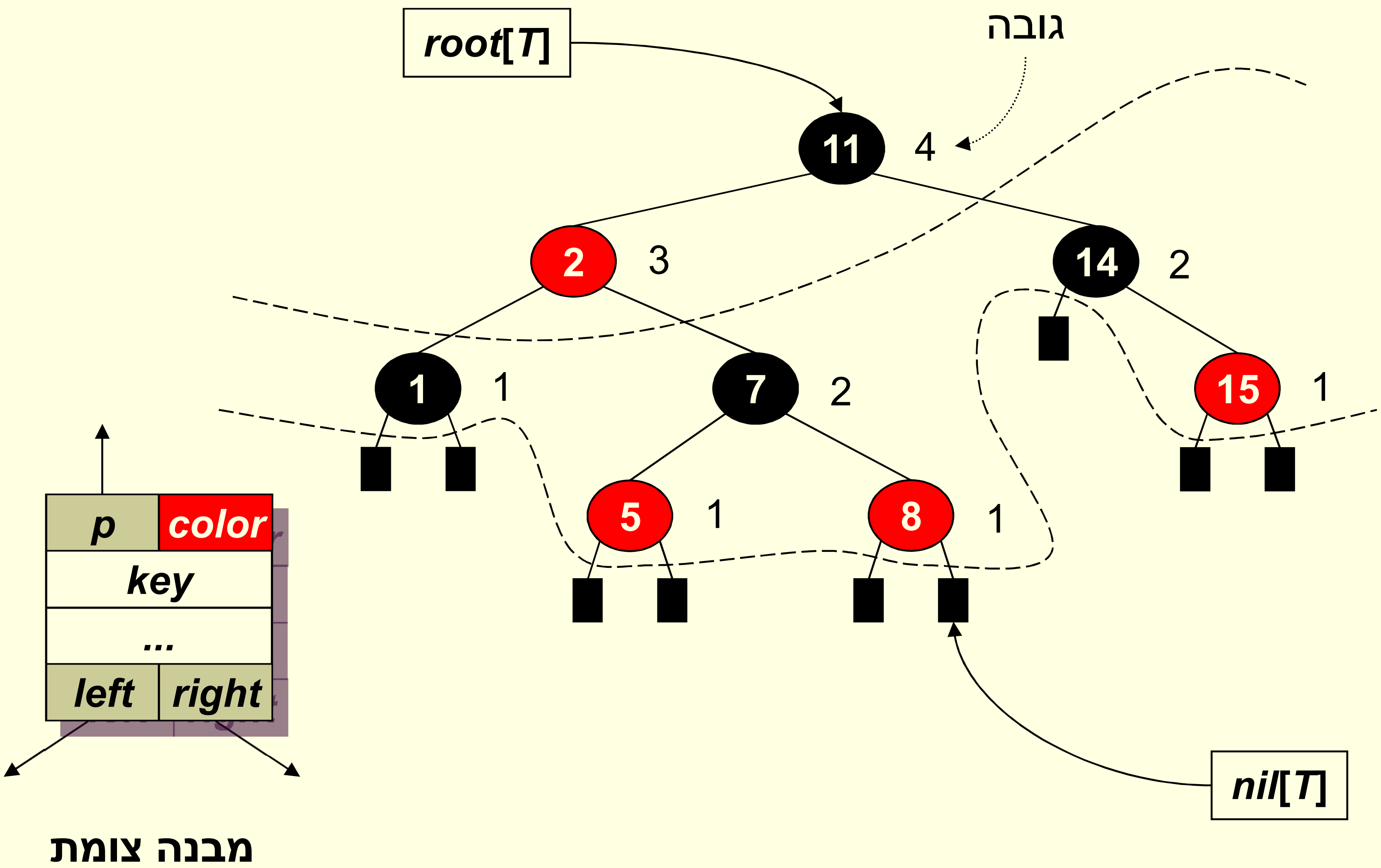
האוניברסיטה הפתוחה

עצים אדומים-שחורים

- **עץ אדום-שחור** הוא עץ חיפוש בינרי בעל התכונות הבאות:

