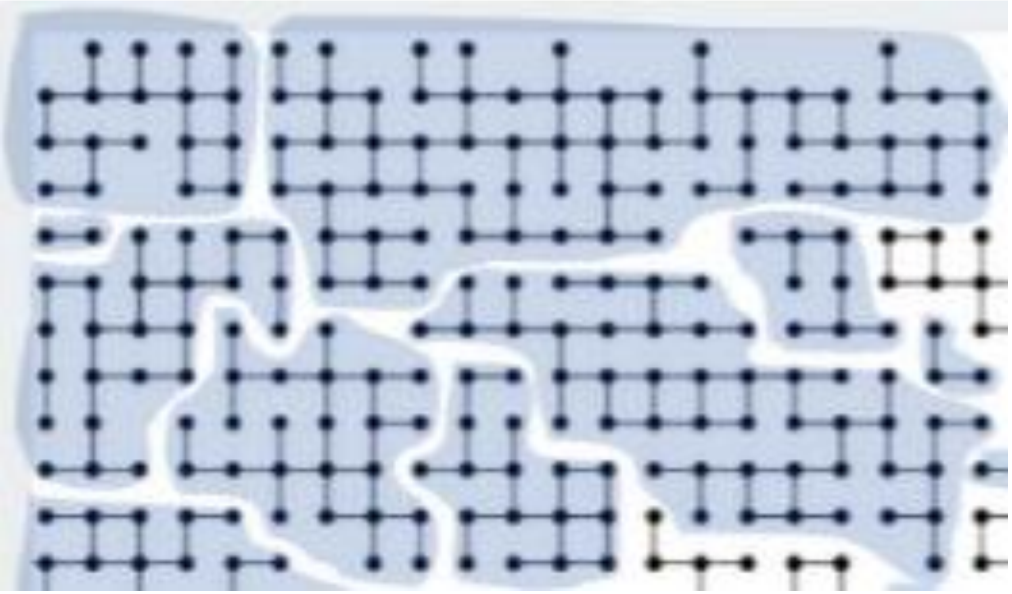
Data Structures

Union/Find – הרצאה 20

איחוד קבוצות זרות (Disjoint sets)

מבנה נתונים אשר מבצע מעקב אחר איברים המחולקים למספר **תתי-קבוצות זרות**

קבוצות זרות = קבוצות ללא חפיפה



- אין משמעות לסדר בין האיברים באותה קבוצה
- מעניין אותנו רק המעקב אחר תכולת הקבוצות הזרות

איחוד קבוצות זרות (Disjoint sets)

הפעולות המוגדרות עבור מבנה זה:

- **יצירת קבוצה (makeSet)** – יצירת קבוצה חדשה המכילה איבר אחד (סינגלטון – singleton)
- **חיפוש (find)** – קביעה איזו קבוצה מכילה איבר ספציפי
  - עוזרת בקביעה האם שני איברים שייכים לאותה קבוצה
- **איחוד (union)** - איחוד שתי קבוצות זרות לקבוצה אחת

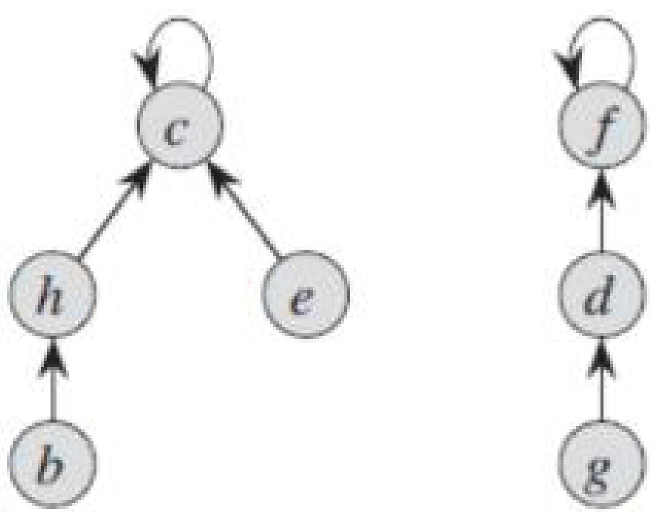
ישנם דרכים רבות לממש מבנה נתונים זה – כאן נתמקד במימוש ע"י **עצים הפוכים – Up trees**

