Data Structures

הרצאה 19 – עץ +B

עץ חיפוש

- גובהו של עץ **בינארי** שלם עם **n** צמתים הוא בערך $\log_2 n$
- גובהו של עץ **5-נארי** שלם עם **n** צמתים **נמוך יותר** - $\log_5 n$
- גובהו של עץ **b-נארי** שלם עם $b \geq 2$ ו- **n** צמתים - $\log_b n$
- שימו לב ש- **b** יכול להיות תלוי ב- **n** !

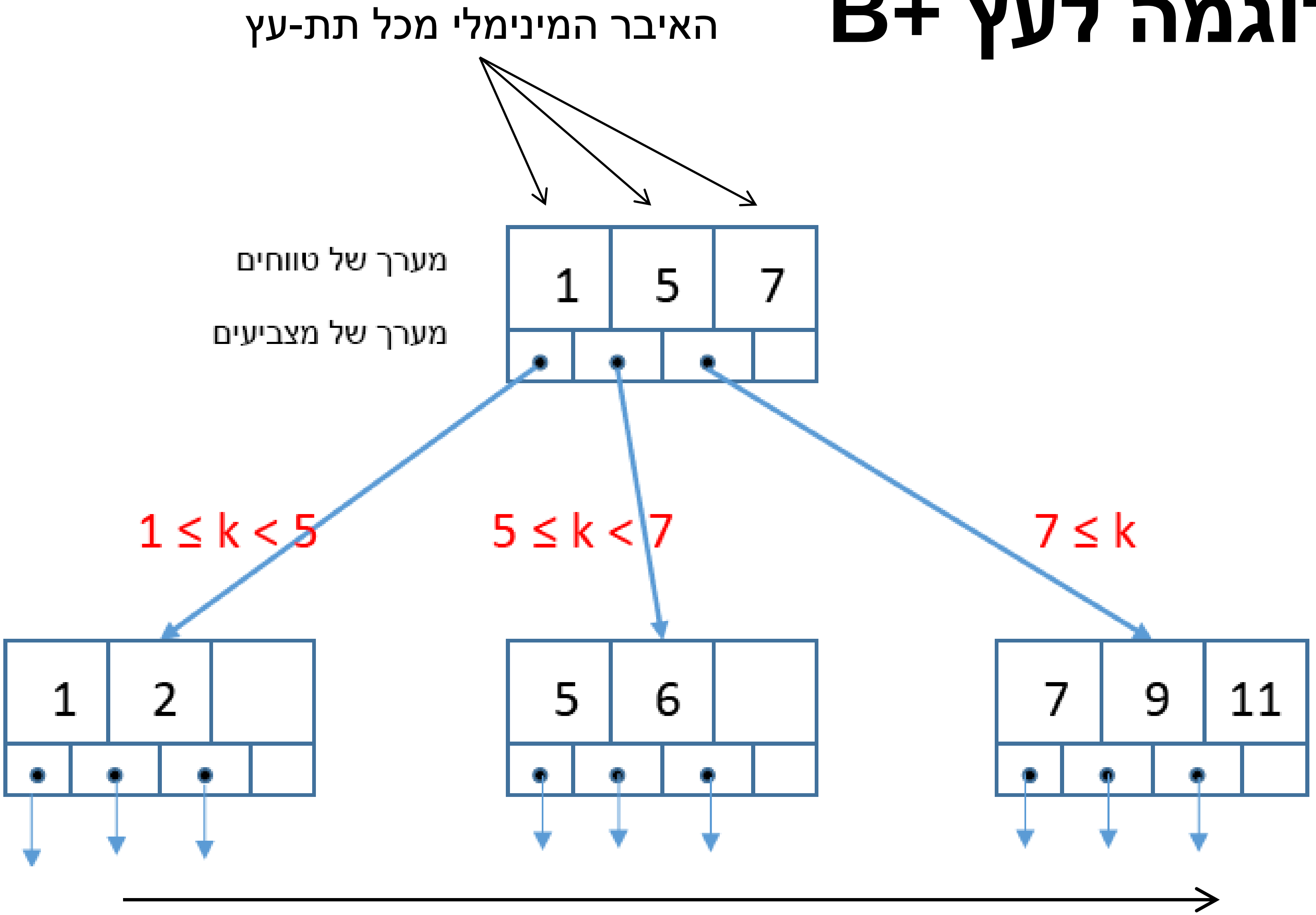
מתי כדאי לנו להשתמש במבנה כזה?

- כאשר רוצים להקטין ככל הניתן את מספר הגישות לצמתים ובתמורה מוכנים שכל צומת יאחסן יותר מידע
- זה חשוב כאשר עובדים עם התקן חיצוני, כמו דיסק-און-קי או כונן קשיח, שהגישה אליו כל פעם היא "יקרה" בזמן

עץ B+

- עץ **B+** הוא **עץ חיפוש** שנוח לשימוש במערכות קבצים
- במערכות אלו מידע מהזיכרון נשלף בבלוקים ב**גודל קבוע**
- שליפת בלוק מהזיכרון היא פעולה בעלות גבוהה יחסית
- לכן במקרים כאלו זהו רעיון טוב לייחס לכל צומת מידע המתאים לגודל הבלוק
- כך, כשנזדקק להסתכל על צומת, ננצל את פעולת השליפה היקרה להחזיר הרבה מידע שימושי, ולא רק פריט אחד
- עץ **B+** הוא עץ חיפוש **לא בינארי** המהווה הכללה של עץ חיפוש בינארי

דוגמה לעץ B+



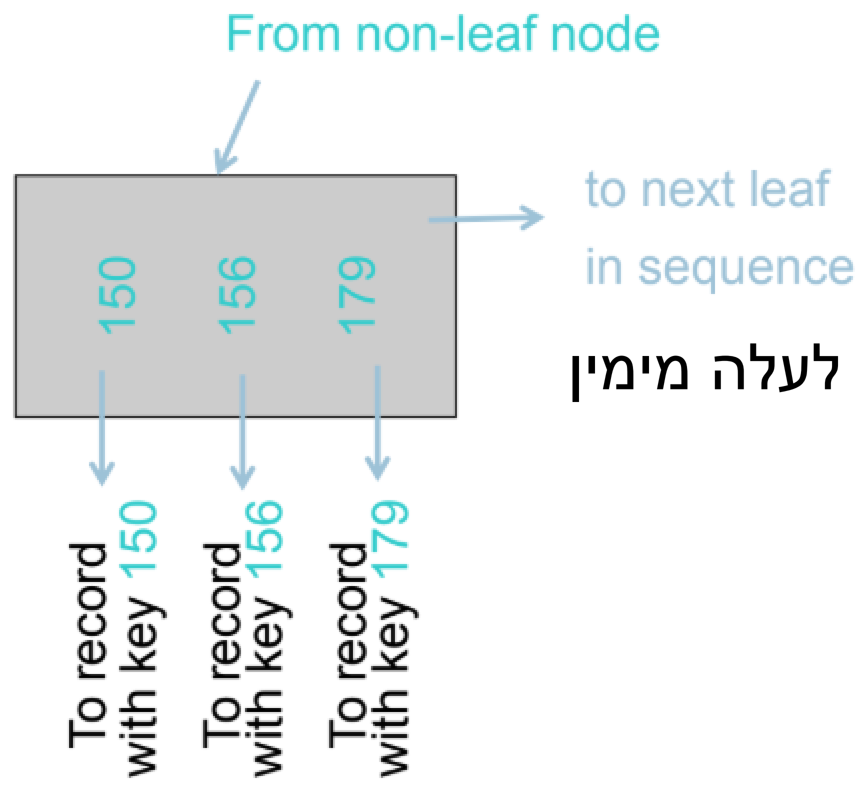
הדפסת **העלים** של עץ **B+** בסדר **in-order** מתאימה לסדרה ממוינת בסדר עולה

דוגמה לעץ B+

מבנה העלים מעט שונה ממבנה הצמתים פנימיים:

במקום להצביע על צמתים נוספים הם מצביעים על **תוכן**

צומת עלה



ובנוסף מצביע לעלה מימין

הגדרת עץ B+ (עם פרמטר b>1)

- **עלים**: היכן שהמצביעים **לנתונים עצמם** נשמרים

כל עלה מאחסן **לכל היותר b** נתונים **ולכל הפחות $\lceil b/2 \rceil$** נתונים

- **צומת פנימי**: מצביע על ילדיו
  - **מספר הילדים** הוא **לכל היותר b** ו**לכל הפחות $\lceil b/2 \rceil$**
  - בנוסף, צומת שומר את הערך הקטן ביותר בכל תת-עץ שלו
- **שורש**: מקיים תנאים פחות נוקשים
  - שורש שהוא גם **עלה** מכיל **לפחות נתון 1 ולכל היותר b** נתונים.