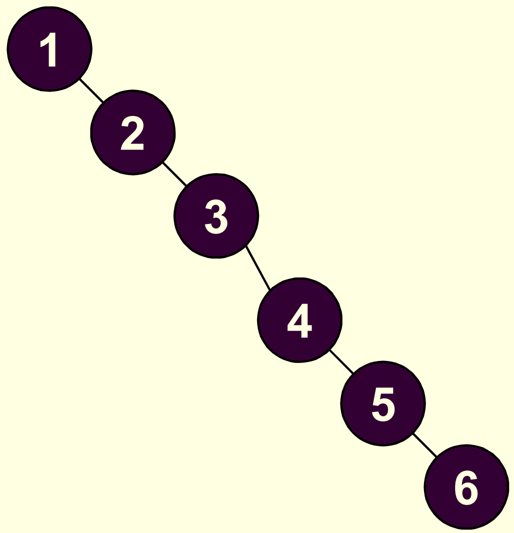
1. כל צומת הוא אדום או שחור
   - לכל צומת x בעץ נוסיף שדה $color[x]$, שערכו BLACK או RED
2. השורש הוא שחור
3. כל עלה הוא שחור
   - נתייחס לכל מצביע nil כאילו הוא מצביע לצומת מיוחד $nil[T]$ שצבעו שחור
   - כל הצמתים "הרגילים" נחשבים כצמתים פנימיים ויש להם שני בנים
4. אם צומת הוא אדום, אז שני בניו שחורים
5. לכל צומת x, כל המסלולים מ- x לצאצאים-עלים שלו מכילים את אותו מספר של צמתים שחורים (x עצמו לא נכלל בספירה זו)
   - לכל צומת x בעץ, נגדיר את $bh(x)$, **"הגובה השחור" של x**, כמספר הצמתים השחורים (לא כולל x עצמו) בכל מסלול מ-x לעלה
   - **"הגובה השחור" של עץ אדום-שחור** הוא הגובה השחור של השורש

עץ אדום-שחור – דוגמה



תרגיל: מבנה של עץ אדום-שחור

- האם העץ שלהלן הוא עץ אדום-שחור?