עצים הפוכים - Up trees

- כל קבוצה זרה מיוצגת ע"י **עץ**, כאשר **שורש העץ נחשב כנציג הקבוצה**
- כל צומת בעץ מחזיק מצביע לאביו (parent node)
- שורש העץ נחשב כאבא של עצמו
- העצים שמיצגים את הקבוצות הזרות מהווים **יער**
- **בדוגמה:** ייצוג הקבוצות הזרות

  {d,**f**,g} -ו {b,**c**,e,h}

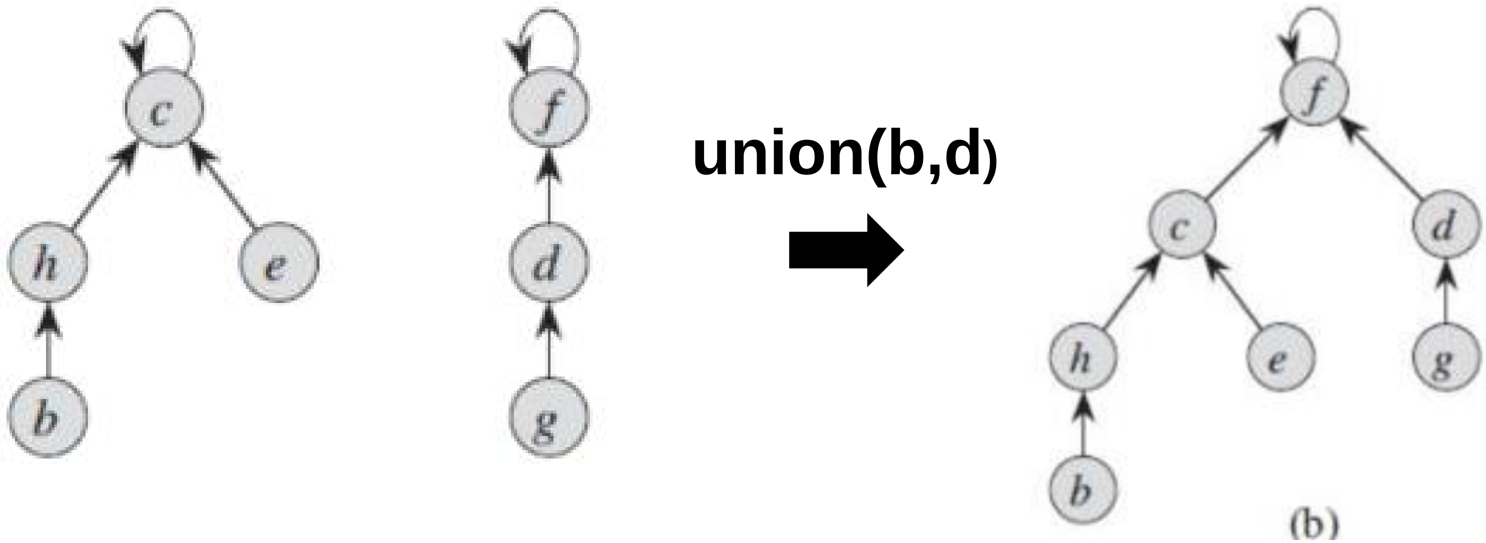
  עם הנציגים **c** ו- **f** בהתאמה



כאמור, אין משמעות לסדר בין האיברים

עצים הפוכים - up trees

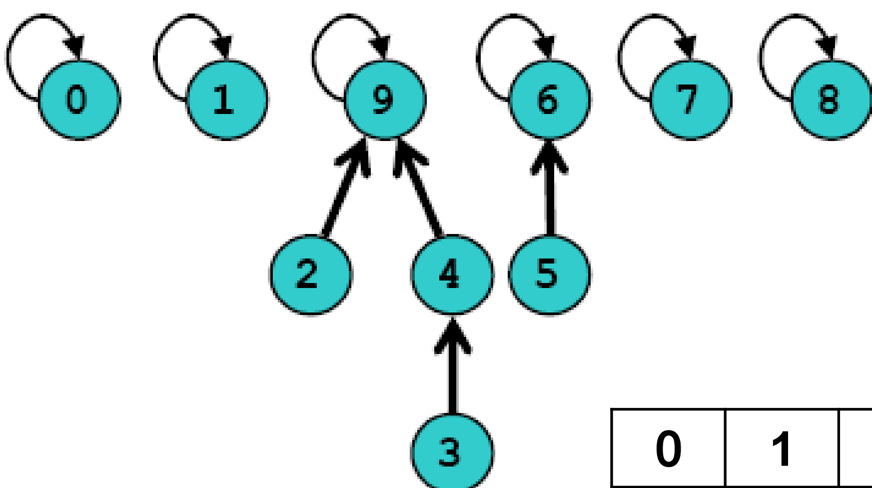
- **makeSet(x)** – יוצרת עץ חדש בגודל **1** עם מזהה **x**
- **find(x)** – הולכת מצומת **x** לשורש לפי מסלול צמתי האב ומחזירה את הנציג בשורש
- **union(x,y)** – מאחדת את הקבוצה המכילה את **x** עם זו המכילה את **y** ע"י חיבור שורשו של אחד כבנו של השני



