האוניברסיטה הפתוחה

מבני נתונים ומבוא לאלגוריתמים
מפגש הנחיה מס' 10

מדעי המחשב, קורס מס' 20407

סמסטר 2016ב

מנחה: ג'ון מרברג

מה ראינו במפגש הקודם?

- עצי חיפוש בינריים
  - שאילתות: חיפוש, מינימום, מקסימום, עוקב, קודם
  - פעולות: הכנסה, מחיקה
- עצים אדומים-שחורים
  - עצי חיפוש בינארי מאוזנים – מוטיבציה
  - הגדרת עץ אדום שחור
  - חסם עליון על גובה של עץ אדום שחור

מפגש עשירי

- נושא השיעור
  - פרק 13בספר – עצים אדומים-שחורים (המשך)
    - פעולות
      - סיבוב שמאלי וימני (פעולות עזר)
      - הכנסה, מחיקה
    - תרגיל: Treap
  - פרק 14 בספר – הרחבה של מבני נתונים
    - מוטיבציה
    - עץ ערכי מיקום

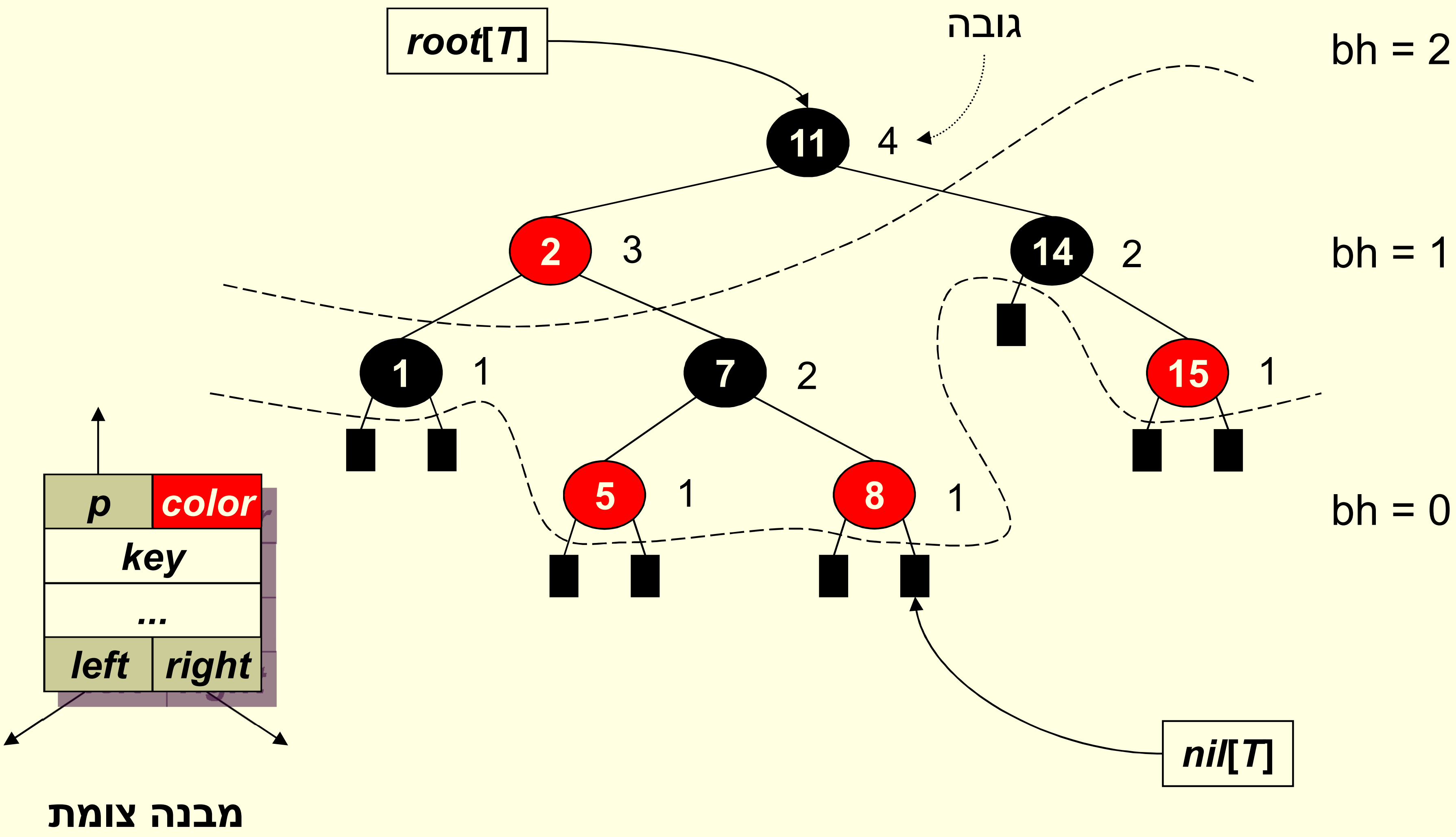
מבוסס על מצגת של ברוך חייקין ואיציק בייז

עצים אדומים-שחורים

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

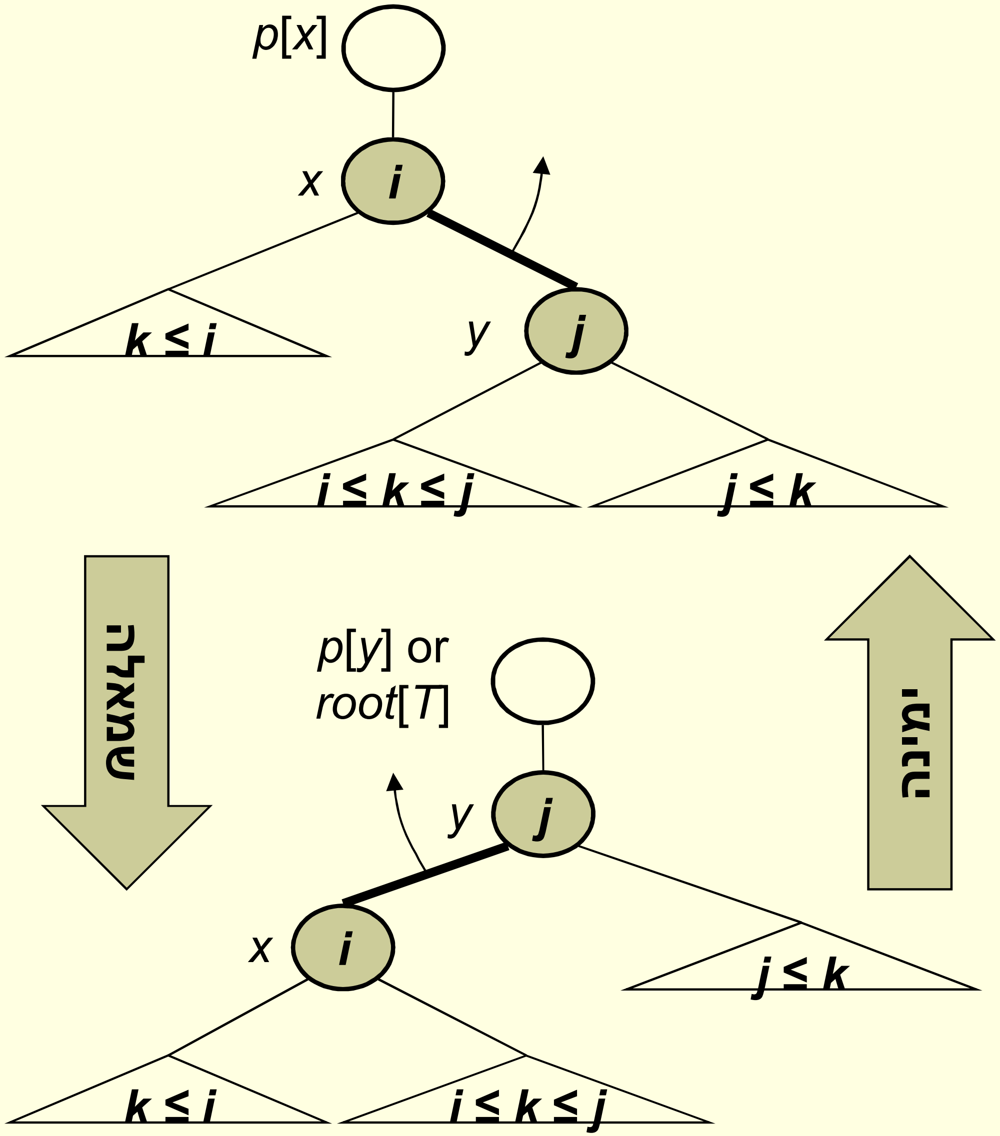
1. כל צומת הוא אדום או שחור
   - לכל צומת *x* בעץ נוסיף שדה *color*[*x*], שערכו BLACK או RED
2. השורש הוא שחור
3. כל עלה הוא שחור
   - נתייחס לכל מצביע nil כאילו הוא מצביע לצומת מיוחד *nil*[*T*] שצבעו שחור
   - כל הצמתים "הרגילים" נחשבים כצמתים פנימיים ויש להם שני בנים
4. אם צומת הוא אדום, אז שני בניו שחורים
5. לכל צומת, כל המסלולים מהצומת לצאצאים-עלים מכילים את אותו מספר של צמתים שחורים
   - 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



פעולות עזר: סיבוב שמאלי וימני

- סיבוב (Rotation) – פעולה <u>מקומית</u> בעץ בינרי, המשנה את המבנה של תת-עץ נתון
  - מתבצע בזמן $O(1)$
  - נשמרת תכונת העח"ב
- סיבוב שמאלי (Left-Rotate)
  - מופעל על השורש x של תת-עץ שבנו הימני *y* אינו nil
  - הופך את *y* לשורש החדש של תת-העץ, ואת *x* לבנו השמאלי
  - הקשת x–y "מסתובבת" נגד כיוון השעון על ציר שנמצא ב-x
- סיבוב ימני (Right-Rotate)
  - פעולה הופכית לסיבוב שמאלי
  - הכיוונים שמאל וימין מתחלפים
  - הקשת מסתובבת עם כיוון השעון