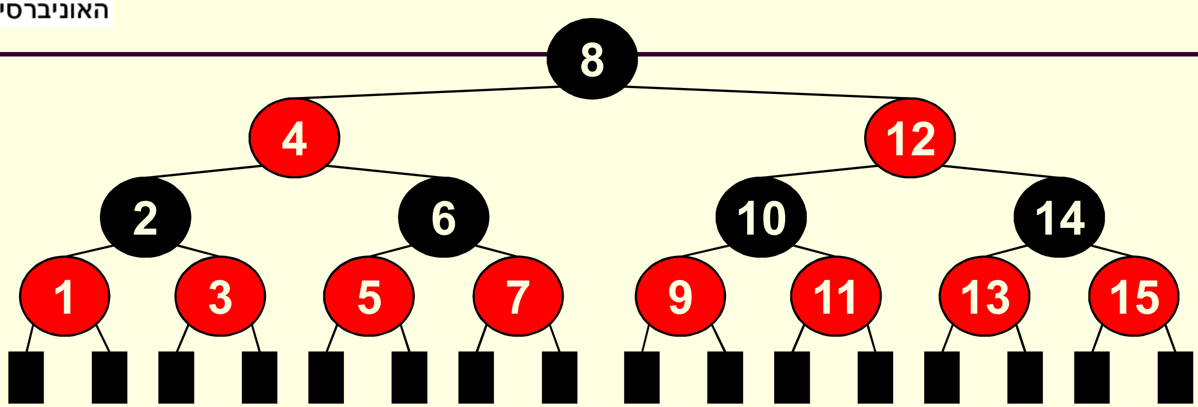


תרגיל 13.1-1:
צביעה של עצים אדומים-שחורים

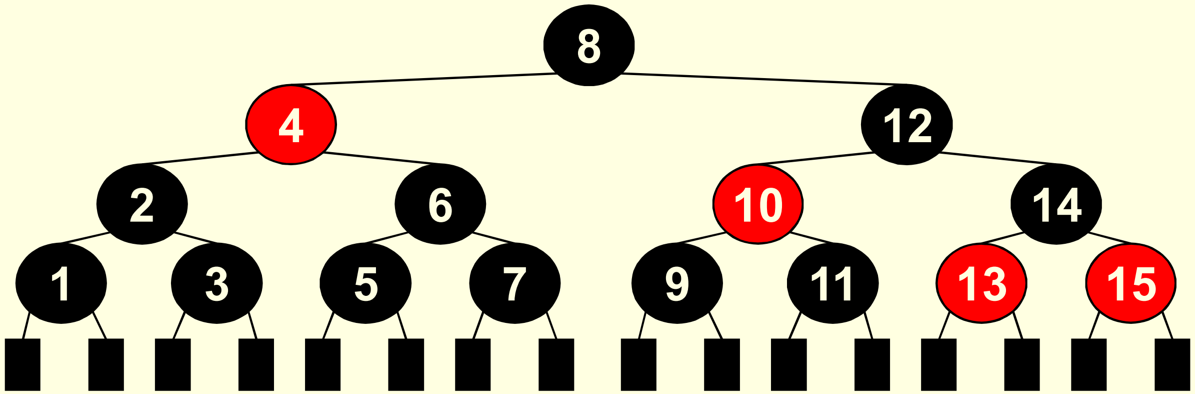
שרטטו את עץ החיפוש הבינרי השלם בגובה 3 על קבוצת המפתחות {1, 2, ..., 15}.

הוסיפו את העלים nil וצבעו את הצמתים בשלוש דרכים שונות, כך שיתקבל גובה שחור אחר בכל פעם: 2, 3, 4

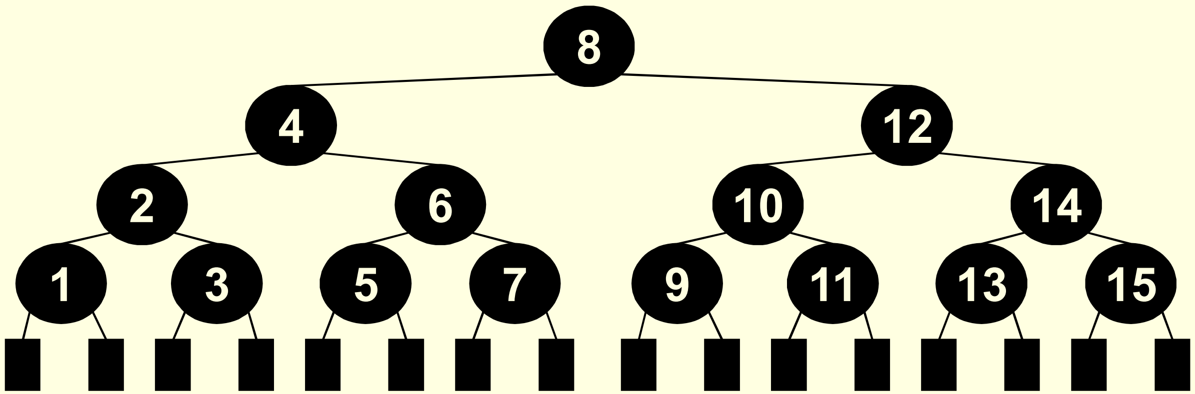
פתרון תרגיל 13.1-1:
צביעה של עצים אדומים-שחורים



גובה-שחור 2
(מינימלי)



גובה-שחור 3
(יש עוד אפשרויות)



גובה-שחור 4
(מקסימלי)