מימוש עצים הפוכים באמצעות מערך

ניתן לממש עץ הפוך באמצעות "**מערך אבות**"

- כל איבר משויך למספר שלם המתאים לאינדקס במערך
- הערך השמור במערך עבור איבר **i** הוא **האינדקס של אביו**
- ערך המערך של **נציג הקבוצה** הוא האינדקס שלו **עצמו**



מערך אבות:

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

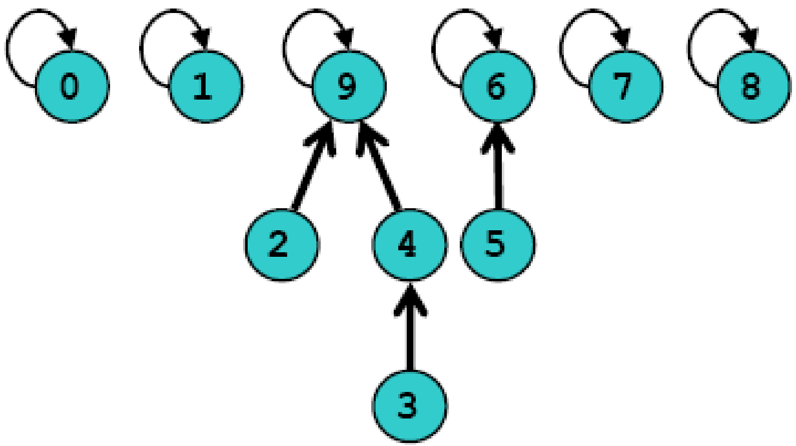
מימוש עצים הפוכים בעזרת מערך

```
void makeSet (int v) //O(1)
        parent[v] = v;
end makeSet

int find (int v) //O(n)
        if (v == parent[v])
                return v
        return find (parent[v])
end find

void union (int a, int b) //O(n)
        a = find(a) //O(n)
        b = find(b) //O(n)
        if (a != b) parent[b] = a
end-union
```

מימוש נאיבי:



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

מימוש עצים הפוכים בעזרת מערך

מימוש נאיבי:

```
void makeSet (int v) //O(1)
        parent[v] = v;
end makeSet

int find (int v) //O(n)
        if (v == parent[v])
                return v
        return find (parent[v])
end find

void union (int a, int b) //O(n)
        a = find(a) //O(n)
        b = find(b) //O(n)
        if (a != b) parent[b] = a
end-union
```

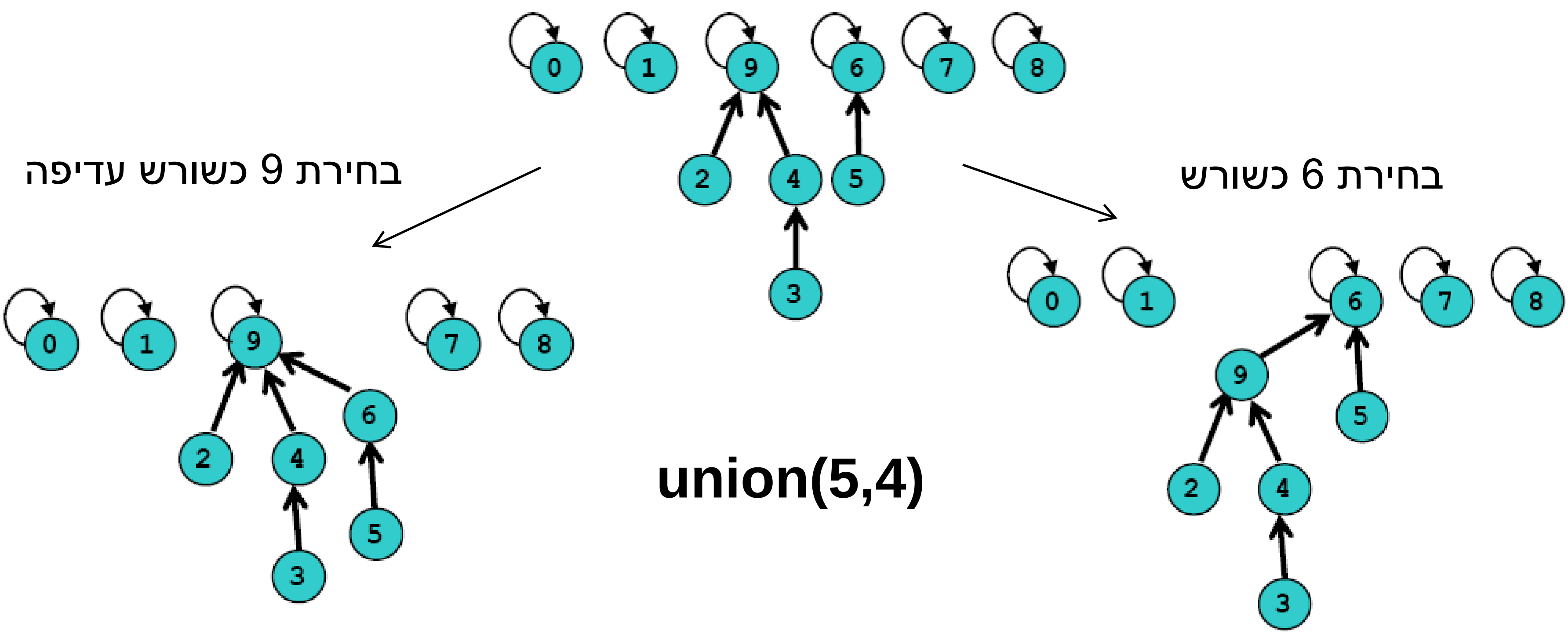
יעילות מימוש זה נמוכה:

- סיבוכיות של כל קריאה ל- **find** היא במקרה הגרוע **O(n)**
- קורה למשל כאשר מחברים רצף איברים יחידים כך שבכל שלב, שורש הקבוצה הנוכחית הופך לבנו של האיבר היחיד