האוניברסיטה הפתוחה

סיבוב שמאלי – האלגוריתם

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
```

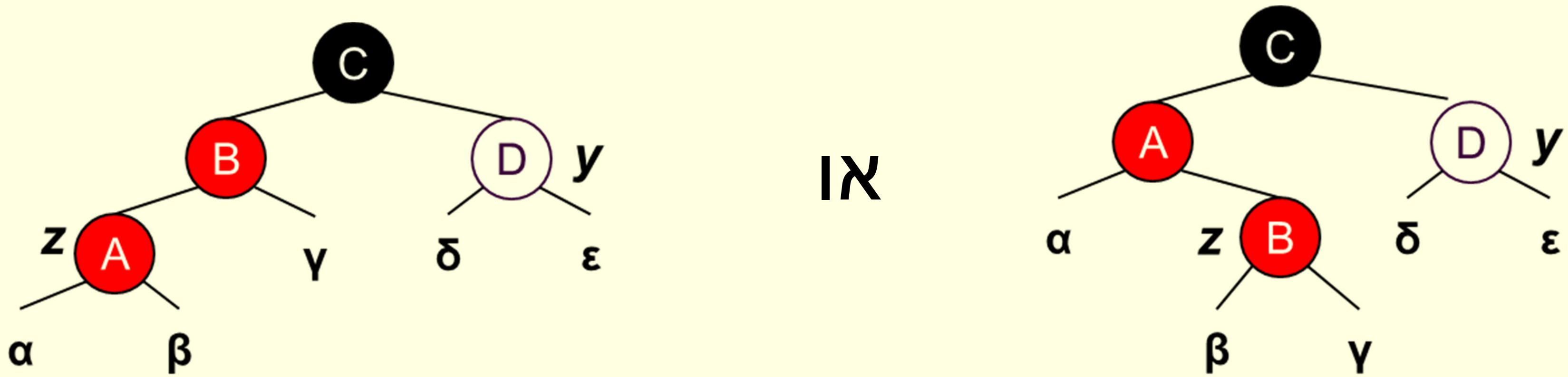
- אלגוריתם לסיבוב שמאלי
  - שורה 1: לפני הסיבוב, הצומת *x* הוא שורש תת-העץ, והצומת *y* הוא הבן הימני שלו
  - שורה 2-4: תת-העץ השמאלי של *y* עובר להיות תת-העץ הימני של *x*
  - שורה 5: אביו של *x* נהיה אביו החדש של y, *ובכך נהיה y לשורש החדש של תת-העץ*
  - שורה 6-7: אם *x* היה גם השורש של העץ כולו, אז *y* נהיה השורש החדש של העץ
  - שורה 8-9: אחרת, אם *x* היה בן שמאלי, אז *y* נהיה הבן השמאלי החדש של אביו של *x*
  - שורה 10: ואחרת, אם *x* היה בן ימני, אז *y* נהיה הבן הימני החדש של אביו של *x*
  - שורה 11-12: *x* נהיה הבן השמאלי החדש של *y*, ובכך עובר כל מה שנמצא מתחת ל-*x* אל החלק השמאלי של תת-העץ של *y*

הכנסה לעץ אדום-שחור

- האלגוריתם RB-Insert(T, z) מכניס צומת חדש z לעץ אדום-שחור T
  - שלב א': משתמשים בפעולה Tree-Insert(T, z) של עח"ב, כאשר מחליפים בקוד כל שימוש של nil ב- nil[T]
  - שלב ב': צובעים באדום את הצומת z
  - שלב ג': מבצעים תיקונים נחוצים בעזרת פעולה חדשה: RB-Insert-Fixup(T,z)
- נבחין כי אחרי שלב ב' יכול להתקיים מצב בו הצומת החדש z מפר את אחת התכונות של עץ אדום שחור, כדלקמן:
  - <u>הפרת תכונה 2</u>: הצומת החדש z הוא השורש והוא אדום (העץ הקודם היה ריק)
    - פשוט מחליפים את צבעו של z לשחור
  - <u>הפרת תכונה 4</u>: הצומת החדש z ואביו שניהם אדומים
    - יש לטפל בכמה תתי-מקרים (שיתוארו בהמשך)
    - נשים לב שהסבא של z תמיד קיים ותמיד שחור (כי האב אדום ולכן אינו השורש)
- הפעולה RB-Insert-Fixup() משתמשת בסיבוב שמאלי וימני כפעולות עזר

תיקון העץ עקב הפרת תכונה 4 לאחר הכנסה

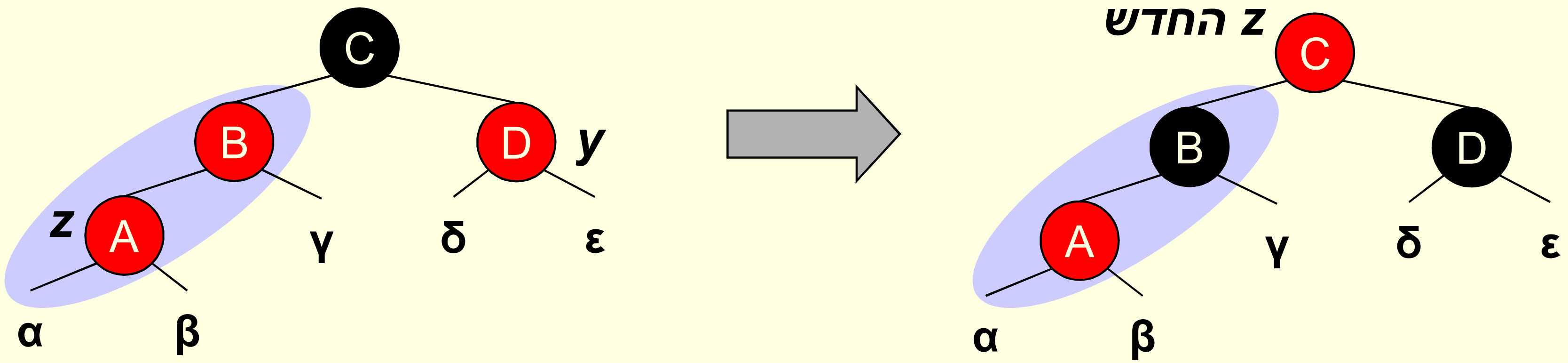
- המצב ה"קשה": הצומת החדש z ואביו שניהם אדומים
  - הסבא של z תמיד קיים ותמיד שחור (כי האב אדום ולכן אינו השורש)
- נניח שאביו של z הוא בן שמאלי (ולפיכך הדוד y הוא בן ימני).



- יש שלושה תתי-מקרים לטיפול
  - מקרה 1 – הדוד y אדום, לא משנה אם z הוא בן ימני או שמאלי
  - מקרה 2 – הדוד y שחור, z בן ימני
  - מקרה 3 – הדוד y שחור, z בן שמאלי
- יש עוד שלושה תתי-מקרים סימטריים, בהם אביו של z הוא בן ימני

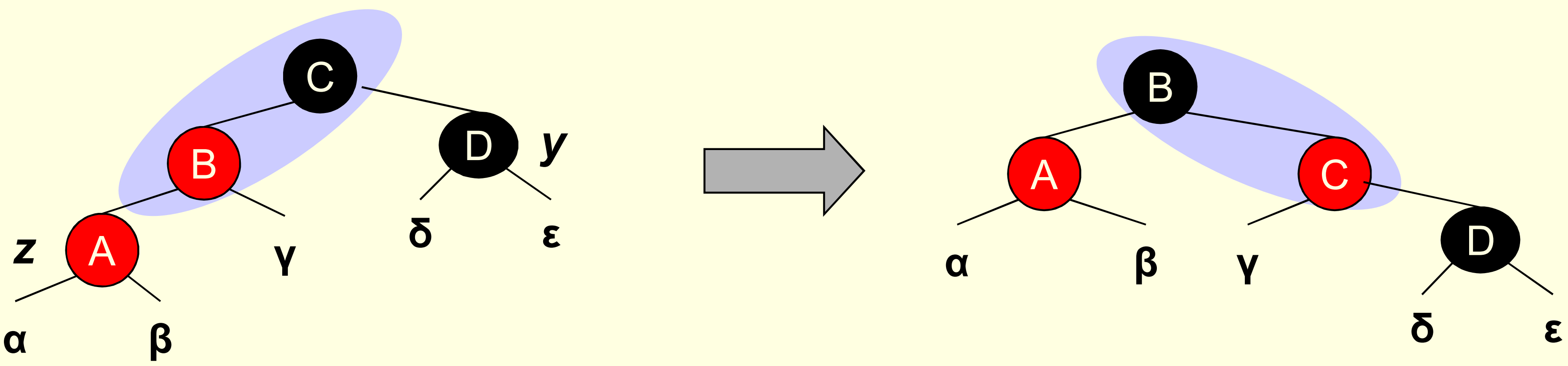
תיקון העץ לאחר ההכנסה – מקרה 1

<u>מצב</u>: הצומת *z*, אביו ודודו אדומים, הסבא שחור
<u>טיפול</u>: צובעים את האב והדוד בשחור, ואת הסבא באדום
<u>תוצאה</u>: תכונה 4 יכולה להיות עכשיו מופרת בסבא שנצבע באדום
<u>המשך</u>: מסמנים את הסבא בתור <u>z החדש</u> וממשיכים לאיטרצית תיקון נוספת (הצומת הבעיתי *z* טיפס שתי רמות למעלה בעץ)



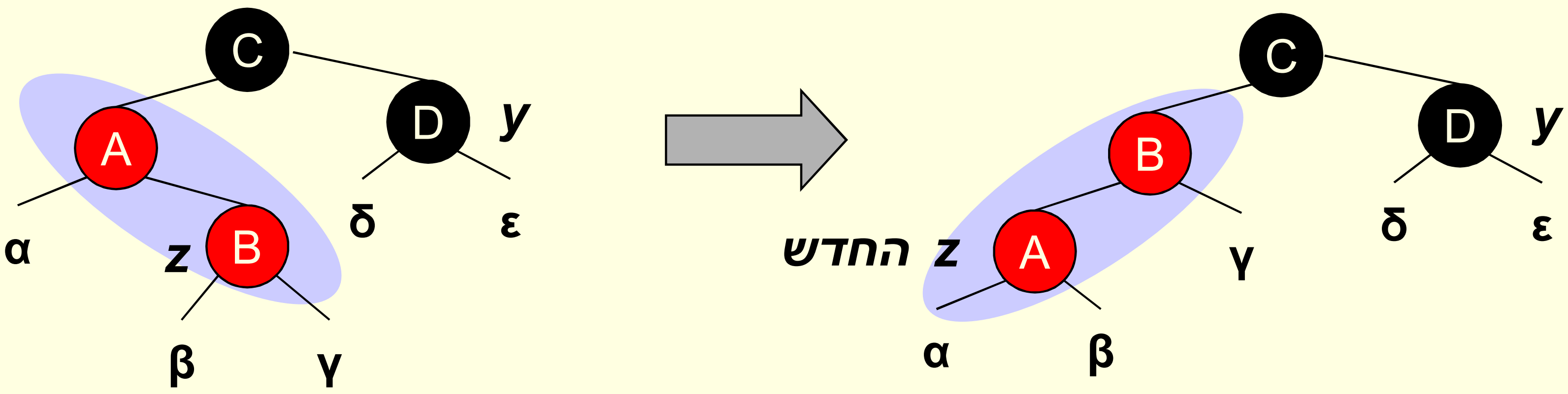
תיקון העץ לאחר ההכנסה – מקרה 3

<u>מצב</u>: הצומת *z* ואביו אדומים, הדוד והסבא שחורים, *z* בן שמאלי
<u>טיפול</u>: צובעים את האב בשחור ואת הסבא באדום ומבצעים סיבוב ימני על הסבא
<u>תוצאה</u>: תכונה 4 אינה מופרת בשום צומת – סיימנו!



תיקון העץ לאחר ההכנסה – מקרה 2

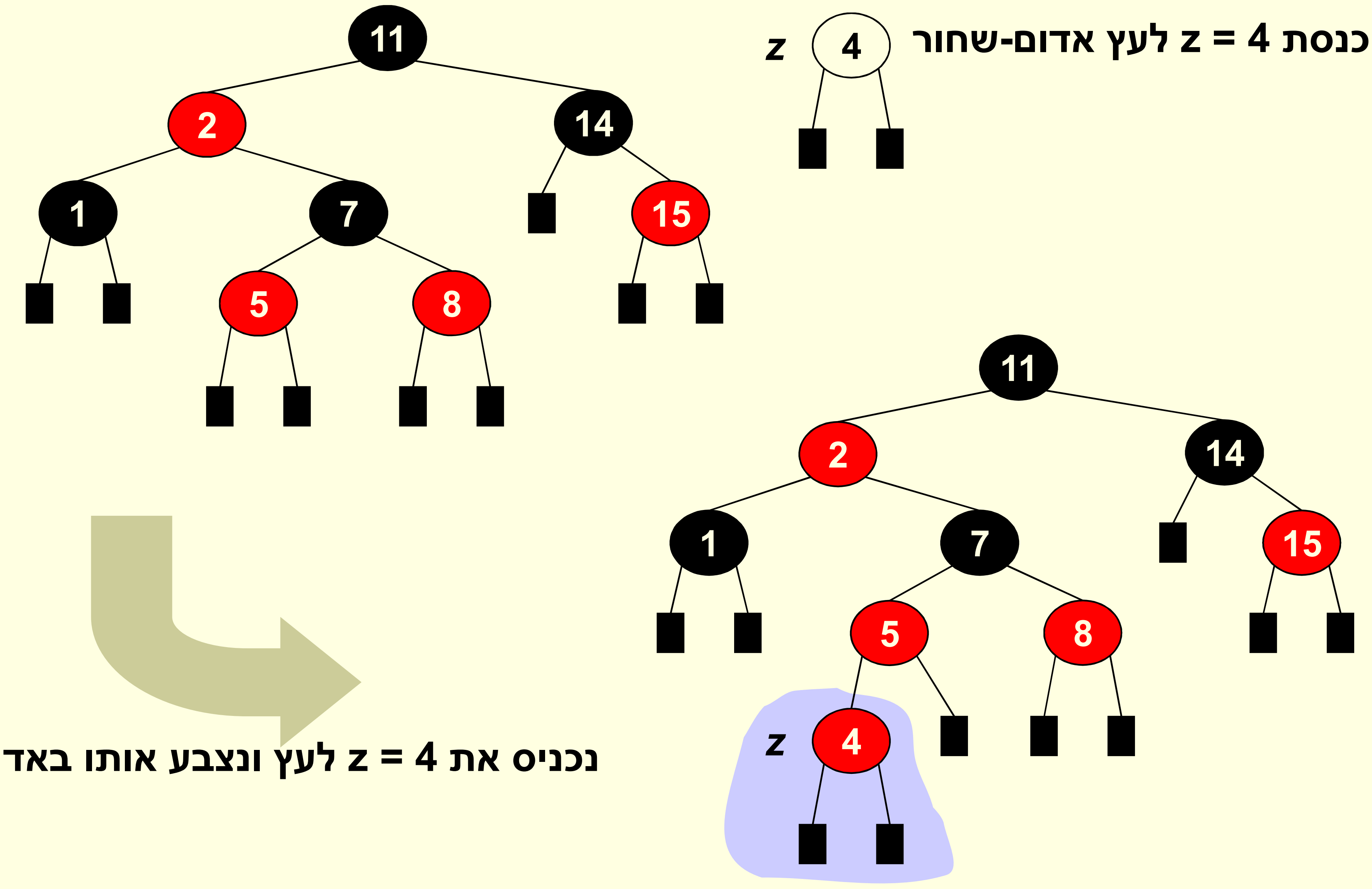
<u>מצב</u>: הצומת *z* ואביו אדומים, הדוד *y* והסבא שחורים, *z* בן ימני של אביו
<u>טיפול</u>: מבצעים על האב סיבוב שמאלי, ומסמנים אותו בתור <u>z החדש</u>
<u>תוצאה</u>: תכונה 4 עדיין מופרת – אבל המצב שהתקבל הוא מקרה 3
<u>המשך</u>: עוברים לטיפול במקרה 3 (ושם יסתיים תהליך התיקון)



δ

הכנסה – דוגמה

הכנסת 4 = z לעץ אדום-שחור



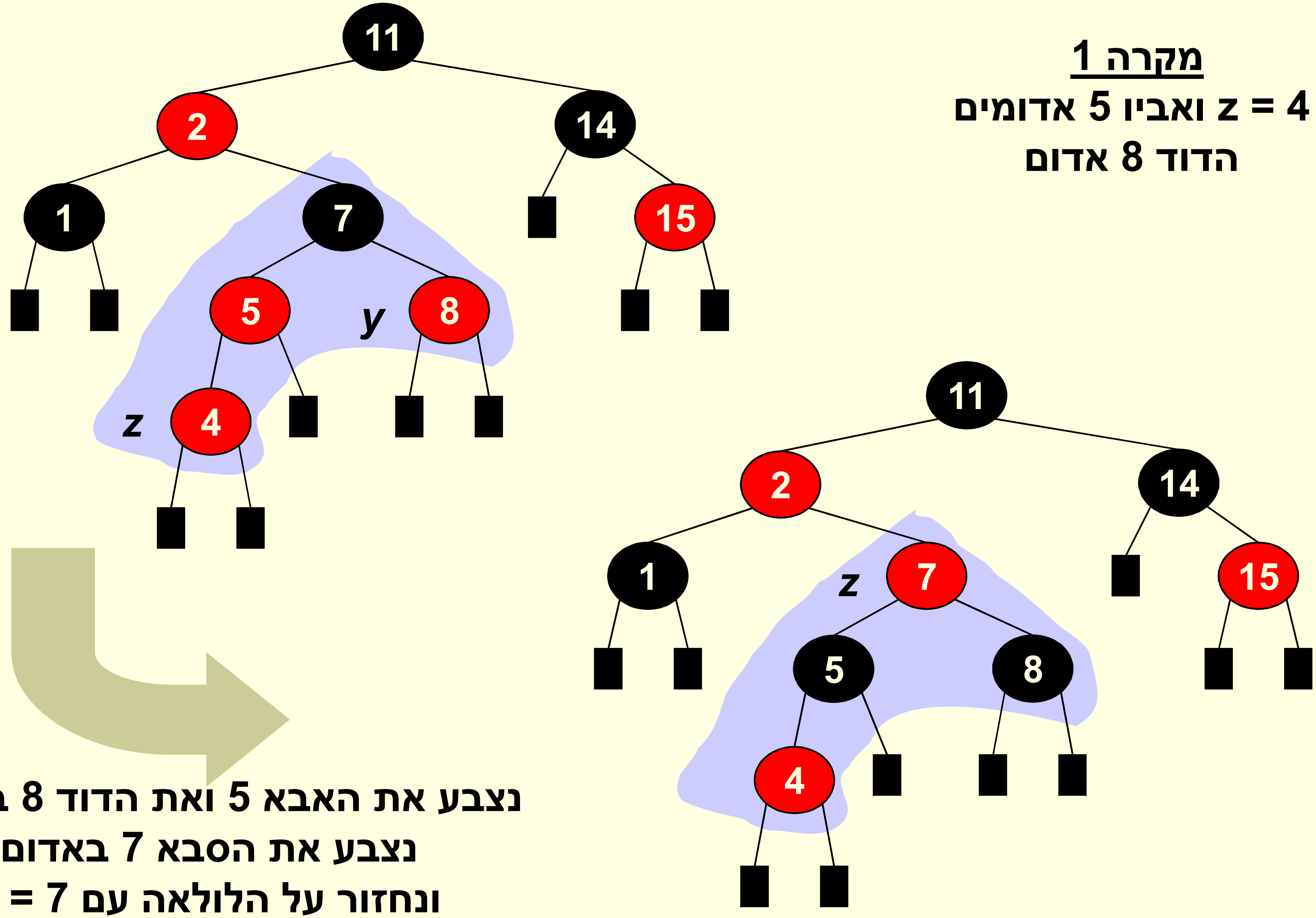
נכניס את 4 = z לעץ ונצבע אותו באדום

הכנסה – דוגמה (המשך)

<u>מקרה 1</u>

z = 4 ואביו 5 אדומים

הדוד 8 אדום

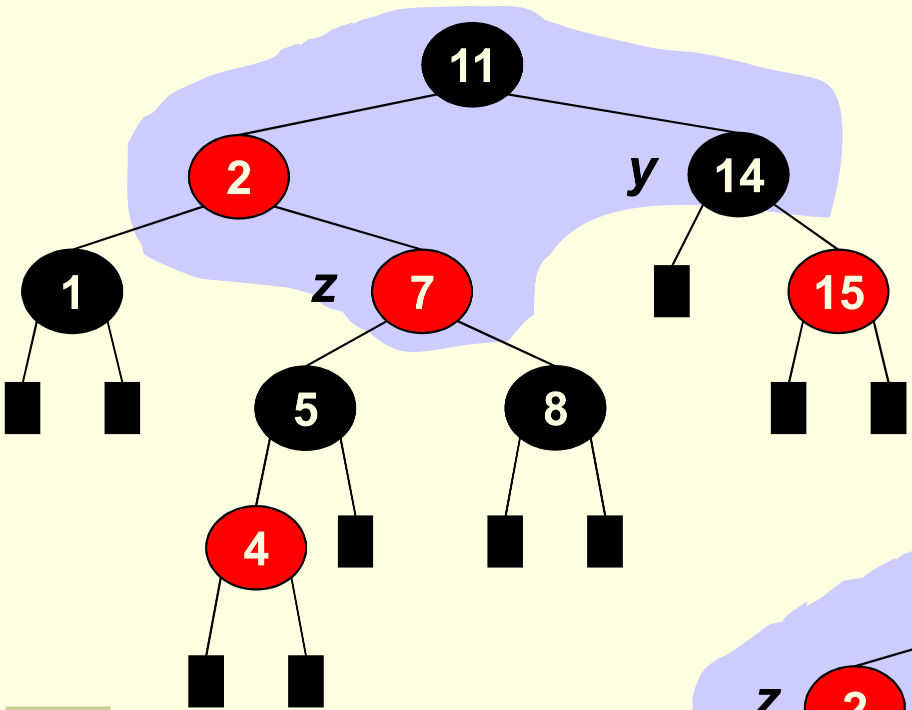


נצבע את האבא 5 ואת הדוד 8 בשחור

נצבע את הסבא 7 באדום

