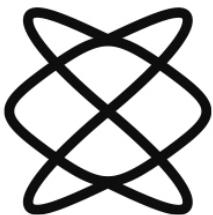


החולג למדעי המחשב (0368)  
מכבני נתונים (2158)  
(גרסה ארוכה)

מרצה: יוסף עזר, שירן ציצ'יק  
מתרגל: דן נתן/ דורפמן  
תשפ"ג, סמסטר א' (2022-2023)

מסכם: רועי מעין



The Raymond and  
Beverly Sackler Faculty  
of Exact Sciences  
Tel Aviv University



## פרק 1 – מבוא

3 .....	סיבוכיות.....
5 .....	רקורסיה.....
10 .....	מערך ורשימה מקושרת.....
14 .....	Amortization.....

## פרק 2 – מבני נתונים

19 .....	עצי חיפוש ביןאריים - BST.....
23 .....	עצי AVL.....
26 .....	עצי דרגות.....
31 .....	עצי B.....
36 .....	טבלאות Hash.....

## פרק 3 – אלגוריתמים לפתור בעיות

45 .....	ערימות ביןאריות.....
49 .....	ערימות ביןמיות ופיבונאצ'י.....
53 .....	בעיית המינון.....
59 .....	בעיית הבחירה.....
64 .....	Union Find.....



## 1 – מבוא

### סיבוכיות

אסימפטוטיקה

:The O Notation

O גודלה – תהאינה  $t(n) = O(g(n))$  שתי פונקציות מהטבעיות לטבעיות. נאמר כי  $t(n) = O(g(n))$  אם קיימים קבועים חיוביים  $c_0, n_0$ , כך שלכל  $n > n_0$  מתקיים:  $t(n) \leq c_0 \cdot g(n)$ .

בסיס הלוגריתם – בכל מצב שבו לא ניתן את בסיס הלוגריתם הכוונה תהיה לבסיס 2. בנוסף, מעבר בין בסיסים (שגדלים מ-1) גורר בסה"כ הכפלה בקבוע, שמלילא לא משפיעה על סדר הגודל. לכן נהוג לרשום פשוט  $O(\log n)$  בלי לציין את בסיס הלוגריתם.

חסמים נוספים:

- O אומגה גודלה –  $t(n) = \Omega(g(n))$  אם קיימים קבועים חיוביים  $c, n_0$ , כך שלכל  $n > n_0$  מתקיים:  $t(n) \geq c \cdot g(n)$ .
- O או קטנה –  $t(n) = o(g(n))$  אם מתקיים  $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = 0$ . הגדירה אחרת כללית יותר, שלא תליה בקיום הגבול: אם לכל קבוע  $0 < c$  קיים קבוע  $n_0$  כך שלכל  $n > n_0$  מתקיים:  $t(n) \leq c \cdot g(n)$ .

שימוש להבדל בין O גודלה ל-o קטנה: בהגדירה באופן דרמטי שאו השווון יתקיים לכל קבוע חיובי  $c$ . ניתן להראות שאם הגבול מההגדרה הראשונה קיים, אז ההגדרות שקולות. לחובנו יותר לעובד עם ההגדרה הראשונה.

- ω אומגה קטנה –  $t(n) = \omega(g(n))$  אם מתקיים  $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \infty$ . הגדירה אחרת כללית יותר, שלא תליה בקיום הגבול: אם לכל קבוע  $0 > c$  קיים קבוע  $n_0$  כך שלכל  $n > n_0$  מתקיים:  $t(n) \geq c \cdot g(n)$ .
- Θ תטא גודלה –  $t(n) = \Theta(g(n))$  אם קיימים קבועים חיוביים  $c_1, c_2, c_0, n_0$  כך שלכל  $n \geq n_0$  מתקיים:  $c_1 g(n) \leq t(n) \leq c_2 g(n)$ . נשים לב כי  $t(n) = O(g(n)) \Leftrightarrow t(n) = \Theta(g(n)) \Leftrightarrow t(n) = \Omega(g(n))$ .

סיכום סימונים:

סימן	משמעות	סוג החסם	ביטוי
$t$ <b>קטנה מאוד אסימפטוטית מ-g</b>	$t(n) = o(g(n))$ $g(n) = \omega(t(n))$	$g$ עליון לא הדוק	$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = 0$
$t$ <b>קטנה אסימפטוטית מ-g</b>	$t(n) = O(g(n))$ $g(n) = \Omega(t(n))$	$g$ עליון	קיימים קבועים $n_0, c$ , כך שלכל $n > n_0$ מתקיים: $t(n) \leq c \cdot g(n)$ .
<b>אותו קצב גידול אסימפטוטי</b>	$t(n) = \Theta(g(n))$	הdock (עליון ותחתון)	קיימים קבועים $c_1, c_2, c_0, n_0$ כך שלכל $n \geq n_0$ מתקיים: $c_1 g(n) \leq t(n) \leq c_2 g(n)$ $\Leftrightarrow t(n) = \Theta(g(n))$ $t(n) = \Omega(g(n)) \wedge t(n) = O(g(n))$
$t$ <b>גדולה אסימפטוטית מ-g</b>	$t(n) = \Omega(g(n))$ $g(n) = O(t(n))$	$g$ תחתון	קיימים קבועים $n_0, c$ , כך שלכל $n > n_0$ מתקיים: $t(n) \geq c \cdot g(n)$ .
$t$ <b>גדולה מאוד אסימפטוטית מ-g</b>	$t(n) = \omega(g(n))$ $g(n) = o(t(n))$	$g$ תחתון לא הדוק	$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \infty$

הערה: הסימון **O** הוא הנפוץ והשימושי ביותר מבין הסימונים האסימפטוטיים. משתמשים בו פעמים רבות כחסם הדוק, על אף שפורמלית הוא חסם עליון בלבד. למשל, בד"כ נאמר שחייבש בינהרי רץ במקורה הגורע בסיבוכיות זטן ( $\log n$ ), כאשר ברור שהכוונה היא שזהו חסם הדוק. לא תמיד טורחים לבטא זאת במפורש ע"י השימוש בסימון  $\Theta$ . השימוש בסימון  $\Theta$  יופיע הרבה כאשר יהיה חשוב לנו להציג את הדיקותו של החסם.

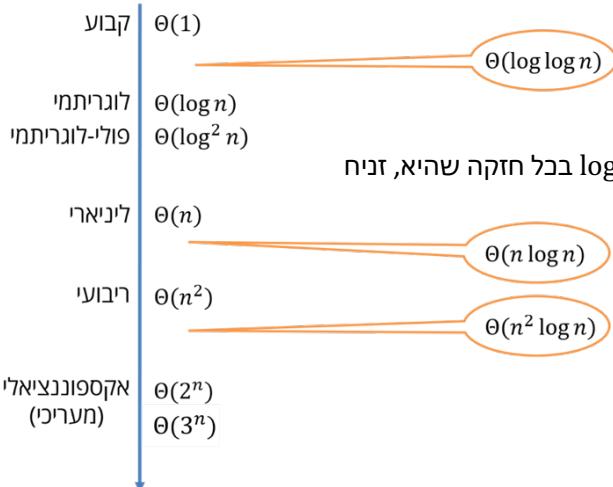
הערות חשובות על השימוש בסימונים:

1. סימונים אסימפטוטיים בביטויים ארכיטמטיים – נוכל לפרש אותם באמצעות אי שוויון, להציב במקומם קבוע c ומשם:

$$O(n) + O(n \log n) \leq c_1 n + c_2 n \log n = O(n \log n), c = c_1 + c_2$$

2. חישור וחילוק ביטויים לא מוגדרים – להיזהר משימוש בחישור ובחילוק עם סימונים אסימפטוטיים. התוצאה לא תמיד מוגדרת היטב. באופן יותר ספציפי,بعد שהבל הבא תקין:  $O(f(n) + g(n)) = O(f(n) + O(g(n)))$ , אותו כלל עם חישור במקום חיבור איןנו תקין.

3. חסמים של שניים או יותר פרמטרים – לעיתים נתנו סיבוכיות של אלגוריתמים כתלות ביוטר מפרמטר אחד. במקרה כי סיבוכיות זמן הריצה היא  $(n+m)^k$  – ומשמעותו כי הסיבוכיות תליהlingenarity בשני הפרמטרים הללו. אם "נקבע" את  $n$  ונתיחס אליו בלבד, הזמן הריצה של האלגוריתםlingenarity יתפרק לזמן  $m$ .

**Toolbox**כללים שימושיים שכדאי לזכור:

1.  $a^k = n^k$  לכל שני קבועים  $0 < k < a$ . כלומר, העלאת  $\log$  בכל חזקה שהוא, זניח אסימפטוטית ביחס לכל חזקה של  $n$ .

$$\lim_{n \rightarrow \infty} n^a - \log^k n = \infty$$

2.  $a^k = n^k$  לכל שני קבועים  $1 < a < k$ . קצב גידול פולינומי זניח לעומת גידול אקספוננציאלי.

3.  $a^n = b^n$  לכל שני קבועים  $1 < a < b$ .

$$\text{טור חשבוני: } \sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

4. סכום סדרה חשבונית:  $S_k = \frac{k(a_1 \cdot a_k)}{2}$

$$\text{טור הנדסי: } (x^n) = \Theta(x^n) = \Theta\left(\frac{x^{n+1}-1}{x-1}\right) \text{ עבור } 1 < x.$$

$$\text{טור הנדסי אינסופי יורד: } \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = \Theta(1) \text{ עבור } |x| < 1.$$

$$\text{טור הרמוני: } (\ln(n)) \leq H_n \leq \ln(n) + 1 \text{ כי } H_n = \sum_{i=1}^n \frac{1}{i} = \Theta(\log n).$$

$$\text{פולינום מדרגה } d: \sum_{i=0}^d a_i n^i = \Theta(n^d) \text{ עבור קבוע } d \text{ ועבור } a_d \neq 0.$$

$$\text{lienarיות הסימונים האסימפטוטיים: } \sum_{i=0}^n O(g(i)) = O(\sum_{i=0}^n g(i)).$$

נקודות נוספות:

$$a^{-k} = \frac{1}{a^k}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_c a^n = n \log_c a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(\frac{1}{a}) = -\log_b a$$

$$a^{\log_b c} = c^{\log_b a}$$

$$a = b^{\log_b a}$$



## ركورסיה

### שיטת האיטרציות

הרענון בשיטת האיטרציות (the Iteration Method) הוא שפטורים את נוסחת הנסיגה ע"י בך שמציבים אותה בתוך עצמה, מספר פעמים, עד שמת\_kvבל ביטוי סגור שאת הסיבוכיות שלו אנו יודעים לחשב.

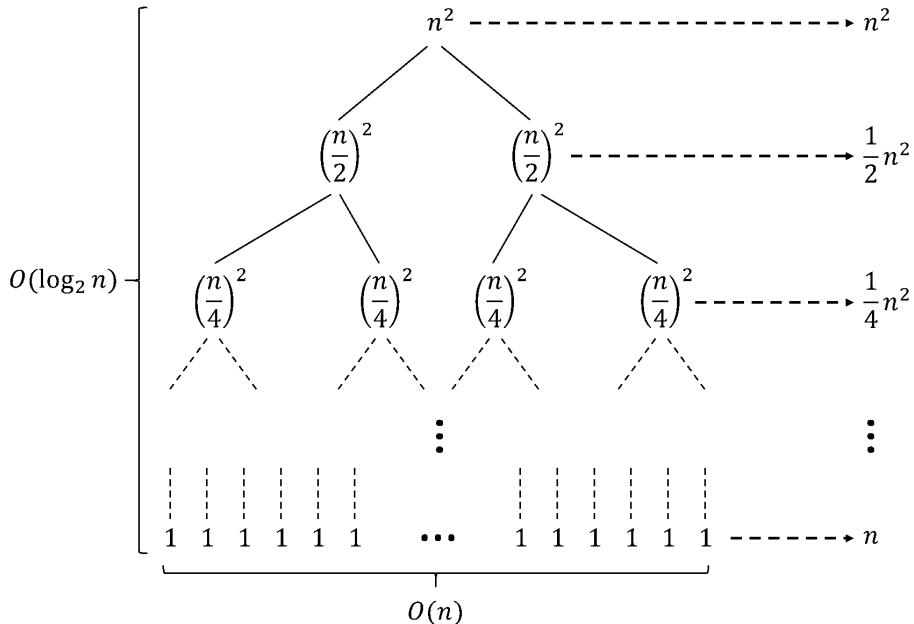
דוגמה	נוסחת בסיגה	פתרון הנוסחה
חיפוש ביןארי	<p>בכל קראיה רקורסיבית של החיפוש הבינארי, או שמוסאים את האיבר שchipshnu, או שימושים לחפש באחד מחצאי הסדרה – באמצעות קראיה רקורסיבית לפונקציה עם סדרה בגודל לכל היותר <math>\left\lfloor \frac{n}{2} \right\rfloor</math>. לכן ניתן לבטא את זמן הריצה של האלגוריתם במקרה הגרוע ע"י:</p> $t(n) = t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$ <p>באשר <math>c</math> מייצג כמהות קבועה של עבודה, לצורך השוואה של המפתח שמחפשים וכו'.</p>	<p>נציב את נוסחת הנסיגה עבורה <math>t\left(\left\lfloor \frac{n}{2} \right\rfloor\right)</math> בתוך הנוסחה עבורה (<math>\left\lfloor \frac{n}{2} \right\rfloor</math>) <math>t</math> וכך להלאה:</p> $\begin{aligned} t(n) &= t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c = t\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 2c = t\left(\left\lfloor \frac{n}{8} \right\rfloor\right) + 3c = \dots \\ &= t\left(\left\lfloor \frac{n}{2^i} \right\rfloor\right) + ic \end{aligned}$ <p>לכן הגענו לאחר מכן <b>צבות של הנוסחה</b>. נבדוק מתי נפסיק להציב, כאשר <b>גען למקורה הבסיס שהוא 0</b>. <math>t</math>. במקרה זה נקבל <math>\left\lfloor \frac{n}{2^i} \right\rfloor &lt; 1 \Leftrightarrow i &lt; \log n</math> ולבסוף <math>i = \Theta(\log n) + 1 = \Theta(\log n) + 1</math>.</p> <p>זה"כ נקבל <math>t(n) = t(0) + c(\log n + 1) = \Theta(\log n)</math></p>
מציאת מקסימום	<p>בכל קראיה רקורסיבית, משווים בין האיבר הנוכחי <math>\text{I}</math> והוכחי לבין המקסימום מבין שאר איברי הסדרה. לצורך בך יש קראיה רקורסיבית לפונקציה עם סדרה באורך <math>1 - a</math> (כאשר <math>a</math> זה הוא אורך הסדרה בקריאה הנוכחית). לכן ניתן לתאר את זמן הריצה ע"י הנוסחה:</p> $t(n) = t(n-1) + c$	<p>בכל קראיה רקורסיבית, משווים בין האיבר הנוכחי <math>\text{I}</math> והוכחי לבין המקסימום מבין שאר איברי הסדרה. לצורך בך יש קראיה רקורסיבית לפונקציה עם סדרה באורך <math>1 - a</math> (כאשר <math>a</math> זה הוא אורך הסדרה בקריאה הנוכחית). לכן ניתן לתאר את זמן הריצה ע"י הנוסחה:</p> $t(n) = t(n-1) + c$
מיון-מיוזג Merge-Sort	<p>בכל קראיה חוצים את הסדרה לשניים וקוראים רקורסיבית לפונקציה על שני החצאים, ואז מבצעים מיוזג של החצאים (כאשר מיוזג עבורה שני חצאים בגודל <math>\frac{n}{2}</math> רצה בסיבוכיות <math>O(n)</math>). לעומת זאת, מספר הצעדים שմבצעת פונקציית המיוזג חסום ע"י <math>c</math> כאשר <math>c</math> קבוע כלשהו, לכן נוכל לתאר את זמן הריצה ע"י הנוסחת הנסיגה:</p> $t(n) = 2t\left(\frac{n}{2}\right) + cn$	<p>הקריאות הרקורסיביות נמשכות עד <math>r &lt; l</math>, כלומר הסדרה באורך 2 לפחות. מכאן שבסדרה באורך <math>n</math> מתקיימת <b>מקרה הבסיס הוא 1</b>. מכיוון שבכל הקראיות הרקורסיביות מתבצעות על סדרות באורך שהוא חצי מהאורך הנוכחי, אחרי <math>i</math> הצעות של נוסחת הנסיגה, גודל כל סדרה יהיה <math>\frac{n}{2^i}</math>, וכן <b>בנשואן</b> <math>(n) = \Theta(\log n) = i</math> גען למקורה הבסיס (1). נפתח את הנוסחה:</p> $\begin{aligned} t(n) &= 2t\left(\frac{n}{2}\right) + cn = 2\left(2t\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\ &= 4t\left(\frac{n}{4}\right) + 2cn = \dots = 2^i t\left(\frac{n}{2^i}\right) + icn \\ &= n \cdot t(1) + cn\log n = \Theta(n\log n) \end{aligned}$
מציאת מקסימום גרסה נוספת (נוספה)	<p>בכל קראיה חוצים את המקטע מ-1 ל-<math>z</math> לשניים, ומחשבים בעזרת שתי קראיות רקורסיביות את המקסימום של כל חלק. עבור מקטע עם <math>z</math> איברים כל חצי בגודל בערך <math>\frac{n}{2}</math> וכן ניתן לתאר את זמן הריצה ע"י נוסחת הנסיגה:</p> $t(n) = 2t\left(\frac{n}{2}\right) + c$	<p>הקריאות נמשכות עד <math>r &lt; l</math>. מקרה הבסיס הוא 1. אחרי <math>i</math> הצעות של נוסחת הנסיגה, גודל כל סדרה יהיה <math>\frac{n}{2^i}</math> וכן <b>בנשואן</b> <math>(n) = \Theta(\log n)</math> גען למקורה הבסיס (1). נפתח את הנוסחת הנסיגה (נשים לב שבמקרה זה לא הגענו לנוסחה סגורה מיד כמו במקרים הקודמים, שמרנו מוחברים מקרים קודמות, על מנת לראות בעיינם שמדובר בסדרה הנדסית, ואז מגיעים לסקום שלה):</p> $\begin{aligned} t(n) &= 2t\left(\frac{n}{2}\right) + c = 2\left(2t\left(\frac{n}{4}\right) + c\right) + c = 4t\left(\frac{n}{4}\right) + 2c + c \\ &= 8t\left(\frac{n}{8}\right) + 4c + 2c + c \\ &= c(1 + 2 + \dots + 2^{i-1}) + 2^i t\left(\frac{n}{2^i}\right) \\ &= c(1 + 2 + \dots + 2^{\log n - 1}) + nt(1) \\ &= c \cdot \sum_{i=0}^{\log n - 1} 2^i + nt(1) \\ &= c \cdot \frac{2^{\log n} - 1}{2 - 1} + nt(1) \\ &= c(n - 1) + nt(1) = \Theta(n) \end{aligned}$



## שיטת עץ הרקורסיבית

בשיטה זו נזכיר את עץ הקריאה הרקורסיביות, ונסכום את כמות העבודה על פניו כל הצמתים. נמחיש שיטה זו על ידי מספר דוגמאות.

1.  $t(n) = 2t\left(\frac{n}{2}\right) + n^2$ . נזכיר את עץ הקריאה, כאשר בכל צומת זמן הריצה הוא הקלט בריבוע, ומtbodyות שתי קריאות נוספות מעבר חצי מהקלט.



נסכום כל שורה בעץ, ואז נסכם על פניו כל השורות:

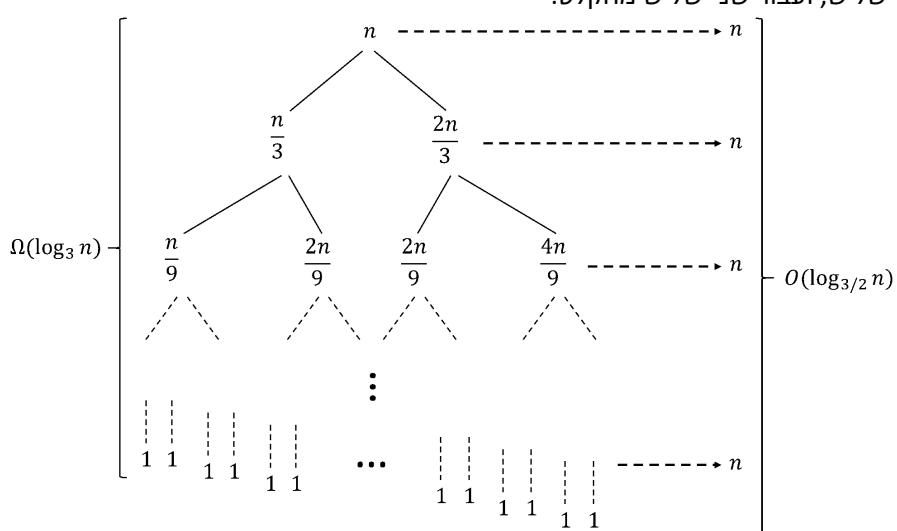
$$t(n) = n^2 + \frac{1}{2}n^2 + \frac{1}{4}n^2 + \dots + \frac{1}{2^l}n^2 + \dots + \frac{1}{n}n^2 = n^2 \left( \sum_{i=0}^{\log_2 n} \frac{1}{2^i} \right)$$

סכום הצמתים בשורה ה-*i*

כדי להקל על החישוב, נחסם את הביטוי בסוגרים על ידי סדרה הנדסית אינסופית:

$$t(n) \leq n^2 \left( \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \right) = n^2 \left( \frac{1}{1 - \frac{1}{2}} \right) = 2n^2 = O(n^2)$$

קיבלנו חסם עליון, אך נוכל להוכיח שהוא הדוק (הוא גם חסם תחתיו):  $t(n) = 2t\left(\frac{n}{2}\right) + n^2 = \Omega(n^2)$ . 2.  $t(n) = t\left(\frac{n}{3}\right) + t\left(\frac{2n}{3}\right) + n^2$ . נזכיר את עץ הקריאה, אשר בכל צומת זמן הריצה הוא כתלות בקלט עצמו, ומtbodyות שתי קריאות נוספות מעבר שלישי מהקלט.





ראשית נשים לב כי גובה העץ אינו אחיד, הعالים לא באוטו עמוק. הגובה של העץ חסום מלמעלה ע"י האורך של המסלול הימני ביותר מהשורש, שבו הצומת בעומק  $i$  עם זמן ריצה  $a^{\frac{2}{3}}$  (ולכן אורך מסלול זה הוא  $a^{\frac{2}{3} \log_3 O}$ ). כמו כן, גובה העץ חסום מלמטה ע"י אורך המסלול השמאלי ביותר מהשורש, שבו הצומת בעומק  $i$  עם זמן ריצה  $a^{\frac{1}{3}}$  (ולכן אורך מסלול זה  $(a^{\frac{1}{3}} \log_3 O)$ )  
**(איןטואיציה:** בנוסחת הנסיגה, כאשר אנו מוצאים עבודה על שליש מהרשימה אנו חותכים אותה מהר מאד, ועל קלט כמו 9, תוך שתי קריאות נגוע לפחות באורך 1 שם נעצור. לעומת זאת, כאשר אנו חותכים את הרשימה לאט יותר, ולקחים שני-שליש ממנה, ניקח יותר זמן להגעה לעלה בעץ, עד לפחות באורך 1).

שני החסמים האלו נבדלים בקבוע (ראינו כבר כי לוג בכל בסיס זה בפועל לוג 2 כי אפשר למצאו את הקבוע ס' שיטן לנו את החסם ההדוק של לוג 2) וכן קיבלנו למעשה שגובה העץ הוא  $(n \log \theta)$ . בכל שורה מתבצעת צ'זבודה וכן סה"כ נקבל  $t(n) = \Theta(n \log n)$ .

הכללה של דוגמא זו:  
 $n + t(\alpha n) + t(\beta n) = t(n)$ , כאשר  $1 < \alpha, \beta < 1, 0 < \beta - \alpha$ . חישוב דומה במקרה זה ייתן לנו אוז הסיבוכיות  $\Theta(n \log n)$ .

מה משנתה כאשר  $1 < \beta + \alpha$ ? במקרה זה ניתן להראות שמתתקבל  $t(n) = \Theta(n)$ . הוכחה זאת רמזו: השתמשו בנוסחה לסכום סדרה הנדסית:

$$\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}$$

#### משפט המאסטר

משפט המאסטר יכול לעזור לנו לפתרון נוסחאות נסיגה מהצורה הבאה:  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$  (1)  $b > a$  כי ברקורסיה אנחנו דורשים תמיד שהבעיות שאנו פותרים יהיו ממש קטנות יותר מהבעיה המקורית). לעומת זאת, מדובר בנוסחה המתארת אלגוריתם רקורסיבי, שמחילק את הבעיה שלו בגודל  $a$ , ל- $a$  תת-בעיות בגודל  $\frac{n}{b}$  כל אחת. בנוסף, האלגוריתם דורש  $f(n)$  בשבייל לבצע את החלוקה הזאת, וכך לאחד את הפתרונות של תת-בעיות האלו, לפתרון של הבעיה המקורית.

המשפט אומר כי ניתן לחסום את (1) אסימפטוטית על פי אחד המקרים הבאים:

מקרה	תנאי	תוצאה	דוגמה
1	$0 > \varepsilon$	$f(n) = O(n^{\log_b a - \varepsilon})$	$t(n) = 3t\left(\frac{n}{3}\right) + c$ $f(n) = c = O(n^{1-\varepsilon}) \Rightarrow t(n) = \Theta(n)$
2		$f(n) = \Theta(n^{\log_b a})$	$t(n) = t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$ $f(n) = c = \Theta(n^0) \Rightarrow t(n) = \Theta(n \log n)$
3	$\varepsilon > 0$ $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), c < 1$ (from a large enough $n$ )	$T(n) = \Theta(f(n))$ הגורם הדומיננטי – הזמן שדרוש בשבייל לפרק את הבעיה לתחתי-בעיות ולצער את תת-פתרונות $t(n) = 3t\left(\frac{n}{2}\right) + n^2$ $f(n) = n^2 = \Omega(n^{\log_2 3 + \varepsilon})$ $3\left(\frac{n}{2}\right)^2 \leq \frac{3}{4}n^2, \text{for } c = \frac{3}{4} < 1$ $\Rightarrow t(n) = \Theta(n^2)$	

- המשפט עובד גם כאשר במקומות  $\frac{n}{b}$  נוסחת הנסיגה מכילה  $\left\lfloor \frac{n}{b} \right\rfloor, \left\lceil \frac{n}{b} \right\rceil$ .
- לעיתים לא מתקיים אף מקרה של שיטת המאסטר, למשל בנוסחת הנסיגה הבאה:  $t(n) = 2t\left(\frac{n}{2}\right) + n \log n$

**הוכחה באינדוקציה**

שיטת זו טוביה למקרים שבהם ויחסית קל לנחש את הפטון לנוסחת הנסיגה, וחוצים להוכיח באופן פורמלי שהזו אכן הפתרון. לעיתים משתמש באחת השיטות הקודמות לכך לנחש מה אמרו להיות הפתרון, ואז נכיח באינדוקציה שהניחוש שלו נכון.

הערה: חשוב לטעון טענה אינדוקטיבית עם הקבועים הנכונים, ולא להחליף את הקבוע תוך כדי ההוכחה.

**טריקים:**

1. החלפת משתנים – נסמן  $m = \log n$ ,  $s(m) = \frac{T(n)}{n}$ .

2. פעולות אלגבריות – נסמן  $t(2^m) = \frac{T(n)}{n}$  וubah'  $s$  ווחילוף משתנים כמו קודם.

**הוכחה מלאה באינדוקציה:**

דוגמה להוכחה עבור נוסחת הנסיגה הבאה:

$$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n = O(n), T(1) = 1$$

הפתרון הם:

- ניסוח טענה מתאימה.
- ניחוש הקבועים – ראשית לצעד האינדוקציה ואז לבסיס.
- ניסוח פתרון מלא: בסיס וצעד (עם הקבועים שמצאנו).

**פתרון באינדוקציה**

נארח מוכחים סדר גודל אסימפטוטי של נוסחת נסיגה באינדוקציה, חשוב מאוד להבין מה הטענה האינדוקטיבית. הטענה האינדוקטיבית היא לא  $T(n) = O(n)$ . הטענה היא: קיימים קבועים  $C, n_0$  (שייקבעו בהמשך) עבורם מתקיים שלכל  $n \geq n_0$ ,

**ניחס הקבועים**

קיים נריצה "לנחש" את הקבועים. חלק זה נועד רק להדרכה, ואיןו בהכרח חלק מהפתרון. הדרך בה "నנחש" את הקבועים היא להסתכל על צעד האינדוקציה ובסיס האינדוקציה, ולהסביר מהם אילוצים על  $C, n_0$ . קיים נסתכל על צעד האינדוקציה, כיוון שבבסיס האינדוקציה אפשר לקבל אילוץ רק עבור  $C$ .

**ניחס הקבועים המתאימים לצעד האינדוקציה:**

נניח את נכונות הטענה עבור ערך  $m$  שקיימים  $1 \leq m \leq n - 1$ , וכן  $n \geq n_0$ .

$$\begin{aligned} T(n) &= \\ 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n &\leq 3 \cdot \left(C \cdot \left\lfloor \frac{n}{4} \right\rfloor\right) + n \leq 3 \cdot \left(C \cdot \frac{n}{4}\right) + n = \\ \frac{3}{4} \cdot C \cdot n + n &= n \cdot \left(\frac{3}{4}C + 1\right) \end{aligned}$$

נרצה למצוא אילוצים על  $n_0$  ו- $C$  כך שקיימים  $n \geq n_0$ ,  $\frac{3}{4}C + 1 \leq C$ .

$n_0 \geq n$ . נחלק ב- $n$  ונקבל את האילוץ

$$C \geq \frac{3}{4}C + 1 \Leftrightarrow \frac{1}{4}C \geq 1 \Leftrightarrow C \geq 4$$

**ניחוש הקבועים המתאימים לבסיס האינדוקציה:**

נרצה שיתקיים  $1 \leq C \cdot T(1) = 1$ . מכאן נקבל את האילוץ  $C \geq 1$ .  
 בסך הכל קיבלנו על  $C \geq \max\{1, 4\} = 4$  את האילוץ  $C \geq \max\{1, 4\}$  אך ננחשת את הקבועים  
 $C = 4, n_0 = 1$ .

**הערה**

את כל מה שעשינו כאן לא חייבים לעשות לפני ההוכחה באינדוקציה.  
 אפשר להגיע לailozim אלו גם "תור כדי", כלומר לפניות ישר להוכחה  
 ובסיומה לומר מה ailozim הנדרשים לנוכנות ההוכחה.

**הפתרון**

אם ניחשנו את הקבועים  $C = 4, n_0 = 1$ , ובעת נכח באינדוקציה את הטענה:  
 $T(n) \leq C \cdot n$ ,  $n \geq n_0 = 1$ ,  $C = 4$  מתחווים שלכל  $n \geq n_0$ , ונכיח  $T(n) \leq C \cdot n$ .

**מקרה הבסיס:**

$$T(1) = 1 \leq 4 = C$$

**צעד האינדוקציה:**

נניח את נכונות הטענה עבור ערך  $m$  שמקיימים  $1 \leq m \leq n - 1$ , ונכיח  $T(m) \leq 4m$ .

$$\begin{aligned} T(n) &= \\ 3T(\lfloor \frac{n}{4} \rfloor) + n &\leq 3 \cdot (C \cdot \lfloor \frac{n}{4} \rfloor) + n \leq 3 \cdot (C \cdot \frac{n}{4}) + n = \\ &n \cdot (\frac{3}{4} \cdot 4 + 1) = 4n \end{aligned}$$

כאשר באו השווין הראשון השתמשנו בהנחה האינדוקציה.  
 הוכחנו את הטענה, ונקבל לפי הגדרה  $T(n) = O(n)$ .

## מערך ורשימה מקוושרת

### ADT List

**רשימה היא אוסף לינארי של נתונים בזיכרון.** כאשר בידינו רשימה, נרצה לבצע פעולות כלשהן על הרשימה. בהקשר לכך אנו מבחינים בין המושג המופשט ADT שմדבר על ההתנהגות של סוגי הנתונים, לבין Data Structure שהוא המימוש הקונקרטי של טיפוס הנתונים. **את ה-ADT רשימה, נוכל למשם בשתי דרכים: באמצעות מערך, או באמצעות רשימה מקוושת.** נזכיר:

Time complexity ( $n$ is the list size)	Array	Linked List
<b>Retrieve(<math>L, i</math>) –</b> return the $i^{\text{th}}$ item of $L$	$O(1)$	$O(i + 1)$
<b>Search(<math>L, b</math>) –</b> return the position of $b$ in $L$ , or $-1$	$O(n)$	$O(n)$
<b>Insert-First(<math>L, b</math>) –</b> insert $b$ as the $0^{\text{th}}$ element	$O(n)$	$O(1)$

בעת נבחן מימושים שונים ל-ADT רשימה, ונסתכל בפרט על הפעולות של **גישה** למקום ה- $i$ , **הכנסה** ו**מחיקה** בהתחלה ובסוף, הכנסה ומחיקה באותה מקום כללי ("באמצע"), ושרשור שתי רשימות.

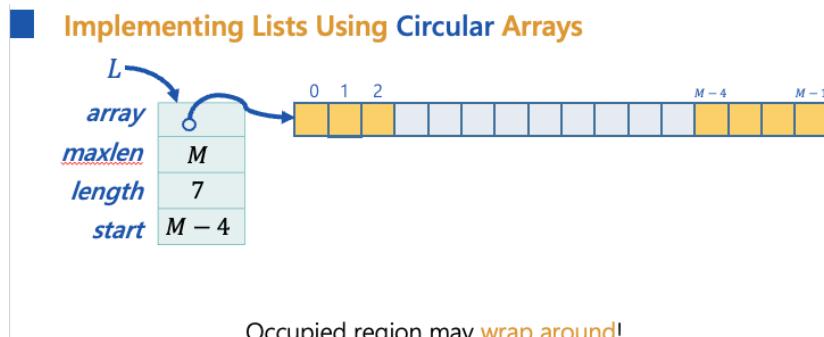
### מימוש באמצעות מערך

#### מימוש – מערך:

- **הכנסה/מחיקה בהתחלה – ( $O(n)$ )**, יש להזיז את כל האיברים המערך בהתאם.
- **הכנסה/מחיקה כללית – ( $i + 1 - n$ ) $O$ .** לאחר מחיקת האיבר במקום ה- $i$  עלינו להזיז את כל האיברים **שנמצאים מימין** באותו מקום אחד שמאליה, כך שלא יוותרו "חורים" במערך. באופן דומה, כדי לבצע הכנסה למקום ה- $i$  עלינו להזיז את כל האיברים **שמיימין** לאותה מקום אחד ימינה כדי לפנות מקום לאיבר החדש. כלומר **אנו מזיזים  $i - n$  איברים**, لكن עלות הביצוע של פעולות אלו היא  $(i + 1 - n)O$ .

#### מימוש – מערך מעגלי:

נבחן וריאציה אחרת, מערכים מעגליים. נסיף שדה שנקרא **start** ומסמן את אינדקס ההתחלתה של המערך (רצף האיברים בפועל). בchiposh איבר אנחנו לבצע  $mod L$ .**maxlen** בזווית שאייברי המערך יכולים לשבת בקצוות האזור בזיכרון, להתחיל בסוף ולהסתדרים בהתחלה.