גובה של עץ אדום-שחור

- **למה 13.1**: גובהו של עץ אדום-שחור המכיל n צמתים פנימיים הוא לכל היותר $2\lg(n + 1)$
- **משמעות:** העץ שומר על איזון טוב, כי הגובה המינימאלי של עץ המכיל n צמתים הוא $\lfloor \log n \rfloor$
- **מסקנה:** אם נשכיל לשמור על תכונות העץ אדום-שחור מובטח שכל הפעולות: הכנסה, מחיקה, חיפוש, מינ', מקס', עוקב וקודם, יתבצעו בזמן $O(\log n)$ במקרה הגרוע

ההוכחה נמצאת בספר הלימוד

הוכחת למה 13.1

- **למה 13.1**: גובהו של עץ אדום-שחור המכיל n צמתים פנימיים הוא לכל היותר $2\lg(n+1)$
  - **טענת עזר**: התת-עץ המושרש בצומת x כלשהו מכיל לפחות $2^{bh(x)} - 1$ צמתים פנימיים. הוכחה באינדוקציה על h, הגובה של x
    - <u>בסיס</u>: עבור $h = 0$, x הוא צומת nil, ואכן $2^{bh(\mathrm{nil})} - 1 = 2^0 - 1 = 0$
    - <u>צעד</u>: x הוא צומת פנימי (יש לו שני בנים). אם הבן אדום, אז גובה השחור שלו נשאר $bh(x)$. אחרת (הבן שחור), גובה השחור שלו הוא $bh(x) - 1$.
    - גובה הבן קטן מגובה האב, לכן עפ"י הנחת האינדוקציה בתת-עץ המושרש בבן יש <u>לפחות</u> $2^{bh(x)-1} - 1$ צמתים פנימיים.
    - סה"כ בתת-עץ המושרש ב-x יש לפחות $2\cdot(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ צמתים פנימיים.
  - מתכונה 4 נובע שלפחות מחצית מהצמתים בכל מסלול בין השורש לעלה (לא כולל השורש) הם שחורים
    - כלומר, $bh(root[T]) \geq h/2$
  - לכן לפי טענת העזר, עבור השורש מתקיים $n \geq 2^{bh(\mathrm{root}(T))} - 1 \geq 2^{h/2} - 1$,
    ומכאן $h \leq 2\lg(n+1)$

תרגיל: "עץ אדום-שחור" לעומת "עץ מאוזן-משקל"

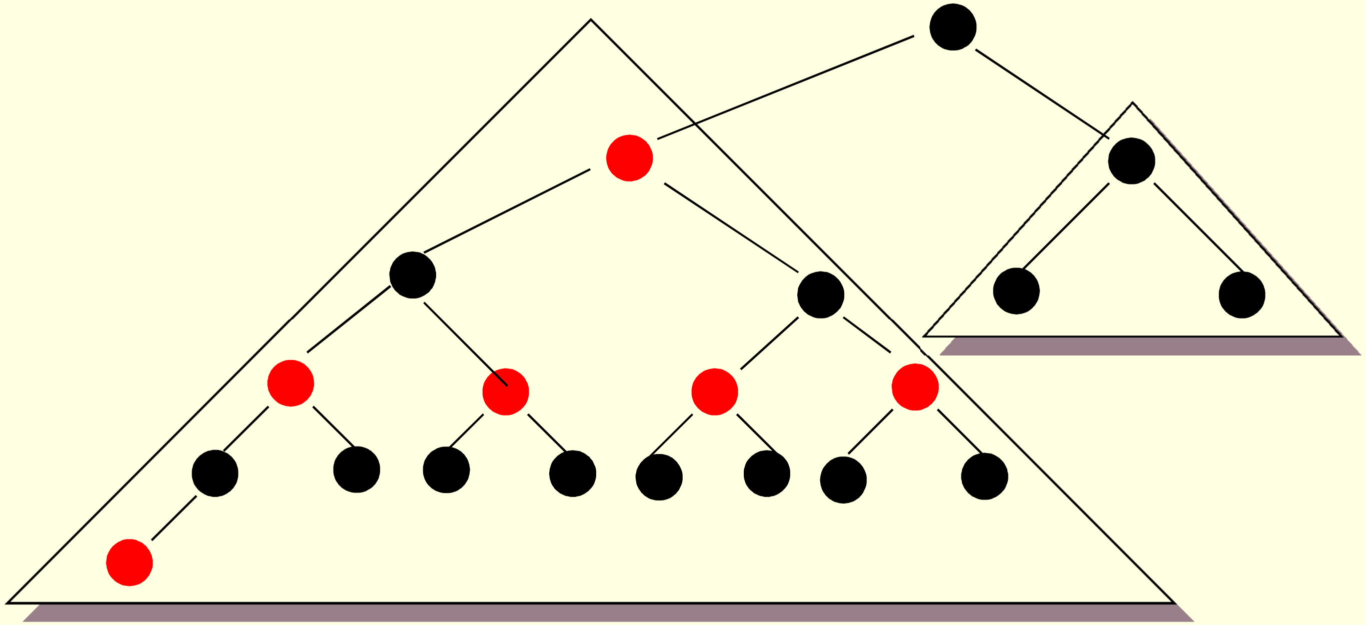
- הגדרה: עץ מאוזן-משקל הוא עץ בינרי שמקיים את התכונה הבאה:
  - מספר הצמתים בתת-עץ אחד של השורש הוא לכל היותר פי שניים ממספר הצמתים בתת-עץ השני של השורש.
  - כלומר בעץ מאוזן-משקל מתקיים $½ \le |T_L|/|T_R| \le 2$