מימוש עצים הפוכים בעזרת מערך

ניתן לשפר משמעותית את הסיבוכיות ע"י שינוי קטן במימוש

- בכל פעם שנבצע פעולת איחוד, נחבר את **שורש העץ הקטן יותר** כבן של **שורש העץ הגדול יותר**
- כאשר **משקל העץ** = כמות הצמתים שהוא מכיל



מימוש עצים הפוכים בעזרת מערך

ניתן לשפר משמעותית את הסיבוכיות ע"י שינוי קטן במימוש:

- בכל פעם שנבצע פעולת איחוד, נחבר את **שורש העץ הקטן יותר** כבן של **שורש העץ הגדול יותר**
- כאשר **משקל העץ** = כמות הצמתים שהוא מכיל
- אינטואיטיבית, זה יגרום לכך שהעצים הנבנים יהיו **מאוזנים** ולא ייראו כמו מסילה ארוכה
- לשם כך נשמור מערך עזר **size** שישמור עבור כל שורש (= נציג של קבוצה) את **משקל** העץ שהוא מייצג

מימוש לפי שיטת Union by Weight

- נגדיר משקל העץ, **size**, כמספר הצמתים שבו
- באיחוד נחבר את **שורש העץ הקטן** כבן של **שורש העץ הגדול**

```
void makeSet (int v) //O(1)
        parent[v] = v
        size[v] = 1
end-makeSet

int find (int v) //O(log₂n)
        if (v == parent[v])
                return v
        return find (parent[v])
end find

void union(int a, int b)
        a = find(a) //O(log₂n)
        b = find(b) //O(log₂n)
        if (a != b)
                if (size[a] ≤ size[b])
                        parent[a] = b
                        size[b] = size[b] + size[a]
                else
                        parent[b] = a
                        size[a] = size[a] + size[b]
                end-if
        end-if
end-union
```

מימוש לפי שיטת Union by Weight

```
void makeSet (int v) //O(1)
        parent[v] = v
        size[v] = 1
end-makeSet

int find (int v) //O(log₂n)
        if (v == parent[v])
                return v
        return find (parent[v])
end find

void union(int a, int b)
        a = find(a) //O(log₂n)
        b = find(b) //O(log₂n)
        if (a != b)
                if (size[a] ≤ size[b])
                        parent[a] = b
                        size[b] = size[b] + size[a]
                else
                        parent[b] = a
                        size[a] = size[a] + size[b]
                end-if
        end-if
end-union
```

זמן ריצה:

- סיבוכיות **find** היא כגובה העץ **h**
- גם סיבוכיות **union** היא כגובה העץ (קוראת פעמיים ל- **find**)
- לכן מספיק להוכיח שגובה כל עץ הוא לוגריתמי במשקלו:

$h = O(\log(size))$

טענה: יהי s **משקל העץ**, אזי הגובה שלו **h** לא עולה על $\log_2 s$

הוכחה: באינדוקציה על משקל העץ s

- **מקרה בסיס**: עבור $s = 1$ הטענה נכונה
- **הנחת האינדוקציה**: הטענה נכונה לכל עץ עם משקל $s > k$
- **צעד האינדוקציה**: עלינו להראות שאם הטענה נכונה עבור 2 עצים, היא תהיה נכונה גם עבור האיחוד שלהם (בשימוש בשיטת האיחוד החדשה)
- נתבונן על שני עצים בעלי s_1 ו- s_2 צמתים שגובהם h_1 ו- h_2 בהתאמה
- נסמן $s = s_1 + s_2$. בלי הגבלת הכלליות נניח $s_1 > s_2$
- לפי הנחת האינדוקציה, $h_1 \le \log s_1$, $h_2 \le \log s_2$
- לאחר איחוד 2 העצים, גובה העץ החדש הוא

$$h = \max(\log s_1, 1 + \log s_2) = \max(\log s_1, \log(2s_2))$$

- נשים לב שמתקיים $2s_2 \le s$ וגם $s_1 \le s$, לכן

$$h \le \max(\log s, \log s) = \log s$$

שיפור סיבוכיות Union/find

ראינו שבאמצעות מימוש **איחוד לפי משקל**, הסיבוכיות של חיפוש ואיחוד משתפרת **מלינארית ללוגריתמית**