Occupied region may **wrap around!**

- **גישה – ( $1$ ) $O$**  (אם נממן מתבצע באן חישוב יותר מורכב מוקדם, אך עדין נשארת אותה הסיבוכיות). נשים לב שאנו זוקקים לבצע  $mod$  בזווית שאמם נבקש לצורך העניין  $(4)O$  איזה הגיע לאינדקס  $M$  שחוורג מגבול המערך, ואם נבצע את החישוב של  $0 = M \bmod M$  נגיע לאינדקס 0 התקין.
- **הכנסה/מחיקה כללית/בסיסי** – בעת רצות- $(1)O$ .
- **הכנסה/מחיקה בהתחלתה/בסיסי** – אפשר להחליט מאי זהה בזווית לזמן עצמאית או באיזה בזווית להזיז איברים כדי ליצור מקום להכנסת האיבר החדש, לבדוק אם היא יש פחת איברים (מיימין או ממשאל) ולכן נקבל  $(i + 1 - n, n + \min\{i + 1, n\})O$ .
- אם האינדקס מקיים  $\frac{n}{2} < i$  אז יש פחות איברים ממשאל, ואם  $\frac{n}{2} > i$  אז יש פחות איברים מיימין.

**סיבום – סיבוכיות מימוש מערך/מערך מעגלי:**

1. מעתה בשנדבר על **מימוש של ADT רשימה באמצעות מערכים**, בעצם **דבר על מערכים מעגלים**, אין שום סיבה לא להשתמש בהם. הם יותר טובים מאשר המימוש של מערכים רגילים.
2. **במערכות מעגליים יש יותר חופש לבחור באיזה ביןון להזיז/לబנ尼斯 איברים חדשים, מימין או משמאלי** (בניגוד למערך רגיל שਮוגבל להכנסה והזזה רק ימינה). לכן בפועל הכנסה/מחיקה ובפועל שרשור נבחר את המספר המינימלי של איברים **עלינו להעתיק**.
3. מערכים באופן כללי אפשריים **זמן גישה של  $O(1)$** .
4. **עלינו לדעת את M מראש, להגדיר מראש מהו גודל המערך שאנו רוצים להקצות** (מגבלה זו אינה קיימת כאשר נממש באמצעות **רשימת מקושתת**).

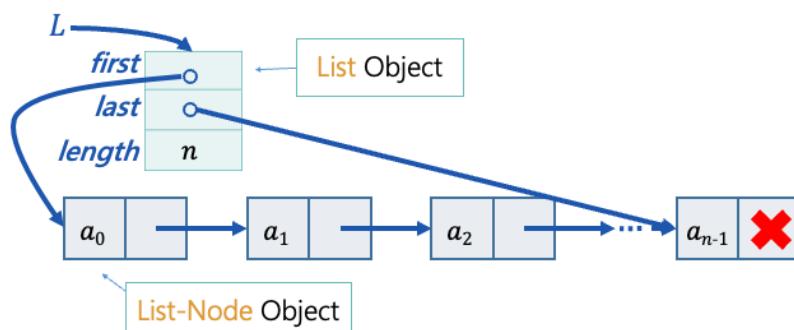
	Arrays	Circular Arrays
Insert/Delete Last	$O(1)$	$O(1)$
Insert/Delete First	$O(n + 1)$	$O(1)$
Insert/Delete (index $i$ )	$O(n - i + 1)$	$O(\min\{i + 1, n - i + 1\})$
Retrieve (index $i$ )	$O(1)$	$O(1)$
Concat (Sizes: $n_1, n_2$ )	$O(n_2 + 1)$	$O(\min\{n_1, n_2\} + 1)$

**מימוש באמצעות רשימה מקווארת**

**מימוש – רשימה מקווארת חד ביוונית:**

### Implementing Lists Using Singly Linked Lists

- Lists of unbounded length
- Support some **additional operations efficiently**



- **גישה –  $(1 + i)O$**  כיון שכבר לא ניתן לחשב את הכתובת ולגשת אליה ישירות, חיברים לעבור על האיברים ידנית.
- **הכנסה/מחיקה בהתחלה –  $(1)O$** .
- **מחיקה בלילית –  $(i + 1)O$** . גם כאן נממש פעולה עוז **Delete-After-Delete**.
- **מחיקה בסוף – כרגע  $(1 + n)O$  אך נפתר זהה בקרוב.**
- **שרות –  $(1)O$** , משחק עם מצביעים.

**מימוש – רשימה מקווארת דו-ביוונית (ומעגלית):**

- **גישה –  $\{1 + i, n\}O$** . שיפור קל כיון שהרשימה מעגלית (נוכל לבחור באיזה ביןון לבצע את הפעולה).
- **מחיקה בסוף –  $(1)O$** . בשונה מרשימה חד-ביוונית, יש לנו באן בכל צומת בנוסף לשדה **next** גם שדה **prev**.
- **הכנסה/מחיקה בלילית –  $\{1 + i, n\}O$** . שיפור קל כיון שהרשימה מעגלית.

### סיבום – סיבוכיות רשימה מקוורת דו-כיוונית ומעגלית/מערך מעגלי:

1. יתרון **למערך** (רכיף בזיכרון) – **גישה בזמן קבוע** על ידי חישוב כתובות.
2. יתרון **לשיטה מקוורת** – מבנה יותר דינמי, אין הגבלה על גודל הרשימה. בנוסף, אמנים יותר קשה להגיע לאיברים בתחום מבנה זהה, אבל ברגע שהגענו קל מאוד לבצע שינוי של מצביעים. לעומת **הכנסה/מחיקה בפועל** (בהתעלם מזמן החיפוש וההגעה לאיבר הרלוונטי) **מתבצעת ב-(1)**.

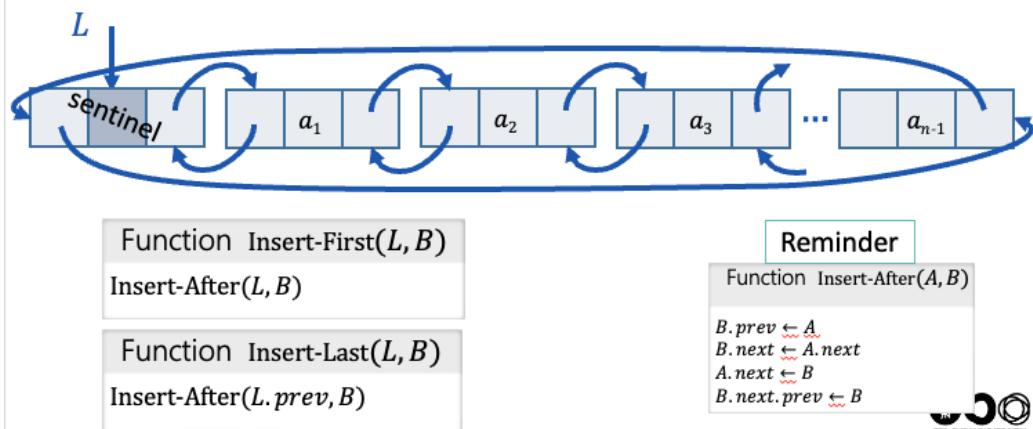
	<b>Circular Arrays</b>	<b>Doubly Linked Lists</b>
<b>Insert/Delete-First/Last</b>	$O(1)$	$O(1)$
<b>Insert/Delete (by index <math>i</math>)</b>	$O(\min\{i+1, n-i+1\})$	$O(\min\{i+1, n-i+1\})$
<b>Retrieve (by index <math>i</math>)</b>	$O(1)$	$O(\min\{i+1, n-i+1\})$
<b>Concat</b>	$O(\min\{n_1, n_2\} + 1)$	$O(1)$

:Sentinels (סנטינלים)

עבור טיפול במקרי קצה, הרעיון הוא להחזיק בתחילתה של הרשימה איבר דמה, שהוא לא אמיתי. מדובר **בצומת שלא מחזיק שום מידע לגבי איבר מהותי בראשימה**. אם יש לנו  $n$  איברים בראשימה, יהיו לנו  $1 + n$  צמתים.

בutex, נוכל להסתכל על Insert-First בטור (Insert-First), ובאופן דומה Insert-After (Sentinel). Insert-Last יהיה Insert-After(A) – איבר שנמצא לפני ה-Sentinel. ב-הכנסה/מחיקה בכל מקום אינה שונה מ-הכנסה/מחיקה בהתחלה או בסוף.

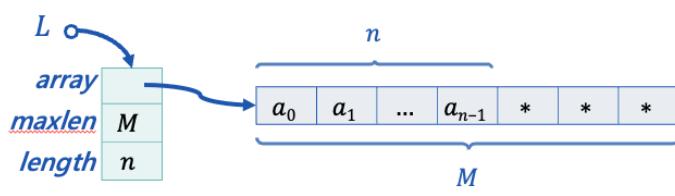
### Insert-First / Insert-Last



### מחסניות ותורים

שלושה מקרים פרטיים חשובים של ADT רשימה הם: **מחסנית (Stack)**, **טור (Queue)** ו**דו-טור (Double-ended Queue)**.

1. **מחסנית** – האיבר האחרון שהוכנס יהיה הראשון לצאת ממנו, FIFO. כיצד נמשך את כל הפעולות ב-(1)?
- a. **מערך** – אם נdag **להכניס ולחוץ איברים רק מסוף המערך**. אין יתרון למערך מעגלי על פני לא מעגלי.
- b. **רשימה דו-כיוונית** – אם נתיחס **לתחילה הרשימה בסוף המחסנית**.



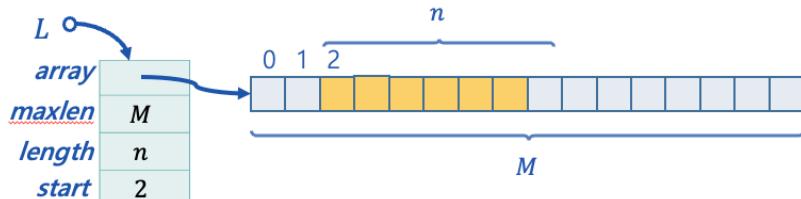
Push( $L, b$ ), Top( $L$ ), Pop( $L$ ) take  $O(1)$  time

All Stack operations in  $O(1)$  time



2. תור – האיבר הראשון שנכנס לטור הוא הראשון לצאת ממנו, FIFO. כיצד נממש את כל הפעולות ב-(1)?

- מערך מעגלי – כיוון שאות מהפעולות (הכניסה בסוף, הוצאה מההתחלה) תאלץ אותו לבצע הזהה של כל שאר איברי התור, **במערך מעגלי** יוכל לפתור זאת בלבד.
- רשימה חד-כיוונית – **נתיחה לואש הרשימה** כתחלת התור, ואם נחזיק מצביעים לתחילה הרשימה ולסופה בכלל לבצע את כל הפעולות בזמן קבוע. אם היינו מתייחסים לראש הרשימה בסוף התור (איפה מבניםים) ובסוף הרשימה כתחלת התור (איפה שמוציאים), כדי למחוק את האיבר האחרון היינו צריכים לעבור על כל איברי התור (בעיה זו הייתה נפתרת ברשימה דו-כיוונית).



Enqueue(L, b), Head(L), Dequeue(L) take  $O(1)$  time

All Queue operations in  $O(1)$  time

### ADT Vector

Function Get(V, i)

```
if Is-Garbage(i)
    return 0 (or other initial value)
else return V[i]
```

אם מפרידים בין הקצת זיכרון לבון האתחול שלו. כדי לאתחול מערך בגודל  $M$  יש צורך לבצע על כל התאים בו וככטבו אליהם ערך ברירת מחדל. لكن הפעולה הזאת בסיבוכיות זמן ריצה ( $O(n)$ ). בעת נגידר את ה-ADT וקטור, ונראה **שיטה מתוחכמת לאתחול מערך בסיבוכיות של  $(1)$  בלבד** (הוקטור יהיה מבנה נתונים נטונם שמנוהג באילו באמת אתחלנו כל תא בו).

אם רצים לבצע  $Get(i)$  ב- $(1)$ , בולומר לשולף את הקואורדינטה ה- $i$ - שבקטור. אם מדובר בקואורדינטה חוקית (מיישו ביצע Set קודם לכן) אז נחזיר את הערך המתאים, ואם מדובר בקואורדינטה ריקה, נרצה להחזיר  $0$  (או כל ערך דיפולטי אחר). הבעיה שאנו נתקלים בה היא כיצד לבדוק האם בתא שניגשנו אליו, יש ערך אמיתי או ערך ברירת מחדל? במקרים אחרות, **אם האינדקס  $i$  מכיל ערך זבל או ערך אמיתי?**

### ADT Graph

גרף מכוון הוא זוג סדור  $\langle V, E \rangle$  כאשר  $V$  היא קבוצה של צמתים,  $E$  קבוצה של קשתות. נסמן  $m = |V|$ ,  $n = |E|$  ונשים לב כי מתקיים  $n^2 \leq m$ . שימושים אפשריים ל-ADT גרף:

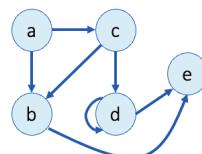
- מטריצת סמיביות/שכ니ות – נשמר מטריצה ריבועית בגודל  $n \times n$ , מדובר בקשת מ- $i$  ל- $j$  בgraf מכוון ולא להיפך.

#### Graph Representation 1: Adjacency Matrix

Let  $A$  be a boolean matrix of size  $n \times n$  so that:

$$A[i, j] = 1 \Leftrightarrow (i, j) \in E$$

Example:



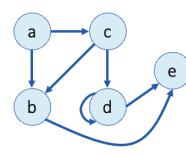
	a	b	c	d	e
a	0	1	1	0	0
b	0	0	0	0	1
c	0	1	0	1	0
d	0	0	0	1	1
e	0	0	0	0	0

- רשימת סמיביות/שכニות – לכל צומת נשמר רשימה של כל הצמתים שיוצאים אליו קשתות מהצומת הנוכחי.

#### Graph Representation 2: Adjacency List

Each node keeps a list of the nodes that are adjacent to it.

Example:



a	c	b
b		e
c	b	d
d	e	d
e		



## Amortization

### מבוא

גישה זו מתחילה את זמן הריצה הכלול של סדרת פעולות. לעיתים, ניתן לקבל בשיטה זו **חסם סיבוכיות הדוק יותר** מאשר שהינו מקבלים מניתוח פשוט של worst case. המוטיבציה היא שnitוח שלוקח בחשבון את **the-worst case** עבור כל פעולה ופעולה **הוא פסומו מידי**. יתכן שבחalker גדול של המקרים, זמן הביצוע של הפעולה קצר יותר מאשר זמן המקרה הגורע. לפיכך, עדיף להסתכל על סיבוכיות הזמן **בסדרה** כולה ולא ברמת הפעולה הבודדת.

- בולם, אנו הולכים לתת חסמים לסדרות של פעולות, באופן זה שאפשר להתייחס לכל פעולה בוודدت,ambil פעולה שרצה בפועלן בזמן הרבה יותר מאשר במקרה-worst case שלה. אז נקבל חסמים יותר הדוקים.
- למרות שפעולות בוודדות יכולות להיות מאוד יקרות, **בממוצע על פני סדרת הפעולות**, הูลות של כל פעולה היא לא כל כך גבוהה. אנו בעצם עבורים לדבר על ממוצע על פני סדרת פעולות.

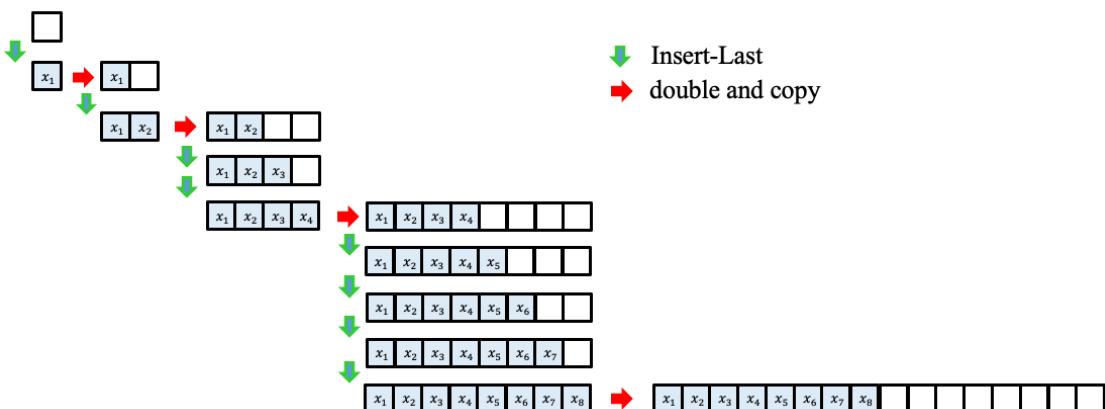
### הכפלת גודל מערך ושיטות ניתוח סיבוכיות

נסתכל על שימוש ל-ADT באמצעות מערך, כאשר גודלו המקסימלי הוא  $M$ . **נניח שהמערך שלו מלא** ואנו רצים להוסיף עוד איבר. אנחנו לא יכולים להרחיב את המערך, אלא **להקצת מקום אחר** בזיכרון מערך גדול יותר, להעתיק לשם האיברים הנוכחיים ושם יהיה מקום לאיברים נוספים בהמשך.

- **הגדלה ב-1:** נניח שאנו מגדילים את המערך ב-1, ואנו מתחילה עם מערך ריק. מה סיבוכיות הזמן של סדרת פעולות של אן הכנסות בכלל, כאשר הגודל עולה ב-1 בכל פעם?
  - $O(n^2) = \frac{n(n+1)}{2} = n + 2 + \dots + 1$ .
  - בכל שלב המערך מתמלא, ובין כל שלב מצריך הקצאה של זיכרון עבור מערך חדש והעתקה של האיברים מהמערך הישן לחישוב. אז לאחר הכנסה הראשונה נדרש להעתיק איבר אחד למערך החדש, ולאחר השניה נדרש להעתיק שני איברים... בכל שלב נעתק את האיברים שהוכנסו למערך עד שלב זה. סה"ב עלות סדרת הפעולות  $O(n^2)$ .

- **במה לוקחת ממוצע כל פעולה Insert-Last?** ( $O(n) = \frac{o(n^2)}{n}$ )
  - **הכפלת (Doubling):** נכפיל את הגודל של המערך. נתחל מאיבר 1, ונגדיל את המערך לגודל 2, ועוד 4, ועוד 8, וכל פעם "נרווח" הכנסות (פעולות קלות, זולות) שנוכל לבצע לפני ששוב צריך לבצע פעולה בבדה של הגדלת המערך (תווך העתקה כל האיברים הנוכחיים). ככל יותר הפעולות הזולות ממננות את הפעולות היקרות יותר – ומאזנות את הูลות הכוללת של כל הסדרה, מorigdot את הממוצע הכללי. נניח שמספר האיברים הסופי  $n$  שהוכנסנו הוא  $2^{k+1} < n \leq 2^k$ .
  - עלות של הכנסות היא  $\Theta(1)$  כי כל אחת לוקחת  $O(1)$ .
  - עלות העתקות של המערכות היא:  $1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1$ .
  - סה"ב נקבל  $(n) = O(2^{k+1}) \leq n + 2n = 3n$

$$\frac{o(n)}{n} = O(1).$$



אם היינו מתעניינים ב-worst case של פעולה בוודדת, כל מה שעשינו לא רלוונטי. אבל אם אנחנו מנתחים את הסיבוכיות של כל הסדרה הזאת, ואומרים בממוצע כמה עולה כל פעולה בסדרה, זה ניתוח amortized ☺

סיבוכיות במקורה הגרוע – הגדרות:

- נניח שמבנה נתונים תומך ב-k סוגים שונים של פעולות:  $T_1, T_2, \dots, T_k$ .
- $T(op)$  – עבר פעולה נתונה  $op$ , כך נסמן את סוג הפעולה  $op$ .
- $worst(T_i)$  – הזמן המקסימלי שלוקח לפעולה בודדת מסווג  $T_i$ .

לכל סדרה של  $n$  פעולות:  $op_1, op_2, \dots, op_n$  נוכל לחת חסם עליון לזמן הביצוע:  $(\sum_{i=1}^n worst(T(op_i)))$ . העניין הוא, שהחסם הזה יכול להיות מאוד לא הדוק (כמו שראינו בהכפלת  $n$  מעריכים – קיבל  $O(n^2)$ ).

סיבוכיות amortized – הגדרות:

- נניח שמבנה נתונים תומך ב-k סוגים שונים של פעולות:  $T_1, T_2, \dots, T_k$ .
- $T(op)$  – עבר פעולה נתונה  $op$ , כך נסמן את סוג הפעולה  $op$ .
- $amort(T_i)$  – לכל סדרה חוקית של פעולות מתקיים:  $(\sum_{i=1}^n amort(T(op_i))) \leq time(op_1, op_2, \dots, op_n)$ . למשל בהכפלת  $n$  מעריכים בגרסת המשופרת, סיבוכיות amortized של הפעולה היא  $O(1)$ , אם נחבר את כל הפעולות ונקבל חסם של  $O(n)$  לסדרת הפעולות.

בולם: השם amortized הוא מקרה פרטי של amortized-case, אך זהו לא החסם ההדוק ביותר.

$$time(op_1, op_2, \dots, op_n) = O\left(\sum_{i=1}^n amort(T(op_i))\right) = o\left(\sum_{i=1}^n worst(T(op_i))\right)$$

שיטת הצבירה (aggregation):

כבר ראינו בהסבר הראשוני. לוקחים בנפרד את כל אחד מסוגי הפעולות, ולכל סוג זה אומרים שהסיבוכיות  $amortized$  זה פשוט הממוצע של סיבוכיות הפעולה לאורך כל הסדרה. אם יש לנו  $m$  פעולות  $op_1, op_2, \dots, op_m$  כלjn מסוג  $T_j$  אז:

$$amort(T_j) = \frac{1}{m} \sum_{i=1}^m time(op_i)$$

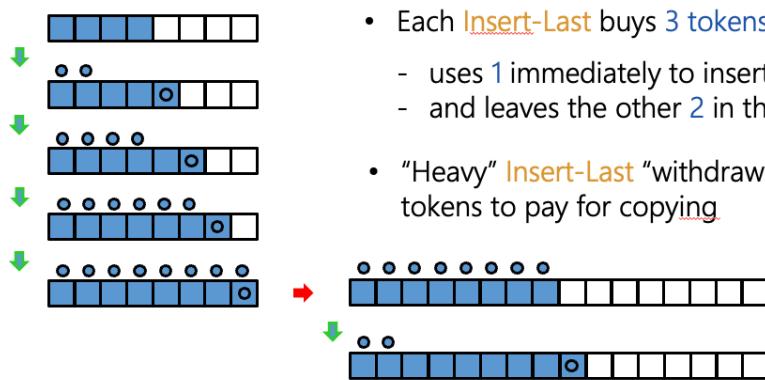
שיטת הבנק (accounting/bank):

אנחנו מנסים לאוזן בין פעולות זולות לפעולות יקרות, על ידי כך שאנחנו שומרים משאנו לצד המשך. במקרה לדבר על זמן, נדבר על מטבעות. לצורך העניין פעלת מסויימת תעלה לנו "מטבע אחד".

- כל פעולה יכולה **לקנות מטבעות** בשבייל לשלים על העבודה שהיא צריכה לעשות.
- פעולות יכולות **לקנות אקסטרה מטבעות** ולהפקיד אותן בבנק לטובות הפעולות – **פעולות זולות** יעשו את זה. הבספ' הזה יצטבר ויחבה בבנק לשימוש עתידי.
- **פעולות יכולות למשוך מטבעות מהבנק** ולממן חלק מהפעולות שלhn – **פעולות יקרות** יעשו את זה.

**בשיטת amortized** או **עלות op של פעולה היא מספר המטבעות שהיא קונה** (כולל אלו שנפקיד בבנק). אם בכלל רגע ורגע בסדרת פעולות על מבנה מסוים, כמוות המטבעות היא גודלה או שווה לאפס, אז לכל סדרת פעולות **מספר המטבעות הכללי** שככל הפעולות **קונות** – זה בעצם עלות הזמן של סדרת הפעולות.

**בדוגמת ההכפלת Doubling** – הפעולה Insert-Last תקינה בכל פעם **3 מטבעות**. באחד היא משתמשת, ועוד 2 הולכים לבנק. כך מצבירים מספיק מטבעות בבנק, כך שכאשר המערך יתמלא ווועתק למערך חדש, **יהו מספיק מטבעות בשבייל לממן את פעולות העתקה**. לאחר מכן שוב, בכל פעם הפעולה קונה 3 מטבעות. **בכל פעם שהמערך יתמלא יהו מספיק מטבעות כדי לממן את העתקה**. הנטהקה הבאה **למערך בגודל פי 2**. הפעולה הקורה של Insert-Last כאשר הzbתעה גם העתקה, גם היא קנתה 3 מטבעות. רוב הפעולות שלא מוננה על ידי פעולות Insert-Last זולות שקרו קודם. הסיבוכיות amortized של פעולה Insert-Last זה מספר המטבעות שהיא קנתה, שזה **3 מטבעות** וכך זה  $O(1)$ .



- Each **Insert-Last** buys 3 tokens.
  - uses 1 immediately to insert item
  - and leaves the other 2 in the "bank"
- "Heavy" **Insert-Last** "withdraws" previous tokens to pay for copying

- טענה: אם במערך יש  $k + \frac{M}{2}$  איברים (כלומר  $k$  איברים מימיין לאמצע), אז הבנק מכיל  $2k$  מטבעות.
- במסקנה ממנה, במקרה הפרטי שבו  $M = a$ , הבנק מכיל  $M$  מטבעות, זה בדיקת הטענה שצריך כדי להעתיק את האיברים למערך חדש בגודל בפול.
- נסיף למבנה גם פעולות Delete-Last ו-Retrieve. נאמר שהל אחית קונה מטבע 1 בזמן שהוא מתבצעת. בסה"כ קיבל שכלל פעולה ברגע במבנה פועלת בסיבוכיות amortized של  $O(1)$ .
- ציריך לשים לב לטענה הקודמת, שכן בעת הבנק מכיל לפחות  $k$  מטבעות. הוא יכול להכיל יותר כי אכן מבצעים גם מחיקה, ואז  $k$  שלו נקטן.

### שיטת הפוטנציאלי (potential):

דמייה לשיטת הבנק, אך באן במקומ להציג מראש כמה מטבעות כל פעולה צריכה לקנות, אנו הולכים להציג איך המאזן בבנק אמרור להשתנות אחריו כל פעולה – למשל המאזן/הפוטנציאלי יעללה ב-3. אנו נגידו את המאזן/הפוטנציאלי בכל מצב של משנה הנתונים.

- $.amort(op_i) = time(op_i) + \Phi_{i-1} - \Phi_i$ . להা  $.amort(op) = time(op) + \Phi_{after} - \Phi_{before}$
- אנו קוראים גם **מחיר המשוחלף**. במנוחי הבנק, זה **הפרש במוות המטבעות בבנק – אחרי הפעולה ולפני הפעולה**.
- $\sum_{i=1}^n amort(op_i) = \sum_{i=1}^n time(op_i) + \sum_{i=1}^n (\Phi_i - \Phi_{i-1}) \geq \sum_{i=1}^n time(op_i) - \Phi_0 = M - n$ . לכן קיבלנו כי הזמן הכלול חסום מלמעלה על ידי סכום סיבוכיות amortized של הפעולות (סכום המחיר המשוחלפים).

בדוגמה הכפלה (Doubling) – מערך בגודל מקסימלי  $M$ , כאשר יש בו ברגע  $n$  איברים.

$$\text{נדיר פונקציית פוטנציאלי: } \Phi(n, M) = \begin{cases} 2n - M & \text{if } n \geq \frac{M}{2} \\ 0 & \text{otherwise} \end{cases}$$

שבי מטבעות (במנוחים של בנק) או **שתי יחידות פוטנציאלי** לכל אחד כזה:  $M - n$ , ובתנאים  $2 \left( n - \frac{M}{2} \right) = 2n - M$ .

הערה: אמרנו קודם כי יש לנו  $k$  איברים מימיין לאמצע. ככלומר במוות האיברים במערך היה  $k = a$ . לפי פונקציית הפוטנציאלי שהגדכנו:  $M - a = 2 \left( \frac{M}{2} + k \right) = 2k$ . וזה בדוק במוות המטבעות בבנק שקדם אמרנו שיש. ככלומר אכן כל מטבע מימיין קיבל 2 "יחידות פוטנציאלי". ככלומר, הפוטנציאלי הטוב הוא מאזן הבנק שמצוון קודם (זה לא קורה תמיד!).

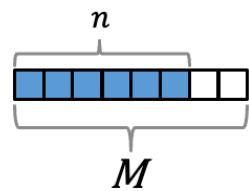
נחשב  $(n, n) amort(InsertLast) = time(InsertLast) + \Phi(M, n+1) - \Phi(M, n)$ . נטען שתחת ההגדרה הזאת נקבל כי  $amort(InsertLast) = O(1)$  וכן  $amort(InsertLast) \leq 3$ .

- בשחזור לא מלא – הרכנסה היא בעלות קבועה של 1. הפרש הפוטנציאלי יהיה קטן או שווה ל-2. עברו הרכנסה המתבצעת לחلك שלפני אמצע המערך, הפוטנציאלי לפני והאחרי זיהוי וההפרש הוא 0. כמו כן, במצב בו נכנס איבר חדש מעבר למרכז, קיבל שהפוטנציאלי עלה עבורי ב-2.
- בשחזור מלא ( $M = n$ ) – אנו מבצעים הכפלה של גודל המערך מ- $M$  ל- $2M$ . ביעון שהרכנסה מבוצעת למערך מלא, אנו מעתיקים את כל  $M$  האיברים מהמערך החדש למערך החדש, ועוד מבצעים את הרכנסה של האיבר בוסף – **סה"כ  $+M$** . לפני הרכנסה הפוטנציאלי הוא  $M$  (כי יש  $\frac{M}{2}$  איברים מעבר לאמצע), ולאחר הרכנסה גודל המערך הוא  $2M$  ויש בו איבר אחד מעבר לאמצע כך שהפוטנציאלי הוא  $2$ , סה"כ:  $2M - M = 2 - 2M - (2M - M) = 2 - 2M + 2 = 2 - M$ . כאמור,  $\Phi(2M, M+1) = \Phi(2M, M) + \Phi(M, 1)$ . גם כאן קיבל בסופו של דבר:  $amort(Insert - Last) = M + 1 + 2 - M = 3 \leq 3$ .



## Potentials – Arrays with Doubling

$$\Phi(M, n) = \begin{cases} 2n - M & \text{if } n \geq \frac{M}{2} \\ 0 & \text{otherwise} \end{cases}$$



*amort*(Insert-Last)  $\leq 3$ :

Case 1: array is **not full** ( $n < M$ )

$$\text{amort}(\text{Insert-Last}) = \underbrace{\text{time}(\text{Insert-Last})}_{1} + \underbrace{\Phi(M, n+1) - \Phi(M, n)}_{\leq 2}$$

Case 2: array is **full** ( $n = M$ )

$$\text{amort}(\text{Insert-Last}) = \underbrace{\text{time}(\text{Insert-Last})}_{M+1} + \underbrace{\Phi(2M, M+1) - \Phi(M, M)}_{2-M}$$

נסיף את הפעולות *Retrieve-Last* ו-*Delete-Last*:

- $\text{amort}(\text{DeleteLast}) = \text{time}(\text{DeleteLast}) + \Phi(M, n-1) - \Phi(M, n) = 1 + (-2|0) = O(1)$ . אם אנחנו נמצאים מימין לאמצע אנחנו נפטרים מפוטנציאל של 2 (**בשיטת הבנק לא נפטרנו מטבעות מיותרות**), ואם אנחנו מוחקים משמאל לאמצע הפוטנציאל לא משתנה. לעומת, העלות יכולה להיות אפילו שלילית.
- $\text{amort}(\text{Retrieve}(i)) = \text{time}(\text{Retrieve}(i)) + \Phi(M, n) - \Phi(M, n-i) = 1 + 0 = O(1)$ . הפוטנציאל לא משתנה כי המבנה לא משתנה.

### מערכות דינמיים

בעת נ עסק במערכות המשנים את גודלם (לא רק הכפלת פא 2 כמו שראינו מוקדם יותר).

הגדלה לא בהכרח פי 2: יש כאן trade-off בחר בין זמן לחיצון. נניח שאנו מגדיל את המערך באיזשהו פקטור בפי  $a$  – פא  $a + 1$  כאשר  $a > 0$ . סיבוכיות *amortized* של פעולה Insert-Last תהיה  $O(\frac{1+a}{a})$ .

### כיווץ מערך גדוֹל:

רענון לא טוב – כאשר  $\frac{M}{2} = a$  כלומר המערך חצי-ריק, נקטין את המערך בחצי. סיבוכיות מקירה גרווע-*amortized* היא  $O(1)$ . כדי להראות את זה צורכים למצאו סדרה של  $m$  פעולות שעולה בסה"כ יותר מאשר ( $m$ ):

נניח שבאשר המערך מלא אנו מכפילים את הגודל, ובאשר אנחנו מוחקים איבר אחד חוזרים למצב שב  $\frac{M}{2} = a$  לצורך לבוע את הגודל לחצי. ואזשוב ננכיס וגעתייק, ושוב נמחק. כל פעולה כזו תקח  $O(a)$  ו- $m$  בכלל לוקחות  $O(mn)$ .

רענון טוב – נכווץ את המערך כאשר  $\frac{M}{4} = a$  ולא חצי. הסיבוכיות של הפעולות ב-*amortized* תהיה  $O(1)$ :

שיטת הבנק – הפעולה Insert-Last – תקינה 3 מטבעות. הפעולה Delete-Last – תקינה 2 מטבעות (1 אקסטרה שיילך לבנק). כך כאשר נגיע ל- $\frac{M}{4} = a$  יהיו לנו לפחות  $\frac{M}{4}$  מטבעות בבנק (לهم אנו זקוקים כדי להעתיק את האיברים למערך המכוזח החדש).

שיטת הפוטנציאלי – בניית פונקציית פוטנציאלי בצורה הבא: כאשר אנחנו נמצאים איברים ממשאל לאמצע, נרצה

$$\Phi(M, n) = \begin{cases} 2n - M & \text{if } n \geq \frac{M}{2} \\ \frac{M}{2} - n & \text{if } n < \frac{M}{2} \end{cases}$$

שפוטנציאל המבנה יהיה לפחות  $k$ :

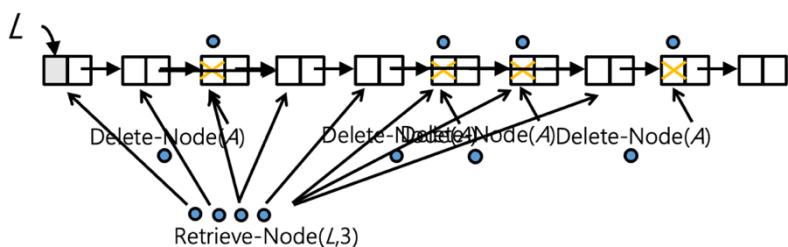
**מחיקה עצלה מרשימה מקווארת**

הבעיה: נתונה לנו רשימה מקווארת חד-כיוונית, ומנו רוצים למחוק איבר שנתון לנו מצביע A אליו.

- פשוט לבצע השמה של **ללאט למפתח** שיש לנו מצביע אליו -  $O(1)$  ב-WC. זה **מחיקה עצלה**.
- תצטרכ להגיע לאיבר ה- $i$ , ובעבר על הרשימה נמחוק פעיות את כל ה指挥ים שיש עליהם סימן של **null**.