$|T_L|$ ו- $|T_R|$ מציינים את מספר הצמתים בתת-העץ השמאלי והימני של השורש

- האם כל עץ אדום-שחור הוא גם עץ מאוזן-משקל?
  הוכיחו או תנו דוגמה נגדית.

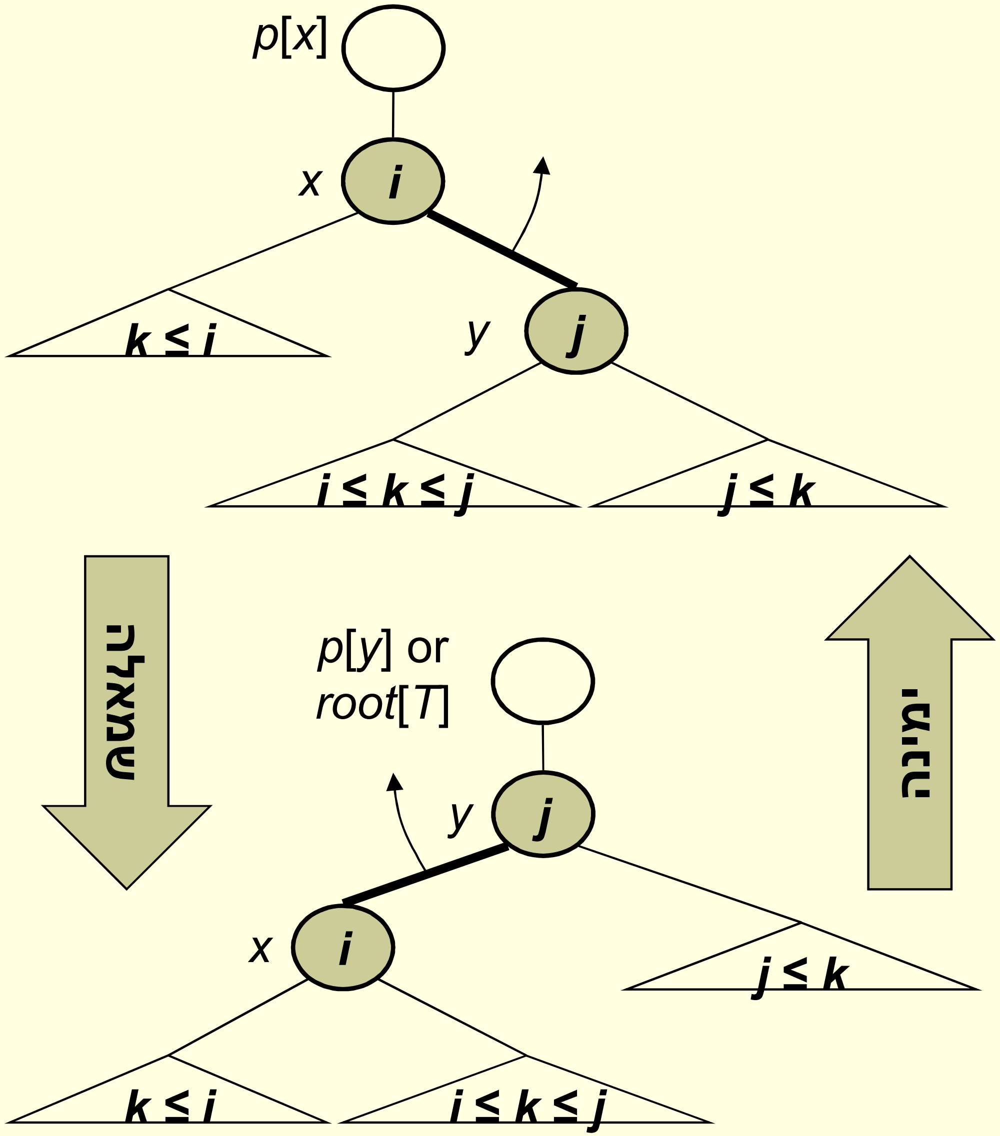
פתרון תרגיל: "עץ אדום-שחור" מול "עץ מאוזן-משקל"