    בפרט, אם העץ שומר פחות מ- **b** איברים, אז השורש הוא עלה שמאחסן את **כל האיברים**
  - מספר הילדים של שורש שהוא **צומת פנימי** הוא **לפחות 2 ולכל היותר b**
- **המרחק מהשורש לכל העלים הוא שווה**

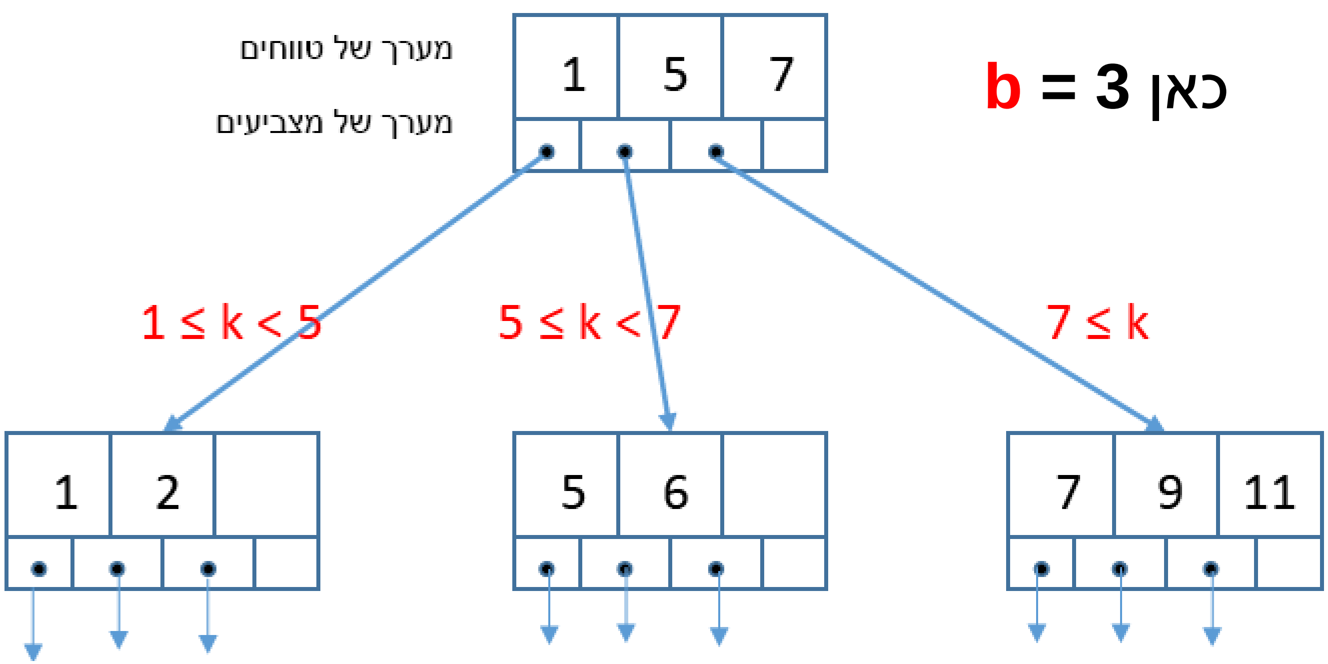
תכונות עץ B+ (עם פרמטר b)

עץ B+ הוא **מאוזן**:

- לכל צומת פנימי (חוץ מאולי השורש) יש $\Theta(b)$ ילדים
- המרחק מהשורש לכל העלים הוא שווה
- לכן מספר הרמות בעץ הוא $O(\log_b n) = O(\log n/\log b)$

(שימו לב ש- **b** יכול להיות תלוי ב- **n!**)

כאן b = 3



חיפוש בעץ B+

החיפוש נעשה כמו בעץ חיפוש רגיל

- כל צומת פנימי שומר את האיבר הקטן ביותר מכל תת-עץ שלו
  לכן ניתן למצוא בקלות עם איזה ילד יש להמשיך את החיפוש
- זמן הריצה לחיפוש הוא $O(b \cdot \log_b n)$
  - יש $O(\log_b n)$ רמות ולכל צומת ברמה יש לכל היותר **b** איברים
- ניתן לשפר את זמן הריצה ע"י שימוש **בחיפוש בינארי בכל צומת** על מנת למצוא עם איזה ילד להמשיך

  במקרה זה זמן הריצה הוא רק:

$$O(\log b \cdot \log_b n) = O(\log b \cdot (\log n / \log b)) = O(\log n)$$

(זכרו ש- **b** יכול להיות תלוי ב- **n** !)

אז איפה השיפור לעומת עץ חיפוש בינארי רגיל?

חיפוש בעץ B+

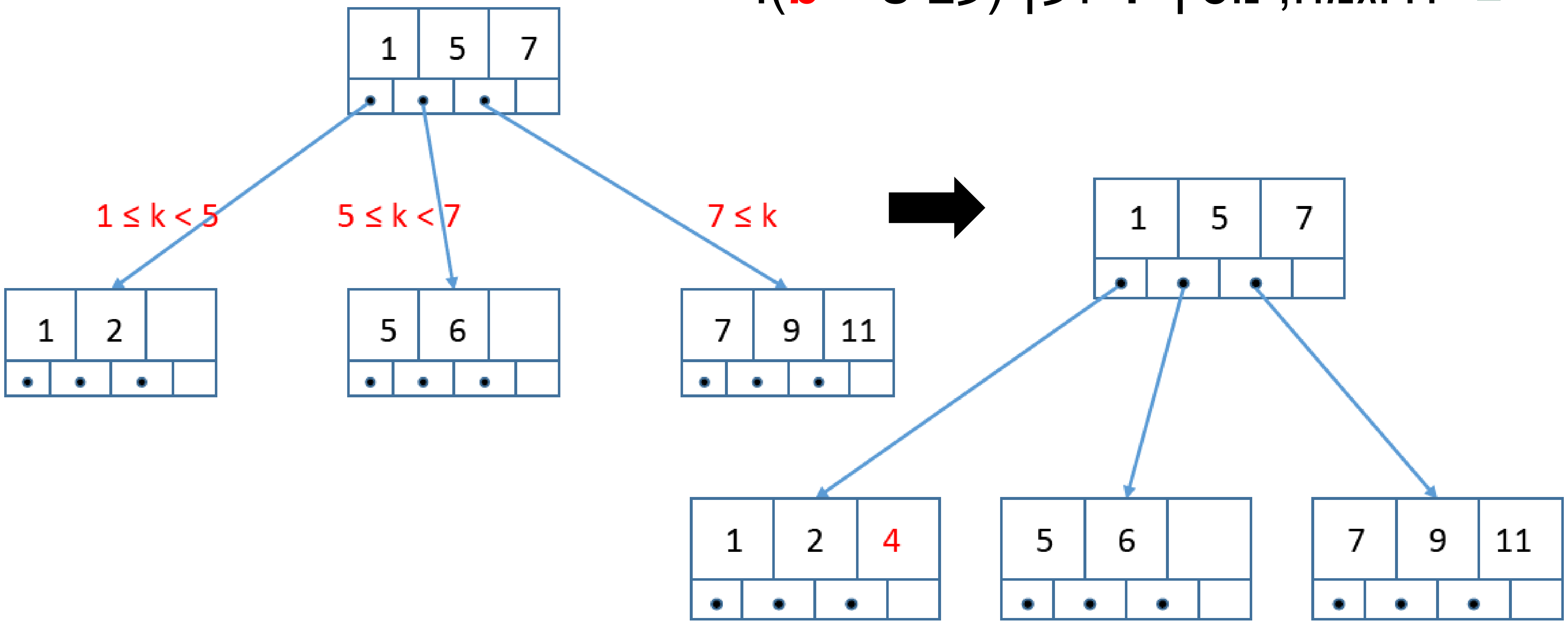
איפה השיפור לעומת עץ חיפוש בינארי רגיל?