מסתבר שהדברים האלה מסתדרים ב-**amortized**. כאשר נבצע **RetrieveNode( $i_k$ )**, הפעולה צריכה לעבור על  $i_k$  איברים, ועוד  $d_k$  שאלות הצמתים שנמחקו בעבר (סימון ב-**null**). לכן נקבל ב-WC שהסיבוכיות היא  $O(i_k + d_k + 1)$ . הבעיה היא ש- $d_k$  הוא ביטוי שכלל לא חסום על ידי  $i_k$ !

רעיון: נשים לב כי בפעולת של **Retrieve**, **אי אפשר למחוק יותר צמתים מאשר נמחקו בעבר בצורה עצלה ב-Delete**. אז אם נמחקו בסך הכל עד עכשווי ב-Delete-Node( $i_k$ ) בדומה לעד 100 צמתים אזי נוכל להגיד כי  $d_k \leq 100$ . תהיה בדקה לא יותר מאשר במota פעולות הממחקה העצלות והזרלות שקרו קודם.



ניתוח בשיטת הבננה:

הפעולה **Delete** תקנה 2 מטבעות – אחד לשימוש עבור הממחקה העצלה, והשני ישמר בבנק לשימוש מאוחר יותר על ידי **Retrieve**. הפעולת **Retrieve** מטבעת נזק – כדי להגיע אל הצומת ה- $i$ .

המשמעות של המטבעות – **Delete** הפקידה קודם (לכל איבר שימחק בעתיד הופקד מטבע לביצוע הממחקה האמיתית בהמשך).

ניתוח בשיטת הפטונצייאל:

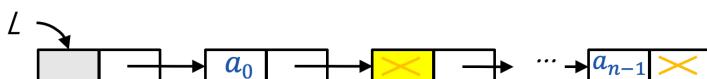
$$\begin{aligned} \text{Insert-After}(A, B) &: \text{buy \$1} = O(1) \\ \text{Delete-Node}(A) &: \text{buy \$2} = O(1) \\ \text{Retrieve-Node}(L, i) &: \text{buy \$}(i+1) = O(i+1) \end{aligned}$$

בשיטת הבנק שמננו מטבע 1 על כל צומת שמחקנו (סימון ב-**null**). לכן הגוינו לבחור באן **פונקציית פוטנציאלי** שתהווה מספר ה-nodes המוחקים (אלו שסימנו אותם כשביצענו מחיקה עצלה).

$$amort_{\Phi}(op) = time(op) + \Phi_{\text{after}} - \Phi_{\text{before}}$$

$$\begin{aligned} amort(\text{Insert-After}(A, B)) &= 1 + 0 = O(1) \\ amort(\text{Delete-Node}(A)) &= 1 + 1 = O(1) \\ amort(\text{Retrieve-Node}(L, i_k)) &= (i_k + d_k + 1) - d_k = O(i_k + 1) \end{aligned}$$

ניתוח בשיטת הצבירה:



What is the W.C. **total cost** of any sequence composed of

- $n$  **Insert-After** operations,
- $d$  lazy **Delete-Node** operations,
- $m$  **Retrieve-node( $i_k$ )** operations on positions  $i_1, i_2, \dots, i_m$  in any order?

$$\text{Total cost} = O\left(n + d + \sum_{k=1}^m (i_k + d_k + 1)\right) = O\left(n + 2d + \sum_{k=1}^m (i_k + 1)\right)$$

$$\begin{aligned} amort(\text{Insert-After}) &= O(1) \\ amort(\text{Delete-Node}) &= O(2) = O(1) \\ amort(\text{Retrieve-Node}(i)) &= O(i+1) \end{aligned}$$

$$\sum_{k=1}^m d_k \leq d$$

זמן ה-amortized **amortized** של כל פעולה, מקיים את אי השווון של הגדרת amortized. אם נכפיל כל פעולה בכמות הפעולות שקרה לנו בסכום, נקבל שהעלות הכוללת אכן קטנה או שווה מסכום זה.

## 2 – מבני נתונים

### עצים חיפוש ביןראיים - BST

#### ADT Dictionary

נדיר טיפוס נתונים מופשט בשם **מילון**: פעולה בשם **Dic-item** שתיצור איבר שיבנס למילון (מפתח + ערך), ופעולות נוספות.

**Dic-Item(*k*, *v*)** – Create a **dictionary item** *x* containing **key *k*** and **value *v***,  
*x.key*, *x.val* – The **key** and **value** contained in **Dic-Item *x***.

<b>Dictionary</b> –	Create an empty dictionary.
<b>Insert</b> –	Insert a given Dic-item.
<b>Delete</b> –	Delete a given Dic-item (assuming it exists).
<b>Search</b> –	Find a Dic-item with a given <b>key</b> (if exists).
<b>Min</b> –	Return a Dic-item with the minimum key.
<b>Max</b> –	Return the Dic-item with the maximum key.
<b>Successor</b> –	Return the successor of a given Dic-item.
<b>Predecessor</b> –	Return the predecessor of a given Dic-item.

Assume that **Dic-Items** have **distinct keys**.

נרצה למשת את כל הפעולות בסיבוכיות (*logn*) *O* בממוצע, אבל (*a*) *O* במקורה הגורע (בהמשך נראה גרסאות אחרות של עצים שבהן יכול לוותר על הסיבוכיות הזאת).

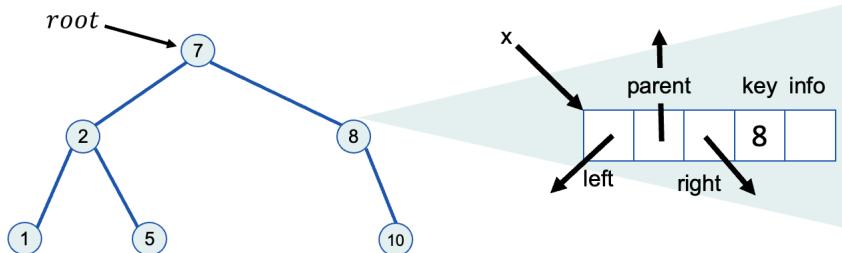
#### הגדרות ותכונות

##### הגדרות בסיסיות:

Definition: A **binary tree** in which each node contains a **Dic-item**, and satisfies the **binary-search-tree** property:

If *y* is in **left** subtree of *x*, then *y.key* < *x.key*.

If *y* is in **right** subtree of *x*, then *y.key* > *x.key*.



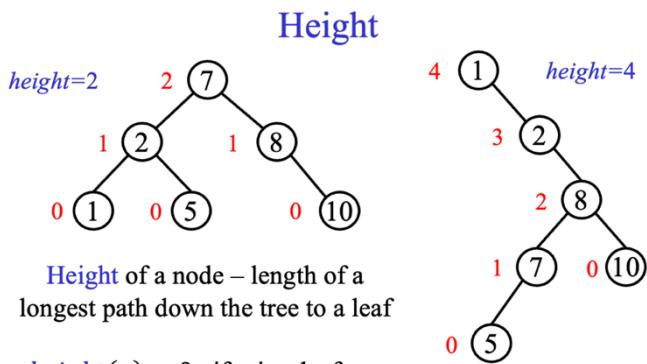
- **עץ ביןארי** – לכל צומת יש לכל היוצר שני בנים. כל צומת מכיל איזשהו איבר שנמצא בתוך המילון.

- **עץ חיפוש ביןארי – המפתחות של האיברים צריכים לקיים תכונה מסוימת.**  
אם *ע* נמצא בתת-העץ השמאלי של *א*, אז המפתח של *ע* צריך להיות קטן מהמפתח של *א*. באופן מקובל, אם *הו* נמצא בתת-העץ ימני הוא צריך להיות גדול יותר. תכונה זו מתקיימת לא רק עבור השורש, אלא לכל צומת בעץ זהה.

- **"עלים חיצוניים"** – מה מכילים מצביעים במקורה שלצומת מסוים אין בן אחד/שניים? המצביעים יצביעו לצומת שלא יוכל אינפורמציה רלוונטי. השיטה הזאת יכולה לחסוך כל מיני מקרים קצה. כדאי לשמר נציג אחד של "בן חסר" ומכל מקום שייהי מצביע לנציג זהה, זה יהיה יותר חסכוני. אפשר גם פשוט להציביע ל-null.



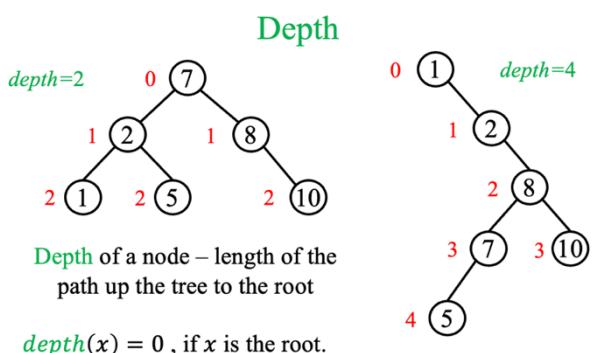
הגדרות:



- **גובה של צומת** – אורך המסלול המקסימלי מהצומת לאיזזחו עלה בעץ. גובה של צומת שהוא עלה הוא 0. נשים לב כי גובה צומת כלשהו הוא **המקסימום מבין הגבהים של ילדיו** (כי אם לא ידועים איזה תחת-עץ יותר גבוה), **ונוסיף אחד**. גובה של עץ הוא גובה של השורש של העץ.

Height of a tree = height of root

14



- **עומק של צומת** – אורך המסלול ממנו אל השורש. לעומת גובה שבו הסתכלנו למטה, בעומק אנחנו מסתכלים למעלה כלפי השורש. עומק השורש הוא 0. עומק של צומת כלשהו הוא **העומק של ההורה של הצומת**, **ונוסיף אחד**.
- **עומק של עץ** הוא העומק המקסימלי של צומת בעץ (העומק המקסימלי מבין כל העלים).
- **עומק העץ** = גובה העץ.

Depth of a tree = maximum depth = height of tree

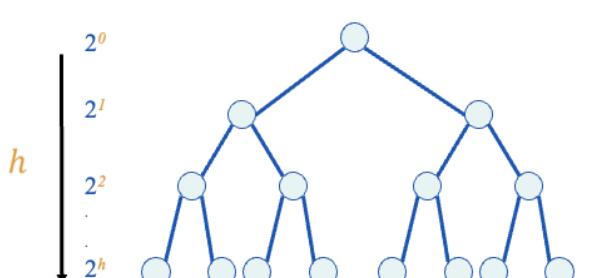
פעולות שאינן משנהות את העץ

**חיפוש (Search):**

בשל התוכנה שמקיימים BST, נדע לנוט בעץ ולבחור ימינה או שמאליה, בהתאם להשוואה שנערכות בין האיבר שאנו חיפשינו לבין ה"שורש" הנוכחי. הפעולה בImplementation הנקרא Tree-Search.

חיפוש מינימום – נמצא את הצומת השמאלי ביותר מהשורש (nge'ulah), והוא יהיה הכטן לפני התוכנה בעץ זה.

- בשורש ימינה – חיפוש מינימום יהיה  $O(1)$  כי זה יהיה השורש (שבמקרה זה אין לו בן שמאליו ולכן הוא הכטן).
- בשורך שמאליה – חיפוש מינימום יהיה  $O(n)$  כי מדובר בעלה.

A complete tree of height  $h$  contains  $n = 2^{h+1} - 1$  nodes

$$h = \lceil \log_2(n+1) - 1 \rceil$$

A tree with  $h = O(\log n)$  is called balanced

ניתוח סיבוכיות – חיפוש:  $O(h + 1)$ . בהנחה שיש  $n$  צמתים בעץ, מתקיים  $1 \leq h \leq \log n$ . העץ היבי גובה שאנו יכולים לקבל הוא "שורך" שבו בכל רמה יש צומת בודד. העץ היבי נמוך שאנו יכולים לקבל הוא בעל רמות מלאות (בכל רמה  $k$  יש  $2^k$  צמתים), ועץ זה נקרא **עץ חיפוש מאוזן**.

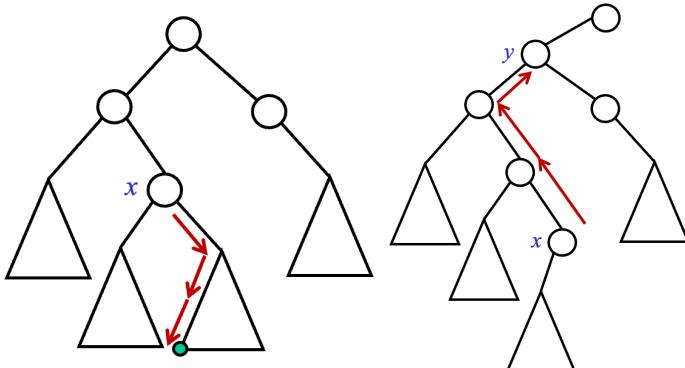


### יעקב/קדם (Successor/Predecessor):

**חישוב עוקב** – אנו מקבלים איבר נתון  $x$ , ואנו רוצים למצוא את האיבר הבא בסדר ממוין. כלומר, האיבר בעל המפתח הכי קטן כל מהפתחות שגדולים מהפתח של  $x$ . כדי למצוא את העוקב:

- בתת-העץ הימני שלו (הפתחות שגדולים ממנו), נלק כל הדרך שמאליה (לאיבר הכי קטן בתת-העץ זהה).
- אם אין בן ימני, נעלם לעלה (דרך צמתים שקטנים מ- $x$ ), עד הפעם הראשונה שבה יוכל לעלות ימינה (מצא מחת-עץ שמאל של צומת שהוא הכי קטן, מבין אלה שגדולים מ- $x$ ).

סיבוכיות:  $O(h + 1)$  כאשר  $1 \leq h \leq n$ .



In order to find successor of item  $x$ :

If  $x$  has a right child,

Go right once, and then left all the way

Otherwise,

Go up from  $x$  until the first turn right

*Predecessor* is symmetric.

### Function Successor( $x$ )

```
if x.right ≠ null then
    ↘ return Min(x.right)
y ← x.parent
while y ≠ null and x = y.right do
    ↘ x ← y
    ↘ y ← x.parent
return y
```

### חישוב קודם – סימטרי:

- בתת-העץ השמאלי שלו (הפתחות שקטנים ממנו), נלק כל הדרך ימינה (לאיבר הכי גדול בתת-העץ זהה).
- אם אין בן שמאלי, נעלם לעלה (דרך צמתים שגדולים מ- $x$ ), עד הפעם הראשונה שבה יוכל לעלות שמאלה (מצא מחת-עץ ימני של צומת שהוא הכי גדול, מבין אלה שקטנים מ- $x$ ).

### פעולות שימושות את העץ

#### הכנסה (Insert):

נமמש את הפעולה Tree-Insert שמקבלת מצביע לשורש העץ, איבר  $z$  שאנו רוצים להכניס. נעשה פעולה שמאוד מזכירה חישוב, נחפש את  $z.key$  וובכל רגע נשמר את הצומת האחרון שהיינו בו. באשר החישוב יסתהים בבישולו, בעוד בדיק להכניס אותו. צומת חדש שנכנס, תמיד נכנס בתור עלה לעץ.

באיזה סדר נכניס טווח ערכים עבור עץ הכיו גרען (גובה)? בסדר ממוין מהקטן **לגדול** (שורץ ימין), בסדר ממוין **מהגדול לקטן** (שורץ שמאלה), בכל שלב נכניס לシリוגן צומת מקסימלי/מינימלי (זיג-זג)

באיזה סדר נכניס טווח ערכים עבור עץ הכיו טוב (נמור)? **נבחר את החציוון** ונכניסו בתור שורץ, בעוד דומה לפעול על האיברים שקטנים מהשורץ, ועל האיברים שגדולים מהשורץ, נחבר את תמי ה-עציים שנוצרו בבניים השמאלי והימני של השורץ.



מחיקה (Delete):

## Deletion of a binary node

If  $z$  has two children,  
let  $y$  be the successor of  $z$

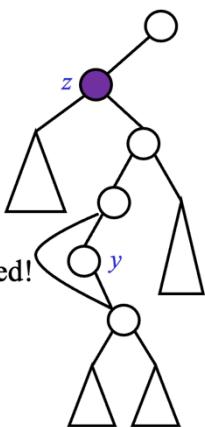
$y$  has no left child

Remove  $y$  from the tree

Replace  $z$  by  $y$

Binary-search-tree property preserved!

Is it enough to let  $z.key \leftarrow y.key$ ?  
And maybe also  $z.info \leftarrow y.info$ ?



נקבל **מצבייע לצומת x** שאנו חנכו רוצים למחוק מהעץ. בעצם נפריד למקרים:

1.  $x$  הוא עלה – פשוט נסיר אותו.
2.  $x$  יש רק אחד – צריך למחוק אותו ולבצע מעקב:  
לחבר את ההורה של  $x$  לדוד שלו (אם קיים).
3.  $x$  יש שני ילדים – נסתכל על העוקב שלו, שהוא  $y$ .  
בזודאות אין  $y$  בן שמאל (אחרת הבן זהה העוקב),  
יש לו לפחות אחד (ימני): נמחק את  $y$  מהעץ (זו  
מחיקה קלה כי אין לו בן שמאל), נחליף את הצומת  $x$  ב- $y$ .

### בנייה BST באופן אקראי

טענה: כאשר עץ חיפוש בינארי נבנה באופן אקראי עם הכנסות של  $n$  מפתחות שונים, **וחוללת גובה העץ היא  $O(\log n)$** .

וריאציה: מה זמן הריצה הממוצע של חיפוש איבר בעץ שנבנה באופן אקראי? יש לנו כאן ממוצע בשני מובנים:

1. על פניו סדר הכנסה – נמצא את הגובה על פני הגבהים השונים שקיבלנו.
2. את מי אנו מתחפשים בעץ – את השורש, או צומת שנמצא בעומק העץ.

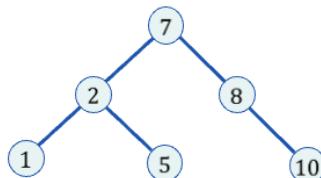
נכיח את המשפט: כמות ההשוואות הממוצעת בעת חיפוש איבר תחת ההנחהות (מתבצעות הכנסות בלבד, וכל הפרמטריות הן שוות הסתברות), היא לפחות  $\log n + 4$ . הוכחה המלאה נמצאת [כאן](#).

### סיוור בעץ (Tree Walks)

ישנן 3 דרכים לסיר בעץ (לעבור על כל צמתיו):

1. In-Order: שמאל-שורש-ימין. **נקבל סדרה ממינית בסדר עולה.** אין שום קשר בין הסיריקה **למבנה העץ**, כל עץ חיפוש בינארי שמכיל את האיברים המופיעים בצורה כלשהי, סיריקה זו עבורה תהיה תמיד ממינית.  
טיפ: לדמיין ציר א', ולסroxק ממשמאלי לימיין את כל האיברים (להרחיב את העץ בדימוי על גבי הציג אם צריך).
2. Pre-Order: שורש-שמאל-ימין. נDIR מאוד לשימוש. בהינתן סיריקה זו של עץ ניתן **לשחזר את העץ באופן ייחידי!**
3. Post-Order: שמאל-ימין-שורש. טוב לחישוב סכום איברים למשל (קדום בנים ואז האב).

### Tree Walks



Pre-Order( $x$ ):

```
If  $x \neq null$  then
  Process( $x$ )
  Pre-Order( $x.left$ )
  Pre-Order( $x.right$ )
```

In-Order( $x$ ):

```
If  $x \neq null$  then
  In-Order( $x.left$ )
  Process( $x$ )
  In-Order( $x.right$ )
```

Post-Order( $x$ ):

```
If  $x \neq null$  then
  Post-Order( $x.left$ )
  Post-Order( $x.right$ )
  Process( $x$ )
```

שימוש אחד בסיוור Post-Order הוא עבור אחסון ופתרון של ביטוי ארכיטמטי. אם ביצע סיוור In-Order ניתקל במקרים של ambiguity כי לא ברור צריכה צרך להוסיף סוגרים בחישוב. בסיוור Post-Order נקבל ביטוי postfix שניתן לחשב בעזרת אלגוריתם שמשתמש במחסנית.

ניתוח סיבוכיות:

טענה: סיבוכיות זמן הריצה של סירור בעץ בעל  $h$  צמתים היא  $O(dn)$  כאשר  $d$  הוא זמן הביקור בצומת בודד ( $0 < d$ ).

נרצה להראות כי  $1 + n = dn$ . הביטוי  $1 + n$  מתייחס לזמן הביקור בצמתי בעץ, הגיעו לצומת כזה וביצוע בדיקה. נוכחים באינדוקציה: נכח את נכונות הטענה לכל  $n < k$  ונוכיח עבור  $n$ .

- מקרה בסיס:  $0 = n$ : קיבל  $1 = 0$ . באשר מגאים לעז ריק יש פעולה בודדת כדי לזרות את זה.

הוכחה:  $T(n) = T(k) + T(n - k - 1) + d$ , חלוקה לתת-עצים משමאל ומימין בתוספת הזמן הביקור בשורש. נשים לב כי  $k \leq n - 1$

$$T(n) = dk + k + 1 + d(n - k - 1) + (n - k - 1) + 1 + d = dn + n + 1 = O(dn)$$

עצוי AVL**הגדרה**

ראינו כי BST יכולים להיות בעלי גובה LINEAR במספר הצמתים (במקרה של שורר למשל) ובכך להוות מימוש לא יעיל מספק עבור פעולות המילון השוכנות. מה שקובע את גובהו של העץ הן פעולות הרכנשה והמחיקה (הפעולות הדינמיות שימושות את העץ). בהקשרים מסוימים נוכל להניח הנחות מגבלות על אופן השימוש במבנה, אך לא תמיד. בשיטת הייעולות זו נפרט ע"י קבוצה של מבני נתונים המכונים עצים חיפוש מאוזנים (balanced search trees). עצים כאלה מכילים מנגנון מוגן **של תיקון העץ** לאחר הרכנשה או מחיקה (ולעתים גם לאחר חיפוש).

נרצה למנוע מצב שבו יש צמתים שיש להם נטיה חזקה לצד ימין או לצד שמאל – כדי לא לקבל עץ מאוד לא מאוזן.

**גורם האיזון (Balance factor)** – לכל צומת, זה ההפרש בין הגובה של תת-העץ השמאלי שלו, לגובה של תת-העץ ימני שלו:

$$BF(v) = h(v.left) - h(v.right)$$

**עץ AVL** – עץ חיפוש BINARİ, אשר לכל צומת  $v$  ביעילות מתקיים:  $|BF(v)| \leq 1$ . כלומר BF המותרים לכל אחד מצמת היעץ הם הערכים הבאים: **1, 0, -1**.

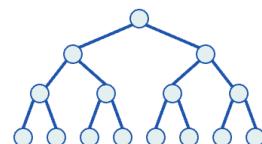
ניתן לראות כי **עץ AVL** בעל  $h$  צמתים מקיים  $O(\log n) = h$ . לעומת זאת, העץ הוא עץ מאוזן. לכן, כל הפעולות יתבצעו באופן בסיבוכיות של  $O(\log n)$ WC-B.

**Upper Bound of an AVL Tree Height****Intuition**

In a balanced tree  $h = O(\log n)$ . Meaning  $n = \Omega(\alpha^h)$ , for some constant  $\alpha > 1$ .

**Literally:** the number of nodes in a balanced tree is exponential in the height of the tree.

For example, in a perfectly balanced binary search tree (complete binary tree),  $h = O(\log n)$  and  $n = \Omega(2^h)$ .



We will prove that the above holds for an **AVL tree** with the constant  $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$  (golden ratio), hence  $h = O(\log_\Phi n)$ .

**הכנסה לעץ AVL****אלגוריתם הכנסה:**

לאחר הכנסה של ערך חדש לעץ, כיצד נתמודד עם צמתים שהם בעט "עברית לAVL"? בולומר ה-BF שליהם בערך מוחלט גדול מ-1. **ניקח צמתים מסוימים ונסדר אותם מחדש**, בפועל שדומה לסיבוב. כך נקבל עץ AVL חוקי. מספר אבחנות:

1. בהכנסת צומת חדש לעץ – הגובה של כל צומת הנמצא על המסלול אליו הכנסנו את הצומת החדש גדול לכל היוטר ב-1. טרם הכנסה העץ הוא AVL תקין, משמעו לכל צומת  $z$  בעץ מתקיים  $1 \leq |z|_{BF} \leq n$ . בולומר, **גורם האיזון משתנה ב-1 לכל היוטר.** **ולכן לא יכול להיות גדול מ-2 בערכו המוחלט.**
2. הכנסה לא משפיע על גובהם של צמתים שלא נמצאים על **מסלול מהשורש לצומת החדש**. בפרט, גורם האיזון שלהם לא ישתנה. לעומת זאת, גובהם של צמתים שנמצאים על המסלול זהה עשוי לשנתנו בעקבות הכנסה, ובעקבות כך גם גורם האיזון שלהם (**זהרי הוא מושפע משינוי של גובה אחד מתי-העצים שלו**).  
בולומר אנו צריכים לטפל רק ב- $O(\log n)$  צמתים, כי הגובה של העץ הוא  $O(\log n)$ .
3. **לא כל הצמתים במסלול הנ"ל משנים את ה-BF שלהם.** לא כל הצמתים במסלול משנים את הגובה שלהם. אם יש צומת על המסלול הנ"ל שהגובה שלו לא השתנה בעקבות הכנסה, אז **כל הצמתים שמעליו לא יחוו שינוי BF שלהם.** בולומר, מספיק לטפל רק באוטם צמתים על המסלול, מהצומת החדש שהוספנו ועד לשורש, עד **המקום הראשון שבו גובה של תת-עץ כלשהו לא השתנה ושם אפשר להפסיק.**

נשים לב כי הסיבוכיות של אלגוריתם זה היא  $O(h + 1) = O(\log n)$ .

**גלגולים (Rotations):**

בזמן שאנו מתחילה לתכנן עץ AVL, הדבר הראשון שנשאל הוא האם העבריין הראשון הרាលן שהגענו אליו מלמטה כלפי מעלה, הוא מסוג 2 או 2+. הדבר השני שנשאל, הוא לגבי אחד מהבנינים של העבריין.

- במקרה שעבריין הוא מסוג 2-, נשאל מה ה-BF של הבן הימני שלו – שיכל להיות  $1 \pm$ . צד ימין גבוה יותר ב-2 מצד שמאל.
- במקרה שעבריין הוא מסוג 2+, נשאל מה ה-BF של הבן השמאלי שלו – שיכל להיות  $1 \pm$ . צד שמאל גבוה יותר ב-2 מצד ימין.

**טענה: בעת הכנסת צומת ל-AVL יידרש לכל היוטר גלגול יחיד.**

**הוכחה:** גלגול אחריה הכנסה, תמיד משחזר את גובה תת-העץ המקורי לפני הכנסה. לפי הבדיקה מספר 3 שראינו קודם, ברגע שצומת מסוים שאנו נתקלים בו במעלה הדרכ לכוון השורש, **לא ישנה את הגובה שלו, אין שום חשש שהצמתים שמעליו יהפכו להיות עבריין AVL, כי הם בכלל לא שינוי את ה-BF שלהם.** מכאן שלאחר הכנסה לעץ AVL, נידרש לבצע לכל היוטר גלגול אחד, אולי אפילו לא נחוץ בכלל. יכול להיות שהגלגול היחיד יבוצע קרוב לשורש (ועדיין נחוץ לפחות עד למעלה). בכל אופן מודובר בגלגול אחד בלבד.

**נסיף לאלגוריתם את 3.4 – אם גילינו עבריין וביצענו גלגול, אפשר פשוט לסיים את האלגוריתם בנקודת הזאת ללא חשש.**

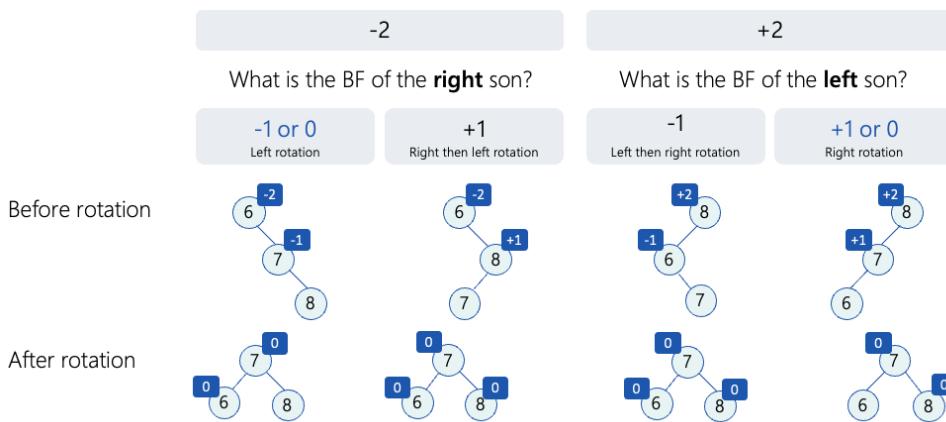
**מחיקה מעץ AVL**

רוב הפרטים דומים למה שקרה בהכנסה:

- מחיקה של צומת מהעץ יכולה לגרום לעבריינות AVL – כאשר ה-BF הוא  $2 \pm$  ולא מעבר לזה.
- העבריינים יכולים להופיע אך ורק על המסלול מהצומת שמחקנו לשורש. **הצומת שמחקנו הוא הצומת שנמחק פיזית** (למשל כאשר אנו באים למחוק צומת ויש לנו שני בני, מוצאים את העוקב שלו ומחליפים בינויהם. בעצם הצומת שנמחק פיזית הוא לא הצומת המקורי עם המפתח שרצינו למחוק, **אלא העוקב שלו**).

נתמקד בהבדלים:

1. ב邏輯 יש עוד מקרים לבסota עברית גלגולים – כאשר יש לנו BF של 0 בעברית מסוג 2±. הם מופיעים במסגרת הגלגולים המוכרים. למשל מצב שלא יכול לקרות בהבננה – שבו יהיה לנו עברית עם 2+ ובן שמאלי עם BF של 0.



2. ב邏輯 יכול להיות שנטורף לבצע גלגול אחד בכל רמה של העץ, החל מהצומת שמחקנו פיזית ועד השורש. לא בהכרח אחריו גלגול אחד אפשר לסייע.

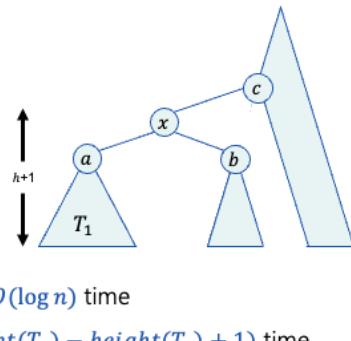
סיבוכיות – גם כאן אנחנו צריכים  $O(\log n)$  כדי למחוק צומת מעץ חיפוש בינארי בלשחו, ועוד להתחילה לעלות לפני מעלה ולבצע לכל היוטר גלגול אחד בכל רמה – זהה שב  $O(h + 1)$ . סה"כ  $O(h + 1)$ .

**איחוד ופיצול**איחוד:

יש לנו שני עצי AVL –  $T_1$ ,  $T_2$  ויש לנו צומת  $x$  שמקיים  $T_2 < x < T_1$  בולם שהמפתח שלו גדול מכל המפתחות שיש ב- $T_1$  וקטן מכל המפתחות שיש ב- $T_2$ . לבצע איחוד זו לא בעיה, אך העץ החדש שיתקבל יוכל לצאת מאוד לאมาตรฐาน (אם העצים המקוריים היוมาตรฐานים בפנים עצם). נפעיל כך:

Join( $T_1, x, T_2$ ) when " $T_1 < x < T_2$ "

Assume  $height(T_1) \leq height(T_2)$



And even  $O(height(T_2) - height(T_1) + 1)$  time

(if heights maintained explicitly, so we can find **b** while going **down**:

- הבן הימני של  $x$  יהיה תט-עץ בלבד של  $T_2$  שיש לו גובה מאד דומה לגובה של  $T_1$ . ניקח את  $a$  להיות הצומת הראשון על השדרה השמאלית של  $T_2$  שגובהו קטן או שווה  $a$ .

- מה יכול להיות גובהו של תט העץ ששורשו  $b$ ?  $h - 1$ . הוא אכן יכול להיות  $2 - h$  כי גם תט-עץ זה הוא עץ AVL ובגלל הפרש הגבהים בין  $a$  ובין צומת  $b$  הוא עשוי להיות גבוה יותר מאשר  $a$ . בסתירה לכך שהוא הראשון שגובהו קטן שווה  $h$ .

- אנו לחבר את  $x$  למי שהיא ההורה הקודם של  $b$  – נסמן אותו ב- $c$ . גובה תט-העץ ששורשו  $c$  יכול להיות  $h + 1, h + 2$ .

נתחיל לתקן את העץ מ- $x$  כלפי מעלה. הדבר הזה עולה  $O(\log n)$ .

- אפשר להגיד ממשו יותר הדוק מזה:  $O(height(T_2) - height(T_1) + 1)$ . אם אנו שומרים בכל צומת את הגבהים באופן מפורש, אז אפשר למצוא את  $c$  במהלך הירידה כלפימטה.

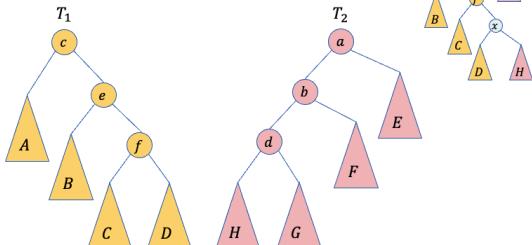
פתרונות:

בהתיקון עץ AVL וצומת  $x$ , אנו רוצים לפצל את העץ לשני עצים  $T_1, T_2$  כך שיתקיים כמו קודם  $T_2 < x < T_1$ . אנחנו נממש פיצול באמצעות פעולת האיחוד.

**Splitting AVL (by Efficient Joins)**

Given an AVL and a node  $x$ , we want to split the tree into  $T_1, T_2$  such that " $T_1 < x < T_2$ ".

Naïve analysis:  $O(\log^2 n)$



- נתחל מ- $x$ , ונסתכל על שני תת-העצים שלו. העץ  $D$  שמכיל מפתחות גדולים יותר, והעץ  $H$  שמכיל מפתחות קטנים יותר. כל אחד מהם יהיה התחלת של עץ אחר חדש. אנו עלולים לפחות מ- $n$  צמתים/גדולים מ- $x$  בהתקאה לעץ החדש הנכון. וכל פעם מאתדים צמתים שהמפתחות שלהם אינם באנדרה של איזוחדים.
- סיבוכיות – איחוד של שני עצים בעלי  $m$  ו- $n$  צמתים עולה  $O(\log mn)$ . כיוון שאנו מתחילה ב- $x$  ועלים לפני מעלה, אנחנו עלולים לכל היותר  $O(\log n)$  רמות כי מדובר על עץ AVL. בסה"כ קיבל סיבוכיות של  $O(n \log m)$ .

- אפשר לחת חסם יותר הדוק – כל איזוחד, לבניית כל עץ חדש  $T_1, T_2$ . בולם יש לנו סכום של הפרש הגבהים בין  $t_i$  לבין תוצאה ה-join שהייתה עד עכשוו. הדבר הזה שווה בסופו  $O(\log n)$ .

**Tighter Analysis**

Recall each join really takes only  $O(\text{height difference} + 1)$

To generate each of  $T_1, T_2$ ,

we need to make a telescopic join series:  $\dots (((t_1, t_2), t_3), t_4), \dots, t_k$

$$\text{time} = O\left(\sum_{i=2}^k |\text{height}(t_i) - \text{height}(\text{join}(t_1, \dots, t_{i-1}))| + 1\right) = O(\log n)$$

**Lemma 1\***:  $\text{height}(t_1) \leq \text{height}(t_2) \leq \dots \leq \text{height}(t_k)$

**Lemma 2\***:  $\text{height}(\text{join}(t_1, \dots, t_i)) \leq \text{height}(t_i) + c$

עצים דרגות**מושיבציה**

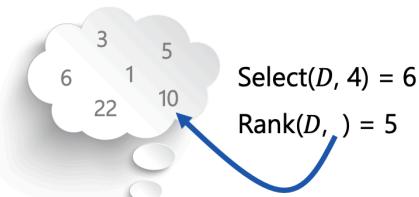
נרחיב את ה-ADTillion באמצעות 2 פעולות נוספות שנרצה לממש.