ונחזור על הלולאה עם z = 7

הכנסה – דוגמה (המשך)

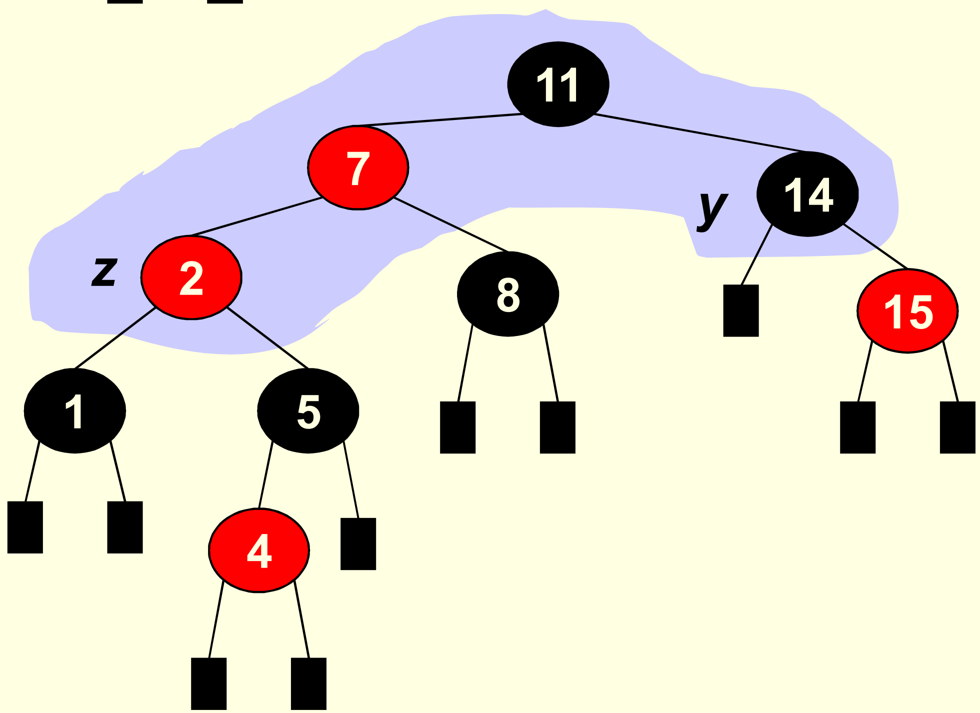


<u>מקרה 2</u>

$z = 7$ ואביו 2 אדומים

הדוד y=14 שחור

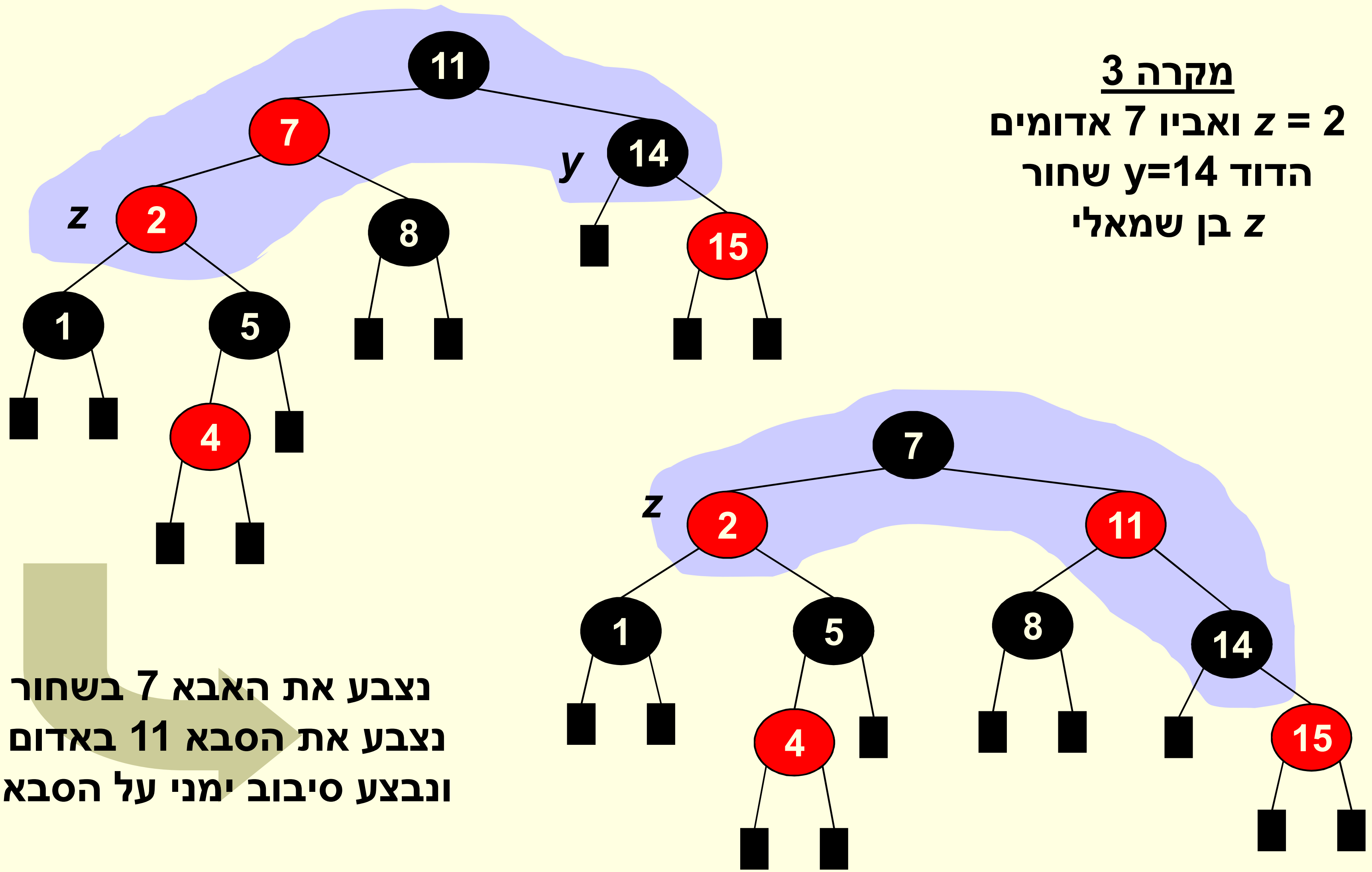
z בן ימני



נבצע סיבוב שמאלי על האב 2

ונמשיך למקרה 3 עם $z = 2$

הכנסה – דוגמה (המשך)



<u>מקרה 3</u>

2 = z ואביו 7 אדומים

הדוד y=14 שחור

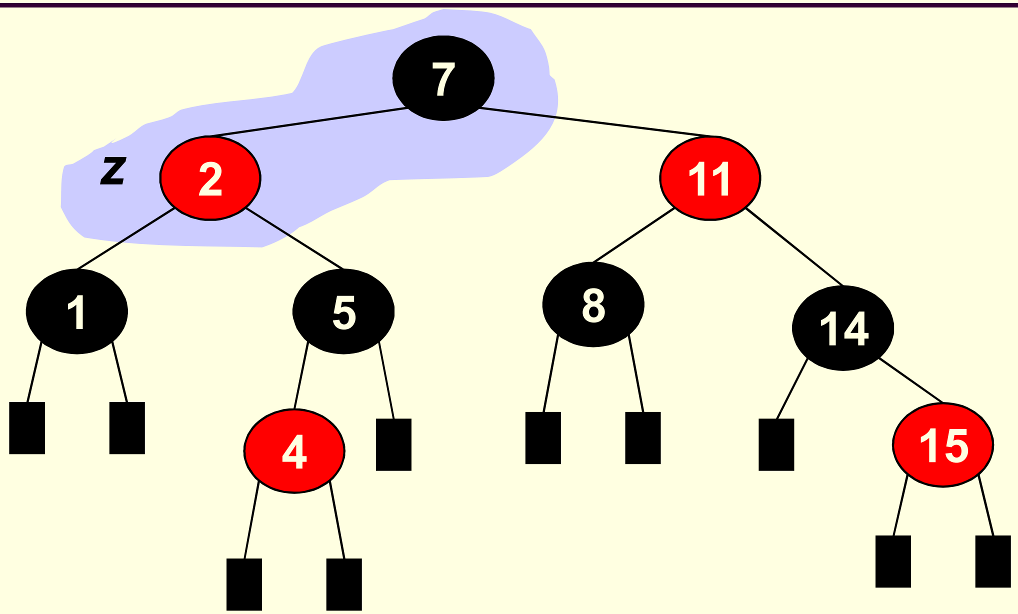
z בן שמאלי

נצבע את האבא 7 בשחור

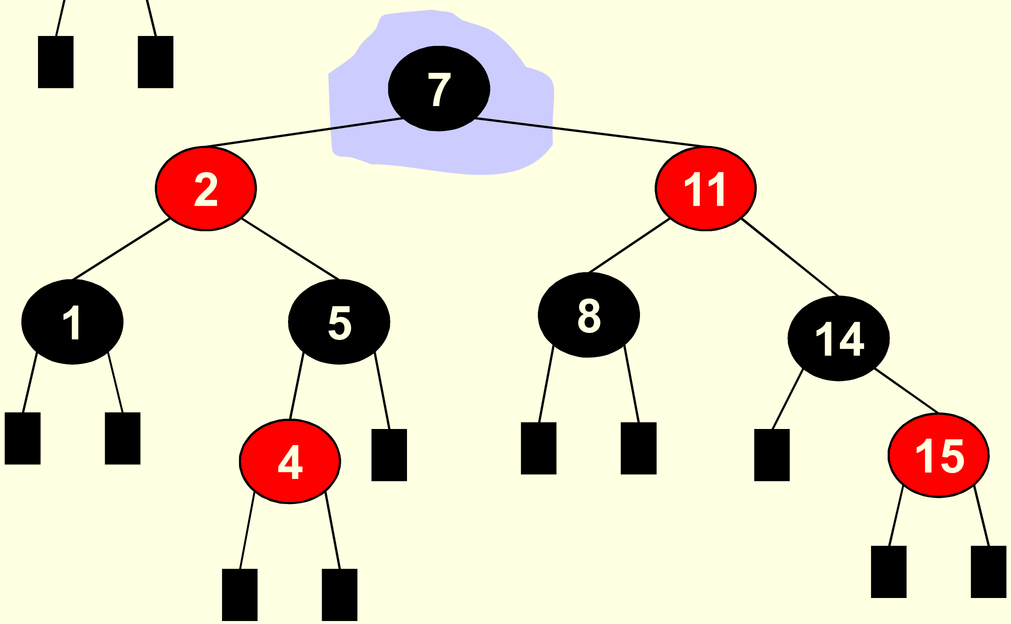
נצבע את הסבא 11 באדום

ונבצע סיבוב ימני על הסבא

הכנסה – דוגמה (סוף)



$z = 2$ אדום
אך אביו 7 שחור



נצבע את השורש בשחור ונסיים
(במקרה זה השורש כבר שחור)

האוניברסיטה הפתוחה

הכנסה – אלגוריתם התיקון

RB-Insert-Fixup(T, z)

1. **while** $color[p[z]]$ = **RED**
2. **do if** $p[z] = left[p[p[z]]]$
3. **then** $y \leftarrow right[p[p[z]]]$
4. **if** $color[y]$ = **RED**

5-8. **then** handle Case 1

9. **else if** $z = right[p[z]]$

10-11. **then** handle Case 2

12-14. handle Case 3

15. **else** handle symmetric cases where $p[z] = right[p[p[z]]]$
16. $color[root[T]] \leftarrow$ **BLACK**

Case 1

5. $color[p[z]] \leftarrow$ **BLACK**
6. $color[y] \leftarrow$ **BLACK**
7. $color[p[p[z]]] \leftarrow$ **RED**
8. $z \leftarrow p[p[z]]$

Case 2

10. Left-Rotate(T, $p[z]$)
11. $z \leftarrow left[z]$ ►z stays at same level

Case 3

12. $color[p[z]] \leftarrow$ **BLACK**
13. $color[p[p[z]]] \leftarrow$ **RED**
14. Right-Rotate(T, $p[p[z]]$)

הכנסה לעץ אדום-שחור – ניתוח זמן הריצה

- במשך כל מהלך התיקון יש סה"כ לכל היותר שני סיבובים
  - במקרה 1 אין סיבוב וממשיכים לאיטרציה הבאה עם הסבא
  - במקרה 2 יש סיבוב וממשיכים למקרה 3 באותה רמה בעץ
  - במקרה 3 יש סיבוב ומסיימים
- זמן הריצה $O(\lg n)$ במקרה הגרוע
  - שלב א' - ההכנסה לעץ מתבצעת בזמן $O(\lg n)$
  - שלב ב' - צביעת הצומת החדש מתבצעת בזמן $O(1)$
  - שלב ג' - התיקונים נעשים לאורך מסלול שאורכו $O(\lg n)$ כלפי השורש
    - בכל רמה במסלול, התיקון מתבצע בזמן $O(1)$
- <u>מסקנה</u>: אפשר לבנות עץ אדום שחור בזמן אופטימאלי $\Theta(n\log n)$
  - מתחילים עם עץ ריק ומבצעים n הכנסות
  - הזמן אופטימלי כי עבור עח"ב מתקיים: בניה + סריקה תוכית = מיון

מחיקה מעץ אדום-שחור

האלגוריתם RB-Delete(T, z) מוחק צומת נתון z מעץ אדום-שחור T

- שלב א': מוחקים צומת y (שנבחר ע"י האלגוריתם) באופן הדומה לפעולה Tree-Delete של עח"ב
  - הצומת y שנבחר למחיקה הוא או הצומת z או Tree-Successor(z)
  - אם $z \neq y$, מעתיקים את כל שדות התוכן של y ל- z
  - מובטח שלצומת הנמחק y יש רק בן יחיד שיסומן x (יתכן ש- $x = nil[T]$)
  - אפשר בקלות לחבר מחדש את x (וכל תת העץ שלו) אל האב של y
  - החיבור מחדש עלול לגרום להפרה של אחת או יותר מהתכונות של אדום-שחור
- שלב ב': מבצעים תיקונים בעזרת פעולה חדשה RB-Delete-Fixup(T,x)
  - אם y שנמחק *היה* אדום - אין צורך בתיקון, כי בנו x שחור (תכונה 4 התקיימה ב-y)
  - אם y שנמחק היה שחור - מופרות כעת התכונות הבאות של עץ שחור אדום
    - תכונה 5 מופרת באב החדש של x (במקור האב של y) וכן בכל אבותיו הקדמונים, כי "חסר" צומת שחור בכל מסלול מהשורש שעבר דרך y
    - תכונה 4 או 2 יכולה גם כן להיות מופרת, כדלקמן:
      - תכונה 4 מופרת אם x וגם אביו החדש (במקור האב של y) שניהם אדומים
      - תכונה 2 מופרת אם x *נהיה* לשורש העץ וצבעו אדום

האוניברסיטה הפתוחה

מחיקה – האלגוריתם הראשי

RB-Delete(T, z)

1. **if** $left[z] = nil[T]$ **or** $right[z] = nil[T]$
2.   **then** $y \leftarrow z$
3.   **else** $y \leftarrow$ Tree-Successor(z)
4. **if** $left[y] \neq nil[T]$