- השיפור הוא בפעולות ה"כבדות" של **שליפת המידע כבלוקים**
- שליפה כזו מתרחשת כל פעם כשיורדים רמה בעץ
- מכיוון שיש $O(\log_b n)$ רמות, החיפוש מבקר רק ב- $O(\log_b n)$ צמתים
- ולכן קורא למערכת הקבצים לשלוף מידע רק $O(\log_b n)$ פעמים במקום $O(\log_2 n)$
- ערך טיפוסי של **b** הוא בין 50 ל- 2000

השימוש בעץ **B+** הוא **נפוץ ביותר**, למשל לבניית אינדקס עבור מערכת הקבצים במערכות הפעלה שונות

הכנסת איבר לעץ B+

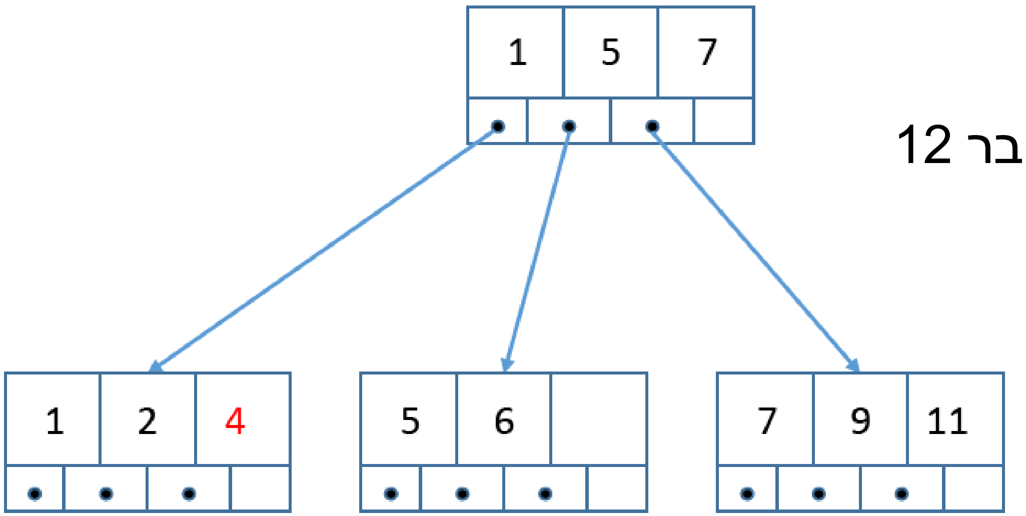
- אחרי שמצאנו על ידי החיפוש הנ"ל באיזה עלה יכנס האיבר החדש, מכניסים אותו לתוך המערך הפנימי של העלה
- אם יש מקום פנוי במערך שבעלה המתאים, פעולה זאת דורשת רק הזזות בתוך המערך על מנת לפנות מקום, בזמן **O(b)**
- לדוגמה, נוסיף **4** לעץ (עם **b = 3**):



הכנסת איבר לעץ +B

אם לעלה כבר יש b איברים אז המערך הפנימי שלו **מלא**

במקרה זה **מפצלים את העלה לשניים**:



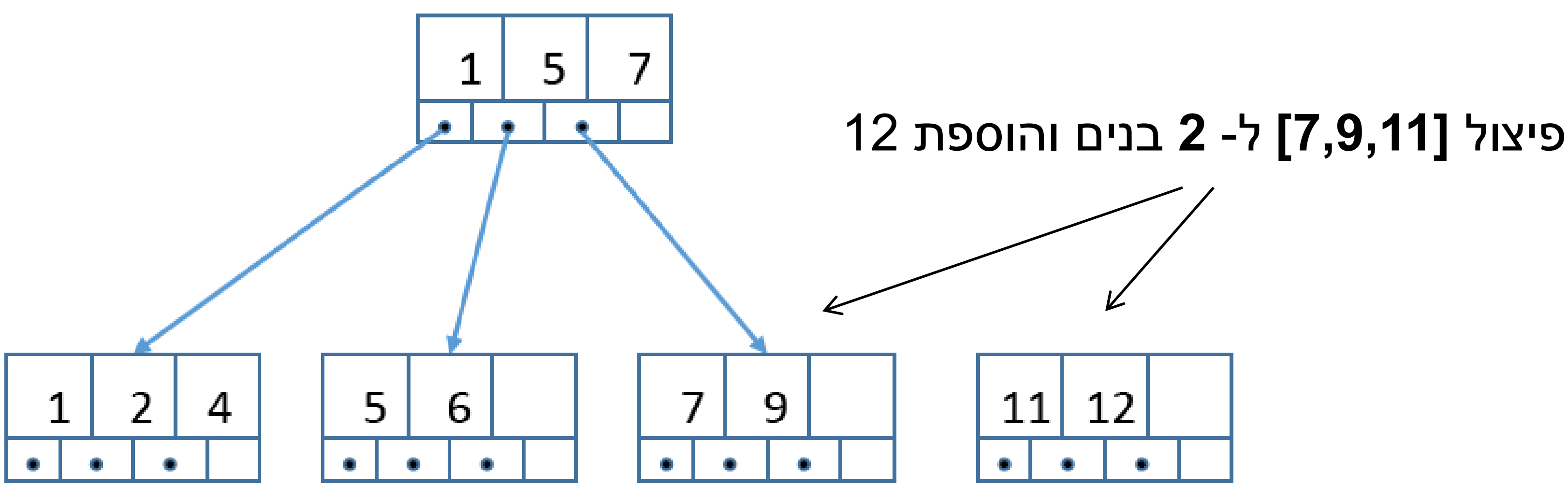
לדוגמה, רוצים להוסיף לעץ את האיבר 12

הכנסת איבר לעץ B+

אם לעלה כבר יש **b** איברים אז המערך הפנימי שלו **מלא**

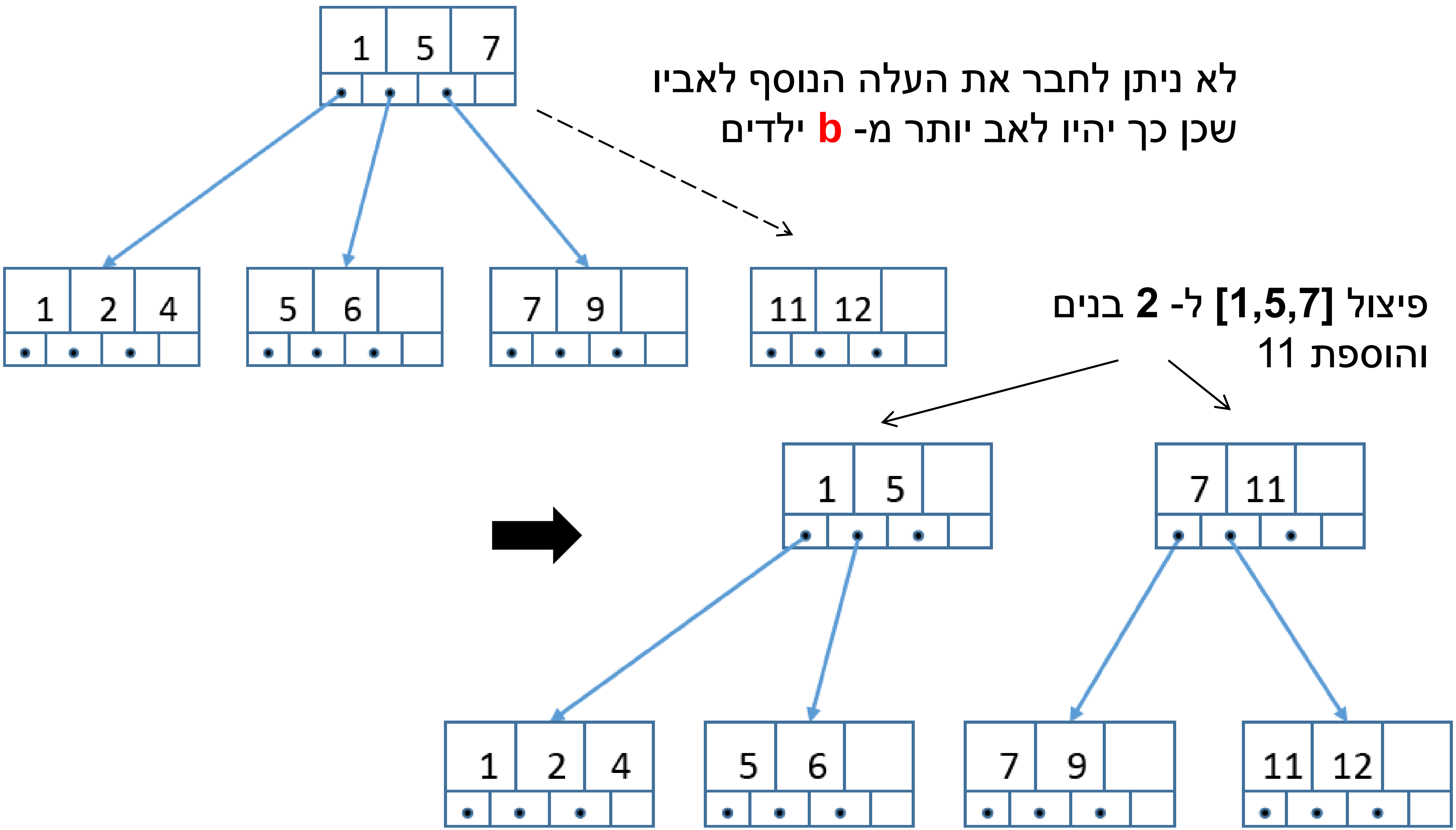
במקרה זה **מפצלים את העלה לשניים**:

- במקומו בונים שני עלים, לראשון חצי מהאיברים, ולשני החצי השני
- אחרי הפיצול, בכל עלה יהיו לפחות $\lceil b/2 \rceil$ נתונים
- מוסיפים את העלים החדשים כילדים של האב בזמן **O(b)** ומעדכנים טווחים



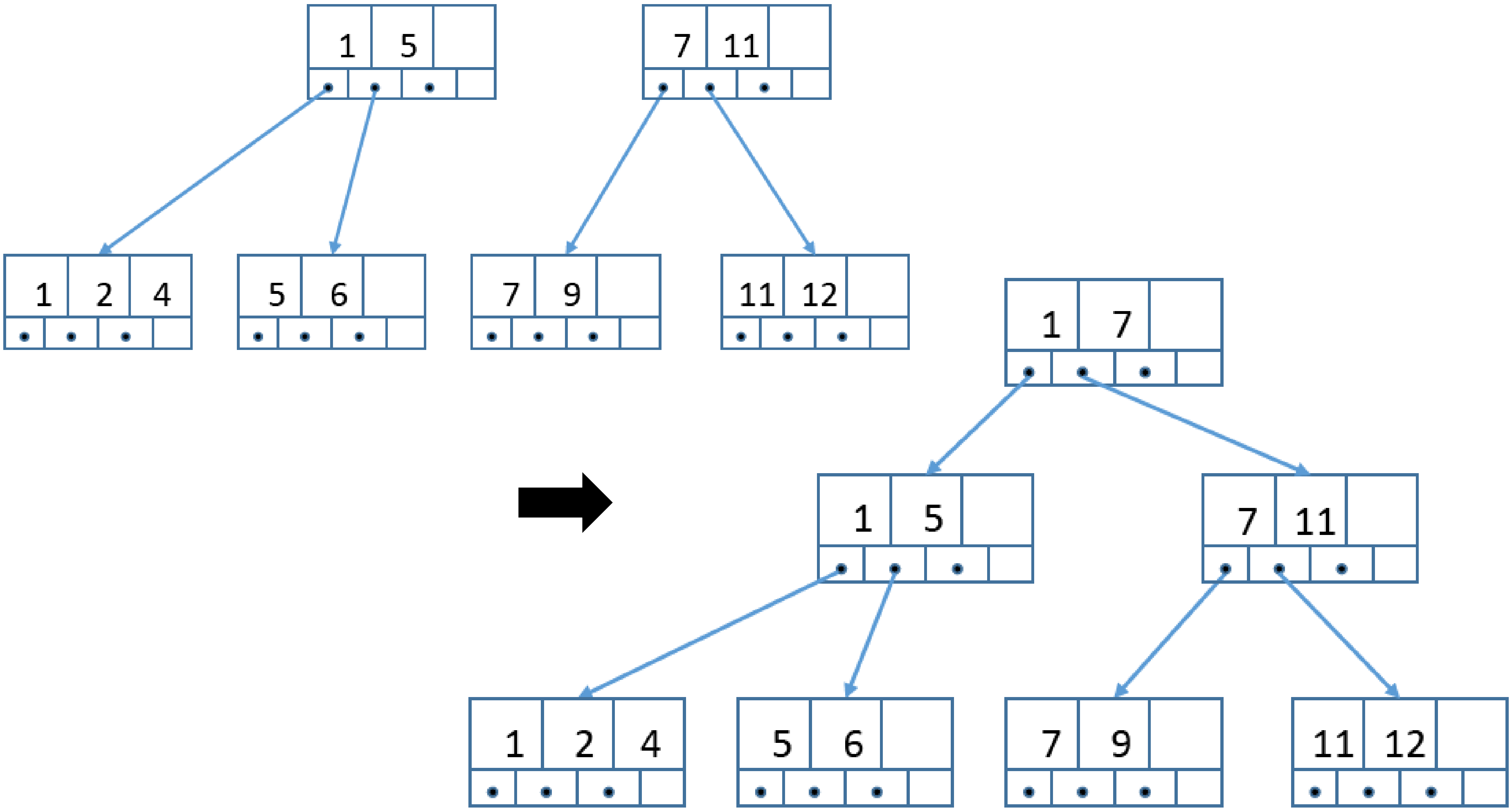
הכנסת איבר לעץ B+

- אם לאב כבר יש **b** ילדים, מפצלים גם אותו, וחוזרים חלילה

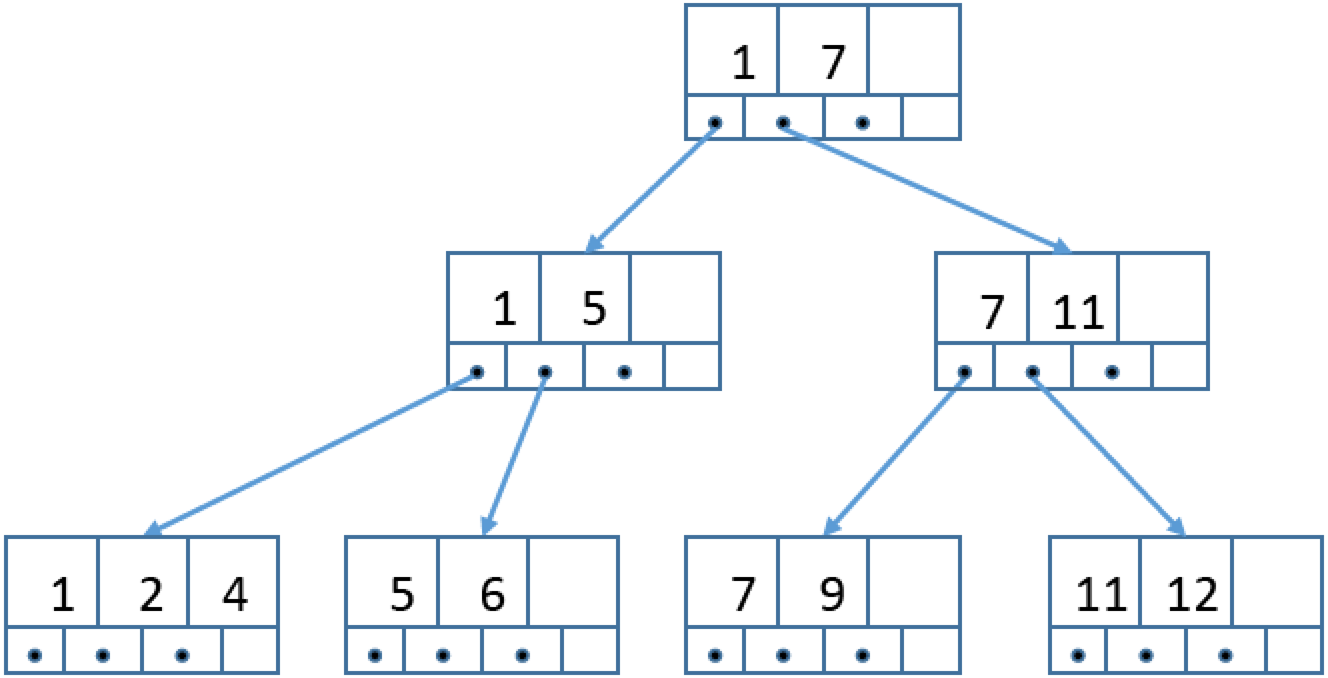


הכנסת איבר לעץ +B

- תתכן סדרת פיצולים שמגיעה עד לשורש – ובמקרה זה **נפצל את השורש** ונבנה צומת **שורש חדש**



הכנסת איבר לעץ B+



- שימו לב שרק פיצול של השורש יכול לשנות את גובה העץ
  כך נשמרת התכונה שהמרחק מהשורש לכל העלים שווה
- זמן ריצה להכנסת איבר $O(b \cdot \log_b n)$