1. Select(D, k) – מקבלת מקום  $n \leq k \leq 1$  ומחרירה את האיבר ה- $k$  הכי קטן. בולם, אם נמיין את האיברים, עברו 1 נקבע את המינימום. עברו  $n = k$  נקבע את המקסימום.
2. (x, Rank(D, x)) – מחרירה את הדירוג/דרוגה של האיבר  $x$ . זהו המיקום שלו בסדר הממוין של האיברים (ה- $k$  שלו).

פעולות אלו הופכויות במובן מסוים, נשים לב:

- Select(D, Rank(D, x)) – אנו רוצים את האיבר ה- $k$  הכי קטן, כאשר  $k$  היא הדרגה של  $x$ . זה פשוט האיבר  $x$ .
- Rank(D, Select(D, k)) – אם רצים את הדרגה של האיבר ה- $k$  הכי קטן, זהו פשוט  $k$ .

For Example:



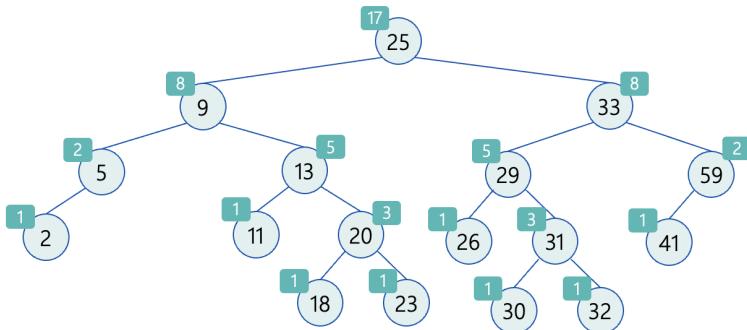
Note that:

$$\text{Select}(D, \text{Rank}(D, x)) = x$$

$$\text{Rank}(D, \text{Select}(D, k)) = k$$



## הגדרה וימוש הפעולות



**עצ' דרגות (rank trees)** – הרחבה לעצי AVL, כאשר בכל צומת נשמר שדה נוסף שנקרא `size`, שייחסיק בכל צומת את גודל תת-העץ השמאלי הזה הוא השורש שלו, כולל השורש עצמו. מובן שעבור עליים השדהזה יהיה בעל ערך 1.

### Tree-Select

עלינו למצוא את האיבר ה- $k$  הכי קטן מתוך קבוצה של איברים. ניעזר בכך שבכל צומת מחזיק מידע על כמה מה צומתים שנמצאים תחתו (כולל אותו), כדי לדעת כיצד למצוא את האיבר המבוקש.

- נניח שאנו מכחשים את האיבר ה-13 הכי קטן, וישנם 9 איברים שקטנים או שווים לשורש (8 בתת-העץ השמאלי + 1 עבורה השורש). אנו יודעים כי **עלינו לפנות ימינה**. האיבר שאנו מכחשים נמצא בתחום הימני (בעת אנחנו מכחשים את האיבר ה- $13 = 9 - 4$ ).
- אם נגלה שיש יותר מ-4 צומתים שקטנים או שווים לשורש, **עלינו לפנות שמאליה**. שם נמצא את האיבר ה-4 הכי קטן.
- כך בעצם מפעלים ברקורסיה את האלגוריתם על כל תת-עץ, תוך עדכון השורש, עד למציאת האיבר המבוקש.

### Tree-Select( $T, k$ )

- $x \leftarrow T.root$
- $r \leftarrow x.left.size + 1$
- If  $k = r$ , then the root is the required element
- Otherwise, if  $k < r$ , search for the  $k^{\text{th}}$  smallest item in the **left** subtree of the root
- Otherwise ( $k > r$ ), search for the  $(k - r)^{\text{th}}$  smallest item in the **right** subtree of the root

סיבוכיות – אנו מבלים בכל רמה בערך זמן קבוע (רק מחליטים לאן ללבת), והסיבוכיות היא לנארית בגובה העץ. אם מדובר בערך מסוין כמו עץ AVL אז הסיבוכיות היא  $O(\log n)$ .

### Tree-Rank

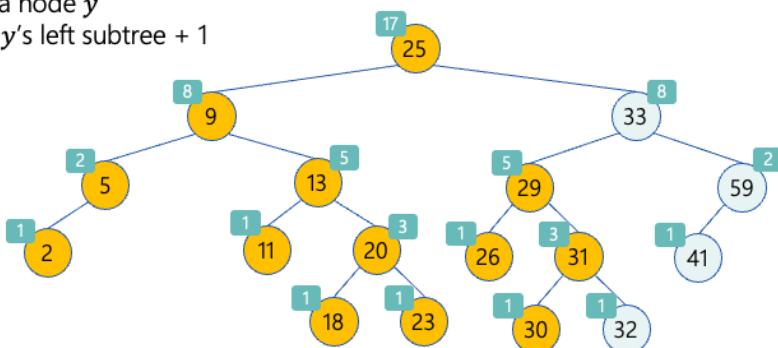
בהתנחת מציבע לצומת בעץ דרגות, علينا למצוא את הדירוג שלו. אפשר פשוט לבצע סידור in-order ואבל זה ב- $O(n)$ .

- נתחיל מצומת, ונסתכל בהמה צומתים יש ממשאלו (קטנים ממנו). ניקח את הדירוג של הבן השמאלי + 1. נתקדם לפני מעלה לביוון השורש, ונעשה אותו מהלך. כל עוד אנחנו **עלים שמאליה**, אין בעיה.
- כאשר אנחנו **עלים ימינה**, **לא צריך לבצע כלום**, כי אם לא מגלים צומתים שקטנים או שווים מהצומת המקורי שלו.

סיבוכיות – גם כאן, בכל רמה בערך עושים פעולות ב- $O(1)$  ובסה"כ נקבל  $O(\log n)$ .

### Tree-Rank( $x$ )

- Initialize a counter with the number of nodes in  $x$ 's left subtree, plus 1 (for  $x$  itself).
- Go up to the root, and:
  - Every time you go up **left** to a node  $y$  add the number of nodes in  $y$ 's left subtree + 1
- Return the final counter value





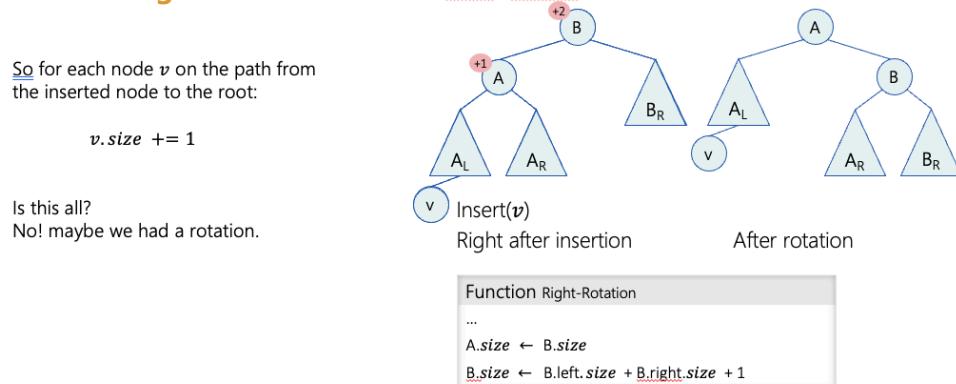
## תחזוקת הגודל בעת הכנסה ומחיקה

ביצד נתחזק את שדה `the-size` בעת הכנסה ומחיקה מוביל לפגוע בסיבוכיות פעולות אלו?

בהכנסה:

- הצמתים הייחדים שימושים את `the-size` שלהם זה צמתים על מסלול הכנסה. מדובר ב- $O(\log n)$  צמתים. את העדכון אפשר לעשות בדרך לעלה (תוך כדי שמחפשים עבריים, אולי מגללים ומתקנים), או בדרך למטה (זמן חיפוש מקום הכנסה).
- צריך לשים לב, שאם נעשה זאת בדרך למטה, לעיתים לא נכניס את הצומת (אם המפתח כבר קיים), ועוד צריכים לחזור אחרה ולהתחל לתקן חזרה את כל `the-size` שניינו.
- במקרה שהיינו צריכים גם לעשות גלגולים (בהכנסה זה רק גלגול אחד), יש בסך הכל  $(1)O$  צמתים שצריך לבדוק את `the-size` שלהם.

### Maintaining the `size` Attribute After Insert



במחיקה:

- כל הצמתים במסלול מהצומת שמחקנו פיזית ועד לשורש יקטינו את `the-size` שלהם ב-1.
- גם כאן נטפל בגלגולים (גם אם יהיה גלגול בכל רמה של העץ), אין הבדל בסיבוכיות.

## הרחבת המבנה

רצינו למשתמש ADT מילון בתוספת הפעולות `Select`, `Rank` בזמן לוגרתי. שום מבנה נתונים שהכרנו עד כה לא התאים. לדוגמה מבנה נתונים מוכבר – עץ AVL. הרחובנו אותו באמצעות השדה `the-size` לכל צומת. הראנו כיצד למשת את הפעולות החדשות. לבסוף, הראנו שפעולות דינמיות כמו הכנסה ומחיקה לא נפגשו מההטוספה, וכיtinם לתחזוק. נניח שהיינו רוצים להוסיף שדה גובה (למשל לחישוב BF בעץ AVL), או שדה עומק לכל צומת. האם נוכל לתחזוק שדות אלו בפעולות של הכנסה ומחיקה?

שדה גובה:

- כן. זה אפשרי כיון **שהגובה של צומת תליי ורק בגביהים של הילדים הישרים שלו**. בשנכניס או בשנמחק צומת צריך לעבור רק על המסלול מהצומת עד השורש, ולבצע חישובים ותיקונים ב- $O(1)$ .
- גם במקרה שנתקלנו בגלגולים, אפשר לעדכן את הגובה בקלות. מעט צמתים משנים את גובהם לאחר הגלגול.

שדה עומק:

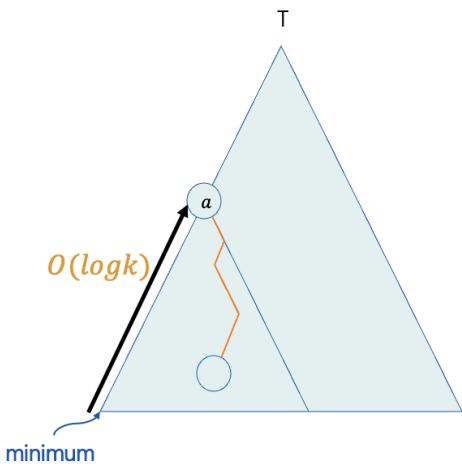
- לא. יש סיטואציות שבהן אנחנו מכנים/**מוחקים**/מגלאים צומת ונדרשים לשנות את העומק של הרבה צמתים בעץ.
- כאשר נבצע גלגול, יש תת-עץ מסוים שבו הצמתים לא שינו את העומק שלהם. אולם, יש הרבה צמתים אחרים שיכו את העומק שלהם ואני נדרשים לעדכן אותם.

**טענה:** נניח שיש לנו עץ חיפוש מאוזן בלשחו  $D$  עם  $n$  צמתים (למשל AVL), ונרצה להרחיב אותו עם שדה `f` בכל צומת. אם האינפורמציה בשדה `f` של צומת מסוים תלויי ארכ' וرك בילדים הישרים של הצומת זהה, אז אפשר לתחזוק את שדה `f` בזמן הכנסה ומחיקה בזמן  $O(\log n)$ .

לסיבום:

שדה	נitin לתחזק ביעילות?
עומק הצומת – p	לא - מוכיח או הוכנסה של צומת לעץ עשויה להשפיע על צמתים רבים בעץ, כפי שראיתם. לכן לא ניתן להחזיק את שדה העומק של כל צומת מוביל לפגוע בסיבוכיות הזמן של פעולות ההוכנסה והמחיקה.
גובה הצומת – h	כן - מוכיח או הוכנסה של צומת לעץ עשויה להשפיע רק על גובהם של אבותיהם הקדמוניים (כלומר, הצמתים שנמצאים על גבי המסלול ממנה לשורש העץ). כיצד נחזק שדה זה? לאחר הוכנסה/מחיקה של צומת, נעה במסלול מאותו הצומת ועד לשורש העץ ונעדכן בהתאם את הגובה של כל צומת על גבי מסלול זה. נזכיר כי גובה של צומת שווה למקסימלי מבין גברי ילווי + 1, וזה הסיבה לביק שתהווינו רלוונטי רק לצמתים שנמצאים על גבי מסלול זה. אורך המסלול הוא לכל היוטר בעומק העץ, ( $\log(n)$ ), ולכן אין פגעה בסיבוכיות של פעולות ההוכנסה והמחיקה. בעת ביצוע גלגולים, יש צורך לבדוק רק מספר קבוע של צמתים.
כמות הצמתים בתת-העץ בעלי מפתח זוגי – even_size	כן - מוכיח או הוכנסה של צומת לעץ עשויה להשפיע רק על ה- <code>even_size</code> של אבותיהם הקדמוניים: אלו הם בדיקות הצמתים בעץ שצומת זה נמצא בתחום העץ שלהם. כיצד נחזק שדה זה? לאחר הוכנסה/מחיקה של צומת עם מפתח זוגי, נעה במסלול מאותו הצומת ועד לשורש העץ ונעדכן בהתאם (נגדיל-ב-1/נקטין-ב-1) את שדה <code>even_size</code> . אורך המסלול הוא לכל היוטר בעומק העץ, ( $\log(n)$ ), ולכן אין פגעה בסיבוכיות של פעולות ההוכנסה והמחיקה.
המייקום של הצומת r – in-order	לא - מוכיח או הוכנסה של צומת לעץ עשויה להשפיע על צמתים רבים בעץ. למשל, אם נמחק את המינימום בעץ, ה- <code>r</code> של שאר צמתיו העץ יקטן ב-1. לכן לא ניתן להחזיק את השדה <code>encl</code> בכל צומת מוביל לפגוע בסיבוכיות הזמן של פעולות ההוכנסה והמחיקה.

### Finger Trees



נציג שכולול של עצי חיפוש ביןaries המאפשר גישה מהירה לאיברים מסוימים בעץ. נרצה לעשות  $O(\log k)$  Select( $T, k$ ) ולא כמו שעשינו עד כה –  $O(\log n)$ . זה ייעיל אם ( $n = k$ ). **בפועל מדובר בעץ AVL עם מצביע למינימום.**

בעצם Finger Trees נשמר מצביע לאיבר המינימלי (אפשר לתחזק מצביע זהה ביליאר בעקבות הוכנסה ומחיקה). נתחילה לעלות מהמינימום לפני מעלה, עד שנגיע לתוך-עץ שיש בו לפחות  $k$  צמתים (בעץ דרגות יש שדה `size`). בעת אנו יודעים שהאיבר שאנו מחפשים נמצא בתחום-העץ הזה. בתחום-העץ זה יש לפחות  $k$  צמתים והם הצמתים היכי קטנים בעץ. כל שאר העץ לא רלוונטי יותר. **כעת בוצע פשטוט Tree-Select( $a, k$ ) באשר נתחילה מהצומת  $a$  ולא מהשורש. בעץ עם לפחות  $k$  צמתים הגובה הוא  $\log k$ .**

### AVL Revisited

נזכר בהוכנסה לעץ AVL. ההוכנסה מורכבת מירידה בעץ בעלות  $O(\log n)$  כדי למצוא את מקום ההוכנסה, ועליה תורן כדי איזון העץ (וביצוע גלגול בסוף) גם בעלות  $O(\log n)$ . נרצה להראות שהעליה לוקחת  $O(1)$  בזמן amortized. בחלק ייחסית קטן של המקרים נדרש לעלות עד למעלה ולעשות גלגול, לרוב נעה מעט.

**טענה 1:** בכל סדרה של הוכנסות בלבד לעץ AVL, העלות amortized של התיקון של העץ (שלב 2) הוא  $O(1)$  להוכנסה.

בדיוון זה – "צמתים מאוזנים" אלו צמתים שעבורם  $BF = 0$ .

אבחנה:

- אם  $BF = 0$  אחרי ההוכנסה, זה אומר **שהגובה של הצומת לא השתנה**. אמנם, צמתים ששיינו את הגובה שלהם אחריה ההוכנסה, זה תמיד יהיה  $-1 \leq BF = 0 \leq 1$ , וזה המקורה שבו נמשך לטפס למעלה.
- כל עוד אני מטפס לפני מעלה, ועובד דרך צמתים ששיינו את הגובה שלהם (בעקבות ההוכנסה), מדובר על **צמתים שהיו מאוזנים והפסיקו להיות כאלה**. לכן, כמות הרמות צריכה לעלות לפני מעלה, לא **גדולה מכמות הצמתים המאווזנים בעץ לפני הפנוי**.
- בנוסף - לאחר ההוכנסה של צומת חדש, ההוכנסה יכולה לתרום עד 2 צמתים מאוזנים, סה"כ 3 צמתים חדשים לכל היוטר בהוכנסה.

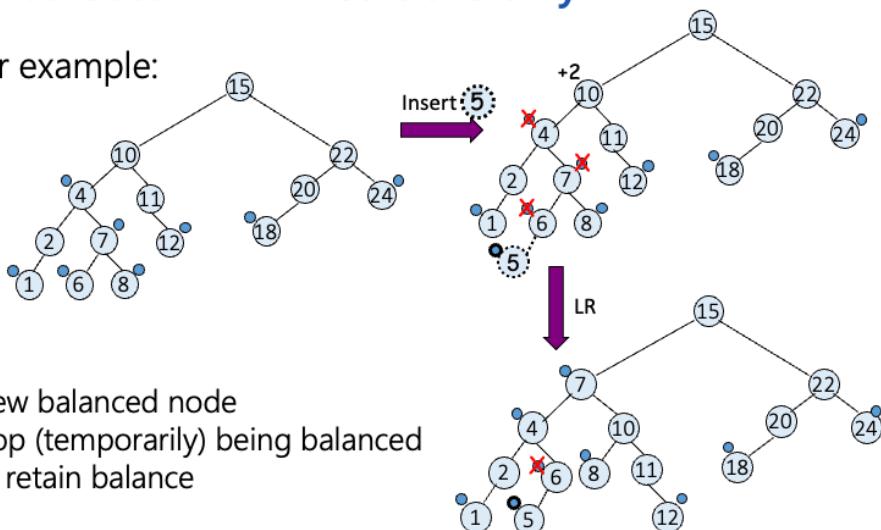
**נכחים באמצעות שיטת הבנק:**

- אנו רוצים לדאוג **שכל צומת מאוזן יהיה מטבע**, כדי שאחר כך בשתיה פעולה יקרה של טיפוס בעץ, יוכל לקחת את המטבע ולהשתמש בו.
- נשים מטבע 1 על כל צומת שמקיים  $BF = 0$  (מאוזן). בזמן הרכנסה – בכל פעם שנוצר צומת מאוזן נפקיד עוד מטבע 1, ואם צומת מאוזן מפסיק להיות בזה, נמשור 1.
- הצומת החדש שהווסף הוא בהכרח מאוזן (עליה), ולכן **נפקיד מטבע 1**.
- געלה כלפי מעלה דרך צמתים ששינו את הגובה שלהם, הם צמתים שהפסיקו להיות מאוזנים. אנו **מבדים מטבע 1** לכל רמה (המטבע הזה ממן לנו את העלייה כלפי מעלה).
- הגענו לעברין AVL ועשינו רוטציה. לכל היוטר שני צמתים חדשים מאוזנים (**נפקיד 2 מטבעות**).
- הגענו לצומת שלא שינה את הגובה שלו. זה צומת שהפרק להיות מאוזן (**נפקיד מטבע 1**).

קיבלנו שפעולות האיזון קונה עד מספר קבוע של מטבעות (3) וכך (1) 0.

## Amortized Cost – AVL: insertions only

Another example:



הערה: באופן שקול, בשיטת הפוטנציאלי נגידיר את הפוטנציאלי להיות מספר הצמתים המאוזנים.

**שימושים אפשריים:**

- 1) **נכניות את המפתחות  $a, \dots, 1$  לעץ AVL בסדר עולה.** ביוון שככל צומת חדש שמוכניסים גדול מכל הצמתים שקיים בעץ בעת ההכנסה, נשמר מצביע לצומת האחרון שהוכנס, ופושט נכניות את הצומת החדש בבנו הימני. בוצרה כזו אנו חוסכים את שלב חישוב מקום ההכנסה (עלות  $O(1)$ ). לגבי עלות איזון העץ – החסם הטריזיאלי הוא  $(n \log n)O$ , עם זאת ראים כי עלות האיזון לאחר הכנסה היא  $O(1)$ . לפיכך עלות כל הסדרה תהיה  $O(n)$ .

טענה 2: גם במקרה של סדרה של הכנסות, ואחריה סדרה של מחיקות. עלות האיזון היא  $O(1)$  amortized.

**מקרה נוספת:** אם יש סדרה מעורבת של הכנסות ומחיקות (בלי הפרדה), באופן הסיבוכיות amortized הוא  $\Omega(\log n)$  – הוכחה לה

אפשר למצוא בתרגיל מודרך בפרק 13.4 בקורס. אפשר לשפר את זה באמצעות מבנה של WAVL (הרחבה של עץ AVL).

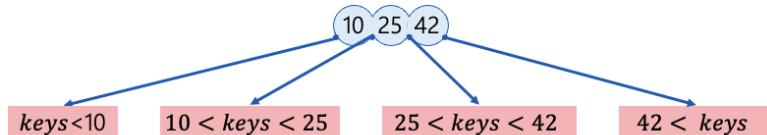


B יניע

מוטיבציה והגדלה

מדובר בסוג נוסף של עצי חיפוש מאוזנים, התחמכים בכל פעולה המילון בסיבוכיות  $\log(n)$ . עם זאת, מבגרו האיזון שלהם שונה, ואין כל גלגולים כלל. במקומות זאת, פעולות התקoon יהיו **פיצול** ו**איחוד** של צמתים, וכן **"תרומת"** מפתחות מצמתים אחרים. עץ B שימושיים מאוד במדוער כמויות הגישות ל ליכרין | בעת ביצוע הפעולות השונות.

- אנו נרצה להרחיב את העצם שלנו (יוטר מ-2 בנים לכל היותר), צמתים יותר רחבים וע"ז יותר שטוח. כל צומת יוכל למסוף מפתחות, ומצביעים לבנים. נבלה יותר זמן בכל צומת, אבל נקבל פחות cache misses/page faults שיקרו בעקבות מצביעים שציריך לעקוב אחריהם.
  - למשל אם יהיו **3** מפתחות, יהיו **4** בנים (**מספר המפתחות ועוד אחד**). נחשב את כמות המפתחות לפי גודל בלוק בזיכרון (על מנת ליעיל את כמות הגישהות לזכרון).



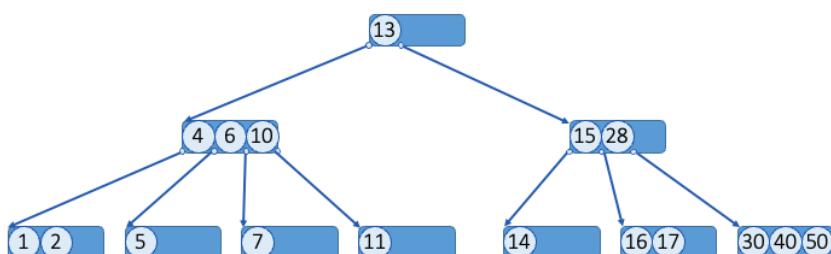
- נציין כי **דרגה של צומת** – מספר הבנים של הצומת, כלומר מספר המפתחות + 1.

**עץ B** – נגדיר בעת עץ B, הנקרא גם עץ (d, 2d)

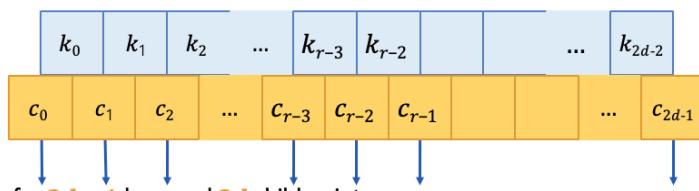
- **לכל צומת עם  $k$  מפתחות שאיום עליה יש בדיקות  $k + 1$  יlid'ים.**
  - **בכל העלים הם באותה רמה/עומק. העץ הוא מאוזן לחלווטין.**
  - **מספר המפתחות בשורש R הוא  $1 \leq R \leq 2d$  ( $\text{יש לו בין } 2 \text{ ל-} 2d \text{ בנים, אם הוא לא עלה.}$ )**
  - **בכל צומת שאים עליה/שורש יש בין  $p$  ל- $2d$  בנים.**
  - **בכל צומת יהיו  $z$  מפתחות, באשר  $1 - 2d \leq r \leq 1$ . בנות המפתחות היא  $(d)0$ . המפתחות ממוקמים!**
  - **d – הדרגה המינימלית של צומת (מספר הבנים המינימלי של כל צומת).**

למשל עבור גע (2,4) בבל צומת [1,3] מפותחת, בבל צומת [2,4] בבלים, מספר המפותחות בשורש הוא [1,3].

B-Tree with minimal degree  $d = 2$ :



**מבנה הצומת** – בכל צומת יש מערך של מפתחות (עד  $1 - 2d$ ). יש לנו מערך של מצביעים (עד  $2d$ ). **ז הוא הדרגה בפועל של הצומת** (כמה הבנים האקטואלית).



- Room for  $2d - 1$  keys and  $2d$  child pointers
  - $r$  - the actual degree
  - $k[0], \dots, k[r - 2]$  - the keys
  - $item[0], \dots, item[r - 2]$  - the associated items (not shown)
  - $c[0], \dots, c[r - 1]$  - the children

Possibly a different representation for leaves



### נכיה שעץ חיפוש B מדרגה p הוא עצ חיפוש מאוזן:

- יש לו שורש אחד בעץ תמייד, ובעומק 1 (בניים של השורש) יש לפחות 2.
- בעומק 2 יש לפחות  $d^2$  צמתים. לכל אחד משני הבנים של השורש, יש במינימום d בניים.
- בעומק h יש לפחות  $2d^{h-1}$  צמתים (עליהם).
- בנעם, אם מספר המפתחות בעץ הוא t, ובכל עלה יש מינימום  $1 - d$  מפתחות, נקבל:  

$$n \geq (d-1)2d^{h-1} \geq d^h \Leftrightarrow h \leq \log_d n$$

כלומר הגובה של עצ B עם פרמטר p ובעל n מפתחות חסום ע"י  $n.O(\log_d n)$ .

### **חיפוש בעץ B**

נרצה לחפש מפתח k בצומת x.

- לשם כך הגדemo את Node-Search(x,k) אשר מחזירה את האינדקס i של המפתח המתאים, ובנוסף גם בוליאני found שמשמעותו אם x רלוונטי. אם קיבלנו False,ណע באיזה מהילדים של הצומת הנוכחי להמשיך את החיפוש. זה מתבצע ב-
- (d) O בחיפוש רגיל.
- הfonקציה הראשית B-Tree-Search(x,k) אשר מתחילה מהשורש, ועד שהוא מוצאת את המפתח, ממשיכהليلך המנתאים ושם מחפשות שוב. גובה העץ הוא  $\log_d n$  וכן זה  $O(\log_d n)$ .

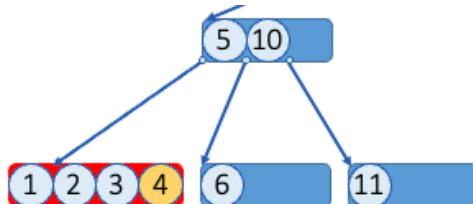
ניתוח סיבוכיות – מספר הgeshotות לזכור הוא  $n.O(\log_d n)$ , שזה גובה העץ. מספר הפעולות (לא רק גישות לזכור אלא גם המעבר על המפתחות)  $n \cdot \log_d n$ . אם בוצע חיפוש ביןארי נקבל:  $n.O(\log_2 d \cdot \log_d n) = n.O(\log_2 d)$ .

### **הכנסה לעץ B (פיצול)**

#### אלגוריתם קע-Down:

כאשר בוצע הכנסה, גם כאן יצבעו תיוקנים אך יותר פשוטים מאשר בAVL. האלגוריתם הזה נקרא קע-Down (ירודים למטה כדי להכניס, ואז עולים ומתקנים).

- אנו מוחפשים ומוצאים את המקום הרלוונטי להכנסה מפתח (לפי תכונת עצ חיפוש). **תמייד הוא נכנס בתור עליה.** מקרה שצריך לטפל בו הוא צומת "גולש" (overflowing), כאשר הכנסנו אליו מפתח שגרם לנו שבערנו את כמה מפתחות המksamילית בצומת -  $1 - d$ . דחפנו מפתח חדש וכעת יש  $2d$  מפתחות, עילנו לתקן.



**Insert(T,4)**

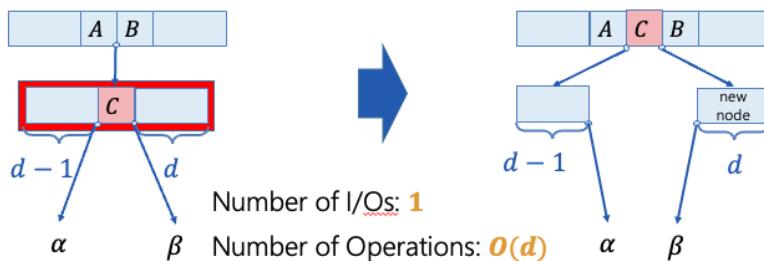
- נבחר את המפתח החיצון, הוא יצטרף לאבא שלו. נפצל את קבוצת המפתחות לאלו שגדולים מהחיצון, ולאלו שקטנים מהחיצון. בעת מהבא יהיו שני מצבים, למפתחות הגדולים ולמפתחות הקטנים.



**Insert(T,4)**

**Insert(T,4)**

- אם בעת האבא "גולש", בוצע עבוריו את אותו הדבר בדיקוק.
- במקרה של פיצול השורש – כיוון שאין אבא לשורש, נפתח רמה חדשה. העץ הפר להיות גובה יותר ב-1, השורש נמצא בrama חדשה אחת למעלה.



בנוגע לעץ AVL שהעץ גדול ממלטת, כאן העץ גדול ממלטת ולכן תמיד נשאר מאוזן. אם השורש מתפצל קיבלנו רמה אחת חדשה למעלה.

#### סיבוכיות:

- **גישה לזכרון – יש רק אחת.** בשביל צומת חדש צריך להקצתו, ולהעתיק אליו  $d$  מפתחות ו- $1+d$  פוינטרים. לצמתים שפוצלו יש הרבה מקום פנוי, יקח זמן עד שנצטרך לפצל שוב. **יש לצומת המפוץל הרבה זמן לנוח.**
  - **מספר הפעולות -  $(d)O$ .** צריכים להעתיק את כל הצמתים (מיינון או משMAL לחץון), זה לוקח  $(d)O$ . את הצומת שעלה למעלה אל האבא, עלינו להכניס בין המפתחות. במקרה הגורע זה  $(d)O$  להבנה ההז.
- מספר הגישות לזכרון הוא  $(n \log_d)O$ , שזה גובה העץ. מספר הפעולות  $n \cdot \log_d \cdot (d)O$ . כמו בחיפוש.

#### חרונונות לשיטה קע-א-עמ:

1. צריך קודם לסרוק למטה, ואז למעלה.
2. צריכים לשמר את ההורם של הצמתים על מחסנית (נגד מחסנית הרקורסיבית). כדי לחזור אליהם בהמשך.
  - a. מאד לא כדאי לשמר פוינטרים להורם, יכולה להיווצר בעיה בעת פיצול.
3. יש צמתים שהם באפין זמני "מרחפים". צריך להזיז אותם למקום אחר.

#### אלגוריתם מdown-Top:

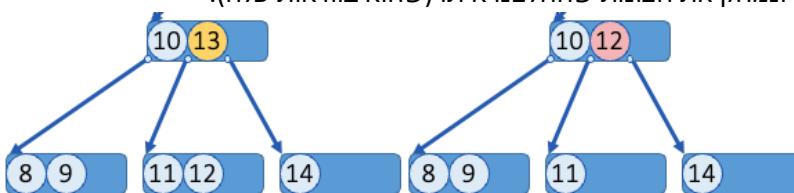
בשיטת זו אנו מבצעים פיצול לפני שאנחנו יודים לצומת מסוים. אם אנו מזדים לצומת מלא ( $1 - 2d$  מפתחות,  $2d$  בנים) (צומת בסכנת פיצול עתידית), נקדמים תרופה למבה ונפצל אותו. למרות, שאולי לא היינו צריכים באמת לפצל אותו בשיטה הקודמת. אךណ שבחזמת האחרון שנגיעה אליו, יוכל להכניס את המפתח ללא דאגה שהיא צורך בפיצול נוספת. חסרון - אנו עלולים לפצל צמתים שבධיעבד לא היינו צריכים לפצל אותם.

**מספר הפיצולים:** נניח שאנו מכינים  $t$  מפתחות אחד אחרי השני לעץ  $(d, 2d)$  ריק. ב-worst case נדרש לעשות  $\frac{n}{d-1}$  פיצולים בכל המסלול, כלומר  $(n \log_d)O$  פיצולים בכל הרכבת. סה"כ  $(n \log_d)O$  פיצולים. **נטען שבפועל נדרש רק  $\frac{n}{d-1}$  פיצולים.** ההסבר – בעץ זה יש לכל היותר  $\frac{n}{d-1}$  צמתים כי ככל צומת יש לפחות  $1 - d$  מפתחות.

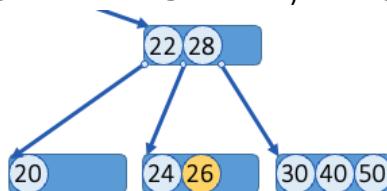
#### מחיקה מעץ B (תרומה/איחוד)

#### אלגוריתם קע-א-עמ:

- מחפשים את המפתח למחיקה.
- אם המפתח לא נמצא בעלה, נעשה משחה לדומה למחיקה של צומת עם 2 בנים בעץ AVL. נמצא את העוקב/הקודם של המפתח. נחליף ביניהם ונמחק את הצומת שהחלפנו אותו (שהוא בוודאות עלה).

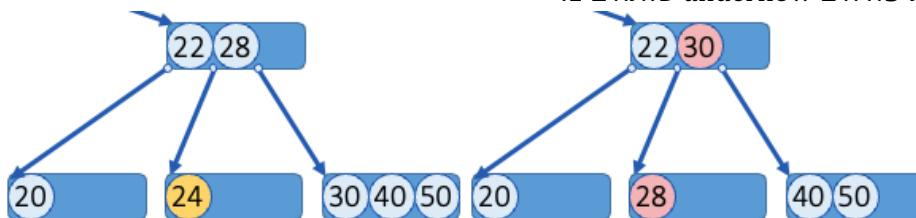


- אם המפתח נמצא בעלה, אפשר פשוט להסיר אותו (ולהזיז את שאר המפתחות שמאליה בהתאם).