5.   **then** $x \leftarrow left[y]$
6.   **else** $x \leftarrow right[y]$
7. $p[x] \leftarrow p[y]$
8. **if** $p[y] = nil[T]$
9.   **then** $root[T] \leftarrow x$
10.   **else if** $y = left[p[y]]$
11.   **then** $left[p[y]] \leftarrow x$
12.   **else** $right[p[y]] \leftarrow x$
13. y**if** $y \neq z$
14.   **then** $key[z] \leftarrow key[y]$
15.   ► copy y's satellite data, too
16. **if** $color[y] =$ BLACK
17.   **then** RB-Delete-Fixup(T, x)
18. **return**

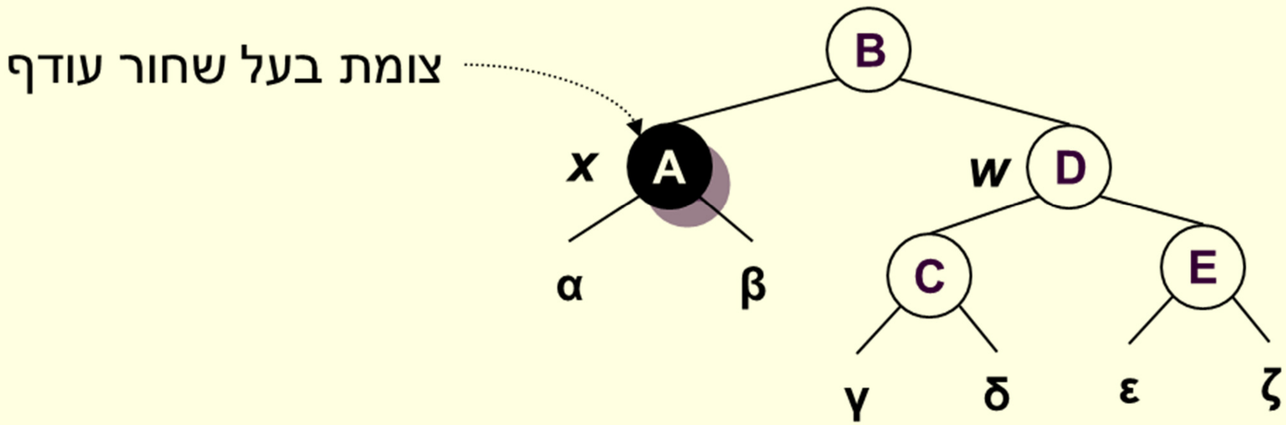
- דומה ל- Tree-Delete, עם השינויים שלהלן (מודגשים בכחול):
  - שורות 1, 4, 8: ההתייחסויות ל-nil הוחלפו ב-*nil*[*T*]
  - שורה 7: ההצבה אינה מותנית כי *x* אינו *nil* (אלא הוא הצומת *nil*[*T*])
  - שורות 16-17: נוספה קריאה מותנית לפעולת התיקון

תיקון העץ לאחר המחיקה

- נסמן ב- x את בנו (היחיד) של הצומת y שנמחק
  - אם y שחור, אז תכונה 5 מופרת כעת בכל האבות הקדמונים של x,
  - *ייתכן שגם תכונה 4 או 2 מופרת בצומת x*
- הרעיון לתיקון: במקום לתקן את תכונה 5 נתקן את תכונה 1
  - נתייחס לצומת x כאילו הוא נושא שני צבעים (צבע ראשי וצבע משני)
    - הצבע המשני נקרא "שחור עודף" – והוא הצבע השחור שהיה מקודם של y
    - הצבע הראשי יכול להיות שחור או אדום
  - לפיכך, תכונה 5 לא מופרת יותר, כי מספר הצמתים השחורים נשמר
  - אבל במקום זה מופרת תכונה 1 – הצומת x אינו אדום או שחור "רגיל"
  - נחפש צומת בעץ שבו ניתן "להיפטר" מן השחור העודף תוך שמירה על שאר התכונות
    - אם צבעו הראשי של x הוא אדום, ניפטר מן השחור העודף ע"י כך שנצבע את x בשחור. זה פותר גם את ההפרה של תכונה 2 או 4, אם קיימת ב-x.
    - אחרת (צבעו הראשי של x הוא שחור)
      - אם x הוא השורש, אפשר פשוט לזרוק את השחור העודף, כי השורש אינו נספר במניין הצמתים השחורים במסלול לעלה
      - אחרת (x אינו השורש), נבצע סדרת סיבובים ושינויי צבע כדי להעביר את השחור העודף לצומת אדום או כלפי מעלה אל השורש (כמפורט בהמשך)

תיקון העץ לאחר המחיקה (המשך)

- המצב ה"קשה": x נושא שחור עודף, צבעו הראשי שחור, והוא לא השורש
  - נניח כי x הוא בן שמאלי. אחיו הימני יסומן w.
    - w חייב להיות צומת פנימי, כי תכונה 5 נשמרת באב המשותף של x ושל w, כאשר x נספר כשחור כפול, בעוד w נספר כשחור בודד



  - יש ארבעה תתי-מקרים לטיפול
    - מקרה 1: w אדום (ולכן בניו וגם אביו שחורים, לפי תכונה 4)
    - מקרה 2: w שחור, בניו שחורים
    - מקרה 3: w שחור, בנו הימני שחור, בנו השמאלי אדום
    - מקרה 4: w שחור, בנו הימני אדום (לא משנה מה צבע הבן השמאלי)
  - יש עוד ארבעה תתי-מקרים סימטריים, כאשר x הוא בן ימני (w אחיו השמאלי)

האוניברסיטה הפתוחה

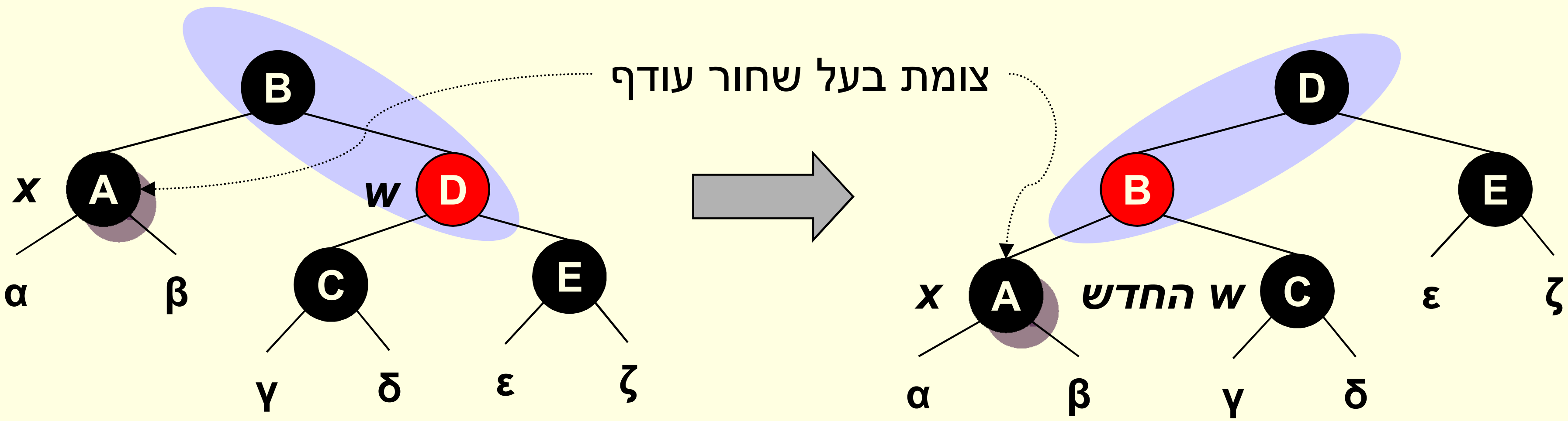
תיקון העץ לאחר המחיקה – מקרה 1

<u>מצב</u>: הצומת x נושא "שחור עודף", אחיו הימני w אדום
האב של x ו- w, ושני הבנים של w חייבים להיות שחורים, לפי תכונה 4

<u>טיפול</u>: מחליפים את הצבעים של w ושל אביו, מבצעים על האב סיבוב שמאלי,
וקובעים את האח החדש של x בתור <u>w החדש</u>

<u>תוצאה</u>: כל התכונות (מלבד 1) נשמרות – בפרט מספר השחורים זהה בכל מסלול;
השחור העודף נשאר באותו צומת x

<u>המשך</u>: *האח החדש w* שחור – לכן עוברים למקרה 2, 3, או 4



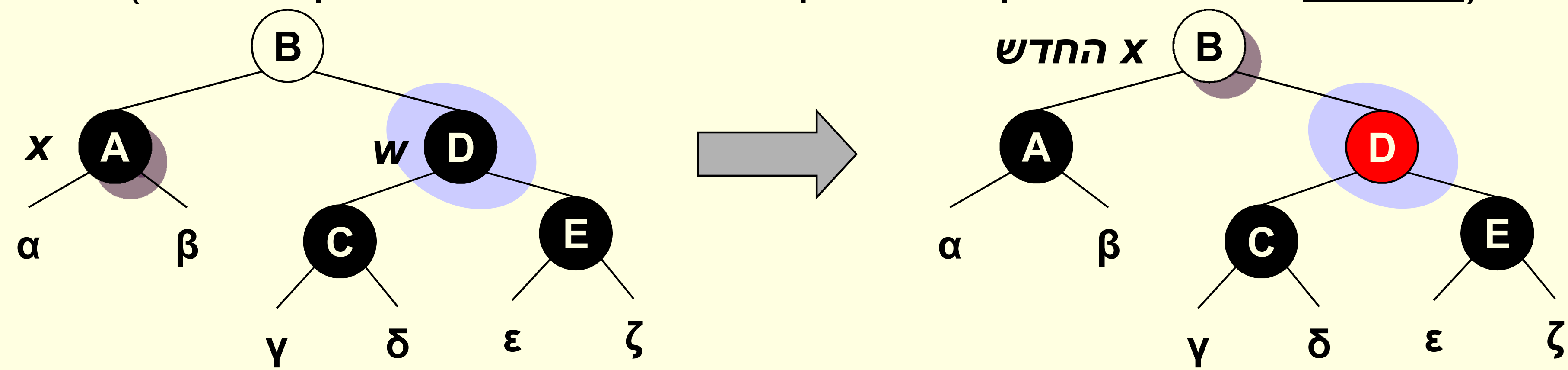
תיקון העץ לאחר המחיקה – מקרה 2

<u>מצב</u>: הצומת *x* נושא "שחור עודף", אחיו הימני *w* ושני בניו שחורים
צבע האב של *x* ו- *w* אינו משנה
(<u>נשים לב</u>: אם הגענו למקרה 2 ממקרה 1, האב הוא אדום)

<u>טיפול</u>: צובעים את *w* באדום, ומסמנים את האב בתור <u>*x* החדש</u>
כמו כן, מעבירים את השחור העודף לאב (כלומר ל- *x* החדש)

<u>תוצאה</u>: כל התכונות (חוץ מ-1) נשמרות – בפרט, מספר השחורים זהה בכל מסלול;