מחיקת איבר מעץ +B

במקרה של מחיקת איבר, מוחקים את האיבר מהעלה

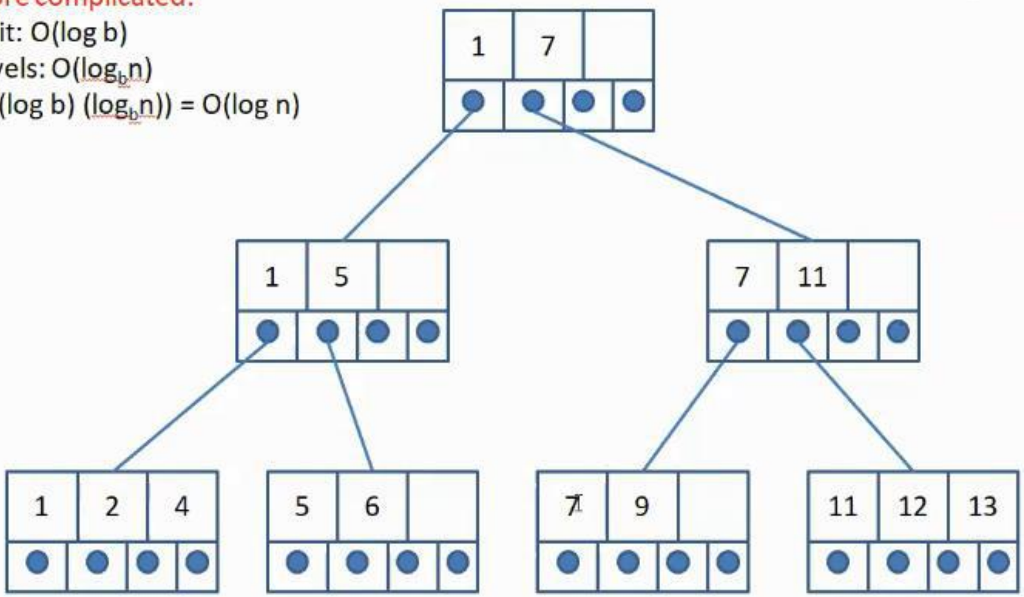
- אם נשארו לעלה לפחות $\lceil b/2 \rceil$ איברים, אין צורך בתיקון (מלבד עדכון הטווחים אם נדרש)
- אם לעלה נשארו פחות מ- $\lceil b/2 \rceil$ עלים, מסתכלים על **אחיו**:
  - אם לאח יש יותר מ- $\lceil b/2 \rceil$ איברים, מעבירים איבר אחד לעלה החסר ומעדכנים את הטווחים באבות
  - אחרת, מחברים את שני העלים ביחד, ומעדכנים את האב
  - עדכון זה יכול לגרום לעוד תיקונים בעץ ברמה הבאה, עד השורש
- זמן ריצה למחיקת איבר $O(b \cdot \log_b n)$

דוגמה: מחיקת 7

כאן $b = 3$

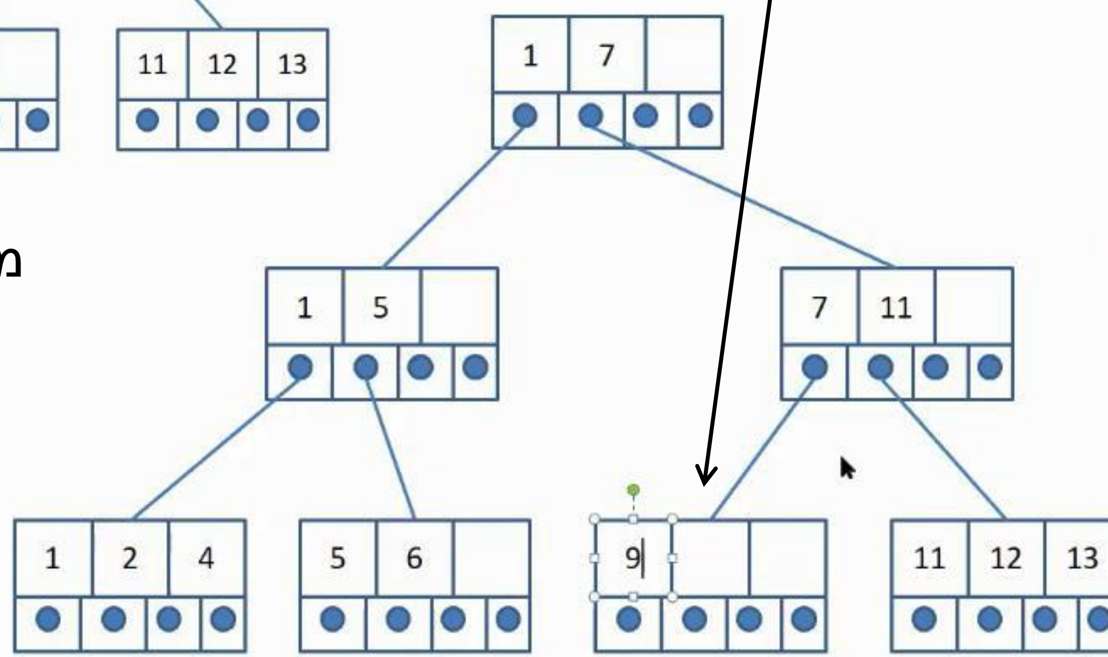
Deletion: Array
Split: $O(b)$
Levels: $O(\log_b n)$
$O(b \log_b n)$

More complicated:
Split: $O(\log b)$
Levels: $O(\log_b n)$
$O((\log b) (\log_b n)) = O(\log n)$



מוחקים את 7

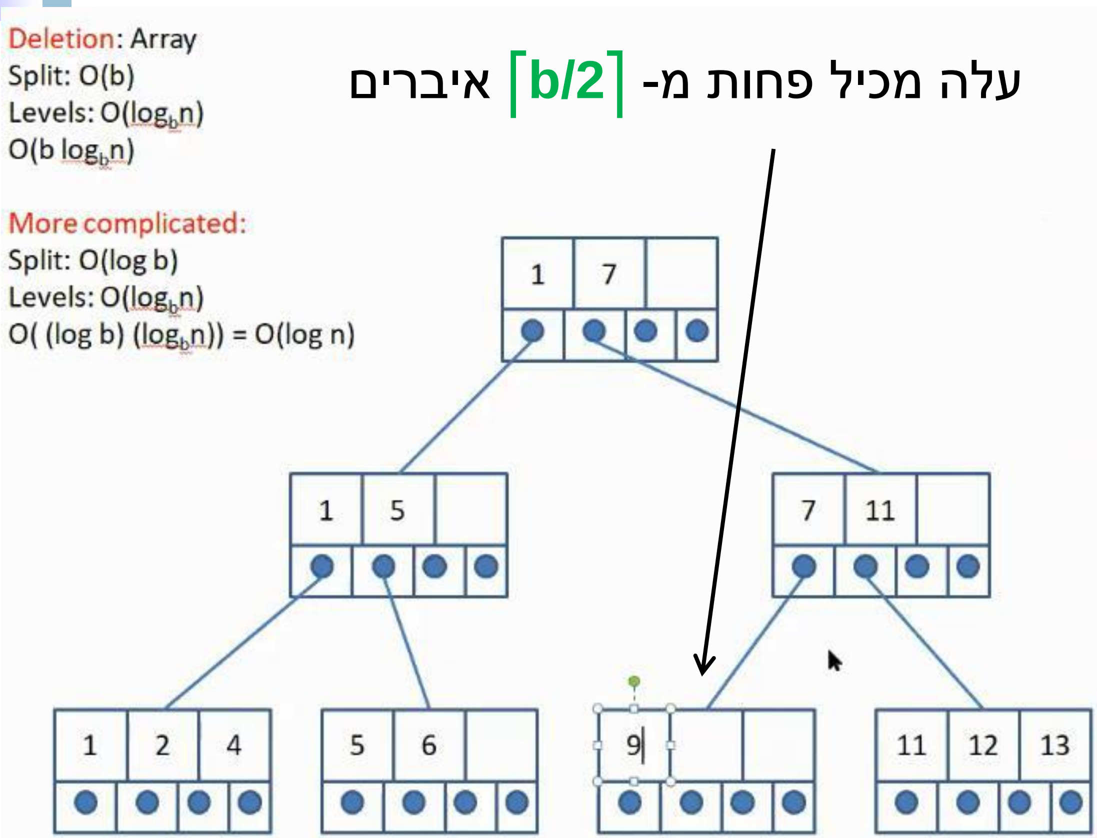
עלה מכיל פחות מ- $\lceil b/2 \rceil$ איברים