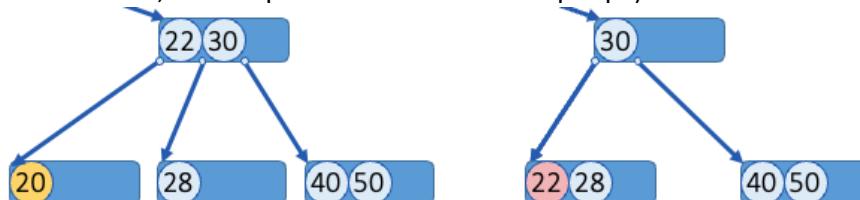




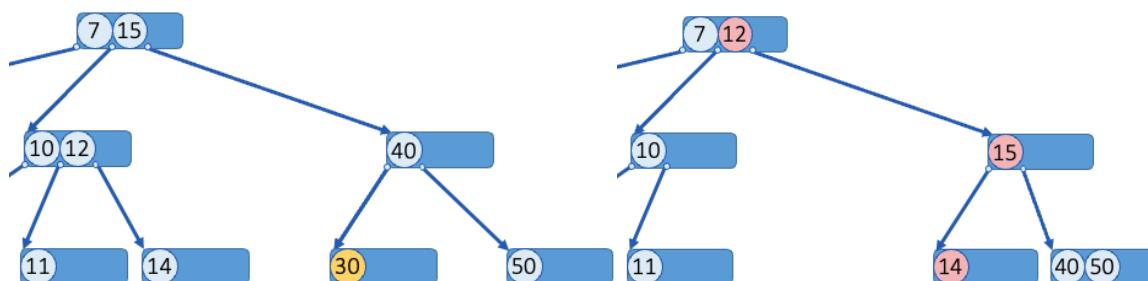
- אם מדובר בפתחה היחיד עליה, לאחר המבוקש יהיה **underflow**, מעט מידי מפתחות (או באופן כללי, פחות מ- $d - 1$  מפתחות). במקרה זה, לצומת יש את, שמננו אפשר **להשאל מפתח**. זו תקרא פועלת **borrow** (למרות שלא מדובר אחר בר). דבר זה גורם לפתחה "העשיר" לעלות לאבא המשותף, ולפתחה מהבא לרדמת אל. במודות מהדרים אחר בר).



- אם לצומת אין **אך** עם מפתחות זמינים, נבצע פעולה **fuse** (היתוך/איחוד) של שני צמתים. בפועל כדי לעשות זאת, אנו בוריד מפתח מהבא לצומת המאוחד (אין צורך בזיהוי הבדיל בין שני בנים, אנו מאחדים אותם).

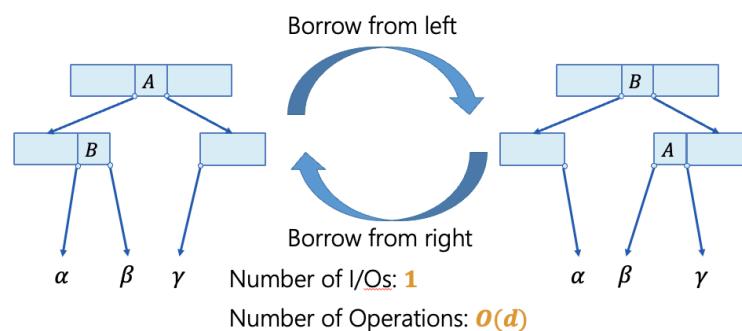


- **יכול להיווצר בעיה בהורה** – שהוא נהיה ריק (הבעיה עלתה כלפי מעלה). גם בזה צריך לטפל.

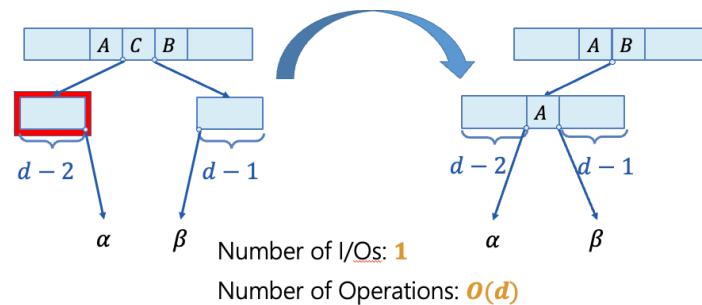


באופן יותר כללי:

### Borrowing



### Fuse



אלגוריתם Top-Down

בירידה כלפי מטה, נdag לפנינו שירודים לצומת מסוים, שהוא יכול לפחות  $d$  מפתחות. גם כאן אנו מקדימים תרופה למכה – אם יש צומת שמכיל מעט מדי מפתחות (אויל בהמשך הוא יאלץ להתאחד עם אחר או לקבל תרומות של מפתחות), נבצע borrow/fuse. בצומת האחרון שנרד אליו, יהיו לפחות  $d$  מפתחות, ואז יוכל פשטוט להוריד צומת אחד, ונגיע ל- $1 - d$  מפתחות.

מספר הפיצולים:

## Number of fuse/split (With bottom-up Insert/Delete)

The number of **split** and **fuse** operations in a sequence of  $m$  insert and delete operations on an initially empty  $(d, 2d)$ -tree is at most  $O(m)$

Amortized no. of **splits/fuses** per update is  **$O(1)$**

$$\Phi(\text{4-node}) = 2 \quad \Phi(\text{2-node}) = 1$$

With *top-down* insertions and deletions, the amortized number of splits/fuses may be  $\Omega(\log_d n)$

**B-Trees Revisited**

כאשר אנו מבצעים הכנסה/מחיקה לעץ B:

- חיפוש המיקום – לוקח  $(\log n)O$ .
- עליה למעלה כדי לתקן את העץ (כולל split, borrow, fuse) – לוקח  $(\log n)O$ . נרצה להראות שהחלק הזה דורש רק  $O(1)O$ .
- האינטואיציה היא שרוב הפעולות דורשות מעט תיקונים. ברגע שצומת פוצל, ייקח עוד הרבה זמן עד שהוא יופצל שוב פעמי.

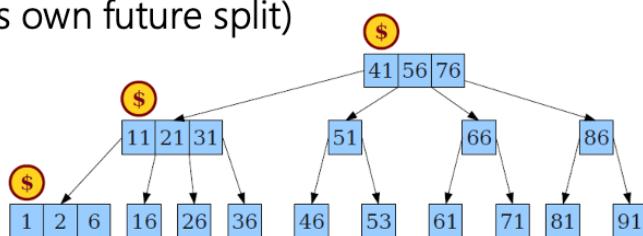
**טענה 1:** בכל רצף של פעולות הכנסה בלבד (לא מחיקה) לעץ B, עלות amortized של התקון היא  **$(1)O$** .

- זה נכון עבור כל עץ  $(a, b)$  כאשר  $2a \geq b$ .
- זה נכון רק לפעולות up-bottom. עבור פעולה top-down ניתן להיות עדין יכול להיות  $\Omega(\log_d n)$ .

נכחות באמצעות שיטת הבנק:

- נשים מטבע 1 אקסטרה לכל צומת שהוא מלא (צומת שיש בו כרגע  $1 - 2d$  מפתחות), ואי אפשר להכניס אליו עוד מפתח מוביל לגרור פיצול. בכל פעם שנוצר כזה צומת, **עלות הכנסה שיצרה אותו תהיה עוד 1 על הצומת הזה**. כלומר, בהכנסות מסוימות יווצרו צמתים מלאים ואז יתמלאו מטבעות בבנק (כדי למן **עלות split בהמשך**).
- כאשר תגיע הכנסה שגורמת ל-overflow בצומת, יש עליו מطبع שטממן את פעולה split (וכך גם עברוABA, שאם צריך לפצל אותו עבשוי, זה אומר שהוא מלא קודם והוא עליו מטבע).

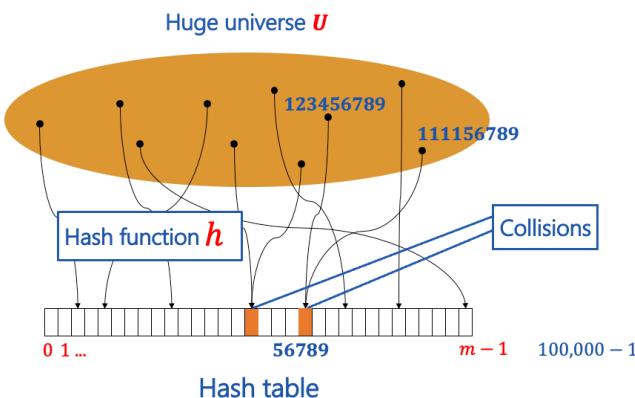
- **Proof:** place a token on each  $2d$ -node (=full) when it is formed (to pay for its own future split)



**טענה 2:** גם במקרה של סדרה מעורבת של הכנסות ומחיקות, עלות האיזון היא  **$O(1)$**  amortized.



## טבלאות Hash



רקע

כל הפעולות מתבצעות בסיבוכיות  $O(1)$ , אבל עם מחיר מסוים: הסיבוכיות הניל תושג רק ב"מקרה הממצוע" ואילו במקרה הגורר, סיבוכיות הפעולות תהיה הרבה פחות מרשימה. באופן יותר פורמלי, אנחנו ננתח את **תוחלת זמן הריצה**. כמו כן, כדי להציג תוצאה זו, נוטר על תמייבה ועילה בעלות כמות מציאות מינימום ומקסימום, ומיציאת עוקב וקדום של איבר נתון. כאשר לא מוגדר סדר מלא על קבוצת המפתחות הרלוונטיות, פעולות אלו מוגדרות לא אפשריות, ואז אין בערך נזק בלהשו.

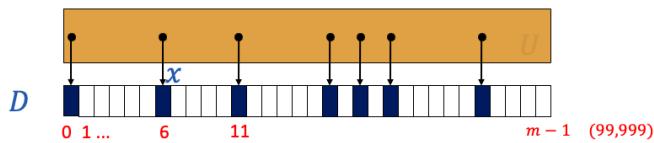
מושיבציה:

- נרצה לתחזק Dictionary עם סט מצומצם של פעולות: Insert, Find, Delete (ללא עוקב/קודם, מינימום/מקסימום). לעומת Dictionary, יתבצע  $O(1)$  בוחולת ( $O(log n)$ , כאן נקבל  $O(1)$  בוחולת).
- עצי AVL בהם סיבוכיות המקרה הגורע היא  $O(n)$ , אך נובן לטפל בכל בעיות שטיפלנו בהן בעזרת עצים מאוזנים, אבל בילגוט בפועלות הדורשות סדר. בנוסף, נוכל לטפל גם בעיות שבנה עצים מאוזנים לא תומכים. אלו הן בעיות שבנה המפתחות לקווים מתחום שאין סדר: למשל תחום מפתחות שכולג גם מספרים טבעיים וגם מחרוזות.

תחום מפתחות קטן – Direct Addressing

מקרה פרטי – תחום המפתחות קטן:  $\{0, \dots, n-1\} = U$  כאשר  $0 = U$  במשמעות  $U = \{0, \dots, n-1\}$ . נוכל לייצג את המילון באמצעות מערך שארכו  $n$  והוא  $|U|$ . שיטה זו נקראת **Direct Addressing**. כל אחת מהפעולות דורשת גישה לאינדקס שהאינו מוגדר המפתח. ביוון שכל מפתח מוגדר לאינדקס יחיד, ובכל אינדקס נמצא מפתח יחיד – כל אחת מהפעולות תתבצע בזמן  $O(1)$ .

### Direct Addressing



$D$ : info. 9 digits ID (integer), grade (real), pointer to record  
 $Insert(D, x) : D[x.key] \leftarrow x$       Special case: Sets.  
 $Find(D, k) : return D[k]$        $D$  is a bit vector  
 $Delete(D, k) : D[k] \leftarrow null$

$O(1)$  time per operation

(assume different items have different keys)

תחום מפתחות גדול – Hash Table

- מה נעשה כאשר  $n \gg |U|$ ? כדי לבצע direct addressing יש צורך בזיכרון שגודלו זהה לגודל המרחב, דבר שידרש זיכרון רב. גם אתחול מערך זה דורש עלות זמן גבוהה יחסית לכמות הסטודנטים. רוב הזיכרון שנתקaza לטובת המערך כלל לא יהיה מונצל.
- עדין נשטמש בזיכרון מגודל יותר שיהיה בסדר הגודל של  $n$  ונמפה את האלמנטים לתוך המערך הזה **באמצעות hash function**.
- ככל להשתמש רק בחמש הספרות האחרונות של תעודה זהה (ולא כל-ה-9). הבעה שנוצרת במבנה זהה היא במובן התנגשויות. עשויות להיות שתי תעודות זהות שייפולו באותו התא.
- שני האתגרים המרכזים:
  - כיצד לבחור פונקציית hash טובה? נשאף לפחות "אחד" במרחב ובلتוי תלוי בין המפתחות. נרצה שהפונקציה תהיה בצורה מכובצת (לא תיקח הרבה מקום), ושהיחס הפונקציה יתבצע ביעילות ובהירות.
  - יש  $|U|=m$  פונקציות אפשריות למיפוי את המרחב  $|U|$  לתוךם  $\{0, \dots, m-1\}$ .
  - כיצד נפתרו את ההתנגשויות?

בחירה פונקציית hash טובה:

- רעיון 1: להשתמש בפונקציה קבועה. זה בעיתי כי עבור מפתחות מסוימים יכולות לקרות תמיד התנגשויות, ו מבחינה אבטחתית.



- רעיון 2: נבהיר פונקציה אקרואית מהתחום  $[m] \rightarrow U \in h$ , באתחול מבנה הנתונים ונשתמש בה בהמשך (אנו זוקקים לה כדי למצוא מפתחות). הפונקציה לא תהיה מצומצמת, נצרך  $(|U|^m) \log m$  ביטים. בעיה נוספת היא שאפשר ליפול באקראי על פונקציה מאוד גורעה שתגרום להרבה התנגשויות.
- **בינתיים נניח שיש לנו פונקציה אידיאלית – עונה על הדרישות שציינו קודם.**

בעת נכיר שתי גישות להتمודדות עם התנגשויות, וננתח במדוק את תוחלת זמן החיפוש בטבלה תחת כל אחת.

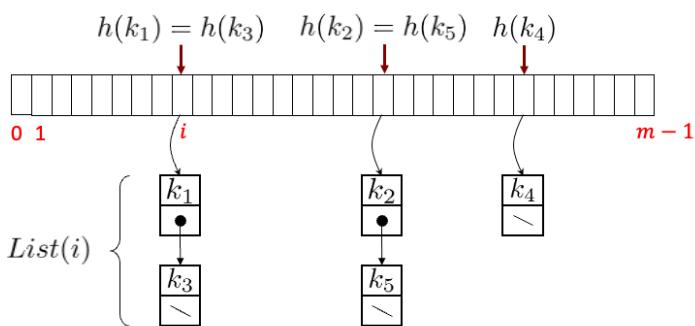
## התנגשויות – Chaining

**פתרון:** נסיף לטבלה ה-Hash רשימות מקשורות. **כל תא יכול מצביע לרשימה מקושרת של איברים.** אנחנו לא מחזקים את המפתחות בטבלה עצמה. במקום זה, **כל המפתחות שהוכנסו למילון ממופים לתא ספציפי**, ומוחסנים ברשימה מקושרת שהטה **זה מצביע אליה**, לשמור את ערכיו המפתחות – כי עברו תא שיש בו התנגשויות, علينا להיות מסוגלים לחפש את המפתח המופיע ברשימה המקושרת שנמצאת באותו תא.

## Hashing with chaining

[Luhn (1953)] [Dumey (1956)]

Each cell points to a [linked list](#) of items



- **Find(k)** – השתמש בפונקציית hash כדי למצוא את התא המתאים, **ומחברים אותו ברשימה המקושרת איבר-איבר.**
- **Insert(k)** – נחפש את המקום המתאים באמצעות **Find**. אם לא נמצא את המפתח המתאים **נסיף אותו לרשימה**.
- **Delete(k)** – נחפש את המקום המתאים באמצעות **Find**. אם נמצא האיבר **נמחק אותו מהרשימה המקושרת**.

### סיכום:

מה שיקבע את סיבוכיות ה-chaining יהיה אורך השרשראות (מספר האיברים בשרשראות). את זה אפשר לנתח על ידי זריקת כדורים לטור סלים (ששකולה לזריקת איברים לטור שרשראות). יש לנו  $\alpha$  כדורים שנזרקים לטור  $m$  סלים. כל הזריקות אקרואיות ובلتוי תלויות זו בזו. מתקיים:

1. **תוחלת מספר הcadors בכל סל היא  $\frac{n}{m}$**  (הסתברות ליפול בטור סל היא  $\frac{1}{m}$  ואנו לוקחים  $m$  כדורים כללו).
2. **הסתברות שני כדורים ספציפיים יתנגשו היא  $\frac{1}{m}$**  (הראשון בתא בlisho, והשני צריך ליפול לאחריו התא בסיכוי הנ"ל).

**נ notch את תוחלת זמן הריצה (Drill):** נניח שכרגע יש  $\alpha$  מפתחות בטבלה. תוחלת הזמן לחיפוש היא  $O(1 + \alpha)$  כאשר  $\alpha = \frac{n}{m}$

**Claim:** Using hashing with chaining, Assuming  $n$  keys are

distributed ideally (i.e. uniformly and independently), on table of size  $m$

The expected time complexity of a search is  $O(1 + \alpha)$ , ( $\alpha = n/m$ )

(expectation taken over the dictionary items and the insertion order)

**Proof:** We will prove that the expected number of items checked during search is

- $\alpha$  for an unsuccessful search (the key is not found)
- $\frac{\alpha}{2}$  For a successful search (the key is found)

Why do we add 1?

נקבל:

Assume that  $h$  is "ideal" $n$  – number of elements in dictionary  $D$  $m$  – size of hash table $\alpha = n/m$  – load factor

	Expected	Worst-case
Successful Search	$1 + \frac{\alpha}{2}$	$n$
Unsuccessful Search	$1 + \alpha$	$n$

\* Insert and Delete can cost  $O(1)$  W.C. not including search time if needed

לסיכום שיטה זו:

**Pros:**

- Simple to implement (and analyze)
- if  $\alpha = O(1)$  average constant time per operation ( $O(1 + \alpha)$ )
- Fairly insensitive to table size ( $m$ )
- Simple hash functions suffice

**Cons:**

- Space wasted on pointers
- Dynamic allocations required
- Many cache misses (I/O operations)

**התנגשויות – Open Addressing**

**פתרון:** המפתחות יושבנו בטבלה עצמה. כאשר מפתח שאנו רוצים להכניס מתנגש עם מפתח אחר, נחפש למפתח מקום חלופי בטבלה. אם המיקום החלופי תפוס, נಲк למקומות חלופי וכך הלאה עד שנמצא מקום פנוי. אנחנו מניחים שיש לנו פונקציה שמתאימה לכל מפתח ולכל אינדקס התנגשות מקום בטבלה.  $m$  המיקומות האלו הם פרטוטזיה על  $m$ .

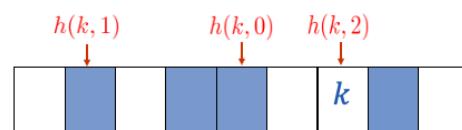
- כל מפתח מקבל סדרה של  $m$  מיפויים לטבלה – לכל מפתח יש  $m$  מקומות שאליهم הוא יכול להיכנס. מדובר במיפויים שונים, וכן זו פרטוטזיה על  $m$ . כך למשל ניתן להבטיח שגם אם נותר מקום יחיד פנוי בטבלה, המפתח יוכל למצוא אליו בסופו של דבר.
- פונקציית  $h$ -hash מקבלת זוג סגור – הפונקציה מקבלת את המפתח, ואת האינדקס הנוכחי בסדרת המיפויים (0 אם זה המיפוי הראשון, 1 אם מדובר במיפוי השני), ומחזיקה אינדקס בטבלה.
- יש חשיבות לסדר המיפויים אותה מקבל המפתח – למשל בשanno מבצעים חיפוש של מפתח בטבלה, ביצוע החיפוש לפי הסדר של המיפויים מאפשר לנו לעוזר באשר אנו מגיעים לתא ריק.

## Hashing without pointers

Assume that  $h : U \times [m] \rightarrow [m]$ 

$$\underbrace{h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1)}$$
Assumed to be a permutation of  $[m]$ Insert key  $k$  in the first free position in the listNo room found  $\rightarrow$  Table is full

To search, follow the same order

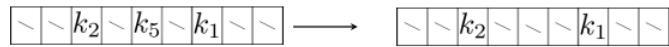


איך נממש את Delete?

- הפתרון הפשוט הוא לסמן במקומות הרלוונטיים null. הפתרון זהה בעייתי – כדי לבצע חיפוש לאיבר שכן נמצא בטבלה, יכול להיות שנספיק את החיפוש בזורה לא נכון כי סימנו null במקומות שאנו עוברים בו במהלך החיפוש.
- פתרון נוסף הוא להחליף את null בסימן X (deleted) – סימון זה אומר **שכאשר נחפש, נציג על תא זהה**. בשנורצתה להשתמש בתא זהה, נוכל להכניס מפתח חדש אליו (הוא יחשב פנוי). אמם, הבעיה בכך היא מבחינת ביצועים. עלולים להיווצר בטבלה מצבים בהם נבעור על פניו תאים רבים המסומנים X לפני שנמצא את המפתח המבוקש (או נגיע ל-null).

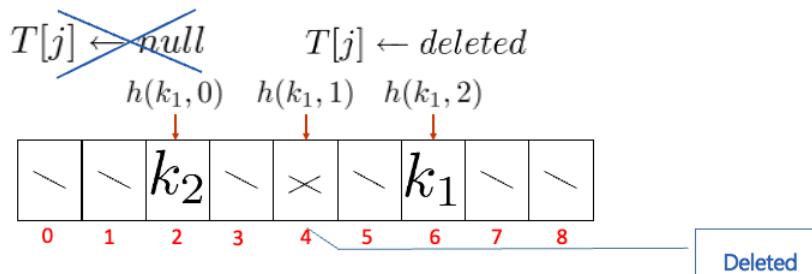
## How do we delete elements?

Simple?



Set the corresponding cells to null? Don't!

Why? Leaving a "null" will cause a search to stop at the null, possibly incorrectly

סיכום:

נניח שיש  $m$  מפתחות במיילון, כאשר גודל הטבלה הוא  $n$ , ופקטור העומס הוא  $\alpha = \frac{n}{m}$ . **נניח תנאים אידיאליים: שעבי ה-hash הם אקראיים לכל מפתח  $k$ , כך שכל פרימוטציה של ערכי ה-hash שהוא קיבל אקראית.** נראה כי תוחלת זמן החיפוש היא:

$$\text{עבור מפתח שלא נמצא בטבלה (חיפוש לא מוצלח)}: \frac{1}{1-\alpha}.$$

נוכיח בעזרת נוסחת הזנב. נחשב את ההסתברות שמספר הניסיונות (probes) גדול או שווה  $i$ . זה המאורע שבו  $i$  התאים הראשונים שבהם אנו מחפשים מקום למפתח **תפוסים**.

**Claim:** Expected no. of probes for an **unsuccessful** search is at most:  $\frac{1}{1-\alpha}$

**Proof:**  $E[\#\text{probes}] = \sum_{i=0}^m i \cdot \Pr[\#\text{probes} = i] = \sum_{i=0}^m \Pr[\#\text{probes} \geq i]$  Tail formula for non-negative discrete random variables

$\Pr[\#\text{probes} \geq i] =$

$$\Pr[\text{the first } i \text{ cells probed are all occupied}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \dots \cdot \frac{n-i+1}{m-i+1} \leq \alpha^i.$$

$$E[\#\text{probes}] \leq 1 + \alpha + \alpha^2 + \dots = \frac{1}{1-\alpha}$$

$$\text{עבור מפתח שנמצא בטבלה (חיפוש מוצלח)}: \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right).$$

נראה לחשב את תוחלת מספר הניסיונות (probes) של חיפוש מוצלח בהינתן שאנחנו מחפשים את המפתח  $-i$ .

זמן החיפוש שווה בדיקות **לזמן ההכנסה של המפתח** למבנה. הזמן שלקח להכניס אותו לממבנה, הוא בדיקות הזמן שלקח לעשותות

חיפוש לא מוצלח בשטבלה היו כל המפתחות שלפניו. את הזמן הזה אנו יודעים לחשב לפי חיפוש לא מוצלח:  $\frac{1}{1-\alpha_i}$  כאשר  $i-1 = \frac{i-1}{m} \alpha_i$ . קיבלנו תוחלת מותנה במפתח  $-i$ .

**נוריד את התנינה במפתח  $-i$** , ההסתברות שהמפתח הוא המפתח  $-i$  הוא  $\frac{1}{n}$ . נסכם על  $n$  המפתחות ונקבל:

$$E = \frac{1}{n} \sum_{i=1}^n \frac{1}{1-\alpha_i}$$

o مكان נשarraה עבודה אלגברית:

$$E[\# \text{ probes in successful search}] =$$

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-i/m} &\approx \quad \frac{1}{n} \int_{i=0}^n \frac{1}{1-i/m} di = \\ \frac{m}{n} \int_{i=0}^n \frac{1}{m-i} di &= \frac{1}{\alpha} \ln\left(\frac{m}{m-n}\right) = \\ \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

### Linear Probing

קודם ניתחנו את hash-open addressing probing, והנחנו שיטתה היא אידיאלית (לטובת הניתוח של הביצועים). השיטה זו אינה פרקטית. השאלה היא איזה פונקציית hash נבחר בסדרה של פונקציות עבור מפתח k. זאת הסדרה  $(h(k, i))$  באשר k הוא המפתח ו-i זה האינדקס של מספר התנטשויות שחוינו (מספר המקומות התפוסים בטבלה שארינו). כל השיטות **בנייה על פונקציה בסיסית h** ששולחת למקום הראשון בטבלה, ואז בשיטות שונות, כדי הגיע מהמקום הראשון למקום אחרים, במקרה שיש התנגשות במקום הראשון.

#### נכיר מספר שיטות:

- **בדיקה לינארית (קפיצות של 1):**  $h'(k+i) mod m$ . קודם כל מגיעים ל-( $k', h(k, i)$ ). אם שם תפוס הולבים ל-(**1**) ( $k+1$ )  $h'$  וכך הלאה.
- **שיטת הריבועית:**  $h'(k+i^2) mod m$ . גודל הקפיצות שkopצים מהמקום הראשון משתנה כפונקציה של מספר התנטשויות. גודל הקפיצות הוא ריבועי.
- **שיטת ה-double hashing:**  $h'(k+i) mod m$ . לבסוף יש את הפונקציה הבסיסית שלו, אבל לבסוף מפתח מחשבים גם פונקציה שנייה ( $h_2(k)$ ) שתלו בכל מפתח k. לכן גודלי הקפיצות שחוויים המפתחות משתנים ממפתח.

#### בדיקה לינארית:

איך זה עובד? מgive מפתח  $k_1$ , מחשבים לו את הפונקציה הבסיסית  $(k_1)' h$  והוא מביאה אותו למקום מסוים בטבלה. אם המקום פנוי, נשים שם את המפתח. אם המקום תפוס, ננסה מקום 1 ימינה. אם הוא פנוי נשים שם את המפתח, אחרת ננסה שוב מקום אחד הלאה.

#### תובנות:

1. רצפים של מפתחות של מקומות תפוסים במערך נוטים להידבק זה לזה, רצפים ארוכים בטבלה נוטים לגודל (הסתברות שמספר חישוף ייפול לטור הרץ גוזלה יותר מליפול לטור רצף קצר יותר).
2. חיפושים עבור מפתח מתבצעים באחור לקליל של המפתח. אם משתמשים ב-clumping, הסיכוי שנמצא את המפתח הזה ב-cache הוא מאד גבוה. הביצועים יהיו מאוד טובים.
3. הניתוח של השיטה הוא יותר מסובך מאשר השיטה האקרואית – לא נבעז אותו.

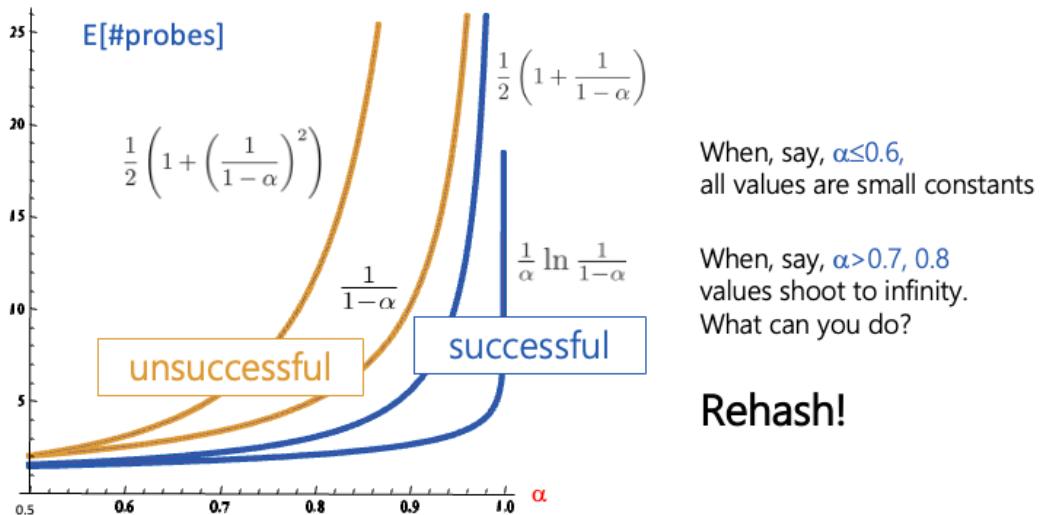
#### סיבוכיות:

נציג התוצאות: הביצועים עבור linear יותר גדולים מהסיבה שיש לנו את ה-clumping שבו אורק השרשראות יותר ארוך.

	Unsuccessful Search	Successful Search
Uniform Probing	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
Linear Probing	$\frac{1}{2} \left( 1 + \left( \frac{1}{1-\alpha} \right)^2 \right)$	$\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$



באשר מקדם העומס גדול מעל 0.7 אנחנו רואים שככל העקומות מזקקות לאינסוף, תוחלת ב모ת העבודה פר חישוב היא מאוד גדולה. מה אפשר לעשות במקרה זה? **rehash**. הטבלה צפופה, لكن ניקח טבלה יותר גדולה, למשל מערך גדול פי 2. מקדם העומס יקטן פי 2. אך נאחסן את המפתחות בטבלה החדשה? יש לנו מ"מ חדש ובעורו צורך נצטרך פונקציית hash חדשה. נדרש לבצע את פעמי **Insert**.



#### מחיקות חכמת:

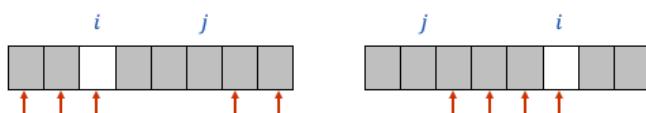
קודם אמרנו שכאלר מוחקים מפתח נסמן במקום שלו **deleted**. זהינו שכאלר מוחקים מפתח נזירים מקומות ריקים בטבלה, ובאלר נחפש מפתח נזירים מסלולים ארוכים שעוברים על פני מקומות ריקים לחינם. **את המסלולים הארוכים האלה ניתן לתקן באמצעות:**

#### **ובצע מחיקות חכמת:**

- נמחק את המפתח  $k_1$ . נסמן בתא שלו **deleted**.
- **נhapus ב מפתחות שנמצאים מינה לתא הריק – מי מהם יכול להיכנס במקום התא הריק.**
  - נסתכל על תא מס' 6 שבו מופיע מפתח  $k_2$ , ונבדוק את ערך הפונקציה 'ה' עליו – שהוא 6. כלומר חיפוש עתידי של מפתח  $k_2$  יתחל בתא 6, אז אסור לנו להזיז אותו לתא מס' 5.
  - נבדוק את מפתח  $k_3$  כאשר  $h(k_3) = h(k_2)$ . זה אומר שchiposh עתידי של מפתח  $k_3$  יתחל בתא 4. אפשר להזיז אותו לתא 5 (הchiposh יעבור עליו).
- מבחינה פורמלית השאלה היא: האם מפתח שנמצא בתא  $j$  יכול לעבור לתא  $i$  (התא הריק)? אם ערך הפונקציה של המפתח שנמצא בתא  $j$  הוא נמצא שמאלה לנקיודה הריקה, אז נבצע לעשות את ההעברה.

Can the key in cell  $j$  be moved to cell  $i$  (empty)?

$$h'(T[j]) \in [j+1, i] \text{ (function is "left" to } i\text{)}$$



$$h'(T[j]) \leq i \text{ or } h'(T[j]) > j \quad j < h'(T[j]) \leq i$$

- העברנו – וכעת נוצר תא חדש ריק. על התא זהה נבצע את אותה הפעולה מחדש.
- **מתי נעצור את התהיליך?** באשר מגעים לסופ' הצבר שבתוכו נמצא המפתח שמחקנו. כלומר, נתקלנו בתא הריק הראשון במערך החל מהאינדקס בו ביצענו מחלוקת. ביוון שההתא הזה הוא ריק, ניתן לומר בוודאות שאין אף מפתח שהובנס לטבלה וממוחפה אליו – ולכן אין צורך לבדוק את התאים שאחריו וניתן לסיים את התהיליך הבדיקה.

תוכנה מעכינת: **באשר משמשים איבר בשיטה הזאת, מצב הטבלה אחרי ההשמטה החכמה, יהיה בדיקתomo מהו שהוא אם האיבר לא היה נכנס לטבלה אי פעם.**

ישנו הסבר על **Double Hashing** ועל **Quadratic Probing** בתרגומים.

**משפחות אוניברסליות**הגדרה:

בציד בוחרים פונקציות  $h$  טובות באקראי? ניעזר **במשפחה אוניברסלית של פונקציות**:  $[m] \rightarrow U \subset H$ . כאשר אנחנו רוצים להפעיל  $h$  אנחנו נבחר באקראי פונקציה אחת מתוך משפחת הפונקציות. התכונות שנרצה שיתקיים:

- המשפחה תהיה קטנה מספיק כך שהיצוג של כל פונקציה יהיה מספיק קטן ומצוצם.
- המשפחה תהיה גדולה מספיק כך שכאשנחנו בוחרים דברים באקראי נקבל התנהגות שהיא אקראית.

המטרה – לכל רצף של פעולות שבוחר, אם נבחר פונקציה באקראי מתחן  $H$ , אז **תוחלת זמן הריצה על רצף הפעולות תהיה קטנה**.

**הגדרה:** תהי  $H$  משפחה של פונקציות:  $[m] \rightarrow U \subset H$ .  **$H$  אוניברסלית  $\Leftrightarrow$  לכל  $U$**

המשמעות של ההגדרה היא שכשלוקחים באקראי פונקציה מהמשפחה, אזו סיכוי ההתנגשות של שני מפתחות כלשהם הוא כמו בפונקציה אקראית אמיתית.

דוגמה והוכחה:

נתובן במשפחה הפונקציות המודולריות: נניח  $U = [p] = p, 1, \dots, p-1$  כאשר  $k$  הוא מספר ראשוני. נגידר פונקציית hash מודולרית כז':  $h_{a,b} : [p] \rightarrow [m]$  כאשר:

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m, \quad 1 \leq a < p, \quad 0 \leq b < p$$

• המשפחה של פונקציות מודולריות  $h_{a,b}$  מוגדרת כז':  $\{h_{a,b} \mid 1 \leq a < p, 0 \leq b < p\}$ .

• במשפחה זו ישן  $(1-p)$  פונקציות בלבד, כלומר אין כאן כיסוי מלא של  $|U|^m$  הפונקציות האפשריות.