מבנה העץ נשמר – אבל השחור העודף מועבר רמה אחת למעלה בעץ

<u>המשך</u>: אם x החדש הוא השורש, או שצבעו הראשי אדום – צובעים אותו בשחור,
ובכך נפטרים מהשחור העודף ומסיימים!
אחרת ממשיכים לאיטרציה נוספת (הצומת הבעיתי *x* טיפס רמה אחת למעלה)
(<u>נשים לב</u>: אם הגענו למקרה 2 ממקרה 1, *x* החדש אדום ולכן מסיימים)

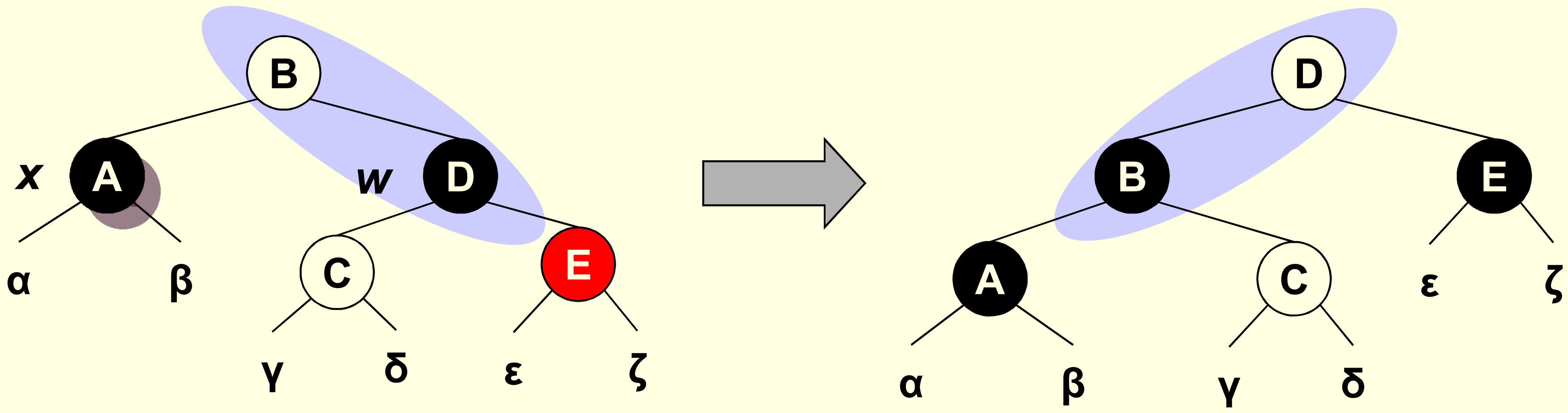


תיקון העץ לאחר המחיקה – מקרה 4

<u>מצב</u>: הצומת הנוכחי *x* מכיל "שחור עודף", אחיו הימני *w* שחור, הבן הימני של *w* אדום,
צבעו של האב של w ו-*x*, וכן *צבעו של* הבן השמאלי של *w* אינם משנים

<u>טיפול</u>: מחליפים בין הצבעים של *w* ושל אביו, צובעים את הבן הימני של *w* בשחור,
ומבצעים על האב סיבוב שמאלי
כמו כן צובעים את השורש בשחור (ייתכן והוא כבר שחור)

<u>תוצאה</u>: כל התכונות נשמרות. סיימנו!
נפטרנו מהשחור העודף ע"י צביעת צומת אדום בשחור.



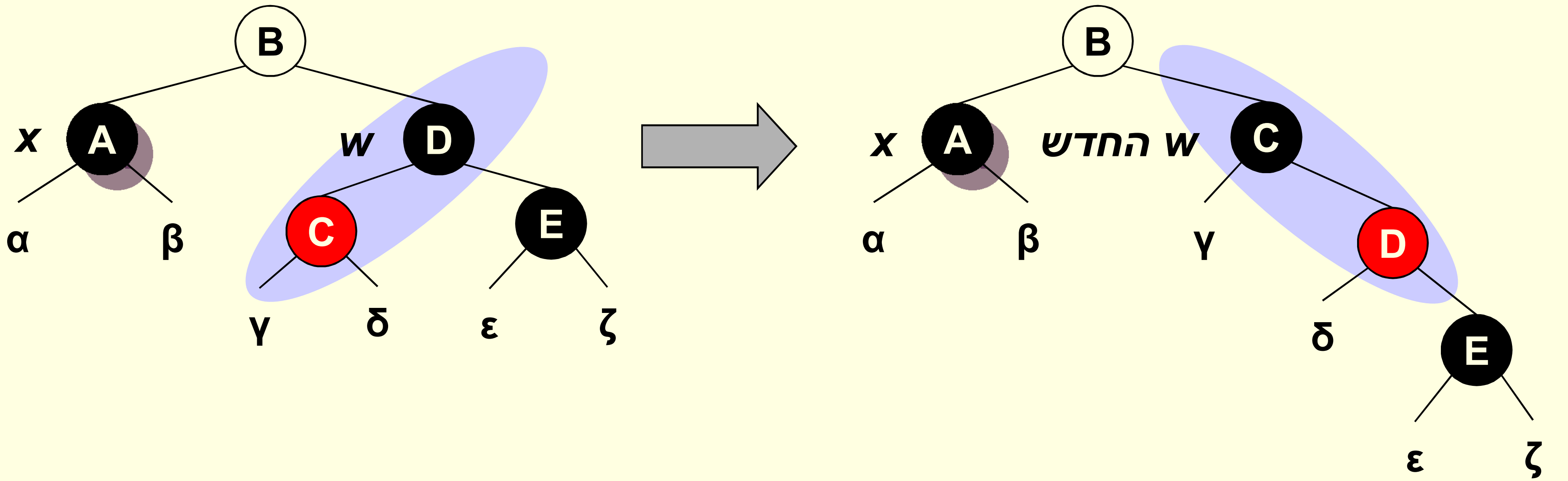
תיקון העץ לאחר המחיקה – מקרה 3

<u>מצב</u>: הצומת הנוכחי *x* מכיל "שחור עודף", אחיו הימני *w* ובנו הימני של *w* שחורים, הבן השמאלי של *w* אדום (צבע האב של x ו- *w* אינו משנה)

<u>טיפול</u>: מחליפים בין הצבעים של *w* ושל בנו השמאלי, מבצעים על *w* סיבוב ימני, וקובעים את האח החדש של *x* בתור <u>*w* החדש</u>

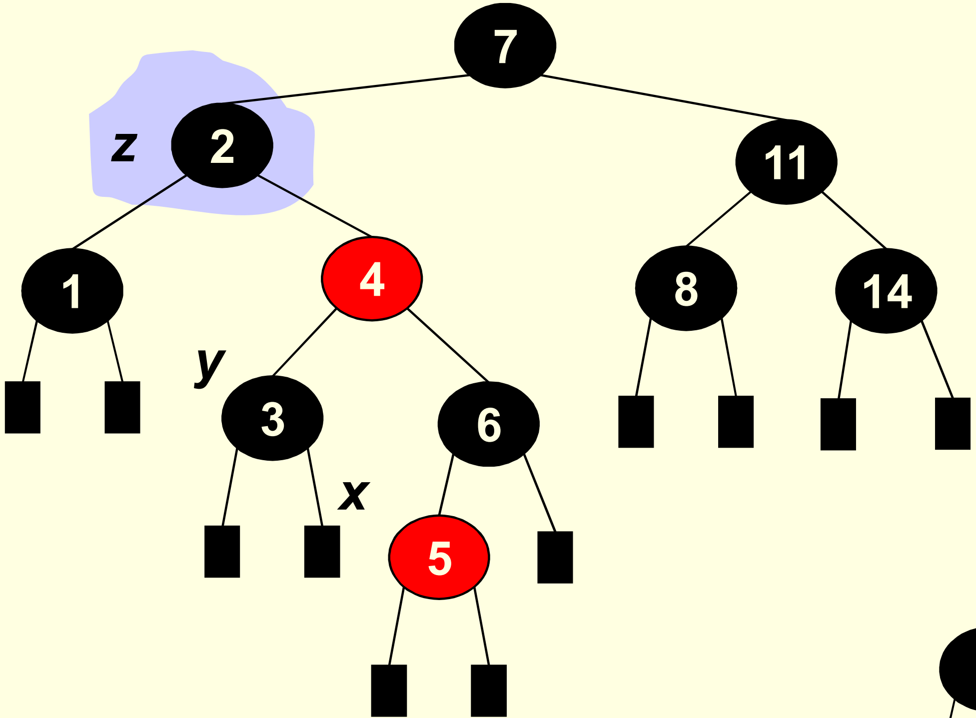
<u>תוצאה</u>: כל התכונות (חוץ מ-1) נשמרות – בפרט, מספר השחורים זהה בכל מסלול; המצב שהתקבל הוא מקרה 4

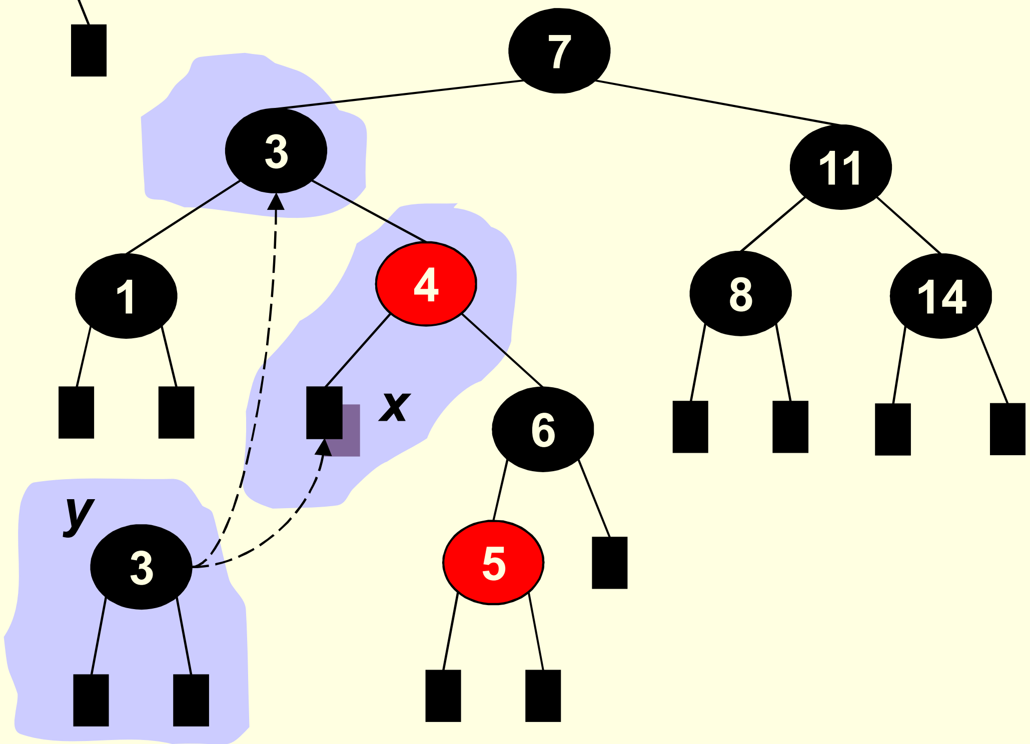
<u>המשך</u>: עוברים לטיפול במקרה 4 (ושם יסתיים תהליך התיקון)



מחיקה – דוגמה

מחיקת $z = 2$ מעץ אדום-שחור
עם $bh(root[T]) = 3$

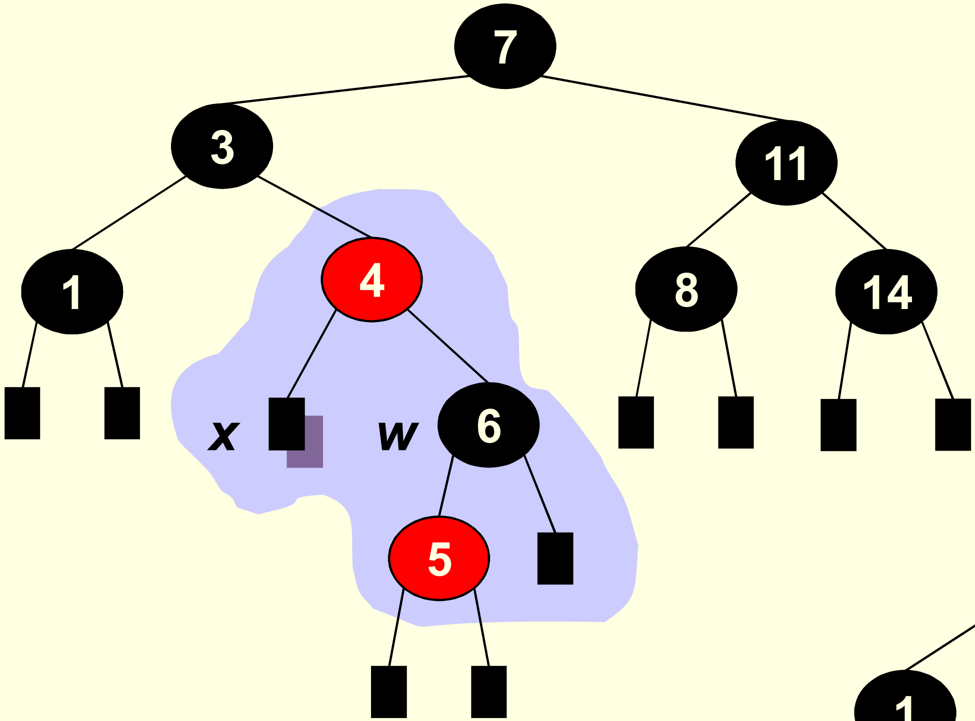




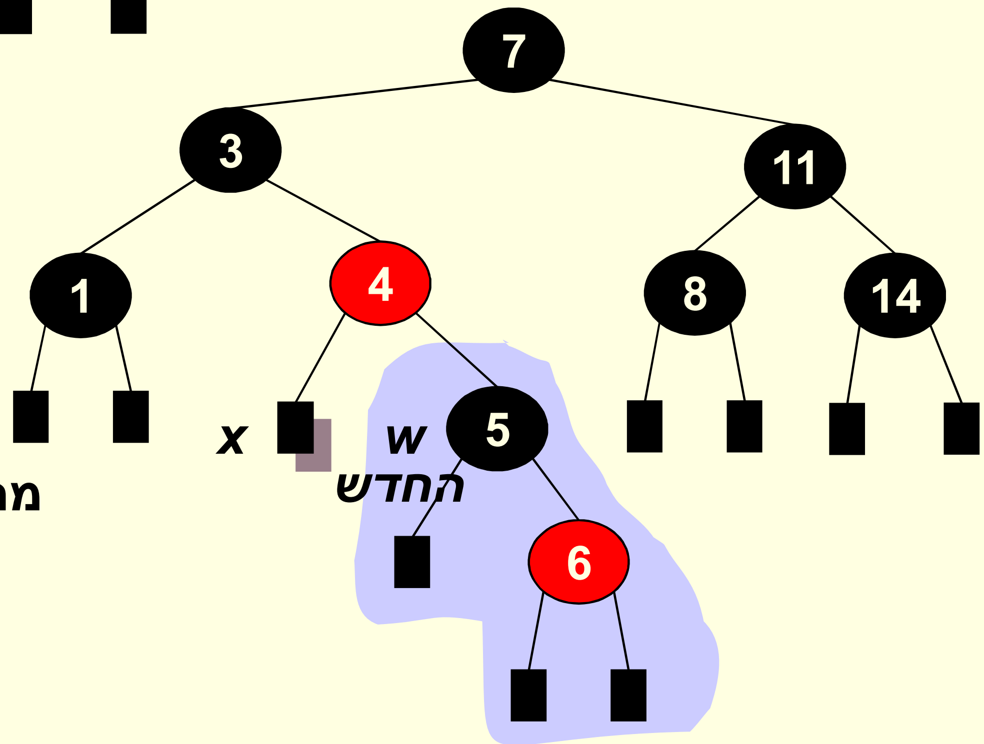
ל-z יש שני בנים, לכן נמצא את העוקב שלו $y = 3$ ונציב $z \leftarrow 3$
נמחק את y מהעץ ע"י חיבור הבן הימני שלו x לאביו של y
הצבע השחור של y עובר להיות "שחור עודף" אצל x

האוניברסיטה הפתוחה

מחיקה – דוגמה (המשך)



y היה שחור, לכן נכנסים לשגרת התיקון;
$x = nil[T]$ אינו השורש והוא שחור;
האח $w = 6$ שחור, בנו הימני $nil[T]$ שחור
ובנו השמאלי 5 אדום;
לכן מתקיים <u>מקרה 3</u>

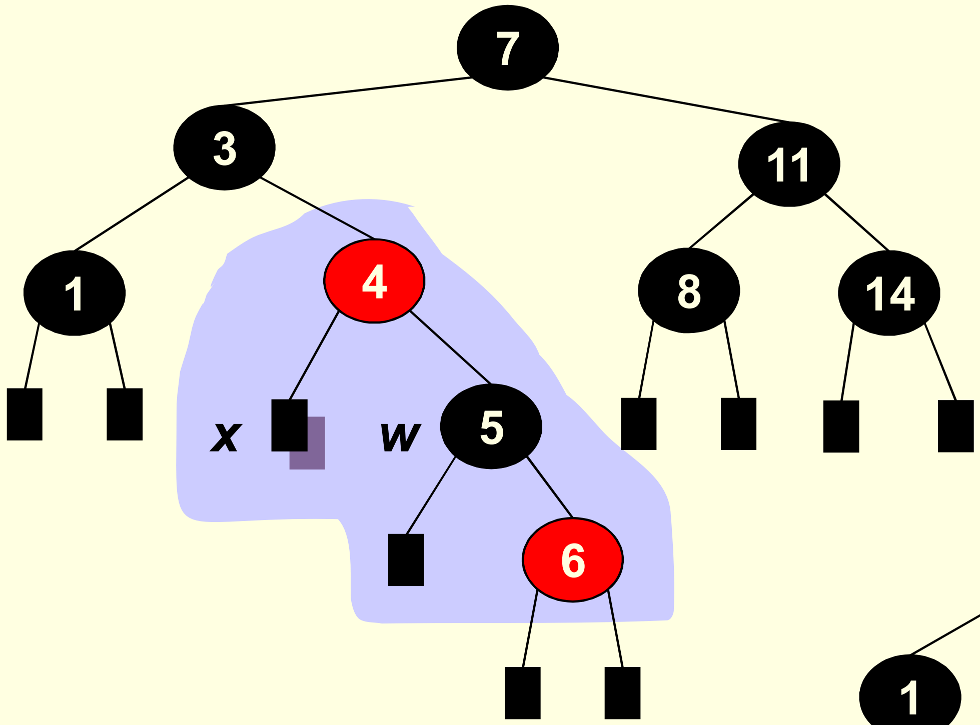


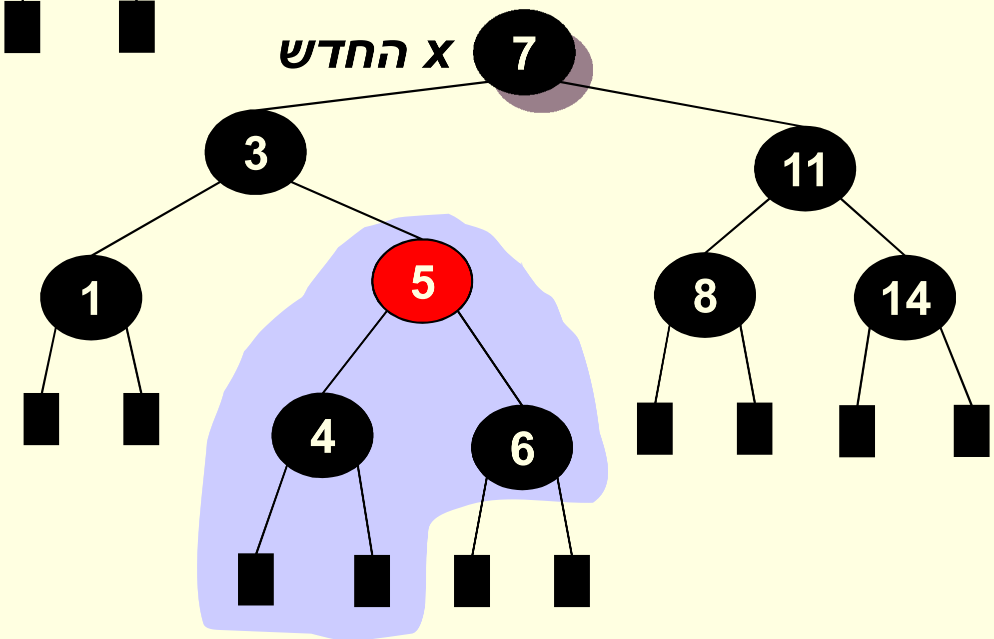
מחליפים את הצבעים של $w = 6$ ובנו השמאלי 5
מבצעים על w סיבוב ימני
קובעים את 5 בתור האח w החדש
וממשיכים למקרה 4

מחיקה – דוגמה (המשך)

<u>מקרה 4</u>

האח 5 = *w* שחור ובנו הימני 6 אדום

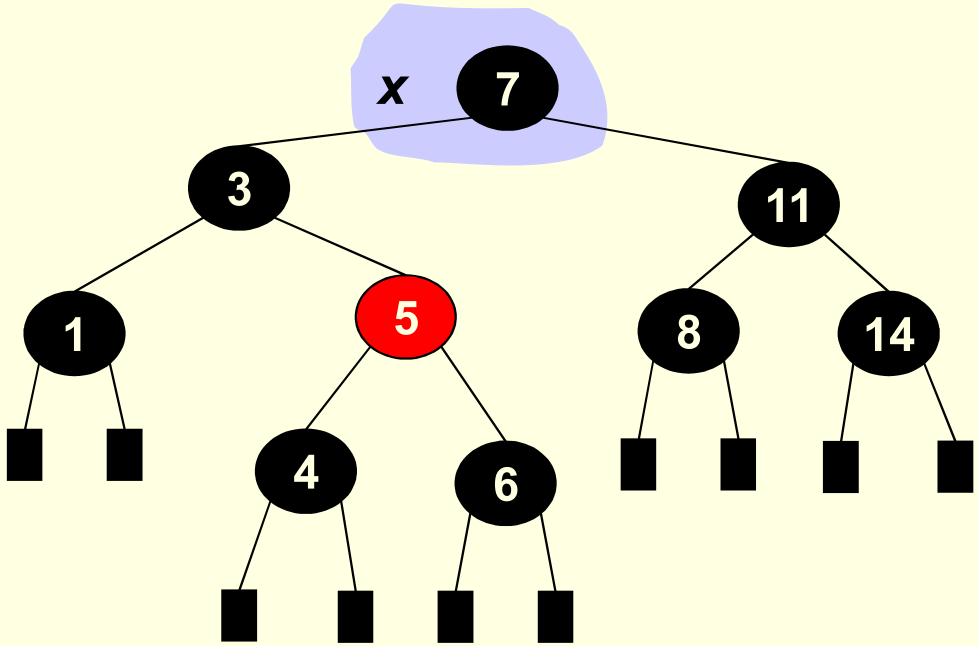




מחליפים את הצבעים של *w* ושל אביו,
צובעים את 6, הבן הימני של *w*, בשחור
מבצעים על האב 4 סיבוב שמאלי.
נשאר רק לסיים בתיקון השורש (נסמן אותו *x* החדש)

האוניברסיטה הפתוחה

מחיקה – דוגמה (סוף)



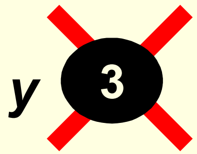
<u>סיום</u>

צובעים את השורש *x* בשחור
(בדוגמה זו הוא כבר היה שחור)

מתקבל עץ אדום-שחור תקין
עם $bh(root[T]) = 3$

בכך מסתיימת שגרת התיקון

לבסוף, מעבירים למיחזור
את *y*, הצומת שנמחק בפועל



מחיקה – אלגוריתם התיקון

RB-Delete-Fixup(T, x)

1. **while** $x \neq root[T]$ **and** $color[x]$ = **BLACK**
2. **do if** $x = left[p[x]]$
3. **then** $w \leftarrow right[p[x]]$
4. **if** $color[w]$ = **RED**
5-8. **then** handle Case 1
9. **if** $color[left[w]]$ = **BLACK and** $color[right[w]]$ = **BLACK**
10-11. **then** handle Case 2
12. **else if** $color[right[w]]$ = **BLACK**
13-16. **then** handle Case 3
17-21. handle Case 4
22. **else** handle symmetric cases where $x = right[p[x]]$
23. $color[x] \leftarrow$ **BLACK**

Case 1

5. $color[w] \leftarrow$ **BLACK**