דוגמה: מחיקת 7

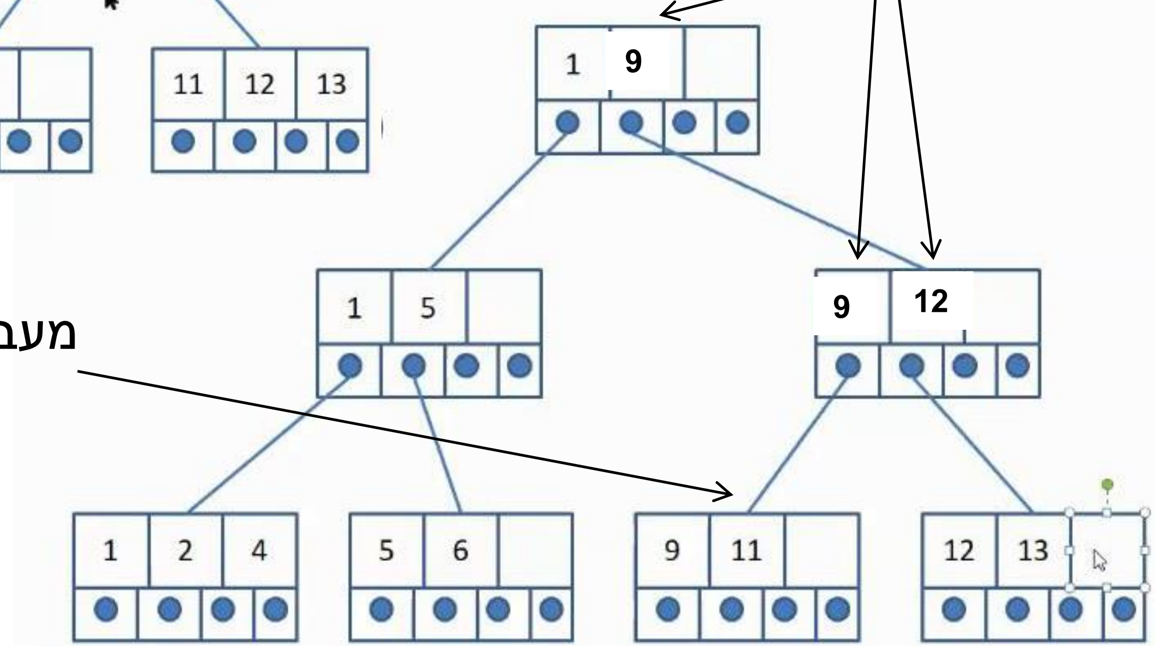
כאן $b = 3$

עלה מכיל פחות מ- $\lceil b/2 \rceil$ איברים

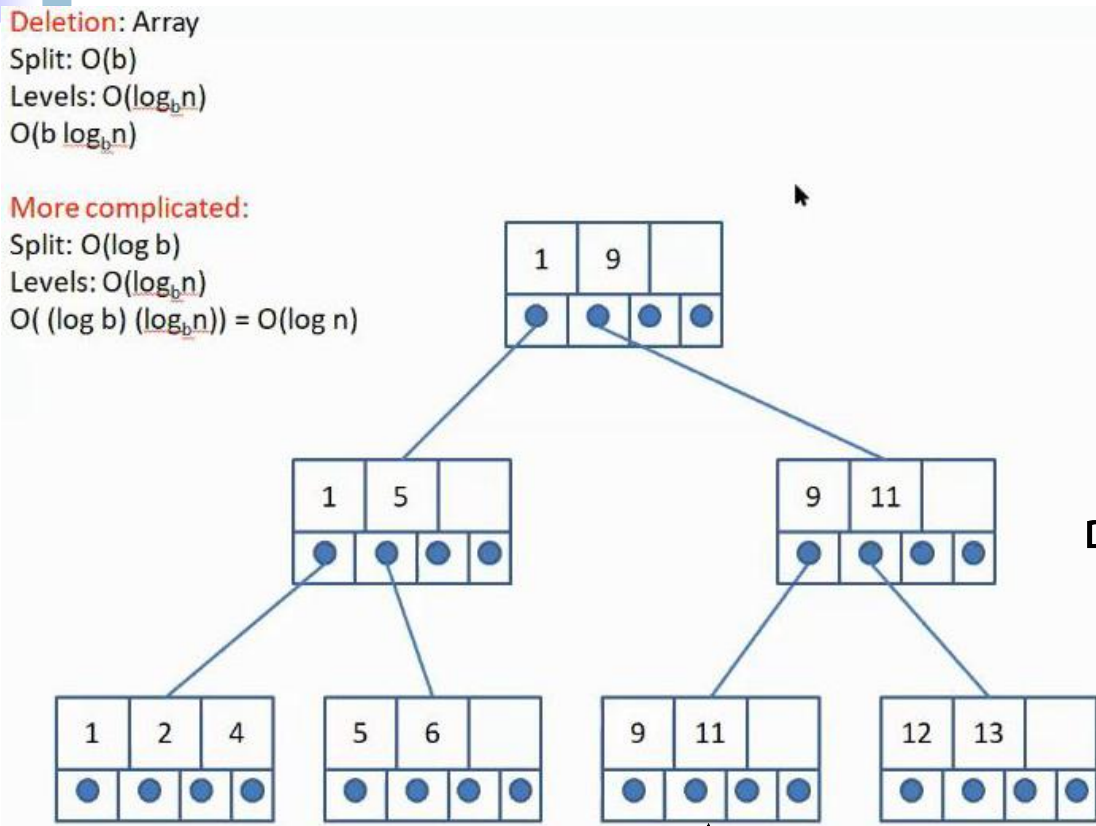


ומעדכנים את האבות

מעבירים איבר מהאח הימני



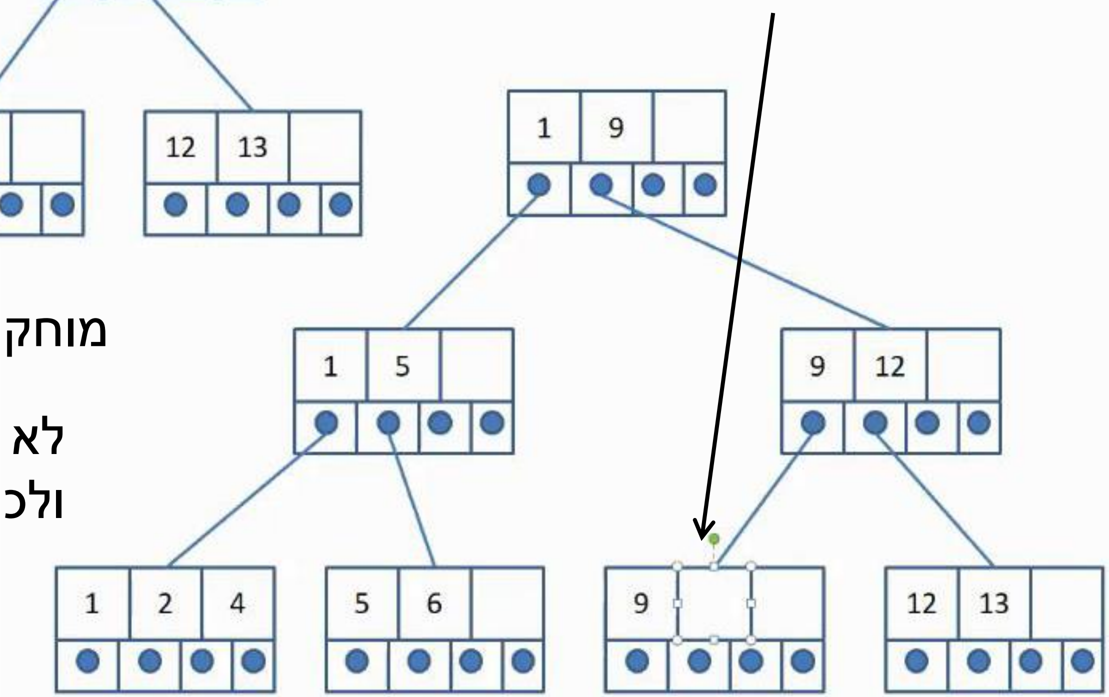
דוגמה: עכשיו נמחק את 11



מוחקים את 11

לא ניתן להעביר איבר מאחיו
ולכן נאחד את העלים

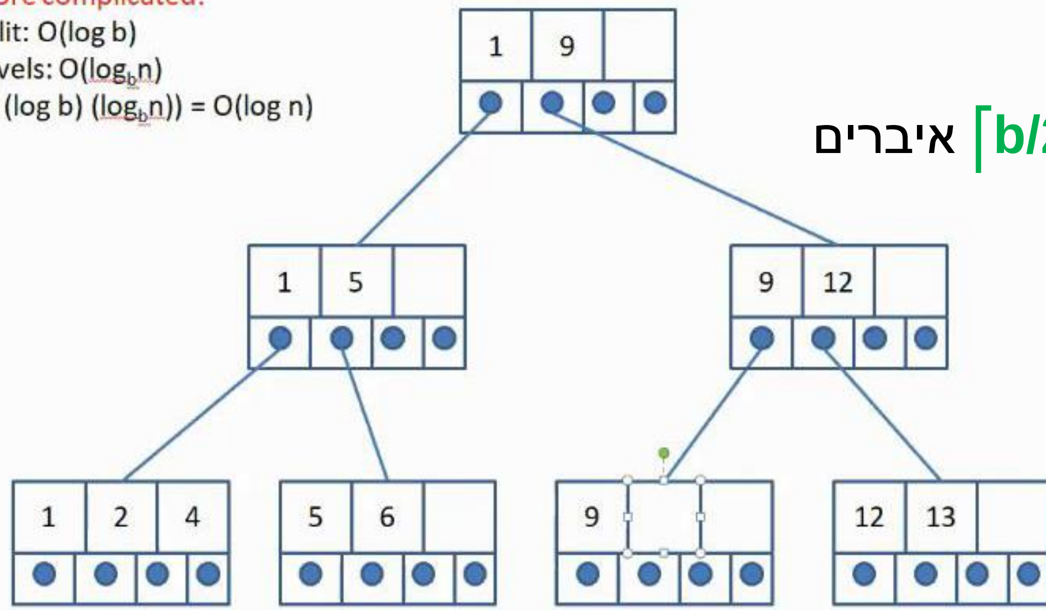
עלה מכיל פחות מ- $\lceil b/2 \rceil$ איברים



דוגמה: מחיקת 11

Deletion: Array
Split: O(b)