- מכיוון שכל עץ נשאר מאוזן לאחר פעולת **union**, פעולת **find** לוקחת רק **O(logn)** זמן
- השיפור התבצע בפעולת **union** ע"י חיבור השורשים באופן מושכל

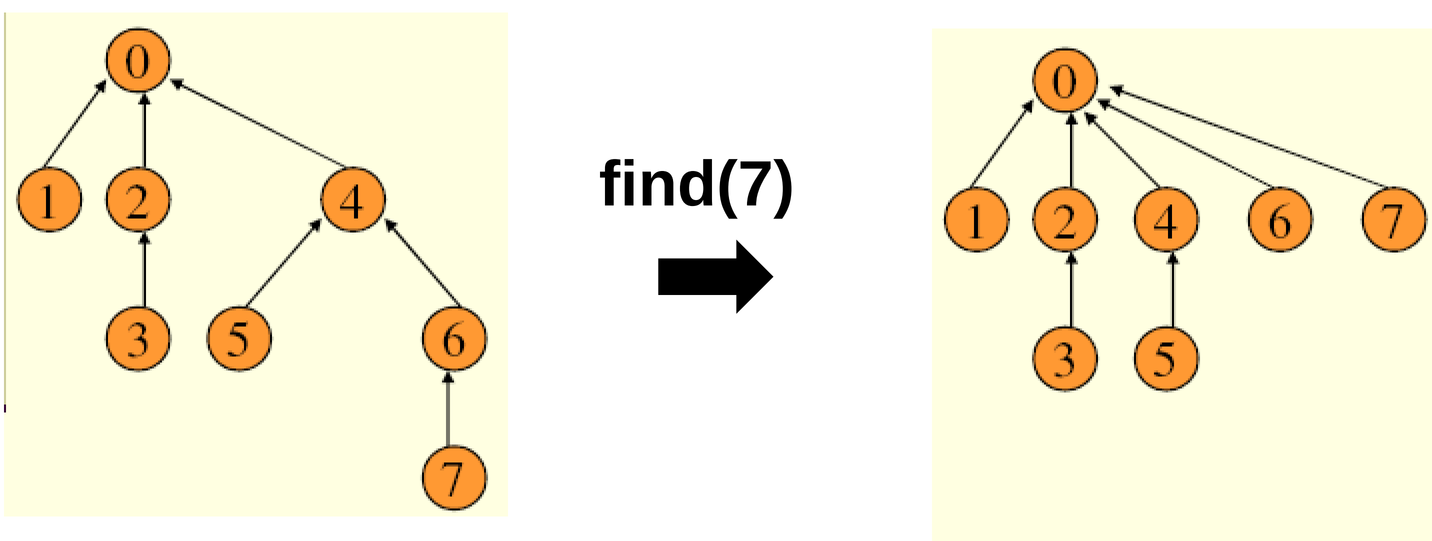
כעת נראה שיפור נוסף - הפעם בפעולת **find**

- כביכול נראה שמימוש זה לא משנה את סדר הגודל הלוגריתמי שכבר קבלנו
- אך ניתוח מתמטי מעמיק יותר מראה שיפור משמעותי בזמן הריצה **המשוערך** והופך אותו ל"**כמעט קבוע**"