6. $color[p[x]] \leftarrow$ **RED**
7. Left-Rotate(T, $p[x]$)
8. $w \leftarrow right[p[x]]$

Case 2

10. $color[w] \leftarrow$ **RED**
11. $x \leftarrow p[x]$

Case 3

13. $color[left[w]] \leftarrow$ **BLACK**
14. $color[w] \leftarrow$ **RED**
15. Right-Rotate(T, w)
16. w $\leftarrow right[p[x]]$

Case 4

17. $color[w] \leftarrow color[p[x]]$
18. $color[p[x]] \leftarrow$ **BLACK**
19. $color[right[w]] \leftarrow$ **BLACK**
20. Left-Rotate(T, $p[x]$)
21. x $\leftarrow root[T]$

מחיקה מעץ אדום-שחור – ניתוח זמן הריצה

האוניברסיטה הפתוחה

- במשך כל מהלך התיקון יש סה"כ לכל היותר 3 סיבובים
  - במקרה 1 יש סיבוב וממשיכים עם מקרה 2 או 3 או 4
  - במקרה 2 אין סיבוב.
    - אם הגענו ממקרה 1, מסיימים.
    - אחרת, ממשיכים לאיטרציה נוספת עם האב
  - במקרה 3 יש סיבוב וממשיכים למקרה 4
  - במקרה 4 יש סיבוב ומסיימים

- זמן הריצה $O(\lg n)$ במקרה הגרוע
  - המחיקה עצמה מתבצעת בזמן $O(\lg n)$ (בגלל החיפוש של העוקב)
  - התיקונים מתבצעים לאורך מסלול שאורכו $O(\lg n)$ כלפי השורש
    - בכל רמה במסלול, התיקון מתבצע בזמן $O(1)$

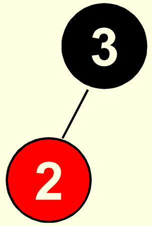
תרגיל 13.4-7: הכנסה ומחיקה

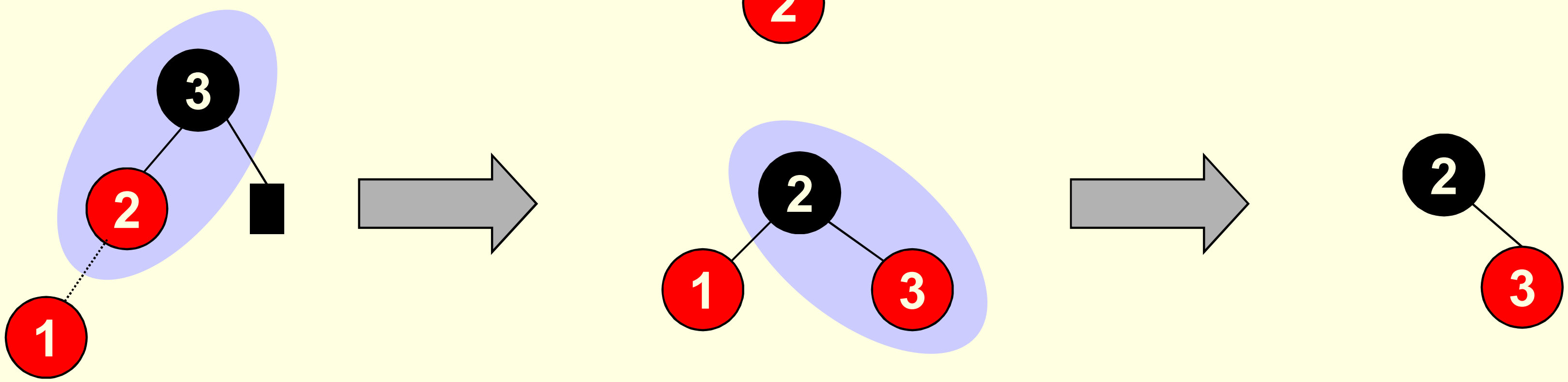
- נניח שמכניסים צומת x לעץ אדום-שחור באמצעות RB-Insert, ומיד לאחר מכן מוחקים אותו באמצעות RB-Delete.
- האם העץ האדום-שחור המתקבל זהה לעץ המקורי?

  הוכיחו או תנו דוגמא נגדית.

פיתרון תרגיל 13.4-7: הכנסה ומחיקה

דוגמה נגדית:

נוסיף את הצומת 1 לעץ  ואח"כ נמחק אותו



הכנסה + תיקון לפי מקרה 3
צובעים את האב בשחור והסבא באדום
מבצעים סיבוב ימני סביב הסבא ומסיימים

מחיקה
(אין צורך בתיקון)

העץ המתקבל שונה מהעץ המקורי!

תרגיל - Treap

- נתון עץ בינרי *T* שבו בכל צומת *x* יש שני ערכים:
  - מפתח *key*[*x*]
  - עדיפות *priority*[*x*]
- *הגדרה: העץ T נקרא Treap* (חיבור של המילה Tree עם המילה Heap), אם *T* הוא עח"ב על המפתחות, ובכל צומת מתקיימת תכונת הערמה על העדיפות.

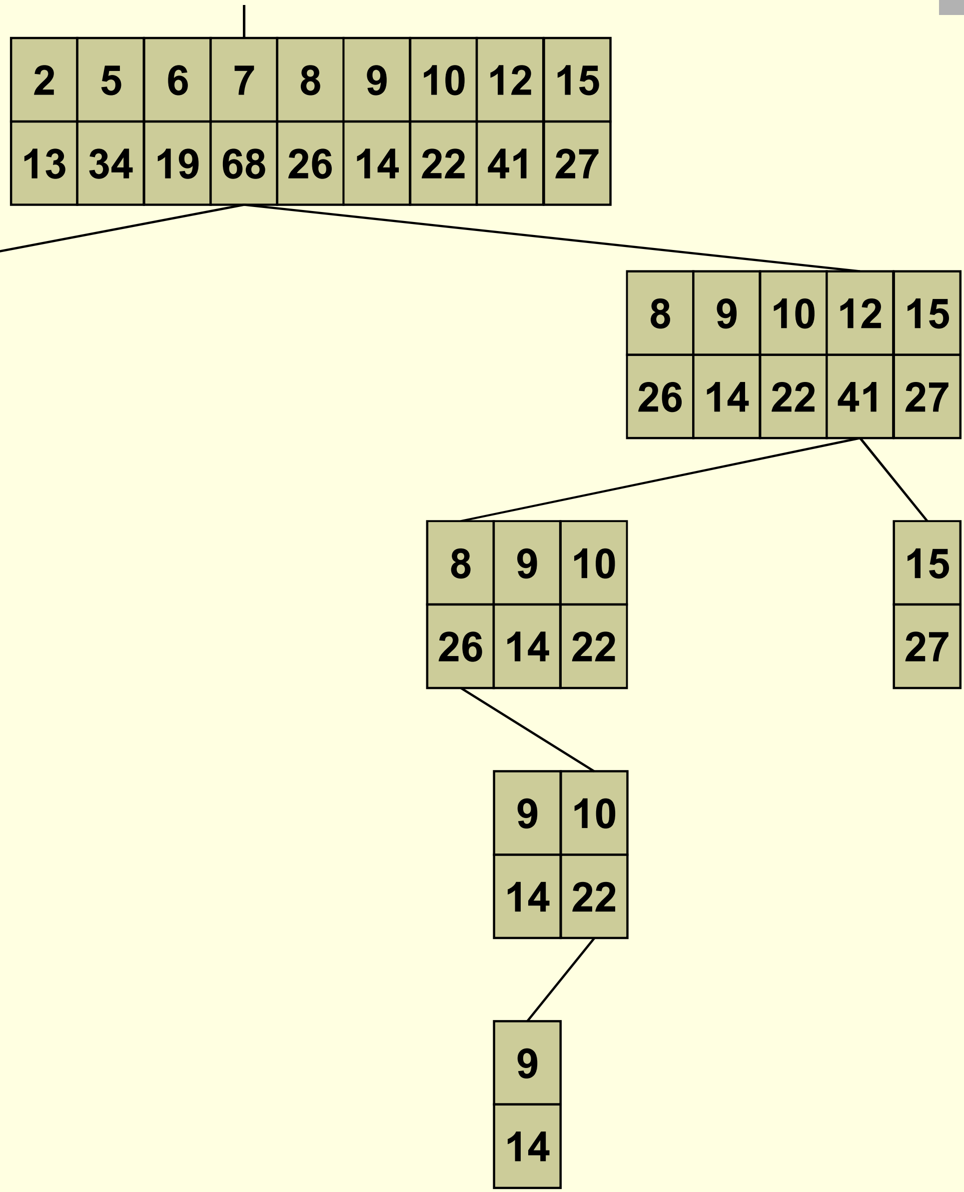
א. ציירו treap הבנוי מהקבוצה הבאה של זוגות סדורים (האיבר הראשון בכל זוג הוא המפתח, והאיבר השני הוא העדיפות): ,(2,13), (5,34), (12, 41), (10,22), (15,27), (9,14), (7,68), (6,19), (8,26)

ב. הוכיחו שלכל קבוצה של זוגות סדורים המורכבים ממפתח ועדיפות קיים treap אחד ויחיד. הניחו שכל המפתחות שונים זה מזה וכל העדיפויות שונות זו מזו.

ג. כתבו אלגוריתם רקורסיבי הבונה treap עבור קבוצה נתונה *S* של *n* זוגות. מהו זמן הריצה של האלגוריתם במקרה הגרוע?

ד. כתבו אלגוריתם איטרטיבי הבונה treap עבור קבוצה נתונה *S* של *n* זוגות. על האלגוריתם לרוץ בסיבוכיות O(*n*log*n*) במקרה הגרוע. רמז: השתמשו בפעולת הכנסה לעח"ב.

תרגיל Treap: פתרון סעיף א



תרגיל Treap: פתרון סעיף ב

- ההוכחה באינדוקציה על גודל הקבוצה (הוכחת בנייה).
- *בסיס*: לקבוצה ריקה יש רק treap אחד – העץ הריק.
- *צעד*: תהי S קבוצה לא ריקה של זוגות. נראה של-S מתאים treap אחד ויחיד.
  - יהי (k, p) הזוג שעדיפותו מקסימלית. העדיפויות שונות, לכן יש רק זוג אחד כזה. כל בניה של treap עבור S חייבת להציב זוג זה בשורש העץ כדי לקיים את תכונת הערמה.
  - נסמן ב- L את הקבוצה המכילה את הזוגות שהמפתחות שלהם קטנים מ-k, ונסמן ב-R את הקבוצה המכילה את הזוגות שהמפתחות שלהם גדולים מ-k. שתי הקבוצות קטנות מ-S ולכן עפ"י הנחת האינדוקציה ישנו treap אחד ויחיד TL המכיל את הזוגות של L, ו-treap אחד ויחיד TRהמכיל את הזוגות של R. כל בנייה של treap עבור S חייבת להציב את TL כתת-עץ שמאלי של השורש, ואת TR כתת-עץ ימני של השורש (אחרת לא תתקיים תכונת עץ החיפוש).