דוגמה נגדית



$|T_L|/|T_R| = 16/3 > 2$

פעולות עזר: סיבוב שמאלי וימני



- סיבוב (Rotation) – פעולה <u>מקומית</u> בעץ בינרי, המשנה את המבנה של תת-עץ
  - מתבצע בזמן $O(1)$
  - נשמר הסדר התוֹכי של המפתחות!
- סיבוב שמאלי (Left-Rotate)
  - מתבצע על שורש התת-עץ x שבנו הימני y אינו nil
  - הופך את y לשורש החדש של התת-עץ, ואת x לבנו השמאלי
  - "מסובב שמאלה" (נגד כיוון השעון) את הקשת $x – y$
- סיבוב ימני (Right-Rotate)
  - כנ"ל, כאשר שמאל וימין מתחלפים

האוניברסיטה הפתוחה

סיבוב שמאלי – האלגוריתם

- אלגוריתם לסיבוב שמאלי
  - לפני: הצומת *x* הוא שורש התת-עץ, הצומת *y* הוא הבן הימני שלו (שורה 1)
  - אחרי: הצומת *y* הוא שורש התת-עץ, הצומת *x* הוא הבן השמאלי שלו (שורות 11-12)
  - התת-עץ השמאלי של *y* הופך להיות התת-עץ הימני של *x* (שורות 2-4)
- הסיבוב משנה את שורש התת-עץ, לכן יש לטפל גם באביו של *x*
  - אם *x* היה שורש העץ *T* – יש לקבוע לעץ שורש חדש (שורות 6-7)
  - אחרת, אם *x* היה הבן השמאלי של אביו, אז *y* יהיה הבן השמאלי החדש של אביו של *x* (שורה 9)
  - ולהיפך, אם *x* היה הבן הימני של אביו (שורה 10)

Left-Rotate(T, x)

Input: A binary tree T, and a non-nil node x in it

```
1.  y ← right[x]
2.  right[x] ← left[y]
3.  if left[y] ≠ nil[T]
4.     then p[left[y]] ← x
5.  p[y] ← p[x]
6.  if p[x] = nil[T]
7.     then root[T] ← y
8.     else if x = left[p[x]]
9.             then left[p[x]] ← y
10.            else right[p[x]] ← y
11. left[y] ← x
12. p[x] ← y
```