דחיסת מסלול – path compression

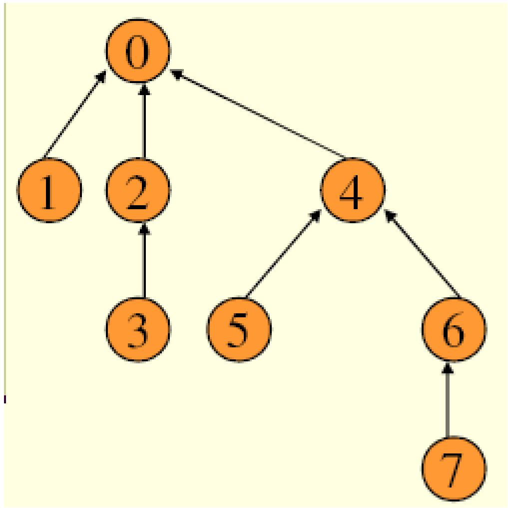
שיטה פשוטה ואפקטיבית להאצת החיפוש המתבצעת ב- **find**

- שיטה זו גורמת לכל צומת **במסלול החיפוש** להפוך לבנו של השורש באותו העץ
- פעולה זו לא משנה את דרגות הצמתים (כלפי מעלה)

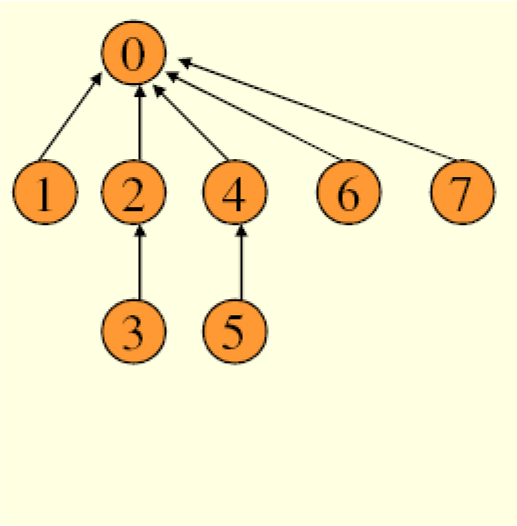


path compression – דחיסת מסלול

```
find(x) // O(α(n))
      if parent[x] != x
         parent[x] = find(parent[x])
      return parent[x]
end-find
```



find(7)



דחיסת מסלול – Path compression

- **האם הועלנו במשהו**? לכאורה הסבוכיות נשארה כפי שהיתה, **לוגריתמית** (בהנחה שהאיחוד מתבצע עדיין לפי שיטת המשקל)
- למעשה, בניתוח סיבוכיות **המקרה הגרוע ביותר** של פעולת **find בודדת**, הקבוע הכופל את הלוג אפילו גדל קצת
- אך ניתוח מעמיק יותר של **הזמן המשוערך לכל פעולה** מראה שהוא קטן מאד
- **זמן משוערך לכל פעולה** (Amortized time)

= זמן ריצה ממוצע על פני מספר רב של פעולות ברצף

- **הרעיון**: על אף שחלק קטן מהפעולות יקח זמן רב, לאורך זמן, רוב הפעולות יתבצעו במהירות
- כך שהזמן **הממוצע לכל פעולת find** הוא **"כמעט קבוע"**

יעילות union/find

טענה: עם שימוש ב- **union-by-weight** ו- **path-compression**, סדרה של **n < m** פעולות (**find/union**) לוקחת **O(m·log*n)** זמן

- **n** = מספר איברים בסה"כ
- **m** = מספר הפעולות שמבוצעות (**union** או **find**)
- **log*n** = פונקצית ה-**"לוגריתם החוזר"** (**iterated logarithm**)

= מספר הפעמים שיש להפעיל **log** על **n** על מנת לקבל ערך 1 =>

- **log*n** עולה **לאט מאד** – היא קטנה מ- **5** לכל $n < 2^{65536}$
- בחישובים מעשיים ניתן להתייחס ל- **log*n** כאל **קבוע** – הערך **5** מתקבל ע"י **n** הגדול בהרבה ממה שמחשבים בימינו מסוגלים לחשב...
- **מסקנה**: זמן משוערך לפעולה במבנה **union/find** הוא כמעט **O(1)**!

הוכחה:

- אולי בשנה הבאה...