  - מכאן שה-treap היחיד עבור S, הוא עץ בינרי T ששורשו x מכיל את הערכים הבאים: *key*[*x*]=*k*, *priority*[*x*]=*p*, *left*[*x*]=*TL*, *right*[*x*]=*TR*.

האוניברסיטה הפתוחה

תרגיל Treap: פתרון סעיף ג

- האלגוריתם הבונה את העץ ממיין את הקלט וקורא לאלגוריתם הרקורסיבי BuildTreap

BuildTreapTree(S)

Input: an array S of (key, priority) pairs

Output: the treap of S

1. $K \leftarrow$ MergeSort(S, $length[S]$) by key
2. $T \leftarrow$ a new treap node
3. $root[T] \leftarrow$ BuildTreap(K, 1, $length[K]$)
4. **return** T

- נכונות: BuildTreap מממש את ההוכחה מסעיף ב'
- זמן ריצה: זמן המיון $O(n\lg n)$ ועוד סה"כ זמני החיפוש בשורה 3.

לפי מבנה העץ בסעיף ב, סה"כ העלות היא $O(n^2)$

BuildTreap(K, p, r)

Input: a sub array $K[p .. r]$ of (key, priority) pairs, sorted by key

Output: root for the treap of K

1. **if** $p > r$
2. **then return nil**
3. $i \leftarrow$ index of pair with maximum priority in $K[p .. r]$
4. $x \leftarrow$ a new treap node
5. $key[x] \leftarrow key[K[i]]$
6. $priority[x] \leftarrow priority[K[i]]$
7. $left[x] \leftarrow$ BuildTreap(K, p, $i - 1$)
8. $right[x] \leftarrow$ BuildTreap(K, $i + 1$, r)
9. **return** x

תרגיל Treap: פתרון סעיף ד

העץ נבנה באמצעות אלגוריתם להכנסת איבר חדש ל-Treap, שיופעל *n* פעמים, על כל הזוגות הסדורים בקלט *S*.

1. הכנס את הצומת לעץ בשיטת ההכנסה לעח"ב, לפי הערך key.
2. אם מיקומו של הצומת שובר את תכונת הערימה לפי priority, הזז את הצומת כלפי השורש ע"י סדרת רוטציות, עד שיגיע למקומו.

<u>נכונות</u>: צעד 1 שומר על תכונת העח"ב, וצעד 2 מתקן את תכונת הערימה בלי לקלקל את תכונת הע"חב (רוטציה שומרת על סדר תוכי).

<u>סבוכיות</u>: $O(\log n)$ לכל הכנסה, סה"כ $O(n\log n)$
מכיוון שהעץ הוא עח"ב, האלגוריתם אופטימאלי

האוניברסיטה הפתוחה

הרחבת מבני נתונים

- לעתים קרובות ניתן "להרחיב" מבנה נתונים קיים כך שיתמוך גם בפעולות נוספות
  - לצורך כך אפשר לשמור מידע נוסף במבנה הנתונים
- ארבעת השלבים לביצוע ההרחבה:

א. בחר מבנה נתונים בסיסי בעל תכונות רצויות
- למשל עץ אדום שחור או ערימה

ב. קבע את המידע הנוסף שיש לאחסן במבנה הנתונים הבסיסי
- למשל שדה נוסף בכל צומת במבנה

ג. ודא שאפשר לתחזק ביעילות את המידע הנוסף במהלך ביצוע הפעולות הרגילות שמשנות את מבנה הנתונים הבסיסי
- למשל בעת הכנסה והוצאה מהמבנה

ד. ממש את הפעולות הנוספות הנדרשות, והוכח את סיבוכיותן
- לפעמים מוכתבת מלכתחילה סיבוכיות נדרשת, ויש לעמוד בה

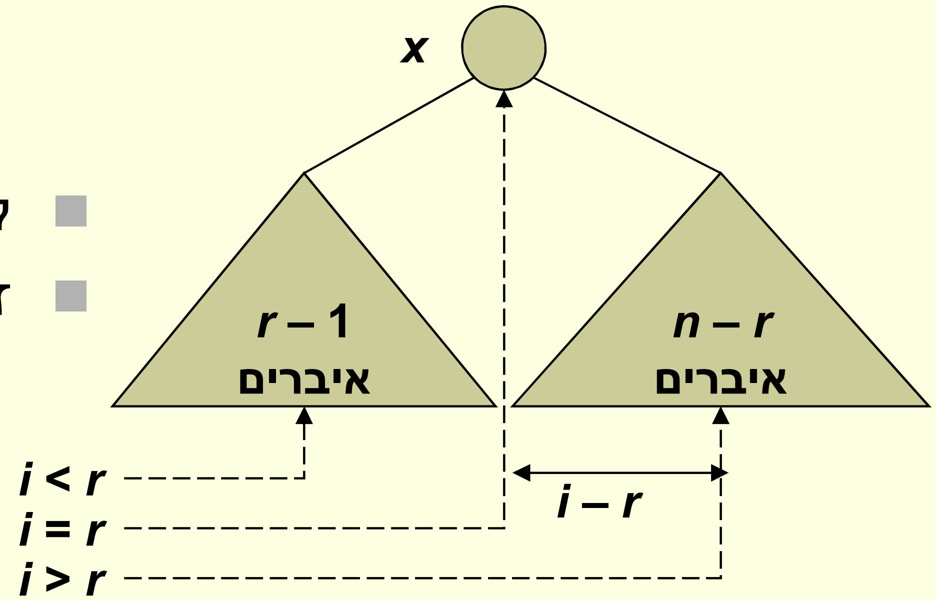
דוגמה – ערכי מיקום דינמיים

- הבעיה:
  - בהינתן קבוצה דינמית S בעלת n מפתחות, רוצים לבצע ביעילות שאילתות של חיפוש המפתח במיקום ה-i, *ושל דירוג מפתח נתון בקבוצה* S.
  - תזכורת: ערך המיקום ה-i הוא המפתח ה-i בגודלו בקבוצה
- נשתמש בהרחבה: עץ ערכי-מיקום (Order-Statistic Tree)
  - שלב א': נבחר כבסיס עץ אדום-שחור T
  - שלב ב': נוסיף לכל צומת x בעץ את השדה $size[x]$, אשר מוגדר כמספר הצמתים בתת-העץ המושרש בצומת x (נגדיר $size[nil[T]] = 0$)
    - בכל צומת x בעץ מתקיים: $size[x] = size[left[x]] + size[right[x]] + 1$
  - שלב ג': נראה כיצד ניתן לשנות את פעולות ההכנסה והמחיקה כך שהשדה *size* של כל צומת יישאר מעודכן
  - שלב ד': נוסיף את הפעולות OS-Select ו-OS-Rank
- הרעיון המרכזי של השימוש בשדה ההרחבה size:
  - יהי x שורש של תת-עץ כלשהו ב- T; אזי דירוגו *של המפתח של* x בקרב קבוצת המפתחות המאוחסנים בתת-העץ הוא: $size[left[x]]+1$

חיפוש מפתח לפי דירוג נתון

- בהינתן צומת *x* ודירוג *i*, מצא בתת-העץ המושרש ב-*x* את הצומת שמפתחו מדורג במקום ה-*i* בסדר הממוין של המפתחות בתת-העץ
- הרעיון הוא להשוות את *i* לדירוג *r* של *שורש תת-העץ x*