חסרונות	יתרונות
<ul style="list-style-type: none"> <li>• ביצועים – לחישוב הפונקציה נדרשות שתי פעולות חילוקה (מודולו) מה שמאט את הביצועים.</li> </ul>	<ul style="list-style-type: none"> <li>• תמציותיות (succinct) – המקום שנדרש כדי לייצג את <math>h_{a,b}</math>, צריך לשמר רק שני שלמים ולשם כך דרישים <math>O(\log p)</math> ביטים.</li> </ul>
	<ul style="list-style-type: none"> <li>• אוניברסליות – <b>הוכחה</b>.</li> </ul>

מספר ההתנגשות הצפוי במשפחות אוניברסליות:שתי תכונות משמעותיות שמתיקיות (**הוכחה**):1. אם נבחר טבלה שמספר התאים בה  $m$  שווה למספר המפתחות שיוכנסו לטבלה זו, תוחלת מספרהתנגשות תהיה קטנה  $\frac{n}{2}$ . נוכל להסיק שבמקרה זה **בהתBURות גבוהה יש פחות מ-m התנגשות**.2. אם נבחר טבלה שמספר התאים בה  $m$  הוא ריבועי

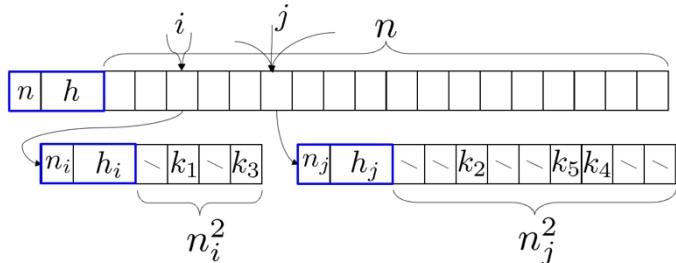
מספר האיברים שיוכנסו לטבלה זו, תוחלת מספר

התנגשות תהיה קטנה מ- $\frac{n}{2}$ . נוכל להסיקשבמקרה זה **בהתBURות גבוהה אין התנגשות כלל**.Suppose that  $|D| = n$ and that  $h$  is randomly chosen from a universal family

Collisions: Key pairs that collide with each other

Property 1: If  $m = n$ , then  $E[|Col|] < \frac{n}{2}$ Corollary 1: "if  $m = n$  most chances that  $< n$  collisions"Property 2: If  $m = n^2$ , then  $E[|Col|] < \frac{1}{2}$ Corollary 2: "if  $m = n^2$  most chances for no collisions!"  
 $n^2$  is a lot, except for small  $n$ 's (will use it soon!)

## Two level hashing



Choose  $m = n$ . Find  $h$  such that  $|col| < n$  (expected 2 trials).

Store the  $n_i$  elements hashed to each  $i$  in small hash tables of sizes  $n_i^2$  using a perfect (no collisions) hash function  $h_i$

מצבייע לטבלת משנה). החיפוש של המפתח נעשה על ידי הליכה קודם לטבלה העיקרית, ולאחריה הליכה לטבלת המשנה המותאמת, שם אמרו המפתח להימצא. על המבנה הדו-שלבי זהה אפשר לקבל את התכונות הבאות:

1. אפשר לקבל מצב שלא יהיה התנגשויות בהסתברות גבוהה.
2. זמן הבניה עד שנגיא ל-hash מושלם צה, יהיה  $O(n)$  בთוחלת.

### מבצע hashing בשני שלבים:

- נבחר את גודל הטבלה הראשית  $m$  להיות מספר האיברים  $n$ .
- נחפש פונקציית hash (נסמנה  $h$ ) שמתמחה את האיברים לטבלה הראשית ושתקיים שמספר ההתנגשויות בטבלה הראשית לא גדול. ספציפית נבחר אותה כך שתקיים  $n < |Col|$ , כלומר שגודלה קבוצת הזוגות המתנגשים תחת  $h$  קטן מ- $n$ .
- נסתכל על האיברים שמופו מקום  $i$  בטבלה הראשית, ונסמן מספרם  $-i$ . נמפה (ונאחסן) איברים אלו בטבלת hash משנית (קטנה) שגודלה הוא  $n_i^2$ . המיפוי בעזרת פונקציית hash "אישית"  $h_i$ . את הפונקציה זו נבחר כך שכל  $i$  האיברים ימופו ללא אף התנגשויות.

### אלגוריתם אקראי למימוש hash מושלם:

בහינתן קבוצה של מפתחות  $x_n, \dots, x_1$ .

1. בניית הטבלה הראשית:

- a. נבחר משפחה אוניברסלית של פונקציות hash:  $h: [n] \rightarrow U$  (למשל אפשר לקחת את  $H_{p,n}$ ).
- b. נבחר משפחה זו פונקציה באקראי:  $h \in H$ .
- c. נחשב את מספר ההתנגשויות ב- $x_n, \dots, x_1$  תחת  $h$ . בلومר את המספר הזוגות המתנגשים תחת  $h$  מתוך קבוצת האיברים. נחזור על בחירת הפונקציה עד למצב שבו יש פחות מ- $m$  התנגשויות.

כעת יש לנו טבלה בגודל  $m$  ופונקציית hash עם פחות מ- $m$  התנגשויות על קבוצת האיברים שלנו.

2. בניית טבלאות המשנה:

- a. עברו כל תא בטבלה הראשית, באינדקסים  $1 - n, 0, \dots, i = 0$ .
- b. אם  $1 > i$  נבחר משפחה אוניברסלית של פונקציות:  $h_i: [n_i^2] \rightarrow U$  (למשל אפשר לקחת את  $H_{p,n_i^2}$ ).
- c. מתוכה נבחר באקראי פונקציה  $h_i \in H_i$ .

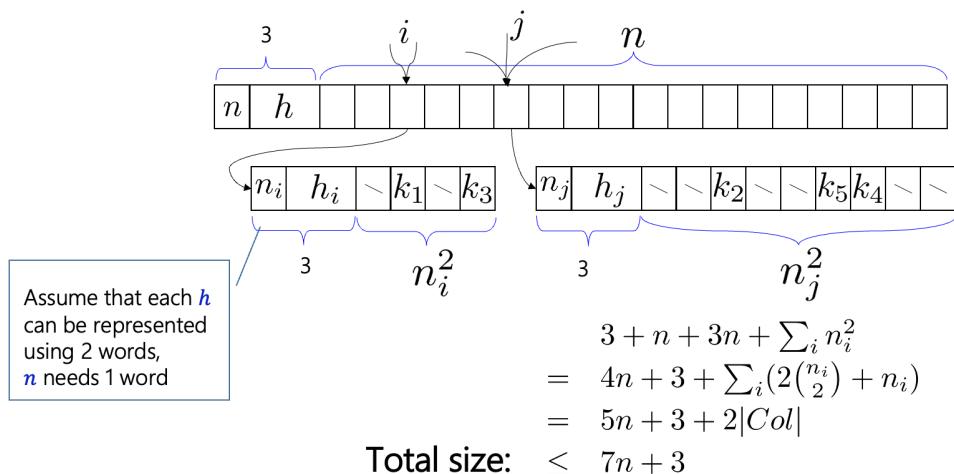
d. נחזור על בחירה אקראית של הפונקציה עד שלא יהיה בכלל התנגשויות בקרב  $i$  האיברים המומופים לתא  $i$  בטבלה הראשית.

כעת יש לנו לכל היותר  $m$  טבלאות קטנות, בשטבלה  $i$  בגודל  $n_i^2$  יש לה פונקציה אישית  $h_i$  שהיא ללא התנגשויות.

### סיכום זיכרון:

כל טבלה (בין  $1 + m$  הטבלאות) דורשת תא אחד שיציין מה גודל הטבלה. בנוסף, צריך לאחסן את הפונקציה המותאמת – ונכון שnitן לבטא אותה באמצעות שתי מילוטות מחשב. בנוסף, צריך מספור תאים לאחסן האיברים בהתאם לגודלי הטבלאות שהוגדרו קודם. לכן טבלה בגודל  $k$  תאים דורשת מקום בגודל  $3 + k$ .

לכן כמות הזיכרון הכוללת הנדרשת חסומה על ידי:



עלות הדיבורן הנדרשת על ידי הטבלה הדו-שלבית היא **ליינארית**.

סיבוכיות זמן: **תוחלת ב민ות הזמן** לבניית טבלה דו-שלבית הינה ( $n$ ) .

## Perfect hashing

Reminder:

“if  $m = n^2$ , chances  $\geq 0.5$  for no collisions!! “

Idea: Given static set of keys  $x_1, \dots, x_n$   
Use a table of size  $m = n^2$

Repeat:

- choose a random  $h \in H_{p,n^2}$  (modular)
  - compute the number of collisions on  $x_1, \dots, x_n$
- until there are **no collisions**

Expected no. of trials until no collisions  $\leq 2$  (Geometric dist.)

לסיכום:

בעזרת הטבלה הדו-שלבית ובשימוש במשפחות אוניברסליות ובהגרלת פונקציות מבינהו, אנחנו יכולים לבנות hash מושלם:

- זמן בנייה -  $O(n)$  בתוחלת.
- עלות מקום -  $O(n)$  ב-WC.
- עלות חיפוש -  $O(1)$  ב-WC.



## 3 – אלגוריתמים לפתרון בעיות

### ערימות בינהריות

#### מבוא

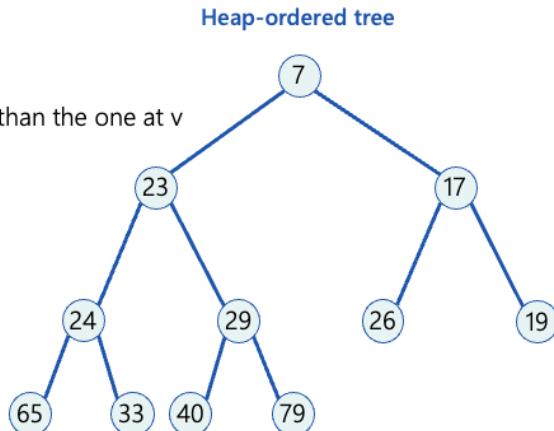
ערימה היא מבנה נתונים שמאפשר גישה מהירה לאיברים מסוימים, כמו למשל המינימום או המקסימום. בשל כך, הן שימושיות בתור מבנה נתונים יעיל ל-ADT שנקרא תור עדיפויות (Priority Queue). תור זה לאעובד בצורה FIFO (לפי סדר הגעה), אלא על פי **עדיפויות** של האיברים. על אף שהיא לא מתחכמת כמו עץ חיפוש בינהרי, פשוטותה של העירמה היא הכוח שלה.

- ערימה בינהרית (Binary Heap) היא עץ בינהרי עם ערכים בתוכו, אך בניגוד לעץ חיפוש בינהרי יש לה תכונות שונות לחולטי:
1. **תכמה מרבנית (אייר הוא בנו)** – עץ בינהרי כמעט מושלם, כלומר הרמות העליונות כולן מלאות לחלווטין, ואם יש רמה שהיא לא מלאה זו הרמה התחתונה. בדומה זו, חסר המלאות הוא רק מצד ימין, אין חורים באמצעות. לעומת זאת, הרמה מלאה מצד שמאל עד לנוקודה בלבד.
  2. **תכמה ערכית (אייר ערכים מתאימים זה לזה)** – כל צומת קטן מילדיו. אין שום יחסים בין הילדים, בין שמאל לימני.

### (Binary) Heap

#### (Binary) Min-Heap is

- An almost complete binary tree
- The items at the children of v are **greater** than the one at v



#### בוחן את היכולות של ערימה בינהרית:

- **גובה העץ** – עץ בינהרי כמעט מושלם וכך:  $H = \log n$ .
- **חיפוש בעץ** – אי אפשר למצאו ביעילות, אנו צריכים לסרוק את כל העץ כדי למצוא ערך מסוים.
- **הוספה לעץ** – נכניס לפינה ימנית תחתונה (ברמה התחתונה). אם הערך קטן ממנו (לא תקין), פשוט **נחליף אותו עם אבא שלו**. אם עדין יש בעיה, נחליף שוב עם האבא עד שכל צומת קטן מילדיו.
- **הפרחתה ערך מסוים** – נרצה להפיחת ערך של מפתח מסויים. התכמה **המבנה נשמרת**, אך התכמה **הערכית עלולה להשتبש**: המפתח של הצומת קטן, ולא ניתן להבטיח כי המפתח של הצומת גדול מהמפתח של ההורה שלו. גם כאן, **התיקון יבוצע על ידי החלפה עם האבא**, ככל שדרש.
- **הגדלת ערך מסוים** – התכמה **הערכית עם אביו**, אבל עם הילד אולי זה השتبש. **נחליף מפתח עם הילד הקטן יותר** כדי שהערכים בכל תחת-העץ יהיו תקינים.

#### תור עדיפויות (Priority Queue):

האיברים נכנסים בסדר שרירותי, אבל לאיברים יש מפתחות. בשובורים איבר להוציא מהטור, לא בוחרים את האיבר הראשון שנכנס, אלא את האיבר עם **המפתח המינימלי** (מסמל את **העדיפות הכי גבוהה**).

1. **פעולות בסיסיות:**
  - a. **Insert(x, Q)** – הכנסת האיבר x לתור.
  - b. **Min(Q)** – מציאת המינימום בתור.
  - c. **Delete-min(Q)** – הוצאת המפתח הנמוך ביותר מהטור.
2. **פעולות שביהם יש לנו גישה לאיבר x** (קשה לבצע חיפוש ביעילות וכן לא דרך חיפוש, נניח שננתונה לנו כבר גישה לאיבר):
  - a. **Decrease-key(x, Q, δ)** – הקטנת המפתח של x ב-δ.
  - b. **Delete(x, Q)** – מחיקת של איבר x מהטור.



## הגדרה

שימוש במערכות מערך:

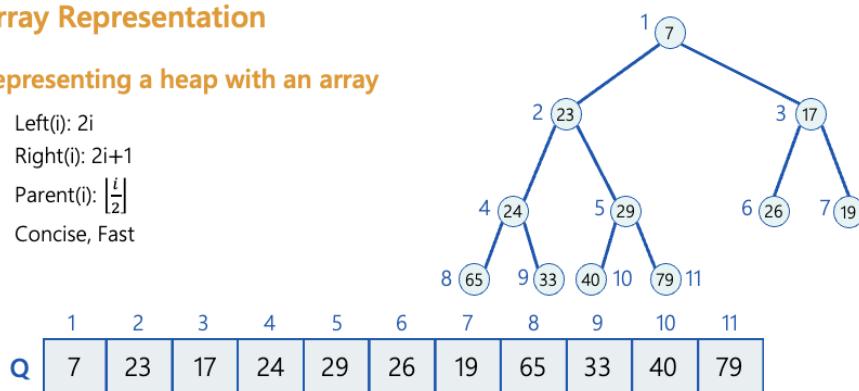
אפשר למשה בצורה פשוטה את העירמה באמצעות מערך, הוא יהיה מאדiesel. ניקח את הערכים, ונשים אותם שכבה-שכבה בתוך המערך, באשר אנו לוחקים את השכבות מלמעלה למטה, ומשמאלי לימין. קיבלנו כי:

- השורש – תמיד נמצא באיבר הראשון, והוא המינימום.
- הפינה اليمنית התחתונה – תמיד האיבר האחרון במערך.
- פירוט – נניח שאנו נמצאים בשורש (אינדקס 1 במערך), אז הינו יושם ליד השמאלי שלו והוא על ידי הכפלת האינדקס ב-2 (אינדקס 2), והינו יושם ליד הימני יהיה הכפלת האינדקס ב-2 ועוד 1 (ונגיע לאינדקס 3).
- כלומר:  $\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$ ,  $\text{Left}(i) = 2i$ ,  $\text{Right}(i) = 2i + 1$ . כדי למצוא אבא:

## Array Representation

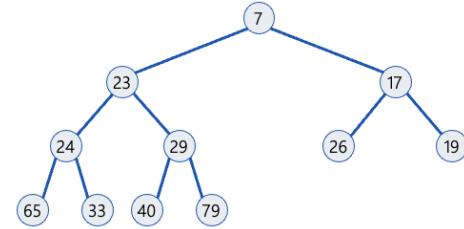
## Representing a heap with an array

- $\text{Left}(i): 2i$
- $\text{Right}(i): 2i+1$
- $\text{Parent}(i): \left\lfloor \frac{i}{2} \right\rfloor$
- Concise, Fast

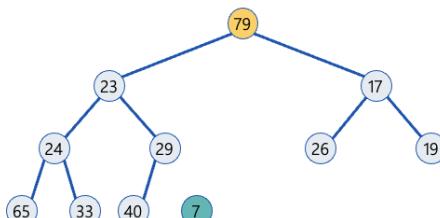


**מיון עירימה ובנייה עירימה****מיון-עירימה (HeapSort)**

באמצעות הפעולות שהגדרכנו ניתן לבצע מיון. בוצע  $\alpha$  פעמים `Insert` של איברים לתוך העירימה, זה יעלה לנו  $O(n \log n)$ . לאחר מכן נבצע  $\alpha$  פעמים `Delete-Min` ונשים את האיברים במערך אחר (שהוא יהיה המערך הממוין). סה"כ העלות היא  $O(n \log n)$ . נראה כיצד לעשות את המיוון הזה `in-place` ללא שימוש במערך נוספת:



Q	7	23	17	24	29	26	19	65	33	40	79
---	---	----	----	----	----	----	----	----	----	----	----



Q	79	23	17	24	29	26	19	65	33	40	7
---	----	----	----	----	----	----	----	----	----	----	---

Q	79	65	40	33	29	26	24	23	19	17	7
---	----	----	----	----	----	----	----	----	----	----	---

- Reverse the array, if want

Q	7	17	19	23	23	24	26	29	33	40	65	79
---	---	----	----	----	----	----	----	----	----	----	----	----

## HeapSort (Williams, Floyd, 1964)

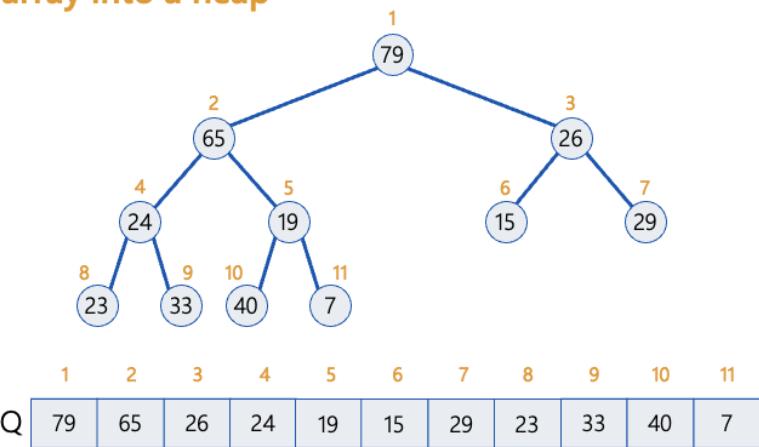
- ✓ • Put the elements in an array
- • Turn the array into a heap
- ✓ • Do delete-min, and put the deleted element at the last position of the array. Repeat N times.
- ✓ • Reverse the array, or use a max-heap

**בנייה עירימה ממערך:**

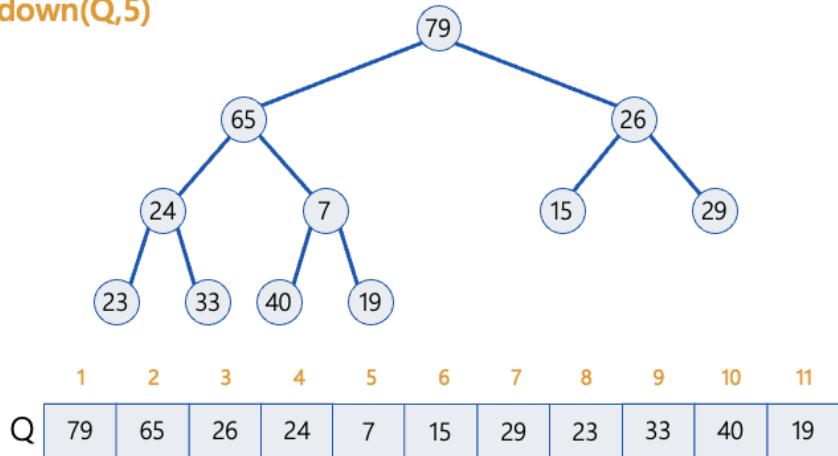
נאיבית אפשר לבצע מבון  $\alpha$  הכנסות לעירימה,  $(n \log n)$ . נרצה לבצע זאת בצורה יותר יעילה. השיטה הכללית היא לעבור על העץ, ובעזרת `Heapify-Down` לדאוג שככל העץ יהיה מסודר. נעבור רמה-רמה בעץ, כאשר **נתחיל מהرمות הנמוכות** וنعבור לرمות יותר גבוהות. כל פעם שנעבור על רמה, **נדאג שמתוחת לרמה העץ הוא במצב של עירימה תקינה**, בעזרת `Heapify-Down`.



### Turn an array into a heap



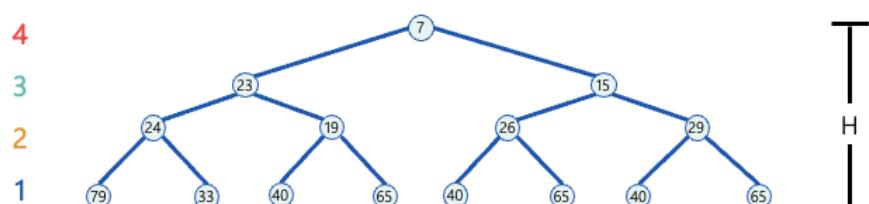
### Heapify-down(Q,5)



סיבוכיות בניית העץ: נחשב את עלויות heapify לכל צומת שעשינו לו את הפעולה זו. העלות של Heapify חסומה על ידי גובה הצומת הרלוונטי. בرمאה התחתונה הגובה הוא 0, אך העלות היא 0. בرمאה מעל הגובה חסום על ידי 1 ולבן העלות חסומה על ידי 1. כך גם בرمאה מעל וכו'. לגבי מספר הצמתים:

- בגובה 1, מספר הצמתים חסום על ידי  $\frac{n}{2}$ . נשים לב שאם בرمאה התחתונה יש רק צומת אחד, אז במתו הצמתים בرمאה שמעל היא בדיקת  $\frac{n}{2} \log n$ . אם נגדיל את מספר הצמתים בرمאה התחתונה, אז ה- $m$  גדול אבל במתו הצמתים לא משתנה. היא עדין תהיה חסומה על ידי  $\frac{n}{2}$ .
- בגובה 2, מספר הצמתים הוא בדיקת חצי ממספר הצמתים שבגובה 1 וכן חסום על ידי  $\frac{n}{4}$ .

נסקלל את העבודה זו ונקבל (הטור האינסופי מסתכם ב-2):



At most  $\frac{n}{2}$  nodes heapified at height 1

At most  $\frac{n}{4}$  nodes heapified at height 2

At most  $\frac{n}{8}$  nodes heapified at height 3

$$\text{Total time} = 1 \frac{n}{2} + 2 \frac{n}{4} + 3 \frac{n}{8} + \dots + 1H = \sum_{h=1}^H h \frac{n}{2^h} < n \sum_{h=1}^{\infty} \frac{h}{2^h} = O(n)$$



ערימות בינומיות ופיבונאצ'י

ערימה בינו מית

הכוון הכללי שלנו הוא לשפר את זמכי הריצה. בשים לב כי אנחנו נverb מניות מקורה גרעוע לכתו amortized

	<b>Binary Heaps</b>	<b>Binomial Heaps</b>	<b>Lazy Binomial Heaps</b>	<b>Fibonacci Heaps</b>
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	<b><math>O(1)</math></b>
Meld	—	$O(\log n)$	$O(1)$	$O(1)$

## עצים בינויים:

The diagram shows the construction of a binary tree  $B_4$  from subtrees  $B_0$  through  $B_3$ . The subtrees are represented as follows:

- $B_0$ : A single white circle.
- $B_1$ : Two white circles connected by a horizontal line.
- $B_2$ : Three white circles arranged in a triangle: one at the top, two below it connected by a horizontal line.
- $B_3$ : Four white circles arranged in a diamond shape: one at the top, two below it connected by a horizontal line, and two more below those connected by a horizontal line.
- $B_4$ : Five white circles arranged in a diamond shape: one at the top, two below it connected by a horizontal line, and two more below those connected by a horizontal line. This structure is highlighted with a pink oval.
- $B_0$ ,  $B_1$ ,  $B_2$ , and  $B_3$  are also shown separately below the main structure.

עדים בינוּמִים הם סדרה של עצים, מ-0 עד אינסוף. באופן הבא:  
 כיצד נוכל לקבל את  $B_4$ ? אפשר לחתך שני  $B_3$  ולחבר אותם, או לחתך שורש ולחבר אליו את כל העצים מ- $B_0$  ועד  $B_3$ . באופן כללי יותר נוכל להרכיב את  $B_k$  (**עץ בינוּמי מדרגה k**) באמצעות חיבור של שני  $B_{k-1}$  או לחתך שורש ולחבר אליו את  $B_0$  עד  $B_{k-1}$ .

## תבונות:

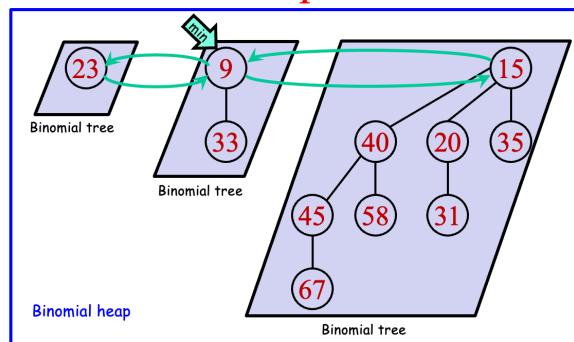
1. לשורש של עץ  $B_k$  יש  $k$  ילדים.
  2. עץ  $B_k$  מכיל  $2^k$  צמתים והעומק שלו הוא  $k$ .
  3. ברמה ה- $i$  של עץ  $B_k$  ישנים  $\binom{k}{i}$  צמתים.

## ערימה בינוומית:

ערימה ביןומית תהיה אוסף של עצים ביןומיים, כאשר בכל עץ מתקיים לכל צומת שהמפתחות של הילדים גדולים מהמפתח שלו. מדובר בראשימה של עצים ביןומיים, אשר נרשא להחזיק **כל היורע עץ אחד מכל דוגמה**. נחזק אתם מהעץ הקטן ביותר.

- אם יש לנו ערימה עם **מ** איברים, יהיו לנו לכל היותר ***log m*** עצים. לא יכולים להיות יותר יותר, בפרט לא יכול להיות עץ עם דרגה יותר גבוהה מ-*log m* כיון שהעץ יוכל יותר מ-*m* =  $2^{\log m}$  איברים.
  - יש דרך לבחור סוג עצים כדי להגיע ל-**מ** איברים. כיון שבכל מספר ניתן לייצוג בינארי בצורה יחידה, זה משליק אוטומטית על דרגות העצים הבינומיים שנבחר, לפי **החזקות של 2 ביצוג הבינארי** (אך אין דרך אחת לסדר את האיברים בעצים אלו).
  - כדי למצוא מינימום בצורה ישרה, נחזק מצביע לשורש המינימלי, ונתחזק אותו.

## Binomial Heap - definition



Each integer  $n$  can be written in a unique way as a sum of powers of 2:  
 $n = 11_{(10)} = 1011_{(2)} = 8+2+1$

At most  
 $\lfloor \log_2(n + 1) \rfloor$  trees

פעולות:

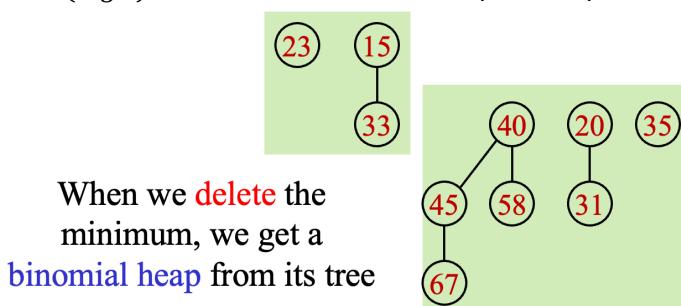
- link (חיבור של שני עצים בינומיים) – ניקח שני עצים באותו הגודל. נשווה בין השורשים שלהם. נחבר את השורש הקטן יותר בטור הבן השמאלי של השורש הגדל יותר.
- meld (איחוד של שתי ערים בינומיות) – בוצע חיבור בדומה לחבר עצים מגודל קטן (ע"י link), ואז נצבור בשארית את העץ לחברו עם העץ הבא, וכן הלאה. לבסוף כל חיבור הוא  $O(\log n)$ .

$$\begin{array}{ccccccc} & B_1 & B_2 & B_3 & & & \\ Q_1: & B_0 & B_1 & - & B_3 & & \\ Q_2: & B_0 & - & B_2 & B_3 & & \\ \hline & & & - & - & B_3 & B_4 \end{array}$$

Like adding binary numbers  
Maintain a pointer to the minimum

$O(\log n)$  time

- Insert (הכנסה) – איבר חדש הוא בעצם עצם עץ  $B_0$ . נפעיל את meld על שתי הערים, כולל עלות  $O(\log n)$ .
- Delete-min (מחיקת המינימום) – לאחר המחיקה קיבל עירמה חדשה בינומית חדשה מחתה-העץ של השורש שמחקנו. בוצע meld בין עירמה זו לעירמה המקורית ללא העץ שבו מחקנו את השורש. גם זה מתבצע ב- $O(\log n)$ .



- הערה: ראיינו שישן 2 דרכים לייצג את העירמה הבינומית. הדריכים האלו משפיעות על השימוש של link, Delete-min, Decrease-key וparent-link.
- link: בדרך הראשונה נצטרך לבצע reverse לשינה לאחר מחיקת המינימום, כדי לשמר על הסדרה ממוינת לפי דרגות העצים הבינומיים, מהקטן לגדול. בדרך השנייה, הרשימה תהיה מעגלית ולא נצטרך לבצע הפוך.
  - Delete-min (הפקחת ערך ממפתח) – מאד דומה ל-Up Heapify שראינו בעירמה בינארית. לשם כך, אנו **צריכים להוסיף** parent-link.

**עירמה בינומית עצלה**

כעת נעבור לנתח amortized, ונראה איך עושים Insert-ב-(1) $O(1)$ ! (נדחה את link עד פועלות Delete-min הבהאה)

**עירמה בינומית עצלה:**

רשימה של עצים בינומיים בדיקון כמו קודם, רק **שנർשה מספר עצים בינומיים מסוומו סוג**. ככלمر נוכל לקבל מצב של רצף עצים שכולם  $B_0$  ואז נקבל בפועל רשיימה מקושורת (aż עצים מגודל 1). איפה העצלות כאן?

- Meld – נשרש את שתי רשימות העצים תוך עדכון המצביע לשורש המינימלי. מתבצע ב-(1) $O(1)$ .
- Insert – נוסיף את האיבר החדש בתור עץ בינומי לרשימה העצים, תוך עדכון המצביע למינימום. מתבצע ב-(1) $O(1)$ .
- Delete-min – לאחר מחיקת המינימום ותיקון הרשימה מחדש, נצטרך **למצוא את המינימום החדש** שיכל להיות כל אחד מהשורשים, יש לנו ת' שורשים ולבן זהה ( $n$ )  **$O(n)$** .

נרצה לבצע נתח amortized cost של Delete-min של  $\log n$ . אנחנו חייבים לבצע תיקונים תוך כדי סדרת הפעולות, אחרת תמיד נוכן להביא את החסם התיכון של ת' הכנסות לעץ, בהן מציאת המינימום החדש הוא ( $a$ ). כדי לטעון **משהו על amortized cost שהוא טוב, אנחנו חייבים לתזקן את העירמה לאחר כל הפעולות**.

בצד נזקן את העץ? בוצע link כדי להוריד את מספר העצים. **תהליך זה נקרא successive linking או consolidating**. מספר סוגים העצים הוא עדין  $\log n$ , יכולים להיות כמה עצים מכל סוג. נבחר את העץ הראשון ונשים אותו ב-bucket. נטעון לעצם הבא, אם יש עץ ב-bucket שמתאים לו, בוצע link בינם ונסים את העץ המחבר בחזרה ב-buckets, וنمשייך את התהליך. **בסוף התהליך נקבל עירמה בינומית רגילה, לא עצלה.**



העולה ב-WC היא  $O(n)$ , לעומת זאת **העלות ב- $O$  amortized תהיה  $O$** . נסמן ב- $L_i$  את מספר links שבעברנו על העץ ה- $i$ . הזמן שלוקח לטפל בעץ ה- $i$  הוא  $1 + L_i$ . נסכם לבלו ונקבל אתjumlah links הכלולות ( $L$ ), בתוספת  $T_0$  (מספר העצים שהוא לפני שהתחלנו את הפעולה Delete-min (Delete-min)). בתוספת 1 –  $k$  (הורדנו את העץ שמחקנו ממנו את השורש, וחשפנו את  $k$  תתי-העצים שהוא הילדיים שלו). נחסום גם את  $L$  במספר העצים הכלול ונקבל:

$$O(T_0 + \log n + L) = O(T_0 + \log n) = O(n)$$

#trees before the process starts  
 #new trees exposed after Delete-Min  $\leq \lfloor \log(n+1) \rfloor$   
 Total #links through the process  
 $T_0 \leq n$   
 $L \leq T_0 + \log n$  why?

ניעזר בשיטת הפוטנציאלי, ונגדיר את הפוטנציאלי להיות **מספר העצים**. סימנו בסוף עם עירמה ביןומית רגילה וכן במספר העצים לאחר הפעולה הוא  $(\log n)O$ .

$$\text{(Scaled) actual cost} = T_0 + \lceil \log_2 n \rceil$$

## Potential = Number of Trees

$$\text{Change in potential} = \Delta\Phi = T_1 - T_0$$

$T_1$  – Number of trees after

$$\begin{aligned} \text{Amortized cost} &= (T_0 + \lceil \log_2 n \rceil) + (T_1 - T_0) \\ &= T_1 + \lceil \log_2 n \rceil \\ &\leq 2 \lceil \log_2 n \rceil \quad \text{As } T_1 \leq \lceil \log_2 n \rceil \end{aligned}$$

ראינו כי ניתוח amortized של כל הפעולות לא פוגע בסיבוכיות, לפי הגדרת הפוטנציאלי שלנו. כל הפעולות נשארות באותו הסיבוכיות שבהן היו קודם.

	Actual cost	Change in potential	Amortized cost
Insert	$O(1)$	1	$O(1)$
Find-min	$O(1)$	0	$O(1)$
Delete-min	$T_0 + \log n$	$T_1 - T_0$	$O(\log n)$
Decrease-key	$O(\log n)$	0	$O(\log n)$
Meld	$O(1)$	0	$O(1)$

עירמת פיבונאצ'י

עירמת פיבונאצ'י:

אוסף של עצים שמקים את הסדר על המפתחות, לשמור מצביע למינימום, אך **העצים לא בהכרח ביןומיים**.

נרצה לבצע Decrease-key ב-(1) $O$ . אחרי שהפחתנו ערך מצומת, ביצענו קא- sift עד השורש, דבר שלוקח  $(\log n)O$ . במקום לבצע את זה, נרצה **להחות את הקשתות שמאורות את כל העירמה**, בכיה נקבל את הפעולה ב-(1) $O$ . סוג העצים שנתקבל לאחר הפירוק הזה, **הם לא בהכרח ביןומיים**.