Levels: $O(\log_b n)$
$O(b \log_b n)$

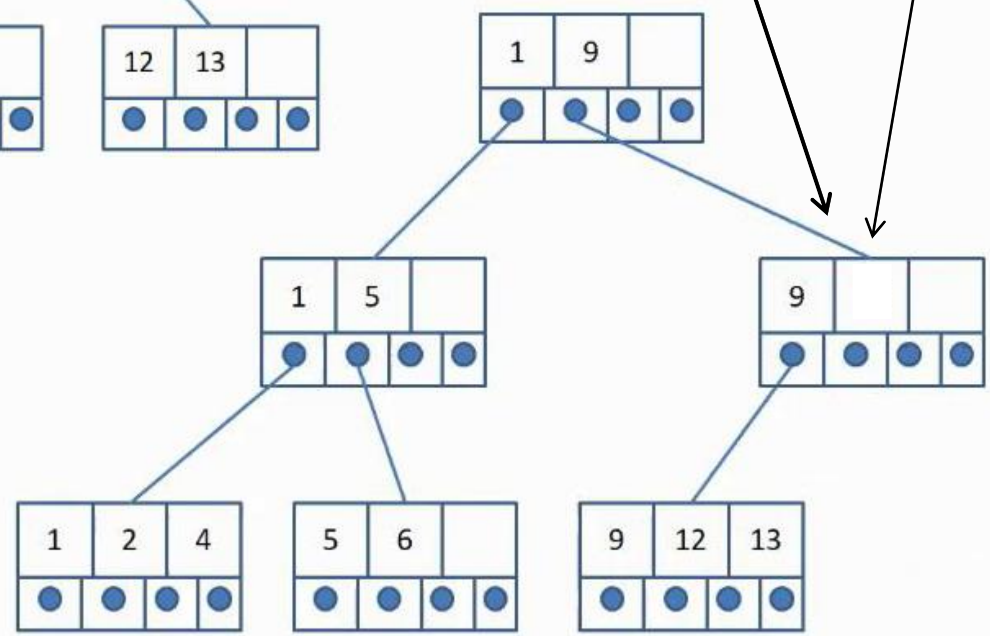
More complicated:
Split: O(log b)
Levels: $O(\log_b n)$
$O((\log b)(\log_b n)) = O(\log n)$



שוב קבלנו צומת המכיל פחות מ- $\lceil b/2 \rceil$ איברים

ועדכון האב

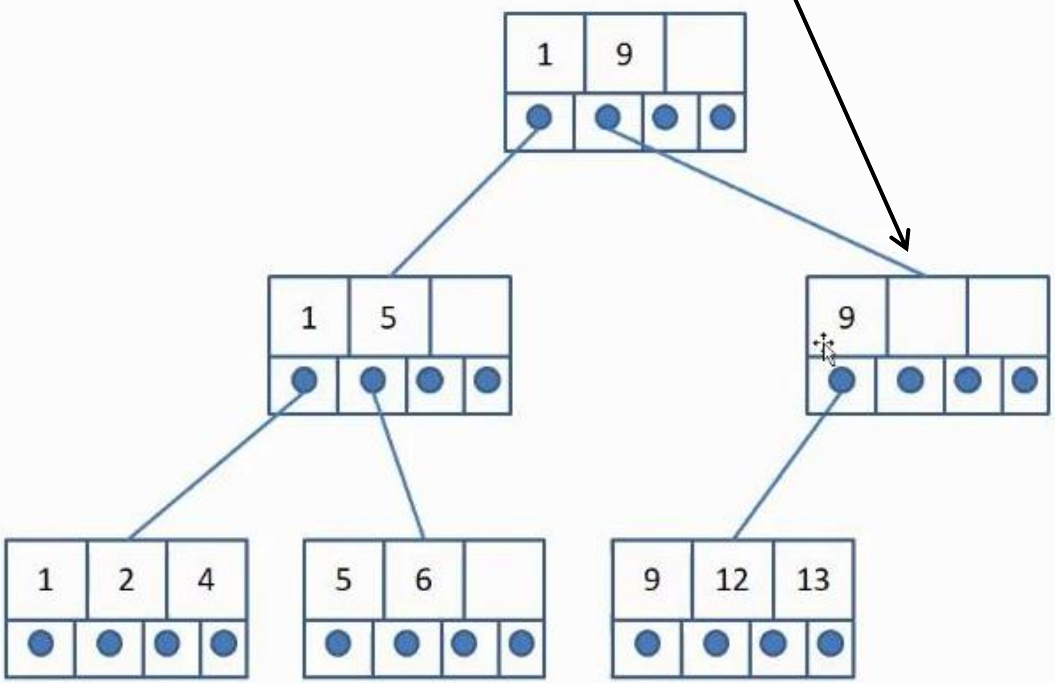
איחוד העלים



דוגמה: נמחק את 1

שוב קבלנו עלה המכיל פחות מ- $\lceil b/2 \rceil$ איברים

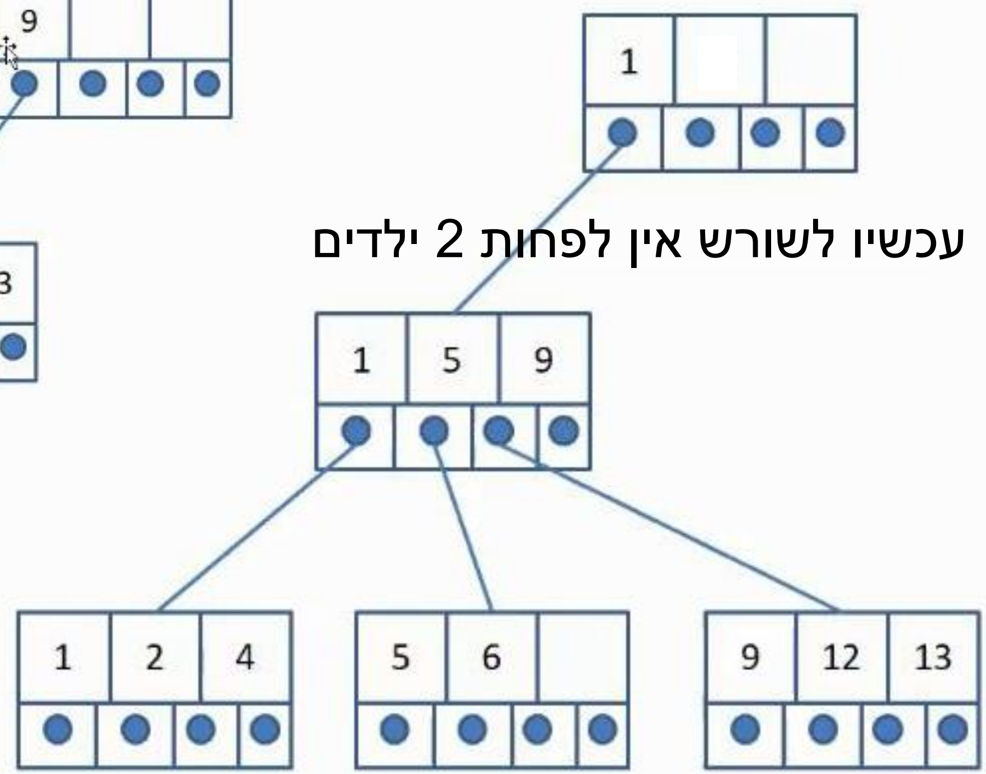
לא ניתן להעביר איבר מאחיו
ולכן נאחד את הצמתים