```
OS-Select(x, i)
r ← size[left[x]] + 1
if i = r
    then return x
if i < r
    then return OS-Select(left[x], i)
    else return OS-Select(right[x], i – r)
```

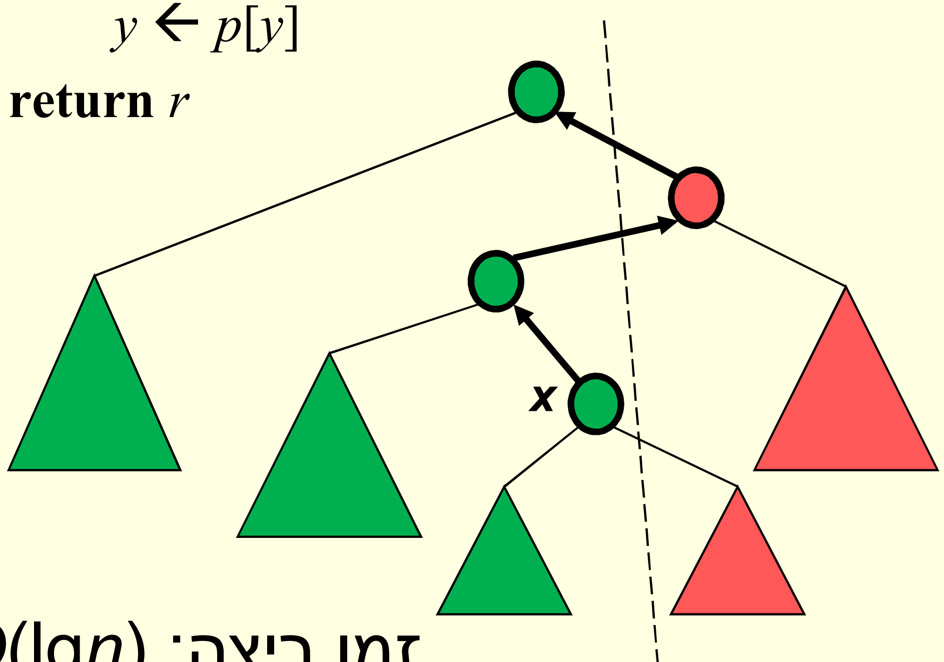


- קריאה חיצונית: **OS-Select**(*root*[*T*], *i*)
- זמן ריצה: $O(\lg n)$

דירוג של מפתח נתון

OS-Rank(T, x)

1. $r \leftarrow size[left[x]] + 1$
2. $y \leftarrow x$
3. **while** $y \neq root[T]$
4. **do if** $y = right[p[y]]$
5. **then** $r \leftarrow r + size[left[p[y]]] + 1$
6. $y \leftarrow p[y]$
7. **return** r



זמן ריצה: $O(\lg n)$

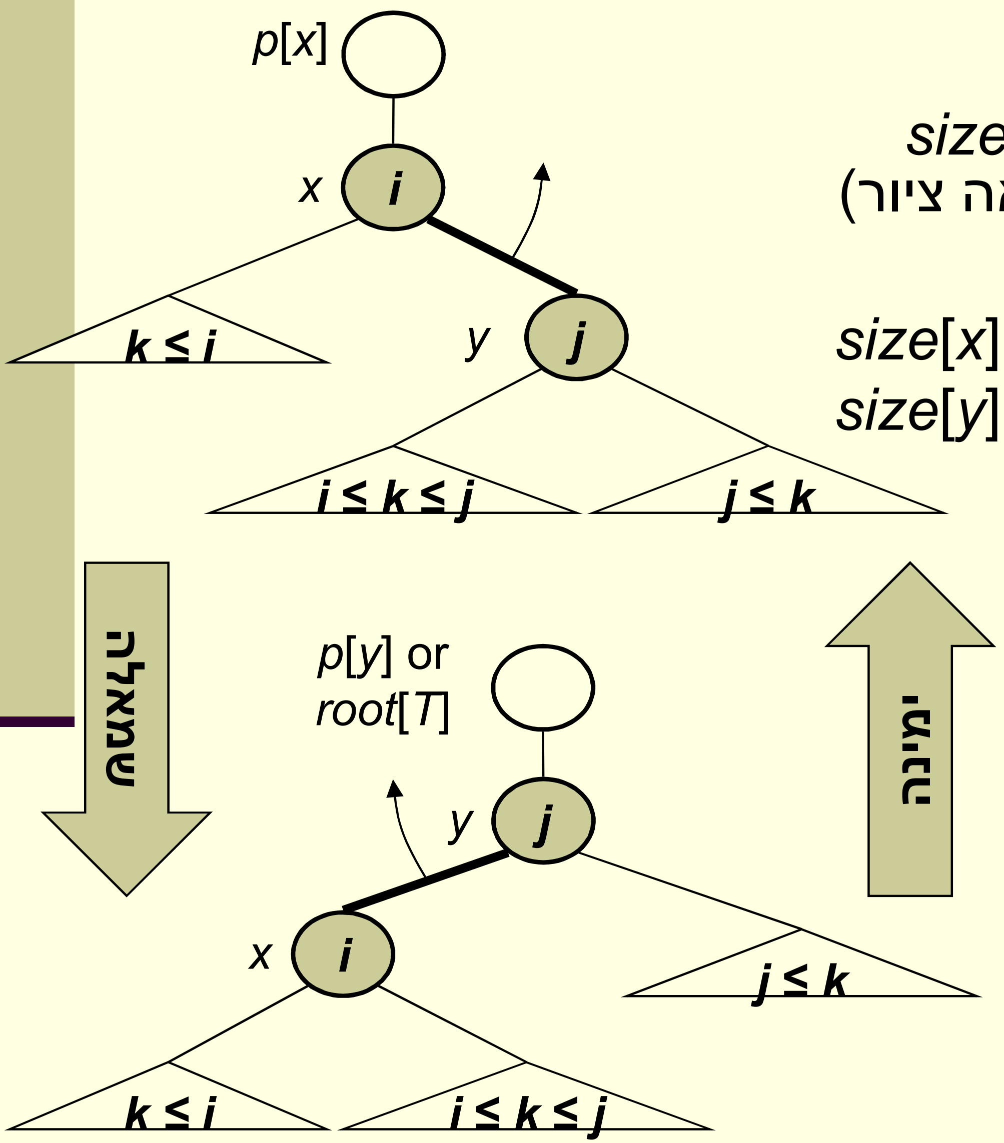
- בהינתן עץ T וצומת x, *מצא את דירוגו* של המפתח של x בסדר הממוין של המפתחות בעץ
- הרעיון: מטפסים מ-x אל שורש העץ וסופרים את מספר הצמתים שנמצאים "משמאל" למסלול הטיפוס
  - אם הצומת הנוכחי y *במסלול* הוא בן ימני, אז אביו וכל צמתי תת-העץ השמאלי של האב נמצאים <u>לפני</u> y בסריקה התוכית, ולכן הם נספרים
  - אחרת (y הוא בן שמאלי), אביו של y וכל צמתי תת-העץ הימני של האב נמצאים <u>אחרי</u> y בסריקה התוכית, ולכן אינם נספרים
- מהי שמורת הלולאה של שורות 3-6?
  - הערך r הוא הדרוג של x בתת-העץ המושרש ב-y

תחזוקת השדה *size* בהכנסה והוצאה מעץ א"ש

- בשלב א' בהכנסת המפתח k לעץ, נוסיף 1 לשדה *size* בכל צומת על המסלול מהשורש לנקודת החיבור
  - העלות היא $O(\log n)$
- בשלב א' בהוצאת הצומת z מהעץ, יהי y הצומת שיוצא בפועל; נפחית 1 מהשדה *size* בכל צומת על המסלול מ-y לשורש
  - העלות היא $O(\log n)$
- בשלב ג', בתיקונים של הכנסה או הוצאה, כל פעולת סיבוב מחייבת תיקון של השדה *size* בשני הצמתים המעורבים בסיבוב (ראה בשקף הבא)
  - העלות היא $O(1)$ לכל פעולת סיבוב
  - מספר הסיבובים חסום על ידי קבוע
- סה"כ עלות התחזוקה של size בכל הכנסה או הוצאה היא לפיכך $O(\log n)$
- <u>מסקנה</u>: תחזוקת השדה *size* אינה משנה את הסיבוכיות האסימפטוטית של הפעולות
- נקרא לפעולות המורחבות: OS-Insert(T, k), OS-Delete(T, z)

השפעת פעולת הסיבוב על השדה *size*

- ההשפעה היא רק על שני הצמתים הלוקחים חלק בסיבוב
  - אחרי הסיבוב נחשב מחדש את הערך *size* בשני צמתים אלה, בסדר המתאים (ראה ציור)
- אחרי סיבוב שמאלי
  - $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$
  - $size[y] \leftarrow size[left[y]] + size[right[y]] + 1$
- אחרי סיבוב ימני
  - השינויים סימטריים, בחילוף סדר החישוב בין *x* ל-*y*



תרגיל 14.1-7
ספירת היפוכים במערך

יהי $A[1 \,..\, n]$ מערך שכל אבריו שונים זה מזה.

- הראו כיצד ניתן להשתמש בעץ ערכי מיקום כדי למנות את מספר ההיפוכים (inversions) במערך בזמן $O(n\lg n)$.

תזכורת (ראו בעיה 2-4):

זוג האינדקסים (i, j) הוא **היפוך** במערך A אם $i < j$ וגם $A[i] > A[j]$

לדוגמה: במערך $A = [2, 3, 8, 6, 1]$ יש חמישה היפוכים:

(1,5), (2,5), (3,4), (3,5), (4,5)

פתרון תרגיל 14.1-7
ספירת היפוכים במערך

- נבנה מהמערך A עץ ערכי מיקום באמצעות n פעולות insert עוקבות
- נתבונן באיבר $A[1]$
  - כל איבר במערך שקטן ממנו בערכו מוסיף היפוך
  - כל איבר כזה מופיע בסריקה התוכית של העץ <u>לפני</u> $A[1]$
- לכן, נחפש בעץ את הצומת x שמפתחו A[1]
  - מספר ההיפוכים בהם משתתף $A[1]$ הוא הדירוג של הצומת x, פחות 1 (יש להפחית 1 כדי לא לספור גם את $A[1]$ עצמו בהיפוכים)
- לאחר שקיבלנו את מספר ההיפוכים בהם $A[1]$ משתתף, נמחק את הצומת x מהעץ
  - המחיקה מבטיחה שבאיטרציה הבאה נחשב בצורה נכונה את מספר ההיפוכים בהם משתתף $A[2]$, ללא חזרות על היפוכים שכבר נספרו
- נחזור על התהליך עם האיברים $A[2], A[3], \ldots, A[n-1]$
  - אין צורך לחשב עבור $A[n]$, כי כל ההיפוכים בהם הוא משתתף כבר נספרו

פתרון תרגיל 14.1-7 (המשך)
ספירת היפוכים במערך

- האלגוריתם

```
Inversion-Count(A)
1.  T ← new order-statistic tree
2.  for i ← 1 to length[A]
3.       do OS-Insert(T, A[i])
4.  count ← 0
5.  for i ← 1 to length[A] – 1
6.      do x ← Tree-Search(T, A[i])
7.          r ← OS-Rank(T, x)
8.          count ← count + r – 1
9.          OS-Delete(T, x)
10. return count
```

- זמן הריצה: $O(n\lg n)$
  - בניית העץ: $O(n\lg n)$
  - ספירת ההיפוכים: $O(n\lg n)$
    - לולאה 5-9 רצה n–1 פעמים
    - חיפוש מפתח (שורה 6), החזרת הדירוג (שורה 7), ומחיקת צומת (שורה 9): $O(\lg n)$ לכל שורה
  - שאר השורות: $O(1)$