הבעיה עם החיתובים – נוכל לקבל עצים כמעט מכל סוג, רחבים ועמוקים. ניזכר בתוכנה עבור עץ ביןומי מדרגה  $k$ , שיש לו  $2^k$  צמתים, ולכן לא יוכל להיות לנו עצים מדרגות יותר גבוהות מ- $\log n$  (כי בעץ זהה היינו מקבלים צמתים). בפרט, אם נבצע חיתובים, נוכל לקבל עץ מדרגה  $k$  שיש לו רק  $k$  ילדים. אנו רוצים שיהיה לו  $(2^k)O$  ילדים, כדי שנתקבל  $O(k) = O(\log n)$ .

Cascading cuts

- כאשר צומת מאבד ילד אחד, נסמן אותו.
- כאשר הצומת מאבד ילד שני, נחתור את הצומת מאבא שלו, ונוסיף אותו לרשימת השורשים (הוא הופך להיות עץ בפni עצמו).
- נסיר במובן את הסימון זהה.
- אם באשר חתכו את הצומת מאבא שלו,ABA שלו, נחתור את הczomot מאבא שלו (כלומר איבד ילד אחד קודם), אז גם נחתור אתABA שלו.
- ונוסיף אותו לרשימת השורשים (ככה נפעפם את התהילה לפני מעלה לכל צומת מסומן).
- **שורשים הם אף פעמי לא מסומנים.**

מה שנוטר להוביich הוא:

$O(\log n)$  מונעים עצים רחבים ונמוכים, ככלומר הדרגות ישבו להיות Cascading cuts .1  
 .amortized יכול לגרום ל-( $n$ ) חיתוכים, אבל רק ( $O(1)$  בניתו Decrease-key .2

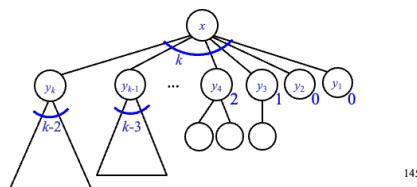
הוכחה 1

למה 1:

## Degrees in Maximally “damaged” trees

**Lemma 1:** Let  $x$  be a node of degree  $k$  and let  $y_1, y_2, \dots, y_k$  be the current children of  $x$ , in the order in which they were linked to  $x$ .

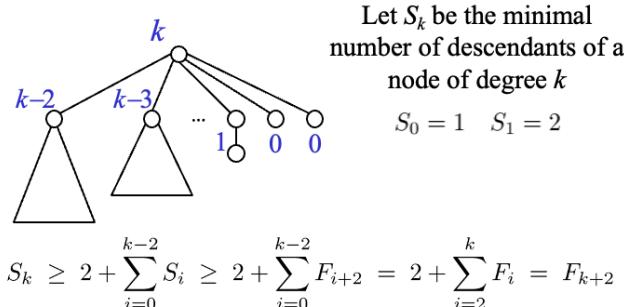
Then, the degree of  $y_i$  is at least  $i-2$ .



**Proof:** When  $y_i$  was linked to  $x$ ,  $y_1, \dots, y_{i-1}$  were already children of  $x$ . At that time, the degree of  $x$  was  $i-1$ . This was also the degree of  $y_i$  (why?) As  $y_i$  is still a child of  $x$ , it lost at most one child.

למה 2:

**Lemma 2:** A node of degree  $k$  in a Fibonacci Heap has at least  $F_{k+2} \geq \phi^k$  descendants, including itself.



## Size of Maximally “damaged” trees

**Lemma 2:** A node of degree  $k$  in a Fibonacci Heap has at least  $F_{k+2} \geq \phi^k$  descendants, including itself.

**Corollary:** In a Fibonacci heap containing  $n$  items, all degrees are at most  $\log_\phi n \leq 1.4404 \log_2 n$

Degrees are again  $O(\log n)$ הוכחה 2

נדיר את פונקציית הפוטנציאל הבאה ונקבל:

$$\text{Potential} = \#\text{trees} + 2 \cdot \#\text{marked nodes}$$

note this 2



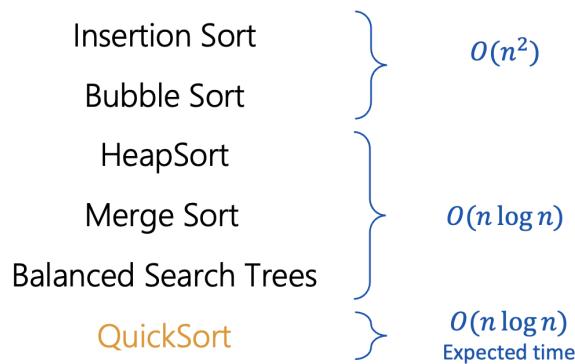
number of cuts performed

	Actual cost	potential: ΔTrees	Potential: 2 · ΔMarks	Amortized cost
Decrease-key	$c$	$+c$	$\leq 2 \cdot (2 - c)$	$O(1)$



## בעית המיין

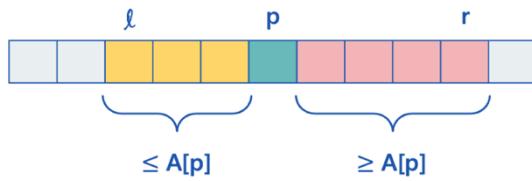
### מיונים מובוסטי השוואות



### מיון מהיר: אלגוריתם

```
Function quicksort (A, ℓ, r)
if ℓ < r then
    p ← partition(A, ℓ, r)
    quicksort(A, ℓ, p - 1)
    quicksort(A, p + 1, r)
```

הרענון – ניקח איבר שרירותי במערך ונקרא לו **טווטן**, ואז נציב מחדש את הערכים במערך כך שמצד שמאל שלו ישבו כל **הערכים שקטנים** ממנו, ומצד ימין שלו ישבו כל **הערכים שגדולים** ממנו, שלב זה נקרא **partition** (חלוקת). ערכים אלו לא בהכרח ממויינים. **בעת ניקח כל אחד מתי המערכות האלו ונמיין אותם רקורסיבית** באמצעות האלגוריתם.



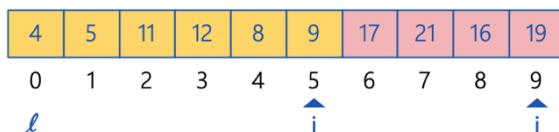
### חלוקת :Lomuto

- נבחר **טווטן** שרירותי, ונחליף אותו עם האיבר הימני ביותר במערך.
- נסרק את האיברים ונחליט עבור כל אחד אם הוא קטן או גדול מה-**טווטן**.
- **נרצה לשמר על שני אזורים בהשוואה לטווטן: קטנים וגדולים.** لكن, כדי לשמר על האזוריים לבצע החלפות של איברים אם יש צורך – נחליף עם האיבר הראשון בקבוצה השנייה.
- נשמר על אינדקסים שמצוירים לסוף של כל אזור:  $i$  מסמל את סוף האזור הצהוב של האיברים הקטנים, ו- $j$  מסמל את סוף האזור הירוד של הגדולים.
- בעת הגענו לסוף המערך:  $r = j$ , ובכיצע הכנסה אחרת של הטווטן במאצע בין הצהובים לירודים.

```
Function partition(A, ℓ, r)
i ← ℓ - 1
for j ← ℓ to r - 1 do
    if A[j] < A[r] then
        i ← i + 1
        A[i] ↔ A[j]
    A[i + 1] ↔ A[r]
return i + 1
```

בכל איבר נגענו רק פעם אחת  
ולכן הסיבוכיות היא  $(n)O$ .

15



### חלוקת :Hoare

- נבחר **טווטן** שרירותי,  
ונחליף אותו עם האיבר  
הימני ביותר במערך.
- ריכז בצד שמאל של המערך את האיברים שקטנים ממנו (צהוב), ובצד ימין את האיברים שגדולים ממנו (ירוד).
- **במהלך האלגוריתם** הצד הצהוב מתקדם ימינה, והצד הירוד מתקדם שמאליה עד שהם נפגשים. אם נתקלנו באיבר צהוב, אנחנו תקועים. אם נתקלנו באיבר ירוד, אנחנו תקועים. נעבור לצד הימני של המערך ונתקדם למרכז. אם נתקלנו באיבר צהוב, אנחנו תקועים. **באשר נתקעים בשני הצדדים,** נבצע החלפה בין שני האיברים הקיצוניים ביותר בכל אחת מהקבוצות (הימני ביותר בצהובים, והשמאלי ביותר בירודים).
- באשר לא מתקיים  $j < i$  האלגוריתם הסתיים.

**הערה:** נשים לב שבסוף החלוקה של Hoare השארנו את הטווטן בצד ימין של המערך, ובאופן זה למעשה למשה הטווטן היה חלק מהתוצאות. המערך הימני שעבורי מבצעים קריאה וקורסיבית, שבו איברים אשר גדולים או שווים לטווטן. לחłówין, יכולנו לסיים את חלוקה Hoare בהחלפת הטווטן עם האיבר השמאלי בתה המערך הימני, ואז היינו מקבלים התנהגות דומה לו של Lomuto: באופן זה בסוף החלוקה הטווטן נמצא במקום הסופי שלו בסדר הממוין, וכן אפשר להמשיך עם קריאות וקורסיביות על שני הצדדים של המערך, שכן כוללות את הטווטן עצמו.

**מיון מהיר: סיבוכיות**

ננתח את הסיבוכיות של האלגוריתם במקרים השונים: הטוב ביותר, הגרוע ביותר, והממוצע. עלות האלגוריתם תושפע ממה:

- ממבצעת פעלת **Partition**, בລומר מהמקום היחסי של **h-sot**.

**: $O(n \log n)$  - Best Case**

במקרה זה המקום הטוב ביותר של **h-sot** הוא באמצע המערך, בລומר **החציון של המערך**.

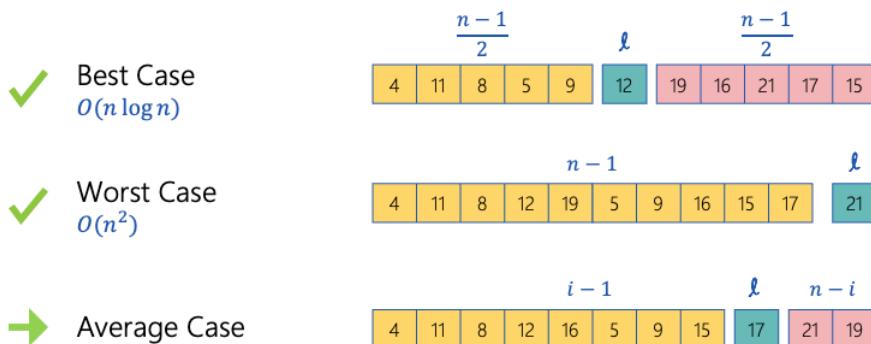
- נתבוס על חלוקת Lomuto. נסמן ב- $C_n$  את מספר השוואות שנבצע על מערך בגודל  $n$ . נחשב רקורסיבית. בסיס האינדוקציה נקבע  $C_0 = C_1 = 0$ . לערך בגודל  $n$  ניאלץ לבצע  $n - 1$  השוואות עבור ה-**Partition** הנוכחי, ועוד נקבע שני מערכיים בגודל  $\frac{n}{2}$ . בລומר  $C_n = 1 + 2C_{\frac{n}{2}}$ . באינדוקציה נקבע  $\log n \leq C_n$ . ניתן לראות בשרטוט עץ רקורסיבי  $\log n$  שכבות של  $n$  עבדה. הסיבוכיות היא  **$n \log n$** .

- מקרה נוסף הוא כאשר המערך מתפרק ל- $\frac{n}{10}$  בצד שמאל ו- $\frac{9n}{10}$  בצד ימין כל הזמן. עץ רקורסיבי לא יהיה מאוזן. העומק של העלה הגבוה ביותר יהיה  $\log_{10} n$ . העומק של העלה הכע عمוק יהיה  $\log_{\frac{9}{10}} n$ . גם במקרה זה נקבע סיבוכיות של  **$n \log n$** .

**: $O(n^2)$  - Worst Case**

במקרה זה **h-sot** יהיה **המקסימום/המינימום במערך**. במקרה זה נפרצל את המערך למערך מאוד גדול שצריך לעבוד עליו. נקבע מערך בגודל  $n - 1$  ומערך בגודל 1.

- בכתב 0,  $C_0 = C_1 = 0$ , ובנוסף  $C_n = n - 1 + C_{n-1}$ . באינדוקציה נקבע  $(n-1)^2 = O(n^2)$ .
- אם המערך הוא ממויין, ונבחר תמיד את **h-sot** בצד ימין, אז במקרה זה נקבל את ה
עלות הגרועה (על אף שהוא כבר ממויין).

**: $O(n \log n)$  - Average Case**

```
Function rand-quicksort (A, ℓ, r)
if ℓ < r then
    p ← rand-partition(A, ℓ, r)
    rand-quicksort(A, ℓ, p - 1)
    rand-quicksort(A, p + 1, r)
```

```
Function rand-partition (A, ℓ, r)
i ← rand(ℓ, r)
A[i] ↔ A[r]
return partition(A, ℓ, r)
```

**מיון מהיר Randomized: סיבוכיות**

במקרה זה **h-sot** יבחר **אקרואית במנור**. כילומר האלגוריתםicut לא יהיה דטרמיניסטי. יצא שלכל קלט, לפחות פעם אחת הריצה תהיה טוב ולפעמים זמן הריצה יהיה רע. לכל קלט, ההסתנהות הרעה מאד לא סבירה (למרות שה-WC הוא אפשרי). אנחנו نطפל בתוחלת זמן הריצה, ליתר דיוק **בתוחלת מספר ההשואות שהאלגוריתם האקרואית יחוות**.

תוחילה נשים לב שעבור אלגוריתם אקרואית, עדין ה-WC יהיה  $O(n^2)$ . אם האיבר האקרואית יהיה המינימום/המקסימום (אפילו אם זה לסייעו). מה ההסתברות שהאלגוריתם האקרואית יבצע את זמן הריצה הגרוע הזה?

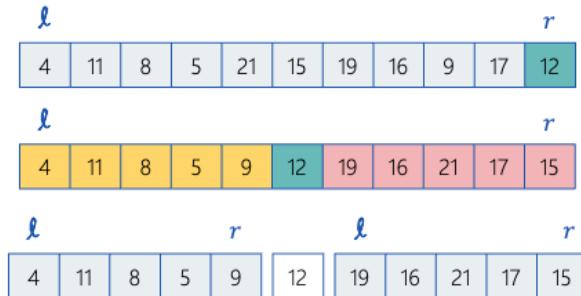
- נגדיר  $C_n$  – מספר ההשואות שבוצעות על המערך בגודל  $n$ . נגדיר  $0 = C_0 = C_1$ .
- העלות בגודל  $n$  היא  $(C_{n-1} + C_{n-i}) + \frac{1}{n} \sum_{i=1}^n (C_{i-1} + C_{n-i}) = n - 1 + \frac{1}{n} \sum_{i=1}^n C_i$ . נשים לב כי מיצג את המקום בסדר המופיע של **h-sot**.
- האקרואית שנבחר. בנוסחה מחשבים תוחלת (ממוצע) של מספר ההשואות שיבוצעו, ע"י חישוב תוחלת מותנה בהינתן שמיום **h-sot** הוא  $\frac{1}{n}$ , ומיצוע על פני כל ערך אפשרי של  $i$ .



נכיה את נוסחת הנסיגה באמצעות שתי לומות:

1. лемה 1: כל זוג מפתחות במערך מושווה לכל היותר פעם אחת.

**Lemma 1** Every two keys are compared at most once by quicksort

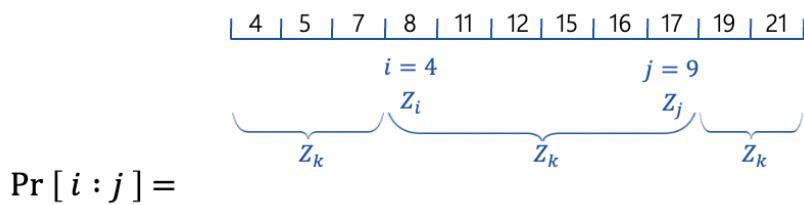


**Proof:** Comparison to pivot, and then pivot goes “solo”

2. лемה 2: ההסתברות שימושים את האיבר ה- $i$ - בגודלו לאיבר ה- $j$  בגודלו (כאשר  $j < i$ ) היא  $\frac{2}{j-i+1}$ . כמובן באשר הפרש האינדקסים של שני איברים גדול יותר, ההסתברות שהם ישווי אחד לשני קטנה (המבנה יגדל). איברים רחוקים, ה-pivot יפריד ביניהם והסבירו שיוויו ביניהם הוא קטן.

בהוכחת lemma 2 השתמשנו בהסתברות שלמה ובהתברות מותנה על מיקום ה-pivot ביחס לשני האיברים, וכך זה מוביל על ההסתברות שהאיברים ישוויו ביניהם.

- אם אף אחד מהאיברים לא נבחר להיות ה-pivot:
  - אם הם שווים ישתייכו לאותו תת-מערך (מיינן או משמאלי ל-pivot) יש סיכוי שיוויו בעתיד.
  - אם הם שונים לתת-מערכות שונים, לא קיימת אפשרות שיוויו בעתיד.
- אם אחד מהאיברים היה ה-pivot:
  - הם ישוויים בהסתברות 1, ולאחר מכן לא ישוויו עוד.



$$\Pr[i:j] =$$

$$\begin{aligned} \rightarrow & \Pr[i:j | k < i] \cdot \Pr[k < i] \\ & + \Pr[i:j | k > j] \cdot \Pr[k > j] \\ & + \Pr[i:j | i \leq k \leq j] \cdot \Pr[i \leq k \leq j] \end{aligned}$$

זה"ב לאחר הוכחה של כל מקרה בנפרד (ללא חישוב הסתברות המיקום של ה-pivot, אלא רק ההסתברות בהינתן המיקום של ה-pivot):

**Summary of Lemma 2 proof in one slide:**

$$\Pr[i:j | k < i] = \frac{2}{(j-i+1)} \quad (\text{by induction})$$

$$\Pr[i:j | k > j] = \frac{2}{(j-i+1)} \quad (\text{by induction})$$

$$\Pr[i:j | i \leq k \leq j] = \Pr[k = i \vee k = j | i \leq k \leq j] = \frac{2}{(j-i+1)}$$

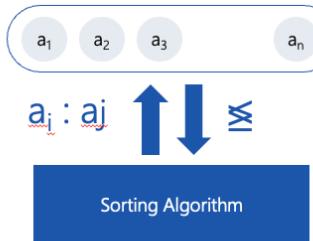
$$\Pr[i:j] = \frac{2}{(j-i+1)}$$

לאחר הוכחת שתי הלומות, **נותר להוכיח** את הטענה כי הסתברות הממוצעת היא  $O(n \log n)$ .



## חסמים תחכਮים למיניהם מבסיסי השוואות

### מודל ההשוואות:



המודל שנשתמש בו הוא **מודל ההשוואות**. במודל זהו יש לנו **צ'אלמנטים** בקלט, ואלגוריתם המיוני ממיין אותם. המבשיר היחיד שבאמצעותו האלגוריתם יכול לגשת לקלט הוא באמצעות בקשה להשוואת שני מפתחות, וקבלת התוצאה.

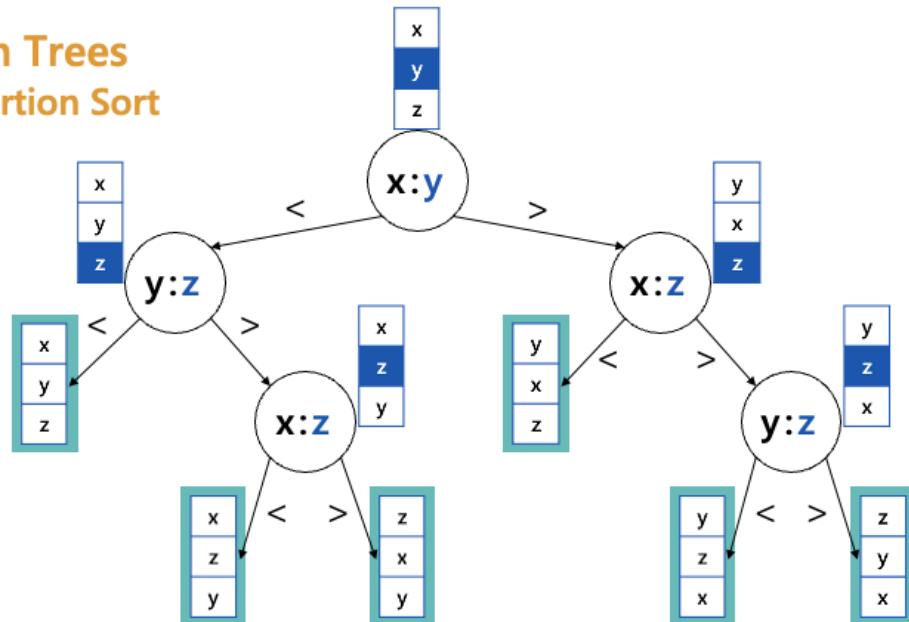
The **only** access that the algorithm has to the input is via **comparisons**

### עצי השוואות:

נדגים על **Insertion Sort**.

משווים את  $z$  עם  $x$  שנמצא מעליו, וכתוצאה מההשוואה מחליטים מה לעשות. אנו מתפצלים לשתי אפשרויות (קטן או גדול). באפשרות אחת,  $x < z$  גלגלו את  $z$  ובעת אנו משווים את  $z$  עם  $x$  – גם כאן נתפצל...

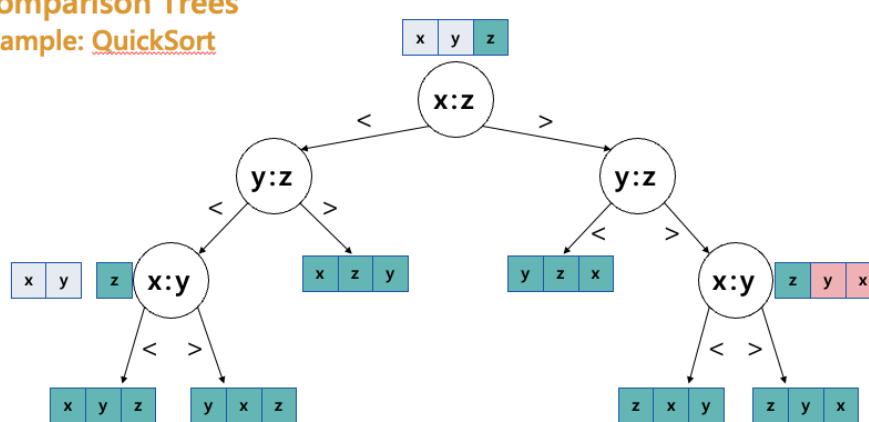
### Comparison Trees Example: Insertion Sort



אם יש לנו **צ'אלמנטים** במערך. מספר העלים בעץ ההשוואות במיון **Sort** יהיה!  
אנו מיציגים את **הסידורים השונים של האיברים במערך**, ויש!  
אנו מוצאים סידור אחד שונה לסידורים אחרים. זהו מספר העלים לא רק-ב-**Insertion Sort** אלא בכל אלגוריתם שהוא "יעיל" במובן שאין שני מסלולים שונים עם תוצאה סופית זהה. העלים מיצגים את **מספר הסדרים האפשריים** שבהם האלגוריתם יכול להסתמך.

ניתן להציג גם על **Quicksort**:

### Comparison Trees Example: QuickSort



משתי הדוגמאות נובל להסיק:

1. כל אלגוריתם מuin מבסיס השוואות, ניתן להמיר **לענ' השוואות**, והענ' זהה עשויה סימולציה של האלגוריתם.
  2. ענ' ההשוואות הוא **עץ בינארי**, כי לכל השווה יש שתי תוצאות אפשריות – ולכן שתי מסלולים שונים בענ'.
  3. כל אחד מהעלים הוא נקודת עצירה של האלגוריתם, ומיצג סדר שונה במערך. מספר העלים הוא !n.
- **אם מספר העלים יהיה קטן מ-!n** זה אומר שבצעם אחד הסדרים פוסף בתור תוצאה סופית, וזה אומר **שהאלגוריתם לא נובן**.
  - אם מספר העלים גדול מ-!n זה אומר שסדר אחד מופיע לפחות פעמיים בענ', ולכן אפשר להגיע אליו בשתי צורות שונות של האלגוריתם ובנראה שהאלגוריתם לא ייעיל במקרה הזה.
  - 4. **המסלול מהשורש לעלה מייצג ריצה של האלגוריתם.** אורכו מסלול מייצג את זמן הריצה עבור קלט שמתאים למסלול ואת כמות ההשוואות שבוצעו בו. לפיכך, **העלות המקסימלית של האלגוריתם היא אורך המסלול הארוך ביותר (עומק העץ)**. **העלות הממוצעת** של האלגוריתם תהיה לקיחת כל הסדרים בחשבון, חישוב זמני הריצה שלהם (עומקיהם) וביצוע ממוצע.

חסם תחתון על עומק עץ בינארי – (הוכחות הלמota, הוכחת המסקנה):

- **למה 1:** אם יש לנו  $m$  עליים בענ', הגובה/העומק המקסימלי של הענ' הוא לפחות  $m \log_2 m$ .
- **למה 2:** אם יש לנו  $m$  עליים בענ', העומק הממוצע של עלה בענ' הוא לפחות  $m \log_2 m$ .
- קיבלנו כי מספר ההשוואות-worst/average case חסום מלמטה על ידי **log n**. כלומר, לכל אלגוריתם מuin מבסיס השוואות **קיים** איזשהו קלט עבורו נדרש זמן זהה לצורך המינון.

**Theorem 1:** Any **comparison-based** sorting algorithm must perform at least  $\log_2(n!)$  comparisons on **some input**.

**Theorem 2:** The **average** number of comparisons, over **all input orders**, performed by any **comparison-based** sorting algorithm is at least  $\log_2(n!)$ .

$$\text{Use: } n! > \left(\frac{n}{2}\right)^{n/2} \rightarrow$$

$$\log_2(n!) > \frac{n}{2} (\log_2 n - \log_2 2) > C n \log_2 n$$

הרנו חסם תחתון לאלגוריתמים דטרמיניסטיים, האם החסם תחתון תופס גם למוצע זמני הריצה של אלגוריתם שהוא אקרראי? טענה נוספת שתוחלת **מספר ההשוואות גם עבור אלגוריתם אקראי חייב לבצע לפחות  $n \log_2 n$  השוואות** (שколоויות 49-55). במצגת Sorting.

**מיונים שאינם מבוססי השוואות**

בעת נדבר על אלגוריתמי מuin שאינם מבוססי השוואות. נראה שבערת אלגוריתמים אלו ניתן למיין בסיבוכיות Worst Case **שhaiia** **نمוכה מהחסם התחתון** שהוכחנו עבור מיונים בהשוואות. בהמשך נראה שגם שגם במודל ההשוואות ניתן למיין בסיבוכיות נמוכה מהחסם התחתון אם הקלט ממון **חלקית**.

**במקום השוואות השתמש בהשומות (Assignments).** במושג "השומות" אנחנו מתייחסים ל"השמות לתוך משתנים". כלומר – אנחנו מרשימים לשתמש בפעולות שמצוות ערכיים לתוך משתנים כדי לבצע את המינון. לדוגמה, כפי שנראה ב-Sort-Count, אנחנו נציב ערכים לתוך המונחים השוכנים.

שימוש להשמה הוא פעולה "חזקת יוטר" מהשוואה. "חזקת" במובן שההשמה יכולה להיות בעלת הרבה תוצאות אפשריות, לעומת הטעות השוואת שלה יש רק שתי תוצאות אפשריות (גדול שווה, קטן). לדוגמה, כפי שנראה, אם ישנו 10 מונחים שונים אז ההשמה (או הגדלת מונה) יכולה להסתיים ב 10 תוצאות שונות (הגדלה של אחד מעשרת המונחים).

**מיון מונה (count sort):**

במיון זה אנחנו מקבלים מערך, שבתוכו המפתחות הם integers **��** כאשר יכולות להיות כפליות (אנו יודעים מראש את הטווח R). חשוב לנו לדאוג שהמספרים הקטנים יופיעו מצד שמאל והגדולים מצד ימין בזיכרון, אך מעבר לה, **מספרים עם אותו ערך ישמרו את הסדר היחסי שלהם**. לכן אנו מוסיףם אותיות לצד המספרים (A – מופיע ראשון של המספר, B – מופיע שני, וכו').



Array	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td>A</td><td>2<sub>A</sub></td><td>3<sub>A</sub></td><td>0<sub>A</sub></td><td>5<sub>A</sub></td><td>3<sub>B</sub></td><td>5<sub>B</sub></td><td>0<sub>B</sub></td><td>2<sub>B</sub></td><td>5<sub>C</sub></td></tr> <tr> <td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr> </table>	A	2 <sub>A</sub>	3 <sub>A</sub>	0 <sub>A</sub>	5 <sub>A</sub>	3 <sub>B</sub>	5 <sub>B</sub>	0 <sub>B</sub>	2 <sub>B</sub>	5 <sub>C</sub>		0	1	2	3	4	5	6	7	8
A	2 <sub>A</sub>	3 <sub>A</sub>	0 <sub>A</sub>	5 <sub>A</sub>	3 <sub>B</sub>	5 <sub>B</sub>	0 <sub>B</sub>	2 <sub>B</sub>	5 <sub>C</sub>												
	0	1	2	3	4	5	6	7	8												

המיון עובד בכמה שלבים:

- נספור כמה איברים יש מכל סוג: נחזק מערך counter, כאשר האינדקס מייצג את האיבר במקור, והערך יהיה כמות המופיעים של האיבר במקור. בעצם יש לנו **מנויים שמספרו כמה יש מכל מספר**.

Counter						
	0	0	0	0	0	0

- נשתמש בספירה זו כדי להכניס את האיברים למערך חדש. נהפוך את המנוים למצטברים, ונספר לכל מפתח, כמה מפתחות קיימים שהם קווים ממנה. משמאלי לימיון באשר נחשב כל פעם לפ' צמדים. בעצם יש לנו **cumulative counter** שאומר לכל מפתח, איפה נשים את האחרון מבניהם במקור התוצאה.

Cumulative Counter						
	2	2	4	6	6	9

- נתחיל עם **האיבר הימני ביותר במקור המקורי**, נבדוק במונה המctruber אם היא צריכה לעבור אותו במקור, **ונקבע את המונה באחד**. נמשיך שמאליה באותה הצורה. המטר חיב להיות מימין לשמאלו, אחרת המיון לא יהיה יizin.

Function Count-Sort ( $A, B, n, R$ )	
<pre> for i ← 0 to <math>R - 1</math> do     <math>C[i] \leftarrow 0</math>  for j ← 0 to <math>n - 1</math> do     <math>C[A[j].key] \leftarrow C[A[j].key] + 1</math>  for i ← 1 to <math>R - 1</math> do     <math>C[i] \leftarrow C[i] + C[i - 1]</math>  for j ← <math>n - 1</math> downto 0 do     <math>C[A[j].key] \leftarrow C[A[j].key] - 1</math>     <math>B[C[A[j].key]] \leftarrow A[j]</math> </pre>	

ניתוח סיבוכיות:

$O(n + R)$ . הסיבוכיות היא  $n + R$  כאשר  $R$  הוא טווח המפתחות. במקרה פרטי, עבר  $n$  מספרים שלמים בטווח  $\{cn, \dots, cn\}$ . הסיבוכיות תהיה  $O(cn)$ .

$R$  לא השתמשו בהשוואות ולכן המיון בזמן ליניארי. זה קורה  $R = O(n)$ . אך אם  $R$  מאד גדול, נפנה למין מהסוג .radix sort – הבא –

Counter setup

Counting

Cumulative Counting

Sorted keys into B

Want to sort  $n$  numbers with  $d$  digits each between 0 and  $R - 1$

2	8	7	1
4	5	9	1
6	5	7	2
1	3	0	1
2	4	7	2
3	5	5	5
7	0	2	2
8	3	9	4
4	8	4	4
3	5	3	6

Example:

$d = 4$

$R - 1 = 9$

במקרה שטווח המפתחות  $R$  מאד גדול, מין בסיס הואiesel. הרעיון הכללי הוא זהה: ניקח את כל המפתחות ונחשבו עליהם בעל מרכיבים  $m-d$  ספרות, באשר כל אחת מהן היא בין 0 ל- $R - 1$ . למשל: תעודת זהות מורכבת מ-9 ספרות שכל אחת מהן היא בין 0 ל-9. אפשר גם לחשב על 3 ספרות שבכל אחת בין 0 ל-999.

- בצע מיון יציב **לפי הספרה הפחות משמעותית** על כל המפתח. נשמר על הסדר המקורי. למשל אפשר להיעזר במיון sort count.
- כעת נמיין באופן יציב **לפי ספרות העשרות**. היציבות ששמרכנו עליה קודם עזרת לנו לשמר על היציבות בעת בשנה ספרה זהה בכל המפתחות.
- כך נמשיך ונמיין לפי כל ספרה את כל המפתחות.

ניתוח הסיבוכיות:

2	8	7	1
4	5	9	1
6	5	7	2
1	3	0	1
2	4	7	2
3	5	5	5
7	0	2	2
8	3	9	4
4	8	4	4
3	5	3	6

מספר ה-phases של המוניים הוא  $d$ . בכל phase בזזה אנו מבצעים sort count: מאתחלים  $R$  מונים, שמיים בהם  $T$  איברים. סה"כ נקבל סיבוכיות של  $O(d(n+R))$ .

במקרה פרטי, אם יש לנו  $n$ ahlen בתחום של  $\{1, \dots, d\}$ , אז נוכל למיין אותם בזמן של  $O(dn)$  (בי נחלק אותם ל- $d$  מספרים שככל אחד מהם בעל  $log n$  ספרות. כל sort count יהיה בעלות  $n = R = O(n)$ ).

למשל עבור הטווח של  $\{1, \dots, 1000^4\}$  (מספרים בעלי 12 ספרות לפחות) נחלק ל-4 קבוצות.

- אנו יודעים שערכו של כל אחד מהאיברים במערך הוא לפחות 0 ולכל היוטר  $-1000^4$ . מסיבה זו ניתן להבטיח שכאשר נסתכל על כל 3 ספרות עשרוניות במספר כ"ספרה" בזוזת בבסיס 1000, נקבל לכל היוטר 4 "ספרות" בכל מספר. נראה **כל מספר בן 4 "ספרות"ycl שצל אחת בבסיס 1000. שכן במקרה זה  $d=R$** .
- המידע על הטווח של האיברים מאפשר לנו להבטיח שניתן ליצגם ע"י מספר קבוע של ספרות (4 במקרה זה) בבסיס שבחרנו. **נרצה ליצג כל מספר בבסיס 1000**, כל שבל "ספרה" בו תהיה בטווח של 999-0. כך נוכל למיין את 1000 המספרים בעלות לינארית.

- Example:  $n = 1000$ , and you have 12 decimal digits.  
Divide into 4 groups, each has 3 decimal digits  
 $\leftrightarrow$  one "thousand digit", ranges 0-999 ( $R$ )

$2 \ 2 \ 3 \ 5 \ 4 \ 1 \ 0 \ 6 \ 7 \ 4 \ 2 \ 9$   
 $\underbrace{\hspace{1cm}}_{d=4} \quad \underbrace{\hspace{1cm}}_{d=4} \quad \underbrace{\hspace{1cm}}_{d=4} \quad \underbrace{\hspace{1cm}}_{d=4}$

- Can sort 1000 numbers, in range:  $0, 1, \dots, 1000^4 - 1$  in  $O(4 * (1000 + 1000))$  time

בעיית הבחירה**QuickSelect**

תחת המטריה של אלגוריתם QuickSelect נראתה שני אלגוריתמים: אחד שהוא אקראי, והשני שהוא דטרמיניסטי. אלגוריתמים אלה יבחן אחד מהשניים באופן שבו הם בוחרים את pivot.