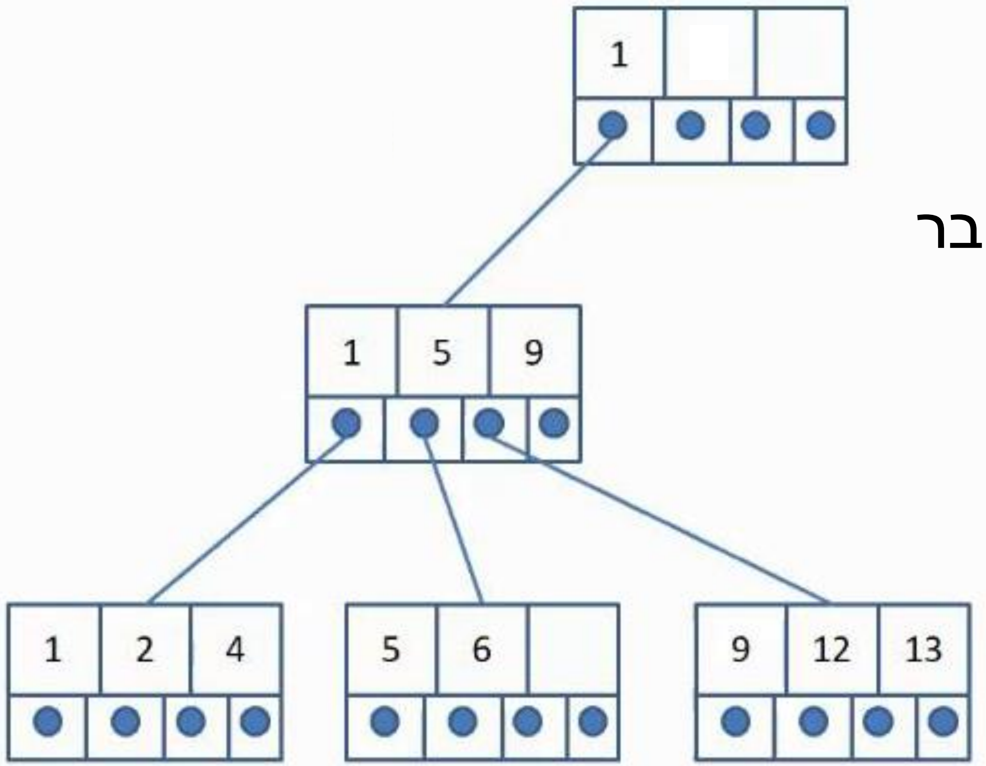
ועדכון האב

עכשיו לשורש אין לפחות 2 ילדים

איחוד הצמתים הפנימיים



דוגמה: נמחק את 11

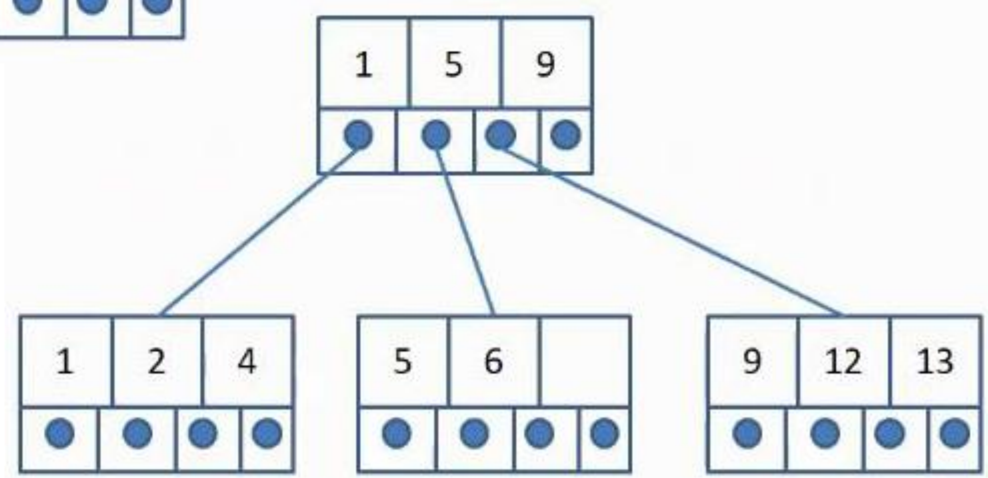


שימו לב כי בדוגמה זו מחיקת איבר הקטינה את גובה העץ ב- 1

עכשיו לשורש אין לפחות 2 ילדים

אז פשוט נמחק את השורש

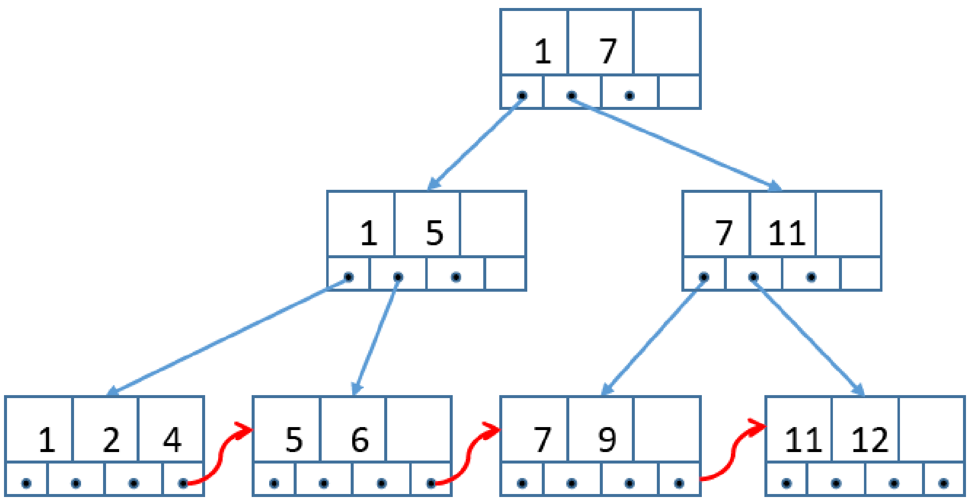
מחיקת השורש



יתרון נוסף של עץ B+

נגיד שאנו רוצים לענות על שאלות **range query**

- כלומר להחזיר את כל האיברים שהם גדולים ממספר **x** וקטנים ממספר **y**
- נוסיף לכל עלה מצביע לאחיו מצד ימין



- עכשיו מספיק לחפש את **x** בעץ וזה לוקח **O(log n)**
- ובעזרת המצביעים לעבור על כל האיברים בכל העלים עד שנגיע ל- **y**
- אם יש **k** איברים באוסף בין **x** ל- **y**, זמן הריצה הוא **O(log n + k)**
- זו דוגמה ליתרון בשמירת הנתונים בעלים, בניגוד לעץ **AVL** או עץ אדום-שחור

האצת הכנסה ומחיקה

- כמו שראינו, שמירת הנתונים במערך גורמת לנו לעשות **O(b)** פעולות בכל צומת בפעולת הכנסה ומחיקה של איבר
- אם בכל צומת נשמור את הנתונים במבנה מאוזן המאפשר חיפוש, פיצול וחיבור בסיבוכיות לוגריתמית, אז גם הכנסת ומחיקת איבר בכל צומת מצריכה רק **O(logb)** פעולות!

  (קיימים עצים מאוזנים כאלה אבל לא נלמד עליהם בקורס זה)
- לכן במימוש כזה, זמן ריצה לחיפוש, הכנסת ומחיקת איבר הינו רק

$$\mathbf{O(\log b \log_b n) = O(\log n)}$$

קוד מימוש עץ **+B** בג'אווה:

- https://www.programcreek.com/java-api-examples/?code=andylamp%2FBPlusTree%2FBPlusTree-master%2Fsrc%2Fmain%2Fjava%2Fds%2Fbplus%2Fbptree%2FBPlusTree.java