**The Algorithm: QuickSelect( $k$ )**

**QuickSelect( $k$ )**  
 (Cormen: Select)  
  
 Arbitrary Deterministic Pivot  
 (pivot on corner)

**Randomized-QuickSelect( $k$ )**  
  
 Random Pivot

**MedofMed-QuickSelect( $k$ )**  
  
 Deterministic Pivot:  
 Median of Medians

תיאור סכמטי של QuickSelect: הרעיון הוא לנסות לעשות את QuickSort **טיפה עצלה**. בולם, אם בוחין בנקודות מסוימות שבהן הוא לא צריך לבצע עבודה, נהיה עצנים.

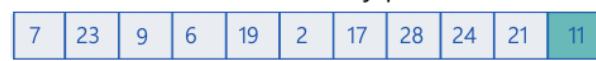
- נבחר pivot, וביצע עליו Partition. קיבל את המערך כאשר מימיינו איברים גדולים ממנו, ומשמאלו איברים קטנים ממנו.
- במקומות המשיק QuickSort ברגיל, ניקח רק את הצד שבו  $k$ -האך שאנו מוחפשים יכול להימצא (או ימין או שמאל).
- במקרה ש- $k > n$  ניקח את הצד ימין, ובמקרה ש- $k < n$  ניקח את הצד שמאל.



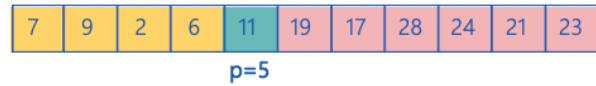
Which value is the 6<sup>th</sup> smallest ( $k=6$ )?



Pick an arbitrary pivot



Partition on a pivot



Where is  $k$  ( $k=6$ )?

- שוב נבחר Pivot, נבצע Partition, ושוב נמשיך בצד הרלוונטי לחיפוש.

- געזר באשר  $p = k$  ומצאנו את האיבר שאנו חיפש.

- 
- 

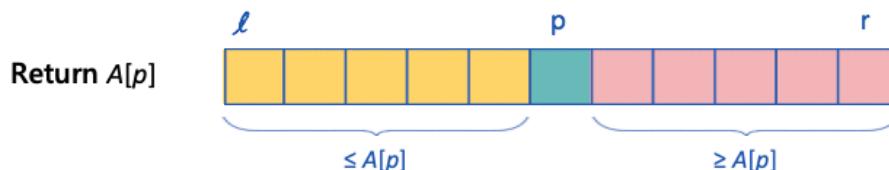
Function QuickSelect ( $A, l, r, k$ )

```

if  $l < r$  then
     $p \leftarrow$  Arbitrary-Partition ( $A, l, r$ )
    if  $p > k$  then
        | QuickSelect ( $A, l, p - 1, k$ )
    Else if  $p < k$  then
        | QuickSelect ( $A, p + 1, r, k$ )
    Else
        | Return  $A[p]$ 

```

Arbitrary  
Pivot



air נבחר את Pivot:

- אם נבחר את המינימום/המקסימום ביצענו חיתוך קטן מאד והסיבוכיות תהיה גדולה.

- אם נבחר באופן רנדומלי, בממוצע הוא ישב בערך באוזן המרכז.

**QuickSelect – אנחנו צריכים את בעיה – אנחנו צריכים את החצינו!** יש כאן מלכוד קטן.  
בשביל למצוא את החצינו! יש כאן מלכוד קטן.

- 
- 
- 

## אלגוריתם QuickSelect רנדומלי

באלגוריתם זה נבחר Pivot אקראית. כל שאר הלוגיקה זהה לתיאור הכללי שתואר קודם, ממשיכים לצד הנכון לאחר ה-Partition.

מה תיראה הסיבוכיות?

- במקרה הטוב – אם יהיה לנו Partition גרוע כמו ב-QuickSort נקבל ביצועים גרועים בהתאם, בЛОמר ( $O(n^2)$ ).
- במקרה הטוב – יכול להיות שנקבל ישר  $k = d$ , ומיד האלגוריתם מסתים. ביצענו Partition אחד ולבן ( $n$ ).

נתחן המקרה הממוצע – ( $O(n)$ ):

מה לגבי המקרה הממוצע? ננתה את תוחלת זמן הריצה של האלגוריתם, כЛОמר תוחלת כמות ההשואות.

- באלגוריתם אנו מחפשים Pivot ואז עושים Partition, ונשארים בסוף עם צד אחד. אז אם התחלנו במבנה שגודלו  $n_0 = n$ , בעת נüber לאחד הצדדים שהוא מבנה בגודל  $n_1$ . בהמשך נüber למבנה בגודל  $n_2$  וכו'. גדלים אלה הולכים וקטנים כל הזמן.
- במאות השלבים באלגוריתם חסומה על ידי  $n$ , כי גודל המבנה החדש קטן לפחות ב-1 כל פעם. כדי לנתח את השלבים נרצה לדעת את כמות ההשואות שהאלגוריתם מבצע בכל אחד מהשלבים, ולנסכם.
- במאות הרשואות בשלב שווה למספר האיברים בשלב. ולכן הקומות הכלליות תהיה  $\dots + n_1 + n_2 + \dots + n_0$ .
- נרצה לחשב את תוחלת הסכום הזה, ששווה לסכום התוחלות כלומר:  $(n)E\sum_i$ .



בעת נגיד:

- למה (הוכחה) – תוחלת מספר האיברים בשלב הבא תהיה לכל היותר  $\frac{3}{4}$  ממספר האיברים בשלב הקודם. אנו מחפשים את תוחלת מספר האיברים שהוסרו, ומגיעים למסקנה שהוא  $\frac{n}{4}$  וכן מספר האיברים שיישארו הוא  $\frac{3n}{4}$ .
- משפט (הוכחה) – תוחלת גודל השלב ה- $i$  תהיה לכל היותר  $n \cdot \left(\frac{3}{4}\right)^i$ .

**Theorem:**  $\mathbb{E}[n_i] \leq \left(\frac{3}{4}\right)^i n$ 

Using: Lemma

Inductive assumption

**Proof method: induction on  $i$ .**

$$\mathbb{E}[n_i | n_{i-1} = x] \leq \left(\frac{3}{4}\right)x$$

$$\mathbb{E}[n_{i-1}] \leq \left(\frac{3}{4}\right)^{i-1} n$$

**Proof for  $i$ :**

$$\begin{aligned} \mathbb{E}[n_i] &= \sum_x \Pr[n_{i-1} = x] \mathbb{E}[n_i | n_{i-1} = x] \\ &\leq \sum_x \Pr[n_{i-1} = x] \left(\frac{3}{4}\right)x \\ &= \left(\frac{3}{4}\right) \mathbb{E}[n_{i-1}] \end{aligned}$$

Condition the expectation

Plug in Lemma

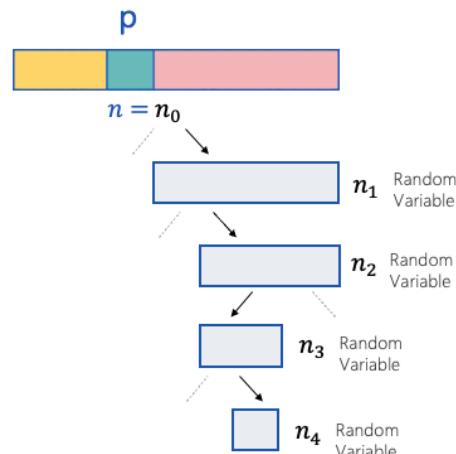
$$\leq \left(\frac{3}{4}\right) \left(\frac{3}{4}\right)^{i-1} n$$

$$= \left(\frac{3}{4}\right)^i n$$

QED

כך נוכיח שסכום התוחלות זהה הוא  $O(n)$ :**Analysis: Average Case (Expected time)****Lemma:**  $\mathbb{E}[n_{i+1}] \leq \frac{3}{4} n_i$ **Theorem:**  $\mathbb{E}[n_i] \leq \left(\frac{3}{4}\right)^i n$ **Result:** Total **expected** number of comparisons is bounded:

$$\sum_i \mathbb{E}[n_i] \leq n \left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots\right) = 4n = O(n)$$

**אלגוריתם QuickSelect חציון החזירנים**מוטיבציה:ראינו שהWC באלגוריתם הרנדומלי היא  $(n^2)$  וזה די מאכזב. באלגוריתם זה נבחר **Pivot** דטרמיניסטי שمدמה/էprox את החציון.

מה שעשינו קודם:

**QuickSelect = Get Pivot + Partition + Recursive Select**

Randomized QuickSelect (worst case):

$$\begin{aligned} T(n) &= O(1) + C_1 n + T(n-1) \\ \Rightarrow T(n) &= O(n^2) \end{aligned}$$



מה שנעשה עבשו (נשאף למצוא את החזיון ולחולק את המערך ל-2):

**QuickSelect = Get Pivot + Partition + Recursive QuickSelect**

Get a central pivot (median):

$$T(n) = ? + C_1 n + T(n/2)$$

Want: both values LOW

$$T(n) = T( ) + C_1 n + T( )$$

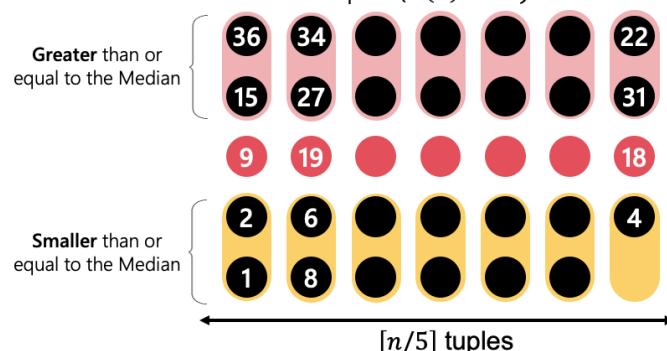
נרצה למצוא פתרון שבו שני ה-T בנוסחת הנסיגה יהיו נמוכים יחסית, ושניהם ביחד יתנו ערכיים נמוכים.

#### פעולות האלגוריתם:

האלגוריתם מבצע את בחירת pivot בשימוש טכנית שנקראת חיזיון החזיונים. נתחילה במערך עם מ איברים.

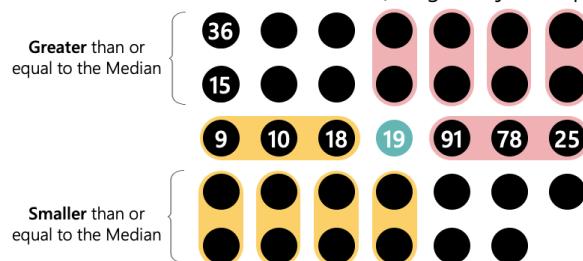
- **חלוק את המערך לחמשיות.** מספר החמשיות הוא  $\lceil \frac{n}{5} \rceil$ .
- **בכל חמישייה נחפש את החזיון שלה.** העלות הכללית של מציאת החזיוונים בכל החמשיות היא  $O(n)$ . לדוגמה: נציג את האיברים שגדולים מהחזיון מעליון, ואת האיברים שקטנים ממנו מתחתיו.

Find the median of each 5-tuples ( $O(n)$  work)



- ניקח את החזיוונים שמצאנו, יש לנו  $\lceil \frac{n}{5} \rceil$  מהם. נמצא להם חזיון באמצעות קריאה לאלגוריתם עצמו רקורסיבית. העלות של הקריאה הזה היא  $(\lceil \frac{n}{5} \rceil)T$ . האזרחים הורודים גדולים שווים לחיזיון החזיונים, והצחובים קטנים שווים לחיזיון החזיונים.

Relations to median of the medians (reorganize just for presentation)



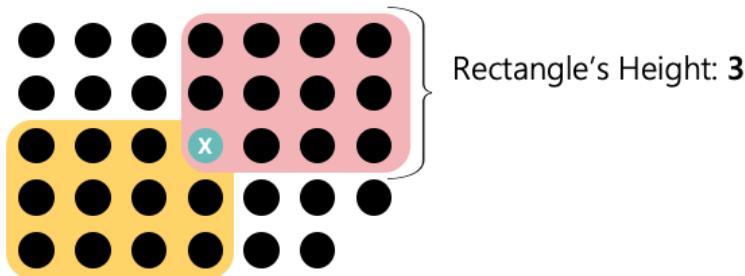
דברנו על המונחים Median Of Medians ועל MedofMed-QuickSelect. בآن נסביר את ההבדלים ביניהם:

- Median Of Medians הינו שם השיטה של **בחירה חזיון החזיונים**, המשמשת במקרה שלנו למציאת pivot.
- MedofMed-QuickSelect הוא השם שאנחנו נתנו (לא מופיע בספרות) **אלגוריתם הרקורסיבי שמבצע את select** תורן.
- שימוש בשיטת Median Of Medians למציאת pivot.

**נתוח סיבוכיות -  $O(n)$  ב-WC:**

ננתח את כמה האלמנטים שצבועים בוורוד/בצהוב:

How many values are greater than or equal to the pivot?



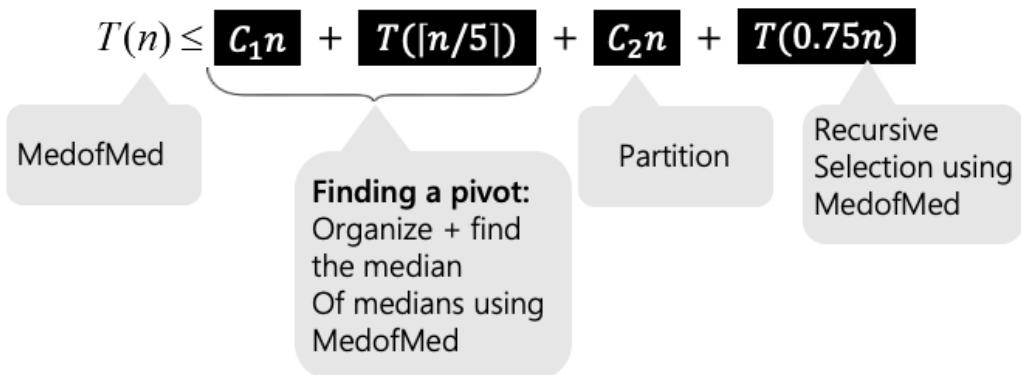
$$\text{Rectangle's Width: } \geq \frac{n}{5 \cdot 2} - 1$$

$$\# \text{of elements in pink} \geq 3(0.1n - 1) - 3 = 0.3n - 6$$

(2nd 3 – right column might be partial)

בעת ננתח את ה-Partition Partition. מספר האלמנטים שירד מהמערך הוא גדול או שווה ל- $a$ . בולם, מספר האלמנטים שיישארו איתנו לשלב הרקורסיבי הבא יהיה חסום מלמעלה על ידי  $0.75n$ . בולם הקריאה חסומה על ידי  $T(0.75n)$ .

בסוף של דבר נקבל:



נשים לב כי יש לנו שתי עליות רקורסיביות, וסכום השברים האלה נותן ביטוי שקטן מ- $n$ . **זה המפתח לעלות הכלולות ( $O(n)$ )**. ישנה הוכחה שראה כיצד פתרון מסוימת הנסיגה מהצורה  $T(n) = T(an) + T(bn) + cn$  כאשר  $a + b < 1$  היה  $O(n)$ .

**כאן** ישנו תרגיל בתרגילים לסיבוכיות של Select במבנים שונים. בנוסף, היות אנחנו יודעים למצוא חיצון בעילות, בולם ב-( $O(n)$ ), ניתן להשתמש בחיצון זה בתורו נזון לאלגוריתם QuickSort ולהווריד את סיבוכיות WC-  $O(n \log n)$ .

Randomized-QuickSelect:  $O(n)$  expectedDeterministic QuickSelect  
(MedofMed):  $O(n)$  worst case→ Can use in QuickSort  $O(n \log n)$  worst case

סיכום:

בשיעור זה רأינו כיצד ניתן לפתור את בעיית הבחירה בזמן **ליניארי** במקרה הגרוע. מכאן, שבעית הבחירה "כליה" יותר חשובה במשמעות המילוי, שכן שלמדנו דורשת (*login*) זמן לפחות **ליניארי**.

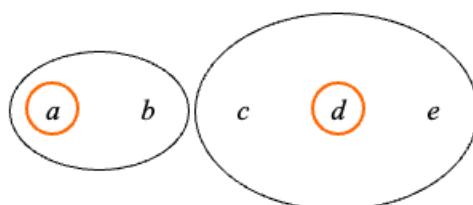
- **האלגוריתם האקראי שלמדנו** פותר את בעיית הבחירה בזמן **ליניארי בתוחלת**. בפועל, זהו אלגוריתם יעיל למד". יחד עם זאת סיבוכיות הזמן במקרה הגרוע היא  $O(n^2)$ .
  - בדומה למילוי מהיר, אלגוריתם זה מחלק את המערך על פי **ציר** שנבחר אקראי. בשונה ממילוי, הוא ממשיך רקורסיבית רק **באחד החלקים ולא בשניהם**.
  - **תוחלת גודל** חלק המערך אליו ממשיך האלגוריתם ברקורסיה היא לכל היוטר  $\frac{3}{4}$  מגודלו הנוכחי.
- **האלגוריתם הדטרמיניסטי (חציון החצינום)** פותר את בעיית הבחירה בזמן **ליניארי במקרה הגרוע**. בפועל, קבואי הזמן של גודלים יחסית, והאלגוריתם לאiesel כל כך עבור ערכיהם קטנים של מ. אלגוריתם זה מבצע שתי קריאות רקורסיביות. בשונה מכל האלגוריתמים הרקורסיביים שראינו קודם, **באן לכל אחת משתי הקריאות הרקורסיביות יש תפקיד שונה**:
  - הקריאה הרקורסיבית הראשונה, על מערך שגודלו בערך  $\frac{1}{5}$  מהמערך בשלב הנוכחי, מתבצעת על **חצינו כל החמשיות** הרצפות במערך. מטרתה לאתר ביןיהם את החצינו. כאמור, **לאתר את חצינו החצינום**. איבר זה ישמש כציר לחלוקת המערך.
  - הקריאה הרקורסיבית השנייה מתבצעת על אחד החלקים לאחר החלוקה - זה שבו נמצא האיבר אותו אנו מחפשים. קריאה רקורסיבית זו תתבצע במקרה הגרוע, **כפי שהובחנו**, על חלק שגודלו לא יותר מאשר  $\frac{3}{4}$  מגודל המערך כולו בשלב הנוכחי.
  - סכום הקבואיים שהוזכרו ( $\frac{1}{5} + \frac{3}{4}$ ) קטן ממש מ-1. עובדה זו מניבה זמן ריצה **ליניארי** בגודל הבעיה.

## Union-Find

### הגדרה ויישומים

נדיר ADT חדש בשם Union-Find (UF) אשר מכיל אוסף של קבוצות (הקבוצות תמיד זרות זו לזו). נתמוך בפעולות הבאות:

- **(info): Make-Set(info)**: אפשר להסתכל על זה בתור סוג של **Insert**, רק שפעלה זו יוצרת סביב האיבר הזה קבוצה (נקבל סינגלטן, כלומר זה בעצם **Insert-as-singleton**). הפעולה מחזירה את x, האיבר הבודד שבעתה יושב בקבוצה בלבד.
- **(x,y): Union(x,y)**: מאחדת את הקבוצות שאליים שייכים האיברים x ו-y.
- **Link(x,y)**: מבצעஇיחוד בין שתי קבוצות, **ההמייצגים שלhn**ם x ו-y.
- אפשר לומר כי **(y,x): Union(y,x)** הוא בעצם: **(Link(Find(y), Find(x))**.
- **Find(x)**: מחזירה ערך שמייצג את הקבוצה שמכילה את x (יכול להיות אחד מהאיברים בקבוצה, שם הקבוצה) בלבד שמתקיימת הדרישה הבאה:  $Find(x) = Find(y) \Leftrightarrow x \text{ and } y \text{ are currently in same set}$ . כלומר הערך הזה יהיה זהה עבור שני איברים אמ"מ הם באותו קבוצה. אם הערך הזה שונה, וביצענו **Union**, מיד אחרי זה אמורים לקבל ערך זהה.



$a \leftarrow \text{Make-Set}()$	$c \leftarrow \text{Make-Set}()$
$b \leftarrow \text{Make-Set}()$	$d \leftarrow \text{Make-Set}()$
$\text{Union}(a,b)$	$e \leftarrow \text{Make-Set}()$
$\text{Find}(b) \rightarrow a$	$\text{Union}(c,d)$
$\text{Find}(a) \rightarrow a$	$\text{Union}(d,e)$
	$\text{Find}(e) \rightarrow d$



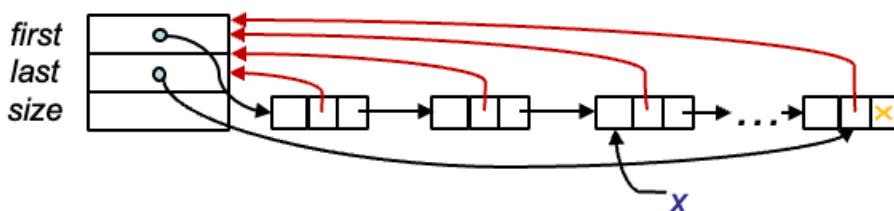
בעת נראה מימושים שונים ואת הסיבוכיות שלהם:

Data structure	Linked lists with union by size	Up-trees with union by size and path compression	
Operations			
		can prove	will prove
Make-Set	O(1)	O(1)	O( $\log^* n$ )
Union( $x,y$ ) <sup>(1)</sup>	O( $\log n$ )	O(1)	O(1)
Find( $x$ )	O(1)	O( $\alpha(n)$ )	O( $\log^* n$ )
		Inverse Ackermann function	Repeated logarithm function

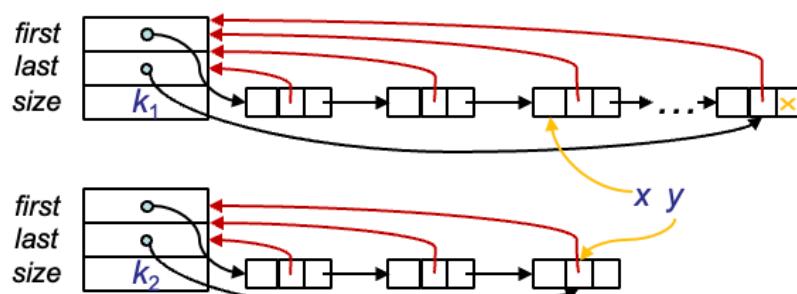
Amortized

<sup>(1)</sup> Ignoring time for 2×Find: Union( $x,y$ ) → Link(Find( $x$ ),Find( $y$ ))**מימוש בעזרת רשימות מקשורות**

כל קבוצה (set) נציג באמצעות **רשימה מקושרת**. הרשימה מכילה את כל האיברים ששייכים אותה קבוצה. לכל איבר יש מצביע לאיבר שמייצג את הקבוצה (head של הרשימה).

**סיבוכיות הפעולות:**

- Make-Set: עובד ב-(1) O(1) ב-WC: בניית רשימה בגודל 1.
- Find: בזכות המצביע שיש לנציג הקבוצה, עובדת ב-(1) O(1) ב-WC.
- Union: גשרשר את שתי הרשימות הקשורות ב-(1) O(ב-WC) כימודר בתיקון מצביעים. נותר לתקן את כל ה-set pointers (ה מצביעים האדומים, המצביעים לאיבר שמייצג את הקבוצה). הסיבוכיות היא (n) ב-WC.

Union( $x,y$ ) –  $O(\min\{k_1,k_2\}) = O(n)$  time

## נכיח זמן amortized של $O(\log n)$

- אחד תמיד את הקבוצה הקטנה עם הגדולה, **union by size**.
- באל set pointer יכול לשנות את היעד שלו לכל יותר  $\log n$  פעמים. בכל פעם שהמצבי עובר קבוצה: אם קבוצה א' וב' התאחדו וקבוצה ב' היא הגדולה יותר, קבוצה א' לפחות כפילה את גודלה (קבוצה ב' לפחות גודלה במנו א'), אז הקבוצה החדשה היא לפחות לפחות פי 2.
- לכן, כל מצביע בצהה (הציג איבר במבנה) ישנה עד  $\log n$  פעמים. לכן  $m$  (מספר האיברים במבנה) פעולות Union יטל  $O(m \log n)$  וכאן ב-**amortized** פועלה בוזחת תעלת  $O(\log n)$ .

### Proof:

- **Make-Set** ו**Find** עושים  $O(1)$  W.C.
- We'll prove the  $m$  Unions take overall  $O(m \log n)$  time
- These Unions involve at most  $2m$  distinct items (at most 2 per Union)
- Now look at any of these items separately, denoted by  $x$   
Whenever the set pointer of  $x$  is changed, the **size** of the set containing  $x$  is at least **doubled**. This can happen at most  $O(\log n)$  times
- In total all the unions required changing  $O(2m \log n)$  pointers
- Spread over  $m$  unions  $\rightarrow O(\log n)$  amortized for Union

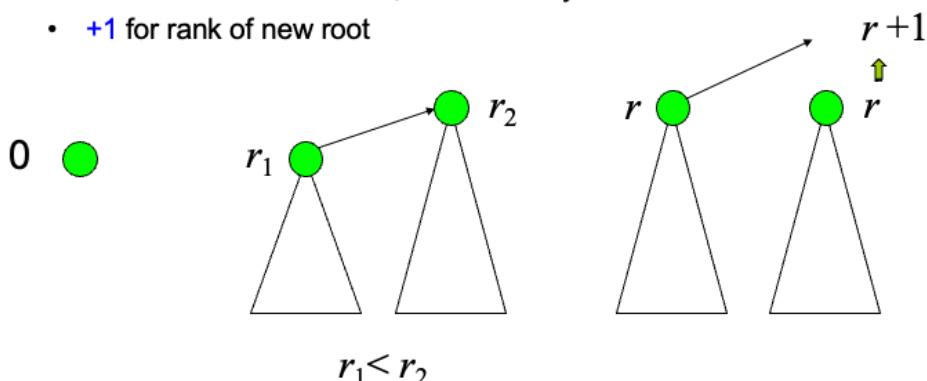
### מימוש בעזרת עצים הפוקים

כל קבוצה נציג באמצעות עץ. כל צומת בעץ בלשונו מייצג איבר במבנה. **הנציג של הקבוצה הוא שורש העץ** (כל הצמתים מצביעים אליו ישירות או בעקיפין). **ישנם מצביעים רק אל ההורים**. עבר צומת  $x$  מסמן את ההורה של  $x$ -ק.א.

- (x) – עוקב אחר המסלול מ- $x$  עד השורש.
- (x,y) – תולה שורש אחד תחת השורש השני (בתוך ילד חדש שלו). ראיינו את זה בערימות ביןומות. נבחר לתלות את העץ הקטן יותר תחת העץ הגדל יותר. אמו מצביעים עלונים **union by rank**.

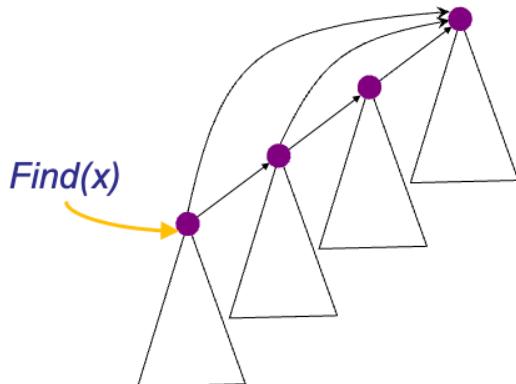
נגדיר rank של סינגלטון כ-0. כשה膺עosesינו נשווה בין ranks (נגידר את זה זמניות בתור **גובה העץ**). נשים לב כי דרגות לעולם לא יכולות לקטן, אך הגבהים של העצים כן יכולים לקטן. השקילות בין rank לגובה תירס בקרוב. **ה-ranks יהיו חסמים עלונים לגובה**.

- Let's define **rank** of a tree to be its **height**
  - This is a **temporary** definition, as you will see soon
- In Union, link **smaller-rank** tree under **larger-rank** tree
  - No change in ranks
- or if both have the **same rank**, link arbitrarily
  - +1 for rank of new root



**תוכנות:**

1. ranks רק עולים, ומהווים חסם עליון לגובה.
2. גבהים ורק יודים (בגלל path compression).
3. אם צומת  $x$  אינו שורש:
  - a. בمسלול כלפי מעלה, הדרגות עלות ממש: מתקיים  $x.rank < x.p.rank$ .
  - b. מהרגע שצומת הפסיק להיות שורש, דרגתו לא תשתנה עוד: תמיד נעשה union עם השורש.
4. שורש מדרגה  $z$  מכיל לפחות  $2^z$  צמתים.
5. יש לכל הוותר  $\frac{n}{2^z}$  צמתים מדרגה  $z$ .
6. ברגע union by rank, ניתן לנו לבצע את Find ב- $O(\log n)$ .

**ביזוע מסלולים (path compression):**

משמעות – נגד שאנו עושים  $Find(x)$ . בשאנו עולים למעלה עד השורש, נתלה כל צומת בדרך תחת השורש העליון ביותר של העץ. בר כל הצמתים בדרך יהיו תלויים תחת השורש. אם לא היו בדרך עוד מושגים, בפעם הבאה שנעשה  $Find$  לצומת בלבדו, נקבל את זה ב- $O(1)$ .

- הערות:
- נדרש 2 מעברים. מעבר ברקורסיה כלפי מעלה עד השורש, ואז בהתקפות חוזרת צירק לעדכן את כל הצמתים בדרך שיפנו לשורש העליון.
  - זה לא משנה את ranks (רק union יככל).
  - **הגבהים עשויים קטנים** – תתי-העצים התקרבו לשורש ולכן הגובה של השורש העליון קטן.

**פונקציית אקרמן**

ניתן להוכיח שאם נבצע  $m$  פעולות בסה"כ, מתוך  $n$  איברים, זה יקח  $(\alpha \cdot m)O(\alpha \cdot m)$  זמן. אך הזמן **באמורטיזד יהיה  $(n)\alpha O(\alpha)$  לפעולה**. לא נוכחים את זה.

מושגים:

- פונקציה חוזרת:  $f^{(i)}(n) = f(f^{(i-1)}(n))$ , for  $i > 0$
- פונקציית אקרמן: כדי להפעיל על פלט  $n$  פעולה במקומות  $k$ , צריך להפעיל את הפעולה הקודמת בהיררכיה ( $A_{k-1}$ ) אבל מספר גדול של פעמים ( $1 + n$  זה כמעט כפול הגדל הקלט).
- פונקציית מגדל -  $Tower(n) = 2^{2^{2^{\dots}}}$  מעלים בחזקה 2  $n$  פעמים.
- מפונקציה שעולה מאוד מהר, נגבור לאחת שיעלה מאוד לאט:  $F(n) = \min\{k \geq 1 | F(k) \geq n\} \rightarrow f(n) = \min\{k \geq 1 | Tower(k) \geq n\}$
- קיבל  $n^* = \log^* n = Tower(n) \rightarrow f(n) = Tower(n) = \min\{k \geq 1 | \log^{(k)} n \leq 1\}$

**The  $\log^* n$  function**

$$\log^* n = \min\{k \geq 1 | Tower(k) \geq n\}$$

$$\log^* n = \min\{k \geq 1 | \log^{(k)} n \leq 1\}$$

<b><math>n</math></b>	<b><math>Tower(n)</math></b>	<b><math>n</math></b>	<b><math>\log^*(n)</math></b>
1	2	0 – 2	1
2	4	3 – 4	2
3	16	5 – 16	3
4	65,536	17 – 65,536	4
5	$2^{65,536}$	$65,537 – 2^{65,536}$	5

“For all practical purposes  $\log^*(n) \leq 5$ ”

35

- פונקציית אקרמן ההופוכה: מחזירה קלט קטן יותר ממה ש- $\log^* n$  מחזירה.

**חסם עליון ( $O(\log^* n)$ )**

- Recall Property 3: At most  $\frac{n}{2^r}$  nodes of rank  $r$

$x.rank$	$level(x)$	$size$
0 .. 2	1	$\leq \frac{n}{2^0} = n$
3 .. 4	2	$\leq \frac{n}{2^3} < \frac{n}{2^2} = \frac{n}{T(2)}$
5 .. 16	3	$\leq \frac{n}{2^5} < \frac{n}{2^4} = \frac{n}{T(3)}$
17 .. 65,536	4	$\leq \frac{n}{2^{17}} < \frac{n}{2^{16}} = \frac{n}{T(4)}$
65,537 .. $2^{65,536}$	5	$\leq \frac{n}{2^{65537}} < \frac{n}{2^{65536}} = \frac{n}{T(5)}$
...	...	
$T(i-1)+1 .. T(i)$	$i > 1$	$\leq \frac{n}{2^{T(i-1)+1}} < \frac{n}{2^{T(i-1)}} = \frac{n}{T(i)}$

לשם פשוטות נזכיר רק את החסם של ( $n \log^* n$ ) על הפעולות amortized של הפעולות .MakeSet

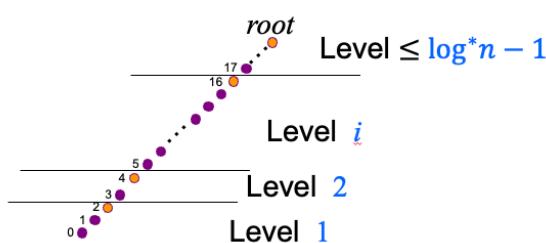
- נגידיר את המושג level (מעמד):  $level(x) = \log^*(x.rank)$
- הכומת  $x$  הוא  $level(x) = \log^*(x.rank)$
- לכל מעמד יש טווח דרגות שמתאים לו. אפשר להסתכל על זה כמו בטור דרגות שכר. ככל שעולים בטוווח, יש הרבה הרבה הרבה אפשרויות, אבל פחות אנשים ישימצאו בטוווח זהה.
- חссום בכל מעמד, את הקומות לפי ה- $rank$ .
- הנמור ביותר באותו המעמד. תכונה: מספר הצמתים ב- $i$   $level$  הוא לפחות  $\frac{n}{T(i)}$ .

נרצה לשאול כמה עולה פעולה Find:

- rank של שורש יכול לאגדול כל הזמן בגובה שלו, כלומר  $log^*$ . לכן לא ניתן לטעות מ- $log^*$  מ-5 אם יש המונן צמתים במבנה הזה, ואם לא  $2/3/4$  או אפילו רק 1.

$$level(root) = \log^*(rank[root]) \leq \log^*(\log n) = \log^* n - 1$$

- Main idea: separate between
  - Pointers along path to a parent of a higher level
  - Pointers along path within the same level of  $x$



- כל פעם ש- $x$  נדרש לשלוט על המעבר דרכו בשינוי המצביע,  $x.rank$  גדול, כלומר הוא מקבל אבא מ- $rank$  גבוהה יותר.

$$2n + \sum_{\ell=2}^{\log^* n - 1} T(\ell) \frac{n}{T(\ell)} = n \log^* n$$

הדבר הזה יכול לקרות לא- $x$  לכל היוטר ( $L$ ). הדבר מגדיל בגובה ( $L$ ). זה נובע מטווח הדרגות במעטם. הדרגה שallow יכולה להיות כל היוטר ( $L$ ). **העלות הכלולות לכל הצמתים בכל פעולות ה- $Find$  (שינוי מצביעים) כאן משמאל. סה"כ נקבל:**

## $O(\log^* n)$ upper bound

Total cost of a sequence of  $n$  make-set's,  $\ell$  link's, and  $f$  find's:

